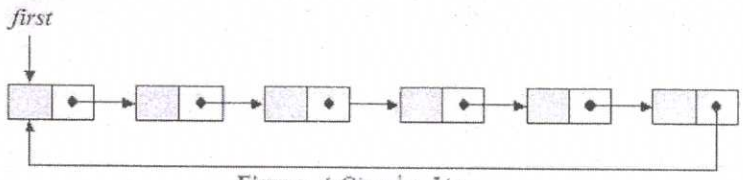
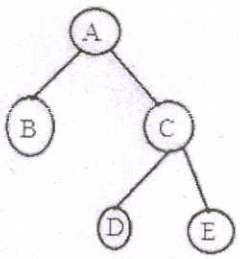
SCHEME OF VALUATION

(Scoring Indicators)

Revision: 15 **Course Code: 4133**

Course Title: DATA STRUCTURES

Qst. No.	Scoring Indicator	Split up score	Sub Total	Total
I	**PART – A**			10 marks
1.	• An abstract data type (ADT) is a set of objects together with a set of operations. How the set of operations is implemented is not mentioned anywhere in the ADT's definition. • Example: array, stack, queue, graph.	2 marks	2 Marks	
2.	i) Queue. ii) Stack.	1 mark for each	2 marks	
3.	• A *circular list* can be obtained by modifying the singly linked list so that the *link* field of the last node points to the first node in the list. Or  *Figure: A Circular List*	Definition : or Figure: 2 marks	2 Marks	
4.	• A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the *left subtree* and the *right subtree.* • Any sample figure  	2 marks	2 Marks	

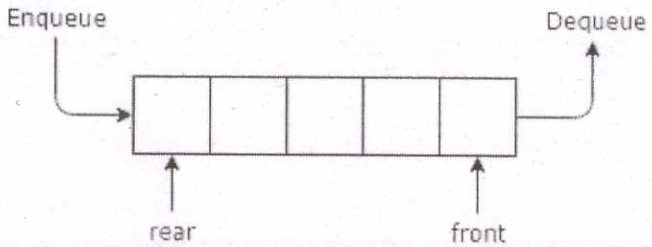
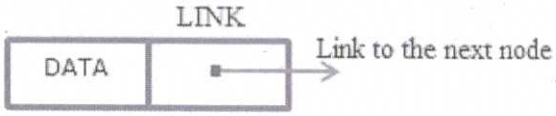
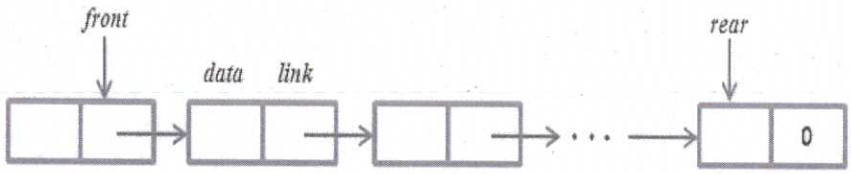
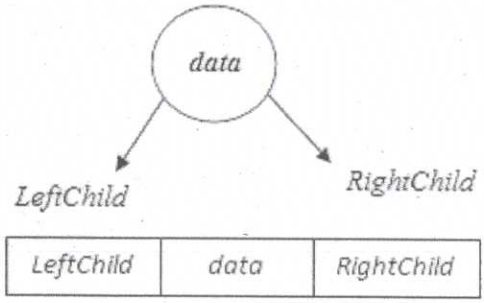
5.	• In linear search time complexity is **O(n)**. • In binary search time complexity is **O(log n)**	1 mark For each	2 marks	
II 1.	**PART – B** **Postfix** • The postfix form of given infix expression is: (A + (B * C) – D) / (E – F) (A + (B C *) – D) / (E F –) (A (B C *) + – D) / (E F –) (A (B C *) + D –) / (E F –) A B C * + D – E F – / **Prefix** • The prefix form of given infix expression is: (A + (B * C) – D) / (E – F) (A + (* B C) – D) / (– E F) (+ A * B C – D) / (– E F) (– + A * B C D) / (– E F) / – + A * B C D – E F	3 marks for each	6 marks	30 marks
2.	**Queue:** • A queue is an ordered collection of homogeneous data elements where insertion and deletion operations take place at two extreme ends i.e., it uses the principle of First In First Out (FIFO). • Ordering of the elements is in linear fashion. The queue insertion (called ***enqueue***) operation takes place at the ***"rear"*** end and deletion (called ***dequeue***) operation takes place at the ***"front"*** end.  	Queue definition: 1 mark + Front & rear: 1 mark + Figure / description: 1 mark + Insert :	6 marks	

	Figure: Queue data structure
	Elements in a queue are termed as ***item***; the number of elements that a queue can accommodate is termed as ***MaxSize***.
	
