Chapter 17 Solusion

https://github.com/frc123/CLRS

12/28/2021

17.1

17.1-1

No. Consider we operate MULTPUSH(S, n) n times. Such n operations cost $\Theta(n^2)$, so the amortized cost is $\Theta(n)$.

Actually, we can MULTPUSH incredible large amount of items, so $O(1)$ of course cannot be bound on the amortized cost of stack operations.

17.1-2

Consider a k-bit counter where each bit in the counter is 1. Now, we perform INCREMENT which flips $k + 1$ bits. Then, we perform DECREMENT which flips $k + 1$ bits again. Hence perform a sequence of length n operations $\langle$INCREMENT, DECREMENT, INCREMENT, DECREMENT, $\cdots$ $\rangle$ cost $\Theta(nk)$ in total.

17.1-3

$$n + \sum_{i=1}^{\lfloor \lg n \rfloor} (2^i - 1) \le n + \sum_{i=0}^{\lg n} 2^i = n + 2^{\lg n + 1} - 1 = n + 2n - 1 = 3n - 1$$

Hence the amortized cost per operation is $O(1)$.

17.2

17.2-1

operation	actual cost	amortized cost
PUSH	1	2
POP	1	2
Copy	s	0

where s is the stack size when it is called which has an upper bound k.

Each operation (PUSH or POP) charges an amortized cost of 2 and actual use 1. After k operations, we have k credits, and copy operation cost at most k. Hence we conclude the total amortized cost is greater than the total actual cost at all times.

17.2-2

Let the amortized cost of each operation be 3. We want to show that

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

for all integers n where

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2,} \\ 1 & \text{otherwise} \end{cases}$$

and $\hat{c}_i = 3$ for all integers i. That is we want to show that

$$3n \geq n + \sum_{i=1}^{\lfloor \lg n \rfloor} (2^i - 1).$$

By exercise 17.1-3, we have

$$n + \sum_{i=1}^{\lfloor \lg n \rfloor} (2^i - 1) \leq 3n - 1.$$

Hence the amortized cost per operation is $O(1)$.

17.2-3

As the hint mentioned, we keep a pointer to the high-order 1 and maintain it during the operations. In each INCREMENT operation, we check if we the high-order 1 moved to a higher order.

Fliping a bit charges 1. Moving the pointer to the high-order 1 charges \$1. Let the amortized cost of each INCREMENT operation be \$4, and let the amortized cost of each RESET operation be \$1. When we set a bit to 1, we actually cost \$1 and retain \$2 as credits for the purpose of setting to 0 and resetting. If we need to update pointer, we charge another \$1. Hence amortized cost of each INCREMENT operation is \$4. Each RESET operation need to move the pointer to -1, so it costs \$1.

```cpp
struct Counter
{
    int length;
    std::vector<bool> bits;
    int high_order_one;

    Counter(int length) : length(length),
        bits(length, 0), high_order_one(-1) {}
};
```

```
10
11   void Increment(Counter& counter)
12   {
13       int i;
14       i = 0;
15       while (i < counter.length && counter.bits[i] == 1)
16       {
17           counter.bits[i] = 0;
18           ++i;
19       }
20       if (i < counter.length)
21       {
22           counter.bits[i] = 1;
23           counter.high_order_one = std::max(i, counter.high_order_one);
24       }
25       else
26       {
27           // overflow
28           counter.high_order_one = -1;
29       }
30   }
31
32   void Reset(Counter& counter)
33   {
34       int i;
35       for (i = 0; i < counter.length; ++i)
36       {
37           counter.bits[i] = 0;
38       }
39       counter.high_order_one = -1;
40   }
```

17.3

17.3-1

Let $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$. Clearly, $\Phi'(D_0) = 0$. We claim the amortized costs using Φ' are the same as the amortized costs using Φ.

$$\hat{c}_i = c_i + \Phi'(D_i) - \Phi(D_{i-1})$$
$$= c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_{i-1}) - \Phi(D_0))$$
$$= c_i + \Phi(D_i) - \Phi(D_{i-1})$$

17.3-2

Let $\Phi(D_0) = 0$ and $\Phi(D_i) = 2(i - 2^{\lfloor \lg i \rfloor})$ for $i \geq 1$.

