

Binary Search Trees

By: Daniel Nkunga

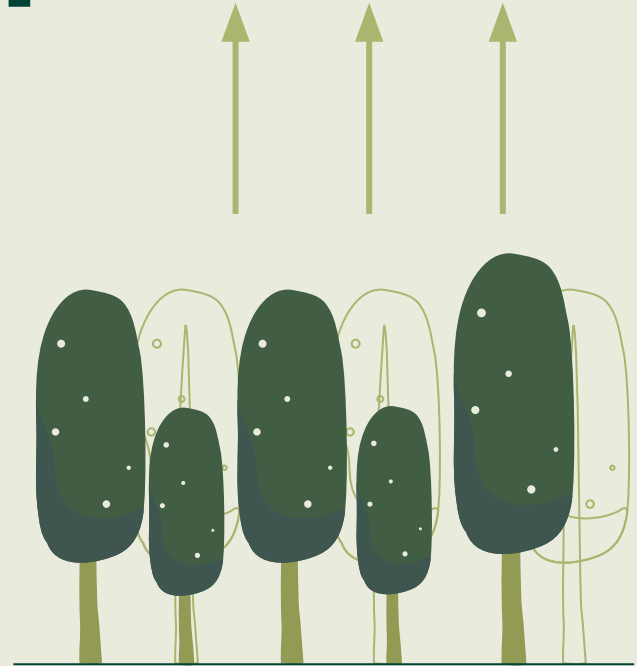


Table of contents



01 Binary Search Trees

Their basics of a BST and my approach to their creation

02 Interview Questions

Four more advanced questions we were challenged to create



Binary Search Tree Basics

Binary Search Tree Basics

A **binary search tree (BST)** is a hierarchical data structure used for organizing keys in a sorted manner. In a BST, each node has at most two **child nodes**: a left child and a right child. The key in each node is greater than all keys in its left subtree and less than all keys in its right subtree.

Common Uses: Efficient Searching, Insertion, and Deletion; Dictionaries, Maintaining Ordered Data



Implementation



```
class TreeNode:

    def __init__(self, value):
        self.left = None
        self.right = None
        self.value = value

    def insert(self, value):
        if value < self.value:
            if self.left is None:
                self.left = TreeNode(value)
            else:
                self.left.insert(value)
        else:
            if self.right is None:
                self.right = TreeNode(value)
            else:
                self.right.insert(value)
```

My BST is implemented under a class called **TreeNode**


- **Initializes** with a value given by the user, and left and right set to None
- **Insert function** properly creates a BST by placing elements smaller to each node to the left of the node and each node larger to the right
 - Nodes that have the same value as their **parent node** are placed to the right
- **Find function** traverses tree following the same process as insertion
 - Complexity of $O(\log n)$ when balanced

```
def inorder_traversal(self):
    if self.left:
        self.left.inorder_traversal()
    print(self.value)
    if self.right:
        self.right.inorder_traversal()

def preorder_traversal(self):
    print(self.value)
    if self.left:
        self.left.preorder_traversal()
    if self.right:
        self.right.preorder_traversal()

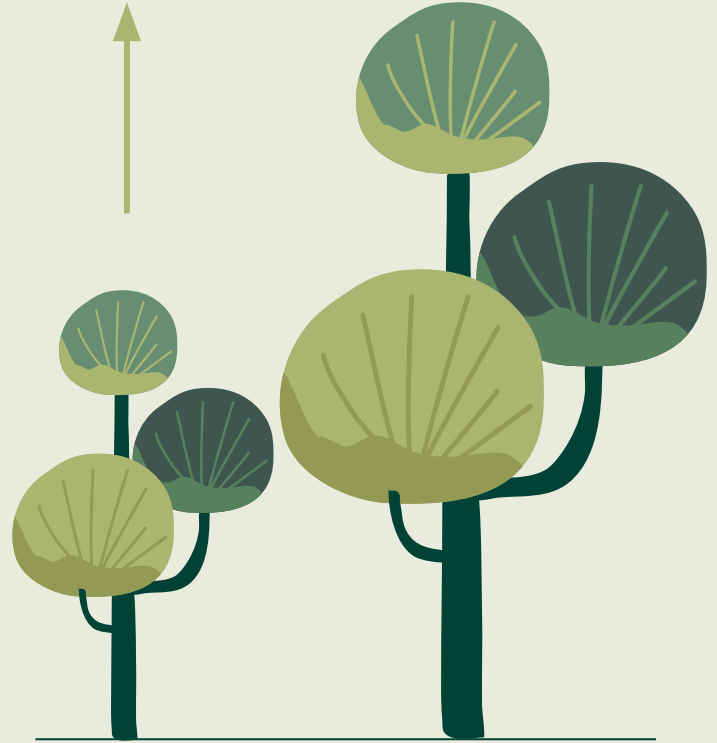
def postorder_traversal(self):
    if self.left:
        self.left.postorder_traversal()
    if self.right:
        self.right.postorder_traversal()
    print(self.value)
```

