Lab07-Amortized Analysis

CS214-Algorithm and Complexity, Xiaofeng Gao & Lei Wang, Spring 2021.

∗ If there is any problem, please contact TA Yihao Xie.
∗ Name:ZIrui Liu Student ID:519021910343 Email: L.prime@sjtu.edu.cn

1. Suppose we perform a sequence of n operations on a data structure in which the i th operation costs i if i is an exact power of 2, and 1 otherwise. Use an accounting method to determine the amortized cost per operation.

   Solution. In the sequence which has n operations, there are $\lfloor \log_2 n \rfloor + 1$ numbers which are power of 2. These numbers have total time cost of $2^{\lfloor \log_2 n \rfloor + 1} - 1$, which approximately is $2 * n$. The rest of the numbers together have a time cost of $n - \lfloor \log_2 n \rfloor - 1$, which approximately is n. So the total time cost is about $3 * n$, which means amortized cost is $O(1)$.

   □

2. Consider an ordinary **binary min-heap** data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in $O(\log n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\log n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works.

   Solution. We assume that D_i to be the state of **min-heap** after i-th operation. We also let n_i to be the size of D_i after i-th operation. If the $i-1$-th operation is INSERT, then $n_i = n_{i-1} + 1$. On the other hand, if the $i-1$-th operation is EXTRACT-MIN, then $n_i = n_{i-1} - 1$. We shall also define $a(i)$ to be parameter function, for EXTRACT-MIN, we define $b_1(i)$ to satisfy that:

   $$T_{Extract-Min(i)} \leq b_1(i) * \ln i \tag{1}$$

   for INSERT, we define $b_2(i)$ to satisfy that:

   $$T_{Insert(i)} \leq b_2(i) * \ln i \tag{2}$$

   We also define that:

   $$a(i) = max\{b_1(i), b_2(i)\} \tag{3}$$

   So inclusively we have:

   $$T(i) \leq a(i) * \ln i \tag{4}$$

   Also, we can always define $a(i)$ to be in non-increasing order.Then we can finally get:

   $$\phi(D_i) = \begin{cases} 0, & i = 0 \\ \sum_{m=1}^{n_i} a(m) \ln m, & i > 0 \end{cases} \tag{5}$$

   Then we will calculate the amortized costs of EXTRACT-MIN and INSERT to prove the correctness.

For INSERT operation:
Since $n_i = n_{i-1} + 1$,

$$c_i \leq a\left(n_{i-1}\right) * \ln n_{i-1} \leq a\left(n_i\right) * \ln n_i \tag{6}$$

$$\begin{aligned}
\hat{c_i} &= c_i + \phi\left(D_i\right) - \phi\left(D_{i-1}\right) \\
&= c_i + \sum_{k=1}^{n_i} a\left(k\right) \ln k - \sum_{k=1}^{n_{i-1}} a\left(k\right) \ln k \\
&= c_i + \sum_{k=1}^{n_i} a\left(k\right) \ln k - \sum_{k=1}^{n_i - 1} a\left(k\right) \ln k \\
&= c_i + a\left(n_i\right) * \ln n_i \\
&\leq a\left(n_{i-1}\right) * \ln n_{i-1} + a\left(n_i\right) * \ln n_i \\
&\leq 2 * a\left(n_i\right) * \ln n_i \\
&= O\left(\ln n_i\right)
\end{aligned} \tag{7}$$

For EXTRACT-MIN operation:
Since $n_i = n_{i-1} - 1$,

$$c_i \leq a\left(n_{i-1}\right) * \ln n_{i-1} \tag{8}$$

$$\begin{aligned}
\hat{c_i} &= c_i + \phi\left(D_i\right) - \phi\left(D_{i-1}\right) \\
&= c_i + \sum_{k=1}^{n_i} a\left(k\right) \ln k - \sum_{k=1}^{n_{i-1}} a\left(k\right) \ln k \\
&= c_i + \sum_{k=1}^{n_i} a\left(k\right) \ln k - \sum_{k=1}^{n_i + 1} a\left(k\right) \ln k \\
&= c_i - a\left(n_i + 1\right) * \ln\left(n_i + 1\right) \\
&\leq a\left(n_{i-1}\right) * \ln n_{i-1} - a\left(n_i + 1\right) * \ln\left(n_i + 1\right) \\
&= a\left(n_{i-1}\right) * \ln n_{i-1} - a\left(n_{i-1}\right) * \ln n_{i-1} \\
&= O\left(1\right)
\end{aligned} \tag{9}$$

So the ϕ created can work.

