SCHEME OF VALUATION

Scoring Indicators

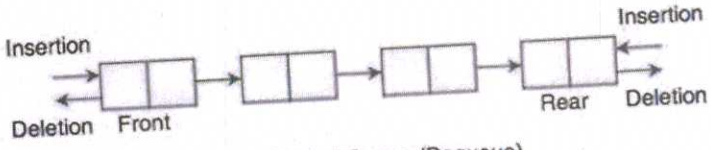
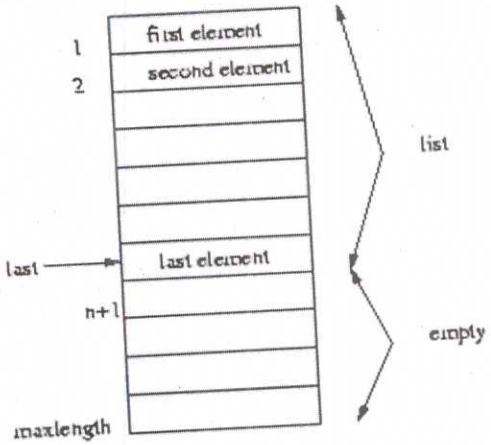
Revision: 2015 **Course Code:4133**

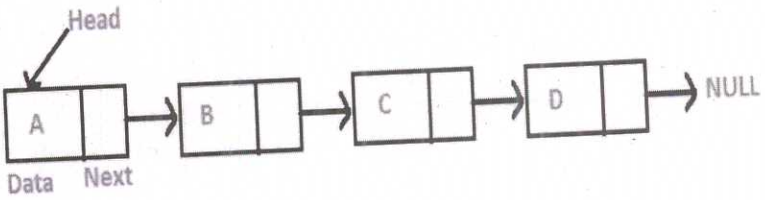
Course Title: DATA STRUCTURES

Qst No	Scoring Indicators	Split up score	Sub Total	Total
	Part -A			
I 1)	Insertion, Deletion, Traversal. Search	0.5*4	2	
2)	Recursion,Infix to postfix Conversion	1+1	2	

3)	class LinkedList
	{
	class Node
	{
	friend class LinkedList;
	T data;
	node *next;
	};
	};
		2	2	10

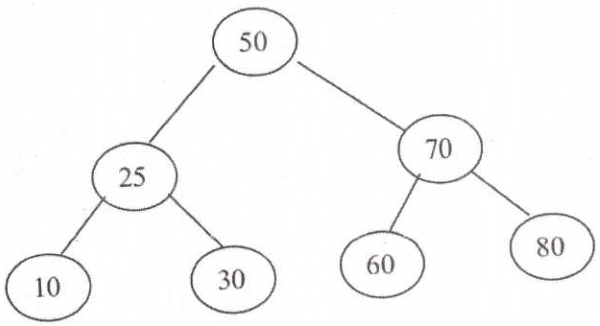
4)	For every node X, All the keys in its left subtree are smaller than the key value in X All the keys in its right subtree are larger than the key value in X	2	2	
5)	An undirected graph is one in which the pair of vertices in an edge is unordered, $(v0, v1) = (v1, v0)$ A directed graph is one in which each edge is a directed pair of vertices, $<v0, v1> != <v1, v0>$	2	2	
	Part –B			
II 1)	a) (A+B)-(C-D)*E/F Postfix-AB+CD-E*F/- b)A-(B+C)/D*E Postfix-ABC+D/E*-	3+3	6	
2)	Dequeue is double ended queue in which insertions and deletion are done at both front end and rear end. Dequeue provides all the capabilities of stacks and queues in a single data structure	Exp-3		

	~~queues in a single data structure~~  Fig. Double Ended Queue (Dequeue)	~~Exp-3~~ Fig-3	6	
3)	A List is a set of data arranged in some sort of order. List is dynamic array which means it uses internally array and extend it when we need it. LIST ADT consist of sequence of zero or more elements $A_1, A_2, A_3, \ldots A_N$ N: length of the list A_1: first element A_N: last element A_i: position i If N=0, then empty list  List ADT operations are printList: print the list makeEmpty: create an empty list find(item): locate the position of item in the list insert(index,item): insert item at position index of the list remove(item): delete the item from the list findKth(index): retrieve the item at the position index		6	6
4)	A linked list is a series of connected nodes. Each node contains at least one data and pointer to the next node in the list. Head: pointer to the first node and the last node points to NULL			

