AS-2261

M.Sc.(First Semester) Examination-2013

Paper -fourth

Subject-Data structure with algorithm

Time: Three Hours] **[Maximum Marks: 60**

Section A **(10×2=20)**

Note Attempts all the questions. All carry equal marks

1.1 The time factor when determining the efficiency of algorithm is measured by
a. Counting microseconds b. counting the number of key operations
c. Counting the number of statements. d. Counting the kilobytes of algorithm
Ans. b. counting the number of key operations

1.2 The number of nodes in a complete binary tree of depth d (with root at depth 0) is.
Ans. 2^{d+1} -1

1.3 The average case of quick sort has order
Ans. O(n log n)

1.4 Inorder to get the information stored in a BST in the descending order, one should traverse it in which of the following order?
(A) left, root, right **(B)** root, left, right **(C)** right, root, left **(D)** right, left, root
Ans. (C) right, root, left

1.5 Every internal node in a B-tree of minimum degree 2 can have
(A) 2, 3 or 4 children **(B)** 1, 2 or 3 children **(C)** 2, 4 or 6 children **(D)** 0, 2 or 4 children
Ans. **(B)** 1, 2 or 3 children

1.6 A full binary tree with 'n' non-leaf nodes contains……..total nodes (leaf+non leaf)
Ans. 2n+1 nodes

1.7 In _________ tree the difference between the height of the left sub tree and height of right sub tree, for each node, is not more than one
Ans. AVL

1.8 The number of comparisons required to sort 5 numbers in ascending order using bubble sort is.
(A) 7 **(B)** 6 **(C)** 10 **(D)** 5
Ans. (C) 10

1.9 In a binary tree, the number of terminal or leaf nodes is 10. The number of nodes with two children is….
Ans. 9

1.10 The complexity of Binary search algorithm is….
Ans. **O(log n)**

Section B **(4×10=40)**

Attempt any four questions out of seven. All carry equal marks (4 ×10=40)

2. Sort the given values using Quick Sort.

65	70	75	80	85	60	55	50	45

Ans.
Here 65 is pivot element at 0th position, let p is a index which position is at 1st position element that is 70 and another index q which position is at last element that is 45.
Now p will search an element whose value is greater than 65 and q will search al element whose value is less than 65, in this way p will increment by 1 and q will decrement by 1, and when these values are found, they will interchanged, and if p and q are meet or crossover then value at q will exchange with pivot element in this way the list is divided into two part in first part all the elements are less than qth position element and in next part all the values are greater than qth position element. Now apply the same procedure to each part. Applying this procedure step by step we will got the following sorted elements.

First 70 is greater than 65 and 45 is less than 65, so 70 and 45 are interchanged

65	45	75	80	85	60	55	50	70

Now p will be at 75 which is greater than 65 and q will be at 50 less than 65 so again exchange

65	45	50	80	85	60	55	75	70

Now p is at 80 greater than 65 and q will be at 55 less than 65 so again exchange

65	45	50	55	85	60	80	75	70

Now p is at 85 greater than 65, and q at 60 less than 65, so again exchange

65	45	50	55	60	85	80	75	70

Now here p and q crossover to each other so value at q that is 60 will exchange with pivot element that is 65 and we get

60	45	50	55	**65**	85	80	75	70

Here the list is divided into two parts, one containing all elements less than 65 and other containing all elements greater than 65.
Now apply same procedure step by step , and we get the following sorted list

45	50	55	60	65	70	75	80	85

3. **What is bubble sort and how do you perform it also write the algorithm?**

Ans. In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed. Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array until we get a sorted array.

Let A be a linear array of n numbers. Sorting of A means rearranging the elements of A so that they are in order. Here we are dealing with ascending order. i.e., A[1] < A[2] < A[3] < ...... A[n].

Suppose the list of numbers A[1], A[2], ………… A[n] is an element of array A. The bubble sort algorithm works as follows:

Step 1: Compare A[1] and A[2] and arrange them in the (or desired) ascending order, so that A[1] < A[2].that is if A[1] is greater than A[2] then interchange the position of data by swap = A[1]; A[1] = A[2]; A[2] = swap. Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Continue the process until we compare A[N – 1] with A[N].

Note: Step1 contains n – 1 comparisons i.e., the largest element is "bubbled up" to the nth position or "sinks" to the nth position. When step 1 is completed A[N] will contain the largest element.

