Programming with Data Structures

CMPSCI 187
Spring 2016

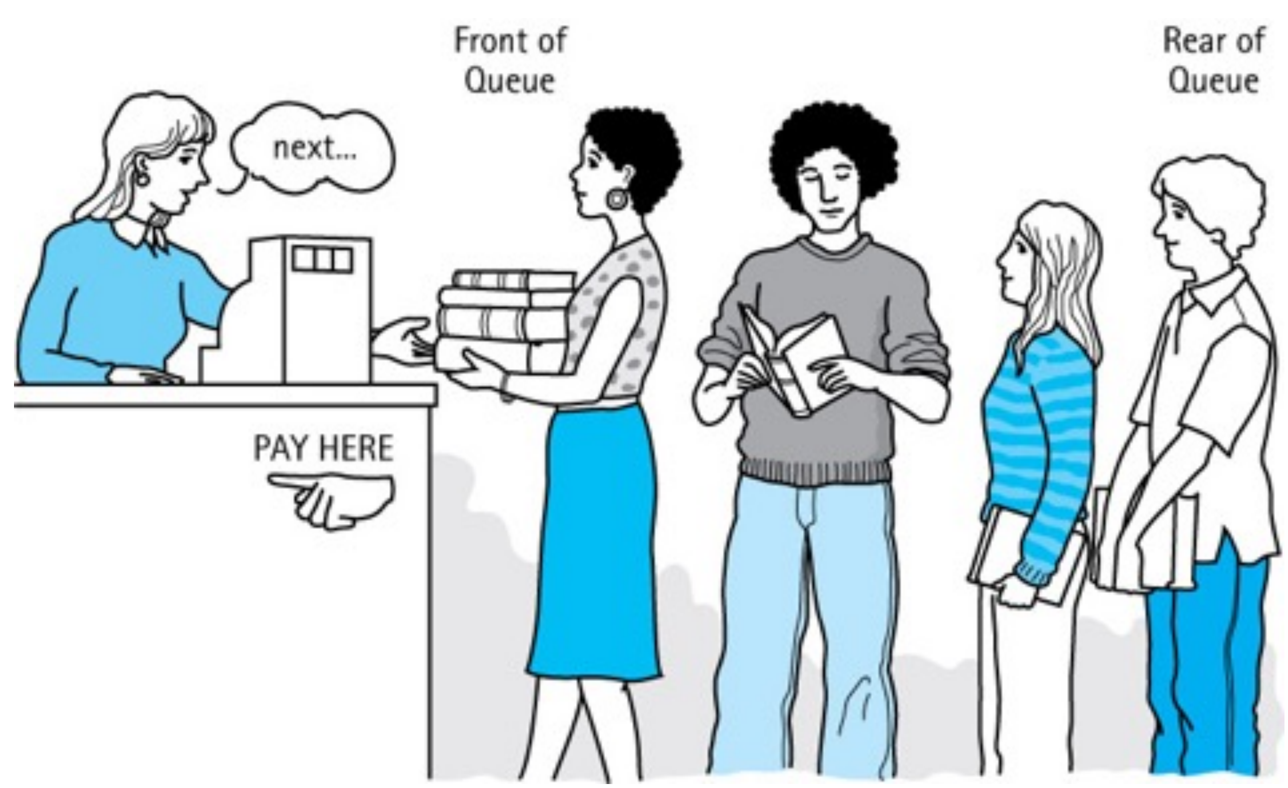
- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Today's Topics

- The Queue ADT (Chapter 5)
  - Basic operation and implementation
  - Application: palindromes
  - Search using stacks and queues

Queue

- The word **Queue** is British for **Line**.
  - The first person that enters the queue gets served first.



Queue

- In computing, a **Queue** is a data structure in which elements are added to the rear and removed from the front; a **First-In-First-Out (FIFO)** structure.
- Similar to a Stack, a Queue is very useful in computer systems:
  - Printer queue
  - Buffers like network buffers and keyboard buffers
  - Process queue
  - Tasks are finished in the order they arrive (almost all types of services).

Queue

- The first element is called the **Front** (or **Head**).
- The last element is called the **Rear** (or **Tail**).
- Two important operations:
  - **Enqueue** — add an element to the rear
  - **Dequeue** — remove the element at the front

