# 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

**Proof.** 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 $\langle \text{INCREMENT}, \text{DECREMENT}, \text{INCREMENT}, \text{DECREMENT}, \cdots \rangle$ cost $\Theta(nk)$ in total. $\square$

### 17.1-3

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

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

## 17.2

### 17.2-1

**Proof.**

| 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.

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

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

```

void Increment(Counter& counter)
{
    int i;
    i = 0;
    while (i < counter.length && counter.bits[i] == 1)
    {
        counter.bits[i] = 0;
        ++i;
    }
    if (i < counter.length)
    {
        counter.bits[i] = 1;
        counter.high_order_one = std::max(i, counter.high_order_one);
    }
    else
    {
        // overflow
        counter.high_order_one = -1;
    }
}

void Reset(Counter& counter)
{
    int i;
    for (i = 0; i < counter.length; ++i)
    {
        counter.bits[i] = 0;
    }
    counter.high_order_one = -1;
}
```

# 17.3

## 17.3-1

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

$$\begin{aligned}\hat{c_i} &= c_i + \Phi'(D_i) - \Phi(D_{i-1}) \\ &= c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_{i-1}) - \Phi(D_0)) \\ &= c_i + \Phi(D_i) - \Phi(D_{i-1})\end{aligned}$$

□

## 17.3-2

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

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

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.

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= 2 - 2(i - \frac{i}{2}) \\ &= 2 - i\end{aligned}$$

$$\begin{aligned}\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= i + 2 - i \\ &= 2\end{aligned}$$

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

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

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

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 $\Phi$ be

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

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 $i$th operation. Then $n_i - 1 = n_{i-1}$.

If $n_i = 1$,

$$\begin{aligned} \hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i \end{aligned}$$

If $n_i \geq 2$,

$$\begin{aligned} \hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq k \lg n_i + k n_i \lg n_i - k n_{i-1} \lg n_{i-1} \\ &= k(\lg n_i + n_i \lg n_i - n_{i-1} \lg n_{i-1}) \\ &= k(\lg n_i + n_i \lg n_i - (n_i - 1) \lg(n_i - 1)) \\ &= k(\lg n_i + n_i \lg n_i - n_i \lg(n_i - 1) + \lg(n_i - 1)) \\ &< k(2 \lg n_i + n_i(\lg n_i - \lg(n_i - 1))) \end{aligned}$$

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

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

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 $i$th operation. Then $n_{i-1} - 1 = n_i$.
If $n_{i-1} = 1$,

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

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

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

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

## 17.3-4

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

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

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

## 17.3-5

**Proof.**

$$\Phi(D_0) = b$$

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

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

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

```
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();
            }
        }
        T top = std::move(s_b_.top());
        s_b_.pop();
        return std::move(top);
    }
```

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

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

Let

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

**Case 1.** Enqueue is performed in the $i$th operation.

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

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

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

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

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

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.

```
class DataStructure
{
public:
    void Insert(int x);
    void DeleteLargerHalf();
    const std::vector<int>& Get() const;
private:
    std::vector<int> arr_;
};

void DataStructure::Insert(int x)
{
    arr_.push_back(x);
}

void DataStructure::DeleteLargerHalf()
{
    size_t median = arr_.size() >> 1;
    LinearSelect(arr_, 0, arr_.size() - 1, (arr_.size() - 1) >> 1);
    arr_.erase(arr_.begin() + median, arr_.end());
}

const std::vector<int>& DataStructure::Get() const
{
    return arr_;
}
```

Assume

$$c_i = \begin{cases} 1 & \text{if } \textsc{Insert} \text{ is performed in the } i\text{th operation,} \\ n_{i-1} & \text{if } \textsc{Delete-Larger-Half} \text{ is performed in the } i\text{th operation} \end{cases}$$

where $n_i$ is $|S|$ after the $i$th operation. Let

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

be the potential function of the data structure.

