

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

R u t g e r s U n i v e r s i t y

2. STACKS AND QUEUES

- ▶ *Stacks*
- ▶ *Resizing arrays*
- ▶ *Queues*
- ▶ *Generics*
- ▶ *Iterators (optional)*
- ▶ *Applications*



INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

R u t g e r s U n i v e r s i t y

-
- ▶ **Office Hours *CHANGE***
 - ▶ *When? Tuesday 5:30 – 7PM*
 - ▶ *Where? Murray Hall 115*

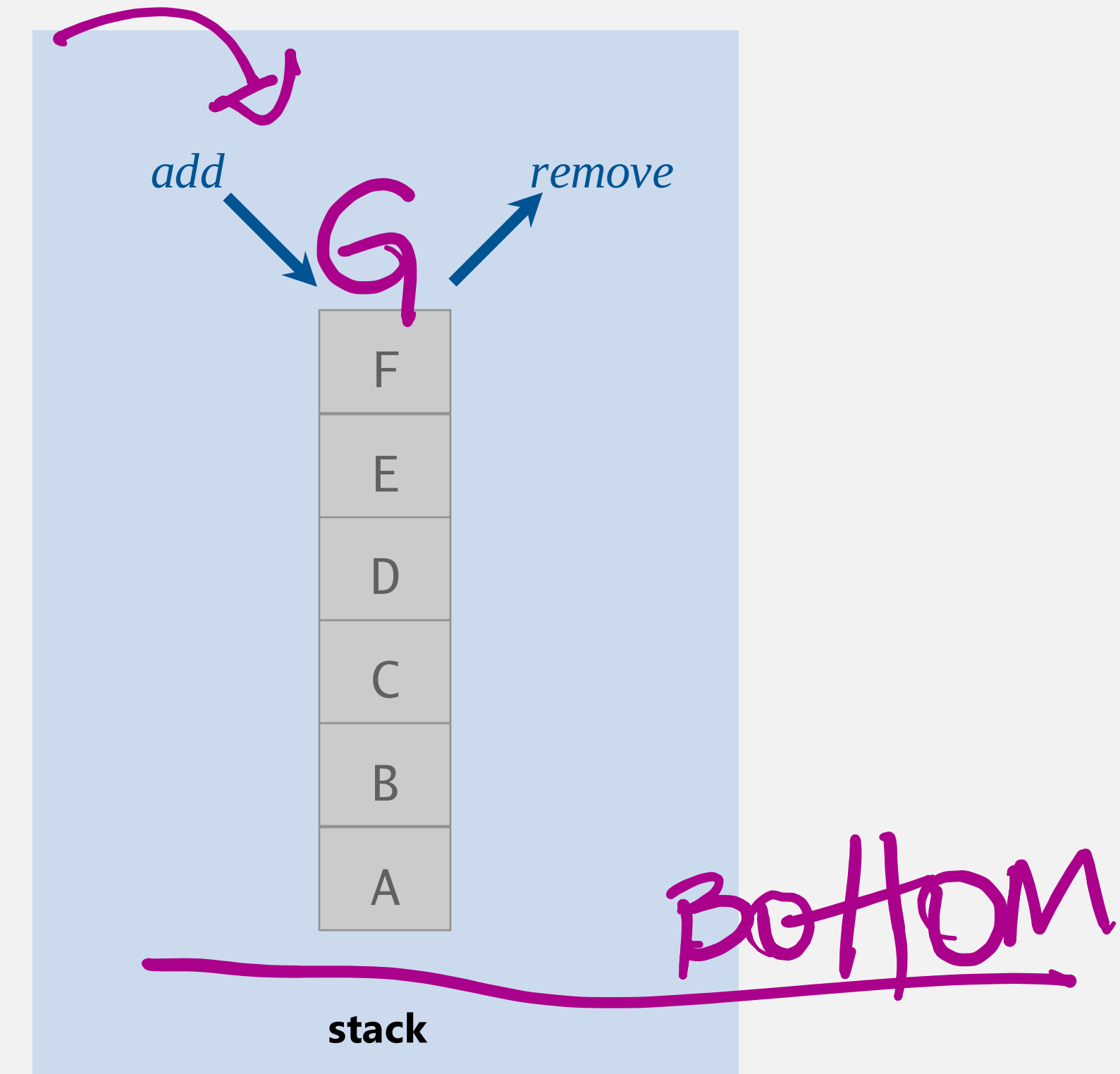
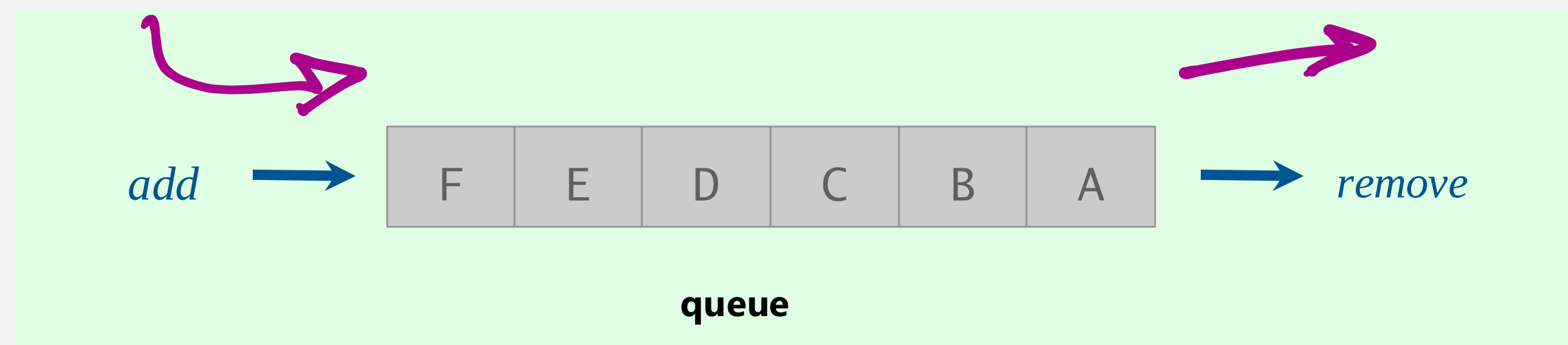
 - ▶ *Exam Conflict?*



Stacks and queues are concepts NOT data structures

Fundamental data types.

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

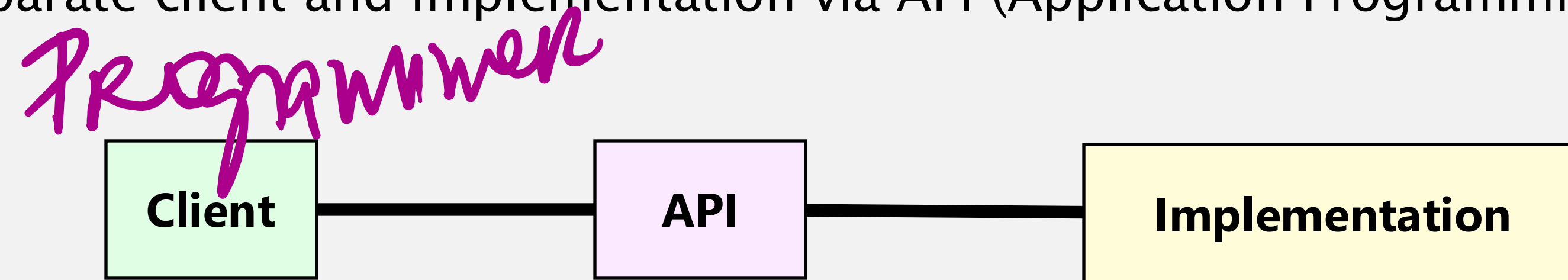


Stack. Examine the item most recently added. ← LIFO = “last in first out”

Queue. Examine the item least recently added. ← FIFO = “first in first out”

Client, implementation, API

Separate client and implementation via API (Application Programming Interface)



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

Client: program that uses the API operations.

Implementation: code that implements the API operations.

Benefits.

- **Design:** create modular, reusable libraries.
- **Performance:** substitute faster implementations.

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

2. STACKS AND QUEUES

- ▶ *Stacks*
- ▶ *Resizing arrays*
- ▶ *Queues*
- ▶ *Generics*
- ▶ *Iterators (optional)*
- ▶ *Applications*



Stack API

Warmup API. Stack of strings data type.

```
public class StackOfStrings
```

const

```
StackOfStrings()
```

create an empty stack

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

add a new string to stack

```
String pop()
```

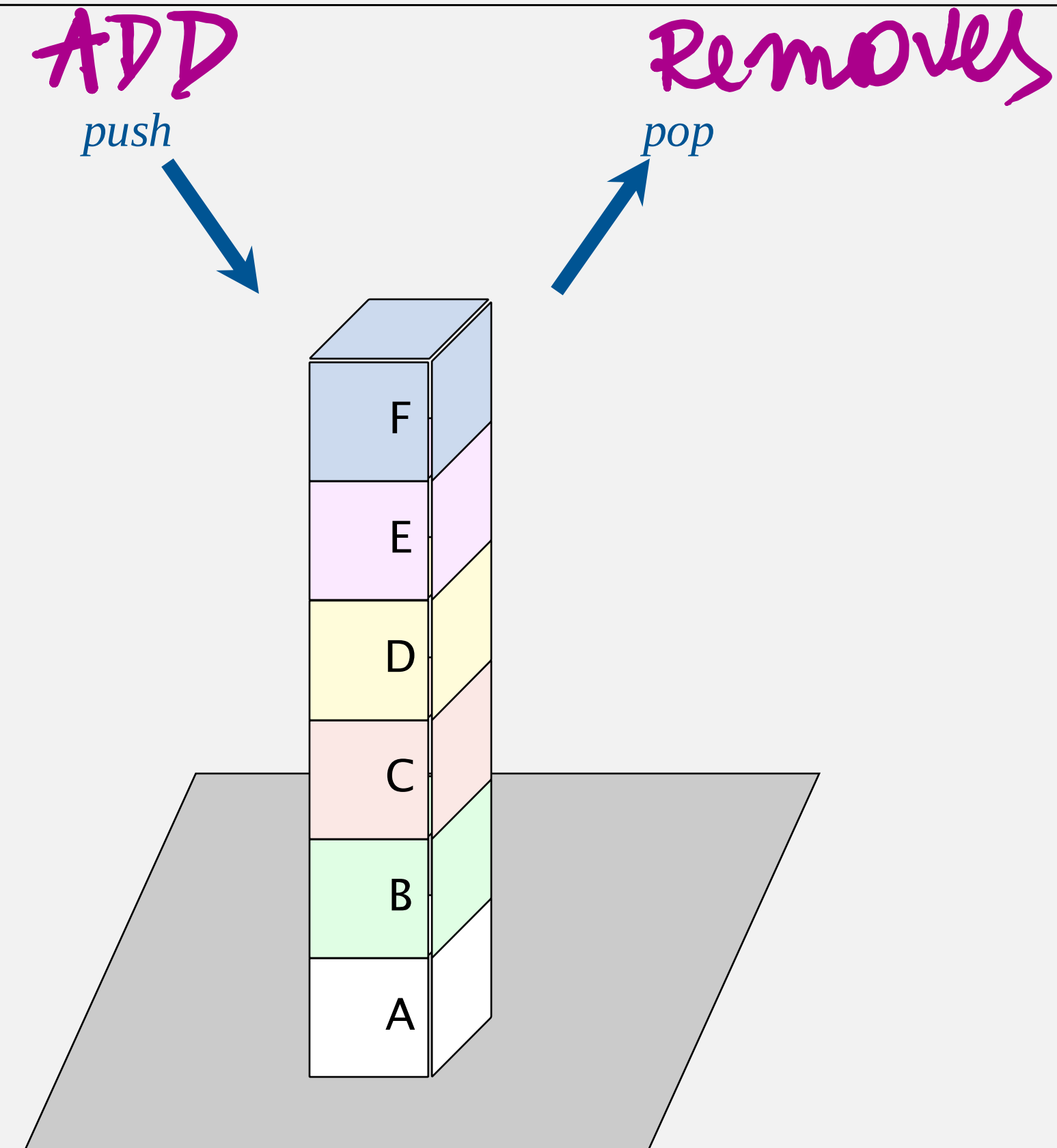
*remove and return the string
most recently added*

```
boolean isEmpty()
```

is the stack empty?

```
int size()
```

number of strings on the stack



Performance goal. Every operation takes $O(1)$ time.

Warmup client. Reverse a stream of strings from standard input.

- Use array $s[]$ to store n items on stack.

ADD push(): where in the array an item should be pushed?

Remove pop(): from where the top item is removed?

Operations:

→ push 11

→ push 43

→ push 89

→ pop

→ push 34



PUSH

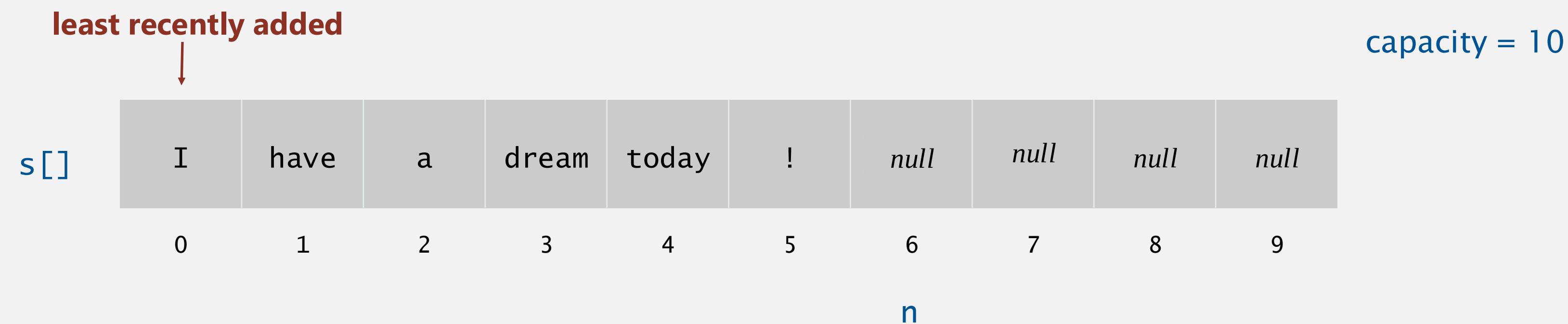
$s[t] = \text{item};$

$t++;$

POP
 $s[t-1]$

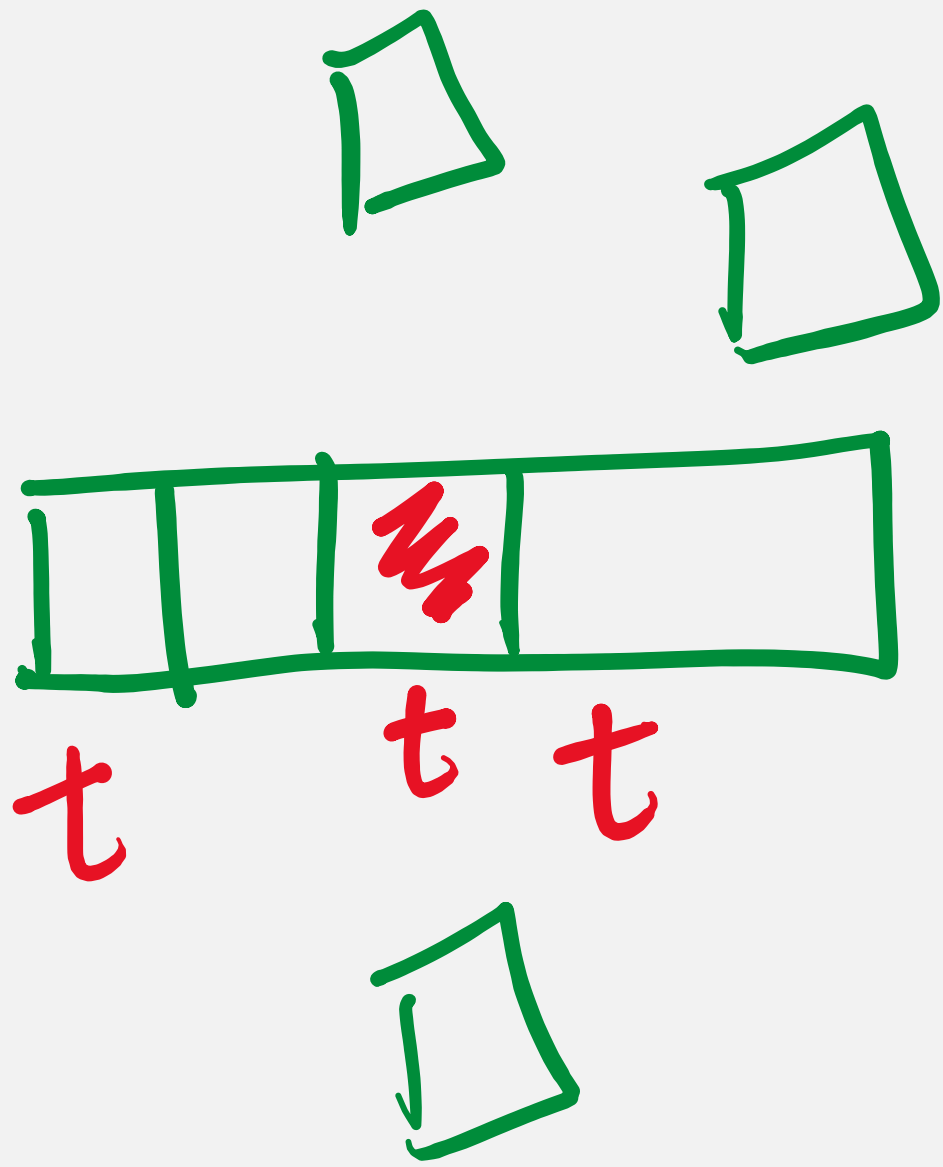
- Use array $s[]$ to store n items on stack.
- $\text{push}()$: add new item at $s[n]$.
- $\text{pop}()$: remove item from $s[n-1]$.

n the index of the first empty slot



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





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

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

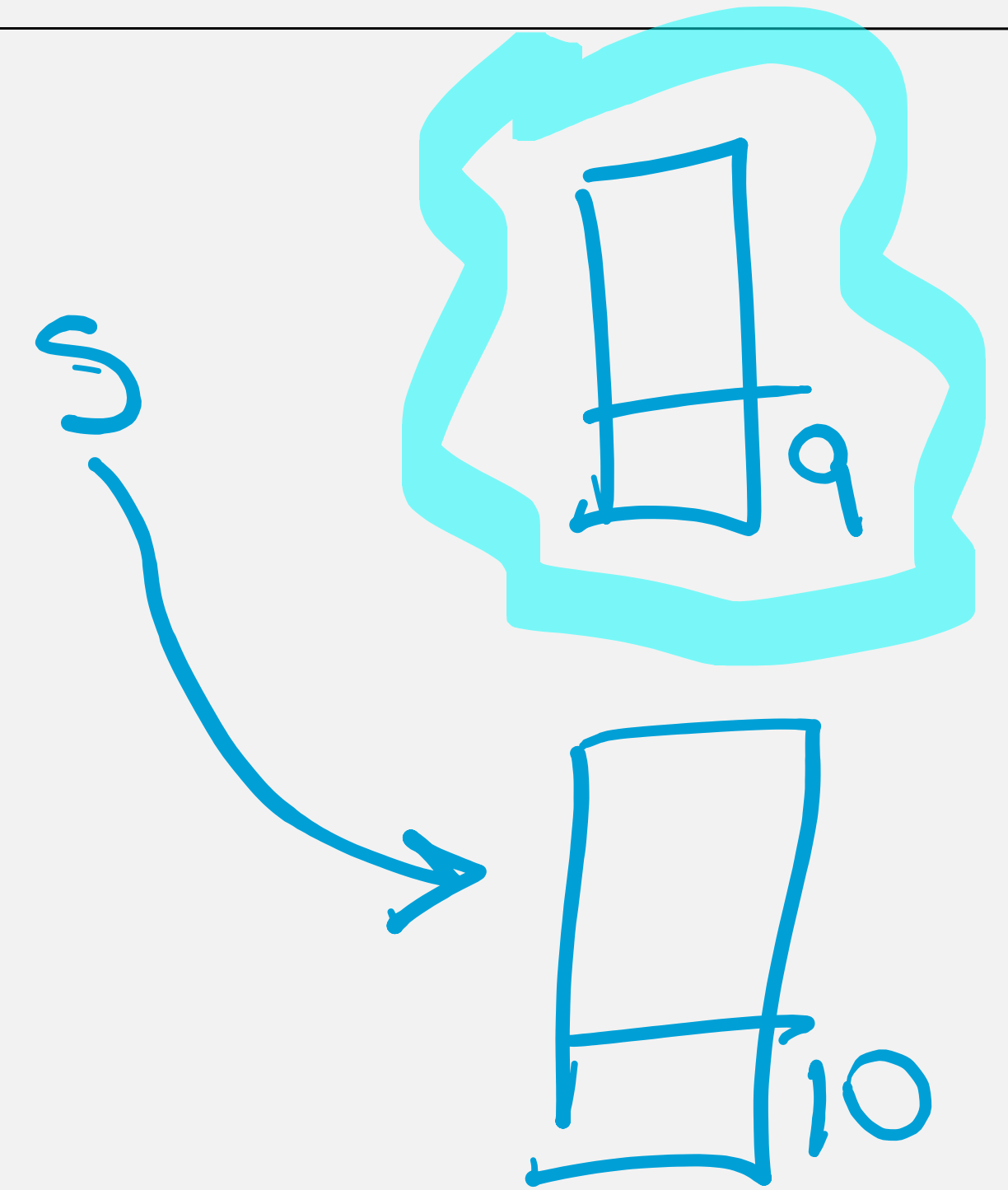
    public boolean isEmpty()
    { return n == 0; }

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

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

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

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



Overflow and underflow.

