Data Structures - Assignment no. 2, January 28, 2008

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. **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.

3. In class you had seen an implementation of a stack using an array. When we had to push an element and the array was full we allocated a new array, twice as large, and copied the old array into the new array. The space required by this implementation is not linear in the number of elements in the stack. To make the space linear, we change the implementation of pop as follows. When we pop an element, if the array becomes less than half full we allocate a new array of size smaller by a factor of 2 from the old array and copy the elements into the new array.

   (a) Show a sequence of n operations that takes $\Omega(n^2)$ time with this new implementation.

   (b) We now change the implementation so that instead of allocating a new array when the current array becomes less than half full we allocate a new array when the current array is less than a quarter full. The size of the new array which we allocate is still half the size of the old array, so it would be 1/2 full right after we copy the old array.

   Prove that with this implementation a sequence of m operations starting from an empty stack takes O(m) time.

4. We implement a binary counter using a linked list, each node of the list contains one digit, from the least significant digit at the first node, to the most significant digit at the last node. The counter supports the operation $add(x)$ which adds the number x to the current value of the counter. The argument x of *add* is also given as linked list of its digits from the least significant to the most significant. We perform a sequence of m operations, starting from a counter with value 0. The i^{th} add operation adds x_i. How long does it take to perform this sequence of operations? Prove your answer.

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 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 $-1, 0, 1, 2$.