Data Structures and Algorithms

Lecture 5 – Queues

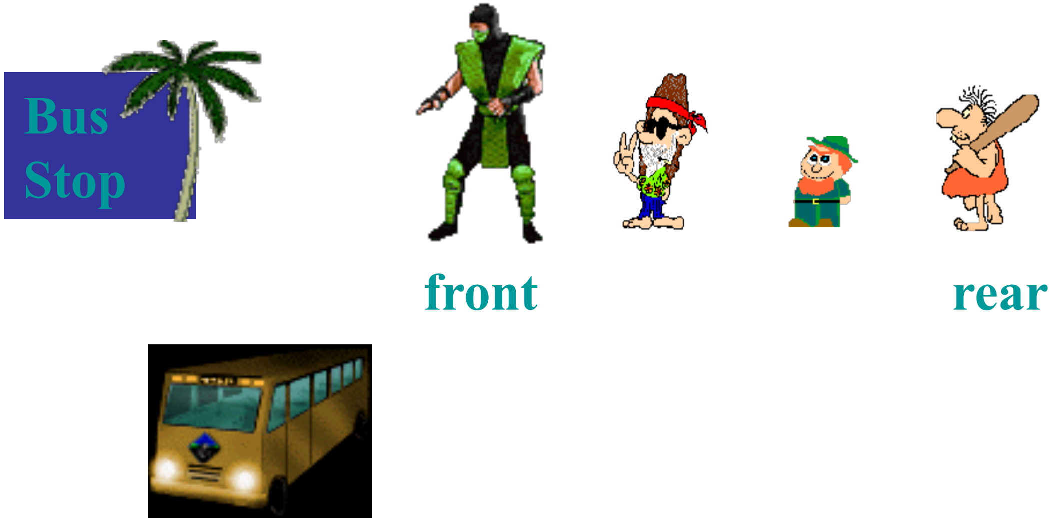
Miao Zhang

Bus Stop Queue

Bus
Stop
front
rear

- ➢ Remove a person from the queue

Bus Stop Queue



Bus Stop

front

rear

Bus Stop Queue

front rear

Bus Stop Queue

Bus Stop

front

rear

- ➢ Add a person to the queue
- ➢ A queue is a FIFO (First-In, First-Out) list.

Definitions

- A queue is an ordered collection of items from which items may be deleted at one end (called the *front* of the queue) and into which items may be inserted at the other end (called the *rear* of the queue)

- A queue is a *data structure* in which elements are removed in the same order they were entered. This is often referred to as FIFO (first in, first out), e.g. print queue, which contains a list of jobs to be printed in order.

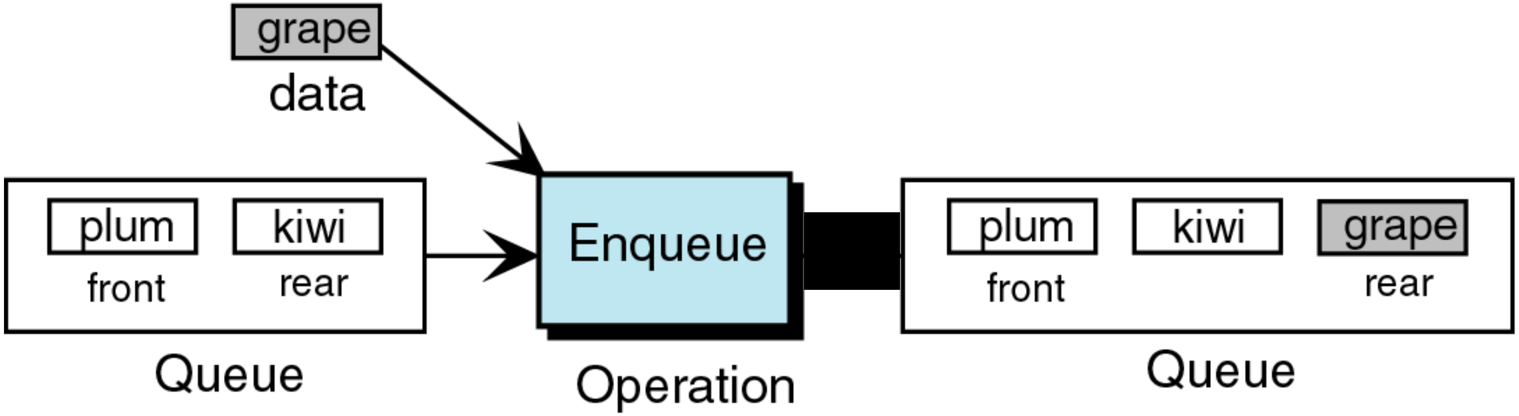
- In contrast, a *stack* is a data structure in which elements are removed in the reverse order from which they were entered. This is referred to as LIFO (last in, first out).

Introduction to Queues

- The queue data structure is very similar to the stack.
- In a stack, all insertions and deletions occur at one end, the top, of the list.
- In the queue, as in the stack, all deletions occur at the head of the list.
- However, all insertions to the queue occur at the tail of the list.

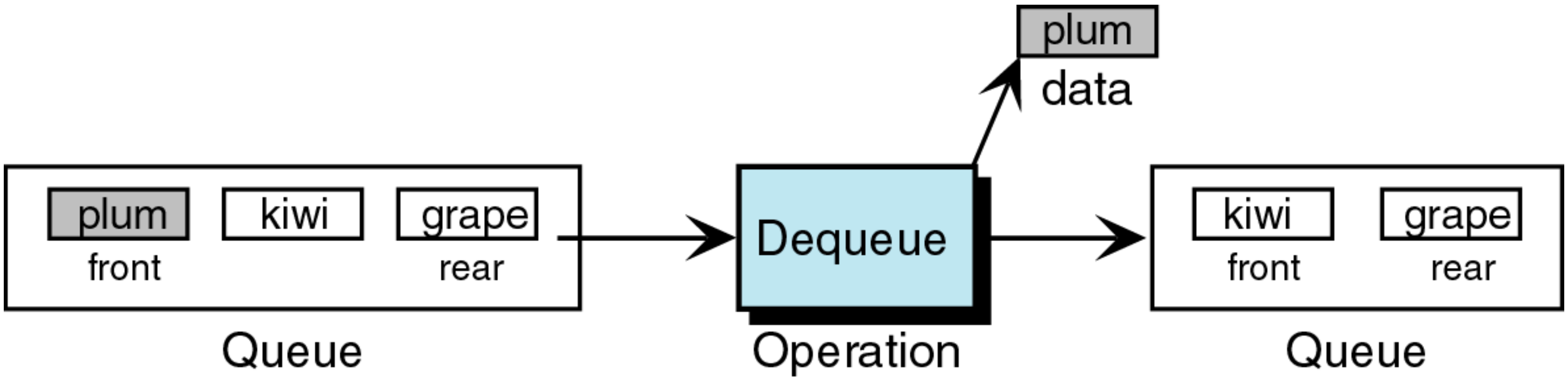
Enqueue

- The queue insert is known as *enqueue*.
- After the data has been inserted, this new element becomes the rear of the queue.



