Lab07-Amortized Analysis

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

∗ If there is any problem, please contact TA Yihao Xie.
∗ Name:Xinhao Zheng Student ID:519021910528 Email: void_zxh@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. For all the kinds of operations, assign amortized cost as follows:

Operation	Real Cost C_{op}	Amortized Cost $\widehat{C_{op}}$
i-th$(i \neq 2^x)$ operations	1	3
2^x-th operations	2^x	2

We regard the time complexity of these operations is $T(n)$ and then we have: $T(n) = \sum_{i=1}^{n} C_i$

Then, let's prove that

$$\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C}_i$$

by introduction.

Define $P(n)$ as the statement "$\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C}_i$".

Basis step. For $n = 1$ or $n = 2$, we have that: $C_1 = 1 \leq \widehat{C_1} = 3$ and $C_1 + C_2 = 1 + 2 = 3 \leq \widehat{C_1} + \widehat{C_2} = 6$. So $P(1)$ and $P(2)$ is true.

Induction hypothesis. For all $k \leq m$, $P(k)$ is true.

Proof of induction step. Let's prove $\sum_{i=1}^{m+1} C_i \leq \sum_{i=1}^{m+1} \widehat{C}_i$

We consider these two case below:

Case 1: $m + 1 \neq 2^x, x \in N$ In this case, $C_{m+1} = 1$

According to Induction hyothesis, we have: $\sum_{i=1}^{m} C_i \leq \sum_{i=1}^{m} \widehat{C}_i$.

Then, we get:

$$\begin{aligned}
\sum_{i=1}^{m+1} C_i &= \sum_{i=1}^{m} C_i + C_{m+1} \\
&\leq \sum_{i=1}^{m} \widehat{C}_i + 1 \\
&\leq \sum_{i=1}^{m} \widehat{C}_i + 3 \\
&= \sum_{i=1}^{m} \widehat{C}_i + C_{m+1} = \sum_{i=1}^{m+1} \widehat{C}_i
\end{aligned}$$

So, in this case, $P(m + 1)$ is ture.

Case 2: $m + 1 = 2^x, x \in N$ In this case, $C_{m+1} = 2^x$

Based on Induction hyothesis, we have: $\sum_{i=1}^{2^{x-1}} C_i \leq \sum_{i=1}^{2^{x-1}} \widehat{C}_i$.

Thus, we get:

$$\sum_{i=1}^{m+1} C_i = \sum_{i=1}^{2^{x-1}} C_i + \sum_{i=2^{x-1}+1}^{2^x} C_i$$
$$\leq \sum_{i=1}^{2^{x-1}} \widehat{C}_i + \sum_{i=2^{x-1}+1}^{2^x} C_i$$

Then, let prove that $\sum_{i=2^{x-1}+1}^{2^x} C_i = \sum_{i=2^{x-1}+1}^{2^x} \widehat{C}_i$

In fact, we have:

$$\sum_{i=2^{x-1}+1}^{2^x} C_i = (2^{x-1}-1) \times 1 + 1 \times 2^x$$

$$\sum_{i=2^{x-1}+1}^{2^x} \widehat{C}_i = (2^{x-1}-1) \times 3 + 1 \times 2 = (2^{x-1}-1) \times 1 + (2^{x-1}-1) \times 2 + 2$$
$$= (2^{x-1}-1) \times 1 + 2^x = \sum_{i=2^{x-1}+1}^{2^x} C_i$$

Thus, $\sum_{i=2^{x-1}+1}^{2^x} C_i = \sum_{i=2^{x-1}+1}^{2^x} \widehat{C}_i$.

So, we get:

$$\sum_{i=1}^{m+1} C_i \leq \sum_{i=1}^{2^{x-1}} \widehat{C}_i + \sum_{i=2^{x-1}+1}^{2^x} C_i = \sum_{i=1}^{2^{x-1}} \widehat{C}_i + \sum_{i=2^{x-1}+1}^{2^x} \widehat{C}_i = \sum_{i=1}^{m+1} \widehat{C}_i$$

All in all, we have $P(m+1)$ is true.

By the axiom of induction, we get: $\forall n \in N$, $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C}_i$

Thus, we have:

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

So $T(n) = O(n)$ and the amortized cost per operation is $O(1)$. □

Appendix: How to generate the idea of the Amortized Cost

This means that in this account method,if the former 2^{i-1} operations have cleared the credit, we let the latter 2^{i-1} operations can afford for themselves.

