Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Useful in "parsing" program (code) files to check for syntactic correctness

W/o stack [Typical answers]

1) Keep 3 counters of open brackets, one for each type

2) Keep a flag storing type of last open bracket

Does not end with last. You also need "second" last, "third" last & so on...

3) On encountering "closed" bracket match its "type" with flag in 2

4) If match in (3) decrement corresponding counter in 1.....

Parentheses Matching Algorithm

ignore variables *, + etc

Algorithm ParenMatch(X,n):
Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i=0 to n-1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.isEmpty() then
            return false {nothing to match with}
        if S.pop() does not match the type of X[i] then
            return false {wrong type}
if S.isEmpty() then
    return true {every symbol matched}
else
    return false {some symbols were never matched}
```

Q: Diffence between HTML tag matching & parenthesis matching?

Ans: "Units" to be matched (alphabet) are <..> & </...>

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Tag Matching Algorithm

- Is similar to parentheses matching:

```
import java.util.StringTokenizer;
import datastructures.Stack;
import datastructures.NodeStack;
import java.io.*;
/** Simpli.ed test of matching tags in an HTML document. */
public class HTML { /** Nested class to store simple HTML tags */
        public static class Tag { String name; // The name of this tag
                boolean opening; // Is true i. this is an opening tag
                public Tag() { // Default constructor
                        name = "";
                        opening = false;
                }
                public Tag(String nm, boolean op) { // Preferred constructor
                name = nm;
                opening = op;
                }
                /** Is this an opening tag? */
                public boolean isOpening() { return opening; }
                /** Return the name of this tag */
                public String getName() {return name; }
        }
        /** Test if every opening tag has a matching closing tag. */
        public boolean isHTMLMatched(Tag[ ] tag) {
                Stack S = new NodeStack(); // Stack for matching tags
                for (int i=0; (i <tag.length) && (tag[i] != null); i++) {
                        if (tag[i].isOpening())
                        S.push(tag[i].getName()); // opening tag; push its name on the stack
                        else {
                                if (S.isEmpty()) // nothing to match
                                        return false;
                                if (!((String) S.pop()).equals(tag[i].getName())) // wrong match
                                        return false;
                        }
                }
                if (S.isEmpty())
                    return true; // we matched everything
                return false; // we have some tags that never were matched
        }
```

object to be pushed into stack (instead of character symbols "(", ")" "{", ...)

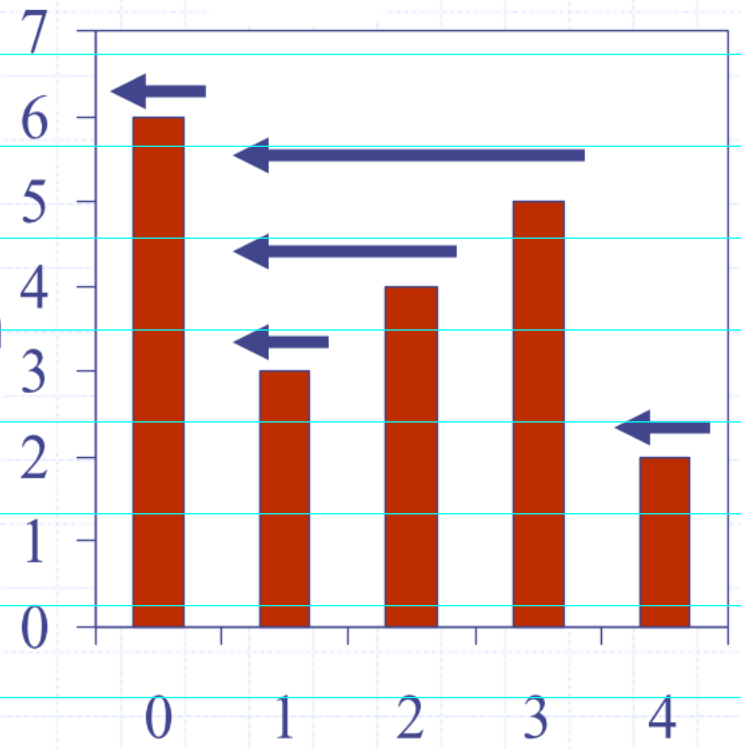
Tag Matching Algorithm, cont.

