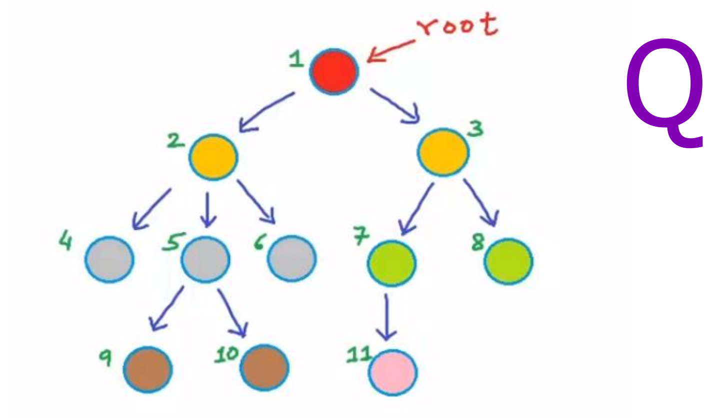
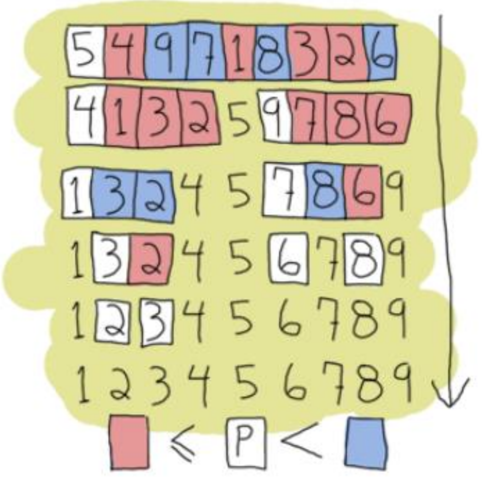
ECE 250 Data Structures & Algorithms



Queues & Stacks



Ziqiang Patrick Huang

Electrical and Computer Engineering

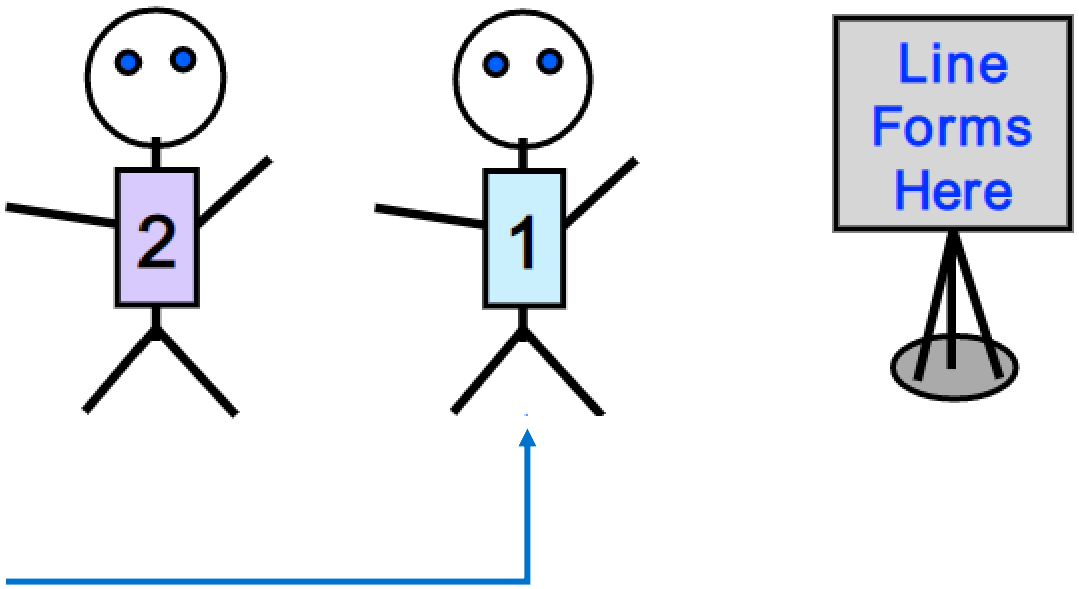
University of Waterloo

WATERLOO | ENGINEERING

Queues: FIFO

- Queues are conceptually like waiting in line
  - First in first out behavior

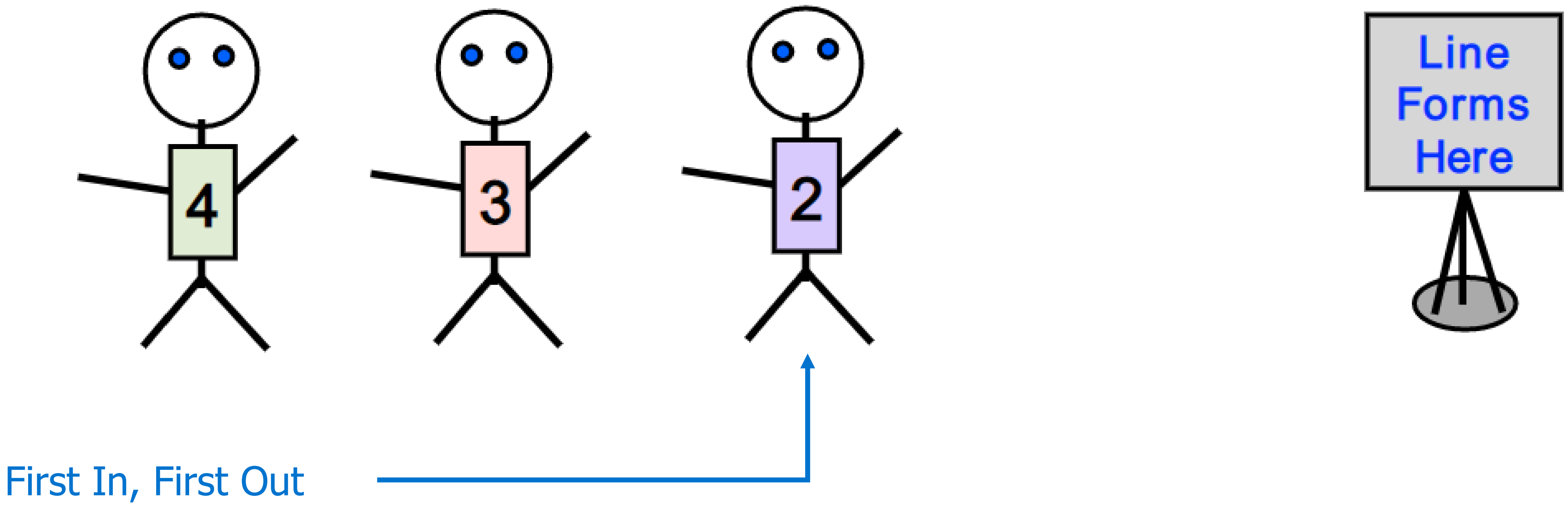
Next!



