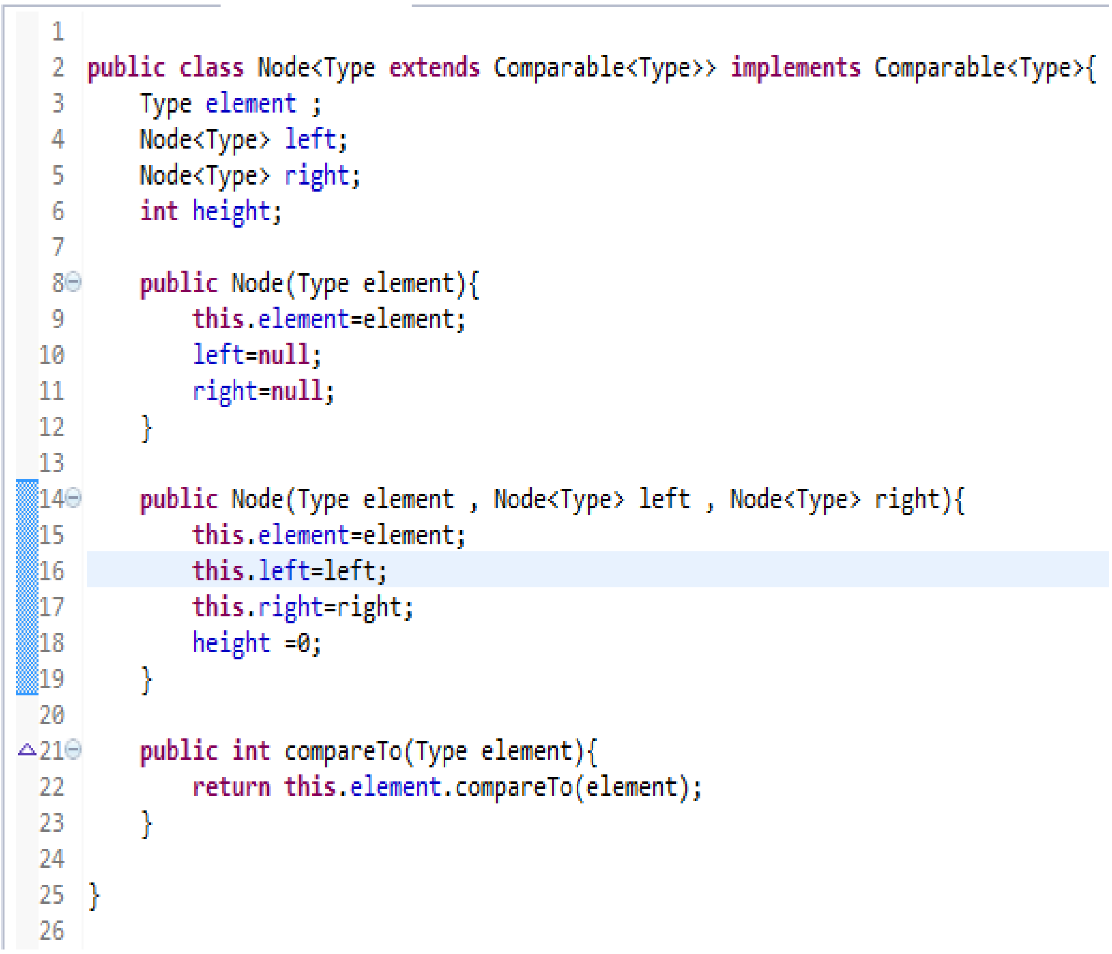
Name : Ahmed Mohamed Fathallah .   ID : 14.

Lab#2 Report

Node Class :

The node has element, right child node , left child node , and its height as attributes . it has two constructors either taking only an element as a parameter or taking an element along with left and right children . it implements comparable in order to be given an element of generic type and implement the compareTo method .



