TREES

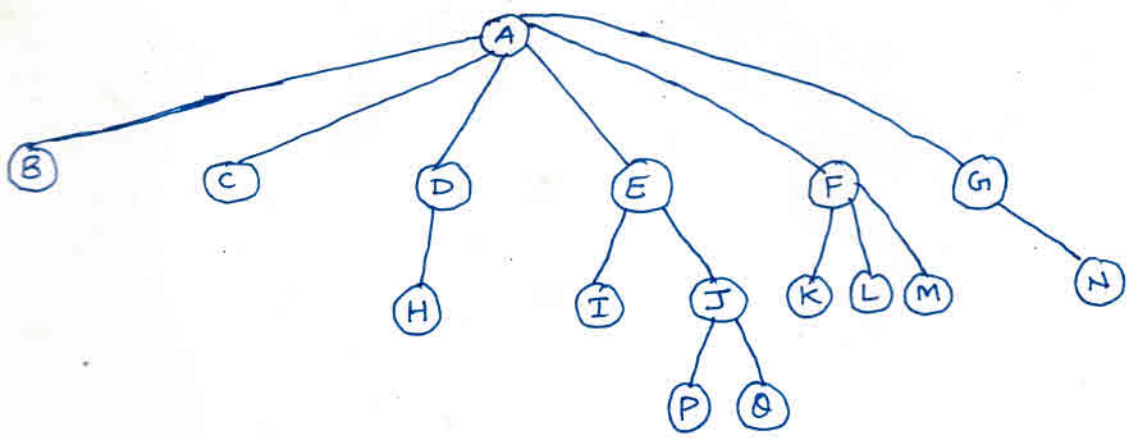
TREE ADT

A Tree is a data structure with a collection of nodes. A tree consits of a distinguished node 'r' called the root, and zero or more non-empty subtrees ($T_1, T_2 .. T_k$) each of whose roots are connected by a directed edge from 'r'.

The root of each subtree is said to be a child of 'r' and 'r' is the parent of each subtree root.

Example:-

A Tree



a) The Root is 'A'

b) Node 'F' has 'A' as a parent and K, L, and M as children.

c) Each node may have an arbitrary number of children, possibly zero

d) Nodes with no children are known as leaves.

e) Nodes with same parent are siblings.

STUDENTSFOCUS.COM

Degree :-

The number of nodes connected to a particular node is called the degree of that node. (For eg) in our example, the node containing data E has a degree 3.

The degree of a leaf node is always one.

Level :

The root node of the tree has level 0. The level of any other child node is one more than the level of its Parent. (For eg) node I is at level 2 and node P is at level 3.

Depth or Height :

The maximum level of any leaf node in the tree is called the depth (or) Height of a tree. (For eg) the depth (or) Height of the tree in our example is 3.

Note :-

The height of a node (n_i) is the length of the longest Path from n_i to a leaf. (For eg) the node (E) is at height 2.

STUDENTSFOCUS.COM

Degree :-

The number of nodes connected to a particular node is called the degree of that node. (For eg) in our example, the node containing data E has a degree 3.

The degree of a leaf node is always one.

Level :

The root node of the tree has level 0. The level of any other child node is one more than the level of its parent. (For eg) node I is at level 2 and node P is at level 3.

Depth or Height :

The maximum level of any leaf node in the tree is called the depth (or) Height of a tree. (For eg) the depth (or) Height of the tree in our example is 3.

Note :-

The height of a node (n_i) is the length of the longest path from n_i to a leaf. (For eg) the node (E) is at height 2.

STUDENTSFOCUS.COM

Implementation of Trees

a) one way to implement a tree would be to have in each node, besides its data, a pointer to each child of the node.

b) However, since the number of children per node can vary so greatly and is not known in advance, it might be infeasible to make the children direct links in the data structure, because there must be too much wasted space.

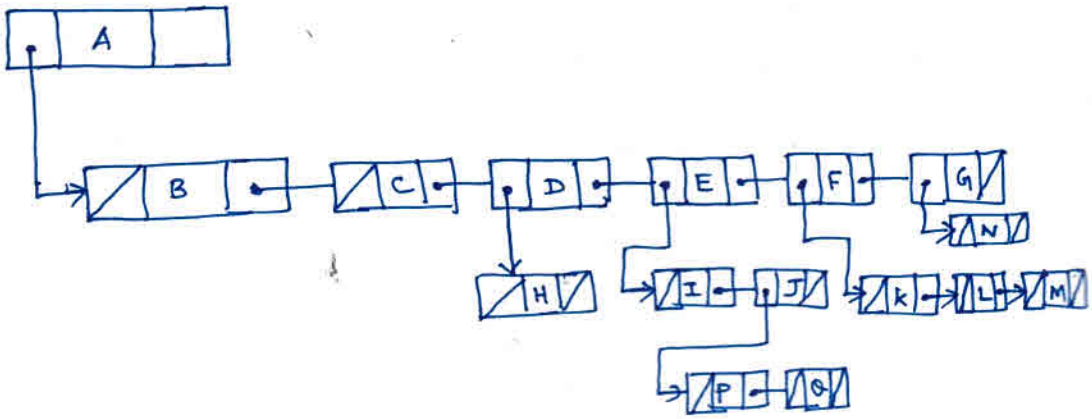
c) The solution is to keep the children of each node in a linked list of tree nodes.

The declaration is,

```
Struct Treenode
{
    ElementType  Element;
    Struct Treenode * firstchild;
    Struct Treenode * NextSibling;
}
```

STUDENTSFOCUS.COM

Example :-

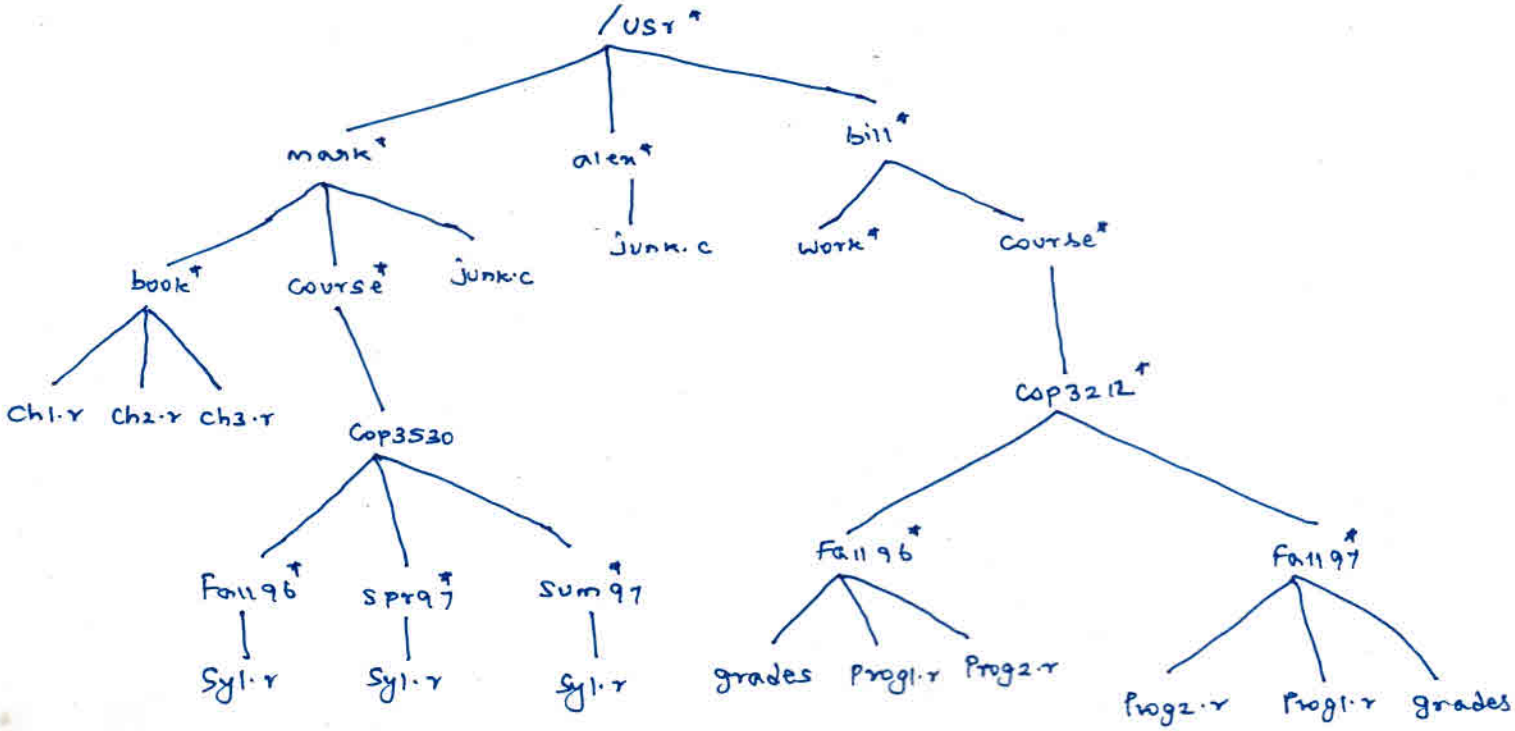


Tree Traversals with an Applications

a) one of the popular uses of tree is the directory structure in many common operating systems, including UNIX, VAX/VMS, and DOS.

Example:

Typical directory in the UNIX File system



STUDENTSFOCUS.COM

b) The root of this directory is /usr. (The asterik next to the name indicates that /usr is itsely a directory.)

c) This hierarchical file system is very popular, because it allows users to organize their data logically.

Routine to List a directory in a Hierarchical file system

```
Static void ListDir(DirectoryorFile D, int Depth)
{
    if (D is a legitimate entry)
    {
        PrintName(D, Depth);
        if (D is a Directory)
            For each child, c of D
                ListDir(c, Depth+1);
    }
}

Void ListDirectory(DirectoryorFile D)
{
    ListDir(D, 0);
}
```

Types of Tree Traversals:

1) Pre-order → Visiting Root, Left and Right child.

2) In order → Visiting Left child, Root and Right child.

