



<https://algs4.cs.princeton.edu>

1.3 STACKS AND QUEUES I

- ▶ *APIs*
- ▶ *array implementations*
- ▶ *resizable arrays*
- ▶ *amortized analysis*
- ▶ *generics*



<https://algs4.cs.princeton.edu>

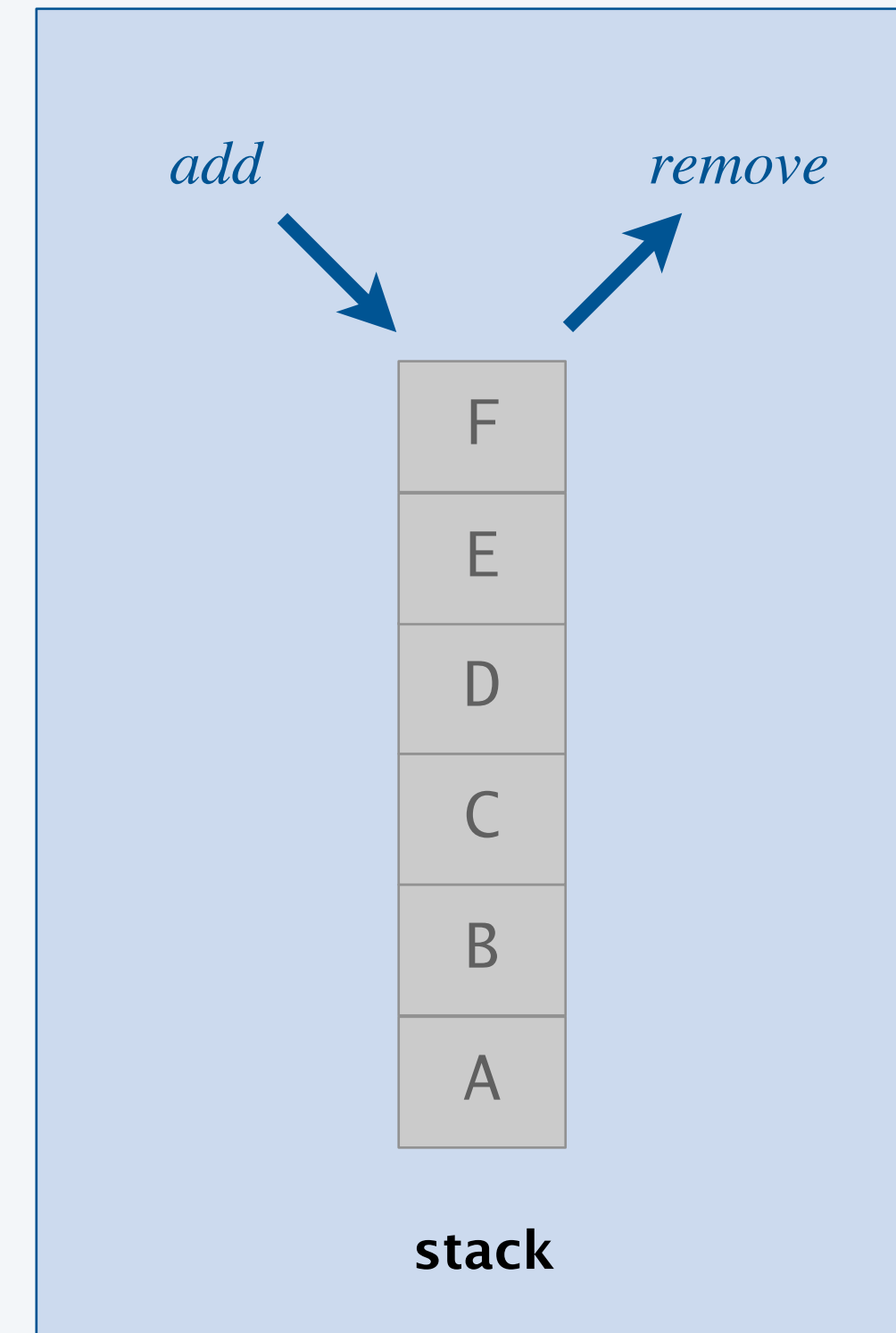
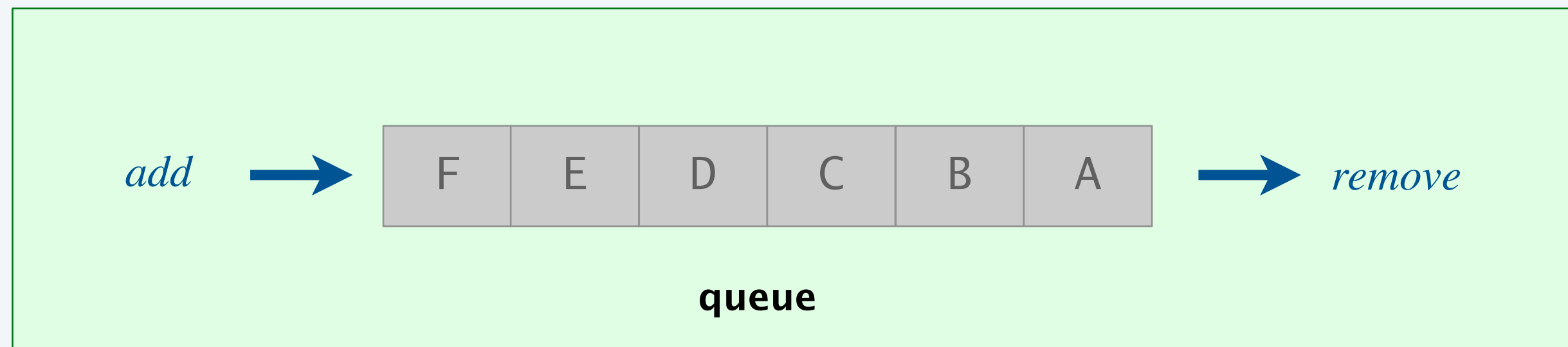
1.3 STACKS AND QUEUES I

- ▶ *APIs*
- ▶ *array implementations*
- ▶ *resizable arrays*
- ▶ *amortized analysis*
- ▶ *generics*

Stacks and queues

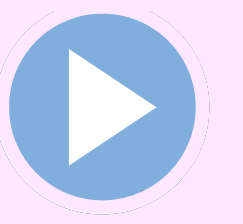
Fundamental data types.

- Value: **collection** of objects.
- Operations: **add**, **remove**, iterate, size, test if empty.
- Intent is clear when we add.
- Which item do we remove?



