Recitation 2: Divide and Conquer I: Median Finding, Amortized Analysis I: Union-Find

1 Median Finding

Given a set S of n numbers, define $rank(x)$ to be the number of elements of S that are $\leq x$. Also define the *upper median* to be the element of rank $\lceil \frac{n+1}{2} \rceil$, and the *lower median* to be the element of rank $\lfloor \frac{n+1}{2} \rfloor$. Note that if n is odd, then these are equal to the median of S. For example, the (upper/lower) median of $S = \{2, -5, 3, 10, 1, -1, 8\}$ is 2.

In general, if n is even, the median is defined to be the mean of the upper and lower median. In this class, unless otherwise stated, we will assume that n is odd and that S has a unique median, which is the element of rank $\frac{n+1}{2}$. We will also assume that the elements are distinct, because it simplifies the reasoning.

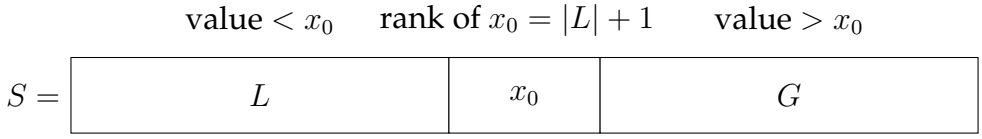
Instead of just finding medians, we will solve a more general problem, which we will call the selection-rank problem:

> Given a set S of n distinct numbers and a number $i \in 1, 2, 3, \ldots, n$, find the element $x \in S$ s.t. $rank(x) = i$ (that is, the i^{th} smallest element).

The naive algorithm is to sort the set and return the i^{th} element. This has runtime $O(n \lg n)$, *e.g.* using merge sort. Can we do better? $O(n)$, perhaps? It turns out – yes, using divide and conquer (Blum, Floyd, Pratt, Rivest, Tarjan 1973)!

The general idea of the algorithm is:

1. Pick some $x_0 \in S$ cleverly – we'll come back to this step later.
2. Divide S into two subsets, L containing all elements less than x_0, and G containing all elements greater than x_0. So $rank(x_0) = |L| + 1$.



3. If $rank(x_0) = i$, then $x = x_0$ and we are done.
   If $rank(x_0) > i$, then the true x must be in the sublist with values smaller than x_0, so recurse on L.

If $rank(x_0) < i$, then the true x must be in the sublist with values larger than x_0, so recurse on G.

Why do we need to pick x cleverly? Unfortunately, an adversary could lead us into a VERY unbalanced recursion. For example, the partition of S into L and G could be so bad that we end up with, say, $L = \phi = \{\}$ and $G = S\backslash\{1\} = \{2, 3, ..., n\}$. In this adversarial scenario, the reduction in size per level is only 1. This means we have $\Theta(n)$ levels with $\Theta(n)$ work per level, which means we have $\Theta(n^2)$ total running time. Choosing an element at random is better – it would defeat the adversary often enough in expectation to have an *expected* runtime of $O(n)$, but it still has a worst case runtime of $\Theta(n^2)$, and it won't work if we want a deterministic algorithm.

So, how can we pick x cleverly? We need to pick x such that $rank(x)$ is not extreme. Of course, $x = median(S)$ would be ideal, but that's what we're trying to solve in the first place!

The key idea here is that we don't actually need the median – we just need an x which is "good enough". Define x to be "c-balanced" if

$$max\{rank(x), n - rank(x)\} \leq c \cdot n \text{ for some constant } c < 1$$

If we can pick an x that is c-balanced in, for example, $O(n)$ time, then our recursion would be $T(n) = T(c \cdot n) + O(n) = O(n)$. The size of each subproblem reduces exponentially as $n, cn, c^2n, \ldots$. That is, it follows a decreasing geometric series and is therefore $O(n)$.

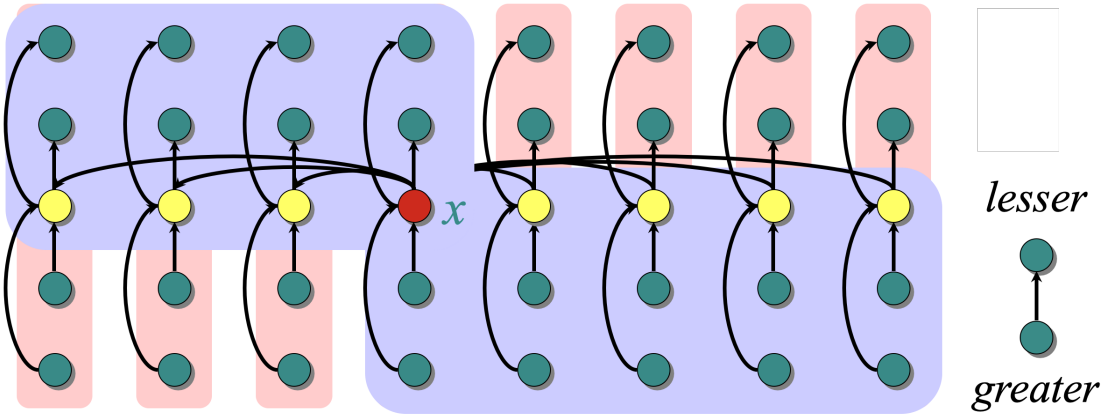
Description of Algorithm

1. Assume $|S| = n$ is a power of 10. If not, pad S (add enough small numbers) to make it so. This increases the size of S by at most a factor of 10.
2. Divide the n elements into $\frac{n}{5}$ groups of 5 elements each.
3. Sort each group of 5 elements and find the median of each group.
4. Recursively find the median x of the $\frac{n}{5}$ group medians.
5. (As before) Compute $L = \{y \in S | y < x\}$ and $G = \{y \in S | y > x\}$.
6. (As before) If $rank(x) = i$, then we are done.
   If $rank(x) > i$, then find the element of rank i in L.
   If $rank(x) < i$, then find the element of rank $(i - rank(x))$ in G.

