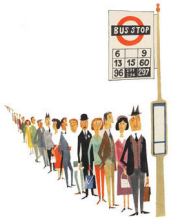
Stacks and Queues

Dr. Anirban Ghosh

School of Computing
University of North Florida

Stacks and Queues

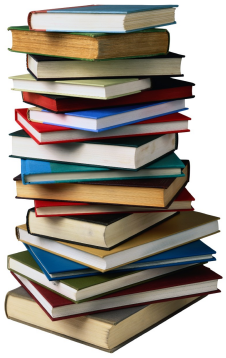


- **Stacks** are **Queues** are two popular easy-to-implement **abstract data types** (ADTs) used in algorithm design
- For an abstract data type, we **describe** the operations that can be executed on the data, but we do not define them
- For instance, the stack ADT can be implemented using a linked-list and an array as well

Stack

Definition

A stack is an ADT where items are inserted (**pushed**) and removed (**popped**) according to the **last-in, first-out** (LIFO) principle



Operations on a stack

Let us denote a stack by S.

1. `S.push(e)`: **adds** element e to the top of S
2. `S.pop()`: **removes** and **returns** the top element from S; returns `null` if S is empty
3. `S.top()`: **returns** the top element from S but does not remove it
4. `S.size()`: **returns** the number of elements stored in S currently
5. `S.isEmpty()`: **returns** `true` if S is empty otherwise returns `false`

push(55)

55

$$55, 3$$

push(69)

$$55, 3, 69$$

push(-88)

$$55, 3, 69, -88$$

push(-17)

$$55, 3, 69, -88, -17$$

$$55, 3, 69, -88, -17$$

`size()` returns 5

$$55, 3, 69, -88, -17$$

`pop()` removes and returns -17

$$55, 3, 69, -88$$

`pop()` removes and returns -88

$$55, 3, 69$$

`top()` returns 69

$$55, 3, 69$$

Applications

- Can be used to implement the **BACK** button in a browser. When you click on new URL, the new URL is visited and the current URL is pushed onto the stack. When you click on the BACK button, the top URL (the last visited URL) is popped and revisited
- Used by text editors to implement the **UNDO** operation. When you change something, the current state is pushed onto the stack before applying the new change. Hitting UNDO will simply revert the document to the previous state
- Method calls, recursion, and compiler design
- Used as an auxiliary data structure for many cornerstone algorithms

An `interface` for stacks

```
public interface StackADT<E> {
   int size();
   boolean isEmpty();
   void push(E e);
   E pop();
   E top();
}
```

To use a stack, one should create a class by implementing this interface

So how to implement a stack using the `StackADT` interface?

- Use an array (stack size must be known in advance)
  See the class `ArrayStack`
- Use a linked-list (can grow arbitrarily)
  See the class `LinkedStack`

Time complexity

Every one of these five stack operations, `size()`, `isEmpty()`, `push()`, `pop()`, `top()`, takes $O(1)$ time

An application: matching parentheses

- Given an algebraic expression, how to check if the parentheses, braces, and brackets in it are properly **balanced** or not
- Some **balanced** ones
  - `(a + b) + (66 * (s / t) - 12) * {p + (z - [2 + 99.1 / (m - x)])}`
    Sequence: `( ) ( ( ) ) { ( [ ( ) ] ) }`
  - `30 + (40 + (9 * (6 - 10) * (1 - (17 + 19 + 99) - 1) / {5 + (8 - [6 + 2 * (1 + (-99)) - 1] - 4) - 1} - 1) - 1) * 88`
    Sequence: `( ( ( ) ( ( ) ) { ( [ ( ) ] ) } ) )`
- Some **unbalanced** sequences
  - `) ( ( ) ) { ( [ ( ) ] ) }`
  - `( { [ ] ) }`
  - `(`

How to solve it?

