Terms:

Stacks
LinkedStack pops & push
Queues

Stacks and Queues

```
Objectives:
– Learn the concepts of stacks and queues.
– Define Stack and Queue ADTs.
– Examine implementation possibilities for Stacks and Queues.
– Examine uses for Stacks and Queues.
```

Lewis&Chase:
(ADTs) 2.12
(LinearNode) p 128-129
(Stacks) 6.1, 6.4, 6.5
(Queues) 7.1, 7.3

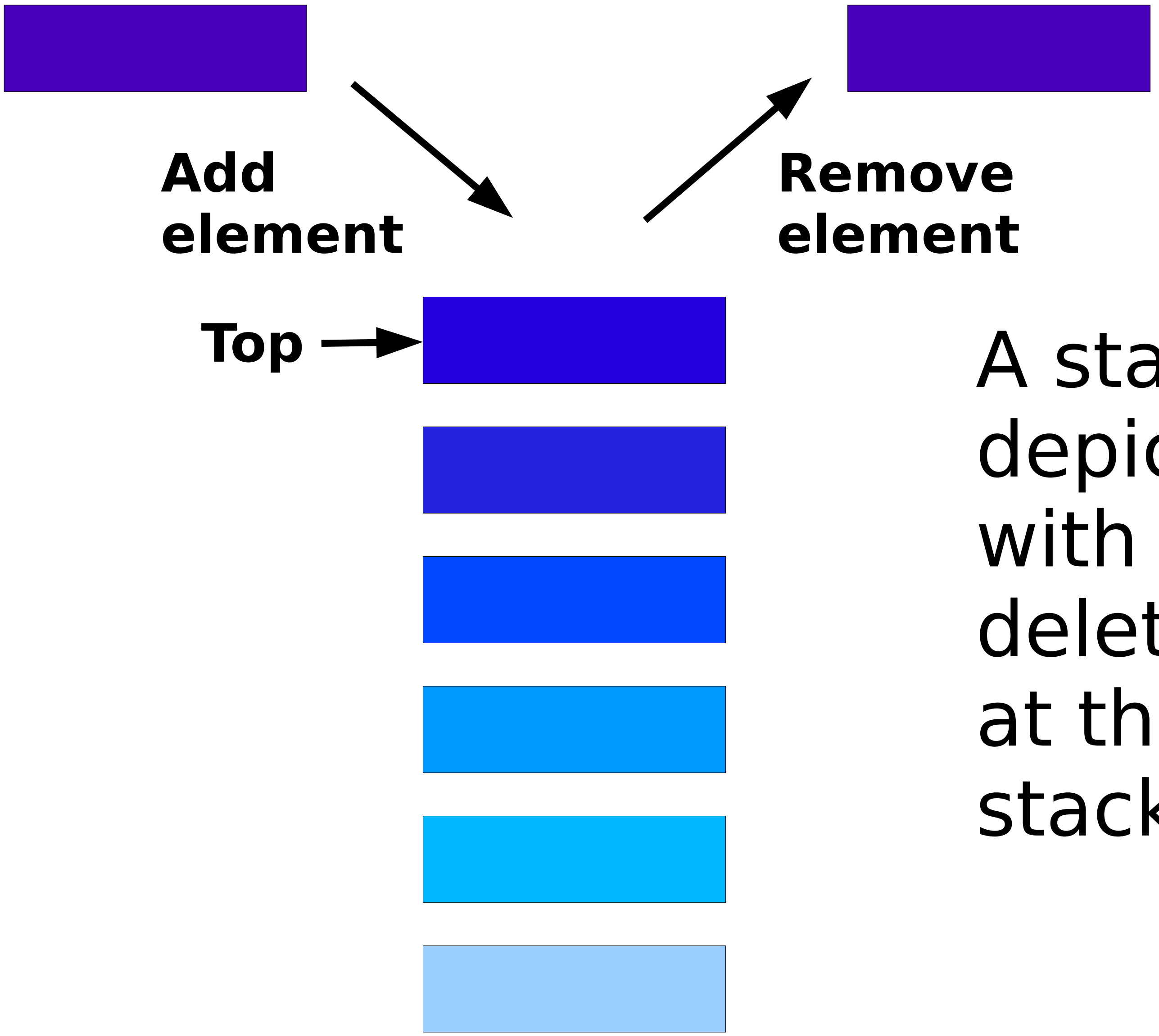
Stacks

We encounter stacks every day.
In loose terms, a stack is a pile of objects.

Stacks

- In computer terms, a **stack** is a linear collection of elements of the same type.
- Elements are added and removed from one end.
- The last element to be put on the stack is the first element to be removed.
- Only one element can be added or removed at a time.
- A stack is ***LIFO*** – **L**ast **I**n, **F**irst **O**ut.

Conceptual View of a Stack



A stack is usually depicted vertically, with additions and deletions occurring at the *top* of the stack.

Stack<T> - Data

- Java defines a `Stack<T>` class in package `java.util`
- A stack's **data** is a collection of objects (all of the same type `T`) in reverse chronological order.
  - The "oldest" item is on the bottom
  - The "newest" item is on the top.

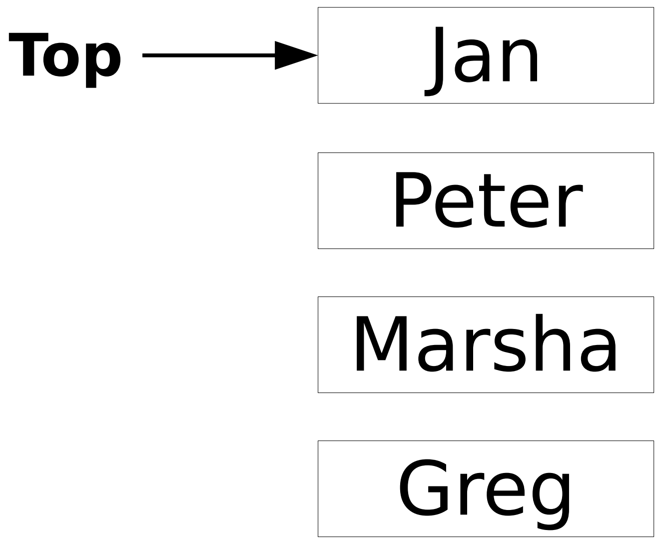
Stack<T> - Operations

- The idea of a stack is that all operations are performed on the top element of a stack, for example:

Stack Demo

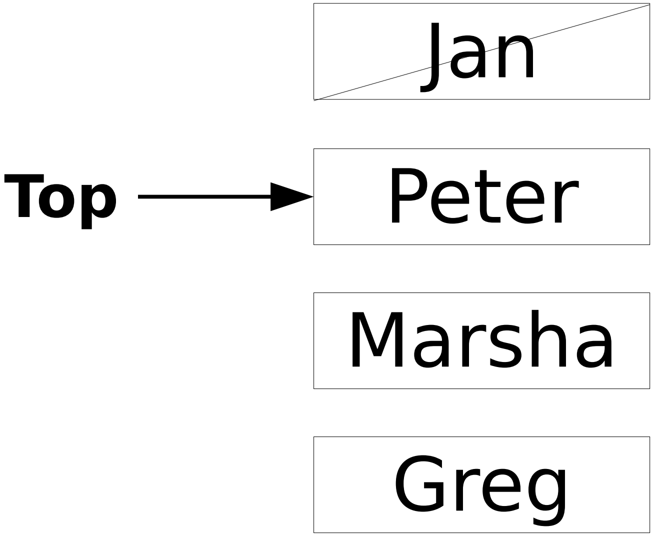
```
Stack<String> myStack = new Stack<String>();
String name = null;
myStack.push(“Greg”);
myStack.push(“Marsha”);
myStack.push(“Peter”);
myStack.push(“Jan”);
name = myStack.pop();
name = myStack.pop();
name = myStack.pop();
myStack.push(“Bobby”);
myStack.push(“Cindy”);
name = myStack.peek();
```

push 加
pop 减



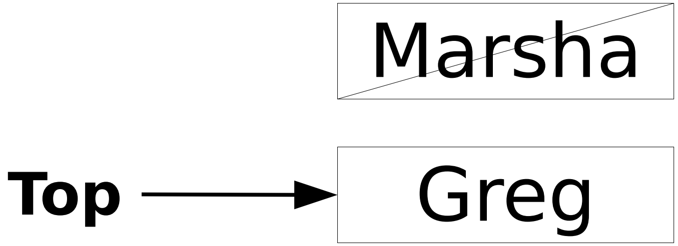
Stack Demo

```
Stack<String> myStack = new Stack<String>();
String name = null;
myStack.push(“Greg”);
myStack.push(“Marsha”);
myStack.push(“Peter”);
myStack.push(“Jan”);
name = myStack.pop();   // "Jan"
name = myStack.pop();
name = myStack.pop();
myStack.push(“Bobby”);
myStack.push(“Cindy”);
name = myStack.peek();
```



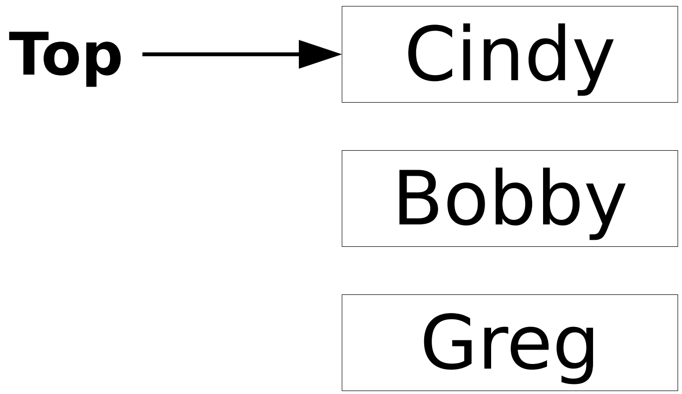
Stack Demo

```
Stack<String> myStack = new Stack<String>();
String name = null;
myStack.push(“Greg”);
myStack.push(“Marsha”);
myStack.push(“Peter”);
myStack.push(“Jan”);
name = myStack.pop();
name = myStack.pop();
name = myStack.pop();  //"Marsha"
myStack.push(“Bobby”);
myStack.push(“Cindy”);
name = myStack.peek();
```



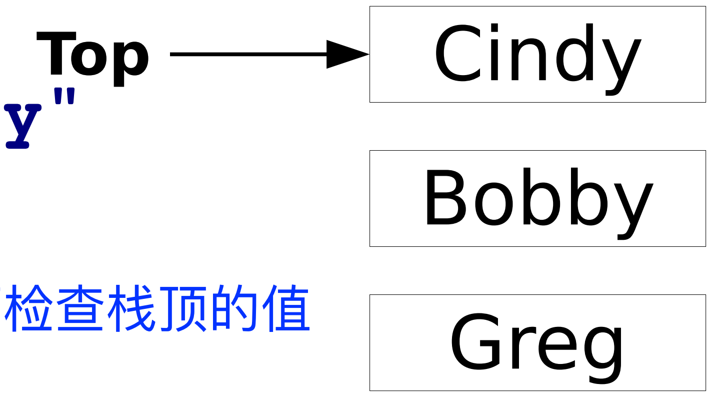
Stack Demo

```
Stack<String> myStack = new Stack<String>();
String name = null;
myStack.push("Greg");
myStack.push("Marsha");
myStack.push("Peter");
myStack.push("Jan");
name = myStack.pop();
name = myStack.pop();
name = myStack.pop();
myStack.push("Bobby");
myStack.push("Cindy");
name = myStack.peek();
```



Stack Demo

```
Stack<String> myStack = new Stack<String>();
String name = null;
myStack.push(“Greg”);
myStack.push(“Marsha”);
myStack.push(“Peter”);
myStack.push(“Jan”);
name = myStack.pop();
name = myStack.pop();
name = myStack.pop();
myStack.push(“Bobby”);
myStack.push(“Cindy”);
name = myStack.peek();   //"Cindy"
```



peek()：用于查看栈顶的元素但不移除它。
它返回栈顶的元素，通常用于在不修改栈内容的情况下检查栈顶的值

现实中的例子

Using Stacks

- One use of a stack that we are all familiar with is the *undo* function in most text editors.
- The operations that you perform (cut, paste, copy,...) are stored on a stack.
- When you choose “undo” from the menu, the last action that you performed gets undone and is popped from the stack.
- Sometimes, undo stacks have a limited capacity (special type of stack called drop-out stack)

Implementing Stacks

- We will create an interface (ADT) in a file called **`StackADT.java`**, which contains the method headings of our operations.
- Then we will look at one way of implementing our **`StackADT`** interface:
  - using links (**`LinkedStack.java`**)

  (Another way: using arrays)
- Find all examples under the “Examples” link on the course webpage

