Amortized Analysis
A Motivating Analogy
Doing the Dishes

• What do I do with a dirty dish or kitchen utensil?

• **Option 1:** Wash it by hand.

• **Option 2:** Put it in the dishwasher rack, then run the dishwasher if it’s full.
Doing the Dishes

- What do I do with a dirty dish or kitchen utensil?
  - **Option 1:** Wash it by hand.
  - **Option 2:** Put it in the dishwasher rack, then run the dishwasher if it’s full.
Doing the Dishes

• What do I do with a dirty dish or kitchen utensil?

• **Option 1:** Wash it by hand.

• **Option 2:** Put it in the dishwasher rack, then run the dishwasher if it’s full.
Doing the Dishes

• Washing every individual dish and utensil by hand is way slower than using the dishwasher, but I always have access to my plates and kitchen utensils.

• Running the dishwasher is faster in aggregate, but means I may have to wait a bit for dishes to be ready.

• (This is an example of a tradeoff between throughput and latency.)
**Key Idea:** Design data structures that trade *per-operation efficiency* for *overall efficiency*. 
Where We’re Going

- **Amortized Analysis (Today)**
  - A little accounting trickery never hurt anyone, right?

- **Binomial Heaps (Thursday)**
  - A fast, flexible priority queue that’s a great building block for more complicated structures.

- **Fibonacci Heaps (Next Tuesday)**
  - A priority queue optimized for graph algorithms that, at least in theory, leads to optimal implementations.
Outline for Today

- **Amortized Analysis**
  - Trading worst-case efficiency for aggregate efficiency.

- **Examples of Amortization**
  - Three motivating data structures and algorithms.

- **Potential Functions**
  - Quantifying messiness and formalizing costs.

- **Performing Amortized Analyses**
  - How to show our examples are indeed fast.
Three Examples
Two-Stack Queues

Dynamic Arrays

Building B-Trees
The Two-Stack Queue

- Out
- In
The Two-Stack Queue

Out

1

In
The Two-Stack Queue

Out

In

2

1
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

Out

4

In

3
2
1
The Two-Stack Queue

Out

In

4

3

2

1
The Two-Stack Queue
The Two-Stack Queue

3
4
Out

2
1
In
The Two-Stack Queue

Out
3
4

In
1
2
The Two-Stack Queue

In

2
3
4

Out

1
In
The Two-Stack Queue

2
3
4

Out

1

In
The Two-Stack Queue

Out

1

In
The Two-Stack Queue
The Two-Stack Queue

Out

1
2
3
4

In
The Two-Stack Queue

1
2
3
4

Out

In
The Two-Stack Queue

In

1

Out

2

3

4
The Two-Stack Queue

In

1

2 3 4

Out

In
The Two-Stack Queue
The Two-Stack Queue

![Diagram of two stacks: Out and In with numbers 1, 2, 3, 4, 5]
The Two-Stack Queue

Out

In

1 2

3 4

6 5
The Two-Stack Queue
The Two-Stack Queue

<table>
<thead>
<tr>
<th>Out</th>
<th>In</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
The Two-Stack Queue

Out

In

1 2 3

4

5

6

7
The Two-Stack Queue
The Two-Stack Queue

Out

In

1 2 3 4

5

6

7
The Two-Stack Queue

Out

In

1  2  3  4

5  6  7
The Two-Stack Queue
The Two-Stack Queue

1 2 3 4

Out

7

In

6 5
The Two-Stack Queue

Out

1 2 3 4

7

In

6

5
The Two-Stack Queue

1 2 3 4

Out

5

In

6

7
The Two-Stack Queue
The Two-Stack Queue

Out

In

1  2  3  4

6
7
5
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

In

Out

1 2 3 4

5

6

7
The Two-Stack Queue

6
7

1 2 3 4 5
The Two-Stack Queue

- Maintain an **In** stack and an **Out** stack.
- To enqueue an element, push it onto the **In** stack.
- To dequeue an element:
  - If the **Out** stack is nonempty, pop it.
  - If the **Out** stack is empty, pop elements from the **In** stack, pushing them into the **Out** stack. Then dequeue as usual.
The Two-Stack Queue

---

**Clean Dishes**

---

**Dirty Dishes**
The Two-Stack Queue

Clean Dishes

Dirty Dishes

1
The Two-Stack Queue

Clean Dishes

Dirty Dishes

2

1
The Two-Stack Queue
The Two-Stack Queue

Our dirty dishes are piling up because we didn’t do any work to clean them when we added them in.

Clean Dishes

Dirty Dishes

4
3
2
1
The Two-Stack Queue

Clean Dishes

Dirty Dishes

1

2

3

4
The Two-Stack Queue

Clean Dishes

Dirty Dishes
The Two-Stack Queue

**Clean Dishes**

4

**Dirty Dishes**

3
2
1
The Two-Stack Queue

Clean Dishes

Dirty Dishes

4

3

2

1
The Two-Stack Queue

Clean Dishes

Dirty Dishes

3

4

2

1
The Two-Stack Queue

Clean Dishes

Dirty Dishes

3

4

2

1
The Two-Stack Queue

Clean Dishes

Dirty Dishes

3
4

2
1
The Two-Stack Queue

Clean Dishes

2

3

4

Dirty Dishes

1
The Two-Stack Queue

Clean Dishes

Dirty Dishes
The Two-Stack Queue

Clean Dishes

2
3
4

Dirty Dishes

1
The Two-Stack Queue

- **Clean Dishes**
  - 1
  - 2
  - 3
  - 4

- **Dirty Dishes**
The Two-Stack Queue

Clean Dishes

1
2
3
4

Dirty Dishes
The Two-Stack Queue

Clean Dishes

1
2
3
4

Dirty Dishes

We just cleaned up our entire mess and are back to a pristine state.
The Two-Stack Queue

