

**LECTURE NOTES ON DATA
STRUCTURES THROUGH C**

Chapter 1

Basic Concepts

The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

- **Introduction to Data Structures:**

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

$$\text{Algorithm} + \text{Data structure} = \text{Program}$$

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.1 shows the classification of data structures.

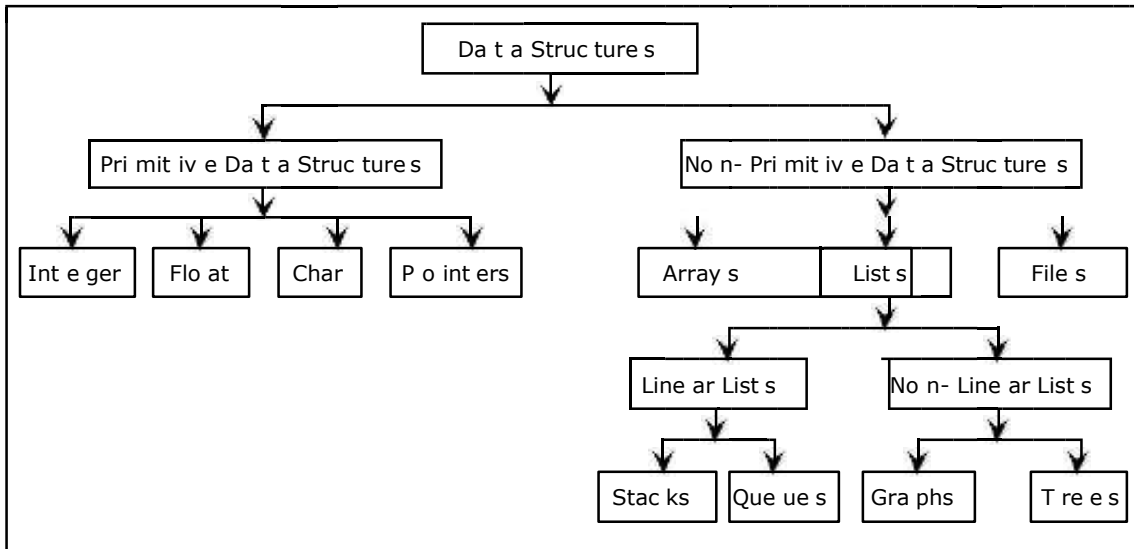


Figure 1. 1. Classification of Data Structures

1.2. Data structures: Organization of data

The collection of data you work with in a program have some kind of structure or organization. No matter how complex your data structures are they can be broken down into two fundamental types:

- Contiguous
- Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure are scattered in memory, but are linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1.2 below illustrates the difference between contiguous and non-contiguous structures.

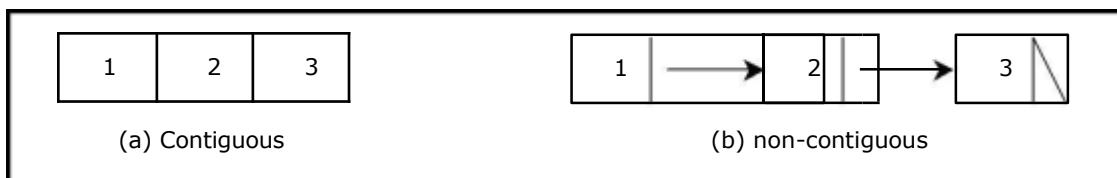


Figure 1.2 Contiguous and Non-contiguous structures compared

Contiguous structures:

Contiguous structures can be broken down further into two kinds: those that contain data items of all the same size, and those where the size may differ. Figure 1.2 shows an example of each kind. The first kind is called the array. Figure 1.3(a) shows an example of an array of numbers. In an array, each element is of the same type, and thus has the same size.

The second kind of contiguous structure is called structure, figure 1.3(b) shows a simple structure consisting of a person's name and age. In a struct, elements may be of different data types and thus may have different sizes.

For example, a person's age can be represented with a simple integer that occupies two bytes of memory. But his or her name, represented as a string of characters, may require many bytes and may even be of varying length.

Couples with the atomic types (that is, the single data-item built-in types such as integer, float and pointers), arrays and structs provide all the `—mortar` you need to built more exotic form of data structure, including the non-contiguous forms.

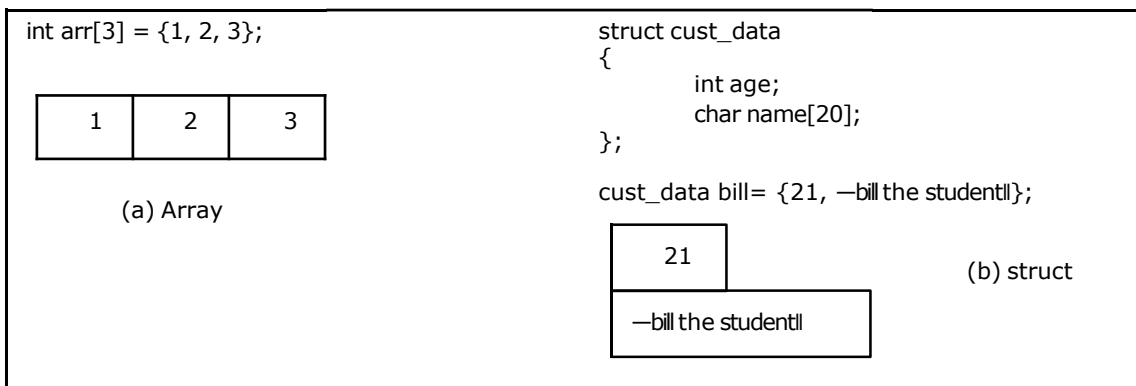


Figure 1.3 Examples of contiguous structures.

Non-contiguous structures:

Non-contiguous structures are implemented as a collection of data-items, called nodes, where each node can point to one or more other nodes in the collection. The simplest kind of non-contiguous structure is linked list.

A linked list represents a linear, one-dimension type of non-contiguous structure, where there is only the notation of backwards and forwards. A tree such as shown in figure 1.4(b) is an example of a two-dimensional non-contiguous structure. Here, there is the notion of up and down and left and right.

In a tree each node has only one link that leads into the node and links can only go down the tree. The most general type of non-contiguous structure, called a graph has no such restrictions. Figure 1.4(c) is an example of a graph.

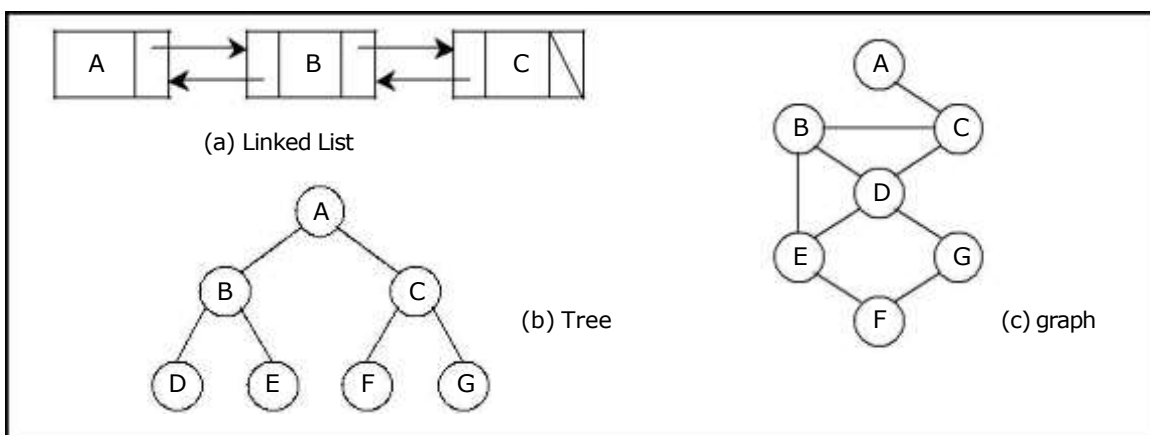


Figure 1.4. Examples of non-contiguous structures

Chapter 2

Searching and Sorting

There are basically two aspects of computer programming. One is data organization also commonly called as data structures. Till now we have seen about data structures and the techniques and algorithms used to access them. The other part of computer programming involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This chapter introduces this important aspect of problem solving.

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

- Linear or sequential search
- Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

- Bubble sort
- Quick sort
- Selection sort and
- Heap sort

There are two types of sorting techniques:

3. Internal sorting
4. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

- **Linear Search:**

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are n elements organized sequentially on a List. The number of

comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need $[(n+1)/2]$ comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is **$O(n)$** .

Algorithm:

Let array $a[n]$ stores n elements. Determine whether element x is present or not.

```
linsrch(a[n], x)
{
    index = 0;
    flag = 0;
    while (index < n) do
    {
        if (x == a[index])
        {
            flag = 1;
            break;
        }
        index ++;
    }
    if(flag == 1)
        printf("Data found at %d position", index);
    else
        printf("data not found");
}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for:	45, we'll look at 1 element before success
	39, we'll look at 2 elements before success
	8, we'll look at 3 elements before success
	54, we'll look at 4 elements before success
	77, we'll look at 5 elements before success
	38 we'll look at 6 elements before success
	24, we'll look at 7 elements before success
	16, we'll look at 8 elements before success
	4, we'll look at 9 elements before success
	7, we'll look at 10 elements before success
	9, we'll look at 11 elements before success
	20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	0	1	2	3	4	5	6	7	8
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

3. Searching for $x = 7$ Search successful, data found at 3rd position.
4. Searching for $x = 82$ Search successful, data found at 7th position.
5. Searching for $x = 42$ Search un-successful, data not found.

1.4. A non-recursive program for Linear Search:

```
\{include <stdio.h>
\{include <conio.h>

main()
{
    int number[25], n, data, i, flag = 0;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements:
"); for(i = 0; i < n; i++)
        scanf("%d", &number[i]);
    printf("\n Enter the element to be Searched: ");
    scanf("%d", &data);
    for( i = 0; i < n; i++)
    {
        if(number[i] == data)
        {
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        printf("\n Data found at location: %d", i+1);
    else
        printf("\n Data not found ");
}
```

1.5. A Recursive program for linear search:

```
6. include <stdio.h>
7. include <conio.h>

void linear_search(int a[], int data, int position, int n)
{
    if(position < n)
```

```

    {
        if(a[position] == data)
            printf("\n Data Found at %d ", position);
        else
            linear_search(a, data, position + 1, n);
    }
else
    printf("\n Data not found");
}

void main()
{
    int a[25], i, n, data;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements:
"); for(i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("\n Enter the element to be seached: ");
    scanf("%d", &data);
    linear_search(a, data, 0, n);
    getch();
}

```

1. BINARY SEARCH

If we have n records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element x , binary search is used to find the corresponding element from the list. In case x is present, we have to determine a value j such that $a[j] = x$ (successful search). If x is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key $a[mid]$, and compare x with $a[mid]$. If $x = a[mid]$ then the desired record has been found. If $x < a[mid]$ then x must be in that portion of the file that precedes $a[mid]$. Similarly, if $a[mid] > x$, then further search is only necessary in that part of the file which follows $a[mid]$.

If we use recursive procedure of finding the middle key $a[mid]$ of the un-searched portion of a file, then every un-successful comparison of x with $a[mid]$ will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between x and $a[mid]$, and since an array of length n can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array $a[n]$ of elements in increasing order, $n \geq 0$, determine whether x is present, and if so, set j such that $x = a[j]$ else return 0.


```

binsrch(a[], n, x)
{
    low = 1; high = n;
    while (low ≤ high) do
    {
        mid = (low + high)/2 if
        (x < a[mid])
            high = mid - 1;
        else if (x > a[mid])
            low = mid +
            1; else return mid;
    }
    return 0;
}

```

low and *high* are integer variables such that each time through the loop either *x* is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if *x* is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for $x = 4$: (This needs 3 comparisons)
 $low = 1, high = 12, mid = 13/2 = 6$, check 20
 $low = 1, high = 5, mid = 6/2 = 3$, check 8
 $low = 1, high = 2, mid = 3/2 = 1$, check 4, **found**

If we are searching for $x = 7$: (This needs 4 comparisons)
 $low = 1, high = 12, mid = 13/2 = 6$, check 20
 $low = 1, high = 5, mid = 6/2 = 3$, check 8
 $low = 1, high = 2, mid = 3/2 = 1$, check 4
 $low = 2, high = 2, mid = 4/2 = 2$, check 7, **found**

If we are searching for $x = 8$: (This needs 2 comparisons)
 $low = 1, high = 12, mid = 13/2 = 6$, check 20
 $low = 1, high = 5, mid = 6/2 = 3$, check 8, **found**

If we are searching for $x = 9$: (This needs 3 comparisons)
 $low = 1, high = 12, mid = 13/2 = 6$, check 20
 $low = 1, high = 5, mid = 6/2 = 3$, check 8
 $low = 4, high = 5, mid = 9/2 = 4$, check 9, **found**

If we are searching for $x = 16$: (This needs 4 comparisons)
 $low = 1, high = 12, mid = 13/2 = 6$, check 20
 $low = 1, high = 5, mid = 6/2 = 3$, check 8
 $low = 4, high = 5, mid = 9/2 = 4$, check 9
 $low = 5, high = 5, mid = 10/2 = 5$, check 16, **found**

If we are searching for $x = 20$: (This needs 1 comparison)
 $low = 1, high = 12, mid = 13/2 = 6$, check 20, **found**

If we are searching for $x = 24$: (This needs 3 comparisons)
 low = 1, high = 12, mid = $13/2 = 6$, check 20
 low = 7, high = 12, mid = $19/2 = 9$, check 39
 low = 7, high = 8, mid = $15/2 = 7$, check 24, **found**

If we are searching for $x = 38$: (This needs 4 comparisons)
 low = 1, high = 12, mid = $13/2 = 6$, check 20
 low = 7, high = 12, mid = $19/2 = 9$, check 39
 low = 7, high = 8, mid = $15/2 = 7$, check 24
 low = 8, high = 8, mid = $16/2 = 8$, check 38, **found**

If we are searching for $x = 39$: (This needs 2 comparisons)
 low = 1, high = 12, mid = $13/2 = 6$, check 20
 low = 7, high = 12, mid = $19/2 = 9$, check 39, **found**

If we are searching for $x = 45$: (This needs 4 comparisons)
 low = 1, high = 12, mid = $13/2 = 6$, check 20
 low = 7, high = 12, mid = $19/2 = 9$, check 39
 low = 10, high = 12, mid = $22/2 = 11$, check 54
 low = 10, high = 10, mid = $20/2 = 10$, check 45, **found**

If we are searching for $x = 54$: (This needs 3 comparisons)
 low = 1, high = 12, mid = $13/2 = 6$, check 20
 low = 7, high = 12, mid = $19/2 = 9$, check 39
 low = 10, high = 12, mid = $22/2 = 11$, check 54, **found**

If we are searching for $x = 77$: (This needs 4 comparisons)
 low = 1, high = 12, mid = $13/2 = 6$, check 20
 low = 7, high = 12, mid = $19/2 = 9$, check 39
 low = 10, high = 12, mid = $22/2 = 11$, check 54
 low = 12, high = 12, mid = $24/2 = 12$, check 77, **found**

The number of comparisons necessary by search element:

- 20 – requires 1 comparison;
- 8 and 39 – requires 2 comparisons;
- 4, 9, 24, 54 – requires 3 comparisons and
- 7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding $37/12$ or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

<i>Index</i>	0	1	2	3	4	5	6	7	8
<i>Elements</i>	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for $x = 101$: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
found		

2. Searching for $x = 82$: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8
found		

3. Searching for $x = 42$: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6
6 not found		

4. Searching for $x = -14$: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1 not found	

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

<i>Index</i>	1	2	3	4	5	6	7	8	9
<i>Elements</i>	-15	-6	0	7	9	23	54	82	101
<i>Comparisons</i>	3	2	3	4	1	3	2	3	4

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding $25/9$ or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x .

If $x < a(1)$, $a(1) < x < a(2)$, $a(2) < x < a(3)$, $a(5) < x < a(6)$, $a(6) < x < a(7)$ or $a(7) < x < a(8)$ the algorithm requires 3 element comparisons to determine that x is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

$$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$$

Time Complexity:

The time complexity of binary search in a successful search is $O(\log n)$ and for an unsuccessful search is $O(\log n)$.

2.2.1. A non-recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

main()
{
    int number[25], n, data, i, flag = 0, low, high, mid;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements in ascending order: ");
    for(i = 0; i < n; i++)
        scanf("%d", &number[i]);
    printf("\n Enter the element to be searched: ");
    scanf("%d", &data);
    low = 0; high = n-1;
    while(low <= high)
    {
        mid = (low + high)/2;
        if(number[mid] == data)
        {
            flag = 1;
            break;
        }
        else
        {
            if(data < number[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
    }
    if(flag == 1)
        printf("\n Data found at location: %d", mid + 1);
    else
        printf("\n Data Not Found ");
}
}
```

• A recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

void bin_search(int a[], int data, int low, int high)
{
    int mid ;
    if( low <= high)
    {
        mid = (low + high)/2;
        if(a[mid] == data)
            printf("\n Element found at location: %d ", mid + 1);
        else
        {
            if(data < a[mid])
                bin_search(a, data, low, mid-1);
            else

```

```

        bin_search(a, data, mid+1, high);
    }
}
else
    printf("\n Element not found");
}
void main()
{
    int a[25], i, n, data;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements in ascending order: ");
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("\n Enter the element to be searched: ");
    scanf("%d", &data);
    bin_search(a, data, 0, n-1);
    getch();
}

```

Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., $X[i]$ with $X[i+1]$ and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array $x[n]$ which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare $X[i]$ and $X[i+1]$ for $i = 0, 1, 2, 3,$ and $4,$ and interchange $X[i]$ and $X[i+1]$ if $X[i] > X[i+1]$. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22 33		

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
    int i, j, temp;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n-i-1 ; j++)
        {
            if (x[j] > x[j+1])
            {
                temp = x[j];
                x[j] = x[j+1];
                x[j+1] = temp;
            }
        }
    }
}

main()
{
    int i, n, x[25];
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter Data:");
    for(i = 0; i < n ; i++)
        scanf("%d", &x[i]);
    bubblesort(x, n);
    printf ("\n Array Elements after sorting: ");
    for (i = 0; i < n; i++)
        printf ("%5d", x[i]);
}
```

Time Complexity:

The bubble sort method of sorting an array of size n requires $(n-1)$ passes and $(n-1)$ comparisons on each pass. Thus the total number of comparisons is $(n-1) * (n-1) = n^2 - 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

Selection Sort:

Selection sort will not require no more than $n-1$ interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through $(n-1)$ times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array $x[0], x[1], \dots, x[n-1]$, and then interchange $x[j]$ with $x[0]$. Then $x[0]$ is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array $x[1], x[2], \dots, x[n-1]$, and then interchange $x[1]$ with $x[j]$. Then $x[0], x[1]$ are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array $x[2], x[3], \dots, x[n-1]$, and then interchange $x[2]$ with $x[j]$. Then $x[0], x[1], x[2]$ are sorted.

Pass (n-1): Find the location j of the smaller of the elements $x[n-2]$ and $x[n-1]$, and then interchange $x[j]$ and $x[n-2]$. Then $x[0], x[1], \dots, x[n-2]$ are sorted. Of course, during this pass $x[n-1]$ will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also $O(n^2)$ for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	2	3	4	5	6	7	8	9	Remarks
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap $a[i]$ & $a[j]$
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap $a[i]$ and $a[j]$
45	50	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap $a[i]$ and $a[j]$
45	50	55	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap $a[i]$ and $a[j]$
45	50	55	60	70	80	75	85	65	Find the fifth smallest element
				i				j	swap $a[i]$ and $a[j]$
45	50	55	60	65	80	75	85	70	Find the sixth smallest element
					i			j	swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	85	80	Find the seventh smallest element
						i j			swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	85	80	Find the eighth smallest element
							i	J	swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	80	85	The outer loop ends.

Non-recursive Program for selection sort:

```
# include<stdio.h>
# include<conio.h>

void selectionSort( int low, int high );

int a[25];

int main()
{
    int num, i= 0;
    clrscr();
    printf( "Enter the number of elements: " );
    scanf("%d", &num);
    printf( "\nEnter the elements:\n" );
    for(i=0; i < num; i++)
        scanf( "%d", &a[i] );
    selectionSort( 0, num - 1 );
    printf( "\nThe elements after sorting are: " );
    for( i=0; i< num; i++ )
        printf( "%d ", a[i] );
    return 0;
}

void selectionSort( int low, int high )
{
    int i=0, j=0, temp=0, minindex;
    for( i=low; i <= high; i++ )
    {
        minindex = i;
        for( j=i+1; j <= high; j++ )
        {
            if( a[j] < a[minindex] )
                minindex = j;
        }
        temp = a[i];
        a[i] = a[minindex];
        a[minindex] = temp;
    }
}
```

Recursive Program for selection sort:

```
#include <stdio.h>
#include<conio.h>

int x[6] = {77, 33, 44, 11, 66};
selectionSort(int);

main()
{
    int i, n = 0;
    clrscr();
    printf ( " Array Elements before sorting: ");
    for (i=0; i<5; i++)
```

```

        printf ("%d ", x[i]);
    selectionSort(n);          /* call selection sort */
    printf ("\n Array Elements after sorting: ");
    for (i=0; i<5; i++)
        printf ("%d ", x[i]);
}

selectionSort( int n)
{
    int k, p, temp, min;
    if (n== 4)
        return (-
    1); min = x[n];
    p = n;
    for (k = n+1; k<5; k++)
    {
        if (x[k] <min)
        {
            min = x[k];
            p = k;
        }
    }
    temp = x[n]; /* interchange x[n] and x[p] */ x[n] =
    x[p];
    x[p] = temp;
    n++ ;
    selectionSort(n);
}

```

Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by —divide and conquerll technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer `__up`` until `a[up] >= pivot`.
2. Repeatedly decrease the pointer `__down`` until `a[down] <= pivot`.
3. If `down > up`, interchange `a[down]` with `a[up]`
4. Repeat the steps 1, 2 and 3 till the `__up`` pointer crosses the `__down`` pointer. If `__up`` pointer crosses `__down`` pointer, the position for pivot is found and place pivot element in `__down`` pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array `a` between positions `low` and `high`.

1. It terminates when the condition `low >= high` is satisfied. This condition will be satisfied only when the array is completely sorted.
2. Here we choose the first element as the `pivot`. So, `pivot = x[low]`. Now it calls the partition function to find the proper position `j` of the element `x[low]` i.e. `pivot`. Then we will have two sub-arrays `x[low], x[low+1], x[j-1]` and `x[j+1], x[j+2], . . . x[high]`.
3. It calls itself recursively to sort the left sub-array `x[low], x[low+1], x[j-1]` between positions `low` and `j-1` (where `j` is returned by the partition function).
4. It calls itself recursively to sort the right sub-array `x[j+1], x[j+2], . . x[high]` between positions `j+1` and `high`.

The time complexity of quick sort algorithm is of **$O(n \log n)$** .

Algorithm

Sorts the elements `a[p], a[q]` which reside in the global array `a[n]` into ascending order. The `a[n + 1]` is considered to be defined and must be greater than all elements in `a[n]`; `a[n + 1] = + ∞`

quicksort (p, q)

```
{
    if ( p < q ) then
    {
        call j = PARTITION(a, p, q+1); // j is the position of the partitioning element
        call quicksort(p, j - 1);
        call quicksort(j + 1 , q);
    }
}
```

partition(a, m, p)

```
{
    v = a[m]; up = m; down = p;           // a[m] is the partition element
    do
    {
        repeat
            up = up + 1;
        until (a[up] ≥ v);

        repeat
            down = down -
            1; until (a[down] ≤ v);
        if (up < down) then call interchange(a, up,
        down); } while (up ≥ down);

    a[m] = a[down];
    a[down] = v;
    return (down);
}
```

```

interchange(a, up, down)
{
    p = a[up];
    a[up] = a[down];
    a[down] = p;
}

```

Example:

Select first element as the pivot element. Move `_up` pointer from left to right in search of an element larger than pivot. Move the `_down` pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the `_up` pointer crosses the `_down` pointer. If `_up` pointer crosses `_down` pointer, the position for pivot is found and interchange pivot and element at `_down` position.

Let us consider the following example with 13 elements to analyze quick sort:

1	2	3	4	5	6	7	8	9	10	11	12	13	Remarks
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	38	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	24								
pivot	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	08	(16)									swap pivot & down
	pivot	down	up										
	(04)	06											swap pivot & down
	04	pivot,											
	down,												
	up												
				16									
				pivot,									
				down									
				,									
				up									
(02	04	06	08	16	24)	38							

							(56	57	58	79	70	45)	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	56	(58	79	70	57)	
							45						swap pivot & down
							pivot,						
							'						
							up						
									(58	79	70	57)	swap up & down
									pivot	up		down	
										57		79	
										down	up		
									(57)	58	(70	79)	swap pivot & down
									57				
									pivot,				
									down				
									'				
									up				
										(70	79)		
										pivot,	up		swap pivot & down
										down			
										70			
												79	
										pivot,	down,	up	
							(45	56	57	58	70	79)	
02	04	06	08	16	24	38	45	56	57	58	70	79	

Recursive program for Quick Sort:

```
# include<stdio.h>
# include<conio.h>
```

```
void quicksort(int, int);
int partition(int, int); void
interchange(int, int); int
array[25];
```

```
int main()
{
    int num, i = 0;
    clrscr();
    printf( "Enter the number of elements: " );
    scanf( "%d", &num);
    printf( "Enter the elements: " );
    for(i=0; i < num; i++)
        scanf( "%d", &array[i] );
    quicksort(0, num -1);
    printf( "\nThe elements after sorting are: " );
}
```

```

        for(i=0; i < num; i++)
            printf("%d ", array[i]);
        return 0;
    }

void quicksort(int low, int high)
{
    int pivotpos;
    if( low < high )
    {
        pivotpos = partition(low, high + 1);
        quicksort(low, pivotpos - 1);
        quicksort(pivotpos + 1, high);
    }
}

int partition(int low, int high)
{
    int pivot = array[low];
    int up = low, down = high;

    do
    {
        do
            up = up + 1;
        while(array[up] < pivot );

        do
            down = down - 1;
        while(array[down] > pivot);

        if(up < down) interchange(up,
            down);

    } while(up < down);
    array[low] = array[down];
    array[down] = pivot;
    return down;
}

void interchange(int i, int j)
{
    int temp;
    temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

```

Priority Queue, Heap and Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

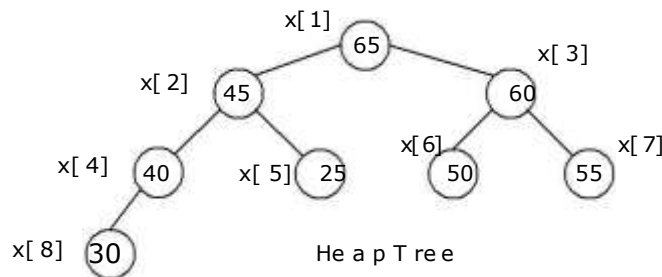
Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location i can be found in location $2*i$.
- The right child of an element stored at location i can be found in location $2*i+1$.
- The parent of an element stored at location i can be found at location $\text{floor}(i/2)$.

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

Max_heap_insert (a, n)

```
{
    //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
    int i, n;
    i = n;
    item = a[n];
    while ( ( i > 1 ) and ( a[ i/2 ] < item ) do
    {
        a[i] = a[ i/2 ] ;           // move the parent down
        i = i/2 ;
    }
    a[i] = item ;
    return true ;
}
```

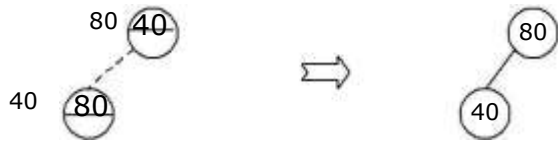
Example:

Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

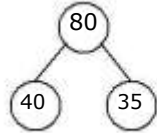
1. Insert 40:



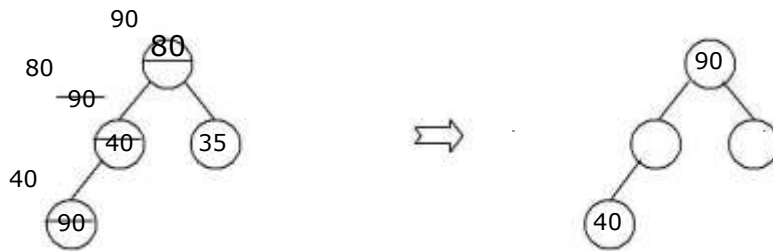
2. Insert 80:



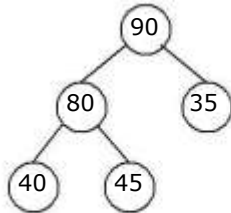
3. Insert 35:



4. Insert 90:



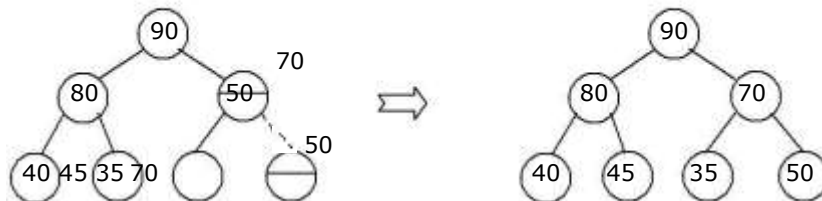
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
 - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
 - Make X as the current node.
 - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

delmax (a, n, x)

// delete the maximum from the heap a[n] and store it in x

```
{
    if (n = 0) then
    {
        write (—heap is empty!);
        return false;
    }
    x = a[1]; a[1] = a[n];
    adjust (a, 1, n-1);
    return true;
}
```

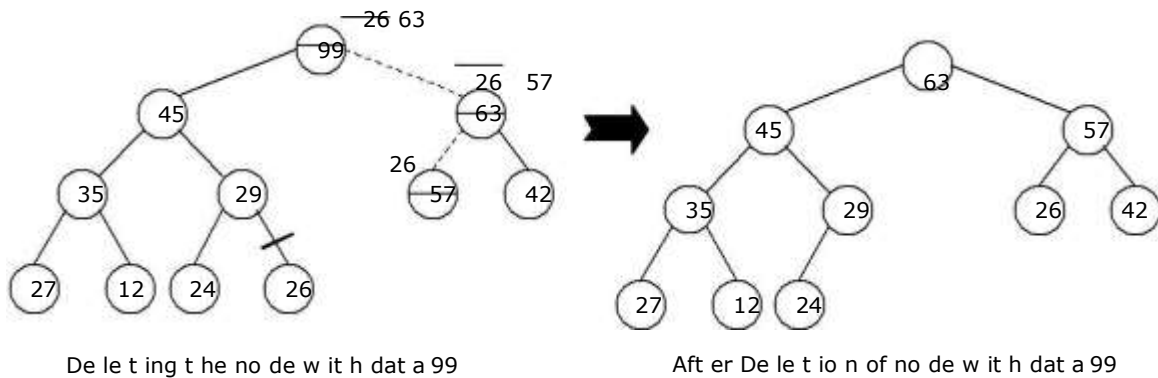
adjust (a, i, n)

// The complete binary trees with roots $a(2*i)$ and $a(2*i + 1)$ are combined with $a(i)$ to form a single heap, $1 \leq i \leq n$. No node has an address greater than n or less than 1. //

```
{
    j = 2 * i ;
    item = a[i] ;
    while (j ≤ n) do
    {
        if ((j < n) and (a (j) < a (j + 1))) then j = j + 1;
        // compare left and right child and let j be the larger
        child if (item ≥ a (j)) then break;
        // a position for item is found else
        a [ j / 2 ] = a[j] // move the larger child up a level j = 2 * j;
    }
    a [ j / 2 ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now,

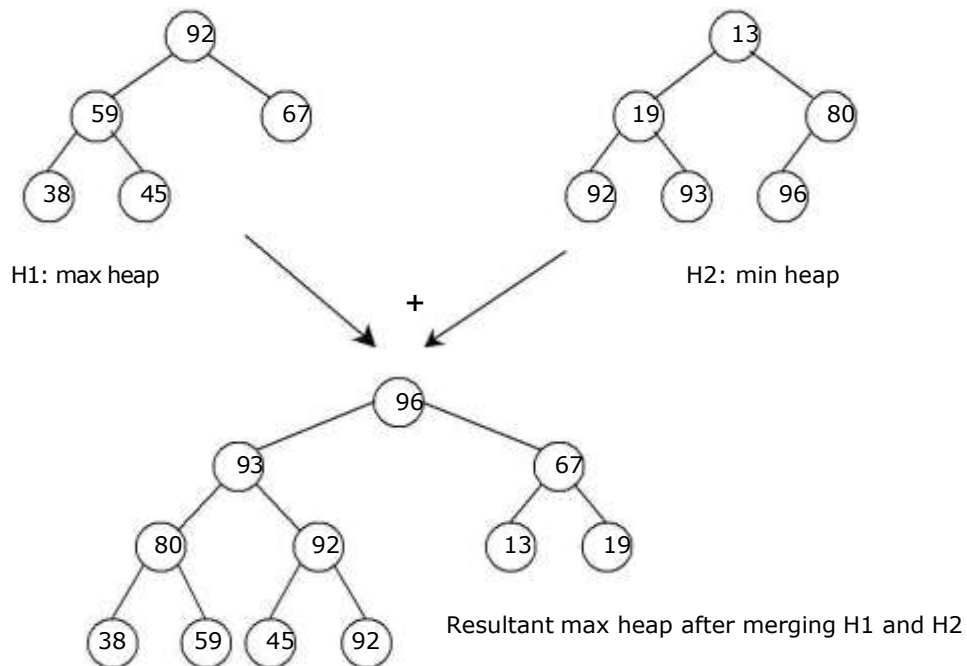
26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leaf node, hence re-heap is completed.



Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.



Application of heap tree:

They are two main applications of heap trees known are:

1. Sorting (Heap sort) and
2. Priority queue implementation.

HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2.
 - a. Remove the top most item (the largest) and replace it with the last element in the heap.
 - b. Re-heapify the complete binary tree.
 - c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements $a[n]$. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
    heapify(a, n);
    for i = n to 2 by - 1 do
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        adjust (a, 1, i - 1);
    }
}
```

heapify (a, n)

```
//Readjust the elements in a[n] to form a heap.
{
    for i = n/2 to 1 by - 1 do adjust (a, i, n);
}
```

adjust (a, i, n)

```
// The complete binary trees with roots  $a(2*i)$  and  $a(2*i + 1)$  are combined with  $a(i)$  to form a single heap,  $1 \leq i \leq n$ . No node has an address greater than  $n$  or less than 1. //
```

```
{
    j = 2 * i ;
    item = a[i] ;
    while (j ≤ n) do
    {
        if ((j < n) and (a (j) < a (j + 1))) then j = j + 1;
        // compare left and right child and let j be the larger
        child if (item ≥ a (j)) then break;
        // a position for item is found else
        a [ j / 2 ] = a[j] // move the larger child up a level j = 2 * j;
    }
    a [ j / 2 ] = item;
}
```

Time Complexity:

Each n insertion operations takes $O(\log k)$, where k is the number of elements in the heap at the time. Likewise, each of the n remove operations also runs in time $O(\log k)$, where k is the number of elements in the heap at the time.

Since we always have $k \leq n$, each such operation runs in $O(\log n)$ time in the worst case.

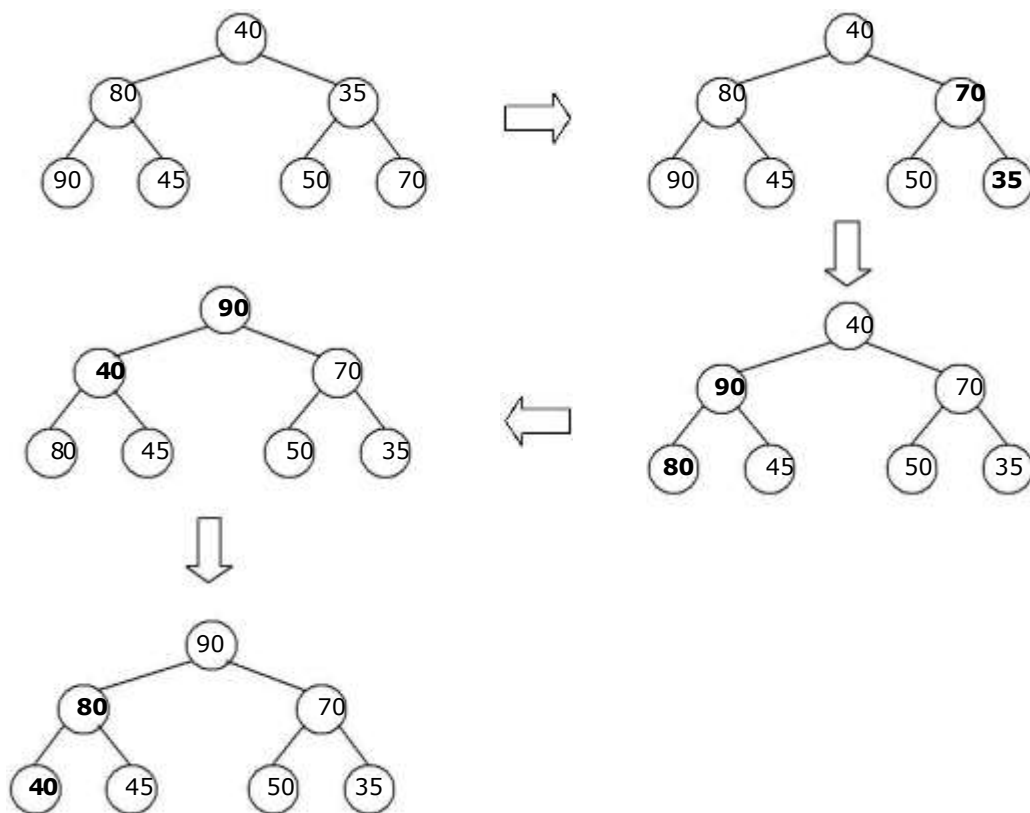
Thus, for n elements it takes $O(n \log n)$ time, so the priority queue sorting algorithm runs in $O(n \log n)$ time when we use a heap to implement the priority queue.

Example 1:

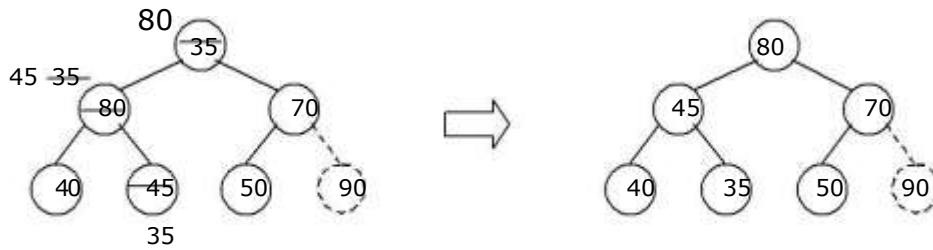
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

Solution:

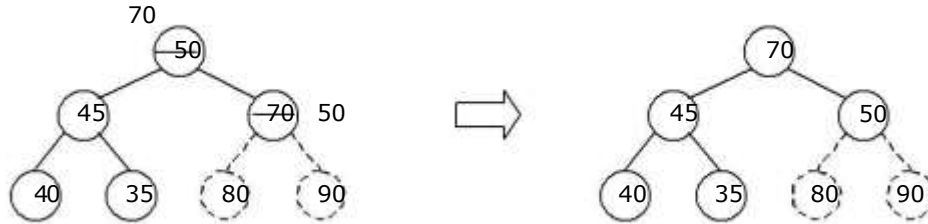
First form a heap tree from the given set of data and then sort by repeated deletion operation:



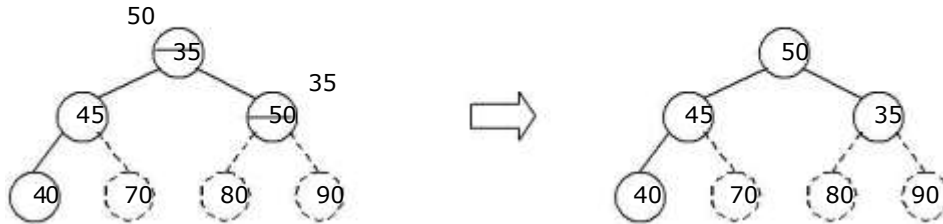
1. Exchange root 90 with the last element 35 of the array and re-heapify



2. Exchange root 80 with the last element 50 of the array and re-heapify



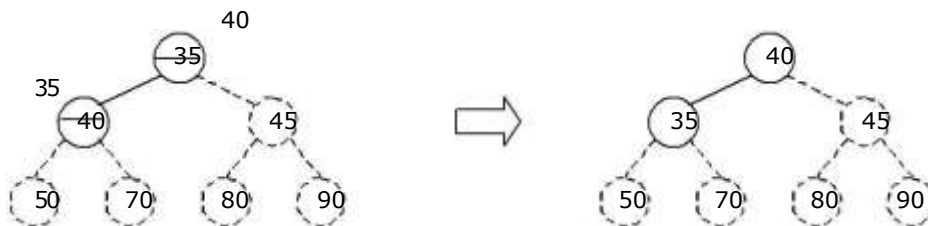
3. Exchange root 70 with the last element 35 of the array and re-heapify



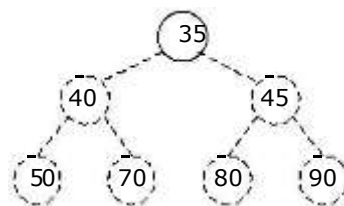
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

Program for Heap Sort:

```
void adjust(int i, int n, int a[])
{
    int j, item; j
    = 2 * i;
    item = a[i];
    while(j <= n)
    {
        if((j < n) && (a[j] < a[j+1]))
            j++;
        if(item >= a[j])
            break;
        else
        {
            a[j/2] = a[j];
            j = 2*j;
        }
    }
    a[j/2] = item;
}

void heapify(int n, int a[])
{
    int i;
    for(i = n/2; i > 0; i--)
        adjust(i, n, a);
}

void heapsort(int n,int a[])
{
    int temp, i;
    heapify(n, a);
    for(i = n; i > 0; i--)
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        adjust(1, i - 1, a);
    }
}

void main()
{
    int i, n, a[20];
    clrscr();
    printf("\n How many element you want: ");
    scanf("%d", &n);
    printf("Enter %d elements: ",
    n); for (i=1; i<=n; i++)
        scanf("%d", &a[i]);
    heapsort(n, a);
    printf("\n The sorted elements are: \n");
    for (i=1; i<=n; i++)
        printf("%5d",
        a[i]); getch();
}
```

Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇	P ₈	P ₉	P ₁₀
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Exercises

1. Write a recursive function to implement binary search and compute its time complexity.
2. Find the expected number of passes, comparisons and exchanges for bubble sort when the number of elements is equal to 10. Compare these results with the actual number of operations when the given sequence is as follows: 7, 1, 3, 4, 10, 9, 8, 6, 5, 2.
3. An array contains n elements of numbers. The several elements of this array may contain the same number x . Write an algorithm to find the total number of elements which are equal to x and also indicate the position of the first such element in the array.
4. Write a function to sort a matrix row-wise and column-wise. Assume that the matrix is represented by a two dimensional array.
5. A very large array of elements is to be sorted. The program is to be run on a personal computer with limited memory. Which sort would be a better choice: Heap sort or Quick sort? Why?
6. Here is an array of ten integers: 5 3 8 9 1 7 0 2 6 4
Suppose we partition this array using quicksort's partition function and using 5 for the pivot. Draw the resulting array after the partition finishes.
7. Here is an array which has just been partitioned by the first step of quicksort: 3, 0, 2, 4, 5, 8, 7, 6, 9. Which of these elements could be the pivot? (There may be more than one possibility!)
8. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.
9. Sort the sequence 3, 1, 4, 5, 9, 2, 6, 5 using insertion sort.

10. Show how heap sort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102.
11. Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quick sort with median-of-three partitioning and a cutoff of 3.

Multiple Choice Questions

1. What is the worst-case time for serial search finding a single item in an array? [D]
A. Constant time
B. Quadratic time
C. Logarithmic time
D. Linear time
2. What is the worst-case time for binary search finding a single item in an array? [B]
A. Constant time
B. Quadratic time
C. Logarithmic time
D. Linear time
3. What additional requirement is placed on an array, so that binary search may be used to locate an entry? [C]
A. The array elements must form a heap.
B. The array must have at least 2 entries
C. The array must be sorted.
D. The array's size must be a power of two.
4. Which searching can be performed recursively ? [B]
A. linear search
B. both
C. Binary search
D. none
5. Which searching can be performed iteratively ? [B]
A. linear search
B. both
C. Binary search
D. none
6. In a selection sort of n elements, how many times is the swap function called in the complete execution of the algorithm? [B]
A. 1
B. n^2
C. $n - 1$
D. $n \log n$
7. Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category? [B]
A. $O(n \log n)$ sorts
B. Interchange sorts
C. Divide-and-conquer sorts
D. Average time is quadratic
8. Suppose that a selection sort of 100 items has completed 42 iterations of the main loop. How many items are now guaranteed to be in their final spot (never to be moved again)? [C]
A. 21
B. 41
C. 42
D. 43
9. When is insertion sort a good choice for sorting an array? [B]
A. Each component of the array requires a large amount of memory
B. The array has only a few items out of place
C. Each component of the array requires a small amount of memory
D. The processor speed is fast

10. What is the worst-case time for quick sort to sort an array of n elements? [D]
 A. $O(\log n)$ C. $O(n \log n)$
 B. $O(n)$ D. $O(n^2)$
11. Suppose we are sorting an array of eight integers using quick sort, and we have just finished the first partitioning with the array looking like this: [A]
 2 5 1 7 9 12 11 10 Which statement is correct?
 A. The pivot could be either the 7 or the 9.
 B. The pivot is not the 7, but it could be the 9.
 C. The pivot could be the 7, but it is not the 9.
 D. Neither the 7 nor the 9 is the pivot
12. What is the worst-case time for heap sort to sort an array of n elements? [C]
 A. $O(\log n)$ C. $O(n \log n)$
 B. $O(n)$ D. $O(n^2)$
13. Suppose we are sorting an array of eight integers using heap sort, and we have just finished one of the reheapifications downward. The array now looks like this: 6 4 5 1 2 7 8 [B]
 How many reheapifications downward have been performed so far?
 A. 1 C. 2
 B. 3 or 4 D. 5 or 6
14. Time complexity of inserting an element to a heap of n elements is of the order of [A]
 A. $\log_2 n$ C. $n \log_2 n$
 B. n^2 D. n
15. A min heap is the tree structure where smallest element is available at the [B]
 A. leaf C. intermediate parent
 B. root D. any where
16. In the quick sort method , a desirable choice for the portioning element will be [C]
 A. first element of list C. median of list
 B. last element of list D. any element of list
17. Quick sort is also known as [D]
 A. merge sort C. heap sort
 B. bubble sort D. none
18. Which design algorithm technique is used for quick sort . [A]
 A. Divide and conqueror C. backtrack
 B. greedy D. dynamic programming
19. Which among the following is fastest sorting technique (for unordered data) [C]
 A. Heap sort C. Quick Sort
 B. Selection Sort D. Bubble sort
20. In which searching technique elements are eliminated by half in each pass . [C]
 A. Linear search C. Binary search
 B. both D. none
21. Running time of Heap sort algorithm is ----- [B]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$

22. Running time of Bubble sort algorithm is -----. [D]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
23. Running time of Selection sort algorithm is -----. [D]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
24. The Max heap constructed from the list of numbers 30,10,80,60,15,55 is [C]
 A. 60,80,55,30,10,15 C. 80,55,60,15,10,30
 B. 80,60,55,30,10,15 D. none
25. The number of swappings needed to sort the numbers 8,22,7,9,31,19,5,13 [D]
 in ascending order using bubble sort is
 A. 11 C. 13
 B. 12 D. 14
26. Time complexity of insertion sort algorithm in best case is [C]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
27. Binary search algorithm performs efficiently on a [C]
 A. linked list C. array
 B. both D. none
28. Which is a stable sort ? [D]
 A. Bubble sort C. Quick sort
 B. Selection Sort D. none
29. Heap is a good data structure to implement [A]
 A. priority Queue C. linear queue
 B. Deque D. none
30. Always Heap is a [A]
 A. complete Binary tree C. Full Binary tree
 B. Binary Search Tree D. none

Chapter 4

Stack and Queue

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

- *Stack.*
- *Queue.*

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

- **STACK:**

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

5. *Push*: is the term used to insert an element into a stack.
6. *Pop*: is the term used to delete an element from a stack.

~~—Push~~ is the term used to insert an element into a stack. ~~—Pop~~ is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

- **Representation of Stack:**

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When an element is added to a stack, the operation is performed by `push()`. Figure 4.1 shows the creation of a stack and addition of elements using `push()`.

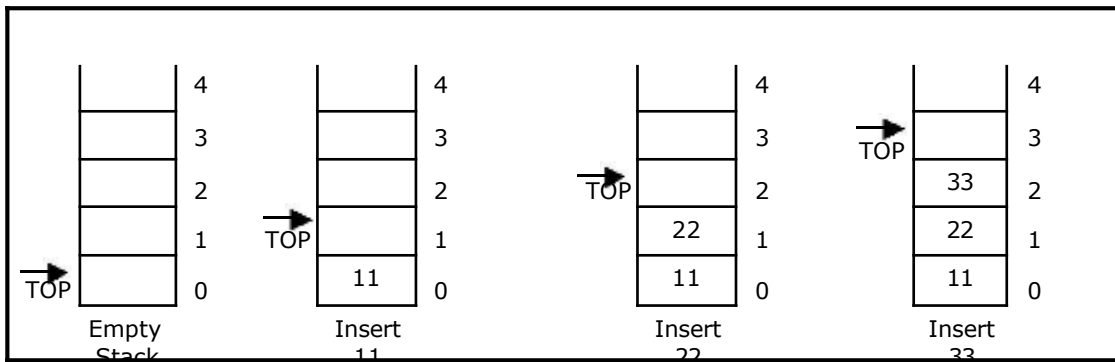


Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using pop().

