CHAPTER 4

Trees

IV.1. Introduction

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non-linear if its elements form a hierarchical classification where data items appear at various levels. Trees and Graphs are widely used non-linear data structures.

In computer science, a tree is a very general and powerful data structure that resembles a real tree. It consists of an ordered set of linked nodes in a connected graph, in which each node has at most one parent node, and zero or more child nodes with a specific order.

In this chapter, in particular, we will explain one of the non-linear data structures known as a tree, which is easy to maintain on the computer.

IV.2. Non-Linear Data Structures

Non-linear data structures are those where data items are not arranged sequentially, unlike linear data structures. In these data structures, elements are stored in a hierarchical or a network-based structure that does not follow a sequential order. These data structures allow efficient searching, insertion, and deletion of elements from the structure. Examples of Non-Linear Data Structures:

- Trees
- Graphs, etc.

Properties of Non-Linear Data Structures

The Non-linear data structures have the following properties.

- Non-linear data structures do not follow a sequential order.
- Elements are stored in a hierarchical or a network-based structure.
- These data structures allow efficient searching, insertion, and deletion of elements.
- Non-linear data structures are used to solve complex problems where data cannot be arranged in a linear manner.

IV.3. Tree Data Structure

The tree is a Non-Linear data structure in which data are stored in a hierarchical structure and consists of nodes connected by edges. It is also defined as a hierarchical collection of nodes. The collection can be empty. Each node has a parent node and zero or more child nodes. The node at the top of the hierarchy is called the root node. Trees are widely used in computer science and are

an essential part of many algorithms and data structures. Trees represent a special case of more general structures known as graphs. Figure 4.1 shows a tree and a non-tree.

A **rooted tree** is a type of simple tree that contains a special node called the **root**. All edges in a rooted tree branch away from this root node, and any node can be selected as the root. An example of a rooted tree is shown below, where node A is the root:

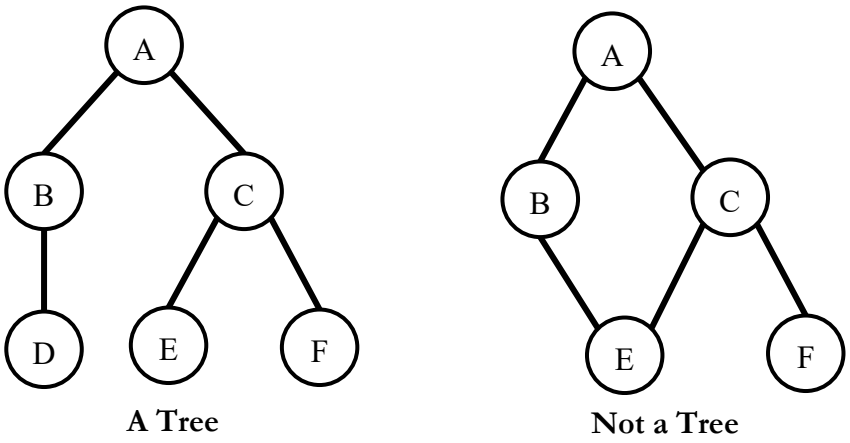
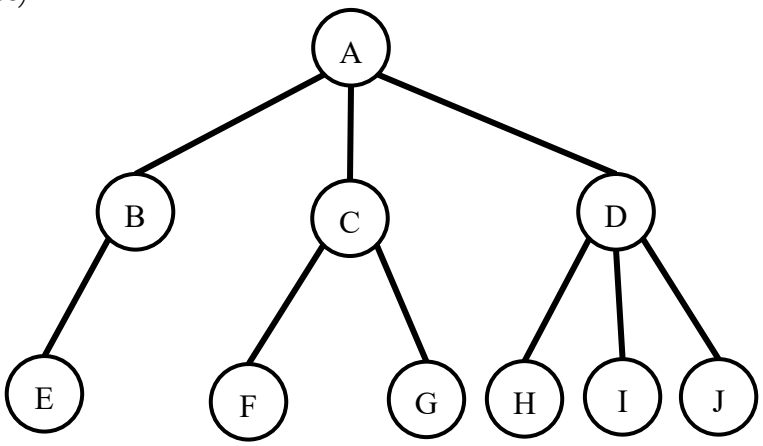


Figure 4.1 : A Tree and not a Tree.

IV.4. Basic Tree Terminology

Below lists some general tree terminology (many of these terms are derived from terms used for family relationships):



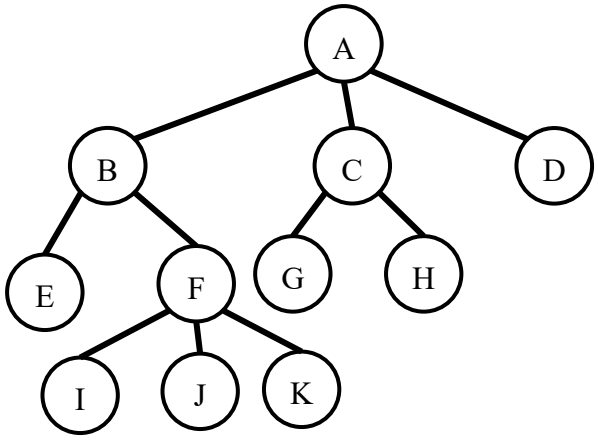
- **Node:** Each element of a tree is called as node. In the previous figure there are **10** nodes.
- **Root:** The root of a tree is the "top-most" vertex in the tree, from which all other nodes are directed away from. In the tree above, node **A** is the root of the tree.
- **Parent:** Parent of a node is the immediate predecessor of a node. In the tree above, **C** is the parent of **F** and **G**.
- **Child:** Each immediate successor of a node is known as child. In the tree above, **B**, **C**, **D** are children of **A**.
- **Sibling:** Two nodes are siblings if they share the same parent. In the tree above, **B**, **C**, and **D** are siblings, since they share the parent **A**.
- **Path:** Path is the sequence of consecutive edges from the source node to the destination node path between **A** and **I** is **(A,D)**,**(D,I)**.
- **Leaf** : is a node that has no child node. In the tree above, **E**, **F**, **G**, **H**, **I,** and **J** are leaves.

- **Descendant:** The descendants of a given node consist of any node that exists along a path from the given node to a leaf node (including the leaf node). In the tree above, nodes **B** and **E** are both descendants of node **A**.
- **Ancestor:** The ancestors of a given node consist of any node that exists along the path from the given node to the root node (including the root node itself). In the tree above, nodes **B** and **A** are both ancestors of node **E**.
- **Degree of a Node:** The number of sub-trees of a node in a given tree. In the tree above,
  - The degree of node **A** is **3**
  - The degree of node **B** is **1**
  - The degree of node **C** is **2**
  - The degree of node **F** is **0**
- **Degree of Tree:** The maximum degree of nodes in a given tree. In the tree above, the maximum degree of nodes **A** and **D** is **3**. So, the degree of Tree is **3**.
- **Terminal Node (Leaf):** A terminal node (or leaf node) is a node that has no children or has a degree zero. In the tree above, nodes **E** to **J** are examples of terminal node (Leaf).
- **Internal Node:** An internal node is a node with children. In the tree above, nodes **A** to **D** are examples of internal nodes.
- **Level:** The entire tree structured is leveled in such a way that the root is always at the level **0**, then its immediate children are at level 1, and their immediate children are at level 2 and so on up
- **Depth:** The depth of a node represents how far it is from the top of the tree. The root has a depth of **0**. If a node is one edge away from the root, it has a depth of **1**. If a node is two edges away from the root, it has a depth of **2**. We can define the depth of a node recursively:
  - **Depth(root) = 0**
  - **Depth(node) = Depth(parent) + 1**
- **Height:** The height of a node represents how far it is from the bottom of the tree. Leaf nodes have a height of **0**. If a node has more than one child, its height is equal to one plus the largest height of any of its children. We can define the height of a node recursively:
  - **Height(leaf) = 0**
  - **Height(node) = max(Height(children)) + 1**

  Remark: In a tree, the largest height of any node and the largest depth of any node should be the same value.

Example :

Consider the following tree.



Which node is the **root**?

Which nodes are **leaf nodes**?

Which nodes are **internal nodes**?

What is the **maximum depth of the tree**?

What is the **height** of node **B**?

In this tree, node **A** is the **root**.

Nodes **D, E, G, H, I, J,** and **K** are **leaves** because they have no children.

Nodes **A**, **B**, **C**, and **F** are **internal nodes** because they have children.

The **maximum depth of the tree** is **3**.

The **height** of **B** is equal to max(height(E), height(F)) + **1**. Since **F** is the child with the larger height (height(E) = **0**, height(F) = **1**), the **height** of **B** is **1 + 1 = 2**.

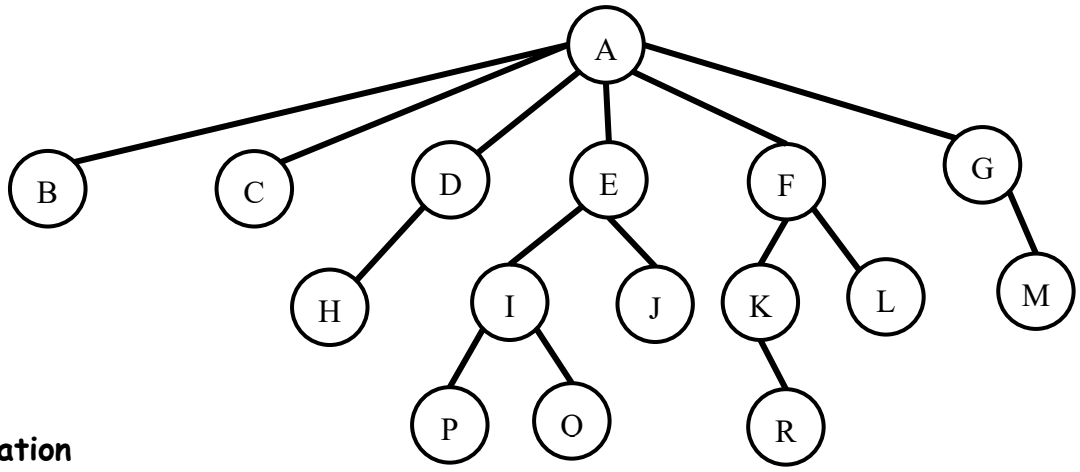
IV.5. Types of the Tree Data Structure

Based on the number of children for each node in the tree, it is classified into two to types.

1) General Tree
2) Binary Tree

IV.5.1. General Tree

In a tree, node can have any number of children. Then it is called as general Tree.



Declaration

```
struct GenericNode {
    int  data;
    GenericNode* children[N];  //Maximum number of children
};
typedef GenericNode *GTREE;
```

Now, we might have a variable of type **GTREE** called GTree: **GTREE** `GTree;`

IV.5.2. Binary Tree

In general, tree nodes can have any number of children. A **binary tree** is a tree-type **non-linear data structure** with a maximum of two children for each parent (each node can have at most two children). A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the left subtree and the right subtree. Furthermore, every node in a **binary tree** has a left and right reference along with the data element.

Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

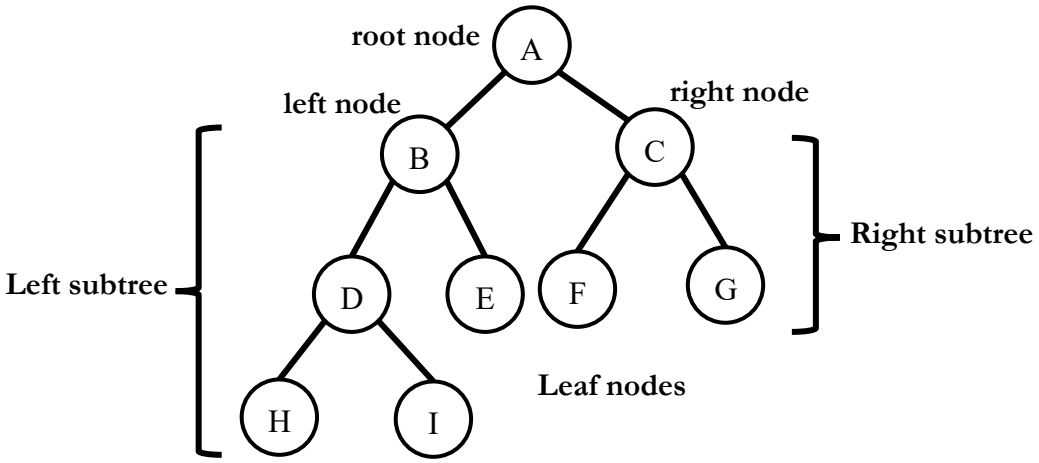


Figure 4.2 : Binary Tree.

