Searching in a Binary Search Tree (Implementation)

This lesson about Searching in Binary Search Tree and how to implement searching functionality in Python.

We'll cover the following

Introduction

In this lesson, we'll implement a search function for binary search trees which will return a node from the tree if the value to be searched matches it. We'll again, implement both an iterative and a recursive solution. Here is a high-level description of the algorithm:

1. Set the 'current node' equal to root.
2. If the value is less than the 'current node's' value, then move on to the left-subtree otherwise move on to the right sub-tree
3. Repeat until the value at the 'current node' is equal to the value searched or it becomes `None`.

4. Return the current node

Iterative Search Implementation

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 = 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)
```

Explanation

In this implementation, the core of the search function is implemented in the `Node` class. The `BinarySearchTree` first checks if `root` is `None`, if so, it returns `False`, otherwise, it calls the `Node` class's `search()` function on the root.

The search function sets `current` to `self` and goes into a while loop which traverses the tree comparing `val` to the values of the left and right child nodes. If `val` is less than the value at the current node, we move on to the left subtree and if it is greater, we move on to the right subtree until we reach a leaf node or the value being searched for.

Recursive Search Implementation

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 = 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


BST = BinarySearchTree(50)
for _ in range(15):
    ele = random.randint(0, 100)
    BST.insert(ele)
```

```
# We have hidden the code for this function but it is available for use!
display(BST.root)
print('\n')

print(BST.search(15))
print(BST.search(50))
```

Explanation

In this implementation, the main part of the function is in the `Node` class. The recursive base-case is if the given node is equal to the one being searched, return `True`. If the base cases are not true, the function checks if the value being searched for is less than or equal to the value of the given node. It moves on to the right or left child accordingly and calls the `search` function on them if they are not `None`, if they are, it returns `False`. If the entire tree has been traversed, it returns `False`.

In the next couple of lessons, we will study the binary search tree deletion function and will also implement it in Python.

Back

Binary Search Tree Insertion (Impleme...

Next

Deletion in a Binary Search Tree

Comple