Chapter 6 Stacks, Queues, and Deques

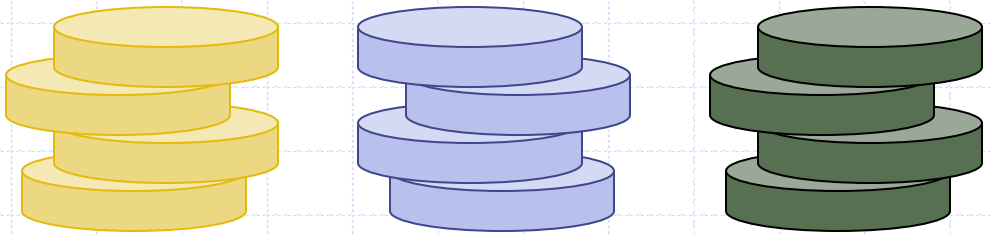
- Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

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 orders
  - 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

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 (**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)		
push(9)		
top( )		
push(4)		
size( )		
pop( )		
push(6)		
push(8)		
pop( )		

Example (cont.)

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

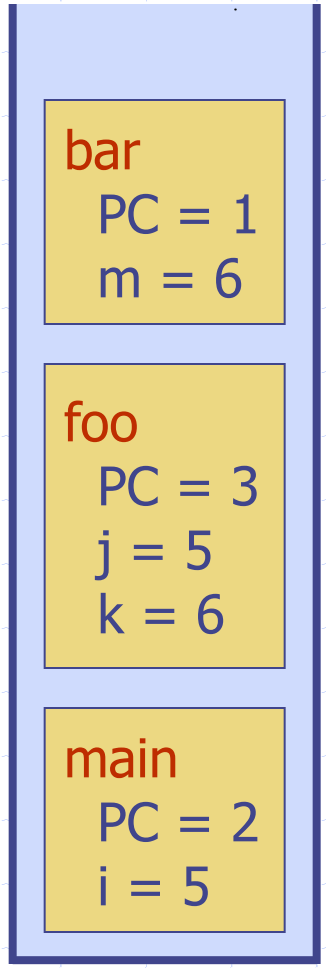
Applications of Stacks

- 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 statement 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

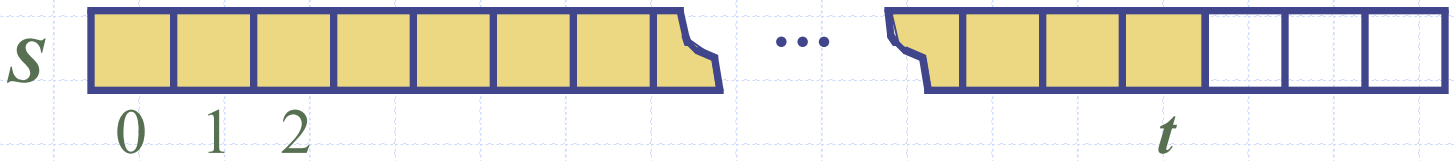
```
main() {
  int i = 5;
  foo(i);
  }

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
- We add elements from left to right
- A variable keeps track of the index of the top element

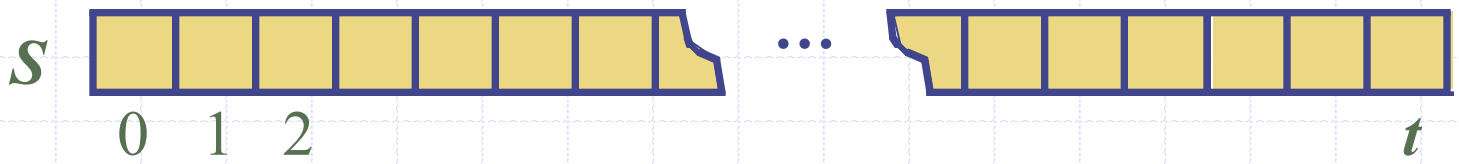
```
Algorithm size()
  return t + 1

