JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self Belief | Self Discipline | Self Respect"

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CS8391 / DATA STRUCTURES

(2017 Regulation)

Year/Semester: II / 03

Prepared by

Dr. K. Tamilarasi,

Professor / Dept. of CSE.

SYLLABUS

CS8391 DATA STRUCTURES L T P C 3 0 0 3

UNIT I LINEAR DATA STRUCTURES – LIST 9

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation —singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operations (Insertion, Deletion, Merge, Traversal).

UNIT II LINEAR DATA STRUCTURES – STACKS, QUEUES 9

Stack ADT – Operations - Applications - Evaluating arithmetic expressions- Conversion of Infix to postfix expression - Queue ADT – Operations - Circular Queue – Priority Queue - deQueue – applications of queues.

UNIT III NON LINEAR DATA STRUCTURES – TREES 9

Tree ADT – tree traversals - Binary Tree ADT – expression trees – applications of trees – binary search tree ADT –Threaded Binary Trees- AVL Trees – B-Tree - B+ Tree - Heap – Applications of heap.

UNIT IV NON LINEAR DATA STRUCTURES - GRAPHS 9

Definition – Representation of Graph – Types of graph - Breadth-first traversal - Depth-first traversal – Topological Sort – Bi-connectivity – Cut vertex – Euler circuits – Applications of graphs.

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES 9

Searching- Linear Search - Binary Search. Sorting - Bubble sort - Selection sort - Insertion sort - Shell sort – Radix sort. Hashing- Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

TOTAL: 45 PERIODS

TEXT BOOKS:

1. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, 2nd Edition, Pearson Education,1997.
2. ReemaThareja, “Data Structures Using C”, Second Edition , Oxford University Press, 2011

REFERENCES:

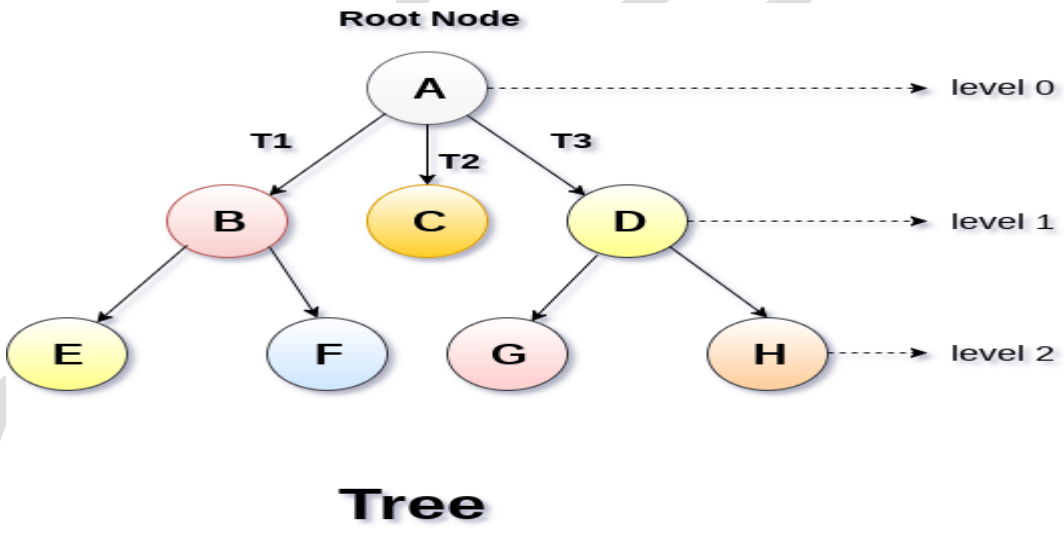
1. Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein, “Introduction to Algorithms", Second Edition, Mcgraw Hill, 2002.
2. Aho, Hopcroft and Ullman, “Data Structures and Algorithms”, Pearson Education,1983.
3. Stephen G. Kochan, “Programming in C”, 3rd edition, Pearson Education.

UNIT III NON LINEAR DATA STRUCTURES – TREES

Tree ADT – tree traversals - Binary Tree ADT – expression trees – applications of trees – binary search tree ADT –Threaded Binary Trees- AVL Trees – B-Tree - B+ Tree - Heap – Applications of heap.

Tree

- A Tree is a **nonlinear data structure** consists of one or more data nodes where **one node is designated as the root** of the tree while the **remaining nodes are called as the children** of the root.
- In a **general tree**, **A node can have any number of children nodes,** but it can have only a single parent.
- The following image shows a tree, where the node A is the root node of the tree while the other nodes can be seen as the children of A.

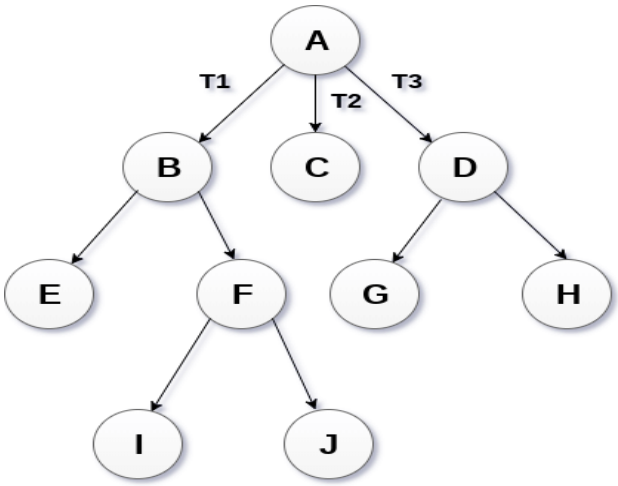


Tree

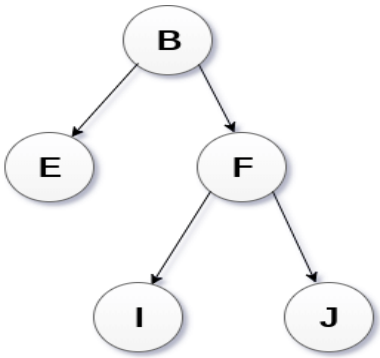
Basic terminology

1. **Root Node**:- The root node is the topmost node in the tree hierarchy **which doesn't have any parent**.
2. **Leaf Node or external node :-** The node of tree, which **doesn't have any child node**, is called leaf node.
3. **Level :** . Root node of the tree is always present at level 0. The level of the any other child node is one more than that of the parent. Ex : level of Node F is 2.

4. **Depth or Height :** The maximum level of any node in the tree is called Depth or Height. For the above tree Depth or height is 2.
5. **Degree:** The number of nodes connected to a particular node is called Degree. For Ex: Degree of node B is 3. Degree of node E is 1.
6. **Subtree or forest**: The nodes other than the root node are partitioned into the non empty sets called subtree.
   If we delete the root and the edges connecting the root to the nodes at level 1, then we get the Subtree with root as the node at level 1.
   The tree T1, T2 and T3 is called sub-trees of the root node.

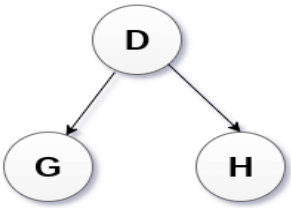


General Tree



Forest 1

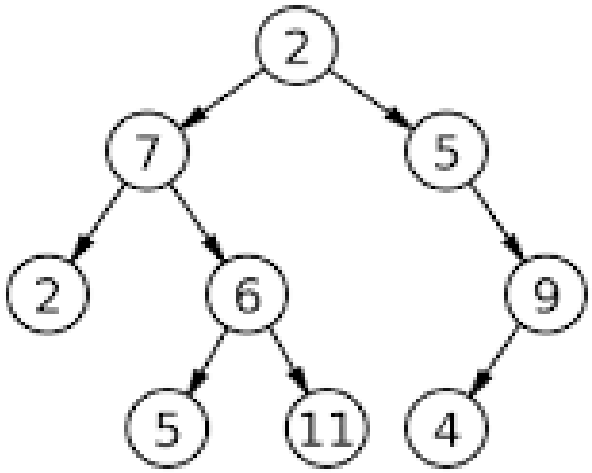
Forest 2



Forest 3

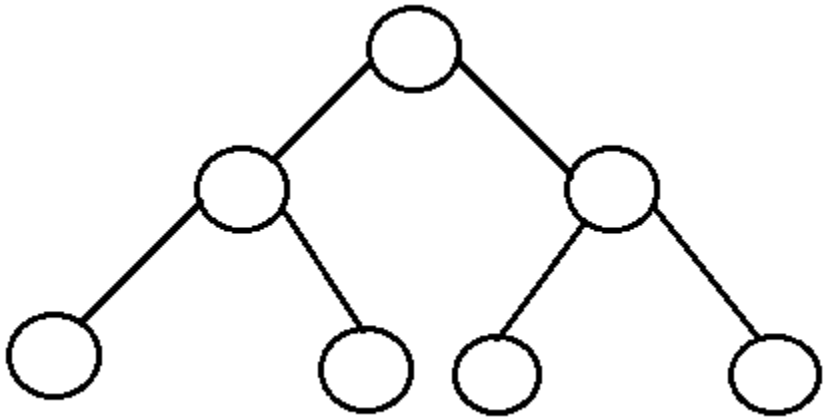
Binary Tree ADT

A binary tree is a tree in which no node can have more than two children. In a binary tree, the topmost element is called root node, and each node has 0, 1 or at the most 2 children.



Types of Binary Tree

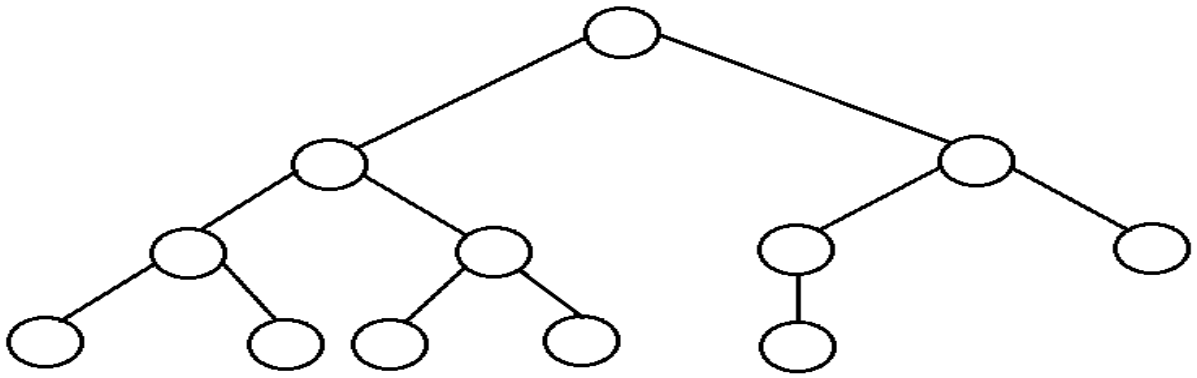
Full binary tree: It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



- A perfect full binary tree with l leaves has n = 2l-1 nodes.
- In perfect full binary tree, $l = 2^h$ and $n = 2^{h+1} - 1$ where, n is number of nodes, h is height of tree and l is number of leaf nodes.

  Ex: Leaf node = 4, then no of nodes n= 2*4-1 = 7.

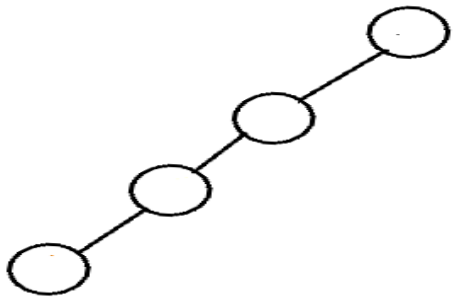
Complete binary tree: It is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



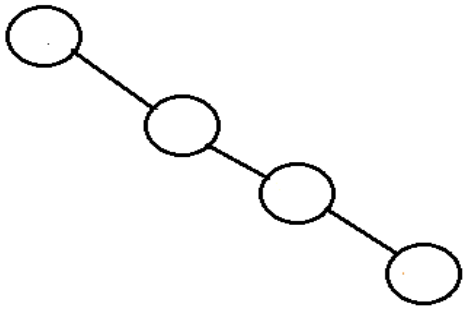
The number of internal nodes in a complete binary tree of n nodes is floor(n/2).

Degenarate tree: It is a tree is where each parent node has only one child node. It behaves like a linked list. There are 2 types, Left Skewed and Right Skewed binary tree.

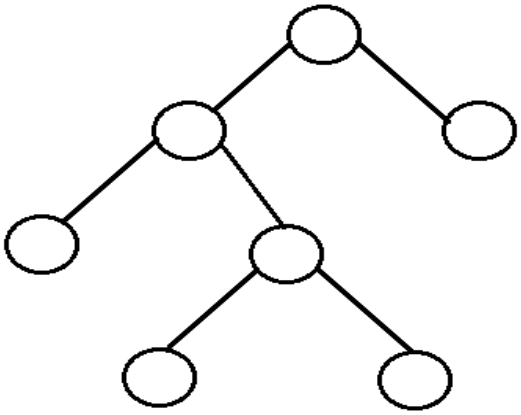
1. **Left Skewed Trees**: Grows only Left Side



2. Right Skewed Trees: Grows only Right Side



Strictly binary tree: It is a tree in which every node in the tree has either 0 or 2 children.



