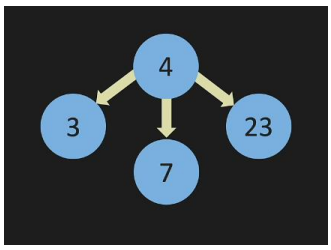
Contents

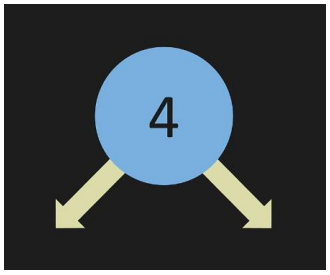
Trees

LL is a special case of tree that doesn't fork.

A tree can fork many times.

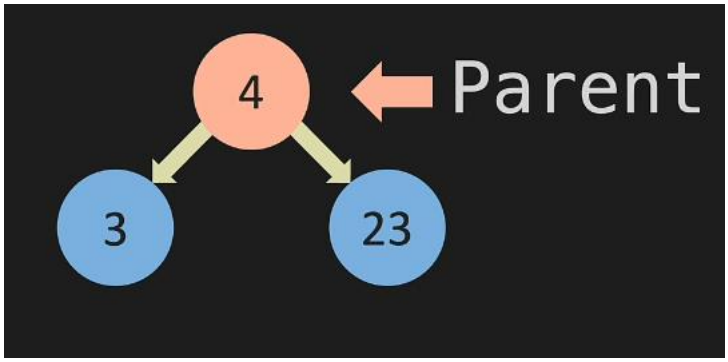


Trees that fork twice are binary trees that we will be focusing on.

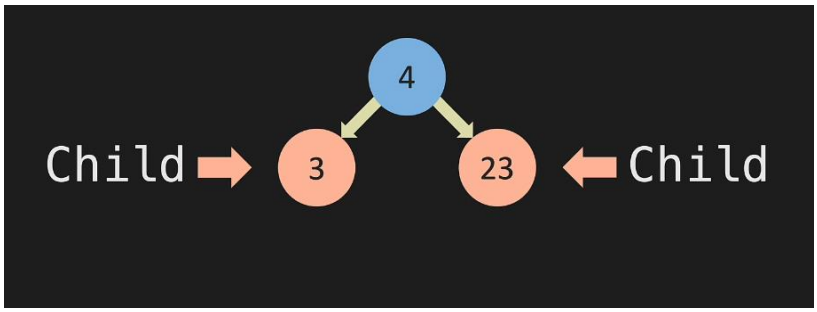


Common terminology general to trees:

Parent Node:

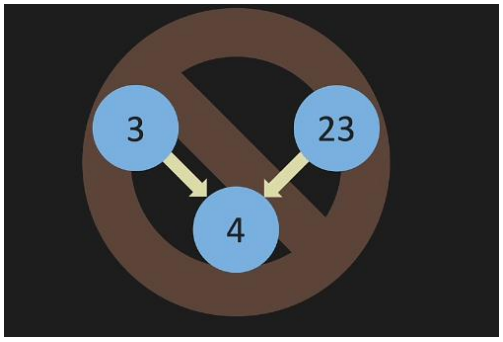


Child nodes:

