TED (10)-3071 Reg. No. …….

(REVISION-2010) Signature ……..……

FORTH SEMESTER DIPLOMA EXAMINATION IN ENGINEERING/
TECHNOLIGY- OCTOBER, 2012

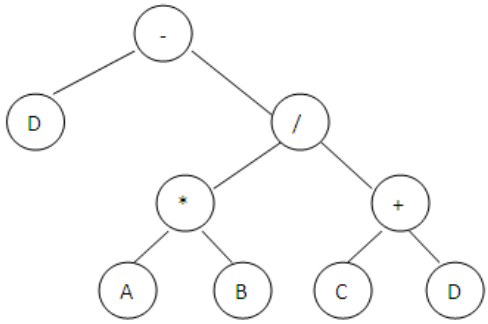
DATA STRUCTURE

(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. List any two non-linear data structures
   a. Trees
   b. Graphs
2. Illustrate a method for dynamic memory allocation
   Malloc() and calloc() functions
3. State the criteria for a Binary search Tree.
   All the nodes in the left sub tree have a value less than that of the root node. Correspondingly all the nodes in the right sub tree have value either greater than or equal to the root node.
4. Draw an expression tree corresponding to the expression (A*B)/(C+D) – E



5. List any two applications of Graph
   a. In circuit networks where points of connections are drawn as vertices and component wires become the edges of the graph.
   b. In maps that draw cities/states/regions and adjacency relations as edges.

PART - B

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

1. State with example the different notations for representing expressions.

   Infix, postfix and prefix notations are three different but equivalent notations of writing algebraic expressions.

   While writing an arithmetic expression using the **infix notation**, the operator is placed between the two operands A and B.although it is easy for us to write expressions using infix notation, but computers find it difficult to parse because they need a lot of information to evaluate the expression. For example, A + B; here the '+' operator is placed between the two operands A and B

   In the **postfix notation**, the operator is placed after the operands. For example, if an expression is written as A +B in infix notation, the same expression can be written as AB+ in postfix notation

   A **prefix expression** is also evaluated from left to right; the operators in this case are placed before the operands. For example, if A + B is an expression in infix notation, then the corresponding expression in prefix notation is given by + AB
2. Write short note on Priority Queue.

   A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

   The general rule of processing the elements of a priority queue is:

   - An element with higher priority is processed before an element with a lower priority
   - Two elements with the same priority are processed on a first-come-first-served(FCFS)

   A priority queue is somewhat similar to a queue, with an important distinction: each item is added to a priority queue with a priority level, and will be later removed from the queue with the highest priority element first. That is, the items are (conceptually) stored in the queue in priority order instead of in insertion order.

   Create a priority queue. The queue must support at least two operations:

1. Insertion. An element is added to the queue with a priority (a numeric value).
2. Top item removal. Deletes the element or one of the elements with the current top priority and return it.

3. Brief 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.

4. State the advantages of Doubly Linked list over singly Linked list

A doubly linked list can be traversed in both directions (forward and backward). A singly linked list can only be traversed in one direction.

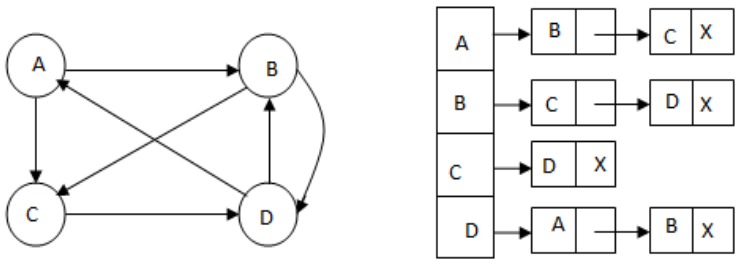
A node on a doubly linked list may be deleted with little trouble, since we have pointers to the previous and next nodes. A node on a singly linked list cannot be removed unless we have the pointer to its predecessor. On the flip side however, a doubly linked list needs more operations while inserting or deleting and it needs more space (to store the extra pointer).