	Three states of the queue here are:
	• Queue is Empty: front = rear = - 1
	• Queue is Full: rear = MaxSize - 1
	
	Insert / Enqueue
	template<class *KeyType*>
	void *Queue<KeyType>* :: *Enqueue*(**const** *KeyType& item*)
	// add item to the queue
	{
	if (*IsQueueFull*( )) *QueueFull*( );
	else{
	if (*IsQueueEmpty*( )) { *front* = 0;}
	Q[++*rear*] = *item*; }
	}
	
	Delete/ Dequeue
	template<class *KeyType*>
	KeyType Queue<KeyType>* :: *Dequeue*( )
	// remove front element from the queue
	{
	if (*IsQueueEmpty*( )) { *QueueEmpty*( ); **return** 0; }
	item = *Q*[*front*];
	if (*front* = = *rear*) { *rear* = *front* = -1; }
	else *front*++;
	return *item*;
	}
		1½ mark
		+
		Delete :
		1½ mark		

3.	***Linked List*** • A *linked list* is an ordered collection of finite, homogeneous data elements called *nodes* where the linear order is maintained by means of links or pointers. • *dynamic data structure* • Here adjacency between the elements is maintained by means of	Definition: 1 mark + Explanation: 1 mark +	6 marks	

	links or *pointers*. A link or pointer is the address of the subsequent element.  *Figure: Node - An element in a linked list* • An element in a linked list is specially termed as node. A node consists of two fields: DATA – to store the actual information and LINK – to point to the next node. *Memory allocation* • Nodes of a predefined type can be created using the C++ command **new.** • Example *ThreeLetterNode *f;* *f* = **new** *ThreeLetterNode;* *Memory Deallocation* • The nodes can be deleted using the **delete** command • Example **delete** *f;*	Node figure: 1 mark + ***new***: 1 ½ marks + ***delete***: 1 ½ mark		
4.	Queue can be implemented using linked list ADT.  *Figure:Linked queue* The direction of links for the queue facilitates insertion and deletion of nodes. In the case of linked queue a node can easily add at the rear. The *Queue* constructor initializes *front* to 0. The queue empty condition is *front* = = 0. In linked queue there is no need to shift queues around to make space.	Description: 2 marks + Figure: 1 mark + Add operation: 3 marks	6 marks	

	Add operation
	void *Queue* :: *Add*(**const int** *y*)
	{
	if (*front* == 0) *front* = *rear* = **new** *QueueNode* (*y*, 0) ; //empty queue
	else *rear* = *rear* → *link* = **new** *QueueNode* (*y*, 0) ;
	//attach node and update rear
	}			

5.	**Linked representation**
	In the Linked representation of Binary Trees, each node has three fields: *LeftChild, data* and *RightChild.* Following figure shows node representations.
	 
	
	Two classes are used to define a tree using linked representation as follows.
	class *Tree;* //forward declaration
	class *TreeNode*
	{
	friend class *Tree;*
	private:
	*TreeNode *LeftChild;*
	char *data;*
	*TreeNode *RightChild;*
	}
	class *Tree*
	{
	public:
	//Tree operations
	private:
	*TreeNode * root;*
	}
	Example
		Node fields: 1 mark + Node figure: 2 marks + Class definition: 2 marks + Example figure: 1 mark	6 marks	

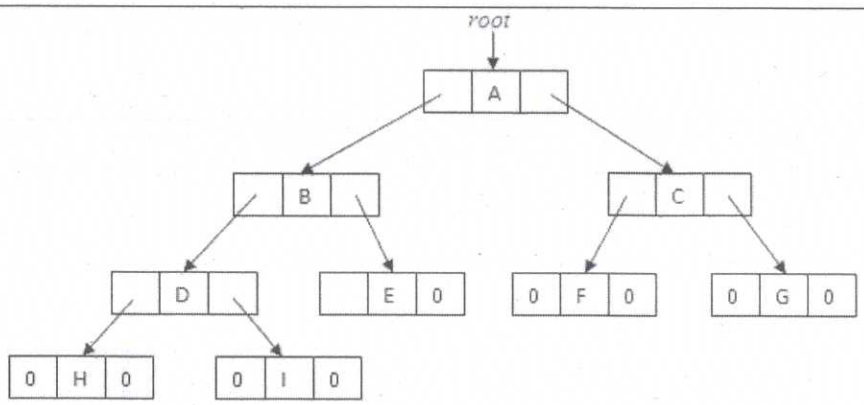


Figure: Sample Linked representation of binary a tree

(Any similar example figure)