Representation of Binary Tree:

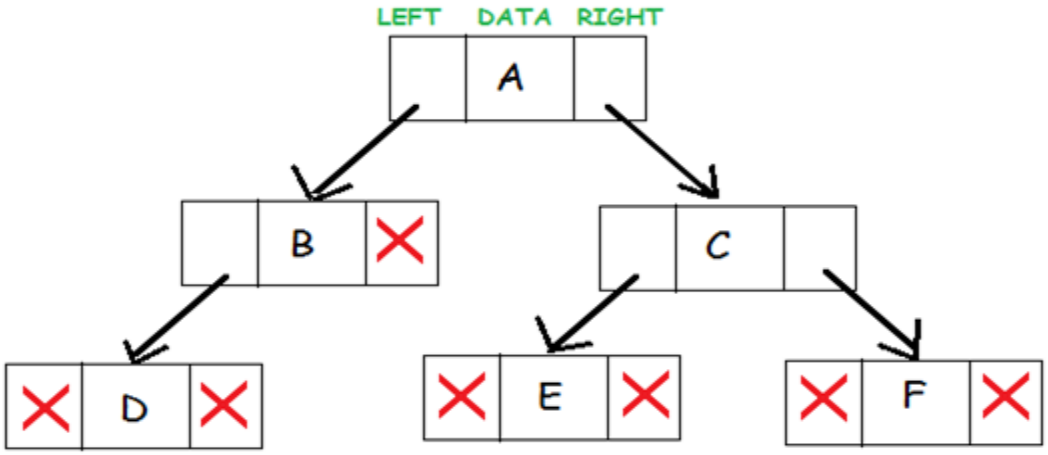
1. Linked List Representation
2. Array Representation

Linked List Representation: In Linked List Representation, Each node contains three components:

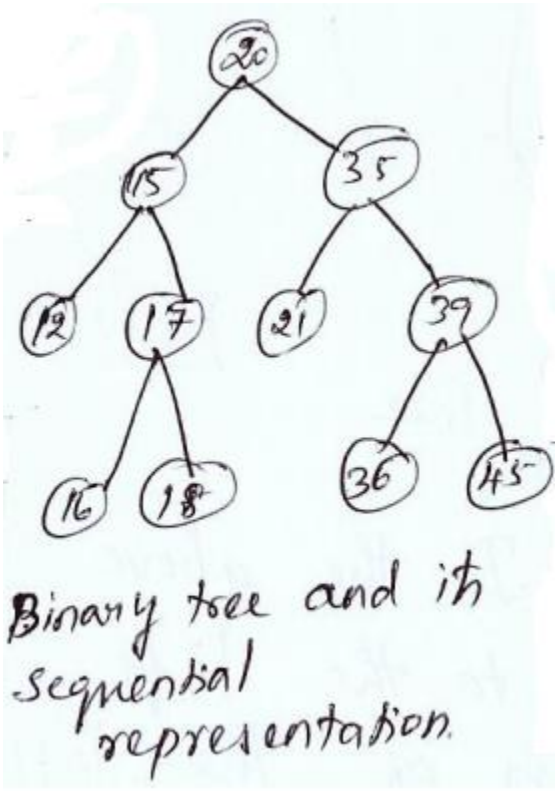
1. Pointer to left subtree
2. Pointer to right subtree
3. Data element

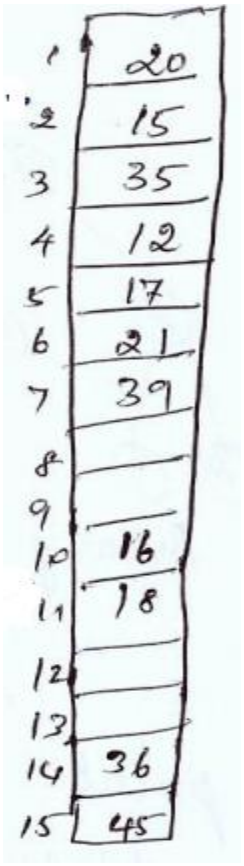
The topmost node in the tree is called the root. An empty tree is represented by NULL pointer.

A representation of binary tree is shown:



Array Representation: It Can be represented using Single one-dimensional array, but it is inefficient as it requires lots of memory space. The root of the tree is represented in the first location, followed by it's children. If a node does not have a children it occupies empty location.

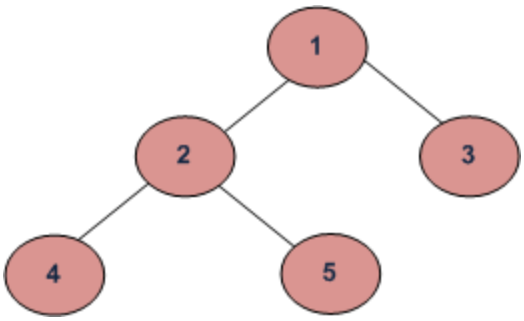




Tree traversals or Binary Tree Traversal

Traversing a binary tree is the process of visiting a node in the binary tree exactly once in a systematic way. There are three types of tree traversal

1. Depth First Traversals : It can be Traversed in 3 Ways
   - Inorder Traversal
   - Preorder Traversal
   - Postorder Traversal
2. Breadth First Traversal (Or Level Order Traversal)



Example Tree

1. **Depth First Traversals:** It can be Traversed in 3 Ways
    - Inorder Traversal (Left-Root-Right)
    - Preorder Traversal (Root-Left-Right)
    - Postorder Traversal (Left-Right-Root)

In-order Traversal (LRoR): In this, we do the following:

- First process left subtree.
- Then, process current root node.
- Then, process right subtree.

Procedure for Inorder Traversal

```
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
     printInorder(node->left);
     printf("%d ", node->data);
    printInorder(node->right);
}
```

Example: Inorder traversal for the above given figure is 4 2 5 1 3 .

Uses of Inorder

- In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.

210-J

Pre-order Traversal: (RoLR): In this technique, we do the following :

- Process data of root node.
- First, traverse left subtree completely.
- Then, traverse right subtree.

Procedure for Preorder Traversal:

```
void printPreorder(struct node* node)
{
    if (node == NULL)
        return;
     printf("%d ", node->data);
     printPreorder(node->left);
     printPreorder(node->right);
}
```

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Uses of Preorder

- Preorder traversal is used to create a copy of the tree.
- Preorder traversal is also used to get prefix expression on of an expression tree.

Postorder Traversal: (LRRo): In this traversal technique we do the following:

- First traverse left subtree completely.
- Then, traverse right subtree completely.
- Then, process data of node.

Procedure for Postorder

```
void printPostorder(struct node* node)
{
    if (node == NULL)
       return;
     printPostorder(node->left);
     printPostorder(node->right);
     printf("%d ", node->data);
}
```

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

Uses of Postorder

- Postorder traversal is also useful to get the postfix expression of an expression tree.

Main Program :

```
#include <stdio.h>
#include <stdlib.h>
struct node
{    int data;
     struct node* left;
     struct node* right;
};
struct node* newNode(int data)
{   struct node* node = (struct node*) malloc(sizeof(struct node));
    node->data = data;       node->left = NULL;       node->right = NULL;
    return(node);
}
int main()
{
     struct node *root  = newNode(1);
     root->left           = newNode(2);
     root->right          = newNode(3);
     root->left->left     = newNode(4);
     root->left->right   = newNode(5);

     printf("Preorder traversal");
     printPreorder(root);

     printf("Inorder traversal");
     printInorder(root);

     printf("Postorder traversal");
     printPostorder(root);
     return 0;
}
```

210-IIT

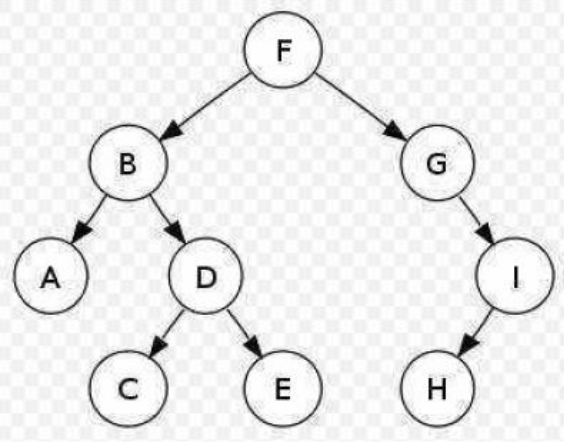
Breadth First or Level Order Traversal: In a breadth-first traversal, the processing proceeds horizontally form the root to all its children, then to its children's children, and so forth until all nodes have been processed. In other words, in breadth traversal, each level is completely processed before the next level is started.

Example: Breadth First Traversal for the above given figure is 1 2 3 4 5.

Procedure for breadth-first traversal

```
printGivenLevel(tree, level)

if tree is NULL then return;

if level is 1, then

   print(tree->data);

else if level greater than 1, then

printGivenLevel(tree->left, level-1);

printGivenLevel(tree->right, level-1);
```

Another example for tree traversal



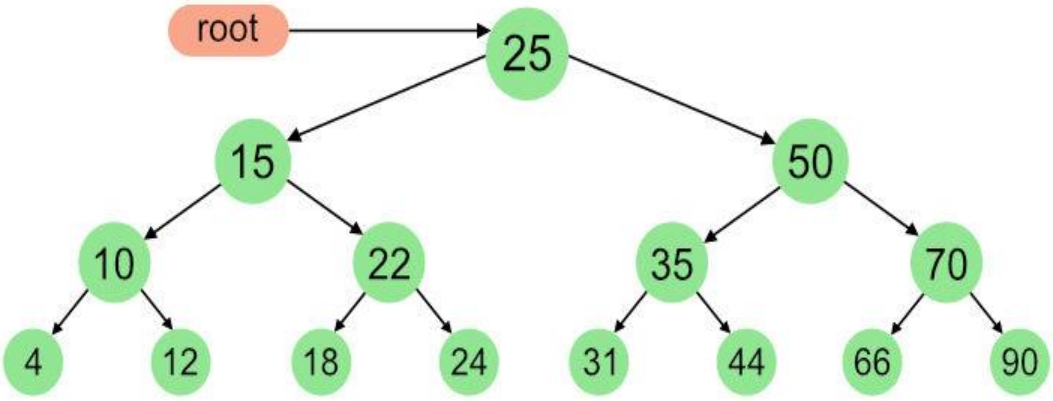
Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right)
Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)

One more example:

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Applications of Trees

1. Trees are used to store simple as well complex datas .
2. B+ trees are used to store tree structures on disc. Used as a index to large number of records.
3. for compiler construction.
4. used in File system directories
5. used in database design
6. used in compression algorithms and cryptographic applications
7. Trees are used in text processing (Dictionaries)

EXPRESSION TREES

Expression tree is a binary tree in which Leaf nodes are operands and internal nodes are operators.

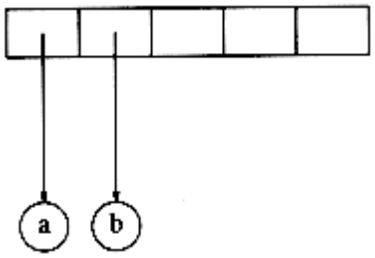
For constructing an expression tree by performing the following steps :

1. Convert the given expression into postfix form
2. Read one symbol at a time from the postfix expression.
3. Check whether the symbol is an operand or operator.
   i. If the symbol is an operand, create a one node tree and push a pointer on to the stack.
   ii. If the symbol is an operator, pop two pointers from the stack namely, T1 and T2 and form a new tree with root as the operator, and T2 as the left child and T1 as the right child.
   iii. A pointer to this new tree is then pushed on to the stack.

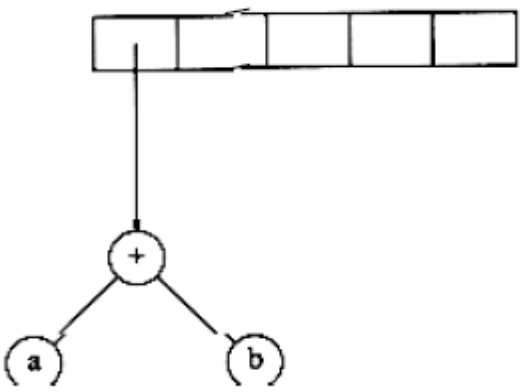
Example: Consider **Postfix form of expression** a b + c d e + * *

Steps to convert expression tree:
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.*

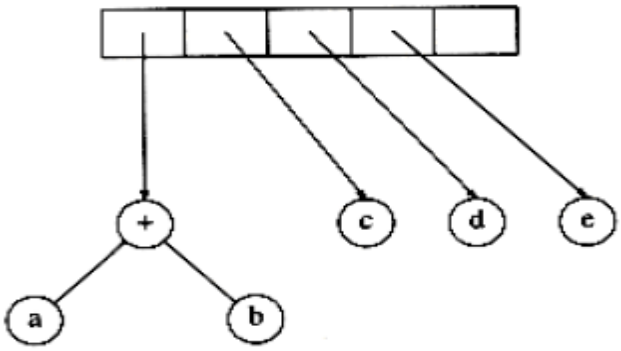
*For convenience, we will have the stack grow from left to right in the diagrams.



