Lab07-Amortized Analysis

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

∗ If there is any problem, please contact TA Yihao Xie.
∗ Name:Yanjie Ze Student ID:519021910706 Email: zeyanjie@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. Denote C_i as the cost of the ith operation, then we will C_i's formula:

   $$C_i = \begin{cases} i, & if\ i\ is\ the\ power\ of\ 2 \\ 1, & otherwise \end{cases}$$

   We can view this problem as a *Table Insert* problem, and the formula can be explained as such:

   - If inserting the ith element doesn't lead to the full table, insert the element. The cost is 1.
   - If inserting the ith element leads to the full table, we should first expand the table and second copy all the old elements and also insert the new element. The cost is i.

   For example, Fig 1 illustrates the process of $insert(4)$. When we try to insert the 4th element, we find that the table will be full after the insertion. So we first expand the table, and then copy and insert. The number in each box shows the cost, and the cost of $insert(4)$ is 4, which consists of 3 copys and 1 insertion.

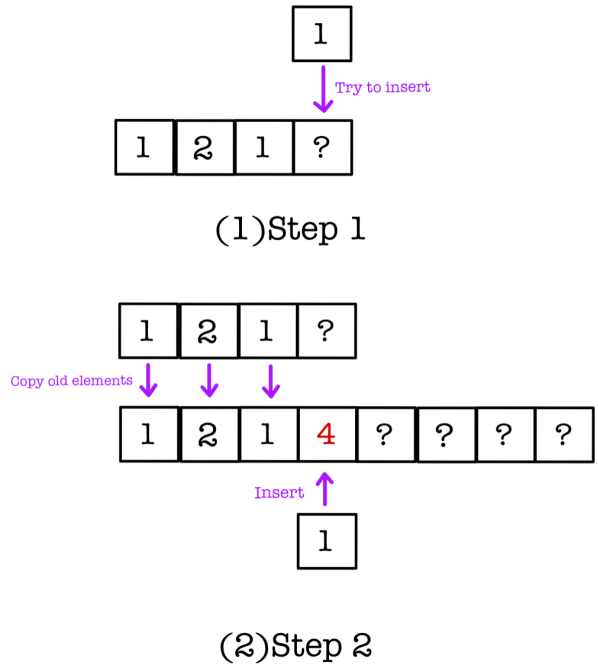

图 1: Insert(4)

   Therefore, for the ith operation, an amortized cost $\hat{C}_i = 3\$$ is charged.

- 1$ pays for the insertion itself.
- 1$ pays for the next copy of itself.
- 1$ pays for the next copy of an old element.

Denote $T(n)$ as the total cost of n operation, we have an upper bound by this **accounting method**:

$$T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \hat{C}_i = 3n$$

And the amortized cost per operation is 3. □

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.
Potential Function:

Define state S_i as the number of elements in the heap. Define the potential function $\Phi(S_i)$:

- INSERT: $\Phi(S_i) - \Phi(S_{i-1}) = \log S_i$
- EXTRACT-MIN: $\Phi(S_i) - \Phi(S_{i-1}) = -\log S_i$

Explanation:

When performing INSERT, the heap will have more elements, which is viewed as the increase of potential energy and its height is also possible to rise. And the potential energy's quantity rests on the current height, which is similar to climbing: the higher position we are in, the more potential energy we have.

When performing EXTRACT-MIN, the heap's top element is extracted, and the heap needs to be adjusted. The action of adjustment will consume the potential energy accumulated before.

Correctness:

Because EXTRACT-MIN will not perform when the heap has no elements, which means:

$$\Phi(S_i) \geq 0 = \Phi(S_0)$$

Amortized Analysis:

- $0 \leq S_i \leq n$
- INSERT: $\hat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1}) = 2\log S_i = O(\log n)$
- EXTRACT-MIN: $\hat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1}) = 0 = O(1)$

□

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

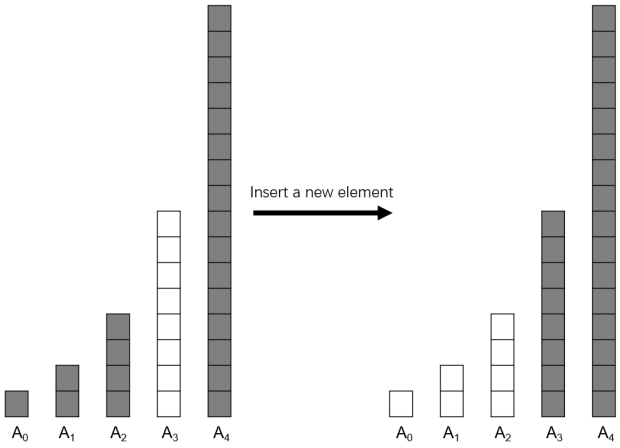

图 2: An example of making room for one new element in the set of arrays.