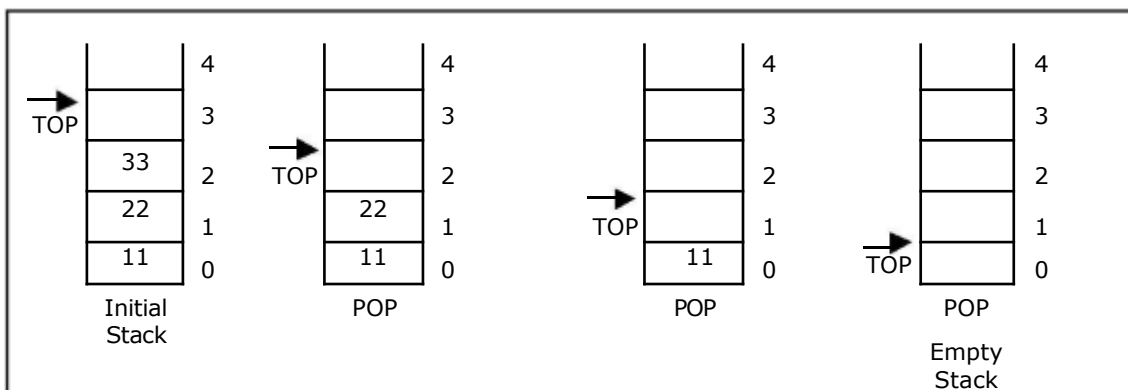


Figure 4.2. Pop operations on stack

Source code for stack operations, using array:

```

5.    include <stdio.h>
6.    include <conio.h>
7.    include <stdlib.h>
8.    define MAX 6 int
stack[MAX];
int top = 0;
int menu()
{
    int ch;
    clrscr();
    printf("\n ... Stack operations using ARRAY... ");
    printf("\n -----*****-----\n");
    printf("\n 1. Push ");
    printf("\n 2. Pop ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice:
"); scanf("%d", &ch);
    return ch;
}
void display()
{
    int i;
    if(top == 0)
    {
        printf("\n\nStack empty..");
    }
}

```

```

        return;
    }
    else
    {
        printf("\n\nElements in stack:");
        for(i = 0; i < top; i++)
            printf("\t%d", stack[i]);
    }
}

void pop()
{
    if(top == 0)
    {
        printf("\n\nStack Underflow..");
        return;
    }
    else
        printf("\n\npopped element is: %d ", stack[--top]);
}

void push()
{
    int data;
    if(top == MAX)
    {
        printf("\n\nStack Overflow..");
        return;
    }
    else
    {
        printf("\n\nEnter data: ");
        scanf("%d", &data);
        stack[top] = data;
        top = top + 1;
        printf("\n\nData Pushed into the stack");
    }
}

void main()
{
    int ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case 1:
                push();
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
        }
        getch();
    } while(1);
}

```

\{

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using *top* pointer. The linked stack looks as shown in figure 4.3.

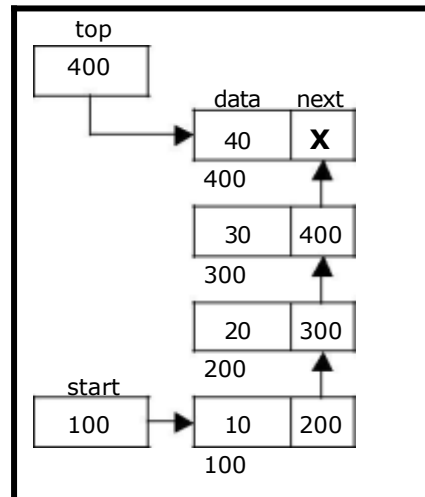


Figure 4.3. Linked stack representation

Source code for stack operations, using linked list:

```
include <stdio.h>
include <conio.h>
include <stdlib.h>

struct stack
{
    int data;
    struct stack *next;
};

void push();
void pop();
void display();
typedef struct stack node;
node *start=NULL;
node *top = NULL;

node* getnode()
{
    node *temp;
    temp=(node *) malloc( sizeof(node)) ;
    printf("\n Enter data ");
    scanf("%d", &temp -> data);
    temp -> next = NULL; return
    temp;
}
void push(node *newnode)
{
    node *temp;
    if( newnode == NULL )
    {
        printf("\n Stack Overflow..");
        return;
    }
}
```

```

    if(start == NULL)
    {
        start = newnode;
        top = newnode;
    }
    else
    {
        temp = start;
        while( temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newnode;
        top = newnode;
    }
    printf("\n\n\t Data pushed into stack");
}
void pop()
{
    node *temp;
    if(top == NULL)
    {
        printf("\n\n\t Stack
underflow"); return;
    }
    temp = start;
    if( start -> next == NULL)
    {
        printf("\n\n\t Popped element is %d ", top -> data);
        start = NULL;
        free(top);
        top = NULL;
    }
    else
    {
        while(temp -> next != top)
        {
            temp = temp -> next;
        }
        temp -> next = NULL;
        printf("\n\n\t Popped element is %d ", top -> data);
        free(top);
        top = temp;
    }
}
void display()
{
    node *temp;
    if(top == NULL)
    {
        printf("\n\n\t\t Stack is empty ");
    }
    else
    {
        temp = start;
        printf("\n\n\n\t\t Elements in the stack: \n");
        printf("%5d ", temp -> data);
        while(temp != top)
        {
            temp = temp -> next;
            printf("%5d ", temp -> data);
        }
    }
}
}

```



```

char menu()
{
    char ch;
    clrscr();
    printf("\n \tStack operations using pointers.. ");
    printf("\n -----*****-----\n");
    printf("\n 1. Push ");
    printf("\n 2. Pop ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice:
"); ch = getche();
    return ch;
}

void main()
{
    char ch;
    node *newnode;
    do
    {
        ch = menu();
        switch(ch)
        {
            case '1' :
                newnode = getnode();
                push(newnode); break;

            case '2' :
                pop();
                break;

            case '3' :
                display();
                break;

            case '4':
                return;
        }
        getch();
    } while( ch != '4' );
}

```

Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: $(A + B) * (C - D)$

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as

polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B - C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and \$ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation (\$ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

- 1 Infix to postfix
- 2 Infix to prefix
- 3 Postfix to infix
- 4 Postfix to prefix
- 5 Prefix to infix
- 6 Prefix to postfix

Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

9. If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
10. If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:

Convert $((A - (B + C)) * D) \uparrow (E + F)$ infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
((
(((
A	A	((
-	A	((-	
(A	((-(
B	A B	((-(
+	A B	((-(+	
C	A B C	((-(+	
)	A B C +	((-	
)	A B C + -	(
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(A B C + - D *	↑(
E	A B C + - D * E	↑(
+	A B C + - D * E	↑(+	
F	A B C + - D * E F	↑(+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert $a + b * c + (d * e + f) * g$ the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	

*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(a b c * +	+ (
d	a b c * + d	+ (
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression $A + B * C - D / E * H$ into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression $A + (B * C - (D / E \uparrow F) * G) * H$ into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	

(A	+ (
B	A B	+ (
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(A B C *	+ (- (
D	A B C * D	+ (- (
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/↑	
F	A B C * D E F	+ (- (/↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

1. Program to convert an infix to postfix expression:

```
# include <string.h>
```

```
char postfix[50];
char infix[50];
char opstack[50]; /* operator stack */ int i, j, top = 0;
```

```
int lesspriority(char op, char op_at_stack)
{
    int k;
    int pv1; /* priority value of op */
    int pv2; /* priority value of op_at_stack */
    char operators[] = {'+', '-', '*', '/', '%', '^', '('};
    int priority_value[] = {0,0,1,1,2,3,4};
    if( op_at_stack == '(' )
        return 0;
    for(k = 0; k < 6; k++)
    {
        if(op == operators[k])
            pv1 = priority_value[k];
    }
    for(k = 0; k < 6; k++)
    {
        if(op_at_stack == operators[k])
            pv2 = priority_value[k];
    }
    if(pv1 < pv2)
        return 1;
    else
        return 0;
}
```

```

void push(char op)      /* op - operator */
{
    if(top == 0)
    {
        opstack[top] = op;
        top++;
    }
    else
    {
        if(op != '(' )
        {
            while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
            {
                postfix[j] = opstack[--top]; j++;
            }
            opstack[top] = op; /* pushing onto stack */
            top++;
        }
    }
}

pop()
{
    while(opstack[--top] != '(' )      /* pop until '(' comes */
    {
        postfix[j] = opstack[top];
        j++;
    }
}

void main()
{
    char ch;
    clrscr();
    printf("\n Enter Infix Expression : ");
    gets(infix);
    while( (ch=infix[i++]) != '\0')
    {
        switch(ch)
        {
            case ' ' : break;
            case '(' :
            case '+' :
            case '-' :
            case '*' :
            case '/' :
            case '^' :
            case '%' :
                push(ch); /* check priority and push */ break;

            case ')' :
                pop();
                break;
            default :
                postfix[j] = ch;
                j++;
        }
    }
    while(top >= 0)
    {
        postfix[j] = opstack[--top];
        j++;
    }
}

```

```

/* before pushing the operator
'op' into the stack check priority
of op with top of opstack if less
then pop the operator from stack
then push into postfix string else
push op onto stack itself */

```

```

    }
    postfix[j] = '\0';
    printf("\n Infix Expression : %s ", infix);
    printf("\n Postfix Expression : %s ", postfix);
    getch();
}

```

- **Conversion from infix to prefix:**

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression $A + B - C$ into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	BC	-	
+	BC	- +	
A	A BC	- +	
End of string	- + A BC	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the infix expression $(A + B) * (C - D)$ into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
))	
D	D)	
-	D) -	
C	CD) -	
(- CD		
*	- CD	*	
)	- CD	*)	
B	B - CD	*)	
+	B - CD	*) +	
A	A B - CD	*) +	
(+ A B - CD	*	
End of string	* + A B - CD	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the infix expression $A \uparrow B * C - D + E / F / (G + H)$ into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
))	
H	H)	
+	H) +	
G	GH) +	
(+ GH		
/	+ GH	/	
F	F + GH	/	
/	F + GH	//	
E	EF + GH	//	
+	// EF + GH	+	
D	D // EF + GH	+	
-	D // EF + GH	+ -	
C	CD // EF + GH	+ -	
*	CD // EF + GH	+ - *	
B	BCD // EF + GH	+ - *	
\uparrow	BCD // EF + GH	+ - * \uparrow	
A	ABCD // EF + GH	+ - * \uparrow	
End of string	+ - * \uparrow ABCD // EF + GH	The input is now empty. Pop the output symbols from the stack until it is empty.	

1. Program to convert an infix to prefix expression:

```
11. include <conio.h>
12. include <string.h>
```

```
char prefix[50];
char infix[50];
char opstack[50]; /* operator stack */ int j, top = 0;
```

```
void insert_beg(char ch)
{
    int k;
    if(j == 0)
        prefix[0] = ch;
    else
    {
        for(k = j + 1; k > 0; k--)
            prefix[k] = prefix[k - 1];
        prefix[0] = ch;
    }
    j++;
}
```



```

int lesspriority(char op, char op_at_stack)
{
    int k;
    int pv1;          /* priority value of op */
    int pv2;          /* priority value of op_at_stack */
    char operators[] = {'+', '-', '*', '/', '%', '^', ''};
    int priority_value[] = {0, 0, 1, 1, 2, 3, 4};
    if(op_at_stack == ')')
        return 0;
    for(k = 0; k < 6; k++)
    {
        if(op == operators[k])
            pv1 = priority_value[k];
    }
    for(k = 0; k < 6; k++)
    {
        if( op_at_stack == operators[k] )
            pv2 = priority_value[k];
    }
    if(pv1 < pv2)
        return 1;
    else
        return 0;
}

void push(char op)          /* op - operator */
{
    if(top == 0)
    {
        opstack[top] = op;
        top++;
    }
    else
    {
        if(op != ')')
        {
            /* before pushing the operator 'op' into the stack check priority of op with
            top of operator stack if less pop the operator from stack then push into postfix
            string else push op onto stack itself */

            while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
            {
                insert_beg(opstack[--top]);
            }
            opstack[top] = op;          /* pushing onto stack */
            top++;
        }
    }
}

void pop()
{
    while(opstack[--top] != ')')
        insert_beg(opstack[top]);
}

void main()
{
    char ch;
    int l, i = 0;
    clrscr();
    printf("\n Enter Infix Expression : ");
}

```

```

gets(infix);
l = strlen(infix);
while(l > 0)
{
    ch = infix[--
l]; switch(ch)
    {
        case ' ' : break;
        case ')' :
        case '+' :
        case '-' :
        case '*' :
        case '/' :
        case '^' :
        case '%' :
            push(ch); /* check priority and push */ break;

        case '(' :
            pop();
            break;
        default :
            insert_beg(ch);
    }
}
while( top > 0 )
{
    insert_beg( opstack[--top]
); j++;
}
prefix[j] = '\0';
printf("\n Infix Expression : %s ", infix);
printf("\n Prefix Expression : %s ", prefix);
getch();
}

```

Conversion from postfix to infix:

Procedure to convert postfix expression to infix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent infix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A (((B*C) - ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A (((B*C) - ((D/(E^F))*G)) H	Push H
*	A (((((B*C) - ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((((B*C) - ((D/(E^F))*G)) * H)	
End of string		The input is now empty. The string formed is infix.

Program to convert postfix to infix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
# define MAX 100

void pop (char*);
void push(char*);

char stack[MAX] [MAX];
int top = -1;
```

```

void main()
{
    char s[MAX], str1[MAX], str2[MAX], str[MAX];
    char s1[2],temp[2];
    int i=0;
    clrscr( );
    printf("\nEnter the postfix expression;
"); gets(s);
    while (s[i]!='\0')
    {
        if(s[i] == ' ')                /*skip whitespace, if any*/
            i++;
        if (s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i] == '+' || s[i] == '/')
        {
            pop(str1);
            pop(str2);
            temp[0]='(';
            temp[1]='\0';
            strcpy(str, temp);
            strcat(str, str2);
            temp[0] = s[i];
            temp[1] = '\0';
            strcat(str,temp);
            strcat(str, str1);
            temp[0] = ')';
            temp[1] = '\0';
            strcat(str,temp);
            push(str);
        }
        else
        {
            temp[0]=s[i];
            temp[1]='\0';
            strcpy(s1,
temp); push(s1);
        }
        i++;
    }
    printf("\nThe Infix expression is: %s", stack[0]);
}

void pop(char *a1)
{
    strcpy(a1,stack[top]);
    top--;
}

void push (char*str)
{
    if(top == MAX - 1)
        printf("\nstack is full");
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}

```

Conversion from postfix to prefix:

Procedure to convert postfix expression to prefix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in front of the operands and push it onto the stack.
5. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression $A B C * D E F ^ / G * - H * +$ into its equivalent prefix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A *BC	Pop two operands and place the operator in front the operands and push the string.
D	A *BC D	Push D
E	A *BC D E	Push E
F	A *BC D E F	Push F
^	A *BC D ^EF	Pop two operands and place the operator in front the operands and push the string.
/	A *BC /D^EF	Pop two operands and place the operator in front the operands and push the string.
G	A *BC /D^EF G	Push G
*	A *BC */D^EFG	Pop two operands and place the operator in front the operands and push the string.
-	A - *BC*/D^EFG	Pop two operands and place the operator in front the operands and push the string.
H	A - *BC*/D^EFG H	Push H
*	A *- *BC*/D^EFGH	Pop two operands and place the operator in front the operands and push the string.
+	+A*- *BC*/D^EFGH	
End of string	The input is now empty. The string formed is prefix.	

Program to convert postfix to prefix expression:

```
# include <conio.h>
# include <string.h>

#define MAX 100
void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top ==-1;

main()
{
    char s[MAX], str1[MAX], str2[MAX], str[MAX];
    char s1[2], temp[2];
    int i = 0;
    clrscr();
    printf("Enter the postfix expression;
"); gets (s);
    while(s[i]!='\0')
    {
        /*skip whitespace, if any */
        if (s[i] == ' ')
            i++;
        if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i]== '+' || s[i] == '/')
        {
            pop (str1); pop
            (str2); temp[0]
            = s[i]; temp[1]
            = '\0';
            strcpy (str, temp);
            strcat(str, str2);
            strcat(str, str1);
            push(str);
        }
        else
        {
            temp[0] = s[i];
            temp[1] = '\0';
            strcpy (s1,
            temp); push (s1);
        }
        i++;
    }
    printf("\n The prefix expression is: %s", stack[0]);
}

void pop(char*a1)
{
    if(top == -1)
    {
        printf("\nStack is empty");
        return ;
    }
    else
    {
        strcpy (a1,
        stack[top]); top--;
    }
}
```

```

void push (char *str)
{
    if(top == MAX - 1)
        printf("\nstack is full");
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}

```

Conversion from prefix to infix:

Procedure to convert prefix expression to infix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression $+ A * - * B C * / D ^ E F G H$ into its equivalent infix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G (E^F)	Pop two operands and place the operator in between the operands and push the string.
D	H G (E^F) D	Push D
/	H G (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
*	H ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
C	H ((D/(E^F))*G) C	Push C
B	H ((D/(E^F))*G) C B	Push B
*	H ((D/(E^F))*G) (B*C)	Pop two operands and place the operator in front the operands and push the string.
-	H ((B*C)-((D/(E^F))*G))	Pop two operands and place the operator in front the operands and push the

string.

*

(((B*C)-((D/(E^F))*G))*H)	
---------------------------	--

Pop two operands and place the operator in front the operands and push the string.

A

(((B*C)-((D/(E^F))*G))*H)	A	
---------------------------	---	--

Push A

+

(A+(((B*C)-((D/(E^F))*G))*H))	
-------------------------------	--

Pop two operands and place the operator in front the operands and push the string.

End of string

The input is now empty. The string formed is infix.

Program to convert prefix to infix expression:

```
# include <string.h>
# define MAX 100
```

```
void pop (char*);
void push(char*);
char stack[MAX] [MAX];
int top = -1;
```

```
void main()
{
```

```
    char s[MAX], str1[MAX], str2[MAX], str[MAX];
    char s1[2],temp[2];
    int i=0;
    clrscr( );
    printf("\nEnter the prefix expression; ");
    gets(s);
    strrev(s);
    while (s[i]!='\0')
```

```
    {
        /*skip whitespace, if any*/
        if(s[i] == ' ')
            i++;
        if (s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i] == '+' || s[i] == '/')
        {
            pop(str1);
            pop(str2);
            temp[0]='(';
            temp[1]='\0';
            strcpy(str, temp);
            strcat(str, str1);
            temp[0] = s[i];
            temp[1] = '\0';
            strcat(str,temp);
            strcat(str, str2);
            temp[0] = ')';
            temp[1] = '\0';
            strcat(str,temp);
            push(str);
        }
        else
        {
            temp[0]=s[i];
            temp[1]='\0';
            strcpy(s1,
            temp); push(s1);
```



```

        }
        i++;
    }
    printf("\nThe infix expression is: %s", stack[0]);
}

void pop(char *a1)
{
    strcpy(a1,stack[top]);
    top--;
}

void push (char*str)
{
    if(top == MAX - 1)
        printf("\nstack is full");
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}

```

Conversion from prefix to postfix:

Procedure to convert prefix expression to postfix expression is as follows:

- Scan the prefix expression from right to left (reverse order).
- If the scanned symbol is an operand, then push it onto the stack.
- If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator after the operands and push it onto the stack.
- Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent postfix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G EF^	Pop two operands and place the operator after the operands and push the string.
D	H G EF^ D	Push D

/	H G DEF^/	Pop two operands and place the operator after the operands and push the string.
*	H DEF^/G*	Pop two operands and place the operator after the operands and push the string.
C	H DEF^/G* C	Push C
B	H DEF^/G* C B	Push B
*	H DEF^/G* BC*	Pop two operands and place the operator after the operands and push the string.
-	H BC*DEF^/G*-	Pop two operands and place the operator after the operands and push the string.
*	BC*DEF^/G*-H*	Pop two operands and place the operator after the operands and push the string.
A	BC*DEF^/G*-H* A	Push A
+	ABC*DEF^/G*-H*+	Pop two operands and place the operator after the operands and push the string.
End of string		The input is now empty. The string formed is postfix.

Program to convert prefix to postfix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>

#define MAX 100

void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top = -1;

void main()
{
    char s[MAX], str1[MAX], str2[MAX], str[MAX];
    char s1[2], temp[2];
    int i = 0;
    clrscr();
    printf("Enter the prefix expression; ");
    gets (s);
    strrev(s);
    while(s[i]!='\0')
    {
        if (s[i] == ' ') /*skip whitespace, if any */ i++;

        if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i]== '+' || s[i] == '/')
        {
            pop (str1); pop
            (str2); temp[0]
            = s[i]; temp[1]
            = '\0';
            strcat(str1,str2);
            strcat (str1, temp);
            strcpy(str, str1);
            push(str);
        }
    }
}
```

```

        else
        {
            temp[0] = s[i];
            temp[1] = '\0';
            strcpy (s1,
                temp); push (s1);
        }
        i++;
    }
    printf("\nThe postfix expression is: %s", stack[0]);
}
void pop(char*a1)
{
    if(top == -1)
    {
        printf("\nStack is empty");
        return ;
    }
    else
    {
        strcpy (a1,
            stack[top]); top--;
    }
}
void push (char *str)
{
    if(top == MAX - 1)
        printf("\nstack is full");
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}
}

```

4.4. Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a <code>*</code> is seen, so 8 and 5 are popped as $8 * 5 = 40$ is pushed
+	5	40	45	6, 45	Next, a <code>+</code> is seen, so 40 and 5 are popped and $40 + 5 = 45$ is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, <code>+</code> pops 3 and 45 and pushes $45 + 3 = 48$ is pushed
*	6	48	288	288	Finally, a <code>*</code> is seen and 48 and 6 are popped, the result $6 * 48 = 288$ is pushed

Example 2:

Evaluate the following postfix expression: $6\ 2\ 3\ +\ -\ 3\ 8\ 2\ /\ +\ * \ 2\ \uparrow\ 3\ +$

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
\uparrow	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Program to evaluate a postfix expression:

```

2 include <conio.h>
3 include <math.h>
4 define MAX 20

int isoperator(char ch)
{
    if(ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^')
        return 1;
    else
        return 0;
}

```

```

void main(void)
{
    char postfix[MAX];
    int val;
    char ch;
    int i = 0, top = 0;
    float val_stack[MAX], val1, val2,
    res; clrscr();
    printf("\n Enter a postfix expression: ");
    scanf("%s", postfix);
    while((ch = postfix[i]) != '\0')
    {
        if(isoperator(ch) == 1)
        {
            val2 = val_stack[--
            top]; val1 = val_stack[-
            top]; switch(ch)
            {
                case '+':
                    res = val1 +
                    val2; break;
                case '-':
                    res = val1 - val2;
                    break;
                case '*':
                    res = val1 * val2;
                    break;
                case '/':
                    res = val1 / val2;
                    break;
                case '^':
                    res = pow(val1, val2);
                    break;
            }
            val_stack[top] = res;
        }
        else
            val_stack[top] = ch-48; /*convert character digit to integer digit */
        top++;
        i++;
    }
    printf("\n Values of %s is : %f ",postfix, val_stack[0] );
    getch();
}

```

Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

Queue:

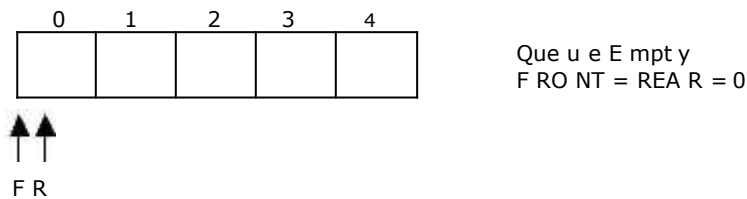
A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a ~~FIFO~~ or First-in-first-out list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

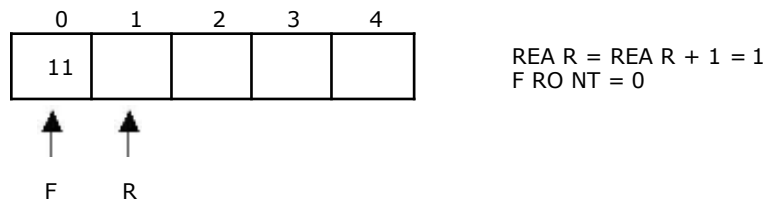
- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

Representation of Queue:

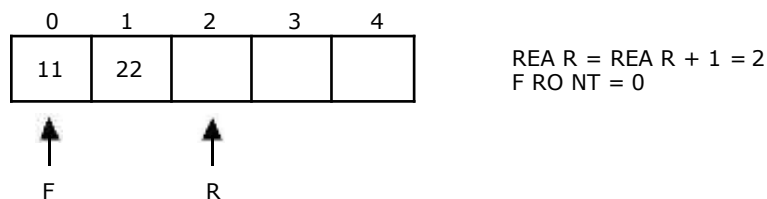
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



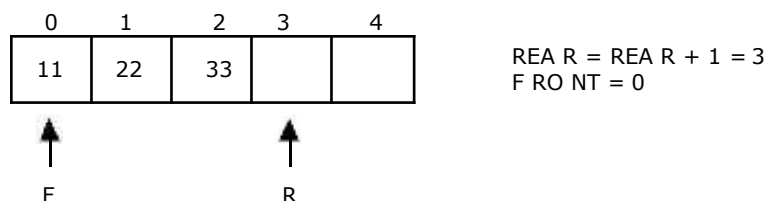
Now, insert 11 to the queue. Then queue status will be:



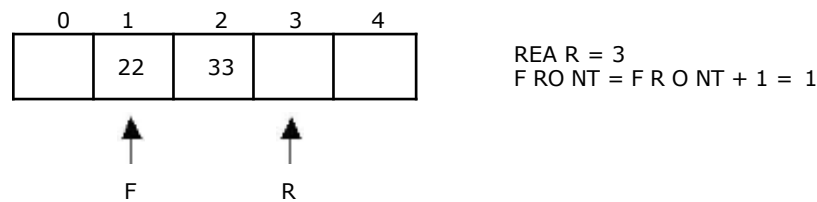
Next, insert 22 to the queue. Then the queue status is:



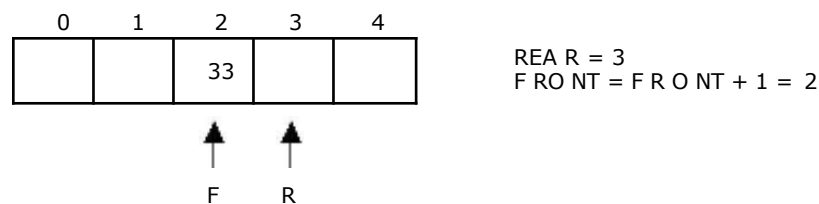
Again insert another element 33 to the queue. The status of the queue is:



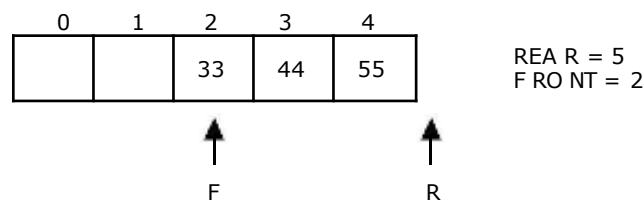
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



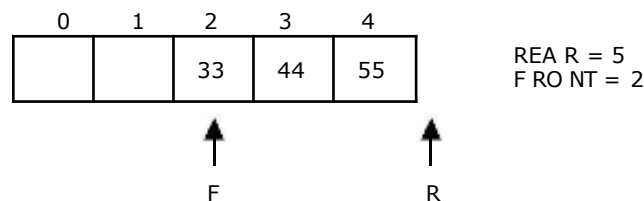
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



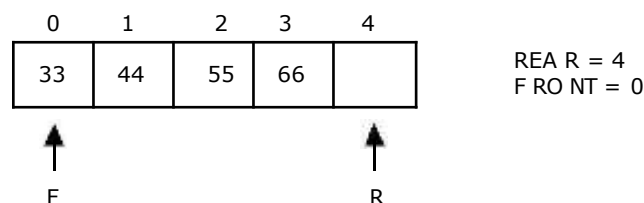
Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue**.

Source code for Queue operations using array:

In order to create a queue we require a one dimensional array $Q(1:n)$ and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and *rear* always points to the last element in the queue. Thus, $front = rear$ if and only if there are no elements in the queue. The initial condition then is $front = rear = 0$. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

insertQ(): inserts an element at the end of queue Q.

deleteQ(): deletes the first element of Q.

displayQ(): displays the elements in the queue.

```
8. include <conio.h>
9. define MAX 6
int Q[MAX];
int front, rear;

void insertQ()
{
    int data;
    if(rear == MAX)
    {
        printf("\n Linear Queue is full");
        return;
    }
    else
    {
        printf("\n Enter data: ");
        scanf("%d", &data);
        Q[rear] = data; rear++;

        printf("\n Data Inserted in the Queue ");
    }
}

void deleteQ()
{
    if(rear == front)
    {
        printf("\n\n Queue is Empty..");
        return;
    }
    else
    {
        printf("\n Deleted element from Queue is %d",
            Q[front]); front++;
    }
}

void displayQ()
{
    int i;
    if(front == rear)
    {
        printf("\n\n\t Queue is Empty");
        return;
    }
    else
    {
        printf("\n Elements in Queue are:
            "); for(i = front; i < rear; i++)
```



```

        {
            printf("%d\t", Q[i]);
        }
    }
}
int menu()
{
    int ch;
    clrscr();
    printf("\n \tQueue operations using ARRAY..");
    printf("\n -----*****-----\n");
    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice:
"); scanf("%d", &ch);
    return ch;
}
void main()
{
    int ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case 1:
                insertQ();
                break;
            case 2:
                deleteQ();
                break;
            case 3:
                displayQ();
                break;
            case 4:
                return;
        }
        getch();
    } while(1);
}

```

Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure 4.4:

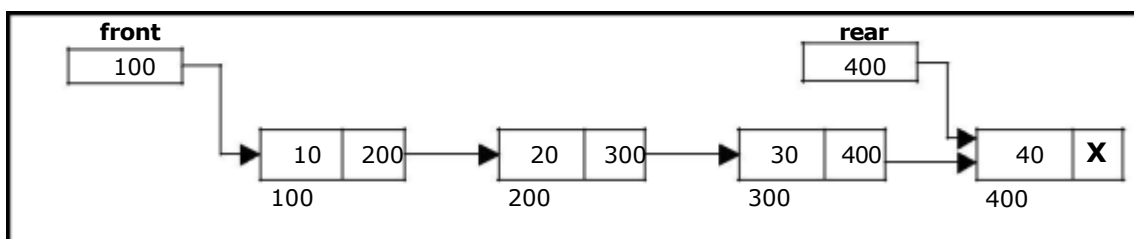


Figure 4.4. Linked Queue representation

Source code for queue operations using linked list:

```
include <stdlib.h>
include <conio.h>

struct queue
{
    int data;
    struct queue *next;
};
typedef struct queue
node; node *front = NULL;
node *rear = NULL;

node* getnode()
{
    node *temp;
    temp = (node *) malloc(sizeof(node)) ;
    printf("\n Enter data ");
    scanf("%d", &temp -> data);
    temp -> next = NULL; return
temp;
}
void insertQ()
{
    node *newnode;
    newnode = getnode();
    if(newnode == NULL)
    {
        printf("\n Queue Full");
        return;
    }
    if(front == NULL)
    {
        front = newnode;
        rear = newnode;
    }
    else
    {
        rear -> next = newnode;
        rear = newnode;
    }
    printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
    node *temp;
    if(front == NULL)
    {
        printf("\n\n\t Empty Queue..");
        return;
    }
    temp = front;
    front = front -> next;
    printf("\n\n\t Deleted element from queue is %d ", temp ->
data); free(temp);
}
}
```

```

void displayQ()
{
    node *temp;
    if(front == NULL)
    {
        printf("\n\n\t\t Empty Queue ");
    }
    else
    {
        temp = front;
        printf("\n\n\n\t\t Elements in the Queue are: ");
        while(temp != NULL )
        {
            printf("%5d ", temp -> data);
            temp = temp -> next;
        }
    }
}

char menu()
{
    char ch;
    clrscr();
    printf("\n \t..Queue operations using pointers.. ");
    printf("\n\t -----*****-----\n");
    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice: ");
    ch = getche();
    return ch;
}

void main()
{
    char ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case '1' :
                insertQ();
                break;
            case '2' :
                deleteQ();
                break;
            case '3' :
                displayQ();
                break;
            case '4':
                return;
        }
        getch();
    } while(ch != '4');
}

```

Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

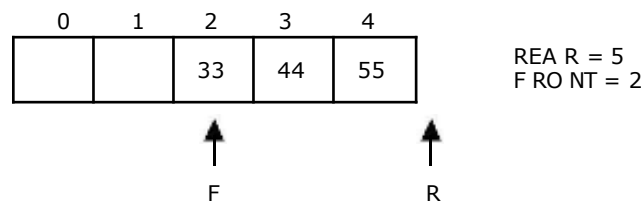
Circular Queue:

A more efficient queue representation is obtained by regarding the array $Q[\text{MAX}]$ as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

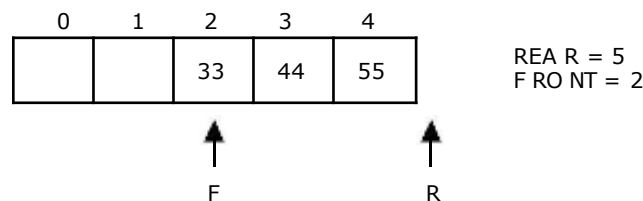
There are two problems associated with linear queue. They are:

- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signaling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

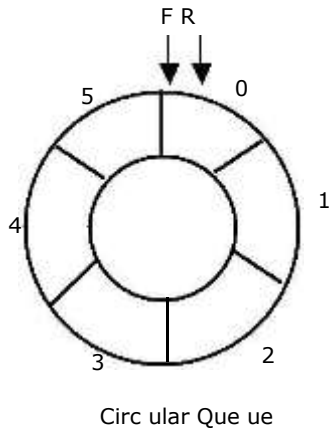


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue**.

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

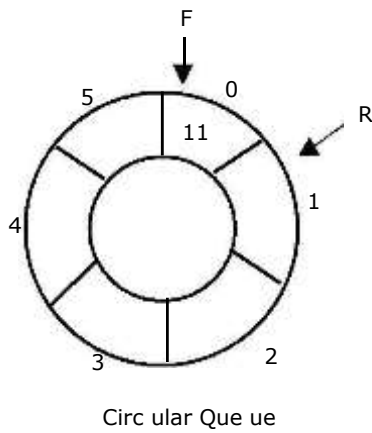
Representation of Circular Queue:

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



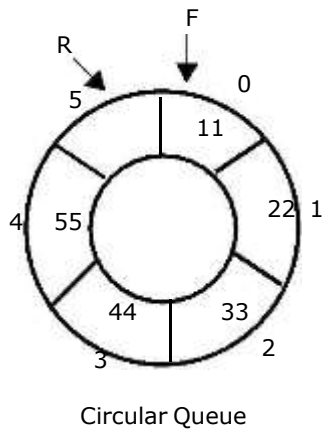
Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

Now, insert 11 to the circular queue. Then circular queue status will be:



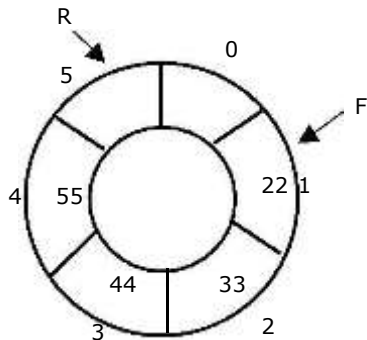
FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

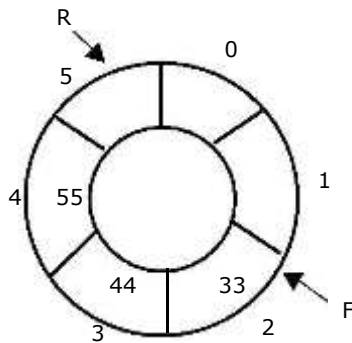
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

$$\begin{aligned} \text{FRONT} &= (\text{FRONT} + 1) \% 6 = 1 \\ \text{REAR} &= 5 \\ \text{COUNT} &= \text{COUNT} - 1 = 4 \end{aligned}$$

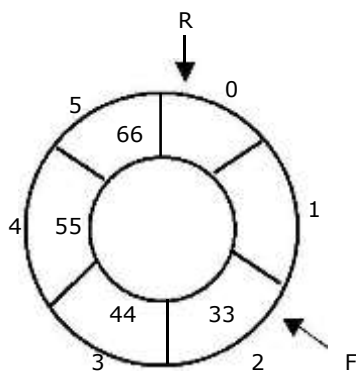
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

$$\begin{aligned} \text{FRONT} &= (\text{FRONT} + 1) \% 6 = 2 \\ \text{REAR} &= 5 \\ \text{COUNT} &= \text{COUNT} - 1 = 3 \end{aligned}$$

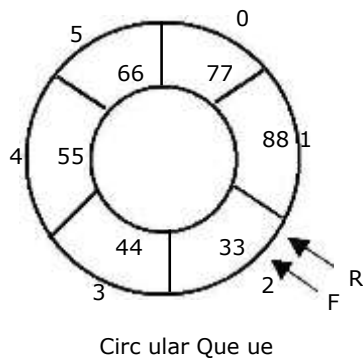
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

$$\begin{aligned} \text{FRONT} &= 2 \\ \text{REAR} &= (\text{REAR} + 1) \% 6 = 0 \\ \text{COUNT} &= \text{COUNT} + 1 = 4 \end{aligned}$$

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



FRONT = 2, REAR = 2
 REAR = REAR % 6 = 2
 COUNT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

Source code for Circular Queue operations, using array:

```
# include <stdio.h>
# include <conio.h>
# define MAX 6

int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;

void insertCQ()
{
    int data;
    if(count == MAX)
    {
        printf("\n Circular Queue is Full");
    }
    else
    {
        printf("\n Enter data: ");
        scanf("%d", &data);
        CQ[rear] = data;
        rear = (rear + 1) % MAX;
        count ++;
        printf("\n Data Inserted in the Circular Queue ");
    }
}

void deleteCQ()
{
    if(count == 0)
    {
        printf("\n\nCircular Queue is Empty..");
    }
    else
    {
        printf("\n Deleted element from Circular Queue is %d ", CQ[front]);
        front = (front + 1) % MAX;
        count --;
    }
}
```

```

void displayCQ()
{
    int i, j;
    if(count == 0)
    {
        printf("\n\n\t Circular Queue is Empty ");
    }
    else
    {
        printf("\n Elements in Circular Queue are: ");
        j = count;
        for(i = front; j != 0; j--)
        {
            printf("%d\t", CQ[i]);
            i = (i + 1) % MAX;
        }
    }
}

int menu()
{
    int ch;
    clrscr();
    printf("\n \t Circular Queue Operations using ARRAY..");
    printf("\n -----*****-----\n");
    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter Your Choice:
"); scanf("%d", &ch);
    return ch;
}

void main()
{
    int ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case 1:
                insertCQ();
                break;
            case 2:
                deleteCQ();
                break;
            case 3:
                displayCQ();
                break;
            case 4:
                return;
            default:
                printf("\n Invalid Choice ");
        }
        getch();
    } while(1);
}

```


Deque:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. Figure 4.5 shows the representation of a deque.

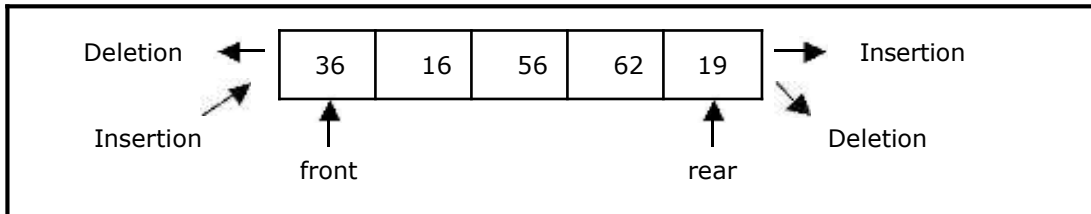


Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

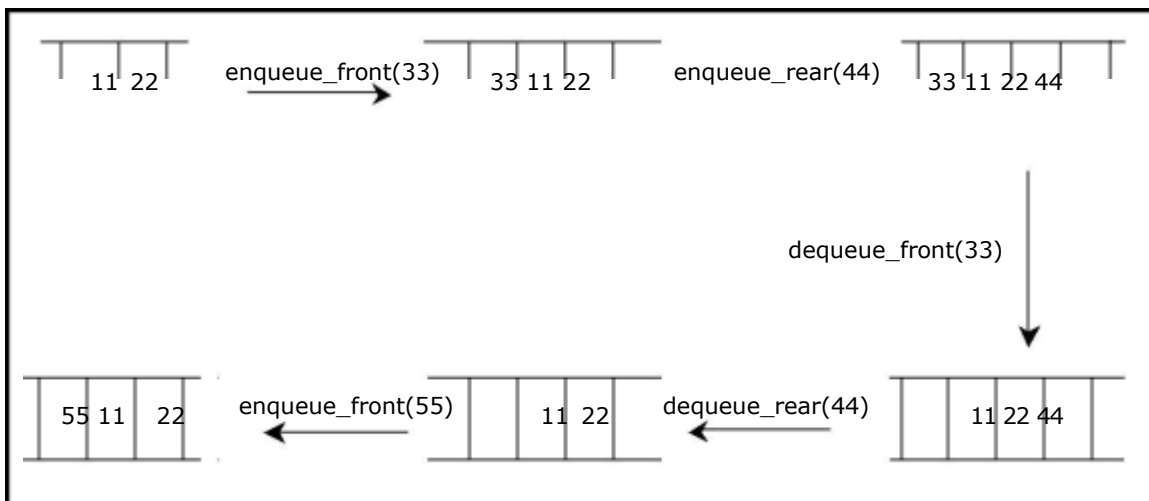


Figure 4.6. Basic operations on deque

There are two variations of deque. They are:

- Input restricted deque (IRD)
- Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

Priority Queue:

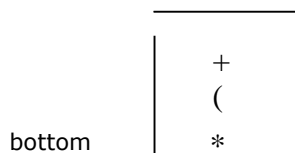
A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

Exercises

1. What is a linear data structure? Give two examples of linear data structures.
2. Is it possible to have two designs for the same data structure that provide the same functionality but are implemented differently?
3. What is the difference between the logical representation of a data structure and the physical representation?
4. Transform the following infix expressions to reverse polish notation:
 - a) $A \uparrow B * C - D + E / F / (G + H)$
 - b) $((A + B) * C - (D - E)) \uparrow (F + G)$
 - c) $A - B / (C * D \uparrow E)$
 - d) $(a + b \uparrow c \uparrow d) * (e + f / d)$
 - f) $3 - 6 * 7 + 2 / 4 * 5 - 8$
 - g) $(A - B) / ((D + E) * F)$
 - h) $((A + B) / D) \uparrow ((E - F) * G)$
5. Evaluate the following postfix expressions:
 - a) $P_1: 5, 3, +, 2, *, 6, 9, 7, -, /, -$
 - b) $P_2: 3, 5, +, 6, 4, -, *, 4, 1, -, 2, \uparrow, +$
 - c) $P_3: 3, 1, +, 2, \uparrow, 7, 4, -, 2, *, +, 5, -$
6. Consider the usual algorithm to convert an infix expression to a postfix expression. Suppose that you have read 10 input characters during a conversion and that the stack now contains these symbols:



Now, suppose that you read and process the 11th symbol of the input. Draw the stack for the case where the 11th symbol is:

- A. A number:
- B. A left parenthesis:
- C. A right parenthesis:
- D. A minus sign:
- E. A division sign:

7. Write a program using stack for parenthesis matching. Explain what modifications would be needed to make the parenthesis matching algorithm check expressions with different kinds of parentheses such as $()$, $[\]$ and $\{ \}$'s.
8. Evaluate the following prefix expressions:
 - a) $+ * 2 + / 14 2 5 1$
 - b) $- * 6 3 - 4 1$
 - c) $+ + 2 6 + - 13 2 4$
9. Convert the following infix expressions to prefix notation:
 - a) $((A + 2) * (B + 4)) - 1$
 - b) $Z - (((X + 1) * 2) - 5) / Y$
 - c) $((C * 2) + 1) / (A + B)$
 - d) $((A + B) * C - (D - E)) \uparrow (F + G)$
 - e) $A - B / (C * D \uparrow E)$
10. Write a copy function to copy one stack to another assuming
 - a) The stack is implemented using array.
 - b) The stack is implemented using linked list.
11. Write an algorithm to construct a fully parenthesized infix expression from its postfix equivalent. Write a construct function for your algorithm.
12. How can one convert a postfix expression to its prefix equivalent and vice-versa?
13. A double-ended queue (deque) is a linear list where additions and deletions can be performed at either end. Represent a deque using an array to store the elements of the list and write the add functions for additions and deletions.
14. In a circular queue represented by an array, how can one specify the number of elements in the queue in terms of front , rear and MAX-QUEUE-SIZE ? Write a delete function to delete the K-th element from the front of a circular queue.
15. Can a queue be represented by a circular linked list with only one pointer pointing to the tail of the queue? Write add and delete operations on such a queue
16. Write a isWellFormed function to test whether a string of opening and closing parenthesis is well formed or not.
17. Represent N queues in a single one-dimensional array. Write functions for add and delete operations on the i^{th} queue
18. Represent a stack and queue in a single one-dimensional array. Write functions for push , pop operations on the stack and add , delete functions on the queue.

Multiple Choice Questions

1. Which among the following is a linear data structure: [D]
 A. Queue
 B. Stack
 C. Linked List
 D. all the above
2. Which among the following is a Dynamic data structure: [A]
 A. Double Linked List C. Stack
 B. Queue D. all the above
3. Stack is referred as: [A]
 A. Last in first out list C. both A and B
 B. First in first out list D. none of the above
4. A stack is a data structure in which all insertions and deletions of entries are made at: [A]
 A. One end C. Both the ends
 B. In the middle D. At any position
5. A queue is a data structure in which all insertions and deletions are made respectively at: [A]
 A. rear and front C. front and rear
 B. front and front D. rear and rear
6. Transform the following infix expression to postfix form: [D]
 $(A + B) * (C - D) / E$
 A. $A B * C + D / -$ C. $A B + C D * - / E$
 B. $A B C * C D / - +$ D. $A B + C D - * E /$
7. Transform the following infix expression to postfix form: [B]
 $A - B / (C * D)$
 A. $A B * C D - /$ C. $/ - D C * B A$
 B. $A B C D * / -$ D. $- / * A B C D$
8. Evaluate the following prefix expression: $* - + 4 3 5 / + 2 4 3$ [A]
 A. 4 C. 1
 B. 8 D. none of the above
9. Evaluate the following postfix expression: $1 4 18 6 / 3 + + 5 / +$ [C]
 A. 8 C. 3
 B. 2 D. none of the above
10. Transform the following infix expression to prefix form: [B]
 $((C * 2) + 1) / (A + B)$
 A. $A B + 1 2 C * + /$ C. $/ * + 1 2 C A B +$
 B. $/ + * C 2 1 + A B$ D. none of the above
11. Transform the following infix expression to prefix form: [D]
 $Z - (((X + 1) * 2) - 5) / Y$
 A. $/ - * + X 1 2 5 Y$ C. $/ * - + X 1 2 5 Y$
 B. $Y 5 2 1 X + * - /$ D. none of the above
12. Queue is also known as: [B]
 A. Last in first out list C. both A and B
 B. First in first out list D. none of the above

13. One difference between a queue and a stack is: [C]
 A. Queues require dynamic memory, but stacks do not.
 B. Stacks require dynamic memory, but queues do not.
 C. Queues use two ends of the structure; stacks use only one.
 D. Stacks use two ends of the structure, queues use only one.
14. If the characters 'D', 'C', 'B', 'A' are placed in a queue (in that order), and then removed one at a time, in what order will they be removed? [D]
 A. ABCD
 B. ABDC
 C. DCAB
 D. DCBA
15. Suppose we have a circular array implementation of the queue class, with ten items in the queue stored at data[2] through data[11]. The CAPACITY is 42. Where does the push member function place the new entry in the array? [D]
 A. data[1]
 B. data[2]
 C. data[11]
 D. data[12]
16. Consider the implementation of the queue using a circular array. What goes wrong if we try to keep all the items at the front of a partially-filled array (so that data[0] is always the front). [B]
 A. The constructor would require linear time.
 B. The get_front function would require linear time.
 C. The insert function would require linear time.
 D. The is_empty function would require linear time.
17. In the linked list implementation of the queue class, where does the push member function place the new entry on the linked list? [A]
 A. At the head
 B. At the tail
 C. After all other entries that are greater than the new entry.
 D. After all other entries that are smaller than the new entry.
18. In the circular array version of the queue class (with a fixed-sized array), which operations require linear time for their worst-case behavior? []
 A. front
 B. push
 C. empty
 D. None of these.
19. In the linked-list version of the queue class, which operations require linear time for their worst-case behavior? []
 A. front
 B. push
 C. empty
 D. None of these operations.
20. To implement the queue with a linked list, keeping track of a front pointer and a rear pointer. Which of these pointers will change during an insertion into a NONEMPTY queue? [B]
 A. Neither changes
 B. Only front_ptr changes.
 C. Only rear_ptr changes.
 D. Both change.
21. To implement the queue with a linked list, keeping track of a front pointer and a rear pointer. Which of these pointers will change during an insertion into an EMPTY queue? [D]
 A. Neither changes
 B. Only front_ptr changes.
 C. Only rear_ptr changes.
 D. Both change.