```
public final static int CAPACITY = 1000;              // Tag array size upper bound
  /* Parse an HTML document into an array of html tags */
  public Tag[ ] parseHTML(BufferedReader r) throws IOException {
      String line; // a line of text
      boolean inTag = false ;                         // true iff we are in a tag
       Tag[ ] tag = new Tag[CAPACITY]; // our tag array (initially all null)
      int count = 0 ;                                 // tag counter
      while ((line = r.readLine()) != null) {
            // Create a string tokenizer for HTML tags (use < and > as delimiters)
            StringTokenizer st = new StringTokenizer(line,"<> \t",true);
            while (st.hasMoreTokens()) {
                  String token = (String) st.nextToken();
                  if (token.equals("<")) // opening a new HTML tag
                        inTag = true;
                  else if (token.equals(">"))  // ending an HTML tag
                        inTag = false;
                  else if (inTag) { // we have a opening or closing HTML tag
                        if ( (token.length() == 0) || (token.charAt(0) != '/') )
                        tag[count++] = new Tag(token, true); // opening tag
                        else  // ending tag
                        tag[count++] = new Tag(token.substring(1), false); // skip the
                  } // Note: we ignore anything not in an HTML tag
            }
      }
      return tag; // our array of tags
  }
   /** Tester method */
  public static void main(String[ ] args) throws IOException {
      BufferedReader stdr;          // Standard Input Reader
      stdr = new BufferedReader(new InputStreamReader(System.in));
      HTML tagChecker = new HTML();
      if (tagChecker.isHTMLMatched(tagChecker.parseHTML(stdr)))
            System.out.println("The input file is a matched HTML document.");
      else
            System.out.println("The input file is not a matched HTML document.");
      }
}
```

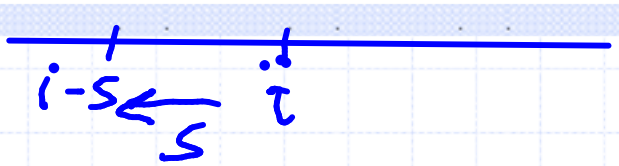
Given an an array X, the span S[i] of X[i] is the maximum number of consecutive elements X[j] immediately preceding X[i] (and including j= i) such that X[j] X[i]. Given an array X[1...n], present algorithms for computing the array of spans S[1...n] with and without making use of the stack data structure.

Computing Spans

- We show how to use a stack as an auxiliary data structure in an algorithm
- Given an an array X, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \leq X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1



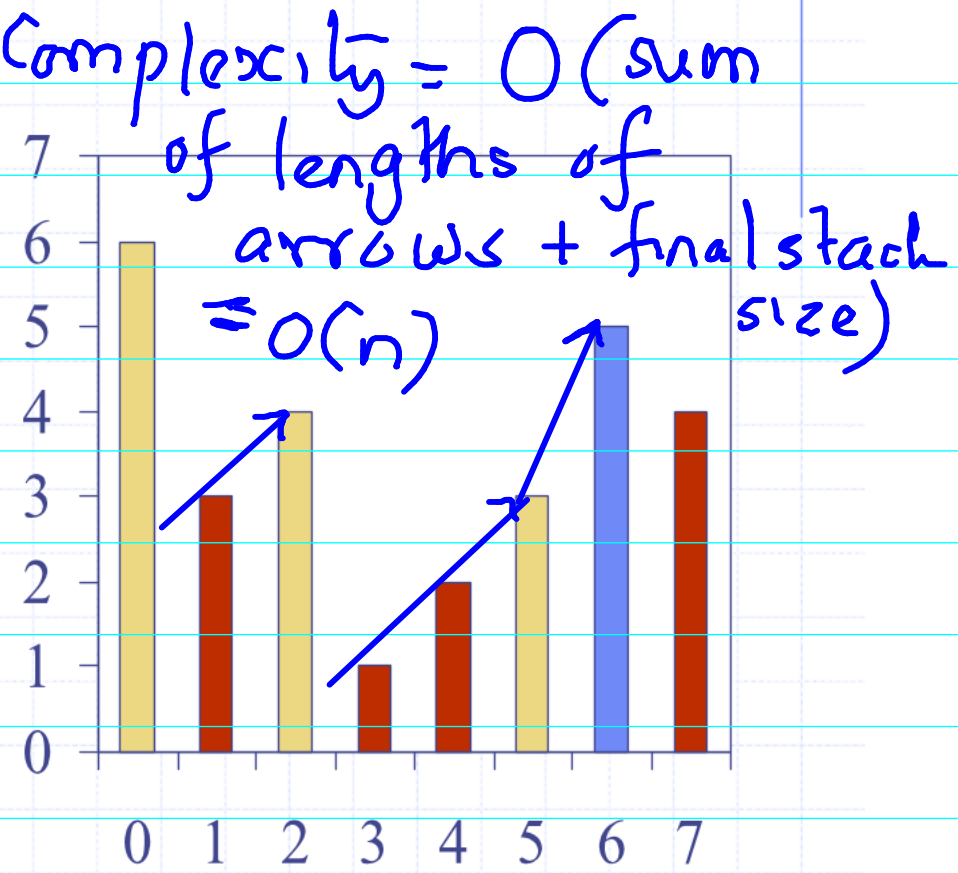
Quadratic Algorithm