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 ← t + 1
    S[t] ← 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)
```

Application: Parsing

Most parsing uses stacks

Examples includes:

- Matching tags in XHTML
- In C++, matching
  - parentheses (. . .)
  - brackets, and [. . .]
  - braces { . . . }

Parsing XHTML

A *markup language* is a means of annotating a document to given context to the text

- The annotations give information about the structure or presentation of the text

The best known example is HTML, or HyperText Markup Language

- We will look at XHTML

Parsing XHTML

XHTML is made of nested

- *opening tags*, e.g., `<some_identifier>`, and
- matching *closing tags*, e.g., `</some_identifier>`

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

Parsing XHTML

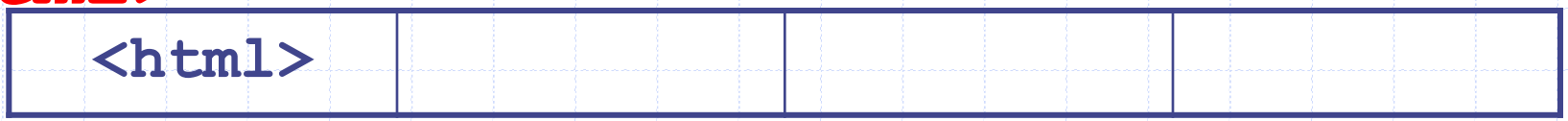
Nesting indicates that any closing tag must match the most recent opening tag

Strategy for parsing XHTML:

- read though the XHTML linearly
- place the opening tags in a stack
- when a closing tag is encountered, check that it matches what is on top of the stack

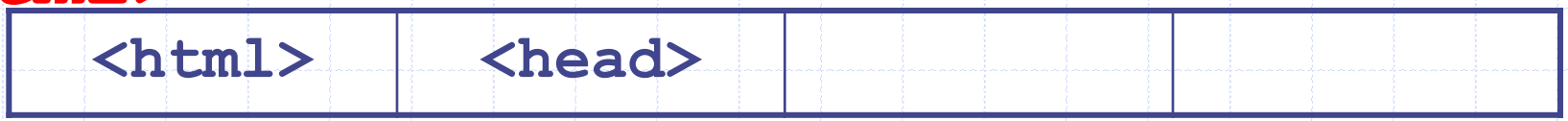
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



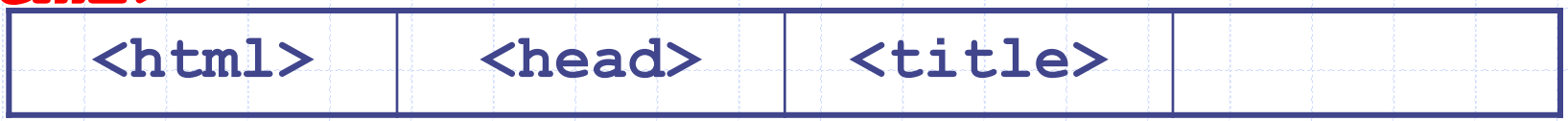
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



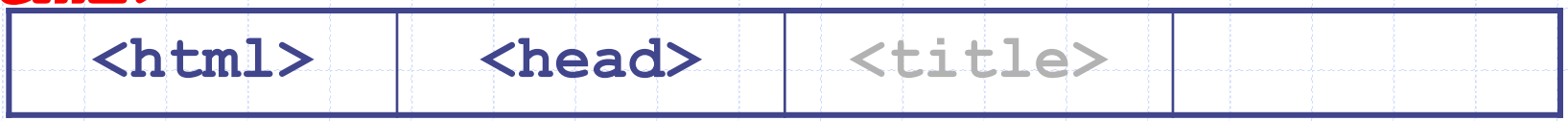
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



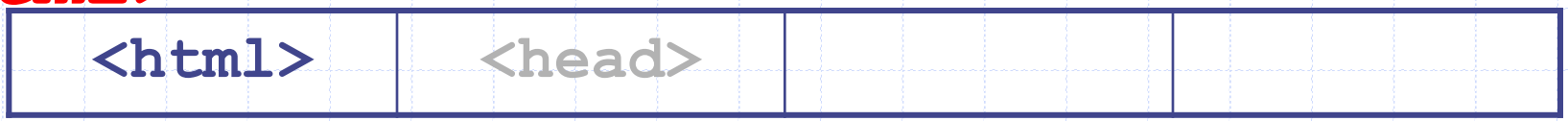
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



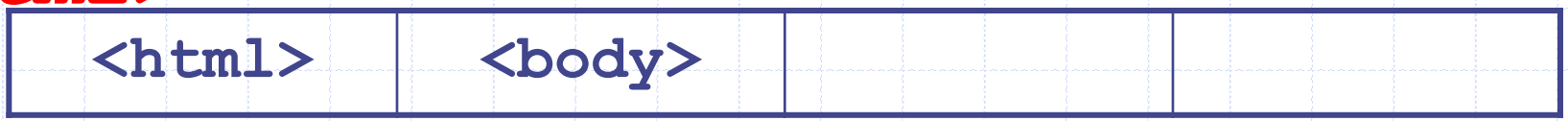
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



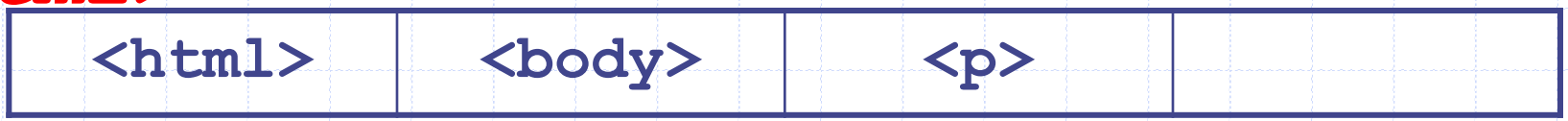
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



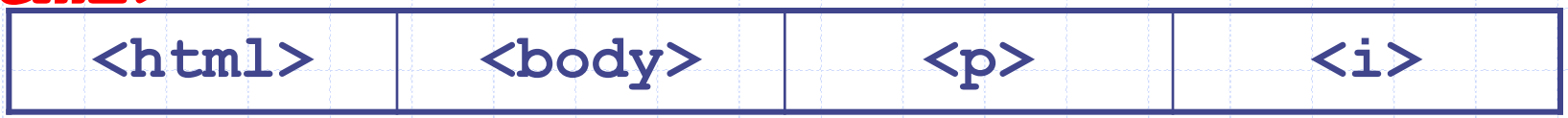
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

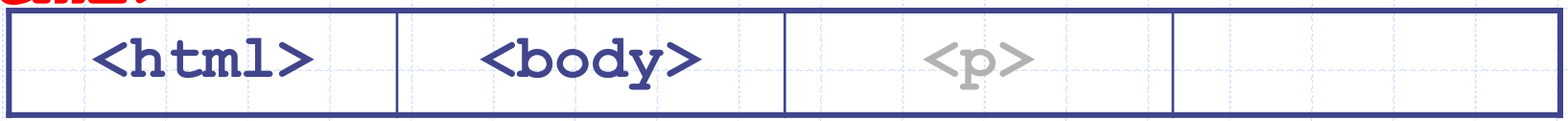


Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

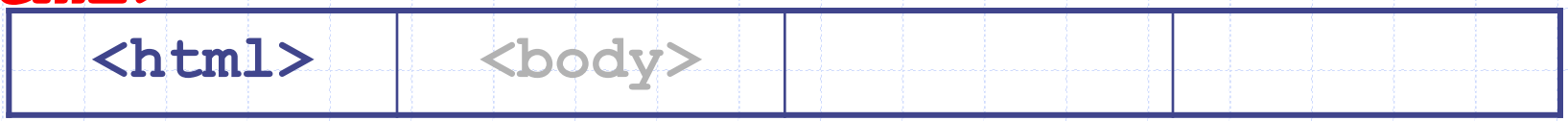
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



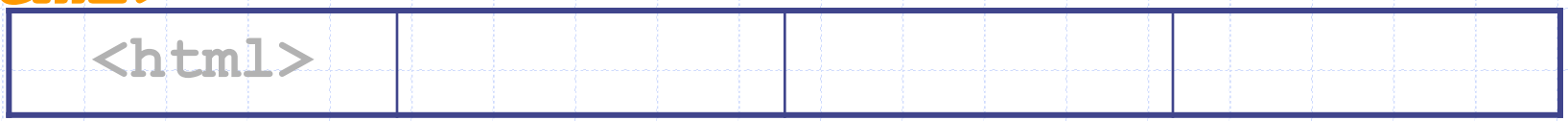
Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



Parsing XHTML

We are finished parsing, and the stack is empty

Possible errors:

- a closing tag which does not match the opening tag on top of the stack
- a closing tag when the stack is empty
- the stack is not empty at the end of the document

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?
}
```