Stack. Remove the item **most** recently added. ← *LIFO = "last in first out"*

Queue. Remove the item **least** recently added. ← *FIFO = "first in first out"*



```
public static double square(double a) {  
    return a*a;  
}
```

variable	a
value	3.0

square(3.0)

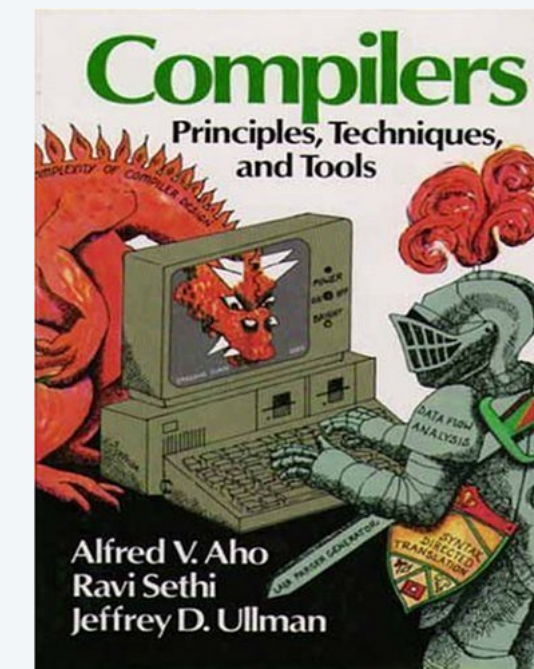
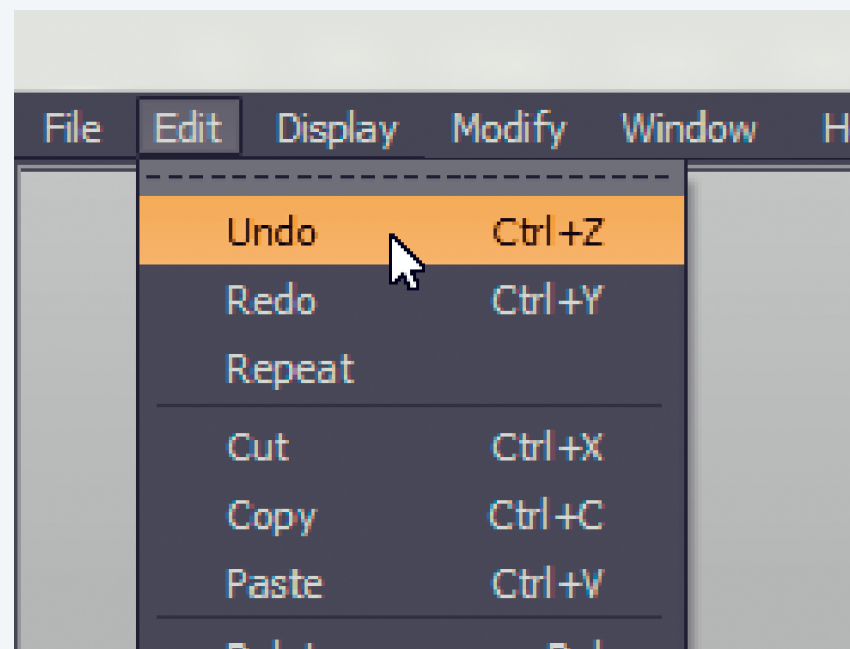
hypotenuse(3.0, 4.0)

main()

function-call stack

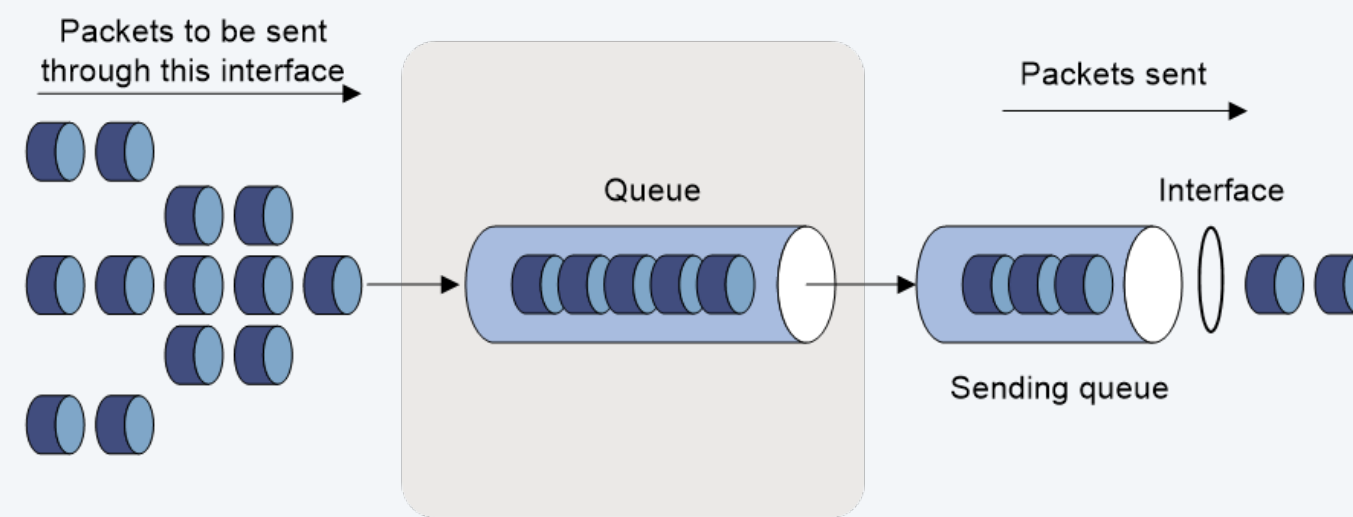
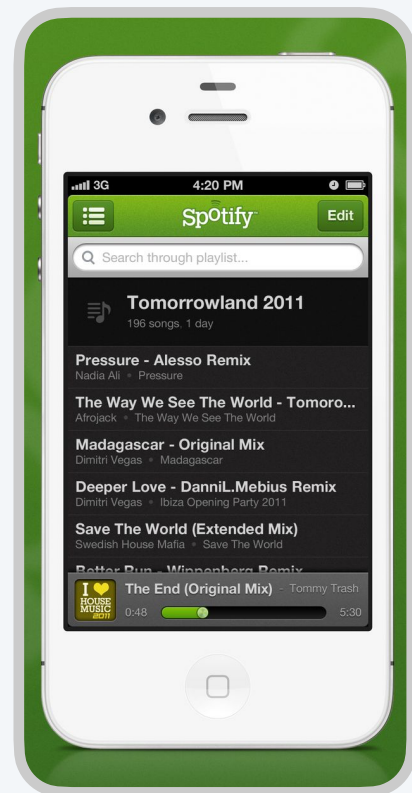
Stack applications

- Rendering text and graphics: PostScript, PDF, ...
- Web browser history: back and forward buttons.
- Function calls: Java virtual machine, Linux kernel, ...
- Undo/redo functionality: text editors, photo editors, games, ...
- Compilers: evaluating expressions, parsing syntax, balanced parentheses, ...
- ...



