



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

1.3 STACKS AND QUEUES II

- *linked lists*
- *stacks (linked-list)*
- *queues (linked-list)*
- *iterators and for-each loops*
- *Java Collections Framework*

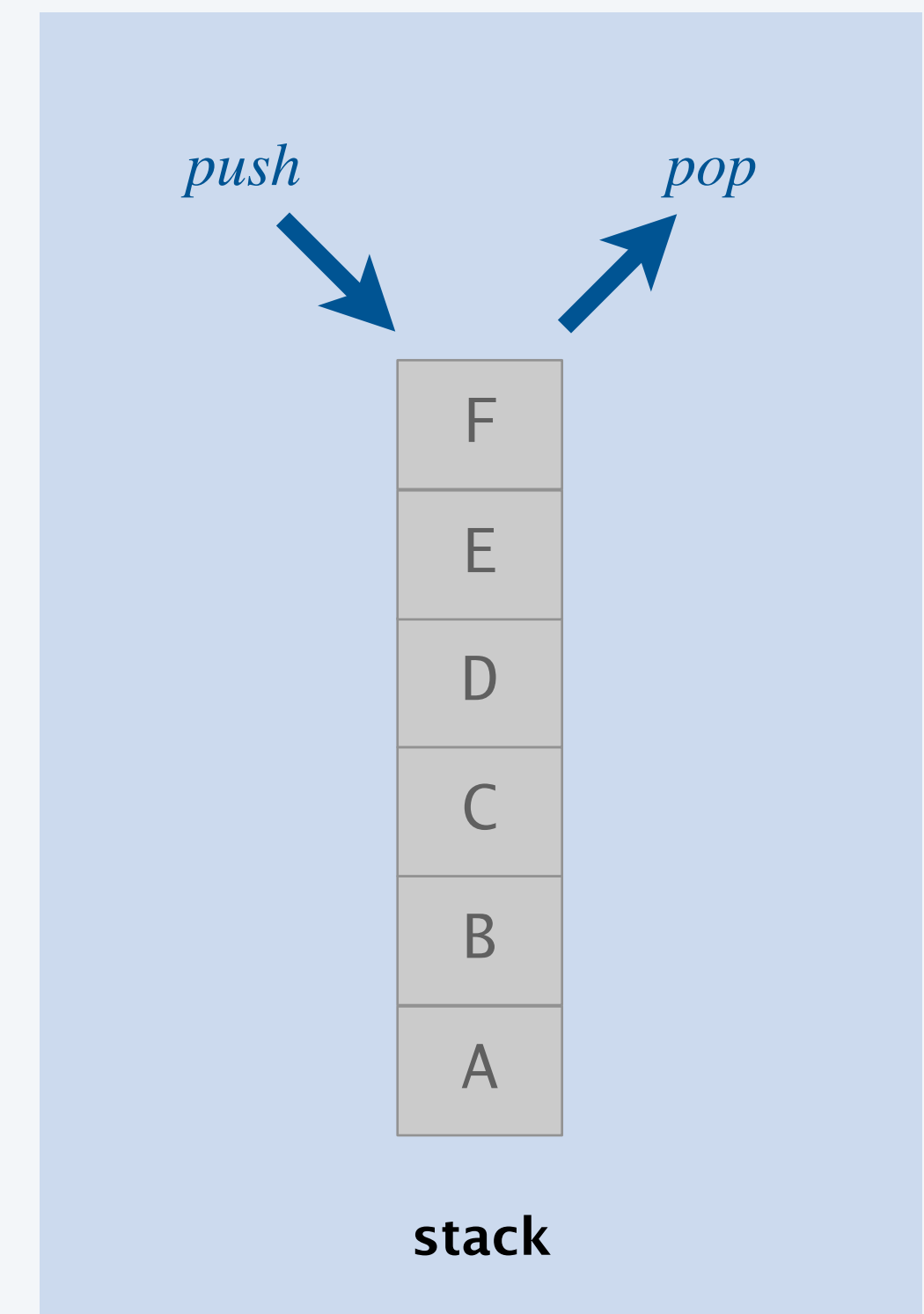
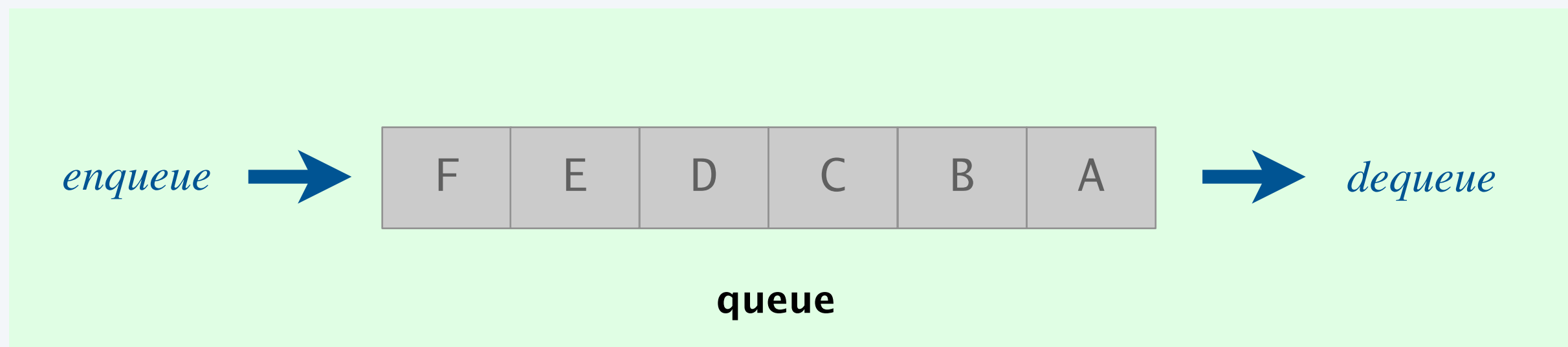
Stacks and queues: review

Fundamental data types.

- Value: a **collection** of items of the same type.
- Core operations: **add**, **remove**, iterate, test if empty.

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

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



Programming assignment 2

Deque. Add/remove items at either **front** or **back**.

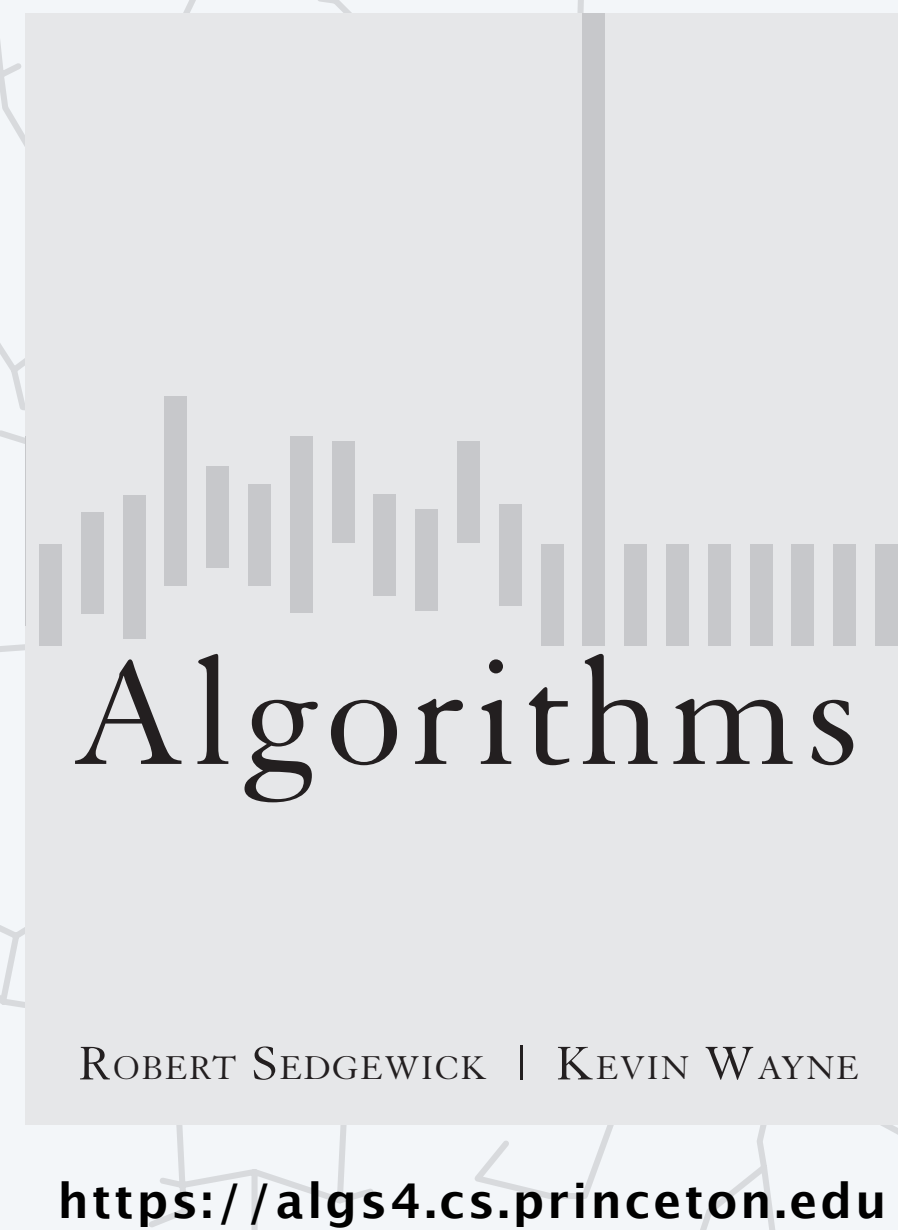
Randomized queue. Remove an item **chosen uniformly at random**.



Your job.

- Step 1. Choose a data structure that meets the performance requirements.
- Step 2. Implement it from scratch.

← *design first; code second*



1.3 STACKS AND QUEUES II

- ▶ *linked lists*
- ▶ *stacks (linked-list)*
- ▶ *queues (linked-list)*
- ▶ *iterators and for-each loops*
- ▶ *Java Collections Framework*

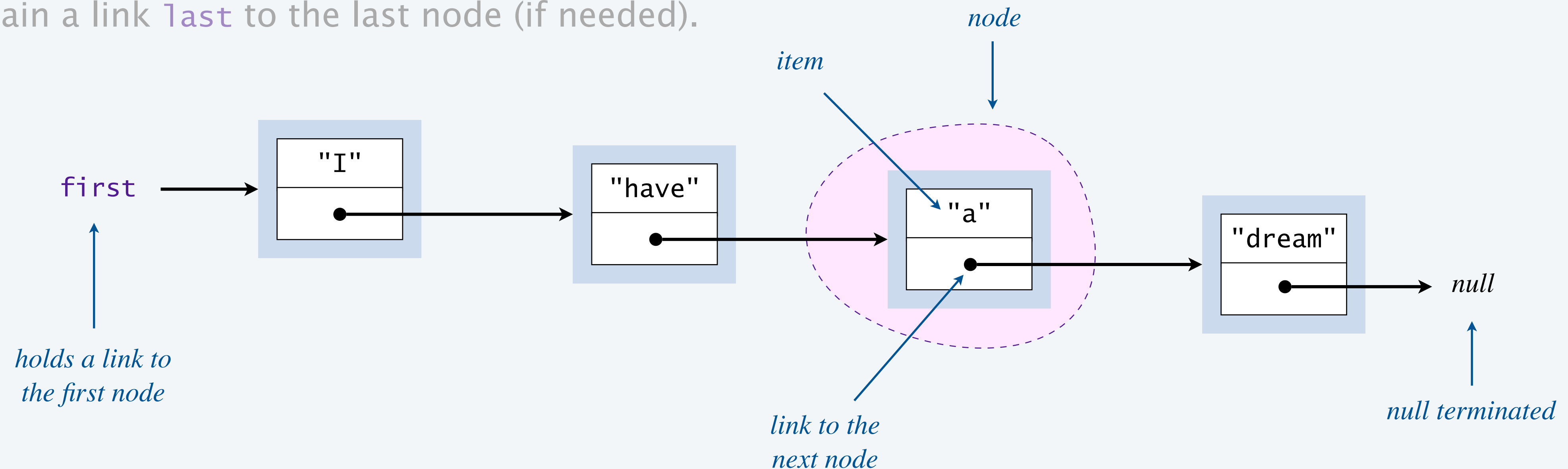
Linked lists

Last lecture. Use a **resizable array** to implement push/pop in **amortized constant time**.