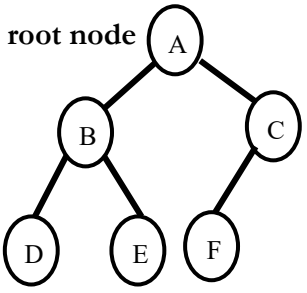
IV.5.2.1. Types of Binary Trees

There are several types of binary trees:

Rooted Binary Tree

A rooted binary tree is a binary tree that satisfies the following two properties:

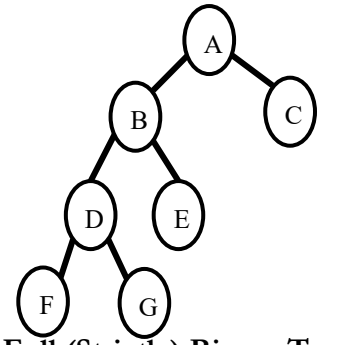
- It has a root node.
- Each node has at most 2 children.



Root Binary Tree

Full / Strictly Binary Tree

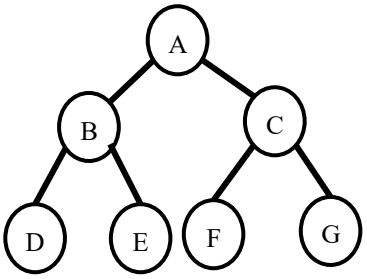
Full binary tree is also called as **Strictly binary tree**. It is a special kind of binary tree that has **either zero children or two children**. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node. Here is the structure of a full (strictly) binary tree.



Full (Strictly) Binary Tree

Perfect Binary Tree

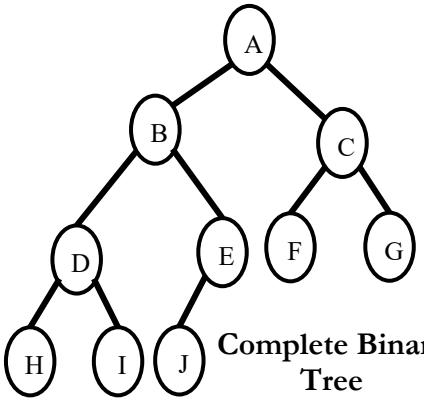
A binary tree is said to be "perfect" if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height "H" has $2^{H+1} - 1$ node. Here is the structure of a perfect binary tree:



Perfect Binary Tree

Complete Binary Tree

An almost complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side. Here is the structure of an almost complete binary tree:



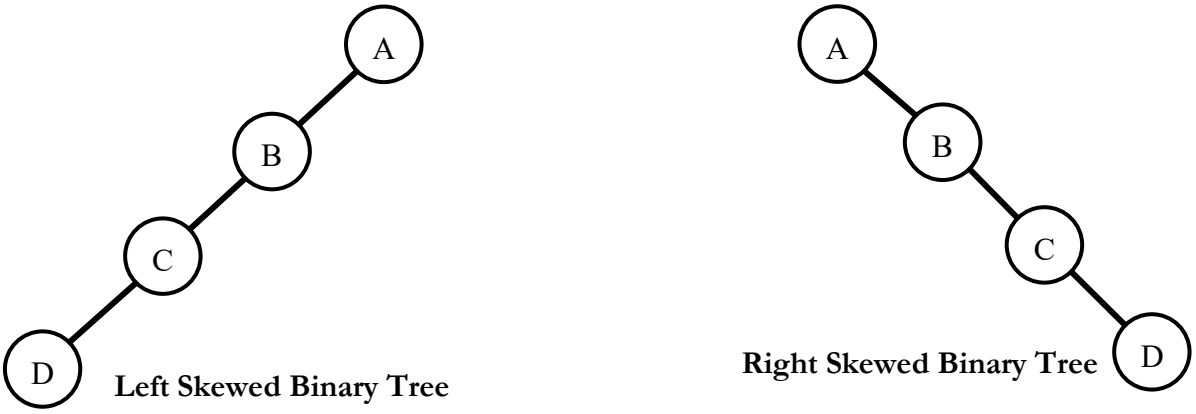
Complete Binary Tree

Skewed Binary Tree

A skewed binary tree is a binary tree that satisfies the following two properties :

- All the nodes except one node has one and only one child.
- The remaining node has no child.

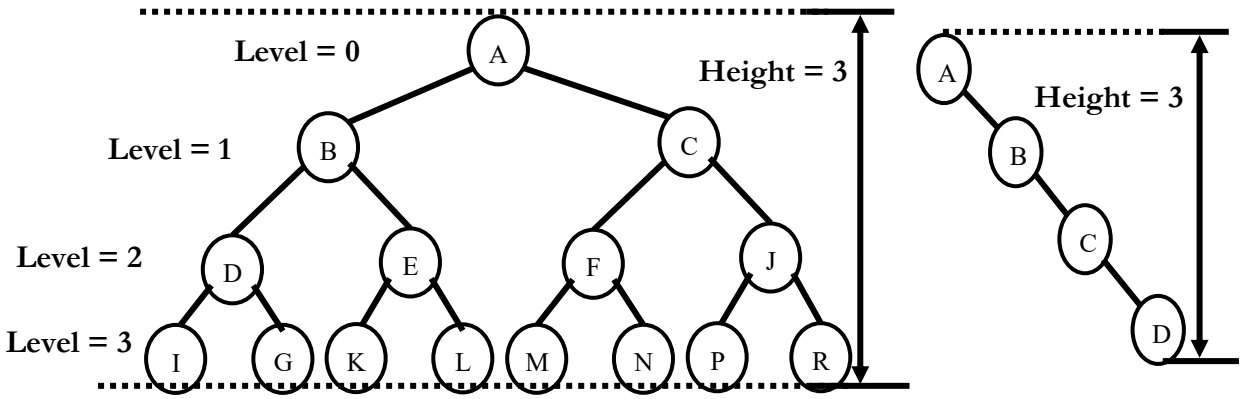
Also, a skewed binary tree is a binary tree of $\boldsymbol{n}$ nodes such that its depth is $(\boldsymbol{n-1})$.



IV.5.2.2. Properties of Binary trees

Some of the important properties of a binary tree are as follows:

- A tree consisting of only a root node has a height of $\boldsymbol{1}$.
- If $\boldsymbol{H}$ is a height of a binary tree, then
  - Maximum number of leaves $= \boldsymbol{2^H}$
  - Maximum number of nodes $= \boldsymbol{2^{H+1} - 1}$
  - Minimum number of nodes $= \boldsymbol{H + 1}$
- If a binary tree contains $\boldsymbol{N}$ nodes at level $\boldsymbol{L}$, it contains at most $\boldsymbol{2 \times N}$ nodes at level $\boldsymbol{L+1}$.
- The maximum number of nodes at any level $\boldsymbol{L}$ in a binary tree $= \boldsymbol{2^L}$
- Total Number of leaf nodes in a Binary Tree is the **Total Number of nodes with 2 children + 1**
- The number of nodes $\boldsymbol{N}$ in a **full** binary tree, where $\boldsymbol{H}$ is the height of the tree, is at least $\boldsymbol{N = 2^{H+1} - 1}$ and at most $\boldsymbol{N = 2 * H + 1}$.
- The total number of edges in a binary tree with $\boldsymbol{N}$ node is $\boldsymbol{N-1}$.
- The number of levels nodes in a **perfect** binary tree, is $\boldsymbol{L = (N+1)/2}$.
- This means that a **perfect** binary tree with $\boldsymbol{L}$ level has $\boldsymbol{N = 2L - 1}$ nodes.
- The number of **internal nodes** in **an almost complete** binary tree of $\boldsymbol{N}$ nodes is $\boldsymbol{N/2}$.



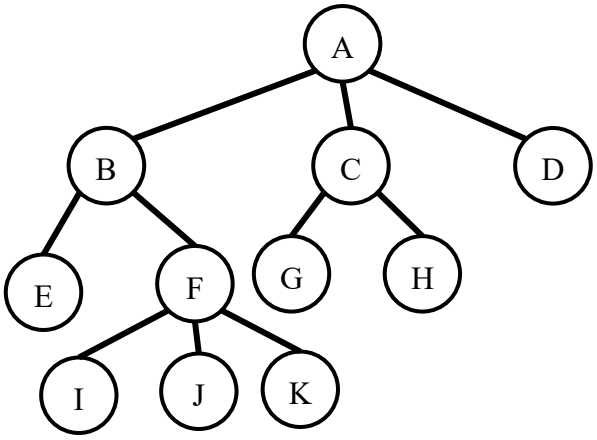
IV.5.2.3. Convert a Generic Tree to Binary Tree

If you are given a generic tree $\boldsymbol{T}$, where each node can have more than two children, you can turn this generic tree into a binary tree $\boldsymbol{T'}$ using the following procedure (starting from the root):

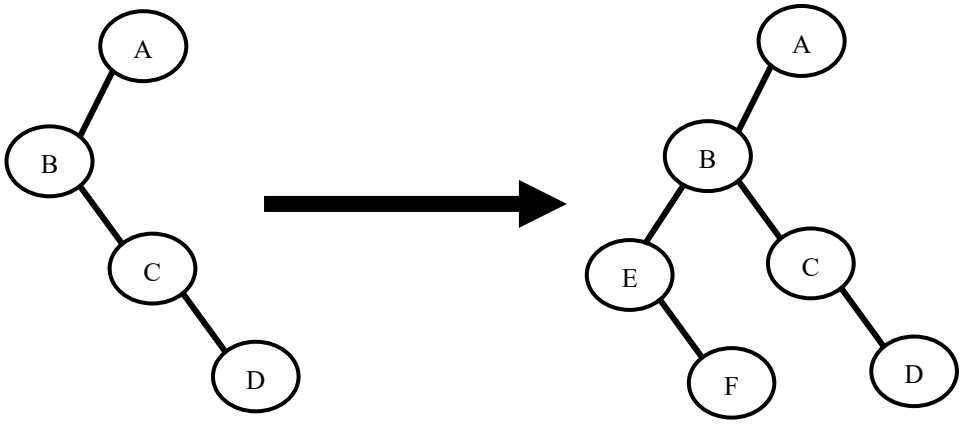
1) If a node $\boldsymbol{v}$ has $\boldsymbol{k}$ children $\boldsymbol{v_1, v_2, v_3, \cdots, v_k}$, first set $\boldsymbol{v_1}$ as the left child of $\boldsymbol{v}$ in $\boldsymbol{T'}$.
2) Then, set $\boldsymbol{v_2, v_3, v_4, \cdots, v_k}$ so, that they become a chain of right children of $\boldsymbol{v_1}$ in $\boldsymbol{T'}$.
3) Repeat these two steps recursively for the remaining nodes in the tree.

