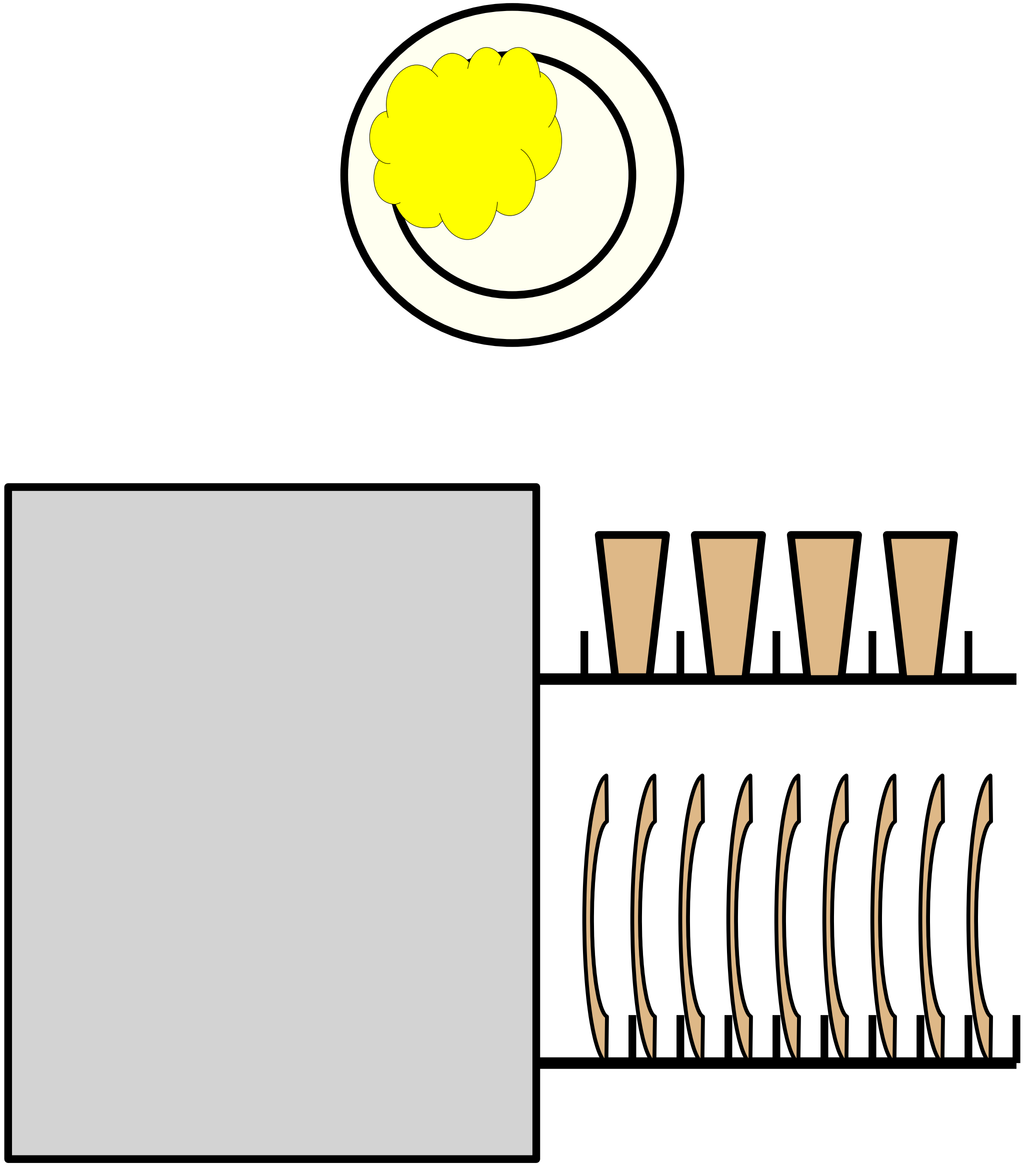
# Amortized Analysis

# A Motivating Analogy

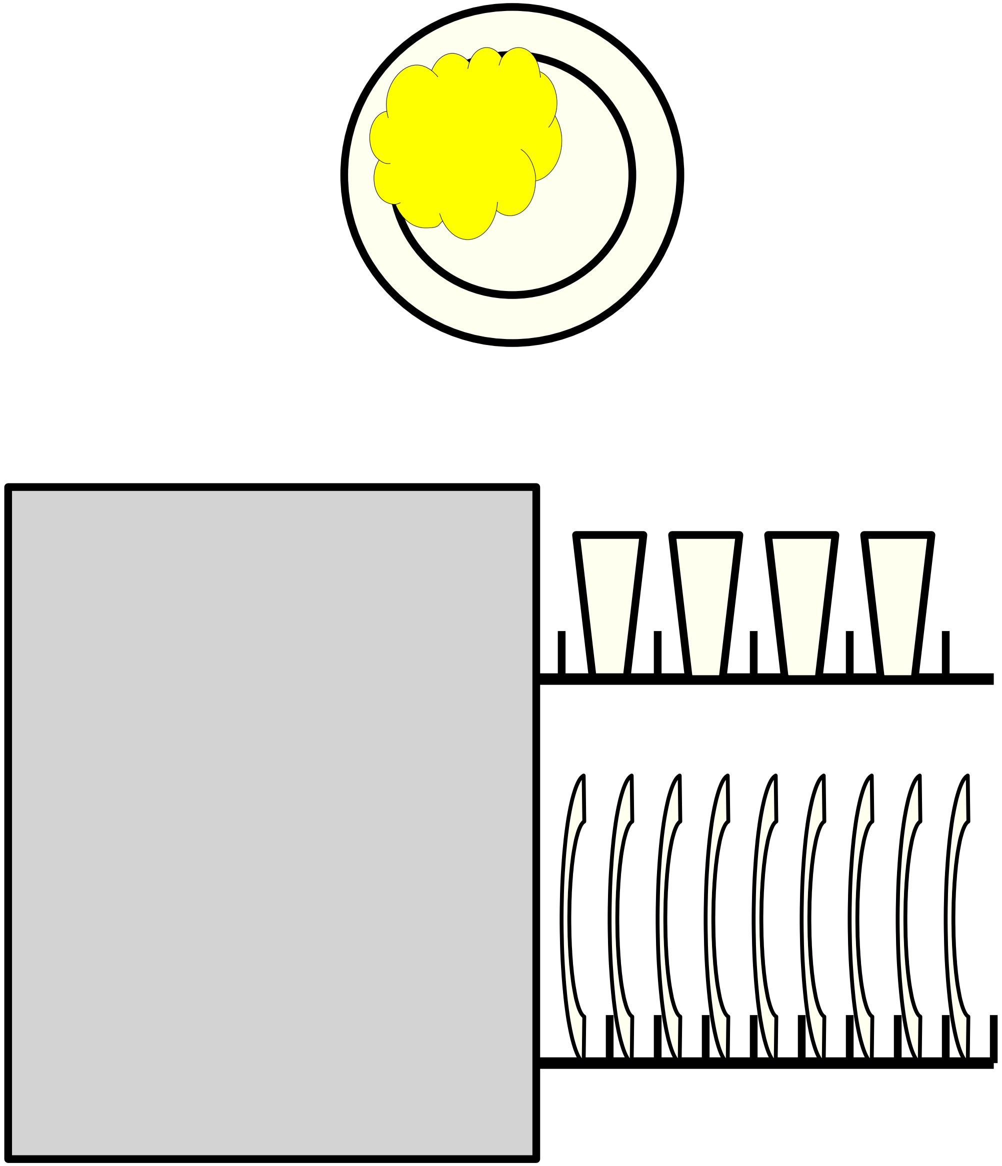
# Doing the Dishes

- 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

- 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*.

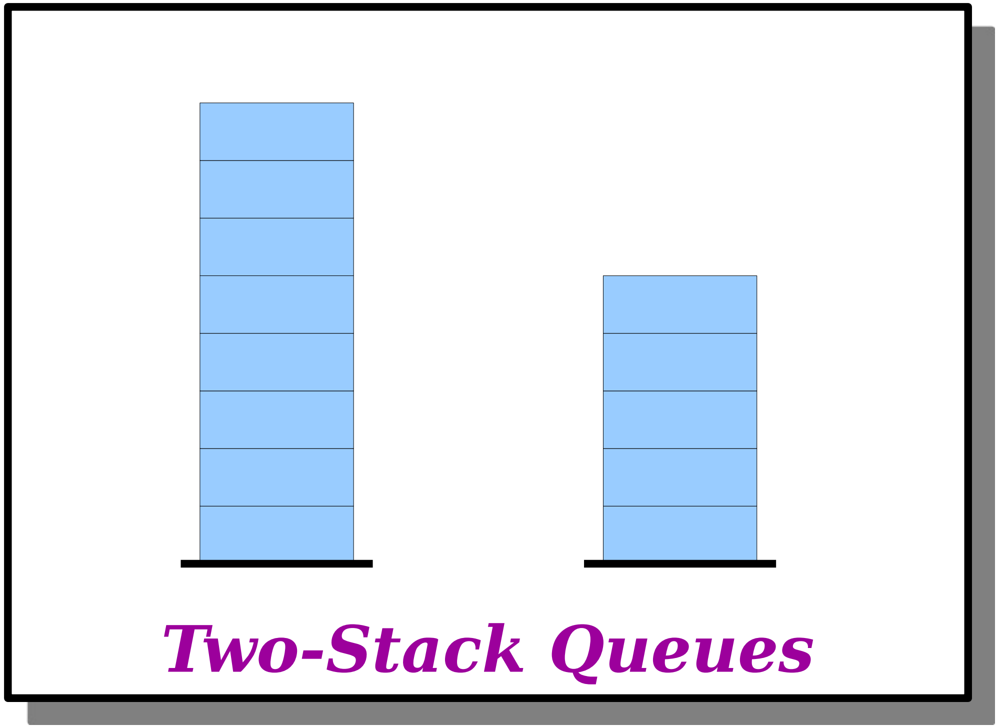
# Where We're Going

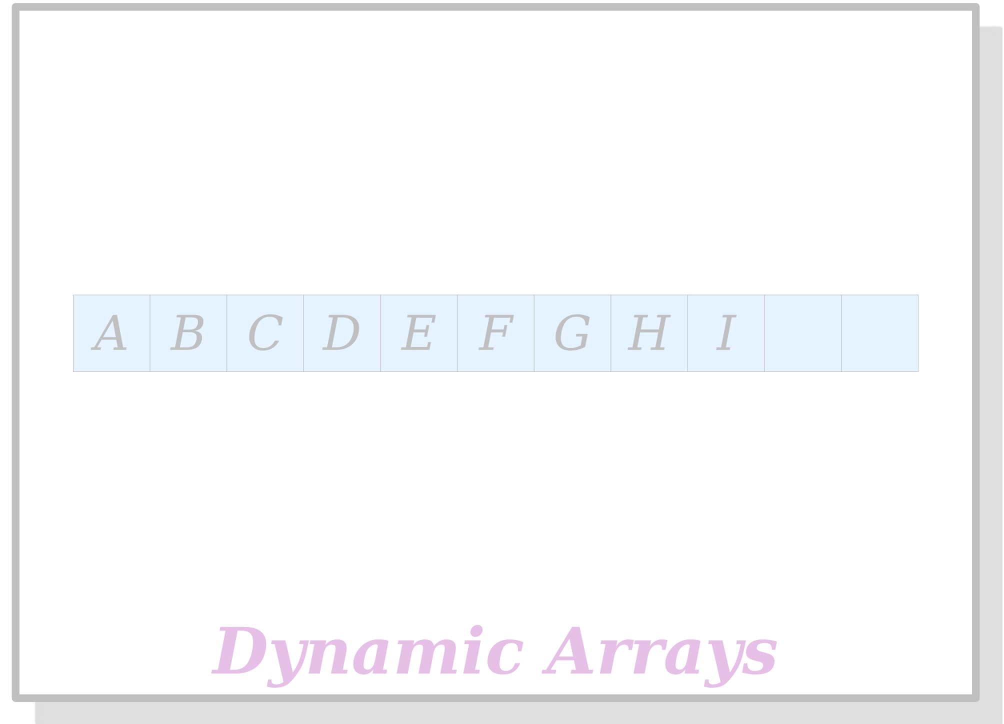
- ***Amortized Analysis (Today)***
  - A little accounting trickery never hurt anyone, right?
- ***Binomial Heaps (Thursday)***
  - A fast, flexible priority queue that's a great building block for more complicated structures.
- ***Fibonacci Heaps (Next Tuesday)***
  - A priority queue optimized for graph algorithms that, at least in theory, leads to optimal implementations.

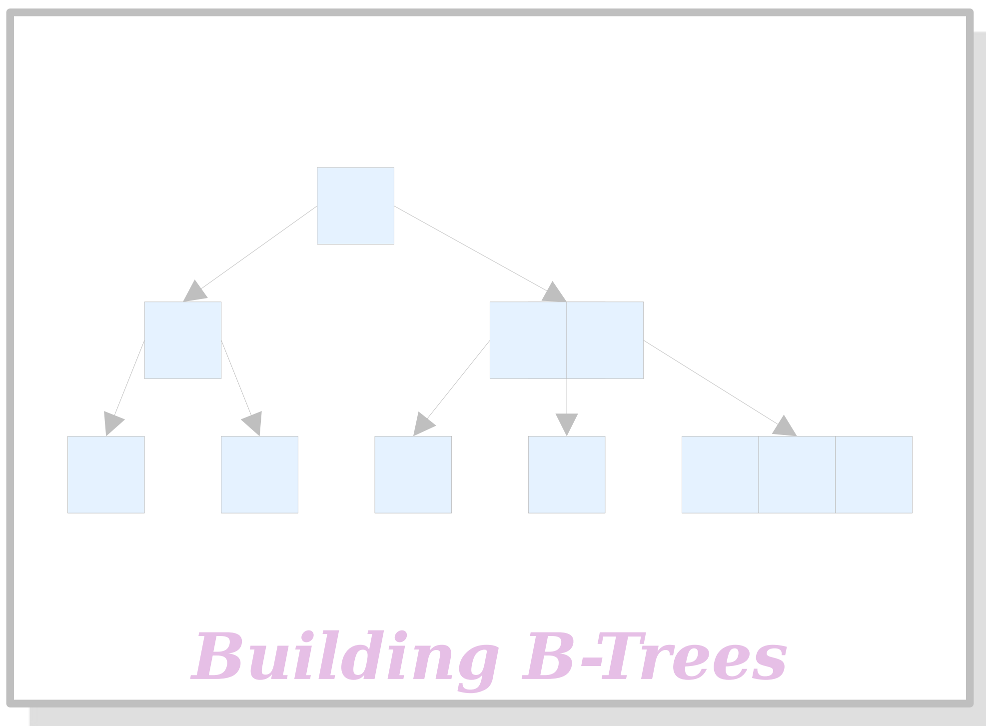
# Outline for Today

- ***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

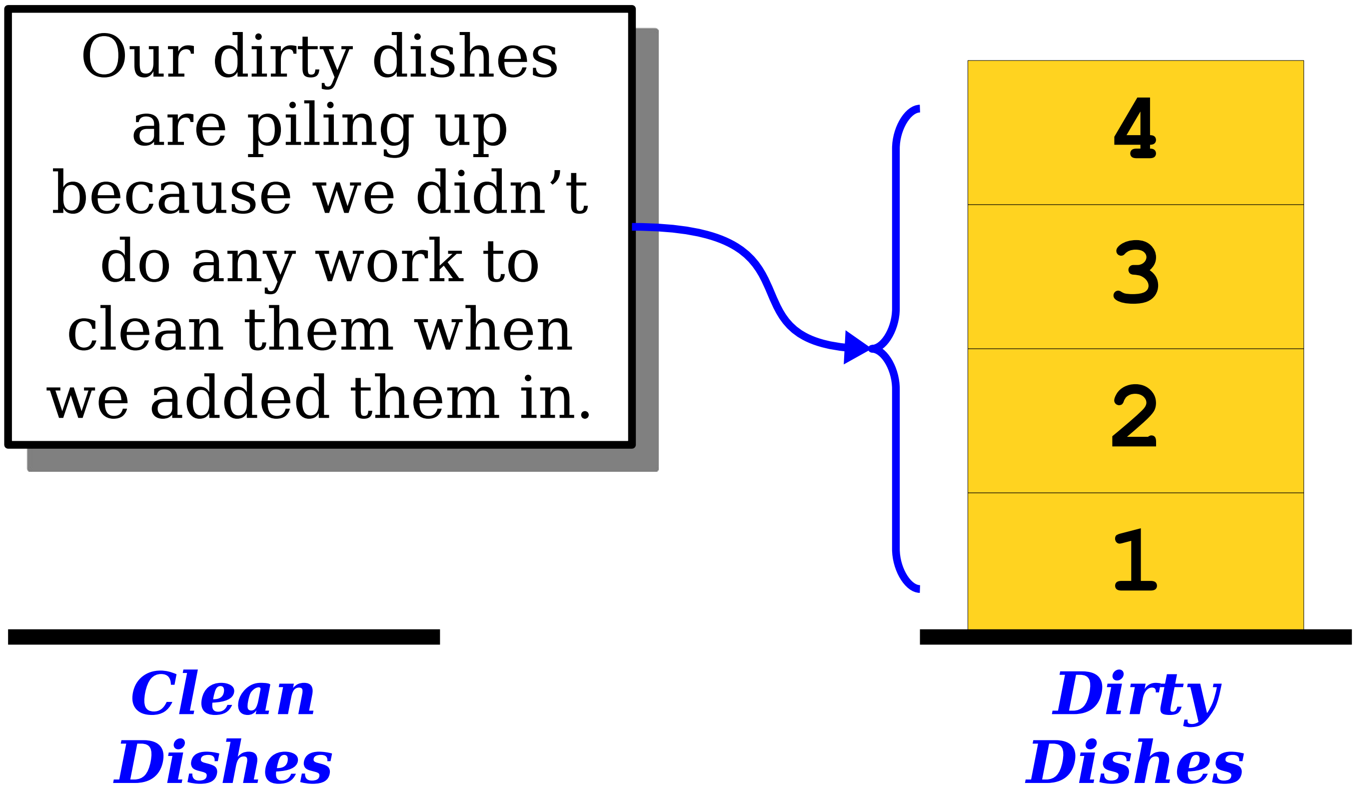

A B C D E F G H I
Dynamic Arrays


Building B-Trees

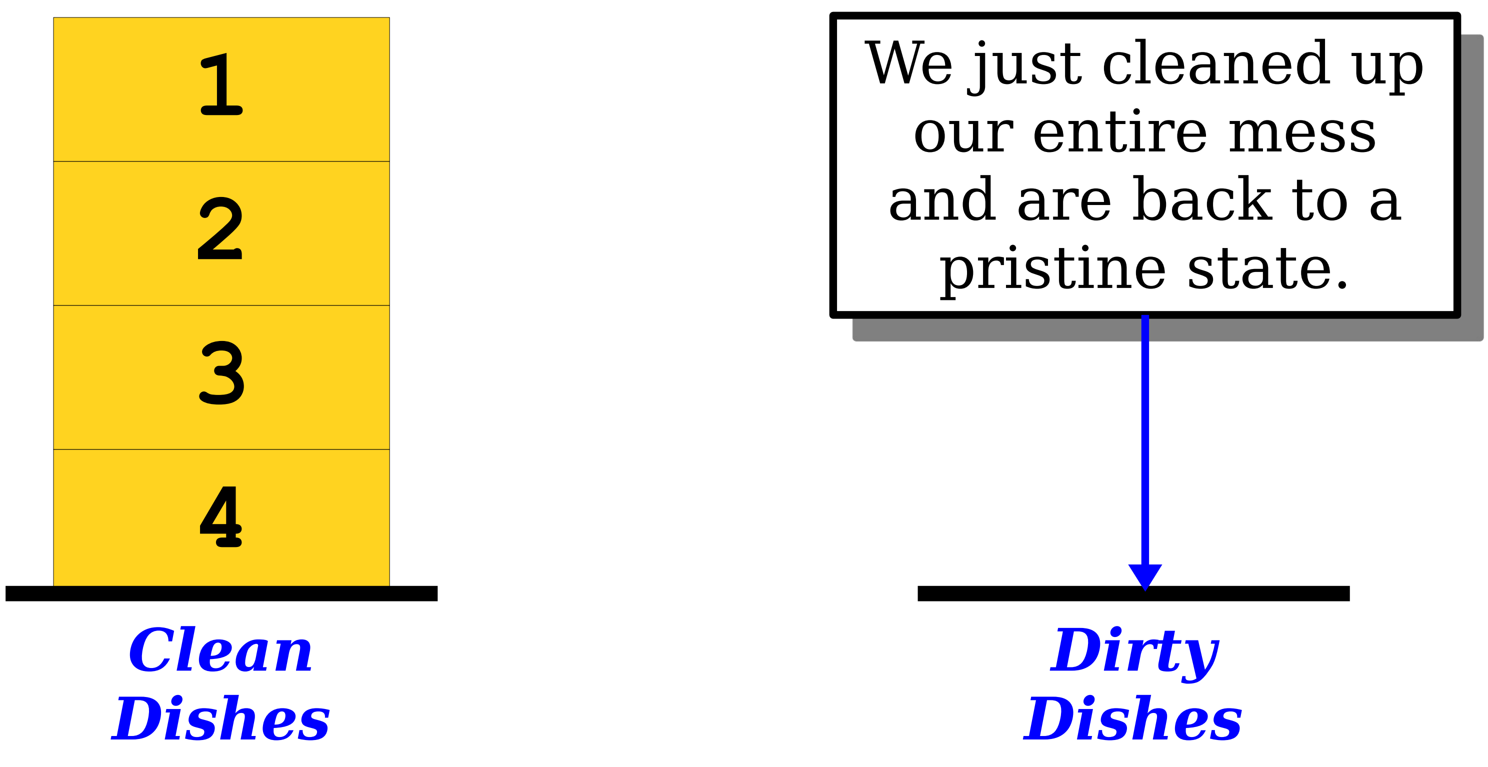
# The Two-Stack Queue

- 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.