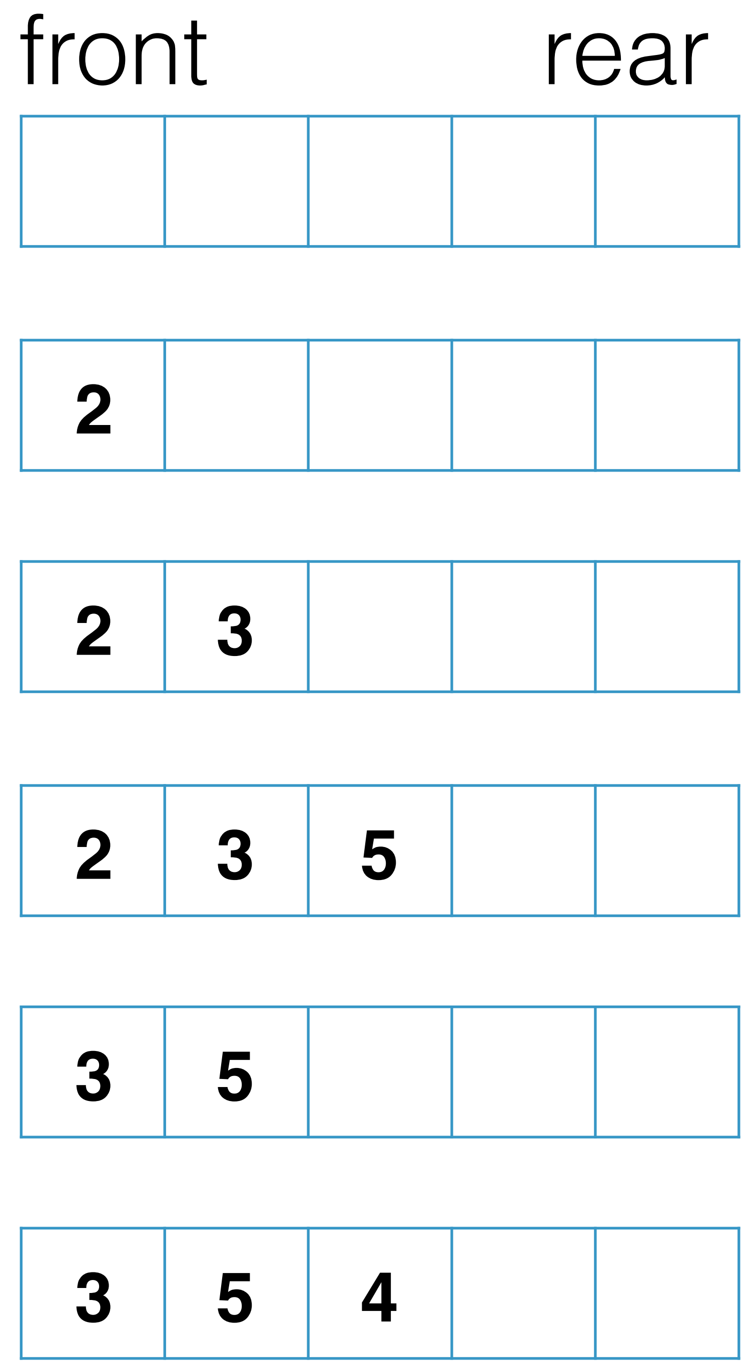
Vegpuff/Wikipedia

Queueing

- In **producer-consumer** settings:
  - Each producer generates new tasks (or elements) to be processed: Enqueue!
  - Each consumers serves / consumes the elements: Dequeue!
  - There may be multiple producers and multiple consumers.
  - Elements may be produced / consumed at different rates. A queue serves as a buffer to handle mismatched rates.

Example operations

- empty queue
- enqueue 2
- enqueue 3
- enqueue 5
- dequeue
- enqueue 4



Clicker Question #1

- enqueue A
- enqueue B
- dequeue
- enqueue D
- enqueue C
- dequeue
- dequeue

Start from an empty queue and perform these operations.

What does the final dequeue operation return: A, B, C, or D?

Clicker Question #2

```
Queue<Dog> q = new Queue<Dog>();
q.enqueue(cardie);    q.enqueue(duncan);
Dog x = q.dequeue();
q.enqueue(ebony);     q.dequeue();
Dog y = q.dequeue();  q.enqueue(x);
System.out.println(q.dequeue());
```

Assuming all the variables are of type **Dog** and have non-null values, which dog's information will be printed by this code?

- (a) Cardie
- (b) Duncan
- (c) Ebony
- (d) none, an exception will be thrown

The Queue Interfaces

- Two versions of the **`Queue`** ADT
  - a bounded version
  - an unbounded version

- Three generic interfaces
  - `QueueInterface<T>`
  - `BoundedQueueInterface<T>`
  - `UnboundedQueueInterface<T>`

Queue Exceptions

- **dequeue** – what if the queue is empty?
  - Throw a `QueueUnderflowException`
  - Also define an `isEmpty()` method to check
- **enqueue** – what if the queue is full (in the bounded version)?
  - Throw a `QueueOverflowException`
  - Also define an `isFull()` method to check

QueueInterface<T>

```
public interface QueueInterface<T>
{
  T dequeue() throws QueueUnderflowException;
  // Throws QueueUnderflowException if this
queue is empty, otherwise removes front
element from this queue and returns it.

  boolean isEmpty();
  // Returns true if this queue is empty,
otherwise returns false.
}
```

BoundedQueueInterface<T>

```
public interface BQI<T> extends QueueInterface<T>
{
  void enqueue(T element) throws
                                  QueueOverflowException;
  // Throws QueueOverflowException if this queue is
full, otherwise adds element to the rear of this
queue.

  boolean isFull();
  // Returns true if this queue is full, otherwise
returns false.
}
```

UnboundedQueueInterface<T>

```
public interface UQI<T> extends QueueInterface<T>
{
  void enqueue(T element);
  // Adds element to the rear of this queue.
}
```

Application: palindromes

- A word or phrase that reads the same forward and backward. Examples:
    - level
    - evil olive
    - stack cats
    - step on no pets

Application: palindromes

- Write pseudo-code for method `ptest(String candidate)` that returns **true** if the input is a palindrome, false otherwise. To start:

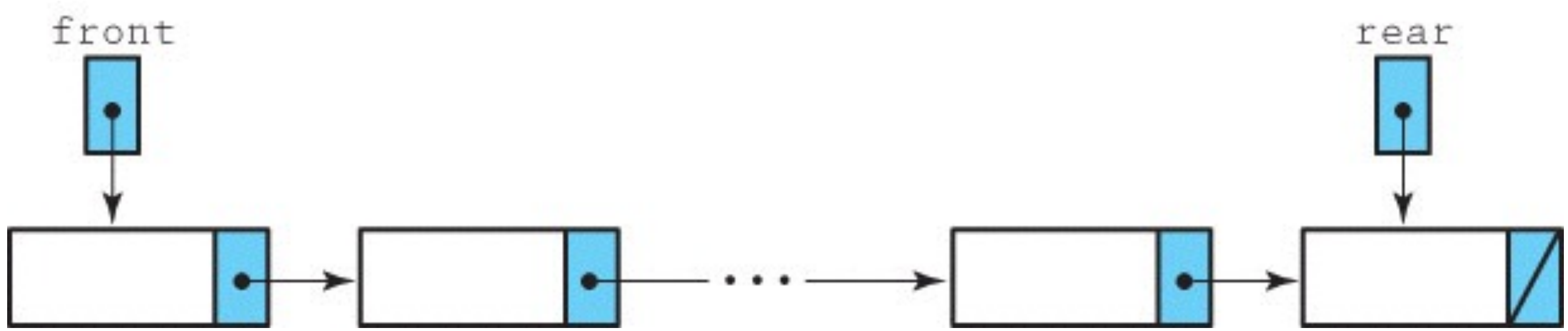
```
Create a new stack s
Create a new queue q

for each character in candidate
  if the character is a letter
    Change the character to lowercase
    Push the character onto the stack s
    Enqueue the character onto the queue q
```

```
// continue from the previous slide

boolean stillPalindrome = true;

while (there are still more characters in the
       structures && stillPalindrome)
  Pop c1 from the stack
  Dequeue c2 from the queue
  if (c1 != c2)
    Set stillPalindrome to false

return (stillPalindrome)
```

A Linked-List Implementation

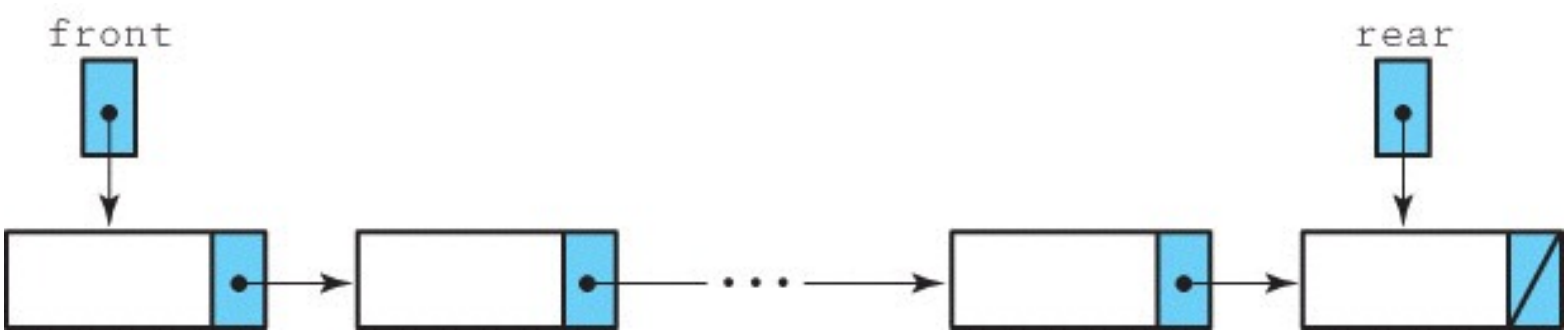
- Implementing a queue with a linked list is quite easy. We will keep the **front** of the queue at the **head** of the list, where we can **dequeue** by removing the head node and returning its contents.

- We need to **enqueue** elements at the **rear** of the list, and (instead of traversing to the end repeatedly) we can save time by maintaining a pointer to the **tail** element of the list.



Linked-Queue Implementation

```
public class LinkedUnbndQueue<T> implements UQI<T>
{
  protected LLNode<T> front;
  protected LLNode<T> rear;
  public LinkedUnbndQueue() {
    front = null;
    rear = null;
  }
}
```



The Enqueue Operation

- To **enqueue** an element, we must make a new node and append it to the end of the list.
- Normally this will just mean making the old rear node point to the new one, and updating the rear pointer.
- But there is a special case **if the queue is empty,** and our new node will be the only element of the queue. What are the values of `front` and `rear`, when the queue is empty?

The Dequeue Operation

- To dequeue, we keep a reference to the element stored at the front node, cut the front node out of the list, and return the element reference.

- If the list is empty we must throw an exception.