6. **Binary Search Tree**

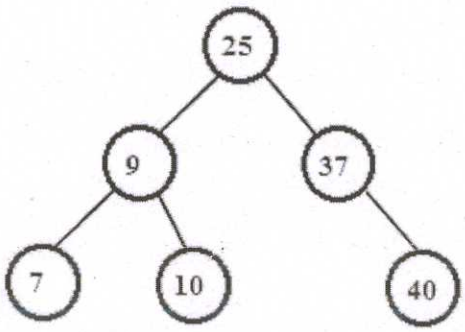
Definition: A binary search tree T is a binary tree; either it is empty or each node in the tree contains an identifier which satisfies the following properties:

1) Every element has a key and no two element have the same key (i.e., the keys are distinct).
2) The keys (if any) in the left subtree are smaller than the key in the root.
3) The keys (if any) in the right subtree are larger than the key in the root.
4) The left and right subtrees are also binary search trees.

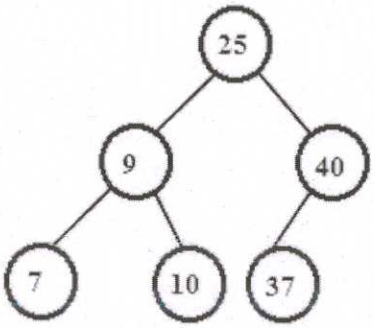
BST

We can draw the BST in two ways. Draw any one.

Option: I



Option: II

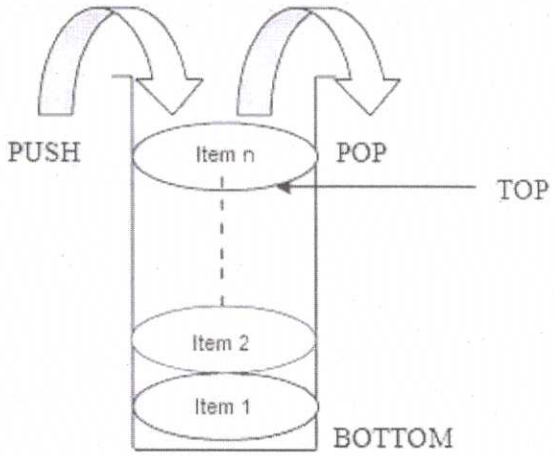


Definition: 3 marks + Figure: 3 marks

6 marks

7.	**Adjacency Matrix** Let G = (*V*,*E*) be a graph with *n* vertices, *n* >= 1. The adjacency matrix of *G* is a 2-dimensional n x n array, say *A*, with the property that • *A*(*i*,*j*) = 1 iff the edge (v*i*,v*j*) (<v*i*,v*j*> for a directed graph) is in *E*(*G*). • *A*(*i*,*j*) = 0 if there is no such edge in *G*. The adjacency matrix for an undirected graph is symmetric as the edge (v*i*,v*j*) is in *E*(*G*) iff the edge (v*j*,v*i*) is also in *E*(G). The adjacency matrix for a directed graph need not be symmetric. Adjacency matrix for the given graph is: $\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$	Definition: 2 mark + Explanation: 1 mark + Adjacency matrix: 3 marks	6 marks	
III a)	**PART C** **Stack ADT** Stack is a linear data structure which contains an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only. It uses the principle of Last In First Out (LIFO). ***Basic operations*** • PUSH & POP The insertion and deletion operations of stack are specially termed as PUSH and POP respectively and the position of the stack where these operations are performed is known as TOP of the stack. • PEEP : Used to show the topmost (data) element in a stack without deleting it.	Stack definition: 2 marks + Operations: 1 mark + Stack Full: 1 mark +	9 marks	60 marks

An element in a stack is termed as ITEM. The maximum number of elements that a stack can accommodate is termed as SIZE.



Stack Full

Stack is full when top = stacksize – 1

Stack Empty

Stack is empty when top = – 1

PUSH

```
/*  push() function inserts an item on top of stack */
template<class T>
void stack<T> :: push(T item)
{
 if (isFull())
    Stack is full;
else
    stack[++top]=item;
}
```

POP

```
/*  pop() function : deletes an item from top of stack */
template<class T>
T stack<T> :: pop()
{
  if(isEmpty())
    Stack is empty;
  else
  {
    T x;
    x = stack[top--];
    return x;
  }
```

