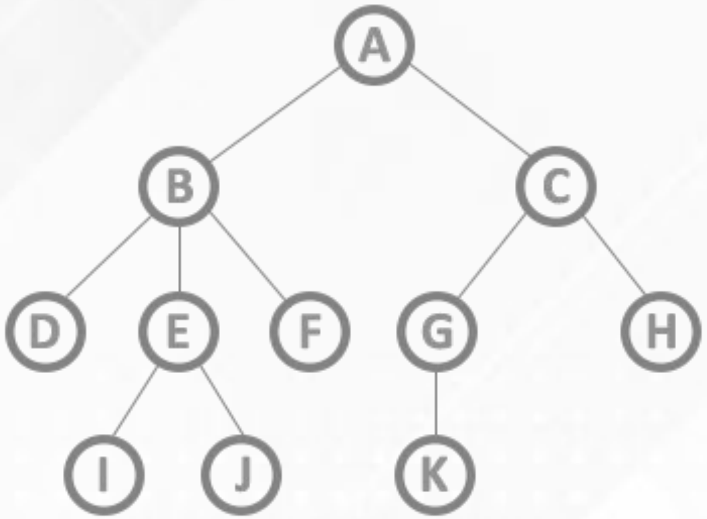
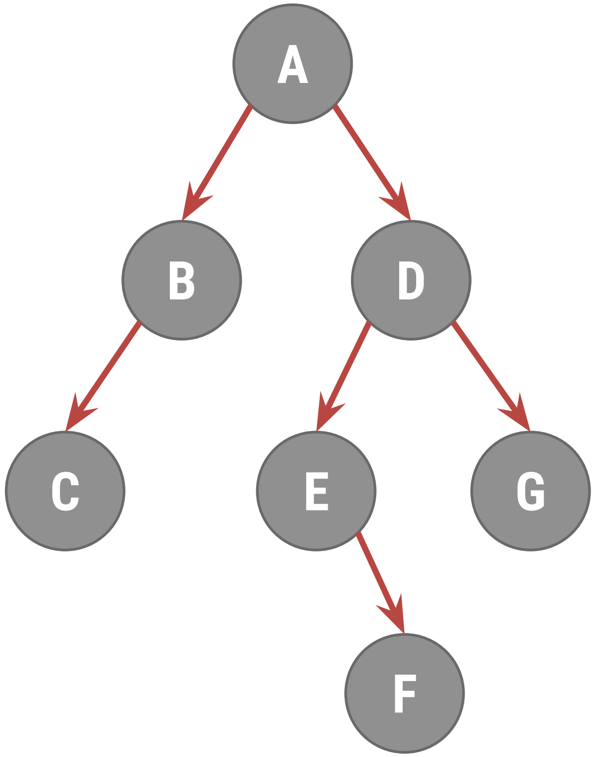
Unit-3

Non-Linear Data Structure (Tree Part-2)



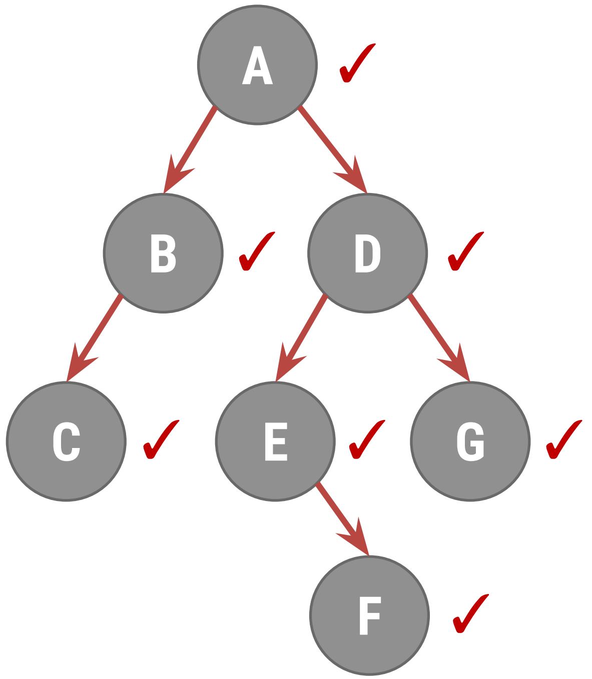
Tree Traversal

- 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.

