https://algs4.cs.princeton.edu

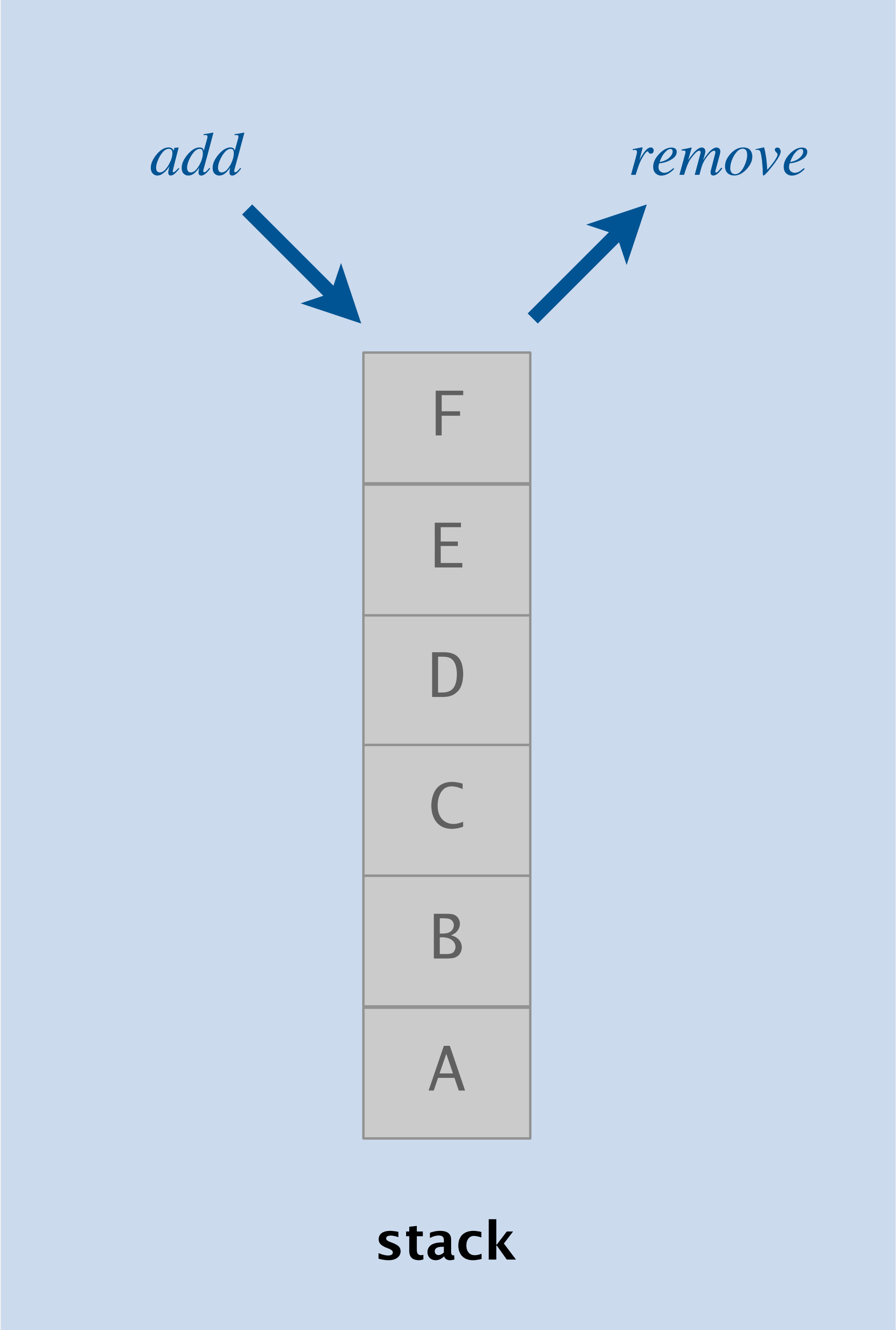
1.3 Stacks and Queues

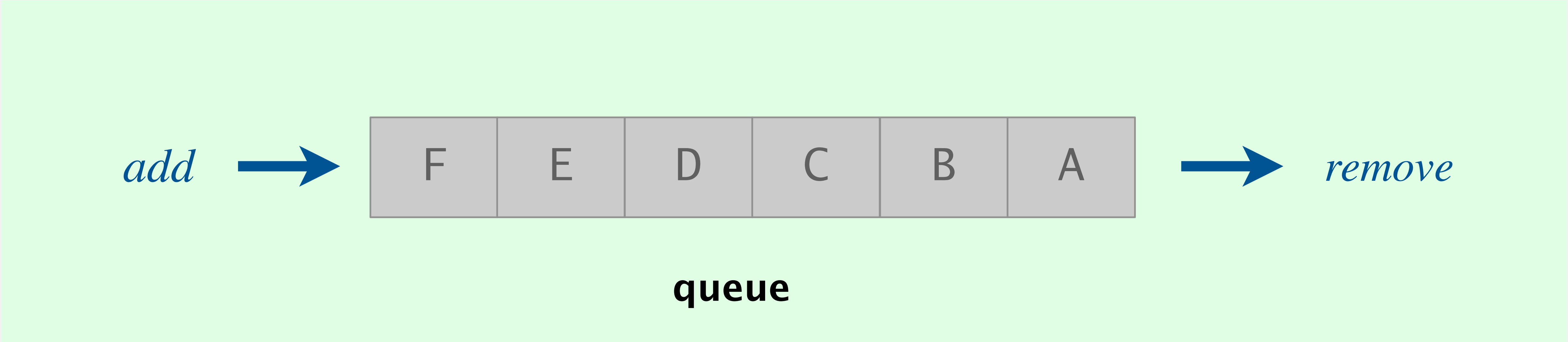
- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *iterators* ← see next lecture and precept

Stacks and queues

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"

Programming assignment 2

Deque. Remove either most recently or least recently added item.

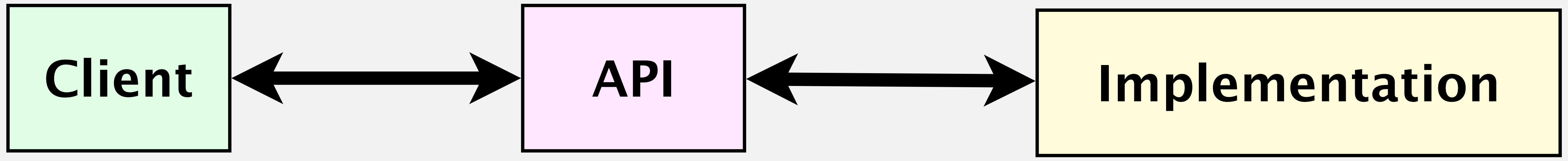
Randomized queue. Remove a random item.

Your job.

- Choose a data structure that meets the performance requirements.
- Implement it from scratch.

Client, implementation, API

Separate client and implementation via API.



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, ...