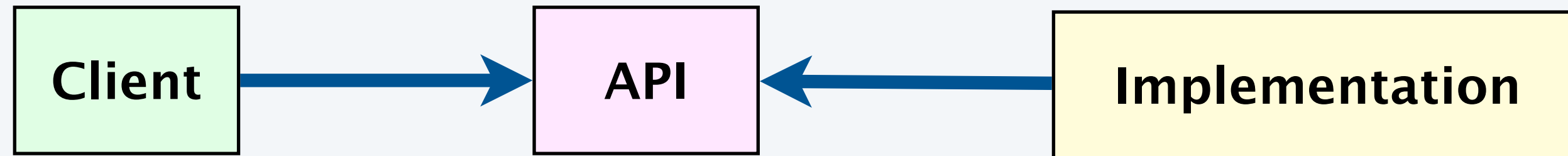
Queue applications

- Media playlists: jukebox, Spotify, Netflix, Peloton, ...
- Requests on a shared resource: printer, CPU, GPU, ...
- Asynchronous data transfer: file I/O, pipes, sockets, ...
- Data buffers: sound card, streaming video, input devices, ...
- Simulations of the real world: customer service, traffic analysis, baggage claim, ...
- ...



Data type design: API, client, and implementation

Separate client and implementation via API.



API: operations that characterize the behavior of a data type.

Client: code that uses a data type through its API.

Implementation: code that implements the API operations.

Benefits.

- **Design:** develop and maintain reusable code.
- **Performance:** substitute faster implementations.

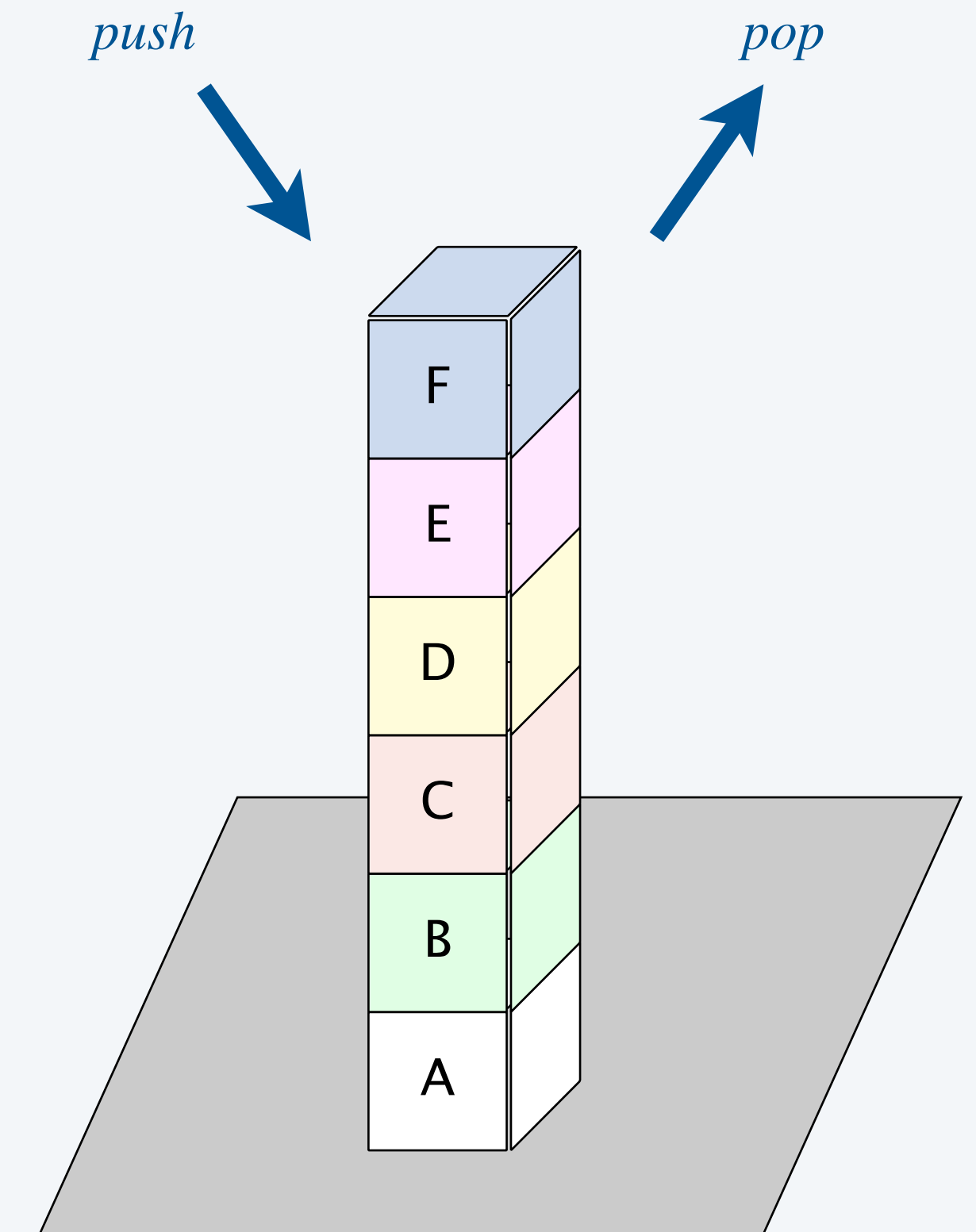
Ex. Stack, queue, priority queue, symbol table, set, union-find, ...

Stack data type. Our textbook data type for stacks.

← available with `javac-als4` and `java-als4` commands

		description
	<code>public class Stack<Item></code>	
	<code>Stack()</code>	<i>create an empty stack</i>
<code>void</code>	<code>push(Item item)</code>	<i>add a new item to the stack</i>
<code>Item</code>	<code>pop()</code>	<i>remove and return the item most recently added</i>
<code>boolean</code>	<code>isEmpty()</code>	<i>is the stack empty?</i>

“generic type parameter”



Performance goals. Every operation takes $\Theta(1)$ time; stack with n items uses $\Theta(n)$ memory.

Queue data type. Our textbook data type for queues.

← available with javac-als4
and java-als4 commands



<code>public class Queue<Item></code>	description
<code>Queue()</code>	<i>create an empty queue</i>
<code>void enqueue(Item item)</code>	<i>add a new item to the queue</i>
<code>Item dequeue()</code>	<i>remove and return the item least recently added</i>
<code>boolean isEmpty()</code>	<i>is the queue empty?</i>

Performance goals. Every operation takes $\Theta(1)$ time; queue with n items uses $\Theta(n)$ memory.

Warmup client

Goal. Read strings from standard input and print in **reverse order**.