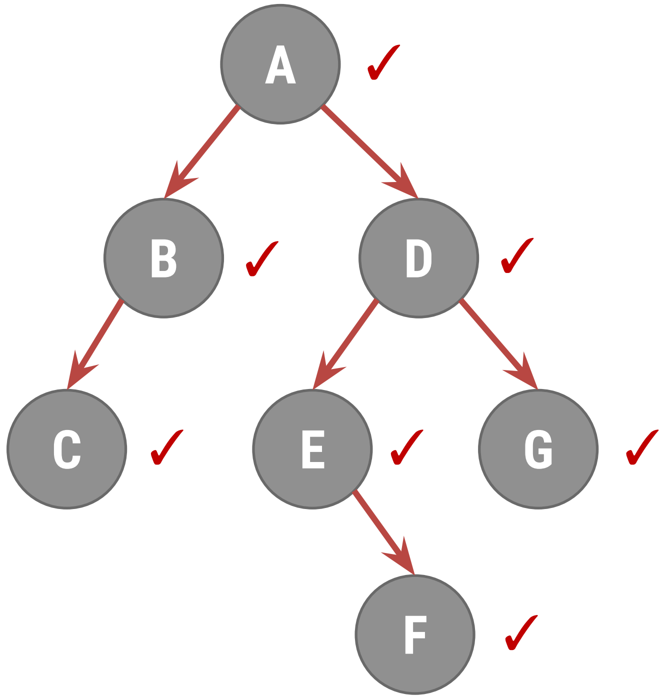


Preorder traversal of a given tree as

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

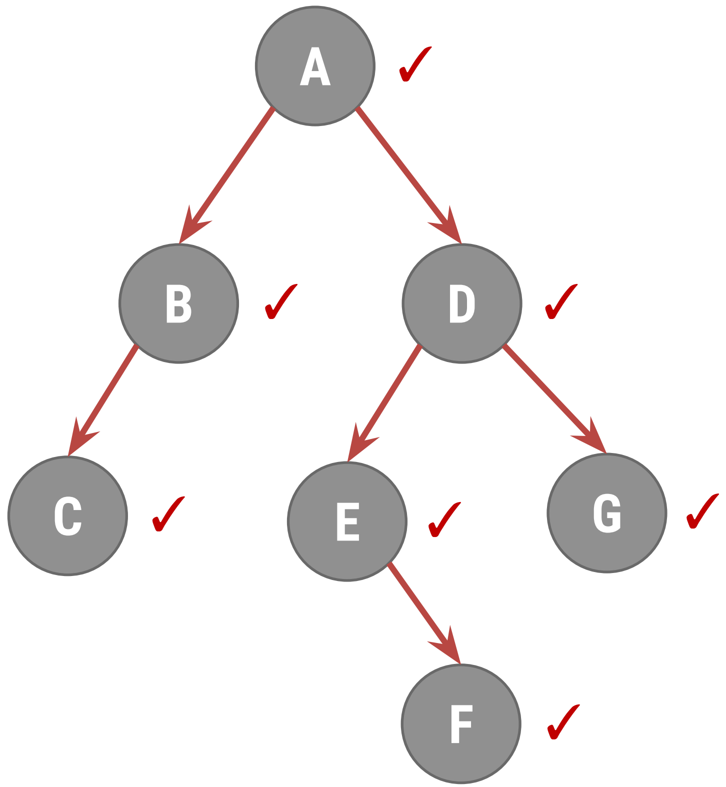


Inorder traversal of a given tree as

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**



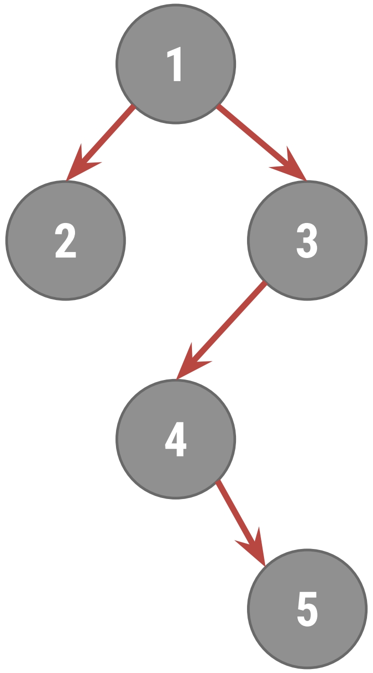
Postorder traversal of a given tree as

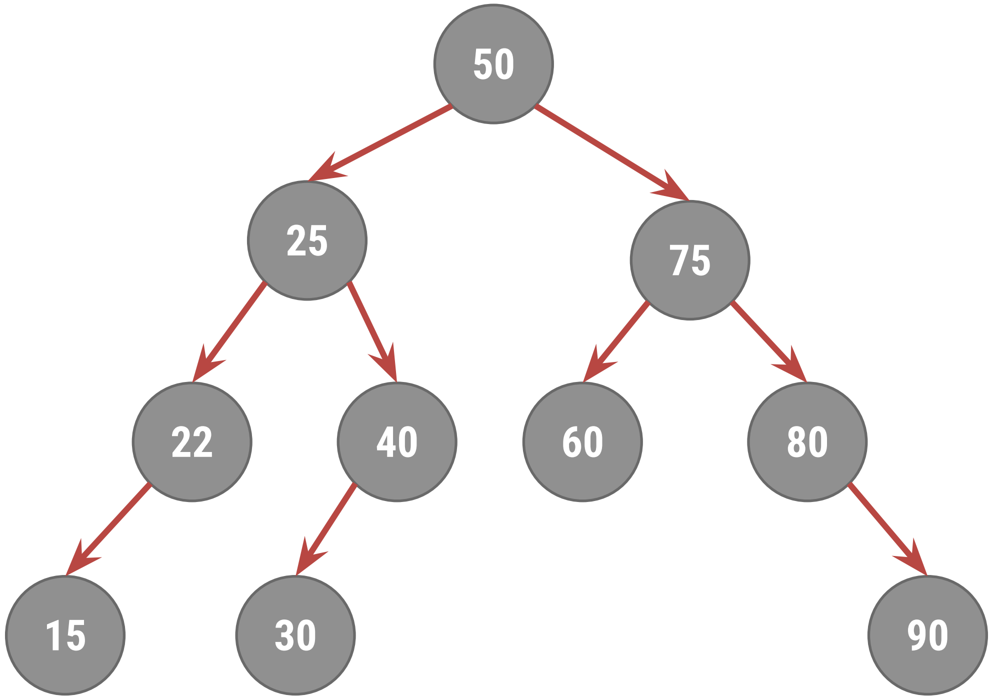
C B F E G D A

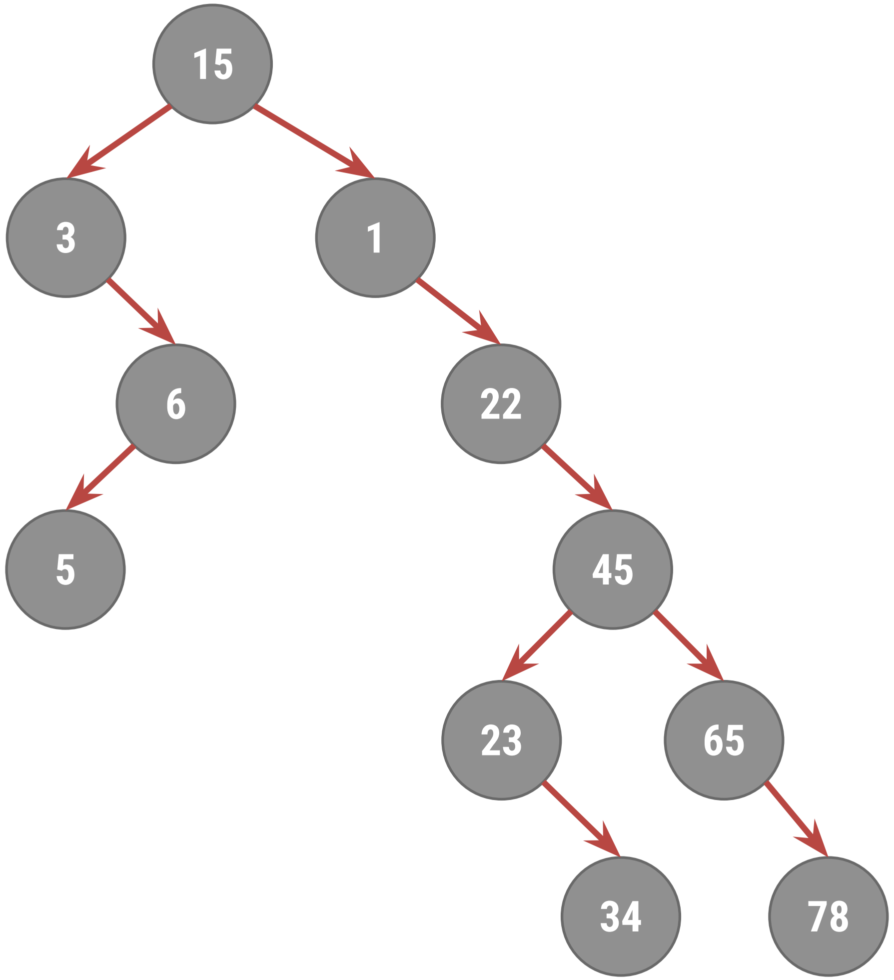
Converse Traversal

- 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



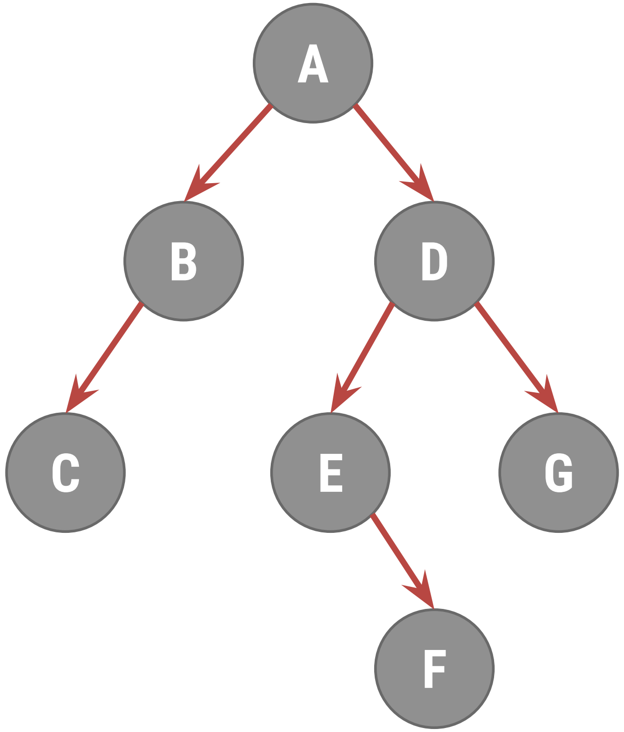


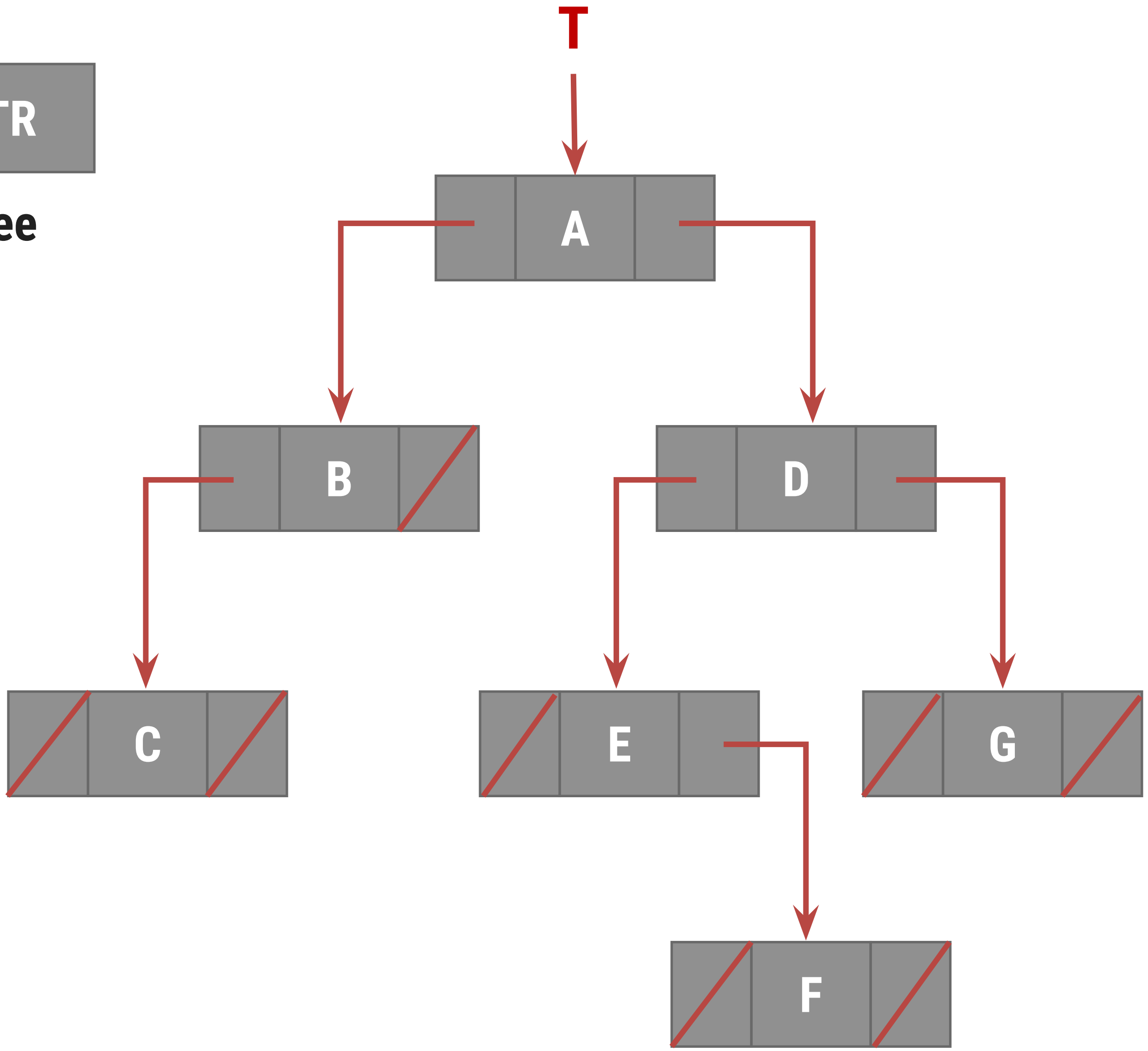


Linked Representation of Binary Tree

LPTR	DATA	RPTR

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
```

Construct Binary Tree from Traversal

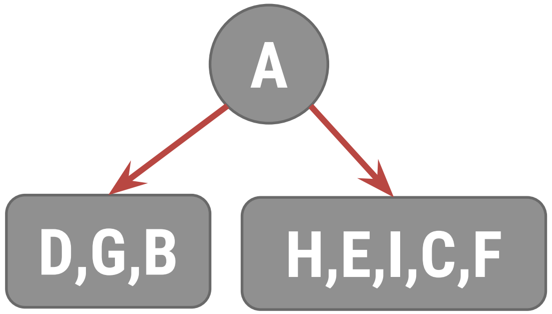
Construct a Binary tree from the given **Inorder** and **Postorder** traversals

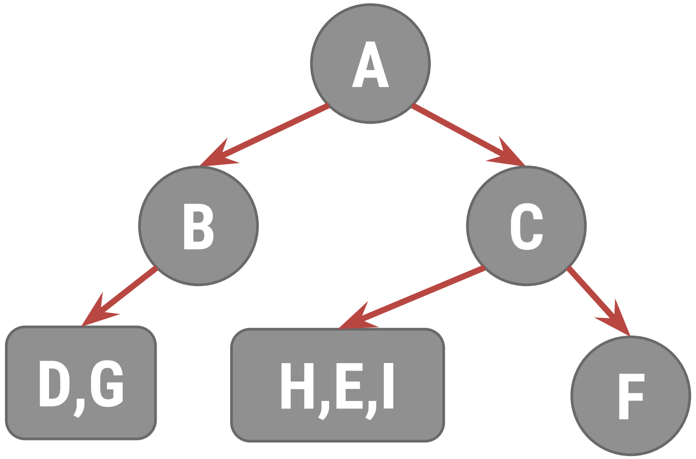