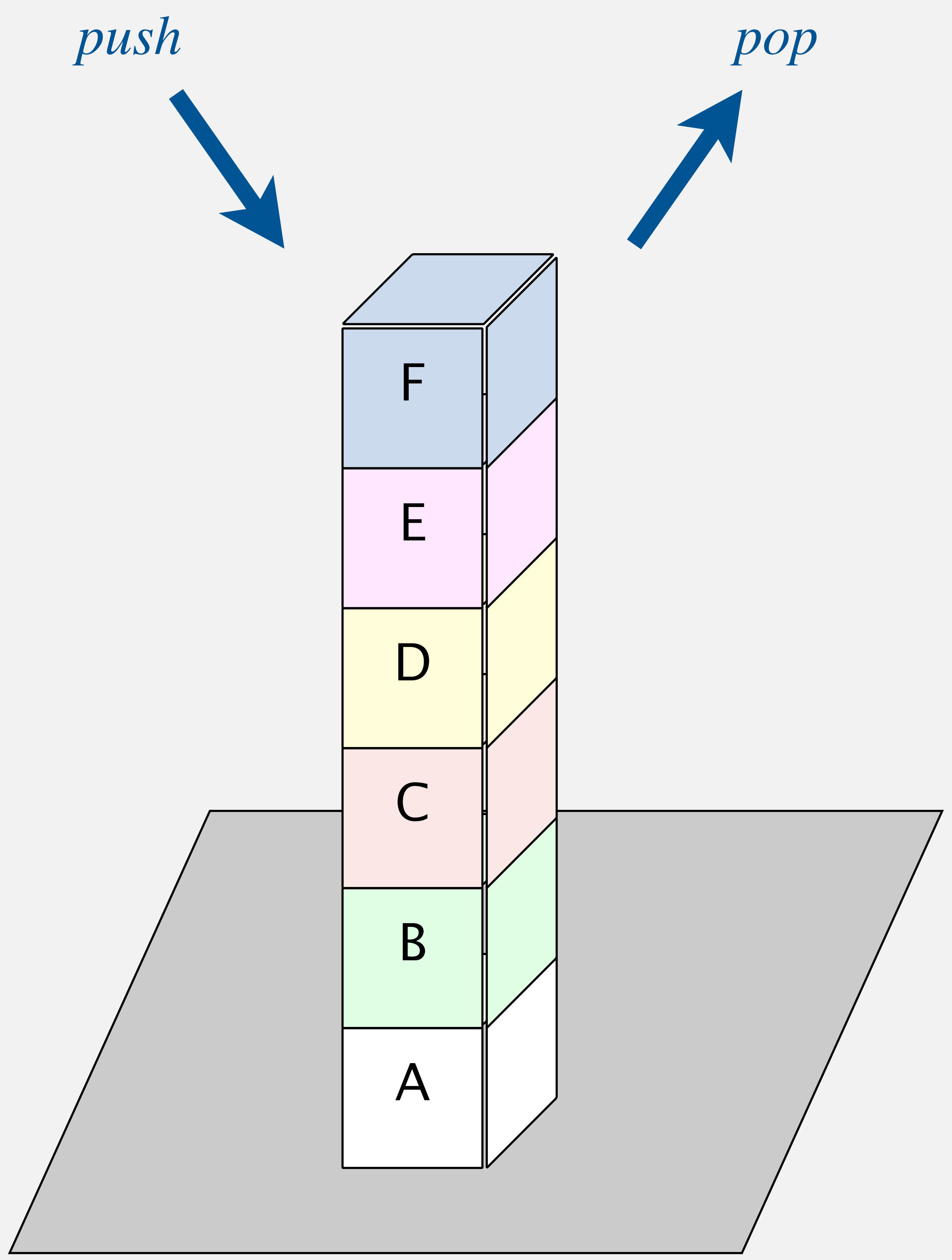
https://algs4.cs.princeton.edu

1.3 Stacks and Queues

- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *iterators*

Stack API

Warmup API. Stack of strings data type.



public class StackOfStrings		
	`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 $\Theta(1)$ time.

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

Function-call stack demo

function arguments
supplied by operating system

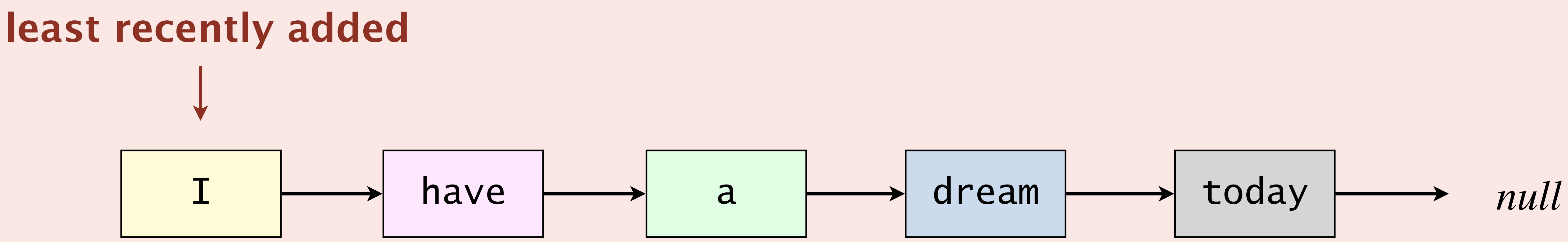
```
public static void main(String[] args) {
   double a = Double.parseDouble(args[0]);
   double b = Double.parseDouble(args[1]);
   double c = hypotenuse(a, b);
}
```

main()

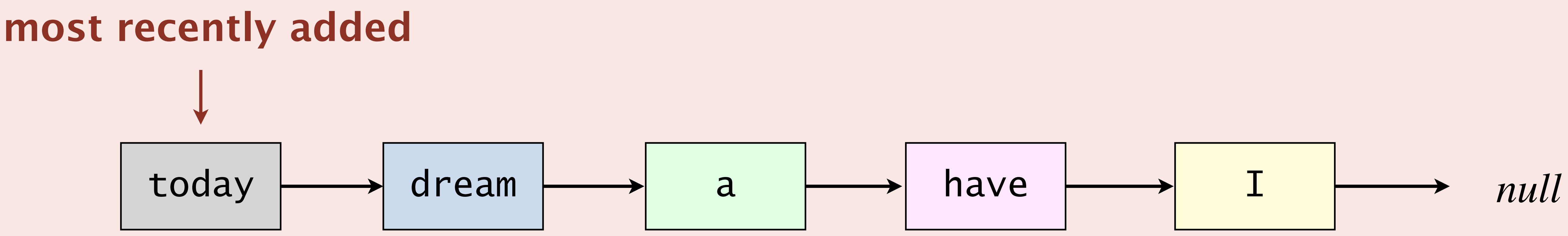
function-call stack

How to implement efficiently a stack with a singly linked list?

A.



B.