Thus every i-th($i \neq 2^x$) operation should pay for the 2^x-th operation.

We regard the cost of every i-th($i \neq 2^x$) operation as $1+P_x$ and the cost for the 2^x-th operation as P_y.

Then we get:

$$\widehat{Cost_{latter}} = (1+P_x) \times (2^{x-1}-1) + P_y$$
$$Cost_{latter} = 1 \times (2^{x-1}-1) + 2^x$$

So, based on the **basic idea of accounting method**, we have:

$$\begin{aligned} Cost_{latter} = 1 \times (2^{x-1} - 1) + 2^x \leq \widehat{Cost_{latter}} &= (1 + P_x) \times (2^{x-1} - 1) + P_y \\ 2^x &\leq P_x \times (2^{x-1} - 1) + P_y \\ 0 &\leq \left(\frac{P_x}{2} - 1\right) \times 2^x + (P_y - P_x) \end{aligned}$$

In fact, $\forall x \in N$, this statment above should be true and then we have:

$$\frac{P_x}{2} = 1 \text{ and } P_y = P_x$$

Thus, $P_x = P_y = 2$ and the cost of every i-th($i \neq 2^x$) operation is 3 and the cost for the 2^x-th operation is 2.

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. Firstly, we regard the state of he min-heap after the former i operations as S_i, the actual cost of i-th as C_i and the amortized cost as $\widehat{C}_i$.

So, we get:

$$\begin{aligned} C_i &= \begin{cases} O(\log n), & i\text{: INSERT instruction} \\ O(\log n), & i\text{: EXTRACT-MIN instruction} \end{cases} \\ &\leq \begin{cases} c \log n, & i\text{: INSERT instruction} \\ c \log n, & i\text{: EXTRACT-MIN instruction} \end{cases} \end{aligned}$$

Then, let's define the potential function as follows:

$$\Phi(S_i) = cn \log(n+1) - dn$$

which n is the number of element in the min-heap.

Thus, for these two operation, we have:

INSERT:

$$\begin{aligned} \Phi(S_i) - \Phi(S_{i-1}) &= cn \log(n+1) - c(n-1) \log n \\ \widehat{C}_i &= C_i + \Phi(S_i) - \Phi(S_{i-1}) \\ &= cn \log(n+1) - c(n-2) \log n - d \\ &\leq c_1 n \log n = O(n \log n) \ \ (c_1 \text{ is a constant, } c_1 > 0) \end{aligned}$$

INSERT:

$$\begin{aligned} \Phi(S_i) - \Phi(S_{i-1}) &= c(n-1) \log n - cn \log(n+1) \\ \widehat{C}_i &= C_i + \Phi(S_i) - \Phi(S_{i-1}) \\ &= cn \log n - cn \log(n+1) + d \\ &\leq c_2 = O(1) \ \ (c_2 \text{ is a constant, } c_2 > 0) \end{aligned}$$

Then, we have:

$$\widehat{C}_i = \begin{cases} O(\log n), & i\text{: INSERT instruction} \\ O(1), & i\text{: EXTRACT-MIN instruction} \end{cases}$$

Thus, this potential function is legal and satisfy the requirements.

Appendix: How to generate the idea of the potential function

Firstly, based on the target, we have:

$$\begin{aligned} \widehat{C}_i &= \begin{cases} O(\log n), & i\text{: INSERT instruction} \\ O(1), & i\text{: EXTRACT-MIN instruction} \end{cases} \\ &\leq \begin{cases} d_1 \log n, & i\text{: INSERT instruction} \\ d_2 & i\text{: EXTRACT-MIN instruction} \end{cases} \quad (d_1 \geq c,\ d_2 \geq 0) \end{aligned}$$

Based on the basic idea of the potential function, we have:

$$\widehat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1})$$

$$\Phi(S_i) - \Phi(S_{i-1}) = \widehat{C}_i - C_i$$

So, we get:

$$\Phi(S_i) - \Phi(S_{i-1}) = \begin{cases} (d_1 - c)\log n, & i\text{: INSERT instruction} \\ d_2 - c\log n, & i\text{: EXTRACT-MIN instruction} \end{cases}$$

Without loss of generality, we let $d_1 = 2c,\ d_2 = 0$.

