Binary Trees

Pontus Hector

Fall 2022

Introduction

This report is conducted as the sixth assignment in the course ID1021. In this report one explores the implementation of binary trees and their benefits and drawbacks.

A Binary Tree

The structure of a binary tree is made out of nodes in which the tree originates in a root node which points to two other nodes (also known as branches) which are further pointing to two nodes each, and so on, which altogether combines into a large list in which the structure reminds of a tree. In this task a node is containing a key element, which is used to define each node, a numeric value, and pointers to the branches (the left and right node). There are many benefits to construct the binary tree in a sorted manner to be able to manage and navigate through the tree and this is what the keys are used for. To construct the binary tree a method called add is used. The add function receives a key element and a value element and start to compare the key element to the first node in the tree and if the first node is equivalent to null, a new node is created with the received key element and value.

```
if (current == null) {
            return new Node(key, value);
```

However if a root node already exists, the key element is compared to the key element of that node. If the key element received is less than the key element of the first node, the key element is instead compared to the left branch of the root and if the key element is larger than the key element of the first node, the key element is compared to the right branch of the root. This comparison occurs recursively up until a null node is reached, where a new node is created with the received elements.

```
else if (key < current.key) {
            current.left = addRecursive(current.left, key, value);
            return current;
```

```
        }
        else if (key > current.key) {
            current.right = addRecursive(current.right, key, value);
            return current;
        }
```

If the key element received already exists as a node the tree, the value of the node is updated to the received one.

Lookup

In this program the method "lookup" is created to extract the value of a node with a given key. The implementation of the search algorithm is also recursive and very similar to the method to construct the tree. The method receives a key element and starts to compare it with the key values of the node. If the key element received is less than the key element of the current node, the key element is instead compared to the left branch of the node whereas if the key element is larger than the key element of the current node, the key element is compared to the right branch of the node. This is reoccurring up until a node with the specified key is found, and the value is retrieved, or until it can be deducted that no node with the specified key exists, in which case we return null.

```
if (current == null) {
             return null;
        }
        else if (current.key == key) {
             return current.value;
        }
        else if (current.key > key) {
           return recursivelookup(current.left, key);
        }
        else if (current.key < key) {
           return recursivelookup(current.right, key);
        }
```

This method is then benchmarked and compared to a binary search algorithm operating over arrays.

Iterations	Nodes/array size (n)	Runtime Tree	Runtime Array
1000000	1000	$38ns$	$64ns$
1000000	2000	$71ns$	$74ns$
1000000	4000	$93ns$	$81ns$
1000000	8000	$121ns$	$85ns$
1000000	16000	$150ns$	$93ns$

Table 1: Binary tree compared to binary search. Runtimes are averages.

This benchmark confirms that the binary search (over a sorted array) is slightly more efficient when operating on large data structures. If all you need to do is query from a sorted array, binary search will generally be faster because of the constants involved. However, if the stored values under consideration need to be updated at runtime without breaking their sorted property then the "lookup" method is considerably more attractive since it maintains logarithmic time complexity on average while the binary search algorithm will grow rather inefficient.

Binary tree iterator

The binary tree iterator is implemented as a way to sequentially wander through the node tree and retrieving the keys and/or values of the nodes visited. In this assignment the iterator is implemented to wander through the nodes in order (from the lowest/first key to the highest/last one), depth first. To wander upwards in the node tree, a stack is used upon which all the nodes wandered through are pushed to and later on popped from to retrieve the last visited node. When constructing the iterator the node tree is wandered through to the leftmost position through a method called *moveLeftmost* to access and start on the first key.

```
private void moveLeftmost(Node current)
            {
                while (current != null) {
                    stack.push(current);
                    current = current.left;
                }
            }
```

From there the function *next* is iteratively called for each time one wants to sequentially move further to the next node relative to the key values. This is done by popping the previous node from the stack and check whether it has a right branch and if the previous node has a right branch the method *moveLeftmost* is called with the right branch as its current node.

```
public Node next()
            {
```

```
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Node current = stack.pop();
        if (current.right != null) {
            moveLeftmost(current.right);
        }
        return current;
    }
```

The method *hasNext* is basically only used as a reference which returns true if the stack is empty and there are no further nodes to visit. This implementation was then benchmarked to conclude the time complexity of iterating through the entire node tree using the iterator. The following measurements were obtained:

Iterations	**Number of nodes (n)**	**Average runtime**
1000000	100	3.7μs
1000000	200	7.4μs
1000000	400	14.3μs
1000000	800	27.9μs
1000000	1600	57.4μs
1000000	3200	118μs

Table 2: Iterative walk through binary tree

This implementation for iterating through the binary tree is evidently of linear time complexity O(n).

With the implementations above it is possible to create an iterator, retrieve some values and then add more nodes to the node tree and continue the iteration. However the new nodes data will only be returned by the iterator if the new nodes key values are larger than of the currently visited node in the iterator. This is since the iterator will not revisit previously visited nodes (nodes with lesser key values) even if we add new nodes to the binary tree.