SCHEME OF VALUATION

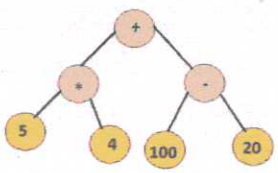
(Scoring Indicators)

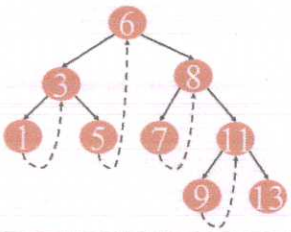
Revision: 15
Course Code: TED (15)-4133
Course Title: DATA STRUCTURES

Qst. No.	Scoring Indicator	Split up score	Sub Total	Total
I	**PART A**			
1	A B C D - * E / F G - * +	2	2	10
2	1. EnqueueFront()—adds elements at the front end of the queue 2. EnqueueRear()—adds elements at the rear end of the queue 3. DequeueFront()—deletes elements from the front end of the queue 4. DequeueRear()—deletes elements from the rear end of the queue	4*0.5	2	
3	1. Where all nodes are connected to form a circle. 2. There is no NULL at the end.	2*1	2	
4	1. Representation of expressions arranged in a tree-like data structure. 2. Leaves as operands of the expression and nodes contain the operators	2*1	2	
5	1. Adjacency Matrix 2. Adjacency List	2*1	2	
II	**PART B**			
1	1. Theta Notations : The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior. 2. Big O Notations : The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. 3. Omega Notations : Ω notation provides an asymptotic lower bound.	3*2	6	

2	1. Let E denote the postfix expression
	2. Let Stack denote the stack data structure to be used & let Top = −1
	3. while(1) do
	begin
	X = get_next_token(E) // Token is an operator, operand, or delimiter
	if(X = #) {end of expression}
	then return
	if(X is an operand)
	then push(X) onto Stack
	else {X is operator}
	begin
	OP1 = pop() from Stack
	OP2 = pop() from Stack
	Tmp = evaluate(OP1, X, OP2)
	push(Tmp) on Stack
	end {If X is operator then pop the correct number of operands from stack for operator X. Perform the operation and push the result, if any, onto the stack}
	end
	4. stop.
		6	6	

3	Hint:
	1. Insert tail node:
	tail->next=temp;
	tail=temp;
	
	2. Delete tail node:
	while(cur!=tail)
	{
	prev=cur;
	cur=cur->next;
	}
	if(cur==tail)
	{
	cout<<"Data deleted is "<<cur->data<<endl;
	prev->next=NULL;
	tail=prev;
	}
	else
	cout<<"Unable to Detete"<<endl;
		2*3	6	

4	I. **Inorder** 1. Traverse the left subtree 2. Visit the root 3. Traverse the right subtree II. **Preorder** 1. Visit the root. 2. Traverse the left subtree 3. Traverse right subtree **III. Postorder** 1. Traverse the left subtree 2. Traverse right subtree 3. Visit the root	3*2	6	
5	**Expression tree:** It is representation of expressions arranged in a tree-like data structure ie .Leaves as operands of the expression and nodes contain the operators  	2*3	6	30

	Threaded binary tree: A threaded binary tree is a binary tree in which all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.  			
6	1. Start with the middle element: ○ If the **target** value is equal to the middle element of the array, then return the index of the middle element. ○ If not, then compare the middle element with the target value, ▪ If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1. ▪ If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1. 2. When a match is found, return the index of the element matched. 3. If no match is found, then return	6	6	

7	Create a |V| x |V| matrix // It represents the distance between every pair of vertices as given
	For each cell (i,j) in M do
	if (i = = j)
	M[i][j] = 0 // For all diagonal elements, value = 0
	if (i , j) is an edge in E
	M[i][j] = weight(i,j) // If there exists a direct edge between the vertices, value = weight of edge
	else
	M[i][j] = infinity // If there is no direct edge between the vertices, value = ∞
	for k from 1 to |V|
	for i from 1 to |V|
		6	6	

