Developing an ADT During the Design of a Solution

- ADT stack operations
  - Create an empty stack
  - Determine whether a stack is empty
  - Add a new item to the stack
  - Remove from the stack the item that was added most recently
  - Remove all the items from the stack
  - Retrieve from the stack the item that was added most recently

Developing an ADT During the Design of a Solution

- A queue
  - First in, first out (FIFO) property
    - The first item added is the first item to be removed

The Abstract Data Type: Developing an ADT During the Design of a Solution

- Specifications of an abstract data type for a particular problem
  - Can emerge during the design of the problem’s solution
  - Examples
    - readAndCorrect algorithm
    - displayBackward algorithm

Refining the Definition of the ADT Stack

- Pseudocode for the ADT stack operations
  - createStack() // Creates an empty stack.
  - isEmpty() // Determines whether a stack is empty.
  - push(newItem) throws StackException // Adds newItem to the top of the stack.
    // Throws StackException if the insertion is not successful.
Refining the Definition of the ADT Stack

- **Pseudocode for the ADT stack operations (Continued)**
  ```java
  pop() throws StackException
  // Retrieves and then removes the top of the stack.
  // Throws StackException if the deletion is not
  // successful.

  popAll()
  // Removes all items from the stack.

  peek() throws StackException
  // Retrieves the top of the stack. Throws
  // StackException if the retrieval is not
  // successful
  ```

Axioms (Optional)

- Axioms are used to define an ADT formally
  - Example
    - Axiom to specify that the last item inserted into
      `stack` is the first item to be removed
      ```java
      stack.push(newItem).pop() = stack
      ```

Checking for Balanced Braces

- **Requirements for balanced braces**
  - Each time you encounter a “}”, it matches an already
    encountered “{”
  - When you reach the end of the string, you have
    matched each “{”

Using the ADT Stack in a Solution

- `displayBackward` and `readAndCorrect` algorithms can be refined by using stack
  operations
- A program can use a stack independently of the
  stack’s implementation

Simple Applications of the ADT Stack: Checking for Balanced Braces

- A stack can be used to verify whether a program
  contains balanced braces
  - An example of balanced braces
    `abc{defg{ijk}{l{mn}}op}qr`
  - An example of unbalanced braces
    `abc{def}{ghij{kl}m`

Checking for Balanced Braces

Input string | Stack as algorithm executes
---|---
(a(b)c) | 1. push “a”
| | 2. push “b”
| | 3. push “c”
| | 4. pop
| | Stack empty \(\Rightarrow\) balanced

(a(b)c) | 1. push “a”
| | 2. push “b”
| | 3. pop
| | Stack empty \(\Rightarrow\) balanced

(a(b)c) | 1. push “a”
| | 2. push “b”
| | 3. push “c”
| | 4. pop
| | Stack not empty \(\Rightarrow\) not balanced

Figure 7-3
Traces of the algorithm that checks for balanced braces
Checking for Balanced Braces

- The exception StackException
  - A Java method that implements the balanced-braces algorithm should do one of the following
    - Take precautions to avoid an exception
    - Provide try and catch blocks to handle a possible exception

Recognizing Strings in a Language

- Language L
  \[ L = \{ w \in \{ \text{char} \}^* : w \text{ is a possible empty string of characters other than $$.} \quad [w = \text{reverse}(w)] \]  
  - A stack can be used to determine whether a given string is in L
    - Traverse the first half of the string, pushing each character onto a stack
    - Once you reach the $, for each character in the second half of the string, pop a character off the stack
    - Match the popped character with the current character in the string

Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A linked list
  - The ADT list
- StackInterface
  - Provides a common specification for the three implementations
- StackException
  - Used by StackInterface
  - Extends java.lang.RuntimeException

An Array-Based Implementation of the ADT Stack

- StackArrayBased class
  - Implements StackInterface
  - Instances
    - Stacks
  - Private data fields
    - An array of Objects called items
    - The index top

A Reference-Based Implementation of the ADT Stack

- A reference-based implementation
  - Required when the stack needs to grow and shrink dynamically
- StackReferenceBased
  - Implements StackInterface
  - top is a reference to the head of a linked list of items
A Reference-Based Implementation of the ADT Stack