- Recall the three pairs we are interested in:

  `( )`, `{ }`, `[ ]`

- Here is a stack-based algorithm for this problem:
  1. Declare a `Character` stack `S`
  2. Scan the input algebraic expression from left to right
  3. If a left symbol `(`, `{`, or `[` is encountered, **push** it onto `S`
  4. If a right symbol `)`, `}`, or `]` is encountered, check the top element in `S`; if these two elements form a matching pair, then pop from `S` otherwise report **INVALID**
  5. At the end, when the string is fully scanned, if `S` is empty, then report **VALID**, else **INVALID**

Sample run

Current symbol scanned	Stack (right is the stack top)	Action taken
( ) (( )) { ([( )]) }	(	push
( ) (( )) { ([( )]) }		pop
( ) (( )) { ([( )]) }	(	push
( ) (( )) { ([( )]) }	((	push
( ) (( )) { ([( )]) }	(	pop
( ) (( )) { ([( )]) }		pop
( ) (( )) { ([( )]) }	{	push
( ) (( )) { ([( )]) }	{ (	push
( ) (( )) { ([( )]) }	{ ([	push
( ) (( )) { ([( )]) }	{ ([(	push
( ) (( )) { ([( )]) }	{ ([	pop
( ) (( )) { ([( )]) }	{ (	pop
( ) (( )) { ([( )]) }	{	pop
( ) (( )) { ([( )]) }		pop

Result: **VALID**

Another sample run

Current symbol scanned	Stack (right is the stack top)	Action taken
({ [ ]) }	(	push
({ [ ]) }	({	push
({ [ ]) }	({ [	push
({ [ ]) }	({	pop
({ [ ]) }	({	**mismatch**

Result: **INVALID**

Code

```java
import java.util.Scanner;
public class ExpresssionChecker {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      String expression = input.nextLine();
      LinkedStack<Character> S = new LinkedStack<>();
      int pos;

      for(pos = 0; pos < expression.length(); pos++) {
         char current = expression.charAt(pos);

         if( current == '(' || current == '{' || current == '[')       S.push(current);
         else if( current == ')' && !S.isEmpty() && S.top() == '(' )   S.pop();
         else if( current == ')' && !S.isEmpty() && S.top() != '(' )   break;
         else if( current == '}' && !S.isEmpty() && S.top() == '{' )   S.pop();
         else if( current == '}' && !S.isEmpty() && S.top() != '{' )   break;
         else if( current == ']' && !S.isEmpty() && S.top() == '[' )   S.pop();
         else if( current == ']' && !S.isEmpty() && S.top() != '[' )   break;
      }

      if( S.isEmpty() && pos == expression.length() )   System.out.println("VALID");
      else System.out.println("INVALID");

      input.close();
   }
}
```

Food for thought

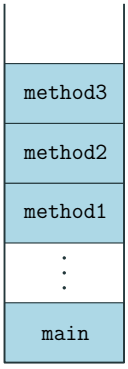
```
<html>
   <head>
      <title>Data Structure</title>
   </head>

   <body>
      <h1>
         <p>Stacks are fun...</p>
      </h1>

      <p>Queues are also fun...</p>
   </body>
</html>
```

Write a program that can check if an HTML file is correctly formatted. This means every opening tag must have its closing tag at an appropriate location in the file.

Method calls and use of stack

- Ever wondered how the control returns to the **caller** method after the **callee** method is done with its execution?

```
public char method1() {
   // ...
   method2();
   counter++; // control returns here
   i = j + 10;
   //...
}
public double method2() {
   // ...
    s = p + s;
    method3();
    arr[q] = arr[q] / (10 * m); // control returns here
   // ...
}
public void method3() {
   // ...
    s = p + s;
   // ...
   return;
}
```



- A stack of **activation record**s is used by the system; push a record when a method starts execution; pop its record when it is done

Queue

Definition

A queue is an ADT where items are inserted (**enequed**) and removed (**dequeued**) according to the **first-in, first-out** (FIFO) principle

Operations on a queue

Let us denote a queue by `Q`.

