Assignment 1 - cf222jf

Excercise 1, the binary search tree. Method Analysis

public void insert(Integer value).

This method is done by using a recursive function to navigate to the tree and find a suitable place for the node to be inserted. So firstly there is a while loop to navigate the tree, and once the place has been found, we break the loop and add the new node. We also check for duplicates, and if there are duplicates we break immediately. The time required for this operation depends on the depth of the tree.

The worst case scenario for this insertion is when we have a non-balanced tree and we have to traverse every node. The time for this would then be **O(N)**

– public Integer mostSimilarValue(Integer value)

This is a bit similar since we also use a recursive function to traverse the tree. This one differs because we start by calculating the root. If the value we are looking for is not the root, then we check to see if the value is higher or lower than the root value. If Higher, we traverse to the right side of the tree, and if lower we traverse to the left part of the tree. This means that we can optimize performance in a balanced tree, as we only need to check one side of the tree.

```
@Override
public Integer mostSimilarValue(Integer value) {
    mostSim = root.getValue();      // have to put this in case the root is actually the most similar one,
    compareSum = Math.abs(root.getValue() - value);

    if (value > root.getValue()) {
        visitForSimilar(root.getRight(), value);
    }
    if (value < root.getValue()) {
        visitForSimilar(root.getLeft(), value);
    }
    if (value == root.getValue()) {
        mostSim = root.getValue();
    }

    System.out.println("The most similar value for the Integer ("+value+") is : "+mostSim);
    return mostSim;
}
public void visitForSimilar(TreeNode node, Integer Value) {
    if (node.getLeft() != null) {
        visitForSimilar(node.getLeft(), Value);
    }
    int absolute = Math.abs(node.getValue()-Value);
    if(node.getValue() == Value) {
        compareSum= 0;
        mostSim = node.getValue();
    }

    else if(absolute < compareSum) {
        compareSum = absolute;
        mostSim = node.getValue();
    }
    if (node.getRight() != null) {
        visitForSimilar(node.getRight(), Value);
    }

}
```

Worst case is still having an unbalanced tree, which in worst case is **O(N)**

– public void printByLevels()

This is done by having an iterator that creates a Node array, and every index such as 0,1,2 …. are the levels and they are all a linked list. So every node on level 2 are linked to the last node on the [2] index. The iterator firstly traverses every node and adds them to this Node array from left to right. This means that we get the correct output when printing as well.

```
@Override
public void printByLevels() {
    // TODO Auto-generated method stub
    TreeIterator itr = new TreeIterator(size, depth ,root);
    itr.PrintLevels();


}
```

```
class TreeIterator implements Iterator{
    TreeNode[] Nodes;
    ListNode[] LevelArray;
    int i = 0;
    int itrCounter;

    public TreeIterator(int size, int depth, TreeNode root) {
        LevelArray = new ListNode[depth+1];
        Nodes = new TreeNode[size];
        Fill(root);
    }

    public void Fill(TreeNode node) {
        if (node.getLeft() != null) {
            Fill(node.getLeft());
        }
        Nodes[i] = node;
        i++;
        addToLevel(node);
        if (node.getRight() != null) {
            Fill(node.getRight());
        }
    }
```

The array called Nodes is just for easier printing of all the nodes.

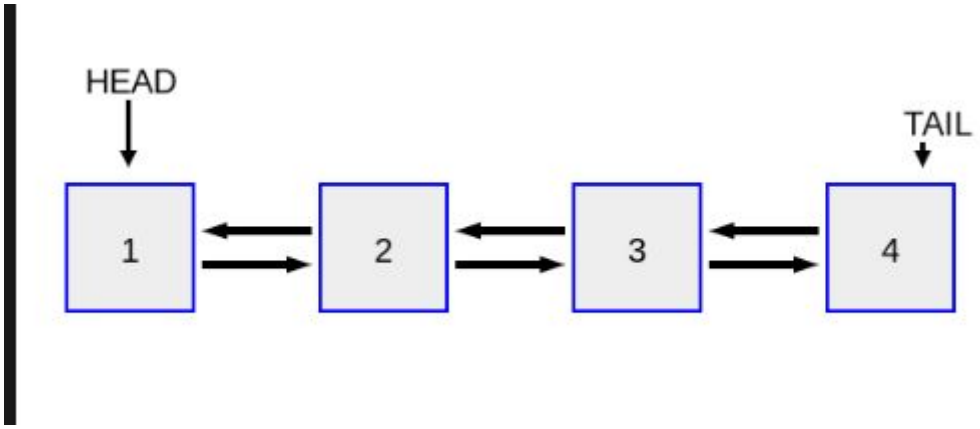
The time complexity for this operation is still dependant on the tree, as we use the same recursive method to traverse the tree and add the nodes. The time complexity is still O(N) for the traversing, and the printing is also O(N) as you have to traverse every node once, so the total time complexity is:

O(N)+O(N) = 2 O(N) = O(N)

Exercise 2 - Implementation of “Sequence with Minimum

I would propose a double linked list where you would have a head and tail. Then we would have 2 pointers, one on the leftmost value and one on the rightmost value. Everything else would be linked together like a nodelist.