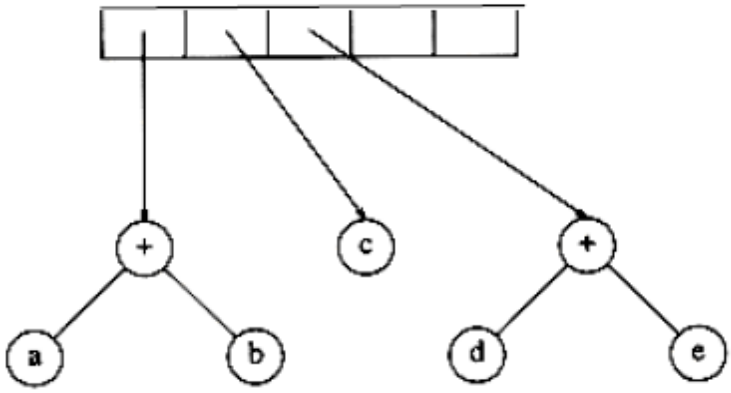
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.*



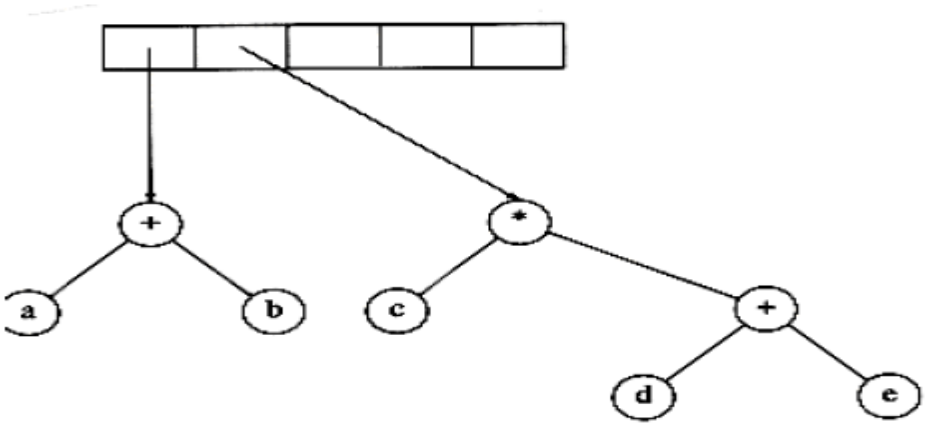
Next, *c*, *d*, and *e* are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



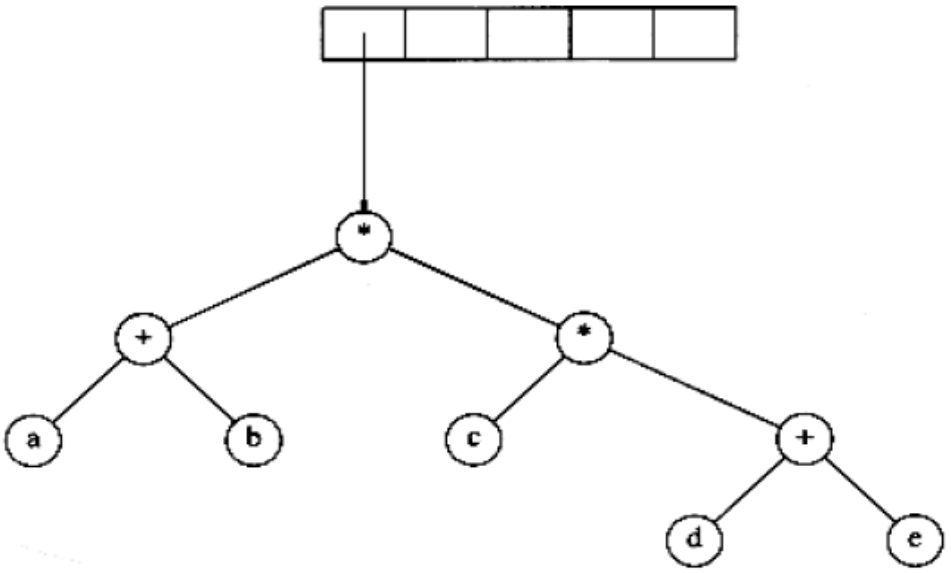
Now a '+' is read, so two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



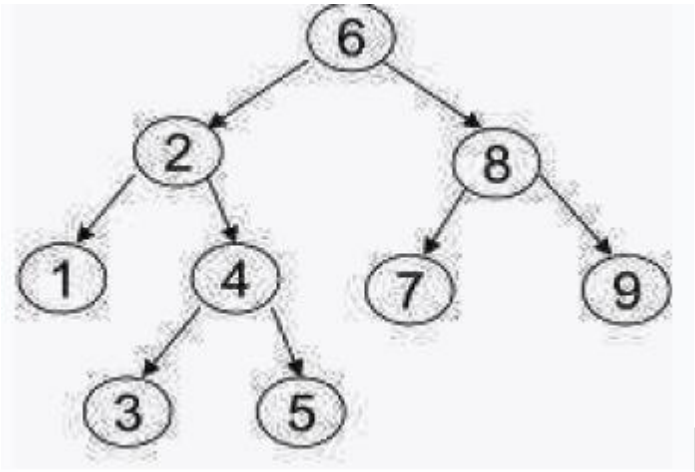
Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



BINARY SEARCH TREE

Binary search tree (**BST**) is a special type of binary tree data structure in which for each node key value,

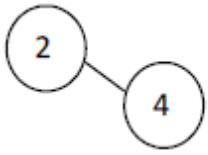
1. The left sub-tree of a node contains only nodes with keys less than the node's key.
2. The right sub-tree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right sub-trees must also be binary search trees.
4. Each node (item in the tree) has a distinct key.



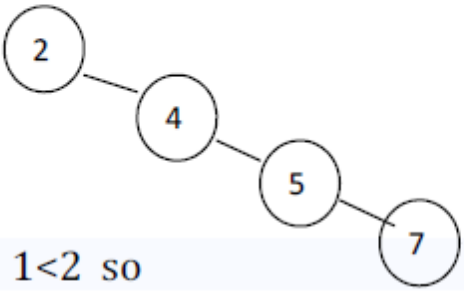
EXAMPLE Construct a BST with nodes 2,4,5,7,1.

The Values in the left subtree must be smaller than the keyvalue to be inserted.
The Values in the right subtree must be larger than the keyvalue to be inserted.

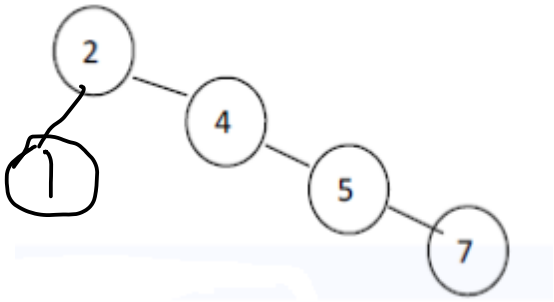
Take the 1st element 2 and compare with 4. 2<4
So



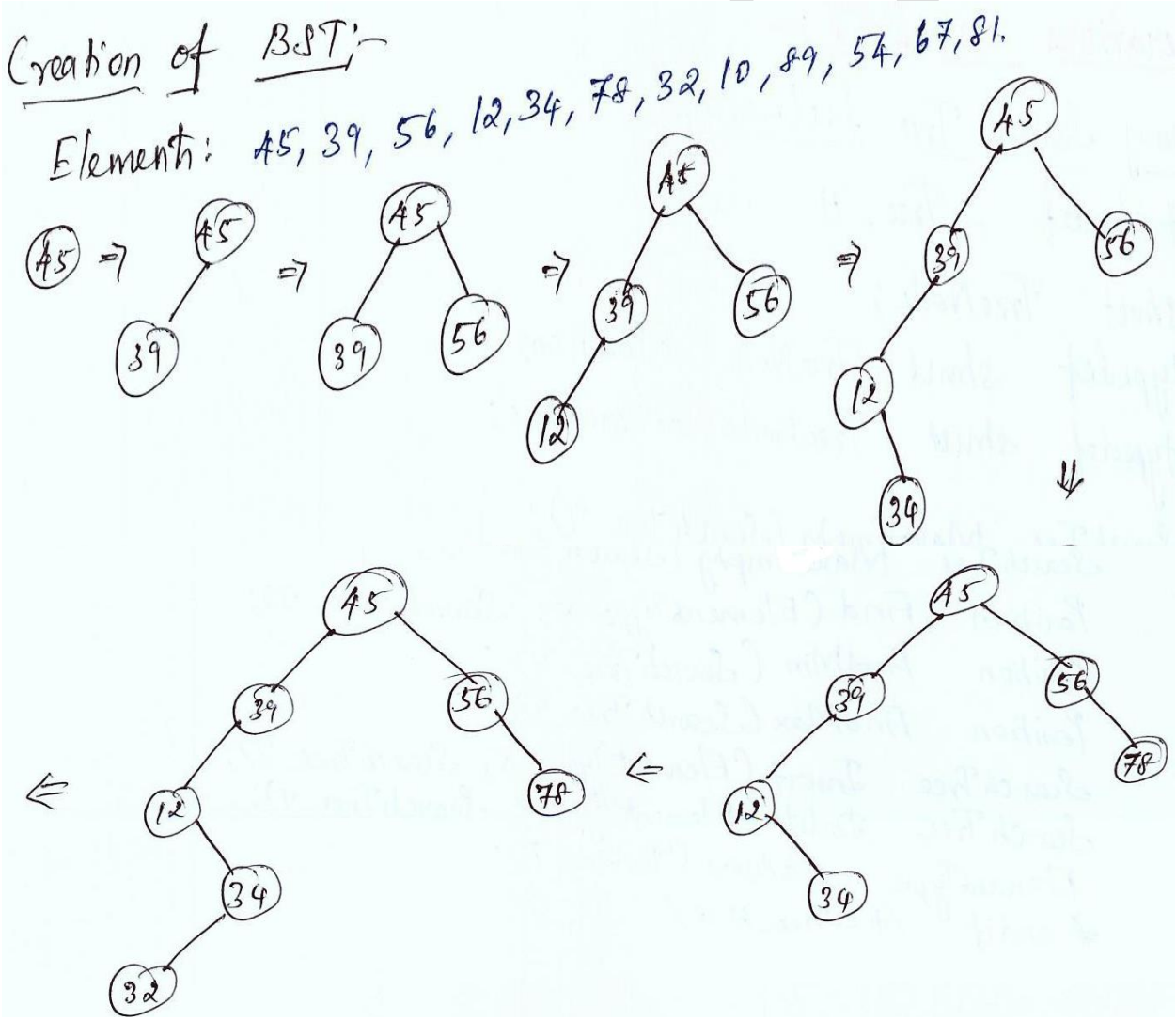
Similarly 2<5,5>4 and 7>2,7>4,7>5



and 1<2 so

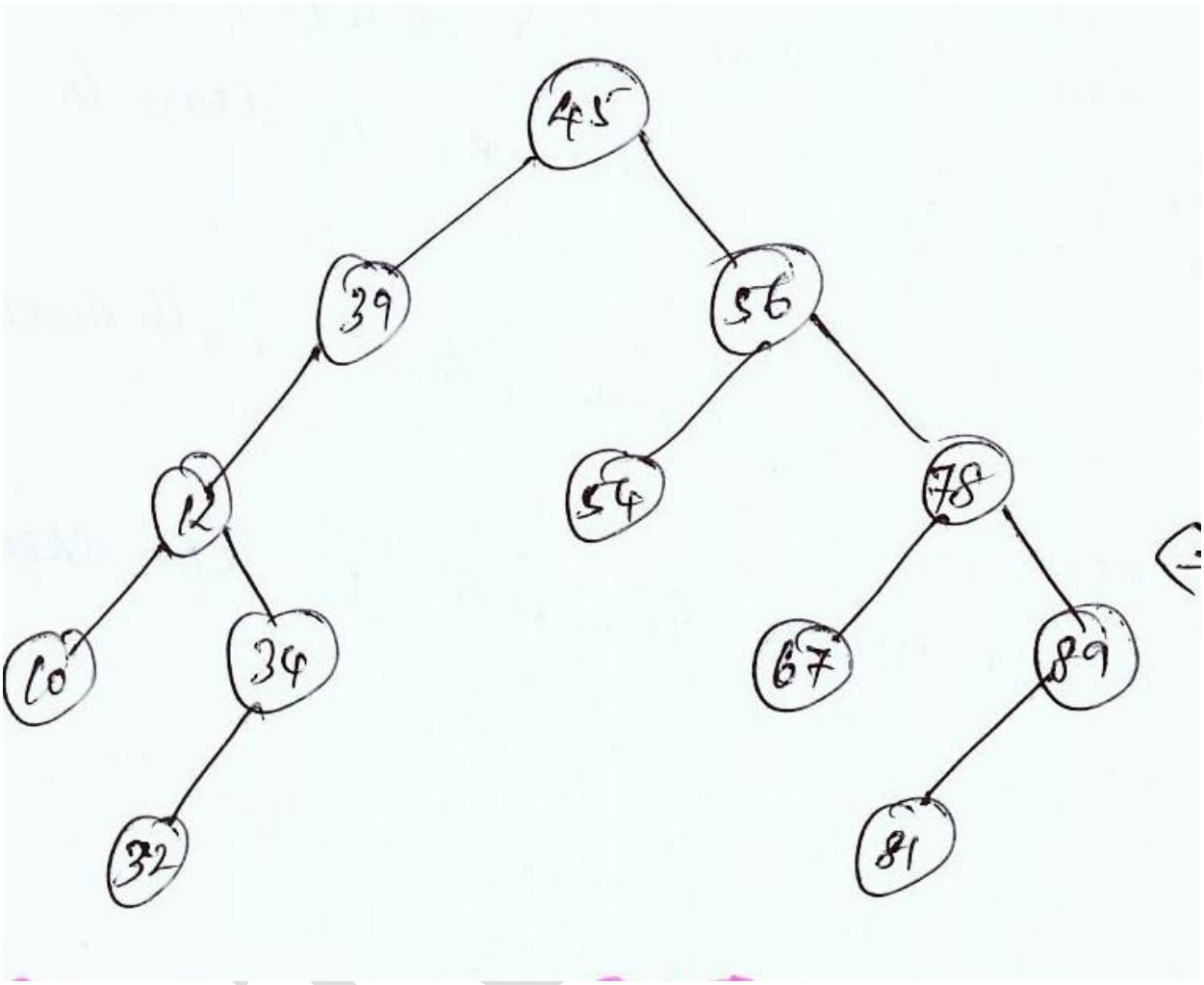


Example 2:



Continue like this ……

Final Tree is



OPERATIONS

Operations on a binary search tree require comparisons between nodes.
The following are the operations that are being done in Binary Search Trees

- Searching - Find Min or Find Max .
- Insertion.
- Deletion.
- Sorting.