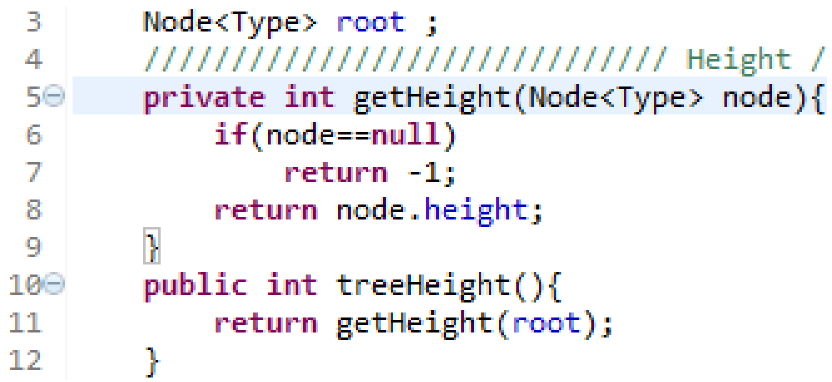
```java

public class Node<Type extends Comparable<Type>> implements Comparable<Type>{
    Type element ;
    Node<Type> left;
    Node<Type> right;
    int height;

    public Node(Type element){
        this.element=element;
        left=null;
        right=null;
    }

    public Node(Type element , Node<Type> left , Node<Type> right){
        this.element=element;
        this.left=left;
        this.right=right;
        height =0;
    }

    public int compareTo(Type element){
        return this.element.compareTo(element);
    }

}

```

AvlTree Class :

It has only one attribute which is the tree root .

Methods :

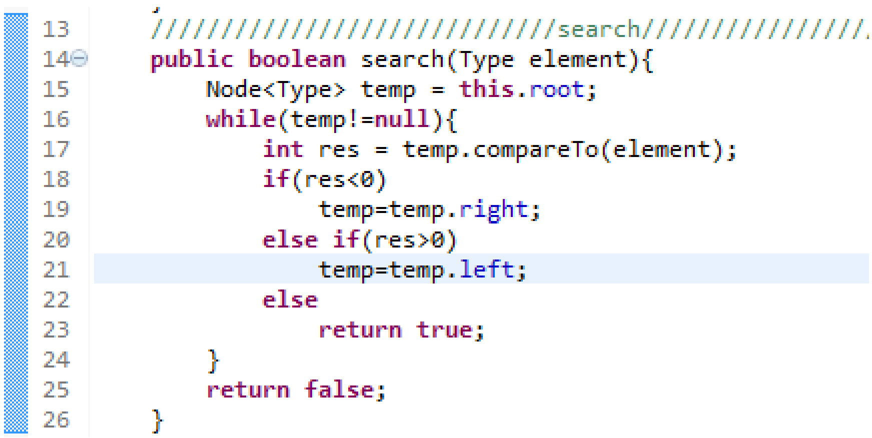
- **Private getHeight** : takes a node as a parameter and returns its height if it is not equal to null , otherwise returns -1.
- **Public treeHeight** : returns the height of the tree root .



```
    Node<Type> root ;
    ////////////////////////////// Height /
    private int getHeight(Node<Type> node){
        if(node==null)
            return -1;
        return node.height;
    }
    public int treeHeight(){
        return getHeight(root);
    }
```

- **Public search** : takes an element as a parameter and compares the element to the current node of the tree –starting from the root - , if it's less than the node the pointer is updated to the left child , if it's greater than the node the pointer is updated to the right child , if it's equal to the end the method returns true .

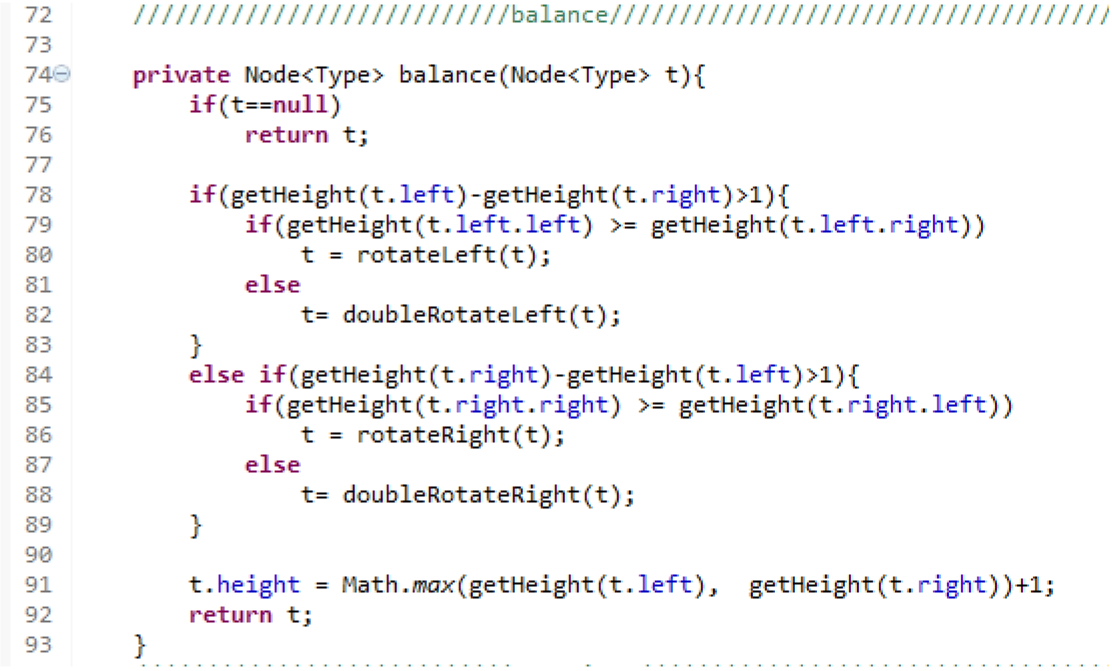
If a leaf node is reached and the element wasnot found the method returns false .



```
    /////////////////////////////search////////////////.
    public boolean search(Type element){
        Node<Type> temp = this.root;
        while(temp!=null){
            int res = temp.compareTo(element);
            if(res<0)
                temp=temp.right;
            else if(res>0)
                temp=temp.left;
            else
                return true;
        }
        return false;
    }
```

