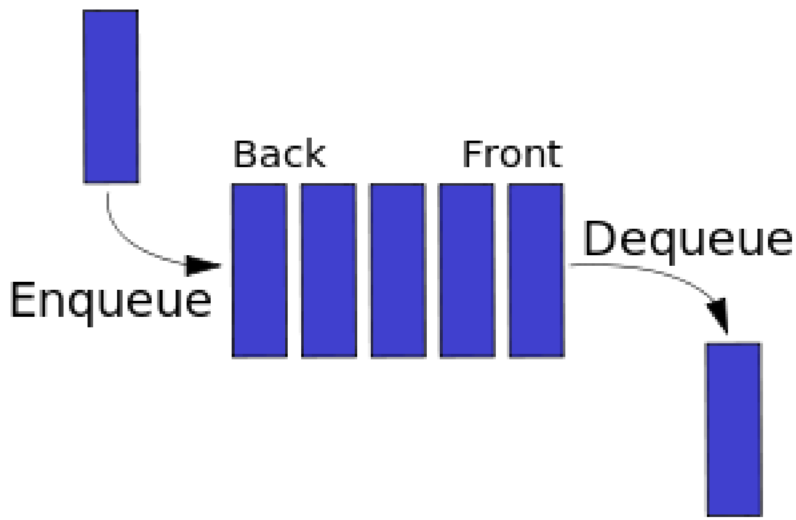
COM 201 – Data Structures and Algorithms

Abstract Data Types – Queue

Assist. Prof. Özge ÖZTİMUR KARADAĞ
Department of Computer Engineering – ALKÜ
Alanya

A Queue



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.

ADT Queue Operations

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

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

Some Queue Operations

Operation	***Queue after operation***	
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

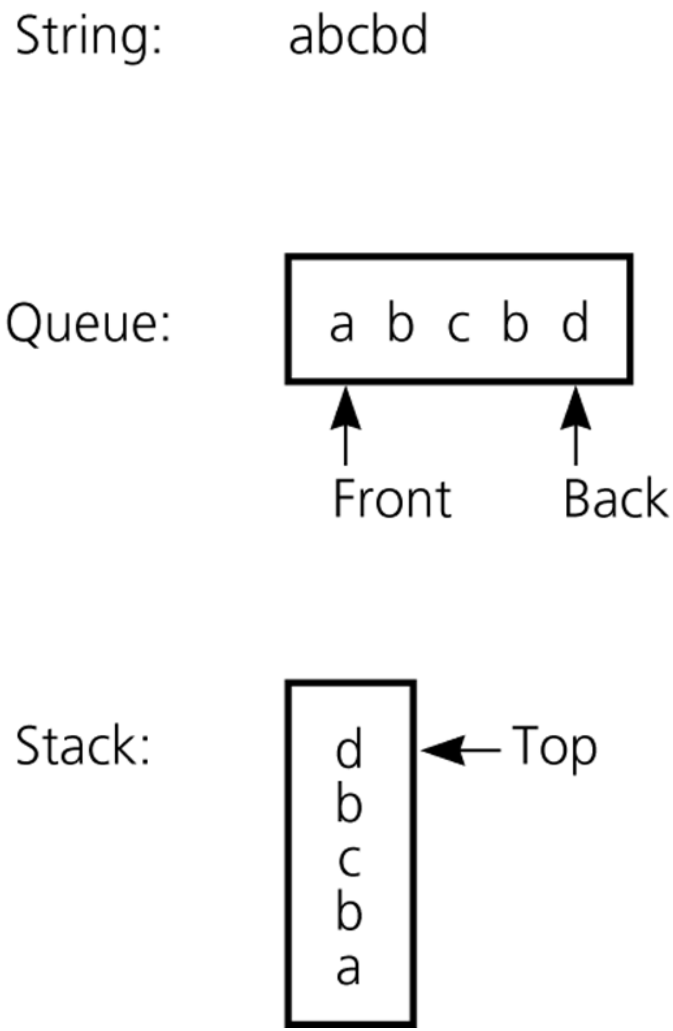
Recognizing Palindromes

- A palindrome
  - A string of characters that reads the same from left to right as its does from right to left
- Solution ideas?

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 -- Algorithm

```
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 charactersAreEqual = false; }
    return charactersAreEqual;
```

Recognizing Palindromes -- Algorithm

```
bool isPal(char str[], int left, int right) {
```

A recursive one???????????

```
}
```

Recognizing Palindromes -- Algorithm

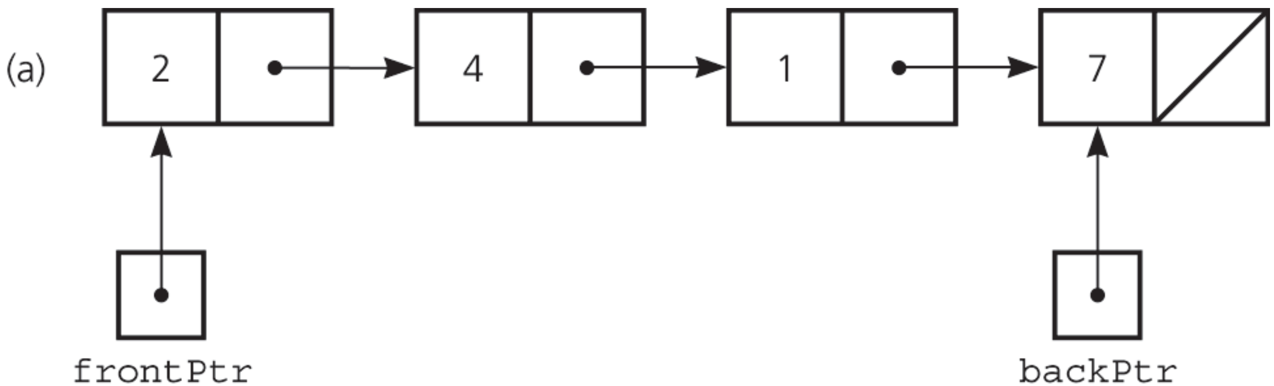
```
bool isPal(char str[], int left, int right) {

//to be called from main as isPal("rotator", 0, 6);
  if (left >= right)  //Could I have used == instead?
    return true;
  if (str[left] == str[right])
    return isPal(str, left+1, right-1);
  return false;

}

//idea: rotator is pal if otato is pal, if tat is pal, if a is pal
```

