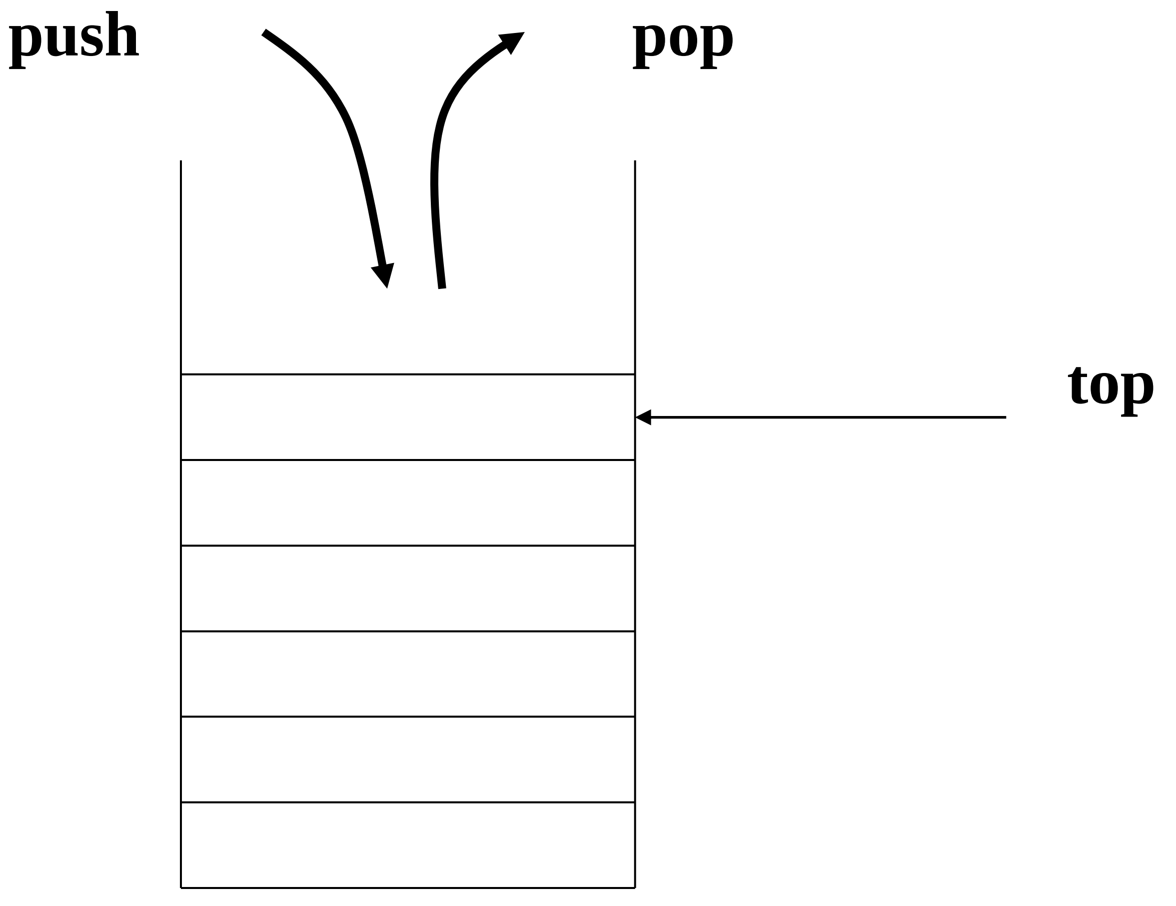
Stacks and Queues

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  – Data stored
  – Operations on the data
  – Error conditions associated with operations

The Stack ADT

- The Stack ADT stores arbitrary objects.
- Insertions and deletions follow the *last-in first-out* (LIFO) scheme.
- It is like a stack of trays:
  - Trays can be added to the top of the stack.
  - Trays can be removed from the top of the stack.
- Main stack operations:
  - **push**(object o): inserts element o
  - **pop**(): removes and returns the last inserted element



Stack

The Stack ADT

- Auxiliary stack operations:
  - **top**(): returns a reference to the last inserted element without removing it
  - **size**(): returns the number of elements stored
  - **isEmpty**(): returns a Boolean value indicating whether no elements are stored

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException.

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Stacks and Computer Languages

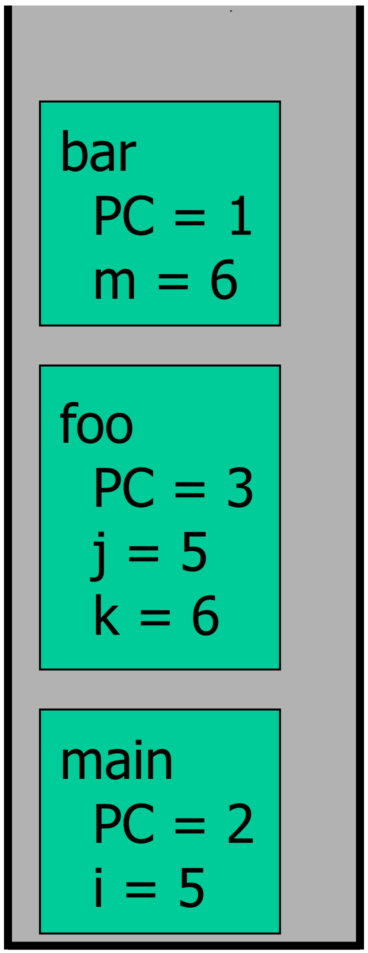
- A stack can be used to check for unbalanced symbols (e.g. matching parentheses)
- <u>Algorithm</u>

1. Make an empty stack.
2. Read symbols until the end of file.
   a. If the token is an opening symbol, push it onto the stack.
   b. If it is a closing symbol and the stack is empty, report an error.
   c. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, report an error.
3. At the end of the file, if the stack is not empty, report an error.