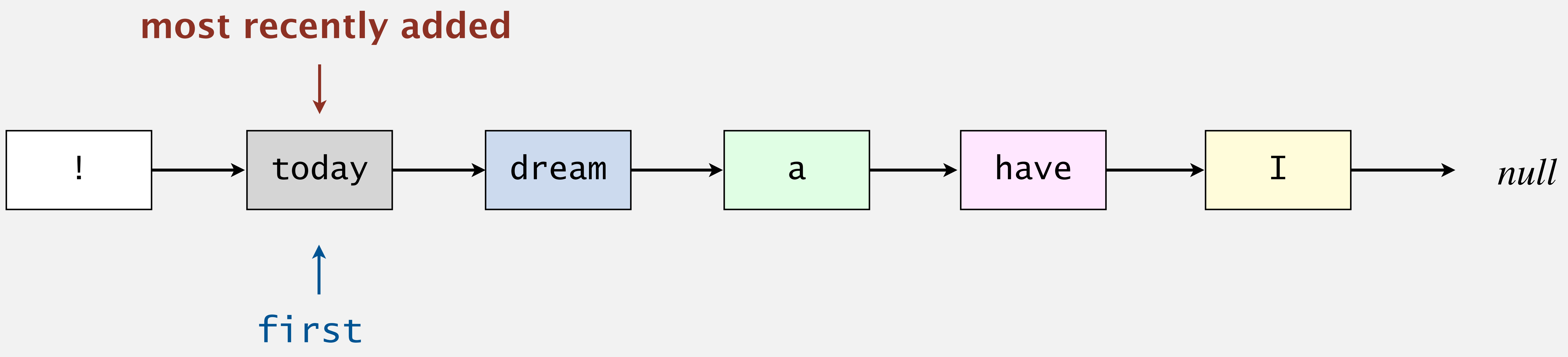


C. *Both A and B.*

D. *Neither A nor B.*

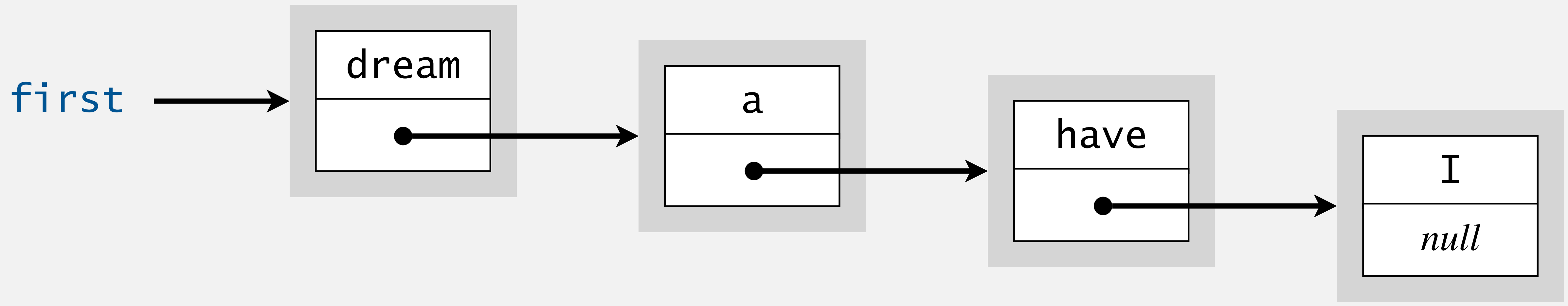
Stack: linked-list implementation

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



Stack pop: linked-list implementation

singly linked list



inner class

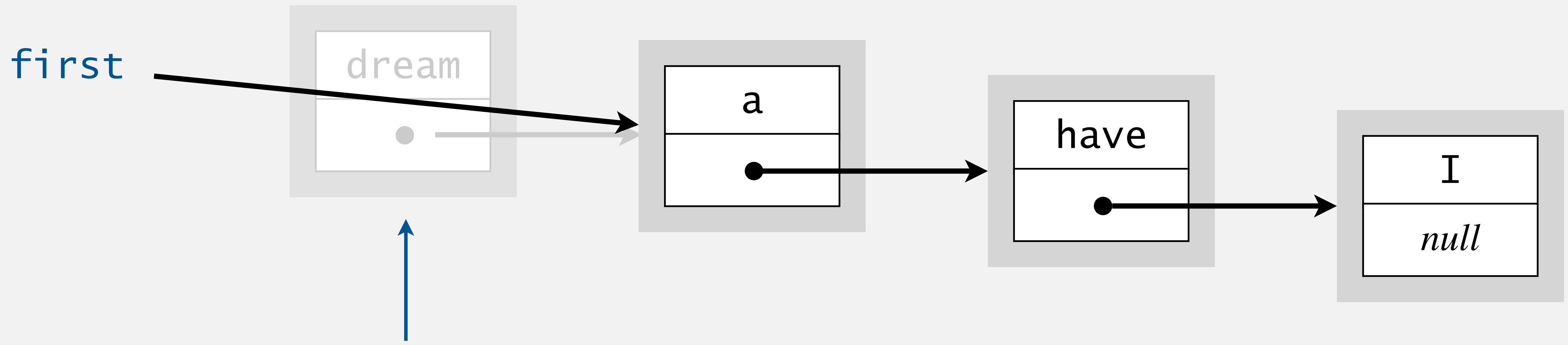
```
private class Node
{
   private String item;
   private Node next;
}
```

save item to return

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

delete first node

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



garbage collector reclaims memory
when no remaining references

return saved item

```
return item;
```

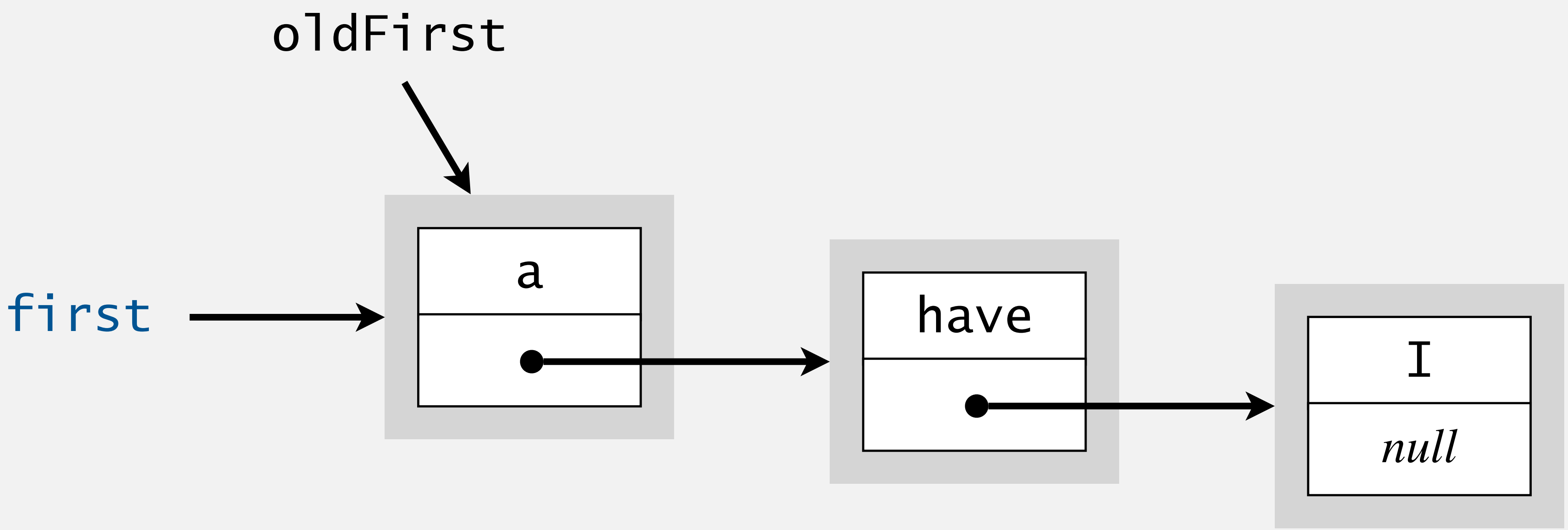
Stack push: linked-list implementation

inner class

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

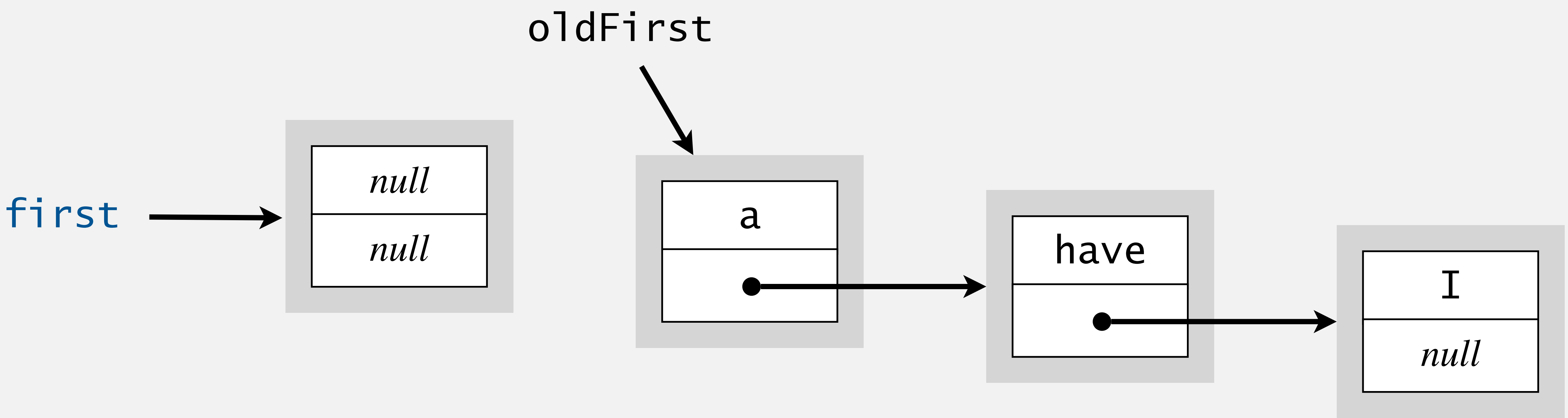
save a link to the list

```
Node oldFirst = first;
```



