CHAPTER 3 & 4

Stacks & Queues

The Collection Framework
Stack Abstract Data Type

- A stack is one of the most commonly used data structures in computer science
- A stack can be compared to a Pez dispenser
  - Only the top item can be accessed
  - You can extract only one item at a time
- The top element in the stack is the last added to the stack (most recently)
- The stack’s storage policy is Last-In, First-Out, or LIFO

Java Collections: Stack

- The Java API includes a `Stack` class as part of the package `java.util`:

```java
Stack<String> myStringStack = new Stack<String>();
Stack<Place> myPlacesStack = new Stack<Places>();

myStringStack.push("Deepak");
myPlacesStack.push(new Place("19010", "Bryn Mawr", "PA"));

etc.
```
Specification of the Stack Abstract
Data Type

- Only the top element of a stack is visible; therefore the number of operations performed by a stack are few
- We need the ability to
  - test for an empty stack (empty)
  - inspect the top element (peek)
  - retrieve the top element (pop)
  - put a new element on the stack (push)

<table>
<thead>
<tr>
<th>Methods</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean empty()</td>
<td>Returns true if the stack is empty; otherwise, returns false.</td>
</tr>
<tr>
<td>E peek()</td>
<td>Returns the object at the top of the stack without removing it.</td>
</tr>
<tr>
<td>E pop()</td>
<td>Returns the object at the top of the stack and removes it.</td>
</tr>
<tr>
<td>E push(E obj)</td>
<td>Pushes an item onto the top of the stack and returns the item pushed.</td>
</tr>
</tbody>
</table>

A Stack of Strings

- “Rich” is the oldest element on the stack and “Jonathan” is the youngest (Figure a)
- String last = names.peek(); stores a reference to “Jonathan” in last
- String temp = names.pop(); removes “Jonathan” and stores a reference to it in temp (Figure b)
- names.push(“Philip”); pushes “Philip” onto the stack (Figure c)
Other examples of stacks

- Back button in browser
- Palindrome checker
  Go hang a salami, I'm a lasagna hog!
- Matching parentheses
- Expression evaluation
- `printStackTrace()`

Queue

- The queue, like the stack, is a widely used data structure
- A queue differs from a stack in one important way
  - A stack is LIFO list – Last-In, First-Out
  - while a queue is FIFO list, First-In, First-Out
Queue Abstract Data Type

- A queue can be visualized as a line of customers waiting for service
- The next person to be served is the one who has waited the longest
- New elements are placed at the end of the line

Print Queue

- Operating systems use queues to
  - keep track of tasks waiting for a scarce resource
  - ensure that the tasks are carried out in the order they were generated
- Print queue: printing is much slower than the process of selecting pages to print, so a queue is used
Specification for a Queue Interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean offer(E item)</td>
<td>Inserts item at the rear of the queue. Returns true if successful; returns false if the item could not be inserted.</td>
</tr>
<tr>
<td>E remove()</td>
<td>Removes the entry at the front of the queue and returns it if the queue is not empty. If the queue is empty, throws a NoSuchElementException.</td>
</tr>
<tr>
<td>E poll()</td>
<td>Removes the entry at the front of the queue and returns it; returns null if the queue is empty.</td>
</tr>
<tr>
<td>E peek()</td>
<td>Returns the entry at the front of the queue without removing it; returns null if the queue is empty.</td>
</tr>
<tr>
<td>E element()</td>
<td>Returns the entry at the front of the queue without removing it. If the queue is empty, throws a NoSuchElementException.</td>
</tr>
</tbody>
</table>

- The **Queue** interface implements the **Collection** interface (and therefore the **Iterable** interface), so a full implementation of **Queue** must implement all required methods of **Collection** (and the **Iterable** interface)

Class **LinkedList** Implements the **Queue** Interface

- The **LinkedList** class provides methods for inserting and removing elements at either end of a double-linked list, which means all **Queue** methods can be implemented easily
- The Java 5.0 **LinkedList** class implements the **Queue** interface

    Queue<String> names = new LinkedList<String>();

