Lecture 21 - Amortized Analysis

Eric A. Autry

Last time: MST

This time: Amortized Analysis

Next time: NP Hard Problems

Project 3 due Tuesday the 27th.

Homework 7 due Thursday the 29th.

Project 4 (covering MST algorithms) assigned tomorrow will be due Thursday, December 6th.

Homework 7 Discussion

Homework 7 Discussion

- Greedy Ferries

Homework 7 Discussion

- Greedy Ferries
- Rural Cell Phone Towers

Homework 7 Discussion

- Greedy Ferries
- Rural Cell Phone Towers
- Greedy Party Planning

Kruskal's Algorithm Implementation

```
def Kruskal(graph, edges):
    # Initialize all singleton sets for each vertex.
    for vertex in graph:
        makeset(vertex)

    # Initialize the empty MST.
    X = {}

    # Sort the edges by weight.
    edges.sort()

    # Loop through the edges in increasing order.
    for e in edge:

        # If the min edge crosses a cut, add it to our MST.
        u, v = e.vertices
        if find(u) ≠ find(v):
            X.append(e)
            union(u,v)
```

$$
\begin{aligned}
|V| \cdot \texttt{makeset} \quad + \quad 2|E| \cdot \texttt{find} \quad + \quad (|V|-1) \cdot \texttt{union} \\
= |V| \cdot O(1) + 2|E| \log |V| + (|V|-1) \log |E| \\
\in O(|E| \log |V|).
\end{aligned}
$$

Disjoint Sets

We need a data structure with three operations:

- `makeset(v)`:
  create a singleton set containing vertex v

  ```
  def makeset(v):
      v.π = v
      v.height = 0
  ```

- `find(v)`:
  find which set vertex v belongs to (used for finding cuts)

  ```
  def find(v):
      while v != v.π:
          v = v.π
      return v
  ```

- `union(u,v)`:
  merge the sets containing vertices u and v

Union-by-Rank

```
def union(u,v):
    # First, find the root of the tree for u
    # and the tree for v.
    ru = find(u)
    rv = find(v)

    # If the sets are already the same, return.
    if ru == rv:
        return

    # Make shorter set point to taller set.
    if ru.height > rv.height:
        rv.π = ru
    elif ru.height < rv.height:
        ru.π = rv
    else:
        # Same height, break tie.
        ru.π = rv

        # Tree got taller, increment rv.height.
        rv.height += 1
    return
```

Path Compression

Idea: why follow parent pointers all the way up to the root?

Path Compression

Idea: why follow parent pointers all the way up to the root?

Why not just have each vertex point directly to the root?

Path Compression

Idea: why follow parent pointers all the way up to the root?

Why not just have each vertex point directly to the root?

When we perform a find operation, we are finding a vertex's root.

Path Compression

Idea: why follow parent pointers all the way up to the root?

Why not just have each vertex point directly to the root?

When we perform a find operation, we are finding a vertex's root.

So let's **compress** the path during a find operation by updating the 'parent' of each vertex on the path.

Path Compression

Idea: why follow parent pointers all the way up to the root?

Why not just have each vertex point directly to the root?

When we perform a find operation, we are finding a vertex's root.

So let's **compress** the path during a find operation by updating the 'parent' of each vertex on the path.

```
def find(v):
    # If we are not at the root.
    if v != v.π:
        # Set our parent to be the root,
        # which is also the root of our parent!
        v.π = find(v.π)

    # Return the root, which is now our parent!
    return v.π
```

Path Compression

```
def find(v):
    # If we are not at the root.
    if v != v.π:
        # Set our parent to be the root,
        # which is also the root of our parent!
        v.π = find(v.π)
    # Return the root, which is now our parent!
    return v.π
```

What is the new runtime using this path compression?

Path Compression

```
def find(v):
    # If we are not at the root.
    if v != v.π:
        # Set our parent to be the root,
        # which is also the root of our parent!
        v.π = find(v.π)
    # Return the root, which is now our parent!
    return v.π
```

What is the new runtime using this path compression?

To compute the runtime, we will need to look at **sequences** of `find` and `union` operations, starting from an empty data structure, and determine the **average cost per operation**.

