Breadth-first search

Algorithms and data structures ID1021

Johan Montelius

Fall 2024

Introduction

As an an alternative to our depth-first search strategy in the previous assignment we will now implement a *breadth-first* strategy. Ina breadth-first strategy we traverse a tree one level at at time; it is very useful if we know that the item that we are looking for should be close to the root or if we fare that a branch might be infinite (how is this possible?). When implementing the strategy we will make use of a queue so refresh your memory and start by looking up your implementation of a queue.

one level at a time

Get a pen and a paper and draw a binary tree with the root `A` linked to tow nodes `B` and `C`. The node `B` is linked to nodes `D` and `E` and `C` to `F` and `G`. Draw also the fourth level and mark the nodes: `H` ... `O`.

If we traverse this tree using a depth-first strategy the order would be anything but alphabetic. If we however traverse it in breadth-first order, we will visit the node in alphabetic order: `A`, `B`, `C`, `D`, `E` .... This is not as easy as it i looks; or rather it looks easy if you have the paper in front of you but if you try to implement it it becomes difficult.

Assume we start in the root of the tree; finding the left and right branches is of course trivial and exploring this level is of course not a problem. The problem arise if we now follow the left branch to explore nodes `D` and `E`. After having seen `E` we need to get back to node `C` and even worse, after having seen `G` we should continue with `H` ... how do wee keep track of this?

The trick is to use a *queue*; we always select the first node to visit from the queue and add the children of this node to the queue. Give it a try using the pen and paper before starting your implementation. Begin with an empty queue and then add the node `A`. Now:

- if the queue is empty, we are done, if not,

- take the first node from the queue and print its value
- add the left and right branches to the queue if they exist,
- repeat.

Try with different sized trees, make sure that you understand how the queue is used before starting the implementation. Include a small picture of your pen drawn picture in the report.

the queue

In the assignment when you implemented a queue you might have implemented a queue that could hold only values of type `int`. You need to change this so you have a queue that can hold references to nodes of the tree. This should be very similar to how you adapted the stack in the depth-first assignment.

Once you have a queue that works it should be surprisingly simple to implement the breadth-first traversal. If it works you have all the pieces you need to solve the final task.

a lazy sequence

What you have now is a procedure that traverses the tree in a breadth-first order. The problem is that it will traverse the whole tree but you might want to only traverse part of the tree depending on some criteria. You task is now to implement a new data structure that holds the position in the traversal so that you can request the next elements in the sequence one by one.

Implement a method `Sequence sequence()` of three that returns a data structure that holds a queue (the queue that you have implemented). The class `Sequence` should now provide a method `int next()` that returns the next value in the sequence and updates the queue so that all other values can eventually be returned.

If this works you should be able to construct a tree, extract the first three values, take a break and then extract another two. What happens if you add values to the tree in the break? What could happen if you removed values?