- creates a new **Queue** reference, **names**, that stores references to **String** objects
- The actual object referenced by **names** is of type **LinkedList<String>**, but because **names** is a type **Queue<String>** reference, you can apply only the **Queue** methods to it
The **Collection Framework**

Java Collections: Queue

- The Java API includes a `Queue` interface as part of the package `java.util`:
  ```java
  Queue<String> myStringQueue = new LinkedList<String>();
  Queue<Place> myPlacesQueue = new LinkedList<Places>();
  
  myStringQueue.offer("Deepak");
  myPlacesQueue.offer(new Place("19010", "Bryn Mawr", "PA");
  
  etc.
  ```
Examples of Queues

- Simulations of real life situations: Service Queues
- Scheduling processes in Operating Systems
- Keep track of state in systematic searches

Stacks & Queues

```
java.util.Stack<E>

boolean empty()
E peek()
E pop()
  Both raise
    EmptyStackException
E push(e)
+ all List<E> operations

java.util.Queue<E>

boolean add(e)
boolean offer(e)
E remove()
E poll()
E peek()
E element()
  - Return T/F/null
  - Raise
    NoSuchElementException
```
Finding Palindromes

- Palindrome: a string that reads identically in either direction, letter by letter (ignoring case)
  - kayak
  - "I saw I was I"
  - “Able was I ere I saw Elba”
  - "Level madam level"

- Problem: Write a program that reads a string and determines whether it is a palindrome
Finding Palindromes (cont.)

<table>
<thead>
<tr>
<th>Data Fields</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>private String inputString</td>
<td>The input string.</td>
</tr>
<tr>
<td>private Stack&lt;Character&gt; charStack</td>
<td>The stack where characters are stored.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>public PalindromeFinder(String str)</td>
<td>Initializes a new PalindromeFinder object, storing a reference to the parameter str in inputString and pushing each character onto the stack.</td>
</tr>
<tr>
<td>private void fillStack()</td>
<td>Fills the stack with the characters in inputString.</td>
</tr>
<tr>
<td>private String buildReverse()</td>
<td>Returns the string formed by popping each character from the stack and joining the characters. Empties the stack.</td>
</tr>
<tr>
<td>public boolean isPalindrome()</td>
<td>Returns true if inputString and the string built by buildReverse have the same contents, except for case. Otherwise, returns false.</td>
</tr>
</tbody>
</table>

import java.util.*;

public class PalindromeFinder {
    private String inputString;
    private Stack<Character> charStack = new Stack<Character>();

    public PalindromeFinder(String str) {
        inputString = str;
        fillStack(); // fills the stack with the characters in inputString
    }

    ...
Finding Palindromes (cont.)

- Solving using a stack:
  - Push each string character, from left to right, onto a stack

```
private void fillStack() {
    for(int i = 0; i < inputString.length(); i++) {
        charStack.push(inputString.charAt(i));
    }
}
```

Finding Palindromes (cont.)

- Solving using a stack:
  - Pop each character off the stack, appending each to the StringBuilder result

```
private String buildReverse(){
    StringBuilder result = new StringBuilder();
    while(!charStack.empty()) {
        result.append(charStack.pop());
    }
    return result.toString();
}
```
Finding Palindromes (cont.)

...  

public boolean isPalindrome() {
    return inputString.equalsIgnoreCase(buildReverse());
}

Queue Applications

Discrete Event Simulation
**Discrete Event Simulation**

- Single Queue, single server
  - Arrive
  - Depart

- Single Queue, multiple servers
  - Arrive
  - Depart

- Multiple Queue, multiple servers
  - Arrive
  - Depart

**Example: Single Queue, Single Server**

- Arrival process
  - How customers arrive: What is inter-arrival time?
    - E.g. between 1-3 min
  - Service mechanism: How long will service take?
    - E.g. 0.5 to 2.0 min
  - Queue characteristics: FIFO
Example Data

<table>
<thead>
<tr>
<th>Customer</th>
<th>Inter-arrival Time</th>
<th>Service Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>1.9</td>
<td>1.7</td>
</tr>
<tr>
<td>C2</td>
<td>1.3</td>
<td>1.8</td>
</tr>
<tr>
<td>C3</td>
<td>1.1</td>
<td>1.5</td>
</tr>
<tr>
<td>C4</td>
<td>1.0</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Queue Simulation

