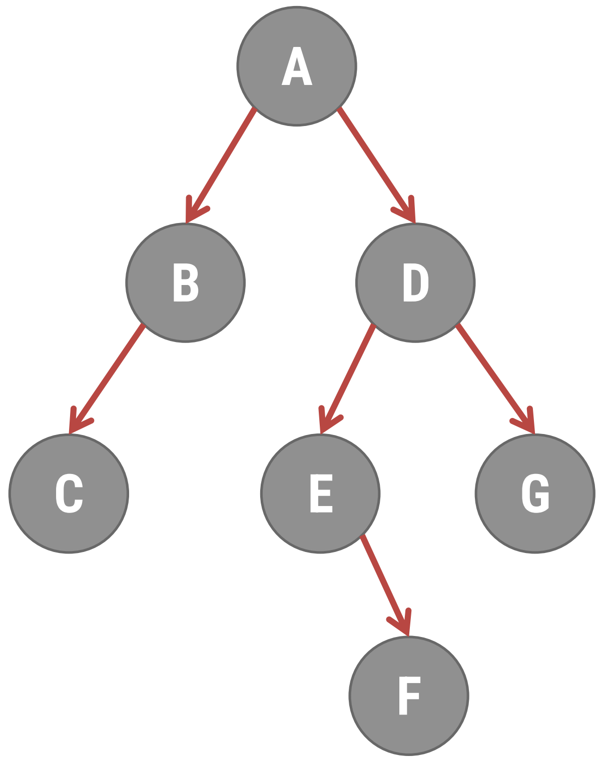
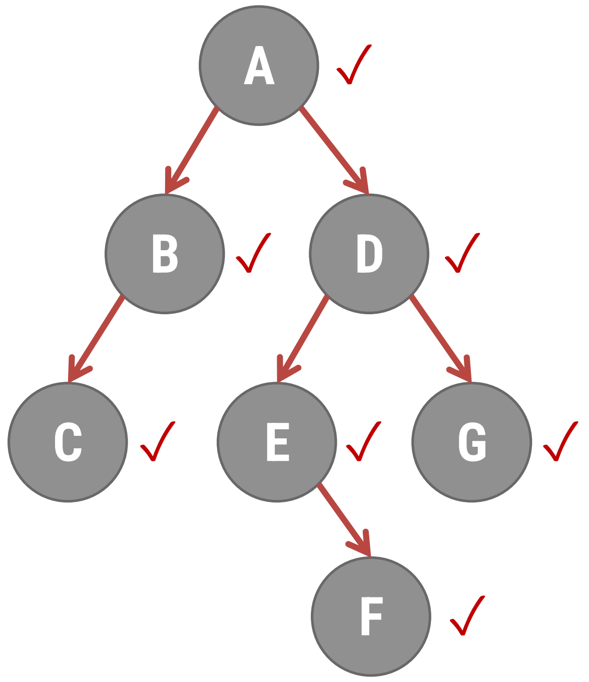
Tree Traversal

- The most common operations performed on tree structure is that of traversal.
- This is a **procedure by which each node in the tree is processed exactly once** in a systematic manner.
- There are three ways of traversing a binary tree.
  1. Preorder Traversal
  2. Inorder Traversal
  3. Postorder Traversal



Preorder Traversal

- Preorder traversal of a binary tree is defined as follow
  1. **Process** the **root node**
  2. **Traverse** the **left subtree** in preorder
  3. **Traverse** the **right subtree** in preorder
- If particular **subtree is empty** (i.e., node has no left or right descendant) the traversal is performed by **doing nothing.**
- In other words, a **null subtree** is **considered to be fully traversed** when it is encountered.

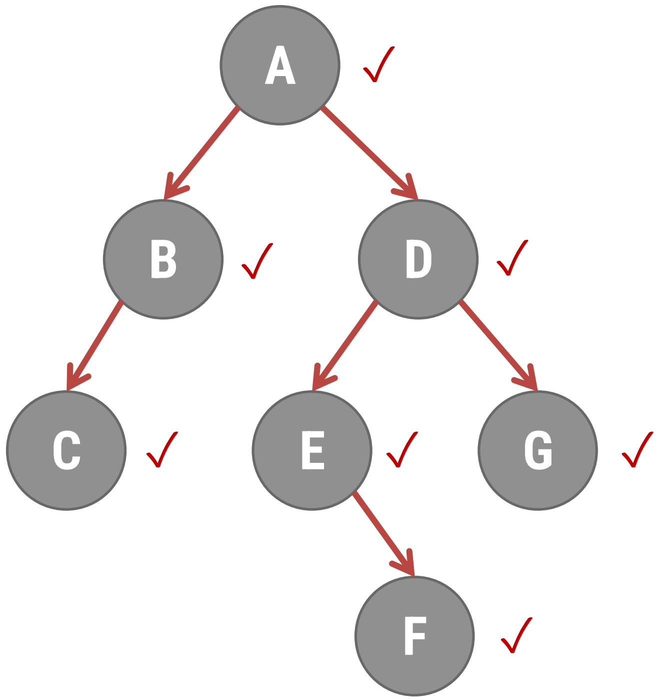


Preorder traversal of a given tree as

A B C D E F G

Inorder Traversal

- Inorder traversal of a binary tree is defined as follow
  1. **Traverse** the **left subtree** in Inorder
  2. **Process** the **root node**
  3. **Traverse** the **right subtree** in Inorder

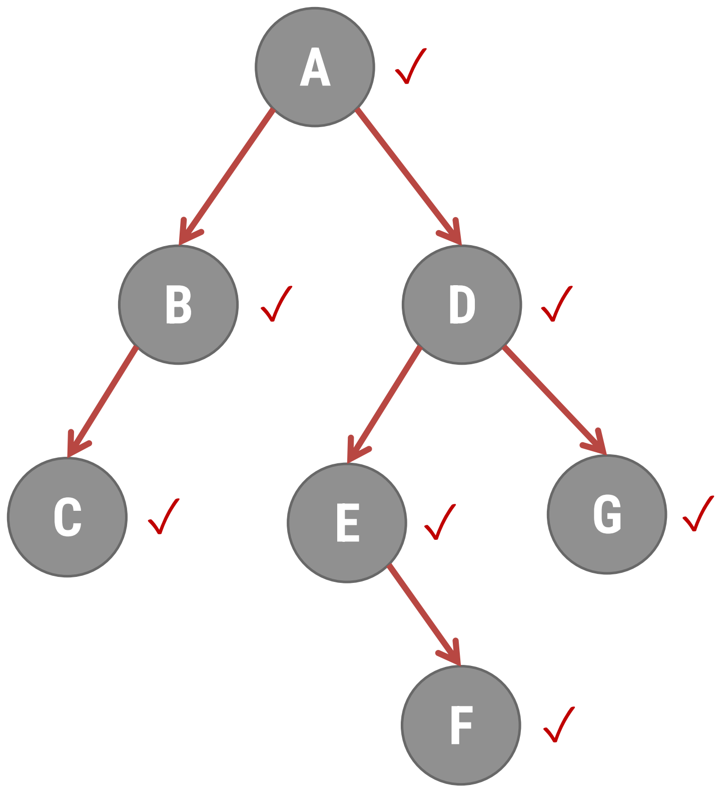


Inorder traversal of a given tree as

C B A E F D G

Postorder Traversal

- Postorder traversal of a binary tree is defined as follow
  1. **Traverse** the **left subtree** in Postorder
  2. **Traverse** the **right subtree** in Postorder
  3. **Process** the **root node**



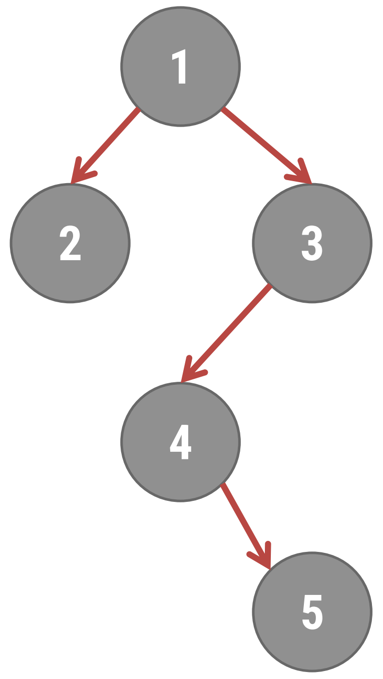
Postorder traversal of a given tree as

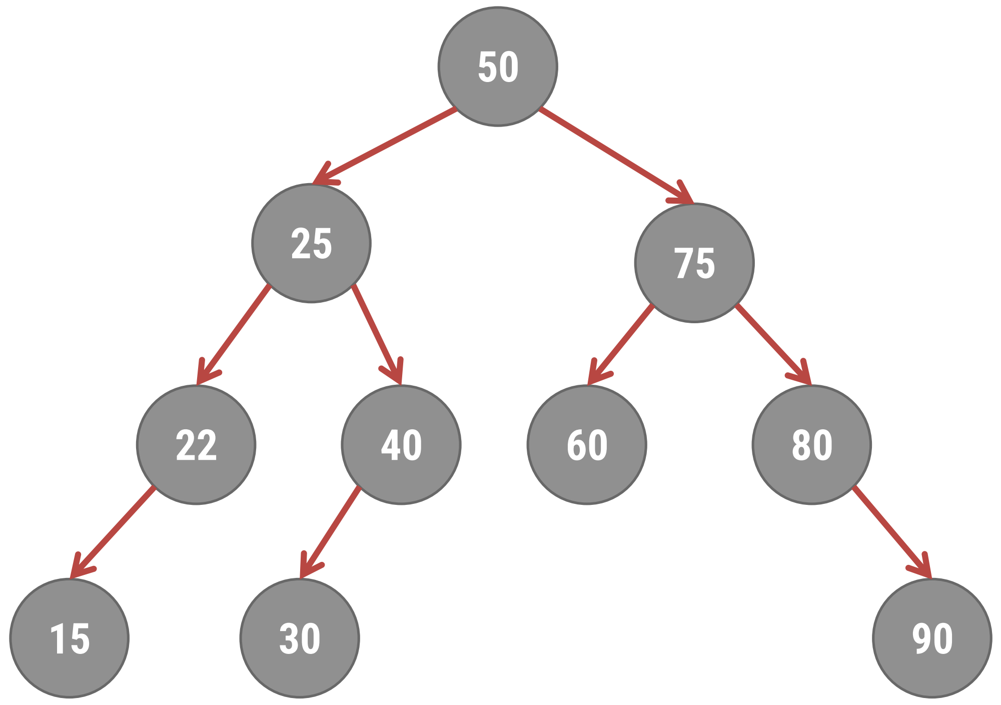
C B F E G D A

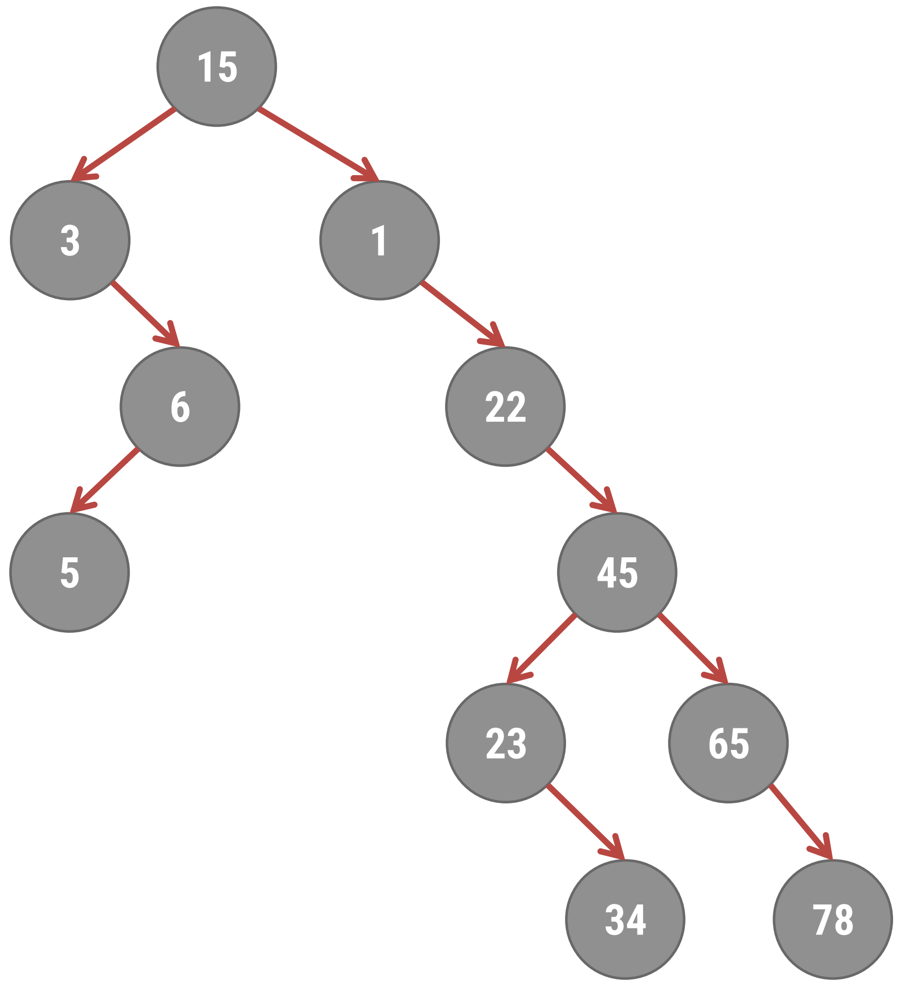
Converse Traversal

- If we ***interchange left and right words in the preceding definitions***, we obtain three new traversal orders which are called
  - **Converse Preorder** Traversal: A D G E F B C
  - **Converse Inorder** Traversal: G D F E A B C
  - **Converse Postorder** Traversal: G F E D C B A