Implementations of the ADT Queue

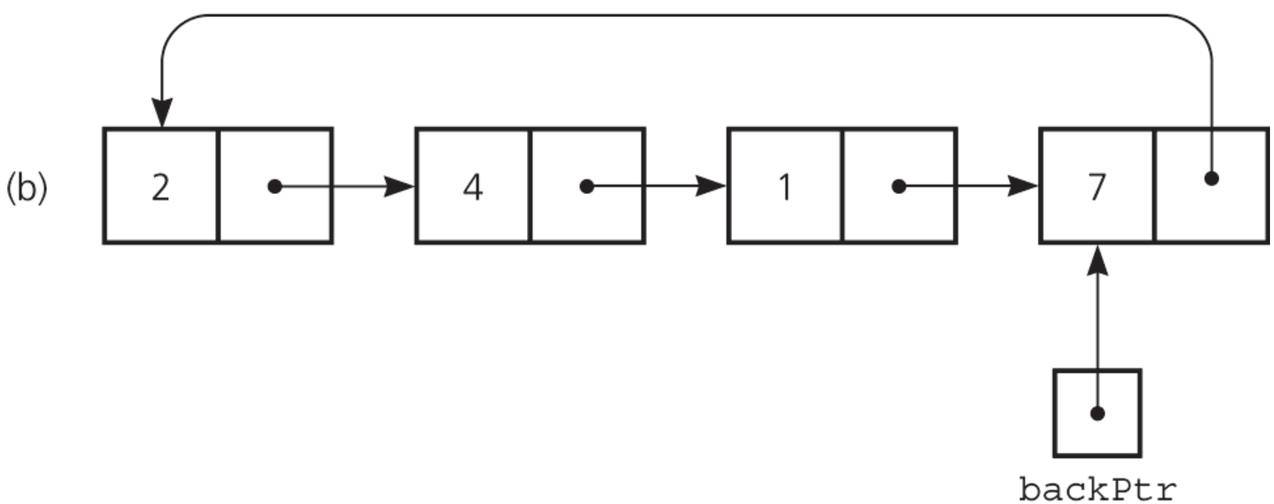
- Pointer-based implementations of queue
  - A linear linked list with two external references
    - A reference to the front
    - A reference to the back
  - A circular linked list with one external reference
    - A reference to the back

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

Pointer-based implementations of queue



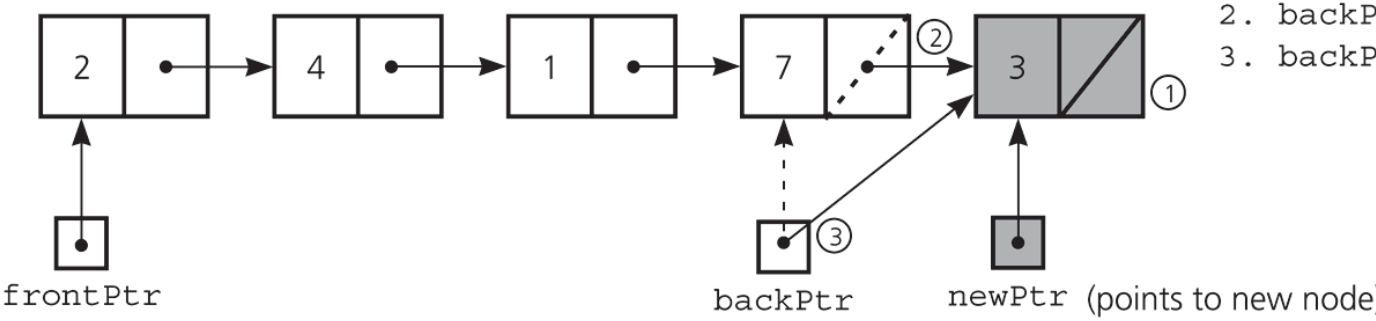
a linear linked list with two external pointers



a circular linear linked list with one external pointer

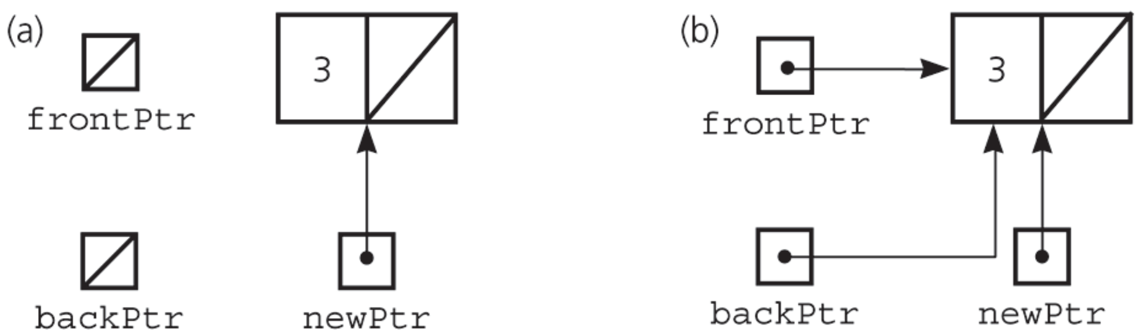
A linked list Implementation -- enqueue

Inserting an item into a nonempty queue




```
1. newPtr->next = NULL;
2. backPtr->next = newPtr;
3. backPtr = newPtr;
```