Dequeue

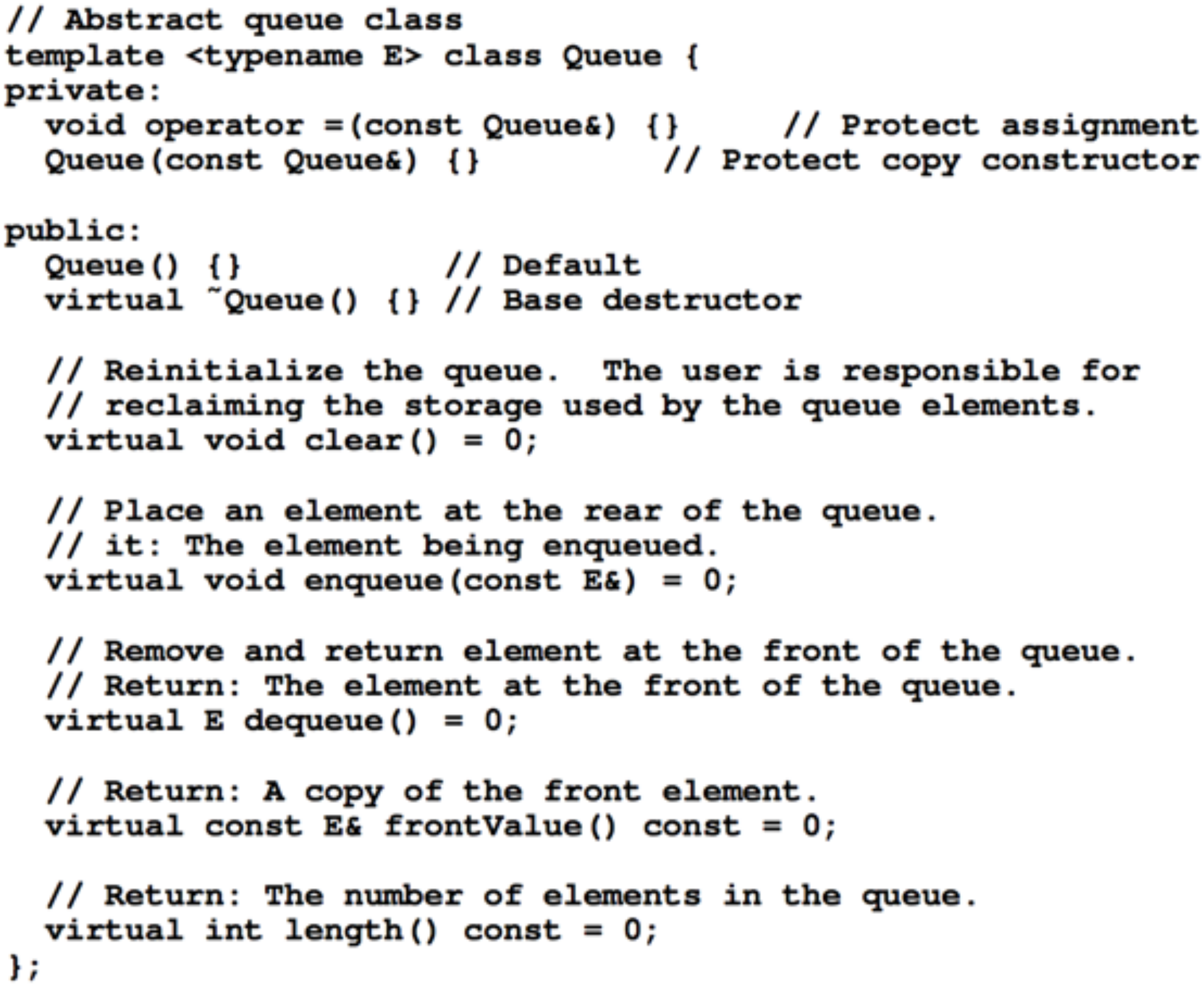
- The queue delete operation is known as *dequeue*.
- The data at the front of the queue is returned to the user and deleted from the queue.



Queue Operations

- Initialize the queue, *Q*, to be the empty queue.
- Determine whether or not if the queue *Q* is empty.
- Determine whether or not if the queue *Q* is full.
- Insert (enqueue) a new item at the rear of the queue *Q*.
- Remove (dequeue) an item from the front of *Q*, provided *Q* is nonempty.

The Abstract Class queue



```
// Abstract queue class
template <typename E> class Queue {
private:
  void operator =(const Queue&) {}     // Protect assignment
  Queue(const Queue&) {}         // Protect copy constructor

public:
  Queue() {}            // Default
  virtual ˜Queue() {} // Base destructor

  // Reinitialize the queue.  The user is responsible for
  // reclaiming the storage used by the queue elements.
  virtual void clear() = 0;

  // Place an element at the rear of the queue.
  // it: The element being enqueued.
  virtual void enqueue(const E&) = 0;

  // Remove and return element at the front of the queue.
  // Return: The element at the front of the queue.
  virtual E dequeue() = 0;

  // Return: A copy of the front element.
  virtual const E& frontValue() const = 0;

  // Return: The number of elements in the queue.
  virtual int length() const = 0;
};
```

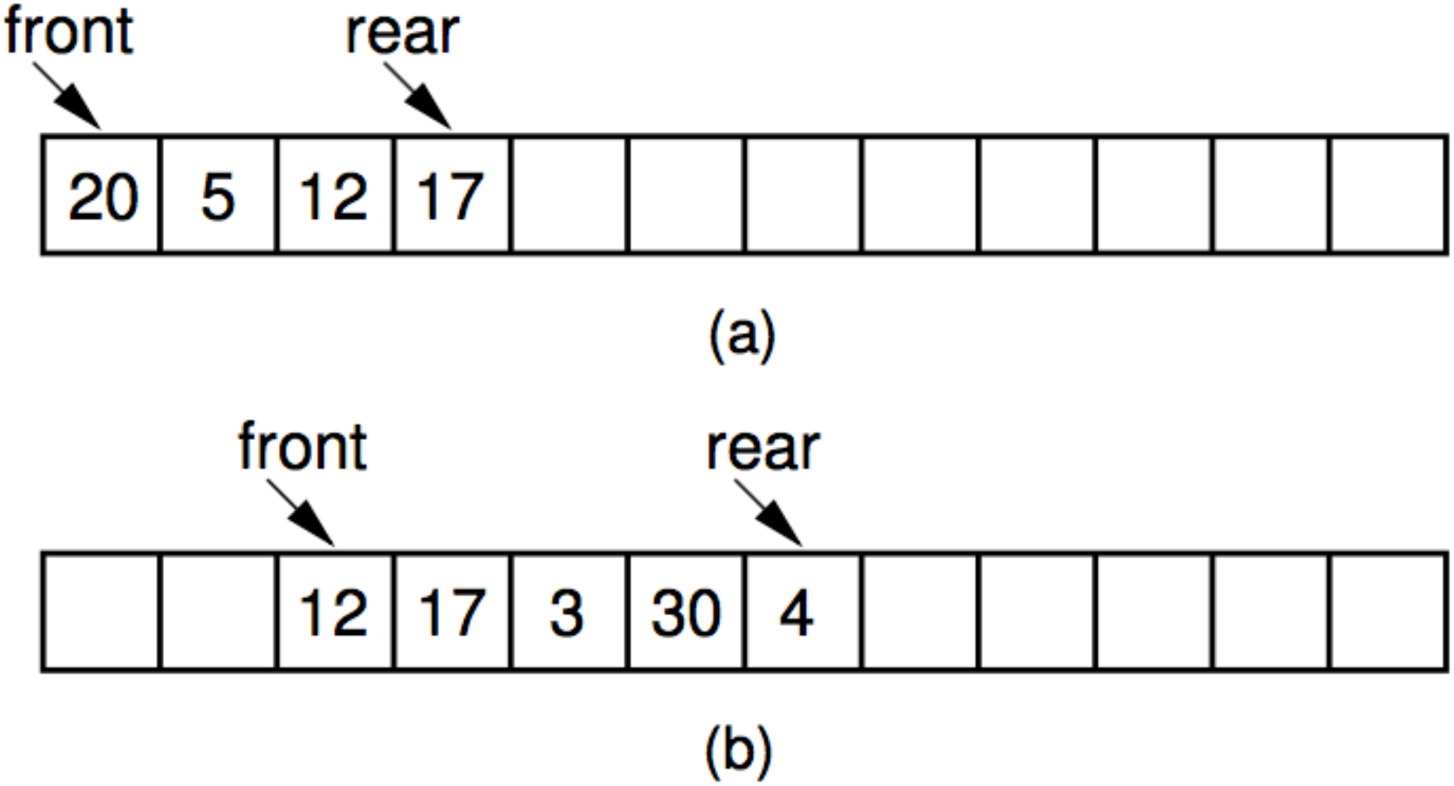
Queue Implementation

Array-based representation

Linked representation

Array-based queue

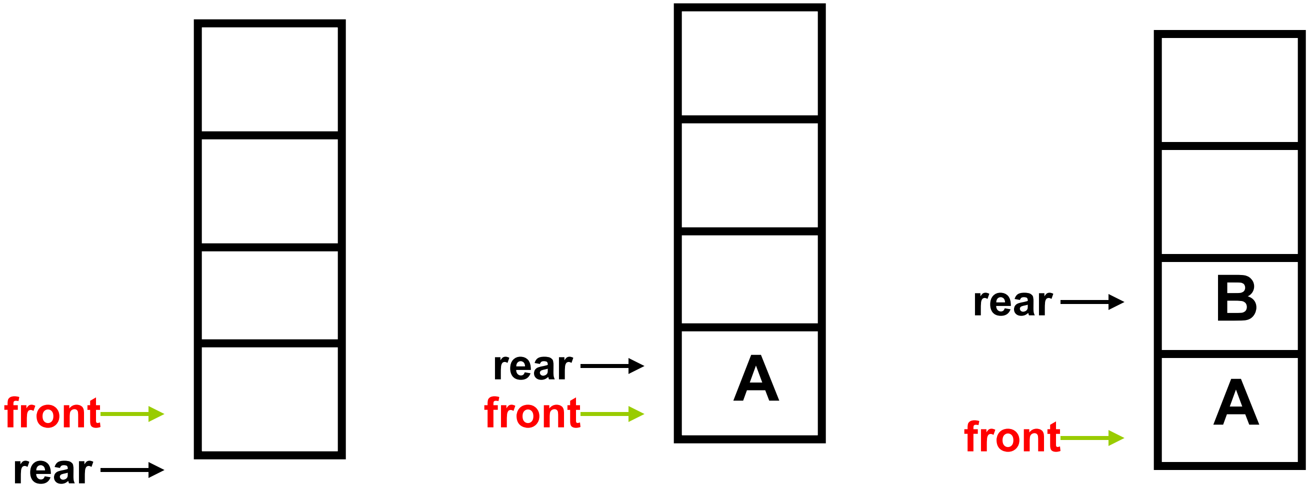
Array-based queue



(a)

