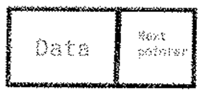
Scoring Indicators

Code: 4133(15)

Qn: No	Scoring Indicators	Split score	Total score
I 1	Tree,Graph	1 X 2	2
I 2	Time complexity analysis estimates the time to run an algorithm.example O(n) for linear search	2	2
I 3	Node  	2	2
I 4	A binary tree is a hierarchical non linear data structure composed of nodes, where each node has at most two children, referred to as the left child and the right child. The topmost node in a binary tree is called the root.	2	2
I 5	An undirected graph with an edge between every pair of vertices.	2	2
II 1	**A data structure is a method for organizing and storing data, which would allow efficient data retrieval and usage.** The data appearing in our data structure is processed by means of certain **operations.** **Traversing** Accessing each record exactly once so that certain items in the record may be processed.(This accessing or processing is sometimes called 'visiting" the records.) **Searching** Finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions. **Inserting** Adding new records to the structure. **Deleting** Removing a record from the structure. **Access:** Retrieving or updating the value of a specific element within the data structure. **Sorting:** Arranging the elements of the data structure in a particular order. **Merging:** Combining two data structures into one.	6 (def-2+any 4 operations)	6
II 2	1. **Stack:** Empty 2. **A (5):** Push 5 onto the stack (Stack: 5) 3. **B (4):** Push 4 onto the stack (Stack: 5, 4) 4. ***:** Pop 4 and 5 from the stack. Perform 4 * 5 = 20. Push 20 back (Stack: 20) 5. **C (6):** Push 6 onto the stack (Stack: 20, 6) 6. **+:** Pop 6 and 20 from the stack. Perform 20 + 6 = 26. Push 26 back (Stack: 26) 7. **D (7):** Push 7 onto the stack (Stack: 26, 7)	6(Result -2 marks+steps 4 marks)	6

Scoring Indicators

Code: 4133(15)

<table>
<tr><td></td><td>8. -: Pop 7 and 26 from the stack. Perform 26 - 7 = 19. Push 19 back (Stack: 19)

Result: The final element in the stack is **19**.</td><td></td><td></td></tr>
<tr><td>II 3</td><td>
<table>
<tr><th>Feature</th><th>Array</th><th>Linked List</th></tr>
<tr><td>Structure</td><td>Contiguous memory locations</td><td>Non-contiguous memory locations (nodes with data & pointers)</td></tr>
<tr><td>Size</td><td>Fixed at creation</td><td>Dynamic</td></tr>
<tr><td>Access</td><td>Fast random access (O(1))</td><td>Slower traversal (O(n) in worst case)</td></tr>
<tr><td>Insertion/Deletion</td><td>Expensive (requires shifting elements)</td><td>Efficient (modify pointers)</td></tr>
<tr><td>Memory Usage</td><td>Potentially wasteful (fragmentation)</td><td>More efficient for sparse data sets</td></tr>
<tr><td>Use Cases</td><td>Fast random access, known size</td><td>Frequent insertions/deletions, dynamic size</td></tr>
</table>
</td><td>6(any 3 comparisons)</td><td>6</td></tr>
<tr><td>II 4</td><td>**Binary Tree Example**

Here's a sample binary tree with the value 8 at the root:

    8
   / \
  3   10
 / \   \
1   6   14

Tree Traversals

There are three main ways to traverse a binary tree:

Inorder Traversal: Visits the left subtree, then the root, and then the right subtree.
Preorder Traversal: Visits the root, then the left subtree, and then the right subtree.
Postorder Traversal: Visits the left subtree, then the right subtree, and then the root.

Performing Traversals on the Example Tree:

Inorder Traversal: 1, 3, 6, 8, 10, 14

Preorder Traversal: 8, 3, 1, 6, 10, 14

Postorder Traversal: 1, 6, 3, 14, 10, 8</td><td>6(2 mrks –binary tree example.
1 mark- types of traversals –
3 marks for traversal results)</td><td>6</td></tr>
</table>

Scoring Indicators

Code: **4133(15)**

II 5

Linear search(sequential search)

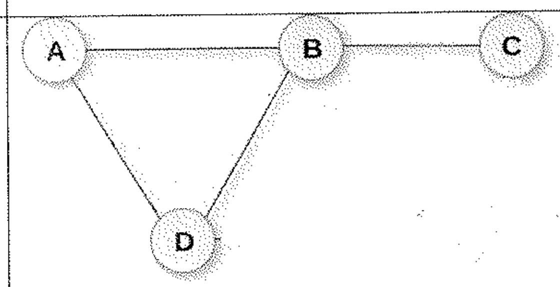
It is a method for finding a particular value in a list, that consists in checking every one of its elements, one at a time and in sequence, until the desired one is found. Let us take an example.