$$\Phi(D_i) - \Phi(D_{i-1}) = 2(i - 2^{\lfloor \lg i \rfloor}) - 2((i - 1) - 2^{\lfloor \lg(i-1) \rfloor})$$
$$= 2 - 2(2^{\lfloor \lg i \rfloor} - 2^{\lfloor \lg(i-1) \rfloor})$$

Note that

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2,} \\ 1 & \text{otherwise} \end{cases}$$

Case 1. i is an exact power of 2.

$$\Phi(D_i) - \Phi(D_{i-1}) = 2 - 2(i - \frac{i}{2})$$
$$= 2 - i$$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= i + 2 - i$$
$$= 2$$

Case 2. i is not an exact power of 2.
Then $2^{\lfloor \lg i \rfloor} = 2^{\lfloor \lg(i-1) \rfloor}$.

$$\Phi(D_i) - \Phi(D_{i-1}) = 2$$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 1 + 2$$
$$= 3$$

Hence the amortized cost per operation is $O(1)$.

17.3-3

The idea is to let the potential be proportional to the sum of the height of every node in the min-heap. Note that an binary heap is a complete binary tree.

$$\sum_{j=1}^{n} \lfloor \lg j \rfloor \leq \lg(n!) \leq n \lg n$$

Let Φ be

$$\Phi(D_i) = \begin{cases} 0 & \text{if } n_i = 0, \\ kn_i \lg n_i & \text{if } n_i > 0 \end{cases}$$

for some constant k where n_i is the number of nodes in D_i. Also, we have

$$c_i \leq \begin{cases} k_1 \lg n_i & \text{if INSERT is performed in the } i\text{th operation and } n_i \geq 2, \\ k_2 \lg n_{i-1} & \text{if EXTRACT-MIN is performed in the } i\text{th operation and } n_{i-1} \geq 2 \end{cases}$$

Let $k = \max(k_1, k_2)$.

Case 1. INSERT is performed in the ith operation. Then $n_i - 1 = n_{i-1}$.

If $n_i = 1$,

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= c_i$$

If $n_i \geq 2$,

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$\leq k \lg n_i + kn_i \lg n_i - kn_{i-1} \lg n_{i-1}$$
$$= k(\lg n_i + n_i \lg n_i - n_{i-1} \lg n_{i-1})$$
$$= k(\lg n_i + n_i \lg n_i - (n_i - 1) \lg(n_i - 1))$$
$$= k(\lg n_i + n_i \lg n_i - n_i \lg(n_i - 1) + \lg(n_i - 1))$$
$$< k(2 \lg n_i + n_i(\lg n_i - \lg(n_i - 1)))$$

Note that $\forall x \in \mathbb{R}, 1 + x \leq e^x$. Then

$$n_i(\lg n_i - \lg(n_i - 1)) = n_i \lg \frac{n_i}{n_i - 1}$$
$$= n_i \lg(1 + \frac{1}{n_i - 1})$$
$$\leq n_i \lg(e^{\frac{1}{n_i - 1}})$$
$$= \frac{n_i}{n_i - 1} \lg e$$
$$= (1 + \frac{1}{n_i - 1}) \lg e$$
$$\leq 2 \lg e$$

Hence

$$\hat{c}_i < k(2 \lg n_i + 2 \lg e)$$

We conclude $\hat{c}_i = O(\lg n)$ for INSERT.

Case 2. EXTRACT-MIN is performed in the ith operation. Then $n_{i-1} - 1 = n_i$.
If $n_{i-1} = 1$,

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= c_i$$

If $n_{i-1} \geq 2$,

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$\leq k \lg n_{i-1} + kn_i \lg n_i - kn_{i-1} \lg n_{i-1}$$
$$= k(\lg n_{i-1} + n_i \lg n_i - n_{i-1} \lg n_{i-1})$$
$$= k(\lg n_{i-1} + (n_{i-1} - 1) \lg(n_{i-1} - 1) - n_{i-1} \lg n_{i-1})$$
$$< k(\lg n_{i-1} - \lg(n_{i-1} - 1))$$
$$= k \lg(1 + \frac{1}{n_{i-1} - 1})$$
$$\leq k \lg e^{\frac{1}{n_{i-1}-1}}$$
$$= \frac{k}{n_{i-1} - 1} \lg e$$

We conclude $\hat{c}_i = O(1)$ for EXTRACT-MIN.