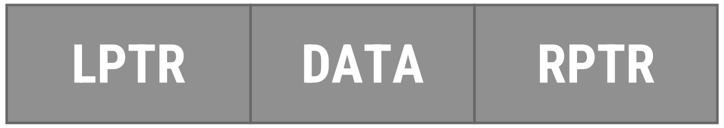
Write Pre/In/Post Order Traversal



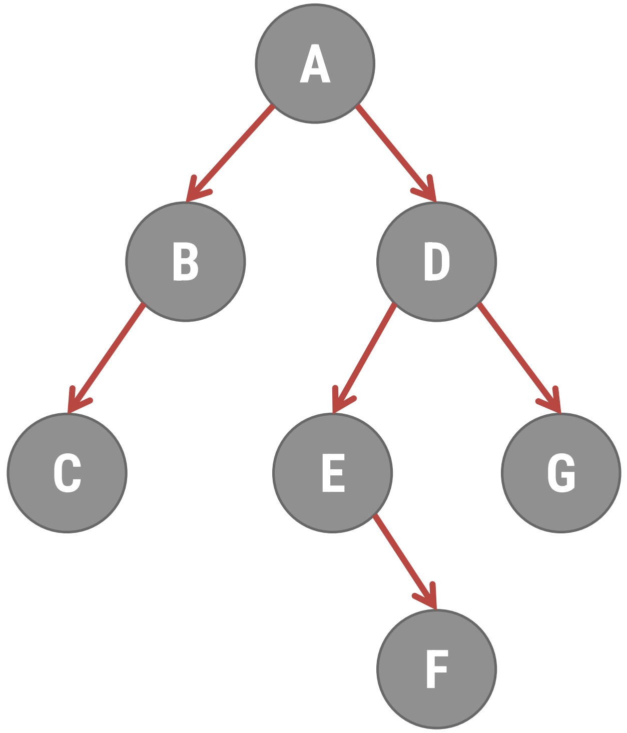


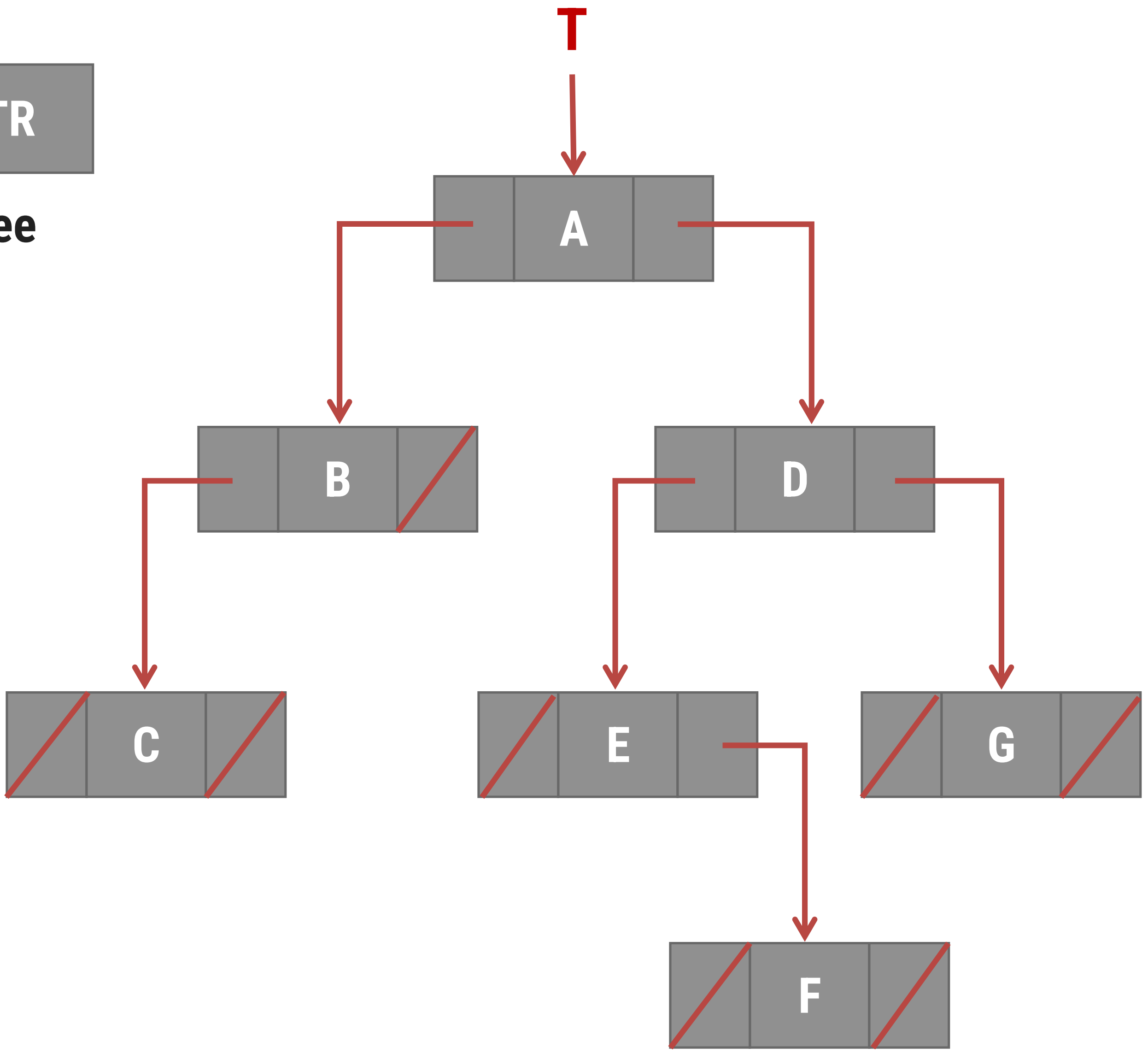


Linked Representation of Binary Tree



Typical node of Binary Tree





Algorithm of Binary Tree Traversal

- Preorder Traversal - Procedure: RPREORDER(T)
- Inorder Traversal - Procedure: RINORDER(T)
- Postorder Traversal - Procedure: RPOSTORDER(T)

Procedure: RPREORDER(T)

- This procedure **traverses the tree** in **preorder**, in a recursive manner.
- **T is root node address** of given binary tree
- Node structure of binary tree is described as below

LPTR	DATA	RPTR

Typical node of Binary Tree

```
1. [Check for Empty Tree]
   IF     T = NULL
   THEN   write (‘Empty Tree’)
          return
   ELSE   write (DATA(T))
2. [Process the Left Sub Tree]
   IF     LPTR (T) ≠ NULL
   THEN   RPREORDER (LPTR (T))
3. [Process the Right Sub Tree]
   IF     RPTR (T) ≠ NULL
   THEN   RPREORDER (RPTR (T))
4. [Finished]
   Return
```

Procedure: RINORDER(T)

- This procedure **traverses the tree** in **InOrder**, in a recursive manner.
- **T is root node address** of given binary tree.
- Node structure of binary tree is described as below.

LPTR	DATA	RPTR

Typical node of Binary Tree

```
1. [Check for Empty Tree]
   IF    T = NULL
   THEN  write (‘Empty Tree’)
         return
2. [Process the Left Sub Tree]
   IF    LPTR (T) ≠ NULL
   THEN  RINORDER (LPTR (T))
3. [Process the Root Node]
   write (DATA(T))
4. [Process the Right Sub Tree]
   IF    RPTR (T) ≠ NULL
   THEN  RINORDER (RPTR (T))
5. [Finished]
   Return
```

Procedure: RPOSTORDER(T)

- This procedure **traverses the tree** in **PostOrder**, in a recursive manner.
- **T is root node address** of given binary tree.
- Node structure of binary tree is described as below.

LPTR	DATA	RPTR

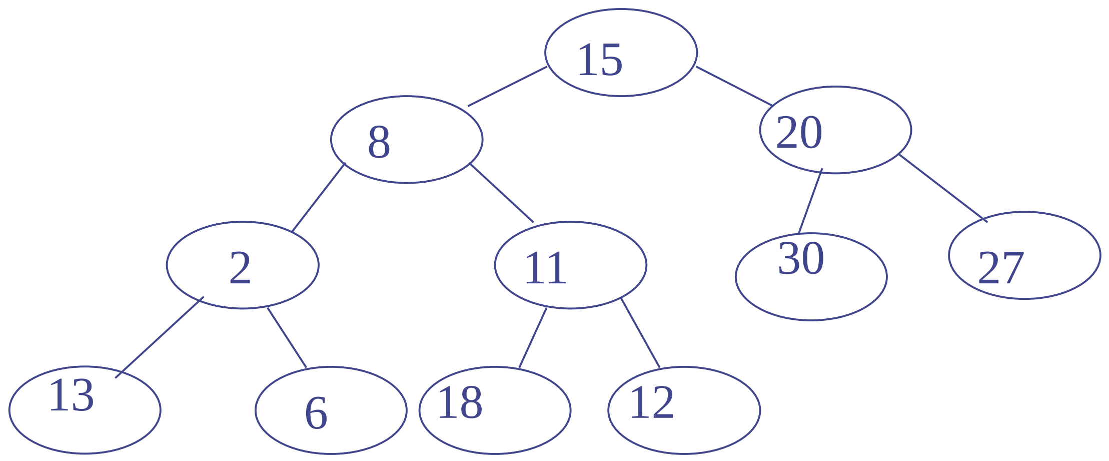
Typical node of Binary Tree

```
1. [Check for Empty Tree]
   IF    T = NULL
   THEN  write ('Empty Tree')
         return
2. [Process the Left Sub Tree]
   IF    LPTR (T) ≠ NULL
   THEN  RPOSTORDER (LPTR (T))
3. [Process the Right Sub Tree]
   IF    RPTR (T) ≠ NULL
   THEN  RPOSTORDER (RPTR (T))
4. [Process the Root Node]
   write (DATA(T))
5. [Finished]
   Return
```

Array Representation of Full Trees and Almost Complete Trees

- A canonically label-able tree, like full binary trees and almost complete binary trees, can be represented by an array A of the same length as the number of nodes
- A[k] is identified with node of label k
- That is, A[k] holds the data of node k
- Advantage:
  - no need to store left and right pointers in the nodes □ save memory
  - Direct access to nodes: to get to node k, access A[k]

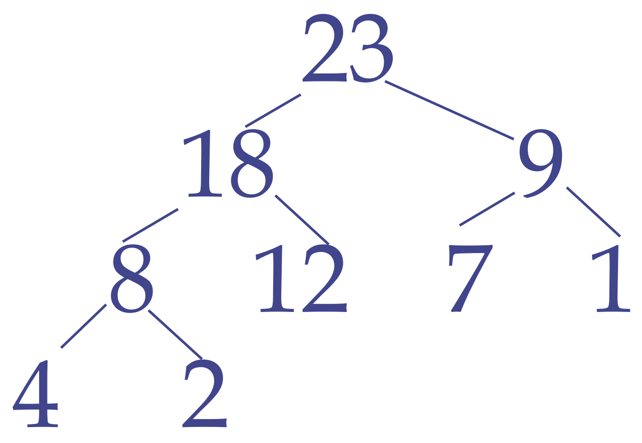
Illustration of Array Representation



15	8	20	2	11	30	27	13	6	18	12
1	2	3	4	5	6	7	8	9	10	11

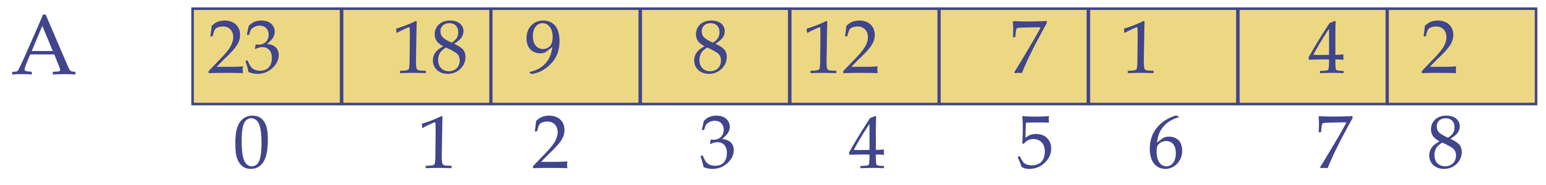
- Notice: Left child of A[5] (of data 11) is A[2*5]=A[10] (of data 18), and its right child is A[2*5+1]=A[11] (of data 12).
- Parent of A[4] is A[4/2]=A[2], and parent of A[5]=A[5/2]=A[2]

Array Representation [For index 0]



for the item in A[i]:

leftChild	is in A[2i+1]
rightChild	is in A[2i+2]
parent	is in A[(i-1)/2]



