Breadth first search in binary trees in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment I take a look at breadth first search (as opposed to depth first search), meaning that we will traverse the tree one level at a time. After using it to print the entire list, I also implement a lazy method that performs a breadth-first traversal too, but only evaluates it one by one, as the next value is needed. This is particularly useful, when we know we only need to look at the top couple of rows, or if a branch may be infinite.

Implementation

For the implementation, we will make use of the (doubly linked) queue I implemented for assignment 5. I had to alter it a little so that it holds items of `BinaryTree.Node`, the `enqueue()` method can add a node to the end, while the `dequeue()` method can return the Node from the front of the queue.
The idea is to use the queue while printing the current level to store the references to the next level. It is a similar strategy to what we had before for the width-first search, but now we need to use a queue instead of a stack so that we keep going from left to right through each level.
We were asked to first draw it on paper and try and understand what needs to be implemented. I went through the tree on the right on Figure 1, showing the output as well as what is supposed to be stored in the queue at that moment. Based on this drawing, I could add a `print()` method to my binary tree that prints the values in a breadth-first manner.

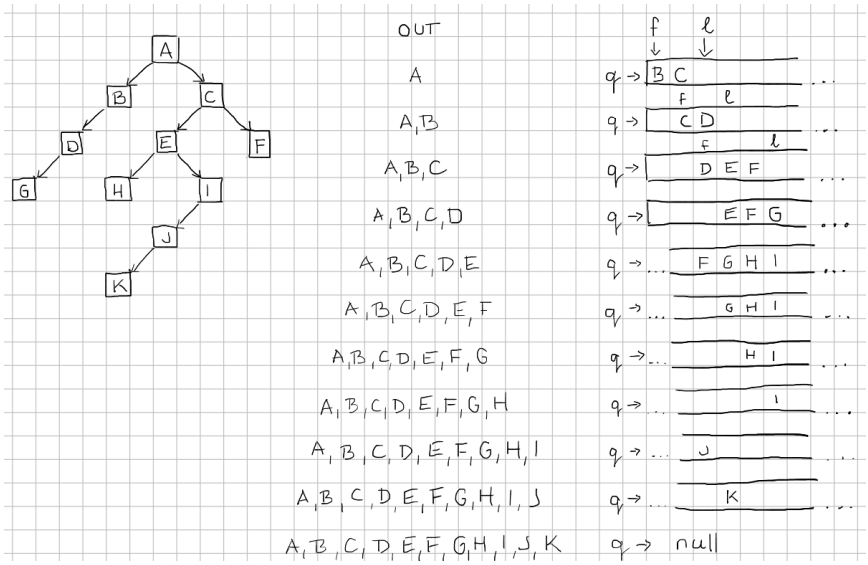

Figure 1

The same idea in actual code:

```
public void print() {
    DoublyLinkedQueue q = new DoublyLinkedQueue();
    Node current=this.root;
    while(current!=null){
        System.out.println(current.value);
        if(current.left!=null){
            q.enqueue(current.left);
        }
        if(current.right!=null){
            q.enqueue(current.right);
        }
        current=q.dequeue();
    }
}
```

As you can see, for every node, we need to print it but then before moving on, we save its `left` and `right` branch to our queue (if they exist) to be able to pick them back up once we're done with the rest of the level. The dequeue operation then returns the node we need to handle next, so we set current equal to that and continue.
Important to note, that if the tree was empty, nothing is printed. We could've of course handled this separately at the beginning, if we wanted to e.g. print a message for the user clarifying that the tree is empty.

a lazy sequence

The purpose of a lazy sequence is to allow printing only, say, the first x elements (and then maybe perform some kind of operation on them before printing another y elements, etc). This way we may be able to avoid evaluating certain things, that are not going to be used.
I created another class called Sequence that could hold a (doubly linked) queue. It has a constructor that can create a new queue and place the root in there; and an `Integer next()` operation, that prints the next value and updates the queue so that we can access all elements afterwards. My code is shown below:

```
//in class Sequence
public Sequence(BinaryTree.Node root){
    this.q = new DoublyLinkedQueue();
    this.q.enqueue(root);
}

public Integer next(){
    BinaryTree.Node r = q.dequeue();
    if (r==null) return null;
    else{
        if (r.left!=null) q.enqueue(r.left);
        if (r.right!=null) q.enqueue(r.right);
        return r.value;
    }
}

//in class BinaryTree
public Sequence sequence(){
    return new Sequence(this.root);
}

//now it can be called as such:
BinaryTree bt = new BinaryTree();
(...) //adding some elements to the tree
Sequence seq = bt.sequence();
System.out.println(seq.next());
```

Problems

If we were to add or delete elements (in between printing) to/from nodes that have already been evaluated, we would have no guarantee that what we're printing is still correct. Since we don't know which level the addition

will fall in, it would be rather difficult to determine whether it would cause any kind of contradiction between our tree and what we're printing, same with deletion. So the main problem is that this lazy sequence of ours is neither a sort of "snap-shot" taken at the time of its creation, nor is it an up-to-date version. For now, I think the best idea is to just restart the whole sequence if we need to change the tree.

Summary

In this assignment, I learnt about breadth-first search in binary trees using a queue I previously created, and even implemented a lazy sequence method for printing a tree one-by-one. I also recognise this latter ones uncertainties when the tree is modified, but I leave the question unsolved for now.