17.3-4

$$\Phi(D_n) - \Phi(D_0) = s_n - s_0$$

Since $\hat{c}_i = 2$,

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0)$$
$$= 2n + s_0 - s_n$$

17.3-5

$$\Phi(D_0) = b$$

Since $\hat{c}_i \leq 2$,

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0)$$
$$\leq 2n + b - \Phi(D_n)$$

Since $\Phi(D_n) \geq 0$,

$$\sum_{i=1}^{n} c_i \leq 2n + b$$

Since $n = \Omega(b)$,

$$\sum_{i=1}^{n} c_i = O(n)$$

17.3-6

```cpp
template <typename T>
class Queue
{
public:
    void Enqueue(T& x);
    void Enqueue(T&& x);
    T Dequeue();
private:
    std::stack<T> s_a_;
    std::stack<T> s_b_;
};

template <typename T>
void Queue<T>::Enqueue(T& x)
{
    s_a_.push(x);
}

template <typename T>
void Queue<T>::Enqueue(T&& x)
{
    s_a_.emplace(std::move(x));
}

template <typename T>
T Queue<T>::Dequeue()
{
    if (s_b_.empty())
    {
        while (s_a_.empty() == false)
        {
            s_b_.emplace(std::move(s_a_.top()));
            s_a_.pop();
        }
    }
```

```
36        T top = std::move(s_b_.top());
37        s_b_.pop();
38        return std::move(top);
39    }
```

Assume each of $s\_a\_.push$ (or $emplace$), $s\_a\_.pop$, $s\_b\_.push$ (or $emplace$), $s\_b\_.pop$ costs \$1. Then

$$c_i = \begin{cases} 1 & \text{if ENQUEUE is performed in the } i\text{th operation,} \\ 1 & \text{if DEQUEUE is performed in the } i\text{th operation and } D_{i-1}.s\_b\_ \text{ is not empty,} \\ 2 \cdot (D_{i-1}.s\_a\_.size()) + 1 & \text{if DEQUEUE is performed in the } i\text{th operation and } D_{i-1}.s\_b\_ \text{ is empty} \end{cases}$$

Let
$$\Phi(D_i) = 3 \cdot (D_i.s\_a\_.size()) + (D_i.s\_b\_.size())$$

Case 1. ENQUEUE is performed in the ith operation.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 1 + 3 \cdot (D_i.s\_a\_.size() - D_{i-1}.s\_a\_.size()) + (D_i.s\_b\_.size() - D_{i-1}.s\_b\_.size())$$
$$= 1 + 3 \cdot 1 + 0$$
$$= 4$$

Case 2. DEQUEUE is performed in the ith operation and $D_{i-1}.s\_b\_$ is not empty.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 1 + 3 \cdot (D_i.s\_a\_.size() - D_{i-1}.s\_a\_.size()) + (D_i.s\_b\_.size() - D_{i-1}.s\_b\_.size())$$
$$= 1 + 3 \cdot 0 - 1$$
$$= 0$$

Case 3. DEQUEUE is performed in the ith operation and $D_{i-1}.s\_b\_$ is empty.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= (2 \cdot (D_{i-1}.s\_a\_.size()) + 1) + 3 \cdot (D_i.s\_a\_.size() - D_{i-1}.s\_a\_.size()) + (D_i.s\_b\_.size() - D_{i-1}.s\_b\_.size())$$
$$= (2 \cdot (D_{i-1}.s\_a\_.size()) + 1) - 3 \cdot (D_{i-1}.s\_a\_.size()) + (D_{i-1}.s\_a\_.size() - 1)$$
$$= 0$$

Thus, we conclude that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

17.3-7

Note that section 9.3 provides an approach of selection in worst-case linear time.

```cpp
1   class DataStructure
2   {
3   public:
4       void Insert(int x);
5       void DeleteLargerHalf();
6       const std::vector<int>& Get() const;
7   private:
8       std::vector<int> arr_;
9   };
10
11  void DataStructure::Insert(int x)
12  {
13      arr_.push_back(x);
14  }
15
16  void DataStructure::DeleteLargerHalf()
17  {
18      size_t median = arr_.size() >> 1;
19      LinearSelect(arr_, 0, arr_.size() - 1, (arr_.size() - 1) >> 1);
20      arr_.erase(arr_.begin() + median, arr_.end());
21  }
22
23  const std::vector<int>& DataStructure::Get() const
24  {
25      return arr_;
26  }
```

Assume

$$
c_i = \begin{cases} 1 & \text{if INSERT is performed in the } i\text{th operation,} \\ n_{i-1} & \text{if DELETE-LARGER-HALF is performed in the } i\text{th operation} \end{cases}
$$

where n_i is $|S|$ after the ith operation. Let

$$\Phi(D_i) = 2n_i$$

be the potential function of the data structure.

