

Lecture 8: Amortized Analysis

Michael Dinitz

September 18, 2025

601.433/633 Introduction to Algorithms

Introduction

Typically been considering “static” or “one-shot” problems: given input, compute correct output as efficiently as possible.

Introduction

Typically been considering “static” or “one-shot” problems: given input, compute correct output as efficiently as possible.

Data structures: *sequence* of operations!

- ▶ Dictionary: insert, insert, insert, lookup, insert, lookup, lookup, ...

Introduction

Typically been considering “static” or “one-shot” problems: given input, compute correct output as efficiently as possible.

Data structures: *sequence* of operations!

- ▶ Dictionary: insert, insert, insert, lookup, insert, lookup, lookup, ...

Last time: analyzed the (worst-case) cost of each operation.

What about (worst-case) cost of *sequence* of operations?

Definition & Example

Definition

The *amortized cost* of a sequence of n operations is the total cost of the sequence divided by n .

“Average cost per operation” (but no randomness!)

Definition & Example

Definition

The *amortized cost* of a sequence of **n** operations is the total cost of the sequence divided by **n** .

“Average cost per operation” (but no randomness!)

Example: 100 operations of cost **1**, then **1** operation of cost **100**

- ▶ Normal worst-case analysis: **100**
- ▶ Amortized cost: **$200/101 \approx 2$**

Definition & Example

Definition

The *amortized cost* of a sequence of n operations is the total cost of the sequence divided by n .

“Average cost per operation” (but no randomness!)

Example: 100 operations of cost **1**, then **1** operation of cost **100**

- ▶ Normal worst-case analysis: **100**
- ▶ Amortized cost: **$200/101 \approx 2$**

If we care about total time (e.g., using data structure in larger algorithm) then worst-case too pessimistic

Amortized Analysis

Still want worst-case, but worst-case over *sequences* rather than single operations.

Maybe only possible way to have an expensive operation is to have a bunch of cheap operations: amortized cost always small!

Amortized Analysis

Still want worst-case, but worst-case over *sequences* rather than single operations.

Maybe only possible way to have an expensive operation is to have a bunch of cheap operations: amortized cost always small!

Definition

If the amortized cost of *every* sequence of n operations is at most $f(n)$, then the *amortized cost* or *amortized complexity* of the algorithm is at most $f(n)$.

Example: Stack From Array

Stack Using Array

Stack:

- ▶ Last In First Out (LIFO)
- ▶ Push: add element to stack
- ▶ Pop: Remove the most recently added element.

Stack Using Array

Stack:

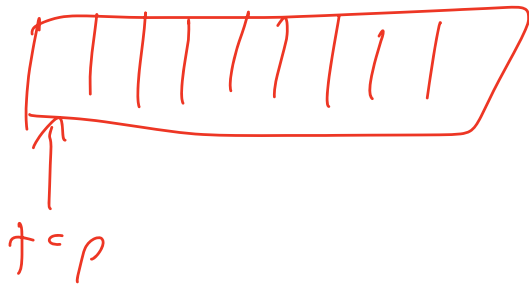
- ▶ Last In First Out (LIFO)
- ▶ Push: add element to stack
- ▶ Pop: Remove the most recently added element.

Building a stack with an array A:

Stack Using Array

Stack:

- ▶ Last In First Out (LIFO)
- ▶ Push: add element to stack
- ▶ Pop: Remove the most recently added element.



Building a stack with an array A:

- ▶ Initialize: $\text{top} = 0$
- ▶ Push(x): $A[\text{top}] = x$; $\text{top}++$
- ▶ Pop: $\text{top}--$; Return $A[\text{top}]$

End of Array

What if array is full (n elements)?

End of Array

What if array is full (n elements)?

Make new, bigger array, copy old array over

- ▶ Cost: free to create new array, each copy costs **1**
- ▶ Worst case: a single Push could cost $\Omega(n)$!



End of Array

What if array is full (n elements)?

Make new, bigger array, copy old array over

- ▶ Cost: free to create new array, each copy costs **1**
- ▶ Worst case: a single Push could cost $\Omega(n)$!

New array has size $n + 1$:

End of Array

What if array is full (n elements)?

Make new, bigger array, copy old array over

- ▶ Cost: free to create new array, each copy costs **1**
- ▶ Worst case: a single Push could cost $\Omega(n)$!

New array has size $n + 1$:

- ▶ Sequence of n Push operations. Total cost: $\sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2)$.
- ▶ Amortized cost: $\Theta(n)$ (same as worst single operation!)