Let there are a list of numbers. 1,3,8,6,5,8,2,9
We have to search if 5 is there in the list or not.
So, we start from the first element from 1.
then we see 3,8,6 and then find 5.
As soon as we find the search element 5 in the list we stop searching,otherwise we continue the search upto the last element.

```
LinearSearch(int a[],int n)

{

for(i=0;i<=n-1;i++)

{
if(a[i]==m)
{
c=1;
break;
}
}
if(c==0)
printf("\nThe number is not in the list");
else
printf("\nThe number is found");
}
```

6(3marks description+3marks algorithm) | 6

II 6



	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	0	0

Any such example –undirected graph/directed graph

6 (grph fig-2marks+adjacency matrix -4marks) | 6

II 7

pop operation is to read the value from the top pointer and move the

6 | 6

Scoring Indicators

Code: 4133(15)

<table>
<tr><td></td><td>pointer to the next node, if there are nodes in a stack.
<pre>
int pop()
{   node * cur;
        if(top==NULL)
                cout<<" stack empty";
        else
        {
                val=top->data;
                cur=top;
                top=top->next;
                delete cur; //free memory
        }
}
</pre></td><td></td><td></td></tr>
<tr><td>III a</td><td>Infix to Postfix Conversion Algo :
Scan the Infix string from left to right.
Initialise an empty stack.
If the scannned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty Push the character to stack.
If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.
Repeat this step till all the characters are scanned.
After all characters are scanned, we have to add any character that the stack may have to the Postfix string. If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
Example :A + B * C becomes A B C * +

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.</td><td>10
(algorithm 5marks, Explanation 5 marks
)</td><td>15</td></tr>
</table>

Scoring Indicators

Code: 4133(15)

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when the are both popped off in lines 6 and 7, their order will be reversed.

III b

A **priority queue** is a data structure for maintaining a collection of items each with an associated key (or priority). A priority queue supports the following operations:

* insert - add an item and its key to the priority queue

* maximum (or minimum) - return the item of highest priority

* extractMax (or extractMin) - remove the item of highest priority and return it as well as the now standard isEmpty, isFull, and size operations.

If each item's priority is taken to be time at which it is inserted, then we get a first-in-first-out structure (i.e. a queue), thus the name priority queue.

deque: a double-ended queue

* can add and remove only from either end

* useful to represent a line where an element can "cut in" at the front if needed

* can be implemented with a linked list with head and tail references add and remove , or an array with sliding front and back indexes

"double-ended queue" – queue-like linear data structure that supports insertion and deletion of items at both ends of the queue.

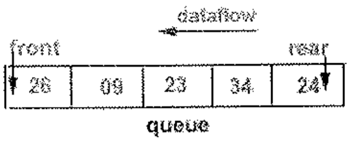
5 (explanation - 2.5*2)

Scoring Indicators

Code: 4133(15)

<table>
<tr><td></td><td>name "double-ended-stack" would be equally appropriate</td><td></td><td></td></tr>
<tr><td>IVa</td><td>Queue

Queue is a linear data structure that operates in a FIFO manner. That means the element comes in first will be removed first from the memory. The main operations in a queue are insertion and deletion of elements. In a queue data structure two pointers namely front and rear are used for these operations. The insertion in a queue takes place at the rear end of a queue and the deletion takes place at the front end of the queue.



ADT

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.
}
Queue insertion logic is as follows,

1.correct the front pointer for the first insertion

2.check for the queue full condition, rear==qsize-1

3. if queue not full rear=rear+1

4.Queue[rear]=val

Queue deletion is as follows,

1.check for the queue empty condition , front==-1

2.if queue not empty val=Queue[front];

3.increment front to the next position, front=front+1;</td><td>9

(queue concept - 3,3+3 for operations)</td><td>15</td></tr>
</table>

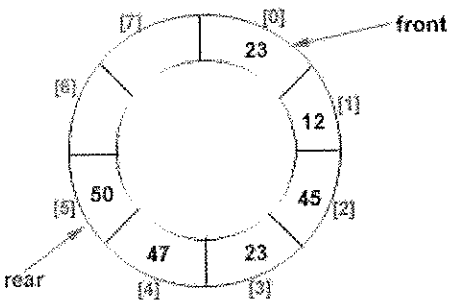
Scoring Indicators

Code: 4133(15)

<table>
<tr><td></td><td>4.reset front and rear if front=rear+1, means no value present in queue</td><td></td><td></td></tr>
<tr><td>IV
b</td><td>

Circular Queue

In case of an ordinary queue the insertion take place from the rear end and the deletion take place from the front end. In the case of deletion from the front end the data are deleted but the space of the queue is not utilized for the further storage. So this problem is solved by using a circular queue. Even if the rear is full but there is space at the front end then the data can be stored in the front end until the queue overflows.



