1. What is data structure?

    Data Structure is an arrangement of data in computer's memory (or sometimes on a disk) so that we can retrieve it & manipulate it correctly and efficiently.

2. List out the areas in which data structures are applied extensively?

    Data structures are essential in almost every aspect where data is involved. In general, algorithms that involve efficient data structure is applied in the following areas:

    Numerical analysis,
    Operating system,
    Statistical analysis
    Compiler Design,
    Operating System,
    Database Management System,
    Statistical analysis package,
    Numerical Analysis,
    Graphics,
    Artificial Intelligence,
    Simulation

3. What are the major data structures used in the following areas :
    RDBMS, Network data model & Hierarchical data model.

    ➢ RDBMS   Array (i.e. Array of structures)
    ➢ Network data model Graph
    ➢ Hierarchical data model Trees

4. What are the limitations of array over linked list?
    *   Array does not grow dynamically (i.e length has to be known)
    *   Inefficient memory management.
    *   In ordered array insertion is slow.
    *   In both ordered and unordered array deletion is slow.
    *   Large number of data movements for insertion & deletion
    *   which is very expensive in case of array with large number of elements.

5. Sorting Algorithm:

Bubble sort:

```
for( int i = 0; i < passes; i++ ) {
        for( int j = 0; j < comps - i; j++ )
        {
          if( arr[j+1] < arr[j] )
          {
           int temp = arr[j+1];
           arr[j+1] = arr[j];
           arr[j] = temp;
          }
        }
}
```

➢ Number of iterations : (n-1)(n-1)
➢ $T \alpha\ n^2 - 2n + 1$
➢ If $n > 1, n^2 > n$ then all lower order terms are ignored
➢ $T \alpha\ n^2$
➢ Time complexity $= O(n^2)$

Selection Sort:

```
for( int i = 0; i < size-1; i++ ) {
     int min = i;
     for( int j = i+1; j < size; j++ ) {
        if( arr[j] < arr[min] ) {
          min = j;
        }
     }
     int temp = arr[i];
     arr[i] = arr[min];
     arr[min] = temp;
  }
```

➢ Number of iterartions:$1+2+3+.....+n-1=n(n-1)/2$
➢ $T \alpha\ n(n-1)/2$
➢ $T \alpha\ n^2 - n$
➢ If $n > 1, n^2 > n$ then all lower order terms are ignored
➢ $T \alpha\ n^2$
➢ Time complexity $= O(n^2)$

Insertion Sort:

```
for( int  j = 0; j < size; j++ ) {
     for( int i = 0; i < j; i++ ) {
       if( arr[j] < arr[i] ) {
          int temp = arr[j];
          arr[j] = arr[i];
          arr[i] = temp;
       }
     }
  }
```

➢   Base case: $O(n)$
➢   Average/worst case: $O(n^2)$

Quick Sort :

```
void quickSort( int * arr, int left, int right ) {
    if( left >= right ) {
    return;
  }
  int leftHold = left, rightHold = right;
  int pivot = arr[left];
  while( left < right ) {
     while( arr[right] > pivot && left != right ) {
      right--;
    }
     if( left != right ) {
      arr[left] = arr[right];
      left++;
    }
     while( arr[left] < pivot && left != right ) {
      left++;
    }
     if( left != right ) {
      arr[right] = arr[left];
      right--;
    }
  }
  arr[left] = pivot;
  quickSort(arr, leftHold, left-1 );
  quickSort(arr, left+1, rightHold );   }
```

> ➤ Average/worst case: O(n log n)
> ➤ Worst case: O(n²)

Merge Sort:

```
if( left < right ) {
      int mid = ( left + right ) / 2;
      mergeSort( arr, left, mid );
      mergeSort( arr, mid+1, right );
      merge( arr, left, mid, mid+1, right );
      }
```

Average/worst case: O(n log n)

