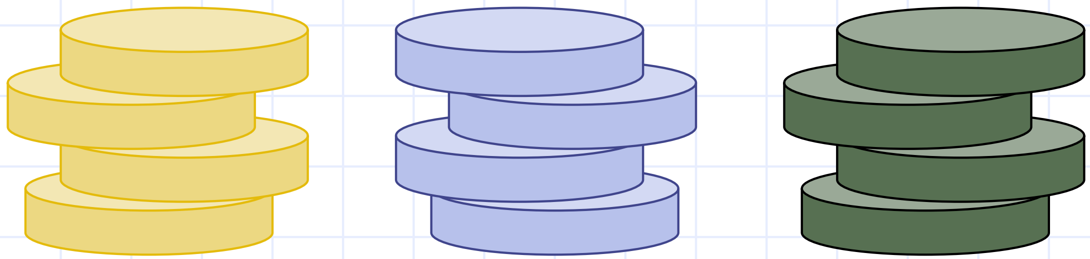
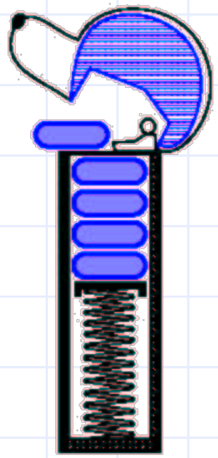
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Stacks



The Stack ADT



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Assumes null is returned from top() and pop() when stack is empty
- Different from the built-in Java class java.util.Stack

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

Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	–	(7)
push(9)	–	(7, 9)
top()	9	(7, 9)
push(4)	–	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

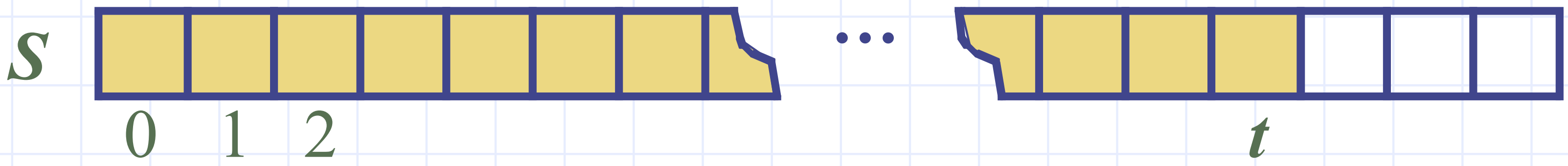
Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

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

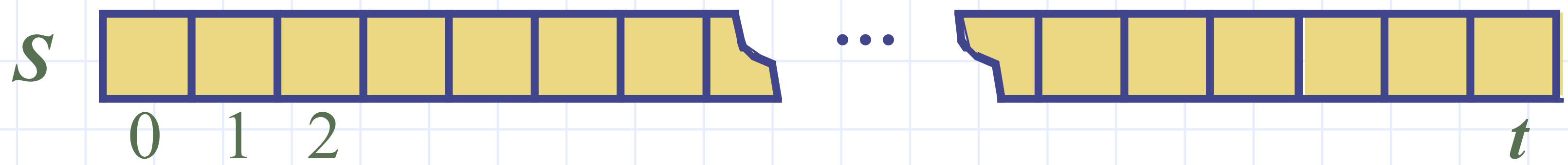
```
Algorithm size()
   return t + 1

Algorithm pop()
   if isEmpty() then
      return null
   else
      t ← 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 *IllegalStateException*
  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

Array-based Stack in Java

```
public class ArrayStack<E>
      implements Stack<E> {

   // holds the stack elements
   private E[ ] S;

   // index to top element
   private int top = -1;

   // constructor
   public ArrayStack(int capacity) {
      S = (E[ ]) new Object[capacity]);
   }
```

```
   public E pop() {
    if isEmpty()
          return null;
     E temp = S[top];
     // facilitate garbage collection:
     S[top] = null;
     top = top – 1;
     return temp;
   }
…  (other methods of Stack interface)
```

Example Use in Java

```
public class Tester {

  // … other methods
  public intReverse(Integer a[]) {
      Stack<Integer> s;
      s = new ArrayStack<Integer>();

      … (code to reverse array a) …
   }
```

```
public floatReverse(Float f[]) {
    Stack<Float> s;
    s = new ArrayStack<Float>();

    … (code to reverse array f) …
 }
```

Parentheses Matching

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

Parenthesis Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) { // this is a right delimiter
      if (buffer.isEmpty( )) // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
        return false; // mismatched delimiter
      }
    }
  return buffer.isEmpty( ); // were all opening delimiters matched?
}
```

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?

