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

Amortized analysis is *not* average case analysis.

- *Average Case Analysis*: the expected cost of each operation
- *Amortized analysis*: the average cost of each operation *in the worst case*
- Probability is not involved in amortized analysis

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
- Amortized analysis assumes the data structure starts in the initialized state (e.g. empty stack, binary counter at 0, etc)

Aggregate Analysis

- We get an upperbound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$, which is also the amortized cost per operation

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 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
- 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 $i = 0, 1, \ldots, k-1$
- 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 finally find a zero bit which it flips to a 1

Increment

```
Increment(A){
  i = 0;
  while(i<k && A[i]=1){
    A[i] = 0;
    i++;
  }
  if (i<k)
    A[i] = 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

- In general, for $i = 0, \ldots \lfloor \log n \rfloor$, bit $A[i]$ flips $\left\lfloor n/2^i \right\rfloor$ times in a sequence of n calls to Increment on an initially zero counter
- For $i > \lfloor \log n \rfloor$, bit $A[i]$ never flips
- Total number of flips in the sequence of n calls is thus

$$\sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} \quad (1)$$

$$= 2n \quad (2)$$

Aggregate Analysis

- Thus the worst-case time for a sequence of n Increment operations on an initially empty 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 inevitable pop or multipop of that item
- Note that if we do n operations, the total amount of taxes we collect is then $2n$

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 Method for 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 exactly 2 dollars in taxes each time increment is called
- Thus over n calls to Increment, we collect $2n$ 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 $2n$ 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
- Show that the cost of n stack operations, including copying the stack, is $O(n)$

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
- Show that the cost of n stack operations, including copying the stack, 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:

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

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 less than the total amortized cost

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

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.

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

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

- Thus, the amortized cost of the ith Increment is

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

Binary Counter

- Since Increment only changes one bit from a 0 to a 1, the amortized cost of Increment is 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 call to increment is no more than $2n$
- (Same as saying that the amortized cost is 2)

Potential Method Recipe

1. Define a potential function for the data structure that is 1) initially equal to zero and 2) is always nonnegative.
2. The amortized cost of an operation is its actual cost plus the change in potential.

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
- In general, however, there may be no way of interpreting the potential as "taxes"

A Good Potential Function

- Different potential functions lead to different amortized time bounds
- Trick to using the method is to get 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's no general technique for doing this other than trying lots of possibilities

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 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 Tables

- Consider the situation where we do not know in advance the number of items that will be stored in a table, but we want constant time access
- We might allocate a fixed amount of space for the table only to find out later that this was not enough space
- In this case, we need to copy over all objects stored in the original table into a new larger table
- Similarly, if many objects are deleted, we might want to reduce the size of the table

Dynamic Tables

- The data structure that we want is a Dynamic Table (aka Dynamic Array)
- We can show using amortized analysis that the amortized cost of an insertion and deletion into a Dynamic Table is $O(1)$ even though worst case cost may be much larger

Load Factor

- For a nonempty table T, we define the "load factor" of T, $\alpha(T)$, to be the number of items stored in the table divided by the size (number of slots) of the table
- We assign an empty table (one with no items) size 0 and load factor of 1
- Note that the load factor of any table is always between 0 and 1
- Further if we can say that the load factor of a table is always at least some constant c, then the unused space in the table is never more than $1 - c$

Table Expansion

- Assume that the table is allocated as an array
- A table is full when all slots are used i.e. when the load factor is 1
- When an insert occurs when the table is full, we need to expand the table
- The way we will do this is to allocate an array which is twice the size of the old array and then copy all the elements of the old array into this new larger array
- If only insertions are performed, this ensures that the load factor is always at least 1/2

Pseudocode

