

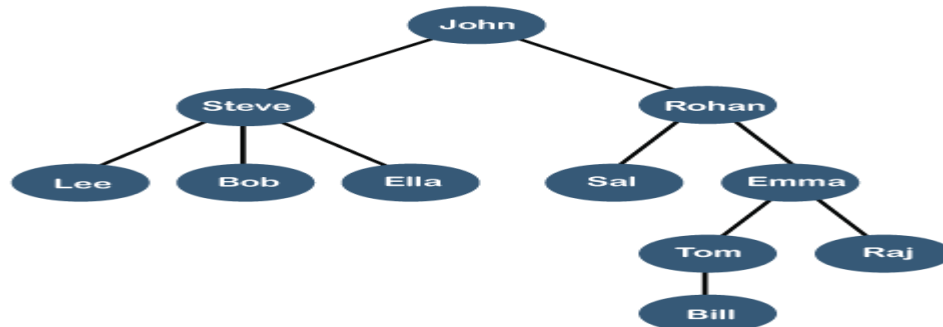
UNIT - II

Trees Part-I

Trees Part-I: Binary Search Trees: Definition and Operations, AVL Trees: Definition and Operations, Applications. B Trees: Definition and Operations.

TREE:

- A **tree** is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.

Each arrow shown in the above figure is known as a **link** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree**. If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an **internal**
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Applications of trees

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.

- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a $\log N$ time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

- General tree
- Binary tree
- Binary search tree
- AVL tree
- B-tree
- Red black tree
- Splay tree

Binary Tree VS Binary Search Tree(5 Marks)

Basis of Comparison	BINARY TREE	BINARY SEARCH TREE
Definition	BINARY TREE is a nonlinear data structure where each node can have at most two child nodes.	BST is a ordered binary tree that In a Binary search tree, the value of the left node must be smaller than the parent node, and the value of the right node must be greater than the parent node.
Types	<ul style="list-style-type: none"> • Full binary tree • Complete binary tree • Extended Binary tree and more 	<ul style="list-style-type: none"> • AVL tree • Splay Tree • T-trees and more
Operations	BINARY TREE is unordered hence slower in process of insertion, deletion, and searching.	Insertion, deletion, searching of an element is faster in BST than BINARY TREE due to the ordered characteristics
Structure	In BINARY TREE there is no ordering in terms of how the nodes are arranged	In BST the left subtree has elements less than the nodes element and the right subtree has elements greater than the nodes element.

Basis of Comparison	BINARY TREE	BINARY SEARCH TREE
Data Representation	Data Representation is carried out in a hierarchical format.	Data Representation is carried out in the ordered format.
Duplicate Values	Binary trees allow duplicate values.	Binary Search Tree does not allow duplicate values.
Complexity	Time complexity is usually $O(n)$.	Time complexity is usually $O(\log n)$.
Application	It is used for retrieval of fast and quick information and data lookup.	It works well at element deletion, insertion, and searching.
Usage	It serves as the foundation for implementing Full Binary Tree, BSTs, Perfect Binary Tree, and others.	It is utilized in the implementation of Balanced Binary Search Trees such as AVL Trees, Red Black Trees, and so on.

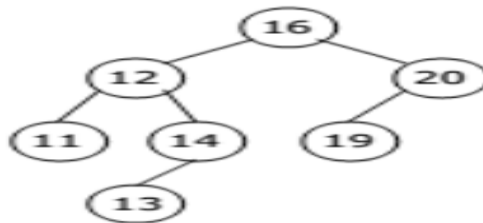
1. Binary Search Trees

1.1. What is a Binary Search Tree:

A binary search tree is a binary tree. It may be empty. If it is not empty, then for a set S , T is a labeled binary tree in which each node u is labeled by an element or key such that it satisfies the following properties:

1. The elements in the left sub-tree are smaller than the root.
2. The elements in the right sub-tree are larger than the root.
3. The left and right sub-trees are also binary search trees.

Example



Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

1.2. Operations on Binary Search Tree:

The following operations are performed on a binary search tree:

1. Search (5 Marks)
2. Insertion (5 Marks)
3. Deletion (5 Marks)

1.2.1. Search Operation:

- In a binary search tree, the search operation is performed with $O(\log n)$ average time complexity.
- The search operation is performed as follows:

Algorithm:

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display " Element is found" and terminate the function.

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left sub-tree.

Step 6 - If search element is larger, then continue the search process in right sub-tree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

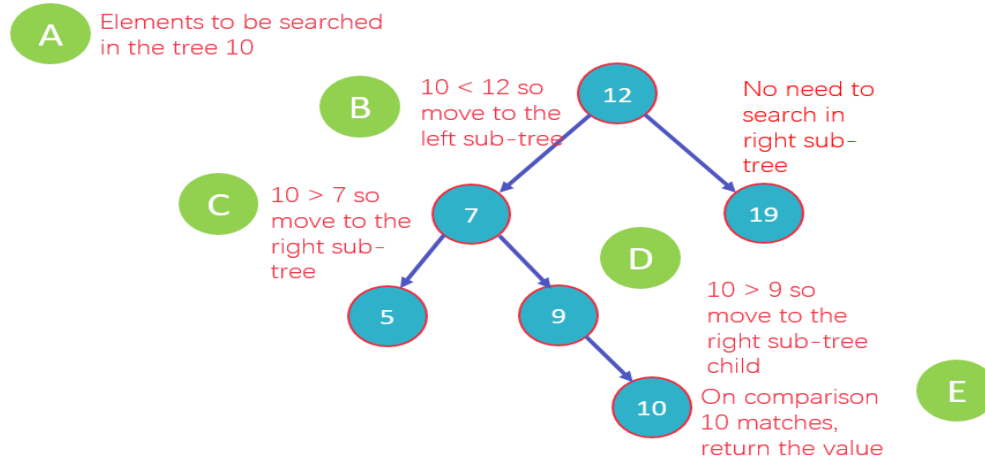
Pseudocode:

```
def search(root, value)
```

```
    if root.key == value then    //check if value is equal to the key val
    {
        print "The node is present"
        return
    }
    if value < root.key then
    {
        if root.left then search(root.left,value)
        else print "The node is empty in the tree!"
    }
    else:
    {
        if root.right then search(root.right,value)
        else print("\nThe node is empty in the tree!")
    }
}
```

Example:

Search Operation



- A. The element to be searched is 10
- B. Compare the element with the root node 12, $10 < 12$, hence you move to the left sub-tree. No need to analyze the right sub-tree
- C. Now compare 10 with node 7, $10 > 7$, so move to the right sub-tree
- D. Then compare 10 with the next node, which is 9, $10 > 9$, look in the right sub-tree child
- E. 10 matches with the value in the node, $10 = 10$, return the value to the user.

