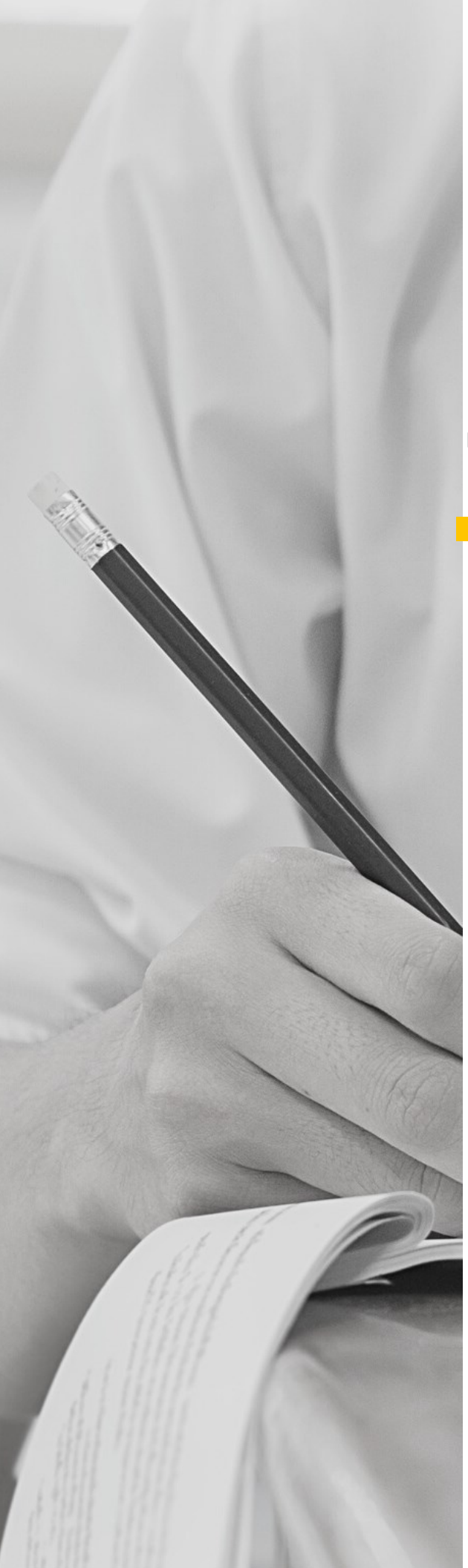


Prof. Trupesh
Patel's Notes

Tree

Data Structures & Algorithms

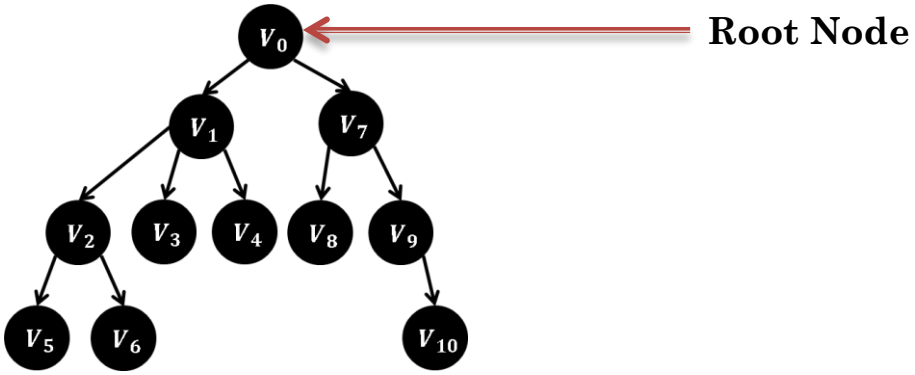
- Tree – definitions & concept
- Binary tree traversal
- Threaded binary tree
- Binary search trees
- AVL TREES
- B Tree
- B+ Tree

♣ TREE – Definitions & Concepts

- A **Node** is an entity that contains a value and pointers to its child nodes.
- A **leaf node** is a node that does not have any child node.
- **Edge** is a link between two nodes.
- A **root node** of a tree is a top most node of a tree.
- **Depth of a Node** is the number of edges from root to the current node.
- **Height of a Tree** is the height of the root node or the depth of the deepest node.
- **Degree of a Node** is the total number of branches of that node.
- **Forest** is a collection of disjoint trees is called a forest.
- Two or more nodes with the same parent are called **siblings**
- A **path** is a sequence of nodes in which each node is adjacent to the next one.
- An **ancestor** is any node in the path from the root of a given node. A **descendent** is any node in all of the paths from a given node to a leaf.
- A **subtree** is any connected structure below the root.
- A **binary tree** is a tree in which no node can have more than two children.

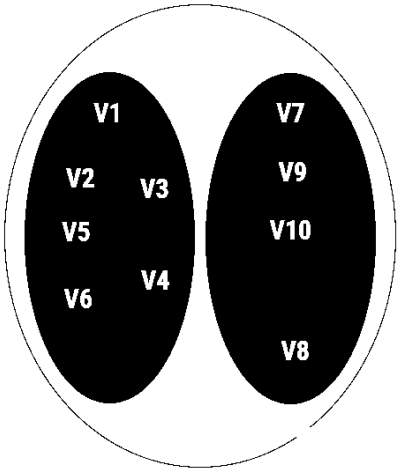
♣ Directed Tree:

- A **directed tre**e is an acyclic digraph which has one node called its root with in degree 0, while all other nodes have in degree 1.
- Every directed tree must have at least one node.
- An isolated node is also a directed tree.



