TED (10)-3071 Reg. No. …….

(REVISION-2010) Signature ……..……

FORTH SEMESTER DIPLOMA EXAMINATION IN ENGINEERING/
TECHNOLIGY- MARCH, 2013

DATA STRUCTURE

(Common to CT and IF) [*Time:* 3 hours
(Maximum marks: 100) Marks

PART –A
(Maximum marks: 10)

I. Answer all questions in a sentence

1. Write an example for Postfix expression and state the order of its evaluation
   Postfix notation of [A + B] * C is
   [AB+] *C
   AB+C*, the order of evaluation is always from left to right. Even brackets cannot alter the order of evaluation.
2. Illustrate an advantage of doubly linked list over singly linked list
   A linked list does not allow random access of data. Nodes in a linked list can be accessed only in sequential manner. We can add any number of elements in the list. This is not possible in the case of array
3. Define the term sibling in a tree
   If N is a node in a tree, T that has a left successor s_1 and a right successor s_2 then N iscalled the parent of s_1 and s_2. S_1 ands_2 are said to be *siblings.*
4. Write the structure of a Node in a Binary tree which store Roll No. and Name of a student

```
struct node
{
        intRoll_no;
        string name;
        struct node *next;
}
```

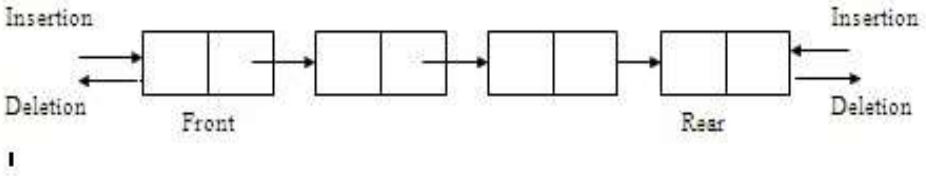
5. Distinguish between directed and undirected graph
   - Directed graphs: G=(V,E) where E is composed of ordered pairs of vertices; i.e. the edgeshave direction and point from one vertex to another.
   - Undirected graphs: G=(V,E) where E is composed of unordered pairs of vertices; i.e. theedges are bidirectional.

PART – B

II. Answer *any five* questions. Each question carries 6 marks

1. Write short note on Double ended Queue

A double-ended queue (dequeue, often abbreviated to deque, pronounced *deck*) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).[1] It is also often called a head-tail linked list, though properly this refers to a specific data structureimplementation



This differs from the queue abstract data type or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

- An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.
- An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.

Both the basic and most common list types in computing, queues and stacks can be considered specializations of deques, and can be implemented using deques.

2. Summarize the basic Data structure operations

The basic Data structure operations are

- Insertion : Inserting a data element to a Data structure
- Deletion: Deleting a data element to a Data structure
- Traversal :Means accessing each and every element from a Data structure
- Search : Searching an element in Data structure
- Insert at front : Inserting a data element to a Data structure in its first position
- Insert after : Inserting a data element to a Data structure in to given position
- Delete at front : Delete an element from first position of a Data structure

- Delete given data : Deleting a data element of a Data structure from given position
- Sorting : Sorting the elements of a Data structure either in ascending or descanting order

3. Write an algorithm to search for a particular data item in a singly linked list

```
Step 1:  [INITIALIZE] SET PTR = START
Step 2:  Repeat Sep 3 while PTR! = NULL
Step 3:          IF POS = PTR
                      Go To step 5
                 ELSE
                      SET PTR = PTR->NEXT
                 [END OF IF]
 [END OF LOOP]
Step 4:  SET POS = NULL
Step 5:  EXIT
```

4. State the advantages of Linked List over Arrays

An array is a linear collection of elements and a linked list is a linear collection of nodes. But unlike array a linked list does not store its nodes in consecutive memory locations.

Another advantage of a linked list over an array is that a linked list does not allow random access of data. Nodes in a linked list can be accessed only in sequential manner. We can add any number of elements in the list. This is not possible in the case of array. For example, if we declare an array as *int marks [10]*, then the array can store maximum of 10 data elements, but not even one more than that. There is no such restriction in case of a linked list.

5. State the properties of a binary tree

- The left sub-tree of a node contains only nodes with keys less than the node's key.
- The right sub-tree of a node contains only nodes with keys greater than the node's key.
- The left and right sub-tree each must also be a binary search tree.
- Each node can have up to two successor nodes.
- There must be no duplicate nodes.
- A unique path exists from the root to every other node.

