VE281

Data Structures and Algorithms

Queues

Review

- Generic programming
  - Data-type independent way of programming
- Iterators for containers
- Stacks
  - Methods: **`push`**, **`pop`**, **`size`**, etc.
  - Implementations: arrays versus linked lists
  - Applications

Outline

- Queues

Queues

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

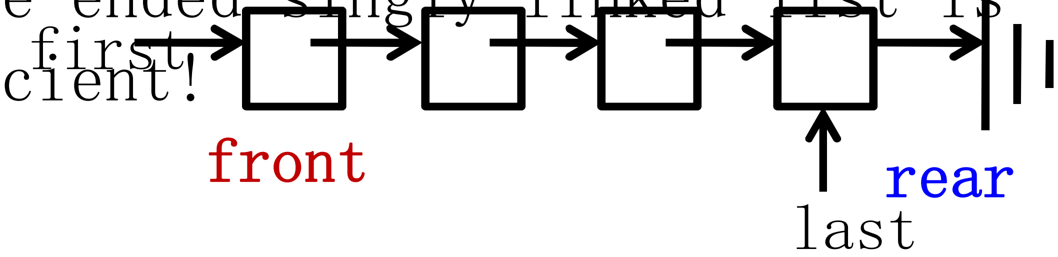
Methods of Queue

- **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

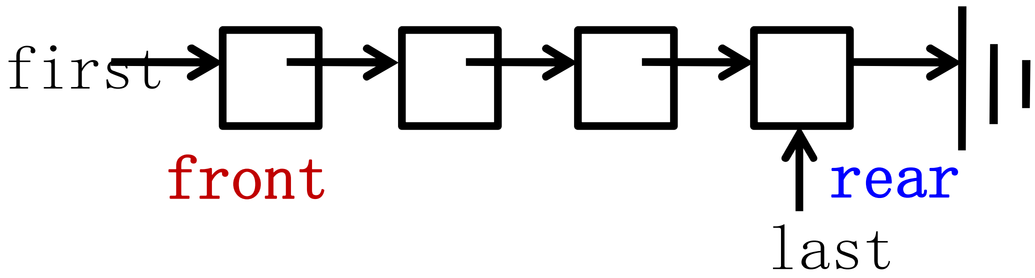
- Which type of linked list should we choose?
  - We need fast **enqueue** and **dequeue** operations.

- Double-ended singly-linked list is sufficient!



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

Queues Using Linked Lists



- **size(): return size;**
- **isEmpty(): return (size == 0);**
- **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,
  - what is the complexity of **enqueue**?
  - what is the complexity of **dequeue**?

- A better way is to let the elements "drift" within the array.

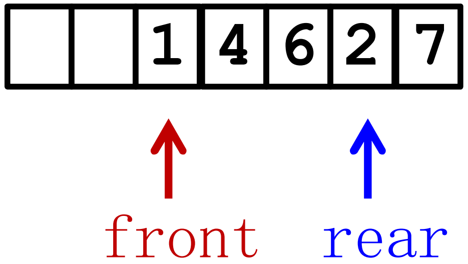
2	3	1	4	6		

enqueue(6);

dequeue();

dequeue();

