

Amortized Analysis

Jared Saia

University of New Mexico

Outline

Amortized Analysis

- Aggregate, Taxation and Potential Methods
- Dynamic Table
- Union Find Data Structure

Amortized Analysis

“I will gladly pay you Tuesday for a hamburger today” - Wellington Wimpy

- In amortized analysis, time required to perform a sequence of data structure operations is averaged over all the operations performed
- Typically used to show that the average cost of an operation is small for a sequence of operations, even though a single operation can cost a lot

Amortized Analysis

- Average-case analysis bounds expected cost under an input distribution.
- Amortized analysis bounds total cost for every sequence of operations.
- No probability distribution is required.

Types of Amortized Analysis

- *Aggregate Analysis*
- *Accounting or Taxation Method*
- *Potential method*
- We'll see each method used for 1) a stack with the additional operation MULTIPOP and 2) a binary counter
- Our examples start with an empty stack or a counter at 0.

Aggregate Analysis

- We get an upper bound $T(n)$ on the total cost of any sequence of n operations. An amortized upper bound per operation is $T(n)/n$.

Stack with Multipop

- Recall that a standard stack has the operations PUSH and POP
- Each of these operations runs in $O(1)$ time, so let's say the cost of each is 1
- Now for a stack S and number k , let's add the operation MULTIPOP which removes the top k objects on the stack
- Multipop just calls Pop either k times or until the stack is empty

Multipop

- Q: What is the running time of $\text{Multipop}(S, k)$ on a stack of s objects?
- A: The cost is $\min(s, k)$ pop operations
- If there are n stack operations, in the worst case, a single Multipop can take $O(n)$ time

Multipop Analysis

- Let's analyze a sequence of n push, pop, and multipop operations on an initially empty stack
- The worst case cost of a multipop operation is $O(n)$ since the stack size is at most n , so the worst case time for any operation is $O(n)$
- Hence a sequence of n operations costs $O(n^2)$

The Problem

- This analysis is technically correct, but overly pessimistic
- While some of the multipop operations can take $O(n)$ time, not all of them can
- We need some way to average over the entire sequence of n operations

Aggregate Analysis

- In fact, the total cost of n operations on an initially empty stack is $O(n)$
- Why? Because each object can be popped at most once for each time that it is pushed
- Hence the number of times POP (including calls within Multipop) can be called on a nonempty stack is at most the number of Push operations which is $O(n)$

Aggregate Analysis

- Hence for any value of n , any sequence of n Push, Pop, and Multipop operations on an initially empty stack takes $O(n)$ time
- The average cost of an operation is thus $O(n)/n = O(1)$
- Thus all stack operations have an *amortized* cost of $O(1)$

Another Example

Another example where we can use aggregate analysis:

- Consider the problem of creating a k bit binary counter that counts upward from 0, wrapping to 0 on overflow
- We use an array $A[0..k-1]$ of bits as the counter
- A binary number x that is stored in A has its lowest-order bit in $A[0]$ and highest order bit in $A[k-1]$ ($x = \sum_{i=0}^{k-1} A[i] * 2^i$)

Binary Counter

- Initially $x = 0$ so $A[i] = 0$ for all $i = 0, 1, \dots, k - 1$
- To add 1 to the counter, we use a simple procedure which scans the bits from right to left, zeroing out 1's until it finds a zero bit to set to 1, or reaches the end

Increment

Increment(A)

$i \leftarrow 0$

while $i < k$ **and** $A[i] = 1$ **do**

$A[i] \leftarrow 0$

$i \leftarrow i + 1$

if $i < k$ **then**

$A[i] \leftarrow 1$

Analysis

- It's not hard to see that in the worst case, the increment procedure takes time $\Theta(k)$
- Thus a sequence of n increments takes time $O(nk)$ in the worst case
- Note that again this bound is correct but overly pessimistic
 - not all bits flip each time increment is called!

Aggregate Analysis

- In fact, we can show that a sequence of n calls to Increment has a worst case time of $O(n)$
- $A[0]$ flips every time Increment is called, $A[1]$ flips over every other time, $A[2]$ flips over every fourth time, ...
- Thus if there are n calls to increment, $A[0]$ flips n times, $A[1]$ flips $\lfloor n/2 \rfloor$ times, $A[2]$ flips $\lfloor n/4 \rfloor$ times

Aggregate Analysis

- Bit $A[i]$ flips $\left\lfloor n/2^i \right\rfloor$ times in n increments of an initially zero k -bit counter.
- Thus the total number of flips is

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{k-1} 2^{-i} < 2n.$$

Aggregate Analysis

- Thus the worst-case time for a sequence of n Increment operations on an initially zero counter is $O(n)$
- The average cost of each operation in the worst case then is $O(n)/n = O(1)$

Accounting or Taxation Method

- The second method is called the accounting method in the book, but a better name might be the *taxation* method
- Suppose it costs us a dollar to do a Push or Pop
- We can then measure the run time of our algorithm in dollars (Time is money!)

