Lab07-Amortized Analysis

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

∗ If there is any problem, please contact TA Yihao Xie.
∗ Name:Xin Xu Student ID:519021910726 Email: xuxin20010203@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. The amortized cost per operation is 3. For any ith operation that satisfying $2^k + 1 \leqslant i \leqslant 2^{k+1}, k \geqslant 1$, 1 is paid for the operation itself for current use, 1 is paid for the operation itself for future use, the final 1 is paid for the jth operation that $2^{k-1} + 1 \leqslant j \leqslant 2^k$. So the sum of payment from $2^k + 1$ to 2^{k+1} is $2^k + 2^{k+1} + 1$, which is one more that the total actual payment with the same range. So, the amortized cost is actually an upper bound of the real cost, and it's the same with the first two operations. In a nutshell, because this amortized analysis provides an upper bound, it's right. □

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. Let $D(v_i)$ denotes the height of node v_i. The potential function is that $\Phi = \sum_{i=1}^{n} D(v_i)$. It's easy to know that $\Phi(0) = 0, \Phi(n) \geqslant 0$.
   And for the nth insertion, if the total height is unchanged, $\Phi(n) - \Phi(n-1) = O(\log n)$, and if the height increases, $\Phi(n) - \Phi(n-1) = O(\log n)$, too. And since the worst case of insertion takes $O(\log n)$, the amortized cost$= O(\log n)$.
   And for the nth deletion, $\Phi(n) - \Phi(n-1) = -O(\log n)$ just as discussed above, while the lower down operation takes $O(\log n)$, which can be a compensation. So, the amortized cost is $O(1)$. □

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 1. Inserting or popping an element take $O(1)$ time.

   (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?

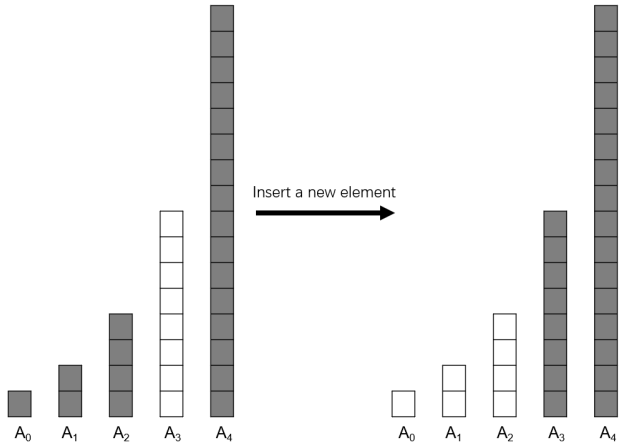

图 1: An example of making room for one new element in the set of arrays.

Solution. (a) The worst case is that the first i arrays are all full. If the examination to judge an array full or not takes $O(1)$, the total cost to add a new element into the set of arrays containing n elements is $\log(n+1)+2n+1$. So, in the worst case, it takes $O(n)$.

(b) With the discussion in previous question, the examination to judge an array full or not can be omitted. Thus, the cost list of the first n operations satisfies the follow properties: Firstly, the $2^k, (k \geqslant 1)$th operation costs $2^{k+1}-1$; Secondly, the list is actually a copy and paste process to double itself except the $2^k, (k \geqslant 1)$th operation.
So, the total cost of the first $n(n=2^m)$ operations is $n/2+3n/2^2+7n/2^3+15n/2^4+\ldots+2n-1 = mn-1+1/n+2n-1 = mn+2n+1/n-2$. And the amortized cost is $(mn+2n+1/n-2)/n = O(m) = O(\log n)$.

(c) In the wost case, the search should compare all the elements. So, it takes $O(n)$.

(d) The list of cost has the same structure of problem (b). The $2^k, (k \geqslant 1)$th operation cost includes $2^{k+1}-1$ to locate elements and the cost of comparation. To figure out the cost of comparison, firstly we should know the cost to resort two sorted arrays with the same lengh n. In the worst case, the comparison is $2n-1$ for every element in the two arrays must be compared except for the last one. So, the comparison cost of $2^k, (k \geqslant 1)$th operation is $\sum_{i=1}^{k}(2i-1) = k^2 < 2^{k+1}-1$. So, if we double the value of each element in the list of problem (b), we can get an upper bound of problem (d) as well. And the double process doesn't change the time complexity.
So, the amortized cost of adding an element in the case (c) considering the cost of comparison is $O(\log n)$ too.

□

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