
https://algs4.cs.princeton.edu

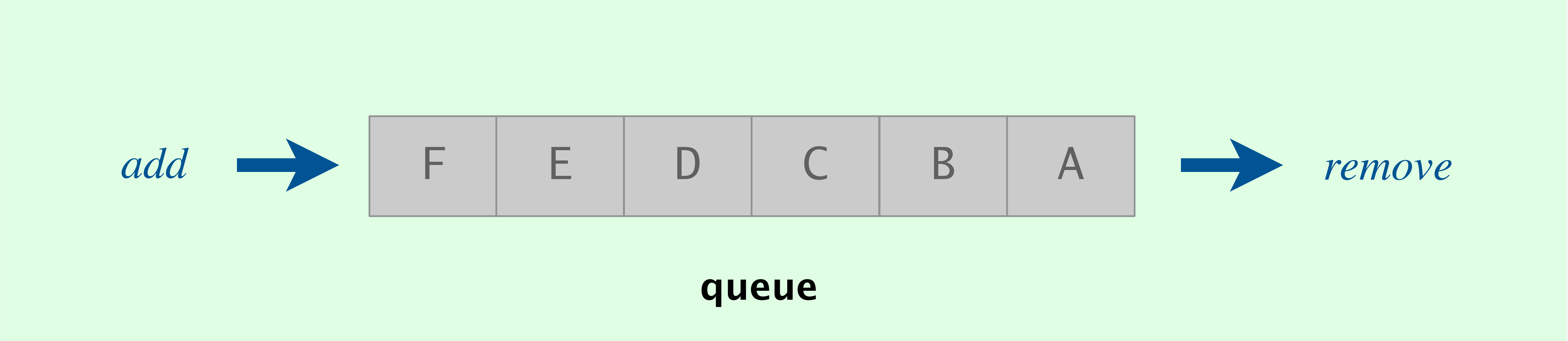
# 1.3 Stacks and Queues

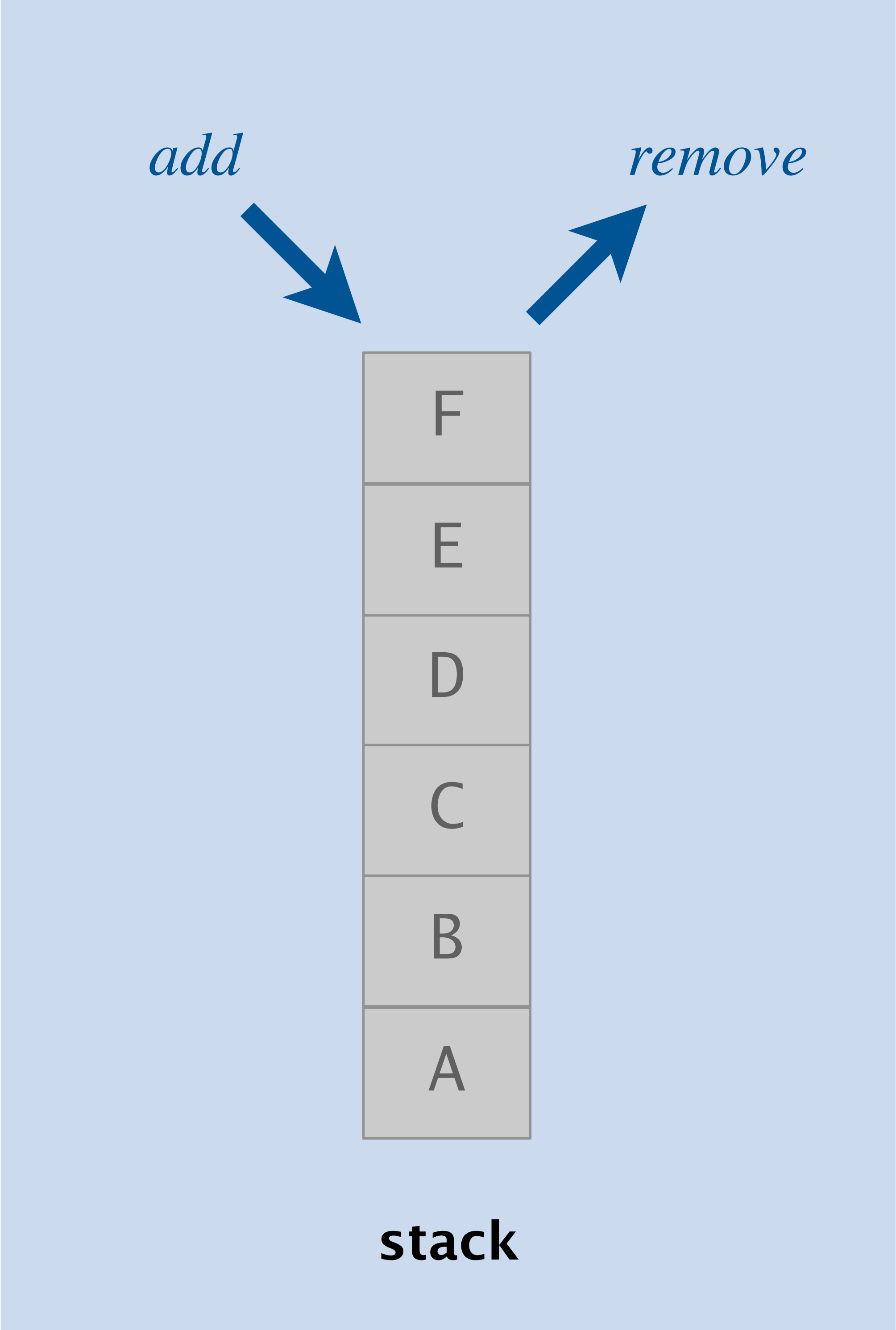
- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *iterators* ← *see next lecture and precept*

# Stacks and queues

Fundamental data types.

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




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

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

# Programming assignment 2

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

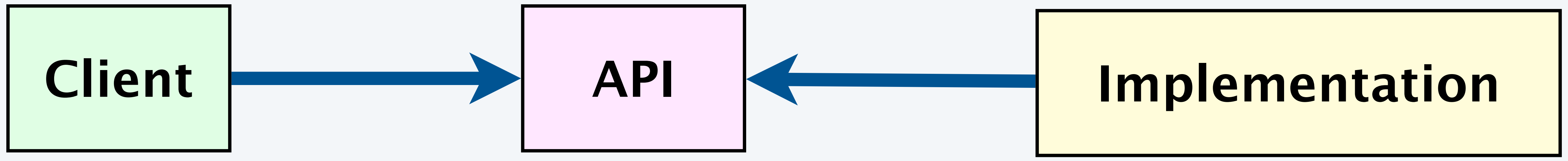
Randomized queue. Remove a random item.


Your job.

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

# Data type design: API, client, and implementation

Separate client and implementation via API.




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

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

Implementation: code that implements the API operations.

Benefits.