♣ Types of representation:

❖ Venn Diagram

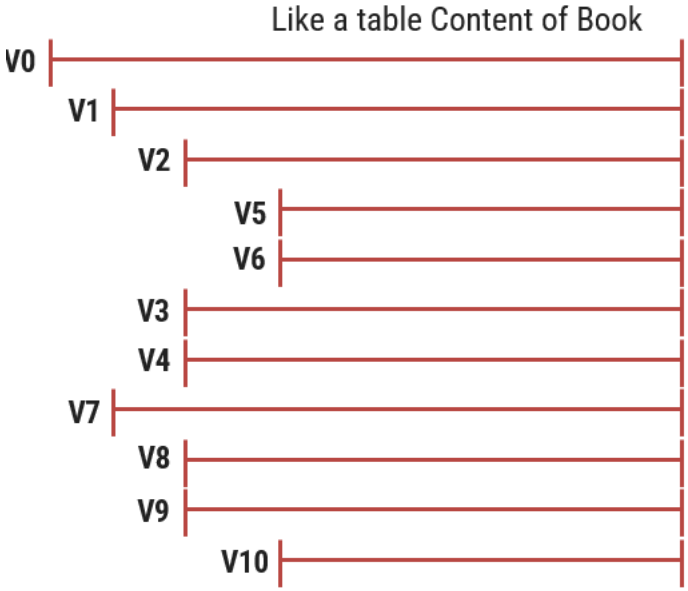


❖ Nesting of Parenthesis

$(V_0\ (V_1\ (V_2\ (V_5)\ (V_6)\ )\ (V_3)\ (V_4)\ )\ (V_7\ (V_8)\ (V_9\ (V_{10})\ )\ )$

Nesting of Parenthesis

❖ Like table content of Book

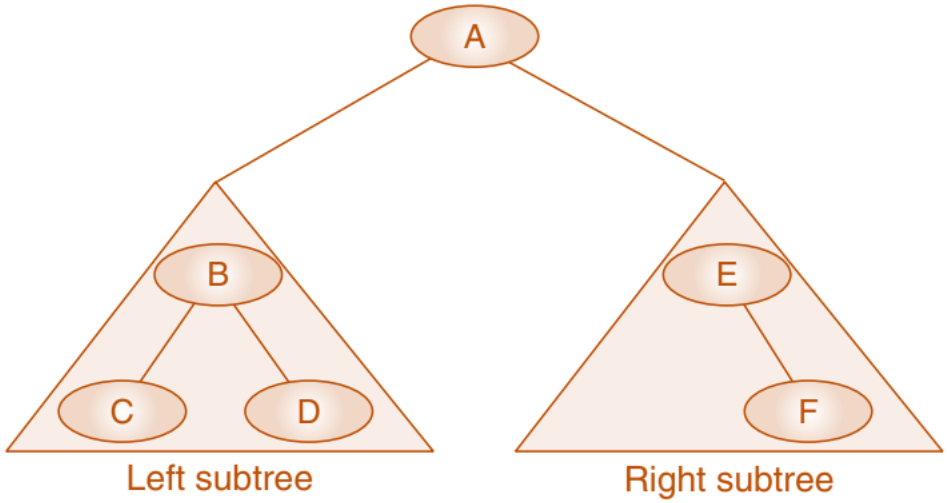


❖ Level Format

1 V_0
2 V_1
3 V_2
4 V_5
4 V_6
3 V_3
3 V_4
2 V_7
3 V_8
3 V_9
4 V_{10}

♣ Binary Tree Traversal

Binary Tree: A binary tree is a tree in which no node can have more than two subtrees; the maximum outdegree for a node is two.