C++ Run-time Stack

- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```

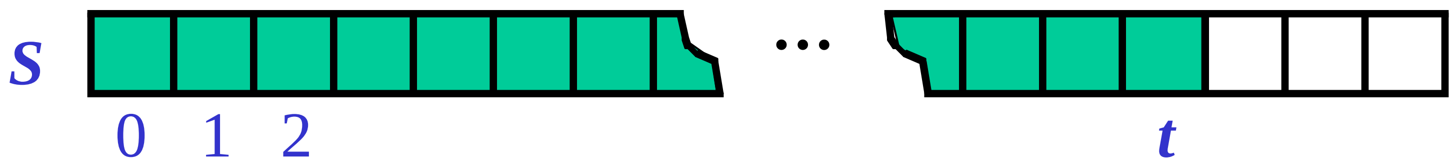


Array-based Stack

- A simple way of implementing the Stack ADT uses an array.
- We add elements from left to right.
- A variable keeps track of the index of the top element

Algorithm *size*()
  return $t + 1$

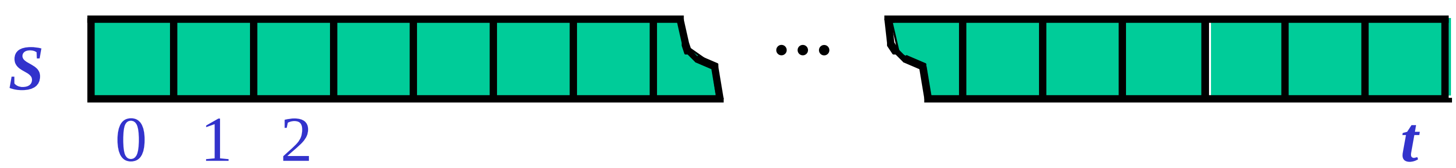
Algorithm *pop*()
  if *isEmpty*() **then**
    throw *EmptyStackException*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm ***push***(***o***)
  if $t = S.length - 1$ **then**
    throw ***FullStackException***
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined *a priori* , and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Stack Interface in C++

```
template <class Object>
class Stack
{
   public:
        Stack(int c = 1000);
        int size() const;
        bool isEmpty( ) const;
        const Object & top()const throw(StackEmptyException);
        void push(const Object & x) throw(StackFullException);
        Object pop() throw(StackEmptyException);
    private:
        int capacity;          // stack capacity
        Object *S;             // stack array
        int top;               // top of stack
};
```

Array-based Stack in C++

```
// Stack class implementation
Stack(int c) {
   capacity = c;
   S = new Object[capacity];
   top =  -1;
}
int size() const {
   return (top + 1);
}
bool isEmpty() const {
   return (top < 0);
}
```

Stack Implementation (cont.)

```
Object& top() throw(StackEmptyException) {
   if (isEmpty())
      throw StackEmptyException("Access to empty stack");
   return S[top]; }

void push(const Object& elem) throw(StackFullException) {
   if (size() == capacity)
      throw StackFullException("Stack overflow");
   S[++top] = elem;
}

Object pop() throw(StackEmptyException) {
   if (isEmpty())
      throw StackEmptyException("Access to empty stack");
   return S[top--];
}
```

Example

- *Reading a line of text and writing it out backwards.*

```
int main( )
{
  Stack<char> s;
  char c;
  while ((c=getchar() )!='\n')
       s.push(c);

  while( !s.isEmpty( ) )
   cout << s.pop( ) << endl;

  return 0;
}
```

A Simple Calculator

- Calculators can evaluate infix expressions, such as 5 + 2.
- In an infix expression a binary operator has arguments to its left and right.
  - e.g. 1 + 2 * 3

    9 – 5 –3

    2 ^ 3 ^ 2
- When there are several operators, precedence and associativity determine how the operators are processed.

  10 – 3 – 2 ^ 3 * 4 / 5 / 10 ^ 2

Postfix Machines

- In a postfix expression a binary operator follows its operands.
  - e.g. 5 2 +

    1 2 3 * +

    10 3 – 2 3 ^ 4 * 5 / 10 2 ^ / -
- A postfix expression can be evaluated as follows:
  - Operands are pushed into a single stack.
  - An operator pops its operands and then pushes the result.
  - At the end of the evaluation, the stack should contain only one element, which represents the result.