3

```
void merge( int * arr, int leftStart, int leftEnd, int rightStart, int rightEnd ) {
  int temp[100];
  int tempPos = leftStart;
  int numEle = ( rightEnd - leftStart ) + 1;
  while( ( leftStart <= leftEnd ) && ( rightStart <= rightEnd ) ) {
    if( arr[leftStart] < arr[rightStart] ) {
      temp[tempPos] = arr[leftStart];
      tempPos++;      leftStart++;
    }
    else {
      temp[tempPos] = arr[rightStart];
      tempPos++;      rightStart++;
    }
}
  while( leftStart <= leftEnd ) {
   temp[tempPos] = arr[leftStart];
   tempPos++;
   leftStart++;
 }
  while( rightStart <= rightEnd ) {
   temp[tempPos] = arr[rightStart];
   tempPos++;      rightStart++;
 }
  for( int i = 1; i <= numEle; i++ ) {
   arr[rightEnd] = temp[rightEnd];
   rightEnd--;
 }  }
```

Binary Search:

```
while(left<=right)
{
        mid=(left + right)/2;
        if(key==arr[mid])
            return mid;
        if(key<arr[mid])
            right=mid-1;
       else
            left=mid+1;
}
```

6. Linked List?

We can overcome the drawbacks of the sequential storage.

If the items are explicitly ordered, that is each item contained within itself the address of the next Item.

In Linked Lists elements are logically adjacent, need not be physically adjacent. Node is divide into two part.

info	link

Operation can be performed on linked List:
1. insert()
2. insertAtFirst()
3. insertAtLast()
4. insertByPos()
5. deleteAtFirst()
6. deleteAtLast()
7. deleteByPos()
8. deleteByVal()
9. deleteAllNode()
10. Reserve()
11. midElement()

1.   Insert by value

```
Node * newNode = new Node(data);
        if (newNode == NULL) {
                return false;
        }
        if (head == NULL) {
                head = newNode;
                return true;
        }
        Node * last = head;
        while (last->getNext() != NULL) {
                last = last->getNext();
        }
        last->setNext(newNode);
        return true;
```

Algorithm for insert node:

1. Create node
2. Check for empty list
   a) If list is empty
      Set head is equal to newnode
   b) If list is not empty,
      ➤ Create temp node
      ➤ Locate temp to last node
      ➤ Set last node next to newnode

2.   To Insert node at first location: O(1)

```
bool insertAtFirst()
{
if(head==NULL)
{
    head=newnode;
    return true;
}
newnode->next=head;
head=newnode;
return true;
}
```

Algorithm for insert at first

1.   check for empty list
   a.   if list is empty
        set head=newnode
   b.   if list is not empty
        set newnode->next=head;
        head=newnode;

3.   To Insert node at last location: O(n)

```
bool insertAtLast()
{
if(head==NULL)
{
    head=newnode;
    return true;
}
Node* last=head;
//locate temp to last node
while(last->next!=NULL)
{
last=last->next;
}
last->next=newnode;
return true;
```

Algorithm for insert at last

1.   check for empty list
   a.   if list is empty
        set head=newnode
   b.   if list is not empty
        i.   locate last pointer to
              last node
        ii.   set last next to
              newnode
              last->next=newnode

4. Insert node at given location: O(n)

```cpp
bool insertAtPos(int pos,int val)
{
    //check for invalid position
    if (position <= 0 || ( head == NULL && position > 1 ))
    {
            return false;
    }

    Node * newNode = new Node(data);
    //check for newnode get memory
    if (newNode == NULL) {
            return false;
    }
    //check for position 1
    if (position == 1) {
            if (head != NULL) {
                    newNode->setNext(head);
            }
            head = newNode;
            return true;
    }

    Node * prev = head;
    // position other than 1
    for (int i = 1; i < position - 1; i++) {
            prev = prev->getNext();

            if (prev == NULL) {
                    delete newNode;          Invalid position
                    return false;
            }
    }
    newNode->setNext(prev->getNext());
    prev->setNext(newNode);
    return true;
}
```

5. Delete node at first location: O(1)

```
bool deleteAtFirst()
{
If (head==NULL)
{
        return false;
}
Node* del =head;
head=del->next;
delete[] del;
return true;
}
```