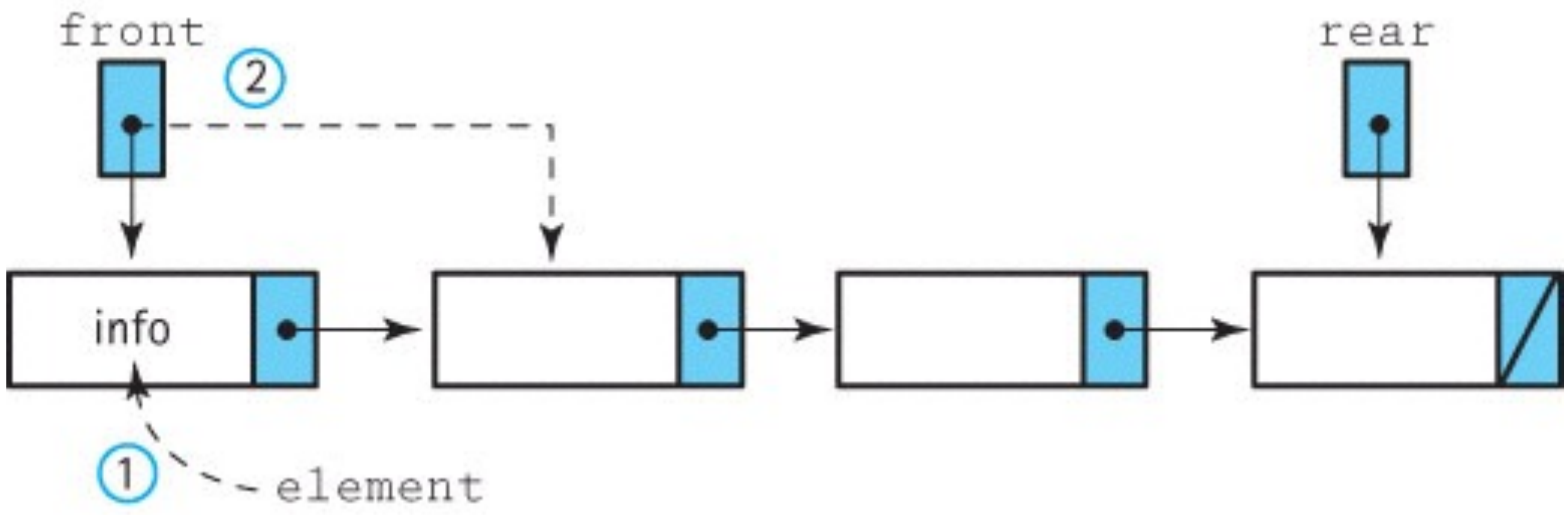
- One complication occurs if the node we are removing is the only one in the queue. How to detect this case? What will happen in this case?

The Dequeue Operation

- To dequeue, we keep a reference to the element stored at the front node, cut the front node out of the list, and return the element reference.

- If the list is empty we must throw an exception.

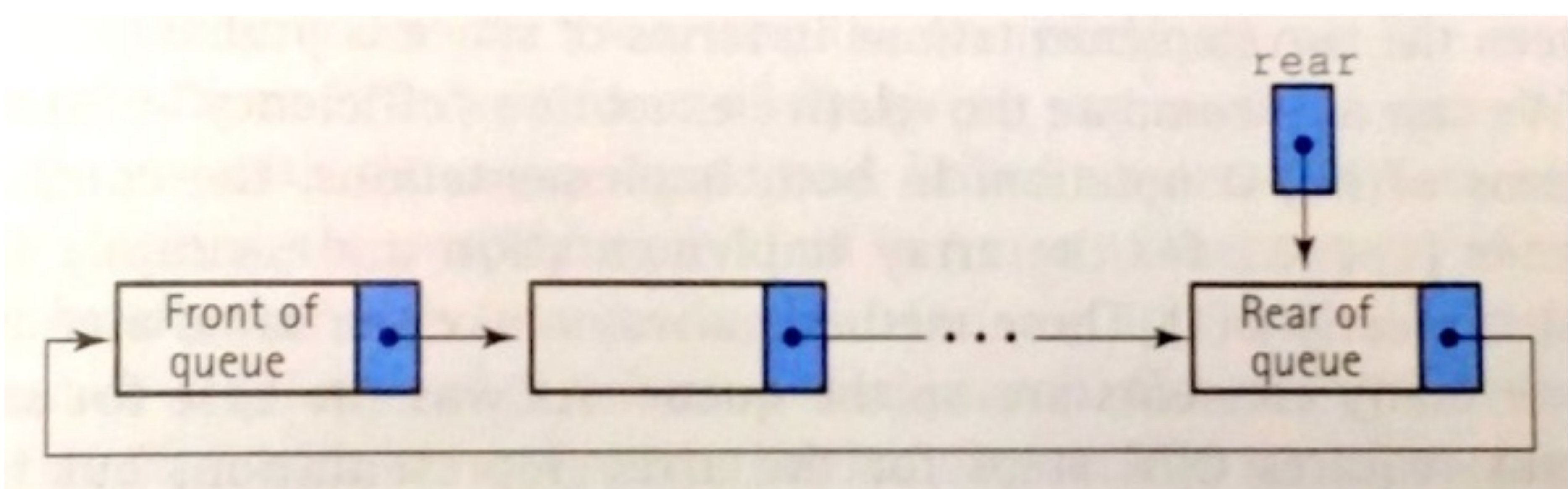
- One complication occurs if the node we are removing is the only one in the queue. We detect this situation as **`front == null`** (after dequeuing) and handle it by setting **`rear`** to **`null`** as well. If we didn't do this, the rear pointer would still point to the removed node.




```
public T dequeue() {
  if (isEmpty())
    throw new QueueUnderflowException("empty queue");
  else {
    T element = front.getInfo();
    front = front.getLink();
    if (front == null) rear = null;
    return element;
  }
}
```

A Circular Linked Queue

- We can make the Linked Queue a bit more efficient by making the list ‘circular’ instead of ‘linear’.
- Instead of keeping both a front pointer and a rear pointer, we can keep just the **rear** pointer, and have its link field point to the front node. This means the **rear node has a non-null link**!
- Question: how do we dequeue?



A Circular Linked Queue

- Here the front node can be referenced as `rear.getLink()`, and we dequeue the front node by `rear.setLink(rear.getLink().getLink())`, of course after remembering its contents.

Clicker Question #3

`rear.setLink(rear.getLink().getLink())`

What would happen if you run the above line of code on a circular linked list that contains only one node? (Hint: think about what a circular list with just one node would look like?)

- (a) The node would be correctly removed
- (b) It throws a `NullPointerException`
- (c) This would cause `rear` to be set to `null`.
- (d) Nothing would change.

A Circular Linked Queue

- Here the front node can be referenced as `rear.getLink()`, and we dequeue the front node by `rear.setLink(rear.getLink().getLink())`, of course after remembering its contents.