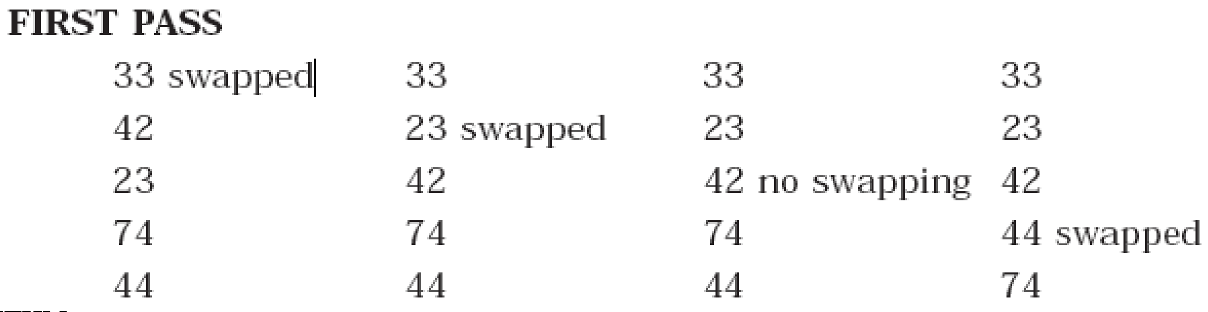
Step 2: Repeat step 1 with one less comparisons that is, now stop comparison at A [n – 1] and possibly rearrange A[N – 2] and A[N – 1] and so on.

Note: in the first pass, step 2 involves n–2 comparisons and the second largest element will occupy A[n-1]. And in the second pass, step 2 involves n – 3 comparisons and the 3rd largest element will occupy A[n – 2] and so on.

Step n – 1: compare A[1]with A[2] and arrange them so that A[1] < A[2]

After n – 1 steps, the array will be a sorted array in increasing (or ascending) order.

The following figures will depict the various steps (or PASS) involved in the sorting of an array of 5 elements. The elements of an array A to be sorted are: 42, 33, 23, 74, 44



FIRST PASS

33 swapped	33	33	33
42	23 swapped	23	23
23	42	42 no swapping	42
74	74	74	44 swapped
44	44	44	74

ALGORITHM

Let A be a linear array of *n* numbers. Swap is a temporary variable for swapping (or interchange) the position of the numbers.

1. Input *n* numbers of an array A
2. Initialise *i* = 0 and repeat through step 4 if (*i* < *n*)
3. Initialize *j* = 0 and repeat through step 4 if (*j* < *n* – *i* – 1)
4. If (A[*j*] > A[*j* + 1])

(*a*) Swap = A[*j*]

(*b*) A[*j*] = A[*j* + 1]

(*c*) A[*j* + 1] = Swap

5. Display the sorted numbers of array A
6. Exit.

TIME COMPLEXITY

The time complexity for bubble sort is calculated in terms of the number of comparisons *f (n)* (or of number of loops); here two loops (outer loop and inner loop) iterates (or repeated) the comparisons. The number of times the outer loop iterates is determined by the number of elements in the list which is asked to sort (say it is *n*). The inner

loop is iterated one less than the number of elements in the list (*i.e.*, *n*-1 times) and is reiterated upon every iteration of the outer loop
$f(n) = (n-1) + (n-2) + ... + 2 + 1 = n(n-1) = O(n2)$.

BEST CASE

In this case you are asked to sort a sorted array by bubble sort algorithm. The inner loop will iterate with the 'if' condition evaluating time that is the swap procedure is never called. In best case outer loop will terminate after one iteration, *i.e.*, it involves performing one pass, which requires n–1 comparisons $f(n) = O(n)$

WORST CASE

In this case the array will be an inverted list (*i.e.*, 5, 4, 3, 2, 1, 0). Here to move first element to the end of the array, *n*–1 times the swapping procedure is to be called. Every other element in the list will also move one location towards the start or end of the loop on every iteration. Thus n times the outer loop will iterate and n (n-1) times the inner loop will iterate to sort an inverted array $f(n) = (n(n-1))/2 = O(n2)$.