Find

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 key stored in T.

Find operation for binary search trees

```
Position find(structtreenode T, intnum)
{
While(T!=NULL) {
if(num>T-->data) {
T=T-->right;
if(num<T-->data)
T=T-->left; }
else if(num< T-->data)
{
T=T-->left;
if(num>T-->data)
T=T-->right;
}
if(T-->data==num)
break; }
return T; }
```

Find_min and Find_max

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 in the BST.
To perform a Findmax, start at the root note and go right as long as there is right child. The stopping point is the largest element.

Recursive implementation of find_min & find_max for binary search Trees

// Finding Minimum Position

```
findmin(searchtree T)
{
if(T==NULL)
return NULL;
else if(T-->left==NULL)
return T;
else
return findmin(T-->left);
}
```

206-JIT

// Finding Maximum Position

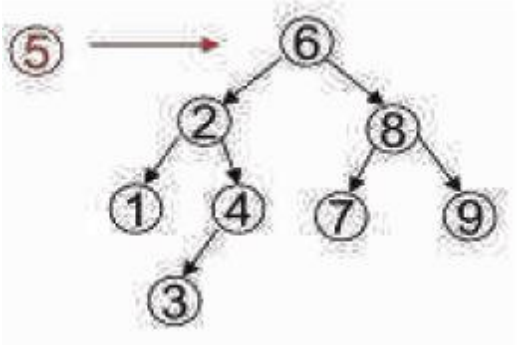
```
findmax(searchtree T)
{
if(T==NULL)
return NULL;
else if(T-->right==NULL) return T;
else
return findmax(T-->right);
}
```

Insert

To insert *x* into tree *T*, proceed down with the following steps.

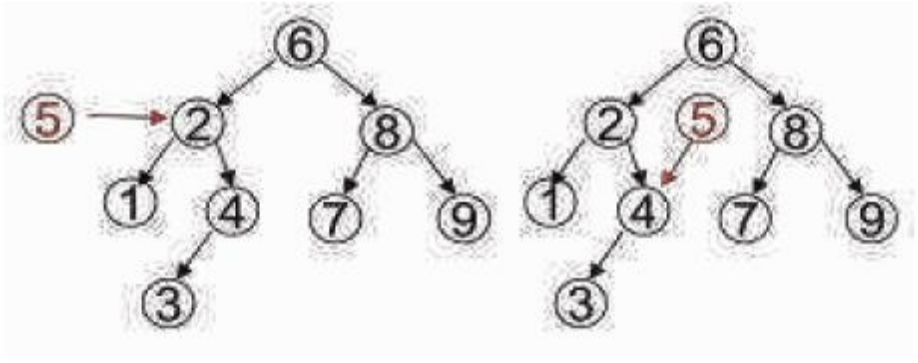
1. Allocate memory for the new node X
2. The insertion of an element X into the tree is as follows
3. check the root node of the tree is null
4. if the condition is true then, then the new node is the root node. otherwise follow the next few steps,
5. compare the new node with the root node key value.
   - If the new node is less than the root node, Traverse the left subtree recursively until it reaches left Null, then left is assigned to new node.
   - If the new node is greater than the root node, Traverse the right subtree recursively until it reaches right Null, then right is assigned to new node.

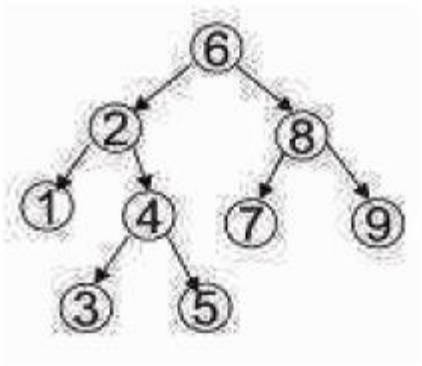
EXAMPLE Insert node 5 in given tree



CSE-JIT

STEP 1: Now 5<6 and 5>2 and 5<4 so





Thus 5 is inserted.

Procedure for Insertion into a binary search tree

```
Searchtree insert(elementtype X, Searchtree T)
{
If(T== NULL) {
/* create and return a one node tree*/ T=malloc(sizeof(structtreenode));
If(T==NULL)
Fatalerror("Out of Space");
Else {
T-->element=X;
T-->left=T-->right=NULL;
}
}
Else if(x<T-->element) T-->left=insert(X,T-->left);
Else if(X>=T-->left) T-->right=insert(X,T-->right); Return T;
```

}

Delete

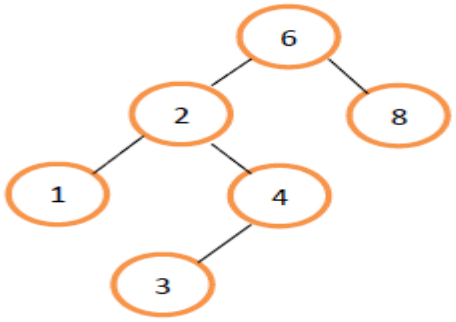
The hardest operation is deletion in BST. It can be done in 3 ways.

- Deleting a leaf node
- Deleting a node with one child
- Deleting a node with Two child

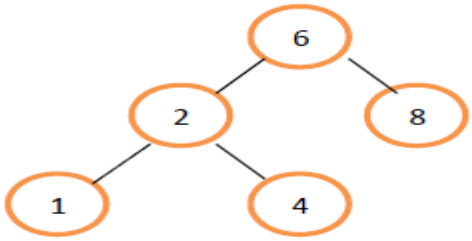
Case 1: Deleting a leaf node: If the node is a leaf, it can be deleted immediately.

Steps are

1. Search the parent of the leaf node and make the link to the leaf node as NULL.
2. Release the memory of the deleted node.



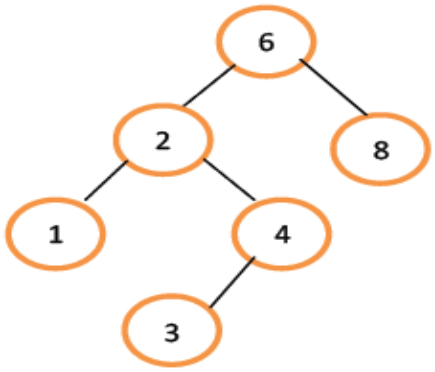
Before Deletion



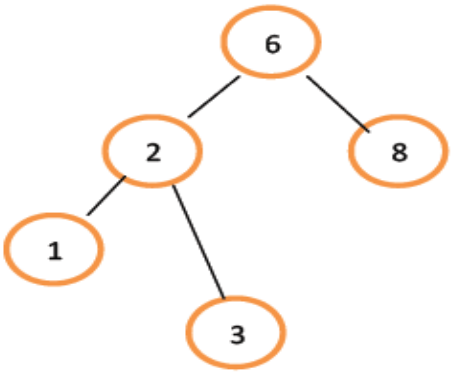
After Deletion of 3

Case 2: Deleting a node with one child:

1. Search the parent of the node to be deleted.
2. Assign the link of the parent node to the child of the node to be deleted.
3. Release the memory for the deleted node.



Before deletion of 4

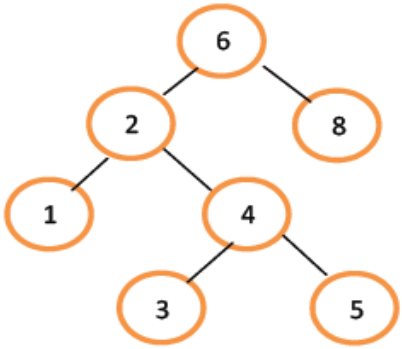


After deletion of 4

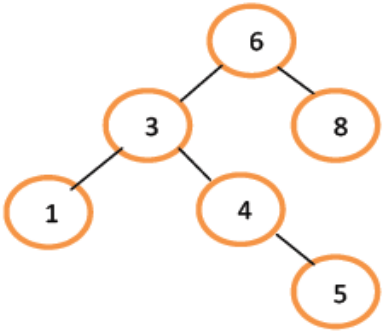
Case 3: Deleting a node with two child:

It is difficult to delete a node which has two children.
So, a general strategy has to be followed.

1. Replace the data of the node to be deleted with either the **<u>smallest element from the right subtree</u>** or the largest element from the left subtree.



Before deletion of 2



After deletion

Deletion routine for binary search trees

```
Searchtree delete(elementtype X, searchtree T)
{
positiontmpcell;
 if(T==NULL)
error("element not found");
```

```
else if(X<T-->element)
 T-->left=delete(X,T-->left);
Else if(X>T-->element)
T-->right=delete(X,T-->right);
Else if(T-->left != NULL && T-->right!=NULL)
{
/* Replace with smallest in right subtree*/
 Tmpcell=findmin(T-->right);
T-->element=tmpcell-->element;
T-->right=delete(T-->element,T-->right);
 }
Else
 {
/* One or Zero children*/
tmpcell=T; if(T-->left==NULL)
T=T-->right;
Else if(T-->right==NULL) T=T-->left;
Free(tmpcell);
 }
Return T;
}
```

Applications of Binary Search Trees

- implementation of a dynamic dictionary. because a dictionary is an ordered list that is required to be searched frequently, and is also required to be updated (insertion and deletion mode) frequently. So it can be implemented by making the entries in a dictionary into the nodes of a binary search tree.
- A more efficient implementation of a dynamic dictionary involves considering a key to be a sequence of characters, and instead of searching by comparison of entire keys, we use these characters to determine a multi-way branch at each step. This will allow us to make a 26-way branch according to the first letter, followed by another branch according to the second letter and so on.

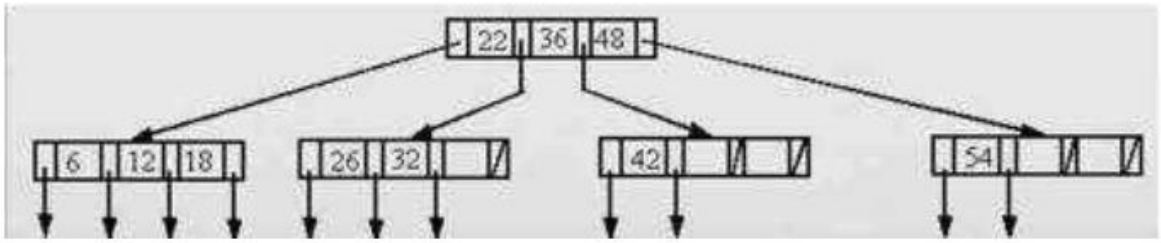
B-TREES

A **B-tree is a balanced** m- ordered multiway search tree, where m > 2. B-Tree of order m can have at most m-1 keys and m children.

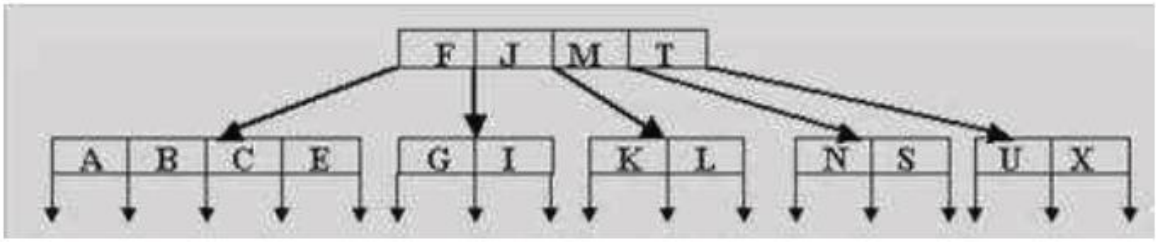
206-JIT

One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

Example: A B-tree of order 4



Example: A B-tree of order 5



Properties of B Tree:

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.
5. It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

Operations on B Tree

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following:

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Insertions

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Note:
If M is odd, split the node from the median. If M is even , then split as right-bias or left bias.
right-bias: The node is split such that its right subtree has more keys than the left subtree.
left-bias: The node is split such that its left subtree has more keys than the right subtree.

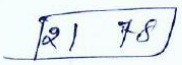
Example ②:

Insert the elements into a B-tree of order 5.

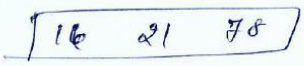
78, 21, 16, 11, 97, 85, 74, 63, 45, 42, 57, 20, 14, 19, 32, 30, 31.