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

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


https://algs4.cs.princeton.edu

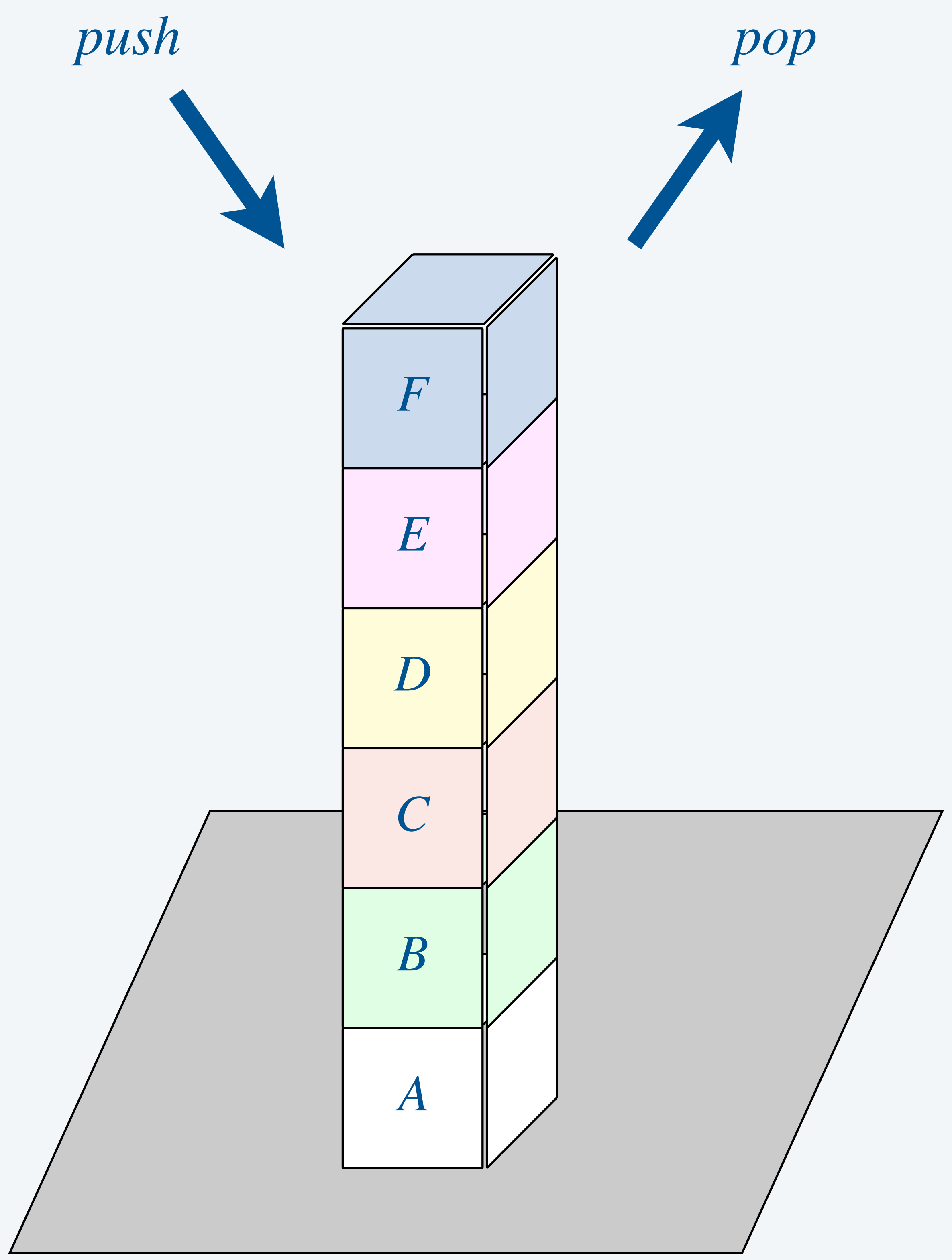
# 1.3 Stacks and Queues

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

# Stack API

**Warmup API.** Stack of strings data type.

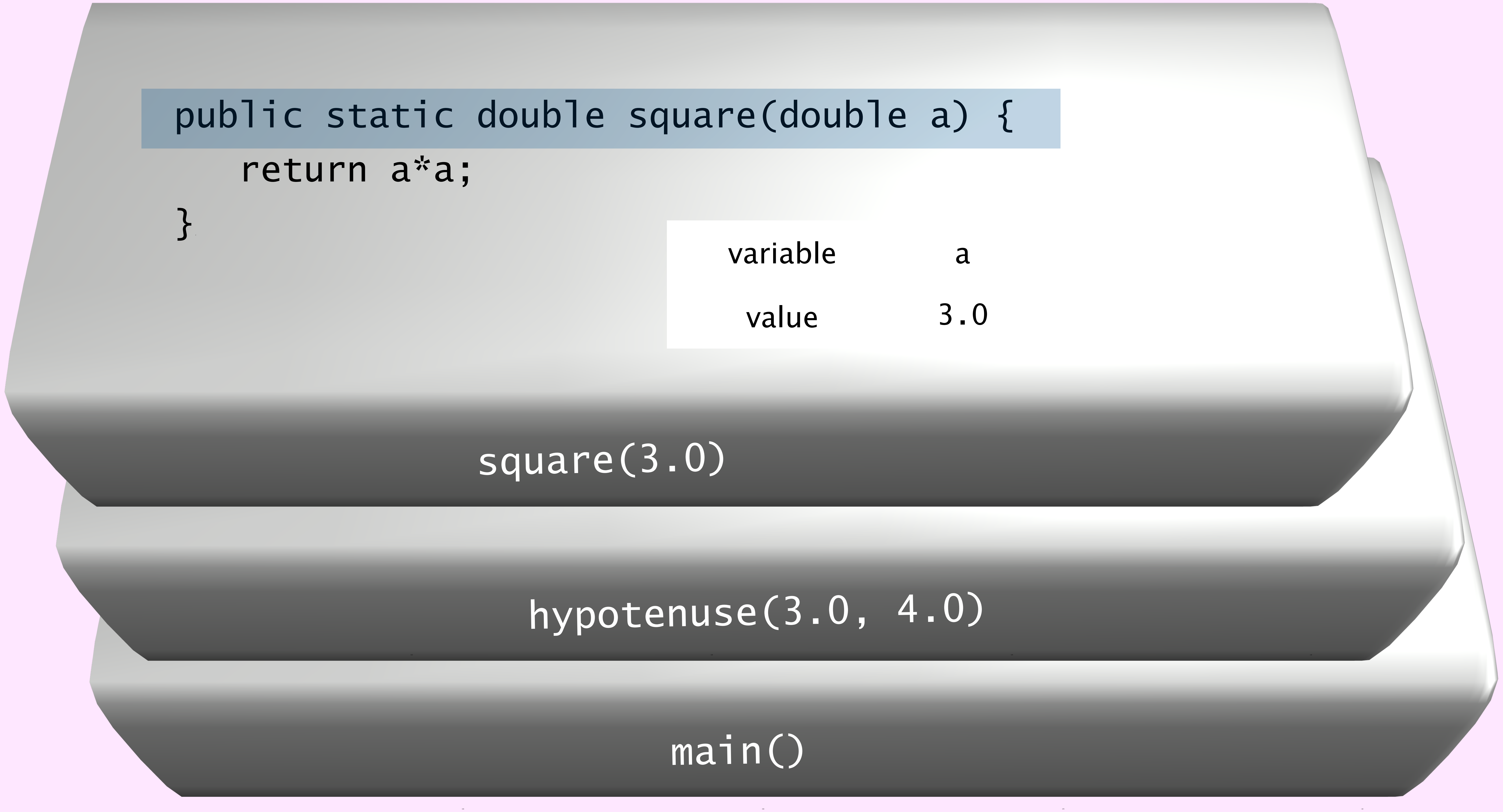
| public class StackOfStrings | | |
|---|---|---|
| | StackOfStrings() | *create an empty stack* |
| void | push(String item) | *add a new string to stack* |
| String | pop() | *remove and return the string most recently added* |
| boolean | isEmpty() | *is the stack empty?* |
| int | size() | *number of strings on the stack* |