Stack Empty:
1 mark
+
Push:
2 marks
+
Pop:
2 marks

III b)	**Data Structure** ***Definition of Data Structure***: A data structure is a method for organizing and storing data which would allow efficient data retrieval and usage. Common data structures include Array, Linked list, Hash table, Graph, Heap, Tree, Stack and Queue. Data structures are broadly divided into two as: • Linear data structure • Non-linear data structure **Linear data structure** • It is a structure that organizes its data elements one after the other. • Organizes their data elements in a linear fashion where data elements are attached one after the other. Traversed one after the other and only one element can be directly reached. • Easy to implement since computer's memory is also organized in a linear fashion. • Example: Array, Linked List, Stack, Queue **Non-linear data structure** • It is a structure constructed by attaching a data element to several other data elements in such a way that it reflects a specific relationship among them. • The elements are not organized in a sequential order. It branches to more than one node and cannot be traversed in a single run. • Example: Tree, Graph	Data structure Defn: 1 mark + Linear data structure: 2 marks + Example: ½ mark + Nonlinear data structure: 2 marks + Example: ½ mark	6 marks	
IV a)	**Infix to Postfix conversion using Stack ADT** • To fix the order of evaluation, we assign a priority to each operator. Then within any pair of parenthesis, the operators with the highest priority will be evaluated.		9 marks	

- A set of operator priorities in C++ is :

Operator	Priority
Unary minus, !	1
*, /, %	2
+, -	3
<, <=, >=, >	4
==, !=	5
&&	6
||	7

- In infix to postfix conversion, two priorities for operators are used: ***isp*** (in-stack priority) and ***icp*** (in-coming propriety). The *isp* and *icp* of all operators in above table is the priority given for that.

  isp('(') returns 8 and icp('(') returns 0. And isp('#') returns 8.

These priorities result in the following rule: *Operators are taken out of the stack as long as their in-stack priority is numerically less than or equal to the in-coming priority of the new operator.*

Algorithm

void ***postfix(expression e)***

//Output the postfix form of the infix expression e.

//A function *NextToken* is used to get the next token form *e*.

//The function uses the stack *stack*. It is assumed that the last token in *e* is '#'. '#' is used at the bottom of the stack.

```
{
  Stack <token>  stack;            //Initialize stack
  token y;
  stack . Add('#');
  for (token  x = NextToken(e); x != '#'; x = NextToken(e))
  {
    if (x is an operand)
```

Description:
3 marks
+
Algorithm:
6 marks

```
        cout<<x;
    else if (x == ')')          //unstack until '('
        for(y = *stack.Delete(y); y != '('; y = *stack.Delete(y))
            cout<<y;
    else                        //x is an operator
    {
      for(y = *stack.Delete(y); isp(y) <= icp(x); y = *stack.Delete(y))
            cout<<y;
      stack.Add(y);  //restack the last y that was unstacked.
      stack.Add(x);
    }
  } //end of expression; empty stack
  while(!stack . IsEmpty())          cout<<*stack . Delete(y);
}       //end of postfix
```

IV b)

Circular Queue

A more efficient queue representation is obtained by regarding the array *CQ*[*MaxSize*] as circular.

When *rear* = *MaxSize* – 1, the next element is entered as *CQ*[0] in case that spot is free. The queue would be empty or full if it satisfies the two states below

1. Circular queue is empty means *front* = *rear* = -1
2. Circular queue is full means *front* = ((*rear* + 1) % *MaxSize*)

. The insert & delete operations are termed as circular enqueue and dequeue.

```
template<class KeyType>
void CQueue<KeyType> :: CEnqueue(const KeyType& item)
// add item to the queue
{
    if (IsCQueueFull( )) { CQueueFull( ); return; }
    else if (IsCQueueEmpty( )) { front = 0; rear = 0; }
    else rear = (rear + 1) % MaxSize;
```

Circular queue: 1 mark + Empty & full: 1 mark + Enqueue(): 2 marks + Dequeue(): 2 marks

6 marks

	CQ[rear] = item;
	}
	template<class *KeyType>*
	void *CQueue<KeyType> :: CDequeue( )*
	// remove front element from the queue
	{
	if (*IsCQueueEmpty( )*) { *CQueueEmpty( ); **return** 0; }*
	item = CQ[front];
	if (*front = = rear*) { *rear = front* = -1; }
	else *front* = (*front* + 1) % *MaxSize;*
	return *item;*
	}			

V a)	**Insert an element into the front of a list**
	On inserting a new node into the front of list, first check whether the list is empty. If the list is empty, set the *first* and *last* points to the *newnode*. If the list is not empty, the link field of *first* is set to the *link* field of *newnode* and the *first* pointer is set to *newnode*.
	Algorithm
	template <class *Type>*
	void *List <Type> :: Attach(Type k)*
	{
	*ListNode <Type> * newnode* = **new** *ListNode <Type>(k) ;*
	if (*first* == 0) *first = last = newnode ;*
	else
	{
	newnode → link = first → link;
	first = newnode ;
	}
	}
	The insertion of a new node into the front of the linked list is shown in the figure above.
		Algorithm (at front): 3 marks
		+
		Explanation: 1 ½ marks
		+
		Algorithm (at end): 3 marks
		+
		Explanation: 1 ½ marks
			9 marks	