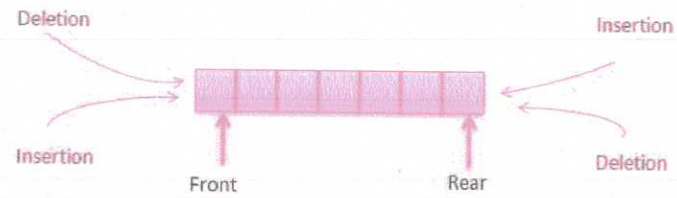
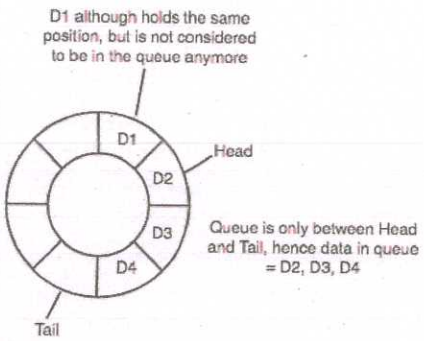
	for j from 1 to |V| if M[i][j] > M[i][k] + M[k][j] M[i][j] = M[i][k] + M[k][j]			
	PART C			
III				

a	1. Scan expression E from left to right, character by character, till character is '#'
	ch = get_next_token(E)
	2. while(ch != '#')
	if(ch = ')')
	then ch = pop()
	while(ch !='(')
	Display ch
	ch = pop()
	end while
	if(ch = operand)
	display the same
	if(ch = operator) then
	if(ICP > ISP) then push(ch)
	else
	while(ICP <= ISP)
	pop the operator and display it
	end while
	ch = get_next_token(E)
	end while
	3. if(ch = #)
	then
	while(!emptystack())
	pop and display
	4. Stop
		9	9	

b	Step 1 - Include all the header files which are used in the program and declare all the user defined functions. Step 2 - Define a 'Node' structure with two members data and next. Step 3 - Define a Node pointer 'top' and set it to NULL. Step 4 - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method. **PUSH(value):** 1. Create a newnode with given value 2. If(top==NULL) then set newnode->next=NULL. 3. Else set newnode->next=top 4. Set top=newnode. **POP(value):** 1. If (top==NULL) then display stack is empty, deletion not possible. 2. Else set node *temp=top Set top=top->next, Free(temp)	6	6	

IV				

a	Queue ADT: collection of items in which items may be inserted at rear end and deleted at front end.(FIFO)
	Primitive operations:
	1.Insert(Que , x)
	If(notQuefull)
	Que[++rear]==x
	2. Delete(Que)
	If(notEmptyQue)
	X=Que[front]
	Front--
		9	9	15

b	**Dequeue:** In Double-ended Queue elements can be added or deleted at either front end or rear end.  **Circular queue:** Linear data structure in which operations are performed based on FIFO principle and last position is connected back to first position to make a circle.  	3+3	6	
V				

