# 17 Amortized Analysis

## 17.1 Aggregate analysis

### 17.1-1

> If the set of stack operations included a $\text{MULTIPUSH}$ operation, which pushses $k$ items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

No. The time complexity of such a series of operations depends on the number of pushes (pops vice versa) could be made. Since one $\text{MULTIPUSH}$ needs $\Theta(k)$ time, performing $n$ $\text{MULTIPUSH}$ operations, each with $k$ elements, would take $\Theta(kn)$ time, leading to amortized cost of $\Theta(k)$.

### 17.1-2

> Show that if a $\text{DECREMENT}$ operation were included in the $k$-bit counter example, $n$ operations could cost as much as $\Theta(nk)$ time.

The logarithmic bit flipping predicate does not hold, and indeed a sequence of events could consist of the incrementation of all $1$s and decrementation of all $0$s; yielding $\Theta(nk)$.

### 17.1-3

> 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 aggregate analysis to determine the amortized cost per operation.

Let $n$ be arbitrary, and have the cost of operation $i$ be $c(i)$. Then we have,

$$
\begin{aligned}
\sum_{i=1}^n c(i) & = \sum_{i=1}^{\lceil \lg n \rceil} 2^i + \sum_{i \le n \text{ is not a power of } 2} 1 \\
& \le \sum_{i=1}^{\lceil \lg n \rceil} 2^i + n \\
& = 2^{1 + \lceil \lg n \rceil} - 1 + n \\
& \le 2n - 1 + n \\
& \le 3n \in O(n).
\end{aligned}
$$

To find the average, we divide by $n$, and the amortized cost per operation is $O(1)$.

## 17.2 The accounting method

## 17.2-1

> Suppose we perform a sequence of stack operations on a stack whose size never exceeds $k$. After every $k$ operations, we make a copy of the entire stack for backup purposes. Show that the cost of $n$ stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

For every stack operation, we charge twice.

- First, we charge the actual cost of the stack operation.
- Second, we charge the cost of copying an element later on.

Since we have the size of the stack never exceed $k$, and there are always $k$ operations between backups, we always overpay by at least enough. Therefore, the amortized cost of the operation is constant, and the cost of the $n$ operation is $O(n)$.

## 17.2-2

> Redo Exercise 17.1-3 using an accounting method of analysis.

Let $c_i$ = cost of $i$th operation.

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

Charge $3$ (amortized cost $\hat{c_i}$) for each operation.

- If $i$ is not an exact power of $2$, pay

$$1\$, and store$$

  2$ as credit.
- If $i$ is an exact power of $2$, pay $$i$, using stored credit.

| Operation | Cost | Actual cost | Credit remaining |
|---|---|---|---|
| 1 | 3 | 1 | 2 |
| 2 | 3 | 2 | 3 |
| 3 | 3 | 1 | 5 |
| 4 | 3 | 4 | 4 |
| 5 | 3 | 1 | 6 |
| 6 | 3 | 1 | 8 |
| 7 | 3 | 1 | 10 |
| 8 | 3 | 8 | 5 |
| 9 | 3 | 1 | 7 |
| 10 | 3 | 1 | 9 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

Since the amortized cost is $$3$ per operation, $\sum\limits_{i = 1}^n \hat c_i = 3n$.

We know from Exercise 17.1-3 that $\sum\limits_{i=1}^n \hat{c_i} < 3n$.

Then we have

$$\sum_{i=1}^n \hat{c_i} \ge \sum_{i=1}^n c_i \Rightarrow \text{credit} = \text{amortized cose} - \text{actual cost} \ge 0.$$

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of $n$ operations is $O(n)$.

## 17.2-3

> Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it $0$). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of $n$ $\text{INCREMENT}$ and $\text{RESET}$ operations takes time $O(n)$ on an initially zero counter. ($\textit{Hint:}$ Keep a pointer to the high-order $1$.)

(Removed)

# 17.3 The potential method

## 17.3-1

> Suppose we have a potential function $\Phi$ such that $\Phi(D_i) \ge \Phi(D_0)$ for all $i$, but $\Phi(D_0) \ne 0$. Show that there exists a potential fuction $\Phi'$ such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \ge 0$ for all $i \ge 1$, and the amortized costs using $\Phi'$ are the same as the amortized costs using $\Phi$.

Define the potential function $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$ for all $i \ge 1$.

Then

$$\Phi'(D_0) = \Phi(D_0) - \Phi(D_0) = 0,$$

and

$$\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \ge 0.$$

The amortized cost is

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

## 17.3-2

> Redo Exercise 17.1-3 using a potential method of analysis.