Taxation Method for Multipop

- Instead of paying for each Push and Pop operation when they occur, let's tax the pushes to pay for the pops
- I.e. we tax the push operation 2 dollars, and the pop and multipop operations 0 dollars
- Then each time we do a push, we spend one dollar of the tax to pay for the push and then *save* the other dollar of the tax to pay for the later pop or multipop of that item
- Note that if we do n operations, the total amount of taxes we collect is at most $2n$

Taxation Method

- Like any good government (ha ha) we need to make sure that: 1) our taxes are low and 2) we can use our taxes to pay for all our costs
- We already know that our taxes for n operations are no more than $2n$ dollars
- We now want to show that we can use the 2 dollars we collect for each push to pay for all the push, pop and multipop operations

Taxation Method

- This is easy to show. When we do a push, we use 1 dollar of the tax to pay for the push and then store the extra dollar with the item that was just pushed on the stack
- Then all items on the stack will have one dollar stored with them
- Whenever we do a Pop, we can use the dollar stored with the item popped to pay for the cost of that Pop
- Moreover, whenever we do a Multipop, for each item that we pop off in the Multipop, we can use the dollar stored with that item to pay for the cost of popping that item

Taxation Method

- We've shown that we can use the 2 tax on each item pushed to pay for the cost of all pops, pushes and multipops.
- Moreover we know that this taxation scheme collects at most $2n$ dollars in taxes over n stack operations
- Hence we've shown that the amortized cost per operation is $O(1)$

Taxation: Binary Counter

- Let's now use the taxation method to show that the amortized cost of the Increment algorithm is $O(1)$
- Let's say that it costs us 1 dollar to flip a bit
- What is a good taxation scheme to ensure that we can pay for the costs of all flips but that we keep taxes low?

Taxation Scheme

- Let's tax the algorithm 2 dollars to set a bit to 1
- Now we need to show that: 1) this scheme has low total taxes and 2) we will collect enough taxes to pay for all of the bit flips
- Showing overall taxes are low is easy: Each time Increment is called, it sets at most one bit to a 1
- So we collect at most 2 dollars in taxes each time increment is called
- Thus over n calls to Increment, we collect at most $2n$ dollars in taxes

Taxation Scheme

- We now need to show that our taxation scheme has enough money to pay for the costs of all operations
- This is easy: Each time we set a bit to a 1, we collect 2 dollars in tax. We use one dollar to pay for the cost of setting the bit to a 1, then we *store* the extra dollar on that bit
- When the bit gets flipped back from a 1 to a 0, we use the dollar already on that bit to pay for the cost of the flip!

Binary Counter

- We've shown that we can use the 2 tax each time a bit is set to a 1 to pay for all operations which flip a bit
- Moreover we know that this taxation scheme collects at most $2n$ dollars in taxes over n calls to Increment
- Hence we've shown that the amortized cost per call to Increment is $O(1)$

In Class Exercise

- A sequence of Pushes and Pops is performed on a stack whose size never exceeds k
- After every k operations, a copy of the entire stack is made for backup purposes, which takes $O(k)$ time.
- Show that the cost of n stack operations is $O(n)$

Exercise

- Q1: What is your taxation scheme?
- Q2: What is the maximum amount of taxes this scheme collects over n operations?
- Q3: Show that your taxation scheme can pay for the costs of all operations

Potential Method

- The most powerful method (and hardest to use)
- Builds on the idea from physics of potential energy
- Instead of associating taxes with particular operations, represent prepaid work as a *potential* that can be spent on later operations
- Potential is a function of the entire data structure

Potential Function

- Let D_i denote our data structure after i operations
- Let Φ_i denote the potential of D_i
- Let c_i denote the cost of the i -th operation (this changes D_{i-1} into D_i)
- Then the amortized cost of the i -th operation, a_i , is defined to be the actual cost plus the change in potential:

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

Potential Method

- So the *total* amortized cost of n operations is the actual cost plus the change in potential:

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

Potential Method

- Our task is to define a potential function so that
 1. $\Phi_0 = 0$
 2. $\Phi_i \geq 0$ for all i
- If we do this, the total *actual* cost of any sequence of operations will be at most the total amortized cost

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

Binary Counter Example

- For the binary counter, we can define the potential Φ_i after the i -th *Increment* operation to be the number of bits with value 1
- Initially all bits are 0 so $\Phi_0 = 0$, further $\Phi_i \geq 0$ for all $i > 0$, so this is a legal potential function

Binary Counter

- 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.

$$c_i = \#(0 \rightarrow 1) + \#(1 \rightarrow 0)$$

$$\Phi_i - \Phi_{i-1} = \#(0 \rightarrow 1) - \#(1 \rightarrow 0)$$

- Thus, the amortized cost of the i th Increment is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \cdot \#(0 \rightarrow 1)$$

Binary Counter

- Since Increment changes at most one bit from a 0 to a 1, the amortized cost of Increment is at most 2 (using this potential function)
- Recall that for a legal potential function, $\sum_{i=1}^n c_i \leq \sum_{i=1}^n a_i$ thus the total cost for n calls to increment is no more than $2n$
- (Amortized cost at most 2)

Potential Method Recipe

1. Choose Φ with $\Phi_0 = 0$ and $\Phi_i \geq 0$.
2. Bound $a_i = c_i + \Phi_i - \Phi_{i-1}$ for each operation.
3. Sum and telescope: $\sum_{i=1}^n c_i \leq \sum_{i=1}^n a_i$.

Binary Counter Example

- For the binary counter, the potential was exactly the total unspent taxes paid using the taxation method
- So it gave us the same amortized bound
- Potential tracks credit for the entire data structure.