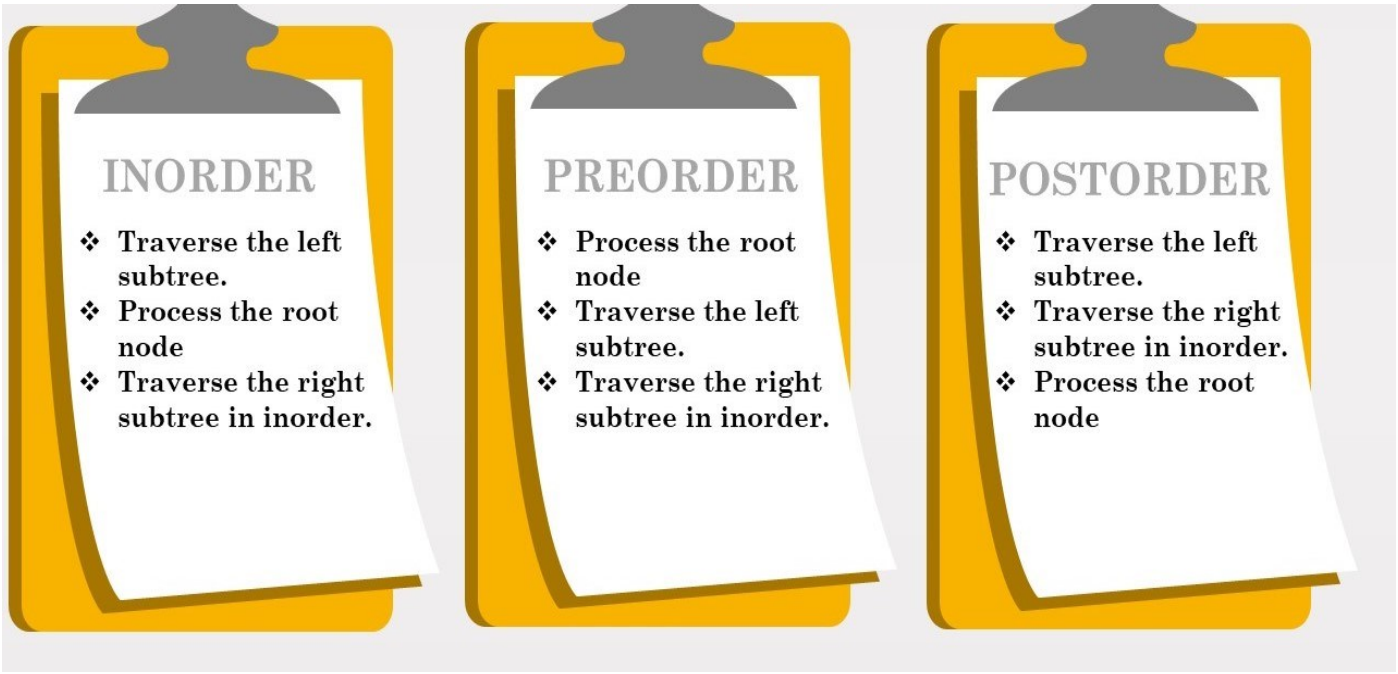


Binary tree

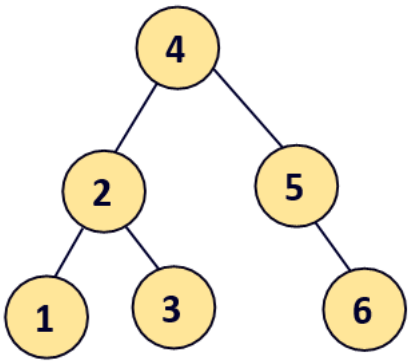
Traversing a tree implies to visit every node in the tree.
Order of visiting various node decides variant of tree traversal.

Binary TREE Traversal – Definitions & Concepts

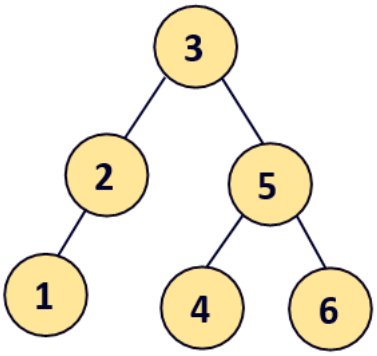
It is a process to visit each node at least once for a given tree.



Inorder:

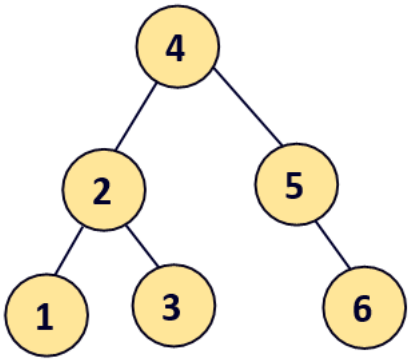


Inorder: 1 2 3 4 5 6

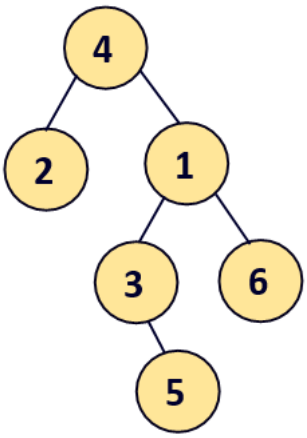


Inorder: 1 2 3 4 5 6

Preorder:

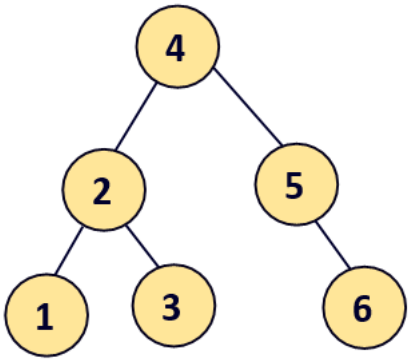


Preorder: 4 2 1 3 5 6

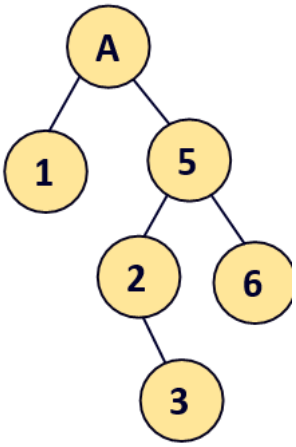


Preorder: 4 2 1 3 5 6

Postorder:



Postorder: 1 3 2 6 5 4



Postorder: 1 3 2 6 5 A

Example:

You are given two traversal Inorder and Preorder of Binary seach tree as: Inorder: g d h b e a f j c Preorder : a b d g h e c f j

Solution:

Inorder: g d h b e a f j c

Preorder: a b d g h e c f j

Step 1

Scan the preorder traversal from left to right one element at a time. Use thatcurrently scanned element in in-order traversal to get its left subtree and

Pre-order : a b d g h e c

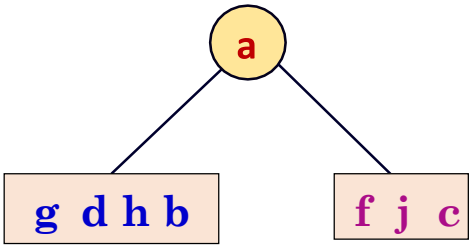
Scanning Preorder Traversalfrom left to right

a is root node so use traversal to get its right and

In-order : g d h b e a f j

a is the root of the

g d h b eare in the left f j c are in the right



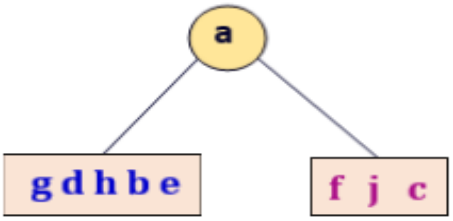
Step 2

Next element under scan is b. Use currently scanned **element b** in left sub-tree of a to get further its left sub-tree and right sub-tree.