Why are binary search tree retrievals more efficient than sequential list retrievals:

- For a list of n elements stored as a sequential list, the worst case time complexity of searching an element both in the case of successful search or unsuccessful search is $O(n)$. This is so since in the worst case, the search key needs to be compared with every element of the list.
- However, in the case of a binary search tree, searching for a given key k results in discounting half the binary search tree at every stage of its comparison with a node on its path from the root downwards. The best case time complexity for the retrieval operation is therefore $O(1)$ when the search key k is found in the root itself. The worst case occurs when the search key is found in one of the leaf nodes whose level is equal to the height h of the binary search tree. The time complexity of the search is then given by $O(h)$.
- In some cases binary search trees may grow to heights that equal n , the number of elements in the associated set, thereby increasing the time complexity of a retrieval operation to $O(n)$ in the worst case

1.2.2 Insertion:

- In a binary search tree, the insertion operation is performed with $O(\log n)$ average time complexity.
- In binary search tree, new node is always inserted as a leaf node.
- The insertion operation is performed as follows:

Algorithm:

Step 1 - Create a newNode with given value and set its left and right to NULL.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is Empty, then set root to newNode.

Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).

Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.

Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

Pseudocode:

def insert(node, key):

```
// If the tree is empty, return a new node
if node is None then return Node(key)

// Otherwise recur down the tree
if key < node.key then node.left = insert(node.left, key)
else:
    node.right = insert(node.right, key)

//return the (unchanged) node pointer
return node
```

Example:

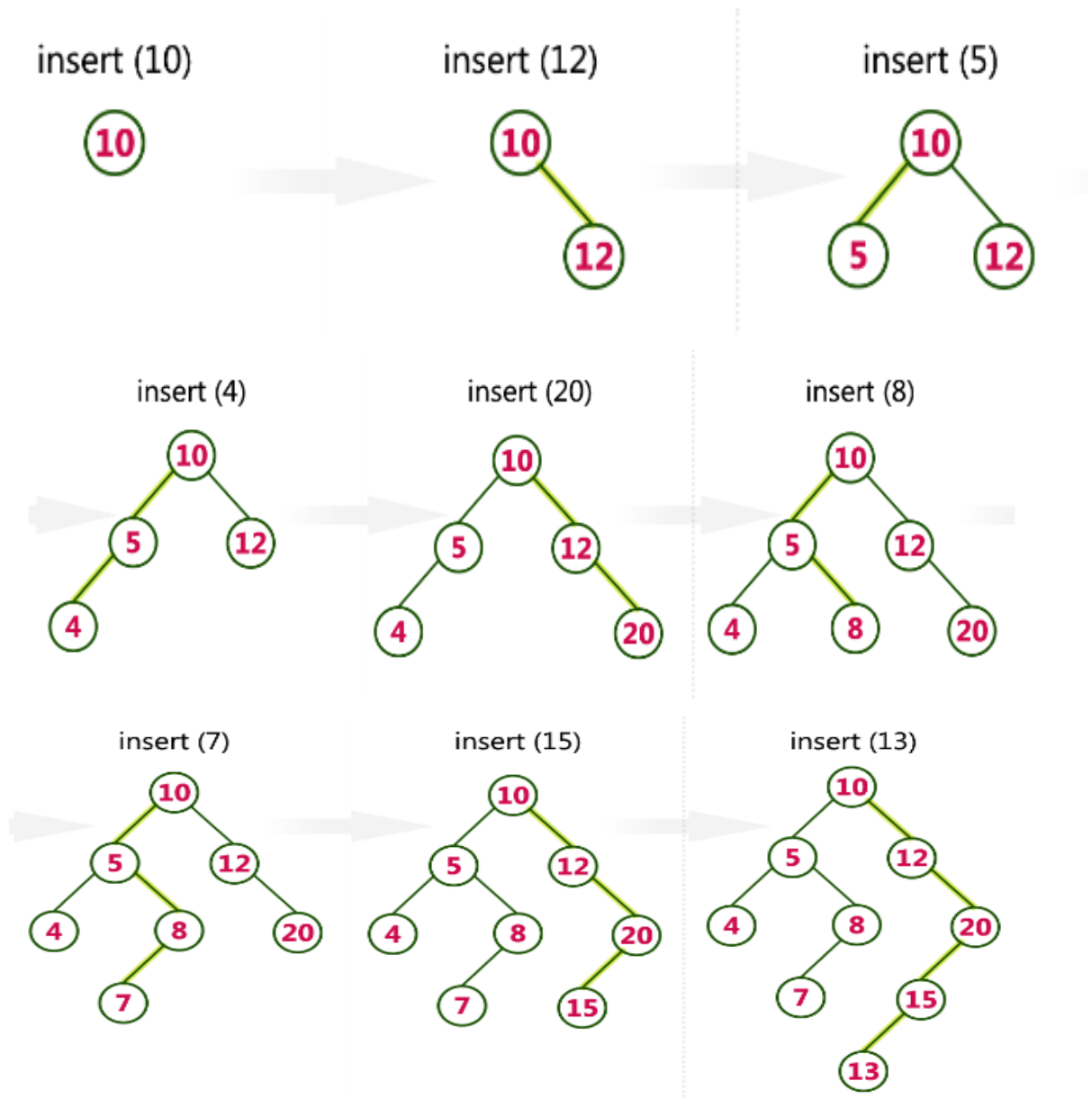
Construct a Binary Search Tree by inserting the following sequence of numbers...

10, 12, 5, 4, 20, 8, 7, 15 and 13

Solution:

10, 12, 5, 4, 20, 8, 7, 15 and 13

- Above elements are inserted into a Binary Search Tree as follows:



1.2.3 Deletion:

- In a binary search tree, the deletion operation is performed with $O(\log n)$ average time complexity.
- Deleting a node from Binary search tree includes following three cases:
Case 1: Deleting a Leaf node (A node with no children)
Case 2: Deleting a node with one child
Case 3: Deleting a node with two children

Algorithm:

- The algorithm steps of deletion operation in an BST tree are:
 1. Locate the node to be deleted
 2. There are three cases for deleting a node

1. If the node does not have any child, then remove the node
2. If the node has one child node, replace the content of the deletion node with the child node and remove the node
3. If the node has two children nodes, find the inorder successor node ' k^1 ' or inorder predecessor node ' k^2 ' and replace the contents of the deletion node with the ' k^1 ' or ' k^2 ' followed by removing the node.

Pseudocode:

def deleteNode(root, key):

 // Base Case

 if root is None then return root

 if key < root.key then root.left := deleteNode(root.left, key)

 elif(key > root.key) then root.right := deleteNode(root.right, key)
 else:

 //Node with only one child or no child

 if root.left is None:

 temp := root.right

 root := None

 return temp

 elif root.right is None:

 temp := root.left

 root := None

 return temp

 // Node with two children:

 // Get the inorder successor

 // (smallest in the right subtree)

 temp := minValueNode(root.right)

 # Copy the inorder successor's

 # content to this node

 root.key := temp.key

 # Delete the inorder successor

 root.right := deleteNode(root.right, temp.key)

return root

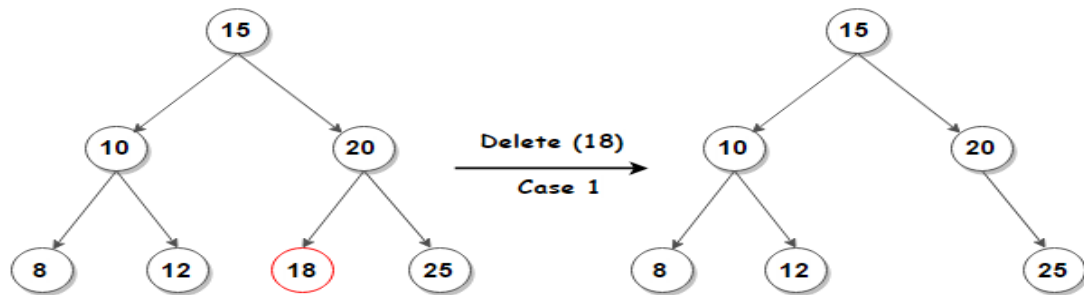
Case 1- Deleting a Leaf node (A node with no children):

We use the following steps to delete a leaf node from BST:

Step 1 - Find the node to be deleted using search operation.