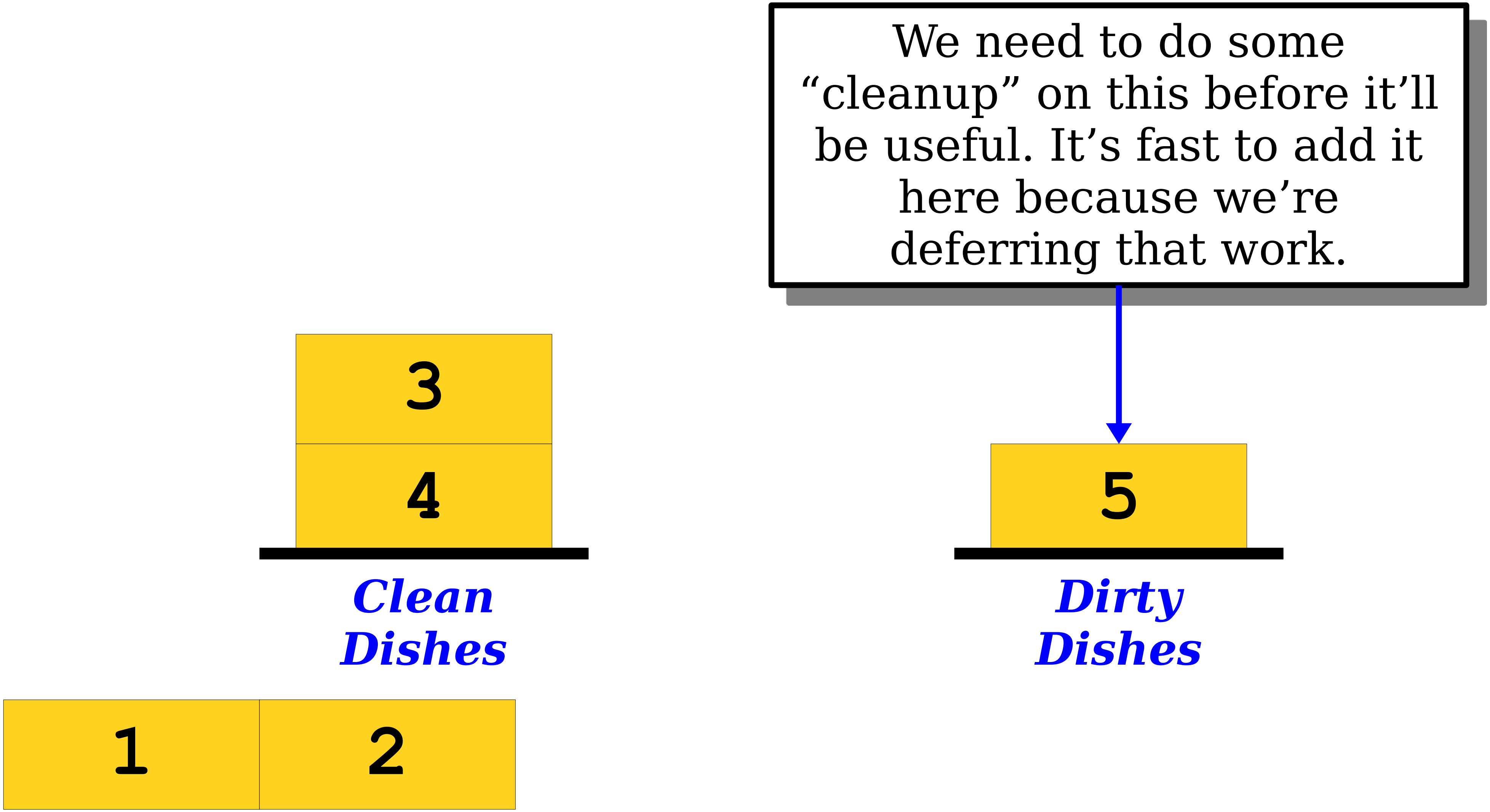
# The Two-Stack Queue



# The Two-Stack Queue



# The Two-Stack Queue



# 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 Θ($n$) if the ***Out*** stack is empty.

# The Two-Stack Queue

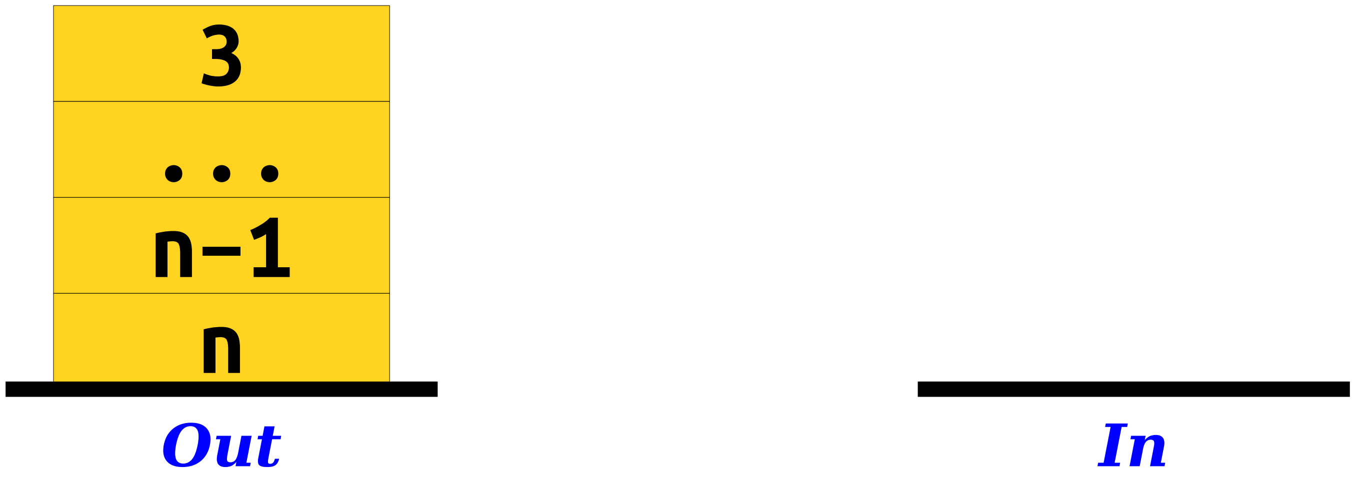
- ***Intuition:*** We only do expensive dequeues after a long run of cheap enqueues.
- Think "dishwasher:" we very slowly introduce a lot of dirty dishes that get cleaned up all at once.
- Provided we clean up all the dirty dishes at once, and provided that dirty dishes accumulate slowly, this is a fast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

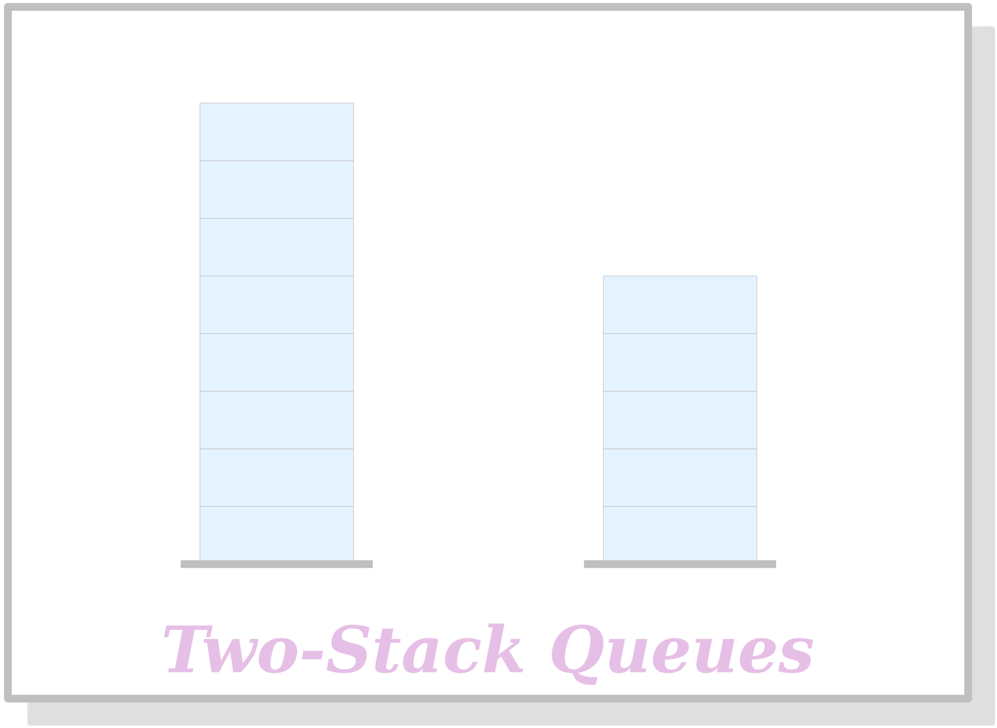
# The Two-Stack Queue

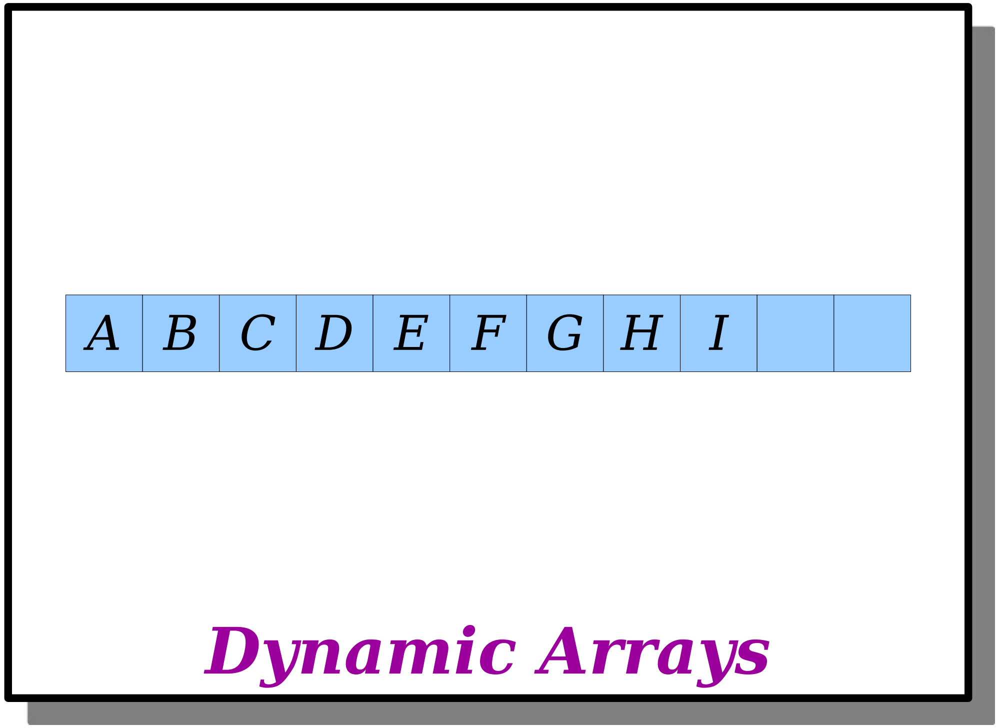
- ***Key Fact:*** Any series of $n$ operations on an (initially empty) two-stack queue will take time O($n$).
- ***Why?***
- Each item is pushed into at most two stacks and popped from at most two stacks.
- Adding up the work done per element across all $n$ operations, we can do at most O($n$) work.

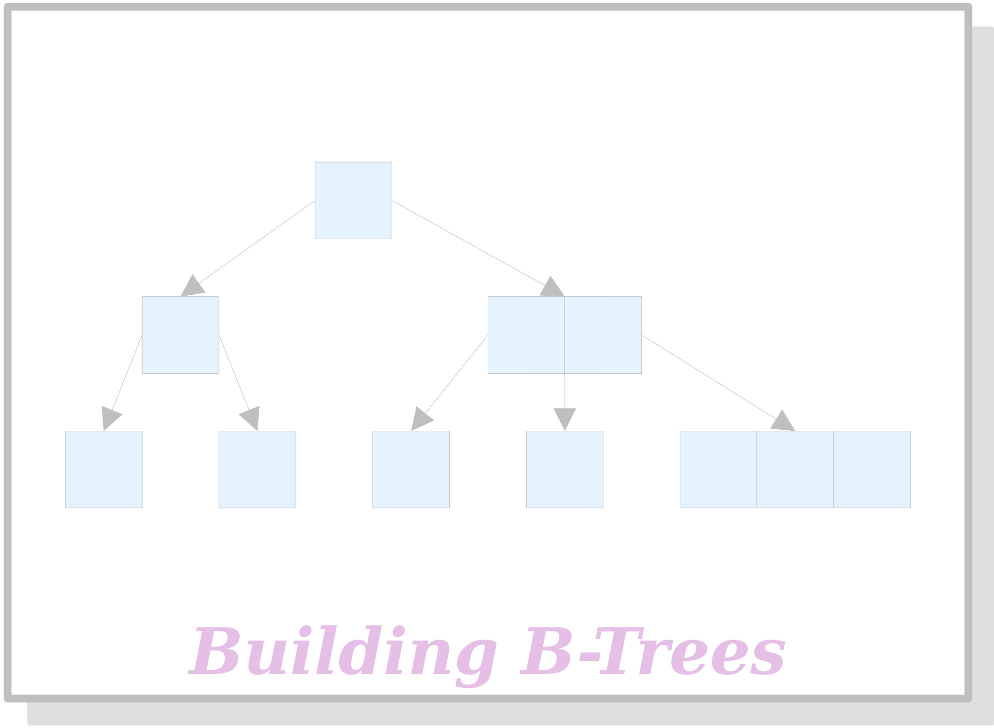
# The Two-Stack Queue

- 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?



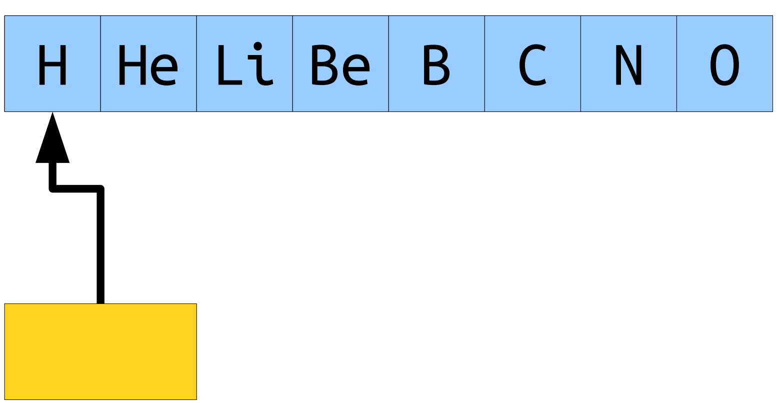

Two-Stack Queues


A B C D E F G H I
Dynamic Arrays


Building B-Trees

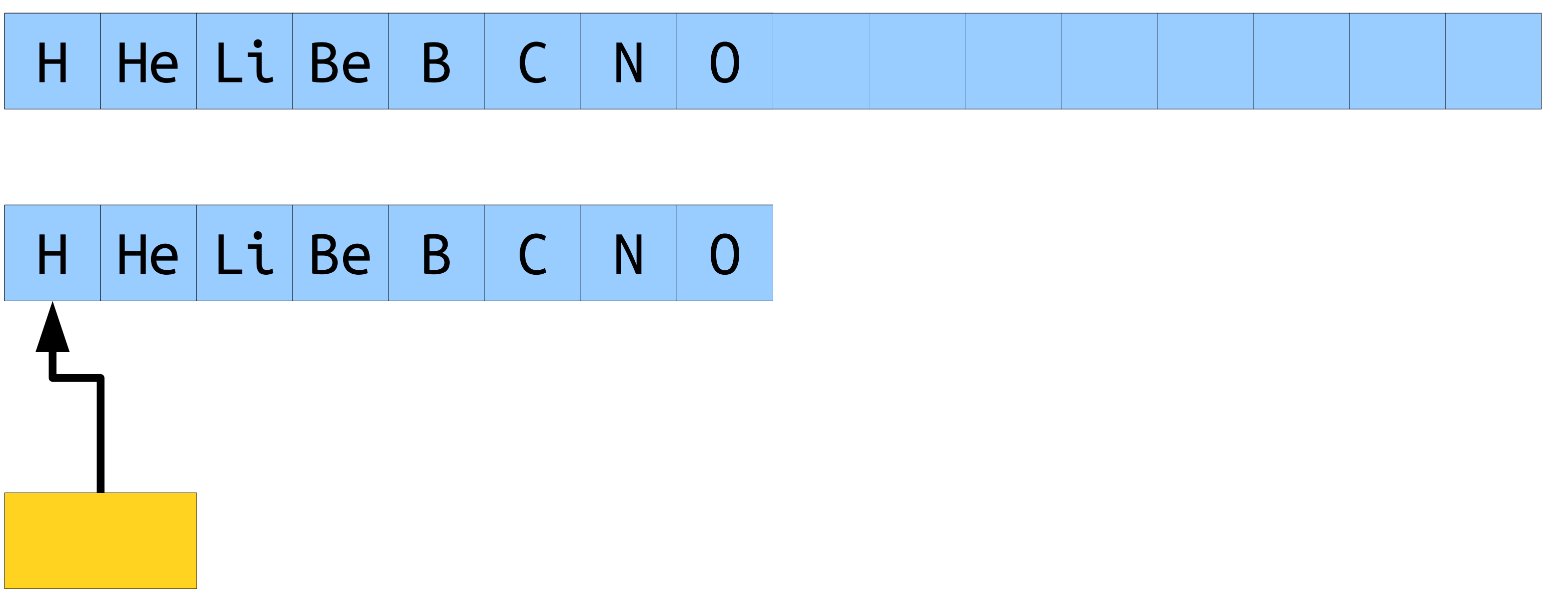
# Dynamic Arrays

- 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.



