Algorithms Fall 2016 Homework 4 Solution

October 17, 2016

PROBLEM 1 : MULTIPUSH

No. The time cost of stack operations depends on the number of pushes. Multipush needs $O(k)$ time, so the worst case of performing n stack operations is to perform n multipush that pushes k items to stack each time. The time cost is $O(nk)$ and the amortized cost is $O(k)$.

PROBLEM 2: NOT BELOW 1/3 FULL

Let c_i be the actual cost of the ith operation, $\hat{c}_i$ denote the armotized cost of the ith operation with respect potential function Φ, num_i denote the number of items stored in the table after ith operation, $size_i$ denote the total size of the table after ith operation. We have $\Phi_0 = |2num_0 - size_0| = 0$ and $\Phi_i \geq 0 = \Phi_0$. Thus, the total armotized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost.
To analyze the amortized cost of a TABLE-DELETE operation, we consider the case that the ith operation is TABLE-DELETE. Then we have $\frac{num_{i-1}}{size_{i-1}} \geq \frac{1}{3}$ and $num_i = num_{i-1} - 1$. We will calculate the amortized cost as follows:

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = c_i + |2num_i - size_i| - |2num_{i-1} - size_{i-1}|$$

We need to consider two cases:

1. **The load factor does not drop below $\frac{1}{3}$**

   We have $num_i \geq \frac{1}{3} size_{i-1}$. We do not need to contract the table, so $size_i = size_{i-1}$. The cost for removing one item from the table is 1. We have the amortized cost as follows:

$$\begin{aligned}\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} &= 1 + |num_i - size_i| - |2num_{i-1} - size_{i-1}| \\ &= 1 + |num_{i-1} - size_{i-1} - 2| - |num_{i-1} - size_{i-1}| \\ &\leq 1 + |(2num_{i-1} - size_{i-1} - 2) - (num_{i-1} - size_{i-1})| = 3\end{aligned}$$

2. **The load factor drops below $\frac{1}{3}$**

   we have
$$\frac{num_i}{size_{i-1}} < \frac{1}{3} \leq \frac{num_{i-1}}{size_{i-1}} \Rightarrow 2num_i < \frac{2}{3} size_{i-1} \leq 2num_i + 2$$
   We need to contract the table and then we have:
$$size_i = \lfloor \frac{2}{3} size_{i-1} \rfloor \Rightarrow \frac{2}{3} size_{i-1} - 1 \leq size_i \leq \frac{2}{3} size_{i-1}$$
   By combining above two statements, we have $2num_i - 1 \leq size_i \leq 2num_i + 2$ and thus $|2num_{i-1} - size_{i-1}| \leq 2$. Since $\frac{num_i}{size_{i-1}} < \frac{1}{3}$, we have:
$$|2num_{i-1} - size_{i-1}| = size_{i-1} - 2num_{i-1} \geq 3num_i - 2num_{i-1} \geq num_{i-1}$$
   The cost for removing one item and moving the remaining num_i items into the contracted table is $num_i +$ 1. We have:
$$\begin{aligned}\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} &= (num_i + 1) + |2num_i - size_i| - |2num_{i-1} - size_{i-1}| \\ &\leq (num_i + 1) + 2 - num_i = 3\end{aligned}$$

In both cases the amortized cost is at most 3 and thus bounded above by a constant.

PROBLEM 3 : DYNAMIC BINARY SEARCH

a) SEARCH

We note that each array is sorted but elements in different arrays bear no particular relationship to each other. So we can perform SEARCH operation by applying normal binary search on each array A_i (for $i = 0, 1, ..., k-1$) individually. In the worst case, all arrays are full and we perform unsuccessful SEARCH operation on each array. The time cost for performing an unsuccessful binary search is $O(\lg|A_i|)$. The total time cost is