Preorder : a b d g h e c f j

Current element under scan i.e element b

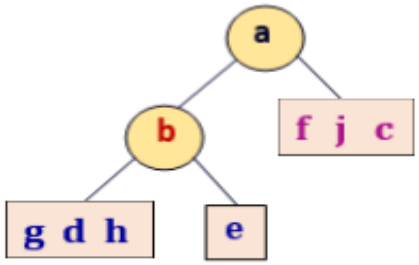
b is root node in the left subtree of a.

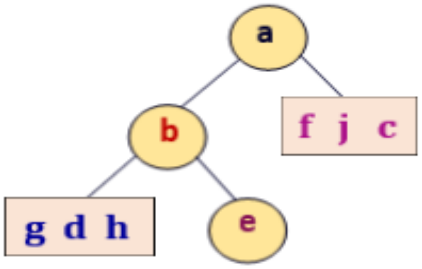


Left Subtree of a : g d h b e

b is the root

g d h are in the left subtree of b e j is in the right subtree of b

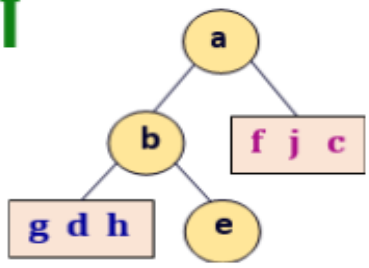




Step 3

Next element under scan is d. Use currently scanned **element d** in left sub-tree of b to get further its left sub-tree and right sub-tree.

Preorder : a b d g h e c f j

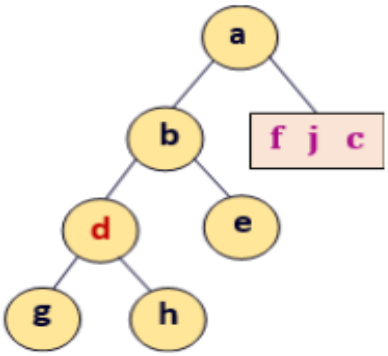


d is root node in the left subtree of b.

Left Subtree of b : g d h

d is the root

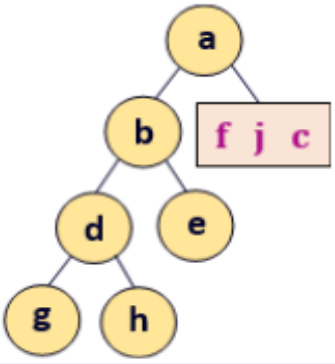
g is in the left subtree of d **h is in the right subtree of d**



Step 4

Next scanned element is g, and **g has no left or right child.** So stop and go for scanning next element.

Preorder : a b d g h e c f j



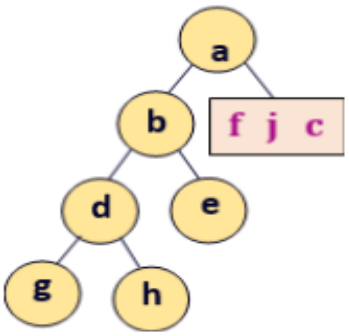
g is a single value node. Node g has no left or right child. So move to next step.

Step 5,6

Next scanned elements are h and e. Element h and e has no left or right child. So stop and go for scanning next element.

In Step 5 Preorder : a b d g h e c f j

In Step 6 Preorder : a b d g h e c f j



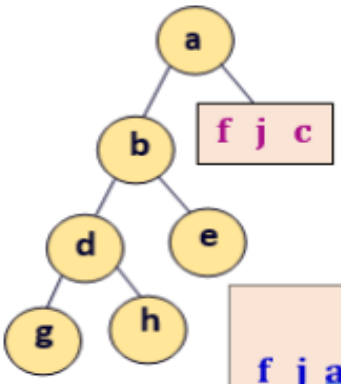
Step 7

Next element under scan is c. Use currently scanned **element c** in right sub-tree of a to get further its left sub-tree and right sub-tree.

Preorder : a b d g h e c f j

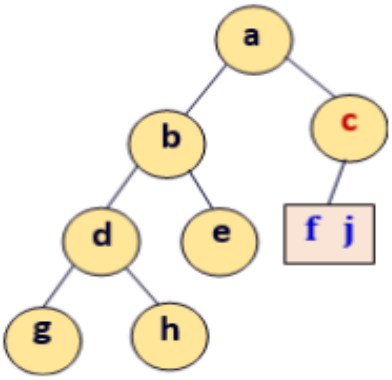
c is root node in the right subtree of a.