Example :

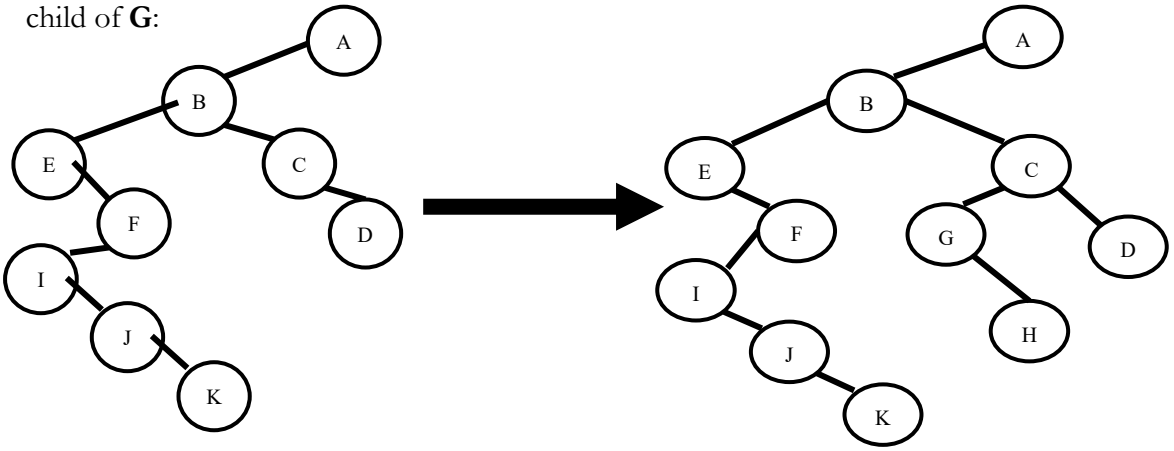
For instance, suppose we are given the following generic tree, and we want to convert it into a binary tree:



1) Using the algorithm discussed previously, we first consider the root node $\boldsymbol{A}$. Node $\boldsymbol{A}$ has three children: $\boldsymbol{B, C}$, and $\boldsymbol{D}$, so we set $\boldsymbol{B}$ as the left child of $\boldsymbol{A}$, and then set $\boldsymbol{C}$ and $\boldsymbol{D}$ as a chain of right children of $\boldsymbol{B}$:
2) We will repeat these steps recursively. Next, we look at **B**, which has two children: **E** and **F**. We set **E** as the left child of **B**, and set **F** as the right child of E:



3) Node **E** has no children, so we do not have to do any additional work for **E**. Node **F**, however, has three children: **I, J**, and **K**. We will set node **I** as the left child of node **F**, and set nodes **J** and **K** as a chain of right children of node **I**.
4) Node C has two children: **G** and **H**. We will set **G** as the left child of **C**, and **H** as the right child of **G**:



IV.5.2.4. Representation of Binary Trees

There are two ways in which a binary tree can be represented. They are:

A. Array-Based Tree Implementation
B. Pointer-Based Tree Implementation

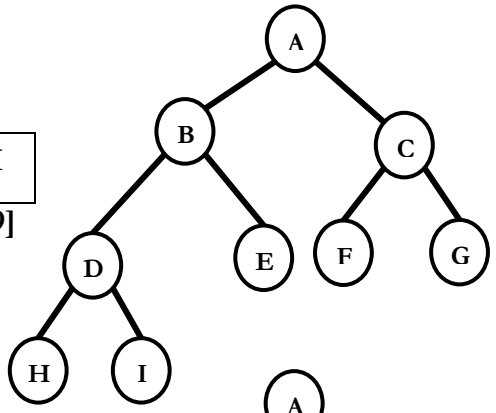
A. Array-Based Tree Implementation

If we implement a binary tree using an array, we essentially flatten out the tree and store its nodes sequentially in memory, level by level.

To implement a binary tree using an array, we will follow these rules:

- Elements start being stored from position $\mathbf{1}$, with position $\mathbf{0}$ remaining vacant.
- The root of the binary tree is placed at index $\mathbf{1}$ of the array.
- The left child of the node at index $\boldsymbol{i}$ should be placed at index $\boldsymbol{2i}$. If node $\boldsymbol{i}$ has no left child, then index $\boldsymbol{2i}$ should be empty.
- The right child of the node at index $\boldsymbol{i}$ should be placed at index $\boldsymbol{2i+1}$. If node $\boldsymbol{i}$ has no right child, then index $\boldsymbol{2i+1}$ should be empty.
- The empty positions in the tree where no node is connected are represented in the array using $\boldsymbol{-1}$, indicating the absence of a node.

Example 1:

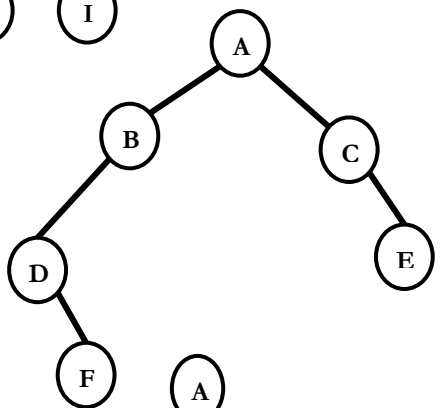


	A	B	C	D	E	F	G	H	I
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

However, binary trees need not be complete, so we cannot avoid potential gaps in our underlying array.

Example 2:

For example, if we implemented the following binary tree using an array, indices $\mathbf{5}, \mathbf{6}$, and $\mathbf{8}$ would be empty:

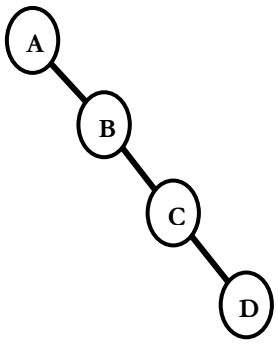


	A	B	C	D			E		F
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Because indices in the underlying array may be skipped, an array-based approach can be space-prohibitive for sparse trees (as memory would be wasted by allocating space for nodes that do not exist).

Example 3:

In the worst case, we could get a stick like this, which would require us to allocate a giant array that remains mostly empty:



	A	B					C		F						D
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

Space Complexity

The worst case occurs when we have a stick, as shown in the example above. When this happens, only a single node can exist at each depth of the tree. This is because the maximum number of nodes that can exist at any depth $\boldsymbol{d}$ is $\mathbf{2^{d-1}}$, assuming the root has a depth of $\mathbf{1}$. Thus, the number of array positions needed to support that depth $\boldsymbol{d}$ is $\mathbf{2^{d-1}}$.

If we have a stick, the depth of our tree would be $\boldsymbol{n}$, and the array size we need to support a depth of d is: $\mathbf{2^0 + 2^1 + 2^2 + \cdots + 2^{n-2} + 2^{n-1} = 2^n - 1 = \mathcal{O}(2^n)}$

The space complexity is $\boldsymbol{\mathcal{O}(2^n)}$, which forces us to use $\boldsymbol{\mathcal{O}(2^n)}$ memory.

B. Pointer-Based Tree Implementation

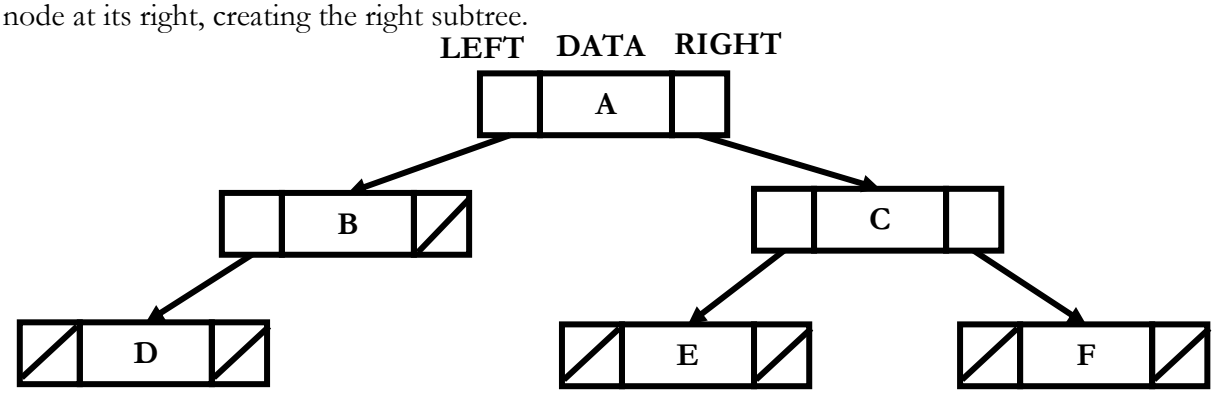
An alternative to the array-based approach is the pointer-based approach. In this approach, pointers are used to link the various nodes of the tree.

Binary Tree Components

There are three **binary tree components**. Every binary tree node has these three components associated with it. It becomes an essential concept for programmers to understand these three binary tree components:

1) Data element
2) Pointer to left subtree
3) Pointer to right subtree

These three binary tree components represent a node. The data resides in the middle. The left pointer points to the child node, forming the left sub-tree. The right pointer points to the child node at its right, creating the right subtree.



Binary Tree Node Declarations

```
struct BTNode {
        int data;
        BTNode *left;
        BTNode *right;
};
typedef BTNode *BTREE;
```

Now, we might have a variable of type **BTREE** called BTree: **BTREE** `BTree;`

The pointers storing **NULL** value indicates that there is no node attached to it.

An alternative would be to store a parent pointer within each node, as shown in the Node definition below. **If** the node is the root, the parent pointer is **NULL** (**nullptr**);

Otherwise, it points to the node's parent.

Using this approach, the time complexity of finding a node's parent would be $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$, since we have direct access to the parent.

```
struct BTNode {
      int data;
      BTNode *parent;
      BTNode *left;
      BTNode *right;
};
```

However, this implementation is rarely used because it is memory intensive, and the benefit of a parent isn't often worth the extra memory. Usually, we do not need a parent pointer, since the behavior of a parent pointer can be emulated using recursion (i.e., when a recursive call unrolls, we automatically move from a child to its parent; there's no need to store an explicit pointer).

IV.5.2.5. Tree Traversals

If you want to process every node in a tree, you will have to complete a **tree traversal**: a systematic method for processing every node in a tree. In this section, we will look at four different tree traversal methods: the **preorder**, **inorder**, **postorder**, and **level-order traversals**, each of which visit the nodes of a tree in a different order. The first three of these traversals (preorder, inorder, postorder) are recursive, while the level-order traversal is iterative.

The names of these traversals actually provide insight into their behavior: the prefix of the traversal name ("pre", "in", and "post") tells us when to process the parent node (i.e., the node you are currently visiting) relative to its children. In a **preorder traversal**, the parent node is processed **before** its left and right children; in a **postorder traversal**, the parent node is processed **after** its left and right children; and in an **inorder traversal**, the parent node is processed **in between** its left and right children.

A. Preorder Traversal

To conduct a preorder traversal, you would complete the following steps, in this order:

1) process the parent node (the node the recursion is currently on)
2) recursively process the left subtree
3) recursively process the right subtree

The code for a preorder traversal is shown below:

```
void preorder(BTREE BTree) {
if (!BTree) return;
```

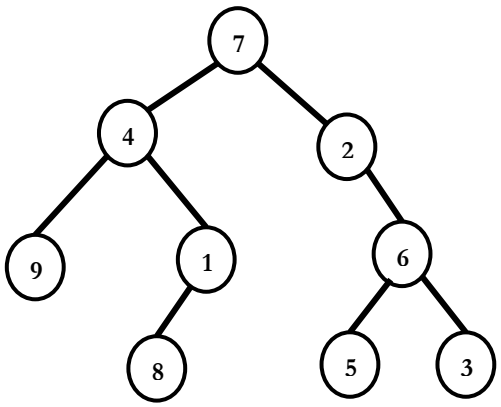
```
process(BTree -> data);          // process the current node
preorder(BTree -> left);         // recurse into left child
preorder(BTree -> right);        // recurse into right child
} // preorder()
```

Note that the **process()** function in the code above is just a placeholder for the work that is done on each node.

Example:

For example, to print out the nodes in a preorder traversal, we use a print statement in **process()** or replace **process()** with a print statement:

```
void process(int data) {
      cout << data << endl;
}
```



Exercise :

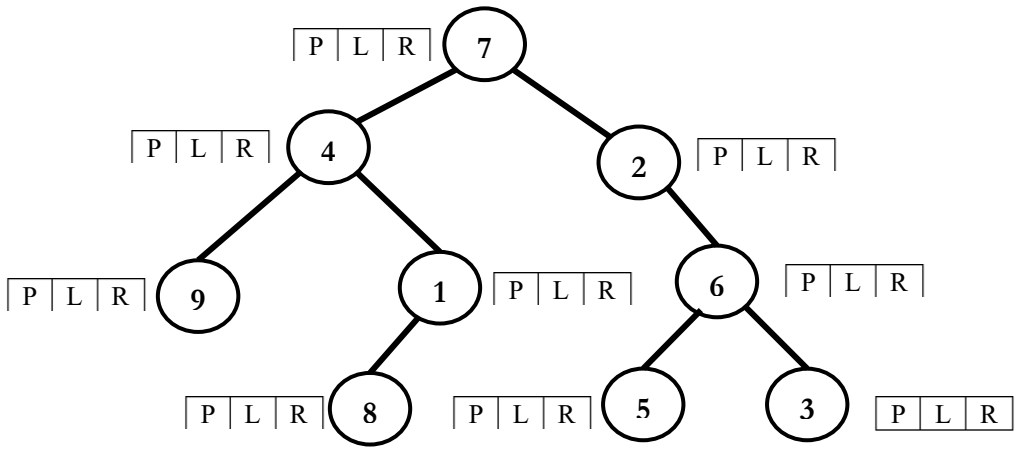
What is the preorder traversal of the following tree?

That is, if you were to conduct a preorder traversal of the tree, in what order would you process the nodes?