Better Idea

Instead of increasing from n to $n + 1$:

Better Idea

Instead of increasing from n to $n + 1$: increase to $2n$

Better Idea

Instead of increasing from n to $n + 1$: increase to $2n$

Consider *any* sequence of n operations.

- ▶ Have to double when array has size $2, 4, 8, 16, 32, 64, \dots, \cancel{[\log n]}$ } $\log n$
- ▶ *Total* time spent doubling: at most $\sum_{i=1}^{\lceil \log n \rceil} 2^i \leq 2n = \Theta(n)$
- ▶ Any operation that doesn't cause a doubling costs $O(1)$
- ▶ Total cost at most $O(n) + n \cdot O(1) = O(n)$
- ▶ Amortized cost at most $O(1)$

Better Idea

Instead of increasing from n to $n + 1$: increase to $2n$

Consider *any* sequence of n operations.

- ▶ Have to double when array has size $2, 4, 8, 16, 32, 64, \dots, \lfloor \log n \rfloor$
- ▶ *Total* time spent doubling: at most $\sum_{i=1}^{\lfloor \log n \rfloor} 2^i \leq 2n = \Theta(n)$
- ▶ Any operation that doesn't cause a doubling costs $O(1)$
- ▶ Total cost at most $O(n) + n \cdot O(1) = O(n)$
- ▶ Amortized cost at most $O(1)$

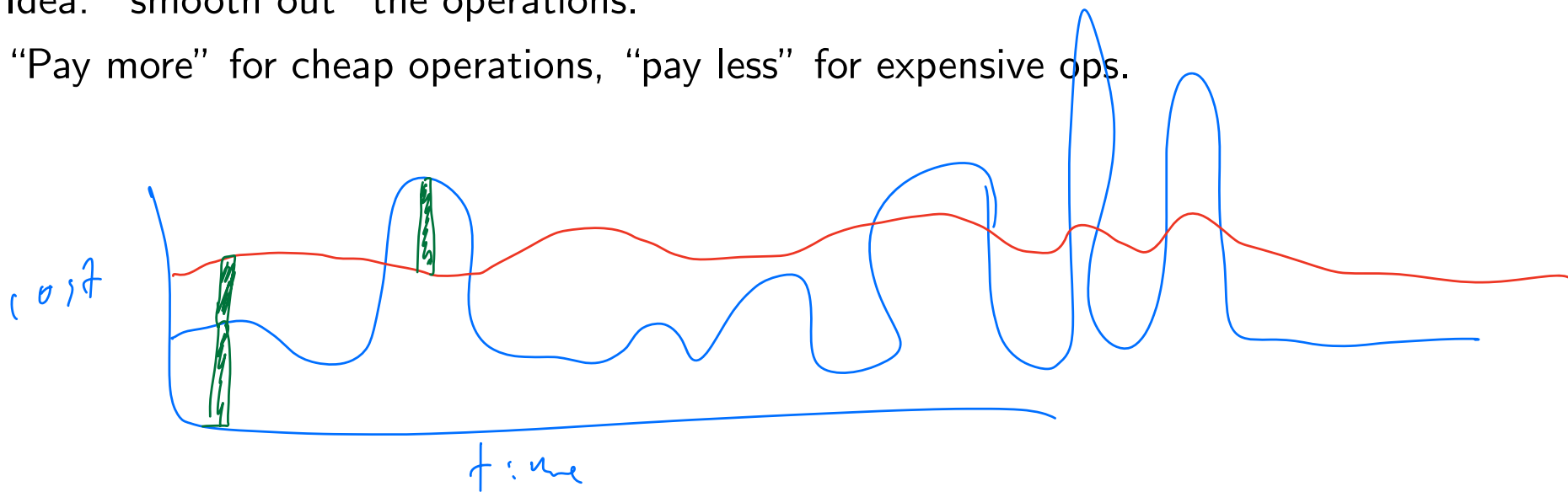
Amortized analysis explains why it's better to double than add 1 !

More Complicated Analysis: Piggy Banks and Potentials

Basic Bank: Informal

Can be hard to give good bound directly on total cost.

- ▶ Lots of variance: some operations very expensive, some very cheap.
- ▶ Idea: “smooth out” the operations.
- ▶ “Pay more” for cheap operations, “pay less” for expensive ops.



Basic Bank: Informal

Can be hard to give good bound directly on total cost.