StackADT.java

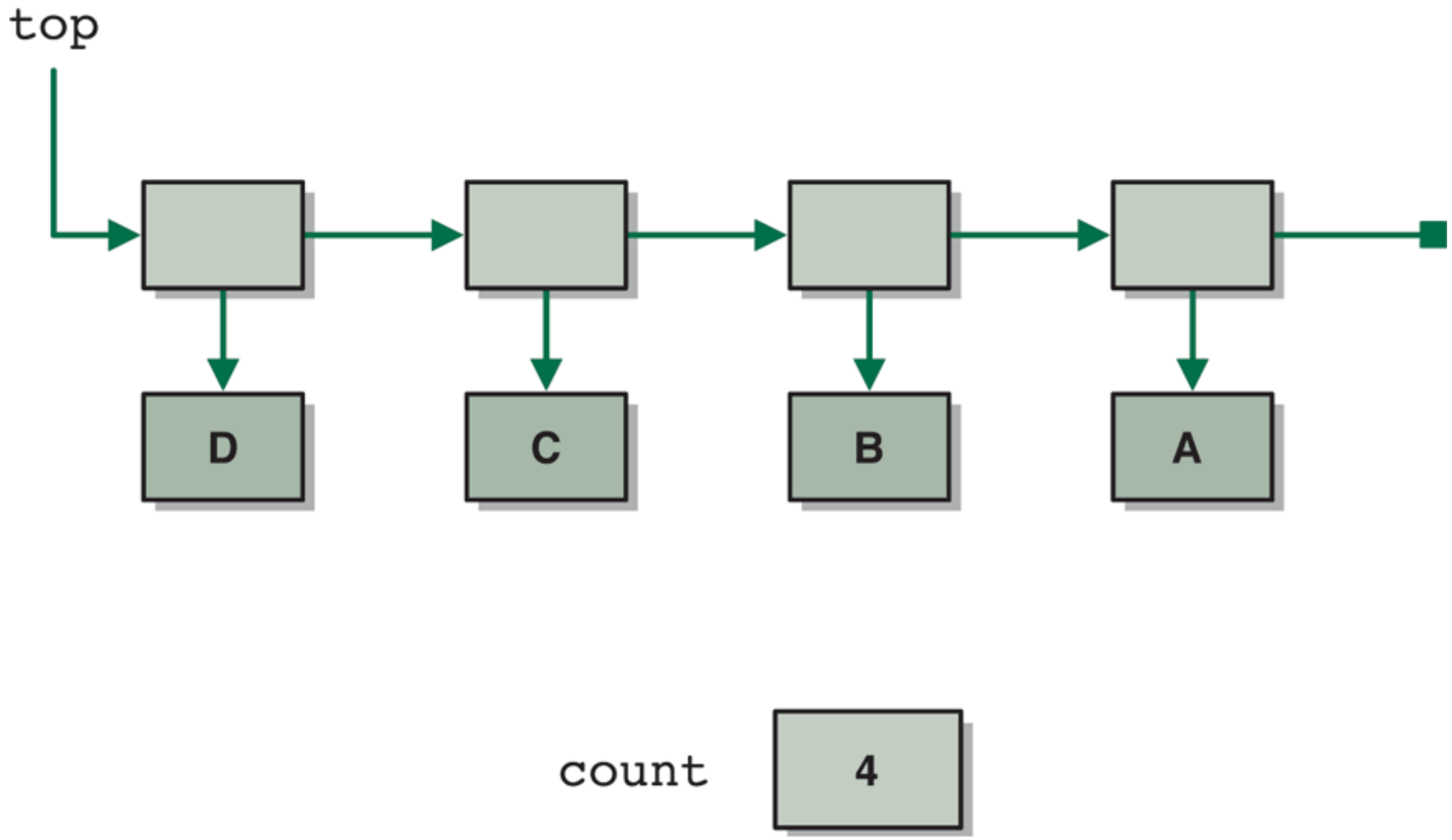
```
public interface StackADT<T> {

    // Adds one element to the top
    public void push(T element);

    // Remove and return top element
    public T pop();

    // Return without removing top element
    public T peek();

    // Return true if stack is empty
    public boolean isEmpty();

    // Return the number of elements
    public int size();

    // Return a string representation of the stack
    public String toString();
}
```

StackADT<T> - Linked Implementation

- We will use the **`LinearNode`** class (defined in L&C page 128-129) to represent a node on the stack.
- Notice that the **`LinearNode`** class is very similar to our **`ListNode`** class. It has get- and set-methods because it is not an inner class of the **`LinkedStack`** class.

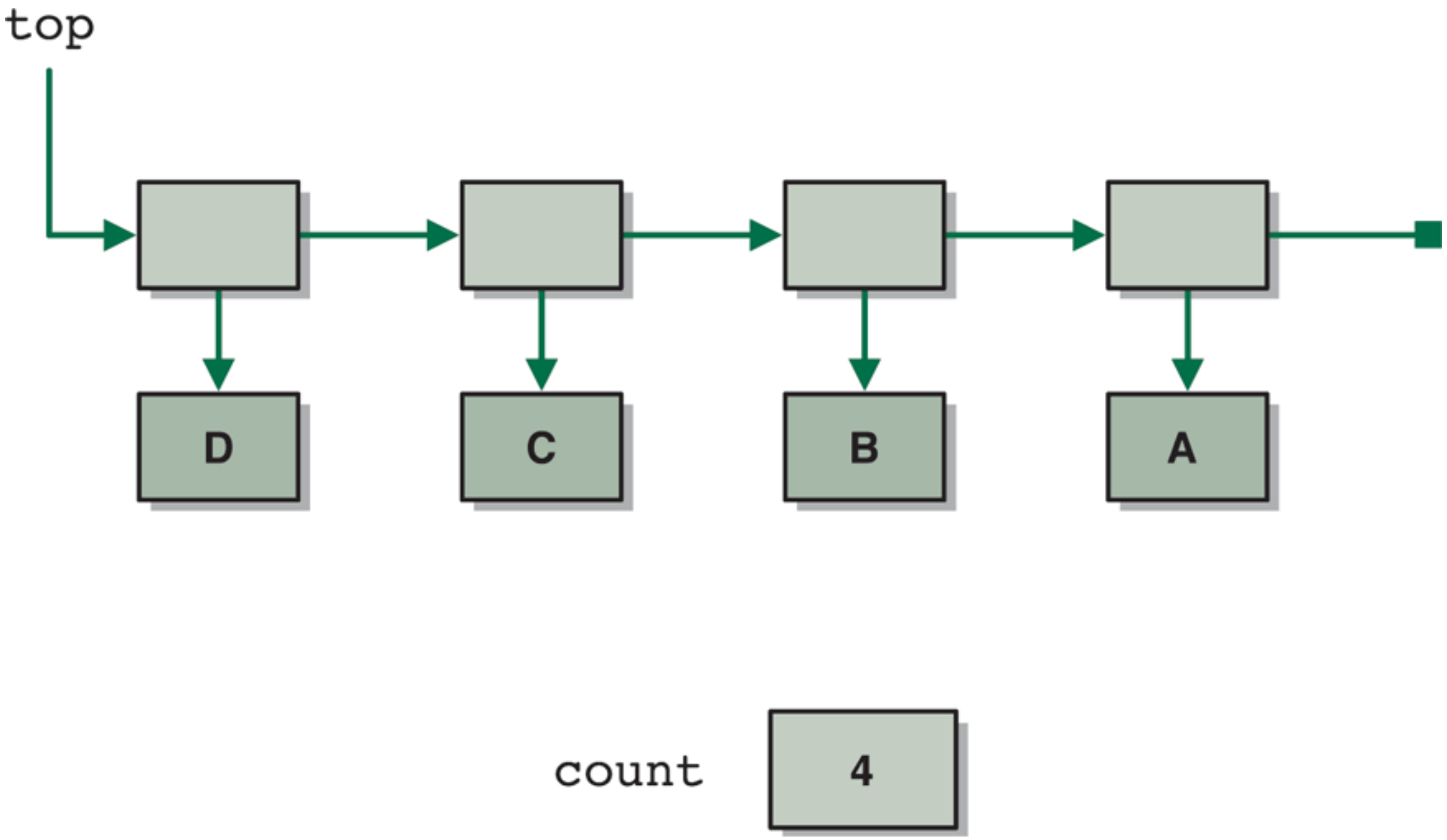
StackADT<T> - Linked Implementation

- A stack is represented as a linked list of nodes, with a reference to the top of the stack and a count of the number of nodes in the stack.



StackADT<T> - Linked Implementation

- Notice that we put the top of our stack at the front of the list.



LinkedStack<T>

```
public class LinkedStack<T> implements StackADT<T> {
    private LinearNode<T> top;
    private int count;

    <Constructors>

    <methods required by the StackADT interface>

}
```

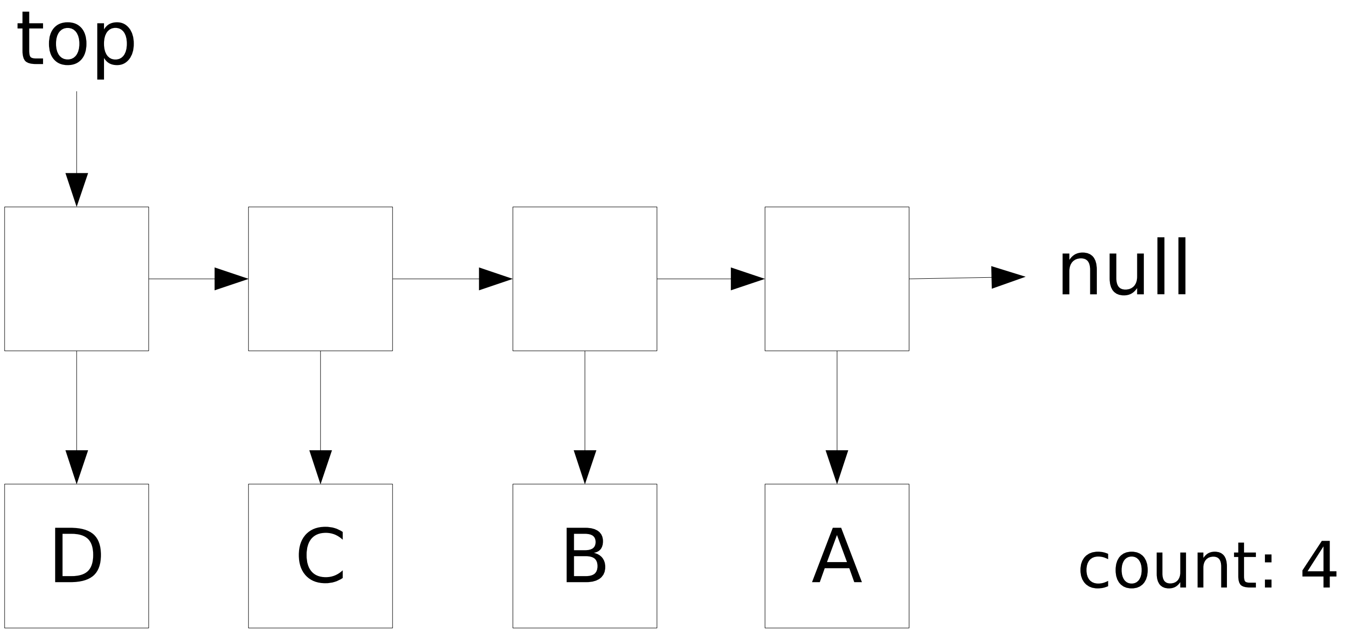
LinkedStack - Push

- The push operation has 4 steps:
  - create a new node containing the data
  - set the new node's next reference to top
  - set top to the new node
  - increment the size of the stack

LinkedStack - Push

- Let's push element "E" onto this stack:

```
public class LinkedStack<T> implements StackADT<T> {
   private LinearNode<T> top;
   private int count;
```