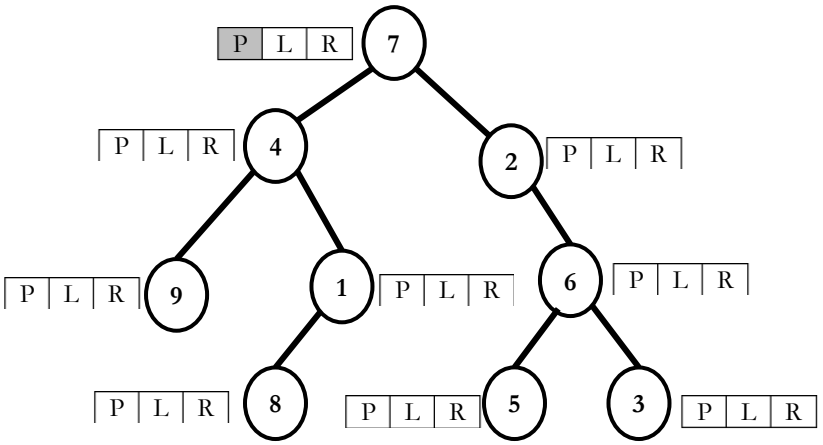
In a preorder traversal, we will always process a node before we recursively process its left and right children. There are three steps we have to complete at each node: process its value, recursively visit its left child, and then recursively visit its right child.

To illustrate this process, we will label each node with three letters "***P*** **,** ***L***, and ***R***" that represent each of these three steps.

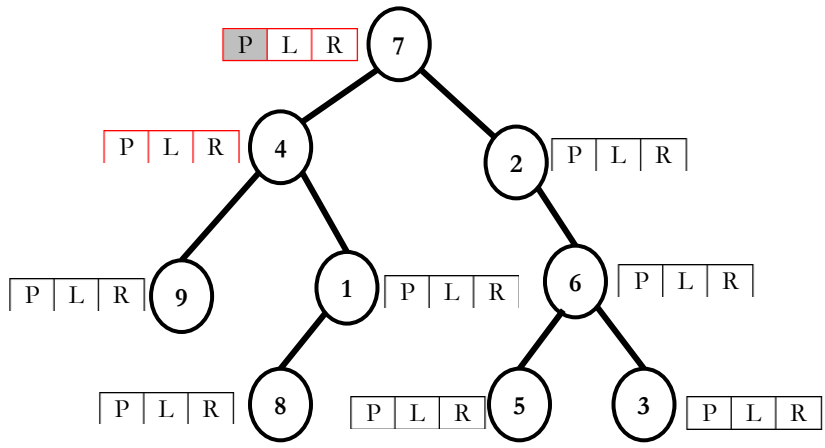
- We will mark each letter as "completed" as soon as we finish each step (i.e., we will mark the ***P*** step of a node as completed after we finish processing its value, we will mark the ***L*** step of a node as completed as soon as we finish processing its left subtree, and we will mark the ***R*** step of a node as completed as soon as we finish processing its right subtree).
- In a preorder traversal, ***P*** must be completed before ***L***, and ***L*** must be completed before ***R***.



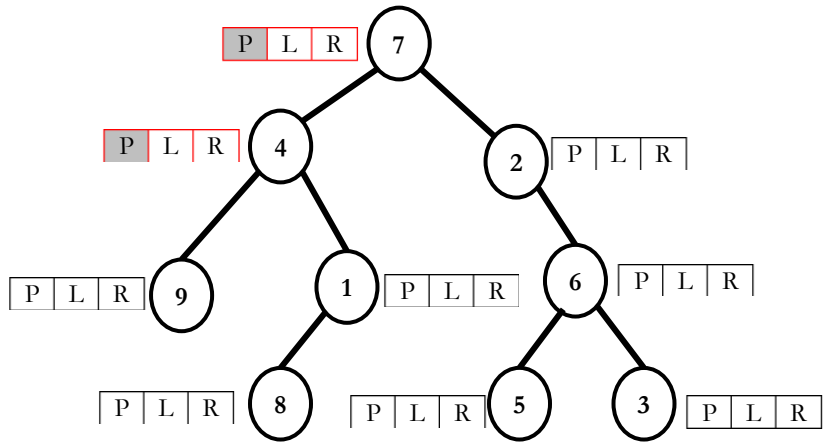
We start at the root (**7**), which is our current stack frame. The first step is to process the data of the current node, so the first element in our preorder traversal is **7**. In fact, the first element of a preorder traversal is always the root, since a node is always processed before its children.

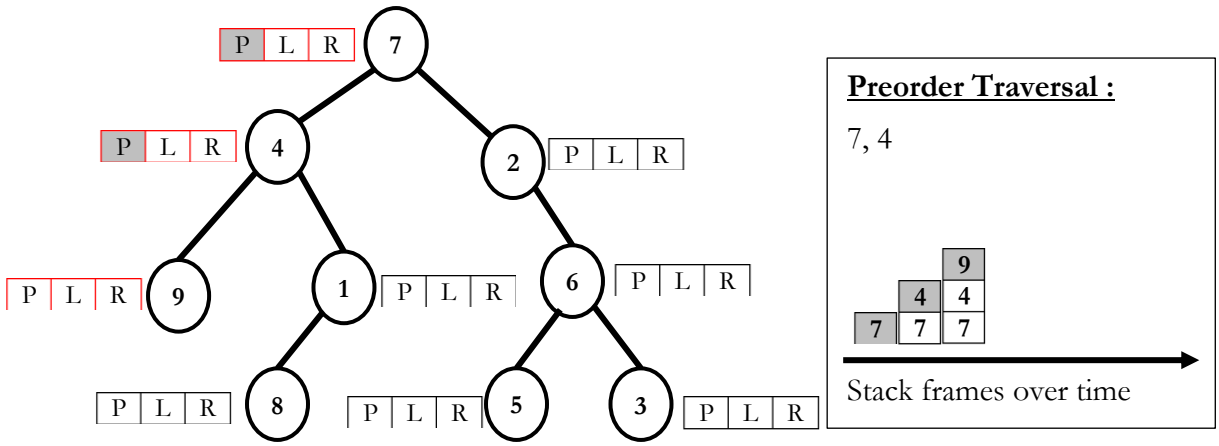


After processing the value of the root node, we mark its ***P*** step as complete. The next step is to recursively process the left child (***L***), so we make a recursive call on node **4**, which becomes the value of our current stack frame.

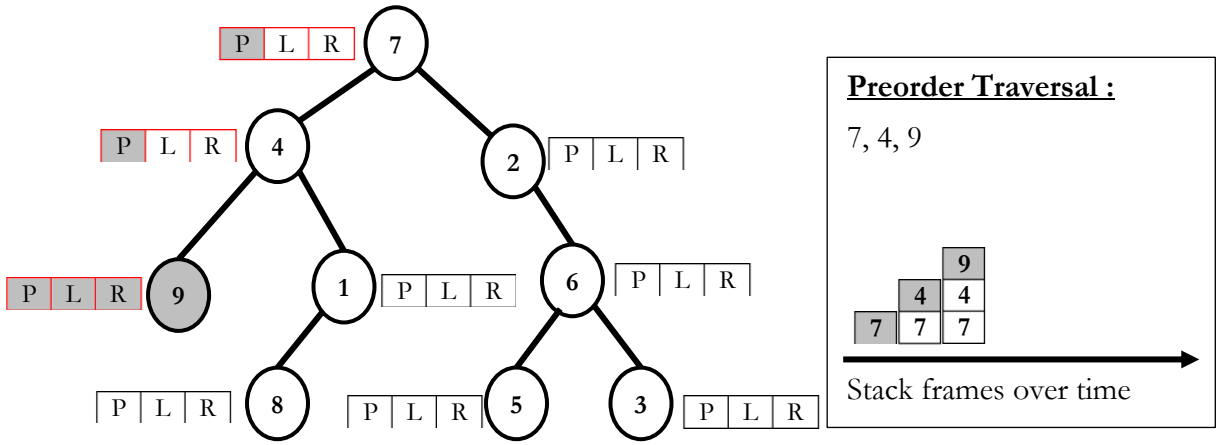


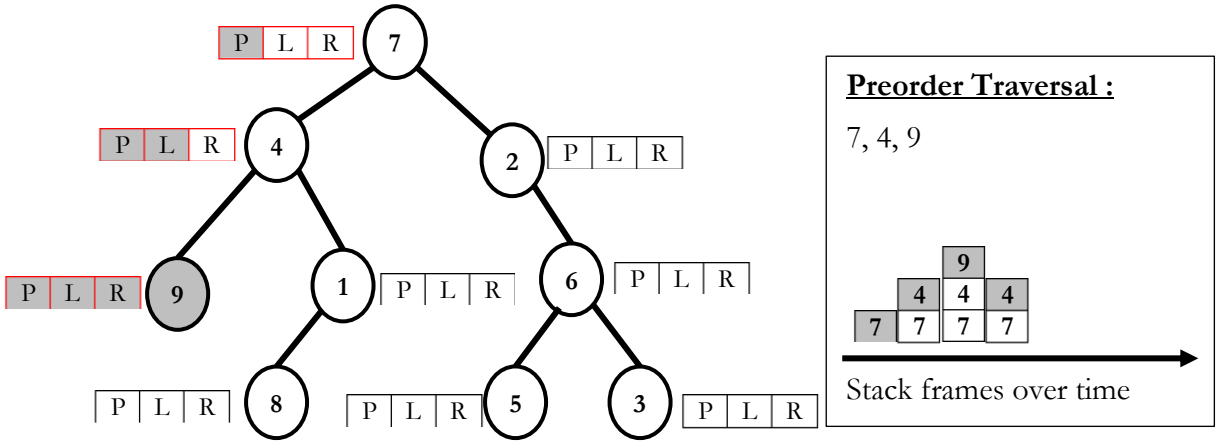
First, we will process the value of our current node, so **4** is the next value in our preorder traversal. We then mark the ***P*** step of node **4** as completed and make a recursive call on the left child of **4** (node **9**), which becomes the value of our current stack frame.



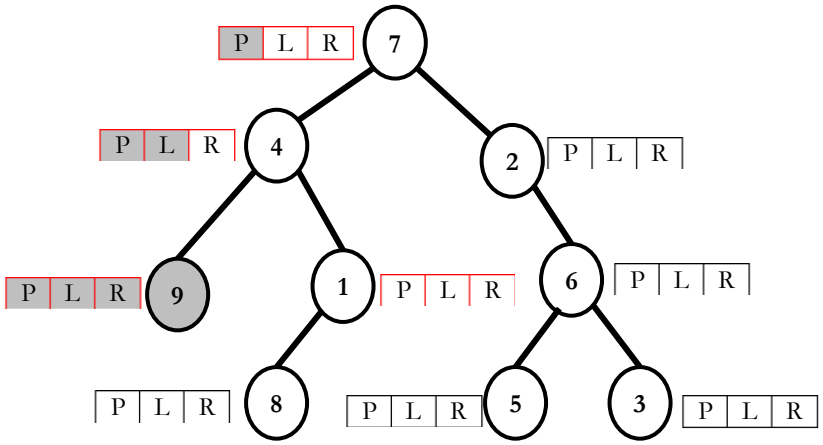


We process the value of our current node, so **9** is next in our preorder traversal. Since **9** has no children, the recursive calls on its left and right children can be completed trivially, and we can mark the ***L*** and ***R*** steps of node **9** as complete. Since all the work for node **9** is finished, the recursive call unrolls, and we return to the stack frame of node **4**. The ***L*** step of node **4** is now complete.



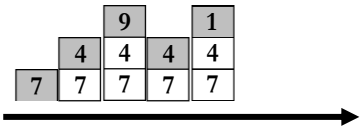


Since the recursive call on the left child is done, the next step is to make a recursive call on the right child of node **4**, which is node **1**. Node **1** thus becomes the value of our current stack frame.



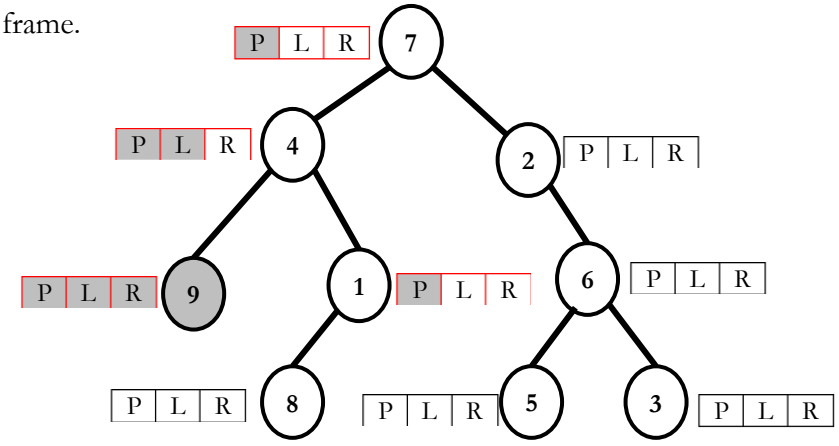
Preorder Traversal :

7, 4, 9



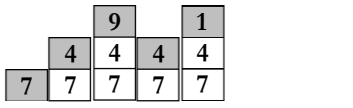
Stack frames over time

We first process the value of our current node, so **1** is next in our preorder traversal. We then make a recursive call on **1**'s left child, or node **8**. Node **8** then becomes the node on our current stack frame.