Define the potential function $\Phi(D_0) = 0$, and $\Phi(D_i) = 2i - 2^{1+\lfloor \lg i \rfloor}$ for $i > 0$. For operation 1,

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2i - 2^{1+\lfloor \lg i \rfloor} - 0 = 1.$$

For operation $i(i > 1)$, if $i$ is not a power of $2$, then

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2i - 2^{1+\lfloor \lg 1 \rfloor} - (2(i-1) - 2^{1+\lfloor \lg(i-1) \rfloor}) = 3.$$

If $i = 2^j$ for some $j \in \mathbb{N}$, then

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = i + 2i - 2^{1+j} - (2(i-1) - 2^{1+j-1}) = i + 2i - 2i - 2i + 2 + i = 2.$$

Thus, the amortized cost is $3$ per operation.

## 17.3-3

> Consider an ordinary binary min-heap data structure with $n$ elements supporting the instructions $\text{INSERT}$ and $\text{EXTRACT-MIN}$ in $O(\lg n)$ worst-case time. Give a potential function $\Phi$ such that the amortized cost of $\text{INSERT}$ is $O(\lg n)$ and the amortized cost of $\text{EXTRACT-MIN}$ is $O(1)$, and show that it works.

Make the potential function be equal to $\sum_{i=1}^n \lg i$ where $n$ is the size of the min-heap. Then, there is still a cost of $O(\lg n)$ to $\text{INSERT}$, since only an amount of amortization that is $\lg n$ was spent to increase the size of the heap by $1$.

However, the amortized cost of $\text{EXTRACT-MIN}$ is $0$, as all its actual cost is compensated.

## 17.3-4

> What is the total cost of executing $n$ of the stack operations $\text{PUSH}$, $\text{POP}$, and $\text{MULTIPOP}$, assuming that the stack begins with $s_0$ objects and finishes with $s_n$ objects?

Let $\Phi$ be the potential function that returns the number of elements in the stack. We know that for this potential function, we have amortized cost $2$ for $\text{PUSH}$ operation and amortized cost $0$ for $\text{POP}$ and $\text{MULTIPOP}$ operations.

The total amortized cost is

$$\sum_{i=1}^n \hat{c_i} = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0).$$

Using the potential function and the known amortized costs, we can rewrite the equation as

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

which gives us the total cost of $O(n + (s_0 - s_n))$. If $s_n \ge s_0$, then this equals to $O(n)$, that is, if the stack grows, then the work done is limited by the number of operations.

(Note that it does not matter here that the potential may go below the starting potential. The condition $\Phi(D_n) \ge \Phi(D_0)$ for all $n$ is only required to have $\sum_{i=1}^{n} \hat{c_i} \ge \sum_{i=1}^{n} c_i$, but we do not need for that to hold in this application.)

## 17.3-5

> Suppose that a counter begins at a number with $b$ $1$s in its binary representation, rather than at $0$. Show that the cost of performing $n$ $\text{INCREMENT}$ operations is $O(n)$ if $n = \Omega(b)$. (Do not assume that $b$ is constant.)

$$\begin{aligned} \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c_i} - \Phi(D_n) + \Phi(D_0) \\ &= n - x + b \\ &\le n - x + n \\ &= O(n). \end{aligned}$$

## 17.3-6

> Show how to implement a queue with two ordinary stacks (Exercise 10.1-6) so that the amortized cost of each $\text{ENQUEUE}$ and each $\text{DEQUEUE}$ operation is $O(1)$.

We'll use the accounting method for the analysis. Assign cost $3$ to the $\text{ENQUEUE}$ operation and $0$ to the $\text{DEQUEUE}$ operation. Recall the implementation of 10.1-6 where we enqueue by pushing on to the top of stack 1, and dequeue by popping from stack 2.

If stack 2 is empty, then we must pop every element from stack 1 and push it onto stack 2 before popping the top element from stack 2. For each item that we enqueue we accumulate 2 credits. Before we can dequeue an element, it must be moved to stack 2. Note: this might happen prior to the time at which we wish to dequeue it, but it will happen only once overall. One of the 2 credits will be used for this move. Once an item is on stack 2 its pop only costs $1$ credit, which is exactly the remaining credit associated to the element. Since each operation's cost is $O(1)$, the amortized cost per operation is $O(1)$.

## 17.3-7

> Design a data structure to support the following two operations for a dynamic multiset $S$ of integers, which allows duplicate values:

> $\text{INSERT}(S, x)$ inserts $x$ into $S$.
>
> $\text{DELETE-LARGER-HALF}(S)$ deletes the largest $\lceil |S|/2 \rceil$ elements from $S$.
>
> Explain how to implement this data structure so that any sequence of $m$ $\text{INSERT}$ and $\text{DELETE-LARGER-HALF}$ operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of $S$ in $O(|S|)$ time.

We'll store all our elements in an array, and if ever it is too large, we will copy all the elements out into an array of twice the length.

To delete the larger half, we first find the element $m$ with order statistic $\lceil |S|/2 \rceil$ by the algorithm presented in section 9.3. Then, scan through the array and copy out the elements that are smaller or equal to $m$ into an array of half the size.

Since the delete half operation takes time $O(|S|)$ and reduces the number of elements by $\lfloor |S|/2 \rfloor \in \Omega(|S|)$, we can make these operations take ammortized constant time by selecting our potential function to be linear in $|S|$.