Inorder : D G B **A** H E I C F
Postorder : G D B H I E F C **A**

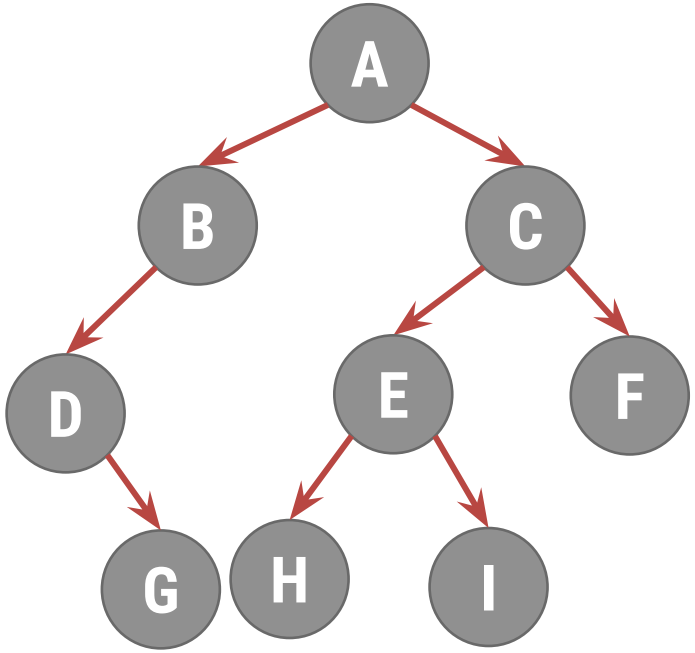
- Step 1: Find the root node
  - Preoder Traversal – first node is root node
  - Postoder Traversal last node is root node
- Step 2: Find Left & Right Sub Tree
  - Inorder traversal gives Left and right sub tree

Postorder : G D B H I E F C **A**

Inorder : D G B **A** H E I C F

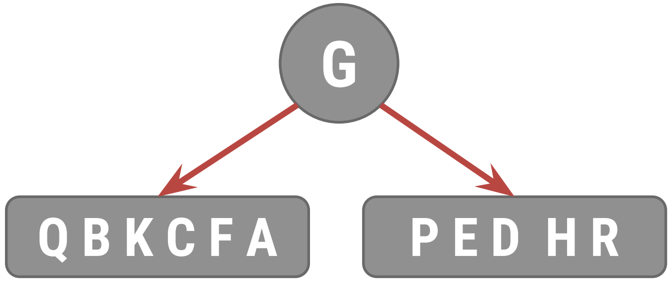


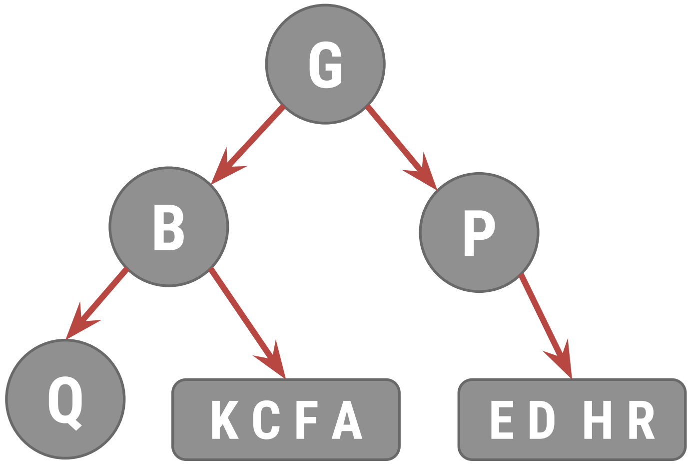


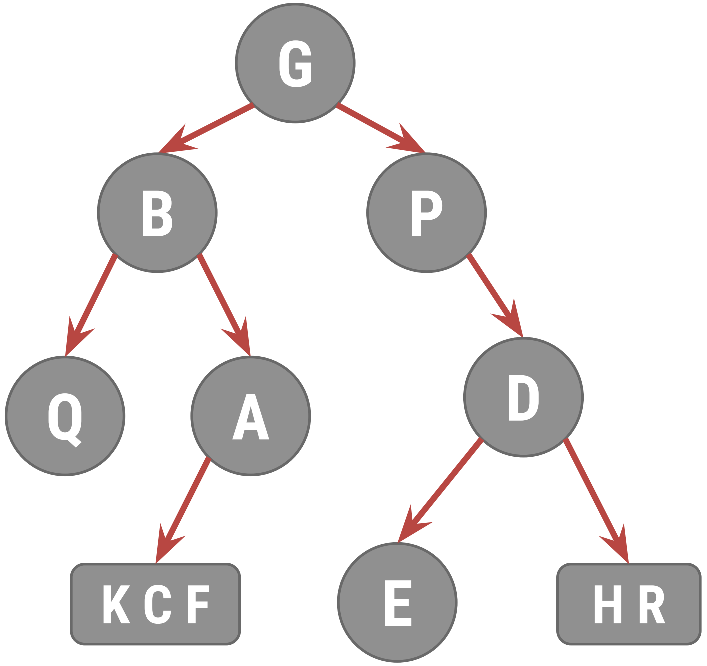


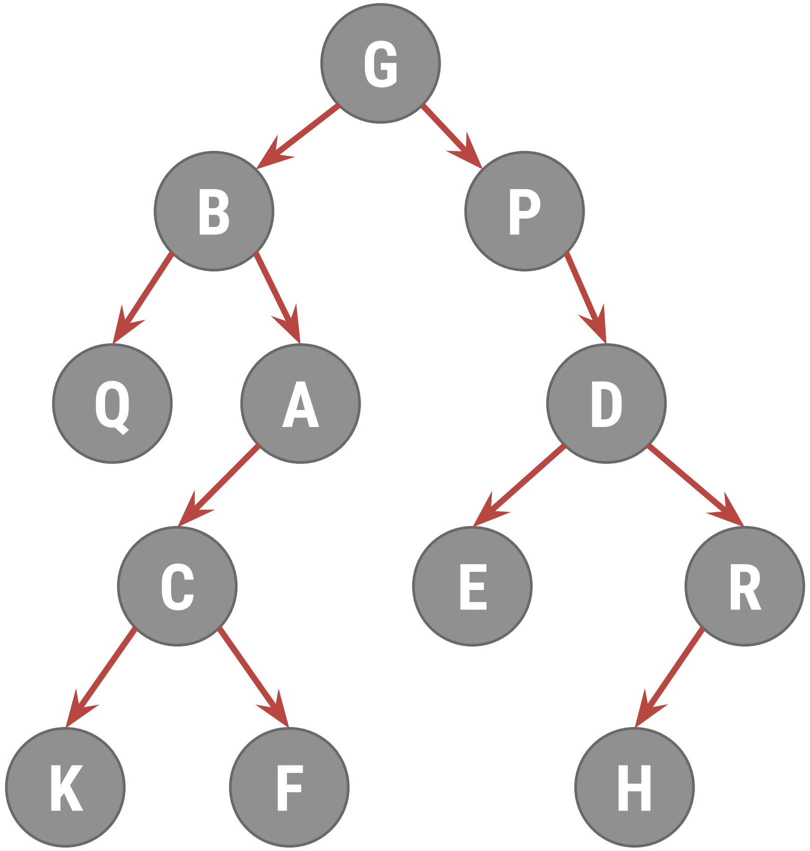
Construct Binary Tree from Traversal

Preorder : G B Q A C K F P D E R H | **Inorder :** Q B K C F A G P E D H R



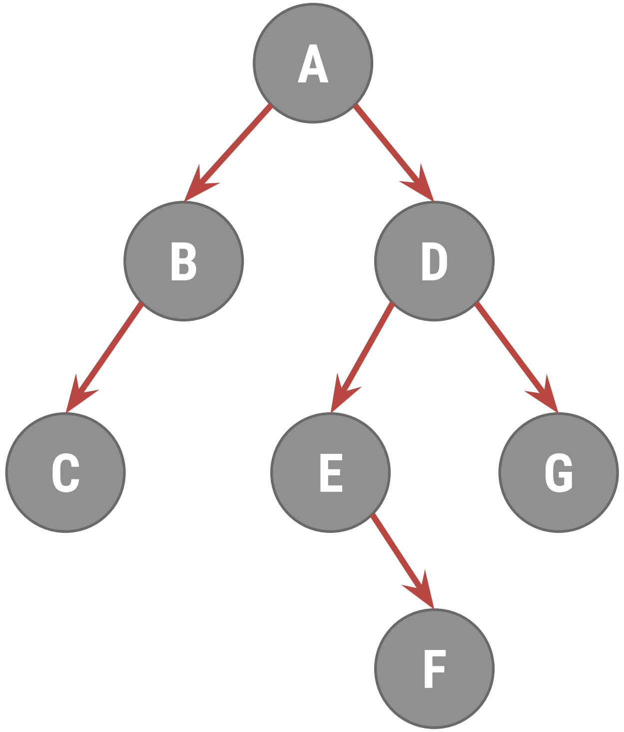


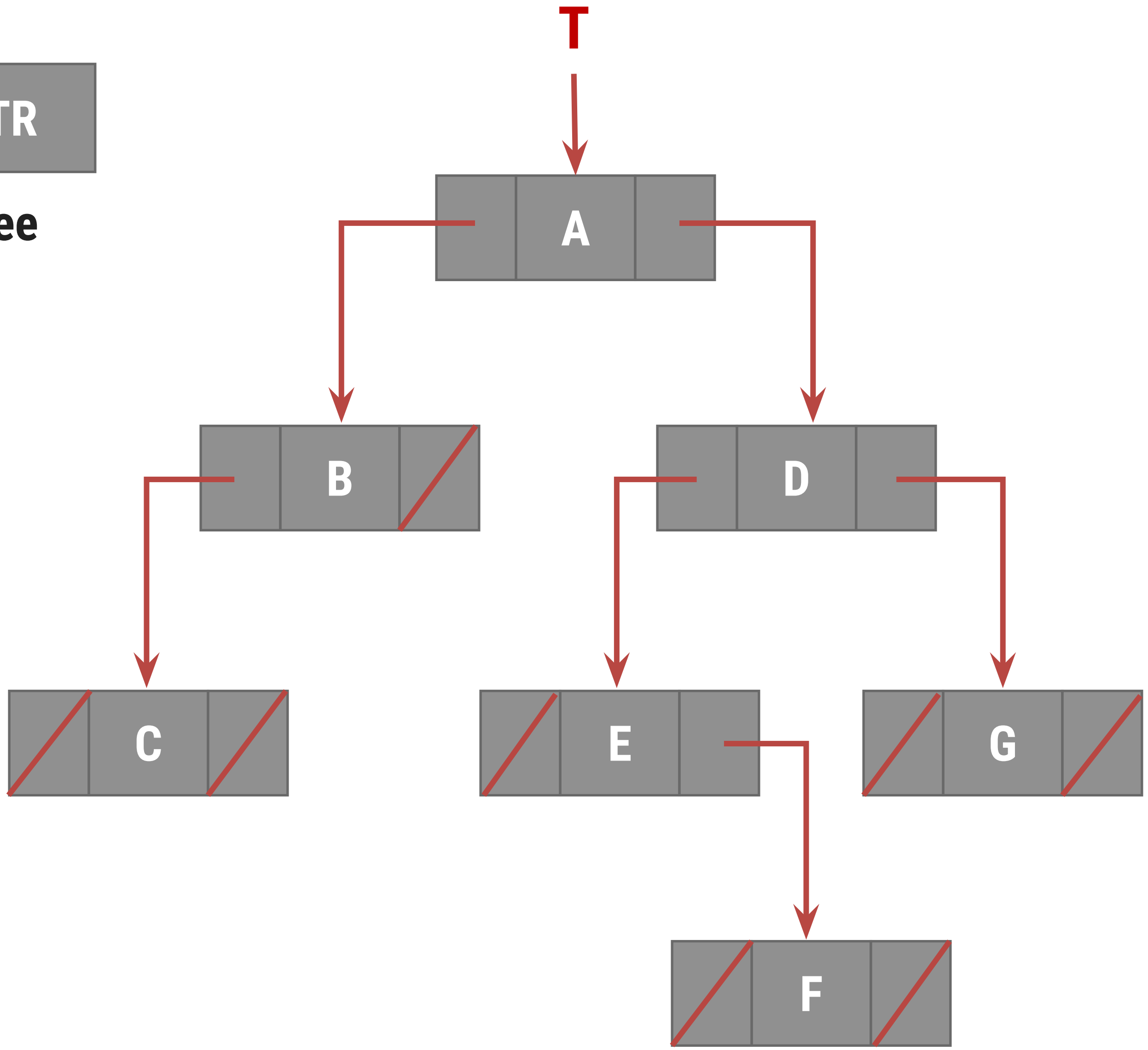




Linked Representation of Binary Tree

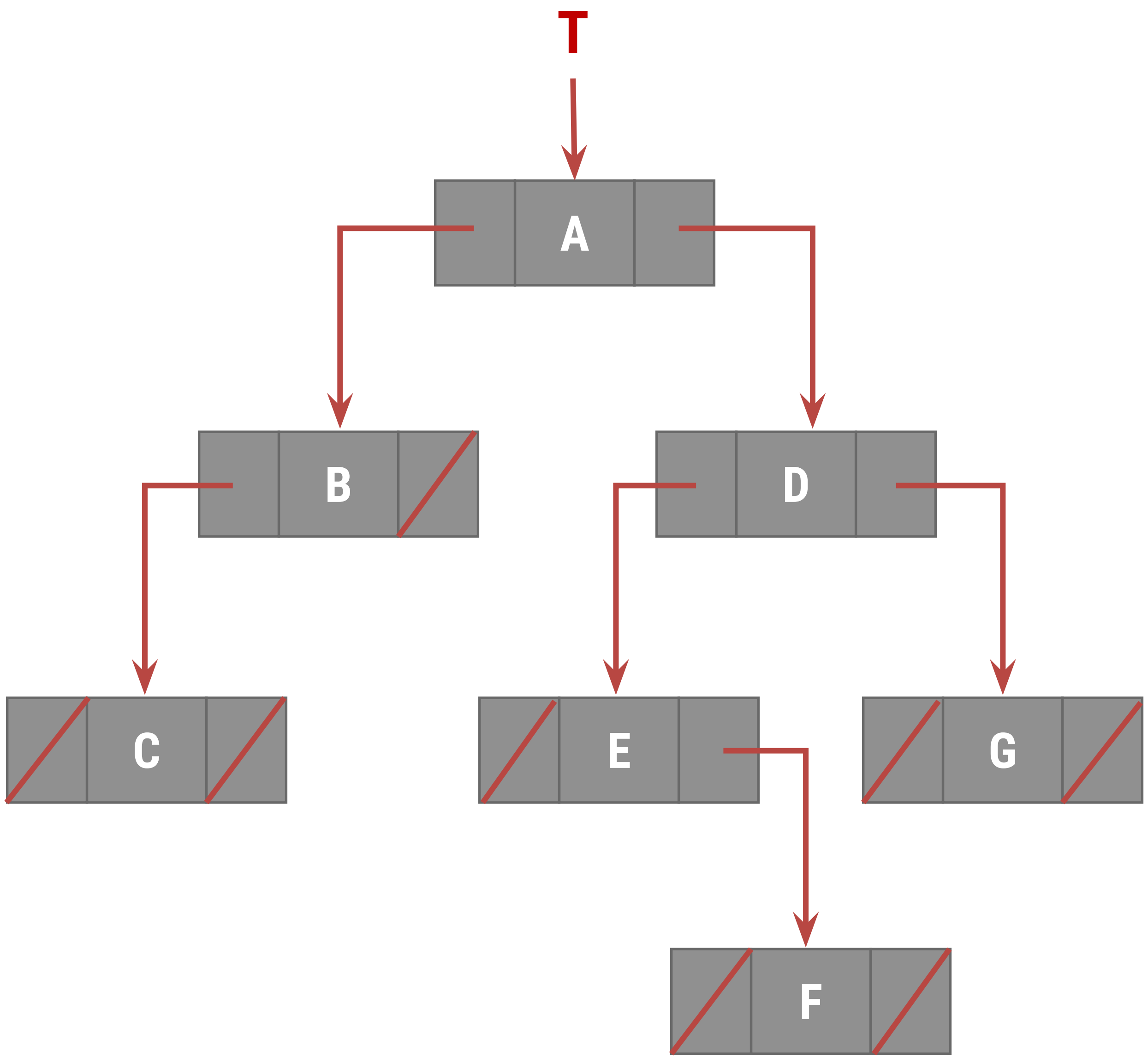
Typical node of Binary Tree





Threaded Binary Tree

