CSE 548: Algorithms

Amortized Analysis

R. Sekar

Amortized Analysis

Amortization

The spreading out of capital expenses for intangible assets over a specific period of time (usually over the asset's useful life) for accounting and tax purposes.

- A clever trick used by accountants to average large one-time costs over time.
- In algorithms, we use amortization to spread out the cost of expensive operations.
    - Example: Re-sizing a hash table.

Topics

Summation or Aggregate Method

- Some operations have high worst-case cost, but we can show that the worst case does not occur every time.
- In this case, we can average the costs to obtain a better bound

Summation or Aggregate Method

- Some operations have high worst-case cost, but we can show that the worst case does not occur every time.
- In this case, we can average the costs to obtain a better bound

Summation

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

Summation or Aggregate Method

- Some operations have high worst-case cost, but we can show that the worst case does not occur every time.
- In this case, we can average the costs to obtain a better bound

Summation

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

Note: We are not making an "average case" argument about inputs. *We are still talking about worst-case performance.*

Summation Example: Binary Counter

- What is the worst-case runtime of *incr*?

Incr(B[0..])

$i = 0$

while $B[i] = 1$

  $B[i] = 0$

  $i{+}{+}$

$B[i] = 1$

Summation Example: Binary Counter

Incr($B[0..]$)

$i = 0$
while $B[i] = 1$
  $B[i] = 0$
  $i++$
$B[i] = 1$

- What is the worst-case runtime of *incr*?
  - Simple answer: $O(\log n)$, where $n = \#$ of *incr*'s performed

Summation Example: Binary Counter

Incr($B[0..]$)

$i = 0$

while $B[i] = 1$

  $B[i] = 0$

  $i{+}{+}$

$B[i] = 1$

- What is the worst-case runtime of *incr*?
  - Simple answer: $O(\log n)$, where $n = \#$ of *incr*'s performed
- What is the *amortized* runtime for n *incr*'s?

Summation Example: Binary Counter

Incr($B[0..]$)

$i = 0$
while $B[i] = 1$
  $B[i] = 0$
  $i++$
$B[i] = 1$

- What is the worst-case runtime of *incr*?
  - Simple answer: $O(\log n)$, where $n = \#$ of *incr*'s performed
- What is the *amortized* runtime for n *incr*'s?
  - Easy to see that an *incr* will touch $B[i]$ once every 2^i operations.

Summation Example: Binary Counter

Incr($B[0..]$)

$i = 0$

while $B[i] = 1$

  $B[i] = 0$

  $i++$

$B[i] = 1$

- What is the worst-case runtime of *incr*?
  - Simple answer: $O(\log n)$, where $n = \#$ of *incr*'s performed
- What is the *amortized* runtime for n *incr*'s?
  - Easy to see that an *incr* will touch $B[i]$ once every 2^i operations.
  - Number of operations is thus

$$n \sum_{i=0}^{\log n} \frac{1}{2^i} = 2n$$

Summation Example: Binary Counter

Incr(B[0..])

$i = 0$
while $B[i] = 1$
  $B[i] = 0$
  $i{+}{+}$
$B[i] = 1$

- What is the worst-case runtime of *incr*?
  - Simple answer: $O(\log n)$, where $n = \#$ of *incr*'s performed
- What is the *amortized* runtime for n *incr*'s?
  - Easy to see that an *incr* will touch $B[i]$ once every 2^i operations.
  - Number of operations is thus

$$n \sum_{i=0}^{\log n} \frac{1}{2^i} = 2n$$

- Thus, amortized cost per *incr* is $O(1)$

Charging Method

Certain operations charge more than their cost so as to pay for other operations. This allows total cost to be calculated while ignoring the second category of operations.

Charging Method

Certain operations charge more than their cost so as to pay for other operations. This allows total cost to be calculated while ignoring the second category of operations.

- In the counter example, we charge 2 units for each operation to change a 0-bit to 1-bit.

Charging Method

Certain operations charge more than their cost so as to pay for other operations. This allows total cost to be calculated while ignoring the second category of operations.

- In the counter example, we charge 2 units for each operation to change a 0-bit to 1-bit.
- Pays for the cost of later flipping the 1-bit to 0-bit.

Charging Method

Certain operations charge more than their cost so as to pay for other operations. This allows total cost to be calculated while ignoring the second category of operations.

- In the counter example, we charge 2 units for each operation to change a 0-bit to 1-bit.
- Pays for the cost of later flipping the 1-bit to 0-bit.
  - *Important: ensure you have charged enough.*
    - We have satisfied this: a bit can be flipped from 1 to 0 only once after it is flipped from 0 to 1.

Charging Method

Certain operations charge more than their cost so as to pay for other operations. This allows total cost to be calculated while ignoring the second category of operations.

- In the counter example, we charge 2 units for each operation to change a 0-bit to 1-bit.
- Pays for the cost of later flipping the 1-bit to 0-bit.
  - *Important: ensure you have charged enough.*
    - We have satisfied this: a bit can be flipped from 1 to 0 only once after it is flipped from 0 to 1.
- Now we ignore costs of 1 to 0 flips in the algorithm
  - There is only one 0-to-1 bit flipping per call of *incr*!
  - So, *incr* only costs 2 units for each invocation!

Stack Example

- Consider a stack with two operations:

  push(x): Push a value x on the stack

  pop(k): Pop off the top k elements

Stack Example

- Consider a stack with two operations:

  push(x): Push a value x on the stack

  pop(k): Pop off the top k elements

- What is the cost of a mix of n push and pop operations?

- *Key problem:* Worst-case cost of a *pop* is $O(n)$!

