The Self-Balancing Binary Search Tree

Jayden Patel

The Assignment

Create a self-balancing Binary Search Tree with the following features:

- Add
- Delete
- Find

Then, benchmark the add function:

- Graph n vs. time (3 data sets: yours, another TSBBST, and a non-balancing BST)
- Graph n vs. depth (3 data sets: yours, a non-balancing BST, and ideal depth)

Also benchmark the delete function:

- Add a lot of items
- Remove them one by one at random
- Graph the remaining size vs. depth

Helper Functions

- The first function simply returns the difference between the height of the current node’s children.
- The second function returns the minimum value node in a tree (or sub tree) starting at the given node.
- The third function updates the current node’s height using its children’s height.

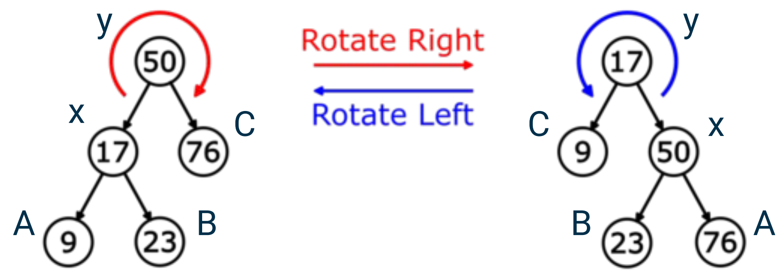
```python
def balance_factor(self):
    return (self.left.height if self.left else 0) -
           (self.right.height if self.right else 0)
```

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

```python
def update_height(self):
    self.height = 0
    left_height = self.left.height if self.left else 0
    right_height = self.right.height if self.right else 0
    self.height = 1 + max(left_height, right_height)
```

Rotating Nodes

- In a BST, rotations are performed on unbalanced branches.
- Capital letters represent subtrees, while lowercase letters represent values.
- In a right rotation, the current node's value is replaced with the value to the left. Then, a new node for the old value is added to the right.
- The sub trees are placed back in order:
  - A: right of the new node
  - B: left of the new node
  - C: left of the new root node

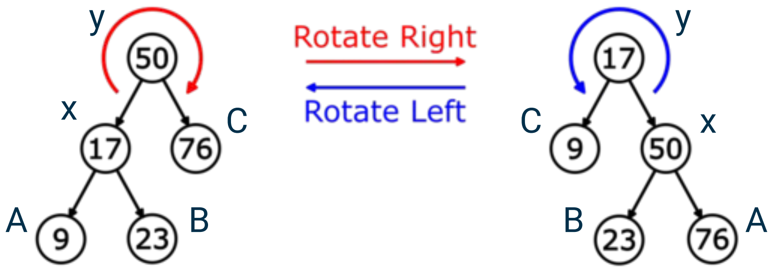


```
def rotate_right(self):
    if self.left:
        x = self.left.value
        y = self.value
        A = self.left.left
        B = self.left.right
        C = self.right

        self.value = x
        self.right = Node(y)
        self.left = A
        self.right.left = B
        self.right.right = C
        self.right.update_height()
        self.update_height()
    else: raise Exception("invalid rotation")
```

Rotating Nodes cont.

- Rotations in BSTs are symmetric.
- This means that to rotate left, the code is the same except every direction is swapped.



```python
def rotate_left(self):
    if self.right:
        x = self.right.value
        y = self.value
        A = self.right.right
        B = self.right.left
        C = self.left

        self.value = x
        self.left = Node(y)
        self.right = A
        self.left.right = B
        self.left.left = C
        self.left.update_height()
        self.update_height()
    else: raise Exception("invalid rotation")
```

Balancing

- When the balance function is called, it balances the subtree starting at the current node.
- Using the balance factor function, it rotates the subtree to restore balance.
    - A **positive** balance factor means that the tree has too many nodes on the **left** (requires right rotation).
    - A **negative** balance factor means that the tree has too many nodes on the **right** (requires left rotation).

```
def balance_tree(self):
    if self.balance_factor() > 1:
        if self.left.balance_factor() < 0:
            self.left.rotate_left()
        self.rotate_right()
    elif self.balance_factor() < -1:
        if self.right.balance_factor() > 0:
            self.right.rotate_right()
        self.rotate_left()
```

Balancing cont.

- In both the add and delete functions, the tree is balanced at the end of the operation.
- Because both functions recursively work down the tree as they add (or delete) nodes, the whole tree is balanced from the bottom up once they are done.

```
def add(self, value):
    ...
    self.balance_tree()
```

```
def delete(self, next, key):
    ...
    next.balance_tree()
```

Finding Nodes

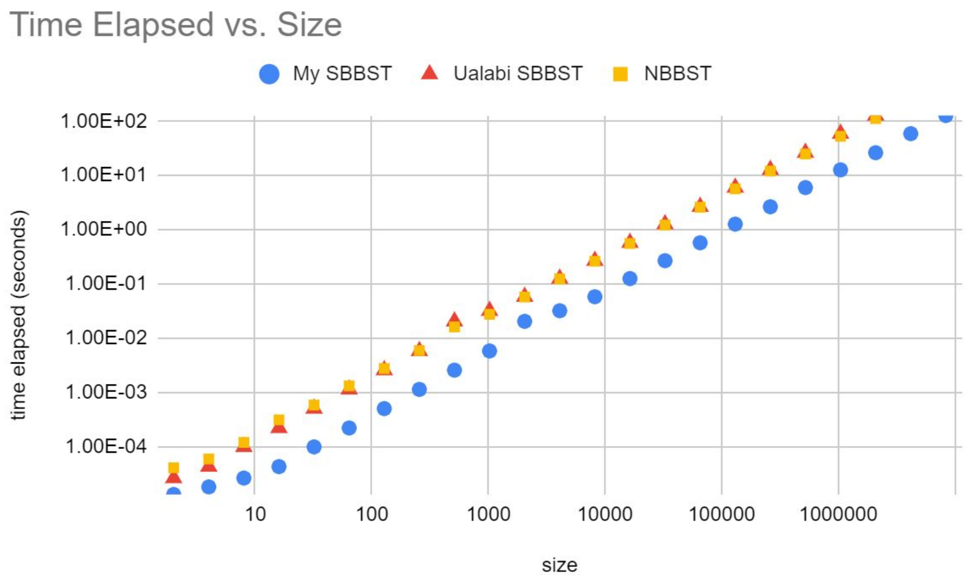
- The tree uses the same binary search logic to find the node containing a given value.
- If the tree does not find the value in the spot it must be, it returns false.

```
def contains(self, value):
    if self.value == value: return True
    if value < self.value and self.left:
        return self.left.contains(value)
    if value > self.value and self.right:
        return self.right.contains(value)
    return False
```