0	1	2	3	4	5	6	7	8
23	18	9	8	12	7	1	4	2

Size

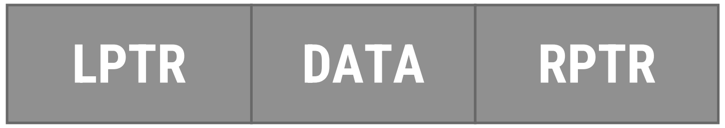
9

Tree Structure

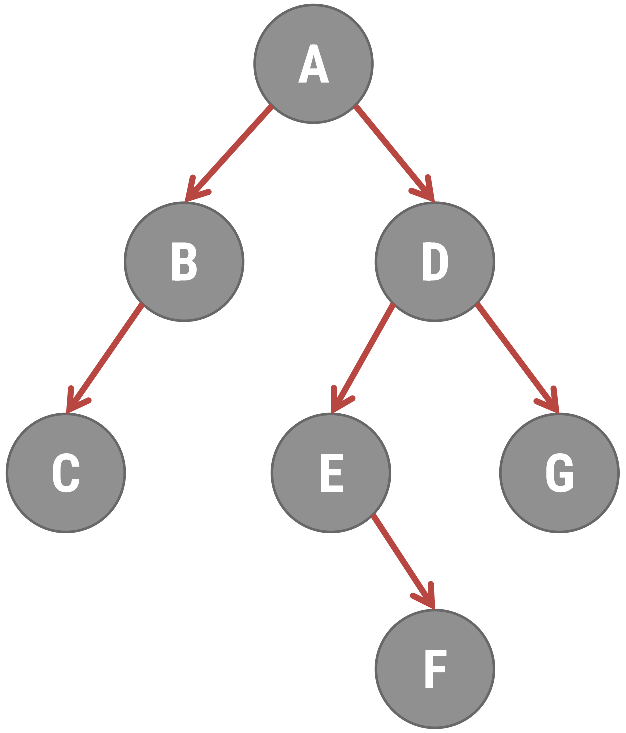
A node of a binary tree can be represented in C as follows:

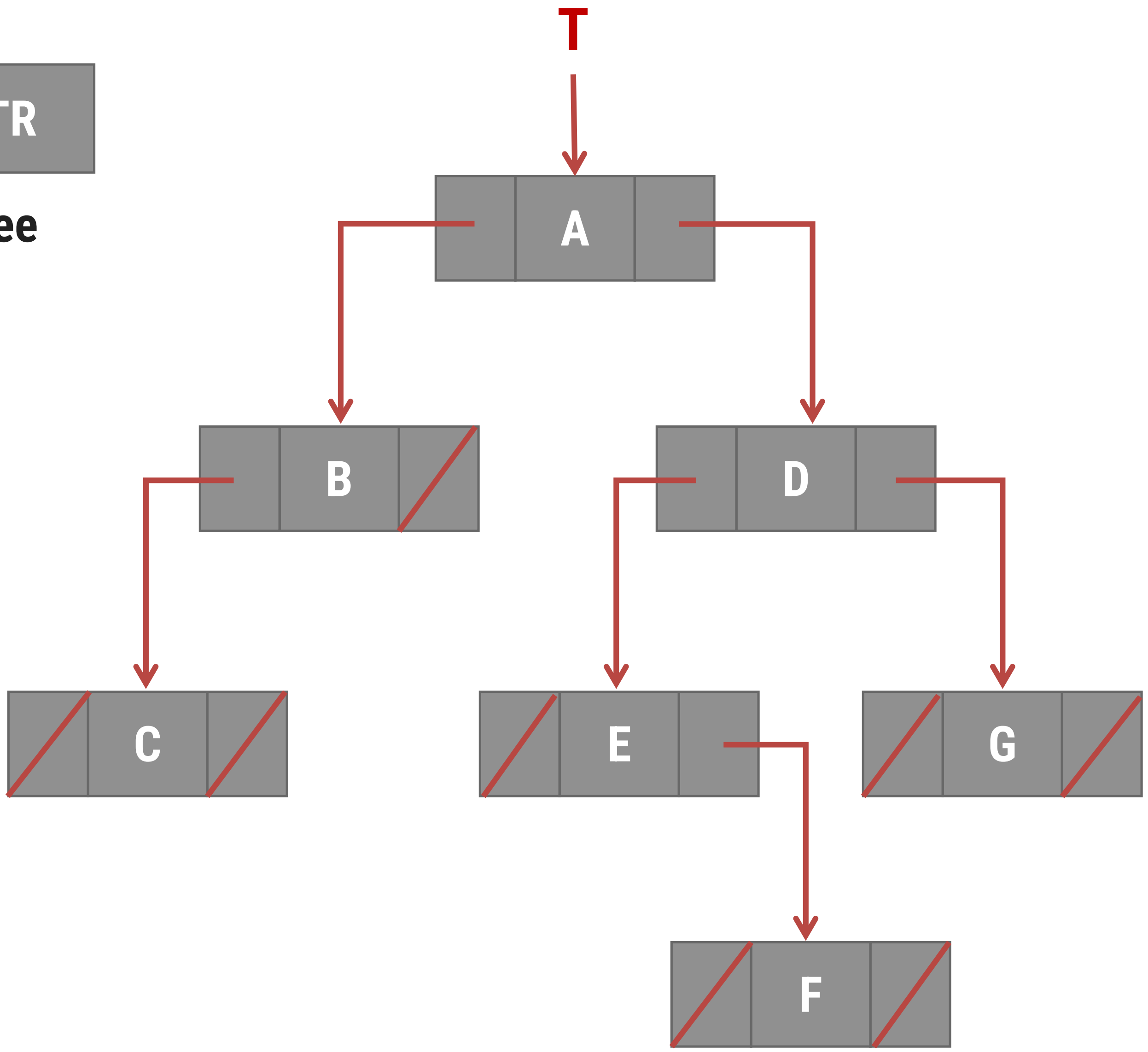
```
struct BTreeNode {
   int data;
   BTreeNode *left;
   BTreeNode *right;
};
```

Linked Representation of Binary Tree



Typical node of Binary Tree



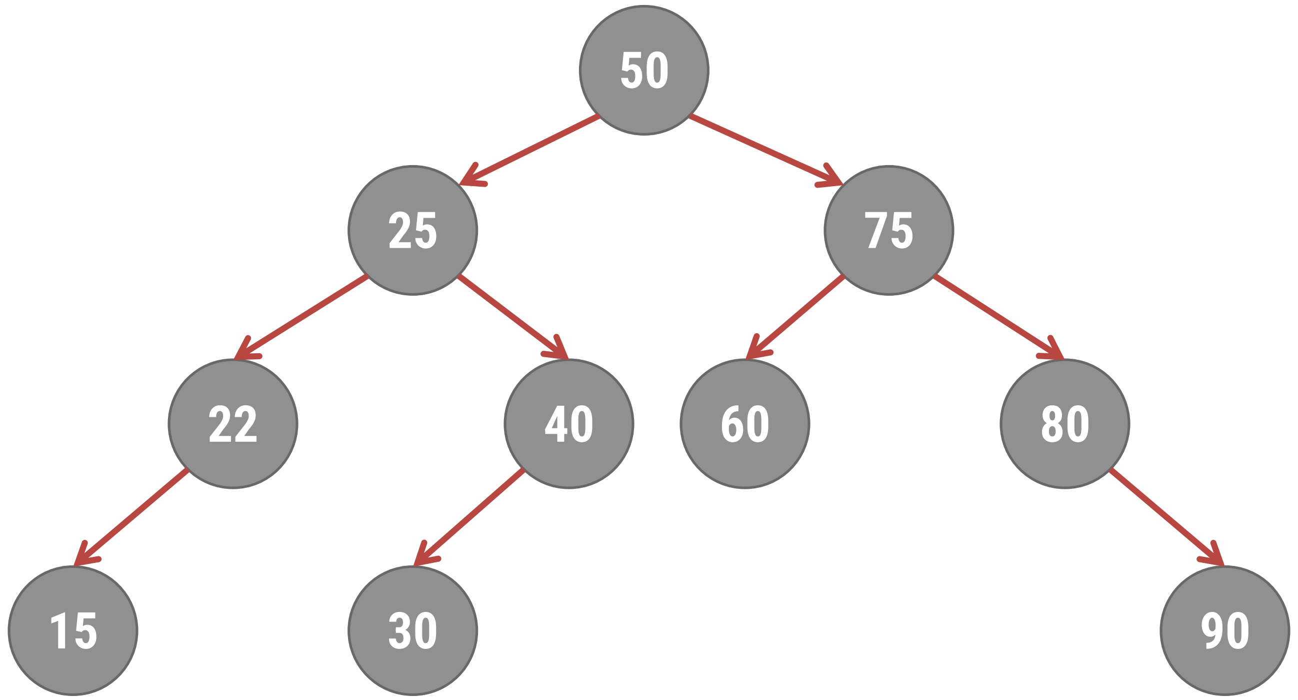


Binary Search Tree (BST)

- A **binary search tree** is a **binary tree** in which **each node** possessed a key that **satisfy** the **following conditions**
  1. All **key** (if any) in **the left sub tree** of the root **precedes the key** in the **root**
  2. The **key in the root precedes** all **key** (if any) in the **right sub tree**
  3. The **left and right sub trees** of the root are again **search trees**

Construct Binary Search Tree (BST)

Construct binary search tree for the following data
50 , 25 , 75 , 22 , 40 , 60 , 80 , 90 , 15 , 30



Construct binary search tree for the following data
10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5

Search a node in Binary Search Tree

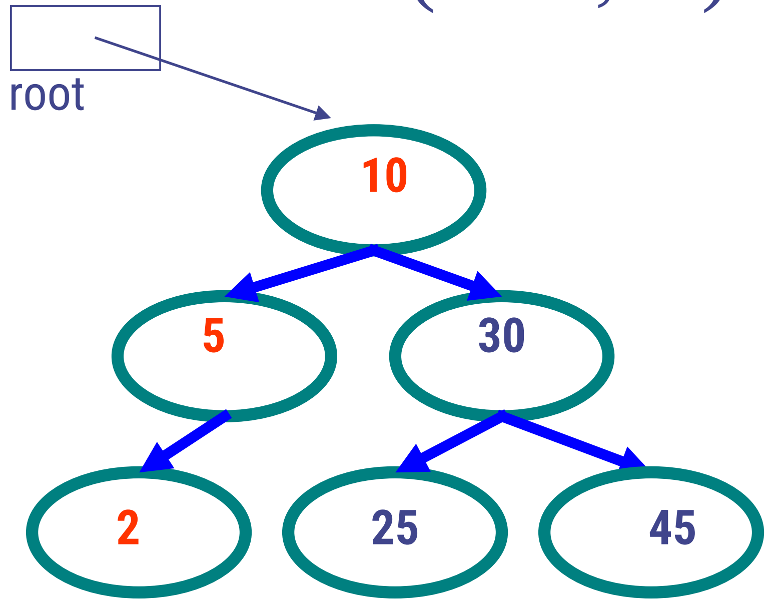
- To search for target value.
- We first compare it with the key at root of the tree.
- If it is not same, we go to either Left sub tree or Right sub tree as appropriate and repeat the search in sub tree.
- If we have **In-Order List** & we want to search for specific node it requires **O(n) time.**
- In case of **Binary tree** it requires **O(Log$_2$n)** time to search a node.

Binary Search Tree

- Binary search tree definition
  - A set of nodes T is a binary search tree if either of the following is true
    - T is empty
    - Its root has two subtrees such that each is a binary search tree and the value in the root is greater than all values of the left subtree but less than all values in the right subtree
- A data structure for efficient searching, insertion and deletion

Example Binary Searches

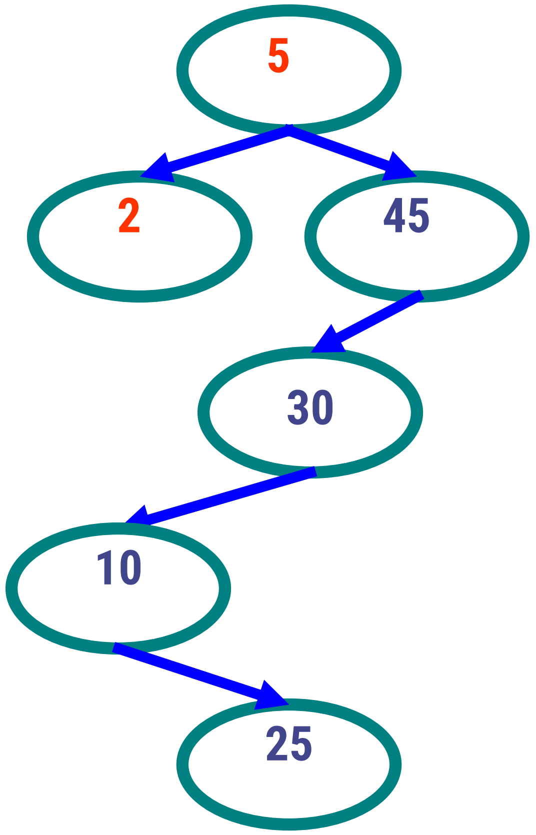
- Find (root, 2)