Queues Using Arrays

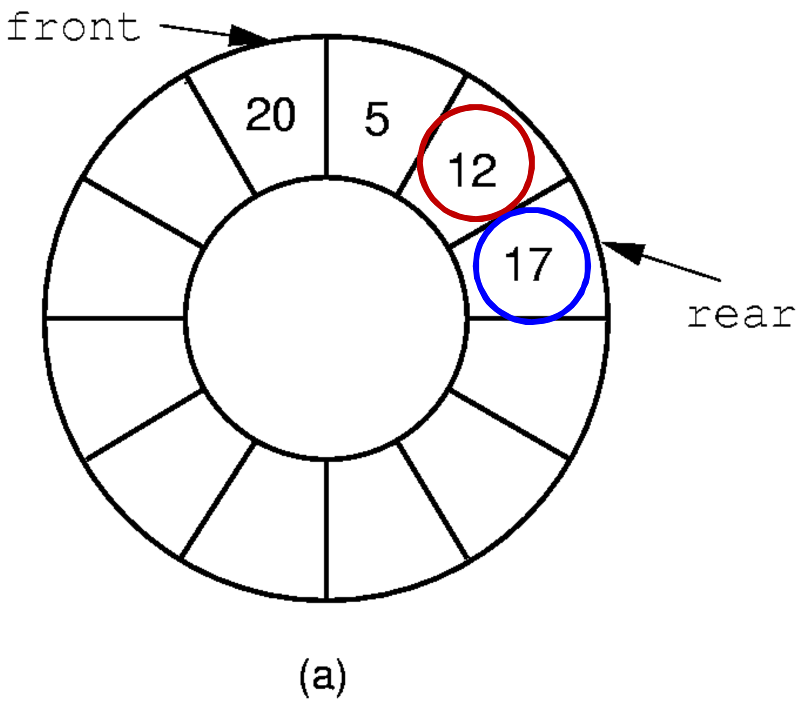


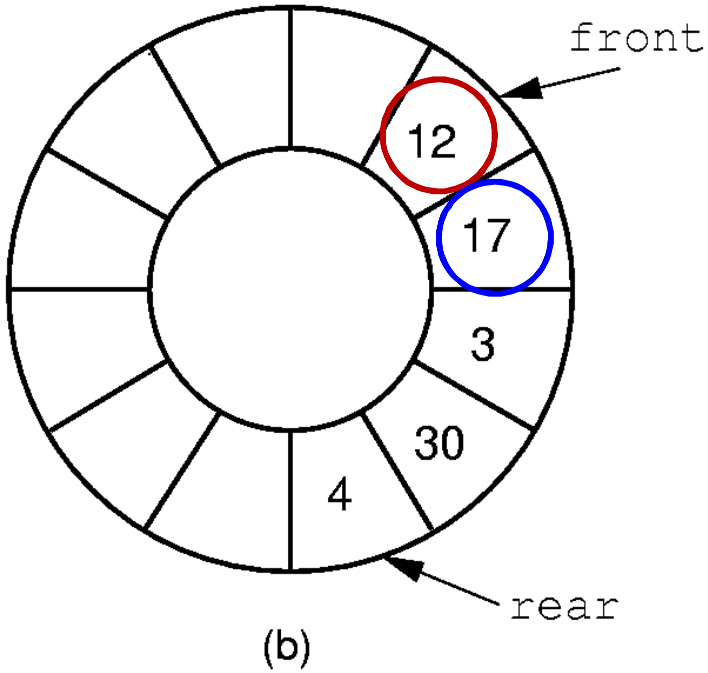
- We maintain two integers to indicate the front and the rear of the queue.

- 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

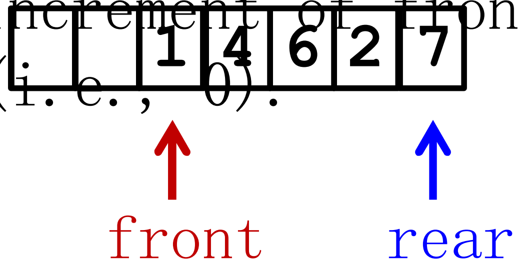
- To solve the problem of memory waste, we use a **circular array**.



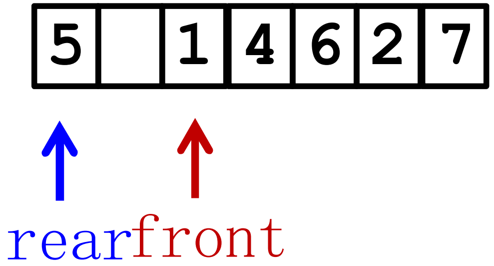


Circular Arrays

- 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

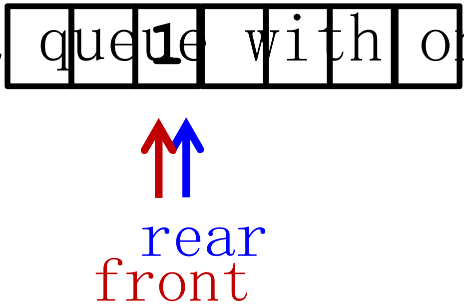
- To realize the "circular" increment, we can use modulo operation:

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

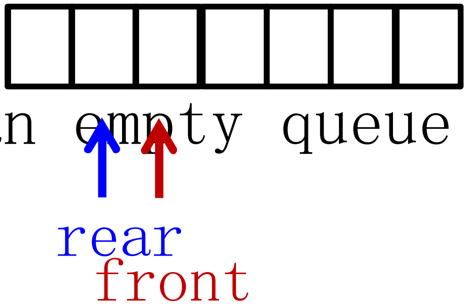
If **front == MAXSIZE-1**, the statement sets **front** to 0.

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?

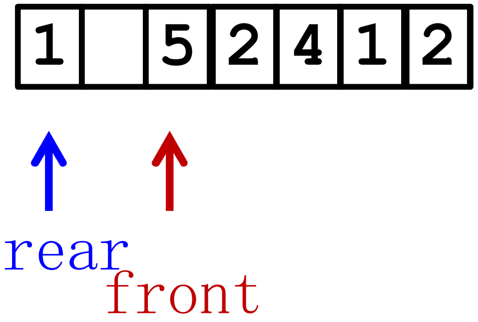


- What will an empty queue look like?

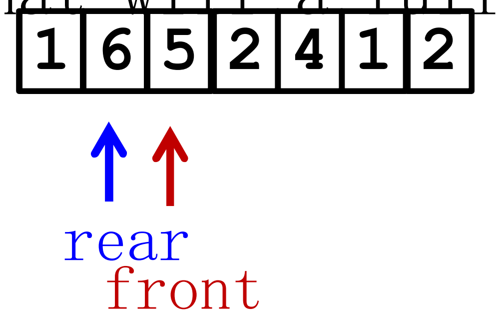


Boundary Conditions

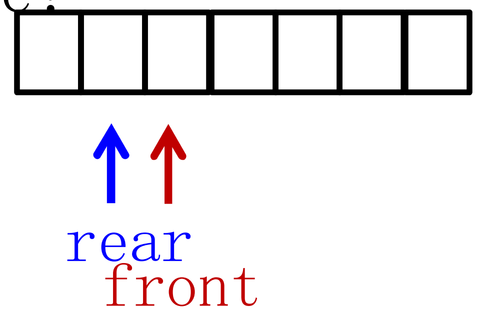
- What will a queue with one empty slot look like?