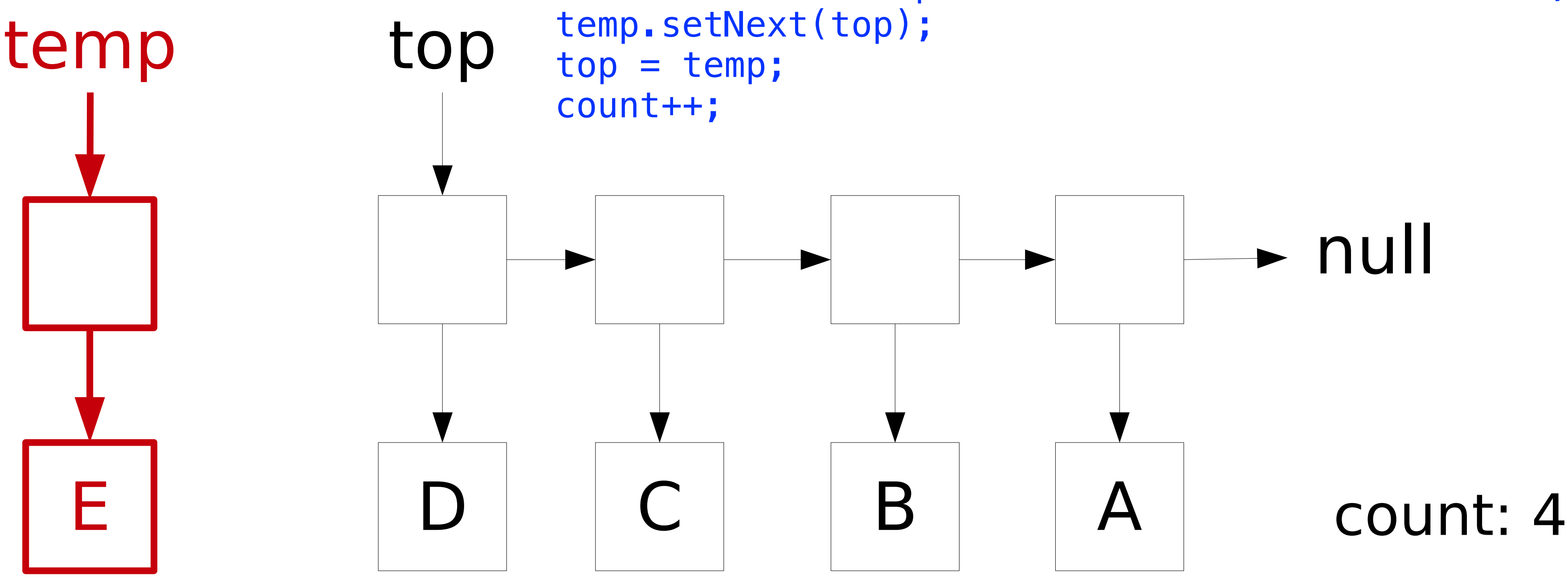


LinkedStack - Push

- Step 1 – create a new node with **`element`** as data:

```
LinearNode<T> temp = new LinearNode<T>(element);
```

```
public class LinkedStack<T> implements StackADT<T> {
   private LinearNode<T> top;
   private int count;}
public void push(T element) {
   LinearNode<T> temp = new LinearNode<T>(element);
   temp.setNext(top);
   top = temp;
   count++;
```

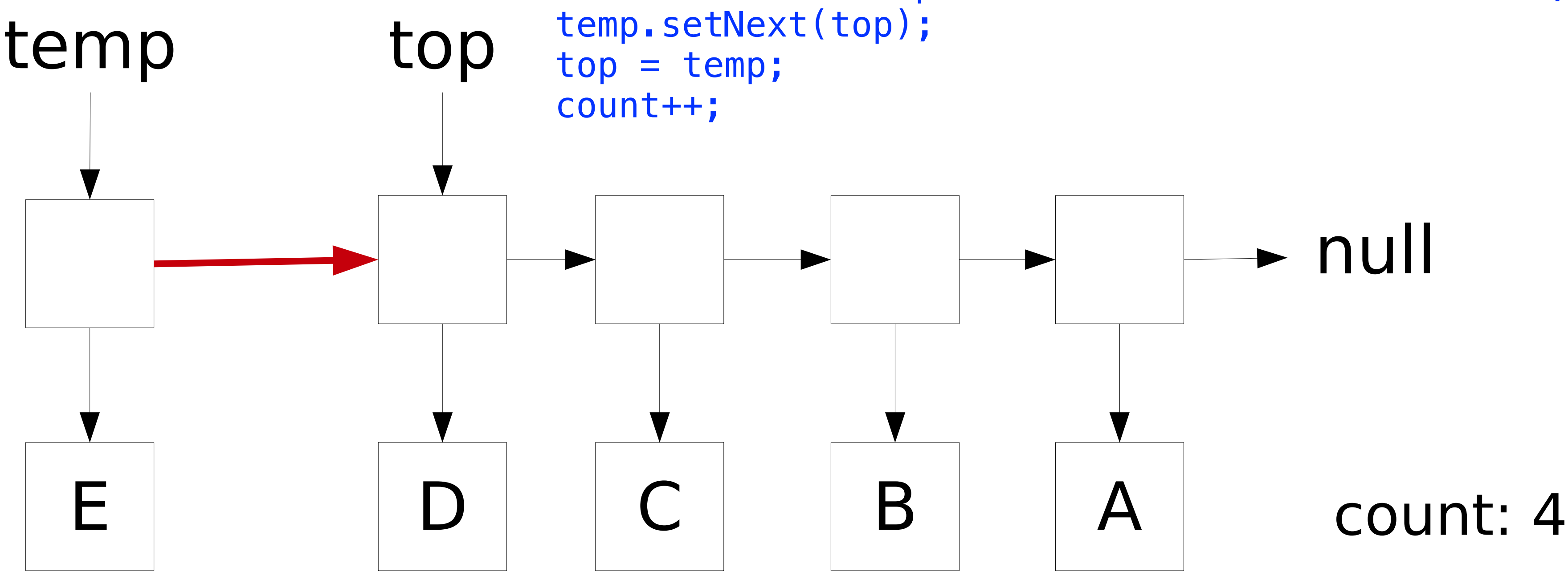


LinkedStack - Push

- Step 2 – set the new node's next reference to **top**:

`temp.setNext(top);`

```
public class LinkedStack<T> implements StackADT<T> {
   private LinearNode<T> top;
   private int count;}

public void push(T element) {
   LinearNode<T> temp = new LinearNode<T>(element);
   temp.setNext(top);
   top = temp;
   count++;
```

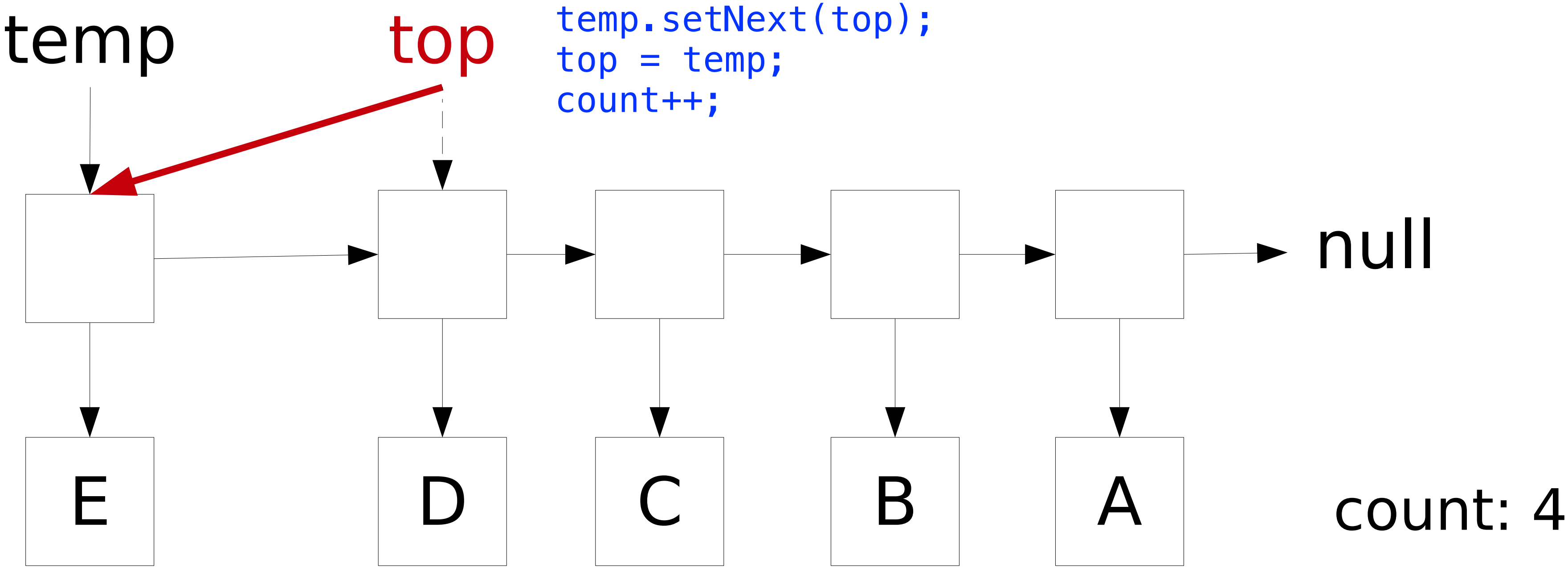


LinkedStack - Push

- Step 3 – set **`top`** to the new node:

`top = temp;`

```
public class LinkedStack<T> implements StackADT<T> {
   private LinearNode<T> top;
   private int count;}

public void push(T element) {
   LinearNode<T> temp = new LinearNode<T>(element);
   temp.setNext(top);
   top = temp;
   count++;
```

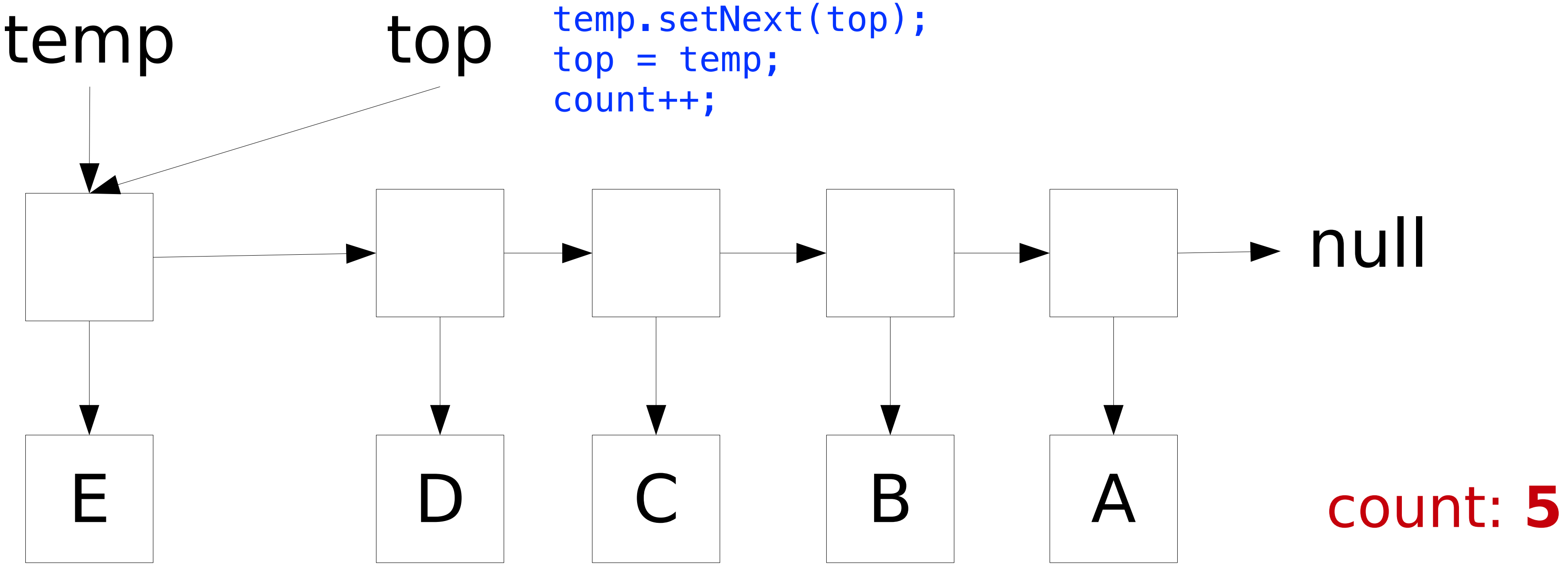


LinkedStack - Push

- Step 4 – increment the size of the stack:

`count++;`

```
public class LinkedStack<T> implements StackADT<T> {
   private LinearNode<T> top;
   private int count;}

public void push(T element) {
   LinearNode<T> temp = new LinearNode<T>(element);
   temp.setNext(top);
   top = temp;
   count++;
```



LinkedStack - Push

```
public class LinkedStack<T> implements StackADT<T> {
  <Instance variables and Constructors>

  public void push(T element) {
     LinearNode<T> temp = new LinearNode<T>(element);

     temp.setNext(top);
     top = temp;
     count++;
  }
}
```

LinkedStack - Pop

- The pop operation has 5 steps:
  - throw an exception if the stack is empty
  - get the data at the top node
  - update top
  - decrement the size of the stack
  - return the data portion of the popped node