Case 1. INSERT is performed in the ith operation.

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= c_i + 2(n_i - n_{i-1}) \\
&= 1 + 2 \cdot 1 \\
&= 3
\end{aligned}
$$

Case 2. Delete-Larger-Half is performed in the ith operation.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= c_i + 2(n_i - n_{i-1})$$
$$= n_{i-1} + 2(\frac{n_{i-1}}{2} - n_{i-1})$$
$$= n_{i-1} - n_{i-1}$$
$$= 0$$

17.4

17.4-1

By Theorem 11.6 and Theorem 11.8, assuming uniform hashing, for $alpha < 1$, we know the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$ and in an successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.

$$\lim_{\alpha \to 1^-} \frac{1}{1-\alpha} = \infty$$

$$\lim_{\alpha \to 1^-} \frac{1}{\alpha} \ln \frac{1}{1-\alpha} = \infty$$

Actually, when $\alpha = 1$, an unsuccessful search costs $\Theta(m) = \Theta(n)$. If we can bound α above by some constant that is strictly less than 1, the expected time of an unsuccessful or successful search is bounded above by some constant also.

Consider the function

$$\Phi_i = 2 \cdot num_i - \beta \cdot size_i$$

Let $\Phi_0 = 0$.

We just need to simply modify the conditional statement in line 4 of Table-Insert to

if $T.num + 1 > \beta \cdot T.size$

where β is some constant that is strictly less than 1 and modify the base case of resizing table by modifing line 3 of Table-Insert to

if $T.size = \lceil \frac{1}{\beta} \rceil$

We want to show that one expansion (twice the size) is enough in order to insert an element into a full table ($T.num + 1 > \beta \cdot T.size$).

Lemma 1. Assume $num \geq 1$. $num \leq \beta \cdot size \implies num + 1 \leq 2\beta \cdot size$

Proof. Note that $\forall x \geq 1, x + 1 \leq 2x$.

$$num \leq \beta \cdot size \iff 2 \cdot num \leq 2\beta \cdot size$$

$$\iff num + 1 \leq 2 \cdot num \leq 2\beta \cdot size$$

$\square$

Claim 2. $num_i \leq \beta \cdot size_i$ for all i.

Proof. We prove by induction. WLOG, assume inserts are performed for all i. Then $num_{i+1} = i+1$.

(Base) $k = 0$: $num_0 = 0 \leq \beta \cdot 0 = \beta \cdot size_i$

$k = 1$: $num_1 = 1 \leq \beta \cdot \lceil \frac{1}{\beta} \rceil = \beta \cdot size_i$ since $x \cdot \lceil \frac{1}{x} \rceil \geq x \cdot \frac{1}{x} = 1$ for all $x > 0$.

(Induction) Fix $k \geq 1$. Suppose that $num_k \leq \beta \cdot size_k$.

Case 1. $num_k + 1 \leq \beta \cdot size_k$

Then $size_{k+1} = size_k$.

$$num_{k+1} = num_k + 1 \leq \beta \cdot size_k = \beta \cdot size_{k+1}$$

Case 2. $num_k + 1 > \beta \cdot size_k$

Then $size_{k+1} = 2 \cdot size_k$. By lemma 1 and inductive hypothesis, we have $num_k + 1 \leq 2\beta \cdot size_k$.

$$num_{k+1} = num_k + 1 \leq 2\beta \cdot size_k = \beta \cdot size_{k+1}$$

$\square$

We also want to show that $\Phi(T)$ is always nonnegative.

Claim 3. $\Phi_i \geq 0$ for all i

Proof. We prove by induction. WLOG, assume inserts are performed for all i. Then $num_{i+1} = i+1$.