Since the insert operation only increases $|S|$ by one, we have that there is only a constant amount of work going towards satisfying the potential, so the total ammortized cost of an insertion is still constant. To output all the elements just iterate through the array and output each.

## 17.4 Dynamic tables

### 17.4-1

> Suppose that we wish to implement a dynamic, open-address hash table. Why might we consider the table to be full when its load factor reaches some value $\alpha$ that is strictly less than $1$? Describe briefly how to make insertion into a dynamic, open-address hash table run in such a way that the expected value of the amortized cost per insertion is $O(1)$. Why is the expected value of the actual cost per insertion not necessarily $O(1)$ for all insertions?

By theorems 11.6-11.8, the expected cost of performing insertions and searches in an open address hash table approaches infinity as the load factor approaches one, for any load factor fixed away from $1$, the expected time is bounded by a constant though. The expected value of the actual cost my not be $O(1)$ for every insertion because the actual cost may include copying out the current values from the current table into a larger table because it became too full. This would take time that is linear in the number of elements stored.

### 17.4-2

> Show that if $\alpha_{i-1} \ge 1/2$ and the $i$th operation on a dynamic table is $\text{TABLE-DELETE}$, then the amortized cost of the operation with respect to the potential function $(17.6)$ is bounded above by a constant.

If $\alpha_{i-1} \ge 1/2$, $\text{TABLE-DELETE}$ cannot ***contract***, so $c_i = 1$ and $size_i = size_{i-1}$.

- **Case 1:** if $\alpha_i \ge 1/2$,

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

- **Case 2:** if $\alpha_i < 1/2$,

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

## 17.4-3

> Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, we contract it by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function
>
> $$\Phi(T) = |2 \cdot T.num - T.size|,$$
>
> show that the amortized cost of a $\text{TABLE-DELETE}$ that uses this strategy is bounded above by a constant.

If $1/3 < \alpha_i \le 1/2$,

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i - 2 \cdot num_i) - (size_i - 2 \cdot (num_i + 1)) \\
&= 3.
\end{aligned}$$

If the $i$th operation does trigger a contraction,

$$\begin{aligned}
\frac{1}{3} size_{i-1} &= num_i + 1 \\
size_{i-1} &= 3(num_i + 1) \\
size_i &= \frac{2}{3} size_{i-1} = 2(num_i + 1).
\end{aligned}$$

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + [2 \cdot (num_i + 1) - 2 \cdot num_i] - [3 \cdot (num_i + 1) - 2 \cdot (num_i + 1)] \\
&= 2.
\end{aligned}$$

# Problem 17-1 Bit-reversed binary counter

Chapter 30 examines an important algorithm called the fast Fourier transform, or $\mathrm{FFT}$. The first step of the $\mathrm{FFT}$ algorithm performs a ***bit-reversal permutation*** on an input array $A[0..n-1]$ whose length is $n = 2^k$ for some nonnegative integer $k$. This permutation swaps elements whose indices have binary representations that are the reverse of each other.

We can express each index $a$ as a $k$-bit sequence $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$, where $a = \sum_{i=0}^{k-1} a_i 2^i$. We define

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

thus,

$$\mathrm{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.$$

For example, if $n = 16$ (or, equivalently, $k = 4$), then $\mathrm{rev}_k(3) = 12$, since the $4$-bit representation of $3$ is $0011$, which when reversed gives $1100$, the $4$-bit representation of $12$.

**a.** Given a function $\mathrm{rev}_k$ that runs in $\Theta(k)$ time, write an algorithm to perform the bit-reversal permutation on an array of length $n = 2^k$ in $O(nk)$ time.

We can use an algorithm based on an amortized analysis to improve the running time of the bit-reversal permutation. We maintain a "bit-reversed counter" and a procedure $\mathrm{BIT\text{-}REVERSED\text{-}INCREMENT}$ that, when given a bit-reversed-counter value $a$, produces $\mathrm{rev}_k(\mathrm{rev}_k(a) + 1)$. If $k = 4$, for example, and the bit-reversed counter starts at $0$, then successive calls to $\mathrm{BIT\text{-}REVERSED\text{-}INCREMENT}$ produce the sequence

$$0000, 1000, 0100, 1100, 0010, 1010, \ldots = 0, 8, 4, 12, 2, 10, \ldots.$$

**b.** Assume that the words in your computer store $k$-bit values and that in unit time, your computer can manipulate the binary values with operations such as shifting left or right by arbitrary amounts, bitwise-$\mathrm{AND}$, bitwise-$\mathrm{OR}$, etc. Describe an implementation of the $\mathrm{BIT\text{-}REVERSED\text{-}INCREMENT}$ procedure that allows the bit-reversal permutation on an $n$-element array to be performed in a total of $O(n)$ time.

**c.** Suppose that you can shift a word left or right by only one bit in unit time. Is it still possible to implement an $O(n)$-time bit-reversal permutation?