Initially cqueue[qsize];

rear=front=0;

Insertion algorithm

if (**! full**()) // if rear+1 % qsize == front

{

rear = (rear + 1) %qsize ;

cqueue[rear] = val;

}

Deletion algorithm

if (**!empty**()) // if front == rear

{

front = (front + 1) % qsize;

val=cqueue[front];

return val;

</td><td>6
(Def -3+
description 3
marks)</td><td></td></tr>
</table>

Scoring Indicators

Code: 4133(15)

	}		

V a	**LINKED LIST ADT**
	template<class T>
	class Linkedlist;
	template<class T>
	class Node
	{ T data;
	Node *next;
	friend class Linkedlist<T>;
	};
	template<class T>
	class Linkedlist
	{ Node<T> *head;
	public: Linkedlist();//intitalize head as null
	void insert(T value);//insert at the last
	void display();//print all the elements –traversal
	int search(T value);//search a value in linked list
	void insertpos(int pos,T val);//insert an element in a position
	void Delete(T val);// delete a value in linked list
	void makeEmpty();//de allocate memory by using delete operator
	};
	A **new node** can be created using dynamic memory allocation operator in c++ ' new'. The statement below is used to create a new node with a value and points to NULL.
	Node * cur;
	cur=new Node<T>;
	cur->data=val;
		10
		ADT-4+algorithm
		4+2
			15

cur->next=NULL;

you can assign the first node's address to head as

head=cur;

The procedure to insert a node into an existing link list is

A pointer **prev** is used to hold the address of the node to which the new node **cur** is to be connected .then,

prev->next=cur;

```
void insert(T value)
    {               Node<T> *cur,*prev;
            if(head==NULL)
            {       cur=new Node<T>;
                    cur->data=value;
                    cur->next=NULL;
                    head=cur;
            }
            else
            {       cur=head;
                    while(cur!=NULL)
                    {
                            prev=cur;
                            cur=cur->next;
                    }
            cur=new Node<T>;
            cur->data=value;
            prev->next=cur;
            }       }
```

algorithm: display(head)

Scoring Indicators

Code: 4133(15)

1.display the node data by cur->data if cur not equal to NULL

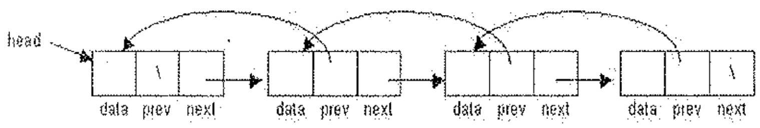
2 move to the next node by cur=cur->next; repeat step 1

```
void display()
        {               Node<T>*cur;
                cur=head;
                cout<<"the elements and address is \n";
                while(cur!=NULL)
                {       cout<<cur->data<<"->";
                        cur=cur->next;
                }
                cout<<"NULL\n";
        }
```

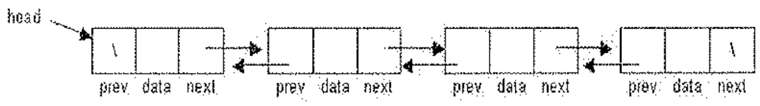
V b **Doubly linked list**

It is a data structure with links to both directions. A node in a doubly linked list contains two address part in addition to the data part. A node will store the address of next node as well as the previous node. Compared to a singly linked list,the forward and backward movement from any node is possible in a doubly linked list. That means the previous data and next data is available in a doubly linked list. The first node's backward link and last node's forward link are given to be NULL

5
(example
2+explanation 3)



or



The definition of a node is ,

class dnode

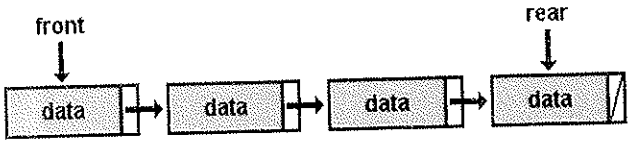
{

	int data; **node * forward;** **node * backward;** };All the operations of singly linked list are possible with doubly linked list.		

