Binary Trees

Dhruba Das

Spring Fall 2023

Introduction

This assignment was a little complex for me; this is because I have not implemented methods or worked with binary trees before, but with the help of the guidelines given in the pdf and the lectures, I was able to complete it. This assignment has 3 problems in total: Implementing a binary tree and its methods, Implementing an iterator, and Benchmarking the lookup function.

Binary Tree Structure

Node Structure

The node structure is essentially the same as that of the doubly linked list, except it now has the two pointers called left and right which points to the left and the right of the current node. Below is the code snippet:

```
public class Node {
        public Integer key;
        public Integer value;
        public Node left, right;

        public Node(Integer k, Integer v){
            this.key = k;
            this.value = v;
            this.left = null;
            this.right = null;
        }
```

Lookup method

This method is used to search for and return the value of a node in the Binary Tree, given it's key. This was a little confusing to implement as we

had to compare the keys and traverse the tree accordingly. Below is the code snippet:

```
private Integer search(Integer k){
            if(this.key == null)
              return null;
            if(this.key == k)
              return value;
            else if(this.key > k)
              if(this.left != null)
                return left.search(k);
              else
                return null;
            else
              if(this.right != null)
                return right.search(k);
              else
                return null;
        }
 ...
 public Integer lookup(Integer k){
            if(root != null){
             root.search(k);
            }
            return null;
        }
```

Add method

This method is used to add a new node to the tree, given the key and the value of the node as the arguments. Below is the code snippet with comments made to clarify the code:

```
public void add(Integer k, Integer v) {
            if (root != null) {
                Node cur = root;

                // Traverse the tree to find the correct position for the new node
                while (cur != null) {
                    if (cur.key == k) {
                        // If the key already exists, update the value
                        cur.value = v;
                        break;
                    } else if (cur.key > k) {
```

```
                    // If the key is smaller, move to the left subtree
                    if (cur.left != null) {
                        cur = cur.left;
                    } else {
                        // If the left node is null, insert a new node
                        cur.left = new Node(k, v);
                        break;  // Break out of the loop after insertion
                    }
                } else {
                    // If the key is larger, move to the right subtree
                    if (cur.right != null) {
                        cur = cur.right;
                    } else {
                        // If the right node is null, insert a new node
                        cur.right = new Node(k, v);
                        break;  // Break out of the loop after insertion
                    }
                }
            }
        } else {
            // If the tree is empty, create a new root node
            root = new Node(k, v);
        }
    }
```

Implementation of the Iterator

The concept of this was a little hard for me to grasp at first, but from what I understood, the objective is to create a class that will iterate through a tree in ascending order , while making use of a stack to keep track of parent nodes.

TreeIterator class

Below is the code snippet for the TreeIterator classes, with comments which explain what each line of code does:

```
private class TreeIterator implements Iterator<Integer> {
            private Node next;
            private Stack<Node> stack;

            public TreeIterator() {
                stack = new Stack<Node>(); //initializes a new stack
                next = root; //start at root
```

```
            while(next.left != null){
                stack.push(next); //push the nodes on the stack as we go down
                next = next.left; // go down to leftmost leaf
            }
        }
```

hasNext() method

This method is used to check (as we traverse in an ascending order) if there exists a next node. Below is the code snippet with comments:

```
@Override
        public boolean hasNext() {
            return next != null;
            // returns true if next isn't null; returns false otherwise
        }
```

next() method

This method was quite difficult to implement, as at first, it was hard for me grasp the concept, even with the guidelines. I had to try out a couple different layouts of the code, before I decided to also look at the guidelines provided by GeeksForGeeks for the iterator for in-order traversal. Then I made some modifications to the code according to the stack I was using. Below is the code snippet with comments for clarification:

```
public Integer next() { // returns value of next node
       if (!hasNext()){
          throw new NoSuchElementException(); //Checks if there is a next node
                }

     Node current = stack.peek(); //current node is set to parent node
     Node nxtVal = stack.pop(); // node that we want the value of

     if(current.right != null){ //checks if next node has a right branch
         current = current.right; // if yes, we move to that branch
             while (current != null){ //move to leftmost node of the right branch
                 stack.push(current);
                  current = current.left;
                    }
                }
     if(stack.isEmpty()){ //checks if we have reached the last node
          next = null;
                }
```

```
else{
   //return to the parent node and keep traversing in ascending order
  next = stack.peek();
           }
return nxtVal.value;
       }
```

Once I ran the test program provided in the pdf of the assignment, it produced the following:

```
next value 102
next value 103
next value 105
next value 106
next value 107
next value 108
```

Since the code worked, it should not be an issue in case we wanted to add new elements to the tree (i.e no loss of elements).

Benchmarks

Below I have the figures which represent the time taken by the lookup function to search for nodes, as the size of the binary tree grows.

Size(n)	**Time taken for Lookup (ms)**
1000	106
2000	124
3000	142
4000	154
5000	164
6000	172
7000	179
8000	185
9000	190
10000	197

Table 1: (For each size, a thousand elements are looked up; for each size, benchmark is run 1000 times; min. time is taken; time in microseconds)

Compared to binary search, it's about 10 times slower, but the time complexity for both lookup and binary search is O($nlog(n)$). Binary search is faster as we can pinpoint the exact location of the middle of an array, whereas for lookup, we assume that the Binary tree is well balanced and we hope to land in the middle.