Today's Topics

- The Queue ADT
  - 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.

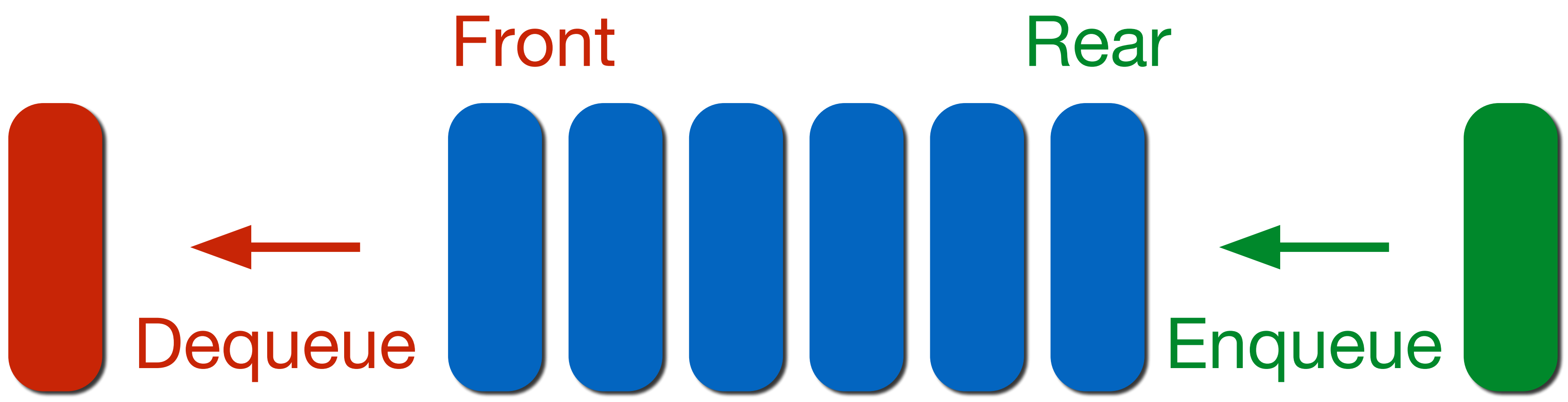
next...
Front of Queue
Rear of Queue
PAY HERE

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

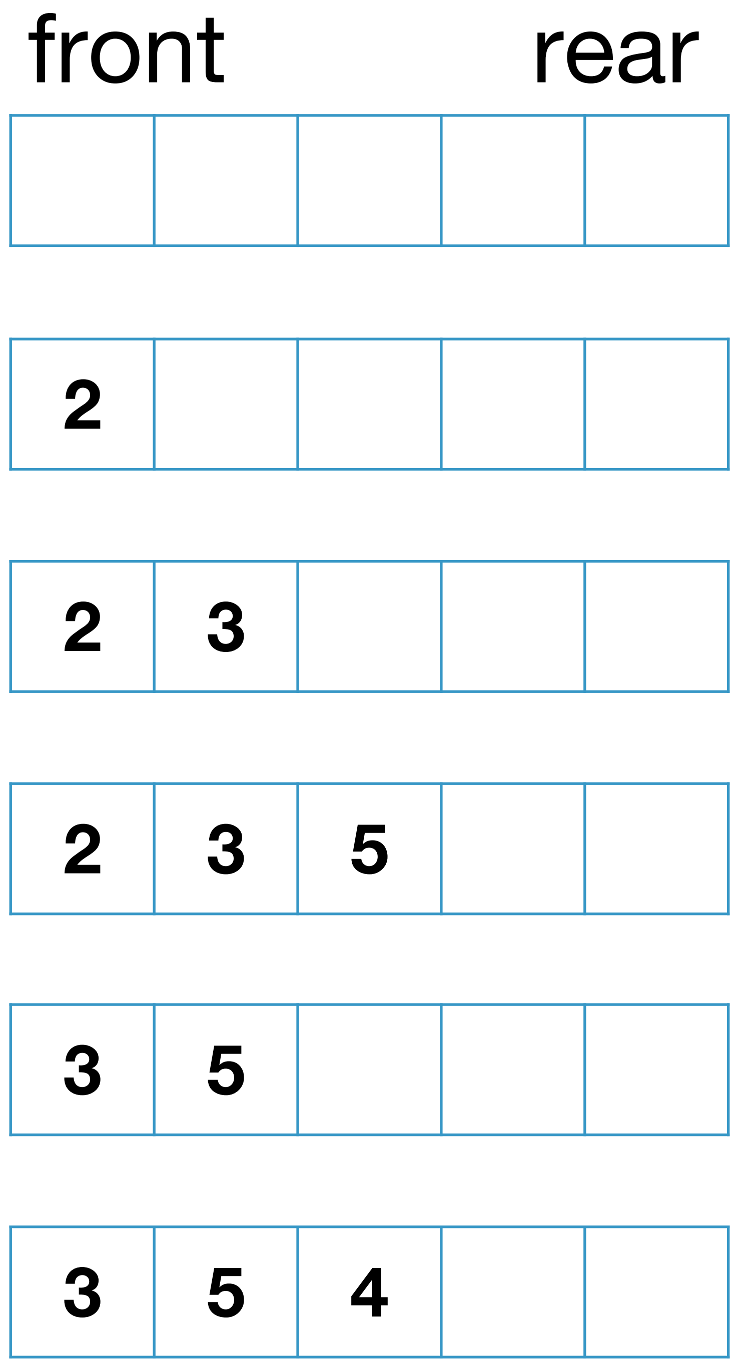
- The first element is called the **Front** (or **Head**).
- The last element is called the **Rear** (or **Tail**).
- Two important operations:
  - **Enqueue** — add an element to the rear
  - **Dequeue** — remove the element at the front



Vegpuff/Wikipedia

Example Operations

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



Clicker Question #1

- enqueue D
- enqueue A
- dequeue
- enqueue B
- 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?

