Challenge 2
Cartesian Trees

VerifyThis at ETAPS 2019
Organizers: Claire Dross, Carlo A. Furia,
Marieke Huisman, Rosemary Monahan, Peter Müller

6–7 April 2019, Prague, Czech Republic

How to submit solutions: send an email to `verifythis19@googlegroups.com` with your solution in attachment. Remember to clearly identify you, stating your group's name and its members.

This challenge[1] contains two parts. We do not expect participants to complete the whole challenge in an hour and a half; you can chose the part you like best/which fits your language of choice best. The first part of the challenge is the easiest, it may be interesting to have a look at it first. Note that you do not need to actually implement and verify the algorithm of the first part to try the second.

1 Part A: All Nearest Smaller Values

For each position in a sequence of values, we define the nearest smaller value to the left, or left neighbor, as the last position among the previous positions that contains a smaller value. More precisely, for each position x in an input sequence s, we define the left neighbor of x in s to be a position y such that:

- $y < x$,
- the element stored at position y in s, written $s[y]$ is smaller than the element stored at position x in s,
- there are no other values smaller than $s[x]$ between y and x.

There are positions which do not have a left neighbor (the first element, or the smallest element in the sequence for example).

We consider here an algorithm which constructs the sequence of the left neighbors of all the elements of a sequence. It works using a stack. At the beginning, the stack is empty. Then, for each position x in the sequence, pop from the stack all the positions until a position y if found such that $s[y]$ is smaller than $s[x]$. If such a position exists, it is the left neighbor of x, otherwise, x does not have a left neighbor. Then, push x on the stack and go to the next position. Here is the algorithm in pseudo-code:

[1] The topic of this challenge was suggested by Gidon Ernst.

```
for every position x in s do
    while not my_stack.is_empty && s[my_stack.peek] >= s[y] do
        my_stack.pop
    done

    if my_stack.is_empty
        left[i] <- 0
    else
        left[i] <- my_stack.peek
    fi

    my_stack.push (x)
done
```

Note that the algorithm above assumes that **indexes start from** 1, and hence it uses 0 to denote that a position has no left neighbor.

As an example, let us consider the sequence $s = [4, 7, 8, 1, 2, 3, 9, 5, 6]$. Here is the sequence of the left neighbors of s using indexes that start from 1: $\texttt{left} = [0, 1, 2, 0, 4, 5, 6, 6, 8]$. The left neighbor of the first element of s is 0 (denoting no valid index), since, as it is the first element of the list, it has no smaller element at its left. It is the same for the fourth element (1), as it is the minimum of the list.

Tasks

Implementation task. Implement the algorithm to compute the sequence of left neighbors from an input sequence. You can use an existing implementation of stacks or use directly a sequence for the stack.

Verification tasks.

1. Verify that, for each index i in the input sequence s, the left neighbor of i in s is smaller than i, that is $\texttt{left}[i] < i$.
2. Verify that, for each index i in the input sequence s, if i has a left neighbor in s, then the value stored in s at the position of the left neighbor is smaller than the value stored at position i, namely, if $\texttt{left}[i]$ is a valid index of s then $s[\texttt{left}[i]] < s[i]$.
3. Verify that, for each index i in the input sequence s, there are no values smaller than $s[i]$ between $\texttt{left}[i] + 1$ and i.

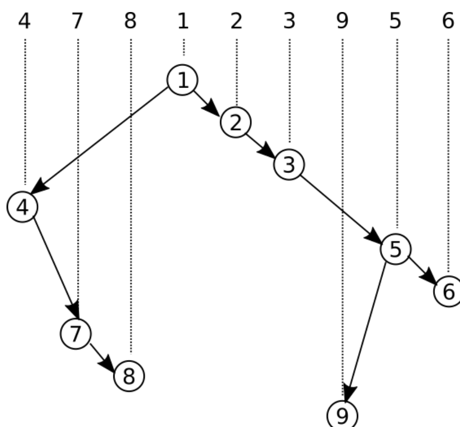
2 Part B: Construction of a Cartesian Tree

To a sequence of *distinct* numbers, we associate a unique *Cartesian* tree. It is a tree constructed so that:

1. The tree contains exactly one node per element in the sequence.
2. The elements are encountered in the order of the sequence when traversing the tree in-order, that is, using a symmetric traversal (first traverse the left subtree, then the node itself, and finally the right subtree).

3. The tree has the heap property, that is, each node in the tree contains a value (not an index) bigger than its parent.

Here is the Cartesian tree constructed for the sequence $\{4, 7, 8, 1, 2, 3, 9, 5, 6\}$:



There are several algorithms to construct a Cartesian tree in linear time from its input sequence. The one we consider here is based on the all nearest smaller values problem (part A of this challenge). Let's consider a sequence of distinct numbers s. First, we construct the sequence of left neighbors for the elements of s using the algorithm described above. Then, we construct the sequence of right neighbors using the same algorithm, but starting from the end of the list. Then, for every position x in the sequence, the parent of x is either:

- The left neighbor of x if x has no right neighbor.
- The right neighbor of x if x has no left neighbor.
- If x has both a left neighbor and a right neighbor, then it is the one which has the larger value.
- If x has no neighbor, then it is the root.

Tasks

Implementation task. Implement the algorithm for the construction of the Cartesian tree. You can implement the tree structure as you prefer, using for example a recursive data-type, a pointer based structure, or a bounded structure inside an array depending on your input language.

Verification tasks.

1. Verify that your algorithm returns a well formed binary tree, with one node per element (or per position) in the sequence.
2. Verify that the resulting tree has the heap property, that is, each non-root node contains a value larger than its parent.
3. **Optional task (advanced)**: Verify that an in-order traversal of the tree traverses elements in the same order as in the sequence.