**Case 1.** INSERT is performed in the $i$th 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 $i$th operation.

$$\begin{aligned}\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + 2(n_i - n_{i-1}) \\ &= n_{i-1} + 2(\frac{n_{i-1}}{2} - n_{i-1}) \\ &= n_{i-1} - n_{i-1} \\ &= 0\end{aligned}$$

# 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 $\alpha$ 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 $\beta$ 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 $num \geq 1$. $num \leq \beta \cdot size \Longrightarrow num + 1 \leq 2\beta \cdot size$

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

$$\begin{aligned} num \leq \beta \cdot size &\Longleftrightarrow 2 \cdot num \leq 2\beta \cdot size \\ &\Longleftrightarrow num + 1 \leq 2 \cdot num \leq 2\beta \cdot size \end{aligned}$$

□

**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 $num_k + 1 \leq 2\beta \cdot size_k$.

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

□

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 $num_k + 1 - \beta \cdot size_k > 0$ □

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 $i$th 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 $i$th 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

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

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

**Proof.**

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

**Proof.**

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

□

Suppose that $\alpha_{i-1} \geq \frac{1}{2}$ and the $i$th 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 $i$th 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.** $\alpha_i < \frac{1}{2}$.

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (\frac{size_i}{2} - num_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (\frac{size_{i-1}}{2} - (num_{i-1} - 1)) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 2 + \frac{3}{2} \cdot size_{i-1} - 3 \cdot num_{i-1} \\
&= 2 + \frac{3}{2} \cdot size_{i-1} - 3\alpha_{i-1} \cdot size_{i-1} \\
&\leq 2 + \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} \\
&= 2
\end{aligned}
$$

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

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

□

## 17.4-3

**Proof.** Suppose that TABLE-DELETE is performed in the $i$th 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 $i$th operation).
Then we have $c_i = 1$ and $size_i = size_{i-1}$.

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= c_i + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= 1 + |2 \cdot (num_{i-1} - 1) - size_{i-1}| - |2 \cdot num_{i-1} - size_{i-1}| \\
&\overset{\triangle}{\leq} 1 + (|2 \cdot num_{i-1} - size_{i-1}| + |-2|) - |2 \cdot num_{i-1} - size_{i-1}| \\
&= 3
\end{aligned}
$$

**Case 2.** $\frac{num_{i-1}-1}{size_{i-1}} < \frac{1}{3}$ (a contraction is triggered in the $i$th operation).
Then we have $c_i = num_i + 1 = num_{i-1}$ and $size_i = \frac{2}{3} \cdot size_{i-1}$.

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

**Lemma 5.**

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

**Proof.**

$$\begin{aligned}
\frac{num_{i-1} - 1}{size_{i-1}} < \frac{1}{3} &\Longrightarrow 3 \cdot (num_{i-1} - 1) < size_{i-1} \\
&\Longrightarrow 2 \cdot (num_{i-1} - 1) < \frac{2}{3} \cdot size_{i-1} \\
&\Longrightarrow 2 \cdot num_{i-1} - \frac{2}{3} \cdot size_{i-1} - 2 < 0
\end{aligned}$$

□

**Lemma 6.**

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

**Proof.**

$$\begin{aligned}
\frac{num_{i-1} - 1}{size_{i-1}} < \frac{1}{3} &\Longrightarrow 3 \cdot (num_{i-1} - 1) < size_{i-1} \\
&\Longrightarrow 2 \cdot num_{i-1} - 3 + num_{i-1} < size_{i-1} \\
&\Longrightarrow 2 \cdot num_{i-1} - size_{i-1} + (num_{i-1} - 3) < 0 \\
&\Longrightarrow 2 \cdot num_{i-1} - size_{i-1} < 3 - num_{i-1}
\end{aligned}$$

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

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

$$\begin{aligned}
\hat{c}_i &= num_{i-1} + |2 \cdot num_{i-1} - \frac{2}{3} \cdot size_{i-1} - 2| - |2 \cdot num_{i-1} - size_{i-1}| \\
&= num_{i-1} - (2 \cdot num_{i-1} - \frac{2}{3} \cdot size_{i-1} - 2) \overset{(--)}{+} (2 \cdot num_{i-1} - size_{i-1}) \\
&= (num_{i-1} - \frac{1}{3} \cdot size_{i-1} - 1) + 3
\end{aligned}$$

