CS3230: Design and Analysis of Algorithms
Semester 2, 2021-22, School of Computing, NUS

Practice Problem Set on Amortized Analysis

March 24, 2022

Instructions

- This problem set is **completely optional**. There is no need to submit the solutions.
- Solutions will be available on a later date. You are strongly encouraged to first try to solve the problems by yourselves first.
- Post on the LumiNUS forums if you will face any problem while solving the questions.

Question 1: You are asked to maintain a list L while supporting the following two operations:

- Insert(x, L): Insert an integer x in the list L (cost is 1).
- ReplaceSum(L): Compute the sum of all the integers in L. Then remove all these integers, and finally just store the computed sum in L (cost is length of the list L).

(a) Use the accounting method to show that the amortized cost of both Insert and ReplaceSum operation is $O(1)$.

(b) Use the potential method to show that the amortized cost of both Insert and ReplaceSum operation is $O(1)$.

Question 2: Let us consider the following *Online List Problem*. In this problem we are given a doubly linked list L containing n items (you may consider them to be distinct), and a sequence of m search operations $s_1, s_2, \cdots, s_m$ with the following restrictions:

- Each search operation s_t will look like: Given an item e_t you have to correctly say whether e_t is in the list L or not.
- The sequence of search operations will arrive in online fashion, i.e., at time t you will get a request for the operation s_t and you have to answer correctly. (One important thing to note, you cannot see future search queries.)
- You can access the linked list only through the HEAD pointer. So if at time[1] t you are asked to search an item e_t which is the third element in the list at that moment, then you have to pay 3 unit cost to find that element.
- During answering a search query you may update the list. However at any point you can only swap two neighboring elements in the list, and each such swap will cost you one unit. (So during any search operation if you will perform k swaps, you will have to pay additional k unit cost.)

Now keeping all these restrictions in mind, ideally we would like to design an algorithm that achieves minimum total cost.

Ooh, that must be a very difficult task! So to make your life simpler, I am not asking you to design an algorithm with the minimum cost. Instead I ask you to consider the following simple heuristic algorithm:

[1]Do not get confused between time and cost. Here I use the term *time* to identify which operation you are going to serve at that particular moment, whereas the cost captures the complexity of each operation.

Whenever we search an item in the list, if we find it we bring that in front of the list (by performing a sequence of swaps).

You may wonder since this is a extremely simple heuristic strategy, its total cost will no way be close to the minimum. However I have a different opinion. Suppose if all the m search operations were given in advance (instead of arriving in online fashion) then the minimum cost that one (optimum algorithm) could achieve is C_{opt}. I claim that the total cost of the above heuristic algorithm is just at most $4C_{opt}$. If you answer the following series of questions, you will also be able to prove this claim.

(a) During the search operation s_t suppose you are asked to search for the item e_t. Let r and r^* be the rank of e_t in the list maintained by the heuristic algorithm (denoted by L) and the optimum algorithm (denoted by L^*) respectively. Further suppose the optimum algorithm performs t^* swaps during the search operation s_t. What is the difference between the cost to perform s_t by the heuristic and the optimum algorithm?

(b) Let us define the *distance* between two lists L and L' containing exactly the same items (but in different order) as number of pairs (x, y) such that relative ordering of x and y in L and L' are different. For example, if $L = 10, 20, 30, 40$ and $L' = 10, 30, 40, 20$, then the distance between them is 2 (since $(20, 30), (20, 40)$ are the pairs with different relative ordering in L and L'). Show that $\phi(t)$, defined as the twice the distance between L and L^* at time t, is a valid potential function.

(c) Show that during performing s_t by the heuristic algorithm, each swap changes the distance between L and L^* by at most one.

(d) Next prove that the change in potential from t-th time to $(t+1)$-th time is at most $4r^*+2t^*-2r-2$.

(e) Now complete the amortize analysis using the above potential function to show that the total cost of the above mentioned heuristic algorithm is at most $4C_{opt}$.

Question 3: In the Set Union problem we have n elements, that each are initially in n singleton sets, and we want to support the following operations:

- Union(A, B): Merge the two sets A and B into one new set $C = A \cup B$ destroying the old sets.
- SameSet(x, y): Return true, if x and y are in the same set, and false otherwise.

We implement it in the following way. Initially, give each set a distinct color. When merging two sets, recolor the smaller (in size) one with the color of the larger one (break ties arbitrarily). Note, to recolor a set you have to recolor all the elements in that set. To answer SameSet queries, check if the two elements have the same color. (Assume that you can check the color an element in $O(1)$ time, and to recolor an element you also need $O(1)$ time. Further assume that you can know the size of a set in $O(1)$ time.)

Use Aggregate method to show that the amortized cost is $O(\log n)$ for Union. That means, show that any sequence of m union operations takes $O(m \log n)$ time. (Note, we start with n singleton sets.)

Question 4: Suppose Alice insists Bob to maintain a dynamic table (that supports both insertion and deletion) in such a way its size must always be a Fibonacci number. She insists on the following variant of the rebuilding strategy. Let F_k denote the k-th Fibonacci number. Suppose the current table size is F_k.

After an insertion, if the number of items in the table is F_{k-1}, allocate a new table of size F_{k+1}, move everything into the new table, and then free the old table. After a deletion, if the number of items in the table is F_{k-3}, we allocate a new hash table of size F_{k-1}, move everything into the new table, and then free the old table.

Use either Potential method or Accounting method to show that for any sequence of insertions and deletions, the amortized cost per operation is still $O(1)$. (If you use Potential method clearly state your potential function. If you use Accounting method clearly state your charging scheme.)