Clean Dishes

Dirty Dishes

1
2
3
4
The Two-Stack Queue

Clean Dishes

1
2
3
4

Dirty Dishes
The Two-Stack Queue

**Clean Dishes**

1

2

3

4

**Dirty Dishes**
The Two-Stack Queue

1

Clean Dishes

2
3
4

Dirty Dishes

1
The Two-Stack Queue

Clean Dishes

Dirty Dishes

1

2

3

4
The Two-Stack Queue

We need to do some “cleanup” on this before it’ll be useful. It’s fast to add it here because we’re deferring that work.
The Two-Stack Queue

Clean Dishes

1 2

Dirty Dishes

3 4 5 6
The Two-Stack Queue

Clean Dishes

3
4

Dirty Dishes

6
5

1
2
The Two-Stack Queue

Clean Dishes

1 2 3

Dirty Dishes

4

6 5
The Two-Stack Queue

Clean Dishes

Dirty Dishes

1  2  3

4

5

6

7
The Two-Stack Queue

Clean Dishes

1  2  3

Dirty Dishes

4
5  6  7
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

Clean Dishes

1 2 3 4

Dirty Dishes

5 6
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

Clean Dishes

Dirty Dishes

1 2 3 4

6 7 5
The Two-Stack Queue

Clean Dishes

Dirty Dishes

1 2 3 4

5

6 7
The Two-Stack Queue
The Two-Stack Queue

Clean Dishes

Dirty Dishes

1  2  3  4
The Two-Stack Queue

Clean Dishes

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Dirty Dishes

| 1 | 2 | 3 | 4 |
The Two-Stack Queue

- **Clean Dishes**
  - 5
  - 6
  - 7

- **Dirty Dishes**
  - 1
  - 2
  - 3
  - 4
The Two-Stack Queue

Clean Dishes

6
7

Dirty Dishes

1 2 3 4 5
The Two-Stack Queue

- Each enqueue takes time $O(1)$.
  - Just push an item onto the $\text{In}$ stack.
- Dequeues can vary in their runtime.
  - Could be $O(1)$ if the $\text{Out}$ stack isn’t empty.
  - Could be $\Theta(n)$ if the $\text{Out}$ stack is empty.
The Two-Stack Queue

- Each enqueue takes time $O(1)$.
  - Just push an item onto the $In$ stack.
- Dequeues can vary in their runtime.
  - Could be $O(1)$ if the $Out$ stack isn’t empty.
  - Could be $\Theta(n)$ if the $Out$ stack is empty.
The Two-Stack Queue

• Each enqueue takes time $O(1)$.
  • Just push an item onto the $\text{In}$ stack.
• Dequeues can vary in their runtime.
  • Could be $O(1)$ if the $\text{Out}$ stack isn’t empty.
  • Could be $\Theta(n)$ if the $\text{Out}$ stack is empty.
The Two-Stack Queue

- **Intuition:** We only do expensive dequeues after a long run of cheap enqueues.
- Think “dishwasher:” we very slowly introduce a lot of dirty dishes that get cleaned up all at once.
- Provided we clean up all the dirty dishes at once, and provided that dirty dishes accumulate slowly, this is a fast strategy!

```
3
...
n-1
n
Out

In
```
The Two-Stack Queue

- **Key Fact:** Any series of $n$ operations on an (initially empty) two-stack queue will take time $O(n)$.

- **Why?**

  Answer at [https://pollev.com/cs166spr23](https://pollev.com/cs166spr23)
The Two-Stack Queue

- **Key Fact:** Any series of \( n \) operations on an (initially empty) two-stack queue will take time \( O(n) \).

- **Why?**
  - Each item is pushed into at most two stacks and popped from at most two stacks.
  - Adding up the work done per element across all \( n \) operations, we can do at most \( O(n) \) work.
The Two-Stack Queue

• It’s correct but misleading to say the cost of a dequeue is $O(n)$.
  • This is comparatively rare.
• It’s wrong, but useful, to pretend the cost of a dequeue is $O(1)$.
  • Some operations take more time than this.
  • However, if we pretend each operation takes time $O(1)$, then the sum of all the costs never underestimates the total.
• **Question:** What’s an honest, accurate way to describe the runtime of the two-stack queue?

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>...</td>
<td>n-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*Out*  

*In*
Two-Stack Queues

Dynamic Arrays

Building B-Trees
Dynamic Arrays

• A *dynamic array* is the most common way to implement a list of values.

• Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

- A *dynamic array* is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

- A *dynamic array* is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

- A *dynamic array* is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

• A **dynamic array** is the most common way to implement a list of values.

• Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

• A *dynamic array* is the most common way to implement a list of values.

• Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

- A *dynamic array* is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

• A *dynamic array* is the most common way to implement a list of values.

• Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

- A **dynamic array** is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

- A *dynamic array* is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

- A *dynamic array* is the most common way to implement a list of values.
- Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.

<table>
<thead>
<tr>
<th>H</th>
<th>He</th>
<th>Li</th>
<th>Be</th>
<th>B</th>
<th>C</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
</table>
A dynamic array is the most common way to implement a list of values.