1. `Q.enqueue(e)`: **adds** an element `e` to the back of `Q`
2. `Q.dequeue()`: **removes** and **returns** the first element from `Q`; returns `null` if `Q` is empty
3. `Q.first()`: **returns** the first element of `Q`, without removing it; returns `null` is `Q` is empty
4. `Q.size()`: **returns** the number of elements stored in `Q` currently
5. `Q.isEmpty()`: **returns** `true` if `Q` is empty, otherwise returns `false`

enqueue(55)

55

$$55, 3$$

$$55, 3, 69$$

enqueue(-88)

$$55, 3, 69, -88$$

enqueue(-17)

$$55, 3, 69, -88, -17$$

$$55, 3, 69, -88, -17$$

`size()` returns 5

$$55, 3, 69, -88, -17$$

dequeue() removes and returns 55

$$3, 69, -88, -17$$

dequeue() removes and returns 3

$$69, -88, -17$$

`first()` returns 69

$$69, -88, -17$$

Applications

- Simulation of real-world queues (airlines, ticket counter, etc.)
- Graph algorithms
- Resource sharing in multi-user systems
- Operating systems
- Computer networks

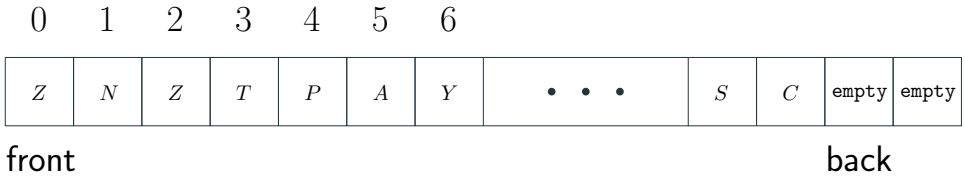
...

An `interface` for queues

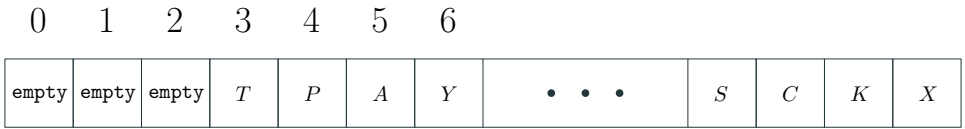
```
public interface QueueADT<E> {
   void enqueue(E e);
   E dequeue();
   int size();
   boolean isEmpty();
   E first();
}
```

To use a queue, one should create a class by implementing this interface

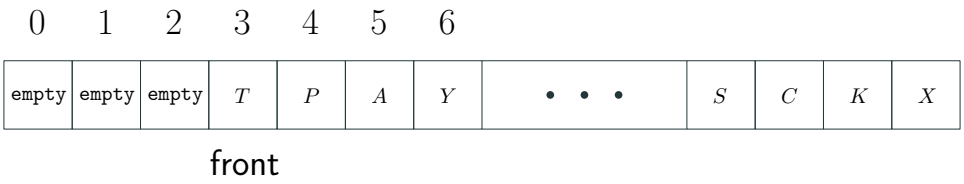
Implementing using array



After 3 dequeues and 2 enqueues the situation is: we have space but we cannot enqueue anymore!

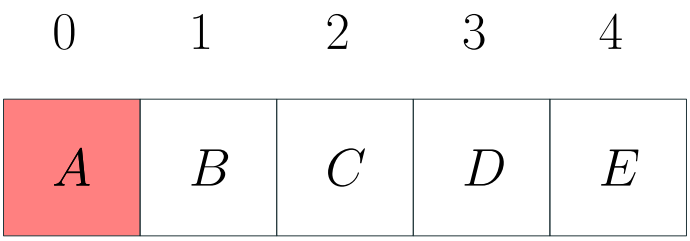


What to do then?



- Trivial solution: shift the content to the left and add new stuff
- Downside? This is very expensive! Takes $O(n)$ time
- We need to implement `enqueue` faster than this
- Solution: wrap around