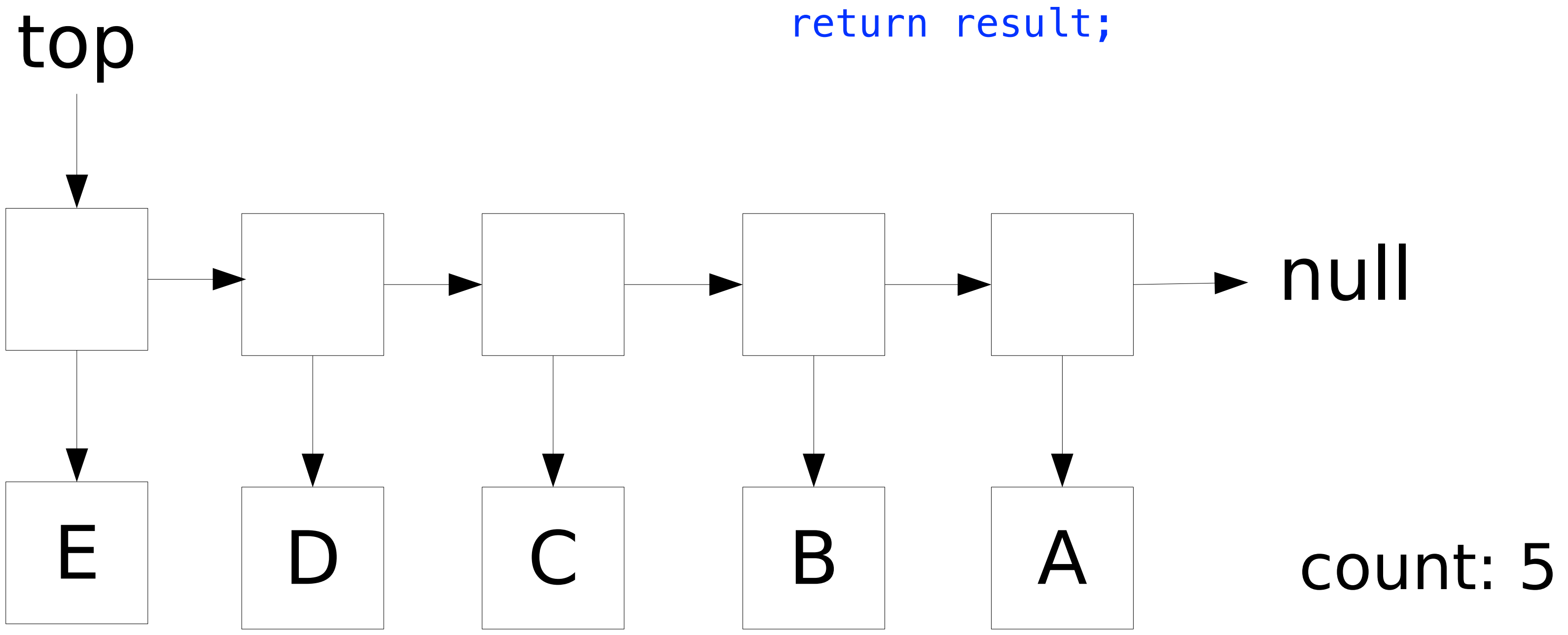
LinkedStack - Pop

```
public class LinkedStack<T> implements StackADT<T> {
   <Instance variables and Constructors>
   <Implementation of push>
   public T pop() {
      if (isEmpty())
           throw new EmptyStackException();
      T result = top.getElement();
      top = top.getNext();
      count--;
      return result;
   }

}
```

LinkedStack - Pop

- Let's pop an element from this stack:

```
public T pop() {
   if (isEmpty())
      throw new EmptyStackException();
   T result = top.getElement();
   top = top.getNext();
   count--;
   return result;
```

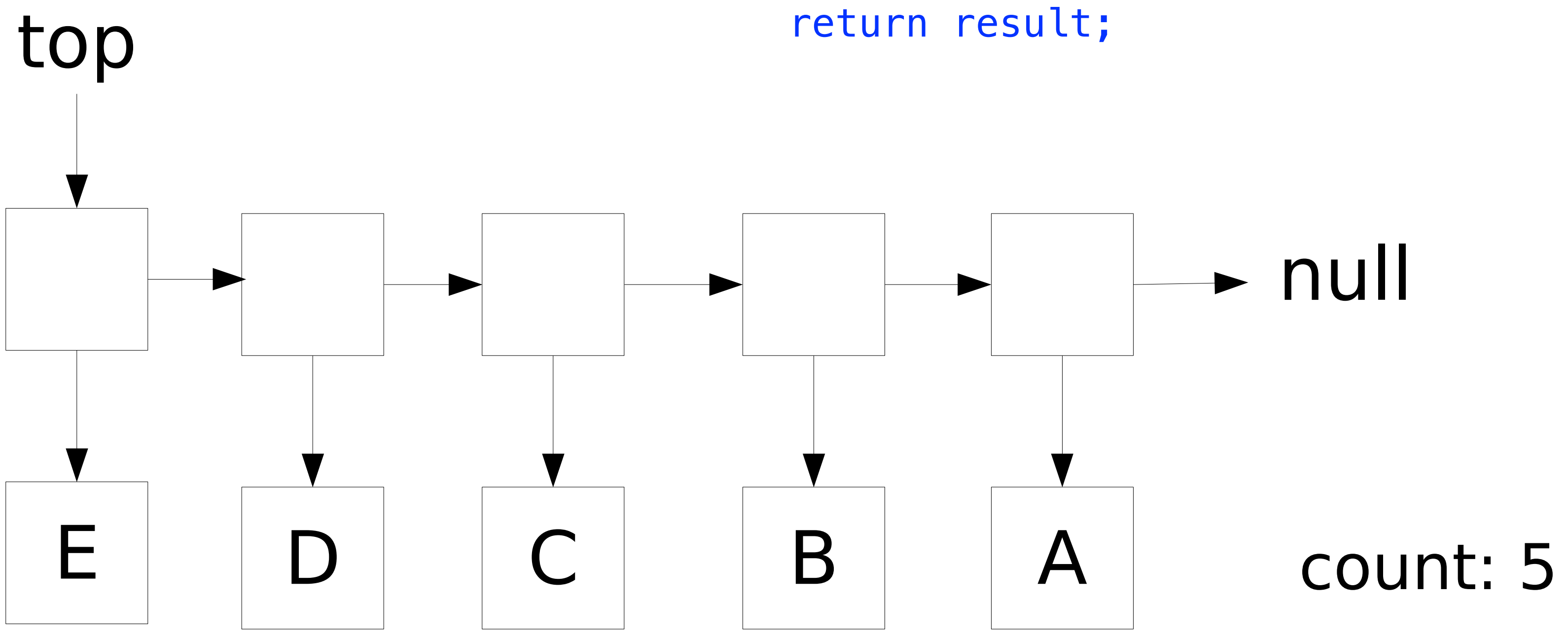


LinkedStack - Pop

- Step 1 – throw an exception if the stack is empty:

```
public T pop() {
   if (isEmpty())
      throw new EmptyStackException();
   T result = top.getElement();
   top = top.getNext();
   count--;
   return result;
```

```
if (isEmpty())
      throw new EmptyStackException();
```

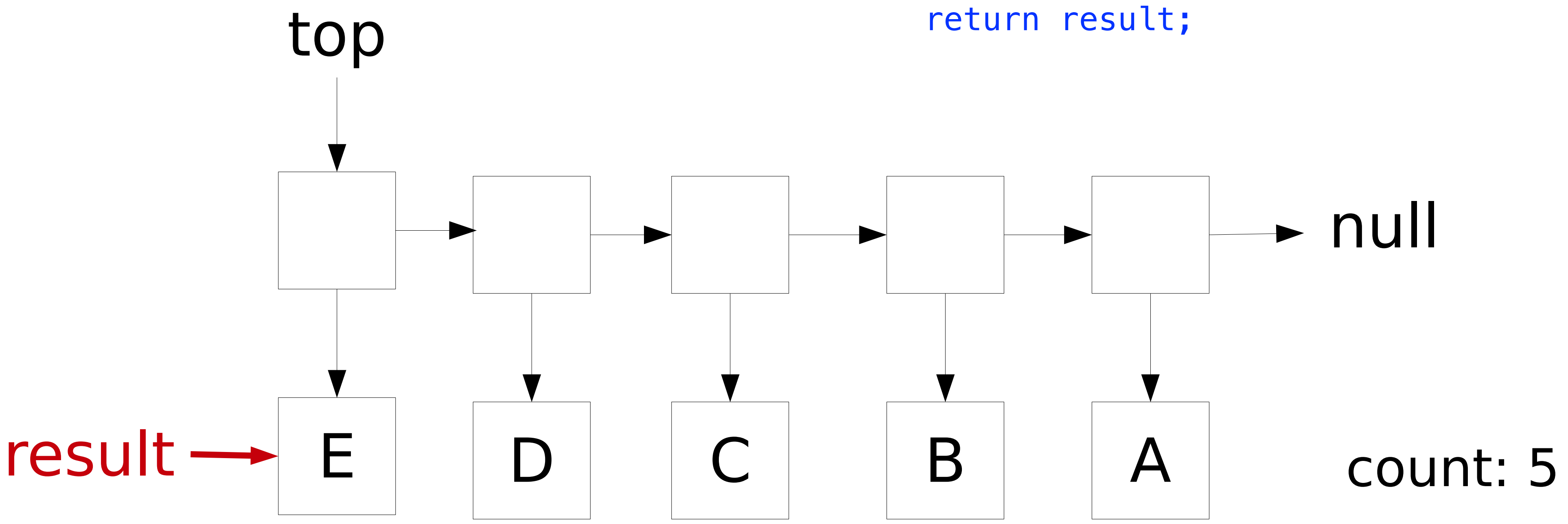


LinkedStack - Pop

- Step 2 – get the data at the top node:

```
T result = top.getElement();
```

```
public T pop() {
   if (isEmpty())
      throw new EmptyStackException();
   T result = top.getElement();
   top = top.getNext();
   count--;
   return result;
```

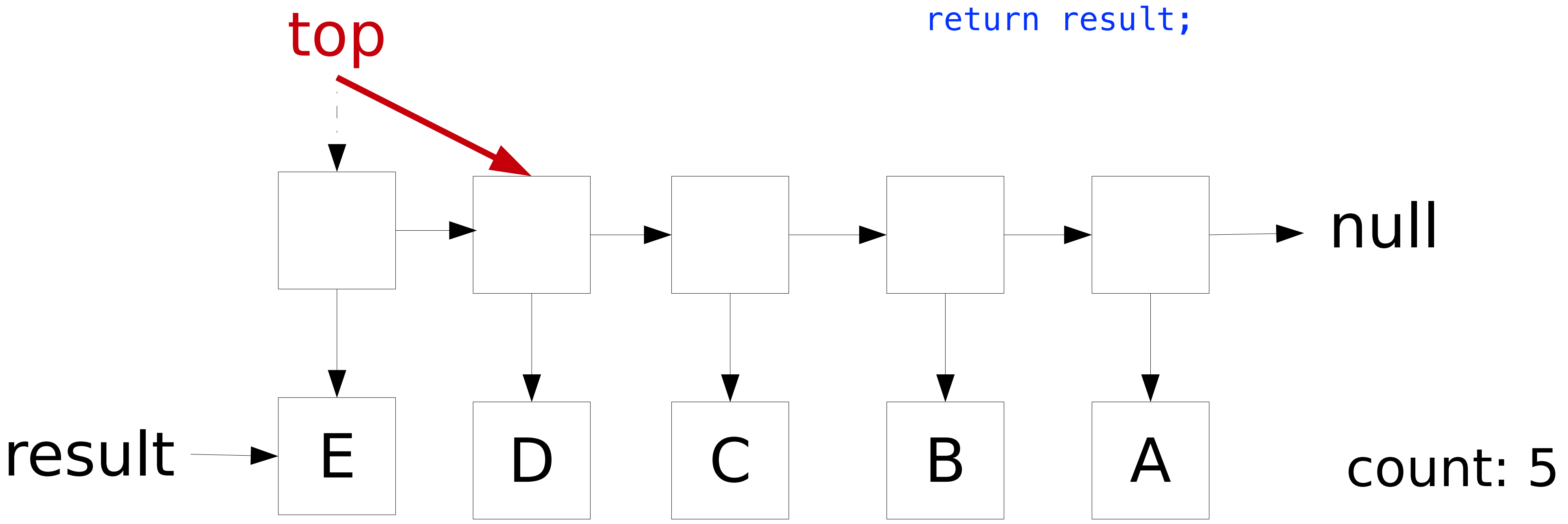


LinkedStack - Pop

- Step 3 – update **`top`**:

`top = top.getNext();`

```
public T pop() {
   if (isEmpty())
      throw new EmptyStackException();
   T result = top.getElement();
   top = top.getNext();
   count--;
   return result;
```