**Lemma 7.**

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

**Proof.**

$$\begin{aligned}
\frac{num_{i-1} - 1}{size_{i-1}} < \frac{1}{3} &\Longrightarrow num_{i-1} - 1 < \frac{1}{3} \cdot size_{i-1} \\
&\Longrightarrow num_{i-1} - \frac{1}{3} \cdot size_{i-1} - 1 < 0
\end{aligned}$$

□

Hence

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

□

# Chapter 17 Problems

## 17-1

### (a)

```
template <typename T>
void BitReversal(std::vector<T>& arr)
{
    size_t k, tmp, i, n, rev;
    n = arr.size();
    std::vector<bool> counter(n, false);
    tmp = n;
    k = -1;
    while (tmp)
    {
        tmp = tmp >> 1;
        ++k;
    }
    for (i = 0; i < n; ++i)
    {
        if (counter[i] == false)
        {
            rev = Rev(k, i);
            std::swap(arr[i], arr[rev]);
            counter[i] = true;
            counter[rev] = true;
        }
    }
}
```

### (b)

Note that

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

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 $rev_k(rev_k(a) + 1) + 1$.

```
size_t BitReversedIncrement(size_t k, size_t a)
{
    size_t i;
```

```
    i = 1 << (k - 1);
    while (i > 0 && (a & i) != 0)
    {
        a = a & ( ~ i );
        i = i >> 1;
    }
    a = a | i;
    return a;
}
```

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

```
template <typename T>
class DynamicBinarySearch
{
public:
    using Iterators = std::pair
        <typename std::list< std::vector<T> >::iterator,
        typename std::vector<T>::iterator>;
    Iterators Search(const T& key);
    void Insert(const T& key);
    void Delete(const Iterators& its);
private:
    std::list< std::vector<T> > arrays_;
};
```

### (a)

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

```
    size_t n, i;
    n = arrays_.size();
    for (typename std::list< std::vector<T> >::iterator it = arrays_.begin();
        it != arrays_.end(); ++it)
    {
        typename std::vector<T>::iterator sub_it =
            std::lower_bound(it->begin(), it->end(), key);
        if (sub_it != it->end() && *sub_it == key)
            return std::make_pair(it, sub_it);
    }
    throw std::out_of_range("the container does not have an element "
                            "with the specified key");
}
```

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)**

```
template <typename T>
void DynamicBinarySearch<T>::Insert(const T& key)
{
    size_t n;
    std::vector<T> merged_arr, tmp;
    typename std::list< std::vector<T> >::iterator it = arrays_.begin();
    n = 1;
    merged_arr.push_back(key);
    while (it != arrays_.end() && it->size() > 0)
    {
        n >>= 1;
        tmp = std::move(merged_arr);
        merged_arr.reserve(n);
        std::merge(tmp.begin(), tmp.end(),
            it->begin(), it->end(),
            std::back_inserter(merged_arr));
        it->clear();
        ++it;
    }
    if (it != arrays_.end())
```

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

Suppose that the $i$th operation clears (or merges) $t_i$ arrays. We have $0 \le t_i \le k-1$ 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 $i$th operation, we merge arrays $A_0, A_1, \cdots, A_{t_i-1}$ and insert the new element, and the running time is

$$\begin{aligned}
\Theta(1 + \sum_{j=0}^{t_i-1}(2^j + total_size(0, j-1))) &= \Theta(1 + \sum_{j=0}^{t_i-1}(2^j + 2^j - 1)) \\
&= \Theta(1 - t_i + 2 \cdot \sum_{j=0}^{t_i-1} 2^j) \\
&= \Theta(1 - t_i + 2 \cdot (2^{t_i} - 1)) \\
&= \Theta(2^{t_i+1} - t_i - 1) \\
&= \Theta(2^{t_i})
\end{aligned}$$