```
Table-Insert(T,x){
  if (T.size == 0){allocate T with 1 slot;T.size=1}
  if (T.num == T.size){
    allocate newTable with 2*T.size slots;
    insert all items in T.table into newTable;
    T.table = newTable;
    T.size = 2*T.size
    }
  T.table[T.num] = x;
  T.num++
}
```

Amortized Analysis

- Note that usually Table-Insert just does an "elementary" insert into the array
- However very occasionally it will do an "expansion". We will say that the cost of an expansion is equal to the size before the expansio occurs
- (This is the cost of moving over all the old elements to the larger table)

Aggregate Analysis

- Let c_i be the cost of the i-th call to Table-Insert.
- If $i-1$ is an exact power of 2, then we'll need to do an expansion and so $c_i = i$
- Otherwise, $c_i = 1$
- The total cost of n Table-Insert operations is thus

$$\begin{aligned}
\sum_{i=1}^{n} c_i &\leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \\
&< n + 2^{\lfloor \log n \rfloor + 1} \\
&= n + 2 \cdot 2^{\lfloor \log n \rfloor} \\
&\leq n + 2n \\
&= 3n
\end{aligned}$$

- Thus the amortized cost of a single operation is 3

Taxation method

- Every time Table-Insert is called, we tax the operation 3 dollars
- Intuitively, the item inserted pays for:
  1. its insertion
  2. moving itself when the table is eventually expanded
  3. moving some other item that has already been moved once when the table is expanded

Taxation Method

- Suppose that the size of the table is m right after an expansion
- Then the number of items in the table is $m/2$
- Each time Table-Insert is called, we tax the operation 3 dollars:
  1. One dollar is used immediately to pay for the elementary insert
  2. Another dollar is stored with the item that is inserted
  3. The third dollar is placed as credit on one of the $m/2$ items already in the table

Taxation Method

- Filling the table again requires $m/2$ total calls to Table-Insert
- Thus by the time the table is full and we do another expansion, each item will have one dollar of credit on it
- This dollar of credit can be used to pay for the movement of that item during the expansion

Potential Method

- Let's now analyze Table-Insert using the potential method
- Let num_i be the num value for the i-th call to Table-Insert
- Let $size_i$ be the size value for the i-th call to Table-Insert
- Then let

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

In Class Exercise

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

- Show that this potential function is 0 initially and always nonnegative
- Compute a_i for the case where Table-Insert does not trigger an expansion
- Compute a_i for the case where Table-Insert does trigger an expansion (note that $num_{i-1} = num_i - 1$, $size_{i-1} = num_i - 1$, $size_i = 2 * (num_i - 1)$)

Table Delete

- We've shown that a Table-Insert has $O(1)$ amortized cost
- To implement Table-Delete, it is enough to remove (or zero out) the specified item from the table
- However it is also desirable to contract the table when the load factor gets too small
- Storage for old table can then be freed to the heap

Desirable Properties

We want to preserve two properties:

- the load factor of the dynamic table is lower bounded by some constant
- the amortized cost of a table operation is bounded above by a constant

Naive Strategy

- A natural strategy for expansion and contraction is to double table size when an item is inserted into a full table and halve the size when a deletion would cause the table to become less than half full
- This strategy guarantees that load factor of table never drops below 1/2

D'Oh

- Unfortunately this strategy can cause amortized cost of an operation to be large
- Assume we perform n operations where n is a power of 2
- The first $n/2$ operations are insertions
- At the end of this, $T.num = T.size = n/2$
- Now the remaining $n/2$ operations are as follows:

$$I, D, D, I, I, D, D, I, I, \ldots$$

where I represents an insertion and D represents a deletion

Analysis

- Note that the first insertion causes an expansion
- The two following deletions cause a contraction
- The next two insertions cause an expansion again, etc., etc.
- The cost of each expansion and deletion is $\Theta(n)$ and there are $\Theta(n)$ of them
- Thus the total cost of n operations is $\Theta(n^2)$ and so the amortized cost per operation is $\Theta(n)$

The Solution

- The Problem: After an expansion, we don't perform enough deletions to pay for the contraction (and vice versa)
- The Solution: We allow the load factor to drop below 1/2
- In particular, halve the table size when a deletion causes the table to be less than 1/4 full
- We can now create a potential function to show that Insertion and Deletion are fast in an amortized sense

Recall: Load Factor

- For a nonempty table T, we define the "load factor" of T, $\alpha(T)$, to be the number of items stored in the table divided by the size (number of slots) of the table
- We assign an empty table (one with no items) size 0 and load factor of 1
- Note that the load factor of any table is always between 0 and 1