Implementation (Cont.)



- The trees can be **displayed** in three separate ways:
 - **Inorder Traversal** prints the nodes in ascending order
 - **Preorder Traversal** prints the nodes as they appear always going left until there are no more left child, then going one node up then right
 - Most commonly used in testing
 - **Postorder Traversal** prints the nodes from the bottom most left note of the tree then goes up to the parent node, to the right node, and prints the lowest element there, following this pattern

Interview Questions



Interview Questions



01 Deletion

Write a function to delete a given node from a BST while maintaining its BST properties.

02 In Order Successor

Write a function to find the in-order successor of a given node in a BST. The in-order successor is the node with the next higher key value in the in-order traversal of the BST.

03 Array to Tree

Given a sorted array, write a function to convert it into a height-balanced BST.

04 Tree Rotation

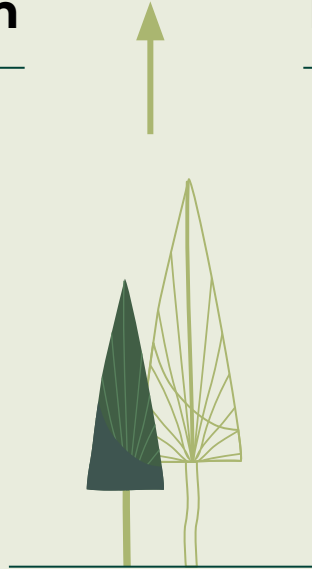
Give a node, write a function that can either rotate it right or left.

1. Deletion

Problem

Deleting a node off of a tree comes down from two main problems:

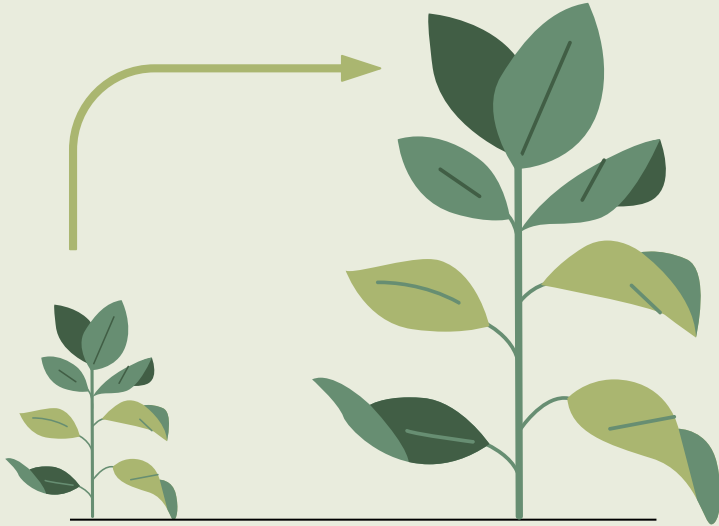
- Removing a node from the tree can leave its children without a parent
- Improperly removed nodes will leave floating trees that will get garbage collected



Implementation

- Create `get_minimum_value_node` function which will check to see if the left child exist
- Recursively create the `delete_node` function to call itself when a node is `None`
- Set nodes to `None` if their children (valuing left children more) are `None`

