Data Structures - Assignment no. 2, March 7, 2007

Remarks:

- **Write both your name and your ID number very clearly on the top of the exercise. Write your exercises in pen, or in clearly visible pencil. Please write *very* clearly.**
- **Recall that 80% of the theoretical exercises must be submitted. The exercises can and must be worked on and submitted alone.**
- **Give correctness and complexity proofs for every algorithm you write.**
- **For every problem, find the most efficient algorithm. A non efficient algorithm will be considered as an incomplete answer.**
- **For every question where you are required to write pseudo-code, also explain your solution in words.**

1. In the lecture you learned how to implement a deque using two stacks (and a temporary stack during split) such that the amortized time of each operation is $O(1)$. Recall that when one stack ran out of elements, our implementation performs the operation SPLIT which moves half of its elements into the other stack. Now we ask you to describe and write pseudo-code for a similar implementation, where the operation SPLIT moves one third of the elements of the non-empty stack to the empty stack. Prove that the cost of m operations is still $O(m)$.

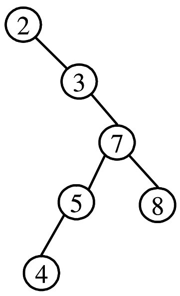
2. Insert the numbers 10,1,9,6,0,6.5 into the top tree, in order from left to right. Which of the three depicted trees is the final result? Only give a final answer, do not explain.

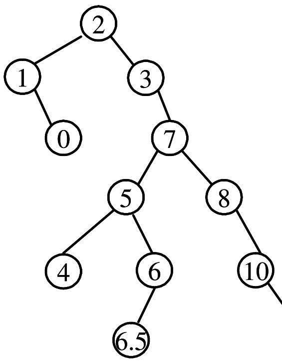
3. You are given a binary search tree, where each node contains a pointer LEFT to its left child, a pointer RIGHT to its right child, a pointer PARENT to its parent, and a key KEY. The keys are integers that can be positive, negative or zero. Describe a procedure that performs the following operation, and then write pseudo-code for it: The operation is given a pointer to the root of the tree, and needs to output the number of nodes who have the property that the sum of the keys in the node's sub-tree is positive. The procedure should take $O(n)$ time, where n is the number of nodes in the tree.
   Hint: Recursion may help.

4. **De-amortization.** In the lecture you learned the "doubling" method that allows to implement a stack using an array without placing a limit on the size of the stack, such that the amortized complexity of each operation is $O(1)$. The method is that every time the array gets full, a new array is allocated whose size is twice the size of the old array, and the old array is copied to the new array.

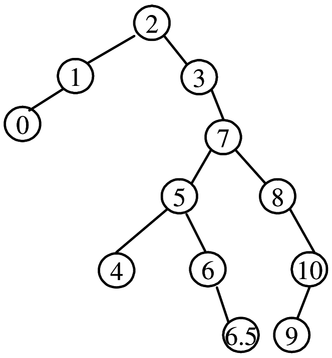
   In this question we ask you to *de-amortize* the doubling technique. Specifically, give an implementation of a stack using an array, such that the size of the stack can increase as much as needed (obviously, the array should increase its size once in a while), but the running time of each operation should be $O(1)$ *in the worst case*. We assume that for any value k, you can allocate an array of size k in one time unit. However, this array is initialized with arbitrary values: copying the old array to the new array still takes time linear in the length of the old array.
   Hint: The data structure should maintain two arrays simultaneously.

5. **Challenge question. Not to be submitted.** In this question you will implement a redundant counter. A redundant counter is a binary-like counter, stored in an infinite array of "bits", which

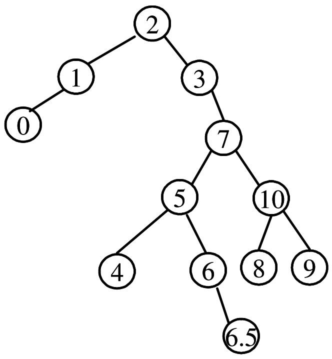




(a)



(b)



(c)

supports the operation *increment*() which adds 1 to the value of the counter. Instead of bits, the digits of the counter are 0, 1, and 2, but the counter is still binary: for example, "2012" represents $2 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 2 \times 2^0$. Thus, every number has more than one possible representation.

(a) Implement a redundant counter as above, such that each *increment*() operation changes only $O(1)$ "bits" *in the worst case.*

(b) Improve the implementation so that you can perform an increment operation in $O(1)$ time in the worst case. You are allowed to store a pointer that can point to a location in the array.

(c) If you solved this and you still want a challenge, try to design a redundant binary counter that supports both increment and decrement operations, both in time $O(1)$. This time you might want to allow even more possible digits, for example $-1, 0, 1, 2$.