22. Suppose top is called on a priority queue that has exactly two entries with equal priority. How is the return value of top selected? [B]
 A. The implementation gets to choose either one.
 B. The one which was inserted first.
 C. The one which was inserted most recently.
 D. This can never happen (violates the precondition)
23. Entries in a stack are "ordered". What is the meaning of this statement? [D]
 A. A collection of stacks can be sorted.
 B. Stack entries may be compared with the '<' operation.
 C. The entries must be stored in a linked list.
 D. There is a first entry, a second entry, and so on.
24. The operation for adding an entry to a stack is traditionally called: [D]
 A. add
 B. append
 C. insert
 D. push
25. The operation for removing an entry from a stack is traditionally called: [C]
 A. delete
 B. peek
 C. pop
 D. remove
26. Which of the following stack operations could result in stack underflow? [A]
 A. is_empty
 B. pop
 C. push
 D. Two or more of the above answers
27. Which of the following applications may use a stack? [D]
 A. A parentheses balancing program.
 B. Keeping track of local variables at run time.
 C. Syntax analyzer for a compiler.
 D. All of the above.
28. Here is an infix expression: $4 + 3 * (6 * 3 - 12)$. Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. What is the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression? [D]
 A. 1
 B. 2
 C. 3
 D. 4
29. What is the value of the postfix expression $6\ 3\ 2\ 4\ +\ -\ *$ [A]
 A. Something between -15 and -100
 B. Something between -5 and -15
 C. Something between 5 and -5
 D. Something between 5 and 15
 E. Something between 15 and 100
30. If the expression $((2 + 3) * 4 + 5 * (6 + 7) * 8) + 9$ is evaluated with * having precedence over +, then the value obtained is same as the value of which of the following prefix expressions? [A]
 A. $++ * + 2\ 3\ 4 * * 5 + 6\ 7\ 8\ 9$
 B. $+ * ++ + 2\ 3\ 4 * * 5 + 6\ 7\ 8\ 9$
 C. $* + + + 2\ 3\ 4 * * 5 + 6\ 7\ 8\ 9$
 D. $+ * ++ + 2\ 3\ 4 + + 5 * 6\ 7\ 8\ 9$
31. Evaluate the following prefix expression: [B]
 $+ * 2 + / 14\ 2\ 5\ 1$
 A. 50
 B. 25
 C. 40
 D. 15

32. Parenthesis are never needed prefix or postfix expression: [A]
A. True C. Cannot be expected
B. False D. None of the above
33. A postfix expression is merely the reverse of the prefix expression: [B]
A. True C. Cannot be expected
B. False D. None of the above
34. Which among the following data structure may give overflow error, even though the current number of elements in it, is less than its size: [A]
A. Simple Queue C. Stack
B. Circular Queue D. None of the above
35. Which among the following types of expressions does not require precedence rules for evaluation: [C]
A. Fully parenthesized infix expression
B. Prefix expression
C. both A and B
D. none of the above
36. Conversion of infix arithmetic expression to postfix expression uses: [D]
A. Stack C. linked list
B. circular queue D. Queue

Hybrid structures:

If two basic types of structures are mixed then it is a hybrid form. Then one part contiguous and another part non-contiguous. For example, figure 1.5 shows how to implement a double-linked list using three parallel arrays, possibly stored a part from each other in memory.

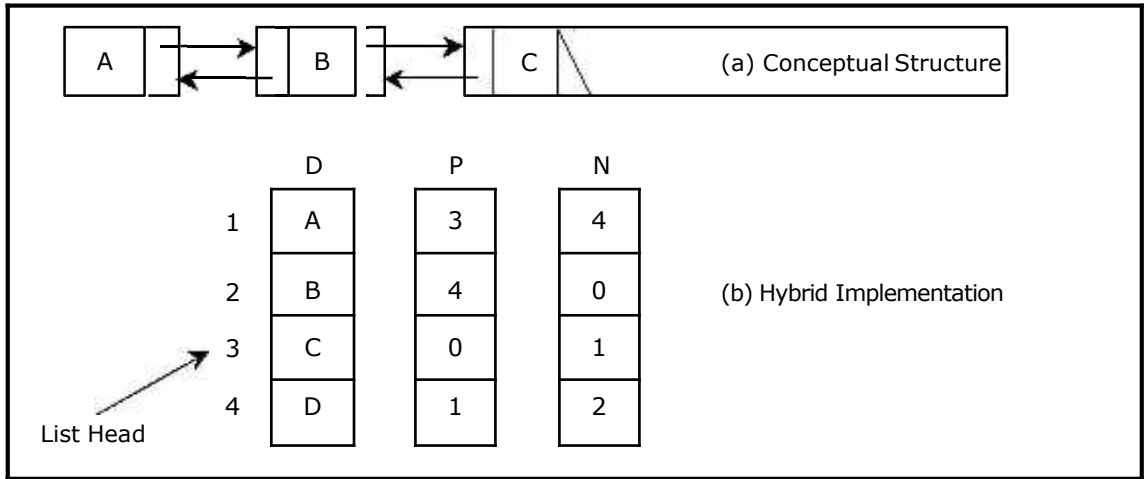


Figure 1.5. A double linked list via a hybrid data structure

The array D contains the data for the list, whereas the array P and N hold the previous and next —pointers—. The pointers are actually nothing more than indexes into the D array. For instance, D[i] holds the data for node i and p[i] holds the index to the node previous to i, where i may or may not reside at position i-1. Like wise, N[i] holds the index to the next node in the list.

1.6. Abstract Data Type (ADT):

The design of a data structure involves more than just its organization. You also need to plan for the way the data will be accessed and processed – that is, how the data will be interpreted actually, non-contiguous structures – including lists, tree and graphs – can be implemented either contiguously or non- contiguously like wise, the structures that are normally treated as contiguously - arrays and structures – can also be implemented non-contiguously.

The notion of a data structure in the abstract needs to be treated differently from what ever is used to implement the structure. The abstract notion of a data structure is defined in terms of the operations we plan to perform on the data.

Considering both the organization of data and the expected operations on the data, leads to the notion of an abstract data type. An *abstract data type* is a theoretical construct that consists of data as well as the operations to be performed on the data while hiding implementation.

For example, a stack is a typical abstract data type. Items stored in a stack can only be added and removed in certain order – the last item added is the first item removed. We call these operations, pushing and popping. In this definition, we haven't specified how items are stored on the stack, or how the items are pushed and popped. We have only specified the valid operations that can be performed.

For example, if we want to read a file, we wrote the code to read the physical file device. That is, we may have to write the same code over and over again. So we created what is known

today as an ADT. We wrote the code to read a file and placed it in a library for a programmer to use.

As another example, the code to read from a keyboard is an ADT. It has a data structure, character and set of operations that can be used to read that data structure.

To be made useful, an abstract data type (such as stack) has to be implemented and this is where data structure comes into play. For instance, we might choose the simple data structure of an array to represent the stack, and then define the appropriate indexing operations to perform pushing and popping.

1.9. Selecting a data structure to match the operation:

The most important process in designing a problem involves choosing which data structure to use. The choice depends greatly on the type of operations you wish to perform.

Suppose we have an application that uses a sequence of objects, where one of the main operations is delete an object from the middle of the sequence. The code for this is as follows:

```
void delete (int *seg, int &n, int posn)
// delete the item at position from an array of n elements.
{
    if (n)
    {
        int i=posn;
        n--;
        while (i < n)
        {
            seq[i] =
            seq[i+1]; i++;
        }
    }
    return;
}
```

This function shifts towards the front all elements that follow the element at position *posn*. This shifting involves data movement that, for integer elements, which is too costly. However, suppose the array stores larger objects, and lots of them. In this case, the overhead for moving data becomes high. The problem is that, in a contiguous structure, such as an array the logical ordering (the ordering that we wish to interpret our elements to have) is the same as the physical ordering (the ordering that the elements actually have in memory).

If we choose non-contiguous representation, however we can separate the logical ordering from the physical ordering and thus change one without affecting the other. For example, if we store our collection of elements using a double-linked list (with previous and next pointers), we can do the deletion without moving the elements, instead, we just modify the pointers in each node. The code using double linked list is as follows:

```
void delete (node * beg, int posn)
//delete the item at posn from a list of elements.
{
    int i = posn;
    node *q = beg;
    while (i && q)
    {
```

```

        i--;
        q = q -> next;
    }

    if (q)
    { /* not at end of list, so detach P by making previous and
       next nodes point to each other
       */ node *p = q -> prev;
        node *n = q ->
        next; if (p)
            p -> next = n;
        if (n)
            n -> prev = P;
    }
    return;
}

```

The process of detecting a node from a list is independent of the type of data stored in the node, and can be accomplished with some pointer manipulation as illustrated in figure below:

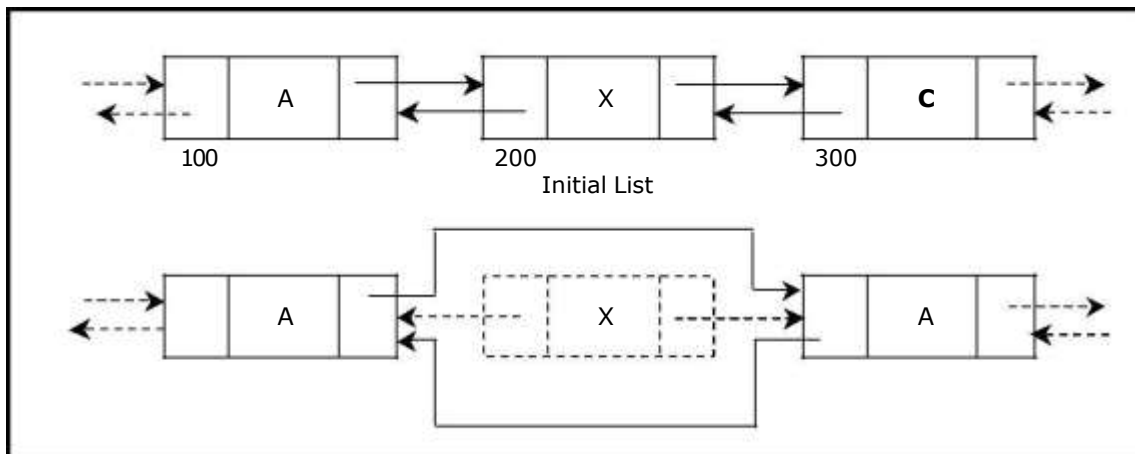


Figure 1.6 Detaching a node from a list

Since very little data is moved during this process, the deletion using linked lists will often be faster than when arrays are used.

It may seem that linked lists are superior to arrays. But is that always true? There are trade offs. Our linked lists yield faster deletions, but they take up more space because they require two extra pointers per element.

1.6. Algorithm

An **algorithm** is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent an algorithm using pseudo language that is a combination of the constructs of a programming language together with informal English statements.

1.8. Practical Algorithm design issues:

Choosing an efficient algorithm or data structure is just one part of the design process. Next, will look at some design issues that are broader in scope. There are three basic design goals that we should strive for in a program:

2. Try to save time (Time complexity).
3. Try to save space (Space complexity).
4. Try to have face.

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to —save face! by preventing the program from locking up or generating reams of garbled data.

1.8. Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the **TIME COMPLEXITY** of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

1.8. Classification of Algorithms

If n is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1 Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, $\log n$ is a doubled whenever n doubles, $\log n$ increases by a constant, but $\log n$ does not double until n increases to n^2 .

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

n. log n This running time arises for algorithms but solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple-nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as —brute-force— solutions to problems. Whenever n doubles, the running time squares.

1.11. Complexity of Algorithms

The complexity of an algorithm M is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size n . Complexity shall refer to the running time of the algorithm.

The function $f(n)$, gives the running time of an algorithm, depends not only on the size n of the input data but also on the particular data. The complexity function $f(n)$ for certain cases are:

1. Best Case : The minimum possible value of $f(n)$ is called the best case.
2. Average Case : The expected value of $f(n)$.
3. Worst Case : The maximum value of $f(n)$ for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

1.10. Rate of Growth

Big-Oh (O), Big-Omega (Ω), Big-Theta (Θ) and Little-Oh

2. $T(n) = O(f(n))$, (pronounced order of or big oh), says that the growth rate of $T(n)$ is less than or equal (\leq) that of $f(n)$
3. $T(n) = \Omega(g(n))$ (pronounced omega), says that the growth rate of $T(n)$ is greater than or equal to (\geq) that of $g(n)$
4. $T(n) = \Theta(h(n))$ (pronounced theta), says that the growth rate of $T(n)$ equals (=) the growth rate of $h(n)$ [if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$]
5. $T(n) = o(p(n))$ (pronounced little oh), says that the growth rate of $T(n)$ is less than the growth rate of $p(n)$ [if $T(n) = O(p(n))$ and $T(n) \neq \Theta(p(n))$].

Some Examples:

$$\begin{aligned}2n^2 + 5n - 6 &= O(2^n) \\2n^2 + 5n - 6 &= O(n^3) \\2n^2 + 5n - 6 &= O(n^2)\end{aligned}$$

$$2n^2 + 5n - 6 \neq O(n)$$

$$\begin{aligned}2n^2 + 5n - 6 &\neq \Omega(2^n) \\2n^2 + 5n - 6 &\neq \Omega(n^3) \\2n^2 + 5n - 6 &= \Omega(n^2)\end{aligned}$$

$$2n^2 + 5n - 6 = \Omega(n)$$

$$\begin{aligned}2n^2 + 5n - 6 &\neq \Theta(2^n) \\2n^2 + 5n - 6 &\neq \Theta(n^3) \\2n^2 + 5n - 6 &= \Theta(n^2)\end{aligned}$$

$$2n^2 + 5n - 6 \neq \Theta(n)$$

$$\begin{aligned}2n^2 + 5n - 6 &= o(2^n) \\2n^2 + 5n - 6 &= o(n^3) \\2n^2 + 5n - 6 &\neq o(n^2)\end{aligned}$$

$$2n^2 + 5n - 6 \neq o(n)$$

1.11. Analyzing Algorithms

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity $f(n)$ of M increases as n increases. It is usually the rate of increase of $f(n)$ we want to examine. This is usually done by comparing $f(n)$ with some standard functions. The most common computing times are:

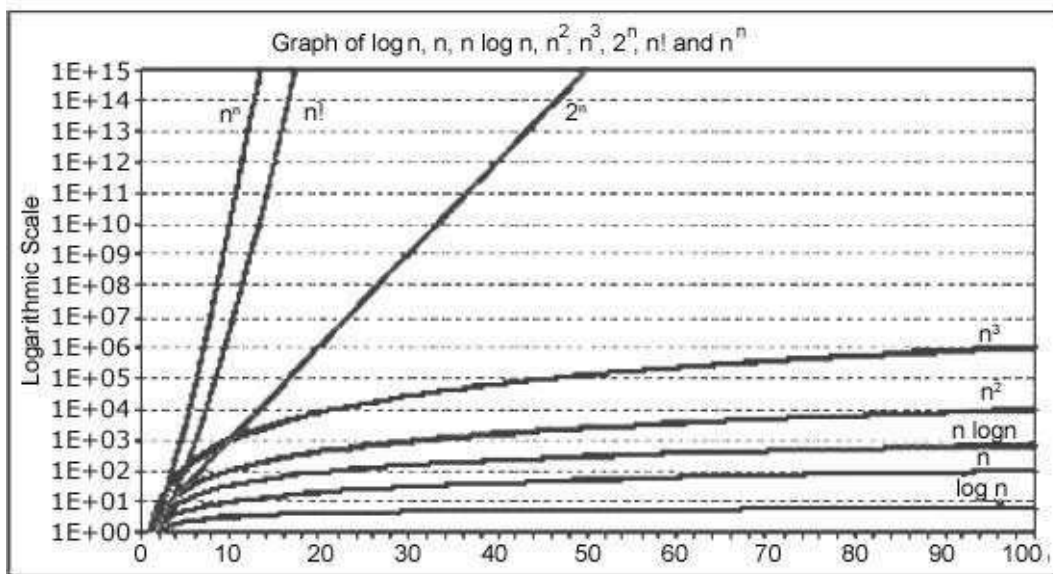
$$O(1), O(\log_2 n), O(n), O(n \cdot \log_2 n), O(n^2), O(n^3), O(2^n), n! \text{ and } n^n$$

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

S.No	$\log n$	n	$n \cdot \log n$	n^2	n^3	2^n
1	0	1	1	1	1	2
2	1	2	2	4	8	4
3	2	4	8	16	64	16
4	3	8	24	64	512	256
5	4	16	64	256	4096	65536

Graph of $\log n, n, n \log n, n^2, n^3, 2^n, n!$ and n^n



$O(\log n)$ does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low-order terms while computing a Big-Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

One way to compare the function $f(n)$ with these standard function is to use the functional O notation, suppose $f(n)$ and $g(n)$ are functions defined on the positive integers with the property that $f(n)$ is bounded by some multiple $g(n)$ for almost all n . Then,

$$f(n) = O(g(n))$$

Which is read as — $f(n)$ is of order $g(n)$ ‖. For example, the order of complexity for:

- Linear search is $O(n)$
- Binary search is $O(\log n)$
- Bubble sort is $O(n^2)$
- Quick sort is $O(n \log n)$

For example, if the first program takes $100n^2$ milliseconds. While the second taken $5n^3$ milliseconds. Then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5n^3/100n^2 = n/20$$

for inputs $n < 20$, the program with running time $5n^3$ will be faster those the one with running time $100n^2$.

Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as n gets large, the ratio of the running times, which is $n/20$, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as $O(n)$ or $O(n \log n)$ are always better.

Exercises

2. Define algorithm.
3. State the various steps in developing algorithms?
4. State the properties of algorithms.
5. Define efficiency of an algorithm?
6. State the various methods to estimate the efficiency of an algorithm.
7. Define time complexity of an algorithm?
8. Define worst case of an algorithm.
9. Define average case of an algorithm.
10. Define best case of an algorithm.
11. Mention the various spaces utilized by a program.

13. Define space complexity of an algorithm.
14. State the different memory spaces occupied by an algorithm.

Multiple Choice Questions

1. _____ is a step-by-step recipe for solving an instance of problem. [A]
A. Algorithm B. Complexity
C. Pseudocode D. Analysis

2. _____ is used to describe the algorithm, in less formal language. [C]
A. Cannot be defined B. Natural Language
C. Pseudocode D. None

3. _____ of an algorithm is the amount of time (or the number of steps) [D]
needed by a program to complete its task.
A. Space Complexity B. Dynamic Programming
C. Divide and Conquer D. Time Complexity

4. _____ of a program is the amount of memory used at once by the [C]
algorithm until it completes its execution.
A. Divide and Conquer B. Time Complexity
C. Space Complexity D. Dynamic Programming

5. _____ is used to define the worst-case running time of an algorithm. [A]
A. Big-Oh notation B. Cannot be defined
C. Complexity D. Analysis

Chapter 4

LINKED LISTS

In this chapter, the list data structure is presented. This structure can be used as the basis for the implementation of other data structures (stacks, queues etc.). The basic linked list can be used without modification in many programs. However, some applications require enhancements to the linked list design. These enhancements fall into three broad categories and yield variations on linked lists that can be used in any combination: circular linked lists, double linked lists and lists with header nodes.

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast. The disadvantages of arrays are:

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Deleting an element from an array is not possible.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Here is a quick review of the terminology and rules of pointers. The linked list code will depend on the following functions:

malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request. It is defined by:

```
void *malloc (number_of_bytes)
```

Since a void * is returned the C standard states that this pointer can be converted to any type. For example,

```
char *cp;  
cp = (char *) malloc (100);
```

Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example,

```
int *ip;  
ip = (int *) malloc (100*sizeof(int));
```

free() is the opposite of `malloc()`, which de-allocates memory. The argument to `free()` is a pointer to a block of memory in the heap — a pointer which was obtained by a `malloc()` function. The syntax is:

```
free (ptr);
```

The advantage of `free()` is simply memory management when we no longer need a block.

Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

7. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
8. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
9. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
10. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

It consumes more space because every node requires a additional pointer to store address of the next node.

Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

$$P(x) = a_0 X^n + a_1 X^{n-1} + \dots + a_{n-1} X + a_n$$

2. Represent very large numbers and operations of the large number such as addition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs.
4. Implement the symbol table in compiler construction

Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the ~~start~~ node.

A single linked list is shown in figure 3.2.1.

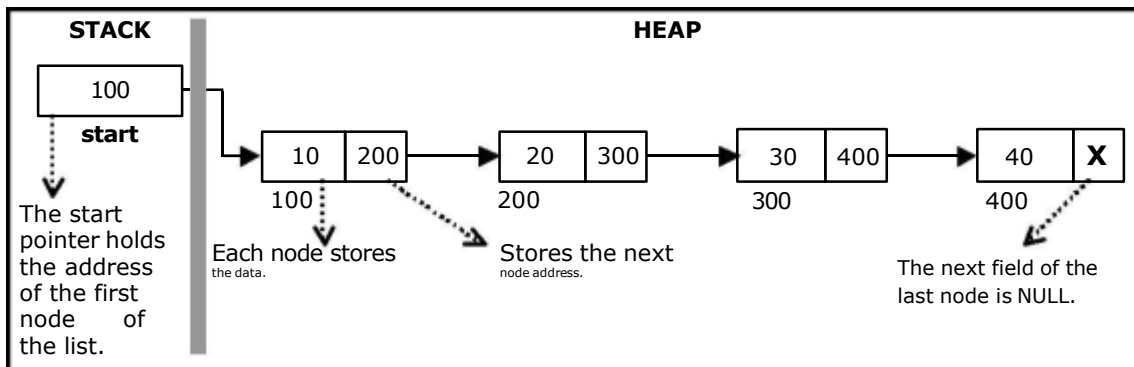


Figure 3.2.1. Single Linked List

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

Implementation of Single Linked List:

Before writing the code to build the above list, we need to create a **start** node, used to create and access other nodes in the linked list. The following structure definition will do (see figure 3.2.2):

- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
- Initialise the start pointer to be NULL.

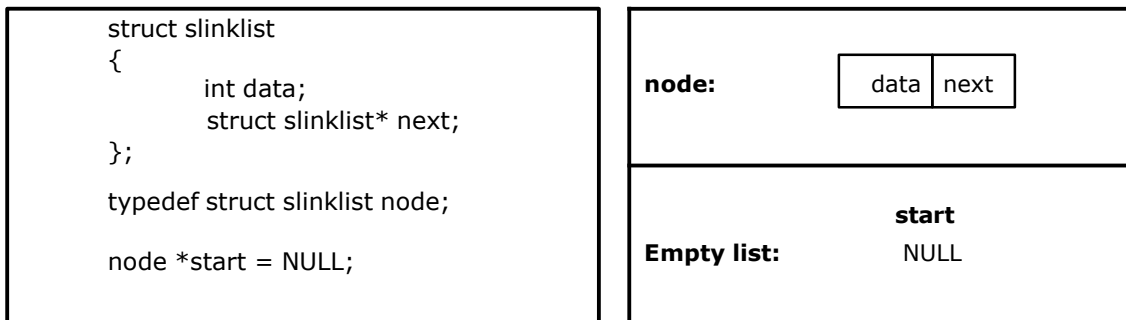


Figure 3.2.2. Structure definition, single link node and empty list

The basic operations in a single linked list are:

5. Creation.
6. Insertion.
7. Deletion.
8. Traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 3.2.3 illustrates the creation of a node for single linked list.

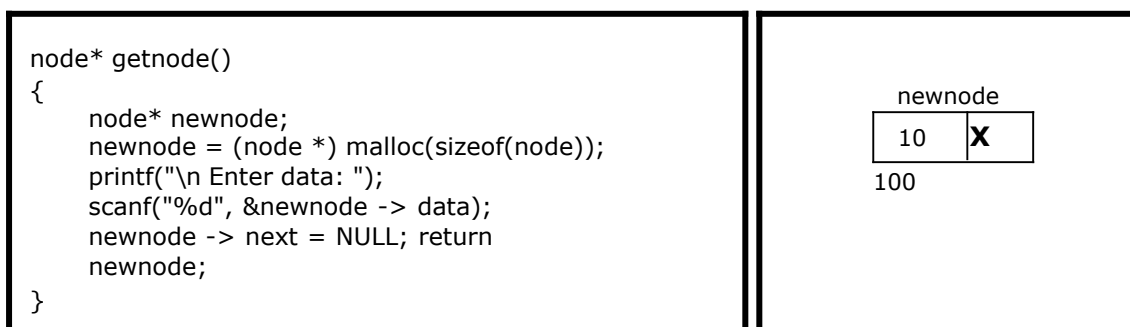


Figure 3.2.3. new node with a value of 10

Creating a Singly Linked List with 'n' number of nodes:

The following steps are to be followed to create n number of nodes:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty, assign new node as start.
`start = newnode;`
- If the list is not empty, follow the steps given below:
 - The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.
 - The start pointer is made to point the new node by assigning the address of the new node.
- Repeat the above steps n times.

Figure 3.2.4 shows 4 items in a single linked list stored at different locations in memory.

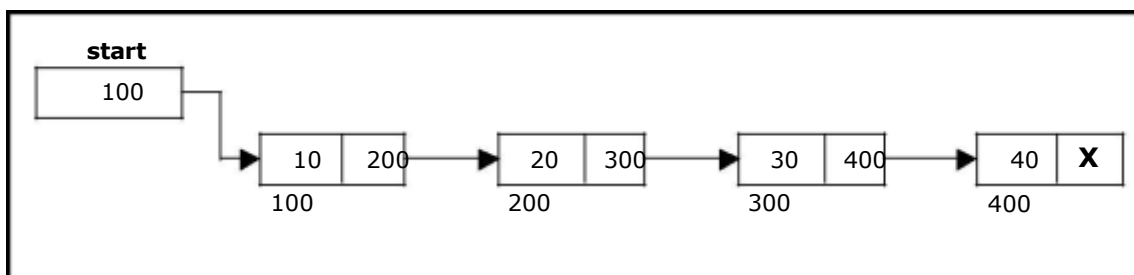


Figure 3.2.4. Single Linked List with 4 nodes

The function `createlist()`, is used to create n number of nodes:

```
void createlist(int n)
{
    int i;
    node * new node;
    node * temp;
    for(i = 0; i < n; i++)
    {
        new node = getnode();
        if(start == NULL)
        {
            start = new node;
        }
        else
        {
            temp = start;
            while(temp -> next != NULL)
                temp = temp -> next;
            temp -> next = new node;
        }
    }
}
```

Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- # Inserting a node at the beginning.
- # Inserting a node at the end.
- # Inserting a node at intermediate position.

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using `getnode()`. `newnode = getnode();`
- If the list is empty then `start = newnode`.
- If the list is not empty, follow the steps given below: `newnode -> next = start;`
`start = newnode;`

Figure 3.2.5 shows inserting a node into the single linked list at the beginning.

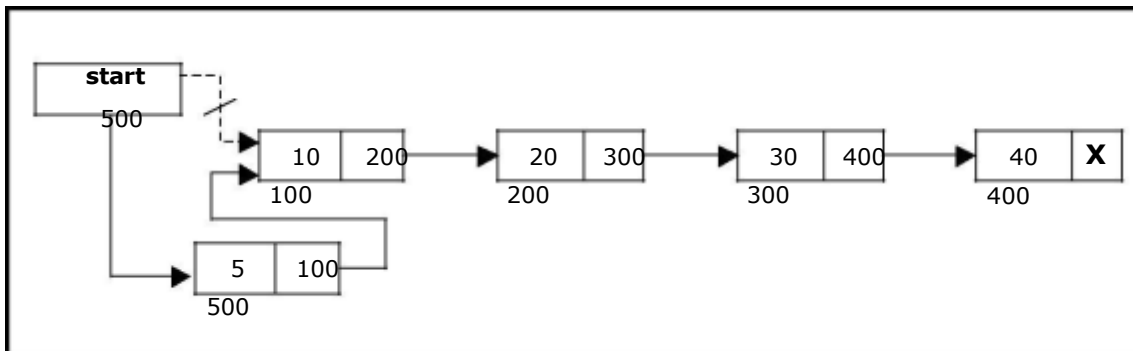


Figure 3.2.5. Inserting a node at the beginning

The function `insert_at_beg()`, is used for inserting a node at the beginning

```
void insert_at_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        newnode -> next =
        start; start = newnode;
    }
}
```

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- # Get the new node using `getnode()`
`newnode = getnode();`
- # If the list is empty then `start = newnode`.
- # If the list is not empty follow the steps given below:
`temp = start;`
`while(temp -> next != NULL)`
`temp = temp -> next;`
`temp -> next = newnode;`

Figure 3.2.6 shows inserting a node into the single linked list at the end.

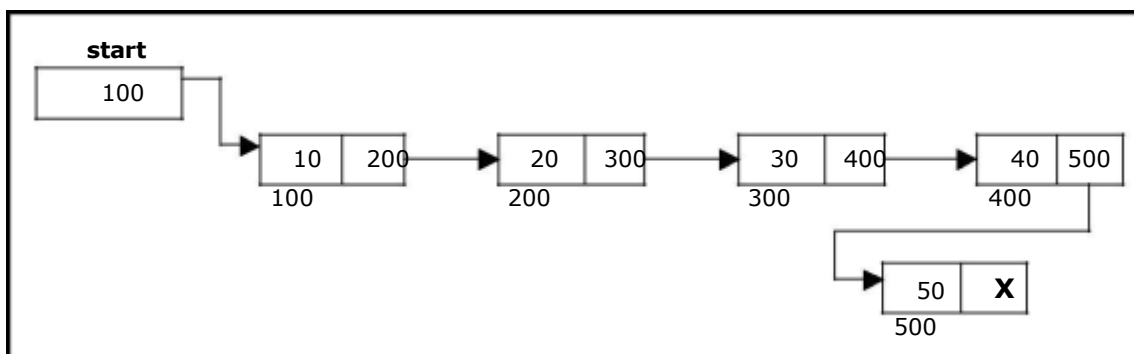


Figure 3.2.6. Inserting a node at the end.

The function `insert_at_end()`, is used for inserting a node at the end.

```
void insert_at_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        temp = start;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newnode;
    }
}
```

Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using `getnode()`.
`newnode = getnode();`

- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below: prev -> next = newnode;
newnode -> next = temp;

12 Let the intermediate position be 3.

Figure 3.2.7 shows inserting a node into the single linked list at a specified intermediate position other than beginning and end.

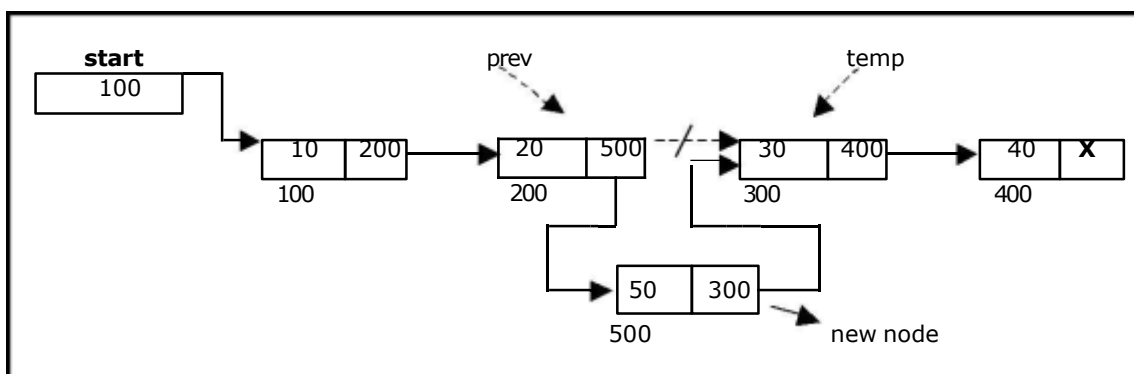


Figure 3.2.7. Inserting a node at an intermediate position.

The function insert_at_mid(), is used for inserting a node in the intermediate position.

```
void insert_at_mid()
{
    node *newnode, *temp, *prev;
    int pos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos > 1 && pos < nodectr)
    {
        temp = prev = start;
        while(ctr < pos)
        {
            prev = temp;
            temp = temp ->
            next; ctr++;
        }
        prev -> next = newnode;
        newnode -> next = temp;
    }
    else
    {
        printf("position %d is not a middle position", pos);
    }
}
```

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display `_Empty List` message.
- If the list is not empty, follow the steps given below:
`temp = start;`
`start = start -> next;`
`free(temp);`

Figure 3.2.8 shows deleting a node at the beginning of a single linked list.

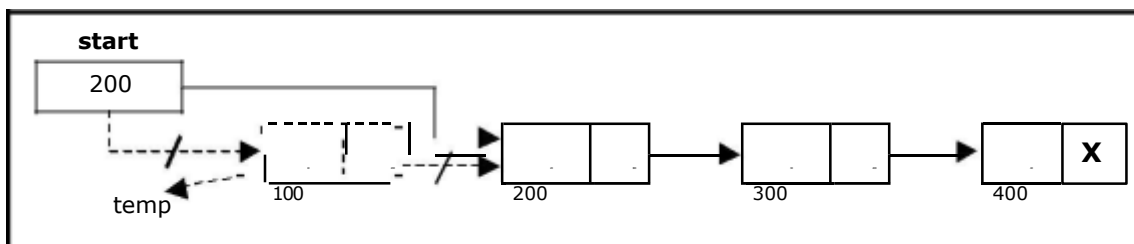


Figure 3.2.8. Deleting a node at the beginning.

The function `delete_at_beg()`, is used for deleting the first node in the list.

```
void delete_at_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes are exist..");
        return ;
    }
    else
    {
        temp = start;
        start = temp -> next;
        free(temp);
        printf("\n Node deleted ");
    }
}
```

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

1. If list is empty then display `_Empty List` message.
2. If the list is not empty, follow the steps given

```
below: temp = prev = start;
while(temp -> next != NULL)
{
    prev = temp;
    temp = temp -> next;
}
prev -> next = NULL;
free(temp);
```

Figure 3.2.9 shows deleting a node at the end of a single linked list.

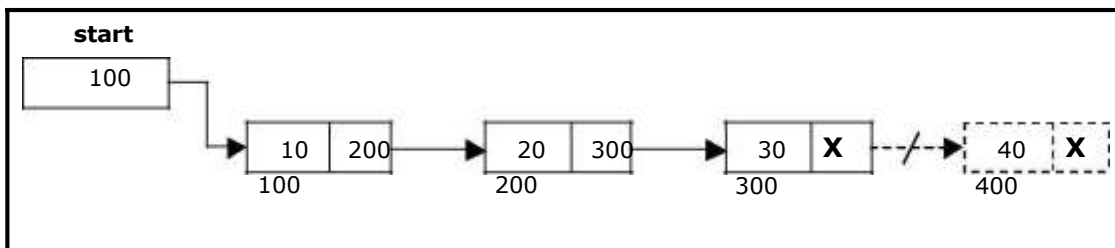


Figure 3.2.9. Deleting a node at the end.

The function `delete_at_last()`, is used for deleting the last node in the list.

```
void delete_at_last()
{
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n Empty
List.."); return ;
    }
    else
    {
        temp = start;
        prev = start;
        while(temp -> next != NULL)
        {
            prev = temp;
            temp = temp -> next;
        }
        prev -> next = NULL;
        free(temp);
        printf("\n Node deleted ");
    }
}
```

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

```
# If list is empty then display '_Empty List' message

# If the list is not empty, follow the steps given below.
  if(pos > 1 && pos < nodelctr)
  {
    temp = prev = start;
    ctr = 1;
    while(ctr < pos)
    {
      prev = temp;
      temp = temp -> next;
      ctr++;
    }
    prev -> next = temp -> next;
    free(temp);
    printf("\n node deleted..");
  }
```

Figure 3.2.10 shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

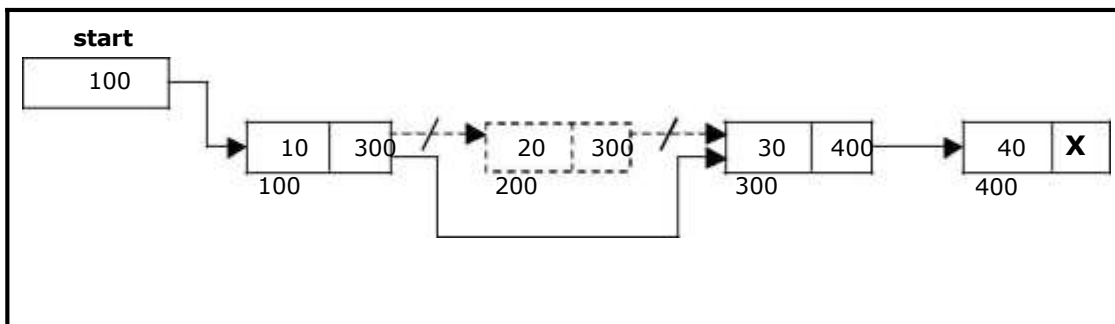


Figure 3.2.10. Deleting a node at an intermediate position.

The function `delete_at_mid()`, is used for deleting the intermediate node in the list.

```
void delete_at_mid()
{
  int ctr = 1, pos,
  nodelctr; node *temp,
  *prev; if(start == NULL)
  {
    printf("\n Empty
    List.."); return ;
  }
  else
  {
    printf("\n Enter position of node to delete: ");
    scanf("%d", &pos);
    nodelctr = countnode(start);
    if(pos > nodelctr)
    {
      printf("\nThis node doesnot exist");
    }
  }
```

```

        if(pos > 1 && pos < nodectr)
        {
            temp = prev = start;
            while(ctr < pos)
            {
                prev = temp;
                temp = temp ->
                    next; ctr ++;
            }
            prev -> next = temp -> next;
            free(temp);
            printf("\n Node deleted..");
        }
        else
        {
            printf("\n Invalid position..");
            getch();
        }
    }
}

```

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

2. Assign the address of start pointer to a temp pointer.
3. Display the information from the data field of each node.

The function *traverse()* is used for traversing and displaying the information stored in the list from left to right.

```

void traverse()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right):
\n"); if(start == NULL )
        printf("\n Empty List");
    else
    {
        while (temp != NULL)
        {
            printf("%d ->", temp ->
                data); temp = temp -> next;
        }
    }
    printf("X");
}

```

Alternatively there is another way to traverse and display the information. That is in reverse order. The function *rev_traverse()*, is used for traversing and displaying the information stored in the list from right to left.

```

void rev_traverse(node *st)
{
    if(st == NULL)
    {
        return;
    }
    else
    {
        rev_traverse(st -> next);
        printf("%d ->", st -> data);
    }
}

```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list using *recursion*.

```

int countnode(node *st)
{
    if(st == NULL)
        return 0;
    else
        return(1 + countnode(st -> next));
}

```

Source Code for the Implementation of Single Linked List:

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

struct slinklist
{
    int data;
    struct slinklist *next;
};

typedef struct slinklist node;

node *start = NULL;
int menu()
{
    int ch;
    clrscr();
    printf("\n 1.Create a list ");
    printf("\n-----");
    printf("\n 2.Insert a node at beginning ");
    printf("\n 3.Insert a node at end");
    printf("\n 4.Insert a node at middle");
    printf("\n-----");
    printf("\n 5.Delete a node from beginning");
    printf("\n 6.Delete a node from Last");
    printf("\n 7.Delete a node from Middle");
    printf("\n-----");
    printf("\n 8.Traverse the list (Left to Right)");
    printf("\n 9.Traverse the list (Right to Left)");
}

```

```

        printf("\n.....");
        printf("\n 10. Count nodes ");
        printf("\n 11. Exit ");
        printf("\n\n Enter your choice: ");
        scanf("%d",&ch);
        return ch;
    }

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> next = NULL; return
    newnode;
}

int countnode(node *ptr)
{
    int count=0;
    while(ptr != NULL)
    {
        count++;
        ptr = ptr -> next;
    }
    return (count);
}

void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    for(i = 0; i < n; i++)
    {
        newnode = getnode();
        if(start == NULL)
        {
            start = newnode;
        }
        else
        {
            temp = start;
            while(temp -> next != NULL)
                temp = temp -> next;
            temp -> next = newnode;
        }
    }
}

void traverse()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right): \n");
    if(start == NULL)
    {
        printf("\n Empty List");
        return;
    }
    else
    {

```

```

        while(temp != NULL)
        {
            printf("%d-->", temp ->
                data); temp = temp -> next;
        }
    }
    printf(" X ");
}

```

```

void rev_traverse(node *start)
{
    if(start == NULL)
    {
        return;
    }
    else
    {
        rev_traverse(start -> next);
        printf("%d -->", start -> data);
    }
}

```

```

void insert_at_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        newnode -> next =
            start; start = newnode;
    }
}

```

```

void insert_at_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        temp = start;
        while(temp -> next != NULL)
        {
            temp = temp -> next;
        }
        temp -> next = newnode;
    }
}

```

```

void insert_at_mid()
{
    node *newnode, *temp, *prev;
    int pos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
}

```



```

    if(pos > 1 && pos < nodectr)
    {
        temp = prev = start;
        while(ctr < pos)
        {
            prev = temp;
            temp = temp ->
                next; ctr++;
        }
        prev -> next = newnode;
        newnode -> next = temp;
    }
    else
        printf("position %d is not a middle position", pos);
}

```

```

void delete_at_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes are exist..");
        return ;
    }
    else
    {
        temp = start;
        start = temp -> next;
        free(temp);
        printf("\n Node deleted ");
    }
}

```

```

void delete_at_last()
{
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n Empty
List.."); return ;
    }
    else
    {
        temp = start;
        prev = start;
        while(temp -> next != NULL)
        {
            prev = temp;
            temp = temp -> next;
        }
        prev -> next = NULL;
        free(temp);
        printf("\n Node deleted ");
    }
}

```

```

void delete_at_mid()
{
    int ctr = 1, pos,
    nodectr; node *temp,
    *prev; if(start == NULL)
    {
        printf("\n Empty List..");
    }
}

```

```

        return ;
    }
    else
    {
        printf("\n Enter position of node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > nodectr)
        {
            printf("\nThis node doesnot exist");
        }
        if(pos > 1 && pos < nodectr)
        {
            temp = prev = start;
            while(ctr < pos)
            {
                prev = temp;
                temp = temp ->
                next; ctr ++;
            }
            prev -> next = temp -> next;
            free(temp);
            printf("\n Node deleted..");
        }
        else
        {
            printf("\n Invalid position..");
            getch();
        }
    }
}

void main(void)
{
    int ch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch(ch)
        {
            case 1:
                if(start == NULL)
                {
                    printf("\n Number of nodes you want to create: ");
                    scanf("%d", &n);
                    createlist(n);
                    printf("\n List created..");
                }
                else
                    printf("\n List is already created..");
                    break;
            case 2:
                insert_at_beg();
                break;
            case 3:
                insert_at_end();
                break;
            case 4:
                insert_at_mid();
                break;
        }
    }
}

```

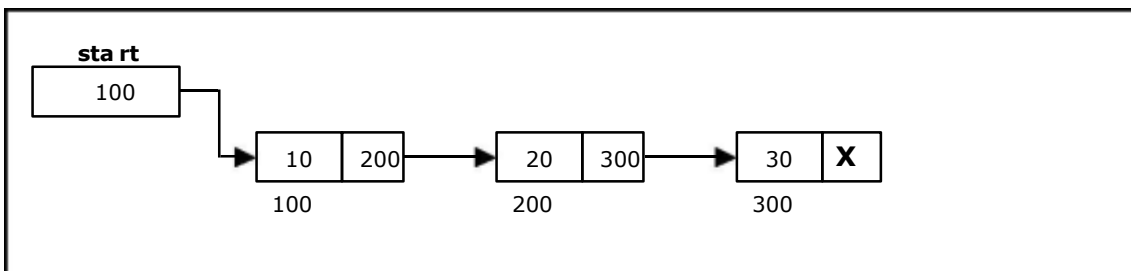
```

case 5:
    delete_at_beg();
    break;
case 6:
    delete_at_last();
    break;
case 7:
    delete_at_mid();
    break;
case 8:
    traverse();
    break;
case 9:
    printf("\n The contents of List (Right to Left): \n");
    rev_traverse(start);
    printf(" X ");
    break;
case 10:
    printf("\n No of nodes : %d ", countnode(start));
    break;
case 11 :
    exit(0);
}
getch();
}
}

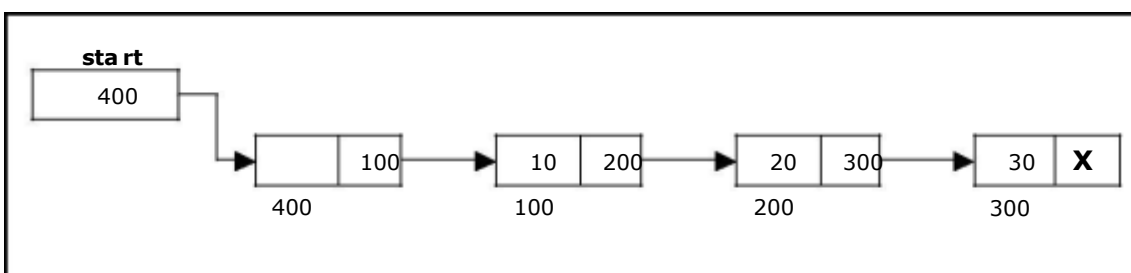
```

Using a header node:

A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list. For example, the picture below shows how the list with data 10, 20 and 30 would be represented using a linked list without and with a header node:



Single Linke d List w it ho ut a he a der no de



Single Linke d List w it h he a der no de

Note that if your linked lists do include a header node, there is no need for the special case code given above for the *remove* operation; node *n* can never be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node *n*.

A double linked list is shown in figure 3.3.1.

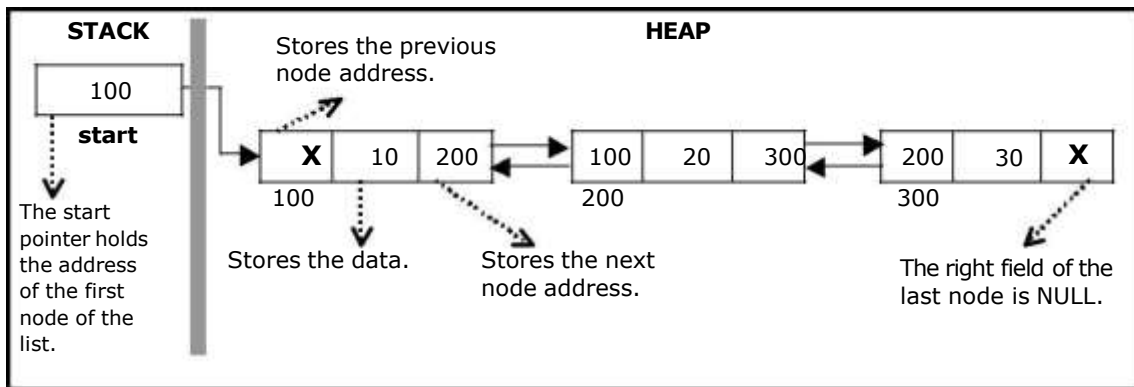


Figure 3.3.1. Double Linked List

The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

The following code gives the structure definition:

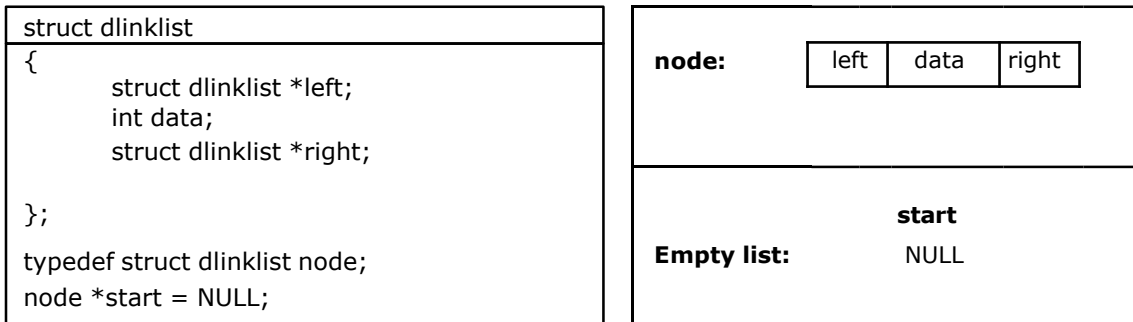


Figure 3.4.1. Structure definition, double link node and empty list

Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL (see figure 3.2.2).

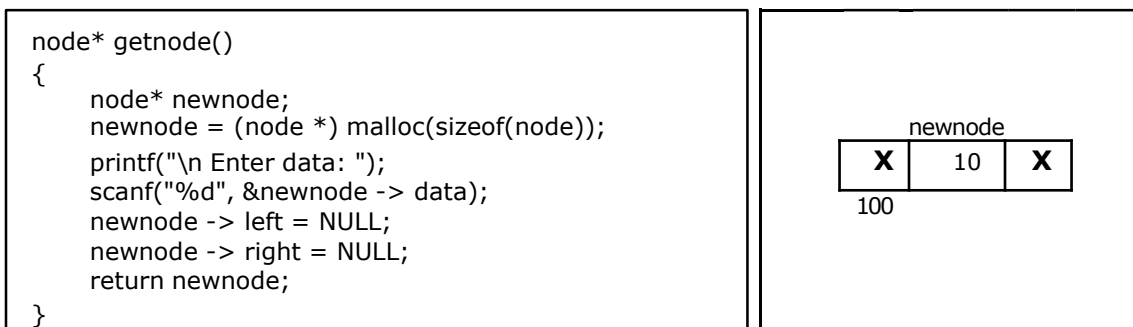


Figure 3.4.2. new node with a value of 10

Creating a Double Linked List with 'n' number of nodes:

The following steps are to be followed to create n number of nodes:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty then `start = newnode`.
- If the list is not empty, follow the steps given below:
 - The left field of the new node is made to point the previous node.
 - The previous nodes right field must be assigned with address of the new node.
- Repeat the above steps n times.

The function `createlist()`, is used to create n number of nodes:

```
void createlist(int n)
{
    int i;
    node * new node;
    node * temp;
    for(i = 0; i < n; i++)
    {
        new node = getnode();
        if(start == NULL)
        {
            start = new node;
        }
        else
        {
            temp = start;
            while(temp -> right)
                temp = temp -> right;
            temp -> right = new node; new
            node -> left = temp;
        }
    }
}
```

Figure 3.4.3 shows 3 items in a double linked list stored at different locations.

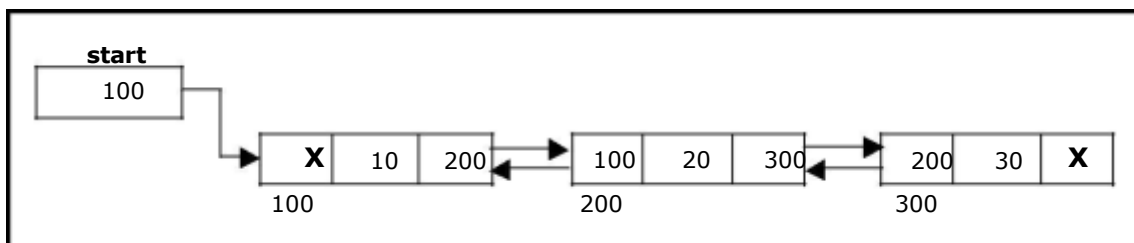


Figure 3.4.3. Double Linked List with 3 nodes

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

Get the new node using `getnode()`.

