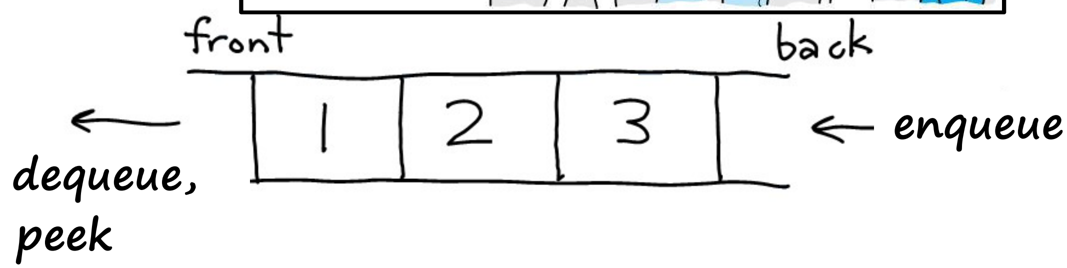
Queue Abstract Data Type

Queues

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- basic queue operations:
  - **enqueue**: Add an element to the back.
  - **dequeue**: Remove the front element.
  - **peek**: Examine the front element.

Queues in computer science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send

- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order

- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Programming with Queues

`enqueue(`**value**`)`	places given value at the back of queue
`dequeue()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; throws a `NoSuchElementException` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
Queue<int> q;
q.enqueue(42);
q.enqueue(-3);
q.enqueue(17);          // front [42, -3, 17] back
cout << q.dequeue();    // 42
```

Queue processing styles

- As with stacks, we must pull contents out of queue to view them.

```
while (!q.isEmpty()) {
    do something with q.dequeue();
}
```

  – another style: Examining each element exactly once.

```
int n = q.size();
for (int i = 0; i < n; i++) {
    do something with q.dequeue();
    (including possibly re-adding it to the queue)
}
```

    - Why do we need the `n` variable?

Mixing stacks and queues

- We often mix stacks and queues to achieve certain effects.
  - Example: Reverse the order of the elements of a queue.

```
Queue<int> q;
q.enqueue(1);
q.enqueue(2);
q.enqueue(3);                        // [1, 2, 3]

Stack<int> s;
while (!q.isEmpty()) {         // Q -> S
    s.push(q.dequeue());
}
while (!s.isEmpty()) {         // S -> Q
    q.enqueue(s.pop());
}
// queue contents are [3, 2, 1]
```

Exercise 1

- Write a method `stutter` that accepts a queue of integers as a parameter and replaces every element of the queue with two copies of that element.
  - `front [1, 2, 3] back`
    becomes
    `front [1, 1, 2, 2, 3, 3] back`

Exercise 2

- Write a method `mirror` that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order.
  - `front [a, b, c] back`
    becomes
    `front [a, b, c, c, b, a] back`

Exercise 3

- Modify the exam score program so that it reads the exam scores into a queue and prints the queue.

```
Yeilding      Janet     87
White         Steven    84
Todd          Kim       52
Tashev        Sylvia    95
...
```

Reading from file

```
ifstream file;
Queue<string> q;    // queue of strings
file.open("data.txt");
while (file.good()){
    getline(file, line);
    q.enqueue(line);
}
file.close();

while(!q.isEmpty()){
    cout << q.dequeue() << endl;
}
  // names and score are all gone; cannot
  // process them any further
```

Exercise 3 (cont.)

- What if we want to further process the exams after printing?
  - E.g. filter out any exams where the student got a score of 100.
  - Then perform reverse and print the remaining students.

Revision

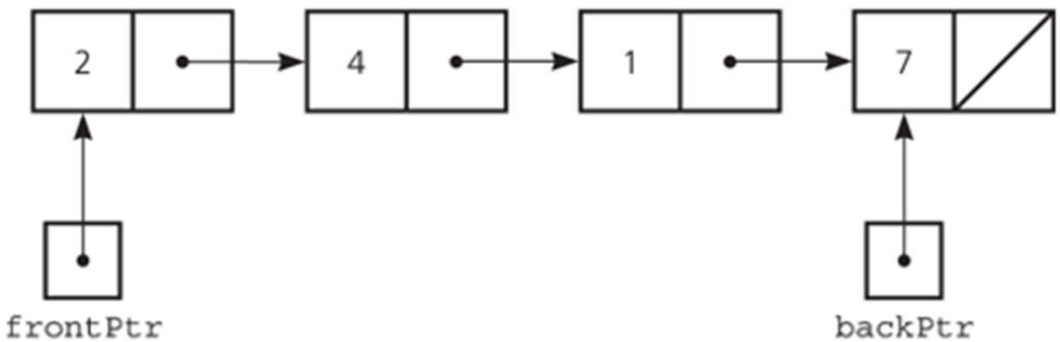
```
ifstream file;
Queue<string> q;    // queue of strings
file.open("data.txt");
while (file.good()){
    getline(file, line);
    q.enqueue(line);
}
file.close();
q.enqueue("");
while(q.peek()!= ""){
    string str = q.dequeue();
    cout << str << endl;
    q.enqueue(str);
}
q.dequeue();