Example

- Evaluate the following postfix expression.

```
8 5 4 * 5 6 2 / + − 2 / +
```

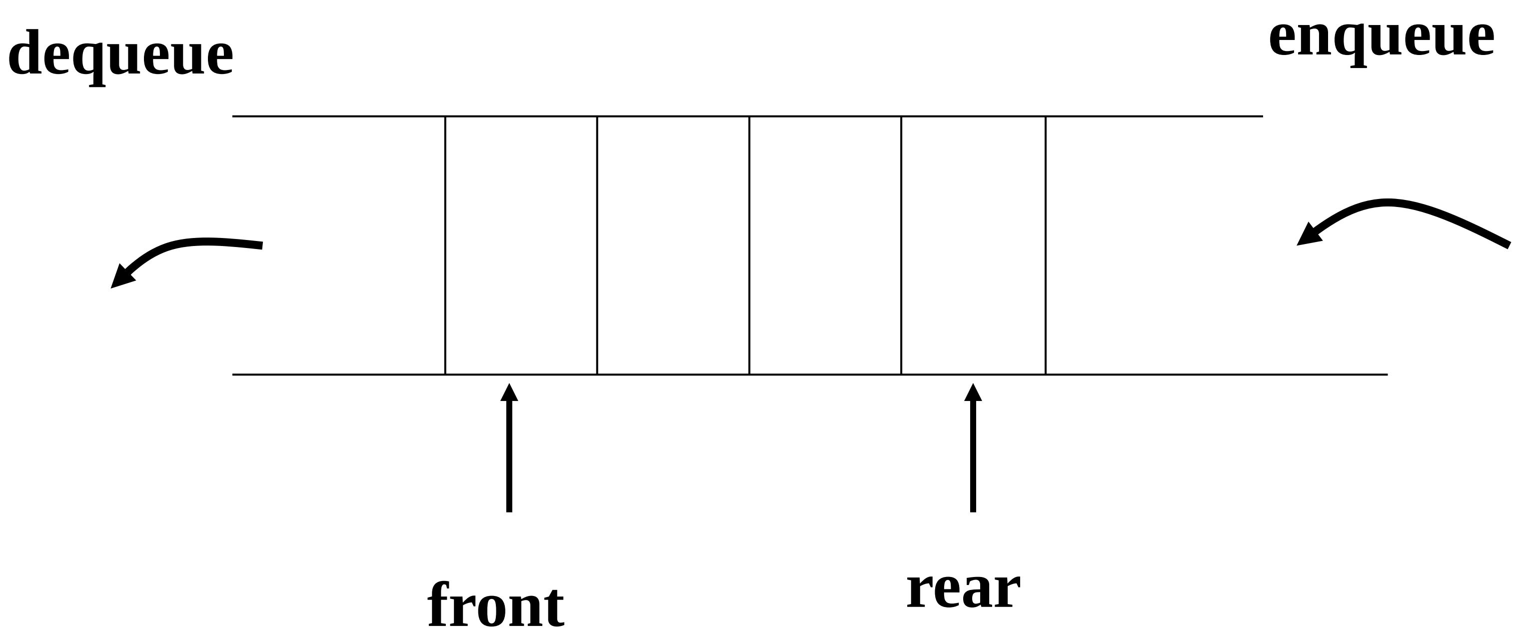
Linked list implementation of Stacks

- In implementing Stack as a linked list the top of the stack is represented by the first item in the linked list.
- To implement push: create a new node and attach it as the new first node.
- To implement pop: advance the top of stack to the second item in the list (if there is one).
- Each operation is performed in constant time.
- See chapter 16 for details.

The Abstract Data Type Queue

- A ***queue*** is a list from which items are deleted from one end (**front**) and into which items are inserted at the other end (**rear,** or **back**)
  - It is like line of people waiting to purchase tickets:
- ***Queue*** is referred to as a **first-in-first-out (FIFO)** data structure.
  - The first item inserted into a queue is the first item to leave
- Queues have many applications in computer systems:
  - Any application where a group of items is waiting to use a shared resource will use a queue. e.g.
    - jobs in a single processor computer
    - print spooling
    - information packets in computer networks.

A Queue



ADT Queue Operations

- ***createQueue()***
  - Create an empty queue
- ***destroyQueue()***
  - Destroy a queue
- ***isEmpty():boolean***
  - Determine whether a queue is empty
- ***enqueue(in newItem:QueueItemType)***
  - Inserts a new item at the end of the queue (at the **rear** of the queue)
- ***dequeue() throw QueueException***

  dequeue(out queueFront:QueueItemType)
  - Removes (and returns) the element at the **front** of the queue
- ***getFront(out queueFront:QueueItemType)***
  - Retrieve the item that was added earliest (without removing)

Some Queue Operations

Operation operation	***Queue after***	
x.createQueue()	an empty queue	
	front ↓	
x.enqueue(5)	5	
x.enqueue(3)	5 3	
x.enqueue(2)	5 3 2	
x.dequeue()	3 2	
x.enqueue(7)	3 2 7	
x.dequeue(a)	2 7	(a is 3)
x.getFront(b)	2 7	(b is 2)

An Application -- Reading a String of Characters

- A queue can retain characters in the order in which they are typed

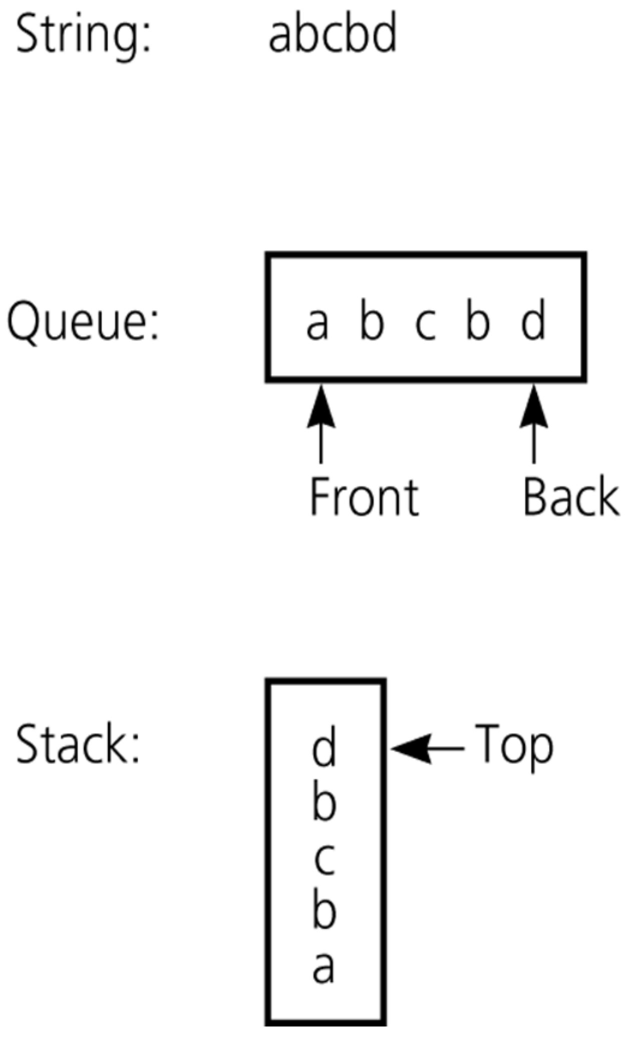
```
aQueue.createQueue()
while (not end of line) {
   Read a new character ch
   aQueue.enqueue(ch)
}
```

- Once the characters are in a queue, the system can process them as necessary

Recognizing Palindromes

- A palindrome
  - A string of characters that reads the same from left to right as its does from right to left
- To recognize a palindrome, a queue can be used in conjunction with a stack
  - A stack reverses the order of occurrences
  - A queue preserves the order of occurrences
- A nonrecursive recognition algorithm for palindromes
  - As you traverse the character string from left to right, insert each character into both a queue and a stack
  - Compare the characters at the front of the queue and the top of the stack

Recognizing Palindromes (cont.)