- Further if we can say that the load factor of a table is always at least some constant c, then the unused space in the table is never more than $1 - c$

The Potential

$$\Phi(t) = \left\{ \begin{array}{ll} 2 * T.num - T.size & \text{if } \alpha(T) \geq 1/2 \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2 \end{array} \right\}$$

- Note that this potential is legal since $\Phi(0) = 0$ and (you can prove that) $\Phi(i) \geq 0$ for all i

Intuition

- Note that when $\alpha = 1/2$, the potential is 0
- When the load factor is 1 ($T.size = T.num$), $\Phi(T) = T.num$, so the potential can pay for an expansion
- When the load factor is 1/4, $T.size = 4 * T.num$, which means $\Phi(T) = T.num$, so the potential can pay for a contraction if an item is deleted

Analysis

- Let's now roll up our sleeves and show that the amortized costs of insertions and deletions are small
- We'll do this by case analysis
- Let num_i be the number of items in the table after the i-th operation, $size_i$ be the size of the table after the i-th operation, and α_i denote the load factor after the i-th operation

Table Insert

- If $\alpha_{i-1} \geq 1/2$, analysis is identical to the analysis done in the In-Class Exercise - amortized cost per operation is 3
- If $\alpha_{i-1} < 1/2$, the table will not expand as a result of the operation
- There are two subcases when $\alpha_{i-1} < 1/2$: 1) $\alpha_i < 1/2$ 2) $\alpha_i \geq 1/2$

Case: $\alpha_i < 1/2$

- In this case, we have

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0 \end{aligned}$$

Case: $\alpha_i \geq 1/2$

$$
\begin{aligned}
a_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 * num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2 * (num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 * num_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3 * \alpha_{i-1} * size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&< \frac{3}{2} * size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3
\end{aligned}
$$

Take Away

- So we've just show that in all cases, the amortized cost of an insertion is 3
- We did this by case analysis
- What remains to be shown is that the amortized cost of deletion is small
- We'll also do this by case analysis

Deletions

- For deletions, $num_i = num_{i-1} - 1$
- We will look at two main cases: 1) $\alpha_{i-1} < 1/2$ and 2) $\alpha_{i-1} \geq 1/2$
- For the case where $\alpha_{i-1} < 1/2$, there are two subcases: 1a) the i-th operation does not cause a contraction and 1b) the i-th operation does cause a contraction

Case: no contraction

- If $\alpha_{i-1} < 1/2$ and the i-th operation does not cause a contraction, we know $size_i = size_{i-1}$ and we have:

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

Case: Contraction

- In this case, $\alpha_{i-1} < 1/2$ and the i-th operation causes a contraction.
- We know that: $c_i = num_i + 1$ and

$$size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$$

- Thus we have:

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

Case: Deletion and $\alpha_{i-1} \geq 1/2$

- In this case, $\alpha_{i-1} \geq 1/2$
- Proving that the amortized cost is constant for this case is left as an exercise to the diligent student
- Hint1: Q: In this case is it possible for the i-th operation to be a contraction? If so, when can this occur? Hint2: Try a case analysis on α_i.

Take Away

- Since we've shown that the amortized cost of every operation is at most a constant, we've shown that any sequence of n operations on a Dynamic table take $O(n)$ time
- Note that in our scheme, the load factor never drops below $1/4$
- This means that we also never have more than $3/4$ of the table that is just empty space

Union-Find

- Union-Find is our last application of amortized analysis
- A union-find data structure maintains a collection $\{S_1, S_2, \ldots S_k\}$ of disjoint dynamic sets
- 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. The representative of the new set is any member of the set.
- 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 Union operation reduces the number of sets by 1
- So after $n - 1$ Union operations, only one set remains
- Thus the number of Union operations 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

Application

- Consider a simplified version of Facebook
- Every person is an object and every set represents a social clique
- Whenever a person in the set S_1 forges a link to a person in the set S_2, then we want to create a new larger social clique $S_1 \cup S_2$ (and delete S_1 and S_2)
- We might also want to find a representative of each set, to make it easy to search through the set
- For obvious reasons, we want these operation of Union, Make-Set and Find-Set to be as fast as possible

Example

- Make-Set(“Bob”), Make-Set(“Sue”), Make-Set(“Jane”), Make-Set(“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

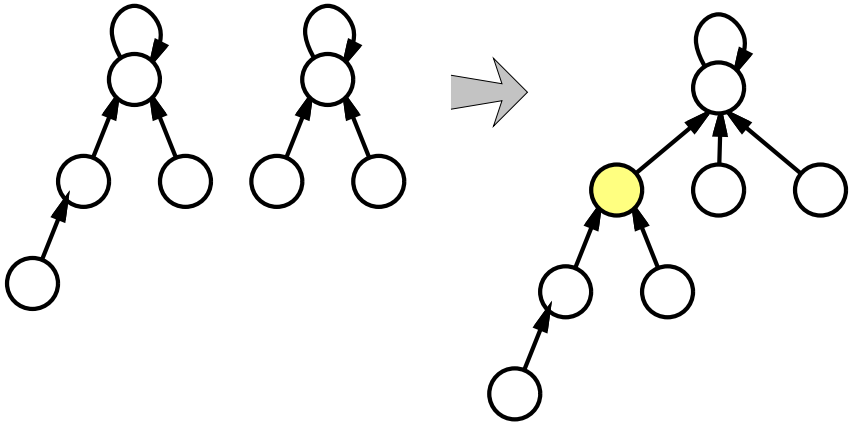
- Make-Set is trivial (we just create one root node)
- Find-Set traverses the parent pointers up to the leader (the root node).
- Union just redirects the parent pointer of one leader to the other.

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

Algorithms