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<>( );
}
```

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
      }
   }
  return buffer.isEmpty( ); // were all opening delimiters matched?
}
```

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?
}
```

Recaps of Stack ADT

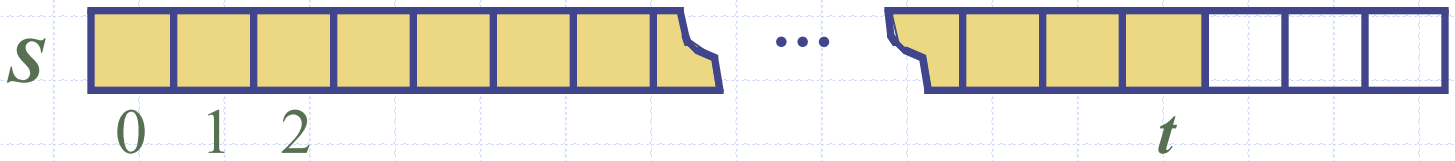
- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out (**LIFO**) scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element

- 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

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]
```



Implementing Stacks with a Singly Linked List

- Should the top of the stack be at the front or back of the list?
- How to implement each method of stacks using methods of SinglyLinkedList?
- Efficiency of this implementation?

Implementing Stacks with a Singly Linked List

- Answer: since we can insert and delete elements in constant time only at the front. With the top of the stack stored at the front of the list, all methods execute in constant time.

Implementation using SinglyLinkedList

Stack Method	*Singly Linked List Method*
size( )	
isEmpty( )	
push(*e*)	
pop( )	
top( )	

Implementation using SinglyLinkedList

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

Implementation using SinglyLinkedList (cont.)

```
1  public class LinkedStack<E> implements Stack<E> {
2    private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
3    public LinkedStack() { }                    // new stack relies on the initially empty list
4    public int size() { return list.size(); }
5    public boolean isEmpty() { return list.isEmpty(); }
6    public void push(E element) { list.addFirst(element); }
7    public E top() { return list.first(); }
8    public E pop() { return list.removeFirst(); }
9  }
```

