Stacks, Queues, and Deques

- Define **Stack** and **Queue** Abstract Data Types
  - Implement an **array-based stack and queue**
  - Implement a **stack and a queue using a singly linked list**
  - Reverse an array using a stack.
- Define **Deque** Abstract Data Types
  - Implement a deque

Recursion - Review

Solve large problems by solving a smaller occurrence of the same problem

- A recursive method must contain:
  - *One or more stopping conditions*: under certain conditions, it would stop the method from calling itself again - known as **base case**
  - *One or more recursive calls*: this is when **a method calls itself -** known as **recursive cases**
  - The recursive cases must eventually lead to a base case

Recursion - Review

- Recursion Examples:
  - **Factorial** function
  - **Binary Search** – runs in **O(log n)** time
  - English **Ruler**
  - **File systems**
- **Linear recursion** – the method performs a single recursive call
  - Linear sum
  - Reversing an array
- **Binary recursion** - the method performs two recursive calls for each non-base case
  - English ruler
  - Binary sum

Recursion - Review

- Analyzing recursive algorithms
  - **Binary Search** – because each recursive call divides the size in half, it runs in **O(log n)** time
  - **Power function** $p(x,n)=x^n$ makes n recursive calls, hence it runs in **O(n)** time
  - **Recursive squaring** of power function uses repeated squaring and reduces the running time to **O(log n)**.
- **Fibonacci** numbers:
  - Binary version is exponential!
  - Linear version runs in O(n) time

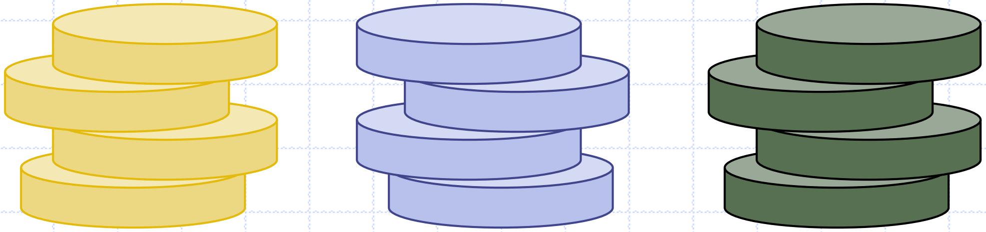
Recursion - Review

- Writing recursive methods:
  - Specify **method parameters** and **returning value**
  - Find a **base case**
  - Write the algorithm with **one or more recursive calls**

- Look for a data structure defined in terms of itself (inductively)

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



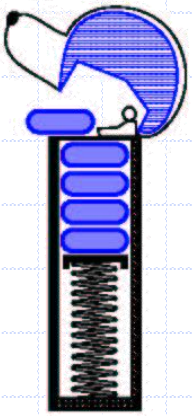
Abstract Data Types (ADTs)

- 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

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell **stocks**
  - The operations supported are
    - order **buy**(stock, shares, price)
    - order **sell**(stock, shares, price)
    - void **cancel**(order)
  - Error conditions:
    - Buy/sell a **nonexistent stock**
    - Cancel a **nonexistent order**

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the **last-in first-out** (LIFO) 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)

Exceptions vs. Returning Null

- Attempting the execution of an operation of an ADT may sometimes cause an error condition
- Java supports a general abstraction for errors, called **exception**
- An exception is said to be “thrown” by an operation that cannot be properly executed

- In **our Stack ADT, we do not use exceptions**
- Instead, we allow operations **pop** and **top** to be performed even if the stack is empty
- For an empty stack, **pop** and **top** simply return null