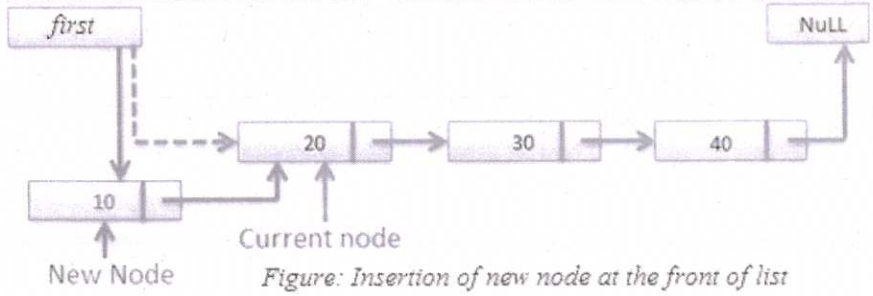


Figure: Insertion of new node at the front of list

Insert an element to the end of a list

Assume a private data member *last* points to the last node in the linked list. On inserting a new node at the end of list, first check whether the list is empty.

- If the list is empty, set the *first* and *last* points to the *newnode*.
- If the list is not empty, the link field of *last* node is set to new node and the *last* pointer is set to *newnode*.

Algorithm

```
template <class Type>
void List <Type> :: Attach( Type k)
{
        ListNode <Type> * newnode = new ListNode <Type>(k) ;
        if ( first == 0) first = last = newnode ;
        else
        {
                last → link = newnode ;
                last = newnode ;
        }
}
```

The insertion of a new node in the end of the linked list is shown in the figure below.

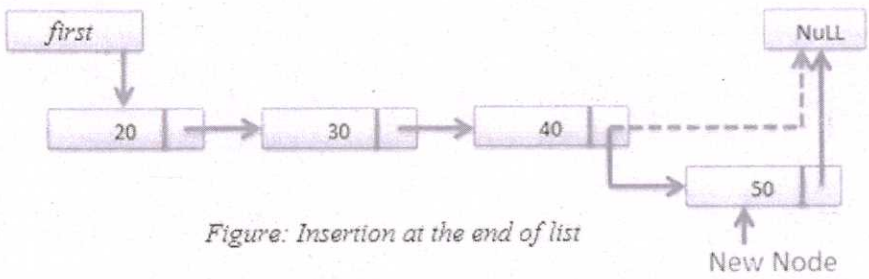


Figure: Insertion at the end of list

V b)	**Find()**
	• It returns the position of the first occurrence of an item.
	
	int List<T>:: find(T item)
	{
	int found=0;
	for(int index =0; index<size; index++)
	if(list[index]==item)
	{ found=1;
	break;
	}
	return found;
	}
	FindKth()
	• It returns the element in some position (specified as an argument).
	// findKth()function
	T List<T>::findKth (int k)
	{
	return(list[k])
	}
		Find()
		Explanation:
		1 mark
		+
		Function:
		2 mark
		+
		findKth()
		explanation:
		1 mark
		+
		Function:
		2 mark
			6 marks	

VI a)	**Linked Stack** In the linked stack, the direction of links for the stack facilitates insertion and deletion of nodes. Here we can easily add a node at the top or delete one from the top. • The *Stack* constructor ensures that *top* is initialized to 0 for each stack. The condition *top* == 0 signals that the stack is empty. • In linked stacks there is no need to shift stacks around to make space.	Explanation: 3 marks + Figure: 1 mark + Add operation: 2 marks +	9 marks	

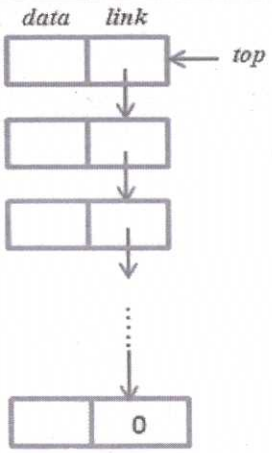


Figure: Linked Stack

Add

```
void Stack :: Add(const int y)
{
        top = new StackNode (y, top) ;

}
```

Delete

```
int *  Stack :: Delete(int&  retvalue)
{
        if (top == 0)
        {
           StackEmpty();  return 0;
        }        // return null pointer constant
        StackNode *delnode  =  top ;
        retvalue = top → data ;        //get data field of top node
        top = top → link ;             //remove top node
        delete delnode;                        // free the node
        return  &retvalue ;            // return pointer to data
}
```

Delete operation: 3 marks

VI b)

Doubly Linked List

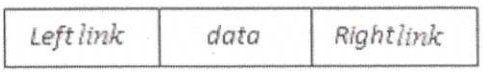
- Doubly lined lists are used to move in either direction.
- Each node has two link data members, one linking in the forward direction and one in the backward direction.
- A node in doubly linked list has at least three fields – *data, llink* (left link), *rlink* (right link).