6. Define the terms : Connected Graph, complete graph and weighted graph

Connected graph: A graph in which there exists a path between any two of its nodes is called a connected graph. That is to say that there are no isolated nodes in a connected graph

Complete graph: A graph G is to be complete if all its nodes are fully connected. That is there is a path from one node to every other node in that graph.

Weighted graph:A graph is said to be weighted if every edge in the graph is assigned some data. In a weighted graph, the edges of the graphs are assigned some weight or length

7. Compute time complexity of quick sort in average case

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases areuniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

$T(N) = T(i) + T(N - i - 1) + cN$

The time to sort the file is equal to

- the time to sort the left partition with **i** elements, plus
- the time to sort the right partition with **N-i-1** elements, plus
- the time to build the partitions

That is time complexity in average case is o(n log n)

PART – C

(Answer *one* full question from each unit. Each question carries 15 marks)

UNIT – I

III.

a) Explain the algorithm for push and pop operations of a stack implemented using array 9

Push Operation

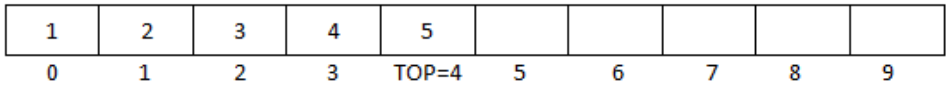
The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value we must check if *TOP = MAX – 1*, because if that is the case, then the stack is full and no more insertions can be further done. If an attempt is made to insert a value in the stack that is already full, an *OVERFLOW* message is printed

Algorithm

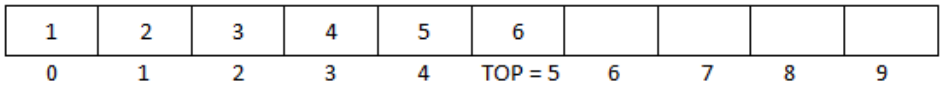
```
Step 1:  IF TOP = MAX - 1 then
              PRINT "OVERFLOW"
         [END OF IF]
```

Step 2: SET TOP = TOP + 1
Step 3: SET STACK [TOP] = VALUE
Step 4: End

In step 1, we first check for the overflow condition. In step 2, *TOP* is incremented so that it points to the next free location in the array. In step 3, the value is stored in the stack array at the location pointed by the *TOP*



Stack before insertion



Stack after insertion

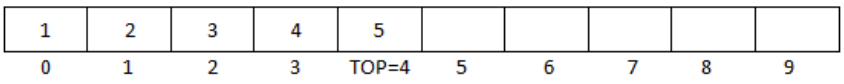
Pop operation

The pop operation is used to delete the topmost element from the stack. However before deleting the value, we must first check if TOP = NULL because if that is the case, the it means the stack is empty and no more deletions can further be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOWmessage is printed.

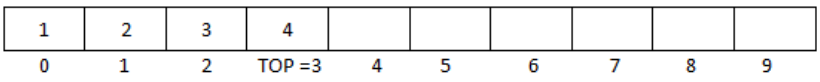
Algorithm

Step 1: IF TOP = NULL then
PRINT "UNDERFLOW"
[END OF IF]
Step 2: SET VAL = STACK [TOP]
Step 3: SET TOP = TOP - 1
Step 4: End

In Step 1 we first check for the underflow condition. In step 2, the value of the location in the stack array pointed by the TOP is stored in VAL. In step 3 TOP is decremented



Stack before Deletion



Stack after deletion

b) Describe the algorithm for converting a decimal number to binary using stack 6

Step 1: Divide the given number by 2.
Step 2: Store the remainder.
Step 3: Repeat the step 1 on the quotient.
Step 4: When quotient is zero, print stored remainders in reverse order.

Example: Let's say we want to convert 27 to binary number.
Step 1: 27/2 =13 Remainder 1
Step 2: 13/2 =6 Remainder 1
Step 3: 6/2 = 3 Remainder 0
Step 4: 3/2 = 1 Remainder 1
Step 5: 1/2 = 0 Remainder 1

When we print the remainder in reverse order, we get **11011** which is binary representation of 27.

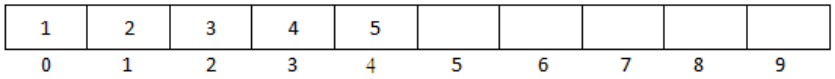
OR

IV. Explain on linear queue and the algorithm to implement its operations using array 15

Queue is a linear data structure where the elements could be inserted at one end called the **"rear"** of the queue and could be retrieved from the other end called **"front"** of the queue.