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

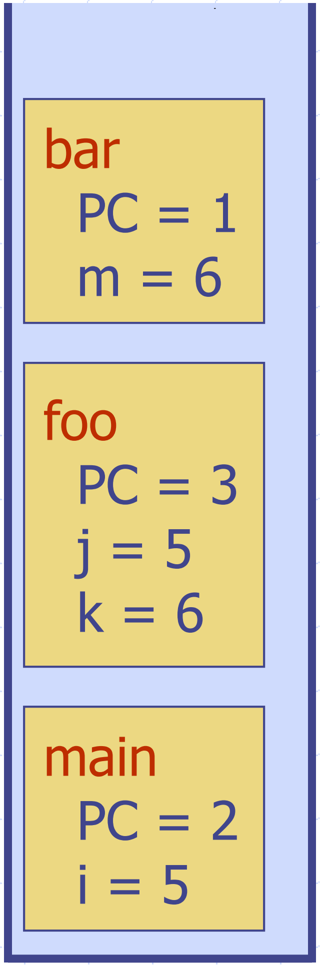
Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the **JVM pushes on the stack a frame** containing
  - Local **variables** and **return value**
  - **Program counter**, keeping track of the statements being executed
- When a method ends, its **frame is popped from the stack** and control is passed to the method on top of the stack
- Allows for recursion

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

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

bar(int m) {
  …
}
```

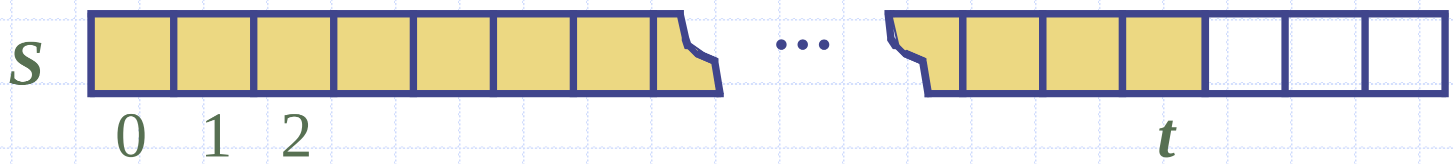


Array-based Stack

- A simple way of implementing the Stack ADT uses an **array**
- Index of the bottom element is 0.
- A variable keeps track of the index of the top element **t**

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

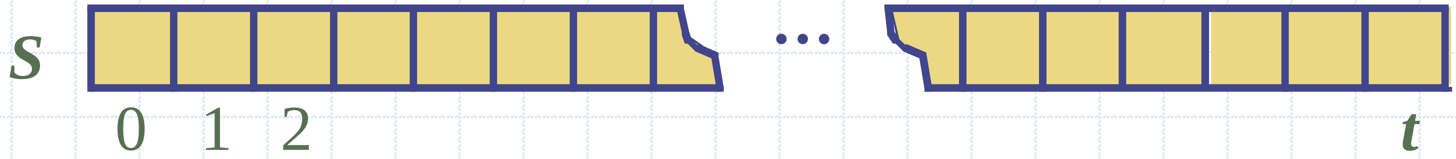
Algorithm *pop*()
    if *isEmpty*() **then**
        return null
    else
        $t \leftarrow t - 1$
        return $S[t + 1]$



Array-based Stack (cont.)

- The array storing the stack elements may become full (***t+1*** elements)
- 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

```
/** A generic method for reversing an array. */
  public static <E> void reverse(E[] a) {
    Stack<E> buffer = new ArrayStack<>(a.length);
    for (int i=0; i < a.length; i++)
      buffer.push(a[i]);
    for (int i=0; i < a.length; i++)
      a[i] = buffer.pop();
  }
/** Tester routine for reversing arrays */
  public static void main(String args[]) {
    Integer[] a = {4, 8, 15, 16, 23, 42};  // autoboxing allows this
    System.out.println("Reversing...");
    reverse(a);
    System.out.println("a = " + Arrays.toString(a));
}
```

Implementing a Stack with a Singly Linked List

- Linked-list approach has memory usage that is always proportional to the number of actual elements currently in the stack, and without an arbitrary capacity limit.
- With the **top of the stack stored at the front of the list**, all methods execute in constant time.
- Use the SinglyLinkedList class from chapter 3 to define a new **LinkedStack** class (**adapter pattern**)

Stack Method	*Singly Linked List Method*
size()	list.size()
isEmpty()	list.isEmpty()
push(e)	list.addFirst(e)
pop()	list.removeFirst()
top()	list.first()

Implementing a Stack with a Singly Linked List

- This class declares a SinglyLinkedList named *list* as a private field, and uses the corresponding methods as shown in previous slide.