```
Algorithm spans1(X, n)                                  #
    Input array X of n integers
    Output array S of spans of X                        #
    S ← new array of n integers                         n
    for i ← 0 to n − 1 do                               n
        s ← 1                                           n
        while s ≤ i ∧ X[i − s] ≤ X[i]                   1 + 2 + …+ (n − 1)
            s ← s + 1                                   1 + 2 + …+ (n − 1)
        S[i] ← s                                        n
    return S                                            1
```

- Algorithm *spans1* runs in $O(n^2)$ time

Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let i be the current index
  - We pop indices from the stack until we find index j such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push x onto the stack



Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

```
Algorithm spans2(X, n)                                  #
    S ← new array of n integers                         n
    A ← new empty stack                                 1
    for i ← 0 to n − 1 do                               n
        while (¬A.isEmpty() ∧
                X[A.top()] ≤ X[i] ) do                  n
            A.pop()                                     n
        if A.isEmpty() then                             n
            S[i] ← i + 1                                n
        else
            S[i] ← i − A.top()                          n
        A.push(i)                                       n
    return S                                            1
```

STACK AXIOMS [format of "axioms" for any data structure]

Let S be any stack and I be any item. Then:

EMPTY(CREATE) ::= True

EMPTY(PUSH(I,S)) ::= False

TOP(CREATE) ::= Error (exceptions in java)

TOP(PUSH(I,S)) ::= I, [I, S]

POP(CREATE) ::= error

POP(PUSH(I,S)) ::= S, I

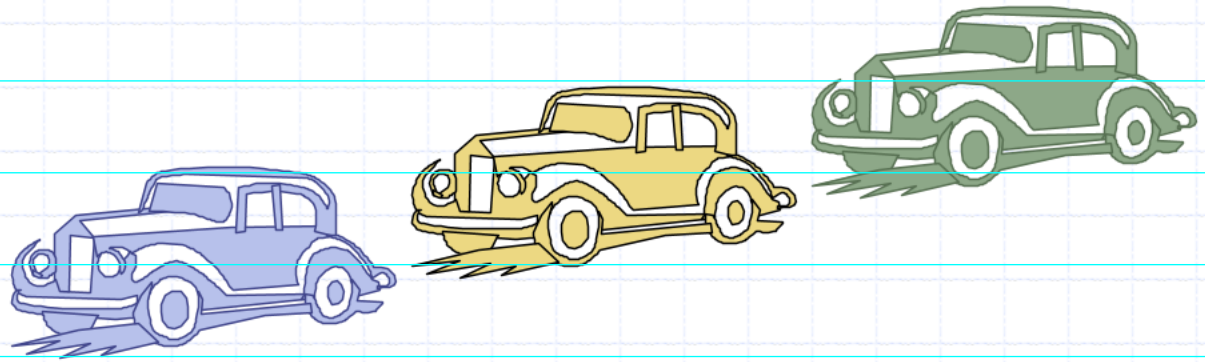
SIZE(CREATE) ::= 0

SIZE(PUSH(I,S)) ::= SIZE(S) + 1;

SIZE(POP(S)) ::= SIZE(S) - 1;