<table>
<thead>
<tr>
<th>T</th>
<th>Arrival</th>
<th>Queue</th>
<th>Server</th>
<th>Depart</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.9</td>
<td>C1</td>
<td></td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>C2</td>
<td>[C2]</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td></td>
<td></td>
<td>C2</td>
<td>C1</td>
</tr>
<tr>
<td>4.3</td>
<td>C3</td>
<td>[C3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>C4</td>
<td>[C4, C3]</td>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td></td>
<td>[C4]</td>
<td>C3</td>
<td>C2</td>
</tr>
<tr>
<td>6.9</td>
<td></td>
<td></td>
<td>C4</td>
<td>C2</td>
</tr>
<tr>
<td>7.8</td>
<td></td>
<td></td>
<td></td>
<td>C4</td>
</tr>
</tbody>
</table>

Application: Lab Printer Simulation

- There is one printer in the Computer Science Lab
- At any given time, there may be as many as 10 students working in the lab
- Each student may print up to twice in an hour
- Print jobs are 1-20 pages long
- There are up to 20 print jobs in an hour
- Question: What is the chance that in any given second there will be a print job scheduled?
Application: Lab Printer Simulation

- There is one printer in the Computer Science Lab
- At any given time, there may be as many as 10 students working in the lab
- Each student may print up to twice in an hour
- Print jobs are 1-20 pages long
- \( \therefore \) There are up to 20 print jobs in an hour
- Question: What is the chance that in any given second there will be a print job scheduled?

\[
\frac{20}{1\ text{\ hour}} \times \frac{1\ text{\ hour}}{60\ min} \times \frac{1\ min}{60\ sec} = \frac{20}{3600} = \frac{1\ task}{180\ sec}
\]

Application: Lab Printer Simulation

- There is one printer in the CS Lab (10 ppm)
- At any given time, there may be as many as 10 students working in the lab
- Each student may print up to twice in an hour
- Print jobs are 1-20 pages long
- \( \therefore \) There are up to 20 print jobs in an hour
- Question: What will the average wait time be for students to receive their printouts?
- Question: What would the average wait time be if the printer were upgraded to 20 ppp?
Implementing a Stack

Section 3.3

Java Collections: Stack

- The Java API includes a Stack class as part of the package java.util:

```java
Stack<String> myStringStack = new Stack<String>();
Stack<Place> myPlacesStack = new Stack<Places>();

myStringStack.push("Deepak");
myPlacesStack.push(new Place("19010", "Bryn Mawr", "PA");

etc.
```
Implementing a Stack with a List Component

- We can write a class, ListStack, that has a List component (in the example below, theData)
- We can use either the ArrayList, or the LinkedList classes, as all implement the List interface. The push method, for example, can be coded as

```
public E push(E obj) {
    theData.add(obj);
    return obj;
}
```

- A class which adapts methods of another class by giving different names to essentially the same methods (push instead of add) is called an adapter class
- Writing methods in this way is called method delegation

Implementing a Stack Using an Array

- If we implement a stack as an array, we would need . . .

```
public class ArrayStack<E> implements StackInt<E> {
    private E[] theData;
    int topOfStack = -1;
    private static final int INITIAL_CAPACITY = 10;

    public ArrayStack() {  
        theData = (E[])new Object[INITIAL_CAPACITY];
    } // ArrayStack()

    Allocate storage for an array with a default

    Keep track of the top of the stack (subscript of the element at the top of the stack; for empty stack = -1)

    There is no size variable or method
```

