

# VE280 Programming and Elementary Data Structures

Paul Weng  
UM-SJTU Joint Institute

## Queue



# Learning Objectives

- Understand what is a queue
- Know how to implement it
- Discover some applications of queue
- Understand what is a deque

# Outline

- Queue
  - Definition
  - Implementation
  - Applications
- Relative: Deque

# Queue

- A “line” of items in which the **first** item inserted into the queue is the **first** one out.
  - Restricted form of a linear list: insert at **one end** and remove from **the other**.
  - FIFO access: first in, first out.

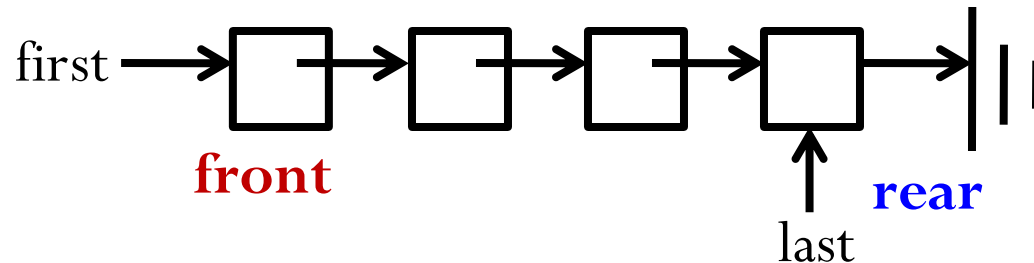


# Methods of Queue

- **size()** : number of elements in the queue.
- **isEmpty()** : check if queue has no elements.
- **enqueue(Object o)** : add object **o** to the **rear** of the queue.
- **dequeue()** : remove the **front** object of the queue if not empty; otherwise, throw **queueEmpty**.
- **Object &front()** : return a reference to the front element of the queue.
- **Object &rear()** : return a reference to the rear element of the queue.

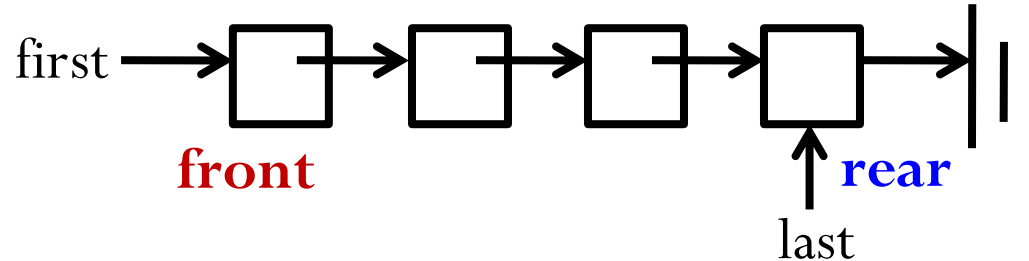
# Queues Using Linked Lists

- Which type of linked list should we choose?
  - We need fast **enqueue** and **dequeue** operations.
- Double-ended singly-linked list is sufficient!



- **enqueue**(Object o) : append object at the end  
`LinkedList::insertLast(Object o) ;`
- **dequeue**() : remove the first node  
`LinkedList::removeFirst() ;`

# Queues Using Linked Lists



- **size() : LinkedList :: size() ;**
- **isEmpty() : LinkedList :: isEmpty() ;**
- **Object &front() :** return a reference to the object stored in the first node.
- **Object &rear() :** return a reference to the object stored in the last node.

# Queues Using Arrays

**Array [MAXSIZE] :**

2	3	1	4			
---	---	---	---	--	--	--

  
front   rear

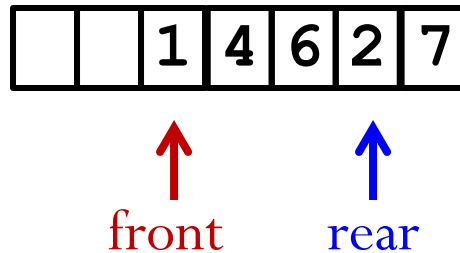
- If we stick to the requirement that the n elements of a queue are the beginning n elements of the array,
  - How many operations for **enqueue**?
    - I.e., independent of n (number of elements) or proportional to n?
  - How many operations for **dequeue**?
- A better way is to let the elements “**drift**” within the array.

enqueue(6);  
dequeue();  
dequeue();

2	3	1	4	6		
---	---	---	---	---	--	--



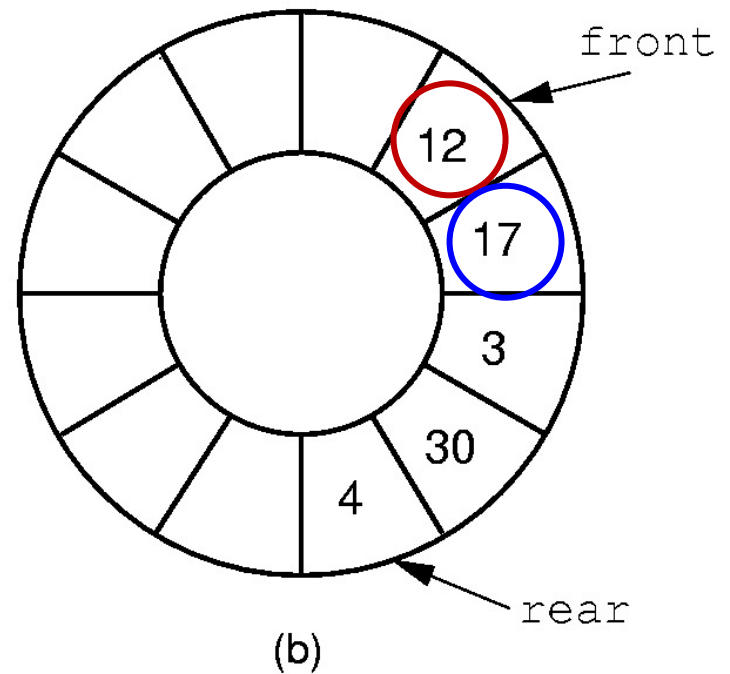
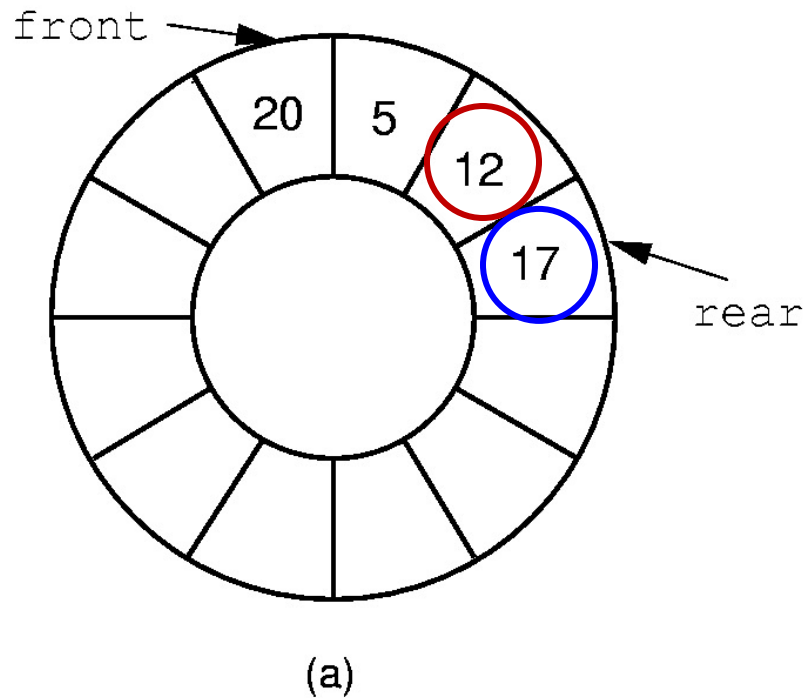
# Queues Using Arrays



