Introduction to Data Structure

Computer is an electronic machine which is used for data processing and manipulation. When programmer collects such type of data for processing, he would require to store all of them in computer's main memory.
In order to make computer work we need to know:

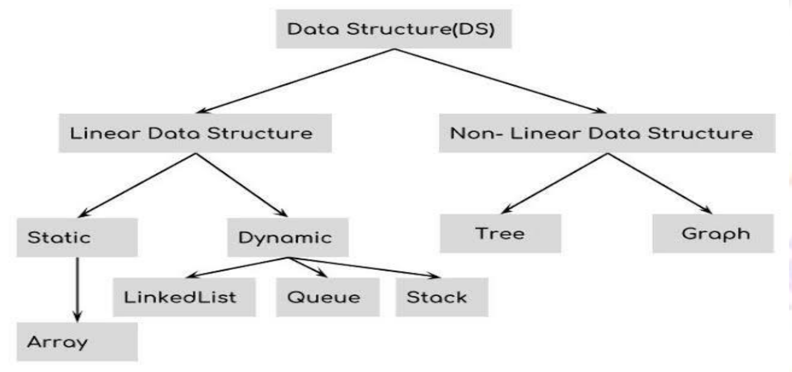
- o Representation of data in computer.
- o Accessing of data.
- o How to solve problem step by step.

For doing this task we use data structure.

What is Data Structure?

Data structure is a representation of the logical and mathematical relationship existing among the data.
Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
We can also define data structure as a mathematical or logical model of a particular organization of data items.

Classification of Data Structure



Array: An array is a fixed-size sequenced collection of elements of the same data type.
Linked List: it is a collection of nodes and every node is divided into two parts. First part is known as INFO field and contains the information about the node, Second part, which is known as LINK or NEXTPOINTER Field contains the address of next node.
Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only. The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation. Stack is also called as Last in First out (LIFO) data structure.
Queue: The data structure which permits the insertion at one end and Deletion at another end, known a Queue. One End at which deletion is occurs is known

as FRONT end and another end at which insertion occurs is known as REAR end. Queue is also called as First in First out (FIFO) data structure.
Tree: A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement. Trees represent the hierarchical relationship between various elements. Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.
Graph: Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes. A tree can be viewed as restricted graph.

Operation on Data Structures
Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types
Traversal: Traversal is a process of visiting each and every node of a list in systematic manner. That means to go through each and every item of the list at least once.
Insert: Insertion is a process of adding one or more items in the given list.
Delete: Deletion is a process of removing one or more items from given list.
Update: Update operation is a process of editing or modifying the data in the Given data structure.
Searching: It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.
Sorting: Sorting is a process of rearranging all data items in a data structure in a desired order. Desired order may be ascending order or descending order.
Merging: Merging is a process of combining the data items of two different sorted list into a single sorted list.

Array: it is a collection of homogeneous data items. In that all elements share common name.
There are three categories of Arrays:
- One Dimensional Array
- Two Dimensional Array
- Multidimensional Array

Traversal

Algorithm TRAVERSEARRAY(A,N)
```
//Here TRAVERSEARRAY is an algorithm which is used to traverse
//all the elements of Array A with N elements.
Step-1 Start
Step-2 set K:=LB.
Step-3 Repeat steps 4 and 5 while K<=UB
Step-4 PROCESS A[K].
```

```
    Step-5 set K:=K+1.
    Step-6 Exit.
```

Program

```
void main()
{
        void show(int a[],int n);
        int arr[5]={1,2,3,4,5};
        int n=5;
        clrscr();
        show(arr,n);
        getch();
}
void show(int arr[],int n)
{
        int k;
        for(k=0;k<n;k++)
                printf("%d, ",arr[k]);
}
```

Insertion into Array:

Algorithm INSERTARRAY(A,N,ITEM,LOC)

```
    //Here INSERTARRAY is an algorithm which is used to insert an ITEM
    //on the Location LOC in an Array A with N elements.
    Step-1 Start
    Step-2 set J:=N.
    Step-3 Repeat steps 4 and 5 while K>=LOC
    Step-4 set A[J]:=A[J-1].
    Step-5 set J:=J-1.
    Step-6 set A[LOC]:=ITEM.
    Step-7 set N:=N+1.
    Step-8 Exit.
```

Program:

```
void main()
{
        void insert(int a[],int itm,int l,int &n);
        int arr[5]={1,2,3,4};
        int n=4,k,loc,item;
        clrscr();
        printf("\nEnter the item and location ");
        scanf("%d%d",&item,&loc);
        printf("\nBefore Insertion\n");
        for(k=0;k<n;k++)
            printf("%d, ",arr[k]);
        insert(arr,item,loc,n);
        printf("\nAfter Insertion\n");
        for(k=0;k<n;k++)
            printf("%d, ",arr[k]);
        getch();
}
void insert(int arr[],int item,int loc,int &n)
{
        int k;
        k=n-1;
        while(k>=loc-1)
        {
            arr[k+1]=arr[k];
            k--;
        }
        arr[loc-1]=item;
        n=n+1;
}
```

Deletion from Array:

Algorithm DELETEARRAY(A,N,ITEM,LOC)

```
    //Here DELETEARRAY is an algorithm which is used to delete an
    //ITEM  from the Location LOC in an Array A with N elements.
    Step-1 Start
    Step-2 set J:=LOC.
    Step-3 set ITEM:=A[LOC].
    Step-4 Repeat steps 4 and 5 while K<N
    Step-5 set A[J]:=A[J+1].
    Step-6 set J:=J+1.
    Step-7 set N:=N-1.
    Step-8 Exit.
```