(b)

Array-based Queue



Empty：

front=0;

rear=-1;

A is inserted：

front=0;

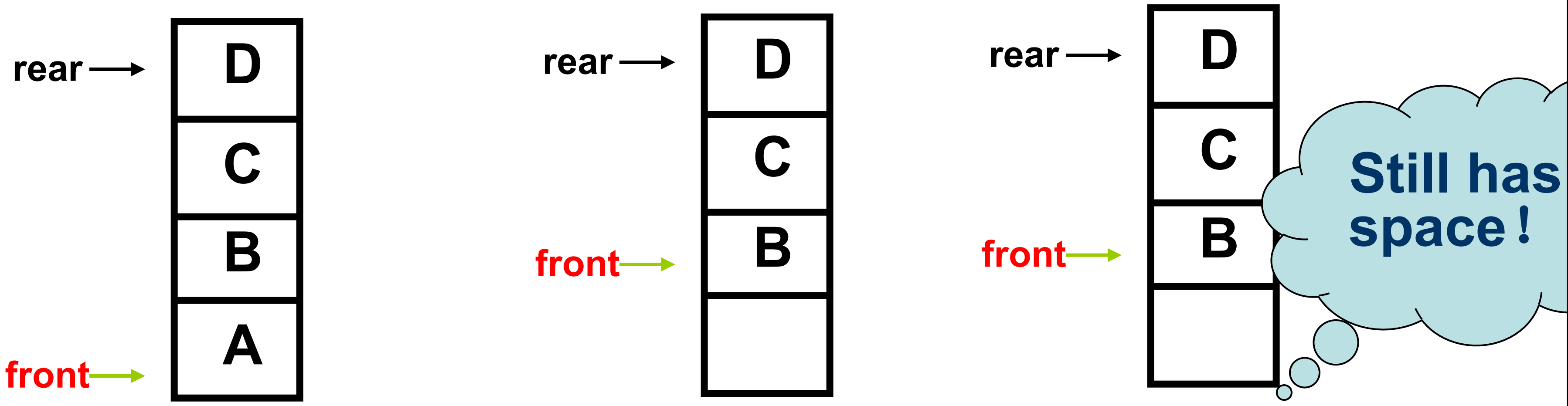
rear=0;

B is inserted：

front=0;

rear=1;

Array-based Queue



Full：

rear=MaxSize-1

Dequeue，

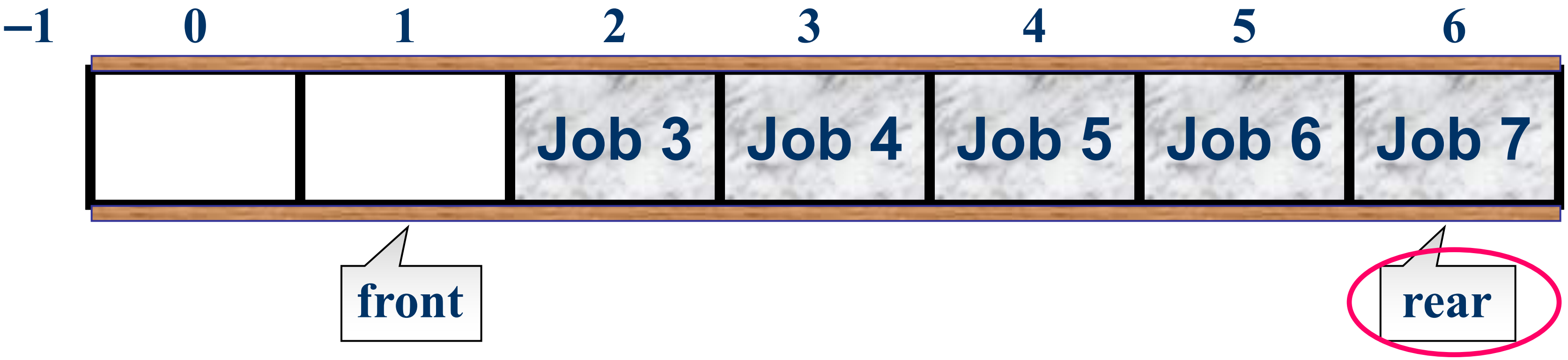
front=1;

Enqueue：

rear=MaxSize-1;

Full! Failed!

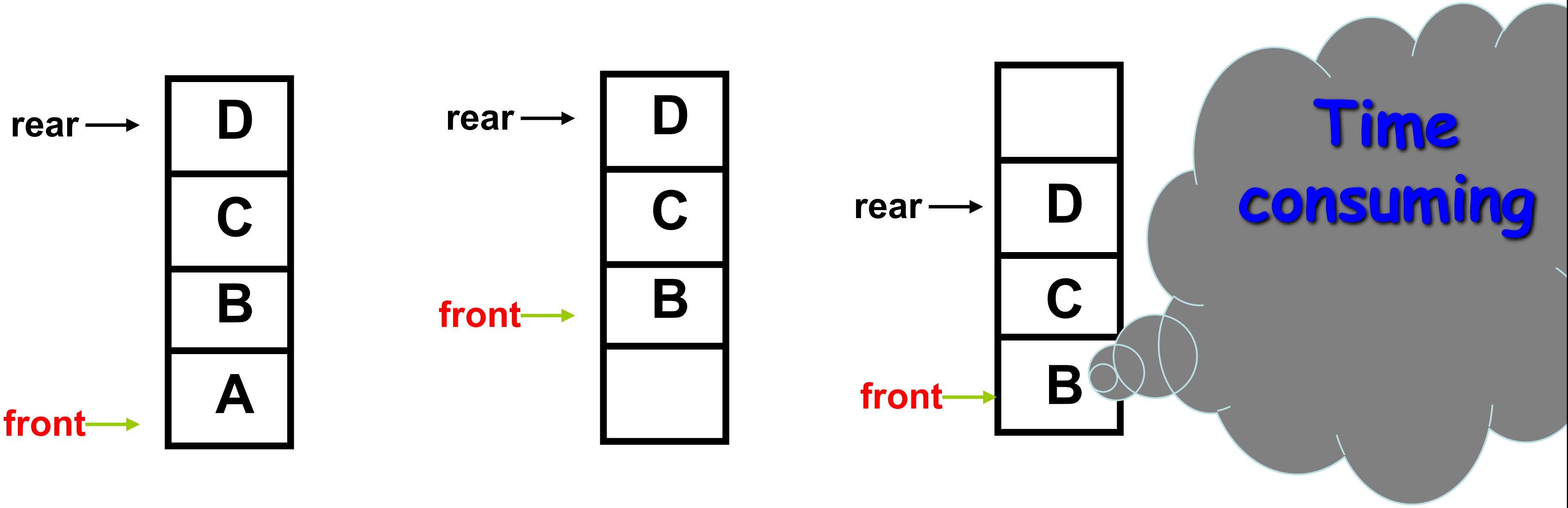
〖Example〗 Job Scheduling in an Operating System



💻 Push Job 1	💻 Push Job 2	💻 Push Job 3
💻 Pop Job 1	💻 Push Job 4	💻 Push Job 5
💻 Push Job 6	💻 Pop Job 2	💻 Push Job 7
💻 Push Job 8		

Are you kidding? Of course I do!
Do you have a better idea?

