Trees

Aditya Gupta

September 2022

1 Introduction

In this assignment, we discuss the tree data structure. This is another node-based data structure where each node can have at max 2 child nodes, the left and the right nodes. The data structure is designed so that it resembles a tree where it begins at the root node and then branches into the left and right child nodes and culminates in the leaf nodes which have null child nodes. This data structure is better known as a binary tree as it can have up to 2 child nodes for each node in the structure. This data structure is formed using nodes that contain a key and a value element and left and right node pointers.

We implement an `add()` function to add nodes to the tree and a `lookup()` function that looks for a specific key inside of the tree. After this, we create an explicit stack to print all the values inside of the tree and implement an iterator for the data structure. We don't know if our tree would be balanced or not as we generate a random tree when we benchmark our tree implementation. We will discuss more about balanced and unbalanced trees later.

2 Binary Tree

In the binary trees, all the nodes contain key-value pairs. The relationship between the keys and the values is like a mathematical function, where every key maps to a value. We previously stated that our tree might be unbalanced and that is because we add random nodes to the tree which means that when we add nodes there is a high likelihood that there might be more nodes that have a key that is smaller or larger than the key of the root node, thus leading to more nodes being placed in the right or the left part of the tree.

This means that such a tree would be unbalanced also giving us the worst-case scenario of getting a linked list instead of a tree as all the nodes could have been added to the right or the left of the root. This is bad as the time complexity of the add and lookup operations is going to be a lot more time-consuming in that case and will thus defeat the entire purpose of having a binary tree to begin with.

The Add node Function

We check the nullity of the root node inside the add function inside the Binary Tree class and if the root is null we set it to a new node containing the key and value passed as an argument to that function. Otherwise, we call the add function inside of the TreeNode class to add the key at an appropriate position inside the tree.

```java
public void add(Integer key, Integer value){
    if(this.key == key){
        this.setValue(value);
        return;
    }
    if(this.key > key){
        if(this.left != null){
            this.left.add(key,value);
        }
        else{
            this.left = new TreeNode(key,value);
        }
    }
    else{
        if(this.right != null){
            this.right.add(key,value);
        }
        else{
            this.right = new TreeNode(key,value);
        }
    }
}
```

If we find the key value inside of the tree we just update the value of that node and return. Otherwise, we check if the current node's key is greater or smaller than the value of the key passed as an argument and based on that we add a node to the left or the right of the current node recursively by going further into the right or left of the current node. Once we recurse to the furthest right or left the position in our tree then we add the node to the tree. This operation has a time complexity of $O(h)$ where h is the height of the tree. In terms of the number of elements, we can talk about its worst-case time complexity of $O(n)$ which would be in the scenario where we have an unbalanced tree with all nodes being to the right or left of the root node, hence acting like a linked list instead of a tree.

The Lookup Function

The lookup function is used to search for a specific key inside of a tree. The lookup function inside of a tree is a lot more efficient than searching for elements inside of the previous data structure that is the linked list, where searching for an element takes linear time. This is more efficient as we reduce the number of paths that we can go down to search for a key by half each time by comparing the value of the key against the current node each time. Thus the time complexity of such a function is of the order $O(log(n))$.

To search for an item we first call the lookup function inside of the Binary Tree class, which checks if the tree is empty or not by checking the nullity of the root. We return null if the tree is empty as we can't locate any key inside an empty data structure. Otherwise, we call the lookup function inside of the Node class that checks if the current node's key value is the same as the key passed as an argument, if it is then we return the current node's value.

Otherwise, we check if the key is less than the current node's key value while the left node of the current node is not null and if this holds true then we recursively call the same function but for the left node. Else, we check if the current node's key is smaller than the key argument while the right node of the current node is not null, if this holds true then we recursively call the same function but for the right node of the current node instead.

```java
public Integer lookup(Integer key){
    if(this.key == key){
        return this.value;
    }
    else{
        if(this.key > key  && left != null){
            return left.lookup(key);
        }
        else if(this.key < key  && right != null){
            return right.lookup(key);
        }
    }
    return null;
}
```

3 Benchmarks

We now benchmark the add() and lookup() functions inside our binary tree implementation. We will also compare the results of the lookup function with the execution time of doing a binary search in an array of equivalent size.