HTML Tag Matching (Java)

```
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer =  new LinkedStack<>( );
  int j = html.indexOf('<'); // find first '<' character (if any)
  while (j != −1) {
    int k = html.indexOf('>', j+1); // find next '>' character
    if (k == −1)
      return false; // invalid tag
    String tag = html.substring(j+1, k); // strip away < >
    if (!tag.startsWith("/")) // this is an opening tag
      buffer.push(tag);
    else { // this is a closing tag
      if (buffer.isEmpty( ))
        return false; // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false; // mismatched tag
    }
    j = html.indexOf('<', k+1); // find next '<' character (if any)
  }
  return buffer.isEmpty( ); // were all opening tags matched?
}
```

Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group evaluated from left to right

Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and equal precedence operations.

Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
    prec(opStk.top())
    doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

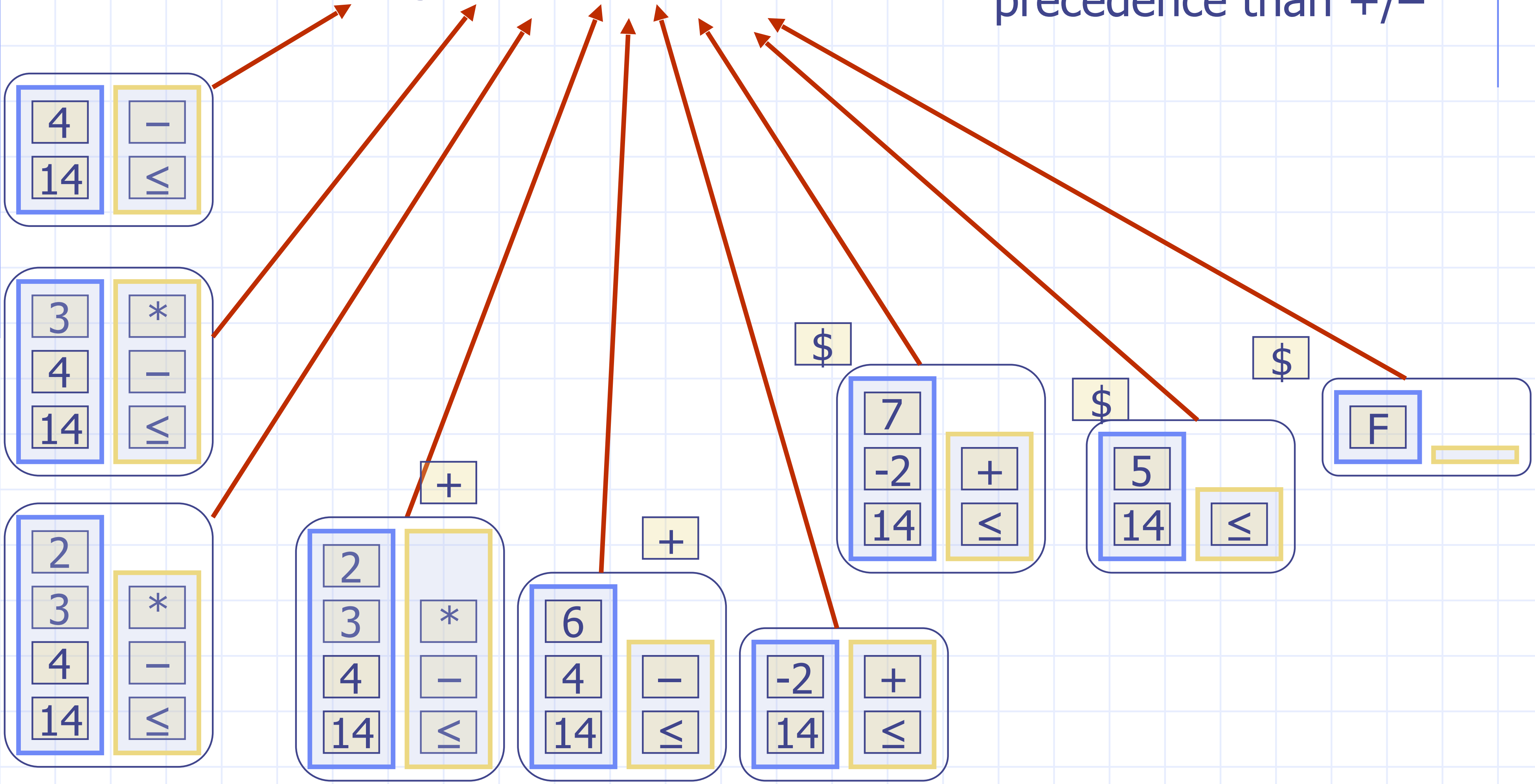
```
while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression

Slide by Matt Stallmann included with permission.

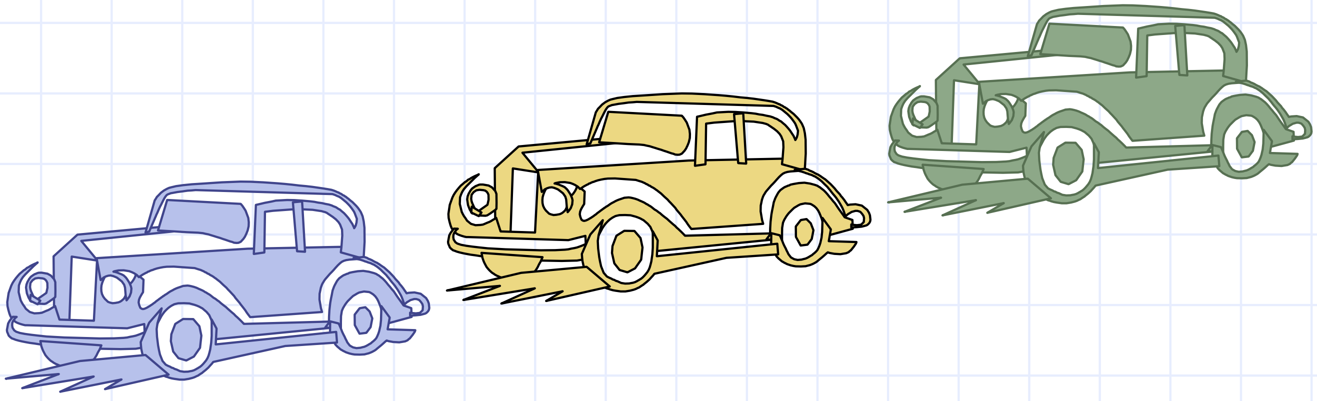
$14 \le 4 - 3 * 2 + 7$

Operator ≤ has lower precedence than +/–



Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Queues



The Queue ADT

- 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
- Auxiliary queue operations:
  - object first(): 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
- Boundary cases:
  - Attempting the execution of dequeue or first on an empty queue returns null

Example

Operation		*Output*	*Q*
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	*5*	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	*3*	(7)	
first()	*7*	(7)	
dequeue()	*7*	()	
dequeue()	*null*	()	
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)	

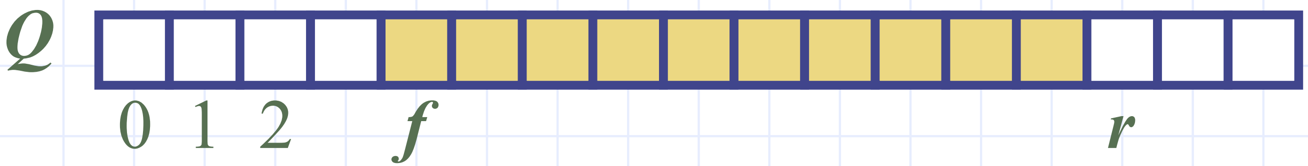
Applications of Queues

- 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

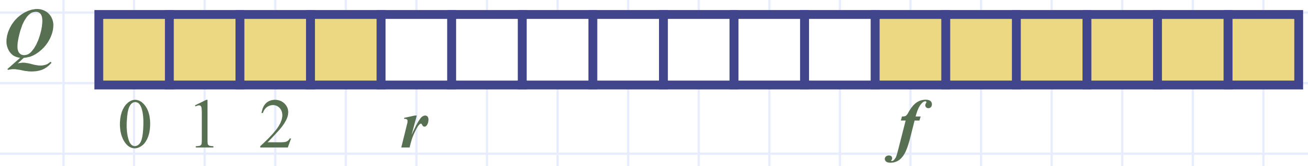
Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