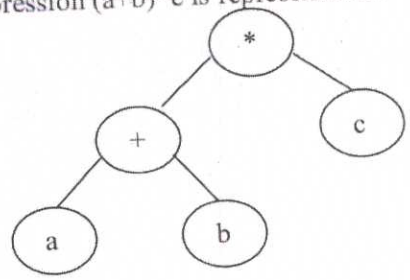


		3+3	6

```
template <class T>
void LinkedList<T> :: minsert(T item, int pos)
{
        if (pos==1)
                binsert(item);
        else if (pos==listSize()+1)
                einsert(item);
        else {
                        Node *temp = head;
                        for (int k = 1; k < pos-1; k++ )
                                temp = temp->next;
                        Node *n=new Node;
                        n->data=item;
                        n->next=temp->next;
                        temp->next=n;
                        size++;
                }
}
```

5)	Binary Search tree 	6	6
6)	Expression tree is a special kind of binary tree used to represent expression. The leaves of the expression tree are operands and the other nodes contain operators.	Exp-3 Fig-3	6

The expression (a+b)*c is represented as



7) Adjacency Matrix is 2-Dimensional Array which has the size VxV, where V is the number of vertices in the graph.
If adjMaxtrix[i][j] = 1 then there is edge between Vertex i and Vertex j, else 0.

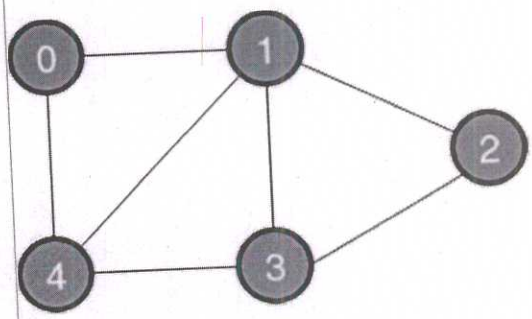
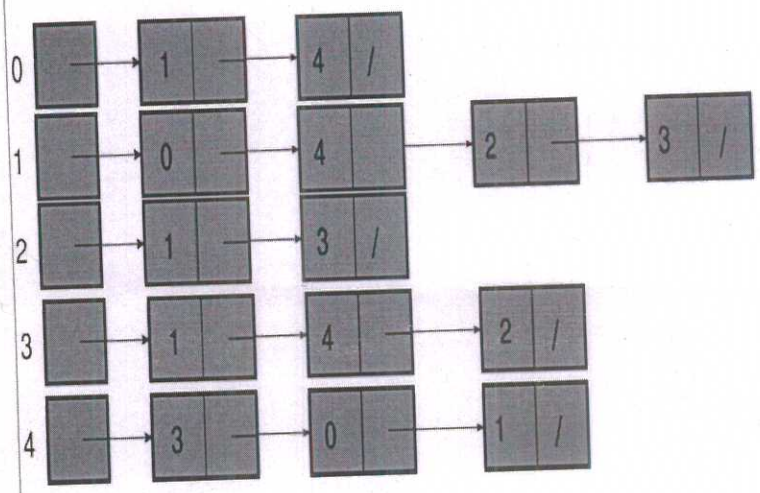


Fig-3 Exp-3 **6**

Adjacency Matrix

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency List is an array of Linked List, where array size is same as number of Vertices in the graph. Every Vertex has a Linked List. Each Node in this Linked list represents the reference to the other vertices which share an edge with the current vertex.