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

Solution.
Lemma 1:*If there exist elements in array A_i, array A_i must be full, with 2^i elements.*

Proof. Lemma 1 is naturally induced from the problem setting.
For array $A_0, A_1, ..., A_n$, they can store $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ elements.
If $A_0, ..., A_n$ are all full, inserting a new element will make all old elements and the new element into array A_{n+1}, which can exactly hold 2^{n+1} elements. Then, A_{n+1} is full and the former arrays are totally empty.
Since n can be any positive integer value, ranging from 1 to ∞, we prove **Lemma 1** is correct. □

As shown in Fig. 3, INSERT(3) fills A_0 to the full and INSERT(4) fills A_2 to the full and empties A_0 and A_1, verifying **Lemma 1**.

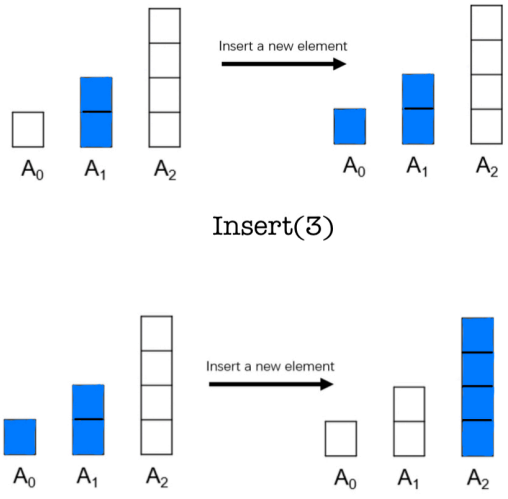

图 3: Process of Inserting

Based on **Lemma 1**, we know if there are n elements in total, **the worst case** is that all elements are filled in $A_0, A_1, ..., A_k$, with no middle empty arrays. In such case, $n = 2^{k+1} - 1, k = 1, 2, ...$

The cost of INSERT(N+1)is:

$$cost_{worst} = n \times cost_{pop} + (n+1) \times cost_{push} = 2n + 1 = O(n)$$

□

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

Solution.

First, we get the expression of C_i.

- When i is odd, C_i is equal to 1 obviously.
- When $i = 2^k$, which is talked about in problem(a), $C_i = 2n - 1$.
- When i isn't odd and is not equal to $i = 2^k$, then i must be an even number between 2^k and 2^{k+1}, where $k = \lfloor \log_2 n \rfloor$. In this case, there exists one or more empty array in array $A_0, A_1, ..., A_k$. So the pattern of the change of C_i is similar to that before. For example, C_1 to C_{32} is:

  1, 3, 1, 7, 1, 3, 1, 15
  1, 3, 1, 7, 1, 3, 1, 31
  1, 3, 1, 7, 1, 3, 1, 15
  1, 3, 1, 7, 1, 3, 1, 63

Therefore, the expression of C_i is :

$$C_i = \begin{cases} 1, & when\ i\ is\ odd. \\ 2i - 1, & when\ i = 2^k, k \in \mathbb{N} \\ C_{i-2^{\lfloor \log_2 i \rfloor}}, & otherwise \end{cases}$$

Since merely based on the expression we can't figure out the total cost easily, we compute the total cost from another perspective.

Assume $n = 2^k$**,** $k \geq 1$. This is because when adding the nth element, the operation will bring the most cost.

Define a **cycle** for a stack is **from EMPTY to FULL, then from FULL to EMPTY**. Because from **Lemma 1** we know the stack only has two state: EMPTY or FULL.

For each stack A_s, $0 \leq s < k$, A_s will go through several cycles: EMPTY to FULL, FULL to EMPTY. By calculating out the number of cycles each stack goes through, we are able to figure out the total cost.

For the stack A_s, the number of cycles it goes through is:

$$num_{cycle}(A_s) = \frac{2^k}{2^{s+1}} = 2^{k-s-1}$$

For one cycle of the stack A_s, the cost is:

$$cost_{cycle}(A_s) = cost_{pop} + cost_{push} = 2^{s+1}$$

Thus, the total cost of a stack is:

$$\begin{aligned} cost_{stack}(A_s) &= num_{cycle}(A_s) \times cost_{cycle}(A_s) \\ &= 2^{s+1} \times 2^{k-s-1} \\ &= 2^k, for\ 0 \leq s < k \end{aligned}$$

Therefore:

$$\begin{aligned} T(n) &= \sum_{s=0}^{k-1} cost_{stack}(A_s) + cost_{stack}(A_k) \\ &= (k-1)2^k + 2^k \\ &= k2^k \\ &= n \log n \end{aligned}$$

Finally, by **Aggregation Analysis**,we get the amortized cost of adding an element is:

$$\frac{T(n)}{n} = O(\log n)$$

□

(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?

Solution. The cost of searching an element in A_i using **Binary Search** is $O(\log_2 |A_i|)$. In the worst case, to search an element, we need to search it in all arrays. What's more, there exists no empty array in $A_0, A_1, ..., A_k, n = 2^{k+1} - 1$.
This is because:

$$\log |A_0| + \log |A_1| + ... + \log |A_k| = \log(|A_0| \times |A_1| ... \times |A_k|) > \log |A_{k+1}|$$

Which means searching elements distributed in different arrays takes more time.

