Lab07-Amortized Analysis

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

∗ If there is any problem, please contact TA Yihao Xie.
∗ Name:RenyangGuan Student ID:519021911058 Email: guanrenyang@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. Define the actual cost of the operation i is C_i and we have

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

   Assuming $n = 2^k$, then the actual total cost is

   $$\sum_{i=1}^{n} C_i = 1 + 2 + (1) + 4 + (1 + 1 + 1) + 8 + \cdots + (2^{k-1} - 1) + 2^k + \cdots \tag{2}$$

   We could think $2^{k-1} - 1$ 1s and one 2^k of an unit. If i is not a power of 2, the $\hat{C}_i$ acts as a prepaid credit. The credit will be used later for the nearest operation with i equaling to a power of 2. Based on the consideration, we could generate the amotsized cost as follows:

Operation	Read Cost C_i	Amortized Cost $\hat{C}_i$
i is a power of 2	i	0
i is not a power of 2	1	4

   Because

   $$\begin{aligned} \sum_{i=1}^{n} C_i &= 1 + 2 + 4 + 8 + \cdots + 2^k + \\ &\quad (2^1 - 1) + (2^2 - 1) + \cdots + (2^{k-1} - 1) + t \\ &= 2^{k+1} - 1 + 2(2^{k-1} - 1) + k - 1 \\ &= 3 \times 2^k + k - 3 \\ &< 3n + \log n - 3 \\ &< 4n \end{aligned} \qquad \begin{aligned} \sum_{i=1}^{n} \hat{C}_i &= 4(2^{-}1 + 2^2 - 1 + \cdots + 2^k - 1) \\ &= 4 \times (2^k - 3 + k) \\ &= 4 \times (n + \log n - 3) \\ &> 4n \\ &> \sum_{i=1}^{n} C_i \end{aligned} \tag{3}$$

   The amortized cost analysis above is true and the amortized cost of each operation 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.
   Basic consideration: An element must be inserted into the binary min-heap before being extracted. As a result, the INSERTION operation prepays the *credit*, which will be used by

EXTRACT-MIN.
Potential function

$$\Phi(n) = n \log n \tag{4}$$

where n is the number of elements in the heap.

Proof:
Proof for correctness: Since the worst-case cost of both INSERT and EXTRACT-MIN is $O(\log n)$, the total amortized cost is

$$\begin{aligned}\sum_{i=1}^{n} \hat{C}_i &= \sum_{i=1}^{n} C_i + \Phi(n) - \Phi(1) \\ &= n \log n + n \log n \\ &= 2n \log n \\ &> \sum_{i=1}^{n} C_i\end{aligned}$$

Proof for the question requirement:
Case 1: the n^{th} operation is INSERT:

$$\begin{aligned}\hat{C}_n &= C_n + \Phi(S_n) - \Phi(S_{n-1}) && (5)\\ &= \log n + n \log n - (n-1)\log(n-1) && (6)\\ &= \log n + n \log \frac{n}{n-1} + \log(n-1) && (7)\\ &= \Theta(\log n) + \Theta(1) + \Theta(\log n) && (8)\\ &= \Theta(\log n) && (9)\end{aligned}$$

Case 2: the n^{th} operation is EXTRACT-MIN:

$$\begin{aligned}\hat{C}_n &= C_n + \Phi(S_n) - \Phi(S_{n-1}) && (10)\\ &= \log n + n \log n - (n+1)\log(n+1) && (11)\\ &= (n+1) \log \frac{n}{n+1} && (12)\\ &= \Theta(1) && (13)\end{aligned}$$

Since C_n is the worst case time complexity of the n^{th} operation and $\hat{C}_i$ gives an upper bound of C_i, the time complexity of INSERT is $O(\log n)$ and that of EXTRACT-MIN 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 **??**. Inserting or popping an element take $O(1)$ time.

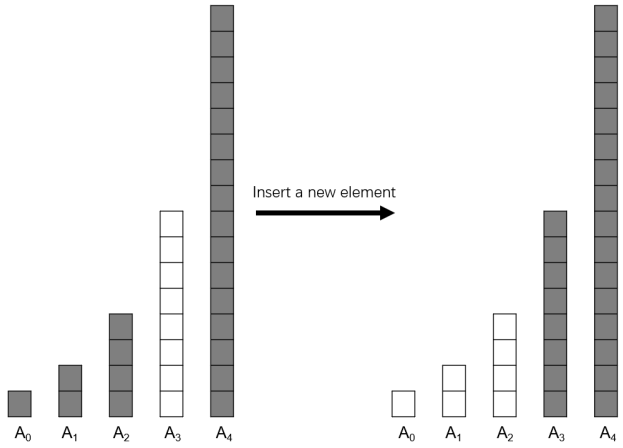

图 1: An example of making room for one new element in the set of arrays.