```
Make-Set(x){
  parent(x) = x;
}
Find-Set(x){
  while(x!=parent(x))
    x = parent(x);
  return x;
}
Union(x,y){
  xParent = Find-Set(x);
  yParent = Find-Set(y);
  parent(yParent) = 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){
  parent(x) = x;
  size(x) = 1;
}
Union(x,y){
  xRep = Find-Set(x);
  yRep = Find-Set(y);
  if (size(xRep)) > size(yRep)){
    parent(yRep) = xRep;
    size(xRep) = size(xRep) + size(yRep);
  }else{
    parent(xRep) = yRep;
    size(yRep) = size(yRep) + 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 height(T) = Max(height(T1),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 it's 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 sets leaders
- The threads of two sets can be merged by the Union algorithm in constant time

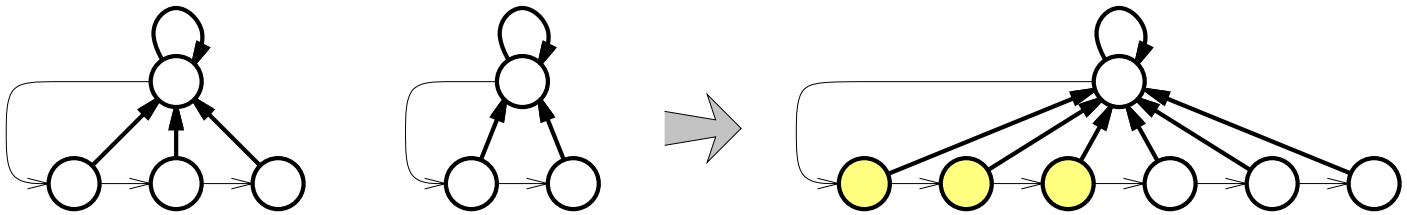
The Code

```
Make-Set(x){
  leader(x) = x;
  next(x) = NULL;
}
Find-Set(x){
  return leader(x);
}
```

The Code

```
Union(x,y){
  xRep = Find-Set(x);
  yRep = Find-Set(y);
  leader(y) = xRep;
  while(next(y)!=NULL){
    y = next(y);
    leader(y) = xRep;
  }
  next(y) = next(xRep);
  next(xRep) = 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){
  leader(x) = x;
  next(x) = NULL;
  size(x) = 1;
}
```

The Code

```
Weighted-Union(x,y){
  xRep = Find-Set(x);
  yRep = Find-Set(y)
  if(size(xRep)>size(yRep){
    Union(xRep,yRep);
    size(xRep) = size(xRep) + size(yRep);
  }else{
    Union(yRep,xRep);
    size(yRep) = size(xRep) + size(yRep);
  }
}
```

Analysis

- The Weighted-Union algorithm still takes $\Theta(n)$ time to merge two n element sets
- However in an amortized sense, it is more efficient:
- A sequence of m Make-Weighted-Set operations and n Weighted-Union operations takes $O(m + n \log n)$ time in the worst case.