a	#include<iostream>
	using namespace std;
	class node
	{
	public:
	int data;
	node *next;
	};
	class ll:public node
	{
	node *head,*tail;
		5+5+5	15	15

```
public:
ll()
{
head=NULL;
tail=NULL;
}
        void create();
        void insert();
        void disp();
        void del();
};
        void ll::create()
        {
        node *temp;
        temp=new node;
        int n;
        cout<<"Enter the data";
        cin>>n;
        temp->data=n;
        temp->next=NULL;
                if(head==NULL)
                {
                head=temp;
                tail=head;
                }
                else
                {
                tail->next=temp;
                tail=temp;
                }
                }
void ll::insert()
{
node *prev,*cur;
prev=NULL;
cur=head;
int count=1,pos,ch,n;
node *temp;
temp=new node;
cout<<"Enter the data"<<endl;
cin>>n;
temp->data=n;
temp->next=NULL;
cout<<"Enter 1.Insert as First node 2. Insert as Tail node 3. in between
node"<<endl;
cin>>ch;
switch(ch)
{
case 1:
        temp->next=head;
        head=temp;
        break;
case 2:
        tail->next=temp;
        tail=temp;
```

```
            break;
case 3:
            cout<<"Enter the position"<<endl;
            cin>>pos;
            while(count!=pos)
            {
            prev=cur;
            cur=cur->next;
            count++;
            }
            if(count==pos)
            {
            temp->next=cur;
            prev->next=temp;
            }
            else
            cout<<"Unable to Insert";
}
}

void ll::del()
{
node *prev,*cur;
prev=NULL;
cur=head;
int count=1,pos,ch,n;
cout<<"Enter 1.Delete First node 2. Delete Tail node 3. Delete in between
node"<<endl;
cin>>ch;
switch(ch)
{
case 1:
            if(head!=NULL)
            {
            cout<<"Data deleted is "<<head->data<<endl;
            head=head->next;
            }
            else
            cout<<"Unable to Detete"<<endl;
            break;
case 2:
            while(cur!=tail)
            {
            prev=cur;
            cur=cur->next;
            }
            if(cur==tail)
            {
            cout<<"Data deleted is "<<cur->data<<endl;
            prev->next=NULL;
            tail=prev;
            }
            else
            cout<<"Unable to Detete"<<endl;
            break;
```

```
case 3:
        cout<<"Enter the position"<<endl;
        cin>>pos;
        while(count!=pos)
        {
        prev=cur;
        cur=cur->next;
        count++;
        }
        if(count==pos)
        {
        cout<<"Data deleted is "<<cur->data<<endl;
        prev->next=cur->next;
        }
        else
        cout<<"Unable to Detete"<<endl;
}
}


void ll::disp()
{
node *temp;
temp=head;
if(temp==NULL)
cout<<"Empty List"<<endl;
while(temp!=NULL)
{
cout<<temp->data<<" ";
temp=temp->next;
}

}
int main()
{
ll L;
.....
return 0;
}
```

VI

a)

```
#include<iostream>
using namespace std;
class node
{
public:
int data;
node *next;
};
        class ll:public node
        {
        node *head,*tail;
        public:
        ll()
```

5+5+5 | 15 | 15

```
        {
        head=NULL;
        tail=NULL;
        }
                void push();
                void disp();
                void pop();
        };
                void ll::push()
                {
                node *temp;
                temp=new node;
                int n;
                cout<<"Enter the data ";
                cin>>n;
                temp->data=n;
                temp->next=NULL;
                        if(head==NULL)
                        {
                        head=temp;
                        tail=head;
                        }
                        else
                        {
                        temp->next=head;
                        head=temp;
                        }
                        }

void ll::pop()
{

        if(head!=NULL)
        {
        cout<<"Data deleted is "<<head->data<<endl;
        head=head->next;
        }
        else
        cout<<"STACK EMPTY"<<endl;
}

int main()
{
ll L;
int ch,t=1;
while(t==1)
{
cout<<"Enter 1. Push 2. Pop 3.Exit"<<endl;
cin>>ch;
switch(ch)
{
case 1: L.push();
        break;
case 2: L.pop();
        break;
```

