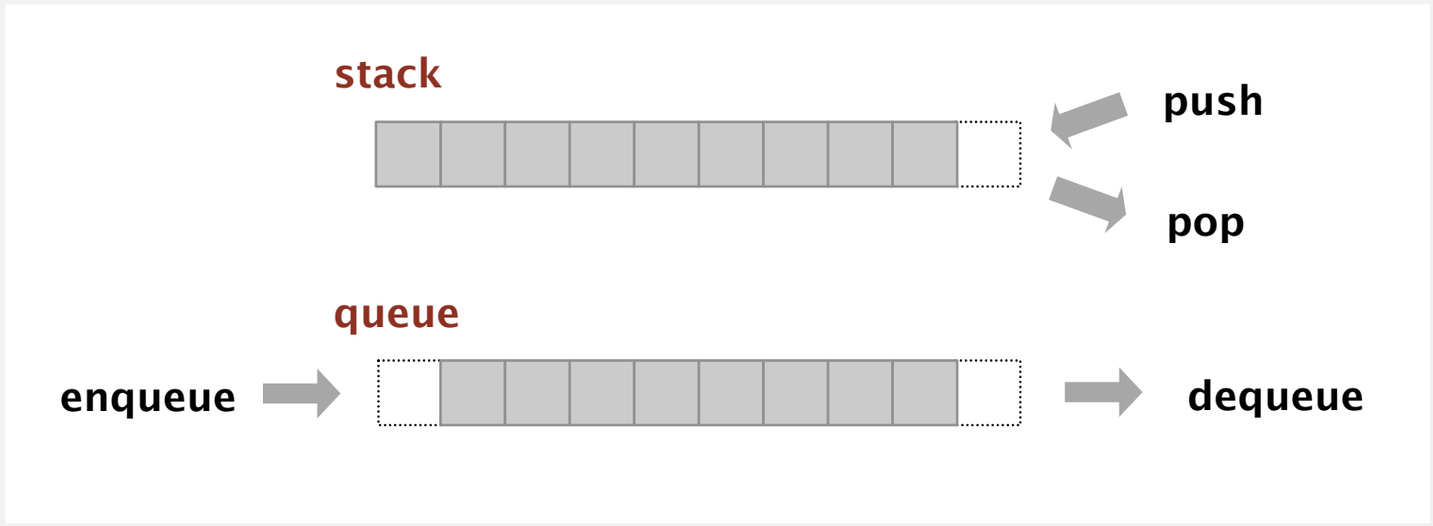
# Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: insert, remove, iterate, test if empty.
- Intent is clear when we insert.
- 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, interface

Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find, ....

Benefits.

- Client can't know details of implementation $\Rightarrow$
  client has many implementation from which to choose.
- Implementation can't know details of client needs $\Rightarrow$
  many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

Client: program using operations defined in interface.

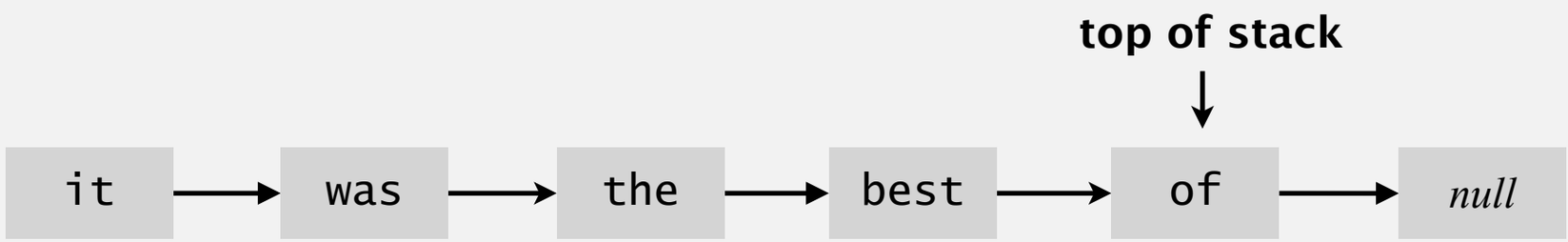
Implementation: actual code implementing operations.

Interface: description of data type, basic operations.