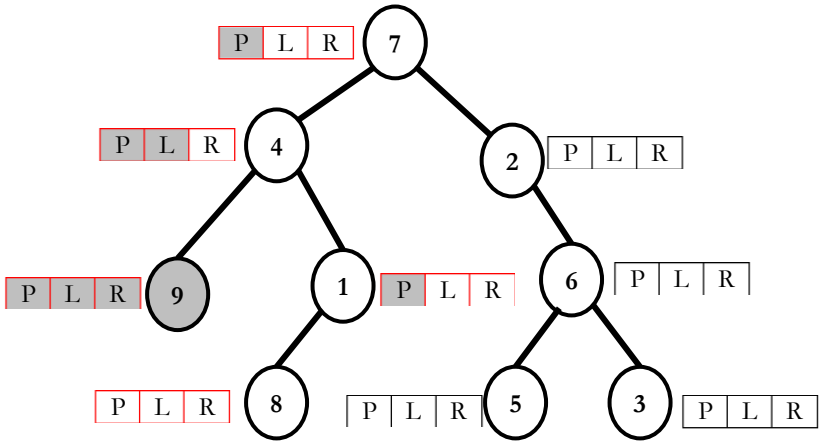


Preorder Traversal :

7, 4, 9,1

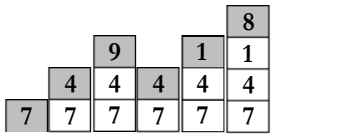


Stack frames over time



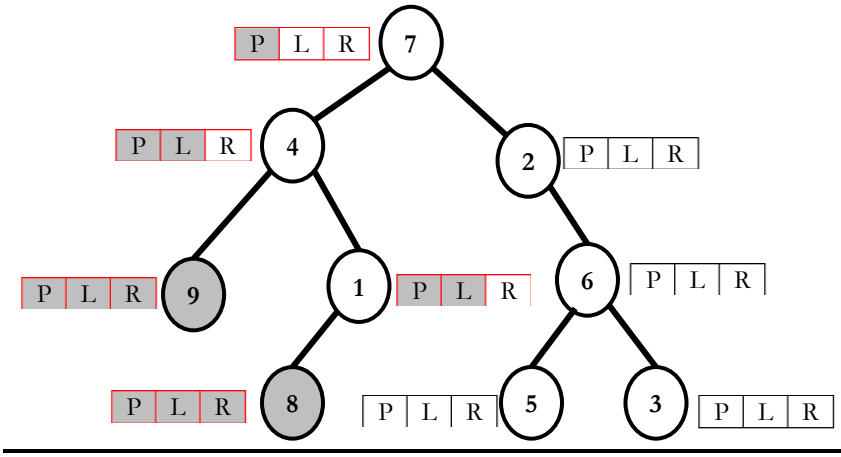
Preorder Traversal :

7, 4, 9,1



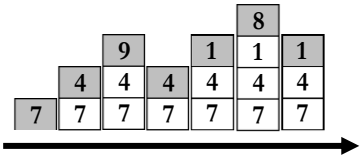
Stack frames over time

The value of node **8** is processed first, so **8** is next in our preorder traversal. Since node **8** has no left or right children, we do not need to do any more work for this node. The recursion unrolls, and we return to the stack frame of node **1**.



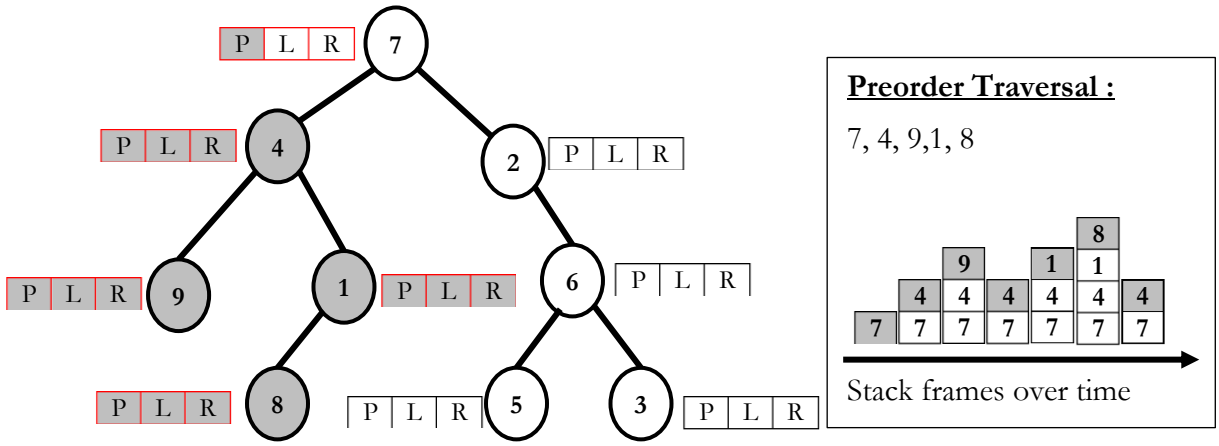
Preorder Traversal :

7, 4, 9,1, 8

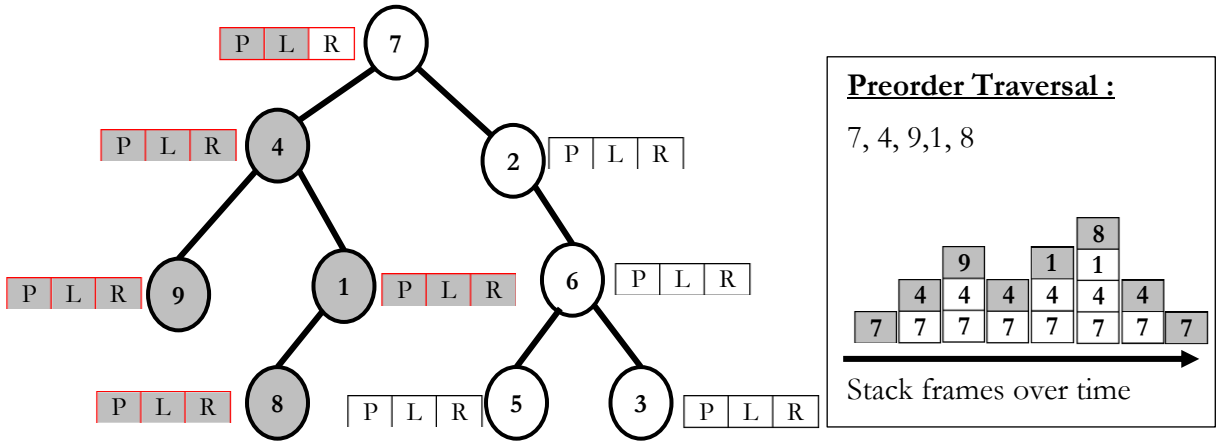


Stack frames over time

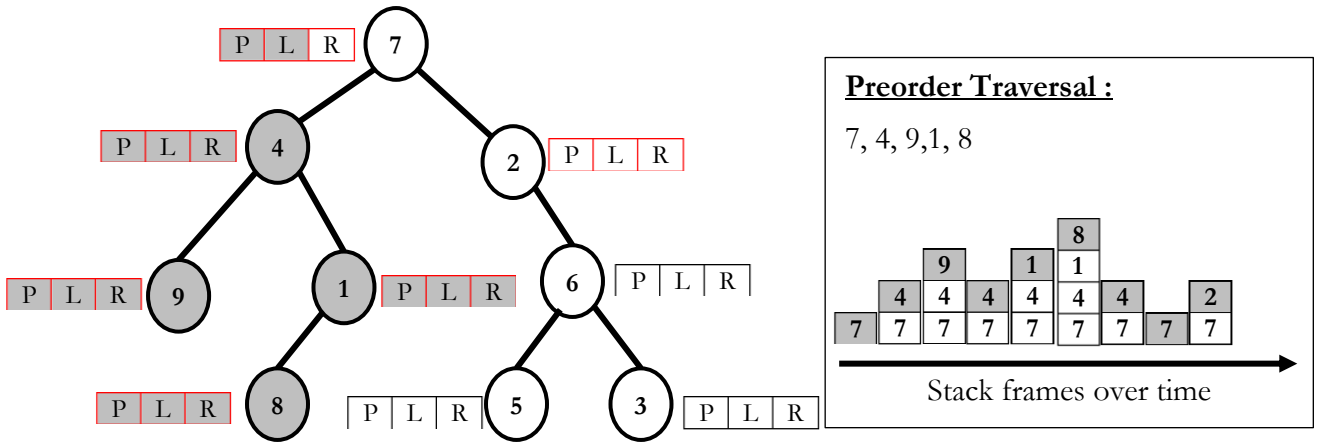
Now, we will make a recursive call on the right child of node **1**. However, node **1** has no right child, so this step can be done trivially. All three steps for node **1** are now complete, so the recursion unrolls, and we return to the stack frame of node **4**.



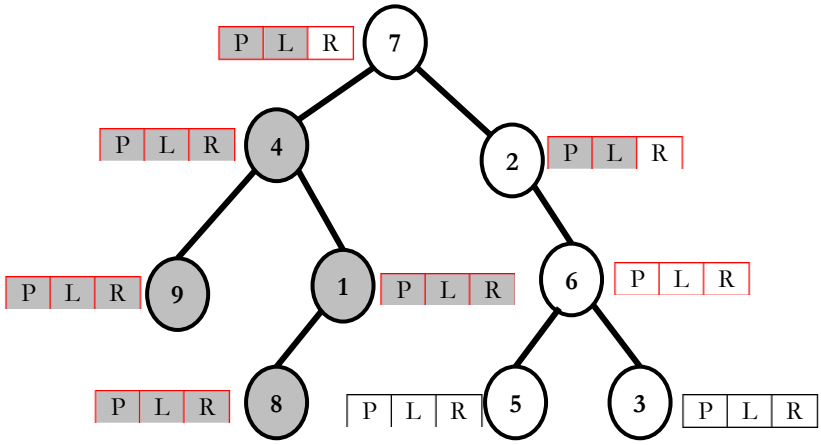
All three steps for node **4** are done, so the recursion unrolls, and we return to the stack frame of node **7**. Since the entire left subtree of node **7** has been completely processed, we can mark the ***L*** step of the root node as complete.



Our next step is to make a recursive call on the right child of **7**, or node **2**.

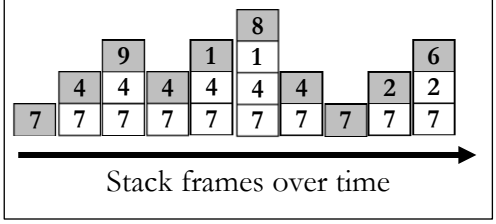


We then process the value of node **2**, so **2** is next in our preorder traversal. Since **2** has no left child, the recursive call on its left child can be completed trivially. We then make a recursive call on the right child of **2**, or node **6**. Node **6** now becomes the node on our current stack frame.