10 > 2, left

5 > 2, left

2 = 2, found

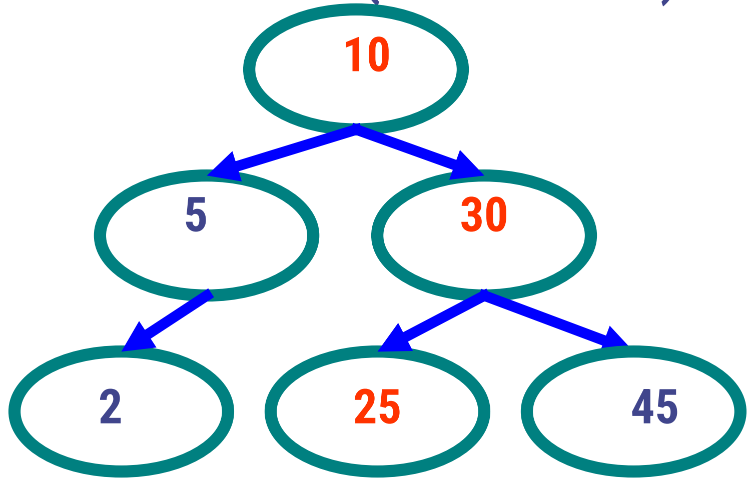


5 > 2, left

2 = 2, found

Example Binary Searches

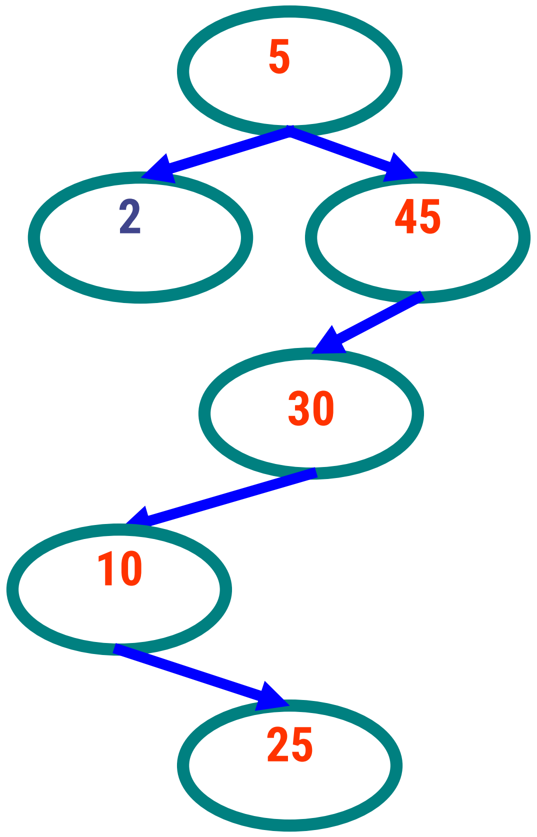
- Find (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

30 > 25, left

10 < 25, right

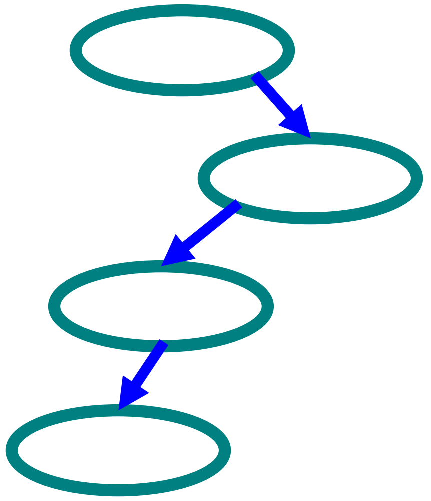
25 = 25, found

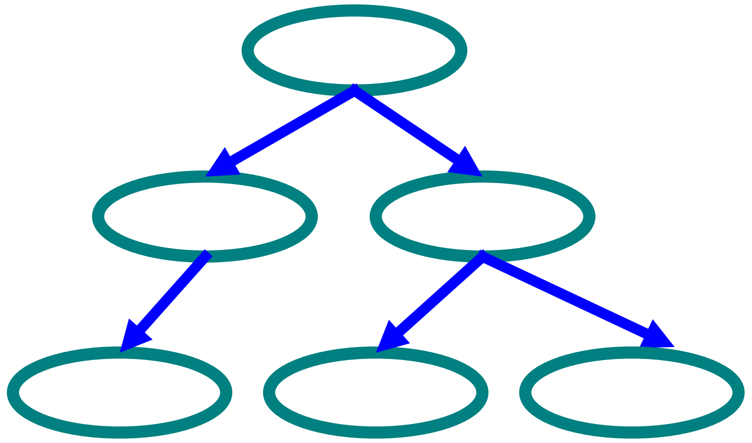
Binary Search Algorithm

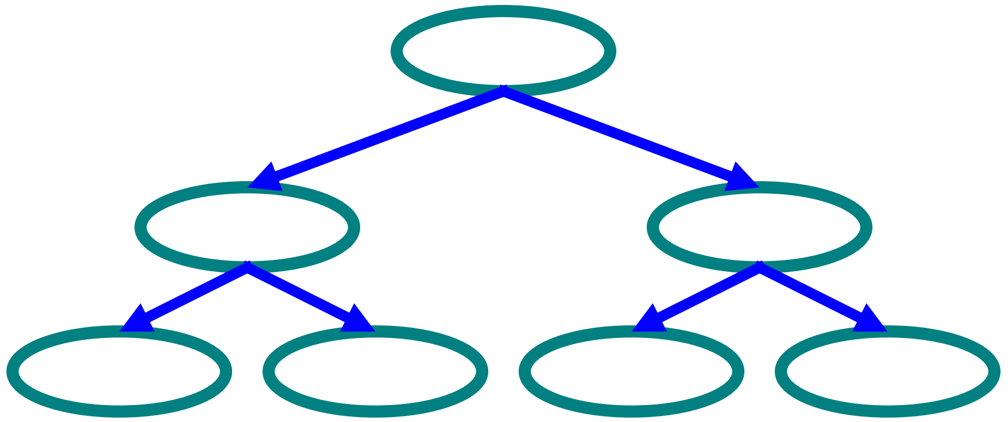
```
Search_Node(Node, Info)
Step1:  [Initialize]
        Flag=0
Step2:  Repeat through step 3 while Node # NULL
Step3:  If Info[Node]=Info
            Flag=1
            Return(Flag)
        else if Info< Info[Node]
            Node=Left_Child[Node]
        else    Node=Right_Child[Node]
Step4:  Return (Flag)
```

Types of Binary Trees

- Degenerate – only one child
- Complete – always two children
- Balanced – “mostly” two children
  - Height of the left and right subtree of any node differ by not more than 1 (also called height-balanced binary tree).


Degenerate binary tree

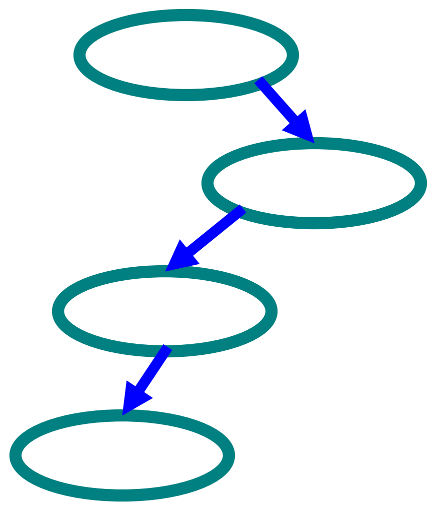

Balanced binary tree

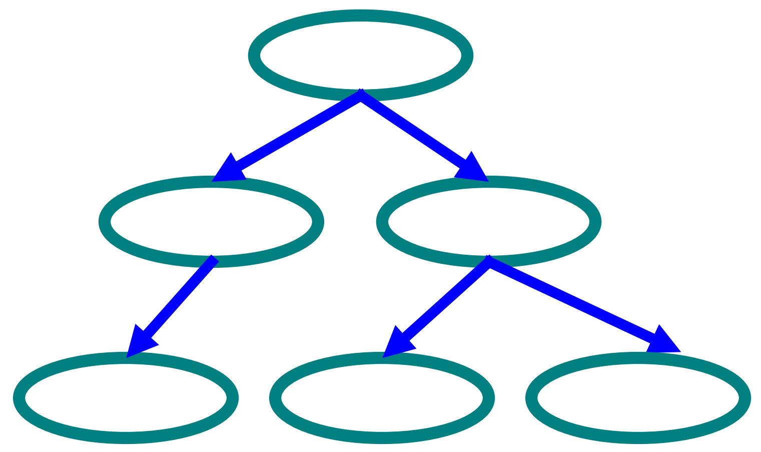

Complete binary tree

Binary Trees Properties

- Degenerate
  - Height = O(n) for n nodes
  - Similar to linked list

- Balanced
  - Height = O(log(n)) for n nodes
  - Useful for searches


Degenerate binary tree


Balanced binary tree

Binary Search Properties

- Time of search
  - Proportional to height of tree
  - Balanced binary tree
    - O(log(n)) time
  - Degenerate tree
    - O(n) time
    - Like searching linked list / unsorted array

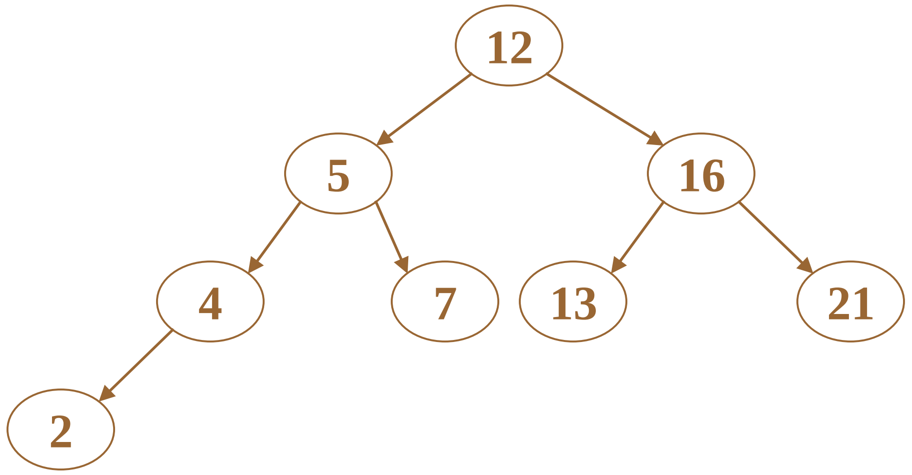
Binary Search Tree Construction

- How to build & maintain binary trees?
  - Insertion
  - Deletion
- Maintain key property (invariant)
  - Smaller values in left subtree
  - Larger values in right subtree

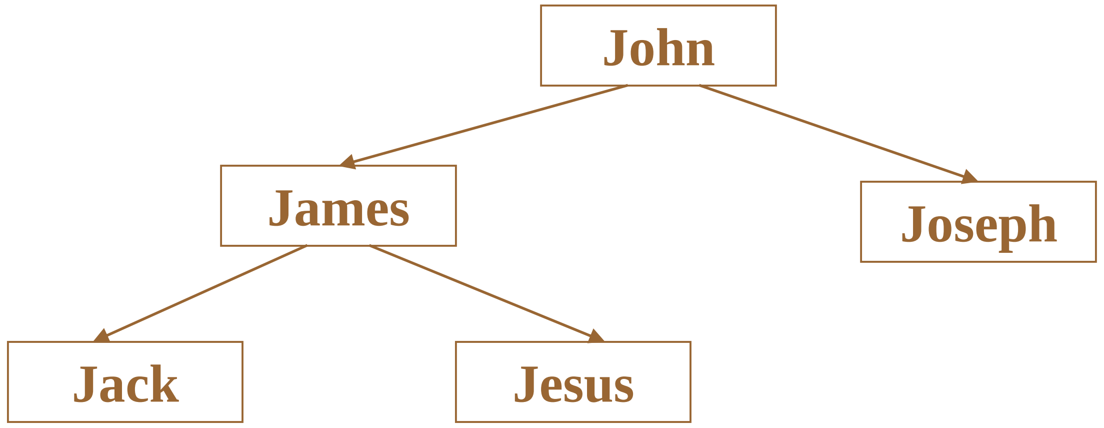
Binary Search Tree – Insertion

- Algorithm
  1. Perform search for value X
  2. Search will end at node Y (if X not in tree)
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y
- Observations
  - O(log(n)) operation for balanced tree
  - Insertions may unbalance tree