$$T(n) = O(\lg|A_0| + \lg|A_1| + ... + \lg|A_{k-1}|)$$

$$= O(\lg 2^0 + \lg 2^1 + ... + \lg 2^{k-1})$$

$$= O(0 + 1 + .. + k - 1)$$

$$= O(k(k-1)/2)$$

$$= O((\lceil \lg n + 1 \rceil)(\lceil \lg n + 1 \rceil - 1))$$

$$= O(\lg^2 n)$$

b) INSERT

To perform INSERT operation, we start from inserting the element into A_0, end if A_i is empty. Initially, we consider the new element to be an array A_{new} of length 1. We start from A_0, if A_0 is empty, we finish INSERT operation by replacing A_0 with A_{new}. Otherwise A_0 is full, we merge sort A_0 and A_{new} and let A_{new} be the result. We know that the length of new A_{new} is 2 and it's the same as A_1, so if A_1 is not empty, we need to continue this process. We stop this process until we find A_i is empty. In the worst case, $A_0, A_1, ..., A_{k-2}$ are all full. We need to perform $k-1$ times of merge sort. The time cost of merge sorting of two arrays of length m is $2m$. The time cost is

$$T(n) = 2|A_0| + 2|A_1| + ... + 2|A_{k-1}| = 2(2^0 + 2^1 + 2^{k-2}) = 2(2^{k-2} - 1) = O(2^k) = O(n)$$

We use aggregate method to analyze the amortized running time of the INSERT operation. Suppose we perform INSERT operation for n times. According to the binary representation of $n(< n_{k-1}, n_{k-2}, ..., n_0 >)$, we have $n_j = 1$ when A_j is full. We can see that n_0 changes every time, n_1 changes every 2th time,..., n_{k-1}th changes every 2^kth time. The change indicates a merge operation. So we have a total running time for performing INSERT n times:

$$O(\sum_{r=0}^{k} \lceil n/2^r \rceil) = O(nk) = O(n \lg n)$$

The amortized cost of an INSERT operation is $O(\lg n)$

Analysis using accounting method: We charge k to insert an element, 1 pays for the insertion and $k-1$ is kept for it being involved in merge sort of later insertions. Since each element would only be moved to array with higher index, so $k-1$ suffices to pay for moving an element from A_0 to A_{k-1}. We need to pay for nk to insert n elements. The amortized cost is $O(k) = O(\lg n)$.

c) DELETE

Here are the implementation of SEARCH operation, suppose we want to delete an element x from the array.

1. Use SEARCH to find the array A_i which contains x.
2. Find the array A_j which is full and of smallest j.
3. Delete x from A_i. If $i \neq j$, remove the last element y from A_j and insert y to A_i.
4. Now A_j has $2^j - 1$ elements. We move the elements in A_j to the empty arrays $(A_0, A_1, ..., A_{j-1})$ by following steps: move the first element to A_0, move the next 2^1 elements to A_1 and so forth. Since the capacity of $(A_0, A_1, ..., A_{j-1})$ is $2^j - 1$, after moving, A_j should be empty.

The worst case is that x is found in A_{k-1} and $(A_0, A_1, ..., A_{k-3})$ are all empty. In this case we need $O(\lg^2 n)$ to perform the SEARCH, $O(\lg n)$ to find the first full array A_{k-2}, $O(n)$ time to insert y to A_{k-1} and $O(n)$ time to move elements of A_{k-2}. The time cost is $O(n)$.

Analysis of interleaving INSERT and DELETE operations: If we analyze the time cost by simply interleaving the INSERT and DELETE operations on a set of empty arrays, it would be easy since there is no more than 1 element in the arrays. The time cost for each operation is $O(1)$. To better understand this problem, we can consider the following sequence of n operations:

1. Perform INSERT for $n/3$ times.

2. Perform pairs of DELETE and INSERT for $n/3$ times.

According to the previous analysis, the time cost in the worst case is $O(n^2)$, and hence the worst-case per operation is $O(n)$. We use aggregate method to analyze the amortized running time of this sequence of operations. In analysis of a), we know that the time cost for performing $n/3$ times of INSERT if $O(\frac{n}{3}\lg n/3) = O(n\lg n)$, after n times of INSERT, we will create a single full array with size of $n/3$. According to analysis of b) and c), for each pair of DELETE/INSERT, we need $O(n)$ for the DELETE to remove the element from the array and move the remaining elements to the arrays with lower indexes, and $O(n)$ for the INSERT to recombine these elements. The amortized cost is:

$$(O(n\lg n) + \frac{n}{3}O(2n))/n = O(n)$$

It is not better than the worst-case cost per operation.