```
newnode=getnode();
```

- If the list is empty then $start = newnode$.
- If the list is not empty, follow the steps given below:

```
newnode -> right = start;  
start -> left = newnode;  
start = newnode;
```

The function `dbl_insert_beg()`, is used for inserting a node at the beginning. Figure 3.4.4 shows inserting a node into the double linked list at the beginning.

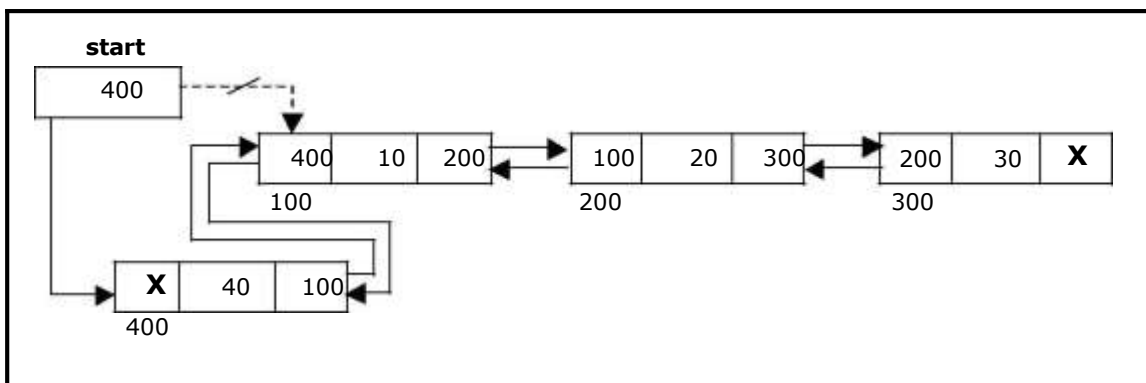


Figure 3.4.4. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using
`getnode() newnode=getnode();`
- If the list is empty then $start = newnode$.
- If the list is not empty follow the steps given

```
below: temp = start;  
while(temp -> right != NULL)  
    temp = temp -> right;  
temp -> right = newnode;  
newnode -> left = temp;
```

The function `dbl_insert_end()`, is used for inserting a node at the end. Figure 3.4.5 shows inserting a node into the double linked list at the end.

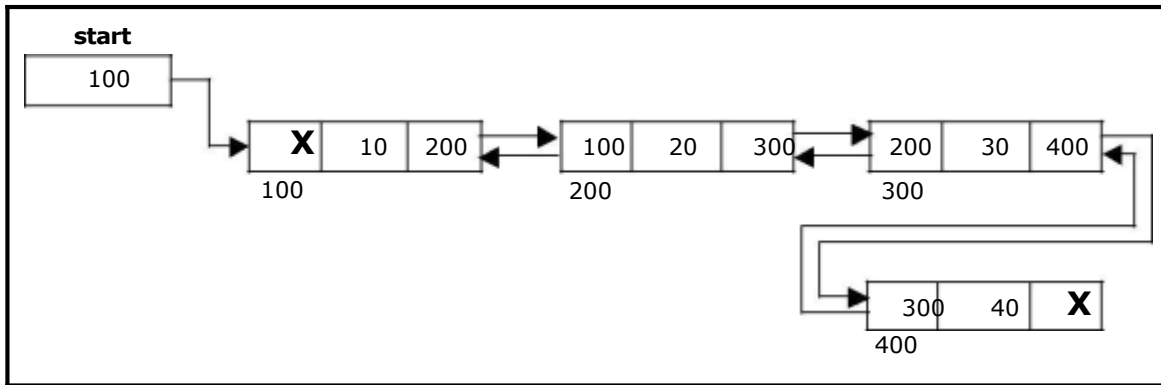


Figure 3.4.5. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using `getnode()`.

```
newnode=getnode();
```

- # Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by `countnode()` function.
- # Store the starting address (which is in `start` pointer) in `temp` and `prev` pointers. Then traverse the `temp` pointer upto the specified position followed by `prev` pointer.
- # After reaching the specified position, follow the steps given below:

```
newnode -> left = temp; newnode
-> right = temp -> right; temp ->
right -> left = newnode; temp ->
right = newnode;
```

The function `dbl_insert_mid()`, is used for inserting a node in the intermediate position. Figure 3.4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

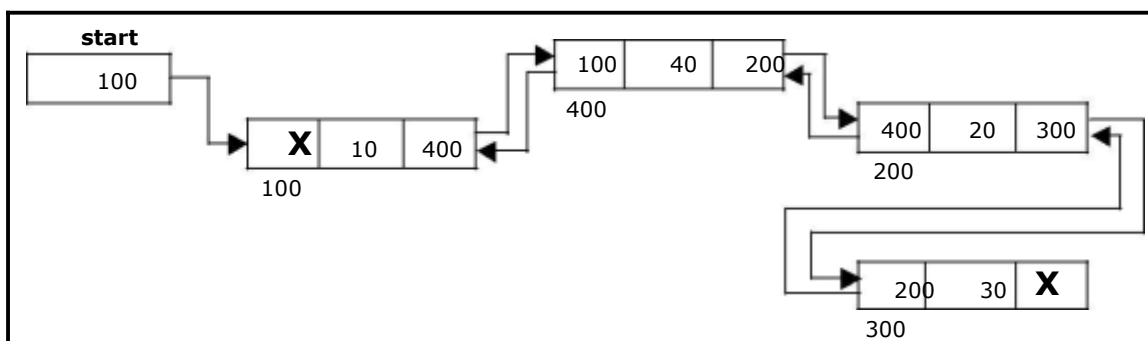


Figure 3.4.6. Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display `'_Empty List'` message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

The function `dbl_delete_beg()`, is used for deleting the first node in the list. Figure 3.4.6 shows deleting a node at the beginning of a double linked list.

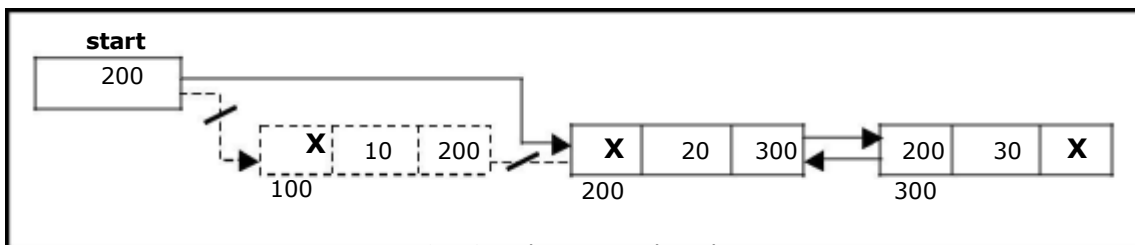


Figure 3.4.6. Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display `'_Empty List'` message
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
    temp = temp -> right;
}
temp -> left -> right = NULL;
free(temp);
```

The function `dbl_delete_last()`, is used for deleting the last node in the list. Figure 3.4.7 shows deleting a node at the end of a double linked list.

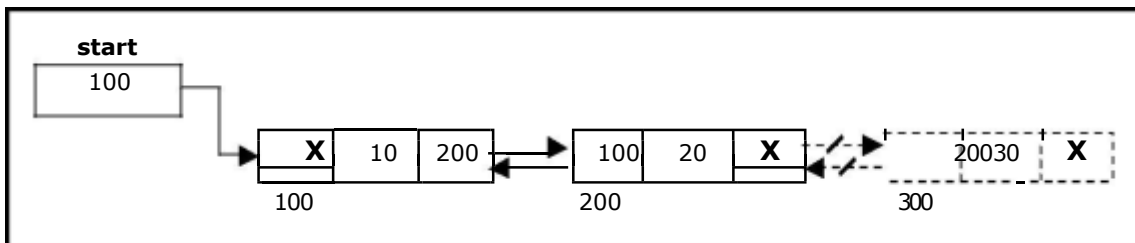


Figure 3.4.7. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

- 5 If list is empty then display `_Empty List` message.
- 6 If the list is not empty, follow the steps given below:
 - 40 Get the position of the node to delete.
 - 41 Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
 - 42 Then perform the following steps:

```
if(pos > 1 && pos <
nodectr)
{
    temp = start;
    i = 1;
    while(i < pos)
    {
        temp = temp -> right;
        i++;
    }
    temp -> right -> left = temp -> left;
    temp -> left -> right = temp -> right;
    free(temp);
    printf("\n node deleted..");
}
```

The function `delete_at_mid()`, is used for deleting the intermediate node in the list. Figure 3.4.8 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

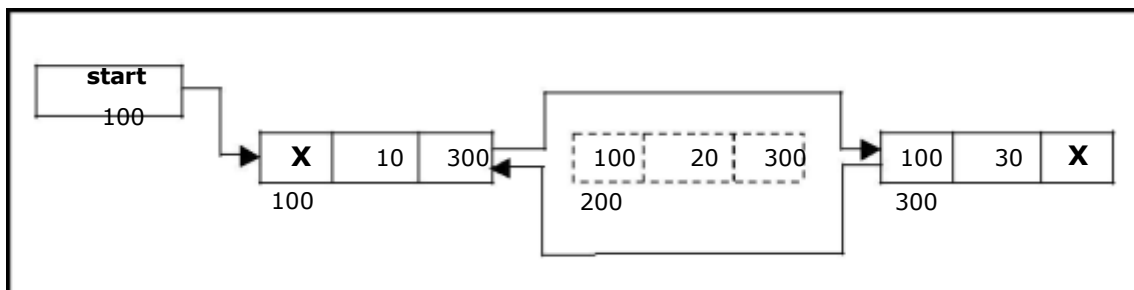


Figure 3.4.8 Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function `traverse_left_right()` is used for traversing and displaying the information stored in the list from left to right.

The following steps are followed, to traverse a list from left to right:

5. If list is empty then display `_Empty List` message.
6. If the list is not empty, follow the steps given below:

```

temp = start;
while(temp != NULL)
{
    print temp -> data;
    temp = temp -> right;
}

```

Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left()* is used for traversing and displaying the information stored in the list from right to left. The following steps are followed, to traverse a list from right to left:

6. If list is empty then display `'_Empty List'` message.

7. If the list is not empty, follow the steps given below:

```

temp = start;
while(temp -> right != NULL)
    temp = temp -> right;
while(temp != NULL)
{
    print temp -> data;
    temp = temp -> left;
}

```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list (using recursion).

```

int countnodes(node *start)
{
    if(start == NULL)
        return 0;
    else
        return(1 + countnodes(start ->right));
}

```

A Complete Source Code for the Implementation of Double Linked List:

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct dlinklist
{
    struct dlinklist *left;
    int data;
    struct dlinklist *right;
};

typedef struct dlinklist node;
node *start = NULL;

```

```

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> left = NULL;
    newnode -> right = NULL;
    return newnode;
}

int countnode(node *start)
{
    if(start == NULL)
        return 0;
    else
        return 1 + countnode(start -> right);
}

int menu()
{
    int ch;
    clrscr();
    printf("\n 1.Create");
    printf("\n-----");
    printf("\n 2. Insert a node at beginning ");
    printf("\n 3. Insert a node at end");
    printf("\n 4. Insert a node at middle");
    printf("\n.....");
    printf("\n 5. Delete a node from beginning");
    printf("\n 6. Delete a node from Last");
    printf("\n 7. Delete a node from Middle");
    printf("\n.....");
    printf("\n 8. Traverse the list from Left to Right ");
    printf("\n 9. Traverse the list from Right to Left ");
    printf("\n.....");
    printf("\n 10.Count the Number of nodes in the list");
    printf("\n 11.Exit ");
    printf("\n\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}

void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    for(i = 0; i < n; i++)
    {
        newnode = getnode();
        if(start == NULL)
            start = newnode;
        else
        {
            temp = start;
            while(temp -> right)
                temp = temp -> right;
            temp -> right = newnode;
            newnode -> left = temp;
        }
    }
}

```

```

void traverse_left_to_right()
{
    node *temp;
    temp = start;
    printf("\n The contents of List:
"); if(start == NULL )
        printf("\n Empty List");
    else
    {
        while(temp != NULL)
        {
            printf("\t %d ", temp -> data);
            temp = temp -> right;
        }
    }
}

void traverse_right_to_left()
{
    node *temp;
    temp = start;
    printf("\n The contents of List:
"); if(start == NULL)
        printf("\n Empty List");
    else
    {
        while(temp -> right != NULL)
            temp = temp -> right;
    }
    while(temp != NULL)
    {
        printf("\t%d", temp ->
data); temp = temp -> left;
    }
}

void dll_insert_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
        start = newnode;
    else
    {
        newnode -> right = start;
        start -> left = newnode;
        start = newnode;
    }
}

void dll_insert_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
        start = newnode;
    else
    {
        temp = start;
        while(temp -> right != NULL)
            temp = temp -> right;
        temp -> right = newnode;
        newnode -> left = temp;
    }
}

```

```

void dll_insert_mid()
{
    node *newnode,*temp;
    int pos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos - nodectr >= 2)
    {
        printf("\n Position is out of range..");
        return;
    }
    if(pos > 1 && pos < nodectr)
    {
        temp = start;
        while(ctr < pos - 1)
        {
            temp = temp -> right;
            ctr++;
        }
        newnode -> left = temp; newnode
        -> right = temp -> right; temp ->
        right -> left = newnode; temp ->
        right = newnode;
    }
    else
        printf("position %d of list is not a middle position ", pos);
}

```

```

void dll_delete_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty
list"); getch();
        return ;
    }
    else
    {
        temp = start;
        start = start -> right;
        start -> left = NULL;
        free(temp);
    }
}

```

```

void dll_delete_last()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty
list"); getch();
        return ;
    }
    else
    {
        temp = start;
        while(temp -> right != NULL)

```

```

        temp = temp -> right;
        temp -> left -> right = NULL;
        free(temp);
        temp = NULL;
    }
}

void dll_delete_mid()
{
    int i = 0, pos, nodectr;
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty List");
        getch();
        return;
    }
    else
    {
        printf("\n Enter the position of the node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > nodectr)
        {
            printf("\nthis node does not exist"); getch();
            return;
        }
        if(pos > 1 && pos < nodectr)
        {
            temp = start; i = 1;
            while(i < pos)
            {
                temp = temp -> right;
                i++;
            }
            temp -> right -> left = temp -> left;
            temp -> left -> right = temp -> right;
            free(temp);
            printf("\n node deleted..");
        }
        else
        {
            printf("\n It is not a middle position..");
            getch();
        }
    }
}

void main(void)
{
    int ch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch( ch)
        {
            case 1 :
                printf("\n Enter Number of nodes to create: ");
                scanf("%d", &n);
                createlist(n);

```

```

        printf("\n List
        created.."); break;
    case 2 :
        dll_insert_beg();
        break;
    case 3 :
        dll_insert_end();
        break;
    case 4 :
        dll_insert_mid();
        break;
    case 5 :
        dll_delete_beg();
        break;
    case 6 : dll_delete_last();
        break;

    case 7 :
        dll_delete_mid();
        break;
    case 8 :
        traverse_left_to_right();
        break;
    case 9 :
        traverse_right_to_left();
        break;

    case 10 :
        printf("\n Number of nodes: %d", countnode(start));
        break;
    case 11:
        exit(0);
    }
    getch();
}
}

```

Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure 3.6.1.

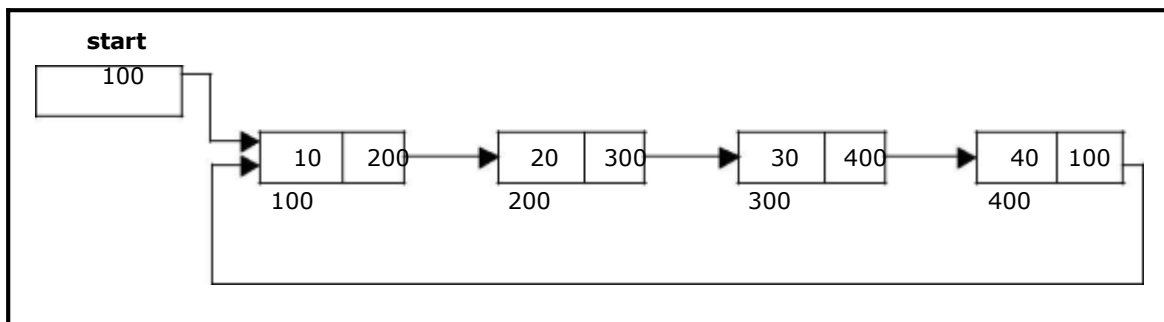


Figure 3.6.1. Circular Single Linked List

The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create n number of nodes:

10. Get the new node using
`getnode(). newnode =
getnode();`
11. If the list is empty, assign new node as
`start. start = newnode;`
12. If the list is not empty, follow the steps given
below:
`temp = start;
while(temp -> next != NULL)
temp = temp -> next;
temp -> next = newnode;`
 - Repeat the above steps n times.
 - `newnode -> next = start;`

The function `createlist()`, is used to create n number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

- Get the new node using
`getnode(). newnode =
getnode();`
- If the list is empty, assign new node as start.
`start = newnode; newnode
-> next = start;`
- If the list is not empty, follow the steps given below:
`last = start;
while(last -> next != start)
last = last -> next;
newnode -> next =
start; start = newnode;
last -> next = start;`

The function `cll_insert_beg()`, is used for inserting a node at the beginning. Figure 3.6.2 shows inserting a node into the circular single linked list at the beginning.

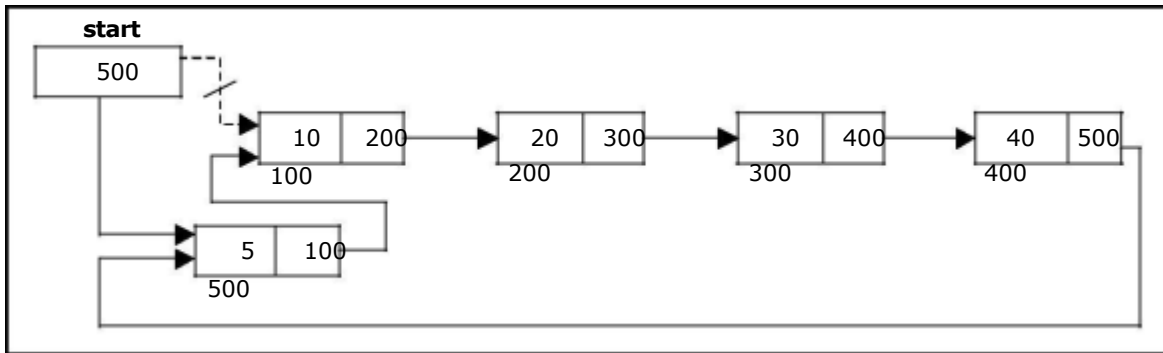


Figure 3.6.2. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using `getnode()`. `newnode =`
- If the list is empty, assign new node as start.
`start = newnode; newnode -> next = start;`
- If the list is not empty follow the steps given below:
`temp = start;`
`while(temp -> next != start)`
`temp = temp -> next;`
`temp -> next = newnode;`
`newnode -> next = start;`

The function `cll_insert_end()`, is used for inserting a node at the end.

Figure 3.6.3 shows inserting a node into the circular single linked list at the end.

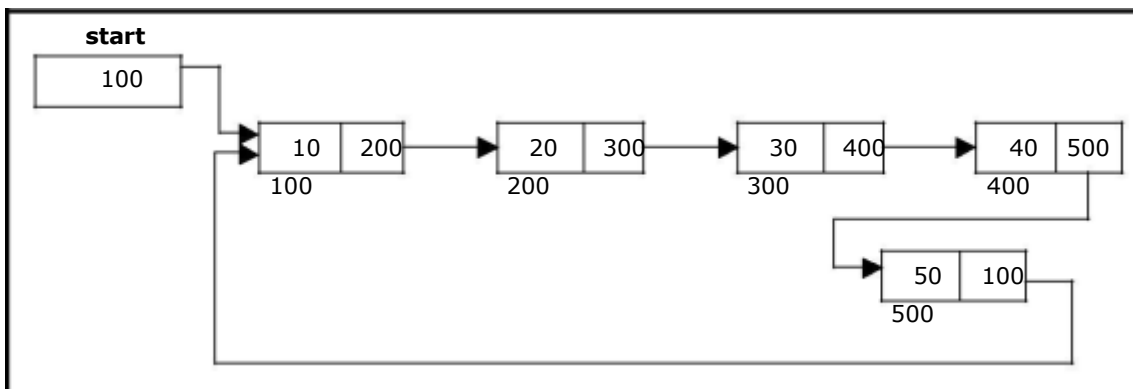


Figure 3.6.3 Inserting a node at the end.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

1. If the list is empty, display a message `_Empty List``.
2. If the list is not empty, follow the steps given

```
below: last = temp = start;  
while(last -> next != start)  
    last = last -> next;  
start = start -> next;  
last -> next = start;
```

4.8.2. After deleting the node, if the list is empty then `start = NULL`.

The function `cll_delete_beg()`, is used for deleting the first node in the list. Figure 3.6.4 shows deleting a node at the beginning of a circular single linked list.

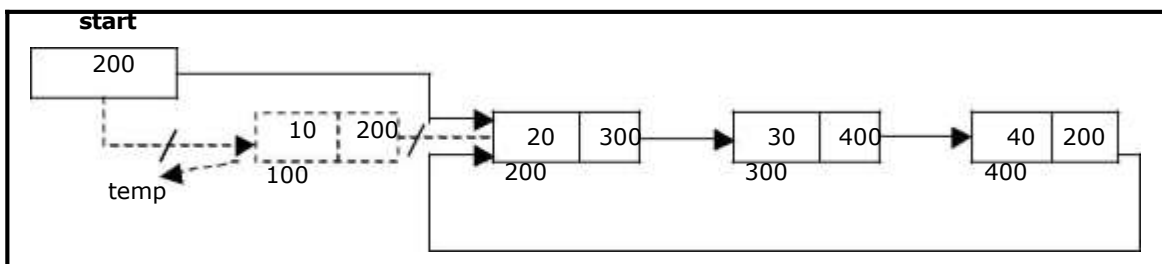


Figure 3.6.4. Deleting a node at beginning.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- # If the list is empty, display a message `_Empty List``.
- # If the list is not empty, follow the steps given below:

```
temp = start;  
prev = start;  
while(temp -> next != start)  
{  
    prev = temp;  
    temp = temp -> next;  
}  
prev -> next = start;
```

4.9. After deleting the node, if the list is empty then `start = NULL`.

The function `cll_delete_last()`, is used for deleting the last node in the list.

Figure 3.6.5 shows deleting a node at the end of a circular single linked list.

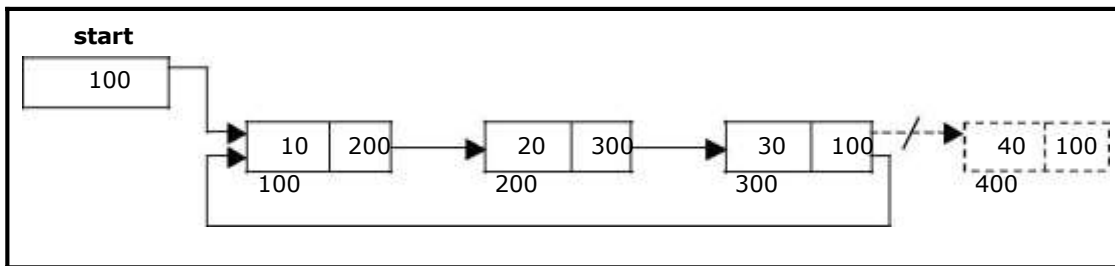


Figure 3.6.5. Deleting a node at the end.

Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

- If list is empty then display `_Empty List` message.
- If the list is not empty, follow the steps given

```
below: temp = start;
do
{
    printf("%d ", temp -> data);
    temp = temp -> next;
} while(temp != start);
```

- **Source Code for Circular Single Linked List:**

```
include <stdio.h>
include <conio.h>
include <stdlib.h>

struct cslinklist
{
    int data;
    struct cslinklist *next;
};

typedef struct cslinklist node;

node *start = NULL;

int nodectr;

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> next = NULL; return
    newnode;
}
```

```

int menu()
{
    int ch;
    clrscr();
    printf("\n 1. Create a list ");
    printf("\n\n-----");
    printf("\n 2. Insert a node at beginning ");
    printf("\n 3. Insert a node at end");
    printf("\n 4. Insert a node at middle");
    printf("\n\n.....");
    printf("\n 5. Delete a node from beginning");
    printf("\n 6. Delete a node from Last");
    printf("\n 7. Delete a node from Middle");
    printf("\n\n.....");
    printf("\n 8. Display the list");
    printf("\n 9. Exit");
    printf("\n\n.....");
    printf("\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}

void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    nodectr = n;
    for(i = 0; i < n ; i++)
    {
        newnode = getnode();
        if(start == NULL)
        {
            start = newnode;
        }
        else
        {
            temp = start;
            while(temp -> next != NULL)
                temp = temp -> next;
            temp -> next = newnode;
        }
    }
    newnode ->next = start;    /* last node is pointing to starting node */
}

void display()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right): ");
    if(start == NULL )
        printf("\n Empty List");
    else
    {
        do
        {
            printf("\t %d ", temp -> data);
            temp = temp -> next;
        } while(temp !=
start); printf(" X ");
    }
}

```

```

void cl_insert_beg()
{
    node *newnode, *last;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode; newnode
        -> next = start;
    }
    else
    {
        last = start;
        while(last -> next != start)
            last = last -> next;
        newnode -> next =
        start; start = newnode;
        last -> next = start;
    }
    printf("\n Node inserted at beginning..");
    nodectr++;
}

```

```

void cl_insert_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL )
    {
        start = newnode; newnode
        -> next = start;
    }
    else
    {
        temp = start;
        while(temp -> next != start)
            temp = temp -> next;
        temp -> next = newnode;
        newnode -> next = start;
    }
    printf("\n Node inserted at end..");
    nodectr++;
}

```

```

void cl_insert_mid()
{
    node *newnode, *temp, *prev;
    int i, pos ;
    newnode = getnode(); printf("\n
    Enter the position: ");
    scanf("%d", &pos);
    if(pos > 1 && pos < nodectr)
    {
        temp =
        start; prev =
        temp; i = 1;
        while(i < pos)
        {
            prev = temp;
            temp = temp ->
            next; i++;
        }
        prev -> next = newnode;
        newnode -> next = temp;
    }
}

```

```

        nodectr++;
        printf("\n Node inserted at middle..");
    }
    else
    {
        printf("position %d of list is not a middle position ", pos);
    }
}

void cll_delete_beg()
{
    node *temp, *last;
    if(start == NULL)
    {
        printf("\n No nodes
        exist.."); getch();
        return ;
    }
    else
    {
        last = temp = start;
        while(last -> next != start)
            last = last -> next;
        start = start -> next;
        last -> next = start;
        free(temp);
        nodectr--;
        printf("\n Node deleted..");
        if(nodectr == 0)
            start = NULL;
    }
}

void cll_delete_last()
{
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n No nodes
        exist.."); getch();
        return ;
    }
    else
    {
        temp = start;
        prev = start;
        while(temp -> next != start)
        {
            prev = temp;
            temp = temp -> next;
        }
        prev -> next = start;
        free(temp); nodectr-
        -;
        if(nodectr == 0) start
            = NULL;
        printf("\n Node deleted..");
    }
}

```

```

void cll_delete_mid()
{
    int i = 0, pos;
    node *temp, *prev;

    if(start == NULL)
    {
        printf("\n No nodes
        exist.."); getch();
        return ;
    }
    else
    {
        printf("\n Which node to delete: ");
        scanf("%d", &pos);
        if(pos > nodectr)
        {
            printf("\nThis node does not
            exist"); getch();
            return;
        }
        if(pos > 1 && pos < nodectr)
        {
            temp=start;
            prev = start;
            i = 0;
            while(i < pos - 1)
            {
                prev = temp;
                temp = temp -> next ;
                i++;
            }
            prev -> next = temp -> next;
            free(temp);
            nodectr--;
            printf("\n Node Deleted..");
        }
        else
        {
            printf("\n It is not a middle position..");
            getch();
        }
    }
}

void main(void)
{
    int result;
    int ch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch(ch)
        {
            case 1 :
                if(start == NULL)
                {
                    printf("\n Enter Number of nodes to create: ");
                    scanf("%d", &n);
                    createlist(n);
                    printf("\nList created..");
                }
        }
    }
}

```

```

else
    printf("\n List is already Exist..");
break;
case 2 :
    cll_insert_beg();
break;
case 3 :
    cll_insert_end();
break;
case 4 :
    cll_insert_mid();
break;
case 5 :
    cll_delete_beg();
break;
case 6 : cll_delete_last();
break;

case 7 :
    cll_delete_mid();
break;
case 8 :
    display();
break;
case 9 :
    exit(0);
}
getch();
}
}

```

Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node. A circular double linked list is shown in figure 3.8.1.

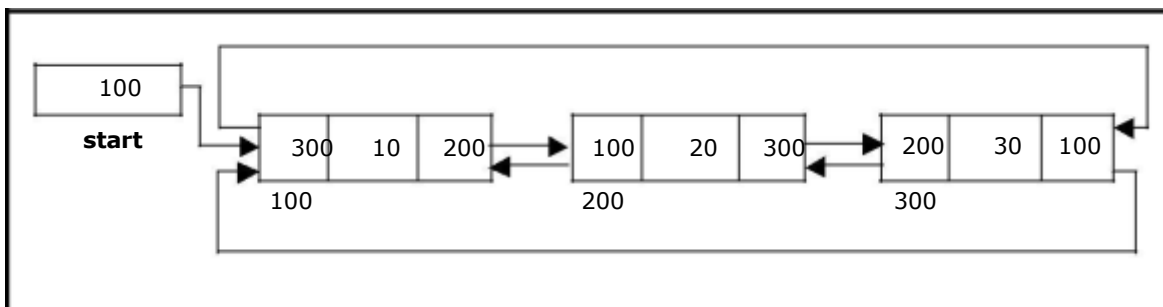


Figure 3.8.1. Circular Double Linked List

The basic operations in a circular double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a Circular Double Linked List with 'n' number of nodes:

The following steps are to be followed to create n number of nodes:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty, then do the following
`start = newnode;`
`newnode -> left = start;`
`newnode -> right = start;`
- If the list is not empty, follow the steps given below:
`newnode -> left = start -> left;`
`newnode -> right = start;`
`start -> left -> right = newnode;`
`start -> left = newnode;`
- Repeat the above steps n times.

The function `cdll_createlist()`, is used to create n number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using `getnode()`.
`newnode=getnode();`
- If the list is empty, then
`start = newnode;`
`newnode -> left = start;`
`newnode -> right = start;`
- If the list is not empty, follow the steps given below:
`newnode -> left = start -> left;`
`newnode -> right = start;`
`start -> left -> right = newnode;`
`start -> left = newnode;`
`start = newnode;`

The function `cdll_insert_beg()`, is used for inserting a node at the beginning. Figure 3.8.2 shows inserting a node into the circular double linked list at the beginning.

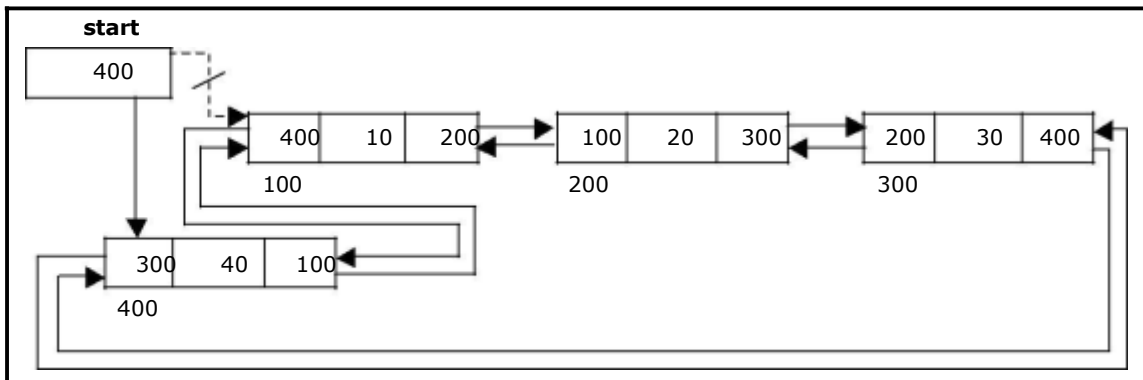


Figure 3.8.2. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using `getnode()`
`newnode=getnode();`
- If the list is empty, then
`start = newnode;`
`newnode -> left = start;`
`newnode -> right = start;`
- If the list is not empty follow the steps given below:
`newnode -> left = start -> left;`
`newnode -> right = start;`
`start -> left -> right = newnode;`
`start -> left = newnode;`

The function `cdll_insert_end()`, is used for inserting a node at the end. Figure 3.8.3 shows inserting a node into the circular linked list at the end.

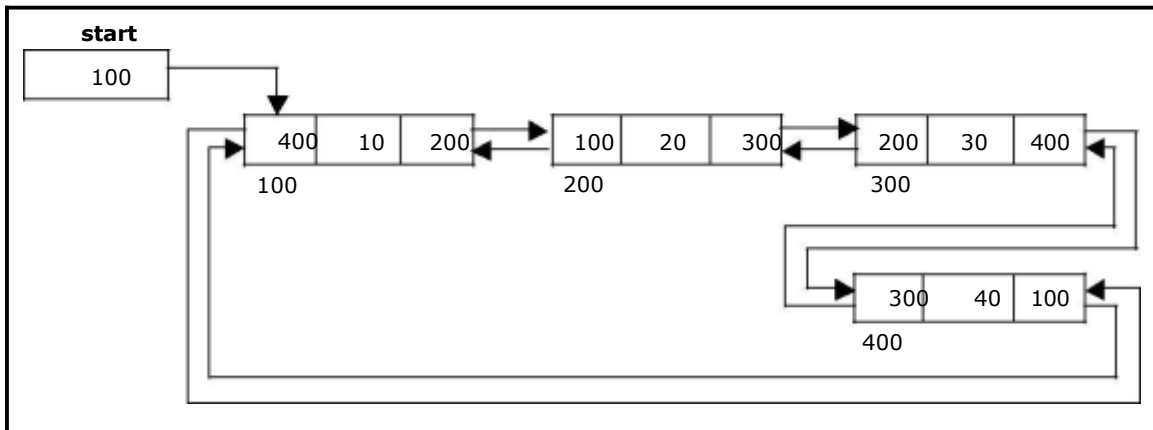


Figure 3.8.3. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using `getnode()`.
`newnode=getnode();`
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by `countnode()` function.
- Store the starting address (which is in `start` pointer) in `temp`. Then traverse the `temp` pointer upto the specified position.
- After reaching the specified position, follow the steps given below:
`newnode -> left = temp;`
`newnode -> right = temp -> right;`
`temp -> right -> left = newnode;`
`temp -> right = newnode;`
`nodectr++;`

The function `cdll_insert_mid()`, is used for inserting a node in the intermediate position. Figure 3.8.4 shows inserting a node into the circular double linked list at a specified intermediate position other than beginning and end.

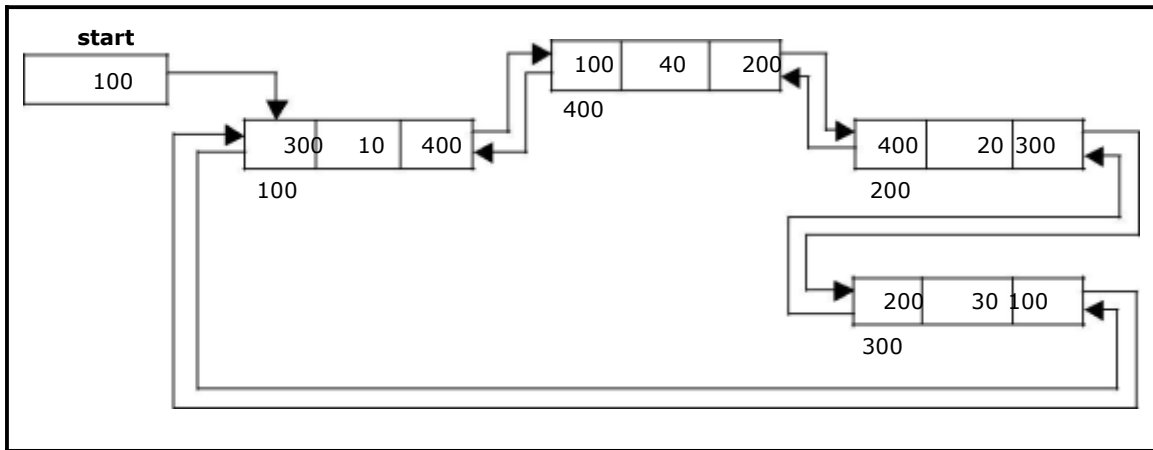


Figure 3.8.4. Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display `'_Empty List'` message.
- If the list is not empty, follow the steps given

```
below: temp = start;
start = start -> right;
temp -> left -> right = start;
start -> left = temp -> left;
```

The function `cdll_delete_beg()`, is used for deleting the first node in the list. Figure 3.8.5 shows deleting a node at the beginning of a circular double linked list.

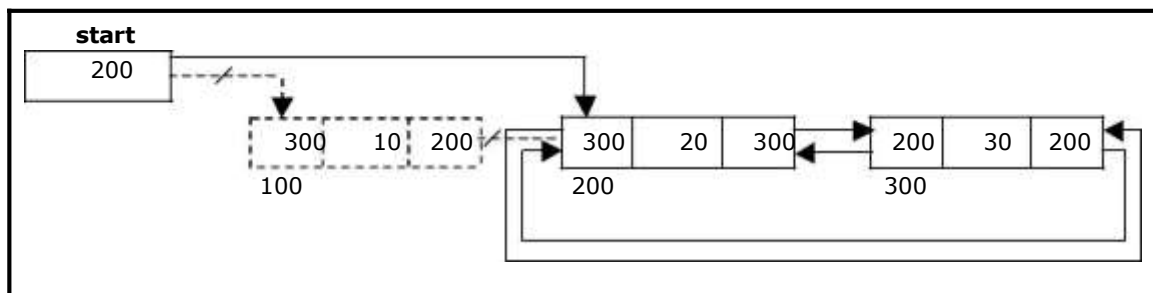


Figure 3.8.5. Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display `'_Empty List'` message
- If the list is not empty, follow the steps given below:

```

temp = start;
while(temp -> right != start)
{
    temp = temp -> right;
}
temp -> left -> right = temp -> right;
temp -> right -> left = temp -> left;

```

The function `cdll_delete_last()`, is used for deleting the last node in the list. Figure 3.8.6 shows deleting a node at the end of a circular double linked list.

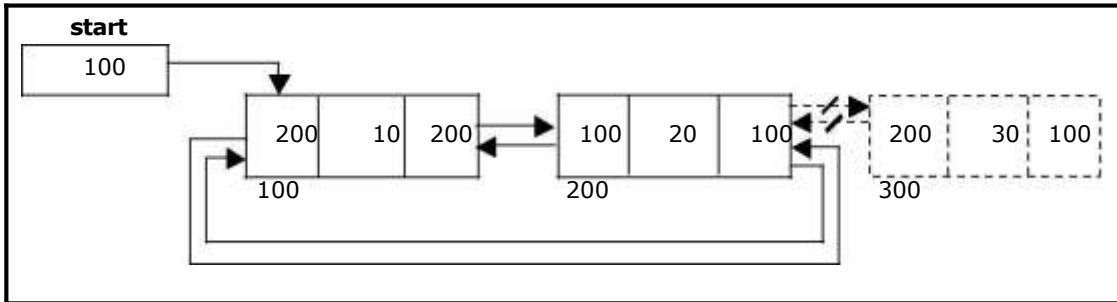


Figure 3.8.6. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display `‘_Empty List’` message.
- If the list is not empty, follow the steps given below:
 - Get the position of the node to delete.
 - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
 - Then perform the following steps:

```

if(pos > 1 && pos < nodectr)
{
    temp = start;
    i = 1;
    while(i < pos)
    {
        temp = temp -> right ;
        i++;
    }
    temp -> right -> left = temp -> left;
    temp -> left -> right = temp -> right;
    free(temp);
    printf("\n node deleted..");
    nodectr--;
}

```

The function `cdll_delete_mid()`, is used for deleting the intermediate node in the list.

Figure 3.8.7 shows deleting a node at a specified intermediate position other than beginning and end from a circular double linked list.

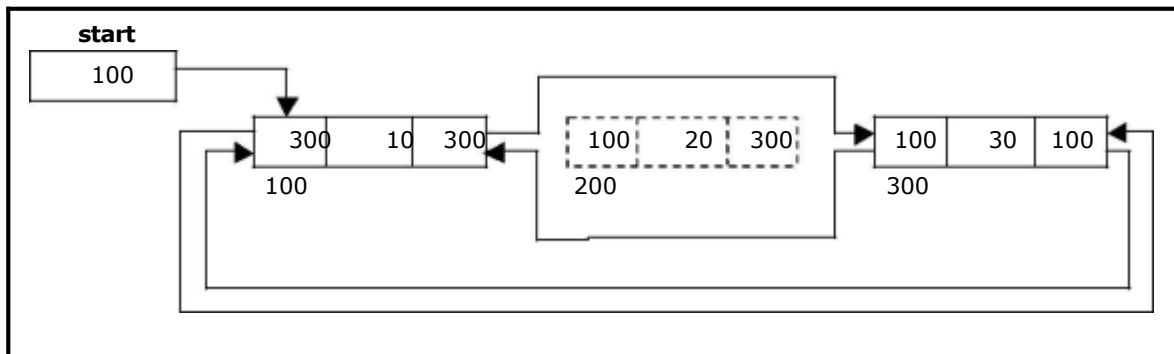


Figure 3.8.7. Deleting a node at an intermediate position

Traversing a circular double linked list from left to right:

The following steps are followed, to traverse a list from left to right:

- E. If list is empty then display `__Empty List'` message.
- F. If the list is not empty, follow the steps given below:


```
temp = start;
Print temp -> data;
temp = temp -> right;
while(temp != start)
{
    print temp -> data;
    temp = temp -> right;
}
```

The function `cdll_display_left_right()`, is used for traversing from left to right.

Traversing a circular double linked list from right to left:

The following steps are followed, to traverse a list from right to left:

- E. If list is empty then display `__Empty List'` message.
- F. If the list is not empty, follow the steps given below:


```
temp = start;
do
{
    temp = temp -> left;
    print temp -> data;
} while(temp != start);
```

The function `cdll_display_right_left()`, is used for traversing from right to left.

Source Code for Circular Double Linked List:

```
include <stdio.h>
include <stdlib.h>
include <conio.h>
```

```

struct cdlinklist
{
    struct cdlinklist
    *left; int data;
    struct cdlinklist *right;
};

typedef struct cdlinklist
node; node *start = NULL;
int nodectr;

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> left = NULL;
    newnode -> right = NULL;
    return newnode;
}

int menu()
{
    int ch;
    clrscr();
    printf("\n 1. Create ");
    printf("\n\n.....");
    printf("\n 2. Insert a node at Beginning");
    printf("\n 3. Insert a node at End");
    printf("\n 4. Insert a node at Middle");
    printf("\n\n.....");
    printf("\n 5. Delete a node from Beginning");
    printf("\n 6. Delete a node from End");
    printf("\n 7. Delete a node from Middle");
    printf("\n\n.....");
    printf("\n 8. Display the list from Left to Right");
    printf("\n 9. Display the list from Right to Left");
    printf("\n 10.Exit");
    printf("\n\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}

void cdll_createlist(int n)
{
    int i;
    node *newnode, *temp;
    if(start == NULL)
    {
        nodectr = n;
        for(i = 0; i < n; i++)
        {
            newnode = getnode();
            if(start == NULL)
            {
                start = newnode;
                newnode -> left = start;
                newnode ->right = start;
            }
            else
            {
                newnode -> left = start -> left;

```

```

newnode -> right = start; start
-> left->right = newnode;
start -> left = newnode;
    }
}
else
    printf("\n List already exists..");
}

void cdll_display_left_right()
{
    node *temp;
    temp = start;
    if(start == NULL)
        printf("\n Empty List");
    else
    {
        printf("\n The contents of List:
"); printf(" %d ", temp -> data);
        temp = temp -> right;
        while(temp != start)
        {
            printf(" %d ", temp -> data);
            temp = temp -> right;
        }
    }
}

void cdll_display_right_left()
{
    node *temp;
    temp = start;
    if(start == NULL)
        printf("\n Empty List");
    else
    {
        printf("\n The contents of List:
"); do
        {
            temp = temp -> left;
            printf("\t%d", temp -> data);
        } while(temp != start);
    }
}

void cdll_insert_beg()
{
    node *newnode;
    newnode = getnode();
    nodectr++;
    if(start == NULL)
    {
        start = newnode;
        newnode -> left = start;
        newnode -> right = start;
    }
    else
    {
        newnode -> left = start -> left;
        newnode -> right = start;
        start -> left -> right = newnode;
        start -> left = newnode;
    }
}

```

```

        start = newnode;
    }
}

void cdll_insert_end()
{
    node *newnode,*temp;
    newnode = getnode();
    nodectr++;
    if(start == NULL)
    {
        start = newnode;
        newnode -> left = start;
        newnode -> right = start;
    }
    else
    {
        newnode -> left = start -> left;
        newnode -> right = start;
        start -> left -> right = newnode;
        start -> left = newnode;
    }
    printf("\n Node Inserted at End");
}

void cdll_insert_mid()
{
    node *newnode, *temp, *prev;
    int pos, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    if(pos - nodectr >= 2)
    {
        printf("\n Position is out of range..");
        return;
    }
    if(pos > 1 && pos <= nodectr)
    {
        temp = start;
        while(ctr < pos - 1)
        {
            temp = temp -> right;
            ctr++;
        }
        newnode -> left = temp; newnode
        -> right = temp -> right; temp ->
        right -> left = newnode; temp ->
        right = newnode; nodectr++;

        printf("\n Node Inserted at Middle.. ");
    }
    else
        printf("position %d of list is not a middle position", pos);
}

void cdll_delete_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes exist..");
    }
}

```



```

        getch();
        return ;
    }
    else
    {
        nodectr--;
        if(nodectr == 0)
        {
            free(start);
            start = NULL;
        }
        else
        {
            temp = start;
            start = start -> right;
            temp -> left -> right = start;
            start -> left = temp -> left;
            free(temp);
        }
        printf("\n Node deleted at Beginning..");
    }
}

void cdll_delete_last()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes
        exist.."); getch();
        return;
    }
    else
    {
        nodectr--;
        if(nodectr == 0)
        {
            free(start);
            start = NULL;
        }
        else
        {
            temp = start;
            while(temp -> right != start)
                temp = temp -> right;
            temp -> left -> right = temp -> right;
            temp -> right -> left = temp -> left;
            free(temp);
        }
        printf("\n Node deleted from end ");
    }
}

void cdll_delete_mid()
{
    int ctr = 1, pos;
    node *temp;
    if( start == NULL)
    {
        printf("\n No nodes
        exist.."); getch();
        return;
    }
}

```

```

else
{
    printf("\n Which node to delete: ");
    scanf("%d", &pos);
    if(pos > nodectr)
    {
        printf("\nThis node does not
        exist"); getch();
        return;
    }
    if(pos > 1 && pos < nodectr)
    {
        temp = start;
        while(ctr < pos)
        {
            temp = temp -> right ;
            ctr++;
        }
        temp -> right -> left = temp -> left;
        temp -> left -> right = temp -> right;
        free(temp);
        printf("\n node deleted..");
        nodectr--;
    }
    else
    {
        printf("\n It is not a middle position..");
        getch();
    }
}
}

void main(void)
{
    int ch,n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch( ch)
        {
            case 1 :
                printf("\n Enter Number of nodes to create: ");
                scanf("%d", &n);
                cdll_createlist(n);
                printf("\n List
                created.."); break;
            case 2 : cdll_insert_beg();
                break;

            case 3 : cdll_insert_end();
                break;

            case 4 :
                cdll_insert_mid();
                break;
            case 5 : cdll_delete_beg();
                break;

            case 6 :
                cdll_delete_last();
                break;
        }
    }
}

```

```

        case 7 :
            cdll_delete_mid();
            break;
        case 8 :
            cdll_display_left_right();
            break;
        case 9 :
            cdll_display_right_left();
            break;
        case 10:
            exit(0);
    }
    getch();
}
}

```

9. Comparison of Linked List Variations:

The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the *prev* fields as well as the *next* fields; the more fields that have to be maintained, the more chance there is for errors.

The major advantage of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node). Also, some applications lead naturally to circular list representations. For example, a computer network might best be modeled using a circular list.

10. Polynomials:

A polynomial is of the form: $\sum_{i=0}^n c_i x^i$

Where, c_i is the coefficient of the i^{th} term and
 n is the degree of the polynomial

Some examples are:

$$\begin{aligned}
 &5x^2 + 3x + 1 \\
 &12x^3 - 4x \\
 &5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0
 \end{aligned}$$

It is not necessary to write terms of the polynomials in decreasing order of degree. In other words the two polynomials $1 + x$ and $x + 1$ are equivalent.

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$ illustrates in figure 3.10.1.

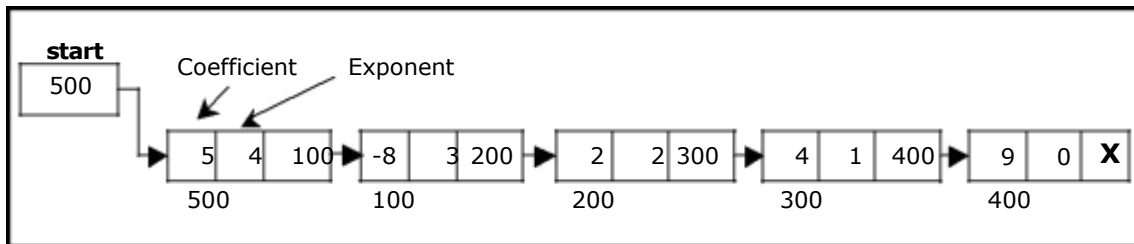


Figure 3.10.1. Single Linked List for the polynomial $F(x) = 5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

Source code for polynomial creation with help of linked list:

```
#include <conio.h>
#include <stdio.h>
#include <malloc.h>

struct link
{
    float coef;
    int expo;
    struct link *next;
};

typedef struct link node;
node * getnode()
{
    node *tmp;
    tmp =(node *) malloc( sizeof(node) );
    printf("\n Enter Coefficient : ");
    fflush(stdin); scanf("%f",&tmp->coef);
    printf("\n Enter Exponent : ");
    fflush(stdin);
    scanf("%d",&tmp->expo);
    tmp->next = NULL;
    return tmp;
}
node * create_poly (node *p )
{
    char ch;
    node *temp,*newnode;
    while( 1 )
    {
        printf ("\n Do U Want polynomial node (y/n): ");
        ch = getch();
        if(ch == 'n')
            break;
        newnode = getnode();
        if( p == NULL )
            p = newnode;
        else
        {
            temp = p; while(temp->next != NULL )
                temp = temp->next;
            temp->next = newnode;
        }
    }
    return p;
}
```

```

void display (node *p)
{
    node *t = p;
    while (t != NULL)
    {
        printf("+ %.2f", t -> coef);
        printf("X^ %d", t -> expo);
        t =t -> next;
    }
}

void main()
{
    node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
    clrscr();
    printf("\nEnter First Polynomial..(in ascending-order of exponent)");
    poly1 = create_poly (poly1);
    printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
    poly2 = create_poly (poly2);
    clrscr();
    printf("\n Enter Polynomial 1:
"); display (poly1);
    printf("\n Enter Polynomial 2:
"); display (poly2);
    getch();
}

```

Addition of Polynomials:

To add two polynomials we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials follow the following steps:

- Read two polynomials.
- Add them.
- Display the resultant polynomial.

Source code for polynomial addition with help of linked list:

```

#include <conio.h>
#include <stdio.h>
#include <malloc.h>

struct link
{
    float coef;
    int expo;
    struct link *next;
};

typedef struct link node;

node * getnode()
{
    node *tmp;

```

```

        tmp =(node *) malloc( sizeof(node) );
        printf("\n Enter Coefficient : ");
        fflush(stdin); scanf("%f",&tmp-
        >coef);
        printf("\n Enter Exponent : ");
        fflush(stdin);
        scanf("%d",&tmp->expo);
        tmp->next = NULL;
        return tmp;
    }

node * create_poly (node *p )
{
    char ch;
    node *temp,*newnode;
    while( 1 )
    {
        printf ("\n Do U Want polynomial node (y/n):
        "); ch = getche();
        if(ch == 'n')
            break;
        newnode = getnode();
        if( p == NULL )
            p = newnode;
        else
        {
            temp = p; while(temp-
            >next != NULL )
                temp = temp->next;
            temp->next = newnode;
        }
    }
    return p;
}

void display (node *p)
{
    node *t = p;
    while (t != NULL)
    {
        printf("+ %.2f", t -> coef);
        printf("X^ %d", t -> expo);
        t = t -> next;
    }
}

void add_poly(node *p1,node *p2)
{
    node *newnode;
    while(1)
    {
        if( p1 == NULL || p2 == NULL
        ) break;
        if(p1->expo == p2->expo )
        {
            printf("+ %.2f X ^%d",p1->coef+p2->coef,p1->expo);
            p1 = p1->next; p2 = p2->next;
        }
        else
        {
            if(p1->expo < p2->expo)

```

```

        {
            printf("+ %.2f X ^%d",p1->coef,p1->expo);
            p1 = p1->next;
        }
        else
        {
            printf(" + %.2f X ^%d",p2->coef,p2->expo); p2 = p2->next;
        }
    }
}
while(p1 != NULL )
{
    printf("+ %.2f X ^%d",p1->coef,p1->expo);
    p1 = p1->next;
}
while(p2 != NULL )
{
    printf("+ %.2f X ^%d",p2->coef,p2->expo);
    p2 = p2->next;
}
}

void main()
{
    node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
    clrscr();
    printf("\nEnter First Polynomial..(in ascending-order of exponent)");
    poly1 = create_poly (poly1);
    printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
    poly2 = create_poly (poly2);
    clrscr();
    printf("\n Enter Polynomial 1:"); display (poly1);
    printf("\n Enter Polynomial 2:"); display (poly2);
    printf( "\n Resultant Polynomial :"); add_poly(poly1, poly2);
    display (poly3);
    getch();
}

```

Exercise

1. Write a function to split a given list of integers represented by a single linked list into two lists in the following way. Let the list be $L = (l_0, l_1, \dots, l_n)$. The resultant lists would be $R_1 = (l_0, l_2, l_4, \dots)$ and $R_2 = (l_1, l_3, l_5, \dots)$.
2. Write a function to insert a node before a node pointed to by X in a single linked list.
3. Write a function to delete a node pointed to by X from a single linked list.
4. Suppose that an ordered list $L = (l_0, l_1, \dots, l_n)$ is represented by a single linked list. It is required to append the list $M = (l_n, l_0, l_1, \dots, l_n)$ after another ordered list M represented by a single linked list.

5. Implement the following function as a new function for the linked list toolkit.

 Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

 Postcondition: The return value is the number of occurrences of 42 in the data field of a node on the linked list. The list itself is unchanged.
6. Implement the following function as a new function for the linked list toolkit.

 Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

 Postcondition: The return value is true if the list has at least one occurrence of the number 42 in the data part of a node.
7. Implement the following function as a new function for the linked list toolkit.

 Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

 Postcondition: The return value is the sum of all the data components of all the nodes. NOTE: If the list is empty, the function returns 0.
8. Write a `concatenate` function to concatenate two circular linked lists producing another circular linked list.
9. Write `eval` functions to compute the following operations on polynomials represented as singly connected linked list of nonzero terms.
 1. Evaluation of a polynomial
 2. Multiplication of two polynomials.
10. Write a `matrix` function to represent a sparse matrix having `m` rows and `n` columns using linked list.
11. Write a `print` function to print a sparse matrix, each row in one line of output and properly formatted, with zero being printed in place of zero elements.
12. Write `add` functions to:
 1. Add two $m \times n$ sparse matrices and
 2. Multiply two $m \times n$ sparse matrices.

Where all sparse matrices are to be represented by linked lists.

13. Consider representing a linked list of integers using arrays. Write a `delete` function to delete the i^{th} node from the list.

- A. `X -> bwd -> fwd = X -> fwd;`
`X -> fwd -> bwd = X -> bwd`
- B. `X -> bwd -> fwd = X -> bwd;`
`X -> fwd -> bwd = X -> fwd`
- C. `X -> bwd -> bwd = X -> fwd;`
`X -> fwd -> fwd = X -> bwd`
- D. `X -> bwd -> bwd = X -> bwd;`
`X -> fwd -> fwd = X -> fwd`
10. Which among the following segment of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to the first element of the list and **start** pointer points to beginning of the list? [B]
- A. `X -> bwd = X -> fwd;`
`X -> fwd = X -> bwd`
- B. `start = X -> fwd;`
`start -> bwd = NULL;`
- C. `start = X -> fwd;`
`X -> fwd = NULL`
- D. `X -> bwd -> bwd = X -> bwd;`
`X -> fwd -> fwd = X -> fwd`
11. Which among the following segment of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to the last element of the list? [C]
- A. `X -> fwd -> bwd = NULL;`
- B. `X -> bwd -> fwd = X -> bwd;`
- C. `X -> bwd -> fwd = NULL;`
- D. `X -> fwd -> bwd = X -> bwd;`
12. Which among the following segment of code counts the number of elements in the double linked list, if it is assumed that X points to the first element of the list and *ctr* is the variable which counts the number of elements in the list? [A]
- A. `for (ctr=1; X != NULL;`
`ctr++) X = X -> fwd;`
- B. `for (ctr=1; X != NULL;`
`ctr++) X = X -> bwd;`
- C. `for (ctr=1; X -> fwd != NULL;`
`ctr++) X = X -> fwd;`
- D. `for (ctr=1; X -> bwd != NULL;`
`ctr++) X = X -> bwd;`
13. Which among the following segment of code counts the number of elements in the double linked list, if it is assumed that X points to the last element of the list and *ctr* is the variable which counts the number of elements in the list? [B]
- A. `for (ctr=1; X != NULL;`
`ctr++) X = X -> fwd;`
- B. `for (ctr=1; X != NULL;`
`ctr++) X = X -> bwd;`
- C. `for (ctr=1; X -> fwd != NULL;`
`ctr++) X = X -> fwd;`
- D. `for (ctr=1; X -> bwd != NULL;`
`ctr++) X = X -> bwd;`

14. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the double linked list. The **start** pointer points to beginning of the list? [B]

- A. X -> bwd = X -> fwd;
X -> fwd = X -> bwd;
- B. X -> fwd = start;
start -> bwd = X;
start = X;
- C. X -> bwd = X -> fwd;
X -> fwd = X -> bwd;
start = X;
- D. X -> bwd -> bwd = X -> bwd;
X -> fwd -> fwd = X -> fwd

15. Which among the following segments of inserts a new node pointed by X to be inserted at the end of the double linked list. The **start** and **last** pointer points to beginning and end of the list respectively? [C]

- A. X -> bwd = X -> fwd;
X -> fwd = X -> bwd
- B. X -> fwd = start;
start -> bwd = X;
- C. last -> fwd = X;
X -> bwd = last;
- D. X -> bwd = X -> bwd;
X -> fwd = last;

16. Which among the following segments of inserts a new node pointed by X to be inserted at any position (i.e neither first nor last) element of the double linked list? Assume **temp** pointer points to the previous position of new node. [D]

- A. X -> bwd -> fwd = X -> fwd;
X -> fwd -> bwd = X -> bwd
- B. X -> bwd -> fwd = X -> bwd;
X -> fwd -> bwd = X -> fwd
- C. temp -> fwd = X;
temp -> bwd = X -> fwd;
X -> fwd = x
X -> fwd -> bwd = temp
- D. X -> bwd = temp;
X -> fwd = temp -> fwd;
temp -> fwd = X;
X -> fwd -> bwd = X;

17. A single linked list is declared as follows:

[A]

```
struct sllist
{
    struct sllist *next;
    int data;
}
```

Where next represents links to adjacent elements of the list.

Which among the following segments of code deletes the element pointed to by **X** from the single linked list, if it is assumed that X points to neither the first nor last element of the list? **prev** pointer points to previous element.

- A. prev -> next = X -> next;
free(X);
- B. X -> next = prev-> next;
free(X);
- C. prev -> next = X -> next;
free(prev);
- D. X -> next = prev -> next;
free(prev);

18. Which among the following segment of code deletes the element pointed to by X from the single linked list, if it is assumed that X points to the first element of the list and **start** pointer points to beginning of the list?

[B]

- A. X = start -> next;
free(X);
- B. start = X -> next;
free(X);
- C. start = start -> next;
free(start);
- D. X = X -> next;
start = X;
free(start);

19. Which among the following segment of code deletes the element pointed to by X from the single linked list, if it is assumed that X points to the last element of the list and **prev** pointer points to last but one element?

[C]

- A. prev -> next = NULL;
free(prev);
- B. X -> next = NULL;
free(X);
- C. prev -> next = NULL;
free(X);
- D. X -> next = prev;
free(prev);

20. Which among the following segment of code counts the number of elements in the single linked list, if it is assumed that X points to the first element of the list and *ctr* is the variable which counts the number of elements in the list? [A]

- A. for (ctr=1; X != NULL; ctr++)
X = X -> next;
- B. for (ctr=1; X != NULL; ctr--)
X = X -> next;
- C. for (ctr=1; X -> next != NULL; ctr++)
X = X -> next;
- D. for (ctr=1; X -> next != NULL; ctr--)
X = X -> next;

21. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the single linked list. The **start** pointer points to beginning of the list? [B]

- A. start -> next = X;
X = start;
- B. X -> next = start;
start = X
- C. X -> next = start -> next;
start = X
- D. X -> next = start;
start = X -> next

22. Which among the following segments of inserts a new node pointed by X to be inserted at the end of the single linked list. The **start** and **last** pointer points to beginning and end of the list respectively? [C]

- A. last -> next = X;
X -> next = start;
- B. X -> next = last;
last ->next = NULL;
- C. last -> next = X;
X -> next = NULL;
- D. last -> next = X -> next;
X -> next = NULL;

22. Which among the following segments of inserts a new node pointed by X to be inserted at any position (i.e neither first nor last) element of the single linked list? Assume **prev** pointer points to the previous position of new node. [D]

- A. X -> next = prev -> next;
prev -> next = X -> next;
- B. X = prev -> next;
prev -> next = X -> next;
- C. X -> next = prev;
prev -> next = X;
- D. X -> next = prev -> next;
prev -> next = X;

24. A circular double linked list is declared as follows:

[A]

```
struct cdllist
{
    struct cdllist *fwd, *bwd;
    int data;
}
```

Where fwd and bwd represents forward and backward links to adjacent elements of the list.

Which among the following segments of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to neither the first nor last element of the list?

- A. X -> bwd -> fwd = X -> fwd;
X -> fwd -> bwd = X -> bwd;
- B. X -> bwd -> fwd = X -> bwd;
X -> fwd -> bwd = X -> fwd;
- C. X -> bwd -> bwd = X -> fwd;
X -> fwd -> fwd = X -> bwd;
- D. X -> bwd -> bwd = X -> bwd;
X -> fwd -> fwd = X -> fwd;

25. Which among the following segment of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to the first element of the list and **start** pointer points to beginning of the list?

[D]

- A. start = start -> bwd;
X -> bwd -> bwd = start;
start -> bwd = X -> bwd;
- B. start = start -> fwd;
X -> fwd -> fwd = start;
start -> bwd = X -> fwd
- C. start = start -> bwd;
X -> bwd -> fwd = X;
start -> bwd = X -> bwd
- D. start = start -> fwd;
X -> bwd -> fwd = start;
start -> bwd = X -> bwd;

26. Which among the following segment of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to the last element of the list and **start** pointer points to beginning of the list?

[B]

- A. X -> bwd -> fwd = X -> fwd;
X -> fwd -> fwd = X -> bwd;
- B. X -> bwd -> fwd = X -> fwd;
X -> fwd -> bwd = X -> bwd;
- C. X -> fwd -> fwd = X -> bwd;
X -> fwd -> bwd = X -> fwd;
- D. X -> bwd -> bwd = X -> fwd;
X -> bwd -> bwd = X -> bwd;

27. Which among the following segment of code counts the number of elements in the circular double linked list, if it is assumed that X and **start** points to the first element of the list and *ctr* is the variable which counts the number of elements in the list? [A]
- A. for (ctr=1; X->fwd != start; ctr++) X = X -> fwd;
- B. for (ctr=1; X != NULL; ctr++) X = X -> bwd;
- C. for (ctr=1; X -> fwd != NULL; ctr++) X = X -> fwd;
- D. for (ctr=1; X -> bwd != NULL; ctr++) X = X -> bwd;
28. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the circular double linked list. The **start** pointer points to beginning of the list? [B]
- A. X -> bwd = start; X -> fwd = start -> fwd; start -> bwd-> fwd = X; start -> bwd = X; start = X
- B. X -> bwd = start -> bwd; X -> fwd = start; start -> bwd-> fwd = X; start -> bwd = X; start = X
- C. X -> fwd = start -> bwd; X -> bwd = start; start -> bwd-> fwd = X; start -> bwd = X; start = X
- D. X -> bwd = start -> bwd; X -> fwd = start; start -> fwd-> fwd = X; start -> fwd = X; X = start;
29. Which among the following segment of code inserts a new node pointed by X to be inserted at the end of the circular double linked list. The **start** pointer points to beginning of the list? [C]
- A. X -> bwd = start; X -> fwd = start -> fwd; start -> bwd -> fwd = X; start -> bwd = X; start = X
- B. X -> bwd = start -> bwd; X -> fwd = start; start -> bwd -> fwd = X; start -> bwd = X; start = X
- C. X -> bwd = start -> bwd; X-> fwd = start; start -> bwd -> fwd = X; start -> bwd = X;
- D. X -> bwd = start -> bwd; X -> fwd = start; start -> fwd-> fwd = X; start -> fwd = X; X = start;
30. Which among the following segments of inserts a new node pointed by X to be inserted at any position (i.e neither first nor last) element of the circular double linked list? Assume **temp** pointer points to the previous position of new node. [D]
- A. X -> bwd -> fwd = X -> fwd; X -> fwd -> bwd = X -> bwd;
- B. X -> bwd -> fwd = X -> bwd; X -> fwd -> bwd = X -> fwd;
- C. temp -> fwd = X; temp -> bwd = X -> fwd; X -> fwd = X; X -> fwd -> bwd = temp;
- D. X -> bwd = temp; X -> fwd = temp -> fwd; temp -> fwd = X; X -> fwd -> bwd = X;

Chapter 5

Graphs

Introduction to Graphs:

Graph G is a pair (V, E) , where V is a finite set of vertices and E is a finite set of edges. We will often denote $n = |V|$, $e = |E|$.

A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and the edges by lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is our undirected edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph in figure 6.5.1(a) is an undirected graph. If all the edges are directed; then the graph is a directed graph. The graph of figure 6.5.1(b) is a directed graph. A directed graph is also called as digraph. A graph G is connected if and only if there is a simple path between any two nodes in G .

A graph G is said to be complete if every node a in G is adjacent to every other node v in G . A complete graph with n nodes will have $n(n-1)/2$ edges. For example, Figure 6.5.1.(a) and figure 6.5.1.(d) are complete graphs.

A directed graph G is said to be connected, or strongly connected, if for each pair (u, v) for nodes in G there is a path from u to v and also a path from v to u . On the other hand, G is said to be unilaterally connected if for any pair (u, v) of nodes in G there is a path from u to v or a path from v to u . For example, the digraph shown in figure 6.5.1 (e) is strongly connected.

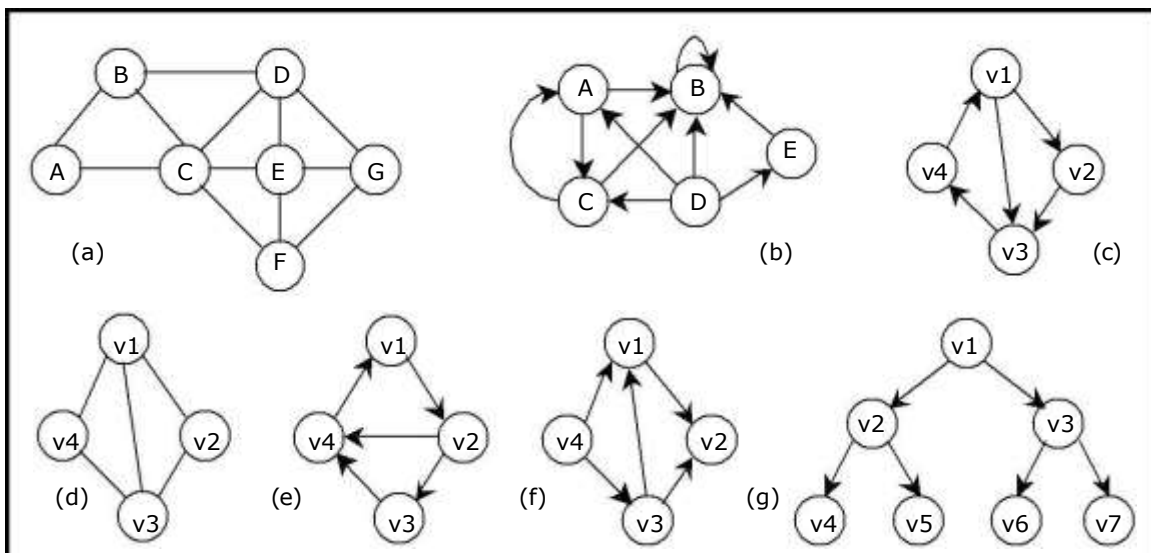


Figure 6.5.1 Various Graphs

We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called weighted graph.

The number of incoming edges to a vertex v is called in-degree of the vertex (denote $\text{indeg}(v)$). The number of outgoing edges from a vertex is called out-degree (denote $\text{outdeg}(v)$). For example, let us consider the digraph shown in figure 6.5.1(f),

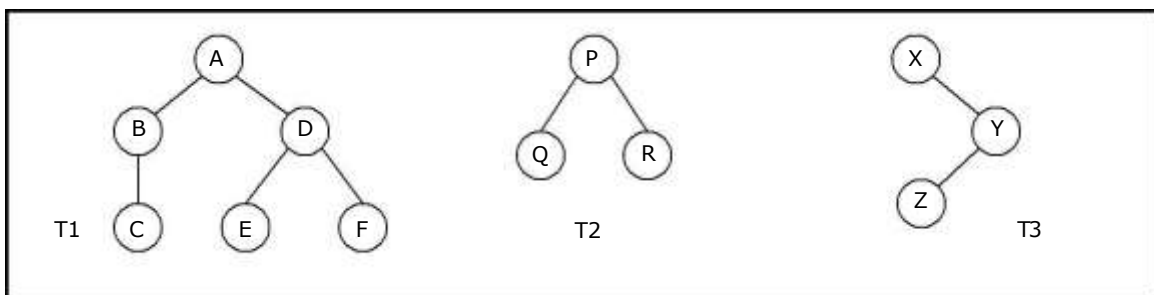
$$\begin{array}{ll} \text{indegree}(v_1) = 2 & \text{outdegree}(v_1) = 1 \\ \text{indegree}(v_2) = 2 & \text{outdegree}(v_2) = 0 \end{array}$$

A path is a sequence of vertices (v_1, v_2, \dots, v_k) , where for all i , $(v_i, v_{i+1}) \in E$. A path is simple if all vertices in the path are distinct. If there is a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 6.5.1(a), figure 6.5.1(c) and figure 6.5.1(d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 6.5.1 (f) and figure 6.5.1 (g) are acyclic graphs.

A graph $G' = (V', E')$ is a sub-graph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T_1 , T_2 and T_3 .



A Forest F

A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren't any sequences of edges that go around in a loop). A spanning tree of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G . A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G . Then

- *Any two vertices in T are connected by a unique simple path.*
- *If any edge is removed from T , then T becomes disconnected.*
- *If we add any edge into T , then the new graph will contain a cycle.*
- *Number of edges in T is $n-1$.*

Representation of Graphs:

There are two ways of representing digraphs. They are:

- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional $n \times n$ matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.

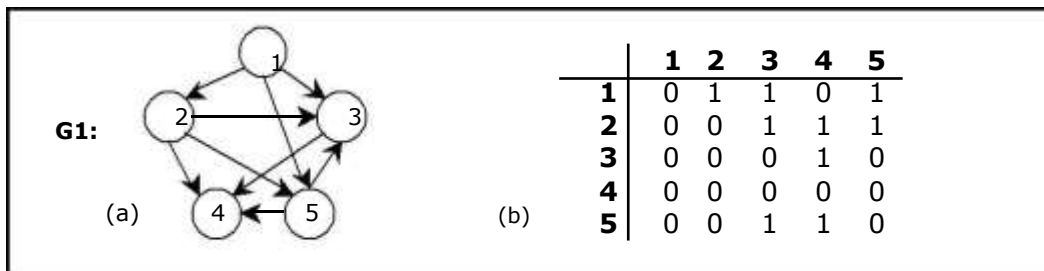


Figure 6.5.2. A graph and its Adjacency matrix

Figure 6.5.2(b) shows the adjacency matrix representation of the graph G_1 shown in figure 6.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 6.5.3(b) shows the cost adjacency matrix representation of the graph G_2 shown in figure 6.5.3(a).

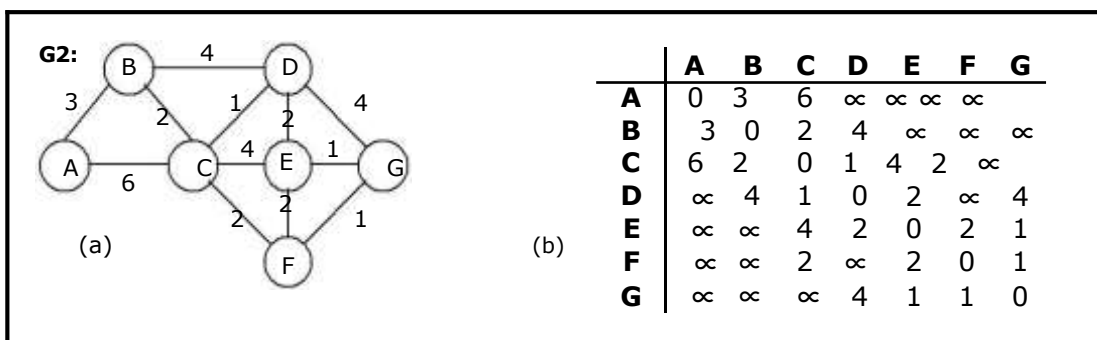


Figure 6.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:

In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array Adj[1, 2, n] of pointers where for $1 \leq v \leq n$, Adj[v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 6.5.4(a), the adjacency list in shown in figure 6.5.4 (b).

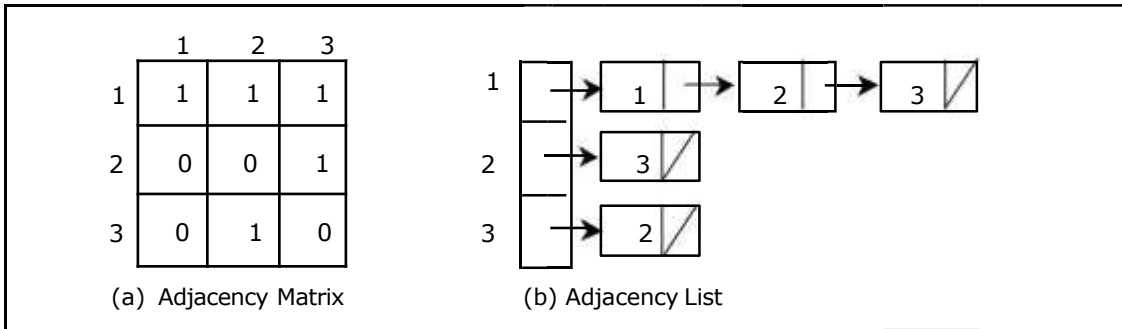


Figure 6.5.4 Adjacency matrix and adjacency list

Incidence Matrix:

In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge } j \text{ incident to } v_i \\ 0 & \text{otherwise} \end{cases}$$

Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.

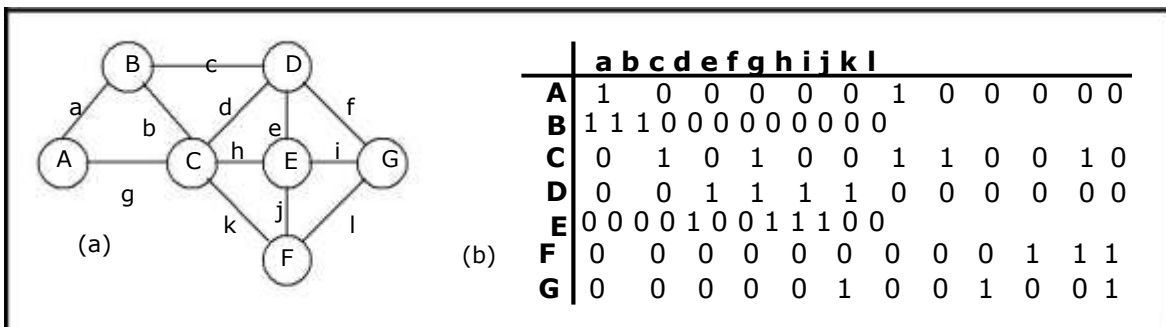


Figure 6.5.4 Graph and its incidence matrix

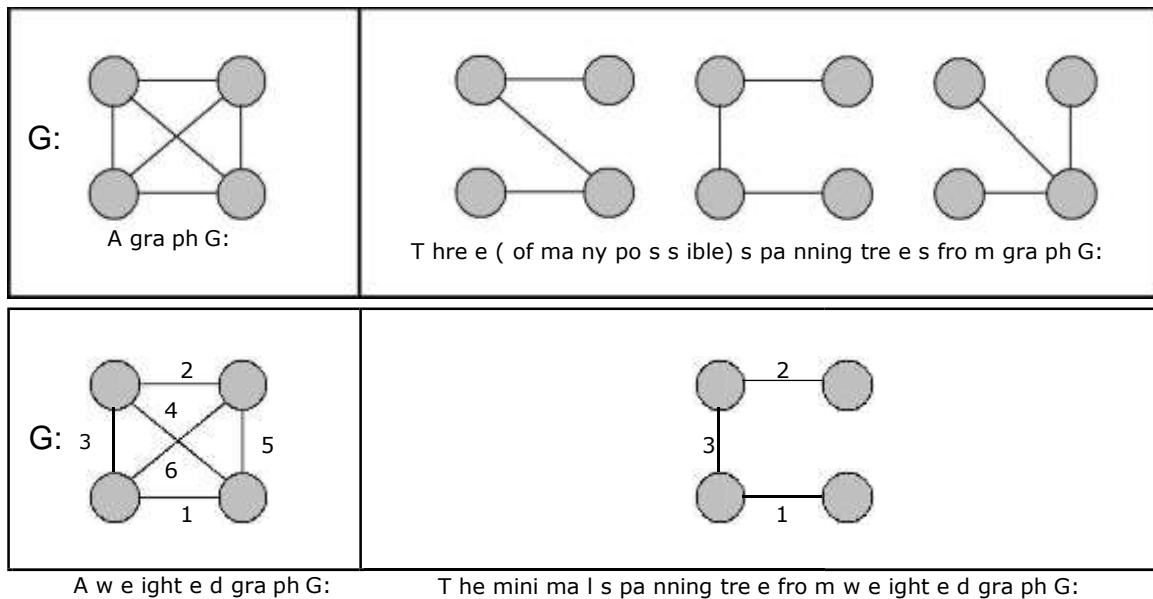
Figure 6.5.4(b) shows the incidence matrix representation of the graph G1 shown in figure 6.5.4(a).

Minimum Spanning Tree (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T . Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Example:



Let's consider a couple of real-world examples on minimum spanning tree:

10. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
11. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

- \{ Kruskal's algorithm and
- \{ Prim's algorithm.

Both algorithms differ in their methodology, but both eventually end up with the MST. *Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections* in determining the MST. In *Prim's algorithm at any instance of output it represents tree* whereas in *Kruskal's algorithm at any instance of output it may represent tree or not*.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until $(n - 1)$ edges have been added. Sometimes two or more edges may have the same cost.

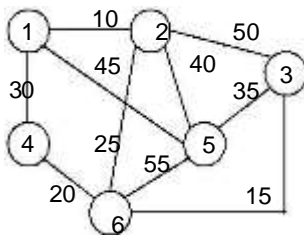
The order in which the edges are chosen, in this case, does not matter. Different MST's may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal's Algorithm for minimal spanning tree is as follows:

- 1.9. Make the tree T empty.
 - 1.10. Repeat the steps 3, 4 and 5 as long as T contains less than $n - 1$ edges and E is not empty otherwise, proceed to step 6.
 - 1.11. Choose an edge (v, w) from E of lowest cost.
 - 1.12. Delete (v, w) from E.
 - 1.13. If (v, w) does not create a cycle in T
 then Add (v, w) to T
 else discard (v, w)
6. If T contains fewer than $n - 1$ edges then print no spanning tree.

Example 1:

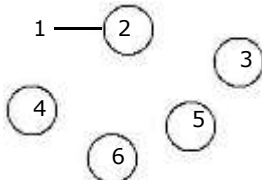
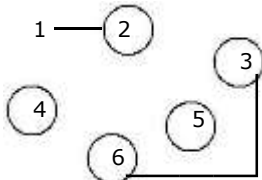
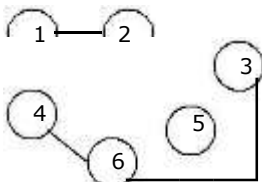
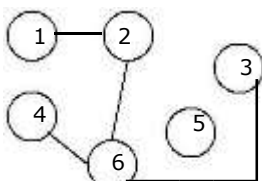
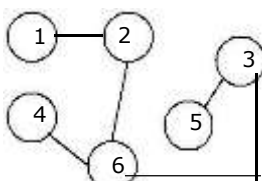
Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

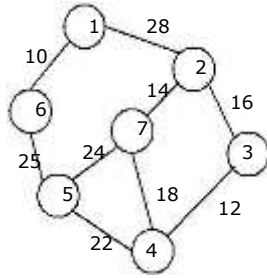
Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 2:

Construct the minimal spanning tree for the graph shown below:



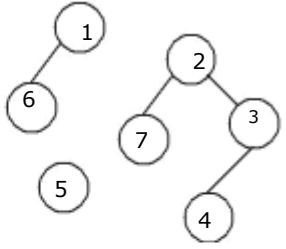
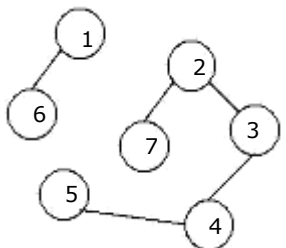
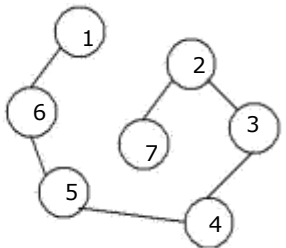
Solution:

Arrange all the edges in the increasing order of their costs:

Cost	10	12	14	16	18	22	24	25	28
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject	is discarded as its inclusion creates a cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 99.

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim's Algorithm:

E is the set of edges in G. cost [1:n, 1:n] is the cost adjacency matrix of an n vertex graph such that cost [i, j] is either a positive real number or ∞ if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in the minimum-cost spanning tree. The final cost is returned.

Algorithm Prim (E, cost, n, t)

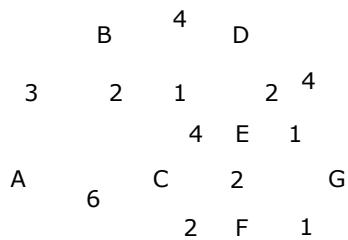
```