Program:

```
void main()
{
    void delet(int a[],int l,int &n);
    int arr[5]={1,2,3,4,5};
    int n=5,k,loc;
    clrscr();
    printf("\nEnter the location ");
    scanf("%d",&loc);
    printf("\nBefore Deletion\n");
    for(k=0;k<n;k++)
        printf("%d, ",arr[k]);
    delet(arr,loc,n);
    printf("\nAfter Deletion\n");
    for(k=0;k<n;k++)
        printf("%d, ",arr[k]);
    getch();
}
void delet(int arr[],int loc,int &n)
{
    int k;
    k=loc-1;
    while(k<=n-1)
    {
        arr[k]=arr[k+1];
        k++;
    }
    n=n-1;
}
```

Sorting and Searching

Bubble sort: Bubble sort, sometimes referred as sinking sort. It is the simplest way of sorting the elements. It compares each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Algorithm BUBBLESORT(A,N)

```
    //Here BUBBLESORT is an algorithm which is used to Sort all
    //elements of an Array A with N elements using Bubble Sort
    //Technique.
    Step-1 Start
    Step-2 set K:=1.
    Step-3 Repeat steps 4 to 8 while K<N
    Step-4 set J:=1.
    Step-5 Repeat steps 6 and 7 while J<=N-K
    Step-6 if A[J]>A[J+1] then
                a. Set TEMP:=A[J].
                b. Set A[J]:=A[J+1].
                c. Set A[J+1]:=TEMP.
    Step-7 set J:=J+1.
    Step-8 set K:=K+1.
    Step-9 Exit.
```

Program:

```
void main()
{
        void bubble_sort(int a[],int n);
        int arr[10]={99,487,982,273,167,65,61,157,553,46};
        int temp,k,n=10;
        clrscr();
        printf("\nBefore sorting\n");
        for(k=0;k<n;k++)
        {
                        printf("%d, ",arr[k]);
        }
        bubble_sort(arr,n);
        printf("\nAfter sorting\n");
        for(k=0;k<n;k++)
        {
                        printf("%d, ",arr[k]);
        }
        getch();
}
void bubble_sort(int a[],int n)
{
        int temp,k,j;
                        k=1;
                        while(k<n)
                        {
                              j=0;
                              while(j<n-k)
                              {
                                      if(a[j]>a[j+1])
                                      {
                                              temp=a[j];
                                              a[j]=a[j+1];
                                              a[j+1]=temp;
                                      }
                                      j++;
                              }
                              k++;
                        }
}
```

Selection Sort: The idea of algorithm is quite simple. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array. At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes empty, algorithm stops.

Algorithm SELECTIONSORT(A,N)

```
    //Here SELECTIONSORT is an algorithm which is used to Sort all
    //elements of an Array A with N elements using Selection Sort
    //Technique.
    Step-1 Start
    Step-2 Set K:=1.
    Step-3 Repeat steps 4 to 13 while K<N
    Step-4 Set MIN:=A[K].
    Step-5 Set LOC:=K.
    Step-6 Set J:=K+1.
    Step-7 Repeat steps 8 and 9 while J<=N
    Step-8 if MIN>A[J] then
                a. Set MIN:=A[J].
                b. Set LOC:=J.
    Step-9 Set J:=J+1.
    Step-10 Set TEMP:=A[K].
    Step-11 Set A[K]:=A[LOC].
    Step-12 Set A[LOC]:=TEMP.
    Step-13 Set K:=K+1.
    Step-14 Exit.
```

Program:

```
void main()
{
	void sel_sort(int arr[],int n);
	int arr[]={11,55,22,133,44,66,77,8,21,34};
	int k,n=10;
	clrscr();
	printf("\nBefore sorting\n");
	for(k=0;k<n;k++)
	{
		printf("%d, ",arr[k]);
	}
	sel_sort(arr,n);
	printf("\nAfter sorting\n");
	for(k=0;k<n;k++)
	{
		printf("%d, ",arr[k]);
	}
	getch();
}
void sel_sort(int arr[],int n)
{
        int min,j,k,temp,loc;
        k=0;
        while(k<n)
        {
                min=arr[k];
                loc=k;
                j=k+1;
                while(j<n)
                {
                        if(arr[j]<min)
                        {
                                min=arr[j];
                                loc=j;
                        }
                        j++;
                }
                temp=arr[k];
                arr[k]=arr[loc];
                arr[loc]=temp;
                k++;
        }
}
```

Insertion Sort: The idea of algorithm is quite simple. Insertion sort is the simple sorting algorithm which sorts the array by shifting elements one by one. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array. At every step, algorithm selects the first element of unsorted array and set it on its correct position in the sorted part. When unsorted part becomes empty, algorithm stops.

Algorithm INSERTIONSORT(A,N)

```
        //Here INSERTIONSORT is an algorithm which is used to Sort all
        //elements of an Array A with N elements using Insertion Sort
        //Technique.
        Step-1 Start
        Step-2 Set A[0]=-∞.
        Step-3 Set K:=2.
        Step-4 Repeat steps 5 to 11 while K<N
        Step-5 Set TEMP:=A[K].
```

```
Step-6 Set PTR:=K-1.
Step-7 Repeat steps 8 & 9 while TEMP<A[PTR]
Step-8 Set A[PTR+1]:=A[PTR].
Step-9 Set PTR:=PTR-1.
Step-10 Set A[PTR+1]+1:=TEMP.
Step-11 Set K:=K+1.
Step-12 Exit.
```

Program

```
void main()
{
	void ins_sort(int arr[],int n);
	int arr[]={11,55,22,133,44,66,77,8,21,34};
	int k,n=10;
	clrscr();
	printf("\nBefore sorting\n");
	for(k=0;k<n;k++)
	{
		printf("%d, ",arr[k]);
	}
	ins_sort(arr,n);
	printf("\nAfter sorting\n");
	for(k=0;k<n;k++)
	{
		printf("%d, ",arr[k]);
	}
	getch();
}
void ins_sort(int arr[],int n)
{
	int k,j,temp;
	k=0;
	while(k<n)
	{
		temp=arr[k];
		j=k-1;
		while(j>=0)
		{
			if(arr[j]<=temp)
			{
				break;
			}
			arr[j+1]=arr[j];
```

```
                j--;
            }
            arr[j+1]=temp;
            k++;
        }
}
```

Linear/Sequential Search In computer science, linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.
Linear search is the simplest search algorithm.
This technique can be used in sorted and unsorted list.