Steps 2 and 3 take $O(n)$ time, step 4 is $T(\frac{n}{5})$ (recursively), step 5 is $O(n)$ to iterate through the whole list, and step 6 is either $T(|L|)$ or $T(|G|)$ depending on which subset we recurse.

Proof of Correctness

The **key observation** is that the "median of medians" element x we obtain at the end of step 4 is $\frac{3}{4}$-balanced, meaning $max\{|L|, |G|\} \leq \frac{3}{4} \cdot n$.



The above diagram provides a visual proof. Note the key which indicates that elements get bigger to the right and down. First consider how many elements are $\leq x$ (shaded in the top left corner). We know that at least $\frac{1}{2} \cdot \frac{n}{5} = \frac{n}{10}$ group medians are $\leq x$. For each of those $\frac{n}{10}$ medians, additionally two other elements in that group are smaller than that group median. So at least $\frac{3n}{10}$ elements are $\leq x$. Similarly, at least $\frac{3n}{10}$ elements are greater than x. Thus,

$$|L| \geq \frac{3n}{10} > \frac{n}{4} \quad \Longrightarrow \quad n - rank(x) \leq \frac{3}{4} \cdot n$$

Now, consider how many elements are $\geq x$ (shaded in bottom right corner). Similarly, there are at least three elements for each of the $\frac{n}{10}$ group medians above x that are all above x, so

$$|G| \geq \frac{3n}{10} > \frac{n}{4} \quad \Longrightarrow \quad rank(x) \leq \frac{3}{4} \cdot n$$

So x is indeed $\frac{3}{4}$-balanced.

Runtime Analysis

The runtime recurrence is:

$$T(n) = T\left(\frac{3}{4}n\right) + T\left(\frac{n}{5}\right) + O(n)$$

The first term, $T(\frac{3}{4}n)$ comes from the standard conquer step and represents the time it takes to solve the subproblem (recursing on L or G). The second term, $T(\frac{n}{5})$ is the additional time needed to find x, by using median of medians, in step 4. The $O(n)$ is for everything else.

We claim that $T(n) = \Theta(n)$, that is, $T(n) \leq c_1 \cdot n$ for some constant c_1.

Proof. By induction! Suppose our claim is true for all problems of size n or smaller. Let c_2 be the constant from the $O(n)$ term. Then,

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}n\right) + c_2 n \\ &\leq \frac{c_1 n}{5} + \frac{3c_1 n}{4} + c_2 n \\ &= \frac{19}{20}c_1 n + c_2 n \\ &= c_1 n + \left(c_2 - \frac{c_1}{20}\right)n \\ &< c_1 n \text{ if we set } c_1 > 20c_2 \end{aligned}$$

Why did we guess linear time? (And not, say, $n \log n$?) We have reason to guess so because $\frac{1}{5} + \frac{3}{4} < 1$. There is a large constant factor reduction in problem size per step, and the subproblem size follows a geometric series (so the overall time is roughly just the time in the first step of the recursion)!

Note that **we specifically chose groups of 5** to recursively find the median of medians. What happens if we try the same algorithm with groups of 3 instead?

Using the same argument as before, we know that the median of medians is greater than at least two elements, in at least half of the groups. That is, the median of medians is greater than $\frac{2n}{6}$ elements. This guarantees that our chosen x is $\frac{2}{3}$-balanced.

With groups of 3, we now have $\frac{n}{3}$ total groups, so finding the median of medians will take $T(\frac{n}{3})$ time. Our recursion becomes:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$$

Since $\frac{n}{3} + \frac{2n}{3} = 1$, this will ***not*** solve to linear time. Instead, we're back to $O(n \log n)$ time.

What if we try groups of 7 instead? Using the same analysis, we can see that finding the median of medians in this case will take $T(\frac{n}{7})$ time. The median of medians will be greater than at least 4 elements, in half the groups, thus larger than $\frac{4n}{14}$ elements. This guarantees that our chosen x is $\frac{5n}{7}$-balanced.

This provides a recursion of:

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + O(n)$$

which will also solve to $O(n)$ time. The moral of the story is – the number of groups must be chosen carefully in order to guarantee a linear-time algorithm.

2 Amortized Analysis

The main idea behind amortized analysis is to give a tight upper-bound for a *sequence of operations* that is better than using worst-case analysis on individual operations.

For a given data-structure D, let $0, 1, .. m-1$ be a sequence of m operations on the data structure. In amortized analysis, we are not interested in the cost of any individual operation, but rather the total cost of the entire sequence of operations. We can then average over the number of operations to get the amortized cost per operation. Hence an operation has amortized cost $T(n)$ if any subsequence of k operations costs at most $k \cdot T(n)$.