{
  Let (k, l) be an edge of minimum cost in E;
  mincost := cost [k, l];
  t [1, 1] := k; t [1, 2] := l;
  for i :=1 to n do // Initialize near
    if (cost [i, l] < cost [i, k]) then near [i] := l;
    else near [i] := k;
  near [k] :=near [l] := 0;
  for i:=2 to n - 1 do // Find n - 2 additional edges for t.
  {
    Let j be an index such that near [j]  $\neq$  0
    and cost [j, near [j]] is minimum;
    t [i, 1] := j; t [i, 2] := near [j];
    mincost := mincost + cost [j, near
    [j]]; near [j] := 0
    for k:= 1 to n do // Update near[].
      if ((near [k]  $\neq$  0) and (cost [k, near [k]] > cost [k,
      j])) then near [k] := j;
  }
  return mincost;
}

```

EXAMPLE:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



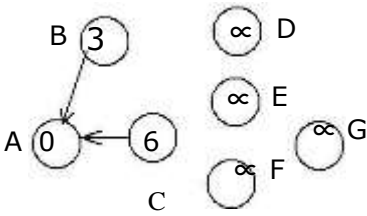
Solution:

The cost adjacency matrix is

	0	3	6	∞	∞	∞	∞
	3	0	2	4	∞	∞	∞
	6	2	0	1	4	2	∞
	∞	4	1	0	2	∞	4
	∞	∞	4	2	0	2	1
	∞	∞	2	∞	2	0	1
	∞	∞	∞	4	1	1	0

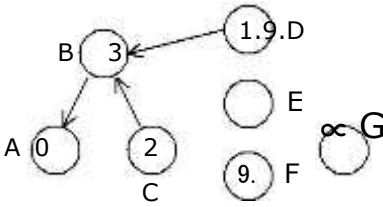
The stepwise progress of the prim's algorithm is as follows:

Step 1:



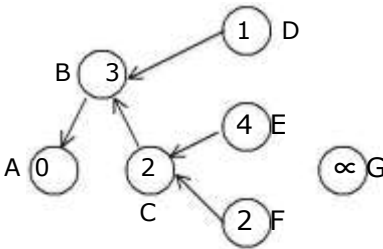
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∞	∞	∞	∞
Next	*	A	A	A	A	A	A

Step 2:



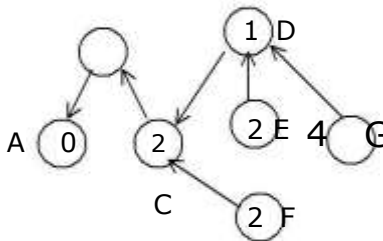
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∞	∞	∞
Next	*	A	B	B	A	A	A

Step 3:



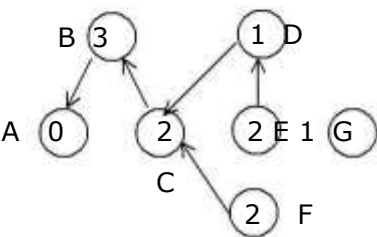
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∞
Next	*	A	B	C	C	C	A

Step 4:



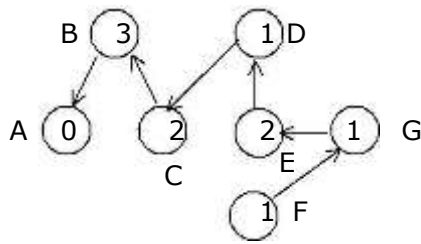
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



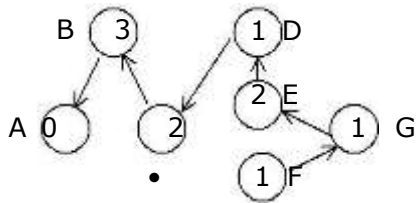
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

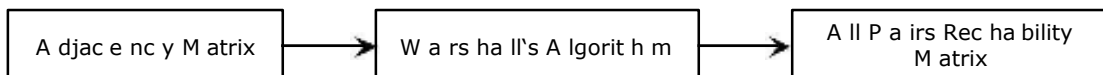
Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

Reachability Matrix (Warshall's Algorithm):

Warshall's algorithm requires knowing which edges exist and which does not. It doesn't need to know the lengths of the edges in the given directed graph. This information is conveniently displayed by adjacency matrix for the graph, in which a '1' indicates the existence of an edge and '0' indicates non-existence.



It begins with the adjacency matrix for the given graph, which is called A_0 , and then updates the matrix n times, producing matrices called A_1, A_2, \dots, A_n and then stops.

In warshall's algorithm the matrix A_i contains information about the existence of i -paths. A one entry in the matrix A_i will correspond to the existence of i -paths and zero entry will correspond to non-existence. Thus when the algorithm stops, the final matrix A_n , contains the desired connectivity information.

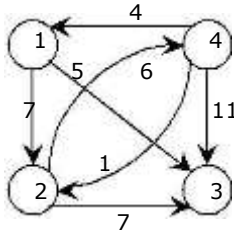
A one entry indicates a pair of vertices, which are connected and zero entry indicates a pair, which are not. This matrix is called a *reachability matrix* or *path matrix* for the graph. It is also called the *transitive closure* of the original adjacency matrix.

The update rule for computing A_i from A_{i-1} in warshall's algorithm is:

$$A_i[x, y] = A_{i-1}[x, y] \vee (A_{i-1}[x, i] \wedge A_{i-1}[i, y]) \quad \text{----} \quad (1)$$

Example 1:

Use warshall's algorithm to calculate the reachability matrix for the graph:



We begin with the adjacency matrix of the graph A_0

$$A_0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{matrix}$$

The first step is to compute A_1 matrix. To do so we will use the updating rule – (1).

Before doing so, we notice that only one entry in A_0 must remain one in A_1 , since in Boolean algebra $1 + (\text{anything}) = 1$. Since these are only nine zero entries in A_0 , there are only nine entries in A_0 that need to be updated.

$$\begin{aligned} A_1[1, 1] &= A_0[1, 1] \vee (A_0[1, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0 \\ A_1[1, 4] &= A_0[1, 4] \vee (A_0[1, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0 \\ A_1[2, 1] &= A_0[2, 1] \vee (A_0[2, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0 \\ A_1[2, 2] &= A_0[2, 2] \vee (A_0[2, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0 \\ A_1[3, 1] &= A_0[3, 1] \vee (A_0[3, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0 \\ A_1[3, 2] &= A_0[3, 2] \vee (A_0[3, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0 \\ A_1[3, 3] &= A_0[3, 3] \vee (A_0[3, 1] \wedge A_0[1, 3]) = 0 \vee (0 \wedge 1) = 0 \\ A_1[3, 4] &= A_0[3, 4] \vee (A_0[3, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0 \\ A_1[4, 4] &= A_0[4, 4] \vee (A_0[4, 1] \wedge A_0[1, 4]) = 0 \vee (1 \wedge 0) = 0 \end{aligned}$$

$$A_1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{matrix}$$

Next, A_2 must be calculated from A_1 ; but again we need to update the 0 entries,

$$\begin{aligned} A_2[1, 1] &= A_1[1, 1] \vee (A_1[1, 2] \wedge A_1[2, 1]) = 0 \vee (1 \wedge 0) = 0 \\ A_2[1, 4] &= A_1[1, 4] \vee (A_1[1, 2] \wedge A_1[2, 4]) = 0 \vee (1 \wedge 1) = 1 \\ A_2[2, 1] &= A_1[2, 1] \vee (A_1[2, 2] \wedge A_1[2, 1]) = 0 \vee (0 \wedge 0) = 0 \\ A_2[2, 2] &= A_1[2, 2] \vee (A_1[2, 2] \wedge A_1[2, 2]) = 0 \vee (0 \wedge 0) = 0 \\ A_2[3, 1] &= A_1[3, 1] \vee (A_1[3, 2] \wedge A_1[2, 1]) = 0 \vee (0 \wedge 0) = 0 \\ A_2[3, 2] &= A_1[3, 2] \vee (A_1[3, 2] \wedge A_1[2, 2]) = 0 \vee (0 \wedge 0) = 0 \end{aligned}$$

$$\begin{aligned}
A_2[3, 3] &= A_1[3, 3] \vee (A_1[3, 2] \wedge A_1[2, 3]) = 0 \vee (0 \wedge 1) = 0 \\
A_2[3, 4] &= A_1[3, 4] \vee (A_1[3, 2] \wedge A_1[2, 4]) = 0 \vee (0 \wedge 1) = 0 \\
A_2[4, 4] &= A_1[4, 4] \vee (A_1[4, 2] \wedge A_1[2, 4]) = 0 \vee (1 \wedge 1) = 1
\end{aligned}$$

$$A = \begin{matrix} & \begin{matrix} 1 & 0 & 1 & 1 \end{matrix} \\ \begin{matrix} 2 \\ 3 \end{matrix} & \begin{matrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{matrix} \\ & \begin{matrix} 4 & 1 & 1 & 1 \end{matrix} \end{matrix}$$

This matrix has only seven 0 entries, and so to compute A_3 , we need to do only seven computations.

$$\begin{aligned}
A_3[1, 1] &= A_2[1, 1] \vee (A_2[1, 3] \wedge A_2[3, 1]) = 0 \vee (1 \wedge 0) = 0 \\
A_3[2, 1] &= A_2[2, 1] \vee (A_2[2, 3] \wedge A_2[3, 1]) = 0 \vee (1 \wedge 0) = 0 \\
A_3[2, 2] &= A_2[2, 2] \vee (A_2[2, 3] \wedge A_2[3, 2]) = 0 \vee (1 \wedge 0) = 0 \\
A_3[3, 1] &= A_2[3, 1] \vee (A_2[3, 3] \wedge A_2[3, 1]) = 0 \vee (0 \wedge 0) = 0 \\
A_3[3, 2] &= A_2[3, 2] \vee (A_2[3, 3] \wedge A_2[3, 2]) = 0 \vee (0 \wedge 0) = 0 \\
A_3[3, 3] &= A_2[3, 3] \vee (A_2[3, 3] \wedge A_2[3, 3]) = 0 \vee (0 \wedge 0) = 0 \\
A_3[3, 4] &= A_2[3, 4] \vee (A_2[3, 3] \wedge A_2[3, 4]) = 0 \vee (0 \wedge 0) = 0
\end{aligned}$$

$$A = \begin{matrix} & \begin{matrix} 1 & 0 & 1 & 1 \end{matrix} \\ \begin{matrix} 2 \\ 3 \end{matrix} & \begin{matrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{matrix} \\ & \begin{matrix} 4 & 1 & 1 & 1 \end{matrix} \end{matrix}$$

Once A_3 is calculated, we use the update rule to calculate A_4 and stop. This matrix is the reachability matrix for the graph.

$$\begin{aligned}
A_4[1, 1] &= A_3[1, 1] \vee (A_3[1, 4] \wedge A_3[4, 1]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1 \\
A_4[2, 1] &= A_3[2, 1] \vee (A_3[2, 4] \wedge A_3[4, 1]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1 \\
A_4[2, 2] &= A_3[2, 2] \vee (A_3[2, 4] \wedge A_3[4, 2]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1 \\
A_4[3, 1] &= A_3[3, 1] \vee (A_3[3, 4] \wedge A_3[4, 1]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0 \\
A_4[3, 2] &= A_3[3, 2] \vee (A_3[3, 4] \wedge A_3[4, 2]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0 \\
A_4[3, 3] &= A_3[3, 3] \vee (A_3[3, 4] \wedge A_3[4, 3]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0 \\
A_4[3, 4] &= A_3[3, 4] \vee (A_3[3, 4] \wedge A_3[4, 4]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0
\end{aligned}$$

$$A = \begin{matrix} & \begin{matrix} 1 & 1 & 1 & 1 \end{matrix} \\ \begin{matrix} 2 \\ 3 \\ 4 \end{matrix} & \begin{matrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \end{matrix}$$

Note that according to the algorithm vertex 3 is not reachable from itself 1. This is because as can be seen in the graph, there is no path from vertex 3 back to itself.

Traversing a Graph

Many graph algorithms require one to systematically examine the nodes and edges of a graph G . There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.

During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N , as follows:

1. STATUS = 1 (Ready state): The initial state of the node N .
2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.
3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first search are called depth first spanning trees. The spanning trees obtained using breadth first search are called Breadth first spanning trees.

Breadth first search and traversal:

The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A . Then we examine all the neighbors of A . Then we examine all the neighbors of neighbors of A . And so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:

This algorithm executes a BFT on graph G beginning at a starting node A .

Initialize all nodes to the ready state (STATUS = 1).

1. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).
2. Repeat the following steps until QUEUE is empty:
 - a. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).
 - b. Add to the rear of QUEUE all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
3. Exit.

Depth first search and traversal:

Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFS except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.

The algorithm for depth first traversal on a graph G is as follows.

This algorithm executes a DFT on graph G beginning at a starting node A.

5. Initialize all nodes to the ready state (STATUS = 1).
6. Push the starting node A into STACK and change its status to the waiting state (STATUS = 2).
7. Repeat the following steps until STACK is empty:

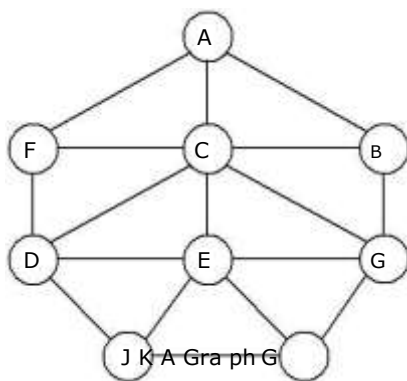
Pop the top node N from STACK. Process N and change the status of N to the processed state (STATUS = 3).

Push all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

8. Exit.

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: **A F C B D E G J K.**

Depth-first search and traversal:

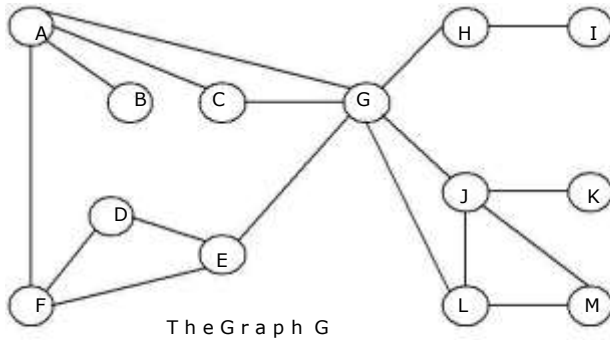
The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: **A F D J K G E C B.**

Example 2:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.

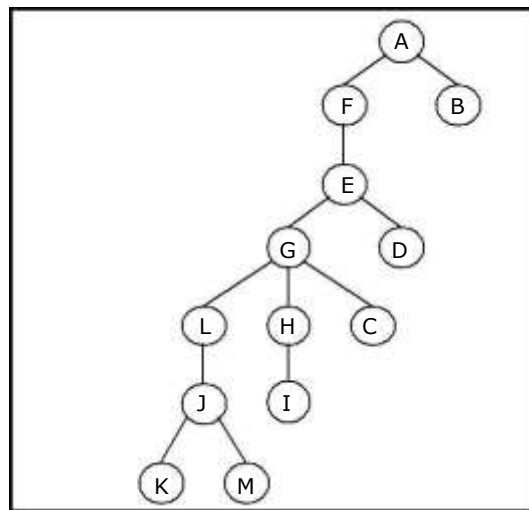


The Graph G

Node	Adjacency List
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M

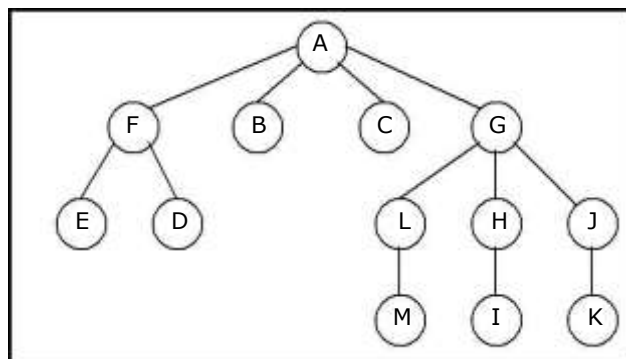
The Adjacency List for the graph G

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A F E G L J K M H I C D B**. The depth first spanning tree is shown in the figure given below:



Depth first Traversal

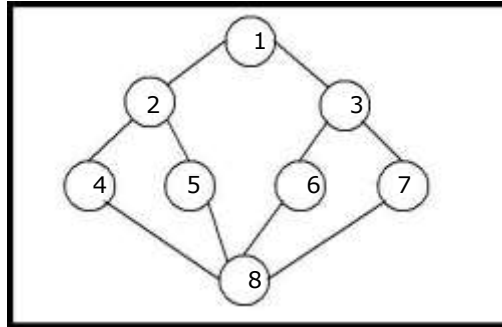
If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A F B C G E D L H J M I K**. The breadth first spanning tree is shown in the figure given below:



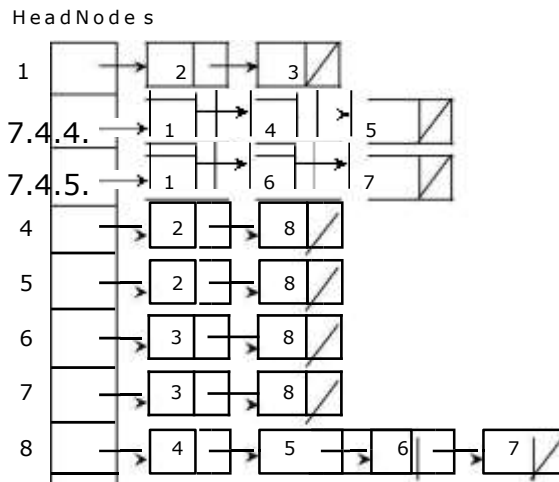
Breadth first traversal

Example 3:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



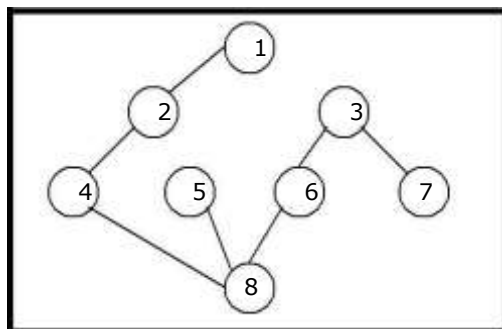
Graph G



A d j a c e n c y l i s t f o r g r a p h G

Depth first search and traversal:

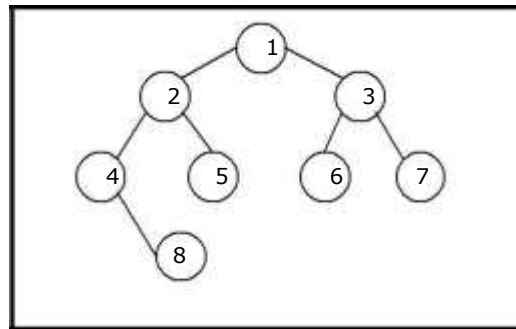
If the depth first is initiated from vertex 1, then the vertices of graph G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:



Depth First Spanning Tree

Breadth first search and traversal:

If the breadth first search is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 3, 4, 5, 6, 7, 8. The breadth first spanning tree is as follows:

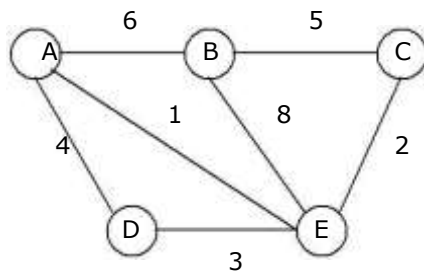


Breadth First Spanning Tree

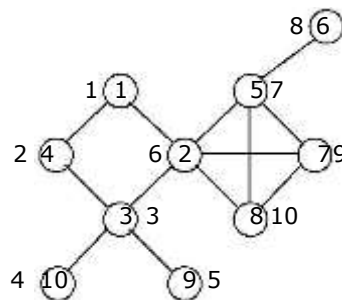
EXERCISES

6. Show that the sum of degrees of all vertices in an undirected graph is twice the number of edges.
7. Show that the number of vertices of odd degree in a finite graph is even.
8. How many edges are contained in a complete graph of n vertices.
9. Show that the number of spanning trees in a complete graph of n vertices is $2^{n-1} - 1$.
10. Prove that the edges explored by a breadth first or depth first traversal of a connected graph form a tree.
11. Explain how existence of a cycle in an undirected graph may be detected by traversing the graph in a depth first manner.
12. Write a function to generate the incidence matrix of a graph from its adjacency matrix.
13. Give an example of a connected directed graph so that a depth first traversal of that graph yields a forest and not a spanning tree of the graph.
14. Rewrite the algorithms `BFSearch` and `DFS` so that it works on adjacency matrix representation of graphs.
15. Write a function to find out whether there is a path between any two vertices in a graph (i.e. to compute the transitive closure matrix of a graph)
16. Write a function to delete an existing edge from a graph represented by an adjacency list.
17. Construct a weighted graph for which the minimal spanning trees produced by Kruskal's algorithm and Prim's algorithm are different.

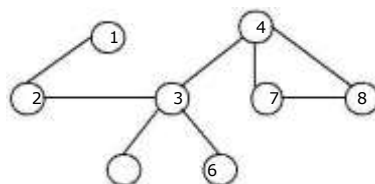
5. Describe the algorithm to find a minimum spanning tree T of a weighted graph G . Find the minimum spanning tree T of the graph shown below.



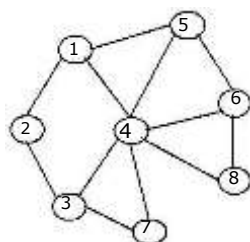
5. For the graph given below find the following:
 Linked representation of the graph.
 Adjacency list.
 Depth first spanning tree.
 Breadth first spanning tree.
 Minimal spanning tree using Kruskal's and Prim's algorithms.



- 3.6. For the graph given below find the following:
 Linked representation of the graph.
 Adjacency list.
 Depth first spanning tree.
 Breadth first spanning tree.
 Minimal spanning tree using Kruskal's and Prim's algorithms.

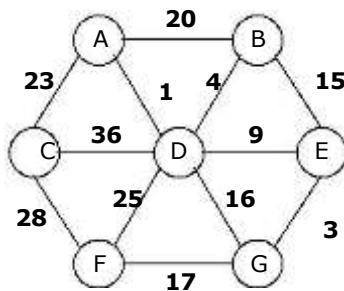


- For the graph given below find the following:
 Linked representation of the graph.
 Adjacency list.
 Depth first spanning tree.
 Breadth first spanning tree.
 Minimal spanning tree using Kruskal's and Prim's algorithms.



Multiple Choice Questions

1. How can the graphs be represented? [D]
 A. Adjacency matrix C. Incidence matrix
 B. Adjacency list D. All of the above
2. The depth-first traversal in graph is analogous to tree traversal: [C]
 A. In-order C. Pre-order
 B. Post-order D. Level order
3. The children of a same parent node are called as: [C]
 A. adjacent node C. Siblings
 B. non-leaf node D. leaf node
4. Complete graphs with n nodes will have _____ edges. [C]
 A. $n - 1$ C. $n(n-1)/2$
 B. $n/2$ D. $(n - 1)/2$
5. A graph with no cycle is called as: [C]
 A. Sub-graph C. Acyclic graph
 B. Directed graph D. none of the above
6. The maximum number of nodes at any level is: [B]
 A. n C. n + 1
 B. 2^n D. 2n



Node	Adjacency List
A	B C D
B	A D E
C	A D F
D	A B C E F G
E	B D G
F	C D G
G	F D E

FIGURE 1 and its adjacency list

7. For the figure 1 shown above, the depth first spanning tree visiting sequence is: [B]
 A. A B C D E F G C. A B C D E F G
 B. A B D C F G E D. none of the above
8. For the figure 1 shown above, the breadth first spanning tree visiting sequence is: [B]
 A. A B D C F G E C. A B C D E F G
 B. A B C D E F G D. none of the above
9. Which is the correct order for Kruskal's minimum spanning tree algorithm to add edges to the minimum spanning tree for the figure 1 shown above: [B]
 • (A, B) then (A, C) then (A, D) then (D, E) then (C, F) then (D, G)
 • (A, D) then (E, G) then (B, D) then (D, E) then (F, G) then (A, C)
 • both A and B
 • none of the above
10. For the figure 1 shown above, the cost of the minimal spanning tree is: [A]
 A. 57 C. 48
 B. 68 D. 32

11. A simple graph has no loops. What other property must a simple graph have? [D]
 A. It must be directed. C. It must have at least one vertex.
 B. It must be undirected. D. It must have no multiple edges.
12. Suppose you have a directed graph representing all the flights that an airline flies. What algorithm might be used to find the best sequence of connections from one city to another? [D]
 A. Breadth first search. C. A cycle-finding algorithm.
 B. Depth first search. D. A shortest-path algorithm.
13. If G is an directed graph with 20 vertices, how many boolean values will be needed to represent G using an adjacency matrix? [D]
 A. 20 C. 200
 B. 40 D. 400
14. Which graph representation allows the most efficient determination of the existence of a particular edge in a graph? [B]
 A. An adjacency matrix. C. Incidence matrix
 B. Edge lists. D. none of the above
15. What graph traversal algorithm uses a queue to keep track of vertices which need to be processed? [A]
 A. Breadth-first search. C Level order search
 B. Depth-first search. D. none of the above
16. What graph traversal algorithm uses a stack to keep track of vertices which need to be processed? [B]
 A. Breadth-first search. C Level order search
 B. Depth-first search. D. none of the above
17. What is the expected number of operations needed to loop through all the edges terminating at a particular vertex given an adjacency matrix representation of the graph? (Assume n vertices are in the graph and m edges terminate at the desired node.) [D]
 A. $O(m)$ C. $O(m^2)$
 B. $O(n)$ D. $O(n^2)$
18. What is the expected number of operations needed to loop through all the edges terminating at a particular vertex given an adjacency list representation of the graph? (Assume n vertices are in the graph and m edges terminate at the desired node.) [A]
 A. $O(m)$ C. $O(m^2)$
 B. $O(n)$ D. $O(n^2)$
19. [B]

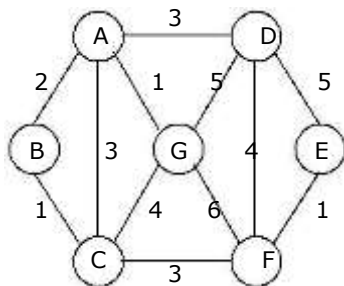


FIGURE 3

For the figure 3, starting at vertex A, which is a correct order for Prim's minimum spanning tree algorithm to add edges to the minimum spanning tree?

3. (A, G) then (G, C) then (C, B) then (C, F) then (F, E) then (E, D)
4. (A, G) then (A, B) then (B, C) then (A, D) then (C, F) then (F, E)
5. (A, G) then (B, C) then (E, F) then (A, B) then (C, F) then (D, E)
6. (A, G) then (A, B) then (A, C) then (A, D) then (A, D) then (C, F)

20. For the figure 3, which is a correct order for Kruskal's minimum spanning [C] tree algorithm to add edges to the minimum spanning tree?

4. (A, G) then (G, C) then (C, B) then (C, F) then (F, E) then (E, D)
5. (A, G) then (A, B) then (B, C) then (A, D) then (C, F) then (F, E)
6. (A, G) then (B, C) then (E, F) then (A, B) then (C, F) then (D, E)
7. (A, G) then (A, B) then (A, C) then (A, D) then (A, D) then (C, F)

21. Which algorithm does not construct an in-tree as part of its processing? []

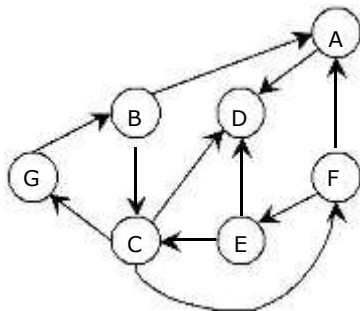
- A. Dijkstra's Shortest Path Algorithm
- B. Prim's Minimum Spanning Tree Algorithm
- C. Kruskal's Minimum Spanning Tree Algorithm
- D. The Depth-First Search Trace Algorithm

22. The worst-case running time of Kruskal's minimum-cost spanning tree algorithm on a graph with n vertices and m edges is: []

- A. C.
- B. D.

23. An adjacency matrix representation of a graph cannot contain information of: [D]

- A. Nodes
- B. Edges
- C. Direction of edges
- D. Parallel edges



Node	Adjacency List
A	D
B	A C
C	G D F
D	----
E	C D
F	E A
G	B

FIGURE 4 and its adjacency list

24. For the figure 4, which edge does not occur in the depth first spanning tree resulting from depth first search starting at node B: [B]

- A. $F \rightarrow E$
- B. $E \rightarrow C$
- C. $C \rightarrow G$
- D. $C \rightarrow F$

25. The set of all edges generated by DFS tree starting at node B is: [A]

- A. B A D C G F E
- B. A D
- C. B A C D G F E
- D. Cannot be generated

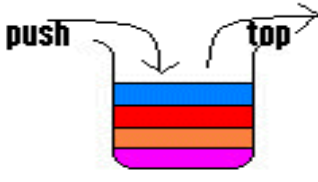
26. The set of all edges generated by BFS tree starting at node B is: [C]

- A. B A D C G F E
- B. A D
- C. B A C D G F E
- D. Cannot be generated

Elementary Data Structures

Stacks, Queues, Lists, and Related Structures

Stacks, lists and queues are primitive data structures fundamental to implementing any program requiring data storage and retrieval. The following tables offer specific information on each type of data structure. The rest of the web page offers information about implementing and applying these data structures.

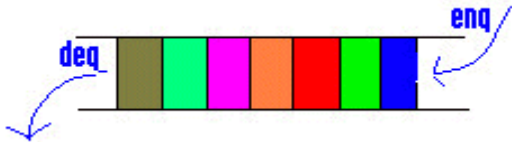
ADT	Mathematical Model	Operations
Stack		Push(S,Data) Pop(S) Makenull(S) Empty(S) Top(S)

The mathematical model of a stack is LIFO (last in, first out). Data placed in the stack is accessed through one path. The next available data is the last data to be placed in the stack. In other words, the "newest" data is withdrawn.

The standard operations on a stack are as follows:

- PUSH(S, Data) :** Put 'Data' in stack 'S'
- POP(S) :** Withdraw next available data from stack 'S'
- MAKENULL(S) :** Clear stack 'S' of all data
- EMPTY(S) :** Returns boolean value 'True' if stack 'S' is empty; returns 'False' otherwise
- TOP(S) :** Views the next available data on stack 'S'. This operation is redundant since one can simply POP(S), view the data, then PUSH(S,Data)

All operations can be implemented in O(1) time.

ADT	Mathematical Model	Operations
Queue		Enqueue(Q) Dequeue(Q) Front(Q) Rear(Q) Makenuil(Q)

The mathematical model of a queue is FIFO (first in, first out). Data placed in the queue goes through one path, while data withdrawn goes through another path at the "opposite" end of the queue. The next available data is the first data placed in the queue. In other words, the "oldest" data is withdrawn.

The standard operation on a queue are as follows:

- ENQUEUE(Q, Data) :** Put 'Data' in the rear path of queue 'Q'
- DEQUEUE(Q) :** Withdraw next available data from front path of queue 'Q'
- FRONT(Q) :** Views the next available data on queue 'Q'
- REAR(Q) :** Views the last data entered on queue 'Q'
- MAKENULL(Q) :** Clear queue 'Q' of all data.

All operations can be implemented in $O(1)$ time.

ADT	Mathematical Model	Operations
Deque		Inject(D,Data) Eject(D) Dequeue(D) Enqueue(D,Data) Front(D) Rear(D) MakeNull(D)

The mathematical model of a deque is similar to a queue. However, a deque is a "double-ended" queue. The model allows data to be entered and withdrawn from the front and rear of the data structure.

The standard operations on a deque are as follows:

INJECT(D, Data) :	Put 'Data' in front path of deque 'D'
EJECT(D) :	Withdraw next available data from rear path of deque 'D'
ENQUEUE(D, Data) :	Put 'Data' in rear path of deque 'D'
DEQUEUE(D) :	Withdraw next available data from front path of deque 'D'
FRONT(D) :	Views the next available data from front path of deque 'D'
REAR(D) :	Views the next available data from rear path of deque 'D'
MAKENULL(D) :	Clear deque 'D' of all data.

All operations can be implemented in $O(1)$ time.

ADT	Mathematical model	Operations
List	$[x_1, x_2, \dots, x_n]$	Concatenate(L1, L2) Access(L, i) Sublist(L, [i...j])

The mathematical model of a list is a string of data. The model allows data to be added or deleted anywhere in the list.

The standard operations on a list are as follows:

CONCATENATE(L1, L2) : Two lists L1, L2 are joined as follows,

$$L1 = [x_1, x_2, x_3, \dots, x_n]$$

$$L2 = [y_1, y_2, y_3, \dots, y_m]$$

$$L = [x_1, x_2, x_3, \dots, x_n, y_1, y_2, \dots, y_m]$$

ACCESS(L, i) : Returns data x_i

SUBLIST(L, [i...j]) : Returns $[x_i, x_{i+1}, \dots, x_j]$
Variations of this are **sublist(L, [i...])** which returns $[x_i, x_{i+1}, \dots, x_n]$, and **SUBLIST(L, [...i])**, which returns $[x_1, x_2, \dots, x_i]$

CONCATENATE, ACCESS, and SUBLIST are called "ATOMIC" operations. Using these three operations we can make other operations.

Examples:

INSERT_AFTER(L, x, i) : Inserts data 'x' after member x_i in list 'L'.

This can be implemented by the following operations:

**CONCATENATE(SUBLIST(L,[...i])),
CONCATENATE([x], SUBLIST(L,[i+1...]))**

DELETE(L, i) :Removes x_i from list 'L'.

This can be implemented as the following operations:

**CONCATENATE(SUBLIST(L[...i-1])),
SUBLIST(L, [i+1...]))**

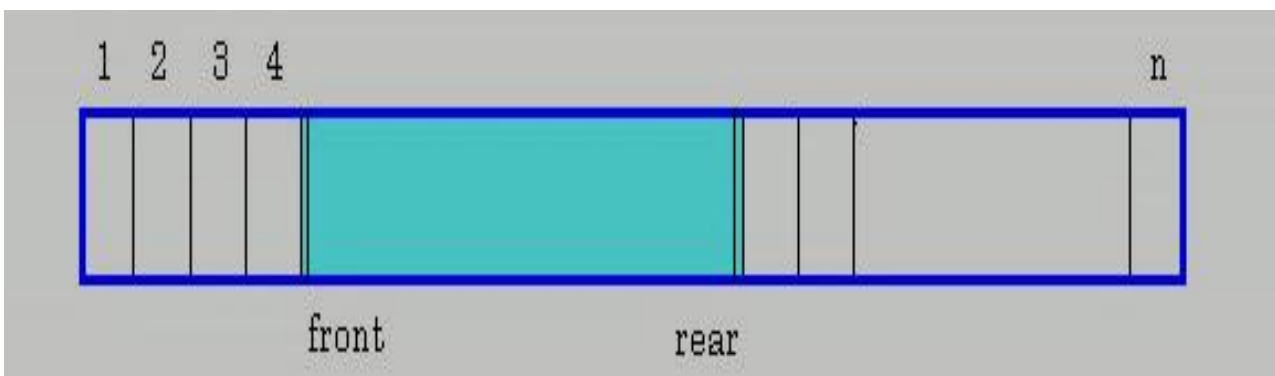
The atomic operations may be used to describe the standard stack, queue and deque operations.

Implementations

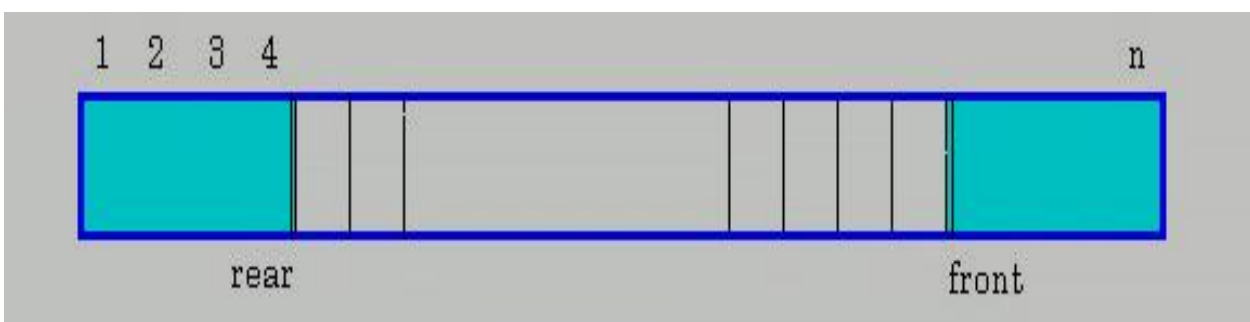
The ADTs can be implemented by various data structures.

1. Tables/Arrays:

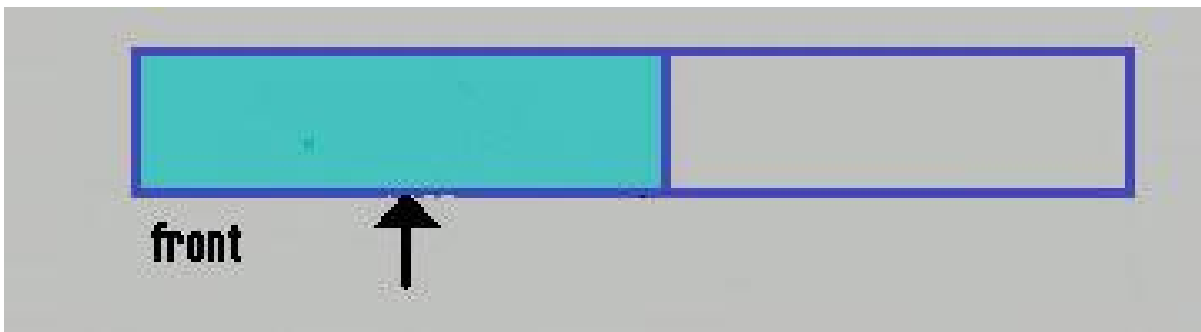
A static array can be used to implement the data structures. Consider, for example, a queue implemented as an array.



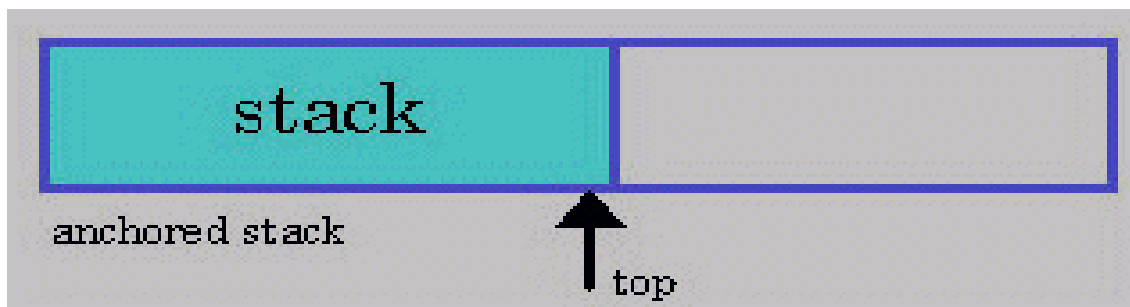
As data is added to the rear of the queue, the cell labelled 'rear' (the cell containing the last enqueued data) is incremented to the right. If data is dequeued, the cell labelled 'front' (the cell containing the first enqueued data) is also incremented to the right. Subsequently, the data could end up flanked by empty cells.



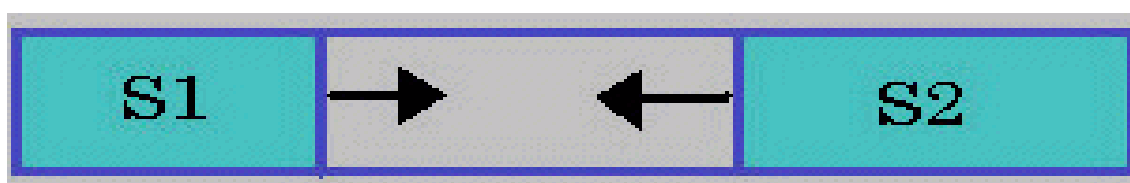
If the cell labelled 'rear' is the last cell in the array, AND there are empty cells at the beginning of the array (due to previous dequeues) then 'rear' will wrap to the beginning of the array on the next enqueue. This will result in data at the beginning and end of the array, with empty cells in the middle.



An ANCHORED LIST prevents this movement of data in the array. Data is always left skewed. In our queue example, if data is dequeued then the whole array must be shifted one cell to the left.



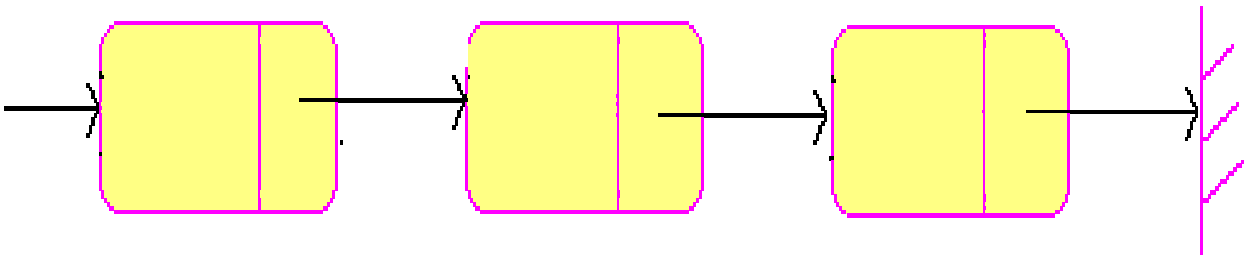
Anchored lists are more appropriate for stacks, where the non-anchored end of the list represents the top of the stack.



Two stacks can be anchored at opposite ends of an array. As the stacks fill with data, they will "grow" towards each other. The above figure illustrates this concept. By filling the array with the two stacks anchored at opposite ends, the user can have the utility of two stacks while using the storage of one array.

2. Linked Lists:

A series of structures connected with pointers can be used to implement the data structures. Each structure contains data with one or more pointers to neighbouring structures.



There are several variants of the linked list structure:

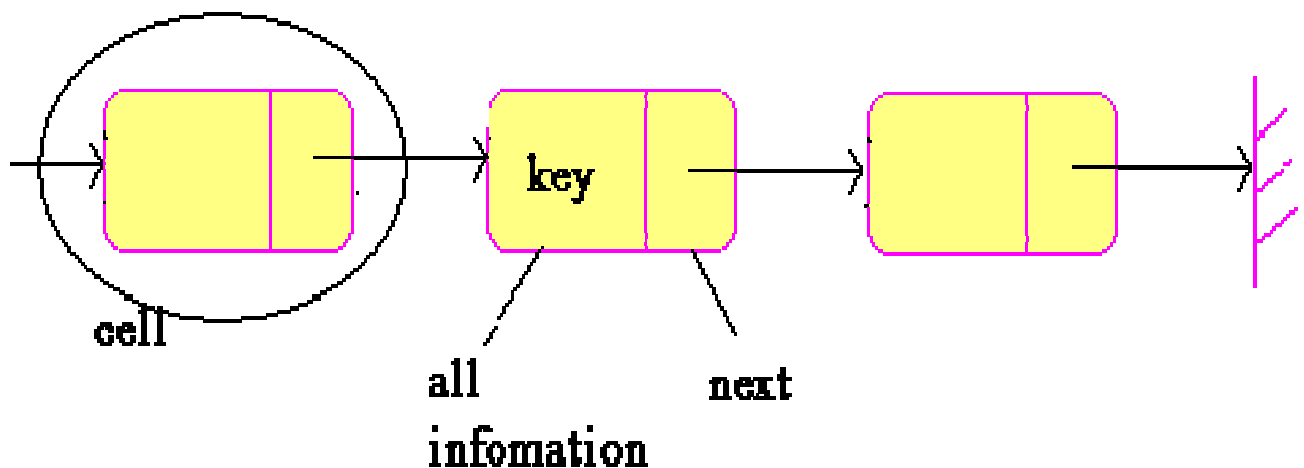
Endogenous / Exogenous lists:

- endogenous lists have the data stored in the structure's KEY. The KEY is data stored within the structure.**
- exogenous lists have the data stored outside the structure. Instead of a KEY, the structure has a pointer to the data in memory. Exogenous lists do**

not require data to be moved when individual cells are moved in a list; only the pointers to data must be changed. This can save considerable cost when dealing with large data in each cell. Another benefit of exogenous lists is many cells can point to the same data. Again, this may be useful depending on the application.

Here are example declarations of endogenous and exogenous structures:

```
struct endogenous { data_type key,  
    struct endogenous * next  
};  
struct exogenous { data_type * data,  
    struct exogenous * next  
};
```



Circular / Non-circular lists:

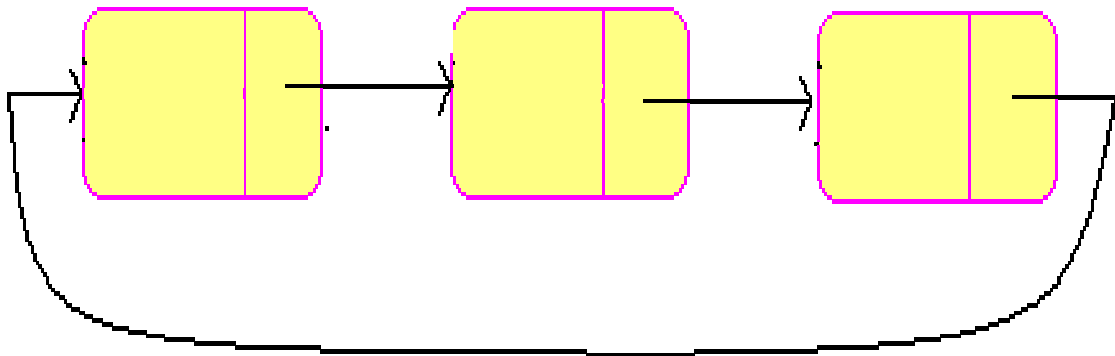
- a circular list has the last cell in the array pointing to the first cell in the array. Specifically, the last cell's 'next' pointer references the first cell.

Representation in C : last_cell.next = &first_cell

- the last cell in a non-circular list points to nothing.

Representation in C : last_cell.next = NULL

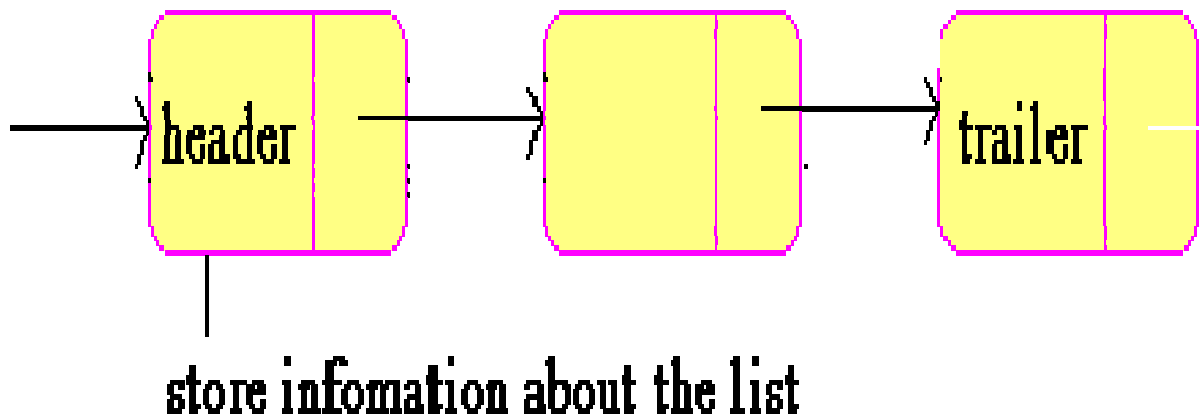
Circular lists are useful for representing polygons (for example) because one can trace a path continuously back to where one started. This is useful for representing a polygon because there is essentially no starting or ending point. Thus, we would like an implementation to illustrate this.



With/Without a Header/Trailer:

- a header node is a dummy first node in the list. It is not part of the data, but rather contains some information about the list (eg. size).

- a trailer node is at the end of a list (its contents marks the end).

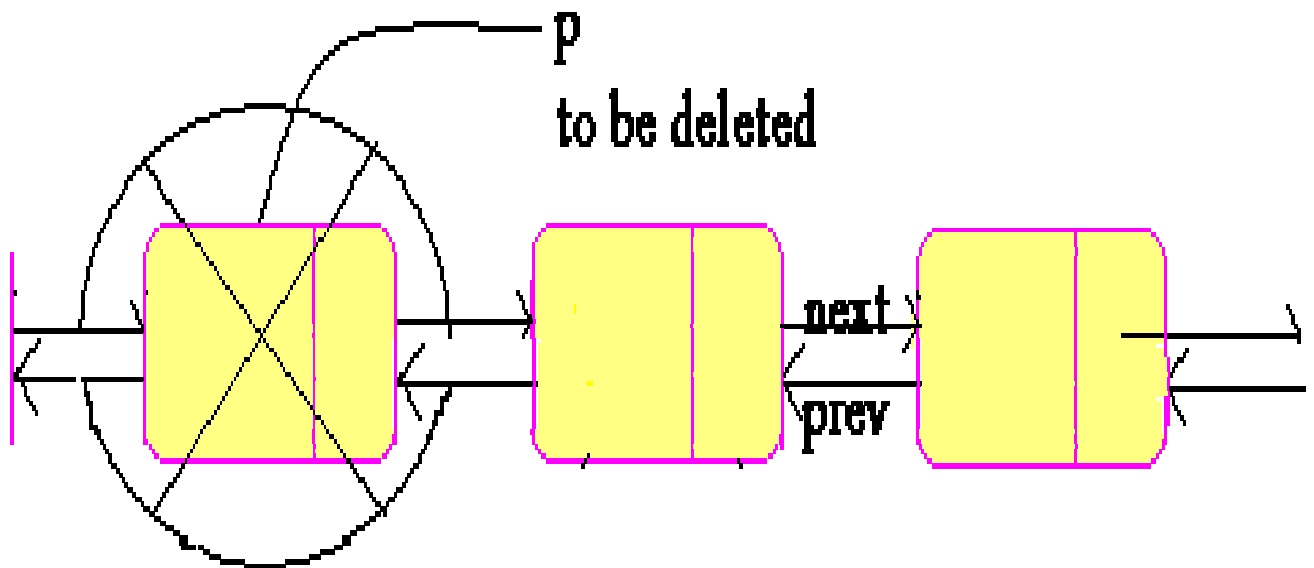


Doubly Linked List:

-each node in a doubly linked list is a structure with two pointers to link with neighbouring nodes. One pointer points to the next node in the list, and the other pointer points to the previous node in the array. This implementation is useful for deleting nodes. The algorithm can be performed in $O(1)$ time. Deleting nodes in a singly linked list can be done in $\Omega(n)$ time. Therefore, doubly linked lists can be very useful in applications requiring a lot of deletions. The pseudocode for the delete algorithm is as follows:

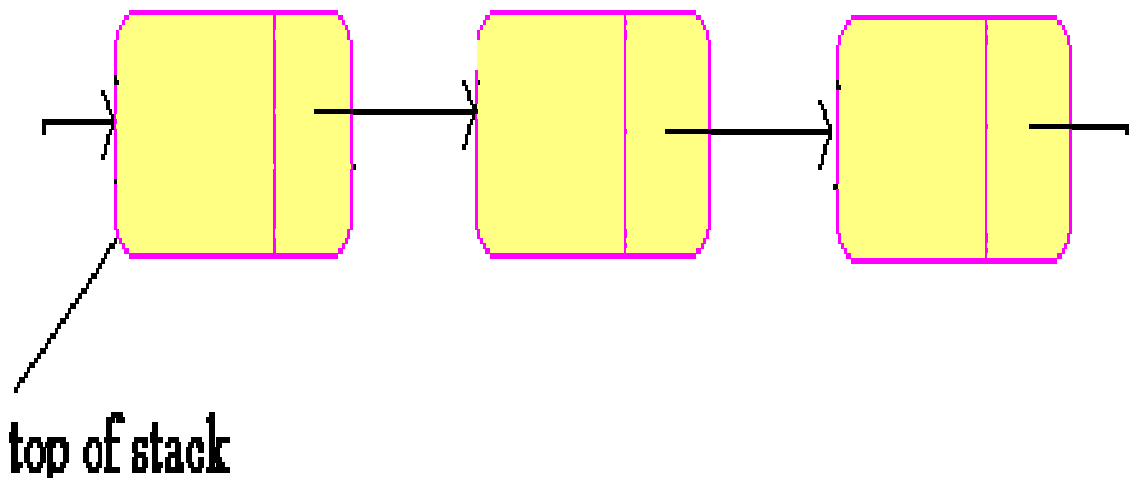
p -> prev -> next = p -> next

p-> next -> prev = p -> prev

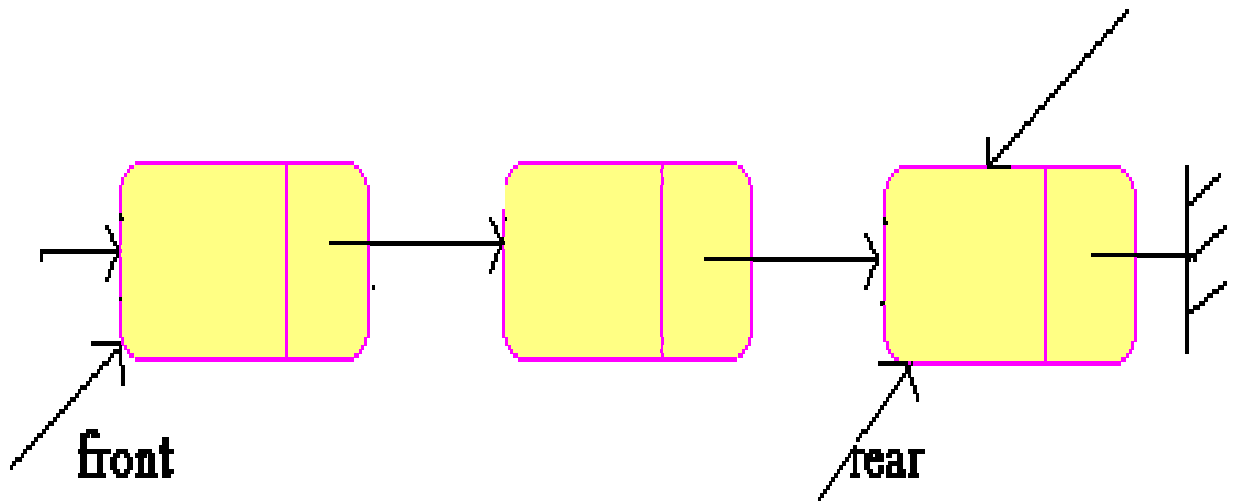


3. Utility of implementations:

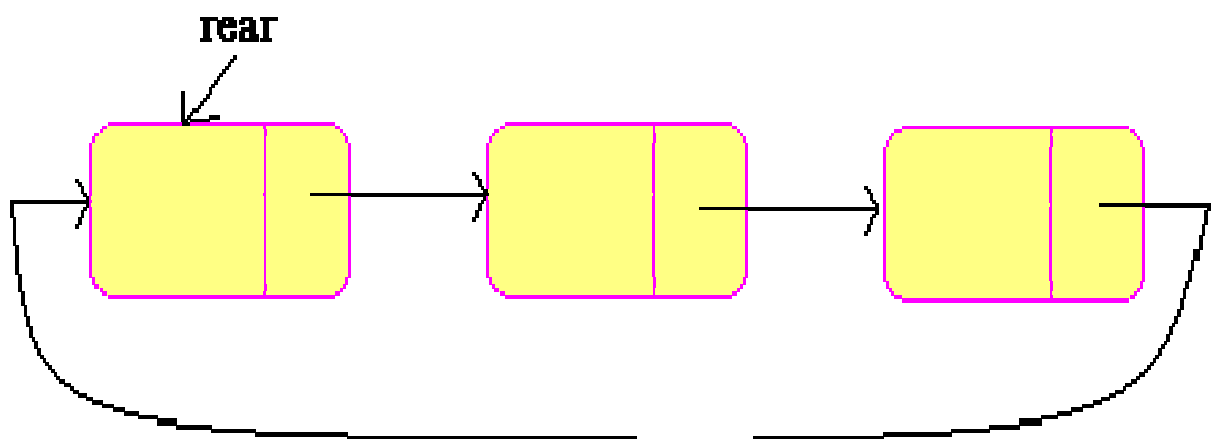
-Stacks can easily be implemented as a linked list. One needs a pointer to the top of the stack (which is the front of the list).



-Queues can easily be implemented as a linked list as well. One needs a pointer to the front of the queue (which is the front of the list); as well, one needs a pointer to the rear of the queue (which is the end of the list).



It is better, however, to implement a queue as a circular list. This circular list has a pointer to the list pointing at the rear rather than the front of the list.



-Concatenating sublists can easily be done with a linked list. Conversely, concatenating sublists with an array implementation is expensive.

-Referencing sublists with a linked list implementation is expensive. Conversely, sublists can be referenced easily with an array implementation.

Applications

1. Polynomial ADT:

A polynomial can be represented with primitive data structures. For example, a polynomial represented as $a_k x^k + a_{k-1} x^{k-1} + \dots + a_0$ can be represented as a linked list. Each node is a structure with two values: a_i and i . Thus, the length of the list will be k . The first node will have (a_k, k) , the second node will have $(a_{k-1}, k-1)$ etc. The last node will be $(a_0, 0)$.

The polynomial $3x^9 + 7x^3 + 5$ can be represented in a list as follows: $(3,9) \rightarrow (7,3) \rightarrow (5,0)$ where each pair of integers represent a node, and the arrow represents a link to its neighbouring node.

Derivatives of polynomials can be easily computed by proceeding node by node. In our previous example the list after computing the derivative would be represented as follows: $(27,8)$

--> (21,2). The specific polynomial ADT will define various operations, such as multiplication, addition, subtraction, derivative, integration etc. A polynomial ADT can be useful for symbolic computation as well.

2. Large Integer ADT:

Large integers can also be implemented with primitive data structures. To conform to our previous example, consider a large integer represented as a linked list. If we represent the integer as successive powers of 10, where the power of 10 increments by 3 and the coefficient is a three digit number, we can make computations on such numbers easier. For example, we can represent a very large number as follows:

$$513(10^6) + 899(10^3) + 722(10^0).$$

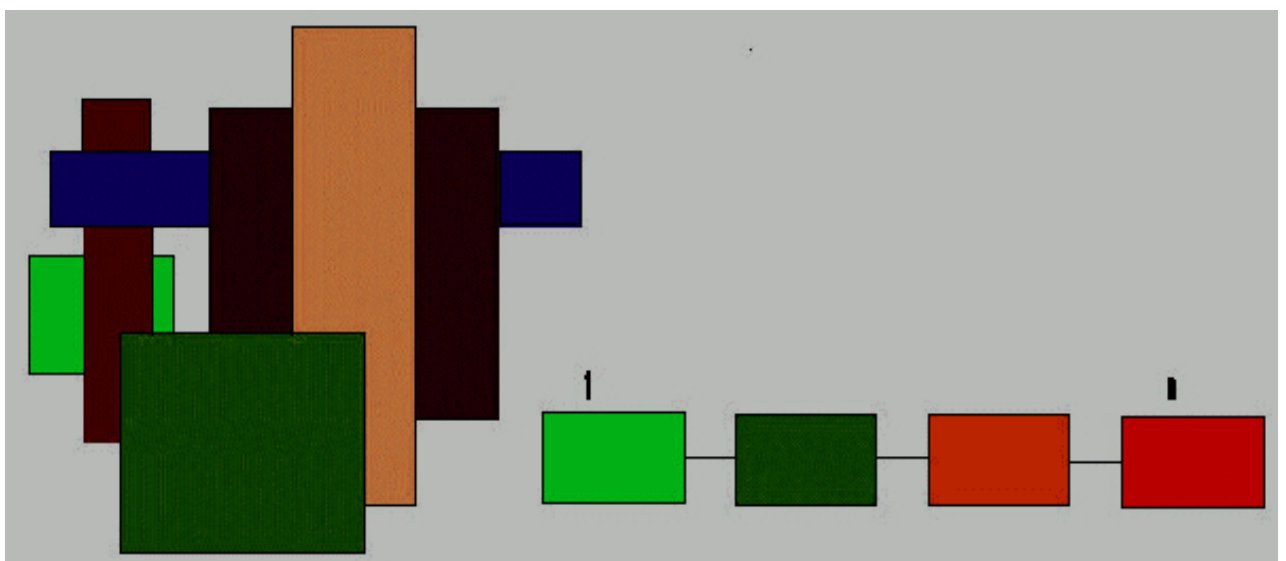
Using this notation, the number can be represented as follows:

$$(513) \text{ --> } (899) \text{ --> } (722).$$

The first number represents the coefficient of the 10^6 term, the next number represents the coefficient of the 10^3 term and so on. The arrows represent links to adjacent nodes. The specific ADT will define operations on this representation, such as addition, subtraction, multiplication, division, comparison, copy etc.

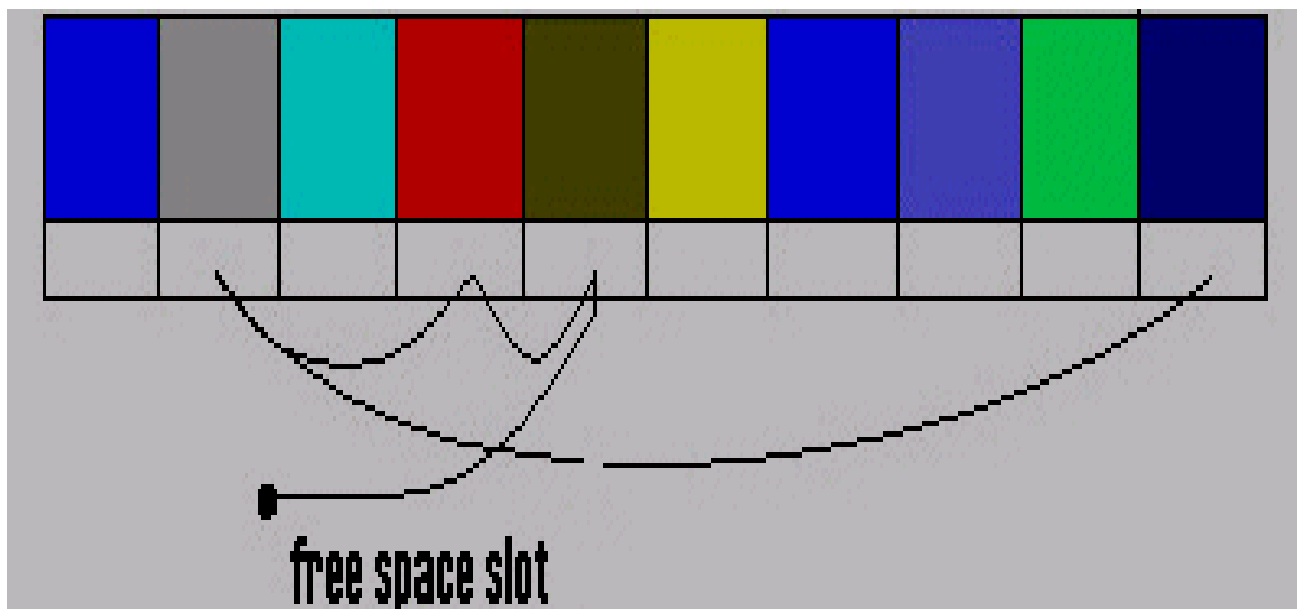
3. Window Manager ADT:

A window interface can be represented by lists. Consider an environment with many windows. The first node in the list could represent the current active window. Subsequent windows are further along the list. In other words, the n^{th} window corresponds to the n^{th} node in the list. The ADT can define several functions, such as `Find_first_window` which would bring a window clicked upon to the front of the list (make it active). Other functions could perform window deletion or creation.



4. Management of free space:

When memory is requested, a list of available blocks of memory might be useful. Again, a list could represent blocks in memory available to the user, with nodes containing pointers to these available blocks. The list can be used like a stack (LIFO). The last freed memory becomes the next available to the user. Such lists are called 'free space lists' or 'available space lists'. Since addition and deletion of nodes is at one end, these lists behave like stacks. All operations on free space lists can be done in $O(1)$ time.



Stacks

1. Stack based languages:

Expressions can be evaluated using a stack. Given an expression in a high-level language (for example, $(a + b) * c$) the compiler will transform this expression to postfix form. The postfix form of the above example is $ab + c *$, where a and b are operands, $+$ and $*$ are operators, and the expression is scanned left to right. The expression is pushed on the stack and evaluated as it is popped. The following algorithm illustrates the process:

```
makenull(S)
y <- POP(S)
read(char)
if (char) is operand{
  PUSH (char, S)
}
if (char) is operator{
  x <- POP(S)
  z <- evaluate "y char x"
  PUSH (z, S)
}
```

In the end the stack will hold one element (the result).

2. Text Editor:

A text editor can be implemented with a stack. Characters are pushed on a stack as the user enters text. Commands to delete one character or a command to delete a series of characters (for example, a sentence or a word) would also push a character on a stack. However, the character would be a unique identifier to know how many characters to delete. For example, an identifier to delete one character would pop the stack once. An identifier to delete a sentence would pop all characters until the stack is empty or a period is encountered.

3. Postscript:

Postscript is a full-fledged interpreted computer language in which all operations are done by accessing a stack. It is the language of choice for laser printers. For example, the Postscript section of code

1 2 3 4 5 6 ADD MUL SUB 7 ADD MUL ADD
represents:

1 2 3 4 5 6 + * - 7 + * +

which in turn represents:

$1 + (2 * ((3 - (4 * (5 + 6))) + (7)))$

Very much as in the stack-based language example, the expression can be evaluated from left to right. Expressions written in the form

given above are called postfix expressions. Their easy evaluation with the help of a stack makes them natural candidates for the organization of expressions by compilers.

4. Scratch pad:

**Stacks are used to write down instructions that you can not act on immediately. For example, future work to be done by the program, information that may be useful later, and so forth (just as with a scratch pad). An example of this is the rat-in-maze problem (see below). A stack can be used to solve the problem of traversing a maze. One must keep track of previously explored routes, or else an infinite loop could occur. For example, with no previous knowledge of exploring a specific route unsuccessfully, one can enter a path, find no solution to the maze, exit the path through the same route as entrance, then enter the same unsuccessful path all over again. This problem can be solved with the help of a stack. If we consider each step through a maze a cell, the following algorithm will traverse a maze successfully with the help of a stack 'S':
(For all cells)**

```

Visited(cell) <- false
S <- Start
Visited(start) <-true
While not EMPTY(S) do{
if TOP(S) has an empty adjacent square then
<Q< (TOP(S))="EMPTY" SQUARE< DD>
VISITED(q) <- true
if q = 'TARGET CELL' then stop
PUSH(q, S) /*S has your path*/
else POP(S)
}

```

5. Recursion:

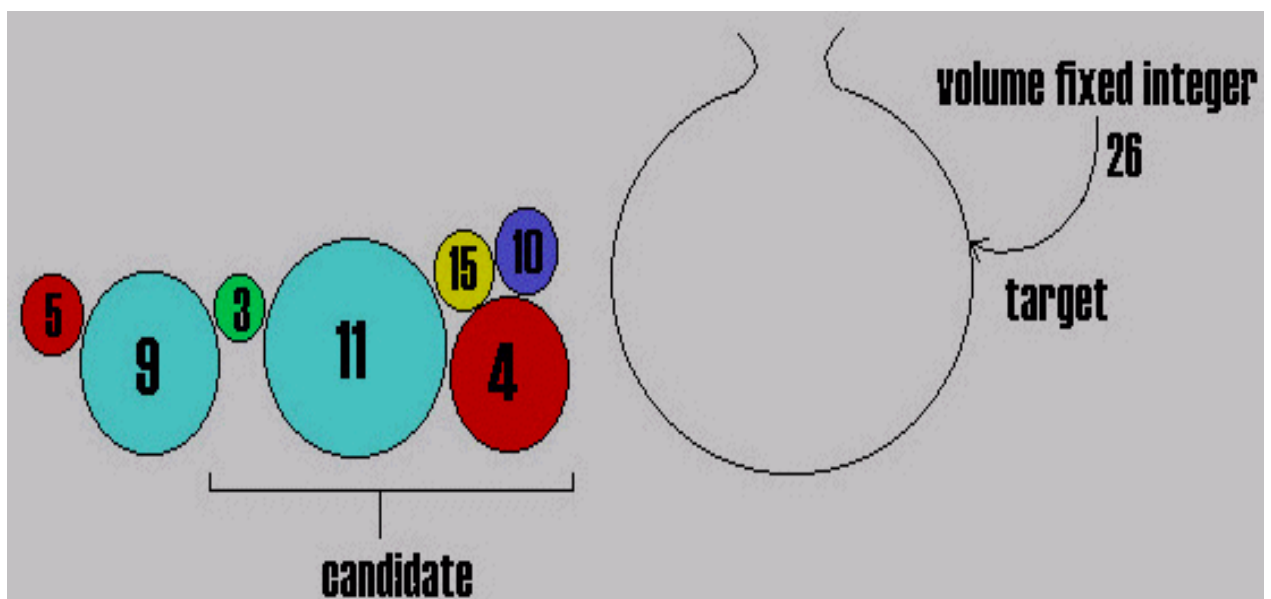
Stacks are used in recursions. Every recursive program can be rewritten iteratively using a stack. One related problem is the knapsack problem:

Consider a knapsack with volume represented as a fixed integer. One is given a series of items of varying size (the size of the objects is represented as an integer). The knapsack problem is to find a combination of items that will fit exactly into the knapsack (i.e. no unused space). The function call is written as 'knapsack(target: , candidate:)' where 'target' is the amount of space left in the sack, and 'candidate' is the reference to the item being

considered to be added. The function returns a boolean result; 'true' if target can be filled exactly using a subset of the items numbered "candidate, ..., n". Here 'n' is the total number of items. Define size[.] as an array of sizes of the items. The following is a recursive solution to the problem:

```
knapsack(target,candidate)
if target = 0 then return "true"
if candidate > n or target < 0 then return
"false"
if knapsack(target - size(candidate) , candidate
+ 1) then
return "true"
else return knapsack(target, candidate + 1)
```

A knapsack of size 26 can be filled with items of size 15 and 11.



UNIT-II

Divide and Conquer

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from $O(n^2)$ to $O(n \log n)$ to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide-and-conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

DANDC (P)

```
{
  if SMALL (P) then return S (p);
  else
  {
    divide p into smaller instances  $p_1, p_2, \dots, p_k, k \geq 1$ ;
    apply DANDC to each of these sub problems;
    return (COMBINE (DANDC ( $p_1$ ), DANDC ( $p_2$ ), ..., DANDC ( $p_k$ )));
  }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p_1, p_2, \dots, p_k are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, $T(n)$ is the time for DANDC on 'n' inputs

$g(n)$ is the time to complete the answer directly for small inputs and

$f(n)$ is the time for Divide and Combine

Binary Search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that $a[j] = x$ (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key $a[mid]$, and compare 'x' with $a[mid]$. If $x = a[mid]$ then the desired record has been found. If $x < a[mid]$ then 'x' must be in that portion of the file that precedes $a[mid]$, if there at all. Similarly, if $a[mid] > x$, then further search is only necessary in that part of the file which follows $a[mid]$. If we use recursive procedure of finding the middle key $a[mid]$ of the un-searched portion of a file, then every un-successful comparison of 'x' with $a[mid]$ will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and $a[mid]$, and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm Algorithm

BINSRCH (a, n, x)

```
// array a(1 : n) of elements in increasing order, n ≥ 0,
// determine whether 'x' is present, and if so, set j such that x = a(j)
// else return j
```

```
{
    low :=1 ; high :=n ;
    while (low ≤ high) do
    {
        mid :=|(low + high)/2|
        if (x < a [mid]) then high:=mid - 1;
        else if (x > a [mid]) then low:= mid + 1
        else return mid;
    }
    return 0;
}
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

<i>Index</i>	1	2	3	4	5	6	7	8	9
<i>Elements</i>	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101	low	high	mid
	1	9	5
	6	9	7
	8	9	8
	9	9	9
			found

Number of comparisons = 4

2. Searching for x = 82	low	high	mid
	1	9	5
	6	9	7
	8	9	8
			found

Number of comparisons = 3

3. Searching for x = 42	low	high	mid
	1	9	5
	6	9	7
	6	6	6
	7	6	not found

Number of comparisons = 4

4. Searching for x = -14	low	high	mid
	1	9	5
	1	4	2
	1	1	1
	2	1	not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

<i>Index</i>	1	2	3	4	5	6	7	8	9
<i>Elements</i>	-15	-6	0	7	9	23	54	82	101
<i>Comparisons</i>	3	2	3	4	1	3	2	3	4

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding $25/9$ or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If $x < a[1]$, $a[1] < x < a[2]$, $a[2] < x < a[3]$, $a[5] < x < a[6]$, $a[6] < x < a[7]$ or $a[7] < x < a[8]$ the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

$$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$$

The time complexity for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$.

Successful searches			un-successful searches
$\Theta(1)$, Best	$\Theta(\log n)$, average	$\Theta(\log n)$ worst	$\Theta(\log n)$ best, average and worst

Analysis for worst case

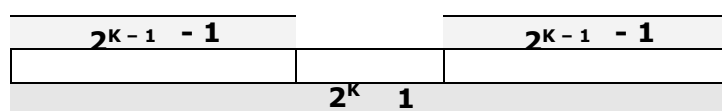
Let $T(n)$ be the time complexity of Binary search

The algorithm sets mid to $\lceil (n+1) / 2 \rceil$

Therefore,

$$\begin{aligned} T(0) &= 0 \\ T(n) &= 1 && \text{if } x = a[\text{mid}] \\ &= 1 + T(\lceil (n+1) / 2 \rceil - 1) && \text{if } x < a[\text{mid}] \\ &= 1 + T(n - \lceil (n+1) / 2 \rceil) && \text{if } x > a[\text{mid}] \end{aligned}$$

Let us restrict 'n' to values of the form $n = 2^k - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\lceil \frac{n+1}{2} \rceil = \lceil \frac{2^k - 1 + 1}{2} \rceil = 2^{k-1}$ for $k > 1$

$$\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor + 1}{2} \right\rfloor$$

Giving,

$$\begin{aligned} T(0) &= 0 \\ T(2^k - 1) &= 1 && \text{if } x = a[\text{mid}] \\ &= 1 + T(2^{k-1} - 1) && \text{if } x < a[\text{mid}] \\ &= 1 + T(2^{k-1} - 1) && \text{if } x > a[\text{mid}] \end{aligned}$$

In the worst case the test $x = a[\text{mid}]$ always fails, so

$$\begin{aligned} w(0) &= 0 \\ w(2^k - 1) &= 1 + w(2^{k-1} - 1) \end{aligned}$$

This is now solved by repeated substitution:

$$\begin{aligned}
 w(2^k - 1) &= 1 + w(2^{k-1} - 1) \\
 &= 1 + [1 + w(2^{k-2} - 1)] \\
 &= 1 + [1 + [1 + w(2^{k-3} - 1)]] \\
 &= \dots\dots\dots \\
 &= \dots\dots\dots \\
 &= i + w(2^{k-i} - 1)
 \end{aligned}$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^k - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

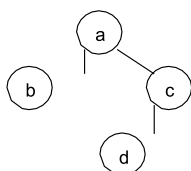
for $n = 2^k - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form $2^k - 1$ weakens the result. In practice this does not matter very much, $w(n)$ is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form $2^k - 1$.

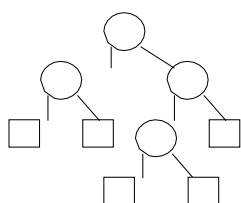
External and Internal path length:

The lines connecting nodes to their non-empty sub trees are called edges. A non-empty binary tree with n nodes has n-1 edges. The size of the tree is the number of nodes it contains.

When drawing binary trees, it is often convenient to represent the empty sub trees explicitly, so that they can be seen. For example:

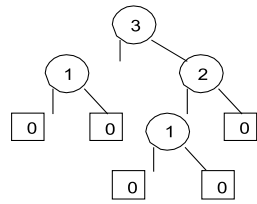


The tree given above in which the empty sub trees appear as square nodes is as follows:

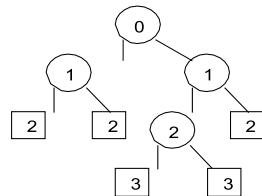


The square nodes are called as external nodes $E(T)$. The square node version is sometimes called an extended binary tree. The round nodes are called internal nodes $I(T)$. A binary tree with n internal nodes has n+1 external nodes.

The height $h(x)$ of node 'x' is the number of edges on the longest path leading down from 'x' in the extended tree. For example, the following tree has heights written inside its nodes:



The depth $d(x)$ of node 'x' is the number of edges on path from the root to 'x'. It is the number of internal nodes on this path, excluding 'x' itself. For example, the following tree has depths written inside its nodes:



The internal path length $I(T)$ is the sum of the depths of the internal nodes of 'T':

$$I(T) = \sum_{x \in I(T)} d(x)$$

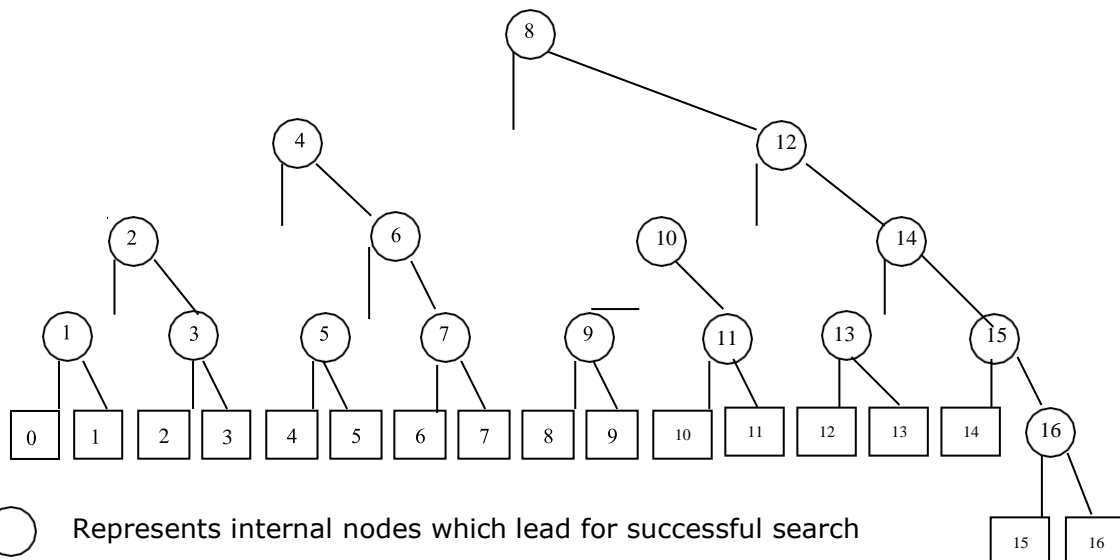
The external path length $E(T)$ is the sum of the depths of the external nodes:

$$E(T) = \sum_{x \in E(T)} d(x)$$

For example, the tree above has $I(T) = 4$ and $E(T) = 12$.

A binary tree T with 'n' internal nodes, will have $I(T) + 2n = E(T)$ external nodes.

A binary tree corresponding to binary search when $n = 16$ is



- Represents internal nodes which lead for successful search
- External square nodes, which lead for unsuccessful search.

Let C_N be the average number of comparisons in a successful search.

C'_N be the average number of comparison in an un successful search.

Then we have,

$$C_N = 1 + \frac{\text{internal path length of tree}}{N}$$

$$C'_N = \frac{\text{External path length of tree}}{N + 1}$$

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1$$

External path length is always 2N more than the internal path length.

Merge Sort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge sort in the *best case*, *worst case* and *average case* is $O(n \log n)$ and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays 'a' and 'b', an output array 'c', and three counters, *a ptr*, *b ptr* and *c ptr*, which are initially set to the beginning of their respective arrays. The smaller of $a[a \text{ ptr}]$ and $b[b \text{ ptr}]$ is copied to the next entry in 'c', and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to 'c'.

To illustrate how merge process works. For example, let us consider the array 'a' containing 1, 13, 24, 26 and 'b' containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
<i>h ptr</i>			

5	6	7	8
2	15	27	28
<i>j ptr</i>			

1	2	3	4	5	6	7	8
1							
<i>i ptr</i>							

and then 2 and 13 are compared. 2 is added to 'c'. Increment *b ptr* and *c ptr*.

1	2	3	4
1	13	24	26
	<i>h ptr</i>		

5	6	7	8
2	15	27	28
<i>j ptr</i>			

1	2	3	4	5	6	7	8
1	2						
	<i>i ptr</i>						

then 13 and 15 are compared. 13 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
	<i>h ptr</i>		

5	6	7	8
2	15	27	28
	<i>j ptr</i>		

1	2	3	4	5	6	7	8
1	2	13					
		<i>i ptr</i>					

24 and 15 are compared. 15 is added to 'c'. Increment *b ptr* and *c ptr*.

1	2	3	4
1	13	24	26
		<i>h ptr</i>	

5	6	7	8
2	15	27	28
	<i>j ptr</i>		

1	2	3	4	5	6	7	8
1	2	13	15				
			<i>i ptr</i>				

24 and 27 are compared. 24 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
		<i>h ptr</i>	

5	6	7	8
2	15	27	28
		<i>j ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24			
				<i>i ptr</i>			

26 and 27 are compared. 26 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
			<i>h ptr</i>

5	6	7	8
2	15	27	28
		<i>j ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24	26		
					<i>i ptr</i>		

As one of the lists is exhausted. The remainder of the b array is then copied to 'c'.

1	2	3	4
1	13	24	26
			<i>h ptr</i>

5	6	7	8
2	15	27	28
		<i>j ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24	26	27	28
							<i>i ptr</i>

Algorithm

```

Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
    if (low < high)
    {
        mid := |(low + high)/2|           //finds where to split the set
        MERGESORT(low, mid)              //sort one subset
        MERGESORT(mid+1, high)           //sort the other subset
        MERGE(low, mid, high)            // combine the results
    }
}
    
```

Algorithm MERGE (low, mid, high)
 // a (low : high) is a global array containing two sorted subsets
 // in a (low : mid) and in a (mid + 1 : high).
 // The objective is to merge these sorted sets into single sorted
 // set residing in a (low : high). An auxiliary array B is used.

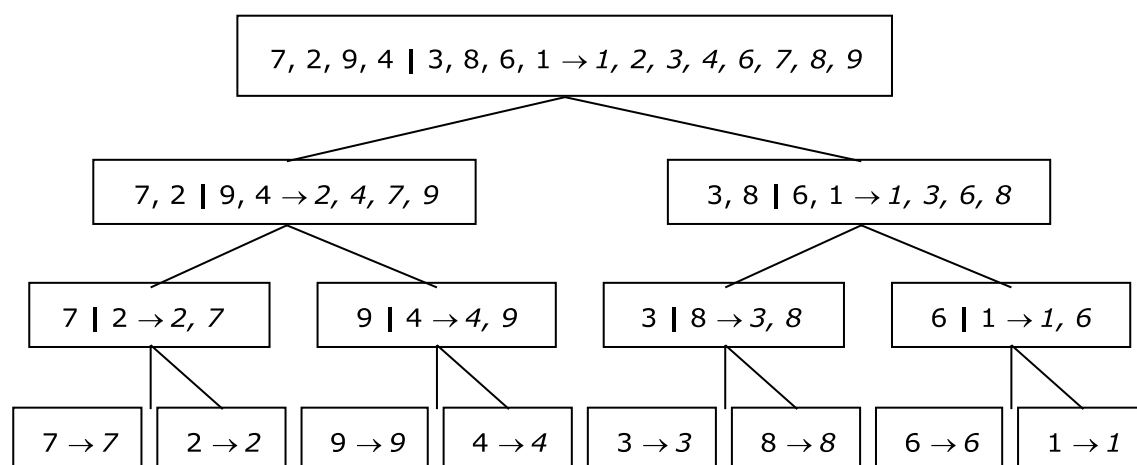
```

{
  h := low; i := low; j := mid + 1;
  while ((h ≤ mid) and (j ≤ high)) do
  {
    if (a[h] ≤ a[j]) then
    {
      b[i] := a[h]; h := h + 1;
    }
    else
    {
      b[i] := a[j]; j := j + 1;
    }
    i := i + 1;
  }
  if (h > mid) then
  for k := j to high do
  {
    b[i] := a[k]; i := i + 1;
  }
  else
  for k := h to mid do
  {
    b[i] := a[k]; i := i + 1;
  }
  for k := low to high do
  a[k] := b[k];
}

```

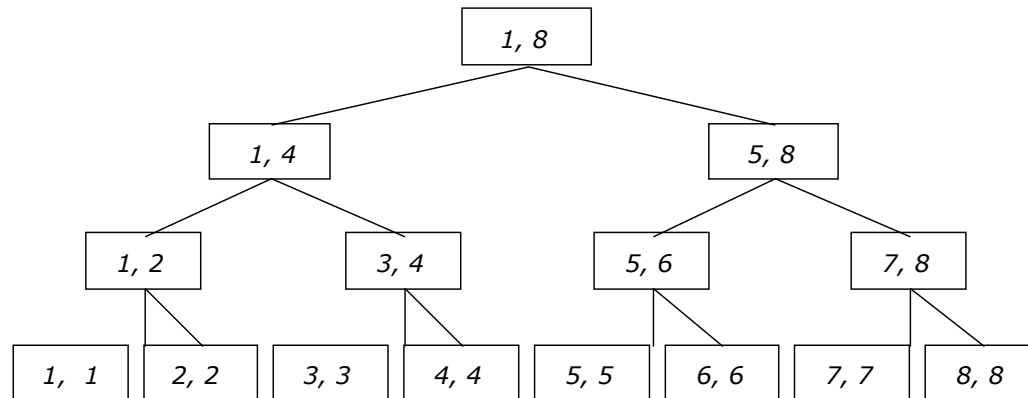
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:



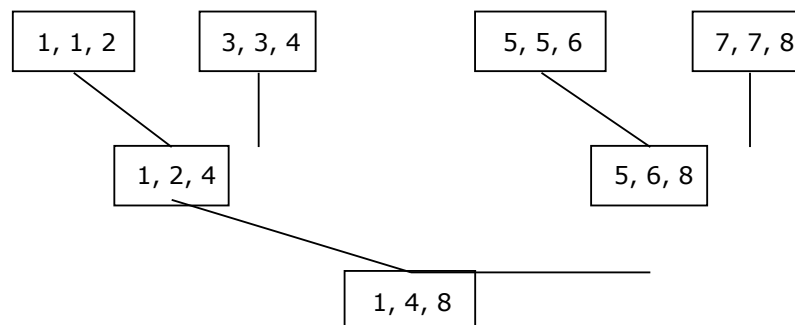
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

For $n = 1$, the time to merge sort is constant, which we will denote by 1. Otherwise, the time to merge sort 'n' numbers is equal to the time to do two recursive merge sorts of size $n/2$, plus the time to merge, which is linear. The equation says this exactly:

$$T(1) = 1$$

$$T(n) = 2 T(n/2) + n$$

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right-hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute $n/2$ into this main equation

$$\begin{aligned} 2 T(n/2) &= 2 (2 (T(n/4)) + n/2) \\ &= 4 T(n/4) + n \end{aligned}$$

We have,

$$\begin{aligned} T(n/2) &= 2 T(n/4) + n \\ T(n) &= 4 T(n/4) + 2n \end{aligned}$$

Again, by substituting $n/4$ into the main equation, we see that

$$\begin{aligned} 4T (n/4) &= 4 (2T(n/8)) + n/4 \\ &= 8 T(n/8) + n \end{aligned}$$

So we have,

$$\begin{aligned} T(n/4) &= 2 T(n/8) + n \\ T(n) &= 8 T(n/8) + 3n \end{aligned}$$

Continuing in this manner, we obtain:

$$T(n) = 2^k T(n/2^k) + K \cdot n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

$$\begin{aligned} T(n) &= 2^{\log_2 n} T\left(\frac{2^k}{2^k}\right) + \log_2 n \cdot n \\ &= n T(1) + n \log n \\ &= n \log n + n \end{aligned}$$

Representing this in O notation:

$$T(n) = \mathbf{O(n \log n)}$$

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is $O(n \log n)$, it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is $O(n \log n)$.*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two $n \times n$ matrices A and B, yielding result matrix 'C' as follows :

```
for i := 1 to n do
  for j :=1 to n do
    c[i, j] := 0;
    for K: = 1 to n do
      c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us consider three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then c_{ij} can be found by the usual matrix multiplication algorithm,

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

This leads to a divide-and-conquer algorithm, which performs $n \times n$ matrix multiplication by partitioning the matrices into quarters and performing eight $(n/2) \times (n/2)$ matrix multiplications and four $(n/2) \times (n/2)$ matrix additions.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 8 T(n/2) \end{aligned}$$

Which leads to $T(n) = O(n^3)$, where n is the power of 2.

Strassen's insight was to find an alternative method for calculating the C_{ij} , requiring seven $(n/2) \times (n/2)$ matrix multiplications and eighteen $(n/2) \times (n/2)$ matrix additions and subtractions:

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) B_{11}$$

$$R = A_{11} (B_{12} - B_{22})$$

$$S = A_{22} (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U.$$

This method is used recursively to perform the seven $(n/2) \times (n/2)$ matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 7 T(n/2) \end{aligned}$$

Solving this for the case of $n = 2^k$ is easy:

$$\begin{aligned} T(2^k) &= 7 T(2^{k-1}) \\ &= 7^2 T(2^{k-2}) \\ &= \text{-----} \\ &= \text{-----} \\ &= 7^i T(2^{k-i}) \end{aligned}$$

$$\begin{aligned} \text{Put } i = k & \\ &= 7^k T(1) \\ &= 7^k \end{aligned}$$

$$\begin{aligned} \text{That is, } T(n) &= 7^{\log_2 n} \\ &= n^{\log_2 7} \\ &= O(n^{\log_2 7}) = O(n^{2.81}) \end{aligned}$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence w_1, w_2, \dots, w_n take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

Hoare has devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is $O(n \log n)$.

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until $a[i] \geq \text{pivot}$.
- Repeatedly decrease the pointer 'j' until $a[j] \leq \text{pivot}$.

- If $j > i$, interchange $a[j]$ with $a[i]$
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition $low \geq high$ is satisfied. This condition will be satisfied only when the array is completely sorted.
- Here we choose the first element as the 'pivot'. So, $pivot = x[low]$. Now it calls the partition function to find the proper position j of the element $x[low]$ i.e. pivot. Then we will have two sub-arrays $x[low]$, $x[low+1], \dots \dots x[j-1]$ and $x[j+1], x[j+2], \dots \dots x[high]$.
- It calls itself recursively to sort the left sub-array $x[low], x[low+1], \dots \dots x[j-1]$ between positions low and $j-1$ (where j is returned by the partition function).
- It calls itself recursively to sort the right sub-array $x[j+1], x[j+2], \dots \dots x[high]$ between positions $j+1$ and $high$.

Algorithm Algorithm

QUICKSORT(low, high)

/* sorts the elements $a(low), \dots \dots, a(high)$ which reside in the global array $A(1 : n)$ into ascending order $a(n+1)$ is considered to be defined and must be greater than all elements in $a(1 : n)$; $A(n+1) = +\infty$ */

```
{
  if low < high then
  {
    j := PARTITION(a, low, high+1);
    // J is the position of the partitioning element
    QUICKSORT(low, j - 1);
    QUICKSORT(j + 1, high);
  }
}
```

Algorithm PARTITION(a, m, p)

```
{
  V ← a(m); i ← m; j ← p; // A (m) is the partition element
  do
  {
    loop i := i + 1 until a(i) ≥ v // i moves left to right
    loop j := j - 1 until a(j) ≤ v // p moves right to left
    if (i < j) then INTERCHANGE(a, i, j)
  } while (i ≥ j);
  a[m] := a[j]; a[j] := V; // the partition element belongs at position P
  return j;
}
```

Algorithm INTERCHANGE(a, i, j)

```
{
    P:=a[i];
    a[i] := a[j];
    a[j] := p;
}
```

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	2	3	4	5	6	7	8	9	10	11	12	13	Remarks
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j
					02			57					
						j	i						
(24	08	16	06	04	02)	38	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	24								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	08	(16)									swap pivot & j
	pivot, j	i											
	(04)	06											swap pivot & j
	04												
	pivot, j, i												
				16									
				pivot, j, i									
(02	04	06	08	16	24)	38							
							(56	57	58	79	70	45)	

							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	56	(58	79	70	57)	swap pivot & j
							45						swap pivot & j
									(58	79	70	57)	swap i & j
									pivot	i		j	
										57		79	
										j	i		
									(57)	58	(70	79)	swap pivot & j
									57				
												(70	79)
											pivot,	i	swap pivot & j
											j		
											70		
												79	
												pivot,	
												j, i	
								(45	56	57	58	70	79)
02	04	06	08	16	24	38	45	56	57	58	70	79	

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take $T(0) = T(1) = 1$, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + Cn \quad - \quad (1)$$

Where, $i = |S_1|$ is the number of elements in S_1 .

Worst Case Analysis

The pivot is the smallest element, all the time. Then $i=0$ and if we ignore $T(0)=1$, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + Cn \quad n > 1 \quad - \quad (2)$$

Using equation - (1) repeatedly, thus

$$T(n - 1) = T(n - 2) + C(n - 1)$$

$$T(n - 2) = T(n - 3) + C(n - 2)$$

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

$$\begin{aligned}
 T(n) &= T(1) + \sum_{i=2}^n i \\
 &= O(n^2)
 \end{aligned}
 \tag{3}$$

Best Case Analysis

In the best case, the pivot is in the middle. To simplify the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big - oh answer.

$$T(n) = 2 T(n/2) + C n \tag{4}$$

Divide both sides by n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + C \tag{5}$$

Substitute n/2 for 'n' in equation (5)

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + C \tag{6}$$

Substitute n/4 for 'n' in equation (6)

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + C \tag{7}$$

Continuing in this manner, we obtain:

$$\frac{T(2)}{2} = \frac{T(1)}{1} + C \tag{8}$$

We add all the equations from 4 to 8 and note that there are log n of them:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + C \log n \tag{9}$$

$$\text{Which yields, } T(n) = C n \log n + n = O(n \log n) \tag{10}$$

This is exactly the same analysis as merge sort, hence we get the same answer.

Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

$$T(n) = \text{comparisons for first call on quicksort} \\ + \\ \left\{ \sum_{1 \leq n_{\text{left}}, n_{\text{right}} \leq n} [T(n_{\text{left}}) + T(n_{\text{right}})] \right\} n = (n+1) + 2 [T(0) + T(1) + T(2) + \dots + T(n-1)]/n$$

$$nT(n) = n(n+1) + 2 [T(0) + T(1) + T(2) + \dots + T(n-2) + T(n-1)]$$

$$(n-1)T(n-1) = (n-1)n + 2 [T(0) + T(1) + T(2) + \dots + T(n-2)]$$

Subtracting both sides:

$$nT(n) - (n-1)T(n-1) = [n(n+1) - (n-1)n] + 2T(n-1) = 2n + 2T(n-1)$$

$$nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)$$

$$T(n) = 2 + (n+1)T(n-1)/n$$

The recurrence relation obtained is:

$$T(n)/(n+1) = 2/(n+1) + T(n-1)/n$$

Using the method of substitution:

$$T(n)/(n+1) = 2/(n+1) + T(n-1)/n$$

$$T(n-1)/n = 2/n + T(n-2)/(n-1)$$

$$T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)$$

$$T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)$$

$$\cdot$$

$$\cdot$$

$$\cdot$$

$$T(3)/4 = 2/4 + T(2)/3$$

$$T(2)/3 = 2/3 + T(1)/2 \quad T(1)/2 = 2/2 + T(0)$$

Adding both sides:

$$T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + \dots + T(2)/3 + T(1)/2]$$

$$= [T(n-1)/n + T(n-2)/(n-1) + \dots + T(2)/3 + T(1)/2] + T(0) +$$

$$[2/(n+1) + 2/n + 2/(n-1) + \dots + 2/4 + 2/3]$$

Cancelling the common terms:

$$T(n)/(n+1) = 2[1/2 + 1/3 + 1/4 + \dots + 1/n + 1/(n+1)]$$

$$T(n) = (n+1)2 \left[\sum_{2 \leq k \leq n+1} 1/k \right]$$

$$= 2(n+1) [\quad - \quad]$$

$$= 2(n+1)[\log(n+1) - \log 2]$$

$$= 2n \log(n+1) + \log(n+1) - 2n \log 2 - \log 2$$

$$\mathbf{T(n) = O(n \log n)}$$

3.8. Straight insertion sort:

Straight insertion sort is used to create a sorted list (initially list is empty) and at each iteration the top number on the sorted list is removed and put into its proper

place in the sorted list. This is done by moving along the sorted list, from the smallest to the largest number, until the correct place for the new number is located i.e. until all sorted numbers with smaller values comes before it and all those with larger values comes after it. For example, let us consider the following 8 elements for sorting:

<i>Index</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>
<i>Elements</i>	27	412	71	81	59	14	273	87

Solution:

Iteration 0:	unsorted	412	71	81	59	14	273	87	
	Sorted	27							
Iteration 1:	unsorted	412	71	81	59	14	273	87	
	Sorted	27	412						
Iteration 2:	unsorted	71	81	59	14	273	87		
	Sorted	27	71	412					
Iteration 3:	unsorted	81	39	14	273	87			
	Sorted	27	71	81	412				
Iteration 4:	unsorted	59	14	273	87				
	Sorted	274	59	71	81	412			
Iteration 5:	unsorted	14	273	87					
	Sorted	14	27	59	71	81	412		
Iteration 6:	unsorted	273	87						
	Sorted	14	27	59	71	81	273	412	
Iteration 7:	unsorted	87							
	Sorted	14	27	59	71	81	87	273	412

UNIT

3

Greedy Method

GENERAL METHOD

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm*.

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm*.

CONTROL ABSTRACTION

Algorithm Greedy (a, n)

```
// a(1 : n) contains the 'n' inputs
{
    solution := ;           // initialize the solution to empty
    for i:=1 to n do
    {
        x := select (a);
        if feasible (solution, x) then
            solution := Union (Solution, x);
    }
    return solution;
}
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from 'a', removes it and assigns its value to 'x'. Feasible is a Boolean valued function, which determines if 'x' can be included into the solution vector. The function Union combines 'x' with solution and updates the objective function.

KNAPSACK PROBLEM

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight w_i and the knapsack has a capacity 'm'. If a fraction x_i , $0 < x_i < 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'. The problem is stated as:

$$\begin{aligned} & \text{maximize} \quad \sum_{i=1}^n p_i x_i \\ & \text{subject to} \quad \sum_{i=1}^n w_i x_i \leq M \quad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n \end{aligned}$$

The profits and weights are positive numbers.

Algorithm

If the objects are already been sorted into non-increasing order of $p[i] / w[i]$ then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that  $p[i] / w[i] > p[i + 1] / w[i + 1]$ .
// m is the knapsack size and x[1: n] is the solution vector.
{
    for i := 1 to n do x[i] := 0.0           // initialize x
    U := m;
    for i := 1 to n do
    {
        if (w(i) > U) then break;
        x [i] := 1.0; U := U - w[i];
    }
    if (i ≤ n) then x[i] := U / w[i];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only $O(n)$ time.

Example:

Consider the following instance of the knapsack problem: $n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

1. First, we try to fill the knapsack by selecting the objects in some order:

x_1	x_2	x_3	$w_i x_i$	$p_i x_i$
1/2	1/3	1/4	$18 \times 1/2 + 15 \times 1/3 + 10 \times 1/4 = 16.5$	$25 \times 1/2 + 24 \times 1/3 + 15 \times 1/4 = 24.25$

2. Select the object with the maximum profit first ($p = 25$). So, $x_1 = 1$ and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit ($p = 24$). So, $x_2 = 2/15$

x_1	x_2	x_3	$w_i x_i$	$p_i x_i$
1	2/15	0	$18 \times 1 + 15 \times 2/15 = 20$	$25 \times 1 + 24 \times 2/15 = 28.2$

3. Considering the objects in the order of non-decreasing weights w_i .

x_1	x_2	x_3	$w_i x_i$	$p_i x_i$
0	2/3	1	$15 \times 2/3 + 10 \times 1 = 20$	$24 \times 2/3 + 15 \times 1 = 31$

4. Considered the objects in the order of the ratio p_i / w_i .

p_1/w_1	p_2/w_2	p_3/w_3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / w_i . Select the object with the maximum p_i / w_i ratio, so, $x_2 = 1$ and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / w_i ratio, so $x_3 = 1/2$ and the profit earned is 7.5.

x_1	x_2	x_3	$w_i x_i$	$p_i x_i$
0	1	1/2	$15 \times 1 + 10 \times 1/2 = 20$	$24 \times 1 + 15 \times 1/2 = 31.5$

This solution is the optimal solution.

JOB SEQUENCING WITH DEADLINES

When we are given a set of 'n' jobs. Associated with each Job i , deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit p_i is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array $d [1 : n]$ is used to store the deadlines of the order of their p-values. The set of jobs $j [1 : k]$ such that $j [r], 1 \leq r \leq k$ are the jobs in 'j' and $d (j [1]) \leq d (j [2]) \leq \dots \leq d (j [k])$. To test whether $J \cup \{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d [J[r]] \leq r, 1 \leq r \leq k+1$.

Example:

Let $n = 4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	OPTIMAL
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

Algorithm GreedyJob (d, J, n)

// J is a set of jobs that can be completed by their deadlines.

```
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J U {i} can be completed by their dead lines)
            then J := J U {i};
    }
}
```

OPTIMAL MERGE PATTERNS

Given 'n' sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge 'n' sorted files together. This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly $n + m$ record moves, the obvious choice choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{
    treenode * lchild;
    treenode * rchild;
};
```

Algorithm TREE (n)

```
// list is a global of n single node binary trees
{
    for i := 1 to n - 1 do
    {
        pt new treenode
        (pt lchild) least (list); // merge two trees with smallest
lengths
        (pt rchild) least (list);
        (pt weight) ((pt lchild) weight) + ((pt rchild) weight);
        insert (list, pt);
    }
    return least (list); // The tree left in list is the merge
tree
}
```

Example 1:

Suppose we are having three sorted files X_1 , X_2 and X_3 of length 30, 20, and 10 records each. Merging of the files can be carried out as follows:

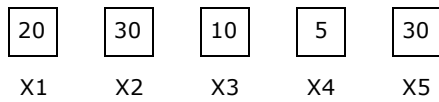
S.No	First Merging	Record moves in first merging	Second merging	Record moves in second merging	Total no. of records moves
1.	$X_1 \& X_2 = T_1$	50	$T_1 \& X_3$	60	$50 + 60 = 110$
2.	$X_2 \& X_3 = T_1$	30	$T_1 \& X_1$	60	$30 + 60 = 90$

The Second case is optimal.

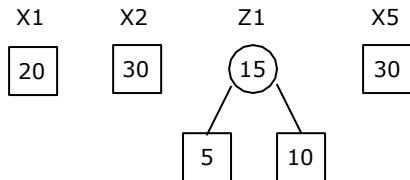
Example 2:

Given five files (X_1, X_2, X_3, X_4, X_5) with sizes (20, 30, 10, 5, 30). Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.

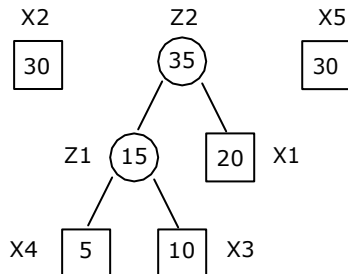
Solution:



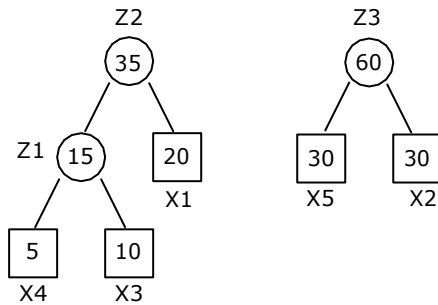
Merge X_4 and X_3 to get 15 record moves. Call this Z_1 .



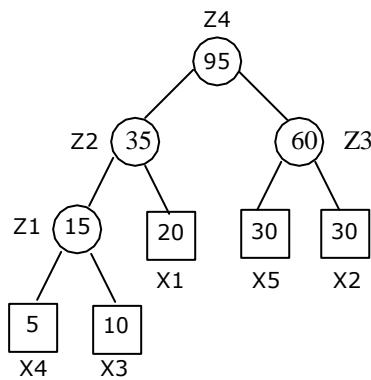
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2 .



Merge X_2 and X_5 to get 60 record moves. Call this Z_3 .



Merge Z_2 and Z_3 to get 90 record moves. This is the answer. Call this Z_4 .



Therefore the total number of record moves is $15 + 35 + 60 + 95 = 205$. This is an optimal merge pattern for the given problem.

Huffman Codes

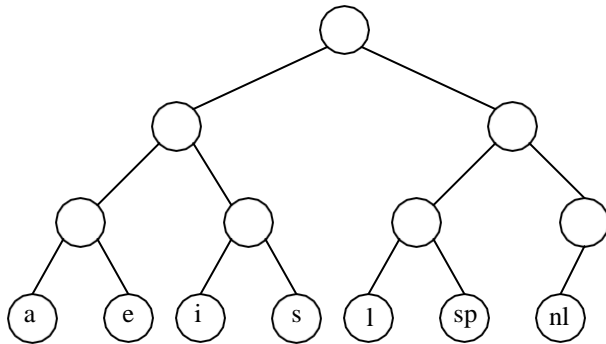
Another application of Greedy Algorithm is file compression.

Suppose that we have a file only with characters a, e, i, s, t, spaces and new lines, the frequency of appearance of a's is 10, e's fifteen, twelve i's, three s's, four t's, thirteen banks and one newline.

Using a standard coding scheme, for 58 characters using 3 bits for each character, the file requires 174 bits to represent. This is shown in table below.

<u>Character</u>	<u>Code</u>	<u>Frequency</u>	<u>Total bits</u>
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	9
T	100	4	12
Space	101	13	39
New line	110	1	3

Representing by a binary tree, the binary code for the alphabets are as follows:

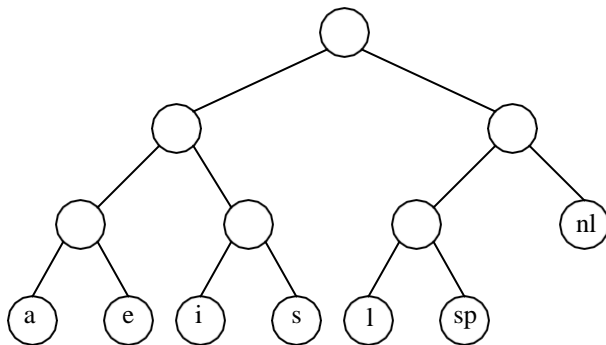


The representation of each character can be found by starting at the root and recording the path. Use a 0 to indicate the left branch and a 1 to indicate the right branch.

If the character c_i is at depth d_i and occurs f_i times, the cost of the code is equal to $d_i f_i$

With this representation the total number of bits is $3 \times 10 + 3 \times 15 + 3 \times 12 + 3 \times 3 + 3 \times 4 + 3 \times 13 + 3 \times 1 = 174$

A better code can be obtained by with the following representation.



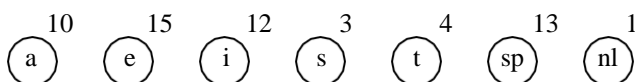
The basic problem is to find the full binary tree of minimal total cost. This can be done by using Huffman coding (1952).

Huffman's Algorithm:

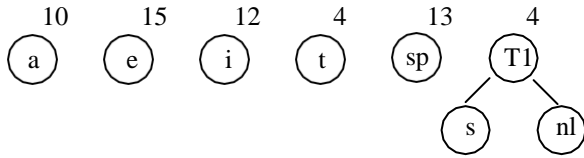
Huffman's algorithm can be described as follows: We maintain a forest of trees. The weights of a tree is equal to the sum of the frequencies of its leaves. If the number of characters is 'c'. $c - 1$ times, select the two trees T_1 and T_2 , of smallest weight, and form a new tree with sub-trees T_1 and T_2 . Repeating the process we will get an optimal Huffman coding tree.

Example:

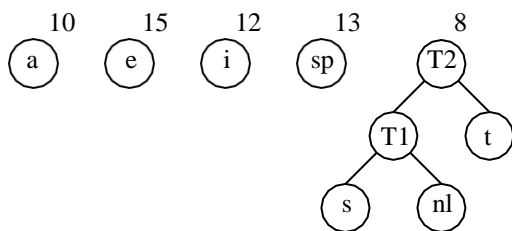
The initial forest with the weight of each tree is as follows:



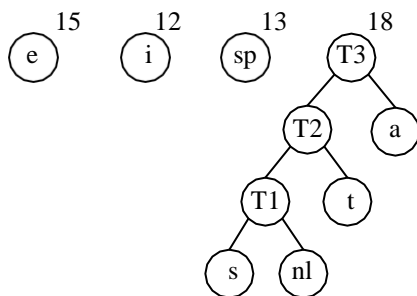
The two trees with the lowest weight are merged together, creating the forest, the Huffman algorithm after the first merge with new root T_1 is as follows: The total weight of the new tree is the sum of the weights of the old trees.



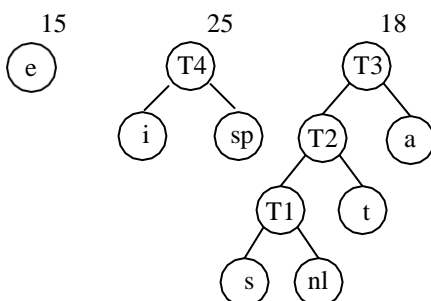
We again select the two trees of smallest weight. This happens to be T_1 and t , which are merged into a new tree with root T_2 and weight 8.



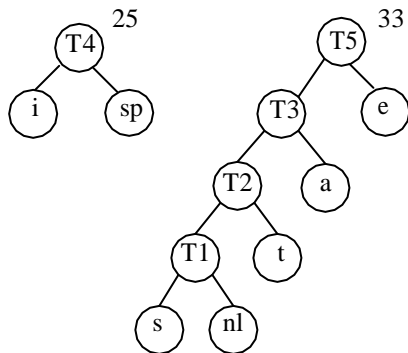
In next step we merge T_2 and a creating T_3 , with weight $10+8=18$. The result of this operation is



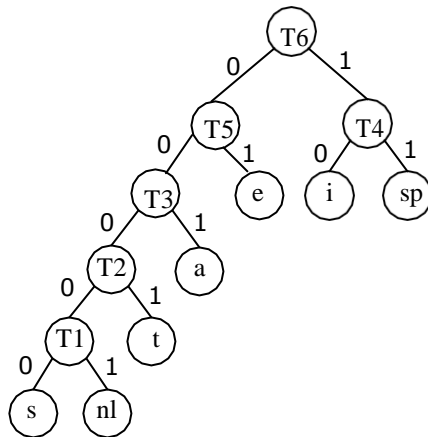
After third merge, the two trees of lowest weight are the single node trees representing i and the blank space. These trees merged into the new tree with root T_4 .



The fifth step is to merge the trees with roots e and T₃. The results of this step is



Finally, the optimal tree is obtained by merging the two remaining trees. The optimal trees with root T₆ is:



The full binary tree of minimal total cost, where all characters are obtained in the leaves, uses only 146 bits.

Character	Code	Frequency	Total bits (Code bits X frequency)
A	001	10	30
E	01	15	30
I	10	12	24
S	00000	3	15
T	0001	4	16
Space	11	13	26
New line	00001	1	5
		Total :	146

GRAPH ALGORITHMS

Basic Definitions:

Graph G is a pair (V, E) , where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote $n := |V|$, $m := |E|$.

Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If $(u, v) \in E$, then vertices u , and v are adjacent.

We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called **weighted**.

Degree of a vertex v is the number of vertices u for which $(u, v) \in E$ (denote $\text{deg}(v)$). The number of **incoming edges** to a vertex v is called **in-degree** of the vertex (denote $\text{indeg}(v)$). The number of **outgoing edges** from a vertex is called **out-degree** (denote $\text{outdeg}(v)$).

Representation of Graphs:

Consider graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$.

Adjacency matrix represents the graph as an $n \times n$ matrix $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

We may consider various modifications. For example for weighted graphs, we may have

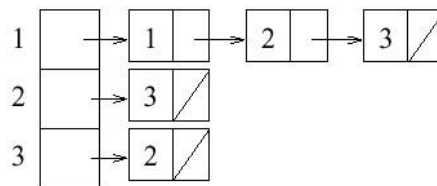
$$a_{i,j} = \begin{cases} w(v_i, v_j), & \text{if } (v_i, v_j) \in E, \\ \text{default}, & \text{otherwise,} \end{cases}$$

Where default is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then default can be ∞ , meaning value larger than any other value).

Adjacency List: An array $\text{Adj}[1 \dots n]$ of pointers where for $1 \leq v \leq n$, $\text{Adj}[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.

	1	2	3
1	1	1	1
2	0	0	1
3	0	1	0

Adjacency matrix



Adjacency list

Paths and Cycles:

A path is a sequence of vertices (v_1, v_2, \dots, v_k) , where for all i , $(v_i, v_{i+1}) \in E$. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices $(v_1, v_2, \dots, v_k, v_{k+1} = v_1)$, where for all i , $(v_i, v_{i+1}) \in E$ and all vertices in the cycle are distinct except pair v_1, v_{k+1} .

Subgraphs and Spanning Trees:

Subgraphs: A graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v . If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G . A single graph can have multiple spanning trees.

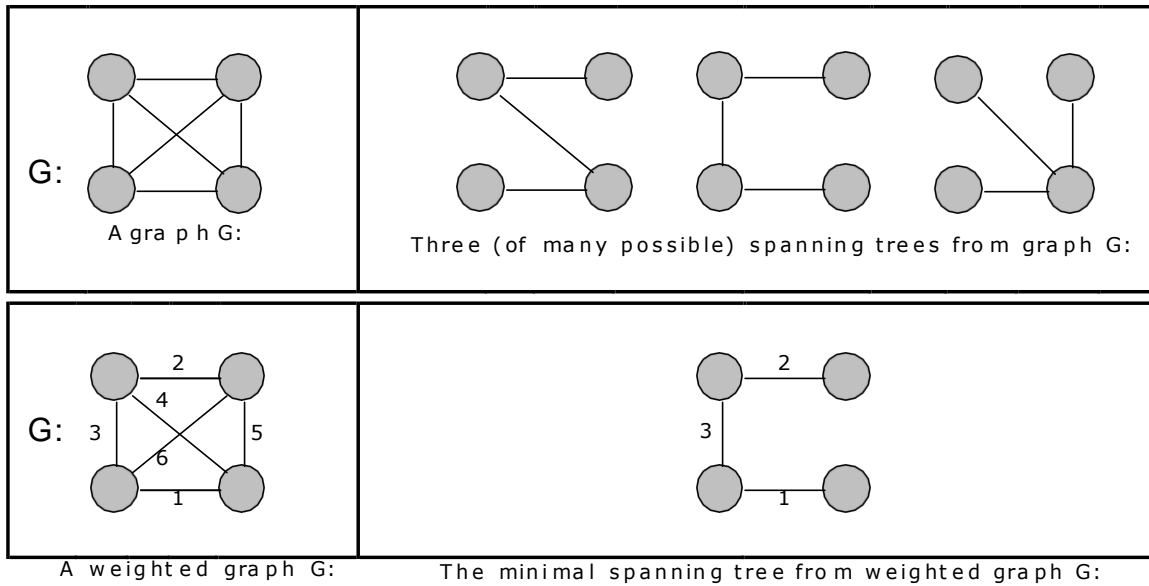
Lemma 1: *Let T be a spanning tree of a graph G . Then*

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T , then T becomes disconnected.*
3. *If we add any edge into T , then the new graph will contain a cycle.*
4. *Number of edges in T is $n-1$.*

Minimum Spanning Trees (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T . The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.