- We maintain two integers to indicate the front and the rear of the queue.
- However, as items are added and removed, the queue “drifts” toward the end.
  - Eventually, there will be no space to the right of the queue, even though there is space in the array.

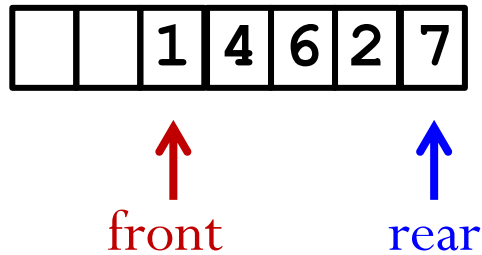
# Queues Using Arrays

- To solve the problem of memory waste, we use a **circular array**.

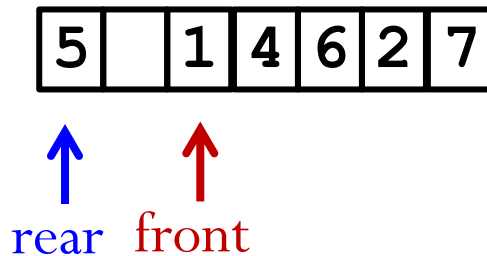


# Circular Arrays

- We can implement a circular array using a plain linear array:
  - When front/rear equals the **last** index (i.e., MAXSIZE-1), increment of front/rear gives the **first** index (i.e., 0).



enqueue (5)



# Circular Arrays

- To realize the “circular” increment, we can use modulo operation:

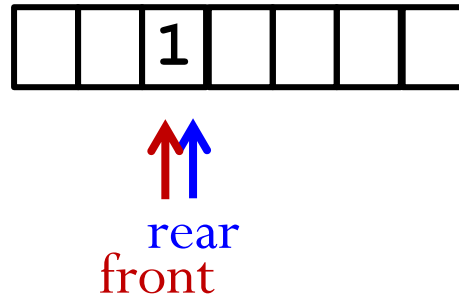
```
front = (front+1) % MAXSIZE;
```

```
rear = (rear+1) % MAXSIZE;
```

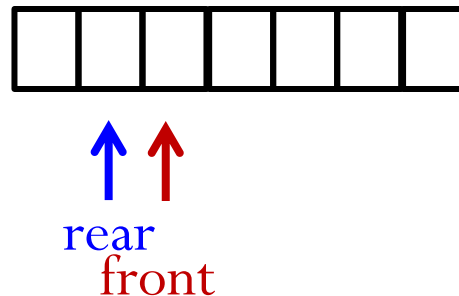
If **front (or rear) == MAXSIZE-1**, the statement sets **front (or rear)** to 0.

# Boundary Conditions

- Suppose that **front** points to the **first** element in the queue and that **rear** points to the **last** element in the queue.
- What will a queue with one element look like?

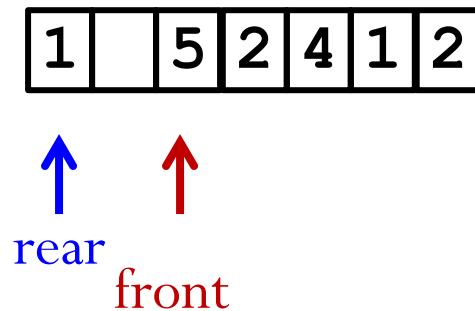


- What will an empty queue look like?

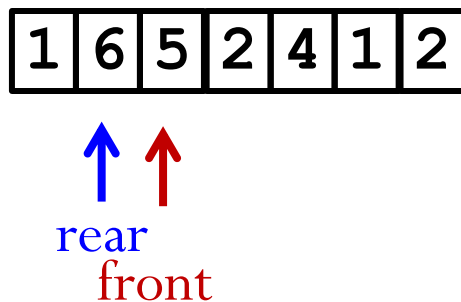


# Boundary Conditions

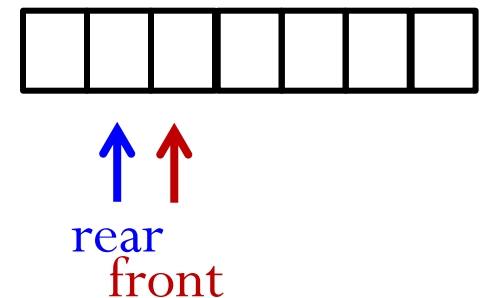
- What will a queue with one empty slot look like?