Answer on next slide

The Generic Queue Interface

```
public interface QueueInterface<T> {
  T dequeue() throws
              QueueUnderflowException;

  void enqueue(T e) throws
                    QueueOverflowException;

  boolean isEmpty();
  boolean isFull();
}
```

Queue Exceptions

- **Dequeue** – what if the queue is empty?
  - Throw a `QueueUnderflowException`
  - You can call `isEmpty()` to check
- **Enqueue** – what if the queue is full (in the array-based version)?
  - Throw a `QueueOverflowException`
  - You can call `isFull()` to check

Application: Palindromes

- A word or phrase that reads the same forward and backward (ignoring case and spaces). Examples:
  - **`level`**
  - **`Evil olive`**
  - **`stack cats`**
  - **`step on no pets`**

Application: Palindromes

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

```
s = new Stack<Character>();
q = new Queue<Character>();
for each character c in string
  if (c is a letter) {
    change c to lowercase;
    s.push(c);
    q.enqueue(c);
  }
}
```

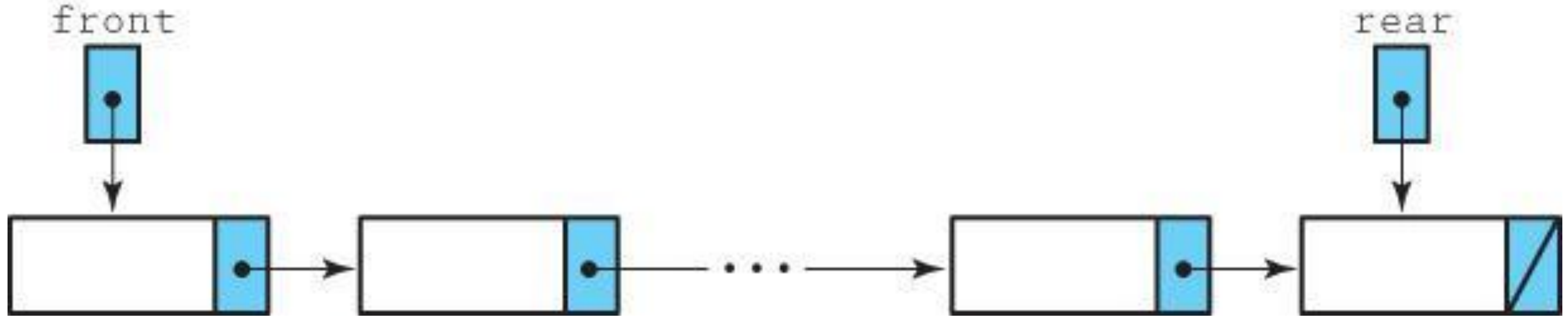
```
// continue from the previous slide
boolean isPalindrome = true;

while (!s.isEmpty()) {
  c1 = s.pop();
  c2 = q.dequeue();
  if (c1 != c2) {
    isPalindrome = false; break;
  }
}
return isPalindrome;
```

A Linked-List Implementation

- Implementing a queue with a linked list is quite easy. The **front** of the queue is the **head** of the list. We can **dequeue** by removing the head node and returning its contents.

- We need to **enqueue** elements at the **rear** of the list. For efficiency, instead of traversing the list repeatedly, we can keep track of a **tail / rear** pointer.

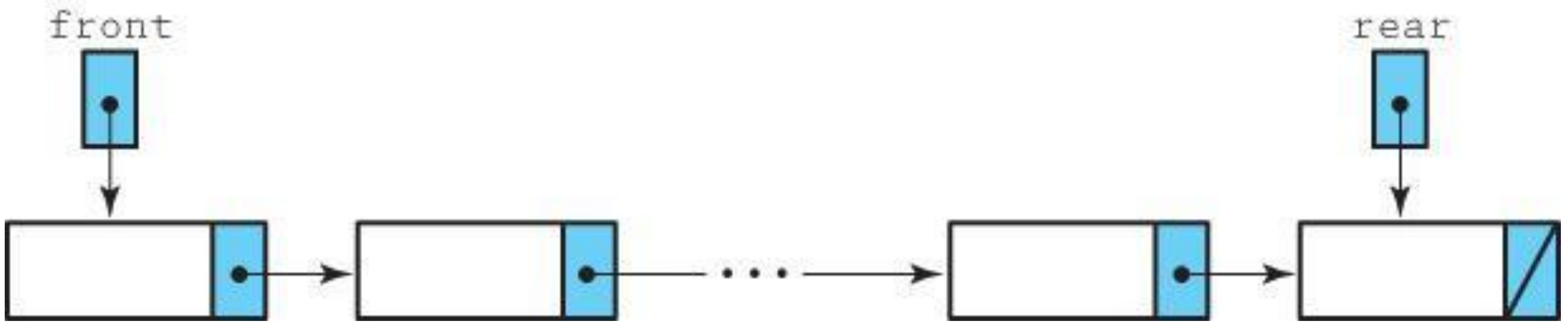


Linked-Queue Implementation

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

  protected LLNode<T> front = null;
  protected LLNode<T> rear = null;