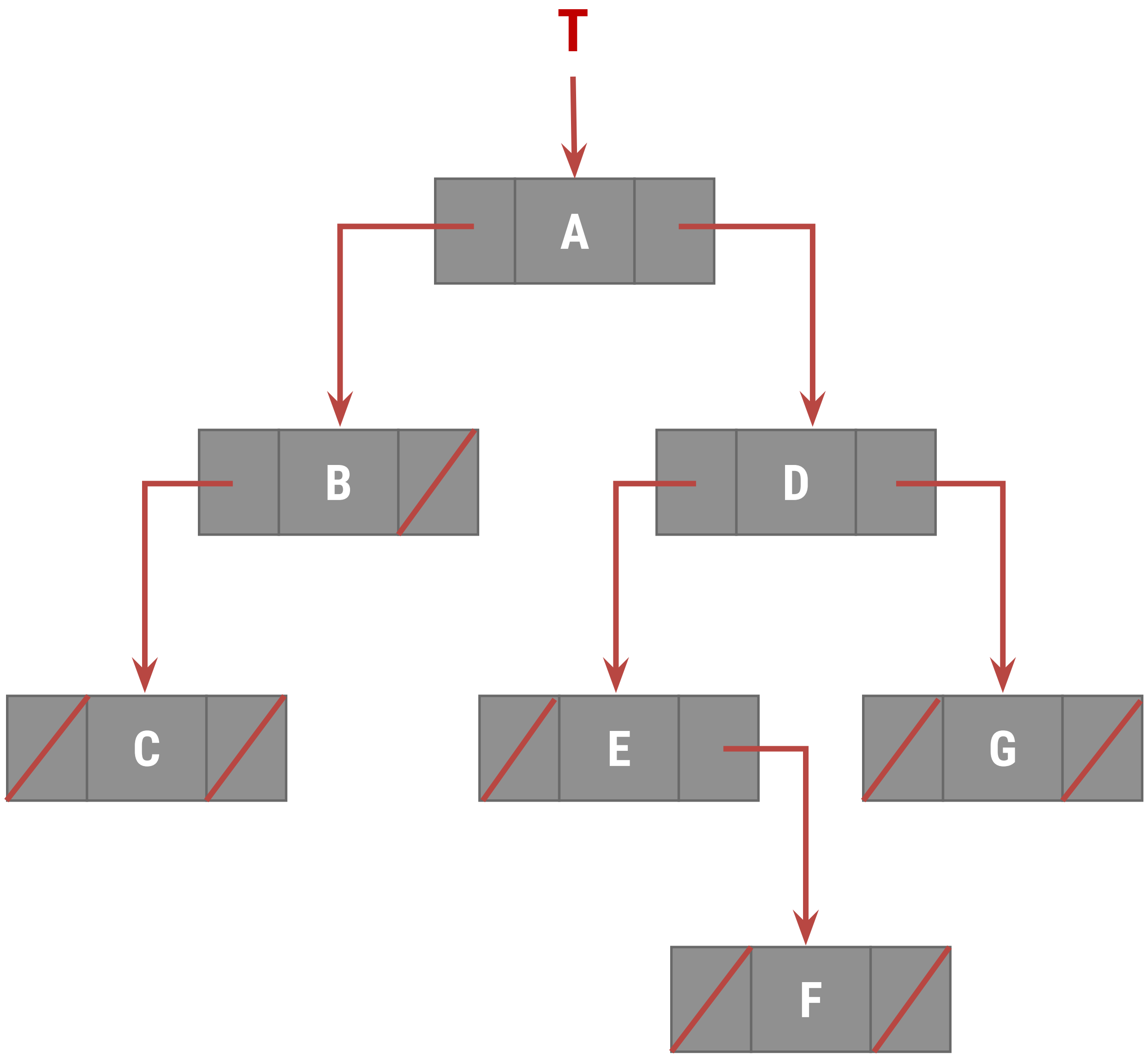
- The **wasted NULL** links in the binary tree storage representation can be **replaced by threads**
- A binary **tree** is **threaded according** to particular **traversal order**. e.g.: Threads for the inorder traversals of tree are pointers to its higher nodes, for this traversal order



- **In-Order** - **C B A E F D G**
- **Pre-Order** - **A B C D E F G**
- **Post-Order** - **C B F E G D A**

Threaded Binary Tree

- **In-Threaded Binary Tree**
  - **If left link of node P is null**, then this link is **replaced by** the **address of its predecessor**
  - **If right link of node P is null**, then this link is **replaced by** the **address of its successor**
- Because the left or right **link** of **a node** can denote **either structural link** or **a thread**, we must somehow be able to distinguish them



- **In-Order** - C B A E F D G