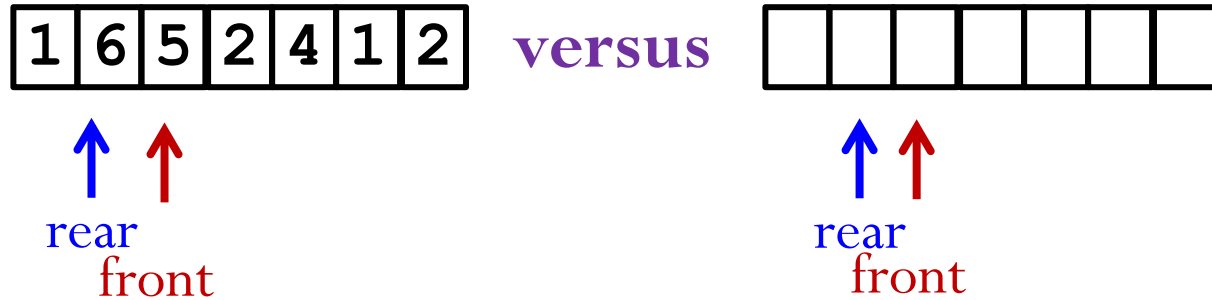
- What will a full queue look like?



versus an  
empty queue



# Boundary Conditions



- To distinguish between the full array and the empty array, we need a flag indicating **empty** or **full**, or a **count** on the number of elements in the queue.

# Queues Using Arrays

- **enqueue(Object o)** : increment **rear**, wrapping to the beginning of the array if the end of the array is reached; if **rear** becomes **front**, reallocate arrays. Insert **o** at the position of **rear**
- **dequeue()** : if empty, throw **queueEmpty**; increment **front**, wrapping to the beginning of the array if the end of the array is reached.
- **isEmpty()** : **return (count == 0) ;**
- **size()** : **return count;**





# Which statements are true?

Select all the correct answers.

- **A.** A linked list requires less memory than a circular array when  $\text{size} = \text{capacity}$ .
- **B.** A linked list requires more memory than a circular array when  $\text{size} = \text{capacity}$ .
- **C.** An implementation with a linked list is computationally more efficient.
- **D.** An implementation with a circular array is computationally more efficient.



# Outline

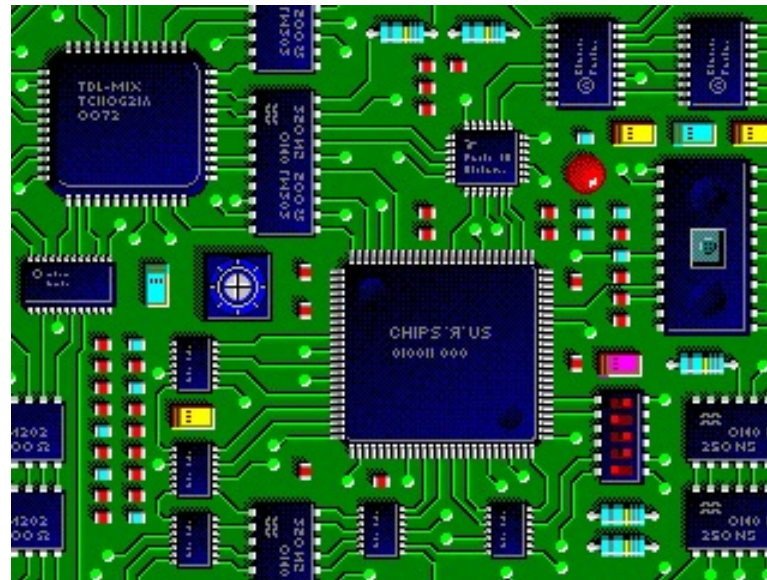
- Queue
  - Implementation
  - Applications
  - Relative: Deque

# Application of Queues

- Request queue of a web server
  - Each user can send a request.
  - The arriving requests are stored in a **queue** and processed by the computer in a **first-come-first-serve** way.

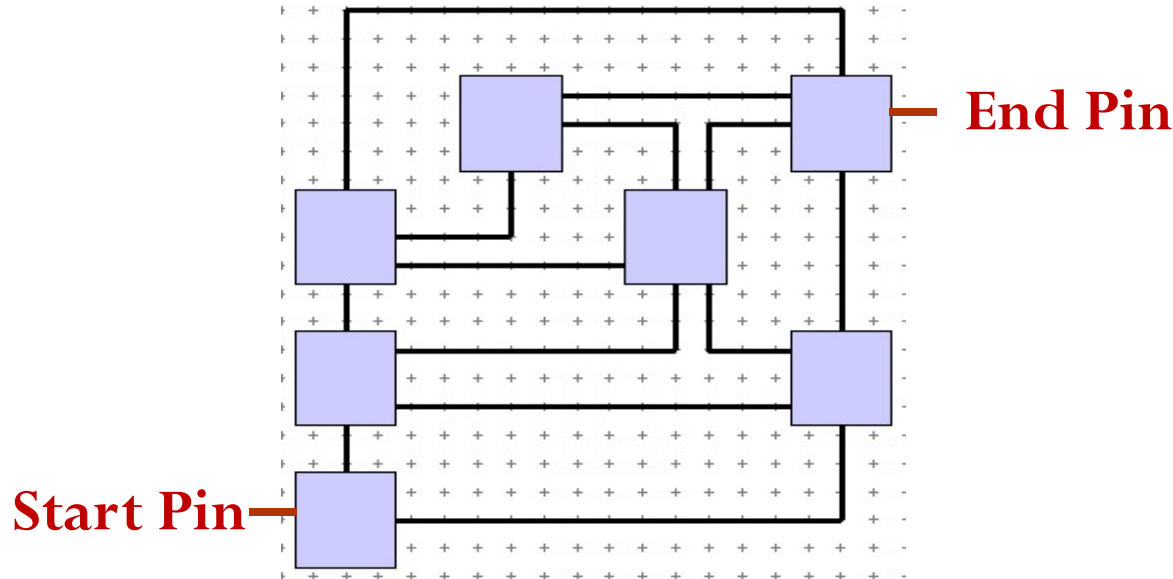
# Application of Queue: Wire Routing

- Select paths to connect all pairs of pins that need to be connected together.
- An important problem in **electronic design automation**.



# A Simplified Problem

- Condition: We have all blocks laid on the chip. We also have some of the wires routed.
- Problem: We want to connect the next pair of pins.
- Constraint: we can only draw wires **horizontally** or **vertically**.