Path Compression

```
def find(v):
    # If we are not at the root.
    if v != v.π:
        # Set our parent to be the root,
        # which is also the root of our parent!
        v.π = find(v.π)
    # Return the root, which is now our parent!
    return v.π
```

What is the new runtime using this path compression?

To compute the runtime, we will need to look at **sequences** of `find` and `union` operations, starting from an empty data structure, and determine the **average cost per operation.**

This is called an **amortized analysis.**

Three Methods of Amortized Analysis

Three Methods of Amortized Analysis

- Aggregation/Aggregate Method
  - Find the aggregate (total) cost of n operations, and use that to compute the average cost per operation.

Three Methods of Amortized Analysis

- Aggregation/Aggregate Method
  - Find the aggregate (total) cost of n operations, and use that to compute the average cost per operation.
- Accounting Method
  - Assign each operation an amount of ‘work currency’ (we will call these rubles) that can pay for future operations.

Three Methods of Amortized Analysis

- Aggregation/Aggregate Method
  - Find the aggregate (total) cost of n operations, and use that to compute the average cost per operation.
- Accounting Method
  - Assign each operation an amount of ‘work currency’ (we will call these rubles) that can pay for future operations.
- Potential Method
  - The saved ‘work currency’ is instead tracked through a potential function that is dependent on the state of the data structure.

Dynamically Allocated Arrays

Think back to Homework 4...

Dynamically Allocated Arrays

Think back to Homework 4...

Say we have an array and want to compute the cost of n append operations using the algorithm where we double the size of the array when necessary.

Dynamically Allocated Arrays

Think back to Homework 4...

Say we have an array and want to compute the cost of n append operations using the algorithm where we double the size of the array when necessary.

What you did in Homework 4 was the aggregation method:

Dynamically Allocated Arrays

Think back to Homework 4...

Say we have an array and want to compute the cost of n append operations using the algorithm where we double the size of the array when necessary.

What you did in Homework 4 was the aggregation method:

- 1 cost per initial write for a total of n.

Dynamically Allocated Arrays

Think back to Homework 4...

Say we have an array and want to compute the cost of n append operations using the algorithm where we double the size of the array when necessary.

What you did in Homework 4 was the aggregation method:

- 1 cost per initial write for a total of n.
- 2 cost each time something is copied, which happens when the array's size is a power of 2.

Dynamically Allocated Arrays

Think back to Homework 4...

Say we have an array and want to compute the cost of n append operations using the algorithm where we double the size of the array when necessary.

What you did in Homework 4 was the aggregation method:

- 1 cost per initial write for a total of n.
- 2 cost each time something is copied, which happens when the array's size is a power of 2.
- But this can happen at most k times where $k = \log_2 n$.

Dynamically Allocated Arrays

Think back to Homework 4...

Say we have an array and want to compute the cost of n append operations using the algorithm where we double the size of the array when necessary.

What you did in Homework 4 was the aggregation method:

- 1 cost per initial write for a total of n.
- 2 cost each time something is copied, which happens when the array's size is a power of 2.
- But this can happen at most k times where $k = \log_2 n$.
- So when n is a power of 2, the number of total copies is:

$$2^0 + 2^1 + \cdots + 2^k = 2^{k+1} - 1 = 2n - 1$$

Dynamically Allocated Arrays

Think back to Homework 4...

Say we have an array and want to compute the cost of n append operations using the algorithm where we double the size of the array when necessary.

What you did in Homework 4 was the aggregation method:

- 1 cost per initial write for a total of n.
- 2 cost each time something is copied, which happens when the array's size is a power of 2.
- But this can happen at most k times where $k = \log_2 n$.
- So when n is a power of 2, the number of total copies is:

$$2^0 + 2^1 + \cdots + 2^k = 2^{k+1} - 1 = 2n - 1$$

- This means that the total work was $n + 2(2n - 1)$, which means on average $5 - 2/n \in O(1)$ amortized cost.

Dynamically Allocated Arrays

Now let's analyze this algorithms using the accounting method:

Dynamically Allocated Arrays

Now let's analyze this algorithms using the accounting method:

Give each element 5 rubles when it is first appended.

