Stacks

Chapter 6

The Abstract Data Type Stack

- Operations on a stack
  - Last-in,
  - First-out behavior.
- Applications demonstrated
  - Evaluating algebraic expressions
  - Searching for a path between two points

Developing an ADT During the Design of a Solution

- Consider typing a line of text on a keyboard
  - Use of backspace key to make corrections
  - You type `abcc←ddde←←←eg←fg`
  - Corrected input will be `abcdefg`
- Must decide how to store the input line.

Developing an ADT During the Design of a Solution

```
// Read the line, correcting mistakes along the way
while (not end of line)
{
    Read a new character ch
    if (ch is not a '←')
        Add ch to the ADT
    else
        Remove from the ADT (and discard) the item that was added most recently
}
```

- Initial draft of solution.
- Two required operations
  - Add new item to ADT
  - Remove item added most recently

Developing an ADT During the Design of a Solution

```
// Read the line, correcting mistakes along the way
while (not end of line)
{
   Read a new character ch
   if (ch is not a '←')
      Add ch to the ADT
   else if (the ADT is not empty)
      Remove from the ADT and discard the item that was added most recently
   else
      Ignore the '←'
}
```

- Read and correct algorithm.
- Third operation required
  - See whether ADT is empty

Developing an ADT During the Design of a Solution

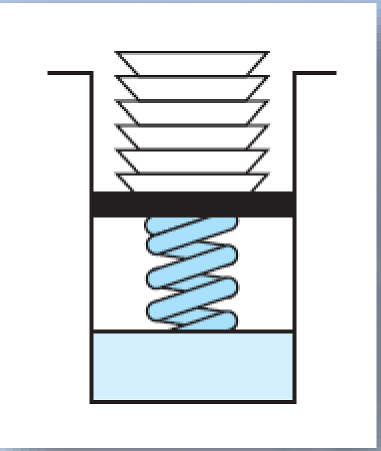
```
// Display the line in reverse order
while (the ADT is not empty)
{
    Get a copy of the item that was added to the ADT most recently and assign it to ch
    Display ch
    Remove from the ADT and discard the item that was added most recently
}
```

- Write-backward algorithm
- Fourth operation required
  - Get item that was added to ADT most recently.

Specifications for the ADT Stack

- See whether stack is empty.
- Add new item to the stack.
- Remove from and discard stack item that was added most recently.
- Get copy of item that was added to stack most recently.

