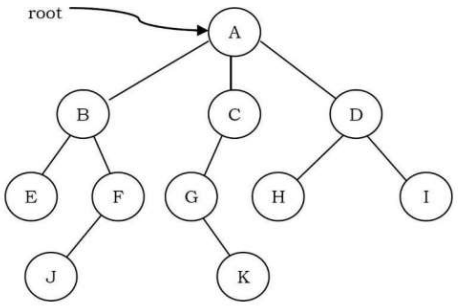
DS-Trees-Unit-III

19 April 2025 22:30

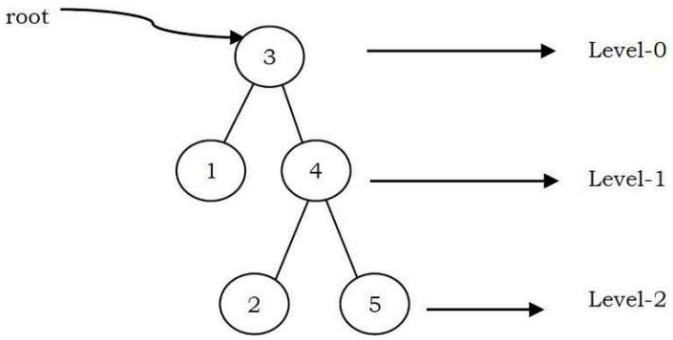
What is a Tree?

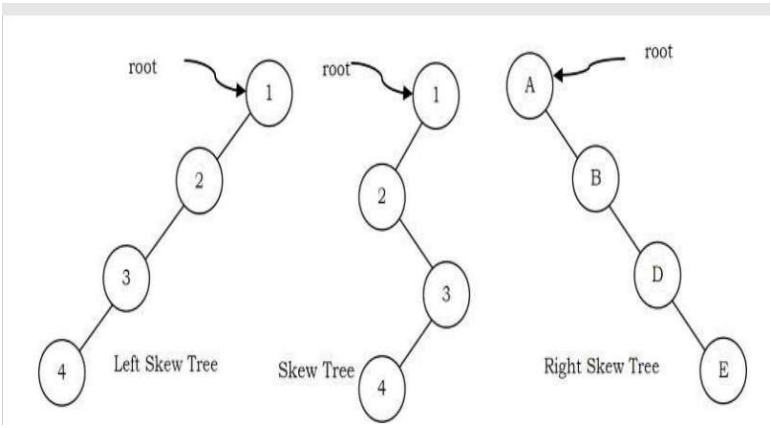
A tree is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes.
Tree is an example of a non-linear data structure. A tree structure is a way of representing the hierarchical nature of a structure in a graphical form.
In trees ADT (Abstract Data Type), the order of the elements is not important.



• The root of a tree is the node with no parents. There can be at most one root node in a tree (node A in the above example).
• An edge refers to the link from parent to child (all links in the figure).
• A node with no children is called leaf node (E,J,K,H and I).
• Children of same parent are called siblings (B,C,D are siblings of A, and E,F are the siblings of B).
• A node p is an ancestor of node q if there exists a path from root to q and p appears on the path. The node q is called a descendant of p. For example, A,C and G are the ancestors of if.
• The set of all nodes at a given depth is called the level of the tree (B, C and D are the same level). The root node is at level zero.

• The depth of a node is the length of the path from the root to the node (depth of G is 2, A – C – G).
• The height of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the height of B is 2 (B – F – J).
• Height of the tree is the maximum height among all the nodes in the tree and depth of the tree is the maximum depth among all the nodes in the tree. For a given tree, depth and height returns the same value. But for individual nodes we may get different results.
• The size of a node is the number of descendants it has including itself (the size of the subtree C is 3).
• If every node in a tree has only one child (except leaf nodes) then we call such trees skew trees. If every node has only left child then we call them left skew trees. Similarly, if every node has only right child then we call them right skew trees.





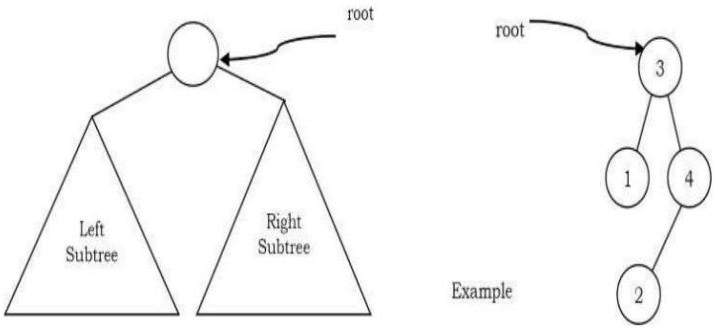
==========

Binary Trees:

A tree is called binary tree if each node has zero child, one child or two children.