Dynamically Allocated Arrays

Now let's analyze this algorithms using the accounting method:

Give each element 5 rubles when it is first appended.

- 1 ruble will pay for the first write when it is appended.

Dynamically Allocated Arrays

Now let's analyze this algorithms using the accounting method:

Give each element 5 rubles when it is first appended.

- 1 ruble will pay for the first write when it is appended.
- 2 rubles will pay for the first time it is copied.

Dynamically Allocated Arrays

Now let's analyze this algorithms using the accounting method:

Give each element 5 rubles when it is first appended.

- 1 ruble will pay for the first write when it is appended.
- 2 rubles will pay for the first time it is copied.
- 2 rubles will pay for a copy operation for an element in the first half of the array during that first copy.

Dynamically Allocated Arrays

Now let's analyze this algorithms using the accounting method:

Give each element 5 rubles when it is first appended.

- 1 ruble will pay for the first write when it is appended.
- 2 rubles will pay for the first time it is copied.
- 2 rubles will pay for a copy operation for an element in the first half of the array during that first copy.

This covers all of the necessary work, meaning that we have $5 \in O(1)$ amortized cost.

Dynamically Allocated Arrays

Using the potential method:

Dynamically Allocated Arrays

Using the potential method: We define the amortized cost of an operation to be

$$T_{amortized} = T_{actual} + C\left(\Phi_{after} - \Phi_{before}\right),$$

- where T_{actual} is the actual work done by the given operation, and Φ is the potential function.

Dynamically Allocated Arrays

Using the potential method: We define the amortized cost of an operation to be

$$T_{amortized} = T_{actual} + C\left(\Phi_{after} - \Phi_{before}\right),$$

- where T_{actual} is the actual work done by the given operation, and Φ is the potential function.

For the dynamically allocated array, define:

$$\Phi = 2n - N, \quad C = 2$$

- where n is the number of elements currently in the array and N is the current size of the array.

Dynamically Allocated Arrays

Using the potential method: We define the amortized cost of an operation to be

$$T_{amortized} = T_{actual} + C\left(\Phi_{after} - \Phi_{before}\right),$$

- where T_{actual} is the actual work done by the given operation, and Φ is the potential function.

For the dynamically allocated array, define:

$$\Phi = 2n - N, \quad C = 2$$

- where n is the number of elements currently in the array and N is the current size of the array.

When $n < N$, the potential increases by only 2, and the actual cost is $T_{actual} = 1$ write. So the amortized cost of this operation is $T_{amortized} = 1 + 2(2) = 5 \in O(1)$.

Dynamically Allocated Arrays

We define the amortized cost of an operation to be

$$T_{amortized} = T_{actual} + C\left(\Phi_{after} - \Phi_{before}\right).$$

For the dynamically allocated array, define:

$$\Phi = 2n - N, \quad C = 2$$

where n is the number of elements currently in the array and N is the current size of the array.

Dynamically Allocated Arrays

We define the amortized cost of an operation to be

$$T_{amortized} = T_{actual} + C\left(\Phi_{after} - \Phi_{before}\right).$$

For the dynamically allocated array, define:

$$\Phi = 2n - N, \quad C = 2$$

where n is the number of elements currently in the array and N is the current size of the array.

- But, when $n = N$, the array needs to be resized.

Dynamically Allocated Arrays

We define the amortized cost of an operation to be

$$T_{amortized} = T_{actual} + C\left(\Phi_{after} - \Phi_{before}\right).$$

For the dynamically allocated array, define:

$$\Phi = 2n - N, \quad C = 2$$

where n is the number of elements currently in the array and N is the current size of the array.

- But, when $n = N$, the array needs to be resized.
- When the array with n elements is resized, the actual cost is $2n + 1$ for the n copy operations.

Dynamically Allocated Arrays

We define the amortized cost of an operation to be

$$T_{amortized} = T_{actual} + C\left(\Phi_{after} - \Phi_{before}\right).$$

For the dynamically allocated array, define:

$$\Phi = 2n - N, \quad C = 2$$

where n is the number of elements currently in the array and N is the current size of the array.

- But, when $n = N$, the array needs to be resized.
- When the array with n elements is resized, the actual cost is $2n + 1$ for the n copy operations.
- However, the potential function decreases from n to 2 because the size of the array N has doubled, then we inserted one new element.