In a single link list, you have only a pointer pointing to the next node. While in a double link list, you have previous and next pointers. Hence you can traverse only in a single direction in a single linked list while in a double linked list, you can traverse in either directions. Hence operations like deleting a successor or a predecessor, printing the list in reverse order, etc can be easily done in double linked list compared to a single link list

5. Write an algorithm to search for an element in a BST.

```
Step 1:  IF TREE->DATA = VAL OR TREE = NULL then
                Return TREE
            ELSE
                IF VAL < TREE->DATA
                        Return searchElement(TREE->LEFT, VAL)
                ELSE
                        Return searchElement(TREE->RIGHT, VAL)
                [END OF IF]
            [END OF IF]
Step 2: End
```

6. Draw a Directed Graph and represent it using an Adjacency list.



7. Write the algorithm for implementing bubble sort.

```
BUBBLE_SORT(A, N)
Step 1:  Repeat steps 2 For I = 0 to N - 1
Step 2:        Repeat For J = 0 to N – I
Step 3:              If A[J] > A[J + 1], then
                                        SWAP A[J] and A[J + 1]
                     [End of Inner Loop]
               [End of Outer Loop]
Step 4: EXIT
```

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

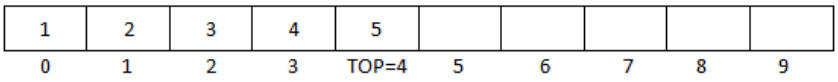
Push Operation

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

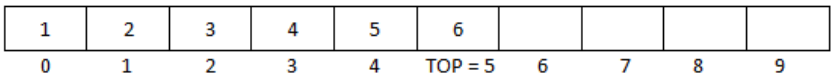
Algorithm

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

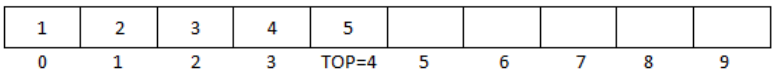
Pop operation

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.

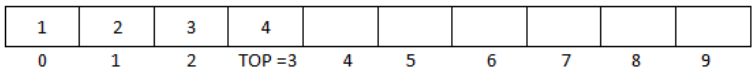
Algorithm

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

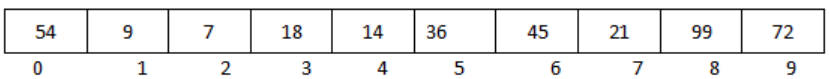
(b) Summarize the applications of stack 6

- Reverse the order of data(we have already seen its example in Recursion)
- Convert infix expression into postfix
- Convert postfix expression into infix
- Backtracking problem
- System stack is used in every recursive function
- Converting a decimal number into its binary equivalent

OR

IV. Explain circular queue and the algorithm to implement its operations 15

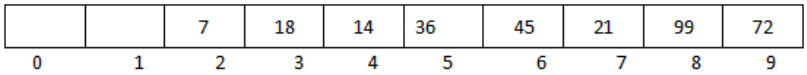
In linear list, insertions can be done only at one end called the *rear*and deletion always done from the other end called the *front.*



Queue

Here *front* = 0 and *rear* = 9

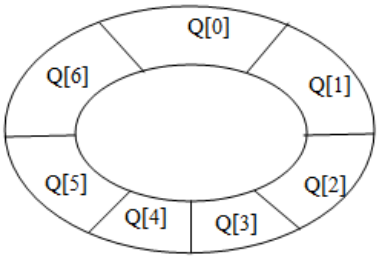
If we want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletion are made. The queue will then be given below



Queue after successive deletions

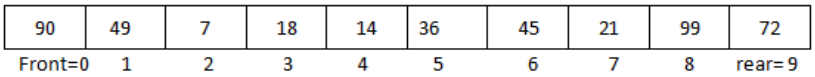
Here *front* = 2 and *rear* = 9

Suppose we want to insert a new element in the queue. Even though there is space is available, the OVERFLOW condition still exists because the condition *rear = Max – 1*still holds true. This is a major drawback of a linear queue.



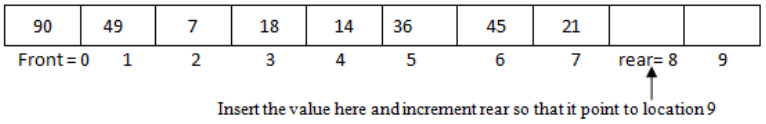
In a circular queue, the first index comes right after last index. A circular queue will be full, only when *front = 0*and *rear = max – 1*.A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations. The only difference will be in the code that performs insertion and deletion operations. For insertion, we now have to check for the following three conditions:

- If front = 0 and rear = MAX – 1, then print that the circular queue is full.