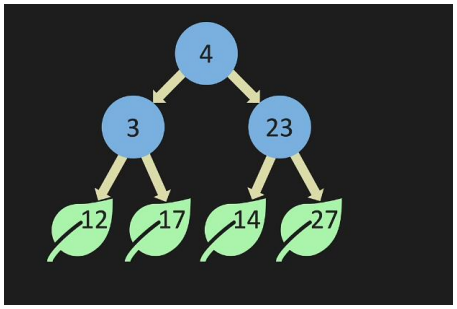


These 2 child nodes are **siblings** as they share the same parent node

Every node can have only one parent node



Nodes with no children are called **leaf** nodes

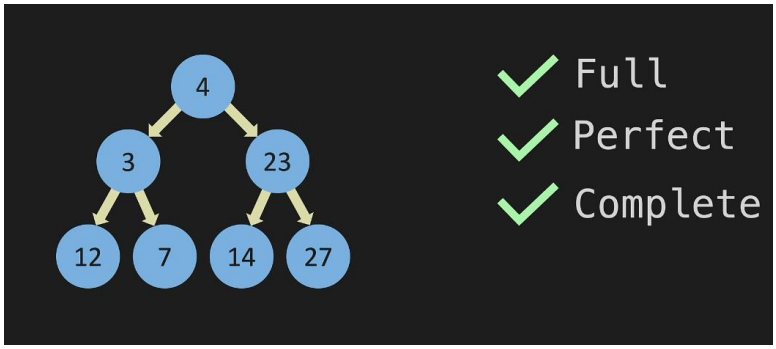


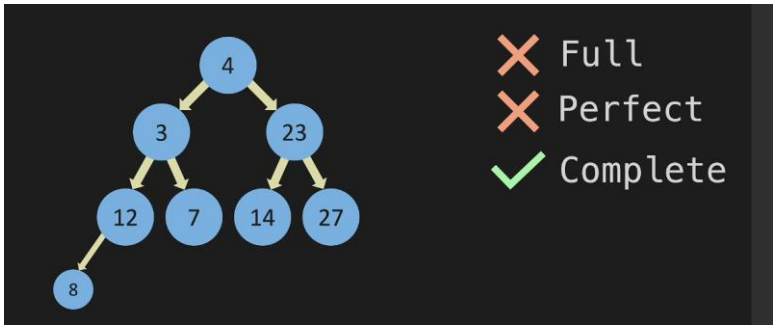
Binary Trees

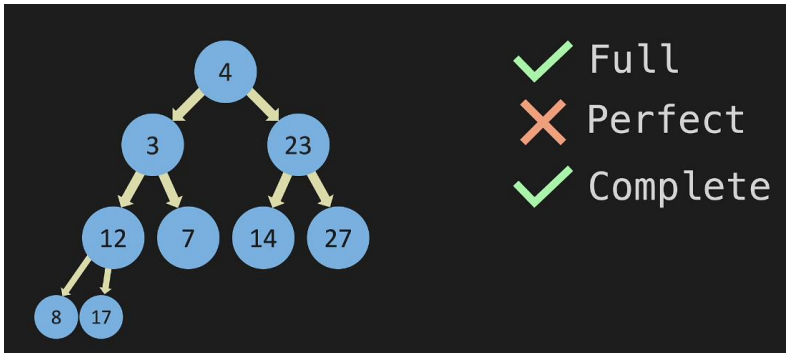
Full Tree – Either has 0 nodes or 2 nodes

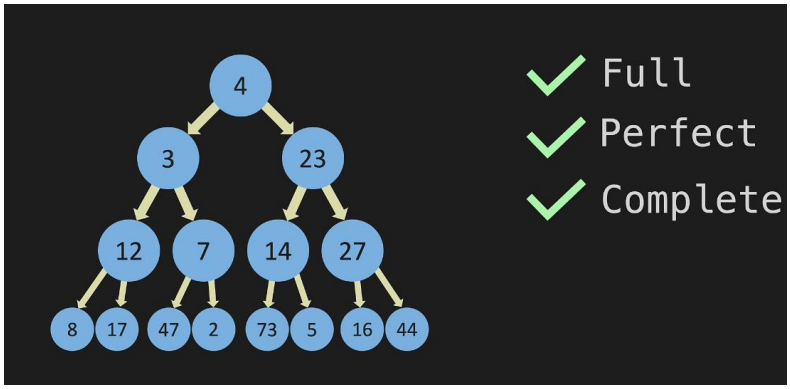
Perfect Tree – The height of the tree has all nodes at that level

Complete tree – Filling the tree from left to right with no gaps





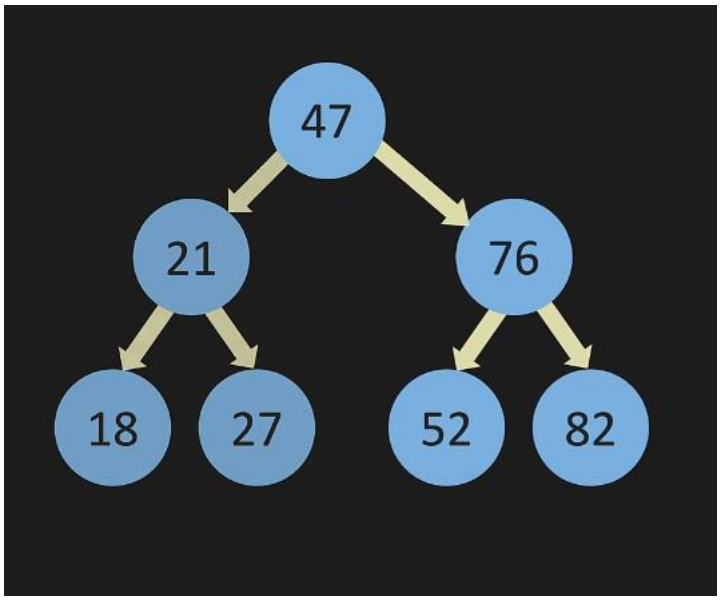




Binary Search Tree (BST):