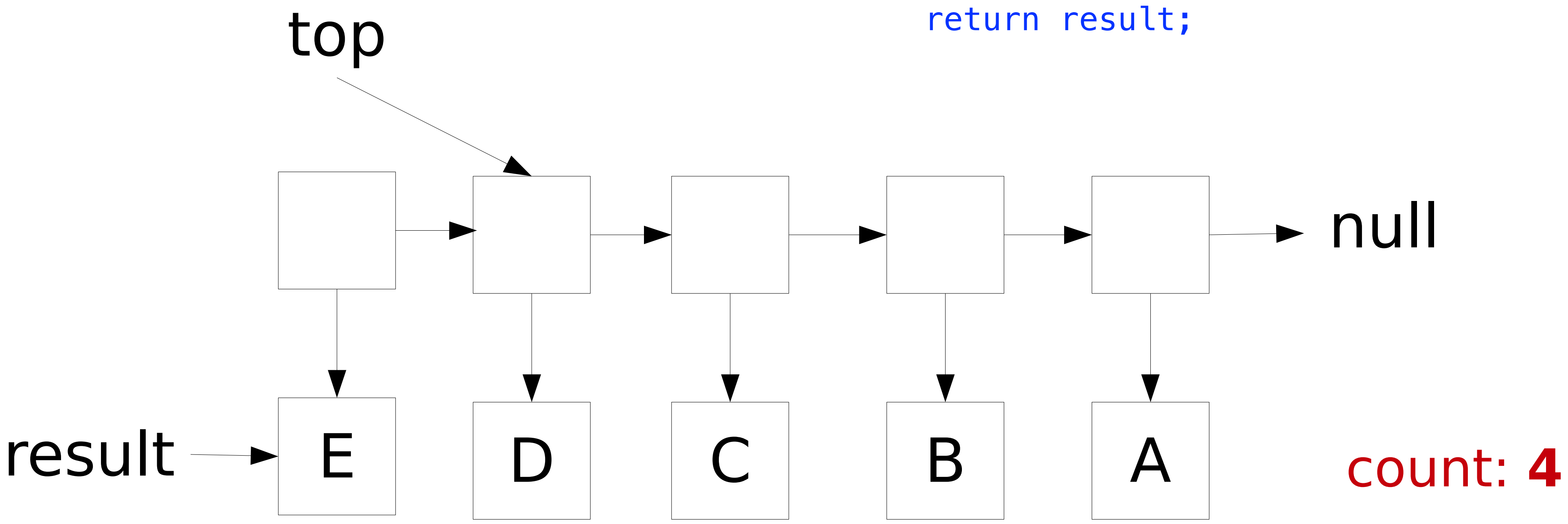


LinkedStack - Pop

- Step 4 – decrement the size of the stack:

```
count--;
```

```
public T pop() {
   if (isEmpty())
      throw new EmptyStackException();
   T result = top.getElement();
   top = top.getNext();
   count--;
   return result;
```



LinkedStack - Pop

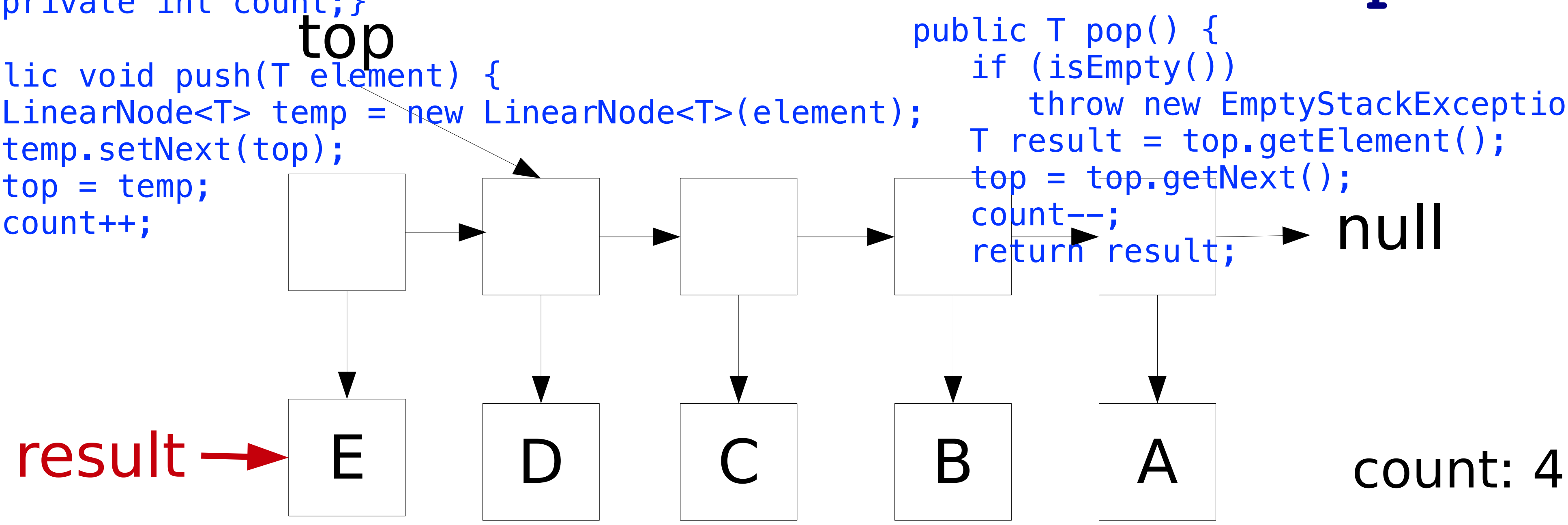
- Step 5 – return the data popped:

```
return result;
```

```
public class LinkedStack<T> implements StackADT<T>
{
   private LinearNode<T> top;
   private int count;}

public void push(T element) {
   LinearNode<T> temp = new LinearNode<T>(element);
   temp.setNext(top);
   top = temp;
   count++;
```

Notice that **pop** is the inverse of **push**

```
public T pop() {
   if (isEmpty())
      throw new EmptyStackException();
   T result = top.getElement();
   top = top.getNext();
   count--;
   return result;
```



Queues

- In a **queue,** elements are added at one end and removed from the other.
- Any waiting line is a queue:
  - checkout line at a store
  - cars at a stoplight

Queues

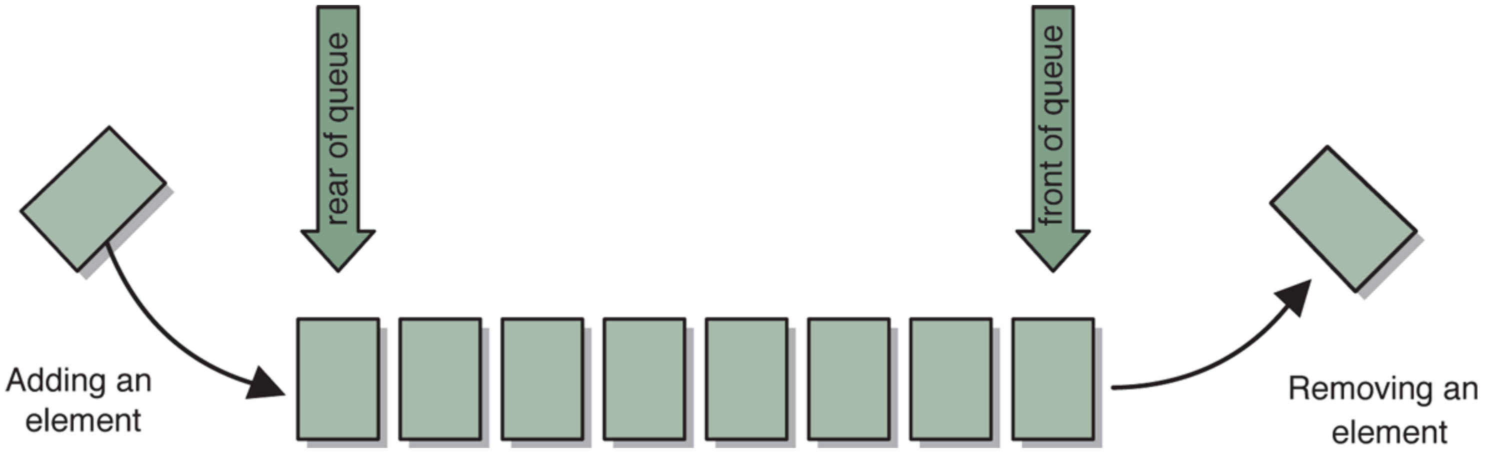
`Stack – FILO: First in, Last Out`

- A queue is ***FIFO*** – **F**irst **I**n, **F**irst **O**ut.
- Like a stack, only one element can be added or removed at a time.
- Unlike a stack, which operates on only one end of the collection, a queue operates on both ends.

Conceptual View of a Queue

Stack : vertically

A queue is usually depicted **horizontally**, with **additions** occurring at the *rear* (or *tail*) and **deletions** occurring at the *front* (or *head*).



QueueADT<T> - Data

- A queue's **data** is a collection of objects (all of the same type `T`) in chronological order.

  Stack – in **reverse** chronological order.
  – The “oldest” item is on the bottom
  – The “newest” item is on the top.

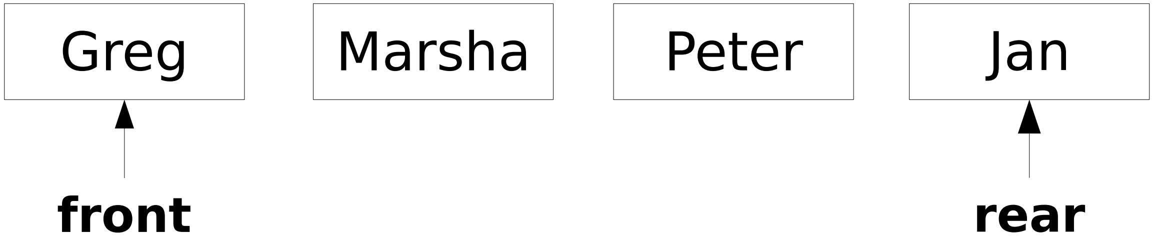
  - The “oldest” item is at the front
  - The “newest” item is at the back.
  - Items are processed on a first come, first served basis

QueueADT<T> - Operations

	Operation	Description
push	`enqueue`	Adds an element to the rear of the queue.
pop	`dequeue`	Removes an element from the front of the queue.
	`first`	Examines the element at the front of the queue.
	`isEmpty`	Determines if the queue is empty.
	`size`	Determines the number of elements on the queue.
	`toString`	Returns a string representation of the queue.

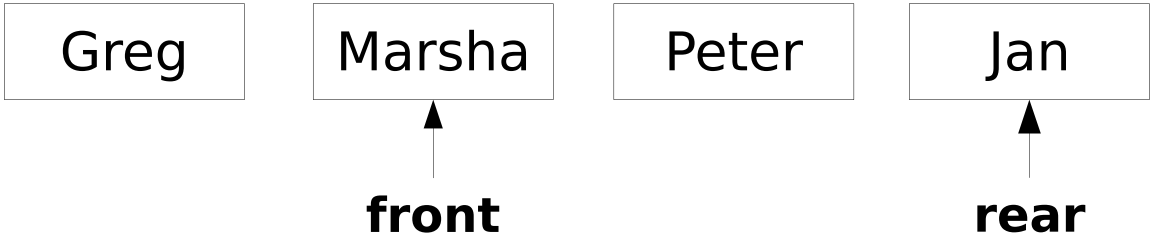
Queue Demo

```
Queue<String> myQueue = new Queue<String>();
String name = null;
myQueue.enqueue(“Greg”);
myQueue.enqueue(“Marsha”);
myQueue.enqueue(“Peter”);
myQueue.enqueue(“Jan”);
name = myQueue.dequeue();
name = myQueue.dequeue();
name = myQueue.dequeue();
myQueue.enqueue(“Bobby”);
myQueue.enqueue(“Cindy”);
name = myQueue.first();
```



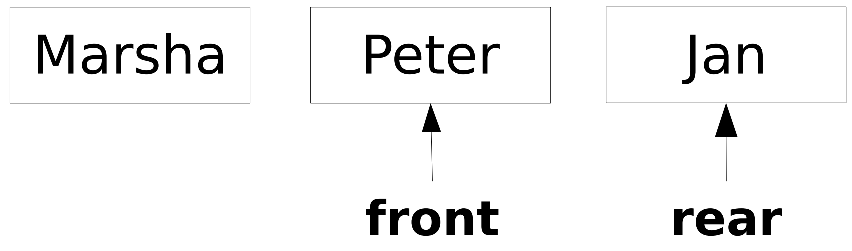
Queue Demo

```
Queue<String> myQueue = new Queue<String>();
String name = null;
myQueue.enqueue(“Greg”);
myQueue.enqueue(“Marsha”);
myQueue.enqueue(“Peter”);
myQueue.enqueue(“Jan”);
name = myQueue.dequeue();   // "Greg"
name = myQueue.dequeue();
name = myQueue.dequeue();
myQueue.enqueue(“Bobby”);
myQueue.enqueue(“Cindy”);
name = myQueue.first();
```



