Lab 7 Trees

Weichen Chai

Fall 2022

Introduction

This lab report seeks to answer the questions presented in "Trees" for ID1021. In this assignment, the goal is to analyse the binary tree, which splits into right and left. The operations we will be coding and analysing will be adding and searching for, removal of an item, and an iterator.

Tasks

Add and lookup

To add elements, we first construct the binary tree, in which nodes has a key, value and left and right. If we want to keep the tree sorted, we follow the rule of, if the value of the new node is smaller than current nodes' we go to the left, if its greater than the current node, we go to the right child, when it is null, we insert a new node. The lookup operation is very similar to the add operation in identifying the node, it follows the same principle.

```
public void add(Integer key, Integer value) {
        root = addR(root, key, value);
    }

private Node addR(Node n, Integer k, Integer v) {
        if (n == null){
            return new Node(k, v);
        }
        int c = k.compareTo(n.key);
        if (c < 0){
            n.left  = addR(n.left,  k, v);
        }
        else if (c > 0) {
            n.right = addR(n.right, k, v);
        }
        else {
```

```
            n.value = v;
        }
        return n;
    }
```

Lookup function:

```
public Integer lookup(Integer key) {
        return lookupR(root, key);
    }
    private Integer lookupR (Node n, Integer key) {
        if (n == null) {
            return null;
        }
        int c = key.compareTo(n.key);
        if(c < 0) {
            return lookupR(n.left, key);
            }
        else if (c > 0) {
            return lookupR(n.right, key);
            }
        else {
            return n.value;
        }
    }
```

The question presented in this tasks asks us to compare the lookup algorithm used in binary tree to binary search algorithm from the previous assignment. As we have concluded from the previous tasks, sorted arrays, binary search has the time complexity of O(log n).

Array Size	**Binary Tree**	**Search Algorithm**
1000	106	95
2000	125	104
3000	135	126
4000	145	137
5000	158	145
10000	198	187
11000	199	189
12000	205	182

Its clear that the binary tree lookup follows a O(log n), similar to the binary search algorithm. The worst case time complexity for the binary tree would be similar to a linked list, which is O(n), as it would have to go

through the entire tree to find that node, but in the case there the tree is distributed into a more balanced manner, the general search algorithm will take O(log n), which is the case here, as it ran for 1000000 iterations. The binary search is visibly faster than tree search, and this is due to the fact that array access is faster than nodes, the usage of Integer may also slow down the process, as it takes longer to execute than primitive data types such as int.

Depth first traversal

The second task we have requires for us to implement a method in which we traverse through all the items in a tree, where we go through the tree from left to right, this method is called depth first, where we go to the leftmost element before considering what to do after that. The code shown in this task recursively prints the items in the left branch, and print the nodes whilst traverse through the nodes. Another way to do this is to use an iterator and a stack, which will be shown in the next section.

Iterator

When implementing an iterator, we would need to finish the TreeIterator partially given to us in the task.

```
public TreeIterator(Node c) {
    next = c;
    stack.push(next);
    while(next.left != null){
        next = next.left;
        stack.push(next);
    }
}
```

In this iterator, it has a stack that can be pushed and poped to store nodes, and node next, to indicate the printed node in the tree. Similar to the print method in the previous section, the TreeIndicator will traverse through the leftmost(smallest) branch and push the nodes it encounters to a stack.

The iterator will also contain hasNext() and next() functions, shown as follows.

```
public boolean hasNext() {
            return !stack.isEmpty();

        }
```

```
public Integer next() {
            if (!hasNext()) {
                return null;
            }
            Node next = stack.pop();
            if (next.right != null) {
                stack.push(next.right);
                Node c = next.right;
                while(c.left != null){
                c = c.left;
                stack.push(c);
            }
        }
        return next.value;
}

  public void remove() {
        throw new UnsupportedOperationException();
    }
```

The hasNext() checkes if the stack has an empty value, it will return true if the stack is not empty. The next() function will return null if stack is empty, otherwise it will pop the top element placed in the stack, which is the leftmost leaf, it checks if there are right branches in the from the leftmost leaf, if there appears to be a right branch, then it will push its right to the stack, and continues the process of checking left branch.

Using a stack

When we run the codes given to us in the task to test with the iterator, the correct results will show up, as it contains all the items we want to add without losing values, however when we add items in the for each loop and further experiment with the codes, there will be instances where the iterator does not show node with smaller key than the current, in this case, the iterator does not show the newly added element, however, the element is actually added already in the tree. When the iterator runs from the beginning again, the added element will emerge.