For any given node,

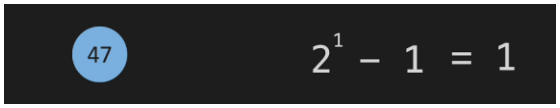
1. the value on it's left with be smaller than the value of the node.
2. the value on it's right will be greater than the value of the node.



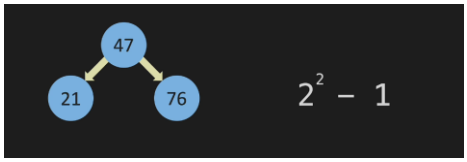
BST Big O

As the level of the tree increases to say n, the total number of nodes is 2^n -1

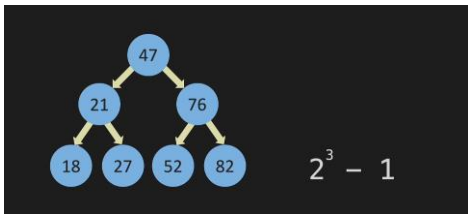
When n = 1



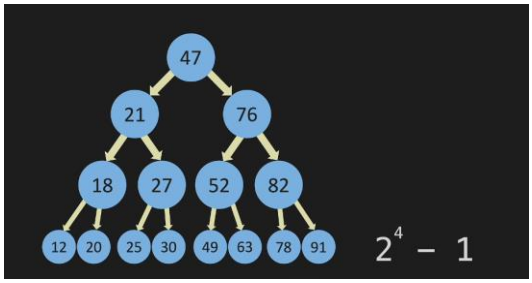
When n = 2



When n = 3

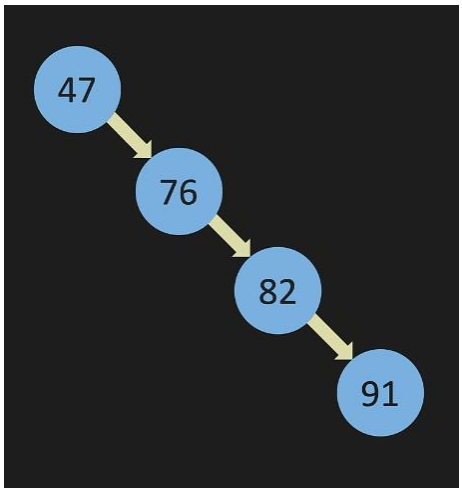


When n = 4

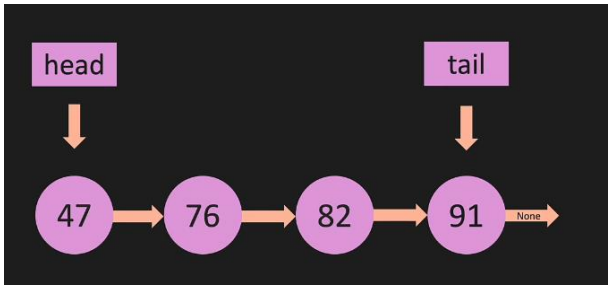


For insert, lookup and remove in the **best and average case** we need to run **log(n) operations**

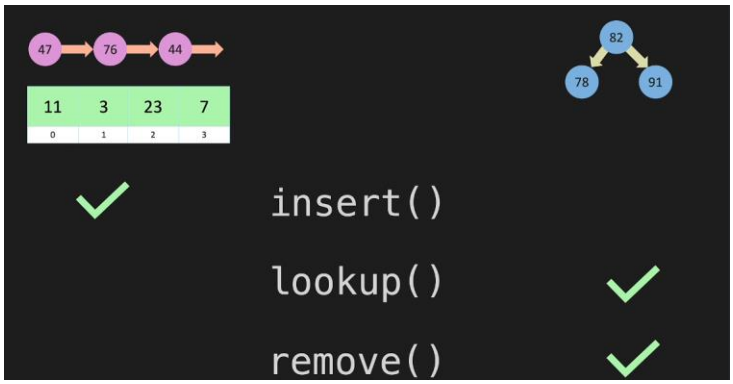
In the worst case, the tree doesn't fork at all