A Good Potential Function

- Different potential functions lead to different amortized time bounds
- Choose a potential that pays for expensive operations.
- Save a little on cheap operations; spend the savings on expensive ones.
- Look for stored work: items to pop, bits to reset, or elements to copy.

Stack Example

- Consider again a stack with Multipop
- Define the potential function Φ on the stack to be the number of objects on the stack
- This potential function is “legal” since $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all $i > 0$

Push

- Let's now compute the costs of the different stack operations on a stack with s items
- If the i -th operation on the stack is a push operation on a stack containing s objects, then

$$\Phi_i - \Phi_{i-1} = (s + 1) - s = 1$$

- So $a_i = c_i + 1 = 2$

Multipop

- Let the i -th operation be $\text{Multipop}(S, k)$ and let $k' = \min(k, s)$ be the number of objects popped off the stack. Then

$$\Phi_i - \Phi_{i-1} = (s - k') - s = -k'.$$

- Further $c_i = k'$.
- Thus,

$$a_i = -k' + k' = 0$$

- (We can show similarly that the amortized cost of a pop operation is 0)

Wrapup

- The amortized cost of each of these three operations is $O(1)$
- Thus the worst case cost of n operations is $O(n)$

Dynamic Arrays

- Store items in an array with constant-time indexed access.
- Table-Insert appends; Remove-Last removes the last item.
- Resize by copying items when the array becomes full or too sparse.
- Goal: $O(1)$ amortized cost per operation and space proportional to the number of items.

Load Factor

- $T.num$ is the number of items; $T.size$ is the capacity.
- For a nonempty table, $\alpha(T) = T.num/T.size$.
- An empty table has capacity 0; define its load factor to be 1.
- If $\alpha(T) \geq c$, at most a fraction $1 - c$ of its slots are unused.

Table Expansion

- On insertion into a full table, double its capacity and copy its items.
- The first insertion allocates one slot.
- With insertions only, the table is always at least half full.

Table-Insert

Table-Insert(T, x)

if $T.size = 0$ **then**

 allocate $T.table$ with one slot

$T.size \leftarrow 1$

if $T.num = T.size$ **then**

 allocate $newTable$ with $2T.size$ slots

 copy the $T.num$ items into $newTable$

 free $T.table$

$T.table \leftarrow newTable$

$T.size \leftarrow 2T.size$

$T.table[T.num] \leftarrow x$

$T.num \leftarrow T.num + 1$

Insertion Costs

- Count one unit for an insertion and one for each item copied.
- Insertion without expansion costs 1.
- Expansion from capacity s costs $s + 1$, including the new item.

Aggregate Analysis

For $n \geq 1$ insertions into an initially empty table:

$$\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2^{\lfloor \log n \rfloor + 1} \leq 3n.$$

- The copies form a geometric series.
- The amortized cost per insertion is at most 3.

Taxation Method

- Charge each insertion 3: spend 1 and save 2.
- After doubling to capacity s , copying leaves $s/2$ items.
- The next $s/2$ insertions, including the triggering insertion, save s credits for the next expansion.
- The first insertion saves enough to pay for the first expansion.

Potential for Insertions

Let num_i and $size_i$ be the item count and capacity after operation i .

$$\Phi_i = 2num_i - size_i.$$

- Initially $\Phi_0 = 0$.
- The table is at least half full, so $\Phi_i \geq 0$.

In-Class Exercise

Use $a_i = c_i + \Phi_i - \Phi_{i-1}$ to show that insertion costs at most 3 amortized.

1. First insertion into an empty table.
2. Insertion without expansion.
3. Insertion into a full table of capacity s .

Remove-Last

- On a nonempty table, remove $T.table[T.num - 1]$ and decrement $T.num$.
- If no items remain, free the array and set $T.size = 0$.
- Otherwise, we may shrink the array when it becomes sparse.

A Naive Resizing Rule

- Double capacity when inserting into a full table.
- Halve capacity when removal leaves the table less than half full.
- The table stays at least half full, but may resize too often.

Resizing Too Often

Start with N items in a full table, where N is a power of 2.

Operation	Items	Capacity	Resize
Insert	$N + 1$	$2N$	Expand
Remove-last	N	$2N$	None
Remove-last	$N - 1$	N	Contract
Insert	N	N	None
Insert	$N + 1$	$2N$	Expand

Repeating the pattern causes $\Theta(N)$ expansions and contractions, each costing $\Theta(N)$: $\Theta(N^2)$ total work in $\Theta(N)$ operations.

