

Problem Solving using Stack and Queue (additional)

WIA1002/WIB1002 :
Data Structure

HTML Tag Matching

- ❑ For fully-correct HTML, each opening tag should pair with a matching closing tag. For examples, `<body>` pairs with `</body>`, `<center>` pairs with `</center>`, etc. In general, each `<name>` should pair with a matching `</name>`
- ❑ An HTML document (a) and its rendering (b) are shown below.

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

HTML Tag Matching

- Ideally, an HTML document should have matching tags, although most browsers tolerate a certain number of mismatching tags.
- The checking of whether an html file has matching tags can be done using **stack**.
- Assuming that the entire HTML document is read into a string, a method isHTMLMatched can be written to do the checking:
 - make a left-to-right pass through the raw string, using index j to track progress.
 - indexOf method of the String class, which optionally accepts a starting index as a second parameter, locates the '<' and '>' characters that define the tags.
 - Substring method of the String class, returns the substring starting at a given index and optionally ending right before another given index.
 - Opening tags are pushed onto the stack, and matched against closing tags as they are popped from the stack

HTML Tag Matching

```
public static boolean isHTMLMatched(String html) {  
    Stack<String> buffer = new Stack<>( );  
    int j = html.indexOf('<'); // find first '<' character (if any)  
    while (j != -1) {  
        int k = html.indexOf('>', j+1); // find next '>' character  
        if (k == -1)  
            return false; // invalid tag  
        String tag = html.substring(j+1, k); // strip away < >  
        if (!tag.startsWith("/")) // this is an opening tag  
            buffer.push(tag);  
        else { // this is a closing tag  
            if (buffer.isEmpty( ))  
                return false; // no tag to match  
            if (!tag.substring(1).equals(buffer.pop( )))  
                return false; // mismatched tag  
        }  
        j = html.indexOf('<', k+1); // find next '<' character (if any)  
    }  
    return buffer.isEmpty( ); // were all opening tags matched?  
}
```

HTML Tag Matching

- Question 1: Write a Java program that read an HTML file into a string that is passed into the isHTMLMatched method to check whether the file has matching tags. Test your program using another HTML file which has one of the closing tags missing.