The results of inserting a string into both a queue and a stack

Recognizing Palindromes

```
isPal(in str:string) : boolean          // Determines whether str is a palindrome or
    not
    aQueue.createQueue();    aStack.createStack();
    len = length of str;
    for (i=1 through len) {
            nextChar = ith character of str;
            aQueue.enqueue(nextChar);
            aStack.push(nextChar);
    }
    charactersAreEqual = true;
    while (aQueue is not empty and charactersAreEqual) {
            aQueue.getFront(queueFront);
            aStack.getTop(stackTop);
            if (queueFront equals to stackTop) { aQueue.dequeue();  aStack.pop()};
    }
            else chractersAreEqual = false; }
    return charactersAreEqual;
```

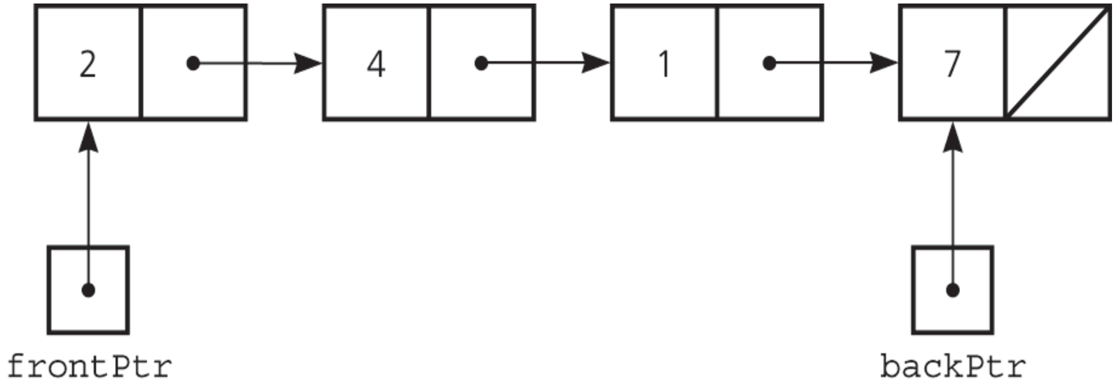
Implementations of the ADT Queue

- Pointer-based implementations of queue
  - A linear 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

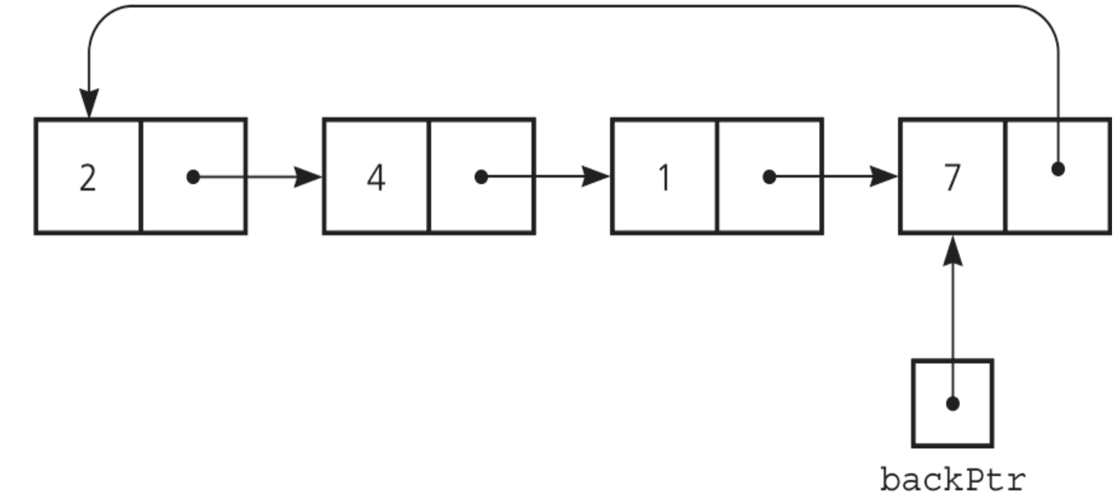
Pointer-based implementations of queue

(a)



a linear linked list with two external pointers

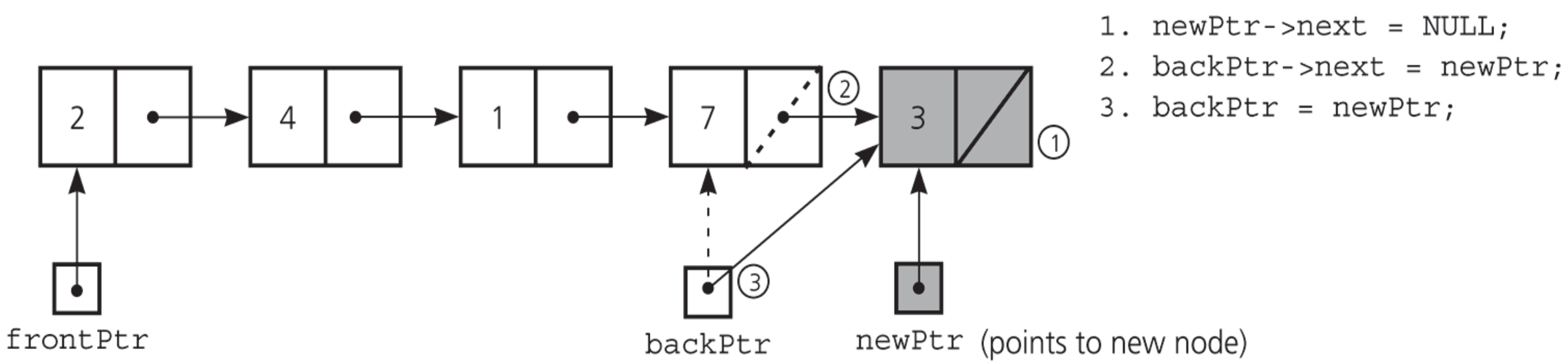
(b)



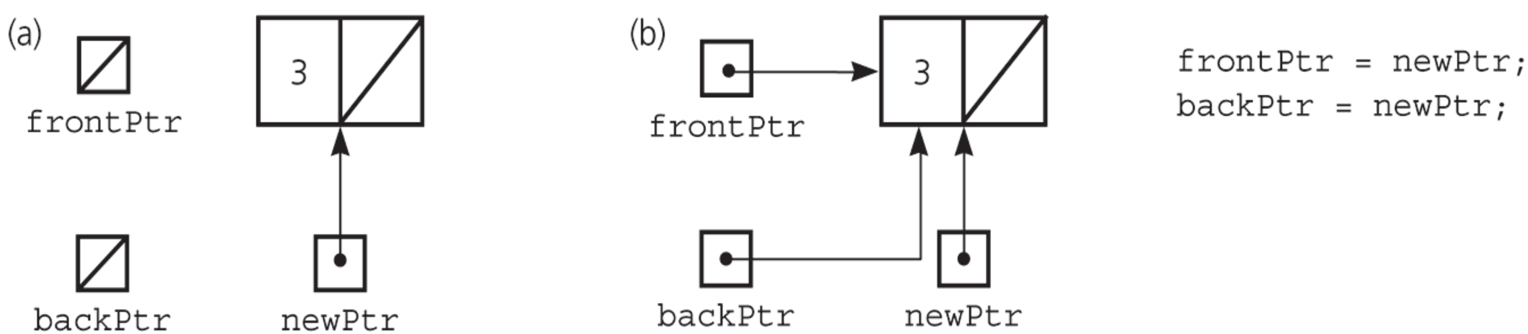
a circular linear linked list with one external pointer