(Base)

$$\Phi_0 = 0$$

$$\Phi_1 = 2 \cdot num_1 - \beta \cdot size_1 = 2 \cdot 1 - \beta \cdot \lceil \frac{1}{\beta} \rceil > 2 - \beta \cdot (\frac{1}{\beta} + 1) = 1 - \beta > 0$$

(Induction) Fix $k \geq 1$. Suppose that $\Phi_k = 2 \cdot num_k - \beta \cdot size_k \geq 0$.

Case 1. $num_k + 1 \leq \beta \cdot size_k$

Then $size_{k+1} = size_k$.

$$\Phi_{k+1} = 2 \cdot num_{k+1} - \beta \cdot size_{k+1} = 2 \cdot (num_k + 1) - \beta \cdot size_k > 2 \cdot num_k - \beta \cdot size_k \overset{\text{IH}}{\geq} 0$$

Case 2. $num_k + 1 > \beta \cdot size_k$

Then $size_{k+1} = 2 \cdot size_k$.

$$\Phi_{k+1} = 2 \cdot num_{k+1} - \beta \cdot size_{k+1} = 2 \cdot (num_k + 1) - 2\beta \cdot size_k = 2 \cdot (num_k + 1 - \beta \cdot size_k) > 0$$

Since $num_k + 1 - \beta \cdot size_k > 0$ $\square$

We want to analysis the expected amortized cost. By Theorem 11.6, we assume

$$E[c_i] = \begin{cases} 1 & \text{if the } i\text{th insert operation does not trigger an expansion,} \\ num_i & \text{if the } i\text{th insert operation does trigger an expansion} \end{cases}$$

If the ith insert operation does not trigger an expansion, then we have $size_i = size_{i-1}$, and the expected amortized cost of the operation is

$$
\begin{aligned}
E[\hat{c}_i] &= E[c_i] + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - \beta \cdot size_i) - (2 \cdot num_{i-1} - \beta \cdot size_{i-1}) \\
&= 1 + (2 \cdot num_i - \beta \cdot size_i) - (2 \cdot (num_i - 1) - \beta \cdot size_i) \\
&= 3 .
\end{aligned}
$$

If the ith insert operation does trigger an expansion, then we have $size_i = 2 \cdot size_{i-1}$, and the expected amortized cost of the operation is

$$
\begin{aligned}
E[\hat{c}_i] &= E[c_i] + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \cdot num_i - \beta \cdot size_i) - (2 \cdot num_{i-1} - \beta \cdot size_{i-1}) \\
&= (num_{i-1} + 1) + (2 \cdot (num_{i-1} + 1) - \beta \cdot 2 \cdot size_{i-1}) - (2 \cdot num_{i-1} - \beta \cdot size_{i-1}) \\
&= 3 + num_{i-1} - \beta \cdot size_{i-1} \\
&< 3. \qquad\qquad \text{(by claim 2)}
\end{aligned}
$$

Note that $E[c_i] = num_i$, which is linear, in this case because we need to copy all num_{i-1} elements to the new allocated table.

17.4-2

Claim 4. $\forall num \geq 2, \frac{num}{size} \geq \frac{1}{2} \implies \frac{num-1}{size} \geq \frac{1}{4}$

Proof.

$$
\begin{aligned}
\frac{num}{size} \geq \frac{1}{2} &\iff 2 \cdot num \geq size \\
&\iff 4(num - 1) \geq 2 \cdot num \geq size \qquad\qquad \text{(since } num \geq 2) \\
&\iff \frac{num - 1}{size} \geq \frac{1}{4}
\end{aligned}
$$

$\square$

Suppose that $\alpha_{i-1} \geq \frac{1}{2}$ and the ith operation is TABLE-DELETE. Then $num_i = num_{i-1} - 1$. By the claim, we know that a contraction will not be triggered in the ith operation if $\alpha_{i-1} \geq \frac{1}{2}$ and $num_{i-1} \geq 2$. If $num_{i-1} = 1$, then $num_i = 0$, which is trivial. Assume $num_{i-1} \geq 2$ in the following analysis. Then $c_i = 1$ and $size_i = size_{i-1}$.

Case 1. $\alpha_i < \frac{1}{2}$.