}
```

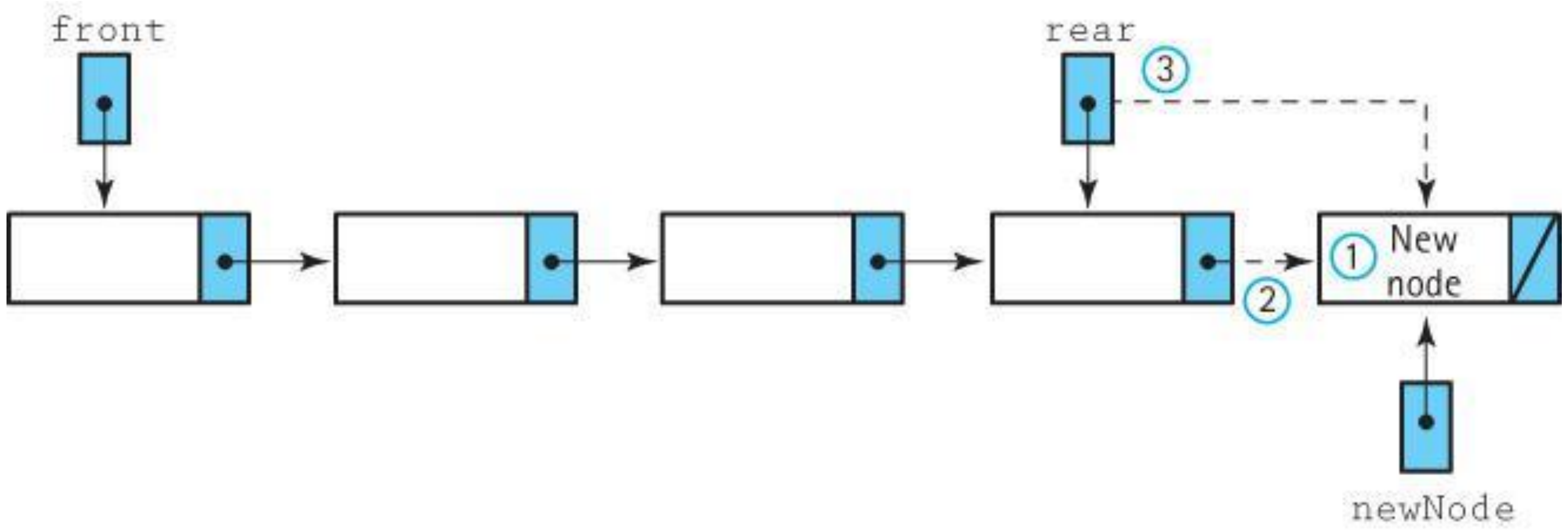


The **Enqueue** Operation

- 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 current rear node point to the new one, and updating the rear pointer. But remember, we need to handle an edge case, which is when **the queue is empty.**

- This is exactly the tail insertion method in linked list, as we've learned already.



```
public void enqueue(T element) {
  LLNode<T> newNode = new LLNode<T>(element);
  if (rear == null) {
    front = newNode;
  else
    rear.setLink(newNode);
  rear = newNode;
}
```

The **Dequeue** Operation

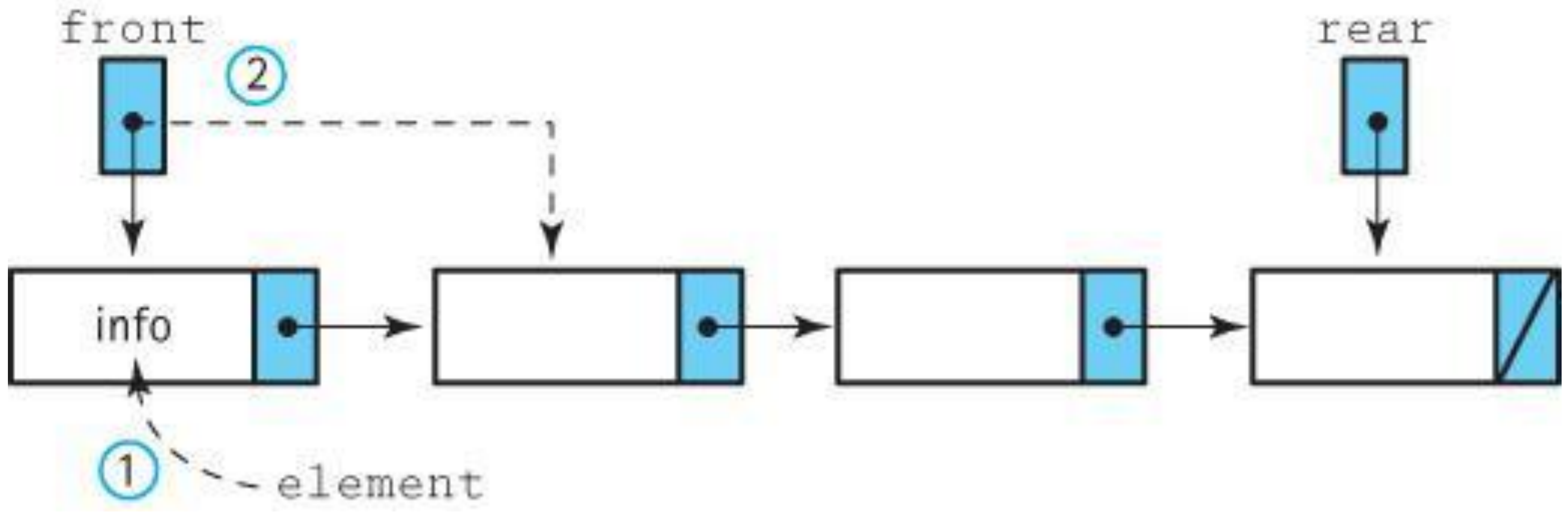
- To **dequeue**, we keep a reference to the element stored at the front node, remove the front node out of the list, and return the element.
- If the list is empty we must throw an exception.
- One edge case occurs if the node we are removing is **the only one** in the queue. How to detect this case? What will happen in this case?

The **Dequeue** Operation

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

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

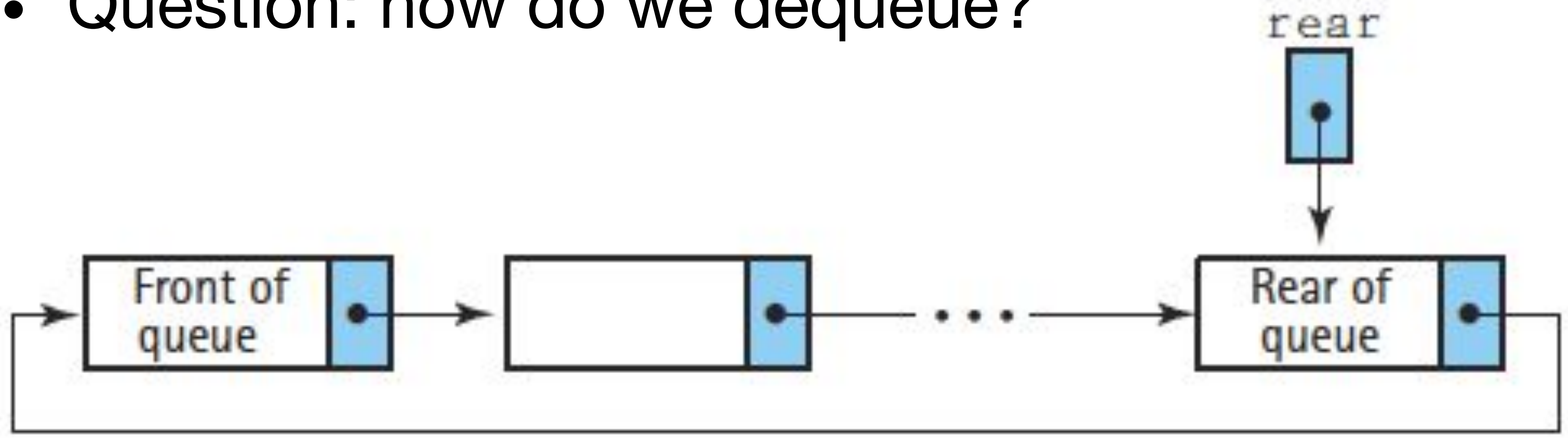
- One edge case occurs if the node we are removing is **the only one** in the queue. We detect this by **checking `front == null`** (after dequeuing) and handle it by setting `rear` to `null` as well. If we didn't do this, the rear pointer would still point to the removed node.