RHS is state after execution of functions on LHS

Queues



The Queue ADT (§4.3)

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue

FIFO
LILO

- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

QUEUE AXIOMS

Let Q be any queue and I be any item. Then:

EMPTY(CREATE) ::= True

EMPTY(ENQUEUE(I,S)) ::= False

FRONT(CREATE) ::= Error

FRONT(ENQUEUE(I,[X,S])) ::= X

DEQUEUE(CREATE) ::= error

DEQUEUE(ENQUEUE(I,[X,S])) ::= X, [S,I]

SIZE(CREATE) ::= 0

SIZE(ENQUEUE(I,S)) ::= SIZE(S) + 1;

SIZE(DEQUEUE(S)) ::= SIZE(S) - 1;

Queue Example

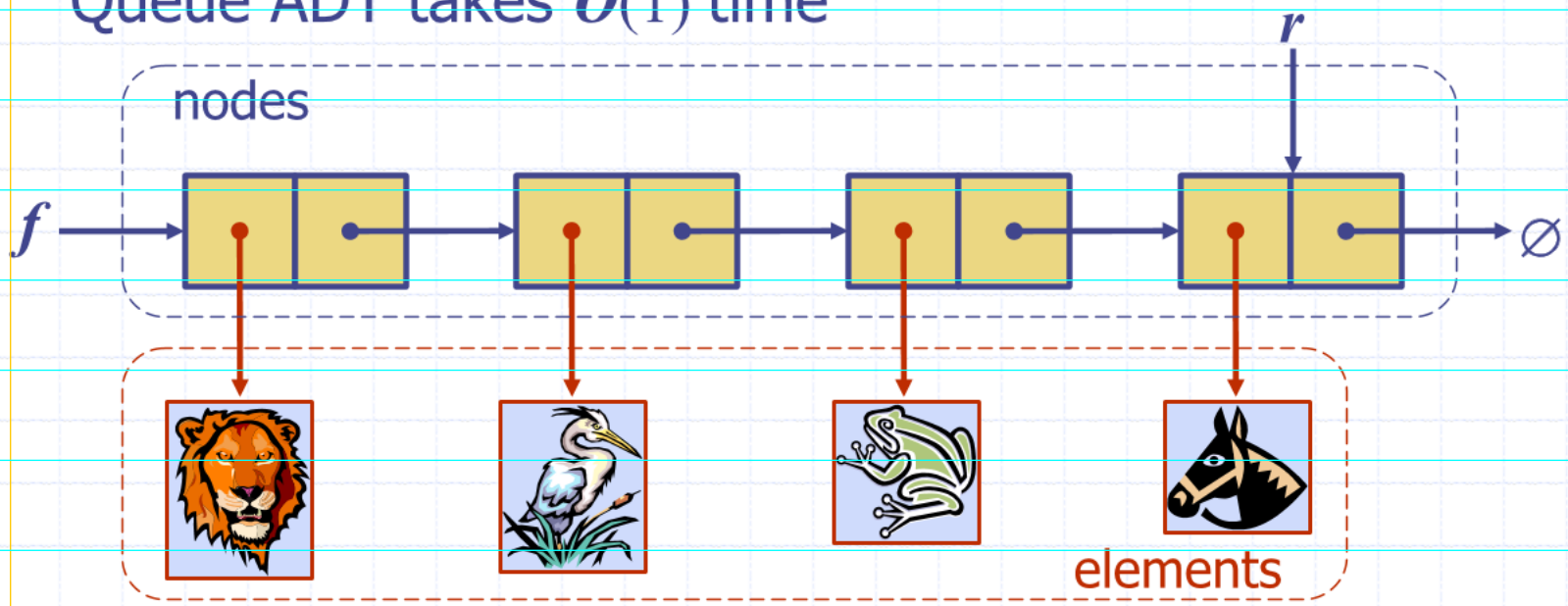
Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Queue with a Singly Linked List

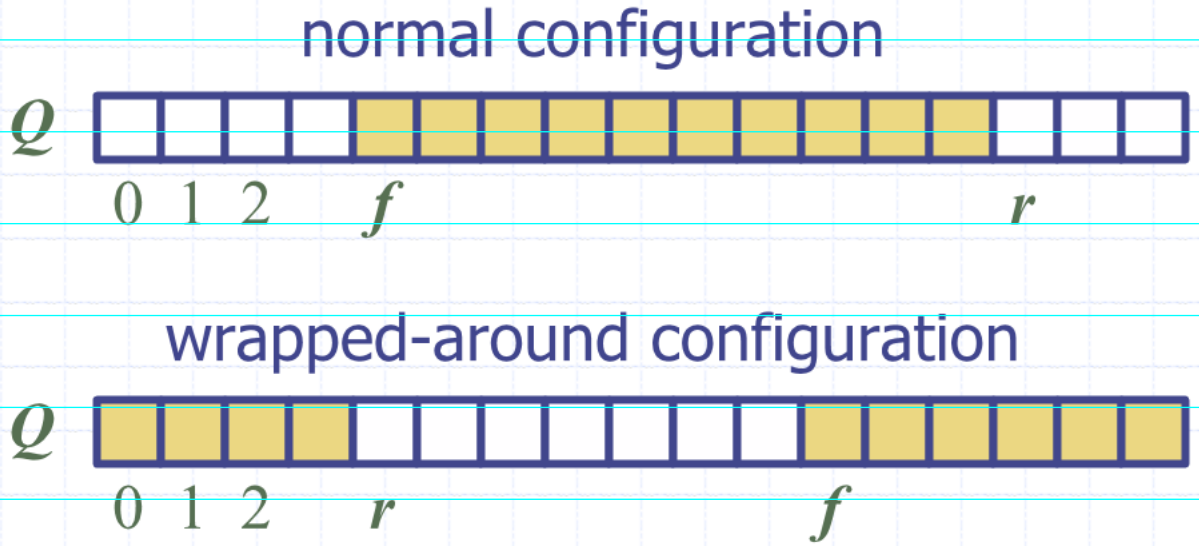
- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Q: Queue using an array?

Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

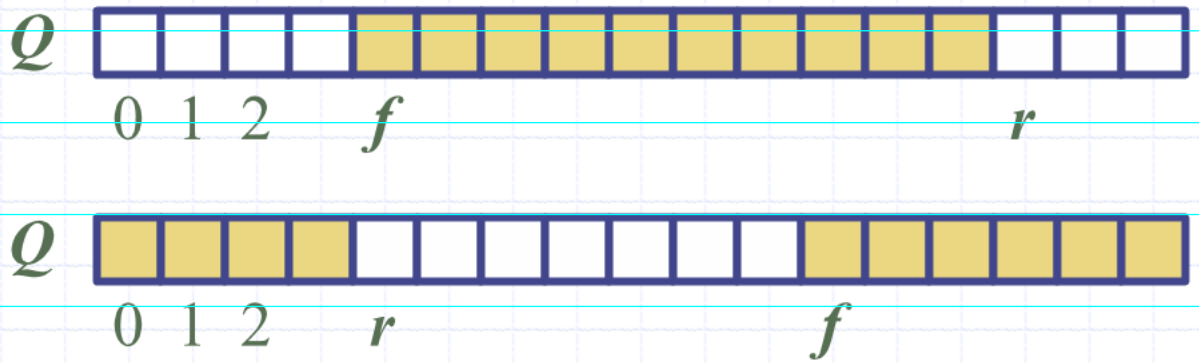


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

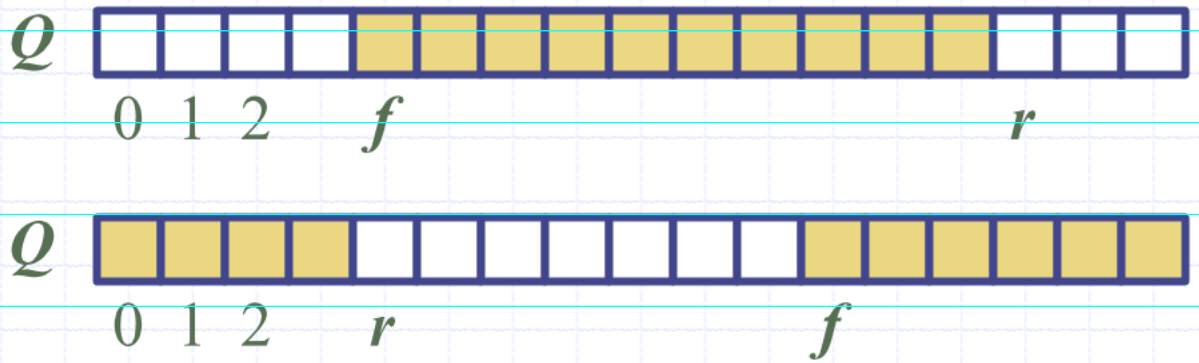