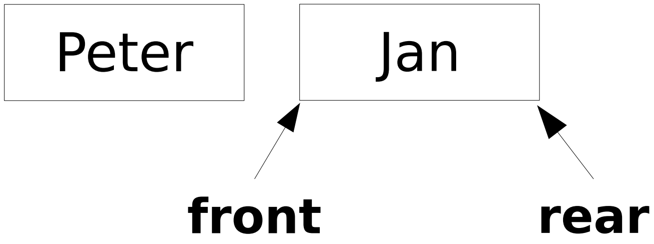
Queue Demo

```
Queue<String> myQueue = new Queue<String>();
String name = null;
myQueue.enqueue(“Greg”);
myQueue.enqueue(“Marsha”);
myQueue.enqueue(“Peter”);
myQueue.enqueue(“Jan”);
name = myQueue.dequeue();
name = myQueue.dequeue();   // "Marsha"
name = myQueue.dequeue();
myQueue.enqueue(“Bobby”);
myQueue.enqueue(“Cindy”);
name = myQueue.first();
```



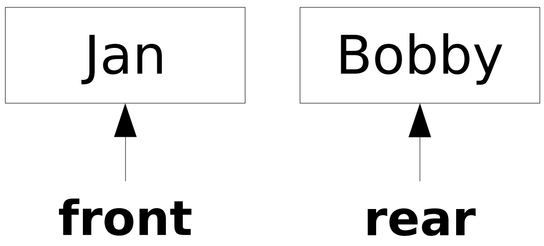
Queue Demo

```
Queue<String> myQueue = new Queue<String>();
String name = null;
myQueue.enqueue(“Greg”);
myQueue.enqueue(“Marsha”);
myQueue.enqueue(“Peter”);
myQueue.enqueue(“Jan”);
name = myQueue.dequeue();
name = myQueue.dequeue();
name = myQueue.dequeue();   // "Peter"
myQueue.enqueue(“Bobby”);
myQueue.enqueue(“Cindy”);
name = myQueue.first();
```



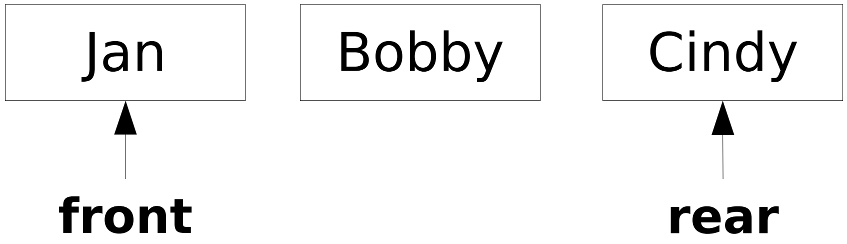
Queue Demo

```
Queue<String> myQueue = new Queue<String>();
String name = null;
myQueue.enqueue(“Greg”);
myQueue.enqueue(“Marsha”);
myQueue.enqueue(“Peter”);
myQueue.enqueue(“Jan”);
name = myQueue.dequeue();
name = myQueue.dequeue();
name = myQueue.dequeue();
myQueue.enqueue(“Bobby”);
myQueue.enqueue(“Cindy”);
name = myQueue.first();
```



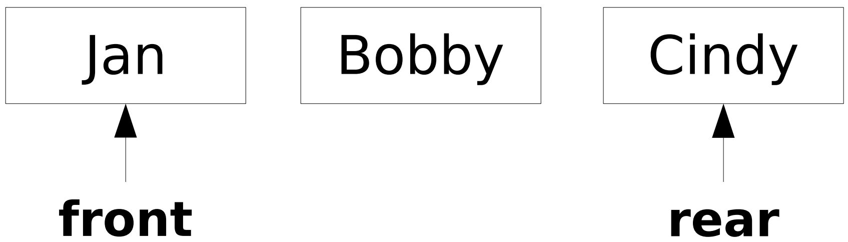
Queue Demo

```
Queue<String> myQueue = new Queue<String>();
String name = null;
myQueue.enqueue(“Greg”);
myQueue.enqueue(“Marsha”);
myQueue.enqueue(“Peter”);
myQueue.enqueue(“Jan”);
name = myQueue.dequeue();
name = myQueue.dequeue();
name = myQueue.dequeue();
myQueue.enqueue(“Bobby”);
myQueue.enqueue(“Cindy”);
name = myQueue.first();
```



Queue Demo

```
Queue<String> myQueue = new Queue<String>();
String name = null;
myQueue.enqueue(“Greg”);
myQueue.enqueue(“Marsha”);
myQueue.enqueue(“Peter”);
myQueue.enqueue(“Jan”);
name = myQueue.dequeue();
name = myQueue.dequeue();
name = myQueue.dequeue();
myQueue.enqueue(“Bobby”);
myQueue.enqueue(“Cindy”);
name = myQueue.first();    // "Jan"
```



Using Queues

- It is an everyday occurrence to wait in line – at a bakery or a bank, for example.
- Businesses are concerned with the time their customers must wait to be served.
- If the waiting time is too long, customers will be dissatisfied, but it is expensive to hire more employees to wait on customers.
- Computer simulation allows us to predict the effect of adding more cashiers.

QueueADT.java

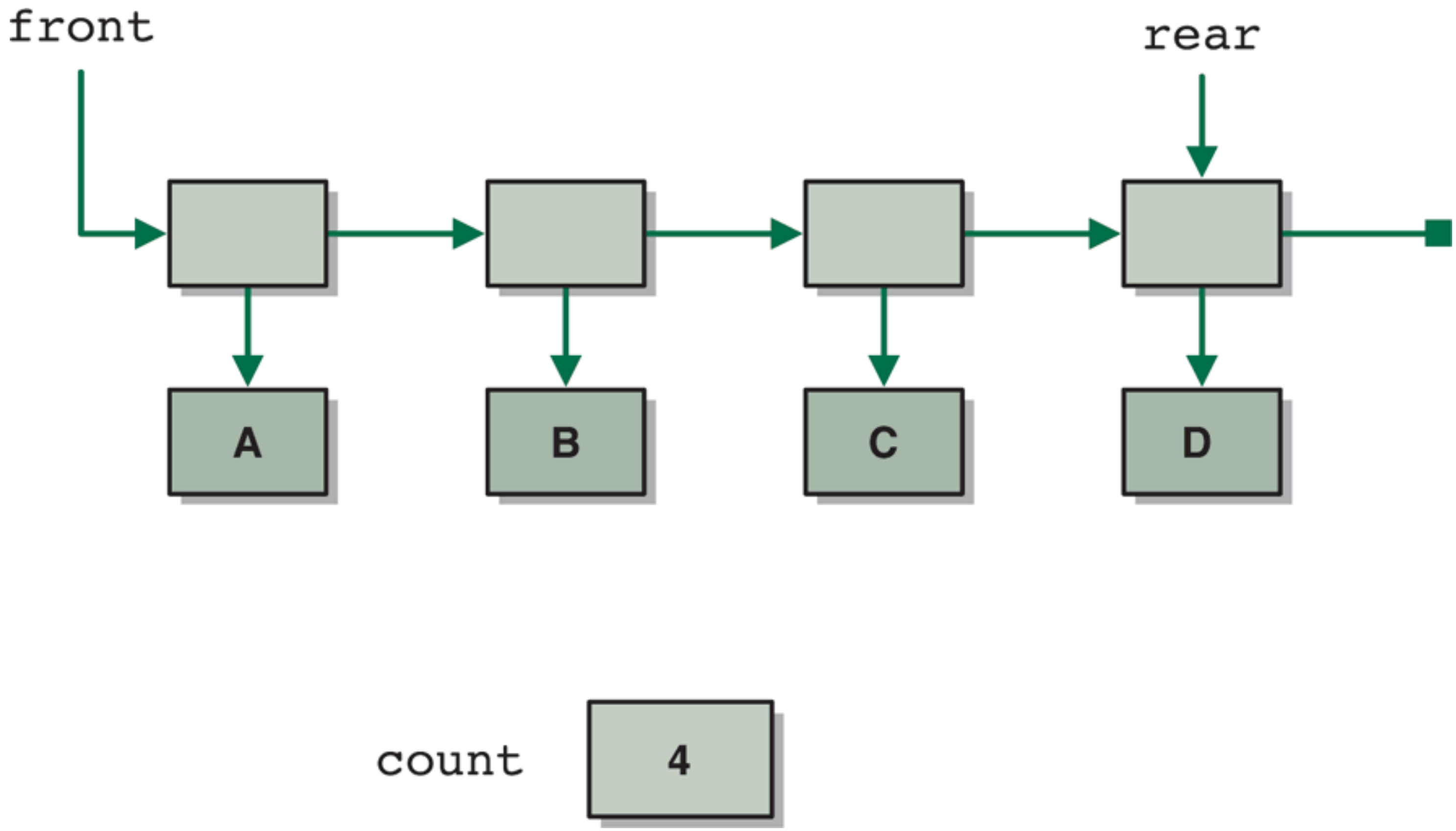
```
public interface QueueADT<T> {

    // Add a new entry to the back of the queue.
    public void enqueue(T element);

    // Remove and return the front element
    public T dequeue();

    // Return (don't remove) the front element
    public T first();

    // Return true if the queue is empty,false otherwise
    public boolean isEmpty();

    // Remove all entries from the queue.
    public int size();

    // Return a string representation of the queue
    public String toString();
}
```

QueueADT<T> - Linked Implementation

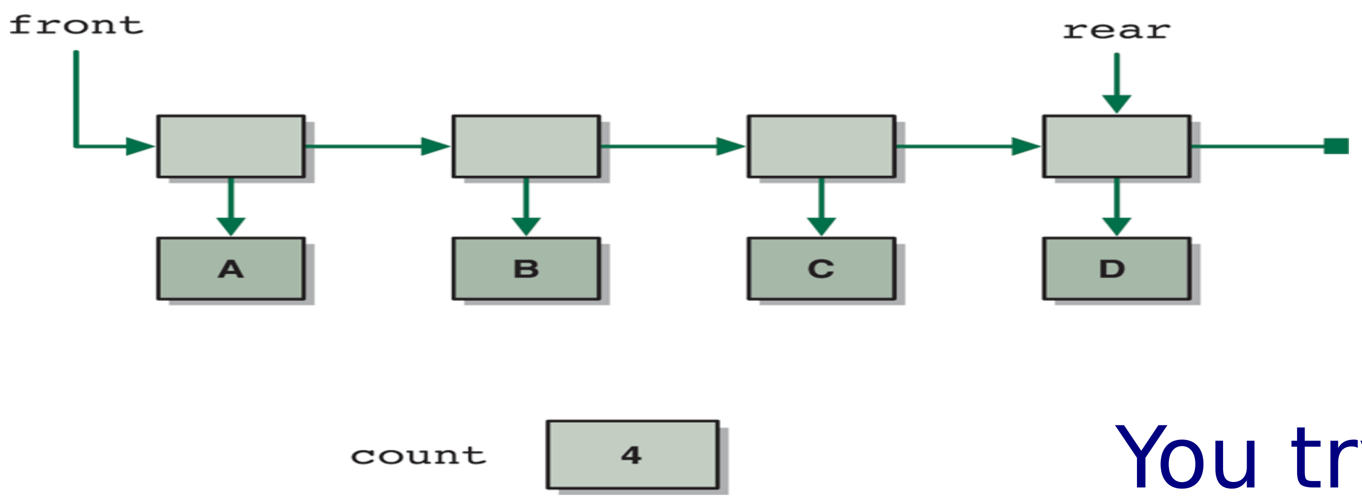
- Like stacks, queues can be implemented using an array or a linked list.
- A linked version can use the `LinearNode` class.
- In addition to keeping a reference to the front of the list, we will also keep a second reference to the end.
- An integer `count` will keep track of the number of elements in the queue.

QueueADT<T> - Linked Implementation

- A queue is represented as a linked list of nodes, with references to the front and rear, and a count of the number of nodes in the queue.