- Underflow: throw exception if `pop()` called on an empty stack.
- Overflow: use “resizing array” for array implementation. [stay tuned]

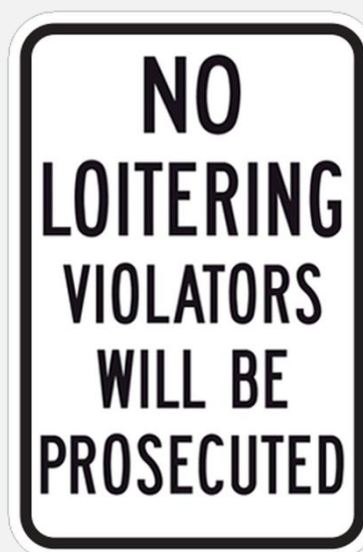
Null items. We allow `null` items to be added.

Duplicate items. We allow an item to be added more than once.

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

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

loitering



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

no loitering

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

A.

least recently added



I	have	a	dream	today	!	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
0	1	2	3	4	5	6	7	8	9

B.

most recently added



!	today	dream	a	have	I	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
0	1	2	3	4	5	6	7	8	9

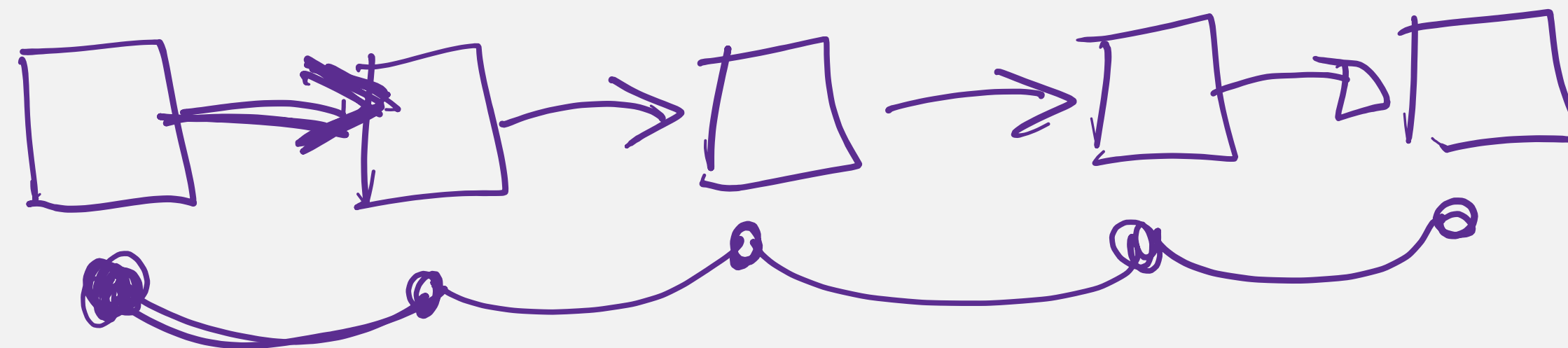
can't store things in reverse order because you would need to shift all elements to the left when you remove a[0]

C. *Both A and B.*

D. *Neither A nor B.*

INTRODUCTION TO LINKED LISTS

Node



Simplest singly-linked data structure – Linked List

Linked list

- A recursive data structure.
- Def. A *linked list* is null or a reference to a *node*.
- Def. A *node* is a data type that contains a reference to a node.
- Unwind recursion: A linked list is a sequence of nodes.

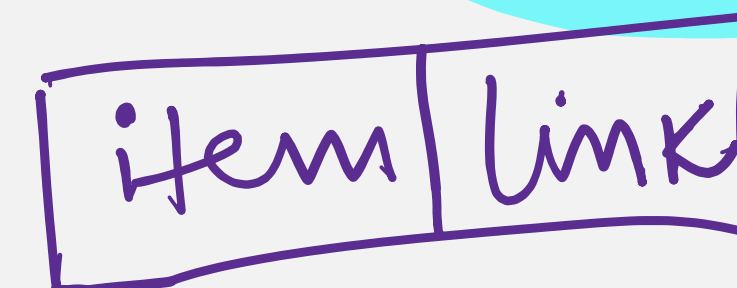
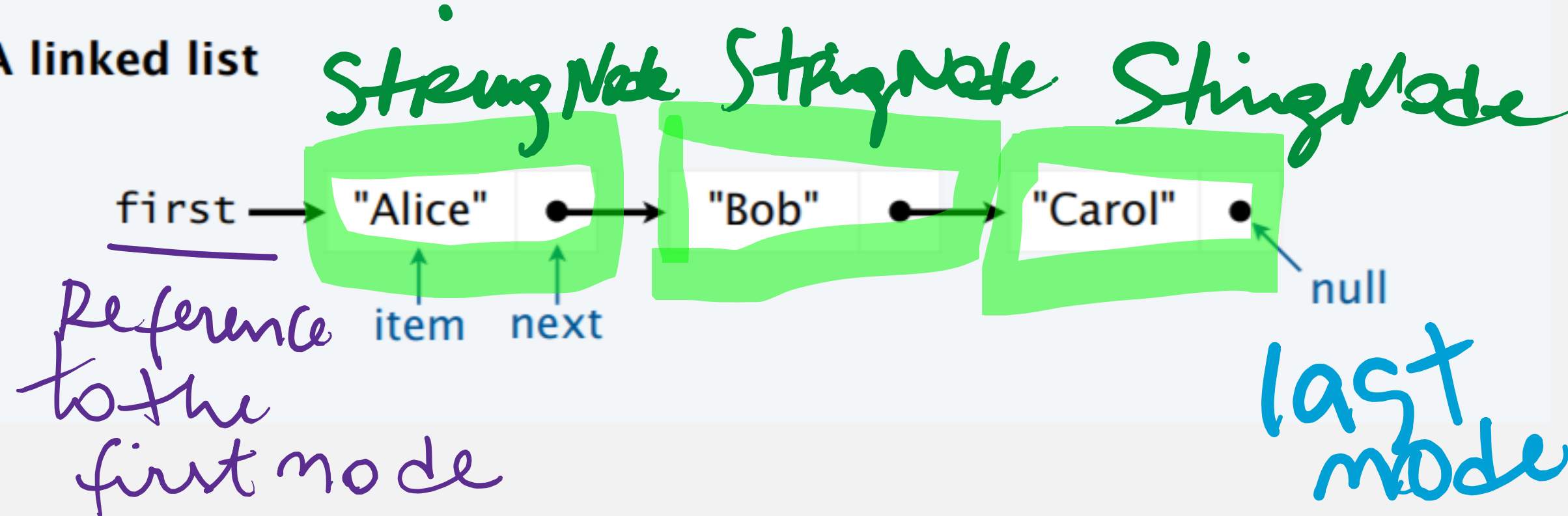
Representation

- Use a private *nested class* Node to implement the node abstraction.
- For simplicity, start with nodes having two values: a String and a Node.

```
private class Node
{
    private String item;
    private Node next;
}
```

Link

A linked list



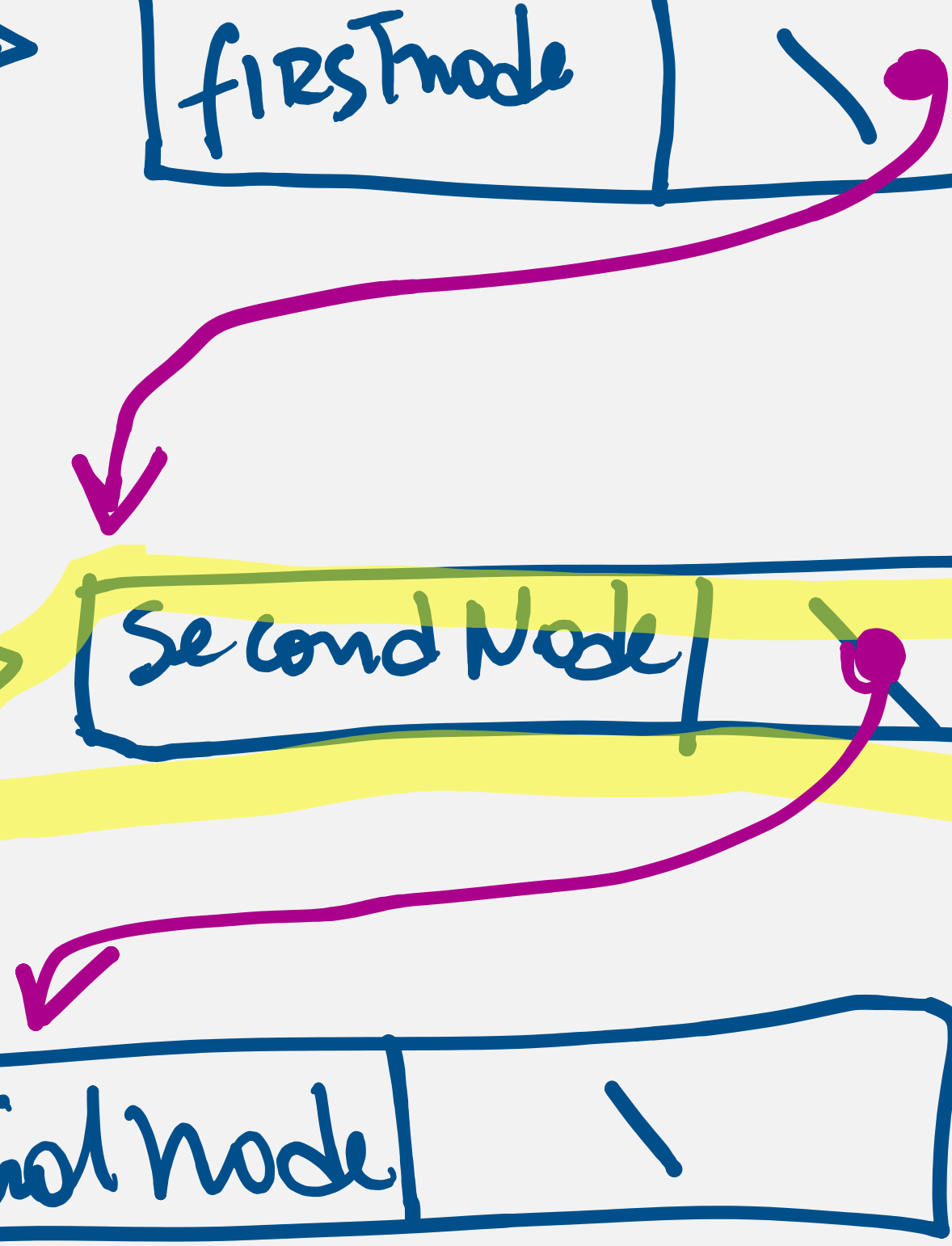
i:0 /

s₁ → CS112

m₁ → firstNode

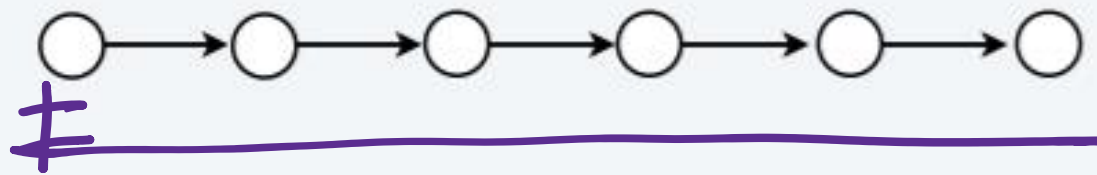
m₂ ✗ → Second Node

m₃ → third Node

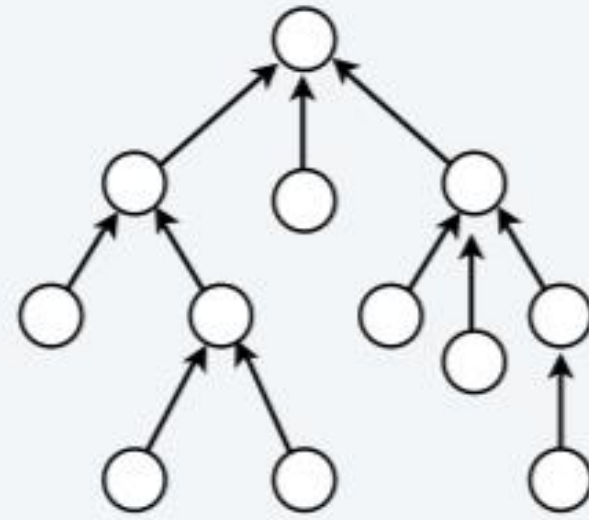


Singly Linked Data Structures

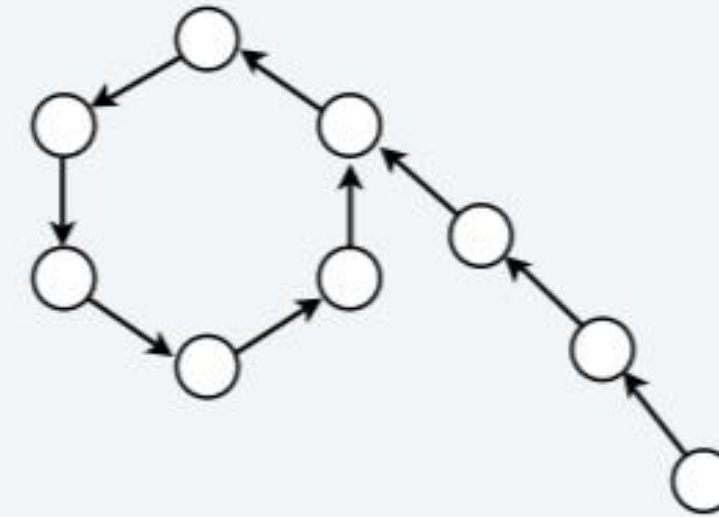
Linked list (this lecture)



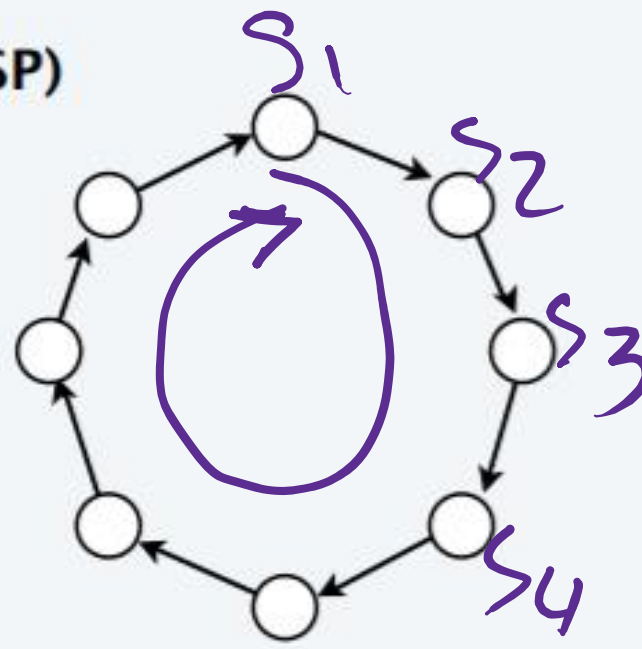
Tree



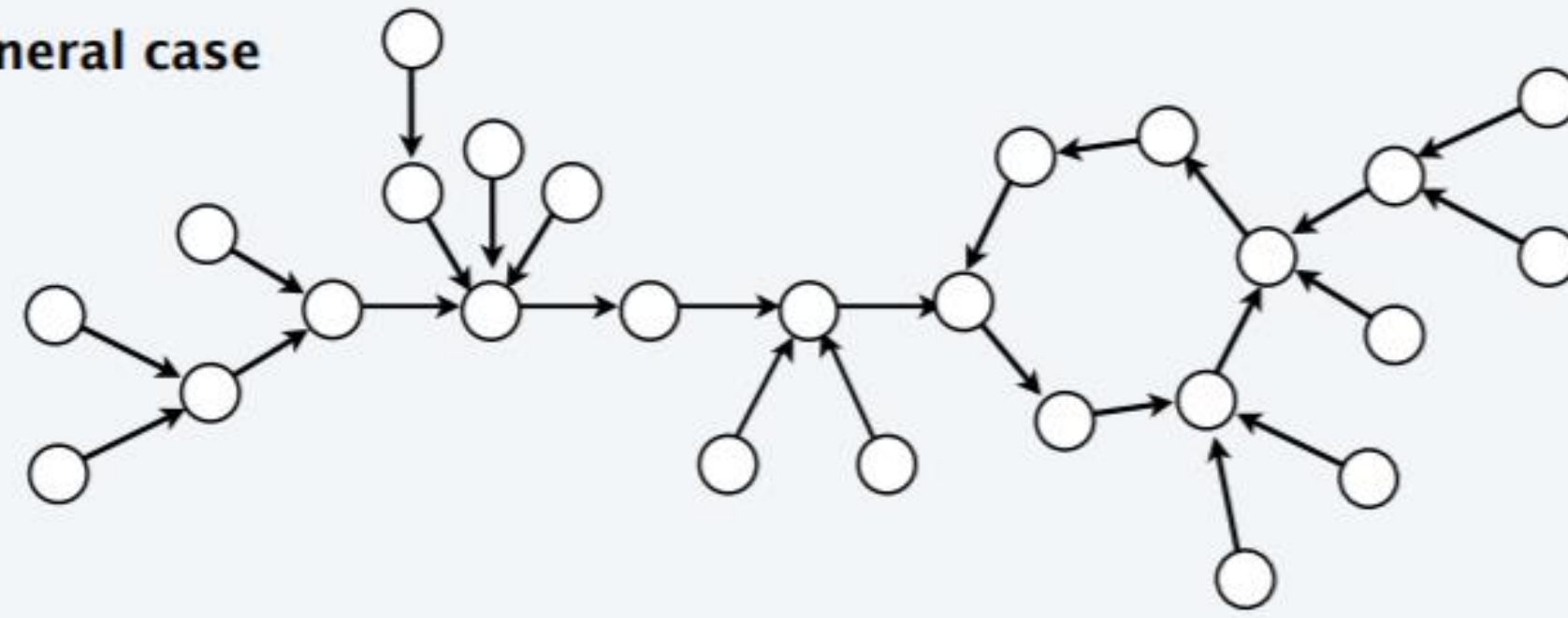
Rho



Circular list (TSP)



General case



Multiply linked structures: many more possibilities!

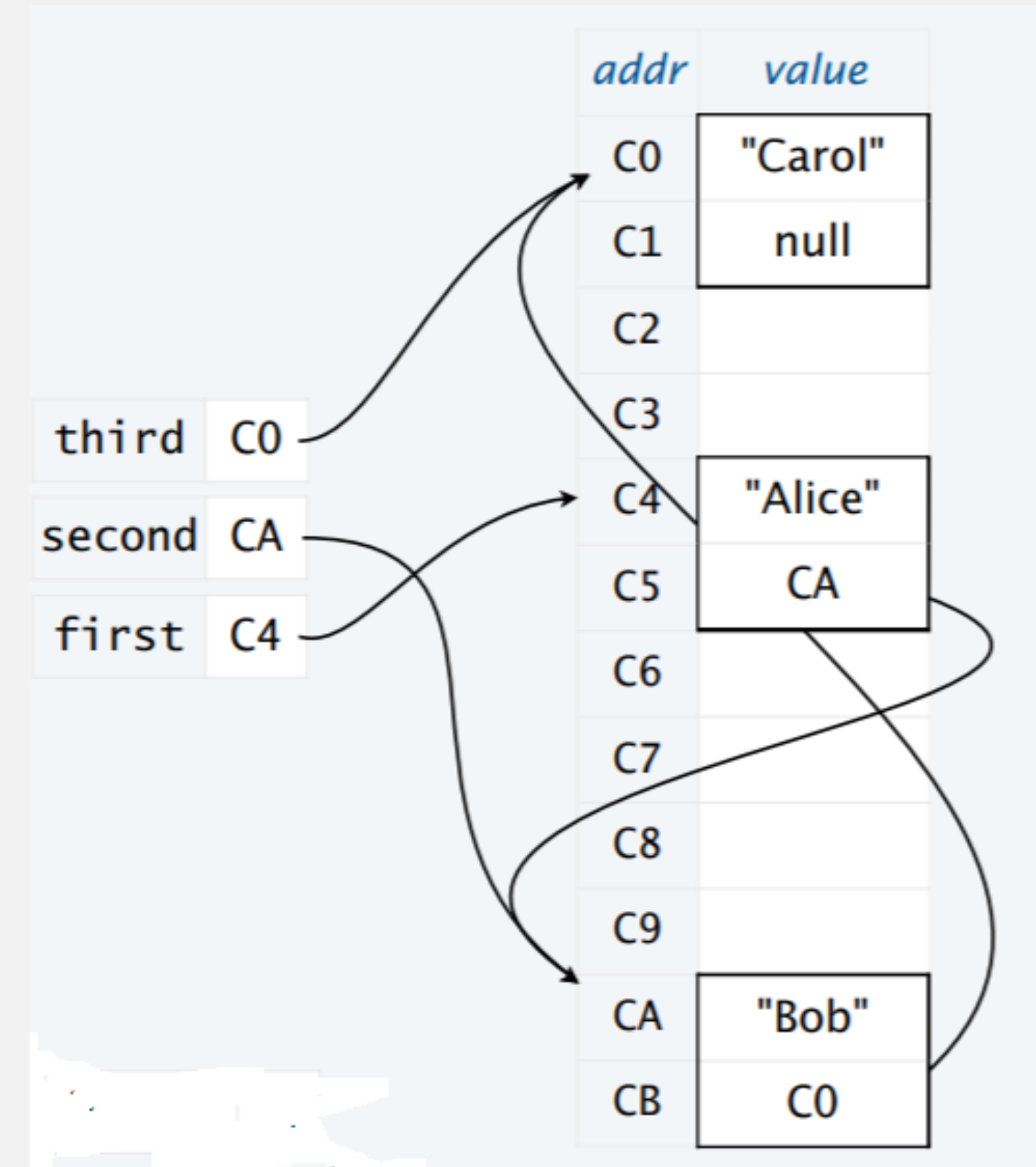
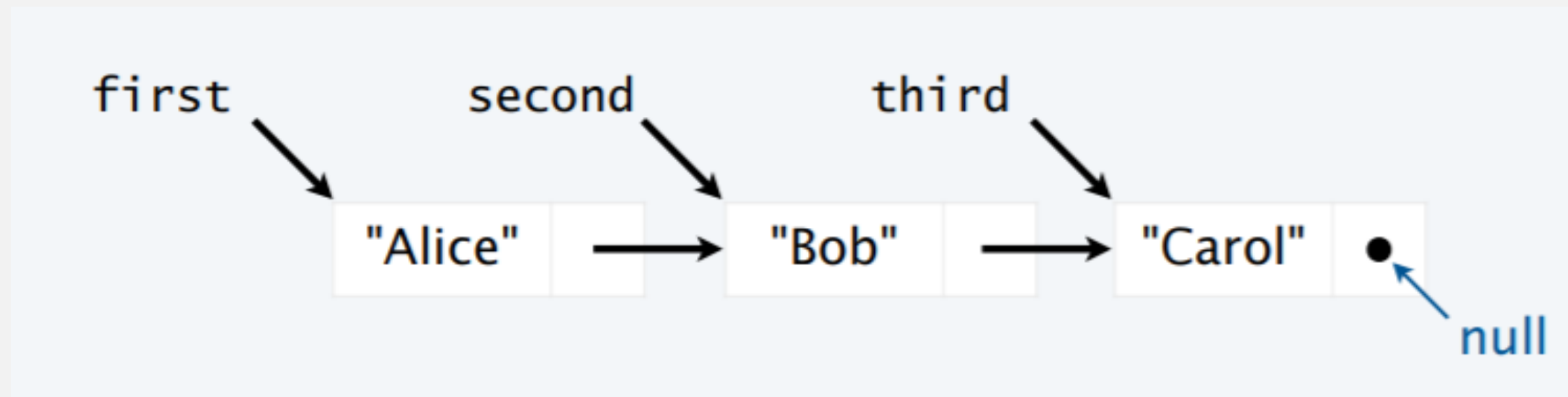
From the point of view of a particular object, all of these structures look the same.