Algorithm LINEARSEARCH(A,N,ITEM,LOC)

```
        //Here LINEARSEARCH is an algorithm which is used to find the
        //location LOC of the ITEM in the Array A with N elements using
        //Linear Search Technique.
        Step-1 Start
        Step-2 Set LOC=-1.
        Step-3 Set K:=0.
        Step-4 Repeat steps 5 & 6 while K<N
        Step-5 if ITEM=A[K].
                        Then set LOC:=K and go to step 7.
        Step-6 Set K:=K+1.
        Step-7 if LOC = -1
                        Then print "Search is unsuccessful".
                   Otherwise
                        Print "Search is Successful and Location is ", (LOC+1).
        Step-8 Exit.
```

Program

```
void main()
{
        int lin_search(int a[],int item,int n);
        int arr[10]={11,77,22,33,44,55,21,34,53,66};
        int loc,item,n=10;
        clrscr();
        printf("\nEnter the item to search ");
        scanf("%d",&item);
        loc=lin_search(arr,item,n);
        if(loc!=0)
                printf("\nSearch is Successful and Location is %d",loc);
        else
                printf("\nSearch is UnSuccessful");
        getch();
}
int lin_search(int a[],int item,int n)
{
        int k,lc=0;
        k=0;
        while(k<n)
        {
                if(item==a[k])
                {
                        lc=k+1;
                        break;
                }
                k++;
        }
        return lc;
}
```

Binary Search If we have an array that is sorted, we can use a much more efficient algorithm called a Binary Search. In binary search each time we divide array into two equal half and compare middle element with search element. If middle element is equal to search element then we got that element and return that index otherwise if middle element is less than search element we look right part of array and if middle element is greater than search element we look left part of array.

Algorithm BINARYSEARCH(A,N,ITEM,LOC,LB,UB)

```
    //Here BINARYSEARCH is an algorithm which is used to find the
    //location LOC of the ITEM in the Array A with N elements using
    //Binary Search Technique.LB is the Lower bound of A and UB is the
    //Upper bound of A.
    Step-1 Start
    Step-2 Set LOC=-1.
    Step-3 Set BEG:=LB & END:=UB.
    Step-4 Repeat steps 5 & 6 while BEG<=END
    Step-5 Set MID:=(INT) (BEG+END)/2.
    Step-6 if ITEM = A[MID]
```

```
                Then set LOC:=MID and go to step 7.
            Else if ITEM<A[MID]
                Then set END:=MID-1.
            Else
                Set BEG:=MID+1.
    Step-7 if LOC = -1
                Then print "Search is unsuccessful".
            Otherwise
                Print "Search is Successful and Location is ", (LOC+1).
    Step-8 Exit.
```

Program

```
void main()
{
	int bin_search(int a[],int item,int n);
	int arr[10]={99,87,82,73,67,65,61,57,53,46};
	int loc,item,n=10;
	clrscr();
	printf("\nEnter the item to search ");
	scanf("%d",&item);
	loc=bin_search(arr,item,n);
	if(loc!=0)
		printf("\nSearch is Successful and Location is %d",loc);
	else
		printf("\nSearch is UnSuccessful");
	getch();
}
int bin_search(int a[],int item,int n)
{
	int mid,beg,end,lc=0;
	beg=0;
	end=n-1;
	while(beg<=end)
	{
		mid=(beg+end)/2;
		if(item==a[mid])
		{
			lc=mid+1;
			break;
		}
		else if(a[mid]<item)
		{
			end=mid-1;
```

```
            }
            else
            {
                    beg=mid+1;
            }
    }
    return lc;
}
```

Stack: A linear list which allows insertion and deletion of an element at one end only is called stack. The insertion operation is called as **PUSH** and deletion operation as **POP**. Since insertion and deletion operations are performed at one end of a stack, the elements can only be removed in the opposite orders from that in which they were added to the stack; such a linear list is referred to as a LIFO (last in first out) list.

Applications of Stack

- Recursion
- Keeping track of function calls
- Evaluation of expressions
- Reversing characters
- Servicing hardware interrupts
- Solving combinatorial problems using backtracking.

Procedure PUSH (ST, TOP, MAX, ITEM)

```
//This procedure inserts an element ITEM onto the TOP of a stack
//ST. MAX is the maximum capacity of ST.
    Step 1 Start.
    Step-2 if TOP>=MAX then
            Print "Overflow" and goto step 5.
    Step-3 Set TOP:=TOP+1.
    Step-4 Set ST[TOP]:=ITEM.
    Step-5 Exit.
```

Procedure POP (ST, TOP, MAX, ITEM)

```
//This procedure deletes an element ITEM from the TOP of a stack
//ST. MAX is the Maximum capacity of ST.
    Step 1 Start.
    Step-2 if TOP<=-1 then
            Print "Underflow" and goto step 5.
    Step-3 Set ITEM := ST[TOP].
    Step-4 Set TOP:=TOP-1.
```

Step-5 Exit.