**a.** Initialize a second array of length $n$ to all trues, then, going through the indices of the original array in any order, if the corresponding entry in the second array is true, then swap the element at the current index with the element at the bit-reversed position, and set the entry in the second array corresponding to the bit-reversed index equal to false. Since we are running $\mathrm{rev}_k < n$ times, the total runtime is $O(nk)$.

**b.** Doing a bit reversed increment is the same thing as adding a one to the leftmost position where all carries are going to the left instead of the right.

```
BIT-REVERSED-INCREMENT(a)
    let m be a 1 followed by k - 1 0s
    while m bitwise-AND is not zero
        a = a bitwise-XOR m
        shift m right by 1
    m bitwise-OR a
```

By a similar analysis to the binary counter (just look at the problem in a mirror), this $\text{BIT-REVERSED-INCREMENT}$ will take constant ammortized time. So, to perform the bit-reversed permutation, have a normal binary counter and a bit reversed counter, then, swap the values of the two counters and increment. Do not swap however if those pairs of elements have already been swapped, which can be kept track of in a auxiliary array.

**c.** The $\text{BIT-REVERSED-INCREMENT}$ procedure given in the previous part only uses single shifts to the right, not arbitrary shifts.

## Problem 17-2 Making binary search dynamic

> Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.
>
> Specifically, suppose that we wish to support $\text{SEARCH}$ and $\text{INSERT}$ on a set of $n$ elements. Let $k = \lceil \lg(n + 1) \rceil$, and let the binary representation of $n$ be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. We have $k$ sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k - 1$, the length of array $A_i$ is $2^i$. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all $k$ arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.
>
> **a.** Describe how to perform the $\text{SEARCH}$ operation for this data structure. Analyze its worst-case running time.
>
> **b.** Describe how to perform the $\text{INSERT}$ operation. Analyze its worst-case and amortized running times.
>
> **c.** Discuss how to implement $\text{DELETE}$.

**a.** We linearly go through the lists and binary search each one since we don't know the relationship between one list an another. In the worst case, every list is actually used. Since list $i$ has length $2^i$ and it's sorted, we can search it in $O(i)$ time. Since $i$ varies from $0$ to $O(\lg n)$, the runtime of SEARCH is $O(\lg^2 n)$.

**b.** To insert, we put the new element into $A_0$ and update the lists accordingly. In the worst case, we must combine lists $A_0, A_1, \ldots, A_{m-1}$ into list Am. Since merging two sorted lists can be done linearly in the total length of the lists, the time this takes is $O(2^m)$. In the worst case, this takes time $O(n)$ since $m$ could equal $k$.

We'll use the accounting method to analyse the amoritized cost. Assign a cost of $\lg n$ to each insertion. Thus, each item carries $\lg n$ credit to pay for its later merges as additional items are inserted. Since an individual item can only be merged into a larger list and there are only $\lg n$ lists, the credit pays for all future costs the item might incur. Thus, the amortized cost is $O(\lg n)$.

**c.** Find the smallest $m$ such that $n_m \neq 0$ in the binary representation of $n$. If the item to be deleted is not in list $A_m$, remove it from its list and swap in an item from Am, arbitrarily. This can be done in $O(\lg n)$ time since we may need to search list $A_k$ to find the element to be deleted. Now simply break list $A_m$ into lists $A_0, A_1, \ldots, A_{m-1}$ by index. Since the lists are already sorted, the runtime comes entirely from making the splits, which takes $O(m)$ time. In the worst case, this is $O(\lg n)$.

## Problem 17-3 Amortized weight-balanced trees

> Consider an ordinary binary search tree augmented by adding to each node $x$ the attribute $x.size$ giving the number of keys stored in the subtree rooted at $x$. Let $\alpha$ be a constant in the range $1/2 \le \alpha < 1$. We say that a given node $x$ is ***$\alpha$-balanced*** if $x.left.size \le \alpha \cdot x.size$ and $x.right.size \le \alpha \cdot x.size$. The tree as a whole is ***$\alpha$-balanced*** if every node in the tree is $\alpha$-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.
>
> **a.** A $1/2$-balanced tree is, in a sense, as balanced as it can be. Given a node $x$ in an arbitrary binary search tree, show how to rebuild the subtree rooted at $x$ so that it becomes $1/2$-balanced. Your algorithm should run in time $\Theta(x.size)$, and it can use $O(x.size)$ auxiliary storage.
>
> **b.** Show that performing a search in an $n$-node $\alpha$-balanced binary search tree takes $O(\lg n)$ worst-case time.
>
> For the remainder of this problem, assume that the constant $\alpha$ is strictly greater than $1/2$. Suppose that we implement $\text{INSERT}$ and $\text{DELETE}$ as usual for an $n$-node binary search tree, except that after every such operation, if any node in the tree is no longer $\alpha$-balanced, then we "rebuild" the subtree rooted at the highest such node in the tree so that it becomes $1/2$-balanced.
>
> We shall analyze this rebuilding scheme using the potential method. For a node $x$ in a binary search tree $T$, we define
>
> $$\Delta(x) = |x.left.size - x.right.size|,$$
>
> and we define the potential of $T$ as
>
> $$\Phi(T) = c \sum_{x \in T : \Delta(x) \ge 2} \Delta(x),$$
>
> where $c$ is a sufficiently large constant that depends on $\alpha$.