3) Postorder → Visiting Left child, Right Child and Root.

STUDENTSFOCUS.COM

Preorder Traversal of a tree in our example:

```
/usr
mark
    book
      ch1.r
      ch2.r
      ch3.r
    course
        cop3520
            Fall96
                syl.r
            Spr97
                syl.r
            Sum97
                syl.r
    Junk.c
Alex
    junk.c
bill
    work
    course
        cop3212
            Fall96
                grades
                Prog1.r
                Prog2.r
            Fall 97
                Prog2.r
                Prog1.r
                grades
```

STUDENTSFOCUS.COM

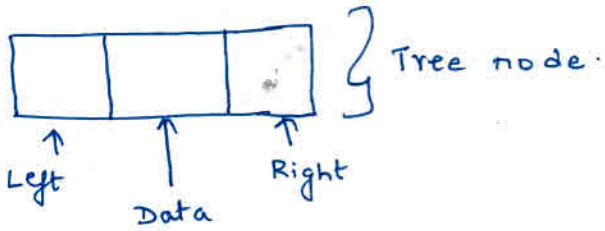
Binary Tree ADT :-

A Binary Tree is a tree in which no node can have more than two children.

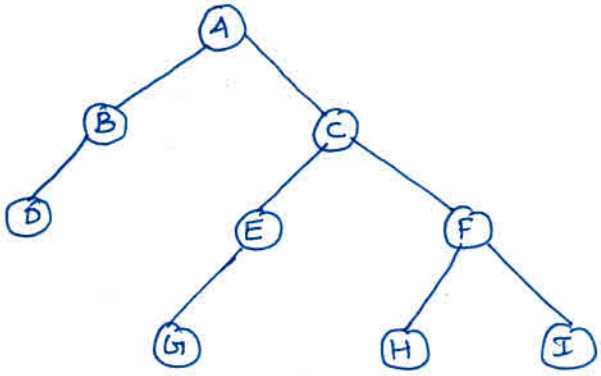
A Binary tree consits of a finite set of elements that can be ~~Partioned~~ Partitioned into three distincts subsets called the Root, the Left and the Right subtree.

If there are no elements in the binary tree it is called an empty binary tree.

Representation of a Binary Tree



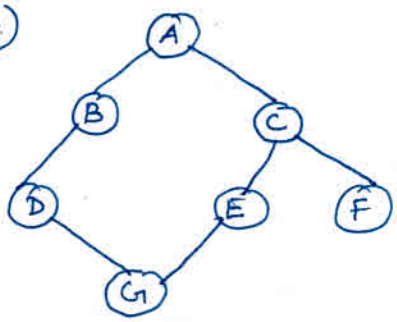
Example of a Binary Tree



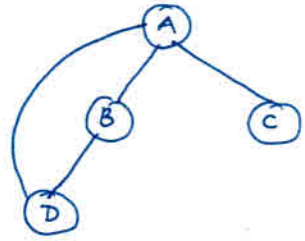
STUDENTSFOCUS.COM

Example of Trees that are not Binary trees

a)



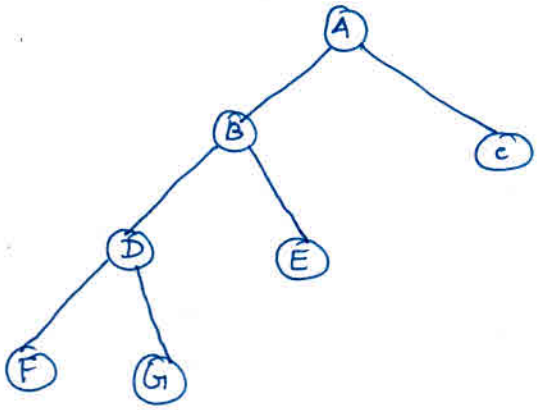
b)



Strictly Binary Tree

A Binary Tree is called a Strictly Binary Tree if every non-leaf node in a binary tree has non-empty left and right sub trees.

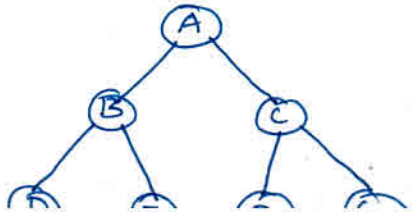
Example:-



Complete Binary Tree:-

A Strictly Binary tree in which all of the leaf nodes are at the same level is called a Complete binary tree.

Example:-



STUDENTSFOCUS.COM

Implementation of Binary Trees

a) Because a binary tree has at most two children, we can keep direct pointers to them.

b) The declaration of tree nodes is similar in structure to that for doubly linked lists. (i.e, a node in a doubly linked lists consits of the information field plus two pointers (left and Right)).

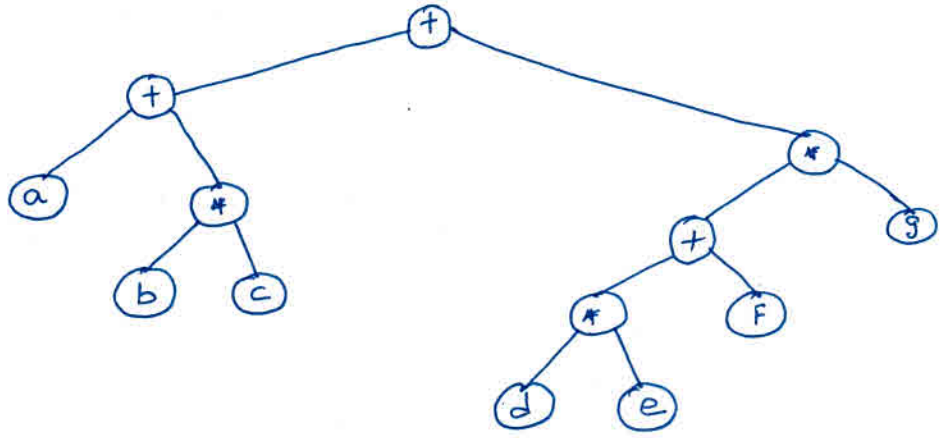
Example Node Declaration

```
Struct Treenode
{
    ElementType Element;
    Struct Treenode * Left;
    Struct Treenode * right;
};
```

Expression Trees construction

Example:

Expression Tree for $(a+b*c)+((d*e+f)*g)$



STUDENTSFOCUS.COM

a) The leaves of an expression tree are operands, such as constants or variable names, and the other nodes contain operators.

b) This tree happens to be binary, because all of the operations are binary. (It is also possible for a node to have only one child, as is the case with Unary Minus operator).

c) We can evaluate an expression tree, T, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.

d) In our example, the left subtree evaluates to a+(b*c) and the right subtree evaluates to ((d*e)+f)*g. The entire tree therefore represents (a+(b*c))+(((d*e)+F)*g)

e) We can produce an infix expression by recursively producing a parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression.

This general strategy (left, node, right) is known as an inorder traversal.

f) An alternate traversal strategy is to recursively print out the left subtree, the right subtree and then the operator. If we apply this above strategy to our example, the output is abc*+de*F+g*+, which is a postfix representation.

STUDENTSFOCUS.COM

g) A third traversal strategy is to print out the operator first and then recursively print out the left and right sub trees. The resulting expression for our example is, ++a*bc*+*defg. This is a prefix notation and the strategy is called as Pre-order traversal.

Constructing an Expression tree:-

Algorithm for Converting a postfix expression into an expression tree.

a) Read the input expression one at a time.

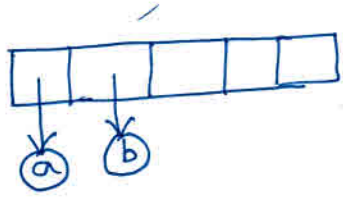
b) If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack.

c) If the symbol is an operator, we pop pointers to two trees T_1, and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack.

Example:-

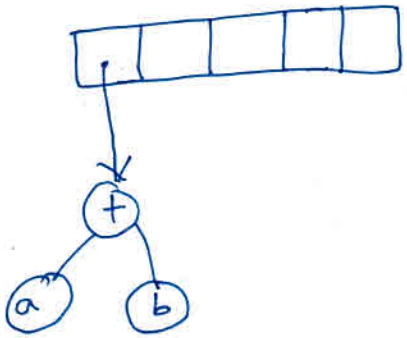
The input is: ab+cde+**

a) The first two symbols are operands, so we create one node trees and push pointers to them onto a stack,

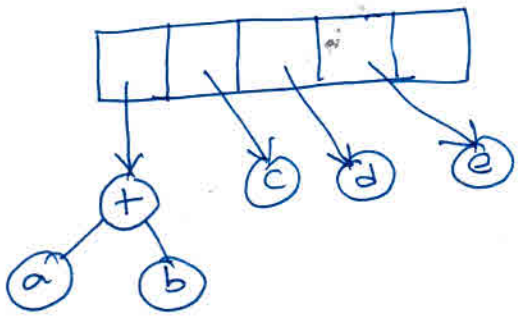


STUDENTSFOCUS.COM

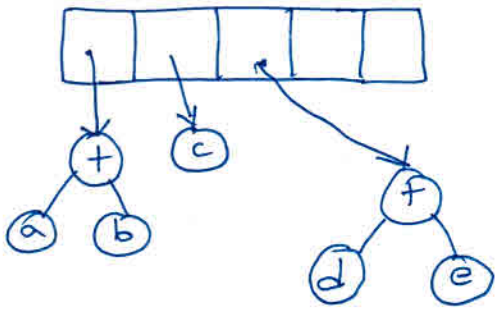
b) Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



c) Next, c, d, e are read, for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.