Pointer-Based Implementation -enqueue

Inserting an item into a nonempty queue



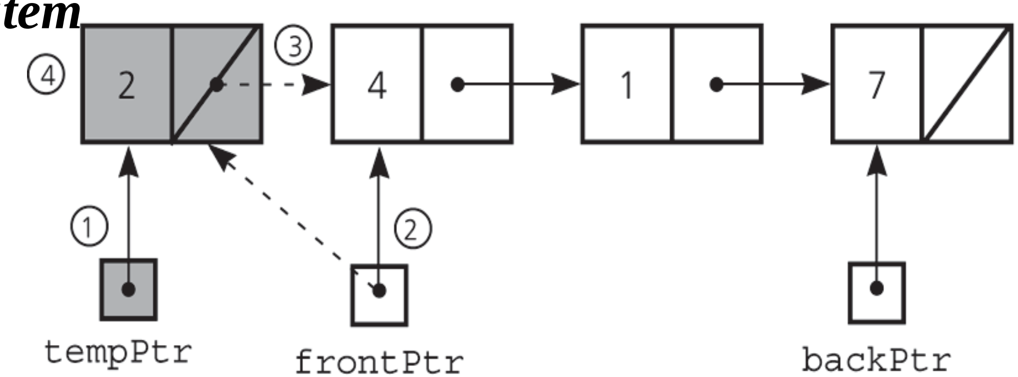
Inserting an item into an empty queue



a) before insertion *b) after insertion*

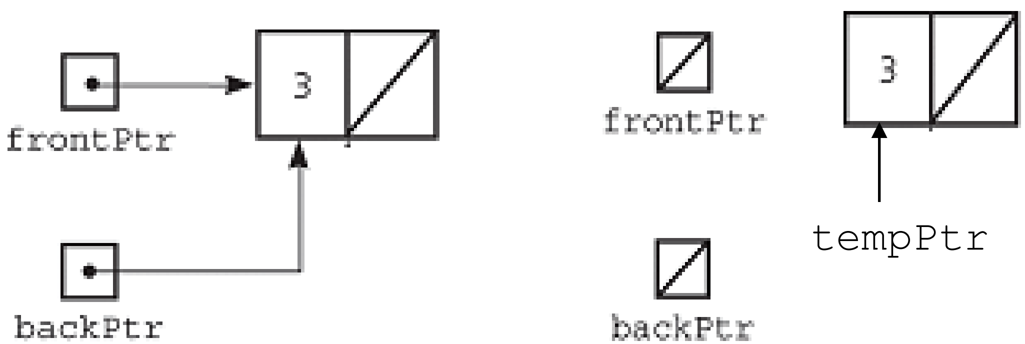
Pointer-Based Implementation -- dequeue

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;
```

Header File

```
#include "QueueException.h"
typedef desired-type-of-queue-item QueueItemType;
class Queue {
public:
    Queue();                        // default constructor
    Queue(const Queue& Q);          // copy constructor
    ~Queue();                        // destructor

    bool isEmpty() const;   // Determines whether the queue is empty.
    void enqueue(QueueItemType newItem);  // Inserts an item at the back
    of a queue.
    void dequeue() throw(QueueException);   // Dequeues the front of a
    queue.
            // Retrieves and deletes the front of a queue.
    void dequeue(QueueItemType& queueFront) throw(QueueException);
            // Retrieves the item at the front of a queue.
    void getFront(QueueItemType& queueFront) const
    throw(QueueException);
```

Header File

```
private:
    // The queue is implemented as a linked list with one external pointer
    // to the front of the queue and a second external pointer to the back
    // of the queue.
    struct QueueNode
    {       QueueItemType  item;
            QueueNode      *next;
     }; // end struct
     QueueNode *backPtr;
     QueueNode *frontPtr;
}
```

constructor, deconstructor, isEmpty

```
#include "QueueP.h"  // header file

Queue::Queue() : backPtr(NULL), frontPtr(NULL){}  // default
    constructor

Queue::~Queue() {            // destructor
   while (!isEmpty())
      dequeue();       //  backPtr  and frontPtr are NULL at this point
}

bool Queue::isEmpty() const{          // isEmpty
   return backPtr == NULL;
}
```

enqueue

```
void Queue::enqueue(QueueItemType newItem) {     // enqueue
    // create a new node
    QueueNode *newPtr = new QueueNode;

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

    // 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
}
```

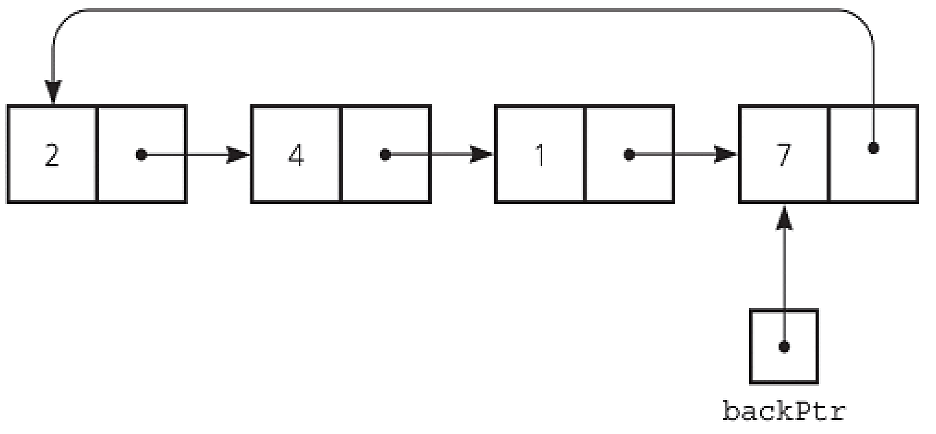
dequeue