Description: 1 mark + Figure: 1 mark

6 marks

- A doubly linked list may or may not be circular. Besides these three nodes a special node has been added called a ***head node***.

Figure: a node in doubly linked list



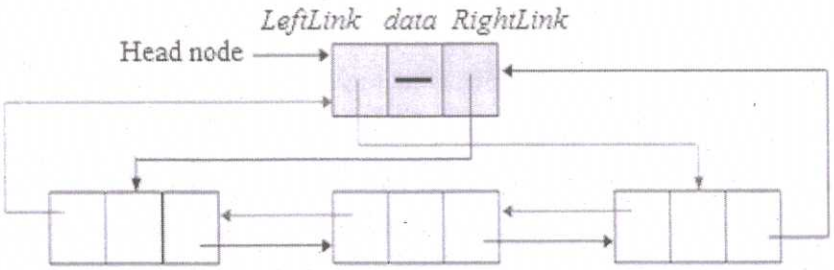


Figure: Doubly linked circular list with head node

Insertion

Insertion into a doubly linked circular list algorithm.

```
void DblList :: Insert (DblListNode p, DblListNode *x)
//insert node p to the right of node x
{
        P → llink = x; p → rlink = x → rlink;
        x → rlink → llink = p;  x → rlink = p;

}
```

Deletion

Function *DblList::Delete* deletes node *x* from the list. *x* now points to a node that is no longer part of the list.

```
void DblList :: Delete (DblListNode *x)
{
        if (x == first)
                cout<<"Deletion of head node not permitted"<<endl;
        else
        {
                x → llink → rlink = x → rlink;
                x → rlink → llink = x → llink;
                delete x;
        }

}
```

+

Insert:

2 mark

+

Delete:

2 mark

VII a)

Insertion into a Binary Search Tree

- To insert a new element, we must first verify that its key is different from those of existing elements.
- To do this, a search is carried out.
- If the search is unsuccessful, then the element is inserted at the point the search terminated.

Insert algorithm: 3 marks + Explanation: 1½ marks

+

Search algorithm: 3 marks + Explanation: 1½ marks

9 marks

Algorithm

```
int BST<T>::insert(T item)  //inserts item into BST.
                            //Returns 1 if successful
{
  treenode< T >*p = root,*q = NULL;
  while(p)
  {
     q = p;
     if (p → data = = item)
          return 0;            //item already in BST
     if(item < p → data)
          p = p → left;
     else
          p = p → right;
  }
// makes the new BST node
  p = new treenode < T >;
  p → left = p → right = NULL;
  p → data = item;

  if ( ! root)            // if it is the first BST node
     root = p;
  else if (item < q → data)
     q → left = p;
  else
     q → right = p;
  return 1;            //node successfully inserted

}
```

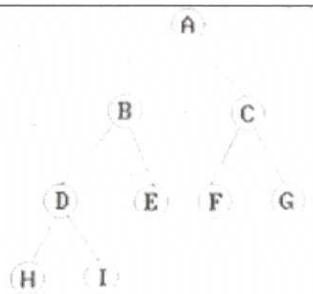
Searching a Binary Search Tree

When searching a binary search tree for an element with key x.

- We begin at the root.
  - If the root is 0, then the search tree contains no elements and

	the search is unsuccessful.
	o Otherwise, we compare x with the key in the root.
	▪ If x equals this key, then the search terminates successfully.
	▪ If x is less than the key in the root, then no element in the right subtree can have key value x, and only the left subtree is to be searched.
	▪ If x is larger than the key in the root, then only the right subtree needs to be searched.
	• The subtrees may be searched recursively.
	
	Algorithm
	template<class T>
	treenode< T > *BST< T > :: find(treenode< T > *tree, T item)
	{
	if (!tree)
	return NULL;
	if (tree → data = = item)
	return tree;
	else if(item < tree → data)
	return find(t → left, x);
	else
	return find(t → right, x);
	
	}			

VII b)	**Threaded binary tree** • It is a binary tree in which the null links are replaced by pointers – called threads – to other nodes in the tree. • If the RCHILD(P) is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in *inorder*. • A null LCHILD link at node P is replaced by a pointer to the node which immediately precedes node P in *inorder*. • Example:	Definition: 2 marks + Description: 2 marks + Example: 2 marks	6 marks	



This tree has 9 nodes and 10 null links or 0-links, which have been replaced by threads. If we traverse the tree in *inorder*, the nodes will be visited in the order *H, D, I, B, E, A, F, C, G*.