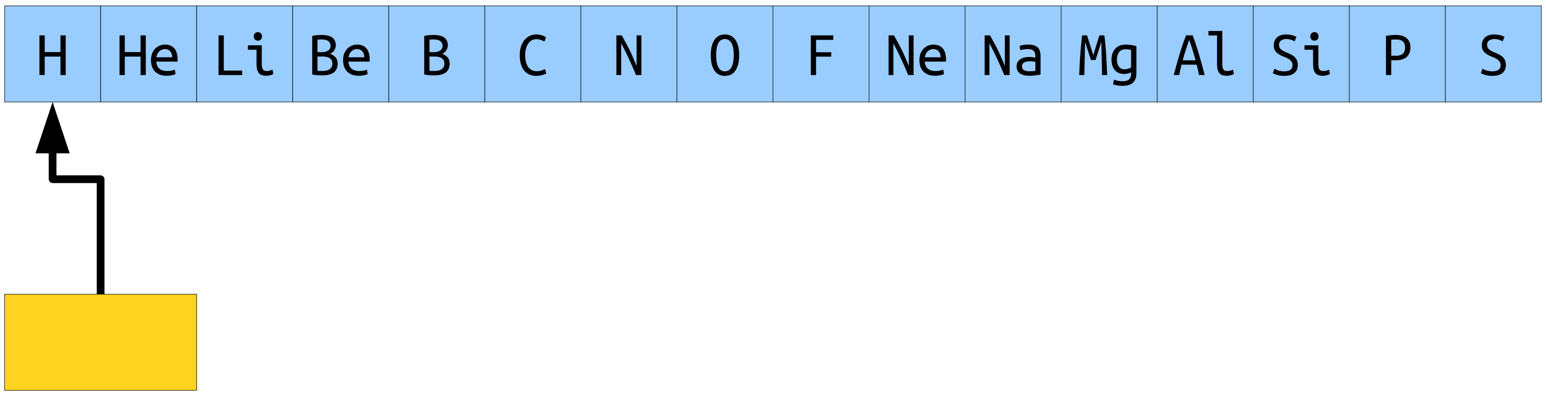
# Dynamic Arrays

- 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.



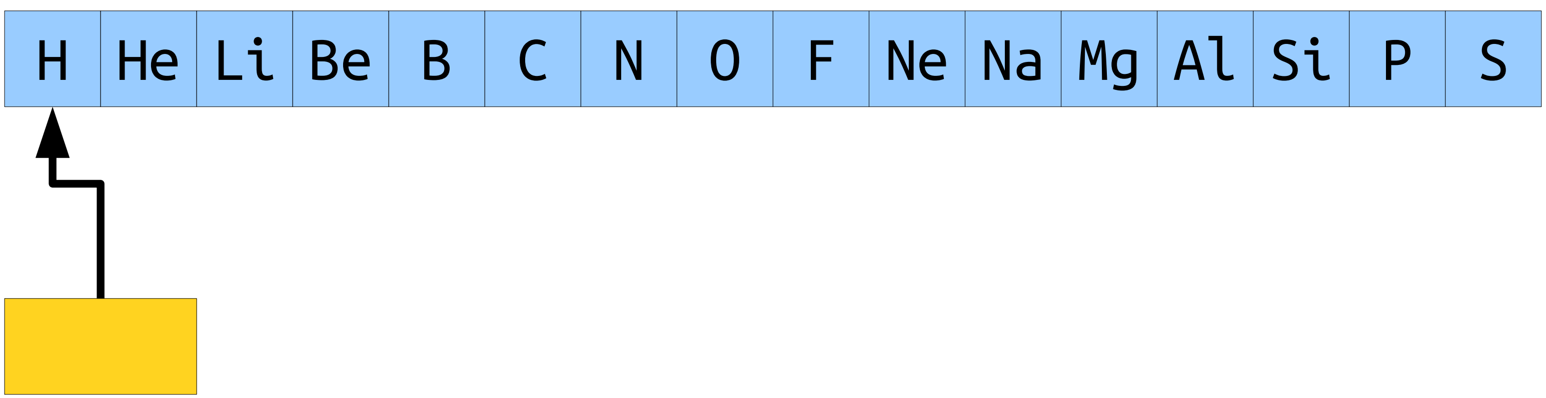
# Dynamic Arrays

- 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

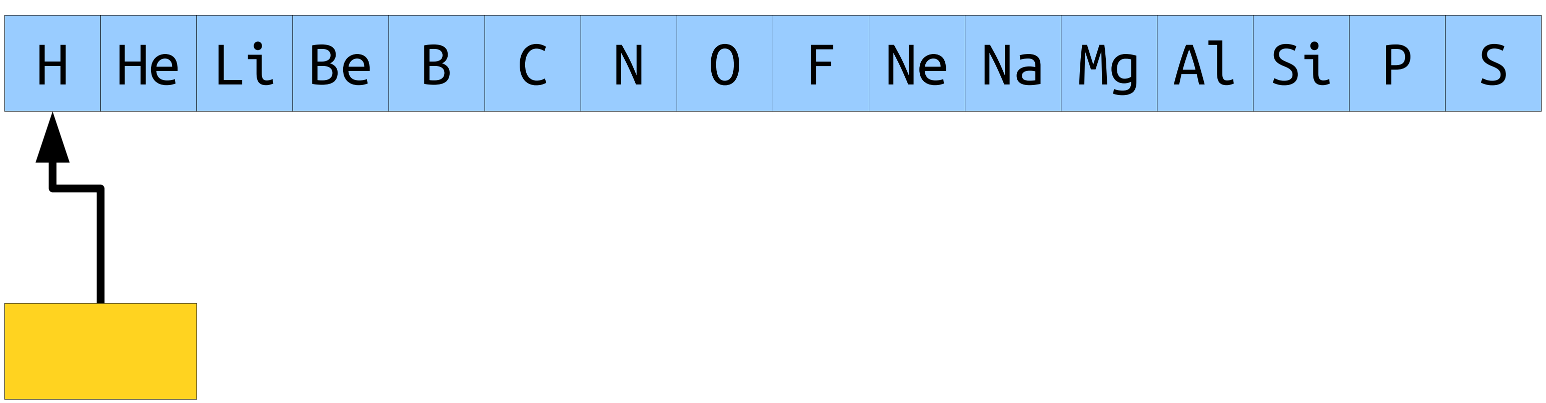
- Most appends to a dynamic array take time O(1).
- Infrequently, we do Θ(*n*) work to copy all *n* elements from the old array to a new one.
- Think “dishwasher:”
  - We slowly accumulate “messes” (filled slots).
  - We periodically do a large “cleanup” (copying the array).
- ***Claim:*** The cost of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



# Dynamic Arrays

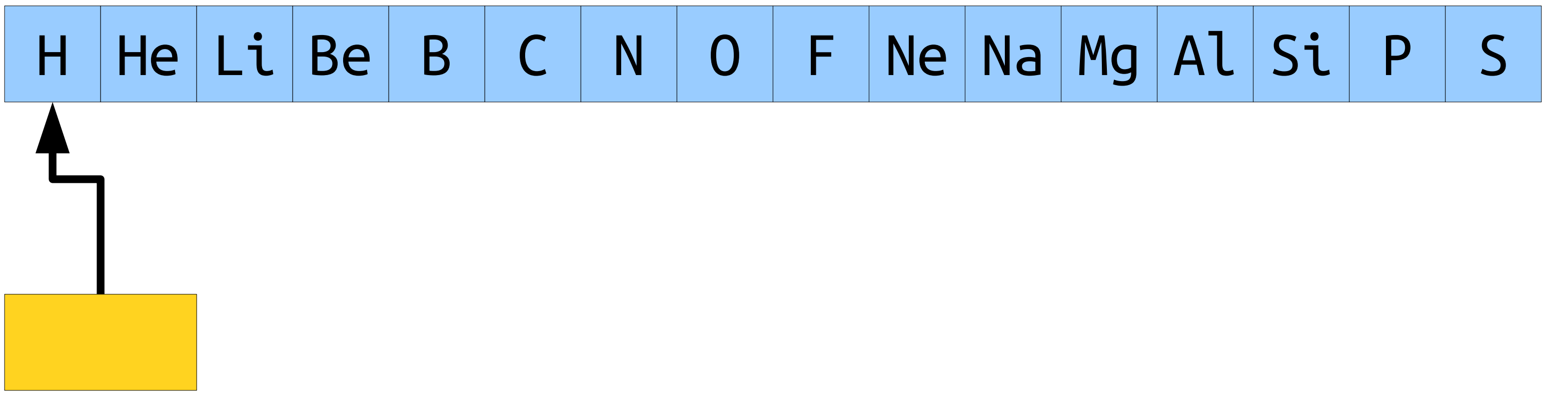
- ***Claim:*** Appending $n$ elements always takes time $O(n)$.
- The array doubles at sizes $2^0, 2^1, 2^2, \ldots,$ etc.
- The very last doubling is at the largest power of two less than $n$. This is at most $2^{\lfloor \log_2 n \rfloor}$. (Do you see why?)
- Total work done across all doubling is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

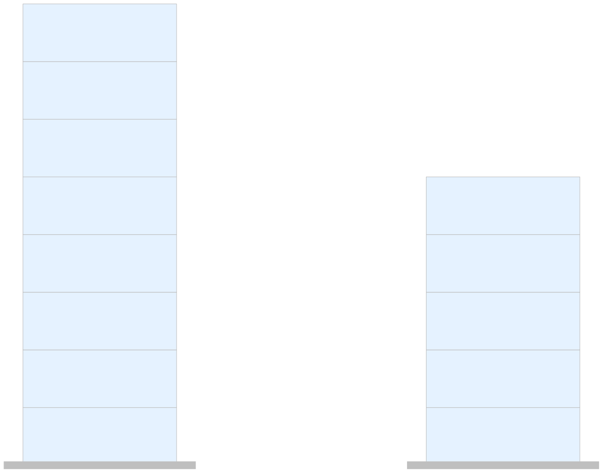
$$2^0 + 2^1 + \ldots + 2^{\lfloor \log_2 n \rfloor} = 2^{\lfloor \log_2 n \rfloor + 1} - 1$$
$$\leq 2^{\log_2 n + 1}$$
$$= 2n.$$



# Dynamic Arrays

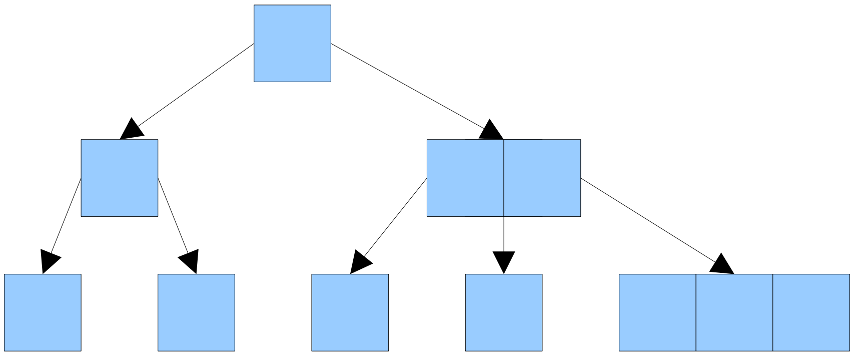
- It's correct but misleading to say the cost of an append is O($n$).
  - This is comparatively rare.
- It's wrong, but useful, to pretend that the cost of an append is O(1).
  - Some operations take more time than this.
  - However, pretending each operation takes O(1) time never underestimates the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





Two-Stack Queues

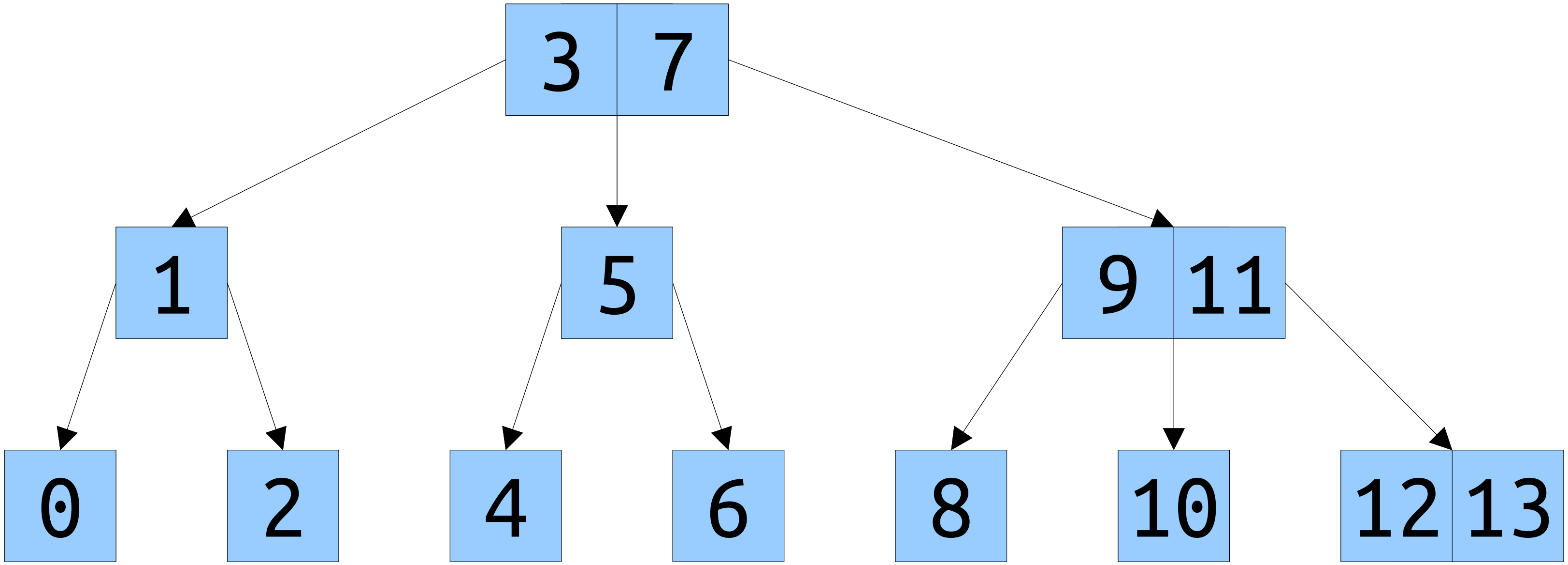

Dynamic Arrays



**Building B-Trees**

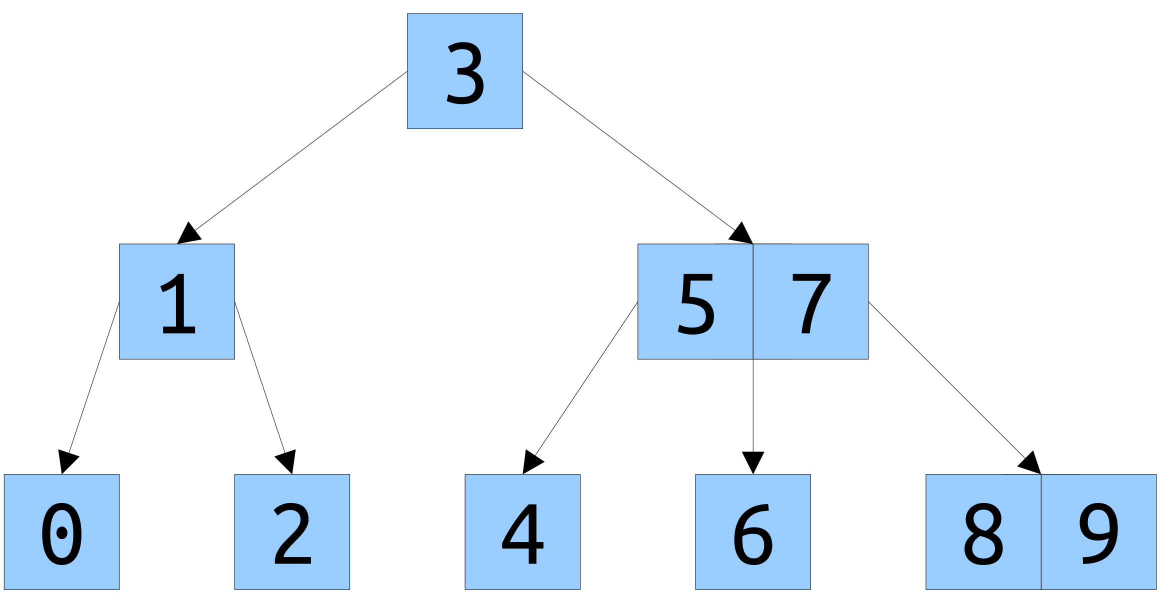
# Building B-Trees

- 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?