3.1 `add()` Function:

Case	n	Time in ns
1	250	209
2	500	135
3	1000	121
4	2000	144
5	4000	210
6	8000	197
7	16000	329
8	32000	550
9	64000	1021
10	128000	2066

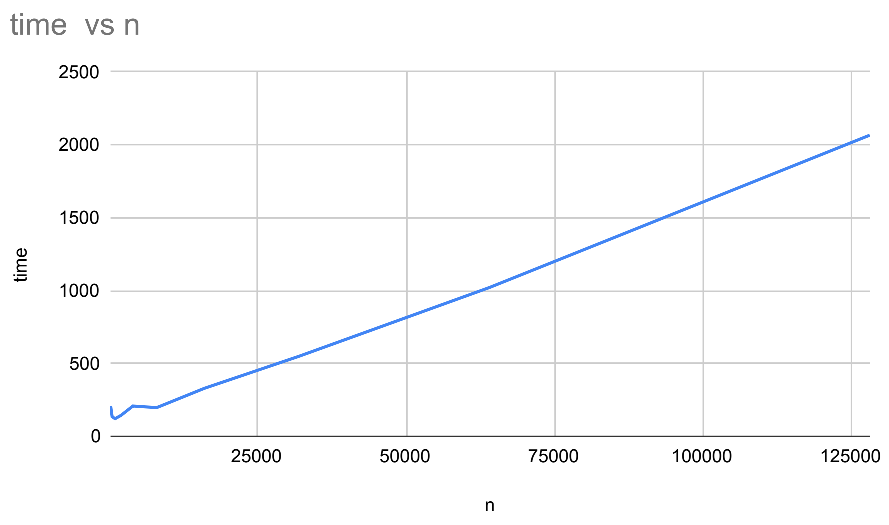

Figure 1: Linked Lists Case 1: A - Increases; B - Fixed

We see from the graphs and tables above that the time taken to add a new key-value pair to the tree takes linear time or we can say that this operation has a time complexity of $O(n)$. This is true as we have to traverse the tree whenever we add a new node to the tree to place it in its appropriate position based on the order of nodes inside of the tree. Here we can say that it isn't very different from its other node-based counterpart the linked list which also takes linear time to append elements to the end of the list if we don't keep track of the last element.

3.2 `lookup()` Function:

Case	n values	Time for binary search in ns	Time lookup in ns
1	250	328	233
2	500	610	210
3	1000	794	330
4	2000	889	502
5	4000	1181	578
6	8000	1420	606
7	16000	1641	973

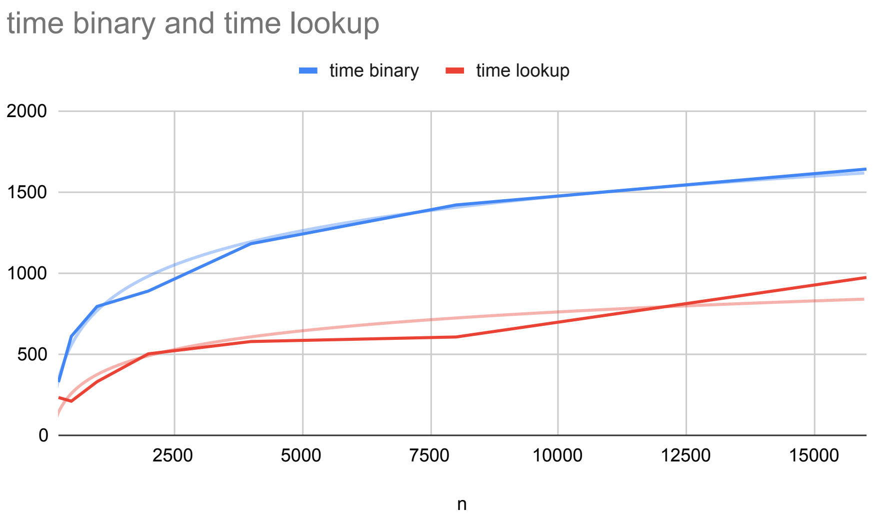

Figure 2: Lookup in a tree vs Binary search

Now we compare the time taken for the lookup function and compare it against the binary search algorithm for an array of the same size. We can see from the graphs that for both of the functions we have a logarithmic trend or we can say that both the operations have a time complexity of $O(log(n))$. This is because how we reduce the number of paths in which we search for a specific key by half each time in the lookup function as discussed in the lookup function section of this report. And in the case of the binary search, we use the divide and conquer strategy to solve the problem leading to the same logarithmic trend we saw for the lookup operation. We see that even though both the algorithms follow a similar trend however the binary search takes more time than the time taken to do the lookup operation.