Examples

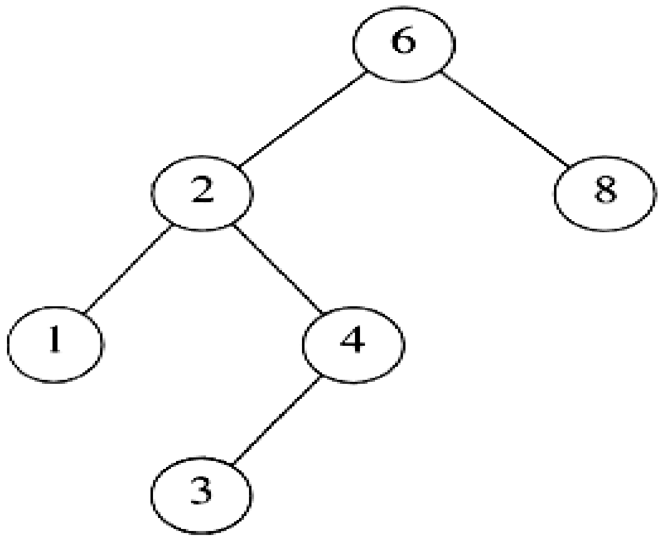


BST for numbers

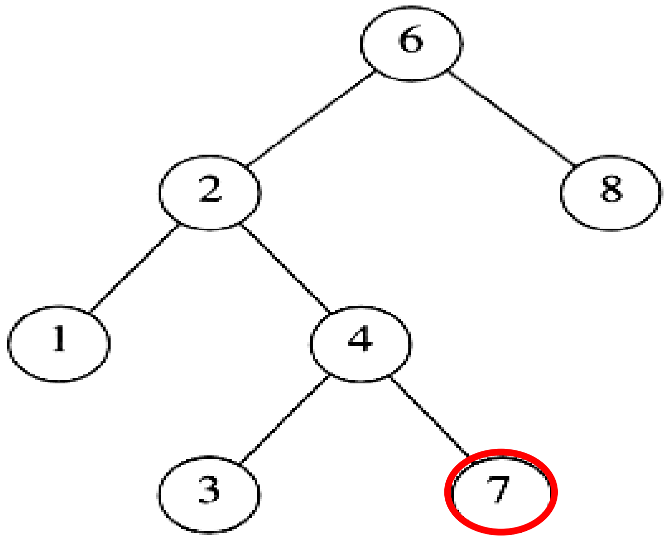


BST for names

Binary Search Trees



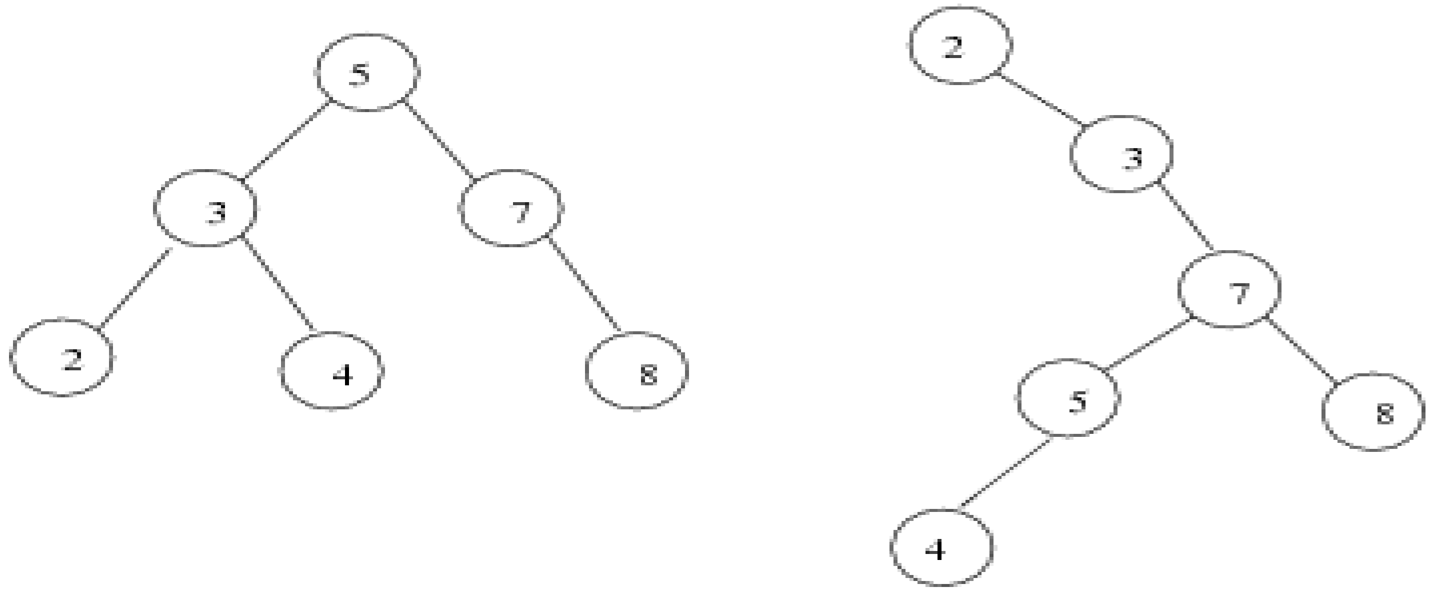
A binary search tree



Not a binary search tree

Binary Search Trees

The same set of keys may have different BSTs



- Average depth of a node is O(log N)
- Maximum depth of a node is O(N)

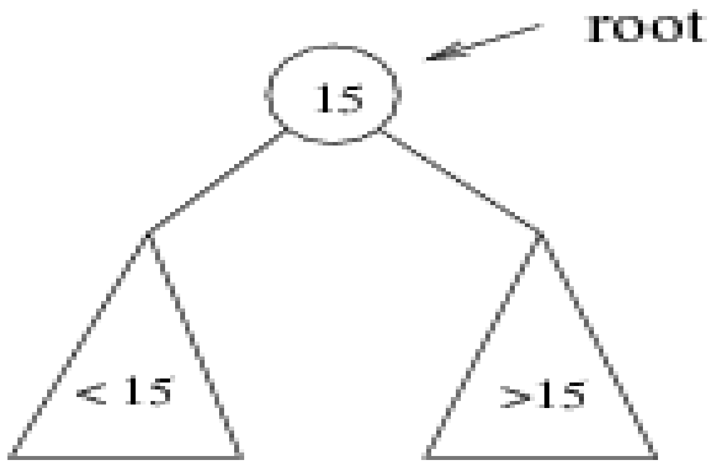
Constructing a BST

- **Base case:**
if tree is empty, create new node for item

- **Recursive case:**
if key < root's value, add to the left subtree, otherwise to the right subtree

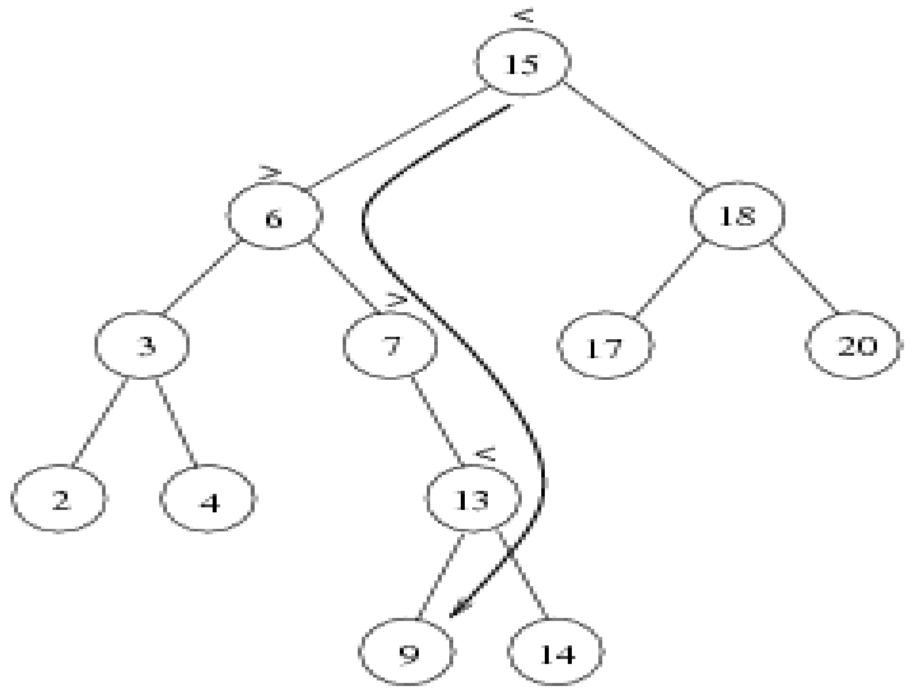
Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Example: Search for 9 ...

Searching BST

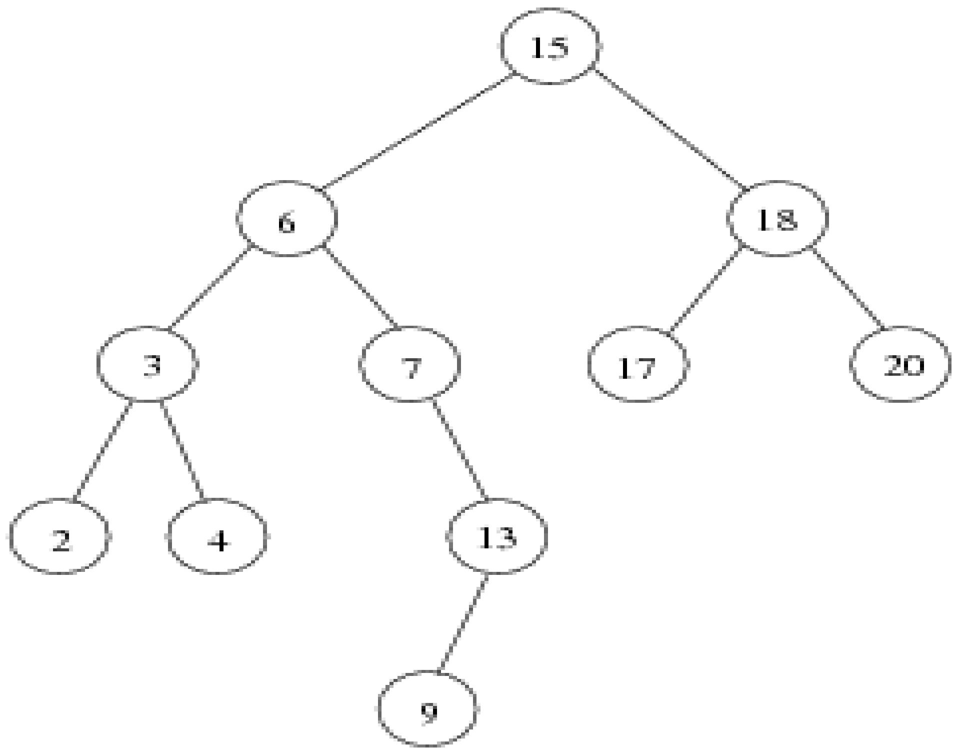


Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Inorder Traversal of BST

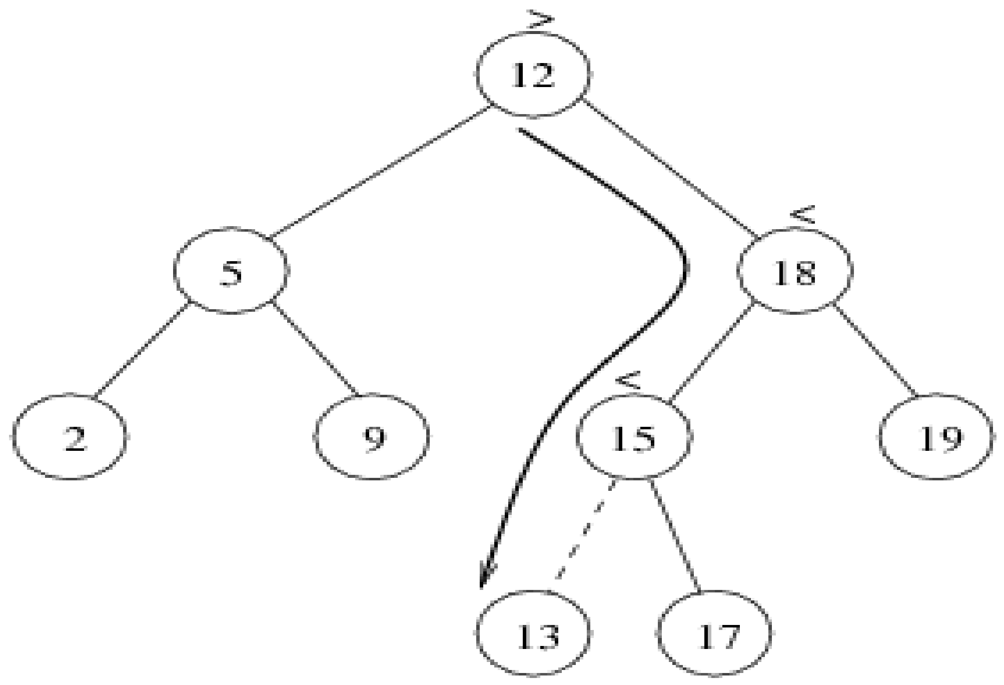
- Inorder traversal of BST prints out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Insertion

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed



- Time complexity = O(height of the tree)

Inserting an Element in a Binary Search Tree

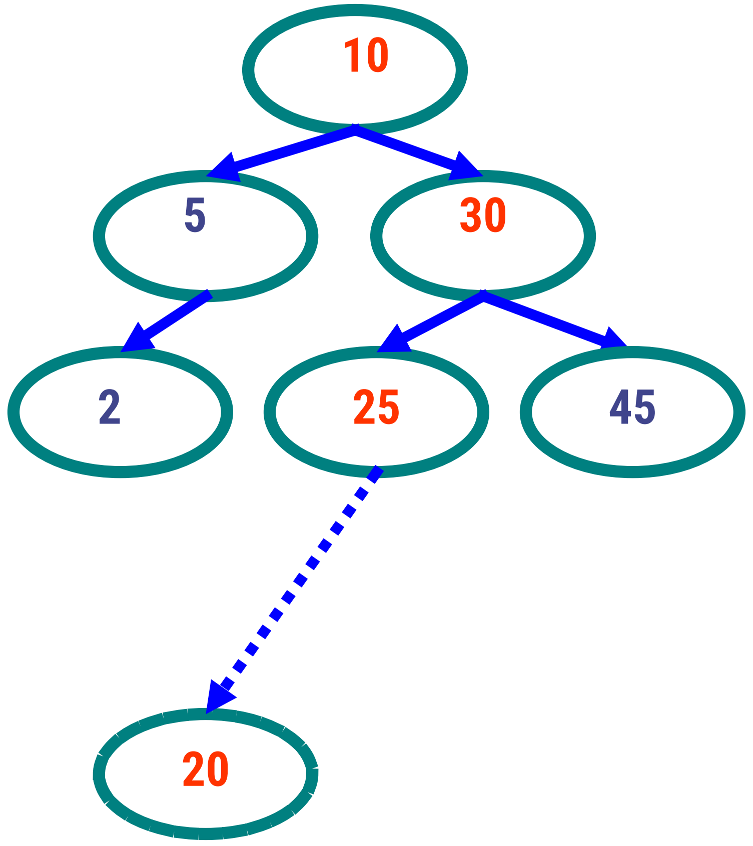
- Inserting an element in a binary search tree involves creating a new node and re-organizing.
- The parent nodes link to accommodate the new node.
- Typically insertions will always be on a terminal node.
- We will never be inserting a node in the middle of a binary search tree.