Part –C

III a)

```
void postfix(char* e)
{
        Stack<char>  stack;                  // declares a character stack.
        char y;
        stack.push('#');
        for ( char  x=nextToken(e) ; x!='#'; x=nextToken(e) )
        {
                if(x is an operand)
                        cout<<x;
                else if(x==')')                  //Unstack until'('
                      for ( y=stack.pop() ; y!='(' ; y=stack.pop() )
                                cout<<y;
                else
                {
                        /* x is an operator  */
                        for ( y=stack.pop() ; isp(y)<=icp(x) ; y=stack.pop() )
                                cout<<y;
                        stack.push(x);
                }
        }
        while ( !stack.isEmpty() )
                cout<<stack.pop(y);
}
```

8	**8**	**15**

b) A stack is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle. Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed. Inserting an item is known as "pushing" onto the stack. "Popping" off the stack is synonymous with removing an item.

A stack is an abstract data type (ADT) that provides the following methods:

push(o): Inserts object o onto top of stack
Input: Object; Output: none

pop(): Removes the top object of stack and returns it; if stack is empty an error occurs.
Input: none; Output: Object

size(): Returns the number of objects in stack
Input: none; Output: integer

isEmpty(): Return a boolean indicating if stack is empty.
Input: none; Output: boolean

top(): return the top object of the stack, without removing it; if the stack is empty an error occurs.
Input: none; Output: Object

```
template<class T>
class stack
{
        private:
                int stacksize, top;
                T *astack;
        public:
                stack (int size); //constructor
                int isFull();//returns 1 if full, 0 if not full
                int isEmpty();//returns 1 if Empty, 0 is not empty
                void push(T item);//pushes item to stack
                T pop();   //pops one item from stack

};
template<class T>
stack::stack(int size)
{
        stacksize=size;
        astack=new T[stacksize];
        top=-1;
}
template<class T>
int stack<T>::isFull()
{
        if (top==stacksize-1)
                return 1;
        else
                return 0;
}
```

7 7

```
template<class T>
int stack<T>::isEmpty()
{
        if (top==-1)
                return 1;
        else
                return 0;
}
template<class T>
void stack<T>::push(T item)
{
        if (isFull())
                Stack is full;
        else
                stack[++top]=item;
}
template<class T>
T stack<T>::pop()
{
        if(isEmpty())
                Stack is empty;
        else
        {
                T x;
                x=stack[top--];
                return x;
        }
}
```

IV a) A Queue is an ordered list in which all insertion takes place at one end and all deletions take place at the opposite end. Since the first element removed is the first element inserted, queues are also known as First In First Out (FIFO) lists. The end at which insertions are taken place is called '*rear*' and the end at which deletions take place is called '*front*'.

In circular queue the rear end is connected to the front end to make it circular.