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

Evaluating Arithmetic Expressions

$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$

Operator precedence

* has precedence over +/–

Operator ≤ has lower precedence than +/–

Associativity

operators of the same precedence group evaluated from left to right

Example: $(x - y) + z$ rather than $x - (y + z)$

Evaluating Arithmetic Expressions

$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$

Operator precedence

* has precedence over +/–

Operator ≤ has lower precedence than +/–

Associativity

operators of the same precedence group evaluated from left to right

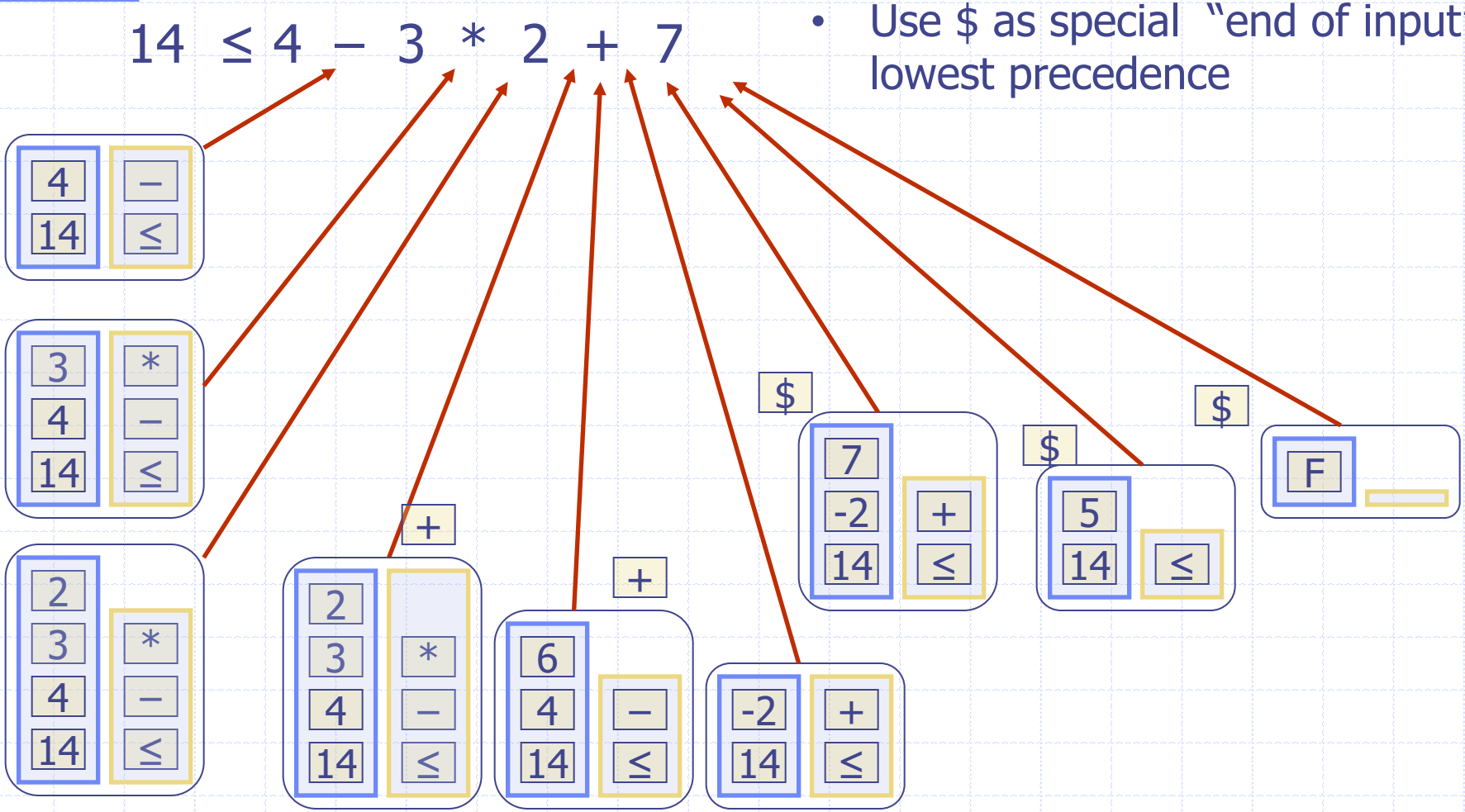
Example: $(x - y) + z$ rather than $x - (y + z)$

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

Algorithm on an Example Expression

Slide by Matt Stallmann included with permission.

- Operator ≤ has lower precedence than +/–
- Use $ as special "end of input" token with lowest precedence



