Pseudocode

insert(data)

```
cur = root
if root = null:
    root = new Node(data)
    return

else:
    while cur != null:
        if data < cur.item:
            if cur.left = null:
                cur.left = new Node(data)
                return
            else:
                cur = cur.left
        else if data > cur.item:
            if cur.right = null:
                cur.right = new Node(data)
                return
            else:
                cur = cur.right
        else:
            return
```

tree was empty

go left if input is less than current node

go right if input is greater than current node

remove(item)

```
cur = root
parent = root
if root = null:
    return

else:
    while true:
        if item < cur.item
            if cur.left = null:
                return
            else:
                parent = cur
                cur = cur.left
        else if item > cur.item
            if cur.right = null:
                return
            else:
                parent = cur
                cur = cur.right

        else:
            if (cur.left = null & cur.right = null)
                if (root.left = null & root.right = null):
                    root = null
                    return
                else:
```

Searching for value to delete

⇒ if it's in a leaf

> checking if it's root

```
        if parent.left != null:
            if(parent.left.item = item):
                parent.left = null
                return
        if parent.right != null:
            if (parent.right.item = item):
                parent.right = null
                return
```

checking whether if it's left child or right child then deleting accordingly

```
    else if cur.left = null & cur.right != null:
```

→ it has right child but not left child

```
        subtreeroot = cur
        if cur.right.left != null:
            cur = cur.right
            while cur.left.left != null:
                cur = cur.left
            subtreeroot.item = cur.left.item
            if cur.left.right != null:
                cur.left = cur.left.right
                return
            else:
                cur.left = null
                return
        else:
            cur.item = cur.right.item
            cur.right = cur.right.right
            return
```

right subtree's root has left child

Swaps the data of node to delete with the least value in the right subtree

if node with least value in the right subtree has right child

if right subtree's root has no left child

```
    else:
        subtreeroot = cur
        if cur.left.right != null:
            cur = cur.left
            while cur.right.right != null:
                cur = cur.right
            subtreeroot.item = cur.right.item
            if cur.right.left != null:
                cur.right = cur.right.left
                return
            else:
                cur.right = null
                return
        else:
            cur.item = cur.left.item
            cur.left = cur.left.left
            return
```

Same with else if condition for the cases where node to delete has either both children or has left child & no right child.

compareTo(node) ⇒ I implemented it in Node class

```
if this = null & node = null:
    return true
if(this.left = null && this.right = null) && (node.left = null && node.right = null):
    if this.item = node.item:
        return true
    else:
        return false
else if(this.left = null && this.right != null) && (node.left = null && node.right != null):
    if this.item = node.item:
        return this.right.compareTo(node.right)
    else:
        return false
```

base case & if both trees are empty

if both nodes have no child, only compare the data

only have right children if data are equal, compare right children

```
else if(this.left!=null && this.right=null) && (node.left!=null && node.right=null):
    if this.item= node.item:
        return this.left.compareTo(node.left)
    else:
        return false
else if(this.left!=null && this.right!=null) && (node.left!=null && node.right!=null):
    if this.item= node.item:
        return this.left.compareTo(node.left) && this.right.compareTo(node.right)
    else:
        return false
else:
    return false
```

same case with the above for left children

if have both children, compare both children if data are equal

isBalanced(node)

```
if this=null || (this.left=null & this.right=null):
    return true
else if (this.left=null & this.right!=null):
    if this.right.getHeight>1:
        return false
    else:
        return true
else if (this.left!=null & this.right=null):
    if this.left.getHeight>1:
        return false
    else:
        return true
else:
    if (abs(this.left.getHeight - this.right.getHeight) <= 1):
        return this.left.isBalanced & this.right.isBalanced
    else:
        return false
```

if root is null or node has no child

if node has only right child, check right child's height

same with above for left child

if node has both children, check whether both children are balanced

isBalanced(BST)

```
root.findHeight()
return this.root.isBalanced()
```

set heights of each node then call node implementation of isBalanced

findHeight(Node)

```
if (this.left=null & this.right=null):
    this.height=1
else if (this.left!=null & this.right=null):
    this.height=1+this.left.findHeight()
else if (this.left=null & this.right!=null):
    this.height=1+this.right.findHeight()
else:
    this.height=1+max(this.left.findHeight(), this.right.findHeight())
return this.height
```

if node has no child

if has only left child, 1+height of left

same with above for right child

if has both children 1+ max of their heights

Complexity Analysis

insert

First, I do a search in the BST to find the location that I should insert and it's time complexity is always O(height). In the worst case, height is n therefore search's complexity is O(n). Inserting is done in constant operations, therefore time complexity is O(n).

Since function is not recursive, there is maximum of 1 function call in the stack, therefore space complexity is O(1).

remove

First, I do a search in the BST similar to insert function and search complexity is again O(n). When I find the correct node to delete there are constant operations of swapping data & deleting nodes etc. therefore time complexity is O(n).

Since function is not recursive, there is maximum of 1 function call in the stack, therefore space complexity is O(1).

compareTo(node)

In this function, I simply check whether each node's left child, right child and item are equal to each other or not and I do it recursive until I reach leaves of the tree. Therefore we have $c * n$ operations in total. Therefore, time complexity is O(n).

In the worst case before compareTo calls for leaf nodes returned, there are n compareTo calls in the runtime stack. Therefore, space complexity is O(n).

isBalanced(Node)

In the worst case (BST is balanced), for each non-terminal node, we check the absolute value of height difference of its children and it's less than or equal to 1, we call isBalanced for both children. There are approximately $\frac{n}{2}$ nonterminal nodes therefore $c * \frac{n}{2}$ operations. Therefore, time complexity is O(n).

Before we return isBalanced calls for leaf nodes, there are approximately n isBalanced calls in runtime stack therefore space complexity is $O(n)$.

findHeight

In this recursive function, I simply set leaf nodes' heights to 1 and all other nodes' heights to maximum of its children's heights. There are approximately $\frac{n}{2}$ terminal nodes whose height are set to 1. ($c_1 \times \frac{n}{2}$ operations). For each nonterminal node (approximately $\frac{n}{2}$ again) I do $1 + \max(\text{height left}, \text{height right})$ so its $c_2 \times \frac{n}{2}$. Total number of operations are $c_3 \times n$, therefore time complexity is $O(n)$.

Before we return findHeight calls for leaf nodes, there are approximately n findHeight calls in runtime stack therefore space complexity is $O(n)$.

isBalanced (BST)

findHeight has $O(n)$ time and space complexity, isBalanced(Node) has $O(n)$ time and space complexity. Therefore, this function does also have $O(n)$ time and space complexity because total # of ops. are $c_1 \times n$ and maximum # of function calls present in the stack is also n.

I have completed this assignment individually, without support from anyone else. I hereby accept that only the below listed sources are approved to be used during this assignment:

(i) Course textbook,

(ii) All material that is made available to me by the professor (e.g., via Blackboard for this course, course website, email from professor / TA),

(iii) Notes taken by me during lectures.

I have not used, accessed or taken any unpermitted information from any other source. Hence, all effort belongs to me.

Umur Berkay Karakaş