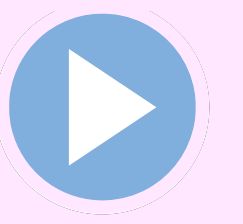
This lecture. Use a **singly linked list** to implement push/pop in **worst-case constant time**.

Singly linked list.

- Each **node** stores an item and a **link (pointer)** to the next node.
- The last node's next link is **null**.
- Maintain a link **first** to the first node.
- Maintain a link **last** to the last node (if needed).



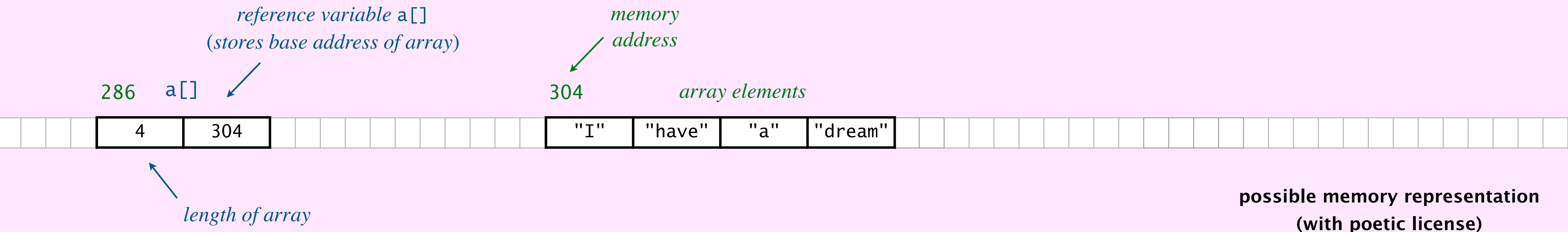
Possible memory representation of an array



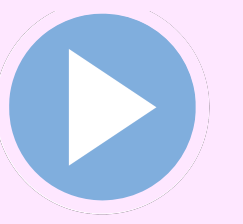
Array. Stores its elements **contiguously** in memory.

Consequences.

- Accessing the k^{th} element takes $\Theta(1)$ time (base address + offset).
- The array length is fixed once the array is created.



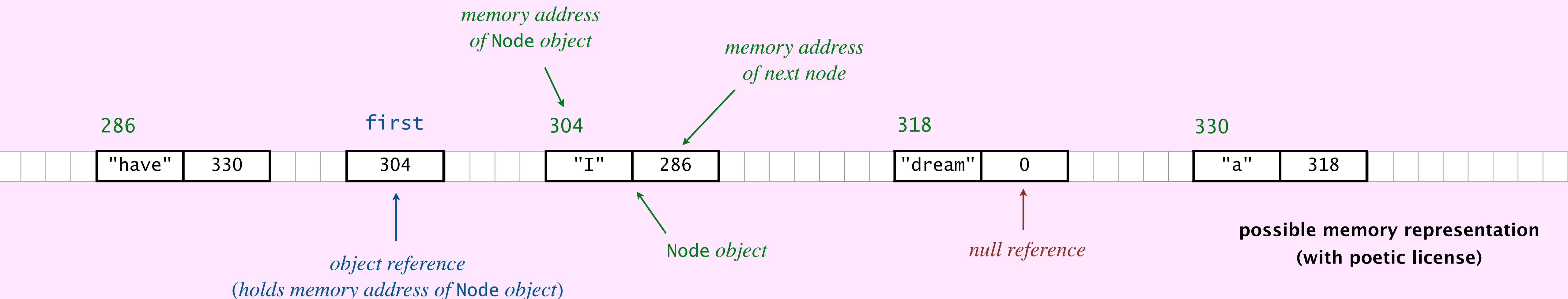
Possible memory representation of a singly linked list



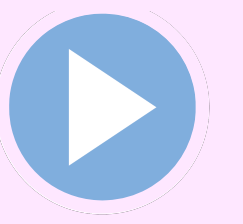
Linked list. May store its nodes **non-contiguously** in memory.

Consequences.

- Accessing the k^{th} node takes $\Theta(k)$ time (must follow k links).
- The list length is not fixed; it can grow or shrink one node at a time.



Building a singly linked list in Java

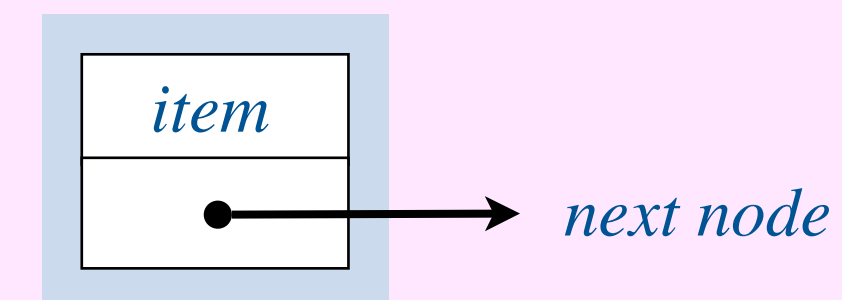


Node data type. Each **Node** object contains:

- an item
- a **reference** to the next **Node**

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

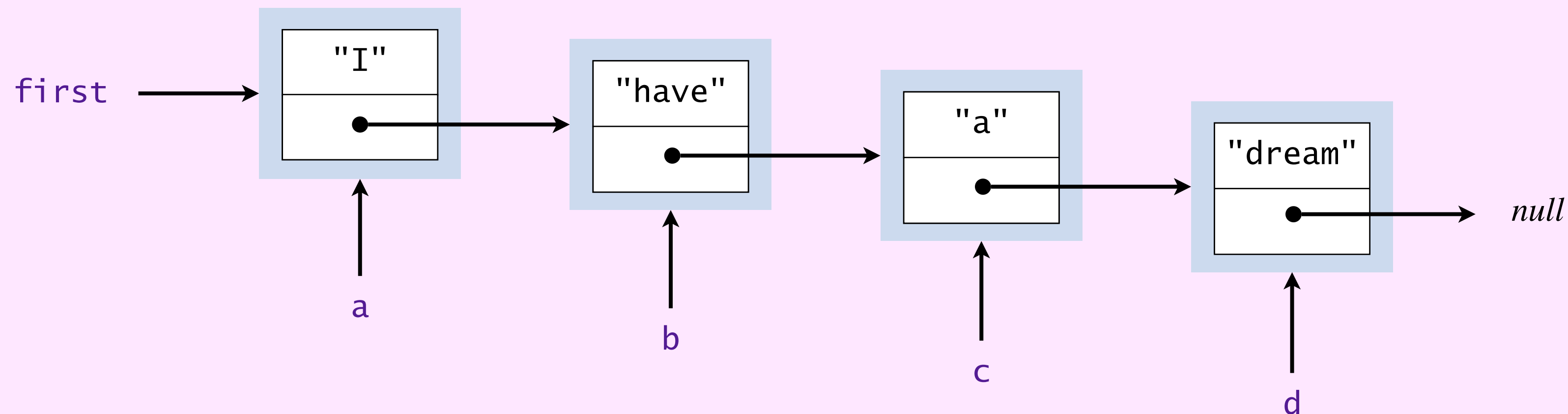
Node data type



Node object

```
Node a = new Node();  
Node b = new Node();  
Node c = new Node();  
Node d = new Node();  
a.item = "I";  
b.item = "have";  
c.item = "a";  
d.item = "dream";  
a.next = b;  
b.next = c;  
c.next = d;  
d.next = null;  
Node first = a;
```

creating a 4-node linked list

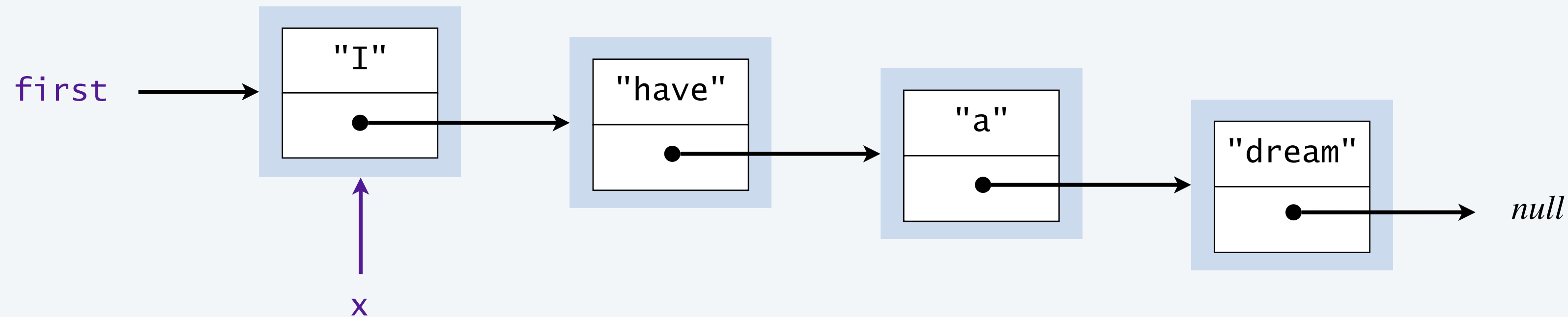


Traversing a singly linked list

Goal. Systematically process each node in a singly linked list.

Solution. Start at *first*, then repeatedly follow *next* links (using this *for*-loop idiom).

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

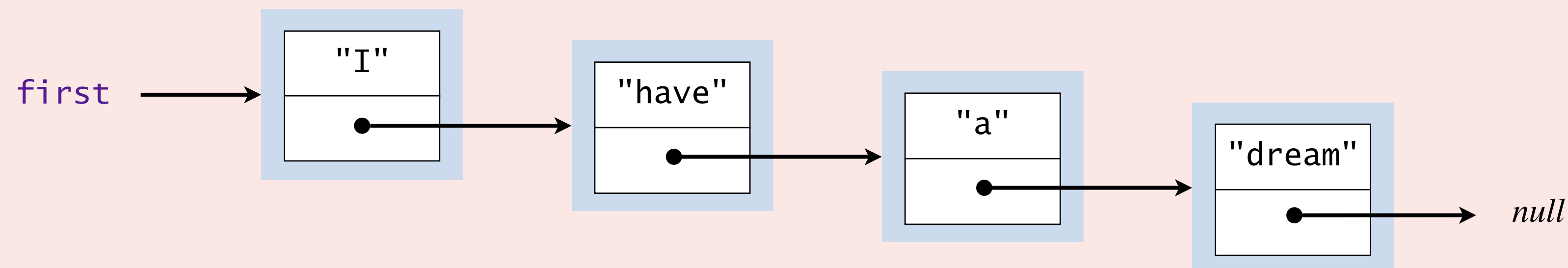




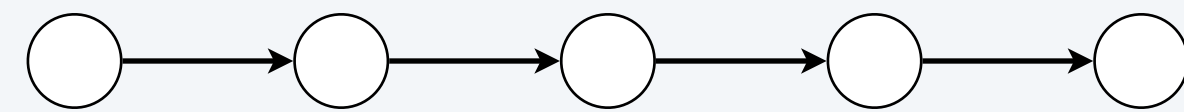
What effect does this statement have on the linked list starting at *first* ?