Step 2 - Delete the node.

Example:



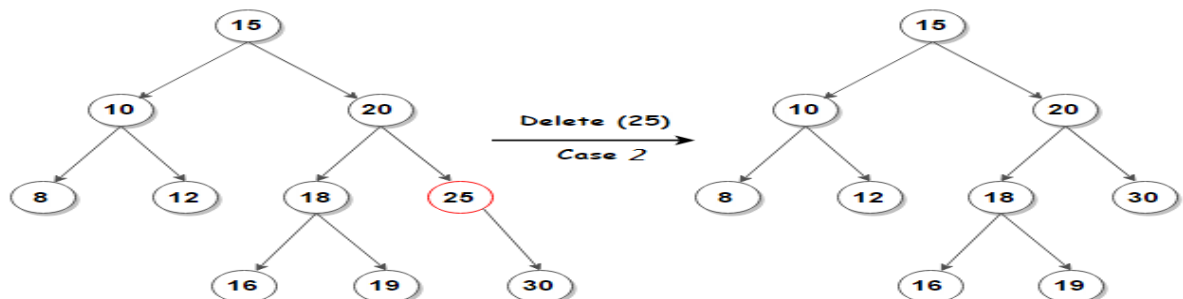
Case 2- Deleting a node with one child:

We use the following steps to delete a leaf node from BST:

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has only one child then replace the node with its child.

Example:



Case 3- Deleting a node with two children:

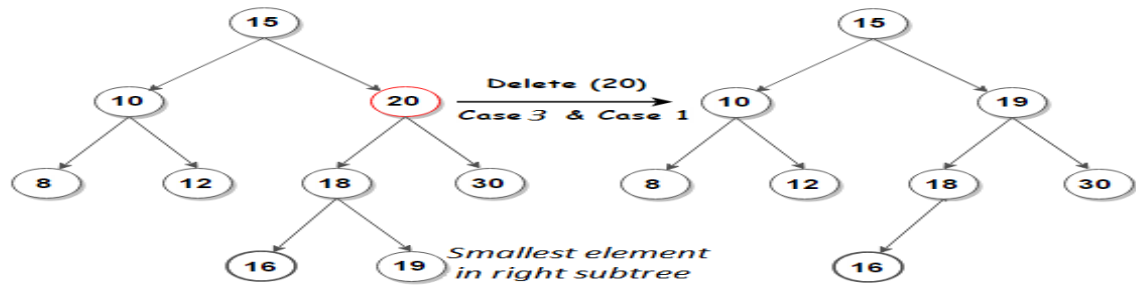
We use the following steps to delete a node with two children from BST:

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has two children, then find the largest node in its left sub-tree i.e predecessor (OR) the smallest node in its right sub-tree i.e successor.

Step 3 - Replace the deleting node with either successor or predecessor node

Example:



1.3. Applications of Binary Search Tree: (2 Marks)

1. BSTs are used for indexing and multi-level indexing.
2. They are also helpful to implement various searching algorithms.
3. It is helpful in maintaining a sorted stream of data.
4. Tree Map and Tree Set data structures are internally implemented using self-balancing BSTs

1.4. Advantages of Binary Search Tree: (2 Marks)

1. Time Complexity of insert, delete, search operations is to $O(\log N)$ where N is the number of nodes in the tree.
2. We have an ordering of keys stored in the tree. Any time we need to traverse the increasing (or decreasing) order of keys, we just need to do the in-order (and reverse in-order) traversal on the tree.
3. We can implement order statistics with binary search tree - N^{th} smallest, N^{th} largest element. This is because it is possible to look at the data structure as a sorted array.
4. We can also do range queries - find keys between N and M ($N \leq M$).
5. BST can also be used in the design of memory allocators to speed up the search of free blocks (chunks of memory), and to implement best fit algorithms where we are interested in finding the smallest free chunk with size greater than or equal to size specified in allocation request.

1.5. Limitations of Binary Search Tree: (2 Marks)

1. BST's have an average time complexity of $\Theta(\log(n))$ for inserting, deleting, and finding, but in the worst case, all of these operations have a time complexity of $O(n)$.
2. BST's forms skewed trees as follows:



3. In Skewed BST's, all operations are performed in $O(n)$ time

1.6. Skewed Binary Search Trees:

- A skewed binary search tree is a type of binary search tree in which all the nodes have only either one child or no child.
- In Skewed BST's, all operations are performed in $O(n)$ time.
- There are 2 types of skewed trees:

1. Left Skewed Tree:

In this all the nodes are having a left child or no child at all.

2. Right Skewed Tree:

In this all the nodes are having a right child or no child at all.



1.7. Time Complexity of BST: (2 Marks)

Operation	Best-Case	Average-Case	Worst-Case
Searching	$O(h)$	$O(h)$	$O(n)$
Insertion	$O(h)$	$O(h)$	$O(n)$
Deletion	$O(h)$	$O(h)$	$O(n)$
Where, h is height of the BST and n is number of nodes in BST			

****END****

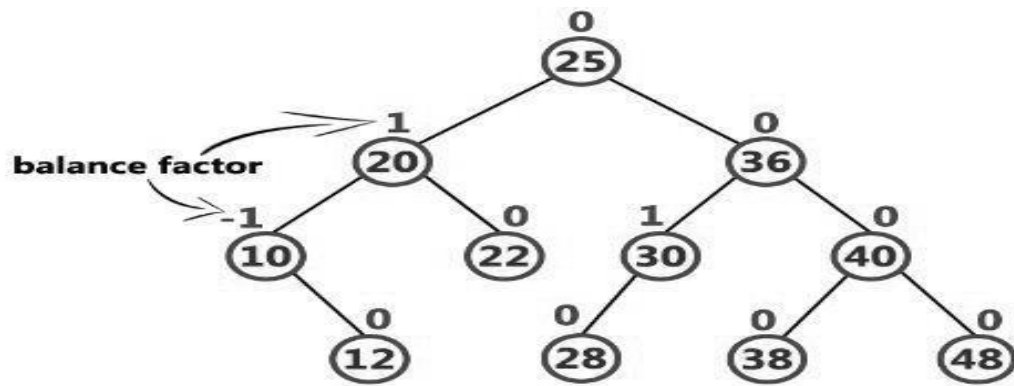
2. AVL Trees:

2.1. What is Height Balanced Trees?

- A tree which performs any operation in $O(\log n)$ in worst case, then it is called Height Balanced Tree.
- Example:**
AVL Tree, Red-Black Tree, Splay tree, B- Tree etc...,

2.2. What an AVL Tree? (2 Marks)

- AVL Tree is height balanced Binary Search Tree which performs all operations in $O(\log n)$.
- AVL Tree is invented by G.M. Adelson-Velski and E.M. Landis.
- Definition:**
An AVL Tree is a height balanced binary search tree, it may be empty. If it is not empty then every node has balance factor is either -1, 0, or +1.
- Balance Factor is calculated as:
Balance Factor = height of left sub tree – height of right sub tree
- Example:**



2.3. What is a rotation? Why rotations are required in AVL Tress? (2 Marks)

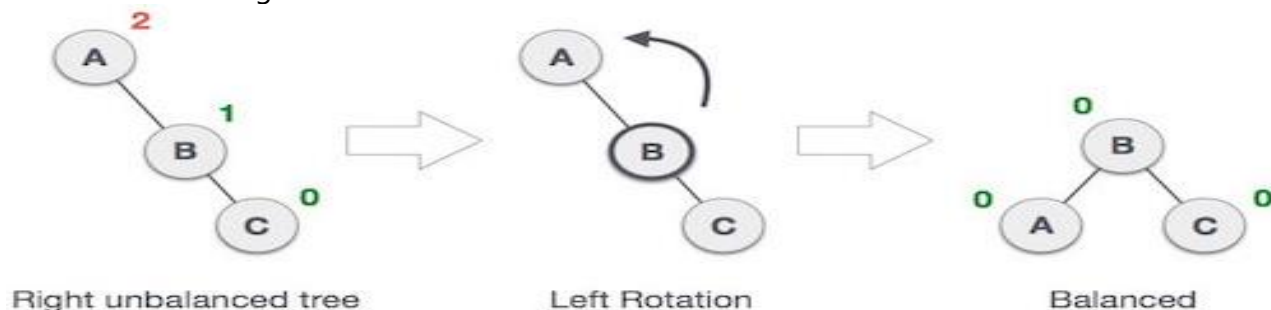
- Rotation is the process of moving nodes either to left or to right to make the tree balanced.
- In AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the tree is in balance, otherwise the tree is imbalance.
- Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.

2.4. List and Explain Different types of rotations in AVL Trees? (5 Marks)

- We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:
 1. L L rotation: Inserted node is in the left subtree of left subtree of A
 2. R R rotation : Inserted node is in the right subtree of right subtree of A
 3. L R rotation : Inserted node is in the right subtree of left subtree of A
 4. R L rotation : Inserted node is in the left subtree of right subtree of A
- Where node A is the node whose balance Factor is other than -1, 0, 1.
- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

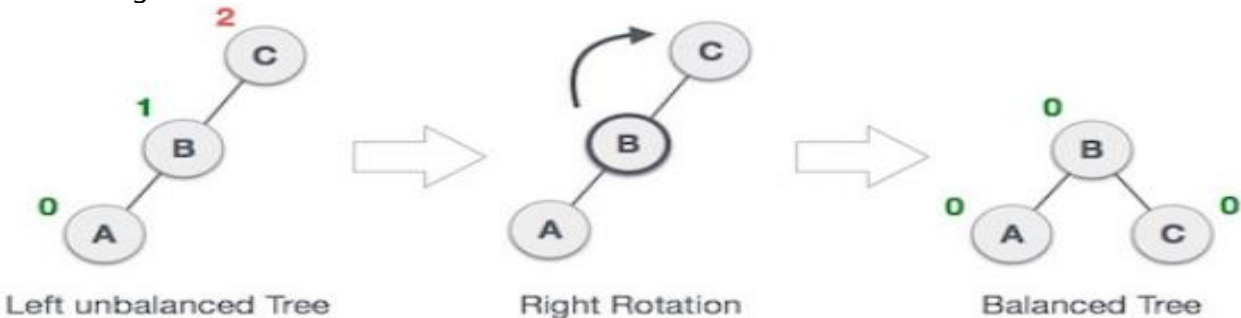
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

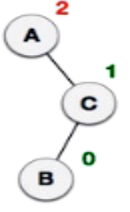
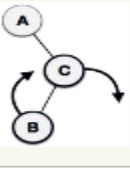
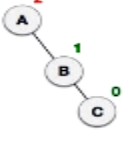
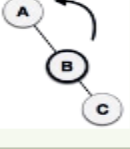
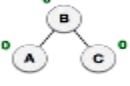
Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

Let us understand each and every step very clearly:

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A , has become the left subtree of B .
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or

State	Action
	<p>A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A</p>
	<p>As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.</p>
	<p>After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.</p>
	<p>Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.</p>

2.5.Operations on AVL Trees:

- The following operations are performed on AVL tree:
 - Search (5 Marks)
 - Insertion (5 Marks)
 - Deletion (5 Marks)

2.5.1. Search Operation:

- The Search Operation in AVL Trees is same as in BST.
- The search operation is performed as follows:

Algorithm:

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display " Element is found" and terminate the function.

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left sub-tree.

Step 6- If search element is larger, then continue the search process in right sub-tree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

2.5.2. Insert Operation:

- In AVL tree, new node is always inserted as a leaf node.
- The insertion operation is performed as follows:

Algorithm:

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 following **Rotations** to make it balanced and go for next operation.

1. If there is an imbalance in the left child's right sub-tree, perform a left-right rotation
2. If there is an imbalance in the left child's left sub-tree, perform a right rotation
3. If there is an imbalance in the right child's right sub-tree, perform a left rotation
4. If there is an imbalance in the right child's left sub-tree, perform a right-left rotation

Example refer from class notes

2.5.3. Delete Operation:

- The deletion operation in the AVL tree is the same as the deletion operation in BST. In the AVL tree, the node is always deleted as a leaf node and after the deletion of the node, the balance factor of each node is modified accordingly. Rotation operations are used to modify the balance factor of each node.
- The algorithm steps of deletion operation in an AVL tree are:
 3. Locate the node to be deleted
 4. There are three cases for deleting a node
 1. If the node does not have any child, then remove the node
 2. If the node has one child node, replace the content of the deletion node with the child node and remove the node
 3. If the node has two children nodes, find the inorder successor node 'k¹' or inorder predecessor node 'k²' and replace the contents of the deletion node with the 'k¹' or 'k²' followed by removing the node.
 5. After deletion update the balance factor of the AVL tree
 6. 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.