An Implementation That Uses the ADT List

• The ADT list can be used to represent the items in a stack
• If the item in position 1 of a list represents the top of the stack
  - push(newItem) operation is implemented as add(1, newItem)
  - pop() operation is implemented as get(1)
  - peek() operation is implemented as get(1)

Comparing Implementations

• All of the three implementations are ultimately array based or reference based
• Fixed size versus dynamic size
  - An array-based implementation
    • Uses fixed-sized arrays
    - Prevents the push operation from adding an item to the stack if the stack’s size limit has been reached
  - A reference-based implementation
    • Does not put a limit on the size of the stack

The Java Collections Framework Class Stack

• JCF contains an implementation of a stack class called Stack (generic)
• Derived from Vector
• Includes methods: peek, pop, push, and search
• search returns the 1-based position of an object on the stack
Converting Infix Expressions to Equivalent Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only “to the right” with respect to the operands
  - All parentheses are removed

Evaluating Postfix Expressions

- A postfix calculator
  - Requires you to enter postfix expressions
    - Example: 2, 3, 4, +, *
  - When an operand is entered, the calculator
    - Pushes it onto a stack
  - When an operator is entered, the calculator
    - Applies it to the top two operands of the stack
    - Pops the operands from the stack
    - Pushes the result of the operation on the stack

Application: Algebraic Expressions

- When the ADT stack is used to solve a problem, the use of the ADT’s operations should not depend on its implementation
- To evaluate an infix expressions
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression

Evaluating Postfix Expressions

<table>
<thead>
<tr>
<th>Key entered</th>
<th>Calculator action</th>
<th>Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>push 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>push 3</td>
<td>2 3</td>
</tr>
<tr>
<td>4</td>
<td>push 4</td>
<td>2 3 4</td>
</tr>
<tr>
<td>+</td>
<td>operand2 = pop stack (4)</td>
<td>2 3</td>
</tr>
<tr>
<td></td>
<td>operand1 = pop stack (3)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>result = operand1 + operand2 (7)</td>
<td>2 7</td>
</tr>
<tr>
<td>*</td>
<td>operand2 = pop stack (7)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>operand1 = pop stack (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>result = operand1 * operand2 (14)</td>
<td>14</td>
</tr>
</tbody>
</table>

Figure 7.8
The action of a postfix calculator when evaluating the expression 2 * (3 + 4)

Converting Infix Expressions to Equivalent Postfix Expressions

<table>
<thead>
<tr>
<th>op</th>
<th>stack (bottom to top)</th>
<th>postfixExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>abcd</td>
<td>a</td>
</tr>
</tbody>
</table>

Figure 7.9
A trace of the algorithm that converts the infix expression a - (b + c) * d/e to postfix form
Application: A Search Problem

• High Planes Airline Company (HPAir)
  - Problem
    • For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city

A Nonrecursive Solution that Uses a Stack

• The solution performs an exhaustive search
  - Beginning at the origin city, the solution will try every possible sequence of flights until either
    • It finds a sequence that gets to the destination city
    • It determines that no such sequence exists
• The ADT stack is useful in organizing an exhaustive search
• Backtracking can be used to recover from a wrong choice of a city

Representing the Flight Data

• The flight map for HPAir is a graph
  - Adjacent vertices
    • Two vertices that are joined by an edge
  - Directed path
    • A sequence of directed edges

A Nonrecursive Solution that Uses a Stack

A Nonrecursive Solution that Uses a Stack

A Recursive Solution

• Possible outcomes of the recursive search strategy
  - You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination
  - You reach a city C from which there are no departing flights
  - You go around in circles
A Recursive Solution

- A refined recursive search strategy

```java
searchR(originCity, destinationCity)
   Mark originCity as visited
   if (originCity is destinationCity) {
      Terminate -- the destination is reached
   } else {
      for (each unvisited city C adjacent to originCity) {
         searchR(C, destinationCity)
      }
   }
```

The Relationship Between Stacks and Recursion

- The ADT stack has a hidden presence in the concept of recursion
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a nonrecursive version of a recursive algorithm

Summary

- ADT stack operations have a last-in, first-out (LIFO) behavior
- Algorithms that operate on algebraic expressions are an important application of stacks
- A stack can be used to determine whether a sequence of flights exists between two cities
- A strong relationship exists between recursion and stacks