Stack Example

- Consider a stack with two operations:

  push(x): Push a value x on the stack

  pop(k): Pop off the top k elements

- What is the cost of a mix of n push and pop operations?
- *Key problem:* Worst-case cost of a *pop* is $O(n)$!
- *Solution:*
  - Charge 2 units for each push: covers the cost of pushing, and also the cost of a subsequent pop

Stack Example

- Consider a stack with two operations:

  push(x): Push a value x on the stack

  pop(k): Pop off the top k elements

- What is the cost of a mix of n push and pop operations?
- *Key problem:* Worst-case cost of a *pop* is $O(n)$!
- *Solution:*
  - Charge 2 units for each push: covers the cost of pushing, and also the cost of a subsequent pop
  - A pushed item can be popped only once, so we have charged enough

Stack Example

- Consider a stack with two operations:

  push(x): Push a value x on the stack

  pop(k): Pop off the top k elements

- What is the cost of a mix of n push and pop operations?
- *Key problem:* Worst-case cost of a *pop* is $O(n)$!
- *Solution:*
  - Charge 2 units for each push: covers the cost of pushing, and also the cost of a subsequent pop
  - A pushed item can be popped only once, so we have charged enough
  - Now, ignore pop's altogther, and trivially arrive at $O(1)$ amortized cost for the sequence of push/pop operations!

Potential Method

Define a potential for a data structure that is initially zero, and is always non-negative. The amortized cost of an operation is the cost of the operation minus the change in potential.

Potential Method

Define a potential for a data structure that is initially zero, and is always non-negative. The amortized cost of an operation is the cost of the operation minus the change in potential.

- Analogy with “potential” energy. “Potential” is prepaid cost that can be used subsequently
  - as the data structure changes and “releases” stored energy

Potential Method

Define a potential for a data structure that is initially zero, and is always non-negative. The amortized cost of an operation is the cost of the operation minus the change in potential.

- Analogy with "potential" energy. "Potential" is prepaid cost that can be used subsequently
  - as the data structure changes and "releases" stored energy
- A more sophisticated technique that allows "charges" or "taxes" to be stored within nodes in a data structure and used subsequently at a later time.

Potential Method: Illustration

Stack:

- Each push costs 2 units because a push increases potential energy by 1.
- Pops can use the energy released by reduction in stack size!

Potential Method: Illustration

Stack:

- Each push costs 2 units because a push increases potential energy by 1.
- Pops can use the energy released by reduction in stack size!

Counter:

- Define potential as the number one 1-bits
- Changing a 0 to 1 costs 2 units, one for the operation and one to pay for increase in potential
- Changes of 1 to 0 can now be paid by released potential.

Hash Tables

- To provide expected constant time access, collisions need to be limited

Hash Tables

- To provide expected constant time access, collisions need to be limited
- This requires hash table resizing when they become too full
  - But this requires all entries to be deleted from current table and inserted into a table that is larger — *a very expensive operation.*

Hash Tables

- To provide expected constant time access, collisions need to be limited
- This requires hash table resizing when they become too full
  - But this requires all entries to be deleted from current table and inserted into a table that is larger — *a very expensive operation.*
- *Options:*

1. Try to guess the table size right; if you guessed wrong, put up with the pain of low performance.

Hash Tables

- To provide expected constant time access, collisions need to be limited
- This requires hash table resizing when they become too full
  - But this requires all entries to be deleted from current table and inserted into a table that is larger — *a very expensive operation.*
- *Options:*
  1. Try to guess the table size right; if you guessed wrong, put up with the pain of low performance.
  2. Quit complaining, bite the bullet, and rehash as needed;

Hash Tables

- To provide expected constant time access, collisions need to be limited
- This requires hash table resizing when they become too full
  - But this requires all entries to be deleted from current table and inserted into a table that is larger — *a very expensive operation.*
- *Options:*

1. Try to guess the table size right; if you guessed wrong, put up with the pain of low performance.
2. Quit complaining, bite the bullet, and rehash as needed;
3. *Amortize*: Rehash as needed, *and* prove that it does not cost much!

Amortized Rehashing

Amortize the cost of rehashing over other hash table operations

Amortized Rehashing

Amortize the cost of rehashing over other hash table operations

Approach 1: Rehash after a large number (say, 1K) operations.

Total cost of 1K ops = 1K for the ops + 1K for rehash = 2K

Amortized Rehashing

Amortize the cost of rehashing over other hash table operations

Approach 1: Rehash after a large number (say, 1K) operations.

Total cost of 1K ops = 1K for the ops + 1K for rehash = 2K

Note: We may have at most 1K elements in the table after 1K operations, so we may need to rehash at most 1K times.

So, amortized cost is just 2!

Amortized Rehashing

Amortize the cost of rehashing over other hash table operations

Approach 1: Rehash after a large number (say, 1K) operations.

Total cost of 1K ops = 1K for the ops + 1K for rehash = 2K

Note: We may have at most 1K elements in the table after 1K operations, so we may need to rehash at most 1K times.

So, amortized cost is just 2!

Are we done?

Amortized Rehash (2)

Are we done?

Consider total cost after 2K, 3K, and 4K operations:

$$T(2K) \;=\; 2K + 1K \text{ (first rehash) } + 2K \text{ (second rehash) } = 5K$$

Amortized Rehash (2)

Are we done?

Consider total cost after 2K, 3K, and 4K operations:

$$T(2K) \;=\; 2K + 1K \text{ (first rehash) } + 2K \text{ (second rehash) } = 5K$$

$$T(3K) \;=\; 3K + 1K \text{ (}1^{st}\text{ rehash) } + 2K \text{ (}2^{nd}\text{ rehash) } + 3K \text{ (}3^{rd}\text{...) } = 9K$$

Amortized Rehash (2)

Are we done?

Consider total cost after 2K, 3K, and 4K operations:

$$
\begin{aligned}
T(2K) &= 2K + 1K \text{ (first rehash) } + 2K \text{ (second rehash) } = 5K \\
T(3K) &= 3K + 1K \text{ (}1^{st}\text{ rehash) } + 2K \text{ (}2^{nd}\text{ rehash) } + 3K \text{ (}3^{rd}\text{...) } = 9K \\
T(4K) &= 4K + 1K + 2K + 3K + 4K = 14K
\end{aligned}
$$

Amortized Rehash (2)

Are we done?