Rotations and Example refer from class notes

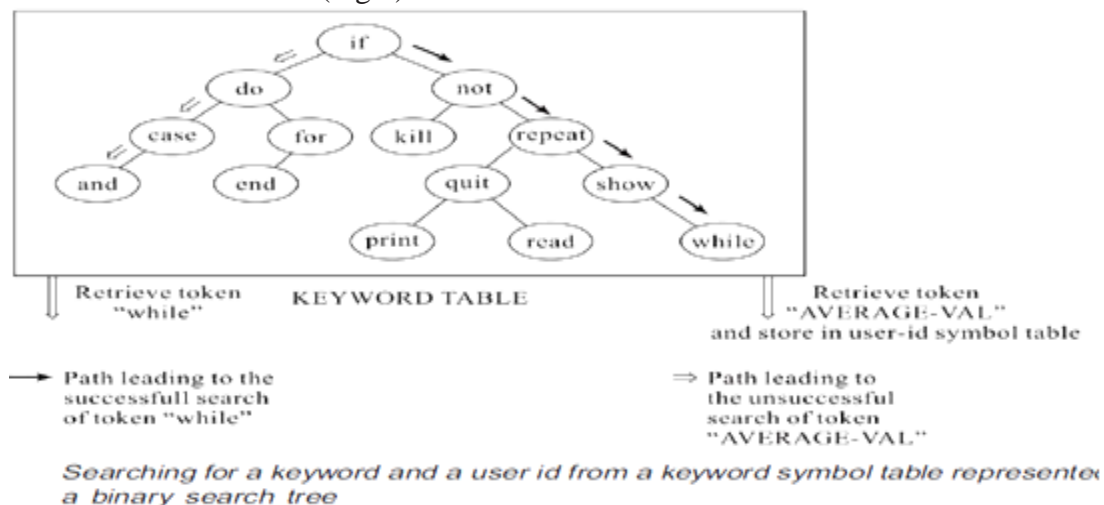
END

2.6.Applications of AVL Trees: (5 Marks)

1. AVL trees are mostly used for in-memory sorts of sets and dictionaries.
2. AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
3. It is used in applications that require improved searching apart from the database applications.
4. It is use mainly used in corporate sectors where the have to keep the information about the employees working there and their change in shifts.

2.6.1: Representation of symbol tables in compiler design:

- Compilers are translators that translate a source programming language code into a target programming language code viz., machine code or Assembly level language code.
- The various phases in the design of compilers include Lexical analysis, Syntactic analysis, Semantic analysis, Intermediate code generation, Code optimization and Code generation.
- During lexical analysis, which is the first phase of the compiler, the source program is scanned character by character to identify the keywords, user identifiers, constants, labels etc which are termed as tokens. These tokens are stored in data structures called symbol table.
- Thus there are individual symbol tables for keywords, user identifiers, constants etc.
- The optimal binary search tree may lead to inefficient retrievals due to the imbalance of nodes. It is in such a case that the application of AVL search trees becomes visible. Representing the keyword table as an AVL search tree ensures the retrieval of tokens in $O(\log n)$ time in the worst case.



2.7. Properties of AVL Trees:

1. Maximum possible number of nodes in AVL tree of height $H = 2^{H+1} - 1$.
2. Minimum number of nodes in AVL Tree of height H is given by a recursive relation:

$$N(H) = N(H-1) + N(H-2) + 1$$

Where, $N(0) = 1$ and $N(1) = 2$.

3. Minimum possible height of AVL Tree using N nodes = $\lfloor \log_2 N \rfloor$
4. Maximum height of AVL Tree using N nodes is calculated using recursive relation:

$$N(H) = N(H-1) + N(H-2) + 1$$

Where, $N(0) = 1$ and $N(1) = 2$.

Time Complexity: (2 Marks)

OPERATION	BEST CASE	AVERAGE CASE	WORST CASE
Insert	$O(\log n)$	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$
Search	$O(1)$	$O(\log n)$	$O(\log n)$
Traversal	$O(\log n)$	$O(\log n)$	$O(\log n)$

Space Complexity:

BEST CASE	AVERAGE CASE	WORST CASE
$O(n)$	$O(n)$	$O(n)$

*****END*****

3. **M-Way Trees:**

Refer from Class Notes

3.1. Advantages of M-Way Trees:

- M-way search trees give the same advantages to m-way trees that binary search trees gave to binary trees - they provide fast information retrieval and update.

3.2. Disadvantages of M-Way Trees:

- M-way trees have the same problems that binary search trees had - they can become unbalanced, which means that the construction of the tree becomes of vital importance.

*****END*****

4. B Trees:

4.1. Introduction:

- B-Tree is a height balanced M-way tree.
- The operations of B-Tree like insertion, deletion, search require $O(h)$, where h is height of B-Tree.

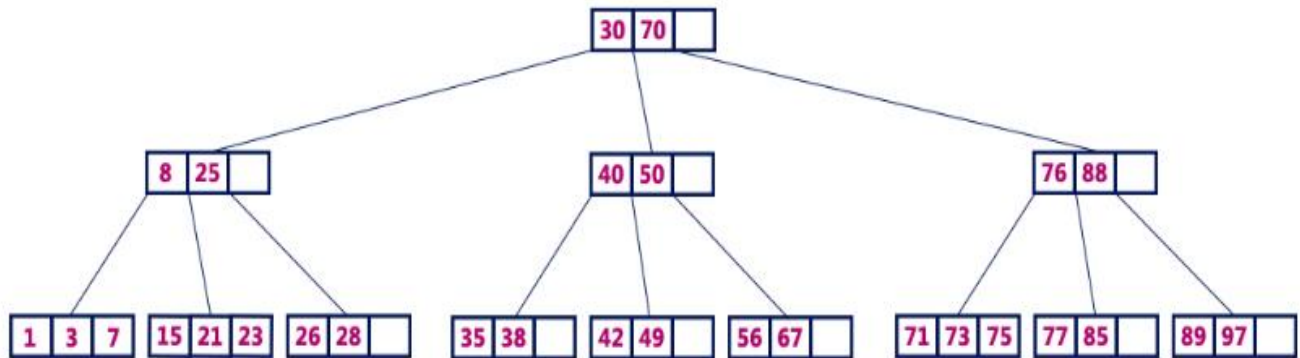
4.2. Definition: (2 Marks)

B-Tree of order m may be empty. If it is not empty, then it satisfies the below properties:

1. All leaf nodes must be at same level.
2. All nodes except root must have at least $\lceil m/2 \rceil - 1$ keys and maximum of $(m-1)$ keys.
3. All non leaf nodes except root (i.e. all internal nodes) must have at least $\lceil m/2 \rceil$ children.
4. If the root node is a non leaf node, then it must have at least 2 children.
5. A non leaf node with $n-1$ keys must have n number of children.
6. All the key values in a node must be in Ascending Order.

Example:

B-Tree of Order 4



4.3. Operations on B Tree:

- The following operations are performed on a B-Tree:
 1. Search (5 Marks)
 2. Insertion (5 Marks)
 3. Deletion (5 Marks)

4.3.1. Search Operation:

- The search operation in B-Tree is similar to the search operation in Binary Search Tree.
- In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree).