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)**

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

```
    // (i.e. the full array with smallest index in arrays_)
    typename std::list< std::vector<T> >::iterator
        first_full_it = arrays_.begin();
    while (first_full_it->size() == 0)
    {
        ++first_full_it;
    }
    // delete the element and refill the array with first_full_it->back()
    typename std::vector<T>::iterator processing_element_it = its.second,
        target_element_it = std::lower_bound
        (its.first->begin(), its.first->end(), first_full_it->back());
    if (target_element_it > processing_element_it)
    {
        ++processing_element_it;
        while (target_element_it != processing_element_it)
        {
            *(processing_element_it - 1) = std::move(*processing_element_it);
            ++processing_element_it;
        }
        *(processing_element_it - 1) = std::move(first_full_it->back());
    }
    else
    {
        while (target_element_it != processing_element_it)
        {
            *processing_element_it = std::move(*(processing_element_it - 1));
            --processing_element_it;
        }
        *processing_element_it = std::move(first_full_it->back());
    }
    // split first_full_it and assign them into arrays with smaller index in arrays_
    n = 1;
    typename std::vector<T>::iterator element_it = first_full_it->begin();
    for (typename std::list< std::vector<T> >::iterator it = arrays_.begin();
        it != first_full_it; ++it)
    {
        it->assign(std::move_iterator(element_it),
            std::move_iterator(element_it) + n);
        element_it = element_it + n;
        n <<= 1;
```

```
    }
    first_full_it->clear();
    // remove empty arrays (optional)
    if (arrays_.back().empty())
    {
        arrays_.pop_back();
    }
}
```

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

## 17-3

### (a)

```
void RebuildTreeTranverse(Node* root, std::vector<Node*>& elements)
{
    if (root != nullptr)
    {
        RebuildTreeTranverse(root->left, elements);
        elements.push_back(root);
        RebuildTreeTranverse(root->right, elements);
    }
}

Node* RebuildTreeBuild(std::vector<Node*>& elements, int lower, int upper)
{
    int middle;
    if (lower > upper)
        return nullptr;
    middle = lower + ((upper - lower) >> 1);
    elements[middle]->size = upper - lower + 1;
    elements[middle]->left = RebuildTreeBuild(elements, lower, middle - 1);
    elements[middle]->right = RebuildTreeBuild(elements, middle + 1, upper);
    return elements[middle];
}

Node* RebuildTree(Node* root)
{
    std::vector<Node*> elements;
    elements.reserve(root->size);
```

```
    RebuildTreeTranverse(root, elements);
    return RebuildTreeBuild(elements, 0, root->size - 1);
}
```

**(b)**

**Proof.** Let $T(n)$ be the running time of a search on a $\alpha$-balanced binary search tree with $n$ elements. We have

$$T(n) \leq T(\alpha \cdot n) + \Theta(1) = T(\frac{n}{1/\alpha}) + \Theta(1)$$

Since $\alpha < 1$, we have $1/\alpha > 1$. By the master theorem, we have

$$T(\frac{n}{1/\alpha}) + \Theta(1) = \Theta(\lg n)$$

Hence we have

$$T(n) = O(\lg n)$$

□

**(c)**

**Proof.** Since $c$ is a sufficiently large constant, we may assume $c > 0$. $\Delta(x)$ is an absolute value. Obviously, $\Phi(T)$ is always nonnegative.

In order to show that a 1/2-balanced tree $T$ has potential 0, we need to show that

$$\Delta(x) = |x.left.size - x.right.size| \leq 1$$

for all $x \in T$.

Let $x \in T$. WLOG, assume $x.left.size \geq x.right.size$, so we want to show that

$$x.left.size - x.right.size \leq 1$$

Note that

$$x.size = x.left.size + x.right.size + 1$$

Since $T$ is a 1/2-balanced tree, we have

$$x.left.size \leq \frac{1}{2} \cdot x.size$$

Then

$$\begin{aligned} x.left.size - x.right.size &= x.left.size - (x.size - x.left.size - 1) \\ &= 2 \cdot x.left.size - x.size + 1 \\ &\leq 2 \cdot \frac{1}{2} \cdot x.size - x.size + 1 \\ &= 1 \end{aligned}$$

□

### (d)

Note that we rebuild the subtree to be 1/2-balanced, instead to $\alpha$-balanced.