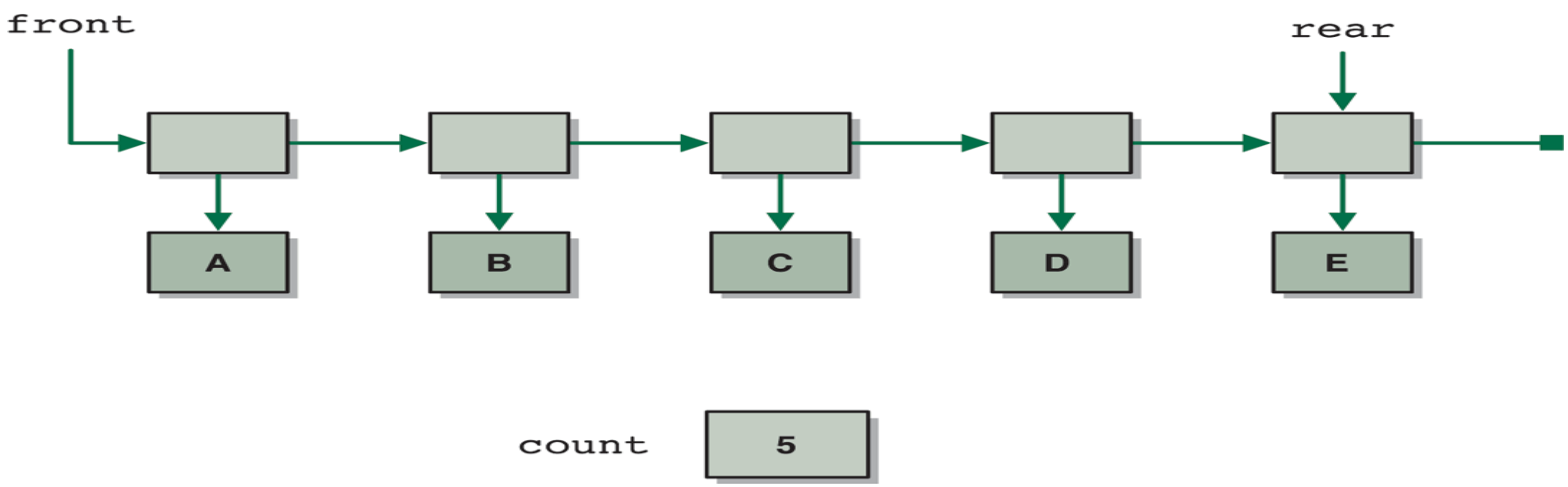


LinkedQueue - Enqueue



You try: write the enqueue method

After enqueing "E":

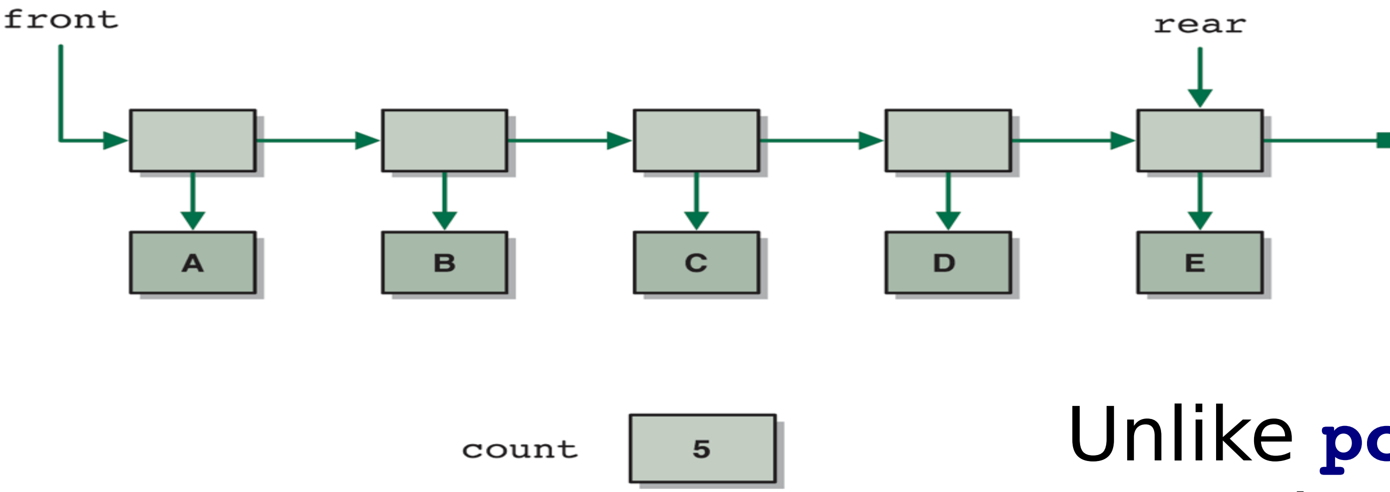


LinkedQueue - Enqueue

```
public class LinkedQueue<T> implements QueueADT<T> {

   <Instance variables and Constructors>

   public void enqueue(T element) {
      LinearNode<T> node = new LinearNode<T>(element);

      if (isEmpty())
          front = node;
      else
          rear.setNext(node);
      rear = node;
      count++;
   }
}
```

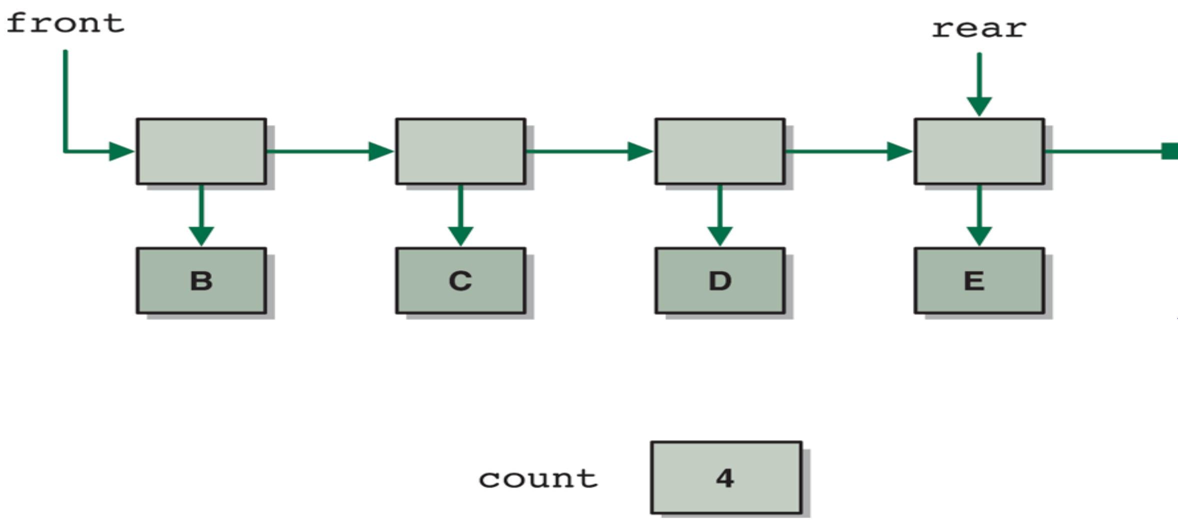
```
public class LinkedStack<T> implements StackADT<T> {
   private LinearNode<T> top;
   private int count;}

public void push(T element) {
   LinearNode<T> temp = new LinearNode<T>(element);
   temp.setNext(top);
   top = temp;
   count++;
```

LinkedQueue - Dequeue



Unlike **pop** and **push** on a stack, **dequeue** is not the inverse of **enqueue**

After dequeing:



You try: write the dequeue method

LinkedQueue - Dequeue

```
public class LinkedQueue<T> implements QueueADT<T> {
   <Instance variables and Constructors>
   <implementation of enqueue>

   public T dequeue() throws EmptyCollectionException {
      if (isEmpty())
          throw new EmptyCollectionException("queue");

      T result = front.getElement();
      front = front.getNext();
      count--;

      if (isEmpty())
          rear = null;
      return result;
   }
}
```

```
public T pop() {
   if (isEmpty())
      throw new EmptyStackException();
   T result = top.getElement();
   top = top.getNext();
   count--;
   return result;
```

FULL CODES:

```java
public class LinkedQueue<T> implements QueueADT<T> {
    private int count;
    private Node<T> front, rear;

    // Constructor and other methods

    public void enqueue(T element) {
        Node<T> node = new Node<>(element);
        if (isEmpty()) {
            front = node;
        } else {
            rear.setNext(node);
        }
        rear = node;
        count++;
    }

    public T dequeue() throws EmptyCollectionException {
        if (isEmpty())
            throw new EmptyCollectionException("queue");

        T result = front.getElement();
        front = front.getNext();
        count--;

        // 这里明确地检查并设置 rear 为 null
        if (isEmpty())
            rear = null;

        return result;
    }

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

    // Other methods like size, front, etc.

    private static class Node<T> {
        private T element;
        private Node<T> next;

        public Node(T element) {
            this.element = element;
            this.next = null;
        }

        public T getElement() {
            return element;
        }

        public Node<T> getNext() {
            return next;
        }

        public void setNext(Node<T> next) {
            this.next = next;
        }
    }

    public static class EmptyCollectionException
                         extends RuntimeException {
        public EmptyCollectionException(String collection) {
            super("The " + collection + " is empty.");
        }
    }
}
```

Example 1: TicketCounter

- Let's examine a program that simulates customers waiting in line at a ticket counter.
- The goal is to find out how many cashiers are needed to keep the average wait time below 7 minutes.
- See **`TicketCounter.java`** and **`Customer.java`** under “L&C Src” on the course webpage.

TicketCounter

- We will determine the average waiting time if there is 1 cashier, 2 cashiers, ... 10 cashiers and print the average waiting time for each number of cashiers.
- Assume that:
  - customers arrive on average every 15 seconds
  - buying a ticket takes 2 minutes (120 seconds) once a customer reaches the cashier

TicketCounter

- We define a class **`Customer`** to represent one person in line.
- Each customer has an **`arrivalTime`** and a **`departureTime`**, in seconds.
- Each customer is constructed with an arrival time 15 seconds after the previous customer constructed.
- A customer's total waiting time (calculated by method **`totalTime`**) is the difference between their **`arrivalTime`** and **`departureTime`**.

TicketCounter

- The result of the simulation shows that after 7 cashiers, the customers don't have to wait at all.
- To keep the average wait time below 7 minutes (420 seconds), we need 6 cashiers.

Number of Cashiers:	1	2	3	4	5	6	7	8	9	10
Average Time (sec):	5317	2325	1332	840	547	355	219	120	120	120

Example 2 (Selftest): Word Ladder

- A word ladder transforms a word into another one
- From “rock” to “cash” a word ladder can be build:

rock

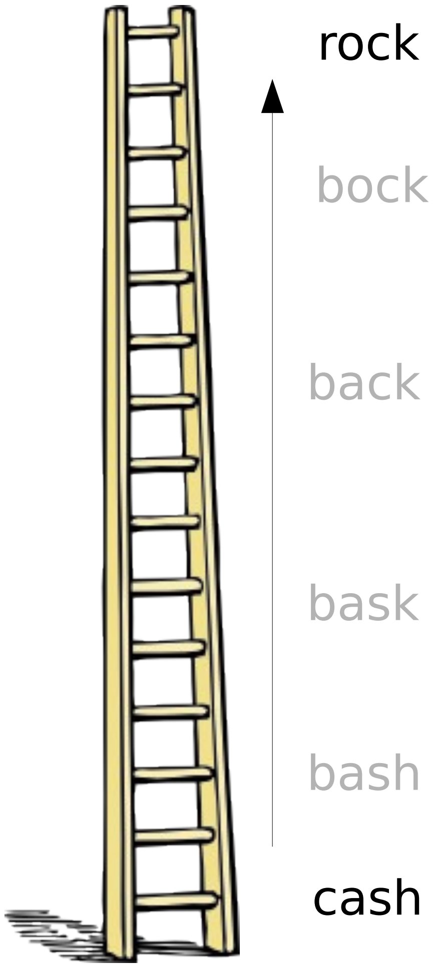
bock

back

bask

bash

cash

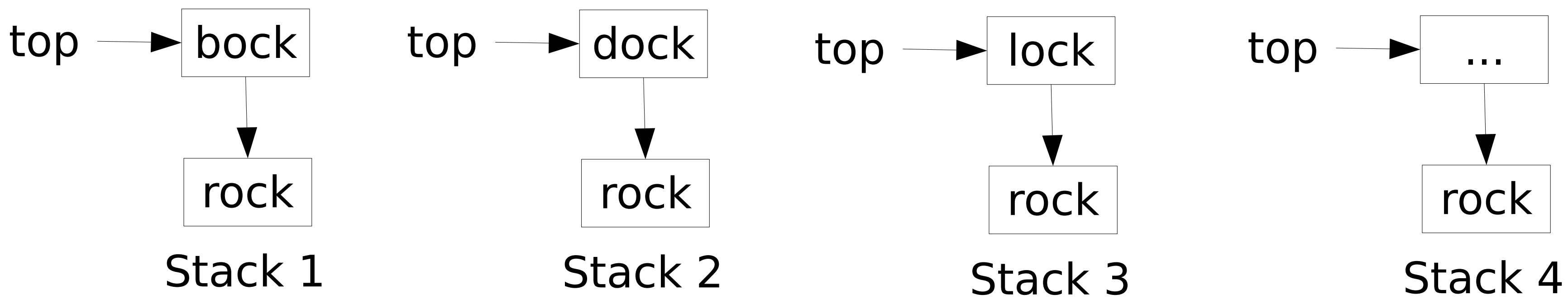


WordLadder

- Preconditions:
  - With every step, just one character is changed
  - Every intermediate word should be a useful one
  - Starting and end word are of the same length
    `start.length() == end.length()`