// complete the rest of the exercise
```

Implementations of Queue

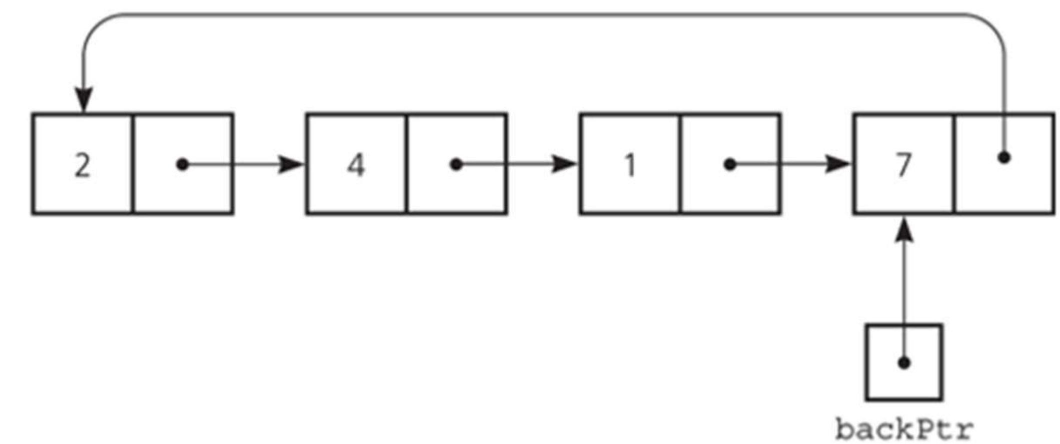
- Pointer-based implementations of queue
  - A linked list with two external references
    - A reference to the front
    - A reference to the back
  - A circular linked list with one external reference
    - A reference to the back

- Array-based implementations of queue
  - A naive array-based implementation of queue
  - A circular array-based implementation of queue

Pointer-based implementations of queue



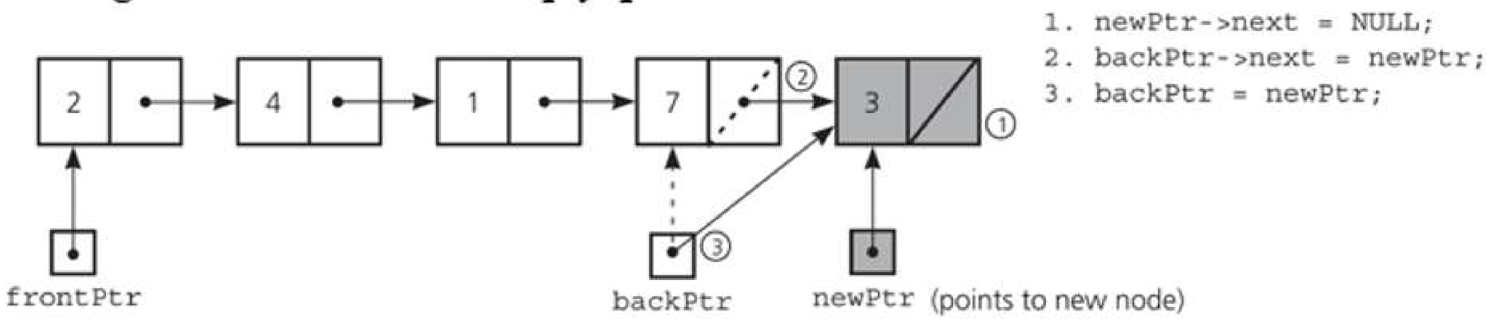
a linear linked list with two external pointers



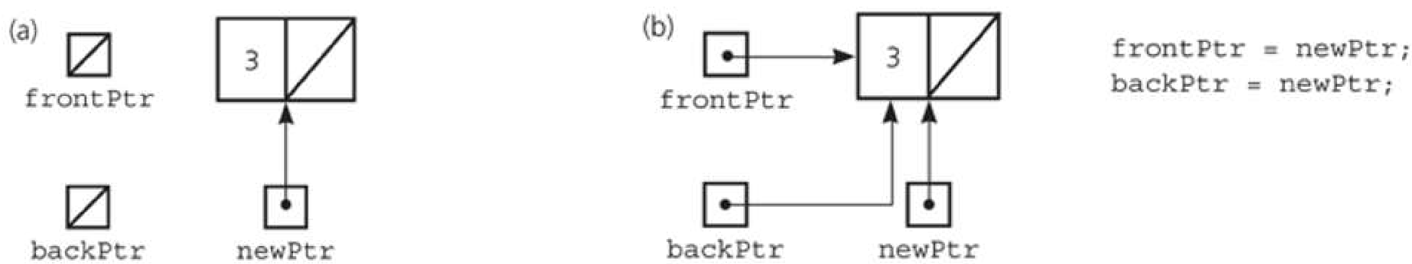
a circular linear linked list with one external pointer

Linked list Implementation – enqueue

Inserting an item into a nonempty queue



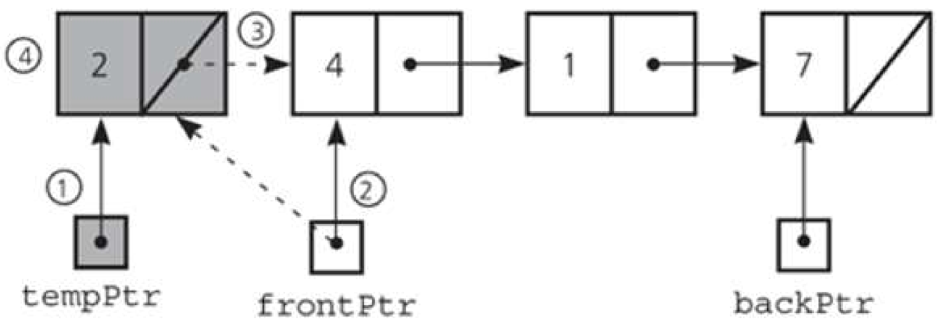
Inserting an item into an empty queue



a) before insertion *b) after insertion*

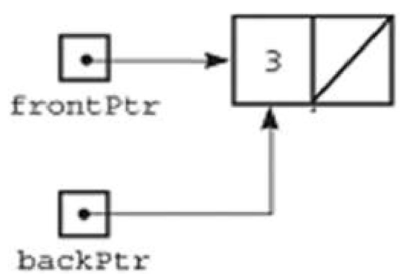
Linked list Implementation – dequeue

Deleting an item from a queue of more than one item

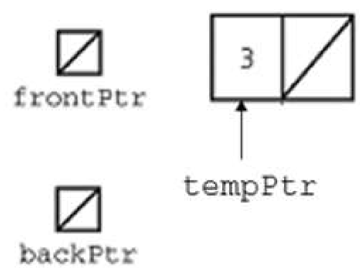