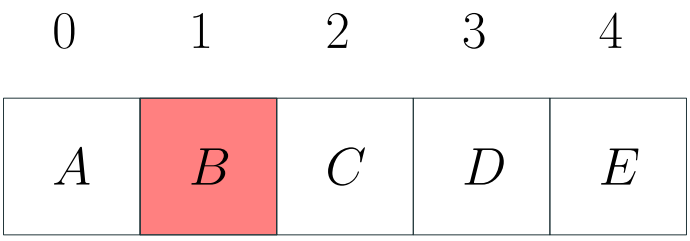
Using the mod operator for wrapping around



$$i = 0$$

Next index: $(i + 1) \bmod 5 = 1 \bmod 5 = 1$

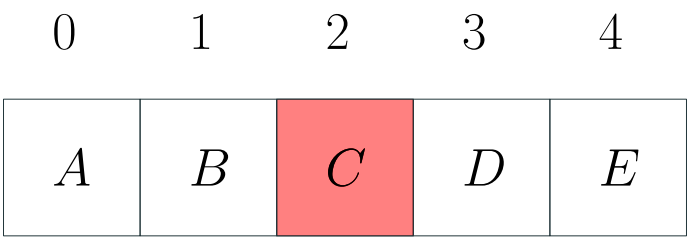
Using the mod operator for wrapping around



$$i = 1$$

Next index: $(i + 1) \bmod 5 = 2 \bmod 5 = 2$

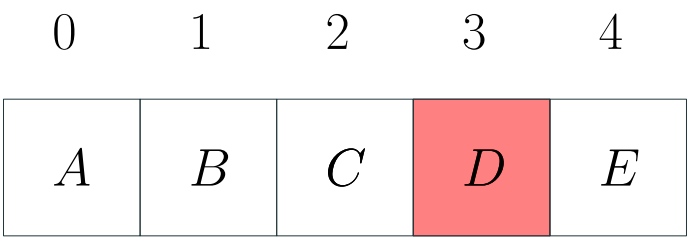
Using the mod operator for wrapping around



$$i = 2$$

Next index: $(i + 1) \bmod 5 = 3 \bmod 5 = 3$

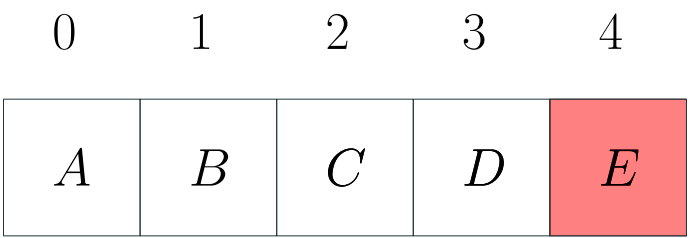
Using the mod operator for wrapping around



$$i = 3$$

Next index: $(i + 1) \bmod 5 = 4 \bmod 5 = 4$

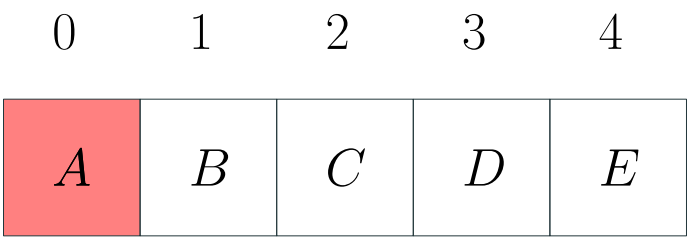
Using the mod operator for wrapping around



$$i = 4$$

Next index: $(i + 1) \bmod 5 = 5 \bmod 5 = 0$

Using the mod operator for wrapping around



$$i = 0$$

Next index: $(i + 1) \bmod 5 = 1 \bmod 5 = 1$

Using this wrap-around approach for array-based queues

0	1	2	3	4	5	6					
Z	N	Z	T	P	A	Y	• • •	S	C	empty	empty

front back

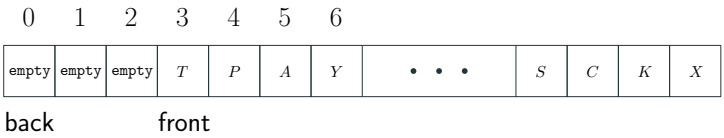
0	1	2	3	4	5	6					
empty	empty	empty	T	P	A	Y	• • •	S	C	K	X

front

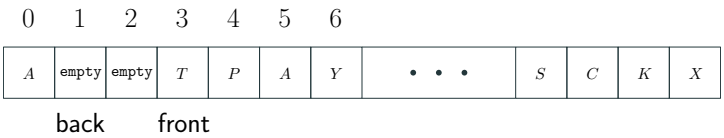
0	1	2	3	4	5	6					
empty	empty	empty	T	P	A	Y	• • •	S	C	K	X

back front

Using this wrap-around approach for array-based queues



After `Q.enqueue('A');`, we get



So how to implement a stack using the `QueueADT` interface?