Specifications for the ADT Stack



LIFO: The last item inserted onto a stack is the first item out

FIGURE 6-1 A stack of cafeteria plates

Specifications for the ADT Stack

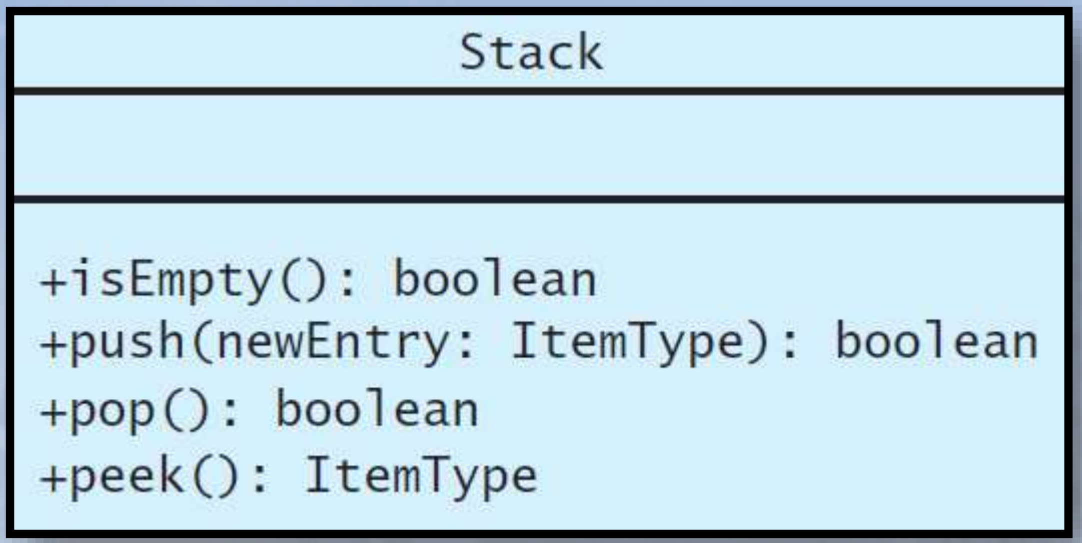


FIGURE 6-2 UML diagram for the class Stack

Specifications for the ADT Stack

```cpp
/** @file StackInterface.h */
#ifndef STACK_INTERFACE_
#define STACK_INTERFACE_

template<class ItemType>
class StackInterface
{
public:
   /** Sees whether this stack is empty.
    @return  True if the stack is empty, or false if not. */
   virtual bool isEmpty() const = 0;

   /** Adds a new entry to the top of this stack.
    @post  If the operation was successful, newEntry is at the top of the stack.
    @param newEntry  The object to be added as a new entry.
    @return  True if the addition is successful or false if not. */
   virtual bool push(const ItemType& newEntry) = 0;
```

LISTING 6-1 A C++ interface for stacks

Specifications for the ADT Stack

```
   /** Removes the top of this stack.
    @post  If the operation was successful, the top of the stack
       has been removed.
    @return  True if the removal is successful or false if not. */
   virtual bool pop() = 0;

   /** Returns a copy of the top of this stack.
    @pre  The stack is not empty.
    @post  A copy of the top of the stack has been returned, and
       the stack is unchanged.
    @return  A copy of the top of the stack. */
   virtual ItemType peek() const = 0;

   /** Destroys this stack and frees its assigned memory. */
   virtual ~StackInterface() { }
}; // end StackInterface
#endif
```

LISTING 6-1 A C++ interface for stacks

Specifications for the ADT Stack

- Axioms for multiplication

$$(a \times b) \times c = a \times (b \times c)$$
$$a \times b = b \times a$$
$$a \times 1 = a$$
$$a \times 0 = 0$$

- Axioms for ADT stack

```
(Stack()).isEmpty() = true
(Stack()).pop() = false
(Stack()).peek() = error
(aStack.push(item)).isEmpty() = false
(aStack.push(item)).peek() = item
(aStack.push(item)).pop() = true
(aStack.push(item)).pop() ⇒ aStack
```

Checking for Balanced Braces

- Example of curly braces in C++ language
  - Balanced `abc{defg{ijk}{l{mn}}op}qr`
  - Not balanced `abc{def}}{ghij{kl}m`
- Requirements for balanced braces
  - For each **}**, must match an already encountered **{**
  - At end of string, must have matched each **{**

Checking for Balanced Braces

```
for (each character in the string)
{
    if (the character is a '{')
        aStack.push('{')
    else if (the character is a '}')
        aStack.pop()
}
```

Initial draft of a solution.

Checking for Balanced Braces

```
// Checks the string aString to verify that braces match.
// Returns true if aString contains matching braces, false otherwise.
checkBraces(aString: string): boolean
{
   aStack = a new empty stack
   balancedSoFar = true
   i = 0                          // Tracks character position in string

   while (balancedSoFar and i < length of aString)
   {
      ch = character at position i in aString
      i++

      // Push an open brace
      if (ch is a '{')
         aStack.push('{')

      // Close brace
      else if (ch is a '}')
      {
```

Detailed pseudocode solution.

Checking for Balanced Braces

```
      // Close brace
      else if (ch is a '}')
      {
         if (!aStack.isEmpty())
            aStack.pop()      // Pop a matching open brace
         else                 // No matching open brace
            balancedSoFar = false
      }
      // Ignore all characters other than braces
   }

   if (balancedSoFar and aStack.isEmpty())
      aString has balanced braces
   else
      aString does not have balanced braces
}
```

Detailed pseudocode solution.