First In, First Out

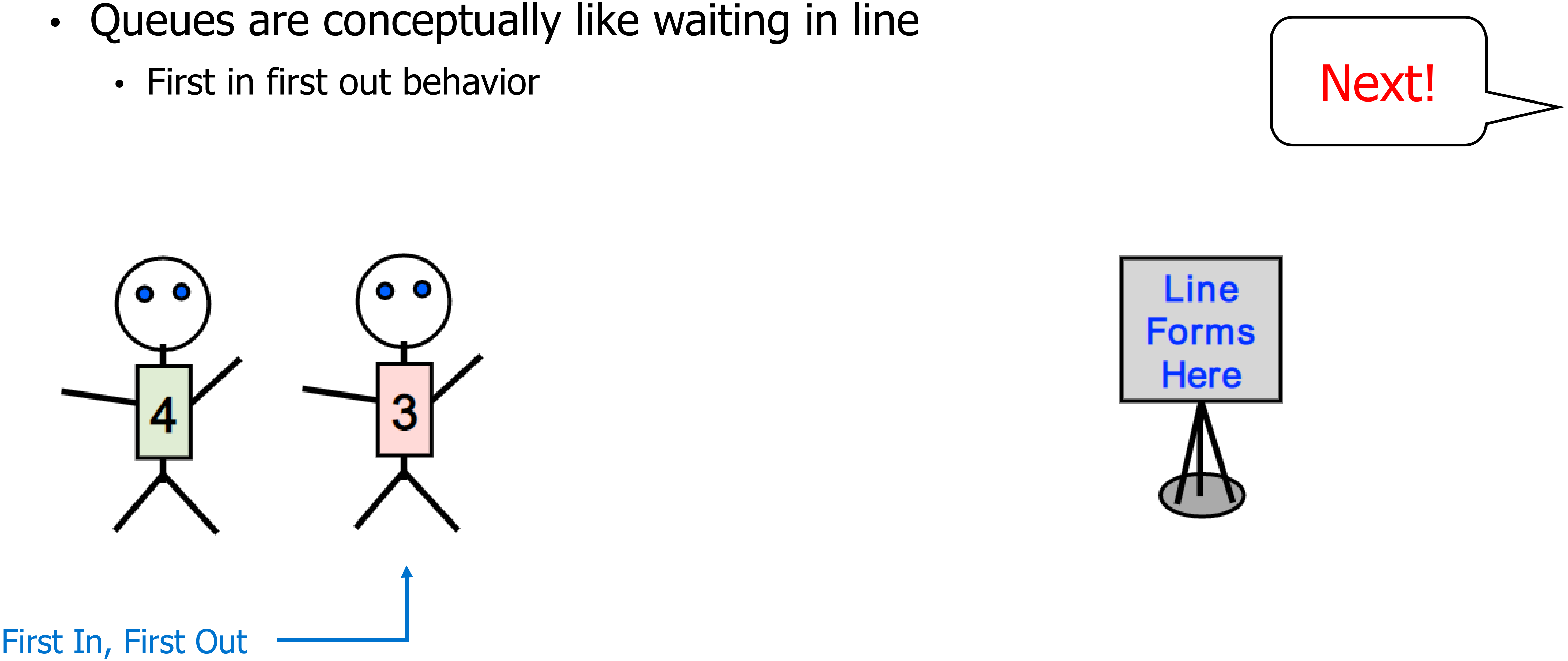
Queues: FIFO

- Queues are conceptually like waiting in line
  - First in first out behavior



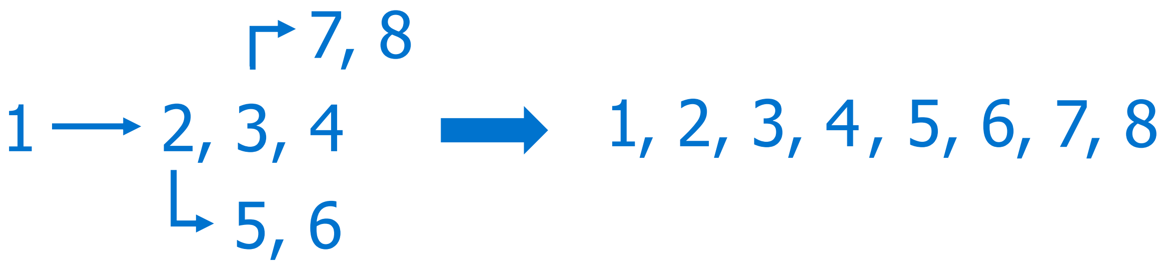
Queues: FIFO

- Queues are conceptually like waiting in line
  - First in first out behavior



Queues in Programming

- “Waiting in line” happens all the time in programs
  - Networked programs: in coming connection requests
    - Queued by OS until application can handle them
  - Things to do at certain time may be queued
    - Might be “priority queue” (later)
  - Some algorithms use queues
    - Compute “more things to do”
    - Put them in a queue
    - Take “next thing to do” from the queue



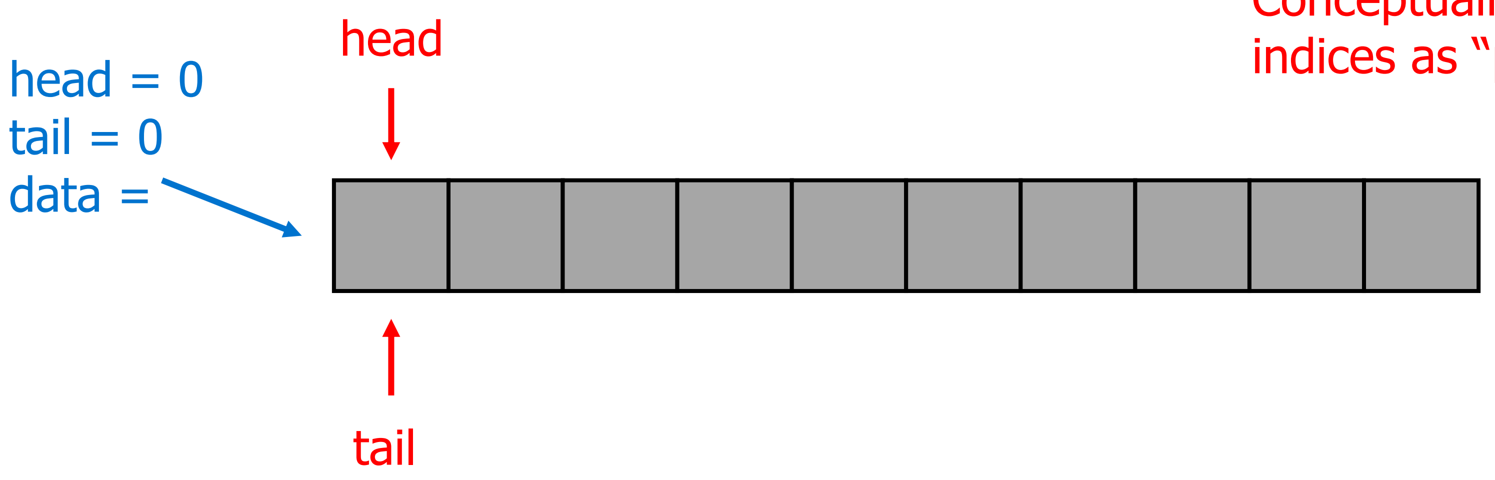
7, 8

1 2, 3, 4

5, 6

1, 2, 3, 4, 5, 6, 7, 8

Queue Implementation with Array

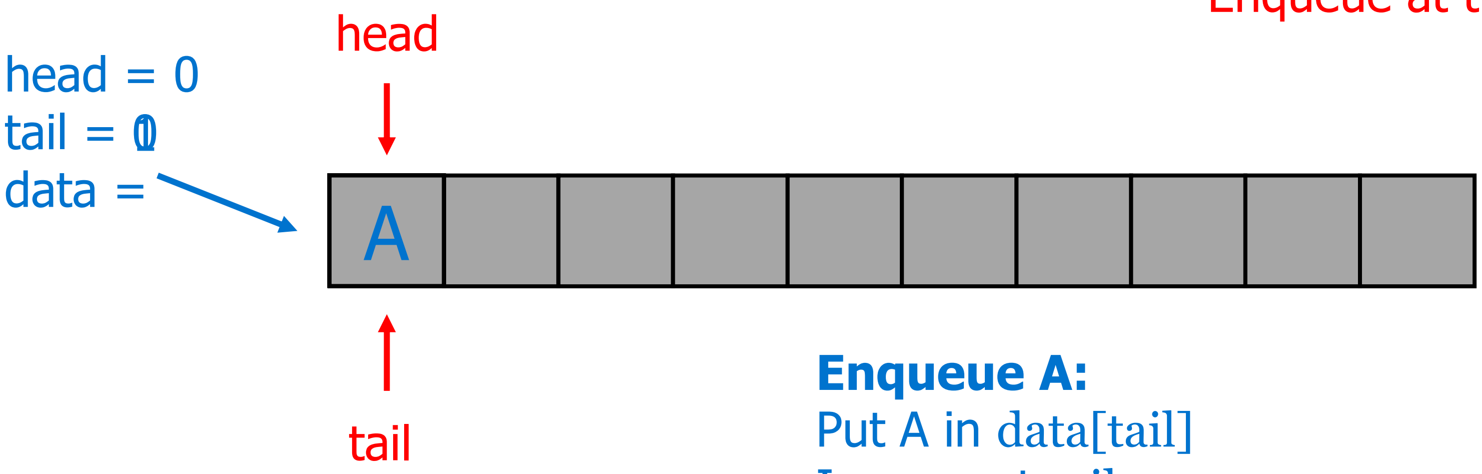


Conceptually, think of head and tail indices as “pointing” at the queue

- We could implement Queue with an array
    - Particularly good if “fixed size” queue

Queue Implementation with Array

Enqueue at the tail, and increase the index.

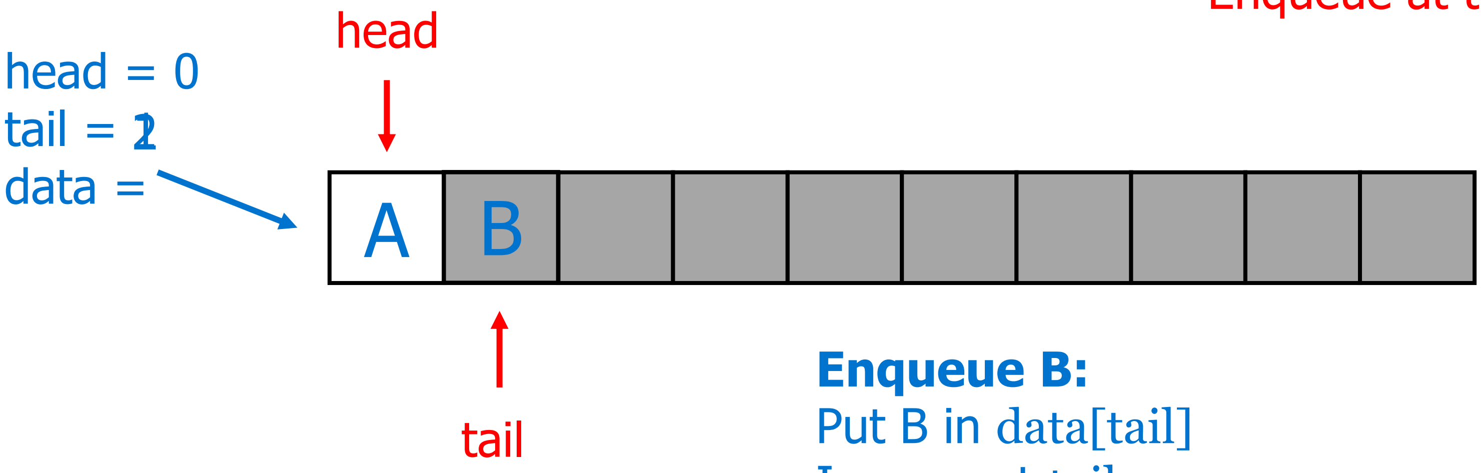


Enqueue A:
Put A in data[tail]
Increment tail

- We could implement Queue with an array
  - Particularly good if “fixed size” queue

Queue Implementation with Array

Enqueue at the tail, and increase the index.