Solution 1



Full:

rear=MaxSize-1

Dequeue,

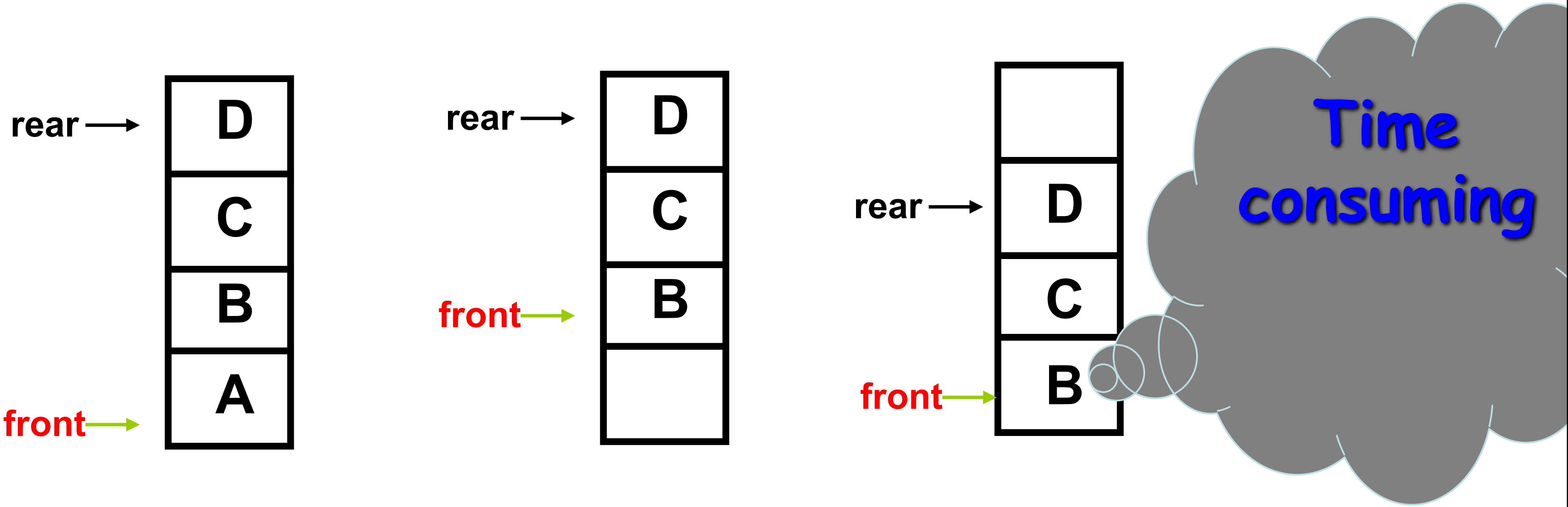
front=1;

Adjust after dequeue operation

front=0;

rear=rear-1;

Solution 1



Time consuming

Full:

rear=MaxSize-1

Dequeue,

front=1;

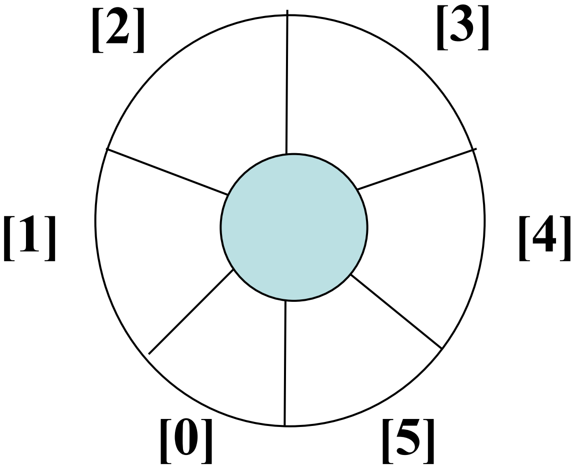
Adjust after dequeue operation

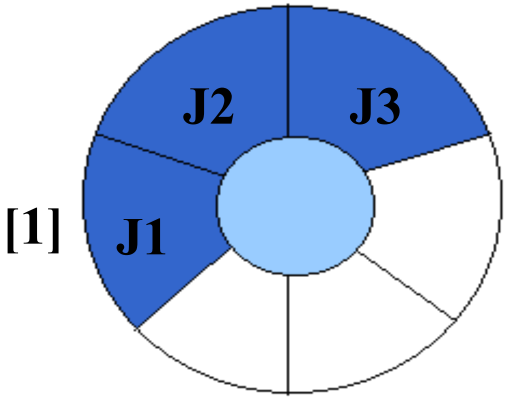
front=0;

rear=rear-1;

Solution2: Circular Queue

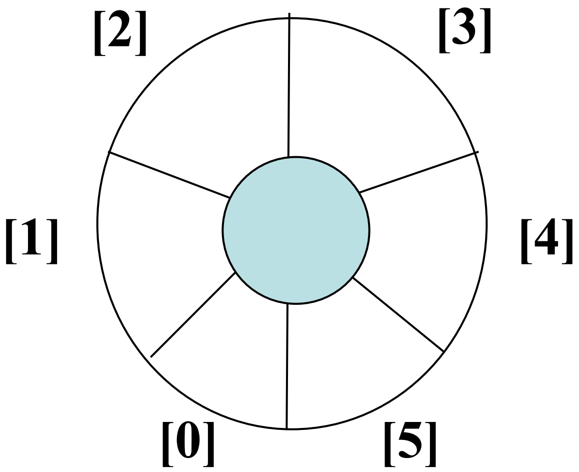
- To avoid the costly operation of coping all the elements again we employ another method called circular queue

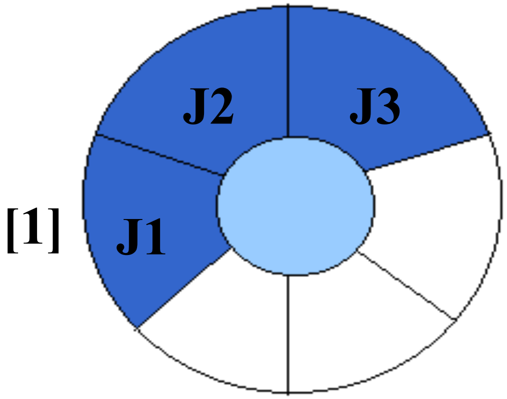




Solution2: Circular Queue

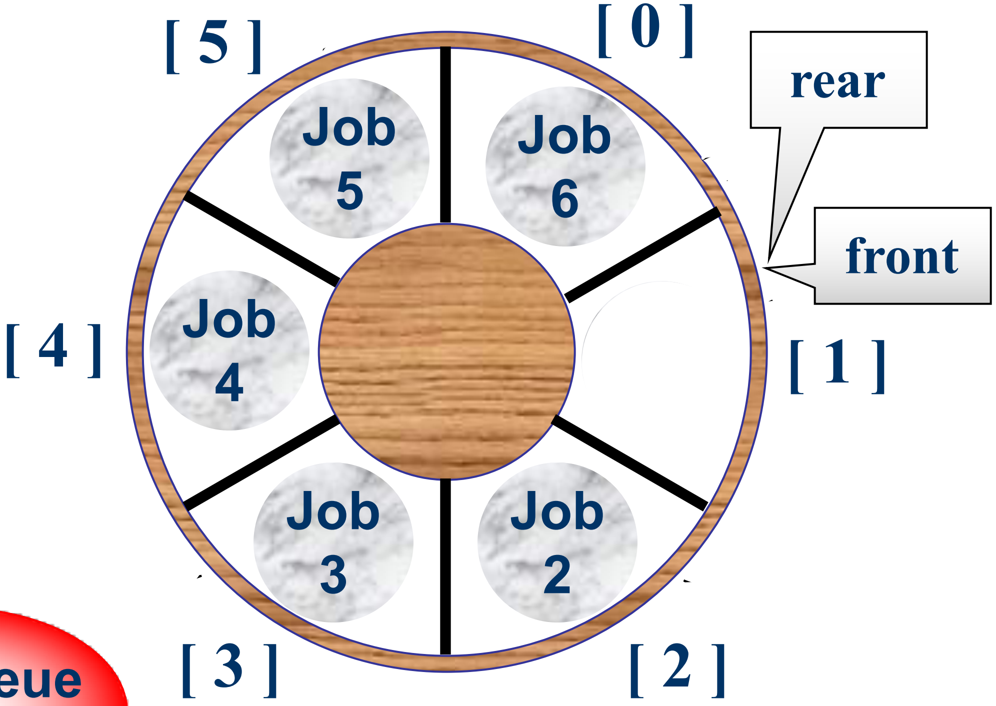
- To avoid the costly operation of coping all the elements again we employ another method called circular queue