```
1. tempPtr = frontPtr;
2. frontPtr = frontPtr->next
3. tempPtr->next = NULL;
4. delete tempPtr;
```

Deleting an item from a queue with one item



before deletion



after deletion

```
tempPtr = frontPtr;
frontPtr = NULL;
backPtr = NULL;
delete tempPtr;
```

Linked List implementation- Queue Node Class

```
// QueueNode class for the nodes of the Queue

template <class Object>
class QueueNode
{
    public:
        QueueNode(const Object& e = Object(), QueueNode* n = nullptr)
            : item(e), next(n) {}

        Object item;
        QueueNode* next;
};
```

Linked list Implementation – Queue Class

```
#include "QueueException.h"

template <class T>
class Queue {
public:
   Queue();                                   // default constructor
   Queue(const Queue& rhs);                   // copy constructor
   ~Queue();                                  // destructor
   Queue& operator=(const Queue & rhs); //assignment operator
   bool isEmpty() const;
   void enqueue(const T& newItem);
   T dequeue() throw(QueueException);
   T peek() const throw(QueueException);
private:
   QueueNode<T> *backPtr;
   QueueNode<T> *frontPtr;
};
```

Linked List Implementation – constructor, deconstructor, isEmpty

```
template<class T>
Queue<T>::Queue() : backPtr(nullptr), frontPtr(nullptr){}

template<class T>
Queue<T>::~Queue() {        // destructor
   while (!isEmpty())
      dequeue();    // backPtr and frontPtr are null at this point
}

template<class T>
bool Queue<T>::isEmpty() const{
   return backPtr == nullptr;
}
```

Linked list Implementation – enqueue

```
template<class T>
void Queue<T>::enqueue(const T& newItem) {
       // create a new node
       QueueNode<T> *newPtr = new QueueNode<T>;

       // set data portion of new node
       newPtr->item = newItem;
       newPtr->next = nullptr;

       // insert the new node
       if (isEmpty())         // insertion into empty queue
          frontPtr = newPtr;
       else                   // insertion into nonempty queue
          backPtr->next = newPtr;

       backPtr = newPtr;      // new node is at back
}
```

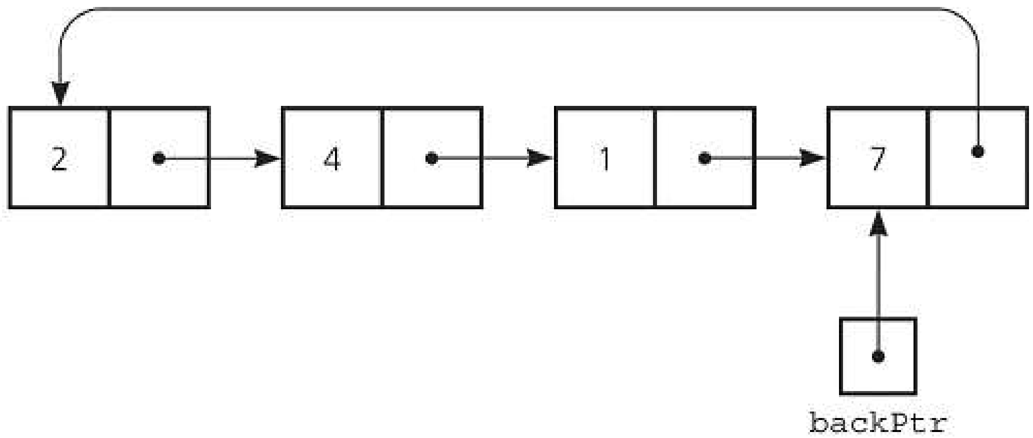
Linked list Implementation – dequeue

```
template<class T>
T Queue<T>::dequeue() throw(QueueException) {
   if (isEmpty())
       throw QueueException(
          "QueueException: Empty queue, cannot dequeue");
   else {     // queue is not empty; remove front
      QueueNode<T> *tempPtr = frontPtr;
      T queueFront = frontPtr->item;
      if (frontPtr == backPtr) {   // one node in queue
         frontPtr = nullptr;
         backPtr = nullptr;
      }
      else
         frontPtr = frontPtr->next;
      tempPtr->next = nullptr;     // defensive strategy
      delete tempPtr;
      return queueFront;
    }
}
```

Linked list Implementation – peek