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

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

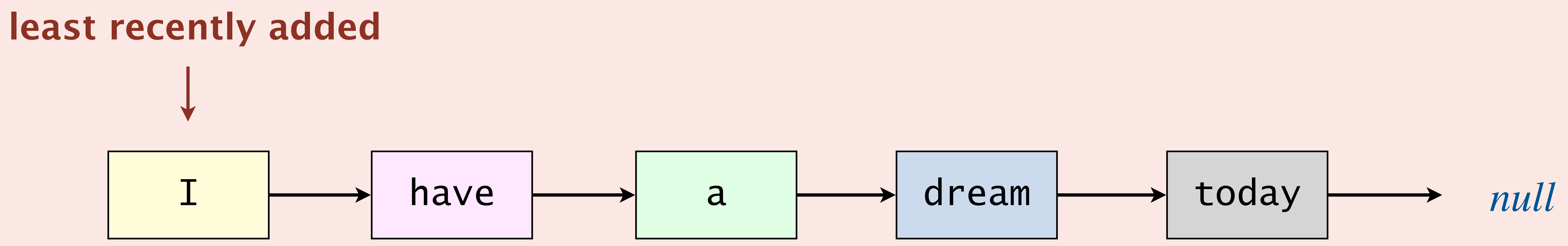
# Function-call stack demo



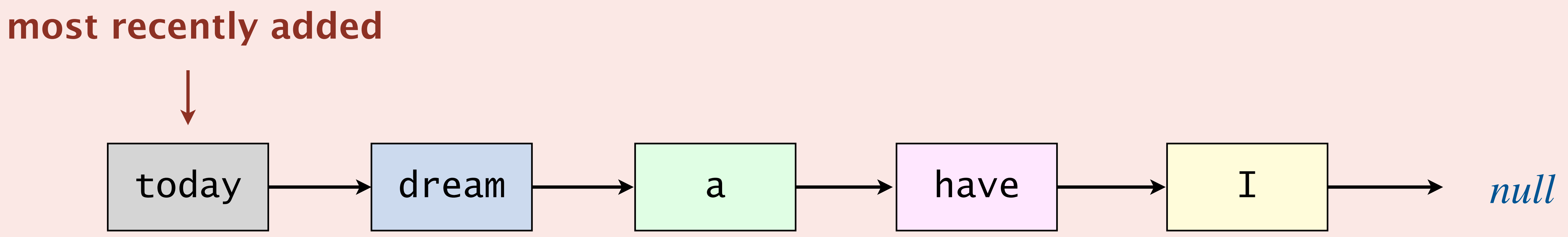

**function-call stack**

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

**A.**




**B.**

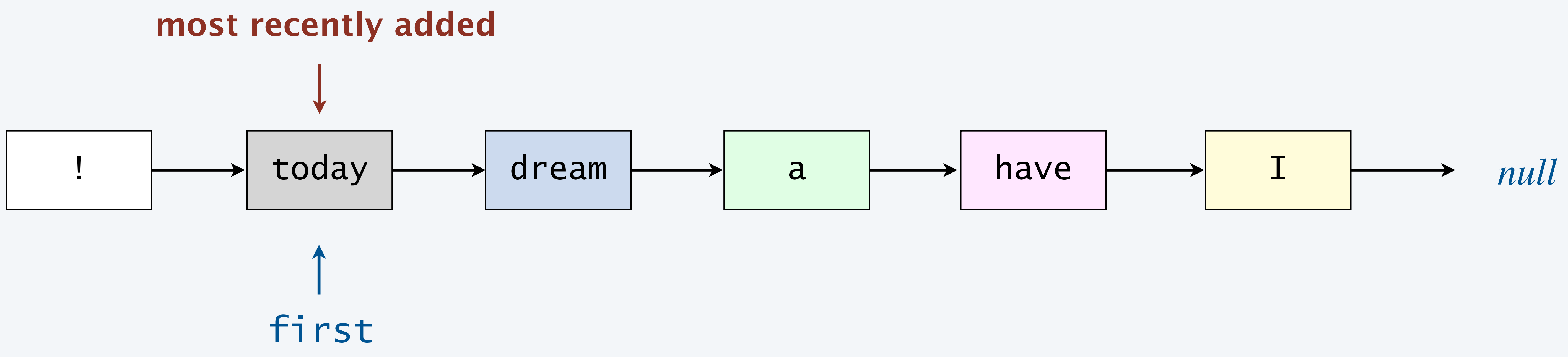



**C.** *Both A and B.*

**D.** *Neither A nor B.*

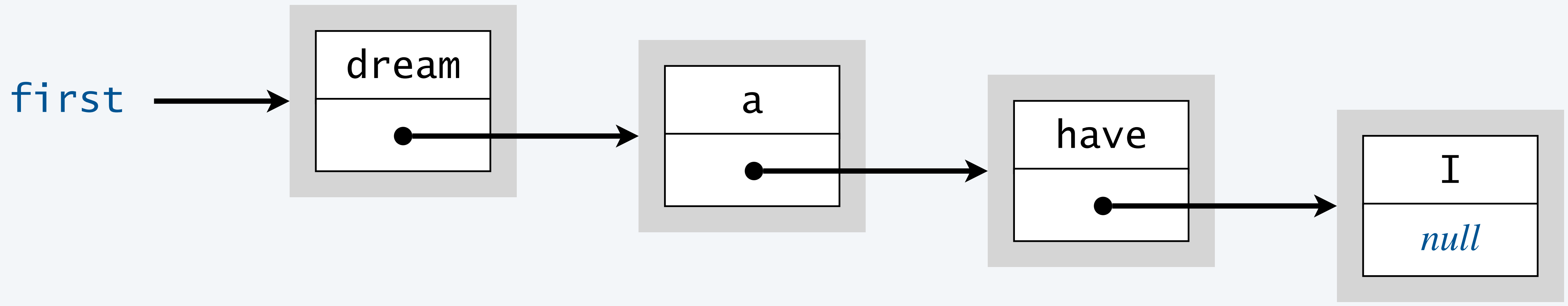
# Stack: linked-list implementation

- 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**




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

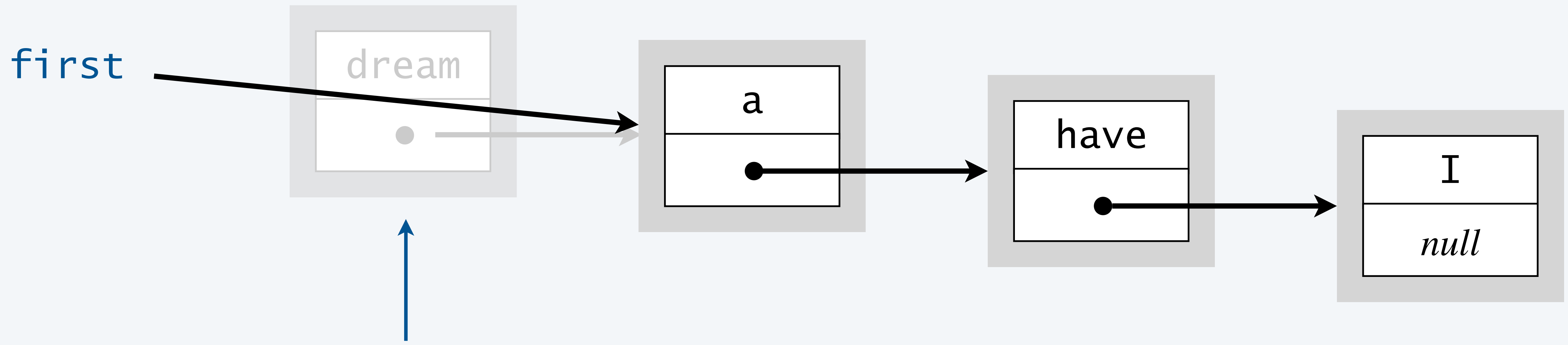
**nested class**

**save item to return**

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

**delete first node**

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




*garbage collector reclaims memory when no remaining references*

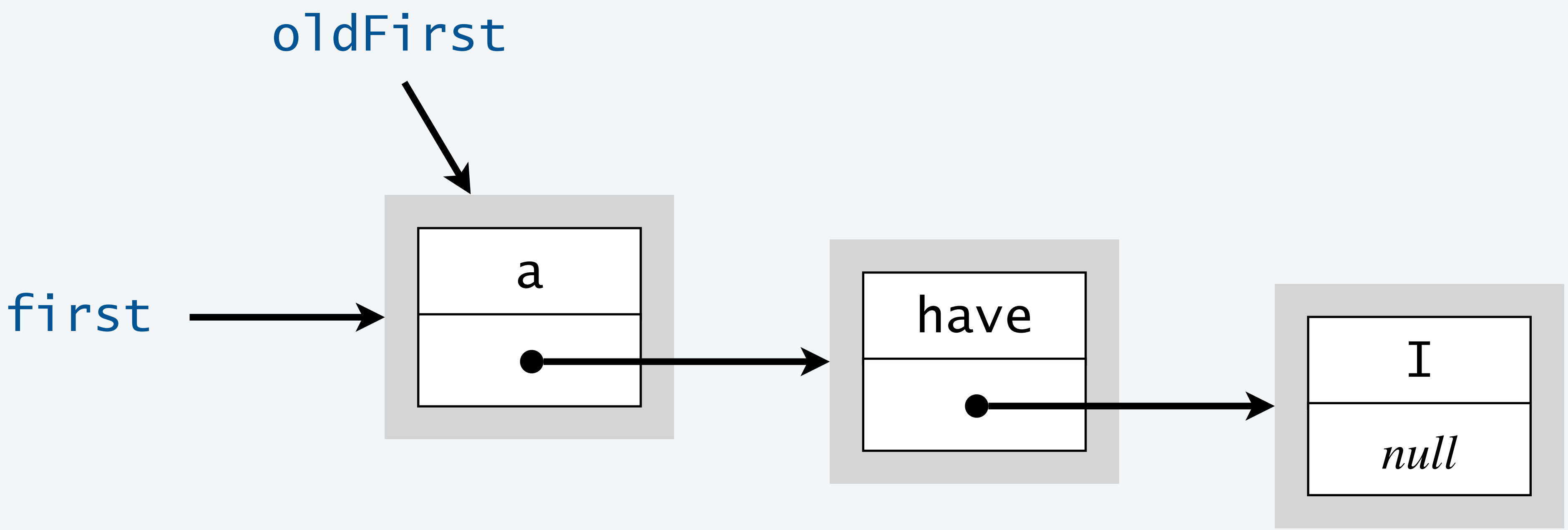
**return saved item**

```
return item;
```