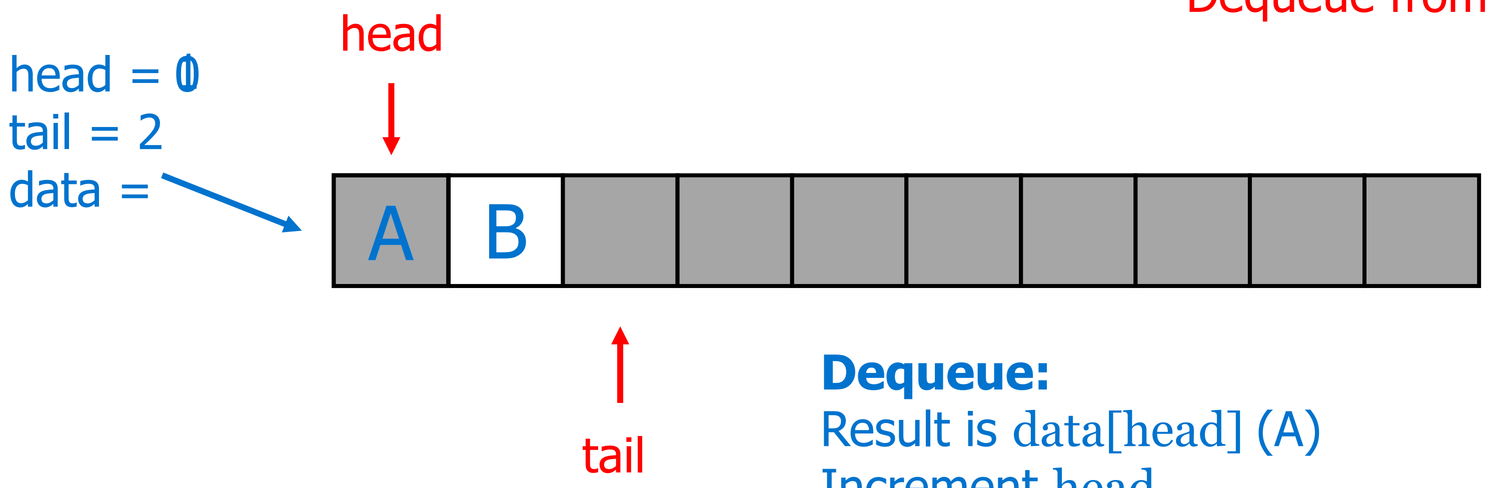


Enqueue B:
Put B in data[tail]
Increment tail

- We could implement Queue with an array
  - Particularly good if “fixed size” queue

Queue Implementation with Array

Dequeue from the head, and increment head index

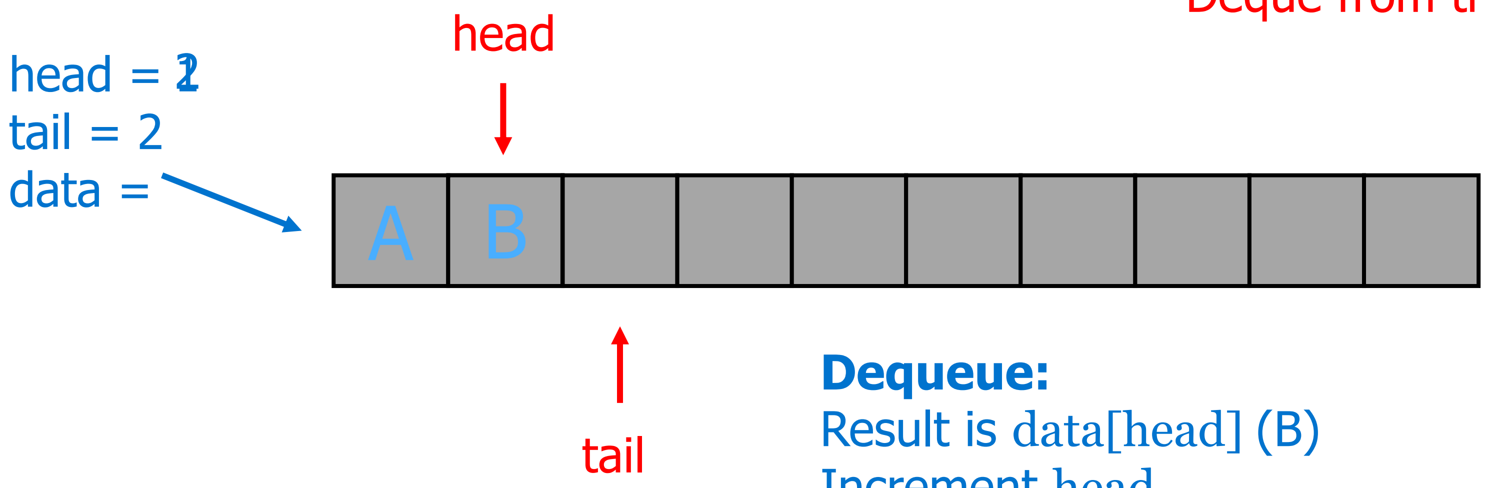


Dequeue:
Result is data[head] (A)
Increment head

- We could implement Queue with an array
  - Particularly good if “fixed size” queue

Queue Implementation with Array

Deque from the head, and increment head index



head

head = 1 2

tail = 2

data =

A

B

tail

Dequeue:

Result is data[head] (B)

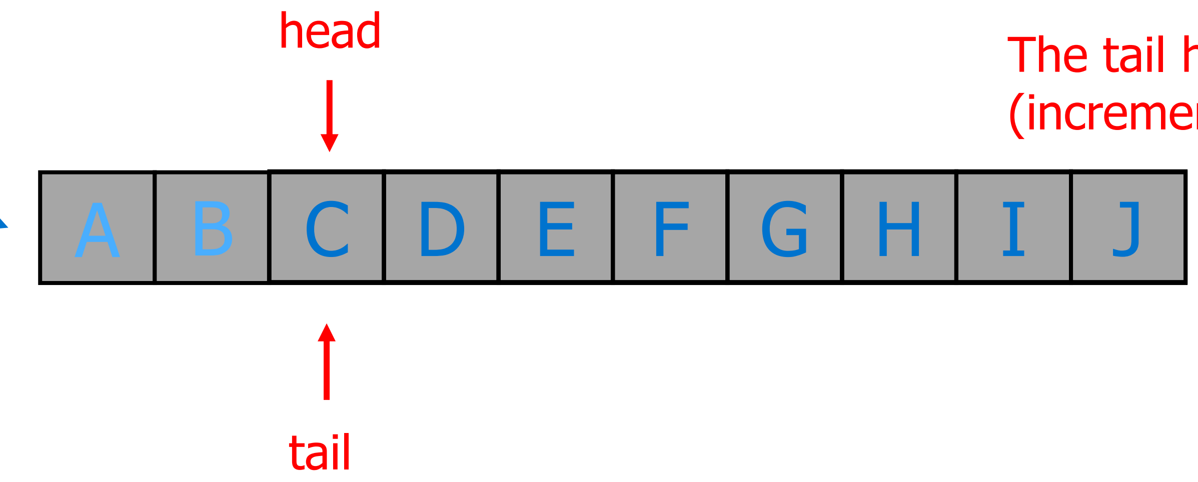
Increment head

- We could implement Queue with an array
  - Particularly good if "fixed size" queue

Queue Implementation with Array

Suppose we enqueue more things (C, D, …, J)

head = 2
tail = 92
data =



A	B	C	D	E	F	G	H	I	J

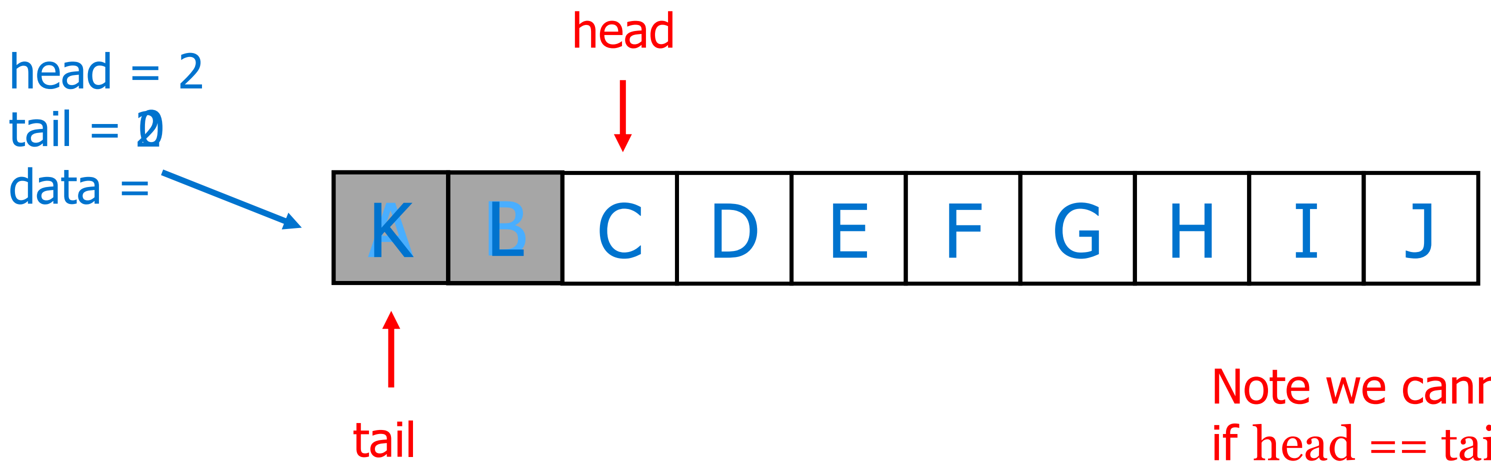
The tail has to wrap back around to 0 (increment, mod the array size)

e.g., (9 + 1) % 10 = 0

(a + 1) % b → increment a and "wrap it around" back to 0 when it reaches b

- We could implement Queue with an array
  - Particularly good if "fixed size" queue