- ▶ Lots of variance: some operations very expensive, some very cheap.
- ▶ Idea: “smooth out” the operations.
- ▶ “Pay more” for cheap operations, “pay less” for expensive ops.

Use a “bank” to keep track of this

- ▶ Cheap operation: add to the bank
- ▶ Expensive operation: take from the bank

Basic Bank: Informal

Can be hard to give good bound directly on total cost.

- ▶ Lots of variance: some operations very expensive, some very cheap.
- ▶ Idea: “smooth out” the operations.
- ▶ “Pay more” for cheap operations, “pay less” for expensive ops.

Use a “bank” to keep track of this

- ▶ Cheap operation: add to the bank
- ▶ Expensive operation: take from the bank

Charge cheap operations more, use extra to pay for expensive operations

Basic Bank: Formal

Bank L .

- ▶ Initially $L = 0$
- ▶ L_i = value of bank after operation i (so $L_0 = 0$).

Basic Bank: Formal

Bank L .

- ▶ Initially $L = 0$
- ▶ L_i = value of bank after operation i (so $L_0 = 0$).

Operation i :

- ▶ Cost c_i
- ▶ “Amortized cost” $c'_i = c_i + \Delta L = c_i + L_i - L_{i-1}$

Basic Bank: Formal

Bank L .

- ▶ Initially $L = 0$
- ▶ L_i = value of bank after operation i (so $L_0 = 0$).

Operation i :

- ▶ Cost c_i
- ▶ “Amortized cost” $c'_i = c_i + \Delta L = c_i + L_i - L_{i-1} \implies c_i = c'_i + L_{i-1} - L_i$

Basic Bank: Formal

Bank L .

- ▶ Initially $L = 0$
- ▶ L_i = value of bank after operation i (so $L_0 = 0$).

Operation i :

- ▶ Cost c_i
- ▶ “Amortized cost” $c'_i = c_i + \Delta L = c_i + L_i - L_{i-1} \implies c_i = c'_i + L_{i-1} - L_i$

Total cost of sequence:

$$\sum_{i=1}^n c_i = \sum_{i=1}^n (c'_i + L_{i-1} - L_i) = \sum_{i=1}^n c'_i + \sum_{i=1}^n (L_{i-1} - L_i) = \left(\sum_{i=1}^n c'_i \right) + L_0 - L_n$$

Basic Bank: Formal

Bank L .

- ▶ Initially $L = 0$
- ▶ L_i = value of bank after operation i (so $L_0 = 0$).

Operation i :

- ▶ Cost c_i
- ▶ “Amortized cost” $c'_i = c_i + \Delta L = c_i + L_i - L_{i-1} \implies c_i = c'_i + L_{i-1} - L_i$

Total cost of sequence:

$$\sum_{i=1}^n c_i = \sum_{i=1}^n (c'_i + L_{i-1} - L_i) = \sum_{i=1}^n c'_i + \sum_{i=1}^n (L_{i-1} - L_i) = \left(\sum_{i=1}^n c'_i \right) + L_0 - L_n$$

So if $L_0 = 0$ and $L_n \geq 0$ (bank not negative): $\sum_{i=1}^n c_i \leq \sum_{i=1}^n c'_i$

Basic Bank: Formal

Bank L .

- ▶ Initially $L = 0$
- ▶ L_i = value of bank after operation i (so $L_0 = 0$).

Operation i :

- ▶ Cost c_i
- ▶ “Amortized cost” $c'_i = c_i + \Delta L = c_i + L_i - L_{i-1} \implies c_i = c'_i + L_{i-1} - L_i$

Total cost of sequence:

$$\sum_{i=1}^n c_i = \sum_{i=1}^n (c'_i + L_{i-1} - L_i) = \sum_{i=1}^n c'_i + \sum_{i=1}^n (L_{i-1} - L_i) = \left(\sum_{i=1}^n c'_i \right) + L_0 - L_n$$

So if $L_0 = 0$ and $L_n \geq 0$ (bank not negative): $\sum_{i=1}^n c_i \leq \sum_{i=1}^n c'_i \leq \sum_{i=1}^n f(i) = n \cdot f(n)$

- ▶ If $c'_i \leq f(n)$ for all i , then “true” amortized cost $(\sum_{i=1}^n c_i)/n$ also at most $f(n)$!