The nodes will just have a value, and 2 pointers to the previous and the next node.



– public void insertRight(Integer value)

Since we have a pointer to the rightmost value, we simply add another node and move the pointer to the new node. Since we already know where to go, the time complexion for this task will be **O(1)** since the operation is constant and not dependant on size.

```
@Override
public void insertRight(Integer value) {
    if (left == null && right == null) {
        left = new Node(value);
        right= left;
    }
    else {
        right.setNext(new Node(value));
        right.getNext().setPrevious(right);
        right = right.getNext();
    }

}
```

– public Integer removeRight()

This one will also be quick since we know which value to remove and we have a pointer to it already. So we remove this by changing the pointer to the previous node and remove the connection to it. Therefore the time complexity is constant and for this operation will be **O(1).**

```java
@Override
public Integer removeRight() {
    int value;
    if (right == null) {
        System.out.println("there is no right node");
    }
    else if (left == right) {       // error handling if there is a single node left.
        value = right.getValue();
        left = null;
        right = null;
        return value;
    }
    else {
        value = right.getValue();
        right = right.getPrevious();
        right.setNext(null);
        return value;
    }
    value = (Integer) null;
    return value;
}
```

– public void insertLeft(Integer value)

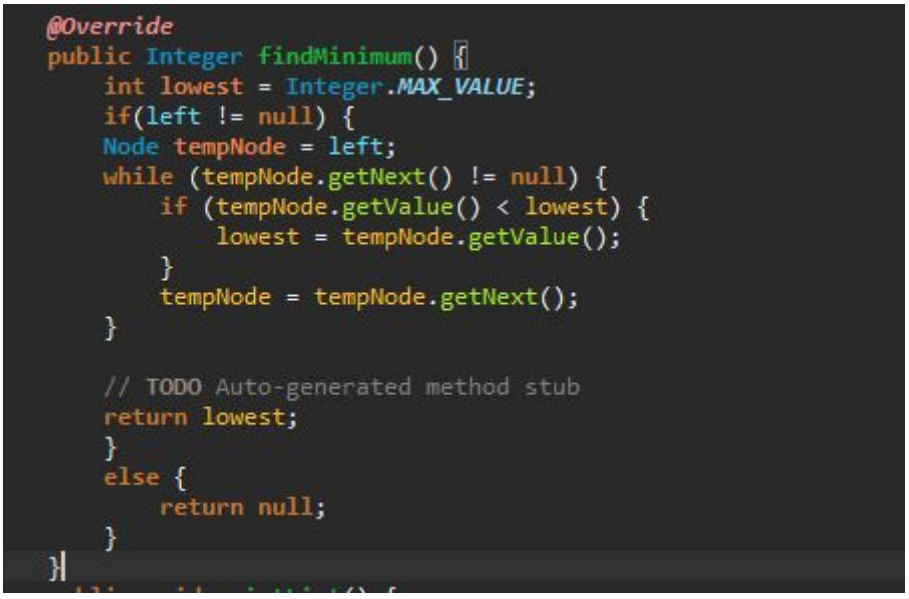
Same as the InsertRight function.

– public Integer removeLeft()

Same as the remove left function.

– public Integer findMinimum()

This one is slower than the rest of the functions as this one is dependant on the number of nodes in the list. This function goes from the left (tail) of the list and iterates over all nodes in the list and saves the lowest value which is then later returned. **Worst case:** The node with the lowest value will be the furthest away and the last node to find. This means that the finding of the lowest value is linear since we have to iterate over all the nodes. Therefore the time complexity for this is **O(N)**



```java
@Override
public Integer findMinimum() {
    int lowest = Integer.MAX_VALUE;
    if(left != null) {
    Node tempNode = left;
    while (tempNode.getNext() != null) {
        if (tempNode.getValue() < lowest) {
            lowest = tempNode.getValue();
        }
        tempNode = tempNode.getNext();
    }

    // TODO Auto-generated method stub
    return lowest;
    }
    else {
        return null;
    }
}
```