Dynamically Allocated Arrays

We define the amortized cost of an operation to be

$$T_{amortized} = T_{actual} + C\left(\Phi_{after} - \Phi_{before}\right).$$

For the dynamically allocated array, define:

$$\Phi = 2n - N, \quad C = 2$$

where n is the number of elements currently in the array and N is the current size of the array.

- But, when $n = N$, the array needs to be resized.
- When the array with n elements is resized, the actual cost is $2n + 1$ for the n copy operations.
- However, the potential function decreases from n to 2 because the size of the array N has doubled, then we inserted one new element.
- So the amortized cost is $T_{amortized} = 2n + 1 + 2(2 - n) = 5 \in O(1)$.

Minqueues

A Minqueue is an abstract data type that supports the following operations:

Minqueues

A Minqueue is an abstract data type that supports the following operations:

- `Enqueue`(x): inserts the number x into the Minqueue.

Minqueues

A Minqueue is an abstract data type that supports the following operations:

- `Enqueue`(x): inserts the number x into the Minqueue.
- `Deque`(): removes the element that has been in the Minqueue the longest.

Minqueues

A Minqueue is an abstract data type that supports the following operations:

- `Enqueue`(x): inserts the number x into the Minqueue.
- `Deque`(): removes the element that has been in the Minqueue the longest.
- `Find-Min`(): returns the smallest value in the Minqueue *without* removing it.

Minqueues

A Minqueue is an abstract data type that supports the following operations:

- `Enqueue`(x): inserts the number x into the Minqueue.
- `Deque`(): removes the element that has been in the Minqueue the longest.
- `Find-Min`(): returns the smallest value in the Minqueue *without* removing it.

We can implement a Minqueue using two regular queues, what we will call the primary queue and the helper queue.

Minqueues

A Minqueue is an abstract data type that supports the following operations:

- `Enqueue`(x): inserts the number x into the Minqueue.
- `Deque`(): removes the element that has been in the Minqueue the longest.
- `Find-Min`(): returns the smallest value in the Minqueue *without* removing it.

We can implement a Minqueue using two regular queues, what we will call the primary queue and the helper queue.

The primary queue will simply act like a regular queue to keep track of first in, first out ordering.

Minqueues

A Minqueue is an abstract data type that supports the following operations:

- `Enqueue`(x): inserts the number x into the Minqueue.
- `Deque`(): removes the element that has been in the Minqueue the longest.
- `Find-Min`(): returns the smallest value in the Minqueue *without* removing it.

We can implement a Minqueue using two regular queues, what we will call the primary queue and the helper queue.

The primary queue will simply act like a regular queue to keep track of first in, first out ordering.

The helper queue will always contain a subset of elements in the primary queue that are stored in sorted order.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

- We then check the value in front of it in the helper queue.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

- We then check the value in front of it in the helper queue.
- If the new value is smaller than its predecessor, then the predecessor is removed from the helper queue.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

- We then check the value in front of it in the helper queue.
- If the new value is smaller than its predecessor, then the predecessor is removed from the helper queue.
- We repeat until the new value is correctly located in the helper queue.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

- We then check the value in front of it in the helper queue.
- If the new value is smaller than its predecessor, then the predecessor is removed from the helper queue.
- We repeat until the new value is correctly located in the helper queue.
- Note: it will still be at the rear of the helper queue because we removed all elements larger than it.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

- We then check the value in front of it in the helper queue.
- If the new value is smaller than its predecessor, then the predecessor is removed from the helper queue.
- We repeat until the new value is correctly located in the helper queue.
- Note: it will still be at the rear of the helper queue because we removed all elements larger than it.
- Why? Because they will be removed from the primary queue before the new value will, so they can never be the minimum value in the queue.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

- We then check the value in front of it in the helper queue.
- If the new value is smaller than its predecessor, then the predecessor is removed from the helper queue.
- We repeat until the new value is correctly located in the helper queue.
- Note: it will still be at the rear of the helper queue because we removed all elements larger than it.
- Why? Because they will be removed from the primary queue before the new value will, so they can never be the minimum value in the queue.