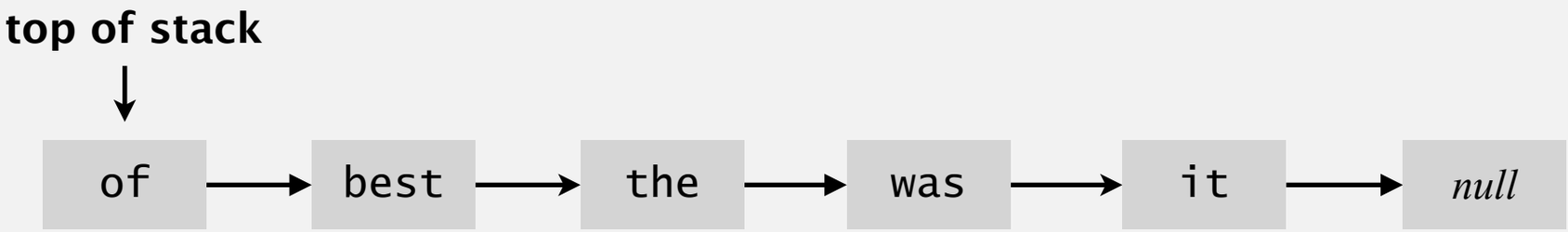
# How to implement a stack with a linked list?

A. Can't be done efficiently with a singly-linked list.

B.

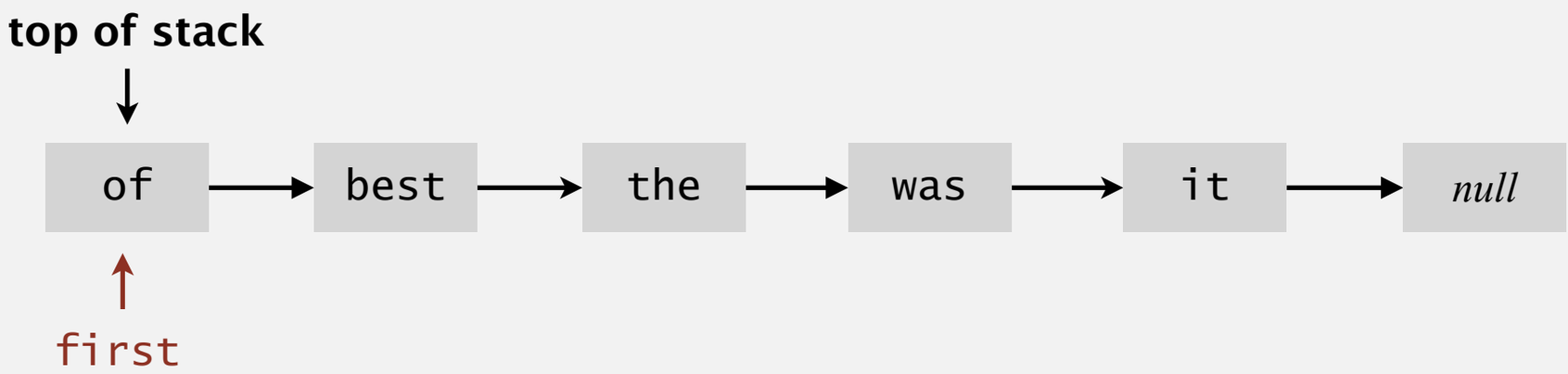



C.



# 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

inner class

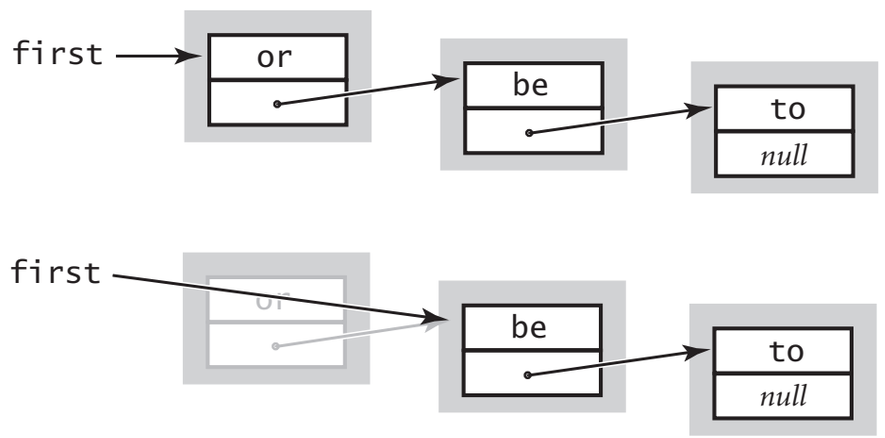
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