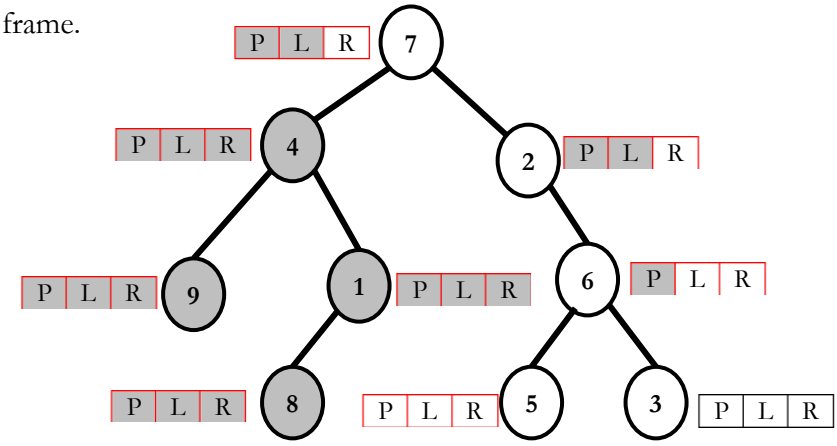
Preorder Traversal :

7, 4, 9,1, 8, 2



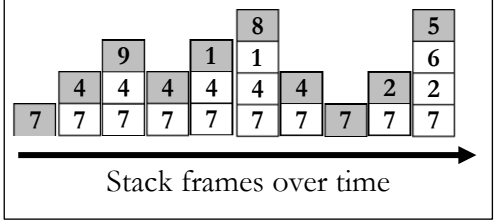
Stack frames over time

We now process the value of node **6** and add it to our preorder traversal. Then, we make a recursive call on the left child of node **6**, or node **5**. Node **5** now becomes the node on our current stack frame.



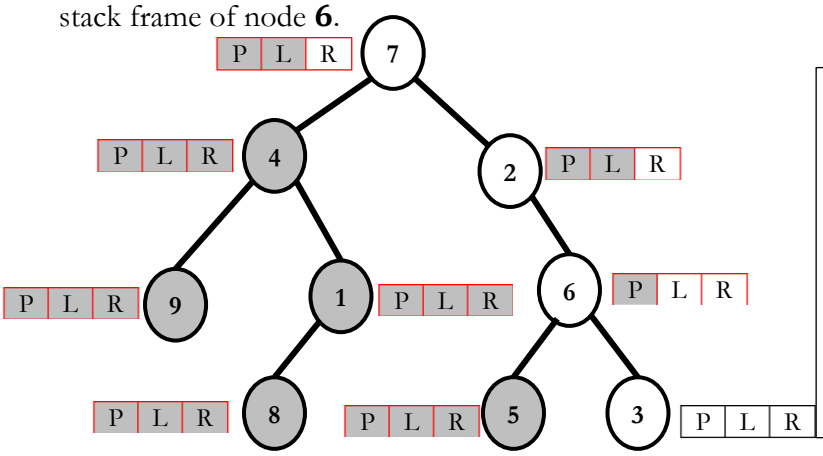
Preorder Traversal :

7, 4, 9,1, 8, 2, 6



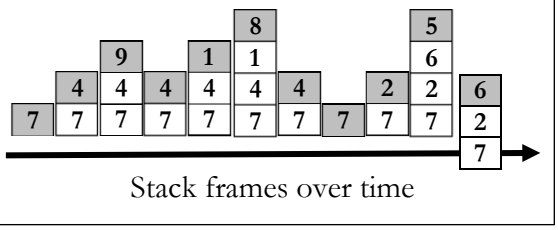
Stack frames over time

We process the value of node **5** by adding it to our preorder traversal. Since **5** has no left or right children, no more work needs to be done on this node. The recursion unrolls, and we return to the stack frame of node **6**.

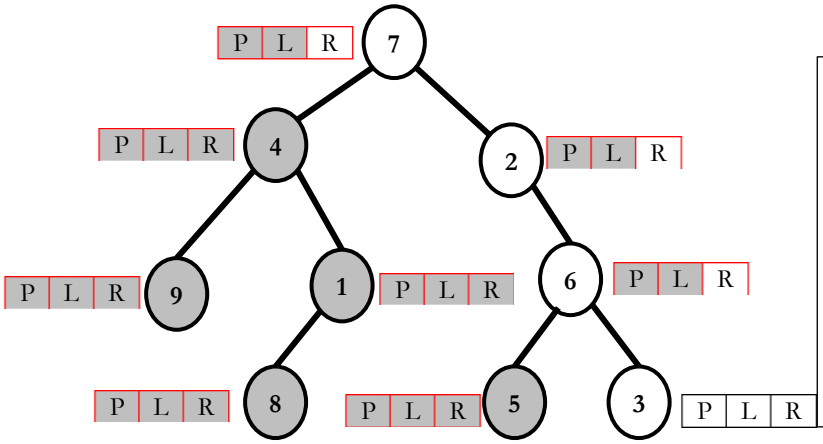


Preorder Traversal :

7, 4, 9,1, 8, 2, 6,5

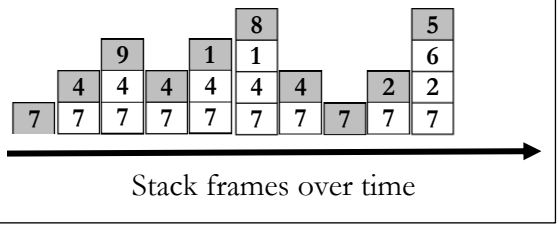


Stack frames over time



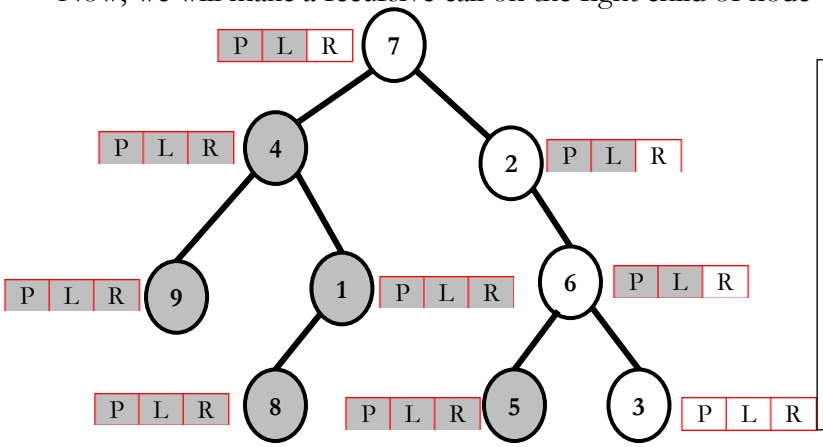
Preorder Traversal :

7, 4, 9,1, 8, 2, 6,5



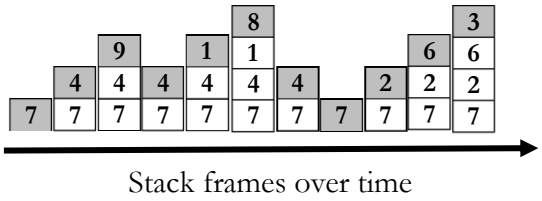
Stack frames over time

Now, we will make a recursive call on the right child of node **6**, or node **3**.



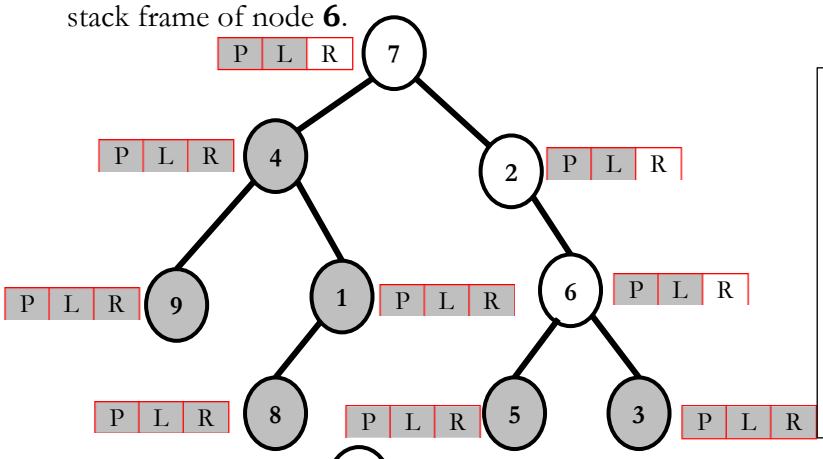
Preorder Traversal :

7, 4, 9,1, 8, 2, 6,5



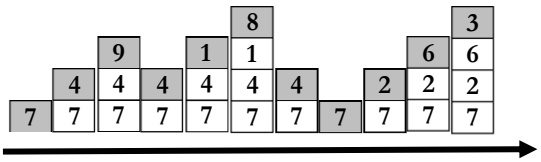
Stack frames over time

We process the value of node **3** by adding it to our preorder traversal. Since **3** has no left or right children, no more work needs to be done on this node. The recursion unrolls, and we return to the stack frame of node **6**.

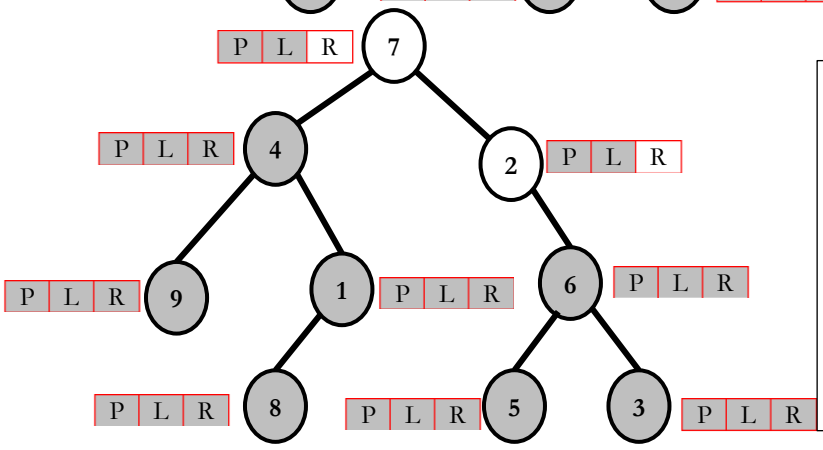


Preorder Traversal :

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

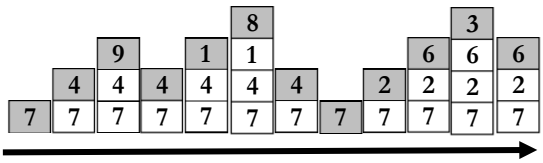


Stack frames over time



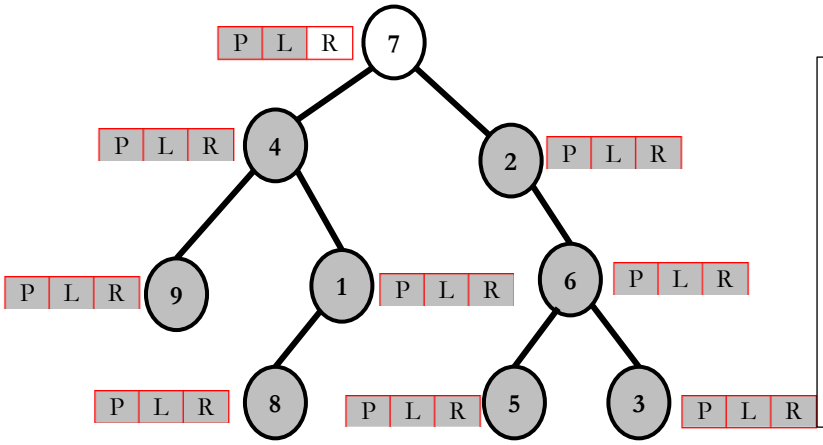
Preorder Traversal :

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



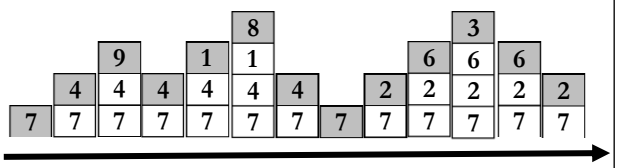
Stack frames over time

All three steps for node **6** have been completed, so the recursion unrolls, and we return to the stack frame of node **2**.



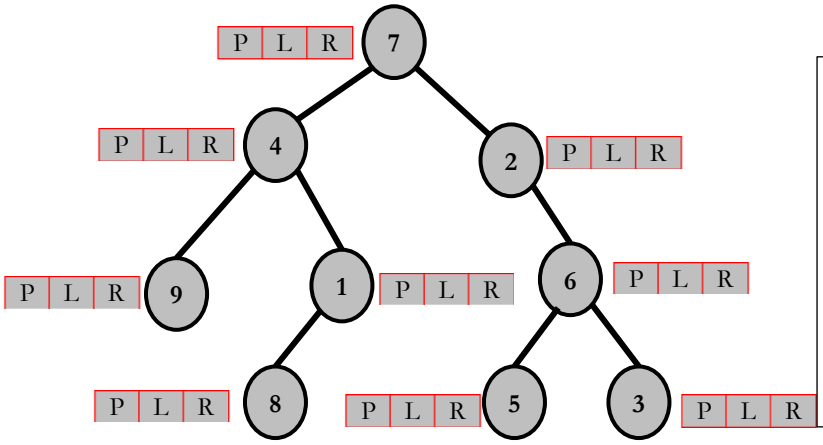
Preorder Traversal :

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



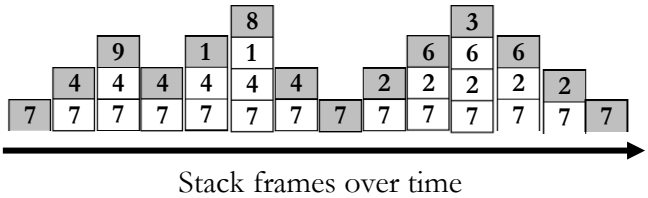
Stack frames over time

All three steps for node **2** have also been completed, so the recursion unrolls, and we return to the stack frame of node **7**.