When an element is `dequeued`, it is removed from the front of the primary queue.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

- We then check the value in front of it in the helper queue.
- If the new value is smaller than its predecessor, then the predecessor is removed from the helper queue.
- We repeat until the new value is correctly located in the helper queue.
- Note: it will still be at the rear of the helper queue because we removed all elements larger than it.
- Why? Because they will be removed from the primary queue before the new value will, so they can never be the minimum value in the queue.

When an element is `dequeued`, it is removed from the front of the primary queue.

- We also check to see if it needs to be removed from the helper queue.

Minqueues

When an element is `enqueued`, it is placed in the primary queue, and at the rear of the helper queue.

- We then check the value in front of it in the helper queue.
- If the new value is smaller than its predecessor, then the predecessor is removed from the helper queue.
- We repeat until the new value is correctly located in the helper queue.
- Note: it will still be at the rear of the helper queue because we removed all elements larger than it.
- Why? Because they will be removed from the primary queue before the new value will, so they can never be the minimum value in the queue.

When an element is `dequeued`, it is removed from the front of the primary queue.

- We also check to see if it needs to be removed from the helper queue.

For `Find-Min`, we simply look at the front of the helper queue.

Minqueues

We will prove that any sequence of n operations will have $O(1)$ amortized cost per operation.

Minqueues

We will prove that any sequence of n operations will have $O(1)$ amortized cost per operation.

Aggregation Method:

Minqueues

We will prove that any sequence of n operations will have $O(1)$ amortized cost per operation.

Aggregation Method:

Clearly finding the minimum n times will cost a total of n reads.

Minqueues

We will prove that any sequence of n operations will have $O(1)$ amortized cost per operation.

Aggregation Method:

Clearly finding the minimum n times will cost a total of n reads.

Running n `dequeue` operations will cost a total of $O(n)$ because at worst each operation will:

Minqueues

We will prove that any sequence of n operations will have $O(1)$ amortized cost per operation.

Aggregation Method:

Clearly finding the minimum n times will cost a total of n reads.

Running n `dequeue` operations will cost a total of $O(n)$ because at worst each operation will:

- Remove an element from the primary queue.

Minqueues

We will prove that any sequence of n operations will have $O(1)$ amortized cost per operation.

Aggregation Method:

Clearly finding the minimum n times will cost a total of n reads.

Running n `dequeue` operations will cost a total of $O(n)$ because at worst each operation will:

- Remove an element from the primary queue.
- Check the helper queue.

Minqueues

We will prove that any sequence of n operations will have $O(1)$ amortized cost per operation.

Aggregation Method:

Clearly finding the minimum n times will cost a total of n reads.

Running n `dequeue` operations will cost a total of $O(n)$ because at worst each operation will:

- Remove an element from the primary queue.
- Check the helper queue.
- Remove an element from the helper queue.

Minqueues

Aggregation Method:

Minqueues

Aggregation Method:

Finally, for n `enqueue` operations, there can be a total of $2n$ insertions.

Minqueues

Aggregation Method:

Finally, for n `enqueue` operations, there can be a total of $2n$ insertions.

- What about the removal process when we insert into the helper queue?

Minqueues

Aggregation Method:

Finally, for n `enqueue` operations, there can be a total of $2n$ insertions.

- What about the removal process when we insert into the helper queue?
- Each inserted value can only be removed at most once.

Minqueues

Aggregation Method:

Finally, for n `enqueue` operations, there can be a total of $2n$ insertions.

- What about the removal process when we insert into the helper queue?
- Each inserted value can only be removed at most once.
- So no more than n total removals.

Minqueues

Aggregation Method:

Finally, for n `enqueue` operations, there can be a total of $2n$ insertions.

- What about the removal process when we insert into the helper queue?
- Each inserted value can only be removed at most once.
- So no more than n total removals.

So the total of n `Find-Min` **plus** n `dequeue` **plus** n `enqueue`:

$$n + n + 3n \in O(n).$$

Minqueues

Aggregation Method:

Finally, for n `enqueue` operations, there can be a total of $2n$ insertions.

- What about the removal process when we insert into the helper queue?
- Each inserted value can only be removed at most once.
- So no more than n total removals.

