Amortization
The goode workes that men don whil they ben in good lif al amortised by synne folwyng.  
— Geoffrey Chaucer, “The Persones [Parson’s] Tale” (c.1400)

I will gladly pay you Tuesday for a hamburger today.  
— J. Wellington Wimpy, “Thimble Theatre” (1931)

I want my two dollars!  
— Johnny Gasparini [Demian Slade], “Better Off Dead” (1985)

A dollar here, a dollar there. Over time, it adds up to two dollars.  
— Jarod Kintz, The Titanic Would Never Have Sunk if It Were Made out of a Sink (2012)

15 Amortized Analysis

15.1 Incrementing a Binary Counter

It is a straightforward exercise in induction, which often appears on Homework 0, to prove that any non-negative integer \( n \) can be represented as the sum of distinct powers of 2. Although some students correctly use induction on the number of bits—pulling off either the least significant bit or the most significant bit in the binary representation and letting the Recursion Fairy convert the remainder—the most commonly submitted proof uses induction on the value of the integer, as follows:

**Proof:** The base case \( n = 0 \) is trivial. For any \( n > 0 \), the inductive hypothesis implies that there is set of distinct powers of 2 whose sum is \( n - 1 \). If we add \( 2^0 \) to this set, we obtain a multiset of powers of two whose sum is \( n \), which might contain two copies of \( 2^0 \). Then as long as there are two copies of any \( 2^i \) in the multiset, we remove them both and insert \( 2^{i+1} \) in their place. The sum of the elements of the multiset is unchanged by this replacement, because \( 2^{i+1} = 2^i + 2^i \). Each iteration decreases the size of the multiset by 1, so the replacement process must eventually terminate. When it does terminate, we have a set of distinct powers of 2 whose sum is \( n \). □

This proof is describing an algorithm to increment a binary counter from \( n - 1 \) to \( n \). Here’s a more formal (and shorter!) description of the algorithm to add 1 to a binary counter. The input \( B \) is an (infinite) array of bits, where \( B[i] = 1 \) if and only if \( 2^i \) appears in the sum.

```
INCREMENT(B[0..\infty]):
  i ← 0
  while B[i] = 1
    B[i] ← 0
    i ← i + 1
  B[i] ← 1
```

We’ve already argued that \( \text{INCREMENT} \) must terminate, but how quickly? Obviously, the running time depends on the array of bits passed as input. If the first \( k \) bits are all 1s, then \( \text{INCREMENT} \) takes \( \Theta(k) \) time. The binary representation of any positive integer \( n \) is exactly \( \lfloor \lg n \rfloor + 1 \) bits long. Thus, if \( B \) represents an integer between 0 and \( n \), \( \text{INCREMENT} \) takes \( \Theta(\log n) \) time in the worst case.
15.2 Counting from 0 to $n$

Now suppose we call \texttt{INCREMENT} $n$ times, starting with a zero counter. How long does it take to count from 0 to $n$? If we only use the worst-case running time for each \texttt{INCREMENT}, we get an upper bound of $O(n \log n)$ on the total running time. Although this bound is correct, we can do better; in fact, the total running time is only $\Theta(n)$. This section describes several general methods for deriving, or at least proving, this linear time bound. Many (perhaps even all) of these methods are logically equivalent, but different formulations are more natural for different problems.

15.2.1 Summation

Perhaps the simplest way to derive a tighter bound is to observe that \texttt{INCREMENT} doesn’t flip $\Theta(\log n)$ bits every time it is called. The least significant bit $B[0]$ does flip in every iteration, but $B[1]$ only flips every other iteration, $B[2]$ flips every 4th iteration, and in general, $B[i]$ flips every $2^i$th iteration. Because we start with an array full of 0’s, a sequence of $n$ \texttt{INCREMENT}s flips each bit $B[i]$ exactly $\lceil n/2^i \rceil$ times. Thus, the total number of bit-flips for the entire sequence is

$$\sum_{i=0}^{\lfloor \log n \rfloor} \frac{n}{2^i} < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$ 

(More precisely, the number of flips is exactly $2n - \#1(n)$, where $\#1(n)$ is the number of 1 bits in the binary representation of $n$.) Thus, on average, each call to \texttt{INCREMENT} flips just less than two bits, and therefore runs in constant time.

This sense of “on average” is quite different from the averaging we consider with randomized algorithms. There is no probability involved; we are averaging over a sequence of operations, not the possible running times of a single operation. This averaging idea is called amortization—the amortized time for each \texttt{INCREMENT} is $O(1)$. Amortization is a sleazy clever trick used by accountants to average large one-time costs over long periods of time; the most common example is calculating uniform payments for a loan, even though the borrower is paying interest on less and less capital over time. For this reason, it is common to use “cost” as a synonym for running time in the context of amortized analysis. Thus, the worst-case cost of \texttt{INCREMENT} is $O(\log n)$, but the amortized cost is only $O(1)$.

Most textbooks call this particular technique “the aggregate method”, or “aggregate analysis”, but these are just fancy names for computing the total cost of all operations and then dividing by the number of operations.

**The Summation Method.** Let $T(n)$ be the worst-case running time for a sequence of $n$ operations. The amortized time for each operation is $T(n)/n$.

15.2.2 Taxation

A second method we can use to derive amortized bounds is called either the accounting method or the taxation method. Suppose it costs us a dollar to toggle a bit, so we can measure the running time of our algorithm in dollars. Time is money!

Instead of paying for each bit flip when it happens, the Increment Revenue Service charges a two-dollar increment tax whenever we want to set a bit from zero to one. One of those dollars is spent changing the bit from zero to one; the other is stored away as credit until we need to reset the same bit to zero. The key point here is that we always have enough credit saved up to pay for
the next \textsc{Increment}. The amortized cost of an \textsc{Increment} is the total tax it incurs, which is exactly \$2, since each \textsc{Increment} changes just one bit from 0 to 1.

It is often useful to distribute the tax income to specific pieces of the data structure. For example, for each \textsc{Increment}, we could store one of the two dollars on the single bit that is set for 0 to 1, so that \textit{that} bit can pay to reset itself back to zero later on.

**Taxation Method 1.** Certain steps in the algorithm charge you taxes, so that the total cost incurred by the algorithm is never more than the total tax you pay. The amortized cost of an operation is the overall tax charged to you during that operation.

A different way to schedule the taxes is for every bit to charge us a tax at every operation, regardless of whether the bit changes or not. Specifically, each bit $B[i]$ charges a tax of $\frac{1}{2^i}$ dollars for each \textsc{Increment}. The total tax we are charged during each \textsc{Increment} is $\sum_{i \geq 0} 2^{-i} = 2$ dollars. Every time a bit $B[i]$ actually needs to be flipped, it has collected exactly \$1, which is just enough for us to pay for the flip.

**Taxation Method 2.** Certain portions of the data structure charge you taxes at each operation, so that the total cost of maintaining the data structure is never more than the total taxes you pay. The amortized cost of an operation is the overall tax you pay during that operation.

In both of the taxation methods, our task as algorithm analysts is to come up with an appropriate ‘tax schedule’. Different ‘schedules’ can result in different amortized time bounds. The tightest bounds are obtained from tax schedules that just barely stay in the black.

### 15.2.3 Charging

Another common method of amortized analysis involves \textit{charging} the cost of some steps to some other, earlier steps. The method is similar to taxation, except that we focus on where each unit of tax is (or will be) spent, rather than where is it collected. By charging the cost of some operations to earlier operations, we are overestimating the total cost of any sequence of operations, since we pay for some charges from future operations that may never actually occur.

**The Charging Method.** Charge the cost of some steps of the algorithm to earlier steps, or to steps in some earlier operation. The amortized cost of the algorithm is its actual running time, minus its total charges to past operations, plus its total charge from future operations.

For example, in our binary counter, suppose we charge the cost of clearing a bit (changing its value from 1 to 0) to the previous operation that sets that bit (changing its value from 0 to 1). If we flip $k$ bits during an \textsc{Increment}, we charge $k - 1$ of those bit-flips to earlier bit-flips. Conversely, the single operation that sets a bit receives at most one unit of charge from the next time that bit is cleared. So instead of paying for $k$ bit-flips, we pay for at most two: one for actually setting a bit, plus at most one charge from the future for clearing that same bit. Thus, the total amortized cost of the \textsc{Increment} is at most two bit-flips.

We can visualize this charging scheme as follows. For each integer $i$, we represent the running time of the $i$th \textsc{Increment} as a stack of blocks, one for each bit flip. The $j$th block in the $i$th stack is white if the $i$th \textsc{Increment} changes $B[j]$ from 0 to 1, and shaded if the $i$th \textsc{Increment} changes $B[j]$ from 1 to 0. If we moved each shaded block onto the white block directly to its left, there would at most two blocks in each stack.
15.2.4 Potential

The most powerful method (and the hardest to use) builds on a physics metaphor of ‘potential energy’. Instead of associating costs or taxes with particular operations or pieces of the data structure, we represent prepaid work as potential that can be spent on later operations. The potential is a function of the entire data structure.

Let $D_i$ denote our data structure after $i$ operations have been performed, and let $\Phi_i$ denote its potential. Let $c_i$ denote the actual cost of the $i$th operation (which changes $D_{i-1}$ into $D_i$). Then the amortised cost of the $i$th operation, denoted $a_i$, is defined to be the actual cost plus the increase in potential:

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

So the total amortized cost of $n$ operations is the actual total cost plus the total increase in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$ 

A potential function is valid if $\Phi_i - \Phi_0 \geq 0$ for all $i$. If the potential function is valid, then the total actual cost of any sequence of operations is always less than the total amortized cost:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

For our binary counter example, we can define the potential $\Phi_i$ after the $i$th increment to be the number of bits with value 1. Initially, all bits are equal to zero, so $\Phi_0 = 0$, and clearly $\Phi_i > 0$ for all $i > 0$, so this is a valid potential function. We can describe both the actual cost of an increment and the change in potential in terms of the number of bits set to 1 and reset to 0.

$$c_i = \#\text{bits changed from 0 to 1} + \#\text{bits changed from 1 to 0}$$

$$\Phi_i - \Phi_{i-1} = \#\text{bits changed from 0 to 1} - \#\text{bits changed from 1 to 0}$$

Thus, the amortized cost of the $i$th increment is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \#\text{bits changed from 0 to 1}$$

Since increment changes only one bit from 0 to 1, the amortized cost increment is 2.

**The Potential Method.** Define a potential function for the data structure that is initially equal to zero and is always non-negative. The amortized cost of an operation is its actual cost plus the change in potential.
For this particular example, the potential is precisely the total unspent taxes paid using the taxation method, so it should be no surprise that we obtain precisely the same amortized cost. In general, however, there may be no natural way to interpret change in potential as “taxes” or “charges”. Taxation and charging are useful when there is a convenient way to distribute costs to specific steps in the algorithm or components of the data structure. Potential arguments allow us to argue more globally when a local distribution is difficult or impossible.

Different potential functions can lead to different amortized time bounds. The trick to using the potential method is to come up with the best possible potential function. A good potential function goes up a little during any cheap/fast operation, and goes down a lot during any expensive/slow operation. Unfortunately, there is no general technique for finding good potential functions, except to play around with the data structure and try lots of possibilities (most of which won't work).

### 15.3 Incrementing and Decrementing

Now suppose we wanted a binary counter that we can both increment and decrement efficiently. A standard binary counter won’t work, even in an amortized sense; if we alternate between $2^k$ and $2^k - 1$, every operation costs $\Theta(k)$ time.

A nice alternative is represent each integer as a pair $(P, N)$ of bit strings, subject to the invariant $P \land N = 0$ where $\land$ represents bit-wise AND. In other words,

*For every index $i$, at most one of the bits $P[i]$ and $N[i]$ is equal to 1.*

If we interpret $P$ and $N$ as binary numbers, the actual value of the counter is $P - N$; thus, intuitively, $P$ represents the “positive” part of the pair, and $N$ represents the “negative” part. Unlike the standard binary representation, this new representation is not unique, except for zero, which can only be represented by the pair $(0, 0)$. In fact, every positive or negative integer can be represented has an *infinite* number of distinct representations.

We can increment and decrement our double binary counter as follows. Intuitively, the `INCREMENT` algorithm increments $P$, and the `DECREMENT` algorithm increments $N$; however, in both cases, we must change the increment algorithm slightly to maintain the invariant $P \land N = 0$.

\[
\text{INCREMENT}(P, N): \quad \begin{align*}
i &\leftarrow 0 \\
\text{while } P[i] &= 1 \\
P[i] &\leftarrow 0 \\
i &\leftarrow i + 1 \\
\text{if } N[i] &= 1 \\
&\text{else} \\
P[i] &\leftarrow 1
\end{align*}
\]

\[
\text{DECREMENT}(P, N): \quad \begin{align*}
i &\leftarrow 0 \\
\text{while } N[i] &= 1 \\
N[i] &\leftarrow 0 \\
i &\leftarrow i + 1 \\
\text{if } P[i] &= 1 \\
&\text{else} \\
N[i] &\leftarrow 1
\end{align*}
\]

\[
P = 10001 \quad P = 10010 \quad P = 10011 \quad P = 10000 \quad P = 10000 \quad P = 10000 \quad P = 10001 \\
N = 01100 \quad N = 01100 \quad N = 01100 \quad N = 01000 \quad N = 01000 \quad N = 01001 \quad N = 01010 \\
P - N = 5 \quad P - N = 6 \quad P - N = 7 \quad P - N = 8 \quad P - N = 7 \quad P - N = 6 \quad P - N = 7
\]

Incrementing and decrementing a double-binary counter.