> **c.** Argue that any binary search tree has nonnegative potential and that a $1/2$-balanced tree has potential $0$.
>
> **d.** Suppose that $m$ units of potential can pay for rebuilding an $m$-node subtree. How large must $c$ be in terms of $\alpha$ in order for it to take $O(1)$ amortized time to rebuild a subtree that is not $\alpha$-balanced?
>
> **e.** Show that inserting a node into or deleting a node from an $n$-node $\alpha$-balanced tree costs $O(\lg n)$ amortized time.

**a.** Since we have $O(x.\text{size})$ auxiliary space, we will take the tree rooted at $x$ and write down an inorder traversal of the tree into the extra space. This will only take linear time to do because it will visit each node thrice, once when passing to its left child, once when the nodes value is output and passing to the right child, and once when passing to the parent. Then, once the inorder traversal is written down, we can convert it back to a binary tree by selecting the median of the list to be the root, and recursing on the two halves of the list that remain on both sides. Since we can index into the middle element of a list in constant time, we will have the recurrence

$$T(n) = 2T(n/2) + 1,$$

which has solution that is linear. Since both trees come from the same underlying inorder traversal, the result is a $\text{BST}$ since the original was. Also, since the root at each point was selected so that half the elements are larger and half the elements are smaller, it is a $1/2$-balanced tree.

**b.** We will show by induction that any tree with $\le \alpha^{-d} + d$ elements has a depth of at most $d$. This is clearly true for $d = 0$ because any tree with a single node has depth $0$, and since $\alpha^0 = 1$, we have that our restriction on the number of elements requires there to only be one. Now, suppose that in some inductive step we had a contradiction, that is, some tree of depth $d$ that is $\alpha$ balanced but has more than $\alpha - d$ elements.

We know that both of the subtrees are alpha balanced, and by being alpha balanced at the root, we have

$$\text{root}.left.\text{size} \le \alpha \cdot \text{root}.\text{size},$$

which implies

$$\text{root}.\text{right}.\text{size} > \text{root}.\text{size} - \alpha \cdot \text{root}.\text{size} - 1.$$

So,

$$\begin{aligned} \text{root}.\text{right}.\text{size} & > (1 - \alpha)\text{root}.\text{size} - 1 \\ & > (1 - \alpha)\alpha - d + d - 1 \\ & = (\alpha - 1 - 1)\alpha - d + 1 + d - 1 \\ & \ge \alpha - d + 1 + d - 1, \end{aligned}$$

which is a contradiction to the fact that it held for all smaller values of $d$ because any child of a tree of depth d has depth $d - 1$.

**c.** The potential function is a sum of $\Delta(x)$ each of which is the absolute value of a quantity, so, since it is a sum of nonnegative values, it is nonnegative regardless of the input $\text{BST}$.

If we suppose that our tree is $1/2$-balanced, then, for every node $x$, we'll have that $\Delta(x) \le 1$, so, the sum we compute to find the potential will be over no nonzero terms.

**d.**

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

$$\begin{aligned}
\Delta(x) &= x.left.size - x.right.size \\
&\ge \alpha \cdot m - ((1-\alpha)m - 1) \\
&= (2\alpha - 1)m + 1.
\end{aligned}$$

$$\begin{aligned}
m &\le c((2\alpha - 1)m + 1) \\
c &\ge \frac{m}{(2\alpha - 1)m + 1} \\
&\ge \frac{1}{2\alpha}.
\end{aligned}$$

**e.** Suppose that our tree is $\alpha$ balanced. Then, we know that performing a search takes time $O(\lg(n))$. So, we perform that search and insert the element that we need to insert or delete the element we found. Then, we may have made the tree become unbalanced. However, we know that since we only changed one position, we have only changed the $\Delta$ value for all of the parents of the node that we either inserted or deleted. Therefore, we can rebuild the balanced properties starting at the lowest such unbalanced node and working up.

Since each one only takes ammortized constant time, and there are $O(\lg(n))$ many trees made unbalanced, tot total time to rebalanced every subtree is $O(\lg(n))$ ammortized time.

## Problem 17-4 The cost of restructuring red-black trees

> There are four basic operations on red-black trees that perform ***structural modifications***: node insertions, node deletions, rotations, and color changes. We have seen that $\text{RB-INSERT}$ and $\text{RB-DELETE}$ use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.
>
> **a.** Describe a legal red-black tree with $n$ nodes such that calling $\text{RB-INSERT}$ to add the $(n + 1)$st node causes $\Omega(\lg n)$ color changes. Then describe a legal red-black tree with $n$ nodes for which calling $\text{RB-DELETE}$ on a particular node causes $\Omega(\lg n)$ color changes.
>
> Although the worst-case number of color changes per operation can be logarithmic, we shall prove that any sequence of $m$ $\text{RB-INSERT}$ and $\text{RB-DELETE}$ operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case. Note that we count each color change as a structural modification.

