Assignment 9 Report - Breadth-first Search in Java

Dean Tsankov

October 12, 2024

Introduction

The purpose of this report is to present the implementation of a breadth-first traversal algorithm over the tree structure from the previous assignment. In that said assignment we used a depth-first traversal method to do operations on the structure. With that algorithm we go as far down into the structure as possible, but it might be that the operation we would like to perform, let us say search for an element, does not require us to go too deep or we have some notion that the element we are searching for is closer to the root. In such a case we would rather traverse the tree in terms of layers. In essence go through the branches by degrees of distance from the root. This also prevents us from entering an infinite branch (when we have no way of knowing the amount of branches going down from a node) and proceeding through it for an unnecessary arbitrarily long amount of time.

With BFS we will heavily rely on a queue structure in order to perform our traversal as opposed to the stack used in DFS.

The layer approach

The steps which we have to take in order to execute the breadth-first traversal are not too complicated, but require some consideration. As the queue is used to keep track of which nodes have yet to be traversed and in what order, it is imperative to know in what way to enqueue nodes. The main idea goes as follows:

- We check if the queue is empty if it is not we continue on, if it is we are done.
- We dequeue an element from the queue and for the purposes of a print function we print its value. Otherwise we might choose to perform a different action on the element.

- We then enqueue both of this item's left and right nodes, if it has any, to the queue.
- Finally, we go on to repeat this process until we exit at the first step.

We can certainly notice that following these steps we structure the queue in order of the "layers" of the tree and simultaneously perform operations on each element in that order. Bellow is a sketch of the way I understand how the algorithm works on a tree. The process is not showed as finished as it would become too convoluted to understand, but I hope it is enough to get the general point across:

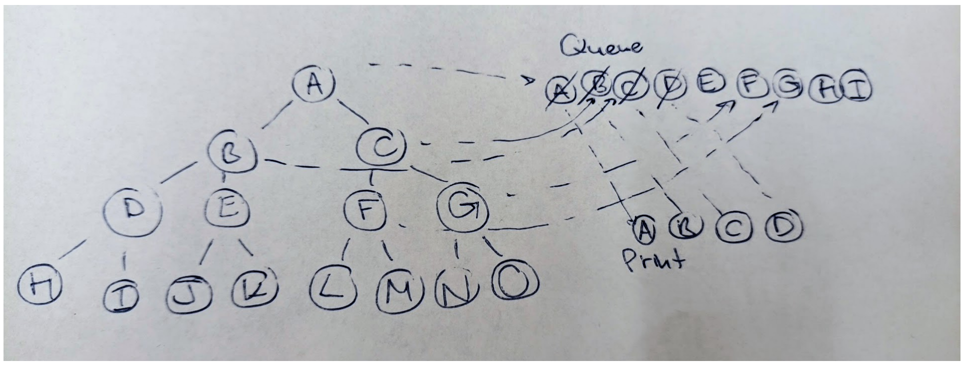

Figure 1: Breadth first traversal paper schematic

Coding the method

In terms of the code for the function as stated before it was not particularly hard to implement. One requirement which had to be considered was that the queue used needed to be able to work with the `Node` class the tree uses. Or alternatively defined to work with a generic type passed on initialisation of the object. Thankfully, I had the foresight to have already defined my array queue implementation with a generic type held value. That leaves only following the above described steps of the traversal.

```
(...)
public void printBFS() {
    ArrayQueue<Node> que = new ArrayQueue<Node>();
    Node itr = this.root;

    que.enqueue(itr);

    while (!que.isEmpty()) {
        itr = que.dequeue();
```

```
            System.out.println(itr.value);
            if (itr.left != null) {
                que.enqueue(itr.left);
            }
            if (itr.right != null) {
                que.enqueue(itr.right);
            }
        }
    }
    (...)
```

I first enqueue the root of the tree and begin the loop with printing the first element in the queue as well as dequeue-ing it and then adding its two children, if they exist, in the structure.

If we would like to test this implementation with a similar example to the one in my sketch the tree would have to work with key-value defined nodes since otherwise as it is the characters of the alphabet would not be structured in the way they are, but would rather yield a very unbalanced tree. I did however test how the function performs with standard integer inputs and it does in fact print the tree in order of the layers.

A scheduled sequence

The breadth-first traversal works as intended, but we could notice that in this way we will always go trough the whole tree without having control when a next node should be accessed. It will give us a lot of flexibility to be able to perform this.

A way we can do that is by introducing a new data structure which holds the queue responsible for keeping track of the crawl over the tree. From this structure we would request the next element in the order and this way gain control over when it happens.

There would also have to be a method in the tree class itself where we work with this structure. Here is one method which in essence does the same print functionality, but could easily be exchanged for a more specific task.

```
    (...)
    public Sequence sequence() {
        Sequence res = new Sequence<T>(this.root);
        while(!res.internalEmpty()){
            System.out.println(res.next());
        }
        return res;
    }
    (...)
```

The sequence class takes a single argument on initialisation and that is the beginning of the tree or even sub-tree we would like to work over. Next is the class definition itself:

```
(...)
public class Sequence<T> {
private ArrayQueue<BinaryTree.Node> que;

public Sequence(BinaryTree.Node root) {
    this.que = new ArrayQueue<BinaryTree.Node>();
    this.que.enqueue(root);
}

public T next() {
    BinaryTree.Node itr;
    if (!this.que.isEmpty()) {
        itr = que.dequeue();
        if (itr.left != null) {
            que.enqueue(itr.left);
        }
        if (itr.right != null) {
            que.enqueue(itr.right);
        }
        return (T) itr.value;
    } else {
        return null;
    }
}

public boolean internalEmpty() {
    return que.isEmpty();
}
(...)
```

The `next()` function is an almost exact copy of the BFS algorithm from the previous section. Something that raised a concern in me, though, is that in order to work with both the Node and its left and right references here, their access modifiers had to be set to public. As I understand this is not a good practice, but I did not see a different way to expose them to an outside class otherwise, and unless I am misunderstanding the task altogether, we are required to have access to both of these properties.

It appears that the solution to my concern was to have the `Sequence` data structure not as an outside class, but as a sub-class to the `tree` structure. This arrangement allows us to keep the private modifiers of the Node class and still call each next node subsequently. we also do not particularly use any functionality since only a tree structure would be able to make use of the `Sequence` class anyways.

Sufficed to say now we do have a method call which returns the exact next value in the tree when we want it.

If in between calls of the `next()` function there were elements added to the tree we would get the following outcomes:

- If the current first element in the queue to be removed is so to speak above the "layer" where the addition happened then the same would have no problem to be reflected in the subsequent return values, since we are simply working with references and not deep links.
- If the addition itself was somewhere above the layer where the current first element in the queue lies, then we would have no way to continue our sequence and expect correct results, in which case we should initialise it again and then the changes would be reflected.

Similar notions apply to a case where we dynamically remove an element from the tree.