Add Time Benchmark

Graph: the time elapsed to add *n* items to the BST vs. *n*

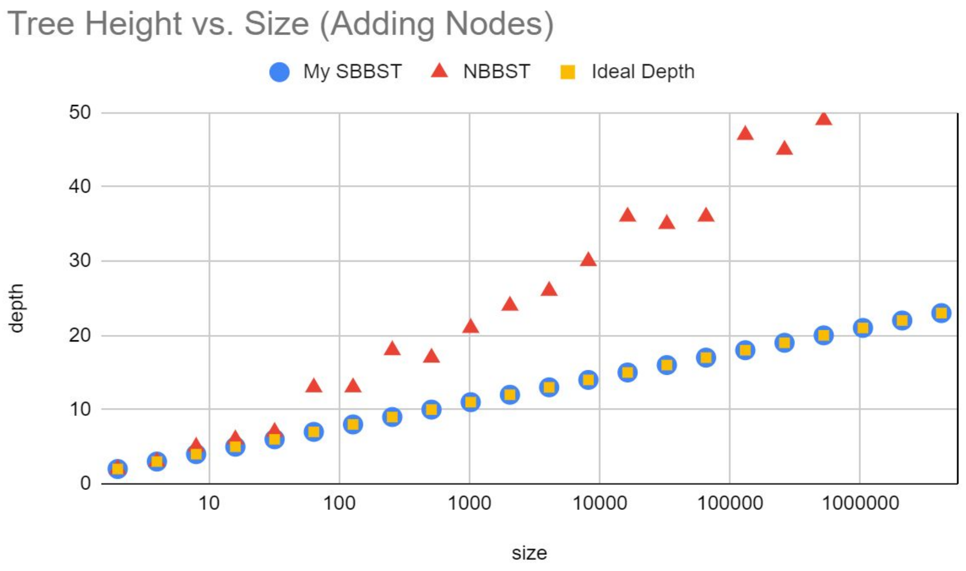
- Blue: My self-balancing BST
- Red: Self-balancing BST made by Ualabi (GitHub Repo)
- Yellow: My BST with no balancing



Add Depth Benchmark

Graph: the height of the tree when *n* nodes are added vs. *n*

- Blue: My self-balancing BST
- Red: My BST with no balancing
- Yellow: The ideal depth line
    - For a completely full and perfect tree of depth n, the number of nodes equals $2^n - 1$.

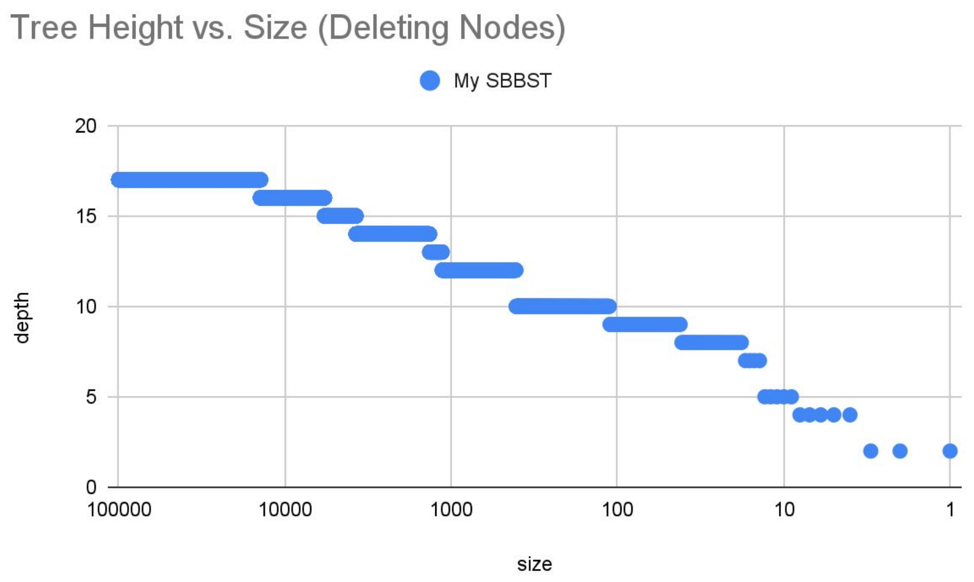


Delete Depth Benchmark

The tree is randomly walked down until the program randomly decides to delete. If the program reaches a leaf, it stops and deletes.

Graph: the height of the tree vs. *n* as items are deleted

```
while tree.root:
    node = tree.root
    while random.random() > 0.25:
        direction = random.random() < 0.5
        if direction and node.left: node = node.left
        elif node.right: node = node.right
        else: break
    tree.delete(node.value)
```



The Self-Balancing Binary Search Tree

Jayden Patel