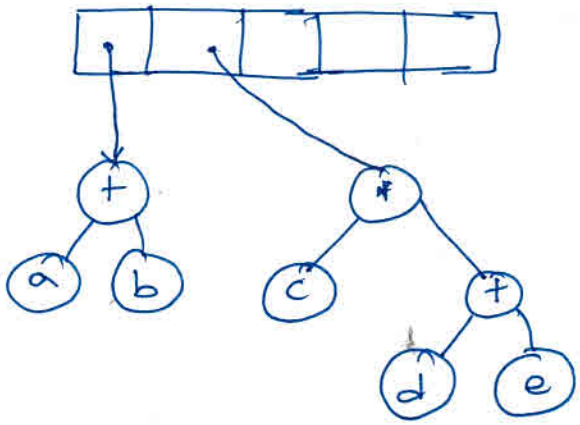


d) Next, '+' is read, so two trees are merged

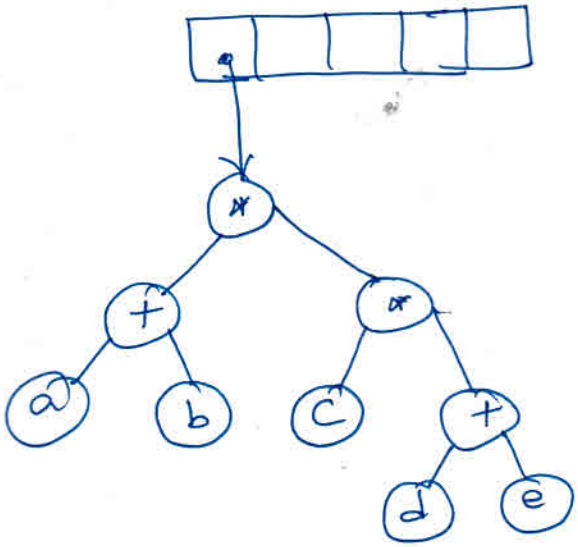


e) Next, '*' is read, so we pop two trees and form a new tree with a '*' as root.

STUDENTSFOCUS.COM



f) Finally, the last symbol is read. (i.e, '*'). two trees are merged, and a pointer to the final tree is left on the stack.

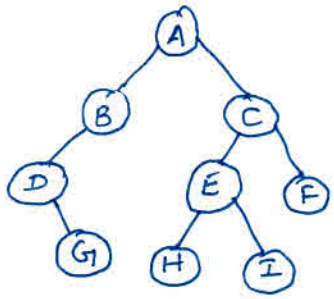


STUDENTSFOCUS.COM

Binary Tree Traversal Examples:

Example 1:

Consider the tree



Pre order Traversal:

1. Visit Root
2. Visit Left Subtree
3. Visit Right Subtree

Answer: A B D G C E H I F

In order Traversal:

1. Visit Left Subtree
2. Visit Root
3. Visit Right Subtree

Answer: D G B A H E I C F

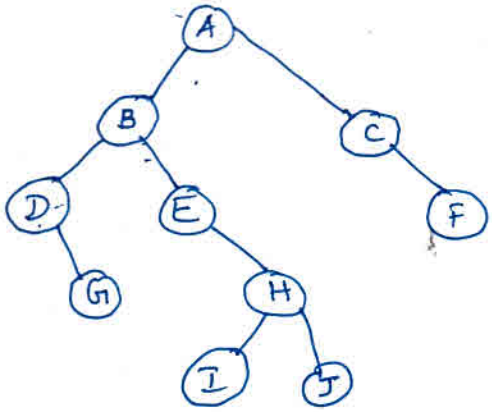
Post order Traversal

1. Visit left subtree
2. Visit Right subtree
3. Visit Root

Answer: G D B H I E F C A

STUDENTSFOCUS.COM

Binary Tree Traversal Example: 2



1. Pre-order Traversal:

ABDGEHIJCF

2. In-order Traversal

DGBEIHJACF

3. Post-order Traversal

GDIJHEBFCA

STUDENTSFOCUS.COM

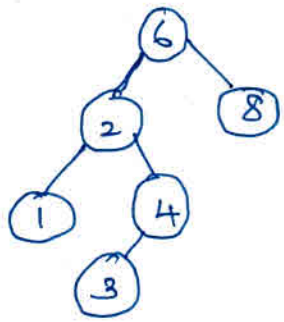
Applications of Binary Tree:-

Binary Search Tree ADT (BST)

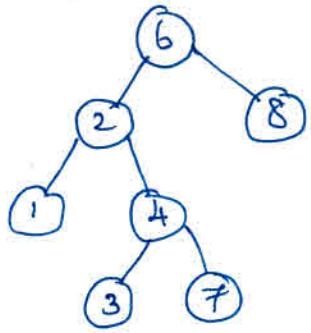
a) An important application of Binary trees is their use in searching.

b) The Property that makes a binary tree into a Binary Search Tree (BST) is that for every node, x, in the tree, the values of all the keys in its left sub tree are smaller than the key value in, x, and the values of all the keys in its right sub tree are larger than the key value in, x.

Examples:-



→ Binary Search Tree (BST)



→ Not a BST

STUDENTSFOCUS.COM

Operations in BST.

1. MakeEmpty:

This operation is mainly for initialization. Usually Recursive routines are written for Performing operations in a BST.

Routine to make an empty tree

```
MakeEmpty (SearchTree T)
{
    if (T != NULL)
    {
        MakeEmpty (T->Left);
        MakeEmpty (T->Right);
        Free (T);
    }
    return NULL;
}
```

2. Find

a) This operation generally requires returning a Pointer to the node in tree 'T' that has key 'X', or NULL if there is no such node.

b) If T is NULL, then we can just return NULL. otherwise, if the key stored at T is x, we can return T. otherwise, we make a recursive call on a subtree of T, either left or right, depending on the relationship of x to the

STUDENTSFOCUS.COM

Routine for Find operation for BST.

```
Find ( ElementType X, SearchTree T)
{
    if ( T == NULL)
     return NULL;
    if ( X < T -> Element )
     return find (X, T -> Left);
    else
    if ( X > T -> Element)
     return find ( X, T -> right);
    else
       return T;
}
```

3. Find min and Find max:

a) These routines return the position of the smallest and largest elements in the tree, respectively.

b) To perform a Findmin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

c) The Find max routine is the same, except that branching is to the right child.

STUDENTSFOCUS.COM

Routine of Findmin for BST

```
FindMin ( SearchTree T)
{
    if (T == NULL)
    return NULL;
    else
    if (T -> left == NULL)
     return T;
     else
       return { Findmin (T -> Left);
}
```

Routine of Findmax for BST

```
Findmax (SearchTree T)
{
    if (T == NULL)
    return NULL;
    else
    if (T -> right == NULL)
    return T;
    else
    return Findmax (T -> right);
}
```

STUDENTSFOCUS.COM

Non-Recursive implementation of Find Max for BST

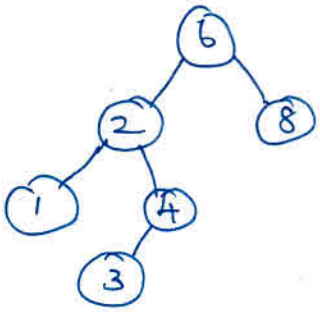
```
Findmax (SearchTree T)
{
    if (T != NULL)
    while (T->right != NULL)
     T = T->right;
     return T;
}
```

4) Insert.

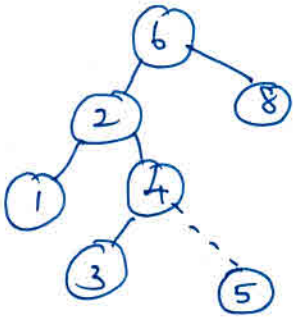
a) To insert 'x' into tree T, proceed down the tree as you would with a 'Find'. If 'x' is found do nothing. otherwise, insert 'x' at the last spot on the path traversed.

Example:

BST before inserting element 5



BST after inserting element 5



STUDENTSFOCUS.COM

Routine for Inserting an element into BST.

```
Insert (ElementType x, SearchTree T)
{
    if (T == NULL)
    {
      T = malloc (sizeof (struct TreeNode));
      if (T == NULL)
        FatalError ("out of space");
      else
      {
        T -> Element = x;
        T -> left = T -> right = NULL;
      }
    }
    if (x < T -> Element)
        T -> left = Insert (x, T -> left);
    else
    if (x > T -> element)
        T -> right = Insert (x, T -> right);

    return T;
}
```

Construct a BST by inserting the following sequence of numbers.

STUDENTSFOCUS.COM

5) Delete.

a) once we have found the node to be deleted, we need to consider several possibilities.

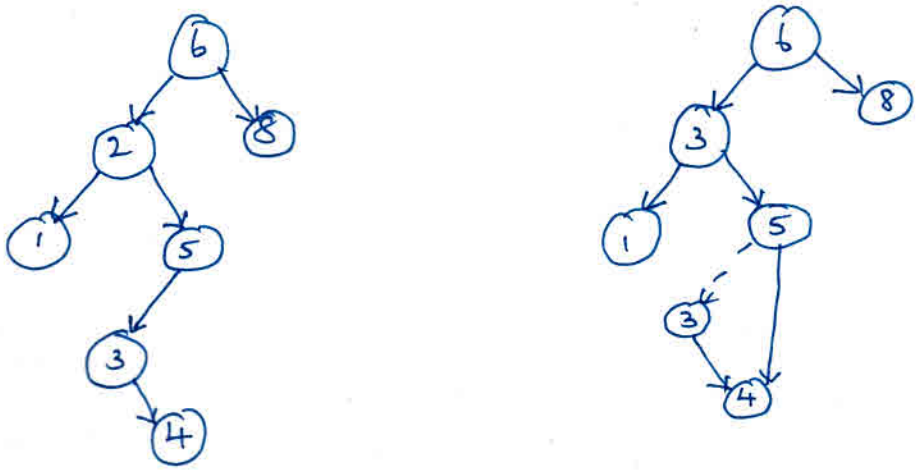
i) if the node is a leaf, it can be deleted immediately.

ii) if the node has one child, the node can be deleted after its parent adjust a pointer to bypass the node.

iii) If the node has two children, the general strategy is to replace the data of this node with the smallest data of the right subtree and recursively delete that node. Because the smallest node in the right subtree cannot have a left child, the second delete is an easy one.