- In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has.
- The search operation is performed as follows:

Algorithm:

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with first key value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

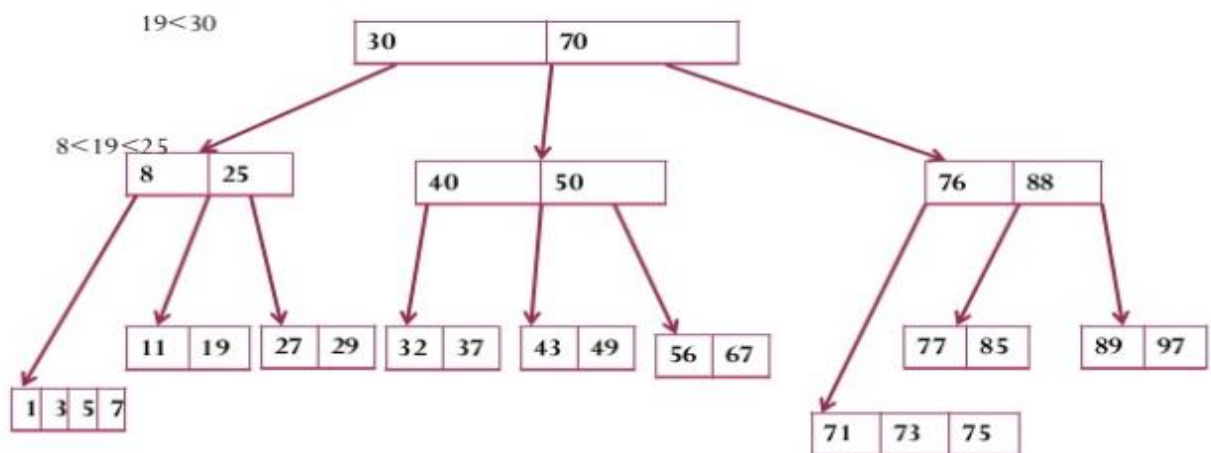
Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left sub-tree.

Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

• **Example:**



Suppose we want to search for 19. Searching will start from the root node so first look at node [30 70] , the key is not there & since $19 < 30$, we move to leftmost child of root which is [8 25]. The key is not present in this node also 19 lies between 8 and 25 so we move to node [11 19] where we get the desired key.

4.3.2. Insert Operation:

- In a B-Tree, a new element must be added only at the leaf node.
- The insertion operation is performed as follows:

Algorithm:

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example:

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



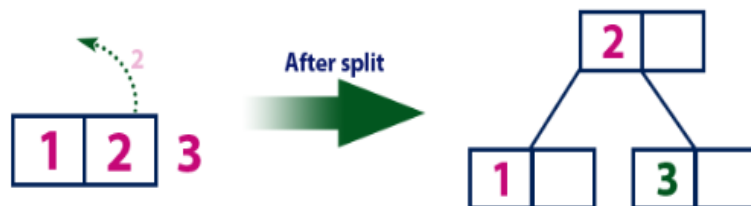
insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



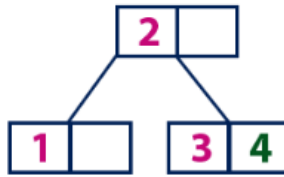
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't have an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't have a parent. So, this middle value becomes a new root node for the tree.



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



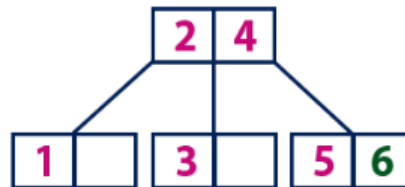
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



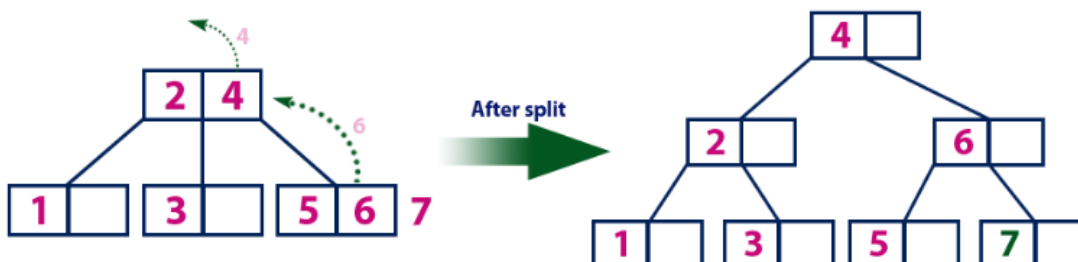
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



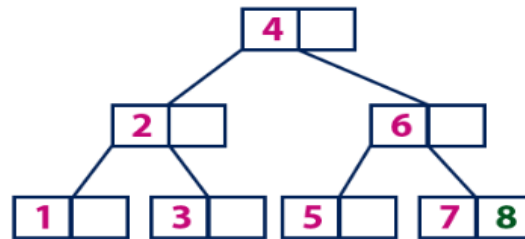
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



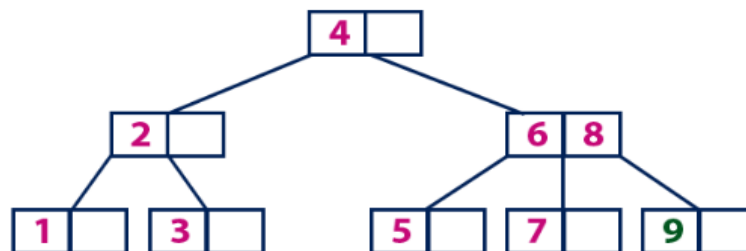
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



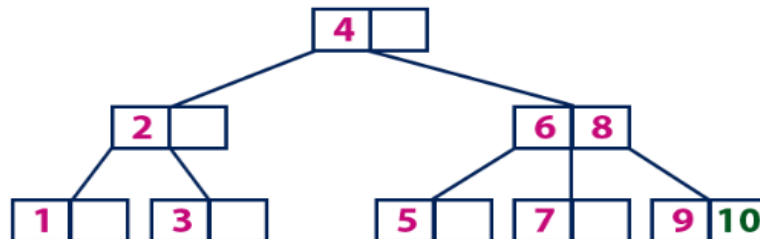
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8'. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



4.3.3. Delete Operation:

- The delete operation has more rules than insert and search operations.
- The Deletion operation as follows: :

Algorithm:

Step-1: Run the search operation and find the target key in the nodes.

Note: Three conditions applied based on the location of the target key, as explained in the following steps.

Step-2: If the target key is in the leaf node

Case-1: Target is in the leaf node, more than min keys.

- Deleting this will not violate the property of B Tree

Case-2: Target is in leaf node, it has min key nodes.