Queue implements First in First out (FIFO) Order on its elements, that is elements inserted first into the queue would be retrieved first from the queue and vice-versa. Typically queue follows FIFO order, but there could a pre-defined priority for the elements that decides the deletion sequence of the queue elements, in such cases the queue does not follow the traditional FIFO order, such queues are termed as Priority Queue.

Basically FIFO queue are also priority queue as the element inserted first into the queue gets the highest priority. Queue q is an array of size=10 elements. Rear is the position/index of last inserted element initially rear =-1. Front is the position/index of last retrieved element initially front=0

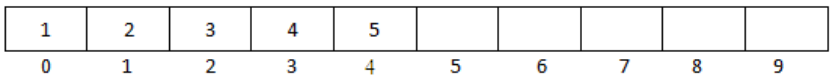


QUEUE

A Queue has two basic operations: *insert* and *delete.* The insert operation adds an element to the end of queue of the stack and the delete operation removes the element from the front or start of the queue.

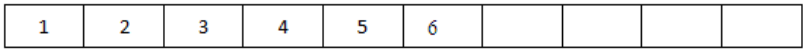
Insert Operation

The insert operation is used to insert an element into the queue. The new element is added as the last element of the queue.



QUEUE

To insert an element with the value 9, we first check if FRONT = NULL. If the condition holds, then the queue is empty. So we allocate memory for a new node, store the *value* in its *data* part and *null* is in its next part. The new node will then be called the FRONT. However, if FRONT! = NULL, then we will insert the new node at the beginning of the linked stack and name his new node as TOP. Thus the updated stack becomes



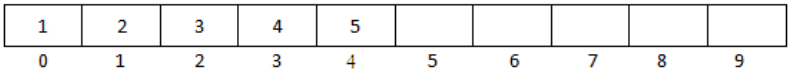
Queue after insertion of a new elemet

Algorithm

Step 1: IF REAR = MAX – 1, then;
WRITE "OVERFLOW"
[END OF IF]
Step 2: IF FRONT == -1 and REAR = -1, then;
SET FRONT = REAR = 0
ELSE
SET REAR =REAR + 1
[END OF IF]
Step 3: SET QUEUE [REAR] = NUM
Step 4: EXIT

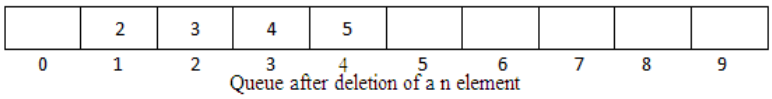
Delete Operation

The delete operation is used to delete the element that was first inserted in a queue. That is the delete operation delete the element whose address is stored in the FRONT. However before deleting the value, we must first check if FRONT = NULL, because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty an UNDERFLOW message is printed.



QUEUE

To delete an element, we first check if FRONT = NULL. If the condition is false, then we delete the first node printed by FRONT. The FRONT will now pointed to the second element of the linked queue. Thus the updated queue will become



Queue after deletion of a n element

Algorithm

Step 1: IF FRONT = – 1, OR FRONT > REAR, then;
WRITE "ONDERFLOW"
ELSE
SET FRONT = FRONT + 1
SET VAL = QUEUE [FRONT]
[END OF IF]
Step 2: EXIT

UNIT – II

V.

a) Explain the algorithm to insert, delete and traverse nodes in a singly linked list 10

Insertion

Insert at the head

Insert a new node at the head of the list is straightforward. The main idea is that we create a new node, set its next link to refer to the current head, and then set head to point to the new node.

Insert at the tail

If we keep a reference to the tail node, then it would be easy to insert an element at the tail of the list. Assume we keep a tail node in the class of SLinkedList, the idea is to create a new node, assign its next reference to point to a null object, set the next reference of the tail to point to this new object, and then assign the tail reference itself to this new node. Initially both head and tail point to null object.

In a sorted list

Sometimes we must insert (to some specific location) new nodes to a list. It is important that we traverse the list to find the place to insert the node. In the traversal process, we must maintain a reference to previous and next nodes of the list. Then we will insert a new node between previous and next.