> **b.** Some of the cases handled by the main loop of the code of both $\text{RB-INSERT-FIXUP}$ and $\text{RB-DELETE-FIXUP}$ are ***terminating***: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of $\text{RB-INSERT-FIXUP}$ and $\text{RB-DELETE-FIXUP}$, specify which are terminating and which are not. ($\text{Hint}$: Look at Figures 13.5, 13.6, and 13.7.)
>
> We shall first analyze the structural modifications when only insertions are performed. Let $T$ be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in $T$. Assume that $1$ unit of potential can pay for the structural modifications performed by any of the three cases of $\text{RB-INSERT-FIXUP}$.
>
> **c.** Let $T'$ be the result of applying Case 1 of $\text{RB-INSERT-FIXUP}$ to $T$. Argue that $\Phi(T') = \Phi(T) - 1$.
>
> **d.** When we insert a node into a red-black tree using $\text{RB-INSERT}$, we can break the operation into three parts. List the structural modifications and potential changes resulting from lines 1–16 of $\text{RB-INSERT}$, from nonterminating cases of $\text{RB-INSERT-FIXUP}$, and from terminating cases of $\text{RB-INSERT-FIXUP}$.
>
> **e.** Using part (d), argue that the amortized number of structural modifications performed by any call of $\text{RB-INSERT is O(1)}$.
>
> We now wish to prove that there are $O(m)$ structural modifications when there are both insertions and deletions. Let us define, for each node $x$,
>
> $$w(x) = \begin{cases} 0 & \text{if x is red,} \\ 1 & \text{if x is black and has no red children,} \\ 0 & \text{if x is black and has one red children,} \\ 2 & \text{if x is black and has two red children.} \end{cases}$$
>
> Now we redefine the potential of a red-black tree $T$ as
>
> $$\Phi(T) = \sum_{x \in T} w(x),$$
>
> and let $T'$ be the tree that results from applying any nonterminating case of $\text{RB-INSERT-FIXUP}$ or $\text{RB-DELETE-FIXUP to T}$.
>
> **f.** Show that $\Phi(T') \le \Phi(T) - 1$ for all nonterminating cases of $\text{RB-INSERT-FIXUP}$. Argue that the amortized number of structural modifications performed by any call of $\text{RB-INSERT-FIXUP is O(1)}$.
>
> **g.** Show that $\Phi(T') \le \Phi(T) - 1$ for all nonterminating cases of $\text{RB-DELETE-FIXUP}$. Argue that the amortized number of structural modifications performed by any call of $\text{RB-DELETE-FIXUP is O(1)}$.
>
> **h.** Complete the proof that in the worst case, any sequence of $m$ $\text{RB-INSERT}$ and $\text{RB-DELETE}$ operations performs $O(m)$ structural modifications.

**a.** If we insert a node into a complete binary search tree whose lowest level is all red, then there will be $\Omega(\lg n)$ instances of case 1 required to switch the colors all the way up the tree. If we delete a node from an all-black, complete binary tree then this also requires $\Omega(\lg n)$ time because there will be instances of case 2 at each iteration of the **while** loop.

**b.** For $\text{RB-INSERT}$, cases 2 and 3 are terminating. For $\text{RB-DELETE}$, cases 1 and 3 are terminating.

**c.** After applying case 1, $z$'s parent and uncle have been changed to black and $z$'s grandparent is changed to red. Thus, there is a ned loss of one red node, so $\Phi(T') = \Phi(T) - 1$.

**d.** For case 1, there is a single decrease in the number of red nodes, and thus a decrease in the potential function. However, a single call to $\text{RB-INSERTFIXUP}$ could result in $\Omega(\lg n)$ instances of case 1. For cases 2 and 3, the colors stay the same and each performs a rotation.

**e.** Since each instance of case 1 requires a specific node to be red, it can't decrease the number of red nodes by more than $\Phi(T)$. Therefore the potential function is always non-negative. Any insert can increase the number of red nodes by at most $1$, and one unit of potential can pay for any structural modifications of any of the 3 cases. Note that in the worst case, the call to $\text{RB-INSERT}$ has to perform $k$ case-1 operations, where $k$ is equal to $\Phi(T_i) - \Phi(T_{i-1})$. Thus, the total amortized cost is bounded above by $2(\Phi(T_n) - \Phi(T_0)) \le n$, so the amortized cost of each insert is $O(1)$.

**f.** In case 1 of $\text{RB-INSERT}$, we reduce the number of black nodes with two red children by $1$ and we at most increase the number of black nodes with no red children by $1$, leaving a net loss of at most $1$ to the potential function. In our new potential function, $\Phi(T_n) - \Phi(T_0) \le n$. Since one unit of potential pays for each operation and the terminating cases cause constant structural changes, the total amortized cost is $O(n)$ making the amortized cost of each $\text{RB-INSERT-FIXUP}$ $O(1)$.