Inserting an item into an empty queue

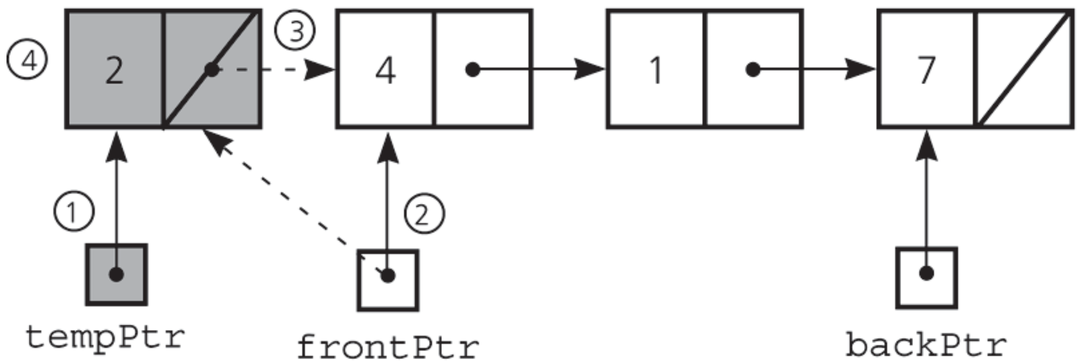


a) before insertion *b) after insertion*

```
frontPtr = newPtr;
backPtr = newPtr;
```

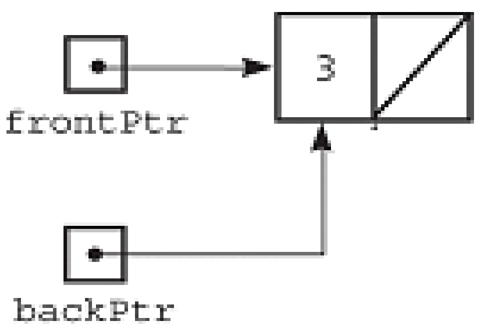
A Linked list Implementation -- dequeue

Deleting an item from a queue of more than one item

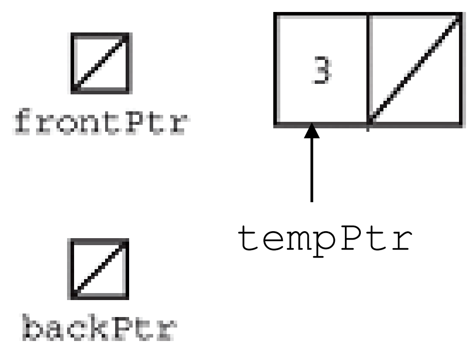



```
1. tempPtr = frontPtr;
2. frontPtr = frontPtr->next;
3. tempPtr->next = NULL;
4. delete tempPtr;
```

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 = NULL)
            : item(e), next(n) {}

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

A Linked list Implementation – Queue Class

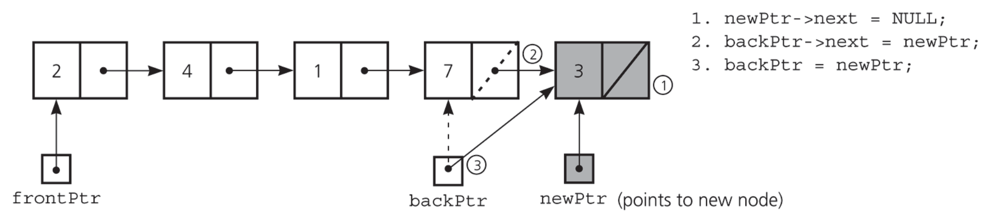
```
#include "QueueException.h"

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;      // Determines whether the queue is empty
   void enqueue(const T& 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(T& queueFront) throw(QueueException);
        // Retrieves the item at the front of a queue.
   void getFront(T& queueFront) const throw(QueueException);
private:
   QueueNode<T> *backPtr;
   QueueNode<T> *frontPtr;
}
```

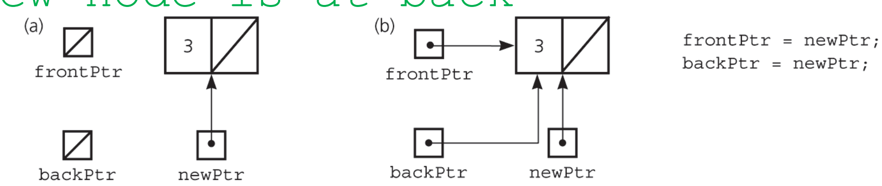
Linked List Implementation – constructor, deconstructor, isEmpty

```
template<class T>
Queue<T>::Queue() : backPtr(NULL), frontPtr(NULL){}  // default
  constructor

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 == NULL;
}
```

A Linked list Implementation – enqueue

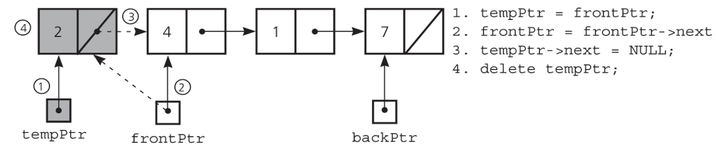
```
template<class T>
void Queue<T>::enqueue(const T& newItem) {
      // create a new node
      QueueNode<T> *newPtr = new QueueNode;

      // 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
}
```