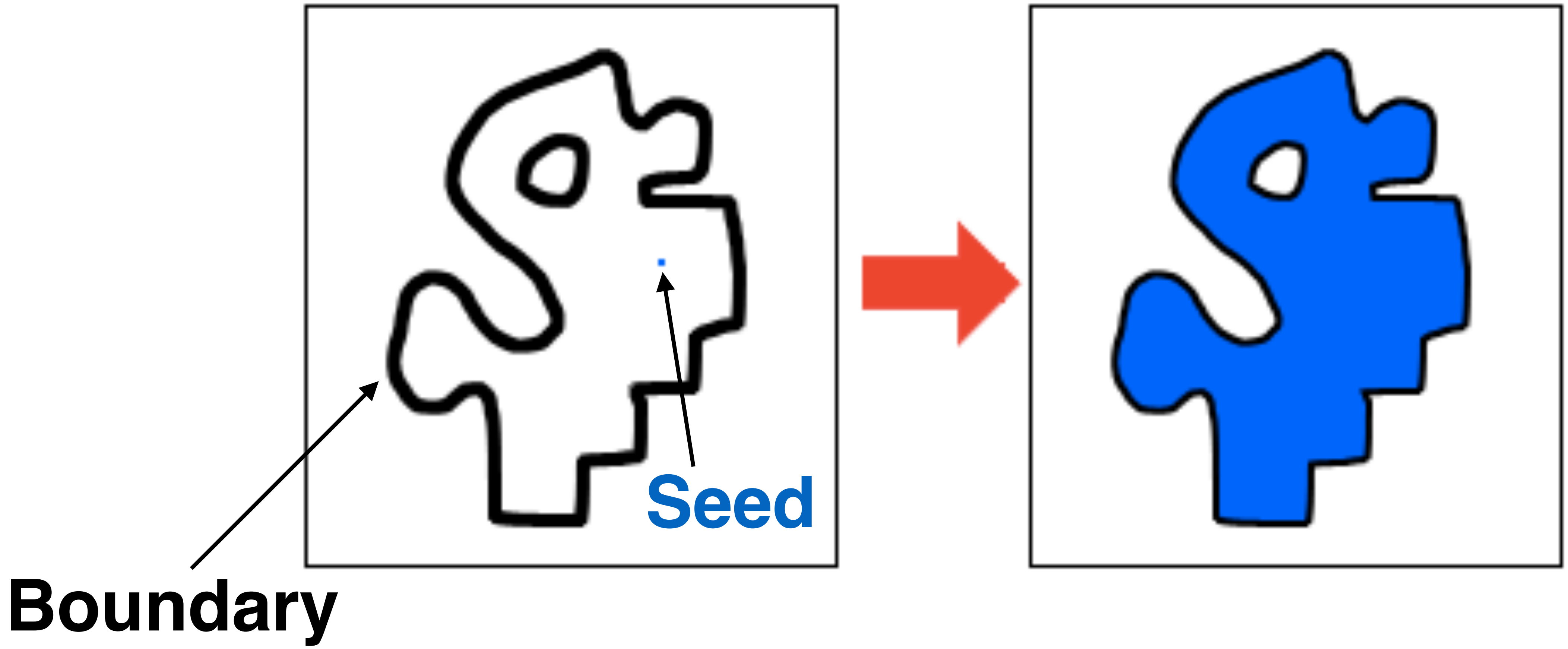
- The important special case is when there is **exactly one node** in the circular list. Here we must set `rear` to `null` if we dequeue. We can recognize this situation by checking if `rear` points to itself (which means the front node is itself). Of course we also need special code to enqueue onto an empty queue.

Stacks vs. Queues

- **Stacks:**
  - LIFO (Last-In-First-Out)
  - Push and pop both modify the top element
  - Computer systems use stacks to manage method calls, including recursive method calls.
- **Queues:**
  - FIFO (First-In-First-Out)
  - Enqueue modifies the rear element; Dequeue modifies the front element.
  - Computer systems use queues to manage buffers, printing jobs, etc.

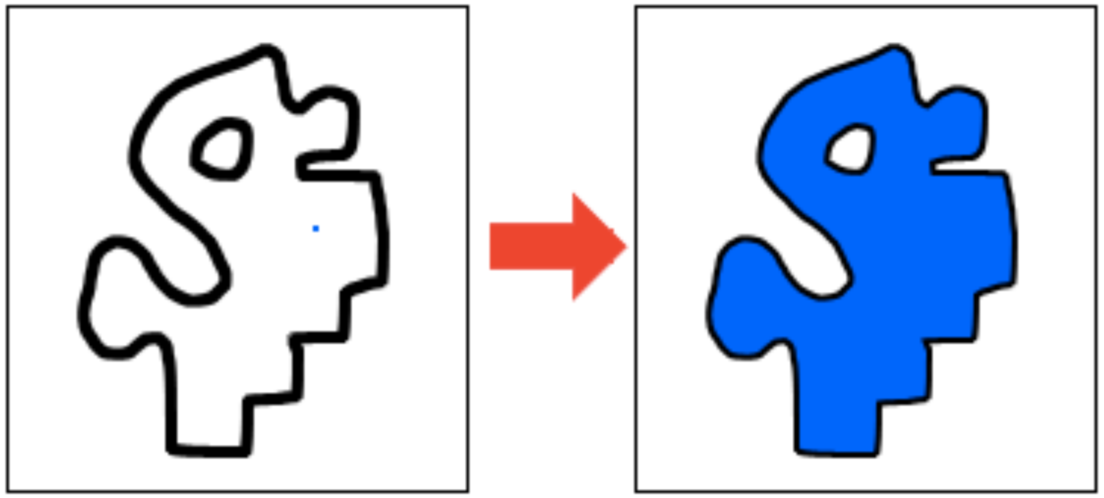
The Flood Fill Algorithm

- A common tool in many paint software, used to fill a **connected** region of pixels with a different color.
- Also known as Bucket Fill, or Seed Fill. Example:



The Flood Fill Algorithm

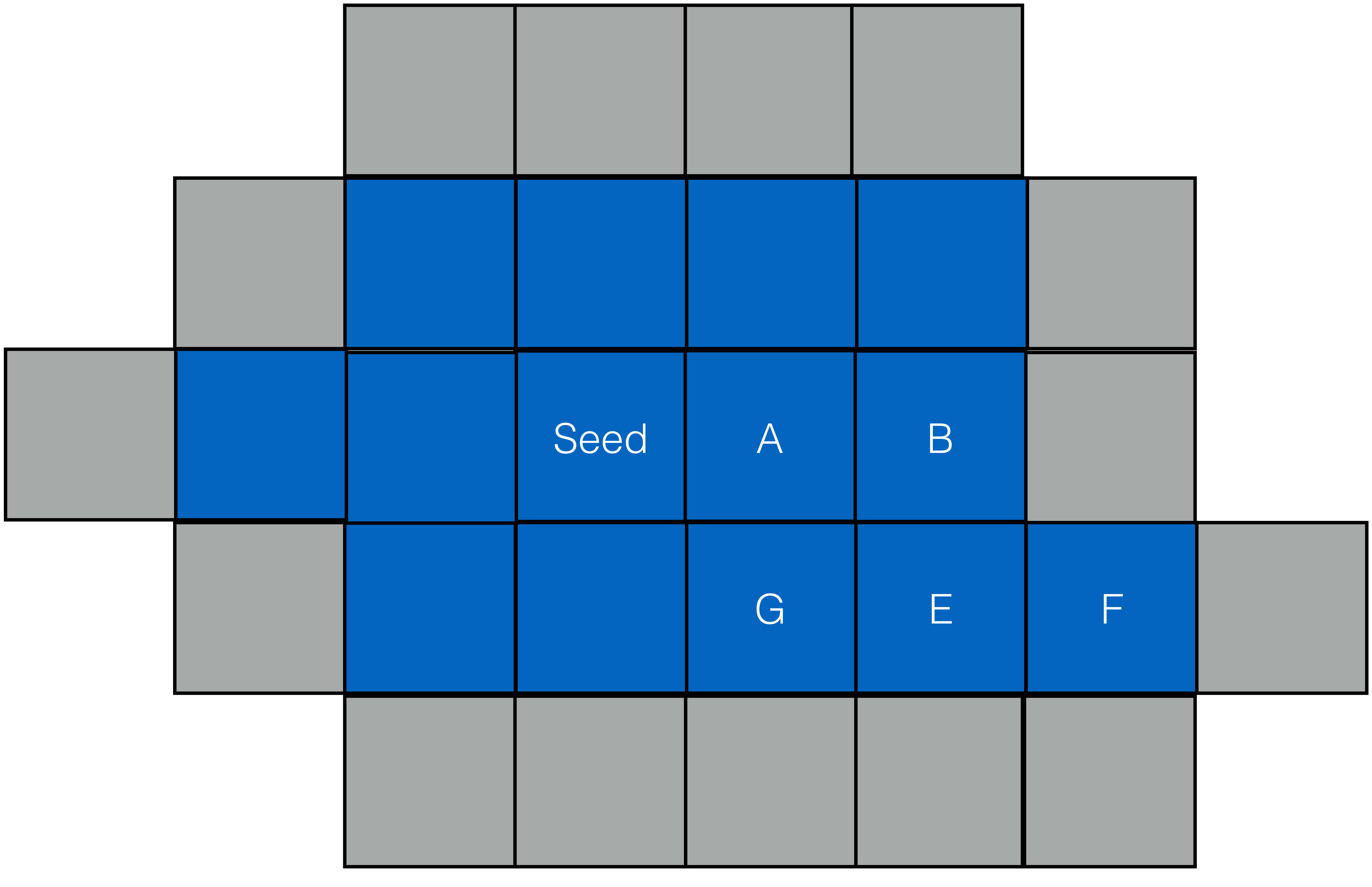
- You are given the location of a **seed** pixel, a **target** color (usually the color of the seed pixel, white in this example), and a **replacement** color (blue here).
- A color other than white is regarded as boundary.
- The algorithm starts with the seed pixel, color it blue, then visits its **four neighbors**. If a neighbor's color is still white, we replace it by blue, and proceed to visit its neighbors in turn.
- This is recursion!



The Flood Fill Algorithm

```
public void floodfill(int x, int y,
                      Color tar, Color rep) {
  if (tar.equals(rep)) return;
  if (!image.getPixelColor(x, y).equals(tar)) return;
  image.setPixelColor(x, y, rep);
  floodfill(x+1, y, tar, rep);   // east
  floodfill(x-1, y, tar, rep);   // west
  floorfill(x, y+1, tar, rep);   // south
  floorfill(x, y-1, tar, rep);   // north
}
```

The Flood Fill Algorithm



Clicker Question #4

Using the Flood Fill algorithm we just learned, in what order will the four corner pixels (marked 1, 2, 3, 4) be colored blue?