# Stack push: linked-list implementation

**save a link to the list**

```
Node oldFirst = first;
```

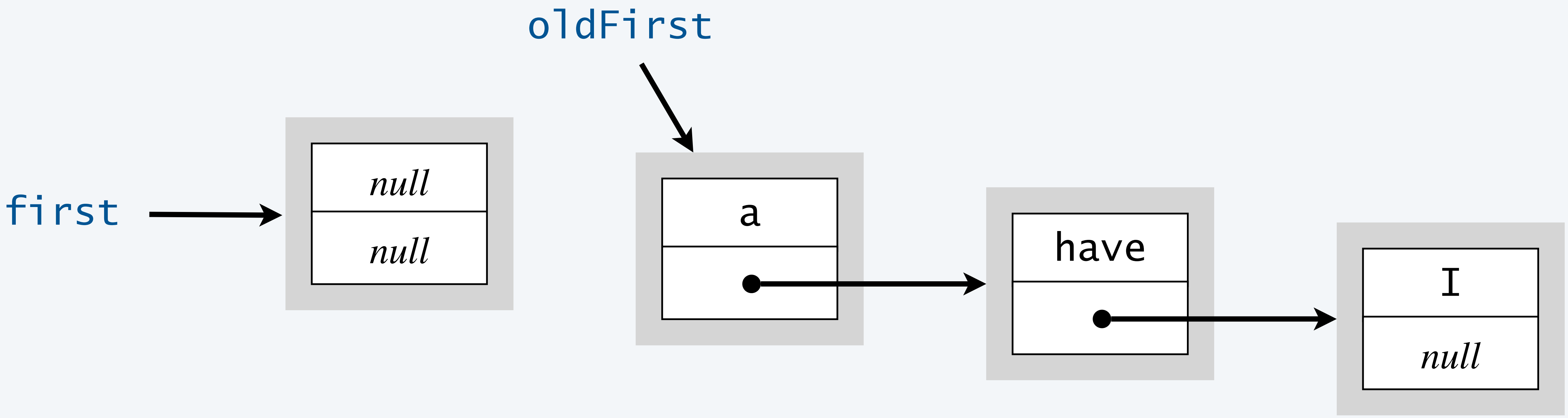



```
private class Node {
   private String item;
   private Node next;
}
```

**nested class**

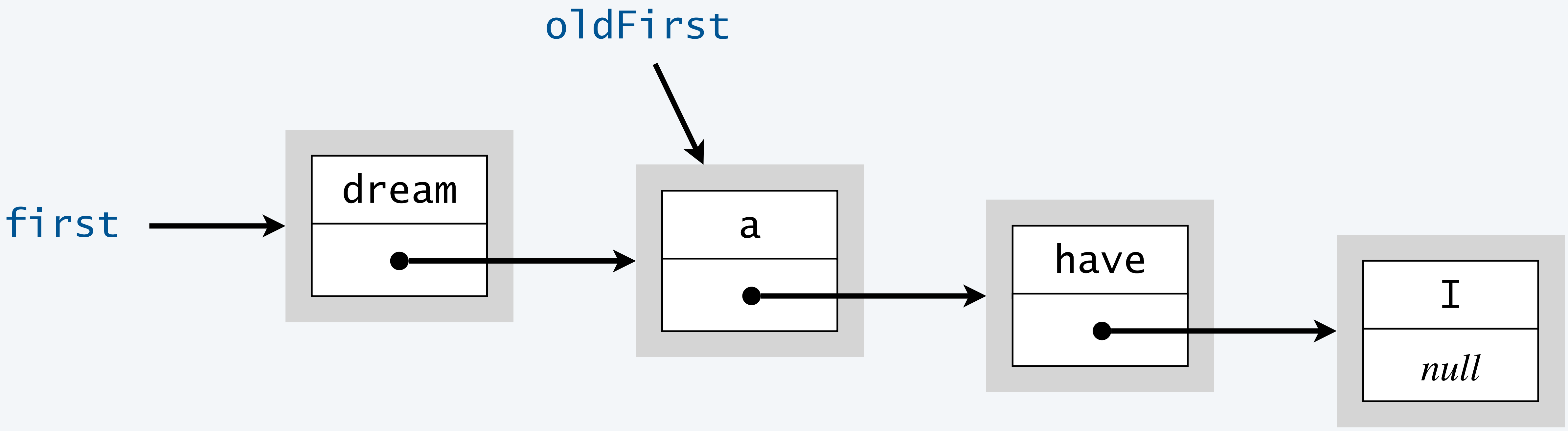
**create a new node at the front**

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




**initialize the instance variables in the new Node**

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



# Stack: linked-list implementation

```
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 nested class*
*(access modifiers for instance variables of such a class don't matter)*

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

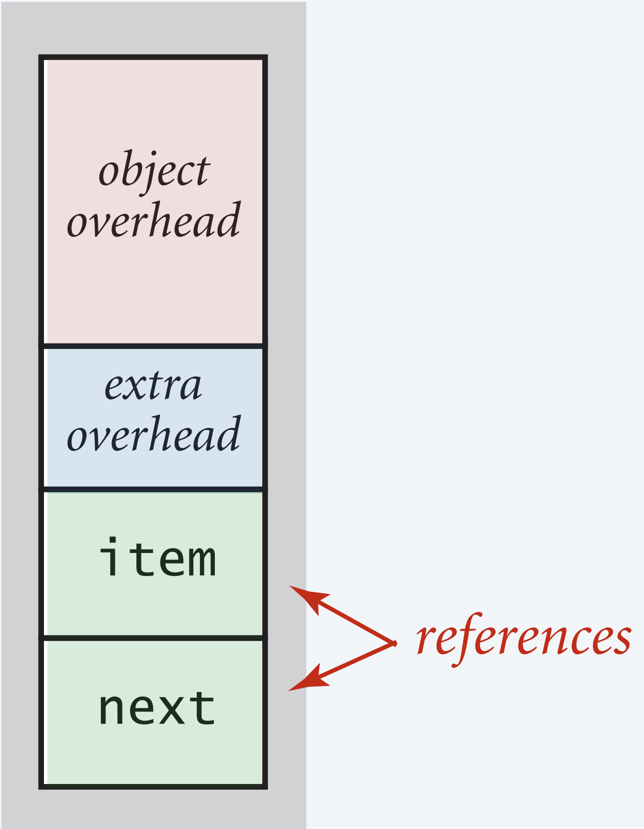
# Stack: linked-list implementation performance

**Proposition.** Every operation takes $\Theta(1)$ time.

**Proposition.** A `LinkedStackOfStrings` with $n$ items has $n$ `Node` objects and uses $\sim 40\,n$ bytes.

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

nested class




16 *bytes (object overhead)*

8 *bytes (non-static nested class extra overhead)*

8 *bytes (reference to* `String`*)*

8 *bytes (reference to* `Node`*)*

---

40 *bytes per stack* `Node`

**Remark.** This counts the memory for the stack itself, including the string references.

[ but not the memory for the string objects, which the client allocates ]

# 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]`.

least recently added

*capacity* = 10

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| `s[]` | `I` | `have` | `a` | `dream` | `today` | `!` | *null* | *null* | *null* | *null* |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | | | | | | | `n` | | | |

**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

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

Overflow. Use "resizing array" for array implementation. [next section]

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

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

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

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

**loitering**

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

**no loitering**

Algorithms

Robert Sedgewick | Kevin Wayne

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?

Naive approach.

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

Too expensive.

- Need to copy all items to a new array, for each push/pop.
- Array accesses to add item $k$ : $1 + 2(k-1)$
- Array accesses to add first $n$ items: $n + (0 + \underbrace{2 + 4 + 6 + \ldots + 2(n-1)}) \sim n^2$.

*to copy $k$−1 elements from old array to new array (ignoring cost to create new array)*

*add items to array*

*array resizing to lengths $2, 3, 4, \ldots, n$*

*infeasible for large $n$*

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 length, and copy items.

*"repeated doubling"*

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

   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;
   }

}
```

*double size of array if full*

*helper method does the resizing*

# Stack: resizing-array implementation

Q. How to grow array?

*"repeated doubling"*

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

Cost is reasonable.