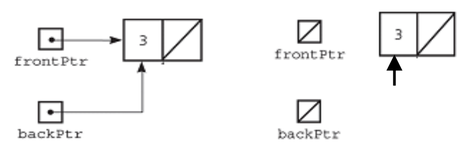




A Linked list Implementation – dequeue

```
template<class T>
void 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;
      if (frontPtr == backPtr) {          // one node in queue
         frontPtr = NULL;
         backPtr = NULL;
      }
      else
         frontPtr = frontPtr->next;

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

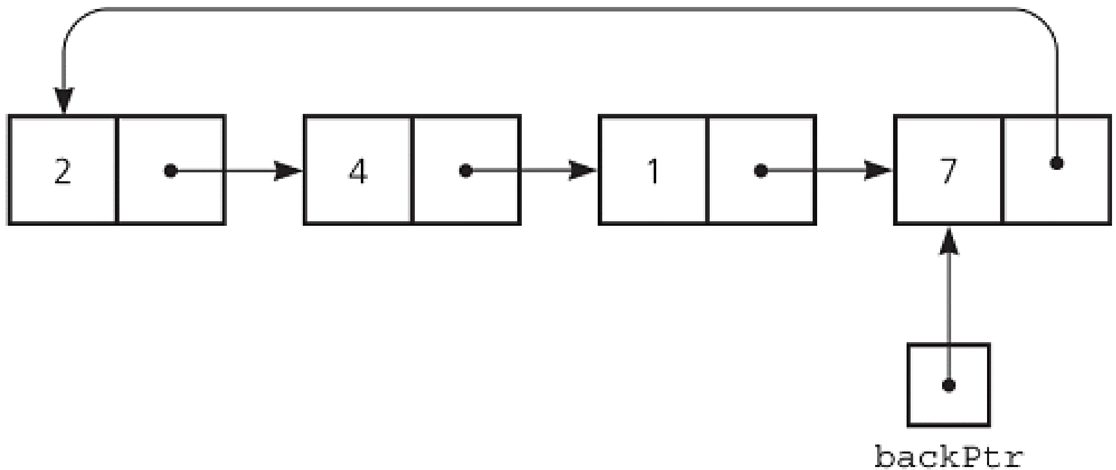




A Pointer-Based Implementation – dequeue, getFront

```
template<class T>
void Queue<T>::dequeue(T& 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
   }
}
template<class T>
void Queue<T>::getFront(T& 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 ?

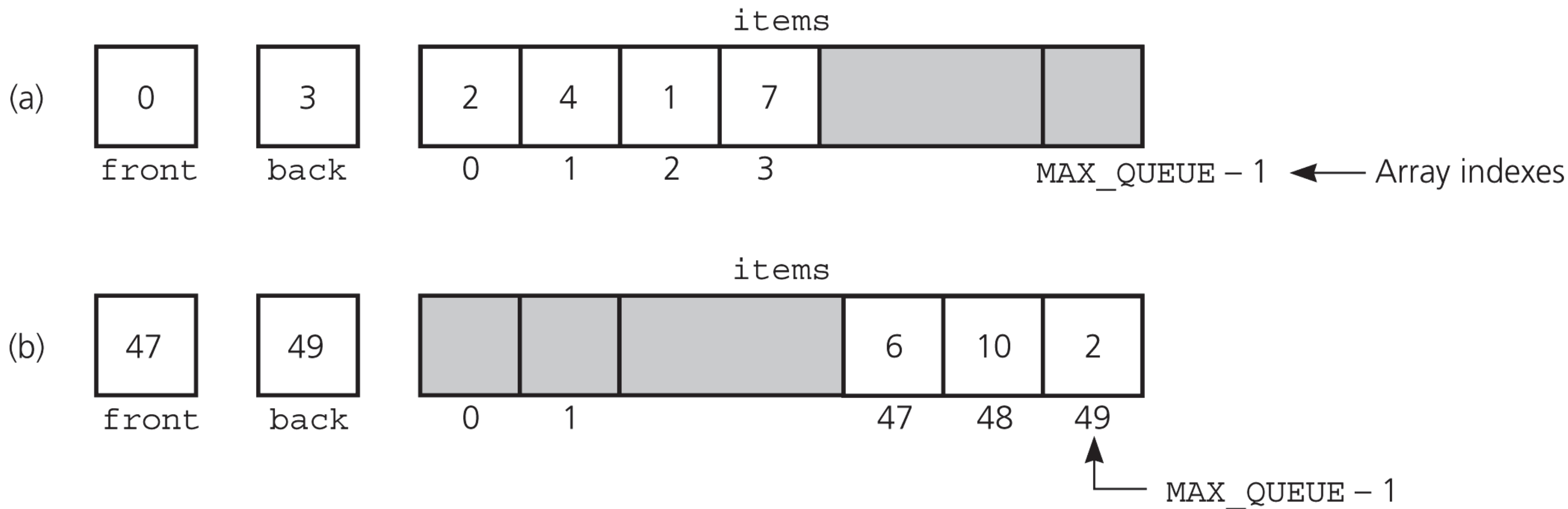
isEmpty ?

enqueue ?

dequeue ?