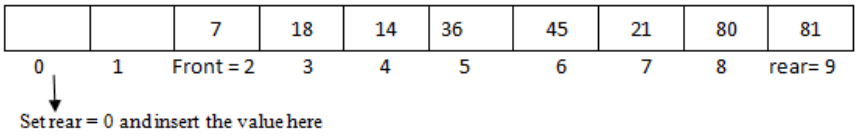
Full Queue

- If rear!= MAX – 1, then the value will be inserted and rear will be incremented as illustrated as below



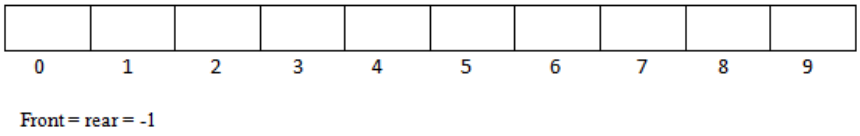
Queue with vacant location

- If front! = 0 and rear = Max – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there, as shown below



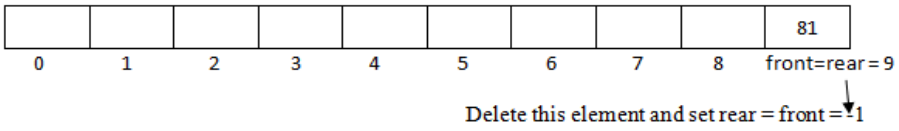
Queue with vacant location

- If front = -1, then there are no elements in the queue. So an underflow condition will be reported.



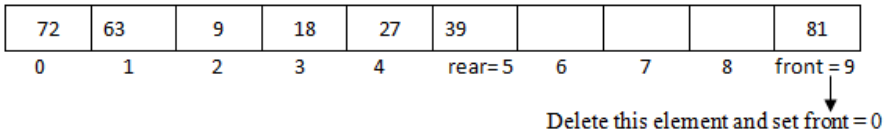
Empty Queue

- If the queue is not empty and after returning the value on the front, front = rear, then the queue has now become empty and so, front and rear are set to -1. This is illustrated as shown in figure



Queue with a single element

- If the queue is not empty and after returning the value on the front, front = MAX – 1, then front is set to 0.



Queue with front = Max before deletion

Algorithm to insert an element in a circular queue

Step 1: IF FRONT = 0 and REAR = MAX –1, then
PRINT "UNDERFLOW"
ELSE IF FRONT = -1 and REAR = -1, then;
SET FRONT = REAR = 0
ELSE IF REAR = MAX -1 and FRONT!= 0, then;
SETREAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 2: SET QUEUE[REAR] = VAL
Step 3: End

In step 1, we make 3 checks. First for the overflow condition, second to see if the queue is initially empty, third to see if the REAR end has already reach the maximum capacity while there are certain free location before the FRONT end. In step 2 the value is stored in the queue array at the location pointed by the REAR.

Algorithm *to delete an element from a circular queue*

Step 1: IF FRONT =–1, then
PRINT "UNDERFLOW"
SET VAL = -1
[END OF IF]
Step 2: SET VAL = QUEUE [FRONT]
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END OF IF]
[END OF IF]
Step 4: EXIT

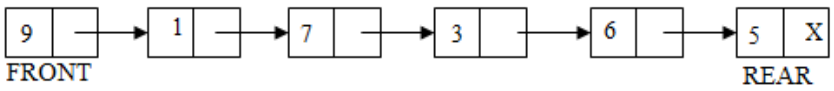
In step 1 we check for the underflow condition. In step 2 the value of queue at the location pointed by FRONT is stored in VAL. In step 3 we make 3 checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these tests.