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

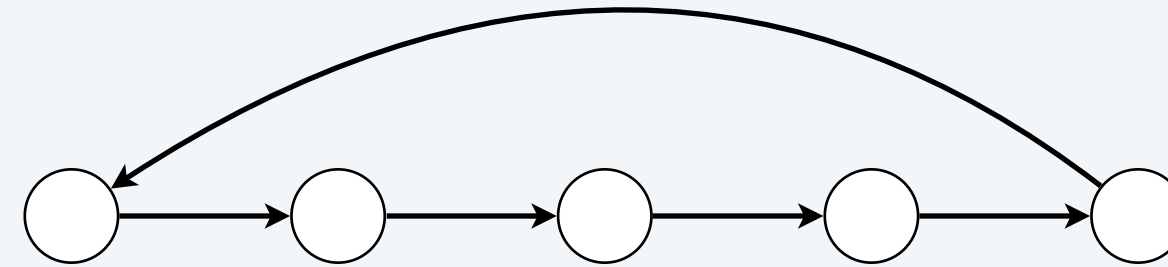
- A. Removes the node containing "I"
- B. Removes the node containing "have"
- C. Removes the node containing "a"
- D. Leaves the linked list unchanged



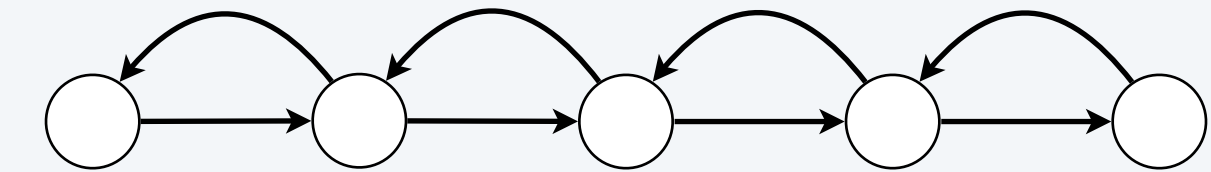
Linked data structures: nodes connected by links



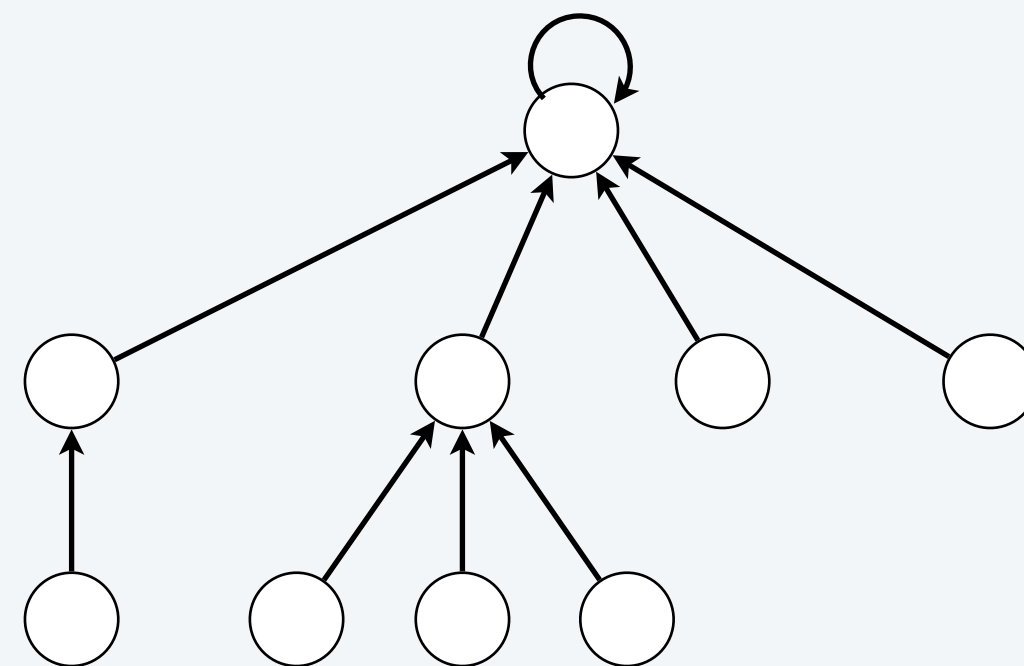
null-terminated singly linked list
(1 link per node)



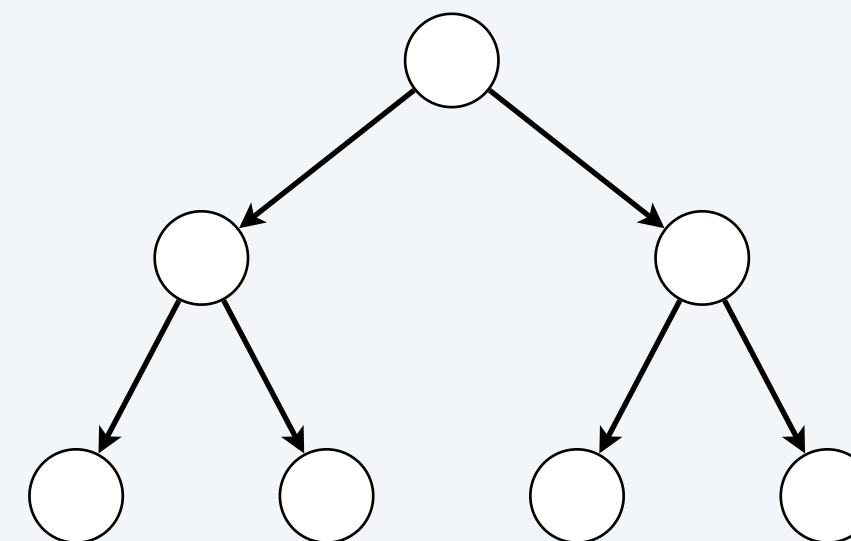
circular singly linked list
(1 link per node)



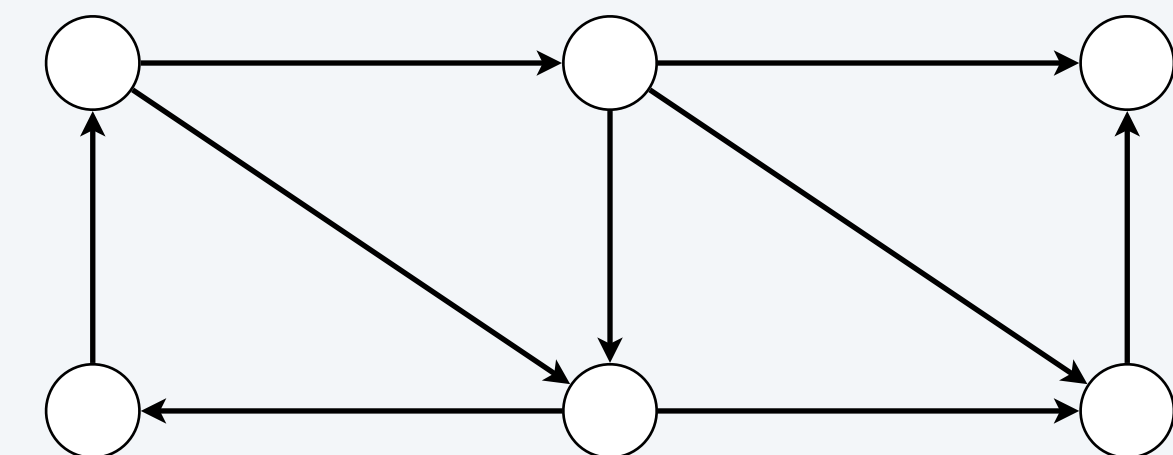
doubly linked list
(2 links per node)



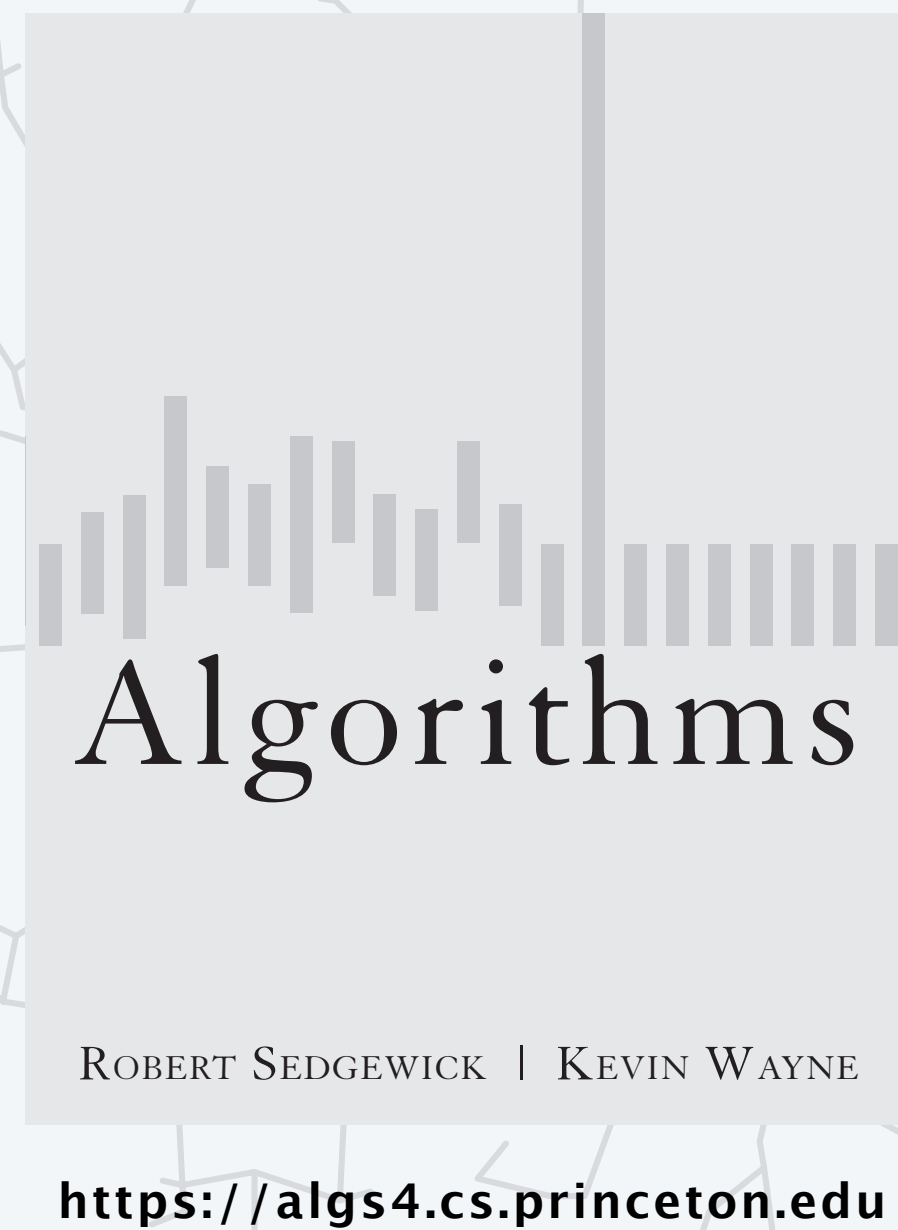
parent-link tree
(1 link per node)



binary tree
(2 links per node)



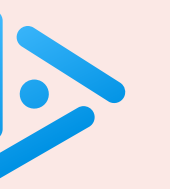
directed graph
(variable number of links per node)



1.3 STACKS AND QUEUES II

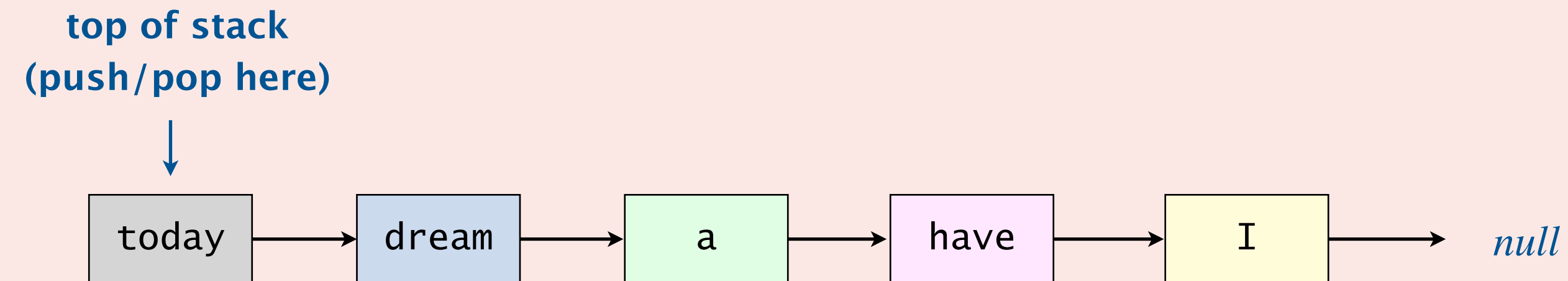
- ▶ *linked lists*
- ▶ *stacks (linked-list)*
- ▶ *queues (linked-list)*
- ▶ *iterators and for-each loops*
- ▶ *Java Collections Framework*



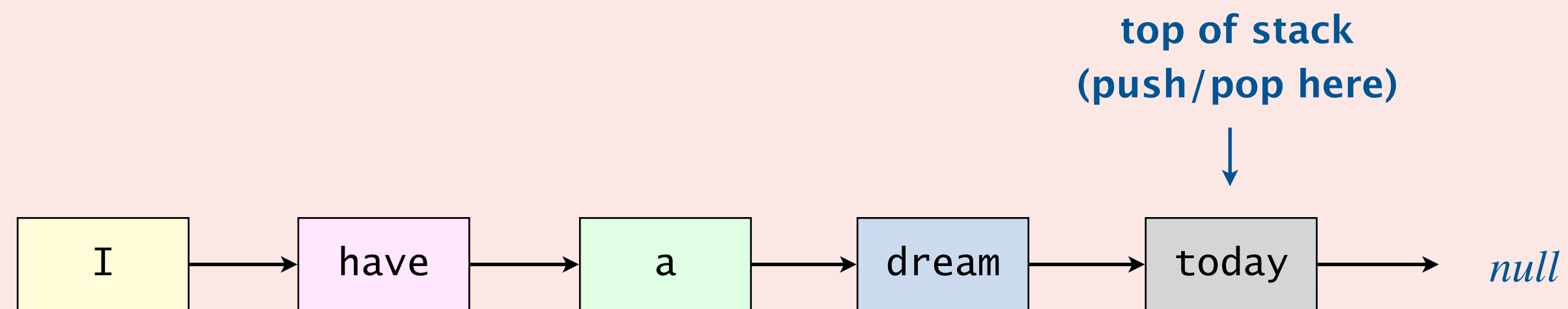


You are implementing a stack with a singly linked list.
Which representation supports $\Theta(1)$ push and $\Theta(1)$ pop?