Right Subtree of a : f j c



c is the root

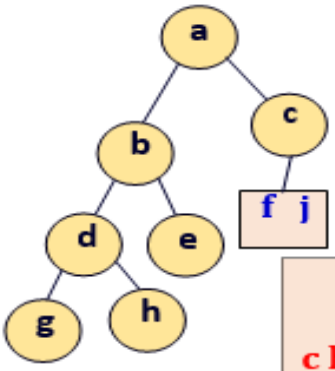
f j are in the left subtree of c **c has no right subtree**



Step 8

Next element under scan is f. Use currently scanned **element f** in left sub-tree of c to get further its left sub-tree and right sub-tree(if any).

Preorder : a b d g h e c f j

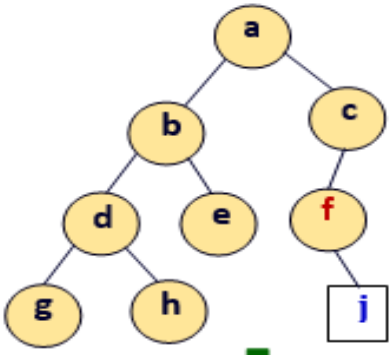


f is root node in the left subtree of c.

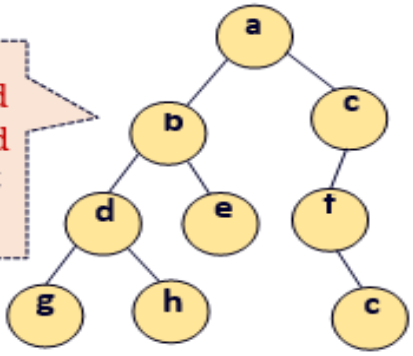
Right Subtree of c : f j

f is the root

c has no right child **j is in the right subtree of f**



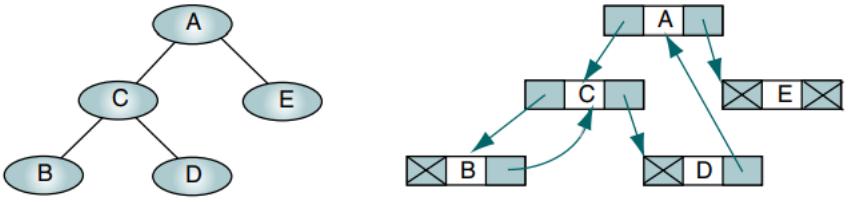
This is original binary tree, which we have reconstructed from given inorder and preorder traversal. **So height of this binary tree is 3.**



♣ Threaded Binary Tree

- In a threaded tree, null pointers are replaced with pointers to their successor nodes.
- in the inorder traversal of a binary tree, we must traverse the left subtree, the node, and the right subtree. Because an inorder traversal is a depth-first traversal, we follow left pointers to the far-left leaf.

- When we find the far-left leaf, we must begin backtracking to process the right subtrees we passed on our way down. This is especially inefficient when the parent node has no right subtree.



(a) Binary tree (b) Threaded binary tree

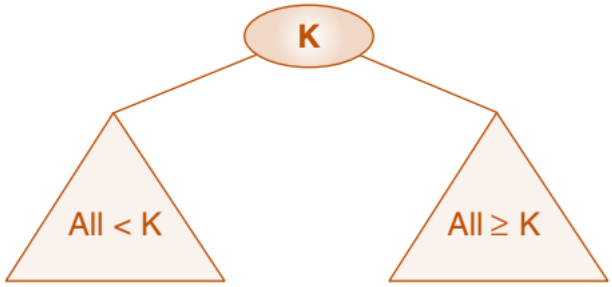
- The inorder traversal of this tree is **BCDAE**.
- When we traverse the tree in inorder sequence, we follow the left pointers to get the first node, B.

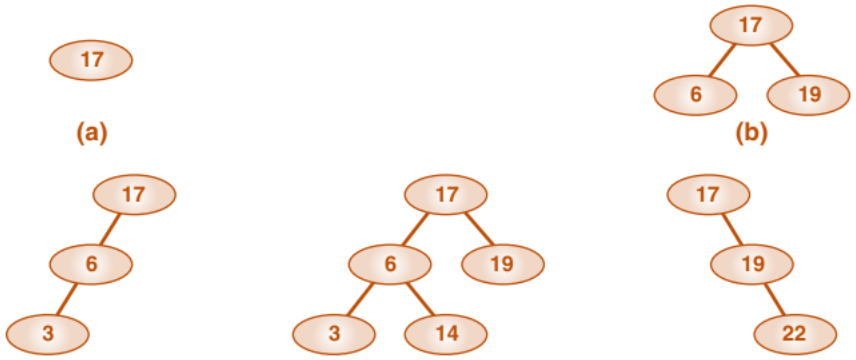
  However, after locating the far-left node, we must go back to C, which is why we need recursion or a stack. Note that when we are processing B, we do not need recursion or a stack because B's right subtree is empty. Similarly, when we finish processing node D using recursion or stacks, we must return to node C before we go to A. But again, we do not need to pass through C.
- The next node to be processed is the root, A.
- From this small example, it should be clear that the nodes whose right subtrees are empty create more work when we use recursion or stacks. This leads to the threaded concept: when the right subtree pointer is empty, we can use the pointer to point to the successor of the node.
- In other words, we can use the right null pointer as a thread. To build a threaded tree, first build a standard binary search tree. Then traverse the tree, changing the null right pointers to point to their successors.