Checking for Balanced Braces

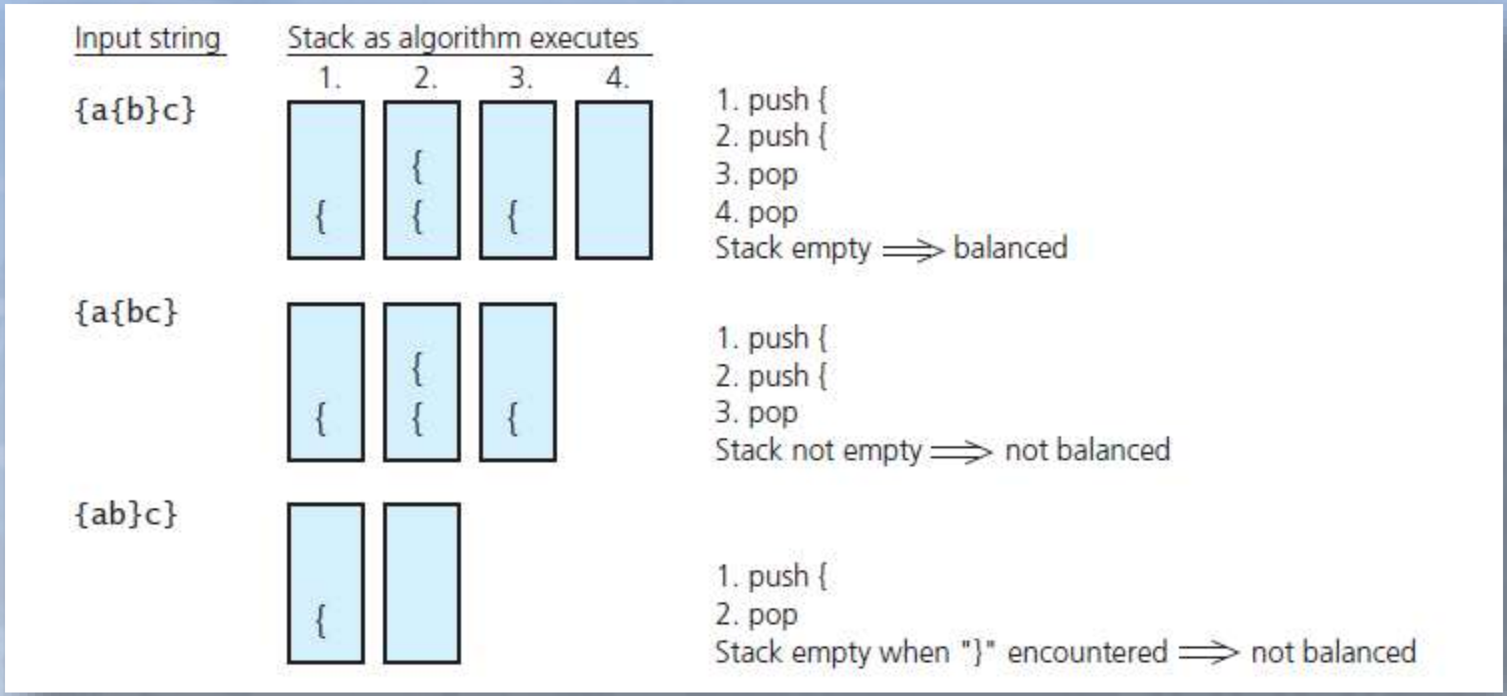


FIGURE 6-3 Traces of algorithm that checks for balanced braces

Recognizing Strings in a Language

- Given a definition of a language, *L*
  - Special palindromes
  - Special middle character $
  - Example ABC$CBA ε *L*, but AB$AB $\not\in$ *L*
- A stack is useful in determining whether a given string is in a language
  - Traverse first half of string
  - Push each character onto stack
  - Reach $, undo, pop character, match or not

Recognizing Strings in a Language

```
// Checks the string aString to verify that it is in language L.
// Returns true if aString is in L, false otherwise.
recognizeString(aString: string): boolean
{
   aStack = a new empty stack

   // Push the characters that are before the $ (that is, the characters in s) onto the stack
   i = 0                          // Tracks character position in string
   ch = character at position i in aString
   while (ch is not a '$')
   {
      aStack.push(ch)
      i++
      ch = character at position i in aString
   }

   // Skip the $
   i++

   // Match the reverse of s
   inLanguage = true              // Assume string is in language
   while (inLanguage and i < length of aString)
   {
```

Algorithm to recognize string in language *L*

Recognizing Strings in a Language

```
inLanguage = true                    // Assume string is in language
while (inLanguage and i < length of aString)
{
    if (!aStack.isEmpty())
    {
        stackTop = aStack.peek()
        aStack.pop()
        ch = character at position i in aString
        if (stackTop equals ch)
            i++                      // Characters match
        else
            inLanguage = false       // Characters do not match (top of stack is not ch)
    }
    else
        inLanguage = false           // Stack is empty (first half of string is shorter
                                     // than second half)
}
if (inLanguage and aStack.isEmpty())
    aString is in language
else
    aString is not in language
}
```

Algorithm to recognize string in language *L*

Using Stacks with Algebraic Expressions

- Strategy
  - Develop algorithm to evaluate postfix
  - Develop algorithm to transform infix to postfix
- These give us capability to evaluate infix expressions
  - This strategy easier than *directly* evaluating infix expression