Denote $\Phi_j$ as the potential of $T$ after the $j$th operation. Denote $size_j(w)$ as the $w.size$ after the $j$th operation for all $w \in T$. Denote $\Delta_j(w)$ as the $\Delta(w)$ after the $j$th operation for all $w \in T$. Suppose that $T$ is not $\alpha$-balanced after the $(i-1)$th operation and is 1/2-balanced after the $i$th operation. By part (c), we have $\Phi_i = 0$. Let $x$ be the root node of $T$ after the $(i-1)$th operation. We have

$$\Phi_{i-1} \geq c \cdot \Delta_{i-1}(x)$$

WLOG, assume $size_{i-1}(x.left) \geq size_{i-1}(x.right)$. Since $T$ is not $\alpha$-balanced after the $(i-1)$th operation,

$$size_{i-1}(x.left) > \alpha \cdot size_{i-1}(x)$$

must be true. Then

$$\begin{aligned}
\Delta_{i-1}(x) &= size_{i-1}(x.left) - size_{i-1}(x.right) \\
&= size_{i-1}(x.left) - (size_{i-1}(x) - size_{i-1}(x.left) - 1) \\
&= 2 \cdot size_{i-1}(x.left) - size_{i-1}(x) + 1 \\
&> 2\alpha \cdot size_{i-1}(x) - size_{i-1}(x) + 1 \\
&= (2\alpha - 1) \cdot size_{i-1}(x) + 1 \\
&> (2\alpha - 1) \cdot size_{i-1}(x)
\end{aligned}$$

Suppose that $size_{i-1}(x) = m$. That is we are rebuilding an $m$-node subtree in the $i$th operation. Hence the amortized cost of the $i$th operation is

$$\begin{aligned}
m + \Phi_i - \Phi_{i-1} &= m - \Phi_{i-1} \\
&\leq m - c \cdot \Delta_{i-1}(x) \\
&< m - c \cdot (2\alpha - 1) \cdot m
\end{aligned}$$

In order to let the amortized time be $O(1)$, we let $c \geq \frac{1}{2\alpha - 1}$.

### (e)

**Proof.** Suppose that we perform insert or delete in the $i$th operation. Let $h$ be the height of $T$, By part (b), we derive the $h = O(\lg n)$. Then we pay $O(\lg n)$ for actual inserting or deleting the node. If we do not rebuild the tree, then $\Phi_i - \Phi_{i-1} \leq h + 1 = O(\lg n)$. If we rebuild the tree, then we rebuild the subtreee rooted at the highest non-$\alpha$-balanced node, and, by part (d), the amortized cost of rebuild is $O(1)$. □

## 17-4

### (a)

We notice that the only locations to have repeated coloring are fixup functions (RB-INSERT-FIXUP and RB-DELETE-FIXUP) and, by the analysis of insert and delete, we know the **while** loop of fixup functions repeat only if case 1 of RB-INSERT-FIXUP or case 2 of RB-DELETE-FIXUP occurs.

**insert**

In order to causes $\Omega(\lg n)$ color changes, we want the case 1 of RB-INSERT-FIXUP occurs $\Omega(\lg n)$ times. Consider the situation of the case 1 occurs every iteration. This situation occurs only if the color of parent of $z$ is RED and the color of uncle of $z$ is RED unless the parent of $z$ is the root node in every iteration; i.e. ($z == T.root$ or ($z.p.color == RED$ and $y.color == RED$)); equivalent to ($z == T.root$ or ($z.p.p.left.color == RED$ and $z.p.p.right.color == RED$)) in every iteration. For instance, consider a full, complete red-black tree where all nodes on level 0 are black, all nodes on level 1 are red, all nodes on level 2 are black, all nodes on level 3 are red, all nodes on level 4 are black, all nodes on level 5 are red...

**delete**

In order to causes $\Omega(\lg n)$ color changes, we want the case 2 of RB-DELETE-FIXUP occurs $\Omega(\lg n)$ times. Consider the situation of we are deleting a leaf node, and the case 2 occurs every iteration. For instance, consider a full, complete red-black tree where all nodes in the tree are black.