Insert after/before a given node

To insert a new node after/before the specified node in a singly linked list, first we get the number of the node in an existing singly linked list after which the new node is to be inserted. This is based on the assumption that the nodes of the list are numbered serially from 1 to n. The list is then traversed to get a pointer to the node, whose number is given. If this pointer is x, then the link field of the new node is made to point to the node pointed to by x, and the link field of the node pointed to by x is made to point to the new node.

The algorithm for above cases is, Description: Here START is a pointer variable which contains the address of first node. NEW is a pointer variable which will contain address of new node. N is the value after which new node is to be inserted and ITEM is the value to be inserted.

```
Step 1. If (START == NULL) Then
Step 2.     Print: Linked-List is empty. It must have at least one node
Step 3. Else
Step 4.     Set PTR = START, NEW = START
Step 5.     Repeat While (PTR != NULL)
Step 6.             If (PTR->INFO == N) Then
Step 7.                     NEW = New Node
Step 8.                     NEW->INFO = ITEM
Step 9.                     NEW->LINK = PTR->LINK
Step 10.                    PTR->LINK = NEW
Step 11.                    Print: ITEM inserted
Step 12.            ELSE
Step 13.                    PTR = PTR->LINK
                    [End of Step 6 If]
                [End of While Loop]
        [End of Step 1 If]
Step 14. Exit
```

Deletion

There are 4 cases,

- The first node is deleted
- The last node deleted
- The node after a given node is deleted
- The node is deleted from a sorted linked list

Here START is a pointer variable which contains the address of first node. PTR is a pointer variable which contains address of node to be deleted. PREV is a pointer variable which points to previous node. ITEM is the value to be deleted.

```
Step 1. If (START == NULL) Then [Check whether list is empty]
Step 2.             Print: Linked-List is empty.
Step 3. Else If (START->INFO == ITEM) Then
                    [Check if ITEM is in 1st node]
Step 4.             PTR = START
Step 5.             START = START->LINK [START now points to 2nd node]
Step 6.             Delete PTR
Step 7. Else
Step 8.             PTR = START, PREV = START
Step 9.             Repeat While (PTR != NULL)
Step 10.                    If (PTR->INFO == ITEM) Then [If ITEM matches with PTR->INFO]
Step 11.                            PREV->LINK = PTR->LINK [Assign LINK field of PTR to PREV]
Step 12.                            Delete PTR
Step 13.                    Else
Step 14.                            PREV = PTR [Assign PTR to PREV]
Step 15.                            PTR = PTR->LINK [Move PTR to next node]
                            [End of Step 10 If]
                    [End of While Loop]
```

Step 16. Print: ITEM deleted
[End of Step 1 If]
Step 17. Exit

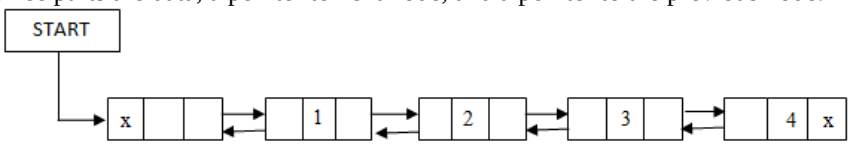
Traversal

Stepping through, or traversing, all the entries of a linked list from beginning to end following the chain of references is a useful operation in practice. Moreover, this process may be enhanced to search for and locate specific nodes.

Algorithm traverseList()
step 1: SET curNode = head
step 2: Repeate step 3 & 4 while curNode != null
Step 3: Apply process to curNode->data
step 4: SET curNode = curNode.getNext()
[END OF LOOP]
Step 5: EXIT

b) Write short note on Doubly linked list 5

A doubly linked list or a two – way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore it consists of three parts, and not just two. Then the three parts are data, a pointer to next node, and a pointer to the previous node.