And this is equivalent to Linked List



So, the worst case of tree is same as LL



Operation	BST (Best and Avg case)	LL (Worst case)
Lookup	O (log n)	O (n)
Insert	O (log n)	O (1)
Remove	O (log n)	O (n)

BST- Constructor

```
1. class Node:
2.     def __init__(self, value):
3.         self.value = value
4.         self.left = None
5.         self.right = None
6.
7.
8. class BinarySearchTree:
9.     def __init__(self):
10.        self.root = None
```

The Node Class

This class is used to create nodes, which are the basic units in the tree. Each node will have a value and links to its left and right child nodes.

- **def __init__(self, value):** This is the constructor method that gets called when a new Node object is created.
- **self.value** = value; Stores the value of the node. This value is passed in when you create the node.

- **self.left** = None; Initializes the left child of the node as None. This means it doesn't have a left child yet.
- **self.right** = None; Initializes the right child of the node as None. This means it doesn't have a right child yet.

The BinarySearchTree Class

This class is used to create a Binary Search Tree, which will be made up of Node objects.

- **def __init__(self):**This is the constructor method that gets called when a new BinarySearchTree object is created.
- **self.root** = None; Initializes the root of the tree as None. This means the tree is empty to start with.

When you first create a BinarySearchTree, it won't have any nodes. You'll use methods like insert and contains to add nodes and check for values in the tree.

BST- insert

```
1. def insert(self, value):
2.         new_node = Node(value)
3.         if self.root is None:
4.             self.root = new_node
5.             return True
6.         temp = self.root
7.         while (True):
8.             if new_node.value == temp.value:
9.                 return False
10.            if new_node.value < temp.value:
11.                if temp.left is None:
12.                    temp.left = new_node
13.                    return True
14.                temp = temp.left
15.            else:
16.                if temp.right is None:
17.                    temp.right = new_node
18.                    return True
19.                temp = temp.right
```

Create New Node

The code **new_node = Node(value)** creates a new node with the value you want to insert.

Is the Tree Empty?

The line **if self.root is None:** checks if the tree is empty.

If it's empty, **self.root = new_node** makes the new node the root of the tree.

Start at Root

temp = self.root sets a temporary variable, **temp**, to the root so we can start there.

Loop Until Spot Found

while(True): makes a loop that will keep going until it finds the right spot for the new node.

Duplicate Check

if new_node.value == temp.value: checks for duplicate values.

If a duplicate is found, it exits by returning **False**.

Should We Go Left?

if new_node.value < temp.value: checks if the new value is less than the current node's value.

Insert to the Left

if temp.left is None: checks if the left child spot is empty.

If so, **temp.left = new_node** puts the new node there.

Move Left and Continue

temp = temp.left means, if the left spot isn't empty, move left and continue looking.

Or Should We Go Right?

If the new node's value is greater, the code moves to the **else:** part.

Insert to the Right

if temp.right is None: checks if the right child spot is empty.

If so, **temp.right = new_node** puts the new node there.

Move Right and Continue

temp = temp.right means, if the right spot isn't empty, move right and continue looking.

Omega (best case) and Theta (average case) are both (log n). However, worst case is O(n) and Big O measures worst case. The typically treat Binary Search Trees as O(log n) but technically they are O(n)

BST – contains/lookup

```
1.  def contains(self, value):
2.          temp = self.root
3.          while (temp is not None):
4.              if value < temp.value:
5.                  temp = temp.left
6.              elif value > temp.value:
7.                  temp = temp.right
8.              else:
9.                  return True
10.         return False
```

- **Start at the Root**:
    - **temp = self.root**
    - **temp** starts at the root of the tree.
- **Loop Until End or Value Found**:
    - **while (temp is not None):**
    - Keep checking as long as we have a node to look at.
- **Is Value Less than Current Node?**:
    - **if value < temp.value:**
    - If so, we should go left.
- **Move to Left Child**:
    - **temp = temp.left**
    - We move to the left child of the current node.
- **Is Value Greater than Current Node?**:
    - **elif value > temp.value:**
    - If so, we should go right.
- **Move to Right Child**:
    - **temp = temp.right**
    - We move to the right child of the current node.