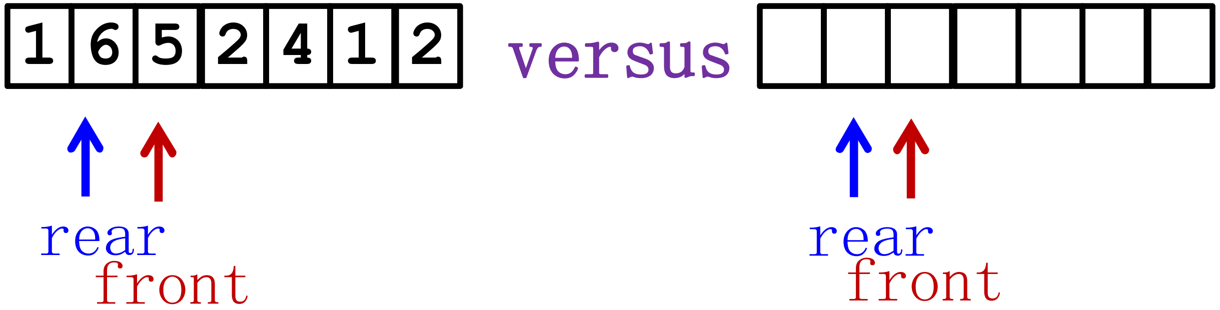
- What will a full queue look like?



versus an empty queue

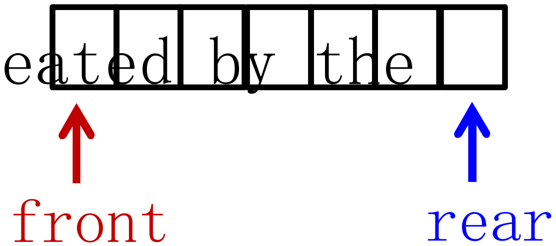


Boundary Conditions



- 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.

- Question: what will a queue created by the default constructor look like?



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.

- **dequeue():** increment **front**, wrapping to the beginning of the array if the end of the array is reached; if empty, throw **queueEmpty**.

- **isEmpty(): return (count == 0);**

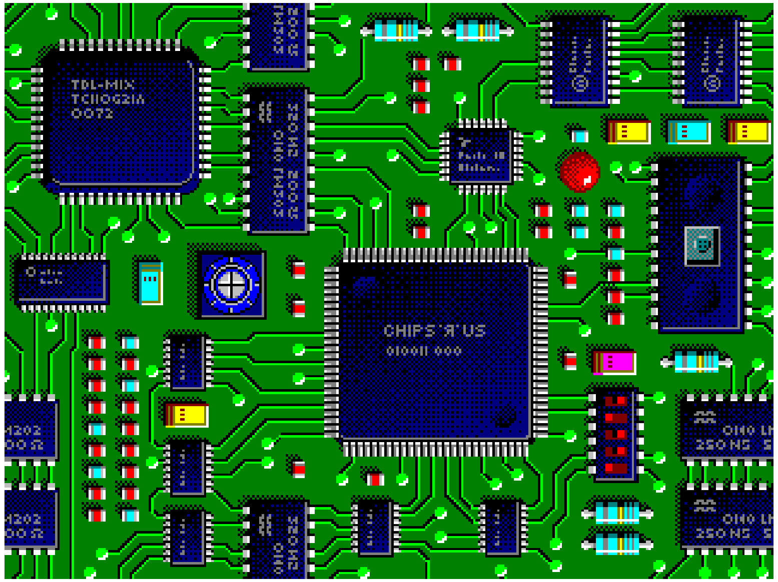
- **size(): return count;**

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.

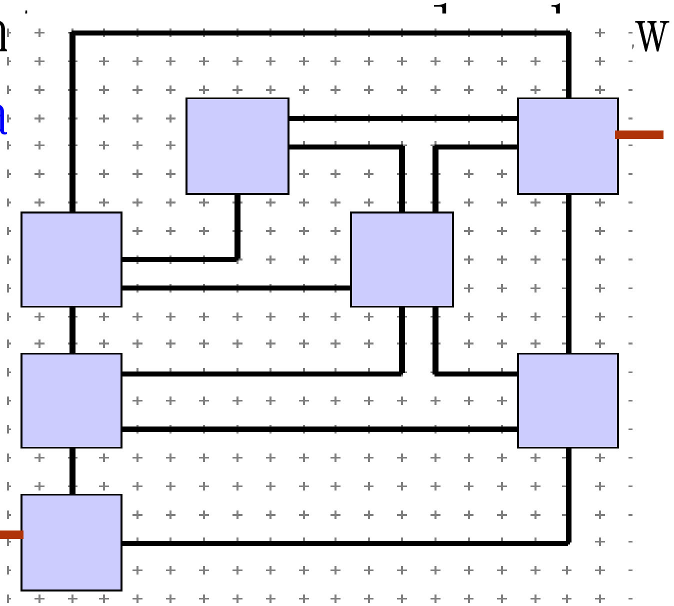
Application of Queue: Wire Routing

- 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.
- Constrain … w wires horizonta …