Then, we have:

$$\Phi(S_i) - \Phi(S_{i-1}) = \begin{cases} c\log n, & i\text{: INSERT instruction} \\ -c\log n, & i\text{: EXTRACT-MIN instruction} \end{cases}$$

Consider the variable n for different states, we update this equation above as follow:

$$\Phi(S_i) - \Phi(S_{i-1}) = \begin{cases} cn\log(n+1) - c(n-1)\log n, & i\text{: INSERT instruction} \\ c(n-1)\log n - cn\log(n+1), & i\text{: EXTRACT-MIN instruction} \end{cases}$$

Then, eventually, we have:

$$\Phi(S_i) = cn\log(n+1)$$

where n is the number of elements in the min-heap on the state S_i and let S_0 be the state where n=0 and obviously have $\forall n \in N,\ \Phi(S_n) \geq \Phi(S_0)$ □

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?

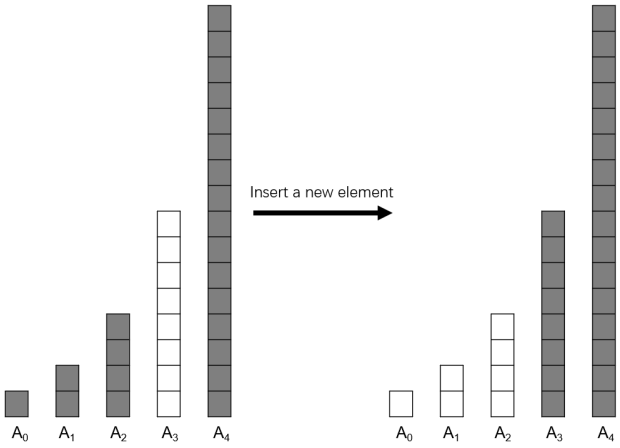

Figure 1: An example of making room for one new element in the set of arrays.

Solution. We suppose that the new element is added to A_{i+1} and regard that the time cost for popping or inserting an element is c and the total time cost for this adding is T. (These take $O(1)$ time)

Then, base on the question, we have:

$$T = c + \sum_{j=0}^{i} 2^j \times c = c + (2^{i+1} - 1) \times c = c \cdot 2^{i+1}$$

Thus, in the worst case, we have the biggest i which means that all the element all in the former $i + 1$ arrays and the former i is full.

So, we have:

$$n = \sum_{j=0}^{i} 2^j + X = 2^{i+1} - 1 + X,\ 0 \leq X \leq 2^{i+1}$$

Thus, we get:$i = \lfloor \log_2 n \rfloor$ and we get:

$$T = c \cdot 2^{i+1} = c \cdot 2^{\lfloor \log_2 n \rfloor + 1} = O(n)$$

So, in the worst case, it takes $O(n)$ time 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*.

Solution. Firstly, to prove that statement, we think the problem in another way which based on the Lemma below:

Lemma: $\forall i \in N$, array A_i is either full or empty.

Proof. **For** $i = 0$, the length of A_0 is 1 so that A_0 always be either full or empty.

For other $i > 0$, let's consider the time when the elements will be insert to A_i.

Because we always intend to insert the new element into A_0 and step by step to find a empty array to store the elements, A_i is empty before the inserting.

Then, when the elements are inserted into A_i, we can know that when the new element comes, the former $i - 1$ arrays are full so that there will be $1 + \sum_{j=0}^{i-1} 2^j = 2^i - 1 + 1 = 2^i$ elements inserted into A_i.

Thus, after inserting this new element, A_i will be full and in each process of inserting a new element, there will be either elements inserted into $A_i,\ i > 0$ or not.

So, array A_i, $i > 0$ is either full or empty.

All in all, $\forall i \in N$, array A_i is either full or empty.

Now, based on the **Lemma** above, actually we can use a binary number B_i to represent the state of array A_i. (0: empty, 1: full)

Thus, the process of inserting a new element can be regard as the process of incrementing a **binary counter** $B_0, B_1, ..., B_m, ...$