So the total of n `Find-Min` **plus** n `dequeue` **plus** n `enqueue`:

$$n + n + 3n \in O(n).$$

We average to get $O(1)$ amortized time.

Minqueues

Accounting Method:

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

- 1 ruble is spent to insert it into the primary queue.

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

- 1 ruble is spent to insert it into the primary queue.
- 1 ruble is spent to insert it into the helper queue.

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

- 1 ruble is spent to insert it into the primary queue.
- 1 ruble is spent to insert it into the helper queue.
- 1 ruble is spent for the comparison in the helper queue that does not result in a removal.

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

- 1 ruble is spent to insert it into the primary queue.
- 1 ruble is spent to insert it into the helper queue.
- 1 ruble is spent for the comparison in the helper queue that does not result in a removal.
- 1 ruble is spent to remove it from the primary queue.

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

- 1 ruble is spent to insert it into the primary queue.
- 1 ruble is spent to insert it into the helper queue.
- 1 ruble is spent for the comparison in the helper queue that does not result in a removal.
- 1 ruble is spent to remove it from the primary queue.
- 1 ruble is spent to check for removal from the helper queue.

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

- 1 ruble is spent to insert it into the primary queue.
- 1 ruble is spent to insert it into the helper queue.
- 1 ruble is spent for the comparison in the helper queue that does not result in a removal.
- 1 ruble is spent to remove it from the primary queue.
- 1 ruble is spent to check for removal from the helper queue.
- 1 ruble is spent to remove it from the helper queue.

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

- 1 ruble is spent to insert it into the primary queue.
- 1 ruble is spent to insert it into the helper queue.
- 1 ruble is spent for the comparison in the helper queue that does not result in a removal.
- 1 ruble is spent to remove it from the primary queue.
- 1 ruble is spent to check for removal from the helper queue.
- 1 ruble is spent to remove it from the helper queue.

How are we covering the cost for the removal process when we insert into the helper queue?

Minqueues

Accounting Method:

First, since `Find-Min` is a very simple function, we will simply charge 1 ruble for each of these operations.

Now, we assign each value 6 rubles when it is first `enqueued`.

- 1 ruble is spent to insert it into the primary queue.
- 1 ruble is spent to insert it into the helper queue.
- 1 ruble is spent for the comparison in the helper queue that does not result in a removal.
- 1 ruble is spent to remove it from the primary queue.
- 1 ruble is spent to check for removal from the helper queue.
- 1 ruble is spent to remove it from the helper queue.

How are we covering the cost for the removal process when we insert into the helper queue?

- Each element that gets removed pays 1 of its rubles.

Minqueues

Potential Method:

Minqueues

Potential Method:

Think-Pair-Share: what potential function should we use?

Minqueues

Potential Method:

Think-Pair-Share: what potential function should we use?

Define the potential function to be the size of the helper queue.

Minqueues

Potential Method:

Think-Pair-Share: what potential function should we use?

Define the potential function to be the size of the helper queue.

- Note: it starts at 0, can never be negative, and can be at most n after n operations.

Minqueues

Potential Method:

Think-Pair-Share: what potential function should we use?

Define the potential function to be the size of the helper queue.

- Note: it starts at 0, can never be negative, and can be at most n after n operations.

Aside from the removal process when we insert into the helper queue, all `dequeue` and `enqueue` operations are $O(1)$ time (basic insert and removal operations).

Minqueues

Potential Method:

Think-Pair-Share: what potential function should we use?

Define the potential function to be the size of the helper queue.

- Note: it starts at 0, can never be negative, and can be at most n after n operations.

Aside from the removal process when we insert into the helper queue, all `dequeue` and `enqueue` operations are $O(1)$ time (basic insert and removal operations).

Additionally, these operations will only result in a constant increase in the potential.

Minqueues

Potential Method:

Minqueues

Potential Method:

Now consider the removal process when we insert into the helper queue.

Minqueues

Potential Method:

Now consider the removal process when we insert into the helper queue.

- We have to pay for some number of removal operations to get rid of the larger elements in the helper queue.

Minqueues

Potential Method:

Now consider the removal process when we insert into the helper queue.