- **Value Found**:
    - **else:**
    - If it's not less or greater, it must be equal. We found it!
- **Return True**:
    - **return True**
    - We found the value, so we return **True**.
- **End of Loop - Value Not Found**:
    - If we reach the end of the loop, the value was not found in the tree.
- **Return False**:
    - **return False**
    - We didn't find the value, so we return **False**.

In summary, the **contains** method starts at the root and moves through the tree, going left or right depending on the value. It returns **True** if it finds the value and **False** if it doesn't.

Recursive BST:

contains

```
1.
        def __r_contains(self, current_node, value):
2.          if current_node == None:
3.              return False
4.          if value == current_node.value:
5.              return True
6.          if value < current_node.value:
7.              return self.__r_contains(current_node.left, value)
8.          if value > current_node.value:
9.              return self.__r_contains(current_node.right, value)
10.
11.
12.     def r_contains(self, value):
13.         return self.__r_contains(self.root, value)
```

The **r_contains** method takes a value as input and returns **True** if the value is found in the binary search tree, and **False** otherwise. It does this by calling a private recursive helper method called **__r_contains**.

The **__r_contains** method takes two arguments: a **current_node** representing the node in the binary search tree that is currently being searched, and the **value** that is being searched for. It recursively searches the binary search tree for the **value** starting from the **current_node**. If the **value** is found, it returns **True**. If the **value** is not found and **current_node** is **None**, it returns **False**. If the **value** is not equal to the **current_node** value, it determines whether to search the left or right subtree based on whether the **value** is less than or greater than the **current_node** value, respectively. It then calls itself recursively with the appropriate child node and the **value**.

The **r_contains** method simply calls the **__r_contains** method with the root node of the binary search tree and the **value** that is being searched for, and returns the output of the **__r_contains** method.

Insert

```
1.  def __r_insert(self, current_node, value):
2.      if current_node == None:
3.          return Node(value)
4.      if value < current_node.value:
5.          current_node.left = self.__r_insert(current_node.left, value)
6.      if value > current_node.value:
7.          current_node.right = self.__r_insert(current_node.right, value)
8.      return current_node
9.
10. def r_insert(self, value):
11.     if self.root == None:
12.         self.root = Node(value)
13.     self.__r_insert(self.root, value)
```

The **r_insert** method and its helper method **__r_insert** are used to insert values into a binary search tree (BST) using a recursive approach. Here's an explanation of the provided code:

1. **__r_insert** method: This is a private helper method for the **r_insert** method. It takes two arguments - the current node and the value to be inserted. The method is recursive and works as follows: a. If the current node is **None**, it means the value can be inserted at this position. A new node with the given value is created and returned. b. If the value is less than the current node's value, the method is called recursively on the left child of the current node. c. If the value is greater than the current node's value, the method is called recursively on the right child of the current node. d. Finally, the current node is returned.
2. **r_insert** method: This is the public method to insert a value into the binary search tree. If the root is **None**, it creates a new node with the given value and sets it as the root. Then it calls the **__r_insert** helper method with the root and the value as arguments.

Minimum value – helper function

```
1.  def min_value(self, current_node):
2.      while current_node.left is not None:
3.          current_node = current_node.left
4.      return current_node.value
```

This code defines a method called **min_value** that finds the minimum value in a binary search tree (BST) starting from a given node (**current_node**). The method takes one argument, **current_node**, which is the node from where the search for the minimum value begins.

In a BST, the left subtree contains only nodes with values less than the parent node's value, and the right subtree contains only nodes with values greater than the parent node's value. Therefore, the minimum value in a BST is located in the leftmost node of the tree.