*access library
in algs4.jar
(typically omitted)*

```
import edu.princeton.cs.algs4.Stack;
import edu.princeton.cs.algs4.StdIn;
import edu.princeton.cs.algs4.StdOut;

public class Reverse {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();

        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            stack.push(s);
        }

        while (!stack.isEmpty()) {
            String s = stack.pop();
            StdOut.print(s + " ");
        }
        StdOut.println();
    }
}
```

*“type argument”
(can be any reference type)*

*declare and
create stack*

*read strings from
standard input and
push onto stack*

*pop all strings
and print to
standard output*

```
~/cos226/stacks> javac-algs4 Reverse.java
~/cos226/stacks> java-algs4 Reverse
I have a dream today
<Ctrl-D>
today dream a have I
```



<https://algs4.cs.princeton.edu>

1.3 STACKS AND QUEUES I

- ▶ *APIs*
- ▶ *array implementations*
- ▶ *resizable arrays*
- ▶ *amortized analysis*
- ▶ *generics*

Stack API (warmup)

Warmup API. Stack of **strings** data type, with fixed **maximum capacity**.

```
public class FixedCapacityStackOfStrings
```

```
    FixedCapacityStackOfStrings(int capacity)
```

```
    void push(String item)
```

```
    String pop()
```

```
    boolean isEmpty()
```

create an empty stack

add a new string to stack

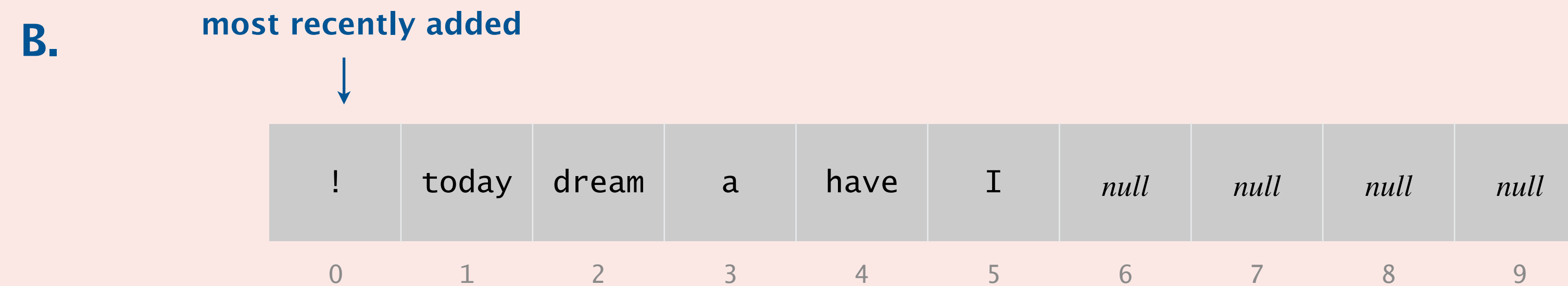
*remove and return the string
most recently added*

is the stack empty?

↑
*artificial limit
(stay tuned)*



How to implement efficiently a fixed-capacity stack with an array?

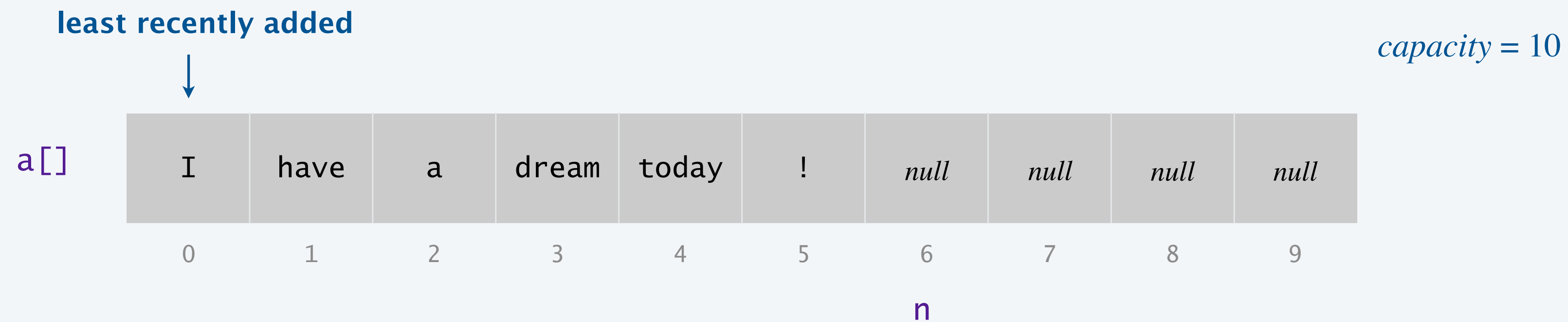


C. *Both A and B.*

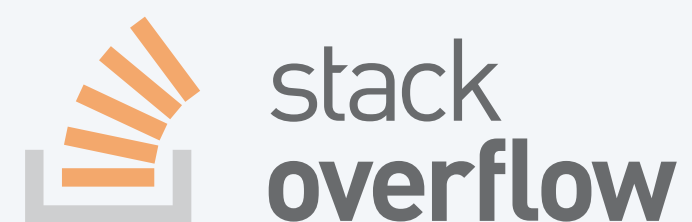
D. *Neither A nor B.*

Fixed-capacity stack: array implementation

- Use array `a[]` to store n items on stack.
- Push: add new item at `a[n]`.
- Pop: remove item from `a[n-1]`.



Defect. Stack overflows when n exceeds *capacity*. [stay tuned]



Fixed-capacity stack: array implementation

```
public class FixedCapacityStackOfStrings {
    private String[] a;
    private int n = 0;

    public FixedCapacityStackOfStrings(int capacity) {
        a = new String[capacity];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(String item) {
        a[n++] = item;
    }

    public String pop() {
        return a[--n];
    }
}
```

*post-increment operator:
use as index into array;
then increment n*

*pre-decrement operator:
decrement n;
then use as index into array*

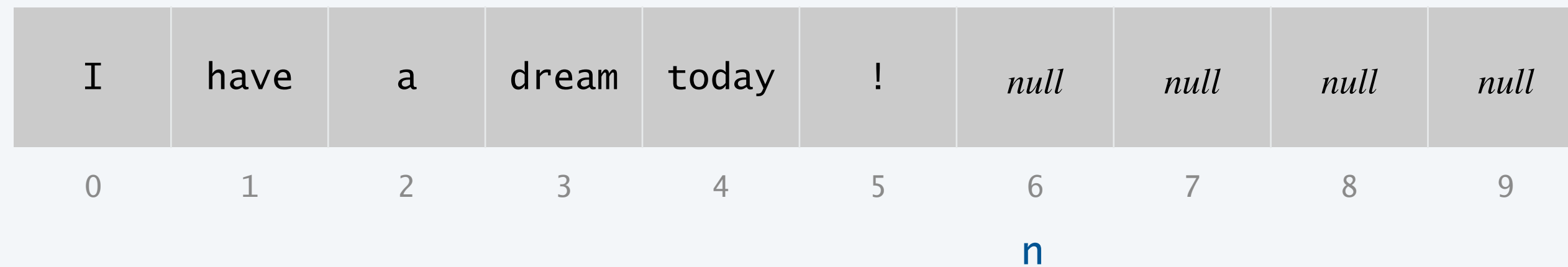
Stack considerations

Underflow. Throw exception if `pop()` called when stack is empty.

Overflow. Use “resizable array” to avoid overflow. [next section]

Null items. For simplicity, we allow `null` items to be added.