Consider total cost after 2K, 3K, and 4K operations:

$$\begin{aligned} T(2K) &= 2K + 1K \text{ (first rehash) } + 2K \text{ (second rehash) } = 5K \\ T(3K) &= 3K + 1K \text{ (}1^{st}\text{ rehash) } + 2K \text{ (}2^{nd}\text{ rehash) } + 3K \text{ (}3^{rd}\text{...) } = 9K \\ T(4K) &= 4K + 1K + 2K + 3K + 4K = 14K \end{aligned}$$

Hmmm. This is growing like n^2, so amortized cost will be $O(n)$

Need to try a different approach.

Amortized Rehash (3)

Approach 2: Double the hash table whenever it gets full

Say, you start with an empty table of size N. For simplicity, assume only insert operations.

Amortized Rehash (3)

Approach 2: Double the hash table whenever it gets full

Say, you start with an empty table of size N. For simplicity, assume only insert operations.

You invoke N insert operations, then rehash to a $2N$ table.

$$T(N) = N + N \text{ (rehashing } N \text{ entries)} \ = 2N$$

Amortized Rehash (3)

Approach 2: Double the hash table whenever it gets full

Say, you start with an empty table of size N. For simplicity, assume only insert operations.

You invoke N insert operations, then rehash to a $2N$ table.

$$T(N) = N + N \text{ (rehashing } N \text{ entries)} \ = 2N$$

Now, you can insert N more before needing rehash.

$$T(2N) = T(N) + N + 2N \text{ (rehashing } 2N \text{ entries)} \ = 5N$$

Amortized Rehash (3)

Approach 2: Double the hash table whenever it gets full

Say, you start with an empty table of size N. For simplicity, assume only insert operations.

You invoke N insert operations, then rehash to a $2N$ table.

$$T(N) = N + N\,(\text{rehashing } N \text{ entries}) \;= 2N$$

Now, you can insert N more before needing rehash.

$$T(2N) = T(N) + N + 2N\,(\text{rehashing } 2N \text{ entries}) \;= 5N$$

Now, you can insert $2N$ more before needing rehash:

$$T(4N) = T(2N) + 2N + 4N\,(\text{rehashing } 4N \text{ entries}) \;= 11N$$

Amortized Rehash (3)

Approach 2: Double the hash table whenever it gets full

Say, you start with an empty table of size N. For simplicity, assume only insert operations.

You invoke N insert operations, then rehash to a $2N$ table.

$$T(N) = N + N \text{ (rehashing } N \text{ entries)} \;= 2N$$

Now, you can insert N more before needing rehash.

$$T(2N) = T(N) + N + 2N \text{ (rehashing } 2N \text{ entries)} \;= 5N$$

Now, you can insert $2N$ more before needing rehash:

$$T(4N) = T(2N) + 2N + 4N \text{ (rehashing } 4N \text{ entries)} \;= 11N$$

The general recurrence is $T(n) = T(n/2) + 1.5n$, which is linear.

So, amortized cost is constant!

Amortized Rehash (4)

Alternatively, we can think in terms of *charging*.

Each insert operation can be charged 3 units of cost:

- One for the insert operation
- One for rehashing of this element at the end of this run of inserts
- One for rehashing an element that was already in the hash table when this run began

Amortized Rehash (4)

Alternatively, we can think in terms of *charging*.

Each insert operation can be charged 3 units of cost:

- One for the insert operation
- One for rehashing of this element at the end of this run of inserts
- One for rehashing an element that was already in the hash table when this run began

A run contains as many elements as the hash table at the beginning of run — so we have accounted for all costs.

Amortized Rehash (4)

Alternatively, we can think in terms of *charging*.

Each insert operation can be charged 3 units of cost:

- One for the insert operation
- One for rehashing of this element at the end of this run of inserts
- One for rehashing an element that was already in the hash table when this run began

A run contains as many elements as the hash table at the beginning of run — so we have accounted for all costs.

Thus, rehashing

- increases the costs of insertions by a factor of 3.
- lookup costs are unchanged.

Amortized Rehash (5)

- Alternatively, we can think in terms of *potential*.
- Hash table as a spring: as more elements are inserted, the spring has to be compressed to make room.

Amortized Rehash (5)

- Alternatively, we can think in terms of *potential.*
- Hash table as a spring: as more elements are inserted, the spring has to be compressed to make room.
- Let $|H|$ denote the capacity and α the occupancy of H
- Define potential as 0 when $\alpha \leq 0.5$ and $2(\alpha - 0.5)|H|$ otherwise.

Amortized Rehash (5)

- Alternatively, we can think in terms of *potential*.
- Hash table as a spring: as more elements are inserted, the spring has to be compressed to make room.
- Let $|H|$ denote the capacity and α the occupancy of H
- Define potential as 0 when $\alpha \leq 0.5$ and $2(\alpha - 0.5)|H|$ otherwise.
- Immediately after resize, let the hash table capacity be k. Note $\alpha \leq 0.5$ so potential is 0.

Amortized Rehash (5)

- Alternatively, we can think in terms of *potential.*
- Hash table as a spring: as more elements are inserted, the spring has to be compressed to make room.
- Let $|H|$ denote the capacity and α the occupancy of H
- Define potential as 0 when $\alpha \leq 0.5$ and $2(\alpha - 0.5)|H|$ otherwise.
- Immediately after resize, let the hash table capacity be k. Note $\alpha \leq 0.5$ so potential is 0.
- Each insert (after α reaches 0.5) costs 3 units: one for the operation, and 2 for the increase in potential.