- **Private balance :** the method takes a node as a parameter and returns a node also . it checks if the node is unbalanced and wether the longer subtree is left or right , then it check wether the longer subtree of the long child is left or right in order to determine one of four cases :
    - Left child , left subtree : requires left single rotation.
    - Left child , right subtree : requires left double rotation.
    - Right child , right subtree :requires right single rotation.
    - Right child , left subtree : requires right double rotation .

  At the end of the method the height of the node is updated and the node is returned .



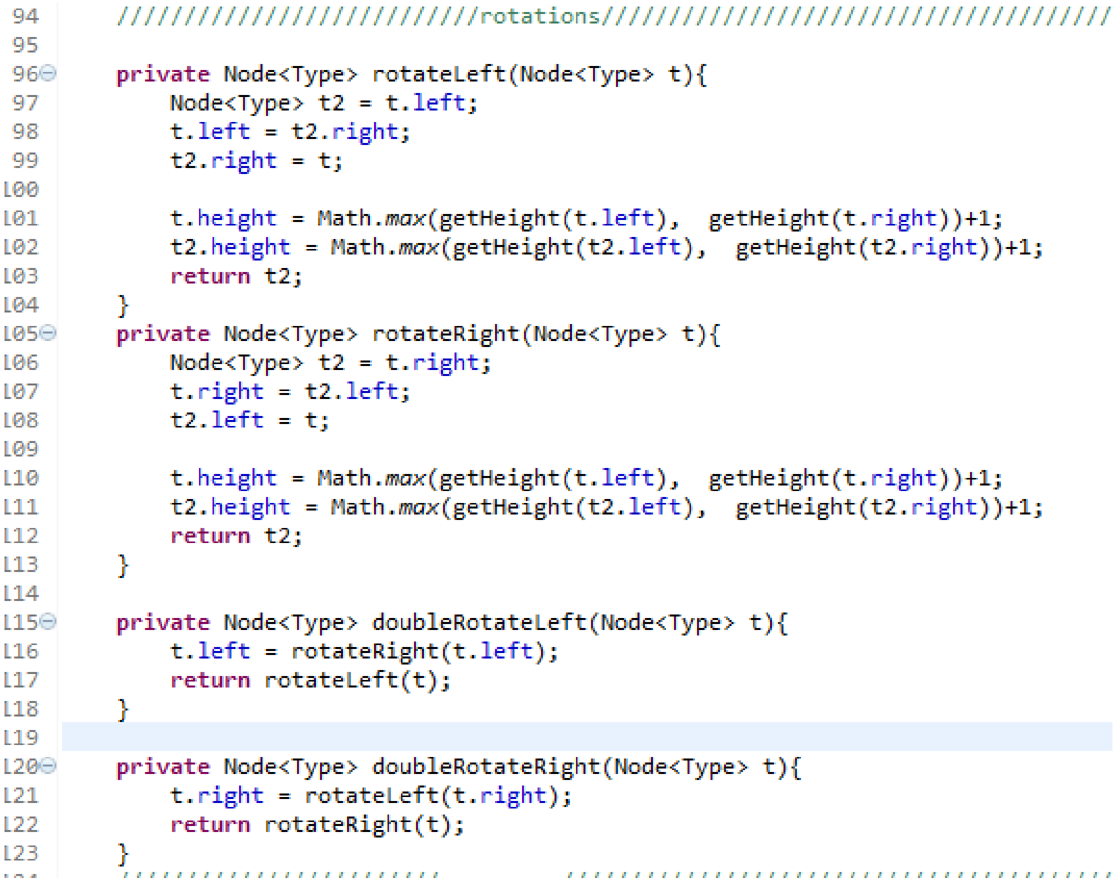
```java
///////////////////////////balance////////////////////////////////////

private Node<Type> balance(Node<Type> t){
    if(t==null)
        return t;

    if(getHeight(t.left)-getHeight(t.right)>1){
        if(getHeight(t.left.left) >= getHeight(t.left.right))
            t = rotateLeft(t);
        else
            t= doubleRotateLeft(t);
    }
    else if(getHeight(t.right)-getHeight(t.left)>1){
        if(getHeight(t.right.right) >= getHeight(t.right.left))
            t = rotateRight(t);
        else
            t= doubleRotateRight(t);
    }

    t.height = Math.max(getHeight(t.left),  getHeight(t.right))+1;
    return t;
}
```