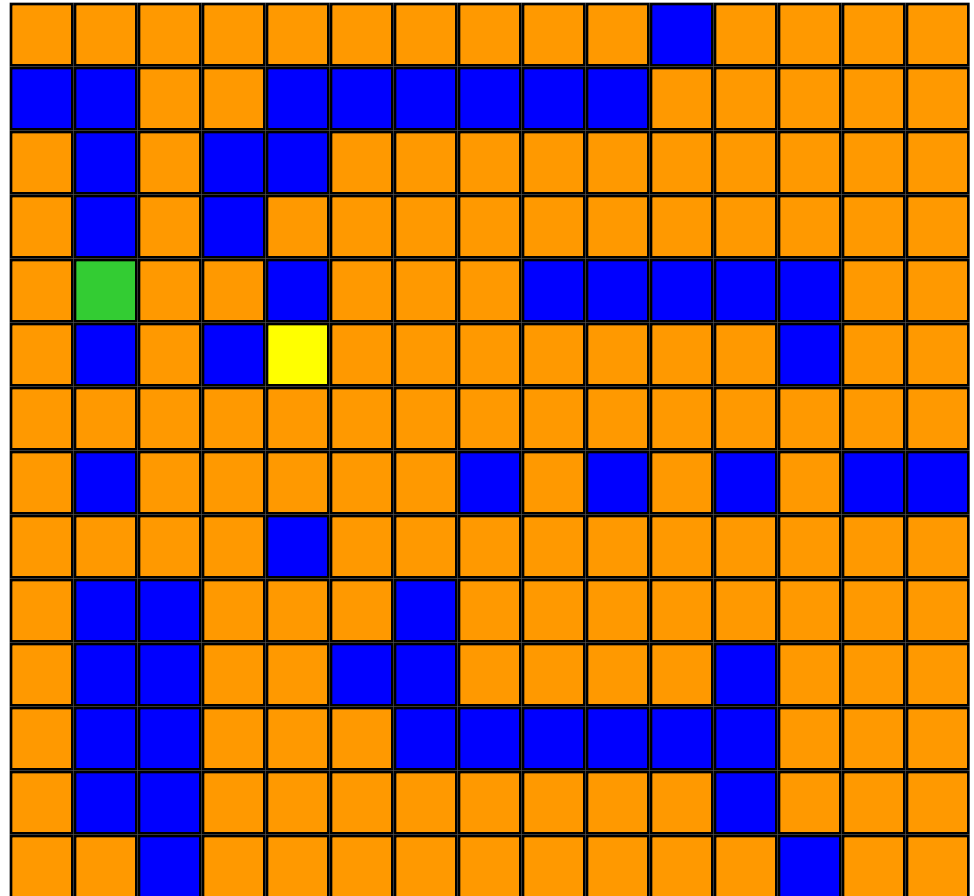
# Modeling as a Grid

 Start Pin

 End Pin

- Blue squares are **blocked** squares.
- Orange squares are **available** to route a wire.

How to find a path from the start pin to the end pin?




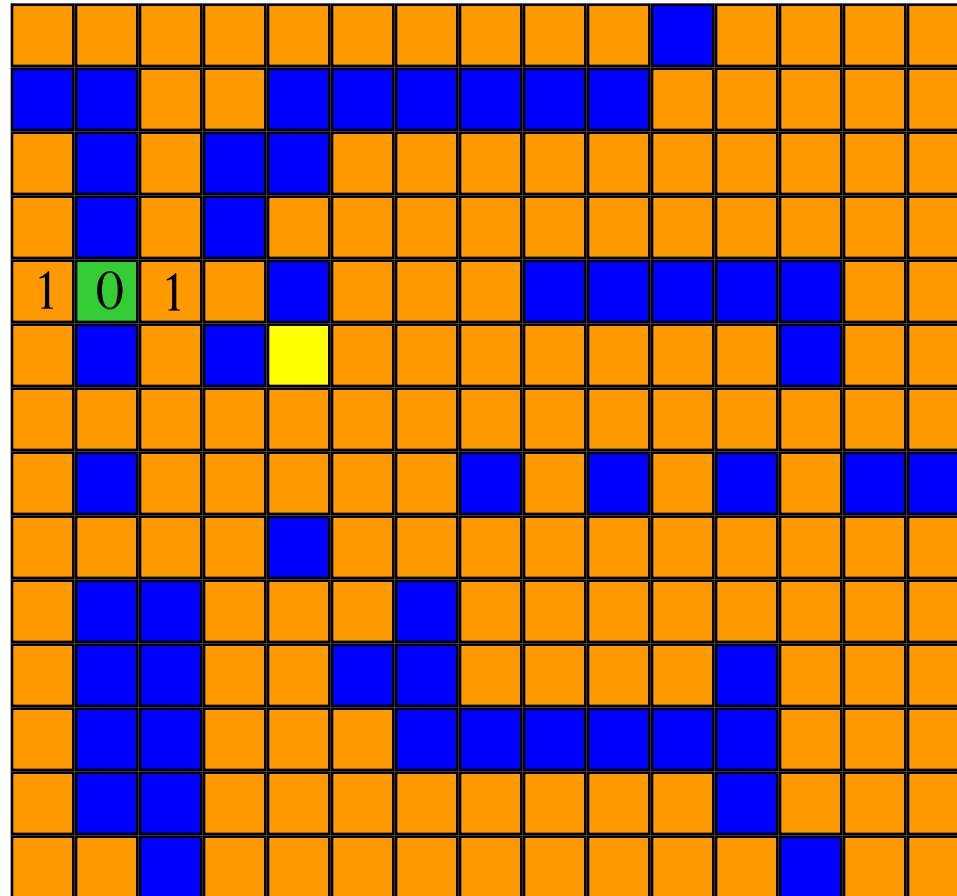
# Wire Routing: Lee's Algorithm

- A **queue** of reachable squares from the start pin is used.
- The cell of the start pin is set with a distance value of 0.
- It is enqueued into an initial empty queue.
- **While** the queue is not empty.
  - A cell is **dequeued** from the queue and made the **examine cell**.
  - Is the examine cell the end pin? If yes, path found and return.
  - Otherwise, all **unreached unblocked** squares adjacent to the **examine cell** are marked with their distance (this is 1 more than the distance value of the **examine cell**) and **enqueued**.
- When queue becomes empty but end pin has not been reached yet, this means no path found.

