Stacks and Queues

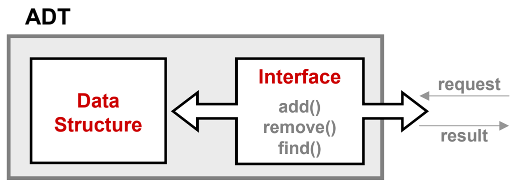
EECS2030: Advanced Object Oriented Programming
Fall 2017

CHEN-WEI WANG

What is a Stack?

- A *stack* is a collection of objects.
- Objects in a *stack* are inserted and removed according to the *last-in, first-out (LIFO)* principle.
  - *Cannot* access *arbitrary* elements of a stack
  - *Can* only access or remove the *most-recently inserted* element

The Stack ADT



- *Accessors*
  - ***top***
  - ***size***
  - ***isEmpty***
- *Mutators*
  - ***push***
  - ***pop***

Stack: Illustration

OPERATION	RETURN VALUE	STACK CONTENTS
–	–	∅
isEmpty	*true*	∅
push(5)	–	5
push(3)	–	$\begin{array}{c}3\\5\end{array}$
push(1)	–	$\begin{array}{c}1\\3\\5\end{array}$
size	3	$\begin{array}{c}1\\3\\5\end{array}$
top	1	$\begin{array}{c}1\\3\\5\end{array}$
pop	1	$\begin{array}{c}3\\5\end{array}$
pop	3	5
pop	5	∅

Implementing Stack ADT: Array (1)

```java
public class ArrayedStack {
  private static final int MAX_CAPACITY = 1000;
  private String[] data;
  private int t; /* top index */
  public ArrayedStack() {
    data = new String[MAX_CAPACITY];
    t = -1; }
  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }
  public String top() {
    if (isEmpty()) { /* Error: Empty Stack. */ }
    else { return data[t]; } }
  public void push(String e) {
    if (size() == MAX_CAPACITY) { /* Error: Stack Full. */ }
    else { t ++; data[t] = e; } }
  public String pop() {
    String result;
    if (isEmpty()) { /* Error: Empty Stack */ }
    else { result = data[t]; data[t] = null; t --; }
    return result; }
}
```

Implementing Stack ADT: Array (2)

```java
@Test
public void testArrayedStack() {
  ArrayedStack s = new ArrayedStack();
  assertTrue(s.size() == 0 && s.isEmpty());
  try { String top = s.top();
        fail("Empty stack should have caused an exception."); }
  catch(IllegalArgumentException e) { }
  s.push("Alan");
  s.push("Mark");
  s.push("Tom");
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());
  String oldTop = s.pop();
  assertEquals("Tom", oldTop);
  String newTop = s.top();
  assertEquals("Mark", newTop);
  oldTop = s.pop();
  assertEquals("Mark", oldTop);
  newTop = s.top();
  assertEquals("Alan", newTop);
}
```

Implementing Stack ADT: Array (3)

Running Times of *Array*-Based *Stack* Operations?

ArrayedStack **Method**	**Running Time**
size	O(1)
isEmpty	O(1)
top	O(1)
push	O(1)
pop	O(1)

Q: What if the preset capacity turns out to be insufficient?

A: $O(n)$ time to grow the array size and copy existing contents!

Implementing Stack ADT: Singly-Linked List (1)

```java
public class LinkedStack {
  private SinglyLinkedList list; /* assumed: head, tail, size */
  ...
}
```

Question:

Stack Method	**Singly-Linked List Method**	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
top	list.first	list.last
push	list.addFirst	list.addLast
pop	list.removeFirst	list.removeLast

Which *implementation strategy* should be chosen? Either?

Implementing Stack ADT: Singly-Linked List (2)

- If the *front of list* is treated as the *top of stack*, then:
  - All stack operations remain $O(1)$.
  - *No resizing* is necessary!
- If the *back of list* is treated as the *top of stack*, then:
  - Still *no resizing* is necessary!
  - The pop operation (via removeLast) takes $O(n)$!

Application (1): Reversing an Array

```
public static void reverse(String[] a) {
  ArrayedStack buffer = new ArrayedStack();
  for (int i = 0; i < a.length; i ++) {
   buffer.push(a[i]);
  }
  for (int i = 0; i < a.length; i ++) {
   a[i] = buffer.pop();
  }
}
```

```
@Test
public void testReverseViaStack() {
  String[] names = {"Alan", "Mark", "Tom"};
  String[] reverseOfNames = {"Tom", "Mark", "Alan"};
  StackUtilities.reverse(names);
  assertArrayEquals(reverseOfNames, names);
}
```

Application (2): Matching Delimiters

- **Problem**

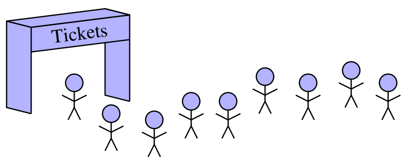
  Opening delimiters: (, [, {
  Closing delimiters:),], }
  e.g., *Correct:* "()(())([()])"
  e.g., *Incorrect:*
  - "([])" [mismatched opening and closing]
  - "{{]" [more openings than closings]
  - "{}]" [more closings than openings]
- Can we simply say $s.equals(reverseOf(s)) \Rightarrow isMatched(s)$?
  - e.g., "[()]" is matched, and its reverse are equal.
  - ***NO***! e.g., "([])[()]" matched, but different from its reverse.
- **Sketch of Solution**
  - When a new *opening* delimiter is found, *push* it to the stack.
  - When a new *closing* delimiter is found:
    - If it matches the *top* of the stack, then *pop* off the stack.
    - Otherwise, an error is found!
  - Finishing reading the input, an empty stack means a success!