VI a	algorithm Search(val)
	
	1.cur=head
	
	2.check whether cur->data=val , if so value present
	
	3.cur=cur->next goto step 2
	
	4.if cur=NULL, element not present.
	
	int search(T value) //find
	{ Node<T>*cur;
	cur=head;
	while(cur!=NULL)
	{ if(value==cur->data)
	{
	cout<<"the element is present\n";
	break;
	}
	else
	{cur=cur->next; }
	}if(cur==NULL)
	{
	cout<<"the element is not present\n";
	return 0; }
	}
		7	15

Scoring Indicators

Code: 4133(15)

VI
b
	Queue implementation using linked list
	
	class node
	{
	int data
	node *next;
	};
	node * front=NULL,*rear=NULL;
	
	
	
	Insertion is done at the rear side using the rear pointer. For the first node , make the rear and front to that node.
	
	void insertion(int val)
	{
	node *cur;
	cur=new node<T>;
	if(!cur)
	cout<<"Queue full";
	else
	{
	if(rear==NULL)
	{
	cur->data=val;
	rear=front=cur;
	}
	else
			8	

Code: 4133(15)

	{
	cur->data=val;
	rear->next=cur;
	rear=cur;
	}
	}
	}
	Deletion is done at the front side,
	void deletion()
	{
	node *cur;
	if(front==NULL)
	printf("\nQueue is empty");
	else
	{
	cur=front;
	printf("\nDeleted element is : %d",cur->data);
	front=front->next;
	delete cur;
	}
	}		

VII	A **binary search tree**, sometimes abbreviated **BST**, is another special kind of binary tree that allows us to store data.. To build a binary search tree from a set of input numbers: 1. Make the first input the **root** of the BST. 2. For each remaining input, recursively compare the input to the root of the tree. a. If the input is **less than** the root, it becomes the		15

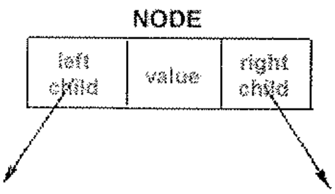
left child of the root (or, recursively, it goes into the left subtree.)

b. If the input is **greater than** the root, it becomes the **right child** of the root (or, recursively, it goes into the right subtree.)

Linked list implementation -structure definition of a node in a binary tree

15

Bst concepts-2mark,algorithms 5+5, Example 3



TREE ADT

```
class node {
   node *left;   // left subtree has smaller elements
   node *right;  // right subtree has larger elements
   int data;
        friend class BST<T>;
};
```

```
template<class T>
class BST
{
private:
        node<T> *root; // Initially root=NULL;

Public:

        int isEmpty();
```

Scoring Indicators

Code: 4133(15)

```
        void inOrder(node<T> *cur);

        void preOrder(node<T> *cur);

        void postOrder(node<T> *cur);

        int insert(T item);

        int delete(T item);

        int find(node<T> *cur, T item);

}
```

insertion

The binary tree **insertion** follows a very simple principle -- for the new element to be added, compare it with the current element in the tree. If its value is less than the current element in the tree then move towards the left side of that element or else to its right. If there is no sub tree on the left, make your new element as the left child of that current element or else compare it with the existing left child and follow the same rule. Exactly same has to done for the case when your new element is greater than the current element in the tree but this time with the right child.

algorithm:insertion

1.starting from the root, find the position of the new node to be inserted, or to find the parent node to which the new node to be inserted.

2.if new value is < parent's value assign the new node as left of parent otherwise assign the new node as right of parent node.

```
insertion (T val)

{

        node* prev,*cur;

                if(root==NULL)
        //tree is empty , creating the root

                {

                        cur=new node<T>;

                        cur->data=val;

                        cur->left=NULL;cur->right=NULL;
```

Scoring Indicators

Code: 4133(15)

```
                root=prev=cur;
    //root  has starting address
            }
            else
            {
                prev=cur=root;
    // starting from root
                while(val!=cur->data && cur!=NULL)
    //finding the parent
                {
    //if parent's left or right equal Null we can insert new
node there
                        prev=cur;

                        if(val<prev->data)
                            cur=prev->left;
                        else
                            cur=prev->right;
                }
                if(val==prev->data)
                        cout<<"Duplicate value";
                else if (val<prev->data)
                {
                        cur=new node<T>;
                        cur->data=val;
                cur->left=NULL;cur->right=NULL;
                        prev->left=cur;
                }
                else
```

```
                {
                        cur=new node<T>;
                        cur->data=val;
                cur->left=NULL;cur->right=NULL;
                        prev->right=cur;
                }
        }
}
```