Example:-

Deletion of Node(2) with two children



STUDENTSFOCUS.COM

Routine for Deleting an element in a BST.

```
Delete (Element Type x, SearchTree T)
{
    SearchTree Tmpcell;
    if (T==NULL)
        Error("Element Not Found");
    else
    if (x < T->Element)        /* Go left */
        T->left = Delete (x, T->left);
    else
    if (x > T->Element)        /* Go right */
        T->right = Delete (x, T->right);
    else   /* Found element to be deleted */
    if (T->left && T->right)   /* Two children */
    {
        /* Replace with smallest in right subtree */
        Tmpcell = T;
        if (T->left == NULL)
            T = T->right;
        else if (T->right == NULL)
            T = T->left;
        free (Tmpcell);
    }
    return T;
}
```

STUDENTSFOCUS.COM

AVL TREES (or Balanced Binary Trees)

a) An AVL (Adelson-Velskii, and Landis) tree is a binary search tree with a balanced condition.

b) An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1.

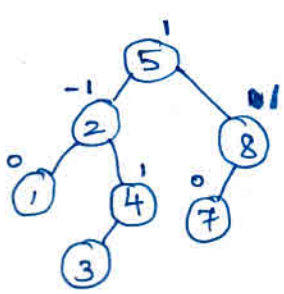
c) The value of "balancefactor" of any node is -1, 0, or 1. If it is other than these three values then the tree is not balanced. or it is not an AVL tree.

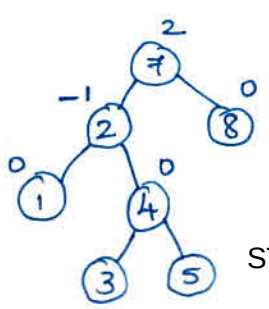
i) if the "balancefactor" of any node is -1, then the height of the right subtree of that node is one more than the height of its left sub-tree

ii) If the value of the "balancefactor" of any node is 0, then the height of its left and right subtree is exactly same.

iii) If the value of the "balancefactor" of any node is 1, then the height of the left subtree of that node is one more than the height of its right subtree.

Example:





STUDENTSFOCUS.COM

Note:-

The balance factor of any node is calculated by Subtracting the height of the right subtree of the node from the height of the left subtree.

d) All the operations that are done ~~using~~ in an Binary Search Tree (BST) are equally applicable and can be done in AVL Trees.

e) During insertion of a new node, care should be taken to ensure that the tree is Perfectly balanced after the insertion of new node.

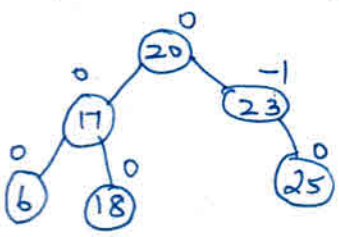
Insertion of a Node in an AVL Tree.

a) We can Insert a new node in an AVL tree by finding its appropriate position. But insertion of a new node involves certain overheads since the tree may become Unbalanced due to the increase in its height.

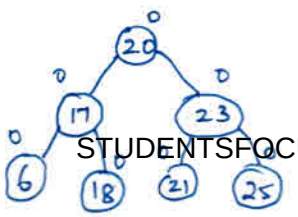
b) If the new node is inserted as a child of any non-leaf node then there will be no effect on its balance, as the height of the tree does not increase.

Example:-

Before Addition of 21



After Addition of 21

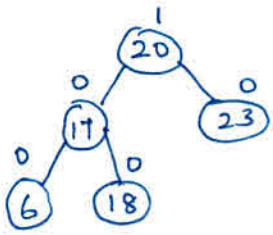


STUDENTSFOCUS.COM

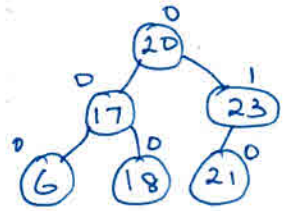
c) If the new node is inserted as a child of leaf node of subtree (left or right) of shorter height then there will be no effect on the balance of an AVL tree.

Example:

Before addition of 21



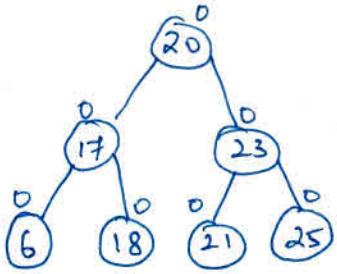
After addition of 21



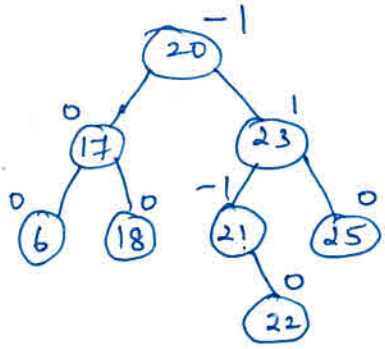
d) If the height of the left and the right sub tree is same then the insertion of a new node on any leaf node does not disturb the balance of an AVL tree. In this case, even if the height of a sub tree increases by one there will be no effect on the balance of the tree.

Example

Before addition of 22



After Addition of 22



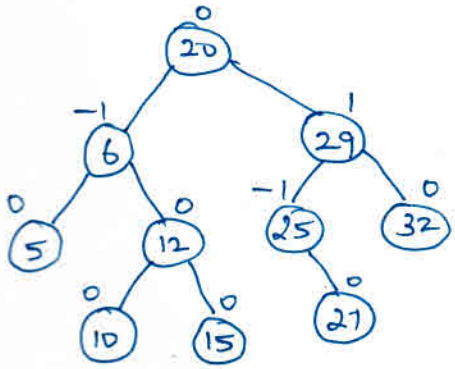
STUDENTSFOCUS.COM

e) If the new node is inserted as a child of leaf node then there is a possibility that the tree may become unbalanced. This depends upon whether the node is inserted to a left-sub tree or to a right sub tree, which in turn changes the balance factor of the nodes.

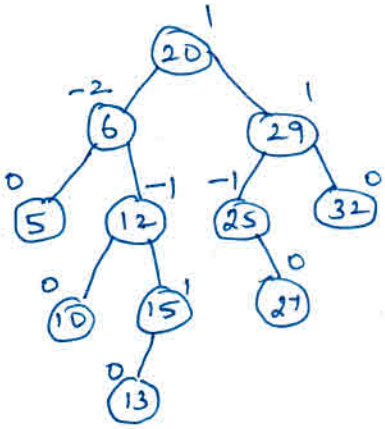
f) If the new node is inserted as a child of the leaf node of taller sub tree (left or right) then the AVL tree becomes unbalanced and the tree no longer remains an AVL Tree.

Example:-

Before addition of 13



After addition of 13



g) To re balance and make it an AVL tree the nodes need to be properly adjusted. So after the insertion of a new node the tree is traversed starting from the new node to the node whose balance factor is disturbed. The nodes are adjusted such a way that the resultant

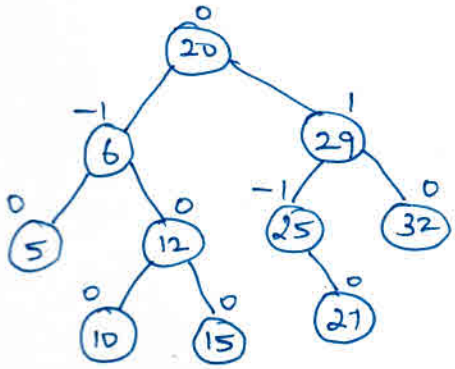
STUDENTSFOCUS.COM

e) If the new node is inserted as a child of leaf node then there is a possibility that the tree may become unbalanced. This depends upon whether the node is inserted to a left-sub tree or to a right sub tree, which in turn changes the balance factor of the nodes.

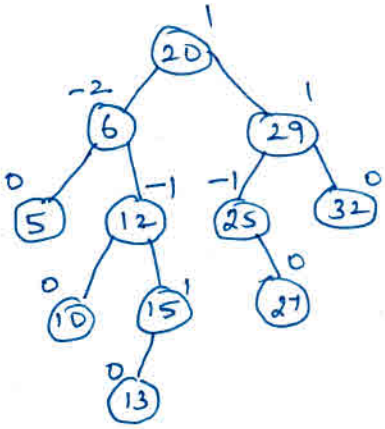
f) If the new node is inserted as a child of the leaf node of taller sub tree (left or right) then the AVL tree becomes unbalanced and the tree no longer remains an AVL Tree.

Example:-

Before addition of 13



After addition of 13



g) To re balance and make it an AVL tree the nodes need to be properly adjusted. So after the insertion of a new node the tree is traversed starting from the new node to the node whose balance factor is disturbed. The nodes are adjusted such a way that the resultant

STUDENTSFOCUS.COM

h. Let us call the node that must be rebalanced as 'α'.

The violation might occur in four cases.

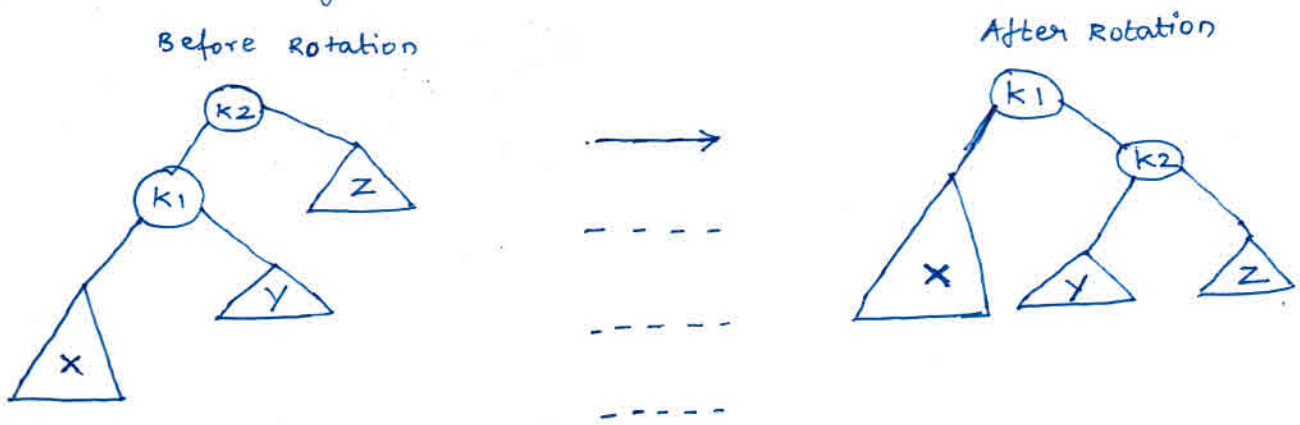
1) An insertion into the left subtree of the left child of 'α'
2) An insertion into the right subtree of the left child of 'α'
3) An insertion into the left subtree of the right child of 'α'
4) An insertion into the right subtree of the right child of 'α'

Cases 1 and 4 are mirror image symmetries with respect to α, Similarly cases 2 and 3 are mirror image symmetries.

i. The first case, in which the insertion occurs on the "outside" (i.e. left-left (or) right-right), is fixed by a single rotation of the tree. The second case, in which the insertion occurs on the "inside" (i.e. left-right (or) right-left) is handled by double rotation.