Empty tree is also a valid binary tree. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

Generic Binary Tree



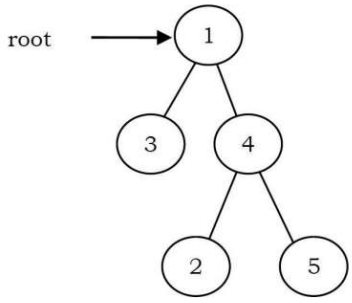
Types of Binary Trees:

---

1. Strict Binary tree
2. Full Binary tree
3. Complete Binary tree

Strict Binary Tree:

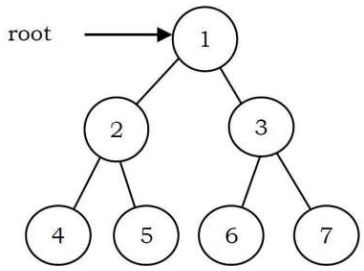
A binary tree is called strict binary tree if each node has exactly two children or no children.



Screen clipping taken: 19-04-2025 23:41

Full Binary Tree:

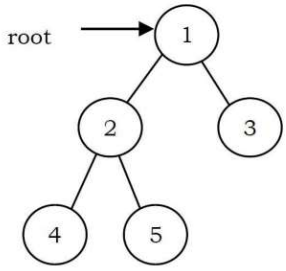
A binary tree is called full binary tree if each node has exactly two children and all leaf nodes are at the same level.



Screen clipping taken: 19-04-2025 23:42

Complete Binary Tree:

Before defining the complete binary tree, let us assume that the height of the binary tree is h. In complete binary trees, if we give numbering for the nodes by starting at the root (let us say the root node has 1) then we get a complete sequence from 1 to the number of nodes in the tree. While traversing we should give numbering for NULL pointers also. A binary tree is called complete binary tree if all leaf nodes are at height h or h – 1 and also without any missing number in the sequence



Screen clipping taken: 19-04-2025 23:43

Binary Tree Representation

A binary tree can be represented mainly in 2 ways:

a) Sequential Representation

b) Linked Representation

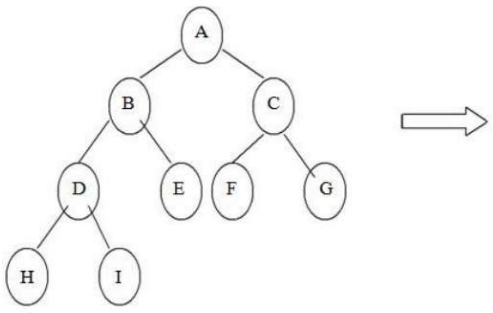
a) Sequential Representation

The simplest way to represent binary trees in memory is the sequential representation that uses one dimensional array.

1) The root of binary tree is stored in the 1 st location of array

2) If a node is in the jth location of array, then its left child is in the location 2J+1 and its right child in the location 2J+2

The maximum size that is required for an array to store a tree is 2d+1 -1, where d is the depth of the tree.



POSITION	ARRAY
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
.	.
.	.
.	.
.	.
.	.

Screen clipping taken: 20-04-2025 16:34

Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular node is fast because of the random access.

Disadvantages of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory. For example in the skewed tree half of the array is unutilized.
2. In this type of representation the maximum depth of the tree has to be fixed. Because we have decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we coose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.
3. The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate positions so that the meaning of binary tree can bepreserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

Screen clipping taken: 20-04-2025 16:35

b) Linked Representation

Linked representation of trees in memory is implemented using pointers. Since each node in a binary tree can have maximum two children, a node in a linked representation has two pointers for both left and right child, and one information field. If a node does not have any child, the corresponding pointer field is made NULL pointer.

In linked list each node will look like this:

Left Child	Data	Right Child

Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.

2. Insertions and deletions which are the most common operations can be done without moving the nodes.

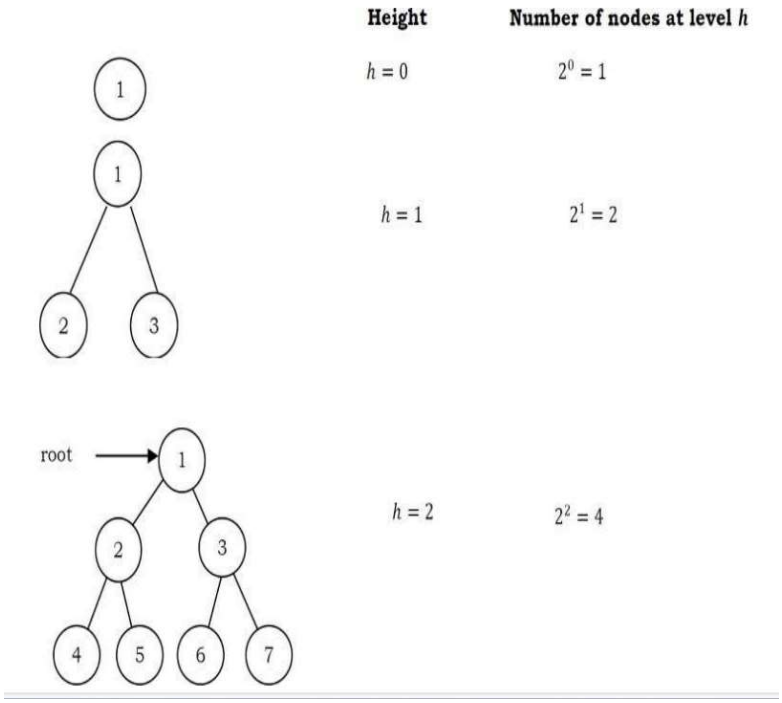
Disadvantages of linked representation:

1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.