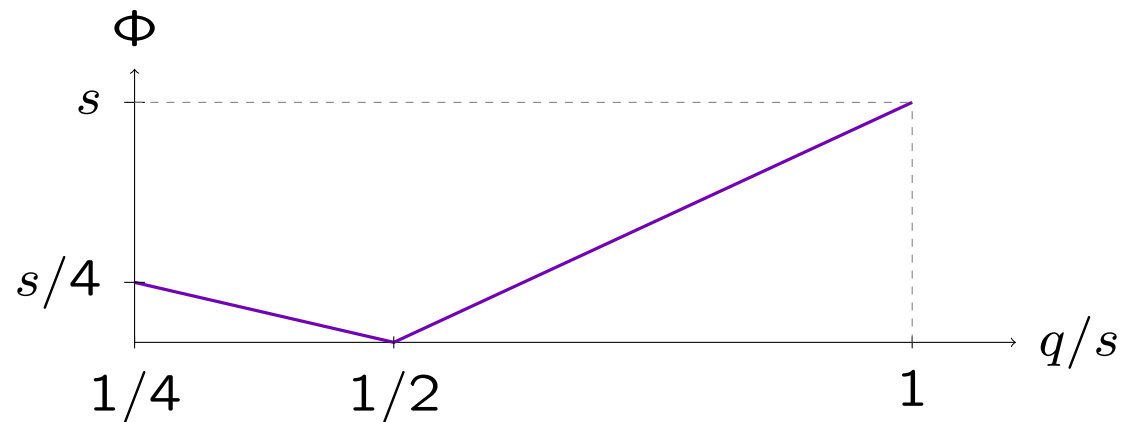
Leave Room Between Resizes

- Still double capacity on insertion into a full table.
- Halve capacity only when removal leaves it less than $1/4$ full.
- Many operations now separate consecutive resizes.
- A nonempty table stays at least $1/4$ full.

Potential for Resizing

For item count q and capacity s , use

$$\Phi(q, s) = \begin{cases} 2q - s & q \geq s/2, \\ s/2 - q & q < s/2. \end{cases}$$



For fixed $s > 0$, potential is zero at half full. After each operation, $1/4 \leq q/s \leq 1$ for a nonempty table. For an empty table, $\Phi(0, 0) = 0$.

Amortized Costs

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

Operation	Actual cost	Amortized cost
Insert, no resize	1	≤ 3
Insert, expansion	$s + 1$	3
Remove-last, no resize	1	≤ 2
Remove-last, contraction	$1 + q_{\text{after}}$	≤ 2

The first insertion and removal of the last item also cost at most 2 amortized.

Expansion Calculation

Before insertion: $q = s$. After doubling and inserting: $q' = s + 1$, $s' = 2s$.

$$\Phi_{\text{before}} = s, \quad \Phi_{\text{after}} = 2(s + 1) - 2s = 2.$$

$$a_i = (s + 1) + 2 - s = 3.$$

The drop in potential pays for copying the old items.

Contraction Calculation

For capacity $s \geq 8$, removal triggers contraction from $q = s/4$ to $q' = s/4 - 1$, with new capacity $s' = s/2$.

$$c_i = 1 + q' = s/4, \quad \Phi_{\text{before}} = s/4, \quad \Phi_{\text{after}} = s/4 - q' = 1.$$

$$a_i = s/4 + 1 - s/4 = 1.$$

Example: Start with $q = 4$ items and capacity $s = 16$.

Remove-last leaves $q' = 3$; shrink to $s' = 8$.

Actual cost: 4. Potential: $4 \rightarrow 1$. Amortized cost: $4 + 1 - 4 = 1$.

Removing the last item frees the table and costs at most 1 amortized.

Dynamic-Array Takeaway

- Append and remove-last each cost $O(1)$ amortized.
- Any sequence of n operations from an empty table costs $O(n)$.
- For a nonempty table, capacity is at most four times the item count.

Union-Find

- Union-Find is our last application of amortized analysis
- It maintains disjoint sets $S_1, S_2, \dots, S_k$.
- Each set is identified by a representative which is a member of that set

Operations

Union-find supports the following three operations:

- **Make-Set(x):** creates a new set whose only member (and representative) is x
- **Union(x,y):** unites the sets that contain x and y (call them S_x and S_y) into a new set that is $S_x \cup S_y$. The new set is added to the data structure while S_x and S_y are deleted. If the representatives agree, do nothing.
- **Find-Set(x):** Returns a pointer to the representative of the (unique) set containing x

Analysis

- We will analyze this data structure in terms of two parameters:
 1. n , the number of Make-Set operations
 2. m , the total number of Make-Set, Union, and Find-Set operations
- Since the sets are always disjoint, each successful merge reduces the number of sets by 1
- So after $n - 1$ successful merges, only one set remains
- Thus the number of successful merges is at most $n - 1$

Discussion

- Note also that since the Make-Set operations are included in the total number of operations, we know that $m \geq n$
- We will in general assume that the Make-Set operations are the first n performed

Connected Components

- Each vertex starts in its own set.
- When an edge (x, y) is added, call $\text{Union}(x, y)$.
- Vertices are connected exactly when their representatives agree.
- We will use this in Kruskal's minimum spanning tree algorithm.

Example

- Start with singleton sets for Bob, Sue, Jane, and Joe.
- Union("Bob" , "Joe")
there are now three sets $\{Bob, Joe\}, \{Jane\}, \{Sue\}$
- Union("Jane" , "Sue")
there are now two sets $\{Bob, Joe\}, \{Jane, Sue\}$
- Union("Bob" , "Jane")
there is now one set $\{Bob, Joe, Jane, Sue\}$

Discussion and Outline

- **We can implement Union-Find with amortized cost for all operations that is $O(\log^* n)$. ($\log^*$ is the iterated log function.)**
- Details of this implementation and analysis are given in the remainder of these slides.
- We will later see that Union Find is used in Kruskal's minimum spanning tree algorithm, and you'll see an application in the homework

First Approach: Unbalanced Trees

- One of the easiest ways to store sets is using trees.
- Each object points to another object, called its *parent*, except for the leader of each set, which points to itself and thus is the root of the tree.