♣ Binary Search Trees

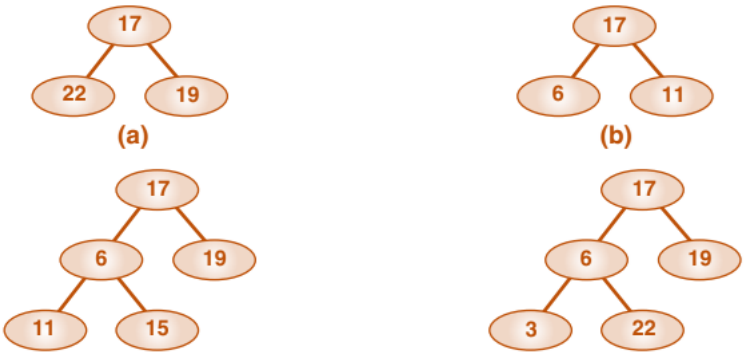
A binary tree can be a Binary search trees (BST) with the below mentioned properties.

- All the elements in the left subtree are always less than the root element.
- All the elements in the right subtree are always greater than the root element.
- Each subtree is itself a binary search tree.





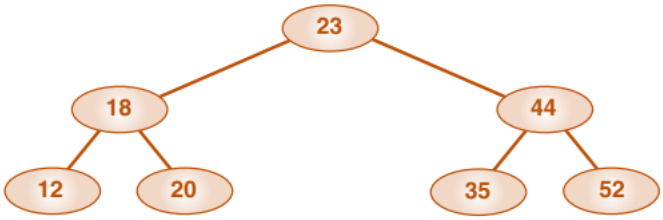
Valid binary search tree



Invalid binary search tree

♣ Operations on Binary Search Trees

Traversals:



Binary Search Tree

Pre order : 23 18 12 20 44 35 52

Post order : 12 20 18 35 52 44 23

Inorder : 12 18 20 23 35 44 52

Inorder traversal of BST always produces a sequenced list.

Searches:

Find the smallest node: we can see that smallest node is available at the leaf node of the BST, so to search smallest node, we have to follow left sub branches until we get to a leaf.

Find the Smallest node in BST

Algorithm: findSmallestBST (root)

This algorithm finds the smallest node in a BST. Pre root is a pointer to a nonempty BST or subtree Return address of smallest node.

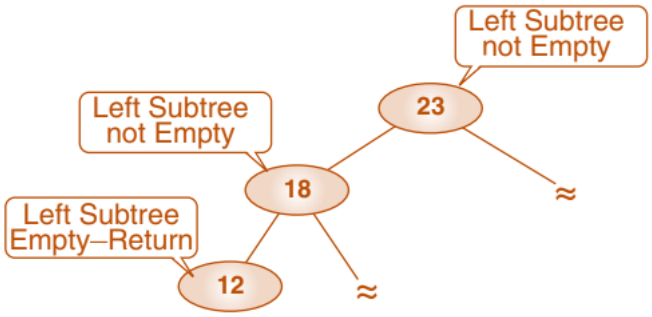
1. if (left subtree empty)

1 return (root)

2. end if

3. return findSmallestBST (left subtree)

end findSmallestBST



Find the Largest node in BST

We have to follow right subtree from root node till leaf.

Algorithm: findLargestBST (root)

This algorithm finds the largest node in a BST.

Pre root is a pointer to a nonempty BST or subtree Return address of largest node returned

1 if (right subtree empty)

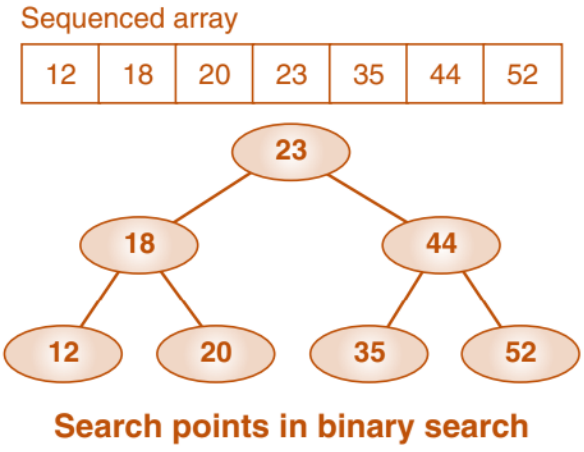
1 return (root)

2 end if

3 return findLargestBST (right subtree)

end findLargestBST

Find the Requested node in BST



Search points in binary search

Find a given node in a binary search tree.

- Assume we are looking for node 20. We begin by comparing the search argument, 20, with the value in the tree root. Because 20 is less than the root value, 23, and because we know that all values less than the root lie in its left subtree, we go left.
- We now compare the search argument with the value in the subtree, 18. This time the search argument is greater than the root value, 18. Because we know that values greater than the tree root must lie in its right subtree, we go right and find our desired value.