Note: A common misconception is to conflate amortized analysis for average-case analysis. This is not true. Amortized analysis is in fact a *worst-case* analysis, but bounds a sequence of operations rather than an individual operation.

The three most common methods for amortized analysis are the **Aggregate** Method, the **Accounting** Method, and finally the most powerful, **Potential** Method. We'll examine how each of these methods can be used to analyze the performance of two specific data structures.

2.1 Data Structures

First we will describe two data structures, without using amortized analysis. We will later use amortized analysis to improve the worst-case bounds on the runtime for sequences of operations.

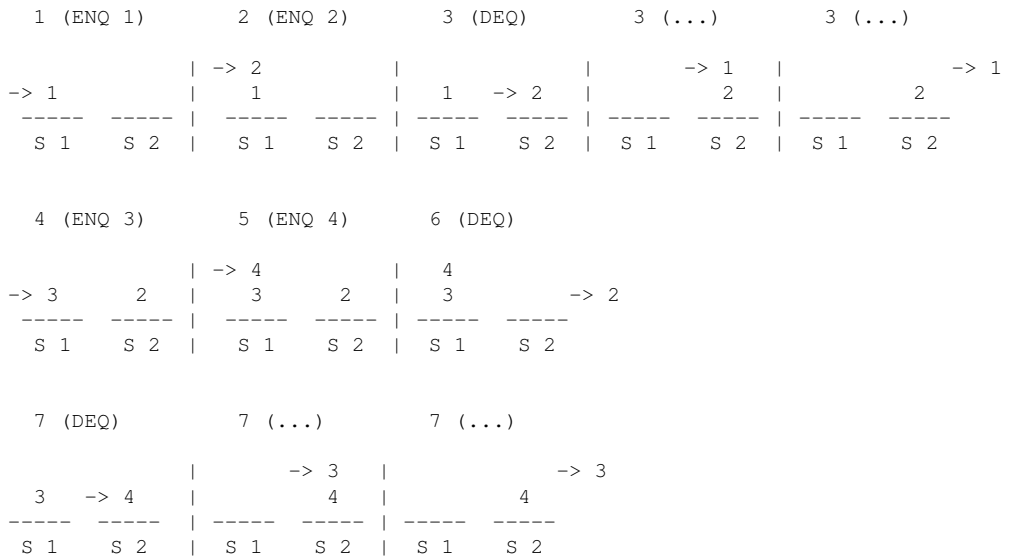
2.1.1 Queue using Two Stacks

Recall that a Stack has operations PUSH(X) which adds x to the top of the stack, and POP() which removes and returns the items at the top of the stack (LIFO - last in first out). A Queue supports operations ENQUEUE(X) which adds x to the back of the queue and DEQUEUE() which removes and returns the item at the front of the queue (FIFO - first in first out).

Given two stacks s_1 and s_2, we will implement a Queue data structure as follows:

ENQUEUE(X): Push x onto s_1
DEQUEUE(): If s_2 empty, for all items in s_1, pop them out of s_1 and push them onto s_2. Then pop and return item from s_2.

Worst-Case: The worst-case cost of an ENQUEUE operation is $O(1)$ since it makes only 1 push. The worst-case cost of a DEQUEUE operation is $O(n)$, where n is the total number of elements, because it may have to perform $O(n)$ pops to transfer items from s_1 to s_2.



2.1.2 Binary Counter

This data structure is an m-bit binary counter which counts from 0 to $2^m - 1$. It is represented by a list of m binary digits which are all initially set to 0: $[0, 0, 0 \ldots, 0]$. The data structure supports only one operation, INCREMENT() which increments its value by 1. The operation works as follows:

INCREMENT(): Start from the right most bit and scan to the left flipping each bit from 1 to 0. When encountering the first 0, flip it to a 1 and terminate. (Note that this correctly increments the binary counter by 1).

Worst-Case: The worst-case cost of the INCREMENT() operation is $O(m)$ since the counter may be in a state $[0, 0, 1, 1, 1, \ldots, 1]$ where there are $O(m)$ 1's to flip to 0

2.2 Aggregate Analysis Method

In Aggregate Analysis, we compute the total cost of n operations by simply adding them up. We can then divide this cost by n to get the amortized cost. This is usually the most straightforward, but often infeasible, method for amortization.

2.2.1 Queue using Two Stacks

Imagine a sequence of n operations (any mixture of ENQUEUE's and DEQUEUE's) on this data structure. Naively, we can bound the cost of this sequence of operations by $O(n^2)$ by looking at the worst-case cost of each operation. However, let's count the number of

underlying PUSH and POP operations. Since an element must be added to the queue in order to be removed, the number of DEQUEUE operations is at most the number of ENQUEUE operations. Additionally, once an element gets removed from the Queue, there can be no more operation costs incurred on it.

Hence, for each element x that gets added to the Queue, it costs one PUSH to add it to s_1, one POP + PUSH to move it from s_1 to s_2 and then finally one POP to remove it from s_2. Therefore, for each element that can ever be added to the Queue, there can be at most a constant number of PUSH and POP operations incurred on it. And since there can be at most n elements added to the Queue, the total cost of any sequence of n operations is $O(n)$.

Therefore, the amortized cost is $O(1)$.