Steps for inserting an element in a binary search tree

1. If the tree is empty, then make the root node point to this node
2. If the tree contains nodes then compare the data with node's data and take appropriate path till the terminal node is reached
3. If data < node's data then take the left sub tree otherwise take the right sub tree
4. When there are no more nodes add the node to the appropriate place of the parent node.

Example Insertion

- Insert (20)



10 < 20, right

30 > 20, left

25 > 20, left

Insert 20 on left

Deletion

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.

Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass that node

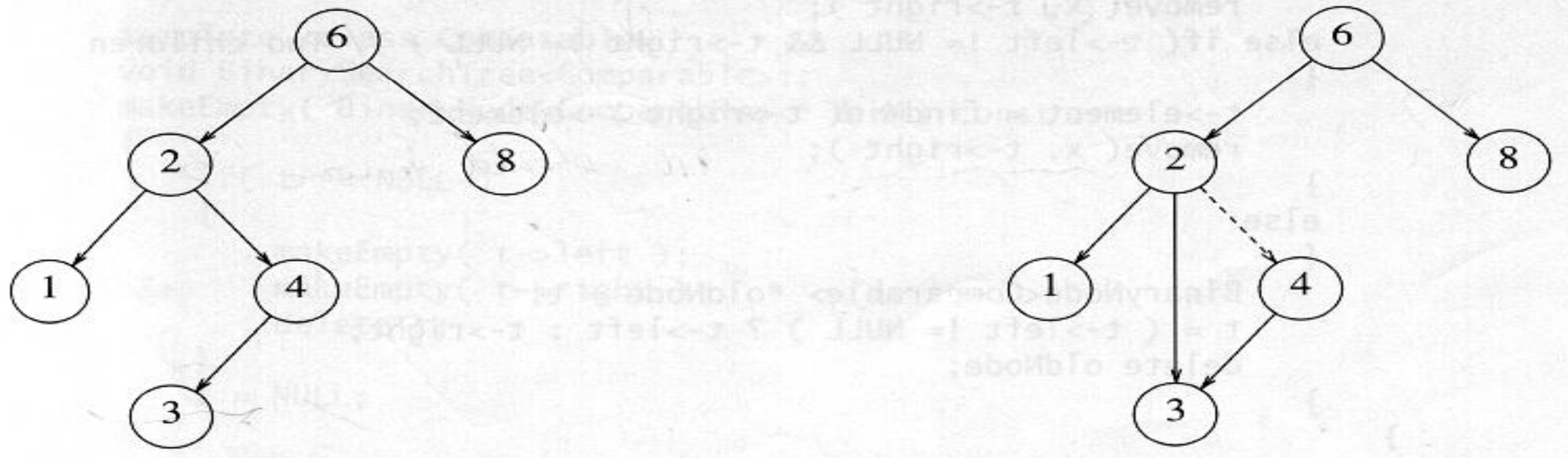


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion Case 3

- Case 3: the node has 2 children
  - Replace the key of that node with the minimum element at the right subtree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

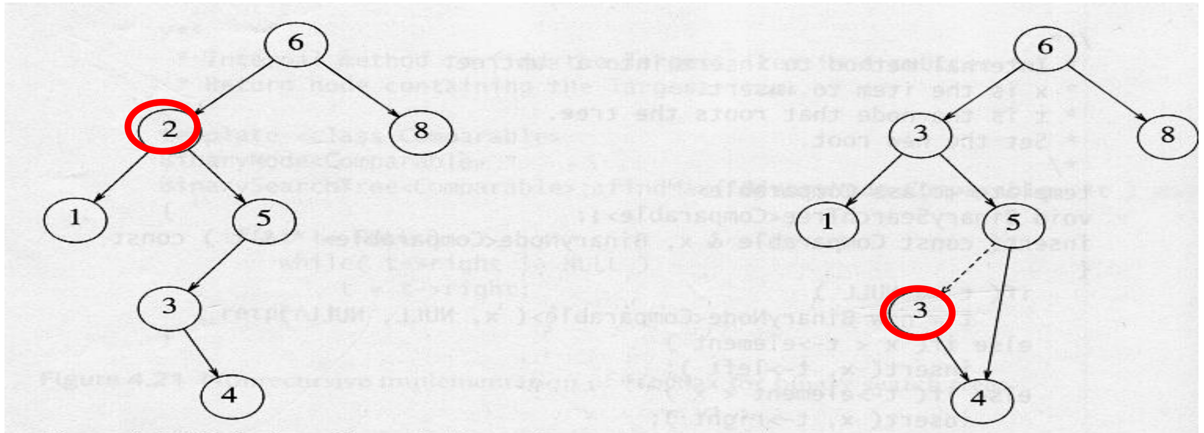


Figure 4.25 Deletion of a node (2) with two children, before and after

Deleting an element from a binary tree

a) If the node being deleted is a terminal node then deletion is a straightforward procedure of just making the parent node point to

NULL.

b) If the node being deleted has only one child then the parent node will have to point to the child node of the node being deleted.

c) If the node being deleted has both the child's then we first need to find the inorder successor of the node being deleted and replace the node being deleted with this node. (The inorder successor of a node can be obtained by taking the right node of the current node and traversing in the left till we reach the left most node.)

Binary Search Tree – Deletion

- Algorithm
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else // must delete internal node

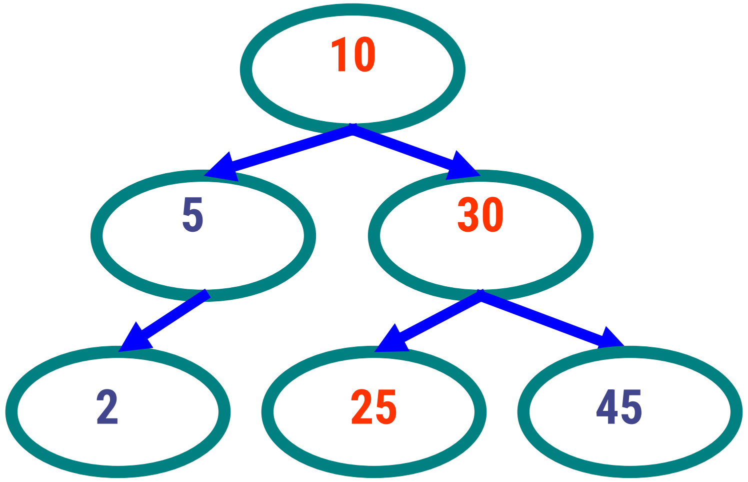
     a) Replace with largest value Y on left subtree

     OR smallest value Z on right subtree

     b) Delete replacement value (Y or Z) from subtree
- Observation
  - O(log(n)) operation for balanced tree
  - Deletions may unbalance tree

Example Deletion (Leaf)

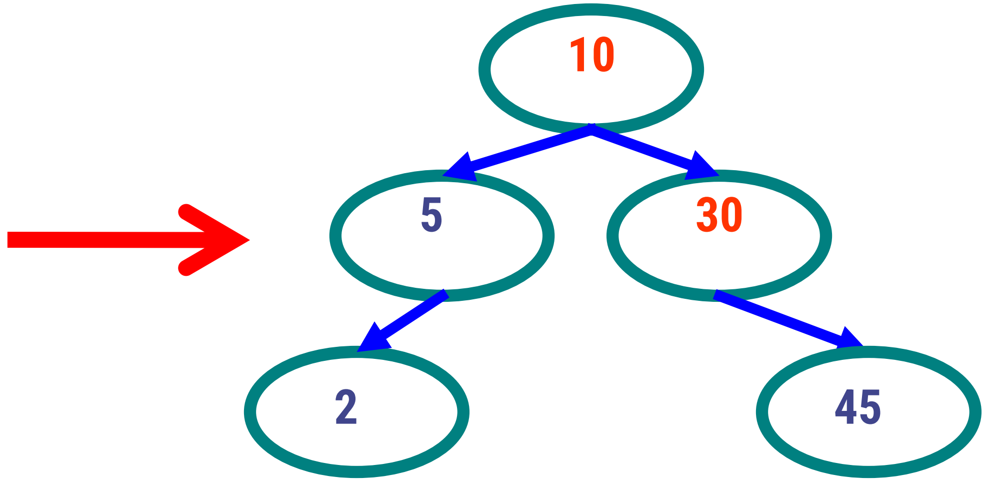
- Delete (25)



10 < 25, right

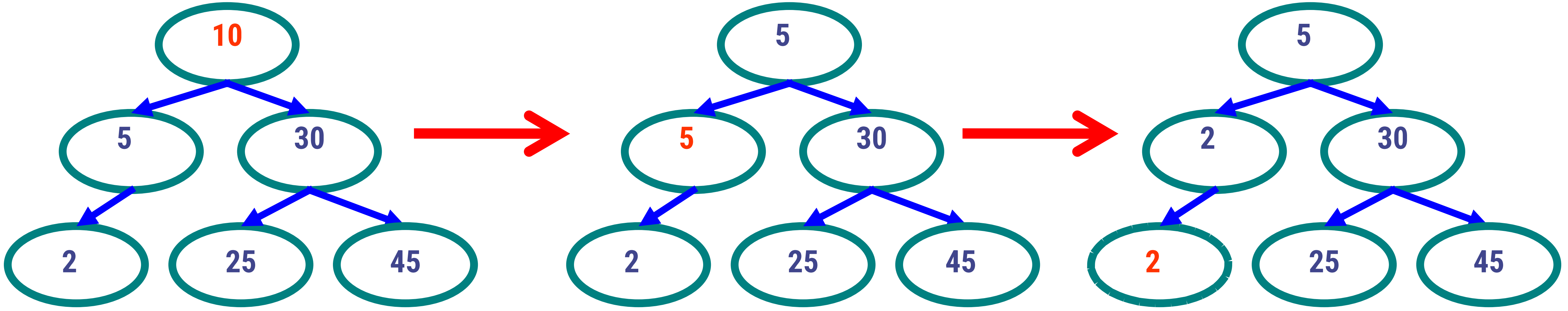
30 > 25, left

25 = 25, delete



Example Deletion (Internal Node)

- Delete (10)



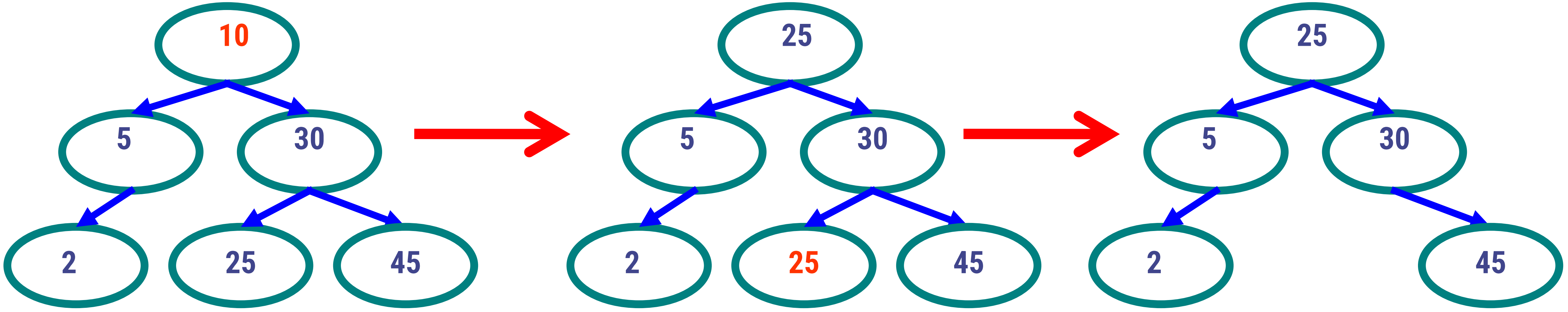
Replacing 10 with **largest** value in left subtree

Replacing 5 with **largest** value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)