Properties of Binary Trees:

---

1. For the following properties, let us assume that the height of the tree is h. Also, assume that root node is at height zero



Screen clipping taken: 20-04-2025 07:05

From the diagram we can infer the following properties:

- The number of nodes n in a full binary tree is $2^{h+1} - 1$. Since, there are h levels we need to add all nodes at each level $[2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1]$.
- The number of nodes n in a complete binary tree is between 2^h (minimum) and $2^{h+1} - 1$ (maximum). For more information on this, refer to *Priority Queues* chapter.
- The number of leaf nodes in a full binary tree is 2^h.
- The number of NULL links (wasted pointers) in a complete binary tree of n nodes is $n + 1$.

Screen clipping taken: 20-04-2025 07:07

Operations on Binary Trees
Basic Operations
• Inserting an element into a tree
• Deleting an element from a tree
• Searching for an element
• Traversing the tree
Auxiliary Operations
• Finding the size of the tree
• Finding the height of the tree
• Finding the level which has maximum sum
• Finding the least common ancestor (LCA) for a given pair of nodes, and many more.

Applications of Binary Trees
Following are the some of the applications where binary trees play an important role:
• Expression trees are used in compilers.
• Huffman coding trees that are used in data compression algorithms.
• Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items in O(logn) (average).
• Priority Queue (PQ), which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (in worst case).

Binary Tree Traversals :

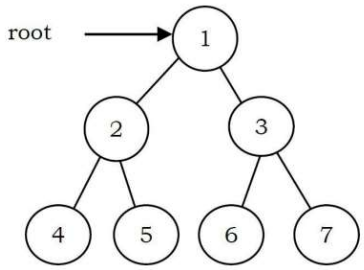
. The process of visiting all nodes of a tree is called tree traversal.
Each node is processed only once but it may be visited more than once. In tree structures there are many different ways.
Traversal Possibilities Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type.

These steps are:
performing an action on the current node (referred to as “visiting” the node and denoted with “D”),
traversing to the left child node (denoted with “L”), and
traversing to the right child node (denoted with “R”).

1. LDR: Process left subtree, process the current node data and then process right subtree
2. LRD: Process left subtree, process right subtree and then process the current node data
3. DLR: Process the current node data, process left subtree and then process right subtree
4. DRL: Process the current node data, process right subtree and then process left subtree
5. RDL: Process right subtree, process the current node data and then process left subtree
6. RLD: Process right subtree, process left subtree and then process the current node data

Classifying the Traversals:
---
if we are classifying based on current node (D) and if D comes in the middle then it does not matter whether L is on left side of D or R is on left side of D. Similarly, it does not matter whether L is on right side of D or R is on right side of D. Due to this, the total 6 possibilities are reduced to 3 and these are:

- Preorder (DLR) Traversal
- Inorder (LDR) Traversal
- Postorder (LRD) Traversal



Preorder traversal is defined as follows:
- Visit the root.
- Traverse the left subtree in Preorder.
- Traverse the right subtree in Preorder.

The nodes of tree would be visited in the order: 1 2 4 5 3 6 7
Time Complexity: O(n). Space Complexity: O(n).

InOrder Traversal
In Inorder Traversal the root is visited between the subtrees. Inorder traversal is defined as follows:
- Traverse the left subtree in Inorder.
- Visit the root.
- Traverse the right subtree in Inorder.

The nodes of tree would be visited in the order: 4 2 5 1 6 3 7
Time Complexity: O(n). Space Complexity: O(n)

PostOrder Traversal
In postorder traversal, the root is visited after both subtrees. Postorder traversal is defined as follows:
- Traverse the left subtree in Postorder.
- Traverse the right subtree in Postorder.
- Visit the root.

The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1
Time Complexity: O(n). Space Complexity: O(n).

Level Order Traversal
Level order traversal is defined as follows:
- Visit the root.
- While traversing level (, keep all the elements at level (+ 1 in queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.

The nodes of the tree are visited in the order: 1 2 3 4 5 6 7

==========

Binary Search Trees (BSTs):

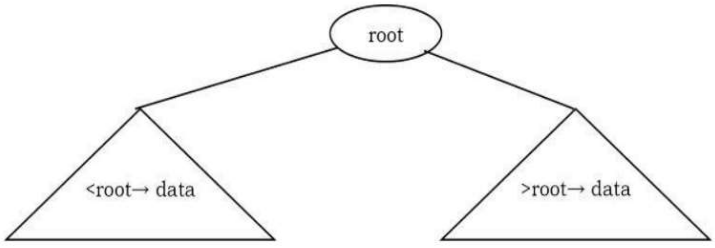
As the name suggests, the main use of this representation is for searching. In this representation we impose restriction on the kind of data a node can contain. As a result, it reduces the worst case average search operation to O(logn).

So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at left sub-tree < root node value < right sub-tree values.