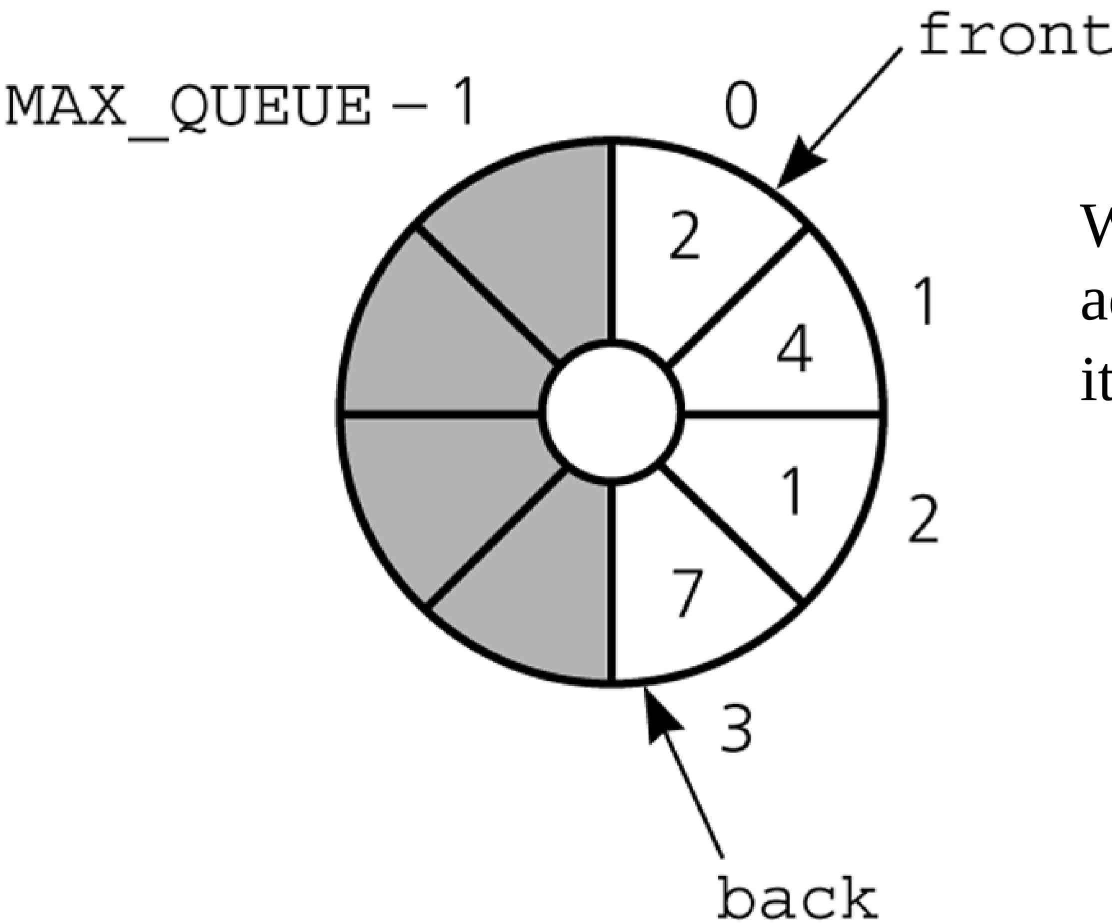
getFront ?

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.

A Circular Array-Based Implementation



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

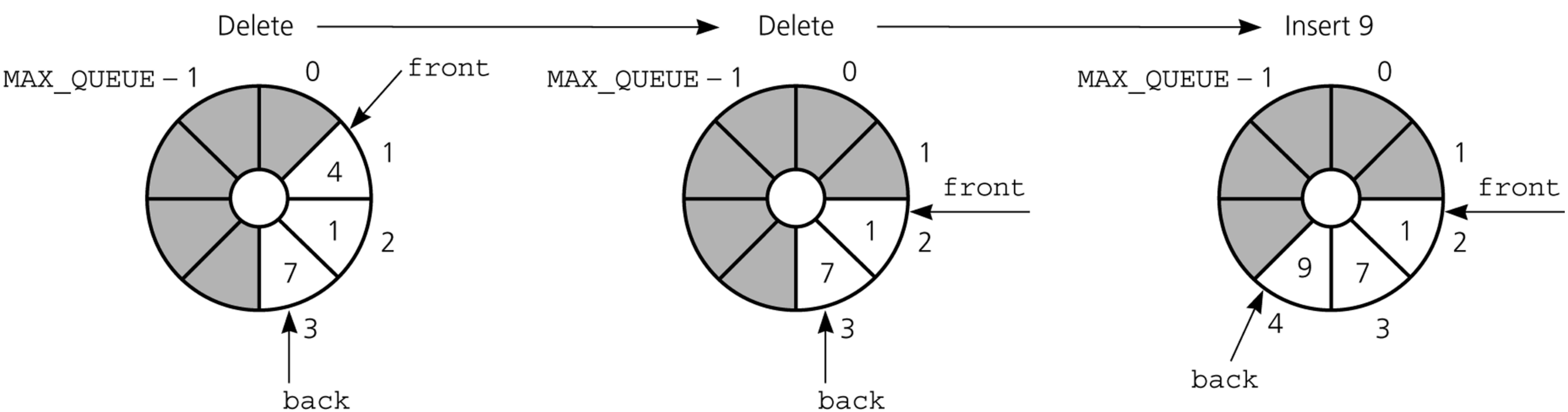
The effect of some operations of the queue