- **Private rotateLeft** : takes a node as a parameter and performs single rotation with left child , the node's left child is assigned with the right subtree of its left child . and the left child's right child is assigned with the given node . both heights are adjusted and the left child is returned .

- **Private doubleRotateLeft** : takes a node as a parameter and performs double rotation with the left child . a single right rotation is performed to the left child then a single left rotation is performed to the given node and is returned .

- **Private rotateRight :** performs single rotation with right child . same as the rotateLeft method but with mirroring .

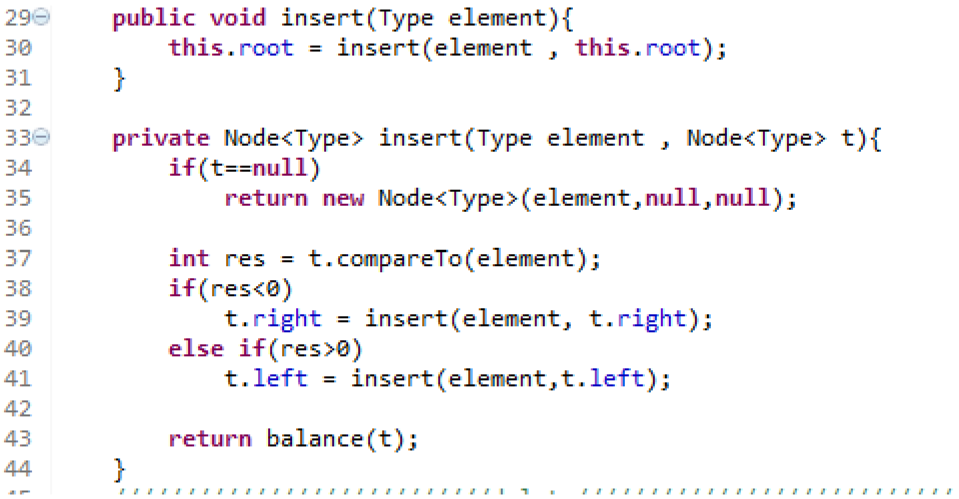
- **Private doubleRotateRight :** performs double rotation with right child . same as doubleRotateLeft method but with mirroring.



```
    ///////////////////////////rotations//////////////////////////////////////

    private Node<Type> rotateLeft(Node<Type> t){
        Node<Type> t2 = t.left;
        t.left = t2.right;
        t2.right = t;

        t.height = Math.max(getHeight(t.left),  getHeight(t.right))+1;
        t2.height = Math.max(getHeight(t2.left),  getHeight(t2.right))+1;
        return t2;
    }
    private Node<Type> rotateRight(Node<Type> t){
        Node<Type> t2 = t.right;
        t.right = t2.left;
        t2.left = t;

        t.height = Math.max(getHeight(t.left),  getHeight(t.right))+1;
        t2.height = Math.max(getHeight(t2.left),  getHeight(t2.right))+1;
        return t2;
    }

    private Node<Type> doubleRotateLeft(Node<Type> t){
        t.left = rotateRight(t.left);
        return rotateLeft(t);
    }

    private Node<Type> doubleRotateRight(Node<Type> t){
        t.right = rotateLeft(t.right);
        return rotateRight(t);
    }
```

- **Private insert :** takes an element and the tree root as parameters and returns the root . goes through the tree comparing the element to each node , if it is less than the node it goes to the left child , if it is greater it goes to the right child . when a child of a leaf node is reached(null) we return a new node with this element. Going up the recursion balance method is called for each node in order to balance the tree with rotations-if needed- and adjust the heights too .