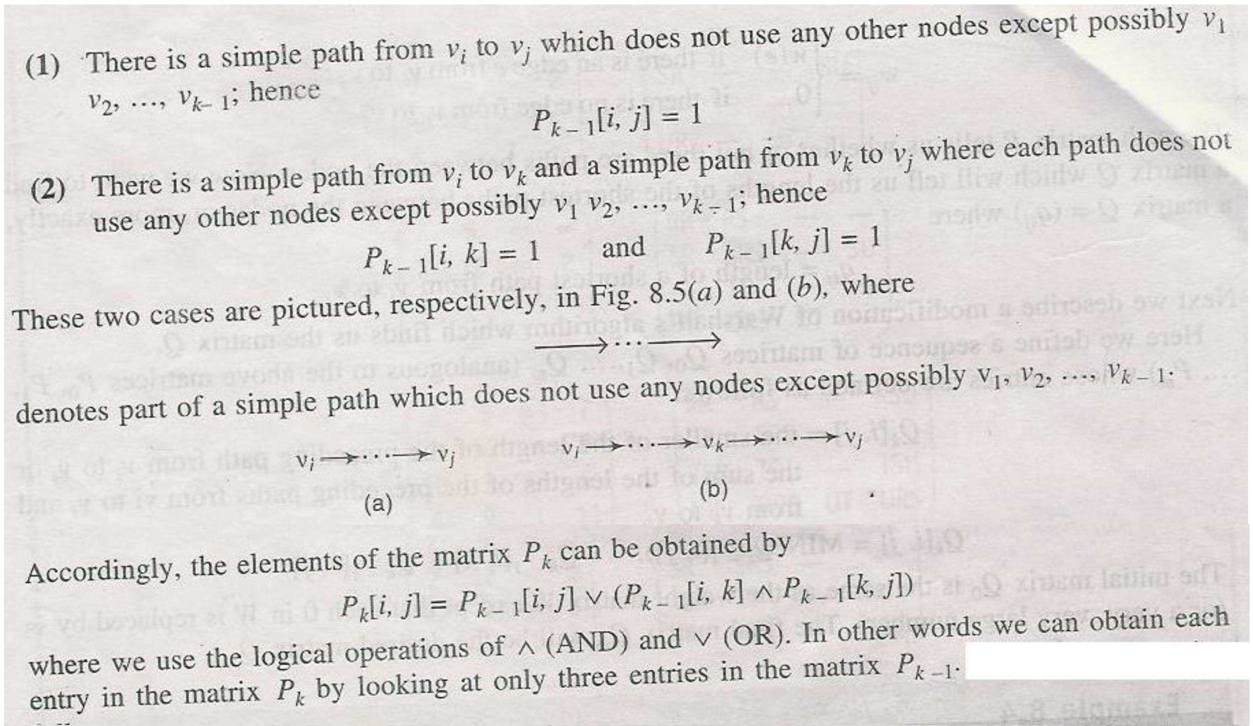
AVERAGE CASE

Average case is very difficult to analyse than the other cases. In this case the input data(s) are randomly placed in the list. The exact time complexity can be calculated only if we know the number of iterations, comparisons and swapping. In general, the complexity of average case is: $f(n) = (n(n-1))/2 = O(n2)$.

4. **Describe warshall's algorithm with example.**

Ans. Warshall's Algorithm

Let G be directed weighted graph with m nodes v1,v2,…..vm. Suppose we want to find the path matrix P of the graph G. warshall gave an algorithm for this purpose.
First we define m-square Boolean matrices P0, P1,P2,…..Pm as follows. Let Pk[i,j] denote the ij entry of the matrix Pk, Then we define :
1 if there is simple path from vi to vj which does not use any other nodes except possibly v1,v2…vk
Pk[i,j] =
0 otherwise
In other words ,
P0[i,j]=1 if there is an edge from vi to vj
P1[i,j]=1 if there is a simple path from vi to vj which does not use any other nodes except possibly v1
P2[i,j]=1 if there is a simple path from vi to vj which does not use any other nodes except possibly v1 and v2.
Here we observe that the matrix P0= A the adjacency matrix of G. and since G has m nodes, the last matrix Pm=P, the path matrix of G.
Warshall observe that Pk[i,j] =1 can occur only if one of the following two cases occurs:



(1) There is a simple path from v_i to v_j which does not use any other nodes except possibly v_1 $v_2, \ldots, v_{k-1}$; hence

$$P_{k-1}[i, j] = 1$$

(2) There is a simple path from v_i to v_k and a simple path from v_k to v_j where each path does not use any other nodes except possibly v_1 $v_2, \ldots, v_{k-1}$; hence

$$P_{k-1}[i, k] = 1 \quad \text{and} \quad P_{k-1}[k, j] = 1$$

These two cases are pictured, respectively, in Fig. 8.5(a) and (b), where

$$\longrightarrow \cdots \longrightarrow$$

denotes part of a simple path which does not use any nodes except possibly $v_1, v_2, \ldots, v_{k-1}$.

