AVL Tree

Team Members.

- Mahmoud Mohamed Hussein 70
- Moamen Raafat Elbaroudy 52
- Moustafa Mahmoud 75

Problem Statement.

- Implement an AVL tree that supports the following operations:
  - **Search:** Search for a specific element in an AVL Tree.
  - **Insertion:** Insert a new node in an AVL tree. Tree balance must be maintained via the **rotation** operations.
  - **Deletions:** Delete a node from an AVL tree. Tree balance must be maintained via the **rotation** operations.
  - **Get Tree Height:** Return the height of the AVL tree. This is the longest path from the root to a leaf-node.

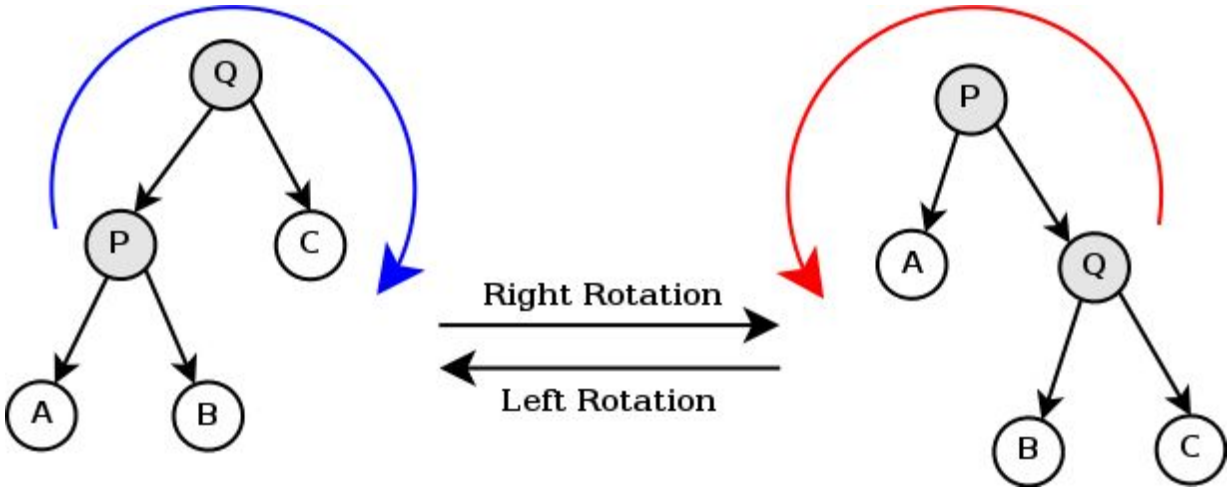
- Implement a simple English dictionary, supporting the following functionalities:
  - **Load Dictionary:** You will be provided with a text file, “dictionary.txt”, containing a list of words. Each word will be in a separate line. You should load the dictionary into an AVL Tree data structure to support efficient insertions, deletions and search operations.
  - **Get Dictionary Size:** Returns the current size of your dictionary.
  - **Insert Word:** Takes a word from the user and inserts it, only if it is not already in the dictionary. Otherwise, print the appropriate error message (e.g. “ERROR: Word already in the dictionary!”).
  - **Look-up a Word:** Takes a word from the user and returns “True” or “False” according to whether it is found or not.
  - **Remove Word:** Takes a word from the user and removes it from the dictionary. It returns true in case of successful deletion, false otherwise (e.g. if the word is not in the dictionary).

Algorithms And Data Structures.

AVL tree is a self-balancing binary search tree. The heights of the two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is done to restore this property. Search, insertion, and deletion all take $O(\log n)$ time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

- **Search:** It’s done by comparing the given key to the current node’s key then move to the left child, to the right or returns the current node. $\rightarrow O(\log n)$.
- **Insertion:** It’s done by comparing the given key to the current node’s key then move to the left child or to the right. When a leaf node is reached you can insert the new node then update the tree and rebalancing is done. $\rightarrow O(\log n)$.
- **Deletion:** It’s done by comparing the given key to the current node’s key then move to the left child, to the right or the required node is reached so it is removed after that you update the heights and rebalance the tree. $\rightarrow O(\log n)$.
- **Get Tree Height:** you only need to return the maximum height.

Insertion and Deletion operation may need node rotation.