Inorder traversal:

```
void inordertraversal(node<T> * cur) {
  if (cur != NULL) {
    inordertraversal(cur->left);  // print left subtree
    cout << cur->data << endl; // print this node
    inordertraversal(cur->right); // print right subtree
  }
}
```

Preorder traversal:

```
void preordertraversal(node<T> * cur) {
  if (cur != NULL) {
    cout << cur->data << endl; // print this node
    preordertraversal(cur->left);  // print left subtree
    preordertraversal(cur->right); // print right subtree
  }
}
```

Postorder traversal:

```
void postordertraversal(node<T>* cur) {
  if (cur != NULL) {
```

Scoring Indicators

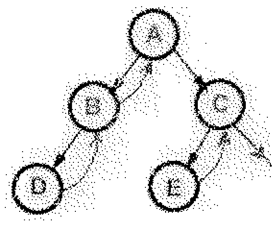
Code: 4133(15)

	postordertraversal(cur->left); // print left subtree
	
	postordertraversal(cur->right); // print right subtree
	
	cout << cur->data << endl; // print this node
	
	}
	
	}		

VIII a	**Expression Trees** An expression tree represents an algebraic expression in such a way that stores its structure and shows how the order of operations applies. The expression tree's structure removes the need to talk about parentheses, as the structure encodes precedence. When we have a single expression based on a binary operator, we draw the expression tree as follows: • The operator is the **root** of the tree. • The operands are the **children**. Because some operations are *not* commutative, order does matter. The operand before the operator is the **left child** and the operand after the operator is the **right child**. Thus, we get a tree with a root and two children. **Eg:** Expression tree for x + y: + / \\ x y When we wish to work with more complicated expressions, we invoke the recursive nature of binary trees. When an operand is an expression rather than a single variable or constant, we simply put the expression tree for that expression in lieu of the operand. **Eg:** Expression tree for x + y * z: + / \\ x * / \\ y z	9 8 Explanation 4+example 4	15

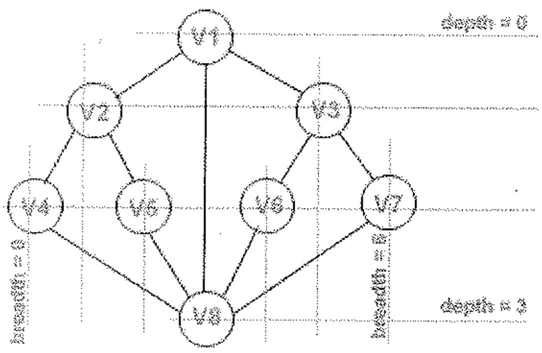
Scoring Indicators

Code: 4133(15)

VIII b	A **Threaded Binary Tree** is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time. Since traversing the three is the most frequent operation, a method must be devised to improve the speed. This is where Threaded tree comes into picture.
	
	If the right link of a node in a tree is NULL, it can be replaced by the address of its inorder successor. An extra field called the rthread is used. If rthread is equal to 1, then it means that the right link of the node points to the inorder successor. If its equal to 0, then the right link represents an ordinary link connecting the right subtree.
	
	
	
	B has no right child and its inorder successor is A and so a thread has been made in between them. Similarly, for D and E. C has no right child but it has no inorder successor even, so it has a hanging thread.
	
	Structure definition of a node in a threaded binary tree
	
	class tnode {
	tnode *left; // left subtree has smaller elements
	tnode *right; // right subtree has larger elements
	int data;
	int rthread; // 0 or 1
	};
		7
		
		Fig 3 + explanation 4	

IXa.	There are two types of graph traversals. 1. Depth first search (DFS) . here from a node it will go to the adjacent node , then it will go to its adjacent node etc. This way from a node v it will traverse to its most depth position. 2. Breadth first search (BFS) ,here from a node it will visit all the adjacent nodes of that node only. So a breadth wise visiting is done	9 Explanation 4 + algorithm 5	15

<table>
<tr>
<td></td>
<td>here.



bfs:v1,v2,v3,v8,v4,v5,v6,v7

BFS algorithm:

bfs(v)
{
    visited(v)=true
    addQ(v)
    while(not emptyQ)
    {
        v=deleteQ
        for all w adjacent to v
            if(not visited(w))
                visited(w)=true
                addQ(w)
    }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IXb</td>
<td>Bubble sort

The basic idea underlying the bubble sort is to pass through the file sequentially several times.Each pass consists of comparing each element in the file with its successor (a[i] and a[i+1] and interchanging the two elements if they are not in proper order.</td>
<td>6</td>
<td></td>
</tr>
</table>