Loitering. Holding an object reference when it is no longer needed.



```
public String pop() {  
    return a[--n];  
}
```

loitering

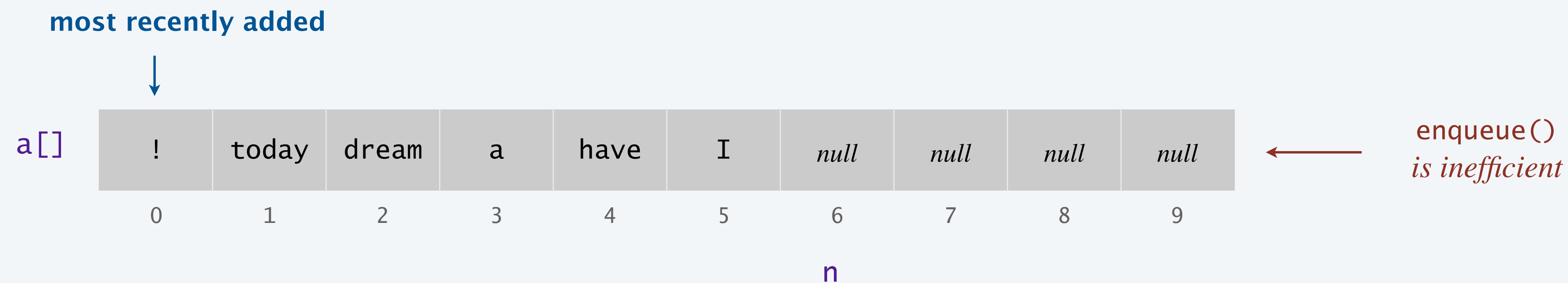
```
public String pop() {  
    String item = a[n-1];  
    a[n-1] = null;  
    n--;  
    return item;  
}
```

no loitering

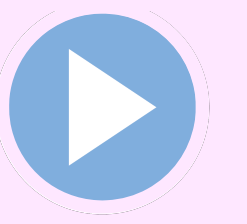


Fixed-capacity queue: array implementation

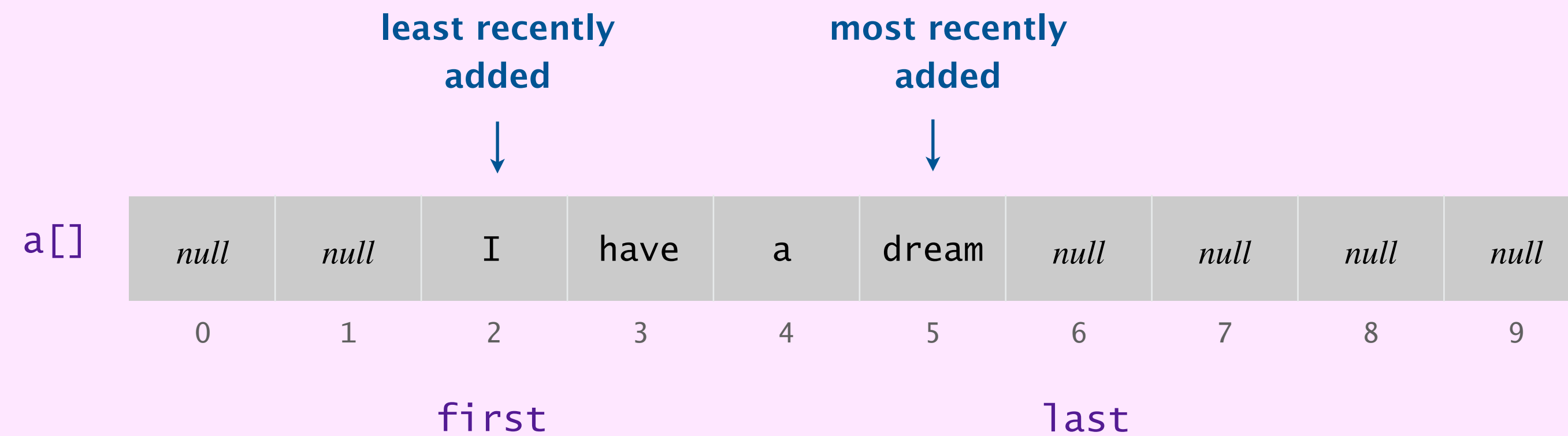
Goal. Implement a **queue** using a **fixed-capacity array** so that all operations take $\Theta(1)$ time.



Fixed-capacity queue: array implementation demo



Goal. Implement a **queue** using a **fixed-capacity array** so that all operations take $\Theta(1)$ time.



Fixed-capacity queue: array implementation

```
public class FixedCapacityQueueOfStrings {
    private String[] a;
    private int first = 0;
    private int last = 0;

    public FixedCapacityQueueOfStrings(int capacity) {
        a = new String[capacity];
    }

    public void enqueue(String item) {
        a[last] = item;
        last++;
        if (last == a.length) last = 0;
    }

    public String dequeue() {
        first++;
        if (first == a.length) first = 0;
        return a[first];
    }
}
```

circular wraparound



<https://algs4.cs.princeton.edu>

1.3 STACKS AND QUEUES I

- ▶ *APIs*
- ▶ *array implementations*
- ▶ *resizable arrays*
- ▶ *amortized analysis*
- ▶ *generics*



How to grow and shrink the array length?

- A.** Increase by 1 before each *push*;
decrease by 1 after each *pop*.
- B.** Increase by $2 \times$ in *push* when array becomes full;
decrease by $2 \times$ in *pop* when array becomes 50% full.
- C.** *Either A or B.*
- D.** *Neither A nor B.*

Stack: resizable-array implementation

Problem. Requiring client to provide maximum capacity does not implement API!

Q. How to **grow** and **shrink** the array automatically? *← referred to as a {resizable, dynamic, extendable} array*

Naive approach.

- Push: increase length of array $a[]$ by 1.
- Pop: decrease length of array $a[]$ by 1.

Too expensive.

- Need to copy all items to a new array, for each push/pop.

• Array accesses to add item k : $1 + 2(k - 1)$ *← to copy $k-1$ elements from old array to new array (ignoring cost to create new array)*

• Array accesses to add first n items: $n + \underbrace{(2 + 4 + 6 + \dots + 2(n - 1))}_{\substack{\text{array resizing to lengths} \\ 2, 3, 4, \dots, n}} \sim n^2.$ *↑ infeasible for large n*

Challenge. Ensure that array resizing happens infrequently.

Stack: resizable-array implementation

Q. How to **grow** the array?

A. If array is full, create a new array of **twice** the length, and copy items.

“geometric expansion”

```
public class ResizableArrayStackOfStrings {
    private String[] a;
    private int n = 0;

    public ResizableArrayStackOfStrings() {
        a = new String[1];
    }

    public void push(String item) {
        if (n == a.length) resize(2 * a.length);
        a[n++] = item;
    }

    private void resize(int capacity) {
        String[] copy = new String[capacity];
        for (int i = 0; i < n; i++)
            copy[i] = a[i];
        a = copy;
    }
}
```