- **Pre-Order** - A B C D E F G
- **Post-Order** - C B F E G D A

Threaded Binary Tree

- **Method 1:-** Represent **thread a Negative address**
- **Method 2:-** To have a **separate Boolean flag** for each of left and right pointers, node structure for this is given below

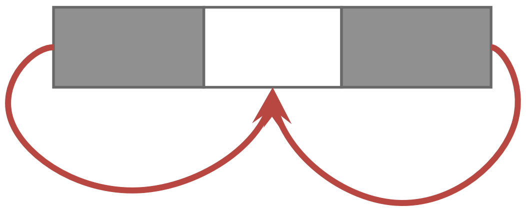
LPTR	LTHREAD	DATA	RTHREAD	RPTR

Typical node of Threaded Binary Tree

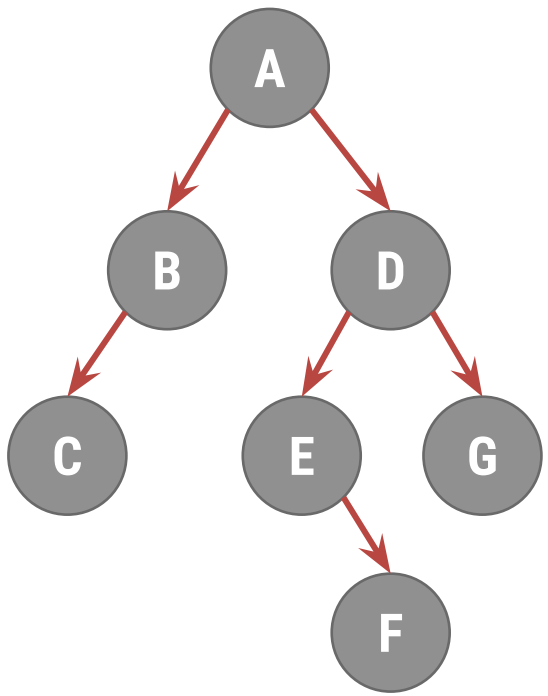
- **LTHREAD = true =** Denotes leaf thread link
- **LTHREAD = false =** Denotes leaf structural link
- **RTHREAD = true =** Denotes right threaded link
- **RTHREAD = false =** Denotes right structural link

Head node is simply another node which serves as the predecessor and successor of first and last tree nodes.
Tree is attached to the left branch of the head node.

HEAD

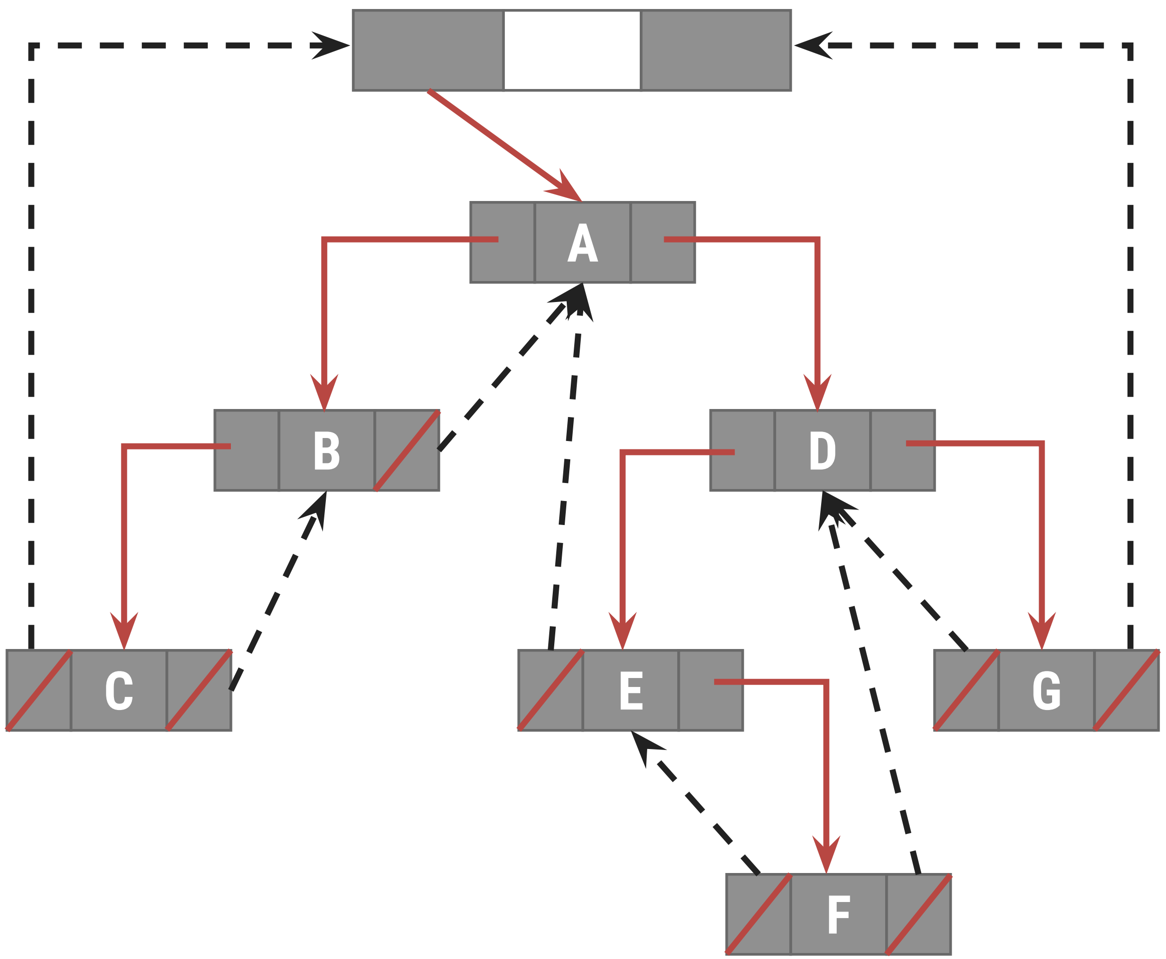


Threaded Binary Tree



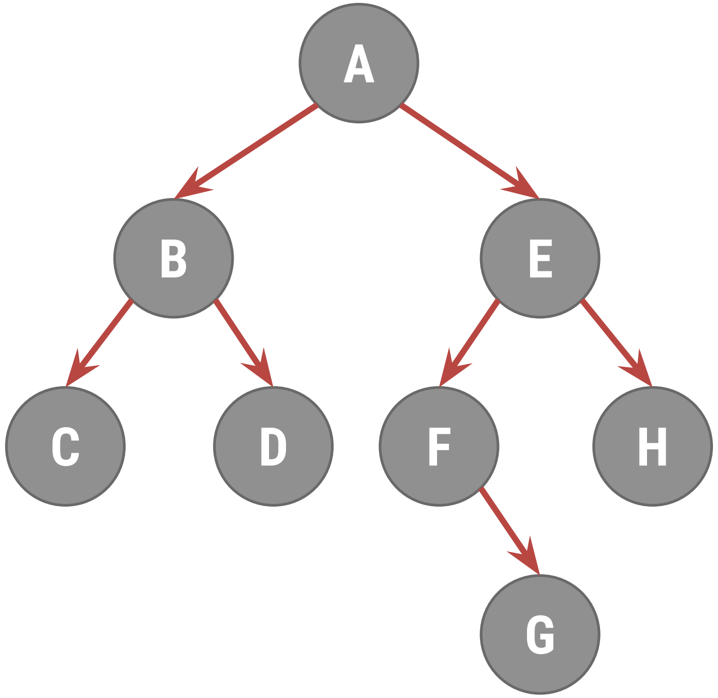
Inorder Traversal