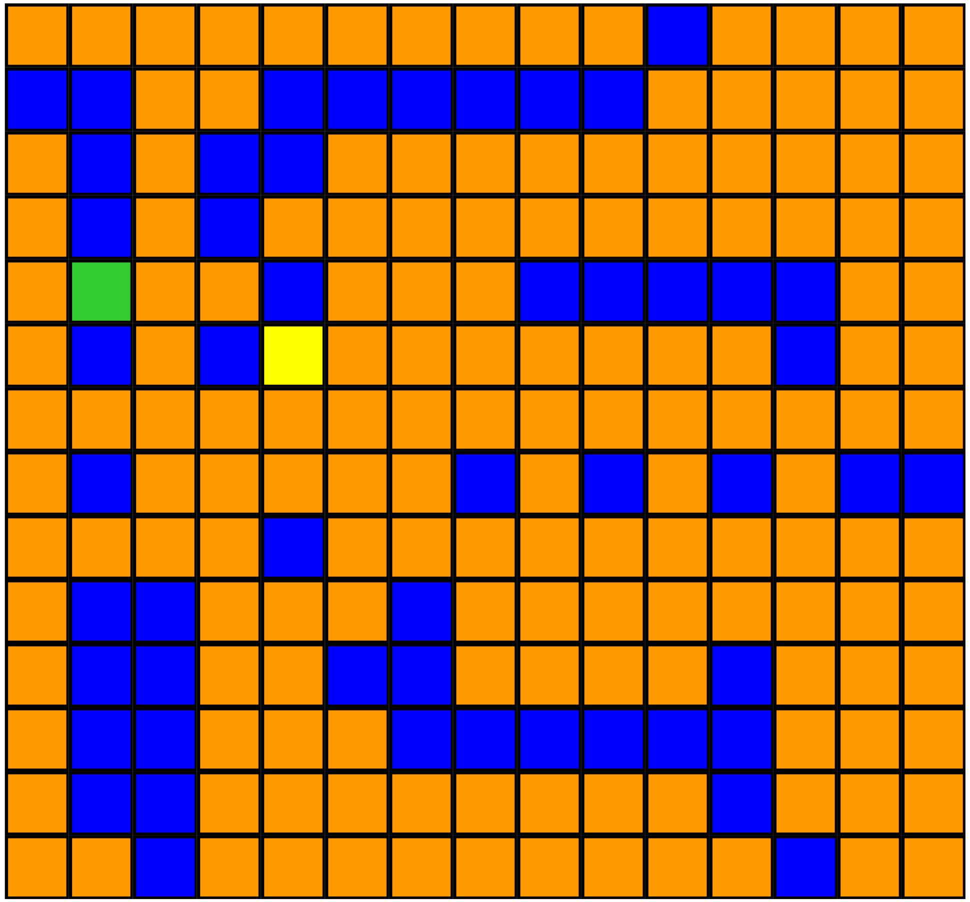
Modeling as a Grid

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 queue is initially empty, and the square of the start pin is the examine cell.
  - This cell has a distance value of 0.

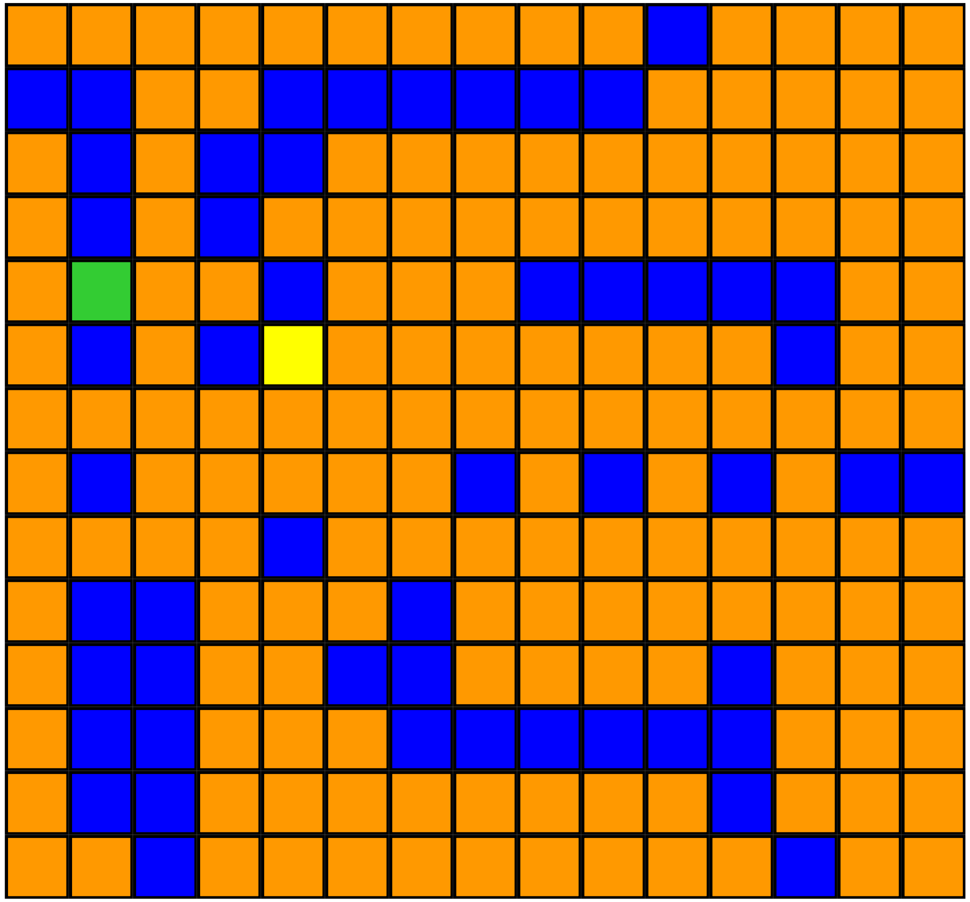
Loop

- 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 added to the queue.
- Then a cell is removed from the queue and made the new examine cell.
- This process is repeated until the end pin is reached (path found) or the queue becomes