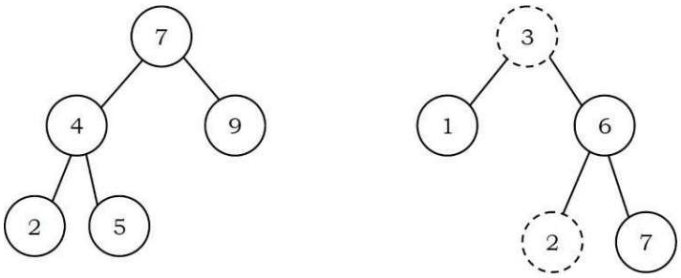
Binary Search Tree Property:

In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called binary search tree property. Note that, this property should be satisfied at every node in the tree.

- The left subtree of a node contains only nodes with keys less than the nodes key.
- The right subtree of a node contains only nodes with keys greater than the nodes key.
- Both the left and right subtrees must also be binary search trees.



Example: The left tree is a binary search tree and the right tree is not a binary search tree (at node 6 it's not satisfying the binary search tree property).



Screen clipping taken: 20-04-2025 15:58

Binary Search Tree Declaration:

There is no difference between regular binary tree declaration and binary search tree declaration. The difference is only in data but not in structure. But for our convenience we change the structure name as:

Ex:

```
Struct BinarySearchTreeNode{
   int data
   struct BinarySearchTreeNode *left;
   struct BinarySearchTreeNode *right;
};
```

Operations on Binary Search Trees:

Main operations: Following are the main operations that are supported by binary search trees:

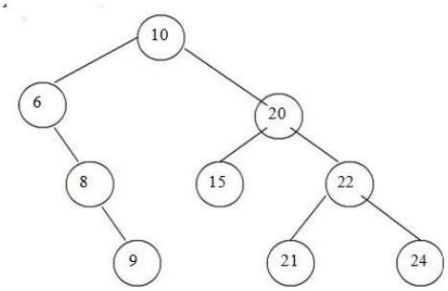
- Find/ Find Minimum / Find Maximum element in binary search trees
- Inserting an element in binary search trees
- Deleting an element from binary search trees

Auxiliary operations: Checking whether the given tree is a binary search tree or not

- Finding kth -smallest element in tree
- Sorting the elements of binary search tree and many more

Insertion of a node in binary search tree.

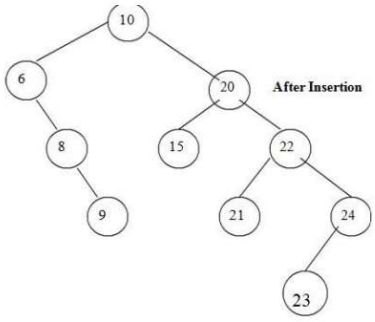
While inserting any node in binary search tree, look for its appropriate position in the binary search tree. We start comparing this new node with each node of the tree. If the value of the node which is to be inserted is greater than the value of the current node we move on to the right sub-branch otherwise we move on to the left sub-branch. As soon as the appropriate position is found we attach this new node as left or right child appropriately.



Before Insertion

In the above fig, if we wan to insert 23. Then we will start comparing 23 with value of root node i.e. 10. As 23 is greater than 10, we will move on right sub-tree. Now we will compare 23 with 20 and move right, compare 23 with 22 and move right. Now compare 23 with 24 but it is less than 24. We will move on left branch of 24. But as there is node as left child of 24, we can attach 23 as left child of 24.

Screen clipping taken: 20-04-2025 16:50



Deletion of a node from binary search tree.

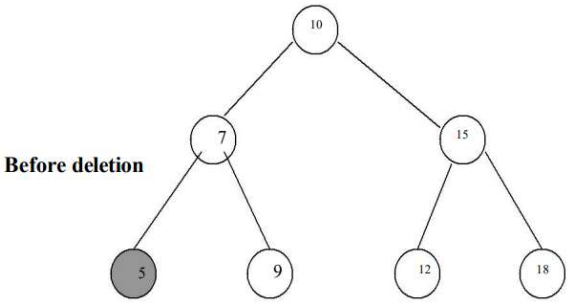
For deletion of any node from binary search tree there are three which are possible.

i. Deletion of leaf node.
ii. Deletion of a node having one child.
iii. Deletion of a node having two children.

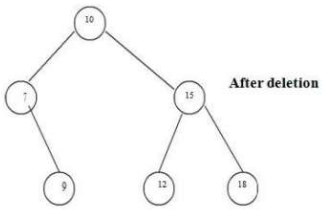
Deletion of leaf node.

This is the simplest deletion, in which we set the left or right pointer of parent node as NULL.

Screen clipping taken: 20-04-2025 16:51



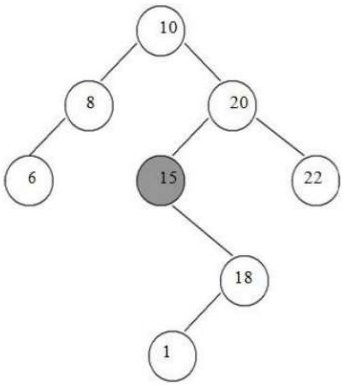
From the above fig, we want to delete the node having value 5 then we will set left pointer of its parent node as NULL. That is left pointer of node having value 7 is set to NULL.