- Use an array (queue size must be known in advance)
  See the class `ArrayQueue`
- Use a linked-list (can grow arbitrarily)
  See the class `LinkedQueue`

Time complexity

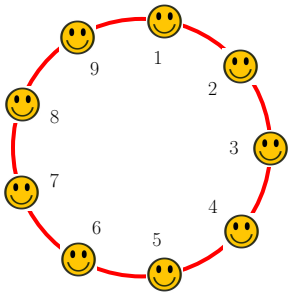
Every one of these five queue operations, `size()`, `isEmpty()`, `enqueue()`, `dequeue()`, `first()`, takes $O(1)$ time

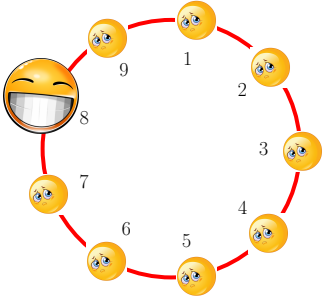
An application: the Josephus problem

The problem

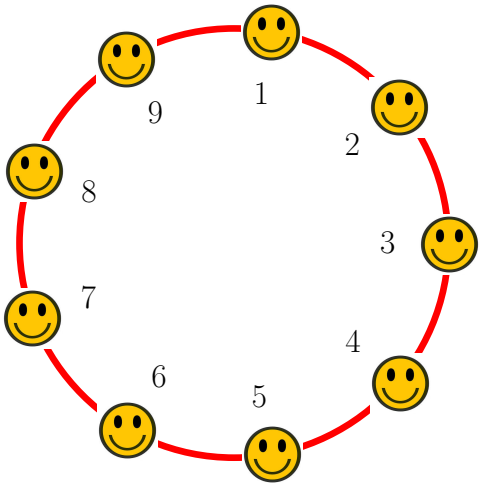
A group of n people agree to play the following fun game. They arrange themselves on a circle (at positions numbered from 1 to n) and proceed around the circle clockwise, eliminating every mth person until only one person is left.

The last remaining person wins the game. How to figure out the winning position?