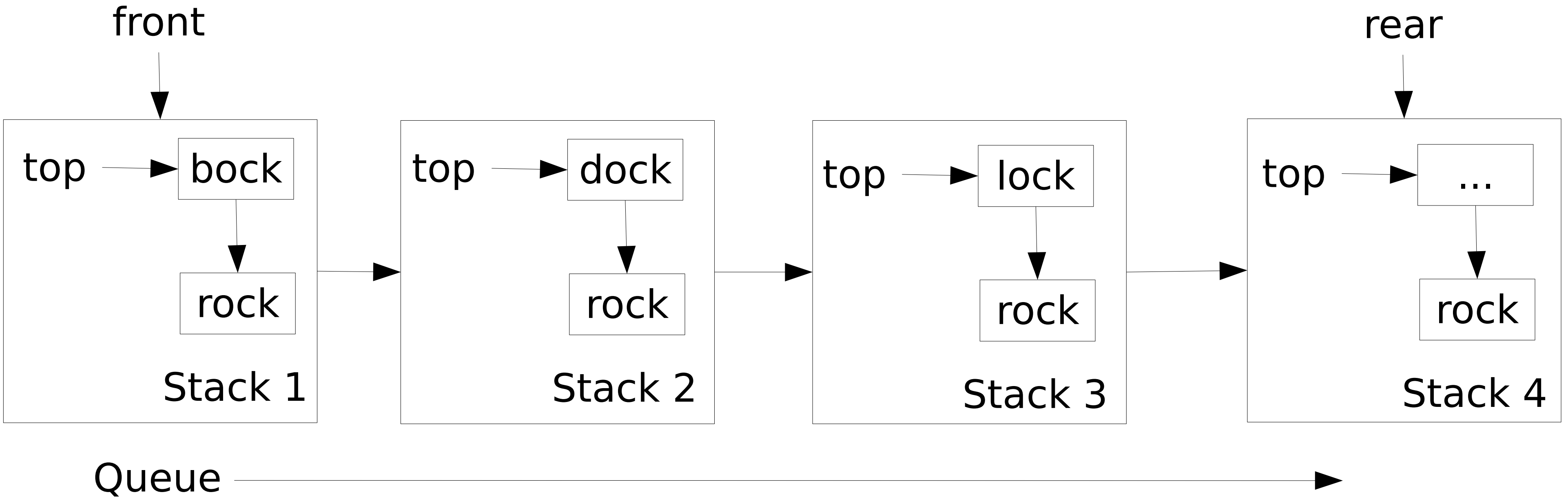
WordLadder

- Task: Build a wordlist between two strings `s1` (rock) and `s2` (cash)
- Algorithm: Search all words with one character difference to `s1`, e.g. bock, dock, lock, etc.
- For every word, built a stack and put `s1` and the actual word on it



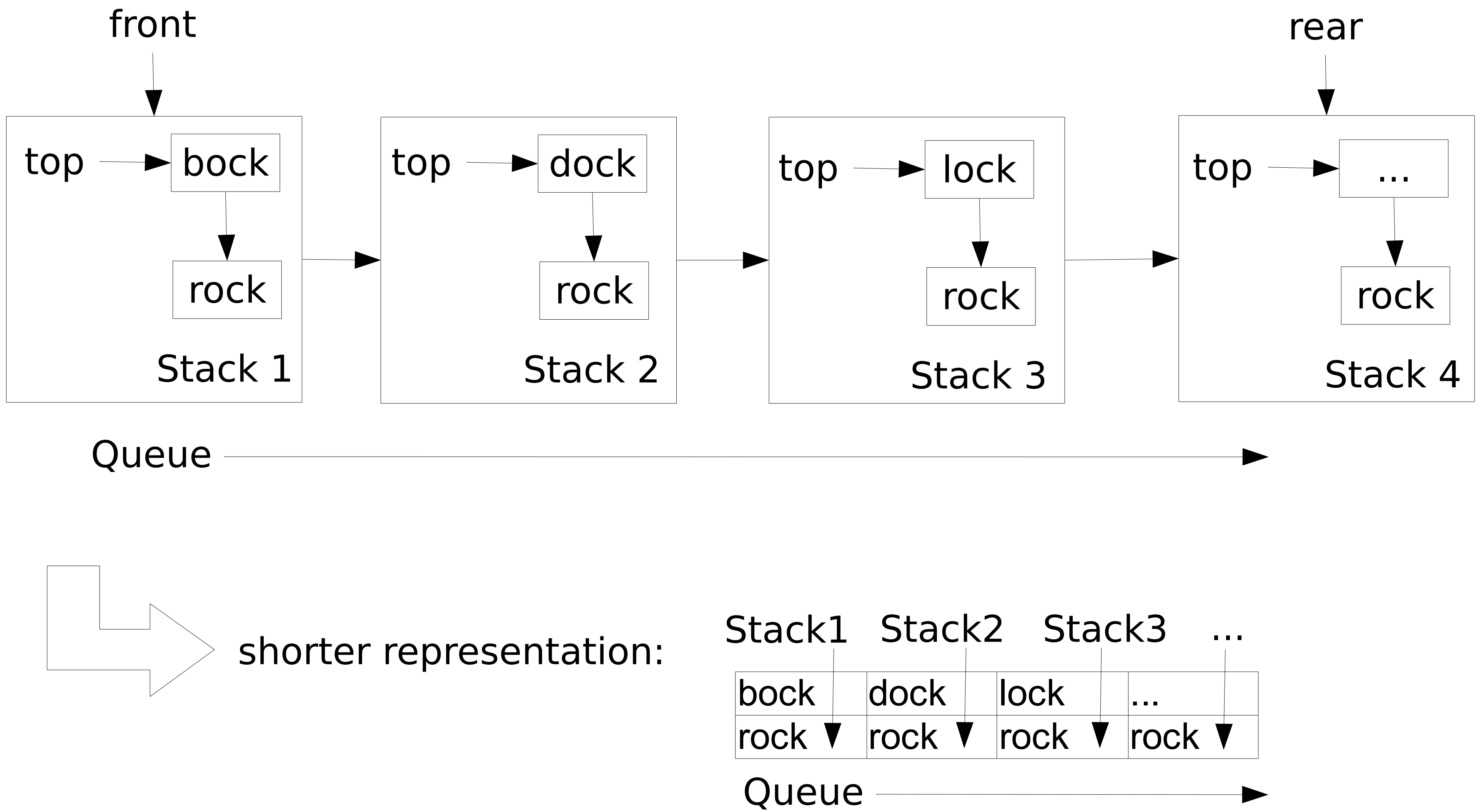
WordLadder

- Put all the stacks on a queue:



WordLadder

- Put all the stacks on a queue:



WordLadder

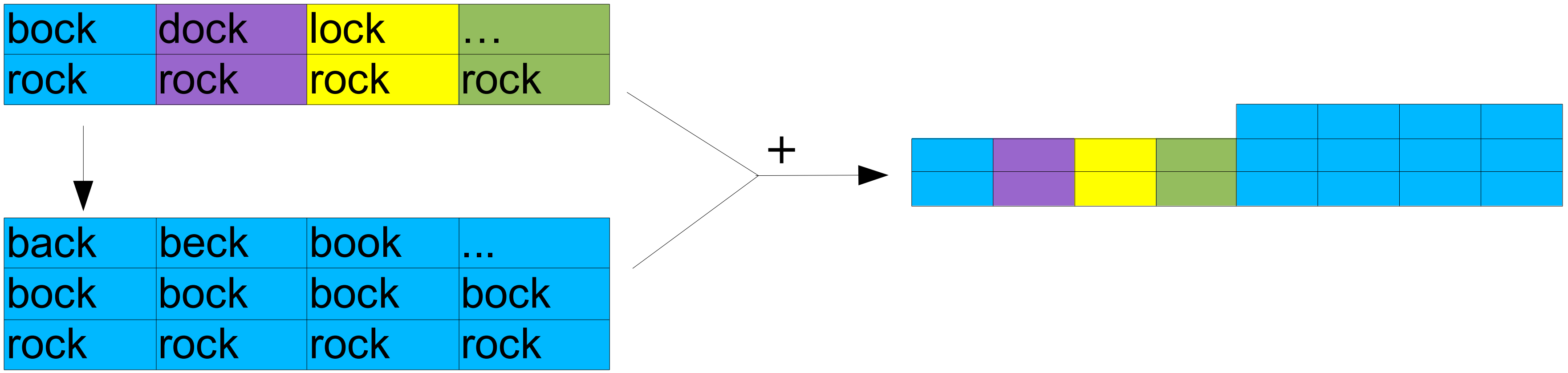
- Take the first stack from the queue and compare it's top word (bock) to **`s2`** (cash). If they are equal, you are done.

```
(“bock” != s2)
```

bock	dock	lock	...
rock	rock	rock	rock

WordLadder

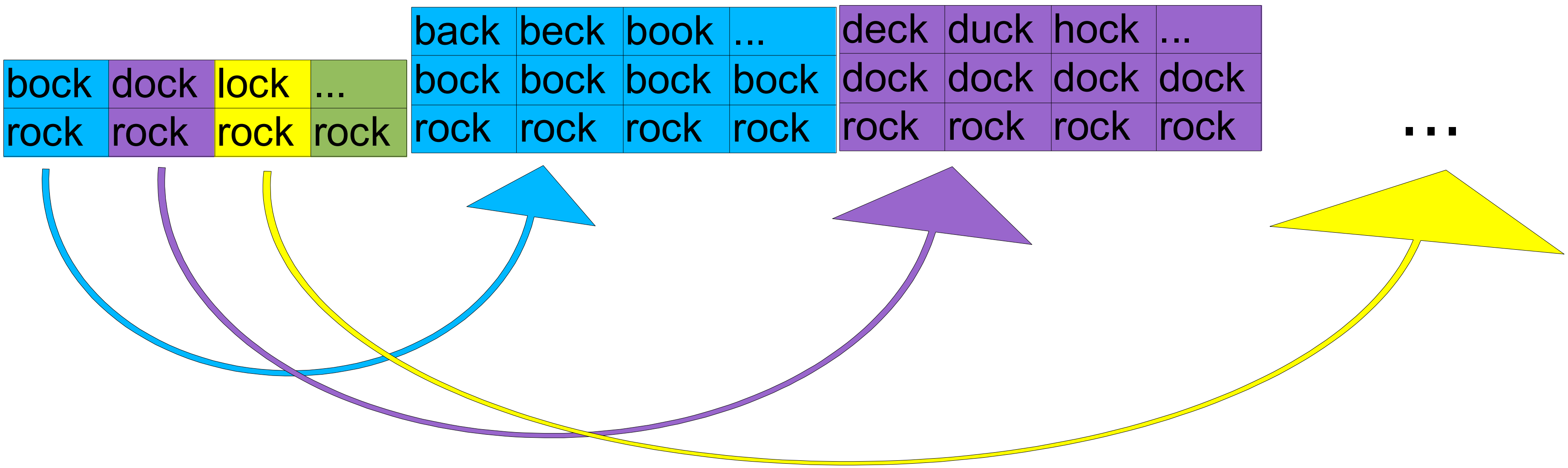
- Take the first stack from the queue and compare it's top word (bock) to **s2** (cash). If they are equal, you are done.
- If not, create a new queue with stacks for the top word and add it to the main queue:



WordLadder

- Repeat this for every stack on your queue, until the queue is empty or you find a stack which contains `s2` as the top word: this stack is your ladder.
- **Don't use the same word a second time! This results in an infinite loop!**

WordLadder


bock
rock
dock
rock
lock
rock
...
rock
back
bock
rock
beck
bock
rock
book
bock
rock
...
bock
rock
deck
dock
rock
duck
dock
rock
hock
dock
rock
...
dock
rock
...

WordLadder

- We will use a dictionary of english words
- The dictionary contains on every line one word
- The first word of a line is the entry, the following are words which have just one different character:

```
babe babu baby bade bake bale [...]
```

WordLadder

- For storing the words from the dictionary, we need an (inner) class **`Word`**:

```
private class Word implements Comparable<Word> {
        private String word;
        private ArrayList<String> neighbors;
         ...
```

```
   -> babe babu baby bade bake bale [...]
```

WordLadder

- A method **`readFile`** reads the dictionary into an **`ArrayList<Word>`**:

```
 private void readFile(String filename) throws
  FileNotFoundException {
...
    Collections.sort(dict);
```

WordLadder

- The constructor takes the filename of the dictionary as argument and calls **`readFile`**
- The main method asks for two arguments on the commandline, creates a new object **`WordLadderSolver`** and calls its method **`solve`**:

```
ArrayList<String> ladder =
          ladderSolver.solve(args[0], args[1]);
```

- Your task (self test): finish the method **`solve`**!