```
template<class T>
T Queue<T>::peek() const throw(QueueException) {
   if (isEmpty())
      throw QueueException(
          "QueueException: empty queue, cannot peek");
   else        // queue is not empty; retrieve front
      return(frontPtr->item);
}
```

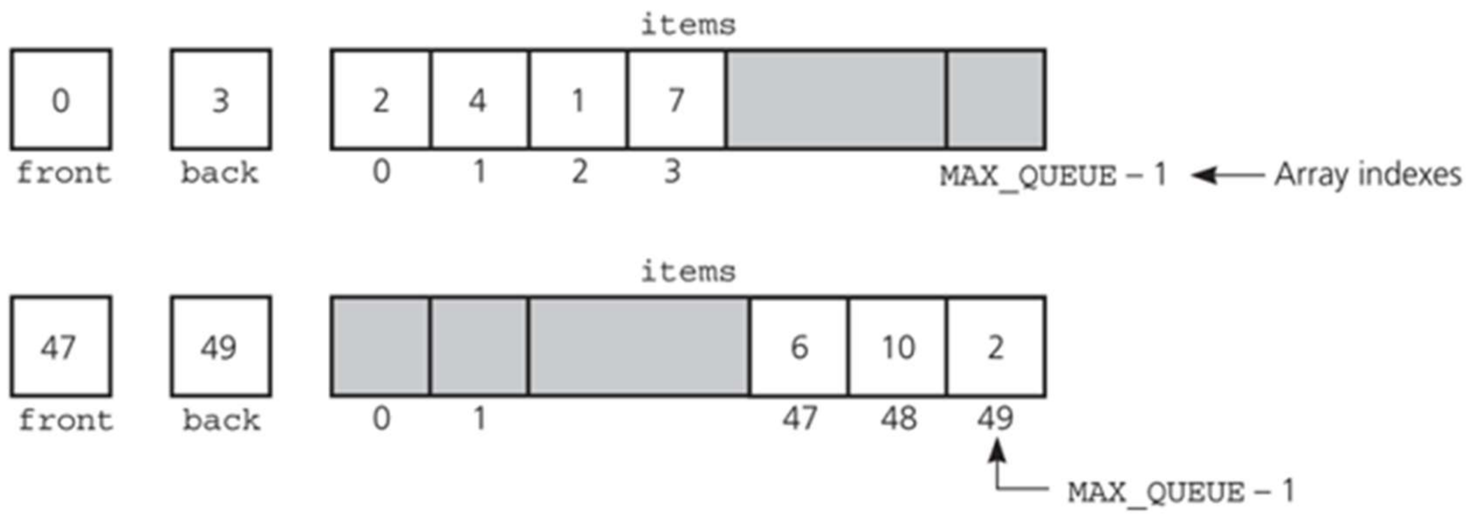
Queue as a circular linked list with one external pointer



Queue Operations

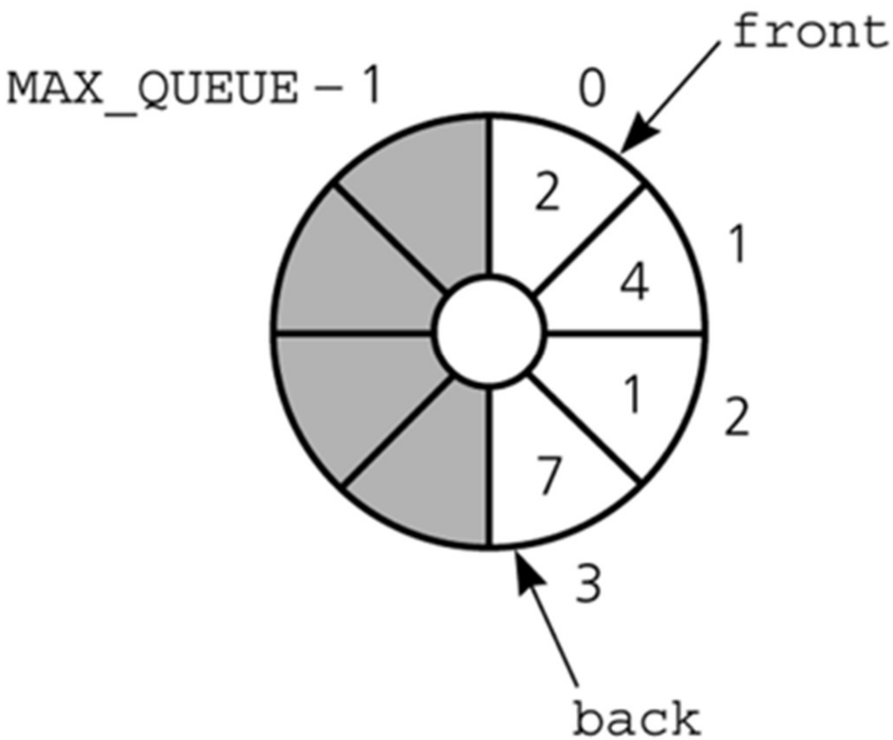
constructor ?
isEmpty ?
enqueue ?
dequeue ?
getFront ?

A Naive Array-Based Implementation of Queue



- Rightward drift can cause the queue to appear full even though the queue contains few entries.
- We may shift the elements to left in order to compensate for rightward drift, but shifting is expensive (O(n))

A Circular Array-Based Implementation



When either **front** or **back** advances past **MAX_QUEUE-1** it wraps around to 0.

Circular array eliminates rightward drift.

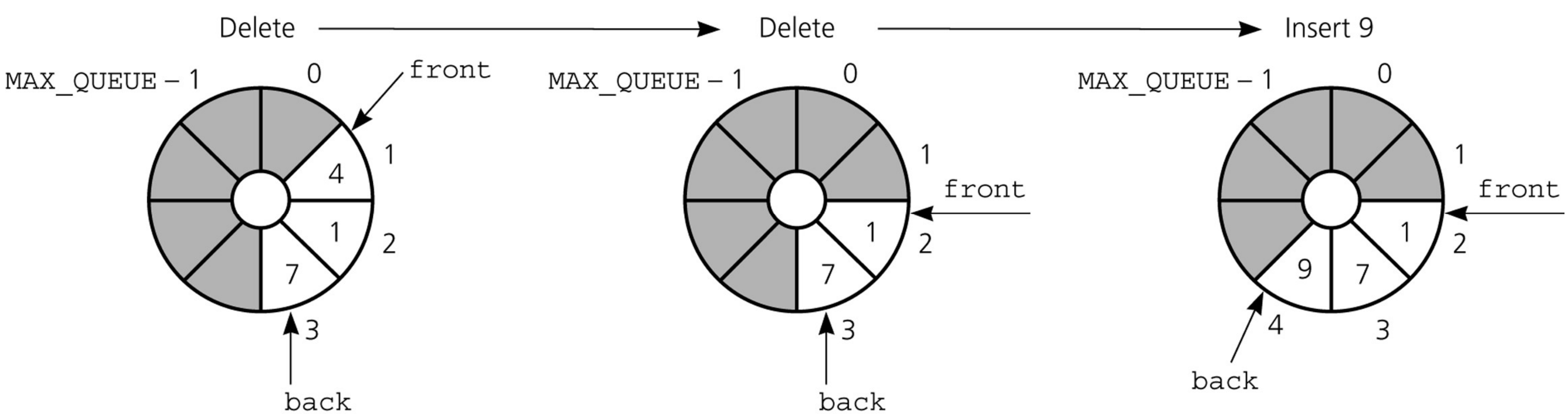
The effect of some operations of the queue

Initialize: `front=0;  back=MAX_QUEUE-1;`

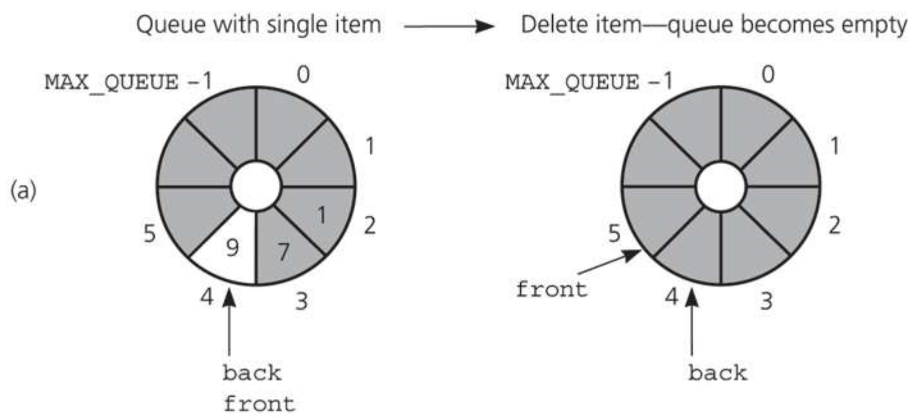
Insertion : `back = (back+1) % MAX_QUEUE;`
`items[back] = newItem;`

NOT ENOUGH