- Deleting this will violate the property of B Tree.
- Target node can borrow key from immediate left node, or immediate right node (sibling)
- The sibling will say yes if it has more than minimum number of keys
- The key will be borrowed from the parent node, the max value will be transferred to a parent, the max value of the parent node will be transferred to the target node, and remove the target value

Case-3: Target is in the leaf node, but no siblings have more than min number of keys

- In such cases target node is merge with either left siblings or right siblings along with intermediate element from parent

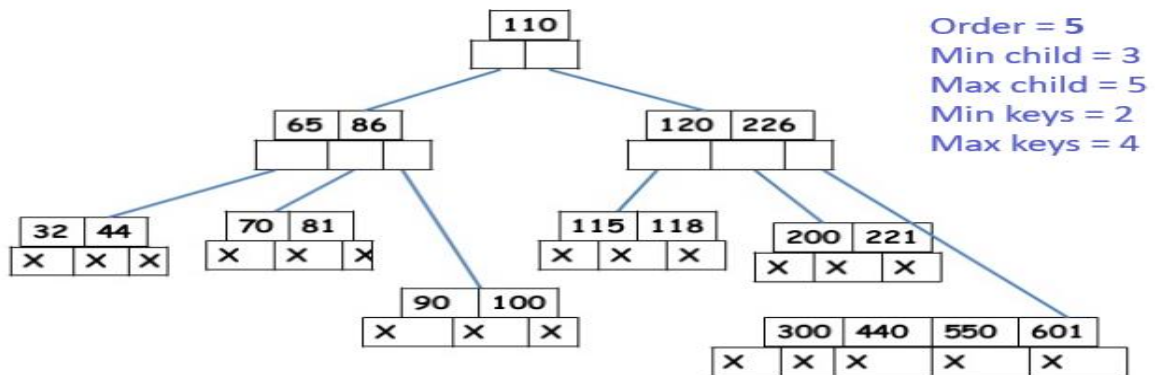
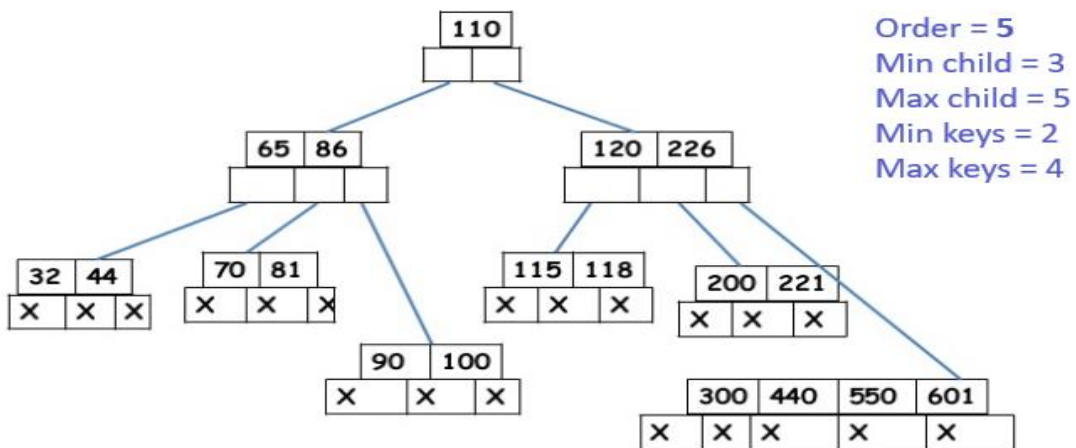
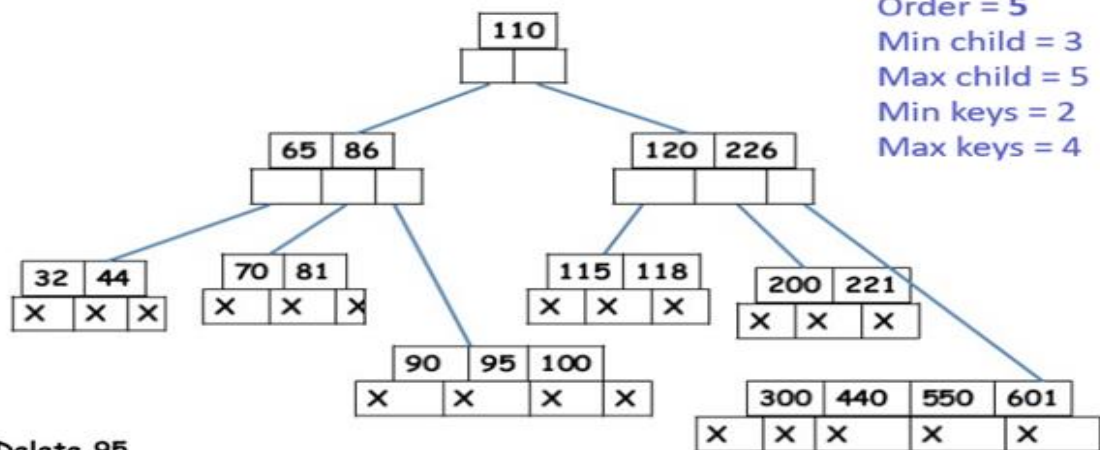
Step-3: If the target key is in an internal node

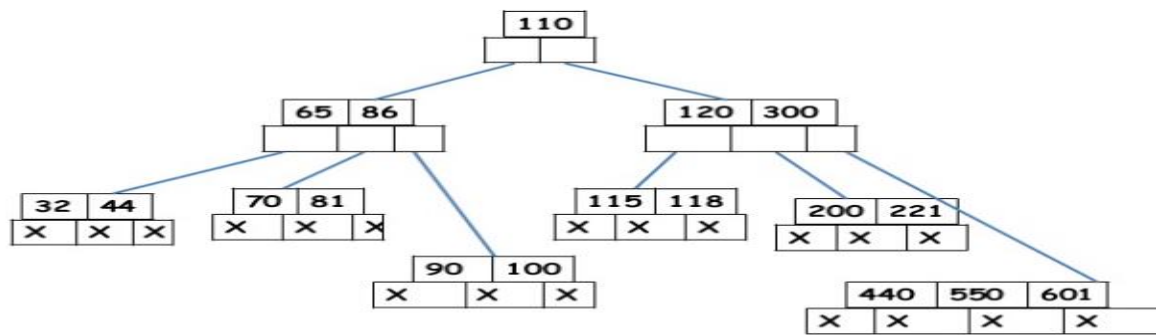
- Either choose, in- order predecessor or in-order successor
- In case the of in-order predecessor, the maximum key from its left subtree will be selected
- In case of in-order successor, the minimum key from its right subtree will be selected
- If the target key's in-order predecessor has more than the min keys, only then it can replace the target key with the max of the in-order predecessor
- If the target key's in-order predecessor does not have more than min keys, look for in-order successor's minimum key.
- If the target key's in-order predecessor and successor both have less than min keys, then merge the predecessor node and successor node.