- **Rotate Left Method.**

```
private INode<T> leftRotate(INode<T> node) {
    INode<T> right = node.getRightChild();
    INode<T> leftOfRight = right.getLeftChild();
    right.setLeftChild(node);
    node.setRightChild(leftOfRight);
    updateHeight(node);
    updateHeight(right);
    return right;
}
```

- **Rotate Right Method.**

```
private INode<T> rightRotate(INode<T> node) {
    INode<T> left = node.getLeftChild();
    INode<T> rightOfLeft = left.getRightChild();
    left.setRightChild(node);
    node.setLeftChild(rightOfLeft);
    updateHeight(node);
    updateHeight(left);
    return left;
}
```

To Iterate through the we can make use of the “java.util.Iterator**” interface is implement an iterator.**

- **Iterator class.**

```
private class TreeIterator implements Iterator<T> {
    private Stack<INode<T>> stack;

    public TreeIterator() {
        stack = new Stack<INode<T>>();
        INode<T> it = root;
        while (it != null) {
            stack.push(it);
            it = it.getLeftChild();
        }
    }

    @Override
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    @Override
    public T next() {
        INode<T> ret = stack.pop();
        if (ret.getRightChild() != null) {
            stack.push(ret.getRightChild());
            while (stack.peek().getLeftChild() != null) {
                stack.push(stack.peek().getLeftChild());
            }
        }
        return ret.getValue();
    }

}
```

AVL Tree Testing.

- A comparison between the results of the methods of java **TreeSet** and the out **AVLTree.**

```
@Test
public void testAVLTree() {
    int testSize = RANDOM.nextInt(100);
    int elementsSize = RANDOM.nextInt(10000);
    for (int j = 0; j < testSize; j++) {
        IAVLTree<Integer> actual = new AVLTree<Integer>();
        Set<Integer> expected = new TreeSet<Integer>();
        for (int i = 0; i < elementsSize; i++) {
            int temp = RANDOM.nextInt();
            expected.add(temp);
            actual.insert(temp);
            if (RANDOM.nextBoolean())
                assertEquals(expected.contains(temp), actual.search(temp));
            temp = RANDOM.nextInt();
            if (RANDOM.nextBoolean())
                assertEquals(expected.remove(temp), actual.delete(temp));
            if (RANDOM.nextBoolean())
                assertEquals(expected.contains(temp), actual.search(temp));
        }
    }
}
```

```java
@Test
public void testIterator() {
    Set<Integer> expected = new TreeSet<Integer>();
    AVLTree<Integer> actual = new AVLTree<Integer>();
    int testSize = RANDOM.nextInt(1000000);
    for (int i = 0; i < testSize; i++) {
        int element = RANDOM.nextInt();
        actual.insert(element);
        expected.add(element);
    }
    assertEquals(new ArrayList<>(expected), actual.toList());
}
```

Sample Runs.

dictionary.txt file

```
dell
toshiba
hp
sony
apple
lenovo
fujitsu
amd
intel
toyota
bmw
lada
```

```
[1] Insert a new word.
[2] Search for a word.
[3] Delete a word.
[4] Print dictionary size.
[5] Print the hight of the  dictionary's avl tree.
[6] exit.
>>> 1
>>> dell
Word already exists!
[1] Insert a new word.
[2] Search for a word.
[3] Delete a word.
[4] Print dictionary size.
[5] Print the hight of the  dictionary's avl tree.
[6] exit.
>>> 2
>>> nvidia
The word does not exist.
[1] Insert a new word.
[2] Search for a word.
[3] Delete a word.
[4] Print dictionary size.
[5] Print the hight of the  dictionary's avl tree.
[6] exit.
>>> 3
>>> hp
[1] Insert a new word.
[2] Search for a word.
[3] Delete a word.
[4] Print dictionary size.
[5] Print the hight of the  dictionary's avl tree.
[6] exit.
>>> 4
12
[1] Insert a new word.
[2] Search for a word.
[3] Delete a word.
[4] Print dictionary size.
[5] Print the hight of the  dictionary's avl tree.
[6] exit.
>>> 2
>>> hp
The word does not exist.
[1] Insert a new word.
[2] Search for a word.
[3] Delete a word.
[4] Print dictionary size.
[5] Print the hight of the  dictionary's avl tree.
[6] exit.
>>> 5
3
[1] Insert a new word.
[2] Search for a word.
[3] Delete a word.
[4] Print dictionary size.
[5] Print the hight of the  dictionary's avl tree.
[6] exit.
>>> 6
```