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$$= 1 + (\frac{size_i}{2} - num_i) - (2 \cdot num_{i-1} - size_{i-1})$$

$$= 1 + (\frac{size_{i-1}}{2} - (num_{i-1} - 1)) - (2 \cdot num_{i-1} - size_{i-1})$$

$$= 2 + \frac{3}{2} \cdot size_{i-1} - 3 \cdot num_{i-1}$$

$$= 2 + \frac{3}{2} \cdot size_{i-1} - 3\alpha_{i-1} \cdot size_{i-1}$$

$$\leq 2 + \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1}$$

$$= 2$$

Case 2. $\alpha_i \geq \frac{1}{2}$.

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$$= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$$

$$= 1 + (2 \cdot (num_{i-1} - 1) - size_{i-1}) - (2 \cdot num_{i-1} - size_{i-1})$$

$$= -1$$

17.4-3

Suppose that TABLE-DELETE is performed in the ith operation. Then $num_i = num_{i-1} - 1$.

Case 1. $\frac{num_{i-1} - 1}{size_{i-1}} \geq \frac{1}{3}$ (a contraction is not triggered in the ith operation).

Then we have $c_i = 1$ and $size_i = size_{i-1}$.

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$$= c_i + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}|$$

$$= 1 + |2 \cdot (num_{i-1} - 1) - size_{i-1}| - |2 \cdot num_{i-1} - size_{i-1}|$$

$$\overset{\triangle}{\leq} 1 + (|2 \cdot num_{i-1} - size_{i-1}| + |-2|) - |2 \cdot num_{i-1} - size_{i-1}|$$

$$= 3$$

Case 2. $\frac{num_{i-1} - 1}{size_{i-1}} < \frac{1}{3}$ (a contraction is triggered in the ith operation).

Then we have $c_i = num_i + 1 = num_{i-1}$ and $size_i = \frac{2}{3} \cdot size_{i-1}$.

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$$= c_i + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}|$$

$$= num_{i-1} + |2 \cdot (num_{i-1} - 1) - \frac{2}{3} \cdot size_{i-1}| - |2 \cdot num_{i-1} - size_{i-1}|$$

$$= num_{i-1} + |2 \cdot num_{i-1} - \frac{2}{3} \cdot size_{i-1} - 2| - |2 \cdot num_{i-1} - size_{i-1}|$$

Lemma 5.

$$2 \cdot num_{i-1} - \frac{2}{3} \cdot size_{i-1} - 2 < 0$$

Proof.

$$\frac{num_{i-1} - 1}{size_{i-1}} < \frac{1}{3} \implies 3 \cdot (num_{i-1} - 1) < size_{i-1}$$

$$\implies 2 \cdot (num_{i-1} - 1) < \frac{2}{3} \cdot size_{i-1}$$

$$\implies 2 \cdot num_{i-1} - \frac{2}{3} \cdot size_{i-1} - 2 < 0$$

$\square$

Lemma 6.

$$\forall num_{i-1} \geq 3, 2 \cdot num_{i-1} - size_{i-1} < 0$$

Proof.

$$\frac{num_{i-1} - 1}{size_{i-1}} < \frac{1}{3} \implies 3 \cdot (num_{i-1} - 1) < size_{i-1}$$

$$\implies 2 \cdot num_{i-1} - 3 + num_{i-1} < size_{i-1}$$

$$\implies 2 \cdot num_{i-1} - size_{i-1} + (num_{i-1} - 3) < 0$$

$$\implies 2 \cdot num_{i-1} - size_{i-1} < 3 - num_{i-1}$$

Clearly, $\forall num_{i-1} \geq 3, 3 - num_{i-1} \leq 0$. $\square$

The subcase of $num_{i-1} < 3$ is trivial. Assume $num_{i-1} \geq 3$ in the later analysis.

Then

$$\hat{c}_i = num_{i-1} + |2 \cdot num_{i-1} - \frac{2}{3} \cdot size_{i-1} - 2| - |2 \cdot num_{i-1} - size_{i-1}|$$

$$= num_{i-1} - (2 \cdot num_{i-1} - \frac{2}{3} \cdot size_{i-1} - 2) \overset{(--)}{+} (2 \cdot num_{i-1} - size_{i-1})$$

$$= (num_{i-1} - \frac{1}{3} \cdot size_{i-1} - 1) + 3$$

Lemma 7.

$$num_{i-1} - \frac{1}{3} \cdot size_{i-1} - 1 < 0$$

Proof.