Deletion : `front = (front+1) % MAX_QUEUE;`

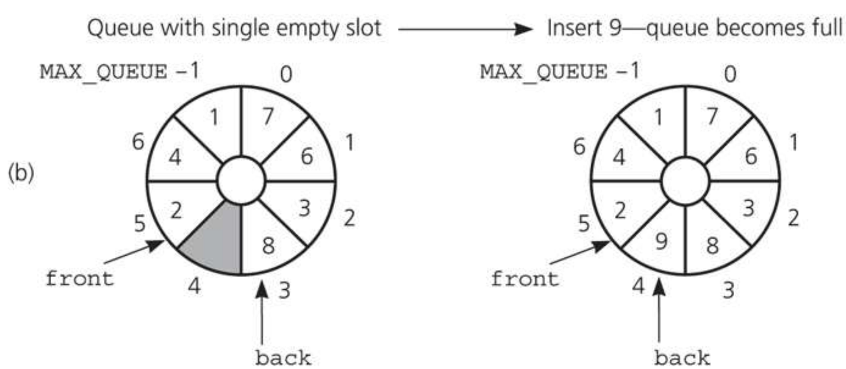


PROBLEM – Queue is Empty or Full



? Empty

`(back+1)%MAX_QUEUE == front`



? Full

`(back+1)%MAX_QUEUE == front`

`front` and **`back`** cannot be used to distinguish between ***queue-full*** and ***queue-empty*** conditions.

Solutions for Queue-Empty/Queue-Full Problem

1. Using a counter to keep the number items in the queue.
   - Initialize count to 0 during creation; Increment count by 1 during insertion; Decrement count by 1 during deletion.
   - count=0 ➔ empty; count=MAX_QUEUE ➔ full
2. Using `isFull` flag to distinguish between the full and empty conditions.
   - When the queue becomes full, set `isFull` to true; When the queue is not full set `isFull` to false;
3. Using an extra array location (and leaving at least one empty location in the queue). (***MORE EFFICIENT***)
   - Declare MAX_QUEUE+1 locations for the array items, but only use MAX_QUEUE of them. We do not use one of the array locations.

Using a counter

- To initialize the queue, set
  - `front` to `0`
  - `back` to `MAX_QUEUE-1`
  - `count` to `0`
- Inserting into a queue