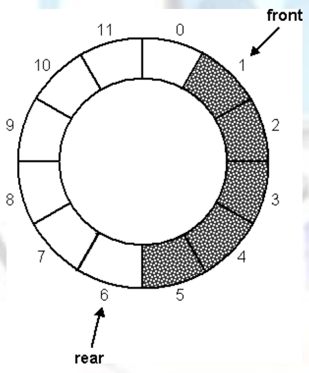
Program

```
#define MAX 5
int stack[MAX],top;
void push(int item);
void pop();
void show();
void main()
{
        int ch,item;
        char cho;
        clrscr();
        top=0;
        do
        {
        printf("\nMenu\n1.Push\n2.Pop\n3.Show\nEnter your choice ");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:
                        printf("Enter the item to insert ");
                        scanf("%d",&item);
                        push(item);
                        break;
                case 2:
                        pop();
                        break;
                case 3:
                        show();
                        break;
                default:
                        printf("\nWrong choice");
        }
        printf("\nDo you want more ");
        scanf(" %c",&cho);
        }while(cho=='y'||cho=='Y');
        getch();
}
void push(int item)
{
        if(top>=MAX)
```

```
	{
		printf("Overflow");
	}
	else
	{
		stack[top]=item;
		top++;
		printf("\nItem is pushed");
	}
}
void pop()
{
	int item;
	if(top<=0)
	{
		printf("Underflow");
	}
	else
	{
		top--;
		item=stack[top];
		printf("\nItem %d is popped",item);
	}
}
void show()
{
	int i;
	if(top!=0)
	{
		for(i=0;i<top;i++)
		{
			printf("\n%d",stack[i]);
		}
	}
	else
	{
		printf("\nStack is empty");
	}
}
```

Queue: A linear list which permits deletion to be performed at one end of the list and insertion at the other end is called queue. The information in such a list is processed FIFO (first in first out) of FCFS (first come first served) pattern. Front is the end of queue from that deletion is to be performed. Rear is the end of queue at which new element is to be inserted. The process to add an element into queue is called **Enqueue.** The process of removal of an element from queue is called **Dequeue**. The familiar and traditional example of a queue is Checkout line at Supermarket Cash Register where the first person in line is usually the first to be checked out.

Circular Queue: A more suitable method of representing simple queue which prevents an excessive use of memory is to arrange the elements Q[1], Q[2]....,Q[n] in a circular fashion with Q[1] following Q[n], this is called circular queue. In a standard queue data structure re-buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue. Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle. It is also called as "Ring buffer".



Dequeue: A dequeue (double ended queue) is a linear list in which insertion and deletion are performed from the either end of the structure. There are two variations of Dqueue:

- **Input restricted dqueue**- allows insertion at only one end and deletion at either end.
- **Output restricted dqueue**- allows deletion from only one end and insertion at either end.

Priority Queue: A queue in which elements are assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

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

A prototype of a priority queue is a timesharing system: Programs of high priority are processed first, and programs with same priority form standard queue.

Procedure INSERT (Q, FRONT, REAR, MAX, ITEM)

```
//This procedure inserts an element ITEM onto the REAR position of
//Queue Q. FRONT is the starting position of Q and REAR is the last
//position of Q. MAX is the maximum capacity of Q.
    Step 1 Start.
    Step-2 if (FRONT = 0 & REAR = MAX-1) OR (FRONT=REAR+1) then
                Print "Overflow" and goto step 5.
    Step-3 if (FRONT=NULL) then
                a. Set FRONT:=0.
                b. Set REAR:=0.
           Else if (REAR=MAX-1)
                   Set REAR:=0.
           Else
                   Set REAR:=REAR+1.
    Step-4 Set Q[REAR]:=ITEM.
    Step-5 Exit.
```

Procedure DELETE (Q, FRONT, REAR, MAX, ITEM)

```
//This procedure deletes an element ITEM from the FRONT position
//of Queue Q. FRONT is the starting position of Q and REAR is the
//last position of Q. MAX is the maximum capacity of Q.
    Step 1 Start.
    Step-2 if (FRONT = NULL) then
                Print "Underflow" and goto step 5.
    Step-3 Set ITEM:= Q[FRONT].
    Step-4 if (FRONT=REAR) then
                a. Set FRONT:=NULL.
                b. Set REAR:=NULL.
           Else if (FRONT=MAX-1)
                   Set FRONT:=0.
           Else
                   Set FRONT:=FRONT+1.
    Step-5 Exit.
```

Program

```
#define MAX 5
int q[MAX],front,rear;
void insert(int item);
void delet();
void show();
void main()
{
        int ch,item;
        char cho;
        clrscr();
        front=-1;
        rear=-1;
        do
        {
        printf("\nMenu\n1.Insert\n2.Delete\n3.Show\nEnter your choice ");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:
                        printf("Enter the item to insert ");
                        scanf("%d",&item);
                        insert(item);
                        break;
                case 2:
                        delet();
                        break;
                case 3:
                        show();
                        break;
                default:
                        printf("\nWrong choice");
        }
        printf("\nDo you want more ");
        scanf(" %c",&cho);
        }while(cho=='y'||cho=='Y');
        getch();
}
void insert(int item)
{
        if((front==0 && rear == MAX-1)||(front==rear+1))
        {
                printf("Overflow");
        }
        else
        {
                if(front == rear && front == -1)
                {
                        front = 0;
                        rear = 0;
                }
                else if(rear==MAX-1)
                {
                        rear=0;
                }
                else
                {
                        rear = rear+1;
                }
                q[rear]=item;
                printf("\nItem is inserted");
        }
}
void delet()
{
        int item;
        if(front==rear && front==-1)
        {
                printf("Underflow");
        }
        else
        {
                item=q[front];
                if(front==rear)
                {
                        front=-1;
                        rear=-1;
                }
                else if(front==MAX-1)
                {
                        front=0;
                }
                else
                {
                        front = front+1;
                }
                printf("\nItem %d is deleted",item);
        }
}
```

```
void show()
{
	int i;
	if(front==rear && front==-1)
	{
		printf("\nQueue is empty");
	}
	else
	{
		if(front<=rear)
		{
			for(i=front;i<=rear;i++)
			{
				printf("\n%d",q[i]);
			}
		}
		else
		{
			for(i=front;i<MAX;i++)
			{
				printf("\n%d",q[i]);
			}
			for(i=0;i<=rear;i++)
			{
				printf("\n%d",q[i]);
			}
		}
	}
}
```

Tree: Tree is a non-linear data structure. It is mainly used to represent data containing a hierarchical relationship between elements, ie. Records, family trees and table of contents. Here we will discuss about a special kind of tree, called Binary tree.