- We have to pay for some number of removal operations to get rid of the larger elements in the helper queue.
- But removing elements from the helper queue decreases our potential function.

Minqueues

Potential Method:

Now consider the removal process when we insert into the helper queue.

- We have to pay for some number of removal operations to get rid of the larger elements in the helper queue.
- But removing elements from the helper queue decreases our potential function.
- These costs offset each other.

Minqueues

Potential Method:

Now consider the removal process when we insert into the helper queue.

- We have to pay for some number of removal operations to get rid of the larger elements in the helper queue.
- But removing elements from the helper queue decreases our potential function.
- These costs offset each other.

So we have $O(1)$ amortized cost for any sequence of n operations.

More discussion of Homework 7

Two Stacks can make a Queue:

More discussion of Homework 7

Two Stacks can make a Queue:

I ask you to use the accounting method, which I find to be the easiest.

More discussion of Homework 7

Two Stacks can make a Queue:

I ask you to use the accounting method, which I find to be the easiest.

But it would be good practice to try the other methods.

Amortized Analysis of Path Compression

Even though our paths are being compressed, we maintain the vertex.height variable as before.

Amortized Analysis of Path Compression

Even though our paths are being compressed, we maintain the vertex.height variable as before.

Just now it doesn't actually correspond to the height of the tree.

Amortized Analysis of Path Compression

Even though our paths are being compressed, we maintain the vertex.height variable as before.

Just now it doesn't actually correspond to the height of the tree.

But, we can still say that there are at most $n/2^k$ vertices of height k.

Amortized Analysis of Path Compression

Even though our paths are being compressed, we maintain the vertex.height variable as before.

Just now it doesn't actually correspond to the height of the tree.

But, we can still say that there are at most $n/2^k$ vertices of height k.

We now observe:

Amortized Analysis of Path Compression

Even though our paths are being compressed, we maintain the vertex.height variable as before.

Just now it doesn't actually correspond to the height of the tree.

But, we can still say that there are at most $n/2^k$ vertices of height k.

We now observe:

- Each find will be constant time for vertices that are at the root or whose paths are already fully compressed.

Amortized Analysis of Path Compression

Even though our paths are being compressed, we maintain the vertex.height variable as before.

Just now it doesn't actually correspond to the height of the tree.

But, we can still say that there are at most $n/2^k$ vertices of height k.

We now observe:

- Each find will be constant time for vertices that are at the root or whose paths are already fully compressed.
- Once a path is compressed, it will only need to be compressed again if there is a new root, i.e., if the root of the tree has an increased vertex.height.

Inverse Tower Function

We will define a function called the tower function (also called the tetration operation):

$$tower(0) = 1, \quad tower(1) = 2, \quad tower(n) = 2^{tower(n-1)}$$

Inverse Tower Function

We will define a function called the tower function (also called the tetration operation):

$$tower(0) = 1, \quad tower(1) = 2, \quad tower(n) = 2^{tower(n-1)}$$

You can see that this is a very fast growing function:

$$tower(5) = 2^{2^{2^{2^{2}}}} = 2^{65536}$$

Inverse Tower Function

We will define a function called the tower function (also called the tetration operation):

$$tower(0) = 1, \quad tower(1) = 2, \quad tower(n) = 2^{tower(n-1)}$$

You can see that this is a very fast growing function:

$$tower(5) = 2^{2^{2^{2^{2}}}} = 2^{65536}$$

The inverse tower function, which we denote by $\log^*(n)$ is very slow growing, and defined by

$$\log^*(n) = \begin{cases} 0, & n \leq 1, \\ 1 + \log^*(n-1), & n > 1. \end{cases}$$

Inverse Tower Function

We will define a function called the tower function (also called the tetration operation):

$$tower(0) = 1, \quad tower(1) = 2, \quad tower(n) = 2^{tower(n-1)}$$

You can see that this is a very fast growing function:

$$tower(5) = 2^{2^{2^{2^{2}}}} = 2^{65536}$$

The inverse tower function, which we denote by $\log^*(n)$ is very slow growing, and defined by

$$\log^*(n) = \begin{cases} 0, & n \leq 1, \\ 1 + \log^*(n-1), & n > 1. \end{cases}$$

So that

$$\log^*(2^{80}) \sim 4.$$

Amortized Analysis of Path Compression

Now it's time for some complex accounting:

Amortized Analysis of Path Compression

Now it's time for some complex accounting:

- Consider a sequence of m `find` operations for a graph that has n vertices.