Algorithm for delete at first

1. check for empty list
   a. if list is empty
      no node will be deleted
      return false;
   b. if list is not empty
      i.   locate del pointer to
           first node
      ii. set head to del next
           head=del->next
      iii. delete del pointer

6. Delete node at last location: O(n)

```
bool deleteAtLast()
{
if(head==NULL)
{
    return false;
}
Node* del=head,*prev=head;
//locate temp to last node
While(del->next!=NULL)
{
prev=del;
del=del->next;
}
prev->next=NULL;
return true;
}
```

Algorithm for delete at last

1. check for empty list
   a. if list is empty
      set head=newnode
   b. if list is not empty
      i.   locate last pointer to
           last node
      ii. set last next to
           newnode
           last->next=newnode

8

7. Delete node at given location: O(n)

```cpp
bool LinkedList::deleteByPos(int position) {

    //Check for invalid pos
    if (position <= 0) {
        return false;
    }
    //Check for position 1
    if (position == 1) {
        Node * del = head;
        head = head->getNext();
        delete del;
        return true;
    }
    // position other than 1
    Node * prev = head;
    for (int i = 1; i < position - 1; i++) {
        prev = prev->getNext();

        if (prev == NULL) {
            return false;
        }
    }
    Node * del = prev->getNext();
    prev->setNext( del->getNext() );
    delete del;
    return true;

}
```

8. Delete node with given value: O(n)

```cpp
bool LinkedList::deleteByVal(int data) {
    if (head == NULL) {
        return false;
    }
    //check for first node
    if (head->getData() == data) {
        Node * del = head;
        head = del->getNext();
        delete del;
        return true;
    }
    //check for given data
    Node * del = head, * prev = head;
    while (del->getData() != data) {
        prev = del;
        del = del->getNext();
        if (del == NULL) {
            return false;          Invalid data
        }
    }
    prev->setNext( del->getNext() );
    delete del;
    return true ;
}
```

9. Delete all node : O(n)

```
while( head!=NULL)
        deleteAtFirst();
```

10. Reverse list

```
void LinkedList:: Reverse() {
        Node * stack[100];
        int top = -1;
        Node * temp = head;
        while (temp) {
                stack[++top] = temp;
                temp = temp->getNext();
        }
        while (top != -1) {
                cout << stack[top--]->getData() << " ";
        }
        cout << endl;
}
```

11. Find out middle element in linked list:

```
Method 1(using two loop):

 count=0;
 while(temp!=NULL)
   {
        count++;
        temp=temp->getNext();
   }
   for(i=0;i<count/2;i++)
   {
        temp=temp->getNext();
   }
   return temp->getData();
```

```
Method 2(using one loop):

Node* temp=head, prev*=head;
while(temp=!NULL && temp->getNext()!=NULL)
{

        temp=temp->getNext()->getNext();
        prev=prev->getNext();
}

        return prev->getData();
```

7. Singly circular Linked List?
   Last node's next pointer keeps address of first (head) node.

1. Insert at circular list = O(n)

```
bool CircularLinkedList :: insert(int data) {
    Node * newNode = new Node( data );
//Check for memory allocation to newnode
    if( newNode == NULL ) {
    return false;
  }
//Check for empty list  (if yes insert node at first)

  if( head == NULL ) {
    head = newNode;
    newNode->setNext( newNode );
    return true;
  }
//create temporary pointer and locate it to last node
  Node * last = head;
  while( last->getNext() != head ) {
    last = last->getNext();
  }
  newNode->setNext( head ); //set last(newnode) node next to head
  last->setNext( newNode );

  return true;
}
```

2. Insert at first location= O(1)

```
bool insertAtFirst()
{
  if(head==NULL)
    {
        head=newnode;
        newnode->next=head;
        return true;
    }
    Node* temp=head;

    while(temp->next!=Null)
    {
            temp=temp->next;
    }
    Newnode->next=head;
    head=newnode;
    temp->next=head;
    return true;
}
```