Tree Implementation

- Make-Set is trivial (we just create one root node)
- Find-Set traverses the parent pointers up to the leader (the root node).
- Union just redirects the parent pointer of one leader to the other.

(Notice that unlike most tree data structures, objects do *not* have pointers down to their children.)

Algorithms

Make-Set(x)

$\text{parent}(x) \leftarrow x$

Find-Set(x)

while $x \neq \text{parent}(x)$ **do**

$x \leftarrow \text{parent}(x)$

return x

Union(x, y)

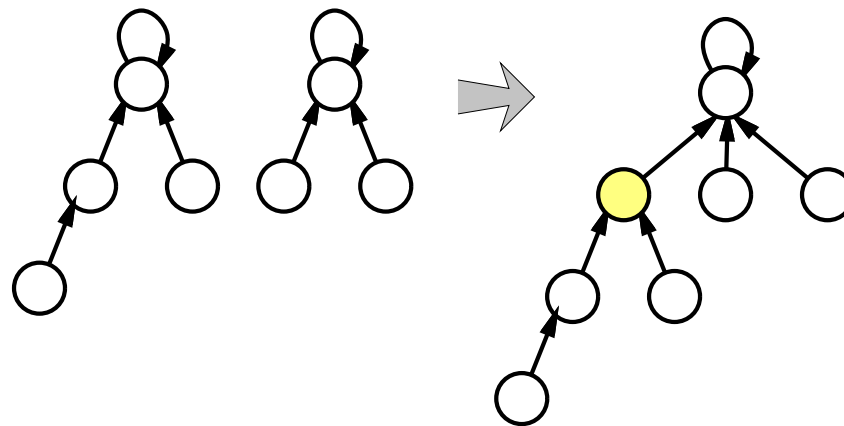
$xParent \leftarrow \text{Find-Set}(x)$

$yParent \leftarrow \text{Find-Set}(y)$

if $xParent = yParent$ **then return**

$\text{parent}(yParent) \leftarrow xParent$

Example



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

Analysis

- Make-Set takes $\Theta(1)$ time
- Union takes $\Theta(1)$ time in addition to the calls to Find-Set
- The running time of Find-Set is proportional to the depth of x in the tree. In the worst case, this could be $\Theta(n)$ time

Problem

- Problem: The running time of Find-Set is very slow
- Q: Is there some way to speed this up?
- A: Yes we can ensure that the heights of our trees remain small
- We can do this by using the following strategy when merging two trees: we make the root of the tree with fewer nodes a child of the tree with more nodes
- This means that we need to always store the number of nodes in each tree, but this is easy

The Code

Make-Set(x)

$\text{parent}(x) \leftarrow x, \quad \text{size}(x) \leftarrow 1$

Union(x, y)

$xRep \leftarrow \text{Find-Set}(x)$

$yRep \leftarrow \text{Find-Set}(y)$

if $xRep = yRep$ **then return**

if $\text{size}(xRep) > \text{size}(yRep)$ **then**

$\text{parent}(yRep) \leftarrow xRep$

$\text{size}(xRep) \leftarrow \text{size}(xRep) + \text{size}(yRep)$

else

$\text{parent}(xRep) \leftarrow yRep$

$\text{size}(yRep) \leftarrow \text{size}(yRep) + \text{size}(xRep)$

Analysis

- It turns out that for these algorithms, all the functions run in $O(\log n)$ time
- We will be showing this is the case in the In-Class exercise
- We will show this by showing that the heights of all the trees are always logarithmic in the number of nodes in the tree

In-Class Exercise

- We will show that the height of our trees are no more than $O(\log x)$ where x is the number of nodes in the tree
- We will show this using proof by induction on, x , the number of nodes in the tree
- We will consider a tree with x nodes and, using the inductive hypothesis (and facts about our algs), show that it has a height of $O(\log x)$

The Facts

- Let T be a tree with x nodes that was created by a call to the Union Algorithm
- Note that T must have been created by merging two trees $T1$ and $T2$
- Let $T2$ be the tree with the smaller number of nodes
- Then the root of T is the root of $T1$ and a child of this root is the root of the tree $T2$
- Key fact: the number of nodes in $T2$ is no more than $x/2$

In-Class Exercise

To prove: Any tree T with x nodes, created by our algorithms, has height no more than $\log x$

- Q1: Show the base case ($x = 1$)
- Q2: What is the inductive hypothesis?
- Q3: Complete the proof by giving the inductive step. (hint: note that $\text{height}(T) = \text{Max}(\text{height}(T1), \text{height}(T2) + 1)$)

Problem

- Q: $O(\log n)$ per operation is not bad but can we do better?
- A: Yes we can actually do much better but it's going to take some cleverness (and amortized analysis)

Shallow Threaded Trees

- One good idea is to just have every object keep a pointer to the leader of its set
- In other words, each set is represented by a tree of height 1
- Then Make-Set and Find-Set are completely trivial, and they both take $O(1)$ time
- Q: What about the Union operation?