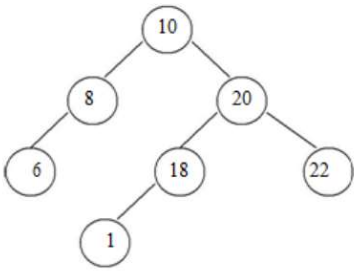
Screen clipping taken: 20-04-2025 16:51

Deletion of a node having one child.

To explain this kind of deletion, consider a tree as given below.

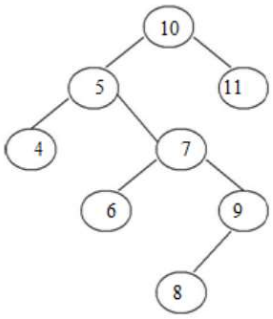


If we want to delete the node 15, then we will simply copy node 18 at place of 16 and then set the node free



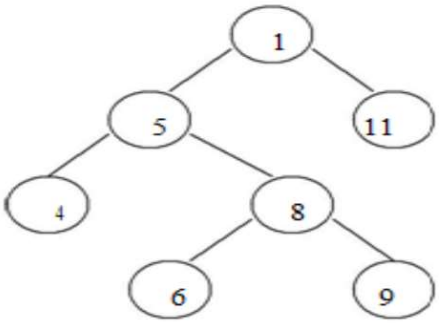
Screen clipping taken: 21-04-2025 17:01

Deletion of a node having two children.
Consider a tree as given below.



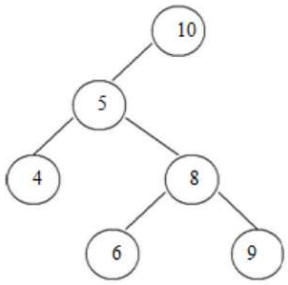
Let us consider that we want to delete node having value 7. We will then find out the inorder successor of node 7. We will then find out the inorder successor of node 7. The inorder successor will be simply copied at location of node 7.

That means copy 8 at the position where value of node is 7. Set left pointer of 9 as NULL. This completes the deletion procedure.



Searching for a node in binary search tree.

In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub branch otherwise on left sub branch. If we reach to leaf node and still we do not get the value of key node then we declare "node is not present in the tree".



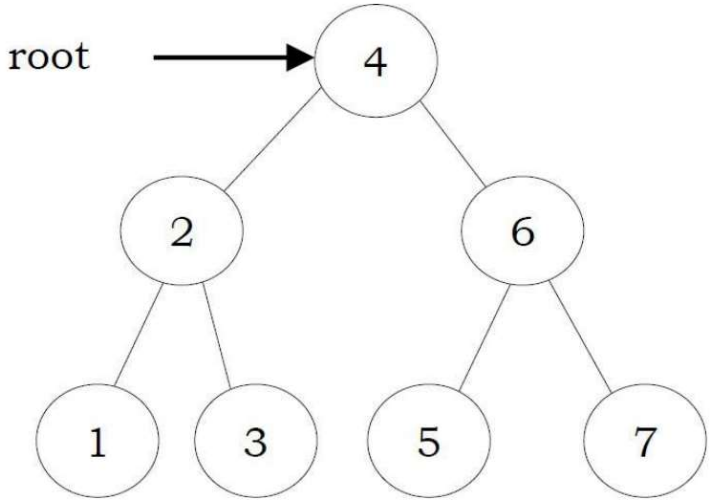
In the above tree, if we want to search for value 9. Then we will compare 9 with root node 10. As 9 is less than 10 we will search on left sub branch. Now compare 9 with 5, but 9 is greater than 5. So we will move on right sub tree. Now compare 9 with 8 but 9 is greater than 8 we will move on right sub branch. As the node we will get holds the value 9. Thus the desired node can be searched.

==========

Balanced Binary Search Trees
In earlier sections we have seen different trees whose worst case complexity is O(n), where n is the number of nodes in the tree. This happens when the trees are skew trees. In this section we will try to reduce this worst case complexity to O(logn) by imposing restrictions on the heights.
In general, the height balanced trees are represented with HB(k), where k is the difference between left subtree height and right subtree height. Sometimes k is called balance factor

Full Balanced Binary Search Trees :

In HB(k), if k = 0 (if balance factor is zero), then we call such binary search trees as full balanced binary search trees. That means, in HB(0) binary search tree, the difference between left subtree height and right subtree height should be at most zero. This ensures that the tree is a full binary tree. For example,