10/26/2017
public E push(E obj) {
    if (topOfStack == theData.length - 1)
        reallocate();
    topOfStack++;
    theData[topOfStack] = obj;
    return obj;
}

public E pop() {
    if (empty()) {
        throw new EmptyStackException();
    }
    return theData[topOfStack--];
} // pop()
Implementing a Stack using an array

```java
import java.util.EmptyStackException;
public class ArrayStack<E> implements StackInt<E> {

    E[] theData;
    int topOfStack = -1; // Initially empty stack.
    private static final int INITIAL_CAPACITY = 10;

    public ArrayStack() {
        theData = (E[]) new Object[INITIAL_CAPACITY];
    } // ArrayStack()

    public E push(E obj) {
        if (topOfStack == theData.length - 1) {
            reallocate();
        }
        topOfStack++;
        theData[topOfStack] = obj;
        return obj;
    } // push()

    public E pop() {
        if (empty()) {
            throw new EmptyStackException();
        }
        return theData[topOfStack--];
    } // pop()

} // class ArrayStack<E>
```

Implementing a Stack as a Linked Data Structure

- We can also implement a stack using a linked list of nodes

```
LinkedStack
  topOfStackRef = null

Node
  data = character
  next = node

Character
  value = 'a'

Character
  value = 'v'

Character
  value = 'a'

Character
  value = 'c'

when the list is empty, pop returns null
```
Implementing a Stack as a Linked Data Structure (cont.)

- Listing 3.5 (LinkedStack.java, pages 168 - 169)

Comparison of Stack Implementations

- The easiest implementation uses a List component (ArrayList is the simplest) for storing data
  - An underlying array requires reallocation of space when the array becomes full, and
  - An underlying linked data structure requires allocating storage for links
  - As all insertions and deletions occur at one end, they are constant time, O(1), regardless of the type of implementation used
Additional Stack Applications

Section 3.4

Postfix and infix notation

- Expressions normally are written in infix form, but it easier to evaluate an expression in postfix form since there is no need to group sub-expressions in parentheses or worry about operator precedence

<table>
<thead>
<tr>
<th>Postfix Expression</th>
<th>Infix Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 7 *</td>
<td>4 * 7</td>
<td>28</td>
</tr>
<tr>
<td>4 7 2 + *</td>
<td>4 * (7 + 2)</td>
<td>36</td>
</tr>
<tr>
<td>4 7 * 20 -</td>
<td>(4 * 7) - 20</td>
<td>8</td>
</tr>
<tr>
<td>3 4 7 * 2 / +</td>
<td>3 + ((4 * 7) / 2)</td>
<td>17</td>
</tr>
</tbody>
</table>
Evaluating Postfix Expressions

- Write a class that evaluates a postfix expression
- Use the space character as a delimiter between tokens

### Data Field | Attribute
---|---
Stack<Integer> operandStack | The stack of operands (Integer objects).

### Method | Behavior
---|---
public int eval(String expression) | Returns the value of expression.
private int evalOp(char op) | Pops two operands and applies operator op to its operands, returning the result.
private boolean isOperator(char ch) | Returns true if ch is an operator symbol.

Evaluating Postfix Expressions (cont.)

1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
Evaluating Postfix Expressions (cont.)

1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result

4 * 7

1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
Evaluating Postfix Expressions (cont.)

1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
Evaluating Postfix Expressions (cont.)

1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result

28 - 20

\[
\begin{array}{cccc}
4 & 7 & \times & 20 & - \\
\end{array}
\]

8

\[
\begin{array}{cccc}
4 & 7 & \times & 20 & - \\
\end{array}
\]
Evaluating Postfix Expressions (cont.)

1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result

Listing 3.6 (PostfixEvaluator.java, pages 173 - 175)
Converting from Infix to Postfix

- Convert infix expressions to postfix expressions
- Assume:
  - expressions consists of only spaces, operands, and operators
  - space is a delimiter character
  - all operands that are identifiers begin with a letter or underscore
  - all operands that are numbers begin with a digit

### Table

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>private Stack&lt;Character&gt; operatorStack</td>
<td>Stack of operators.</td>
</tr>
<tr>
<td>private StringBuilder postfix</td>
<td>The postfix string being formed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>public String convert(String infix)</td>
<td>Extracts and processes each token in infix and returns the equivalent postfix string.</td>
</tr>
<tr>
<td>private void processOperator(char op)</td>
<td>Processes operator op by updating operatorStack.</td>
</tr>
<tr>
<td>private int precedence(char op)</td>
<td>Returns the precedence of operator op.</td>
</tr>
<tr>
<td>private boolean isOperator(char ch)</td>
<td>Returns true if ch is an operator symbol.</td>
</tr>
</tbody>
</table>

Converting from Infix to Postfix
(cont.)