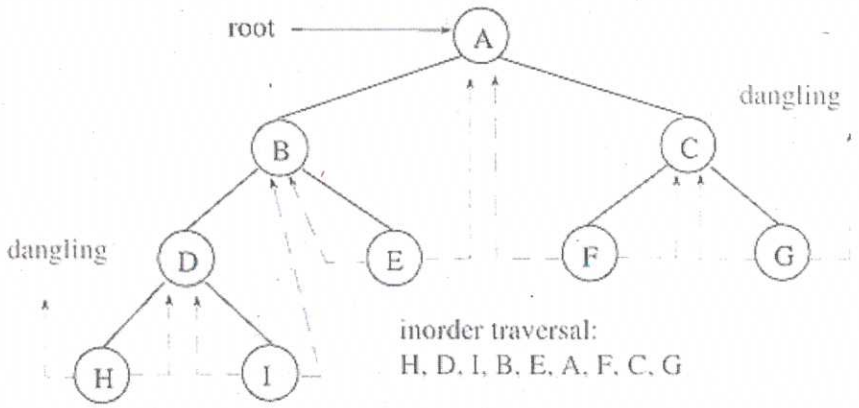
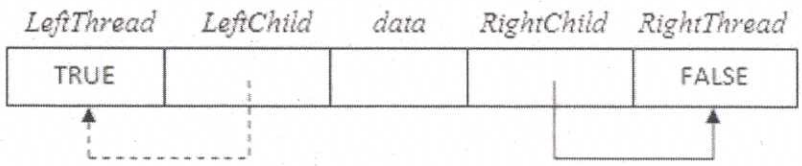


Figure: Threaded Binary Tree

In the memory representation to distinguish between threads and normal pointers, adding two Boolean fields, ***LeftThread*** and ***RightThread***, to the record.

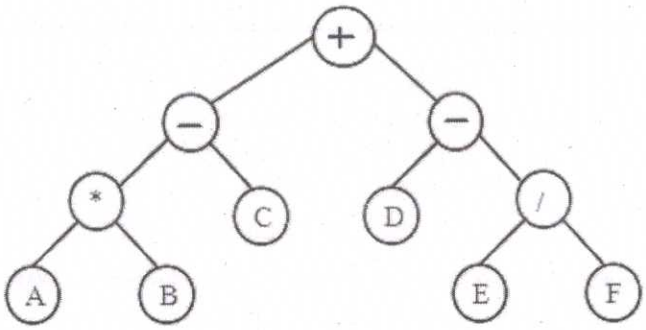
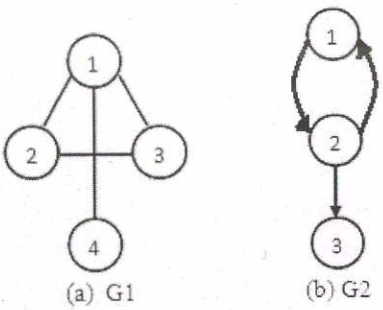
- If t → *LeftThread* == TRUE, then t → *LeftThread* contains a thread; otherwise it contains a pointer to the left child.
- If t → *RightThread* == TRUE, then t → *RightThread* contains a thread; otherwise it contains a pointer to the right child.

An empty binary tree is represented as its head node as in the figure.



VIII a)	**Algorithm**
	
	Inorder
	template<class T>
	void BST<T> :: inOrder(treenode<T> *currentnode)
	{
	if(currentnode!= NULL)
	{
	inOrder(currentnode → left);
	cout<<currentnode → data;
	inOrder(currentnode → right);
	}
	}
	
	Preorder
	template<class T>
	void BST<T> :: preOrder(treenode<T> *currentnode)
	{
	if(currentnode != NULL)
	{
	cout<<currentnode → data;
	preOrder(currentnode → left);
	preOrder(currentnode → right);
	}
	}
	
	Postorder
	template<class T>
	void BST<T> :: postOrder(treenode < T > *currentnode)
	{
	if(currentnode != NULL)
	{
	postOrder(currentnode → left);
	postOrder(currentnode → right);
	cout<<currentnode → data;
	}
	}
		Inorder:
		3 marks
		+
		Preorder:
		3 marks
		+
		Postorder:
		3 marks
			9 marks	

VIII b)	**Expression Trees** • An *expression tree* is a binary tree which stores an arithmetic expression. • The leaves of an expression tree are operands, such as constants or variable names, and all internal nodes are the operators.	Definition: 1 mark	6 marks	