```
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedStack() { }                 // new stack relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(E element) { list.addFirst(element); }
  public E top() { return list.first(); }
  public E pop() { return list.removeFirst(); }
}
```

Code Fragment 6.4: Implementation of a Stack using a SinglyLinkedList as storage.

Review of indexOf methods

- int indexOf(int ch) - returns index position for the given char value.
- int indexOf(int ch, int fromIndex) - returns index position for the given char value and from index.
- Example:

```
String s1="this is index of example";
//passing char value
int index1 = s1.indexOf('s'); //returns the index of s char value
System.out.println(index1); //3
//passing char value and index
int index2 = s1.indexOf('s',5); //returns the index after 5th index
System.out.println(index2); //6
```

Review of substring methods

- public String substring(int startIndex)
- public String substring(int startIndex, int endIndex)
- If you don't specify endIndex, **substring()** method will return all the characters from startIndex.
  - startIndex : starting index is inclusive
  - endIndex : ending index is exclusive
- Example:

```
String s1="javatpoint";
System.out.println(s1.substring(2,4)); //returns va
System.out.println(s1.substring(2)); //returns vatpoint
```

Parentheses Matching example

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

- An Algorithm for Matching Delimiters:
  - use a stack with a single left-to-right scan of the original string

Parentheses Matching example

- Each time we encounter an opening symbol, we push that symbol onto the stack
- Each time we encounter a closing symbol, we pop a symbol from the stack and check that these two symbols form a valid pair.
- If we reach the end of the expression and the stack is empty, then the original expression was properly matched.
- Otherwise, there must be an opening delimiter on the stack without a matching symbol.

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

- We make a **left-to-right pass** through the raw string, using index *j* to track our progress.
- The indexOf method of the String class, which optionally accepts a starting index as a second parameter, locates the '<' and '>' characters that define the tags.
- Method substring, also of the String class, returns the substring starting at a given index and optionally ending right before another given index.
- Opening tags are pushed onto the stack, and matched against closing tags as they are popped from the stack

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 stakcs:

- 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

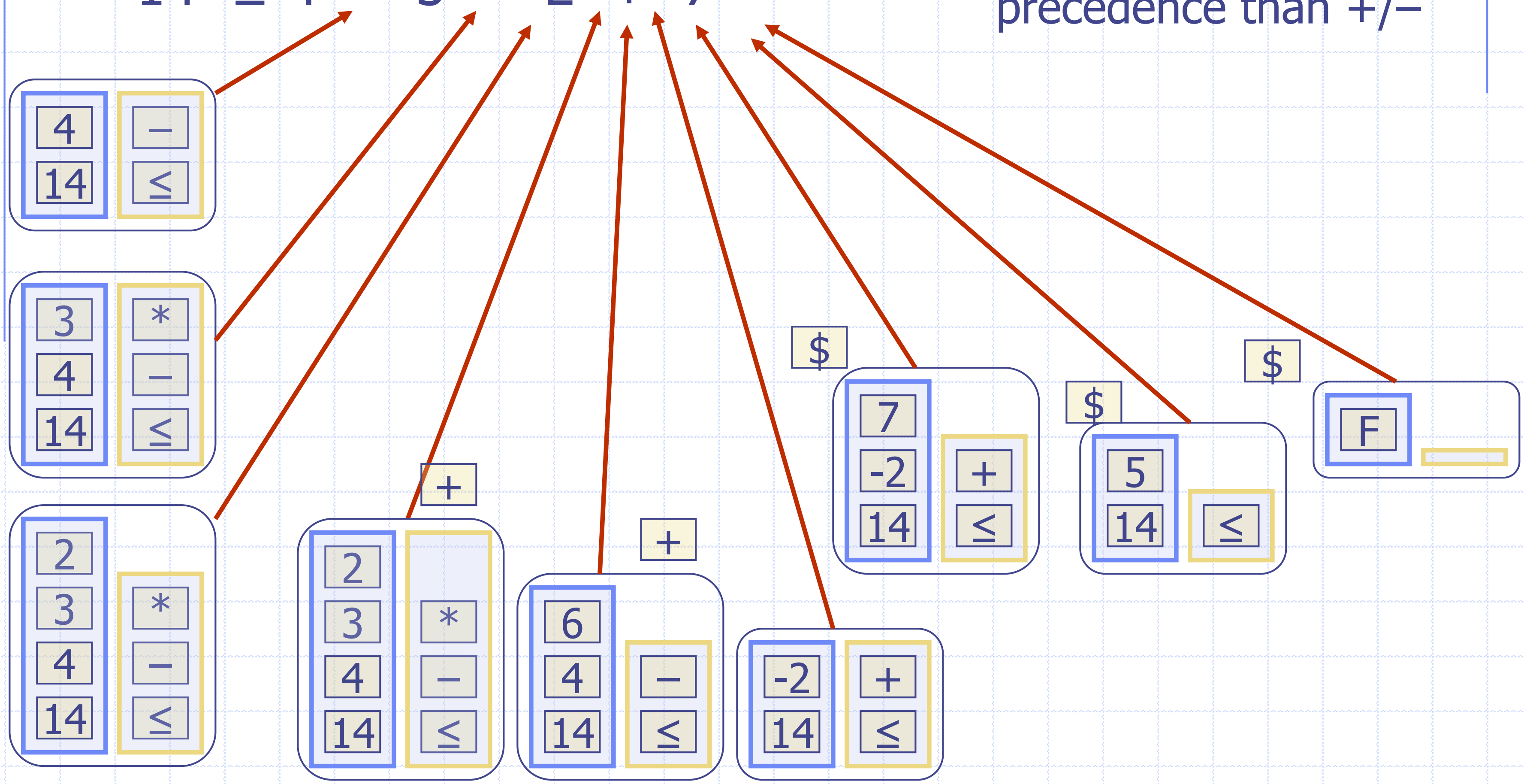
```
while there's another token z
   if isNumber(z) then
        valStk.push(z)
   else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression

Slide by Matt Stallmann included with permission.

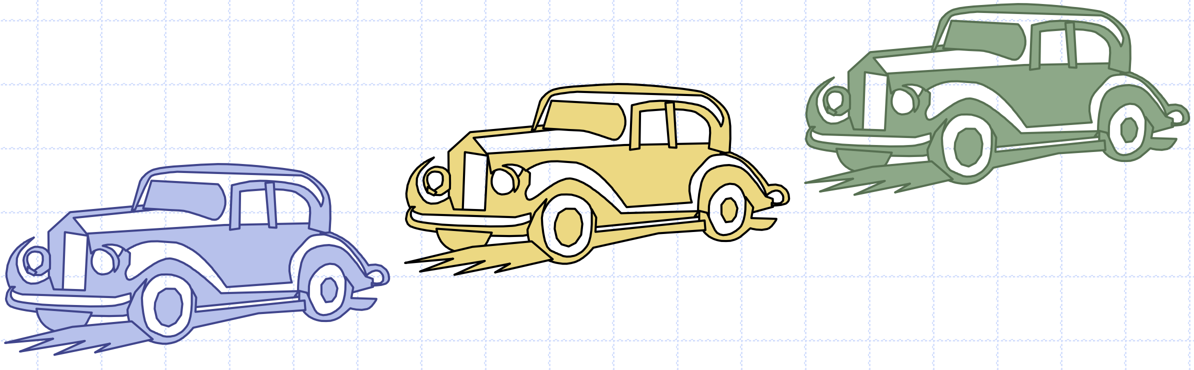
14 ≤ 4 – 3 * 2 + 7

Operator ≤ has lower precedence than +/–



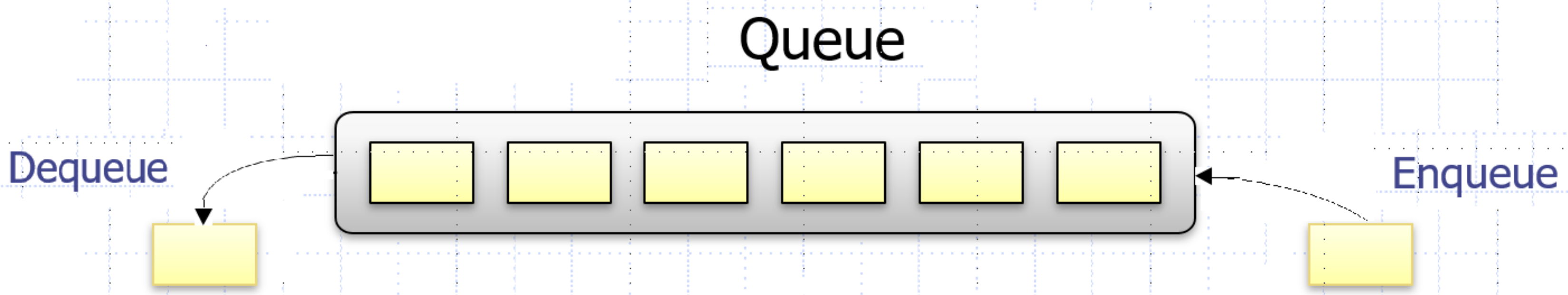
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Queues



