Chapter 3: Stacks

Chapter Objectives

- To learn about the stack data type and how to use its four methods: push, pop, peek, and empty
- To understand how Java implements a stack
- To learn how to implement a stack using an underlying array or a linked list
- To see how to use a stack to perform various applications, including finding palindromes, testing for balanced (properly nested) parentheses, and evaluating arithmetic expressions

Stack Abstract Data Type

- A stack can be compared to a Pez dispenser
  - Only the top item can be accessed
  - Can only extract one item at a time
- A stack is a data structure with the property that only the top element of the stack is accessible
- The stack's storage policy is Last-In, First-Out

FIGURE 5.1
A Pez Dispenser

Specification of the Stack Abstract Data Type

- Only the top element of a stack is visible, and thus, not many operations performed are possible
- Need the ability to
  - Inspect the top element
  - Retrieve the top element
  - Push a new element on the stack
  - Test for an empty stack

Specification of the Stack Abstract Data Type (continued)

- Specification of StackInt<E>

Methods	Behavior
boolean empty()	Returns true if the stack is empty; otherwise, return false.
E peek()	Returns the object at the top of the stack without removing it
E pop()	Returns the object at the top of the stack and removes it.
E push(E obj)	Pushes an item onto the top of the stack and returns the item pushed.

Two Stack Applications

- Palindrome finder
  - Palindrome: string that reads the same in either direction
  - Example: “Able was I ere I saw Elba”
- Parentheses matcher

Palindrome Checking Using an Array

- Input is a String object of say length n
- Convert the String object into its lower case form, say s
- For p=0 to n/2, test whether the p-th symbol of s is equal to the (n-1-p)-th symbol of s; if test fails for any p, declare that s is not a palindrome
- Declare that s is a palindrome

- See the sample program

Palindrome Checking Using a Stack

- Again, obtain the lower-case version of input, s
- Scan s from left to right and put the symbols into a Stack<Character> object, say charStack
  - *Recall autoboxing*
- Note that the height of charStack is equal to the length of s
- For p=0 to n/2, check whether the p-th symbol of s is equal to the symbol that's been popped from charStack

- See the sample code

Another Method Using a Stack

- Again, obtain the lower-case version of input, s
- Scan s from left to right and put the symbols into two Stack<Character> objects, say charStack1 and charStack2
- Keep popping from charStack2 and pushing the characters into an initially empty stack, charStack0; now charStack0 has the reverse of charStack1
- Keep popping from charStack0 and charStack1 concurrently to conduct check

- See the sample code

Parenthesis Matchers

- When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses
  - (a+b*(c/(d-e)))+(d/e)
- Problem is further complicated if braces or brackets are used in conjunction with parenthesis
- Solution is to use stacks!

Solution with a Stack

- Assign a unique index to each pair of parentheses
- Initialize an empty integer stack iStack
- Process the input string, s, from left to right
  - If an open parenthesis is encountered, push its index to iStack
  - If a close parenthesis is encountered,
    - If iStack is empty, declare s is not balanced
    - Pop the index, m, from iStack
    - If m is not equal to the close parenthesis's index, declare s is not balanced
- If iStack is empty, declare s is balanced; o.w., declare s is not balanced

Two Possible Implementations of Stack

- Use a single-linked list
  - Treat the head of the list as the point of push and pop for efficiency
- Use an array
  - Treat the end of the array as the point of push and pop
    - As in ArrayList, double the size if there is no room for a new element

Implementing a Stack with a List Component

- Can use any class that implements the List interface
- Name of class illustrated in the textbook is ListStack<E>
  - ListStack is an ***adapter class*** because it provides an interface by simply adapting the methods available in another class to the interface by giving different names to exiting methods in the class.
    - getLast(), addLast, remove(), size()

Comparison of Stack Implementations

- Extending a Vector (as is done by Java) is a poor choice for stack implementation as all Vector methods are accessible
- Easiest implementation would be to use an ArrayList component for storing data
- All insertions and deletions are constant time regardless of the type of implementation discussed
    - All insertions and deletions occur at one end

Postfix Arithmetic Notation

- Binary operations immediately follow the two operands.
- The notation we normally use is called ***infix** notation.*
- Advantages
  - No need to use parentheses
  - No need to consider precedence
  - Easy for a computer to evaluate expressions in the postfix notation.

Postfix	Infix	Value
4 7 +	4 + 7	11
4 7 + 3 *	(4 + 7) * 3	33
10 7 − 4 2 / *	(10 − 7) * (4 / 2)	6
5 4 3 2 * * *	5 * (4 * (3 * 2))	240

Algorithm for Evaluation Postfix Expression

1. Create an empty stack of integers
2. **while** the next token exists
3. receive the next token
4. **if** the token is an operation
5. ***pop*** the second operand from the stack
6. ***pop*** the first operand from the stack
7. apply the operation to the operands
8. ***push*** the result onto stack
9. **else** ***push*** the token onto stack
10. Return the top element of the stack

(There should be no element remaining in the stack after this.)

Conversion from InFix to PostFix

- Assume that the input does not have parentheses
  - We will perhaps look at the case with parentheses in the future
- Already the input has been processed into a series of tokens (String objects), each of which is either an operand or an operator
  - “5”, “-”, “10”, “+”, “20”, “*”, “30”, “/”, “40”
- From the series of tokens build its postfix expression
  - “5 10 - 20 30 * 40 / +”
- Assume that the tokens are given one after the other

Things to Notice

- Creation of postfix expression is the process of reordering the tokens
- The order in which the operands appear is unchanged, as in
  - “5”, “-”, “10”, “+”, “20”, “*”, “30”, “/”, “40”
  - “5 10 - 20 30 * 40 / +”
- The places where the operators appear have to be determined

Order Determination

- If either “*” or “/” appears, the operator should be placed immediately after the next token (operand)
- If either “+” or “-” appears (call it “o”, the location depends on the next operator
  - If the next one is either “+” or “-” (that is, it’s in the form of X o Y +/- …), place o immediately after Y
  - If the next one is either “*” or “/”, place o immediately after the series of “*” and “/” ends
- Needs to remember only at most two past operands