Building a Linked List

```
Node third = new Node();  
third.item = "Carol";  
third.next = null;
```

```
Node second = new Node();  
second.item = "Bob";  
second.next = third;
```

```
Node first = new Node();  
first.item = "Alice";  
first.next = second;
```



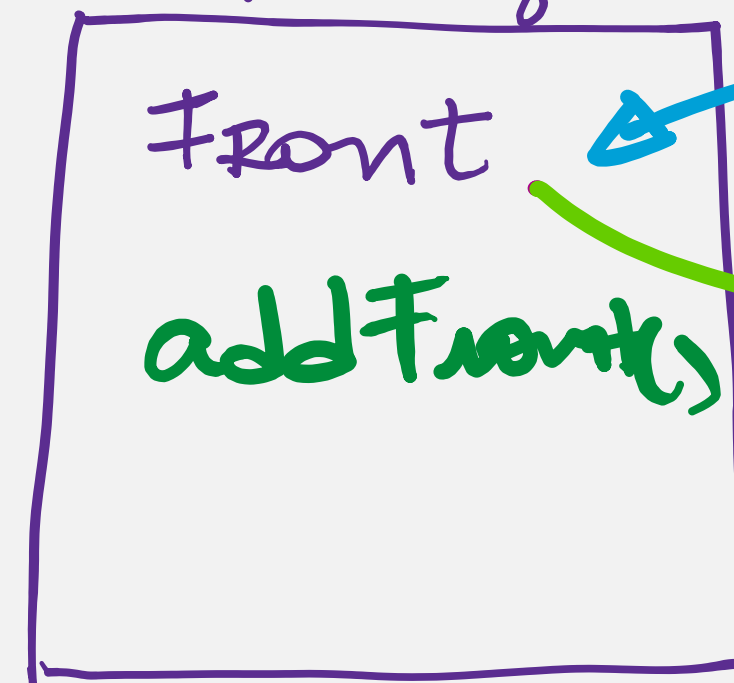
List Processing Code

Standard operations for processing data structured as a singly-linked list

1. Add a node at the beginning
2. Remove and return the node at the beginning
3. Add a node at the end (requires a reference to the last node)
4. Traverse the list (visit every node, in sequence).

\ means null

String LL reference to the first node of LL



②



①

oldFront



③



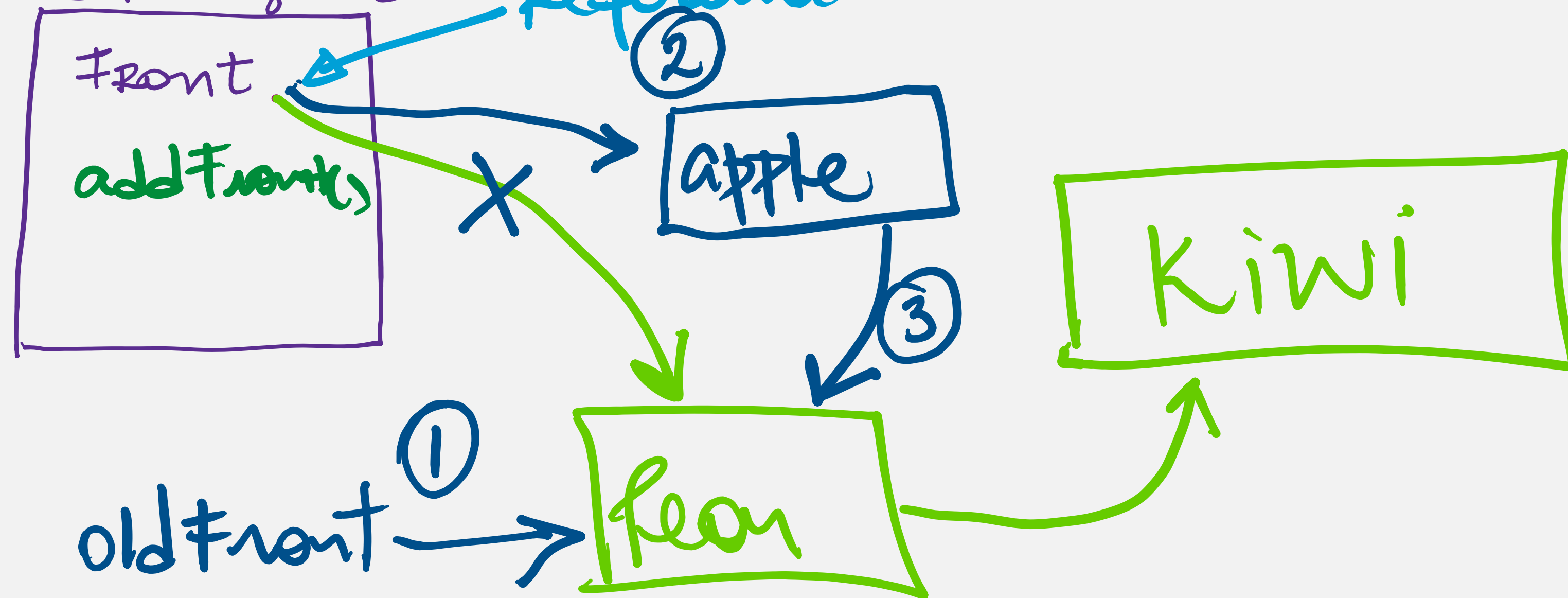
List Processing Code

Standard operations for processing data structured as a singly-linked list

- ① Add a node at the beginning
- ② Remove and return the node at the beginning
- ③ Add a node at the end (requires a reference to the last node)
- ④ Traverse the list (visit every node, in sequence).

\ means null

String LL reference to the first node of LL



Traverse a linked list

Front

↳ apple → pear → Kiwi



`PTR = Front;`

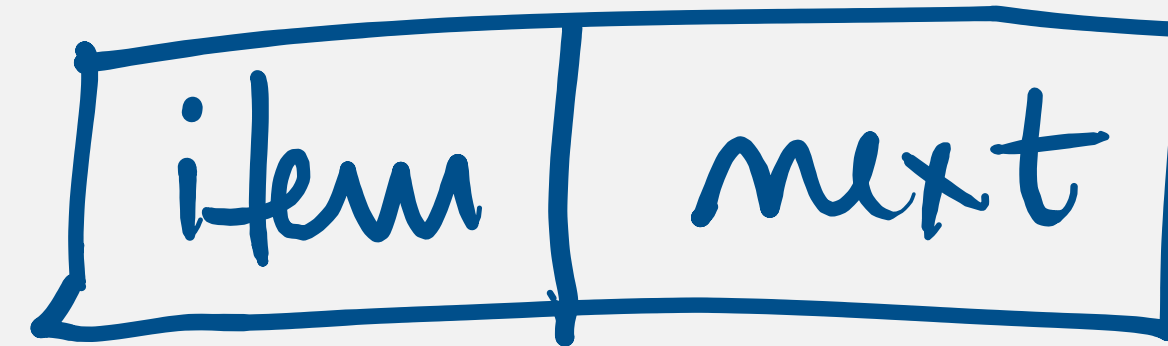
`PTR = PTR.next`

Delete the front node of the linked list

Address of pear is
Front.next

Front

~~→~~ apple → pear → Kiwi

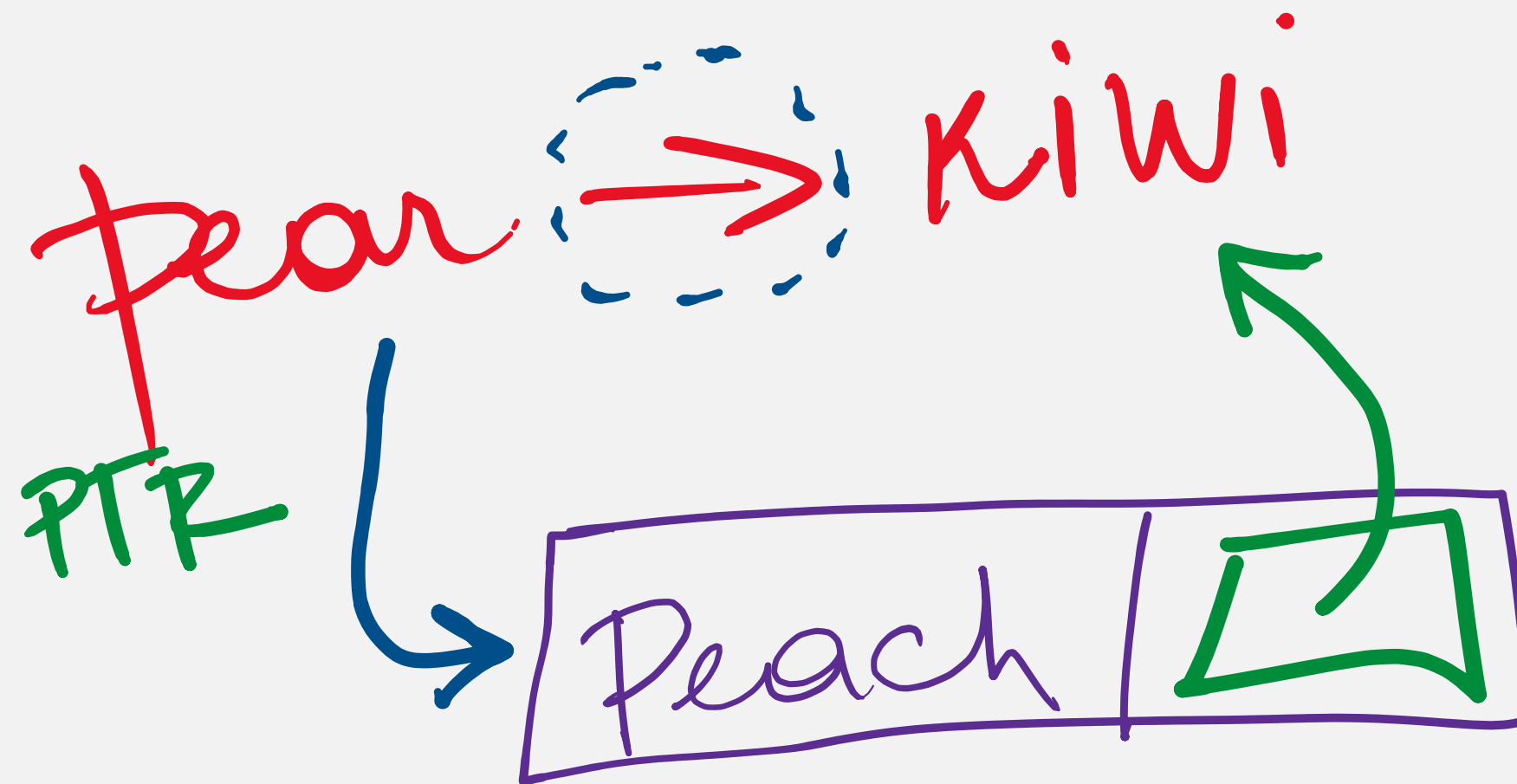
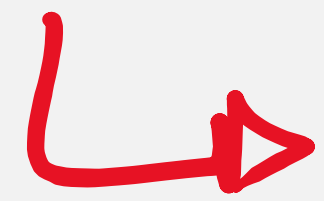


Front

→ pear → Kiwi

Insert after an item in the linked list

Front



$m.next = PTR.next$

$PTR.next = m$

Insert after an item in the linked list

Front



pear ~~→~~ kiwi

PTR



m

what if we
do this @
first?

PTR.next = m

List Processing code – Remove and Return First Item

Goal. Remove and return the first item in a linked list `first`.

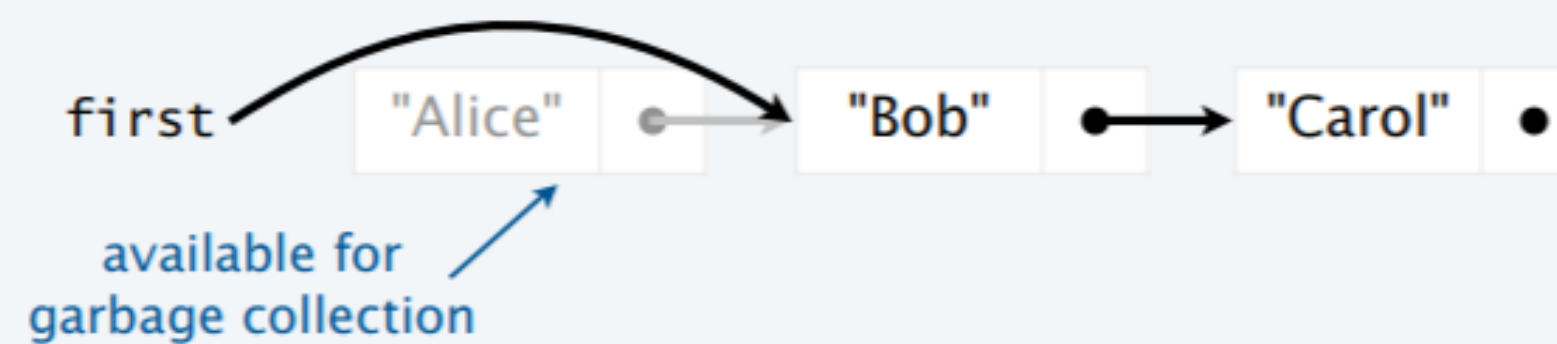
```
item = first.item;
```

`item`
"Alice"



```
first = first.next;
```

`item`
"Alice"



```
return item;
```

`item`
"Alice"



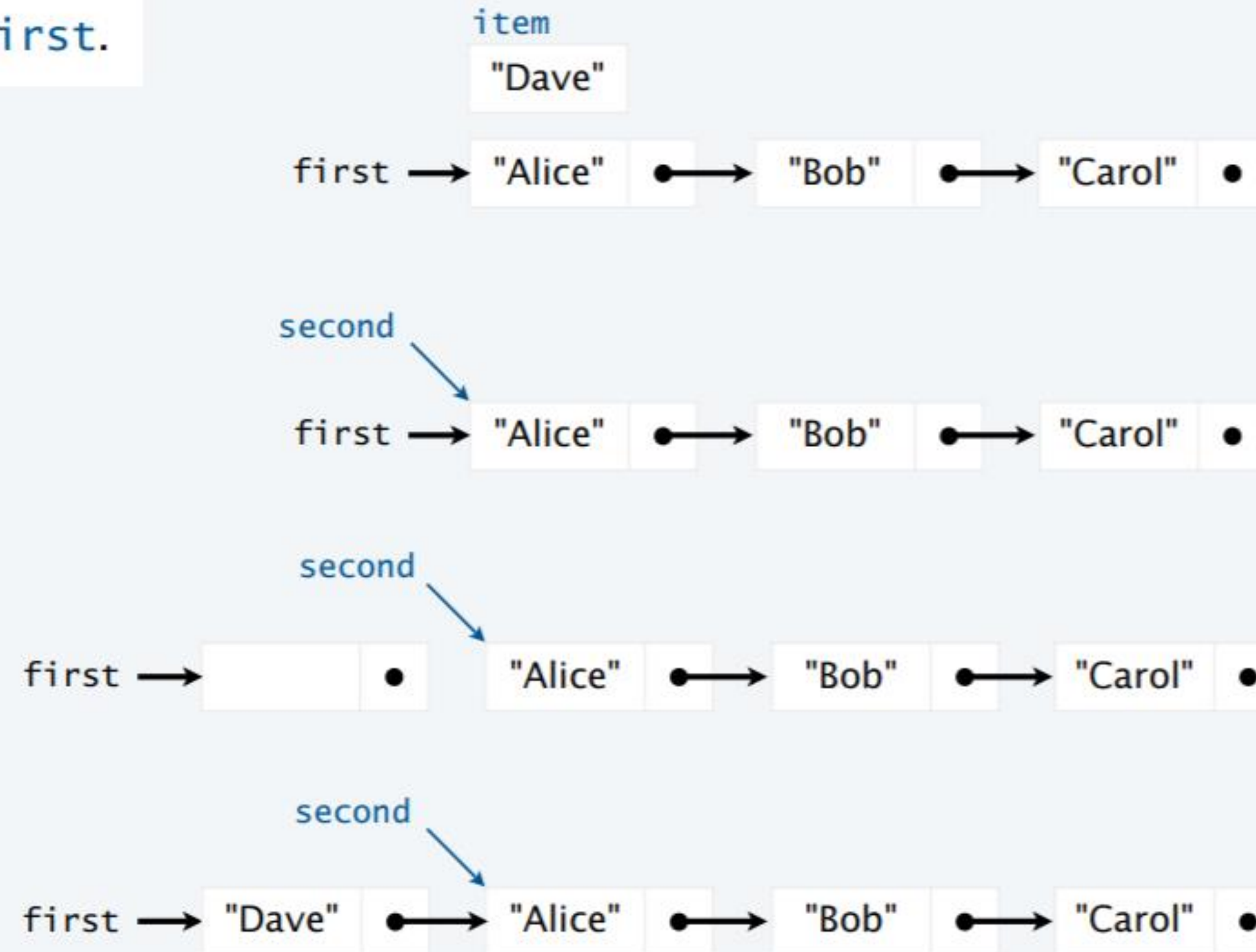
List Processing Code – Add a new node at the beginning

Goal. Add `item` to a linked list `first`.

```
Node second = first;
```

```
first = new Node();
```

```
first.item = item;  
first.next = second;
```

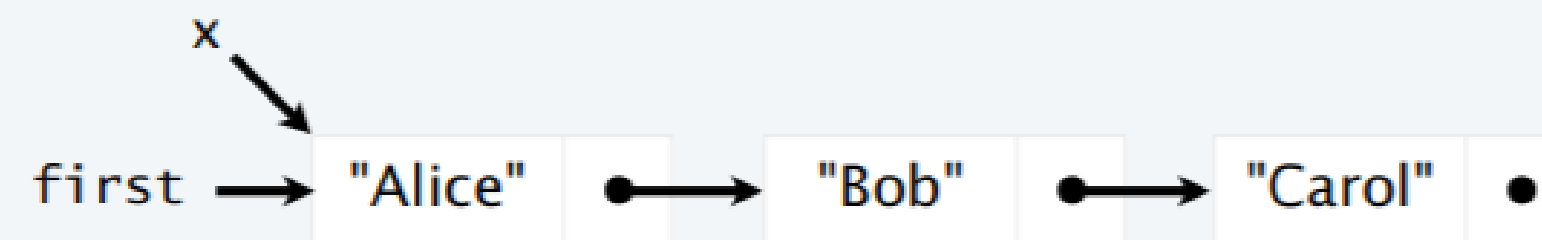


List processing code – Traverse a List

Goal. Visit every node on a linked list **first**.