A.



B.

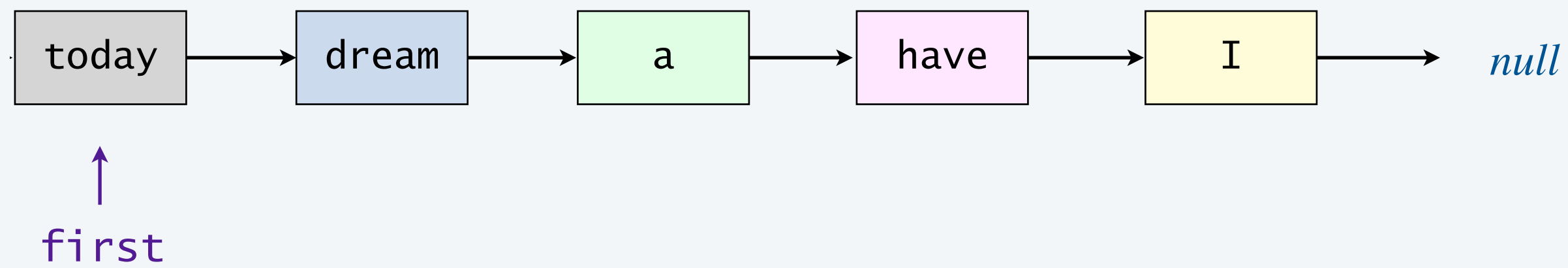


C. Both A and B

D. Neither A nor B

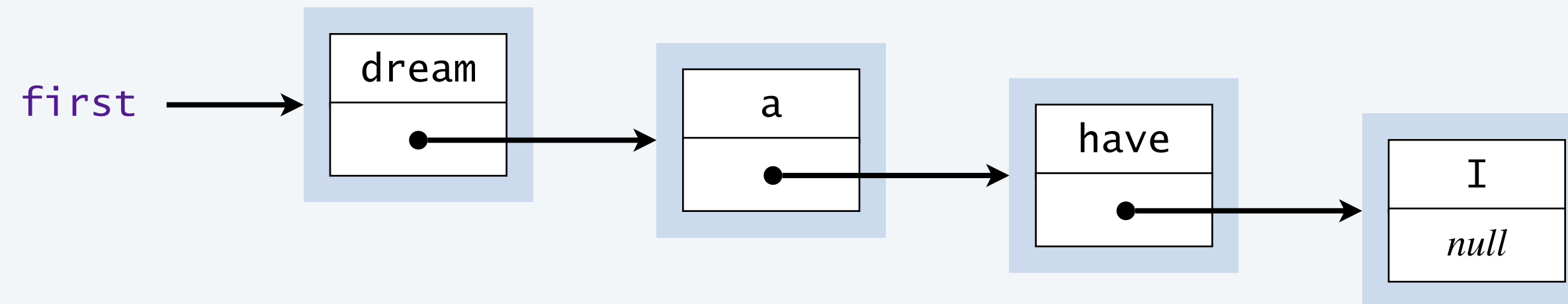
Implementing a stack with a linked list

- Maintain a link *first* to the first node in a singly linked list.
- Push: create a new node; link it to the *first*; update *first*.
- Pop: remove the first node by advancing *first* to the next node.



Implementing a stack with a linked list: pop

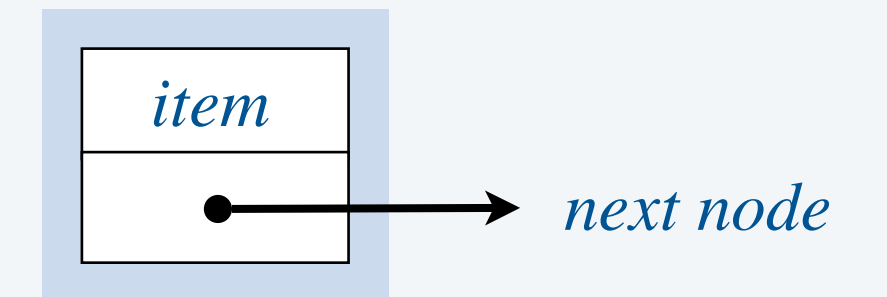
singly linked list



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

save the item to return

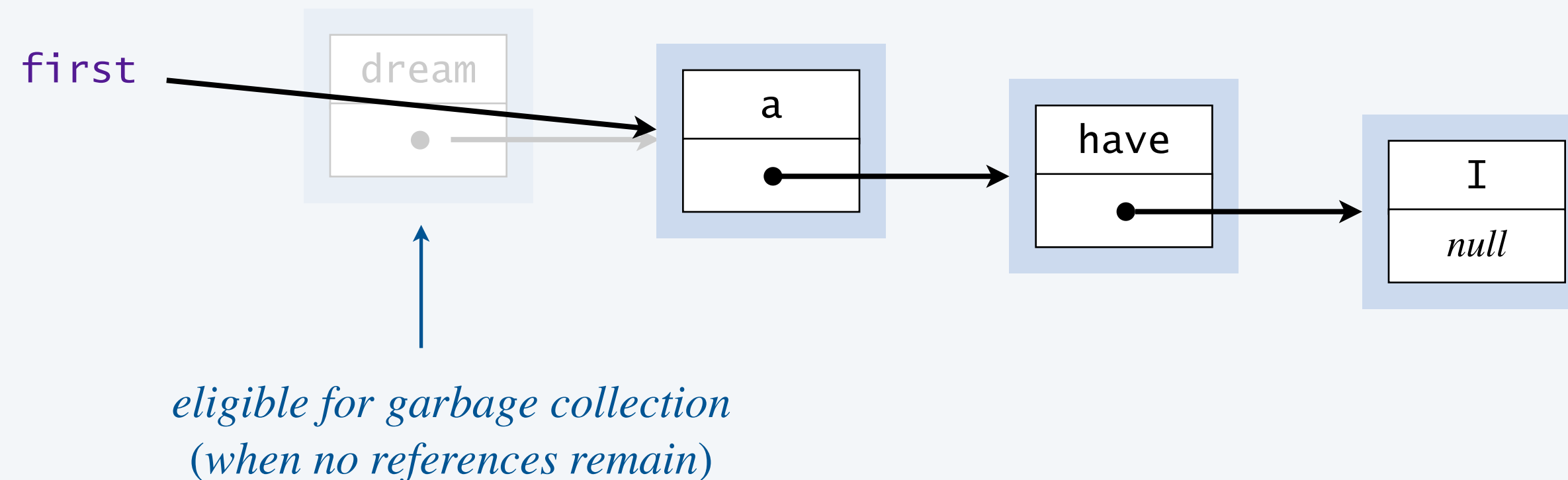
```
String item = first.item;
```



a Node object

remove the first node

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



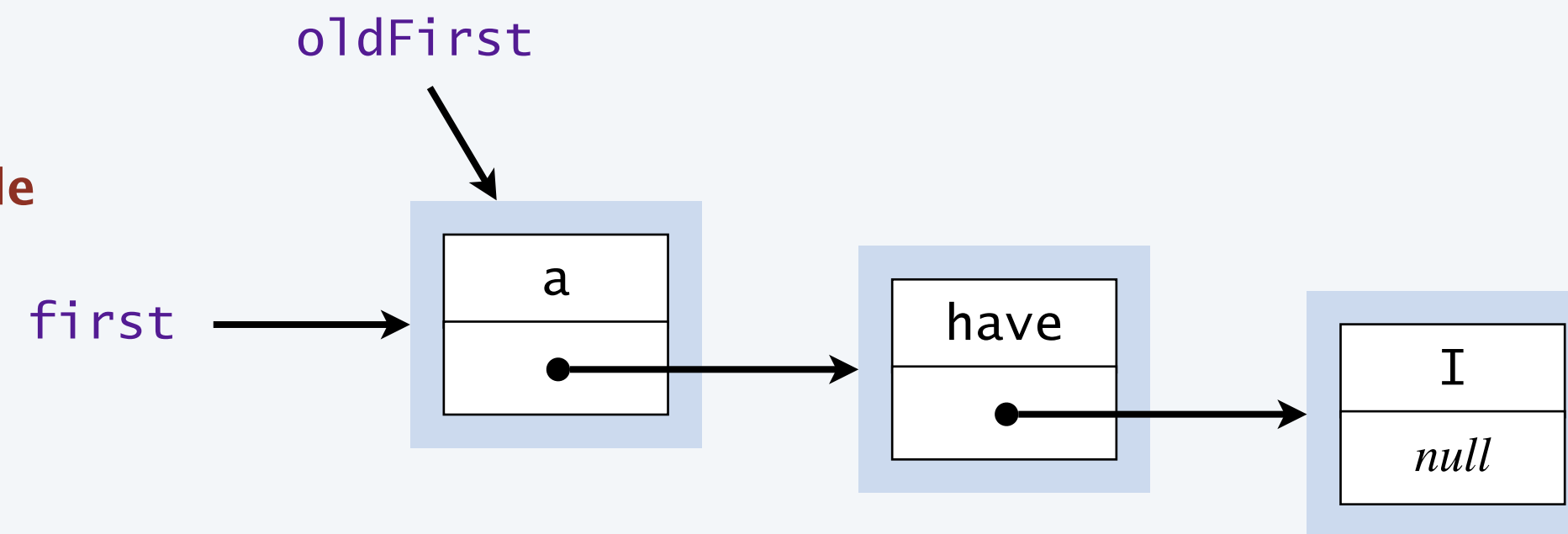
return the saved item

```
return item;
```


Implementing a stack with a linked list: push

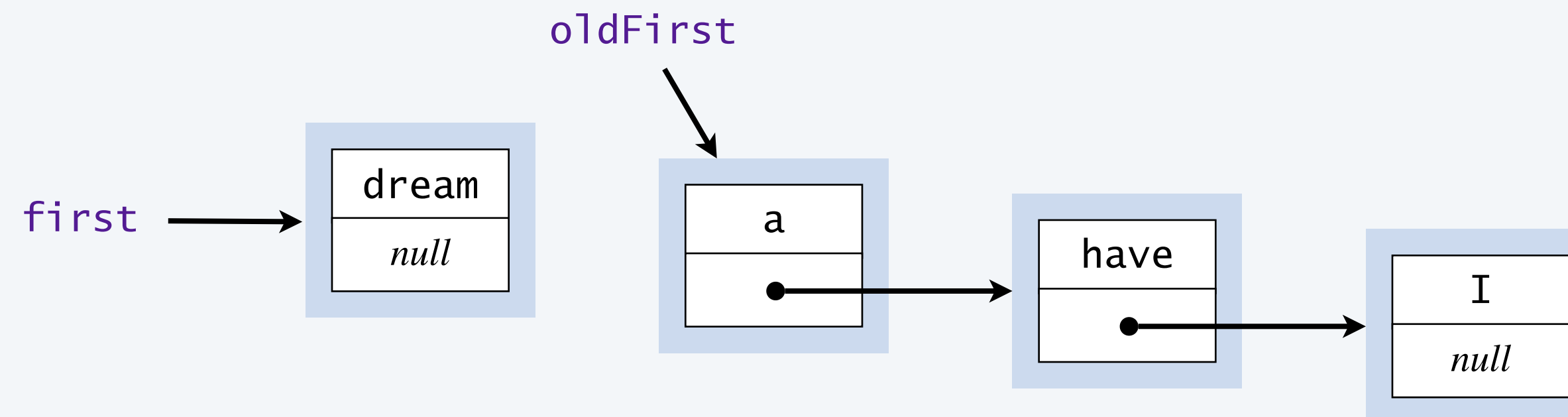
save a reference to the old first node

```
Node oldFirst = first;
```