Variants

Multiple banks

- ▶ Sometimes easier to keep track of / think about.
- ▶ No real difference: could think of one bank = sum of all banks

Variants

Multiple banks

- ▶ Sometimes easier to keep track of / think about.
- ▶ No real difference: could think of one bank = sum of all banks

Potential Functions:

- ▶ “Bank analogy”: we choose how much to deposit/withdraw.
- ▶ New analogy: “potential energy”. Function of state of system.
- ▶ Rename L to Φ : all previous analysis works same!
- ▶ Sometimes easier to think about: just define once at the beginning, instead of for each operation.

Example: Binary Counter

Binary Counter

Super simple setup: binary counter stored in array **A**.

- ▶ Least significant bit in **A[0]**, then **A[1]**, ...
- ▶ Don't worry about length of array (infinite, or long enough)
- ▶ Only operation is increment.
- ▶ Costs **1** to flip any bit.



Binary Counter

Super simple setup: binary counter stored in array \mathbf{A} .

- ▶ Least significant bit in $\mathbf{A}[0]$, then $\mathbf{A}[1]$, ...
- ▶ Don't worry about length of array (infinite, or long enough)
- ▶ Only operation is increment.
- ▶ Costs $\mathbf{1}$ to flip any bit.

n increments. Cost of most expensive increment:

Binary Counter

Super simple setup: binary counter stored in array \mathbf{A} .

- ▶ Least significant bit in $\mathbf{A}[0]$, then $\mathbf{A}[1]$, ...
- ▶ Don't worry about length of array (infinite, or long enough)
- ▶ Only operation is increment.
- ▶ Costs $\mathbf{1}$ to flip any bit.

n increments. Cost of most expensive increment: $\Theta(\log n)$.

1111
10000

Binary Counter

Super simple setup: binary counter stored in array \mathbf{A} .

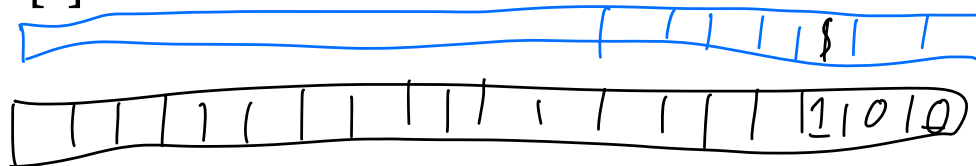
- ▶ Least significant bit in $\mathbf{A}[0]$, then $\mathbf{A}[1]$, ...
- ▶ Don't worry about length of array (infinite, or long enough)
- ▶ Only operation is increment.
- ▶ Costs $\mathbf{1}$ to flip any bit.

n increments. Cost of most expensive increment: $\Theta(\log n)$.

What about amortized cost?

Banks

Bank for every bit $A[i]$



Flip bit i from **0** to **1**: add \$ to bank for i

Flip bit i from **1** to **0**: remove \$ from bank for i

- ▶ No bank ever negative (induction)

Analysis

Do an increment, flips k bits \implies true cost is k .

- ▶ # **0**'s flipped to **1**:
- ▶ # **1**'s flipped to **0**:

— — — — 0 1

Analysis

Do an increment, flips k bits \implies true cost is k .

- ▶ # **0**'s flipped to **1**: **1**
- ▶ # **1**'s flipped to **0**: $k - 1$

Analysis

Do an increment, flips k bits \implies true cost is k .

- ▶ # **0**'s flipped to **1**: **1**
- ▶ # **1**'s flipped to **0**: $k - 1$

Flipping **1** to **0** paid for by bank! Costs **1**, bank decreases by **1**

Analysis

Do an increment, flips k bits \implies true cost is k .

- ▶ # **0**'s flipped to **1**: **1**
- ▶ # **1**'s flipped to **0**: $k - 1$

Flipping **1** to **0** paid for by bank! Costs **1**, bank decreases by **1**

\implies amortized cost at most **1** (cost of flipping **0** to **1**) plus **1** (increase in bank for that bit)
 $= 2$

Analysis

Do an increment, flips k bits \implies true cost is k .

- ▶ # **0**'s flipped to **1**: **1**
- ▶ # **1**'s flipped to **0**: $k - 1$

Flipping **1** to **0** paid for by bank! Costs **1**, bank decreases by **1**

\implies amortized cost at most **1** (cost of flipping **0** to **1**) plus **1** (increase in bank for that bit)
 $= 2$

Global: Change in *total* bank is $-(k - 1) + 1 = -k + 2$

\implies amortized cost $= c + \Delta L = k + (-k + 2) = 2$

Analysis

Do an increment, flips k bits \implies true cost is k .