normal configuration



wrapped-around configuration

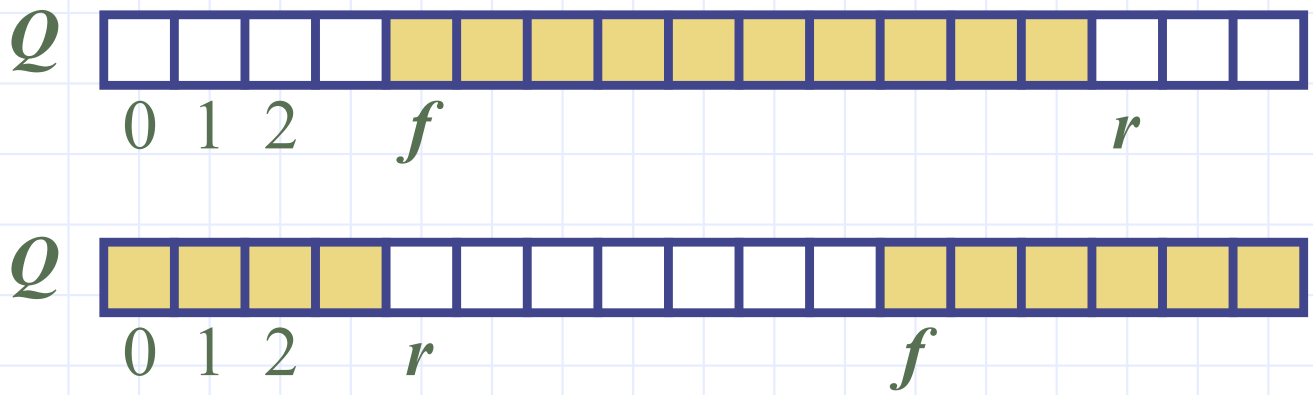


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return *sz*

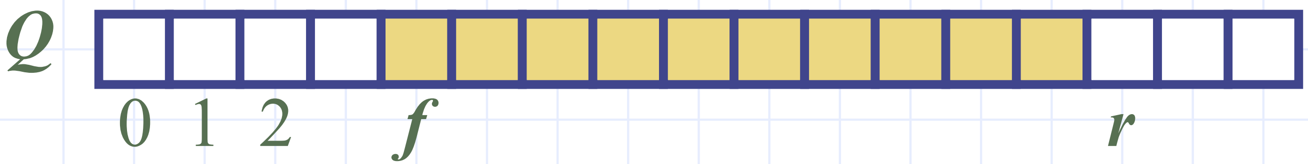
Algorithm *isEmpty*()
  return (*sz* == 0)

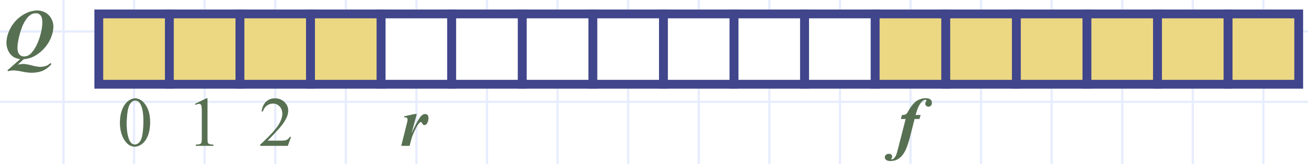


Queue Operations (cont.)

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

Algorithm *enqueue*(o)
  if *size*() = $N - 1$ **then**
    throw *IllegalStateException*
  else
    $r \leftarrow (f + sz) \bmod N$
    $Q[r] \leftarrow o$
    $sz \leftarrow (sz + 1)$

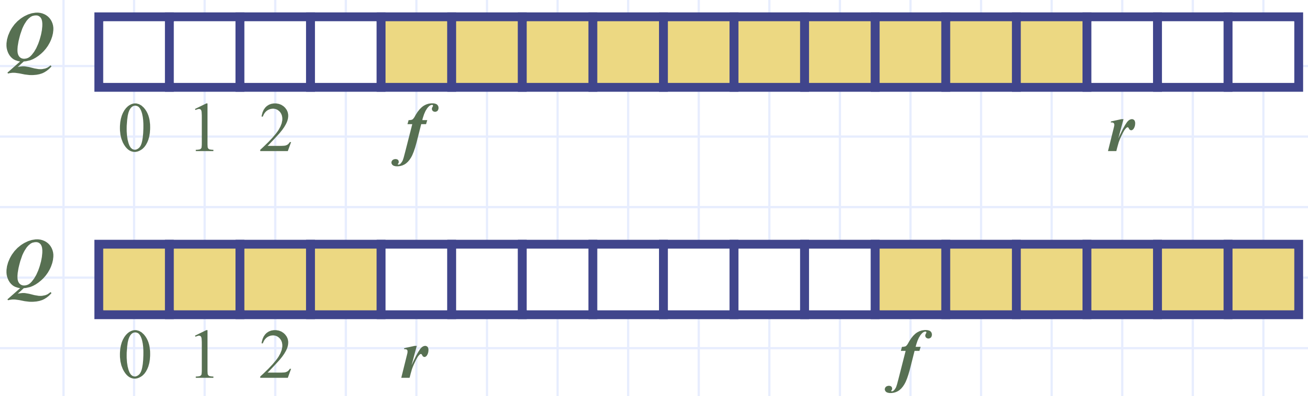




Queue Operations (cont.)

- Note that operation dequeue returns null if the queue is empty

Algorithm *dequeue*()
  if *isEmpty*() **then**
    return *null*
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    $sz \leftarrow (sz - 1)$
    return *o*



Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Assumes that first() and dequeue() return null if queue is empty

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

Array-based Implementation