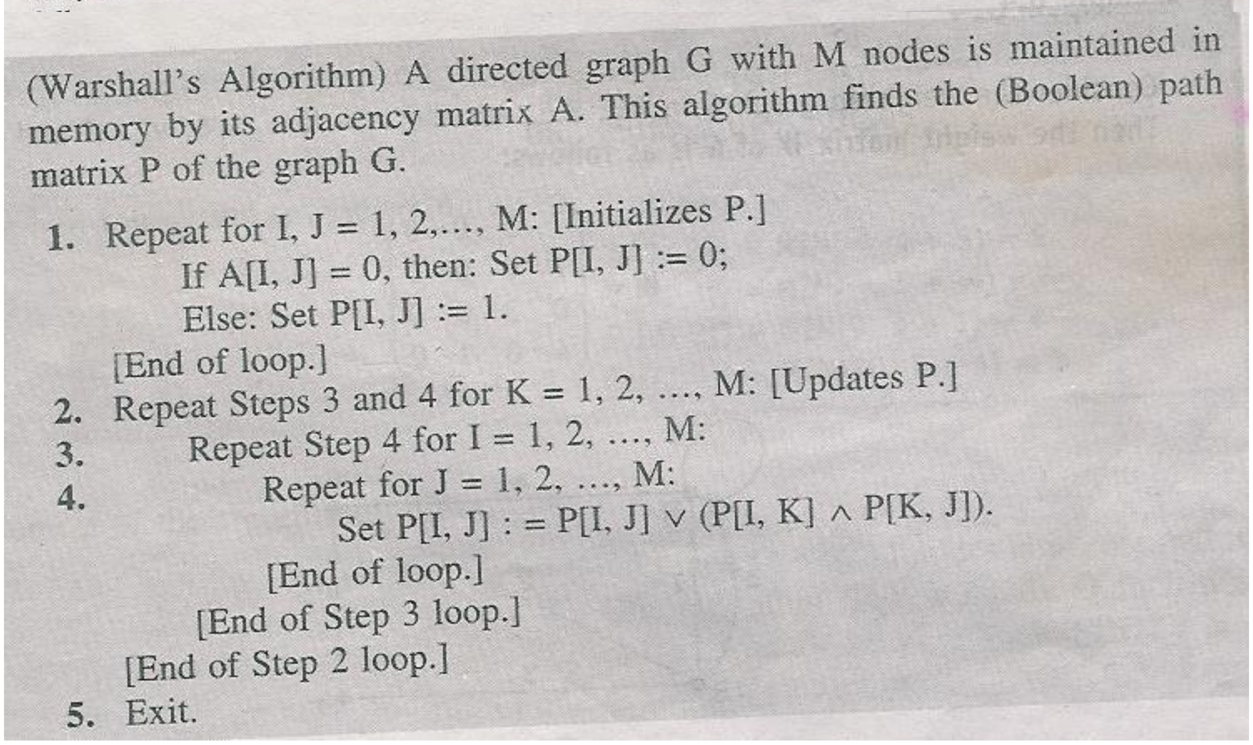
$$v_i \longrightarrow \cdots \longrightarrow v_j \qquad\qquad v_i \longrightarrow \cdots \longrightarrow v_k \longrightarrow \cdots \longrightarrow v_j$$

(a) (b)

Accordingly, the elements of the matrix P_k can be obtained by

$$P_k[i, j] = P_{k-1}[i, j] \vee (P_{k-1}[i, k] \wedge P_{k-1}[k, j])$$

where we use the logical operations of $\wedge$ (AND) and $\vee$ (OR). In other words we can obtain each entry in the matrix P_k by looking at only three entries in the matrix P_{k-1}.



(Warshall's Algorithm) A directed graph G with M nodes is maintained in memory by its adjacency matrix A. This algorithm finds the (Boolean) path matrix P of the graph G.

```
1. Repeat for I, J = 1, 2,..., M: [Initializes P.]
        If A[I, J] = 0, then: Set P[I, J] := 0;
        Else: Set P[I, J] := 1.
   [End of loop.]
2. Repeat Steps 3 and 4 for K = 1, 2, ..., M: [Updates P.]
3.      Repeat Step 4 for I = 1, 2, ..., M:
4.           Repeat for J = 1, 2, ..., M:
                  Set P[I, J] : = P[I, J] ∨ (P[I, K] ∧ P[K, J]).
             [End of loop.]
        [End of Step 3 loop.]
   [End of Step 2 loop.]
5. Exit.
```

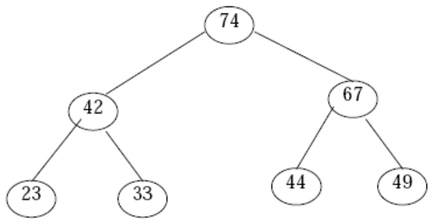
Write any example.