Union

- To do a union, we need to set all the leader pointers of one set to point to the leader of the other set
- To do this, we need a way to visit all the nodes in one of the sets
- We can do this easily by “threading” a linked list through each set starting with the set’s leader
- The threads of two sets can be merged by the Union algorithm in constant time

The Code

Make-Set(x)

leader(x) $\leftarrow x$

next(x) $\leftarrow \text{nil}$

Find-Set(x)

return leader(x)

The Code

Union(x, y)

$xRep \leftarrow \text{Find-Set}(x)$

$yRep \leftarrow \text{Find-Set}(y)$

if $xRep = yRep$ **then return**

$y \leftarrow yRep$

$\text{leader}(y) \leftarrow xRep$

while $\text{next}(y) \neq \text{nil}$ **do**

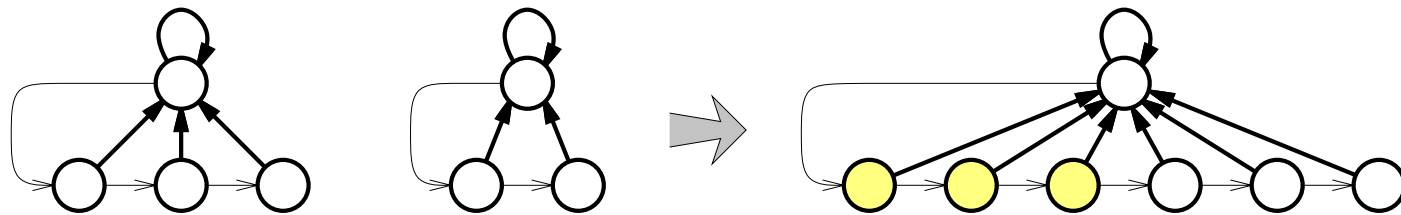
$y \leftarrow \text{next}(y)$

$\text{leader}(y) \leftarrow xRep$

$\text{next}(y) \leftarrow \text{next}(xRep)$

$\text{next}(xRep) \leftarrow yRep$

Example



Merging two sets stored as threaded trees.

Bold arrows point to leaders; lighter arrows form the threads.

Shaded nodes have a new leader.

Analysis

- Worst case time of Union is a constant times the size of the *larger* set
- So if we merge a one-element set with a n element set, the run time can be $\Theta(n)$
- In the worst case, it's easy to see that n operations can take $\Theta(n^2)$ time for this alg

Problem

- The main problem here is that in the worst case, we always get unlucky and choose to update the leader pointers of the larger set
- Instead let's purposefully choose to update the leader pointers of the smaller set
- This will require us to keep track of the sizes of all the sets, but this is not difficult

The Code

Make-Weighted-Set(x)

$\text{leader}(x) \leftarrow x$

$\text{next}(x) \leftarrow \text{nil}$

$\text{size}(x) \leftarrow 1$

The Code

Weighted-Union(x, y)

$xRep \leftarrow \text{Find-Set}(x)$

$yRep \leftarrow \text{Find-Set}(y)$

if $xRep = yRep$ **then return**

if $\text{size}(xRep) > \text{size}(yRep)$ **then**

$\text{Union}(xRep, yRep)$

$\text{size}(xRep) \leftarrow \text{size}(xRep) + \text{size}(yRep)$

else

$\text{Union}(yRep, xRep)$

$\text{size}(yRep) \leftarrow \text{size}(xRep) + \text{size}(yRep)$

Weighted-Union Analysis

- One union may take linear time.
- Let n be the number of elements and m the total number of operations.
- Weighted union with threaded lists takes $O(m + n \log n)$ total time.

Proof

- Whenever the leader of an object x is changed by a call to Weighted-Union, the size of the set containing x increases by a factor of at least 2
- Thus if the leader of x has changed k times, the set containing x has at least 2^k members
- After the sequence of operations ends, the largest set has at most n members
- Thus the leader of any object x has changed at most $\lfloor \log n \rfloor$ times

Proof

- There are n elements.
- Each element changes leaders at most $\lfloor \log n \rfloor$ times.
- Thus all leader-pointer updates cost $O(n \log n)$.

Proof

- Leader-pointer updates cost $O(n \log n)$ in total.
- All other work costs $O(1)$ per operation, hence $O(m)$.
- Total cost: $O(m + n \log n)$.

This is the aggregate method.

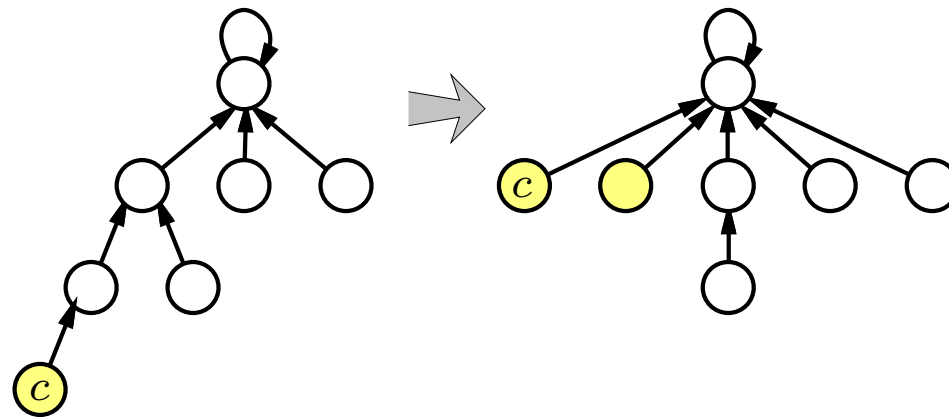
Analysis

- Using Balanced-Trees, *Find-Set* takes logarithmic worst case, *Make-Set* is constant and *Union* is constant except for the calls to *Find-Set*
- Using Shallow-Trees, *Find-Set* is constant and m operations cost $O(m + n \log n)$ total.
- A third method allows us to get both of these operations in *almost* constant amortized time