- Still need to copy all items to a new array but, now, only infrequently.
- Total array accesses to add first $n = 2^i$ items: $\underset{\text{add items to array}}{n} + \underbrace{(2 + 4 + 8 + 16 + \ldots + n)}_{\text{array resizing to lengths } 2,\, 4,\, 8,\, 16,\, \ldots,\, n} \sim \underset{\text{feasible for large } n}{3\,n}$.

# Stack: resizing-array implementation

Q. How to shrink array?

First try.

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

Too expensive for some sequences of operations.

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

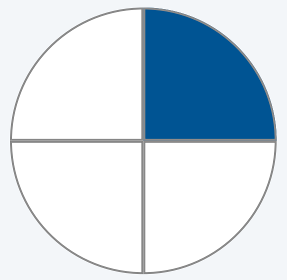
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **full** | I | have | a | dream | | | | |
| **push("today")** | I | have | a | dream | today | *null* | *null* | *null* |
| **pop()** | I | have | a | dream | | | | |
| **push("today")** | I | have | a | dream | today | *null* | *null* | *null*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

# Stack: resizing-array implementation

Q. How to shrink array?

Efficient solution.

- Push: double length of array `s[]` when array is full.
- Pop: halve length 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;
}
```

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

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

# Amortized analysis

Worst-case analysis. Worst-case running time for an individual operation.

Amortized analysis. Worst-case running time for a sequence of operations.

- Amortized cost per operation = total cost / # operations.
- Provides more realistic analysis when some operations are expensive but rare.

**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: memory usage

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

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

```
public class ResizingArrayStackOfStrings {
    private String[] s;     ← 8 bytes × array length
    private int n = 0;

    ⋮
}
```

Remark. This counts the memory for the stack itself, including the string references.

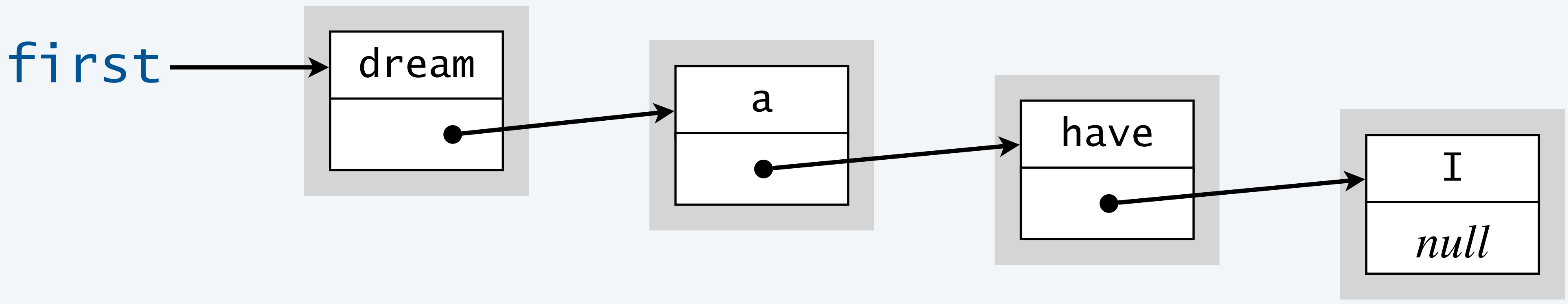
[ but not the memory for the string objects, which the client allocates ]

# 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.

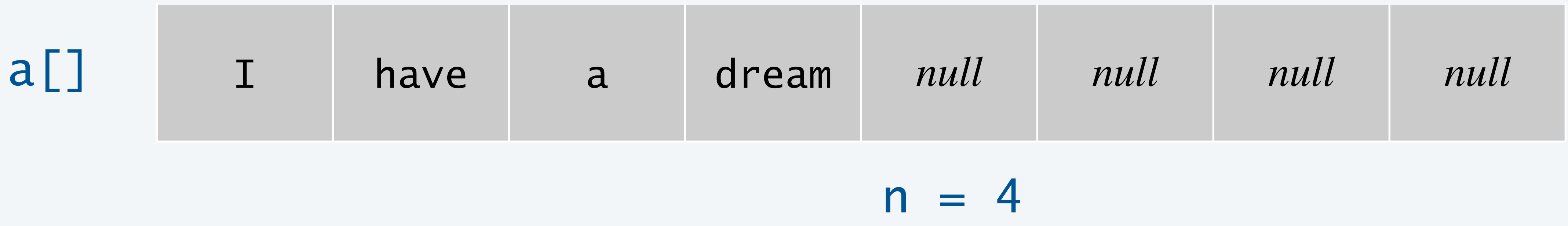
- Stronger performance guarantee.
- More memory.




Resizing-array implementation.

- Weaker performance guarantee.
- Less memory.
- Better use of cache.

*accessing adjacent memory locations (e.g., in an array) is much faster than accessing nonadjacent memory locations (e.g., in a linked list)*




https://algs4.cs.princeton.edu

# 1.3 Stacks and Queues

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

# Queue of strings API

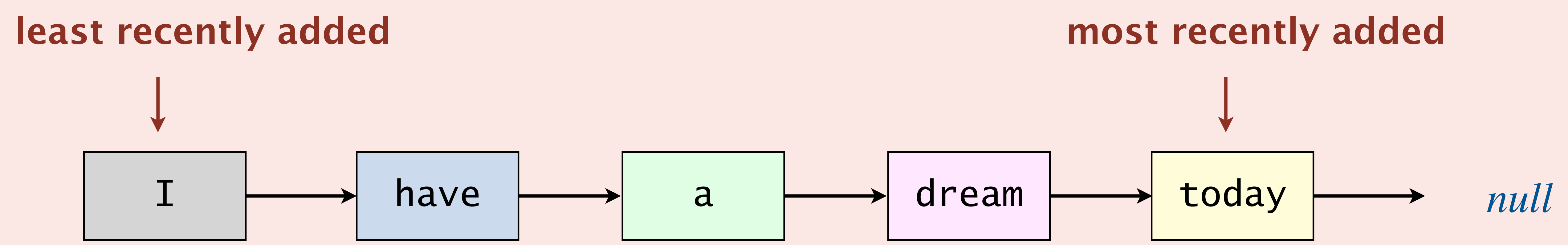

| | | |
|---|---|---|
| `public class QueueOfStrings` | | |
| | `QueueOfStrings()` | *create an empty queue* |
| `void` | `enqueue(String item)` | *add a new string to queue* |
| `String` | `dequeue()` | *remove and return the string least recently added* |
| `boolean` | `isEmpty()` | *is the queue empty?*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

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

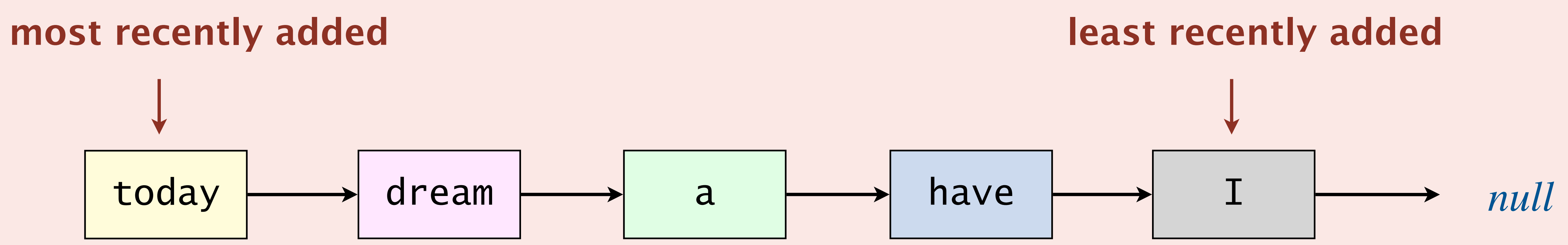
# Stacks and queues: quiz 3

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

**A.**




**B.**

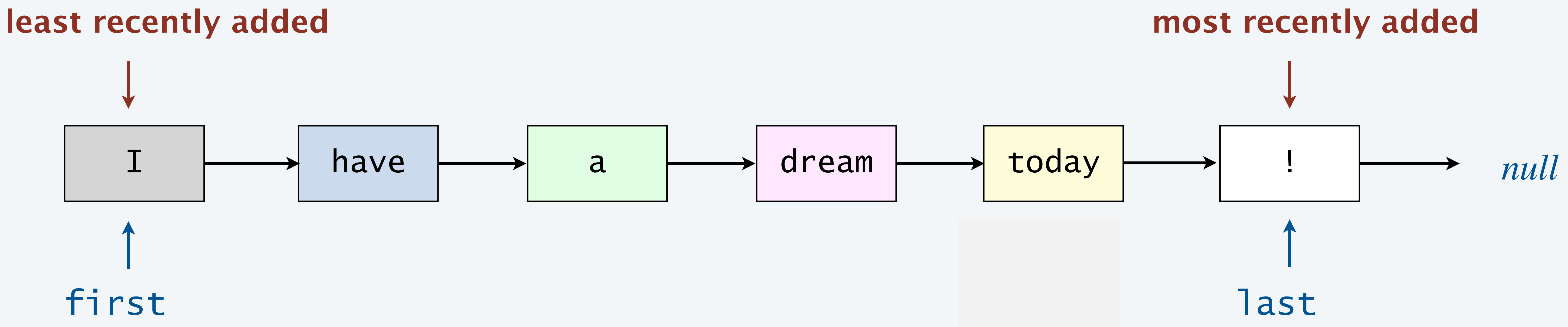



**C.** *Both A and B.*

**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`.



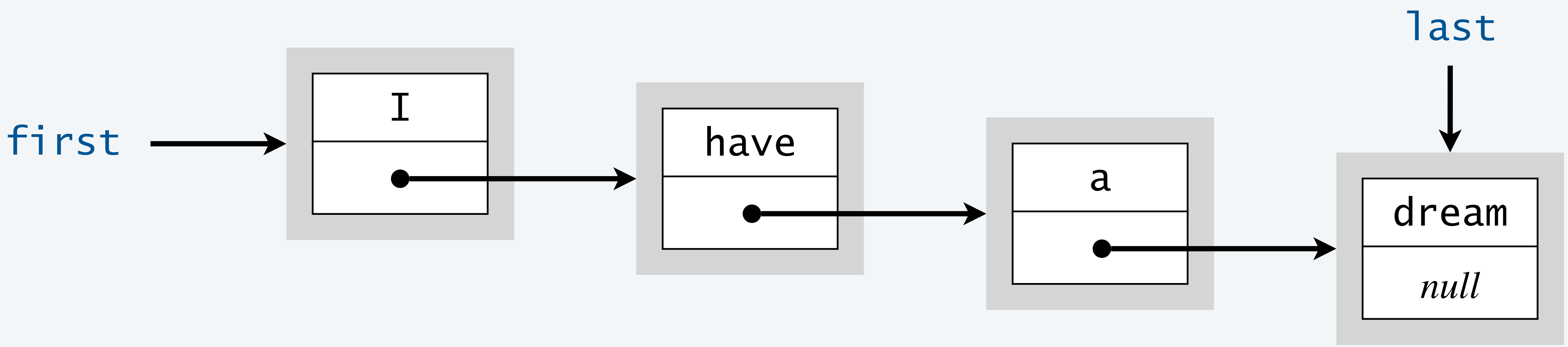
# Queue dequeue: linked-list implementation

Remark. Code is identical to `pop()`.

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

**nested class**

**singly linked list**

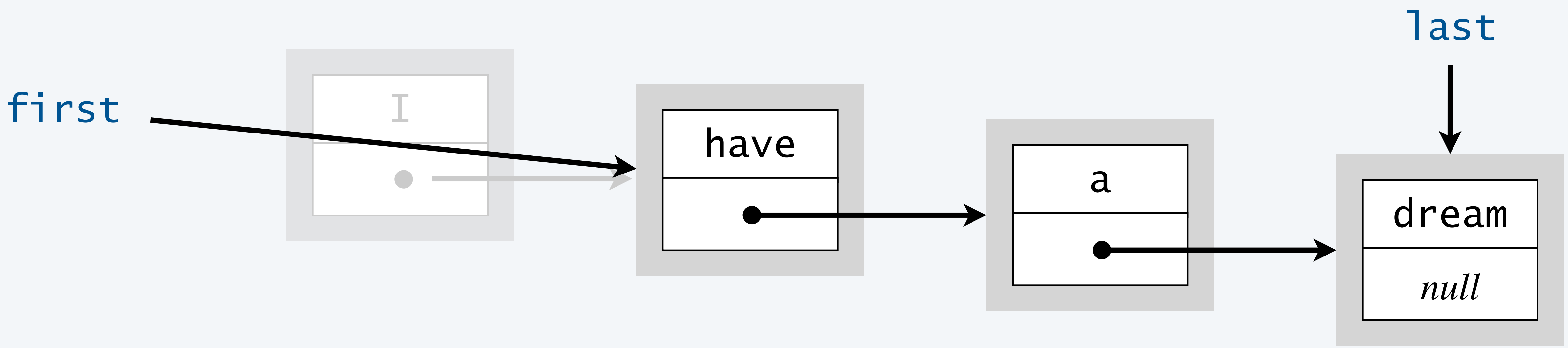



**save item to return**

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

**delete first node**

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




**return saved item**