Screen clipping taken: 29-04-2025 10:48

6.13 AVL (Adelson-Velskii and Landis) Trees

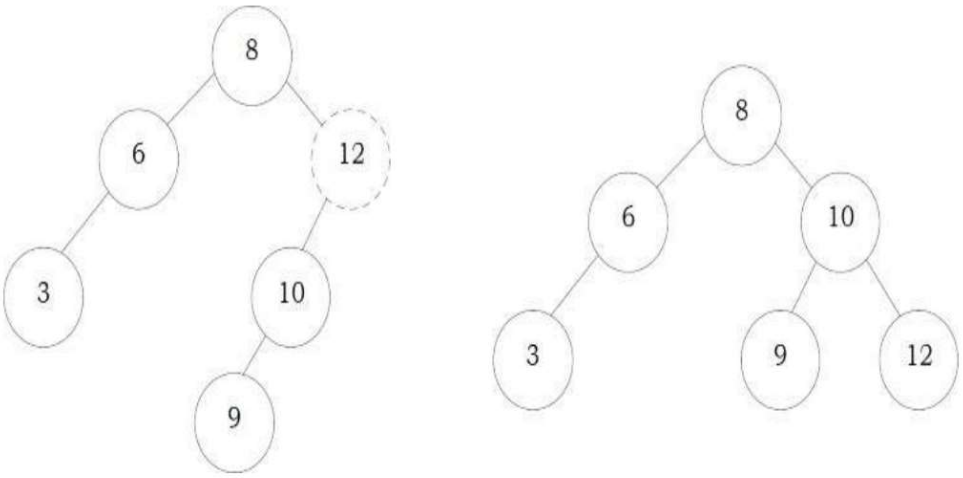
In *HB*(*k*), if *k* = 1 (if balance factor is one), such a binary search tree is called an *AVL tree*. That means an AVL tree is a binary search tree with a *balance* condition: the difference between left subtree height and right subtree height is at most 1.

Properties of AVL Trees

A binary tree is said to be an AVL tree, if:

- It is a binary search tree, and
- For any node *X*, the height of left subtree of *X* and height of right subtree of *X* differ by at most 1.

Screen clipping taken: 29-04-2025 10:50



As an example, among the above binary search trees, the left one is not an AVL tree, whereas the right binary search tree is an AVL tree.

Minimum/Maximum Number of Nodes in AVL Tree

For simplicity let us assume that the height of an AVL tree is h and $N(K)$ indicates the number of nodes in AVL tree with height h. To get the minimum number of nodes with height h, we should fill the tree with the minimum number of nodes possible. That means if we fill the left subtree with height $h - 1$ then we should fill the right subtree with height $h - 2$. As a result, the minimum number of nodes with height h is:

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

In the above equation:

- $N(h-1)$ indicates the minimum number of nodes with height $h - 1$.
- $N(h-2)$ indicates the minimum number of nodes with height $h - 2$.
- In the above expression, "1" indicates the current node.

We can give $N(h-1)$ either for left subtree or right subtree. Solving the above recurrence gives:

$$N(h) = O(1.618^h) \Rightarrow h = 1.44 logn \approx O(logn)$$

Where n is the number of nodes in AVL tree. Also, the above derivation says that the maximum height in AVL trees is $O(logn)$. Similarly, to get maximum number of nodes, we need to fill both left and right subtrees with height $h - 1$. As a result, we get:

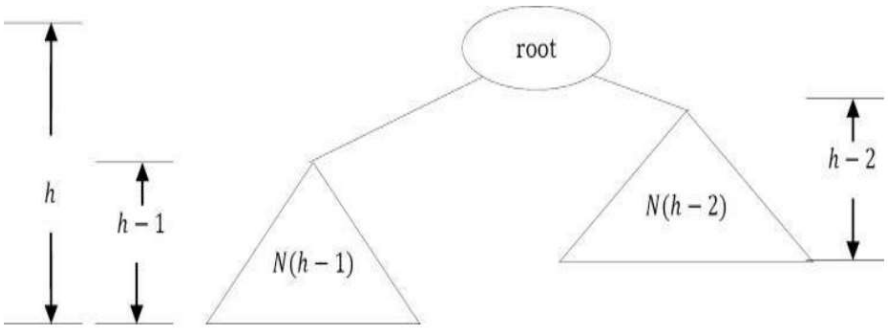
$$N(h) = N(h-1) + N(h-1) + 1 = 2N(h-1) + 1$$

The above expression defines the case of full binary tree. Solving the recurrence we get:

$$N(h) = O(2^h) \Rightarrow h = logn \approx O(logn)$$

$\therefore$ In both the cases, AVL tree property is ensuring that the height of an AVL tree with n nodes is $O(logn)$.

∴ In both the cases, AVL tree property is ensuring that the height of an AVL tree with n nodes is O($logn$).



Screen clipping taken: 29-04-2025 11:05

Rotations

When the tree structure changes (e.g., with insertion or deletion), we need to modify the tree to restore the AVL tree property. This can be done using single rotations or double rotations. Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of a subtree by 1.

So, if the AVL tree property is violated at a node X, it means that the heights of left(X) and right(X) differ by exactly 2. This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(X) and right(X) differ by exactly 2. Rotations is the technique used for restoring the AVL tree property. This means, we need to apply the rotations for the node X.

Screen clipping taken: 29-04-2025 11:08

Types of Violations

Let us assume the node that must be rebalanced is X. Since any node has at most two children, and a height imbalance requires that X's two subtree heights differ by two, we can observe that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of X.
2. An insertion into the right subtree of the left child of X.

3. An insertion into the left subtree of the right child of X.
4. An insertion into the right subtree of the right child of X.