Path Compression

- We start with the unthreaded tree representation (with union by rank)
- Key Observation is that in any *Find-Set* operation, once we get the leader of an object x , we can speed up future Find-Set's by redirecting x 's parent pointer directly to that leader
- We can also change the parent pointers of all ancestors of x all the way up to the root (We'll do this using recursion)
- This modification to Find-Set is called path compression

Path Compression Example



Path compression during $\text{Find}(c)$. Shaded nodes have a new parent.

PC-Find-Set

PC-Find-Set(x)

if $x \neq \text{parent}(x)$ **then**

$\text{parent}(x) \leftarrow \text{PC-Find-Set}(\text{parent}(x))$

return $\text{parent}(x)$

Rank

- For ease of analysis, instead of keeping track of the size of each of the trees, we will keep track of the *rank*
- Each node will have an associated rank
- This rank will give an estimate of the log of the number of elements in the set

Code

PC-MakeSet(x)

$\text{parent}(x) \leftarrow x, \quad \text{rank}(x) \leftarrow 0$

PC-Union(x, y)

$xRep \leftarrow \text{PC-Find-Set}(x)$

$yRep \leftarrow \text{PC-Find-Set}(y)$

if $xRep = yRep$ **then return**

if $\text{rank}(xRep) > \text{rank}(yRep)$ **then**

$\text{parent}(yRep) \leftarrow xRep$

else

$\text{parent}(xRep) \leftarrow yRep$

if $\text{rank}(xRep) = \text{rank}(yRep)$ **then**

$\text{rank}(yRep) \leftarrow \text{rank}(yRep) + 1$

Rank Facts

- If an object x is not the set leader, then the rank of x is strictly less than the rank of its parent
- For a set X , $\text{size}(X) \geq 2^{\text{rank}(\text{leader}(X))}$ (can show using induction on the size of X)
- Since there are n objects, the highest possible rank is $O(\log n)$
- Only set leaders can change their rank

Rank Facts

Can also say that there are at most $n/2^r$ objects with rank r .

- When the rank of a set leader x changes from $r - 1$ to r , mark all nodes in its set. At least 2^r nodes are marked, and no other node with rank r will ever mark those nodes.
- There are n nodes total and any object with rank r marks 2^r of them
- Thus there can be at most $n/2^r$ objects of rank r

Rank Blocks

- Assign x to block $\log^*(\text{rank}(x))$.
- $\log^* x$ is 1+ the number of logs needed to reach a value at most 1.
- The first blocks contain ranks

$\{0, 1\}, \quad \{2\}, \quad \{3, 4\}, \quad \{5, \dots, 16\}, \dots$

└─ log* x definition ─┘

- log* x is the *iterated log* function

$$\log^* x = \begin{cases} 1 & \text{if } x \leq 1 \\ 1 + \log^*(\log x) & \text{if } x > 1 \end{cases}$$

$2 \uparrow\uparrow b$ definition

- $2 \uparrow\uparrow b$ is the *tower* function

$$2 \uparrow\uparrow b = 2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}} \Bigg\}_b = \begin{cases} 1 & \text{if } b = 0 \\ 2^{2 \uparrow\uparrow (b-1)} & \text{if } b > 0 \end{cases}$$

Number of Blocks

- Ranks range from 0 to $\lfloor \log n \rfloor$.
- Thus there are at most $\log^* n$ blocks.
- For $b \geq 2$, block b contains ranks

$$2 \uparrow\uparrow (b - 2) < r \leq 2 \uparrow\uparrow (b - 1).$$

Objects in a Block

There are at most $n/2^r$ objects of rank r . For $b \geq 2$, the number in block b is at most

$$\sum_{r=2^{\uparrow\uparrow(b-2)}+1}^{2^{\uparrow\uparrow(b-1)}} \frac{n}{2^r} < \sum_{r=2^{\uparrow\uparrow(b-2)}+1}^{\infty} \frac{n}{2^r} = \frac{n}{2^{\uparrow\uparrow(b-1)}}.$$

Block 1 contains at most n objects.

Theorem

- **Theorem:** If we use both PC-Find-Set and PC-Union (path compression and union by rank), the worst-case running time of a sequence of m operations, n of which are MakeSet operations, is $O(m \log^* n)$
- Each PC-MakeSet operation takes constant time. PC-Union takes constant time except for calls to PC-Find-Set. Thus, we need only show that the $O(m)$ calls to PC-Find-Set take $O(m \log^* n)$ total time in the worst case
- We will use a kind of accounting method to show this