C B A E F D G



Fully In-Threaded Binary Tree

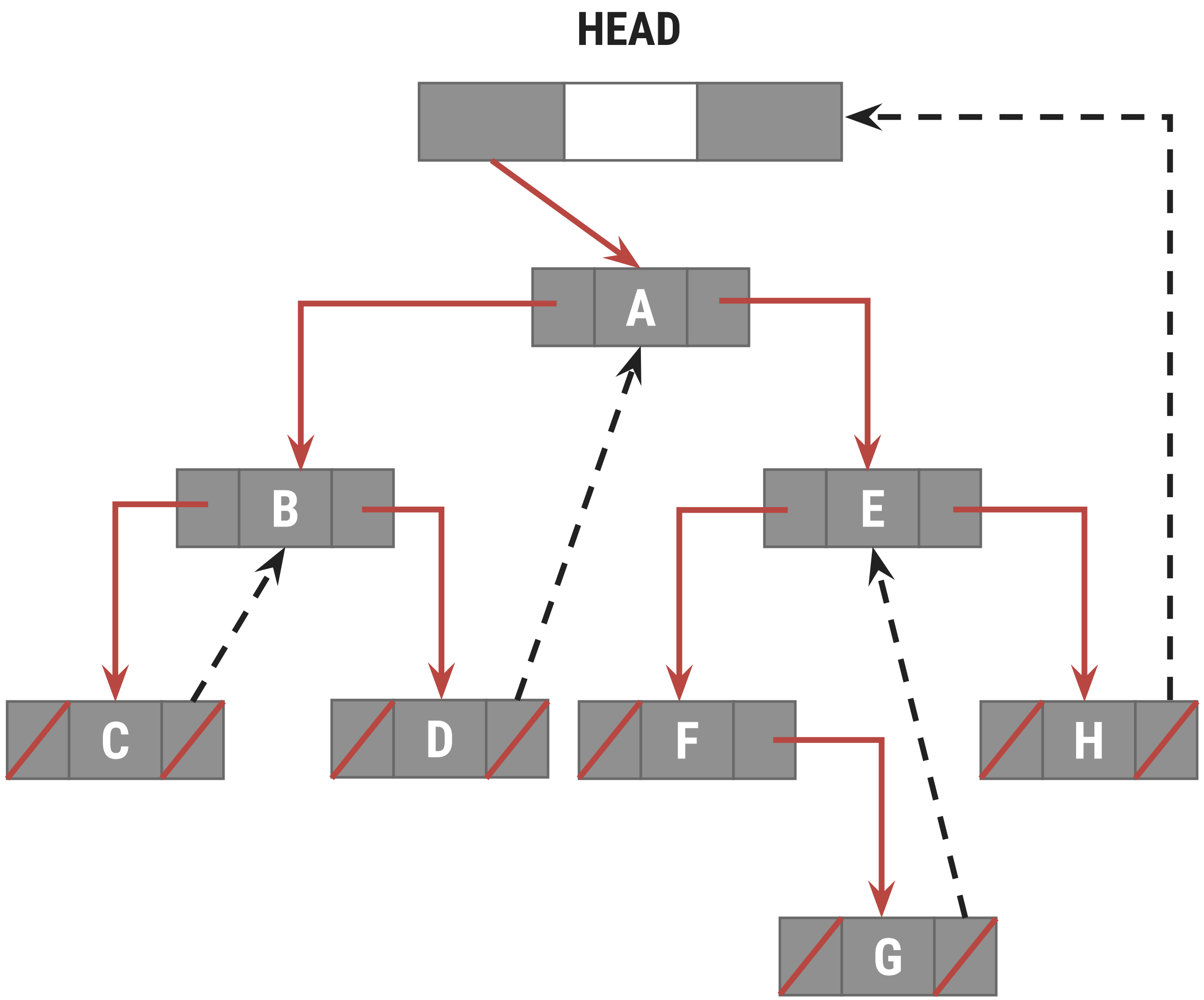
Threaded Binary Tree



Inorder Traversal

C B D A F G E H

Construct Right In-Threaded Binary Tree of given Tree



Advantages of Threaded Binary Tree

- **Inorder traversal is faster** than unthreaded version as stack is not required.
- **Effectively determines** the **predecessor and successor** for inorder traversal, for unthreaded tree this task is more difficult.
- **A stack is required** to provide upward pointing information **in binary tree** which **threading provides without stack**.
- It is possible to **generate successor or predecessor** of any node **without** having over head of **stack** with the help of threading.

Disadvantages of Threaded Binary Tree

- Threaded trees are **unable to share common sub trees.**
- If **Negative addressing is not permitted** in programming language, **two additional fields are required.**