Algorithm searchBST (root, targetKey)

Search a binary search tree for a given value. Pre root is the root to a binary tree or subtree targetKey is the key value requested

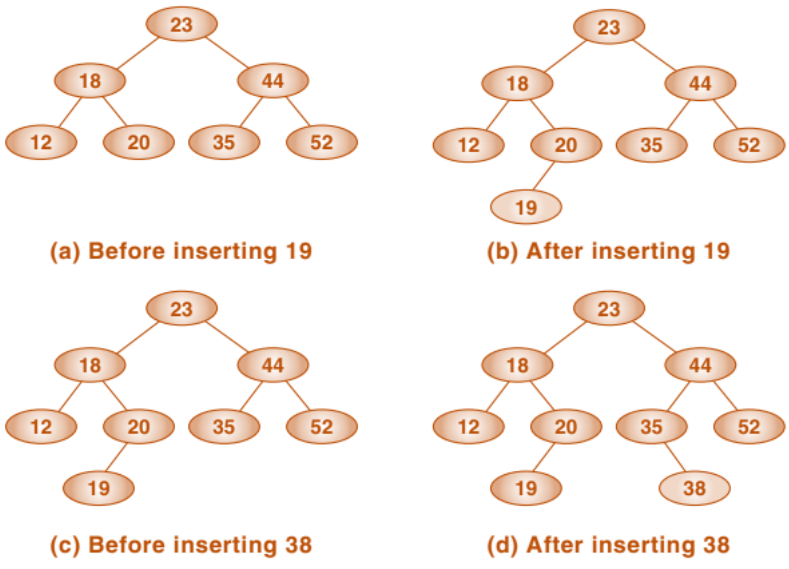
Return the node address if the value is found

null if the node is not in the tree

```
1 if (empty tree)
        Not found
  1 return null
2 end if
3 if (targetKey < root)
  1 return searchBST (left subtree, targetKey)
4 else if (targetKey > root)
  1 return searchBST (right subtree, targetKey)
5 else
      Found target key
    1 return root
6 end if
end searchBST
```

Insertion:

The insert node function adds data to a BST. To insert data all we need to do is follow the branches to an empty subtree and then insert the new node.



BST insertion

Add node to BST:

Algorithm addBST (root, newNode)

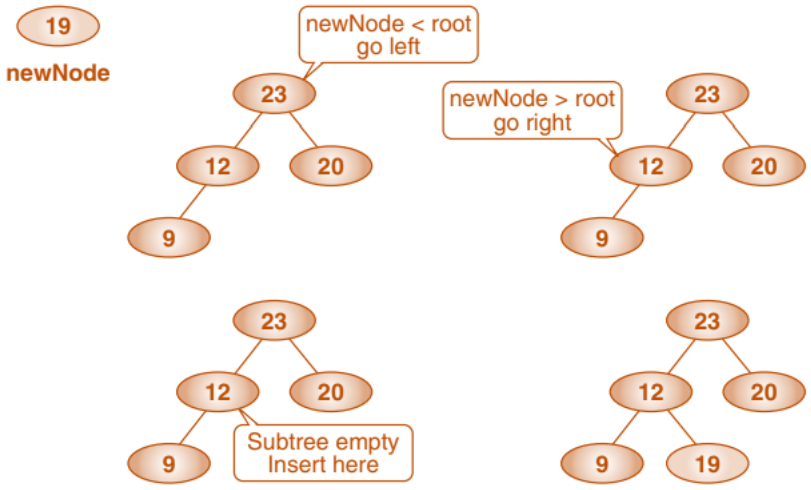
Insert node containing new data into BST using recursion.

Pre root is address of current node in a BST
newNode is address of node containing data

Post newNode inserted into the tree

Return address of potential new tree root

1 if (empty tree)
1 set root to newNode
2 return newNode
2 end if
Locate null subtree for insertion
3 if (newNode < root)
1 return addBST (left subtree, newNode)
4 else
1 return addBST (right subtree, newNode)
5 end if
end addBST



Trace of recursive BST Insert

Deletion:

To delete a node from a binary search tree, we must first locate it. There are four possible cases when we delete a node:

1. The node to be deleted has no children. In this case, all we need to do is delete the node.
2. The node to be deleted has only a right subtree. We delete the node and attach the right subtree to the deleted node's parent.
3. The node to be deleted has only a left subtree. We delete the node and attach the left subtree to the deleted node's parent.

4. The node to be deleted has two subtrees. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced

Algorithm deleteBST (root, dltKey)

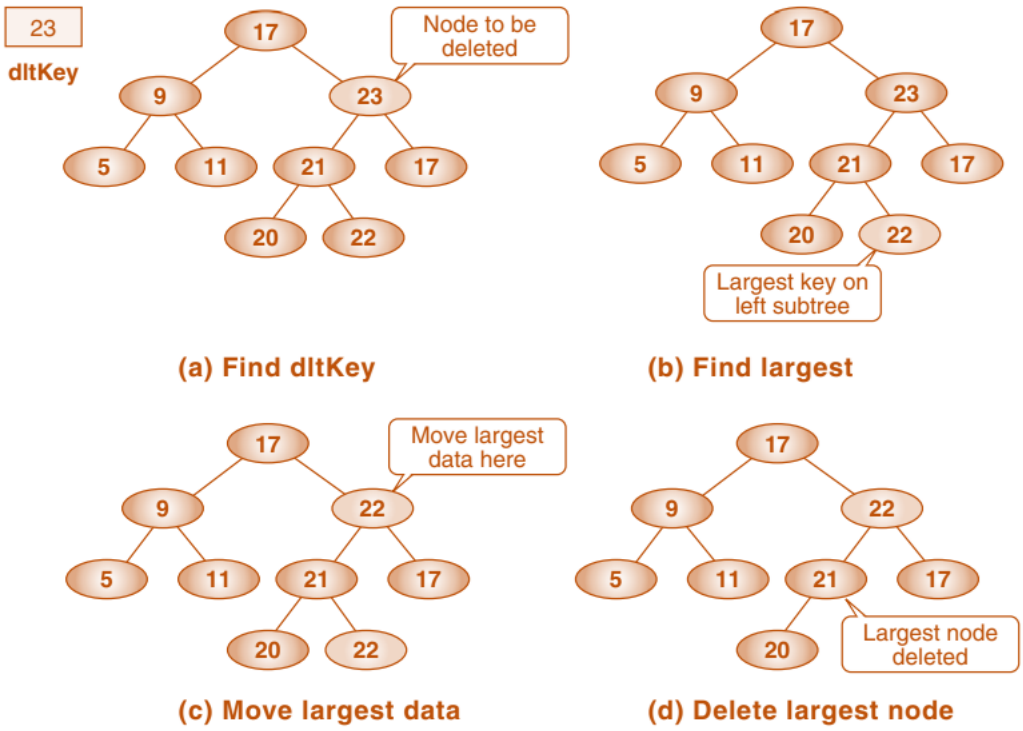
This algorithm deletes a node from a BST.