Evaluating Postfix Expressions

- Infix expression   2 * (3 + 4)
- Equivalent postfix   2 3 4 + *
  - Operator in postfix applies to two operands immediately preceding
- Assumptions for our algorithm
  - Given string is correct postfix
  - No unary, no exponentiation operators
  - Operands are single lowercase letters, integers

Evaluating Postfix Expressions

Key entered	Calculator action		Stack (bottom to top):
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = peek	(4)	2 3 4
	pop		2 3
	operand1 = peek	(3)	2 3
	pop		2
	result = operand1 + operand2	(7)	
	push result		2 7
*	operand2 = peek	(7)	2 7
	pop		2
	operand1 = peek	(2)	2
	pop		
	result = operand1 * operand2	(14)	
	push result		14

FIGURE 6-4 The effect of a postfix calculator on a stack when evaluating the expression 2 * (3 + 4)

Evaluating Postfix Expressions

```
for (each character ch in the string)
{
    if (ch is an operand)
        Push the value of the operand ch onto the stack
    else // ch is an operator named op
    {
        // Evaluate and push the result
        operand2 = top of stack
        Pop the stack

        operand1 = top of stack
        Pop the stack

        result = operand1 op operand2
        Push result onto the stack
    }
}
```

A pseudocode algorithm that evaluates postfix expressions

Infix to Postfix

- Important facts
  - Operands always stay in same order with respect to one another.
  - Operator will move only “to the right” with respect to the operands;
    - If in the infix expression the operand *x* precedes the operator *op,*
    - Also true that in the postfix expression the operand *x* precedes the operator *op* .
  - All parentheses are removed.

Infix to Postfix

```
Initialize postfixExp to the empty string
for (each character ch in the infix expression)
{
    switch (ch)
    {
        case ch is an operand:
            Append ch to the end of postfixExp
            break
        case ch is an operator:
            Save ch until you know where to place it
            break
        case ch is a '(' or a ')':
            Discard ch
            break
    }
}
```

First draft of algorithm to convert infix to postfix

Infix to Postfix

- Determining where to place operators in postfix expression
  - Parentheses
  - Operator precedence
  - Left-to-right association
- Note difficulty
  - Infix expression not always fully parenthesized
  - Precedence and left-to-right association also affect results

Infix to Postfix

ch	operatorStack (top to bottom)	postfixExp	
a		a	
–	–	a	
(	(–	a	
b	(–	a b	
+	+ (–	a b	
c	+ (–	a b c	
*	* + (–	a b c	
d	* + (–	a b c d	
)	+ (–	a b c d *	Move operators from stack to
	(–	a b c d * +	postfixExp until "("
	–	a b c d * +	
/	/ –	a b c d * +	
e	/ –	a b c d * + e	
	–	a b c d * + e /	Copy operators from
		a b c d * + e / –	stack to postfixExp

Figure 6-5 A trace of the algorithm that converts the infix expression a – (b + c * d) /e to postfix form

Infix to Postfix

```
for (each character ch in the infix expression)
{
   switch (ch)
   {
      case operand:        // Append operand to end of postfix expression—step 1
         postfixExp = postfixExp • ch
         break
      case '(':            // Save '(' on stack—step 2
         operatorStack.push(ch)
         break
      case operator:       // Process stack operators of greater precedence—step 3
         while (!operatorStack.isEmpty() and operatorStack.peek() is not a '(' and
                precedence(ch) <= precedence(operatorStack.peek()))
         {
            Append operatorStack.peek() to the end of postfixExp
            operatorStack.pop()
         }
         operatorStack.push(ch) // Save the operator
         break
      case ')':                 // Pop stack until matching '('—step 4
```

Pseudocode algorithm that converts infix to postfix

Infix to Postfix

```
        break
    case ')':                      // Pop stack until matching '('—step 4
        while (operatorStack.peek() is not a '(')
        {
            Append operatorStack.peek() to the end of postfixExp
            operatorStack.pop()
        }
        operatorStack.pop()        // Remove the open parenthesis
        break
    }
}
// Append to postfixExp the operators remaining in the stack—step 5
while (!operatorStack.isEmpty())
{
    Append operatorStack.peek() to the end of postfixExp
    operatorStack.pop()
}
```

Pseudocode algorithm that converts infix to postfix