Preorder Traversal :

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



Stack frames over time

The function returns, and the traversal is complete. The preorder traversal of this tree is: **7, 4, 9, 1, 8, 2, 6, 5, 3**.

Uses of Postorder Traversal:

- Postorder traversal is used to delete the tree.
- Postorder traversal is also useful to get the postfix expression of an expression tree.
- Postorder traversal can help in garbage collection algorithms, particularly in systems where manual memory management is used.

B. Inorder Traversal

To conduct an inorder traversal, you would complete the following steps, in this order:

1) recursively process the left subtree
2) process the parent node (the node the recursion is currently on)
3) recursively process the right subtree

The code for an inorder traversal is shown below:

```
void inorder(BTREE BTree) {
        if (!BTree)
        return;
        inorder(BTree -> left);              // recurse into left child
        process(BTree -> data);              // process the current node
        inorder(BTree -> right);             // recurse into right child
} // inorder()
```

In the previous exercise, we used the same method, the inorder traversal of this tree is: **9, 4, 8, 1, 7, 2, 5, 6, 3**.

Uses of Inorder Traversal:

- In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.

- To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.
- Inorder traversal can be used to evaluate arithmetic expressions stored in expression trees.

C. Postorder Traversal

To conduct a postorder traversal, you would complete the following steps, in this order:

1) recursively process the left subtree
2) recursively process the right subtree
3) process the parent node (the node the recursion is currently on)

The code for a **postorder** traversal is shown below:

```
void postorder(BTREE BTree) {
      if (!root)
            return;
      postorder(BTree -> left);          // recurse into left child
      postorder(BTree -> right);         // recurse into right child
      process(BTree -> data);            // process the current node
} // postorder()
```

In the previous exercise, we used the same method, the postorder traversal of the tree is **9, 8, 1, 4, 5, 3, 6, 2, 7**.

Uses of Preorder Traversal:

- Preorder traversal is used to create a copy of the tree.
- Preorder traversal is also used to get prefix expressions on an expression tree.

D. Level-Order Traversal

The **level-order traversal** is another traversal method that can be used to process the nodes of a tree. In a level-order traversal, nodes are processed in order of increasing depth, where nodes on the same level (i.e., nodes that have the same depth) are processed from left to right.

In the previous exercise, the level-order traversal processes nodes in order of increasing depth, from left to right: **7, 4, 2, 9, 1, 6, 8, 5, 3.**

Unlike the other three traversal methods, the level-order traversal is iterative rather than recursive. To conduct a level-order traversal, a queue is used in place of recursive calls. The code for a level-order traversal is shown below:

```
void level_order(BTREE BTree) {
      if (!BTree)
            return;
      QUEUE bfs = Init_Queue();
      EnQueue (bfs, BTree);              // push root into queue
      while (!isEmpty(bfs)) {            // while queue not empty
```

```
            BTNode* current = DeQueue(bfs);
            process(current -> data);
            if (current->left) {
                  EnQueue(bfs, current->left);
            }
            if (current->right) {
                  EnQueue(bfs, current-> right);
            }
      } // while
} // level_order()
```

In the code, nodes are pushed into the queue in level order (i.e., nodes closer to the root are pushed in before nodes closer to the leaves). Since queues support first-in, first-out (FIFO) behavior, we are able to process the nodes in the same order that they are pushed into the queue. Consider the tree in the previous example:

First, we push (**EnQueue**) the root into the queue, as shown on line 5:

Level-order traversal:

Queue : bfs

7		

Then, as long as the queue is not empty, we would repeat the following:

1) Take out the node at the front (by **peek**) of the queue.
2) Process the node (in this case, print it to the level-order traversal).
3) Push (**EnQueue**) the node's left child into the queue, if one exists.
4) Push (**EnQueue**) the node's right child into the queue, if one exists.

During the **first iteration** of the **while** loop, we take out the node at the front of the queue (node **7**), add it to our level-order traversal, and push **7**'s children (nodes **4** and **2**) into the queue.

Level-order traversal:

7

Queue : bfs

4	2	

During the **second iteration,** we take out the node at the front of the queue (node **4**), add it to our level-order traversal, and push **4**'s children (nodes **9** and **1**) into the queue.

Level-order traversal:

7, 4

Queue : bfs

2	9	1

During the **third iteration**, we take out node **2**, add it to our level-order traversal, and push **2**'s child (node **6**) into the queue.

Level-order traversal:

7, 4, 2

Queue : bfs

9	1	6

During **the fourth iteration**, we take out node **9** and add it to our level-order traversal. Since **9** has no children, nothing gets pushed into the queue.

Level-order traversal:

7, 4, 2, 9

Queue : bfs

1	6	

During **the fifth iteration,** we take out node **1**, add it to our level-order traversal, and push **1**'s child (node **8**) into the queue.

During **the sixth iteration**, we take out node **6**, add it to our level-order traversal, and push **6**'s children (**5** and **3**) into the queue.

Level-order traversal:
7, 4, 2, 9, 1, 6
Queue : bfs

8	5	3

Level-order traversal:
7, 4, 2, 9, 1
Queue : bfs

6	8	

During **the seventh iteration**, we take out node **8** and add it to our level-order traversal. **8** has no children, so nothing gets pushed into the queue.

Level-order traversal:
7, 4, 2, 9, 1, 6, 8
Queue : bfs

5	3	

During **the eighth iteration**, we take out node **5** and add it to our level-order traversal. **5** has no children, so nothing gets pushed into the queue.

Level-order traversal:
7, 4, 2, 9, 1, 6, 8, 5
Queue : bfs

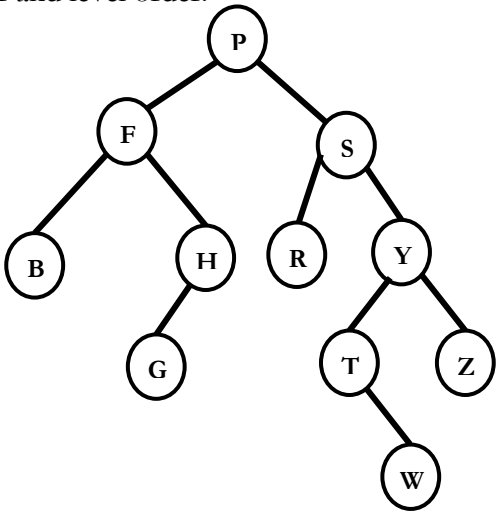
3		

During **the ninth iteration**, we take out node **3** and add it to our level-order traversal. 3 has no children, so nothing gets pushed into the queue. After this iteration, the queue is empty, and our level-order traversal is complete.

Level-order traversal:
7, 4, 2, 9, 1, 6, 8, 5, 3
Queue : bfs

Example :

Traverse the following binary tree in pre, post, inorder and level order.



Preorder traversal yields:

P, F , B, H, G , S, R, Y, T, W, Z

Inorder traversal yields:

B, F , G , H, P, R, S, T, W, Y, Z

Postorder traversal yields:

B, G , H, F , R, W, T, Z, Y, S, P

Level order traversal yields:

P, F , S, B, H, R, Y, G , T, Z, W

IV.5.2.6. Binary Search Trees (BST)

One pitfall of the standard binary tree is that its elements are not ordered in any manner. This can make searching difficult: if you wanted to find a given element, that element could be anywhere in the tree! To address this issue, we will introduce the **binary search tree (BST)**, which is a binary tree whose elements are ordered based on a sorting invariant.

A special type of binary trees, called **binary search tree** is a binary tree in symmetric order. Each node has a key, and for each node in the tree the value of any node in its **left subtree is less** than the value of the node and the value of any node in its **right subtree is greater** than the value of the node.

A standard non-empty binary search tree exhibits the following properties:

- The key of any node is always greater than all of the keys in its left subtree.
- The key of any node is always less than or equal to all of the keys in its right subtree.
- Both the left and right subtrees of any node are also binary search trees themselves.

The following given figure is an example of a binary search tree.

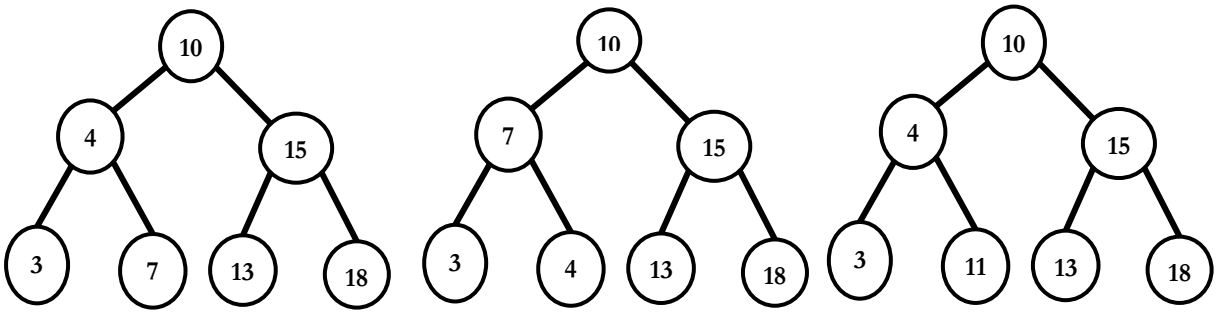
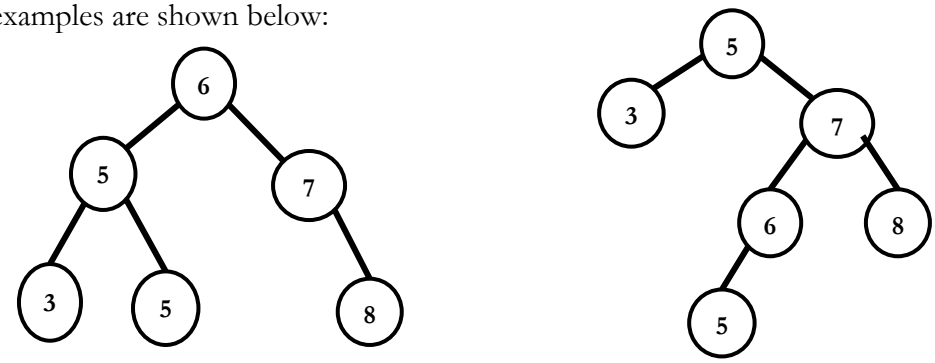


Figure 4.3: Representing a binary search tree

Only the first tree is a binary search tree. The second tree is not a binary search tree because the right child of **7**, or **4**, is smaller than **7**. The third tree is not a binary search tree because **11** is in the left subtree of **10**, even though **11** is larger than **10**.

For a tree to be a valid binary search tree, only the rules specified above need to be met. As a result, it is possible for two binary search trees to have different structures, even if they have the same keys. Two examples are shown below:



Since keys in a binary search tree are ordered in a way such that all keys to the left are smaller and all keys to the right are larger, the inorder traversal of a binary search tree will always visit its keys in sorted order.

IV.5.2.6.1. Operations in Binary Search Trees

Binary search trees are useful for efficient searching, insertion, and deletion operations. Searching for a value in a balanced binary search tree has a time complexity of $\boldsymbol{O(logn)}$, where $\boldsymbol{n}$ is the number of nodes in the tree. This is because each comparison of the search value with a node's value eliminates half of the remaining nodes in the tree, remember in this case we assume that the binary tree is not left or right-skewed.

In a binary search tree, many operations can be performed. Some of the operations are mentioned below in detail.

A. Searching in a Binary Search Tree

Binary search trees support efficient searching since you will only need to look down one side of the tree at each level.

- If you want to search for a target key k, and k is smaller than the current root element you are looking at, then k must be in the left subtree if it exists.
- Otherwise, if k is larger than the current root element, then k must be in the right subtree if it exists.