Now suppose we start from $(0, 0)$ and apply a sequence of $n$ INCREMENTS and DECREMENTS. In the worst case, each operation takes $\Theta(\log n)$ time, but what is the amortized cost? We can’t
use the aggregate method here, because we don’t know what the sequence of operations looks like.

What about taxation? It’s not hard to prove (by induction, of course) that after either \( P[i] \) or \( N[i] \) is set to 1, there must be at least \( 2^i \) operations, either increments or decrements, before that bit is reset to 0. So if each bit \( P[i] \) and \( N[i] \) pays a tax of \( 2^{-i} \) at each operation, we will always have enough money to pay for the next operation. Thus, the amortized cost of each operation is at most \( \sum_{i \geq 0} 2 \cdot 2^{-i} = 4 \).

We can get even better amortized time bounds using the potential method. Define the potential \( \Phi_i \) to be the number of 1-bits in both \( P \) and \( N \) after \( i \) operations. Just as before, we have

\[
c_i = \text{#bits changed from 0 to 1} + \text{#bits changed from 1 to 0}
\]

\[
\Phi_i - \Phi_{i-1} = \text{#bits changed from 0 to 1} - \text{#bits changed from 1 to 0}
\]

\[
\implies a_i = 2 \times \text{#bits changed from 0 to 1}
\]

Since each operation changes at most one bit to 1, the \( i \)th operation has amortized cost \( a_i \leq 2 \).

**15.4 Gray Codes**

An attractive alternate solution to the increment/decrement problem was independently suggested by several students. Gray codes (named after Frank Gray, who discovered them in the 1950s) are methods for representing numbers as bit strings so that successive numbers differ by only one bit. For example, here is the four-bit binary reflected Gray code for the integers 0 through 15:

\[
0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000
\]

The general rule for incrementing a binary reflected Gray code is to invert the bit that would be set from 0 to 1 by a normal binary counter. In terms of bit-flips, this is the perfect solution; each increment of decrement by definition changes only one bit. Unfortunately, the naïve algorithm to find the single bit to flip still requires \( \Theta(\log n) \) time in the worst case. Thus, so the total cost of maintaining a Gray code, using the obvious algorithm, is the same as that of maintaining a normal binary counter.

Fortunately, this is only true of the naïve algorithm. The following algorithm, discovered by Gideon Ehrlich\(^1\) in 1973, maintains a Gray code counter in constant worst-case time per increment! The algorithm uses a separate ‘focus’ array \( F[0..n] \) in addition to a Gray-code bit array \( G[0..n-1] \).

\begin{verbatim}
EHRLICHGRAYINIT(n):
    for i ← 0 to n − 1
        G[i] ← 0
    for i ← 0 to n
        F[i] ← i

EHRLICHGRAYINCREMENT(n):
    j ← F[0]
    F[0] ← 0
    if j = n
        G[n − 1] ← 1 − G[n − 1]
    else
        G[j] = 1 − G[j]
        F[j] ← F[j + 1]
        F[j + 1] ← j + 1
\end{verbatim}

The EHRICHGRAYINCREMENT algorithm obviously runs in $O(1)$ time, even in the worst case. Here's the algorithm in action with $n = 4$. The first line is the Gray bit-vector $G$, and the second line shows the focus vector $F$, both in reverse order:

$$G: 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$

$$F: 3210, 3211, 3220, 3212, 3310, 3311, 3230, 3213, 4210, 4211, 4220, 4212, 3410, 3411, 3240, 3214$$

Voodoo! I won’t explain in detail how Ehrlich’s algorithm works, except to point out the following invariant. Let $B[i]$ denote the $i$th bit in the standard binary representation of the current number. If $B[j] = 0$ and $B[j - 1] = 1$, then $F[j]$ is the smallest integer $k > j$ such that $B[k] = 1$; otherwise, $F[j] = j$. Got that?

But wait — this algorithm only handles increments; what if we also want to decrement? Sorry, I don’t have a clue. Extra credit, anyone?

### 15.5 Generalities and Warnings

Although computer scientists usually apply amortized analysis to understand the efficiency of maintaining and querying data structures, you should remember that amortization can be applied to any sequence of numbers. Banks have been using amortization to calculate fixed payments for interest-bearing loans for centuries. The IRS allows taxpayers to amortize business expenses or gambling losses across several years for purposes of computing income taxes. Some cell phone contracts let you to apply amortization to calling time, by rolling unused minutes from one month into the next month.

It’s also important to remember that **amortized time bounds are not unique**. For a data structure that supports multiple operations, different amortization schemes can assign different costs to exactly the same algorithms. For example, consider a generic data structure that can be modified by three algorithms: **Fold**, **Spindle**, and **Mutilate**. One amortization scheme might imply that **Fold** and **Spindle** each run in $O(\log n)$ amortized time, while **Mutilate** runs in $O(n)$ amortized time. Another scheme might imply that **Fold** runs in $O(\sqrt{n})$ amortized time, while **Spindle** and **Mutilate** each run in $O(1)$ amortized time. These two results are not necessarily inconsistent! Moreover, there is no general reason to prefer one of these sets of amortized time bounds over the other; our preference may depend on the context in which the data structure is used.

### Exercises

1. Suppose we are maintaining a data structure under a series of $n$ operations. Let $f(k)$ denote the actual running time of the $k$th operation. For each of the following functions $f$, determine the resulting amortized cost of a single operation. (For practice, try all of the methods described in this note.)

   (a) $f(k)$ is the largest integer $i$ such that $2^i$ divides $k$.

   (b) $f(k)$ is the largest power of 2 that divides $k$.

   (c) $f(k) = n$ if $k$ is a power of 2, and $f(k) = 1$ otherwise.

   (d) $f(k) = n^2$ if $k$ is a power of 2, and $f(k) = 1$ otherwise.

   (e) $f(k)$ is the index of the largest Fibonacci number that divides $k$.

   (f) $f(k)$ is the largest Fibonacci number that divides $k$. 

7
(g) \( f(k) = k \) if \( k \) is a Fibonacci number, and \( f(k) = 1 \) otherwise.
(h) \( f(k) = k^2 \) if \( k \) is a Fibonacci number, and \( f(k) = 1 \) otherwise.
(i) \( f(k) \) is the largest integer whose square divides \( k \).
(j) \( f(k) \) is the largest perfect square that divides \( k \).
(k) \( f(k) = k \) if \( k \) is a perfect square, and \( f(k) = 1 \) otherwise.
(l) \( f(k) = k^2 \) if \( k \) is a perfect square, and \( f(k) = 1 \) otherwise.

(m) Let \( T \) be a complete binary search tree, storing the integer keys 1 through \( n \). \( f(k) \) is the number of ancestors of node \( k \).
(n) Let \( T \) be a complete binary search tree, storing the integer keys 1 through \( n \). \( f(k) \) is the number of descendants of node \( k \).
(o) Let \( T \) be a complete binary search tree, storing the integer keys 1 through \( n \). \( f(k) \) is the square of the number of ancestors of node \( k \).
(p) Let \( T \) be a complete binary search tree, storing the integer keys 1 through \( n \). \( f(k) = \text{size}(k) \log \text{size}(k) \), where \( \text{size}(k) \) is the number of descendants of node \( k \).
(q) Let \( T \) be an arbitrary binary search tree, storing the integer keys 0 through \( n \). \( f(k) \) is the length of the path in \( T \) from node \( k-1 \) to node \( k \).
(r) Let \( T \) be an arbitrary binary search tree, storing the integer keys 0 through \( n \). \( f(k) \) is the square of the length of the path in \( T \) from node \( k-1 \) to node \( k \).
(s) Let \( T \) be a complete binary search tree, storing the integer keys 0 through \( n \). \( f(k) \) is the square of the length of the path in \( T \) from node \( k-1 \) to node \( k \).

2. Consider the following modification of the standard algorithm for incrementing a binary counter.

\[
\text{INCREMENT}(B[0..\infty]):
\]
\[
i \leftarrow 0
\]
\[
\text{while } B[i] = 1
\]
\[
B[i] \leftarrow 0
\]
\[
i \leftarrow i + 1
\]
\[
B[i] \leftarrow 1
\]

\text{SOMETHINGELSE}(i)

The only difference from the standard algorithm is the function call at the end, to a black-box subroutine called \text{SOMETHINGELSE}.

Suppose we call \text{INCREMENT} \( n \) times, starting with a counter with value 0. The amortized time of each \text{INCREMENT} clearly depends on the running time of \text{SOMETHINGELSE}. Let \( T(i) \) denote the worst-case running time of \text{SOMETHINGELSE}(i). For example, we proved in class that \text{INCREMENT} algorithm runs in \( O(1) \) amortized time when \( T(i) = 0 \).

(a) What is the amortized time per \text{INCREMENT} if \( T(i) = 42 \)?
(b) What is the amortized time per \text{INCREMENT} if \( T(i) = 2^i \)?
(c) What is the amortized time per \text{INCREMENT} if \( T(i) = 4^i \)?
(d) What is the amortized time per \text{INCREMENT} if \( T(i) = \sqrt{2}^i \)?
(e) What is the amortized time per \text{INCREMENT} if \( T(i) = 2^i/(i+1) \)?
3. An **extendable array** is a data structure that stores a sequence of items and supports the following operations.

- `AddToFront(x)` adds `x` to the **beginning** of the sequence.
- `AddToEnd(x)` adds `x` to the **end** of the sequence.
- `Lookup(k)` returns the `k`th item in the sequence, or **NULL** if the current length of the sequence is less than `k`.

Describe a **simple** data structure that implements an extendable array. Your `AddToFront` and `AddToEnd` algorithms should take `O(1)` amortized time, and your `Lookup` algorithm should take `O(1)` worst-case time. The data structure should use `O(n)` space, where `n` is the **current** length of the sequence.

4. An **ordered stack** is a data structure that stores a sequence of items and supports the following operations.

- `OrderedPush(x)` removes all items smaller than `x` from the beginning of the sequence and then adds `x` to the beginning of the sequence.
- `Pop` deletes and returns the first item in the sequence (or **NULL** if the sequence is empty).

Suppose we implement an ordered stack with a simple linked list, using the obvious `OrderedPush` and `Pop` algorithms. Prove that if we start with an empty data structure, the amortized cost of each `OrderedPush` or `Pop` operation is `O(1)`.

5. A **multistack** consists of an infinite series of stacks `S_0, S_1, S_2, ...`, where the `i`th stack `S_i` can hold up to `3^i` elements. The user always pushes and pops elements from the smallest stack `S_0`. However, before any element can be pushed onto any full stack `S_i`, we first pop all the elements off `S_i` and push them onto stack `S_{i+1}` to make room. (Thus, if `S_{i+1}` is already full, we first recursively move all its members to `S_{i+2}`.) Similarly, before any element can be popped from any empty stack `S_i`, we first pop `3^i` elements from `S_{i+1}` and push them onto `S_i` to make room. (Thus, if `S_{i+1}` is already empty, we first recursively fill it by popping elements from `S_{i+2}`.) Moving a single element from one stack to another takes `O(1)` time.

Here is pseudocode for the multistack operations `MPush` and `MPop`. The internal stacks are managed with the subroutines `Push` and `Pop`.

```
MPush(x):
    i ← 0
    while S_i is full
        i ← i + 1
    while i > 0
        i ← i - 1
        for j ← 1 to 3^i
            Push(S_{i+1}, Pop(S_i))
    Push(S_0, x)

MPop(x):
    i ← 0
    while S_i is empty
        i ← i + 1
    while i > 0
        i ← i - 1
        for j ← 1 to 3^i
            Push(S_i, Pop(S_{i+1}))
    return Pop(S_0)
```

(a) In the worst case, how long does it take to push one more element onto a multistack containing `n` elements?
b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where $n$ is the maximum number of elements in the multistack during its lifetime.

c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where $n$ is the maximum number of elements in the multistack during its lifetime.

6. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

- **push(x)**: Add item $x$ to the end of the sequence.
- **pull()**: Remove and return the item at the beginning of the sequence.

It is easy to implement a queue using a doubly-linked list and a counter, so that the entire data structure uses $O(n)$ space (where $n$ is the number of items in the queue) and the worst-case time per operation is $O(1)$.

(a) Now suppose we want to support the following operation instead of pull:

- **MultiPull(k)**: Remove the first $k$ items from the front of the queue, and return the $k$th item removed.

Suppose we use the obvious algorithm to implement MultiPull:

```
MultiPull(k):
    for i ← 1 to k
    x ← pull()
    return x
```

Prove that in any intermixed sequence of push and MultiPull operations, the amortized cost of each operation is $O(1)$.

(b) Now suppose we also want to support the following operation instead of push:

- **MultiPush(x, k)**: Insert $k$ copies of $x$ into the back of the queue.

Suppose we use the obvious algorithm to implement MultiPush:

```
MultiPush(k, x):
    for i ← 1 to k
    push(x)
```
Prove that for any integers \( \ell \) and \( n \), there is a sequence of \( \ell \) MULTI\textsc{Push} and MULTI\textsc{Pull} operations that require \( \Omega(n\ell) \) time, where \( n \) is the maximum number of items in the queue at any time. Such a sequence implies that the amortized cost of each operation is \( \Omega(n) \).

(c) Describe a data structure that supports arbitrary intermixed sequences of MULTI\textsc{Push} and MULTI\textsc{Pull} operations in \( O(1) \) amortized cost per operation. Like a standard queue, your data structure should use only \( O(1) \) space per item.

7. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.
   - \textsc{Push}(x): Add item \( x \) to the end of the sequence.
   - \textsc{Pull}(): Remove and return the item at the beginning of the sequence.
   - \textsc{Size}(): Return the current number of items in the sequence.

   It is easy to implement a queue using a doubly-linked list, so that it uses \( O(n) \) space (where \( n \) is the number of items in the queue) and the worst-case time for each of these operations is \( O(1) \).