Insert 78

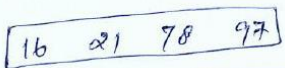
Insert 21



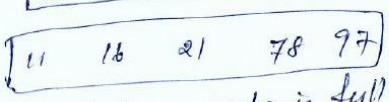
Insert 16



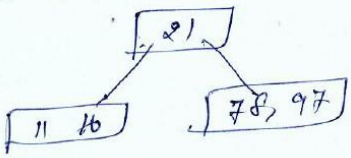
Insert 97



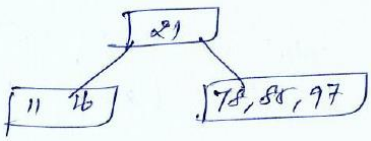
Insert 11



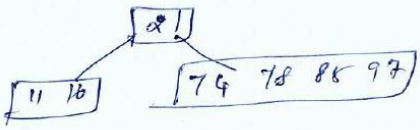
Now the B-tree node is full. Split it into two halves



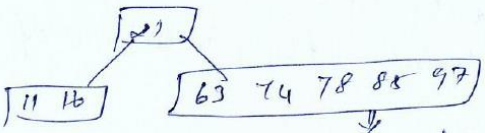
Insert 85



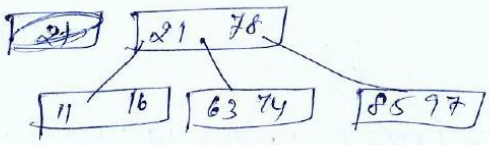
Insert 74



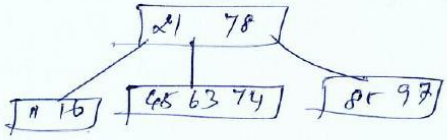
Insert 63



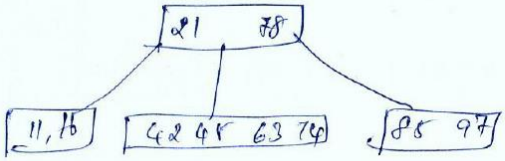
this node is full, Split it into two halves



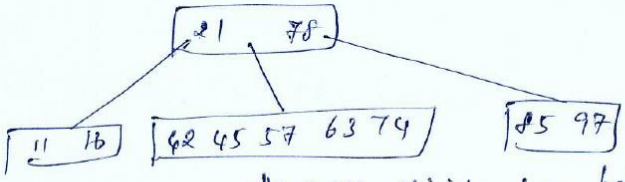
Insert 45



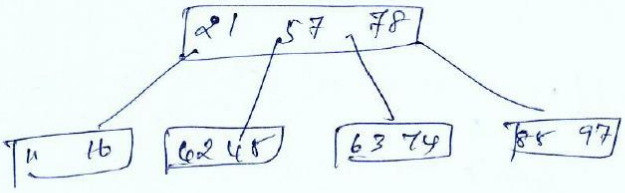
Insert 42



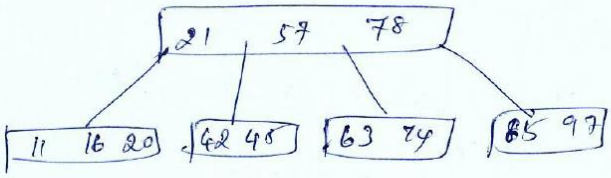
Insert 57



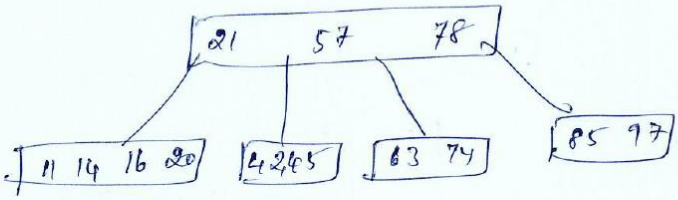
↓ Split it into two halves



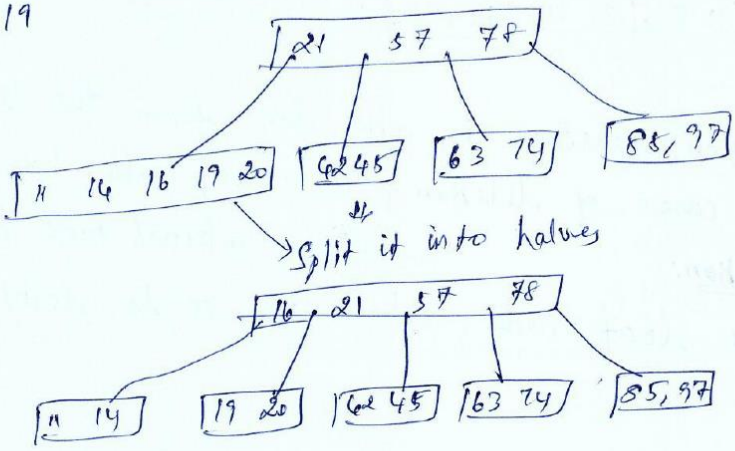
Insert 20

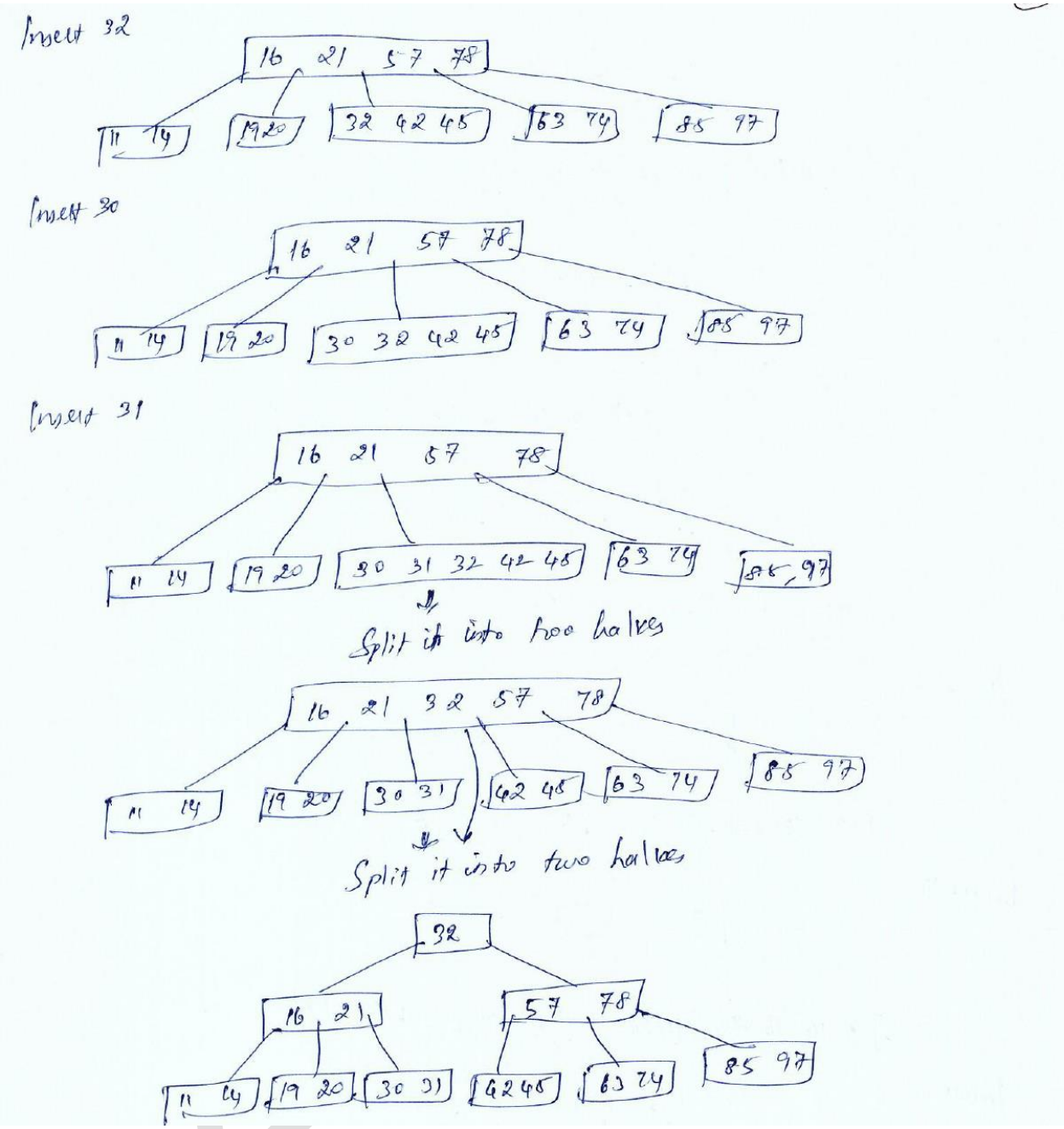


Insert 14



Insert 19





Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.

2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example:

Insert the node 8 into the B Tree of order 5 shown in the following image

Example for deletion

order (m) = 5

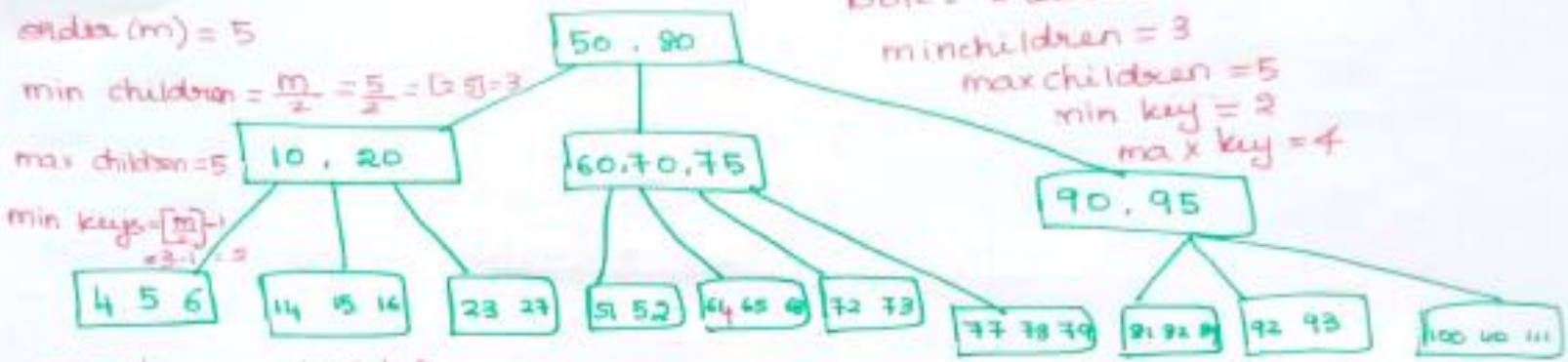
min children = $\frac{m}{2} = \frac{5}{2} = \lceil 2.5 \rceil = 3$

max children = 5

min keys = $\lceil \frac{m}{2} \rceil - 1 = 3 - 1 = 2$

max keys = m - 1 = 5 - 1 = 4

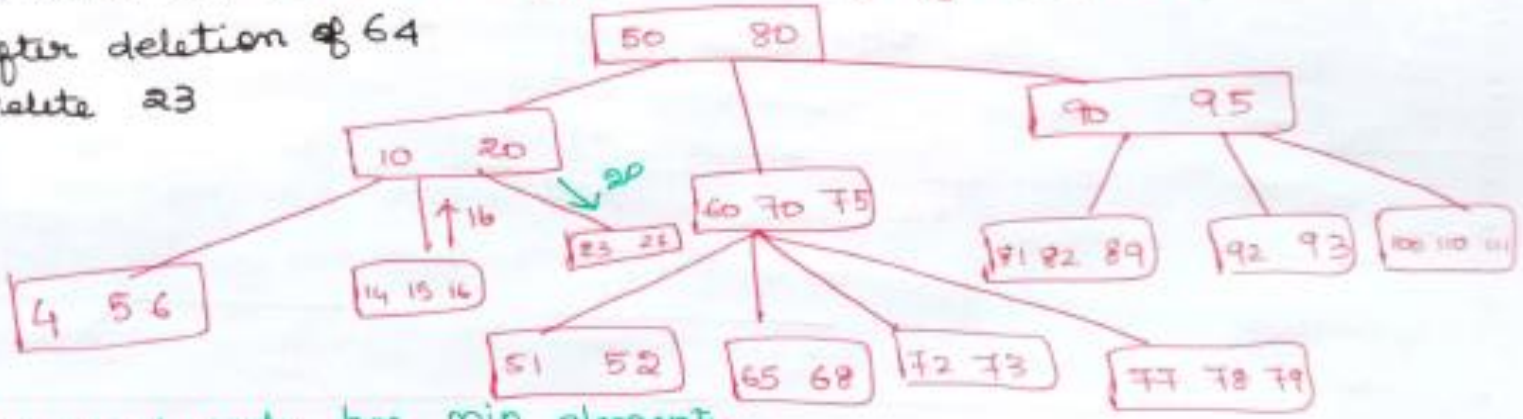
Note: order = 5
min children = 3
max children = 5
min key = 2
max key = 4