```
return item;
```

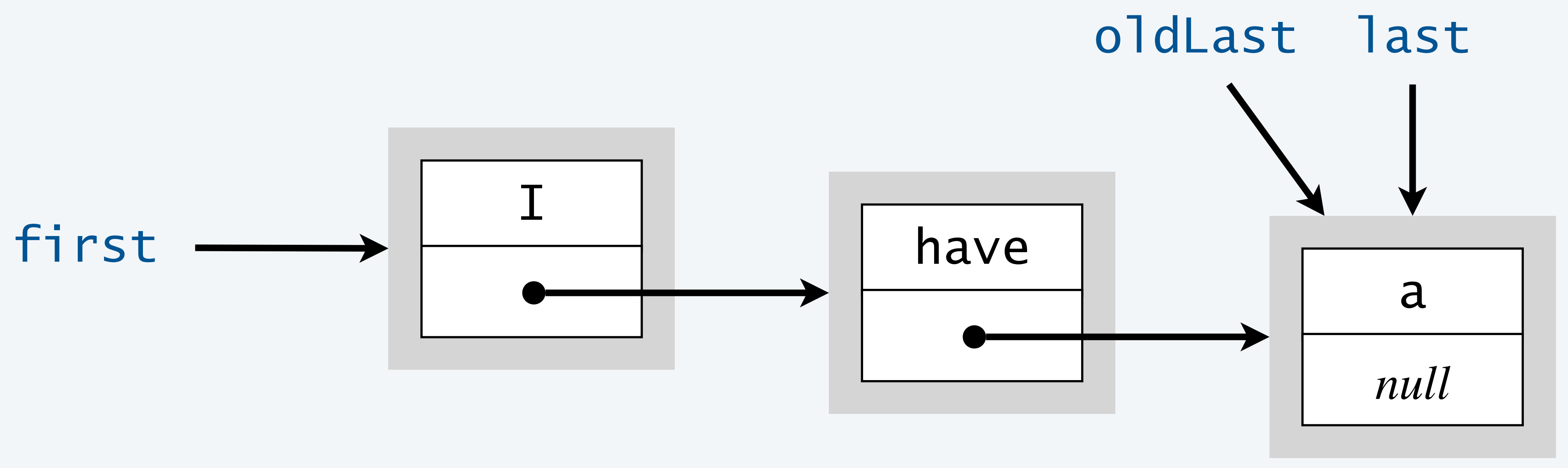
# Queue enqueue: linked-list implementation

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

**nested class**

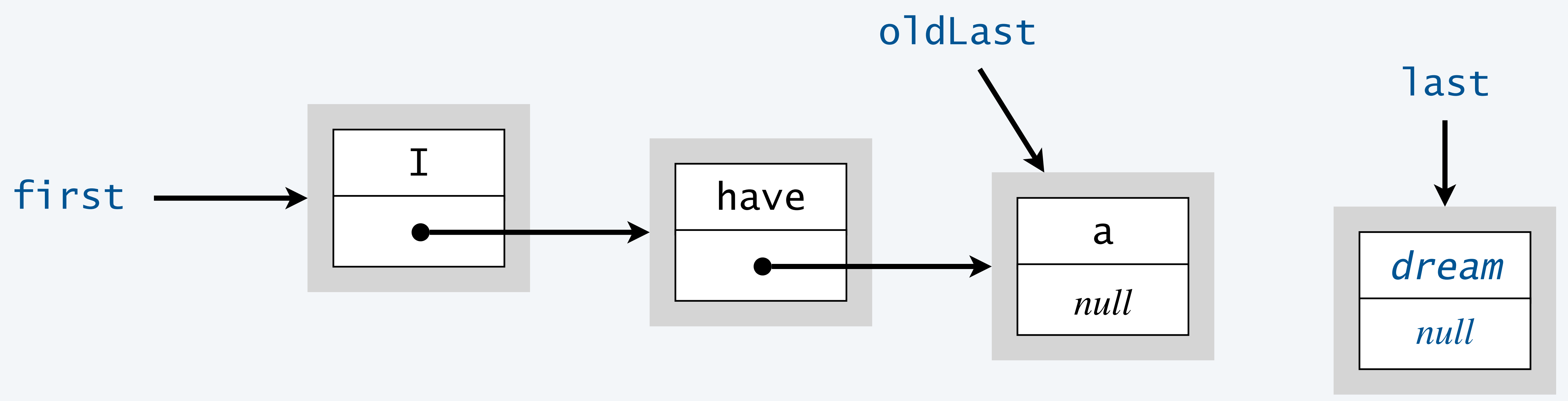
**save a link to the list**