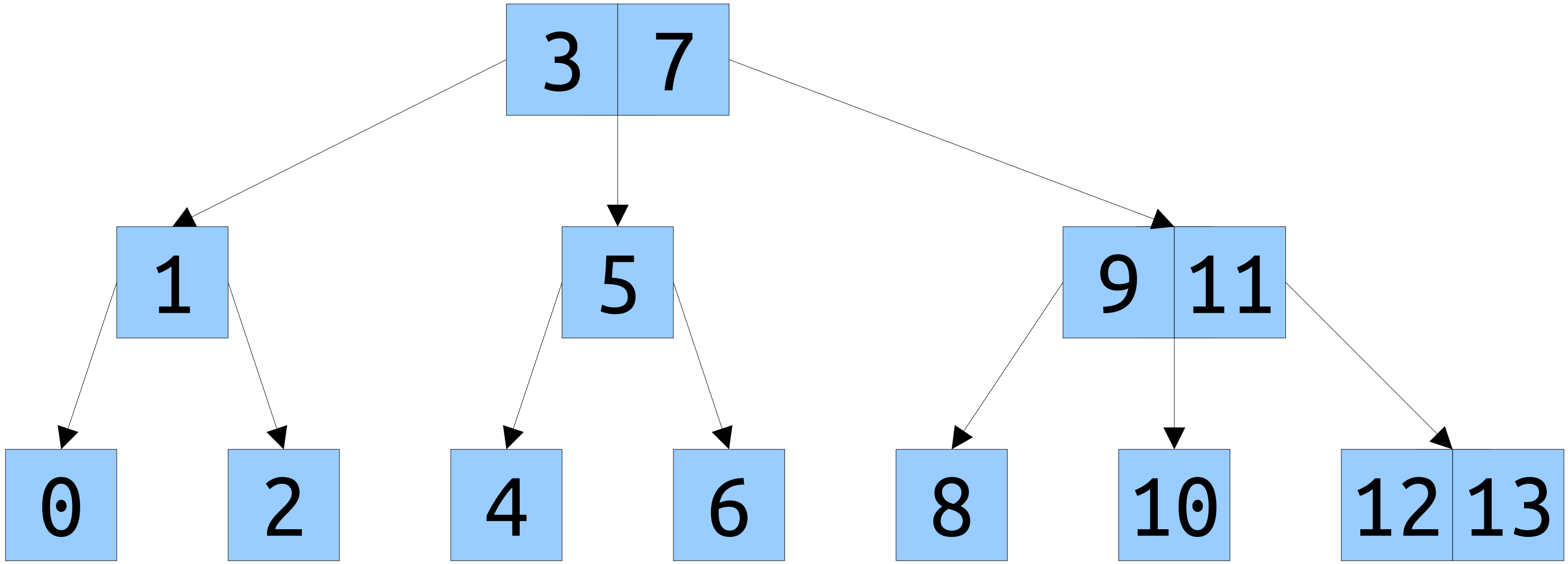
# Building B-Trees

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



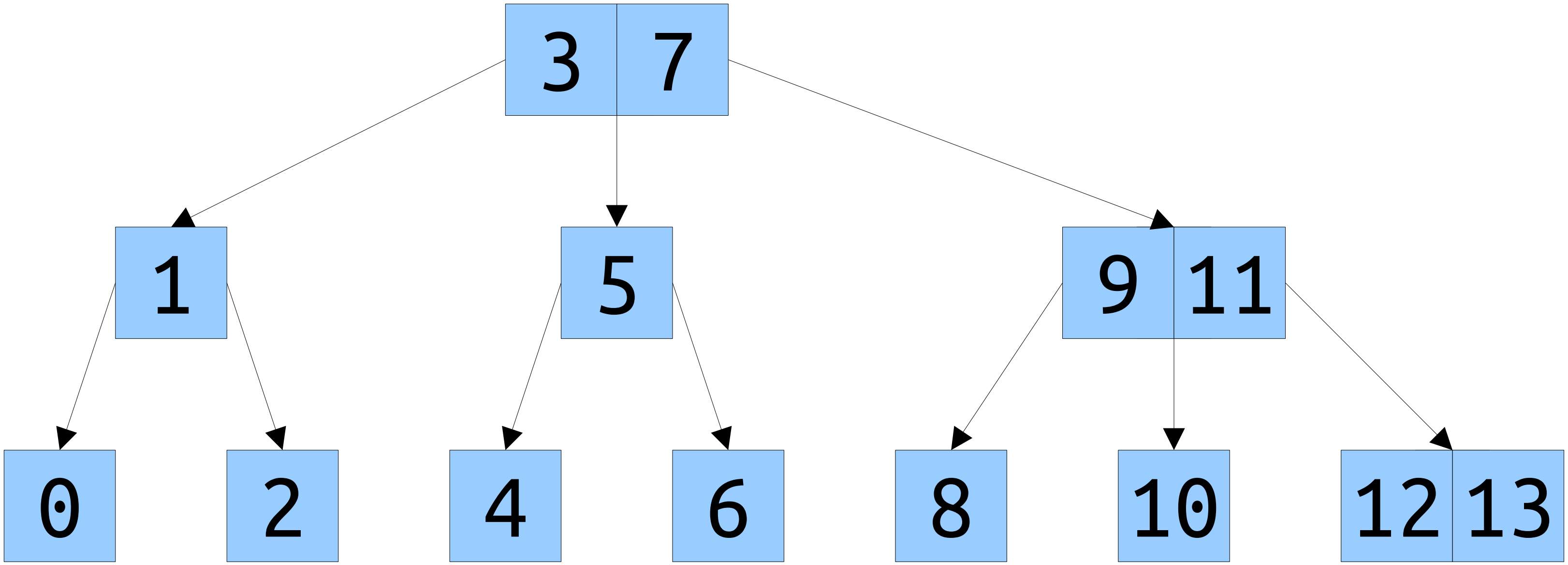
# Building B-Trees

- ***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.
- ***Question:*** How fast is this?



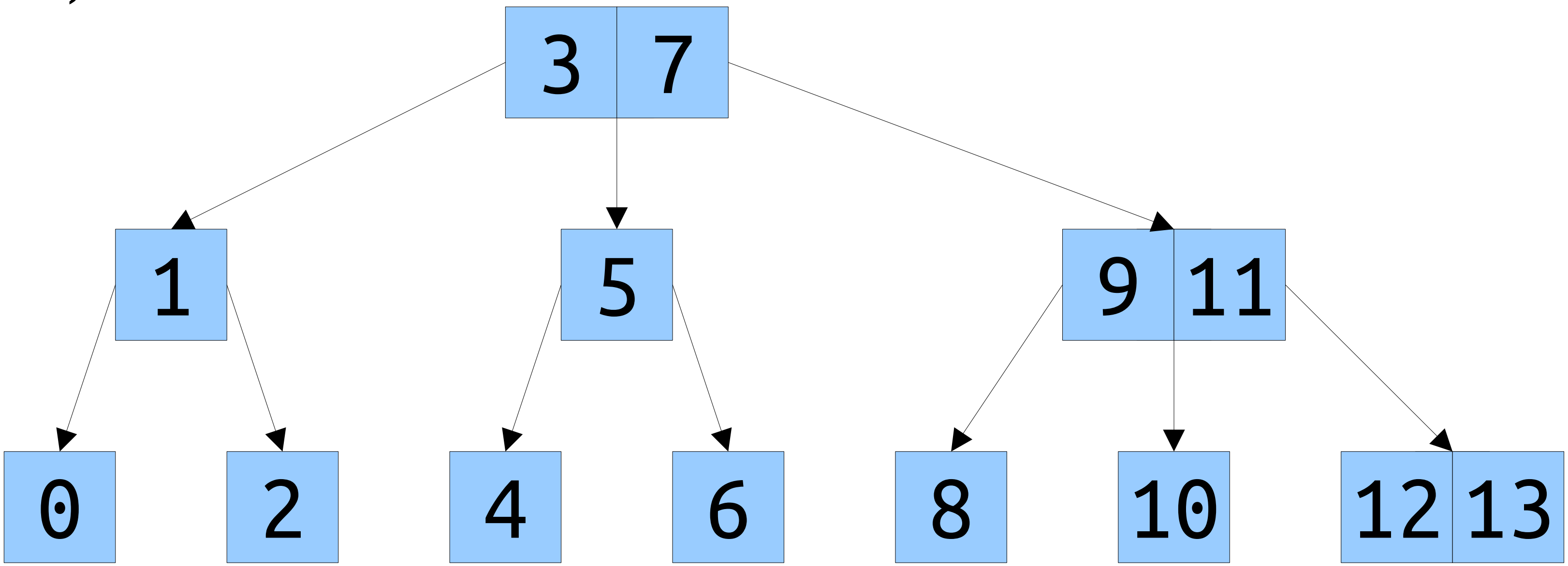
# Building B-Trees

- 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 $O(nb \log_b \text{n})$
- ***Claim:*** The cost of *n* inserts is always O(*n*).



# Building B-Trees

- Of all the $n$ insertions into the tree, a roughly $^1/_b$ fraction will split a node in the bottom layer of the tree (a leaf).
- Of those, roughly a $^1/_b$ fraction will split a node in the layer above that.
- Of those, roughly a $^1/_b$ fraction will split a node in the layer above that.
- (etc.)



# Building B-Trees

- Total number of splits:

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

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

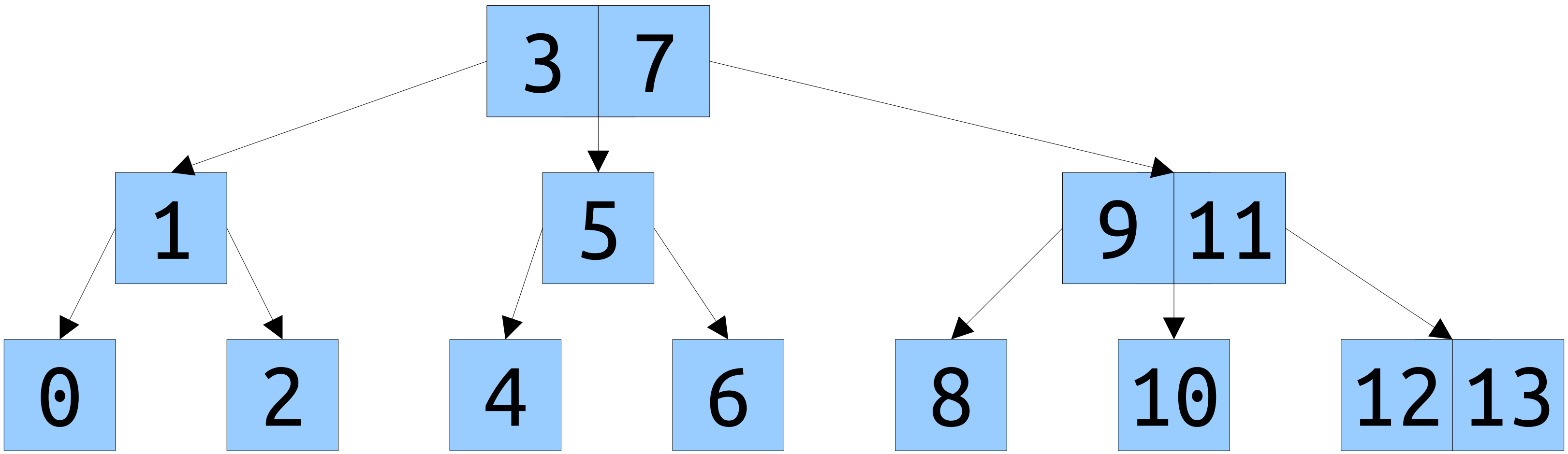
$$= \quad \frac{n}{b} \cdot \Theta(1)$$

$$= \quad \Theta\left(\frac{n}{b}\right)$$

- Total cost of those splits: **$\Theta(n)$**.

# Building B-Trees

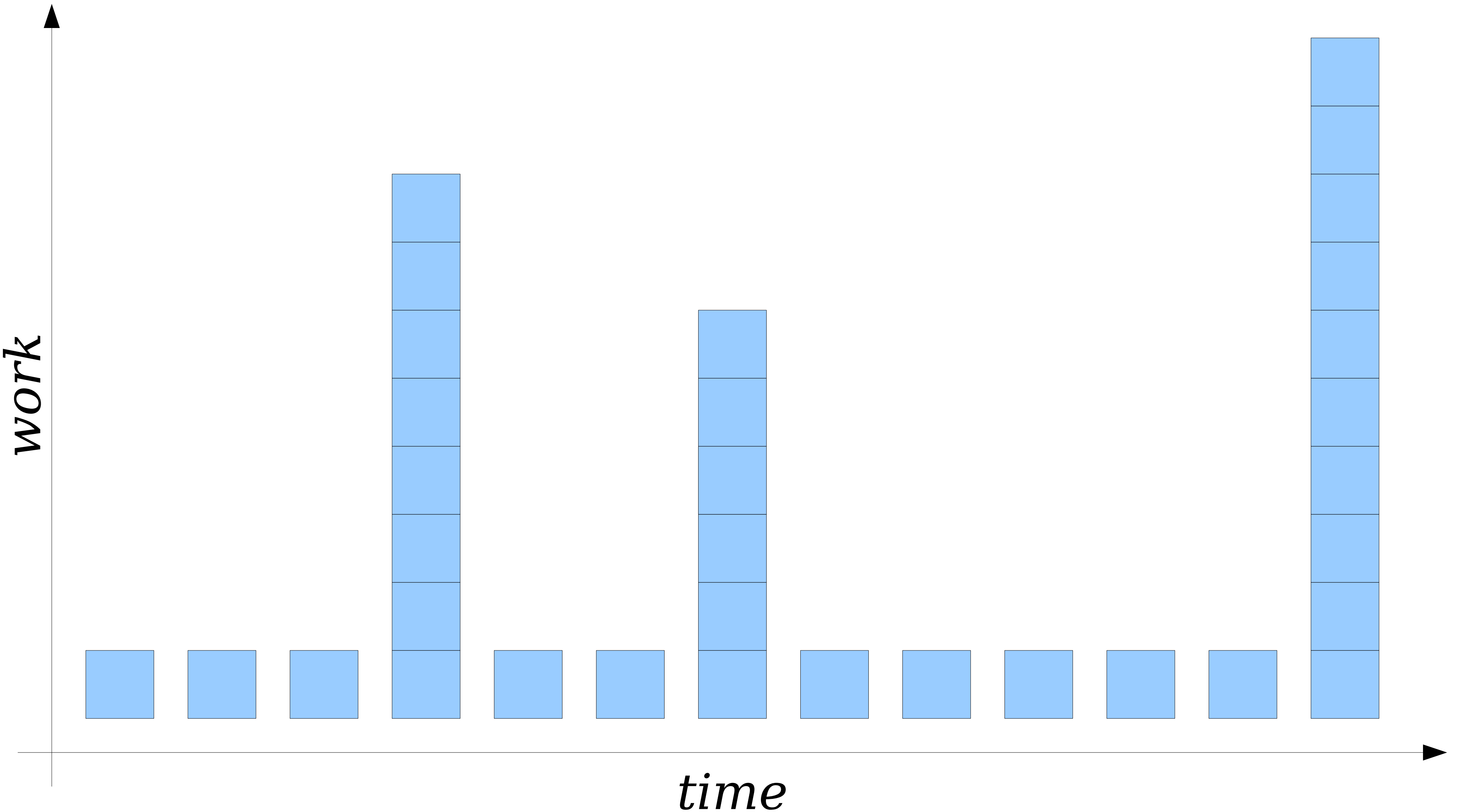
- It is correct but misleading to say the cost of an insert is $O(b \log_b n)$.
  - This is comparatively rare.
- It is wrong, but useful, to pretend that the cost of an insert is O(1).
  - Some operations take more time than this.
  - However, pretending each insert takes time O(1) never underestimates 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?



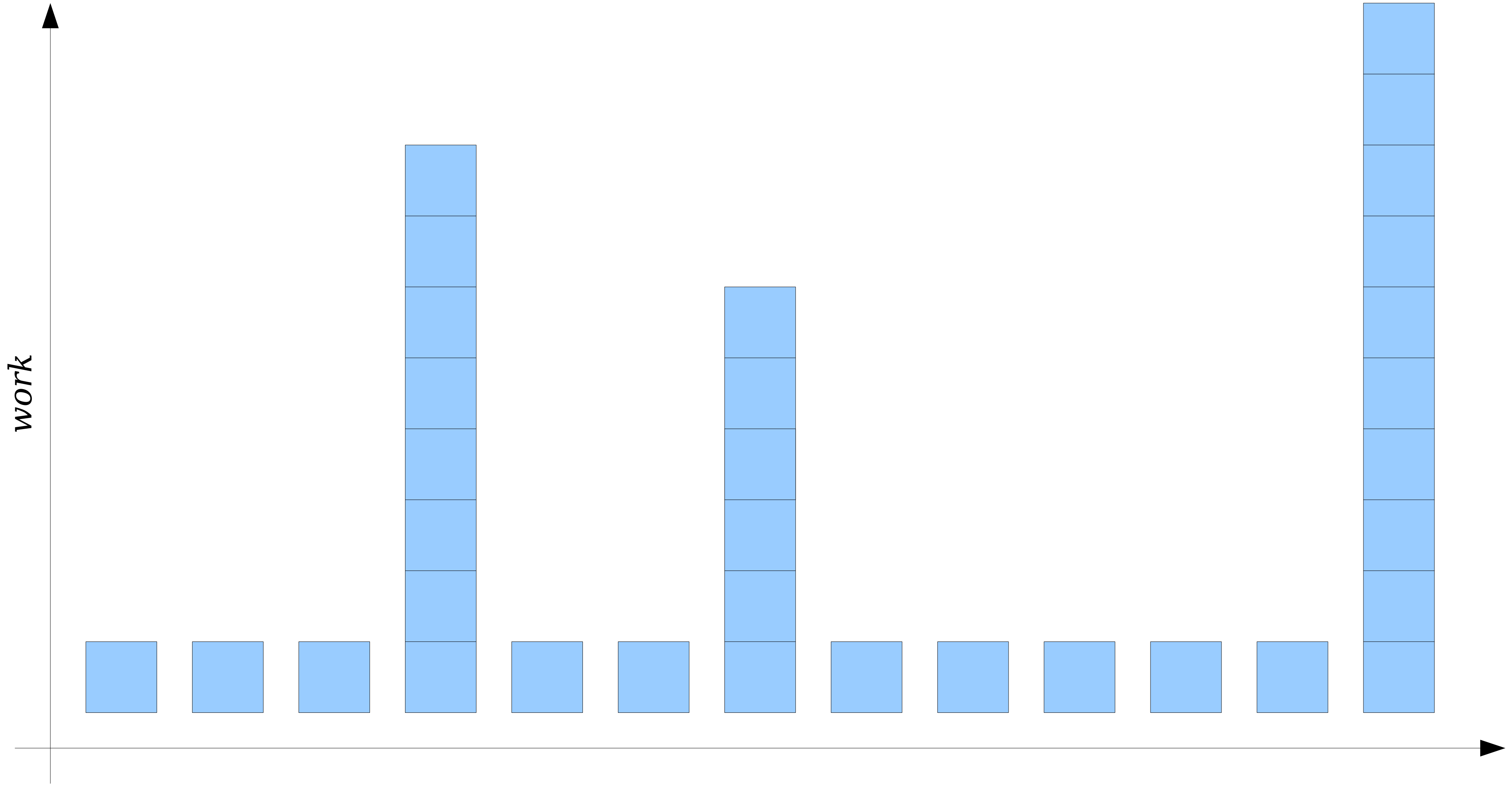
# Amortized Analysis

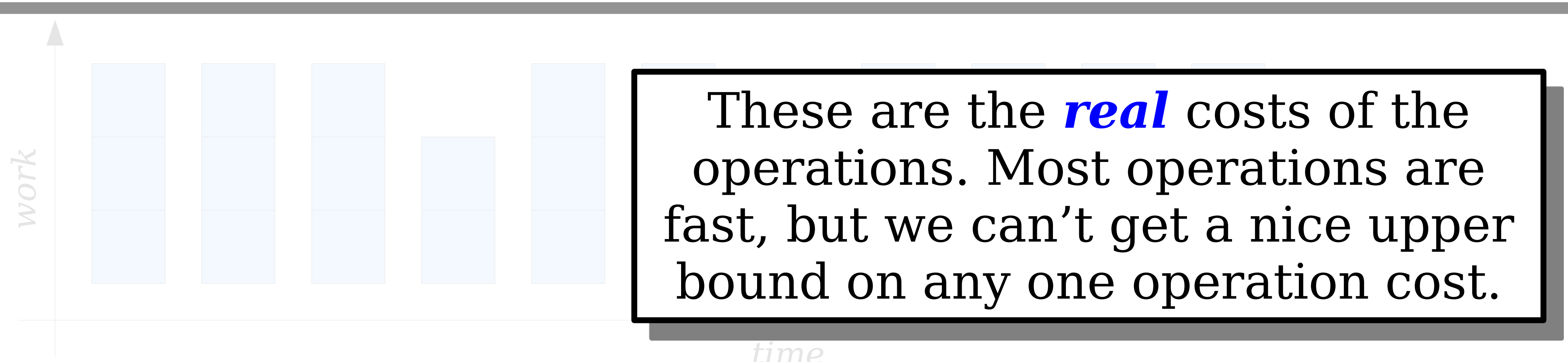
# The Setup

- 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?