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

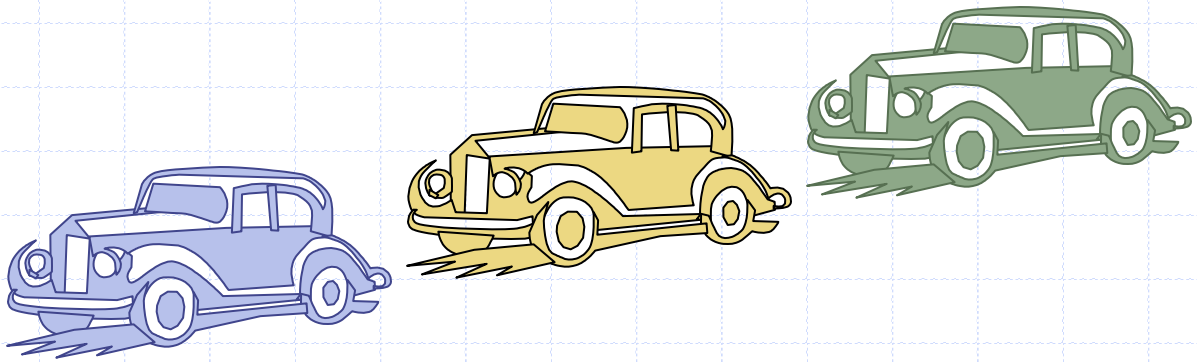
```
while there's another token z
  if isNumber(z) then
        valStk.push(z)
  else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Exercise C-6.19

- ***Postfixnotation*** is an unambiguous way of writing an arithmetic expression without parentheses. It is defined so that if "(*exp*1)**op**(*exp*2)" is a normal fully parenthesized expression whose operation is **op**, the postfix version of this is "*pexp*1 *pexp*2 **op**", where *pexp*1 is the postfix version of *exp*1 and *pexp*2 is the postfix version of *exp*2. The postfix version of a single number or variable is just that number or variable. So, for example, the postfix version of "((5 + 2) ∗ (8 − 3))/4" is "5 2 + 8 3 − ∗ 4 /". Describe a nonrecursive way of evaluating an expression in postfix notation.

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 (**FIFO**)
- 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()			
enqueue(9)			
enqueue(7)			
size()			
enqueue(3)			
enqueue(5)			
dequeue()			

Example

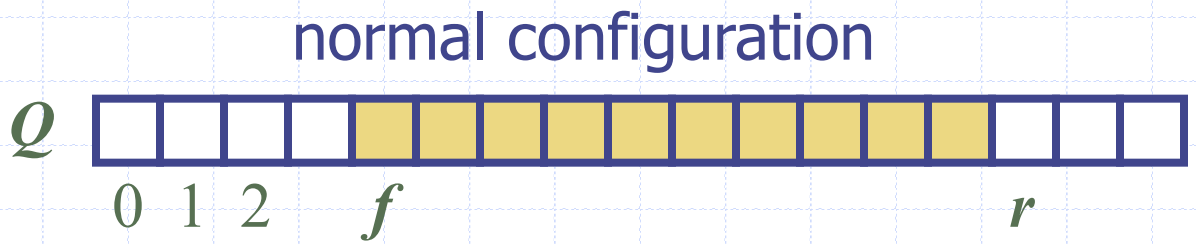
Operation		*Output*	*Q*
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	*5*	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	*3*	(7)	
first()	*7*	(7)	
dequeue()	*7*	()	
dequeue()	*null*	()	
isEmpty()	*true*	()	
enqueue(9)	–	(9)	
enqueue(7)	–	(9, 7)	
size()	*2*	(9, 7)	
enqueue(3)	–	(9, 7, 3)	
enqueue(5)	–	(9, 7, 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

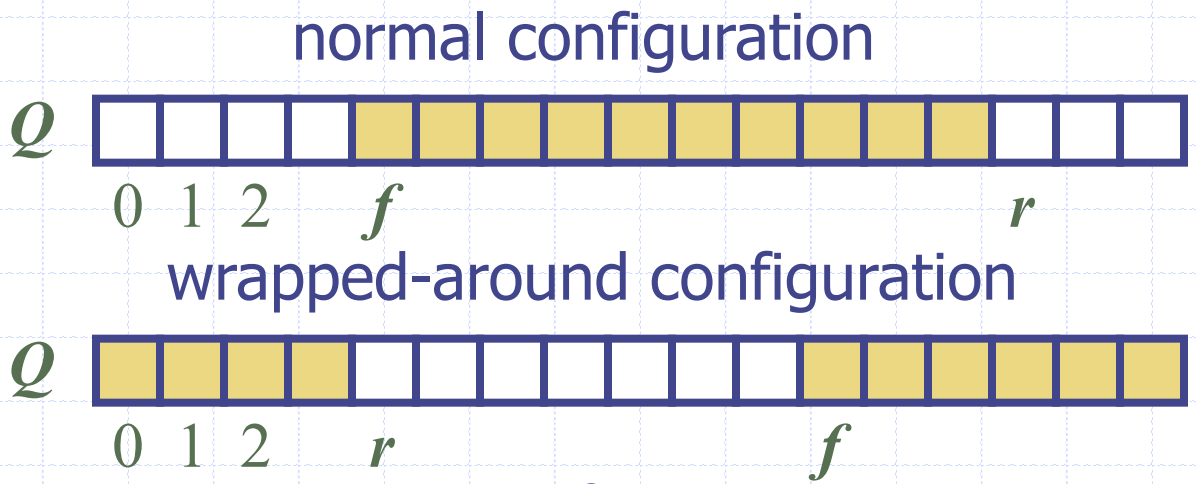
Array-based Queue

- Use an array of size N
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements



Array-based Queue

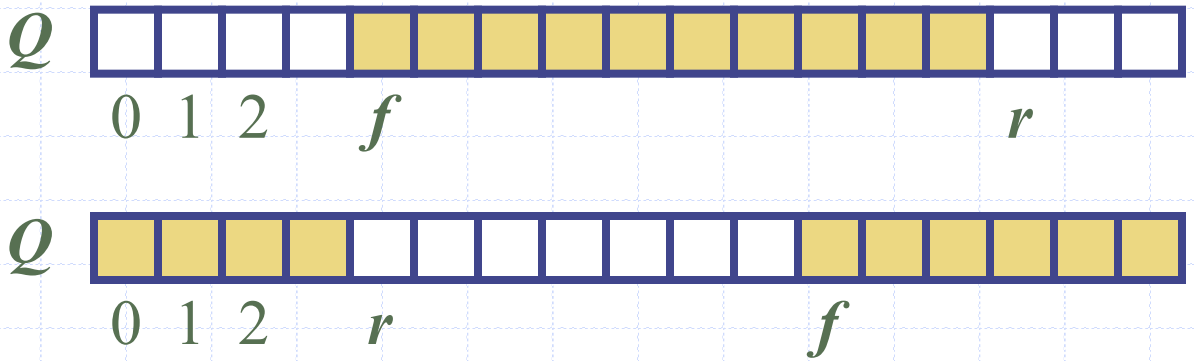
- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue



Queue Operations

- We use the modulo operator (remainder of division)

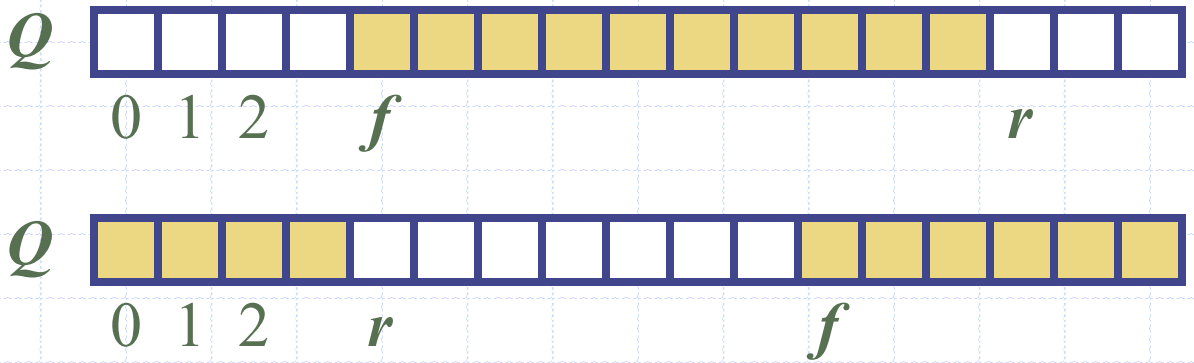
```
Algorithm size()
  return sz