Amortized Rehash (5)

- Alternatively, we can think in terms of *potential*.
- Hash table as a spring: as more elements are inserted, the spring has to be compressed to make room.
- Let $|H|$ denote the capacity and α the occupancy of H
- Define potential as 0 when $\alpha \leq 0.5$ and $2(\alpha - 0.5)|H|$ otherwise.
- Immediately after resize, let the hash table capacity be k. Note $\alpha \leq 0.5$ so potential is 0.
- Each insert (after α reaches 0.5) costs 3 units: one for the operation, and 2 for the increase in potential.
- When α reaches 1, the potential is $2k$. After resizing to $2k$, potential falls to 0, and the released $2k$ cost pays for rehashing $2k$ elements.

Amortized Rehash (6)

- What if we increase the size by a factor less than 2?
  - Is there a threshold $t > 1$ such that expansion by a factor less than t won't yield amortized constant time?
- What happens if we want to support both deletes and inserts, and want to make sure that the table never uses more than k times the actual number of elements?
  - Is there a minimum value of k for which this can be achieved?
  - Do you need a different threshold for expansion and contraction? Are there any constraints on the relationship between these two thresholds to ensure amortized constant time?

Amortized performance of Vectors vs Lists

Linked lists: Data structures of choice if you don't know the # of elements in advance.

Space inefficient: 2x or more memory for very small objects.

Poor cache performance: Pointer chasing is cache unfriendly.

Sequential access: No fast access to kth element.

Vectors: Dynamically-sized arrays have none of these problems. But resizing is expensive.

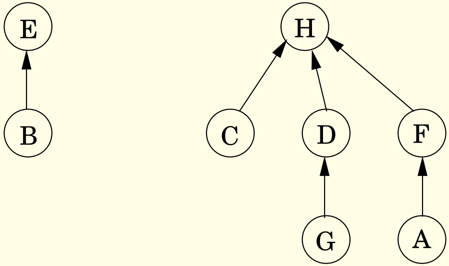
- Is it possible to achieve good amortized performance?
- When should the vector be expanded/contracted?
- What operations can we support in constant amortized time? Inserts? insert at end? concatenation?

Strings: We can raise similar questions as Vectors.

Disjoint Sets

- Represent disjoint sets as “inverted trees”
- Each element has a parent pointer π
- To compute the union of set A with B, simply make B’s root the parent of A’s root.

A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.



Disjoint Sets (2)

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x):  x = π(x)
return x
```

Disjoint Sets (2)

$\underline{\texttt{procedure makeset}}(x)$

$\pi(x) = x$

$\texttt{rank}(x) = 0$

$\underline{\texttt{function find}}(x)$

$\texttt{while } x \neq \pi(x): \quad x = \pi(x)$

$\texttt{return } x$

$\texttt{procedure union}(x, y)$

$r_x = \texttt{find}(x)$

$r_y = \texttt{find}(y)$

$\pi(r_y) = r_x$

Disjoint Sets (2)

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x) :  x = π(x)
return x

procedure union(x, y)
r_x = find(x)
r_y = find(y)
π(r_y) = r_x
```

Complexity

- makeset takes $O(1)$ time

Disjoint Sets (2)

<u>procedure makeset</u>(x)

$\pi(x) = x$

$\text{rank}(x) = 0$

<u>function find</u>(x)

while $x \neq \pi(x)$: $x = \pi(x)$

return x

procedure union(x, y)

$r_x =$ find(x)

$r_y =$ find(y)

$\pi(r_y) = r_x$

Complexity

- makeset takes $O(1)$ time
- find takes time equal to depth of set: $O(n)$ in the worst case.

Disjoint Sets (2)

<u>procedure makeset</u>(x)

$\pi(x) = x$

$\mathtt{rank}(x) = 0$

<u>function find</u>(x)

while $x \neq \pi(x)$: $x = \pi(x)$

return x

procedure union(x, y)

$r_x =$ find(x)

$r_y =$ find(y)

$\pi(r_y) = r_x$

Complexity

- makeset takes $O(1)$ time
- find takes time equal to depth of set: $O(n)$ in the worst case.
- union takes $O(1)$ time on a root element; in the worst case, its complexity matches find.

Disjoint Sets (2)

procedure makeset(x)
$\pi(x) = x$
rank$(x) = 0$

function find(x)
while $x \neq \pi(x)$: $x = \pi(x)$
return x

procedure union(x, y)
$r_x =$ find(x)
$r_y =$ find(y)
$\pi(r_y) = r_x$

Complexity

- makeset takes $O(1)$ time
- find takes time equal to depth of set: $O(n)$ in the worst case.
- union takes $O(1)$ time on a root element; in the worst case, its complexity matches find.

Amortized complexity

- Can you construct a worst-case example, where N operations take $O(N^2)$ time?

Disjoint Sets (2)

<u>procedure makeset</u>(x)
$\pi(x) = x$
$\mathtt{rank}(x) = 0$

<u>function find</u>(x)
while $x \neq \pi(x)$: $x = \pi(x)$
return x

procedure union(x, y)
$r_x =$ find(x)
$r_y =$ find(y)
$\pi(r_y) = r_x$

Complexity

- makeset takes $O(1)$ time
- find takes time equal to depth of set: $O(n)$ in the worst case.
- union takes $O(1)$ time on a root element; in the worst case, its complexity matches find.

Amortized complexity

- Can you construct a worst-case example, where N operations take $O(N^2)$ time?
- Can we improve this?