   Consider the following new operation, which removes every tenth element from the queue, starting at the beginning, in \( \Theta(n) \) worst-case time.

   \[
   \textsc{Decimate}():
   \]
   \[
   \begin{align*}
   &\text{\hspace{2cm} } n \leftarrow \textsc{Size}() \\
   &\text{\hspace{2cm} } \text{for } i \leftarrow 0 \text{ to } n - 1 \\
   &\hspace{4cm} \text{if } i \text{ mod } 10 = 0 \\
   &\hspace{6cm} \textsc{Pull}() \quad \langle \text{result discarded} \rangle \\
   &\hspace{4cm} \text{else} \\
   &\hspace{6cm} \textsc{Push}() \\
   \end{align*}
   \]

   Prove that in any intermixed sequence of \textsc{Push}, \textsc{Pull}, and \textsc{Decimate} operations, the amortized cost of each operation is \( O(1) \).

8. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array \( A[1..n] \) that stores the height of \( n \) buildings on a city block, indexed from west to east. Building \( i \) has a good view of Lake Michigan if and only if every building to the east of \( i \) is shorter than \( i \).

   Here is an algorithm that computes which buildings have a good view of Lake Michigan. What is the running time of this algorithm?

   \[
   \textsc{GoodView}(A[1..n]):
   \]
   \[
   \begin{align*}
   &\text{\hspace{2cm} } \text{initialize a stack } S \\
   &\text{\hspace{2cm} } \text{for } i \leftarrow 1 \text{ to } n \\
   &\hspace{4cm} \text{while } (S \text{ not empty and } A[i] > A[\text{Top}(S)]) \\
   &\hspace{6cm} \text{Pop}(S) \\
   &\hspace{4cm} \textsc{Push}(S, i) \\
   &\text{\hspace{2cm} } \text{return } S
   \end{align*}
   \]
9. Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. [Hint: Do not use potential functions.]

10. Professor Pisano insists that the size of any hash table used in his class must always be a Fibonacci number. He insists on the following variant of the previous global rebuilding strategy. Suppose the current hash table has size $F_k$.

- After an insertion, if the number of items in the table is $F_{k-1}$, we allocate a new hash table of size $F_{k+1}$, insert everything into the new table, and then free the old table.
- After a deletion, if the number of items in the table is $F_{k-3}$, we allocate a new hash table of size $F_{k-1}$, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. [Hint: Do not use potential functions.]

11. Remember the difference between stacks and queues? Good.

(a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The only access you have to the stacks is through the standard subroutines Push and Pop.

(b) A quack is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

- QUACKPush($x$): add a new item $x$ to the left end of the list;
- QUACKPop(): remove and return the item on the left end of the list;
- QUACKPull(): remove the item on the right end of the list.

Implement a quack using three stacks and $O(1)$ additional memory, so that the amortized time for any QUACKPush, QUACKPop, or QUACKPull operation is $O(1)$. In particular, each element in the quack must be stored in exactly one of the three stacks. Again, you cannot access the component stacks except through the interface functions Push and Pop.

12. Let’s glom a whole bunch of earlier problems together. Yay! An random-access double-ended multi-queue or radmuque (pronounced “rad muck”) stores a sequence of items and supports the following operations.

- MULTIpush($x$, $k$) adds $k$ copies of item $x$ to the beginning of the sequence.
• **MULTIPoke**(*x*, *k*) adds *k* copies of item *x* to the end of the sequence.
• **MULTIPop**(*k*) removes *k* items from the beginning of the sequence and returns the last item removed. (If there are less than *k* items in the sequence, remove them all and return NULL.)
• **MULTIPull**(*k*) removes *k* items from the end of the sequence and returns the last item removed. (If there are less than *k* items in the sequence, remove them all and return NULL.)
• **LOOKUP**(*k*) returns the *k*th item in the sequence. (If there are less than *k* items in the sequence, return NULL.)

Describe and analyze a simple data structure that supports these operations using \( O(n) \) space, where *n* is the current number of items in the sequence. **LOOKUP** should run in \( O(1) \) worst-case time; all other operations should run in \( O(1) \) amortized time.

13. Suppose you are faced with an infinite number of counters *x*<sub>*i*</sub>, one for each integer *i*. Each counter stores an integer mod *m*, where *m* is a fixed global constant. All counters are initially zero. The following operation increments a single counter *x*<sub>*i*</sub>; however, if *x*<sub>*i*</sub> overflows (that is, wraps around from *m* to 0), the adjacent counters *x*<sub>*i*−1</sub> and *x*<sub>*i*+1</sub> are incremented recursively.

\[
\text{NUDGE}_m(i): \\
x_i \leftarrow x_i + 1 \\
\text{while } x_i \geq m \\
x_i \leftarrow x_i - m \\
\text{NUDGE}_m(i - 1) \\
\text{NUDGE}_m(i + 1)
\]

(a) Prove that **NUDGE**<sub>3</sub> runs in \( O(1) \) amortized time. [Hint: Prove that **NUDGE**<sub>3</sub> always halts!]

(b) What is the worst-case total time for *n* calls to **NUDGE**<sub>2</sub>, if all counters are initially zero?

14. Now suppose you are faced with an infinite two-dimensional grid of modular counters, one counter *x*<sub>*i*,*j*</sub> for every pair of integers (*i*, *j*). Again, all counters are initially zero. The counters are modified by the following operation, where *m* is a fixed global constant:

\[
\text{2DNUDGE}_m(i, j): \\
x_{i,j} \leftarrow x_{i,j} + 1 \\
\text{while } x_{i,j} \geq m \\
x_{i,j} \leftarrow x_{i,j} - m \\
x_{i,j} \leftarrow x_{i,j} - m \\
\text{2DNUDGE}_m(i - 1, j) \\
\text{2DNUDGE}_m(i, j + 1) \\
\text{2DNUDGE}_m(i, j - 1) \\
\text{2DNUDGE}_m(i + 1, j)
\]

(a) Prove that **2DNUDGE**<sub>5</sub> runs in \( O(1) \) amortized time.

*(b) Prove or disprove: **2DNUDGE**<sub>4</sub> also runs in \( O(1) \) amortized time.*
15. Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of fits, where the $i$th least significant fit indicates whether the sum includes the $i$th Fibonacci number $F_i$. For example, the fitstring $101110_2$ represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a single fitstring in constant amortized time. [Hint: Most numbers can be represented by more than one fitstring!]

16. A doubly lazy binary counter represents any number as a weighted sum of powers of two, where each weight is one of four values: $-1, 0, 1, \text{ or } 2$. (For succinctness, I'll write $\dagger$ instead of $-1$.) Every integer—positive, negative, or zero—has an infinite number of doubly lazy binary representations. For example, the number $13$ can be represented as $1101_2$ (the standard binary representation), or $2\cdot01$ (because $2 \cdot 2^3 - 2^2 + 2^0 = 13$) or $10\cdot1\dagger$ (because $2^4 - 2^2 + 2^1 - 2^0 = 13$) or $\dagger1\dagger200010\dagger$ (because $-2^{10} + 2^9 + 2\cdot2^8 + 2^4 - 2^2 + 2^1 - 2^0 = 13$).

To increment a doubly lazy binary counter, we add 1 to the least significant bit, then carry the rightmost 2 (if any). To decrement, we subtract 1 from the least significant bit, and then borrow the rightmost $\dagger$ (if any).

<table>
<thead>
<tr>
<th>LazyIncrement($B[0..n]$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B[0] \leftarrow B[0] + 1$</td>
</tr>
<tr>
<td>for $i \leftarrow 1$ to $n - 1$</td>
</tr>
<tr>
<td>if $B[i] = 2$</td>
</tr>
<tr>
<td>$B[i] \leftarrow 0$</td>
</tr>
<tr>
<td>$B[i + 1] \leftarrow B[i + 1] + 1$</td>
</tr>
<tr>
<td>return</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LazyDecrement($B[0..n]$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B[0] \leftarrow B[0] - 1$</td>
</tr>
<tr>
<td>for $i \leftarrow 1$ to $n - 1$</td>
</tr>
<tr>
<td>if $B[i] = -1$</td>
</tr>
<tr>
<td>$B[i] \leftarrow 1$</td>
</tr>
<tr>
<td>$B[i + 1] \leftarrow B[i + 1] - 1$</td>
</tr>
<tr>
<td>return</td>
</tr>
</tbody>
</table>

For example, here is a doubly lazy binary count from zero up to twenty and then back down to zero. The bits are written with the least significant bit $B[0]$ on the right, omitting all leading 0’s.

$$0 \dagger\dagger 1 \dagger\dagger 10 \dagger\dagger 11 \dagger\dagger 20 \dagger\dagger 101 \dagger\dagger 110 \dagger\dagger 111 \dagger\dagger 120 \dagger\dagger 201 \dagger\dagger 210$$

$$\dagger\dagger 1011 \dagger\dagger 1020 \dagger\dagger 1101 \dagger\dagger 1110 \dagger\dagger 1111 \dagger\dagger 1120 \dagger\dagger 1201 \dagger\dagger 1210 \dagger\dagger 2011 \dagger\dagger 2020$$

$$\rightarrow 2011 \rightarrow 2010 \rightarrow 2001 \rightarrow 2000 \rightarrow 2\dagger01 \rightarrow 2\dagger0 \rightarrow 2\dagger0\dagger \rightarrow 2\dagger01 \rightarrow 1100 \rightarrow 11\dagger1 \rightarrow 1010$$

$$\rightarrow 1001 \rightarrow 1000 \rightarrow 10\dagger0 \rightarrow 1\dagger10 \rightarrow 1\dagger01 \rightarrow 100 \rightarrow 1\dagger1 \rightarrow 10 \rightarrow 1 \rightarrow 0$$

Prove that for any intermixed sequence of increments and decrements of a doubly lazy binary number, starting with 0, the amortized time for each operation is $O(1)$. Do not assume, as in the example above, that all the increments come before all the decrements.
Everything was balanced before the computers went off line. Try and adjust something, and you unbalance something else. Try and adjust that, you unbalance two more and before you know what's happened, the ship is out of control.

— Blake, Blake's 7, “Breakdown” (March 6, 1978)

A good scapegoat is nearly as welcome as a solution to the problem.

— Anonymous

Let’s play.

— El Mariachi [Antonio Banderas], Desperado (1992)

CAPTAIN: TAKE OFF EVERY 'ZIG'!!
CAPTAIN: YOU KNOW WHAT YOU DOING.
CAPTAIN: MOVE 'ZIG'.
CAPTAIN: FOR GREAT JUSTICE.

— Zero Wing (1992)

## 16 Scapegoat and Splay Trees

### 16.1 Definitions

I’ll assume that everyone is already familiar with the standard terminology for binary search trees—node, search key, edge, root, internal node, leaf, right child, left child, parent, descendant, sibling, ancestor, subtree, preorder, postorder, inorder, etc.—as well as the standard algorithms for searching for a node, inserting a node, or deleting a node. Otherwise, consult your favorite data structures textbook.

For this lecture, we will consider only **full** binary trees—where every internal node has exactly two children—where only the **internal** nodes actually store search keys. In practice, we can represent the leaves with null pointers.

Recall that the **depth** of a node is its distance from the root, and its **height** is the distance to the farthest leaf in its subtree. The height (or depth) of the tree is just the height of the root. The **size** of a node is the number of nodes in its subtree. The size $n$ of the whole tree is just the total number of nodes.

A tree with height $h$ has at most $2^h$ leaves, so the minimum height of an $n$-leaf binary tree is $\lceil \log n \rceil$. In the worst case, the time required for a search, insertion, or deletion to the height of the tree, so in general we would like keep the height as close to $\log n$ as possible. The best we can possibly do is to have a *perfectly balanced* tree, in which each subtree has as close to half the leaves as possible, and both subtrees are perfectly balanced. The height of a perfectly balanced tree is $\lceil \log n \rceil$, so the worst-case search time is $O(\log n)$. However, even if we started with a perfectly balanced tree, a malicious sequence of insertions and/or deletions could make the tree arbitrarily unbalanced, driving the search time up to $\Theta(n)$.

To avoid this problem, we need to periodically modify the tree to maintain ‘balance’. There are several methods for doing this, and depending on the method we use, the search tree is given a different name. Examples include AVL trees, red-black trees, height-balanced trees, weight-balanced trees, bounded-balance trees, path-balanced trees, $B$-trees, treaps, randomized
binary search trees, skip lists, and jumplists. Some of these trees support searches, insertions, and deletions, in \(O(\log n)\) worst-case time, others in \(O(\log n)\) amortized time, still others in \(O(\log n)\) expected time.

In this lecture, I’ll discuss three binary search tree data structures with good amortized performance. The first two are variants of lazy balanced trees: lazy weight-balanced trees, developed by Mark Overmars* in the early 1980s, [14] and scapegoat trees, discovered by Arne Andersson* in 1989 [1, 2] and independently2 by Igal Galperin* and Ron Rivest in 1993 [11]. The third structure is the splay tree, discovered by Danny Sleator and Bob Tarjan in 1981 [19, 16].

16.2 Lazy Deletions: Global Rebuilding

First let’s consider the simple case where we start with a perfectly-balanced tree, and we only want to perform searches and deletions. To get good search and delete times, we can use a technique called global rebuilding. When we get a delete request, we locate and mark the node to be deleted, but we don’t actually delete it. This requires a simple modification to our search algorithm—we still use marked nodes to guide searches, but if we search for a marked node, the search routine says it isn’t there. This keeps the tree more or less balanced, but now the search time is no longer a function of the amount of data currently stored in the tree. To remedy this, we also keep track of how many nodes have been marked, and then apply the following rule:

Global Rebuilding Rule. As soon as half the nodes in the tree have been marked, rebuild a new perfectly balanced tree containing only the unmarked nodes.³

With this rule in place, a search takes \(O(\log n)\) time in the worst case, where \(n\) is the number of unmarked nodes. Specifically, since the tree has at most \(n\) marked nodes, or \(2n\) nodes altogether, we need to examine at most \(\lceil \log n \rceil + 1\) keys. There are several methods for rebuilding the tree in \(O(n)\) time, where \(n\) is the size of the new tree. (Homework!) So a single deletion can cost \(\Theta(n)\) time in the worst case, but only if we have to rebuild; most deletions take only \(O(\log n)\) time.