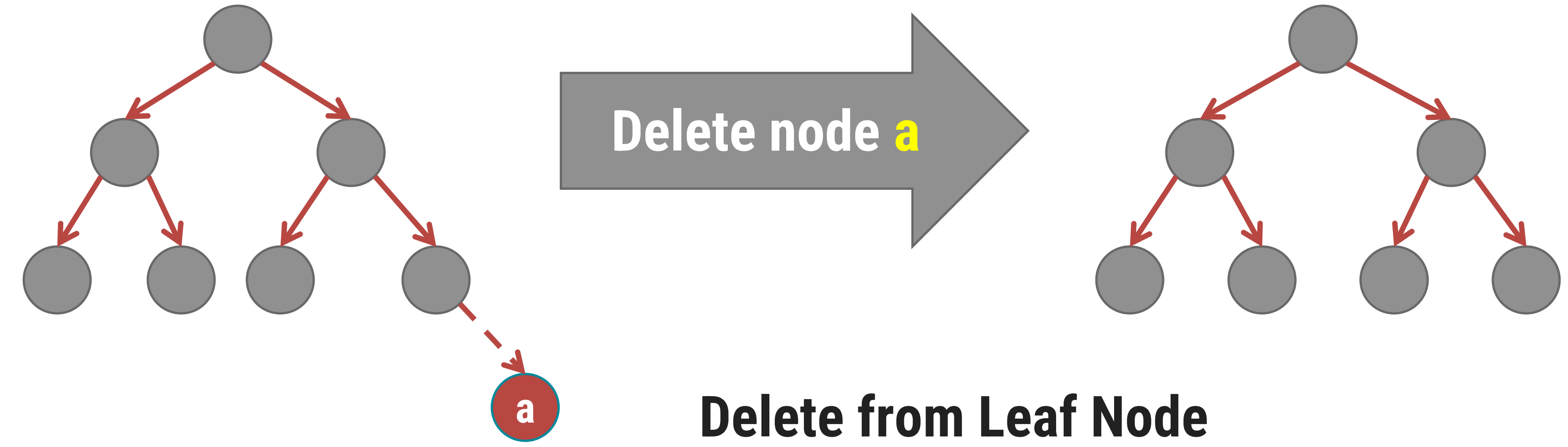


Replacing 10 with **smallest** value in right subtree

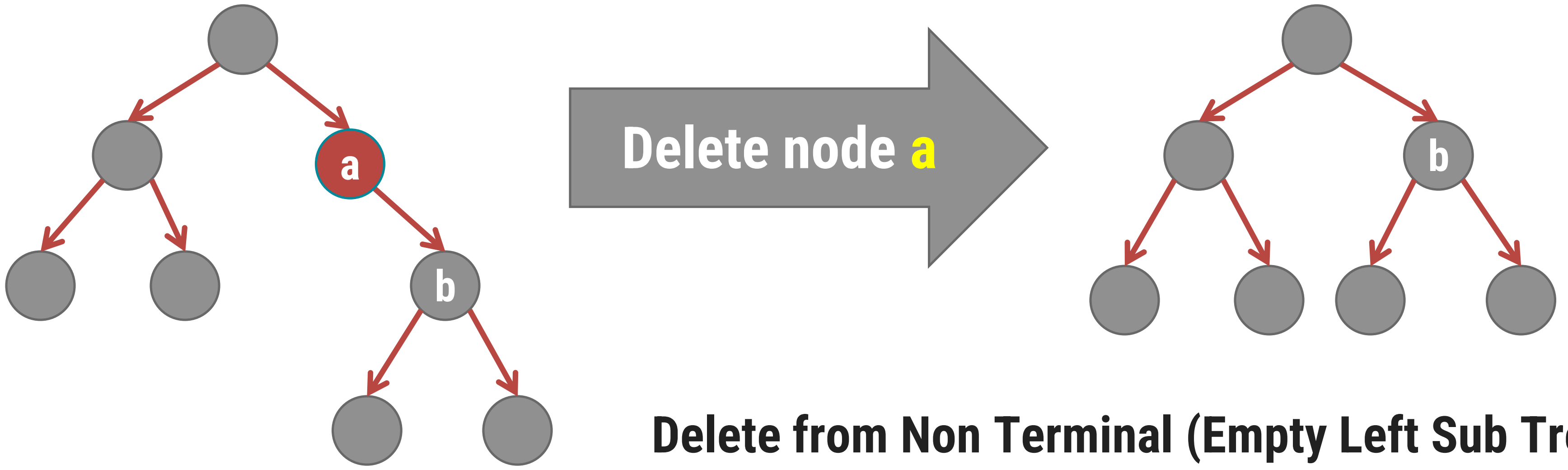
Deleting leaf

Resulting tree

Delete node from Binary Search Tree

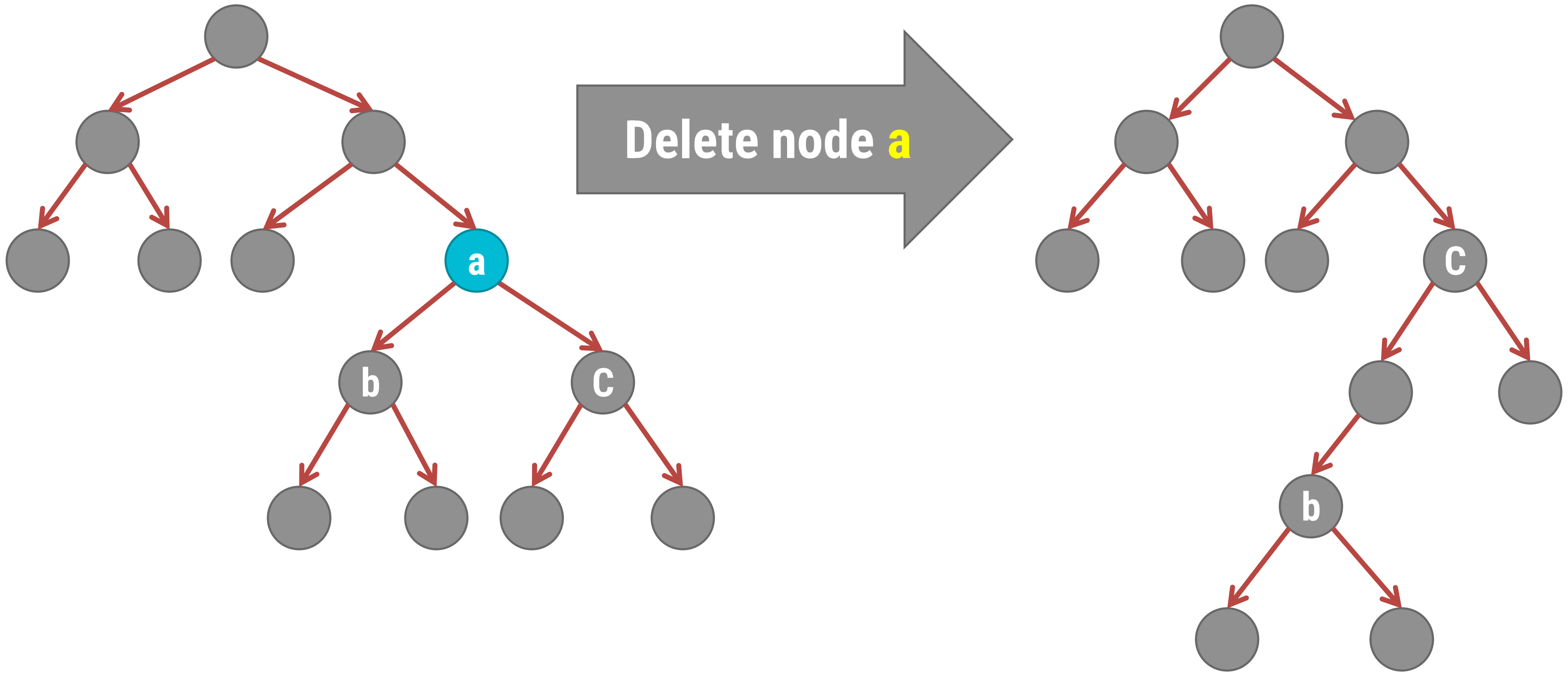


Delete from Leaf Node



Delete from Non Terminal (Empty Left Sub Tree)

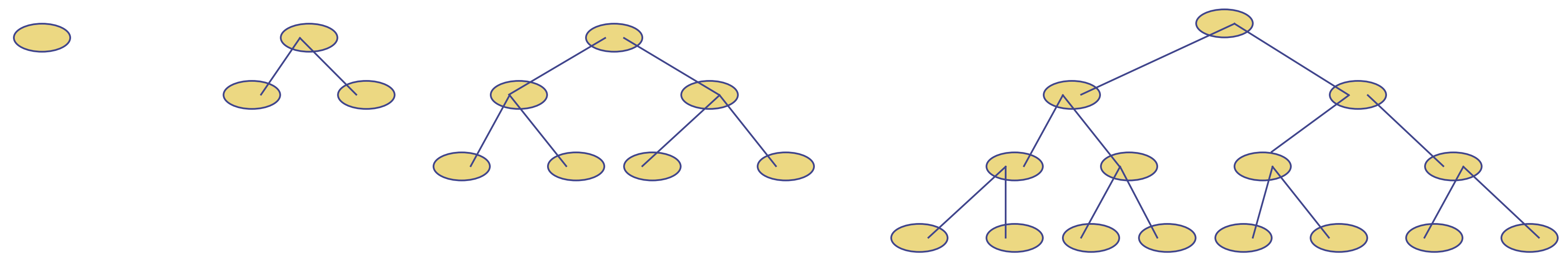
Delete node from BST



Delete from Non Terminal (Neither Sub Tree is Empty)

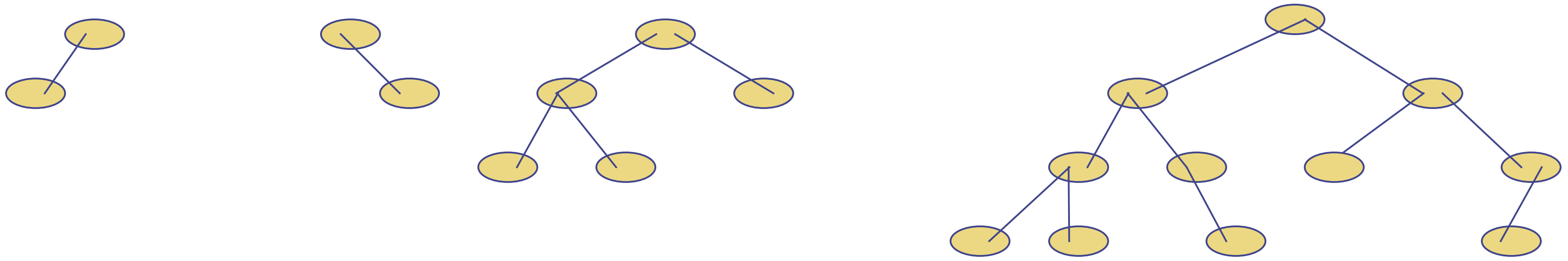
Other Kinds of Binary Trees (Full Binary Trees)

- **Full Binary Tree**: A full binary tree is a binary tree where all the leaves are on the same level and every non-leaf has two children
- The first four full binary trees are:



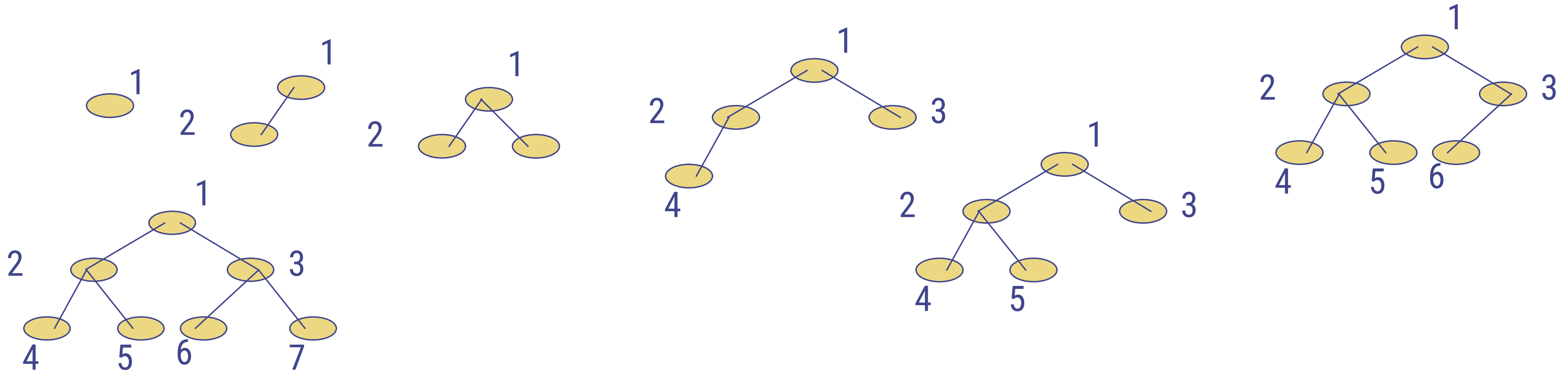
Examples of Non-Full Binary Trees

- These trees are NOT full binary trees:



Other Kinds of Binary Trees (Almost Complete Binary trees)

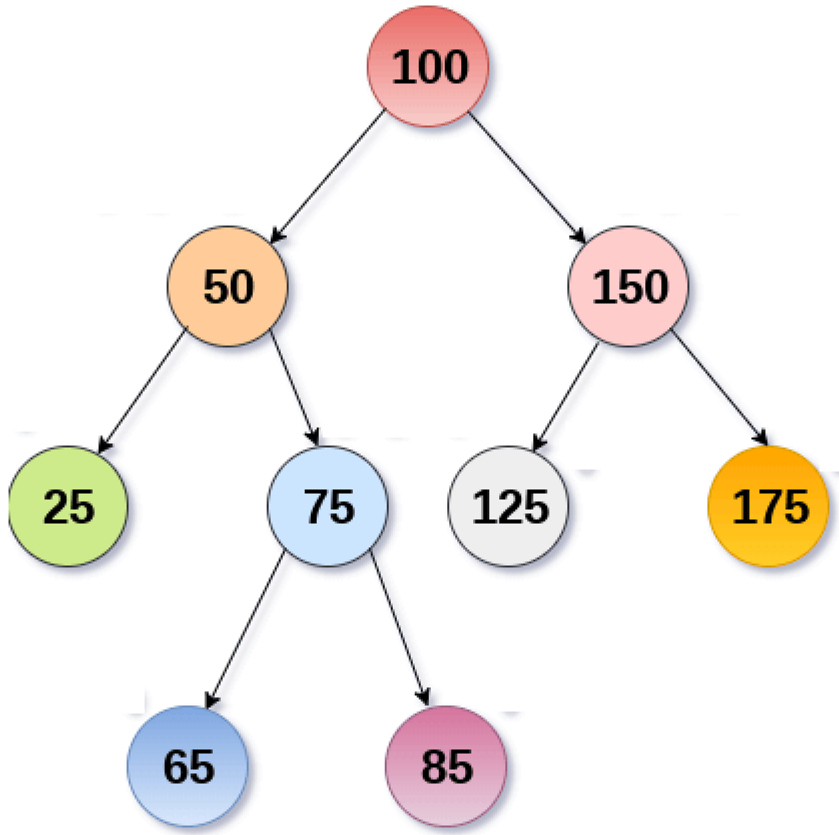
- **Almost Complete <u>Binary Tree</u>**: An almost complete binary tree of n nodes, for any arbitrary nonnegative integer n, is the binary tree made up of the first n nodes of a canonically labeled full binary