3. Insert at last location= O(1)

```
bool insertAtLast()
{
   if(head==NULL)
     {
         head=newnode;
         newnode->next=head;
         return true;
     }
     Node* temp=head;

     while(temp->next!=Null)
     {
             temp=temp->next;
     }
     temp->next=newnode;
     newnode->next=head;

     return true;
}
```

4. delete at first location= O(1)

```
bool deleteAtFirst() circular
{
   if(head==NULL)
     {
         return false;
     }
     Node* del=head;
     head=del->next;
     delete[] del;
     return true;
}
```

5. delete at last location= O(1)

```
bool deleteAtLast()
{
   if(head==NULL)
     {
         return false;
     }
     Node* del=head,*prev=head;
     while(del->next!=NULL)
     {
             prev=del;
             del=del->next;
     }
     Node* del=head;
     head=del->next;
     delete[] del;
     return true;
}
```

8. Doubly Linked List?

1. Insert in doubly list:

```cpp
bool DoublyLinkedList::insert(int data) {

    Node * newNode = new Node(data);

    if (newNode == NULL) {
        return false;
    }

    if (head == NULL) {
        head = newNode;
        return true;
    }

    Node * last = head;
    while (last->getNext() != NULL) {
        last = last->getNext();
    }

    last->setNext(newNode);
    newNode->setPrev(last);

    return true;

}
```

2. delete in doubly list:

```cpp
bool DoublyLinkedList::deletebyPos(int position) {
    if (position <= 0 || head == NULL ) {
        return false;
    }
    if (position == 1) {
        Node * del = head;
        head = head->getNext();
        if (head != NULL) {
            head->setPrev(NULL);
        }
        delete del;
        return true;
    }
    Node * del = head;
    for (int i = 1; i < position; i++) {
        del = del->getNext();
        if (del == NULL) {
            return false;
        }
    }
    del->getPrev()->setNext(del->getNext());
    if (del->getNext() != NULL) {
        del->getNext()->setPrev(del->getPrev());
    }
    delete del;
    return true;

}
```

Stack:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). Following are operations are performed with stack.

1. Push()
2. Pop()
3. Peek()

We can implement stack by two ways,

➢ Using array
➢ Using linkedlist

Stack implementation using Array:

isFull() & isEmpty():

```
bool isEmpty() {
    return top == -1;
}

bool isFull() {
    return top == ( size - 1 );
}
```

Push ():

```
bool push( int data ) {
    if( isFull() ) {
        cout<<"List is full"<<endl;
        return false;
    }

    arr[++top] = data;
    return true;
}
```

Pop ():

```
int pop() {
    if( isEmpty() ) {
            cout<<"List is empty"<<endl;
        return -999;
    }
        return arr[--top];
    }
```

Peek ():

```
int peek() {
        if( isEmpty() ) {
        return -999;
    }
        return arr[top];
    }
```

Stack implementation using Linked List:

push() :

```
bool push(int data)
{
        node* newnode =new node(data);
        if (newnode == NULL)
        {
                return false;
        }
        newnode->setData( data);
        newnode->Setnext( head);
        head = newnode;
        return true;
}
```

pop():

```
bool pop()
{
        node* del = head;
        if (head == NULL)
        {
                return false;
        }
        head = del->getNext();
        delete [] del;
        return ture;

}
```

peek():

```
int peek()
{
        return head->getData();

}
```

Queue:

A Queue is an ordered collection of items into which new items may be inserted at rear end and items are deleted at one end called front.