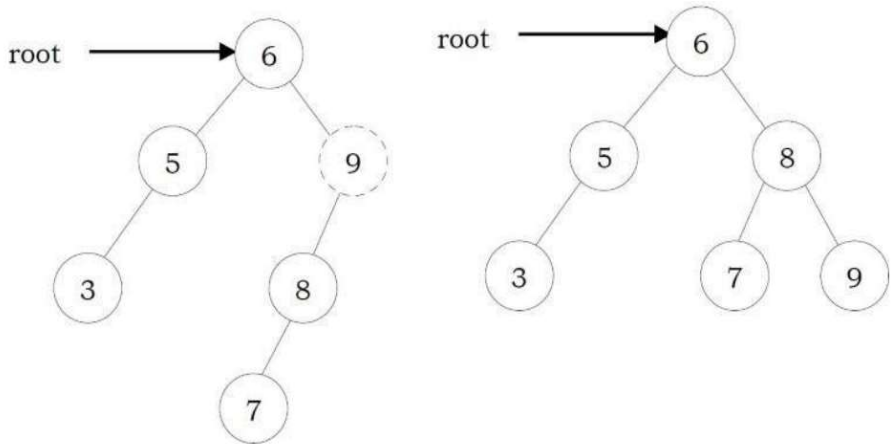
Cases 1 and 4 are symmetric and easily solved with single rotations. Similarly, cases 2 and 3 are also symmetric and can be solved with double rotations (*n*eeds two single rotations).

Screen clipping taken: 29-04-2025 11:08

Single Rotations

Left Left Rotation (LL Rotation) [Case-1]: In the case below, node X is not satisfying the AVL tree property. As discussed earlier, the rotation does not have to be done at the root of a tree. In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.

Screen clipping taken: 29-04-2025 11:10



For example, in the figure above, after the insertion of 7 in the original AVL tree on the left, node 9 becomes unbalanced. So, we do a single left-left rotation at 9. As a result we get the tree on the right.

Screen clipping taken: 29-04-2025 11:10

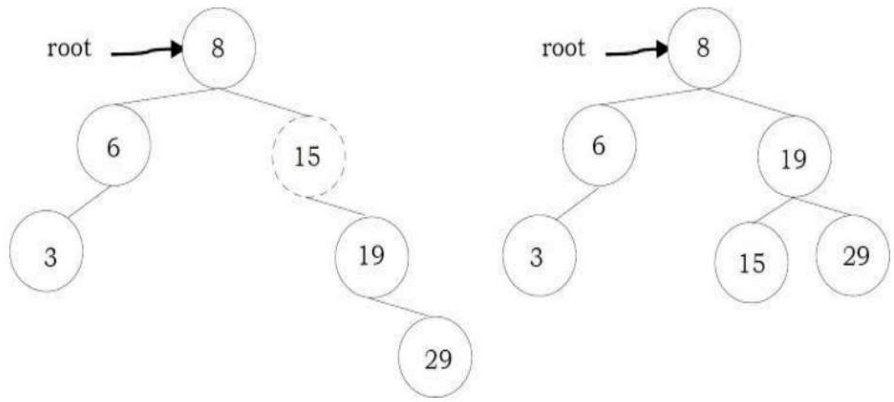
Right Right Rotation (RR Rotation) [Case-4]: In this case, node *X* is not satisfying the AVL tree property.

Screen clipping taken: 29-04-2025 11:11

For example, in the figure, after the insertion of 29 in the original AVL tree on the left, node 15 becomes unbalanced. So, we do a single right-right rotation at 15. As a result we get the tree on the right.

Screen clipping taken: 29-04-2025 11:11

Time Complexity: O(1). Space Complexity: O(1).



Screen clipping taken: 29-04-2025 11:11

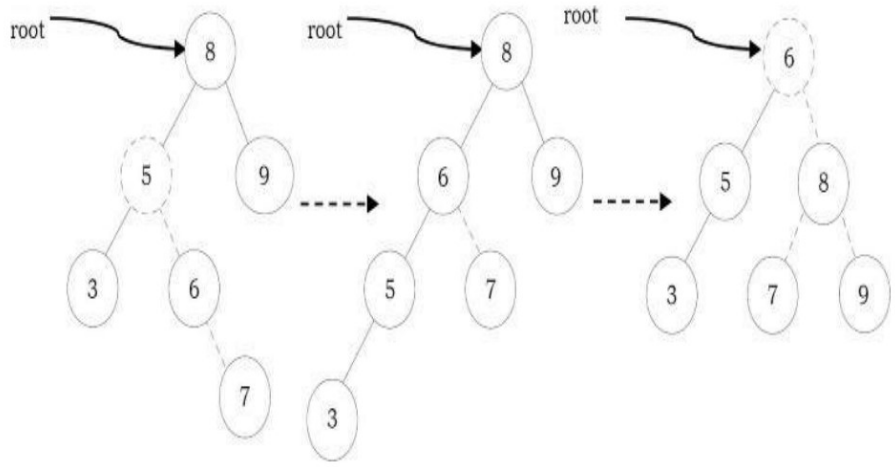
Double Rotations

Left Right Rotation (LR Rotation) [Case-2]: For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations.

Screen clipping taken: 29-04-2025 11:11

As an example, let us consider the following tree: The insertion of 7 is creating the case-2 scenario and the right side tree is the one after the double rotation.