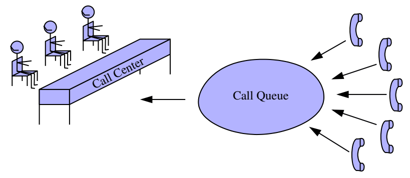
Application (2): Matching Delimiters in Java

```
public static boolean isMatched(String expression) {
  final String open = "([{";
  final String close = ")]}";
  ArrayedStack openings = new ArrayedStack();
  for (int i = 0; i < expression.length(); i ++) {
    String c = Character.toString(expression.charAt(i));
    if(open.indexOf(c) != -1) { openings.push(c); }
    else if (close.indexOf(c) != -1) {
      if( openings.isEmpty() ) { return false; /* e.g., {}) */ }
      else {
        if (open.indexOf( openings.top() ) == close.indexOf(c)) {
          openings.pop(); }
        else { return false; /* e.g., (] */ }
      }
    }
  }
  return openings.isEmpty(); /* e.g., {{ */
}
```

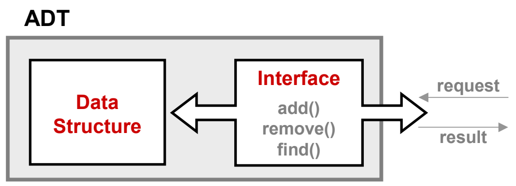
What is a Queue?

- A *queue* is a collection of objects.
- Objects in a *queue* are inserted and removed according to the *first-in, first-out (FIFO)* principle.
  - Each new element joins at the *back* of the queue.
  - *Cannot* access *arbitrary* elements of a queue
  - *Can* only access or remove the *front* of queue: *least-recently (or longest) inserted* element





The Queue ADT



- *Accessors*
  - ***first*** [compare: ***top*** of stack]
  - ***size***
  - ***isEmpty***
- *Mutators*
  - ***enqueue*** [compare: ***push*** of stack]
  - ***dequeue*** [compare: ***pop*** of stack]

Queue: Illustration

Operation	Return Value	Queue Contents
–	–	∅
isEmpty	*true*	∅
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
enqueue(1)	–	(5, 3, 1)
size	3	(5, 3, 1)
dequeue	5	(3, 1)
dequeue	3	1
dequeue	1	∅

Implementing Queue ADT: Array (1)

```
public class ArrayedQueue {
  private static final int MAX_CAPACITY = 1000;
  private String[] data;
  private int r; /* rear index */
  public ArrayedQueue() { data = new String[MAX_CAPACITY]; r = -1;}
  public int size() { return (r + 1); }
  public boolean isEmpty() { return (r == -1); }
  public String first() {
    if (isEmpty()) { /* Error: Empty Queue */ }
    else { return data[0]; } }
  public void enqueue(String e) {
    if (size() == MAX_CAPACITY) { /* Error: Queue Full. */ }
    else { r ++; data[r] = e; } }
  public String dequeue() {
    String result;
    if (isEmpty()) { /* Error: Empty Queue. */ }
    else {
      result = data[0];
      for (int i = 0; i < r; i ++) { data[i] = data[i + 1]; }
      r --; }
    return result; }
}
```

Implementing Queue ADT: Array (2)

```
@Test
public void testArrayedQueue() {
  ArrayedQueue q = new ArrayedQueue();
  assertTrue(q.size() == 0 && q.isEmpty());
  try { String first = q.first();
        fail("Empty queue should have caused an exception."); }
  catch(IllegalArgumentException e) { }
  q.enqueue("Alan");
  q.enqueue("Mark");
  q.enqueue("Tom");
  assertTrue(q.size() == 3 && !q.isEmpty());
  assertEquals("Alan", q.first());
  String oldFirst = q.dequeue();
  assertEquals("Alan", oldFirst);
  String newFirst = q.first();
  assertEquals("Mark", newFirst);
  oldFirst = q.dequeue();
  assertEquals("Mark", oldFirst);
  newFirst = q.first();
  assertEquals("Tom", newFirst);
}
```

Implementing Queue ADT: Array (3)

Running Times of *Array*-Based *Queue* Operations?

ArrayQueue **Method**	**Running Time**
size	O(1)
isEmpty	O(1)
first	O(1)
enqueue	O(1)
dequeue	$O(n)$

Q: What if the preset capacity turns out to be insufficient?

A: $O(n)$ time to grow the array size and copy existing contents!

Implementing Queue ADT: Singly-Linked List (1)

```
public class LinkedQueue {
  private SinglyLinkedList list; /* assumed: head, tail, size */
  ...
}
```

Question:

Queue Method	**Singly-Linked List Method**	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
first	list.first	list.last
enqueue	list.addLast	list.addFirst
dequeue	list.removeFirst	list.removeLast

Which *implementation strategy* should be chosen? Either?

Implementing Queue ADT: Singly-Linked List (2)

- If the *front of list* is treated as the *first of queue*, then:
  - All queue operations remain $O(1)$.
  - *No resizing* is necessary!
- If the *back of list* is treated as the *first of queue*, then:
  - Still *no resizing* is necessary!
  - The dequeue operation (via removeLast) takes $O(n)$!

Index (1)

Index (2)