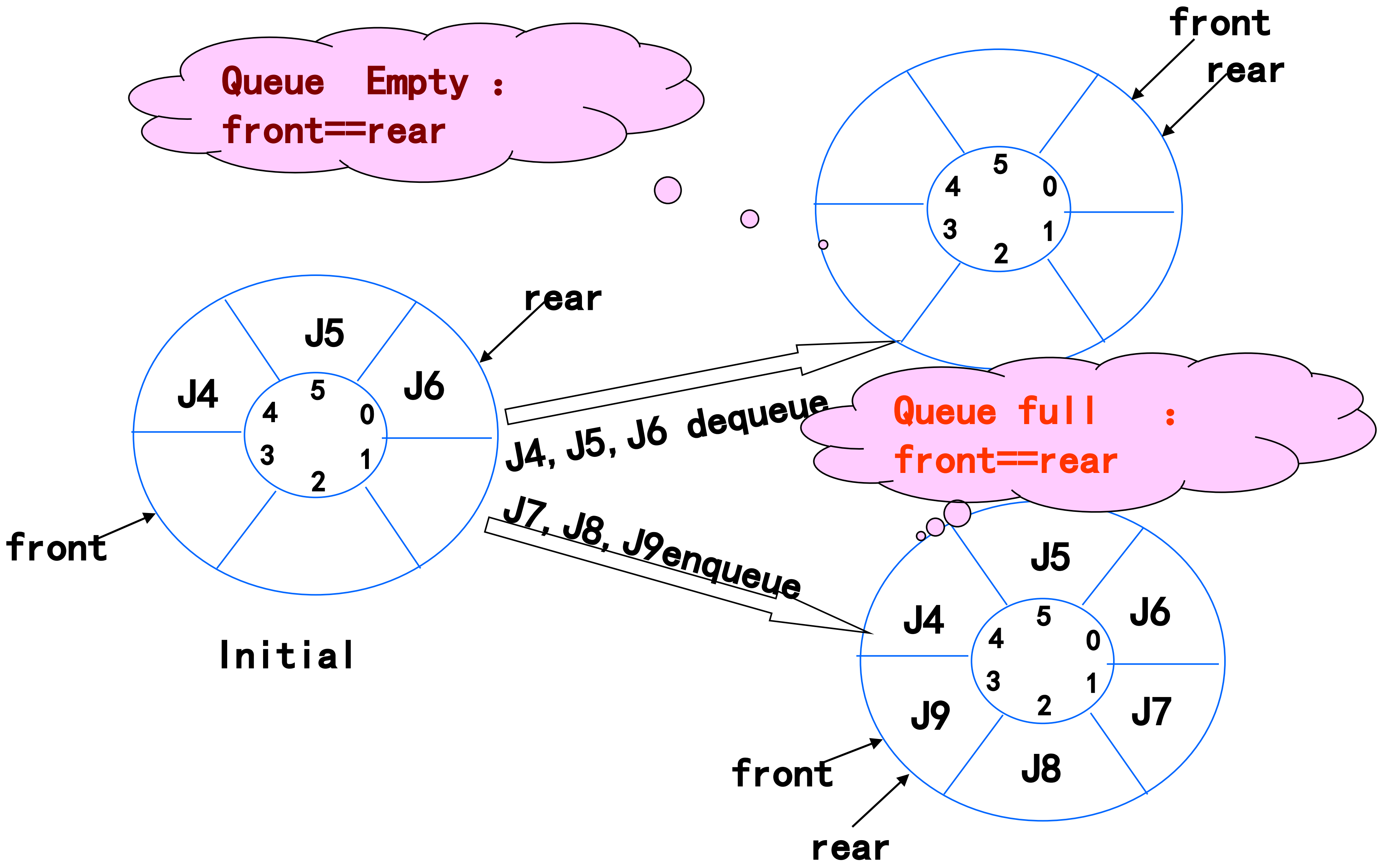
Circular Representation:

- Push Job 1
- Push Job 2
- Push Job 3
- Pop Job 1
- Push Job 4
- Push Job 5
- Push Job 6
- Push Job 7



The queue is full

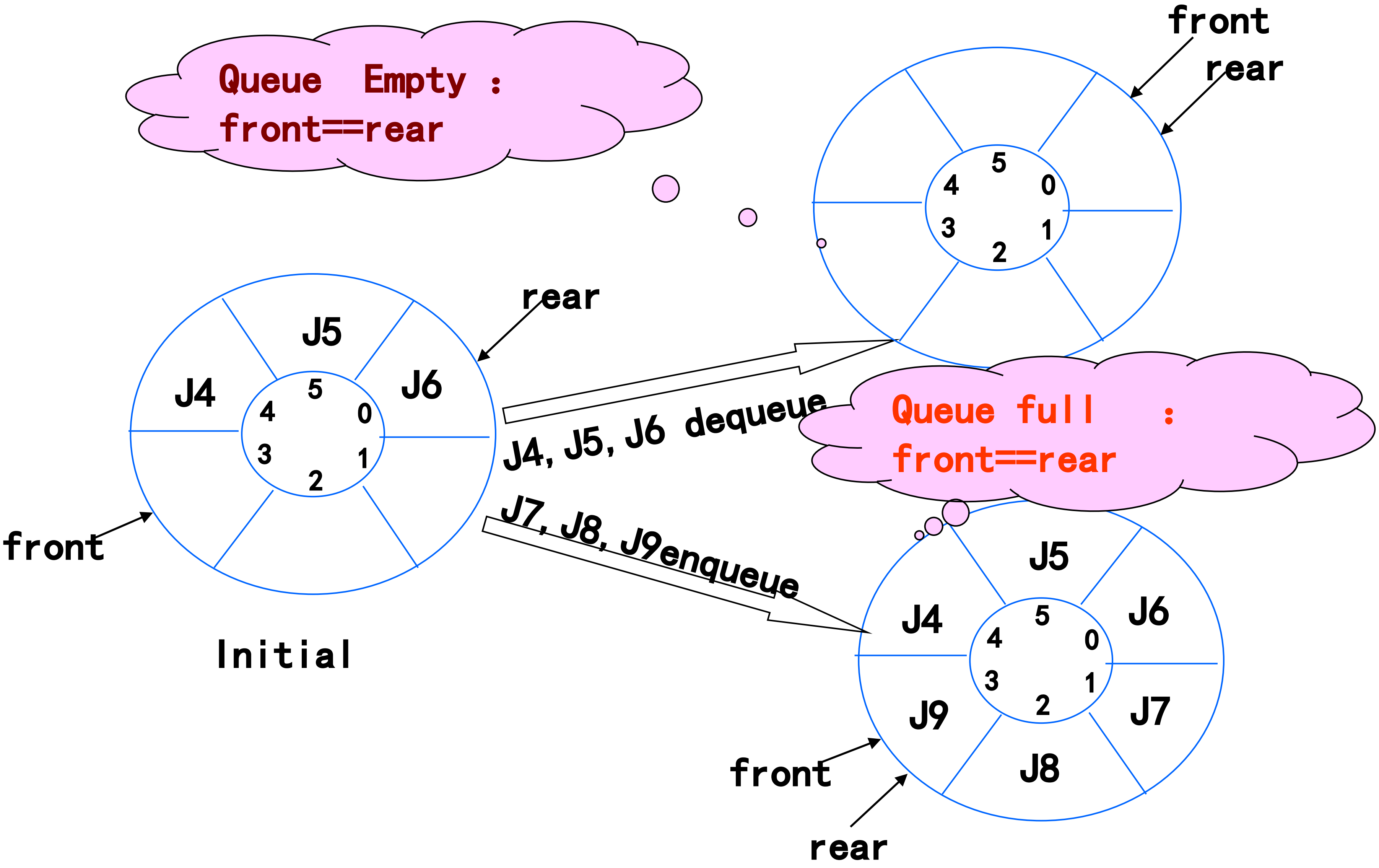
Circular Queue



Queue Empty : front==rear

Queue full : front==rear

Circular Queue



Queue Empty : front==rear

Initial

J4, J5, J6 dequeue

J7, J8, J9enqueue

Queue full : front==rear

Problems with above solution

- How to know if the queue is empty
- How to know if the queue is full
- What is the relation between front and back when queue is full?