① Delete 64 → has more than min no. of keys, so delete

After deletion of 64

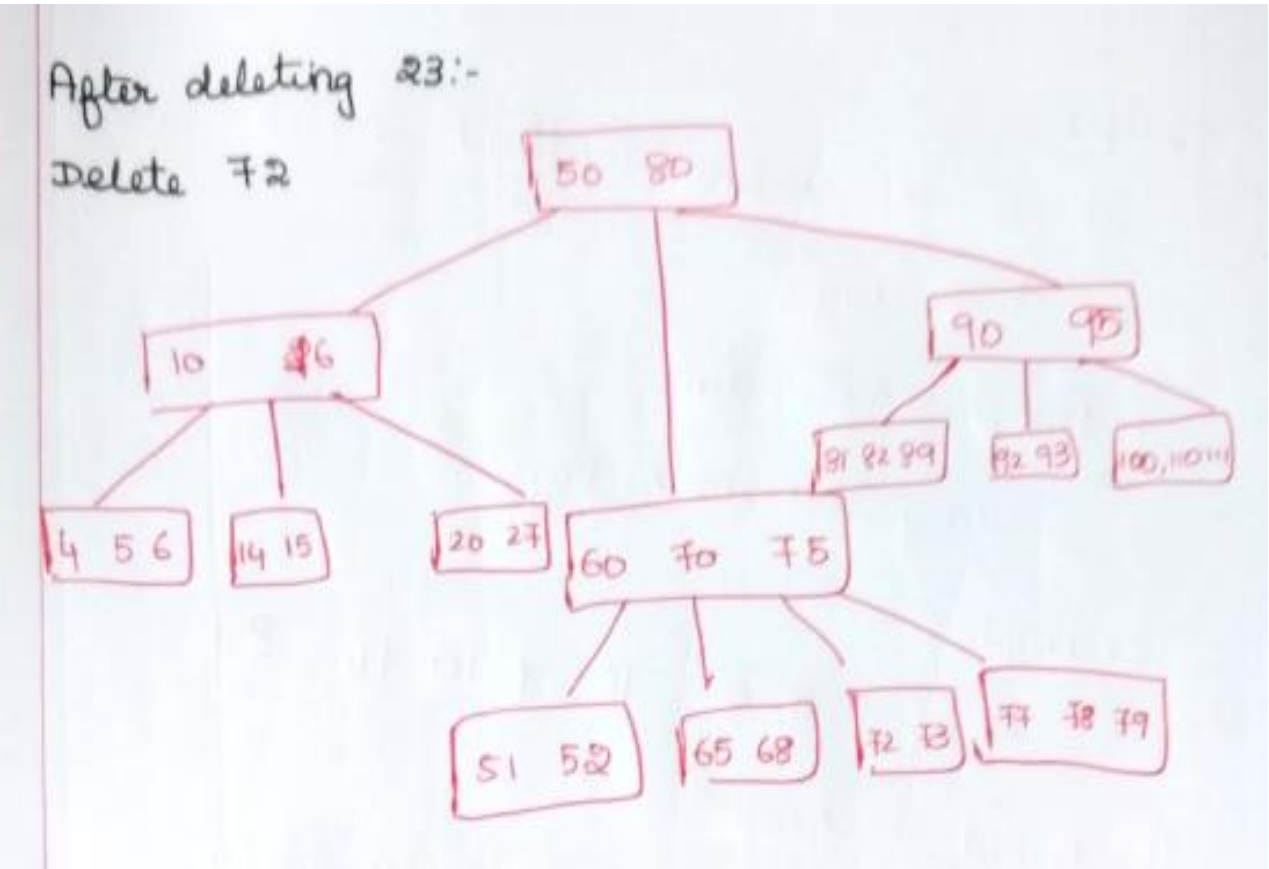
Delete 23



Note: leaf node has min elements

so push max value from left sibling to parent

move inbetween parent down & then delete 23



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

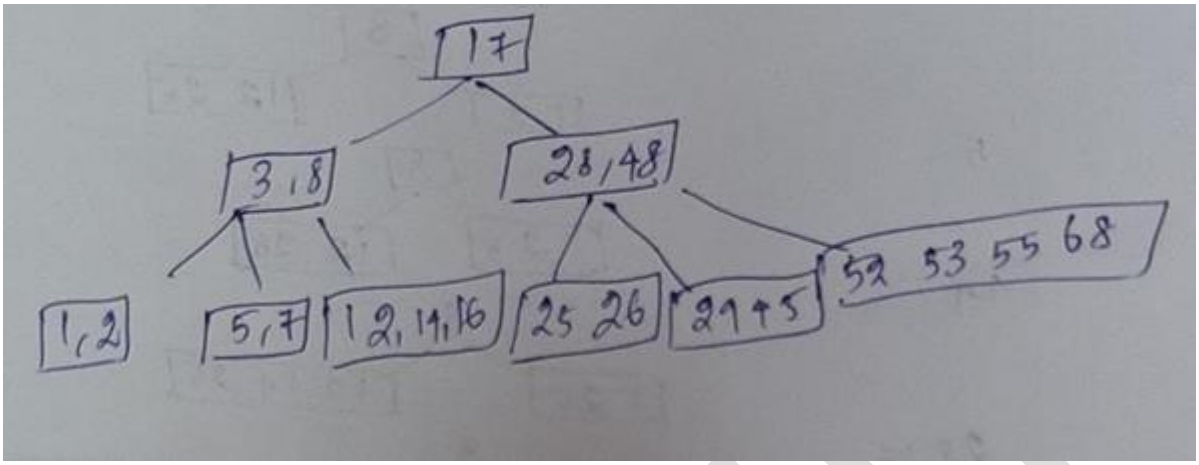
Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

Example 2:

Construct B Tree of order m=5 for the following keys 1, 12, 8, 2, 25, 5, 14, 28, 17, 7, 52, 16, 48, 68, 3, 26, 29, 53, 55, 45 . State the rules for deletion and Delete the keys 8 and 55.

Ans:

Insertion:



Deletion:

B+ Tree

B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient

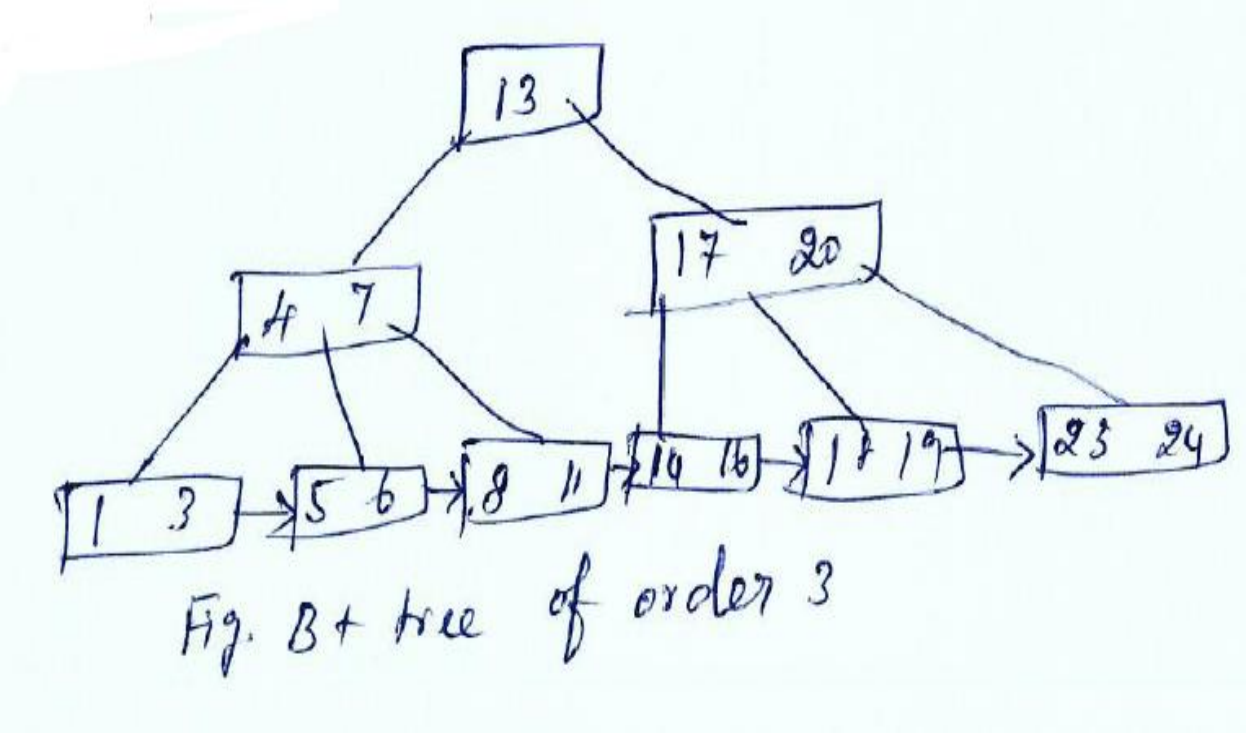


Fig. B+ tree of order 3

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Threaded Binary Trees- AVL Trees - Heap – Applications of heap.

Threaded Binary Trees

The binary tree can have at most two children. But if the tree have only one children, or no children, the link part in the linked list representation remains null. In threaded binary tree representation, empty links are reused as threads.

Types are

1. Single threaded tree - (Left threaded and right threaded)
2. Fully threaded binary tree

Single Left threaded tree:

In this, if some node has no left child, then the left pointer will point to its inorder predecessor. If no predecessor is present, then it will point to header node.

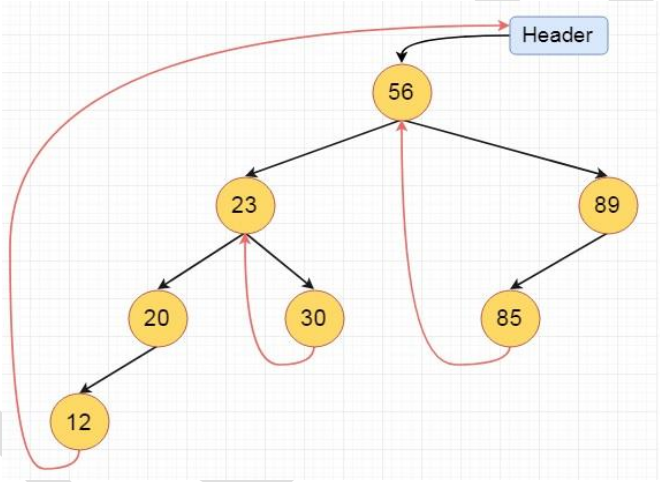


Fig: Single Left threaded tree

Single Right threaded tree:

In this, if some node has no right child, then the right pointer will point to its inorder successor. If no successor is present, then it will point to header node.

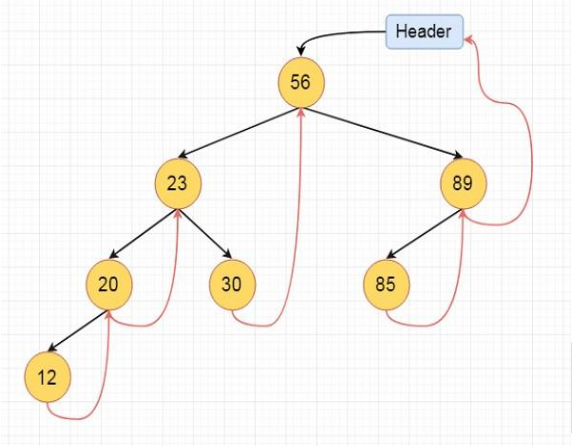


Fig: Single Right threaded tree

Fully threaded binary tree:

In fully threaded binary tree, each node has five fields. Three fields like normal binary tree node, another two fields to store Boolean value to denote whether link of that side is actual link or thread.

Left Thread Flag	Left Link	Data	Right Link	Right Thread Flag

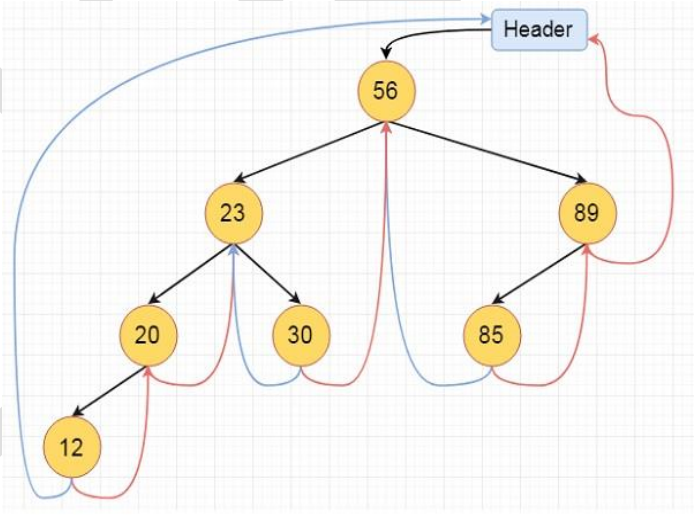


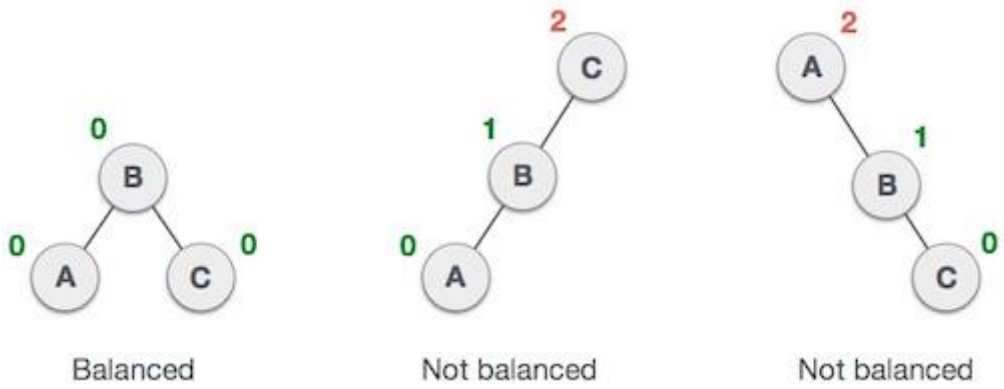
Fig: Fully threaded binary tree

AVL Trees

AVL trees, Named after their inventor Adelson, Velski & Landis are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

BalanceFactor = height(left-subtree) − height(right-subtree)

Example: The first tree is balanced, and the next two trees are not balanced



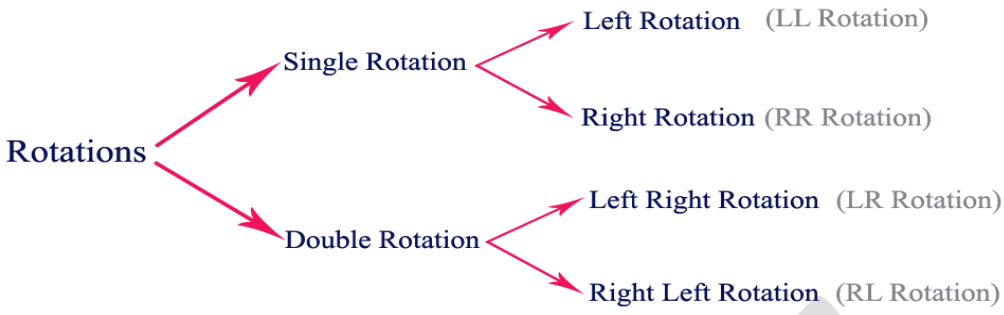
In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Tree Rotations

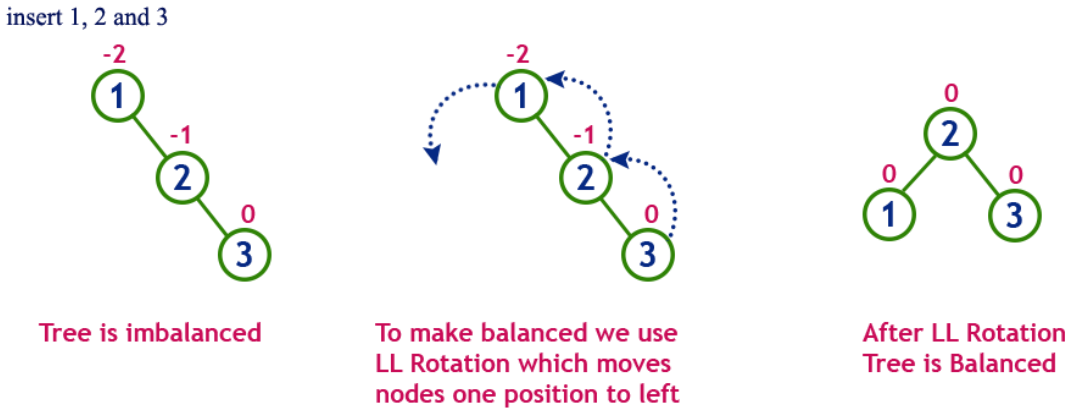
In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node, If the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

There are **four** rotations and they are classified into **two** types.