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

delete first node

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




return saved item

```
return item;
```

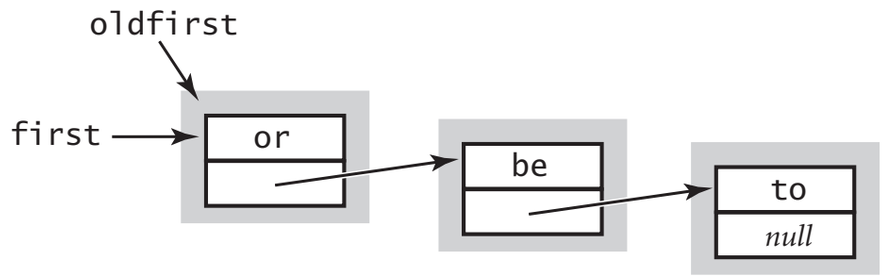
# Stack push: linked-list implementation

**inner class**

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

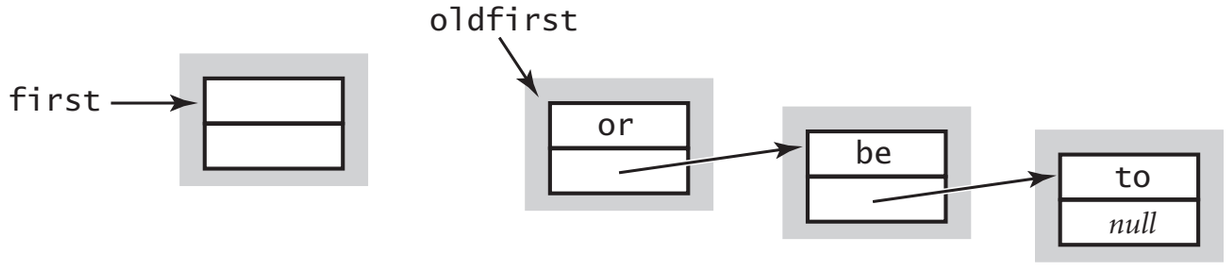
**save a link to the list**

```
Node oldfirst = first;
```



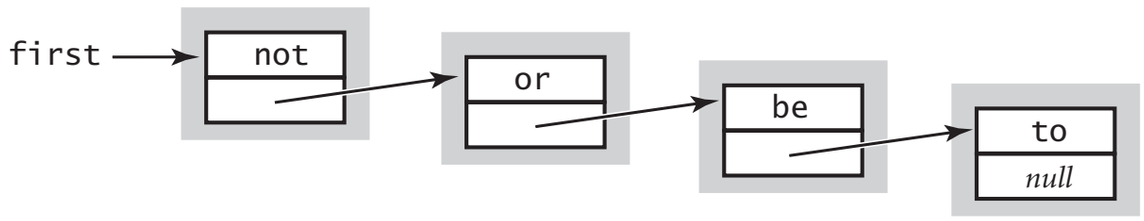

**create a new node for the beginning**

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




**set the instance variables in the new node**

```
first.item = "not";
first.next = oldfirst;
```



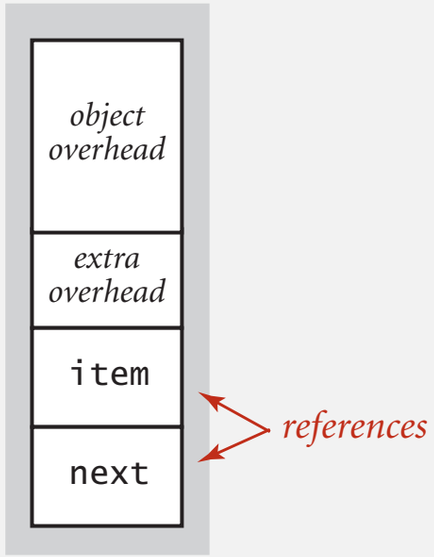
# Stack: linked-list implementation performance

Proposition. Every operation takes constant time in the worst case.

Proposition. A stack with $N$ items 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 accounts for the memory for the stack
(but not the memory for strings themselves, which the client owns).