Queue Implementation with Array

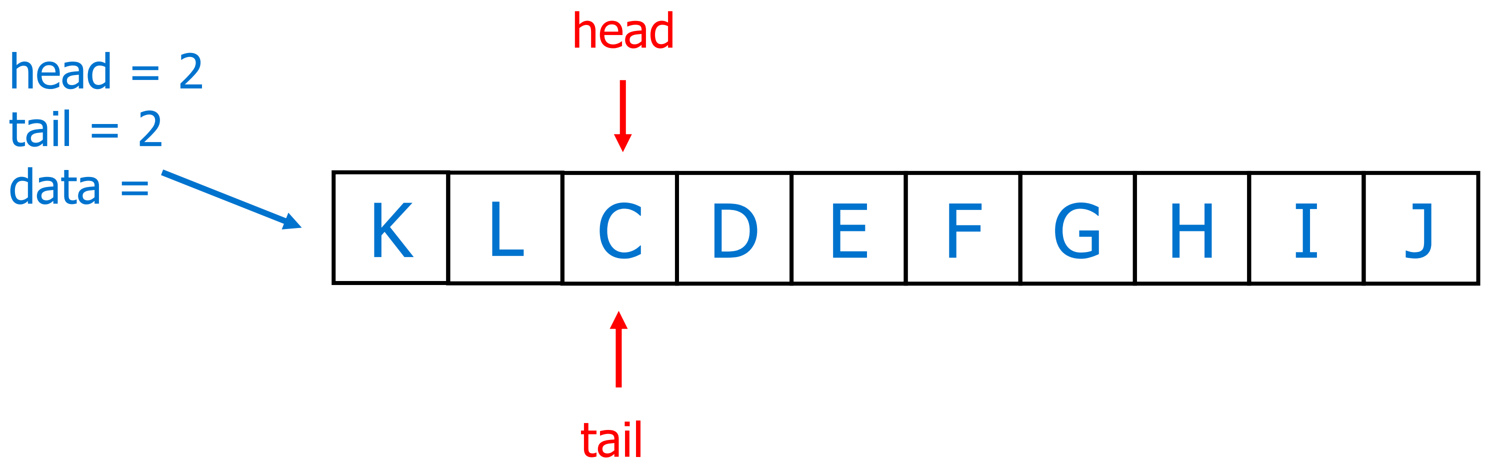


Adding two more items: K, L results in a full queue

Note we cannot check for a full queue by testing if $\mathrm{head} == \mathrm{tail}$: also happens for empty queue. Easiest: store count of items

- We could implement Queue with an array
  - Particularly good if “fixed size” queue

Queue Implementation with Array

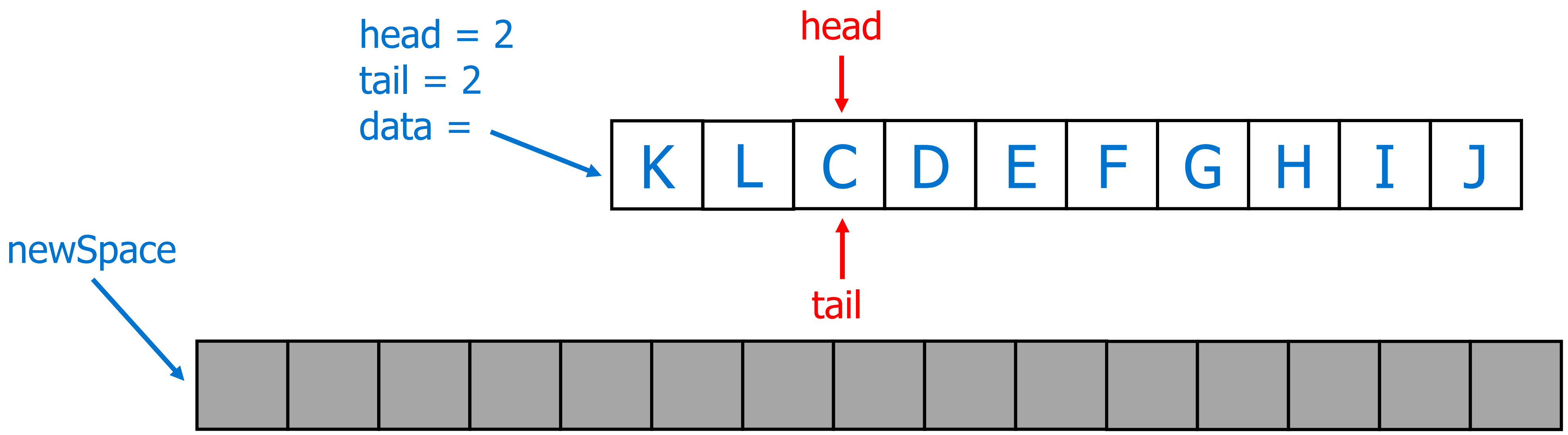


head = 2
tail = 2
data =

K	L	C	D	E	F	G	H	I	J

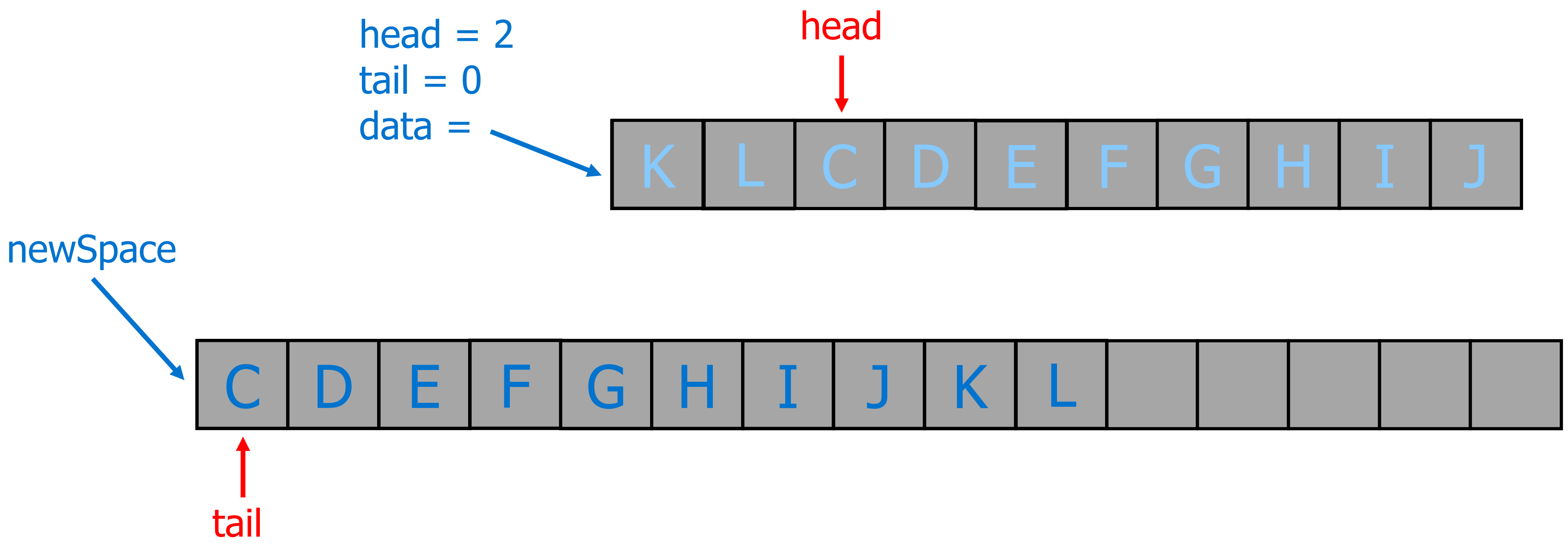
- What if we try to enqueue something when queue is full?
  - Option 1: Its an error (fixed size queue)
    - Provide isFull() in interface, design code which uses to prevent
  - Option 2: Make the queue larger