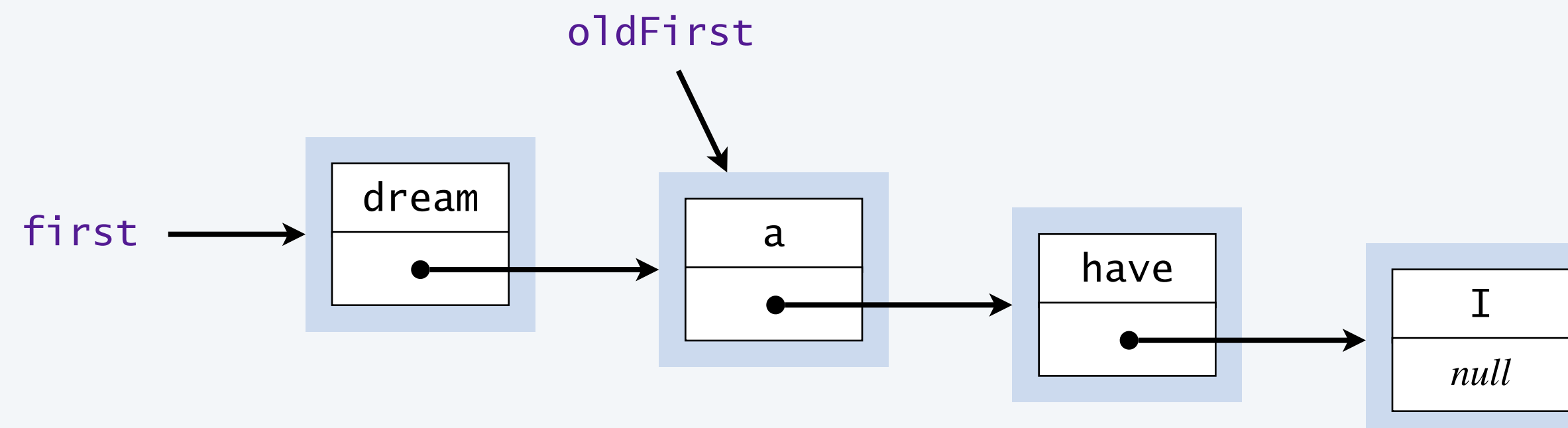
create a new node and update first

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

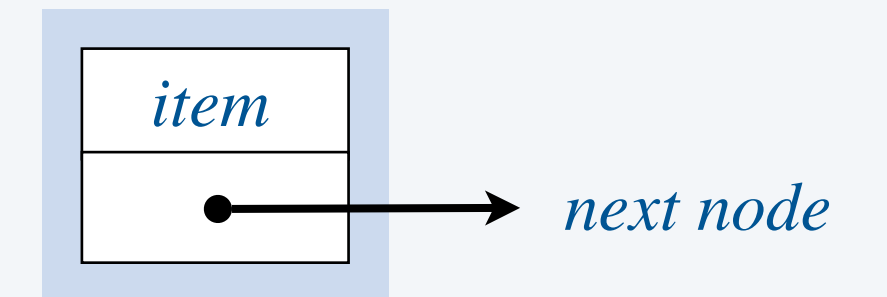


link the new node at the front

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



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



a Node object

Implementing a stack with a linked list: putting it all together

```
public class LinkedStack<Item> {  
    private Node first; // top of stack
```

← *generic stack: Item is a type variable, a placeholder for the concrete type (e.g., String or Integer)*

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

← *private nested class limits visibility to LinkedStack (outer class can still access private instance variables)*

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

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

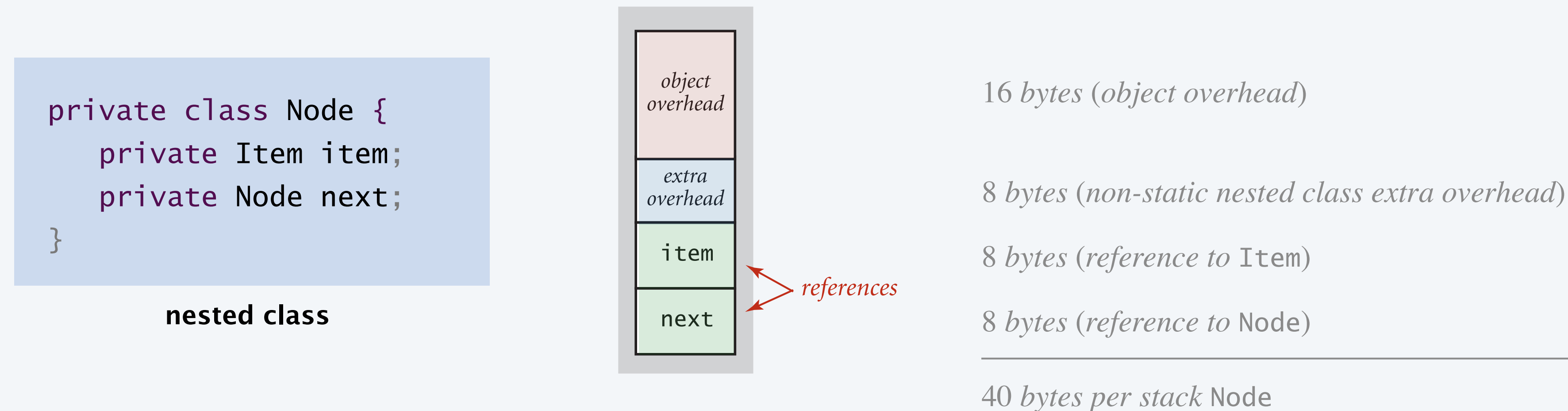
*no constructor declared ⇒
compiler generates a no-argument constructor
(instance variables default to null / 0 / false)*

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


Implementing a stack with a linked list: performance

Proposition. Every push and pop takes $\Theta(1)$ time in the worst case.

Proposition. A `LinkedStack` storing n items allocates n `Node` objects and uses $\sim 40n$ bytes.



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

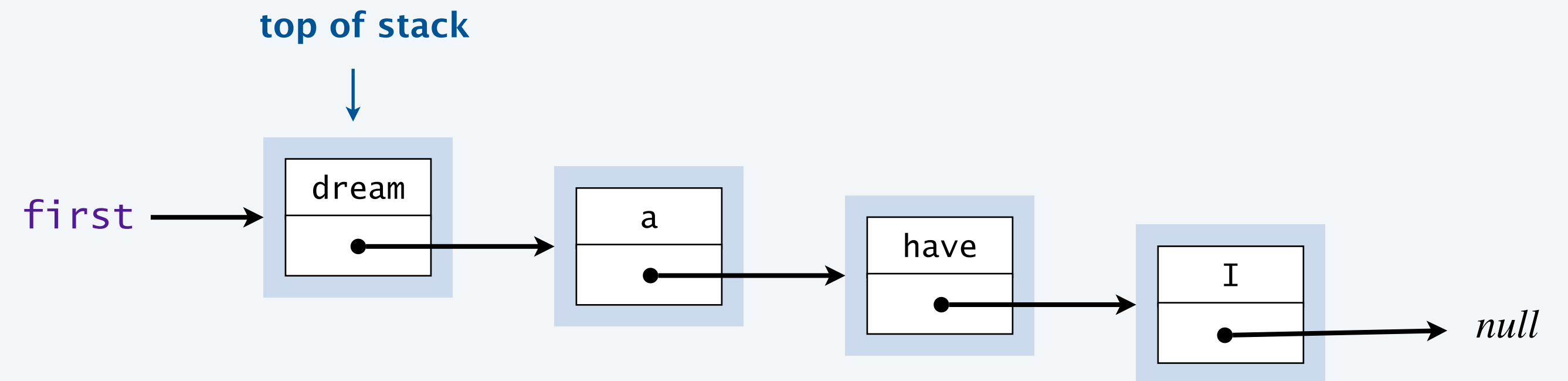
Stack implementations: resizable array vs. linked list

Q. We can implement the stack API with either a **resizable array** or a **linked list**. Which is more efficient?

A. It depends.

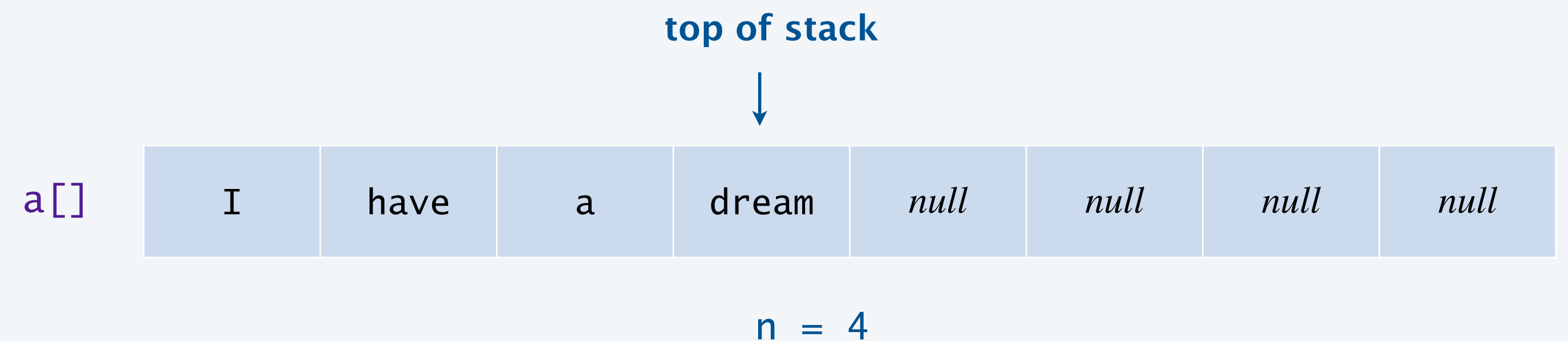
Linked-list implementation.

- $\Theta(1)$ worst-case time for push and pop.
- More memory (node object overhead).

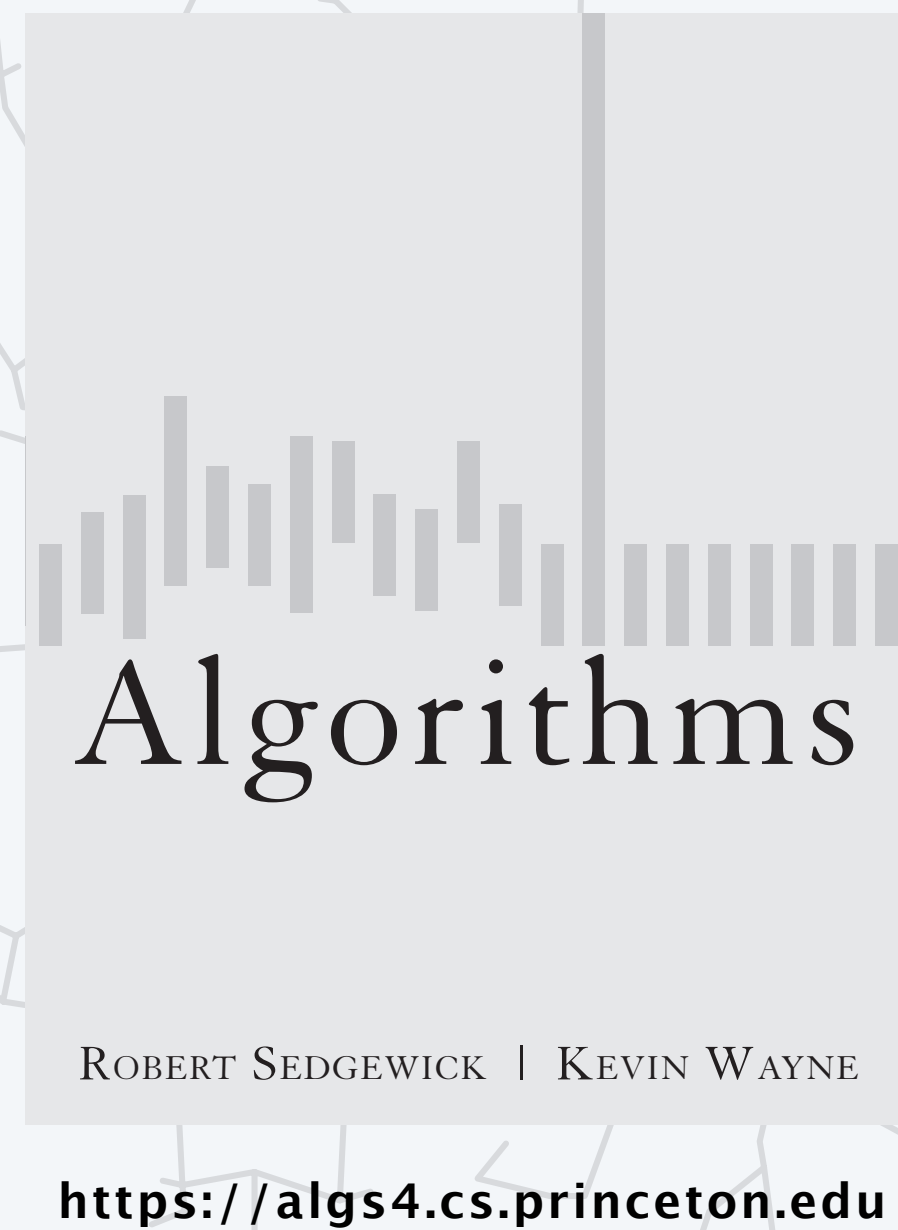


Resizable-array implementation.