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



Queue Operations (cont.)

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

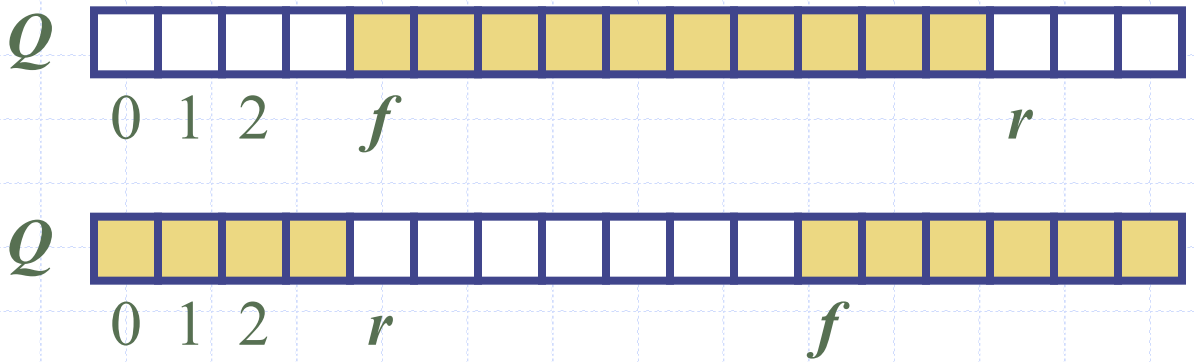
Algorithm *enqueue*(o)
  if $size() = N - 1$ **then**
    throw *IllegalStateException*
  else
    $r \leftarrow (f + sz) \bmod N$
    $Q[r] \leftarrow o$
    $sz \leftarrow (sz + 1)$



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



Array-based Implementation

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                          // generic array used for storage
5    private int f = 0;                          // index of the front element
6    private int sz = 0;                         // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
10   public ArrayQueue(int capacity) {          // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
```

Array-based Implementation (2)

```
21    /** Inserts an element at the rear of the queue. */
22    public void enqueue(E e) throws IllegalStateException {
23      if (sz == data.length) throw new IllegalStateException("Queue is full");
24      int avail = (f + sz) % data.length;       // use modular arithmetic
25      data[avail] = e;
26      sz++;
27    }
28
29    /** Returns, but does not remove, the first element of the queue (null if empty). */
30    public E first( ) {
31      if (isEmpty( )) return null;
32      return data[f];
33    }
34
35    /** Removes and returns the first element of the queue (null if empty). */
36    public E dequeue( ) {
37      if (isEmpty( )) return null;
38      E answer = data[f];
39      data[f] = null;                           // dereference to help garbage collection
40      f = (f + 1) % data.length;
41      sz--;
42      return answer;
43    }
```

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( )	

Implementing a Queue with a Singly Linked List

- Worst-case $O(1)$-time for all operations
- How to implement each method of queues using methods of SinglyLinkedList?

Implementing a Queue with a Singly Linked List (cont.)