- **Insertion** into and **deletion** from threaded binary tree are **more time consuming** because both thread and structural link must be maintained.

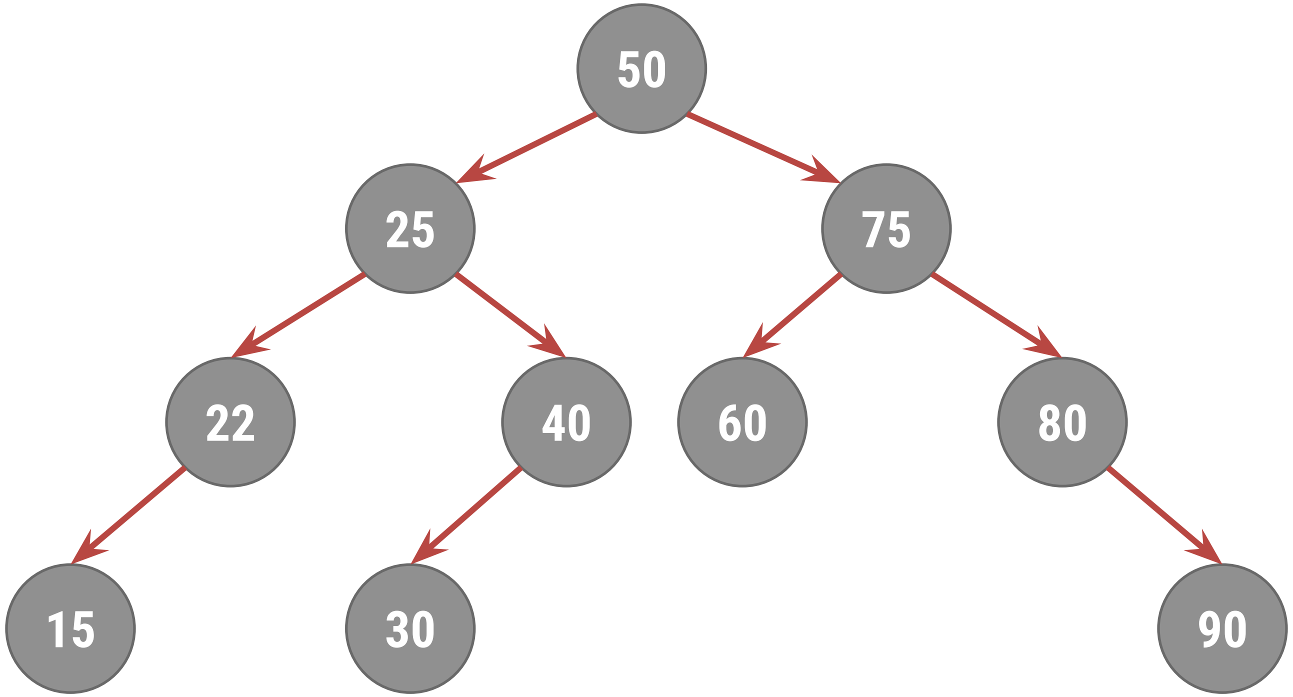
Binary Search Tree (BST)

- 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

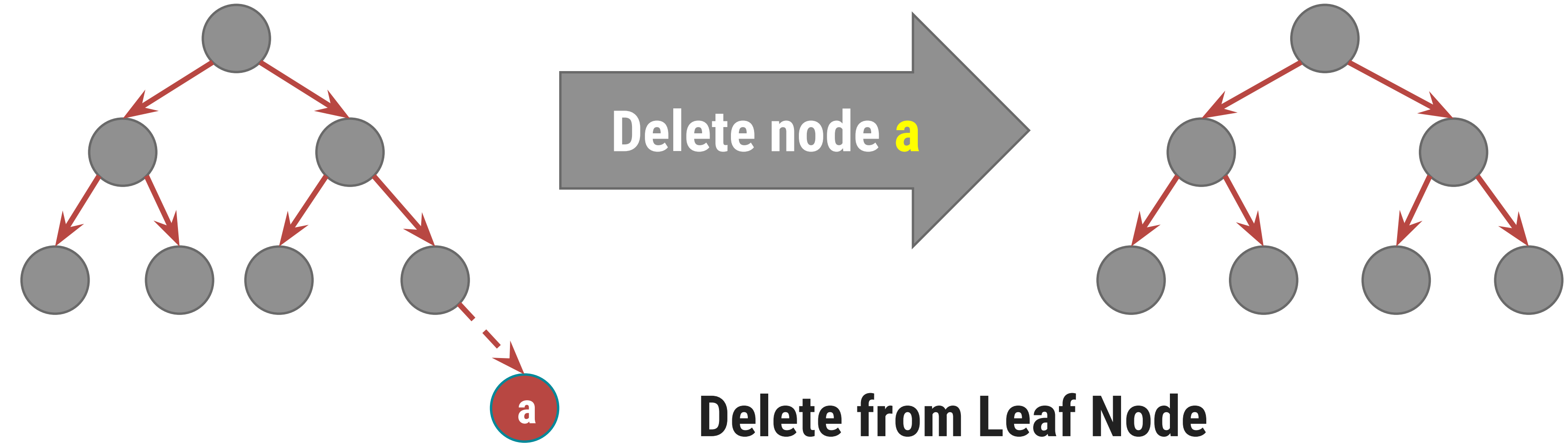


Construct binary search tree for the following data
10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5

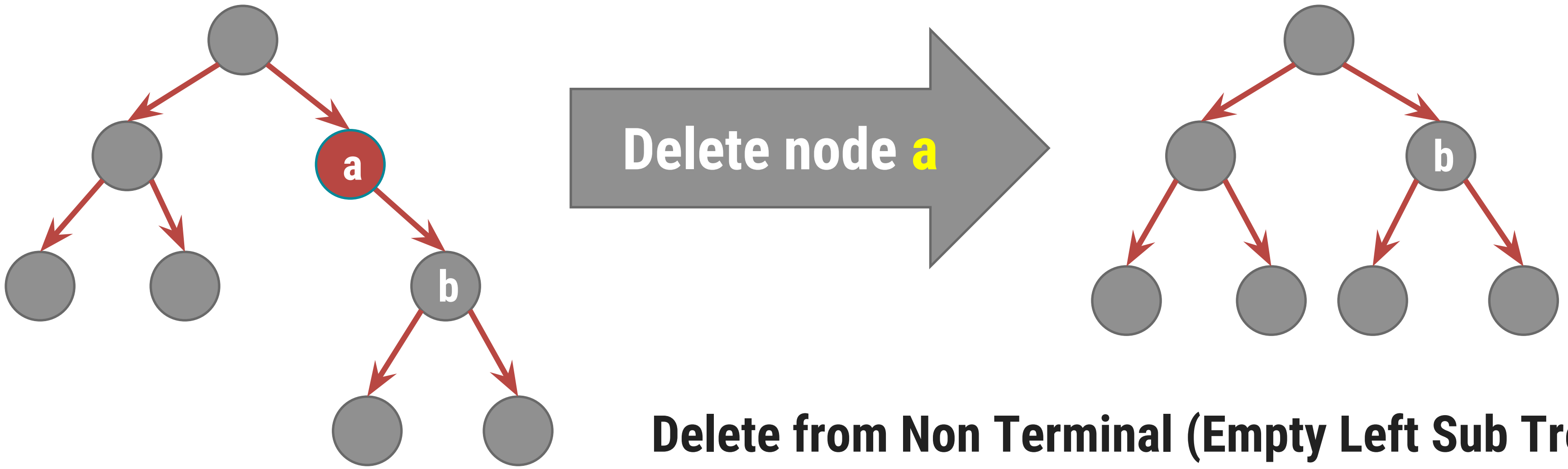
Search a node in Binary Search Tree

- 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.

Delete node from Binary Search Tree



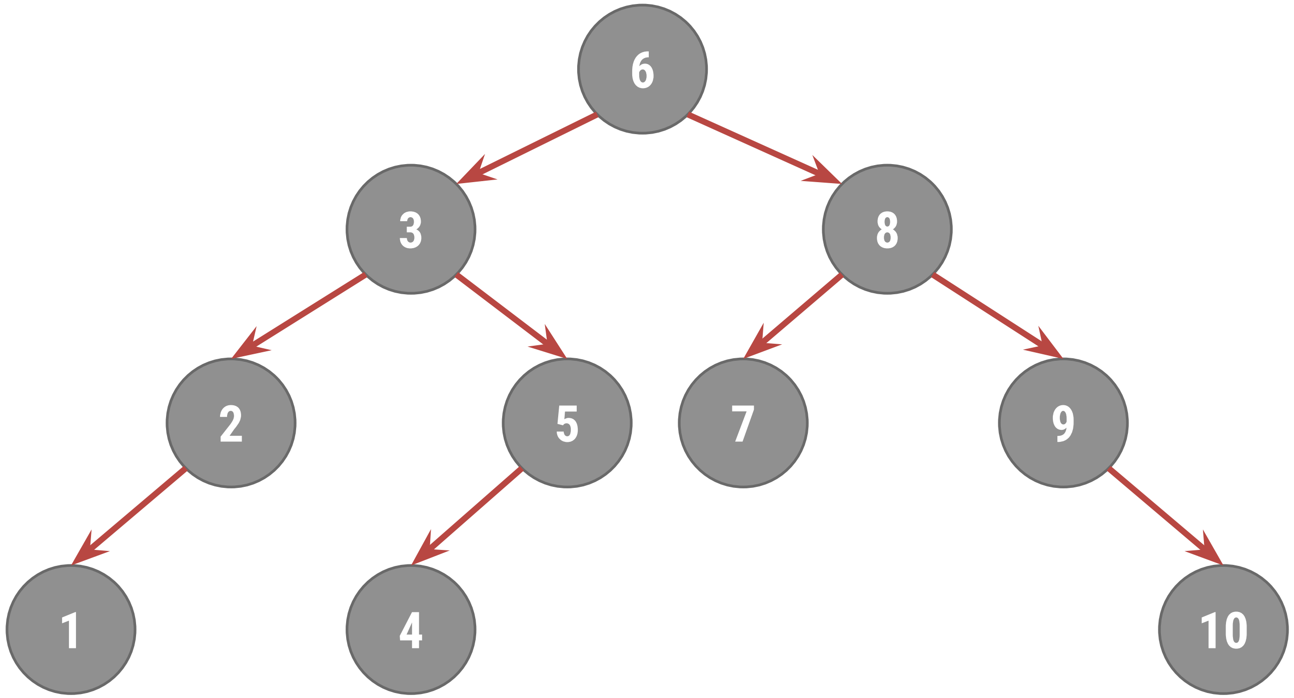
Delete from Leaf Node



Delete from Non Terminal (Empty Left Sub Tree)

Construct Binary Search Tree (BST)

Construct binary search tree for the following data
6 , 3 , 8 , 2 , 5 , 7 , 9 , 10 , 1 , 4

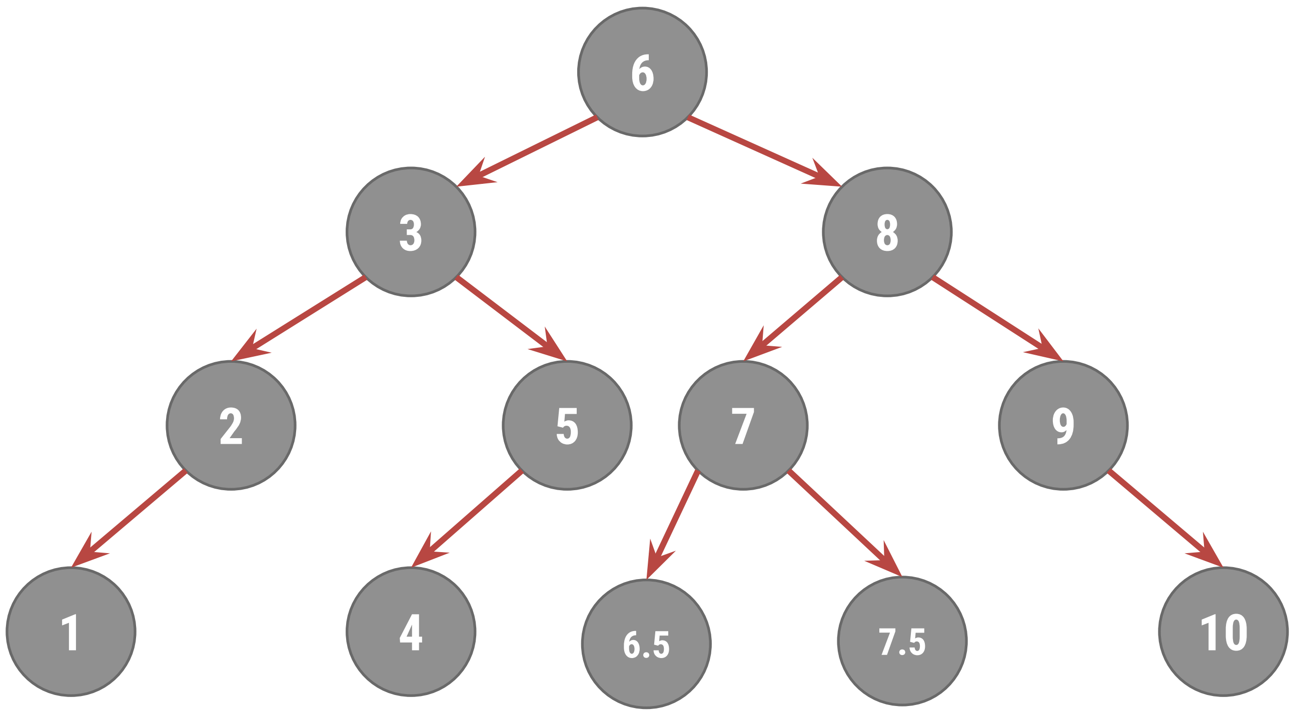