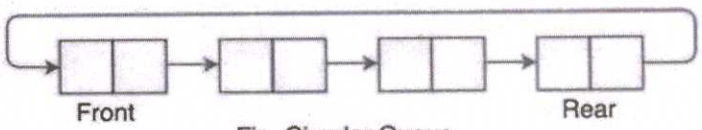


Fig. Circular Queue

10 **10** **15**

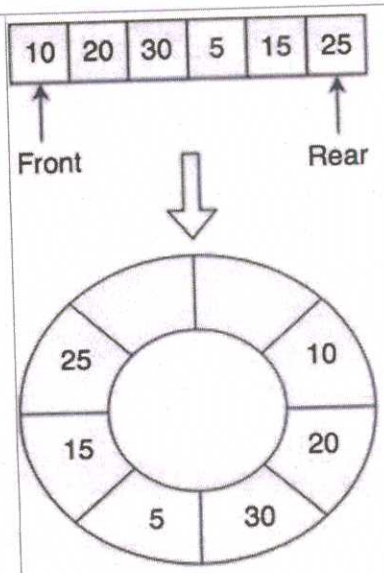


Fig. Circular Queue

```
template<class T>
class Queue
{
        private:
                int front,rear;
                T *queue;
                int qsize;
        public:
                Queue(int size); //Constructor.
                int isFull(); //return 1 for Full,0 for not Full.
                int isEmpty(); //return 1 for Empty, 0 for not Empty
                void Insert(T item);  //Inserts an item at the rear of queue.
                T Delete(); //Deletes an item at the front of queue.

};
//constructor
template<class T>
Queue<T>:: Queue(int size)
{
        qsize=size;
        queue=new T[qsize];
        front=rear= -1;
}
// isFull() function
template<class T>
int Queue<T>::isFull()
{
        if((rear+1)%qsize==front)
```

```
                return 1;
        else
                return 0;
}

//isEmpty()function
template<class T>
int Queue<T>::isEmpty();
{
        if(front==rear || front == -1)
                return 1;
        else
                return 0;
}

//Insert () function
template<class T>
void Queue<T>::Insert(T item)
{
        if(isFull())
                Queue is Full;
        else if (front = -1)
        {
                front = 0;
                rear = 0;
        }
        else
                rear=(rear+1)%qsize;

        queue[rear]=item;
}
//Delete () function
template<class T>
T Queue<T>::Delete()
{
        If(isEmpty())
        {
                Queue is Empty;
                return NULL;
        }

        T x=queue[front];
        front=(front+1)%qsize;
        return x;
}
```

b)	Time Complexity describes the amount of time taken by an algorithm to its completion based on the input. Space Complexity describes the amount of memory needed by an algorithm to its completion based on the input.	5	5	

V a)	class LinkedStack
	{
	private:
	class Node
	{ friend class LinkedStack;
	private:
	T data;
	Node *next;
	} *top;
	int size;
	public:
	LinkedStack(); //creates an empty list
	int listSize(); // return number of nodes
	int isEmpty(); // true if list is empty, false otherwise
	void push(T item); // add T to front of list
	void pop(); // remove front node
	};
	template <class T>
	LinkedStack<T> :: LinkedStack( )
	{
	top=NULL;
	size=0;
	}
	template <class T>
	int LinkedStack<T> :: listSize( )
	{
	return size;
	}
	template <class T>
	int LinkedStack<T> :: isEmpty( )
	{
	if (top==NULL) return 1;
	else return 0;
	}
	template <class T>
	void LinkedStack<T> :: push(T item)
	{
	Node *n = new Node;
	n->data=item;
	n->next=top;
	top=n;
		10	10	15

```
        size++;
}
template <class T>
void LinkedStack<T> :: pop()
{
        if(isEmpty())
        {
                cout<<"\nStack Empty";
        }
        else
        {
        Node *old=top;
        T item=top->data;
        top=top->next;
        size--;
        delete old;
        cout<<"Deleted item:"<<item;
        }
}
```

b)	Static memory is allocated on the stack at compile-time. This means it is fixed in size for the duration of the program. Dynamic memory is allocated on the heap at run-time. The main advantage of using dynamic memory allocation is preventing the wastage of memory. This is because when we use static memory allocation, a lot of memory is wasted because all the memory allocated cannot be utilised.	**5**	**5**	
VI a)		**10**	**10**	**15**

```
class LinkedQueue
{
        private:
                class Node
                {       friend class LinkedQueue;
                        private:
                                T data;
                                Node *next;
                        } *front, *rear;
                int size;

                public:
                        LinkedQueue(); //creates an empty list
                        int listSize();  // return number of nodes
                        int isEmpty(); // true if list is empty
                        void delet(); // remove front node
                        void insert(T item); // add T to end of list
};
```

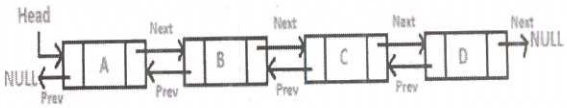
```
template <class T>
LinkedQueue<T> :: LinkedQueue( )
{
        front=NULL;
        rear=NULL;
        size=0;
}
template <class T>
int LinkedQueue<T> :: listSize( )
{
        return size;
}

template <class T>
int LinkedQueue<T> :: isEmpty( )
{
        if (front==NULL) return 1;
        else return 0;
}
template <class T>
void LinkedQueue<T> :: delet()
{
         if(isEmpty())
         {
                 cout<<"\nQueue Empty";
         }
         else
         {
         Node *old=front;
         T item=front->data;
         front=front->next;
         if (front== NULL)
                 rear= NULL;
         size--;
         delete old;
         cout<<"\nDeleted item:"<<item;
         }
}

template <class T>
void LinkedQueue<T> ::insert(T item)
{
         Node *n = new Node;
         n->data=item;
         n->next=NULL;
         if (rear == NULL)
                 rear = front=n;
```

```
        else
                {
                rear->next=n;
                rear=n;
                }
        size++;
}
```

b) **Doubly Linked List (DLL)** contains two pointers, previous pointer, next pointer and data.