```
1  /** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
2  public class LinkedQueue<E> implements Queue<E> {
3    private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
4    public LinkedQueue( ) { }                    // new queue relies on the initially empty list
5    public int size( ) { return list.size( ); }
6    public boolean isEmpty( ) { return list.isEmpty( ); }
7    public void enqueue(E element) { list.addLast(element); }
8    public E first( ) { return list.first( ); }
9    public E dequeue( ) { return list.removeFirst( ); }
10 }
```

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

Implementing a Queue with a Singly Linked List (cont.)

- Worst-case $O(1)$-time for all operations,
- Without any artificial limit on the capacity.
- Because each node stores a next reference, in addition to the element reference, a linked list uses more space per element than a properly sized array of references.
- In practice, the linked-list method is more expensive than the array-based method

Exercise R-6.7

- Suppose an initially empty queue Q has performed a total of 32 enqueue operations, 10 first operations, and 15 dequeue operations, 5 of which returned null to indicate an empty queue. What is the current size of Q?

Round-Robin Scheduling

- Each active process is given its own time slice
- A process is interrupted when the slice ends
- Processes take turns in a cyclic order

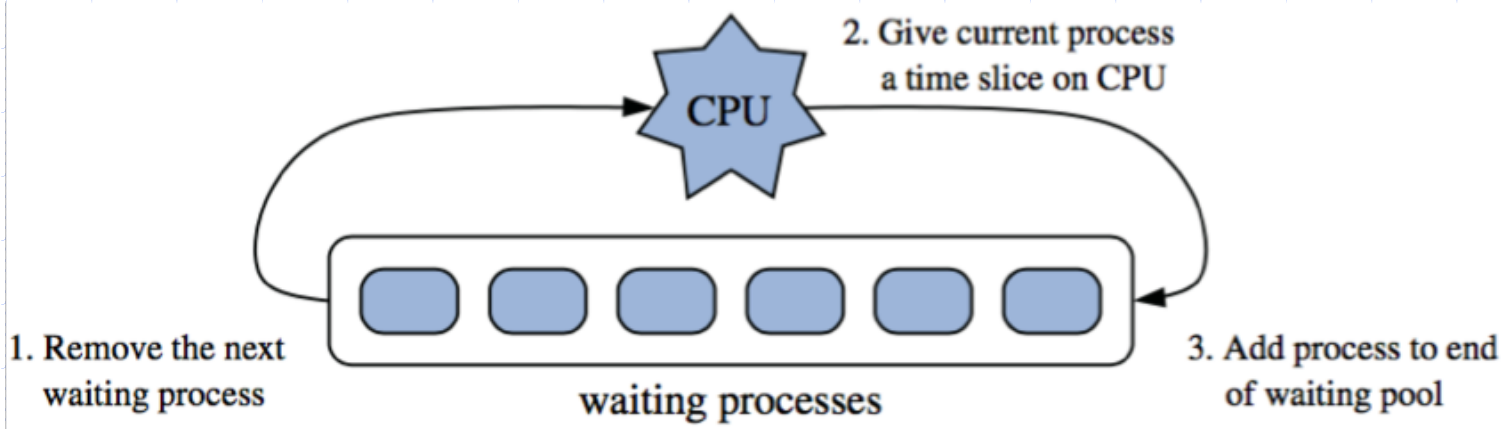


Figure 3.15: The three iterative steps for round-robin scheduling.

Implementation Using Circularly Linked List

- One additional update method:
  - rotate(): Moves the first element to the end of the list.

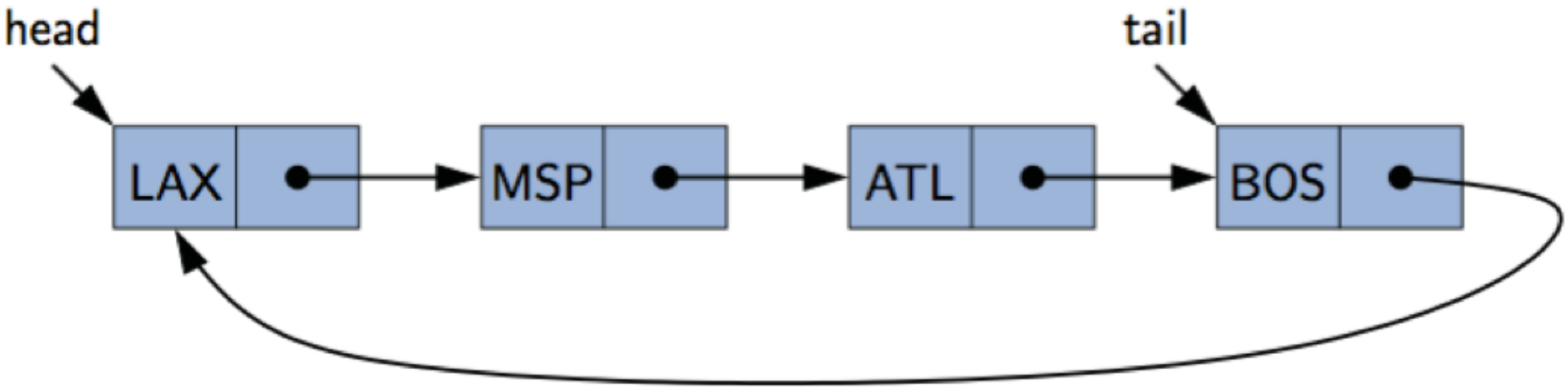
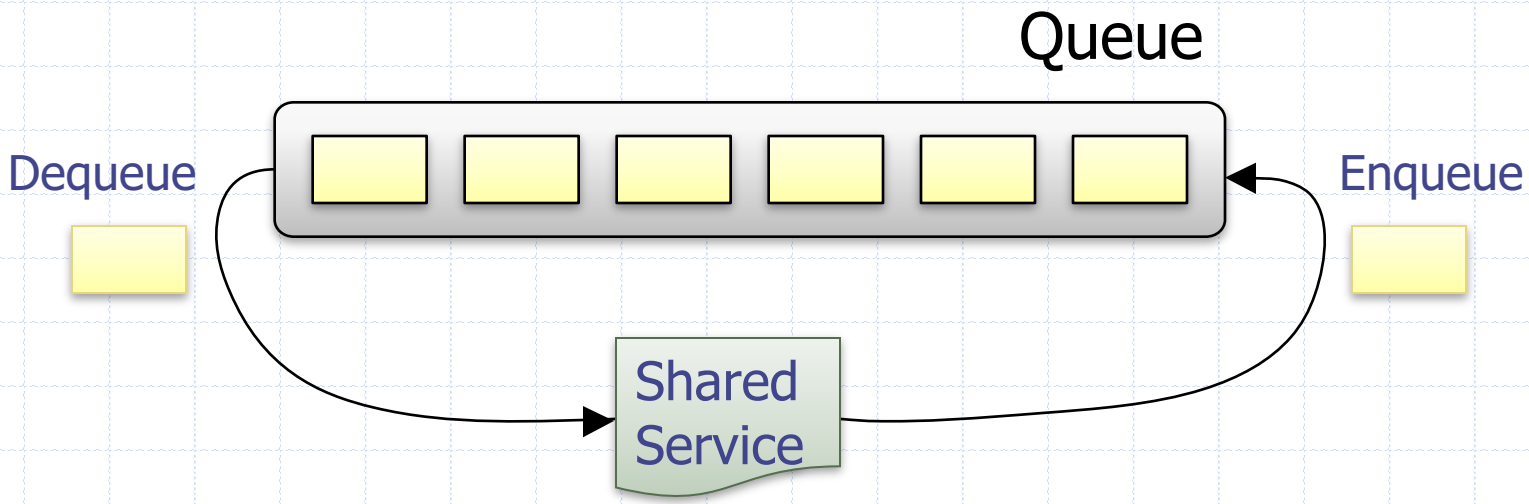


Figure 3.16: Example of a singly linked list with circular structure.

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)



A Circular Queue

- A new LinkedCircularQueue class can be produced by adapting the CircularlyLinkedList class of Section 3.3