2.2.2 Binary Counter

Let's count exactly how many time each bit gets flipped in the process of n INCREMENT operations. The least significant bit gets flipped with each INCREMENT call, so it gets flipped a total of n times. The second-least significant bit gets flipped with every other call, so that's a total of $\frac{n}{2}$ flips, and so on. Hence the total number of bit flips is $n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{2^{m-1}} \leq 2n$. Therefore, the total cost of all n operations is $O(n)$, and the amortized cost is $O(1)$.

2.3 Accounting Method

The Accounting method works by paying in advance the cost of later operations. Intuitively, you can think of it as a savings system for a bank—every time you do a low-cost operation, you put a few extra coins in the bank (paying in advance), and use them to cover the cost of an occasional expensive operation. Remember, this is not an algorithm, but rather an analysis technique to measure exactly how much we spend over a sequence of operations.

Mathematically, for each operation i, let c_i be the actual cost of the operation. We assign an amortized cost $\hat{c}_i$ (coins) to each operation so that the total amortized cost is an upper bound on the total actual cost, $\sum_i \hat{c}_i \geq \sum_i c_i$.

Note: Make sure your accounting analysis maintains a nonnegative bank balance for any possible sequence of operations (you can't spend what you don't have).

2.3.1 Queue using Two Stacks

Let us assign an amortized cost of 4 coins for the ENQUEUE operation. We will show how doing so leads to an amortized cost of 0 coins for the DEQUEUE operation.

When an element x is added to the queue, it uses 1 coin to pay for the initial PUSH onto s_1. It will store the 3 extra coins for later. Two of these coins will be used if this coin is ever moved from s_1 to s_2 and it will have 1 coin left over. This last coin will be used if it is ever popped from s_2 due to a DEQUEUE call. Hence, this ensures that whenever DEQUEUE is called, the item being popped will have enough credit so that the DEQUEUE operation is free.

Therefore, since we pay at most 4 coins for any operation, the amortized cost is $O(1)$.

2.3.2 Binary Counter

Let's assign a cost of 2 coins for flipping a 0 to a 1 (setting a bit). Hence when the INCREMENT operation is called, whenever it sets a bit from 0 to 1, it will use the first coin to pay for the flip and it will place the second coin on the bit to be used for later. Note that since all the bits start at 0, any bit that is a 1 will have a coin on it. It will use it to pay for itself if it ever gets reset to 0. This way, all the costs are always paid for any sequence of INCREMENT operations. Since we pay at most 2 coins, the amortized cost of INCREMENT is $O(1)$.

2.4 Potential Method

The Potential method is the most powerful of the three amortization methods. In this method, you can think of a "potential energy'' associated with your data structure.

Suppose we perform n operations on our data structure D. We define D_0 to be the initial data structure. Then, for $i = 1, 2, .. n$, let D_i be the state of the data structure immediately after performing operation i on D_{i-1}. We define a potential function $\Phi : \{D_0, D_1, .. D_n\} \rightarrow \mathbb{R}$ to be a function which maps each D_i to a real number $\Phi(D_i)$.

Let c_i be the actual cost of operation i. Then, for a given potential function Φ, we define the amortized cost $\hat{c}_i$ of operation i as follows,

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

Or more concisely, $\hat{c}_i = c_i + \Delta\Phi$. Using this definition, we can compute the total amortized

cost of n operations as follows,

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

Therefore, as long as for our given potential has $\Phi(D_n) \geq \Phi(D_0)$ the total amortized cost gives an upper bound on the total actual cost. In practice, since n may not be known before hand, we require $\Phi(D_i) \geq \Phi(D_0)$ for all i. This way, at any given time, the total amortized cost is an upper bound on the total actual cost incurred so far. To further simplify our work, we typically set $\Phi(D_0) = 0$ and simply ensure $\Phi(D_i) \geq 0$ at any time.

The biggest challenge in the potential method is finding a potential function Φ which has small $\Phi(D_n) - \Phi(D_0)$ so that it gives a tight bound on the total actual cost. While this is generally a non trivial problem, the following intuition is helpful.

Intuition: Choose your potential Φ so that $\Delta\Phi$ decreases by a large amount when an expensive operation is performed on your data structure.

We will now apply the potential method to our examples we have been using so far.

2.4.1 Queue using Two Stacks

We define our potential to be $\Phi(D_i) = 2|s_1^i|$, two times the number of elements in s_1 immediately after the i-th operation (s_1^i to refers to s_1 immediately after the i-th operation). The intuition for this choice is that the most expensive operation in our queue is when DEQUEUE has to move all the items out of s_1 into s_2. When this happens, the number of elements in s_1 decreases significantly, therefore decreasing our potential.

First, we must ensure that Φ is a valid potential function. $\Phi(D_0) = 0$ because s_1 starts out empty. $\Phi(D_i) \geq 0$ for any i since the number of items in s_1 can never be negative. This ensures that the amortized cost of n operations is according to Φ is indeed an upper bound on the total actual cost of n operations.

