Dalhousie University
CSCI 6057/4117 — Advanced Data Structures
Winter 2024
Assignment 1

Distributed Tuesday, January 16 2024.
Due at 23:59 Tuesday, January 30 2024.

Guidelines:

1. All assignments must be done individually. The solutions that you hand in must be your own work.
2. Submit a PDF file with your assignment solutions via Brightspace. We encourage you to typeset your solutions using LaTeX. However, you are free to use other software or submit scanned handwritten assignments as long as **they are legible**.
3. Working on the assignments of this course is a learning process and some questions are expected to be challenging. Start working on assignments early.
4. Whenever you are asked to prove something, give sufficient details in your proof. When it is straightforward to prove a lemma/property/observation, simply say so, but do not claim something to be straightforward when it is not.
5. We have the following late policy for course work: `https://dal.brightspace.com/d2l/le/content/311201/Home?itemIdentifier=D2L.LE.Content.ContentObject.ModuleCO-4120326`

Questions:

1. [10 marks] In the *clearable list* problem, we maintain a singly-linked list under the following two update operations:

   `INSERT`, which inserts an element into the list, and the newly inserted element will become the list head;

   `CLEAR`, which removes all the elements from the list.

   It is easy to see that `INSERT` can be performed in $O(1)$ worst-case time, while `CLEAR` can take $O(n)$ time in the worst case, where n is the number of operations performed so far starting from an initially empty list.

   Your task is to show that the amortized times of performing `INSERT` and `CLEAER` are both $O(1)$.

2. [10 marks] In class we discussed the following strategy of supporting insert and delete over resizable arrays: double the array size when inserting into a full array and halve the array size when a deletion would cause the array to become less than half full. We showed that this solution would not guarantee $O(1)$ amortized time for insert or delete by constructing a counterexample. When constructing this counterexample, we assume, for simplicity, that n is a power of 2.

   Now, construct a counterexample for the general case in which n is not necessarily a power of 2. Argue why it is a counterexample.

3. [15 marks] In class we saw the example of incrementing an initially 0 binary counter. Now, suppose that the counter is expressed as a base-3 number. That is, it has digits from $\{0, 1, 2\}$.

   (i) [5 marks] Write down the pseudocode for `INCREMENT` for this counter. The running time should be proportional to the number of digits that this algorithm changes.

   (ii) [5 marks] Define the actual cost of `INCREMENT` to be the exact number of digits changed during the execution of this algorithm. Let $C(n)$ denote the total cost of calling `INCREMENT` n times over an initially 0 base-3 counter. Use amortized analysis to show that $C(n) \leq (3/2)n$ for any positive integer n. To simplify your work, assume that the counter will not overflow during this process.

   Note: I will give the potential function as a hint after the problem statement, though you will learn more if you try to come up with your own potential function.

   (iii) [5 marks] Prove that for any c such that $C(n) \leq cn$ for any positive integer n, the inequality $c \geq 3/2$ holds. Again, assume that the counter will not overflow during this process.

   Hint: Let D_i be the counter after the i-th `INCREMENT`, and Let $|D_i|_d$ be the number of digit d in D_i for $d \in \{0, 1, 2\}$. Define $\Phi(D_i) = |D_i|_1/2 + |D_i|_2$.

4. [15 marks] Let the binary representation of a positive integer n be $n_{k-1}n_{k-2}\cdots n_0$. Thus $k = \lceil \lg(n+1) \rceil$.

   Suppose that we maintain a data structure that is a collection of arrays (NOT resizable arrays) $A_0, A_1, \cdots, A_{k-1}$ that store n integers in total.

   The following three invariants hold for this collection:

   (a) The elements in each A_i are sorted in ascending order;

   (b) The size of each A_i is 2^i, i.e., the maximum number of integers that can be stored in A_i is 2^i;

   (c) Each A_i is either full or empty: if n_i is 0, then A_i is empty; otherwise A_i is full.

Although each array is sorted, we cannot make any assumptions on the relationship between integers in different arrays. To simplify your work, you can assume that integers stored in this data structure are distinct, even under insertion and deletion (it is easy to handle duplicates).

(i) [5 marks] Describe an $O(\lg^2 n)$-time algorithm that, given an integer value x, find out whether it is one of the integers stored in this data structure. Show your analysis of the running time. You are not required to give pseudocode, but feel free to give pseudocode if it helps you explain your algorithm.

(ii) [5 marks] Describe an algorithm that supports the insertion of an integer x into this data structure, such that the three invariants hold after the insertion. In most cases, your algorithm will move elements between arrays and will even allocate one more array if k increases. Your algorithm is expected to be efficient in the amortized sense. Show your analysis. Again, pseudocode is not required but feel free to give it.

Hint: At some stage, your algorithm might merge two sorted sequences.

(iii) [5 marks] Finally, show how to support deletion efficiently in the amortized sense and analyze your algorithm. That is, given an integer x, if it can be found as an element of an array in this data structure, remove it. Note that you need to compact the arrays (you cannot leave "holes" between elements) and also maintain the invariants.

Hint: The running time of deletion would be more than that of insertion. Do your best.