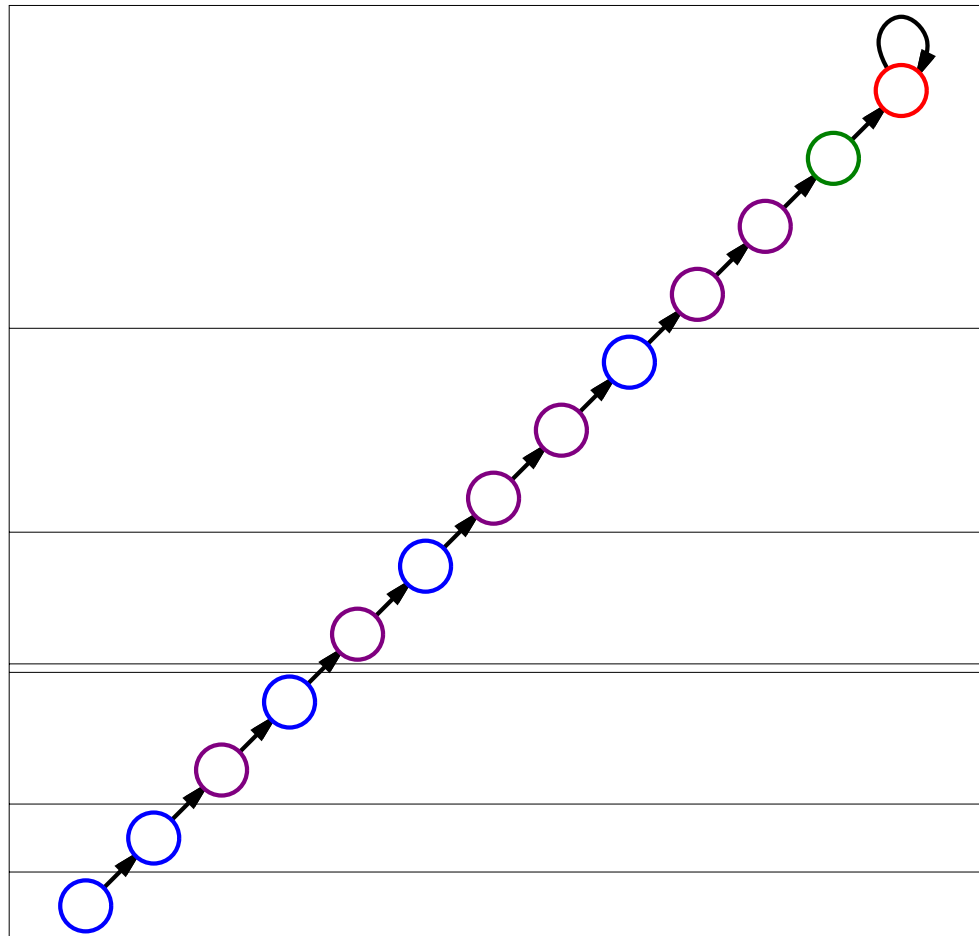
Proof

- The cost of $\text{PC-Find-Set}(x_0)$ is proportional to the number of nodes on the path from x_0 up to its leader
- Each object $x_0, x_1, x_2, \dots, x_l$ on the path from x_0 to its leader will pay a 1 tax into one of several bank accounts
- After all the Find operations are done, the total amount of money in these accounts will give us the total running time

Taxation

- The leader x_ℓ pays into the *leader* account.
- The child of the leader $x_{\ell-1}$ pays into the *child* account.
- Any other object x_i in a different block from its parent x_{i+1} pays into the *block* account.
- Any other object x_i in the same block as its parent x_{i+1} pays into the *path* account.

Example



Different nodes on the find path pay into different accounts: Red=Leader;
Green = Child; Purple = Path; Blue = Block.
Horizontal lines are boundaries between blocks. Only the nodes on the find
path are shown.

Three Easy Accounts

- During any Find operation, one dollar is paid into the leader account
- At most one dollar is paid into the child account
- At most one dollar is paid into the block account for each of at most $\log^* n$ blocks
- Across at most $2m$ Find calls, these accounts receive at most $4m + 2m \log^* n$ dollars

Path Account

- The only remaining difficulty is the Path account
- Consider an object x_i in block b that pays into the path account
- This object is not a set leader so its rank can never change.
- The parent of x_i is also not a set leader, so after path compression, x_i gets a new parent, x_ℓ , whose rank is strictly larger than its old parent x_{i+1}
- Since $rank(parent(x))$ is always increasing, parent of x_i must eventually be in a different block than x_i , after which x_i will never pay into the path account
- *Thus x_i pays into the path account at most once for every rank in block b , or less than $2 \uparrow \uparrow (b - 1)$ times for $b \geq 2$, and at most once for $b = 1$*

Path Account

- For $b \geq 2$, fewer than $n/(2 \uparrow\uparrow (b-1))$ objects each pay fewer than $2 \uparrow\uparrow (b-1)$ dollars: less than n per block.
- Block 1 also pays at most n dollars.
- With at most $\log^* n$ blocks, the path account receives at most $n \log^* n$ dollars.
- Since $n \leq m$, the total is at most

$$4m + 2m \log^* n + n \log^* n = O(m \log^* n).$$

Take Away

- We can now say that each call to PC-Find-Set and PC-Union have amortized cost $O(\log^* n)$, which is significantly better than the worst case cost of $O(\log n)$
- The book shows that PC-Find-Set and PC-Union have amortized cost of $O(A^{-1}(n))$ where $A^{-1}(n)$ (inverse Ackermann function) is an even slower growing function than $\log^* n$