*if the array is full,
double its length*

*helper method
(to resize the array)*

Stack: resizable-array implementation

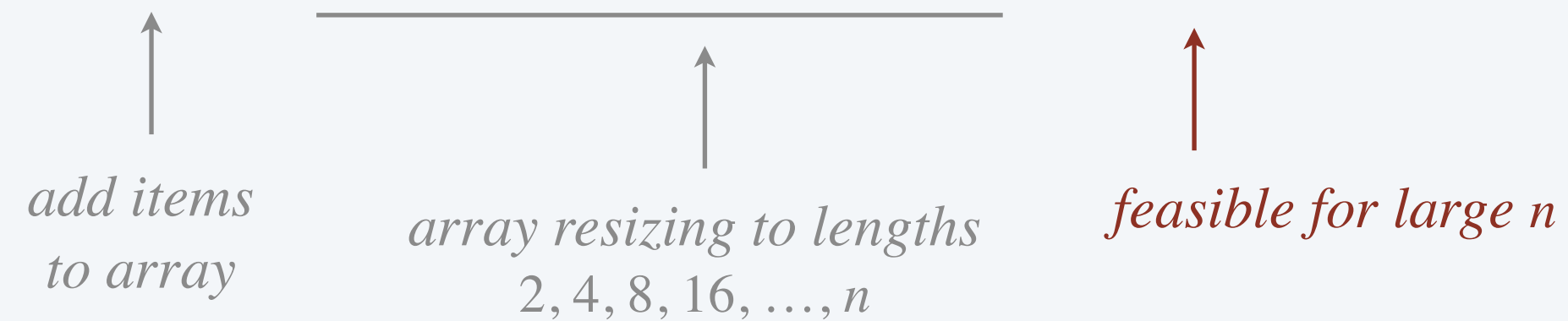
Q. How to **grow** the array?

A. If array is full, create a new array of **twice** the length, and copy items.

“geometric expansion”

Cost is reasonable.

- Still need to copy all items to a new array but, now, that happens infrequently.
- Array accesses to add first $n = 2^i$ items: $n + (2 + 4 + 8 + 16 + \dots + n) \sim 3n$.



Q. Can I use a growth factor other than $\alpha = 2$?

A. Yes. Classic time-space tradeoff.

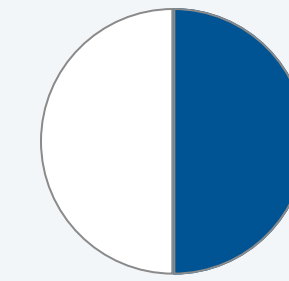
language	data type	α
<i>Java</i>	ArrayList	1.5
<i>C++</i>	vector	1.5
<i>Python</i>	list	1.125
⋮	⋮	⋮

Stack: resizable-array implementation

Q. How to **shrink** the array?

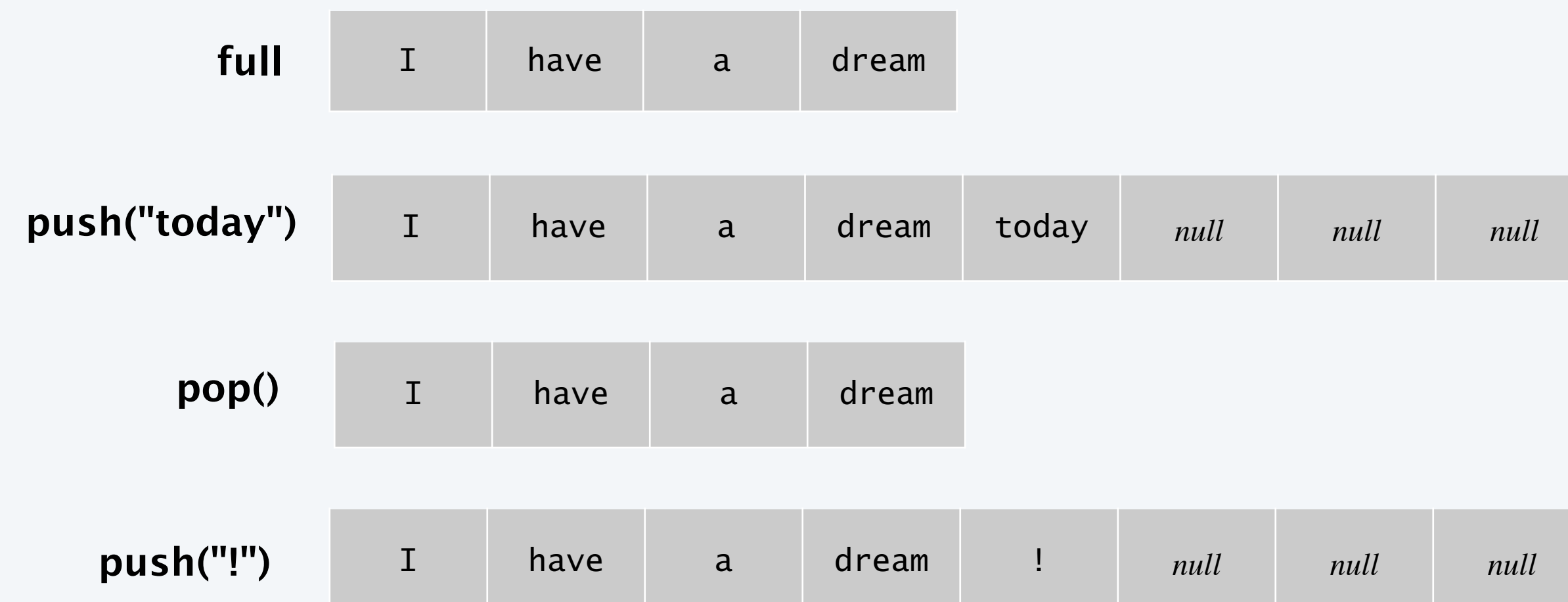
First try.

- Push: double length of array `a[]` when array is full.
- Pop: halve length of array `a[]` when array is **one-half full**.



Too expensive for some sequences of operations.

- Push $n = 2^i$ items to make array full; then, alternate n push and pop operations.
- Each alternating operation triggers an array resizing and takes $\Theta(n)$ time.

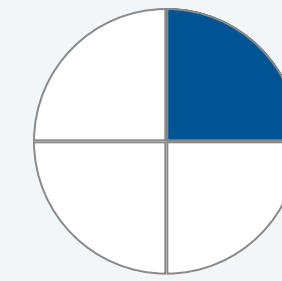


Stack: resizable-array implementation

Q. How to **shrink** the array?

Efficient solution.

- Push: double length of array `a[]` when array is full.
- Pop: halve length of array `a[]` when array is **one-quarter full**.



```
public String pop() {  
    String item = a[--n];  
    a[n] = null;  
    if (n > 0 && n == a.length/4)  
        resize(a.length/2);  
    return item;  
}
```

*if the array is
one-quarter full,
halve its length*

*so, on average, each of the m
operation takes $\Theta(1)$ time*



Proposition. Starting from an empty stack, any sequence of m push/pop operations takes $\Theta(m)$ time.

Intuition. After array resizes to length n , at least $\Theta(n)$ push/pop operations before next array resizing.

Stack resizable-array: memory usage

Proposition. A `ResizableArrayStackOfStrings` with n items use between $\sim 8n$ and $\sim 32n$ bytes of memory.