- Solution

1. Keep it simple… add counter to the queue
2. Keep an empty slot between Front and Rear i.e., items array uses maxsize - 1 elements

Problems with above solution

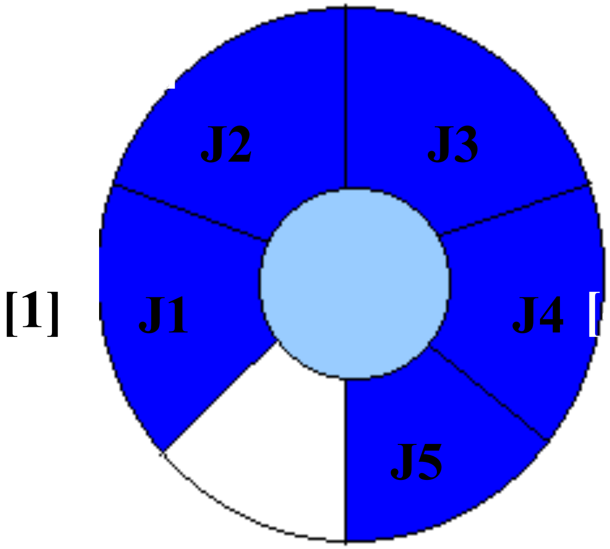
- How to know if the queue is empty
- How to know if the queue is full
- What is the relation between front and back when queue is full?

- Solution

1. Keep it simple… add counter to the queue
2. Keep an empty slot between Front and Rear i.e., items array uses maxsize - 1 elements

Solution b: Leave one empty space when queue is full

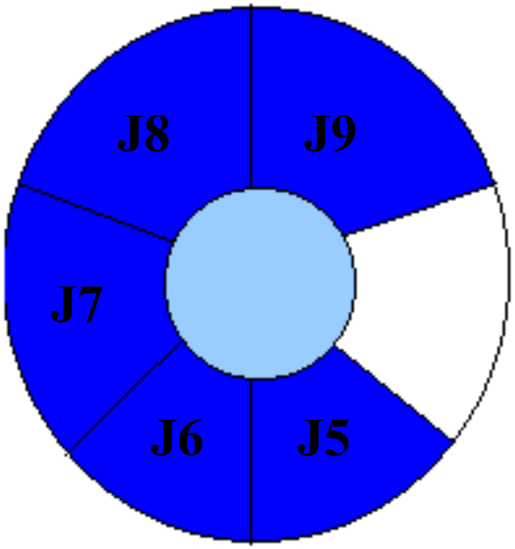
FULL QUEUE



front =0

rear = 5

FULL QUEUE



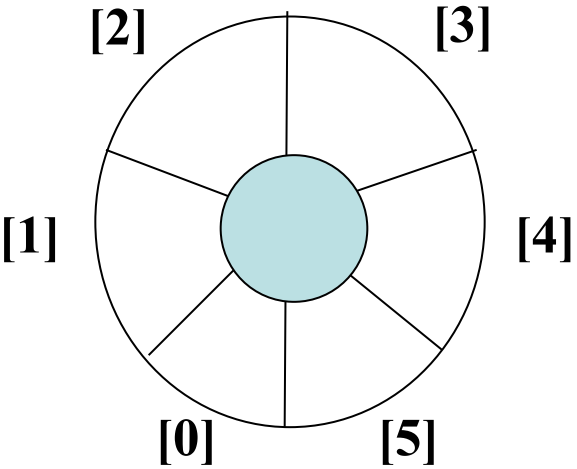
front =4

rear =3

Full queue：

(rear+1)%MaxSize==front

EMPTY QUEUE



front = 0
rear = 0

Empty queue：

front=rear;

Enqueue & Dequeue

Dequeue：

front= (front+1)%MaxSize

Enqueue：

rear= (rear+1)%MaxSize

Array-based queue implementation

```
// Array-based queue implementation
template <typename E> class AQueue: public Queue<E>{

private:
      int maxSize; // Maximum size of queue
      int front; // Index of front element
      int rear; // Index of rear element
      E *listArray; // Array holding queue elements
public:
     AQueue(int size =defaultSize) ;
    ～Aqueue() {delete [] listArray;}

    void MakeEmpty () {
    front=rear=0}
int IsEmpty() {
        return front==rear;}
int IsFull() {
        return (rear+1)%MaxSize==front);}
int Length {
        return (rear-front +MaxSize)%MaxSize;}
```

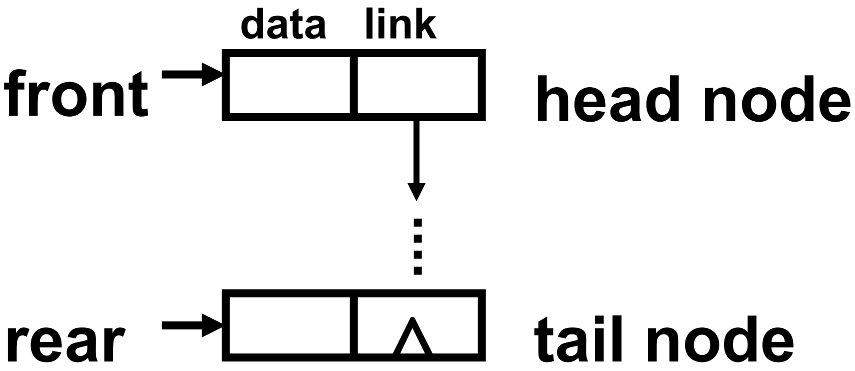
```
void enqueue(const E& it) { // Put "it" in queue
    Assert(!IsFull());
     rear=(rear+1) % maxSize;
     listArray[rear] = it;
}

E dequeue() { // Take element out
    Assert(!IsEmpty());
    front = (front+1) % maxSize; // Circular increment
    return it;
}
```

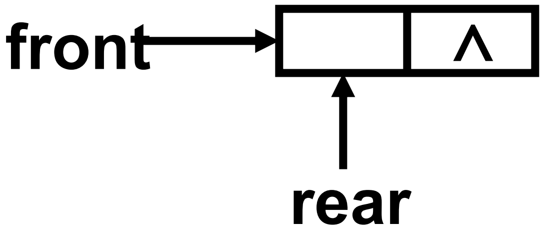
Linked Queue

Linked Queue

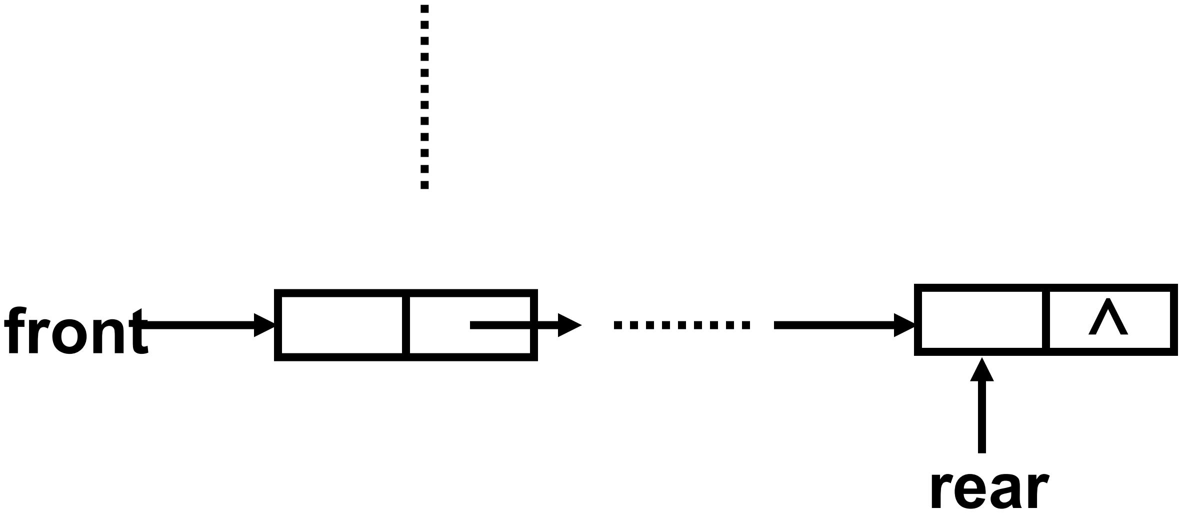
· Linked queue： front and rear are the head and tail pointer, respectively.