Maintain an array slightly bigger than the one you need. When you run out of space, double the array size and copy the elements over.
Dynamic Arrays

- Most appends to a dynamic array take time $O(1)$.
- Infrequently, we do $\Theta(n)$ work to copy all $n$ elements from the old array to a new one.
- Think “dishwasher:”
  - We slowly accumulate “messes” (filled slots).
  - We periodically do a large “cleanup” (copying the array).
- **Claim:** The cost of doing $n$ appends to an initially empty dynamic array is always $O(n)$.
Dynamic Arrays

- **Claim:** Appending $n$ elements always takes time $O(n)$.
- The array doubles at sizes $2^0$, $2^1$, $2^2$, ..., etc.
- The very last doubling is at the largest power of two less than $n$. This is at most $2^\lfloor\log_2 n\rfloor$. (Do you see why?)
- Total work done across all doubling is at most
  \[
  2^0 + 2^1 + \ldots + 2^\lfloor\log_2 n\rfloor = 2^\lfloor\log_2 n\rfloor + 1 - 1 \\
  \leq 2^\lfloor\log_2 n\rfloor + 1 \\
  = 2n.
  \]
Dynamic Arrays

- It’s correct but misleading to say the cost of an append is $O(n)$.
  - This is comparatively rare.
- It’s wrong, but useful, to pretend that the cost of an append is $O(1)$.
  - Some operations take more time than this.
  - However, pretending each operation takes $O(1)$ time never underestimates the true total runtime.
- **Question:** What’s an honest, accurate way to describe the runtime of the dynamic array?
Two-Stack Queues

Dynamic Arrays

Building B-Trees
Two-Stack Queues

Dynamic Arrays

Building B-Trees
Building B-Trees

- You’re given a sorted list of $n$ values and a value of $b$.
- What’s the most efficient way to construct a B-tree of order $b$ holding these $n$ values?
- **One Option:** Think really hard, calculate the shape of a B-tree of order $b$ with $n$ elements in it, then place the items into that B-tree in sorted order.
- Is there an easier option?
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

• **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

• **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

• **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1**: Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

• **Idea 1**: Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

• **Idea 1**: Insert the items into an empty B-tree in sorted order.
Building B-Trees

• **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.

![B-tree diagram with nodes 0, 1, 2, 3, and 4, 5]
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

• **Idea 1**: Insert the items into an empty B-tree in sorted order.
Building B-Trees

• *Idea 1:* Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
Building B-Trees

- **Idea 1:** Insert the items into an empty B-tree in sorted order.
- Cost: $\Omega(n \log_b n)$, due to the top-down search.
- **Can we do better?**
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.

![B-tree diagram]

- root: 3
- left child: 1 (0, 2)
- right child: 5, 7 (4, 6, 8, 9)
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

• **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.

```
0 2 4 6 8 10 11
```

```
3
```

```
1 5 7 9
```
Building B-Trees

**Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

• **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.
Building B-Trees

- **Idea 2:** Since all insertions will happen at the rightmost leaf, store a pointer to that leaf. Add new values by appending to this leaf, then doing any necessary splits.

- **Question:** How fast is this?
Building B-Trees

- The cost of an insert varies based on the shape of the tree.
  - If no splits are required, the cost is $O(1)$.
  - If one split is required, the cost is $O(b)$.
  - If we have to split all the way up, the cost is $O(b \log_b n)$.
- Using our worst-case cost across $n$ inserts gives a runtime bound of $O(nb \log_b n)$
- **Claim:** The cost of $n$ inserts is always $O(n)$. 
Building B-Trees

- Of all the $n$ insertions into the tree, a roughly $\frac{1}{b}$ fraction will split a node in the bottom layer of the tree (a leaf).
- Of those, roughly a $\frac{1}{b}$ fraction will split a node in the layer above that.
- Of those, roughly a $\frac{1}{b}$ fraction will split a node in the layer above that.
- (etc.)
Building B-Trees

- Total number of splits:
Building B-Trees

- Total number of splits:
  \[ \frac{n}{b} \cdot (1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot (\ldots)))) \]
Building B-Trees

• Total number of splits:

\[ \frac{n}{b} \cdot \left( 1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot \left( 1 + \frac{1}{b} \cdot (\ldots) \right) \right) \]

\[ = \frac{n}{b} \cdot \left( 1 + \frac{1}{b} + \frac{1}{b^2} + \frac{1}{b^3} + \frac{1}{b^4} + \ldots \right) \]
Building B-Trees

- Total number of splits:

\[
\frac{n}{b} \cdot \left(1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot (...) )) \right)
\]

\[
= \frac{n}{b} \cdot \left(1 + \frac{1}{b} + \frac{1}{b^2} + \frac{1}{b^3} + \frac{1}{b^4} + \ldots \right)
\]

\[
= \frac{n}{b} \cdot \Theta(1)
\]
Building B-Trees

• Total number of splits:

\[ \frac{n}{b} \cdot (1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot (\cdots)))) \]

\[ = \frac{n}{b} \cdot (1 + \frac{1}{b} + \frac{1}{b^2} + \frac{1}{b^3} + \frac{1}{b^4} + \cdots) \]

\[ = \frac{n}{b} \cdot \Theta (1) \]

\[ = \Theta \left( \frac{n}{b} \right) \]
Building B-Trees

- Total number of splits:

\[
\frac{n}{b} \cdot (1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot (1 + \frac{1}{b} \cdot (\ldots))))
\]

\[
= \frac{n}{b} \cdot (1 + \frac{1}{b} + \frac{1}{b^2} + \frac{1}{b^3} + \frac{1}{b^4} + \ldots)
\]

\[
= \frac{n}{b} \cdot \Theta(1)
\]

\[
= \Theta\left(\frac{n}{b}\right)
\]

- Total cost of those splits: \(\Theta(n)\).
Building B-Trees

- It is correct but misleading to say the cost of an insert is $O(b \log_b n)$.
  - This is comparatively rare.
- It is wrong, but useful, to pretend that the cost of an insert is $O(1)$.
  - Some operations take more time than this.
  - However, pretending each insert takes time $O(1)$ never underestimates the total amount of work done across all operations.
- **Question:** What’s an honest, accurate way to describe the cost of inserting one more value?
Amortized Analysis
The Setup