Doubly linked list

In C, the structure of a doubly linked list is given as,

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
};
```

The prev field of the first node and the next field of the last node will contain NULL. The prev field is used to store the address of the preceding node. This would enable to traverse the list in the backward direction as well.

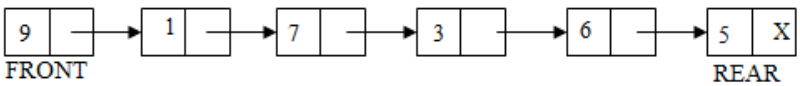
Thus we see that a doubly linked list calls for more space per node and more expensive basic operations. A doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a doubly linked list is that it makes searching twice as efficient.

OR

VI. Explain on queue and the algorithm to implement a queue using singly linked list 15

In case of the queue is a very small one or its maximum size is known advance, then the array implementation of the stack given as efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e. the linked representation is used

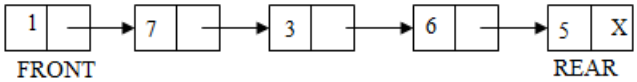
In a linked queue, every element has two parts, one that stores the data and another that stores the address of next element. The START pointer of linked list is used as the FRONT. Here, we will also use another pointer called REAR, which will store the address of last element in the queue. All insertions will be done at the *rear* end and all the deletions are done at the *fron*t end. If FRONT = REAR = NULL, then it indicates that the queue is empty.



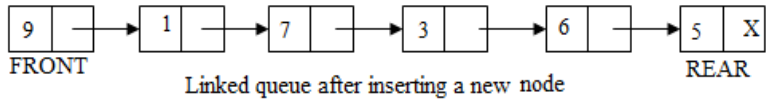
A Queue has two basic operations: *insert* and *delete.* The insert operation adds an element to the end of queue of the stack and the delete operation removes the element from the front or start of the queue.

Insert Operation

The insert operation is used to insert an element into the queue. The new element is added as the last element of the queue.



To insert an element with the value 9, we first check if FRONT = NULL. If the condition holds, then the queue is empty. So we allocate memory for a new node, store the *value* in its *data* part and *null* is in its next part. The new node will then be called the FRONT. However, if FRONT! = NULL, then we will insert the new node at the beginning of the linked stack and name his new node as TOP. Thus the updated stack becomes



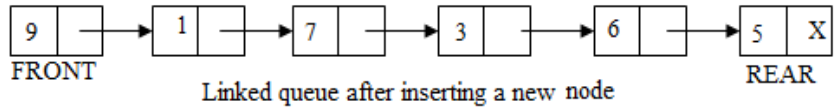
Linked queue after inserting a new node

Algorithm

Step 1: Allocate memory for new node and name it as PTR
Step 2: SET PTR->DATA = VAL
Step 3: IF FRONT = NULL, then
SET FRONT = REAR = PTR;
SET FRONT ->NEXT = REAR->NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR->NEXT = NULL
[END OF IF]
Step 4: EXIT

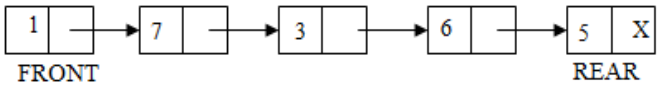
Delete Operation

The delete operation is used to delete the element that was first inserted in a queue. That is the delete operation delete the element whose address is stored in the FRONT. However before deleting the value, we must first check if FRONT = NULL, because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty an UNDERFLOW message is printed.



Linked queue after inserting a new node

To delete an element, we first check if FRONT = NULL. If the condition is false, then we delete the first node printed by FRONT. The FRONT will now pointed to the second element of the linked queue. Thus the updated queue will become



Algorithm

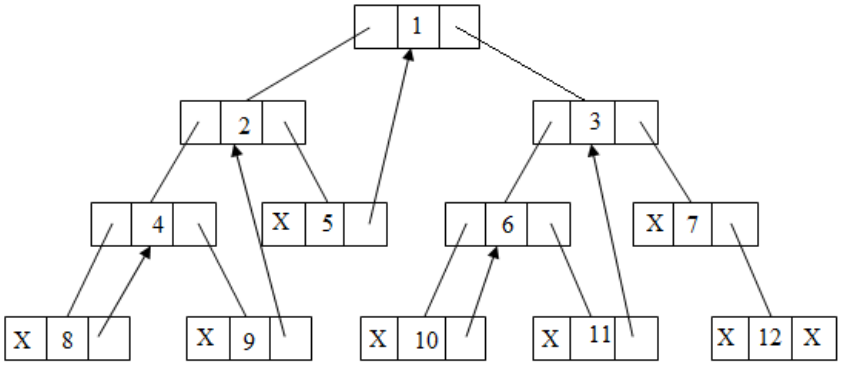
Step 1: IF FRONT = NULL, then
Write "UNDERFLOW"
GOTO STEP 3
[END OF IF]
Step 2: SET PTR = FRONT
Step 3: FRONT = FRONT->NEXT
Step 4: FREE PTR
Step 5: END

UNIT – III

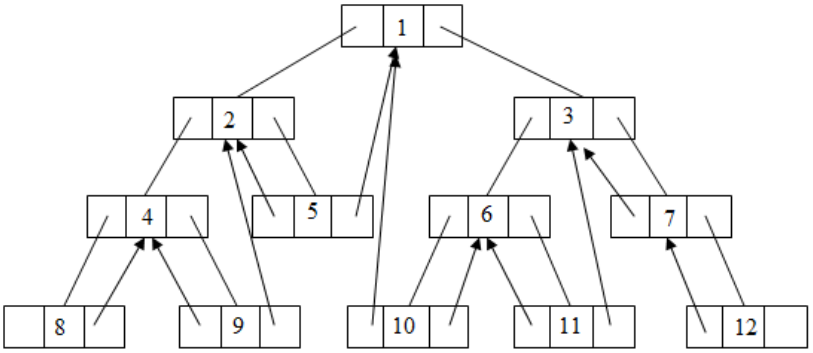
VII.

a) Explain about threaded binary tree 8

A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers. In a linked representation, a number of nodes contain a NULLL pointer, either in their left or right fields or in both. This space that is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information. For example, the NULL entries can be replaced to store a pointer to the in-order predecessor, or the in-order successor of the node. These special pointers are called **threaded trees**. In the linked representation of a threaded binary tree, threads will be represented using dotted lines. A threaded binary tree may correspond to one-way threading or a two-way threading.



Linked representation of a binary tree with one-way threading



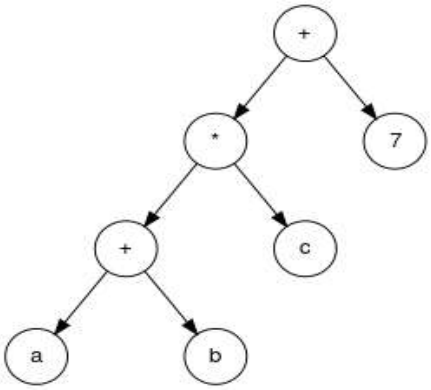
Linked representation of the binary tree with two-way threading

b) Explain about expression tree 7

A **binary expression tree** is a specific application of a binary tree to evaluate certain expressions. Two common types of expressions that a binary expression tree can represent are algebraic and Boolean. These trees can represent expressions that contain both unary and binary operators.

In general, expression trees are a special kind of binary tree. A binary tree is a tree in which all nodes contain zero, one or two children. This restricted structure simplifies the programmatic processing of Expression trees.

The leaves of a binary expression tree are operands, such as constants or variable names, and the other nodes contain operators. These particular trees happen to be binary, because all of the operations are binary, and although this is the simplest case, it is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the unary minus operator. An expression tree, *T*, can be evaluated by applying the operator at the root to the values obtained by recursively evaluating the left and right sub-trees



OR

VIII.

a) Discuss the recursive algorithm to determine the height of a Binary tree 8

Step 1: IF TREE = NULL

RETURN 0