The method works as follows:

1. It starts with a **while** loop that continues as long as **current_node.left** is not **None**. This means that the loop continues as long as there is a left child for the current node.
2. Inside the loop, the method updates the **current_node** to its left child (**current_node.left**). This means that the method keeps traversing the left subtree of the BST to find the leftmost node.
3. Once the loop ends, it means that the **current_node** has no left child, which implies that the current node is the leftmost node in the tree, and thus contains the minimum value.
4. Finally, the method returns the value of the current node, which is the minimum value in the BST starting from the given node.

delete

```
1.  def __delete_node(self, current_node, value):
2.      if current_node == None:
3.          return None
4.      if value < current_node.value:
5.          current_node.left = self.__delete_node(current_node.left, value)
6.      elif value > current_node.value:
7.          current_node.right = self.__delete_node(current_node.right, value)
8.      else:
9.          if current_node.left == None and current_node.right == None:
10.             return None
11.         elif current_node.left == None:
12.             current_node = current_node.right
13.         elif current_node.right == None:
14.             current_node = current_node.left
15.         else:
16.             sub_tree_min = self.min_value(current_node.right)
17.             current_node.value = sub_tree_min
18.             current_node.right = self.__delete_node(current_node.right,
    sub_tree_min)
19.     return current_node
20.
21. def delete_node(self, value):
22.     self.root = self.__delete_node(self.root, value)
```

This code implements a binary search tree (BST), which is a binary tree data structure where each node has at most two child nodes, arranged in a way that the value of the node to the left is less than or equal to the parent node, and the value of the node to the right is greater than or equal to the parent node.

The **delete_node** method is a public method to delete a node with a given value from the BST.

The actual deletion logic is implemented in the private method **__delete_node**.

The logic of the **__delete_node** method is as follows:

1. If the current node is empty (None), it means the node with the given value is not found, so return None.
2. If the value to delete is less than the current node's value, search in the left subtree.
3. If the value to delete is greater than the current node's value, search in the right subtree.
4. If the value to delete is equal to the current node's value, we have found the node to delete. There are three cases: a. If the node has no children, remove the node by returning None. b. If the node has only a left child or only a right child, remove the node by returning the existing child. c. If the node has both children, find the minimum value in the right subtree, replace the current node's value with that minimum value, and then delete the minimum value node in the right subtree.

The reason we use **self.root = self.__delete_node(self.root, value)** instead of **self.__delete_node(self.root, value)** is to update the tree's root node after deletion.

This is necessary if the root node is the one being deleted or if its value is replaced with the minimum value from its right subtree.

By assigning the result of **self.__delete_node(self.root, value)** to **self.root**, we ensure that the root node is updated accordingly, maintaining the integrity of the tree structure.

BST Traversal

BFS: Breadth First Search

It is done row by row

```
1. def BFS(self):
2.         current_node = self.root
3.         queue = []
4.         results = []
5.         queue.append(current_node)
6.
7.         while len(queue) > 0:
8.             current_node = queue.pop(0)
9.             results.append(current_node.value)
10.            if current_node.left is not None:
11.                queue.append(current_node.left)
```

```
            if current_node.right is not None:
                queue.append(current_node.right)
        return results
```

The **BFS** method performs a Breadth-First Search traversal of the binary search tree.

The method starts at the root of the tree and adds it to a **queue** data structure.

Then, the method loops over the elements in the **queue**, removing the first element in the **queue**, appending its value to a **results** list, and adding its left and right child (if they exist) to the **queue**.

The method continues this process until the **queue** is empty, at which point it returns the **results** list containing the values of all nodes in the binary search tree in breadth-first order.

The **BFS** method uses a **queue** to keep track of the nodes to visit in a first-in, first-out (FIFO) order, which is the main characteristic of the BFS algorithm.

The method removes the first element in the **queue** using the **pop(0)** method, which ensures that the next element to be visited is always the one that was added first.

The BFS algorithm traverses the tree level-by-level, visiting all nodes at a particular level before moving on to the next level. The **BFS** method in this implementation achieves this by using a **queue** to keep track of nodes that have been visited, but not processed, in the order they were visited.

DFS – pre-order

```
def dfs_pre_order(self):
        results = []
        def traverse(current_node):
            results.append(current_node.value)
            if current_node.left is not None:
                traverse(current_node.left)
            if current_node.right is not None:
                traverse(current_node.right)
        traverse(self.root)
        return results
```

The **dfs_pre_order** method in the **BinarySearchTree** class performs a Depth-First Search traversal of the binary search tree in pre-order. The pre-order traversal visits the current node first, then the left child, and then the right child.

The method starts by creating an empty list called **results** that will be used to store the values of the visited nodes. Then, the method defines a nested function called **traverse** that takes a **current_node** argument. The **traverse** function appends the value of the **current_node** to the **results** list, and then recursively calls itself to traverse the left child and the right child (if they exist) in pre-order.