Def-2 Fig-3 | **5**

VII a) Tree traversal means visiting each node in the tree exactly once. Tree traversal are of three types

1)Inorder(LDR)

i)Traverse left subtree

ii)Visit the root

iii)Traverse right subtree

2)Preorder(DLR)

i)Visit the root

ii)Traverse left subtree

iii)Traverse right subtree

3)Postorder(LRD)

i)Traverse left subtree

ii)Traverse right subtree

iii)Visit the root

15 | **15** | **15**

Inorder

```
template<class T>
```

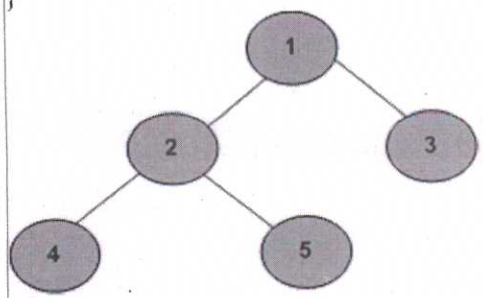
```
void BST<T>::inOrder(treenode<T> *currentnode)
        {
                if(currentnode!=NULL)
                {
                        inOrder(currentnode->left);
                        cout<<currentnode->data;
                        inOrder(currentnode->right);
                }
        }
```

Preorder

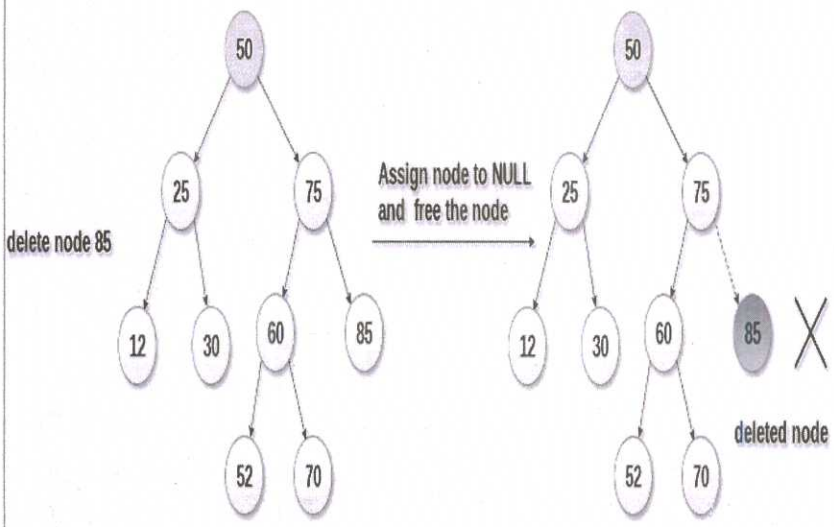
```
template<class T>
void BST<T>::preOrder(treenode<T> *currentnode)
        {
                if(currentnode!=NULL)
                {
                        cout<<currentnode->data;
                        preOrder(currentnode->left);
                        preOrder(currentnode->right);
                }
        }
```

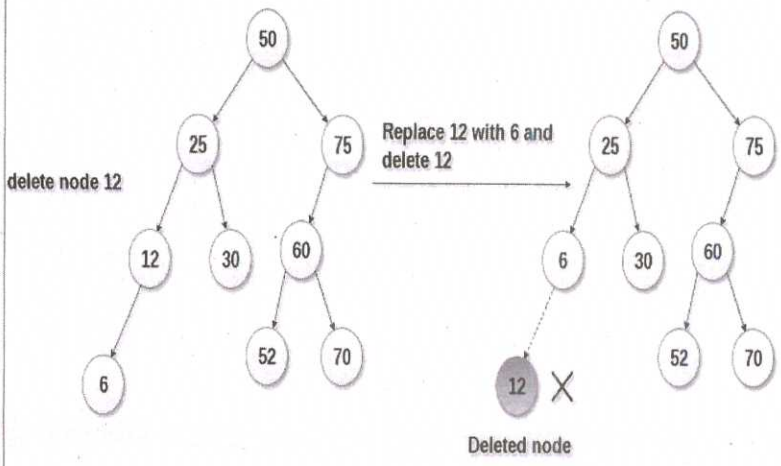
Postorder

```
template<class T>
void BST<T>::postOrder(treenode<T> *currentnode)
{
        if(currentnode!=NULL)
        {
                postOrder(currentnode->left);
                postOrder(currentnode->right);
                cout<<currentnode->data;
        }
}
```