Illustration of Lee's Algorithm

start pin

end pin

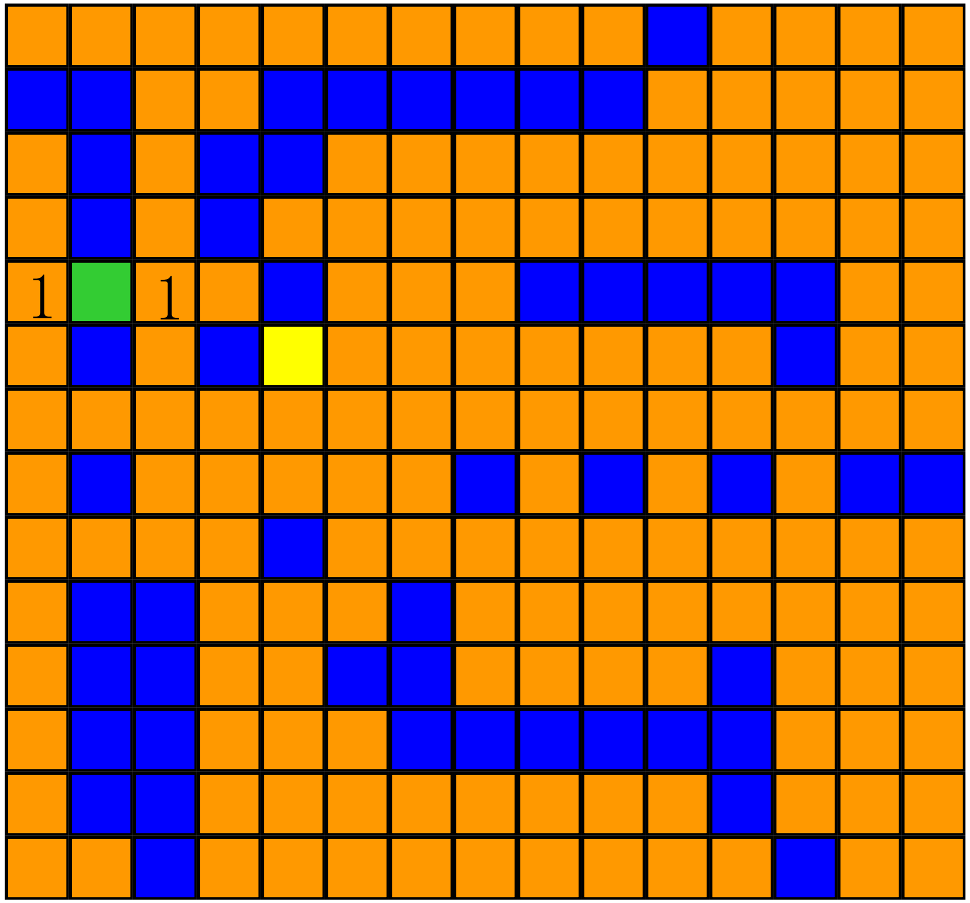


Label all reachable squares 1 unit from start.