Binary Tee: a binary tree is defined as a finite set of elements, called nodes, such that:

- T is empty (called null or empty tree).
- T contains a distinguished node R, called the Root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T1 and T2.

If T does contain a root R, then the two trees T1 and T2 are called, respectively, the left and right subtrees of R. if T1 is nonempty, then its root is called the

left successor of R. similarly, if T2 is nonempty, then its root is called the right successor of R.
Any node N in the binary tree T has either 0, 1 or 2 successors.
Figure:
Complete Binary Tree: Consider any binary tree T. each node of T can have at most two children. Accordingly, one can show that level r of T can have at most 2^r nodes. The tree T is said to be complete if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible. Thus there is a unique complete tree T_n with exactly n nodes.
Figure:
Extended Binary Tree (2-Tree): A binary tree T is said to be a 2-Tree or an extended binary tree if each node N has either 0 or 2 Children. In such a case, the nodes with 2 children are called internal nodes, and the nodes with 0 children are called external nodes. Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes.
Figure:
Linked Representation of Binary Tree: Consider a binary tree T. T will be maintained in memory by means of a linked representation which uses parallel arrays, INFO, LEFT & RIGHT, and a pointer variable ROOT as follows. First of all, each node N of T will correspond to a location K such that:

1. INFO[K] contains the data at the node N.
2. LEFT[K] contains the location of the left child of node N.
3. RIGHT[K] contains the location of the right child of node N.

Furthermore, ROOT will contain the location of the root R of T. if any subtree is empty, then the corresponding pointer will contain the null value, if the tree T itself is empty, then the ROOT will contain the null value.
Figure:

Traversing Binary Tree: There are three standard ways of traversing a binary tree T with root R. These three algorithms, called Preorder, Inorder and Postorder Traversal and that are as follows:

Preorder

Algorithm PREORDER (T, R)

```
//Here PREORDER  is an algorithm which is used to traverse the Binary
//Tree T with Root R using the technique PREORDER.
Step 1: Start.
Step 2: Process the root node.
Step 3: Traverse the left subtree in PREORDER.
Step 4: Traverse the right subtree in PREORDER.
Step 5: Exit.
```

Inorder

Algorithm INORDER (T, R)

```
//Here INORDER is an algorithm which is used to traverse the Binary //Tree
T with Root R using the technique INORDER.
Step 1: Start.
Step 2: Traverse the left subtree in INORDER.
Step 3: Process the root node.
Step 4: Traverse the right subtree in INORDER.
Step 5: Exit.
```

Postorder

Algorithm POSTORDER (T, R)

```
//Here POSTORDER is an algorithm which is used to traverse the Binary
//Tree T with Root R using the technique POSTORDER.
Step 1: Start.
Step 2: Traverse the left subtree in POSTORDER.
Step 3: Traverse the right subtree in POSTORDER.
Step 4: Process the root node.
Step 5: Exit.
```

Example:

Header Nodes (Threads): Suppose a binary tree T is maintained in the memory by means of a linked representation. Sometimes an extra, special node, called header node, is added to the beginning of T. When this extra node is used, the tree pointer variable, which we will call HEAD(instead of ROOT), will point to the header node, and the left pointer of the header node will point to the root of T.

Suppose a binary tree T is empty. Then T will still contain a header node, but the left pointer of the header node will contain the null value. Thus the condition LEFT[HEAD]=NULL will indicate an empty tree.

Figure:

Threads (inorder threading): Consider again the linked representation of a binary tree T. Approximately half of the entries in the pointer fields LEFT and RIGHT will contain null elements. This space may be more efficiently used by replacing the null entries by some other type of information. Specifically, we will replace certain null entries by special pointers which point to nodes higher in the tree. These special pointers are called threads, and binary trees with such pointers are called threaded binary trees.

There may be many ways to thread a binary tree T, each threading will correspond to a particular traversal of T. and also one may choose a one-way or a two-way threading.

1. One-way inorder threading: Figure.
2. Two-way inorder threading: Figure.
3. One-way inorder threading with header node: Figure.

Binary Search Tree: Suppose T is a binary tree. Thane T is called a binary search Tree or binary sorted tree. Is each node N of T has the following property:

The value at N is greater than every value in the left subtree of N and is less than every value in the right subtree of N. it is not difficult to see that this property guarantees that the inorder traversal of T will yield a sorted listing of the elements of T.

Figure:

Searching and inserting in Binary Search Tree: Suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the BST T, or insert ITEM as a new node in it's appropriate place in the tree.

a) Compare ITEM with the root node N of the T.
   1. If ITEM < N, proceed to the left child of N.
   2. If ITEM > N, proceed to the right child of N.
b) Repeat step a) until one of the following occurs:
   1. We meet a node N such that ITEM=N. in this case we say "Search is Successful".
   2. We meet an empty Subtree, which indicates that the "Search is Unsuccessful", and we insert ITEM in place of empty Subtree.

In other words, proceed from the Root R down through the tree T until finding ITEM in T or inserting ITEM as the terminal node in T.