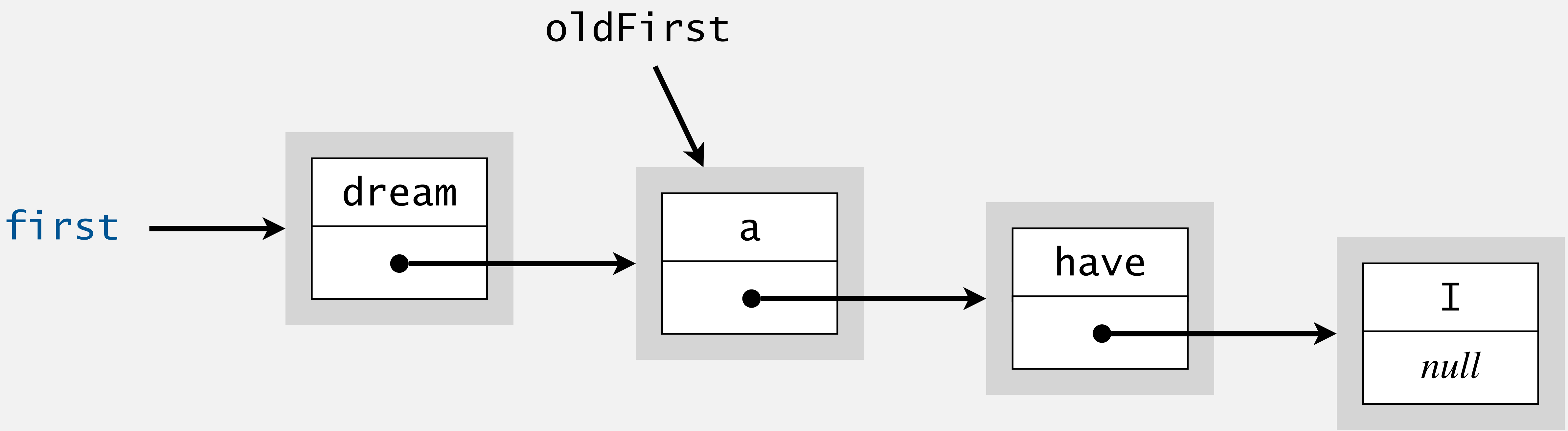
create a new node for the beginning

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



initialize the instance variables in the new Node

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



Stack: linked-list implementation

```java
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;
   }
}
```

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

no Node constructor defined ⇒
Java supplies default no-argument constructor

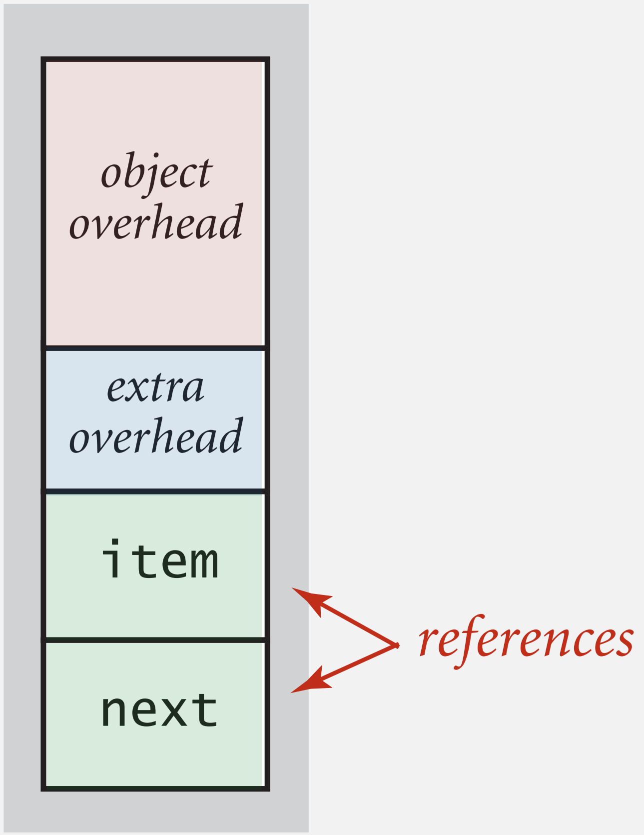
Stack: linked-list implementation performance

Proposition. Every operation takes $\Theta(1)$ 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 (including string references).

(not the memory for the strings themselves, which the client owns)

Stacks and queues: quiz 2

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

A.

least recently added ↓

I	have	a	dream	today	!	*null*	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

B.

most recently added ↓

!	today	dream	a	have	I	*null*	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

C. *Both A and B.*

D. *Neither A nor B.*

Fixed-capacity stack: array implementation

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

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

stack overflow

Fixed-capacity stack: array implementation

```
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];  }
}
```

a cheat
(stay tuned)

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

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

Stack considerations

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

https://algs4.cs.princeton.edu

1.3 Stacks and Queues

- stacks
- ***resizing arrays***
- queues
- generics
- iterators

Stack: resizing-array implementation

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 = $n + (2 + 4 + 6 + \ldots + 2(n-1)) \sim n^2$.

infeasible for large n

1 array access per push

$2(k-1)$ array accesses to expand to size k (ignoring cost to create new array)

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

Q. How to grow array?

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

"repeated doubling"

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

feasible for large n

1 array access per push

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

Stack: resizing-array implementation

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 $\Theta(n)$ time.

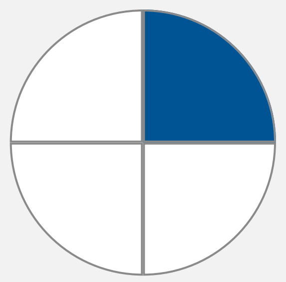
full	I	have	a	dream				
push("today")	I	have	a	dream	today	*null*	*null*	*null*
pop()	I	have	a	dream				
push("today")	I	have	a	dream	today	*null*	*null*	*null*