(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) For the sake of connevence, aussume $n = 2^k - 1$. The worst case occurs when n occupies $[A_0, A_1, \cdots, A_{k-1}]$. All of the n elements have to be popped out to insert the $n + 1$ elements. As a result, it takes $O(n)$ to add a new element in the worst case.

(b) **Proof.**
Let ADDITION denote the only operation in this question, which adds an element in the set of arrays.
Before calculating the amortized time complexity of the sequence of n operations, let us prove two lemmas.

Lemma 1. *Any array in the sets is either empty or full.*

Lemma 2. *Let S_i represent the state where only A_i is full and $A_0, A_1, \cdots, A_{i-1}$ is empty. In the process of changing S_0 to S_q using the ADDITION operation, the array A_p will be filled up 2^{q-p-1} times.*

The proof of the two lemmas are shown below the *Amortized time complexity analysis*.

Amortized time complexity analysis:
After n ADDITION operations, the set will contain n elements. Suppose k is the index of the array with the largest index in this set of n elements. We have:

$$2^k \leq n \leq 1 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$$

So

$$\log(n+1) - 1 \leq k \leq \log n$$

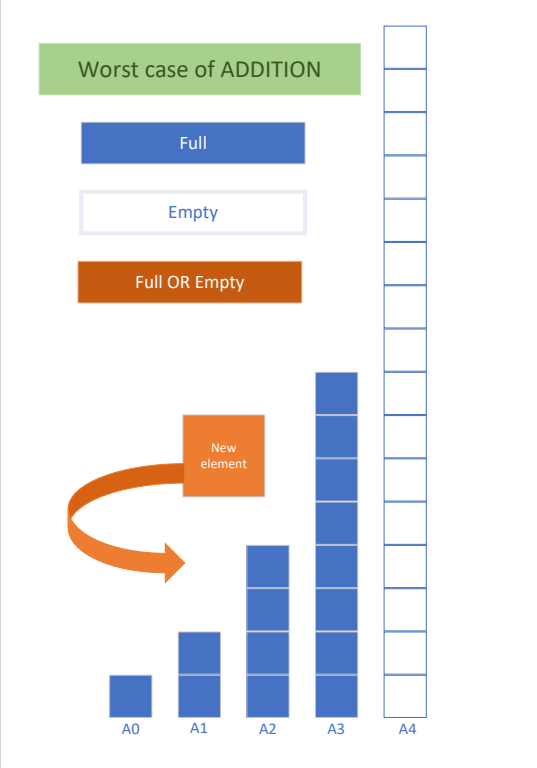

图 2: The worst case of adding an element in the set.

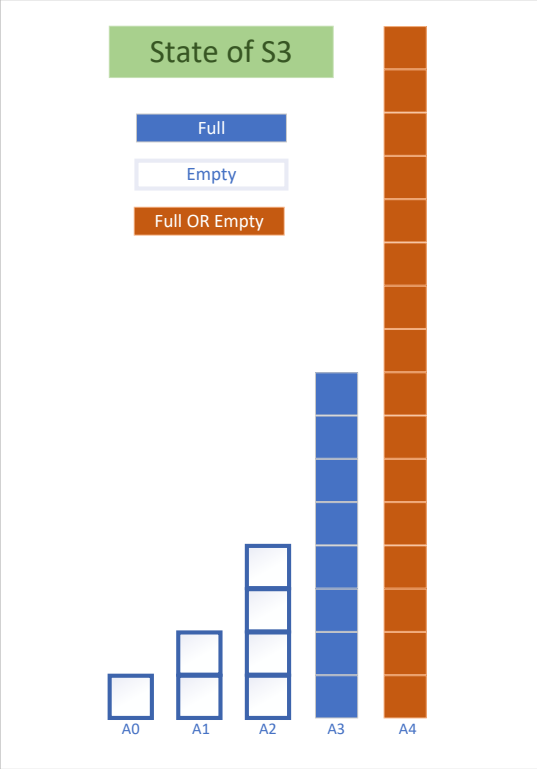

图 3: Implication of state S_3 in Lemma. **??**

which means

$$k \sim \Theta(\log n)$$

Let $\#FILL\ i$ denote the time complexity of changing the set from S_0 to S_i.

$$\begin{aligned} &Total\ amortized\ time\ complexity \\ &= \sum_{A_i\ is\ full} \#FILL\ i \\ &\leq \sum_{i=1}^{k} \#FILL\ i \\ &= \sum_{i=0}^{k-1} 2^{k-i-1} \times (2^i - 1) \\ &= (k-2)2^{k-1} - 1 \\ &= (\Theta(\log n) - 2)2^{\Theta(\log n)-1} - 1 \\ &= \Theta(n \log n) \end{aligned} \tag{14}$$

And

$$\text{Average time complexity} = \frac{O(n \log n)}{n} = O(\log n)$$

Proof of Lemma. ??.