UNIT – II

V. Explain about linked queue and write the algorithm to implement a Linked Queue 15

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

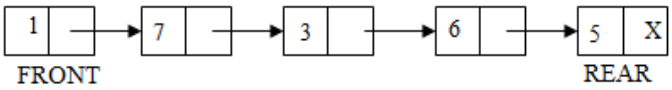
In a linked queue, every element has two parts, one that stores the data and another that stores the address of next element. The START pointer of linked list is used as the FRONT. Here, we will also use another pointer called REAR, which will store the address of last element in the queue. All insertions 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.



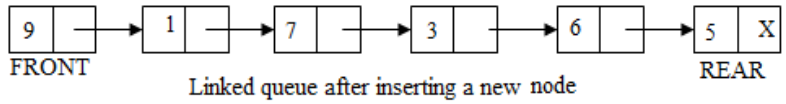
A Queue has two basic operations: *insert* and *delete.* The insert operation adds an element to the end of queue of the stack and the delete operation removes the element from the front or start of the queue.

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



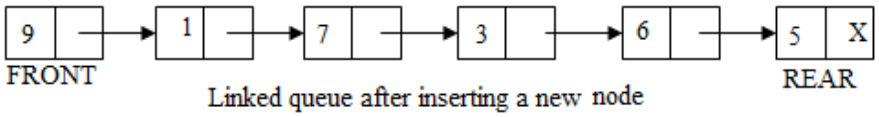
Linked queue after inserting a new node

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

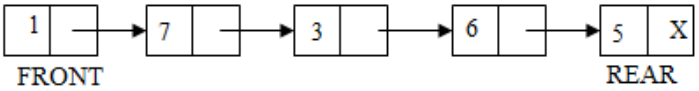
Delete Operation

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

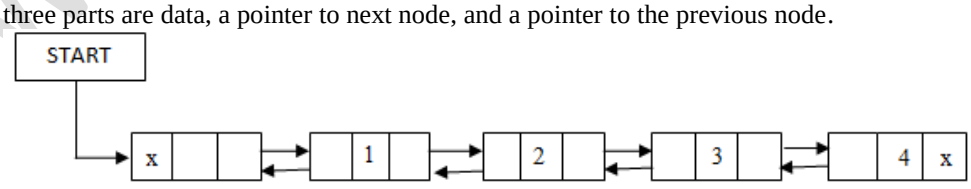
Step 1: IF FRONT = NULL, then
Write "UNDERFLOW"
GOTO STEP 3
[END OF IF]
Step 2: SET PTR = FRONT
Step 3: FRONT = FRONT->NEXT
Step 4: FREE PTR
Step 5: END

OR

VI.