Linked Queue



Empty queue：front = rear = NULL



Linked Queue

```
// Linked queue implementation
template <typename E> class LQueue: public Queue<E> {
private:
    Link<E>* front; // Pointer to front queue node
    Link<E>* rear; // Pointer to rear queue node
    int size; // Number of elements in queue

public:
    LQueue(int sz =defaultSize) // Constructor
    { front = rear = new Link<E>(); size = 0; }
    ˜LQueue() { clear(); delete front; } // Destructor
```

Linked Queue

```
void clear() { // Clear queue
    while (front->next != NULL) { // Delete each link node
        rear = front;
        delete rear;
    }
    rear = front;
    size = 0;
}

void enqueue(const E& it) { // Put element at rear
    rear->next = new Link<E>(it, NULL);
    rear = rear->next;
    size++;
}
```

```
E dequeue() { // Remove element from front
    Assert (size != 0, "Queue is empty");
    E it = front->next->element; // Store dequeued value
    Link<E>* ltemp = front->next; // Hold dequeued link
    front->next = ltemp->next; // Advance front
    if (rear == ltemp) rear = front; // Dequeue last element
        delete ltemp; // Delete link
        size --;
    return it; // Return element value
}
```

Comparison of Array-Based and Linked Queues

- All member functions for both the array-based and linked queue implementations require constant time.

- The space comparison issues are the same as for the equivalent stack implementations.

- Unlike the array-based stack implementation, there is no convenient way to store two queues in the same array, unless items are always transferred directly from one queue to the other.

Application of queues

$(a+b)^i$: print out the coefficients of polynomials

1 1 i=1

1 2 1 2

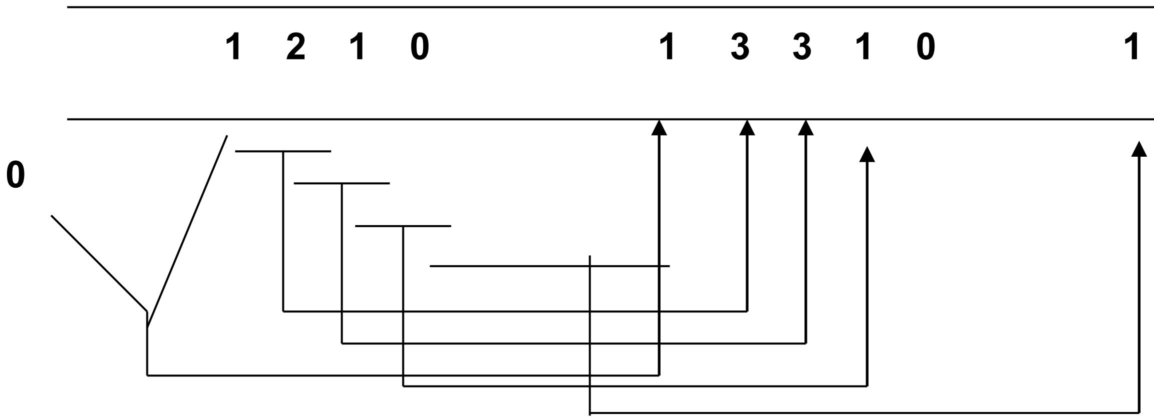
1 3 3 1 3

1 4 6 4 1 4

1 5 10 10 5 1 5

1 6 15 20 15 6 1 6

Application of queues



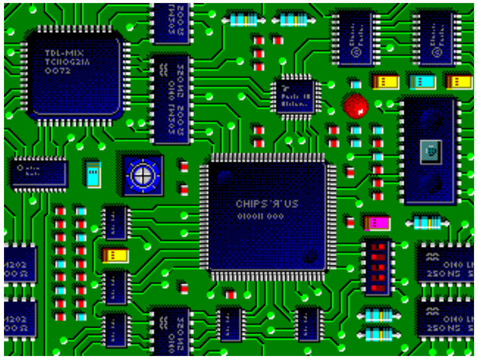
```
Queue q;
q.MakeEmpty();
q.EnQueue(1);
q.EnQueue(1);
Int s=0;

for ( int i=1; i<=n; i++)
{
        cout<<endl;
        q.EnQueue(0);
        for (int j=1;
j<=i+2; j++){
                 int
t=q.DeQueue ();

q.EnQueue(s+t);
                 s=t;
                 if (j!=i+2)
                 cout<<s<< ‘
```

Application: Wire Routing

- Similar to Rat in a Maze problem ,but this time it has to find the shortest path between two points to minimize signal delay
- Used in designing electrical circuit boards



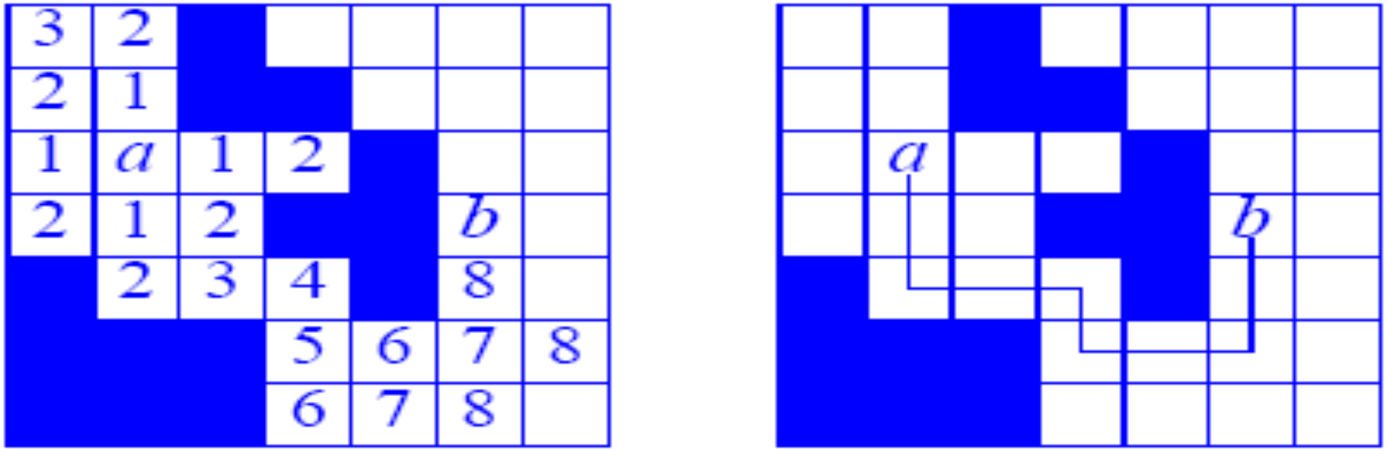
Wire Routing Algorithm

The shortest path between grid positions *a* and *b* is found in two passes

1. Distance-labeling pass (i.e., labeling grids)
2. Path-identification pass (i.e., finding the shortest path)

Wire Routing Algorithm

1. Labeling Grids: Starting from position *a*, label its reachable neighbors 1. Next, the reachable neighbors of squares labeled 1 are labeled 2. This labeling process continues until we either reach *b* or have no more reachable squares. The shaded squares are blocked squares.



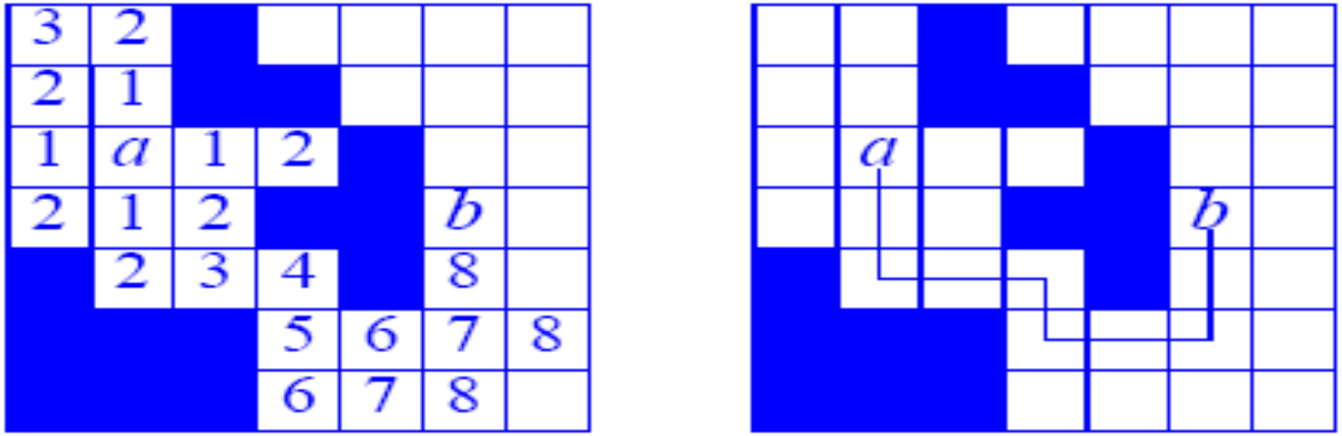
(a) Distance labeling

(b) Wire path

Figure 6.12 Wire routing

Wire Routing Algorithm

2. Finding the shortest path: Starting from position *b*, move to any one of its neighbors labeled one less than *b*'s label. Such a neighbor must exist as each grid's label is one more than that of at least one of its neighbors. From here, we move to one of its neighbors whose label is one less, and so on until we reach *a*.