Stack: resizing-array implementation

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;
}
```

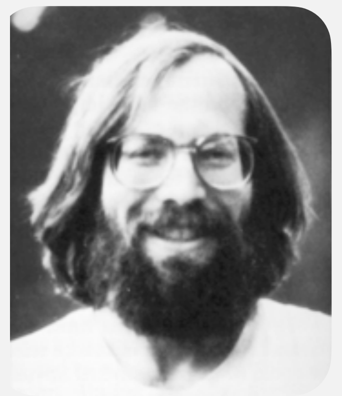
Invariant. Array is between 25% and 100% full.

Stack resizing-array implementation: performance

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

Stack resizing-array implementation: memory usage

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 (including string references).

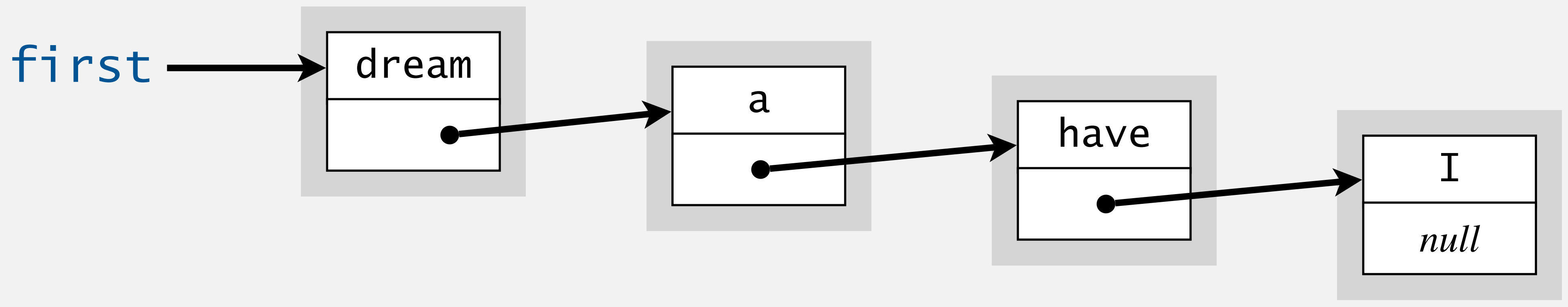
(not the memory for the strings, which the client allocates)

Stack implementations: resizing array vs. linked list

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

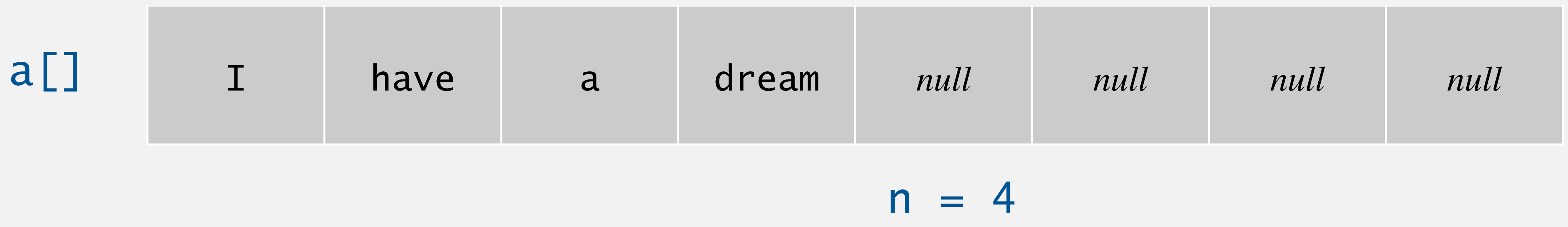
Linked-list implementation.

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



Resizing-array implementation.

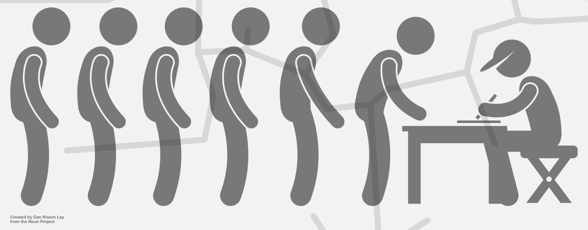
- Weaker performance guarantee: $\Theta(1)$ amortized.
- Less memory.
- Better use of cache.



https://algs4.cs.princeton.edu

1.3 Stacks and Queues

- *stacks*
- *resizing arrays*
- ***queues***
- *generics*
- *iterators*



Queue of strings API

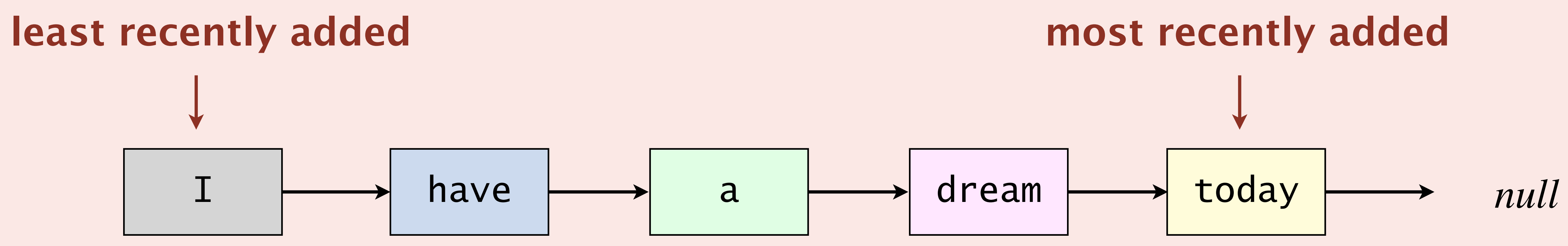
`public class QueueOfStrings`		
	`QueueOfStrings()`	*create an empty queue*
`void`	`enqueue(String item)`	*add a new string to queue*
`String`	`dequeue()`	*remove and return the string least recently added*
`boolean`	`isEmpty()`	*is the queue empty?*
`int`	`size()`	*number of strings on the queue*

Performance goal. Every operation takes $\Theta(1)$ time.

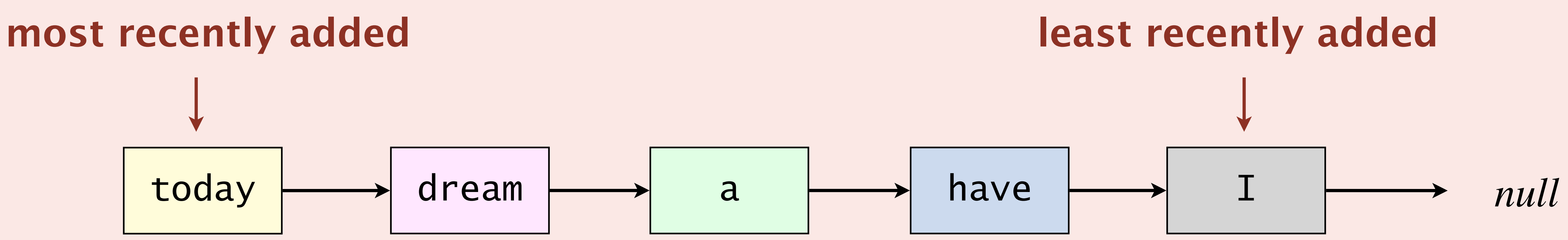
Stacks and queues: quiz 3

How to implement efficiently a queue with a singly linked list?

A.



B.

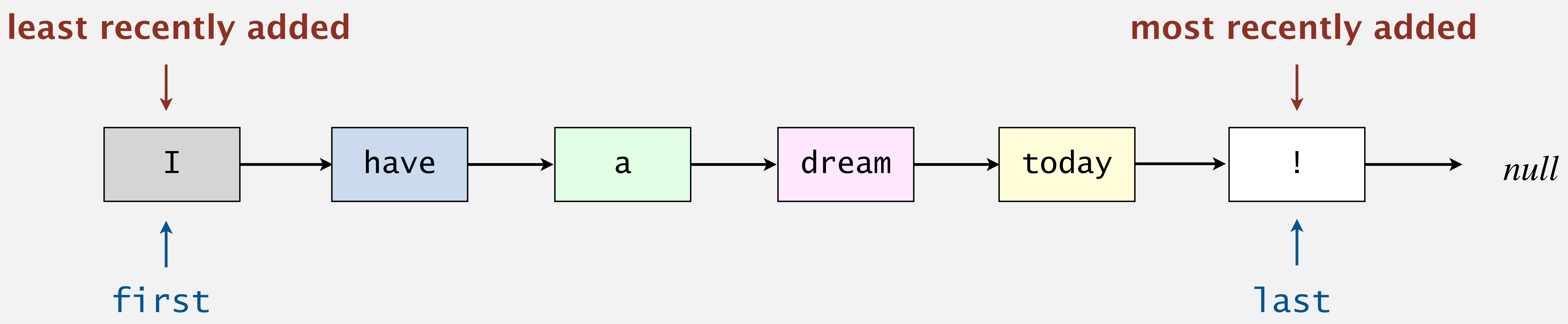


C. *Both A and B.*

D. *Neither A nor B.*

Queue: linked-list implementation

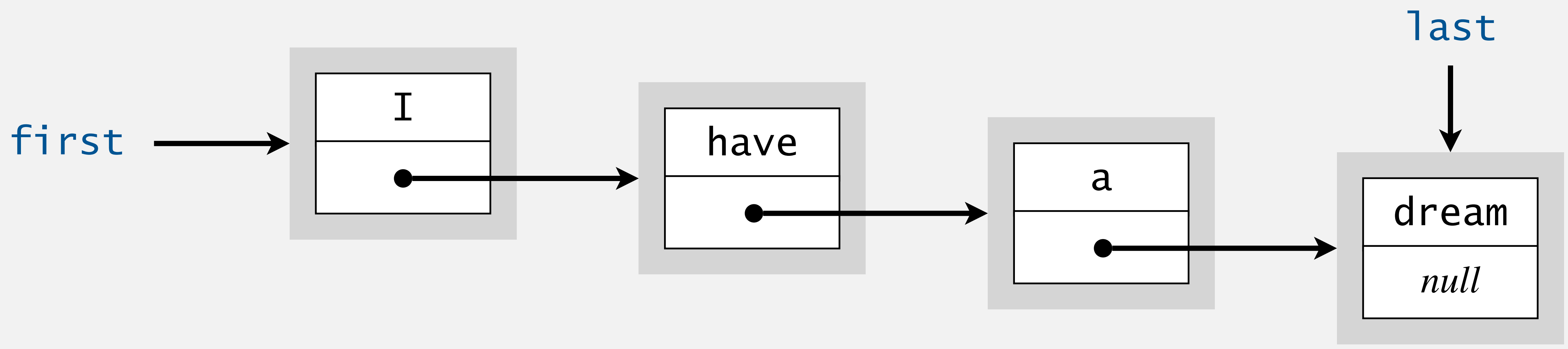
- Maintain one pointer `first` to first node in a singly linked list.
- Maintain another pointer `last` to last node.
- Dequeue from `first`.
- Enqueue after `last`.



Queue dequeue: linked-list implementation

Code is identical to pop().

singly linked list

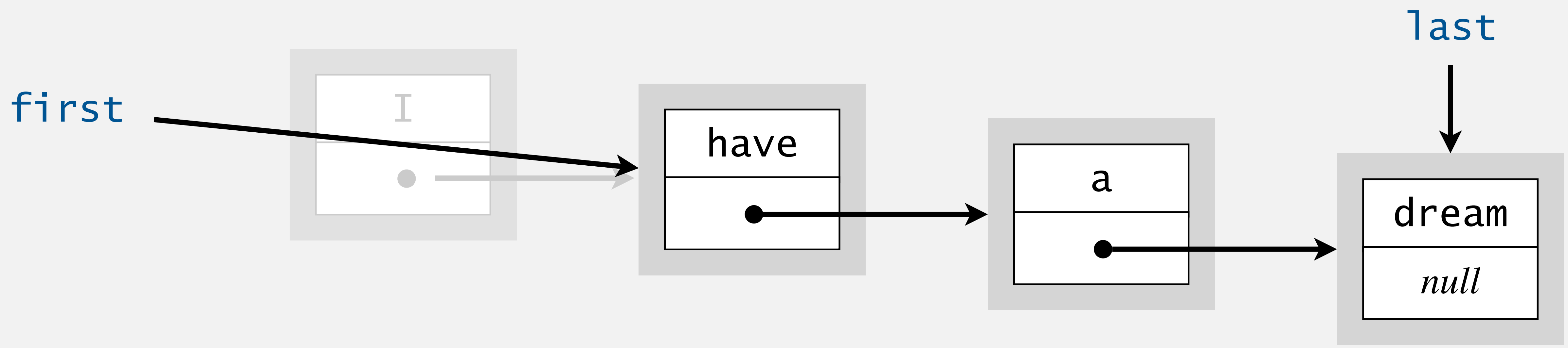


save item to return

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

delete first node

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



return saved item