# Illustration of Lee's Algorithm

 start pin

 end pin




Expand "0"

queue: 0

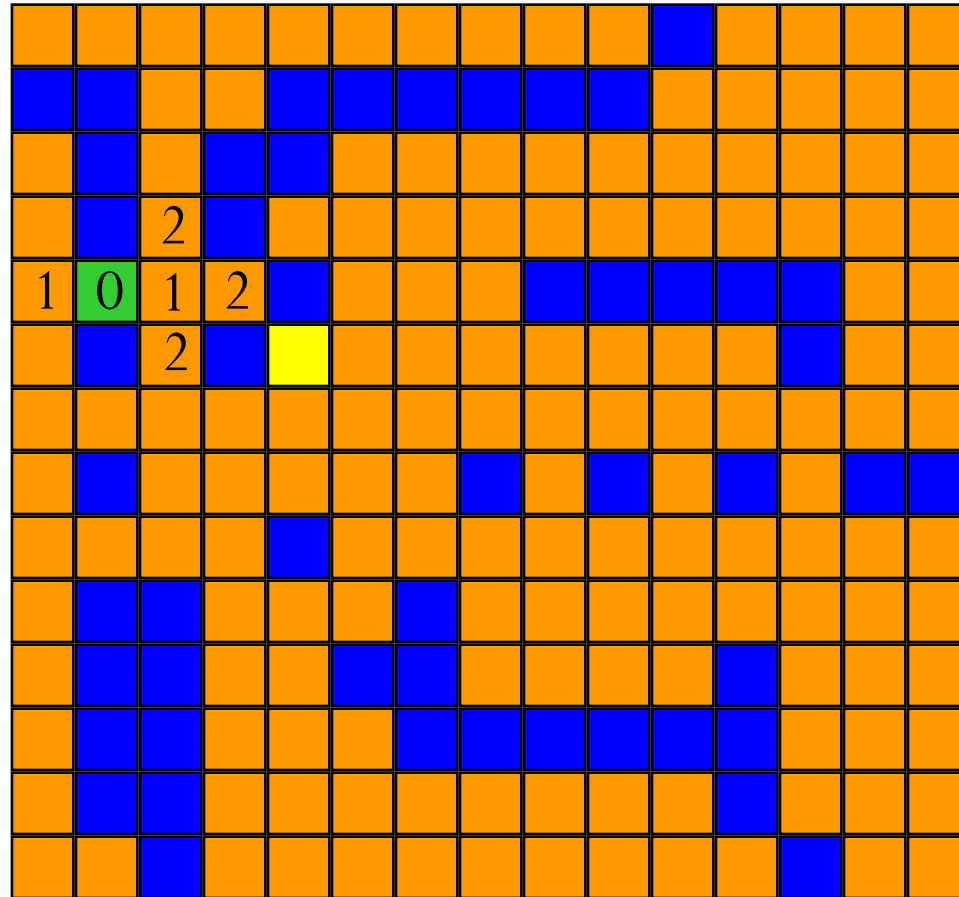


# Illustration of Lee's Algorithm

 start pin

 end pin


Expand right  
“1”



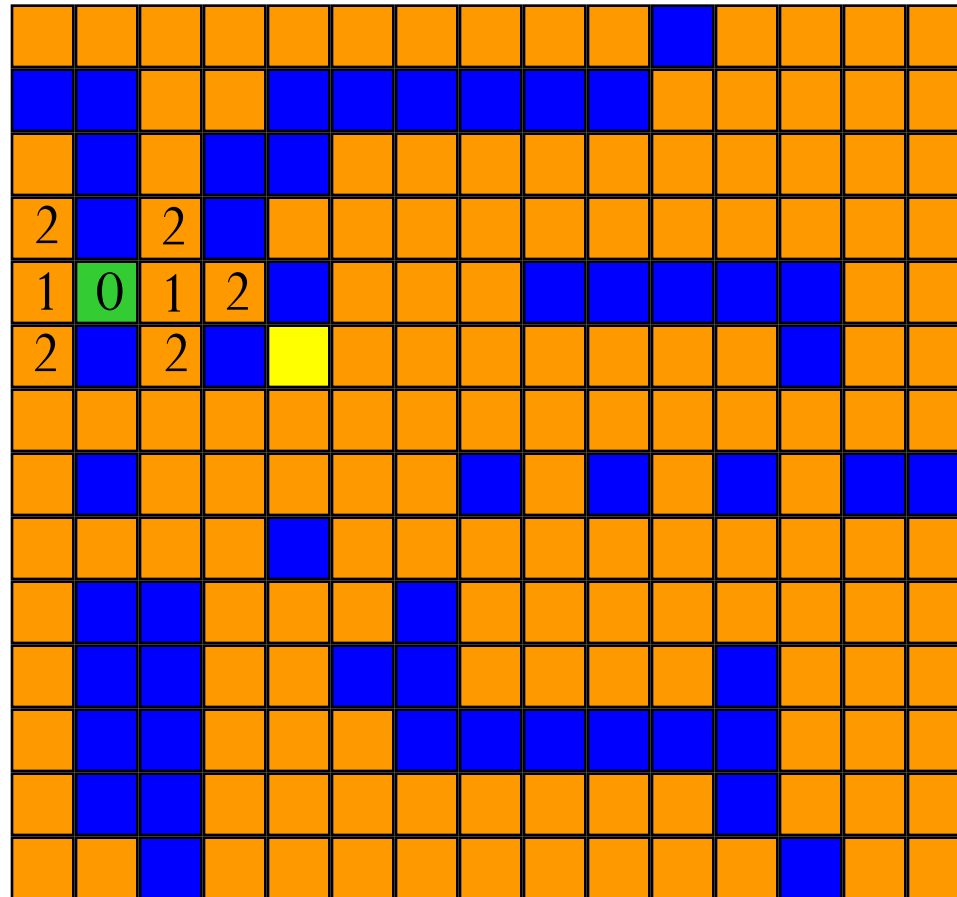
queue: 1, 1

# Illustration of Lee's Algorithm

 start pin

 end pin

Expand left  
“1”