Disjoint Sets with Union by Depth

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x) :  x = π(x)
return x
```

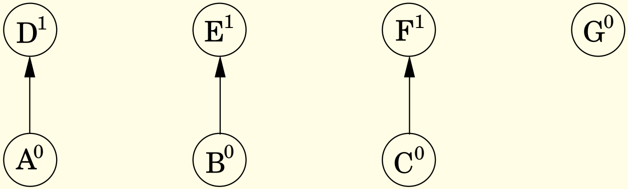
```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y :   return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y) :
        rank(r_y) = rank(r_y) + 1
```

Disjoint Sets with Union by Depth (2)

Figure 5.6 A sequence of disjoint-set operations. Superscripts denote rank.

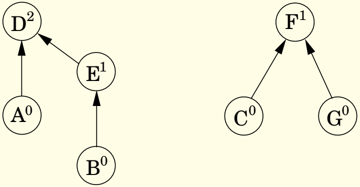
After makeset(A), makeset(B), . . . , makeset(G):

After union(A, D), union(B, E), union(C, F):

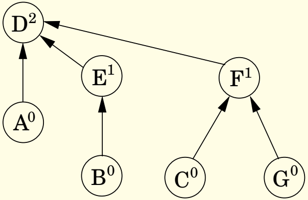


Disjoint Sets with Union by Depth (3)

After $\texttt{union}(C, G), \texttt{union}(E, A)$:



After $\texttt{union}(B, G)$:



Complexity of disjoint sets w/ union by depth

- Asymptotic complexity of `makeset` unchanged.
  - `union` has become a bit more expensive, but only modestly.

Complexity of disjoint sets w/ union by depth

- Asymptotic complexity of `makeset` unchanged.
  - `union` has become a bit more expensive, but only modestly.
- What about `find`?
  - A sequence of N operations can create at most N elements
  - So, maximum set size is $O(N)$

Complexity of disjoint sets w/ union by depth

- Asymptotic complexity of `makeset` unchanged.
  - `union` has become a bit more expensive, but only modestly.
- What about `find`?
  - A sequence of N operations can create at most N elements
  - So, maximum set size is $O(N)$
  - With union by rank, each increase in rank can occur only after a doubling of elements in the set

Complexity of disjoint sets w/ union by depth

- Asymptotic complexity of `makeset` unchanged.
  - `union` has become a bit more expensive, but only modestly.
- What about `find`?
  - A sequence of N operations can create at most N elements
  - So, maximum set size is $O(N)$
  - With union by rank, each increase in rank can occur only after a doubling of elements in the set

Observation

The number of nodes of rank k never exceeds $N/2^k$

Complexity of disjoint sets w/ union by depth

- Asymptotic complexity of `makeset` unchanged.
  - `union` has become a bit more expensive, but only modestly.
- What about `find`?
  - A sequence of N operations can create at most N elements
  - So, maximum set size is $O(N)$
  - With union by rank, each increase in rank can occur only after a doubling of elements in the set

Observation

The number of nodes of rank k never exceeds $N/2^k$

- So, height of trees is bounded by $\log N$

Complexity of disjoint sets w/ union by depth (2)

- Height of trees is bounded by $\log N$

Complexity of disjoint sets w/ union by depth (2)

- Height of trees is bounded by $\log N$
- Thus we have a complexity of $O(\log N)$ for `find`
  - *Question:* Is this bound tight?

Complexity of disjoint sets w/ union by depth (2)

- Height of trees is bounded by $\log N$
- Thus we have a complexity of $O(\log N)$ for `find`
  - *Question:* Is this bound tight?

From here on, we limit union operations to only root nodes, so their cost is $O(1)$.

Complexity of disjoint sets w/ union by depth (2)

- Height of trees is bounded by $\log N$
- Thus we have a complexity of $O(\log N)$ for `find`
  - *Question:* Is this bound tight?

From here on, we limit union operations to only root nodes, so their cost is $O(1)$.

This requires find to be moved out of union into a separate operation, and hence the total number of operations increases, but only by a constant factor.

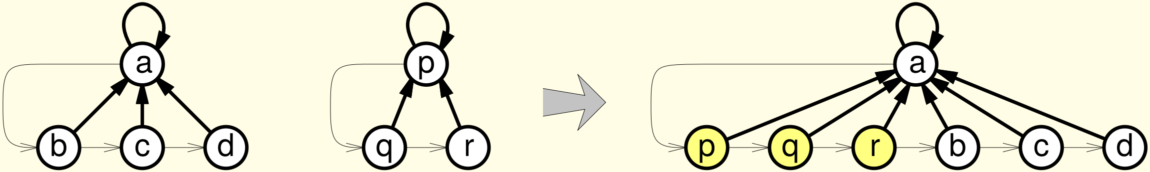
Improving `find` performance

Idea: Why not force depth to be 1? Then `find` will have $O(1)$ complexity!

Improving `find` performance

Idea: Why not force depth to be 1? Then `find` will have $O(1)$ complexity!

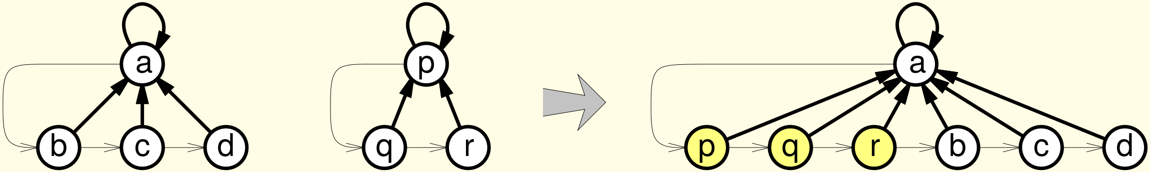
Approach: **Threaded Trees**



Improving `find` performance

Idea: Why not force depth to be 1? Then `find` will have $O(1)$ complexity!

Approach: **Threaded Trees**



Problem: Worst-case complexity of `union` becomes $O(n)$

Solution:

- Merge smaller set with larger set
- Amortize cost of `union` over other operations

Sets w/ threaded trees: Amortized analysis

- Other than cost of updating parent pointers, `union` costs $O(1)$

Sets w/ threaded trees: Amortized analysis

- Other than cost of updating parent pointers, `union` costs $O(1)$
- **Idea:** Charge the cost of updating a parent pointer to an element.

Sets w/ threaded trees: Amortized analysis

- Other than cost of updating parent pointers, `union` costs $O(1)$
- **Idea:** Charge the cost of updating a parent pointer to an element.
- **Key observation:** Each time an element's parent pointer changes, it is in a set that is twice as large as before

Sets w/ threaded trees: Amortized analysis

- Other than cost of updating parent pointers, `union` costs $O(1)$
- **Idea:** Charge the cost of updating a parent pointer to an element.
- **Key observation:** Each time an element's parent pointer changes, it is in a set that is twice as large as before