Types of Queue:

1. **Linear Queue**
2. **Circular Queue**
3. **Priority Queue**
4. **Dequeue (Double Ended Queue)**

Queue implementation using Array:

isFull() & isEmpty():

```
bool isEmpty() {
    return ( front == -1 && rear == -1)||( front > rear );
}

bool isFull()
    {
    return rear = ( size - 1 );
}
```

Insert ():

```
bool insert( int data ){
    if( isFull() ) {
        return false;
    }
    arr[++rear] = data;
    if( front == -1 ) {
        front = 0;
    }
    return true;
}
```

delete() :

```
int deleteData() {
    if( isEmpty() ) {
        return -999;
    }
    return arr[front++];
}
```

Circular Queue implementation using Array:

isFull() & isEmpty():

```
bool isEmpty() {
    return front == rear;
}
bool isFull() {
    return  ( front == -1 && rear ==( size - 1 )) ||
    ( (rear+1) % size == front );
}
```

Insert():

```
bool insert( int data ){
    if( isFull() ){
        return false;
    }

    rear = ( rear + 1 ) % size;
    arr[rear] = data;
    return true;
}
```

delete() :

```
int deleteData() {
    if( isEmpty() ){
        return -999;
    }
    front = ( front + 1 ) % size;
    return arr[front];
}
```

9. Difference between Stack and Queue?

Sr.No	Stack	Queue
1	A Stack Data Structure works on Last In First Out (LIFO) principle.	A Queue Data Structure works on First In First Out (FIFO) principle.
2.	Push and pop operations are done from same end i.e "top"	Push and pop operations are done from same end i.e "rear" and "front" respectively
3.	You can implement multi-stack approach	Threre are four types of Queue i.e linear, circular, priority and dequeue.
4.	Application: • Used in infix to postfix conversion, • scheduling algorithms • depth first search and evaluation of an expression	Application: • Printer mainatains queue of documents to be printed • OS uses queues for many functionalities : Ready Queue, Waiting Queue, Message Queue • To implements algo like Breadth first search.

10. Infix ,prefix, postfix

These are notations to represent math equations.

- Infix: A+B
- Prefix: +AB
- Postfix: AB+

To convert infix to prefix /postfix considers the priorities:

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (type)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Infix to postfix Evaluation rules:
1. Scan the infix exp. From left to right
2. If scan character is left parenthesis then push it into stack

3. If scan character is operand than that will display into postfix expression.
4. If scan character is operator than that will push into operator stack.
5. If scan character has higher precedence than stack operator then scanned operator will push into operator stack.
6. If scan character has less or equal precedence than stack operator then stack operator will pop out from stack to postfix expression and scanned operator will push into operator stack.
7. If scan character is right parenthesis then till left parenthesis of stack all operators will pop to postfix expression and left will remove from stack.
8. Repeat step 1 to step 6 until end of expression

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

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Infix to prefix Evaluation rules:

1. Reserve the input string or start from right of the infix expression
2. Examine the next element in the input.
3. If it is operand , add it to output string .

4. If it is closing parenthesis ,push it on to stack.
5. If it is an operator ,then
➢ if stack is empty , push operator on stack.
➢ If the top of stack is closing parenthesis , push operator on stack
➢   If it has same or higher priority then the top of stack push operator on   stack
➢ If it has lower priority the top, the pop stack and add it to post fix expression and push operator into stack
➢ Else pop the operator from the stack and add it to output string

6. if it is an opening parenthesis ,pop operators from stack and ass them to output string until a closing parenthesis is encountered.pop and discard the closing parenthesis.
7. If there is more input go to step 2.
8. If there is no more input , unstack the remaining operators and them to output string.
9. Reserve the output string.

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

Input	Prefix_Stack	Stack
G	G	(empty)
+	G	+
)	G	+)
)	G	+))
F	GF	+))
-	GF	+)) -
E	GFE	+)) -
(	GFE-	+)
/	GFE-	+) /
)	GFE-	+) /)
D	GFE-D	+) /)
+	GFE-D	+) /) +
C	GFE-DC	+) /) +

11. Tree

A tree consists of nodes connected by edges, which do not form cycle.
For collection of nodes & edges to define as tree, there must be one & only one path from the root to any other node.
A tree is a connected graph of N vertices with N-1 Edges.

Tree Terminologies:

1. **Node:** A node stands for the item of information plus the branches of other items.
2. **Siblings:** Children of the same parent are siblings.