The method then calls the **traverse** function with the root node of the tree to start the pre-order traversal. The **traverse** function recursively visits each node in the tree in pre-order, appending the value of each visited node to the **results** list. Finally, the **dfs_pre_order** method returns the **results** list, which contains the values of all nodes in the binary search tree in pre-order.

The DFS algorithm uses recursion to visit each node in the tree. In this implementation, the **traverse** function takes a **current_node** argument and appends the value of the node to the **results** list. Then, the function recursively calls itself to traverse the left and right child of the current node. The recursive calls continue until there are no more nodes to visit, and the **results** list is returned with the values of all visited nodes.

DFS – post-order

```
1.  def dfs_post_order(self):
2.      results = []
3.      def traverse(current_node):
4.          if current_node.left is not None:
5.              traverse(current_node.left)
6.          if current_node.right is not None:
7.              traverse(current_node.right)
8.          results.append(current_node.value)
9.      traverse(self.root)
10.     return results
```

The **dfs_post_order** method in the **BinarySearchTree** class performs a Depth-First Search traversal of the binary search tree post-order. The post-order traversal visits the left child first, then the right child, and finally the current node.

The **dfs_post_order** method starts by creating an empty list called **results** that will be used to store the values of the visited nodes. Then, the method defines a nested function called **traverse** that takes a **current_node** argument. The **traverse** function recursively calls itself to traverse the left child of the **current_node**, then recursively calls itself to traverse the right child of the **current_node**, and finally appends the value of the **current_node** to the **results** list.

The method then calls the **traverse** function with the root node of the tree to start the post-order traversal. The **traverse** function recursively visits each node in the tree post-order, appending the value of each visited node to the **results** list after visiting its children. Finally, the **dfs_post_order** method returns the **results** list, which contains the values of all nodes in the binary search tree post-order.

DFS – In-order

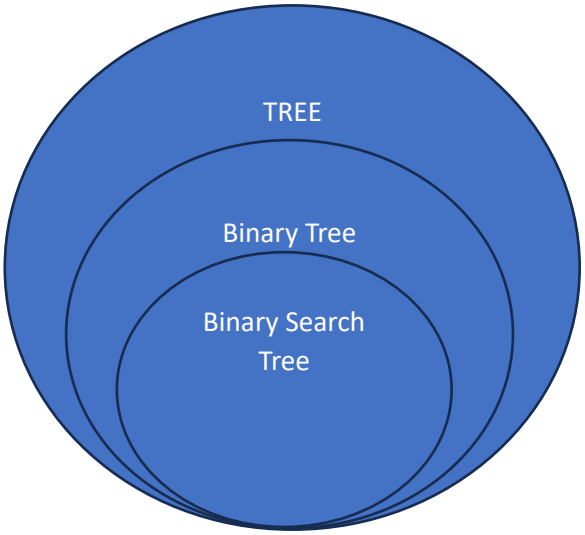
```
1. def dfs_in_order(self):
2.         results = []
3.         def traverse(current_node):
4.             if current_node.left is not None:
5.                 traverse(current_node.left)
6.             results.append(current_node.value)
7.             if current_node.right is not None:
8.                 traverse(current_node.right)
9.         traverse(self.root)
10.        return results
```

The **dfs_in_order** method performs a Depth-First Search traversal of the binary search tree in-order. The in-order traversal visits the left child first, then the current node, and then the right child.

The method starts by creating an empty list called **results** that will be used to store the values of the visited nodes. Then, the method defines a nested function called **traverse** that takes a **current_node** argument. The **traverse** function recursively calls itself to traverse the left child of the **current_node**, appends the value of the **current_node** to the **results** list, and then recursively calls itself to traverse the right child of the **current_node** (if it exists).

The method then calls the **traverse** function with the root node of the tree to start the in-order traversal. The **traverse** function recursively visits each node in the tree in-order, appending the value of each visited node to the **results** list. Finally, the **dfs_in_order** method returns the **results** list, which contains the values of all nodes in the binary search tree in-order.

The DFS algorithm uses recursion to visit each node in the tree. In this implementation, the **traverse** function takes a **current_node** argument and recursively visits the left and right child of the current node in-order. The recursive calls continue until there are no more nodes to visit, and the **results** list is returned with the values of all visited nodes.


TREE
Binary Tree
Binary Search
Tree