4 Iterator and Stack

An iterator is generally an object that allows one to traverse through any collection of data, for example in our case a tree. If we have an iterator for a data structure then we can do itemized loops for the data structure i.e. we can do for-each loop for the data structure to traverse through it. This also means that we don't need to know the size of our collection to be able to iterate over it. The main parts of our custom iterator implementation consist of it which include the $next()$ and $hasNext()$ functions. In our implementation, we also have the next variable of type TreeNode and the explicit stack we implemented using TreeNodes.

The next() function is used to get the next value inside of the collection while the hasNext() function is used to check if there is even a next value that can be obtained. So these functions work together to iterate over the entire collection. The iterator implementation also allows us to use an explicit stack instead of the java stack as asked of us by the assignment.

As we want to use the stack to iterate through the tree we use a recursive function called `FillStackWithTreeNodes()` to put all the values from the tree inside the stack. This function adds all the values in the reverse order as once we go through the stack, the stack always gives us the reverse order when we pop values from it.

The stack is made using a tree where we always add nodes to the right of the root element, thus using the tree as a linked list instead of a balanced tree. Thus we try and make use of the worst-case scenario of a tree to our advantage. This makes it easy to make the explicit stack for this assignment as it's only supposed to contain tree nodes. The stack implementation remains the same as the previous assignments with the only difference being that rather than using the next pointer we use the right pointer of the tree nodes that make up the stack.

4.1 `FillStackWithTreeNodes() Function`

```java
private void FillStackWithTreeNodes(BinaryTree.TreeNode current) {
    if(current.right != null){
        FillStackWithTreeNodes(current.right);
    }
    stack.push(current);
    if(current.left != null){
        FillStackWithTreeNodes(current.left);
    }
}
```

Here we recursively traverse to the furthest point in the right direction from the root and once we reach the largest key inside of the tree or the furthest right node in the tree, we push it to the stack. Then we start doing the same but for the left part of the tree instead. This is implemented in the same way that the print() function in the assignment is done but we go in the opposite order as we want to add items in the reverse order.

4.2 `next()` Function

```
public Integer next() {
    if(!hasNext()){
        throw new NoSuchElementException();
    }
    Integer n = next.key;
    next = stack.pop();
    return n;
}
```

We first check for the existence of the next node and throw a new `NoSuchElementException()` if we don't have any more elements in the stack left to iterate over. Otherwise, we create a temporary variable that is used to store the key of the next node, which we return at the end of the function. Then we pop a node from the stack and set the next node variable as the value that we popped from the stack.

4.3 `hasNext()` Function

```
 public boolean hasNext() {
    return (next != null);
}
```

This function is very simple where we just check for the nullity of the next node variable in the iterator class. This function is called by the next function to check if there is the next element or not and we throw an exception if the hasNext() function returns a false flag.

4.4 Constructor of the TreeIterator Class

In the constructor of the TreeIterator class, we create a new stack every time and Fill it with the tree nodes from the tree we want to iterate over using the tree iterator in the opposite order(as explained before) using the `FillStackWithTreeNodes()` function. After this, we set the next node variable to the node value that we get upon popping the stack.

```
public TreeIterator(BinaryTree tree){
    stack = new Stack();
    FillStackWithTreeNodes(tree.root);
    next = stack.pop();
}
```

We test the TreeIterator class by iterating through all the values inside of a tree using a for-each loop which internally triggers the TreeIterator class. This allows us to retrieve all the values from the tress in ascending order using the stack we created before. We are asked how the iterator will behave in case we add new elements after having retrieved some elements from the tree. The way that I have implemented my iterator makes it so that we add all the elements inside of the tree using the class constructor where we call the `FillStackWithTreeNodes() function` whenever a new iterator is created. This means that if we add new elements after the creation of the iterator it will not be able to retrieve the new elements that have been added as those elements are not part of the internal iterator stack. Thus we will need to create a new iterator whenever we update the tree so as to be able to retrieve all values that are currently available inside of it.

5 Conclusion

The tree data structure is quite useful for being able to search for values, unlike other data structures that we have looked at so far like linked lists which can only perform search operations in linear time. We also compared our lookup function against binary search and we saw that they both operate in the same time complexity. Aside from this, we looked into how we implement a custom iterator for our data structure using an explicit stack which gets allocated on the heap instead of the java stack. We implemented the Java interfaces Iterator for creating the TreeIterator Class and we make the Binary Tree implement the Iterable interface so that we can iterate over it. We then used the iterator to go through all the elements inside of the tree and printed them in ascending order of their key values.