(a) Explain on Doubly Linked List 8

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

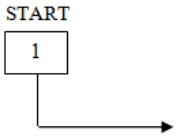
In C, the structure of a doubly linked list is given as,

```
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.



	DATA	PREV	NEXT
1	H	-1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	-1

In figure we see that a variable START is used to store the address of the first node. Here in this example, START = 1, so the first data is stored at address 1, which is H. since this is the first node, it has no previous node and hence stores NULL or -1 in theprevfield. We will traverse the list until we reach a position where the NEXT entry contains -1 or NULL. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list. When we traverse the DATA and NET in this manner, we will finally see that the linked list in the above example stores characters that when put together forms the word HELLO.

Insertion

To do insertion we will take five cases

- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.
- Case 3: The new node is inserted after a given node.
- Case 4: The new node is inserted before a given node.
- Case 5: The new node is inserted in a sorted linked list.

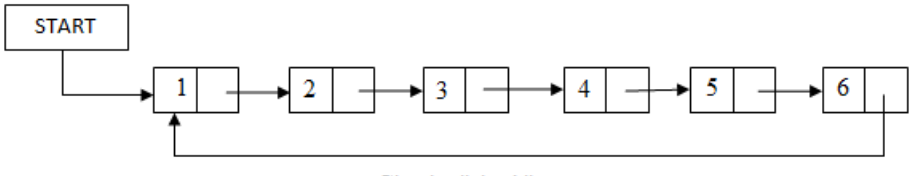
Deletion

To do deletion we will take five cases

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.
- Case 4: The node before a given node is deleted.
- Case 1: The node is deleted from assorted linked list.

(b) Explain on Circular Linked List 7

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward direction, until we reach the same node where we started. Thus a circular linked list has no beginning and no ending.



Circular linked list

The only downside of a circular linked list is the complexity of iteration. Note that there is no storing of NULL value in the list.

Circular linked lists are widely used in the operating system for task maintenance. Take another example where a circular linked list is used when we are suffering the Net, we can use the Back button and Forward button to move to the previous pages that we have already visited. In this case, a circular linked list is used to maintain the

sequence of the list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons. Actually this is done by using either the circular stack or circular queue.

Insertion

To do insertion we will take two cases

- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.

Deletion

To do deletion we will take two cases

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.

UNIT – III

VII.

(a) Discuss the algorithm to delete a node from a Binary Search Tree with suitable example 10

Algorithm to delete a node from the binary search tree

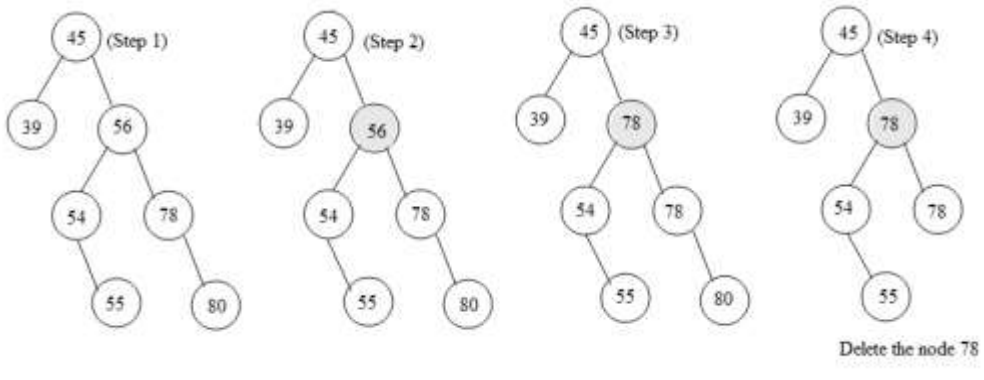
```
Step 1:  IF TREE = NULL, then
                Write "VAL not found in the tree"
         ELSE IF VAL < TREE->DATA
                Delete (TREE->LEFT, VAL)
         ELSE IF VAL > TREE->DATA
                Delete (TREE->RIGHT, VAL)
         ELSE IF TREE->LEFT AND TREE->RIGHT
                SET TEMP = 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
```

In step 1 of the algorithm, we first check if TREE = NULL because if it is true, then the node to be deleted is not present in the tree. However if that is not the case, then we check if the value to be deleted is less than the current node's data. In case the value is less, we call the algorithm recursively on the node's left sub-tree; otherwise the algorithm is called recursively on the node's right sub-tree.

Note that if we have to found the node whose value is equal to VAL, then we check which case of deletion it is. If the node to be deleted has both left and right children, then we find the in – order predecessor of the node by calling findLargestNode (TREE->LEFT) and replace the current node's value with that of its in-order predecessor. Then we call Delete (TREE->LEFT, TEMP->DATA) to delete the initial node of the in-order predecessor. Thus we reduce the case 3 of deletion into either case1 or case2 of deletion.

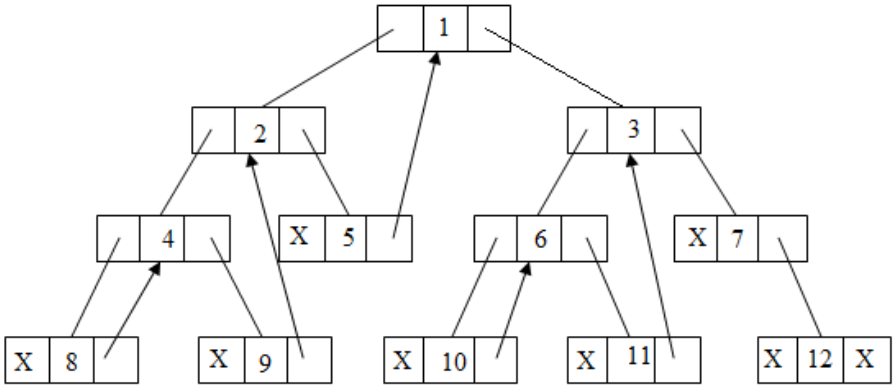
If the node to be deleted does not have any child, then we simply set the node to NULL. Last but not the least, if the node to be deleted has either left or right children but not both, then the current node is replaced by its child node and the initial child node is deleted from the tree.

Example: Deleting node 56 from the given BST

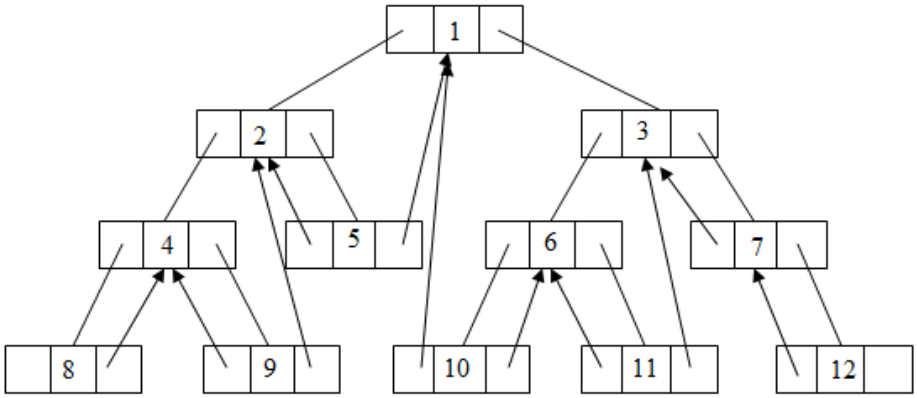


(b) Write short note on Threaded Binary Trees 5

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 peace 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

OR

VIII.

(a) Explain the algorithm to insert nodes in a Binary Search Tree 8

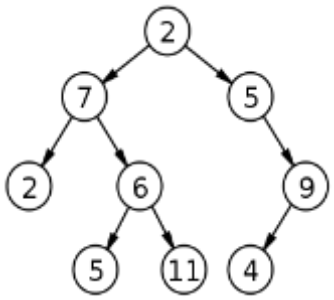
Algorithm