Single Rotation

a) Single Rotation to fix case 1



STUDENTSFOCUS.COM

a) Node k2 violates the AVL balance property, because its left subtree is two levels deeper than its right subtree.

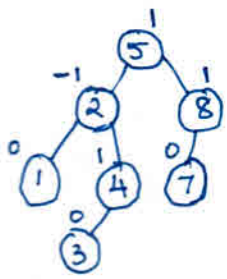
b) The situation depicted is the only possible case 1 scenario that allows k2 to satisfy the AVL property before an insertion but violate it afterwards.

c) To ideally rebalance the tree, we would like to move 'X' up a level and 'Z' down a level.

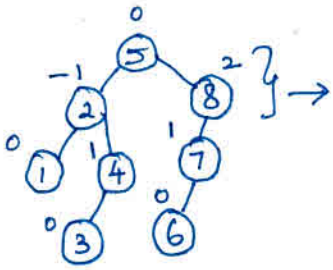
d) The Binary Search tree property tells us that in the original tree k2 > k1, so k2 becomes the right child of k1 in the new tree. 'X' and 'Z' remain as the left child of k1 and right child of k2, respectively. Subtree 'Y', which holds items that are between k1 and k2 in the original tree, can be placed as k2's left child in the new tree and satisfy all the ordering requirements.

Example:-

AVL Tree Before inserting 6

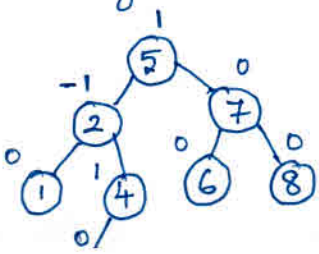


~~AVL~~ Tree After inserting 6



→ Violating AVL Property.
↓
Case 1 Scenario
↓
Solution Single Rotation.

After Single Rotation

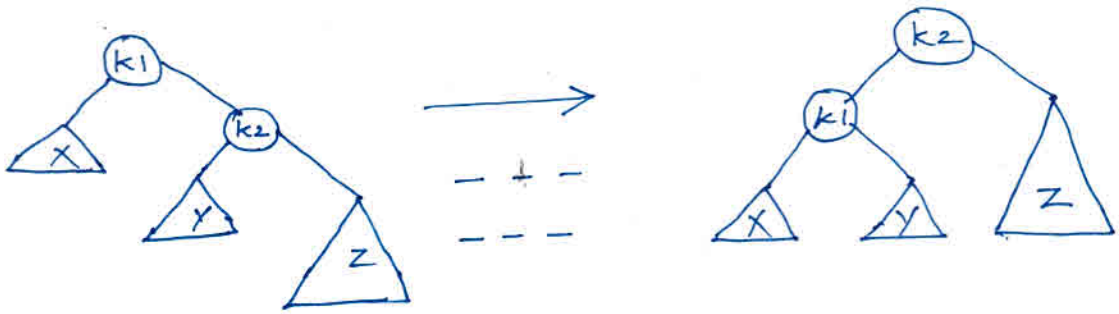


k2 = 8
k1 = 7

STUDENTSFOCUS.COM

Single Rotation to fix Case 4:

Before Rotation



Example:-

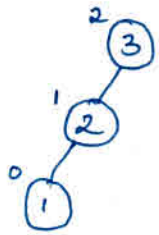
We start with an initially empty AVL tree and insert keys 3, 2, 1 and then 4, 5, 6, 7 in sequential order.

1.

→ After inserting 3

2.

→ After inserting 2

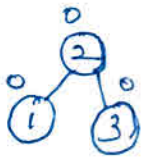
3. 

→ After inserting 1 [case 1 scenario]

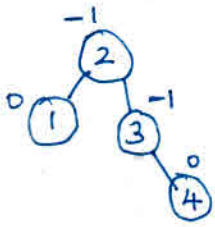
Violates AVL property.

Single Rotation is necessary.

$k_2 = 3$, $k_1 = 2$

4. 

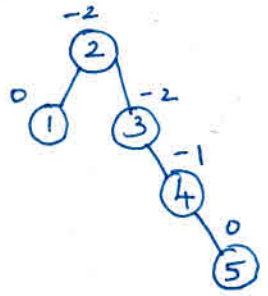
→ After Single Rotation.

5. 

→ After inserting 4.

STUDENTSFOCUS.COM

6.



→ After inserting 5. [case 4 scenario]

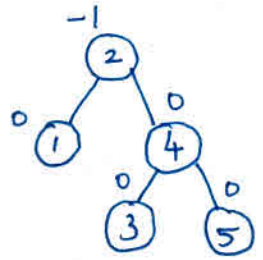
Violates AVL Property at Node 3 and Node 2.

First fix Node 3 violation using Single Rotation.

~~k2 = 3~~, ~~k1 = 4~~

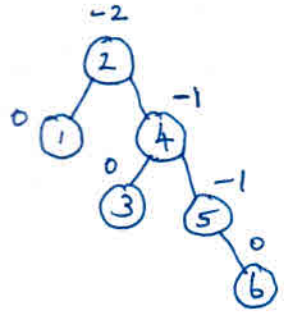
$k_1 = 3$ $k_2 = 4$

7



→ After Single Rotation

8

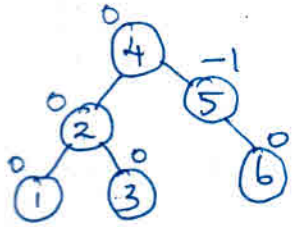


→ After inserting 6. [case 4 scenario]

Violates AVL property at root node 2.

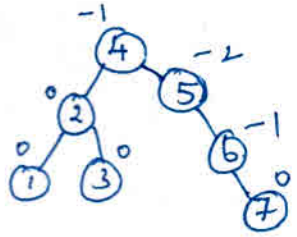
$k_2 = 4$, $k_1 = 2$

9.



→ After Single Rotation.

10.

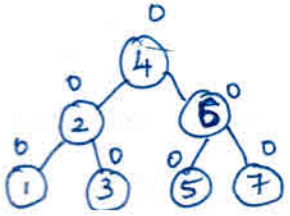


→ After inserting 7. [case 4 scenario]

Violates AVL Property at node 5

$k_2 = 5$, $k_1 = 5$

11.



→ After Single Rotation.

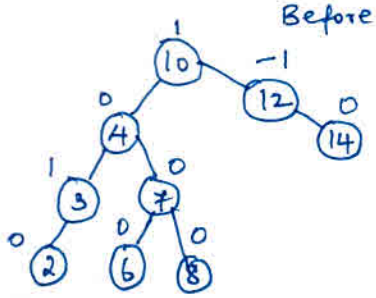
STUDENTSFOCUS.COM

Double Rotation.

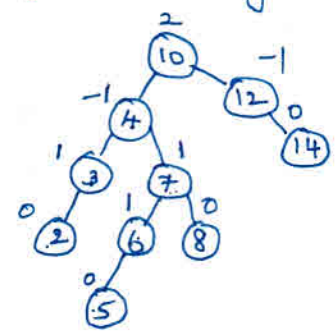
a) The algorithm described for single rotation cannot solve case 2 and case 3 problem.

Example:-

Before inserting 5



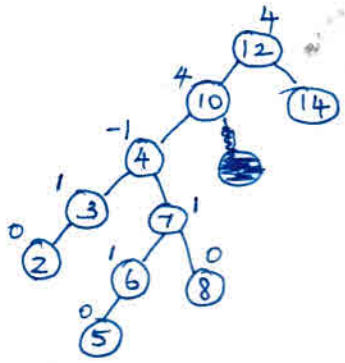
After inserting 5



AVL property violated at root node.

We will now try to solve this violation using single rotation.

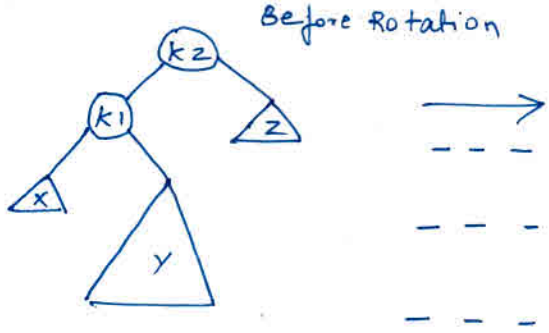
$k_2 = 12, \quad k_1 = 10$



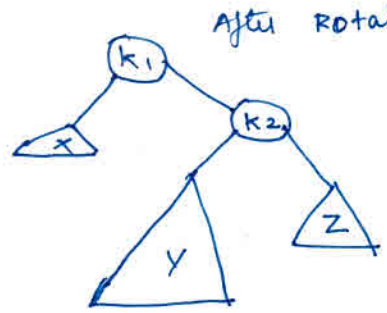
Still the tree is not balanced after performing single rotation.

Single Rotation fails to fix case: 2 (and) case (3)

Before Rotation



After Rotation

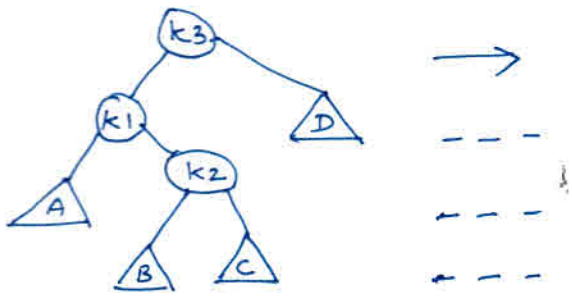


b) The problem is that subtree 'Y' is too deep, and a single rotation does not make it any less deep. Double Rotation is the solution.