  - So, with n operations, you can at most $O(\log n)$ parent pointer updates per element

Sets w/ threaded trees: Amortized analysis

- Other than cost of updating parent pointers, `union` costs $O(1)$
- **Idea:** Charge the cost of updating a parent pointer to an element.
- **Key observation:** Each time an element's parent pointer changes, it is in a set that is twice as large as before
  - So, with n operations, you can at most $O(\log n)$ parent pointer updates per element
- Thus, amortized cost of n operations, consisting of some mix of `makeset`, `find` and `union` is at most $n \log n$

Further improvement

- Can we combine the best elements of the two approaches?

Further improvement

- Can we combine the best elements of the two approaches?
  - Threaded trees use *eager* `union` while the original approach used a *lazy* approach

Further improvement

- Can we combine the best elements of the two approaches?
  - Threaded trees use *eager* `union` while the original approach used a *lazy* approach
    - Eager approach is better for `find`, while being lazy is better for `union`.

Further improvement

- Can we combine the best elements of the two approaches?
  - Threaded trees use *eager* `union` while the original approach used a *lazy* approach
    - Eager approach is better for `find`, while being lazy is better for `union`.
    - So, why not use lazy approach for `union` and eager approach for `find`?

Further improvement

- Can we combine the best elements of the two approaches?
  - Threaded trees use *eager* `union` while the original approach used a *lazy* approach
    - Eager approach is better for `find`, while being lazy is better for `union`.
    - So, why not use lazy approach for `union` and eager approach for `find`?
- **Path compression:** Retains *lazy* `union`, but when a `find(x)` is called, *eagerly* promotes x to the level below the root

Further improvement

- Can we combine the best elements of the two approaches?
  - Threaded trees use *eager* `union` while the original approach used a *lazy* approach
    - Eager approach is better for `find`, while being lazy is better for `union`.
    - So, why not use lazy approach for `union` and eager approach for `find`?
- **Path compression:** Retains *lazy* `union`, but when a `find(x)` is called, *eagerly* promotes x to the level below the root
  - Actually, we promote $x, \pi(x), \pi(\pi(x)), \pi(\pi(\pi(x)))$ and so on.

Further improvement

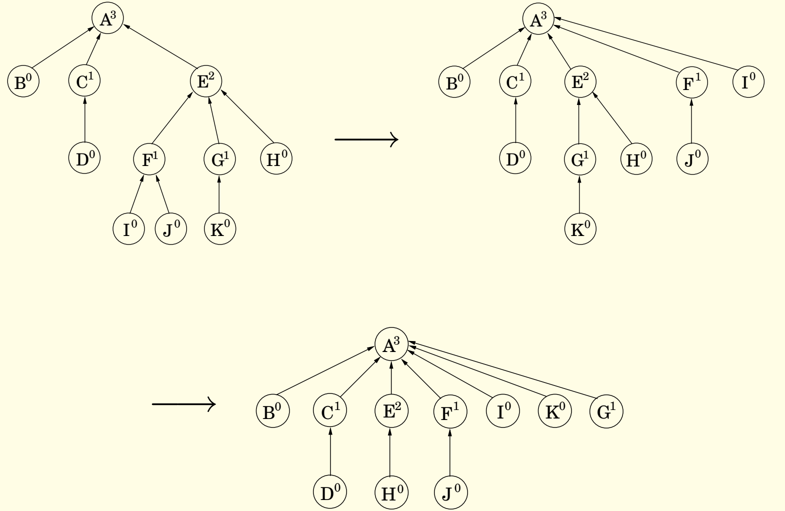
- Can we combine the best elements of the two approaches?
  - Threaded trees use *eager* `union` while the original approach used a *lazy* approach
    - Eager approach is better for `find`, while being lazy is better for `union`.
    - So, why not use lazy approach for `union` and eager approach for `find`?
- **Path compression:** Retains *lazy* `union`, but when a `find(x)` is called, *eagerly* promotes x to the level below the root
  - Actually, we promote $x, \pi(x), \pi(\pi(x)), \pi(\pi(\pi(x)))$ and so on.
  - As a result, subsequent calls to find x or its parents become cheap.

Further improvement

- Can we combine the best elements of the two approaches?
  - Threaded trees use *eager* `union` while the original approach used a *lazy* approach
    - Eager approach is better for `find`, while being lazy is better for `union`.
    - So, why not use lazy approach for `union` and eager approach for `find`?
- **Path compression:** Retains *lazy* `union`, but when a `find(x)` is called, *eagerly* promotes x to the level below the root
  - Actually, we promote $x, \pi(x), \pi(\pi(x)), \pi(\pi(\pi(x)))$ and so on.
  - As a result, subsequent calls to find x or its parents become cheap.
- From here on, we let *rank* be defined by the *union* algorithm
  - For root node, *rank* is same as depth
  - But once a node becomes a non-root, its *rank* stays fixed,
    - *even when path compression decreases its depth.*

Disjoint sets w/ Path compression: Illustration

`find`(I) followed by `find`(K)



Sets w/ Path compression: Amortized analysis

Amortized cost per operation of n set operations is $O(\log^* n)$ where

$$\log^* x = \text{smallest } k \text{ such that } \underbrace{log(log(\cdots log}_{k \text{ times}}(x)\cdots)) = 1$$

Sets w/ Path compression: Amortized analysis

Amortized cost per operation of n set operations is $O(\log^* n)$ where

$$\log^* x = \text{smallest } k \text{ such that } \underbrace{log(log(\cdots log}_{k \text{ times}}(x)\cdots)) = 1$$

Note: $\log^*(x) \leq 5$ for virtually any n of practical relevance. Specifically,

$$\log^*(2^{65536}) = \log^*(2^{2^{2^{2^2}}}) = 5$$

Sets w/ Path compression: Amortized analysis

Amortized cost per operation of n set operations is $O(\log^* n)$ where

$$\log^* x = \text{smallest } k \text{ such that } \underbrace{log(log(\cdots log}_{k \text{ times}}(x)\cdots)) = 1$$

Note: $\log^*(x) \leq 5$ for virtually any n of practical relevance. Specifically,