➔

```
Node x = first;
while (x != null)
{
    StdOut.println(x.item);
    x = x.next;
}
```



StdOut

Alice
Bob
Carol

Pop quiz1 on Linked Lists

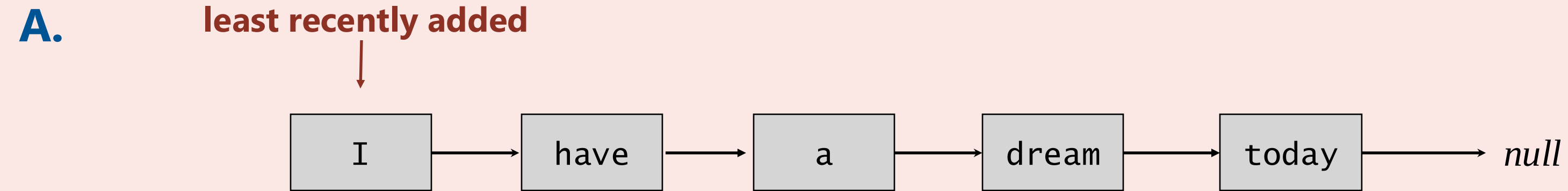
Q. What is the effect of the following code (not-so-easy question)?

```
...
Node list = null;
while (!StdIn.isEmpty())
{
    Node old = list;
    list = new Node();
    list.item = StdIn.readString();
    list.next = old;
}
for (Node t = list; t != null; t = t.next)
    StdOut.println(t.item);
...
```

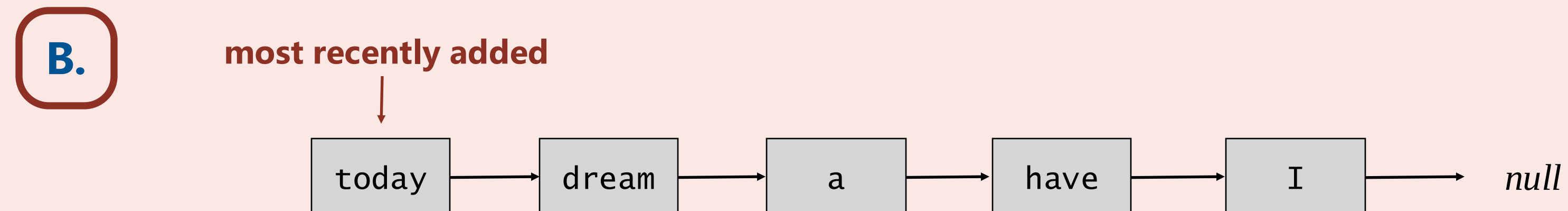
Pop quiz 2 on Linked Lists

Q. What is the effect of the following code (not-so-easy question)?

```
...  
Node list = new Node();  
list.item = StdIn.readString();  
Node last = list;  
while (!StdIn.isEmpty())  
{  
    last.next = new Node();  
    last = last.next;  
    last.item = StdIn.readString();  
}  
...
```

How to implement a stack with a singly linked list?

A would not work because while you can efficiently add items to the end (if you maintain a last pointer), there is no efficient way to remove elements from the end (and update the last pointer)

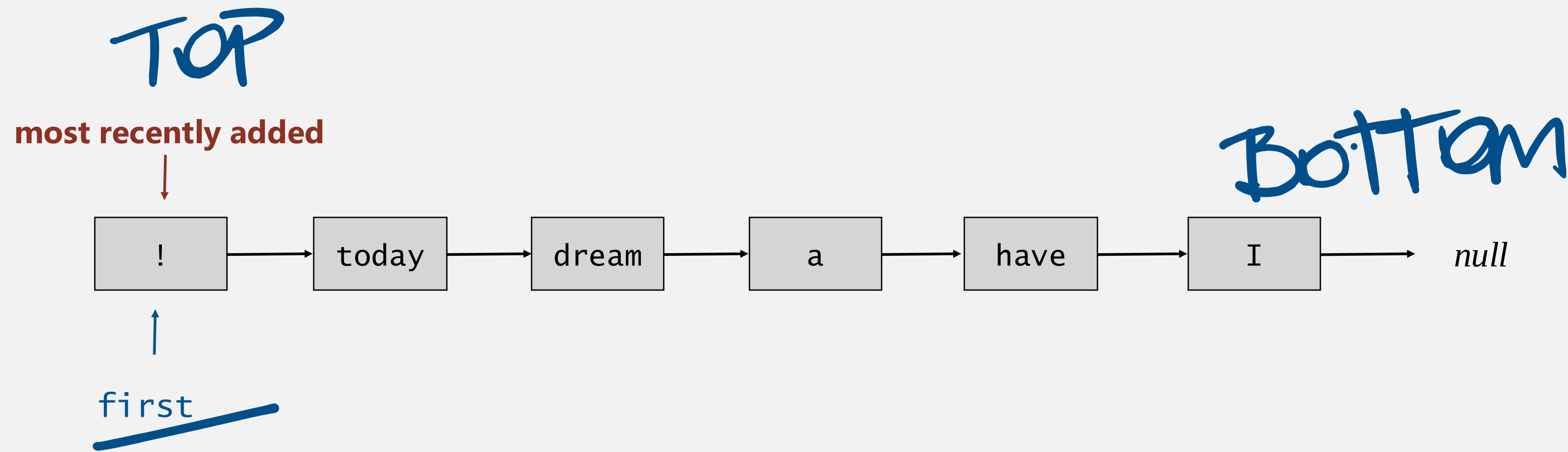


C. *Both A and B.*

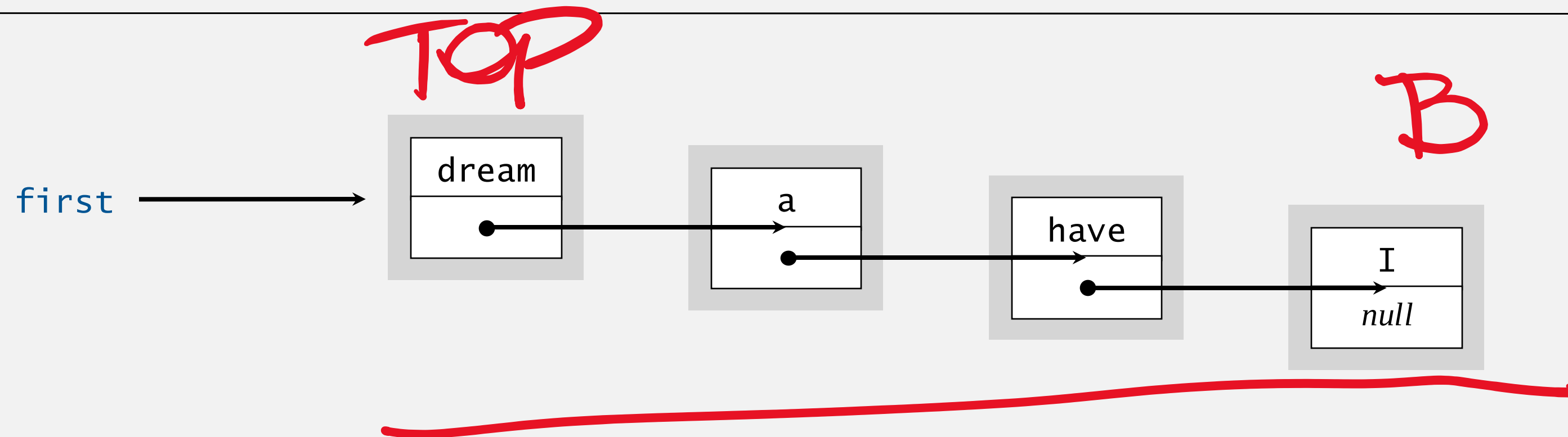
D. *Neither A nor B.*

- Maintain pointer first to first node in a singly linked list.
- Push new item before first.
- Pop item from first.

9, 2, 3



singly linked list



save item to return

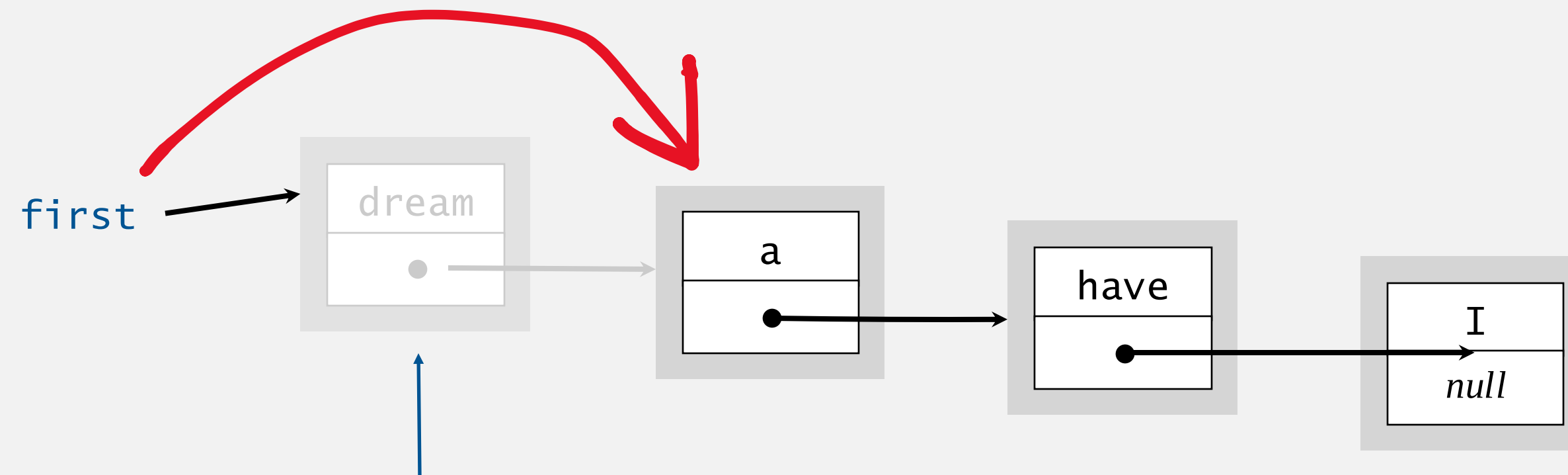
```
String item = first.item;
```

inner class

```
private class Node
{
    private String item;
    private Node next;
}
```

delete first node

```
first = first.next;
```



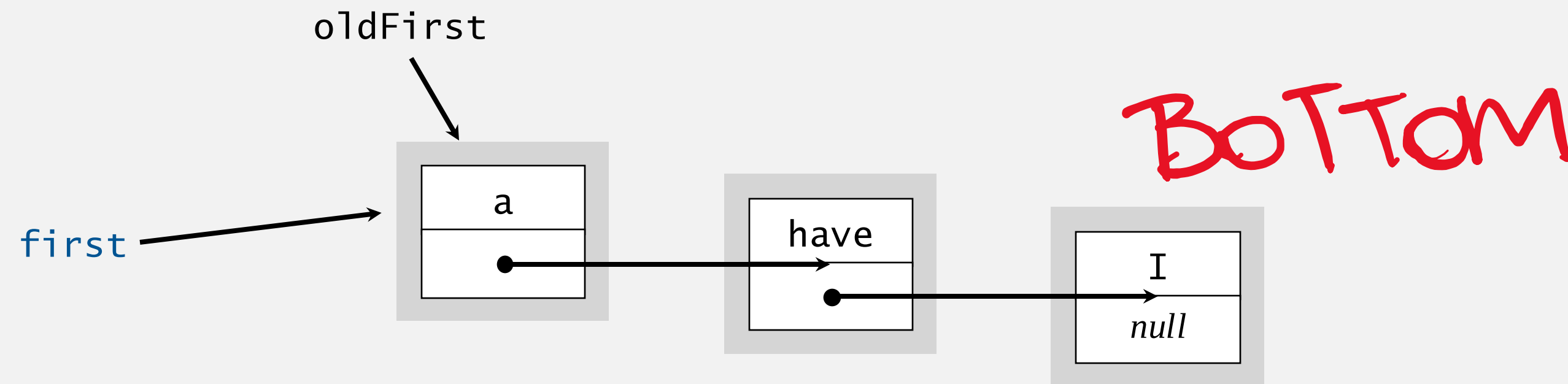
garbage collector reclaims memory
when no remaining references

return saved item

```
return item;
```

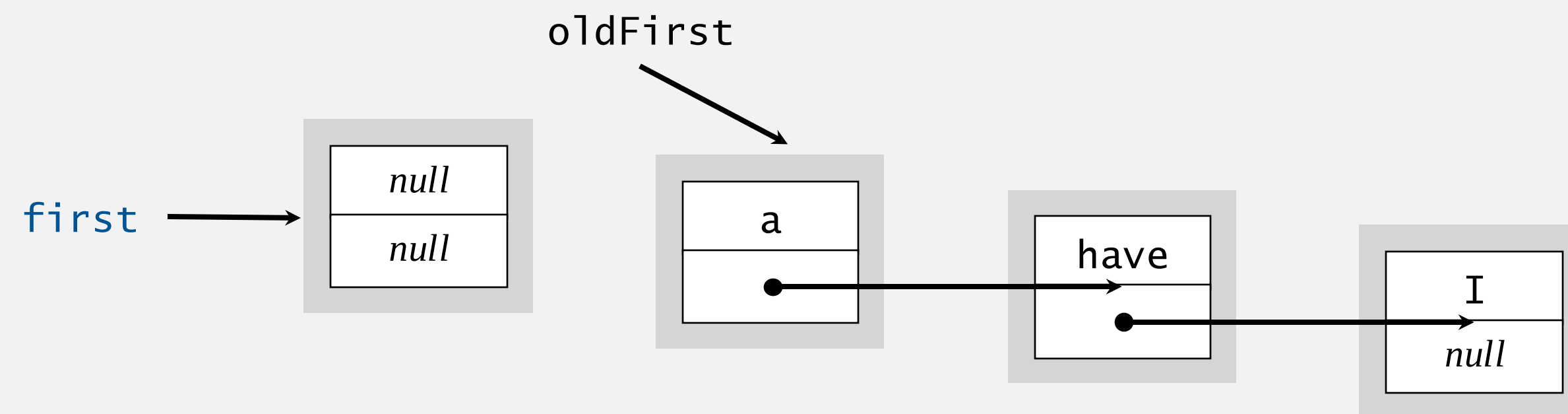
save a link to the list

```
Node oldFirst = first;
```



create a new node for the beginning

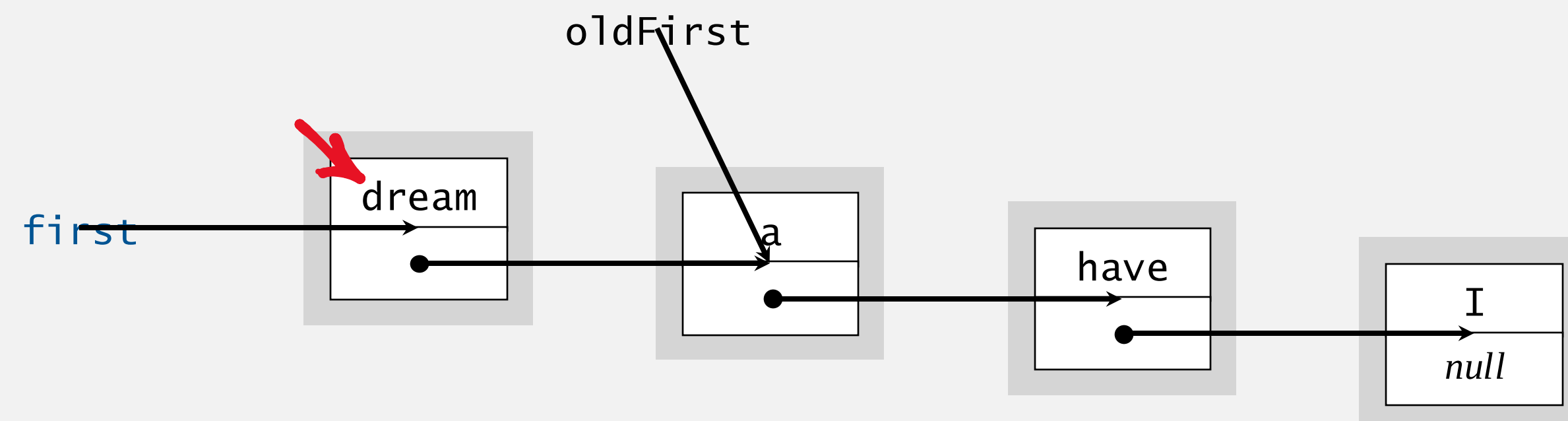
```
first = new Node();
```



initialize the instance variables in the new Node

```
first.item = "dream";
```

```
first.next = oldFirst;
```



inner class

```
private class Node
{
    private String item;
    private Node next;
}
```

```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        private String item;
        private Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(String item)
    {
        Node oldFirst = first;
        first = new Node();
        first.item = item;
        first.next = oldFirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

addFirst

deleteFirst

private inner class
(access modifiers for instance variables of such a class don't matter)

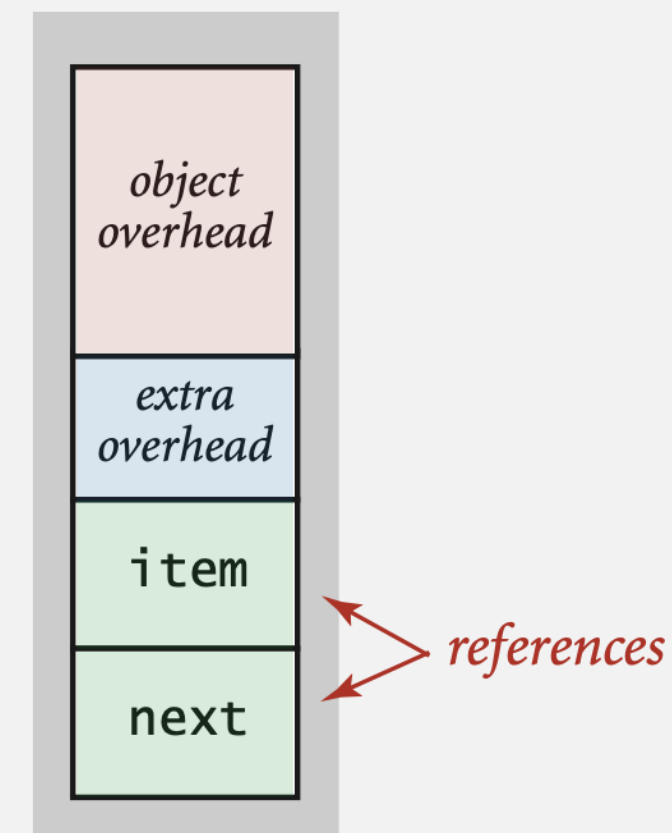
no Node constructor defined $\Rightarrow$
Java supplies default no-argument constructor

Proposition. Every operation takes $O(1)$ time.

Proposition. A stack with n items has n Node objects and uses $\sim 40 n$ bytes.

inner class