1. Deletion: Code Snippet



```
def get_min_value_node(self, node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def delete_node(self, value):
    if self is None:
        return self

    if value < self.value:
        self.left = self.left.delete_node(value)
    elif value > self.value:
        self.right = self.right.delete_node(value)
    else:
        if self.left is None:
            temp = self.right
            self = None
            return temp
        elif self.right is None:
            temp = self.left
            self = None
            return temp
        temp = self.get_min_value_node(self.right)
        self.value = temp.value
        self.right = self.right.delete_node(temp.value)
    return self
```

2. In Order Successor

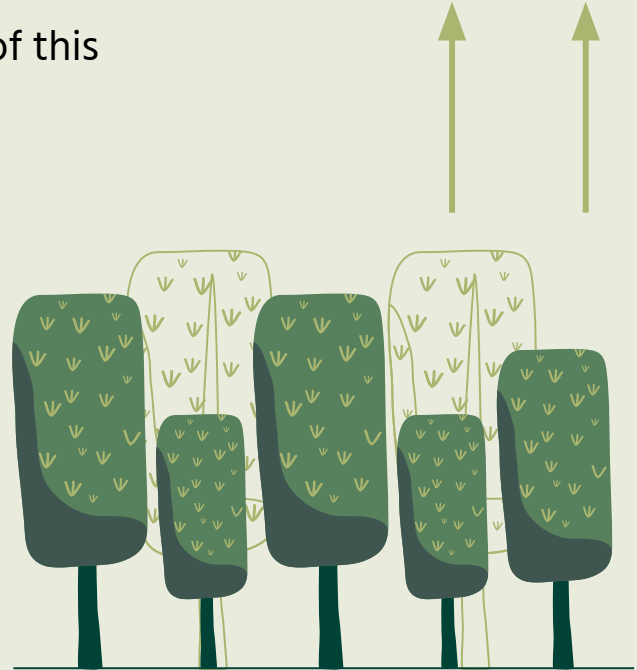
Problem

Tree traversal is complicated and poor implementations of this function will lead to complexities up to $O(n)$

Implementation

The function will remember the current best successor for the node and discard all parts of the tree that can no longer contain that successor:

- If the current node is less than or equal to the target, the function will continue exploring that tree
- If the current node is greater than the best, the tree will be discarded



2. In Order Successor: Code Snippet

```
def in_order_successor(self, target):  
    if self.value <= target:  
        if self.right:  
            return self.right.in_order_successor(target)  
        else:  
            return None  
    else:  
        left_successor = None  
        if self.left:  
            left_successor =  
self.left.in_order_successor(target)  
        return left_successor if left_successor else self
```



3. Array to Tree

Problem

There are many ways to change an array to a tree but many of them will be inefficient in the amount of passes they take or they will create an imbalance tree

- Adding the elements in the order they appear will result in a tree with one really long leg



3. Array to Tree (cont.)

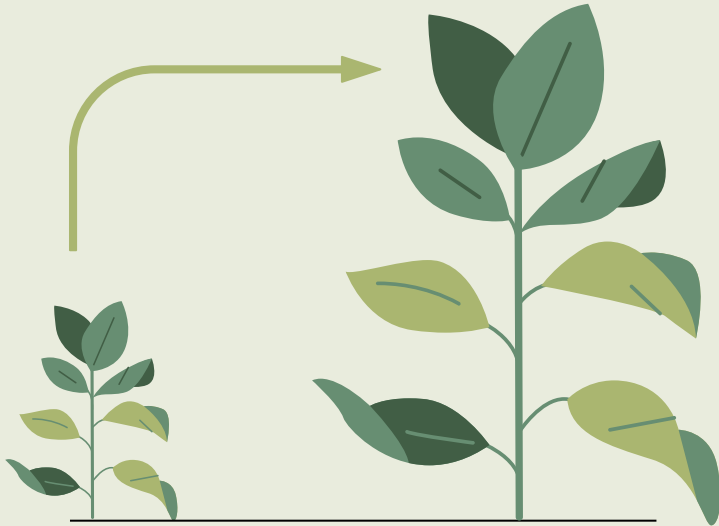
Implementation

Approach is based on the fact that for a balanced tree, the parent node should come from the middle of a sorted array

