Binary Search Tree Insertion (Implementation)

In this chapter, we'll study the implementation of Binary Search Tree insertion in Python through the iterative and recursive approaches.

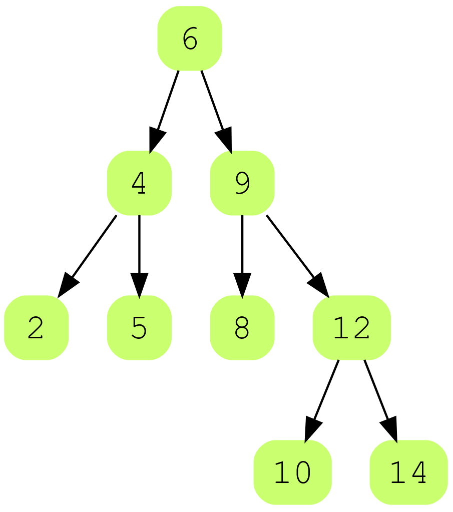
We'll cover the following

Introduction

There are two ways to code a BST insert function

- *Iteratively*
- *Recursively*

We will be implementing *both* in this lesson. We'll end up with the following tree once we implement the BST insert function and insert the elements `[6,4,9,5,2,8,12,10,14]`.



Insert Implementation (Iterative)

main.py

BinarySearchTree.py

Node.py

```python
from Node import Node
from BinarySearchTree import BinarySearchTree
import random


def display(node):
    lines, _, _, _ = _display_aux(node)
    for line in lines:
        print(line)


def _display_aux(node):
    """
    Returns list of strings, width, height,
    and horizontal coordinate of the root.
    """
    # No child.
    if node.rightChild is None and node.leftChild is None:
        line = str(node.val)
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    # Only left child.
    if node.rightChild is None:
        lines, n, p, x = _display_aux(node.leftChild)
        s = str(node.val)
        u = len(s)
        first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
        second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
        shifted_lines = [line + u * ' ' for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, n + u // 2

    # Only right child.
    if node.leftChild is None:
        lines, n, p, x = _display_aux(node.rightChild)
        s = str(node.val)
        u = len(s)
#        first_line = s + x * '_' + (n - x) * ' '
        first_line = s + x * '_' + (n - x) * ' '
        second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
        shifted_lines = [u * ' ' + line for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, u // 2

    # Two children.
    left, n, p, x = _display_aux(node.leftChild)
    right, m, q, y = _display_aux(node.rightChild)
    s = '%s' % node.val
    u = len(s)
    first_line = (x + 1) * ' ' + (n - x - 1) * \
        '_' + s + y * '_' + (m - y) * ' '
    second_line = x * ' ' + '/' + \
        (n - x - 1 + u + y) * ' ' + '\\' + (m - y - 1) * ' '
    if p < q:
        left += [n * ' '] * (q - p)
    elif q < p:
        right += [m * ' '] * (p - q)
    zipped_lines = zip(left, right)
    lines = [first_line, second_line] + \
        [a + u * ' ' + b for a, b in zipped_lines]
    return lines, n + m + u, max(p, q) + 2, n + u // 2


```

Explanation

The `insert(val)` function takes an integer value and if the root exists, it calls the `Node` class's `insert` function on it; otherwise, it makes the root the value to be inserted. The `Node` class's `insert()` function takes care of the meat of the algorithm. Now, while you *can* write an insert function in the `BinarySearchTree` class itself, it's better to write it as part of the `Node` class as it *usually* makes the code cleaner and easier to maintain.

The `insert(val)` function starts from the root of the tree and moves on to the left or right sub-tree depending on whether the value to be inserted is less than or greater than or equal to the node. While traversing, it stores the parent node of each current node and when the current node becomes empty or `None`, it inserts the value there. It does so by making the value to be inserted the current node (which is `None`)'s parent's child. In other words, the value is inserted in place of the current node.

Insert Implementation (Recursive)

main.py

BinarySearchTree.py

Node.py

```
from Node import Node
from BinarySearchTree import BinarySearchTree

import random


def display(node):
    lines, _, _, _ = _display_aux(node)
    for line in lines:
        print(line)


def _display_aux(node):
    """
    Returns list of strings, width, height,
    and horizontal coordinate of the root.
    """
    # No child.
    if node.rightChild is None and node.leftChild is None:
        line = '%s' % node.val
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    # Only left child.
    if node.rightChild is None:
        lines, n, p, x = _display_aux(node.leftChild)
        s = '%s' % node.val
        u = len(s)
        first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
        second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
        shifted_lines = [line + u * ' ' for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, n + u // 2

    # Only right child.
    if node.leftChild is None:
        lines, n, p, x = _display_aux(node.rightChild)
        s = '%s' % node.val
        u = len(s)
        first_line = s + x * '_' + (n - x) * ' '
        second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
        shifted_lines = [u * ' ' + line for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, u // 2

    # Two children.
    left, n, p, x = _display_aux(node.leftChild)
    right, m, q, y = _display_aux(node.rightChild)
    s = '%s' % node.val
    u = len(s)
    first_line = (x + 1) * ' ' + (n - x - 1) * \
        '_' + s + y * '_' + (m - y) * ' '
    second_line = x * ' ' + '/' + \
        (n - x - 1 + u + y) * ' ' + '\\' + (m - y - 1) * ' '
    if p < q:
        left += [n * ' '] * (q - p)
    elif q < p:
        right += [m * ' '] * (p - q)
    zipped_lines = zip(left, right)
    lines = [first_line, second_line] + \
        [a + u * ' ' + b for a, b in zipped_lines]
    return lines, n + m + u, max(p, q) + 2, n + u // 2


BST = BinarySearchTree(50)
for _ in range(15):
    ele = random.randint(0, 100)
    print("Inserting "+str(ele)+":")
    BST.insert(ele)
    # We have hidden the code for this function but it is available
    display(BST.root)
    print('\n')

```

Explanation

This implementation starts with the root and calls the insert function on its left child if `val` is less than the value at the root, and calls it on its right child if `val` is greater than or equal to the value at the root. This continues until a `None` node is reached. At that point, a new node is created and returned to the previous leaf node, which is now the parent of the new node that we just created.

In the next lesson, we will see how BST search works and we'll also implement it in Python.