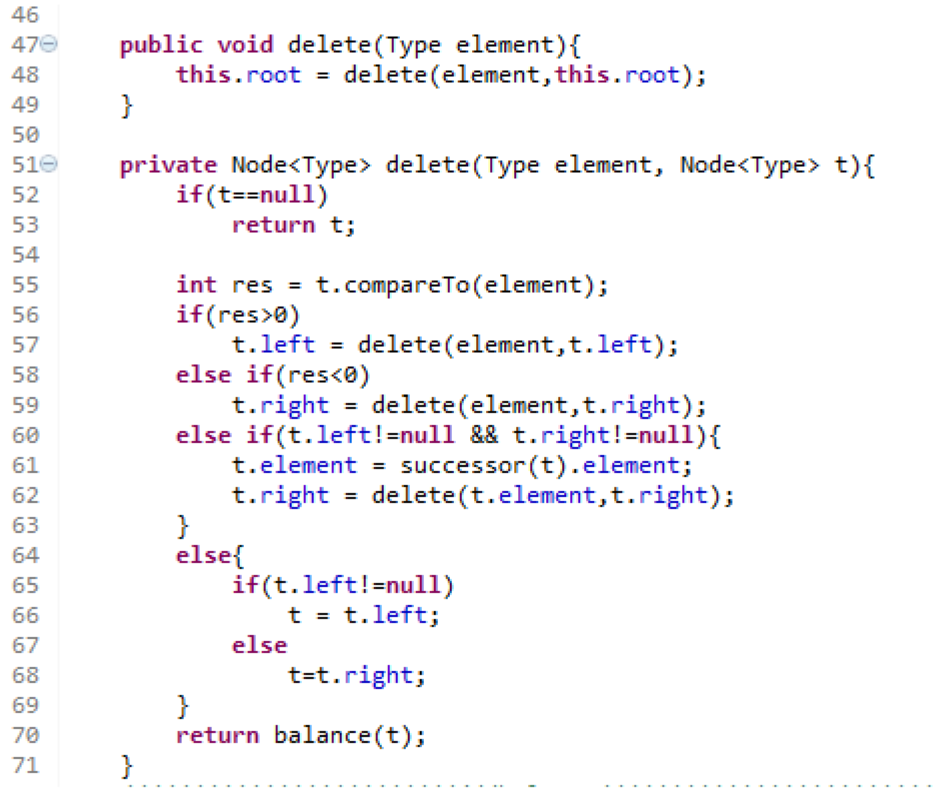
- **Public insert :** takes only the element from the user and calls the insert private method with paramters as the given element and the tree root , the node returned from the private method is stored into the tree root .



```
public void insert(Type element){
    this.root = insert(element , this.root);
}

private Node<Type> insert(Type element , Node<Type> t){
    if(t==null)
        return new Node<Type>(element,null,null);

    int res = t.compareTo(element);
    if(res<0)
        t.right = insert(element, t.right);
    else if(res>0)
        t.left = insert(element,t.left);

    return balance(t);
}
```

- **Private delete :** takes an element and the tree root as parameters traverses the tree as mentioned before till the element is found . if the element had two children it is swapped with its successor then the successor node is deleted . if the node had only one child it is replaced by its child . if the node was a leaf node it is directly removed . going up the recursion balance method is called for each node to rebalance the tree with rotations-if needed- and also adjust heights.

- **Public delete :** takes only an element from the user as a parameter and then calls the private method , the returned node is stored into the tree root .