Queues

- A queue is a linear data structure that represents a waiting list or line (first-in first-out):



- Elements are maintained in a sequence.

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the **first-in first-out** (FIFO) 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 li**sts, bureaucracy
  - **Access to shared resources** (e.g., printer)
  - **Multiprogramming**
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

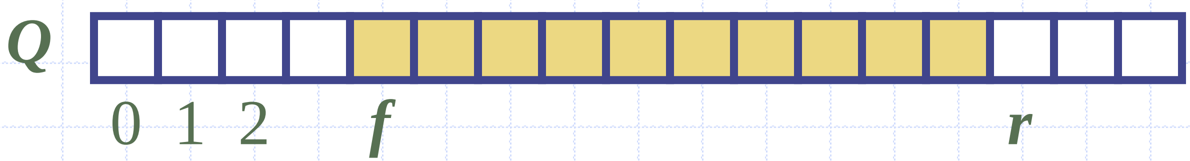
Array-based Queue

- Use an **array of size N** in a **circular fashion**
- Two variables keep track of the **front** and **size:**

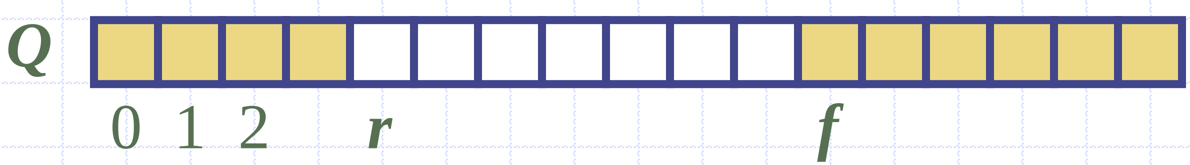
  f - index of the **front** element

  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



- **N** = 17
  - f = 4
  - sz = 10
  - r = 14

  - f = 11
  - sz = 10
  - r = 4

Queue Operations

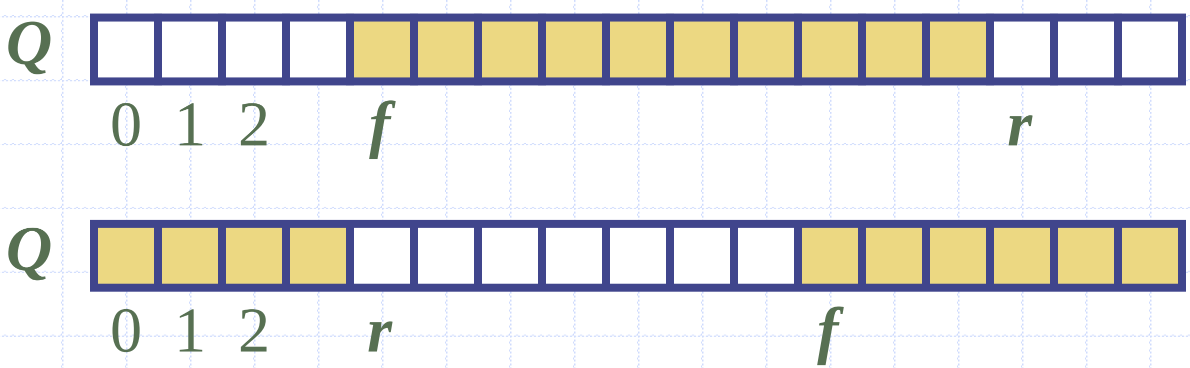
- We use the **modulo** operator (remainder of division) to calculate *r* given f and sz:

$$r = (f + sz) \bmod N$$

Algorithm *size*()
  return *sz*

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

- **N** = 17



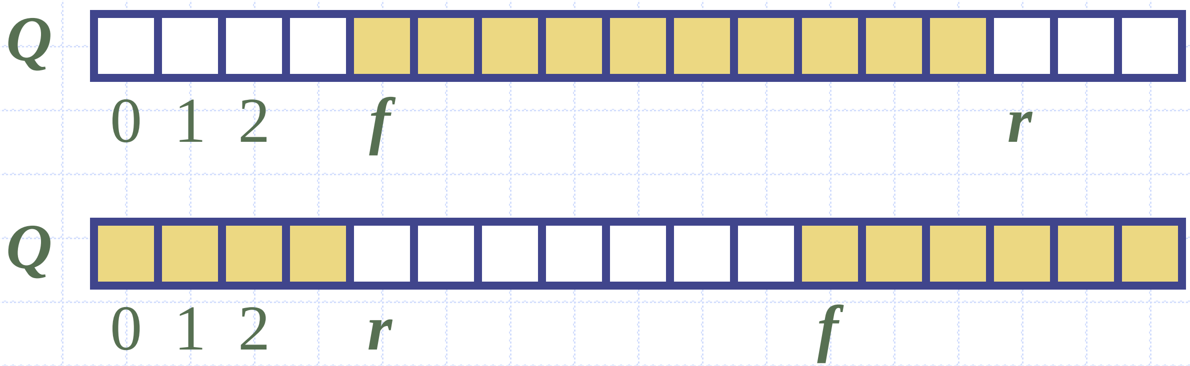
- f = 4
- sz = 10
- r = 14

- f = 11
- sz = 10
- r = 4

Queue Operations (cont.)

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

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



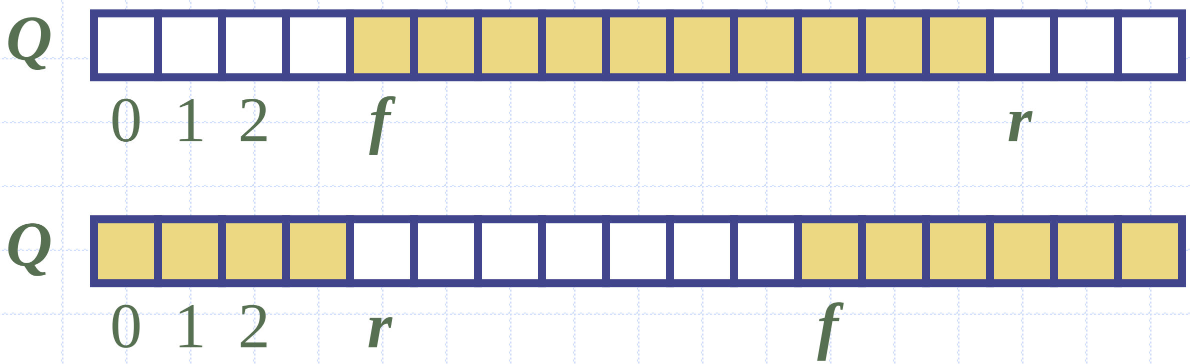
Queue Operations (cont.)

- Note that operation **dequeue** returns null if the queue is empty
- When we dequeue an element use:

  $f = (f + 1) \% N$

  to update *f*.

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



- **N** = 17
- f = 4 becomes f= 5.
- f = 11 becomes f = 12.

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;
}
```

Implementing a Queue with a Singly Linked List