We spend \(O(n)\) time rebuilding, but only after \(\Omega(n)\) deletions, so the amortized cost of rebuilding the tree is \(O(1)\) per deletion. (Here I’m using a simple version of the ‘taxation method’. For each deletion, we charge a $1 tax; after \(n\) deletions, we’ve collected $n, which is just enough to pay for rebalancing the tree containing the remaining \(n\) nodes.) Since we also have to find and mark the node being ‘deleted’, the total amortized time for a deletion is \(O(\log n)\).

16.3 Insertions: Partial Rebuilding

Now suppose we only want to support searches and insertions. We can’t ‘not really insert’ new nodes into the tree, since that would make them unavailable to the search algorithm.⁴ So instead, we’ll use another method called partial rebuilding. We will insert new nodes normally, but whenever a subtree becomes unbalanced enough, we rebuild it. The definition of ‘unbalanced enough’ depends on an arbitrary constant \(\alpha > 1\).

Each node \(v\) will now also store \(\text{height}(v)\) and \(\text{size}(v)\). We now modify our insertion algorithm with the following rule:

---

1Yeah, yeah. Skip lists aren’t really binary search trees. Whatever you say, Mr. Picky.
2The claim of independence is Andersson’s [2]. The two papers actually describe very slightly different rebalancing algorithms. The algorithm I’m using here is closer to Andersson’s, but my analysis is closer to Galperin and Rivest’s.
3Alternatively: When the number of unmarked nodes is one less than an exact power of two, rebuild the tree. This rule ensures that the tree is always exactly balanced.
4Well, we could use the Bentley-Saxe logarithmic method [3], but that would raise the query time to \(O(\log^2 n)\).
Partial Rebuilding Rule. After we insert a node, walk back up the tree updating the heights and sizes of the nodes on the search path. If we encounter a node \( v \) where \( \text{height}(v) > \alpha \cdot \lg(\text{size}(v)) \), rebuild its subtree into a perfectly balanced tree (in \( O(\text{size}(v)) \) time).

If we always follow this rule, then after an insertion, the height of the tree is at most \( \alpha \cdot \lg n \). Thus, since \( \alpha \) is a constant, the worst-case search time is \( O(\log n) \). In the worst case, insertions require \( \Theta(n) \) time—we might have to rebuild the entire tree. However, the amortized time for each insertion is again only \( O(\log n) \). Not surprisingly, the proof is a little bit more complicated than for deletions.

Define the imbalance \( \text{Imbal}(v) \) of a node \( v \) to be the absolute difference between the sizes of its two subtrees:

\[
\text{Imbal}(v) := |\text{size}(\text{left}(v)) - \text{size}(\text{right}(v))|
\]

A simple induction proof implies that \( \text{Imbal}(v) \leq 1 \) for every node \( v \) in a perfectly balanced tree. In particular, immediately after we rebuild the subtree of \( v \), we have \( \text{Imbal}(v) \leq 1 \). On the other hand, each insertion into the subtree of \( v \) increments either \( \text{size}(\text{left}(v)) \) or \( \text{size}(\text{right}(v)) \), so \( \text{Imbal}(v) \) changes by at most 1.

The whole analysis boils down to the following lemma.

**Lemma 1.** Just before we rebuild \( v \)'s subtree, \( \text{Imbal}(v) = \Omega(\text{size}(v)) \).

Before we prove this lemma, let’s first look at what it implies. If \( \text{Imbal}(v) = \Omega(\text{size}(v)) \), then \( \Omega(\text{size}(v)) \) keys have been inserted in the \( v \)'s subtree since the last time it was rebuilt from scratch. On the other hand, rebuilding the subtree requires \( O(\text{size}(v)) \) time. Thus, if we amortize the rebuilding cost across all the insertions since the previous rebuild, \( v \) is charged constant time for each insertion into its subtree. Since each new key is inserted into at most \( \alpha \cdot \lg n = O(\log n) \) subtrees, the total amortized cost of an insertion is \( O(\log n) \).

**Proof:** Since we’re about to rebuild the subtree at \( v \), we must have \( \text{height}(v) > \alpha \cdot \lg \text{size}(v) \). Without loss of generality, suppose that the node we just inserted went into \( v \)'s left subtree. Either we just rebuilt this subtree or we didn’t have to, so we also have \( \text{height}(\text{left}(v)) \leq \alpha \cdot \lg \text{size}(\text{left}(v)) \).

Combining these two inequalities with the recursive definition of height, we get

\[
\alpha \cdot \lg \text{size}(v) < \text{height}(v) \leq \text{height}(\text{left}(v)) + 1 \leq \alpha \cdot \lg \text{size}(\text{left}(v)) + 1.
\]

After some algebra, this simplifies to \( \text{size}(\text{left}(v)) > \text{size}(v) / 2^{1/\alpha} \). Combining this with the identity \( \text{size}(v) = \text{size}(\text{left}(v)) + \text{size}(\text{right}(v)) + 1 \) and doing some more algebra gives us the inequality

\[
\text{size}(\text{right}(v)) < (1 - 1/2^{1/\alpha}) \text{size}(v) - 1.
\]

Finally, we combine these two inequalities using the recursive definition of imbalance.

\[
\text{Imbal}(v) \geq \text{size}(\text{left}(v)) - \text{size}(\text{right}(v)) - 1 > (2 / 2^{1/\alpha} - 1) \text{size}(v)
\]

Since \( \alpha \) is a constant bigger than 1, the factor in parentheses is a positive constant. \( \square \)
16.4 Scapegoat (Lazy Height-Balanced) Trees

Finally, to handle both insertions and deletions efficiently, scapegoat trees use both of the previous techniques. We use partial rebuilding to re-balance the tree after insertions, and global rebuilding to re-balance the tree after deletions. Each search takes $O(\log n)$ time in the worst case, and the amortized time for any insertion or deletion is also $O(\log n)$. There are a few small technical details left (which I won't describe), but no new ideas are required.

Once we've done the analysis, we can actually simplify the data structure. It's not hard to prove that at most one subtree (the scapegoat) is rebuilt during any insertion. Less obviously, we can even get the same amortized time bounds (except for a small constant factor) if we only maintain the three integers in addition to the actual tree: the size of the entire tree, the height of the entire tree, and the number of marked nodes. Whenever an insertion causes the tree to become unbalanced, we can compute the sizes of all the subtrees on the search path, starting at the new leaf and stopping at the scapegoat, in time proportional to the size of the scapegoat subtree. Since we need that much time to re-balance the scapegoat subtree, this computation increases the running time by only a small constant factor! Thus, unlike almost every other kind of balanced trees, scapegoat trees require only $O(1)$ extra space.

16.5 Rotations, Double Rotations, and Splaying

Another method for maintaining balance in binary search trees is by adjusting the shape of the tree locally, using an operation called a rotation. A rotation at a node $x$ decreases its depth by one and increases its parent's depth by one. Rotations can be performed in constant time, since they only involve simple pointer manipulation.

For technical reasons, we will need to use rotations two at a time. There are two types of double rotations, which might be called zig-zag and roller-coaster. A zig-zag at $x$ consists of two rotations at $x$, in opposite directions. A roller-coaster at a node $x$ consists of a rotation at $x$'s parent followed by a rotation at $x$, both in the same direction. Each double rotation decreases the depth of $x$ by two, leaves the depth of its parent unchanged, and increases the depth of its grandparent by either one or two, depending on the type of double rotation. Either type of double rotation can be performed in constant time.

Finally, a splay operation moves an arbitrary node in the tree up to the root through a series of double rotations, possibly with one single rotation at the end. Splaying a node $v$ requires time proportional to $\text{depth}(v)$. (Obviously, this means the depth before splaying, since after splaying $v$ is the root and thus has depth zero!)

16.6 Splay Trees

A splay tree is a binary search tree that is kept more or less balanced by splaying. Intuitively, after we access any node, we move it to the root with a splay operation. In more detail:
• **Search**: Find the node containing the key using the usual algorithm, or its predecessor or successor if the key is not present. Splay whichever node was found.

• **Insert**: Insert a new node using the usual algorithm, then splay that node.

• **Delete**: Find the node \( x \) to be deleted, splay it, and then delete it. This splits the tree into two subtrees, one with keys less than \( x \), the other with keys bigger than \( x \). Find the node \( w \) in the left subtree with the largest key (the inorder predecessor of \( x \) in the original tree), splay it, and finally join it to the right subtree.

Each search, insertion, or deletion consists of a constant number of operations of the form walk down to a node, and then splay it up to the root. Since the walk down is clearly cheaper
than the splay up, all we need to get good amortized bounds for splay trees is to derive good amortized bounds for a single splay.

Believe it or not, the easiest way to do this uses the potential method. We define the rank of a node \( v \) to be \( \lfloor \lg \text{size}(v) \rfloor \), and the potential of a splay tree to be the sum of the ranks of its nodes:

\[
\Phi := \sum_v \text{rank}(v) = \sum_v \lfloor \lg \text{size}(v) \rfloor
\]

It’s not hard to observe that a perfectly balanced binary tree has potential \( \Theta(n) \), and a linear chain of nodes (a perfectly unbalanced tree) has potential \( \Theta(n \log n) \).

The amortized analysis of splay trees boils down to the following lemma. Here, \( \text{rank}(v) \) denotes the rank of \( v \) before a (single or double) rotation, and \( \text{rank}'(v) \) denotes its rank afterwards.

**The Access Lemma.** The amortized cost of a single rotation at any node \( v \) is at most \( 1 + 3 \text{rank}'(v) - 3 \text{rank}(v) \), and the amortized cost of a double rotation at any node \( v \) is at most \( 3 \text{rank}'(v) - 3 \text{rank}(v) \).

Proving this lemma is a straightforward but tedious case analysis of the different types of rotations. For the sake of completeness, I’ll give a proof (of a generalized version) in the next section.

By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node \( v \) is at most \( 1 + 3 \text{rank}'(v) - 3 \text{rank}(v) \), where \( \text{rank}'(v) \) is the rank of \( v \) after the entire splay. (The intermediate ranks cancel out in a nice telescoping sum.) But after the splay, \( v \) is the root! Thus, \( \text{rank}'(v) = \lfloor \lg n \rfloor \), which implies that the amortized cost of a splay is at most \( 3 \lg n - 1 = O(\log n) \).

We conclude that every insertion, deletion, or search in a splay tree takes \( O(\log n) \) amortized time.

**16.7 Other Optimality Properties**

In fact, splay trees are optimal in several other senses. Some of these optimality properties follow easily from the following generalization of the Access Lemma.

Let’s arbitrarily assign each node \( v \) a non-negative real weight \( w(v) \). These weights are not actually stored in the splay tree, nor do they affect the splay algorithm in any way; they are only used to help with the analysis. We then redefine the size \( s(v) \) of a node \( v \) to be the sum of the weights of the descendants of \( v \), including \( v \) itself:

\[
s(v) := w(v) + s(\text{right}(v)) + s(\text{left}(v)).
\]

If \( w(v) = 1 \) for every node \( v \), then the size of a node is just the number of nodes in its subtree, as in the previous section. As before, we define the rank of any node \( v \) to be \( r(v) = \lg s(v) \), and the potential of a splay tree to be the sum of the ranks of all its nodes:

\[
\Phi = \sum_v r(v) = \sum_v \lg s(v)
\]

In the following lemma, \( r(v) \) denotes the rank of \( v \) before a (single or double) rotation, and \( r'(v) \) denotes its rank afterwards.
The Generalized Access Lemma. For any assignment of non-negative weights to the nodes, the amortized cost of a single rotation at any node $x$ is at most $1 + 3r'(x) - 3r(x)$, and the amortized cost of a double rotation at any node $v$ is at most $3r'(x) - 3r(x)$.

Proof: First consider a single rotation, as shown in Figure 1.

$$1 + \Phi' - \Phi = 1 + r'(x) + r'(y) - r(x) - r(y) \quad \text{[only } x \text{ and } y \text{ change rank]}$$

$$\leq 1 + r'(x) - r(x) \quad \text{[} r'(y) \leq r(y) \text{]}$$

$$\leq 1 + 3r'(x) - 3r(x) \quad \text{[} r'(x) \geq r(x) \text{]}$$

Now consider a zig-zag, as shown in Figure 2. Only $w$, $x$, and $z$ change rank.

$$2 + \Phi' - \Phi$$

$$= 2 + r'(w) + r'(x) + r'(z) - r(w) - r(x) - r(z) \quad \text{[only } w, x, z \text{ change rank]}$$

$$\leq 2 + r'(w) + r'(x) + r'(z) - 2r(x) \quad \text{[} r(x) \leq r(w) \text{ and } r'(x) = r(z) \text{]}$$

$$= 2 + (r'(w) - r'(x)) + (r'(z) - r'(x)) + 2(r'(x) - r(x))$$

$$= 2 + \lg \frac{s'(w)}{s'(x)} + \lg \frac{s'(z)}{s'(x)} + 2(r'(x) - r(x))$$

$$\leq 2 + 2\lg \frac{s'(x)/2}{s'(x)} + 2(r'(x) - r(x)) \quad \text{[} s'(w) + s'(z) \leq s'(x), \lg \text{ is concave} \text{]}$$

$$= 2(r'(x) - r(x))$$

$$\leq 3(r'(x) - r(x)) \quad \text{[} r'(x) \geq r(x) \text{]}$$

Finally, consider a roller-coaster, as shown in Figure 3. Only $x$, $y$, and $z$ change rank.

$$2 + \Phi' - \Phi$$

$$= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \quad \text{[only } x, y, z \text{ change rank]}$$

$$\leq 2 + r'(x) + r'(z) - 2r(x) \quad \text{[} r'(y) \leq r(z) \text{ and } r(x) \geq r(y) \text{]}$$

$$= 2 + (r(x) - r'(x)) + (r'(z) - r'(x)) + 3(r'(x) - r(x))$$

$$= 2 + \lg \frac{s(x)}{s'(x)} + \lg \frac{s'(z)}{s'(x)} + 3(r'(x) - r(x))$$

$$\leq 2 + 2\lg \frac{s'(x)/2}{s'(x)} + 3(r'(x) - r(x)) \quad \text{[} s(x) + s'(z) \leq s'(x), \lg \text{ is concave} \text{]}$$

$$= 3(r'(x) - r(x))$$

This completes the proof. \(\Box\)

Observe that this argument works for arbitrary non-negative vertex weights. By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node $x$ is at most $1 + 3r(root) - 3r(x)$. (The intermediate ranks cancel out in a nice telescoping sum.)