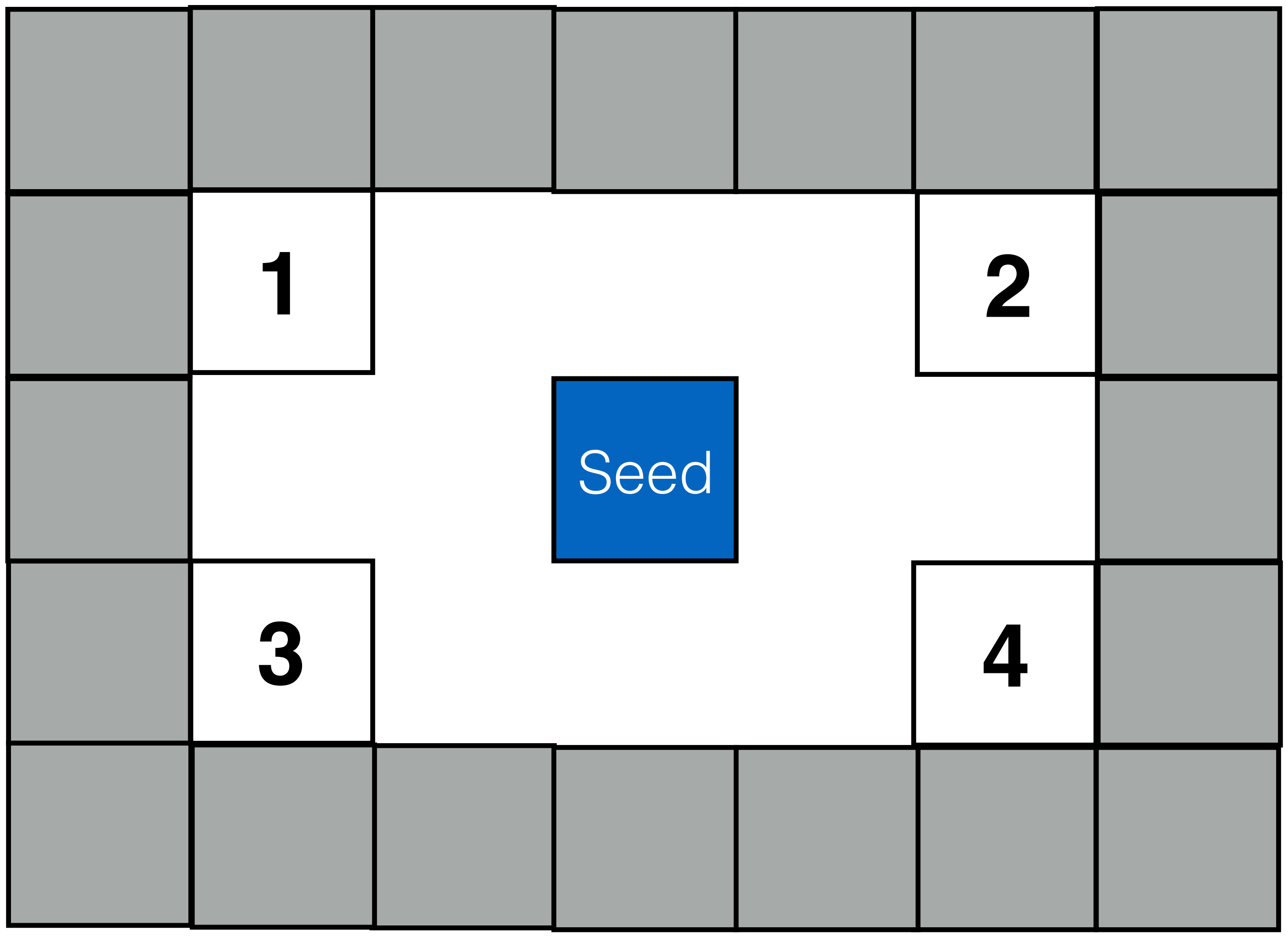
(a) 1, 2, 3, 4

(b) 4, 3, 2, 1

(c) 4, 3, 1, 2

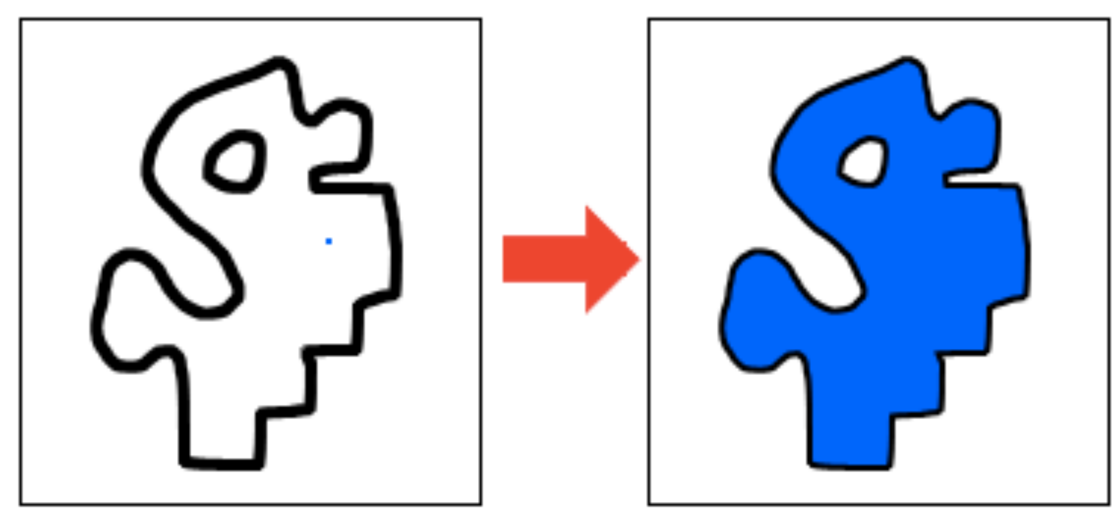
(d) 3, 4, 2, 1

(e) 3, 4, 1, 2



The Flood Fill Algorithm

- What are the base cases here?
- A pixel may be visited multiple times (because there are different paths to reach a pixel from the seed pixel). But it will be colored blue only upon the first visit. The second time you see it, its color is no longer white, hence we don’t perform recursion from this pixel again.
- Recursion implicitly uses the system stack. You can certainly re-implement flood fill with an explicit stack. Think about how to implement it.