- ***This means a function that just adds all elements to on either side of the middle element will also be balanced, just really long
- A function will be called recursively that find the middle element of an array and inserts that into the tree
- Each consecutive next pass will explore either to the rest of the array to either the left or the right



3. Array to Tree: Code Snippet



```
@staticmethod
def array_to_tree_helper(array, start, end):
    if start > end:
        return None
    mid = (start + end) // 2
    root = TreeNode(array[mid])
    root.left = TreeNode.array_to_tree_helper(array, start, mid - 1)
    root.right = TreeNode.array_to_tree_helper(array, mid + 1, end)
    return root

@staticmethod
def array_to_tree(array):
    if not array:
        return None
    return TreeNode.array_to_tree_helper(array, 0, len(array) - 1)
```

4. Tree Rotation

Problem

Tree rotation can be a memory nightmare similar to deletion where massive parts of the tree can just be deleted if not careful

Implementation

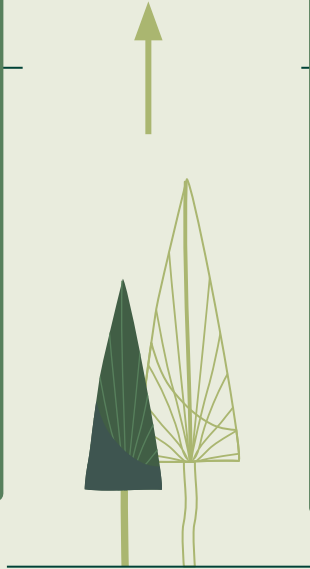
Only the root and side wanting to be switched need to be fully altered, the rest of the trees can be rotated as subtrees [this explanation will be for rotate left]:

- Set the root node to root.right, set the root.left to be the original root node
- Add back subtrees under their new positions



4. Rotation: Code Snippet

```
def rotate_left(self):  
    if not self.right:  
        return  
    a = self.value  
    b = self.right.value  
    A = self.left  
    B = self.right.left  
    C = self.right.right  
    self.value = b  
    self.left = TreeNode(a)  
    self.left.left = A  
    self.left.right = B  
    self.right = C
```



```
def rotate_right(self):  
    if not self.left:  
        return  
    a = self.value  
    b = self.left.value  
    A = self.left.left  
    B = self.left.right  
    C = self.right  
    self.value = b  
    self.right = TreeNode(a)  
    self.right.left = B  
    self.right.right = C  
    self.left = A
```

03

Future Implementations



Future Implementations



Visualization

Currently all models are being tested using preorder traversal and USECA's visualizer. Implementing my own visualizer would really help with the debugging process

Currently looking into:
HTML/JavaScript,

Height Function

Knowing the height of the tree is imperative for the implementation of the other functions. It's implementation also needs to be the most cost efficient because it will be called often in other functions.

Self Balancing

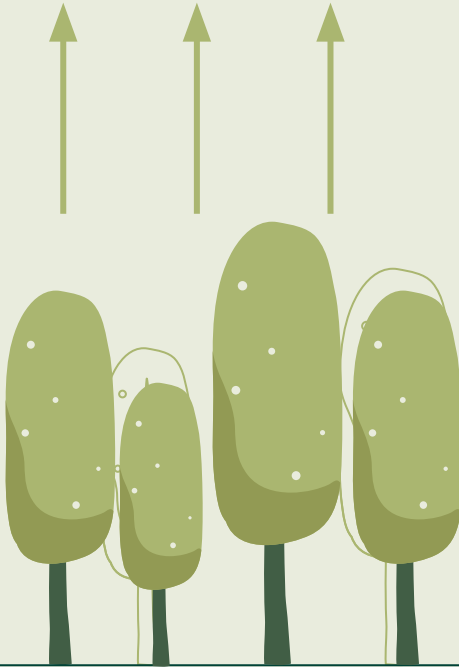
Knowing the height and being able to rotate a tree allows for the creation of self balancing trees. It useful for bigger databases when using BSTs as a actual database backbone but now its more a test in recursion to its extreme.

Thanks!

Do you have any questions?

daniel.s.nkunga@gmail.com

+1 501 237 2428



CREDITS: This presentation template was created by [Slidesgo](#), including icons by [Flaticon](#), and infographics & images by [Freepik](#)