- Always between 25% and 100% full.
- $\sim 8n$ when full. [array length = n]
- $\sim 32n$ when one-quarter full. [array length = $4n$]

```
public class ResizableArrayStackOfStrings {  
    private String[] a; ← 8 bytes × array length  
    private int n = 0;  
  
    ⋮  
}
```

Remark. This counts the memory for the stack itself, including the string references.
[but not the memory for the string objects, which the client allocates]



<https://algs4.cs.princeton.edu>

1.3 STACKS AND QUEUES I

- ▶ *APIs*
- ▶ *array implementations*
- ▶ *resizable arrays*
- ▶ *amortized analysis*
- ▶ *generics*



28	29876	76	19743
55	38576		94875
98476	29846		49483
98236	27650		3928
57654	94874		29
87770	49874		25
48777	69846		
49824	69687		
48765	69684		
99846	93876		
69687	79855		
69684	93876		
93876	87655		
79855	8354		
29847	29876		
69773	39288		
69847			
49484			
79786			
9393			
49876			
49847			
1948			
19			

AMORTIZATION



Worst-case analysis

Worst-case running time. Longest running time for an **individual operation**.

- Gold standard in analysis of algorithms.
 - applies to all inputs (of a given size)
 - provides an ironclad performance guarantee
 - standardizes way to compare different algorithms
- Can be unduly pessimistic.

e.g., when an expensive operation is rare



operation	worst
<i>construct</i>	$\Theta(1)$
<i>push</i>	$\Theta(n)$
<i>pop</i>	$\Theta(n)$

resizable-array stack with n items

```
stack = new ResizableArrayStackOfInts();
for (int i = 0; i < n; i++) {
    stack.push(i);
}
```

takes $\Theta(n)$ time in the worst case, not $\Theta(n^2)$

Amortized analysis

Amortized analysis. Provides a worst-case running time for a **sequence of operations**.

- Let $T(m)$ denote worst-case running time of sequence of m operations.
- **Amortized cost** per operation = $T(m) / m$.
- Provides more robust and realistic analysis.

starting from an empty data structure



Bob Tarjan
(1986 Turing award)

Ex. Starting from an empty stack, any sequence of m push/pop operations takes $\Theta(m)$ time.

operation	worst	amortized
<i>construct</i>	$\Theta(1)$	$\Theta(1)$
<i>push</i>	$\Theta(n)$	$\Theta(1)$
<i>pop</i>	$\Theta(n)$	$\Theta(1)$

constant amortized time

resizable-array stack with n items

```
stack = new ResizableArrayStackOfInts();  
for (int i = 0; i < n; i++) {  
    stack.push(i);  
}
```

takes $\Theta(n)$ time in the worst case, not $\Theta(n^2)$



Suppose that `QuickUnionPathCompressionUF` has the following performance properties.

What is the **worst-case** running time the following code fragment?

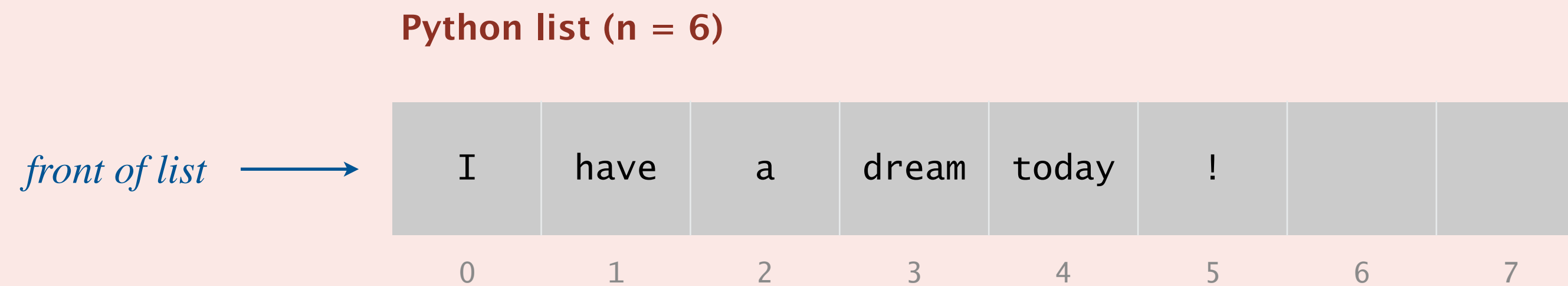
- A. $\Theta(\log n)$
- B. $\Theta(n)$
- C. $\Theta(n \log n)$
- D. $\Theta(n^2)$

```
uf = new QuickUnionPathCompressionUF(n);
for (int i = 0; i < n; i++) {
    if (uf.find(x[i]) != uf.find(y[i]))
        uf.union(x[i], y[i]);
}
StdOut.println(uf.count());
```

operation	worst	amortized
<i>construct</i>	$\Theta(n)$	$\Theta(n)$
<i>union</i>	$\Theta(n)$	$\Theta(\log n)$
<i>find</i>	$\Theta(n)$	$\Theta(\log n)$
<i>count</i>	$\Theta(1)$	$\Theta(1)$



Python implements a `list` as a resizable array (with the first element at index 0). Which of the following can you infer about the **worst-case running times** of various operations, where n is the length of the list?



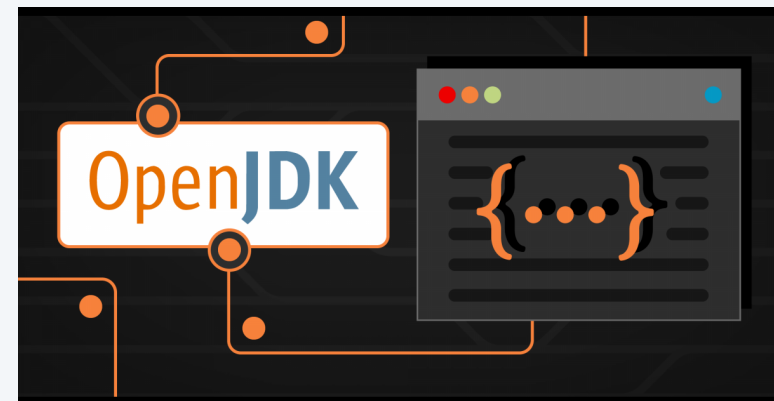
- A. Adding an element to front of list takes $\Theta(1)$ time.
- B. Adding an element to back of list takes $\Theta(1)$ time.
- C. Replacing element i in the list with a new value takes $\Theta(1)$ time.
- D. *None of the above.*

Real-time Java

Java. Rarely provides worst-case performance guarantees.

- Garbage collector: automatically deallocate memory no longer in use.
- Just-in-time compiler: compile bytecode to native machine code at runtime.
- Thread scheduler: determine which thread to execute next.

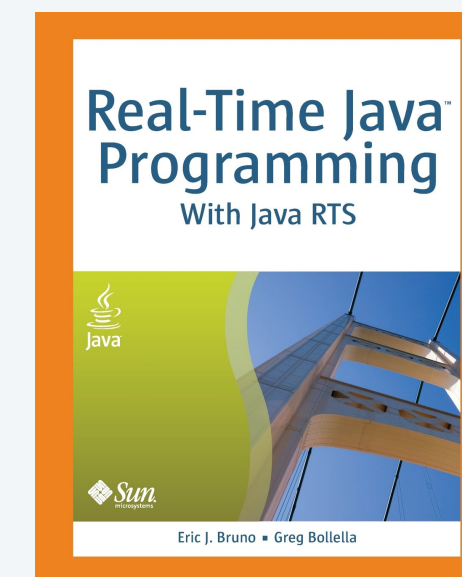
← *operations are expensive,
but run infrequently*



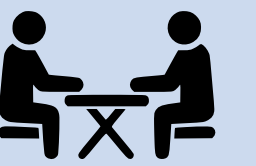
Real-time Java. Provides worst-case performance guarantees.