```
return item;
```

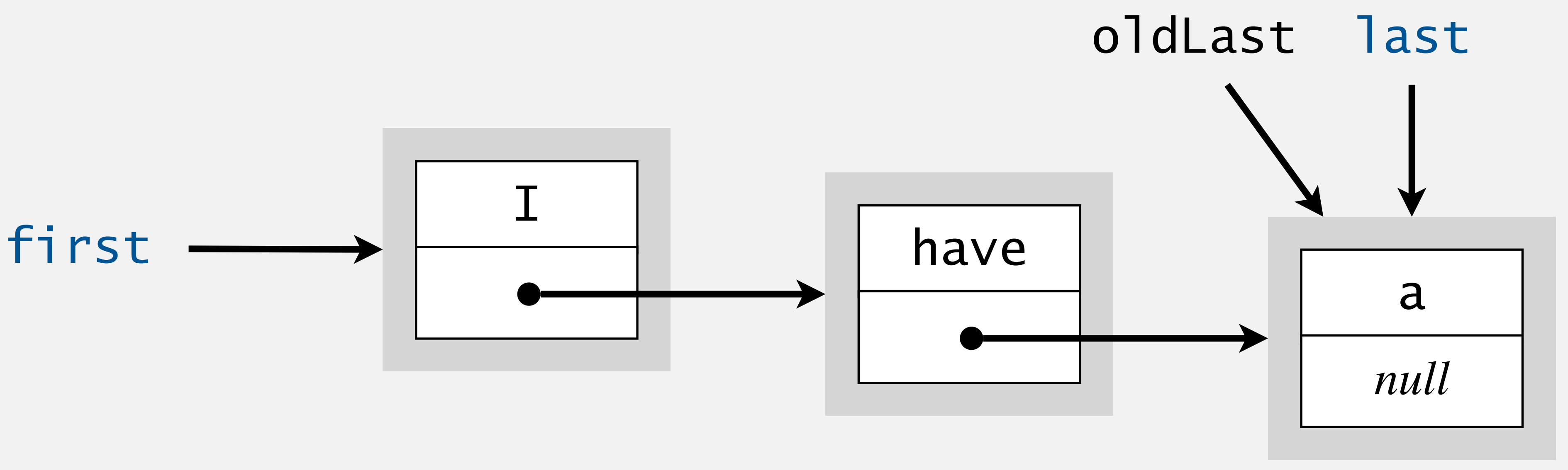
inner class

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

Queue enqueue: linked-list implementation

save a link to the list