Queue Implementation with Array



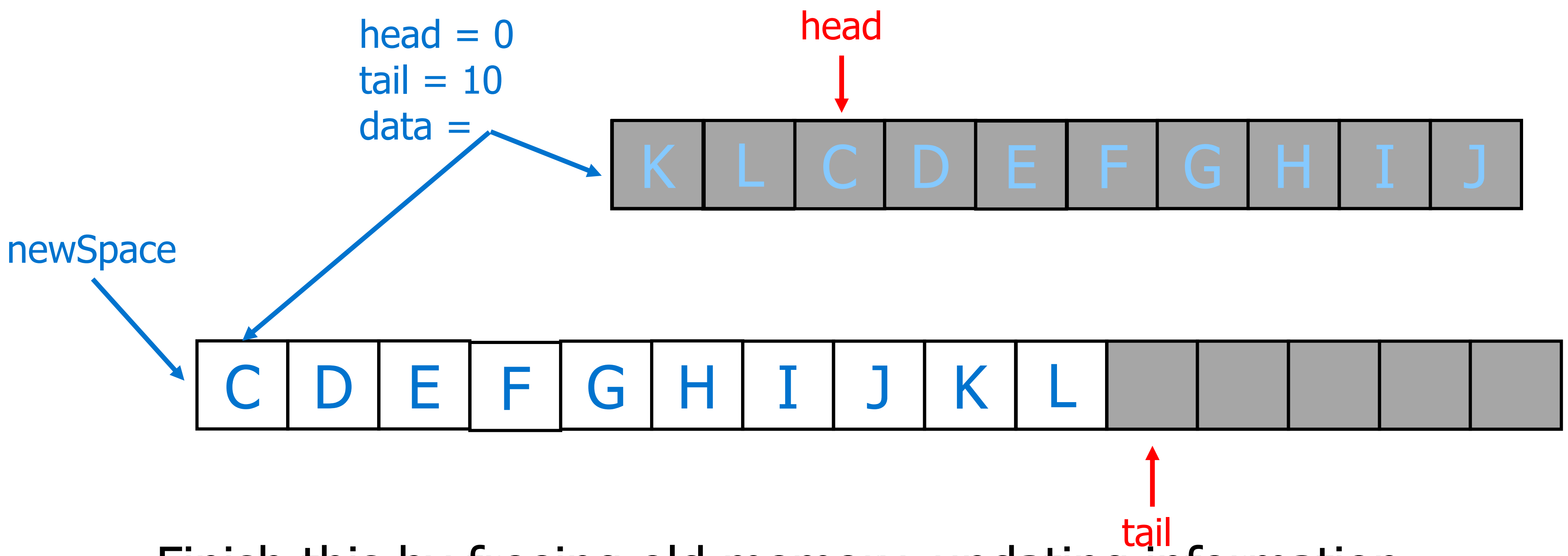
- Growing our queue
  - Need more space (allocate it)
  - Conceptually tail (place to add) moves to the start of new space

Queue Implementation with Array



- Growing our queue
  - Need more space (allocate it)
  - Conceptually tail (place to add) moves to the start of new space
  - Copy the data …

Queue Implementation with Array

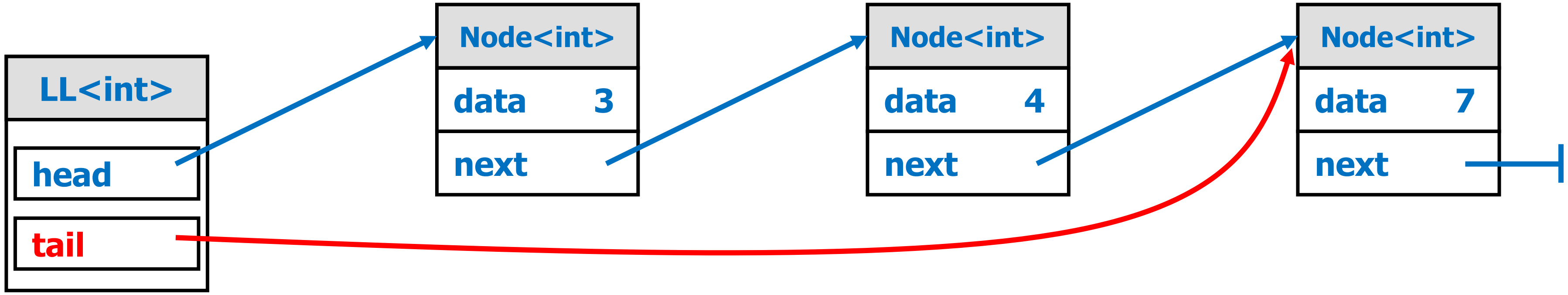


- Finish this by freeing old memory, updating information
  - head = 0 (now conceptually in new space, at the start)
  - delete[] data
  - Data = newSpace

Queue Growth

- Growing the queue: O(n) operation
  - We need to copy N elements from the old to the new
  - Do this occasionally? Fine
  - Do it frequently? Performance will be slow
    - N adds will have O(N^2) performance
  - Can’t make worst case better, but can make average case better
    - Amortize cost of copying over more adds between copies
    - Double size of array each time it needs to grow
      - Now we know we get N adds before we do N work
      - N/N = 1, maintain O(1) average time addition
- Good rule for growing array-based structures in general:
  - Double the size each time you must grow
  - Amortize your copying costs

Queue Implementation with Linked List



addToFront O(1)
removeFromFront O(1)

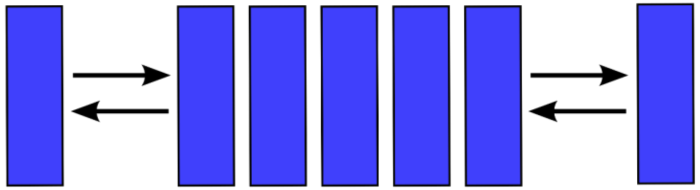
addToBack O(1)
removeFromBack O(n)

- We could also implement Queue with a Linked List
  - enqueue at one end, dequeue at the other
  - enqueue: addToBack (easy with tail pointer)
  - dequeue: removeFromFront

Deques

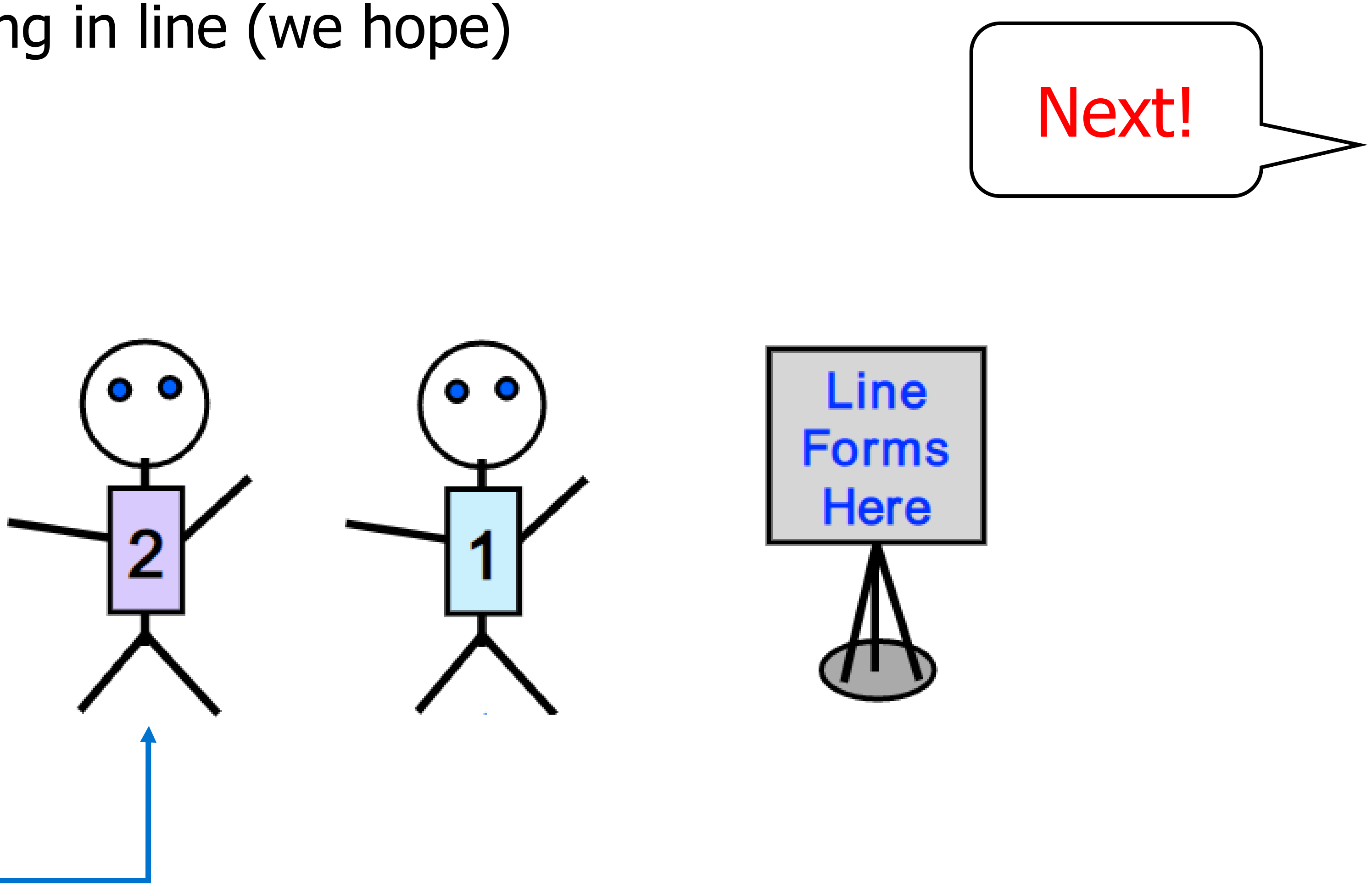
- Sometimes want the ability to add/remove from both ends of the queue
  - Work stealing scheduling algorithm
    - One thread can “steal” work from another thread’s task queue
    - Access own task queues at tail, thieves steal from the head

- Deque (pronounced like “deck”)
  - Short for “double ended queue”
  - No “FIFO” or “LIFO” behavior
  - Can add and remove from both ends