• We now have three examples of data structures where
  • *individual operations may be slow*, but
  • *any series of operations is fast*.
• Giving weak upper bounds on the cost of each operation is not useful for making predictions.
• How can we clearly communicate when a situation like this one exists?
**Key Idea:** Backcharge expensive operations to cheaper ones.
Key Idea: Backcharge expensive operations to cheaper ones.
Key Idea: Backcharge expensive operations to cheaper ones.
**Key Idea:** Backcharge expensive operations to cheaper ones.
**Key Idea:** Backcharge expensive operations to cheaper ones.
**Key Idea:** Backcharge expensive operations to cheaper ones.
**Key Idea:** Backcharge expensive operations to cheaper ones.
**Key Idea:** Backcharge expensive operations to cheaper ones.
**Key Idea:** Backcharge expensive operations to cheaper ones.
**Key Idea:** Backcharge expensive operations to cheaper ones.
Key Idea: Backcharge expensive operations to cheaper ones.
work

<table>
<thead>
<tr>
<th>time</th>
<th>work</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
These are the *real* costs of the operations. Most operations are fast, but we can’t get a nice upper bound on any one operation cost.
These are the *amortized* costs of the operations. Each operation appears fast, and all costs are nicely bounded from above.
Amortized Analysis

• **Key Idea:** Assign each operation a (fake!) cost called its *amortized cost* such that, *for any series of operations performed*, the following is true:

\[ \sum \text{amortized-cost} \geq \sum \text{real-cost} \]

• Amortized costs shift work backwards from expensive operations onto cheaper ones.
  • Cheap operations are artificially made more expensive to pay for future cleanup work.
  • Expensive operations are artificially made cheaper by shifting the work backwards.
Where We’re Going

- The *amortized* cost of an enqueue or dequeue into a two-stack queue is $O(1)$.
- Any sequence of $n$ operations on a two-stack queue will take time

$$n \cdot O(1) = O(n).$$

- However, each individual operation may take more than $O(1)$ time to complete.
Where We’re Going

- The *amortized* cost of appending to a dynamic array is $O(1)$.
- Any sequence of $n$ appends to a dynamic array will take time $n \cdot O(1) = O(n)$.
- However, each individual operation may take more than $O(1)$ time to complete.
Where We’re Going

- The *amortized* cost of inserting a new element at the end of a B-tree, assuming we have a pointer to the rightmost leaf, is $O(1)$.
- Any sequence of $n$ appends will take time $n \cdot O(1) = O(n)$.
- However, each individual operation may take more than $O(1)$ time to complete.
Formalizing This Idea
Assigning Amortized Costs

- The approach we’ve taken so far for assigning amortized costs is called an *aggregate analysis*.
  - Directly compute the maximum possible work done across any sequence of operations, then divide that by the number of operations.
- This approach works well here, but it doesn’t scale well to more complex data structures.
  - What if different operations contribute to / clean up messes in different ways?
  - What if it’s not clear what sequence is the worst-case sequence of operations?
- In practice, we tend to use a different strategy called the *potential method* to assign amortized costs.
Potential Functions

- To assign amortized costs, we’ll need to measure how “messy” the data structure is.
- For each data structure, we define a potential function $\Phi$ that, in a sense, “quantifies messiness.”
  - $\Phi$ is small when the data structure is “clean,” and
  - $\Phi$ is large when the data structure is “messy.”
Potential Functions

• To assign amortized costs, we’ll need to measure how “messy” the data structure is.

• For each data structure, we define a potential function $\Phi$ that, in a sense, “quantifies messiness.”
  
  • $\Phi$ is small when the data structure is “clean,” and
  
  • $\Phi$ is large when the data structure is “messy.”

Low $\Phi$ Dynamic Array

H He Li Be

High $\Phi$ Dynamic Array

H He Li Be B C N O
Potential Functions

- Once we have $\Phi$, we can start looking, for each operation, at how $\Phi$ changes.
  - If an operation makes things “messier,” then $\Phi$ increases.
  - If an operation makes things “cleaner,” then $\Phi$ decreases.
- What we want to have happen:
  - If an operation increases $\Phi$, we artificially raise its cost.
  - If an operation decreases $\Phi$, we artificially lower its cost.
- **Why?**

Answer at [https://pollev.com/cs166spr23](https://pollev.com/cs166spr23)
Potential Functions

- Define the amortized cost of an operation to be
  \[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
  where \( k \) is a constant under our control and \( \Delta \Phi \) is the difference between \( \Phi \) just after the operation finishes and \( \Phi \) just before the operation started:
  \[ \Delta \Phi = \Phi_{\text{after}} - \Phi_{\text{before}} \]

- Intuitively:
  - If \( \Phi \) increases, the data structure got “messier,” and the amortized cost is \textit{higher} than the real cost to account for future cleanup costs.
  - If \( \Phi \) decreases, the data structure got “cleaner,” and the amortized cost is \textit{lower} than the real cost
Why This Works

\[ \sum \text{amortized-cost} = \sum (\text{real-cost} + k \cdot \Delta \Phi) \]
Why This Works

\[ \sum \text{amortized-cost} = \sum (\text{real-cost} + k \cdot \Delta \Phi) \]
\[ = \sum \text{real-cost} + k \cdot \sum \Delta \Phi \]
Why This Works

$$\sum \text{amortized-cost} = \sum ( \text{real-cost} + k \cdot \Delta \Phi )$$

$$= \sum \text{real-cost} + k \cdot \sum \Delta \Phi$$
Why This Works

\[
\sum \text{amortized-cost} = \sum (\text{real-cost} + k \cdot \Delta \Phi)
\]

\[
= \sum \text{real-cost} + k \cdot \sum \Delta \Phi
\]

Think “fundamental theorem of calculus,” but for discrete derivatives!