```
Node oldLast = last;
```

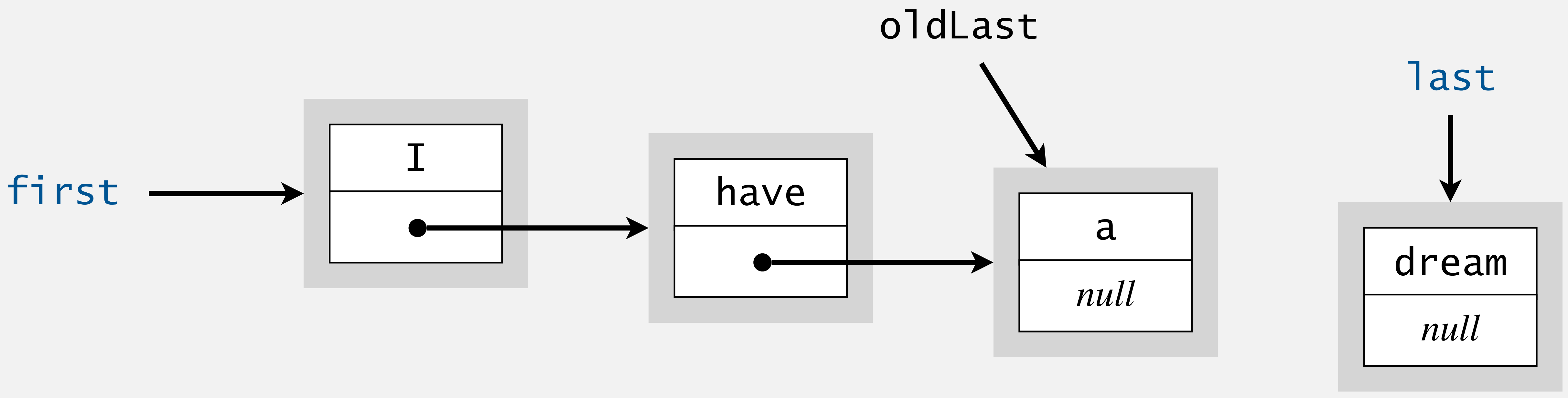


inner class

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

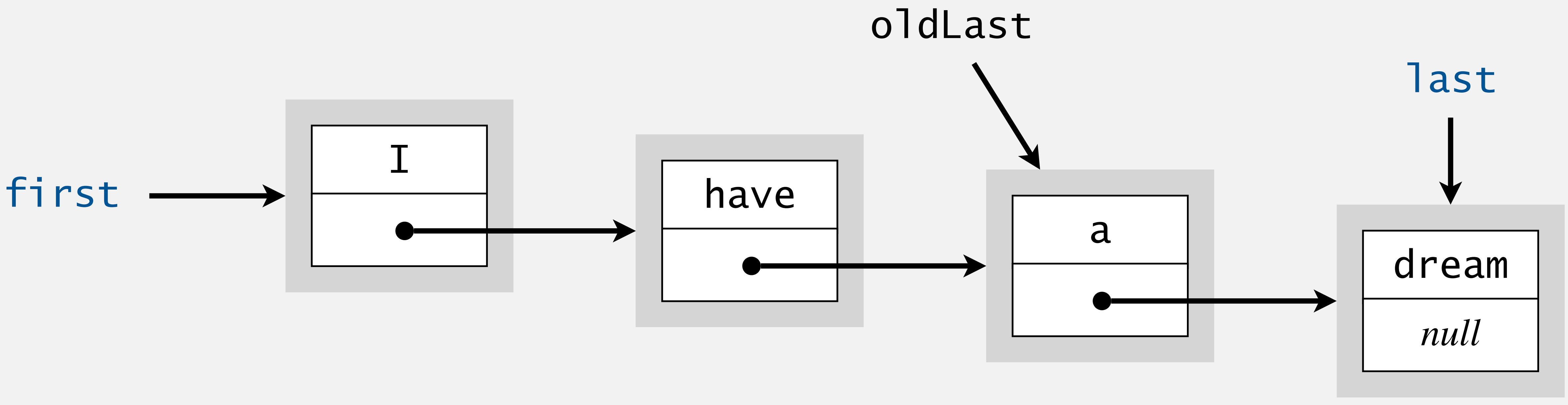
create a new node at the end

```
last = new Node();
last.item = "dream";
```



link together

```
oldLast.next = last;
```



Queue: linked-list implementation

```java
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

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 $\Theta(m)$ time.

https://algs4.cs.princeton.edu

1.3 Stacks and Queues

- stacks
- resizing arrays
- queues
- **generics**
- iterators

Parameterized stack

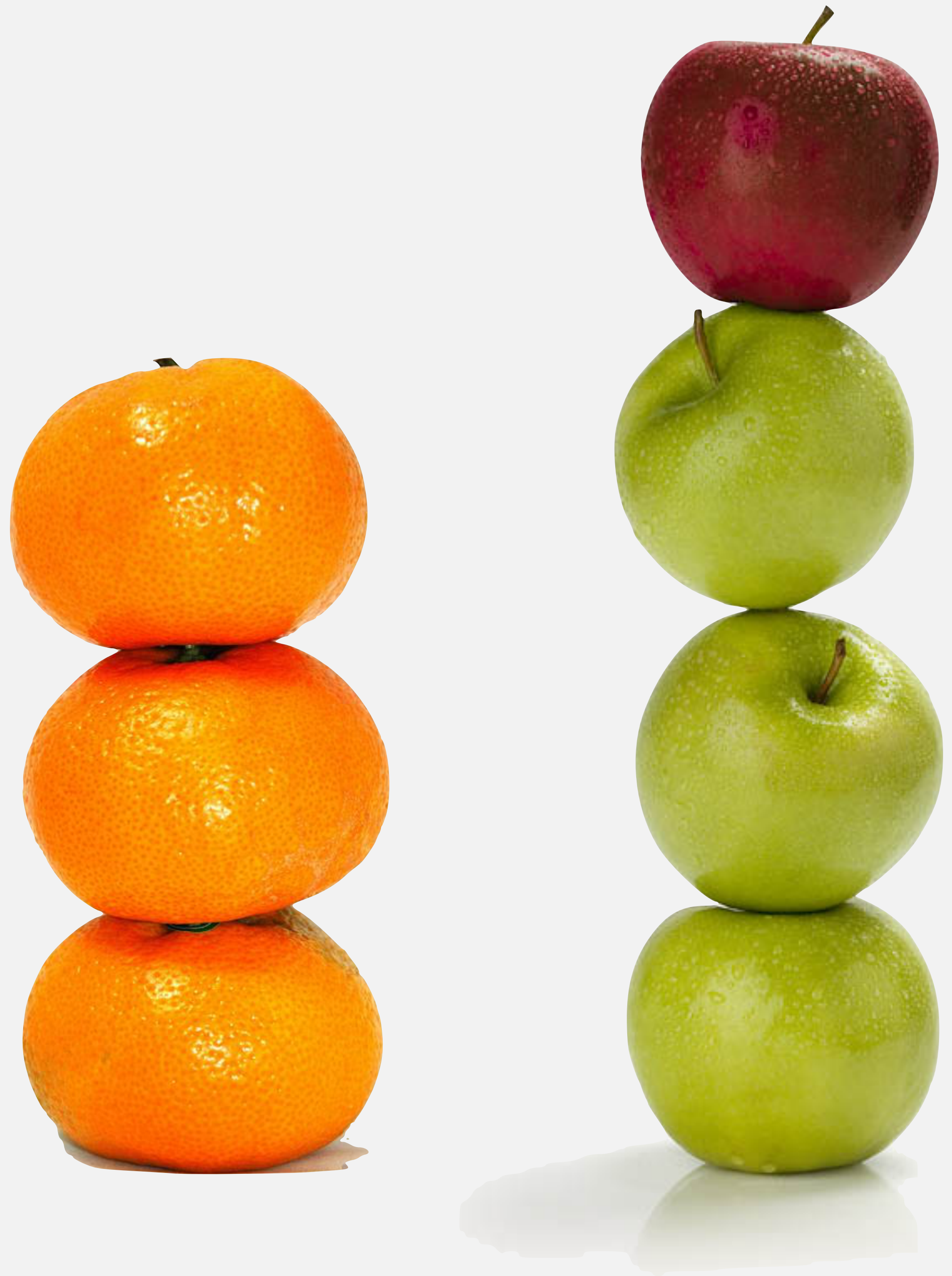
We implemented: StackOfStrings.

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

```
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;
   }
}
```

stack of strings (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 (linked list)

Generic stack: array implementation

The way it should be.

```
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];  }
}
```

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 (fixed-length array) ?