```
Node oldLast = last;
```



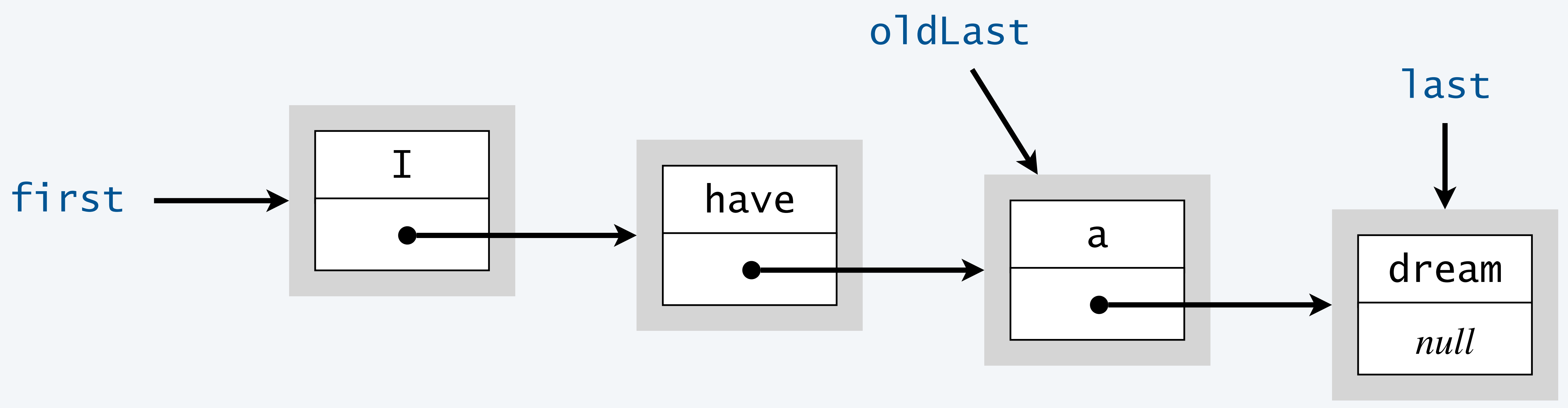

**create a new node at the end**

```
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 case: add to an empty queue*

*corner case: remove down to an empty queue*

# Queue: resizing-array implementation

Goal. Implement a queue using a resizing array so that, starting from an empty queue, any sequence of $m$ operations takes $\Theta(m)$ time.

least recently added

q[]

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

n

`dequeue()` *is inefficient*

most recently added

q[]

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

n

`enqueue()` *is inefficient*

# Queue: resizing-array implementation

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

least recently added ↓ (index 2)

most recently added ↓ (index 5)

| q[] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| | *null* | *null* | `I` | `have` | `a` | `dream` | *null* | *null* | *null* | *null* |


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`, …

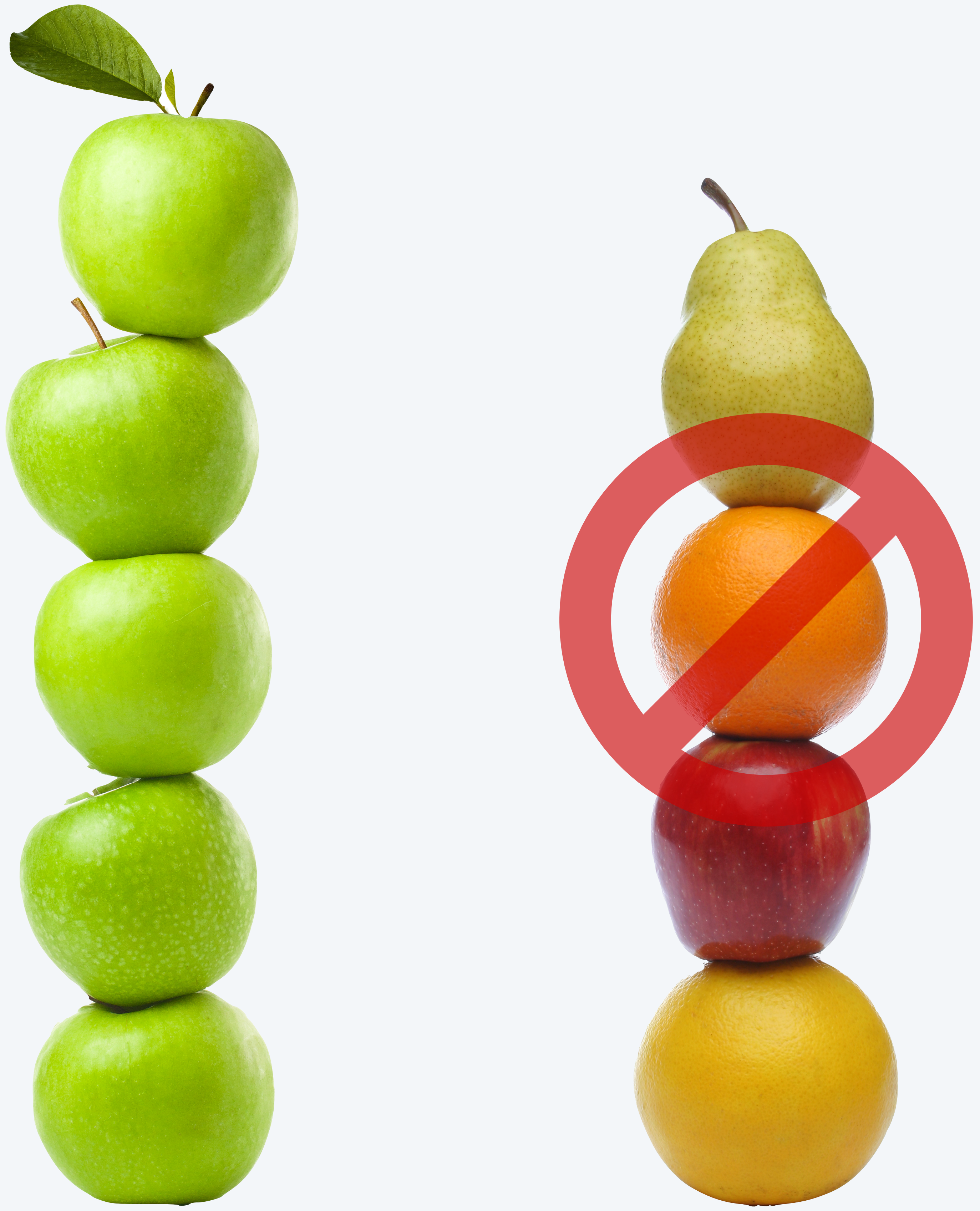
Solution in Java: generics.