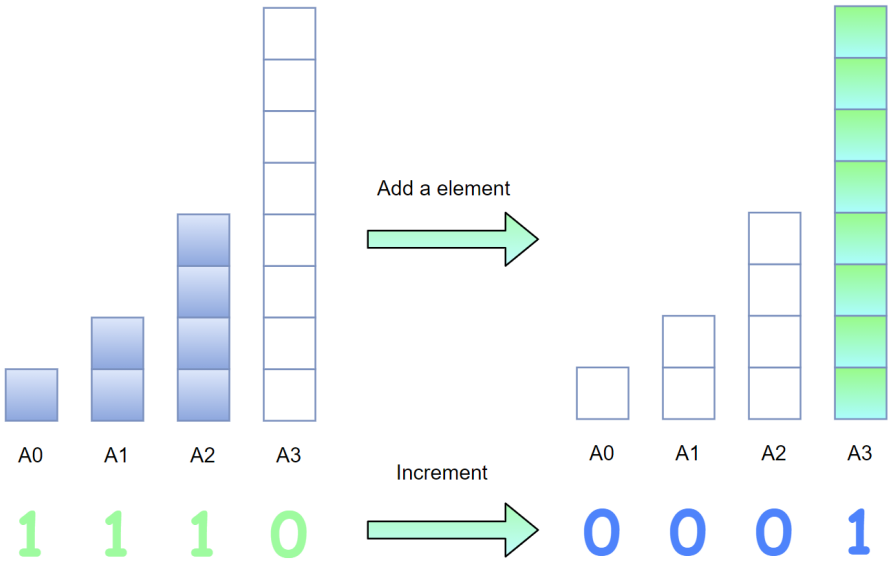

Figure 2: Arrays $\leftrightarrow$ Binary Counter

Then, let's prove the amortized cost of adding an element is $O(\log n)$.

Proof. Suppose that $T(n)$ is the amortized cost of it ,$T'(n)$ is the amortized cost for n times operations on the Binary counter, and C_i is the time cost of i-th operation from the beginning state.

Then we get:$T(n) = T'(n)$.

Because Inserting or popping an element take O(1) time, the operation $flip(1 \to 0)$, $filp(0 \to 1)$ in binary counting will take $O(2^i)$ time. (i is the number ID of arrays)

During a sequence of n INCREMENT operations which stand for the process of inserting a new element:

$$B_0 \text{ flips each time INCREMENT is called} \leftarrow n\text{times};$$
$$B_1 \text{ flips every other time} \leftarrow \lfloor \frac{n}{2} \rfloor \text{times};$$
$$\cdots \cdots \cdots$$
$$B_i \text{ flips} \lfloor \frac{n}{2^i} \rfloor \text{times} \cdots$$

Thus,

$$\begin{aligned} T(n) = T'(n) &= \sum_{i=1}^{n} C_i \\ &= 1 \times 2^0 + 2 \times 2^1 + 1 \times 2^0 + 4 \times 2^2 + \cdots \\ &= \#filp(B_0) \times 2^0 + \#flip(B_1) \times 2^1 + \cdots + \#filp(B_m) \times 2^m \\ &= n \times 2^0 + \frac{n}{2} \times 2^1 + \frac{n}{4} \times 2^2 + \cdots \\ &= mn = n \log n = O(n \log n) \end{aligned}$$

So 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. We regard T_{s_i} be the time cost for searching the array A_i. and $T(n)$ be the total time cost for searching.

Then based on the different method for searching, we get these outcoms below.

Method 1: Sequential search

If we use the Sequential search method to find the element, the time complexity of a m-element array can be $O(m)$.

Thus, we have:$T_{s_i} = O(l_i),\ l_i$ is the length of A_i

Without loss of generality, we let $T_{s_i} = c \cdot l_i$

Thus, we have:

$$T(n) = \sum_{i=0}^{+\infty} T_{s_i} = c \sum_{i=0}^{+\infty} l_i = cn = O(n)$$

So the total time cost is $O(n)$.

Method 2: Binary search

If we use the Binary search method to find the element, the time complexity of a m-element array can be $O(m \log m)$.

Thus, we have:$T_{s_i} = O(l_i \log l_i),\ l_i$ is the length of A_i

Without loss of generality, we let $T_{s_i} = c \cdot \log l_i$

We let $n = \sum_{j=0}^{i} 2^j + X = 2^{i+1} - 1 + X,\ 0 < X \leq 2^{i+1}$ so we have:

$$T(n) = \sum_{j=0}^{+\infty} T_{s_j} = c \sum_{j=0}^{+\infty} \log l_j = c \left(\sum_{j=0}^{i} j + \log X \right) \leq \sum_{j=0}^{i+1} j = \frac{(i+1)(i+2)}{2} = O(\log^2 n)$$

So, $T(n) = O(\log^2 n)$ and it takes $O(\log^2 n)$ time to search an element in the arrays.

