Binary Tree

Jonathan Nilsson Cullgert

Fall 2023

Introduction

In this assignment we are implementing a new linked data structure, a different way to store data. We will be implementing a binary tree. A tree in general is a way to store data by having a root which will be pointing to sets of branches. If a branch is not pointing to anything else, it only has null-pointers, it is a leaf. In this assignment binary trees will be worked on, that is, a tree in which every branch has a maximum of two new branches, leafs or null-pointers. The methods which will be implemented is adding, lookup and printing. The constructor for the binary tree will also be added. The remove method will also be studied but not implemented.

A binary tree

In the binary tree we will start by creating a binary tree, as well as a constructor for nodes. The nodes in this case will have a key, a value as well as a branch to the right and left. In this particular assignment the key and value will both be integers but they could be of any data type. It is important to note that the key has to be comparable to another key so that the tree can become sorted. The binary tree will only hold the root node, if that node is null the tree is empty. It is the nodes that build up the tree.

The binary tree can be sorted in different way but in this assignment the tree will be sorted so that nodes smaller than the current node will be placed on the left branch and nodes that are bigger will be placed on the right branch. The comparisons to see if a node is bigger or smaller is done with their keys, not the value that the node holds.

The first method to be added is the **add()** method. This method will take in a key and a value and add it to the tree as a node, where it fits, so that the tree is still sorted. If a node with the same key is already present in the tree the value of that node will be replaced with the value from the **add()** method.This method is implemented recursively so I implemented it with a helper method. The method checks if the tree is empty, in that case

the root will become the key and value from the **add()** method. If the tree is not empty the helper method will be called upon.

In the helper method we start by comparing the keys of the current node and the key which will be added to the tree. If they are equal we replace the old value with the new value. Otherwise we check if the key from the new node is lesser or greater than the current node. Depending on which it is we will either go the left or right branch. This step is done recursively until the branch which we will move to is null, in that case that branch is replaced with a new node with the key and value from the **add()** method.

```
private void add(Integer key, Integer value){
    if (this.key == key){
        :
    }
    if (this.key > key)
        if (this.left != null)
            this.left.add(key, value);
        else
            this.left = new Node(key, value);
    else
        :
```

The other method which is implemented the **lookup()** method. This method is implemented very similarly to the **add()** method. It takes a key as an argument and then it searches through the tree in the same way as the **add()** method until it finds the key it is searching for. If it at any point jumps to a branch which is null, the method will return null as it means that the key searched for is not in the tree. Otherwise the method return the value of the node from the searched key.

```
Node cur = this.root;
while (cur != null){
    if (cur.key == key)
        return cur.value;
    if (cur.key < key)
        cur = cur.right;
    else
        cur = cur.left;
}
return null;
```

When constructing a binary tree it is important to not do it with an already sorted set of keys. This will skew the binary tree so that it becomes something similar to a linked list. It is important to have the tree somewhat balanced otherwise the advantages of the tree is lost.

Depth first traversal

If we want to go through the entire tree, how do we do it? Since the tree is already sorted by this point we can start by going through the tree with the leftmost node and eventually reach the rightmost node. In this way the tree is gone through to the depth of the tree as far to left as possible. This is the depth first alternative of going through a tree.

The problem with implementing this traversal recursively which is done in this assignment is that it uses the implicit stack from java. This means that a fixed amount of recursion calls can be used in the function until there will be a stack overflow. If an explicit stack is used the size can be changed so the risk for stack overflow is not as prevalent.

Iterator

The next part in the assignment was to implement an iterator. An iterator has two basic methods, **next()** and **hasNext()** which is the necessities for creating an iterator. An iterator is a data structure which can use any datatype and data structure and with that find the next item and return it if necessary.

The way the iterator is implemented is by using a next node which unsurprisingly keeps track of the next node in the tree. It will also be using a stack to keep track of where in the tree the next node is. Since we are using the same principles as the depth first traversal, we will be pushing nodes onto the stack every time we go further down the tree. When we call the **next()** method we will be popping the stack and the node that we return will be the next item in the tree.

The way I implemented the iterator is by pushing down to the left in the tree. When there is no node to the left we will go down to the right one step, then after that see if there is any node to left. So every time we encounter a leaf the iterator will climb the tree and search for more branches. This will continue until the tree has been completely searched. The **hasNext()** method will just return true as long as a new node exists.

```
public Integer next(){
    Node top = stack.pop();
    partialInOrder(top.right);
    return top.key;}

void partialInOrder(Node node){
    while(node!=null){
        stack.push(node);
        node=node.left;}}
```

Using a stack

So the iterator will be pushing and popping onto the stack to keep track of which nodes has been already returned and which node will returned when **next()** is called the next time. The implementation of the stack in this particular task was quite difficult to do since the way the nodes were pushed and popped had to be thought through thoroughly so that the correct node was actually returned each time **next()** was called.

With this implementation of the stack and the iterator it is possible to find a couple of the next elements in the tree, and then add some more elements to the tree, and finally continue searching and returning next elements in the tree. These elements are of course returned in the order they are supposed to be according ot depth first traversal.

Benchmark

As we can see in 1 the time complexity for the **lookup()** method is resembling something similar to a logarithmic function, $O(log(n))$. When I first tried to benchmark the **lookup()** method it always became linear which did not make sense at all, given that the binary tree was somewhat balanced. At first I thought I was filling the tree in the wrong way, by giving it a similar structure to a linked list which would explain why it took linear time find a node. In the end it turned out that I created the binary tree in the wrong place, outside the for-loop when testing different sizes of binary trees. This means that after the first benchmark for 100 elements more elements were just added onto the same binary tree which gave it that linked list structure the more elements that were added onto the tree.

n	Lookup time
100	36
200	181
400	340
600	530
800	649
1000	1062
1200	1070
1400	979

Table 1: A table which shows how long it takes to search for an element in a binary tree