Let's first analyze the ENQUEUE operation. We know that the actual cost is $c_i = 1$ since it only makes one call to PUSH onto s_1. Since the number of elements in s_1 increases by 1, we have the following amortized cost for ENQUEUE:

$$\hat{c}_i = c_i + \Delta\Phi = 1 + 2 = 3$$

Now let's analyze DEQUEUE. There are two cases for DEQUEUE; the first is the easy case where s_2 is non-empty, which gives us an amortized cost of $\hat{c}_i = c_i + \Delta\Phi = 1 + 0 = 1$

since the size of s_1 does not change. In the second case (when s_2 is empty), the actual cost of DEQUEUE is $2|s_1^i| + 1$ since there is a cost of one POP our of s_1 and one PUSH onto s_2 for all items in s_1, and then finally a single POP from s_2. However, when this happens, the number of elements in s_1 decreases by $|s_1^i|$, and our amortized cost is

$$\hat{c}_i = c_i + \Delta\Phi = 2|s_1^i| + 1 - 2|s_1^i| = 1$$

Hence, in both cases, the amortized cost of DEQUEUE is also $O(1)$.

2.4.2 Binary Counter

The most expensive operation is when there is a large number of trailing 1's in the counter and INCREMENT has to set them all to 0 before finally flipping a 0 to a 1 and terminating. Therefore, a good guess for a potential function would be the number of 1s in the counter.

We first ensure that Φ is a valid potential function: $\Phi(D_0) = 0$ because our counter starts at 0, and at any given time, $\Phi(D_i) \geq 0$ since our counter only increments. Therefore, the total amortized cost according to Φ gives an upper bound on the total actual cost after n increments.

Suppose the number of leading 1's before operation i was k. Then the actual cost of the i-th INCREMENT is $c_i = k + 1$ since it has to flip all k bits to 0 and then the final bit from 0 to 1. However, in this process the number of 1's in our binary counter goes down by $k - 1$. Therefore, the amortized cost for INCREMENT is,

$$\hat{c}_i = c_i + \Delta\Phi = k + 1 - (k - 1) = 2 = O(1)$$

3 Union Find Data Structure [Lecture Review]

A *union-find data structure*, also known as a *disjoint-set data structure*, is a data structure that can keep track of a collection of pairwise disjoint sets $\mathcal{S} = \{S_1, S_2, \ldots, S_r\}$ containing a total of n elements. Each set S_i has a single, arbitrarily chosen element that can serve as a representative for the entire set, denoted as $rep[S_i]$.

Specifically, we wish to support the following operations:

- MAKE-SET(x), which adds a new set $\{x\}$ to $\mathcal{S}$ with $rep[\{x\}] = x$.
- FIND-SET(x), which determines which set $S_x \in \mathcal{S}$ contains x and returns $rep[S_x]$.

- UNION(x, y), which replaces S_x and S_y with $S_x \cup S_y$ in $\mathcal{S}$ for any x, y in distinct sets S_x, S_y.

Furthermore, we want to make these as efficient as possible. We will go through the process of creating a data structure that, in an amortized sense, performs spectacularly.

3.1 Motivation

Union-find data structure is used in many different algorithms. A natural use case of union find is keeping track of connected components in an undirected graph where nodes and edges could be added dynamically. We start with the initial connected components as S_i's. As we add a node v or an edge $E = \{(u, v)\}$, we either

1. To make a new node, call MAKE-SET(v) to make a new component for v.
2. To add an edge between existing nodes, call UNION(u, v) to connect all of u's connected components with all of v's connected components.

3.2 Linked List Solution

A naïve way to solve this problem is to represent each set with a doubly linked list like so:

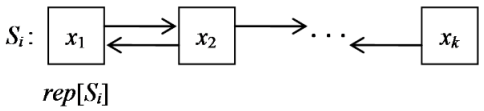


Figure 1: A simple doubly linked list.

We may also have a data structure, such as a hash table mapping elements to pointers, that allows us to access each linked list node in constant time.

We define $rep[S_i]$ to be the element at the head of the list representing S_i.

Here are the algorithms for each operation:

- MAKE-SET(x) initializes x as a lone node. This takes $\Theta(1)$ time in the worst case.
- FIND-SET(x) walks left in the list containing x until it reaches the front of the list. This takes $\Theta(n)$ time in the worst case.
- UNION(x, y) walks to the tail of S_x and to the head of S_y and concatenates the two lists together. This takes $\Theta(n)$ time in the worst case.

3.3 Augmenting the Linked List

We can improve the behavior of our linked list solution by augmenting the linked lists such that each node has a pointer to its representative and that we also keep track of the tail of the list and the number of elements in the list at any time, which we will call its *weight*.

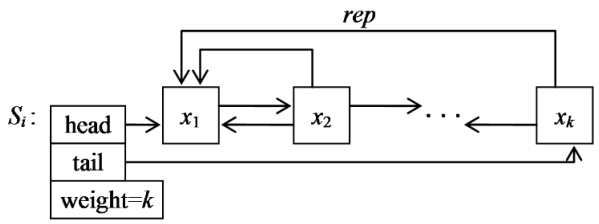


Figure 2: An augmented linked list.

Now, FIND-SET(x) can run in $\Theta(1)$ time.

We also change the behavior of UNION(x, y) to concatenate the lists containing x and y, updating the rep pointers for all the elements in y.

However, this could still require $\Theta(n)$ time in the worst case! Imagine if we called MAKE-SET for every integer between 1 and n and then called UNION$(n-1, n)$, UNION$(n-2, n-1)$, ..., UNION$(1, 2)$, where at the ith union we modify a list of length i. The total cost of all the calls to UNION is $1 + 2 + \ldots + (n-1) = \Theta(n^2)$.