```
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;
```

- Deleting from a queue

```
front = (front+1) % MAX_QUEUE;
--count;
```

- Full: `count == MAX_QUEUE`
- Empty: `count == 0`

Solutions for Queue-Empty/Queue-Full Problem

1. Using a counter to keep the number items in the queue.
   - Initialize count to 0 during creation; Increment count by 1 during insertion; Decrement count by 1 during deletion.
   - count=0 ➔ empty; count=MAX_QUEUE ➔ full
2. Using `isFull` flag to distinguish between the full and empty conditions.
   - When the queue becomes full, set `isFull` to true; When the queue is not full set `isFull` to false;

Using isFull flag

- To initialize the queue, set

  ```
  front = 0;  back = MAX_QUEUE-1; isFull = false;
  ```

- Inserting into a queue

  ```
  back = (back+1) % MAX_QUEUE; items[back] = newItem;
  if ((back+1)%MAX_QUEUE == front)) isFull = true;
  ```

- Deleting from a queue

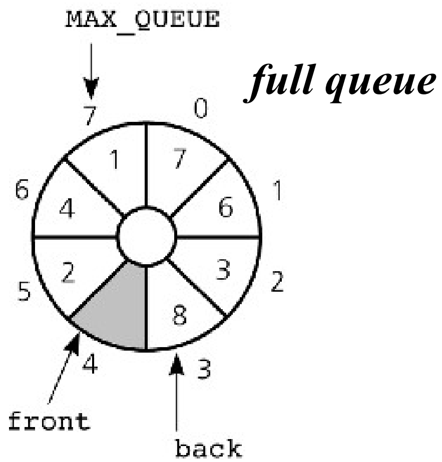
  ```
  front = (front+1) % MAX_QUEUE;
  isFull = false;
  ```

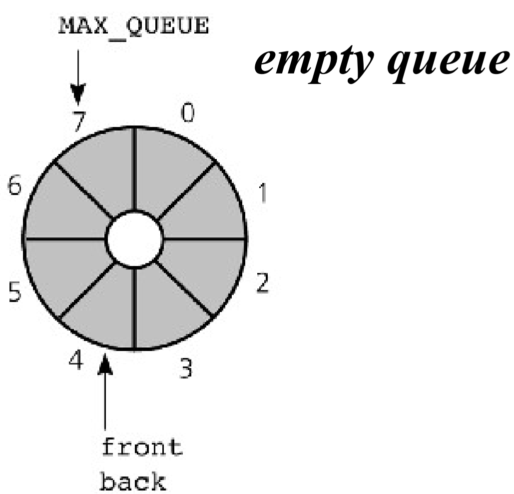
- Full: `isFull == true`
- Empty: `isFull==false && ((back+1)%MAX_QUEUE == front))`

Solutions for Queue-Empty/Queue-Full Problem

1. Using a counter to keep the number items in the queue.
   - Initialize count to 0 during creation; Increment count by 1 during insertion; Decrement count by 1 during deletion.
   - count=0 ➔ empty; count=MAX_QUEUE ➔ full
2. Using `isFull` flag to distinguish between the full and empty conditions.
   - When the queue becomes full, set `isFull` to true; When the queue is not full set `isFull` to false;
3. Using an extra array location (and leaving at least one empty location in the queue). (***MORE EFFICIENT***)
   - Declare MAX_QUEUE+1 locations for the array items, but only use MAX_QUEUE of them. We do not use one of the array locations.

Using an extra array location





- To initialize the queue, allocate (MAX_QUEUE+1) locations

```
front=0;  back=0;
```

- **`front`** holds the index of the location before the front of the queue.
- Inserting into a queue (if queue is not full)

```
back = (back+1) % (MAX_QUEUE+1);
items[back] = newItem;
```

- Deleting from a queue (if queue is not empty)

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

- Full:

```
(back+1)%(MAX_QUEUE+1) == front
```

- Empty:

```
front == back
```

Array-Based Implementation Using a counter – Header File

```
#include "QueueException.h"
const int MAX_QUEUE = maximum-size-of-queue;

template <class T>
class Queue {
public:
   Queue();  // default constructor
   bool isEmpty() const;
   void enqueue(const T& newItem) throw(QueueException);
   T dequeue() throw(QueueException);
   T peek() const throw(QueueException);
private:
   T items[MAX_QUEUE];
        int front;
        int back;
        int count;
};
```

Array-Based Implementation Using a counter – constructor, isEmpty

```
template<class T>
Queue<T>::Queue():front(0), back(MAX_QUEUE-1), count(0) {}


template<class T>
bool Queue<T>::isEmpty() const
{
   return count == 0;
}
```

Array-Based Implementation Using a counter - enqueue

```
template<class T>
void Queue<T>::enqueue(const T& newItem)
   throw(QueueException) {
    if (count == MAX_QUEUE)
        throw QueueException("QueueException: queue full on
   enqueue");
    else {    // queue is not full; insert item
        back = (back+1) % MAX_QUEUE;
        items[back] = newItem;
        ++count;
    }
}
```

Array-Based Implementation Using a counter – dequeue

```
template<class T>
T Queue<T>::dequeue() throw(QueueException) {
   if (isEmpty())
      throw QueueException("QueueException: empty queue, cannot
   dequeue");
   else {  // queue is not empty; remove front
      T val = items[front];
      front = (front+1) % MAX_QUEUE;
      --count;
      return val;
   }
}
```

Array-Based Implementation Using a counter – peek