```
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 <u>non-null link</u>**!
- Question: how do we dequeue?



A Circular Linked Queue

- We can find the front node by: **`rear.getLink()`**
- We can dequeue the front node by `rear.setLink(rear.getLink().getLink())`

Clicker Question #2

```
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) Nothing would change.

(b) The node would be correctly removed

(c) It would throw a `NullPointerException`

(d) It would cause `rear` to be set to `null`.

(e) It would cause a new node to be inserted.

Answer on next slide

A Circular Linked Queue

- An important edge 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 it's also the front node).

- We also need to make a special case for **enqueuing onto an empty queue**.

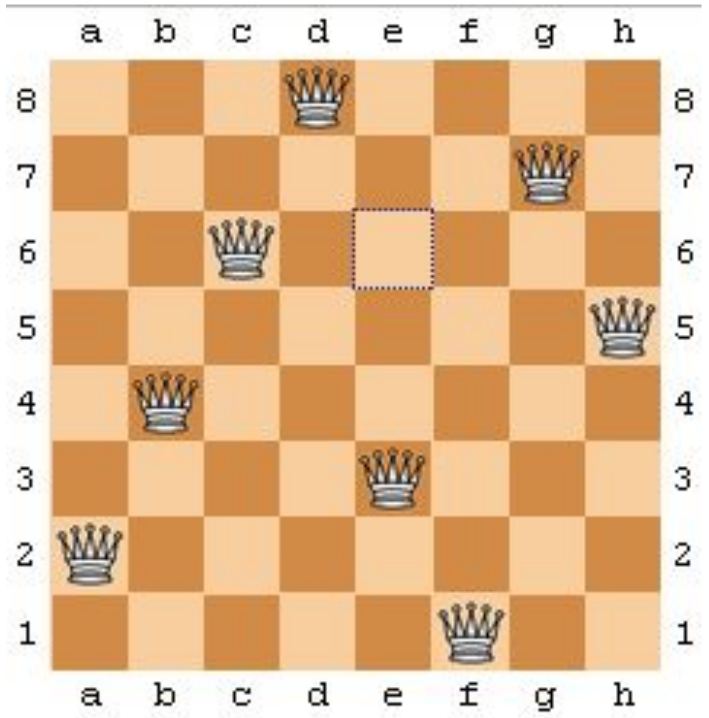
- A great study exercise is to do this yourself.

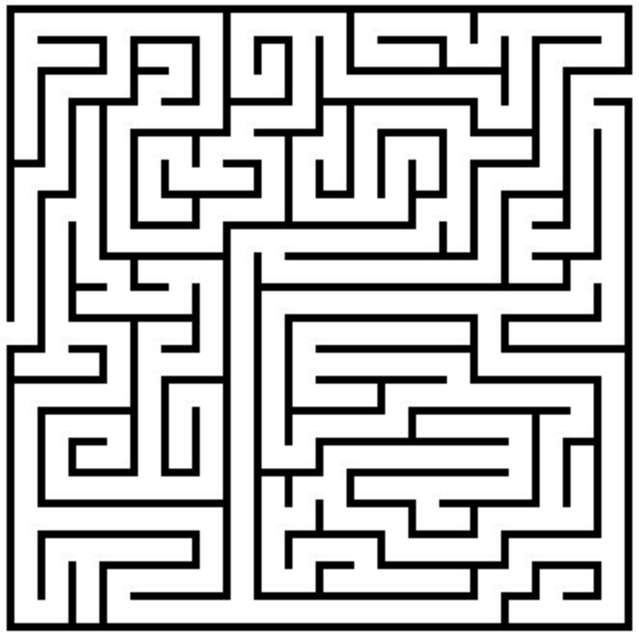
Stacks vs. Queues

- **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.