$$\log^*(2^{65536}) = \log^*(2^{2^{2^{2^2}}}) = 5$$

Note that 2^{65536} is approximately a $20,000$ digit decimal number.

We will never be able to store input of that size, at least not in our universe. (Universe contains may be 10^{100} elementary particles.)

So, we might as well treat $\log^*(n)$ as $O(1)$.

Path compression: Amortized analysis (2)

- For n operations, *rank* of any node falls in the range $[0, \log n]$

Path compression: Amortized analysis (2)

- For n operations, *rank* of any node falls in the range $[0, \log n]$
- Divide this range into following groups:

$$[1], [2], [3–4], [5–16], [17–2^{16}], [2^{16} + 1–2^{65536}], \ldots$$

Each range is of the form $[k–2^{k-1}]$

Path compression: Amortized analysis (2)

- For n operations, *rank* of any node falls in the range $[0, \log n]$
- Divide this range into following groups:

$$[1], [2], [3\text{–}4], [5\text{–}16], [17\text{–}2^{16}], [2^{16}+1\text{–}2^{65536}], \ldots$$

  Each range is of the form $[k\text{–}2^{k-1}]$
- Let $G(v)$ be the group $rank(v)$ belongs to: $G(v) = \log^*(rank(v))$

Path compression: Amortized analysis (2)

- For n operations, *rank* of any node falls in the range $[0, \log n]$
- Divide this range into following groups:

$$[1], [2], [3\text{–}4], [5\text{–}16], [17\text{–}2^{16}], [2^{16}+1\text{–}2^{65536}], \ldots$$

  Each range is of the form $[k\text{–}2^{k-1}]$
- Let $G(v)$ be the group $rank(v)$ belongs to: $G(v) = \log^*(rank(v))$
- Note: when a node becomes a non-root, its rank never changes

Path compression: Amortized analysis (2)

- For n operations, *rank* of any node falls in the range $[0, \log n]$
- Divide this range into following groups:

$$[1], [2], [3\text{–}4], [5\text{–}16], [17\text{–}2^{16}], [2^{16}+1\text{–}2^{65536}], \ldots$$

  Each range is of the form $[k\text{–}2^{k-1}]$

- Let $G(v)$ be the group $rank(v)$ belongs to: $G(v) = \log^*(rank(v))$
- Note: when a node becomes a non-root, its rank never changes

Key Idea

Give an "allowance" to a node when it becomes a non-root. This allowance will be used to pay costs of path compression operations involving this node.

Path compression: Amortized analysis (2)

- For n operations, *rank* of any node falls in the range $[0, \log n]$
- Divide this range into following groups:

$$[1], [2], [3\text{–}4], [5\text{–}16], [17\text{–}2^{16}], [2^{16}+1\text{–}2^{65536}], \ldots$$

  Each range is of the form $[k\text{–}2^{k-1}]$
- Let $G(v)$ be the group $rank(v)$ belongs to: $G(v) = \log^*(rank(v))$
- Note: when a node becomes a non-root, its rank never changes

Key Idea

Give an "allowance" to a node when it becomes a non-root. This allowance will be used to pay costs of path compression operations involving this node.
For a node whose rank is in the range $[k\text{–}2^{k-1}]$, the allowance is 2^{k-1}.

Total allowance handed out

- Recall that number of nodes of rank r is at most $n/2^r$

Total allowance handed out

- Recall that number of nodes of rank r is at most $n/2^r$
- Recall that a node of rank is in the range $[k\text{–}2^{k-1}]$ is given an allowance of 2^{k-1}.

Total allowance handed out

- Recall that number of nodes of rank r is at most $n/2^r$
- Recall that a node of rank is in the range $[k\text{–}2^{k-1}]$ is given an allowance of 2^{k-1}.
- Total allowance handed out to nodes with ranks in the range $[k\text{–}2^{k-1}]$ is therefore given by

$$2^{k-1}\left(\frac{n}{2^k}+\frac{n}{2^{k+1}}+\cdots+\frac{n}{2^{2^{k-1}}}\right)\le 2^{k-1}\frac{n}{2^{k-1}}=n$$

Total allowance handed out

- Recall that number of nodes of rank r is at most $n/2^r$
- Recall that a node of rank is in the range $[k\text{–}2^{k-1}]$ is given an allowance of 2^{k-1}.
- Total allowance handed out to nodes with ranks in the range $[k\text{–}2^{k-1}]$ is therefore given by

$$2^{k-1}\left(\frac{n}{2^k}+\frac{n}{2^{k+1}}+\cdots+\frac{n}{2^{2^{k-1}}}\right)\leq 2^{k-1}\frac{n}{2^{k-1}}=n$$

- Since total number of ranges is $\log^* n$, total allowance granted to all nodes is $n\log^* n$

Total allowance handed out

- Recall that number of nodes of rank r is at most $n/2^r$
- Recall that a node of rank is in the range $[k\text{–}2^{k-1}]$ is given an allowance of 2^{k-1}.
- Total allowance handed out to nodes with ranks in the range $[k\text{–}2^{k-1}]$ is therefore given by

$$2^{k-1}\left(\frac{n}{2^k} + \frac{n}{2^{k+1}} + \cdots + \frac{n}{2^{2^{k-1}}}\right) \leq 2^{k-1}\frac{n}{2^{k-1}} = n$$

- Since total number of ranges is $\log^* n$, total allowance granted to all nodes is $n \log^* n$
- We will spread this cost across all n operations, thus contributing $O(\log^* n)$ to each operation.

Paying for all `find`'s

- Cost of a `find` equals # of parent pointers followed
  - Each pointer followed is updated to point to root of current set.

Paying for all `find`'s

- Cost of a `find` equals # of parent pointers followed
  - Each pointer followed is updated to point to root of current set.
- **Key idea:** Charge the cost of updating $\pi(p)$ to:
  - *Case 1:* If $G(\pi(p)) \neq G(p)$, then charge it to the current `find` operation
  - *Case 2:* Otherwise, charge it to p's allowance.

Paying for all `find`'s

- Cost of a `find` equals # of parent pointers followed
  - Each pointer followed is updated to point to root of current set.
- **Key idea:** Charge the cost of updating $\pi(p)$ to:
  - *Case 1:* If $G(\pi(p)) \neq G(p)$, then charge it to the current `find` operation
    - Can apply only $\log^* n$ times: a leaf's G-value is at least 1, and the root's G-value is at most $\log^* n$.
  - *Case 2:* Otherwise, charge it to p's allowance.

Paying for all `find`'s

- Cost of a `find` equals # of parent pointers followed
  - Each pointer followed is updated to point to root of current set.
- **Key idea:** Charge the cost of updating $\pi(p)$ to:
  - *Case 1:* If $G(\pi(p)) \neq G(p)$, then charge it to the current `find` operation
    - Can apply only $\log^* n$ times: a leaf's G-value is at least 1, and the root's G-value is at most $\log^* n$.
    - Adds only $\log^* n$ to cost of `find`
  - *Case 2:* Otherwise, charge it to p's allowance.

Paying for all `find`'s

- Cost of a `find` equals # of parent pointers followed
  - Each pointer followed is updated to point to root of current set.
- **Key idea:** Charge the cost of updating $\pi(p)$ to:
  - *Case 1:* If $G(\pi(p)) \neq G(p)$, then charge it to the current `find` operation
    - Can apply only $\log^* n$ times: a leaf's G-value is at least 1, and the root's G-value is at most $\log^* n$.
    - Adds only $\log^* n$ to cost of `find`
  - *Case 2:* Otherwise, charge it to p's allowance.
    - Need to show that we have enough allowance to to pay each time this case occurs.

Paying for all `find`'s (2)

- If $\pi(p)$ is updated, then the rank of p's parent increases.

Paying for all `find`'s (2)

- If $\pi(p)$ is updated, then the rank of p's parent increases.
- Let p be involved in a series of `find`'s, with q_i being its parent after the ith `find`. Note

$$rank(p) < rank(q_0) < rank(q_1) < rank(q_2) < \cdots$$

Paying for all `find`'s (2)

- If $\pi(p)$ is updated, then the rank of p's parent increases.
- Let p be involved in a series of `find`'s, with q_i being its parent after the ith `find`. Note

$$rank(p) < rank(q_0) < rank(q_1) < rank(q_2) < \cdots$$

- Let m be the number of such operations before p's parent has a higher G-value than p, i.e., $G(p) = G(q_m) < G(q_{m+1})$.

Paying for all `find`'s (2)

- If $\pi(p)$ is updated, then the rank of p's parent increases.
- Let p be involved in a series of `find`'s, with q_i being its parent after the ith `find`. Note
$$rank(p) < rank(q_0) < rank(q_1) < rank(q_2) < \cdots$$
- Let m be the number of such operations before p's parent has a higher G-value than p, i.e., $G(p) = G(q_m) < G(q_{m+1})$.
- Recall that
  - A $G(p) = r$ then r corresponds to a range $[k\text{–}2^{k-1}]$ where $k \leq rank(p) \leq 2^{k-1}$. Since $G(p) = G(q_m)$, $q_m \leq 2^{k-1}$

Paying for all `find`'s (2)

- If $\pi(p)$ is updated, then the rank of p's parent increases.
- Let p be involved in a series of `find`'s, with q_i being its parent after the ith `find`. Note
$$rank(p) < rank(q_0) < rank(q_1) < rank(q_2) < \cdots$$
- Let m be the number of such operations before p's parent has a higher G-value than p, i.e., $G(p) = G(q_m) < G(q_{m+1})$.
- Recall that
  - A $G(p) = r$ then r corresponds to a range $[k\text{–}2^{k-1}]$ where $k \leq rank(p) \leq 2^{k-1}$. Since $G(p) = G(q_m)$, $q_m \leq 2^{k-1}$
  - The allowance given to p is also 2^{k-1}

Paying for all `find`'s (2)

- If $\pi(p)$ is updated, then the rank of p's parent increases.
- Let p be involved in a series of `find`'s, with q_i being its parent after the ith `find`. Note

$$rank(p) < rank(q_0) < rank(q_1) < rank(q_2) < \cdots$$

- Let m be the number of such operations before p's parent has a higher G-value than p, i.e., $G(p) = G(q_m) < G(q_{m+1})$.
- Recall that
  - A $G(p) = r$ then r corresponds to a range $[k\text{–}2^{k-1}]$ where $k \leq rank(p) \leq 2^{k-1}$. Since $G(p) = G(q_m)$, $q_m \leq 2^{k-1}$
  - The allowance given to p is also 2^{k-1}

  So, there is enough allowance for all promotions up to m.

Paying for all `find`'s (2)

- If $\pi(p)$ is updated, then the rank of p's parent increases.
- Let p be involved in a series of `find`'s, with q_i being its parent after the ith `find`. Note

$$rank(p) < rank(q_0) < rank(q_1) < rank(q_2) < \cdots$$

- Let m be the number of such operations before p's parent has a higher G-value than p, i.e., $G(p) = G(q_m) < G(q_{m+1})$.
- Recall that
  - A $G(p) = r$ then r corresponds to a range $[k\text{–}2^{k-1}]$ where $k \leq rank(p) \leq 2^{k-1}$. Since $G(p) = G(q_m)$, $q_m \leq 2^{k-1}$
  - The allowance given to p is also 2^{k-1}

  So, there is enough allowance for all promotions up to m.
- After $m + 1$th `find`, the `find` operation will pay for pointer updates, as $G(\pi(p)) > G(p)$ from here on.