Breadth First Search

Emanuel Paraschiv

Fall 2024

Introduction

The scope of this assignment is to implement the Breadth First Search algorithm for a Binary Tree. This algorithm appears as a solution for some more interesting cases. For example, if the Node that we want to find is situated closer to the root or if some branches are infinite a depth first algorithm can be inefficient.

The algorithm

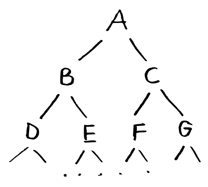

Figure 1

This drawing represents a simple example that we are going to build our algorithm on. The tree has root "A" with children "B" and "C" and so on. Our goal is to build a method that will return the nodes of this Tree in alphabetical order.

It will be easy to print "A" followed by "B" and "C", but the problem arises as we go further down in the tree. How can we keep track of all the Nodes? More importantly, let's say that we reached the end of the row, how can our algorithm jump (from right) to the beginning of the next row (to left)?

The answer to this is to make use of another data structure, one that we have implemented in one of the previous assignments, the `Queue`.

In order to use it in our experiment however, we need to make it of generic type. This way we can adapt it to work with Tree Nodes instead of integers as we did before.

The main idea is the following: we need to use the queue to help us keep track of what element we need to print.

As one may expect, we will start from the root ("A"). First thing we will do is enqueue it:

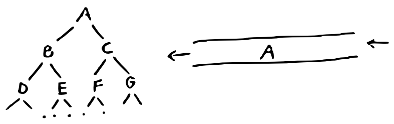

Figure 2

Once the root is enqueued we will repeat a simple procedure that will stop only when the queue is empty. We will dequeue one element at a time, in this example "A" and print its value. Immediately after that, we will enqueue its children, "B" and "C":

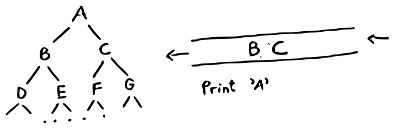

Figure 3

Now we can continue and do the same thing again, from the beginning. First we dequeue one element (now "B"), print it and immediately after that, enqueue its children ("D" and "E"). The mechanism of the queue (FIFO) ensures that the elements will keep their order:

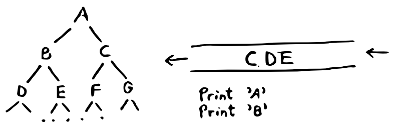

Figure 4

If we continue further, we will print "C" and enqueue "F" and "G" such that the queue will contain: $\leftarrow |"D","E","F","G"| \leftarrow$.

The great thing with this implementation is that it will work no matter the depth of the tree.

Coming back to the implementation in Java, we have to adapt some aspects first. In the beginning we need a queue that accepts Tree Nodes, therefore I made the "QueueImproved.java" class generic (Different than Queue.java in a previous assignment because it has 2 pointers, `head and tail` so the enqueuing operation will be faster). After that, we needed some way to check if the queue is empty. Before we could simply return `head == null`, but as this field is private, I created a method called `isEmpty()` that does exactly this, and we can call it in the `Tree` class.

```
public void bfs(){
        // create queue
        QueueImproved<Node> q = new QueueImproved<>();
        if(root==null){
            throw new IllegalArgumentException("Tree doesn't exist");
        }
        q.enqueue(root);
        Node val;

        while(!q.isEmpty()){  // repeat until there's nth left
            val=q.dequeue();
            System.out.print(val.value+ "; ");
            if (val.left!= null) {
                q.enqueue(val.left);
            }
            if (val.right!=null) {
                q.enqueue(val.right);
            }
        }
    }
```

As we can see above, first we create a queue to store the Nodes. We check is the Tree is empty, case in which we throw an exception. Then we enqueue the root and start the process of searching. As long as we have elements in the queue, we do the following:

- Dequeue the first value and print it;
- If we have one child to the left, we enqueue it;
- If we have a child to the right we enqueue it too;

As we can see, the algorithm has a simple way of functioning. The only trick is to understand the importance of the queue. Notice also that it will work also when the tree is unbalanced: if there are elements only to the right branch and none to left, they will still be printed in order.

The lazy sequence

Now we have a method that will print the whole tree, but what if we don't want that? What if we only need to print a number of Nodes, then stop to take a break and continue after?

To solve this, we can implement a new data structure, a `Sequence`. This part was the most complicated to implements as now I had 3 different classes, all generic that need to access one another.

To start with, we need a new class `Sequence`. The sequence data structure will consist of a queue that will store Tree Nodes.

We need to implement a method that will return a sequence. It works in the following way: We create a queue, we add (if not empty) the root to it and we return this queue in the form of a sequence:

```
QueueImproved<BinaryTree<T>.Node> q =
new QueueImproved<BinaryTree<T>.Node>();
if (root != null) {
    q.enqueue(root);
}
Sequence<T> sequence = new Sequence<T>(q);
return sequence;
```

Of course, we could add the whole tree instead of just the root, but that is not the point. Now we have a sequence that takes every node if iterated, but we need one way to get the value of each node in Breadth-First-Search manner. To come up with this, we can start from the original BFS algorithm implemented earlier. We need to change it to return just one need at a time, and in order to do that all we need is to get rid of the `while()` loop.

However, one problem arises. We can not access the private fields of the Tree from the `Sequence` class, and one way to solve it is using getter methods (`getR()`, `getL()`, `getV()` for right, left and value).

```
public T next() {
    if (queue.isEmpty()) {
        throw new IllegalArgumentException("Tree doesn't exist");
    }
    BinaryTree<T>.Node currentNode = queue.dequeue();
    if (currentNode.getL() != null) {
        queue.enqueue(currentNode.getL());
    }
    if (currentNode.getR() != null) {
        queue.enqueue(currentNode.getR());
    }
    return currentNode.getV();
}
```

Now if we create a sequence that accepts Strings, then we can display each element at a time using a print statement:

```
System.out.println(sequence.next());
```

If we need to print `n` elements at a time, we can create a while loop that iterates until `n`.

After testing it, we can talk about the following scenarios:

- If we add an element after we create the sequence, the situation may vary. If it is far away than the present node then it should technically not create any problem. It will be a problem however if it is closer, because the children nodes of the current dequeued node are enqueued, and the program will jump over it.

- If we delete a node after we create the sequence it will be the same case. I tested removing ”J” which resulted in no error, the sequence just printed everything but J. On the other hand when I tried removing ”B” then the compiler threw an Exception.

Conclusion

Therefore we can draw a conclusion. Breadth First search is a great algorithm that we will find more useful possibly in the future when working with graphs (that can have infinite depth).

Apart from implementing this useful algorithm we also found a way to use it for a smaller sample of a whole tree using sequences. This can prove advantageous when we are dealing with trees with lots of nodes in situations when we need to get only a certain part of that tree.