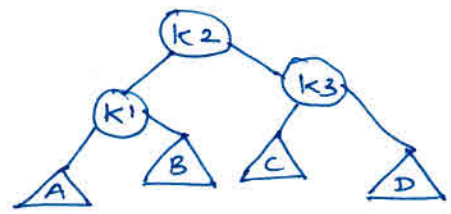
STUDENTSFOCUS.COM

Left - Right double rotation to fix case : 2

Before Rotation

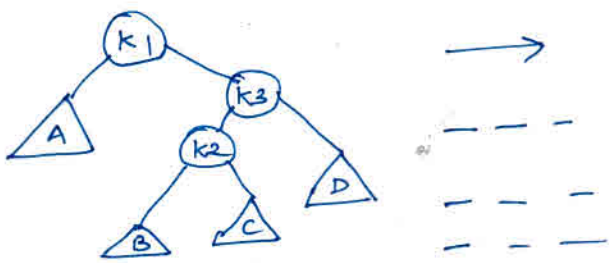


After Rotation

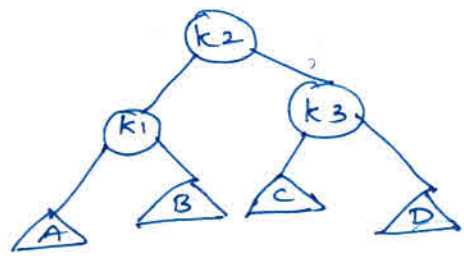


Right - Left double rotation to fix case : 3

Before Rotation



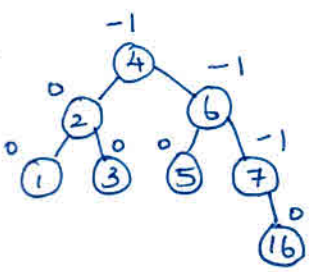
After Rotation



Example:-

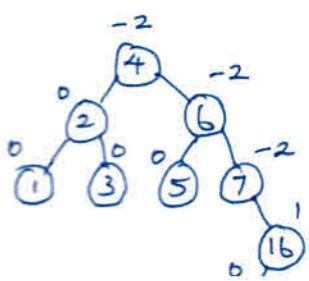
We will continue our previous example by inserting keys 16, 15, 14, 13, 12, 11, 10, 8 and then 9.

1.



After inserting 16.

2.



After inserting 15.

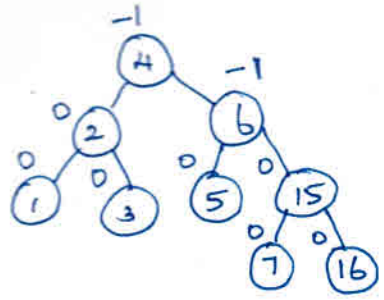
Violates AVL property at node 7

[case - 3 scenario Right-left Double rotation]

STUDENTSFOCUS.COM

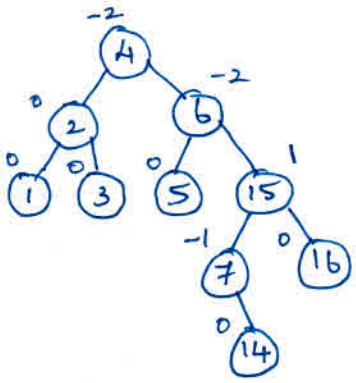
k1 = 7

3.



→ After Double Rotation

4.



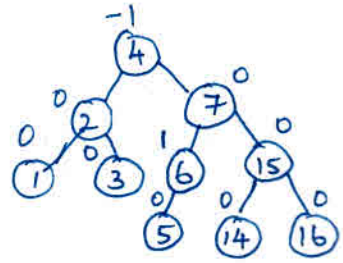
→ After inserting 14

Violates AVL property at node 6.

[Case 2: Righ-left Double Rotation]

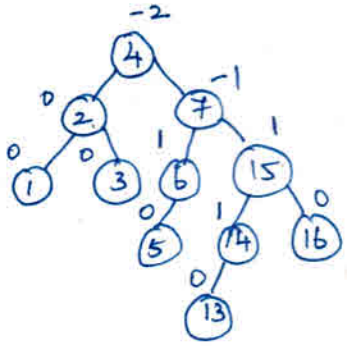
$k_1 = 6$, $k_2 = 7$, $k_3 = 15$

5



→ After Double Rotation.

6



→ After inserting 13.

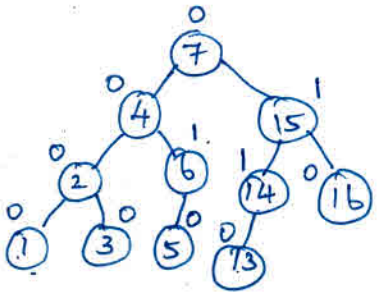
Violates AVL property at root node 4

Since 13 is not between 4 and 7

Single rotation will work.

[Case 4 Scenario.] $k_2 = 7$, $k_1 = 4$

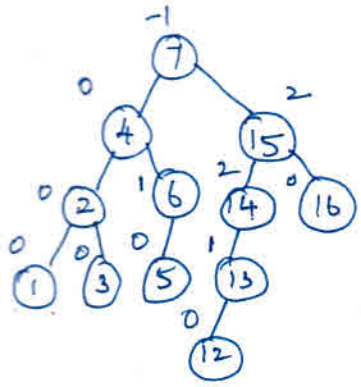
7



→ After Single Rotation

STUDENTSFOCUS.COM

8



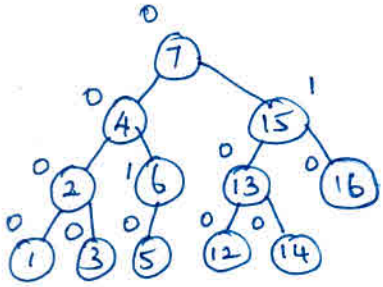
After inserting 12.

AVL Property violates at node 14

[case 1 scenario]

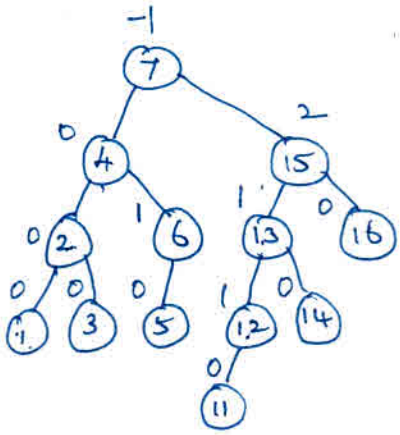
k2 = 14, k1 = 13

9



After Single Rotation.

10.



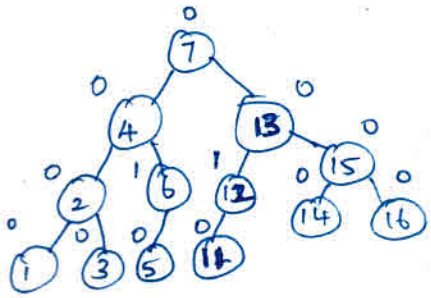
After inserting 11

AVL Property violates at node 15

[Case:1 scenario]

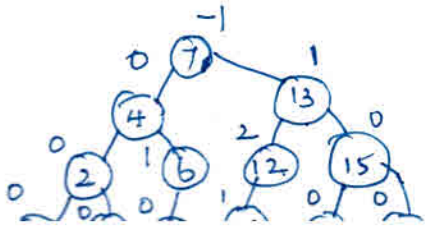
k2 = 15, k1 = 13

11



After Single Rotation.

12



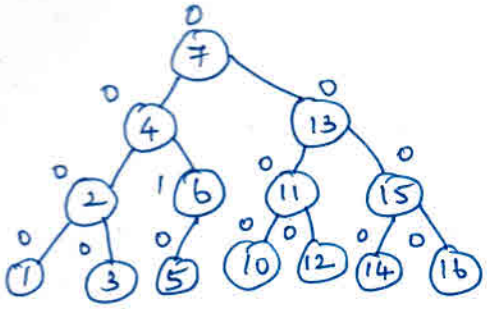
After inserting 10

AVL Property violates at node 12

[case 1: scenario]

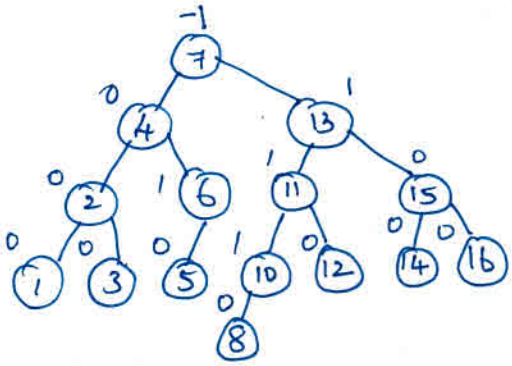
STUDENTSFOCUS.COM

13.



→ After Single Rotation

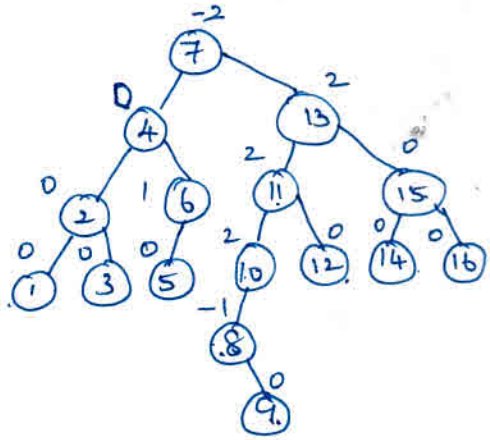
14.



→ After inserting 8.

No Problem.

15.