Amortized Analysis of Path Compression

Now it's time for some complex accounting:

- Consider a sequence of m `find` operations for a graph that has n vertices.
- We will assign each vertex an amount of rubles when it ceases to be the root of its set, giving out a total of $n \log^* n$ rubles across all vertices.

Amortized Analysis of Path Compression

Now it's time for some complex accounting:

- Consider a sequence of m `find` operations for a graph that has n vertices.
- We will assign each vertex an amount of rubles when it ceases to be the root of its set, giving out a total of $n \log^* n$ rubles across all vertices.
- We can then show that each `find` will take $O(\log^* n)$ steps plus some of the rubles.

Amortized Analysis of Path Compression

Now it's time for some complex accounting:

- Consider a sequence of m `find` operations for a graph that has n vertices.
- We will assign each vertex an amount of rubles when it ceases to be the root of its set, giving out a total of $n \log^* n$ rubles across all vertices.
- We can then show that each `find` will take $O(\log^* n)$ steps plus some of the rubles.
- Then the total work is $O((n+m)\log^* n)$.

Amortized Analysis of Path Compression

Now it's time for some complex accounting:

- Consider a sequence of m `find` operations for a graph that has n vertices.
- We will assign each vertex an amount of rubles when it ceases to be the root of its set, giving out a total of $n \log^* n$ rubles across all vertices.
- We can then show that each `find` will take $O(\log^* n)$ steps plus some of the rubles.
- Then the total work is $O((n + m) \log^* n)$.
- But with Kruskal's Algorithm, we do $m = 2|E|$ `find` operations on a graph with $|V|$ vertices, for a runtime of $O(|E| \log^* |V|)$ when the edges are already sorted.

Amortized Analysis of Path Compression

Now it's time for some complex accounting:

- Consider a sequence of m `find` operations for a graph that has n vertices.
- We will assign each vertex an amount of rubles when it ceases to be the root of its set, giving out a total of $n \log^* n$ rubles across all vertices.
- We can then show that each `find` will take $O(\log^* n)$ steps plus some of the rubles.
- Then the total work is $O((n+m)\log^* n)$.
- But with Kruskal's Algorithm, we do $m = 2|E|$ `find` operations on a graph with $|V|$ vertices, for a runtime of $O(|E| \log^* |V|)$ when the edges are already sorted.

This is a very complex proof, and will be saved for the posted lecture notes (you will not be tested on this, but it is an interesting results).

Inverse Ackermann Function

It turns out that this $\log^* n$ bound is not tight.

Inverse Ackermann Function

It turns out that this $\log^* n$ bound is not tight.

There is a better bound that uses what is known as the Inverse Ackermann Function: $\alpha(n)$.

Inverse Ackermann Function

It turns out that this $\log^* n$ bound is not tight.

There is a better bound that uses what is known as the Inverse Ackermann Function: $\alpha(n)$.

It grows slower than the inverse tower function.

Inverse Ackermann Function

It turns out that this $\log^* n$ bound is not tight.

There is a better bound that uses what is known as the Inverse Ackermann Function: $\alpha(n)$.

It grows slower than the inverse tower function.

It actually shows up as a bound in a number of famous algorithms.

Inverse Ackermann Function

It turns out that this $\log^* n$ bound is not tight.

There is a better bound that uses what is known as the Inverse Ackermann Function: $\alpha(n)$.

It grows slower than the inverse tower function.

It actually shows up as a bound in a number of famous algorithms.

In practice however, it is important to remember:

Inverse Ackermann Function

It turns out that this $\log^* n$ bound is not tight.

There is a better bound that uses what is known as the Inverse Ackermann Function: $\alpha(n)$.

It grows slower than the inverse tower function.

It actually shows up as a bound in a number of famous algorithms.

In practice however, it is important to remember:

$$n < 2^{65536} \quad \Rightarrow \quad \alpha(n) \ \leq \ \log^*(n) \ \leq \ 5.$$