You can think of this as a "binary search" of the tree's elements since you are removing half of the search space every time; hence why this data structure is called a binary search tree!

The search algorithm can be implemented both iteratively and recursively. The following iterative implementation returns a pointer to the node with key k if it exists; otherwise, it returns **NULL** (**nullptr**).

```
BTNode* search_BST(BTREE BTree, int k) {
      while (BTree != NULL && k != BTree -> data) {
            if (k < BTree -> data) {
                  BTree = BTree -> left;
            }
            else {
                  BTree = BTree -> right;
            }
      }
      return BTree;
}
```

The following implementation does the same thing, but with recursion instead of iteration. Note that this implementation is tail recursive, since the recursive call is always done last.

```
BTNode* search_BST(BTREE BTree, int k) {
      if (BTree == NULL || BTree -> data == k) {
            return BTree;
            }
      if (k < BTree -> data) {
            return search_BST(BTree -> left, k);
      }
      return search_BST(BTree -> right, k);
}
```

What is the time complexity of searching for a key in a binary search tree? In the worst case, the binary search tree may take the form of a stick, and you may have to search the entire tree to find a given key. This results in $\mathcal{O}(n)$, runtime, where n is the number of nodes in the tree.

The Maximum and the Minimum

- To find the minimum identify the leftmost node, i.e. the farthest node you can reach by following only left branches.

- To find the maximum identify the rightmost node, i.e. the farthest node you can reach by following only right branches.

```
BTNode* Min_BST(BTREE BTree) {
      if (BTree == NULL) {
            return NULL;
      }
      while (BTree -> left) {
            BTree = BTree -> left;
      }
      return BTree;
}
```

```
BTNode* Max_BST(BTREE BTree) {
      if (BTree == NULL) {
            return NULL;
      }
      while (BTree -> right) {
            BTree = BTree -> right;
      }
      return BTree;
}
```

B. Inserting into a Binary Search Tree

To insert a key into a binary search tree, one needs to maintain the binary property of the tree.

To achieve this, the appropriate location for the inserted element is obtained and then the element is inserted into the tree. Therefore, We follow a procedure that is similar to search: we start at the root and trace a path downwards, but this time we want to look for a **NULL** (**nullptr**) position to append the node. The implementation for BST insertion is shown below.

```
void insert_BST(BTREE &BTree, int k) {
      if (BTree == NULL) {
            BTree = Create_BTNode(k);
      }
      else
            if (k < BTree -> data) {
                  insert_BST(BTree -> left, k);
            }
            else {
                  insert_BST(BTree -> right, k);
            }
}
```

Similar to search, the time complexity of insert depends on the height of the tree. In the worst case, the binary search tree could be in the form of a stick, requiring the algorithm to traverse the entire tree before finding the position to insert the new node — this would take $\boldsymbol{O(n)}$ time.

C. Removing from a Binary Search Tree

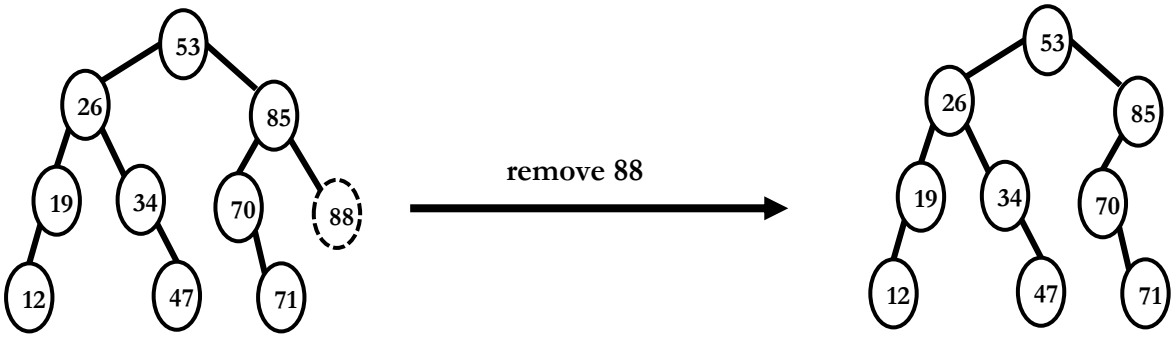
So far, we have covered search and insertion. However, what if we wanted to remove a key from a binary search tree? Removal is slightly more complicated, since we will have to detach a node

while still maintaining the sorted property of a binary search tree. There are four different conditions we have to consider when removing a node from a binary search tree:

1) The node we want to remove has no children.
2) The node we want to remove has no left child.
3) The node we want to remove has no right child.
4) The node we want to remove has two children.

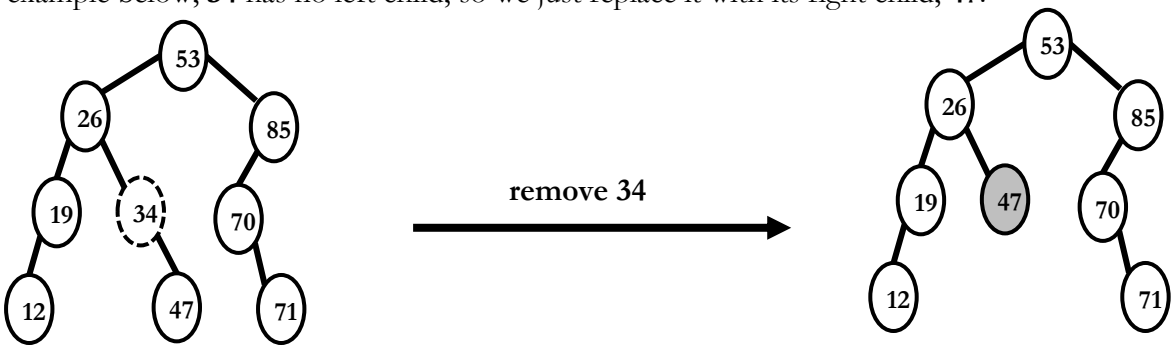
1) The node we want to remove has no children.

This condition is trivial; if we want to remove a leaf node, we can just snip off the leaf. In the example below, **88** has no children, so we just delete it.



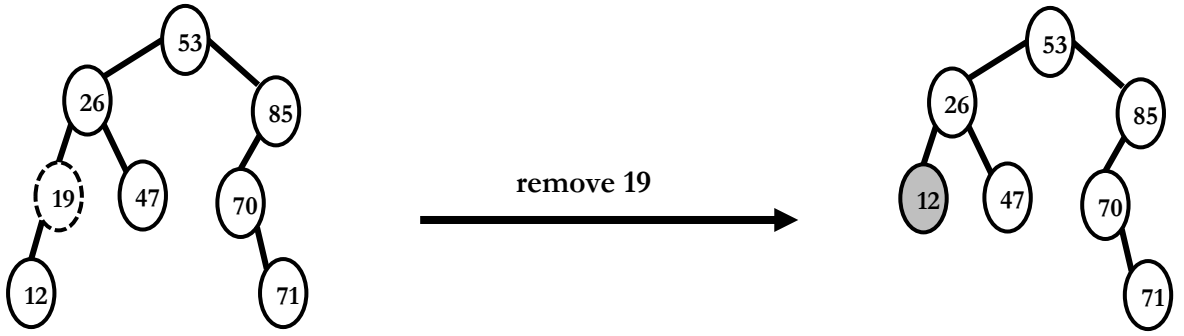
2) The node we want to remove has no left child.

If the node we want to remove has no left child, we can just replace it with its right child. In the example below, **34** has no left child, so we just replace it with its right child, **47**.



3) The node we want to remove has no right child.

If the node we want to remove has no right child, we can just replace it with its left child. In the example below, **19** has no right child, so we just replace it with its left child, **12**



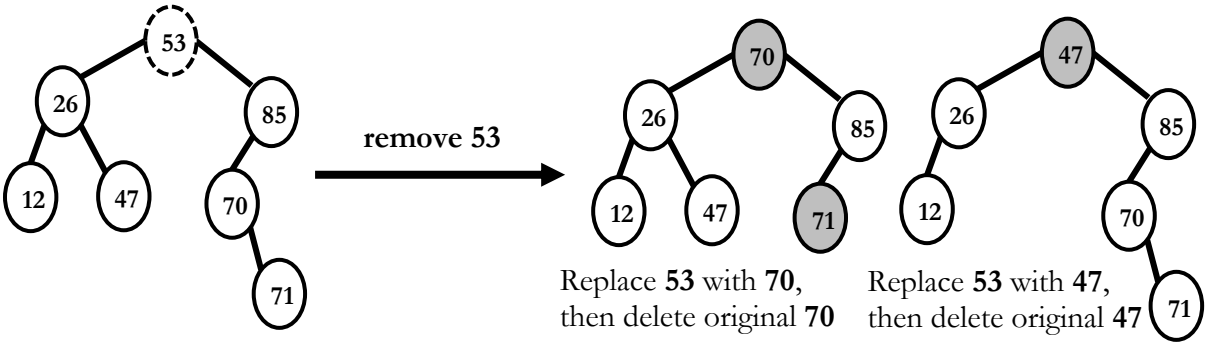
4) The node we want to remove has two children.

This is a trickier case, since the node to remove points to two things instead of one. As a result, we will need a procedure for replacing the deleted node in a way that preserves the binary search tree property.

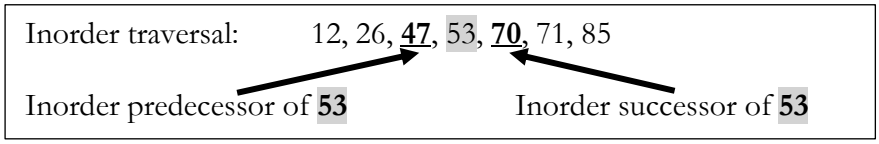
A key observation to make here is that any node in the left subtree of the deleted node must be smaller than any node in the right subtree of the deleted node. Thus, we can preserve the binary search tree property by either

A. replacing the deleted node with the smallest node in its right subtree, or
B. replacing the deleted node with the largest node in its left subtree.

For example, suppose we wanted to remove **53** from the following binary search tree. We can either replace **53** with the smallest element in its right subtree (**70**), or with the largest element in its left subtree (**47**). Both approaches would maintain the binary search tree property:



The smallest node in the right subtree of a given node ***k*** is known as the inorder successor of node ***k***. Similarly, the largest node in the left subtree of a given node ***k*** is known as the inorder predecessor of node ***k***. This is because the inorder successor of ***k*** comes directly after (i.e., succeeds) ***k*** in an inorder traversal, and the inorder predecessor of ***k*** comes directly before (i.e., precedes) ***k*** in an inorder traversal.



Because the inorder successor and predecessor are directly adjacent to the node we want to delete, it is safe to replace the deleted node with either of these values.

The code for remove is shown below. In this version, the removed node is substituted with the inorder successor (the logic for the inorder predecessor version is analogous but not displayed here).

```
// Removes the node from the Binary Search Tree with value "data"
void remove_BST(BTREE &BTree, int data) {
      BTNode* node_to_delete = BTree;
      BTNode* inorder_successor;
      if (BTree == NULL) {
            return;
      }
      else if (data < BTree -> data) {
                  remove_BST(BTree -> left, data);
            }
```

```
    else if (BTree -> data < data) {
            remove_BST(BTree -> right, data);
    }
            else {
                if (BTree ->left == NULL) {
                    BTree = BTree ->right;
                    delete node_to_delete;
                }
                else if (BTree -> right == NULL) {
                        BTree = BTree -> left;
                        delete node_to_delete;
                    }
                else {  // Node to delete has both left and right subtrees
                    inorder_successor = BTree -> right;
                    while (inorder_successor -> left) {
                        inorder_successor = inorder_successor -> left;
                    } // while
                    node_to_delete -> data = inorder_successor -> data;
                    remove_BST(BTree -> right, inorder_successor -> data);
                }
            } // else
} // tree_remove()
```

Replacing a deleted node with either its inorder successor or predecessor is the simplest way to approach deletion, as it maintains the BST property without requiring you to reconfigure the other elements in the tree.

The process for finding the inorder successor or predecessor of any node is also relatively straightforward. To identify the inorder successor of a node, go right once, then go as far left as possible until you reach a node without a left child. To identify the inorder predecessor of a node, go left once, then go as far right as possible until you reach a node without a right child.

IV.5.2.7. Summary of Binary Search Tree Time Complexities

Operation	Best-Case Time	Average-Case Time	Worst-Case Time
Searching	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Inserting	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Deleting	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$