- $\Theta(1)$ amortized time for push and pop.
- Less memory.
- Better cache locality.



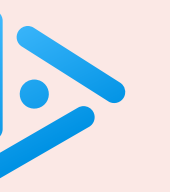
↑
*arrays store items contiguously;
linked lists store items non-contiguously*



1.3 STACKS AND QUEUES II

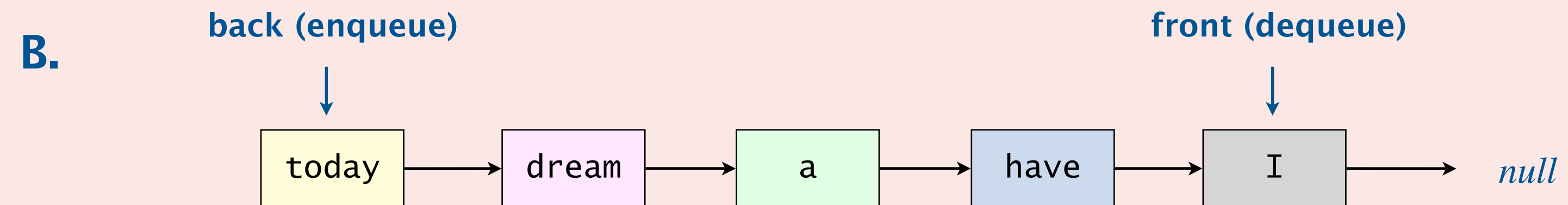
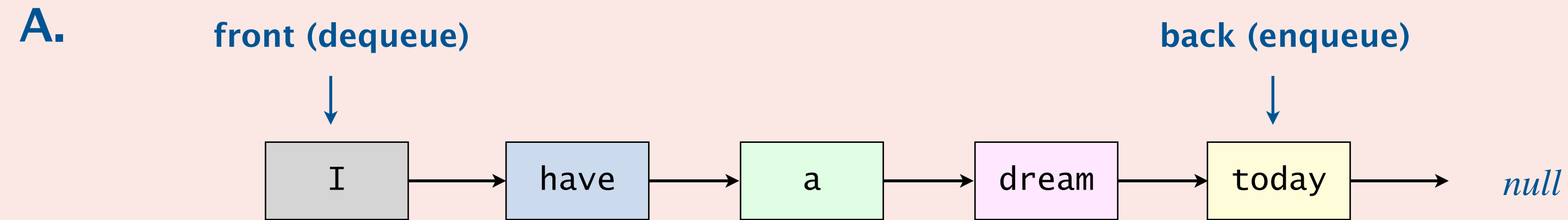
- ▶ *linked lists*
- ▶ *stacks (linked-list)*
- ▶ *queues (linked-list)*
- ▶ *iterators and for-each loops*
- ▶ *Java Collections Framework*





You are implementing a queue with a singly linked list.

Which representation supports $\Theta(1)$ enqueue and $\Theta(1)$ dequeue?

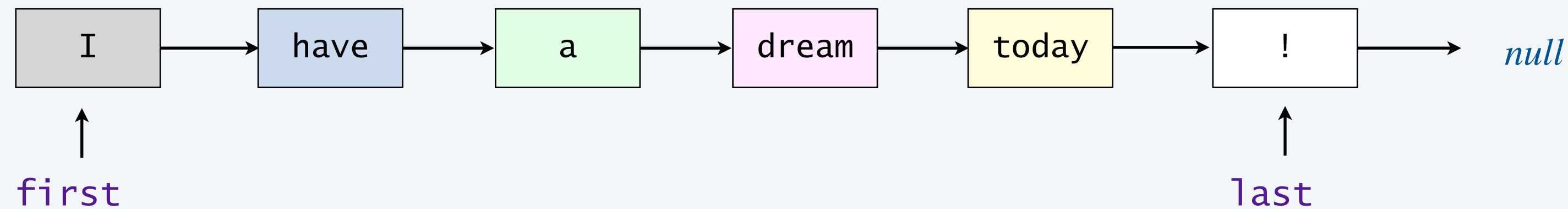


C. Both A and B

D. Neither A nor B

Implementing a queue with a linked list

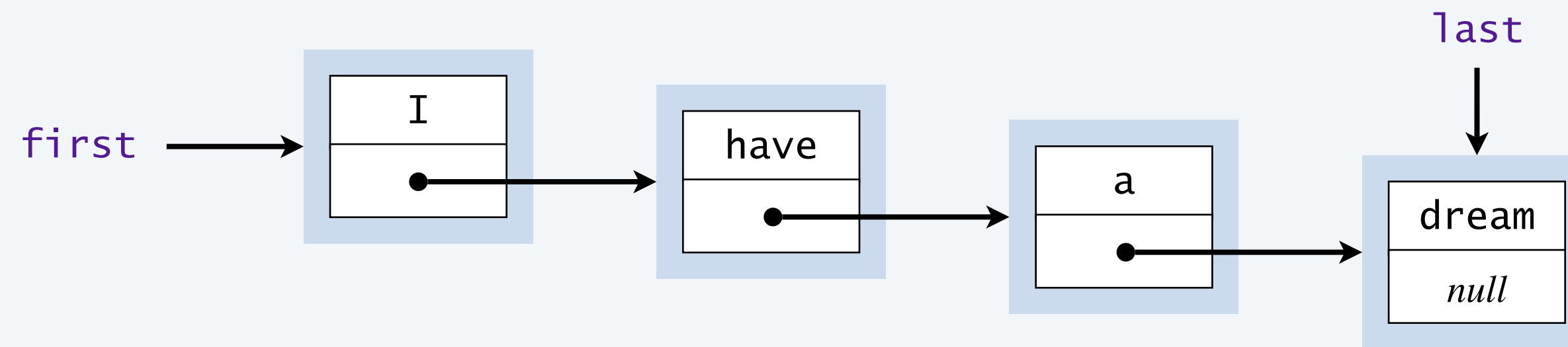
- Maintain a link *first* to the first node in a singly linked list.
- Maintain another link *last* to the last node.
- Enqueue: create a new node; link it after *last*; update *last*.
- Dequeue: remove the first node by advancing *first* to the next node.



Implementing a queue with a linked list: dequeue

Remark. Same basic implementation as `pop()`.

singly linked list

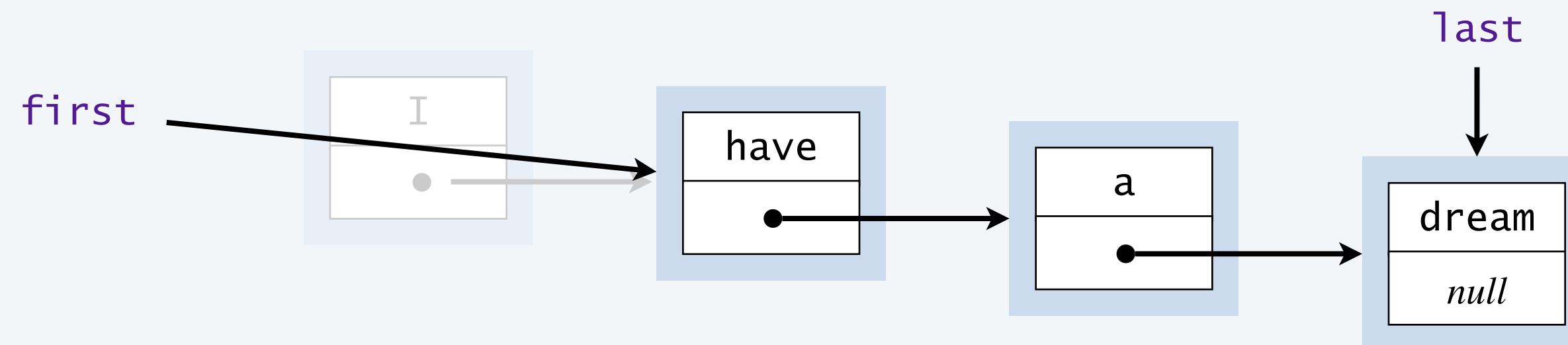


save the item to return

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

remove the first node

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



return the saved item

```
return item;
```

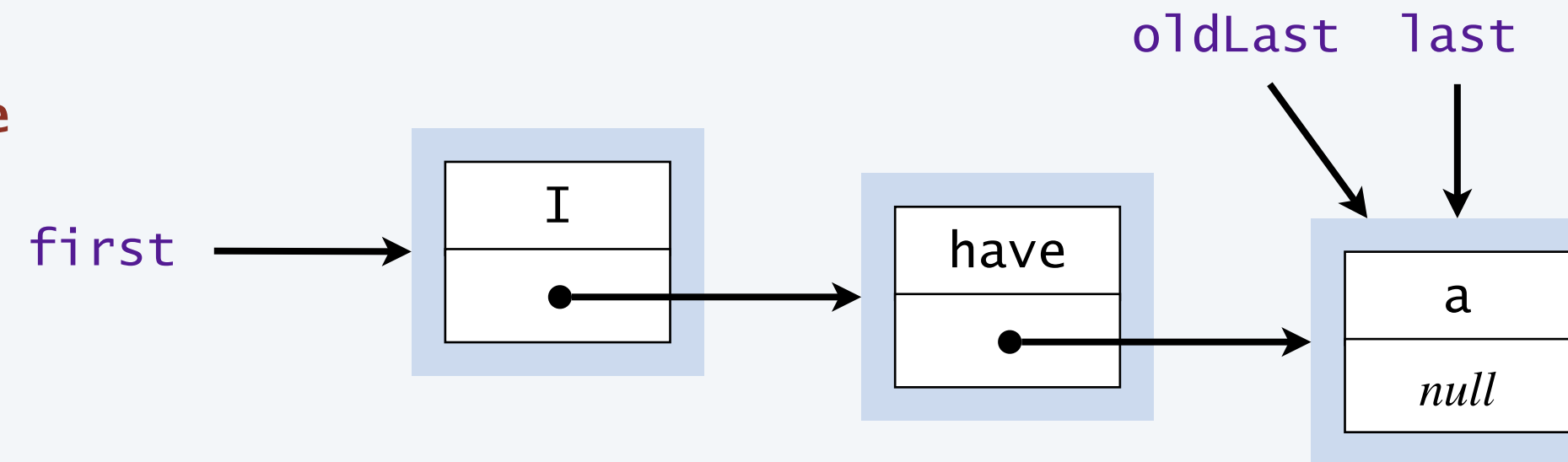
```
private class Node {  
    private String item;  
    private Node next;  
}
```