5. **Describe insertion in heap tree with example**.

Ans A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of the father. It can be sequentially represented as

$$A[j] <= A[(j – 1)/2]$$
$$\text{for } 0 <= [(j – 1)/2] < j <= n – 1$$

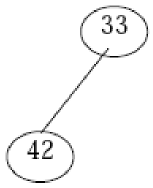
The root of the binary tree (*i.e.*, the first array element) holds the largest key in the heap. This type of heap is usually called descending heap or mere heap, as the path from the root node to a terminal node forms an ordered list of elements arranged in descending order. Following fig shows a heap



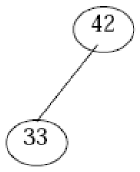
We can also define an ascending heap as an almost complete binary tree in which the value of each node is greater than or equal to the value of its father. This root node has the smallest element of the heap. This type of heap is also called min heap.

A heap H can be created from the following list of numbers 33, 42, 67, 23, 44, 49, 74 as illustrated below :
Step 1: Create a node to insert the first number (*i.e.,* 33) as shown Fig below

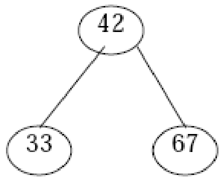
Step 2: Read the second element and add as the left child of 33 as shown Fig. 6.6.
Then restructure the heap if necessary



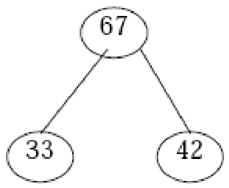
Compare the 42 with its parent 33, since newly added node (*i.e*., 42) is greater than
33 interchange node information as shown Fig



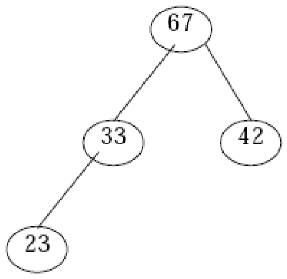
Step 3: Read the 3rd element and add as the right child of 42 as shown Fig. 6.8.
Then restructure the heap if necessary



Compare the 67 with its parent 42, since newly added node (*i.e*., 67) is greater than
42 interchange node information as shown Fig.

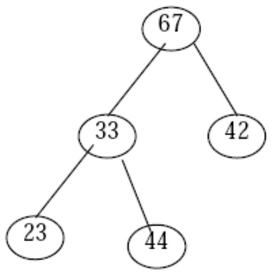


Step 4: Read the 4th element and add as the left child of 33 as shown below. Then restructure the heap if necessary.

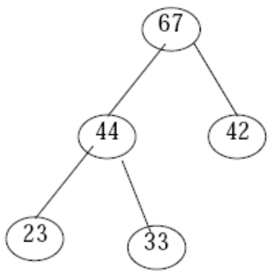


Since newly added node (i.e., 23) is less than its parent 33, no interchange.

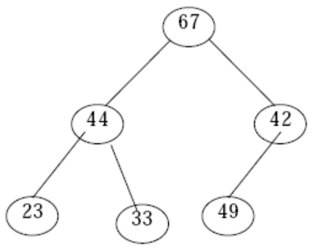
Step 5: Read the 5th element and add as the right child of 33 as shown below. Then restructure the heap if necessary



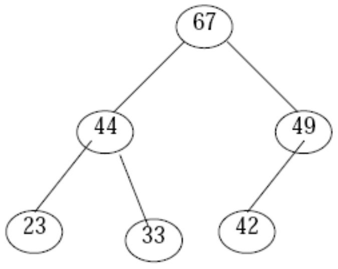
Compare the 44 with its parent 33, since newly added node (*i.e.*, 44) is greater than 33 interchange node information as shown



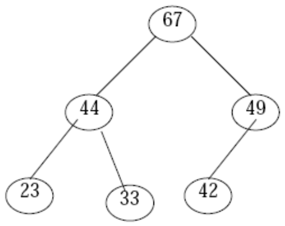
Step 6: Read the 6th element and add as the left child of 42 as shown below. Then restructure the heap if necessary.



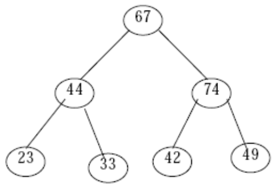
Compare the (newly added node) 49 with its parent 42, since newly added node (*i.e*.49) is greater than 42 interchange node information as shown Fig



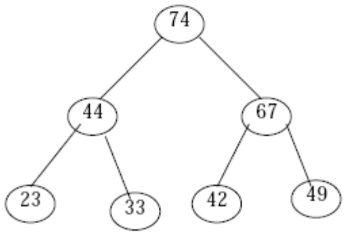
Step 7: Read the 7th element and add as the left child of 49 as shown below. Then restructure the heap if necessary.