Searching With a Queue

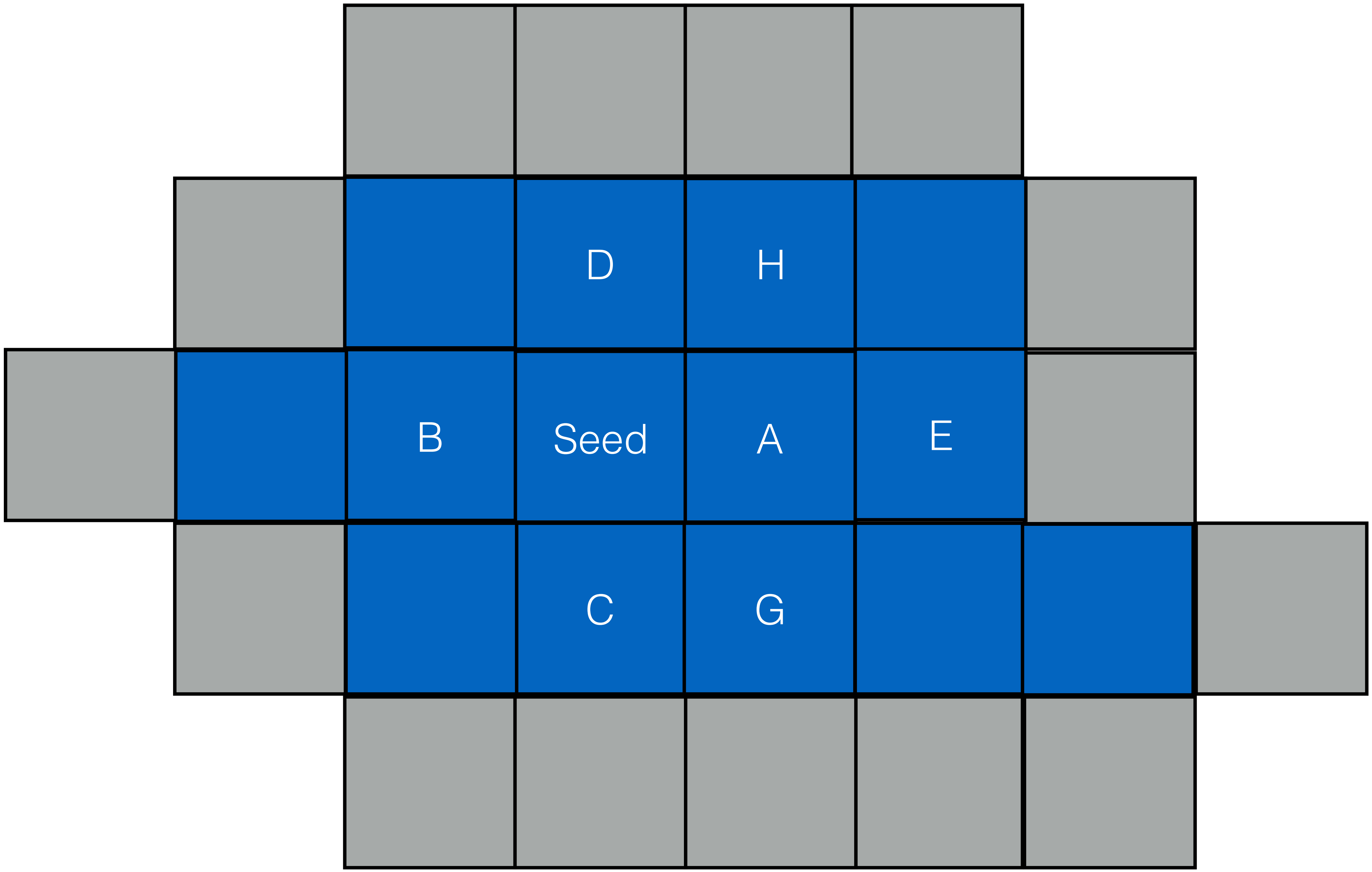
- Searching with a **stack** is called **Depth-First** Search (**DFS**).
    - Imagine the solution space is organized as a tree, DFS explores as far as possible in each tree branch before backtracking.

- Searching with a **queue** is called **Breath-First** Search (**BFS**).
    - BFS explores all the neighbors at the same level first, before moving to the next level of neighbors.

Searching With a Queue

- Imagine using a Queue to implement flood fill.
- Start at the seed pixel and an empty queue, add all four neighbors to the queue.
- Dequeue the first element (the right neighbor of the seed), add all its neighbors to the queue.
- Dequeue the second element (the left neighbor of the seed), add all its neighbors to the queue.
- Proceed until the queue is empty.

Flood Fill with a Queue



Searching With a Queue

- BFS is often used to find the shortest path solution. Because it explores tree nodes at the same level first, as soon as it finds a solution, that would be the shortest path solution from the root of the tree.
- For example:
  - Shortest path out of a maze
  - Shortest distance from the seed pixel to boundary.

Flood Fill with a Queue