```
private class Node
{
    String item;
    Node next;
}
```



16 bytes (object overhead)

8 bytes (inner class extra overhead)

8 bytes (reference to String)

8 bytes (reference to Node)

40 bytes per stack Node

Remark. This counts only the memory for the stack itself.
(not the memory for the strings, which the client owns)

2. STACKS AND QUEUES

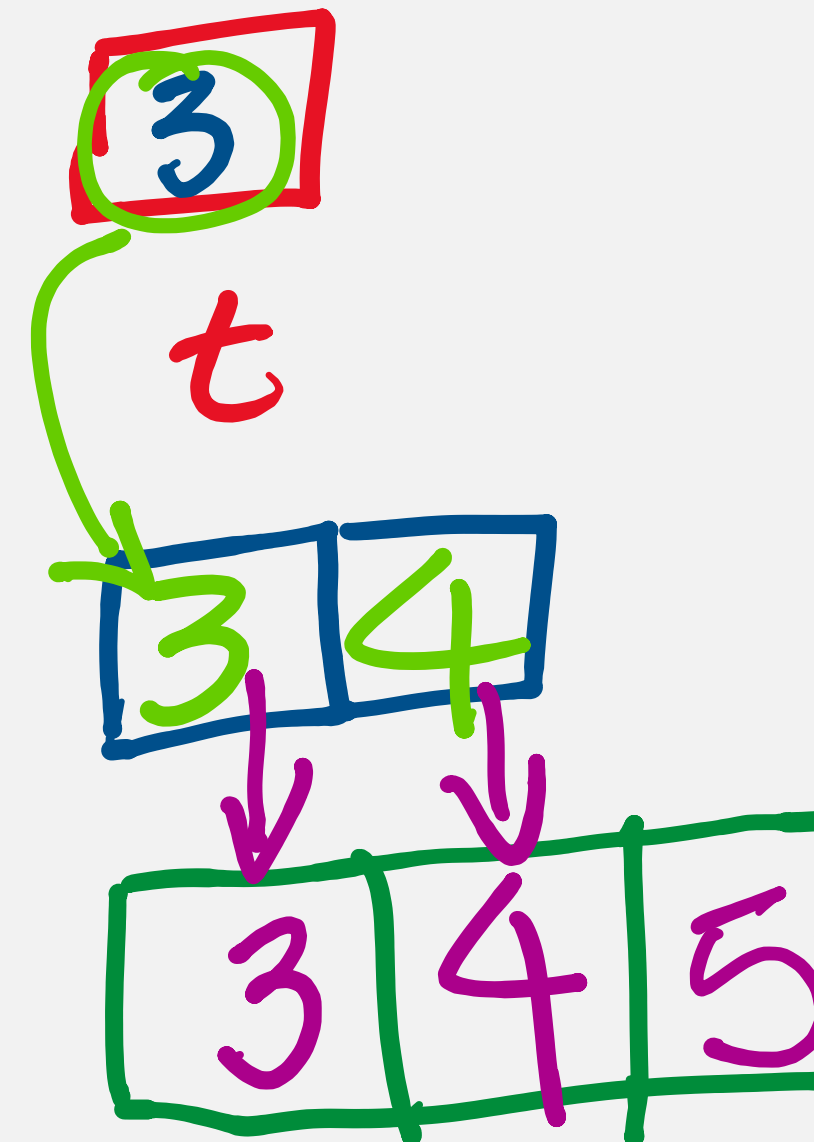
- ▶ *Stacks*
- ▶ *Resizing arrays*
- ▶ *Queues*
- ▶ *Generics*
- ▶ *Iterators (optional)*
- ▶ *Applications (optional)*

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

Q. How to grow and shrink array?

First try.

- `push()`: increase size of array `s[]` by 1.
- `pop()`: decrease size of array `s[]` by 1.



PUSH 3
- PUSH 4
- PUSH 5

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

Q. How to grow and shrink array?

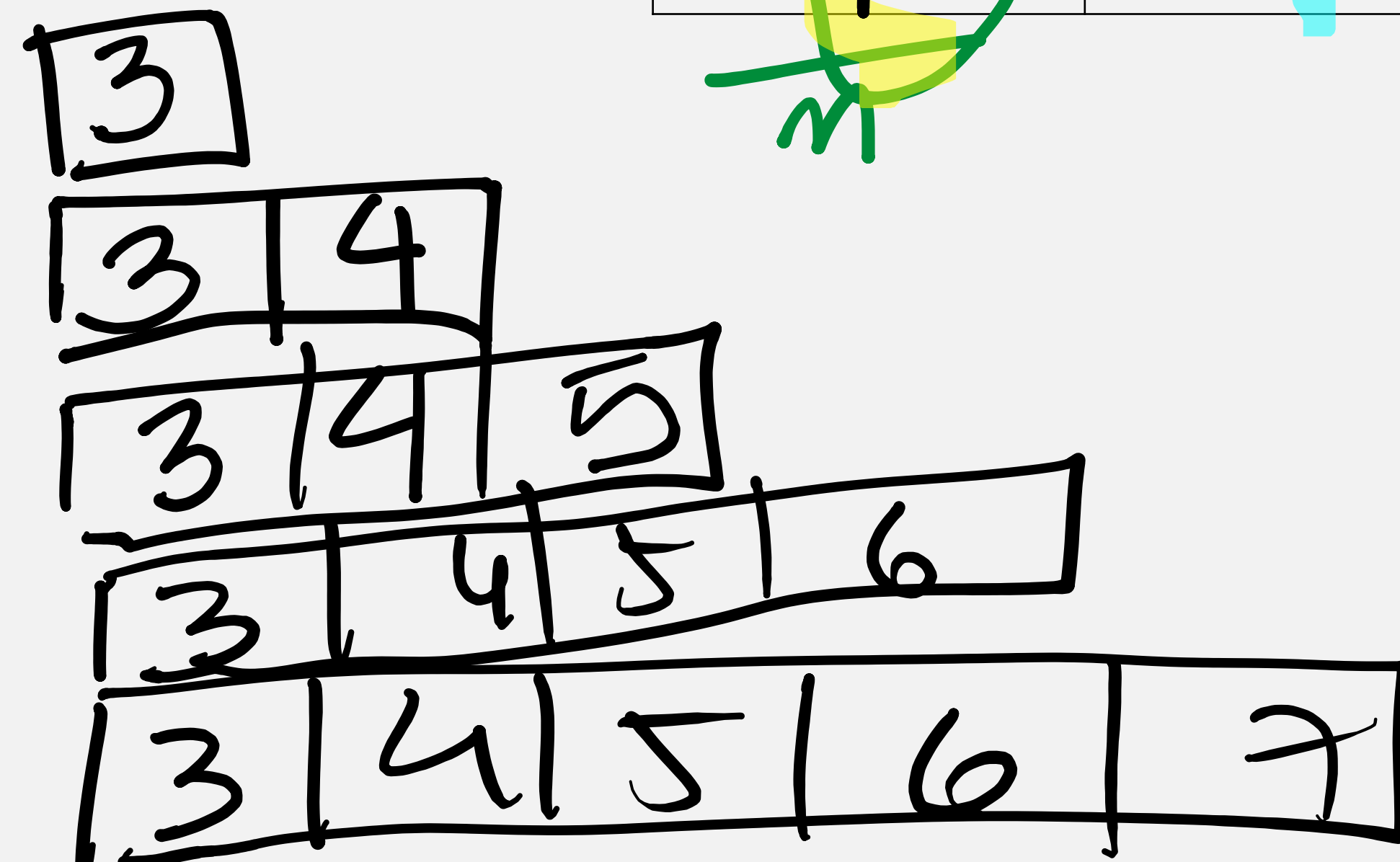
First try.

- `push()`: increase size of array `s[]` by 1.
- `pop()`: decrease size of array `s[]` by 1.

Need to copy all items to a new array, for each operation.

Cost $n=5$

$$= n + \underbrace{2+4+6+\dots+2(n-1)}_{\text{Arithmetic series}}$$



Push (1w)	K (array size)	copy (1r + 1w per position)
1	1	0
1	2	2
1	3	4
1	4	6
1	5	8

$2 \times (n-1)$

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

Q. How to grow and shrink array?

First try.

- `push()`: increase size of array `s[]` by 1.
- `pop()`: decrease size of array `s[]` by 1.

Too expensive.

- Need to copy all items to a new array, for each operation.
 - Array accesses to add first n items = $n + (2 + 4 + 6 + \dots + 2(n-1)) \sim n^2$.
- Arithmetic series
- ↑ ↑
- 1 array access 2(k-1) array accesses to expand to size k
per push (ignoring cost to create new array)
- infeasible for large n

$$\begin{aligned}
 &= n + (n-1)/2 * (2 + 2(n-1)) \\
 &= n + (n-1)/2 * (2 + 2n - 2) \\
 &= n + (n-1)/2 * 2n \\
 &= n + (2n^2 - 2n)/2 \\
 &= n + 2(n^2 - n)/2 \\
 &= n + n^2 - n \\
 &= n^2 - n
 \end{aligned}$$

Challenge. Ensure that array resizing happens infrequently.

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

Q. How to grow and shrink array?

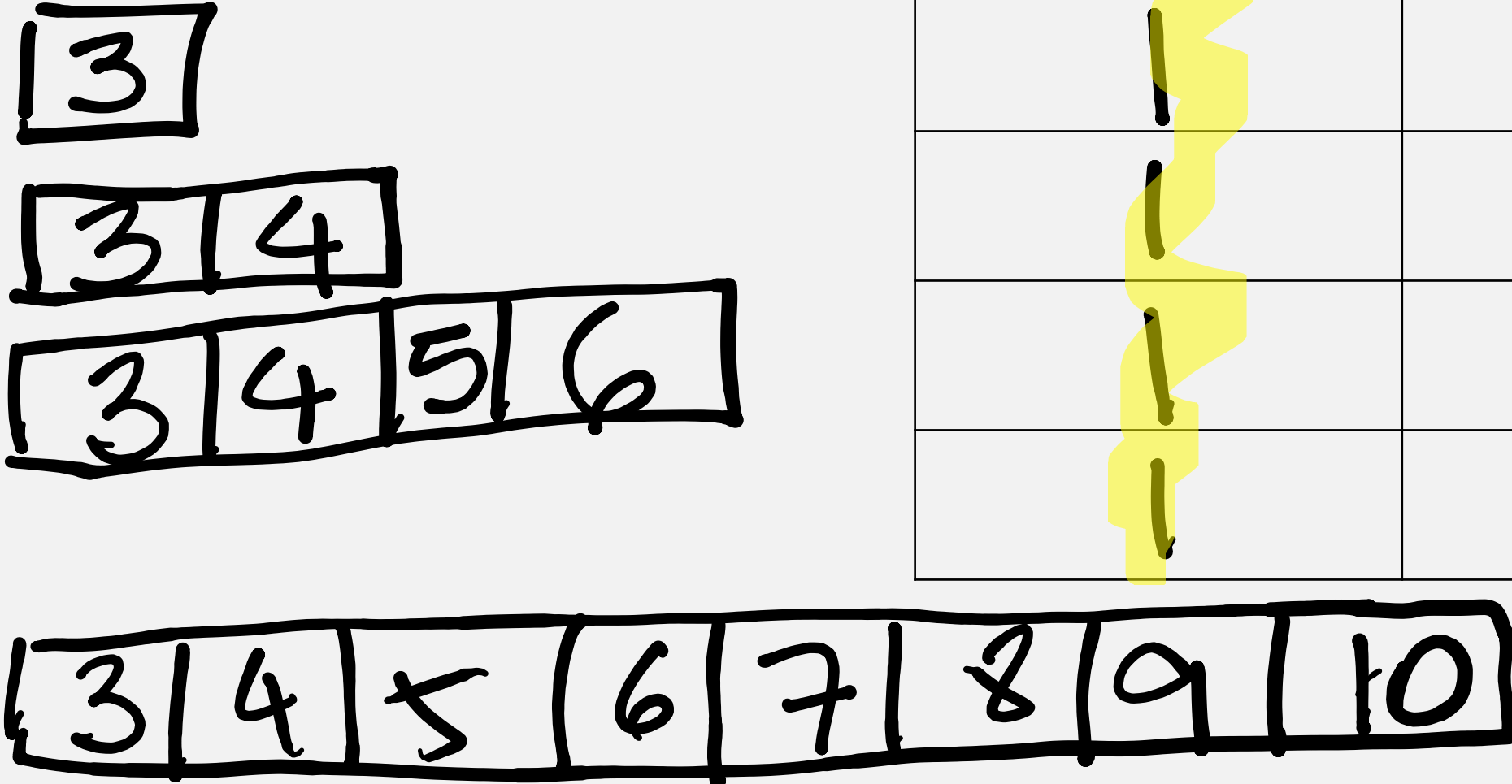
Second try.

- push(): double the size of array s[].
- pop(): half the size of array s[] .

Need to copy all items to a new array, for each operation.

$n + 2 + 4 + 8$
[
geometric series

$n = 8$
 $t = 3 = \log_2 n$



Push (1w)	K (array size)	copy (1r + 1w per position)
1	1	
1	2	2
1	4	4
1	4	0
1	8	8
1	8	0
1	8	0
1	8	0

Q. How to grow array?

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

```
public ResizingArrayStackOfStrings()
{
    s = new String[1];
}

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

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

Array accesses to add first $n = 2^i$ items.

$$n + (2 + 4 + 8 + \dots + n) \sim 3n.$$

1 array access
per push

k array accesses to double to size k
(ignoring cost to create new array)

“repeated doubling”

Geometric series

→ a : first term
 r : ratio
 t : number of terms

$$\begin{aligned} \text{Sum} &= a \left(\frac{1-r^t}{1-r} \right) \\ &= 2 \left(\frac{1-2^{\log n}}{1-2} \right) \\ 2^{\log n} &= n \\ &= 2 \left(\frac{1-n}{1-2} \right) \\ &= 2 \left(\frac{1-n}{-1} \right) \\ &= (2-2n)/-1 \\ &= 2n - 2 \end{aligned}$$

feasible for large n

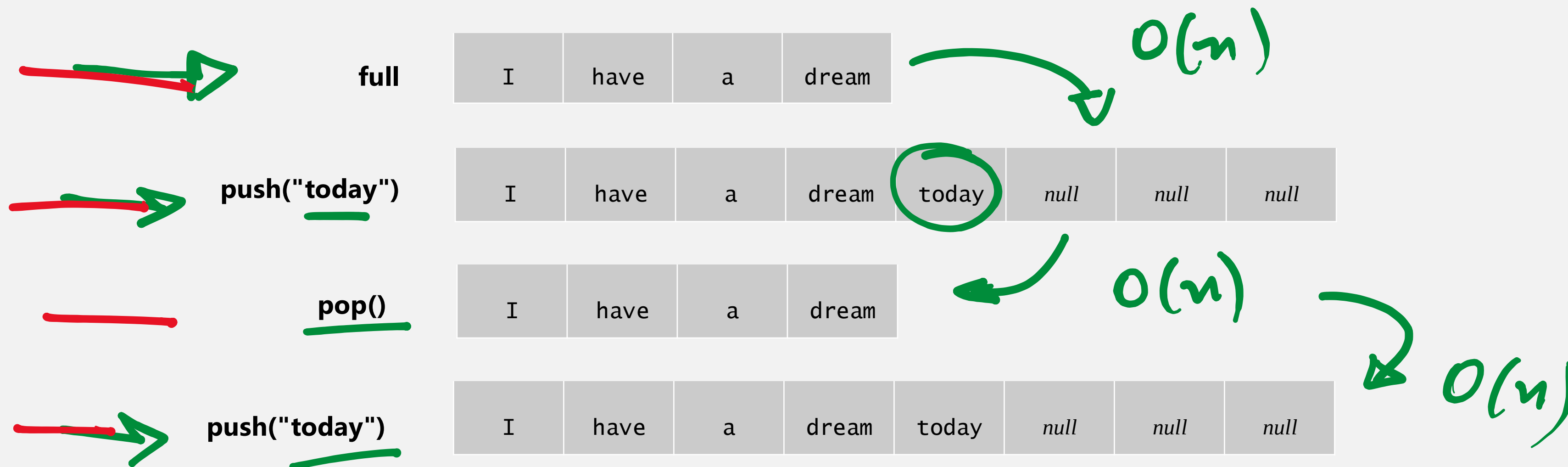
Q. How to shrink array?

First try.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is **one-half full**.

Too expensive in worst case.

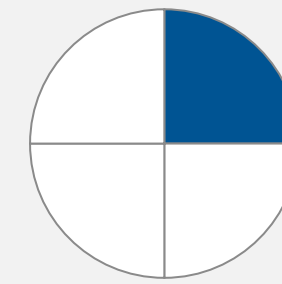
- Consider alternating sequence of push and pop operations when array is full.
- Each operation takes $O(n)$ time.



Q. How to shrink array?

Efficient solution.

- push(): double size of array `s[]` when array is full.
- pop(): halve size of array `s[]` when array is one-quarter full.



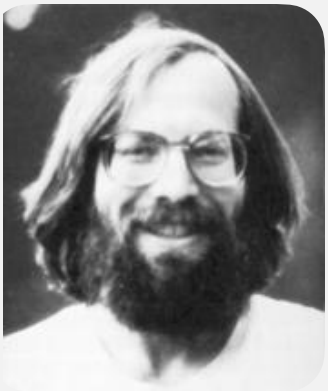
```
public String pop()
{
    String item = s[--n];
    s[n] = null;
    if (n > 0 && n == s.length/4) resize(s.length/2);
    return item;
}
```

Invariant. Array is between 25% and 100% full.

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

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

Amortized analysis. Starting from an empty data structure, **average running time per operation** over a **worst-case** sequence of operations.



Bob Tarjan
(1986 Turing award)

	worst	amortized
construct	1	1
push	n	1
pop	n	1
size	1	1

order of growth of running time
for resizing-array stack with n items

array is full
array is $\frac{1}{4}$ full

Proposition. A `ResizingArrayStackOfStrings` uses between $\sim 8n$ and $\sim 32n$ bytes of memory for a stack with n items.

- $\sim 8n$ when full. [array length = n]
- $\sim 32n$ when one-quarter full. [array length = $4n$]

```
public class ResizingArrayStackOfStrings
{
    private String[] s;
    private int n = 0;
    :
}
```

← 8 bytes × array length

Remark. This counts only the memory for the stack itself.
(not the memory for the strings, which the client allocates)

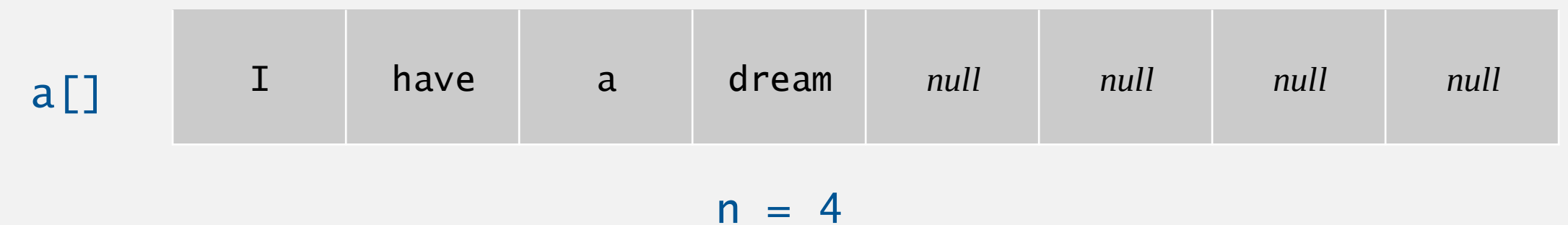
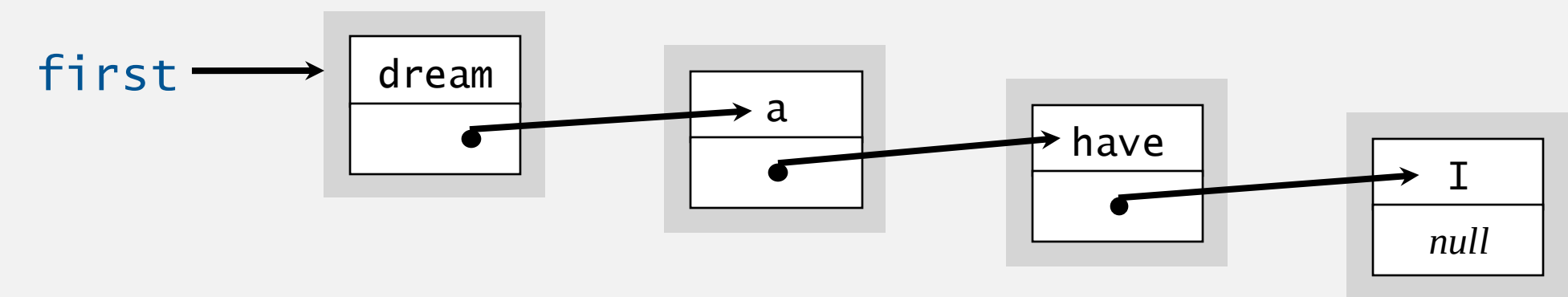
Tradeoffs. Can implement a stack with either resizing array or linked list. Which is better?

Linked-list implementation.

- Stronger performance guarantee: $O(1)$ worst case.
- More memory.

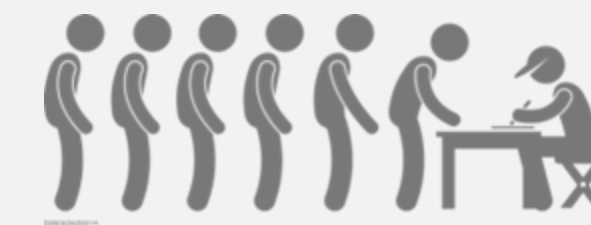
Resizing-array implementation.

- Weaker performance guarantee: $O(1)$ amortized.
- Less memory.
- Better use of cache.

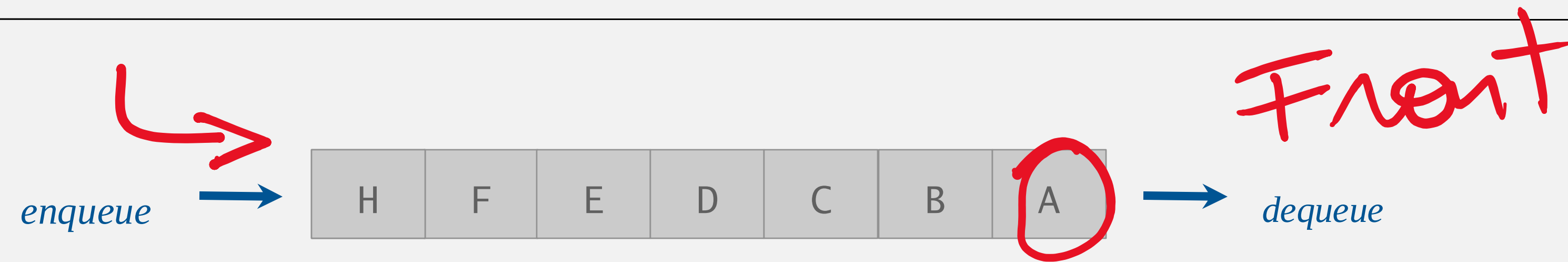


2. STACKS AND QUEUES

- ▶ *Stacks*
- ▶ *Resizing arrays*
- ▶ **Queues**
- ▶ *Generics*
- ▶ *Iterators (optional)*
- ▶ *Applications*



FIFO



```
public class QueueOfStrings
```

```
    QueueOfStrings()
```

create an empty queue

```
    void enqueue(String item)
```

add a new string to queue

```
    String dequeue()
```

*remove and return the
string^[L]_[SEP] least recently added*

```
    boolean isEmpty()
```

is the queue empty?

```
    int size()
```

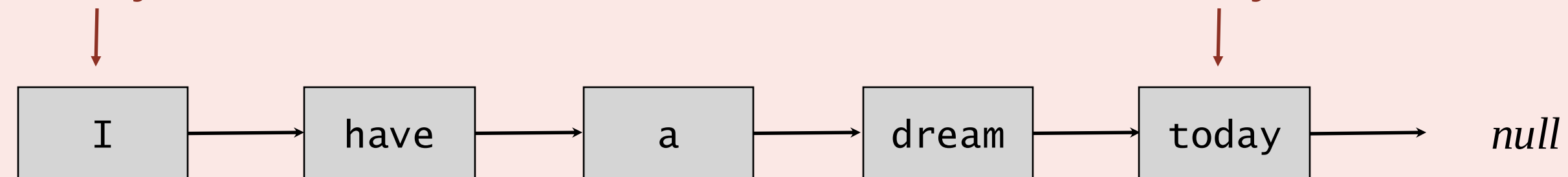
number of strings on the queue

Performance goal. Every operation takes $O(1)$ time.