- ▶ # **0**'s flipped to **1**: **1**
- ▶ # **1**'s flipped to **0**: $k - 1$

Flipping **1** to **0** paid for by bank! Costs **1**, bank decreases by **1**

\implies amortized cost at most **1** (cost of flipping **0** to **1**) plus **1** (increase in bank for that bit)
 $= 2$

Global: Change in *total* bank is $-(k - 1) + 1 = -k + 2$

\implies amortized cost $= c + \Delta L = k + (-k + 2) = 2$

Potential function: let $\Phi = \#1$'s in counter.

\implies amortized cost $= c + \Delta \Phi = k + (-k + 2) = 2$

Example: Simple Dictionary

Setup

Same dictionary problem as last lecture (insert, lookup).

- ▶ Can we do something simple with just arrays (no trees)?
- ▶ Give up on worst-case: try for amortized.
 - ▶ Sorted array: inserts $\Omega(n)$ amortized (i 'th insert could take time $\Omega(i)$)
 - ▶ Unsorted array: lookups $\Omega(n)$ amortized

Setup

Same dictionary problem as last lecture (insert, lookup).

- ▶ Can we do something simple with just arrays (no trees)?
- ▶ Give up on worst-case: try for amortized.
 - ▶ Sorted array: inserts $\Omega(n)$ amortized (i 'th insert could take time $\Omega(i)$)
 - ▶ Unsorted array: lookups $\Omega(n)$ amortized

Solution: array of arrays!

- ▶ $A[i]$ either empty or a *sorted* array of *exactly* 2^i elements
- ▶ No relationship between arrays

Setup

Same dictionary problem as last lecture (insert, lookup).

- ▶ Can we do something simple with just arrays (no trees)?
- ▶ Give up on worst-case: try for amortized.
 - ▶ Sorted array: inserts $\Omega(n)$ amortized (i 'th insert could take time $\Omega(i)$)
 - ▶ Unsorted array: lookups $\Omega(n)$ amortized

Solution: array of arrays!

- ▶ $A[i]$ either empty or a *sorted* array of *exactly* 2^i elements
- ▶ No relationship between arrays

Example: insert **1 – 11**

$$A[0] = [5]$$

$$A[1] = [2, 8]$$

$$A[2] = \emptyset$$

$$A[3] = [1, 3, 4, 6, 7, 9, 10, 11]$$

Algorithm

Note: With n inserts, at most $\log n$ arrays.

Algorithm

Note: With n inserts, at most $\log n$ arrays.

Lookup(x)

Algorithm

Note: With n inserts, at most $\log n$ arrays.

Lookup(x)

- ▶ Binary search in each (nonempty) array
- ▶ Time at most $\sum_{i=0}^{\lceil \log n \rceil} \log(2^i) = \Theta(\log^2 n)$

Algorithm

Note: With n inserts, at most $\log n$ arrays.

Lookup(x)

- ▶ Binary search in each (nonempty) array
- ▶ Time at most $\sum_{i=0}^{\lceil \log n \rceil} \log(2^i) = \Theta(\log^2 n)$

Insert(x):

Algorithm

Note: With n inserts, at most $\log n$ arrays.

Lookup(x)

- ▶ Binary search in each (nonempty) array
- ▶ Time at most $\sum_{i=0}^{\lceil \log n \rceil} \log(2^i) = \Theta(\log^2 n)$

Insert(x):

- ▶ Create array $B = [x]$
- ▶ $i = 0$
- ▶ While $A[i] \neq \emptyset$:
 - ▶ Merge B and $A[i]$ to get B
 - ▶ Set $A[i] = \emptyset$
 - ▶ $i++$
- ▶ Set $A[i] = B$

Algorithm

Note: With n inserts, at most $\log n$ arrays.

Lookup(x)

- ▶ Binary search in each (nonempty) array
- ▶ Time at most $\sum_{i=0}^{\lceil \log n \rceil} \log(2^i) = \Theta(\log^2 n)$

Insert(x):

- ▶ Create array $B = [x]$
- ▶ $i = 0$
- ▶ While $A[i] \neq \emptyset$:
 - ▶ Merge B and $A[i]$ to get B
 - ▶ Set $A[i] = \emptyset$
 - ▶ $i++$
- ▶ Set $A[i] = B$

Example: insert 12 into

$A[0] = [5]$

$A[1] = [2, 8]$

$A[2] = \emptyset$

$A[3] = [1, 3, 4, 6, 7, 9, 10, 11]$

Algorithm

Note: With n inserts, at most $\log n$ arrays.