Figure:

Deleting in a Binary Search Tree: Suppose T is a BST, and suppose an ITEM of information is given, and we are to delete the ITEM from T. This algorithm deletes the ITEM from T.

The way N is deleted from the T depends primarily on the no. of children of Node N. There are 3 cases that are as follows:

Case 1: N has no children. Then N is deleted from T by simply replacing the location of N in the Parent Node P(N) by the Null Pointer.

Case 2: N has Exactly one child. Then N is deleted from T by simply replacing the location of N in the Parent Node P(N) by the location of the only child of N.

Case 3: N has two children. Let S(N) denotes inorder successor of N. Then N is deleted from T by first deleting S(N) from T (by using Case 1 or Case 2) and then replacing node N in T by the node S(N).

Figure:

HEAP & HEAPSORT: Suppose H is a Complete Binary Tree with n elements. Then H is called a HEAP, or a MAXHEAP, if each node N of H has the following property: the value on N is greater than or equal to the value at each of the children of N. Accordingly the value at N is greater than or equal to the value at any of descendants of N. A MINHEAP analogously defined as: the value at N is less than or equal to the value at any of the children of N.

Inserting into a heap: Suppose H is a heap with N elements and suppose an ITEM of information is given. We insert the ITEM into the heap H as follows:

1. First adjoin the ITEM at the end of H so that H still a Complete Binary Tree, but not necessarily a heap.
2. Then let ITEM rise to it's appropriate place in H so that H is finally a heap.

Figure:

Deleting a root of a heap: Suppose H is a heap with N elements and suppose we want to delete the Root R of H. this is accomplished as follows:

1. Assign the Root R to some variable ITEM.
2. Replace the deleted node R by the Last node L of H so that H is still a Complete Binary Tree, but not necessarily a heap.
3. (Reheap) Let L sink to it's appropriate place in H so that H is finally a heap.

Figure:

BTreee(m-way Tree): A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:

- the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
- all leaves are on the same level
- all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
- the root is either a leaf node, or it has from two to *m* children
- a leaf node contains no more than $m - 1$ keys
- The number *m* should always be odd

Construction of BTree: Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

Figure:

Linked List: A Linked List, or One Way List, is a linear collection of Data elements, called nodes, where the linear order is given by means of Pointers. That is, each node is divided into two parts: the first part, called INFO Field, which contains the information of the element, and the second part, called the LINK Field or NEXTPOINTER Field, contains the address of the next node in the list. Every linked list also contains a special pointer, called START, which contains the address of Very first node of the linked list.

Traversal

Algorithm TRAVERSE (LIST, START, INFO, LINK)

```
//Here TREAVERSE is an algorithm, which is used to traverse all nodes of
//Linked list called LIST. START is the pointer that contains the address of
//first node of the LIST. INFO contains the information about the node and
//LINK contains the address of next node of LIST.
Step 1: Start.
Step 2: Set PTR:=START.
Step 3: Repeat steps 4 & 5 while PTR != NULL.
Step 4: PROCESS to INFO[PTR].
Step 5: Set PTR:=LINK[PTR].
Step 6: Exit.
```

Insertion

Algorithm INSERT (LIST, START, INFO, LINK, AVAIL, ITEM, LOC)

```
//Here INSERT is an algorithm, which is used to insert an ITEM on the
//Location LOC in the LIST. START is the pointer that contains the address
//of first node of the LIST. INFO contains the information about the node
//and LINK contains the address of next node of LIST. AVAIL is the address
//of new node that is to be inserted.
Step 1: Start.
Step 2: if AVAIL = NULL then print "OVERFLOW" and goto step-5.
Step 3: Set NEW:=AVAIL and AVAIL:=LINK[AVAIL].
Step 4: Set INFO[NEW]:=ITEM.
Step 5: if LOC=NULL then
            a. Set LINK[NEW]:=START.
            b. Set START:=NEW.
        Else
            a. Set LINK[NEW]:=LINK[LOC].
            b. Set LINK[LOC]:=NEW.
Step 6: Exit.
```

Deletion

Algorithm DELETE (LIST, START, INFO, LINK, LOC, LOCP)

```
//Here DELETE is an algorithm, which is used to delete an ITEM from the
//Location LOC in the LIST. START is the pointer that contains the address
//of first node of the LIST. INFO contains the information about the node
//and LINK contains the address of next node of LIST. LOCP is the address
//of previous node of deleting node.
Step 1: Start.
Step 2: if LOCP = NULL then
            Set START:=LINK[START].
```

```
        Else
             Set LINK[LOCP]:=LINK[LOC].
Step 3: Exit.
```