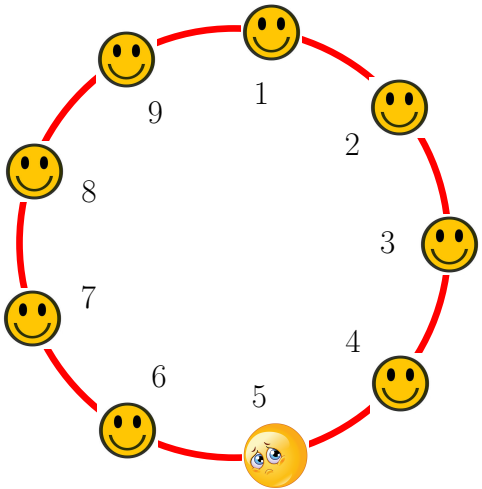




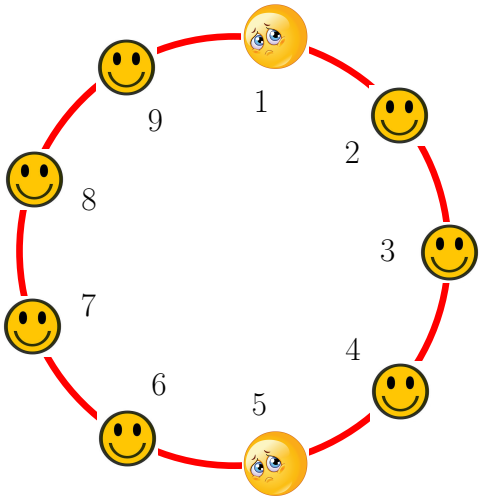
Example: $n = 9, m = 5$



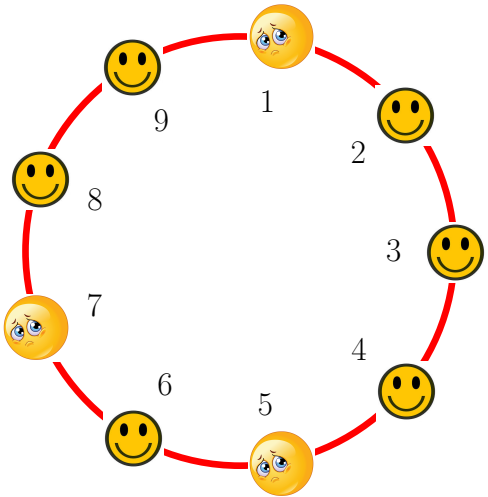
Example: $n = 9, m = 5$



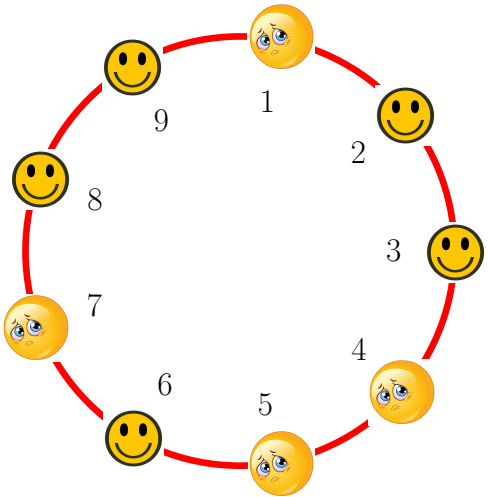
Example: $n = 9, m = 5$



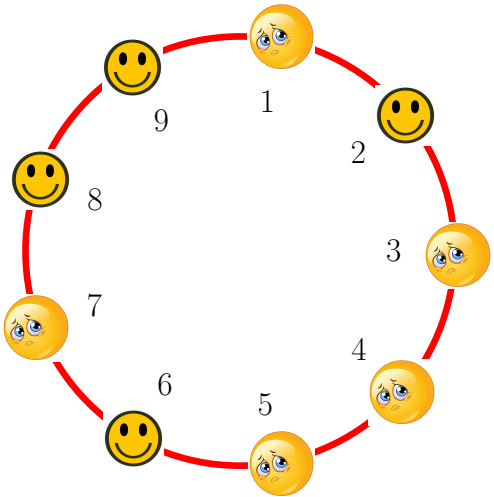
Example: $n = 9, m = 5$



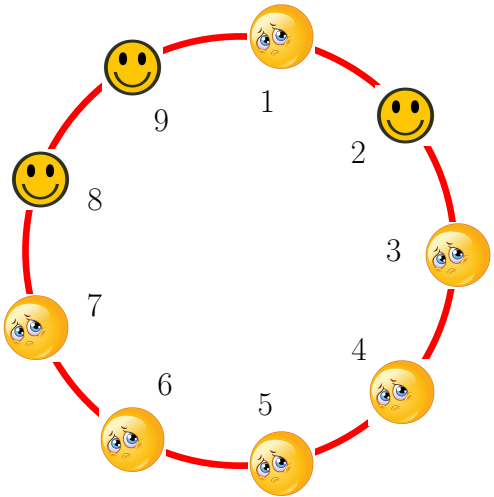
Example: $n = 9, m = 5$



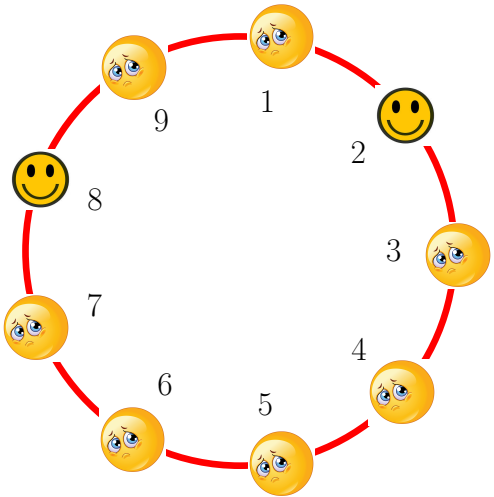
Example: $n = 9, m = 5$



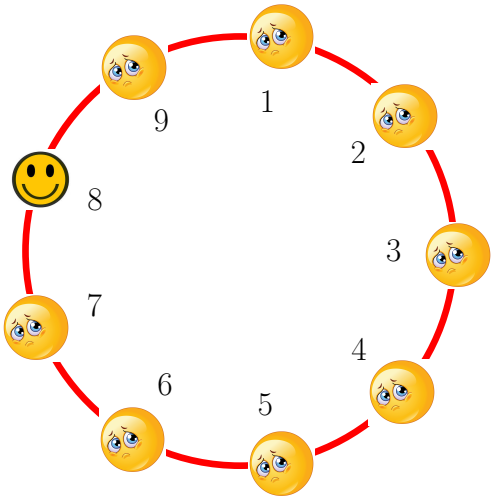
Example: $n = 9, m = 5$



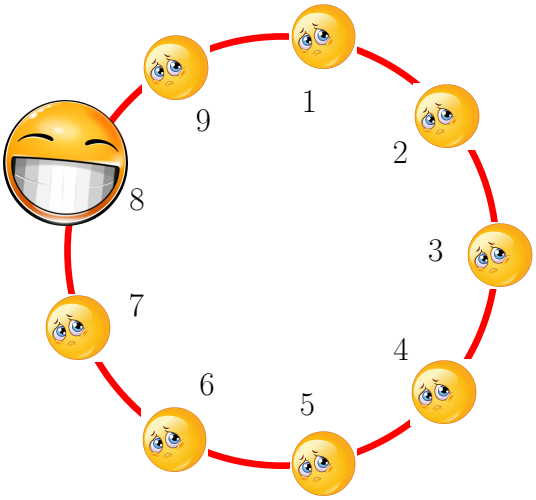
Example: $n = 9, m = 5$



Example: $n = 9, m = 5$



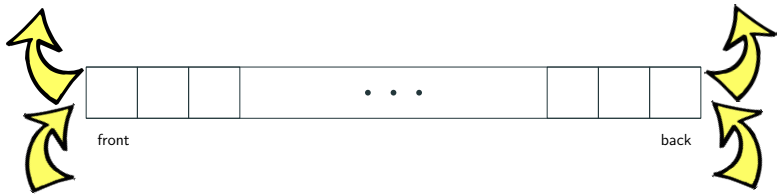
Example: $n = 9, m = 5$



Code

```
import java.util.Scanner;

public class JosephusSolver {
   public static void main(String[] args) {
      int n = 9, m = 5;

      LinkedQueue<Integer> Q = new LinkedQueue<>();

      for (int i = 1; i <= n; i++) // populate the queue with the integers 1,2,...,n
         Q.enqueue(i);

      while ( (Q.size() > 1) ) {
         for (int i = 0; i < m-1; i++) // send the first m-1 elements to the back of the queue
            Q.enqueue(Q.dequeue());

         System.out.println("Eliminating player: " + Q.dequeue() + " ");
      }

      System.out.print("\nWinning position: " + Q.first()); // the sole integer in the queue is the winner
   }
}
```

Double ended queues: Deque (pronounced as DECK)



Supports insertion and removal at both the front and the back

Deque can be used as a stack or a queue

An easy way to implement a deque

Doubly linked-list

Deque ADT

```java
public interface DequeADT<E> {
   int size();
   boolean isEmpty();
   E first();
   E last();
   void addFirst(E e);
   void addLast(E e);
   E removeFirst();
   E removeLast();
}
```

☞ Refer to the class `LinkedDeque` for an implementation

Time complexity

Just like stacks and queues, all operations on deques run in $O(1)$ time

Suggested exercise

Create an array-based iterable deque class

Reading

Stacks and queues from **Chapter 5**

https://opendsa-server.cs.vt.edu/OpenDSA/Books/CS3/html/index.html