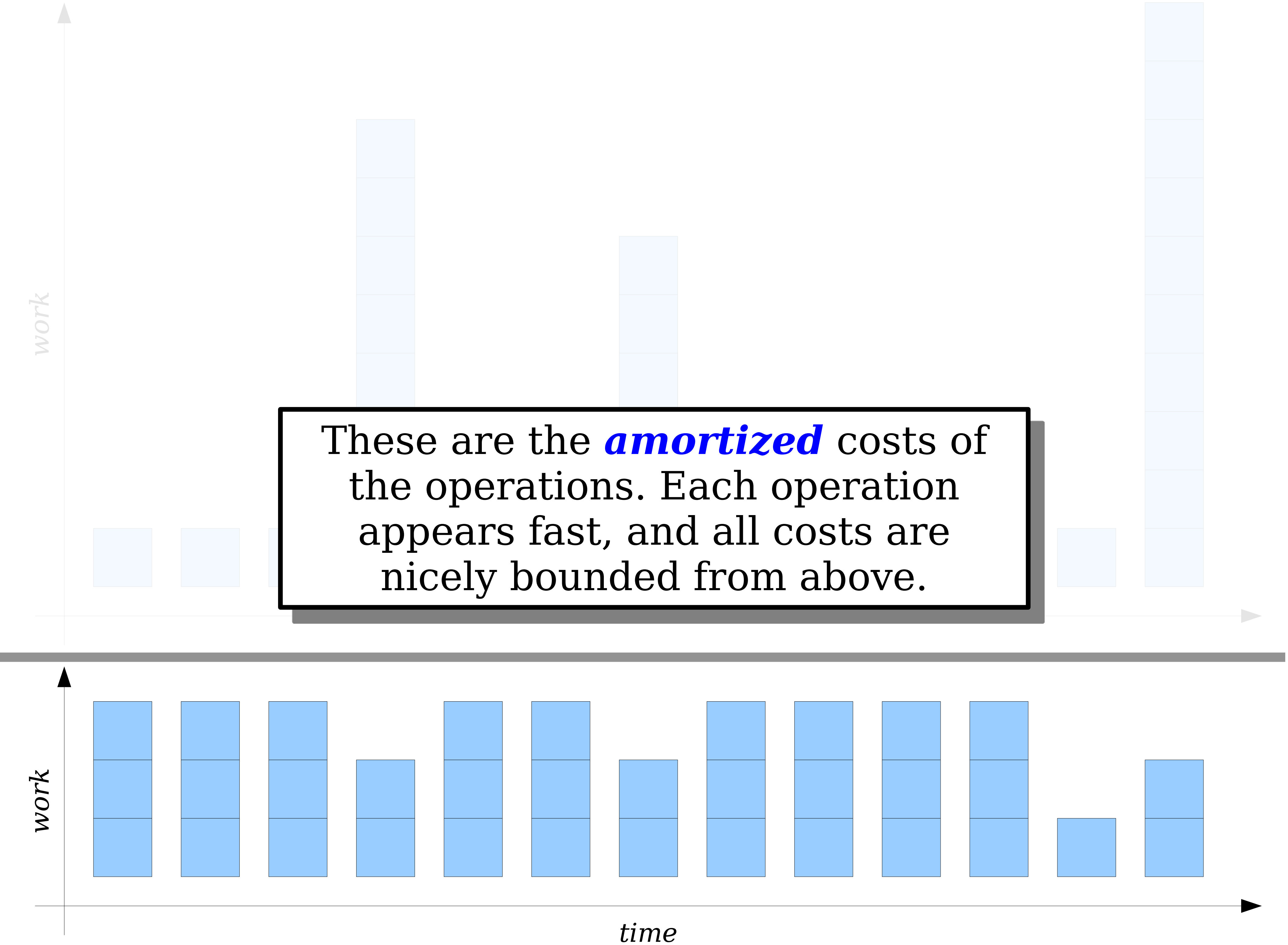



***Key Idea:*** Backcharge expensive operations to cheaper ones.




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.


work
These are the amortized costs of the operations. Each operation appears fast, and all costs are nicely bounded from above.
work
time

# 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:

$$\sum \textit{amortized-cost} \geq \sum \textit{real-cost}$$

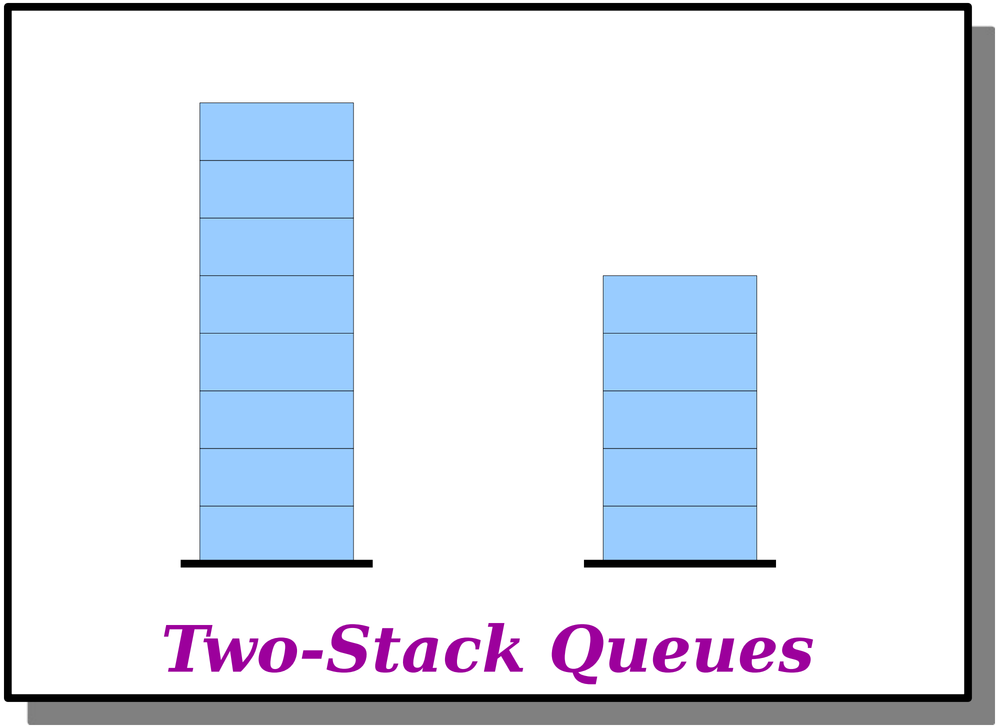
- Amortized costs shift work backwards from expensive operations onto cheaper ones.
  - Cheap operations are artificially made more expensive to pay for future cleanup work.
  - Expensive operations are artificially made cheaper by shifting the work backwards.

# 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.

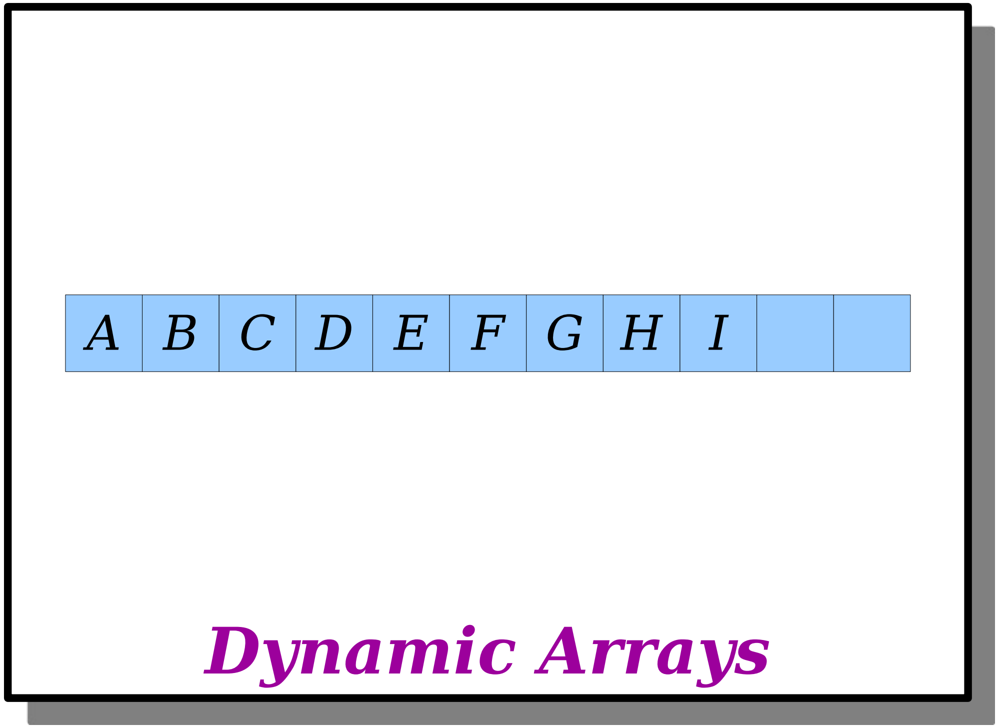


# Where We're Going

- 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.

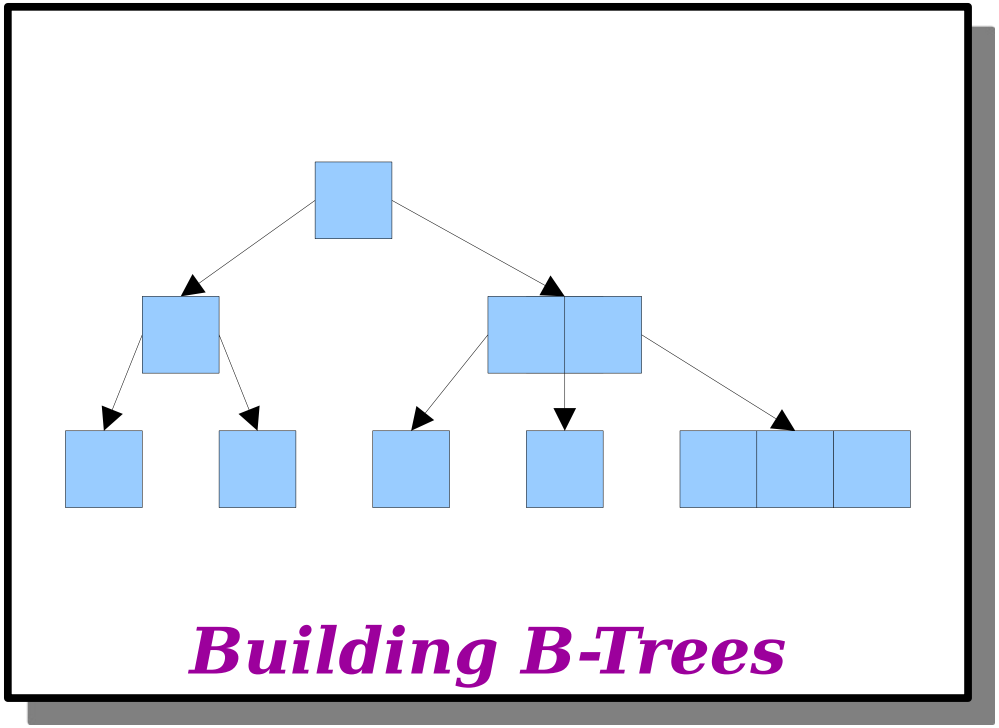


# Where We’re Going

- 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.



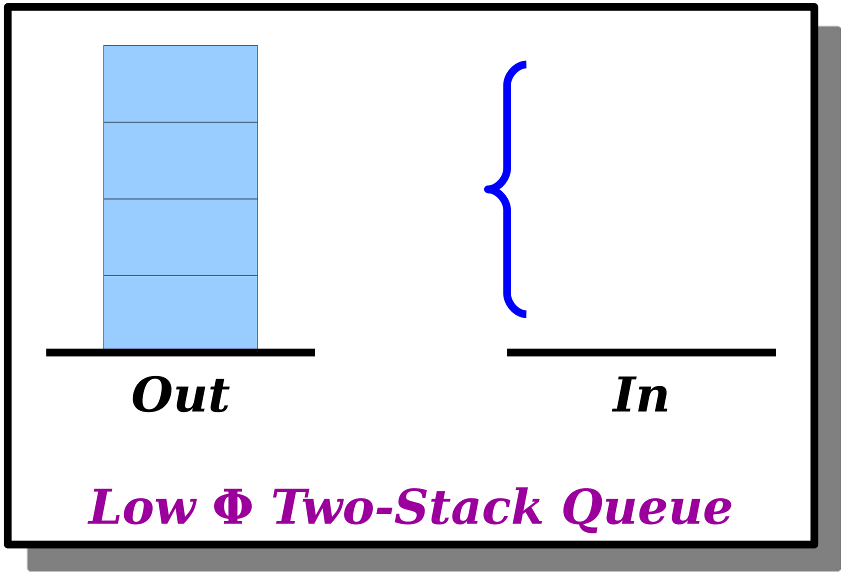
# Formalizing This Idea

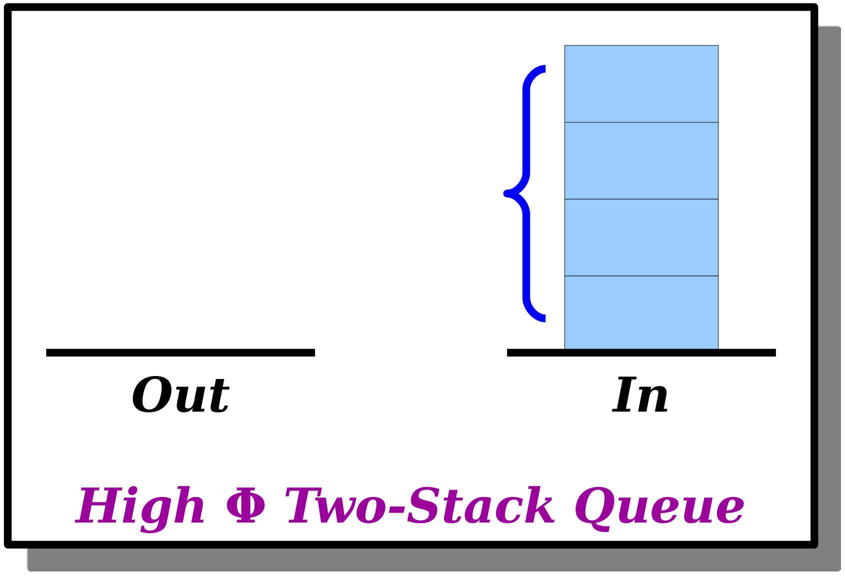
# Assigning Amortized Costs

- 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

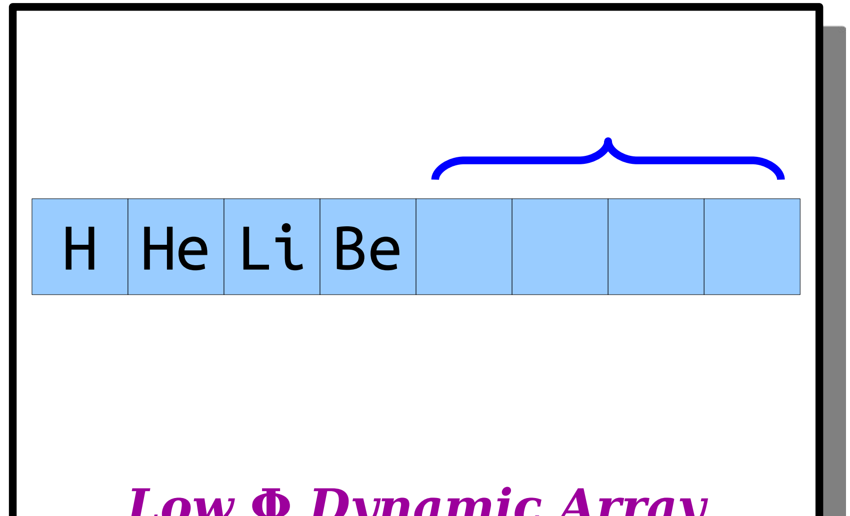
- To assign amortized costs, we'll need to measure how "messy" the data structure is.
- For each data structure, we define a ***potential function*** Φ that, in a sense, "quantifies messiness."
    - Φ is small when the data structure is "clean," and
    - Φ is large when the data structure is "messy."



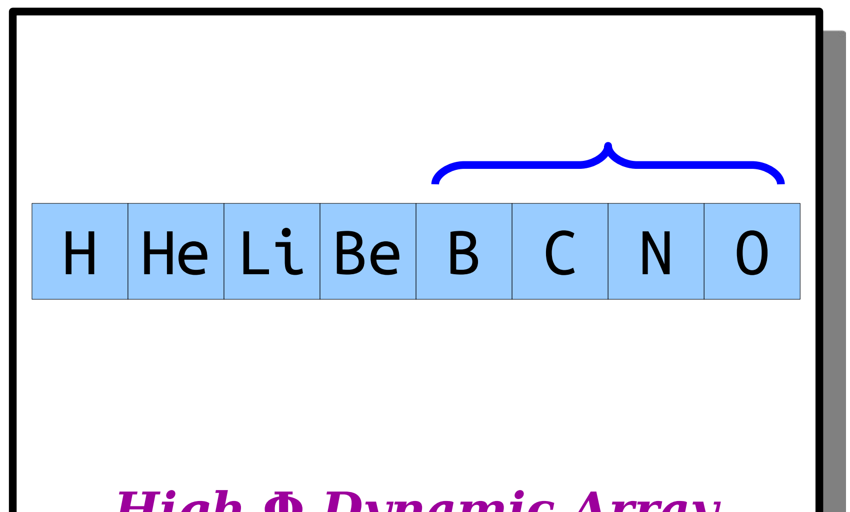



# Potential Functions

- To assign amortized costs, we'll need to measure how "messy" the data structure is.
- For each data structure, we define a ***potential function*** Φ that, in a sense, "quantifies messiness."
  - Φ is small when the data structure is "clean," and
  - Φ is large when the data structure is "messy."