### (b)

By the analysis of insert and delete. the answer is case 2 and 3 for RB-INSERT and case 1, 3, and 4 for RB-DELETE.

### (c)

**Proof.** Suppose that the current iteration in RB-INSERT-FIXUP are applying case 1. Right before the case 1 is applied (right before line 5), we know $z.p.color == RED$ and $y.color == RED$, so $z.p.p.color == BLACK$. After the case 1 is applied (excluede the changing of identity of $z$; i.e. right asfter line 7), we blacken two red nodes and redden one black nodes. Hence the number of red nodes in $T$ decreases by 1. That is $\Phi(T') = \Phi(T) - 1$. □

### (d)

| type | location | potential change |
|---|---|---|
| node insertion | RB-INSERT: line 9 - 16 | $+1$ |
| color change | RB-INSERT-FIXUP: line 5 - 7 (case 1) | $-1$ |
| rotation | RB-INSERT-FIXUP: line 11 (case 2) | 0 |
| color change | RB-INSERT-FIXUP: line 12 - 13 (case 3) | 0 |
| rotation | RB-INSERT-FIXUP: line 14 (case 3) | 0 |

Note that we assume that 1 unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

We conclude that insertion costs 1 and causes potential increases by 1, applying case 1 costs 1 and causes potential decreases by 1, applying case 2 costs 1 and potential unchanged, and applying case 3 costs 1 and potential unchanged.

**(e)**

**Proof.** Each time case 1 (nonterminating) is applied we use the potential change of $-1$ to pay the cost of structural modifications in applying case 1. Case 2 and 3 (terminating) can be performed at most once in each call since they are terminating cases, so the structural modifications of applying cases 2 and 3 are at most 2. Hence the amortized number of structural modifications for each call is

$$\begin{aligned}&(\text{insertion cost}) + (\text{insertion potential change}) + m \cdot ((\text{case 1 cost}) + (\text{case 1 potential change})) +\\&(\text{case 2 cost}) + (\text{case 2 potential change}) + (\text{case 3 cost}) + (\text{case 3 potential change})\\\leq&(1) + (1) + m \cdot ((1) + (-1)) + (1) + (0) + (1) + (0) = 4\end{aligned}$$

□

**(f)**

**Proof.** Suppose that the current iteration in RB-INSERT-FIXUP are applying case 1. In this case, the color of $z.p$ will change from RED to BLACK at line 5, the color of $y$ will change from RED to BLACK at line 6, and the color of $z.p.p$ will change from BLACK to RED at line 7. We know $z.p$ has one red child $z$ since $z.p.color$ is RED before applying case 1 and $z$ is the processing node (so we allow $z$ to be red temporary), so $w(z.p)$ unchanged ($z.p$ is red $\rightarrow$ $z.p$ is black and has one red child). We know $y$ has no red children since $y.color$ is RED before applying case 1, so $w(y)$ increases by 1 ($y$ is red $\rightarrow$ $y$ is black and has no red child). We know $z.p.p$ has two red children ($z.p$ and $y$) before applying case 1 and become red after applying case 1, so $w(z.p.p)$ decreases by 2 ($z.p.p$ is black and has two red children $\rightarrow$ $z.p.p$ is red). We assume the color of $z.p.p.p$ is red here; otherwise, the loop will terminate after this iteration. Hence $\Phi(T') = \Phi(T) - 1$.

We use this potential change of $-1$ to pay the cost of structural modifications in applying case 1 each time, so the amortized number of structural modifications in nonterminating cases is 0. Since we only perform constant times structural modifications in the terminating cases, and potential changes are bounded by constant also, we conclude that the amortized number of structural modifications is $O(1)$. □

**(g)**

**Proof.** Suppose that the current iteration in RB-DELETE-FIXUP are applying case 2. Since case 2 are applied, we know that $x.color == BLACK$ by line 1, $w.left.color == BLACK$ and $w.left.color == BLACK$ by line 9, and $w.color == BLACK$ by line 4-5. In this case, the color of $w$ will change from BLACK to RED at line 10. Thus, we have $w(w)$ decreases by 1 ($w$ is black and has no red children $\rightarrow$ $w$ is red). If $x.p$ is black, then we have $w(x.p)$ decreases by 1 ($x.p$ is black and has no red children $\rightarrow$ $x.p$ is black and has one red child). If $x.p$ is red (actually, the loop will terminate after this iteration if $x.p$ is red), then we have $w(x.p)$ unchanged ($x.p$ is red $\rightarrow$ $x.p$ is red). Hence $\Phi(T') \leq \Phi(T) - 1$.