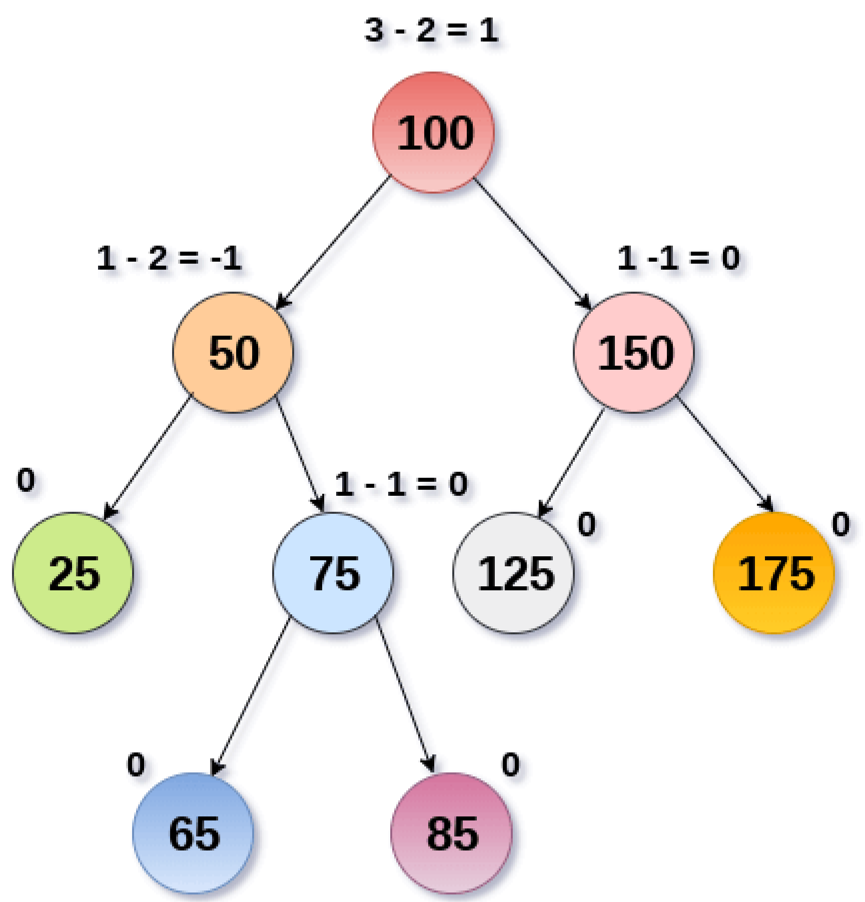
AVL Tree

- AVL Tree can be defined as height balanced binary search tree
- Each node is associated with a balance factor
- Balance Factor (k) = height (left(k)) - height (right(k))
- 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.

- 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.



AVL Tree



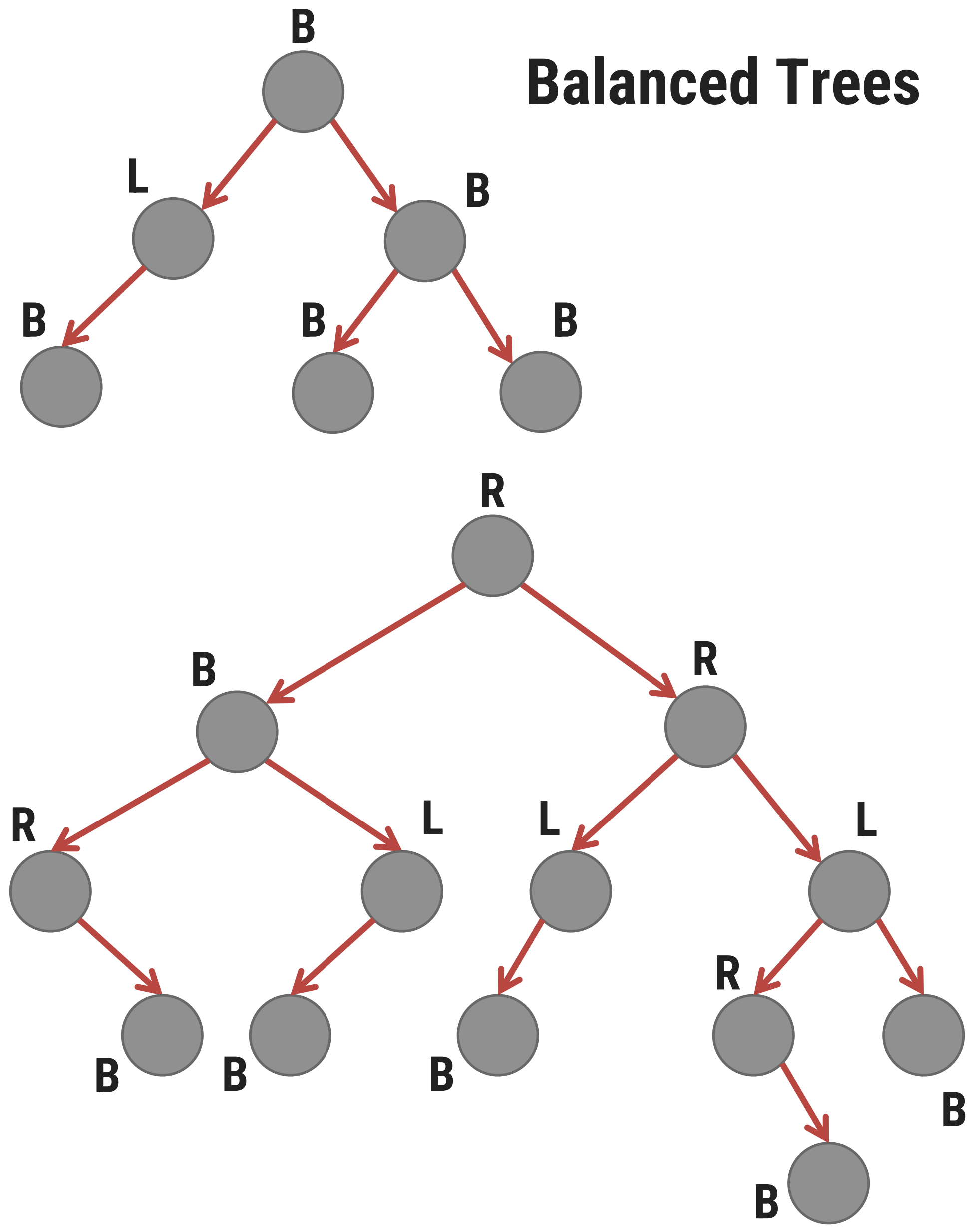
AVL Tree

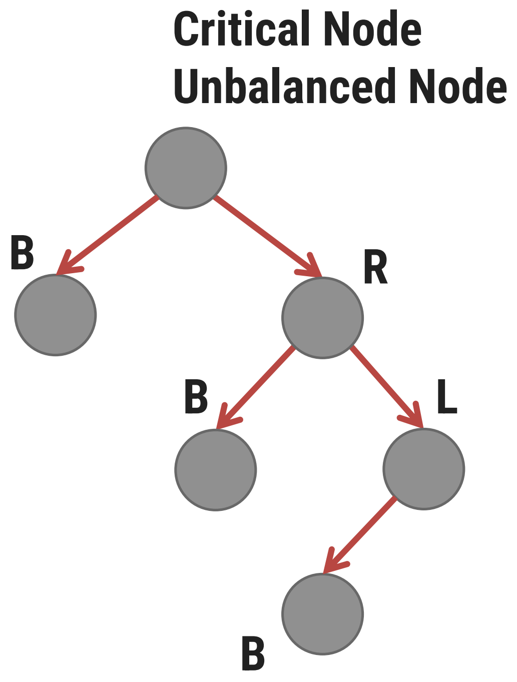
Height Balanced Tree (AVL Tree)

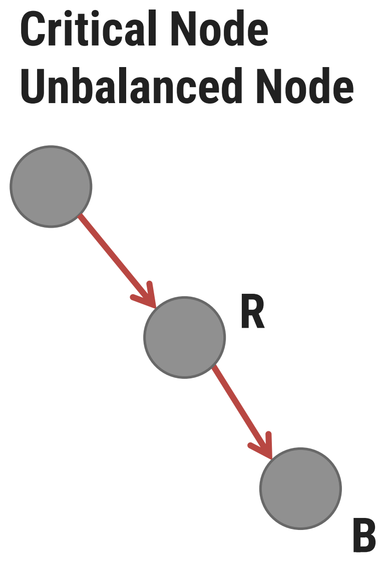
- A tree is called **AVL tree (Height Balanced Tree)**, if each node possessed one of the following properties
  - A **node** is called **left heavy**, if the **longest path in its left sub tree** is **one** longer than the **longest path of its right sub tree**
  - A **node** is called **right heavy**, if the **longest path in its right subtree** is **one** longer than **the longest path of its left sub tree**
  - A **node** is called **balanced**, if the longest path in **both the right and left sub-trees** are equal
- In height balanced tree, each node must be in one of these states
- If there exists a node in a tree where this is not true, then such a tree is called **Unbalanced**

AVL Tree

Balanced Trees





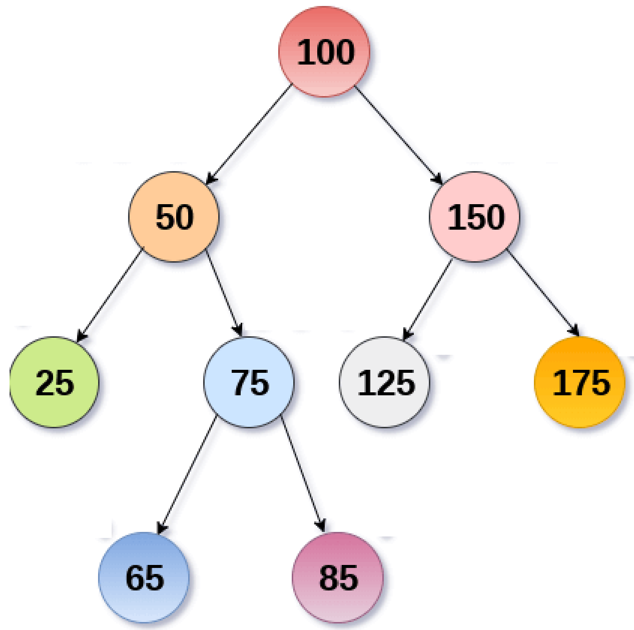


- Sometimes tree becomes unbalanced by inserting or deleting any node
- Then based on position of insertion, we need to rotate the unbalanced node
- **Rotation** is the **process** to **make tree balanced**

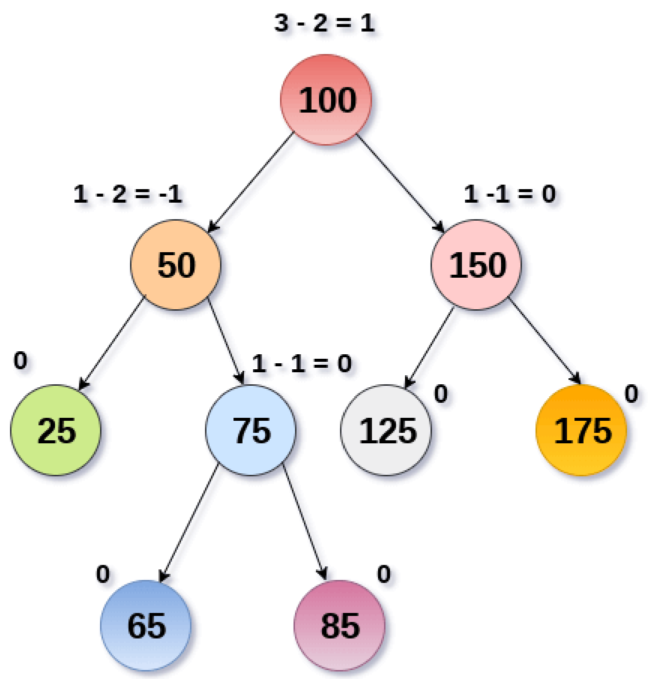
AVL Tree

- AVL Tree can be defined as height balanced binary search tree
- Each node is associated with a balance factor
- Balance Factor (k) = height (left(k)) - height (right(k))
- 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.

- 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.



AVL Tree



AVL Tree