nested class

Implementing a queue with a linked list: enqueue

save a reference to the old last node

```
Node oldLast = last;
```

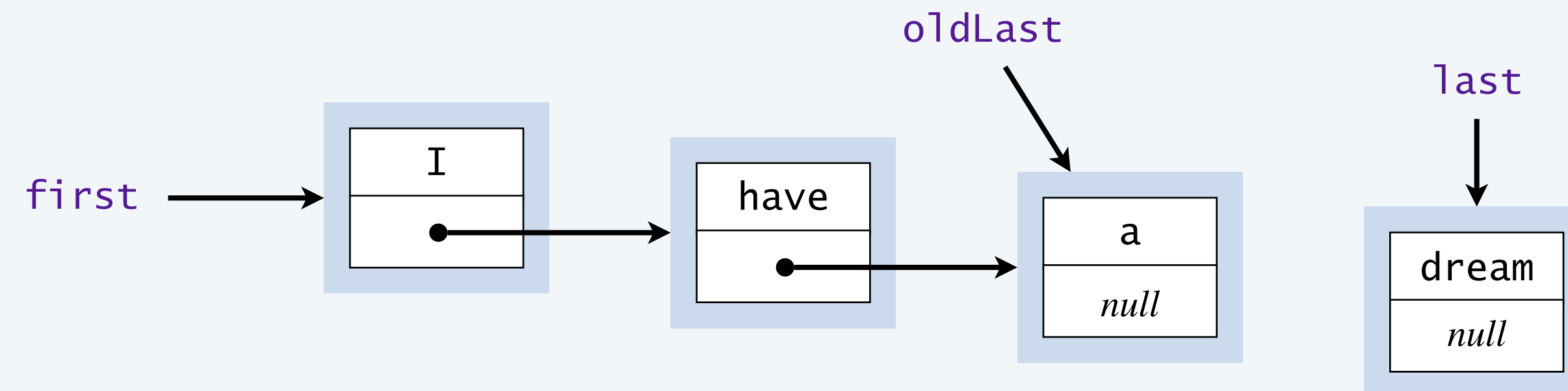


```
private class Node {  
    private String item;  
    private Node next;  
}
```

nested class

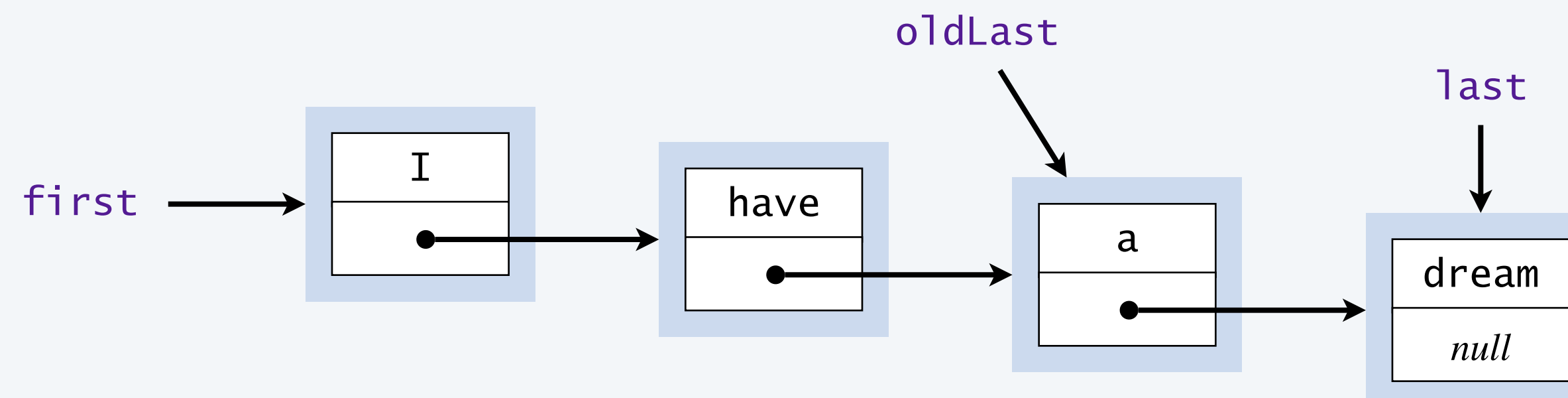
create a new node and update last

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



link the new node at the end

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



Implementing a queue with a linked list: putting it all together

```
public class LinkedQueue<Item> {  
    private Node first, last;  
  
    private class Node {  
        /* identical to LinkedStack */  
    }  
  
    public boolean isEmpty() {  
        return first == null;  
    }  
}
```

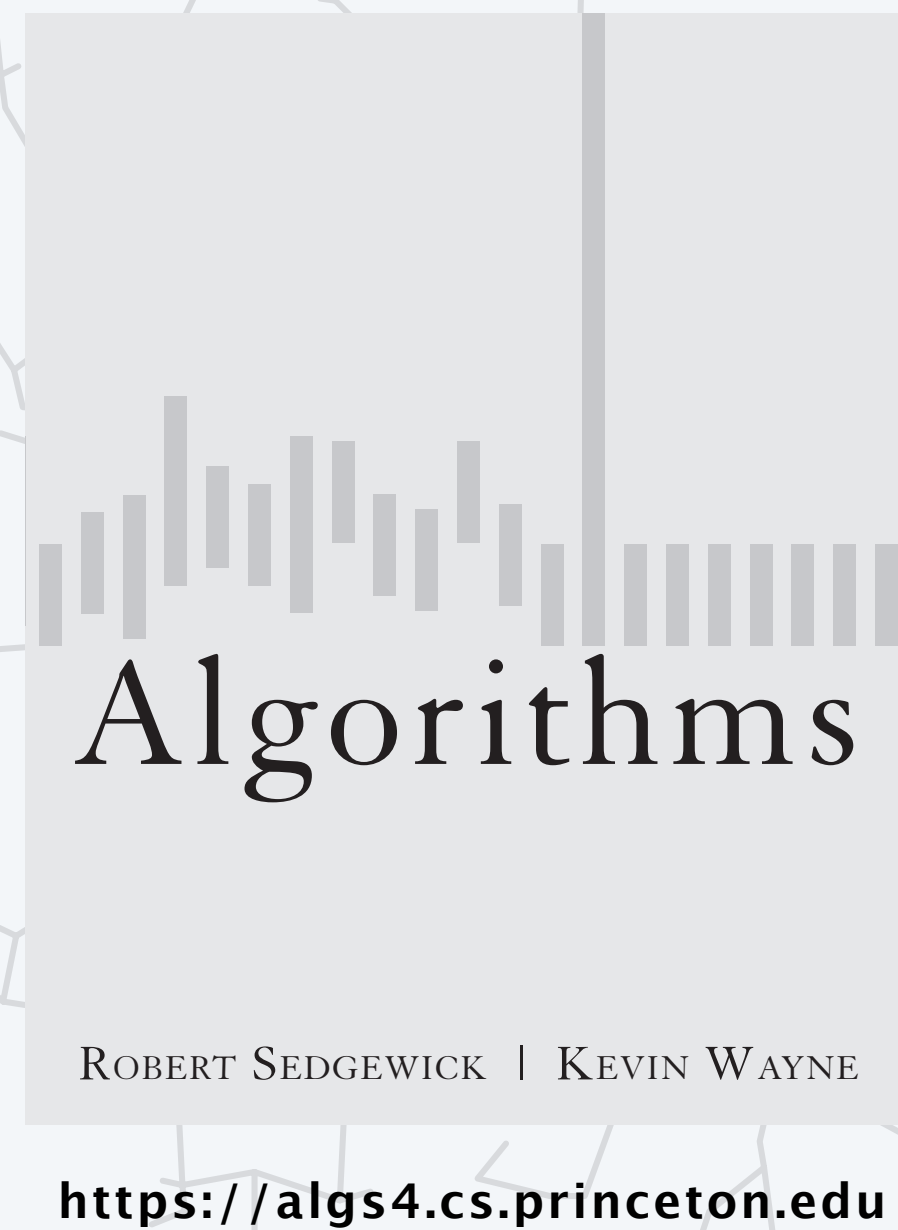
```
    public void enqueue(Item item) {  
        Node oldLast = last;  
        last = new Node();  
        last.item = item;  
        last.next = null;  
        if (first == null) first = last;  
        else  
            oldLast.next = last;  
    }
```

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

```
}
```

*corner case: enqueue into an empty queue
(must set first)*

*corner case: dequeue the last item
(set last = null to avoid loitering)*



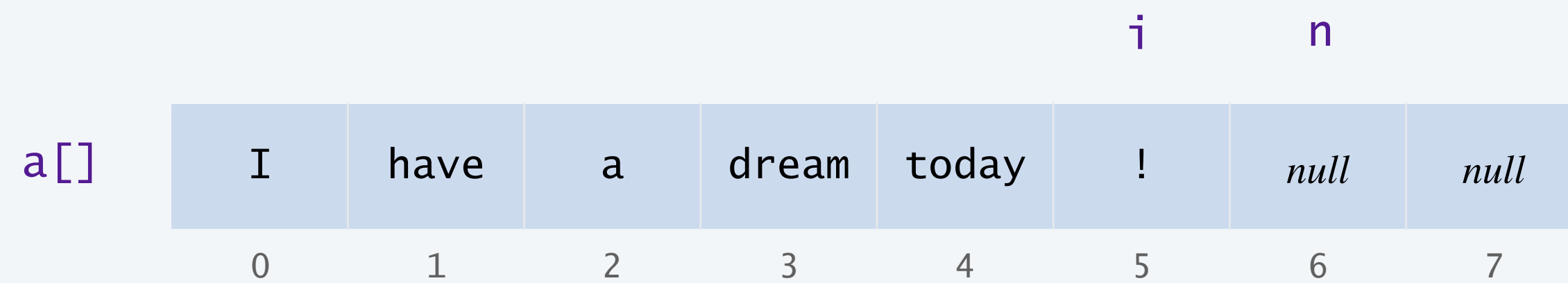
1.3 STACKS AND QUEUES II

- *linked lists*
- *stacks (linked-list)*
- *queues (linked-list)*
- *iterators and for-each loops*
- *Java Collections Framework*

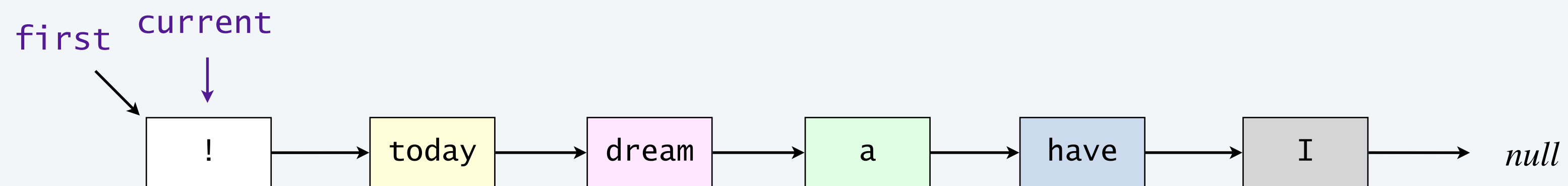
Iteration

Design challenge. Allow a client to **iterate over** a collection: access its items sequentially, without exposing the collection's internal representation.

stack (resizable-array representation)



stack (linked-list representation)



Java solution. Clients use a **for-each** loop.



Java provides concise syntax for iterating over a collection.

“for-each” loop (shorthand)

```
Stack<String> stack = new Stack<>();  
...  
  
for (String s : stack) {  
    // do something with s  
}
```

equivalent code (longhand)

```
Stack<String> stack = new Stack<>();  
...  
  
Iterator<String> iterator = stack.iterator();  
while (iterator.hasNext()) {  
    String s = iterator.next();  
    // do something with s  
}
```

To support a for-each loop:

- A collection must implement `Iterable` by defining `iterator()`, which returns an `Iterator`.
- An `Iterator` represents the **current position** in an iteration.
 - `hasNext()` returns `true` if the iterator has more items
 - `next()` returns the next item and advances the iterator to the next position

Iterator and Iterable interfaces



Java defines two **interfaces** that support for-each loops.  *Java interface = set of related methods that define some behavior (partial API)*

- **Iterable**: specifies `iterator()`, which returns an **Iterator**.
- **Iterator**: specifies `hasNext()` and `next()`.
- Both interfaces are generic.

java.lang.Iterable interface (relevant method)

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

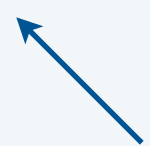
“I am a collection that can be traversed with a for-each loop.”

java.util.Iterator interface (relevant methods)

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

“I represent the state of a traversal.”