\[
\int_a^b f'(x) \, dx = f(b) - f(a)
\]

\[
\sum_{x=a}^b \Delta f(x) = f(b+1) - f(a)
\]

Look up finite calculus if you’re curious to learn more!
Why This Works

\[ \sum \text{amortized-cost} = \sum (\text{real-cost} + k \cdot \Delta \Phi) \]

\[ = \sum \text{real-cost} + k \cdot \sum \Delta \Phi \]

\[ = \sum \text{real-cost} + k \cdot (\Phi_{\text{end}} - \Phi_{\text{start}}) \]

Think “fundamental theorem of calculus,”
but for discrete derivatives!

\[
\int_{a}^{b} f'(x) \, dx = f(b) - f(a) \quad \sum_{x=a}^{b} \Delta f(x) = f(b+1) - f(a)
\]

Look up finite calculus if you’re curious to learn more!
Why This Works

$$\sum_{\text{amortized-cost}} = \sum \left( \text{real-cost} + k \cdot \Delta \Phi \right)$$

$$= \sum \text{real-cost} + k \cdot \sum \Delta \Phi$$

$$= \sum \text{real-cost} + k \cdot (\Phi_{\text{end}} - \Phi_{\text{start}})$$
Why This Works

\[ \sum \text{amortized-cost} \quad = \quad \sum (\text{real-cost} + k \cdot \Delta \Phi) \]

\[ = \quad \sum \text{real-cost} + k \cdot \sum \Delta \Phi \]

\[ = \quad \sum \text{real-cost} + k \cdot (\Phi_{\text{end}} - \Phi_{\text{start}}) \]
Why This Works

\[
\sum \text{amortized-cost} = \sum (\text{real-cost} + k \cdot \Delta \Phi)
\]

\[
= \sum \text{real-cost} + k \cdot \sum \Delta \Phi
\]

\[
= \sum \text{real-cost} + k \cdot (\Phi_{\text{end}} - \Phi_{\text{start}})
\]

Let’s make two assumptions:

\[
\Phi \geq 0. \\
\Phi_{\text{start}} = 0.
\]
Why This Works

\[ \sum_{\text{amortized-cost}} = \sum (\text{real-cost} + k \cdot \Delta \Phi) \]

\[ = \sum \text{real-cost} + k \cdot \sum \Delta \Phi \]

\[ = \sum \text{real-cost} + k \cdot (\Phi_{\text{end}} - \Phi_{\text{start}}) \]

\[ \geq \sum \text{real-cost} \]

Let’s make two assumptions:

\[ \Phi \geq 0. \]

\[ \Phi_{\text{start}} = 0. \]
Why This Works

\[\sum \text{amortized\-cost} = \sum (\text{real\-cost} + k \cdot \Delta \Phi)\]

\[= \sum \text{real\-cost} + k \cdot \sum \Delta \Phi\]

\[= \sum \text{real\-cost} + k \cdot (\Phi_{\text{end}} - \Phi_{\text{start}})\]

\[\geq \sum \text{real\-cost}\]

Assigning costs this way will never, in any circumstance, overestimate the total amount of work done.
The Story So Far

• We will assign amortized costs to each operation such that

\[ \sum \text{amortized-cost} \geq \sum \text{real-cost} \]

• To do so, define a potential function \( \Phi \) such that
  • \( \Phi \) measures how “messy” the data structure is,
  • \( \Phi_{start} = 0 \), and
  • \( \Phi \geq 0 \).

• Then, define amortized costs of operations as

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]

for a choice of \( k \) under our control.
Two-Stack Queues

Dynamic Arrays

Building B-Trees
The Two-Stack Queue
The Two-Stack Queue

\[ \Phi = \text{height of In stack} \]
The Two-Stack Queue

\[ \phi = \text{height of } In \text{ stack} \]
The Two-Stack Queue

$\Phi = \text{height of } \textbf{In} \text{ stack}$

$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi$
The Two-Stack Queue

$\Phi = \text{height of } In \text{ stack}$

$$amortized-cost = real-cost + k \cdot \Delta \Phi$$
$$= O(1) + k \cdot 1$$
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
\[ = O(1) + k \cdot 1 \]
\[ = O(1) \]
The Two-Stack Queue

\[ \Phi = \text{height of } In \text{ stack} \]
The Two-Stack Queue

\( \Phi = \text{height of } \text{In stack} \)

Amortized-cost = real-cost + \( k \cdot \Delta \Phi \)
= \( O(1) + k \cdot 1 \)
= \( O(1) \)
The Two-Stack Queue

Φ = height of \textit{In} stack
The Two-Stack Queue

$\Phi = \text{height of } \mathit{In} \text{ stack}$

$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi$

$= \mathcal{O}(1) + k \cdot 1$

$= \mathcal{O}(1)$
The Two-Stack Queue

\[ \Phi = \text{height of } \text{In stack} \]
The Two-Stack Queue

Φ = height of \textbf{In} stack

\[
\text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \\
= O(1) + k \cdot 1 \\
= O(1)
\]
The Two-Stack Queue

\[ \Phi = \text{height of } \text{In stack} \]
The Two-Stack Queue

\[ \Phi = \text{height of In stack} \]

\[ \begin{align*}
\text{Out} & : 4 \\
\text{In} & : 3 \\
& : 2 \\
& : 1
\end{align*} \]
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]
The Two-Stack Queue

\[ \Phi = \text{height of } In \text{ stack} \]
The Two-Stack Queue

$\Phi = \text{height of } \textbf{In} \text{ stack}$
The Two-Stack Queue

$\Phi = \text{height of } \text{In stack}$
The Two-Stack Queue

\[ \Phi = \text{height of } \textit{In} \text{ stack} \]
The Two-Stack Queue

\[ \Phi = \text{height of } \text{In stack} \]
The Two-Stack Queue

$\Phi = \text{height of } \textbf{In} \text{ stack}$
The Two-Stack Queue

Φ = height of $In$ stack
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]