```
void Queue::dequeue() throw(QueueException) {
  if (isEmpty())
    throw QueueException("QueueException: empty queue, cannot
   dequeue");
  else {   // queue is not empty; remove front
    QueueNode *tempPtr = frontPtr;
    if (frontPtr == backPtr) {          // one node in queue
      frontPtr = NULL;
      backPtr = NULL;
    }
    else
      frontPtr = frontPtr->next;

    tempPtr->next = NULL;               // defensive strategy
    delete tempPtr;
  }}
```

dequeue, getFront

```
void Queue::dequeue(QueueItemType& queueFront)
    throw(QueueException) {
  if (isEmpty())
    throw QueueException("QueueException: empty queue, cannot
    dequeue");
  else {  // queue is not empty; retrieve front
    queueFront = frontPtr->item;
    dequeue();  // delete front
  }}

void Queue::getFront(QueueItemType& queueFront) const
    throw(QueueException) {
  if (isEmpty())
    throw QueueException("QueueException: empty queue, cannot
    getFront");
  else    // queue is not empty; retrieve front
    queueFront = frontPtr->item;
}
```

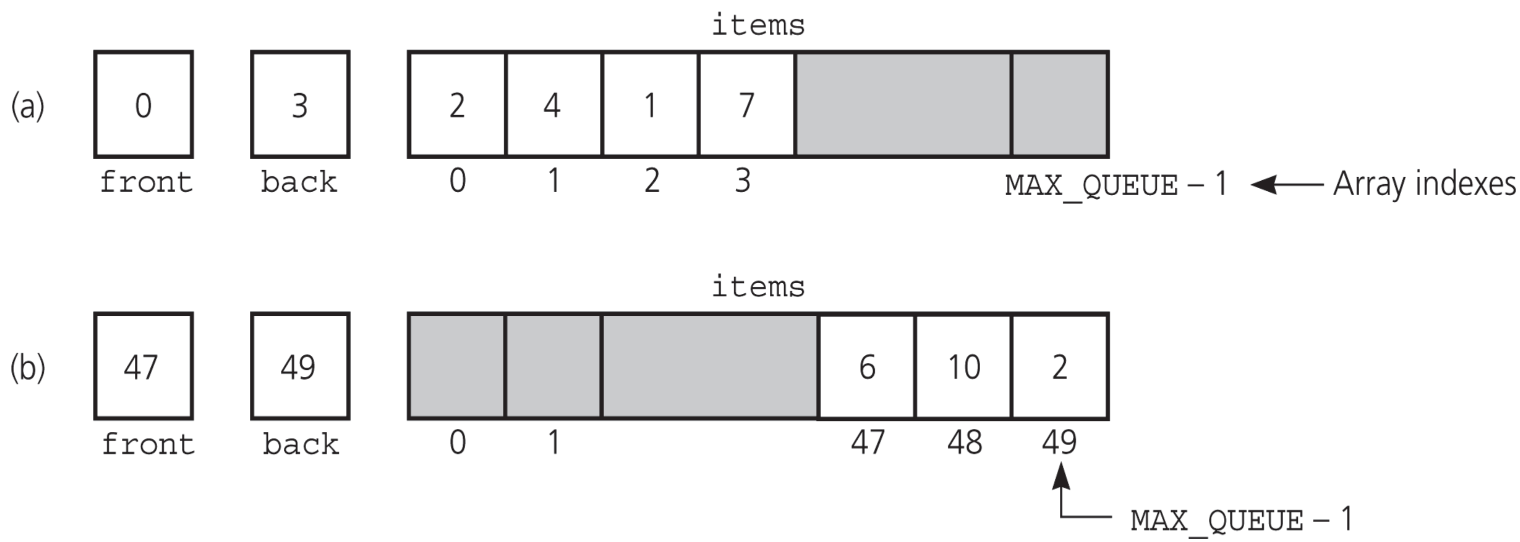
A circular linked list with one external pointer



Queue Operations

constructor ?
isEmpty ?
enqueue ?
dequeue ?
getFront ?

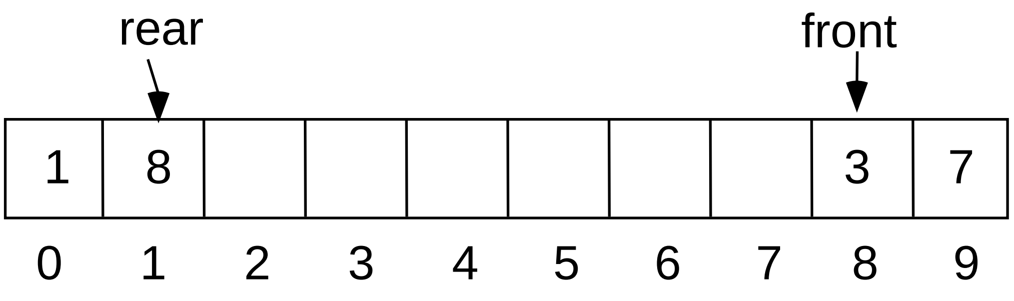
A Naive Array-Based Implementation of Queue



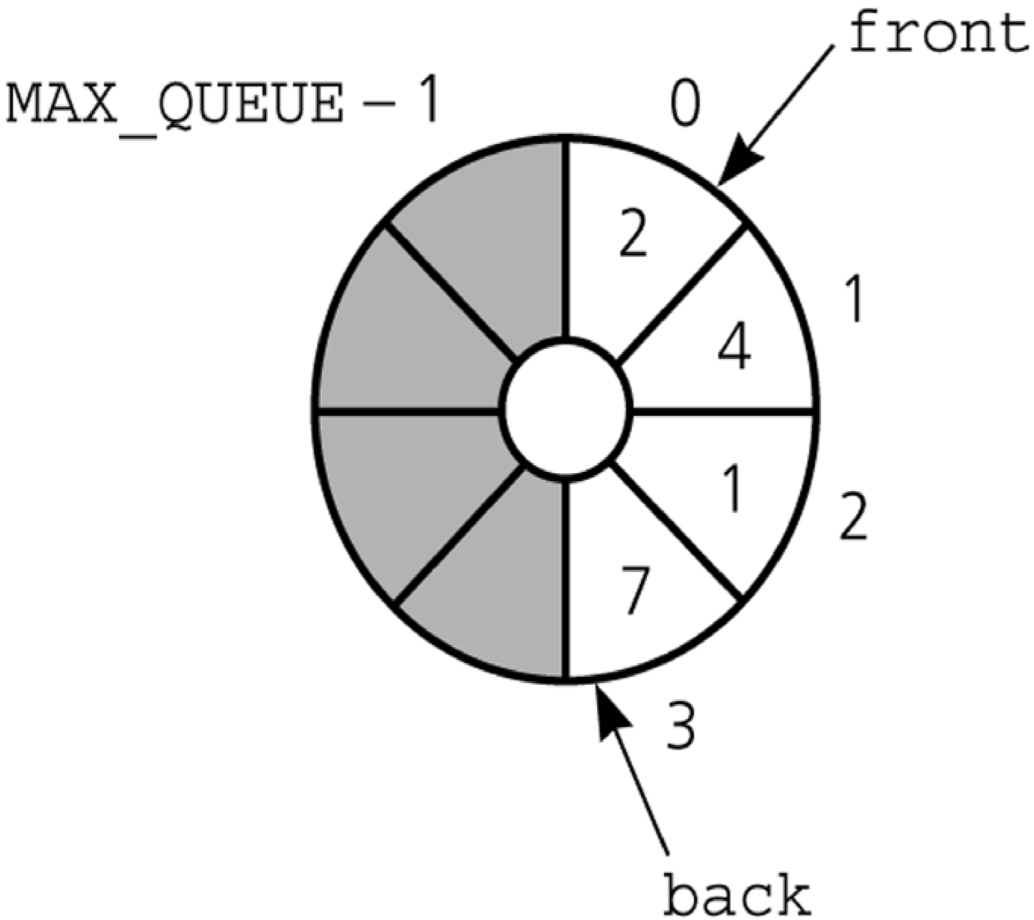