3.4 First Improvement: Smaller into Larger

Our first improvement is to merge the smaller list into the larger list when calling UNION. If we do this, the above scenario will only modify a list of length 1 every time, resulting in $\Theta(n)$ running time for these $n - 1$ UNIONs.

Claim. With the first improvement, the amortized running time of n calls to MAKE-SET followed by n calls to UNION is $O(n \lg n)$.

Proof. [Aggregate Method]

Suppose the cost of moving a rep pointer is 1. Monitor some element x and the set S_x that contains it. After MAKE-SET(x), we have $|S_x| = 1$. When we call UNION(x, y), one of the following will happen:

- If $|S_x| > |S_y|$, then $rep[x]$ stays unchanged, so we need not pay anything on x's behalf, and $|S_x|$ will only increase.
- If $|S_x| \leq |S_y|$, we pay 1 to update $rep[x]$, and $|S_x|$ at least doubles.

S_x can double at most $\lg n$ times, so we update $rep[x]$ at most $\lg n$ times. Therefore across all the n elements, we get an amortized running time of $O(n \lg n)$. □

Proof. [Accounting Method]

The proof is very similar using the accounting method. When we call UNION(x, y), suppose $|S_x| < |S_y|$ without loss of generality. We will charge every element $i \in S_x$ because their pointers $rep[i]$'s are updated. Similarly, each element i can be charged at most $\lg n$ times, because $|S_i|$ at least doubles.

If we use the accounting method, each MAKE-SET(x) stores $\lg n$ coins into the bank for use in future UNION. □

Proof. [Potential Method]

In this case, we define the potential function to be

$$\Phi = \sum_i \lg n - \lg |S_i| = \sum_i \lg \frac{n}{|S_i|}$$

where the sum is taken over every element i in the structure (n is an upper bound on the total number of elements).

When we call MAKE-SET(x), $|S_x| = 1$, so the amortized cost is $O(1) + \Delta\Phi = O(\lg n)$.

When we call UNION(x, y), again suppose $|S_x| < |S_y|$. The actual cost is $|S_x|$. The change in potential is $\Delta\Phi \leq -|S_x|$, because every element in $i \in S_x$ now has $|S_i|$ at least doubled. This means we withdraw $|S_x|$ coins from the bank to pay for this UNION operation. So the amortized cost of UNION is $|S_x| + \Delta\Phi \leq 0$. □

3.5 Forest of Trees Representation

One interesting observation we can make is that we don't really care about the links between nodes in the linked list; we only care about the rep pointers. Essentially, a list can be represented as a forest of trees, where $rep[x]$ will be the root of the tree that contains x:

The algorithms will now be as follows:

- MAKE-SET(x) initializes x as a lone node. This takes $\Theta(1)$ time in the worst case.
- FIND-SET(x) climbs the tree containing x to the root. This takes $\Theta(height)$ time.

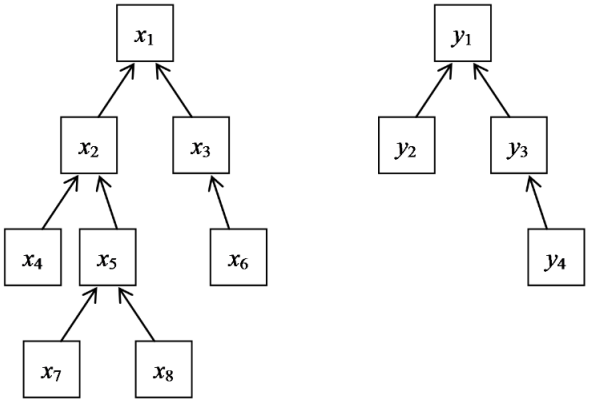


Figure 3: A forest of trees representation of a disjoint set data structure.

- UNION(x, y) climbs to the roots of the trees containing x and y and merges sets the parent of $rep[y]$ to $rep[x]$. This takes $\Theta(height)$ time.

3.5.1 Adapting the First Improvement

Our first trick can be modified to fit the forest-of-trees representation by merging the tree with the smaller height into the tree with the bigger height. One can then show that the height of a tree will still be $O(\lg n)$.

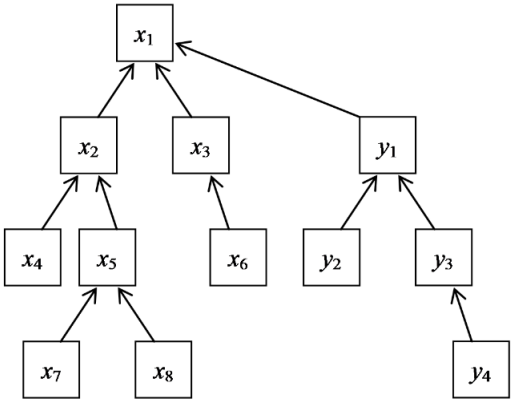


Figure 4: The data structure after calling UNION(x_1, y_1).

3.6 Second Improvement: Path Compression

Let's now improve FIND-SET. When we climb the tree, we learn the representatives of every intermediate node we pass. Therefore we should redirect the rep pointers so a future call to FIND-SET won't have to do the same computations multiple times.