1

2

3

4

Out

In
The Two-Stack Queue

$\Phi = \text{height of } \text{In stack}$

```
1
2
3
4
```

Out

```

```

In
The Two-Stack Queue

$\Phi =$ height of In stack
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]
The Two-Stack Queue

$\Phi = \text{height of } \textit{In} \text{ stack}$

$\textit{amortized-cost} = \textit{real-cost} + k \cdot \Delta \Phi$
The Two-Stack Queue

$\Phi = \text{height of } \textbf{In} \text{ stack}$

$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi$

$= O(h) + k \cdot -h \quad // h = \text{height of } \textbf{In} \text{ stack}$
The Two-Stack Queue

\[ \Phi = \text{height of In stack} \]

\[
\text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \\
= O(h) + k \cdot -h \quad \text{// } h = \text{height of In stack} \\
= O(1) \quad \text{// Choose } k \text{ strategically}
\]
The Two-Stack Queue

\[ \Phi = \text{height of } \text{In stack} \]
The Two-Stack Queue

Φ = height of \textbf{In} stack

\begin{itemize}
  \item 2
  \item 3
  \item 4
\end{itemize}

\textbf{Out}

\textbf{In}
The Two-Stack Queue

\[ \Phi = \text{height of } \text{In stack} \]
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
The Two-Stack Queue

Φ = height of \textbf{In} stack

\begin{align*}
amortized\text{-}cost & = real\text{-}cost + k \cdot \Delta \Phi \\
& = O(1) + k \cdot 0
\end{align*}
The Two-Stack Queue

Φ = height of In stack

 amortized-cost = real-cost + k \cdot \Delta \Phi
= O(1) + k \cdot 0
= O(1)
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]
The Two-Stack Queue

Φ = height of \textit{In} stack

amortized-cost = real-cost + k \cdot \Delta \Phi
= O(1) + k \cdot 1
= O(1)
The Two-Stack Queue

$\Phi = \text{height of In stack}$

Out

In

3

4

6

5
The Two-Stack Queue

$\Phi = \text{height of } \textit{In} \text{ stack}$

 amortized-cost $= \text{real-cost} + k \cdot \Delta \Phi$
$= O(1) + k \cdot 1$
$= O(1)$
The Two-Stack Queue

$\Phi = \text{height of } \textbf{In} \text{ stack}$
The Two-Stack Queue

$\Phi = \text{height of } \textit{In} \text{ stack}$
The Two-Stack Queue

$\Phi = \text{height of } \textbf{In} \text{ stack}$

$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi$

$= O(1) + k \cdot 0$

$= O(1)$
The Two-Stack Queue

\[ \Phi = \text{height of } \textit{In} \text{ stack} \]
The Two-Stack Queue

$\Phi = \text{height of } In \text{ stack}$
The Two-Stack Queue

Φ = height of *In* stack

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
\[ = O(1) + k \cdot 1 \]
\[ = O(1) \]
The Two-Stack Queue

$\Phi = \text{height of } \textit{In} \text{ stack}$

Out

4

In

7

6

5
The Two-Stack Queue

Φ = height of *In* stack
The Two-Stack Queue

Φ = height of \textbf{In} stack
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]

The amortized cost is:

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]

\[ = O(1) + k \cdot 0 \]

\[ = O(1) \]
The Two-Stack Queue

Φ = height of \textbf{In} stack
The Two-Stack Queue

$\Phi = \text{height of } \text{In stack}$
The Two-Stack Queue

Φ = height of \textit{In} stack
The Two-Stack Queue

Φ = height of *In* stack
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]
The Two-Stack Queue

\[ \Phi = \text{height of In stack} \]
The Two-Stack Queue

Φ = height of \textit{In} stack

\begin{itemize}
\item \textbf{Out}:
  \begin{itemize}
  \item 7
  \item 6
\end{itemize}
\item \textbf{In}:
  \begin{itemize}
  \item 5
\end{itemize}
\end{itemize}
The Two-Stack Queue

$\Phi = \text{height of } \textbf{In} \text{ stack}$
The Two-Stack Queue

Φ = height of \textit{In} stack

\begin{itemize}
  \item 5
  \item 6
  \item 7
\end{itemize}
The Two-Stack Queue

Φ = height of In stack
The Two-Stack Queue

$\Phi = \text{height of } \text{In stack}$
The Two-Stack Queue

\[ \Phi = \text{height of } \textbf{In} \text{ stack} \]

**Amortized Cost**

\[
\text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \\
= O(h) + k \cdot -h \quad // \quad h = \text{height of } \textbf{In} \text{ stack} \\
= O(1) \quad // \quad \text{Choose } k \text{ strategically}
\]
**Theorem:** The amortized cost of any enqueue or dequeue operation on a two-stack queue is $O(1)$.

**Proof:** Let $\Phi$ be the height of the *In* stack in the two-stack queue. Each enqueue operation does a single push and increases the height of the *In* stack by one. Therefore, its amortized cost is

$$O(1) + k \cdot \Delta \Phi = O(1) + k \cdot 1 = O(1).$$

Now, consider a dequeue operation. If the *Out* stack is nonempty, then the dequeue does $O(1)$ work and does not change $\Phi$. Its cost is therefore

$$O(1) + k \cdot \Delta \Phi = O(1) + k \cdot 0 = O(1).$$

Otherwise, the *Out* stack is empty. Suppose the *In* stack has height $h$. The dequeue does $O(h)$ work to pop the elements from the *In* stack and push them onto the *Out* stack, followed by one additional pop for the dequeue. This is $O(h)$ total work.