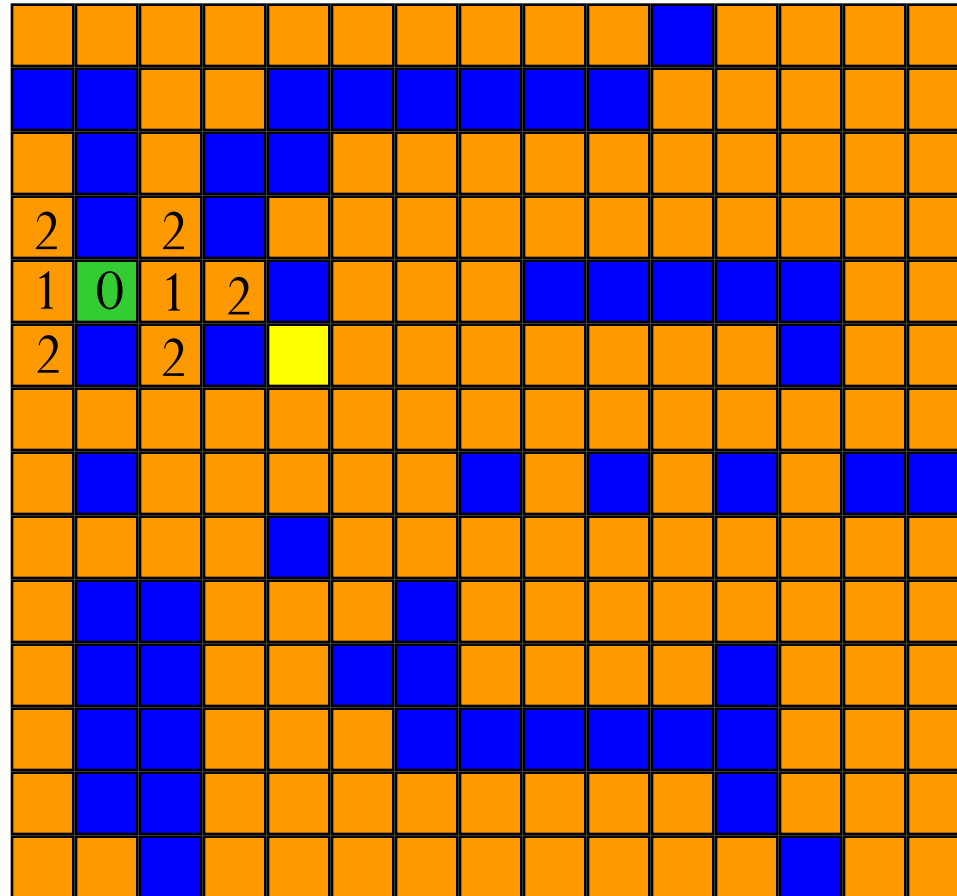


queue: 1,2,2,2

# Illustration of Lee's Algorithm



 start pin

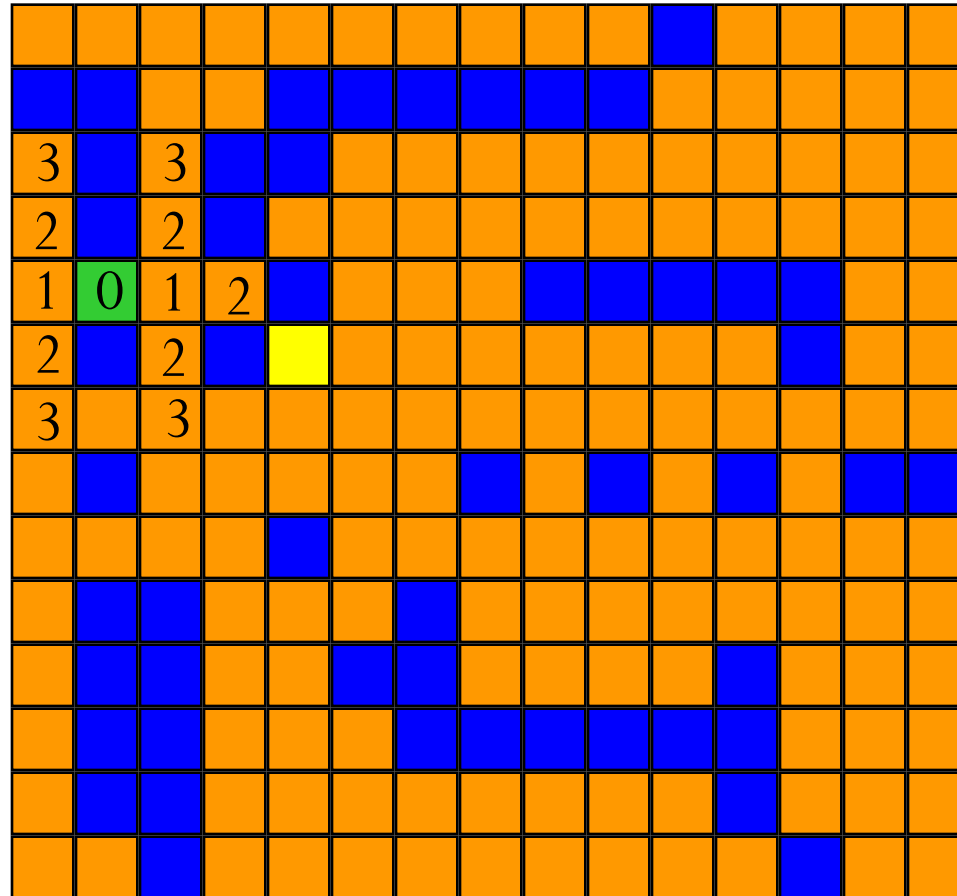
 end pin



Expand and reach all squares 3 units from start.



