Trees

Axel Månson Lokrantz

Spring Fall 2022

Introduction

The next steppingstone from linked lists is the tree structure. A tree is a non-linear data structure where objects are organized in hierarchical relationship. The name comes from the structure itself, where the topmost point of a tree is called root. The root divides into branches and the last node of a branch is called a leaf. There are several different tree structures such as AVL trees, B-trees and Binary trees. In this assignment we worked with the binary trees. A binary tree is non-linnear structure where a node can be connected to at most two descendant nodes (child nodes). The program was written in Java together with Pontus Kinnmark.

Task 1: Binary search tree

A binary search tree is a rooted binary tree with the key of each node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. For example, a tree with a root which holds the key value 5 will pass down key values less than 5 to the left and key values greater than 5 to the right. The first task was to implement two methods add and lookup. The add method adds a new leaf to the tree that maps the key to the value, if the key is already present in the tree we instead update the value of the node with the corresponding key.

```
public void add (Integer key, Integer value){
    Node temp = root;

    while (temp != null){
        if (temp.key == key){
            temp.value = value;
            return;
        }
        if (key < temp.key){
            if (temp.left == null){
                temp.left = new Node(key, value);
                return;
            }
            else
                temp = temp.left;
        }
        if (key > temp.key){
            if(temp.right == null){
                temp.right = new Node(key, value);
                return;
            }
            else
                temp = temp.right;
        }
    }
    root = new Node(key, value);
}
```

Add function for binary search tree..

A temp node is created which we assign to root. If our root has the searched key, we update our root value and return. Through the if statement we continuously evaluate if the keys value is greater or less than the the child nodes and traverse down the tree. If we key is greater or smaller than a leaf node we create a new node and return.

The lookup method had to find and return the value associated to the key. If the key was not found it should return null.

```
public static Node lookup(Node root, Integer key){

    // Base Cases: if root is null or the searched key.
    if (root == null || root.key == key){
        return root;
    }
    // Key is > than root's key.
    if (root.key < key){
        return lookup(root.right, key);
    }
        // Key is < than root's key
        return lookup(root.left, key);
    }
```

A base case is made, if the tree is empty or the searched key is our root we return the root. If the root key is less than the key input we use recursion which is a process in which a method calls itself continuously, moving the root further down the tree to the right. The same procedure repeats if the key is less than the root's key.

n	BS	lookup
100	50	250
200	55	270
400	65	300
800	75	320
1600	85	330
3200	100	350
6400	125	380
12800	160	430

Table 1: Benchmark for binary search on an array of n elements vs. the lookup function on a binary search tree with n number of nodes.

.

When constructing a benchmark for the tree structure it was important that we not only randomized the keys, but also avoided duplicates which would have resulted in a shorter tree since the height would have been n minus all duplicates. Therefor we implemented the following constructur:

```
public BinaryTree(int n){
        Random rnd = new Random();
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
        list.add(i);
        }
        Collections.shuffle(list);
        int[] arr = list.stream().mapToInt(i -> i).toArray();
        for (int i : arr){
            add(i, rnd.nextInt(n));
        }
    }
```

Best case is when we get to search for the root, in which case we have to make one comparison so the time taken would be constant $O(1)$. Average case, height becomes $log(n)$ where n represent the number of nodes in the tree. For example, if we find what we are looking for in the second level we have done 2 comparisons, third level 3 comparisons and so forth all the way up to n. The time complexity therefor becomes $O(log(n))$. Worst case occurs when all keys are added in a sorted order, starting from lets say 1,2,3 ... n our tree would be skewed. No nodes would have been available in the left sub tree and since time taken is the same as the height of the tree the time complexity becomes $O(n)$.

Task 2: Itterator and a stack

For task two we were asked to implement an iterator to go through all nodes utilizing a stack. We chose to build a dynamic stack, which means it can collapse or expand depending on the amount of data that is being transferred.

```
private void resize(int amount) {
        BinaryTree.Node[] newStack = new BinaryTree.Node[amount];
        for (int i = 0; i < this.sp; i++) {
            newStack[i] = this.stack[i];
        }
        this.stack = newStack;
    }
    public void push(BinaryTree.Node item){
        if(this.sp == size()){
            resize(2*size());
        }
        this.stack[this.sp++] = item;
    }
    public BinaryTree.Node pop(){
        BinaryTree.Node poppedNode = this.stack[--this.sp];
        this.stack[this.sp] = null;
        if (this.sp > 0 && this.sp == size()/4)
            resize(size()/2);
        return poppedNode;
    }
```

Above the functions for push, pop and resize can be seen. We used the same logic for our dynamic stack as we did in previous assignments. If the stack is full, we double the stack size and if the stack is $1/4th$ of its size we divide the stack size by 2. A better approach would probably have been to utilize java generics to make the stack work not only with nodes.

```
TreeIterator(BinaryTree.Node root)
    {
        stack = new Stack();
        moveLeft(root);
    }

    private void moveLeft(BinaryTree.Node current)
    {
        while (current != null) {
            stack.push(current);
            current = current.left;
        }
    }
    @Override
    public boolean hasNext()
    {
        return !stack.isEmpty();
    }
    public Integer next()
    {
        if (!hasNext())
            throw new NoSuchElementException();

        BinaryTree.Node current = stack.pop();

        if (current.right != null)
            moveLeft(current.right);

        return current.value;
    }
```

The iterator utilizes next() and hasNext() to perform depth first traversal on the tree. To traverse back to the ancestor once we reached a leaf we use our implemented stack. Consider the following tree, which has the same key values as in the assignment pdf.

```
      5

   2      7

1   3   6   8
```

The iterator is called and inside the constructor we call the moveLeft function with our root as parameter. moveLeft() traverse all the way down to (1) pushing 5,2 and 1 on the stack. (1) is the first element which is fully left from the root. (1) does not have a right child, so the next element is (2) which we just came from. (2) does have a right child, so we iterate the right sub tree. (3) does not have a right child, its parent is (2) which has already been traversed, we pop the last key on the stack (5) which has not been traversed and we stop. (5) has a sub tree on the right, with its left most element (6). We repeat the process until we end up on (8). (8) has no sub tree so we walk up, (7) has already been traversed since came from the right. (5) has already been visited so there is no way to move further up the tree. We have now iterated the whole structure in depth first order from left to right wich yields the following output:

```
next value: 101
next value: 102
next value: 103
next value: 105
next value: 106
next value: 107
next value: 108
```

If we were to create an iterator, call the next() method until we are done iterating the left branch of the tree and then decided to add a node to the left, lets say a key with the value 4 the value would have been "lost". The iterator consider the left branch done and will keep iterating the right branch.