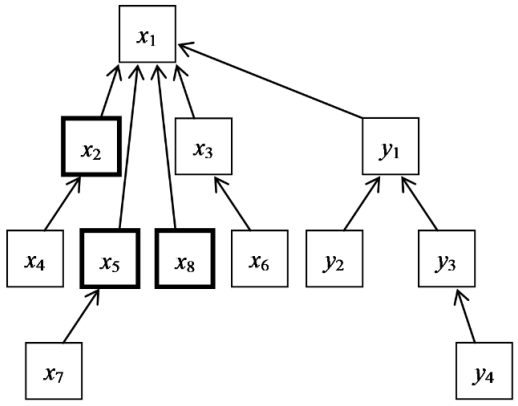


Figure 5: The data structure after calling FIND-SET(x_8).

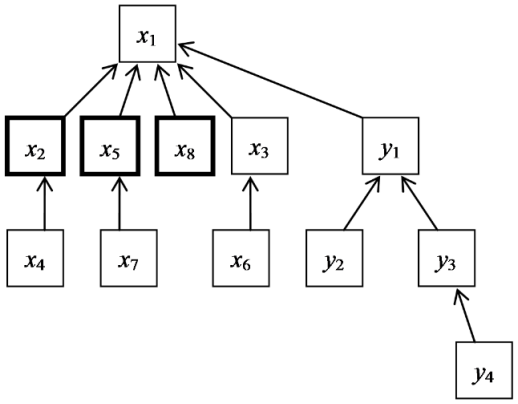


Figure 6: The same tree as in Figure 5, but redrawn. Notice that the x_i's are much more compact than Figure 4.

Claim. Let n be the total number of elements we're keeping track of. With the second improvement alone, the total running time of m operations (including FIND-SET) is $O(m \lg n)$. Therefore, the amortized cost per operation is $O(\lg n)$.

Proof.

Amortized analysis using the potential method. Let us define $U[x_i]$ to be the number of elements in the subtree rooted at x_i. Let $\Phi(x_1, \ldots, x_n) = \sum_i \lg U[x_i]$, where the x_i are

precisely those nodes that have been made. We will show that the amortized cost for each type of operation is $O(\lg(n))$.

3.6.1 Validity of the potential function

First we verify that the potential function is valid—that is, that it preserves the relationship

$$\sum \text{amortized costs} \geq \sum \text{actual costs}.$$

Writing the amortized cost of the i^{th} operation as $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$, and observing the telescoping nature of this sum (i.e., that each Φ_i is cancelled out by that in the amortized cost for the next operation, apart from the first and last terms), we have

$$\sum \hat{c}_i = \sum c_i + \Phi_{\text{final}} - \Phi_{\text{initial}} = \sum c_i + \Delta\Phi.$$

We have $\Phi_{\text{initial}} = 0$ trivially because no sets have been made, so it only remains to show that Φ is always nonnegative, since this guarantees

$$\sum \hat{c}_i = \sum c_i + \Phi_{\text{final}} - \Phi_{\text{initial}} \geq \sum c_i.$$

This is relatively straightforward, as $U[x_i] \geq 1$ for all x_i, meaning that

$$\Phi(x_1, \ldots, x_n) = \sum_i \lg U[x_i] \geq 0,$$

as desired.

3.6.2 MAKE-SET analysis

Note that after k MAKE-SET operations, the potential $\Phi(x_1, \ldots, x_n)$ remains unchanged because each new subtree made has only one element, namely itself, and the new $\log(1)$ term in the sum for the potential $\Phi(x_1, \ldots, x_n) = \sum_i \lg U[x_i]$ is equal to zero.

Therefore, the amortized cost is related to the total cost by

$$\hat{c}_i = c_i + \Delta\Phi,$$

where c_i is $O(1)$ and $\Delta\Phi = 0$, giving that $\hat{c}_i$ is also $O(1)$, as desired.

3.6.3 UNION analysis

Recall that UNION consists of a FIND-SET for each element, followed by a linking of the two representatives. Therefore, we can write decompose the total cost into the sum of the total cost of each FIND-SET and the total cost for the linking. We can treat the change in potential similarly. If the two elements are x_j and x_k, then this can be represented by

$$\begin{aligned}
\hat{c}_i &= c_i + \Delta\Phi \\
&= (c_{x_j,\text{FIND-SET}} + c_{x_k,\text{FIND-SET}} + c_{\text{LINKING}}) + (\Delta\Phi_{x_j,\text{FIND-SET}} + \Delta\Phi_{x_k,\text{FIND-SET}} + \Delta\Phi_{\text{LINKING}}) \\
&= (c_{x_j,\text{FIND-SET}} + \Delta\Phi_{x_j,\text{FIND-SET}}) + (c_{x_k,\text{FIND-SET}} + \Delta\Phi_{x_k,\text{FIND-SET}}) + c_{\text{LINKING}} + \Delta\Phi_{\text{LINKING}} \\
&= \hat{c}_{x_j,\text{FIND-SET}} + \hat{c}_{x_k,\text{FIND-SET}} + c_{\text{LINKING}} + \Delta\Phi_{\text{LINKING}}
\end{aligned}$$

We will see that the amortized cost of FIND-SET is $O(\lg n)$, so it only remains to show that $c_{\text{LINKING}} + \Delta\Phi_{\text{LINKING}}$ is $O(\lg n)$.