```
1  public interface CircularQueue<E> extends Queue<E> {
2    /**
3     * Rotates the front element of the queue to the back of the queue.
4     * This does nothing if the queue is empty.
5     */
6    void rotate();
7  }
```

The Josephus Problem

- People are standing in a circle waiting to be executed. Counting begins at a specified point in the circle and proceeds around the circle in a specified direction. After a specified number of people are skipped, the next person is executed. The procedure is repeated with the remaining people, starting with the next person, going in the same direction and skipping the same number of people, until only one person remains, and is freed.
- **The problem** — given the number of people, starting point, direction, and number to be skipped — is to choose the position in the initial circle to avoid execution.

Titus Flavius Josephus
(37 CE – 100 CE)

The Josephus Problem

```
1  public class Josephus {
2    /** Computes the winner of the Josephus problem using a circular queue. */
3    public static <E> E Josephus(CircularQueue<E> queue, int k) {
4      if (queue.isEmpty( )) return null;
5      while (queue.size( ) > 1) {
6        for (int i=0; i < k−1; i++)     // skip past k-1 elements
7          queue.rotate( );
8        E e = queue.dequeue( );         // remove the front element from the collection
9        System.out.println("       " + e + " is out");
10       }
11     return queue.dequeue( );          // the winner
12   }
13
14   /** Builds a circular queue from an array of objects. */
15   public static <E> CircularQueue<E> buildQueue(E a[ ]) {
16     CircularQueue<E> queue = new LinkedCircularQueue<>( );
17     for (int i=0; i<a.length; i++)
18       queue.enqueue(a[i]);
19     return queue;
20   }
21
22   /** Tester method */
23   public static void main(String[ ] args) {
24     String[ ] a1 = {"Alice", "Bob", "Cindy", "Doug", "Ed", "Fred"};
25     String[ ] a2 = {"Gene", "Hope", "Irene", "Jack", "Kim", "Lance"};
26     String[ ] a3 = {"Mike", "Roberto"};
27     System.out.println("First winner is " + Josephus(buildQueue(a1), 3));
28     System.out.println("Second winner is " + Josephus(buildQueue(a2), 10));
29     System.out.println("Third winner is " + Josephus(buildQueue(a3), 7));
30   }
31 }
```

Deque: Double-Ended Queues

- Queue-like data structure
- Supports insertion and deletion at both the front and back of the queue.
- More general than both the stack and the queue ADTs.

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).

Deque ADT (cont.)

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.

An Example

Example 6.5: *The following table shows a series of operations and their effects on an initially empty deque D of integers.*

Method	Return Value	D
addLast(5)	–	(5)
addFirst(3)	–	(3, 5)
addFirst(7)	–	(7, 3, 5)
first( )	7	(7, 3, 5)
removeLast( )	5	(7, 3)
size( )		
removeLast( )		
removeFirst( )		
addFirst(6)		
last( )		
addFirst(8)		
isEmpty( )		
last( )		

An Example

Example 6.5: *The following table shows a series of operations and their effects on an initially empty deque D of integers.*

Method	Return Value	D
addLast(5)	–	(5)
addFirst(3)	–	(3, 5)
addFirst(7)	–	(7, 3, 5)
first( )	7	(7, 3, 5)
removeLast( )	5	(7, 3)
size( )	2	(7, 3)
removeLast( )	3	(7)
removeFirst( )	7	( )
addFirst(6)	–	(6)
last( )	6	(6)
addFirst(8)	–	(8, 6)
isEmpty( )	false	(8, 6)
last( )	6	(8, 6)

Implementation

- Circular Array
- Doubly Linked List
- Performance of the Deque Operations

Method	Running Time
size, isEmpty	$O(1)$
first, last	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

Exercise C-6.31

- Describe how to implement the deque ADT using two stacks as the only instance variables. What are the running times of the methods?

Summary

- Stacks
- Queues
- Deques