(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

VIII a)	Delete function is used to delete the specified node from a binary search tree. if we delete a node from a binary search tree the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree. **1) The node to be deleted is a leaf node** Replace the leaf node with the NULL and free the allocated space. In the following example, if we are deleting the node 85, it will be replaced with NULL and allocated space will be freed.  **2) The node to be deleted has only one child.** In this case, replace the node with its child and then delete the child node. Replace the child node with the NULL and free the allocated space. In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 will be deleted.	**10**	**10**	**15**



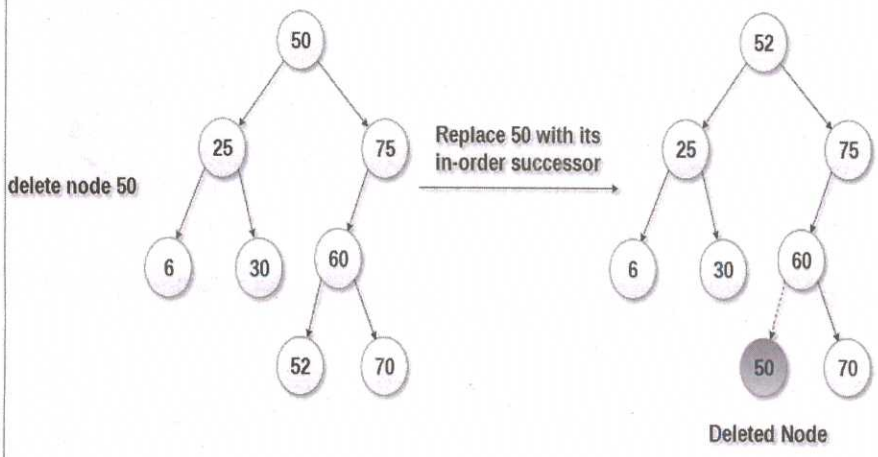
3) The node to be deleted has two children.

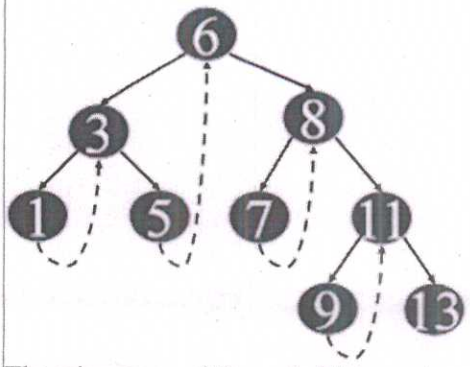
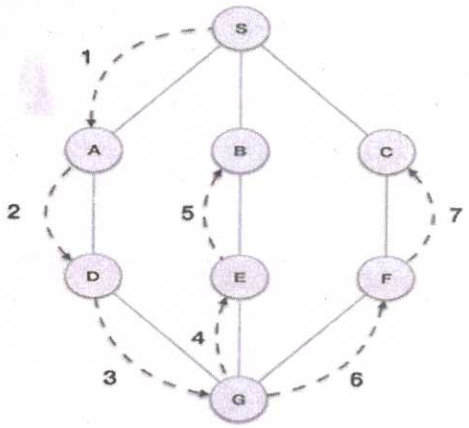
The node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value to be deleted is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