```
template <class T>
T Queue<T>::peek () const throw(QueueException)
{
   if (isEmpty())
      throw QueueException("QueueException: empty queue, cannot
   getFront");
   else
      // queue is not empty; retrieve front
      return(items[front]);
}
```

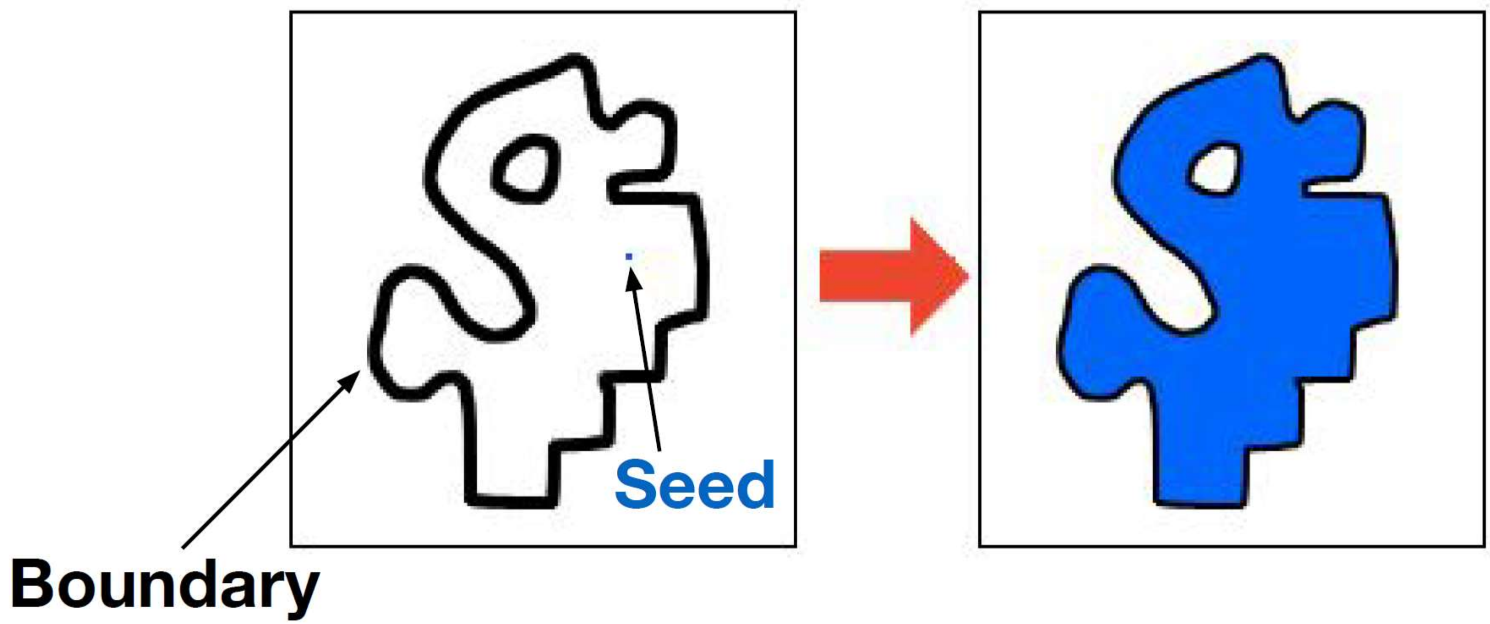
Stacks vs. Queues

- **Stacks:**
  - LIFO (Last-In-First-Out)
  - Push and pop both modify the top element
  - Computer systems use stacks to manage function calls, including recursive function 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