PROGRAM

```
#include<conio.h>
#include<stdio.h>
#include<alloc.h>
struct linked_list
{
        int info;
        struct linked_list *link;
};
typedef struct linked_list node;
void create(node*);
void print(node*);
node *find(node*,int);
node *delet(node *st);
node *insert(node *st);
void main()
{
        node *start,*loc;
        char cho;
        int ch,item;
        start=(node*)malloc(sizeof(node));
        create(start);
        do
        {
                printf("\nMenu\n1. Insert\n2. Delete\n3. show\n4.Search");
                printf("\nEnter your choice ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:
                                start=insert(start);
                                break;
                        case 2:
                                start=delet(start);
                                break;
                        case 3:
                                print(start);
                                break;
```

```
                    case 4:
                        printf("\nEnter the item ");
                        scanf("%d",&item);
                        loc=search(start,item);
                        if(loc==NULL)
                            printf("Unsuccessful");
                        else
                            printf("successful");
                        break;
                    default:
                        printf("\nWrong choice");
                }
                printf("\nDo you want more ");
                scanf("\n%c",&cho);
        }while(cho=='y');
        getch();
}
void create(node *st)
{
printf("\nEnter the number(Type -999 for end) ");
        scanf("%d",&st->info);
        if(st->info==-999)
        {
                st->link=NULL;
        }
        else
        {
                st->link=(node*)malloc(sizeof(node));
                create(st->link);
        }
}
void print(node *st)
{
        if(st->link!=NULL)
        {
                printf("%d->>",st->info);
                print(st->link);
        }
}
node *find(node *pt,int key)
{
        if(pt->link->info==key)
```

```
                return pt;
        else if(pt->link->link==NULL)
                        return NULL;
                else
                        find(pt->link,key);
}
node *search(node *pt,int key)
{
        if(pt->info==key)
                return pt;
        else if(pt->link==NULL)
                        return NULL;
                else
                        find(pt->link,key);
}
node *delet(node *st)
{
        int item;
        node *n,*p;
        printf("\nEnter the item to delete ");
        scanf("%d",&item);
        if(st->info==item)
        {
                st=st->link;
        }
        else
        {
                         n=find(st,item);
                         if(n==NULL)
                         {
                                printf("\nItem is not found");
                         }
                         else
                         {
                                p=n->link->link;
                                n->link=p;
                         }
        }
        return st;
}
node *insert(node *st)
{
        int item,ky;
        node *nw,*n;
        printf("\nEnter the item to insert and key (-999 for last) ");
        scanf("%d%d",&item,&ky);
        if(st->info==ky)
        {
                nw=(node*)malloc(sizeof(node));
                nw->info=item;
                nw->link=st;
                st=nw;
        }
        else
        {
                n=find(st,ky);
                if(n==NULL)
                {
                        printf("\nKey not found");
                }
                else
                {
                        nw=(node*)malloc(sizeof(node));
                        nw->info=item;
                        nw->link=n->link;
                        n->link=nw;
                }
        }
        return st;
}
```

HEADER LINKED LIST: a header linked list is a linked list which always contains a special node, called the header node, at the beginning of the list. The following are two kinds of widely used header lists:

1. **GROUNDED HEADER LIST:** A grounded header list is a header list where last node contains the null pointer.
   Figure:

2. **CIRCULAR HEADER LIST:** A circular header list is a header list where the last node points back to the header node.
   Figure:

POLYNOMIALS:

TWO WAY LINKED LISTS: A two way linked list is a linear collection of data elements, called nodes, where every node N is divided into three parts:

1. **INFO:** it contains the information about the node (i.e. **N**).
2. **FORW:** it contains the address of next Node of **N** in the list.
3. **BACK:** it contains the address of preceding Node of **N** in the list.

Figure:

SPARSE MATRIX (SPARSE ARRAY): Matrices with a relatively high proportion of 0 entries are called Sparse Matrices. Two general types of n-square sparse matrices, which occur in various applications, are pictured in following figure. The first matrix, where all entries above the main diagonal are zero or equivalently, where non-zero entries can only occur on or below the main diagonal, is called a lower triangular matrix. The second matrix, where non-zero entries can only occur on the diagonal or on elements immediately above or below the diagonal, is called Tri-diagonal matrix.

```
4 0 0 0 0
3 5 0 0 0
1 0 6 0 0
7 8 1 3 0
5 2 0 2 8
```

(Triangular Matrix)

```
5 3 0 0 0 0 0
1 4 3 0 0 0 0
0 9 3 6 0 0 0
0 0 2 4 7 0 0
0 0 0 3 1 0 0
0 0 0 0 6 5 8
0 0 0 0 0 3 1
```

(Tri-Diagonal Matrix)

Representation of SPARSE MATRIX in the form of Array:
Figure:
Representation of SPARSE MATRIX in Linked form:
Figure:

TRANSFORMING INFIX EXPRESSION INTO POSTFIX EXPRESSION:

```
Algorithm POSLISH(Q,P)
//Here POSLISH is an algorithm which is used to convert infix expression Q
//into postfix expression P.
Step-1: Start.
Step-2: Push "(" onto the STACK and add ")" to the end of Q.
Step-3: Scan Q from left to right and repeat steps 4 to 7 for each element of
        Q until the STACK is empty.
Step-4: if an operand is encountered, push it onto STACK.
Step-5: if a left parenthesis is encountered, push it onto STACK.
```

```
Step-6: if an operator ⊠ is encountered, then:
        a. Repeatedly pop from STACK and add to P each operator (on the top
           of STACK) which has the same precedence as or higher precedence
           than ⊠.
        b. Add ⊠ to STACK.
Step-7: if right parenthesis is encountered, then:
        a. Repeatedly pop from STACK and add to P each operator (on the top
           of STACK) until a left parenthesis is encountered.
        b. Remove the left parenthesis.
Step-8: Exit
```

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

EVALUATION OF A POSTFIX EXPRESSION:

```
Algorithm EVALPOSTFIX(P, VALUE)
//Here EVALPOSTFIX is an algorithm which is used to evaluate the VALUE
//of postfix expression P.
Step-1: Start.
Step-2: Add ")" to the end of P.
Step-3: Scan P from left to right and repeat steps 4 & 5 for each element of
        P until the sentinel ")" is encountered.
Step-4: if an operand is encountered, push it onto STACK.
Step-5: if an operator ⊠ is encountered, then:
        a) Remove the two top elements of STACK, where A is the top element
           and B is the next-to-top element.
        b) Evaluate B ⊠ A.
        c) Place the result of (b) back on the STACK.
Step-6: Set VALUE equal to the top element on STACK.
Step-7: Exit
```

Question: 5, 6, 2, +, *, 12, 4, /, -

Hashing

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

What is Hash Function?

- A fixed process converts a key to a hash key is known as a **Hash Function.**
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.
  What is Hash Table?
- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class.
- Hash table is synchronized and contains only unique elements.