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

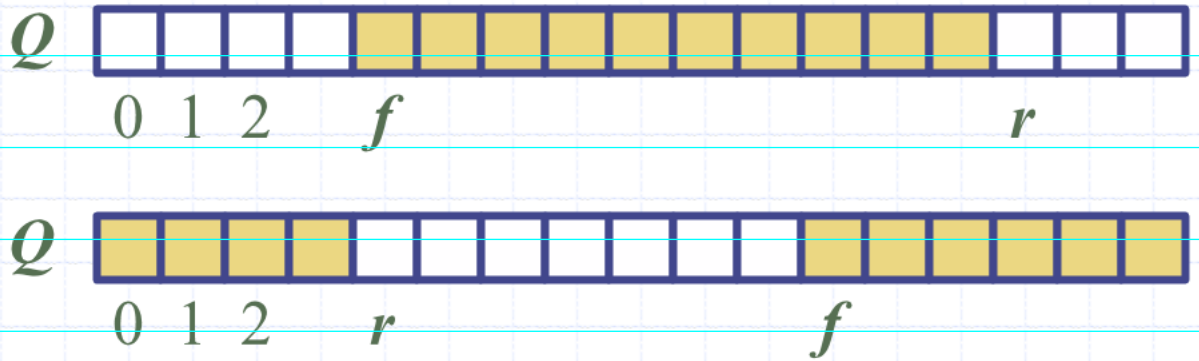
Algorithm *enqueue*(*o*)
  if *size*() = *N* – 1 **then**
    throw *FullQueueException*
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
  if isEmpty() then
    throw EmptyQueueException
  else
    o ← Q[f]
    f ← (f + 1) mod N
    return o
```



Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in Java class

```
public interface Queue {

   public int size();

   public boolean isEmpty();

   public Object front()
         throws EmptyQueueException;

   public void enqueue(Object o);

   public Object dequeue()
         throws EmptyQueueException;
}
```

1] Suppose you were asked to implement a stack using one or more queues. What is the best implementation possible (in terms of efficiency of the push and pop operations)?

2] Now suppose you were asked to implement a queue using one or more stacks. Again, what is the best implementation possible (in terms of efficiency of the enqueue and dequeue operations)?