This analysis has several immediate corollaries. The first corollary is that the amortized search time in a splay tree is within a constant factor of the search time in the best possible static

\textsuperscript{5}This proof is essentially taken verbatim from the original Sleator and Tarjan paper. Another proof technique, which may be more accessible, involves maintaining $\lceil \lg s(v) \rceil$ tokens on each node $v$ and arguing about the changes in token distribution caused by each single or double rotation. But I haven’t yet internalized this approach enough to include it here.
binary search tree. Thus, if some nodes are accessed more often than others, the standard splay algorithm automatically keeps those more frequent nodes closer to the root, at least most of the time.

**Static Optimality Theorem.** Suppose each node \( x \) is accessed at least \( t(x) \) times, and let \( T = \sum_x t(x) \). The amortized cost of accessing \( x \) is \( O(\log T - \log t(x)) \).

**Proof:** Set \( w(x) = t(x) \) for each node \( x \).

For any nodes \( x \) and \( z \), let \( \text{dist}(x,z) \) denote the rank distance between \( x \) and \( y \), that is, the number of nodes \( y \) such that \( \text{key}(x) \leq \text{key}(y) \leq \text{key}(z) \) or \( \text{key}(x) \geq \text{key}(y) \geq \text{key}(z) \). In particular, \( \text{dist}(x,x) = 1 \) for all \( x \).

**Static Finger Theorem.** For any fixed node \( f \) (the finger), the amortized cost of accessing \( x \) is \( O(\log \text{dist}(f,x)) \).

**Proof:** Set \( w(x) = 1/\text{dist}(x,f)^2 \) for each node \( x \). Then \( s(\text{root}) \leq \sum_{i=1}^{\infty} 2/i^2 = \pi^2/3 = O(1) \), and \( r(x) \geq \log w(x) = -2\log \text{dist}(f,x) \).

Here are a few more interesting properties of splay trees, which I’ll state without proof.\(^6\) The proofs of these properties (especially the dynamic finger theorem) are considerably more complicated than the amortized analysis presented above.

**Working Set Theorem** \([\text{16}].\) The amortized cost of accessing node \( x \) is \( O(\log D) \), where \( D \) is the number of distinct items accessed since the last time \( x \) was accessed. (For the first access to \( x \), we set \( D = n \).)

**Scanning Theorem** \([\text{18}].\) Splaying all nodes in a splay tree in order, starting from any initial tree, requires \( O(n) \) total rotations.

**Dynamic Finger Theorem** \([\text{7, 6}].\) Immediately after accessing node \( y \), the amortized cost of accessing node \( x \) is \( O(\log \text{dist}(x,y)) \).

### 16.8 Splay Tree Conjectures

Splay trees are conjectured to have many interesting properties in addition to the optimality properties that have been proved; I’ll describe just a few of the more important ones.

The **Deque Conjecture** \([\text{18}].\) considers the cost of dynamically maintaining two fingers \( l \) and \( r \), starting on the left and right ends of the tree. Suppose at each step, we can move one of these two fingers either one step left or one step right; in other words, we are using the splay tree as a doubly-ended queue. Sundar\(^*\) proved that the total cost of \( m \) deque operations on an \( n \)-node splay tree is \( O((m+n)\alpha(m+n)) \) \([\text{17}].\) More recently, Pettie later improved this bound to \( O(m\alpha^\delta(n)) \) \([\text{15}].\) The Deque Conjecture states that the total cost is actually \( O(m+n) \).

The **Traversal Conjecture** \([\text{16}].\) states that accessing the nodes in a splay tree, in the order specified by a preorder traversal of any other binary tree with the same keys, takes \( O(n) \) time. This is generalization of the Scanning Theorem.

The **Unified Conjecture** \([\text{13}].\) states that the time to access node \( x \) is \( O(\log \min_y (D(y) + d(x,y))) \), where \( D(y) \) is the number of distinct nodes accessed since the last time \( y \) was accessed. This

\( ^6\)This list and the following section are taken almost directly from Erik Demaine’s lecture notes \([\text{5}].\)
would immediately imply both the Dynamic Finger Theorem, which is about spatial locality, and the Working Set Theorem, which is about temporal locality. Two other structures are known that satisfy the unified bound \[4, 13\].

Finally, the most important conjecture about splay trees, and one of the most important open problems about data structures, is that they are dynamically optimal \[16\]. Specifically, the cost of any sequence of accesses to a splay tree is conjectured to be at most a constant factor more than the cost of the best possible dynamic binary search tree that knows the entire access sequence in advance. To make the rules concrete, we consider binary search trees that can undergo arbitrary rotations after a search; the cost of a search is the number of key comparisons plus the number of rotations. We do not require that the rotations be on or even near the search path. This is an extremely strong conjecture!

No dynamically optimal binary search tree is known, even in the offline setting. However, three very similar \(O(\log \log n)\)-competitive binary search trees have been discovered in the last few years: Tango trees \[9\], multisplay trees \[20\], and chain-splay trees \[12\]. A recently-published geometric formulation of dynamic binary search trees \[8, 10\] also offers significant hope for future progress.

References


*Starred authors were graduate students at the time that the cited work was published. **Double-starred authors were undergraduates.

### Exercises

1. (a) An *n*-node binary tree is **perfectly balanced** if either *n* ≤ 1, or its two subtrees are perfectly balanced binary trees, each with at most \( \lfloor n/2 \rfloor \) nodes. Prove that \( I(v) \leq 1 \) for every node \( v \) of any perfectly balanced tree.

   (b) Prove that at most one subtree is rebalanced during a scapegoat tree insertion.

2. In a **dirty** binary search tree, each node is labeled either **clean** or **dirty**. The lazy deletion scheme used for scapegoat trees requires us to **purge** the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

   (a) Describe and analyze an algorithm to purge an arbitrary *n*-node dirty binary search tree in \( O(n) \) time. (Such an algorithm is necessary for scapegoat trees to achieve \( O(\log n) \) amortized insertion cost.)

   * (b) Modify your algorithm so that is uses only \( O(\log n) \) space, in addition to the tree itself. Don't forget to include the recursion stack in your space bound.

   ** (c) Modify your algorithm so that is uses only \( O(1) \) additional space. In particular, your algorithm cannot call itself recursively at all.

3. Consider the following simpler alternative to splaying:

   \[
   \text{MoveToRoot}(v): \quad \begin{cases} 
   \text{while } \text{parent}(v) \neq \text{Null} \\
   \text{rotate at } v
   \end{cases}
   \]

   Prove that the amortized cost of *MoveToRoot* in an *n*-node binary tree can be \( \Omega(n) \). That is, prove that for any integer \( k \), there is a sequence of \( k \) *MoveToRoot* operations that require \( \Omega(kn) \) time to execute.

4. Let \( P \) be a set of *n* points in the plane. The **staircase** of \( P \) is the set of all points in the plane that have at least one point in \( P \) both above and to the right.
(a) Describe an algorithm to compute the staircase of a set of \( n \) points in \( O(n \log n) \) time.

(b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm \textsc{Above}(x, y)\footnote{This is a hypothetical function.} that returns \textsc{true} if the point \((x, y)\) is above the staircase, or \textsc{false} otherwise. Your data structure should use \( O(n) \) space, and your \textsc{Above} algorithm should run in \( O(\log n) \) time.

(c) Describe and analyze a data structure that maintains a staircase as new points are inserted. Specifically, your data structure should support a function \textsc{Insert}(x, y)\footnote{This is a hypothetical function.} that adds the point \((x, y)\) to the underlying point set and returns \textsc{true} or \textsc{false} to indicate whether the staircase of the set has changed. Your data structure should use \( O(n) \) space, and your \textsc{Insert} algorithm should run in \( O(\log n) \) amortized time.

5. Suppose we want to maintain a dynamic set of values, subject to the following operations:

- \textsc{Insert}(x): Add \( x \) to the set (if it isn't already there).
- \textsc{PrintAndDeleteBetween}(a, b): Print every element \( x \) in the range \( a \leq x \leq b \), in increasing order, and delete those elements from the set.

For example, if the current set is \( \{1, 5, 3, 4, 8\} \), then

- \textsc{PrintAndDeleteBetween}(4, 6) prints the numbers 4 and 5 and changes the set to \( \{1, 3, 8\} \);
- \textsc{PrintAndDeleteBetween}(6, 7) prints nothing and does not change the set;
- \textsc{PrintAndDeleteBetween}(0, 10) prints the sequence 1, 3, 4, 5, 8 and deletes everything.
(a) Suppose we store the set in our favorite balanced binary search tree, using the standard **insert** algorithm and the following algorithm for **print** **& delete** **between**:

```
**print** & **delete** between (**a**, **b**):
  **x** ← **successor**(**a**)
  while (**x** ≤ **b**)
    print (**x**)
    **delete**(**x**)
  **x** ← **successor**(**a**)
```

Here, **successor**(**a**) returns the smallest element greater than or equal to **a** (or ∞ if there is no such element), and **delete** is the standard deletion algorithm. Prove that the amortized time for **insert** and **print** **& delete** **between** is \(O(\log N)\), where \(N\) is the maximum number of items that are ever stored in the tree.

(b) Describe and analyze **insert** and **print** **& delete** **between** algorithms that run in \(O(\log n)\) amortized time, where \(n\) is the current number of elements in the set.

(c) What is the running time of your **insert** algorithm in the worst case?

(d) What is the running time of your **print** **& delete** **between** algorithm in the worst case?

6. Say that a binary search tree is **augmented** if every node \(v\) also stores \(size(v)\), the number of nodes in the subtree rooted at \(v\).

   (a) Show that a rotation in an augmented binary tree can be performed in constant time.

   (b) Describe an algorithm **scapegoat select**(**k**) that selects the \(k\)th smallest item in an augmented scapegoat tree in \(O(\log n)\) worst-case time. (The scapegoat trees presented in these notes are already augmented.)

   (c) Describe an algorithm **splay select**(**k**) that selects the \(k\)th smallest item in an augmented splay tree in \(O(\log n)\) amortized time.

   (d) Describe an algorithm **treap select**(**k**) that selects the \(k\)th smallest item in an augmented treap in \(O(\log n)\) expected time.

7. Many applications of binary search trees attach a secondary data structure to each node in the tree, to allow for more complicated searches. Let \(T\) be an arbitrary binary tree. The secondary data structure at any node \(v\) stores exactly the same set of items as the subtree of \(T\) rooted at \(v\). This secondary structure has size \(O(size(v))\) and can be built in \(O(size(v))\) time, where \(size(v)\) denotes the number of descendants of \(v\).

   The primary and secondary data structures are typically defined by different attributes of the data being stored. For example, to store a set of points in the plane, we could define the primary tree \(T\) in terms of the \(x\)-coordinates of the points, and define the secondary data structures in terms of their \(y\)-coordinate.

   Maintaining these secondary structures complicates algorithms for keeping the top-level search tree balanced. Specifically, performing a rotation at any node \(v\) in the primary tree now requires \(O(size(v))\) time, because we have to rebuild one of the secondary structures (at the new child of \(v\)). When we insert a new item into \(T\), we must also insert into one or more secondary data structures.
(a) Overall, how much space does this data structure use in the worst case?
(b) How much space does this structure use if the primary search tree is perfectly balanced?
(c) Suppose the primary tree is a splay tree. Prove that the amortized cost of a splay (and therefore of a search, insertion, or deletion) is \(\Omega(n)\). [Hint: This is easy!]
(d) Now suppose the primary tree \(T\) is a scapegoat tree. How long does it take to rebuild the subtree of \(T\) rooted at some node \(v\), as a function of \(\text{size}(v)\)?
(e) Suppose the primary tree and all secondary trees are scapegoat trees. What is the amortized cost of a single insertion?
(f) Finally, suppose the primary tree and every secondary tree is a treap. What is the worst-case expected time for a single insertion?

8. Suppose we want to maintain a collection of strings (sequences of characters) under the following operations:

- \texttt{NEWSTRING}(a) creates a new string of length 1 containing only the character \(a\) and returns a pointer to that string.
- \texttt{CONCAT}(S, T) removes the strings \(S\) and \(T\) (given by pointers) from the data structure, adds the concatenated string \(ST\) to the data structure, and returns a pointer to the new string.
- \texttt{SPLIT}(S, k) removes the strings \(S\) (given by a pointer) from the data structure, adds the first \(k\) characters of \(S\) and the rest of \(S\) as two new strings in the data structure, and returns pointers to the two new strings.
- \texttt{REVERSE}(S) removes the string \(S\) (given by a pointer) from the data structure, adds the reversal of \(S\) to the data structure, and returns a pointer to the new string.
- \texttt{LOOKUP}(S, k) returns the \(k\)th character in string \(S\) (given by a pointer), or \texttt{NULL} if the length of the \(S\) is less than \(k\).