3. **Degree:** The number of sub trees of a node is called degree. The
4. degree of a tree is the maximum degree of the nodes in the tree.
5. **Leaf Nodes:** Nodes that have the degree as zero are called leaf nodes or terminal nodes. Other nodes are called non terminal nodes.
6. **Ancestor:** The ancestor of a node are all the nodes along the path from the root to that node.
7. **Level:** The level of a node is defined by first letting the root of the tree to be level = 1or level=0.
8. **Height/Depth:** The height or depth of the tree is defined as the maximum level of any node in the tree.

Binary Search Tree
A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –
The left sub-tree of a node has a key less than or equal to its parent node's key.
The right sub-tree of a node has a key greater than to its parent node's key.

Insert () in to Binary search tree:

```cpp
bool BST::insert(int data) {
    //create a node
        Node * newNode = new Node( data );
    //check for newnode and for duplicate data point
        if( newNode == NULL ) {
            return false;
        }
    //check for tree is empty
        if( root == NULL ) {
            //if empty
            root = newNode;
            return true;
        }
    //if tree is not empty create temporary pointer
        Node * temp = root;
    //compare data
        while( temp->getData() != data ) {
            if( data < temp->getData() ) {
            //insert to left
            //check if temp has left child
            if( temp->getLeft() == NULL ) {
                temp->setLeft( newNode );
                return true;
            }
            temp = temp->getLeft();
        }
        else {
            //insert to right
            //check if temp has right child
            if( temp->getRight() == NULL ) {
                temp->setRight( newNode );
                return true;
            }
            temp = temp->getRight();
        }}
    delete newNode;
    return false;
```

Delete () in to Binary search tree:

```
bool BinarySearchTree :: deleteData(int data) {
        if (root == NULL) {
                return false;
        }
        Node * del = root, *parent = root;
        //step 1 locate the node to be deleted along with the parent
        while (data != del->getData()) {
                if (data < del->getData()) {
                        //to left
                        parent = del;
                        del = del->getLeft();
                }
                else {
                        //to right
                        parent = del;
                        del = del->getRight();
                }
                if (del == NULL) {
                        return false;
                }
        }
        while (true) {
                //check if del is terminal
                if (del->getLeft() == NULL && del->getRight() == NULL) {

                        //check if the del is root node
                        if (del == root) {
                                delete del;

                                root = NULL;

                                return true;
                        }

                        if (del == parent->getLeft()) {

                                parent->setLeft(NULL);

                        }
                        else {
                                parent->setRight(NULL);
                        }
                        delete del;
                        return true;
                }
```

```
//if del is non terminal node
            if (del->getLeft()) {
                    //select max from left
                    Node * max = del->getLeft();
                    parent = del;


                    while (max->getRight()) {

                            parent = max;

                            max = max->getRight();
                    }
                    int temp = max->getData();
                    max->setData(del->getData());
                    del->setData(temp);
                    del = max;
            }

            else {
                    //min from right
                    Node * min = del->getRight();
                    parent = del;

                    while (min->getLeft()) {
                            parent = min;
                            min = min->getLeft();
                    }

                    int temp = del->getData();
                    del->setData(min->getData());
                    min->setData(temp);
                    del = min;
            }
        }
        return false;
}
```
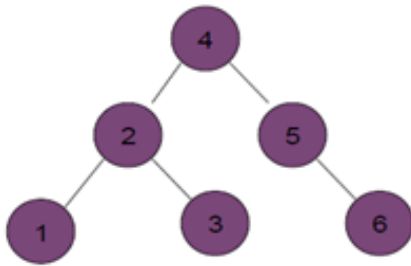
Tree traversals:

> ➢ Preorder(Root-Left-Right)
> 421356
>
> ➢ Inorder(Left-Root-Right)
> 123456
>
> ➢ Postorder(Left-Right-Root)
> 132654