How to implement a queue with a singly linked list?

A.

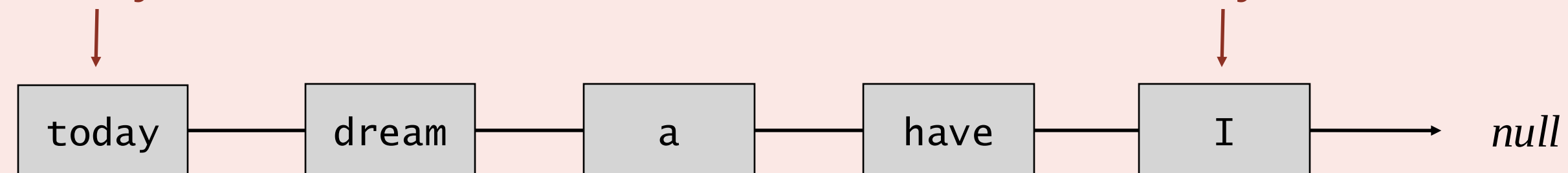
least recently added



most recently added

B.

most recently added



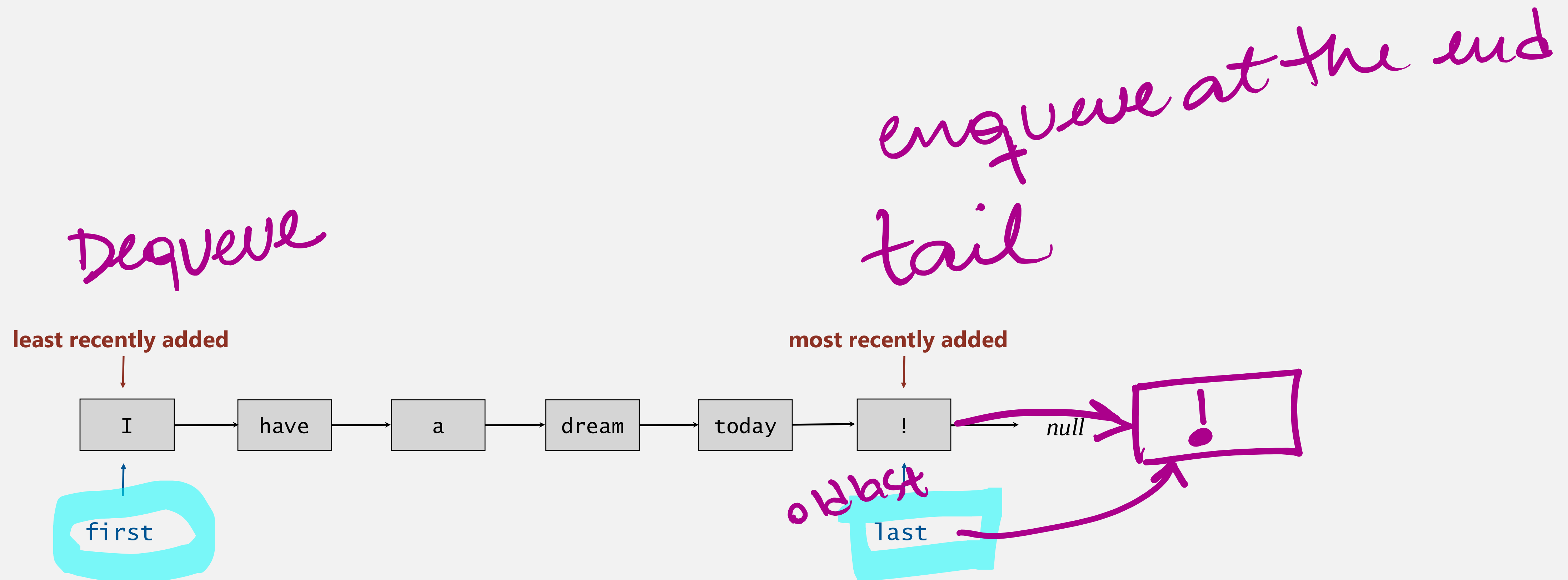
least recently added

B doesn't work because there is no efficient way to delete from the end of the linked list (and update the last pointer) .

C. Both A and B.

D. Neither A nor B.

- Maintain one pointer *first* to first node in a singly linked list.
- Maintain another pointer *last* to last node.
- Dequeue from first.
- Enqueue after *last*.



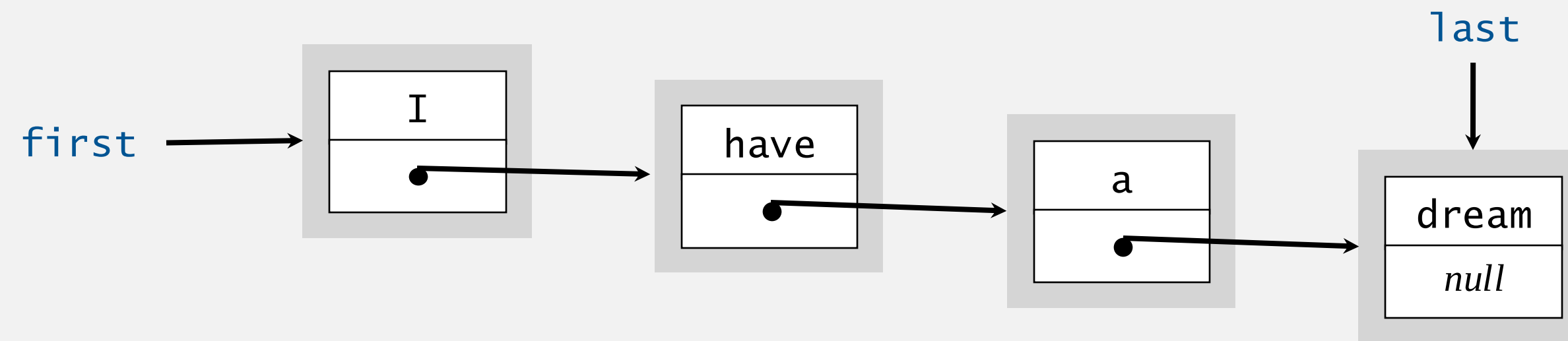
Queue dequeue: linked-list implementation

LO 2.1

Code is identical to pop().

$O(1)$

singly linked list

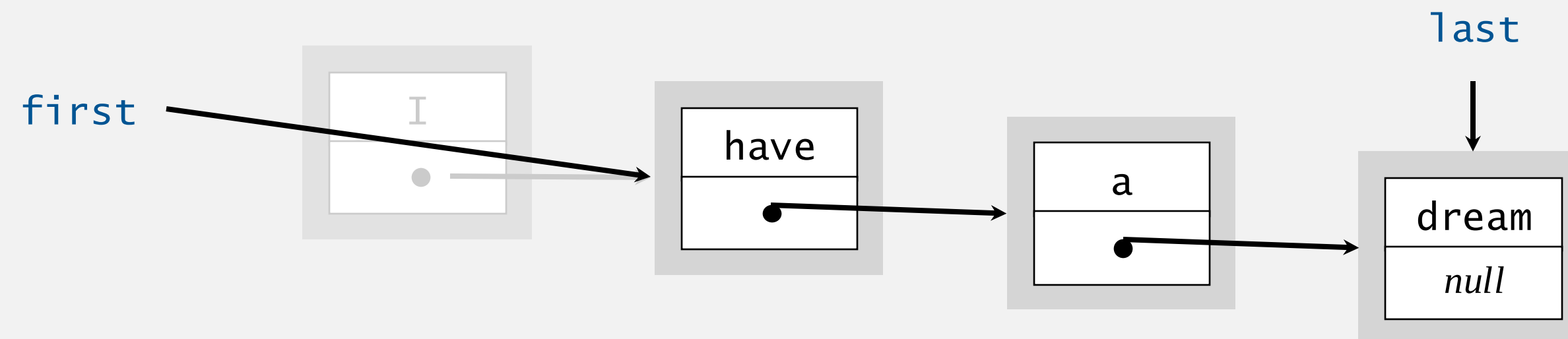


save item to return

① String item = first.item;

delete first node

② first = first.next;



return saved item

③ return item; ↙

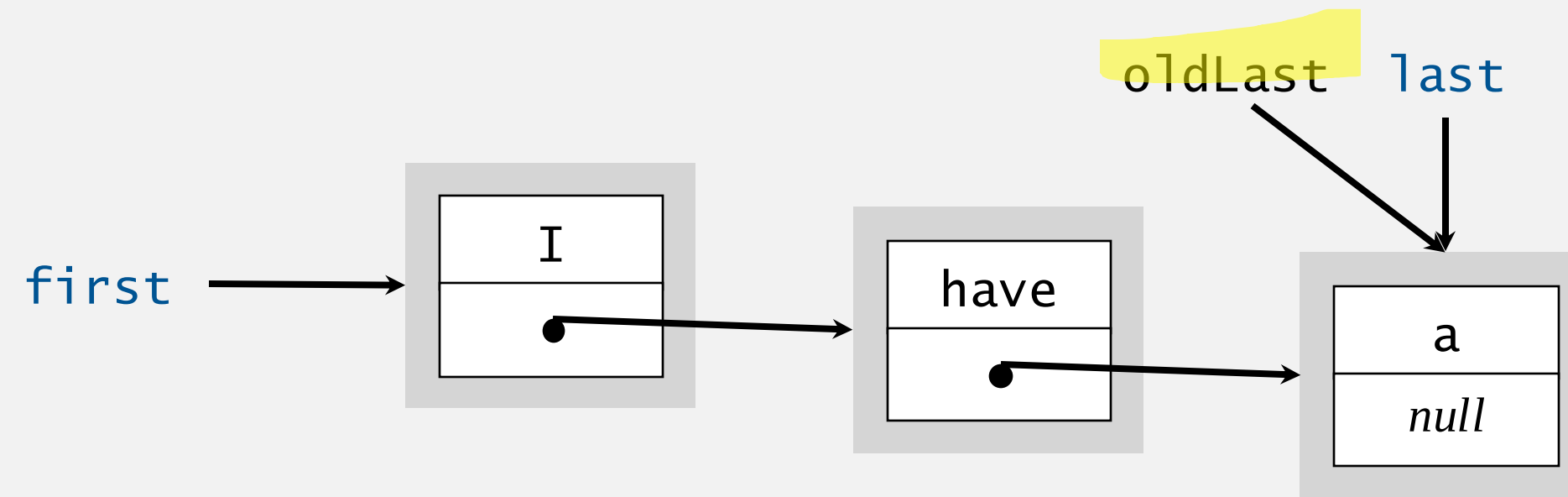
```
inner class
private class Node
{
    private String item;
    private Node next;
}
```

Queue enqueue: linked-list implementation *O(1)*

LO 2.1

save a link to the list

① Node oldLast = last;

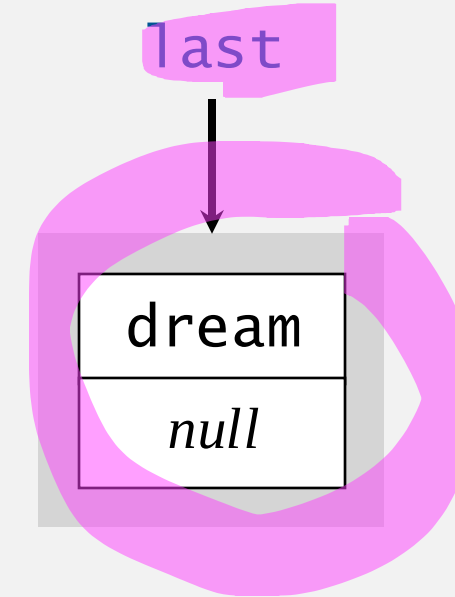
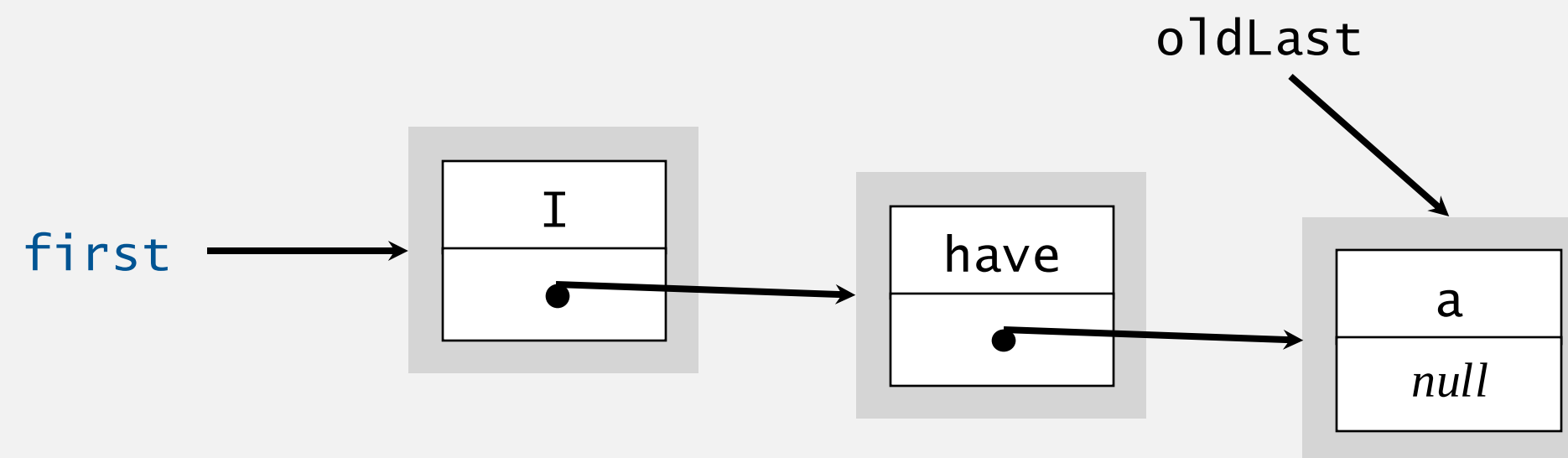


inner class

```
private class Node
{
    private String item;
    private Node next;
}
```

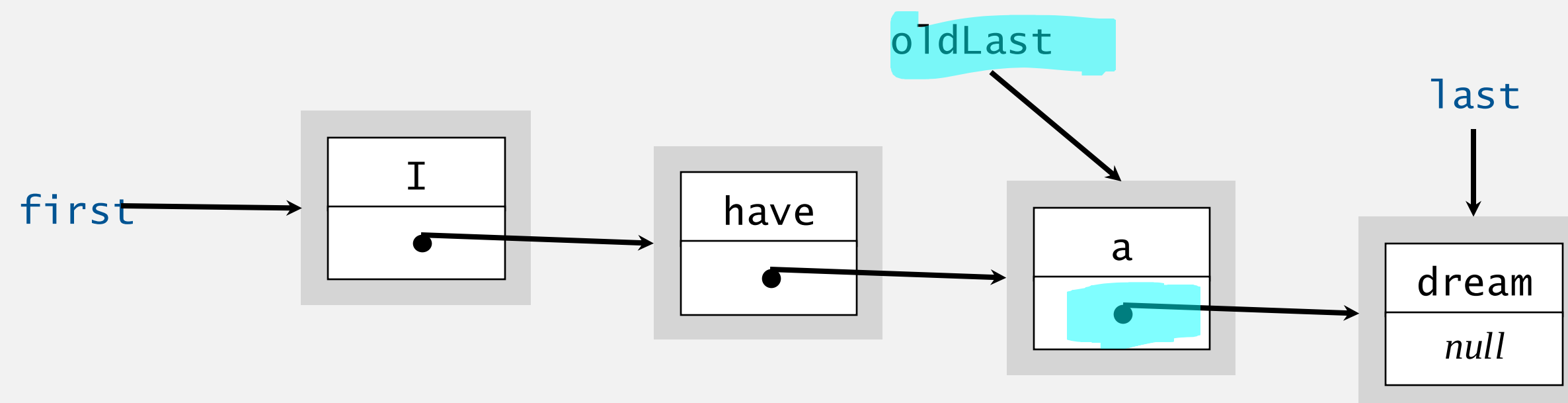
create a new node at the end

② last = new Node();
③ last.item = "dream";



link together

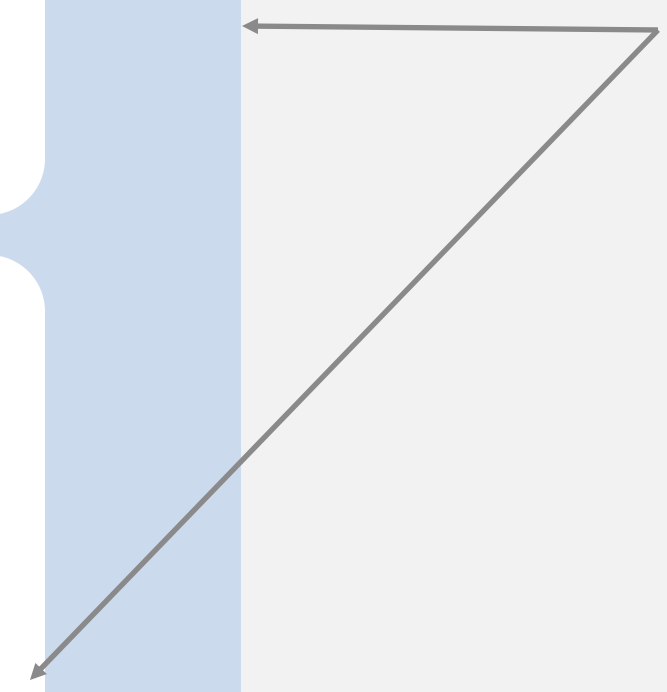
④ oldLast.next = last;



```
public class LinkedQueueOfStrings
{
    private Node first, last;
    private class Node
    { /* same as in LinkedStackOfStrings */ }
    public boolean isEmpty()
    { return first == null; }
    public void enqueue(String item)
    {
        Node oldLast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else oldLast.next = last;
    }

    public String dequeue()
    {
        String item = first.item;
        first = first.next;
        if (isEmpty()) last = null;
        return item;
    }
}
```

corner cases to deal
with empty queue



- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.



Q. How to resize?

QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

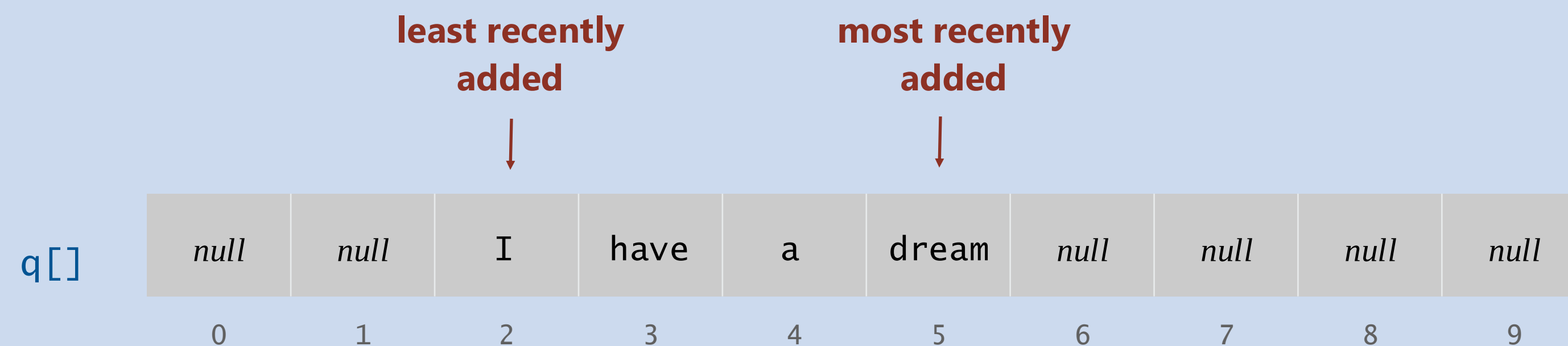
Goal. Implement a **queue** using a **resizing array** so that, starting from an empty queue, any sequence of any sequence of m enqueue and dequeue operations takes $O(m)$ time.

QUEUE: RESIZING-ARRAY IMPLEMENTATION



Goal. Implement a **queue** using a **resizing array** so that, starting from an empty queue, any sequence of any sequence of m enqueue and dequeue operations takes $O(m)$ time.

LO 2.1



QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.