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

- Let n be the number of calls to Make-Weighted-Set and m be the number of calls to Weighted-Union
- We've shown that each of the objects that are not in singleton sets had at most $O(\log n)$ leader changes
- Thus, the total amount of work done in updating the leader pointers is $O(n \log n)$

Proof

- We've just shown that for n calls to Make-Weighted-Set and m calls to Weighted-Union, that total cost for updating leader pointers is $O(n \log n)$
- We know that other than the work needed to update these leader pointers, each call to one of our functions does only constant work
- Thus total amount of work is $O(n \log n + m)$
- Thus each Weighted-Union call has amortized cost of $O(\log n)$

Side Note: We've just used the aggregate method of amortized analysis

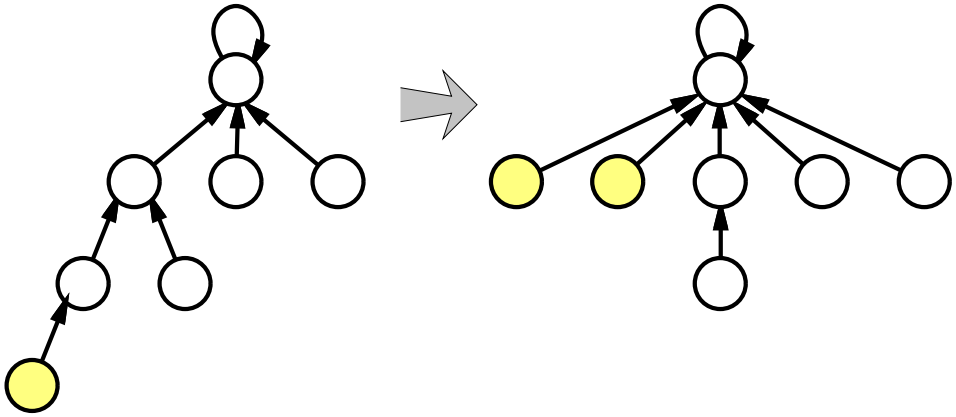
Analysis

- 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, *Union* takes logarithmic amortized time and everything else is constant
- 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 (from Simple-Union)
- 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

Example



Path compression during Find(c). Shaded nodes have a new parent.

PC-Find-Set

```
PC-Find-Set(x){
  if(x!=Parent(x)){
    Parent(x) = PC-Find-Set(Parent(x));
  }
  return 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){
  parent(x) = x;
  rank(x) = 0;
}
PC-Union(x,y){
  xRep = PC-Find-Set(x);
  yRep = PC-Find-Set(y);
  if(rank(xRep) > rank(yRep))
    parent(yRep) = xRep;
  else{
    parent(xRep) = yRep;
    if(rank(xRep)==rank(yRep))
      rank(yRep)++;
    }
}
```

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, size(X) $\geq 2^{rank(leader(X))}$ (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

Blocks

- We will also partition the objects into several numbered blocks
- x is assigned to block number $\log^*(rank(x))$
- Recall: Intuitively, $\log^* n$ is the number of times you need to hit the log button on your calculator, after entering n, before you get something less than or equal to 1
- In other words x is in block b if

$$2 \uparrow\uparrow (b-1) < rank(x) \leq 2 \uparrow\uparrow b,$$

where $\uparrow\uparrow$ is defined as in the next slide

$\log^* x$ 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

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

Number of Blocks

- Every object has a rank between 0 and $\lfloor \log n \rfloor$
- So the blocks numbers range from 0 to $\log^* \lfloor \log n \rfloor = \log^*(n) - 1$
- Hence there are $\log^* n$ blocks

Number of Objects in Block b

- Since there are at most $n/2^r$ objects with any rank r, the total number of objects in block b is at most

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

Theorem

- **Theorem:** If we use both PC-Find-Set and PC-Union (i.e. Path Compression and Weighted Union), 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 any sequence of m PC-Find-Set operations require $O(m \log^* n)$ time in the worst case
- We will use a kind of accounting method to show this