Lookup(x)

- ▶ Binary search in each (nonempty) array
- ▶ Time at most $\sum_{i=0}^{\lceil \log n \rceil} \log(2^i) = \Theta(\log^2 n)$

Insert(x):

- ▶ Create array $B = [x]$
- ▶ $i = 0$
- ▶ While $A[i] \neq \emptyset$:
 - ▶ Merge B and $A[i]$ to get B
 - ▶ Set $A[i] = \emptyset$
 - ▶ $i++$
- ▶ Set $A[i] = B$

Example: insert 12 into

$A[0] = [5]$ (1, 12)
 $A[1] = [2, 8]$ (2, 5, 8, 12)
 $A[2] = \emptyset$ ←
 $A[3] = [1, 3, 4, 6, 7, 9, 10, 11]$

$A[0] = \emptyset$
 $A[1] = \emptyset$
 $A[2] = [2, 5, 8, 12]$
 $A[3] = [1, 3, 4, 6, 7, 9, 10, 11]$

Analysis

Concrete costs:

- ▶ Merging two arrays of size m costs $2m$

Analysis

Concrete costs:

- ▶ Merging two arrays of size m costs $2m$

Worst case:

- ▶ Might need to do a merge for every array if all full
- ▶ Time $\sum_{i=0}^{\lceil \log n \rceil} (2 \cdot 2^i) = \Theta(n)$

Analysis

Concrete costs:

- ▶ Merging two arrays of size m costs $2m$

Worst case:

- ▶ Might need to do a merge for every array if all full
- ▶ Time $\sum_{i=0}^{\lceil \log n \rceil} (2 \cdot 2^i) = \Theta(n)$

Amortized:

- ▶ Merge arrays of length 2^i one out of every 2^i inserts
- ▶ So after n inserts, have merged arrays of length 1 at most n times, arrays of length 2 at most $n/2$ times, arrays of length 4 at most $n/4$ times, ...

Analysis

Concrete costs:

- ▶ Merging two arrays of size m costs $2m$

Worst case:

- ▶ Might need to do a merge for every array if all full
- ▶ Time $\sum_{i=0}^{\lceil \log n \rceil} (2 \cdot 2^i) = \Theta(n)$

Amortized:

- ▶ Merge arrays of length 2^i one out of every 2^i inserts
- ▶ So after n inserts, have merged arrays of length 1 at most n times, arrays of length 2 at most $n/2$ times, arrays of length 4 at most $n/4$ times, ...
- ▶ Total cost at most

$$\sum_{i=1}^{\lceil \log n \rceil} \frac{n}{2^{i-1}} 2^{i+1} = \Theta(n \log n)$$

Analysis

Concrete costs:

- ▶ Merging two arrays of size m costs $2m$

Worst case:

- ▶ Might need to do a merge for every array if all full
- ▶ Time $\sum_{i=0}^{\lceil \log n \rceil} (2 \cdot 2^i) = \Theta(n)$

Amortized:

- ▶ Merge arrays of length 2^i one out of every 2^i inserts
- ▶ So after n inserts, have merged arrays of length 1 at most n times, arrays of length 2 at most $n/2$ times, arrays of length 4 at most $n/4$ times, ...
- ▶ Total cost at most

$$\sum_{i=1}^{\lceil \log n \rceil} \frac{n}{2^{i-1}} 2^{i+1} = \Theta(n \log n)$$

- ▶ Amortized cost at most $\Theta(\log n)$!

Multiple Operations

How do we define amortized analysis of data structures with multiple operations?

Definition

If structure supports k operations, say that operation i has amortized cost at most α_i if for every sequence which performs with at most m_i operations of type i , the total cost is at most $\sum_{i=1}^k \alpha_i m_i$.

Multiple Operations

How do we define amortized analysis of data structures with multiple operations?

Definition

If structure supports k operations, say that operation i has amortized cost at most α_i if for every sequence which performs with at most m_i operations of type i , the total cost is at most $\sum_{i=1}^k \alpha_i m_i$.

- ▶ When analyzing multiple operations, need to use the same bank/potential for all of them!
- ▶ With multiple operations, bounds not necessarily unique. Different amortization schemes could yield different bounds, all of which are correct and non-contradictory.