$$\frac{num_{i-1} - 1}{size_{i-1}} < \frac{1}{3} \implies num_{i-1} - 1 < \frac{1}{3} \cdot size_{i-1}$$

$$\implies num_{i-1} - \frac{1}{3} \cdot size_{i-1} - 1 < 0$$

$\square$

Hence

$$\hat{c}_i = (num_{i-1} - \frac{1}{3} \cdot size_{i-1} - 1) + 3 < 3$$

Chapter 17 Problems

17-1

(a)

```
1   template <typename T>
2   void BitReversal(std::vector<T>& arr)
3   {
4       size_t k, tmp, i, n, rev;
5       n = arr.size();
6       std::vector<bool> counter(n, false);
7       tmp = n;
8       k = -1;
9       while (tmp)
10      {
11          tmp = tmp >> 1;
12          ++k;
13      }
14      for (i = 0; i < n; ++i)
15      {
16          if (counter[i] == false)
17          {
18              rev = Rev(k, i);
19              std::swap(arr[i], arr[rev]);
20              counter[i] = true;
21              counter[rev] = true;
22          }
23      }
24  }
```

(b)

Note that
$$\mathrm{rev}_k(\langle a_{k-1}, a_{k-2}, \cdots, a_0 \rangle) = \langle a_0, a_1, \cdots, a_{k-1} \rangle$$

In order to find $rev_k(a) + 1$, we just need to call INCREMENT on $rev_k(a)$. We observed that we can modify INCREMENT by starting iteration from the high order bit to the low order bit in order to find $rev_k(rev_k(a) + 1) + 1$.

```
1   size_t BitReversedIncrement(size_t k, size_t a)
2   {
3       size_t i;
```

```
4        i = 1 << (k - 1);
5        while (i > 0 && (a & i) != 0)
6        {
7            a = a & ( ˜ i );
8            i = i >> 1;
9        }
10       a = a | i;
11       return a;
12   }
```

Similar to the analysis of INCREMENT, successive call to BIT-REVERSED-INCREMENT produce the sequence in a total of $O(n)$ time.

(c)

Yes. We can modify our BIT-REVERSED-INCREMENT by precompute value of $1 << (k - 1)$ before the first call in order to prevent recomputing this value in each call.

Observed the operation of $i = i >> 1$ always following flipping the bit back to 0. Hence we can use the same analysis in this situation.

17-2

```
1    template <typename T>
2    class DynamicBinarySearch
3    {
4    public:
5        using Iterators = std::pair
6            <typename std::list< std::vector<T> >::iterator,
7            typename std::vector<T>::iterator>;
8        Iterators Search(const T& key);
9        void Insert(const T& key);
10       void Delete(const Iterators& its);
11   private:
12       std::list< std::vector<T> > arrays_;
13   };
```

(a)

```
1    template <typename T>
2    typename DynamicBinarySearch<T>::Iterators
3        DynamicBinarySearch<T>::Search(const T& key)
4    {
```

```
5        size_t n, i;

6        n = arrays_.size();

7        for (typename std::list< std::vector<T> >::iterator it = arrays_.begin();

8            it != arrays_.end(); ++it)

9        {

10           typename std::vector<T>::iterator sub_it =

11               std::lower_bound(it->begin(), it->end(), key);

12           if (sub_it != it->end() && *sub_it == key)

13               return std::make_pair(it, sub_it);

14       }

15       throw std::out_of_range("the container does not have an element "

16                               "with the specified key");

17   }
```

In the worst case, $n_i = 1$ for all $i = 0, 1, \cdots, k-1$ and the binary search is unsuccessful. Then the running time in the worst case is

$$\Theta(\sum_{i=0}^{k-1} n_i \lg(2^i)) = \Theta(\sum_{i=0}^{k-1} i) = \Theta(k^2) = \Theta(\lg^2 n)$$

(b)

```
1    template <typename T>

2    void DynamicBinarySearch<T>::Insert(const T& key)

3    {

4        size_t n;

5        std::vector<T> merged_arr, tmp;

6        typename std::list< std::vector<T> >::iterator it = arrays_.begin();

7        n = 1;

8        merged_arr.push_back(key);

9        while (it != arrays_.end() && it->size() > 0)

10       {

11           n >>= 1;

12           tmp = std::move(merged_arr);

13           merged_arr.reserve(n);

14           std::merge(tmp.begin(), tmp.end(),

15               it->begin(), it->end(),

16               std::back_inserter(merged_arr));

17           it->clear();

18           ++it;

19       }

20       if (it != arrays_.end())
```

```
21          *it = std::move(merged_arr);
22      else
23          arrays_.emplace_back(std::move(merged_arr));
24  }
```