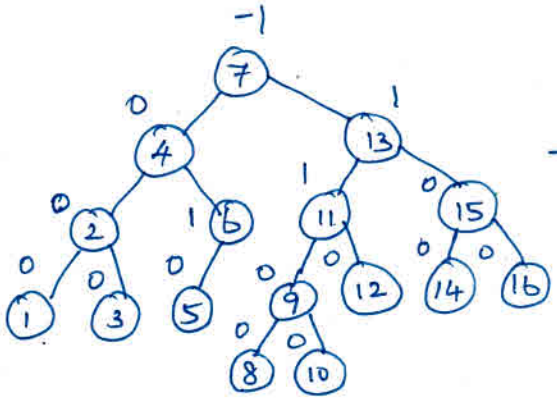
→ After inserting 9

AVL Property Violates at node 10

[Case: 3 Scenario
Left - Right Double Rotation]

$k_3 = 10$, $k_1 = 8$, $k_2 = 9$

16



→ After Double Rotation

STUDENTSFOCUS.COM

Function to compute height of an AVL node

```
Height (Position P)
{
    if (P == NULL)
      return -1;
    else
      return P -> Height;
}
```

Routine for inserting an element into an AVL Tree

```
Insert (ElementType x, AVLTree T)
{
    if (T == NULL)
    {
      T = malloc (sizeof (struct AVLNode));
      if (T == NULL)
        fatalError("out of space");
      else
      {
        T -> Element = x;  T -> Height = 0;
        T -> left = T -> Right = NULL;
      }
    }
  else
    if (x < T -> Element)
    {
      T -> left = Insert (x, T -> left);
      if (Height (T -> left) - Height (T -> right) == 2)
        if (x < T -> left -> Element)
          SingleRotateWithLeft (T);
```

STUDENTSFOCUS.COM

```
else
  if (x > T -> Element)
  {
    T -> right = Insert (x, T -> right);
    if (Height (T -> right) - Height (T -> left) == 2)
      if ( x > T -> right -> Element)
      T = SingleRotate with Right (T);
      else
      T = DoubleRotate with Right (T);
    }
  T -> Height = Max( Height (T -> left), Height (T -> Right)) + 1;
  return T;
}
```

Routine for SingleRotate with Left

```
SingleRotate with Left ( Position k2)
{ Position k1;
  k1 = k2 -> left;
  k2 -> left = k1 -> right;
  k1 -> right = k2;
  k2 -> Height = max( Height (k2 -> left), Height (k2 -> right)) + 1;
  k1 -> Height = Max( Height (k1 -> left), Height (k2 -> Height)) + 1;
  return k1;
}
```

Routine for DoubleRotate with Left

```
DoubleRotate with Left (Position k1)
{
  k3 -> left = SingleRotate with Right (k3 -> left);
  return SingleRotate with Left (k3);
}
```

STUDENTSFOCUS.COM

Priority Queues (Binary Heaps)

a) In a Priority Queue all the elements are assigned some Priority. The order in which the elements could be deleted or processed from the priority queue depends upon this priority.

b) Operating systems uses Priority Queue for scheduling jobs where jobs with higher priority are processed first.

c) A Priority Queue is a data structure that allows at least the following two operations,

i) Insert

ii) DeleteMin (which finds, returns and removes the minimum element in the priority Queue).

Binary Heaps

a) The Implementation of priority Queue is commonly done using binary heap.

b) Heaps have two properties, namely

i) A structure Property.

ii) Heap order property.

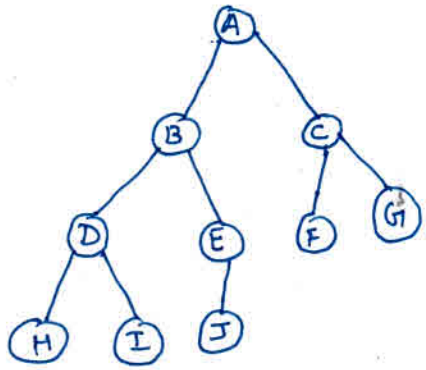
Structure Property

a) A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right, such a tree is known as a complete binary tree.

STUDENTSFOCUS.COM

Example:

A Complete Binary Tree



b) A Complete binary tree of height 'h' has between 2^h and $2^{h+1}-1$ nodes. This implies that height of a Complete binary tree is $\log N$.

c) An important observation is that because a Complete binary tree is so regular, it can be represented in an array and no pointers are necessary.

Array Implementation of Complete binary tree.

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

d) For any element in array position, i, the left child is in Position 2i, the right child is in the cell after the left child (2i+1), and the parent is in position (i/2).

e) The 0th element is called the Sentinel value that is a maximum value and is not the node of the tree. This Sentinel value is required because while addition of

STUDENTSFOCUS.COM

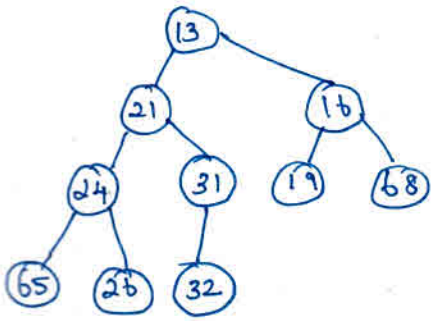
Heap order Property

a) The property that allows operations to be performed quickly is the heap order property.

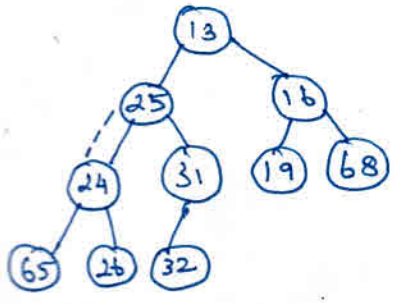
b) Since we want to be able to find the minimum quickly, it makes sense that the smallest element should be at the root. If we consider that any subtree should also be a heap, then any node should be smaller than all of its descendants.

c) In a heap, for every node x, the key in the parent of x, is smaller than (or equal to) the key in x, with the exception of the root (which has no parent).

Example:



→ This complete tree is a heap



→ This complete tree is not a heap. because key 24 appears as a child of key 25 which is a violation of heap order property.

STUDENTSFOCUS.COM

Basic Heap operations:

Insert

a) To insert an element 'x' into the heap, we create a hole in the next available location, since otherwise the tree will not be complete.

b) If 'x' can be placed in the hole without violating heap order, then we have done insertion successfully.

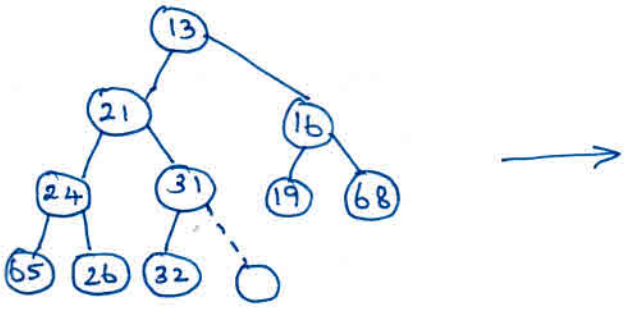
c) Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root.

d) we continue this process until 'x' can be placed in the hole.

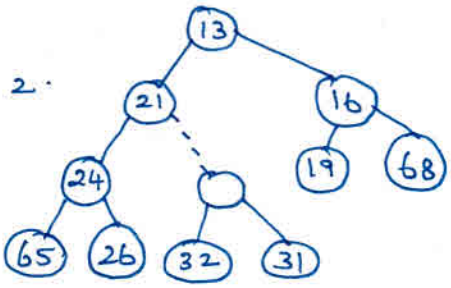
Example:

Attempting to insert 14, creating the hole, and bubbling up hole

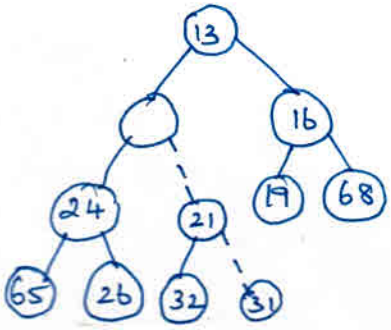
1.



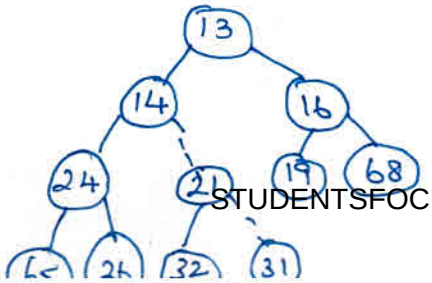
2.



3.



4.



STUDENTSFOCUS.COM

e) This general strategy is known as a "percolate up", the new element is percolated up the heap until the correct location is found.

Routine to insert an element into a binary tree.

```
Void Insert ( ElementType x, PriorityQueue H)
{
    int i;
    if (isfull(H))
    {
      Error ("Priority Queue is full");
      return;
    }
    for (i = ++H->size ; H->Elements[i/2] > x; i /= 2)
        H->Elements[i] = H->Elements[i/2];
        H->Elements[i] = x;
}
```

```
Struct Heap
{
  int capacity;
  int size;
  ElementType *Elements;
};
```

Delete Minimum:-

a) Delete minimum is handled in a similar manner as insertions. Finding minimum is easy but the hard part is removing it.

b) When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows

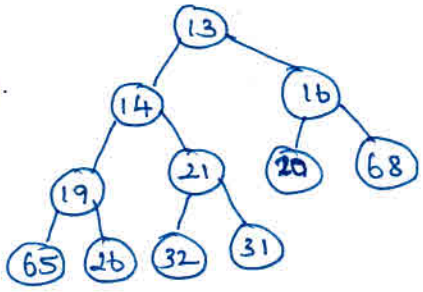
STUDENTSFOCUS.COM

c) If 'x' can be placed in the hole, then we have done successfully. This is unlikely in most cases, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. we repeat this step until 'x' can be placed in the hole. Thus, our action is to place 'x' in its correct spot along a path from the root containing 'minimum' children.