- Pacemakers.
- Industrial robots.
- Air-traffic control.

← *systems with hard
deadlines*



This course. We ignore such issues in our analysis.



Problem. Implement a queue with two stacks so that:

- $\Theta(1)$ extra memory (besides two stacks).
- Starting from an empty queue, any sequence of m queue operations makes $\Theta(m)$ stack operations.

← *amortized analysis*
(worst case bound on sequence of operations)

Applications.

- Job interview.
- Implement an **immutable** or **persistent** queue.
- Implement a queue in a purely **functional programming language**.



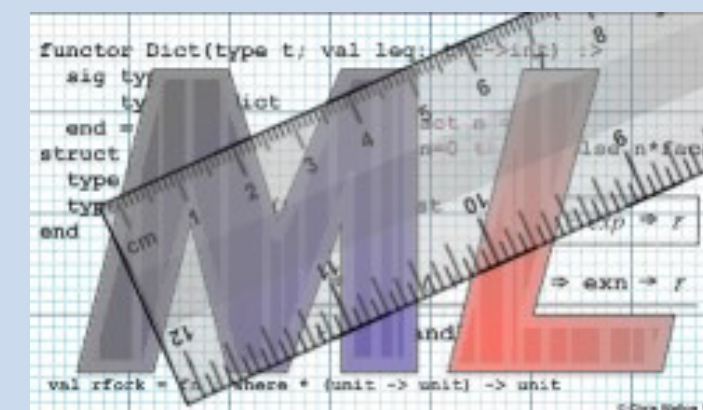
Haskell



Lisp



OCaml





<https://algs4.cs.princeton.edu>

1.3 STACKS AND QUEUES

- ▶ *APIs*
- ▶ *array implementations*
- ▶ *resizable arrays*
- ▶ *amortized analysis*
- ▶ ***generics***

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfApples`, `StackOfOranges`, ...

Solution in Java: generics.

Guiding principle: prefer compile-time errors to run-time errors.

type argument
(use to specify type and invoke constructor)

```
Stack<Apple> stack = new Stack<Apple>();  
Apple apple = new Apple();  
stack.push(apple);  
Orange orange = new Orange();  
stack.push(orange); ← compile-time error  
...
```



Generic stack: array implementation

The way it should be.

```
public class FixedCapacityStackOfStrings {
    private String[] a;
    private int n = 0;

    public Fixed...OfStrings(int capacity)
    { a = new String[capacity]; }

    public boolean isEmpty()
    { return n == 0; }

    public void push(String item)
    { a[n++] = item; }

    public String pop()
    { return a[--n]; }

}
```

stack of strings (fixed-length array)

```
public class FixedCapacityStack<Item> {
    private Item[] a;
    private int n = 0;

    public FixedCapacityStack(int capacity)
    { a = new Item[capacity]; }

    public boolean isEmpty()
    { return n == 0; }

    public void push(Item item)
    { a[n++] = item; }

    public Item pop()
    { return a[--n]; }

}
```

generic stack (fixed-length array) ???

@#\$!* generic array creation
not allowed in Java*

Generic stack: array implementation

The way it should be.

```
public class FixedCapacityStackOfStrings {
    private String[] a;
    private int n = 0;

    public Fixed...OfStrings(int capacity)
    { a = new String[capacity]; }

    public boolean isEmpty()
    { return n == 0; }

    public void push(String item)
    { a[n++] = item; }

    public String pop()
    { return a[--n]; }

}
```

stack of strings (fixed-length array)

```
public class FixedCapacityStack<Item> {
    private Item[] a;
    private int n = 0;

    public FixedCapacityStack(int capacity)
    { a = (Item[]) new Object[capacity]; }

    public boolean isEmpty()
    { return n == 0; }

    public void push(Item item)
    { a[n++] = item; }

    public Item pop()
    { return a[--n]; }

}
```

← *the ugly cast*

generic stack (fixed-length array)

Unchecked cast

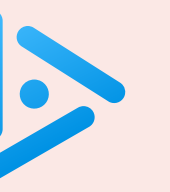
```
~/cos226/queues> javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
    a = (Item[]) new Object[capacity];
           ^
required: Item[]
found:    Object[]
where Item is a type-variable:
  Item extends Object declared in class FixedCapacityStack
1 warning
```

Q. Why does Java require a cast (or reflection)?

Short answer. Backward compatibility.

Long answer. Need to learn about **type erasure** and **covariant arrays**.





How to declare and initialize an empty stack of integers in Java?

- A. `Stack stack = new Stack<>();`
- B. `Stack<int> stack = new Stack();`
- C. `Stack<int> stack = new Stack<int>();`
- D. *None of the above.*

Generic data types: autoboxing and unboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has an associated “**wrapper**” reference type.
- Ex: `Integer` is wrapper type associated with `int`.

Autoboxing. Automatic cast from primitive type to wrapper type.

Unboxing. Automatic cast from wrapper type to primitive type.

primitive	wrapper
<code>int</code>	<code>Integer</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>

```
Stack<Integer> stack = new Stack<Integer>();  
stack.push(17);           // stack.push(Integer.valueOf(17));  
int x = stack.pop();     // int x = stack.pop().intValue();
```

Bottom line. Client code can use generic stack with **any** data type.

Caveat. Performance overhead for primitive types.

Stacks and queues summary

Fundamental data types.

- Value: **collection** of objects.
- Operations: **add**, **remove**, iterate, size, test if empty.

next lecture ←

Stack. [LIFO] Remove the item most recently added.

Queue. [FIFO] Remove the item least recently added.



Efficient implementations.

- Resizable array.
- Singly linked list. ← *next lecture*

Credits

image	source	license
<i>Red Back Button</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Menu with Undo</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Dragon Book</i>	<u>Aho, Sethi, Ullman</u>	
<i>TigerPrint</i>	<u>Auburn</u>	
<i>Network Packets</i>	<u>H3C</u>	
<i>Just Buffering</i>	<u>Red Bubble</u>	
<i>People Standing in Line</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Stack Overflow Logo</i>	<u>VectorLogoZone</u>	
<i>Amortization</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Best/Worst Case</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Bob Tarjan</i>	<u>Heidelberg Laureate</u>	

Credits

image	source	license
<i>Recycling Bin</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>OpenJDK Compiler</i>	<u>RedHat</u>	
<i>Spools of Thread</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Real-Time Java Book</i>	<u>Bruno and Bollella</u>	
<i>Stack of Apples</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Stack of Fruit</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Bad Design</i>	<u>Medium</u>	
<i>Stack of Sweaters</i>	<u>Adobe Stock</u>	<u>Education License</u>

Stacks and queues for adults