## Stack implementations: resizing array vs. linked list

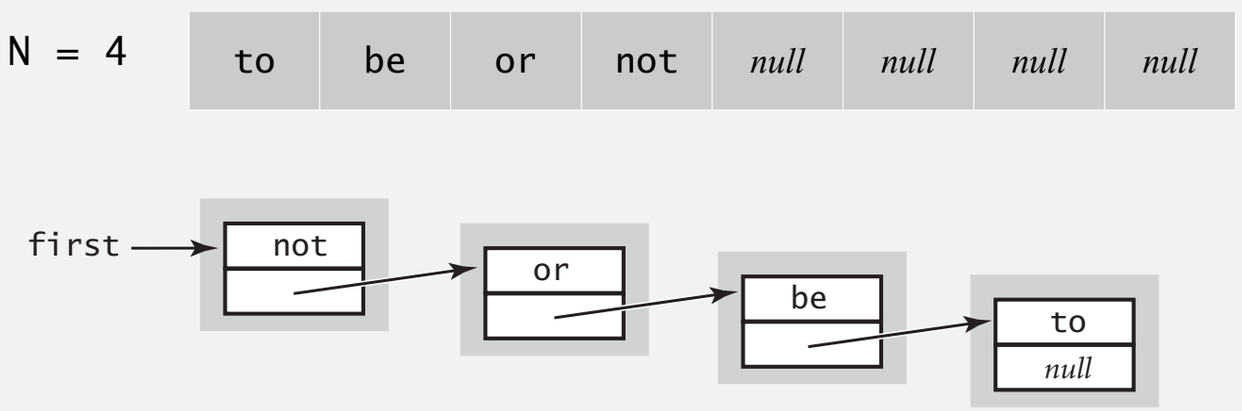
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation.

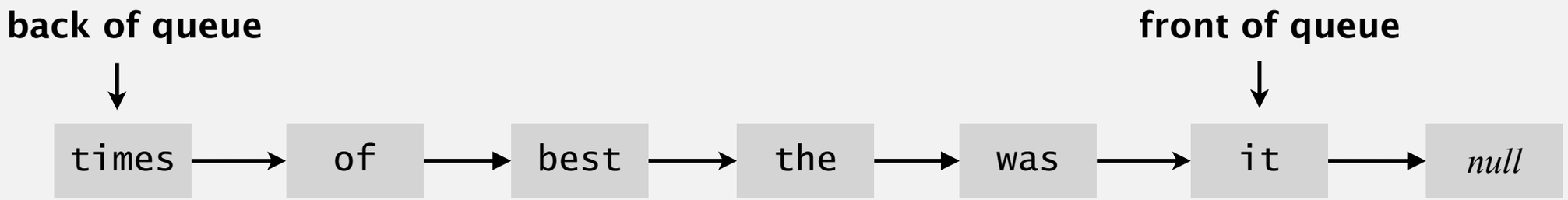
- Every operation takes constant amortized time.
- Less wasted space.



# How to implement a queue with a linked list?

A. Can't be done efficiently with a singly-linked list.

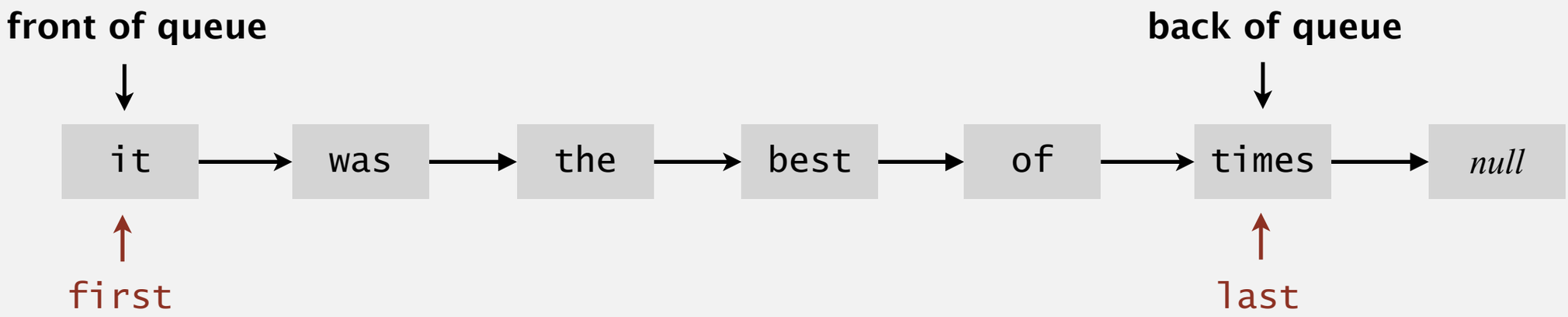
B.