Generic stack: array implementation

The way it should be.

```
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];  }
}
```

stack of strings (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

generic stack (fixed-length array)

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.

Stacks and queues: quiz 5

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

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

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 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();
```

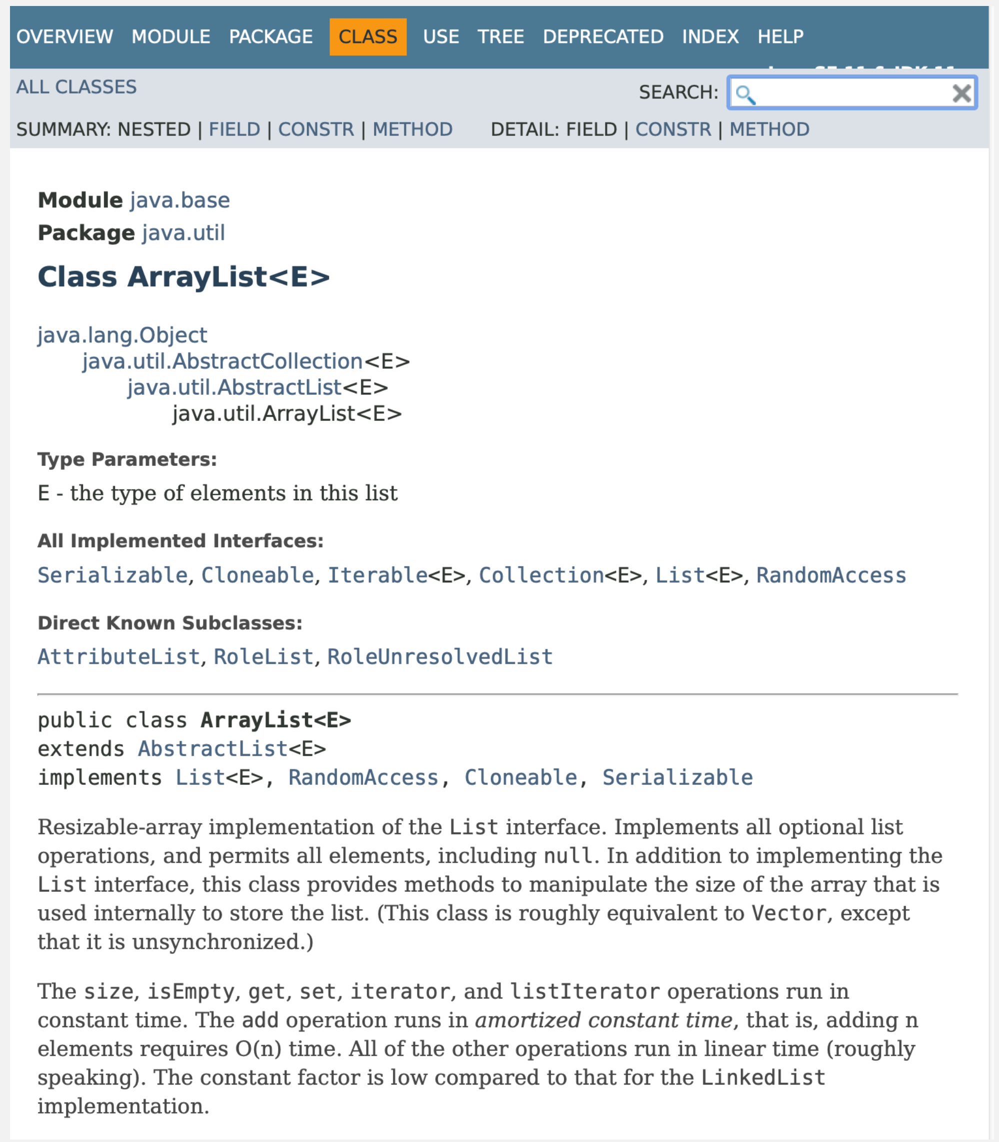
Bottom line. Client code can use generic stack for any type of data.

(but substantial overhead for primitive types)

Java collections framework

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

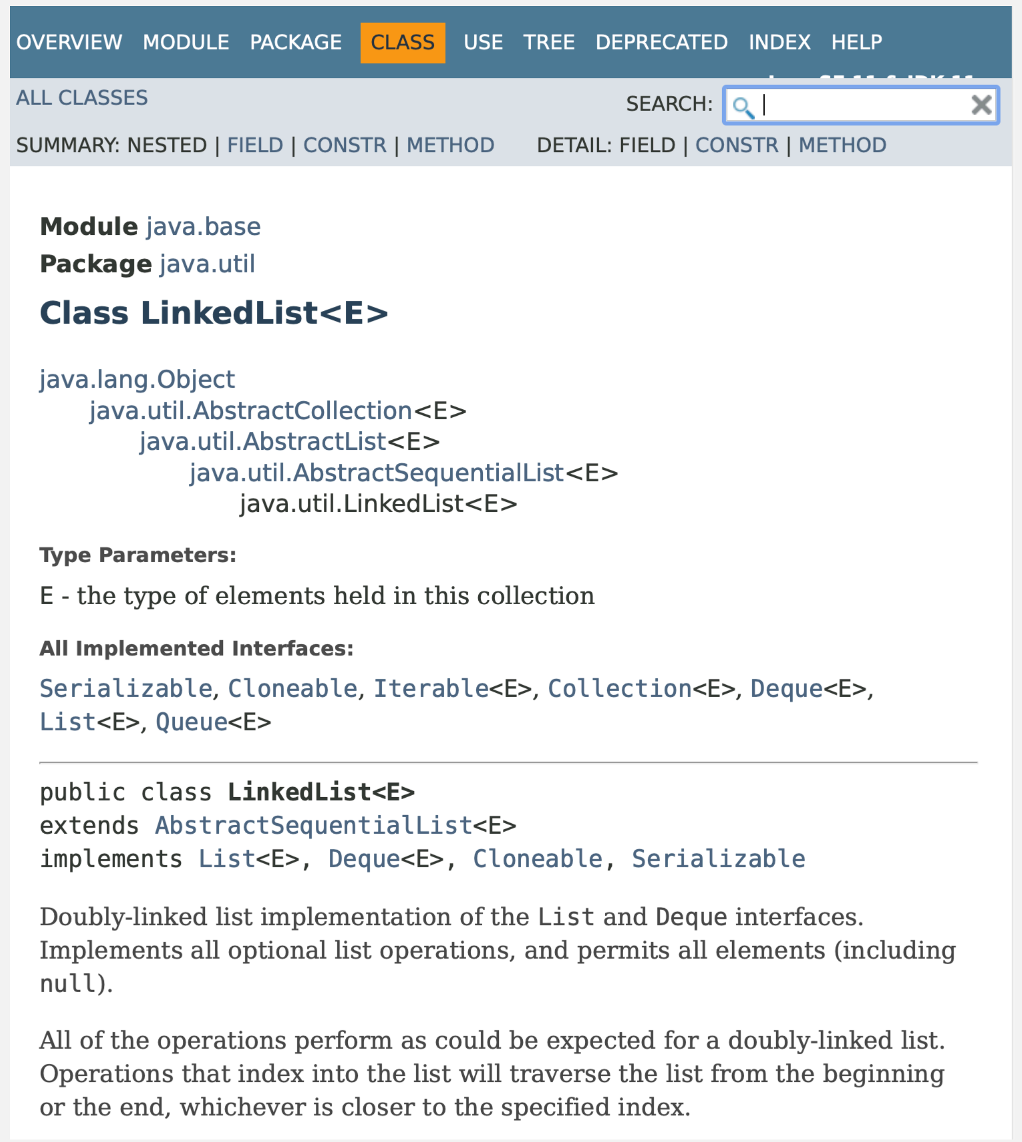
Direct Known Subclasses:

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).