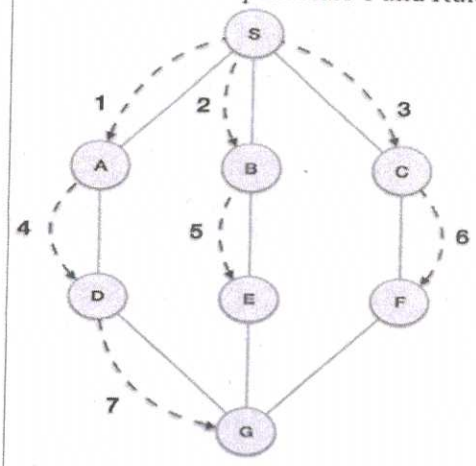
In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

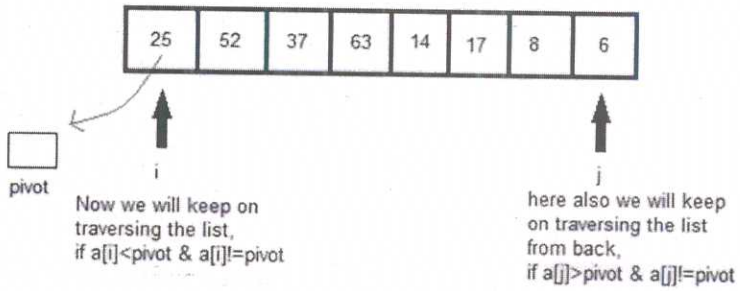
Then replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



b)	A binary tree is threaded by making all right child null pointers point to the inorder successor of the node and all left child null pointers point to the inorder predecessor of the node.  The advantage of **threaded binary trees** is to make inorder traversal faster and do it without stack and without recursion	5	5	
IX a)	DFS (Depth first Search) traverses a graph in a depth wise motion and uses a stack to remember the next vertex to start a search when a dead end occurs in any iteration. Algorithm ▯ Rule 1 – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack. ▯ Rule 2 – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.) ▯ Rule 3 – Repeat Rule 1 and Rule 2 until stack is empty.  DFS TRAVERSAL-S-A-D-G-E-B-F-C	6+6	12	15

	BFS (Breadth first search) traverses a graph in a breadth wise motion and uses a queue to remember the next vertex to start a search when a dead end occurs in any iteration. ▪ Rule 1 – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue. ▪ Rule 2 – If no adjacent vertex found, remove the first vertex from queue. ▪ Rule 3 – Repeat Rule 1 and Rule 2 until queue is empty.  BFS TRAVERSAL-S-A-B-C-D-E-F-G			
b)	Path - A path from vertex *vp* to vertex *vq* in a graph *G*, is a sequence of vertices, *vp*, *vi*1, *vi*2, ..., *vi*n, vq, such that (*vp*, *vi*1), (*vi*1, *vi*2), ..., (*vi*n, v*q*) are edges in thegraph. Cycle - A cycle is a simple path in which the first and the last vertices are the same. Degree - The degree of a vertex is the number of edges incident to that vertex	1+1+1	3	
X a)	Quick Sort sorts any list very quickly. Quick sort is very fast and requires very less additional space. It is based on the rule of Divide and Conquer (also called *partition-exchange sort*). This algorithm divides the list x[] into three parts : ▪ Elements less than the *pivot* element ▪ *Pivot* element ▪ Elements greater than the *pivot* element Notice that if these two conditions hold for a particular *pivot* element placed at position *j*, then the *pivot* element is the *jth* smallest element in the list so	9	9	15

that the *pivot* remains in position j when the array is completely sorted. If the foregoing process is repeated with the sub-array x[0] through x[j-1] and x[j+1] though x[n-1] and any sub-arrays created by the process in successive iterations, the final result is the sorted array.



DIVIDE AND CONQUER - QUICK SORT

For example, in the list of elements 25, 52, 37, 63, 14, 17, 8, 6, we take the first element, i,e: 25 as the pivot. So after the first pass, the list will be changed like this.

6 8 17 14 25 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

The worst case time complexity of quick sort is $O(n^2)$, whereas the best case and average time complexity is O(n log n).

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 (a[i],a[j]);
                else
                {
                        return j;
                }
        }
}
```

b)	Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.	6	6	