**Chapter 7: Stacks**

**Problem:** When you type test on a keyboard, you are likely to make mistakes. To correct the mistakes, you use the backspace key (represented by ) to erase the previous character.

a) What is the corrected input if the following line is typed on a keyboard?

yww←dshr←←wd←e Corrected input: [ywdswe]

b) How might we develop an ADT that could help us solve this problem? What operations would it need?

[add new item to the ADT]

[remove from ADT the item that was added most recently]

[determine whether the ADT is empty]

|  |  |
| --- | --- |
| **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 | **Refining the Definition of the ADT Stack**  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.  **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 |

**Stack Versus a Queue:**

Stack: LIFO

Queue: FIFO

|  |  |
| --- | --- |
| **Problem:** Show how you can use the ADT Stack Operations to read the following input line and return the corrected version as a stack  yww←dshr←←wd←e | **Stack operations:**  0.createStack();  1. push(y)  2. push(w)  3. push(w)  4. pop()  5. push(d)  6. push(s)  7. push(h)  8. push(r)  9. pop( )  10 pop( )  11. push(w)  12. push(d)  13. pop()  14.push(e) |

Stack as operations execute:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| y | w  y | w  w  y | w  y | d  w  y | s  d  w  y |  |  |  |  |  |  |  |  |

**Another Application**: 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 |



**Exercise:** [Self-test Exercise, p. 394:5] Trace the execution of the balanced-braces algorithm for each of the following strings.

|  |  |  |
| --- | --- | --- |
| **Input String** | **Stack** | **Stack Operations:** |
| x{{{y}z} |  |  |
| {x{y{z}}} |  |  |
| {xy{z}}} |  |  |

|  |  |
| --- | --- |
| **// pseudocode to check whether a string has balanced braces**  aStack.createStack()  balancedSoFar = true  i = 0  while (balancedSoFar and i < length of aString) {  ch = character at position i in aString  ++i  // push an open brace  if (ch is '{') {  aStack.push('{')  }  // close brace  else if (ch is '}') {  if (!aStack.isEmpty()) {  openBrace = aStack.pop()  // pop a matching open brace  }  else { // no matching open brace  balancedSoFar = false  } // end if  } // end if  // ignore all characters other than braces  } // end while | // Revision that uses try…catch blocks to  // handle the possible exception  //close brace  else if (ch is '}') {  try {  // try to pop open brace  openBrace = aStack.pop()  } // end try  catch (StackException e) {  balancedSoFar = false // no open brace  } // end catch  } // end if |

**Another Application of a Stack** **Recognizing Strings in a Language**

L = {w$w’: w is a possible empty string of characters other than $, w’ = 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

**Exercise:** Show how the language recognition algorithm (described above) works and draw the contents of the stack at each step:

a) abc$cba

b) abc$bca

|  |  |
| --- | --- |
| // PseudoCode of Language Recognition  // Algorithm  / push the characters before $, that is, the  // characters in w, onto the stack  i = 0  ch = character at position i in aString  while (ch is not '$') {  aStack.push(ch)  ++i  ch = character at position i in aString  } // end while  // skip the $  ++i | // match the reverse of w  inLanguage = true // assume string is in language  while (inLanguage and i < length of aString) {  ch = character at position i in aString  try {  stackTop = aStack.pop()  if (stackTop equals ch) {  ++i // characters match  }  else {  // top of stack is not ch (characters do not match)  inLanguage = false // reject string  } // end if  } // end try  catch (StackException e) {  // aStack.pop() failed, aStack is empty (first half of  // string is shorter than second half)  inLanguage = false  } // end catch  } // end while  if (inLanguage and aStack.isEmpty()) {  aString is in language  }  else {  aString is not in language  } // end if |