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

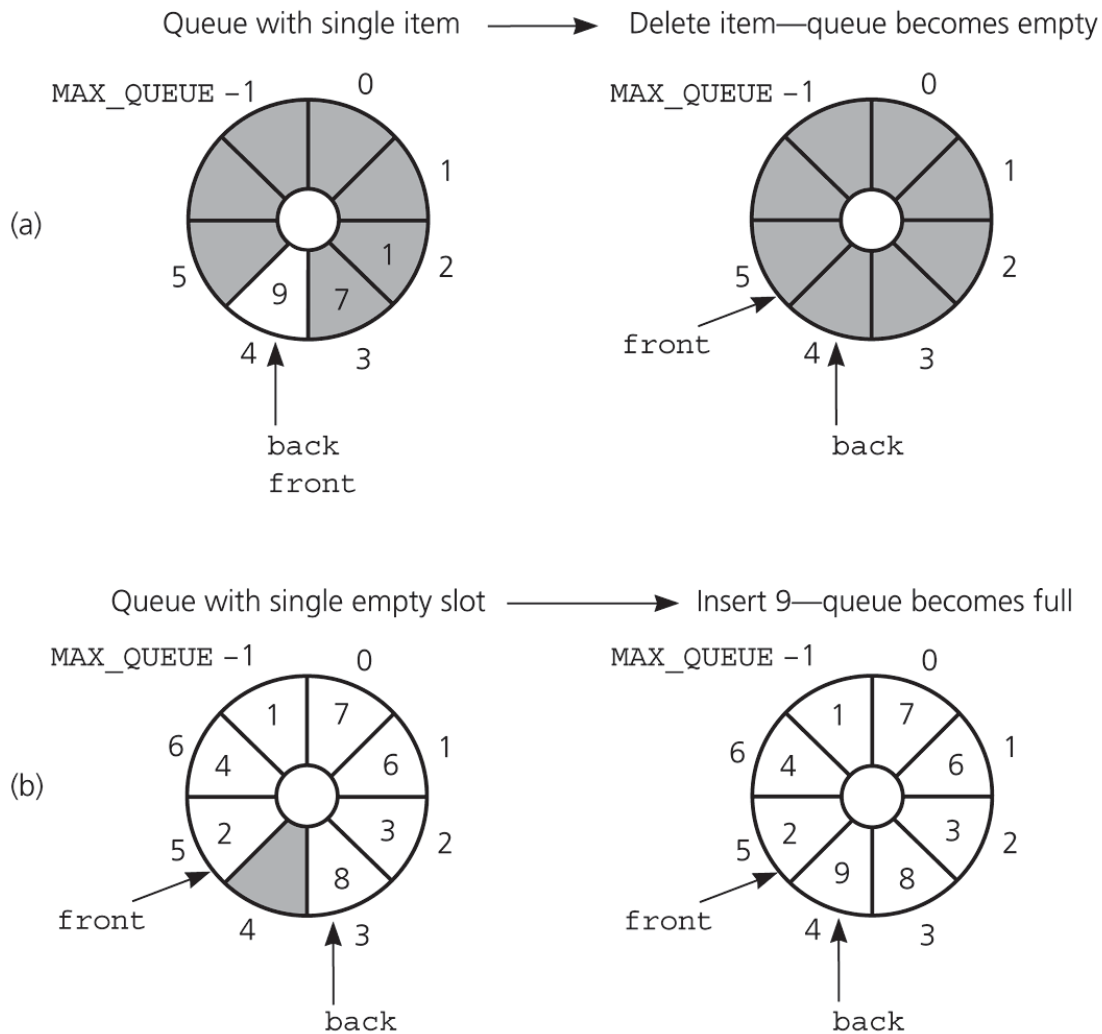
Insertion :
```
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
```

NOT ENOUGH

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



Problem: Q 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).
   - 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;

template <class T>;
class Queue {
public:
   Queue();  // default constructor
   bool isEmpty() const;
        void enqueue(const T& newItem) throw(QueueException);
        void dequeue() throw(QueueException);
        void dequeue(T& queueFront) throw(QueueException);
        void getFront(T& queueFront) 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<classT>
void Queue<T>::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(T& 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;
    }}
```

Array-Based Implementation Using a counter – getFront

```
template <class T>
void Queue<T>::getFront(T& 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

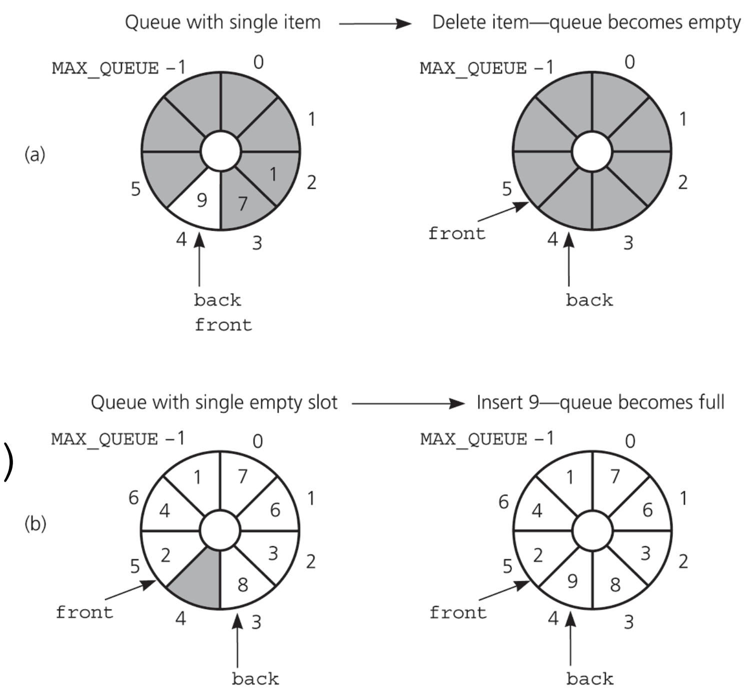
```
back = (back+1) % MAX_QUEUE; items[back] = newItem;
if ((back+1)%MAX_QUEUE == front)) isFull = true;
```

- Deleting from a queue

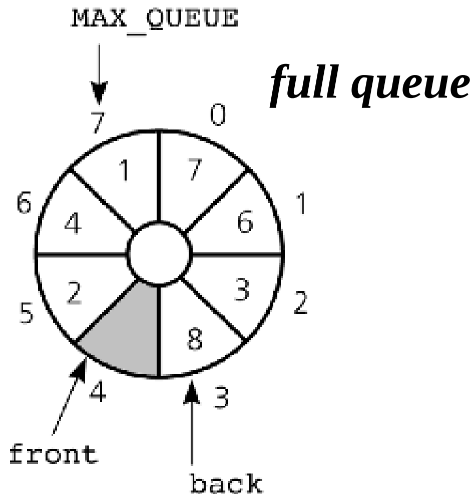
```
front = (front+1) % MAX_QUEUE;
isFull = false;
```