In-Order Traversal : 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10

After Delete a Node (6) : 1 , 2 , 3 , 4 , 5 , 7 , 8 , 9 , 10

Construct Binary Search Tree (BST)

Construct binary search tree for the following data
6 , 3 , 8 , 2 , 5 , 7 , 9 , 10 , 1 , 4 , 6.5 , 7.5



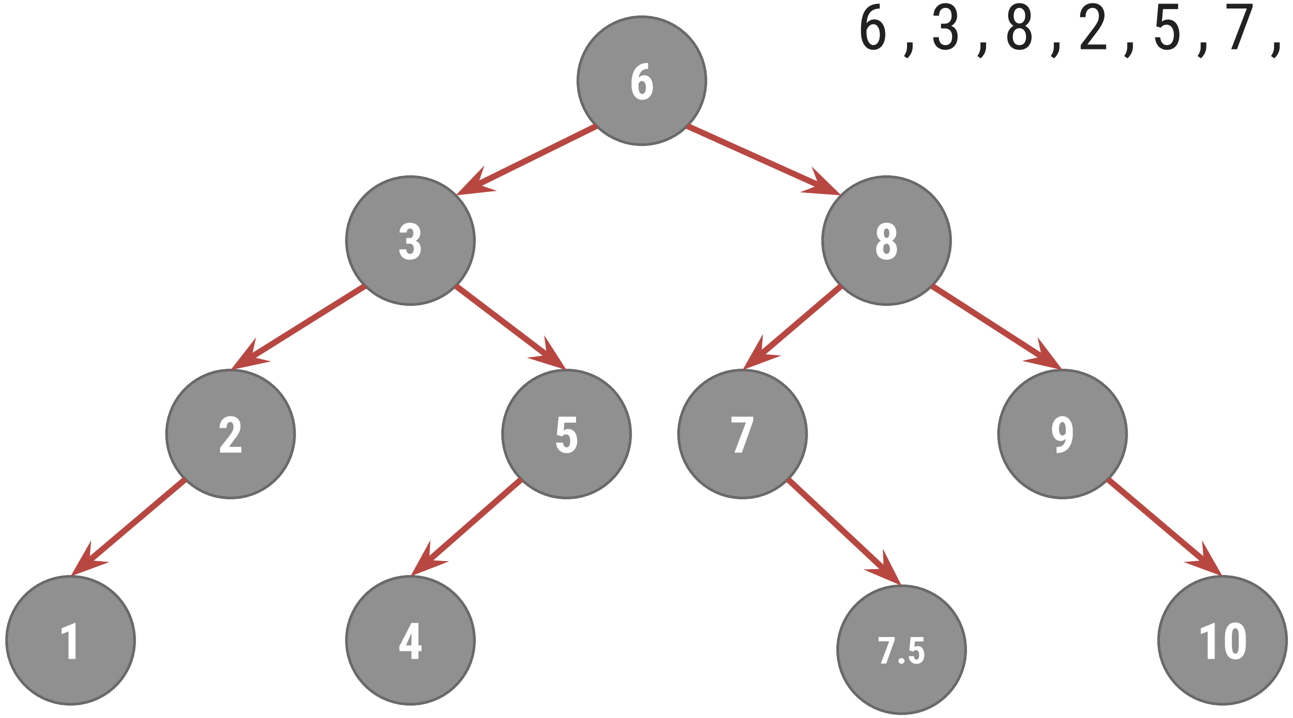
In-Order Traversal : 1 , 2 , 3 , 4 , 5 , 6 , 6.5 , 7 , 7.5 , 8 , 9 , 10

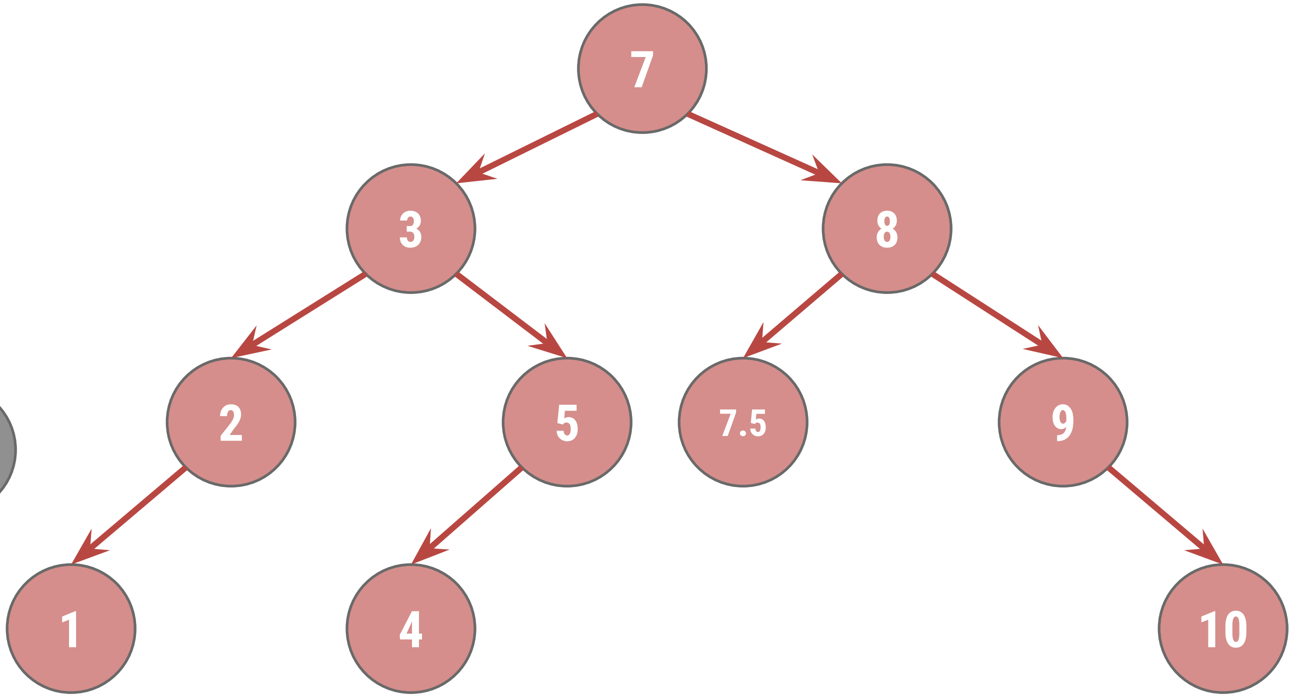
After Delete a Node (6) : 1 , 2 , 3 , 4 , 5 , 6.5 , 7 , 7.5 , 8 , 9 , 10

Construct Binary Search Tree (BST)

Construct binary search tree for the following data

6 , 3 , 8 , 2 , 5 , 7 , 9 , 10 , 1 , 4 , 7.5

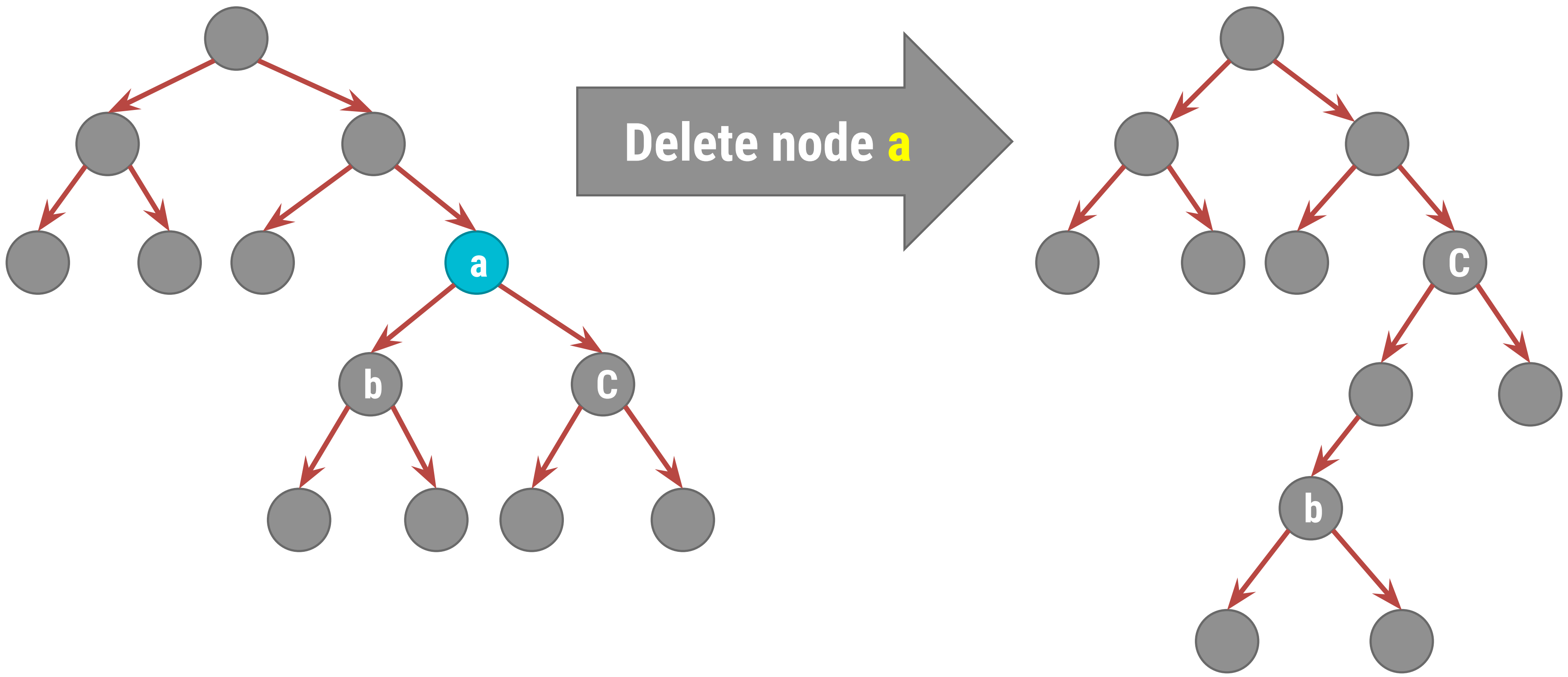




In-Order Traversal : 1 , 2 , 3 , 4 , 5 , 6 , 7 , 7.5 , 8 , 9 , 10

After Delete a Node (6) : 1 , 2 , 3 , 4 , 5 , 7 , 7.5 , 8 , 9 , 10

Delete node from BST



Delete from Non Terminal (Neither Sub Tree is Empty)

Thank You