Ahmed Moustafa Youssef El-Naggar (16)
Hisham Osama AbdelSalam Ghazy (81)

Alexandria University
Faculty of Engineering
Computer and Systems Department

Problem Statement:

This lab assignment focuses on self-balancing binary search trees. It is required to implement AVL trees and use them to implement dictionary of strings.

Algorithms:

1 – **Rotations:**

a) **Single Rotation:** Single rotation is used to fix the insertion in the left sub tree of the left sub tree or the insertion in the right sub tree of the right sub tree of a given node. It does this by rotating the node with one of its children updating the sufficient nodes' references.
b) **Double Rotation:** Double rotation is used to fix the insertion in the left sub tree of the right sub tree or the insertion in the right sub tree of the left sub tree of a given node. It does this by rotating the node with one of its children.

2- Insertion: Insertion is used to add a new node to the structure by comparing its key to the root's key and placing it in it proper place in the last level ensuring the search property of the tree is unaffected and performing the rebalancing process.

3- Searching: Searching is used to check the existence of a certain key by traversing the tree by making comparisons along the path traversed.

4- Deletion: Deletion is used to delete a node if it does exist and search for its successor and places it in its place and then performs the rebalancing process.

Data Structures:

Node: the data structure that represents the element in a tree.

Binary Search Tree: the binary search tree to represent the AVL Tree.

Code Snippets:

```
 * Inserts the node and returns the new root of the tree.
private INode<T> insert(T key, INode<T> node) {
    if (node == null) {
        return new Node<T>(key);
    }
    if (node.getValue().compareTo(key) < 0) {
        node.setRightChild(insert(key, node.getRightChild()));
    } else if (node.getValue().compareTo(key) > 0) {
        node.setLeftChild(insert(key, node.getLeftChild()));
    }

    return balance(node);
}
```


```
private INode<T> balance(INode<T> node) {
        if (node == null) {
            return node;
        }
        if (node.getLeftChild() != null && node.getRightChild() != null) {
            if (node.getLeftChild().getHeight() - node.getRightChild().getHeight() > ALLOWED_IMBALANCE) {
                if (node.getLeftChild().getLeftChild() != null && node.getLeftChild().getRightChild() != null)
                if (node.getLeftChild().getLeftChild().getHeight()
                        - node.getLeftChild().getRightChild().getHeight() > ALLOWED_IMBALANCE) {
                    // perform double rotation with right child
                    node = rotateWithRightChild(node);
                } else {
                    // perform double rotation with left child
                    node = doubleRotateLeftChild(node);
                }
            } else if (node.getRightChild().getHeight() - node.getLeftChild().getHeight() > ALLOWED_IMBALANCE) {
                if (node.getRightChild().getRightChild() != null && node.getRightChild().getLeftChild() != null)
                if (node.getRightChild().getRightChild().getHeight()
                        - node.getRightChild().getLeftChild().getHeight() > ALLOWED_IMBALANCE) {
                    // perform single rotation with right child
                    node = rotateWithRightChild(node);
                } else {
                    // perform double rotation with left child
                    node = doubleRotateRightChild(node);
                }

            }
        } else if (node.getLeftChild() != null) {
            if (node.getLeftChild().getHeight() > ALLOWED_IMBALANCE) {
                node = rotateWithLeftChild(node);
            }
        } else if (node.getRightChild() != null) {
            if (node.getRightChild().getHeight() > ALLOWED_IMBALANCE) {
                node = rotateWithRightChild(node);
            }
        } else {
            ; // Leaf node
```

```
 * Rotates the given node with it left child.
private INode<T> rotateWithLeftChild(INode<T> node) {
    INode<T> leftChild = node.getLeftChild();
    if (leftChild != null) {
        node.setLeftChild(leftChild.getRightChild());
        leftChild.setRightChild(node);
    }
    return leftChild;
}
```

```
 * Performs double rotation to the given.
private INode<T> doubleRotateLeftChild(INode<T> node) {
    node.setLeftChild(rotateWithRightChild(node.getLeftChild()));
    return rotateWithLeftChild(node);
}
```


```
 * Deletes the node with the given.
private INode<T> remove(T key, INode<T> node) {
    if (node == null) {
        return null;
    }
    if (node.getValue().compareTo(key) < 0) {
        node.setRightChild(remove(key, node.getRightChild()));
    } else if (node.getValue().compareTo(key) > 0) {
        node.setLeftChild(remove(key, node.getLeftChild()));
    } else if  (node.getValue().compareTo(key) == 0) {
        if (node.getLeftChild() != null && node.getRightChild() != null) {
            isRemoved = true;
            node = findMinimum(node.getRightChild());
            node.setRightChild(remove(node.getValue(), node.getRightChild()));
        } else {
            isRemoved = true;
            node = (node.getLeftChild() != null) ? node.getLeftChild() : node.getRightChild();
        }
    }
    return balance(node);
}
```

```
@Override
public boolean search(T key) {
    INode<T> currentNode = root;
    while (currentNode != null) {
        if (currentNode.getValue().compareTo(key) < 0) {
            currentNode = currentNode.getRightChild();
        } else if (currentNode.getValue().compareTo(key) > 0) {
            currentNode = currentNode.getLeftChild();
        } else {
            return true;
        }
    }
    return false;
}
```

Ahmed Moustafa Youssef El-Naggar (16)
Hisham Osama AbdelSalam Ghazy (81)

Alexandria University
Faculty of Engineering
Computer and Systems Department

Sample Runs:

```
Insert 5
ROOT 5
5
1

Insert 4
ROOT 5
4 5
2

Insert 3
ROOT 4
3 4 5
2

Insert 1
ROOT 4
1 3 4 5
3

Insert 10
ROOT 4
1 3 4 5 10
3
```

```
Insert 5
ROOT 5
5
1

Insert 4
ROOT 5
4 5
2

Insert 3
ROOT 4
3 4 5
2

Delete 4
ROOT 5
3 5
2

Does 4 exist ?
false

Does 3 exist ?
true

Does 5 exist ?
true
```