By the similar approach of part (f), we conclude that the amortized number of structural modifications is $O(1)$. □

**(h)**

**Proof.** By part (f) and (g), we know the **amorted** number of structural modifications performed by any call of RB-INSERT-FIXUP or RB-DELETE-FIXUP is $O(1)$. Also, the **amorted** number of structural modifications performed by lines 1 - 16 of RB-INSERT and lines 1 - 21 of RB-DELETE is $O(1)$. Therefore, in the worst case, any sequence of $m$ RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in total. □

## 17-5

**(a)**

**Proof.** In the worst case, $H$ find the key at the end of the list each time, and the cost to find such element is $n$ since the linked liest has $n$ elements. We know the size of access sequence is $m$, so the access cost of sequence $\sigma$ is $mn$ in the worst case. □

**(b)**

**Proof.** The cost to find element $x$ is $rank_L(x)$, and the cost to move $x$ to the first position on the list is $rank_L(x) - 1$ since we need to transpose adjacent list elements $rank_L(x) - 1$ times. Hence $c_i = (rank_L(x)) + (rank_L(x) - 1) = 2 \cdot rank_L(x) - 1$. □

**(c)**

**Proof.** Similarly, the cost to find element $x$ is $rank_{L^*_{i-1}}(x)$, and we know the cost to transpose $x$ is $t^*_i$ since $H$ performs $t^*_i$ tranpositions during access $\sigma_i$. Hence $c^*_i = (rank_{L^*_{i-1}}(x)) + (t^*_i) = rank_{L^*_{i-1}}(x) + t^*_i$. □

**(d)**

**Proof.** Note that we are tranposing adjacent list elements. Let $L_i = \langle a_1, a_2, \cdots, a_j, a_{j+1}, \cdots, a_n \rangle$ be a linked list. Suppose that we are tranposing element $a_j$ and $a_{j+1}$ for some integer $j \in [1, n-1]$. Let $L'_i = \langle a_1, a_2, \cdots, a_{j+1}, a_j, \cdots, a_n \rangle$, which is the result of the tranposition. Clearly, for all integer $h \in [1, j-1]$, $a_h$ precedes $a_j$ and $a_{j+1}$ in $L_i$ and $L'_i$, and for all integer $h \in [j+2, n]$, $a_j$ and $a_{j+1}$ precedes $a_h$ in $L_i$ and $L'_i$. Then for all integer $h \in [1, j-1] \cup [j+2, n]$, $(a_h, a_j)$ and $(a_h, a_{j+1})$ are inversions in $L'_i$ if and only if $(a_h, a_j)$ and $(a_h, a_{j+1})$ are inversions in $L_i$. Hence the only pair causes $q_i$, the number of inversions, changes is $(a_j, a_{j+1})$. If $(a_j, a_{j+1})$ is an inversion in $L_i$, then it is not an inversion in $L'_i$, which causes $q_i$ decreases by 1. If $(a_j, a_{j+1})$ is not an inversion in $L_i$, then it is an inversion in $L'_i$, which causes $q_i$ increases by 1. Since $\Phi(L_i) = 2q_i$, we conclude a tranposition either increases the potential by 2 or decreases the potential by 2. □

### (e)

**Proof.** Since $A \cap B = \emptyset$, we have $|A \cup B| = |A| + |B|$, which is the number of elements that precede $x$ in $L_{i-1}$, so $rank_{L_{i-1}}(x) = |A \cup B| + 1 = |A| + |B| + 1$.