Illustration of Lee’s Algorithm

start pin

end pin

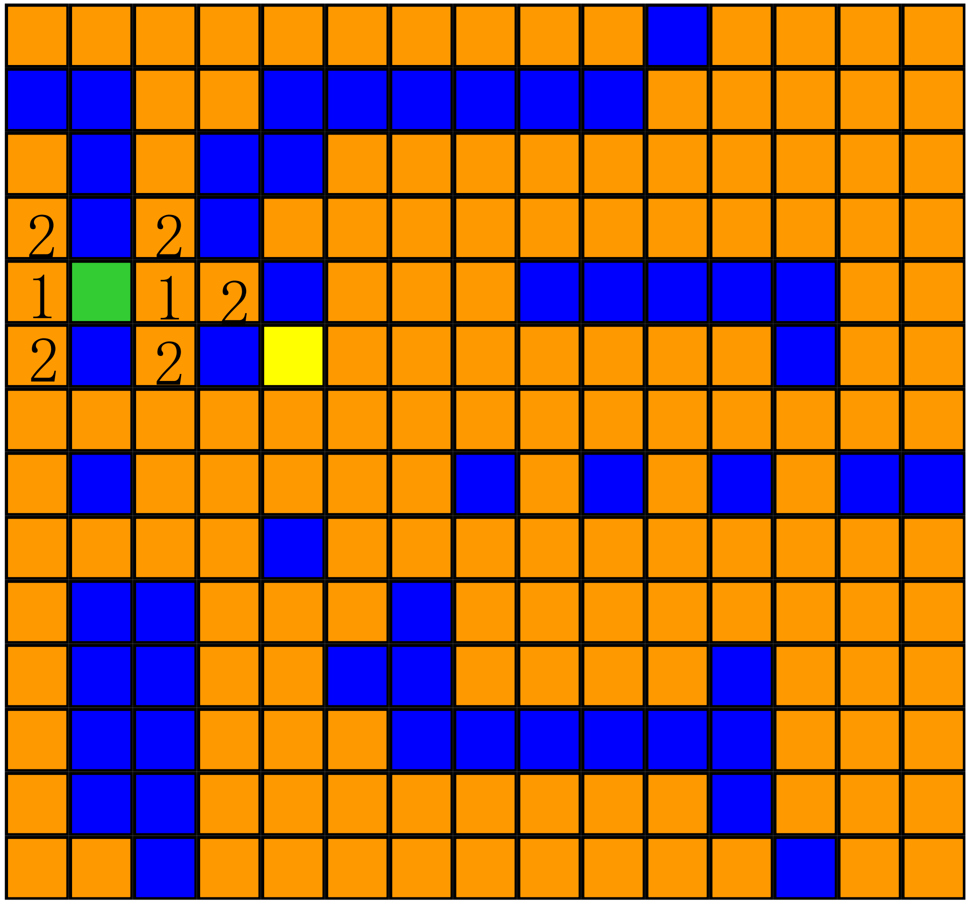


Label all reachable squares 2 unit from start.

Illustration of Lee’s Algorithm

start pin

end pin

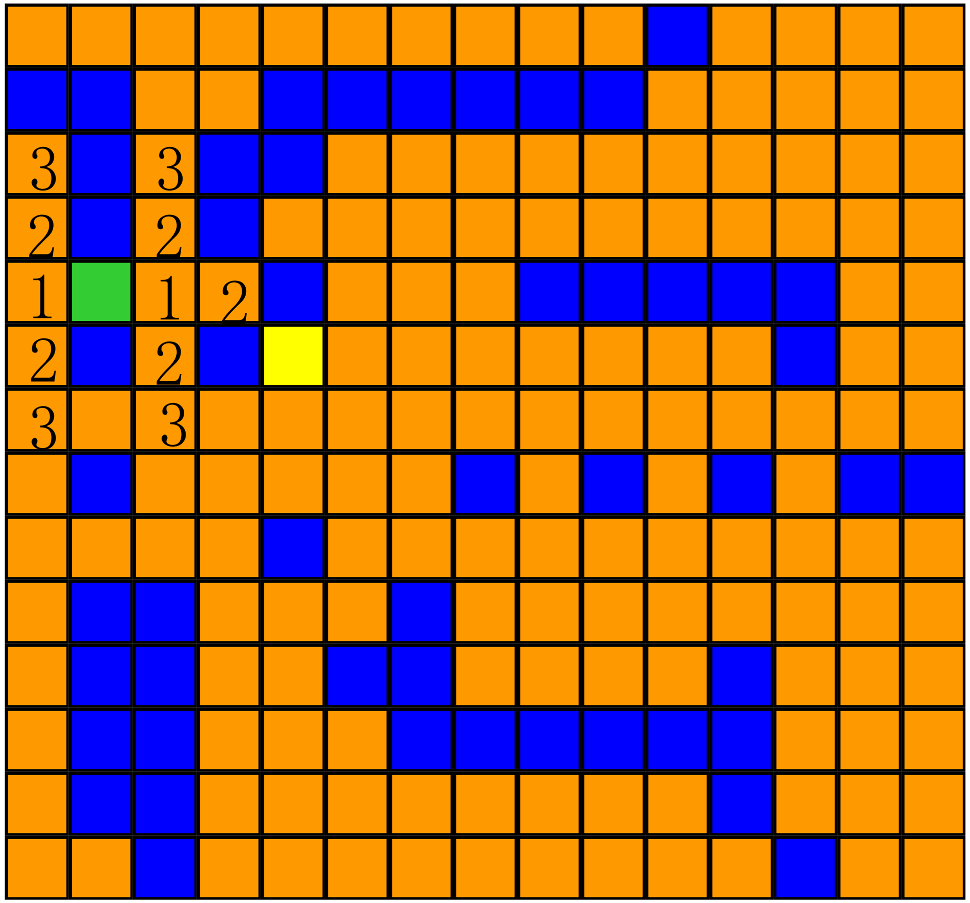


Label all reachable squares 3 unit from start.