***Low Φ Dynamic Array***



***High Φ Dynamic Array***

# Potential Functions

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

# Potential Functions

- Define the amortized cost of an operation to be

  ***amortized-cost = real-cost + k · ΔΦ***

  where $k$ is a constant under our control and ΔΦ is the difference between Φ just after the operation finishes and Φ just before the operation started:

  $$\Delta\Phi = \Phi_{after} - \Phi_{before}$$

- Intuitively:
  - If Φ increases, the data structure got "messier," and the amortized cost is *higher* than the real cost to account for future cleanup costs.
  - If Φ decreases, the data structure got "cleaner," and the amortized cost is *lower* than the real cost

# Why This Works

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

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

$$= \sum real\text{-}cost + k \cdot \left( \Phi_{end} - \Phi_{start} \right)$$

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

$$\begin{aligned}
\sum \textit{amortized-cost} &= \sum \left(\textit{real-cost} + k \cdot \Delta\Phi\right) \\
&= \sum \textit{real-cost} + k \cdot \sum \Delta\Phi \\
&= \sum \textit{real-cost} + k \cdot \left(\Phi_{end} - \Phi_{start}\right) \\
&\geq \sum \textit{real-cost}
\end{aligned}$$

Let's make two assumptions:

$$\Phi \geq 0.$$

$$\Phi_{start} = 0.$$

# Why This Works

$$\begin{aligned}
\sum amortized\text{-}cost &= \sum \left(real\text{-}cost + k \cdot \Delta\Phi\right) \\
&= \sum real\text{-}cost + k \cdot \sum \Delta\Phi \\
&= \sum real\text{-}cost + k \cdot \left(\Phi_{end} - \Phi_{start}\right) \\
&\geq \sum real\text{-}cost
\end{aligned}$$

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

# The Story So Far

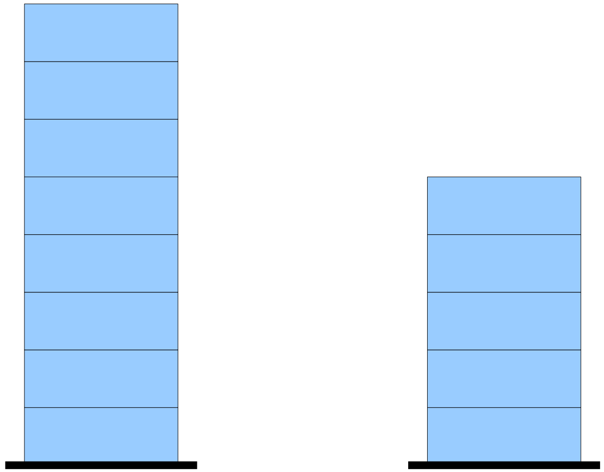
- We will assign amortized costs to each operation such that

$$\sum \textit{amortized-cost} \geq \sum \textit{real-cost}$$

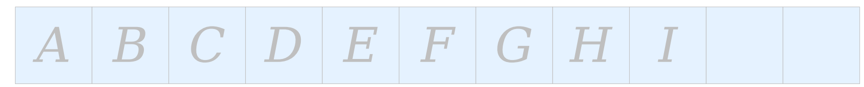
- To do so, define a ***potential function*** Φ such that
  - Φ measures how “messy” the data structure is,
  - $\Phi_{start} = 0$, and
  - $\Phi \geq 0$.
- Then, define amortized costs of operations as

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

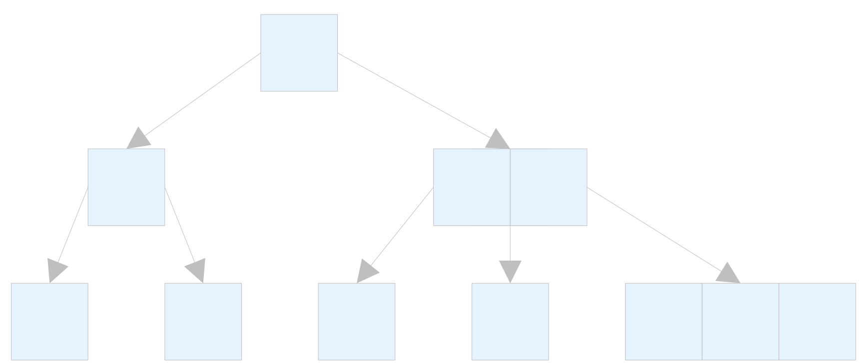
for a choice of $k$ under our control.



**Two-Stack Queues**




**Dynamic Arrays**



**Building B-Trees**

# The Two-Stack Queue

$\Phi$ = height of ***In*** stack


$$\begin{aligned} \textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\ &= O(1) + k \cdot 1 \\ &= \mathbf{O(1)} \end{aligned}$$

# The Two-Stack Queue

Φ = height of ***In*** stack


$$\begin{aligned} \textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\ &= O(1) + k \cdot 1 \\ &= \mathbf{O(1)} \end{aligned}$$

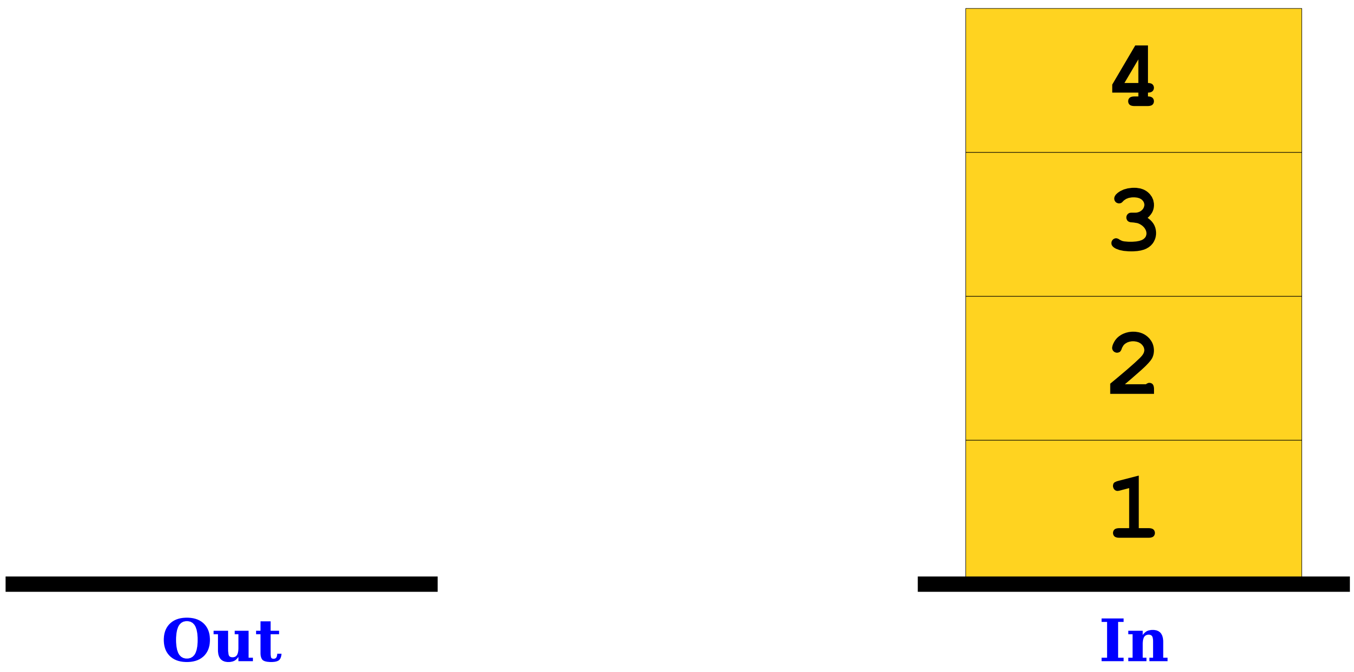
# The Two-Stack Queue

$\Phi$ = height of ***In*** stack


$$\begin{aligned} \textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\ &= O(1) + k \cdot 1 \\ &= \mathbf{O(1)} \end{aligned}$$

# The Two-Stack Queue

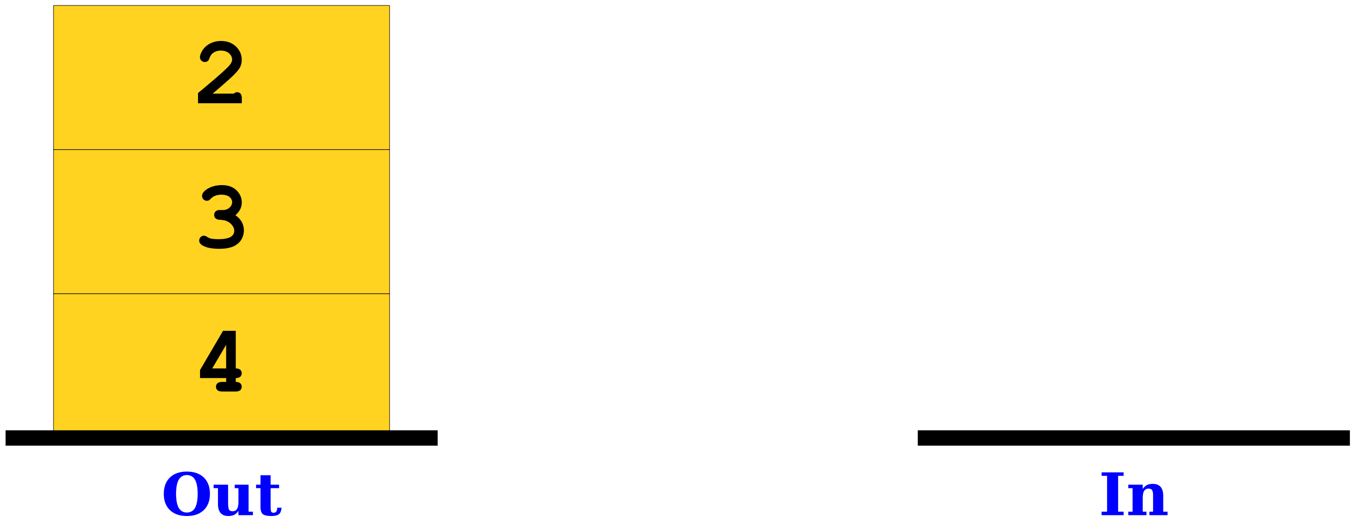
$\Phi$ = height of ***In*** stack




$$\begin{aligned} \textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\ &= O(1) + k \cdot 1 \\ &= \mathbf{O(1)} \end{aligned}$$

# The Two-Stack Queue

Φ = height of ***In*** stack




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

= $O(h) + k \cdot -h$ // $h$ = height of ***In*** stack

= **O(1)** // Choose $k$ strategically

# The Two-Stack Queue

Φ = height of ***In*** stack


$$\begin{aligned} \textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\ &= O(1) + k \cdot 0 \\ &= \mathbf{O(1)} \end{aligned}$$

***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

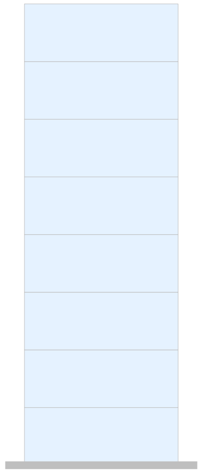
$$O(1) + k \cdot \Delta\Phi = O(1) + k \cdot 0 = O(1).$$

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.

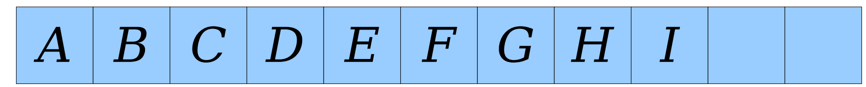
At the beginning of this operation, we have $\Phi = h$. At the end of this operation, we have $\Phi = 0$. Therefore, $\Delta\Phi = -h$, so the amortized cost of the operation is

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

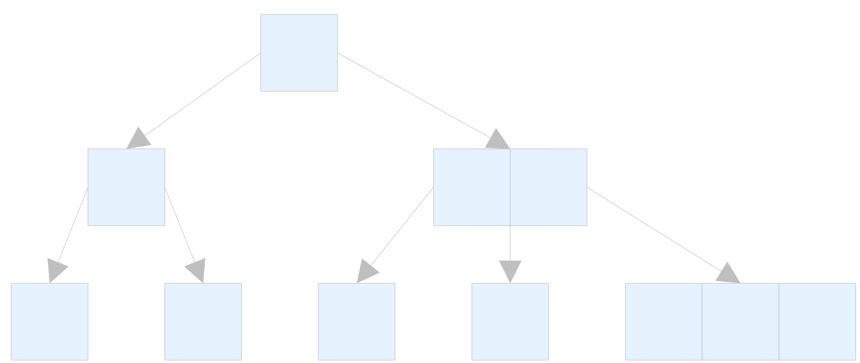
assuming we pick $k$ to cancel out the constant factor hidden in the $O(h)$ term. ■



Two-Stack Queues



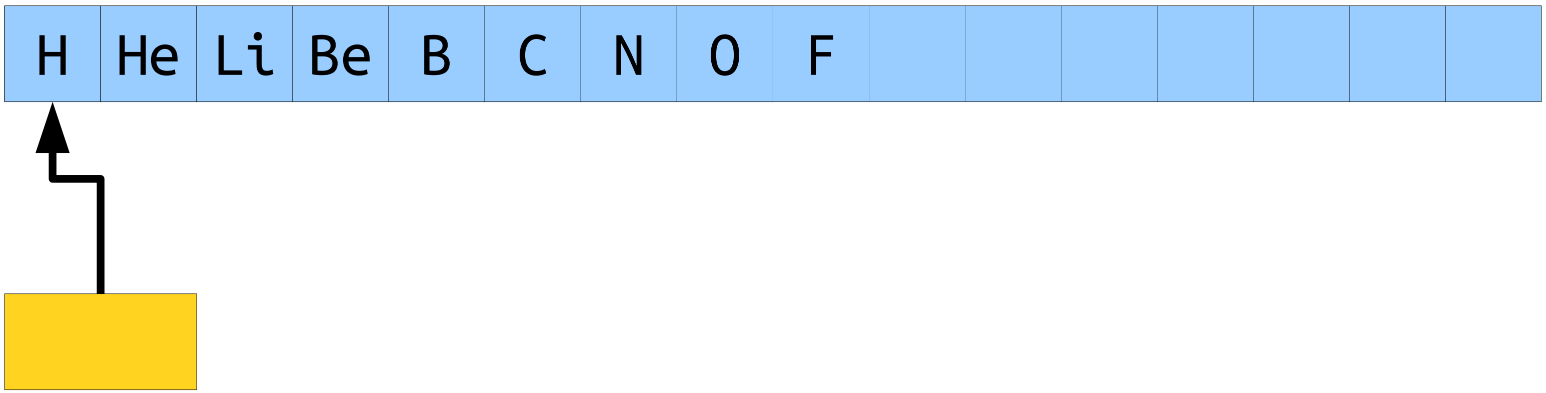

Dynamic Arrays



Building B-Trees

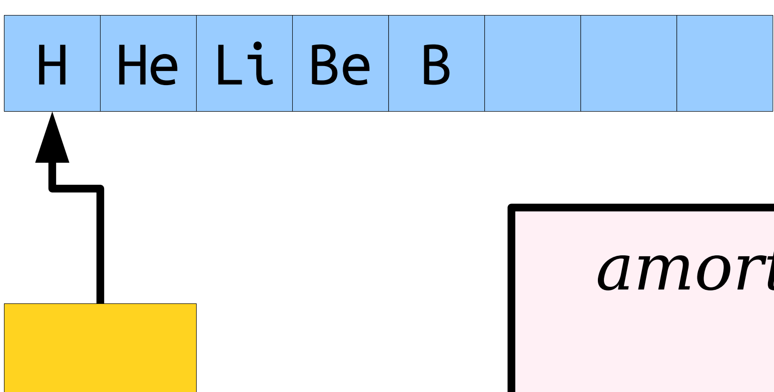
# Analyzing Dynamic Arrays

- ***Goal:*** Choose a potential function Φ such that the amortized cost of an append is O(1).
- ***Initial (wrong!) guess:*** Set Φ to be the number of free slots left in the array.



# Dynamic Arrays

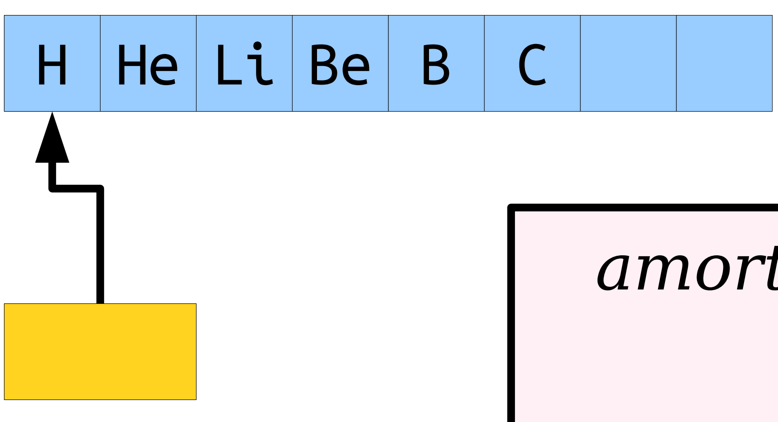
Φ = number of free slots




$$
\begin{aligned}
\textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\
&= O(1) + k \cdot -1 \\
&= \mathbf{O(1)}
\end{aligned}
$$

# Dynamic Arrays

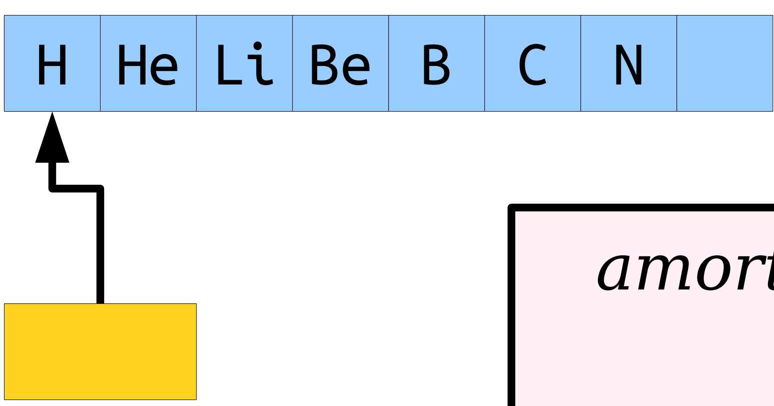
Φ = number of free slots




$$
\begin{aligned}
\textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\
&= O(1) + k \cdot -1 \\
&= \mathbf{O(1)}
\end{aligned}
$$

# Dynamic Arrays

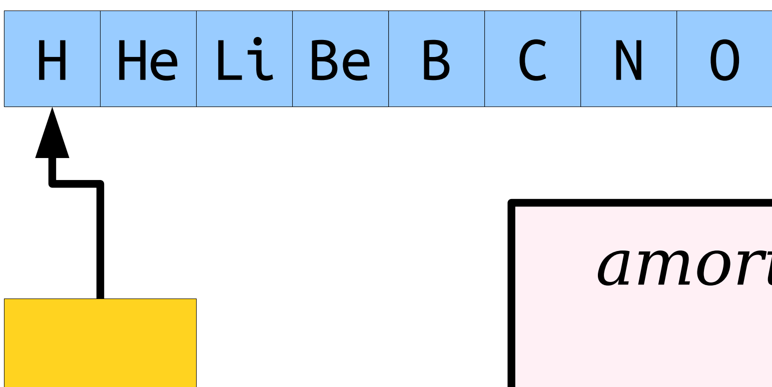
$\Phi$ = number of free slots




$$\begin{aligned} \textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\ &= O(1) + k \cdot -1 \\ &= \mathbf{O(1)} \end{aligned}$$