- Rightward drift can cause the queue to appear full even though the queue contains few entries.
- We may shift the elements to left in order to compensate for rightward drift, but shifting is expensive
- **Solution:** A circular array eliminates rightward drift.

Circular Array Implementation

- The front and rear are the same as the basic model, except: The queue wraps around when the end of the array is reached.



A Circular Array-Based Implementation



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

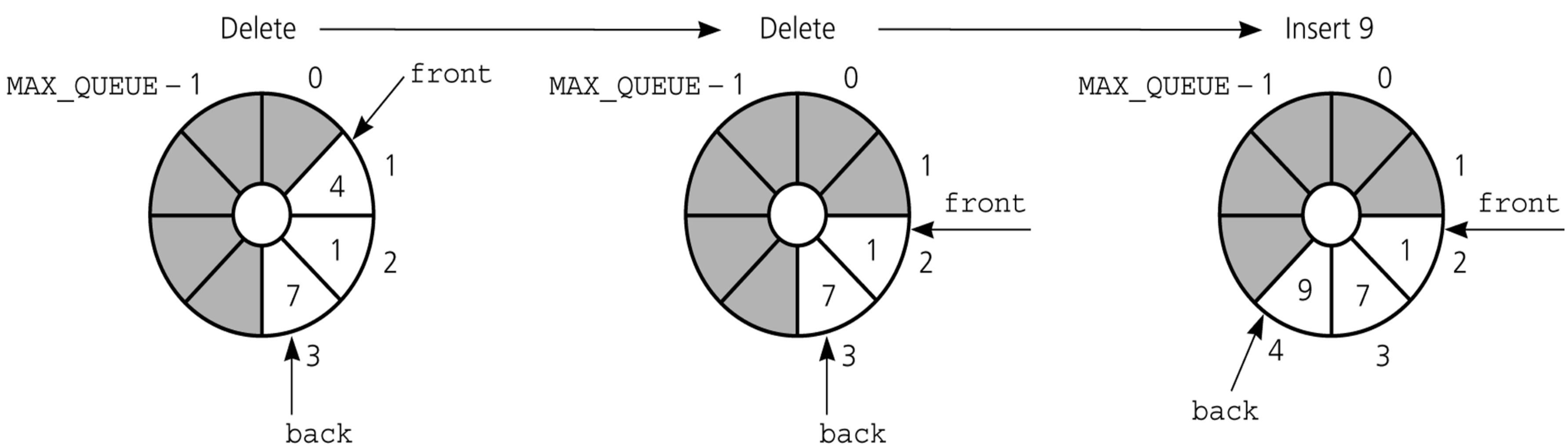
The effect of some operations of the queue

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

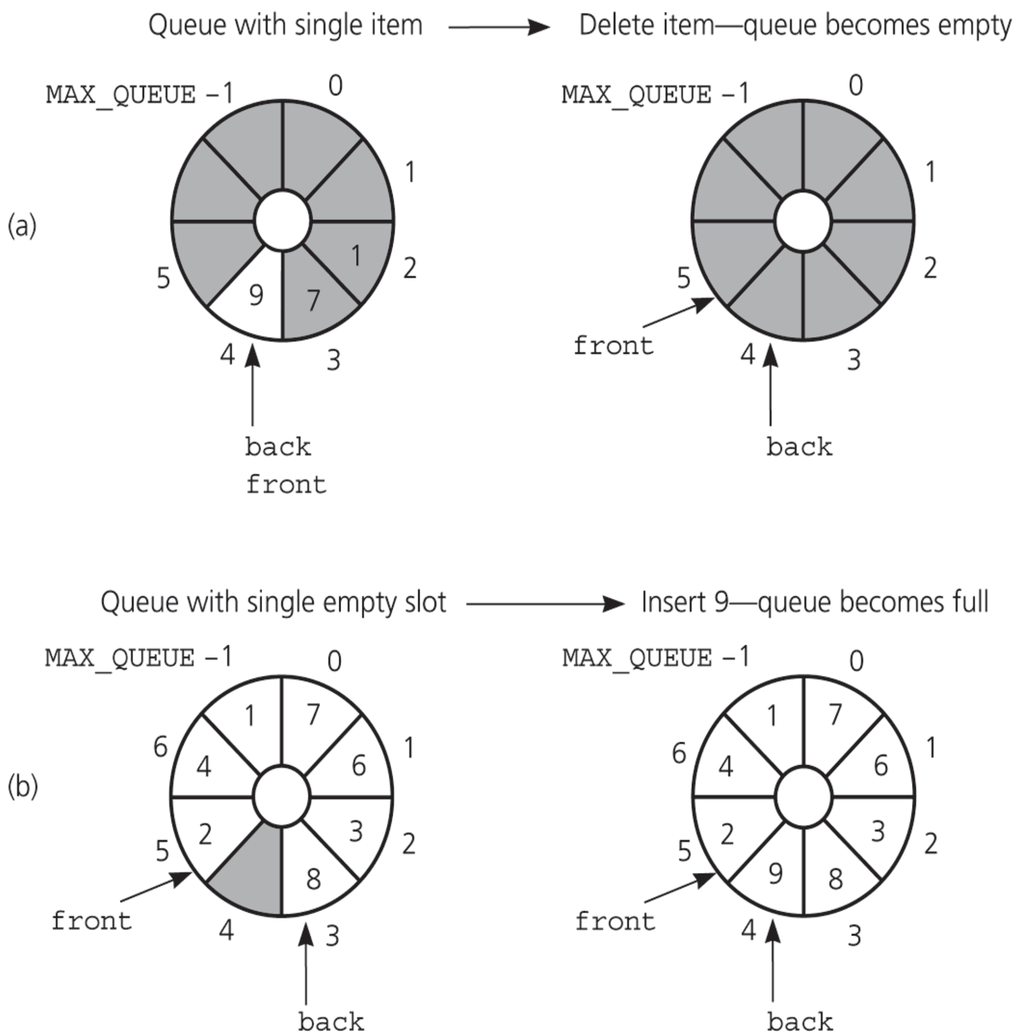
Insertion : `back = (back+1) % MAX_QUEUE;`
`items[back] = newItem;`

NOT ENOUGH

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



PROBLEM – Queue is Empty or Full



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

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

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

So, we need extra mechanism 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 isFullFlag to true; When the queue is not full set isFull flag 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.
   - *Full*: front equals to (back+1)%(MAX_QUEUE+1)
   - *Empty*: front equals to back

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`

Array-Based Implementation Using a counter – Header File