PreOrder:

```
void BST::preOrder() {
    Node * temp = root;
    Node * stack[100];
    int top = -1;
    cout<<"Pre : ";
    while( temp != NULL || top != -1 ) {
        while( temp ) {
            cout<<temp->getData()<<" ";
            stack[++top] = temp;
            temp = temp->getLeft();
        }
        temp = stack[top--];
        temp = temp->getRight();
    }
```

Algorithm for preorder Traversal:

1. Visit node
2. Backup its address (use stack)
3. goto left child
4. If temp is null then pop stack
5. go back to popped parent and goto it's right
6. Repeat till stack is empty or temp is not NULL

Inorder:

```
void BST::inOrder() {
    Node * temp = root;
    Node * stack[100];
    int top = -1;
    cout<<"In : ";
    while( temp != NULL || top != -1 ) {
        while( temp ) {
            stack[++top] = temp;
            temp = temp->getLeft();
        }
        temp = stack[top--];
        cout<<temp->getData()<<" ";
        temp = temp->getRight();
    }
}
```

Algorithm for Inorder Traversal:

1. Push temp to stack
2. Goto left of temp
3. Repeat till temp is NULL
4. now pop stack & get back to popped node
5. visit that popped node and then shift to its right child
6. Repeat till stack is empty or temp is not NULL

26

Postorder:

```
void BST::postOrder() {
    typedef struct {
        Node * address;
        char flag;
    } Pair;
    Node * temp = root;
    Pair stack[100];
    int top = -1;
    while( temp != NULL || top != -1) {
        while( temp ) {
            Pair p;
            p.address = temp; p.flag = 'L';
            stack[++top] = p;
            temp = temp->getLeft();
        }
        Pair p = stack[top--];
        if( p.flag == 'L') {
            temp = p.address->getRight();
            p.flag = 'R';
            stack[++top] = p; }
        else {
            cout<<p.address->getData()<<" ";
        }}}
```

Graph:

A Graph is a collection of nodes, which are called vertices 'V', connected in pairs by line segments, called Edges E.
Sets of vertices are represented as V(G) and sets of edges are represented as E(G).

So a graph is represented as G = (V, E).
There are two types of Graphs
- Undirected Graph
- Directed Graph

Directed Graph are usually referred as Digraph for Simplicity. A directed Graph is one, where each edge is represented by a specific direction or by a directed pair <v1, v2>.Hence <v1, v2> & <v2, v1> represents two different edges.

Terminology related to Graph:

- Out – degree: The number of Arc exiting from the node is called the out degree of the node.
- In- Degree: The number of Arcs entering the node is the In degree of the node.
- Sink node: A node whose out-degree is zero is called Sink node.
- Path: path is sequence of edges directly or indirectly connected between two nodes.
- Cycle: A directed path of length at least L which originates and terminates at the same node in the graph is a cycle

The graphs can be implemented in two ways
1. Array Method
2. Linked List

Graph traversal
1. Breadth first search

```cpp
void Graph::bfs(int v) {
        int * visited = new int[noOfVertices] ;
        int q[100];
        int front = -1, rear = -1;
        q[0] = v;
        front = rear = 0;
        cout << "BFS : ";
        while (front <= rear) {
        v = q[front++];
        if (visited[v] == 0) {
        cout << vertices[v] << " ";
        visited[v] = 1;
        for (int i = 0; i < noOfVertices; i++) {
                if (adjMat[v][i] == 1 && visited[i] == 0) {
                q[++rear] = i;
                }}}
        delete[] visited;

}
```

2. Depth first search

```cpp
void Graph::dfs(int v) {
        int * visited = new int[noOfVertices] ;
        int stack[100];
        int top = -1;
        cout << "DFS : ";
        cout << vertices[v] << " ";
        visited[v] = 1;
        stack[++top] = v;
        while (top != -1) {
                for (int i = 0; i < noOfVertices; i++) {
                        if (adjMat[v][i] == 1 && visited[i] == 0) {
                                cout << vertices[i] << " ";
                                visited[i] = 1;
                                stack[++top] = i;
                                v = i;
                                i = -1;
                        }
                }
                v = stack[top--];}

}
```