# Dynamic Arrays

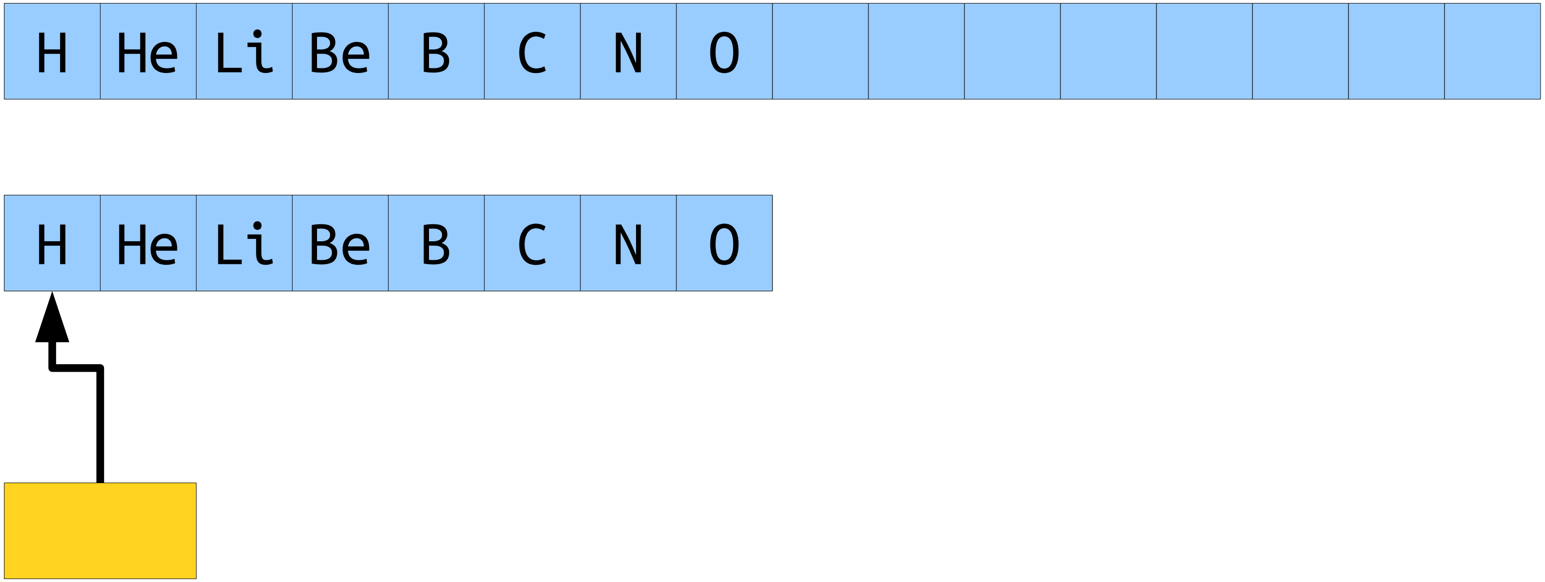
$\Phi$ = number of free slots




$$
\begin{aligned}
\textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\
&= O(1) + k \cdot -1 \\
&= \mathbf{O(1)}
\end{aligned}
$$

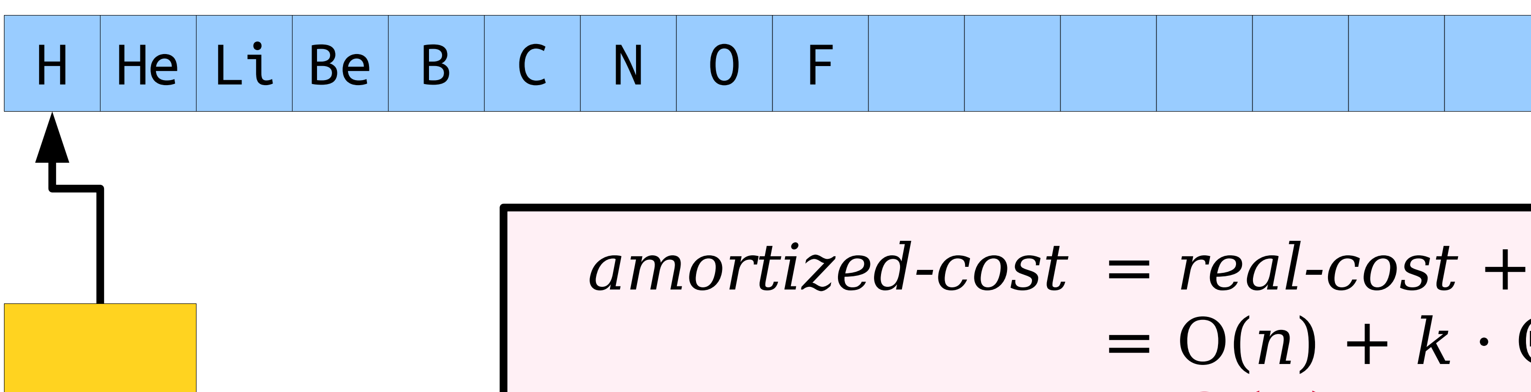
# Dynamic Arrays

Φ = number of free slots



# Dynamic Arrays

$\Phi$ = number of free slots




$$\begin{aligned} \textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\ &= O(n) + k \cdot \Theta(n) \\ &= \mathbf{O(\boldsymbol{n})} \end{aligned}$$

# Analyzing Dynamic Arrays

- ***Intuition:*** Φ 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 Φ be?

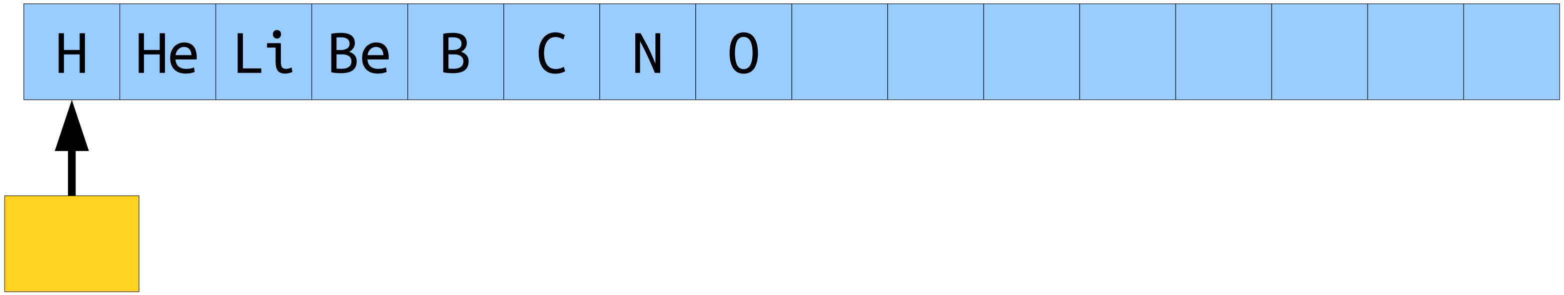
# Analyzing Dynamic Arrays

- The amortized cost of an append is

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

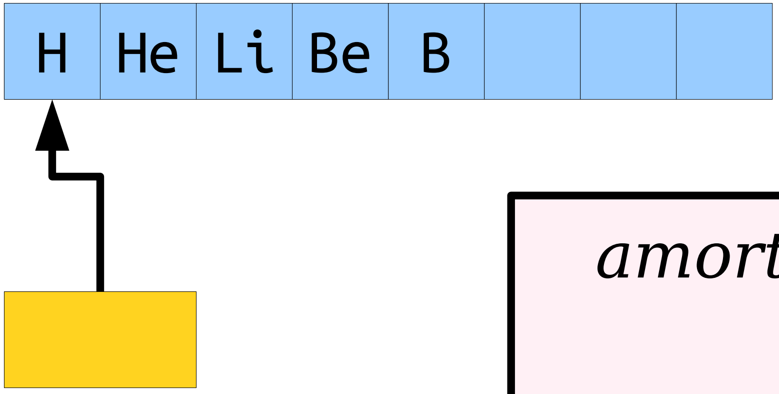
- When we double the array size, our real cost is $\Theta(n)$. We need $\Delta\Phi$ to be something like $-n$.
- ***Goal:*** Pick $\Phi$ so that
  - when there are no slots left, $\Phi \approx n$, and
  - right after we double the array size, $\Phi \approx 0$.
- With some trial and error, we can come up with

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



# Dynamic Arrays

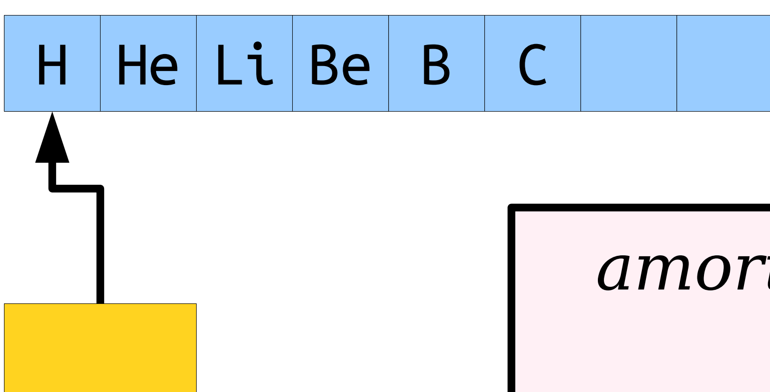
$\Phi$ = *#elems – #free-slots*




$$
\begin{aligned}
\textit{amortized-cost} &= \textit{real-cost} + k \cdot \Delta\Phi \\
&= O(1) + k \cdot 2 \\
&= \mathbf{O(1)}
\end{aligned}
$$

# Dynamic Arrays

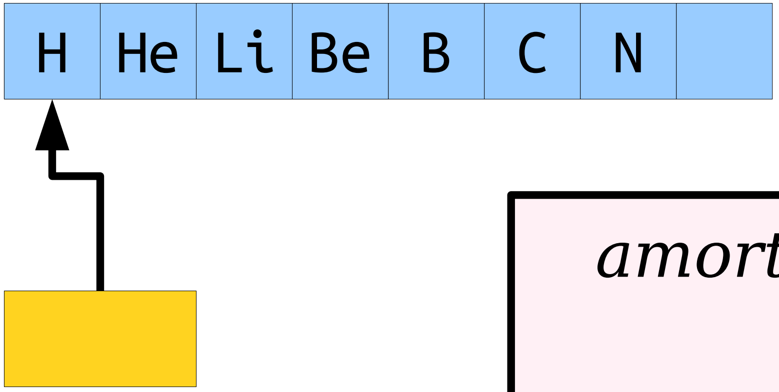
$$\Phi = \#elems - \#free\text{-}slots$$




$$\begin{aligned} amortized\text{-}cost &= real\text{-}cost + k \cdot \Delta\Phi \\ &= \mathrm{O}(1) + k \cdot 2 \\ &= \mathbf{O(1)} \end{aligned}$$

# Dynamic Arrays

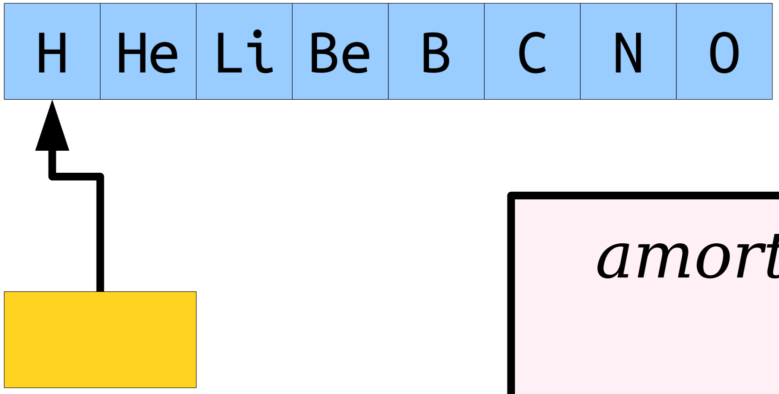
$\Phi = \#elems - \#free\text{-}slots$




$$
\begin{aligned}
amortized\text{-}cost &= real\text{-}cost + k \cdot \Delta\Phi \\
&= O(1) + k \cdot 2 \\
&= \mathbf{O(1)}
\end{aligned}
$$

# Dynamic Arrays

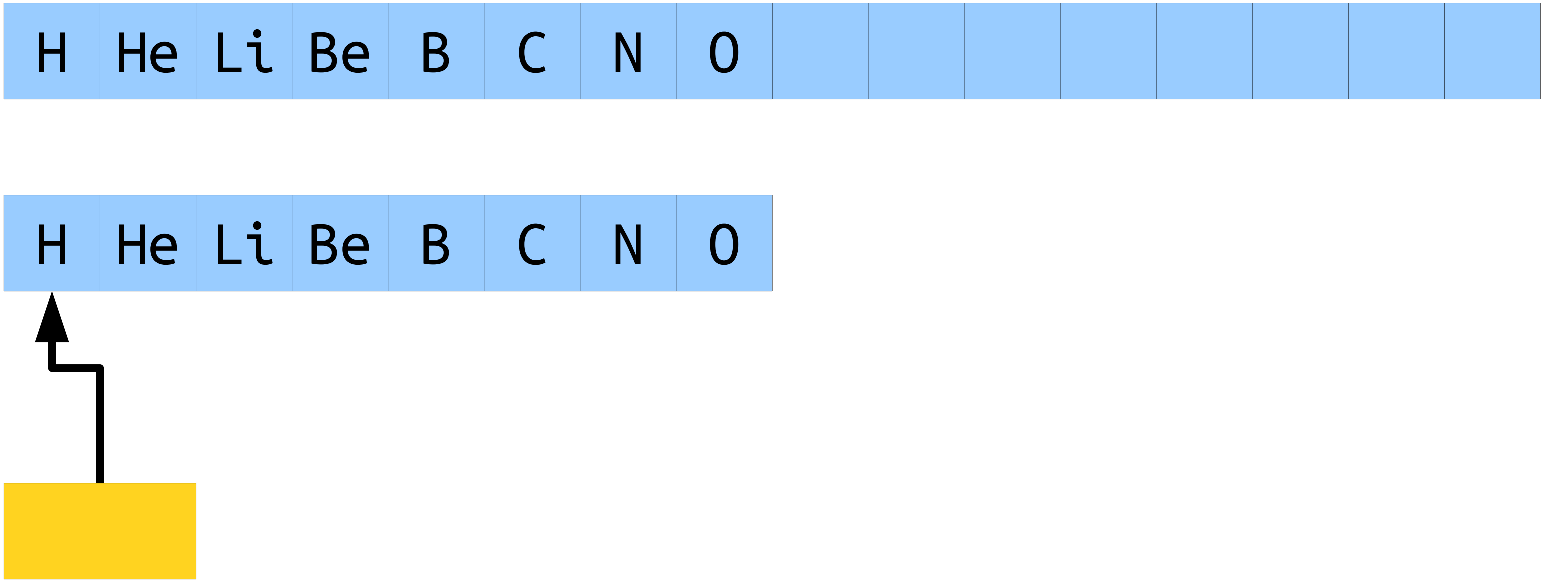
$\Phi = \#elems - \#free\text{-}slots$




$$
\begin{aligned}
amortized\text{-}cost &= real\text{-}cost + k \cdot \Delta\Phi \\
&= O(1) + k \cdot 2 \\
&= \mathbf{O(1)}
\end{aligned}
$$

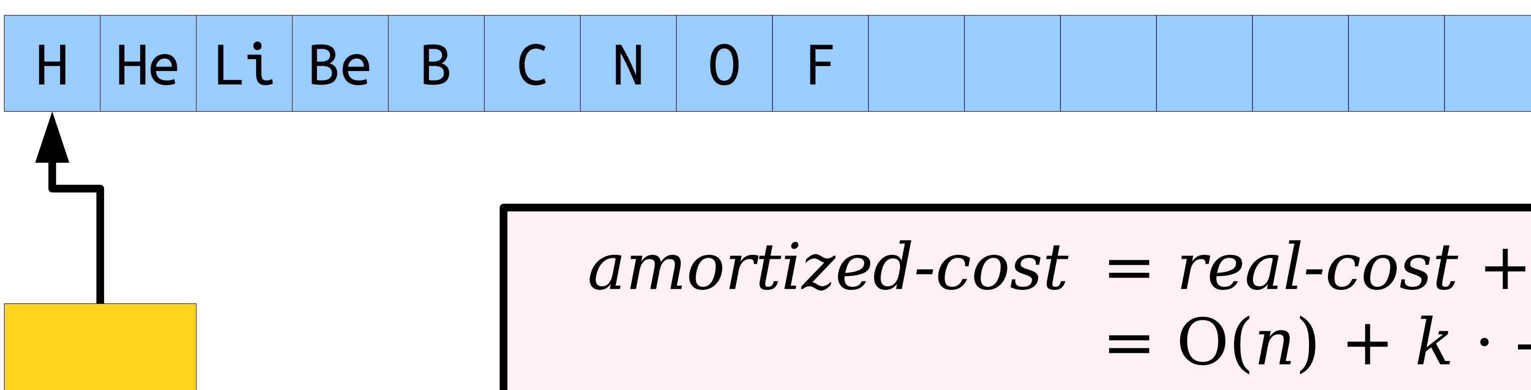
# Dynamic Arrays

$\Phi$ = #*elems* – #*free-slots*



# Dynamic Arrays

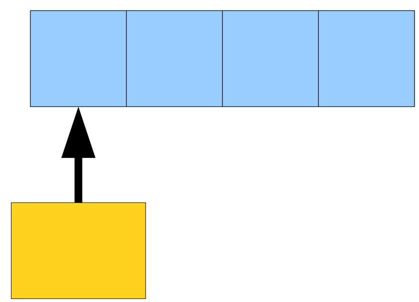
$\Phi = \#elems - \#free\text{-}slots$




$$\begin{aligned} amortized\text{-}cost &= real\text{-}cost + k \cdot \Delta\Phi \\ &= O(n) + k \cdot -\Theta(n) \\ &= \mathbf{O(1)} \text{ // Pick } k \text{ well} \end{aligned}$$

# A Caveat

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



$\Phi = -4$

- ***Quick fix:*** This is an edge case, so set

$$\Phi = \max\{\ 0,\ \#elems - \#free\text{-}slots\ \}$$

***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 $\Phi = \max\{ 0, n - \#\textit{free-slots} \}$, 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 $\Phi = n - \#\textit{free-slots}$.