Compare the (newly added node) 74 with its parent 49, since newly added node (i.e. 74) is greater than 49 interchange node information as shown below.



Compare the recently changed node 74 with its parent 67, since it is greater than 67 interchange node information as shown below



ALGORITHM

Let H be a heap with *n* elements stored in the array HA. This procedure will insert a new element *data* in H. LOC is the present location of the newly added node. And PAR denotes the location of the parent of the newly added node.

```
1. Input n elements in the heap H.
2. Add new node by incrementing the size of the heap H: n = n + 1 and LOC = n
3. Repeat step 4 to 7 while (LOC < 1)
4. PAR = LOC/2
5. If (data <= HA[PAR])
(a) HA[LOC] = data
(b) Exit
6. HA[LOC] = HA[PAR]
7. LOC = PAR
8. HA[1] = data
9. Exit
```

6. **Describe deletion operation in AVL tree with example**.

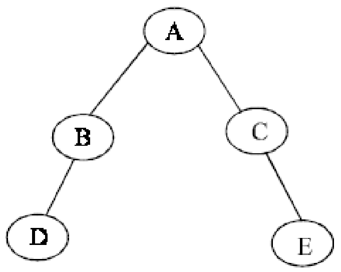
Ans. This algorithm was developed in 1962 by two Russian Mathematicians, G.M. Adel'son Vel'sky and E.M. Landis; here the tree is called AVL Tree. An AVL tree is a binary search tree in which the left and right sub tree of any node may differ in height by at most 1, and in which both the sub trees are themselves AVL Trees. Each node in the AVL Tree possesses any one of the following properties:

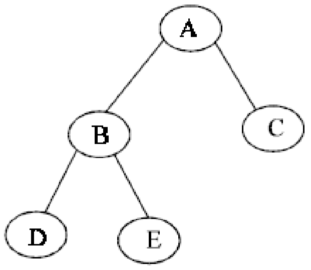
(*a*) A node is called left heavy, if the largest path in its left sub tree is one level larger than the largest path of its right sub tree.

(*b*) A node is called right heavy, if the largest path in its right sub tree is one level larger than the largest path of its left sub tree.

(*c*) The node is called balanced, if the largest paths in both the right and left sub trees are may differ in height by at most 1.

Example of AVL tree





This section gives an algorithm to delete a DATA of information from a binary search tree. First search and locate the node to be deleted. Then any one of the following conditions arises:

1. The node to be deleted has no children
2. The node has exactly one child (or sub tress, left or right sub tree)
3. The node has two children (or two sub tress, left and right sub tree)

Suppose the node to be deleted is N. If N has no children then simply delete the node and place its parent node by the NULL pointer.

If N has one child, check whether it is a right or left child. If it is a right child, then find the smallest element from the corresponding right sub tree. Then replace the smallest node information with the deleted node. If N has a left child, find the largest element from the corresponding left sub tree. Then replace the largest node information with the deleted node.

The same process is repeated if N has two children, *i.e.*, left and right child. Randomly select a child and find the small/large node and replace it with deleted node. NOTE that the tree that we get after deleting a node should also be a binary search tree.

Deleting a node can be illustrated with an example. Consider a binary search tree in Fig. 1. If we want to delete 75 from the tree, following steps are obtained:

Step 1: Assign the data to be deleted in DATA and NODE = ROOT.

Step 2: Compare the DATA with ROOT node, *i.e.*, NODE, information of the tree. Since (50 < 75)

NODE = NODE → RChild

Step 3: Compare DATA with NODE. Since (75 = 75) searching successful. Now we have located the data to be deleted, and delete the DATA. (See following fig 2)