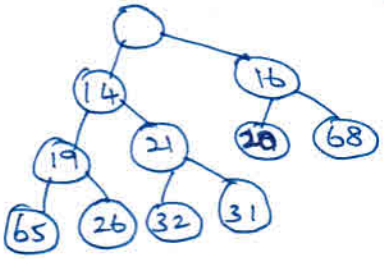
Example:-

13 is removed, so hole is created.

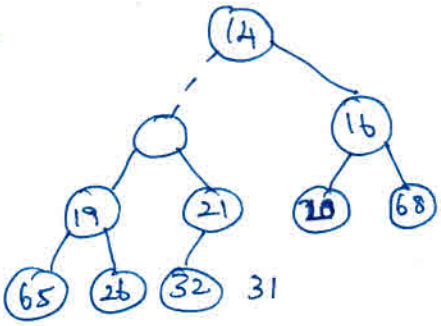
1.



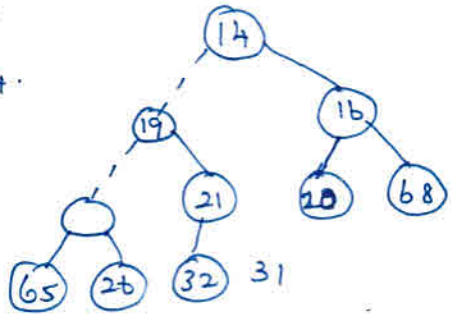
2.



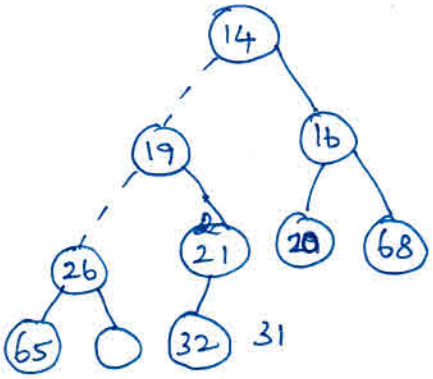
3.



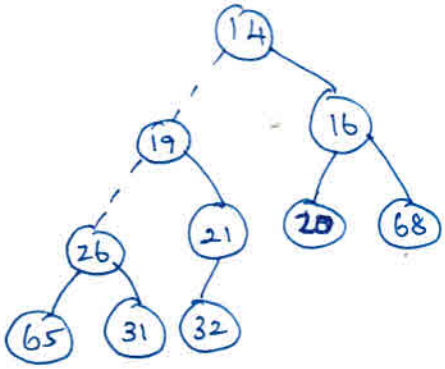
4.



5.



6.



STUDENTSFOCUS.COM

d) This general strategy is known as a "Percolate down".

Routine to perform DeleteMin in a Binary Heap.

```
ElementType DeleteMin (PriorityQueue H)
{
    int i, Child;
    ElementType MinElement, LastElement;
    if (Isempty (H))
    {
      Error ("Priority Queue is Empty");
      return H -> Elements [0];
    }
    MinElement = H -> Elements [1];
    LastElement = H -> Elements [H -> size --];
    for (i=1; i*2 <= H -> size; i=child)
    {
            /* Find Smaller child /*
            Child = i*2;
            if (child != H -> size && H -> Elements [child+1] <
                                      H -> Elements [child])

            child ++;
            /* Percolate one level */
          if (LastElement > H -> Elements [child])
            H -> Elements [i] = H -> Elements [child];
          else
            break;
    }
    H -> Elements [i] = LastElement;
    return MinElement;
}
```

STUDENTSFOCUS.COM

Other Heap operations:

Decrease key:

The Decrease key (P, Δ, H) operation lowers the value of the key at Position 'P' by a positive amount Δ. Since this might violate the heap order, it must be fixed by a percolate up.

Increase key:

The increase key (P, Δ, H) operation increases the value of the key at Position 'P' by a positive amount Δ. This is done with a Percolate down.

Delete

The Delete (P, H) operation removes the node at Position 'P' from the heap. This is done by first Performing Decreasekey (P, ∞, H) and then Performing DeleteMin (H).

Build Heap:

a) The BuildHeap (H) operation takes as input 'N' keys and Places them into an empty heap. The general algorithm is to Place the 'N' keys in any order, maintaining the structure Property.

b) Then, if PercolateDown (i) Percolates down from node i, Perform the following algorithm to create a heap order Property.

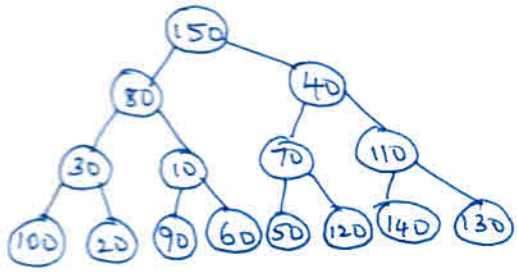
```
for (i = N/2; i > 0; i--)
    PercolateDown (i);
```

STUDENTSFOCUS.COM

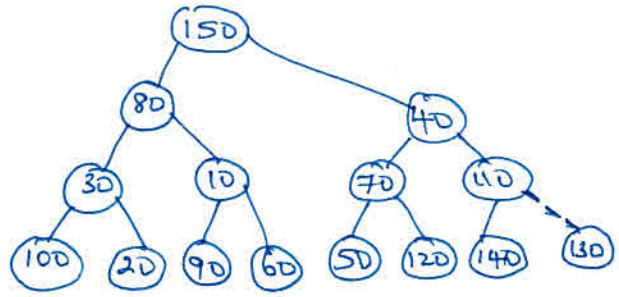
c) Each dashed line corresponds to two Comparisons,

*Example:

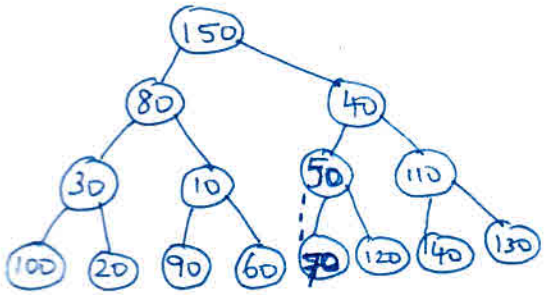
Initial Heap



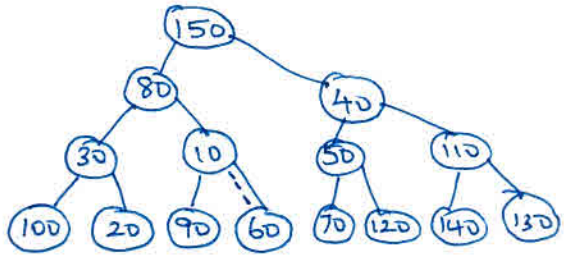
After PercolateDown (7)



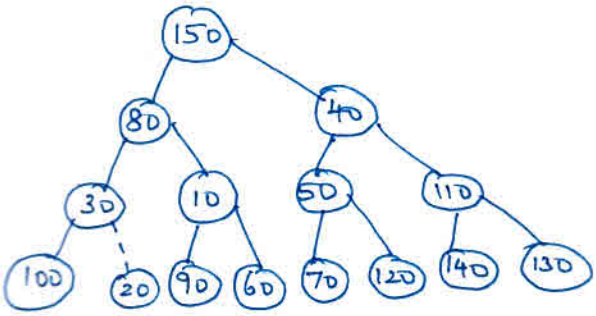
After PercolateDown (6)



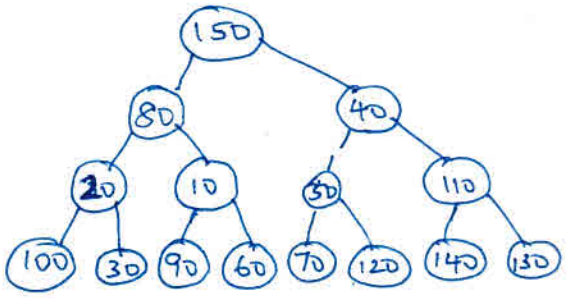
After PercolateDown (5)



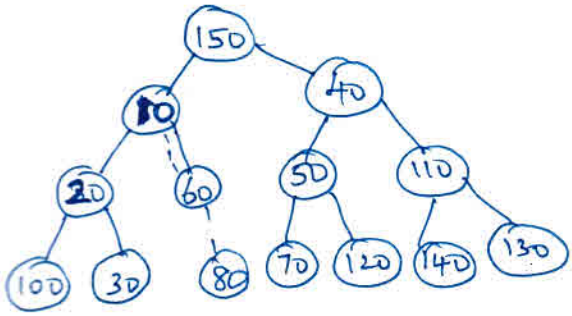
After PercolateDown (4)



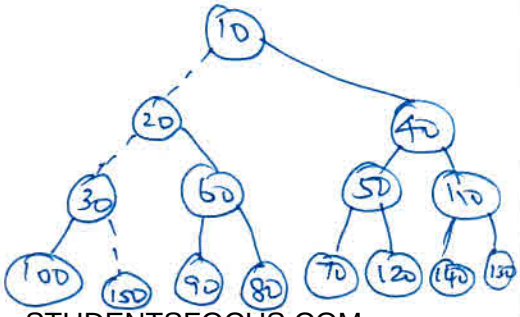
After PercolateDown (3)



After PercolateDown (2)



After PercolateDown (1)



STUDENTSFOCUS.COM

Theorem:-

For the Perfect Binary Tree of height 'h' Containing $2^{h+1}-1$ nodes, the sum of the heights of the nodes is $2^{h+1}-1-(h+1)$

Proof:

It is easy to see that a perfect binary tree consits of 1-node at height 'h', 2-nodes at height $h-1$, 2^2 nodes at height $h-2$, and in general 2^i nodes at height $h-i$.

The sum of the heights of all the nodes is then,

$$S = \sum_{i=0}^{h} 2^i (h-i)$$

$$S = h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \cdots + 2^{h-1}(1) \rightarrow (1)$$

Multiplying equation (1) by 2, gives

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \cdots + 2^h(1) \rightarrow (2)$$

we subtract these two equations and obtain the equation,

(1) - (2) $\quad S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h$

$S = (2^{h+1}-1) - (h+1)$ which proves the theorem.

STUDENTSFOCUS.COM