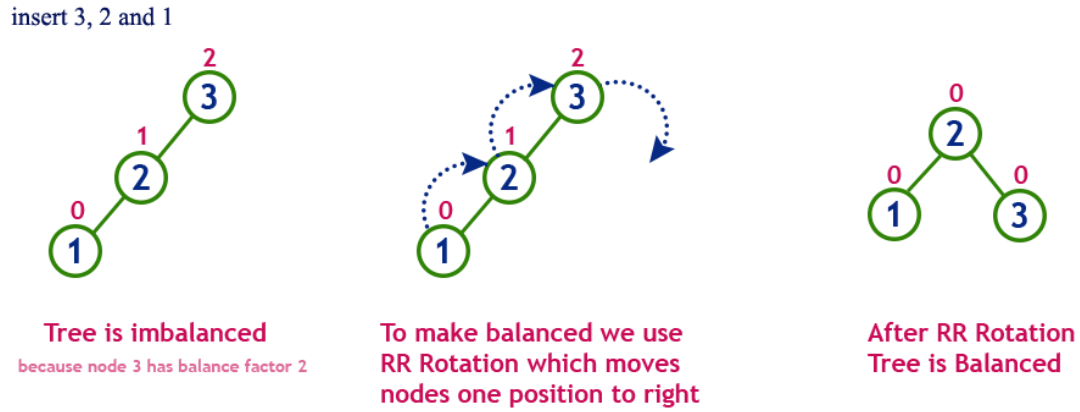
Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



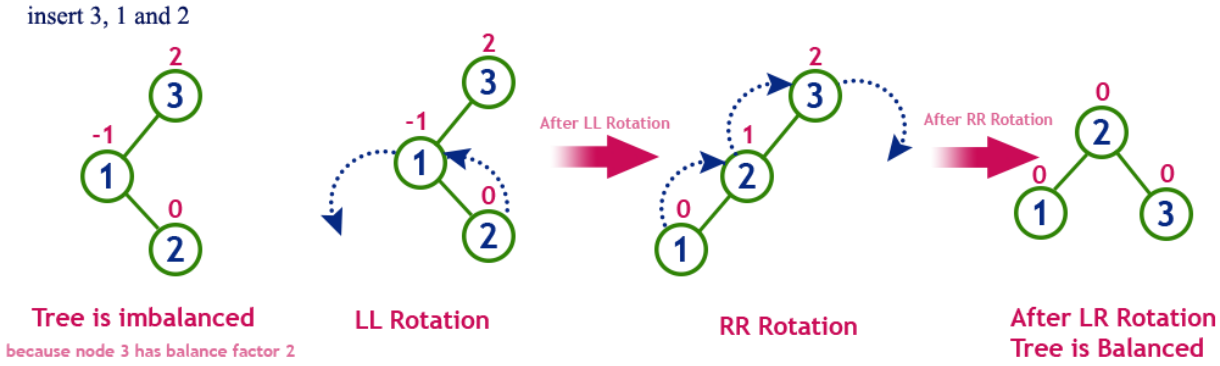
Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



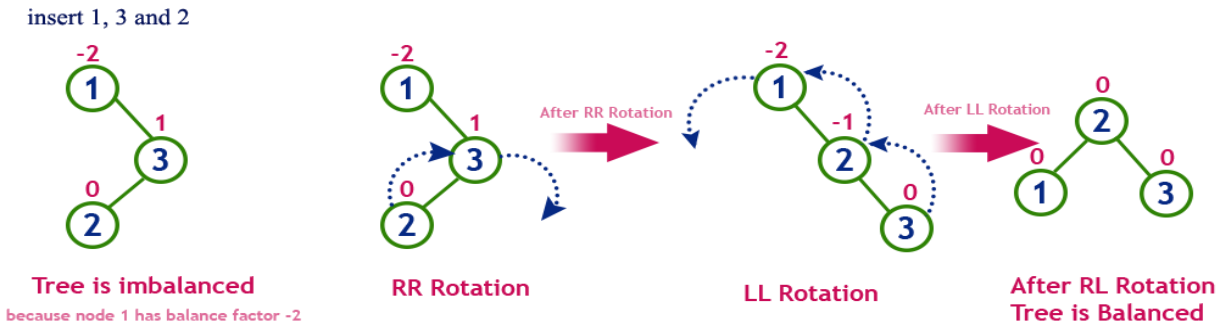
Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

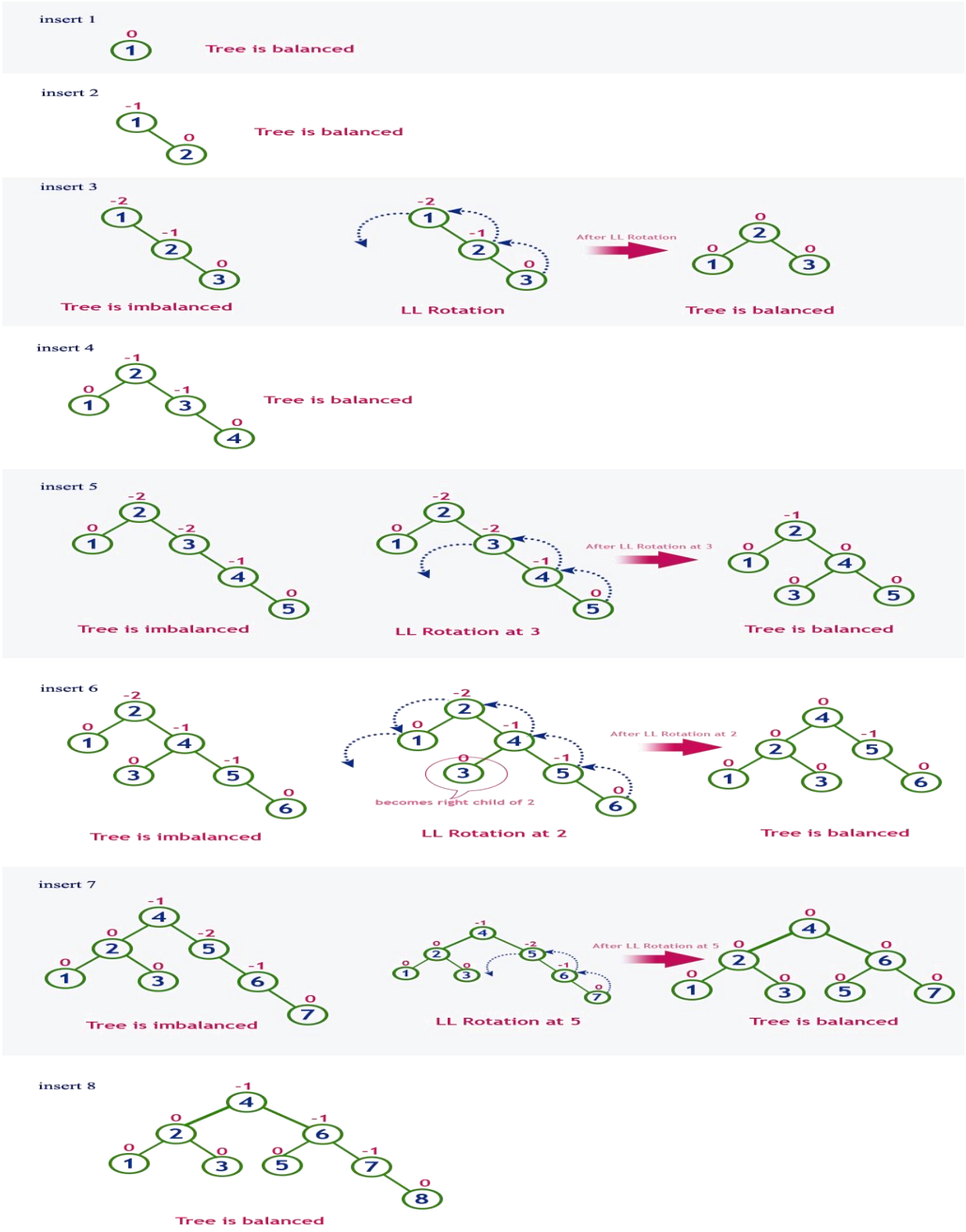
In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the **Balance Factor** of every node.
- Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.



Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

Heap

Heap data structure is a specialized binary tree-based data structure. In a heap data structure, nodes are arranged based on their values. A heap data structure also called as Binary Heap. There are two types of heap data structures.

1. Max Heap
2. Min Heap

Every heap data structure has the following properties...

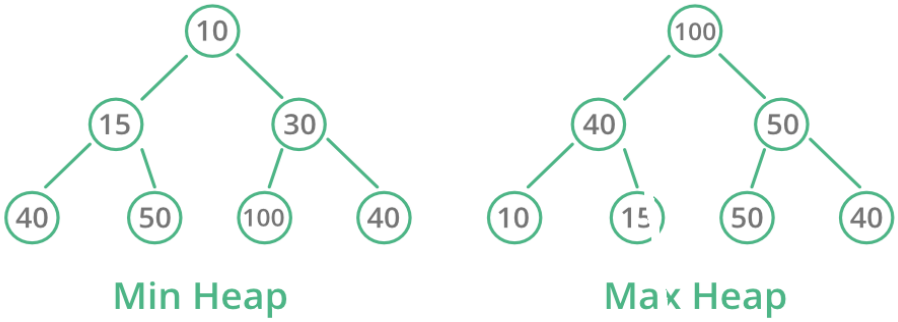
Property #1 (Ordering): Nodes must be arranged in an order according to their values based on Max heap or Min heap.
Property #2 (Structural): All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

Min Heap

In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

Max Heap

In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



Operations on Max Heap

The following operations are performed on a Max heap data structure.

1. Finding Maximum
2. Insertion
3. Deletion

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

Insertion Operation in Min Heap

Note : The procedure to create Max Heap is similar but consider max values instead of max values.

Algorithm for max heap

Inserting one element at a time, with the following steps,

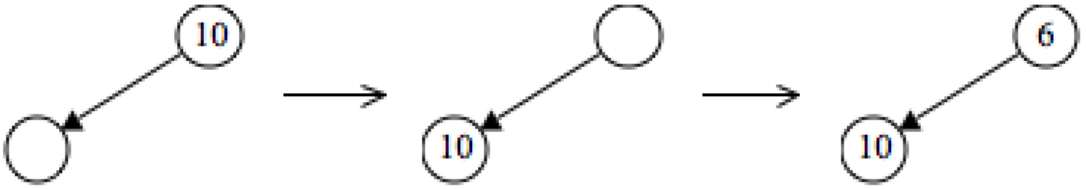
Step 1 – Create a new node at the end of heap.
Step 2 – Assign new value to the node.
Step 3 – Compare the value of this child node with its parent.
Step 4 – If value of parent is greater than child, then swap them.
Step 5 – Repeat step 3 & 4 until Heap property holds.