Searching with Stacks and Queues

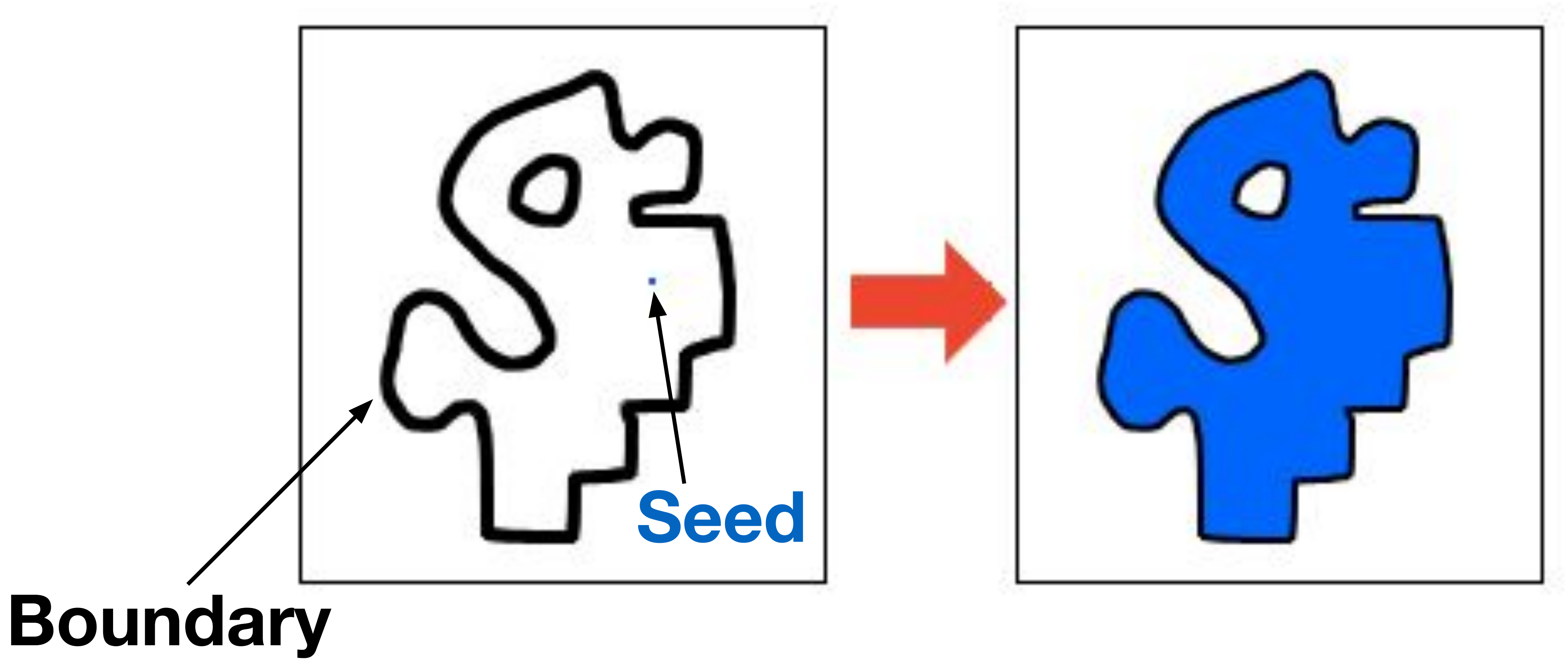
- Stacks and queues are frequently used in **searching problems**: finding some or all solutions to a computational problem, often constraint satisfaction problems. Examples:
  - Eight Queens Puzzle
  - Knight's Tour
  - Sudoku
  - Maze
  - Flood Fill





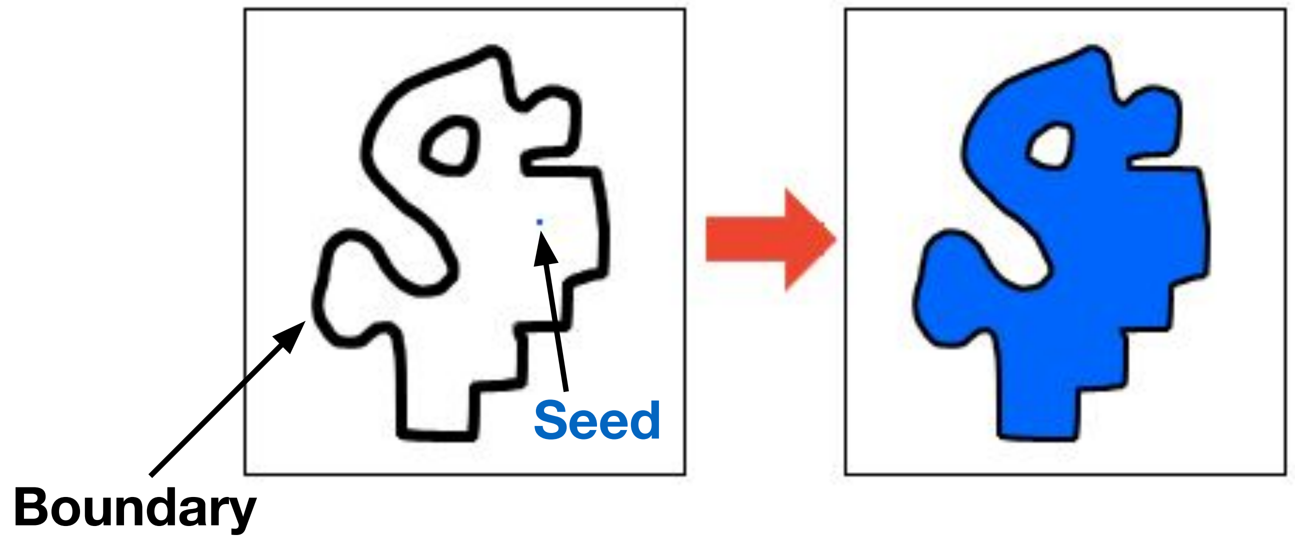
The Flood Fill Algorithm

- 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!

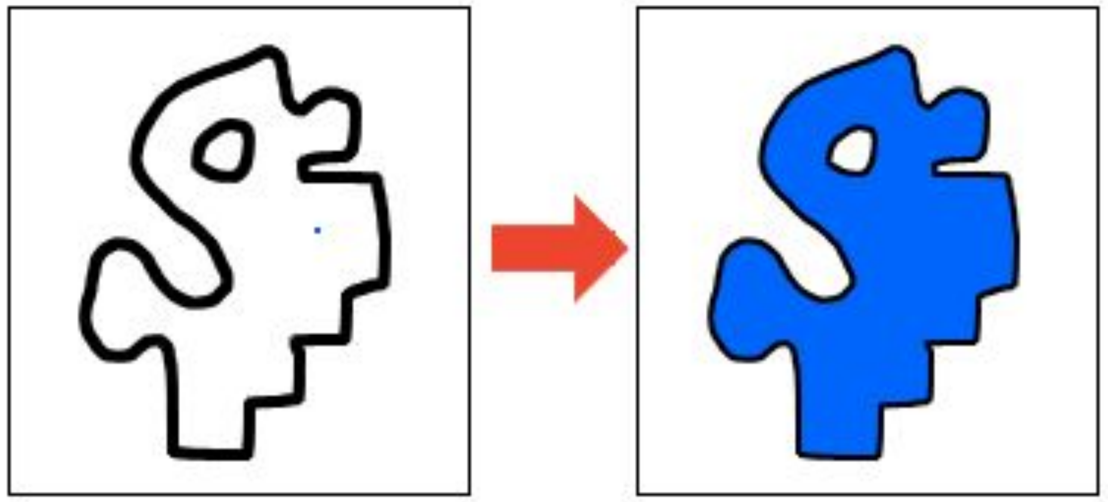


The Flood Fill Algorithm

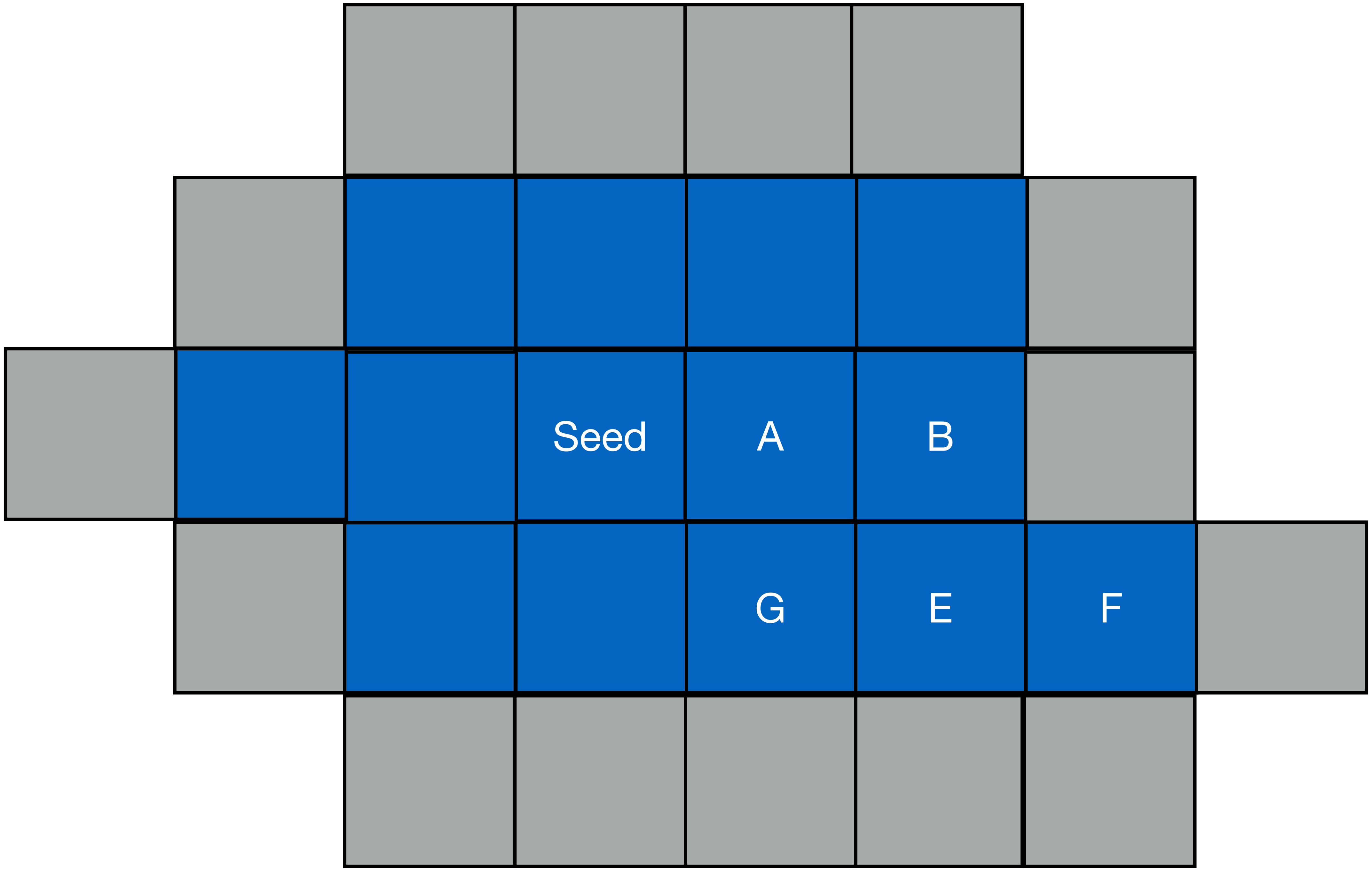
```
// replaces a pixel of target color by new color
// then recursion on the four neighbors.
// assume target color is not the same as new color
public void floodfill(int x, int y,
                      Color tar, Color newcol) {
  if(!image.getPixelColor(x,y).equals(tar))
    return; // already filled or is boundary
  image.setPixelColor(x, y, newcol);
  floodfill(x+1, y, tar, newcol);   // east