Since $A \cap C = \emptyset$, we have $|A \cup C| = |A| + |C|$, which is the number of elements that precede $x$ in $L^*_{i-1}$, so $rank_{L^*_{i-1}}(x) = |A \cup C| + 1 = |A| + |C| + 1$. □

### (f)

**Lemma 8.** Access $\sigma_i$ only causes the changes of the number of inversions that include $x$ (i.e. access $\sigma_i$ does not cause the changes of the number of inversions that do not include $x$).

**Proof.** We claim that access $\sigma_i$ only causes the number of inversions that include $x$ changes Let $a, b \in L$ where $a \neq x$ and $b \neq x$. Suppose that $(a, b)$ is an inversion in list $L_{i-1}$ and is not an inversion in list $L_i$, for the purpose of contraction. WLOG, assume $a$ precedes $b$ in $L_{i-1}$ and $b$ precedes $a$ in $L_{i-1}*$. Obviously, tranposing $x$ will never change the order between $a$ and $b$ by both move-to-front and heuristic $H$. Hence $(a, b)$ must be an inversion in list $L_i$ also. Contraction. Suppose that $(a, b)$ is not an inversion in list $L_{i-1}$ and is an inversion in list $L_i$, and we can have the contraction by the similar approach. □

**Claim 9.** See part (f).

**Proof.** In order to show that $\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t^*_i)$, we need to show that access $\sigma_i$ causes the number of inversions increases at most $|A| - |B| + t^*_i$ (i.e. $q_i - q_{i-1} \leq |A| - |B| + t^*_i$). By the lemma, we only care about the changes of the number of inversions that include $x$. Clearly, the number of inversions include $x$ in $L_{i-1}$ is $|B| + |C|$. In order to find the upper bound of $q_i - q_{i-1}$, we want to maximize the number of inversions that include $x$ in $L_i$. With the move-to-front heuristic, we move $x$ to the first position on $L_i$, so we want to move backward $x$ with the heuristic $H$. Since henuristic $H$ performs $t^*_i$ transpositions durning access $\Phi_i$, we know there are $|A| + |C| + t^*_i$ elements precede $x$ in $L^*_i$ by our scheme, which implies there are $|A| + |C| + t^*_i$ inversions that include $x$ in $L_i$. Hence $q_i - q_{i-1} \leq (|A| + |C| + t^*_i) - (|B| + |C|) = |A| - |B| + t^*_i$. □

### (g)

**Proof.**

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\
&\leq (2 \cdot \text{rank}_{L_{i-1}}(x) - 1) + 2(|A| - |B| + t^*_i) && \text{(by parts (b) and (f))} \\
&= (2 \cdot (|A| + |B| + 1) - 1) + 2(|A| - |B| + t^*_i) && \text{(by part (e))} \\
&= 4 \cdot |A| + 2t^*_i + 1
\end{aligned}$$

$$\begin{aligned}
c^*_i &= \text{rank}_{L^*_{i-1}}(x) + t^*_i && \text{(by part (c))} \\
&= (|A| + |C| + 1) + t^*_i && \text{(by part (e))} \\
&= |A| + |C| + t^*_i + 1
\end{aligned}$$

$$
\begin{aligned}
\hat{c}_i &\leq 4 \cdot |A| + 2t_i^* + 1 \\
&< 4 \cdot |A| + 4 \cdot |C| + 4t_i^* + 4 \\
&= 4(|A| + |C| + t_i^* + 1) \\
&= 4c_i^*
\end{aligned}
$$

□

**(h)**

**Proof.** Since

$$C_{\mathrm{MTF}}(\sigma) = \sum_{i=1}^{m} c_i \leq \sum_{i=1}^{m} \hat{c}_i$$

and

$$C_H(\sigma) = \sum_{i=1}^{m} c_i^* \ ,$$

we have

$$
\begin{aligned}
C_{\mathrm{MTF}}(\sigma) &\leq \sum_{i=1}^{m} \hat{c}_i \\
&\leq \sum_{i=1}^{m} 4c_i^* \qquad \text{(by part (g))} \\
&= 4 \sum_{i=1}^{m} c_i^* \\
&= 4C_H(\sigma)
\end{aligned}
$$

□