At the beginning of this operation, we have $\Phi = h$. At the end of this operation, we have $\Phi = 0$. Therefore, $\Delta \Phi = -h$, so the amortized cost of the operation is

$$O(h) + k \cdot -h = O(1),$$

assuming we pick $k$ to cancel out the constant factor hidden in the $O(h)$ term. ■
Building B-Trees

Two-Stack Queues

Dynamic Arrays
Analyzing Dynamic Arrays

- **Goal:** Choose a potential function $\Phi$ such that the amortized cost of an append is $O(1)$.

- **Initial (wrong!) guess:** Set $\Phi$ to be the number of free slots left in the array.
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]
Dynamic Arrays

\( \Phi = \text{number of free slots} \)
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
\[ = O(1) + k \cdot -1 \]
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]

Amortized cost:

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]

\[ = O(1) + k \cdot -1 \]

\[ = O(1) \]
Dynamic Arrays

$\Phi = \text{number of free slots}$
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]

Amortized Cost:

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]

\[ = O(1) + k \cdot -1 \]

\[ = O(1) \]
Dynamic Arrays

$$\Phi = \text{number of free slots}$$
Dynamic Arrays

Φ = number of free slots

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
\[ = O(1) + k \cdot -1 \]
\[ = O(1) \]
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]

Amortized Cost:

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]

\[ = O(1) + k \cdot -1 \]

\[ = O(1) \]
Dynamic Arrays

$\Phi =$ number of free slots
Dynamic Arrays

$\Phi = \text{number of free slots}$
Dynamic Arrays

Φ = number of free slots
Dynamic Arrays

$\Phi = \text{number of free slots}$

With this choice of $\Phi$, what is the amortized cost of an append to an array of size $n$ when no free slots are left?

Answer at https://pollev.com/cs166spr23
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
Dynamic Arrays

\[ \Phi = \text{number of free slots} \]

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
\[ = O(n) + k \cdot \Theta(n) \]
Dynamic Arrays

$$\Phi = \text{number of free slots}$$

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi$$
$$= O(n) + k \cdot \Theta(n)$$
$$= O(n)$$
Analyzing Dynamic Arrays

- **Intuition:** $\Phi$ should measure how “messy” the data structure is.
  - Having lots of free slots means there’s very little mess.
  - Having few free slots means there’s a lot of mess.
- We basically got our potential function backwards. Oops.
- **Question:** What should $\Phi$ be?
Analyzing Dynamic Arrays

- The amortized cost of an append is
  \[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi. \]
- When we double the array size, our real cost is \( \Theta(n) \). We need \( \Delta \Phi \) to be something like \(-n\).
- **Goal**: Pick \( \Phi \) so that
  - when there are no slots left, \( \Phi \approx n \), and
  - right after we double the array size, \( \Phi \approx 0 \).
- With some trial and error, we can come up with
  \[ \Phi = \#\text{elems} - \#\text{free-slots} \]
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]

Amortized cost:

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
Dynamic Arrays

$$\Phi = \#\text{elems} - \#\text{free-slots}$$

**amortized-cost** = **real-cost** + $$k \cdot \Delta \Phi$$

= $$O(1) + k \cdot 2$$
Dynamic Arrays

$$\Phi = \#\text{elems} - \#\text{free-slots}$$

amortized-cost = real-cost + $k \cdot \Delta \Phi$

= $O(1) + k \cdot 2$

= $O(1)$
Dynamic Arrays

\[ \Phi = \text{#elems} - \text{#free-slots} \]
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]

amortized-cost = real-cost + \( k \cdot \Delta \Phi \)
= \( \mathcal{O}(1) + k \cdot 2 \)
= \( \mathcal{O}(1) \)
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]
Dynamic Arrays

\[ \Phi = \text{#elems} - \text{#free-slots} \]

amortized-cost = real-cost + \( k \cdot \Delta \Phi \)
= \( O(1) + k \cdot 2 \)
= \( O(1) \)
Dynamic Arrays

\[ \Phi = \text{\#elems} - \text{\#free-slots} \]
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]

amortized-cost = real-cost + \( k \cdot \Delta \Phi \)
= \( O(1) + k \cdot 2 \)
= \( O(1) \)
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]
\( \Phi = \#\text{elems} - \#\text{free-slots} \)
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi \]
Dynamic Arrays

\[ \Phi = \# \text{elems} - \# \text{free-slots} \]

amortized-cost = real-cost + \( k \cdot \Delta \Phi \)
= \( O(n) + k \cdot -\Theta(n) \)
Dynamic Arrays

\[ \Phi = \#\text{elems} - \#\text{free-slots} \]

\[ \text{amortized-cost} = \text{real-cost} + k \cdot \Delta\Phi \]
\[ = O(n) + k \cdot -\Theta(n) \]
\[ = O(1) \quad \text{// Pick } k \text{ well} \]
A Caveat

• We require that $\Phi_{\text{start}} = 0$ and that $\Phi \geq 0$.
• What happens when we have a newly-created dynamic array?

Quick fix: This is an edge case, so set
$$\Phi = \max\{0, \#\text{elems} - \#\text{free-slots}\}$$
Theorem: The amortized cost of an append to a dynamic array is O(1).

Proof: Suppose the dynamic array has initial capacity \(2C = O(1)\). Then, define \(\Phi = \max\{0, n - \#\text{free-slots}\}\), where \(n\) is the number of elements stored in the dynamic array. Note that for \(n < C\) that an append simply fills in a free slot and leaves \(\Phi = 0\), so the amortized cost of such an append is O(1). Otherwise, we have \(n > C\) and \(\Phi = n - \#\text{free-slots}\).