Type safety. A for-each loop won't compile unless the collection is **Iterable** (or an array).

 *ensures the (implicit) call to
`iterator()` is valid at compile time*

Stack iterator: resizable-array implementation

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class ResizableArrayStack<Item> implements Iterable<Item> {
    private int n;    // number of items in the stack
    private Item[] a; // stack items
    ...

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

    private class ReverseArrayIterator implements Iterator<Item> {
        private int i = n-1; // index of next item to return

        public boolean hasNext() {
            return i >= 0;
        }

        public Item next() {
            if (!hasNext()) throw new NoSuchElementException();
            return a[i--];
        }
    }
}
```

collection implements the Iterable interface

object you return must implement the Iterator interface

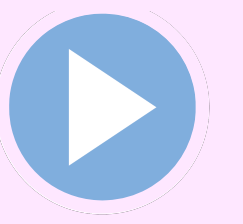
code in inner class can access instance variables in outer class

Iterator API says to throw this exception if called after traversal is complete

i *n*

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

Stack iterator: linked-list implementation (in IntelliJ)



```
import java.util.Iterator;
import java.util.NoSuchElementException;

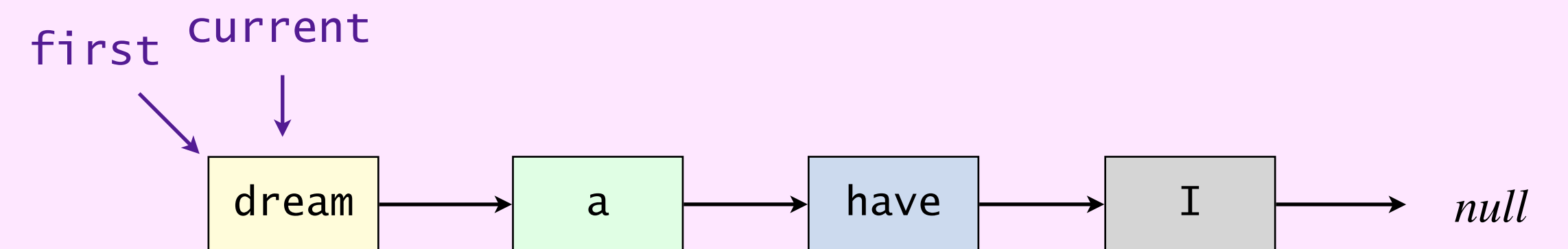
public class LinkedStack<Item> implements Iterable<Item> {
    private Node first;
    ...

    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() {
            if (!hasNext()) throw new NoSuchElementException();
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

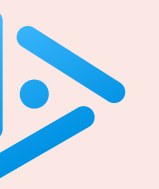




Suppose that you push A, B, and C onto a **stack** (linked list or resizable array), in that order. What does this code do?

```
for (String s : stack)
    for (String t : stack)
        StdOut.println(s + "-" + t);
```

- A. Prints A-A A-B A-C B-A B-B B-C C-A C-B C-C
- B. Prints C-C C-B C-A B-C B-B B-A A-C A-B A-A
- C. Run-time exception
- D. Depends on the implementation

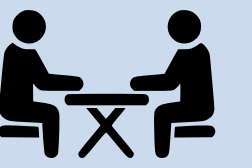


Suppose that you push A, B, and C onto a **stack** (linked list or resizable array), in that order. What does this code do?

```
for (String s : stack) {  
    StdOut.println(s);  
    StdOut.println(stack.pop());  
    stack.push(s);  
}
```

modifies stack

- A. Prints C C B B A A
- B. Prints C C B C A B
- C. Prints C C C C C C C C ...
- D. Run-time exception
- E. Depends on the implementation



Q. What happens if a client modifies a collection **while** iterating over it?

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

concurrent modification

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

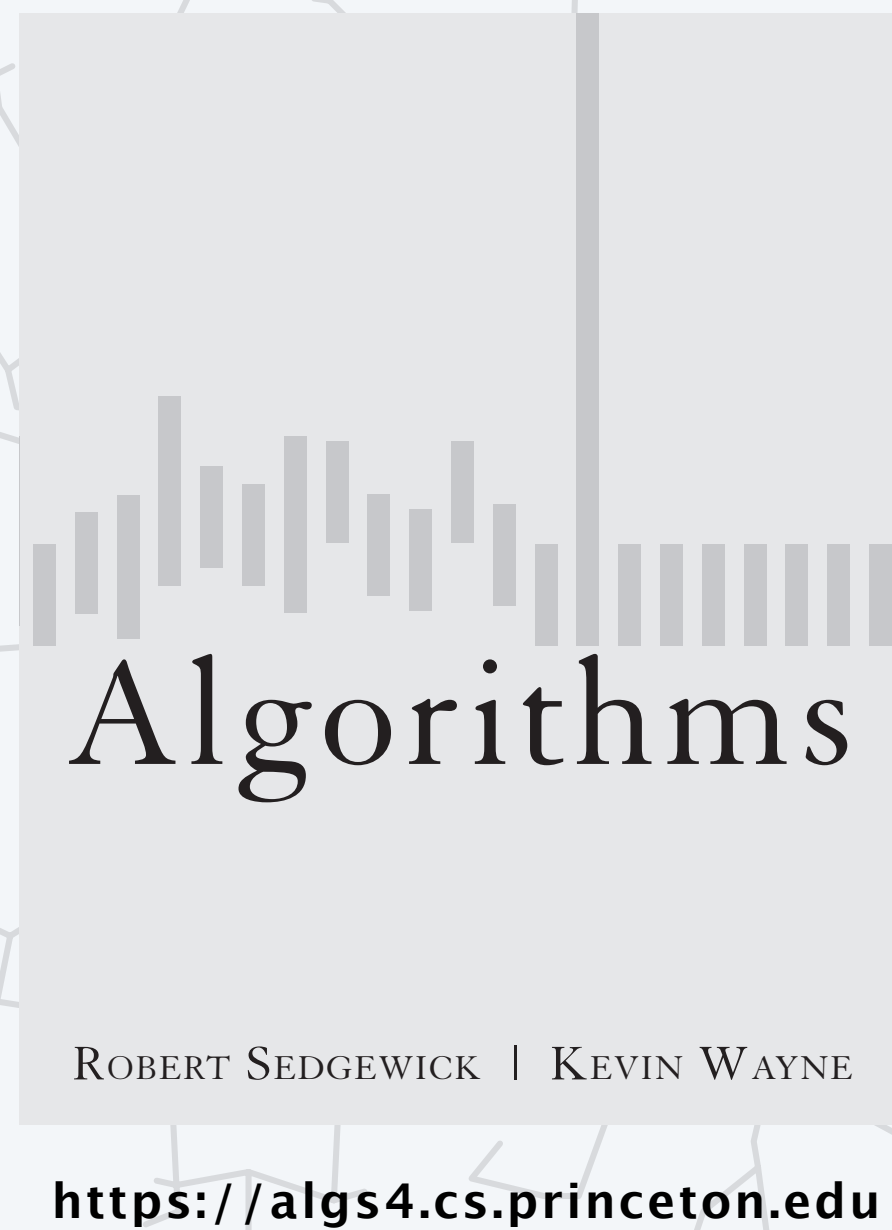
Java iterators: summary

Iterator and Iterable. Two Java interfaces that let a client **iterate over** a collection, without exposing its internal representation.

```
Stack<String> stack = new Stack<>();  
...  
  
// for-each loop  
for (String s : stack) {  
    ...  
}
```

In this course, you will

- Use iterators in client code.
- implement iterators (Assignment 2 only).



1.3 STACKS AND QUEUES II

- *linked lists*
- *stacks (linked-list)*
- *queues (linked-list)*
- *iterators and for-each loops*
- ***Java Collections Framework***

Java Collections Framework

Java libraries for collection data types.

- `java.util.LinkedList` [doubly linked list]
- `java.util.ArrayList` [resizable array]
- `java.util.ArrayDeque` [resizable circular array]
- `java.util.TreeMap` [red-black BST]
- `java.util.HashMap` [hash table]

OVERVIEWMODULEPACKAGECLASSUSETREEDEPRECATEDINDEXHELP

ALL CLASSES

SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHODDETAIL: 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.

OVERVIEWMODULEPACKAGECLASSUSETREEDEPRECATEDINDEXHELP

ALL CLASSES

SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHODDETAIL: 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. Foundation for understanding performance guarantees.

Best practices.

- Use `Stack` and `Queue` from `algs4.jar` for stacks and queues to improve design and efficiency.
- Use `java.util.ArrayList` or `java.util.LinkedList` when other operations needed.
(Some operations take linear time.)

COS 226 story (from Assignment 1)

Goal. Randomly open sites in an n -by- n percolation system; stop when the system percolates.

Jenny.

- Pick (row, col) uniformly at random; if already open, repeat.
- Expected time is $\Theta(n^2)$.

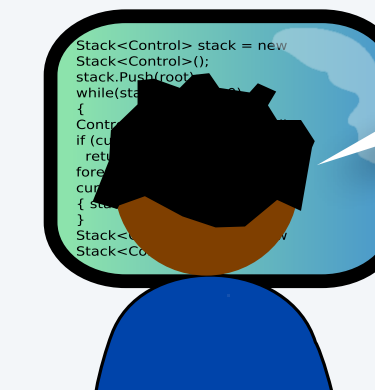
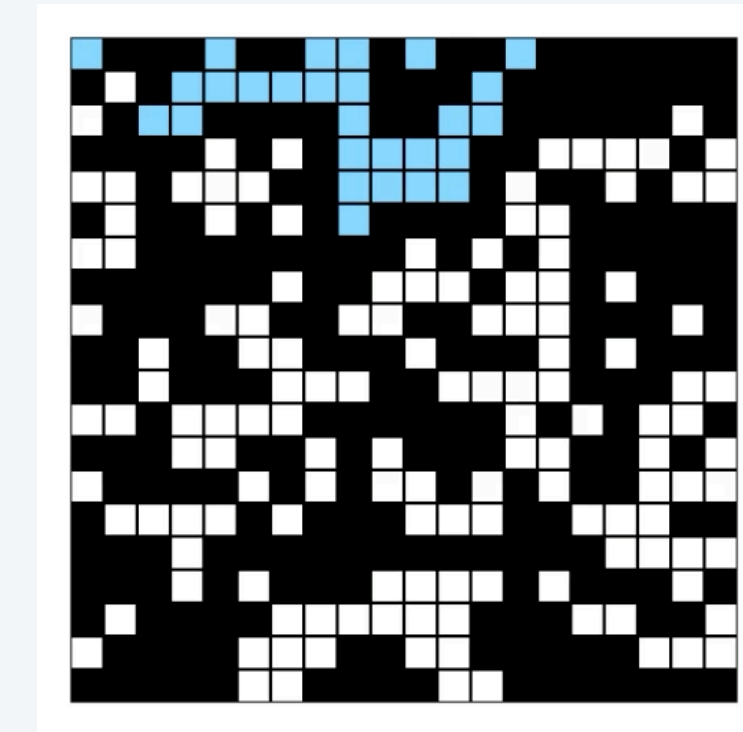
Kenny.

- Create a `java.util.ArrayList` to store the n^2 sites (initially blocked).
- Pick an index at random and remove from list.
- Expected time is $\Theta(n^4)$.

*each removal takes $\Theta(n^2)$
time in the worst case*

Lesson. Understand an API's performance guarantees before using it!

This course. Do not use a library until we have implemented it in class.



Kenny

Why is my program so slow ?

Stacks and queues: summary

Fundamental data types.

- Value: a **collection** of items of the same type.
- Core operations: **add**, **remove**, iterate, test if empty.

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

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

Efficient implementations.

- Resizable array: amortized $\Theta(1)$ time for push/pop or enqueue/dequeue.
- Singly linked list: worst-case $\Theta(1)$ time for push/pop or enqueue/dequeue.



Credits

image	source	license
<i>Assignment Logo</i>	Kathleen Ma '18	by author
<i>Stack of Books</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Long Queue Line</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>People Standing in Line</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Stack of Sweaters</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Programmer Icon</i>	<u>Jaime Botero</u>	<u>public domain</u>
<i>ChatGPT Phone</i>	<u>Adobe Stock</u>	<u>Education License</u>

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. ”*