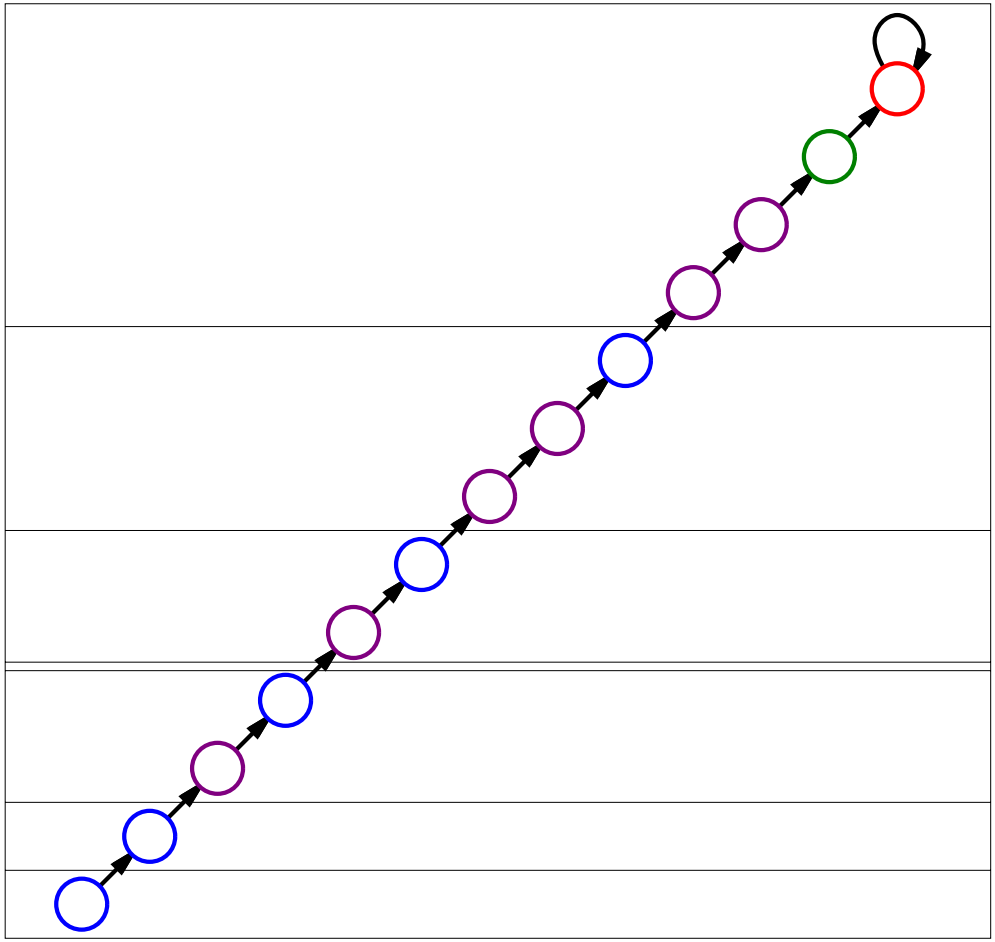
Proof

- The cost of PC-Find-Set(x_0) is proportional to the number of nodes on the path from x_0 up to its leader
- Each object $x_0, x_1, x_2, \ldots, x_l$ 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.

Leader, Child and Block 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 the $\log^* n$ blocks
- Thus when the sequence of m operations ends, these accounts share a total of at most $2m + m \log^* n$ 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 $2 \uparrow\uparrow b$ times total*

Path Account

- Since block b contains less than $n/(2 \uparrow\uparrow b)$ objects, and each of these objects contributes less than $2 \uparrow\uparrow b$ dollars, the total number of dollars contributed by objects in block b is less than n dollars to the path account
- There are $\log^* n$ blocks so the path account receives less than $n \log^* n$ dollars total
- Thus the total amount of money in all four accounts is less than $2m + m \lg^* n + n \lg^* n = O(m \lg^* n)$, and this bounds the total running time of the m operations.

Take Away

- We can now say that each call to PC-Find-Set and PC-Union has 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 has amortized cost of $O(A^{-1}(n))$ where $A^{-1}(n)$ (inverse Ackerman function) is an even slower growing function than $\log^* n$