Pre root is reference to node to be deleted
dltKey is key of node to be deleted

Post node deleted
if dltKey not found, root unchanged

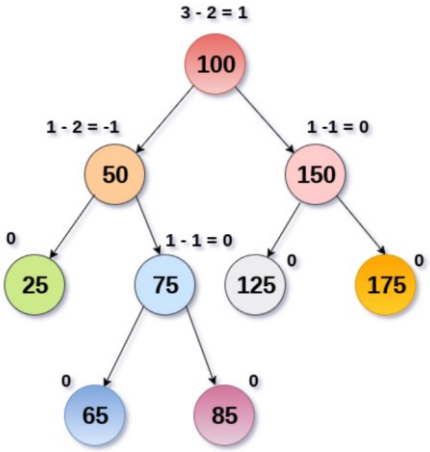
Return true if node deleted, false if not found

```
1 if (empty tree)
        1 return false
2 end if
3 if (dltKey < root)
        1 return deleteBST (left subtree, dltKey)
4 else if (dltKey > root)
        1 return deleteBST (right subtree, dltKey)
5 else
        Delete node found--test for leaf node
        1 If (no left subtree)
                1 make right subtree the root
                2 return true
        2 else if (no right subtree)
                1 make left subtree the root
                2 return true
        3 else
                Node to be deleted not a leaf. Find largest node on left subtree.
                1 save root in deleteNode
                2 set largest to largestBST (left subtree)
                3 move data in largest to deleteNode
                4 return deleteBST (left subtree of deleteNode, key of largest
        4 end if
6 end if
        end deleteBST
```



♣ AVL TREES

- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- Balance Factor (k) = height (left(k)) - height (right(k))
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

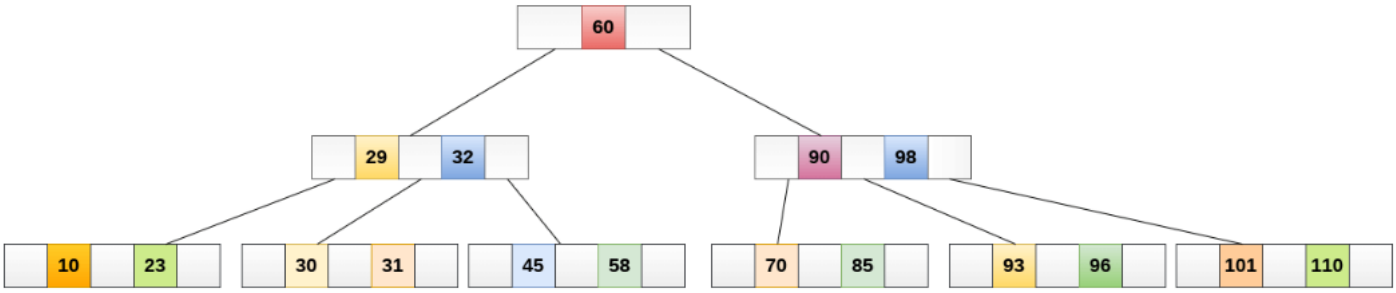


♣ B TREE:

- ❖ B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. 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.
- ❖ 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.

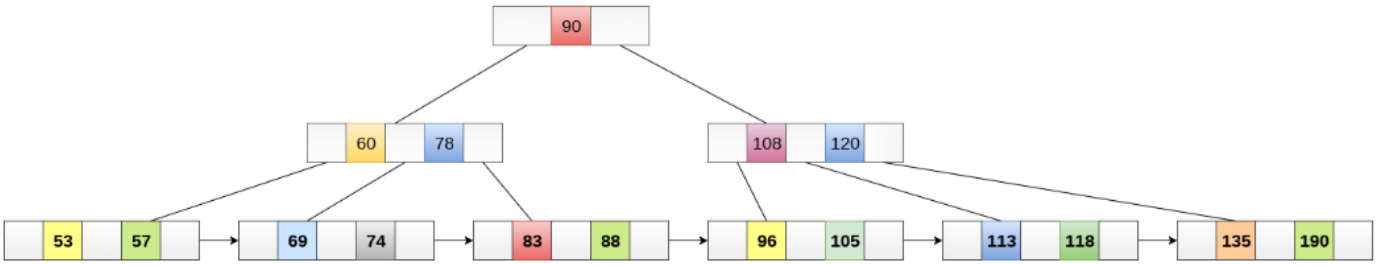
It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

- ❖ A B tree of order 4 is shown in the following image.



♣ B+ TREE:

- ❖ 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.
- ❖ B+ Tree are used to store the large amount of data which cannot be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.
- ❖ The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



♣ 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.