	case 3: t=0;
	}
	}
	return 0;
	}			

VII	#include<iostream>
	using namespace std;
	class node
	{
	public:
	int data;
	node *left,*right;
	};
	class bst:public node
	{
	public:
	node *root;
	bst()
	{
	root=NULL;
	}
	void create();
	void inorder(node*);
	void preorder(node*);
	void postorder(node*);
	};
	void bst::create()
	{
	node *temp;
	temp=new node;
	cout<<"Enter the Data"<<endl;
	cin>>temp->data;
	temp->left=NULL;
	temp->right=NULL;
	if(root==NULL)
	root=temp;
	else
	{
	node *p;
	p=root;
	while(1)
	{
	if(temp->data<p->data)
	{
	if(p->left==NULL)
	{
	p->left=temp;
	break;
	}
	else if(p->left!=NULL)
	p=p->left;
	}
	else if(temp->data>p->data)
	{
		5+5+5	15	15

```
if(p->right==NULL)
{
p->right=temp;
break;
}
else if(p->right!=NULL)
p=p->right;
}
}}}

void bst::inorder(node *m)
{
if(m!=NULL)
{
inorder(m->left);
cout<<m->data<<" ";
inorder(m->right);
}
}

void bst::preorder(node *m)
{
if(m!=NULL)
{
cout<<m->data<<" ";
inorder(m->left);
inorder(m->right);
}
}

void bst::postorder(node *m)
{
if(m!=NULL)
{
inorder(m->left);
inorder(m->right);
cout<<m->data<<" ";
}
}

int main()
{
bst B;
int y=1,ch;
while(y==1)
{
cout<<"Enter the Choice"<<endl;
cout<<"Enter 1.Create 2.Inorder 3.Preorder 4.Postorder 5.Exit"<<endl;
cin>>ch;
switch(ch)
{
case 1: B.create();
        break;
case 2:  B.inorder(B.root);
        break;
```

	case 3: B.preorder(B.root);
	break;
	case 4: B.postorder(B.root);
	break;
	case 5: y=0;
	break;
	default: cout<<"Invalid Option"<<endl;
	}
	}
	return 0;
	}			

VIII				
a	**Binary tree** is a tree data structure where each node has at most two children called left child and right child. **Strictly binary tree**- Every node has exactly two children or none. **Complete binary tree**- Every internal nodes have exactly two children and all leaf nodes are at same level.	2+2+2	6	15
b	Child: Node which is successor of any node. Degree: Total number of children of a node Depth: Total number of edges from root node to a particular node Edge: Connecting link between any two nodes Height: Total number of edges from leaf node to a particular node in the largest path. Leaf: Node which doesn't have a child. Level: In a tree each step from top to bottom is called level Path: Sequence of nodes and edges from one node to another node Siblings: Nodes which belongs to same parent.	9*1	9	
IX	**Graph**: Collection of vertices and edges, in which vertices are connected with edges. Represented as G=(V,E). Graph Traversal: Visiting every vertex and edge exactly once in a well-defined order. Two graph traversals: DFS and BFS **DFS:** Visit all successors of a visited node before visiting any successors of any those successors. Algorithm: DFS-recursive(G,s): Mark s a s visited For all neighbours w of s in graph G:	7.5+7.5	15	15

	If w is not visited: DFS-recursive(G, w) Example: **BFS:** Visit all brothers of a visited node before visiting any successors. Algorithm: 1. Mark starting node as visited and enqueue it into queue. 2. While queue is not empty 3. Dequeue next node to become current node. 4. While there is an unvisited child of current node 5. Mark child as unvisited and enqueue child into queue. Example:			
X				

a	**1. Procedure for Quick Sort:**
	/* low --> Starting index, high --> Ending index */
	quickSort(arr[], low, high)
	{
	if (low < high)
	{
	/* pi is partitioning index, arr[pi] is now at right place */
	pi = partition(arr, low, high);
	quickSort(arr, low, pi - 1); // Before pi
	quickSort(arr, pi + 1, high); // After pi
	}
	}
	2. Procedure for partitioning:
	/* This function takes last element as pivot, places
	the pivot element at its correct position in sorted
	array, and places all smaller (smaller than pivot)
	to left of pivot and all greater elements to right
	of pivot */
	partition (arr[], low, high)
	{
	// pivot (Element to be placed at right position)
	pivot = arr[high];
	i = (low - 1) // Index of smaller element
		4+4+3	11	15

```
    for (j = low; j <= high- 1; j++)
    {
      // If current element is smaller than or equal to pivot
      if (arr[j] <= pivot)
      {
        i++;    // increment index of smaller element
        swap arr[i] and arr[j]
      }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

3. Example

b