Illustration of Lee’ s Algorithm

start pin

end pin

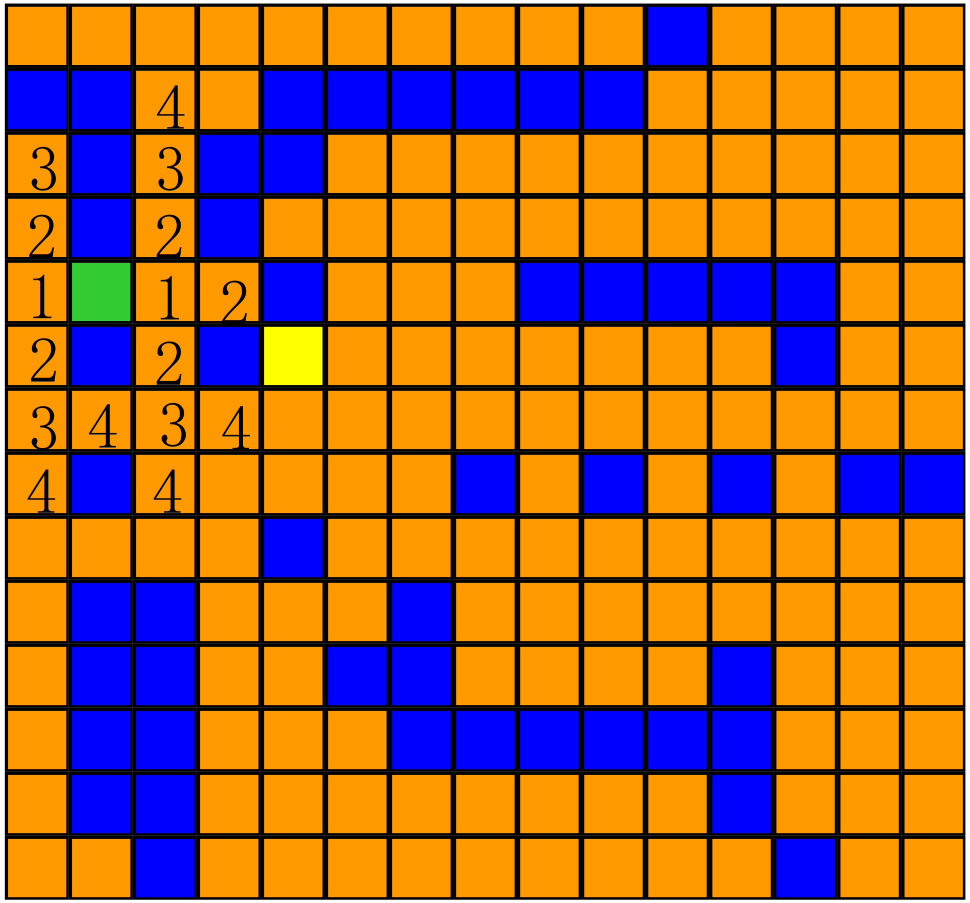


Label all reachable squares 4 unit from start.

Illustration of Lee's Algorithm

start pin

end pin

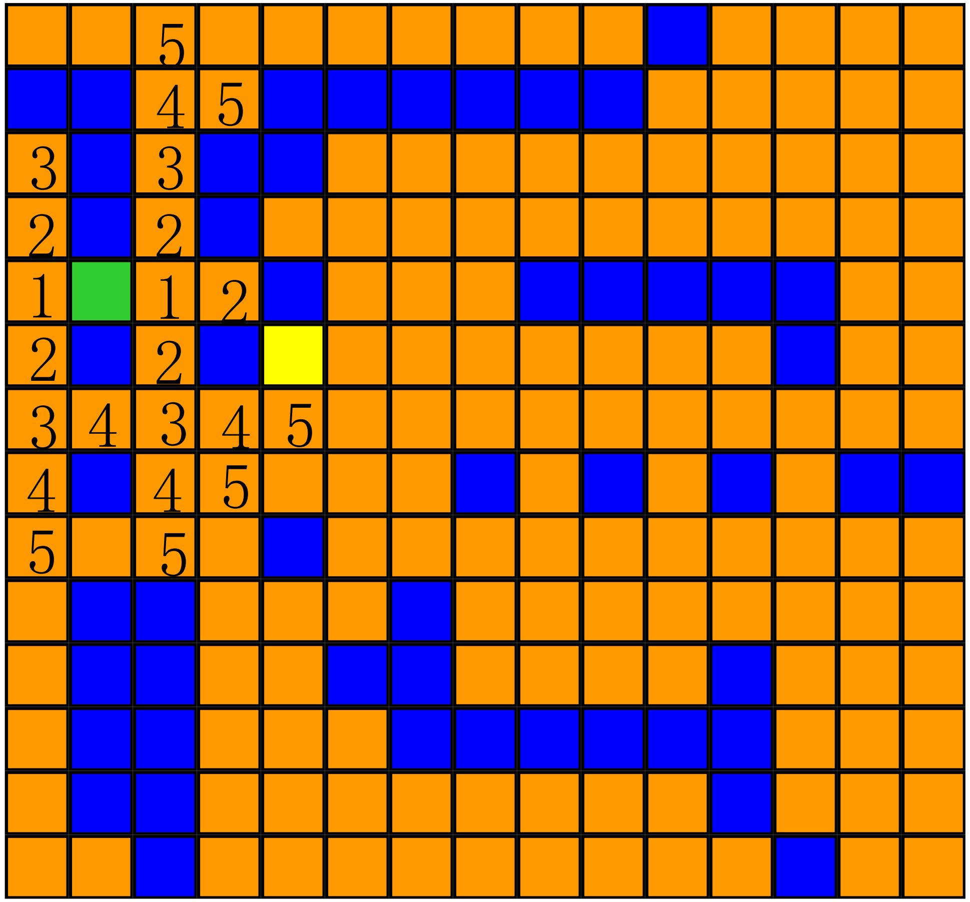


Label all reachable squares 5 unit from start.