C.

# 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

**inner class**

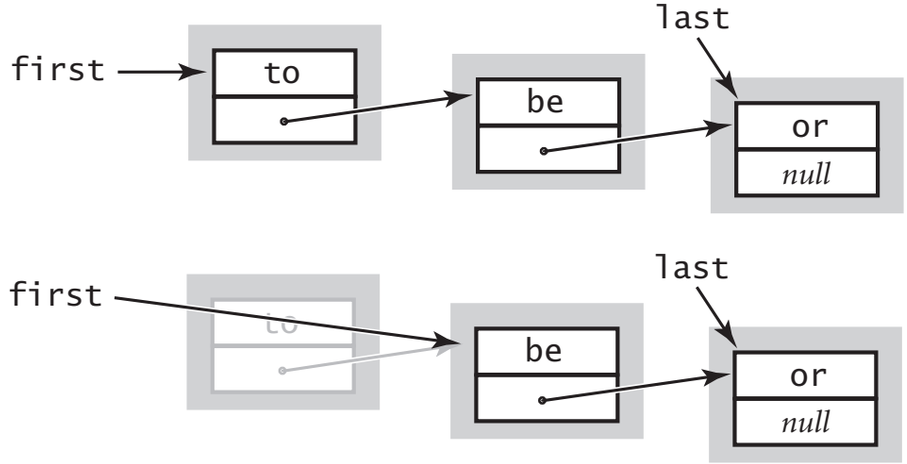
```
private class Node
{
   String item;
   Node next;
}
```

**save item to return**

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

**delete first node**

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




**return saved item**

```
return item;
```

Remark. Identical code to linked-list stack `pop()`.

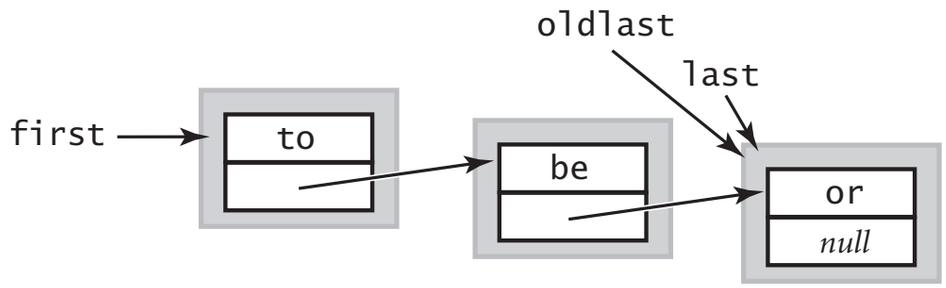
# Queue enqueue: linked-list implementation

inner class

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

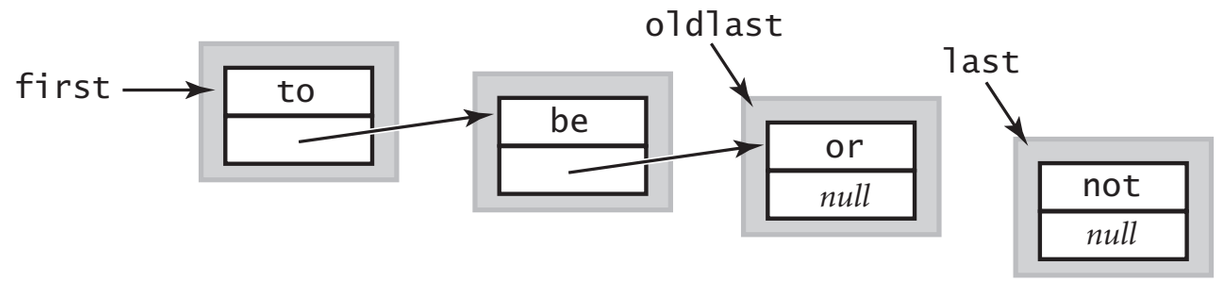
save a link to the last node

```
Node oldlast = last;
```