Stacks: LIFO

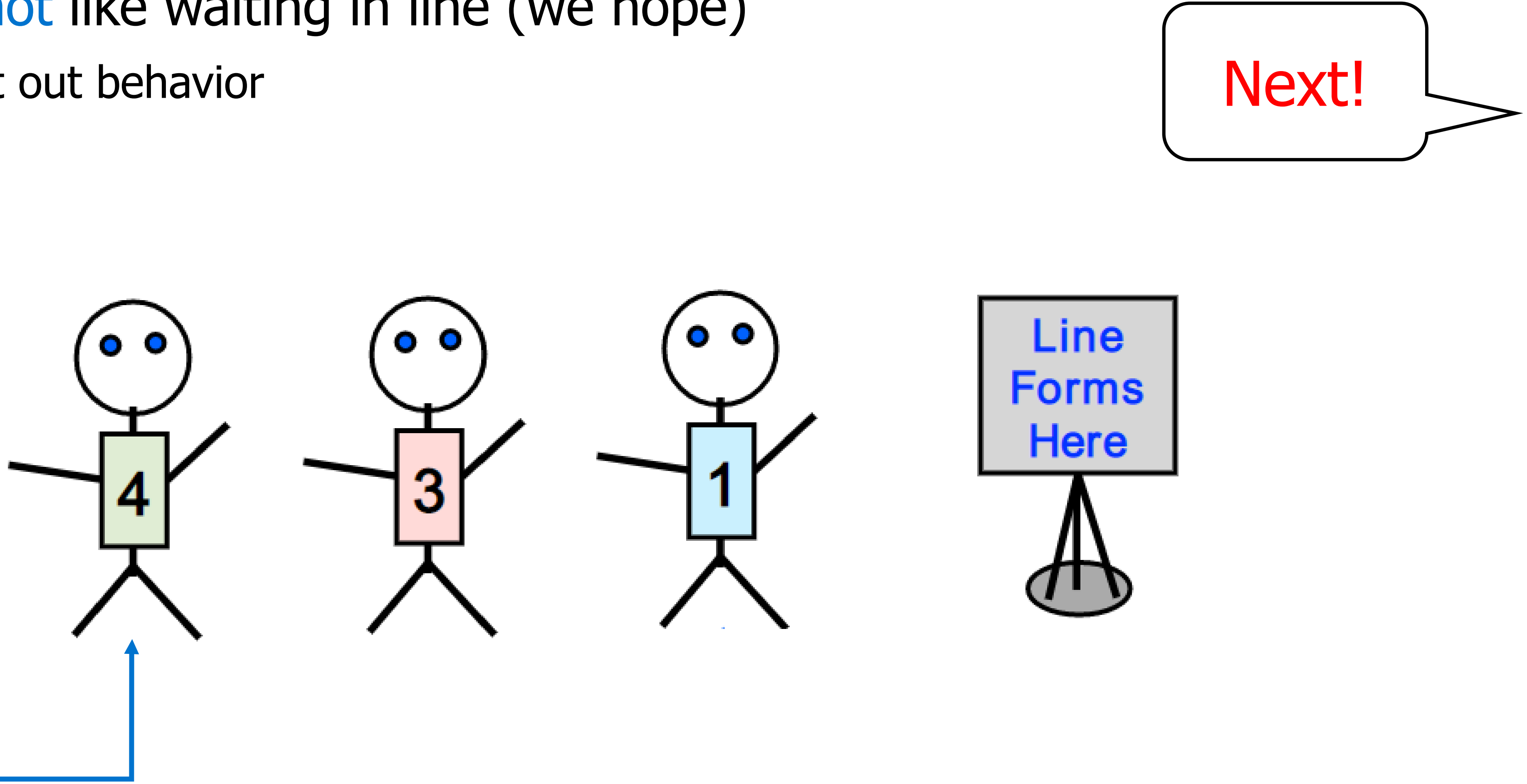
- Stacks are not like waiting in line (we hope)
  - Last in first out behavior



Last In, First Out

Stacks: LIFO

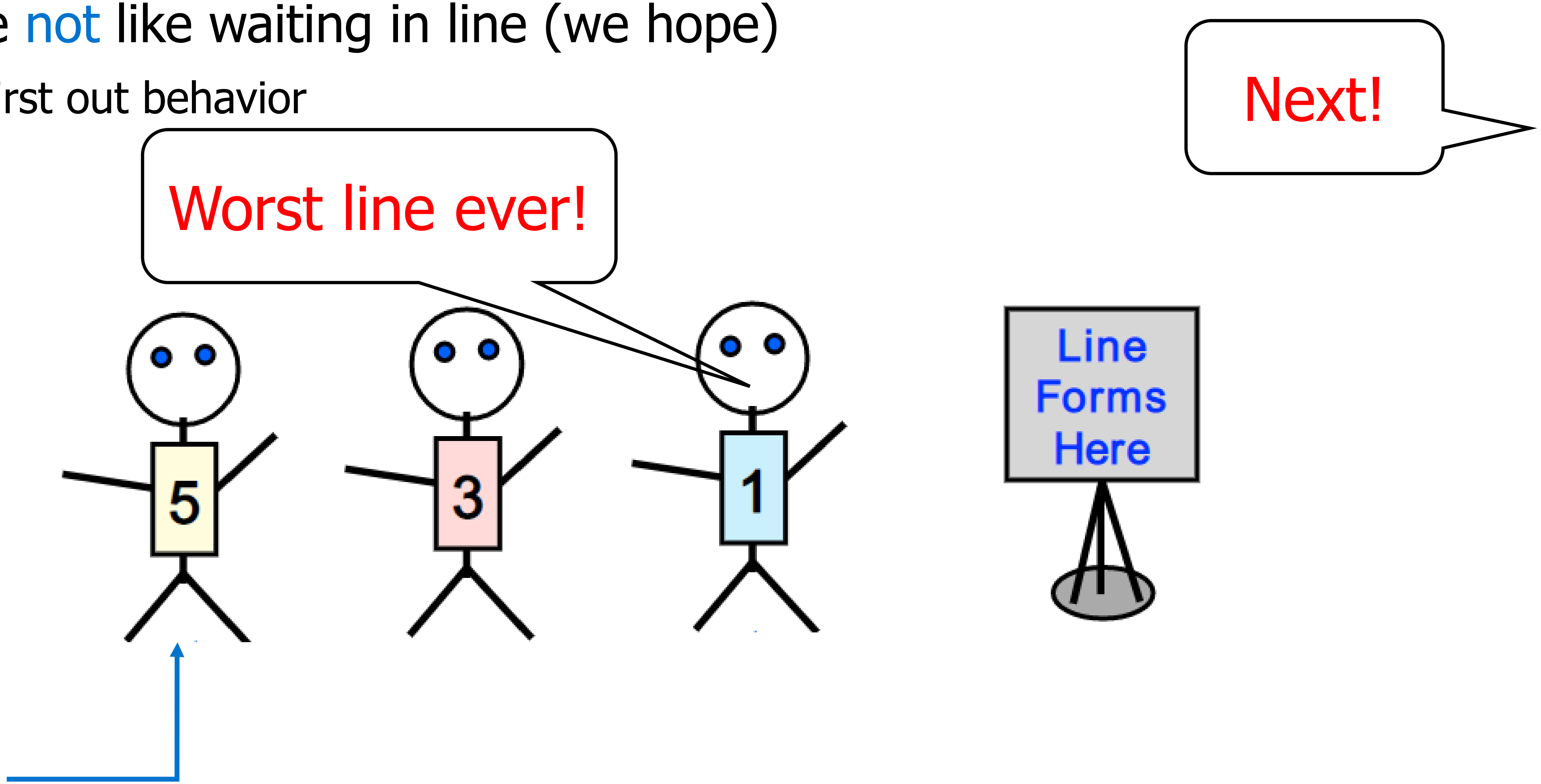
- Stacks are not like waiting in line (we hope)
  - Last in first out behavior



Last In, First Out

Stacks: LIFO

- Stacks are not like waiting in line (we hope)
  - Last in first out behavior



Last In, First Out

Stacks in Programming

- Stacks are not useful for "waiting in line", but …
  - Have already seen one important stack
    - Call stack: tracks local variables, parameters, return locations
    - Implicitly part of language, does not need explicit ADT
  - Useful for reversing things
    - Push each thing on the stack in order
    - Popping the stack gives elements in reverse order
    - Don't overcomplicate simple reversals though!
  - Useful for nested matching
    - Example: nested parenthesis (4 + (3 * 2) - (8 * 9) + 1)
    - Also, html, xml, etc …
  - More generally, useful for parsing
    - Analyzing an input string to determine meaning

Matched Tags Example: HTML

- HTML: balanced tags (e.g., `<B>` for bold, `</B>` ends bold)

```
<html>
<head>
<title>Example Page</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
<\i> just bold</b>
</body>
<html>
```

Use a stack, it starts out empty ….