Thus, the total cost of searching an element is:

$$\begin{aligned} cost_{worst} &= O(\log |A_0|) + O(\log |A_1|) + ... + O(\log |A_k|) \\ &= O(\log(|A_0| \times |A_1| ... \times |A_k|)) \\ &= O(\log(2^{0+1+...+k})) \\ &= O(\log(2^{\frac{k^2+k}{2}})) \\ &= O(k^2) \\ &= O(\log^2 n) \end{aligned}$$

□

(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.
The cost of adding an element without comparison is shown in problem(b). For adding an element and making the array in order, the cost consists of three parts:**pop,sort,push**.
Since the only difference between the problem(b) and the problem(d) is the need of sorting, the extra cost is the cost of sorting.
We still assume $n = 2^k, k \geq 1$, because the 2^kth operation costs most.
From the problem(b) we have known that:

$$num_{cycle}(A_s) = 2^{k-s-1}$$

However, the cost of one cycle is different:

$$cost_{cycle}(A_s) = cost_{pop} + cost_{push} + cost_{sort} = 2^{s+1} + cost_{sort}$$

For one cycle of A_s, we only sort the elements to be pushed in A_s once. Then the problems is: **How can we sort these sorted arrays?**

One try: K-Merge

For the first try, we propose an algorithm using **Min Heap**, with time complexity $O(nlogk)$. The algorithm **K-Merge** is shown in Alg. 1.

Algorithm 1: K-Merge

```
Input:k sorted arrays(increasing):{a_0, a_1, a_2, .., a_k};
Output:One sorted array;
Create an empty Min Heap H;
Create an empty result array A;
for i = 0; i ≤ k; i++ do
    Insert a_i's first element into H;
    Delete this element from a_i;
while H is not empty do
    Pop out the minimal element in H into A;
    Denote this element is from a_s;
    if a_s is not empty then
        Insert a_s's first element into H;
return A
```

Time Complexity: $O(n \log k)$

In alg.1, We fully utilize the sequence of each array, and maintain a Min Heap of size k. Each push and pop operation in Min Heap is $O(\log k)$. For there are n elements in total, the time complexity is $O(n \log k)$, where k is the number of the arrays.

Therefore,

$$\begin{aligned} cost_{sort}(A_s) &= O(2^s log(s-1)) = O(2^s logs) \\ cost_{cycle}(A_s) &= 2^{s+1} + cost_{sort}(A_s) = 2^{s+1} + O(2^s logs) \\ cost_{stack}(A_s) &= num_{cycle}(A_s) \times cost_{cycle}(A_s) \\ &= 2^{k-s-1} \times (2^{s+1} + O(2^s logs)) \\ &= 2^k + O(2^{k-1} logs), 1 \le s \le k-1 \end{aligned}$$

Finally ,we figure out the cost of all:

$$\begin{aligned} T(n) &= \sum_{s=1}^{k-1} cost_{stack}(A_s) + cost_{stack}(A_0) + cost_{stack}(A_k) \\ &= (k-1)2^k + 2^{k-1}O(log((k-1)!)) + 2^k + 2^k + 2^k O(logk) \\ &= O(k2^k) + O(2^k log(k!)) \\ &= O(nlogn) + O(nlog((logn)!)) \\ &= O(nlog((logn)!)) \end{aligned}$$

By **Aggregation Analysis**,we get the amortized cost of adding an element is:

$$\frac{T(n)}{n} = O(log((logn)!))$$

Another Try:Mutual-Merge

Another sort algorithm that works is that we sort k arrays two by two. For example, to sort $a_0, a_1, a_2..., a_k$, We first merge a_0 and a_1 to become a_01, then merge a_01 and a_2 to become a_012...until we merge all arrays. The algorithm called $Mutual - Merge$ is shown in Alg. 2.

Algorithm 2: Mutual-Merge

Input:k sorted arrays(increasing):$\{a_0, a_1, a_2, .., a_k\}$;
Output: One sorted array;
for $i = 0; i < k; i++$ **do**
    Merge a_i and a_{i+1} into $a_{i,i+1}$;
return $a_{0,1,...,k}$

Make the input $A_0, A_1, ..., A_{k-1}$, then we compute the time complexity:

$$\begin{aligned} T &= (|A_0| + |A_1|) + (|A_0| + |A_1| + |A_2|) + ... + (|A_0| + |A_1| + |A_2|... + |A_{k-1}|) \\ &= \sum_{i=2}^{k} 2^i - 1 \\ &= 2^{k+1} + k - 3 \\ &= O(n) \end{aligned}$$

Therefore,

$$\begin{aligned} cost_{sort}(A_s) &= O(2^s) \\ cost_{cycle}(A_s) &= 2^{s+1} + O(2^s) \\ cost_{stack}(A_s) &= num_{cycle}(A_s) \times cost_{cycle}(A_s) \\ &= 2^{k-s-1} \times (2^{s+1} + O(2^s)) \\ &= O(2^k), 1 \leq s \leq k-1 \end{aligned}$$

Finally, the cost of all is:

$$T(n) = O(k2^k) = O(nlogn)$$

And the amortized cost is:

$$\frac{T(n)}{n} = O(logn)$$

Conclusion:

The second way is better than the first way, which uses Alg. 2, with the amortized complexity $O(logn)$.

□

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