```
#include "QueueException.h"
const int MAX_QUEUE = maximum-size-of-queue;
typedef desired-type-of-queue-item QueueItemType;
class Queue {
public:
      Queue();  // default constructor
      bool isEmpty() const;
      void enqueue(QueueItemType newItem) throw(QueueException);
      void dequeue() throw(QueueException);
      void dequeue(QueueItemType& queueFront) throw(QueueException);
      void getFront(QueueItemType& queueFront) const throw(QueueException);
private:
      QueueItemType items[MAX_QUEUE];
      int          front;
      int          back;
      int          count;
};
```

constructor, isEmpty, enqueue

```
Queue::Queue():front(0), back(MAX_QUEUE-1), count(0) {}

bool Queue::isEmpty() const {
  return count == 0);
}

void Queue::enqueue(QueueItemType 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;
  }
}
```

dequeue

```
void Queue::dequeue() throw(QueueException) {
  if (isEmpty())
    throw QueueException("QueueException: empty queue, cannot
    dequeue");
  else {  // queue is not empty; remove front
    front = (front+1) % MAX_QUEUE;
    --count;
  }}

void Queue::dequeue(QueueItemType& queueFront)
    throw(QueueException) {
  if (isEmpty())
    throw QueueException("QueueException: empty queue, cannot
    dequeue");
  else {  // queue is not empty; retrieve and remove front
    queueFront = items[front];
    front = (front+1) % MAX_QUEUE;
    --count;
  }}
```

dequeue

```
void Queue::getFront(QueueItemType& queueFront) const
    throw(QueueException) {
  if (isEmpty())
    throw QueueException("QueueException: empty queue, cannot
    getFront");
  else
    // queue is not empty; retrieve front
    queueFront = items[front];
}
```

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;`

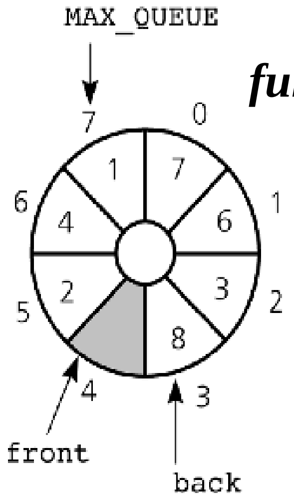
- Deleting from a queue

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

  `isFull = false;`

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

Using an extra array location



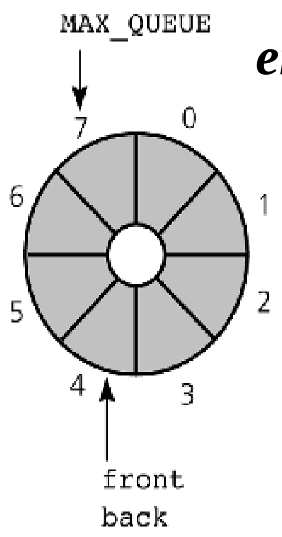
full queue

- 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;
```



empty queue

- 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
```

Comparing Implementations

- Fixed size versus dynamic size
  - A statically allocated array
    - Prevents the `enqueue` operation from adding an item to the queue if the array is full
  - A resizable array or a reference-based implementation
    - Does not impose this restriction on the `enqueue` operation
- Pointer-based implementations
  - A linked list implementation
    - More efficient; no size limit

A Summary of Position-Oriented ADTs

- Position-oriented ADTs: List, Stack, Queue
- Stacks and Queues
  - Only the end positions can be accessed
- Lists
  - All positions can be accessed
- Stacks and queues are very similar
  - Operations of stacks and queues can be paired off as
    - `createStack` and `createQueue`
    - Stack `isEmpty` and queue `isEmpty`
    - `push` and `enqueue`
    - `pop` and `dequeue`
    - Stack `getTop` and queue `getFront`