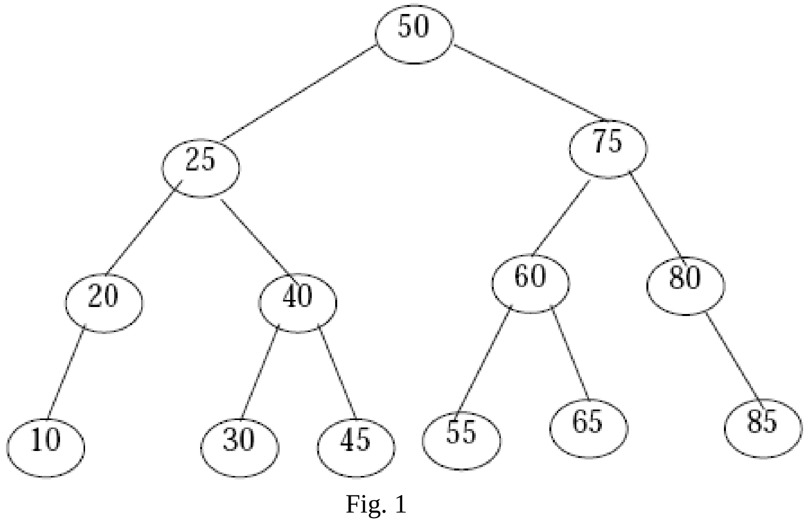


Fig. 1

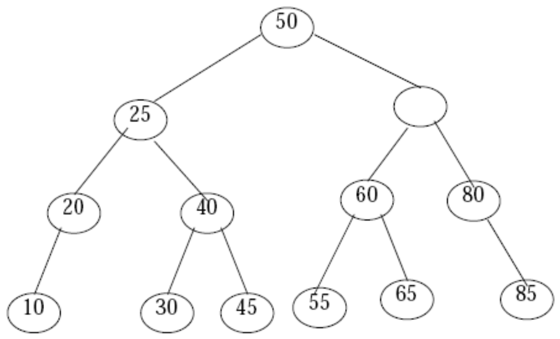


Fig. 2

Step 4: Since NODE (*i.e.*, node where value was 75) has both left and right child choose one. (Say Right Sub Tree) - If right sub tree is opted then we have to find the smallest node. But if left sub tree is opted then we have to find the largest node.

Step 5: Find the smallest element from the right sub tree (*i.e.,* 80) and replace the node with deleted node. (See Fig. 3)

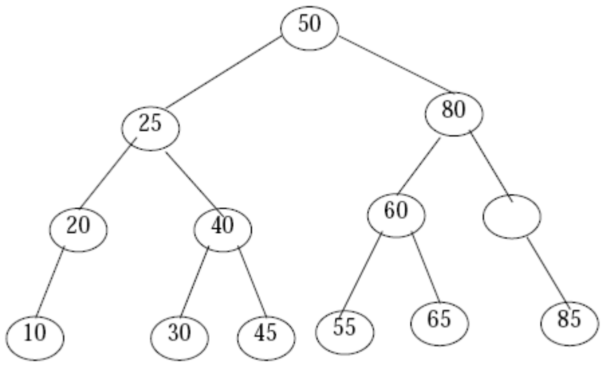


Fig. 3

Step 6: Again the (Node-> Rchild is not equal to NULL) find the smallest element from the right subtree which is 85 and replace it with emty nodes fig 4

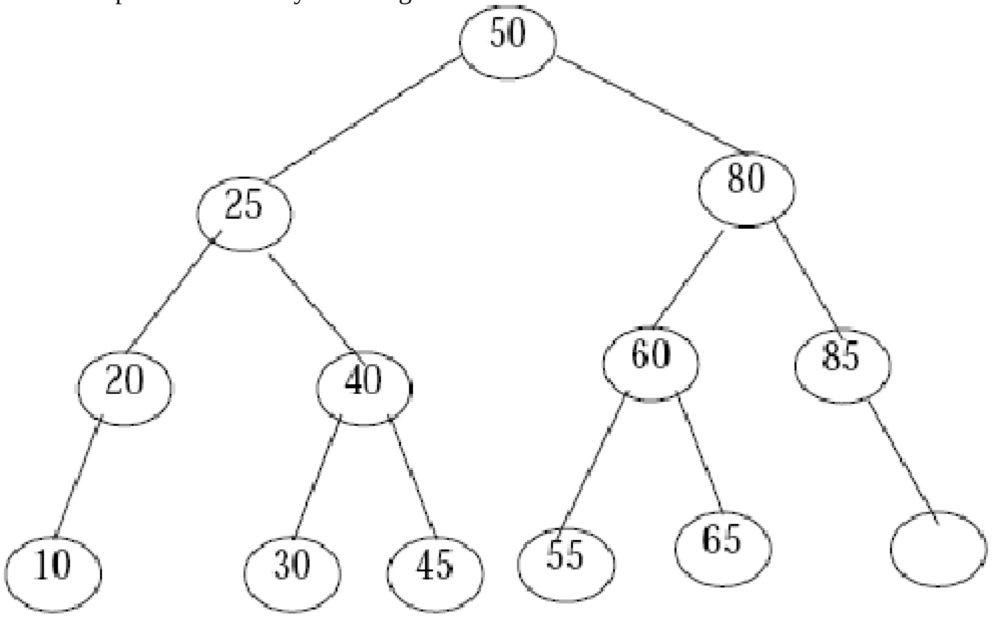


Fig 4

Step 7 : Since (Node-> Rchild= Node->Lchild=Null) delete the NODE and place NULL in the parent node. Fig. 5