Here are some examples:

To explain further upon the Minimum Spanning Tree, and what it applies to, let's consider a couple of real-world examples:

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

Algorithm Kruskal (E, cost, n, t)

```
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
    Construct a heap out of the edge costs using heapify;
    for i := 1 to n do parent [i] := -1;
                                                // Each vertex is in a different set.

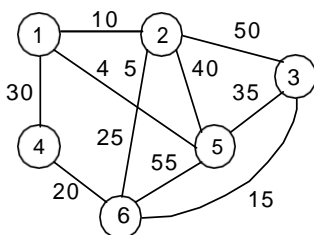
    i := 0; mincost := 0.0;
    while ((i < n -1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap and
        re-heapify using Adjust;
        j := Find (u); k := Find (v);
        if (j ≠ k) then
        {
            i := i + 1;
            t [i, 1] := u; t [i, 2] := v;
            mincost := mincost + cost [u, v];
            Union (j, k);
        }
    }
    if (i = n-1) then write ("no spanning tree");
    else return mincost;
}
```

Running time:

The number of finds is at most $2e$, and the number of unions at most $n-1$. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than $O(n + e)$.

We can add at most $n-1$ edges to tree T . So, the total time for operations on T is $O(n)$.





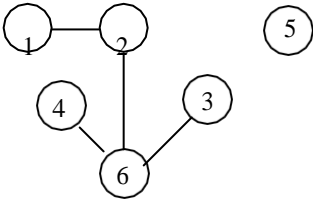
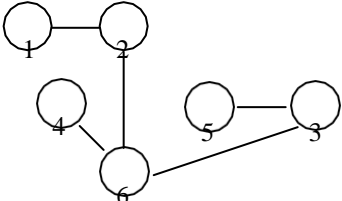
Summing up the various components of the computing times, we get $O(n + e \log e)$ as asymptotic complexity

Example 1:

Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by a set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need to check whether u and v are in the same vertex set. If so, then a cycle is created. If not then no cycle is created. Hence two **Finds** on the vertex sets suffice. When an edge is included in T, two components are combined into one and a **union** is to be performed on the two sets.

Edge	Cost	Spanning Forest	Edge Sets	Remarks
			{1}, {2}, {3}, {4}, {5}, {6}	
(1, 2)	10	1 2 	{1, 2}, {3}, {4}, {5}, {6}	The vertices 1 and 2 are in different sets, so the edge is combined
(3, 6)	15	1 2 3  6	{1, 2}, {3, 6}, {4}, {5}	The vertices 3 and 6 are in different sets, so the edge is combined
(4, 6)	20	1 2 3  4 6	{1, 2}, {3, 4, 6}, {5}	The vertices 4 and 6 are in different sets, so the edge is combined
(2, 6)	25		{1, 2, 3, 4, 6}, {5}	The vertices 2 and 6 are in different sets, so the edge is combined
(1, 4)	30	Reject		The vertices 1 and 4 are in the same set, so the edge is rejected
(3, 5)	35		{1, 2, 3, 4, 5, 6}	The vertices 3 and 5 are in the same set, so the edge is combined

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

Algorithm Algorithm Prim

```

(E, cost, n, t)
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j] is
// either a positive real number or if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost [k, l];
    t [1, 1] := k; t [1, 2] := l;
    for i :=1 to n do // Initialize near
        if (cost [i, l] < cost [i, k]) then near [i] := l;
        else near [i] := k;
    near [k] :=near [l] := 0;
    for i:=2 to n - 1 do // Find n - 2 additional edges for t.
    {
        Let j be an index such that near [j] 0 and
        cost [j, near [j]] is minimum;
        t [i, 1] := j; t [i, 2] := near [j];
        mincost := mincost + cost [j, near [j]];
        near [j] := 0
        for k:= 1 to n do // Update near[.
            if ((near [k] 0) and (cost [k, near [k]] > cost [k, j]))
                then near [k] := j;
    }
    return mincost;
}

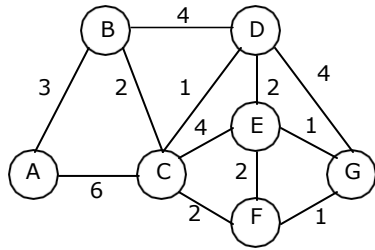
```

Running time:

We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get $O(n^2)$ time when we implement dist with array, $O(n + E \log n)$ when we implement it with a heap.

EXAMPLE 1:

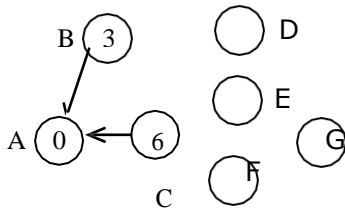
Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



SOLUTION:

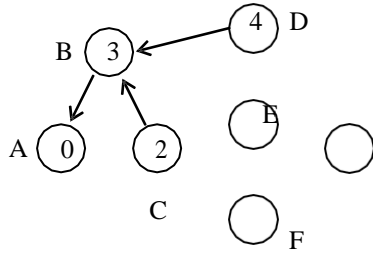
The stepwise progress of the prim's algorithm is as follows:

Step 1:



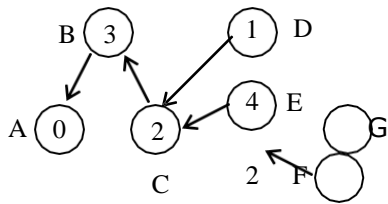
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6				
Next	*	A	A	A	A	A	A

Step 2:



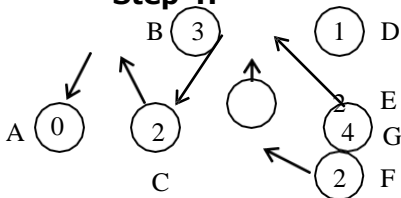
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4			
Next	*	A	B	B	A	A	A

Step 3:



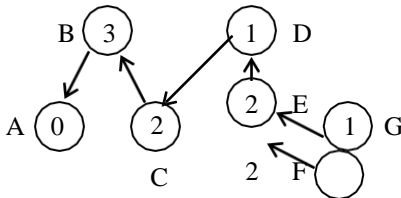
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	2
Next	*	A	B	C	C	C	A

Step 4:



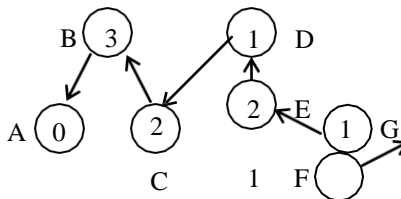
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



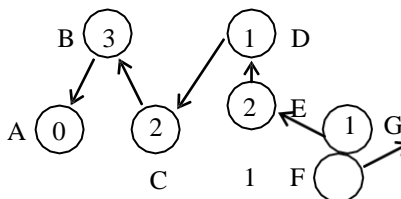
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

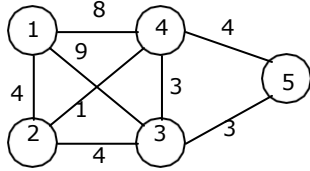
Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

EXAMPLE 2:

Considering the following graph, find the minimal spanning tree using prim's algorithm.

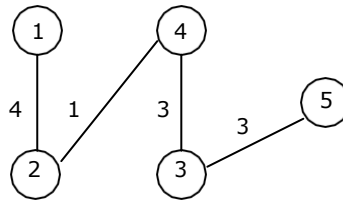


The cost adjacent matrix is

	4	9	8	
4	4	1		
9	4	3	3	
8	1	3	4	
		3	4	

The minimal spanning tree obtained as:

Vertex 1	Vertex 2
2	4
3	4
5	3
1	2



The cost of Minimal spanning tree = 11.

The steps as per the algorithm are as follows:

Algorithm near (J) = k means, the nearest vertex to J is k.

The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (2, 4).

$$K = 2, l = 4$$

$$\text{Min cost} = \text{cost}(2, 4) = 1$$

$$T[1, 1] = 2$$

$$T[1, 2] = 4$$

<p>for i = 1 to 5</p> <p>Begin</p> <p>i = 1 is cost (1, 4) < cost (1, 2) 8 < 4, No Than near (1) = 2</p> <p>i = 2 is cost (2, 4) < cost (2, 2) 1 < , Yes So near [2] = 4</p> <p>i = 3 is cost (3, 4) < cost (3, 2) 1 < 4, Yes So near [3] = 4</p> <p>i = 4 is cost (4, 4) < cost (4, 2) < 1, no So near [4] = 2</p> <p>i = 5 is cost (5, 4) < cost (5, 2) 4 < , yes So near [5] = 4</p> <p>end</p> <p>near [k] = near [l] = 0 near [2] = near[4] = 0</p>	<p style="text-align: center;">Near matrix</p> <table border="1" style="margin: 5px auto;"> <tr><td>2</td><td></td><td></td><td></td><td></td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> </table> <table border="1" style="margin: 5px auto;"> <tr><td>2</td><td>4</td><td></td><td></td><td></td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> </table> <table border="1" style="margin: 5px auto;"> <tr><td>2</td><td>4</td><td>4</td><td></td><td></td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> </table> <table border="1" style="margin: 5px auto;"> <tr><td>2</td><td>4</td><td>4</td><td>2</td><td></td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> </table> <table border="1" style="margin: 5px auto;"> <tr><td>2</td><td>4</td><td>4</td><td>2</td><td>4</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> </table> <table border="1" style="margin: 5px auto;"> <tr><td>2</td><td>0</td><td>4</td><td>0</td><td>4</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> </table>	2					1	2	3	4	5	2	4				1	2	3	4	5	2	4	4			1	2	3	4	5	2	4	4	2		1	2	3	4	5	2	4	4	2	4	1	2	3	4	5	2	0	4	0	4	1	2	3	4	5	<p style="text-align: center;">Edges added to min spanning tree:</p> <p style="text-align: center;">T [1, 1] = 2 T [1, 2] = 4</p>
2																																																														
1	2	3	4	5																																																										
2	4																																																													
1	2	3	4	5																																																										
2	4	4																																																												
1	2	3	4	5																																																										
2	4	4	2																																																											
1	2	3	4	5																																																										
2	4	4	2	4																																																										
1	2	3	4	5																																																										
2	0	4	0	4																																																										
1	2	3	4	5																																																										
<p>for i = 2 to n-1 (4) do</p> <p><u>i = 2</u></p> <p>for j = 1 to 5 j = 1 near(1)0 and cost(1, near(1)) 2 0 and cost (1, 2) = 4</p> <p>j = 2 near (2) = 0</p> <p>j = 3 is near (3) 0 4 0 and cost (3, 4) = 3</p>																																																														

$j = 4$
 $\text{near}(4) = 0$

$J = 5$
 Is $\text{near}(5) = 0$
 $4 = 0$ and $\text{cost}(4, 5) = 4$

select the min cost from the above obtained costs, which is 3 and corresponding $J = 3$

$\text{min cost} = 1 + \text{cost}(3, 4)$
 $= 1 + 3 = 4$

$T(2, 1) = 3$
 $T(2, 2) = 4$

$\text{Near}[j] = 0$
 i.e. $\text{near}(3) = 0$

for ($k = 1$ to n)

$K = 1$
 is $\text{near}(1) = 0$, yes
 $2 = 0$
 and $\text{cost}(1, 2) > \text{cost}(1, 3)$
 $4 > 9$, No

$K = 2$
 Is $\text{near}(2) = 0$, No

$K = 3$
 Is $\text{near}(3) = 0$, No

$K = 4$
 Is $\text{near}(4) = 0$, No

$K = 5$
 Is $\text{near}(5) = 0$
 $4 = 0$, yes
 and is $\text{cost}(5, 4) > \text{cost}(5, 3)$
 $4 > 3$, yes
 than $\text{near}(5) = 3$

$i = 3$

for ($j = 1$ to 5)

$J = 1$
 is $\text{near}(1) = 0$
 $2 = 0$
 $\text{cost}(1, 2) = 4$

$J = 2$
 Is $\text{near}(2) = 0$, No

2	0	0	0	4
1	2	3	4	5

2	0	0	0	3
1	2	3	4	5

$T(2, 1) = 3$
 $T(2, 2) = 4$

J = 3
 Is near (3) 0, no
 Near (3) = 0

J = 4
 Is near (4) 0, no
 Near (4) = 0

J = 5
 Is near (5) 0
 Near (5) = 3 3 0, yes
 And cost (5, 3) = 3

Choosing the min cost from
 the above obtaining costs
 which is 3 and corresponding J
 = 5

Min cost = 4 + cost (5, 3)
 = 4 + 3 = 7

T (3, 1) = 5
 T (3, 2) = 3

Near (J) = 0 near (5) = 0

for (k=1 to 5)

k = 1
 is near (1) 0, yes
 and cost(1,2) > cost(1,5)
 4 > , No

K = 2
 Is near (2) 0 no

K = 3
 Is near (3) 0 no

K = 4
 Is near (4) 0 no

K = 5
 Is near (5) 0 no

i = 4

for J = 1 to 5

J = 1
 Is near (1) 0
 2 0, yes
 cost (1, 2) = 4

j = 2
 is near (2) 0, No

T (3, 1) = 5
 T (3, 2) = 3

2	0	0	0	0
1	2	3	4	5

J = 3
 Is near (3) 0, No
 Near (3) = 0

J = 4
 Is near (4) 0, No
 Near (4) = 0

J = 5
 Is near (5) 0, No
 Near (5) = 0

Choosing min cost from the above it is only '4' and corresponding J = 1

Min cost = 7 + cost (1,2)
 = 7+4 = 11

T (4, 1) = 1
 T (4, 2) = 2

Near (J) = 0 Near (1) = 0

for (k = 1 to 5)

K = 1
 Is near (1) 0, No

K = 2
 Is near (2) 0, No

K = 3
 Is near (3) 0, No

K = 4
 Is near (4) 0, No

K = 5
 Is near (5) 0, No

End.

0	0	0	0	0
---	---	---	---	---

1 2 3 4 5

T (4, 1) = 1
 T (4, 2) = 2

4.8.7. The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

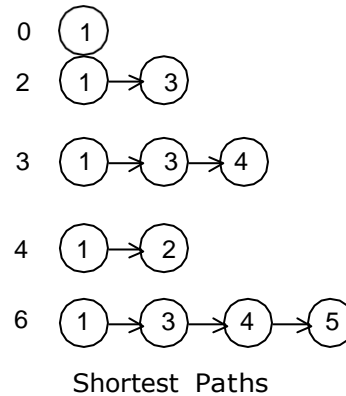
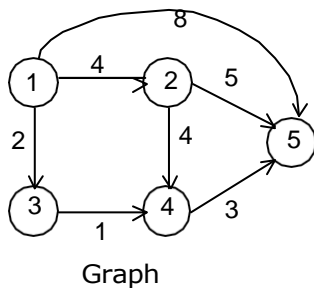
In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the

shortest path between them (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.



Algorithm:

Algorithm Shortest-Paths (v, cost, dist, n)

```

// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{
  for i :=1 to n do
  {
    S [i] := false; // Initialize S.
    dist [i] :=cost [v, i];
  }
  S[v] := true; dist[v] := 0.0; // Put v in S.
  for num := 2 to n - 1 do
  {
    Determine n - 1 paths from v.
    Choose u from among those vertices not in S such that dist[u] is minimum;
    S[u] := true; // Put u in S.
    for (each w adjacent to u with S [w] = false) do
      if (dist [w] > (dist [u] + cost [u, w]) then // Update distances
        dist [w] := dist [u] + cost [u, w];
    }
  }
}
  
```

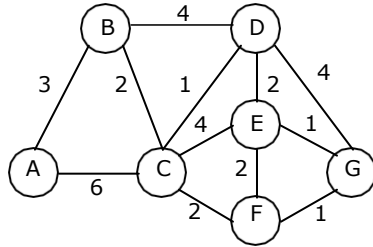
Running time:

Depends on implementation of data structures for dist.

- Build a structure with n elements A
- at most m = E times decrease the value of an item mB
- 'n' times select the smallest value nC
- For array A = O (n); B = O (1); C = O (n) which gives O (n²) total.
- For heap A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.

Example 1:

Use Dijkstra's algorithm to find the shortest path from A to each of the other six vertices in the graph:



Solution:

The cost adjacency matrix is

0	3	6	-	-	-	-
3	0	2	4	-	-	-
6	2	0	1	4	2	-
4	1	0	2	-	-	4
-	4	2	0	2	-	1
-	-	2	-	2	0	1
-	-	-	4	1	1	0

Here - means infinite

The problem is solved by considering the following information:

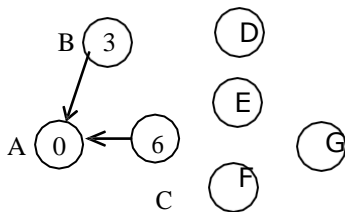
Status[v] will be either '0', meaning that the shortest path from v to v₀ has definitely been found; or '1', meaning that it hasn't.

Dist[v] will be a number, representing the length of the shortest path from v to v₀ found so far.

Next[v] will be the first vertex on the way to v₀ along the shortest path found so far from v to v₀

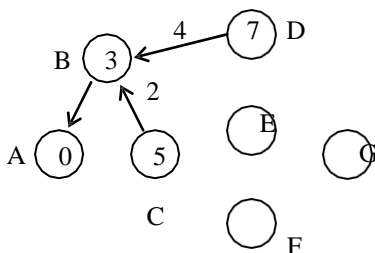
The progress of Dijkstra's algorithm on the graph shown above is as follows:

Step 1:



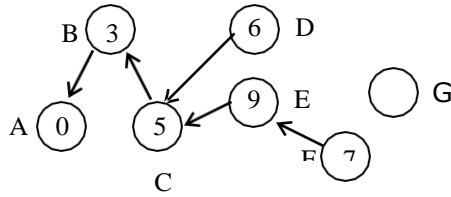
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	-	-	-	-
Next	*	A	A	A	A	A	A

Step 2:



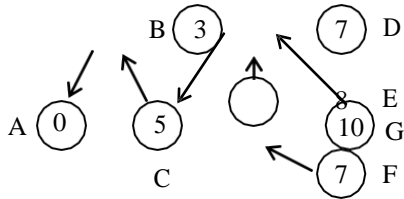
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7	-	-	-
Next	*	A	B	B	A	A	A

Step 3:



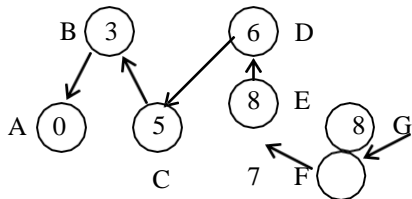
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	
Next	*	A	B	C	C	C	A

Step 4:



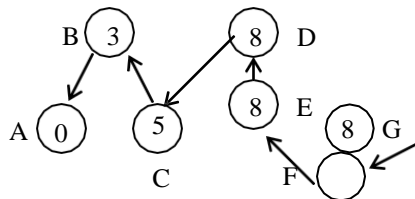
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step 5:



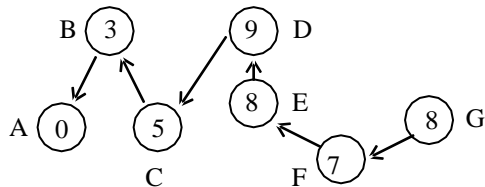
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

To resolve the question by a careful enumeration of solutions via trial and error, continued Gauss, would take only an hour or two. Apparently such inelegant work held little attraction for Gauss, for he does not seem to have carried it out, despite outlining in detail how to go about it.

— Paul Campbell, “Gauss and the Eight Queens Problem: A Study in Miniature of the Propagation of Historical Error” (1977)

I dropped my dinner, and ran back to the laboratory. There, in my excitement, I tasted the contents of every beaker and evaporating dish on the table. Luckily for me, none contained any corrosive or poisonous liquid.

— Constantine Fahlberg on his discovery of saccharin, *Scientific American* (1886)

7 Backtracking

In this lecture, I want to describe another recursive algorithm strategy called **backtracking**. A backtracking algorithm tries to build a solution to a computational problem incrementally. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it simply tries all possible options recursively.

7.1 n Queens

The prototypical backtracking problem is the classical **n Queens Problem**, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym “Schachfreund”) for the standard 8×8 board and by François-Joseph Eustache Lionnet in 1869 for the more general $n \times n$ board. The problem is to place n queens on an $n \times n$ chessboard, so that no two queens can attack each other. For readers not familiar with the rules of chess, this means that no two queens are in the same row, column, or diagonal.

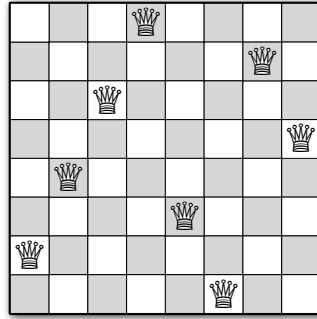
Obviously, in any solution to the n -Queens problem, there is exactly one queen in each row. So we will represent our possible solutions using an array $Q[1..n]$, where $Q[i]$ indicates which square in row i contains a queen, or 0 if no queen has yet been placed in row i . To find a solution, we put queens on the board row by row, starting at the top. A *partial* solution is an array $Q[1..n]$ whose first $r - 1$ entries are positive and whose last $n - r + 1$ entries are all zeros, for some integer r .

The following recursive algorithm, essentially due to Gauss (who called it “methodical groping”), recursively enumerates all complete n -queens solutions that are consistent with a given partial solution. The input parameter r is the first empty row. Thus, to compute all n -queens solutions with no restrictions, we would call $\text{RECURSIVENQUEENS}(Q[1..n], 1)$.

```

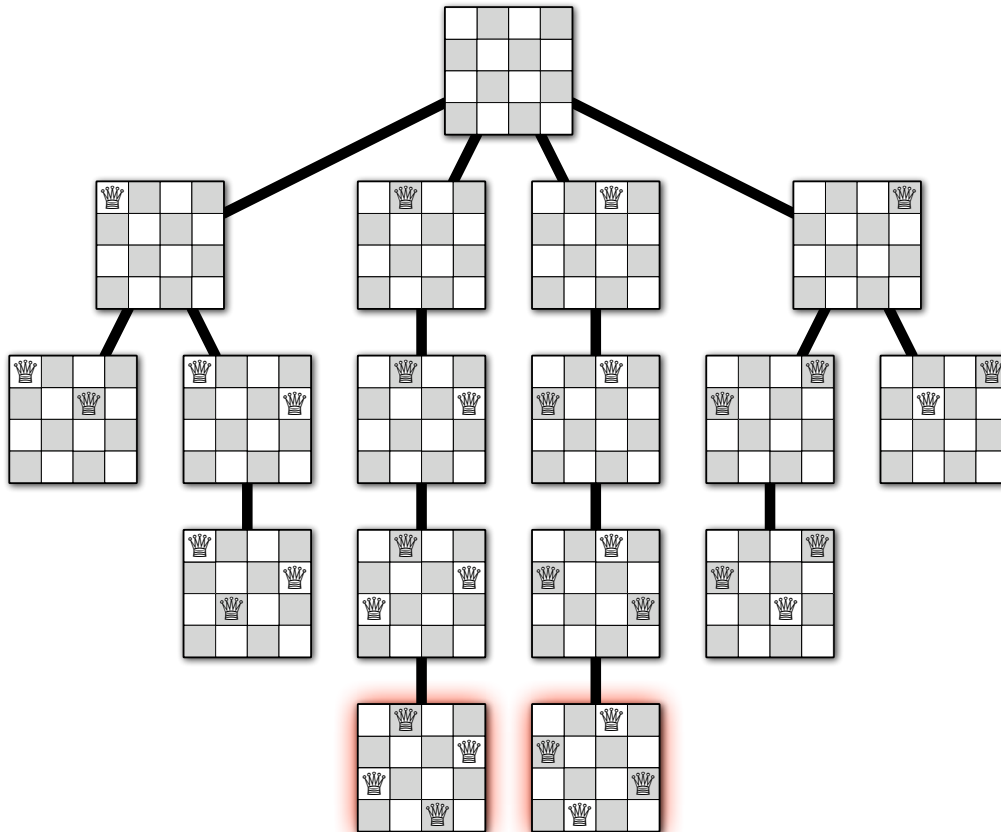
RECURSIVENQUEENS(Q[1..n], r):
  if r = n + 1
    print Q
  else
    for j ← 1 to n
      legal ← TRUE
      for i ← 1 to r - 1
        if (Q[i] = j) or (Q[i] = j + r - i) or (Q[i] = j - r + i)
          legal ← FALSE
      if legal
        Q[r] ← j
        RECURSIVENQUEENS(Q[1..n], r + 1)

```



One solution to the 8 queens problem, represented by the array [4,7,3,8,2,5,1,6]

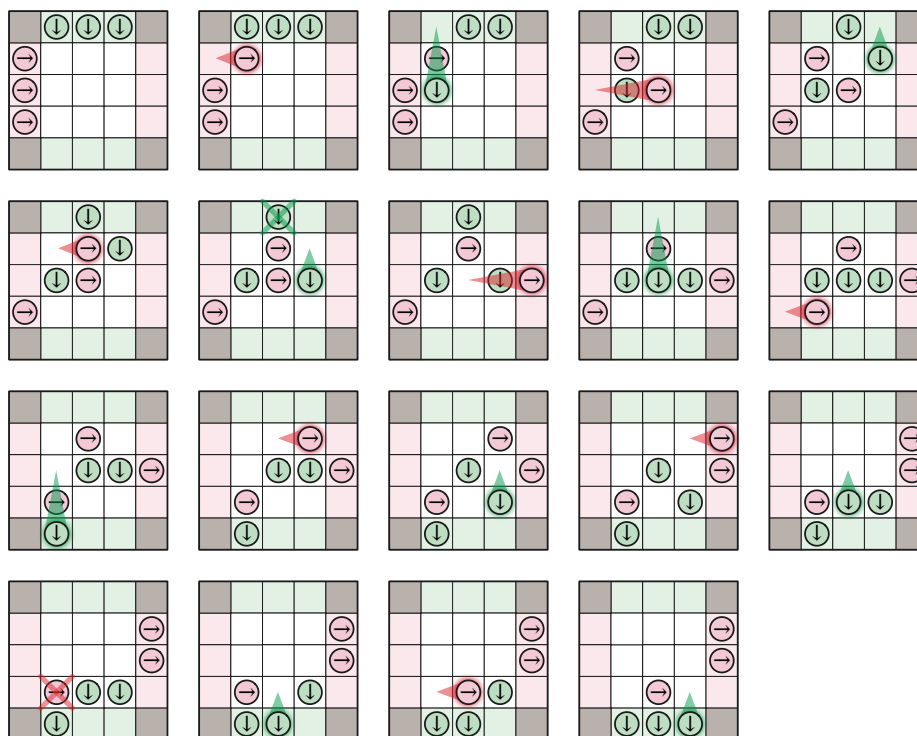
Like most recursive algorithms, the execution of a backtracking algorithm can be illustrated using a *recursion tree*. The root of the recursion tree corresponds to the original invocation of the algorithm; edges in the tree correspond to recursive calls. A path from the root down to any node shows the history of a partial solution to the n -Queens problem, as queens are added to successive rows. The leaves correspond to partial solutions that cannot be extended, either because there is already a queen on every row, or because every position in the next empty row is in the same row, column, or diagonal as an existing queen. The backtracking algorithm simply performs a depth-first traversal of this tree.



The complete recursion tree for our algorithm for the 4 queens problem.

7.2 Game Trees

Consider the following simple two-player game played on an $n \times n$ square grid with a border of squares; let's call the players Horatio Fahlberg-Remsen and Vera Rebaudi.¹ Each player has n tokens that they move across the board from one side to the other. Horatio's tokens start in the left border, one in each row, and move to the right; symmetrically, Vera's tokens start in the top border, one in each column, and move down. The players alternate turns. In each of his turns, Horatio either *moves* one of his tokens one step to the right into an empty square, or *jumps* one of his tokens over exactly one of Vera's tokens into an empty square two steps to the right. However, if no legal moves or jumps are available, Horatio simply passes. Similarly, Vera either moves or jumps one of her tokens downward in each of her turns, unless no moves or jumps are possible. The first player to move all their tokens off the edge of the board wins.



Vera wins the 3×3 game.

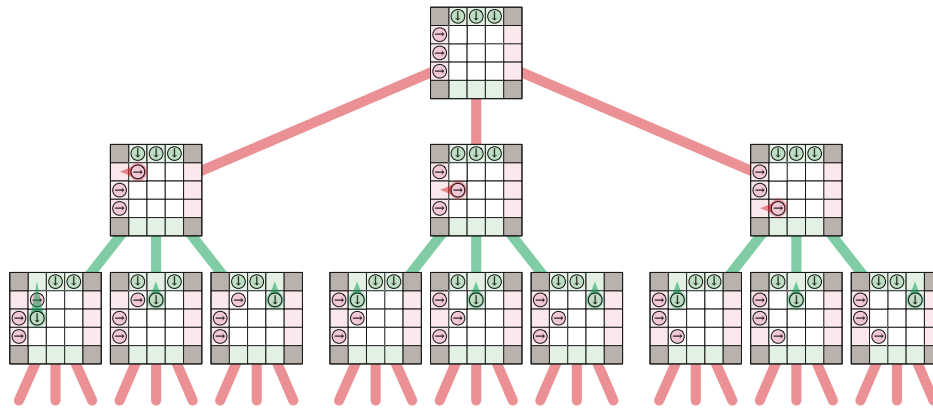
We can use a simple backtracking algorithm to determine the best move for each player at each turn. The *state* of the game consists of the locations of all the pieces and the player whose turn it is. We recursively define a game state to be *good* or *bad* as follows:

- A game state is *bad* if all the opposing player's tokens have reached their goals.
- A game state is *good* if the current player can move to a state that is bad for the opposing player.
- A configuration is *bad* if every move leads to a state that is good for the opposing player.

¹I don't know what this game is called, or even if I'm remembering the rules correctly. I learned it (or something like it) from Lenny Pitt, who recommended playing it with sweetener packets at restaurants.

Constantin Fahlberg and Ira Remsen synthesized saccharin for the first time in 1878, while Fahlberg was a postdoc in Remsen's lab investigating coal tar derivatives. In 1900, Ovidio Rebaudi published the first chemical analysis of *ka'a he'ê*, a medicinal plant cultivated by the Guaraní for more than 1500 years, now more commonly known as *Stevia rebaudiana*.

This recursive definition immediately suggests a recursive backtracking algorithm to determine whether a given state of the game is good or bad. Moreover, for any good state, the backtracking algorithm finds a move leading to a bad state for the opposing player. Thus, by induction, any player that finds the game in a good state on their turn can win the game, even if their opponent plays perfectly; on the other hand, starting from a bad state, a player can win only if their opponent makes a mistake.



The first two levels of the game tree.

All computer game players are ultimately based on this simple backtracking strategy. However, since most games have an enormous number of states, it is not possible to traverse the entire game tree in practice. Instead, game programs employ other heuristics² to *prune* the game tree, by ignoring states that are obviously good or bad (or at least obviously better or worse than other states), and/or by cutting off the tree at a certain depth (or *ply*) and using a more efficient heuristic to evaluate the leaves.

7.3 Subset Sum

Let's consider a more complicated problem, called SUBSETSUM: Given a set X of positive integers and *target* integer T , is there a subset of elements in X that add up to T ? Notice that there can be more than one such subset. For example, if $X = \{8, 6, 7, 5, 3, 10, 9\}$ and $T = 15$, the answer is TRUE, thanks to the subsets $\{8, 7\}$ or $\{7, 5, 3\}$ or $\{6, 9\}$ or $\{5, 10\}$. On the other hand, if $X = \{11, 6, 5, 1, 7, 13, 12\}$ and $T = 15$, the answer is FALSE.

There are two trivial cases. If the target value T is zero, then we can immediately return TRUE, because empty set is a subset of *every* set X , and the elements of the empty set add up to zero.³ On the other hand, if $T < 0$, or if $T \neq 0$ but the set X is empty, then we can immediately return FALSE.

For the general case, consider an arbitrary element $x \in X$. (We've already handled the case where X is empty.) There is a subset of X that sums to T if and only if one of the following statements is true:

- There is a subset of X that *includes* x and whose sum is T .
- There is a subset of X that *excludes* x and whose sum is T .

In the first case, there must be a subset of $X \setminus \{x\}$ that sums to $T - x$; in the second case, there must be a subset of $X \setminus \{x\}$ that sums to T . So we can solve SUBSETSUM(X, T) by reducing it to two simpler instances: SUBSETSUM($X \setminus \{x\}, T - x$) and SUBSETSUM($X \setminus \{x\}, T$). Here's how the resulting recursive algorithm might look if X is stored in an array.

²A heuristic is an algorithm that doesn't work.

³There's no base case like the vacuous base case!

```

SUBSETSUM( $X[1..n], T$ ):
  if  $T = 0$ 
    return TRUE
  else if  $T < 0$  or  $n = 0$ 
    return FALSE
  else
    return (SUBSETSUM( $X[1..n-1], T$ )  $\vee$  SUBSETSUM( $X[1..n-1], T - X[n]$ ))

```

Proving this algorithm correct is a straightforward exercise in induction. If $T = 0$, then the elements of the empty subset sum to T , so TRUE is the correct output. Otherwise, if T is negative or the set X is empty, then no subset of X sums to T , so FALSE is the correct output. Otherwise, if there is a subset that sums to T , then either it contains $X[n]$ or it doesn't, and the Recursion Fairy correctly checks for each of those possibilities. Done.

The running time $T(n)$ clearly satisfies the recurrence $T(n) \leq 2T(n-1) + O(1)$, which we can solve using either recursion trees or annihilators (or just guessing) to obtain the upper bound $T(n) = O(2^n)$. In the worst case, the recursion tree for this algorithm is a complete binary tree with depth n .

Here is a similar recursive algorithm that actually *constructs* a subset of X that sums to T , if one exists. This algorithm also runs in $O(2^n)$ time.

```

CONSTRUCTSUBSET( $X[1..n], T$ ):
  if  $T = 0$ 
    return  $\emptyset$ 
  if  $T < 0$  or  $n = 0$ 
    return NONE
   $Y \leftarrow$  CONSTRUCTSUBSET( $X[1..n-1], T$ )
  if  $Y \neq$  NONE
    return  $Y$ 
   $Y \leftarrow$  CONSTRUCTSUBSET( $X[1..n-1], T - X[n]$ )
  if  $Y \neq$  NONE
    return  $Y \cup \{X[n]\}$ 
  return NONE

```

7.4 The General Pattern

Find a small choice whose correct answer would reduce the problem size. For each possible answer, temporarily adopt that choice and recurse. (Don't try to be clever about which choices to try; just try them all.) The recursive subproblem is often more general than the original target problem; in each recursive subproblem, we must consider *only* solutions that are consistent with the choices we have already made.

7.5 NFA acceptance

Recall that a nondeterministic finite-state automaton, or NFA, can be described as a directed graph, whose edges are called *states* and whose edges have *labels* drawn from a finite set Σ called the *alphabet*. Every NFA has a designated *start* state and a subset of *accepting* states. Any walk in this graph has a label, which is a string formed by concatenating the labels of the edges in the walk. A string w is *accepted* by an NFA if and only if there is a walk from the start state to one of the accepting states whose label is w .

More formally (or at least, more symbolically), an NFA consists of a finite set Q of states, a start state $s \in Q$, a set of accepting states $A \subseteq Q$, and a transition function $\delta : Q \times \Sigma \rightarrow 2^Q$. We recursively extend

the transition function to strings by defining

$$\delta^*(q, w) = \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \bigcup_{r \in \delta(q, a)} \delta^*(r, x) & \text{if } w = ax. \end{cases}$$

The NFA accepts string w if and only if the set $\delta^*(s, w)$ contains at least one accepting state.

We can express this acceptance criterion more directly as follows. We define a boolean function $Accepts?(q, w)$, which is TRUE if the NFA would accept string w if we started in state q , and FALSE otherwise. This function has the following recursive definition:

$$Accepts?(q, w) := \begin{cases} \text{TRUE} & \text{if } w = \varepsilon \text{ and } q \in A \\ \text{FALSE} & \text{if } w = \varepsilon \text{ and } q \notin A \\ \bigvee_{r \in \delta(q, a)} Accepts?(r, x) & \text{if } w = ax \end{cases}$$

The NFA accepts w if and only if $Accepts?(s, w) = \text{TRUE}$.

In the magical world of non-determinism, we can imagine that the NFA always magically makes the right decision when faces with multiple transitions, or perhaps spawns off an independent parallel thread for each possible choice. Alas, real computers are neither clairvoyant nor (despite the increasing use of multiple cores) infinitely parallel. To simulate the NFA's behavior directly, we must recursively explore the consequences of each choice explicitly.

The recursive definition of $Accepts?$ translates directly into the following recursive backtracking algorithm. Here, the transition function δ and the accepting states A are represented as global boolean arrays, where $\delta[q, a, r] = \text{TRUE}$ if and only if $r \in \delta(q, a)$, and $A[q] = \text{TRUE}$ if and only if $q \in A$.

```

ACCEPTS?(q, w[1..n]):
  if n = 0
    return A[q]
  for all states r
    if δ[q, w[1], r] and ACCEPTS?(r, w[2..n])
      return TRUE
  return FALSE

```

To determine whether the NFA accepts a string w , we call $ACCEPTS?(\delta, A, s, w)$.

The running time of this algorithm satisfies the recursive inequality $T(n) \leq O(|Q|) \cdot T(n-1)$, which immediately implies that $T(n) = O(|Q|^n)$.

7.6 Longest Increasing Subsequence

Now suppose we are given a sequence of integers, and we want to find the longest subsequence whose elements are in increasing order. More concretely, the input is an array $A[1..n]$ of integers, and we want to find the longest sequence of indices $1 \leq i_1 < i_2 < \dots < i_k \leq n$ such that $A[i_j] < A[i_{j+1}]$ for all j .

To derive a recursive algorithm for this problem, we start with a recursive definition of the kinds of objects we're playing with: sequences and subsequences.

A sequence of integers is either empty
or an integer followed by a sequence of integers.

This definition suggests the following strategy for devising a recursive algorithm. If the input sequence is empty, there's nothing to do. Otherwise, we only need to figure out what to do with the first element of the input sequence; the Recursion Fairy will take care of everything else. We can formalize this strategy somewhat by giving a recursive definition of subsequence (using array notation to represent sequences):

The only *subsequence* of the empty sequence is the empty sequence.
 A *subsequence* of $A[1..n]$ is either a subsequence of $A[2..n]$
 or $A[1]$ followed by a subsequence of $A[2..n]$.

We're not just looking for just *any* subsequence, but a *longest* subsequence with the property that elements are in *increasing* order. So let's try to add those two conditions to our definition. (I'll omit the familiar vacuous base case.)

The *LIS* of $A[1..n]$ is
 either the LIS of $A[2..n]$
 or $A[1]$ followed by the LIS of $A[2..n]$ with elements larger than $A[1]$,
 whichever is longer.

This definition is correct, but it's not quite recursive—we're defining the object 'longest increasing subsequence' in terms of the slightly *different* object 'longest increasing subsequence with elements larger than x ', which we haven't properly defined yet. Fortunately, this second object has a very similar recursive definition. (Again, I'm omitting the vacuous base case.)

If $A[1] \leq x$, the LIS of $A[1..n]$ with elements larger than x is
 the LIS of $A[2..n]$ with elements larger than x .
 Otherwise, the LIS of $A[1..n]$ with elements larger than x is
 either the LIS of $A[2..n]$ with elements larger than x
 or $A[1]$ followed by the LIS of $A[2..n]$ with elements larger than $A[1]$,
 whichever is longer.

The longest increasing subsequence without restrictions can now be redefined as the longest increasing subsequence with elements larger than $-\infty$. Rewriting this recursive definition into pseudocode gives us the following recursive algorithm.

$LIS(A[1..n])$:
 return LISBIGGER($-\infty, A[1..n]$)

$LISBIGGER(prev, A[1..n])$:
 if $n = 0$
 return 0
 else
 $max \leftarrow LISBIGGER(prev, A[2..n])$
 if $A[1] > prev$
 $L \leftarrow 1 + LISBIGGER(A[1], A[2..n])$
 if $L > max$
 $max \leftarrow L$
 return max

The running time of this algorithm satisfies the recurrence $T(n) \leq 2T(n-1) + O(1)$, which as usual implies that $T(n) = O(2^n)$. We really shouldn't be surprised by this running time; in the worst case, the algorithm examines each of the 2^n subsequences of the input array.

The following alternative strategy avoids defining a new object with the “larger than x ” constraint. We still only have to decide whether to include or exclude the first element $A[1]$. We consider the case where $A[1]$ is excluded exactly the same way, but to consider the case where $A[1]$ is included, we remove any elements of $A[2..n]$ that are larger than $A[1]$ *before* we recurse. This new strategy gives us the following algorithm:

```

FILTER( $A[1..n], x$ ):
   $j \leftarrow 1$ 
  for  $i \leftarrow 1$  to  $n$ 
    if  $A[i] > x$ 
       $B[j] \leftarrow A[i]; j \leftarrow j + 1$ 
  return  $B[1..j]$ 

```

```

LIS( $A[1..n]$ ):
  if  $n = 0$ 
    return 0
  else
     $max \leftarrow \text{LIS}(prev, A[2..n])$ 
     $L \leftarrow 1 + \text{LIS}(A[1], \text{FILTER}(A[2..n], A[1]))$ 
    if  $L > max$ 
       $max \leftarrow L$ 
  return  $max$ 

```

The FILTER subroutine clearly runs in $O(n)$ time, so the running time of LIS satisfies the recurrence $T(n) \leq 2T(n-1) + O(n)$, which solves to $T(n) \leq O(2^n)$ by the annihilator method. This upper bound pessimistically assumes that FILTER never actually removes any elements; indeed, if the input sequence is sorted in increasing order, this assumption is correct.

7.7 Optimal Binary Search Trees

Retire this example? It's not a *bad* example, exactly—certainly it's infinitely better than the execrable matrix-chain multiplication problem from Aho, Hopcroft, and Ullman—but it's not the best *first* example of tree-like backtracking. Minimum-ink triangulation of convex polygons is both more intuitive (geometry FTW!) and structurally equivalent. CFG parsing and regular expression matching (really just a special case of parsing) have similar recursive structure, but are a bit more complicated.

Our next example combines recursive backtracking with the divide-and-conquer strategy. Recall that the running time for a successful search in a binary search tree is proportional to the number of ancestors of the target node.⁴ As a result, the worst-case search time is proportional to the depth of the tree. Thus, to minimize the worst-case search time, the height of the tree should be as small as possible; by this metric, the ideal tree is perfectly balanced.

In many applications of binary search trees, however, it is more important to minimize the total cost of several searches rather than the worst-case cost of a single search. If x is a more ‘popular’ search target than y , we can save time by building a tree where the depth of x is smaller than the depth of y , even if that means increasing the overall depth of the tree. A perfectly balanced tree is *not* the best choice if some items are significantly more popular than others. In fact, a totally unbalanced tree of depth $\Omega(n)$ might actually be the best choice!

This situation suggests the following problem. Suppose we are given a sorted array of **keys** $A[1..n]$ and an array of corresponding **access frequencies** $f[1..n]$. Our task is to build the binary search tree that minimizes the *total* search time, assuming that there will be exactly $f[i]$ searches for each key $A[i]$.

Before we think about how to solve this problem, we should first come up with a good recursive definition of the function we are trying to optimize! Suppose we are also given a binary search tree T with n nodes. Let v_i denote the node that stores $A[i]$, and let r be the index of the root node. Ignoring

⁴An *ancestor* of a node v is either the node itself or an ancestor of the parent of v . A *proper* ancestor of v is either the parent of v or a proper ancestor of the parent of v .

constant factors, the cost of searching for $A[i]$ is the number of nodes on the path from the root v_r to v_i . Thus, the total cost of performing all the binary searches is given by the following expression:

$$\text{Cost}(T, f[1..n]) = \sum_{i=1}^n f[i] \cdot \#\text{nodes between } v_r \text{ and } v_i$$

Every search path includes the root node v_r . If $i < r$, then all other nodes on the search path to v_i are in the left subtree; similarly, if $i > r$, all other nodes on the search path to v_i are in the right subtree. Thus, we can partition the cost function into three parts as follows:

$$\begin{aligned} \text{Cost}(T, f[1..n]) &= \sum_{i=1}^{r-1} f[i] \cdot \#\text{nodes between } \text{left}(v_r) \text{ and } v_i \\ &\quad + \sum_{i=1}^n f[i] \\ &\quad + \sum_{i=r+1}^n f[i] \cdot \#\text{nodes between } \text{right}(v_r) \text{ and } v_i \end{aligned}$$

Now the first and third summations look exactly like our original expression (*) for $\text{Cost}(T, f[1..n])$. Simple substitution gives us our recursive definition for Cost :

$$\text{Cost}(T, f[1..n]) = \text{Cost}(\text{left}(T), f[1..r-1]) + \sum_{i=1}^n f[i] + \text{Cost}(\text{right}(T), f[r+1..n])$$

The base case for this recurrence is, as usual, $n = 0$; the cost of performing no searches in the empty tree is zero.

Now our task is to compute the tree T_{opt} that minimizes this cost function. Suppose we somehow magically knew that the root of T_{opt} is v_r . Then the recursive definition of $\text{Cost}(T, f)$ immediately implies that the left subtree $\text{left}(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$. Similarly, the right subtree $\text{right}(T_{\text{opt}})$ must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$. **Once we choose the correct key to store at the root, the Recursion Fairy automatically constructs the rest of the optimal tree.** More formally, let $\text{OptCost}(f[1..n])$ denote the total cost of the optimal search tree for the given frequency counts. We immediately have the following recursive definition.

$$\text{OptCost}(f[1..n]) = \min_{1 \leq r \leq n} \left\{ \text{OptCost}(f[1..r-1]) + \sum_{i=1}^n f[i] + \text{OptCost}(f[r+1..n]) \right\}$$

Again, the base case is $\text{OptCost}(f[1..0]) = 0$; the best way to organize no keys, which we will plan to search zero times, is by storing them in the empty tree!

This recursive definition can be translated mechanically into a recursive algorithm, whose running time $T(n)$ satisfies the recurrence

$$T(n) = \Theta(n) + \sum_{k=1}^n (T(k-1) + T(n-k)).$$

The $\Theta(n)$ term comes from computing the total number of searches $\sum_{i=1}^n f[i]$.

Yeah, that's one ugly recurrence, but it's actually easier to solve than it looks. To transform it into a more familiar form, we regroup and collect identical terms, subtract the recurrence for $T(n-1)$ to get rid of the summation, and then regroup again.

$$T(n) = \Theta(n) + 2 \sum_{k=0}^{n-1} T(k)$$

$$T(n-1) = \Theta(n-1) + 2 \sum_{k=0}^{n-2} T(k)$$

$$T(n) - T(n-1) = \Theta(1) + 2T(n-1)$$

$$T(n) = 3T(n-1) + \Theta(1)$$

The solution $T(n) = \Theta(3^n)$ now follows from the annihilator method.

Let me emphasize that this recursive algorithm does *not* examine all possible binary search trees. The number of binary search trees with n nodes satisfies the recurrence

$$N(n) = \sum_{r=1}^{n-1} (N(r-1) \cdot N(n-r)),$$

which has the closed-form solution $N(n) = \Theta(4^n / \sqrt{n})$. Our algorithm saves considerable time by searching *independently* for the optimal left and right subtrees. A full enumeration of binary search trees would consider all possible *pairings* of left and right subtrees; hence the product in the recurrence for $N(n)$.

7.8 CFG Parsing

Our final example is the *parsing* problem for context-free languages. Given a string w and a context-free grammar G , does w belong to the language generated by G ? Recall that a context-free grammar over the alphabet Σ consists of a finite set Γ of *non-terminals* (disjoint from Σ) and a finite set of *production rules* of the form $A \rightarrow w$, where A is a nonterminal and w is a string over $\Sigma \cup \Gamma$.

Real-world applications of parsing normally require more information than just a single bit. For example, compilers require parsers that output a parse tree of the input code; some natural language applications require the *number* of distinct parse trees for a given string; others assign probabilities to the production rules and then ask for the *most likely* parse tree for a given string. However, these more general problems can be solved using relatively straightforward generalizations of the following decision algorithm.

★★★★ Backtracking recurrence behind CYK

Exercises

- Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is both a subsequence of A and a subsequence of B . Give a simple recursive definition for the function $lcs(A, B)$, which gives the length of the *longest* common subsequence of A and B .
 - Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common supersequence* of A and B is another sequence that contains both A and B as subsequences. Give a simple recursive definition for the function $scs(A, B)$, which gives the length of the *shortest* common supersequence of A and B .

- (c) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i + 1]$ for all even i , and $X[i] > X[i + 1]$ for all odd i . Give a simple recursive definition for the function $los(A)$, which gives the length of the longest oscillating subsequence of an arbitrary array A of integers.
- (d) Give a simple recursive definition for the function $sos(A)$, which gives the length of the shortest oscillating supersequence of an arbitrary array A of integers.
- (e) Call a sequence $X[1..n]$ *accelerating* if $2 \cdot X[i] < X[i - 1] + X[i + 1]$ for all i . Give a simple recursive definition for the function $lxs(A)$, which gives the length of the longest accelerating subsequence of an arbitrary array A of integers.

For more backtracking exercises, see the next two lecture notes!