Describe and analyze a simple data structure that supports \texttt{NEWSTRING} and \texttt{REVERSE} in \(O(1)\) worst-case time, supports every other operation in \(O(\log n)\) time (either worst-case, expected, or amortized), and uses \(O(n)\) space, where \(n\) is the sum of the current string lengths. [Hint: Why is this problem here?]

9. After the Great Academic Meltdown of 2020, you get a job as a cook's assistant at Jumpin' Jack's Flapjack Stack Shack, which sells arbitrarily-large stacks of pancakes for just four bits (50 cents) each. Jumpin' Jack insists that any stack of pancakes given to one of his customers must be sorted, with smaller pancakes on top of larger pancakes. Also, whenever a pancake goes to a customer, at least the top side must not be burned.

The cook provides you with an unsorted stack of \(n\) perfectly round pancakes, of \(n\) different sizes, possibly burned on one or both sides. Your task is to throw out the pancakes that are burned on both sides (and only those) and sort the remaining pancakes so that their burned sides (if any) face down. Your only tool is a spatula. You can insert the spatula under any pancake and then either \texttt{flip} or \texttt{discard} the stack of pancakes above the spatula.

More concretely, we can represent a stack of pancakes by a sequence of distinct integers between 1 and \(n\), representing the sizes of the pancakes, with each number marked to
indicate the burned side(s) of the corresponding pancake. For example, $1 \underline{4} 3 \underline{2}$ represents a stack of four pancakes: a one-inch pancake burned on the bottom; a four-inch pancake burned on the top; an unburned three-inch pancake, and a two-inch pancake burned on both sides. We store this sequence in a data structure that supports the following operations:

- **POSITION(x):** Return the position of integer $x$ in the current sequence, or 0 if $x$ is not in the sequence.
- **VALUE(k):** Return the $k$th integer in the current sequence, or 0 if the sequence has no $k$th element. **VALUE** is essentially the inverse of **POSITION**.
- **TopBURNED(k):** Return True if and only if the top side of the $k$th pancake in the current sequence is burned.
- **FLIP(k):** Reverse the order and the burn marks of the first $k$ elements of the sequence.
- **DISCARD(k):** Discard the first $k$ elements of the sequence.

(a) Describe an algorithm to filter and sort any stack of $n$ burned pancakes using $O(n)$ of the operations listed above. Try to make the big-Oh constant small.

(b) Describe a data structure that supports each of the operations listed above in $O(\log n)$ amortized time. Together with part (a), such a data structure gives us an algorithm to filter and sort any stack of $n$ burned pancakes in $O(n \log n)$ time.

10. Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ be a sequence of $m$ integers, each from the set $\{1, 2, \ldots, n\}$. We can visualize this sequence as a set of integer points in the plane, by interpreting each element $x_i$ as the point $(x_i, i)$. The resulting point set, which we can also call $X$, has exactly one point on each row of the $n \times m$ integer grid.

(a) Let $Y$ be an arbitrary set of integer points in the plane. Two points $(x_1, y_1)$ and $(x_2, y_2)$ in $Y$ are isolated if (1) $x_1 \neq x_2$ and $y_1 \neq y_2$, and (2) there is no other point $(x, y) \in Y$ with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. If the set $Y$ contains no isolated pairs of points, we call $Y$ a commune.$^7$

Let $X$ be an arbitrary set of points on the $n \times n$ integer grid with exactly one point per row. Show that there is a commune $Y$ that contains $X$ and consists of $O(n \log n)$ points.

$^7$Demaine et al. [8] refer to communes as arborally satisfied sets.
(b) Consider the following model of self-adjusting binary search trees. We interpret \( X \) as a sequence of accesses in a binary search tree. Let \( T_0 \) denote the initial tree. In the \( i \)th round, we traverse the path from the root to node \( x_i \), and then \textit{arbitrarily reconfigure} some subtree \( S_i \) of the current search tree \( T_{i-1} \) to obtain the next search tree \( T_i \). The only restriction is that the subtree \( S_i \) must contain both \( x_i \) and the root of \( T_{i-1} \). (For example, in a splay tree, \( S_i \) is the search path to \( x_i \).) The \textit{cost} of the \( i \)th access is the number of nodes in the subtree \( S_i \).

Prove that the minimum cost of executing an access sequence \( X \) in this model is at least the size of the smallest commune containing the corresponding point set \( X \).

[Hint: Lowest common ancestor.]

*(c)* Suppose \( X \) is a \textit{random} permutation of the integers \( 1, 2, \ldots, n \). Use the lower bound in part (b) to prove that the expected minimum cost of executing \( X \) is \( \Omega(n \log n) \).

★ (d) Describe a polynomial-time algorithm to compute (or even approximate up to constant factors) the smallest commune containing a given set \( X \) of integer points, with at most one point per row. Alternately, prove that the problem is NP-hard.
17 Data Structures for Disjoint Sets

In this lecture, we describe some methods for maintaining a collection of disjoint sets. Each set is represented as a pointer-based data structure, with one node per element. We will refer to the elements as either ‘objects’ or ‘nodes’, depending on whether we want to emphasize the set abstraction or the actual data structure. Each set has a unique ‘leader’ element, which identifies the set. (Since the sets are always disjoint, the same object cannot be the leader of more than one set.) We want to support the following operations.

- **MAKESET**(x): Create a new set \{x\} containing the single element x. The object x must not appear in any other set in our collection. The leader of the new set is obviously x.
- **FIND**(x): Find (the leader of) the set containing x.
- **UNION**(A, B): Replace two sets A and B in our collection with their union A \cup B. For example, **UNION**(A, **MAKESET**(x)) adds a new element x to an existing set A. The sets A and B are specified by arbitrary elements, so **UNION**(x, y) has exactly the same behavior as **UNION**(**FIND**(x), **FIND**(y)).

Disjoint set data structures have lots of applications. For instance, Kruskal’s minimum spanning tree algorithm relies on such a data structure to maintain the components of the intermediate spanning forest. Another application is maintaining the connected components of a graph as new vertices and edges are added. In both these applications, we can use a disjoint-set data structure, where we maintain a set for each connected component, containing that component’s vertices.

### 17.1 Reversed Trees

One of the easiest ways to store sets is using trees, in which each node represents a single element of the set. Each node points to another node, called its **parent**, except for the leader of each set, which points to itself and thus is the root of the tree. **MAKESET** is trivial. **FIND** traverses
parent pointers up to the leader. \texttt{Union} just redirects the parent pointer of one leader to the other. Unlike most tree data structures, nodes do \textit{not} have pointers down to their children.

\begin{verbatim}
\textbf{MakeSet}(x):
  parent(x) ← x
def\texttt{Find}(x):
  while x \neq parent(x)
    x ← parent(x)
  return x
\textbf{Union}(x, y):
  \bar{x} ← \texttt{Find}(x)
  \bar{y} ← \texttt{Find}(y)
  parent(\bar{y}) ← \bar{x}
def\texttt{U/n/sc/i.sc/o.sc/n.sc}(x, y):
  x ← \texttt{F/i.sc/n.sc/d.sc}(x)
  y ← \texttt{F/i.sc/n.sc/d.sc}(y)
  \text{if} depth(\bar{x}) > depth(\bar{y})
    parent(\bar{y}) ← \bar{x}
  \text{else}
    parent(\bar{x}) ← \bar{y}
    depth(\bar{y}) ← depth(\bar{y}) + 1
\end{verbatim}

Merging two sets stored as trees. Arrows point to parents. The shaded node has a new parent.

\textbf{MakeSet} clearly takes Θ(1) time, and \textbf{Union} requires only O(1) time in addition to the two \textbf{Finds}. The running time of \texttt{Find}(x) is proportional to the depth of x in the tree. It is not hard to come up with a sequence of operations that results in a tree that is a long chain of nodes, so that \texttt{Find} takes Θ(n) time in the worst case.

However, there is an easy change we can make to our \textbf{Union} algorithm, called \textit{union by depth}, so that the trees always have logarithmic depth. Whenever we need to merge two trees, we always make the root of the shallower tree a child of the deeper one. This requires us to also maintain the depth of each tree, but this is quite easy.

With this new rule in place, it’s not hard to prove by induction that for any set leader \( \bar{x} \), the size of \( \bar{x}'s \) set is at least \( 2^{\text{depth}(\bar{x})} \), as follows. If \( \text{depth}(\bar{x}) = 0 \), then \( \bar{x} \) is the leader of a singleton set. For any \( d > 0 \), when \( \text{depth}(\bar{x}) \) becomes \( d \) for the first time, \( \bar{x} \) is becoming the leader of the union of two sets, both of whose leaders had depth \( d - 1 \). By the inductive hypothesis, both component sets had at least \( 2^{d-1} \) elements, so the new set has at least \( 2^d \) elements. Later \textbf{Union} operations might add elements to \( \bar{x}'s \) set without changing its depth, but that only helps us.

Since there are only \( n \) elements altogether, the maximum depth of any set is \( \text{lg} \ n \). We conclude that if we use union by depth, both \texttt{Find} and \textbf{Union} run in \( \Theta(\text{log} \ n) \) time in the worst case.

\subsection{17.2 Shallow Threaded Trees}

Alternately, we could just have every object keep a pointer to the leader of its set. Thus, each set is represented by a shallow tree, where the leader is the root and all the other elements are its
children. With this representation, \textsc{MakeSet} and \textsc{Find} are completely trivial. Both operations clearly run in constant time. \textsc{Union} is a little more difficult, but not much. Our algorithm sets all the leader pointers in one set to point to the leader of the other set. To do this, we need a method to visit every element in a set; we will ‘thread’ a linked list through each set, starting at the set’s leader. The two threads are merged in the \textsc{Union} algorithm in constant time.

\begin{center}
\begin{tikzpicture}
  \node (A) at (0,0) [circle,draw] {A};
  \node (B) at (1,0) [circle,draw] {B};
  \node (C) at (2,0) [circle,draw] {C};
  \node (D) at (3,0) [circle,draw] {D};
  \node (E) at (4,0) [circle,draw] {E};
  \node (F) at (5,0) [circle,draw] {F};
  \node (G) at (6,0) [circle,draw] {G};
  \node (H) at (7,0) [circle,draw] {H};
  \node (I) at (8,0) [circle,draw] {I};
  \node (J) at (9,0) [circle,draw] {J};
  \draw (A) -- (B) -- (C) -- (D);
  \draw (E) -- (F) -- (G) -- (H) -- (I) -- (J);
  \draw (A) -- (E);
  \draw (B) -- (F);
  \draw (C) -- (G);
  \draw (D) -- (H);
  \draw (I) -- (J);
\end{tikzpicture}
\end{center}

Merging two sets stored as threaded trees.
Bold arrows point to leaders; lighter arrows form the threads. Shaded nodes have a new leader.

\textsc{MakeSet}(x):
\begin{align*}
\text{leader}(x) & \leftarrow x \\
\text{next}(x) & \leftarrow x \\
\end{align*}

\textsc{Find}(x):
\begin{align*}
\text{return leader}(x)
\end{align*}

\textsc{Union}(x, y):
\begin{align*}
\bar{x} & \leftarrow \text{Find}(x) \\
\bar{y} & \leftarrow \text{Find}(y) \\
y & \leftarrow \bar{y} \\
\text{leader}(y) & \leftarrow \bar{x} \\
\text{next}(y) & \leftarrow \text{next}(\bar{x}) \\
\text{next}(\bar{x}) & \leftarrow y
\end{align*}

The worst-case running time of \textsc{Union} is a constant times the size of the larger set. Thus, if we merge a one-element set with another \(n\)-element set, the running time can be \(\Theta(n)\). Generalizing this idea, it is quite easy to come up with a sequence of \(n\) \textsc{MakeSet} and \(n - 1\) \textsc{Union} operations that requires \(\Theta(n^2)\) time to create the set \(\{1, 2, \ldots, n\}\) from scratch.

\textsc{WorstCaseSequence}(n):
\begin{align*}
\text{MakeSet}(1) \\
\text{for } i \leftarrow 2 \text{ to } n \\
\text{MakeSet}(i) \\
\text{Union}(1, i)
\end{align*}

We are being stupid in two different ways here. One is the order of operations in \textsc{WorstCaseSequence}. Obviously, it would be more efficient to merge the sets in the other order, or to use some sort of divide and conquer approach. Unfortunately, we can’t fix this; we don’t get to decide how our data structures are used! The other is that we always update the leader pointers in the larger set. To fix this, we add a comparison inside the \textsc{Union} algorithm to determine which set is smaller. This requires us to maintain the size of each set, but that’s easy.

\textsc{MakeWeightedSet}(x):
\begin{align*}
\text{leader}(x) & \leftarrow x \\
\text{next}(x) & \leftarrow x \\
\text{size}(x) & \leftarrow 1
\end{align*}

\textsc{WeightedUnion}(x, y):
\begin{align*}
\bar{x} & \leftarrow \text{Find}(x) \\
\bar{y} & \leftarrow \text{Find}(y) \\
\text{if size}(\bar{x}) > \text{size}(\bar{y}) \\
\quad & \begin{align*}
\text{Union}(\bar{x}, \bar{y}) \\
\text{size}(\bar{x}) & \leftarrow \text{size}(\bar{x}) + \text{size}(\bar{y})
\end{align*} \\
\text{else} \\
\quad & \begin{align*}
\text{Union}(\bar{y}, \bar{x}) \\
\text{size}(\bar{y}) & \leftarrow \text{size}(\bar{x}) + \text{size}(\bar{y})
\end{align*}
\end{align*}
The new \texttt{WeightedUnion} algorithm still takes $\Theta(n)$ time to merge two $n$-element sets. However, in an amortized sense, this algorithm is much more efficient. Intuitively, before we can merge two large sets, we have to perform a large number of \texttt{MakeWeightedSet} operations.