**g.** In case 2 of $\text{RB-DELETE}$, we reduce the number of black nodes with two red children by $1$, thereby reducing the potential function by $2$. Since the change in potential is at least negative $1$, it pays for the structural modifications. Since the other cases cause constant structural changes, the total amortized cost is $O(n)$ making the amortized cost of each $\text{RB-DELETE-FIXUP}$ $O(1)$.

**h.** As described above, whether we insert or delete in any of the cases, the potential function always pays for the changes made if they're nonterminating. If they're terminating then they already take constant time, so the amortized cost of any operation in a sequence of $m$ inserts and deletes is $O(1)$, making the toal amortized cost $O(m)$.

## Problem 17-5 Competitive analysis of self-organizing lists with move-to-front

> A ***self-organizing list*** is a linked list of $n$ elements, in which each element has a unique key. When we search for an element in the list, we are given a key, and we want to find an element with that key.
>
> A self-organizing list has two important properties:
>
> 1. To find an element in the list, given its key, we must traverse the list from the beginning until we encounter the element with the given key. If that element is the $k$th element from the start of the

list, then the cost to find the element is $k$.

2. We may reorder the list elements after any operation, according to a given rule with a given cost. We may choose any heuristic we like to decide how to reorder the list.

Assume that we start with a given list of $n$ elements, and we are given an access sequence $\sigma = \langle \sigma_1, \sigma_2, \dots, \sigma_m \rangle$ of keys to find, in order. The cost of the sequence is the sum of the costs of the individual accesses in the sequence.

Out of the various possible ways to reorder the list after an operation, this problem focuses on transposing adjacent list elements-switching their positions in the list—with a unit cost for each transpose operation. You will show, by means of a potential function, that a particular heuristic for reordering the list, move-to-front, entails a total cost no worse than $4$ times that of any other heuristic for maintaining the list order—even if the other heuristic knows the access sequence in advance! We call this type of analysis a ***competitive analysis***.

For a heuristic $H$ and a given initial ordering of the list, denote the access cost of sequence $\sigma$ by $C_H(\sigma)$ Let $m$ be the number of accesses in $\sigma$.

**a.** Argue that if heuristic $H$ does not know the access sequence in advance, then the worst-case cost for $H$ on an access sequence $\sigma$ is $C_H(\sigma) = \Omega(mn)$.

With the ***move-to-front*** heuristic, immediately after searching for an element $x$, we move $x$ to the first position on the list (i.e., the front of the list).

Let $\text{rank}_L(x)$ denote the rank of element $x$ in list $L$, that is, the position of $x$ in list $L$. For example, if $x$ is the fourth element in $L$, then $\text{rank}_L(x) = 4$. Let $c_i$ denote the cost of access $\sigma_i$ using the move-to-front heuristic, which includes the cost of finding the element in the list and the cost of moving it to the front of the list by a series of transpositions of adjacent list elements.

**b.** Show that if $\sigma_i$ accesses element $x$ in list $L$ using the move-to-front heuristic, then $c_i = 2 \cdot \text{rank}_L(x) - 1$.

Now we compare move-to-front with any other heuristic $H$ that processes an access sequence according to the two properties above. Heuristic $H$ may transpose elements in the list in any way it wants, and it might even know the entire access sequence in advance.

Let $L_i$ be the list after access $\sigma_i$ using move-to-front, and let $L_i^*$ be the list after access $\sigma_i$ using heuristic $H$. We denote the cost of access $\sigma_i$ by $c_i$ for move-to-front and by $c_i^*$ for heuristic $H$. Suppose that heuristic $H$ performs $t_i^*$ transpositions during access $\sigma_i$.

**c.** In part (b), you showed that $c_i = 2 \cdot \text{rank}_{L_{i-1}}(x) - 1$. Now show that $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i^*$.

We define an ***inversion*** in list $L_i$ as a pair of elements $y$ and $z$ such that $y$ precedes $z$ in $L_i$ and $z$ precedes $y$ in list $L_i^*$. Suppose that list $L_i$ has $q_i$ inversions after processing the access sequence $\langle \sigma_1, \sigma_2, \dots, \sigma_i \rangle$. Then, we define a potential function $\Phi$ that maps $L_i$ to a real number by $\Phi(L_i) = 2q_i$. For example, if $L_i$ has the elements $\langle e, c, a, d, b \rangle$ and $L_i^*$ has the elements $\langle c, a, b, d, e \rangle$, then $L_i$ has 5