Start reading the input (just strings)

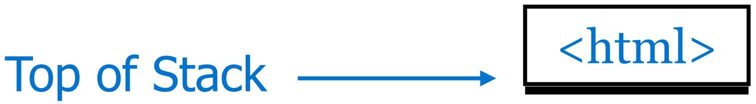
Top of Stack →

Matched Tags Example: HTML

- HTML: balanced tags (e.g., `<B>` for bold, `</B>` ends bold)

```
<html>
<head>
<title>Example Page</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
<\i> just bold</b>
</body>
<html>
```

Encounter an open tag,
push it on the stack



Top of Stack

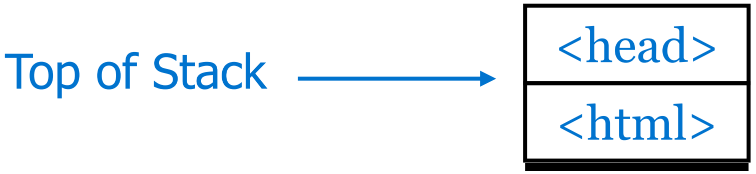
<html>

Matched Tags Example: HTML

- HTML: balanced tags (e.g., `<B>` for bold, `</B>` ends bold)

```
<html>
<head>
<title>Example Page</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
<\i> just bold</b>
</body>
<html>
```

Encounter an open tag, push it on the stack



Top of Stack

<head>

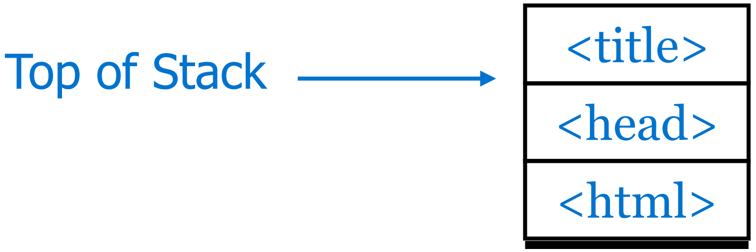
<html>

Matched Tags Example: HTML

- HTML: balanced tags (e.g., `<B>` for bold, `</B>` ends bold)

```
<html>
<head>
<title>Example Page</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
<\i> just bold</b>
</body>
<html>
```

Encounter an open tag, push it on the stack

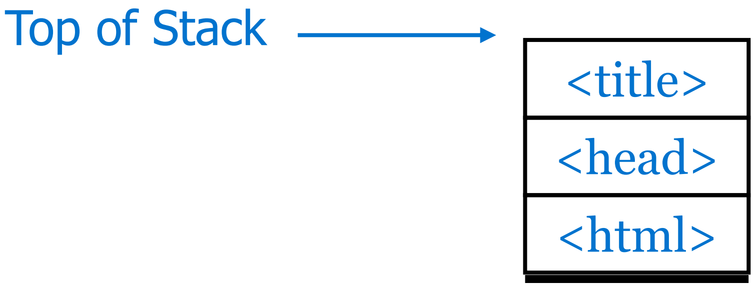


Matched Tags Example: HTML

- HTML: balanced tags (e.g., `<B>` for bold, `</B>` ends bold)

```
<html>
<head>
<title>Example Page</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
<\i> just bold</b>
</body>
<html>
```

All the tags on the stack apply to any (non-tag) we encounter



Top of Stack

<title>

<head>

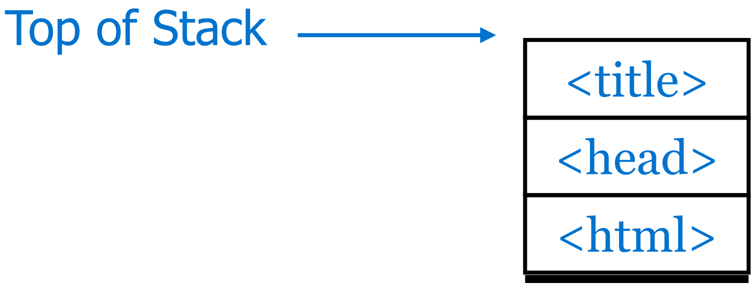
<html>

Matched Tags Example: HTML

- HTML: balanced tags (e.g., `<B>` for bold, `</B>` ends bold)

```
<html>
<head>
<title>Example Page</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
<\i> just bold</b>
</body>
<html>
```

Encounter a close tag:
pop the stack (remove its top)

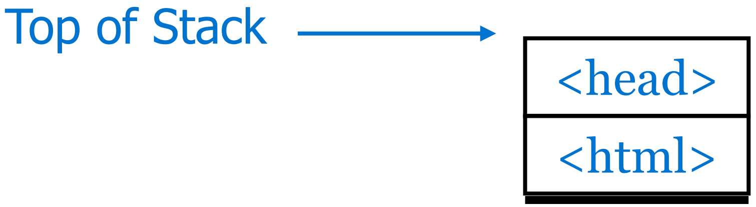


Matched Tags Example: HTML

- HTML: balanced tags (e.g., `<B>` for bold, `</B>` ends bold)

```
<html>
<head>
<title>Example Page</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
<\i> just bold</b>
</body>
<html>
```

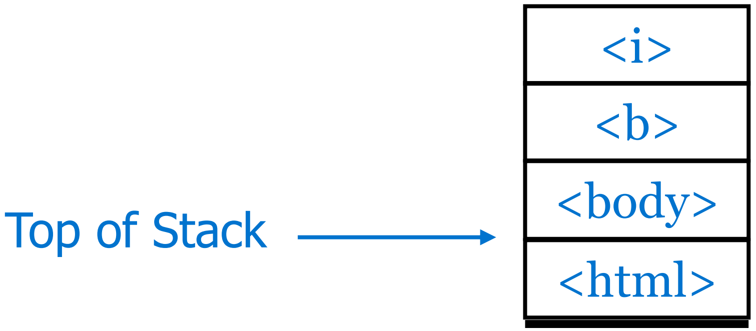
Encounter a close tag:
pop the stack (remove its top)



Matched Tags Example: HTML

- HTML: balanced tags (e.g., `<B>` for bold, `</B>` ends bold)

```
<html>
<head>
<title>Example Page</title>
</head>
<body>
Some text
<b>Some bold text
<i>and bold italics
<\i> just bold</b>
</body>
<html>
```



Top of Stack

<i>

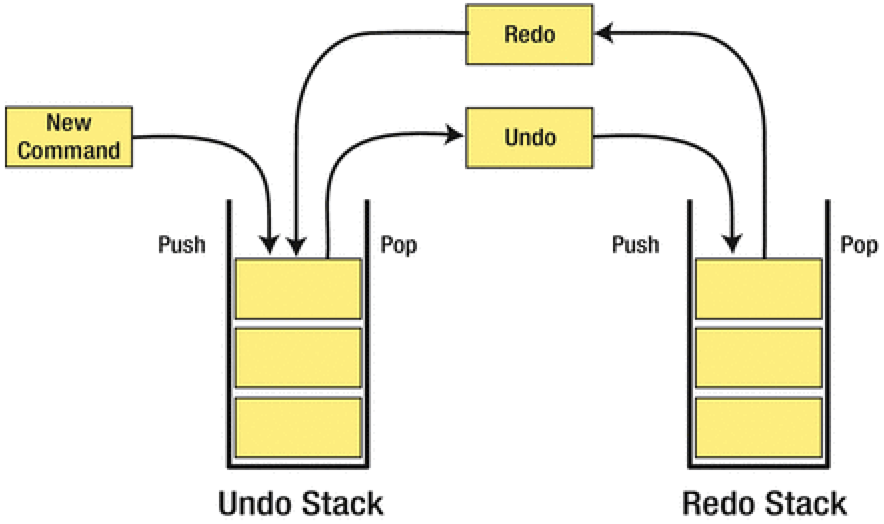
<b>

<body>

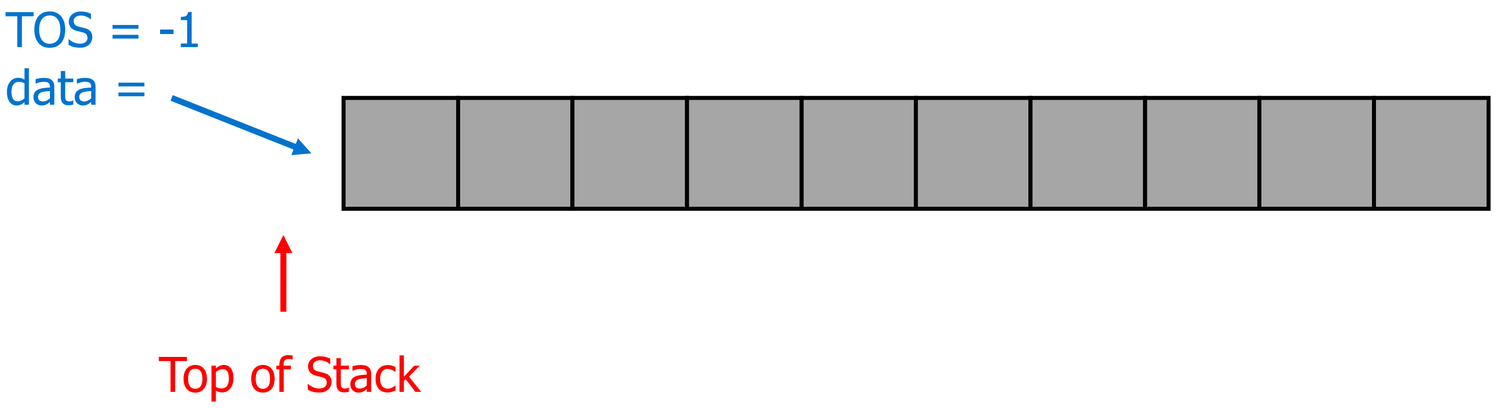
<html>

Stack Example: Undo & Redo

- Many editing tools have “Undo” & “Redo” feature
  - Can be implemented with two stack
  - Push each change (or document state) unto the “Undo Stack”
  - “Undo” pop the top from the “Undo Stack” and push it unto the “Redo Stack”
  - “Redo” pop the top from the “Redo Stack” and push it unto the “Undo Stack”