	• Expression tree is always a binary tree because an arithmetic expression contains either binary operators or unary operators; hence an internal node has at most two children. • Expression tree for given expression is:  	+ Explanation: 2 marks + Figure: 3 marks		
IX a)	**Path** • A *path* from vertex *vp* to vertex *vq* in graph *G* is a sequence of vertices *vp*,*vi*1,*vi*2, ..., *vin*, *vq* such that (*vp*, *vi*1),(*vi*1,*vi*2), ...,(*vin*,*vq*) are edges in *E*(*G*). • If *G'* is directed then the path consists of <*vp*,*vi*1>,<*vi*,*vi*2>, ..., <*vin*,*vq*>, edges in *E*(*G'*). Example. **Cycle** • A *cycle* is a simple path in which the first and last vertices are the same. • Example: *Any similar graph* 1,2,3,1 is a cycle in *G*1. 1,2,1 is a cycle in *G2*.  (a) G1 (b) G2 **Degree of a vertex** • The degree of a vertex is the number of edges incident to that vertex. • In case *G* is a directed graph, we define the ***in-degree*** of a vertex *v* to be the number of edges for which *v* is the head.	3 marks for each (Description: 2 marks + Example: 1 mark)	9 marks	

	• The ***out-degree*** is defined to be the number of edges for which v is the tail. • Example: The degree of vertex 1 in $G1$ is 3.			

IX b)	**Linear Search Algorithm**
	
	LinearSearch (list: array of items)
	{
	size = list.count
	found = 0
	For i = 0 to size - 1
	if(list[i] = searchitem)
	found = 1
	position = i;
	End for
	If(found = 1)
	Display the item found at position.
	Else
	Display the item not found.
	}
	• In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.
	• It is used for unsorted and unordered small list of elements.
		Algorithm: 5 marks
		+
		Explanation: 1 mark
			6 marks	

X a)	**DFS – Depth First Search** • Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration. • In the example given, DFS algorithm traverses from S to A to D to G to E to B first, then to F and last to C. • Working: o **1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack. o **2** – If no adjacent vertex is found, pop up a vertex from the stack. o **3** – Repeat 1 and 2 until the stack is empty.	DFS description: 1 ½ marks + Algorithm / working: 3 marks +	9 marks	

BFS – Breadth First Search

- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
- In the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D.
- Working:
  - **1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
  - **2** – If no adjacent vertex is found, remove the first vertex from the queue.
  - **3** – Repeat 1 and 2 until the queue is empty.

BFS description: 1 ½ marks + Algorithm / working: 3 marks

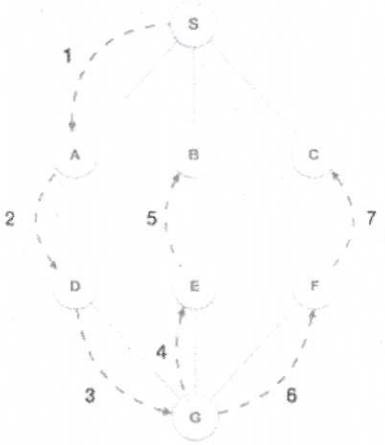


Figure: Depth – First Search

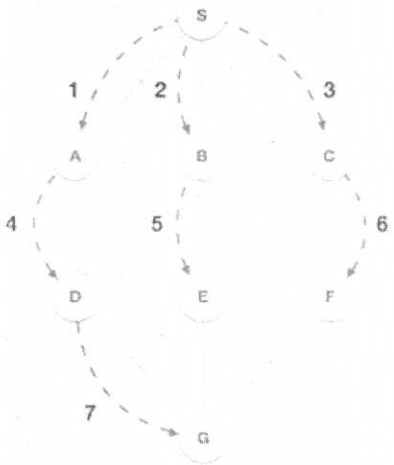


Figure: Breadth – First Search

Algorithm

```
template <class T>
void Graph<T> :: dfs(int s)
{
  for( i = 0; i < nodes; i++)
      visited[i] = 0;                    //mark all nodes as unvisited
  visit(visited, s);
}

template<class T>
void Graph<T> :: bfs(int s)
{
  for(i = 0; i < n; i++)
     visited[i]=0;
```

```
Queue<int> q;                    //q is a Queue.
q.insert(s);                     // s is inserted to q
while(!q.isEmpty())
{
   v=q.delete();
   for(all vertices w adjacent to v)
   if(! visited[w])
   {
      q.insert(w);
      visited[w]=1;
   }
}     }
```

X b)

Algorithm

```
/* x[] is the array, p is starting index, that is 0, and r is the last index,
that is n-1 of array. */

void quicksort(int x[], int p, int r)
{
  if(p < r)
  {
    int q;
    q = partition(x, p, r);
    quicksort(x, p, q);
    quicksort(x, q+1, r);
  }
}
int partition(int x[], int p, int r)
{
  int i, j, pivot, temp;
  pivot = x[p];
  i = p;   j = r;
  while(1)
  {
    while(x[i] < pivot  && x[i] != pivot)
        i++;
    while(x[j] > pivot  && x[j] != pivot)
        j--;
    if(i < j)
        interchange (x[i], x[j]);
    else
    {        return j;     }
} }
```

Algorithm: 6 marks

6 marks