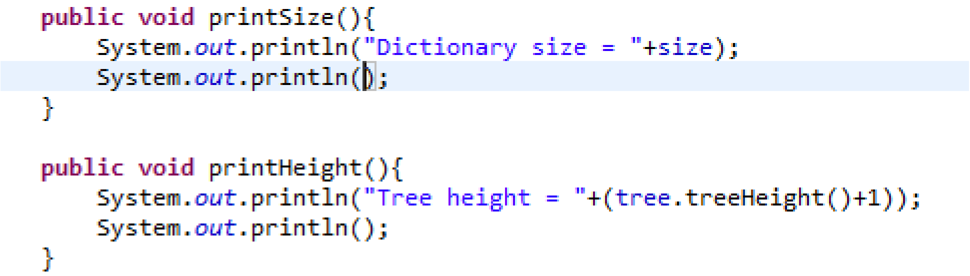
```
    public void delete(Type element){
        this.root = delete(element,this.root);
    }

    private Node<Type> delete(Type element, Node<Type> t){
        if(t==null)
            return t;

        int res = t.compareTo(element);
        if(res>0)
            t.left = delete(element,t.left);
        else if(res<0)
            t.right = delete(element,t.right);
        else if(t.left!=null && t.right!=null){
            t.element = successor(t).element;
            t.right = delete(t.element,t.right);
        }
        else{
            if(t.left!=null)
                t = t.left;
            else
                t=t.right;
        }
        return balance(t);
    }
```

Dictionary Class :

It has 3 attributes : size , Avl tree , and a buffered reader .

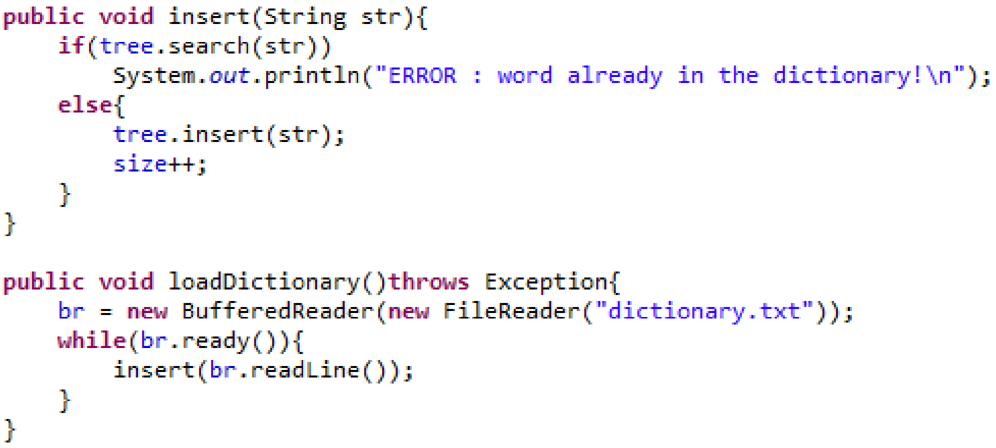
Methods :

- **Public printSize :** prints the size of the dictionary .
- **Public printHeight :** prints the height of the Avl tree .



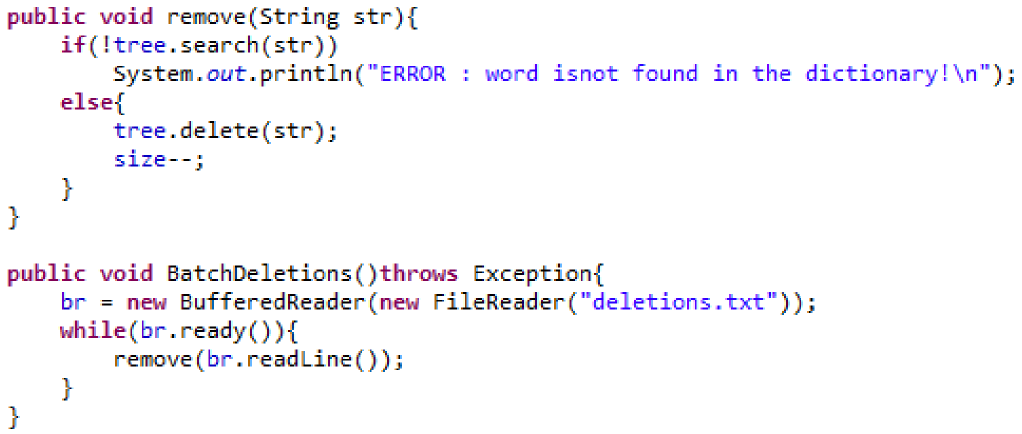
```
public void printSize(){
    System.out.println("Dictionary size = "+size);
    System.out.println();
}

public void printHeight(){
    System.out.println("Tree height = "+(tree.treeHeight()+1));
    System.out.println();
}
```

- **Public insert :** takes a string as a parameter , looks it up in the tree if found it prints an error message . other wise the string is inserted into the tree and the dictionary size is incremented .
- **Public loadDictionary :** reads a list of words from a file and inserts them into the dictionary .