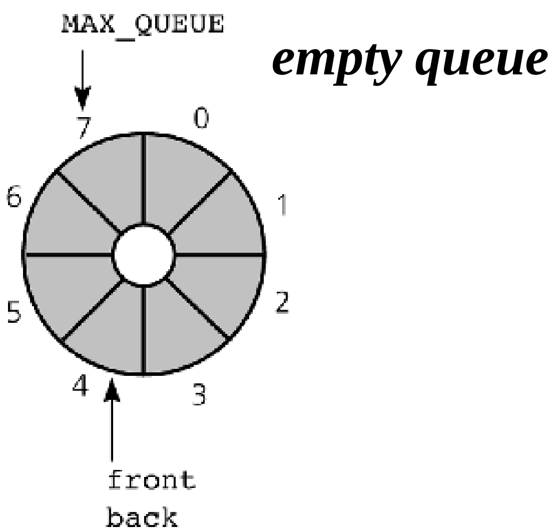
- Full: `isFull == true`
- Empty: `isFull==false`

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



Using an extra array location





- 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:

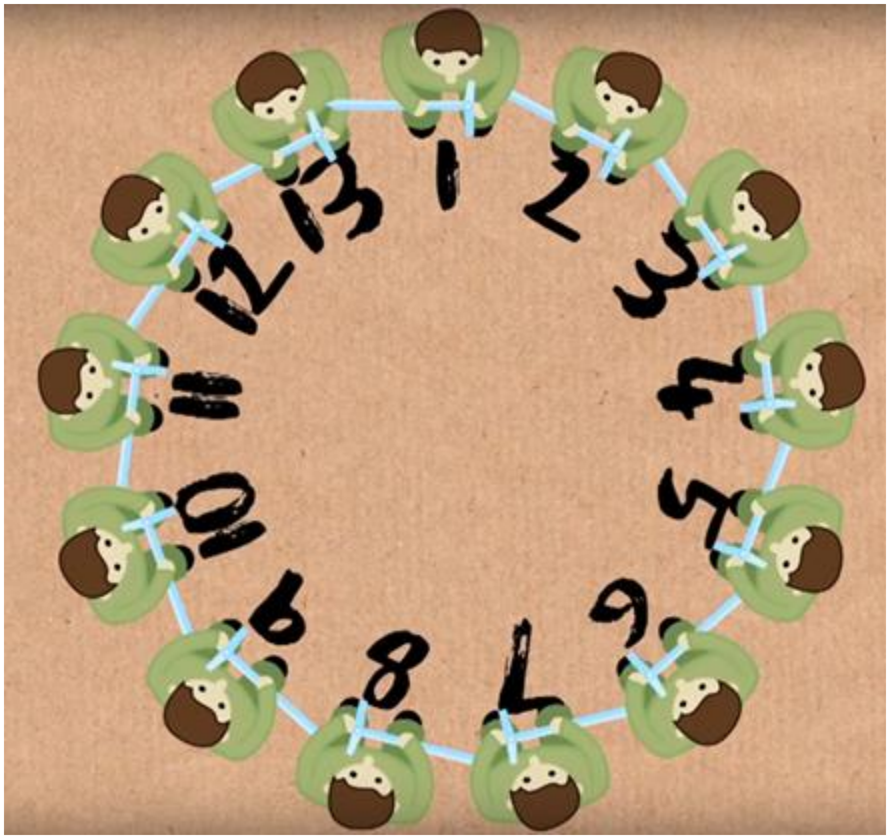
```
front == back
```

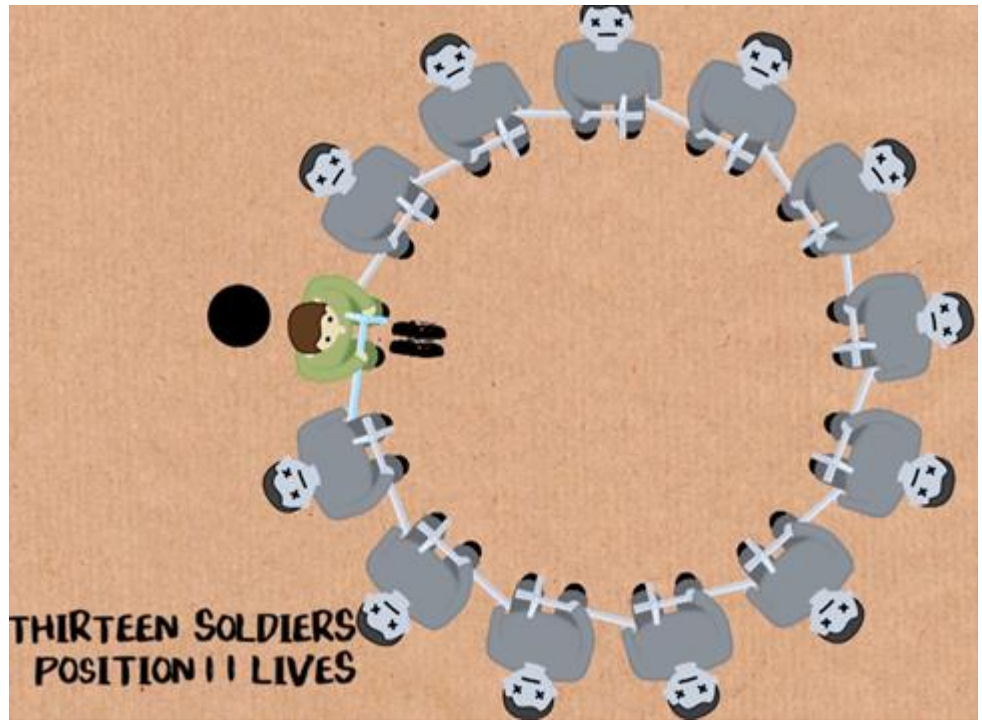
A Summary of ADTs

- ADTs: Stack, Queue.
- Stacks and Queues
  - Only the end 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`

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?