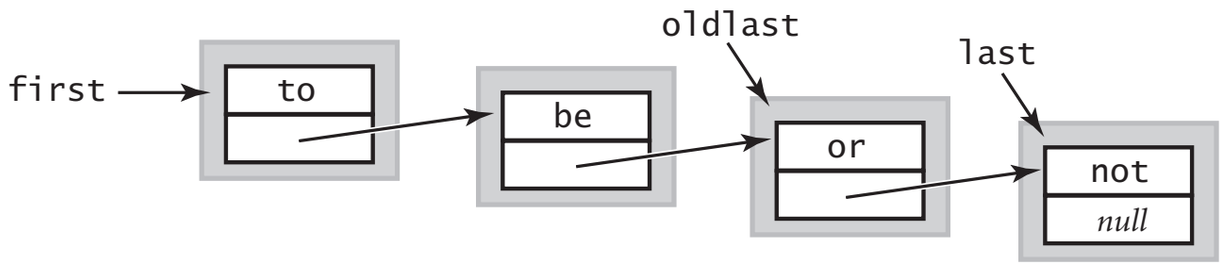

create a new node for the end

```
last = new Node();
last.item = "not";
```



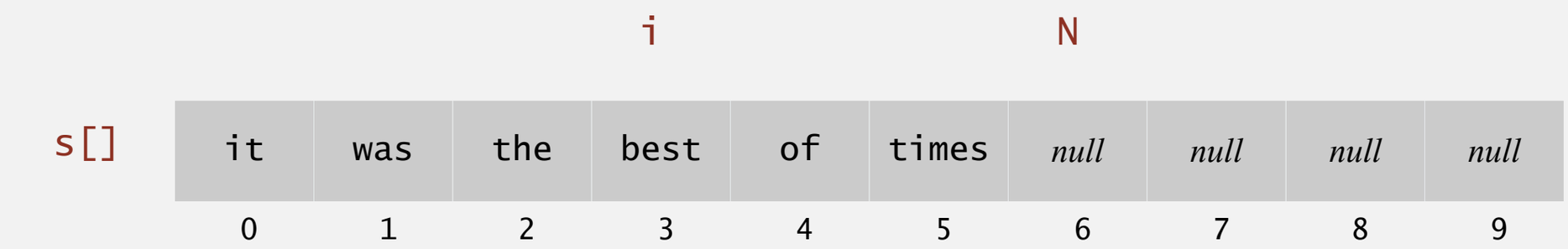

link the new node to the end of the list

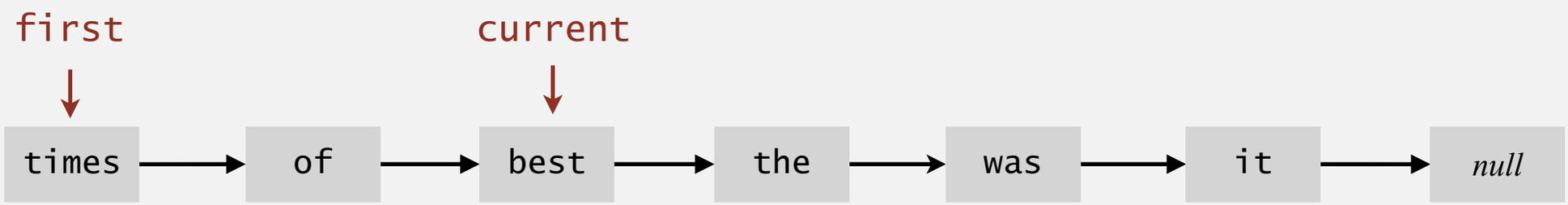
```
oldlast.next = last;
```



# Iteration

Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.




Java solution. Make stack implement the `java.lang.Iterable` interface.

# Java collections library

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

| `public interface List<Item> implements Iterable<Item>` | | |
|---|---|---|
| | `List()` | *create an empty list* |
| `boolean` | `isEmpty()` | *is the list empty?* |
| `int` | `size()` | *number of items* |
| `void` | `add(Item item)` | *append item to the end* |
| `Item` | `get(int index)` | *return item at given index* |
| `Item` | `remove(int index)` | *return and delete item at given index* |
| `boolean` | `contains(Item item)` | *does the list contain the given item?* |
| `Iterator<Item>` | `iterator()` | *iterator over all items in the list* |
| | `...` | |

Implementations. `java.util.ArrayList` uses resizing array;
`java.util.LinkedList` uses linked list.

caveat: only some operations are efficient