Basis step: $n = 0$: A_0 is able to contain only one element, so it is either empty or full.
Induction hypothesis: Suppose that for any n, where $n \leq k$, A_n is either empty or full.
Proof of the induction step: Suppose A_i is the first array ahead of A_{k+1} with no element. The number of elements that need to be placed after the ADDITION operation is $(2^0 + 2^1 + \cdots + 2^{i-1}) + 1 = 2^i$, which is exactly the capacity of A_i. As a result, A_{k+1} remains empty after a ADDITION except for $i = k + 1$, which means A_{k+1} will be full after the operation. □

Proof of Lemma. ??.

Ahead of the operation which changes the state of the set to S_i, $A_0, A_1, \cdots, A_{q-1}$ must be full. In addition, the set must have gone through the states of $S_{q-1}, S_{q-2}, \cdots, Sp$ in turn. As a result, let $a_{p,q}$ denote the number of A_p being filled from S_0 to S_q, we could generate:

$$a_{p,q} = \sum_{i=p}^{q-1} a_{p,i}$$

Let $q \leftarrow q + 1$:

$$a_{p,q+1} = \sum_{i=p}^{q} a_{p,i}$$

By subtraction of the two formula we have:

$$a_{p,q+1} = 2a_{p,q} = \cdots = 2^{(q+1)-1-p} a_{p,p+1} = 2^{(q+1)-1-p}$$

□

□

(c) The worst case occurs when the n elements occupy $A_0, A_1, \cdots, A_k$ and each element in each array must be detected. The worst case time complexity is

$$\begin{aligned} \text{Time complexity} = & \sum_{i=0}^{k} O(\log 2^i) \\ = & O(\log 2^0 \times 2^1 \times \cdots \times 2^k) \\ = & O(\log 2^{\frac{(1+k)\times k}{2}}) \\ = & O(\log^2 n) \end{aligned} \tag{15}$$

since we have proved in (b) that $k \sim O(\log n)$.

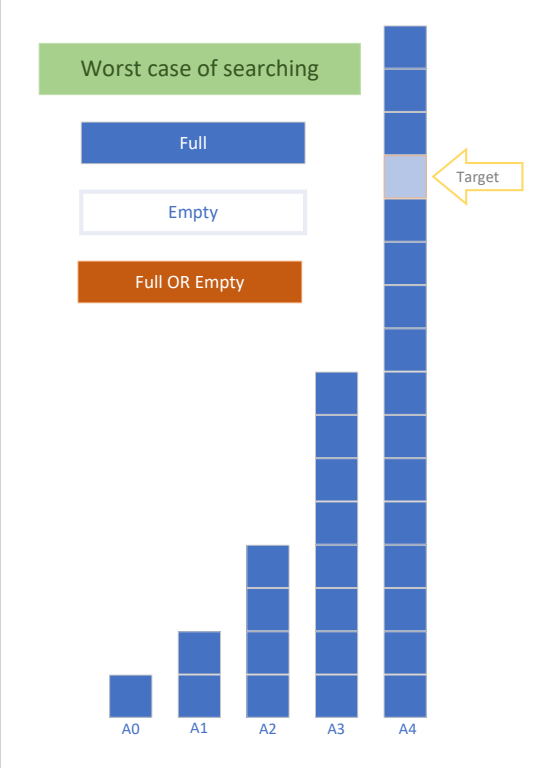

图 4: The worst case of searching an element in the set

(d) If $A_0, A_2, \cdots, A_{k-1}$ are full, the time complexity of the operation clearing $A_0, A_2, \cdots, A_{k-1}$ and occupying A_k is

$$\sum_{i=0}^{k-1} 2^i + O(2^{i+1} - 1) \tag{16}$$

The former 2^i is the time complexity of popping A_i. The later $O(2^{i+1} - 1)$ is the time complexity of merging sort, which is the fastest method to merge two sorted

sequence.
Equation. **??** is

$$\begin{aligned} &\sum_{i=0}^{k-1} 2^i + O(2^{i+1} - 1) \\ =&2^k - 1 + O(2^k) \\ =&O(2^k) \\ =&O(n) \end{aligned} \tag{17}$$

As a result, the step of merge won't increase time complexity. The amortized time complexity remains

$$\text{Amortized time complexity of single operation} = O(\log n)$$

□

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