□

3. Assume we have a set of arrays $A_0, A_1, A_2, \cdots$, where the i^{th} array A_i has a length of 2^i. Whenever an element is inserted into the arrays, we always intend to insert it into A_0. If A_0 is full then we pop the element in A_0 off and insert it with the new element into A_1. (Thus, if A_i is already full, we recursively pop all its members off and insert them with the elements popped from $A_0, ..., A_{i-1}$ and the new element into A_{i+1} until we find an empty array to store the elements.) An illustrative example is shown in Figure **??**. Inserting or popping an element take $O(1)$ time.

图 1: An example of making room for one new element in the set of arrays.

(a) In the worst case, how long does it take to add a new element into the set of arrays containing n elements?

(b) Prove that the amortized cost of adding an element is $O(\log n)$ by *Aggregation Analysis*.

(c) If each array A_i is required to be sorted but elements in different arrays have no relationship with each other, how long does it take in the worst case to search an element in the arrays containing n elements?

(d) What is the amortized cost of adding an element in the case of (c) if the comparison between two elements also takes $O(1)$ time?

Solution. (a):
If we assume each small step to take up 1 unit of time, it would be $n+1$ units that we needed. The worst case would be that already having n elements filling up previous arrays, so it would take n time to move these elements to the next array.Then we add the new element, making the whole time to $n+1$ units, which in another way is $O(n)$.

(b):
We assume that the time needed for the first n elements is $T(n)$. For $n = 2^k$, considering $\frac{n}{2}$, if we add the next element, its time cost along with the first $\frac{n}{2} - 1$ elements, would be exactly $T\left(\frac{n}{2}\right)$.But in this time these $\frac{n}{2}$ elements have already been moved to the array we want them to be in. And the time cost for moving the next $\frac{n}{2}$ filling up an array would be $\frac{n}{2}$. Then we should consider moving these $\frac{n}{2}$ elements to the next array, which would cost another $\frac{n}{2}$ time. So the equation would be like:

$$T(n) = 2 * T\left(\frac{n}{2}\right) + \frac{n}{2} \tag{10}$$

Then we apply the Master Theory, for $a = 2$, $b = 2$, $f(n) = \frac{n}{2}$, we have $T(n) = O(n * \log n)$. Then we apply *Aggregation Analysis*, the amortized cost of adding an element is $O(\log n)$, calculated by $\frac{T(n)}{n}$.

(c):
It would be $(\log n)^2$. The analysis is as follows:
In the worst case, we would have to look through every array to search for the element x we want. And since every array has no relationship with another, so in every array with k elements, we must spent $\log k$ time to search this array. So in conclusion, we need $(\log n)^2$ time, considering all the arrays.

(d):
Considering when currently the arrays A_1,A_2,$A_3 \cdots A_{i-1}$ are full, and we transform all their elements to A_i, it would cost time:

$$\sum_{k=0}^{i-1} 2^k + O\left(2^{k+1} - 1\right) \tag{11}$$

where in this equation 2^k is the time complexity we needed for popping A_k, and $O\left(2^{k+1} - 1\right)$ is the time we needed for merging sort of two sorted arrays.
Adding this equation together, we can get that

$$\sum_{k=0}^{i-1} 2^k + O\left(2^{k+1} - 1\right) = O\left(2^i\right) = O(n) \tag{12}$$

Dividing this time by n, we have constant level of time complexity. So considering what we have got in (b), we can conclude that the amortized cost of adding an element is still $O(\log n)$.

□

Remark: Please include your .pdf, .tex files for uploading with standard file names.