**Theorem 1.** A sequence of $m$ \texttt{MakeWeightedSet} operations and $n$ \texttt{WeightedUnion} operations takes $O(m + n \log n)$ time in the worst case.

**Proof:** Whenever the leader of an object $x$ is changed by a \texttt{WeightedUnion}, the size of the set containing $x$ increases by at least a factor of two. By induction, if the leader of $x$ has changed $k$ times, the set containing $x$ has at least $2^k$ members. After the sequence ends, the largest set contains at most $n$ members. (Why?) Thus, the leader of any object $x$ has changed at most $\lceil \log n \rceil$ times.

Since each \texttt{WeightedUnion} reduces the number of sets by one, there are $m - n$ sets at the end of the sequence, and at most $n$ objects are not in singleton sets. Since each of the non-singleton objects had $O(\log n)$ leader changes, the total amount of work done in updating the leader pointers is $O(n \log n)$. \hfill $\square$

The aggregate method now implies that each \texttt{WeightedUnion} has amortized cost $O(\log n)$.

### 17.3 Path Compression

Using unthreaded trees, \texttt{Find} takes logarithmic time and everything else is constant; using threaded trees, \texttt{Union} takes logarithmic amortized time and everything else is constant. A third method allows us to get both of these operations to have almost constant running time.

We start with the original unthreaded tree representation, where every object points to a parent. The key observation is that in any \texttt{Find} operation, once we determine the leader of an object $x$, we can speed up future \texttt{Finds} by redirecting $x$’s parent pointer directly to that leader. In fact, we can change the parent pointers of all the ancestors of $x$ all the way up to the root; this is easiest if we use recursion for the initial traversal up the tree. This modification to \texttt{Find} is called path compression.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{path_compression.png}
\caption{Path compression during \texttt{Find}(c). Shaded nodes have a new parent.}
\end{figure}

If we use path compression, the ‘depth’ field we used earlier to keep the trees shallow is no longer correct, and correcting it would take way too long. But this information still ensures that \texttt{Find} runs in $\Theta(\log n)$ time in the worst case, so we’ll just give it another name: \textit{rank}. The following algorithm is usually called \texttt{union by rank}:

\begin{Verbatim}
\textbf{Find}(x)
\begin{algorithmic}
\IF {$x \neq \text{parent}(x)$}
\STATE $\text{parent}(x) \leftarrow \text{Find}($parent($x)$)$
\STATE \textbf{return} parent($x$)
\ENDIF
\end{algorithmic}
\end{Verbatim}

If we use path compression, the ‘depth’ field we used earlier to keep the trees shallow is no longer correct, and correcting it would take way too long. But this information still ensures that \texttt{Find} runs in $\Theta(\log n)$ time in the worst case, so we’ll just give it another name: \textit{rank}. The following algorithm is usually called \texttt{union by rank}:
**FIND** still runs in $O(\log n)$ time in the worst case; path compression increases the cost by only most a constant factor. But we have good reason to suspect that this upper bound is no longer tight. Our new algorithm memoizes the results of each **FIND**, so if we are asked to **FIND** the same item twice in a row, the second call returns in constant time. Splay trees used a similar strategy to achieve their optimal amortized cost, but our up-trees have fewer constraints on their structure than binary search trees, so we should get even better performance.

This intuition is exactly correct, but it takes a bit of work to define precisely how much better the performance is. As a first approximation, we will prove below that the amortized cost of a **FIND** operation is bounded by the *iterated logarithm* of $n$, denoted $\log^* n$, which is the number of times one must take the logarithm of $n$ before the value is less than 1:

$$
\lg^* n = \begin{cases} 
1 & \text{if } n \leq 2, \\
1 + \lg^*(\lg n) & \text{otherwise.}
\end{cases}
$$

Our proof relies on several useful properties of ranks, which follow directly from the **UNION** and **FIND** algorithms.

- If a node $x$ is not a set leader, then the rank of $x$ is smaller than the rank of its parent.
- Whenever $\text{parent}(x)$ changes, the new parent has larger rank than the old parent.
- Whenever the leader of $x$’s set changes, the new leader has larger rank than the old leader.
- The size of any set is exponential in the rank of its leader: $\text{size}(x) \geq 2^{\text{rank}(x)}$. (This is easy to prove by induction, hint, hint.)
- In particular, since there are only $n$ objects, the highest possible rank is $\lfloor \lg n \rfloor$.
- For any integer $r$, there are at most $n/2^r$ objects of rank $r$.

Only the last property requires a clever argument to prove. Fix your favorite integer $r$. Observe that only set leaders can change their rank. Whenever the rank of any set leader $x$ changes from $r - 1$ to $r$, mark all the objects in $x$’s set. Since leader ranks can only increase over time, each object is marked at most once. There are $n$ objects altogether, and any object with rank $r$ marks at least $2^r$ objects. It follows that there are at most $n/2^r$ objects with rank $r$, as claimed.

### 17.4 $O(\log^* n)$ Amortized Time

arguments; Seidel and Sharir's analysis relies on a comparatively simple recursive decomposition. (Of course, simple is in the eye of the beholder.)

Seidel and Sharir phrase their analysis in terms of two more general operations on set forests. Their more general **COMPRESS** operation compresses any directed path, not just paths that lead to the root. The new **SHATTER** operation makes every node on a root-to-leaf path into its own parent.

![Diagram showing the compress and shatter operations](attachment:image.png)

Clearly, the running time of **FIND(x)** operation is dominated by the running time of **COMPRESS(x, y)**, where y is the leader of the set containing x. Thus, we can prove the upper bound by analyzing an arbitrary sequence of **UNION** and **COMPRESS** operations. Moreover, we can assume that the arguments of every **UNION** operation are set leaders, so that each **UNION** takes only constant worst-case time.

Finally, since each call to **COMPRESS** specifies the top node in the path to be compressed, we can reorder the sequence of operations, so that every **UNION** occurs before any **COMPRESS**, without changing the number of pointer assignments.

Each **UNION** requires only constant time, so we only need to analyze the amortized cost of **COMPRESS**. The running time of **COMPRESS** is proportional to the number of parent pointer assignments, plus $O(1)$ overhead, so we will phrase our analysis in terms of pointer assignments. Let $T(m, n, r)$ denote the worst case number of pointer assignments in any sequence of at most $m$ **COMPRESS** operations, executed on a forest of at most $n$ nodes, in which each node has rank at most $r$.

The following trivial upper bound will be the base case for our recursive argument.

**Theorem 2.** $T(m, n, r) \leq nr$

**Proof:** Each node can change parents at most $r$ times, because each new parent has higher rank than the previous parent. \square

Fix a forest $F$ of $n$ nodes with maximum rank $r$, and a sequence $C$ of $m$ **COMPRESS** operations on $F$, and let $T(F, C)$ denote the total number of pointer assignments executed by this sequence.
Let \( s \) be an arbitrary positive rank. Partition \( F \) into two sub-forests: a 'low' forest \( F_- \) containing all nodes with rank at most \( s \), and a 'high' forest \( F_+ \) containing all nodes with rank greater than \( s \). Since ranks increase as we follow parent pointers, every ancestor of a high node is another high node. Let \( n_- \) and \( n_+ \) denote the number of nodes in \( F_- \) and \( F_+ \), respectively. Finally, let \( m_+ \) denote the number of \textsc{Compress} operations that involve any node in \( F_+ \), and let \( m_- = m - m_+ \).

Any sequence of \textsc{Compress} operations on \( F \) can be decomposed into a sequence of \textsc{Compress} operations on \( F_+ \), plus a sequence of \textsc{Compress} and \textsc{Shatter} operations on \( F_- \), with the same total cost. This requires only one small modification to the code: We forbid any low node from having a high parent. Specifically, if \( x \) is a low node and \( y \) is a high node, we replace any assignment \( \text{parent}(x) \leftarrow y \) with \( \text{parent}(x) \leftarrow x \).

This modification is equivalent to the following reduction:

\[
\text{Compress}(x, y, F): \quad \langle y \text{ is an ancestor of } x \rangle \\
\begin{array}{l}
\text{if } \text{rank}(x) > s \\
\quad \text{Compress}(x, y, F_+) \quad \langle \text{in } C_+ \rangle \\
\text{else if } \text{rank}(y) \leq s \\
\quad \text{Compress}(x, y, F_-) \quad \langle \text{in } C_- \rangle \\
\text{else} \\
\quad z \leftarrow x \\
\quad \text{while } \text{rank}(\text{parent}_F(z)) \leq s \\
\quad \quad z \leftarrow \text{parent}_F(z) \\
\quad \text{Compress}(\text{parent}_F(z), y, F_+) \quad \langle \text{in } C_+ \rangle \\
\text{Shatter}(x, z, F_-) \\
\text{parent}(z) \leftarrow z \quad \langle ? \rangle 
\end{array}
\]

The pointer assignment in the last line (\( ? \)) looks redundant, but it is actually necessary for the analysis. Each execution of that line mirrors an assignment of the form \( \text{parent}(z) \leftarrow w \), where \( z \) is a low node, \( w \) is a high node, and the previous parent of \( z \) was also a high node. Each of these
‘redundant’ assignments happens immediately after a \textsc{compress} in the top forest, so we perform at most $m_+$ redundant assignments.

Each node $x$ is touched by at most one \textsc{shatter} operation, so the total number of pointer reassignments in all the \textsc{shatter} operations is at most $n$.

Thus, by partitioning the forest $F$ into $F_+$ and $F_-$, we have also partitioned the sequence $C$ of \textsc{compress} operations into subsequences $C_+$ and $C_-$, with respective lengths $m_+$ and $m_-$, such that the following inequality holds:

$$T(F, C) \leq T(F_+, C_+) + T(F_-, C_-) + m_+ + n$$

Since there are only $n/2^i$ nodes of any rank $i$, we have $n_+ \leq \sum_{i>s} n/2^i = n/2^s$. The number of different ranks in $F_+$ is $r - s < r$. Thus, Theorem 2 implies the upper bound

$$T(F_+, C_+) < r n/2^s.$$ \[\text{(1)}\]

Let us fix $s = \lfloor \log^* r \rfloor$, so that $T(F_+, C_+) \leq n$. We can now simplify our earlier recurrence to

$$T(F, C) \leq T(F_-, C_-) + m_+ + 2n,$$

or equivalently,

$$T(F, C) - m \leq T(F_-, C_-) - m_- + 2n.$$ \[\text{(2)}\]

Since this argument applies to any forest $F$ and any sequence $C$, we have just proved that

$$T'(m, n, r) \leq T'(m, n, \lfloor \log^* r \rfloor) + 2n,$$

where $T'(m, n, r) = T(m, n, r) - m$. The solution to this recurrence is $T'(n, m, r) \leq 2n \log^* r$. \text{Voilà!}

\textbf{Theorem 3.} $T(m, n, r) \leq m + 2n \log^* r$

\section*{17.5 Turning the Crank}

There is one place in the preceding analysis where we have significant room for improvement. Recall that we bounded the total cost of the operations on $F_+$ using the trivial upper bound from Theorem 2. But we just proved a better upper bound in Theorem 3! We can apply precisely the same strategy, using Theorem 3 recursively instead of Theorem 2, to improve the bound even more.

Suppose we fix $s = \lfloor \log^* r \rfloor$, so that $n_+ = n/2^s r$. Theorem 3 implies that

$$T(F_+, C_+) \leq m_+ + 2n \frac{\log^* r}{2^s r} \leq m_+ + 2n.$$ \[\text{(3)}\]

This implies the recurrence

$$T(F, C) \leq T(F_-, C_-) + 2m_+ + 3n,$$

which in turn implies that

$$T''(m, n, r) \leq T''(m, n, \log^* r) + 3n,$$
where \( T''(m, n, r) = T(m, n, r) - 2m \). The solution to this equation is \( T(m, n, r) \leq 2m + 3n \lg^{**} r \), where \( \lg^{**} r \) is the iterated iterated logarithm of \( r \):

\[
\lg^{**} r = \begin{cases} 
1 & \text{if } r \leq 2, \\
1 + \lg^{**}(\lg^{*} r) & \text{otherwise}.
\end{cases}
\]

Naturally we can apply the same improvement strategy again, and again, as many times as we like, each time producing a tighter upper bound. Applying the reduction \( c \) times, for any positive integer \( c \), gives us \( T(m, n, r) \leq cm + (c + 1)n \lg^{c} r \), where

\[
\lg^{c} r = \begin{cases} 
\lg r & \text{if } c = 0, \\
1 & \text{if } r \leq 2, \\
1 + \lg^{c}(\lg^{c-1} r) & \text{otherwise}.
\end{cases}
\]

Each time we ‘turn the crank’, the dependence on \( m \) increases, while the dependence on \( n \) and \( r \) decreases. For sufficiently large values of \( c \), the \( cm \) term dominates the time bound, and further iterations only make things worse. The point of diminishing returns can be estimated by the minimum number of stars such that \( \lg^{c} r \) is smaller than a constant:

\[
\alpha(r) = \min \{ c \geq 1 \mid \lg^{c} n \leq 3 \}.
\]

(The threshold value 3 is used here because \( \lg^{c} 5 \geq 2 \) for all \( c \).) By setting \( c = \alpha(r) \), we obtain our final upper bound.