  floodfill(x-1, y, tar, newcol);   // west
  floorfill(x, y+1, tar, newcol);   // south
  floorfill(x, y-1, tar, newcol);   // north
}
```

The Flood Fill Algorithm

- What are the base cases here?
- Image the target color is white, new color is blue. 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 system stack. You can implement flood fill with an explicit stack. Think about how to implement it.



The Flood Fill Algorithm



Clicker Question #3

Using the Flood Fill algorithm we just learned, in what order will the four corner pixels (marked A, B, C, D) be colored blue?

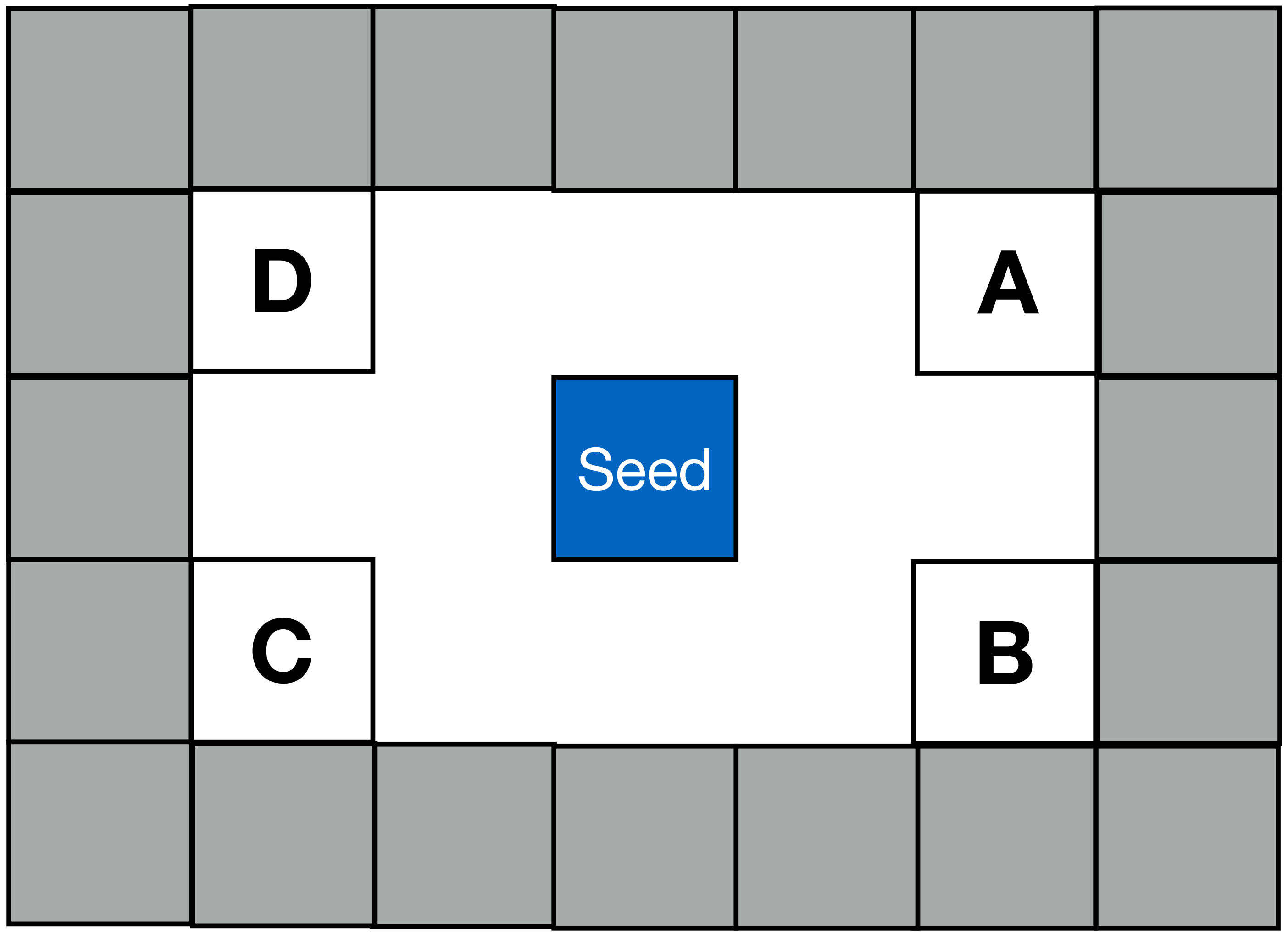
(a) A, B, C, D

(b) B, C, D, A

(c) B, C, A, D

(d) C, D, A, B

(e) B, A, C, D

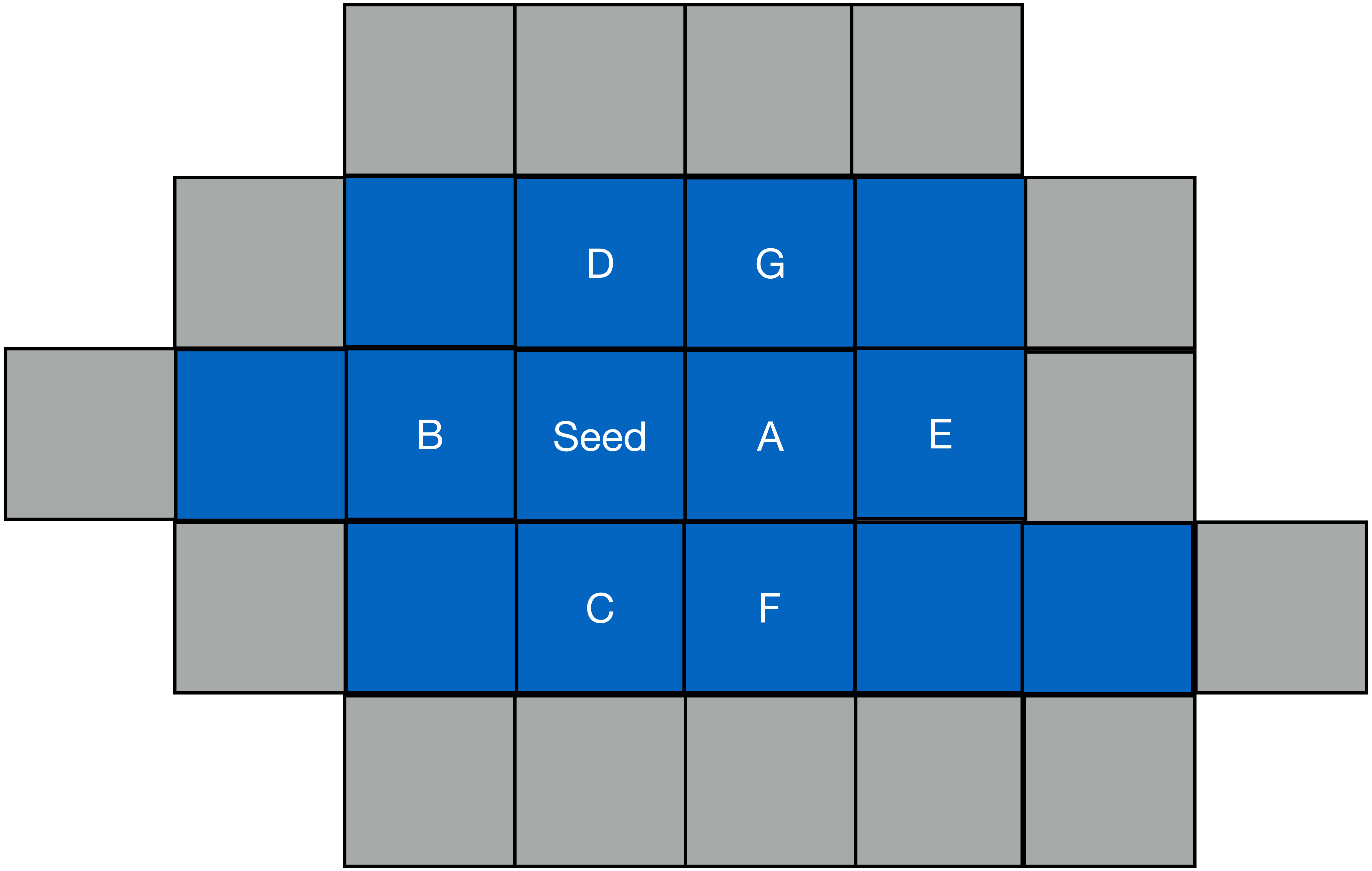


Answer on next slide

Flood Fill 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



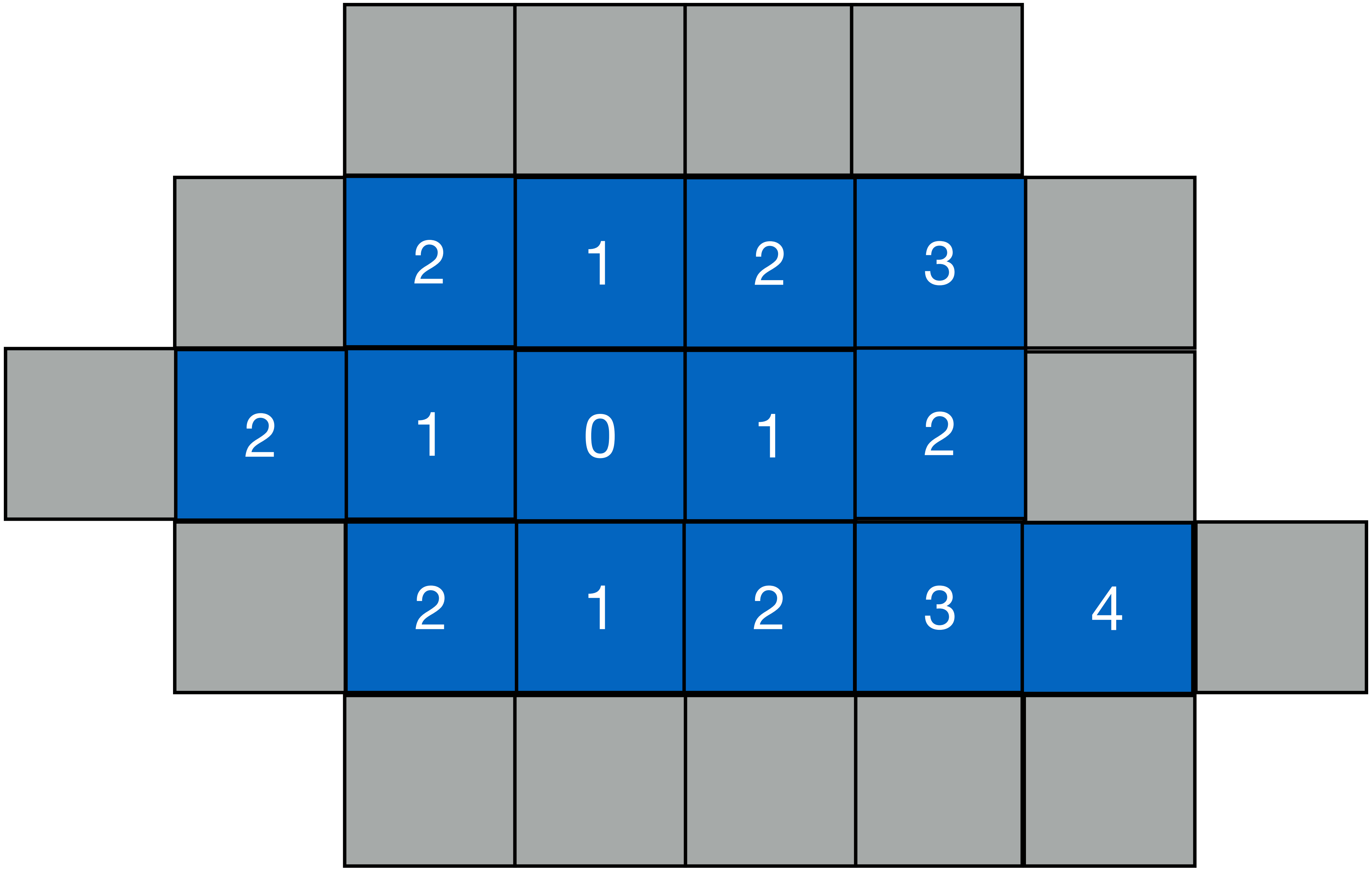
Searching With Queues vs. Stacks

- We will study more about these in the future. To give you heads up:

- Searching with a **Stack** is often called **Depth-First Search (DFS)**. It's often used to find **a** solution as quickly as possible.

- Searching with a **Queue** is called **Breadth-First Search (BFS)**. It's often used to find the **best** (e.g. shortest path) solution. For example, the shortest path out of a maze, the shortest distance from the seed pixel to the boundary.

Queue-based Flood Fill can find the shortest distance from the seed pixel to any pixel in the area.



Questions

Queueing

- In **producer-consumer** settings:
  - Each producer generates new tasks (or elements) to be processed: Enqueue!
  - Each consumer 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.

Searching With a Stack

- A generally useful algorithm is called **backtracking**. It incrementally builds the candidates to the solutions, and abandons a partial solution **s** (i.e. backtracks) as soon as **s** is found to be impossible to form a valid complete solution.
- We can use a stack to store the current partial solution (e.g. a subset of eight queens, or a partial path in a maze). The stack allows us to go back (backtrack) to a previous state, and proceed to build the next partial solution.

Searching With a Stack

- For example, for the **Four Queens** Problem:
    - **We know that each row holds 1 and only 1 queen.**
    - Place the first queen at (1, 1), push it to stack.
    - Place the second queen at (2, 1). Is there a conflict with the first queen? If so, move the second queen to (2, 2), (2, 3) and so on until there is no conflict. Push it to stack.
    - Place the third queen at (3, i) where i is from 1 to 4, until there is no conflict with the previous queens. Push it to stack.
    - If you can’t find a valid spot for a queen (say, the third queen), pop the stack (thus you are back to working on the second queen), search for the **next** valid spot for it and push to stack.