Suppose that the ith operation clears (or merges) t_i arrays. We have $0 \leq t_i \leq k - 1$ for all i. Note that merge two arrays with size m and n runs in $\Theta(m + n)$. The sum of the sizes of arrays $A_0, A_1, \cdots, A_{j-1}$ is

$$total\_size(0, j - 1) = \sum_{h=0}^{j-1} 2^h = 2^j - 1$$

In ith operation, we merge arrays $A_0, A_1, \cdots, A_{t_i-1}$ and insert the new element, and the running time is

$$\Theta(1 + \sum_{j=0}^{t_i-1} (2^j + total\_size(0, j - 1))) = \Theta(1 + \sum_{j=0}^{t_i-1} (2^j + 2^j - 1))$$

$$= \Theta(1 - t_i + 2 \cdot \sum_{j=0}^{t_i-1} 2^j)$$

$$= \Theta(1 - t_i + 2 \cdot (2^{t_i} - 1))$$

$$= \Theta(2^{t_i+1} - t_i - 1)$$

$$= \Theta(2^{t_i})$$

Clearly, the running time of the worst case is

$$\Theta(2^{k-1}) = \Theta(2^{\lceil \lg(n+1) \rceil - 1}) = \Theta(n)$$

when $t_i = k - 1$.

We analysis amortized cost by using aggregate analysis. Similar to the analysis of INCREMENT, array $A[j]$ clears (merge with another 2^j elements and insert them into $A[j+1]$) $\lfloor n/2^j \rfloor$ times in a sequence of n INSERT operations on an initially empty container (array of arrays), and the running time of each time in $\Theta(2 \cdot 2^j) = \Theta(2^j)$. Hence the total cost of a sequence of n INSERT operations is

$$\Theta(\sum_{j=0}^{k-1} \lfloor \frac{n}{2^j} \rfloor 2^j) = \Theta(nk) = \Theta(n \lg n)$$

Then the amortized cost of each INSERT is $\Theta(\lg n)$.

(c)

```
1  template <typename T>
2  void DynamicBinarySearch<T>::Delete(const Iterators& its)
3  {
4      size_t n;
5      // find the full array with the smallest size
```

```
6          // (i.e. the full array with smallest index in arrays_)
7          typename std::list< std::vector<T> >::iterator
8              first_full_it = arrays_.begin();
9          while (first_full_it->size() == 0)
10         {
11             ++first_full_it;
12         }
13         // delete the element and refill the array with first_full_it->back()
14         typename std::vector<T>::iterator processing_element_it = its.second,
15             target_element_it = std::lower_bound
16             (its.first->begin(), its.first->end(), first_full_it->back());
17         if (target_element_it > processing_element_it)
18         {
19             ++processing_element_it;
20             while (target_element_it != processing_element_it)
21             {
22                 *(processing_element_it - 1) = std::move(*processing_element_it);
23                 ++processing_element_it;
24             }
25             *(processing_element_it - 1) = std::move(first_full_it->back());
26         }
27         else
28         {
29             while (target_element_it != processing_element_it)
30             {
31                 *processing_element_it = std::move(*(processing_element_it - 1));
32                 --processing_element_it;
33             }
34             *processing_element_it = std::move(first_full_it->back());
35         }
36         // split first_full_it and assign them into arrays with smaller index in arrays_
37         n = 1;
38         typename std::vector<T>::iterator element_it = first_full_it->begin();
39         for (typename std::list< std::vector<T> >::iterator it = arrays_.begin();
40             it != first_full_it; ++it)
41         {
42             it->assign(std::move_iterator(element_it),
43                 std::move_iterator(element_it) + n);
44             element_it = element_it + n;
45             n <<= 1;
```

```
46          }
47          first_full_it->clear();
48          // remove empty arrays (optional)
49          if (arrays_.back().empty())
50          {
51              arrays_.pop_back();
52          }
53      }
```

In the worst case, delete operation takes $\Theta(n)$ time.

Updating...