Illustration of Lee’s Algorithm

start pin

end pin

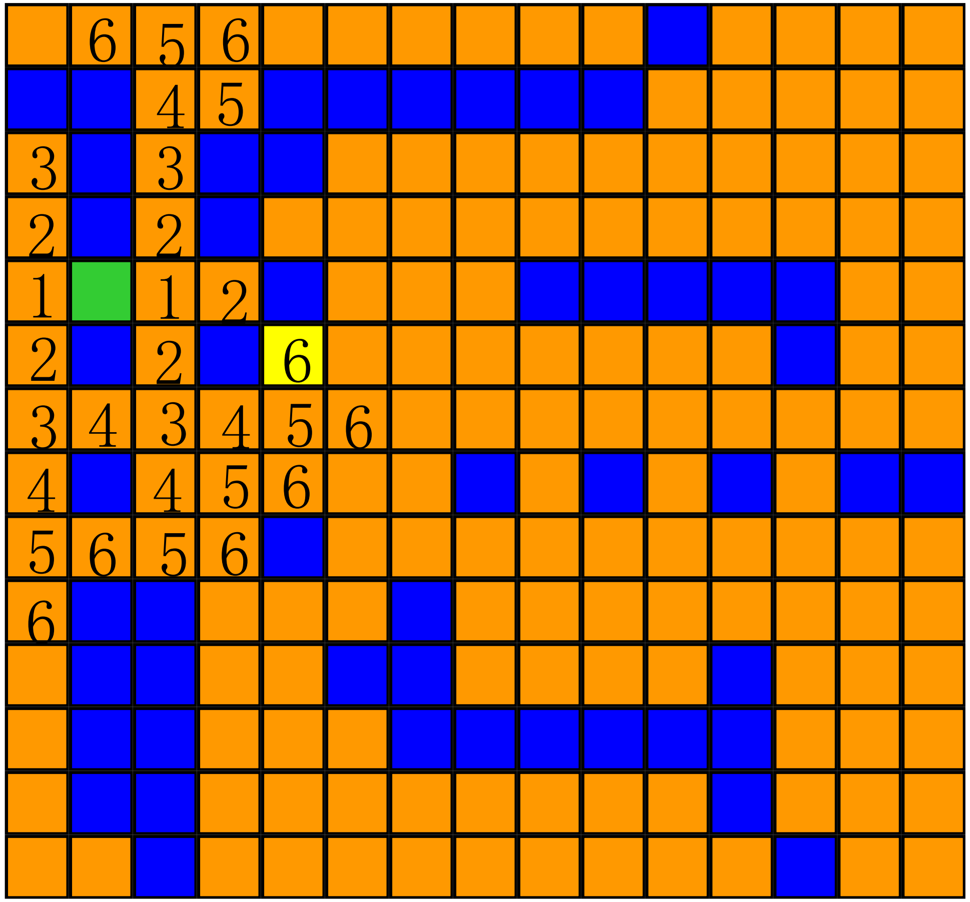


Label all reachable squares 6 unit from start.

Illustration of Lee’s Algorithm

start pin

end pin



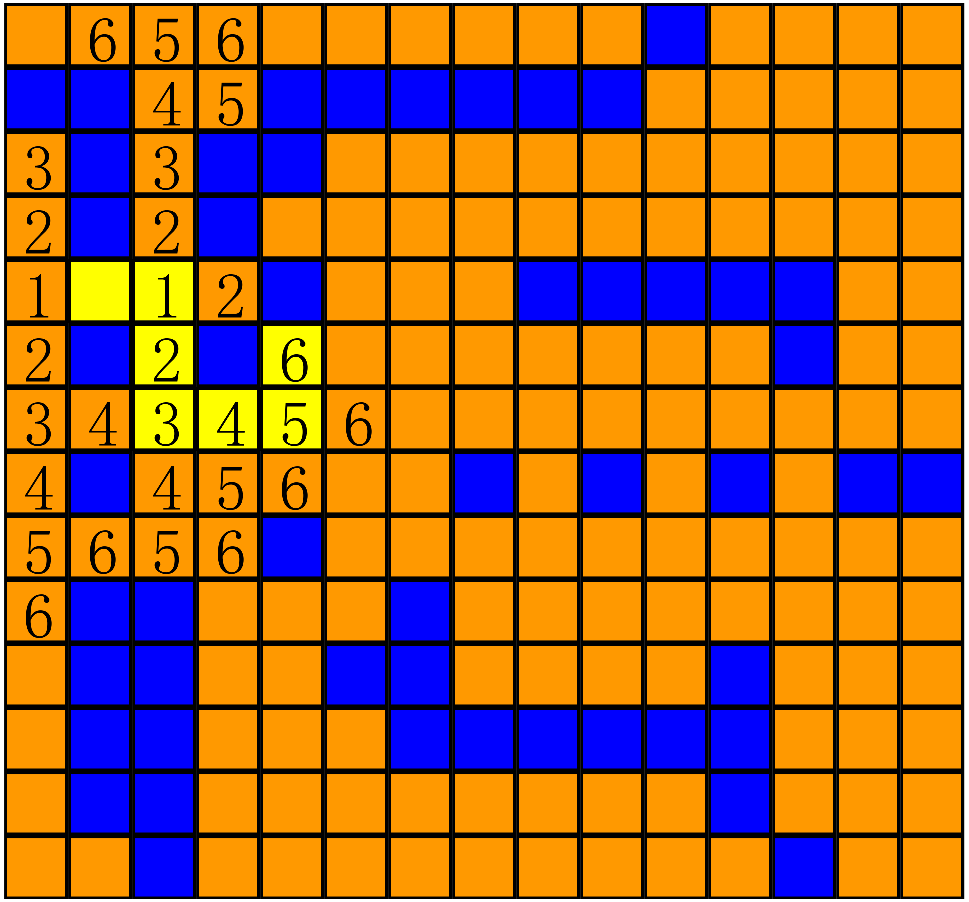
End pin reached.

Trace back.

Illustration of Lee's Algorithm

start pin

end pin



Deque

- Not a proper English word, pronounced as "deck".

- A combination of stack and queue.
  - Items can be inserted and removed from both ends the list.

- Modification 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_rear, pop_front**), but also need to be decremented (**push_front, pop_rear**).