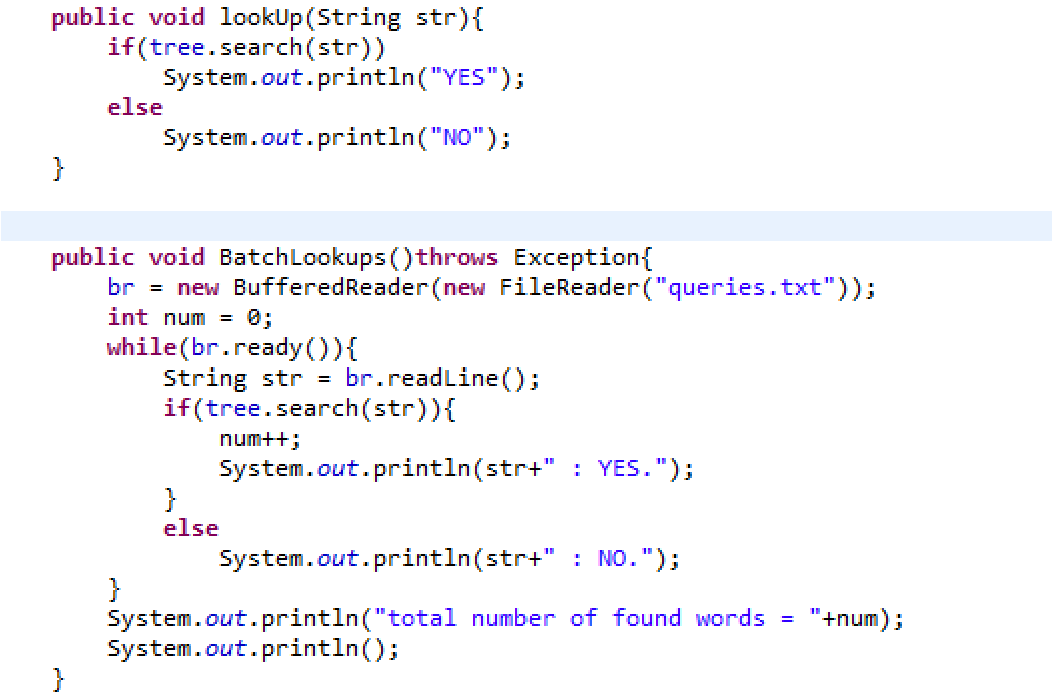
```
public void insert(String str){
    if(tree.search(str))
        System.out.println("ERROR : word already in the dictionary!\n");
    else{
        tree.insert(str);
        size++;
    }
}

public void loadDictionary()throws Exception{
    br = new BufferedReader(new FileReader("dictionary.txt"));
    while(br.ready()){
        insert(br.readLine());
    }
}
```

- **Public remove :** takes a string and searches for it in the tree , if found the string is deleted and size is decremented .if not found an error message is printed .
- **Public batchDeletions :** reads a list of words from a file and remove them from the dictionary .



```
public void remove(String str){
    if(!tree.search(str))
        System.out.println("ERROR : word isnot found in the dictionary!\n");
    else{
        tree.delete(str);
        size--;
    }
}

public void BatchDeletions()throws Exception{
    br = new BufferedReader(new FileReader("deletions.txt"));
    while(br.ready()){
        remove(br.readLine());
    }
}
```

- **Public lookUp** : takes a string and searches the tree for it . if found prints yes , other wise prints no .
- **Public batchLookups** : reads a list of words from a file and look them up , for each word the word is printed along with yes or no as a result for looking it up . at the end the total number of found words is printed .



```
public void lookUp(String str){
    if(tree.search(str))
        System.out.println("YES");
    else
        System.out.println("NO");
}

public void BatchLookups()throws Exception{
    br = new BufferedReader(new FileReader("queries.txt"));
    int num = 0;
    while(br.ready()){
        String str = br.readLine();
        if(tree.search(str)){
            num++;
            System.out.println(str+" : YES.");
        }
        else
            System.out.println(str+" : NO.");
    }
    System.out.println("total number of found words = "+num);
    System.out.println();
}
```