- History: N men surrounded by enemies. Preferred dying rather than captured as slaves. Every men killed the next living men until 1 man is left. That guy (Josephus) then surrendered (did not tell this initially ‘cos others’d turn on him).

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	1
17	3

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	1
17	3

Pattern # 1

Winner always odd!

Makes sense as the first loop kills all the evens

Application: Josephus Problem

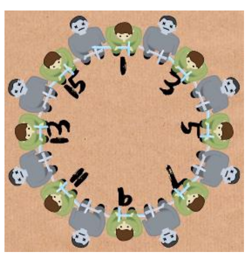
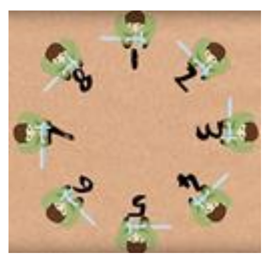
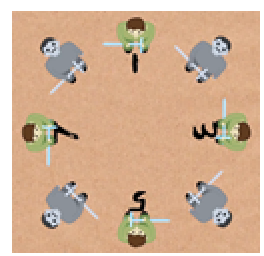
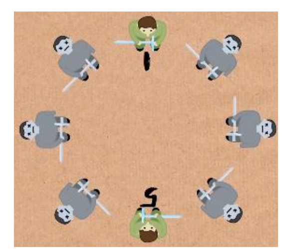
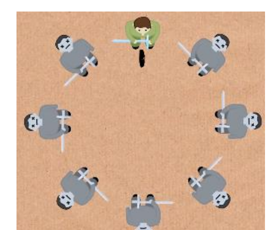
- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

Pattern # 2

Jump by 2; reset at 2^a for some a!

Makes sense: Assume 2^a men in circle. 1 pass removes half of them; at 1; repeat on 2^{a-1} men; so winner is the starting point (1)

Application: Josephus Problem

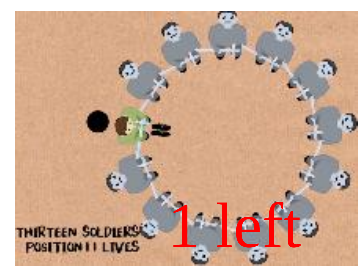
- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

Pattern # 2 (cont'd)

Jump by 2; reset at 2^a for some a!

Makes sense: Assume $2^a + b$ men in circle, where a is the biggest possible power; hence $b < 2^a$ (binary notation idea); after b men we are left with 2^a men, whose winner is the starting point (11 for below)



Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

Pattern # 2 (cont'd)

Jump by 2; reset at 2^a for some a!

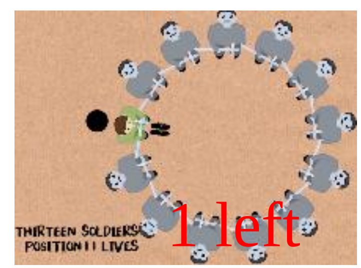
So what is 11 for N=13?

$N = 2^3 + 5$ (a=3, b=5); and $11 = 2*5 + 1$

In general, after b steps, we arrive at the position $2*b + 1$ (every 2nd is killed). Hence,

$$\mathbf{W(N) = 2*b + 1}$$

where $N = 2^a + b$ and $b < 2^a$



Application: Josephus Problem

- Based on $W(N) = 2*b + 1$ where $N = 2^a + b$ and $b < 2^a$ we can write a simple program that returns the winner/survivor index

```
int W(int N) {

}
```

Application: Josephus Problem

- Based on $W(N) = 2*b + 1$ where $N = 2^a + b$ and $b < 2^a$ we can write a simple program that returns the winner/survivor index

```
int W(int N) {
   int a = 0;
   while (N > 1) {
      N /= 2;
      a++;
   }
   return 2*(N – pow(2, a)) + 1;
   //or you could compute pow(2, a) in variable V like this:
   //int V = 1; for (int i = 0; i < a; i++) V *= 2;
}
```

Application: Josephus Problem

- Based on $W(N) = 2*b + 1$ where $N = 2^a + b$ and $b < 2^a$ we can write a simple program that returns the winner/survivor index
- If you do not like math and cannot extract W(N) above, you can write the code using a Queue
- Math gets way complicated for the generic problem where you kill every k^{th} man where $k > 1$

```
int Josephus(Q, k) { //Queue Q is built in advance with e.g., 1, 2, 3, 4, 5, 6.

}
```

Application: Josephus Problem

- Based on $W(N) = 2*b + 1$ where $N = 2^a + b$ and $b < 2^a$ we can write a simple program that returns the winner/survivor index
- If you do not like math and cannot extract W(N) above, you can write the code using a Queue
- Math gets way complicated for the generic problem where you kill every k^{th} man where $k > 1$

```
int Josephus(Q, k) { //Queue Q is built in advance with e.g., 1, 2, 3, 4, 5, 6.
   while (Q.size() > 1) {
      for (i = 1; i <= k-1; i++) //skip the k-1 men without killing
         Q.enqueue( Q.dequeue() );
      killed = Q.dequeue();
   }
   return Q.dequeue(); } //only one left in the Q, the winner ☺
```

References

- Yusuf Sahillioğlu, Data Structures Lecture Notes, METU