> inversions $((e, c), (e, a), (e, d), (e, b), (d, b))$, and so $\Phi(L_i) = 10$. Observe that $\Phi(L_i) \geq 0$ for all $i$ and that, if move-to-front and heuristic $H$ start with the same list $L_0$, then $\Phi(L_0) = 0$.
>
> **d.** Argue that a transposition either increases the potential by $2$ or decreases the potential by $2$.
>
> Suppose that access $\sigma_i$ finds the element $x$. To understand how the potential changes due to $\sigma_i$, let us partition the elements other than $x$ into four sets, depending on where they are in the lists just before the $i$th access:
>
> - Set $A$ consists of elements that precede $x$ in both $L_{i-1}$ and $L^*_{i-1}$.
> - Set $B$ consists of elements that precede $x$ in $L_{i-1}$ and follow $x$ in $L^*_{i-1}$.
> - Set $C$ consists of elements that follow $x$ in $L_{i-1}$ and precede $x$ in $L^*_{i-1}$.
> - Set $D$ consists of elements that follow $x$ in both $L_{i-1}$ and $L^*_{i-1}$.
>
> **e.** Argue that $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$ and $\text{rank}_{L^*_{i-1}}(x) = |A| + |C| + 1$.
>
> **f.** Show that access $\sigma_i$ causes a change in potential of
>
> $$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t^*_i),$$
>
> where, as before, heuristic $H$ performs $t^*_i$ transpositions during access $\sigma_i$.
>
> Define the amortized cost $\hat{c_i}$ of access $\sigma_i$ by $\hat{c_i} = c_i + \Phi(L_i) - \Phi(L_{i-1})$.
>
> **g.** Show that the amortized cost $\hat{c_i}$ of access $\sigma_i$ is bounded from above by $4c^*_i$.
>
> **h.** Conclude that the cost $C_{MTF}(\sigma)$ of access sequence $\sigma$ with move-to-front is at most $4$ times the cost $C_H(\sigma)$ of $\sigma$ withany other heuristic $H$, assuming that both heuristics start with the same list.

**a.** Since the heuristic is picked in advance, given any sequence of requests given so far, we can simulate what ordering the heuristic will call for, then, we will pick our next request to be whatever element will of been in the last position of the list. Continuing until all the requests have been made, we have that the cost of this sequence of accesses is $= mn$.

**b.** The cost of finding an element is $= \text{rank}_L(x)$ and since it needs to be swapped with all the elements before it, of which there are $\text{rank}_L(x) - 1$, the total cost is $2 \cdot \text{rank}_L(x) - 1$.

**c.** Regardless of the heuristic used, we first need to locate the element, which is left where ever it was after the previous step, so, needs $\text{rank}_{L_{i-1}}(x)$. After that, by definition, there are $t_i$ transpositions made, so, $c^*_i = \text{rank}_{L_{i-1}}(x) + t^*_i$.

**d.** If we perform a transposition of elements $y$ and $z$, where $y$ is towards the left. Then there are two cases. The first is that the final ordering of the list in $L^*_i$ is with $y$ in front of $z$, in which case we have just increased the number of inversions by $1$, so the potential increases by $2$. The second is that in $L^*_I z$ occurs before $y$, in which case, we have just reduced the number of inversions by one, reducing the potential by $2$.

In both cases, whether or not there is an inversion between $y$ or $z$ and any other element has not changed, since the transposition only changed the relative ordering of those two elements.

**e.** By definition, $A$ and $B$ are the only two of the four categories to place elements that precede $x$ in $L_{i-1}$, since there are $|A| + |B|$ elements preceding it, it's rank in $L_{i-1}$ is $|A| + |B| + 1$. Similarly, the two categories in which an element can be if it precedes $x$ in $L^*_{i-1}$ are $A$ and $C$, so, in $L^*_{i-1}$, $x$ has $\text{rank}|A| + |C| + 1$.

**f.** We have from part d that the potential increases by $2$ if we transpose two elements that are being swapped so that their relative order in the final ordering is being screwed up, and decreases by two if they are begin placed into their correct order in $L^*_i$.

In particular, they increase it by at most $2$, since we are keeping track of the number of inversions that may not be the direct effect of the transpositions that heuristic $H$ made, we see which ones the Move to front heuristic may of added. In particular, since the move to front heuristic only changed the relative order of $x$ with respect to the other elements, moving it in front of the elements that preceded it in $L_{i-1}$, we only care about sets $A$ and $B$. For an element in $A$, moving it to be behind $A$ created an inversion, since that element preceded $x$ in $L^*_i$. However, if the element were in $B$, we are removing an inversion by placing $x$ in front of it.

**g.**

$$\begin{aligned} \hat{c_i} & \le 2(|A| + |B| + 1) - 1 + 2(|A| - |B| + t^*_i) \\ & = 4|A| + 1 + 2t^*_i \\ & \le 4(|A| + |C| + 1 + t^*_i) \\ & = 4c^*_i. \end{aligned}$$

**h.** We showed that the amortized cost of each operation under the move to front heuristic was at most four times the cost of the operation using any other heuristic. Since the amortized cost added up over all these operation is at most the total (real) cost, so we have that the total cost with movetofront is at most four times the total cost with an arbitrary other heuristic.