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

*type parameter*
*(use to specify type and call constructor)*

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

← *compile-time error*



# Generic stack: 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 Fixed...OfStrings(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 Fixed...OfStrings(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

```
~/cos226/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.

**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 an associated "wrapper" reference type.
- Ex: `Integer` is wrapper type associated with `int`.

Autoboxing. Automatic cast from primitive type to wrapper type.

Unboxing. Automatic cast from wrapper type to primitive type.

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

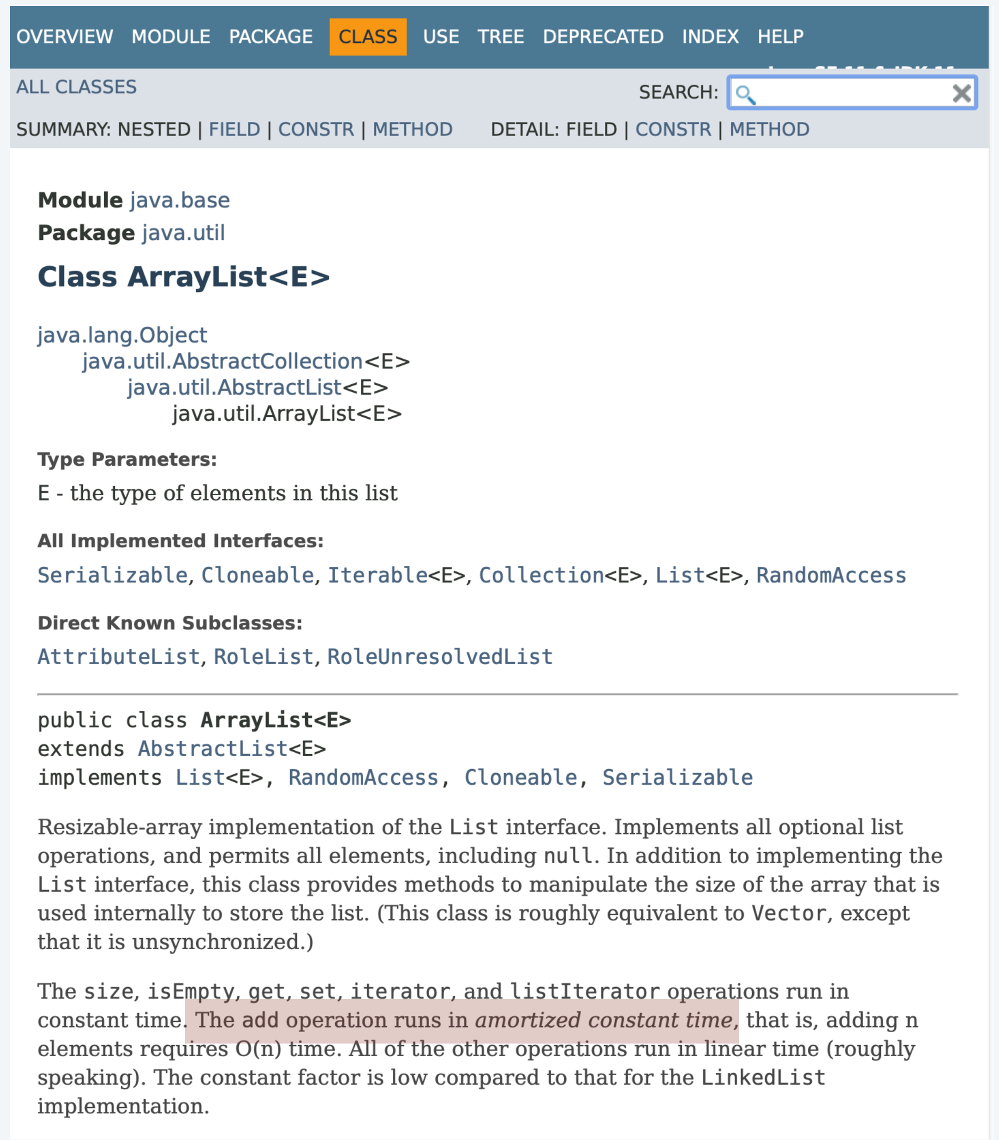
Bottom line. Client code can use generic stack for any type of data.

Caveat. Performance overhead for primitive types.

# Java collections framework

**Java's library of collection data types.**

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


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

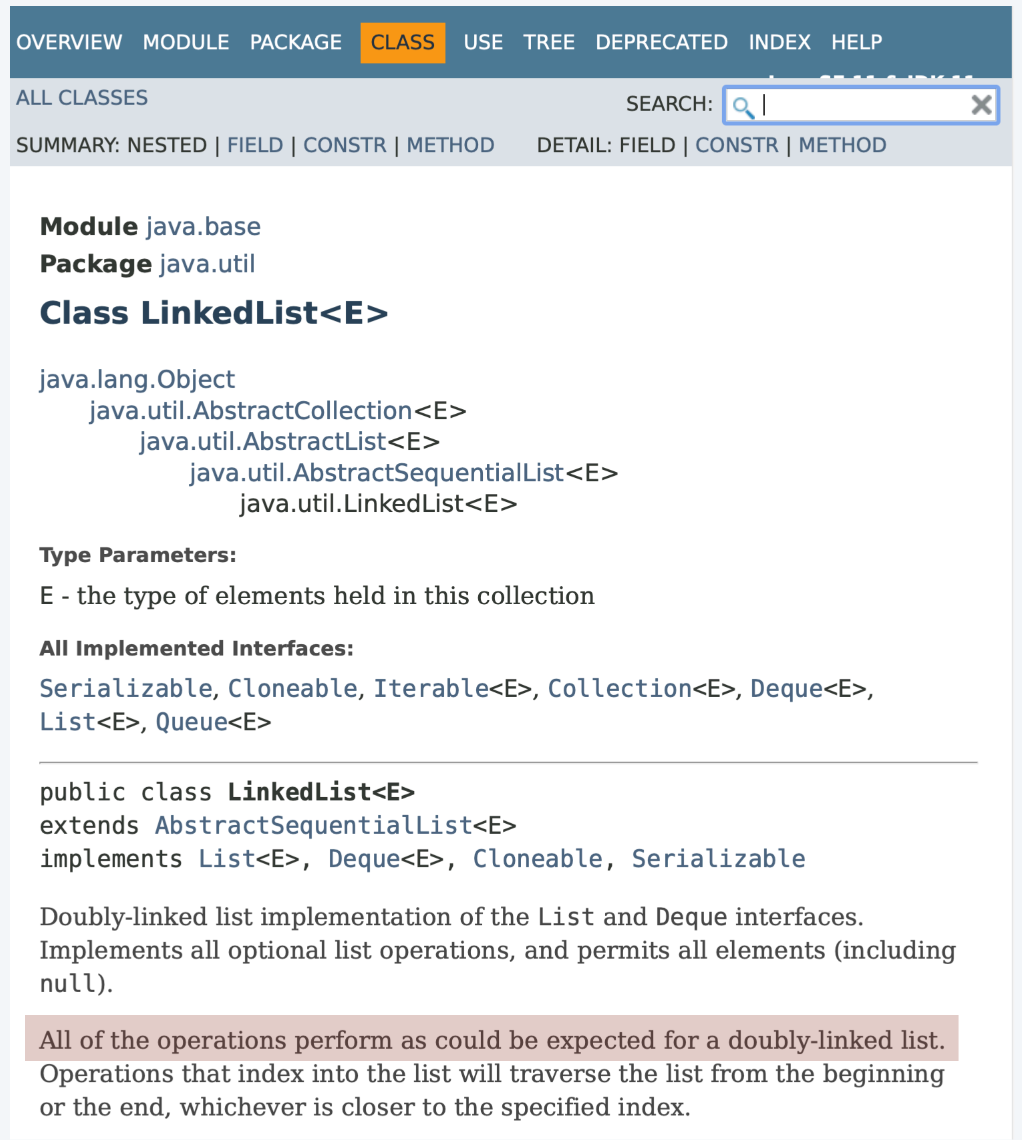
**All Implemented Interfaces:**
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

**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 `Stack` and `Queue` in `algs4.jar` for stacks and queues to improve design and efficiency.
- Use `java.util.ArrayList` or `java.util.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, size, test if empty.

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

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

Efficient implementations.

- Singly linked list.
- Resizing array.

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

# Credits

| image | source | license |
|---|---|---|
| *Assignment Logo* | Kathleen Ma ’18 | by author |
| *Stack of Books* | Adobe Stock | Education License |
| *No Loitering* | signsworldwide.com | |
| *Bob Tarjan* | Heidelberg Laureate | |
| *Long Queue Line* | Adobe Stock | Education License |
| *Stack of Apples* | Adobe Stock | Education License |
| *Stack of Fruit* | Adobe Stock | Education License |
| *Queue of People* | Adobe Stock | Education License |
| *Stack of Sweaters* | Adobe Stock | Education License |
| *ChatGPT Phone* | Adobe Stock | Editorial Use |

# A final thought

" *Linked lists, nodes connected with care,*

*Arrays resizing, with memory to spare.*

*Organizing data, their only need,*

*Helping us, with efficiency indeed.* "