```
        ELSE
                SET HL = HEIGHT (Left Sub-Tree of TREE)
                SET HR = HEIGHT (Right Sub-Tree of TREE)
                SET H = 1 + Maximum of HL & HR
                RETURN H
        [END OF IF]
    Step 2: END
```

In order to determine the height of a binary search tree, we calculate the height of the left sub tree and the right sub-tree. We can solve this easily using recursion. Because, each of the left-Child and right-Child of a node is a sub-tree itself. We first compute the max height of left sub-tree, and then compute the max height of right sub-tree. Therefore, the max height of the current node is the greater of the two max heights + 1. For the base case, the current node is NULL, we return zero. NULL signifies there is no tree; therefore its max height is zero

b) Discuss the algorithm to delete a node from BST 7

Algorithm to delete a node from the binary search tree

```
Step 1:  IF TREE = NULL, then
                Write "VAL not found in the tree"
         ELSE IF VAL < TREE->DATA
                Delete (TREE->LEFT, VAL)
         ELSE IF VAL > TREE->DATA
                Delete (TREE->RIGHT, VAL)
         ELSE IF TREE->LEFT AND TREE->RIGHT
                SET TEMP = findLargestNode (TREE->LEFT)
                SET TREE->DATA = TEMP->DATA
                Delete (TREE->LEFT, TEMP->DATA)
         ELSE
                SET TEMP = TREE
                IF TREE->LEFT = NULL AND TREE->RIGHT = NULL
                        SET TREE = NULL
                ELSE IF TREE->LEFT!= NULL
                        SET TREE = TREE->LEFT
                ELSE
                        SET TREE = TREE->RIGHT
                FREE TEMP
        [END OF IF]
    Step 2: End
```

UNIT – IV

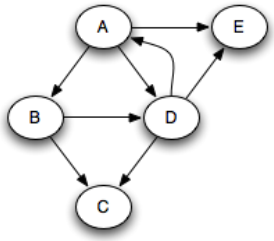
IX. Explain the graph traversal algorithm with example 15

Breadth-first search (BFS) is a graph traversal algorithm that explores nodes in the order of their distance from the roots, where distance is defined as the minimum path length from a root to the node. Its pseudo-code looks like this:

```
// let s be the source node
frontier = new Queue()
mark root visited (set root.distance = 0)
frontier.push(root)
while frontier not empty {
   Vertex v = frontier.pop()
for each successor v' of v {
  if v' unvisited {
  frontier.push(v')
  mark v' visited (v'.distance = v.distance + 1)
  }
  }
}
```

Here the white nodes are those not marked as visited, the gray nodes are those marked as visited and that are in frontier, and the black nodes are visited nodes no longer in frontier. Rather than having a visited flag, we can keep track of a node's distance in the field v.distance. When a new node is discovered, its distance is set to be one greater than its predecessor v.When frontier is a first-in, first-out (FIFO) queue, we get breadth-first search. All the nodes on the queue have a minimum path length within one of each other. In general, there is a set of nodes to be popped off, at some distance *k* from the source, and another set of elements, later on the queue, at distance *k+1*. Every time a new node is pushed onto the queue, it is at distance *k+1* until all the nodes at distance *k* are gone, and *k* then goes up by one. Therefore newly pushed nodes are always at a distance at least as great as any other gray node.

Suppose that we run this algorithm on the following graph, assuming that successors are visited in alphabetic order from any given node:



In that case, the following sequence of nodes pass through the queue, where each node is annotated by its minimum distance from the source node A. Note that we're pushing onto the right of the queue and popping from the left.

A0 B1 D1 E1 C2

Clearly, nodes are popped in distance order: A, B, D, E, C. This is very useful when we are trying to find the shortest path through the graph to something. When a queue is used in this way, it is known as a **worklist**; it keeps track of work left to be done.

Depth-first search

What if we were to replace the FIFO queue with a LIFO stack? In that case we get a completely different order of traversal. Assuming that successors are pushed onto the stack in *reverse* alphabetic order, the successive stack states look like this:

A

B D E

C D E

D E

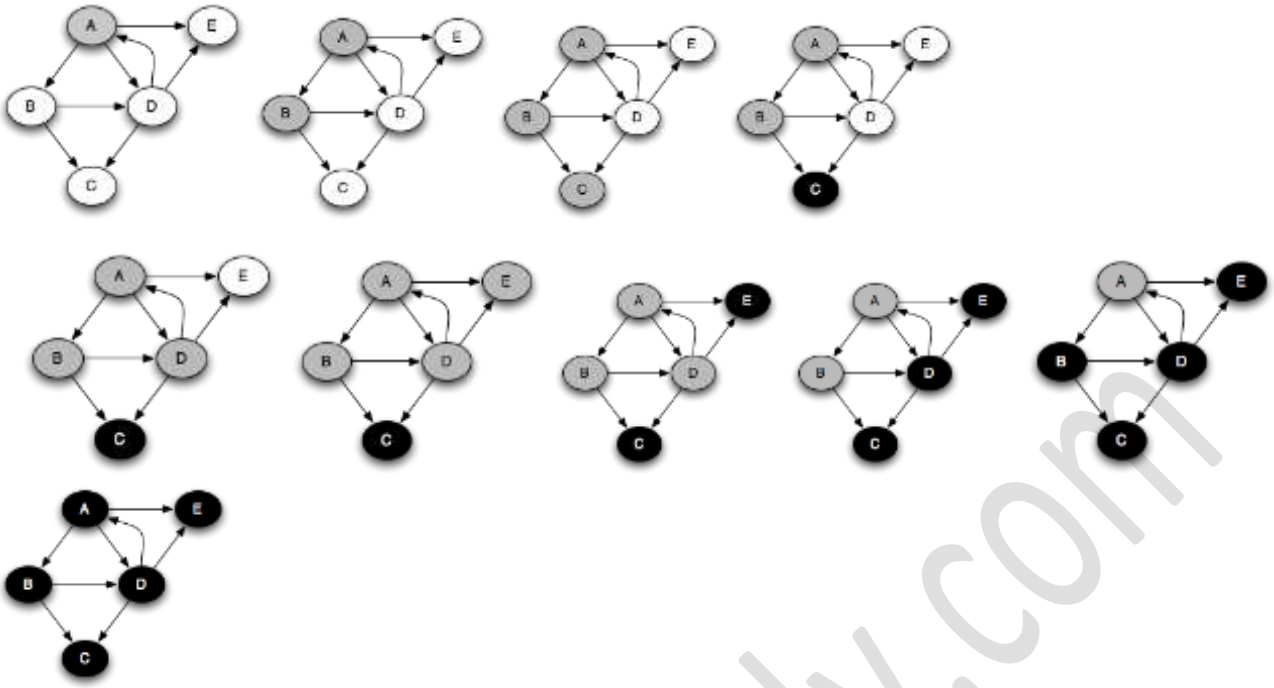
E

With a stack, the search will proceed from a given node as far as it can before backtracking and considering other nodes on the stack. For example, the node E had to wait until all nodes reachable from B and D were considered. This is a **depth-first search**.

A more standard way of writing depth-first search is as a recursive function, using the program stack as the stack above. We start with every node white except the starting node and apply the function DFS to the starting node:

```
DFS(Vertex v) {

mark v visited

set color of v to gray

for each successor v' of v {

   if v' not yet visited {

   DFS(v')

   }

   }

set color of v to black

}
```

You can think of this as a person walking through the graph following arrows and never visiting a node twice except when backtracking, when a dead end is reached. Running this code on the graph above yields the following graph colorings in sequence;those are reminiscent of but a bit different from what we saw with the stack-based version.



OR

X.

a) Discuss the different methods to represent directed graph in memory with suitable examples 10

There are two common methods are used to represent graphs. They are:

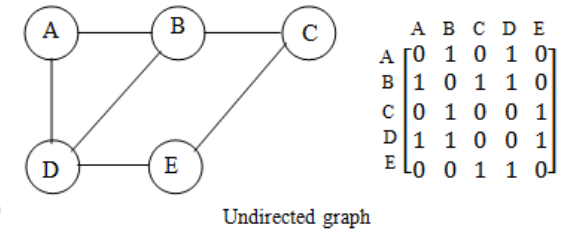
- Sequential representation by adjacency matrix
- Linked representation using adjacency list

Adjacency matrix representation

An adjacency matrix is used to represent which nodes are adjacent to one another. Two nodes are adjacent if there is an edge connecting them.

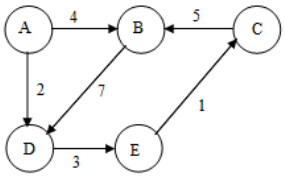
In a directed graph G, node v is adjacent to node u, and then there is definitely an edge from u to v. i.e. if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G have any nodes, and then adjacency matrix will have the dimensions of n x n.

Graphs and their corresponding adjacency matrix



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

Undirected graph



	A	B	C	D	E
A	0	4	0	2	0
B	0	0	0	7	0
C	0	5	0	0	0
D	0	0	0	0	3
E	0	0	1	0	0

Weighted graph

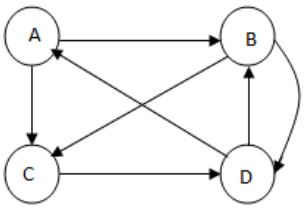
Adjacency list

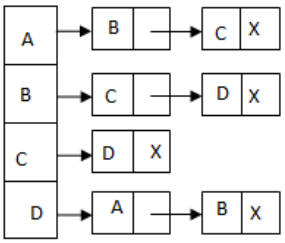
The adjacency list is another way in which graph can be represented in computer's memory. This structure consist of a list of all nodes in G. furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages are:

- It easy to follow and clearly shows the adjacent nodes of a particular node
- It is often used for storing graphs that have a small-to-moderate number of edges
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list

Example





b) Discuss the algorithm for Linear Search and compute its complexity 5

Linear search is the simplest search algorithm. It is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

LINEAR_SEARCH (A, N, VAL, POS)

Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 0
Step 3: Repeat Step 4 while I<N
Step 4: IF A [I] = VAL, then
SET POS = I
PRINT POS
Go to step 6
[END OF IF]
[END OF LOOP]
Step 5: PRINT "Value Not Present In the array"
Step 6: EXIT

Case	Best Case	Worst Case	Average Case
item is present	O(1)	O(n)	O(n/2) = O(n)
item is not present	O(n)	O(n)	O(n)