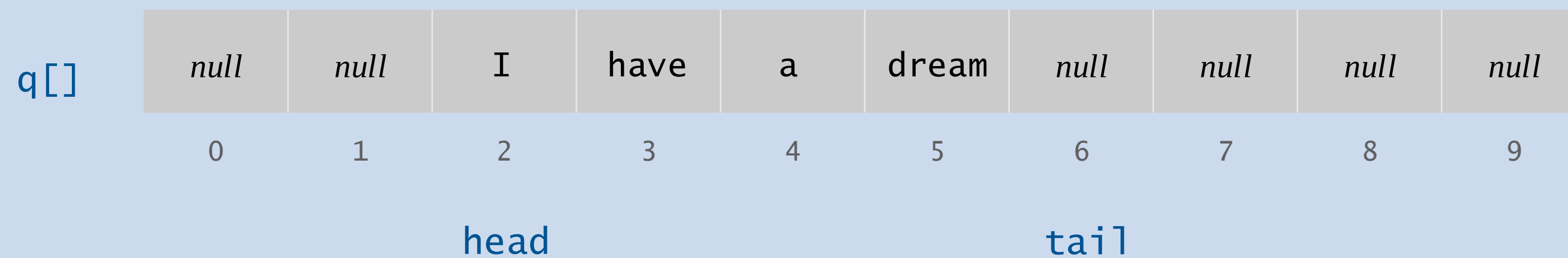
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

enqueue today



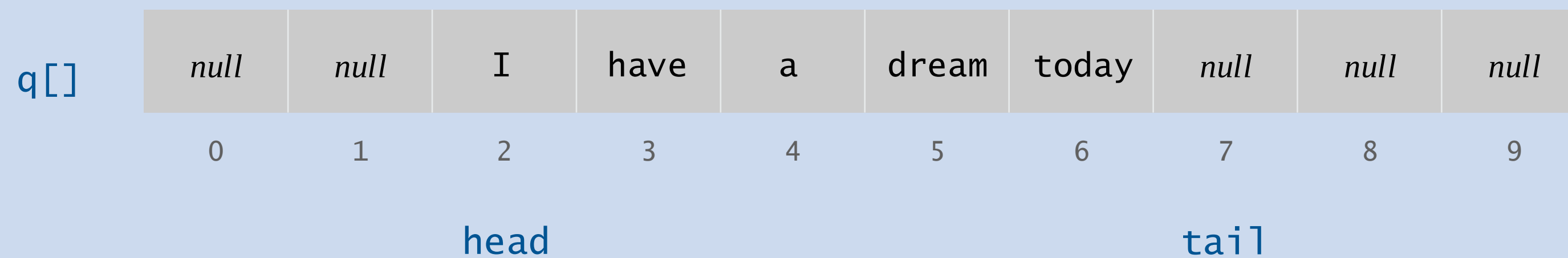
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

enqueue !



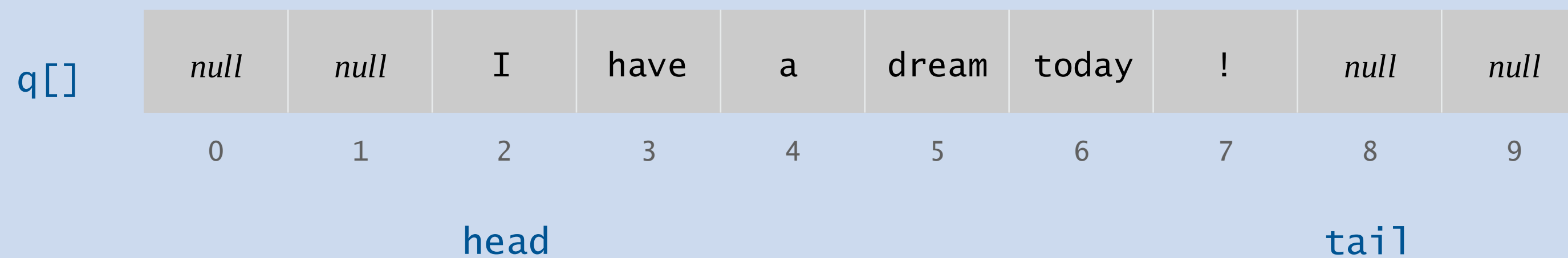
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

dequeue



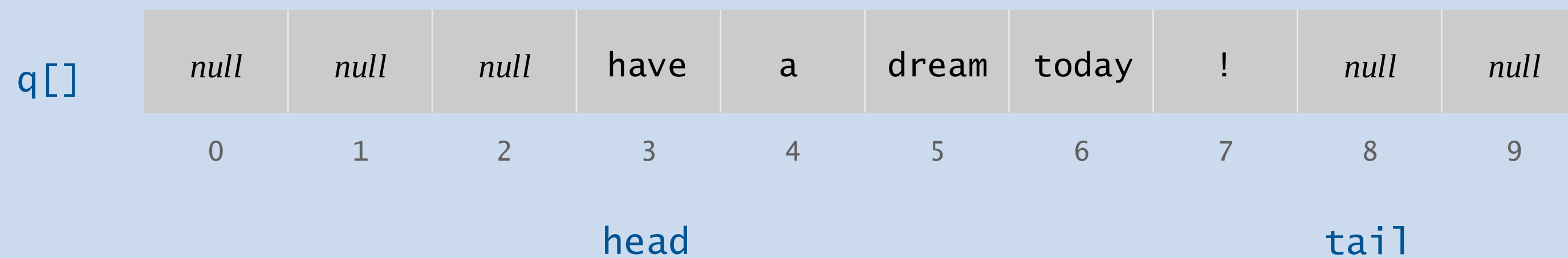
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

dequeue



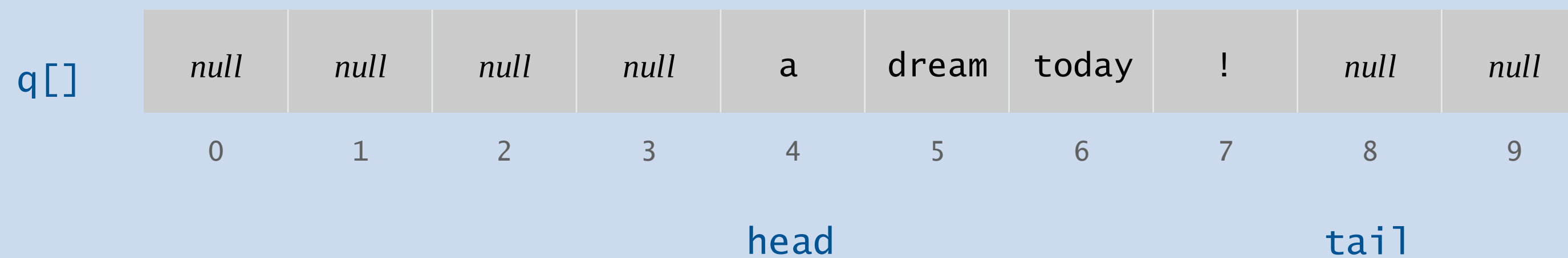
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

enqueue I



have

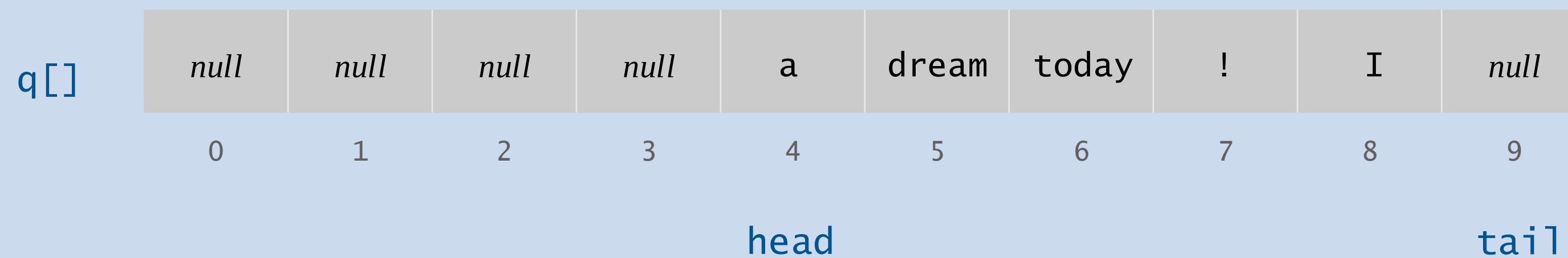
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

enqueue have

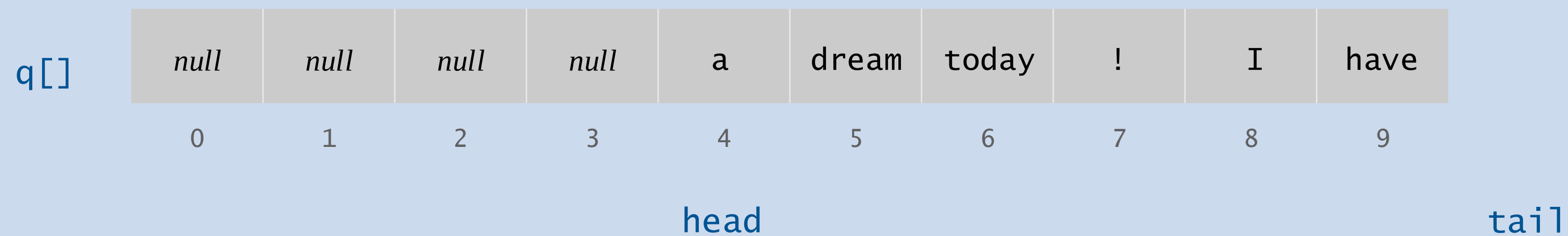


QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.



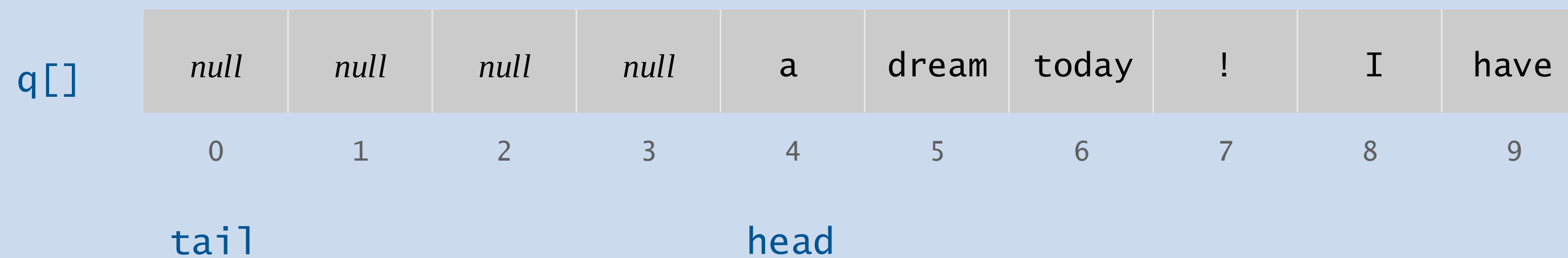
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

enqueue a



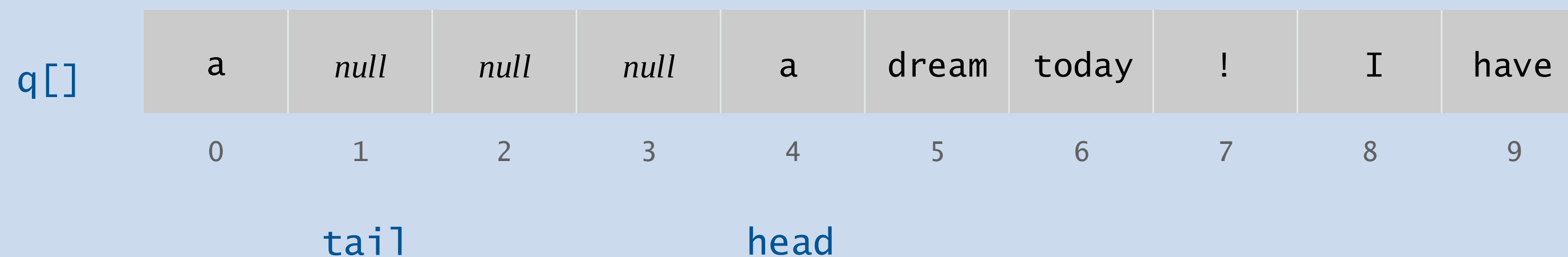
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

Wrap around array



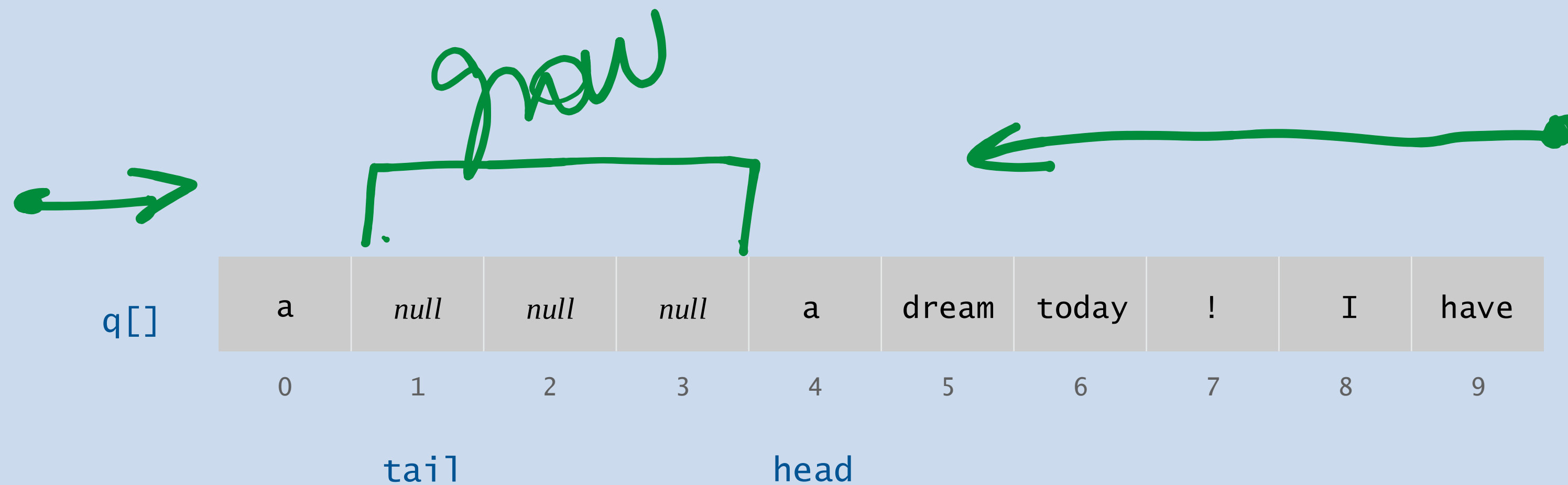
QUEUE: RESIZING-ARRAY IMPLEMENTATION



LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the capacity.

Q. How to resize?



STACK IMPLEMENTATION SUMMARY

LO 2.9

	Pop()	Push()	isEmpty()	Size()	notes
Fixed Array	$O(1)$	$O(1)$	$O(1)^*$	$O(1)^*$	Stack size is limited
Resizable Array	$O(1)$	$O(1)$ on average**	$O(1)^*$	$O(1)^*$	Stack size is unlimited
Linked List	$O(1)$	$O(1)$	$O(1)^*$	$O(1)^*$	Stack size is unlimited

- * Assumes $O(1)$ extra memory to hold the number of elements in the stack
- ** based on amortized analysis (see videos/book for details)

QUEUE IMPLEMENTATION SUMMARY

LO 2.9

	enqueue()	dequeue()	isEmpty()	Size()
Fixed Array	$O(1)^*$	$O(1)^*$	$O(1)^{**}$	$O(1)^{**}$
Resizable Array	$O(1)$ on average ^{***}	$O(1)$ on average ^{***}	$O(1)^{**}$	$O(1)^{**}$
Linked List	$O(1)$	$O(1)$	$O(1)^{**}$	$O(1)^{**}$

- * Assumes pointers to head and tail (circular array)
- ** Assumes $O(1)$ memory to store size of the queue
- *** based on amortized analysis (see videos/book for details)

Arrays and Linked List are some of the most fundamental memory organization methods.

Arrays. Organizes memory as a contiguous block of memory

Linked List. Organizes a linked collection of memory blocks that are drawn from free heap space.

Question. When is it appropriate to select one or the other

Answer. In general,

- if the application stack/queue space requirements are known in advance, an array is a sufficient structure.
 - E.g., use a stack to evaluate a postfix expression
- If the application stack/queue space requirements are not known in advance, it is appropriate to use a linked structure.
 - Use a queue to process data packets flowing through a hub, where data traffic can vary based on the time of the day.

2. STACKS AND QUEUES

- ▶ *Stacks*
- ▶ *Resizing arrays*
- ▶ *Queues*
- ▶ **Generics**
- ▶ *Iterators (optional)*
- ▶ *Applications*

Parameterized stack

We implemented: StackOfStrings.

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

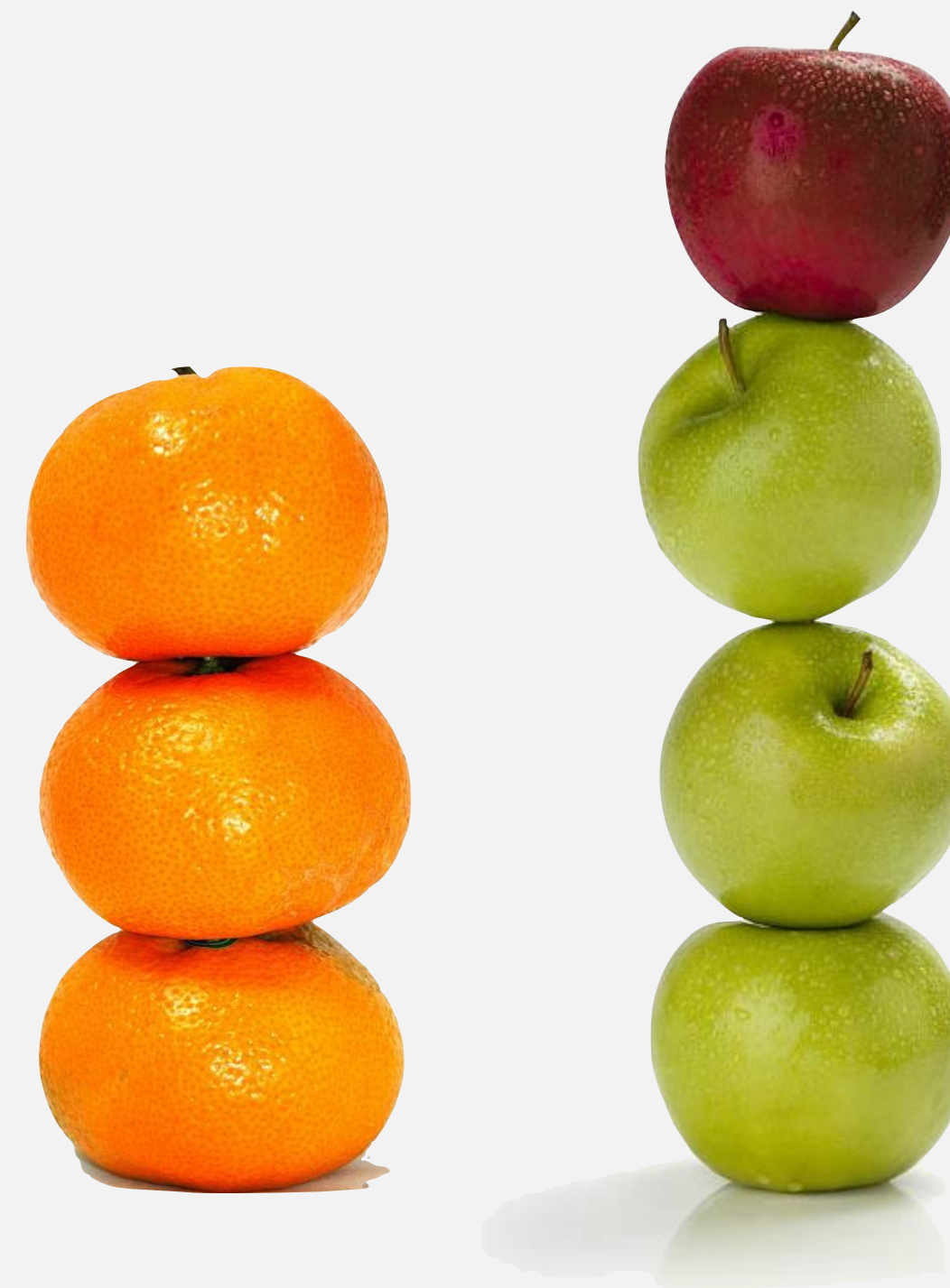
Solution in Java: generics.

type parameter
(use syntax both to specify type and to call constructor)

```
Stack<Apple> stack = new Stack<Apple>();  
Apple apple = new Apple();  
Orange orange = new Orange();  
stack.push(apple);  
stack.push(orange);
```

...