Screen clipping taken: 29-04-2025 11:11



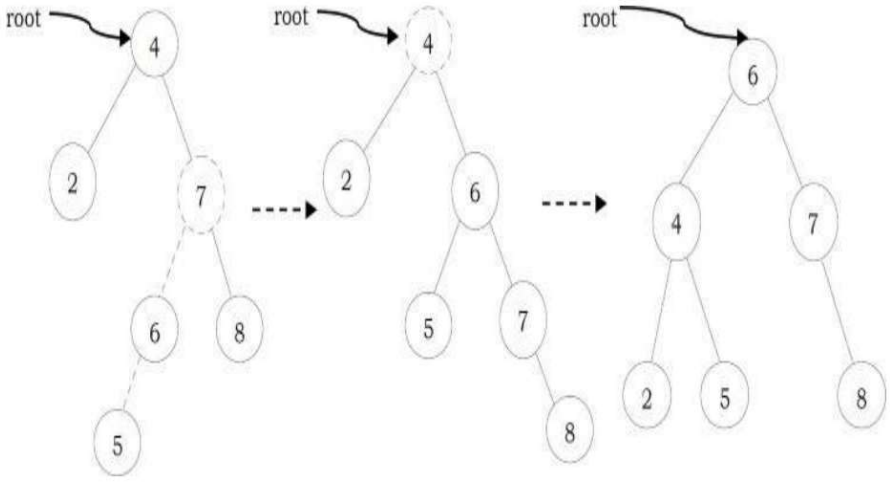
Screen clipping taken: 29-04-2025 11:12

Right Left Rotation (RL Rotation) [Case-3]: Similar to case-2, we need to perform two rotations to fix this scenario.

Screen clipping taken: 29-04-2025 11:12

As an example, let us consider the following tree: The insertion of 6 is creating the case-3 scenario and the right side tree is the one after the double rotation.

Screen clipping taken: 29-04-2025 11:12



Screen clipping taken: 29-04-2025 11:12

Insertion into an AVL tree

Insertion into an AVL tree is similar to a BST insertion. After inserting the element, we just need to check whether there is any height imbalance. If there is an imbalance, call the appropriate rotation functions.

Screen clipping taken: 29-04-2025 11:13

6.14.3 B-Trees

B-Tree is like other self-balancing trees such as AVL and Red-black tree such that it maintains its balance of nodes while opertions are performed against it. B-Tree has the following properties:

- Minimum degree "£" where, except root node, all other nodes must have no less than $t - 1$ keys
- Each node with n keys has $n + 1$ children
- Keys in each node are lined up where $k_1 < k_2 < .. k_n$
- Each node cannot have more than 2t-l keys, thus 2t children
- Root node at least must contain one key. There is no root node if the tree is empty.
- Tree grows in depth only when root node is split.

Screen clipping taken: 29-04-2025 11:17

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceeding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

A b-tree has a minumum number of allowable children for each node known as the *minimization factor*. If t is this *minimization factor*, every node must have at least $t - 1$ keys. Under certain circumstances, the root node is allowed to violate this property by having fewer than $t - 1$ keys. Every node may have at most $2t - 1$ keys or, equivalently, $2t$ children.

Screen clipping taken: 29-04-2025 11:22

To *search* the tree, it is similar to binary tree except that the key is compared multiple times in a given node because the node contains more than 1 key. If the key is found in the node, the search terminates. Otherwise, it moves down where at child pointed by ci where key $k < k_i$.

Key *insertions* of a B-tree happens from the bottom fasion. This means that it walk down the tree from root to the target child node first. If the child is not full, the key is simply inserted. If it is full, the child node is split in the middle, the median key moves up to the parent, then the new key

is inserted. When inserting and walking down the tree, if the root node is found to be full, it's split first and we have a new root node. Then the normal insertion operation is performed.

Key *deletion* is more complicated as it needs to maintain the number of keys in each node to meet the constraint. If a key is found in leaf node and deleting it still keeps the number of keys in the nodes not too low, it's simply done right away. If it's done to the inner node, the predecessor of the key in the corresonding child node is moved to replace the key in the inner node. If moving the predecessor will cause the child node to violate the node count constraint, the sibling child nodes are combined and the key in the inner node is deleted.

Screen clipping taken: 29-04-2025 11:22

Screen clipping taken: 29-04-2025 10:50

AVL Trees:

AVL TREES

Adelsion Velski and Lendis in 1962 introduced binary tree structure that is balanced with respect to height of sub trees.
The tree can be made balanced and because of this retrieval of any node can be done in O(log n) times, where n is total number of nodes. From the name of these scientists the tree is called AVL tree.

Definition:

An empty tree is height balanced if T is a non empty binary tree with TL and TR as its left and right sub trees. The T is height balanced if and only if
i. TL and TR are height balanced.
ii. hL-hR <= 1 where hL and hR are heights of TL and TR.

The idea of balancing a tree is obtained by calculating the balance factor of a tree.

Definition of Balance Factor:

The balance factor BF(T) of a node in binary tree is defined to be hL-hR where hL and hR are heights of left and right sub trees of T.
For any node in AVL tree the balance factor i.e. BF(T) is -1, 0 or +1.