Mehdi Hmidi (mh223vk)

1. Analyze the worst-case time complexity of operations mostSimilarValue and printByLevels.

1.1. MostSimilarValue

```
@Override public Integer mostSimilarValue (Integer value) {
    if (contains (value)) return value;
    return similar (root, value, 0);
}

private Integer similar(Node comparator, Integer value, Integer s0) {
    if(comparator.key>value && comparator.left != null)
        if(Math.abs(comparator.key - value) >= Math.abs(comparator.left.key - value))
            return similar(comparator.left,value, comparator.left.key);
        else
            return similar(comparator.left, value, comparator.key);

    else if(comparator.key<value && comparator.right != null )
        if(Math.abs(comparator.right.key - value) >= Math.abs(comparator.key - value))
            return similar(comparator.right, value, comparator.key);
        else
            return similar(comparator.right, value, comparator.right.key);
    return s0;
}
```

- Searching in a BST starts at the root and yields a recursive algorithm.
- This process is binary, thus **search is O (log N)**.
- A BST has O (h) worst-case runtime complexity, where h is the height of the tree. Since s binary search tree with n nodes has a minimum of O (log N) levels, it takes at least O (log N) comparisons to find a particular node.
- Unfortunately, a binary search tree can degenerate to a linked list, effectively reducing the search time to O (n). That is the worst-case scenarios for a BST and it involves a completely unbalanced tree, which degenerates into a single-linked list. This presents a challenge since we do not get the "halving at every step" characteristic that gives a Binary Search Tree its power.

1.2. PrintByLevels

The exercise instructions describe a **Level order traversal that** processes the nodes level by level. The method would traverse the root, and then its children, then its grandchildren, and so on. Unlike the other traversal methods, it seems that ***a recursive version does not exist.***

The possible implementation I read about, is similar to the non-recursive preorder traversal algorithm. The only difference is that a stack is replaced with a FIFO queue. But since we are not allowed to use such data structures. I will content myself with O (N^2).

- PrintLevel () takes O (N) time where N is the number of nodes in the tree. So time complexity of ***printByLevels ()*** is O (N) + O (N-1) + O (N-2) +... + O (1) which is O (N^2).

```
@Override public void printByLevels(){ //using Breadth First Search traversal.
    int h = depthTree(root);
    for (int i=1; i<=h; i++){
        System.out.print("Depth "+(i-1)+" :");
        printLevel(root, i);
        System.out.println("\n");
    }
}

private int depthTree(Node root) {
    if (root == null) return 0;
    else {
        int left = depthTree(root.left);
        int right = depthTree(root.right);
        if (left > right) return(left+1);
        else return(right+1);
    }
}

private void printLevel(Node root , int level) {
    if (root == null) return;
    if (level == 1) System.out.print(root.key + "  ");
    else if (level > 1) {
        printLevel(root.left, level-1);
        printLevel(root.right, level-1);
    }
}
```

I could have done this: Using a Queue, which significantly lowers the Time Complexity to O (N) where **N** is number of nodes in the binary tree.

```
@Override public void printByLevels(){ //using Breadth First Search traversal.
    if(root == null) return;
    Queue<Node> q =new LinkedList<>();
    q.add(root);
    while(true){
        int nodeCount = q.size();
        if(nodeCount == 0) break;
        while(nodeCount > 0) {
            Node node = q.peek();
            System.out.print(node.key + " ");
            q.remove();
            if(node.left != null) q.add(node.left);
            if(node.right != null) q.add(node.right);
            nodeCount--;
        }
        System.out.println();
    }
}
```

2. Proposed Data structure

In order to achieve, O (1) for insert/remove right and insert/remove left, I chose to implement **a doubly linked list**, since it allows convenient access in the obvious way by storing two pointers. To comply with the operations needed, the DLL implementation differs from the text book solution. The insert/remove methods make use of the head and tail node to execute the concept.

In that regard, the assignment also describes a *find minimum method that should not surpass worst-case O (N)*.

A straight forward implementation would compare the nodes until reaching the end to make sure the minimum value is found. However, to surpass this theoretical limit, I read through encoding methods and discovered that by adding a **Key** to my nested **Node** class I can retain information about the min value and *intuitively link the other node values in a scheme that would allow me in the event of a remove operation to* ***recalculate the min value in O (1) time.***

Now, the solution is not perfect the encoding does need an extra if statement when the ADT only holds one node to work so that the **find min** operation executes correctly.

```
//O(1)
@Override public Integer findMinimum(){
    if (head == null)
        throw new RuntimeException("List is empty!");
    if(head==tail)
        return head.getValue();
    return minEle.getValue();
}
```

Same with Insert/remove operations, the implementation is straight forward and takes O (1) as the lines of codes remain **in the boundary of constant time**. Lastly, **ToString ()** takes O (N).

```
//O(1)
@Override public void insertRight(Integer value) {
    Node added = new Node(value,value);
    if (head == null){
        minEle=added;
        head = added;
        tail = added;
    } else{
       if( added.getEncoding() < minEle.getEncoding()) {
// When inserting the values are encoded to retrace back the minimum if it's removed.
           added=new Node(value,2*added.getEncoding() - minEle.getEncoding());
           minEle=new Node(value,value);
       }
       added.prev = tail;
       this.tail.next = added;
       this.tail = added;
    }
    this.size++;
}
```