# Illustration of Lee's Algorithm

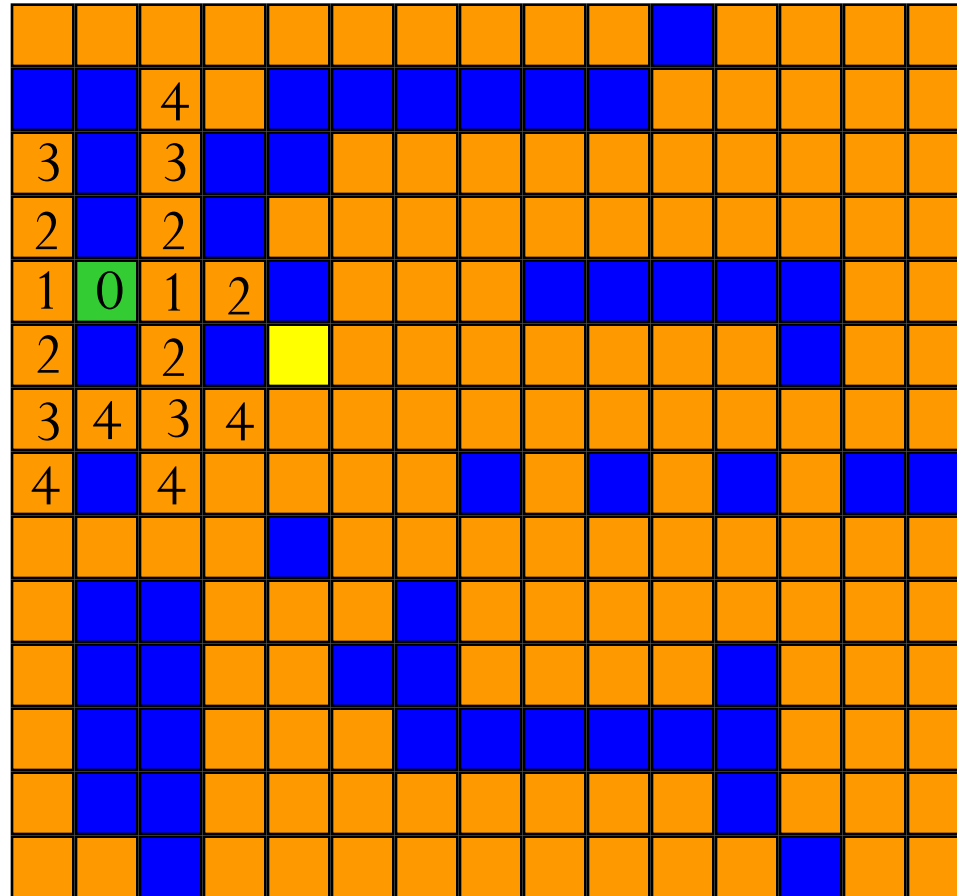
 start pin  
 end pin



Expand and reach all squares 4 units from start.



# Illustration of Lee's Algorithm

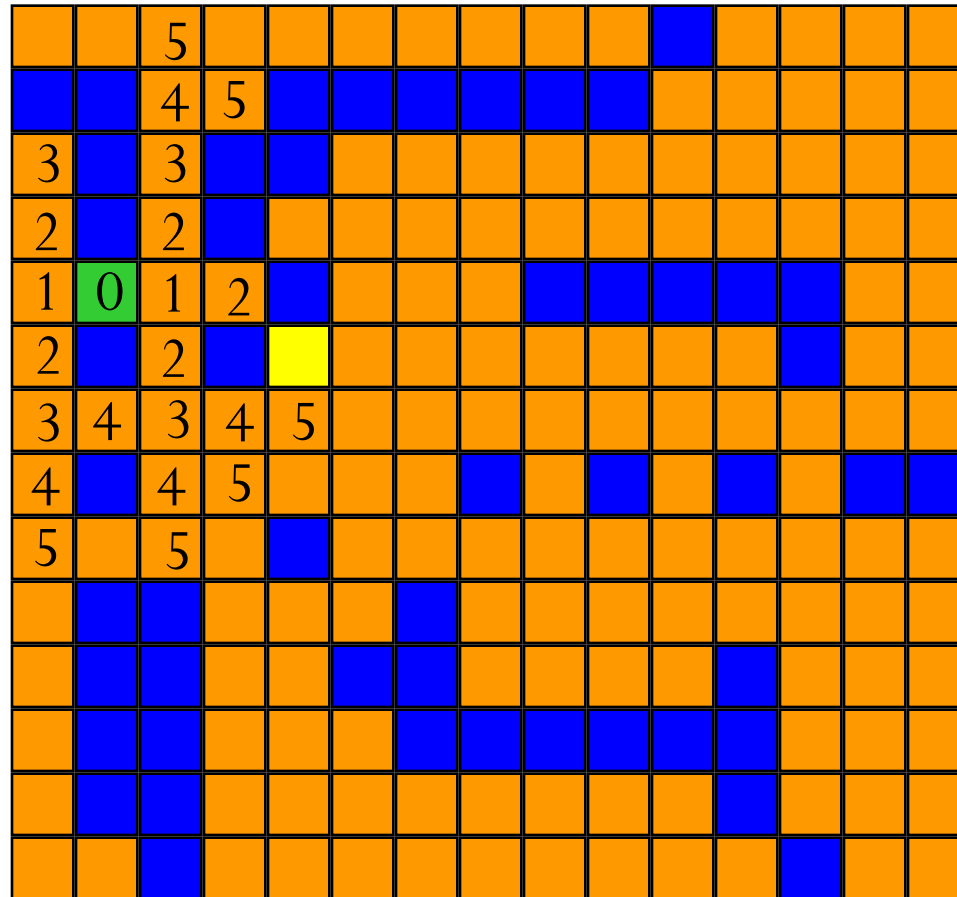
 start pin  
 end pin



Expand and reach all squares 5 units from start.