Thus, let us only analyze the cost of the LINKING portion of the UNION for now.

We know that $c_{\text{LINKING}} = 1$ because this step only entails updating a single pointer.

Now, we need only consider $\Delta\Phi_{\text{LINKING}}$.

Suppose that the root of the tree containing x_j discovered with FIND-SET(x_j) is r_j and the root of the tree containing x_k discovered with FIND-SET(x_k) is r_k. Without loss of generality, suppose that UNION(x_j, x_k) attaches r_k as a child of r_j.

Linking only increases U for r_j, as it changes from $U[r_j]$ to $U[r_j] + U[r_k]$. Thus, the difference in potential is given by

$$\begin{aligned}
\Delta\Phi &= \lg(U[r_j] + U[r_k]) - \lg(U[r_k]) \\
&= \lg \frac{U[r_j] + U[r_k]}{U[r_k]}.
\end{aligned}$$

Because there are n nodes, the sum $U[r_j] + U[r_k]$ is at most n, and thus, the last expression demonstrates that $\Delta\Phi$ is bounded above by $\lg n$.

Therefore, assuming that the amortized cost of FIND-SET is $O(\lg n)$, we have that the amortized cost of UNION is also $O(\lg n)$.

3.6.4 FIND-SET analysis

Finally, we consider the amortized cost of the FIND-SET(x_j) operation with path compression. As in the amortized analysis for FIND-SET, let us call the root of the tree containing

x_j by r_j.

As before, we have $\hat{c}_i = c_i + \Delta\Phi$, so we would like to find the actual cost c_i and the change in potential $\Delta\Phi$ for this operation.

At each step of our FIND-SET algorithm, we trace up the tree from x_j by looking from up the point from a child to its parent until we reach the root of the tree r_j, which is the representative for this set. Suppose that x_j is at a depth h in the tree. Then this process, as well as the process of redirecting pointers for path compression, has a cost of h.

Let us enumerate the nodes traversed when tracing up from x_j to r_j by $r_j = v_0, v_1, \ldots, v_h = r_j$. During the operation, we redirect each of $v_0, v_1, \ldots, v_{h-1}$ to point to v_h. Assume, without loss of generality, that we start redirecting v_{h-1}, then v_{h-2}, etc. The total change in potential is the sum of the change of potential after each redirection. We note that v_{h-1} already points to v_h. When we redirect v_{h-2}, only v_{h-1}'s size changes, and specifically, it changes from $U[v_{h-1}]$ to $U[v_{h-1}] - U[v_{h-2}]$. Similarly, when we redirect v_l, only $U[v_{l+1}]$ changes, and specifically, it changes from $U[v_{l+1}]$ to $U[v_{l+1}] - U[v_l]$.

Therefore, the overall change in potential is given by

$$\Delta\Phi = \sum_{l=1}^{h-2} (\lg(U[v_{l+1}] - U[v_l]) - \lg U[v_{i+1}]) = \sum_{l=1}^{h-2} \lg \frac{U[v_{l+1}] - U[v_l]}{U[v_{i+1}]}.$$

It follows that the amortized cost is given by

$$\begin{aligned}
\hat{c}_i &= c_i + \Delta\Phi \\
&= h + \sum_{l=1}^{h-2} \lg \frac{U[v_{l+1}] - U[v_l]}{U[v_{l+1}]} \\
&= 2 + \sum_{l=1}^{h-2} \left(1 + \lg \frac{U[v_{l+1}] - U[v_l]}{U[v_{l+1}]}\right).
\end{aligned}$$

Now, we note that because $U[v_l]$ and $U[v_{l+1}]$ are non-negative for any l, the function inside the logarithm is less than 1, implying that the logarithm is negative. Thus, the i^{th} term in the summation will be positive if, and only if, the logarithm is greater than -1. This happens if, and only if, $U[v_{l+1}] \geq 2U[v_l]$. However, this implies a doubling of the size of the tree at this level, but the tree's size is bounded above by the total number of nodes n. Thus, only $\lg n$ such doublings are possible.

It follows that at most $\lg n$ of the terms in the summation can be positive, and because the logarithm is negative, each of these terms is bounded above by 1. Thus, the sum is bounded above by $\lg n$.

Finally, this implies that the amortized cost $\hat{c}_i$ of FIND-SET is $O(\lg n)$, as desired. □

3.7 Combining Both Improvements

If we do both improvements to the tree representation, we get spectacular behavior where the amortized cost of each operation is almost constant!

We define a function, $A_k(j)$, with a similar structure to the well-known Ackermann function, as follows:

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$$

where $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$. This function explodes very quickly:

$$\begin{aligned} A_0(1) &= 2 \\ A_1(1) &= 3 \\ A_2(1) &= 7 \\ A_3(1) &= 2047 \\ A_4(1) &\gg 2^{2048} \end{aligned}$$

Now consider its "inverse" as follows:

$$\alpha(n) = \min\{k : A_k(1) \geq n\}$$

While not technically constant, this value is pretty darn close, even for very large values of n. For practical purposes, you can easily assume that $\alpha(n) \leq 4$.

Theorem. With both improvements, the total running time of m operations is $O(m\alpha(n))$.

The proof is very long and very tricky. If you're interested, check out Section 21.4 in CLRS.