```
/** Implementation of the queue ADT using a fixed-length array. */
public class ArrayQueue<E> implements Queue<E> {
  // instance variables
  private E[ ] data;                          // generic array used for storage
  private int f = 0;                          // index of the front element
  private int sz = 0;                         // current number of elements

  // constructors
  public ArrayQueue() {this(CAPACITY);}       // constructs queue with default capacity
  public ArrayQueue(int capacity) {           // constructs queue with given capacity
    data = (E[ ]) new Object[capacity];       // safe cast; compiler may give warning
  }

  // methods
  /** Returns the number of elements in the queue. */
  public int size() { return sz; }

  /** Tests whether the queue is empty. */
  public boolean isEmpty() { return (sz == 0); }

```

Array-based Implementation (2)

```
/** Inserts an element at the rear of the queue. */
public void enqueue(E e) throws IllegalStateException {
  if (sz == data.length) throw new IllegalStateException("Queue is full");
  int avail = (f + sz) % data.length;      // use modular arithmetic
  data[avail] = e;
  sz++;
}

/** Returns, but does not remove, the first element of the queue (null if empty). */
public E first() {
  if (isEmpty()) return null;
  return data[f];
}

/** Removes and returns the first element of the queue (null if empty). */
public E dequeue() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null;                          // dereference to help garbage collection
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```

Comparison to java.util.Queue

- Our Queue methods and corresponding methods of java.util.Queue:

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue()	remove()	poll()
first()	element()	peek()
size()	size()	
isEmpty()	isEmpty()	

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)