# Illustration of Lee's Algorithm

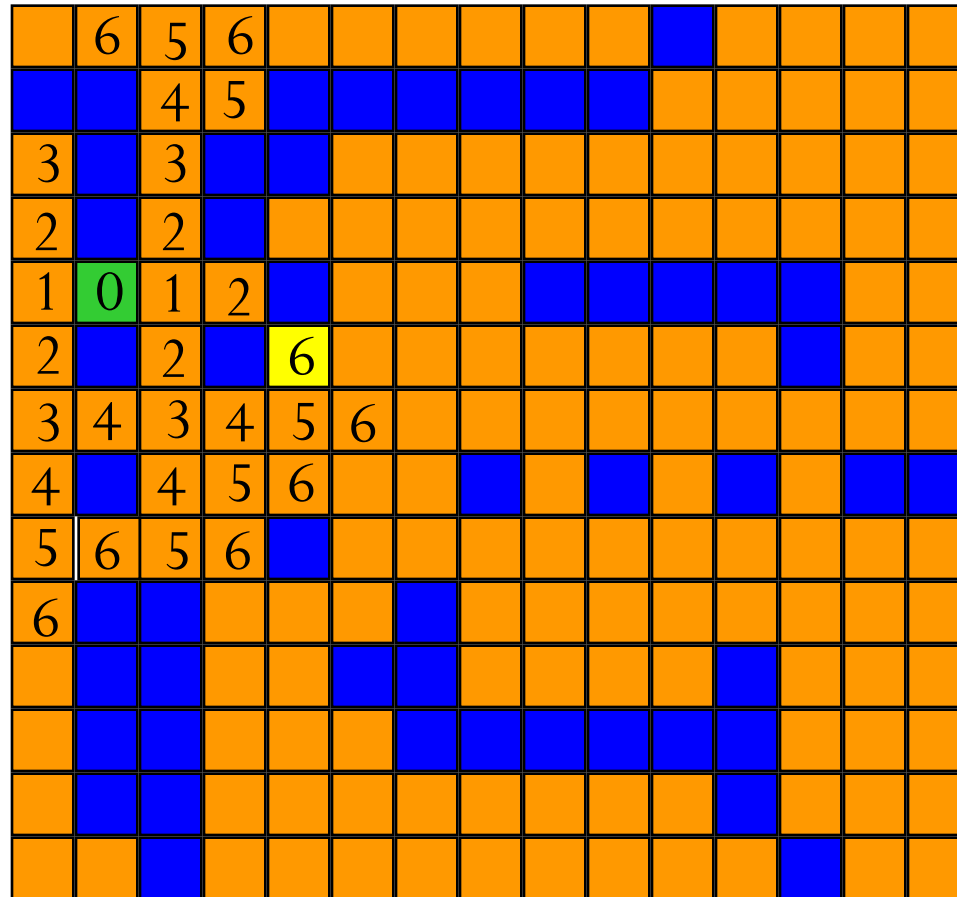
 start pin  
 end pin



Expand and reach all squares 6 units from start.



# Illustration of Lee's Algorithm

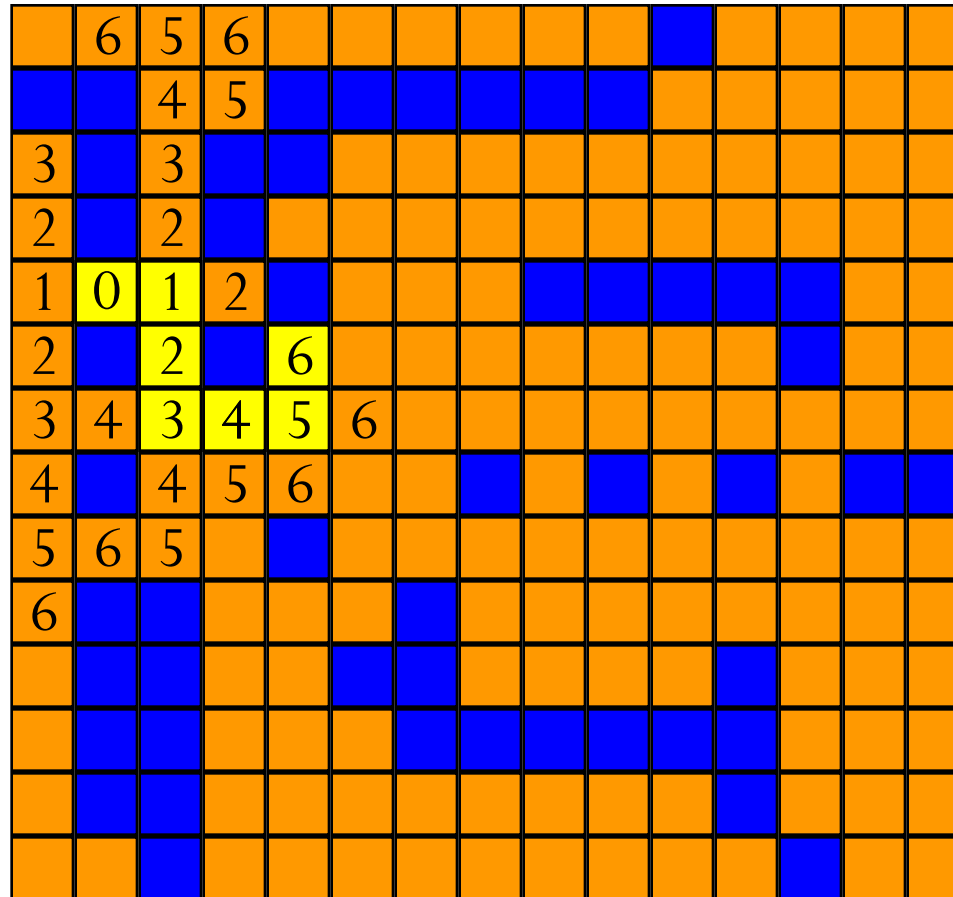
 start pin  
 end pin



End pin reached. Trace back.

# Illustration of Lee's Algorithm

 start pin  
 end pin







# A queue can be used:

Select all the correct answers.

- **A.** to handle printing jobs on a printer.
- **B.** to implement a buffer.
- **C.** to implement a waiting list.
- **D.** to share a CPU among different processes.



# Outline

- Queue
  - Definition
  - Implementation
  - Applications
- Relative: Deque

# Deque

- Not a proper English word, pronounced as “deck”.
  - Means double-ended queue
- A combination of stack and queue.
  - Items can be inserted and removed from **both ends** of the list.
- Methods supported:
  - `push_front(Object o)`
  - `push_back(Object o)`
  - `pop_front()`
  - `pop_back()`

# Deque Implementation

- Linked list
  - Which type of linked list will you choose to support fast insertion and removal?
  - Double-ended doubly-linked list
- Circular array
  - front and rear not only need to be incremented (**push\_back**, **pop\_front**), but also need to be decremented (**push\_front**, **pop\_back**).

# Reference

- **Problem Solving with C++ (8<sup>th</sup> Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 13.2 **Queue**