**Theorem 4.** \( T(m, n, r) \leq m\alpha(r) + 3n(\alpha(r) + 1) \)

We can assume without loss of generality that \( m \geq n \) by ignoring any singleton sets, so this upper bound can be further simplified to \( T(m, n, r) = O(m\alpha(r)) = O(m\alpha(n)) \). It follows that if we use union by rank, \textsc{Find} with path compression runs in \( O(\alpha(n)) \) amortized time.

Even this upper bound is somewhat conservative if \( m \) is larger than \( n \). A closer estimate is given by the function

\[
\alpha(m, n) = \min \{ c \geq 1 \mid \lg^{c}(\lg n) \leq m/n \}.
\]

It’s not hard to prove that if \( m = \Theta(n) \), then \( \alpha(m, n) = \Theta(\alpha(n)) \). On the other hand, if \( m \geq n \lg^{****} n \), for any constant number of stars, then \( \alpha(m, n) = O(1) \). So even if the number of \textsc{Find} operations is only slightly larger than the number of nodes, the amortized cost of each \textsc{Find} is constant.

\( O(\alpha(m, n)) \) is actually a tight upper bound for the amortized cost of path compression; there are no more tricks that will improve the analysis further. More surprisingly, this is the best amortized bound we obtain for any pointer-based data structure for maintaining disjoint sets; the amortized cost of every \textsc{Find} algorithm is at least \( \Omega(\alpha(m, n)) \). The proof of the matching lower bound is, unfortunately, far beyond the scope of this class.\(^3\)

---

17.6 The Ackermann Function and its Inverse

The iterated logarithms that fell out of our analysis of path compression are the inverses of a hierarchy of recursive functions defined by Wilhelm Ackermann in 1928.\(^4\)

\[ 2 \uparrow^c n := \begin{cases} 
2 & \text{if } n = 1 \\
2n & \text{if } c = 0 \\
2 \uparrow^{c-1} (2 \uparrow^c (n-1)) & \text{otherwise}
\end{cases} \]

For each fixed integer \(c\), the function \(2 \uparrow^c n\) is monotonically increasing in \(n\), and these functions grow incredibly faster as the index \(c\) increases. \(2 \uparrow n\) is the familiar power function \(2^n\). \(2 \uparrow\uparrow n\) is the tower function:

\[ 2 \uparrow\uparrow n = \underbrace{2 \uparrow 2 \ldots \uparrow 2}_{n} = 2^{2^{\ldots^{2}}}. \]

John Conway named \(2 \uparrow\uparrow\uparrow n\) the wower function:

\[ 2 \uparrow\uparrow\uparrow n = \underbrace{2 \uparrow\uparrow 2 \ldots \uparrow\uparrow 2}_{n}. \]

And so on, et cetera, ad infinitum.

For any fixed \(c\), the function \(\log^c n\) is the inverse of the function \(2 \uparrow^{c+1} n\), the \((c + 1)\)th row in the Ackerman hierarchy. Thus, for any remotely reasonable values of \(n\), say \(n \leq 2^{256}\), we have \(\log^c n \leq 5\), \(\log^{\ast+} n \leq 4\), and \(\log^{\ast} n \leq 3\) for any \(c \geq 3\).

The function \(\alpha(n)\) is usually called the inverse Ackerman function.\(^5\) Our earlier definition is equivalent to \(\alpha(n) = \min\{c \geq 1 \mid 2 \uparrow^{c+2} 3 \geq n\}\); in other words, \(\alpha(n) + 2\) is the inverse of the third column in the Ackermann hierarchy. The function \(\alpha(n)\) grows much more slowly than \(\log^c n\) for any fixed \(c\); we have \(\alpha(n) \leq 3\) for all even remotely imaginable values of \(n\). Nevertheless, the function \(\alpha(n)\) is eventually larger than any constant, so it is not \(O(1)\).

<table>
<thead>
<tr>
<th>2 \uparrow n</th>
<th>(n = 1)</th>
<th>(n = 2)</th>
<th>(n = 3)</th>
<th>(n = 4)</th>
<th>(n = 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2n</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>2 \uparrow n</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>2 \uparrow\uparrow n</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>65536</td>
<td>2(^{65536})</td>
</tr>
<tr>
<td>2 \uparrow\uparrow\uparrow n</td>
<td>2</td>
<td>4</td>
<td>65536</td>
<td>(2^{65536})</td>
<td>(2^{65536})</td>
</tr>
<tr>
<td>2 \uparrow\uparrow\uparrow\uparrow n</td>
<td>2</td>
<td>4</td>
<td>(2^{65536})</td>
<td>(2^{65536})</td>
<td>(2^{65536})</td>
</tr>
<tr>
<td>2 \uparrow\uparrow\uparrow\uparrow\uparrow n</td>
<td>2</td>
<td>4</td>
<td>(2^{65536})</td>
<td>(2^{65536})</td>
<td>(2^{65536})</td>
</tr>
<tr>
<td>2 \uparrow\uparrow\uparrow\uparrow\uparrow\uparrow n</td>
<td>2</td>
<td>4</td>
<td>(2^{65536})</td>
<td>(2^{65536})</td>
<td>(2^{65536})</td>
</tr>
</tbody>
</table>

Small (!!!) values of Ackermann’s functions.

\(^4\)Ackermann didn’t define his functions this way—I’m actually describing a slightly cleaner hierarchy defined 35 years later by R. Creighton Buck—but the exact details of the definition are surprisingly irrelevant! The mnemonic up-arrow notation for these functions was introduced by Don Knuth in the 1970s.

\(^5\)Strictly speaking, the name ‘inverse Ackerman function’ is inaccurate. One good formal definition of the true inverse Ackerman function is \(\tilde{\alpha}(n) = \min\{c \geq 1 \mid \log^{\ast} n \leq c\} = \min\{c \geq 1 \mid 2 \uparrow^{c+2} 3 \geq n\}\). However, it’s not hard to prove that \(\tilde{\alpha}(n) \leq \alpha(n) + 1\) for all sufficiently large \(n\), so the inaccuracy is completely forgivable. As I said in the previous footnote, the exact details of the definition are surprisingly irrelevant!
17.7 To infinity... and beyond!

Of course, one can generalize the inverse Ackermann function to functions that grow arbitrarily more slowly, starting with the iterated inverse Ackermann function

$$\alpha^*(n) = \begin{cases} 1 & \text{if } n \leq 4, \\ 1 + \alpha^*(\alpha(n)) & \text{otherwise}, \end{cases}$$

then the iterated iterated iterated inverse Ackermann function

$$\alpha^{c}(n) = \begin{cases} \alpha(n) & \text{if } c = 0, \\ 1 & \text{if } n \leq 4, \\ 1 + \alpha^{c}(\alpha^{c-1}(n)) & \text{otherwise}, \end{cases}$$

and then the diagonalized inverse Ackermann function

$$\text{Head-asplode}(n) = \min\{c \geq 1 \mid \alpha^c n \leq 4\},$$

and so on ad nauseam. Fortunately(?), such functions appear extremely rarely in algorithm analysis. Perhaps the only naturally-occurring example of a super-constant sub-inverse-Ackermann function is a recent result of Seth Pettie,⁶ who proved that if a splay tree is used as a double-ended queue — insertions and deletions of only smallest or largest elements — then the amortized cost of any operation is $O(\alpha^*(n))$!

Exercises

1. Consider the following solution for the union-find problem, called **union-by-weight**. Each set leader $x$ stores the number of elements of its set in the field $weight(x)$. Whenever we UNION two sets, the leader of the smaller set becomes a new child of the leader of the larger set (breaking ties arbitrarily).

   **MAKESET(x):**
   
   ```
   parent(x) ← x
   weight(x) ← 1
   ```

   **FIND(x):**
   
   ```
   while x ≠ parent(x)
   x ← parent(x)
   return x
   ```

   **UNION(x, y):**
   
   ```
   x ← FIND(x)
   y ← FIND(y)
   if weight(x) > weight(y)
     parent(y) ← x
     weight(x) ← weight(x) + weight(y)
   else
     parent(x) ← y
     weight(x) ← weight(x) + weight(y)
   ```

   Prove that if we use union-by-weight, the worst-case running time of FIND(x) is $O(\log n)$, where $n$ is the cardinality of the set containing $x$.

2. Consider a union-find data structure that uses union by depth (or equivalently union by rank) **without** path compression. For all integers $m$ and $n$ such that $m \geq 2n$, prove that there is a sequence of $n$ MAKESET operations, followed by $m$ UNION and FIND operations, that require $\Omega(m \log n)$ time to execute.

---

3. Suppose you are given a collection of up-trees representing a partition of the set \( \{1, 2, \ldots, n\} \) into disjoint subsets. **You have no idea how these trees were constructed.** You are also given an array \( \text{node}[1..n] \), where \( \text{node}[i] \) is a pointer to the up-tree node containing element \( i \). Your task is to create a new array \( \text{label}[1..n] \) using the following algorithm:

```plaintext
LABELEVERYTHING:
for i ← 1 to n
    label[i] ← FIND(node[i])
```

(a) What is the worst-case running time of `LABELEVERYTHING` if we implement `FIND` without path compression?

(b) **Prove** that if we implement `FIND` using path compression, `LABELEVERYTHING` runs in \( O(n) \) time in the worst case.

4. Consider an arbitrary sequence of \( m \) `MAKESET` operations, followed by \( u \) `UNION` operations, followed by \( f \) `FIND` operations, and let \( n = m + u + f \). Prove that if we use union by rank and `FIND` with path compression, all \( n \) operations are executed in \( O(n) \) time.

5. Suppose we want to maintain an array \( X[1..n] \) of bits, which are all initially zero, subject to the following operations.

   - `LOOKUP(i)`: Given an index \( i \), return \( X[i] \).
   - `BLACKEN(i)`: Given an index \( i < n \), set \( X[i] ← 1 \).
   - `NEXTWHITE(i)`: Given an index \( i \), return the smallest index \( j \geq i \) such that \( X[j] = 0 \). (Because we never change \( X[n] \), such an index always exists.)

If we use the array \( X[1..n] \) itself as the only data structure, it is trivial to implement `LOOKUP` and `BLACKEN` in \( O(1) \) time and `NEXTWHITE` in \( O(n) \) time. But you can do better! Describe data structures that support `LOOKUP` in \( O(1) \) worst-case time and the other two operations in the following time bounds. (We want a different data structure for each set of time bounds, not one data structure that satisfies all bounds simultaneously!)

(a) The worst-case time for both `BLACKEN` and `NEXTWHITE` is \( O(\log n) \).

(b) The amortized time for both `BLACKEN` and `NEXTWHITE` is \( O(\log n) \). In addition, the worst-case time for `BLACKEN` is \( O(1) \).

(c) The amortized time for `BLACKEN` is \( O(\log n) \), and the worst-case time for `NEXTWHITE` is \( O(1) \).

(d) The worst-case time for `BLACKEN` is \( O(1) \), and the amortized time for `NEXTWHITE` is \( O(\alpha(n)) \). [Hint: There is no `WHITEN`.]

6. Suppose we want to maintain a collection of strings (sequences of characters) under the following operations:

   - `NEWSTRING(a)` creates a new string of length 1 containing only the character \( a \) and returns a pointer to that string.
• \textsc{Concat}(S, T) removes the strings \(S\) and \(T\) (given by pointers) from the data structure, adds the concatenated string \(ST\) to the data structure, and returns a pointer to the new string.
• \textsc{Reverse}(S) removes the string \(S\) (given by a pointer) from the data structure, adds the reversal of \(S\) to the data structure, and returns a pointer to the new string.
• \textsc{Lookup}(S, k) returns the \(k\)th character in string \(S\) (given by a pointer), or \textsc{Null} if the length of the \(S\) is less than \(k\).

Describe and analyze a \textit{simple} data structure that supports \textsc{Concat} in \(O(\log n)\) amortized time, supports every other operation in \(O(1)\) worst-case time, and uses \(O(n)\) space, where \(n\) is the sum of the current string lengths. Unlike the similar problem in the previous lecture note, there is no \textsc{Split} operation. \textit{[Hint: Why is this problem here?]}  

7. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an \(n \times n\) bitmap \(B[1..n, 1..n]\).

For example, given the bitmap below as input, your algorithm should return the number \(9\), because the largest connected black component (marked with white dots on the right) contains nine pixels.

(b) Design and analyze an algorithm \textsc{Blacken}(i, j) that colors the pixel \(B[i,j]\) black and returns the size of the largest black component in the bitmap. For full credit, the \textit{amortized} running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

For example, at each step in the sequence below, we blacken the pixel marked with an \(X\). The largest black component is marked with white dots; the number underneath shows the correct output of the \textsc{Blacken} algorithm.

(c) What is the \textit{worst-case} running time of your \textsc{Blacken} algorithm?

*8. Consider the following game. I choose a positive integer \(n\) and keep it secret; your goal is to discover this integer. We play the game in rounds. In each round, you write a list of \textit{at most} \(n\) integers on the blackboard. If you write more than \(n\) numbers in a single round, you lose. (Thus, in the first round, you must write only the number \(1\); do you see why?) If \(n\) is one of the numbers you wrote, you win the game; otherwise, I announce which of
the numbers you wrote is smaller or larger than \( n \), and we proceed to the next round. For example:

<table>
<thead>
<tr>
<th>You</th>
<th>Me</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>It's bigger than 1.</td>
</tr>
<tr>
<td>4, 42</td>
<td>It's between 4 and 42.</td>
</tr>
<tr>
<td>8, 15, 16, 23, 30</td>
<td>It's between 8 and 15.</td>
</tr>
<tr>
<td>9, 10, 11, 12, 13, 14</td>
<td>It's 11; you win!</td>
</tr>
</tbody>
</table>

Describe a strategy that allows you to win in \( O(\alpha(n)) \) rounds!