```
void BubleSort()
{
int n, i, arr[50], j, temp;
cout<<"Enter total number of elements :";
cin>>n;
cout<<"Enter "<<n<<" numbers :";
        for(i=0; i<n; i++)
        {
                cin>>arr[i];
        }
cout<<"Sorting array using bubble sort technique...\n";
        for(i=0; i<(n-1); i++)
        {
                for(j=0; j<(n-i-1); j++)
                {
                        if(arr[j]>arr[j+1])
                        {
                                temp=arr[j];
                                arr[j]=arr[j+1];
                                arr[j+1]=temp;
                        }
                }
        }
cout<<"Sorted list in ascending order :\n";
        for(i=0; i<n; i++)
        {
                cout<<arr[i]<<" ";
        }
}
```

4 | 4

QUESTION WISE ANALYSIS

COURSE: TED (15)-4133

VERSION: 1

COURSE: DATA STRUCTURES

Qn No.	Specific Outcome (as per syllabus)	Module	Content Details	Score	Time in Minutes
I (1)	1.2.4	I	Describe infix, prefix and postfix Expressions	2	10
I (2)	1.3.4	I	Describe Priority Queue and Dequeue	2	
I (3)	2.2.6	II	Discribe about doubly linked lists and circular linked lists.	2	
I (4)	3.1.5	III	Describe Expression trees and Threaded binary trees.	2	
I (5)	4.1.2	IV	Explain graph representations – adjacency matrix method and adjacency list method.	2	
II (1)	1.1.1	I	Explain efficiency of algorithms, complexity and big O notation.	6	50
II (2)	1.2.6	I	Explain evaluation of postfox expression using stack ADT	6	
II (3)	2.2.1	II	Explain linked list and its operations – Find, MakeEmpty, PrintList, FindKth, Insert, Delete, Successor, Predecessor etc.	6	
II (4)	3.1.1	III	Explain binary tree, key terms related to trees and traversal methods.	6	
II (5)	3.1.5	III	Describe Expression trees and Threaded binary trees.	6	
II (6)	4.2.1	IV	Explain and implement linear search and binary search algorithms	6	
II (7)	4.1.5	IV	Describe and implement Warshall's algorithm for all-pairs shortest path	6	
III (a)	1.2.5	I	Explain infix to postfix conversion using Stack ADT	9	30
III(b)	1.2.3	I	Describe Stack ADT with push(), pop(), stackfull() and stackempty()	6	
IV (a)	1.3.1	I	Describe Queue and its operations – Insert and Delete.	9	
IV (b)	1.3.2, 1.3.4	I	Describe circular queue and its array representation. Describe Priority Queue and Dequeue	6	
V	2.2.3	II	Describe LinkedList ADT with find(), makeEmpty(), printList(), findKth(), insert(), delete() etc.	15	30
VI	2.2.4	II	Describe algorithm for implementing stack with LinkedList ADT.	15	

VII	3.1.3	III	Explain binary search trees (BST) and its operations – traversals, insertion, deletion and find.	15	30
VIII (a)	3.1.1	III	Explain binary tree, key terms related to trees and traversal methods.	6	
VIII (b)	3.1.1	III	Explain binary tree, key terms related to trees and traversal methods.	9	
IX (a)	4.1.3	IV	Describe graph traversals – DFS and BFS	15	30
X(a)	4.2.2	IV	Explain and implement bubble sort and quick sort algorithms	11	
X(b)	4.2.2	IV	Explain and implement bubble sort and quick sort algorithms	4	
Total Time					180

CODE: TED (15)-4133

TYPE: TED

COURSE: DATA STRUCTURES

BLUE PRINT

SL No:	Module	Type of Questions							
		Part A		Part B		Part C		Total	
		No. of Questions	Score	No. of Questions	Score	No. of Questions	Score	No. of Questions	Score
1	UNIT I	2	4	2	12	2	30	6	46
2	UNIT II	1	2	1	6	2	30	4	38
3	UNIT III	1	2	2	12	2	30	5	44
4	UNIT IV	1	2	2	12	2	30	5	44
Total		5	10	7	42	8	120	20	172