Trees

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the sixth assignment "Trees" on the course "ID2021" TIDAB, KTH. To further investigate the area of linked data structures this assignment is about tree structures and specifically *binary trees*. A tree structure is when we similarly to a linked list have an origin but it can then separate into several branches. In the case of a binary tree always two branches. In this report we will look into and analyze the operations of constructing a tree, adding nodes and searching for an element. The base structure for the *binary tree* and its internal class *Node* is given in the assignment and the task also involves going trough the list at first with the internal java stack and printing the elements in order and then using the stack we created during assignment five with linked lists. To successfully go through the elements with our own stack we will use the built in *iterator class*.

Creating a *binary tree*

The first task is to compare the lookup algorithm for *binary tree* with binary search and then bench-mark the execution time for a growing data set for the lookup algorithm. Once the *binary tree* is created it has already split the list into halves where one value is bigger the lookup operation then compares them to each other. The binary tree combined with the lookup algorithm is very similar to binary search. However if the binary tree is not balanced it might be a bit worse but still within the same time complexity.

Table 1 shows the bench-marks for the run time on a growing data set first with non ordered keys and then with ordered keys.

Size(n)	**10**	**100**	**1000**	**10000**	**100000**
unsorted(ns)	150	200	250	400	1650
sorted(ns)	1500	3800	5900	10000	

Table 1: Mesurements of the run-time as a function of *binary tree* size. Run-time in nano seconds.

We can see from the result that similarly as with binary search we get a time complexity of $O(log(n))$. But when the keys are ordered we just continue to adding in the same direction since it will always be larger or smaller depending on how we have ordered it. We will then instead get a time complexity of O(n) which is much slower.

Explicit stack and iterator

The next part of the assignment is to create an iterator using an explicit stack, in this case we will use the one that was created for assignment 5, linked lists. To be able to use this stack we will change linked lists to carry a *binary tree* node instead of an integer value. The code for this stack implementation is shown below:

```
public class Stack {
    LinkedList element = null;

    public void push(BinaryTree.Node node){
        element = new LinkedList(node, element);
    }
    public BinaryTree.Node pop(){
        BinaryTree.Node temp = this.element.head();
        element = this.element.tail();
        return temp;
    }
    public boolean isEmpty(){
        if (element == null){
            return true;
        }
        else{
            return false;
        }
    }
}
```

This stack is the tool that will help us to traverse backwards through the tree. The iterator will have a function that firstly goes as far as it can left

(smallest value) it then starts popping values, thus going back up the tree and checking if they have a value on the right or if it is null. If there is a value on the right we will go as far left we can down that route and repeat. This is implemented through the iterator methods that overrides the ones already defined in java. This way we are able to use the functionality of an iterator and make use of existing code that utilizes *hasnext* and *next*. The code for these implementations is shown below.

The function that goes down through the tree:

```
private void goLeft(BinaryTree.Node cur){
        while (cur != null){
            stack.push(cur);
            cur = cur.left;
        }
    }
```

Has next and next:

```
    @Override
    public boolean hasNext(){
        return !stack.isEmpty();
    }
    @Override
    public Integer next(){
        if (!hasNext()){
            throw new NoSuchElementException();
        }
        BinaryTree.Node cur = stack.pop();

        if (cur.right != null){
            goLeft(cur.right);
        }
        return cur.key;
    }
```

Through this we will be able to add any number of unsorted keys and have them returned in the correct order. If we create an iterator and retrieve a few elements but then add new elements, depending och where they end up in the tree we could then miss them if we have already passed those branches on the tree.

In conclusion a binary tree seems to be great way of storing data after a hierarchy and then manipulating and accessing it when needed. In the case of the doubly linked list we had the address to all nodes which will usually not be case, a binary tree might then be better in some cases.