Consider any append. If the append does not trigger a resize, it does O(1) work, increases \(n\) by one, and decreases \(\#\text{free-slots}\) by one, so the amortized cost is

\[
O(1) + k \cdot \Delta \Phi = O(1) + k \cdot 2 = O(1).
\]

Otherwise, the operation copies \(n\) elements into a new array twice as large as before, increasing the number of free slots to \(n\), then fills one of those slots. Just before the operation we had \(\Phi = n\), and just after the operation we have \(\Phi = 2\). Therefore, the amortized cost is

\[
O(n) + k \cdot \Delta \Phi = O(n) + k \cdot (2 - n) = O(n) - nk + 2k,
\]

which can be made to equal O(1) by choosing the the \(k\) term to match the constant hidden in the O(\(n\)) term. ■
Some Exercises

- Suppose we grow the array not by a factor of two, but by a fixed constant \( \alpha > 1 \). Find a choice of \( \Phi \) so that the amortized cost of an append is \( O(1) \).

- Suppose we also allow elements to be removed from the array, and when it’s \( \frac{1}{4} \) full we shrink it by a factor of two. Find a choice of \( \Phi \) so the amortized cost of appending or removing the last element is \( O(1) \).
Two-Stack Queues

Dynamic Arrays

Building B-Trees
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

• **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

• **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

• **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- **Algorithm:** Store a pointer to the rightmost leaf. To add an item, append it to the rightmost leaf, splitting and kicking the median key up if we are out of space.
Building B-Trees

- What is the actual cost of appending an element?
  - Suppose that we perform splits at $L$ layers in the tree.
  - Each split takes time $\Theta(b)$ to copy and move keys around.
  - Total cost: $\Theta(bL)$.
- **Goal:** Pick a potential function $\Phi$ so that we can offset this cost and make each append cost amortized $O(1)$.
Building B-Trees

- Our potential function should, intuitively, quantify how “messy” our data structure is.

- Some observations:
  - We only care about nodes in the right spine of the tree.
  - Nodes in the right spine slowly have keys added to them. When they split, they lose (about) half of their keys.

- Idea: Set $\Phi$ to be the number of keys in the right spine of the tree.
Building B-Trees

- Let $\Phi$ be the number of keys on the right spine.
- Each split moves (roughly) half the keys from the split node into a node off the right spine.
Building B-Trees

- Let $\Phi$ be the number of keys on the right spine.
- Each split moves (roughly) half the keys from the split node into a node off the right spine.
Building B-Trees

- Let $\Phi$ be the number of keys on the right spine.
- Each split moves (roughly) half the keys from the split node into a node off the right spine.
Building B-Trees

- Let $\Phi$ be the number of keys on the right spine.
- Each split moves (roughly) half the keys from the split node into a node off the right spine.
Building B-Trees

- Let $\Phi$ be the number of keys on the right spine.
- Each split moves (roughly) half the keys from the split node into a node off the right spine.
Building B-Trees

- Let $\Phi$ be the number of keys on the right spine.
- Each split moves (roughly) half the keys from the split node into a node off the right spine.
Building B-Trees

- Let $\Phi$ be the number of keys on the right spine.
- Each split moves (roughly) half the keys from the split node into a node off the right spine.
- Change in potential per split: $-\Theta(b)$.
- Net $\Delta \Phi$: $-\Theta(bL)$. 
Building B-Trees

• Actual cost of an append that does $L$ splits: $O(bL)$.
• $\Delta \Phi$ for that operation: $-\Theta(bL)$.
• Amortized cost: $O(1)$.
**Theorem:** The amortized cost of appending to a B-tree by inserting it into the rightmost leaf node and applying fixup rules is $O(1)$.

**Proof:** Assume we are working with a B-tree of order $b$. Let $\Phi$ be the number of nodes on the right spine of the B-tree.

Suppose we insert a value into the tree using the algorithm described above. Suppose this causes $L$ nodes to be split. Each of those splits requires $\Theta(b)$ work for a net total of $\Theta(bL)$ work.

Each of those $L$ splits moves $\Theta(b)$ keys off of the right spine of the tree, decreasing $\Phi$ by $\Theta(b)$ for a net drop in potential of $-\Theta(bL)$. In the layer just above the last split, we add one more key into a node, increasing $\Phi$ by one. Therefore, $\Delta \Phi = -\Theta(bL)$.

Overall, this tells us that the amortized cost of inserting a key this way is

$$\Theta(bL) + k \cdot \Delta \Phi = \Theta(bL) - k \cdot \Theta(bL),$$

which can be made to be $O(1)$ by choosing $k$ to equate the constants hidden in the $O$ and $\Theta$ terms. ■
More to Explore

- You can implement a **deque** (a doubly-ended queue) using a B-tree with pointers to the first and last leaves.
  - This is sometimes called a **finger tree**.
  - Finger trees are used extensively in purely functional programming languages.
  - By extending the analysis from here, you can show the amortized cost of appending or removing from each end of the finger tree is $O(1)$.

- Red/black trees are modeled on 2-3-4 trees. You can build a red/black tree from $n$ sorted keys in time $O(n)$ this way.
  - **Great exercise:** Explore how to do this, and work out what choice of $\Phi$ to make.
To Summarize
Amortized Analysis

- Some data structures accumulate messes slowly, then clean up those messes in single, large steps.
- We can assign amortized costs to operations. These are fake costs such that summing up the amortized costs never underestimates the sum of the real costs.
- To do so, we define a potential function $\Phi$ that, intuitively, measures how “messy” the data structure is. We then set

$$\text{amortized-cost} = \text{real-cost} + k \cdot \Delta \Phi.$$  

- For simplicity, we assume that $\Phi$ is nonnegative and that $\Phi$ for an empty data structure is zero.
Next Time

- **Binomial Heaps**
  - A very clever way to build a priority queue.
- **Lazy Binomial Heaps**
  - Designing for amortization.