← compile-time error



Generic stack: linked-list implementation

stack of strings (linked list)

```
public class LinkedStackOfStrings
{
    private Node first = null;
    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

generic stack (linked list)

```
public class Stack<Item>
{
    private Node first = null;

    private class Node
    {
        Item item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(Item item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

generic type name



Generic stack: array implementation

The way it should be.

stack of strings (fixed-length array)

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

    public ...StackOfStrings(int capacity)
    { s = new String[capacity]; }

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

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

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

Generic stack of strings (fixed-length array)

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

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

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

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

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

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

Generic stack: array implementation

The way it should be.

stack of strings (fixed-length array)

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

    public ...StackOfStrings(int capacity)
    { s = new String[capacity]; }

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

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

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

generic stack (fixed-length array)

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

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

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

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

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

← the ugly cast

Unchecked cast

```
~/Desktop/queues> javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
    s = (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**.





Which of the following is a correct way to declare and initialize an empty stack of integers?

- A. `Stack stack = new Stack<int>();`
- B. `Stack<int> stack = new Stack();`
- C. `Stack<int> stack = new Stack<int>();`

D. *None of the above.* **Need the wrapper class Integer**

Generic data types: autoboxing and unboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a “**wrapper**” reference type.
- Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast from primitive type to wrapper type.

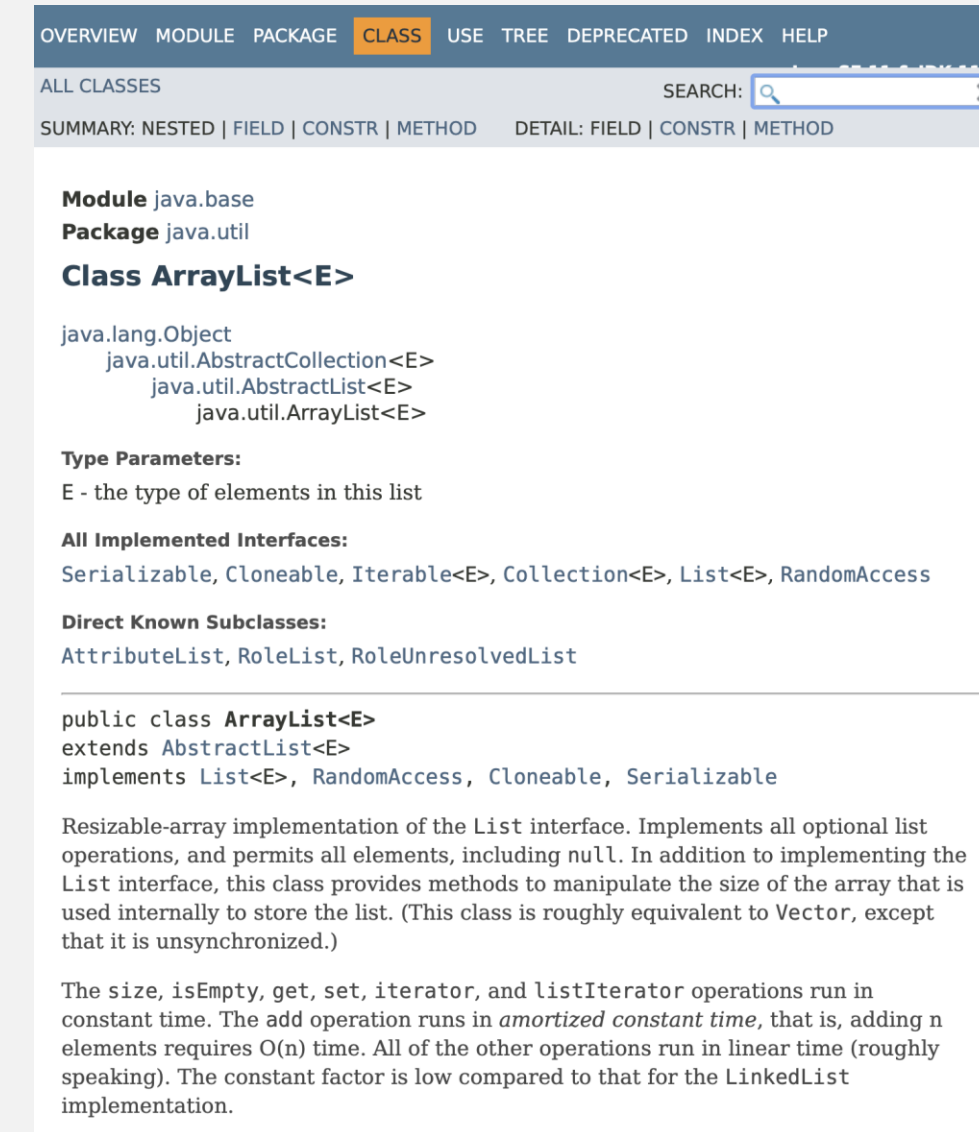
Unboxing. Automatic cast from wrapper type to primitive type.

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

Bottom line. Client code can use generic stack for **any** type of data.
(but substantial overhead for primitive types)

Java's library of collection data types.

- `java.util.ArrayList` [resizing array]
- `java.util.LinkedList` [doubly linked list]
- `java.util.ArrayDeque`



OVERVIEW MODULE PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

ALL CLASSES SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util
Class `ArrayList<E>`

java.lang.Object
java.util.AbstractCollection<E>
java.util.AbstractList<E>
java.util.ArrayList<E>

Type Parameters:
E - the type of elements in this list

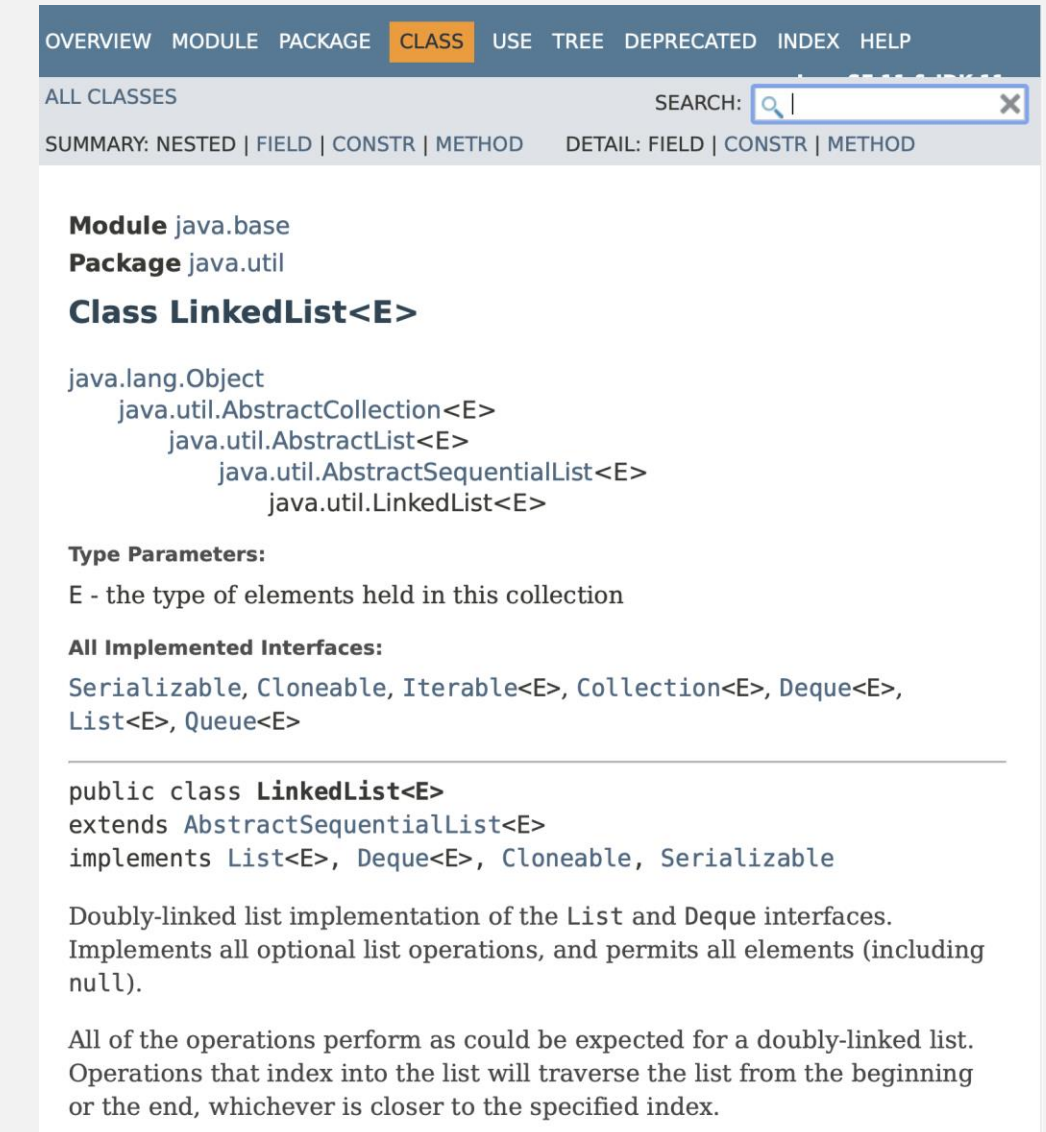
All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:
AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>  
    extends AbstractList<E>  
    implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.



OVERVIEW MODULE PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

ALL CLASSES SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util
Class `LinkedList<E>`

java.lang.Object
java.util.AbstractCollection<E>
java.util.AbstractList<E>
java.util.AbstractSequentialList<E>
java.util.LinkedList<E>

Type Parameters:
E - the type of elements held in this collection

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>  
    extends AbstractSequentialList<E>  
    implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

This course. Implement from scratch (once).

Beyond. Basis for understanding performance guarantees.

Best practices.

- Use our Stack and Queue for stacks and queues to improve design and efficiency.
- Use Java's ArrayList or LinkedList when other ops needed (but remember that some ops are inefficient).

Stacks and queues summary

Fundamental data types.

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

Stack. Examine the item most recently added (LIFO).

Queue. Examine the item least recently added (FIFO).



Efficient implementations.

- Singly linked list.
- Resizing array.

Next time. Advanced Java (including **iterators** for collections).

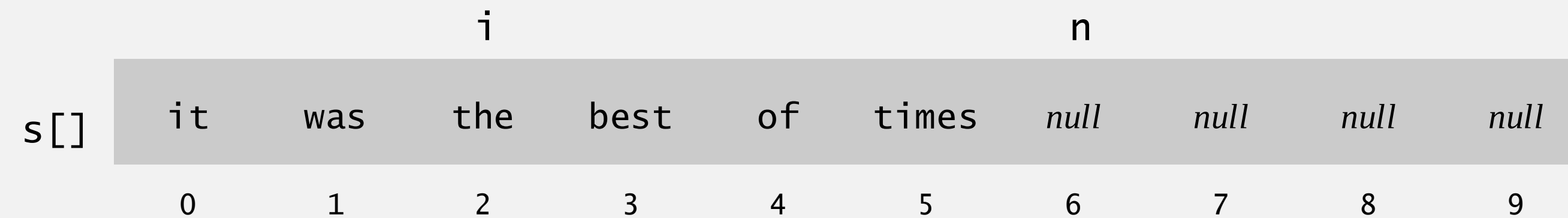
2. STACKS AND QUEUES

- ▶ *Stacks*
- ▶ *Resizing arrays*
- ▶ *Queues*
- ▶ *Generics*
- ▶ *Iterators (optional)*
- ▶ *Applications*

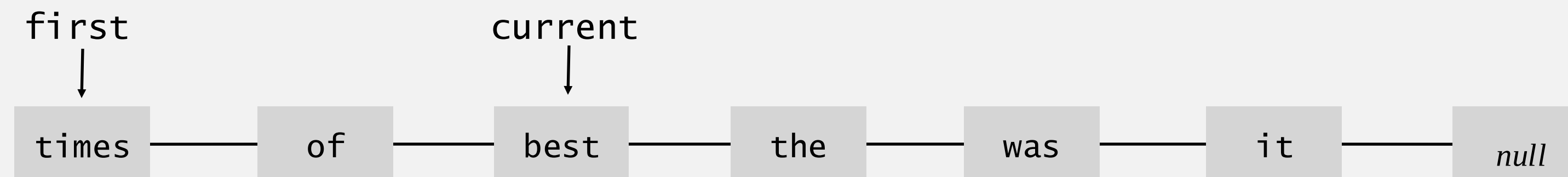
Iteration

Design challenge. Allow client to iterate over the stack items, without exposing the internal representation of the stack.

resizing-array representation



linked-list representation



Java solution. Use a **foreach** loop.

Foreach loop

Java provides elegant syntax for iterating over items in a collection.

"foreach" loop (shorthand)

```
Stack<String> stack;  
...  
  
for (String s : stack)  
    ...
```

equivalent code (longhand)

```
Stack<String> stack;  
...  
  
Iterator<String> i = stack.iterator();  
while (i.hasNext())  
{  
    String s = i.next();  
} ...  
}
```

To make user-defined collection support foreach loop:

- Data type must have a method named `iterator()`.
- The `iterator()` method returns an object that has two core methods:
 - the `hasNext()` method returns `false` when there are no more items
 - the `next()` method returns the next item in the collection

Iterators

To support foreach loops, Java provides two interfaces.

- Iterator interface: `next()` and `hasNext()` methods.
- Iterable interface: `iterator()` method that returns an Iterator.
- Both should be used with generics.

java.util.Iterator interface

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
}
```

java.lang.Iterable interface

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

Type safety.

- Implementation must use these interfaces to support foreach loop.
- Client program won't compile unless implementation do.

Stack iterator: linked-list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

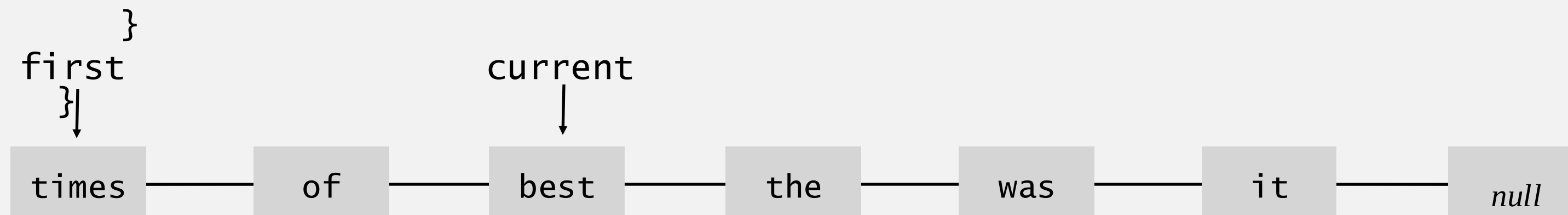
    public Iterator<Item> iterator() { return new LinkedIterator(); }

    private class LinkedIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() { return current != null; }

        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

throw NoSuchElementException
if no more items in iteration



Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = n;

        public boolean hasNext() { return i > 0; }
        public Item next()      { return s[--i]; }
    }
}
```

	i					n				
s[]	it	was	the	best	of	times	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
	0	1	2	3	4	5	6	7	8	9

ITERATION: CONCURRENT MODIFICATION



Q. What if client modifies the data structure while iterating?

A. A **fail-fast iterator** throws a `java.util.ConcurrentModificationException`.

concurrent modification

```
for (String s : stack)
    stack.push(s);
```

Q. How to detect concurrent modification?

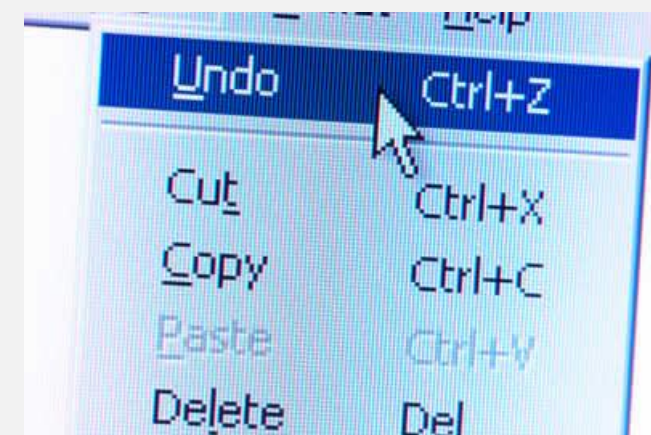
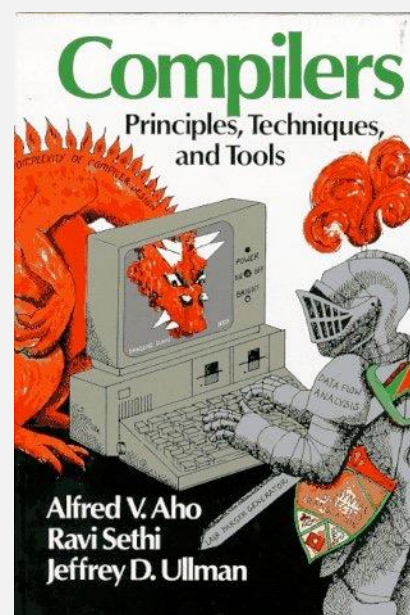
A.

2. STACKS AND QUEUES

- ▶ *Stacks*
- ▶ *Resizing arrays*
- ▶ *Queues*
- ▶ *Generics*
- ▶ *Iterators (optional)*
- ▶ *Applications*

Stack applications

- Java virtual machine.
- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Function-call stack during execution of a program.
- ...



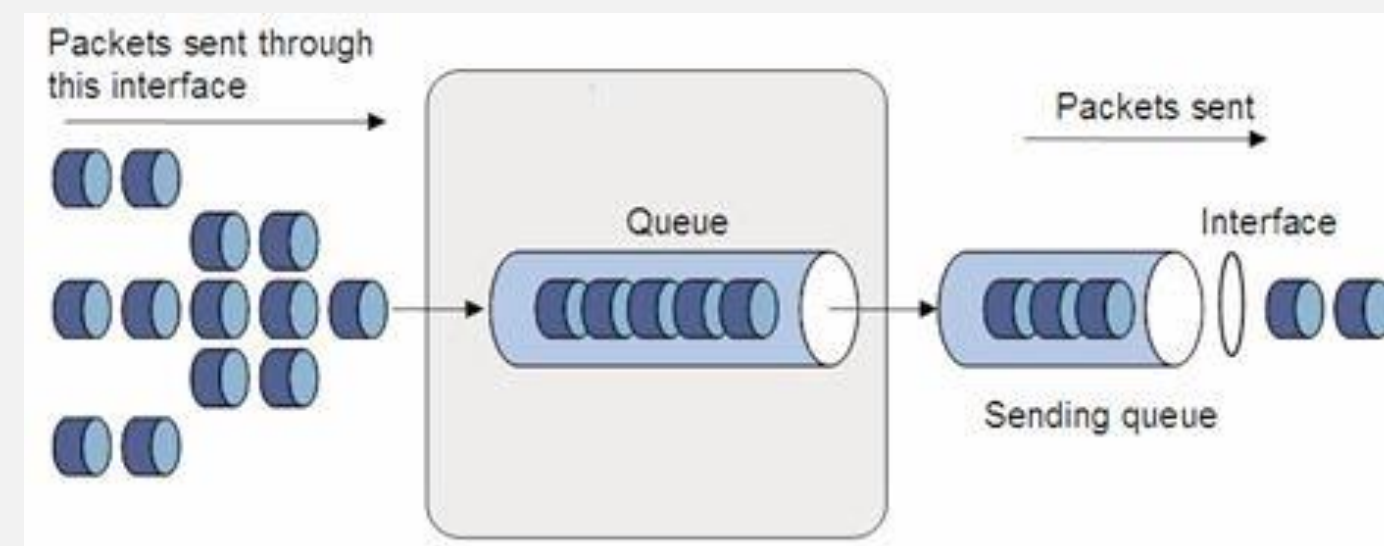
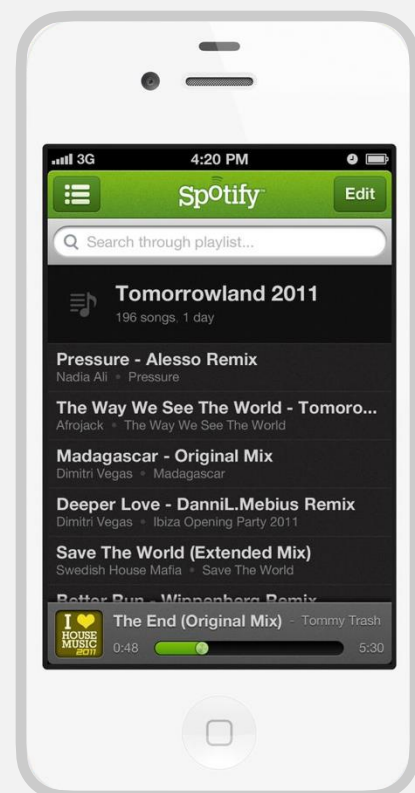
Queue applications

Familiar applications.

- Spotify playlist.
- Data buffers (iPod, TiVo, sound card, streaming video, ...).
- Asynchronous data transfer (file I/O, pipes, sockets, ...).
- Dispensing requests on a shared resource (printer, processor, ...).

Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.



List interface. `java.util.List` is API for a sequence of items.

```
public interface List<Item> extends Iterable<Item>
```

<code>List()</code>	<i>create an empty list</i>
---------------------	-----------------------------

<code>boolean isEmpty()</code>	<i>is the list empty?</i>
--------------------------------	---------------------------

<code>int size()</code>	<i>number of items</i>
-------------------------	------------------------

<code>void add(Item item)</code>	<i>add item to the end</i>
----------------------------------	----------------------------

<code>Iterator<Item> iterator()</code>	<i>iterator over all items in the list</i>
--	--

<code>Item get(int index)</code>	<i>return item at given index</i>
----------------------------------	-----------------------------------

<code>Item remove(int index)</code>	<i>return and delete item at given index</i>
-------------------------------------	--

<code>boolean contains(Item item)</code>	<i>does the list contain the given item?</i>
--	--

<code>⋮</code>	
----------------	--

Implementations. `java.util.ArrayList` uses a resizing array;

`java.util.LinkedList` uses a doubly linked list.

↖ ↗
Caveat: not all operations are efficient!

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.



Java 1.3 bug report (June 27, 2001)

The `iterator` method on `java.util.Stack` iterates through a `Stack` from the bottom up. One would think that it should iterate as if it were popping off the top of the `Stack`.



status (closed, will not fix)

It was an incorrect design decision to have `Stack` extend `Vector` ("is-a" rather than "has-a"). We sympathize with the submitter but cannot fix this because of compatibility.

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.



`java.util.Queue`. An interface, not an implementation of a queue.

Best practices. Use our `Stack` and `Queue` for stacks and queues;
use `java.util.ArrayList` or `java.util.LinkedList` when appropriate.

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

R u t g e r s U n i v e r s i t y

STACKS AND QUEUES

- ▶ *Stacks*
- ▶ *Resizing arrays*
- ▶ *Queues*
- ▶ *Generics*
- ▶ *Iterators (optional)*
- ▶ *Applications*