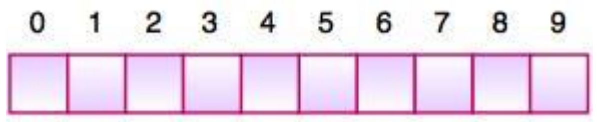
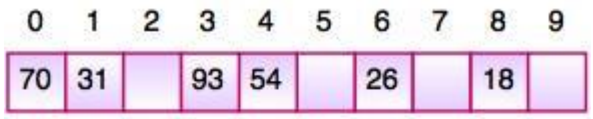


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.
- Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :**Hash Key = Key Value % Number of Slots in the Table**
- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	**Value % No. of Slots**	**Hash Value**
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3



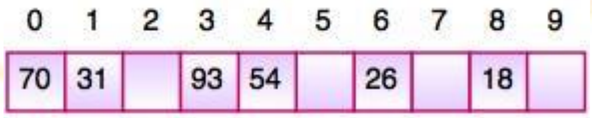
0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example , λ = 6/10.
- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

  Linear Probing
- Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs.
- Linear probing was invented by Gene Amdahl, Elaine M. McGraw and Arthur Samuel in 1954 and analyzed by Donald Knuth in 1963.
- It is a component of open addressing scheme for using a hash table to solve the dictionary problem.
- The simplest method is called Linear Probing. Formula to compute linear probing is:
  P = (1 + P) % (MOD) Table_size

For example,



0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

If we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem.

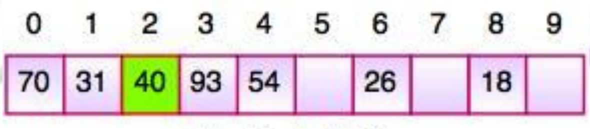
Linear probing solves this problem:

P = H(40)

40 % 10 = **0**

Position 0 is occupied by 70. so we look elsewhere for a position to store 40.

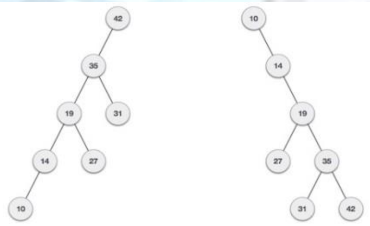
Using Linear Probing:
P= (P + 1) % table-size
(0 + 1) % 10 = **1**
But, position 1 is occupied by 31, so we look elsewhere for a position to store 40.
Using linear probing, we try next position : (1 + 1) % 10 = 2
Position 2 is empty, so 40 is inserted there.



0	1	2	3	4	5	6	7	8	9
70	31	40	93	54		26		18	

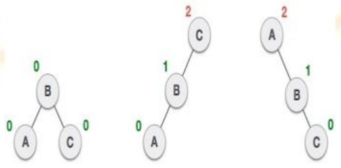
Fig. Hash Table

AVL Tree
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



If input 'appears' non-increasing manner If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.
Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.
Here we see that the first tree is balanced and the next two trees are not balanced −



Balanced Not balanced Not balanced

In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

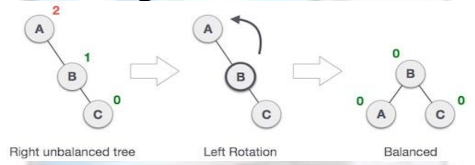
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

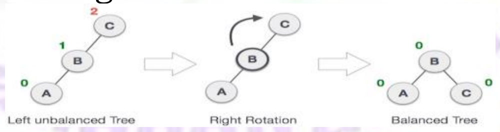
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

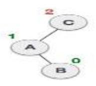
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

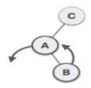
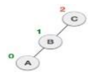
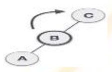
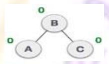


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

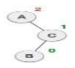
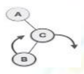
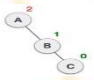
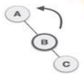
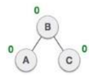
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation
The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Merge Sort: The merge sort algorithm is based on the classical divide-and-conquer paradigm. It operates as follows:

DIVIDE: Partition the n-element sequence to be sorted into two subsequences of n/2 elements each.
CONQUER: Sort the two subsequences recursively using the merge sort.
COMBINE: Merge the two sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.

Quick Sort: Quicksort is the currently fastest known sorting algorithm and is as follows:

- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- Quicksort, like merge sort, is a divide-and-conquer recursive algorithm.

Algorithm QUICKSORT(A,N,BEG,END,LOC)

```
//Here QUICKSORT is an algorithm which is used to Sort all elements of an Array //A with N elements using Quick Sort Technique. BEG and END are the boundary //positions of A. LOC keeps the track of pivot element.
  Step-1 Start
  Step-2 Set LEFT:=BEG, RIGHT:=END & LOC:=BEG.
  Step-3 Scan from RIGHT to LEFT:
      a. Repeat while A[LOC]<=A[RIGHT] & LOC!=RIGHT
                 Set RIGHT:=RIGHT-1.
      b. if LOC=RIGHT then Return.
      c. if A[LOC]>A[RIGHT] then:
             • Set   TEMP:=A[LOC].
             • Set   A[LOC]:=A[RIGHT].
             • Set   A[RIGHT]:=TEMP.
             • Set   LOC:=RIGHT.
             • Go to Step-4.
  Step-4 Scan from LEFT to RIGHT:
      a. Repeat while A[LOC]>=A[LEFT] & LOC!=LEFT
                 Set LEFT:=LEFT+1.
      b. if LOC=LEFT then Return.
      c. if A[LOC]<A[LEFT] then:
             • Set   TEMP:=A[LOC].
             • Set   A[LOC]:=A[LEFT].
             • Set   A[LEFT]:=TEMP.
             • Set   LOC:=LEFT.
             • Go to Step-4.
```