Insert into a heap the following values in order: 10,6,20,5, 16, 17, 13,2

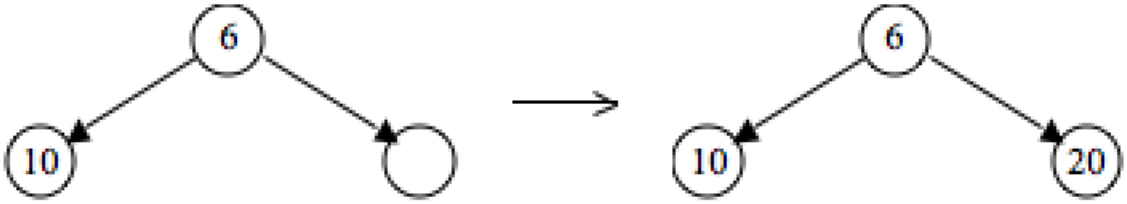
We will use smaller values has higher priority as our priority ordering.

insert 10:

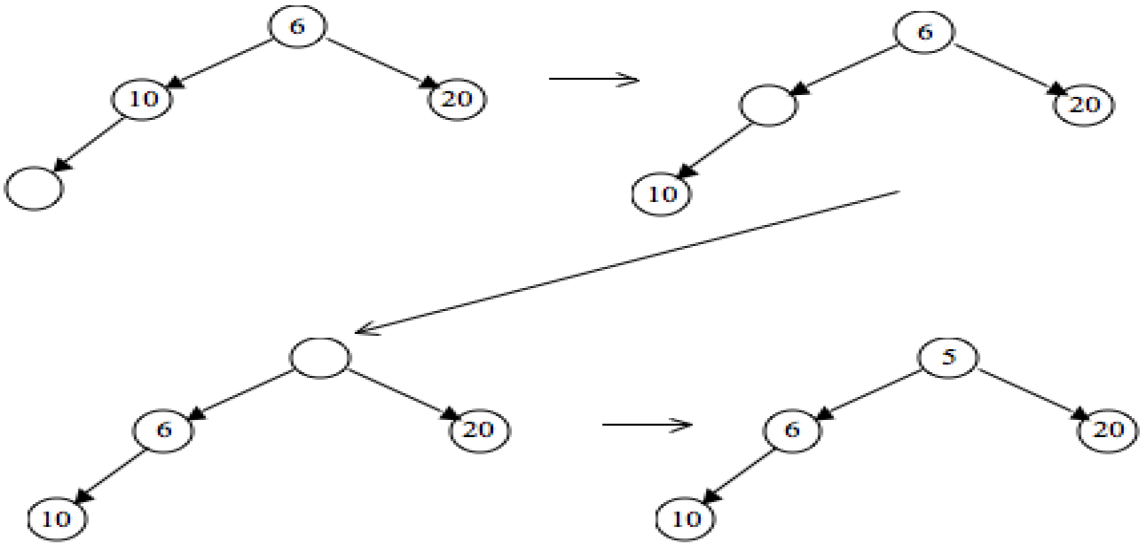
insert 6:

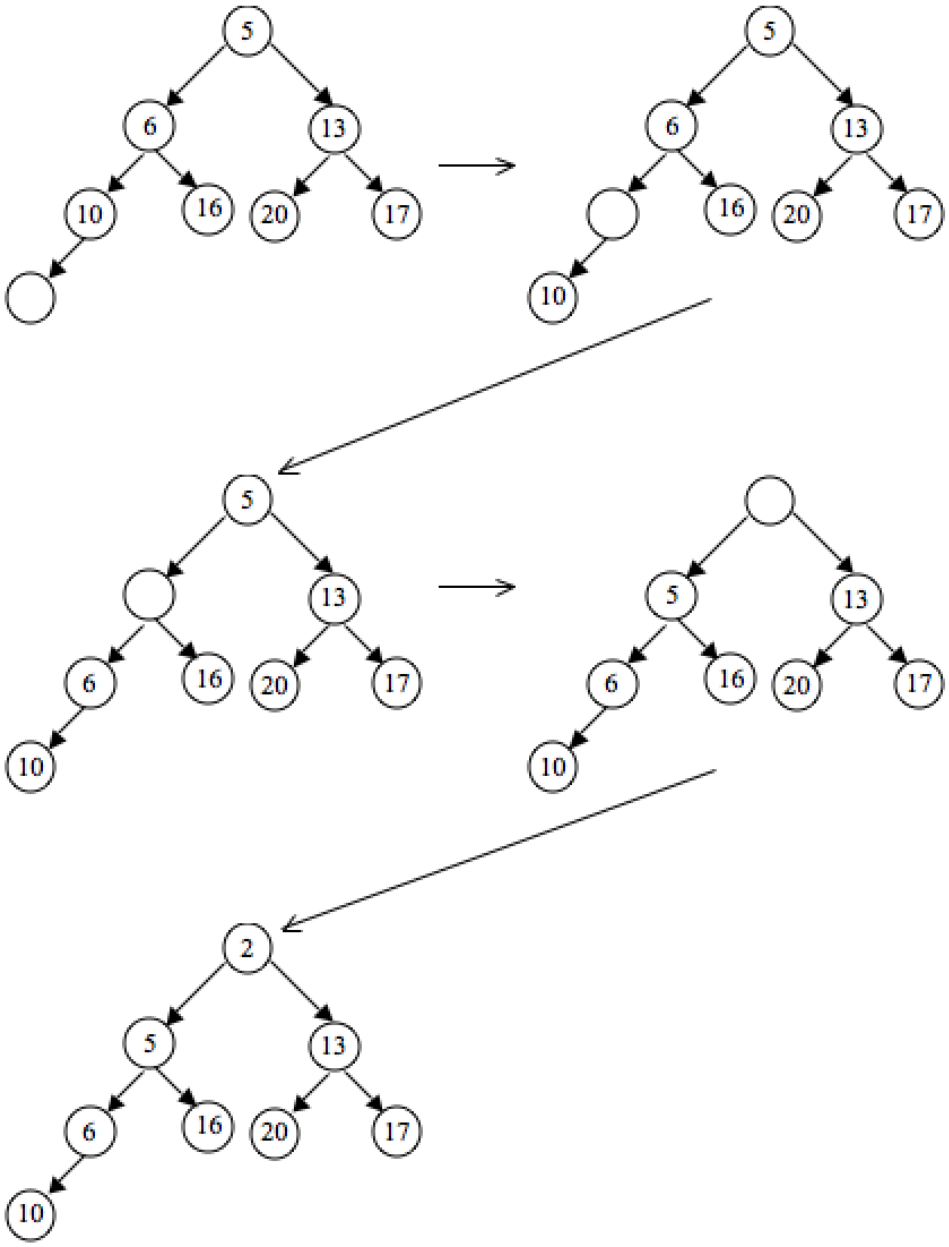


insert 20:

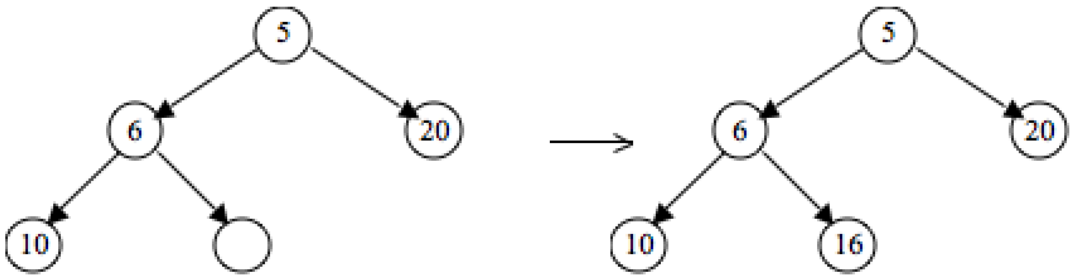


insert 5:

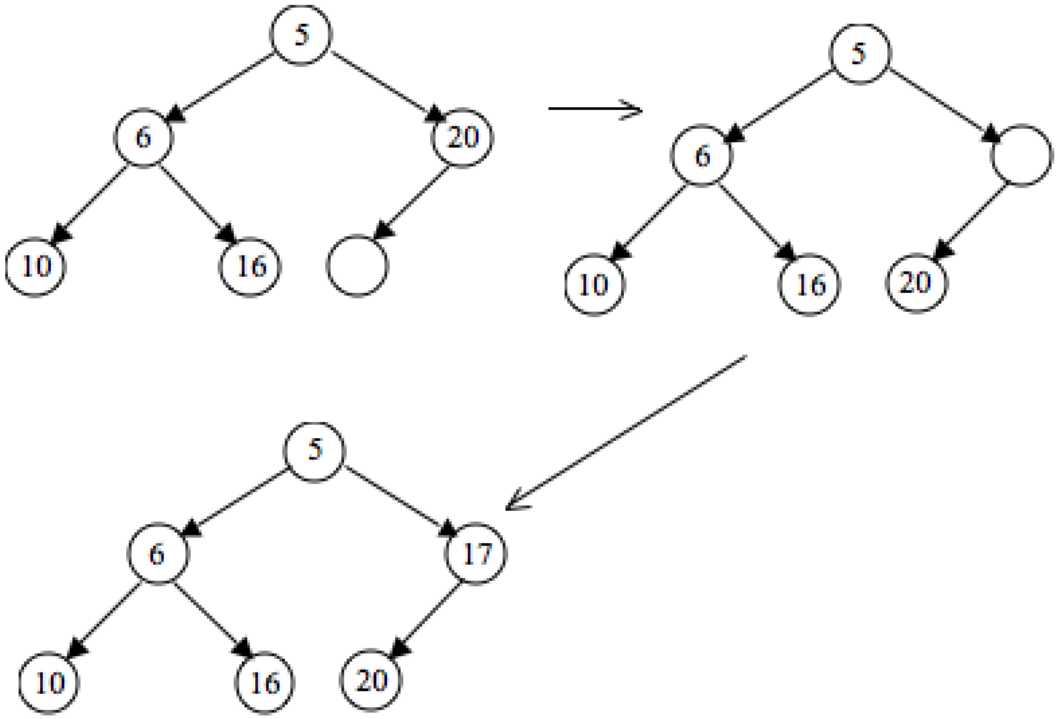



5
6
13
10
16
20
17
5
6
13
16
20
17
10
5
13
6
16
20
17
10
5
13
6
16
20
17
10
2
5
13
6
16
20
17
10

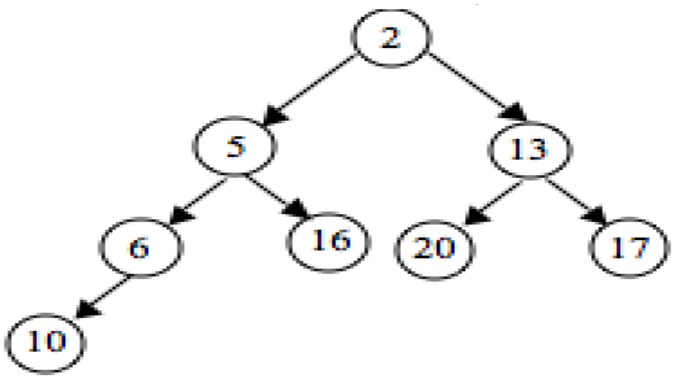
insert 16:


5
6
20
10
5
6
20
10
16

insert 17:



Repeat the steps for **13 and 2. The final tree is**



Delete from a binary heap

This is the process of removing the highest priority value from the binary heap. The deleting an element must ensure

- complete binary tree structure

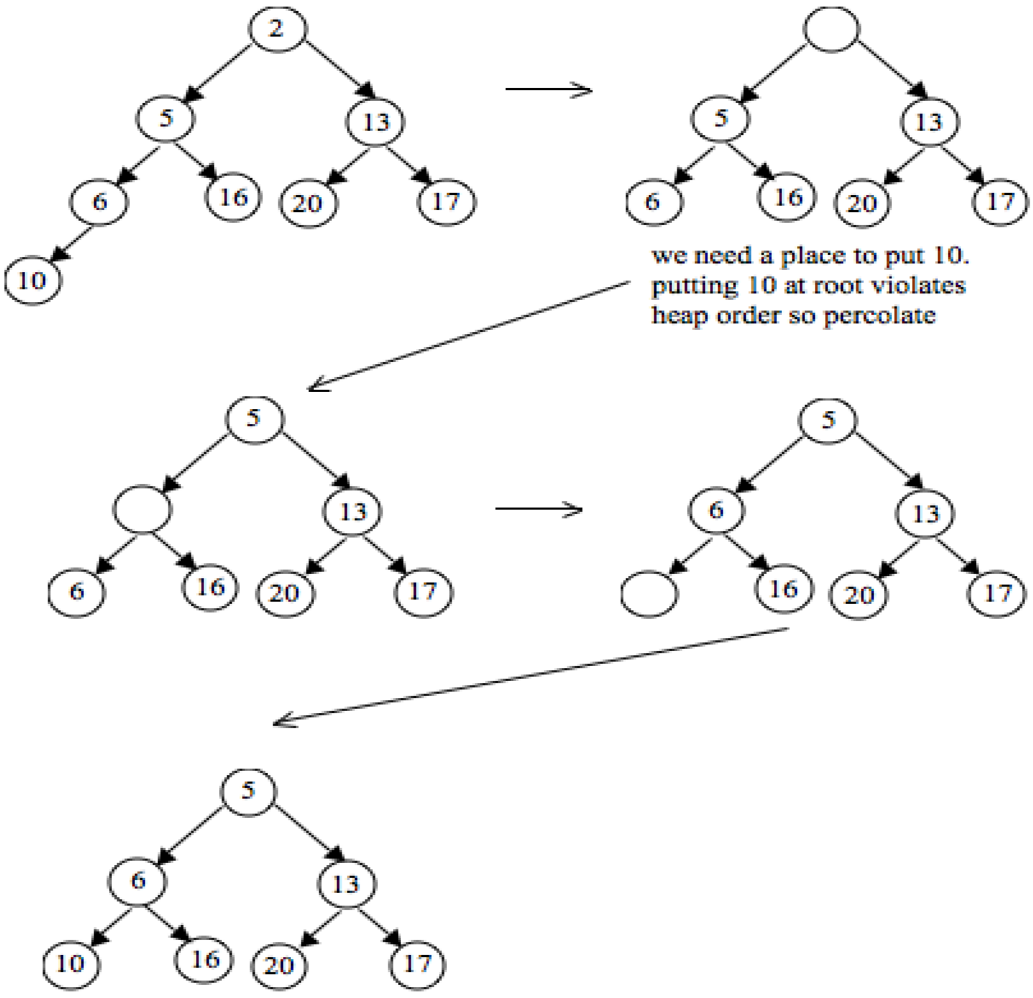
- heap order property

is maintained after the removal of an element.

Deletion Procedure

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.
- Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.

The process of moving the empty spot down the heap is called ***percolate down.***



Applications of Heap

1) Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.

2) Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.

3) Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.

4) Many problems can be efficiently solved using Heaps. See following for example.

a) K'th Largest Element in an array.
b) Sort an almost sorted array
c) Merge K Sorted Arrays.

210