```
Step 1: IF TREE = NULL, then
            Allocate memory for TREE
            SET TREE->DTA = VAL
            SET TREE->LEFT = TREE->RIGHT = NULL
        ELSE
            IF VAL < TREE->DATA
                    Insert (TREE->LEFT, VAL)
            ELSE
                    Insert (TREE->RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: End
```

The insert function is used to add a new node with a given value at the correct position in the binary search. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. In step 1of the algorithm, every call to the insert function checks if the current node of the TREE is NULL. If it is NULL, the algorithm simply adds the node; else it looks at the current node's value and then recurs down the left or right sub-tree. However if the current node's value is less than that of the new node, then the left sub tree is traversed, else the right sub tree is traversed. The insert function continuous moving down the levels of a binary tree until it reaches a leaf node. The new node is added in place of the leaf node by following the rules of

BST. i.e. if the new node's value is greater than that of parent node, the new node is inserted in the right sub-tree, else it is inserted in the left sub-tree.

(b) Explain the algorithm to traverse a BST by in – order. 7



Algorithm In-order (tree)

Step 1: Traverse the left sub tree, i.e., call In-order (left-sub tree)
Step 1: Visit the root.
Step 1: Traverse the right sub tree, i.e., call In-order (right-sub tree)

When trying to figure out what the algorithm for this problem should be, you should take a close look at the way the nodes are traversed. In an in order traversal,If a given node is the parent of some other node(s) then we traverse to the left child. If there is no left child, then go to the right child, and traverse the sub tree of the right child until you encounter the leftmost node in that sub tree. Then process that left child. And then you process the current parent node. And then, the traversal pattern is repeated.

So, it looks like each sub-tree within the larger tree is being traversed in the same pattern – which should make you start thinking in terms of breaking this problem down into sub-trees. And anytime a problem is broken down into smaller problems that keep repeating, you should immediately start thinking in **recursion** to find the most efficient solution. So, let's take a look at the 2 largest sub-trees and see if we can come up with above algorithm

UNIT – IV

IX.

(a) Discuss the different methods of representing graph. 8

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

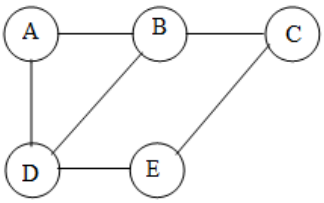
- Sequential representation by adjacency matrix
- Linked representation using adjacency list

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.

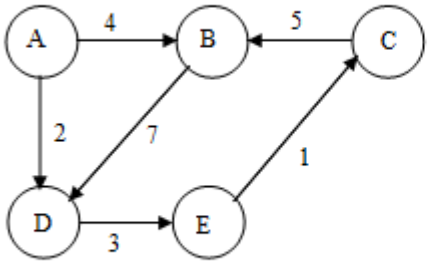
In a directed graph G, node v is adjacent to node u, and then there is definitely an edge from u to v. i.e. if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G have any nodes, and then adjacency matrix will have the dimensions of n x n.

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

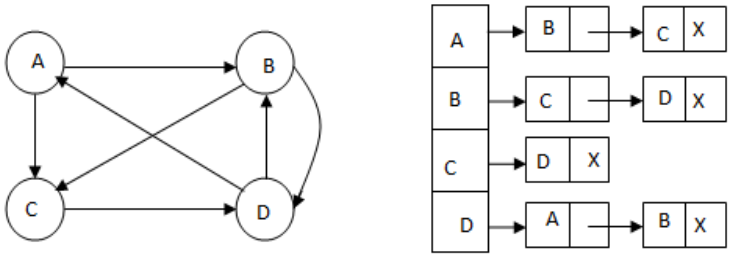
Adjacency list

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) Trace out the steps while sorting the set of Numbers {27, 10, 36, 18, 25, 45} using Quick Sort Algorithm 7

27	10	36	18	25	45

- We choose the first element as the pivot. Set loc = 0, left = 0 and right = 0

27	10	36	18	25	45

Loc right

left

- Scan from right to left. Since a [loc] < a [right], decrease the value of right

27	10	36	18	25	45

Loc right

left

- Since a[loc] > a [right] interchange the values & set loc = right

25	10	36	18	27	45

left right

loc

- Start scanning from left to right. Since a [loc] > a[right], increment the value of left

25	10	36	18	27	45

left Loc right

- Since a[loc] < a [right] interchange the values & set loc = left

25	10	27	18	36	45

Left&loc right

- Start scanning from right to left. Since a [loc] < a[right], decrement the value of right

25	10	27	18	36	45

left right Loc

- Since a[loc] > a [right] interchange the 2 values & set loc = right

25	10	18	27	36	45

left right Loc

- Start scanning from left to right. Since a [loc] > a[right], increment the value of left

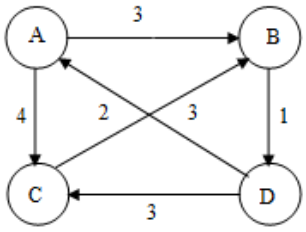
25	10	18	27	36	45

right left Loc

Now left = loc, so the procedure terminates. The pivot element (27) is placed in its correct place. All the elements smaller than 27 are placed before it and those the greater than 27 are placed after it. The left sub array contain 25, 10, 18 and right sub array contains 36 and 45 are sorted in same manner.

OR

X. Draw a weighted graph and apply Warshall's Shortest Path Algorithm. 15



$$\begin{bmatrix} 0 & 3 & 3 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 \\ 2 & 0 & 5 & 0 \end{bmatrix}$$

$$Q_0 = \begin{bmatrix} 9999 & 3 & 3 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 9999 \\ 2 & 9999 & 5 & 9999 \end{bmatrix}$$

$$Q_1 = \begin{bmatrix} 9999 & 3 & 3 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 9999 \\ 2 & 5 & 5 & 9999 \end{bmatrix}$$

$$Q_2 = \begin{bmatrix} 9999 & 3 & 3 & 4 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 4 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

$$Q_3 = \begin{bmatrix} 9999 & 3 & 3 & 4 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 6 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

$$Q_4 = Q\begin{bmatrix} 6 & 3 & 3 & 4 \\ 3 & 6 & 6 & 1 \\ 6 & 3 & 9 & 4 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$