All in all, we have:

$$T(n) = \begin{cases} O(n), & \text{using Sequential search} \\ O(\log^2 n), & \text{using Binary search} \end{cases}$$

□

(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. In this case, when the new element is added to A_{i+1}, we need to merge the former i arrays.

Here, we use the idea of **Merge Sort Algorithm** to merge two arrays.

Because the comparison between two elements takes $O(1)$ time, without loss of generality, we regard the cost of ti as 1.

Then, the cost for merge the m-length array and n-length array is $(m+n)$.

Here, we have to merge i arrays with the length $1, 2, 4, \cdots$.

Obviously, based on the **Greedy algorithm** of problem *Merging fruits*, we should merge from small array to the big step by step to get the optimal time cost which means that we merge A_0, A_1, $A_0\&A_1, A_2$ and so on. The example of Merge process is shown below.

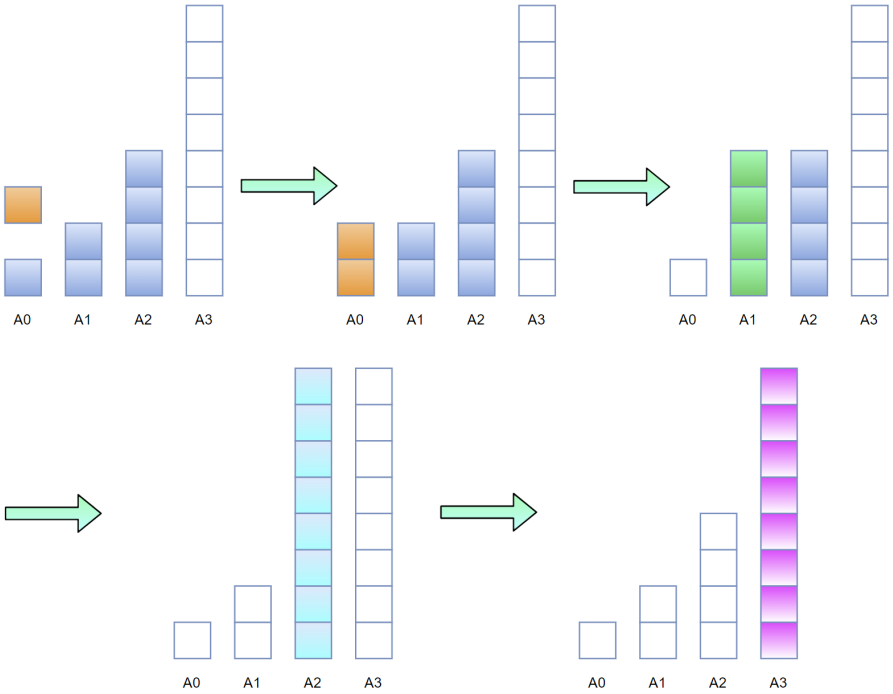

Figure 3: Merge process

We regard the time cost for merge the former i arrays as T_{M_i} and the length of array A_i as l_i.

Then, we have:

$$\begin{aligned} T_{M_i} &= (1+1) + (2+2) + ((2+2)+4) + ((2+2+4)+8) + \cdots \\ &= \sum_{j=1}^{i} \left(1 + \sum_{k=0}^{j-1} l_k + l_j\right) \\ &= \sum_{j=1}^{i} (\sum_{k=0}^{j} l_k + 1) \\ &= \sum_{j=1}^{i} 2^{j+1} = 2^{i+2} - 4 = O(2^i) \end{aligned}$$

Besides, the time cost of inserting and popping during this process is $O(2^i)$ so the total time cost for inserting a new element into A_{i+1} is $O(2^i)$ and let's define it as T_{sum_i} $(T_{sum_i} \leq c \cdot 2^i)$.

Based on that, let's consider the amortized cost of adding an element in the case of (c) and regard it as $T(n)$.

Thus, we have:

$$\begin{aligned} T(n) &= T_{sum_0} + T_{sum_1} + T_{sum_0} + T_{sum_2} + \cdots \\ &= cn \times 2^0 + \frac{cn}{2^1} \times 2^1 + \frac{cn}{2^2} \times 2^2 + \cdots \\ &= cn \log n = O(n \log n) \end{aligned}$$

So the amortized cost of adding an element is $\frac{T(n)}{n} = O(\log n)$. □

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