```
/** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedQueue() { }                     // new queue relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
}
```

Code Fragment 6.11: Implementation of a Queue using a SinglyLinkedList.

- Each method of SinglyLinkedList class runs in $O(1)$ worst-case time.
- Therefore, each method of our LinkedQueue adaptation also runs in $O(1)$ worst-case time.

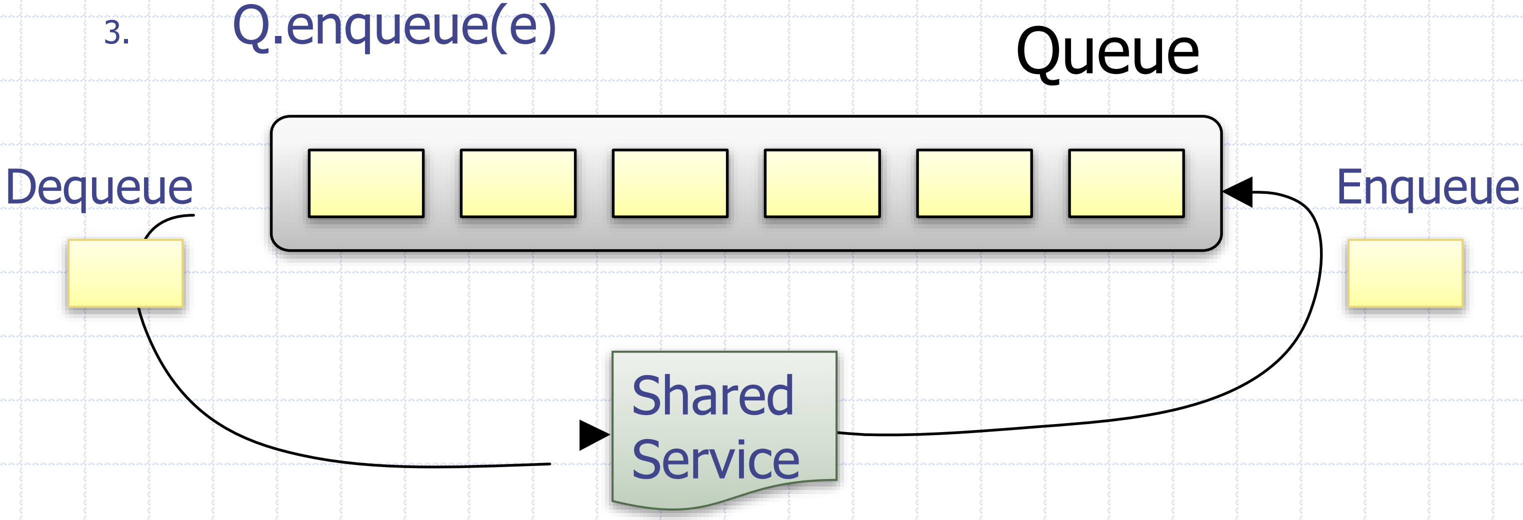
Comparison to java.util.Queue

- 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)



Double-Ended Queues

- A queue-like data structure that **supports insertion and deletion at both the front and the back of the queue**.
  - Such a structure is called a ***double-ended queue***, or ***deque***, which is usually pronounced "deck".
- The Deque Abstract Data Type:
  - **addFirst**(*e*): Insert a new element ***e*** at the front of the deque.
  - **addLast**(*e*): Insert a new element ***e*** at the back of the deque.
  - **removeFirst**( ): Remove and return the first element of the deque (or null if the deque is empty).
  - **removeLast**( ): Remove and return the last element of the deque (or null if the deque is empty).

Double-Ended Queues

- Additionally, the deque ADT will include the following accessors:
  - **first**( ): Returns the first element of the deque, without removing it (or null if the deque is empty).
  - **last**( ): Returns the last element of the deque, without removing it (or null if the deque is empty).
  - **size**( ): Returns the number of elements in the deque.
  - **isEmpty**( ): Returns a boolean indicating whether the deque is empty.

Double-Ended Queues

```
public interface Deque<E> { 6 /** Returns the number of elements in the deque. */
        int size( );
        /** Tests whether the deque is empty. */
        boolean isEmpty( );
        /** Returns, but does not remove, the first element of the deque (null if empty). */
        E first( );
         /** Returns, but does not remove, the last element of the deque (null if empty). */
        E last( );
        /** Inserts an element at the front of the deque. */
        void addFirst(E e);
        /** Inserts an element at the back of the deque. */
        void addLast(E e);
        /** Removes and returns the first element of the deque (null if empty). */
        E removeFirst( );
        /** Removes and returns the last element of the deque (null if empty). */
        E removeLast( );
}
```

Double-Ended Queues

- **Implementing a Deque**
  - Using a Circular Array - a representation similar to the ArrayQueue class
  - Using a Doubly Linked List
    - The DoublyLinkedList class from Section 3.4.1 already implements the entire Deque interface; we simply need to add the declaration “implements Deque<E>” to that class definition in order to use it as a deque.
- **Performance of the Deque Operations**
  - every method runs in $O(1)$ time.