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

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

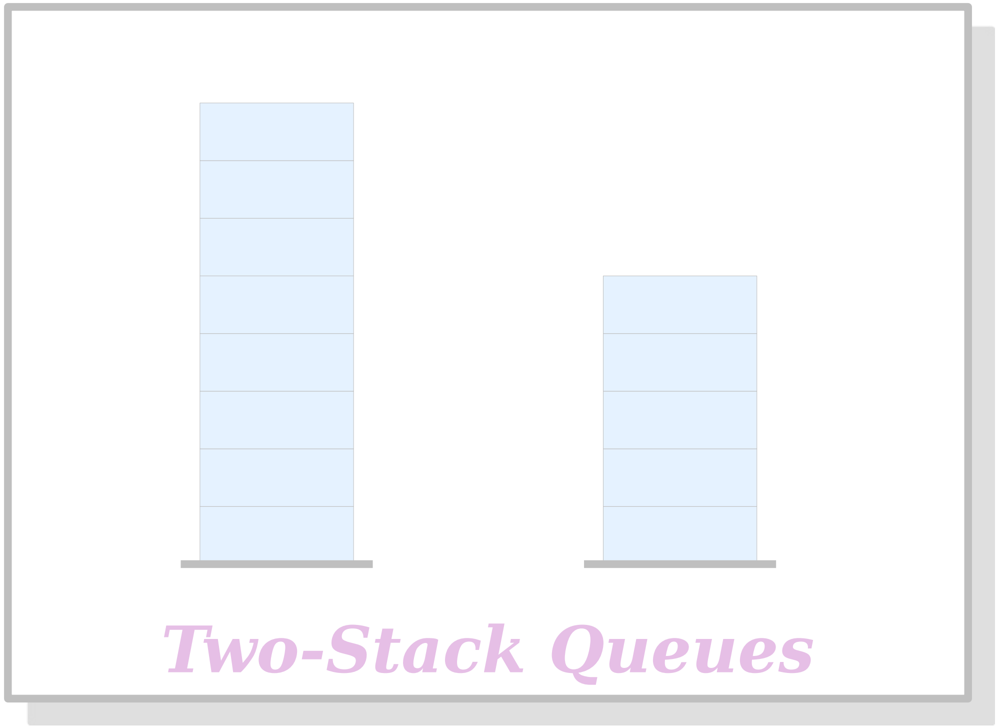
Otherwise, the operation copies $n$ elements into a new array twice as large as before, increasing the number of free slots to $n$, then fills one of those slots. Just before the operation we had $\Phi = n$, and just after the operation we have $\Phi = 2$. Therefore, the amortized cost is

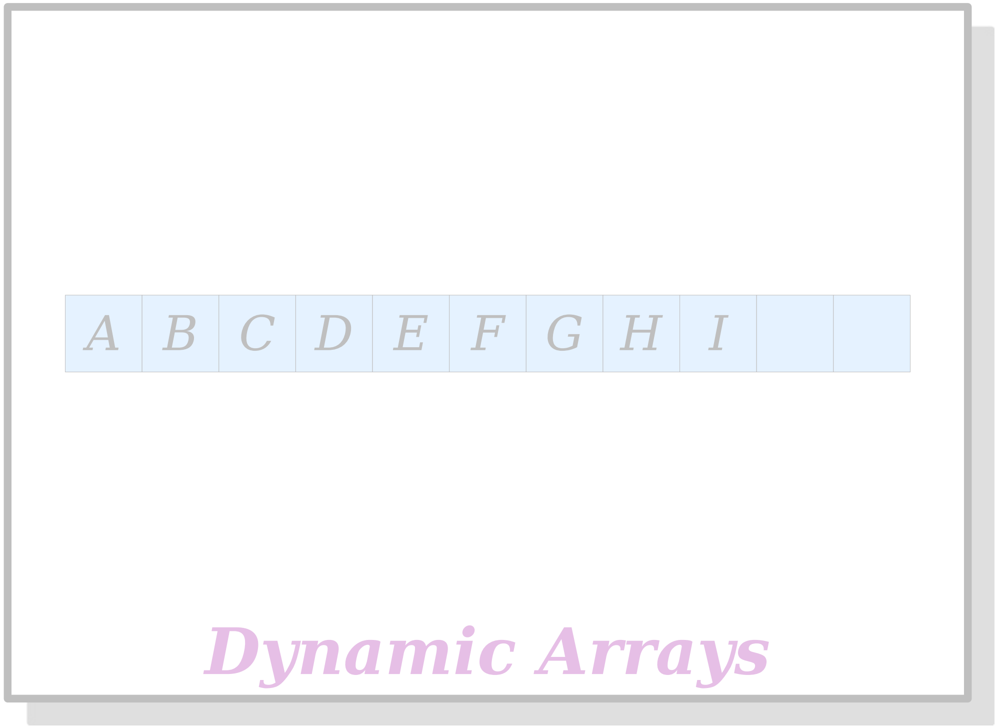
$$O(n) + k \cdot \Delta\Phi = O(n) + k \cdot (2 - n) = O(n) - nk + 2k,$$

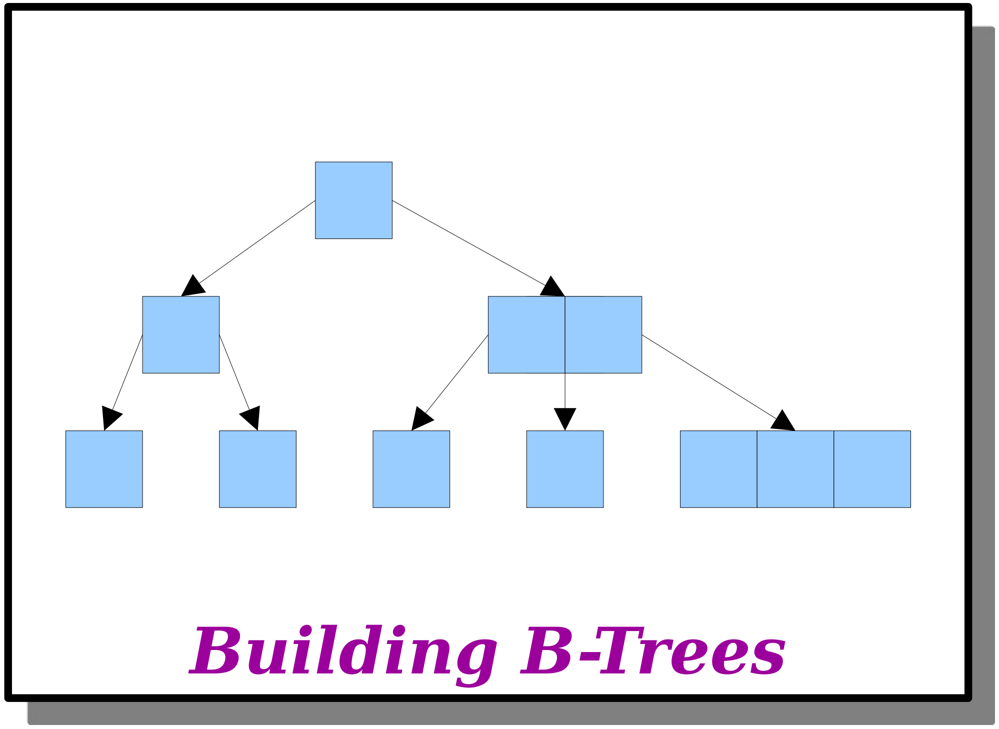
which can be made to equal O(1) by choosing the the $k$ term to match the constant hidden in the $O(n)$ term. ■

# 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 ¼ 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).

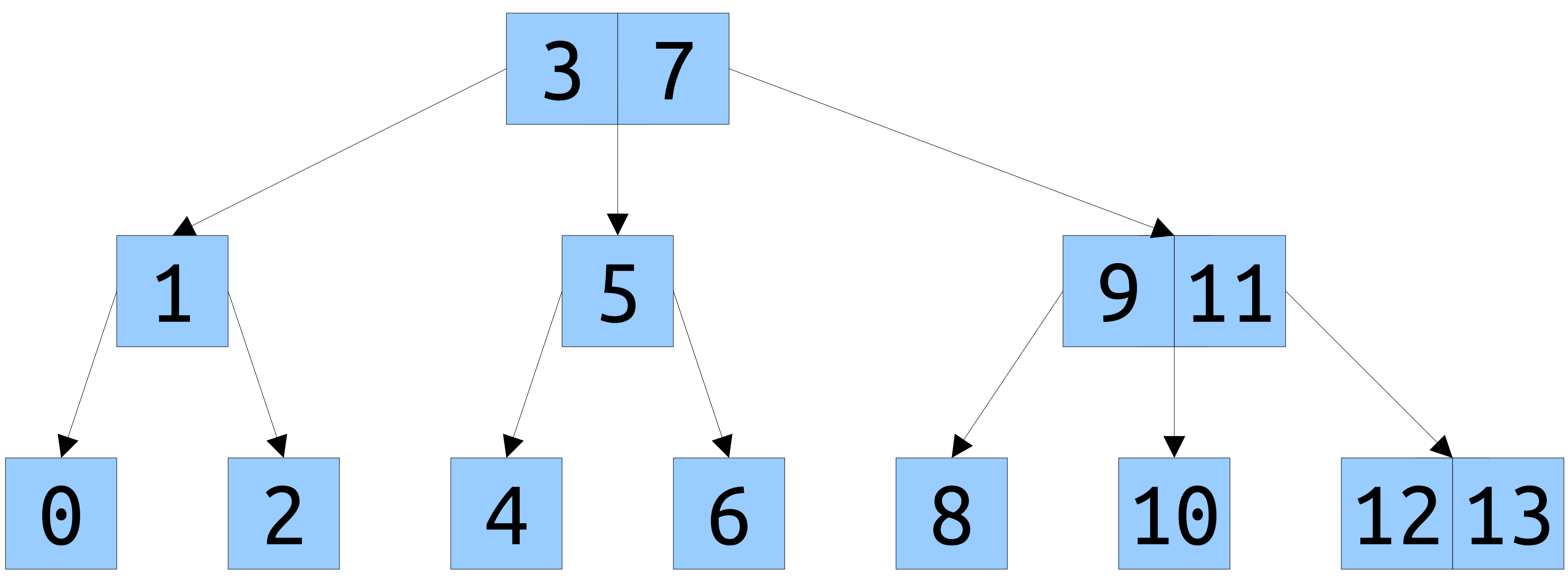

Two-Stack Queues


A B C D E F G H I
Dynamic Arrays


Building B-Trees

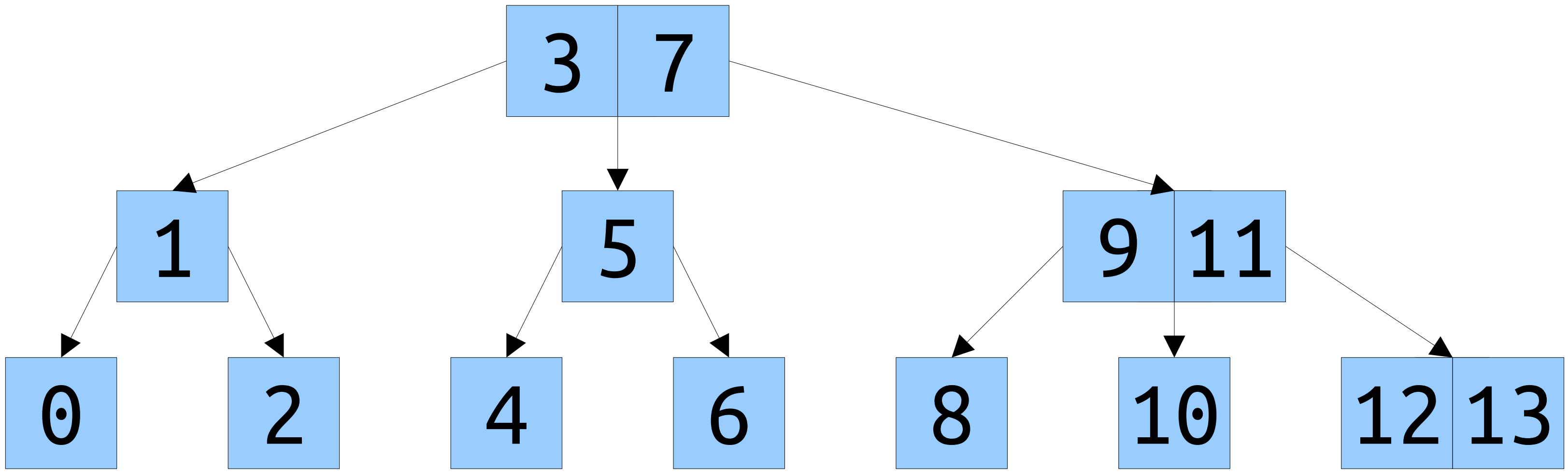
# Building B-Trees

- ***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.



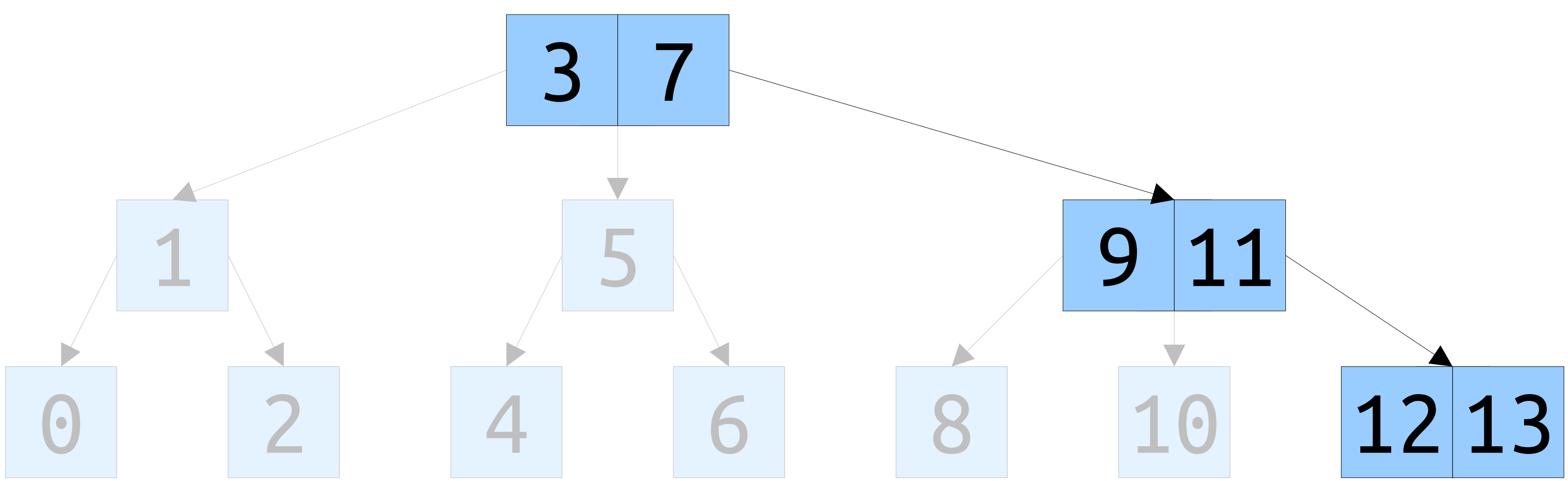
# Building B-Trees

- 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: $\boldsymbol{\Theta(bL)}$.
- ***Goal:*** Pick a potential function $\Phi$ so that we can offset this cost and make each append cost amortized O(1).



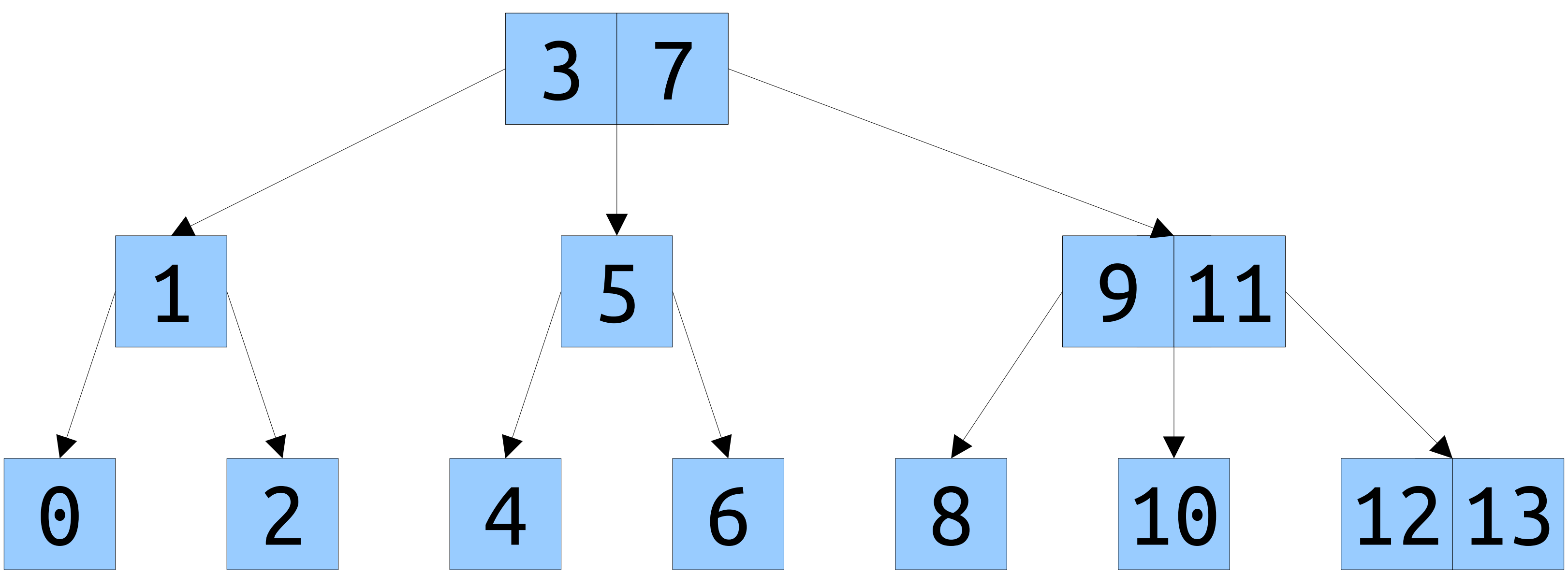
# Building B-Trees

- 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 Φ to be the number of keys in the right spine of the tree.



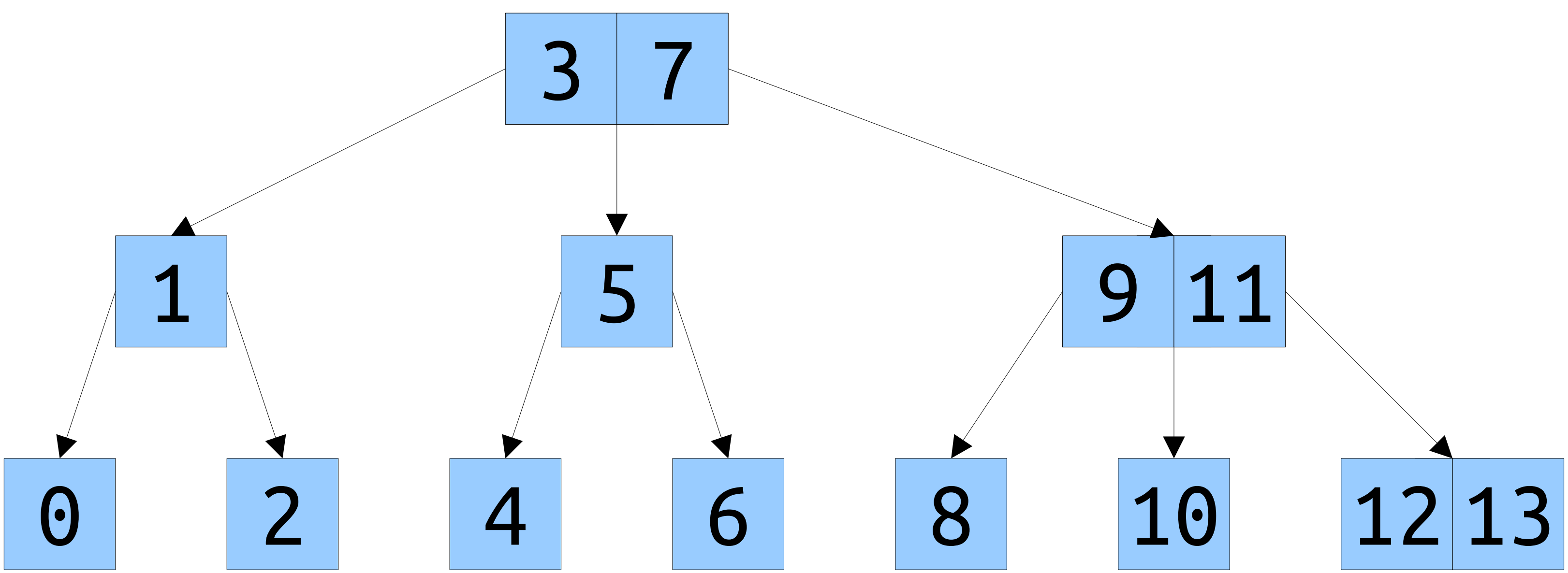
# Building B-Trees

- 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.
- 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 Θ 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 Φ 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 Φ that, intuitively, measures how "messy" the data structure is. We then set

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

- For simplicity, we assume that Φ is nonnegative and that Φ 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.