3	2	▨				
2	1	▨	▨			
1	*a*	1	2	▨		
2	1	2	▨	▨	*b*	
▨	2	3	4	▨	8	
▨	▨	▨	5	6	7	8
▨	▨	▨	6	7	8	

(a) Distance labeling

		▨				
		▨	▨			
	a			▨		
			▨	▨	*b*	
▨				▨		
▨	▨	▨				
▨	▨	▨				

(b) Wire path

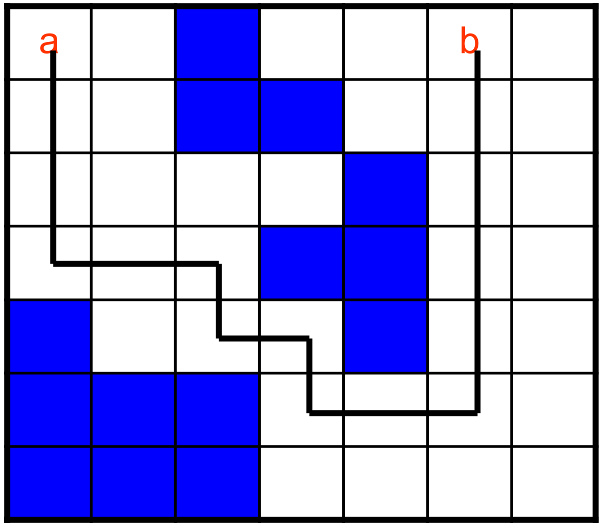
Figure 6.12 Wire routing

Wire Routing Algorithm Exercise

Consider the wire-routing grid. You are to route a wire between *a*=(1, 1) and *b*=(1, 6). Label all grid positions that are reached in the distance-labeling pass by their distance value. Then use the methodology of the path-identification pass to mark the shortest wire path.

a	1	■			b	
1	2	■	■		14	
2	3	4	5	■	13	14
3	4	5	■	■	12	13
■	5	6	7	■	11	12
■	■	■	8	9	10	11
■	■	■	9	10	11	

a		■			b	
		■	■			
				■		
			■	■		
■				■		
■	■	■				
■	■	■				



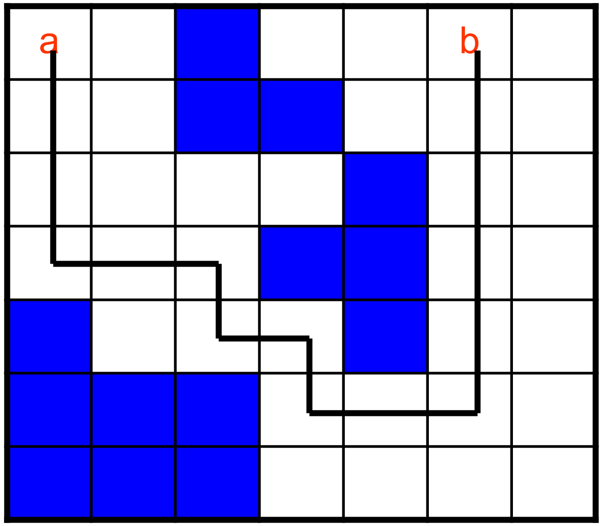
Is there only one shortest path?

Wire Routing Algorithm Exercise

Consider the wire-routing grid. You are to route a wire between *a*=(1, 1) and *b*=(1, 6). Label all grid positions that are reached in the distance-labeling pass by their distance value. Then use the methodology of the path-identification pass to mark the shortest wire path.

a	1	■			b	
1	2	■	■		14	
2	3	4	5	■	13	14
3	4	5	■	■	12	13
■	5	6	7	■	11	12
■	■	■	8	9	10	11
■	■	■	9	10	11	

a		■			b	
		■	■			
				■		
			■	■		
■				■		
■	■	■				
■	■	■				



Is there only one shortest path?

Implementing Wire Routing Algorithm

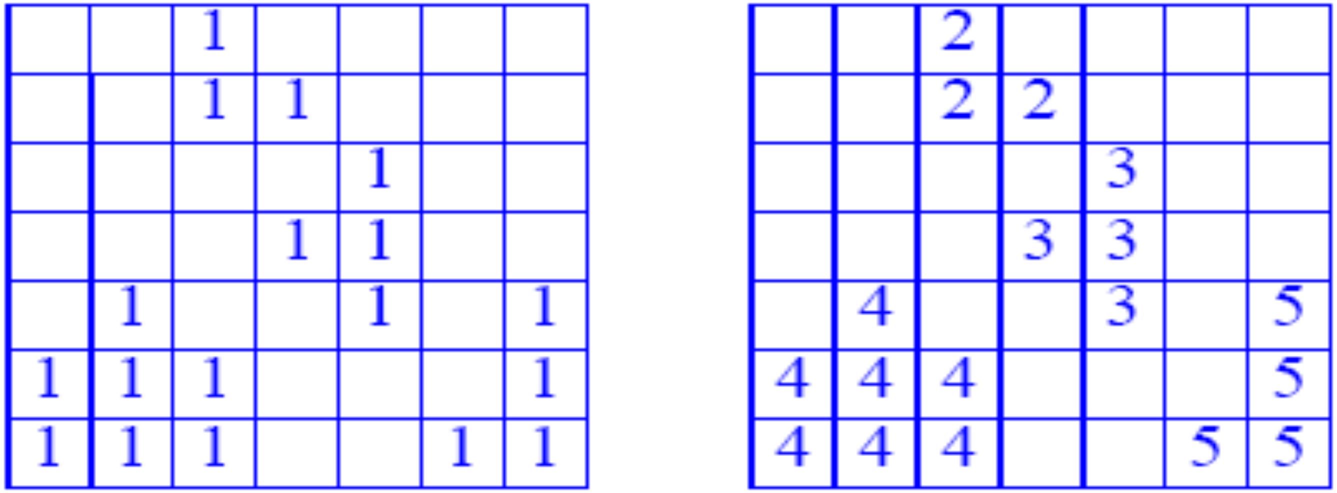
- An *m x m* grid is represented as a 2-D array with a 0 representing an open position and a 1 representing a blocked position
- the grid is surrounded by a wall of 1s
- the array offsets helps us move from a position to its neighbors
- A queue keeps track of labeled grid positions whose neighbors have yet to be labeled
- What is the time complexity of the algorithm?

  ➔ O(m^2) for labeling grids & O(length of the shortest path) for path construction

Application: Image-Component Labeling

- **Background information:**
  - A digitized image is an *m x m* matrix of pixels.
  - In a binary image, each pixel is 0 or 1.
  - A 0 pixel represents image background, while a 1 represents a point on an image component.
  - Two pixels are adjacent if one is to the left, above, right, or below the other.
  - Two component pixels that are adjacent are pixels of the same image component.
- **Problem:** Label the component pixels so two pixels get the same label if they are pixels of the same image component.

Image-Component Labeling Example



		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

(a) A 7 × 7 image

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

(b) Labeled components

Figure 6.13 Image-component labeling

- **The blank squares represent background pixels and 1s represent component pixels**
- **Pixels (1,3), (2,3), and (2,4) are from the same component**

Image-Component Labeling Algorithm

- Idea:Similar to the wire-routing problem
- Steps:

1. Scan pixels one by one (row-major search).
2. When encounter a 1 pixel, give a unique component identifier.
3. Find all neighbors by expanding from that pixel and give the unique identifier.
4. Repeat for all pixels.

Homework

- Please refer to Icourse，Huawei Cloud.

- Due date for quiz: 23:30 2022/3/29
- Due date for homework: 23:30 2022/4/3
- Due data for online lab assignment: 2022/4/3 23：30
- Due data for offline lab assignment: 2022/4/4 18:00