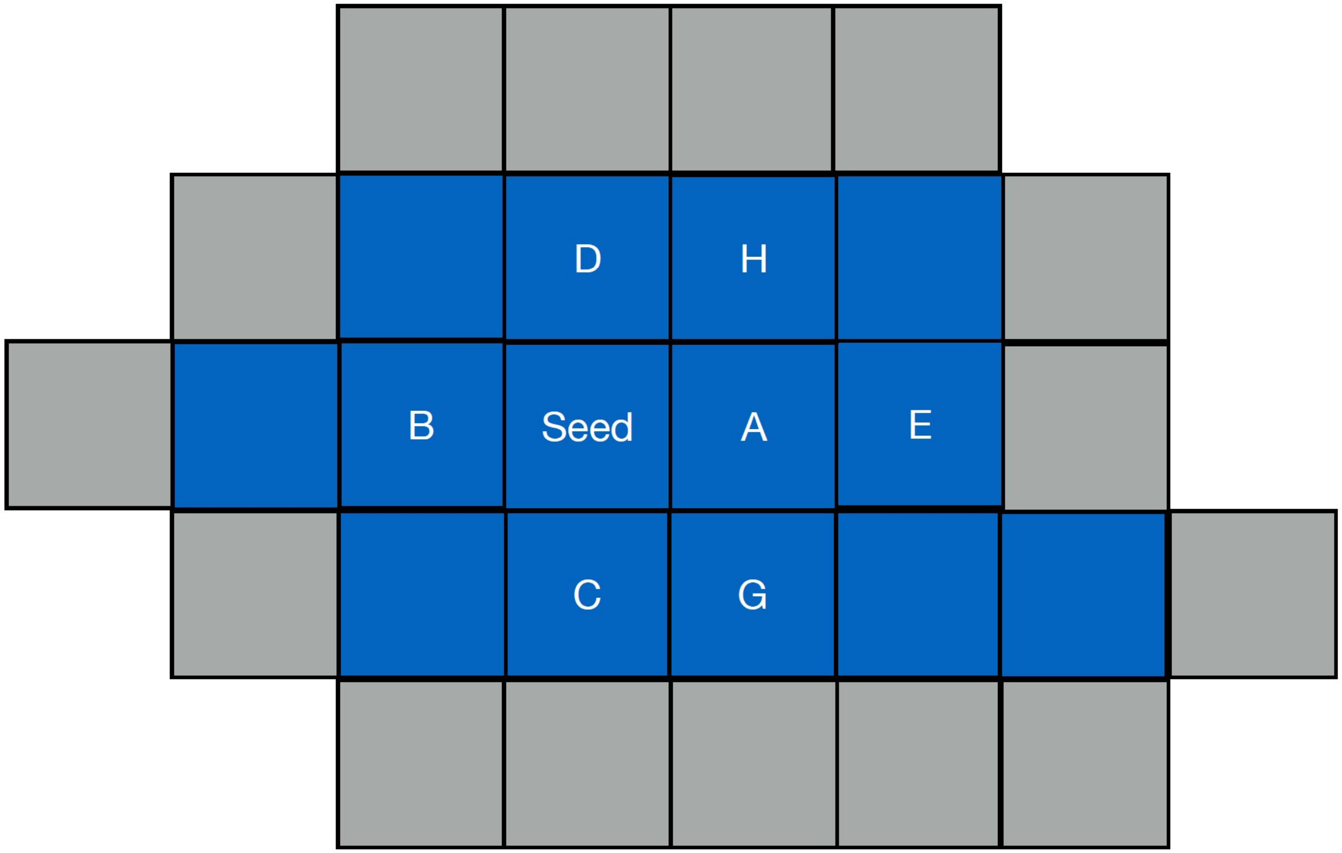
- A common tool in many paint software, used to fill a **connected** region of pixels with a different color.
- Also known as Bucket Fill, or Seed Fill. Example:



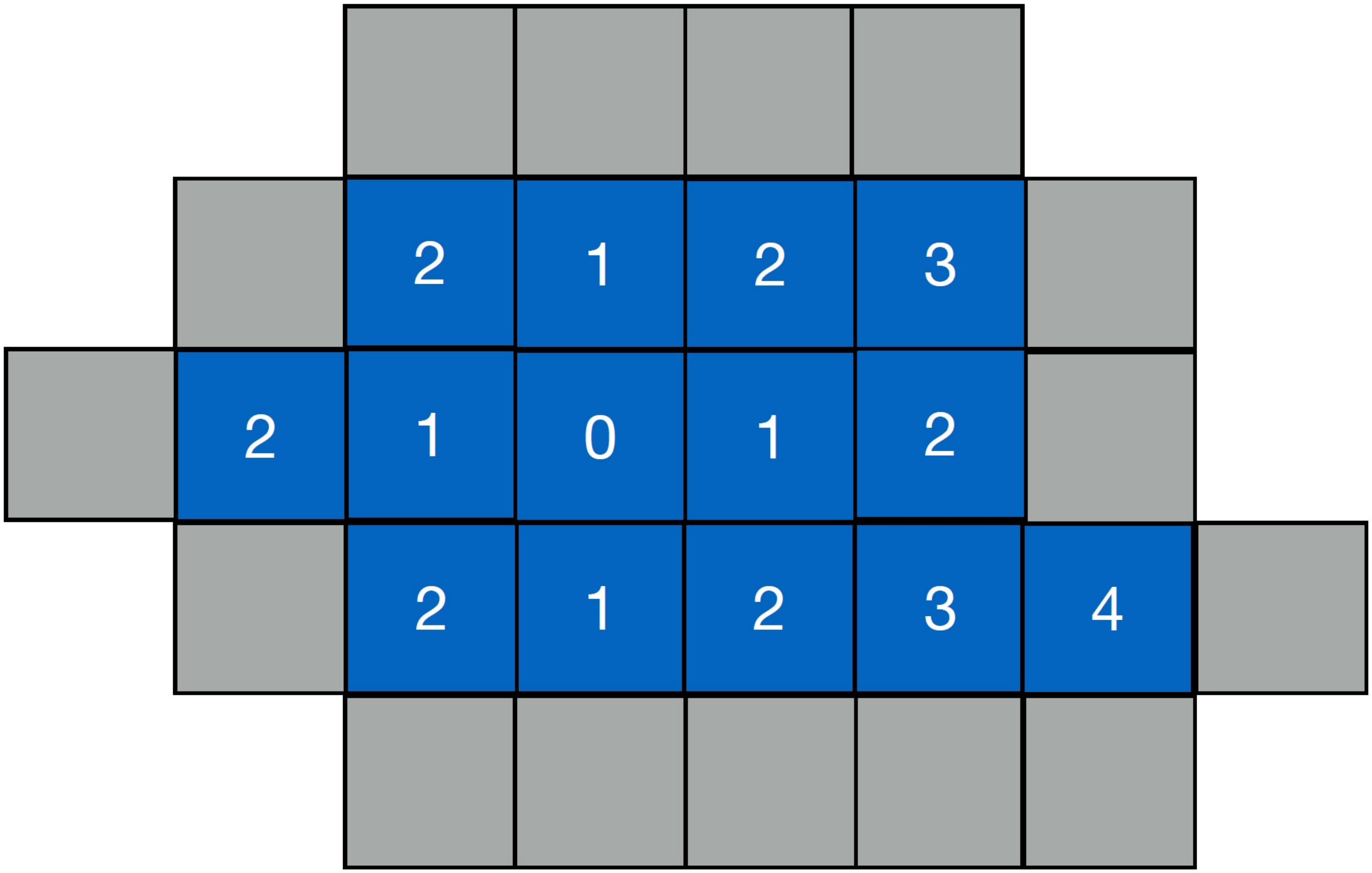
Flood Fill With a Queue

- 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



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



Searching With Queues vs. Stacks

- 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.

- We will study more about these search methods in the future.