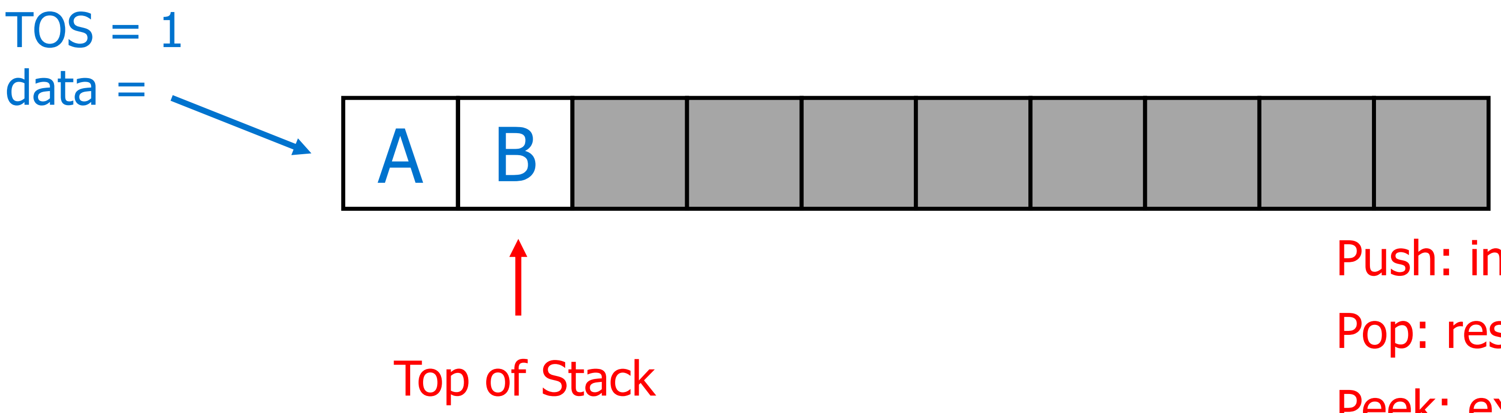


Stack Implementation with Array



- We can also implement a stack with an array
  - Track the "top of the stack with one index ("tos")
    - Last index used (-1 on empty stack)

Stack Implementation with Array



TOS = 1

data =

A

B

Top of Stack

Push: increment TOS, store data there

Pop: result is data[TOS], decrement TOS

Peek: examine data[TOS]

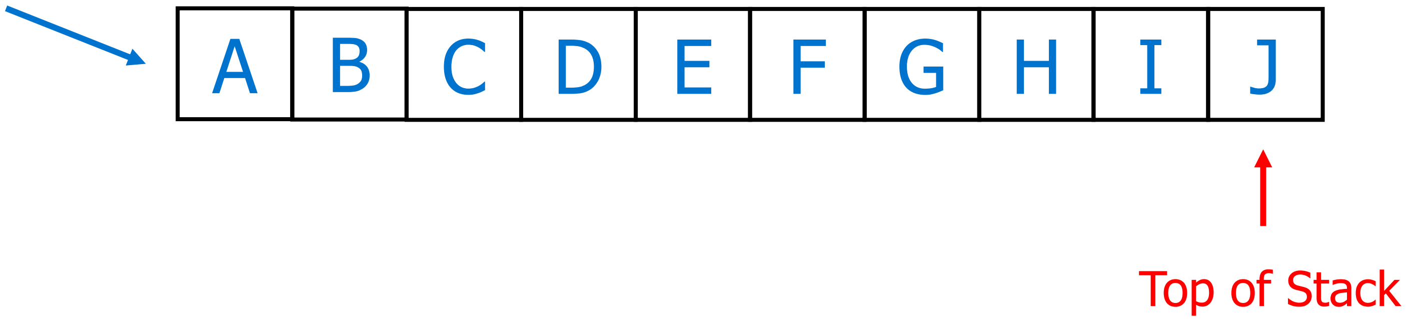
- We can also implement a stack with an array
  - Track the "top of the stack with one index ("tos")
    - Last index used (-1 on empty stack)

Stack Implementation with Array

If we push a bunch of elements … Our stack is full

Similar options to Queue: error or grow

TOS = 9
data =

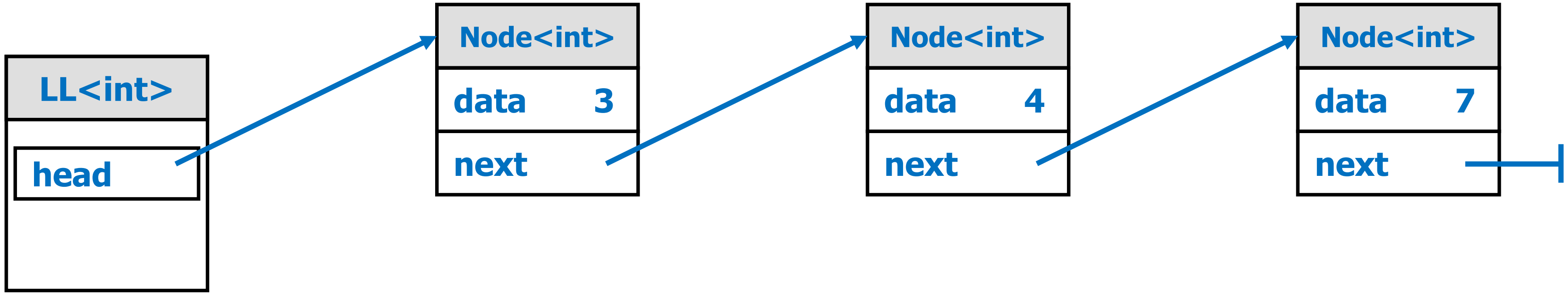


A	B	C	D	E	F	G	H	I	J

Top of Stack

- We can also implement a stack with an array
  - Track the "top of the stack with one index ("tos")
    - Last index used (-1 on empty stack)

Stack Implementation with Linked List



addToFront O(1)
removeFromFront O(1)

- We could also implement stack with a Linked List
  - Push: addToFront
  - Pop: removeFromFront

Queue/Stack Implementations: Array vs Linked List

	Enqueue/Push	Dequeue/Pop	Peek	Resize
Array-Based	O(1)*	O(1)	O(1)	O(n)
LinkedList-Based	O(1)	O(1)	O(1)	O(1)

- Looks like linked list always wins, why bother using array?
  - A bit space overhead
  - Frequently allocating/deallocating nodes
  - Not all O(1) operations are created the same
    - Array access has "spatial locality", can be exploited by caches (you will learn in ECE 222)
    - Linked List? Not so much, nodes can be anywhere

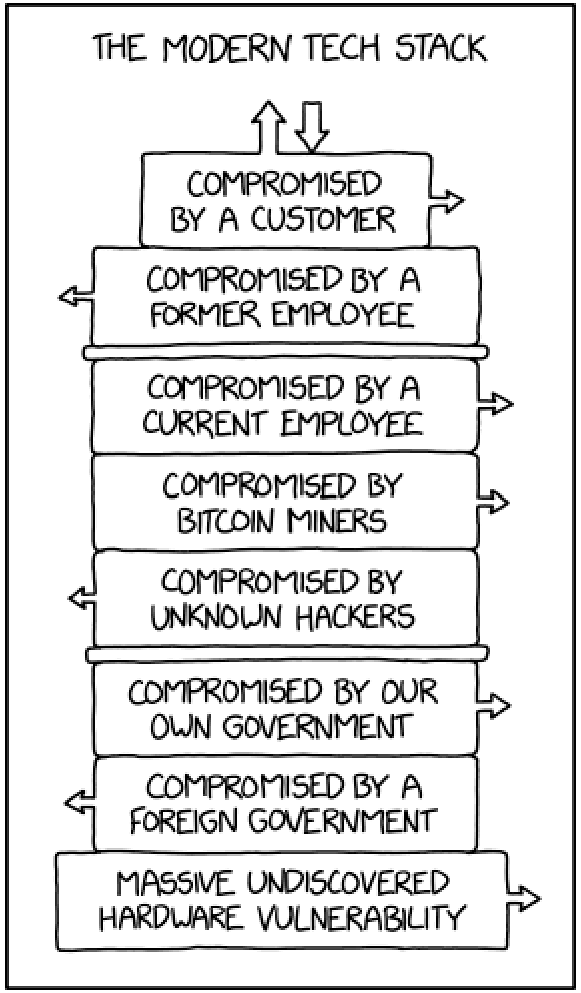
Wrap Up

- In this lecture we talked about
  - More applications of Queues & Stacks
  - Introduced Deque (very briefly)
  - Implementations of Queue/Stack using array and linked list

- Next up
  - Trees & Binary Search Trees

Suggested Complimentary Readings

- Data Structure and Algorithms in C++: Chapter 3.6 – 3.7
- Introduction to Algorithms: Chapter 10.1


THE MODERN TECH STACK
COMPROMISED BY A CUSTOMER
COMPROMISED BY A FORMER EMPLOYEE
COMPROMISED BY A CURRENT EMPLOYEE
COMPROMISED BY BITCOIN MINERS
COMPROMISED BY UNKNOWN HACKERS
COMPROMISED BY OUR OWN GOVERNMENT
COMPROMISED BY A FOREIGN GOVERNMENT
MASSIVE UNDISCOVERED HARDWARE VULNERABILITY

source

Acknowledgement

- This slide builds on the hard work of the following amazing instructors:
  - Andrew Hilton (Duke)
  - Mary Hudachek-Buswell (Gatech)