Step-4: If the target key is in a root node

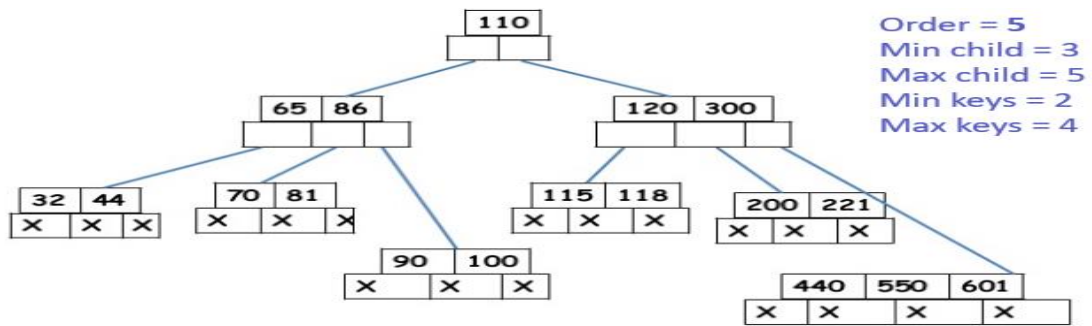
- Replace with the maximum element of the in-order predecessor subtree and Min element of inoder successor subtree
- In-order predecessor subtree and Inoder successor subtree has min keys then merge the root node element with either left child or right child

Example:

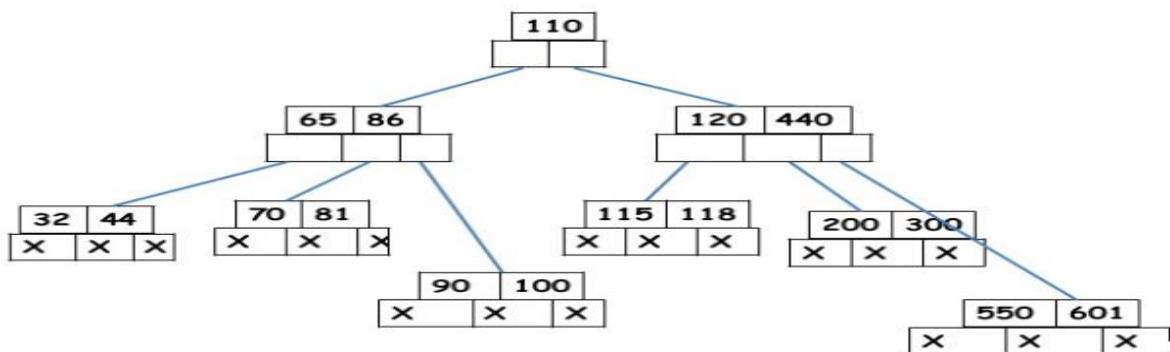




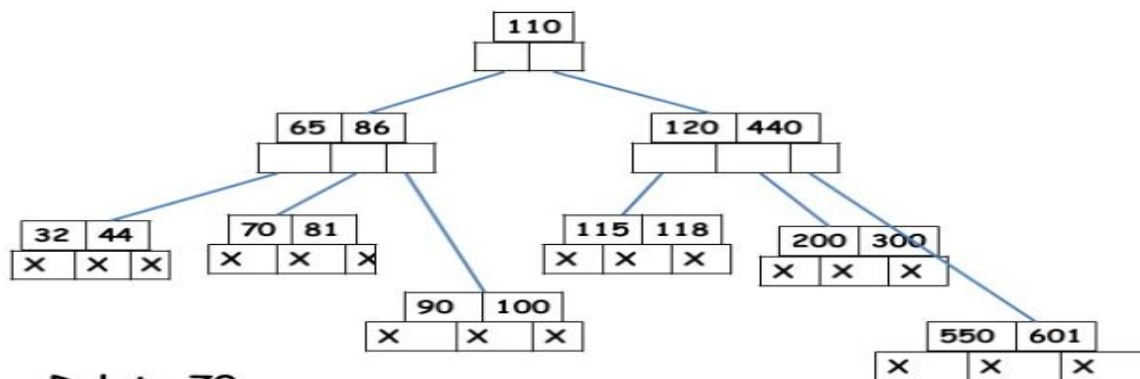
B-tree after deleting 226



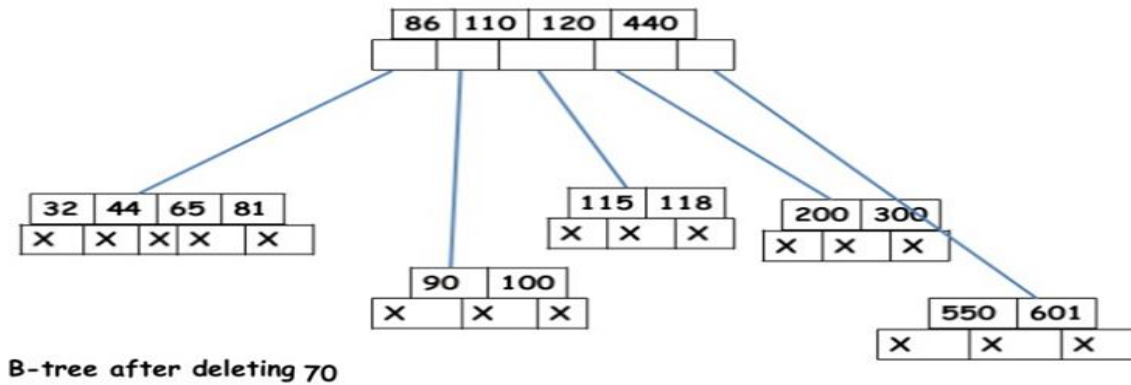
Delete 221



B-tree after deleting 221



Delete 70



4.3.4. Advantages of B Tree: (2 Marks)

1. Reduces the number of reads made on the disk
2. B Trees can be easily optimized to adjust its size (that is the number of child nodes) according to the disk size
3. It is a specially designed technique for handling a bulky amount of data.
4. It is a useful algorithm for databases and file systems.
5. A good choice to opt when it comes to reading and writing large blocks of data.

4.3.5. Disadvantages of B Trees (2 Marks)

1. B-tree has lower branching factor when compared to b+ tree, which increases tree height and average number of disk seeks.
2. B-trees are not very efficient, when compared to other balanced trees, when they reside in RAM.
3. B-tree is much more complicated than a hash table to implement.

4.3.6. Applications of B Trees: (2 Marks)

1. **Database or File System** - Consider having to search details of a person in Yellow Pages (directory containing details of professionals). Such a system where you need to search a record among millions can be done using B Tree.
2. **Search Engines and Multilevel Indexing** - B tree Searching is used in search engines and also querying in MySql Server.
3. **To store blocks of data** - B Tree helps in faster storing blocks of data in secondary storage due to the balanced structure of B Tree.

4.3.7. 4.3.7 Why Do You Need B Tree Data Structure? (2 Marks)

- The B tree data structure is needed due to the following reasons:
 - B Tree is an extension of M-way tree. While B trees are self-balanced, M way trees can be balanced, skewed or any way. In case of external storage, there is a need for faster access. An M-way tree can help ease searches for external storage more efficiently than a normal Binary Search Tree. However, due to the self-balancing nature of a B tree, it had fewer levels and thus the access-time is cut short to a huge extent by a B Tree.

- A B tree facilitates ordered sequential access and simplifies insertion and deletion of records when there are millions of records. This is possible due to the reduced height and balanced nature of the B tree

4.3.8. **Time complexity** (2 Marks)

- The time complexity for insertion, deletion, and search operations takes $O(\log n)$ time where n is the number of keys stored in the tree.
- **Space complexity**
 - The space complexity for a B-Tree is $O(n)$, where n is the number of keys in the tree.