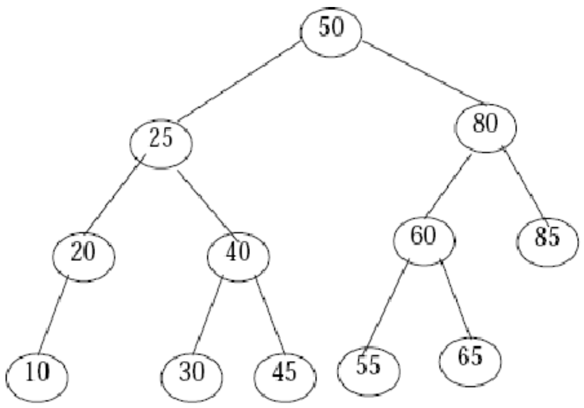


Fig 5

Step 8: Exit

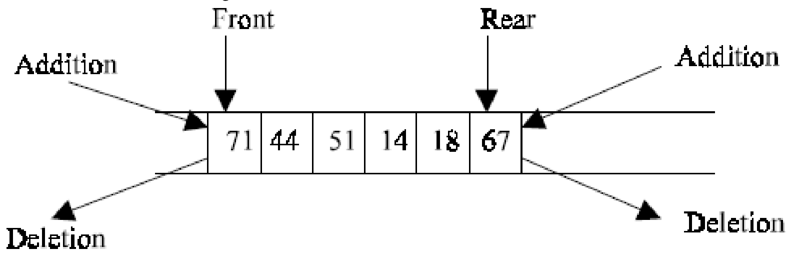
ALGORITHM

NODE is the current position of the tree, which is in under consideration. LOC is the place where node is to be replaced. DATA is the information of node to be deleted.

```
1. Find the location NODE of the DATA to be deleted.
2. If (NODE = NULL)
(a) Display "DATA is not in tree"
(b) Exit
3. If(NODE → Lchild = NULL)
(a) LOC = NODE
(b) NODE = NODE → RChild
4. If(NODE → RChild= =NULL)
(a) LOC = NODE
(b) NODE = NODE → LChild
5. If((NODE → Lchild not equal to NULL) && (NODE → Rchild not equal to NULL))
(a) LOC = NODE → RChild
6. While(LOC → Lchild not equal to NULL)
(a) LOC = LOC → Lchild
7. LOC → Lchild = NODE → Lchild
8. LOC → RChild= NODE → RChild
9. Exit
```

7. **Write the algorithm of deletion at rear end in dequeue with example.**

Ans. A deque is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation). ie; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.



There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

1. Input restricted deque
2. Output restricted deque

An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.

An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

The possible operation performed on deque is
1. Add an element at the rear end
2. Add an element at the front end
3. Delete an element from the front end
4. Delete an element from the rear end

Let Q be the array of MAX elements. *front* (or *left*) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. DATA will contain the element just deleted.

DELETE AN ELEMENT FROM THE RIGHT SIDE or REAR END or RIGHT MOST OF THE DE-QUEUE

```
1. If (left == – 1)
(a) Display “Queue Underflow”
(b) Exit
2. DATA = Q [right]
3. If (left == right)
(a) left = – 1
(b) right = – 1
4. Else
(a) if(right == 0)
(i) right = MAX-1
(b) else
(i) right = right-1
5. Exit
```

Example

User can take any list of element in de queue and show with the help of algorithm that how the elements are deleted from the rear end

8. Write the algorithm of prefix evaluation with example.

Ans. Following algorithm finds the RESULT of an arithmetic expression P written in prefix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

```
1. Scan P from right to left and repeat Steps 2 and 3 for each element of P until the P is finished.
2. If an operand is encountered, put it on STACK.
3. If an operator is encountered, then:
(a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
(b) Evaluate A operator B.
(c) Place the result on to the STACK.
4. Result equal to the top element on STACK.
5. Exit.
```

Example *3 2

1. Scan from right to left
2. Scanned symbol is 2 (an operand) , push onto stack
3. Next scanned symbol is 3 (an operand), push onto stack
4. Next scanned symbol is * (an operator), pop two topmost element from stack and evaluate with *, i.e. 3*2= 6.
5. Prefix expression is finished , so result is 6