- Example: convert
  \[ w - 5.1 / \text{sum} \times 2 \]
  to its postfix form
  \[ w 5.1 \text{sum} / 2 \times - \]
Converting from Infix to Postfix (cont.)

<table>
<thead>
<tr>
<th>Next Token</th>
<th>Action</th>
<th>Effect on operatorStack</th>
<th>Effect on postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Append w to postfix.</td>
<td></td>
<td>w</td>
</tr>
<tr>
<td>-</td>
<td>The stack is empty</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>5.1</td>
<td>Append 5.1 to postfix</td>
<td>w 5.1</td>
<td>w 5.1</td>
</tr>
<tr>
<td>/</td>
<td>precedence(/) &gt; precedence(-),</td>
<td>w 5.1</td>
<td>w 5.1</td>
</tr>
<tr>
<td></td>
<td>Push / onto the stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>Append sum to postfix</td>
<td>w 5.1 sum</td>
<td>w 5.1 sum /</td>
</tr>
<tr>
<td>*</td>
<td>precedence(*) equals precedence(//)</td>
<td>w 5.1 sum /</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pop / off of stack and append to postfix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Converting from Infix to Postfix (cont.)

<table>
<thead>
<tr>
<th>Next Token</th>
<th>Action</th>
<th>Effect on operatorStack</th>
<th>Effect on postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>precedence(*) &gt;</td>
<td>w 5.1 sum /</td>
<td></td>
</tr>
<tr>
<td></td>
<td>precedence(-),</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Push * onto the stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Append 2 to postfix</td>
<td>w 5.1 sum /</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End of input</td>
<td>Stack is not empty, Pop * off of stack and append to postfix</td>
<td>w 5.1 sum / 2 *</td>
<td></td>
</tr>
<tr>
<td>End of input</td>
<td>Stack is not empty, Pop - off of stack and append to postfix</td>
<td>w 5.1 sum / 2 * -</td>
<td></td>
</tr>
</tbody>
</table>
Converting from Infix to Postfix (cont.)

Algorithm for Method convert

1. Initialize postfix to an empty StringBuilder.
2. Initialize the operator stack to an empty stack.
3. while there are more tokens in the infix string
4.     Get the next token.
5.     if the next token is an operand
6.         Append it to postfix.
7.     else if the next token is an operator
8.         Call processOperator to process the operator.
9.     else
10.        Indicate a syntax error.
11. Pop remaining operators off the operator stack and append them to postfix.

Converting from Infix to Postfix (cont.)

Algorithm for Method processOperator

1. if the operator stack is empty
2.     Push the current operator onto the stack.
3. else
4.     Peek the operator stack and let topOp be the top operator.
5.     if the precedence of the current operator is greater than the precedence of topOp
6.         Push the current operator onto the stack.
7.     else
8.         while the stack is not empty and the precedence of the current operator is less than or equal to the precedence of topOp
9.             Pop topOp off the stack and append it to postfix.
10.            if the operator stack is not empty
11.             Peek the operator stack and let topOp be the top operator.
12.            Push the current operator onto the stack.
Converting from Infix to Postfix
(cont.)

- Listing 3.7 (InfixToPostfix.java, pages 181 - 183)

Converting from Infix to Postfix
(cont.)

- Testing
  - Use enough test expressions to satisfy yourself that the conversions are correct for properly formed input expressions
  - Use a driver to catch InfixToPostfix.SyntaxErrorException

- Listing 3.8 (TestInfixToPostfix.java, page 184)
Converting Expressions with Parentheses

- The ability to convert expressions with parentheses is an important (and necessary) addition

- Modify `processOperator` to push each opening parenthesis onto the stack as soon as it is scanned

- When a closing parenthesis is encountered, pop off operators until the opening parenthesis is encountered

- Listing 3.9 (*InfixToPostfixParens.java, pages 186 - 188*)