Simulating a Waiting Line

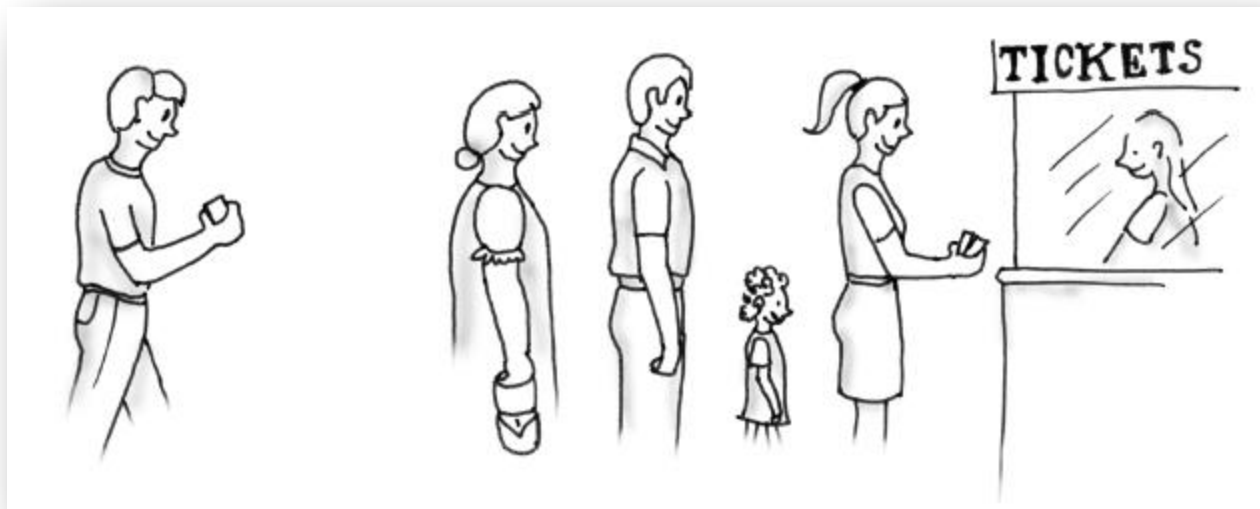


FIGURE 10-3 A line, or queue, of people

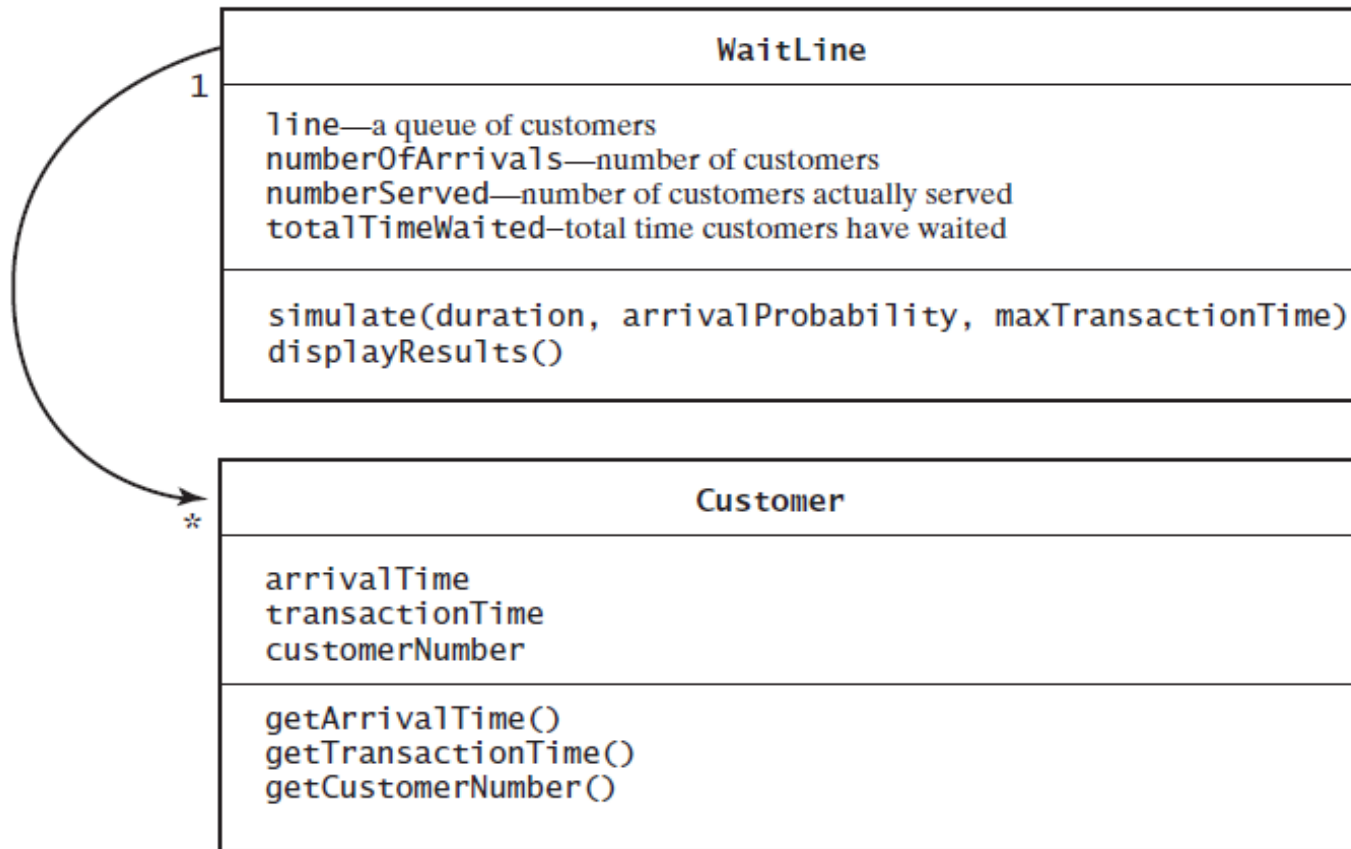
Simulating a Waiting Line

<i>WaitLine</i>
<i>Responsibilities</i>
<i>Simulate customers entering and leaving a waiting line</i>
<i>Display number served, total wait time, average wait time, and number left in line</i>
<i>Collaborations</i>
<i>Customer</i>

FIGURE 10-4 A CRC card for the class **WaitLine**

Simulating a Waiting Line

FIGURE 10-5 A diagram of the classes WaitLine and Customer



Simulating a Waiting Line

```
Algorithm simulate(duration, arrivalProbability, maxTransactionTime)
transactionTimeLeft = 0
for (clock = 0; clock < duration; clock++)
{
    if (a new customer arrives)
    {
        numberOfArrivals++
        transactionTime = a random time that does not exceed maxTransactionTime
        nextArrival = a new customer containing clock, transactionTime, and
                     a customer number that is numberOfArrivals
        line.enqueue(nextArrival)
    }
    if (transactionTimeLeft > 0) // If present customer is still being served
        transactionTimeLeft--
    else if (!line.isEmpty())
    {
        nextCustomer = line.dequeue()
        transactionTimeLeft = nextCustomer.getTransactionTime() - 1
        timeWaited = clock - nextCustomer.getArrivalTime()
        totalTimeWaited = totalTimeWaited + timeWaited
        numberServed++
    }
}
```

Algorithm for **simulate**

Simulating a Waiting Line

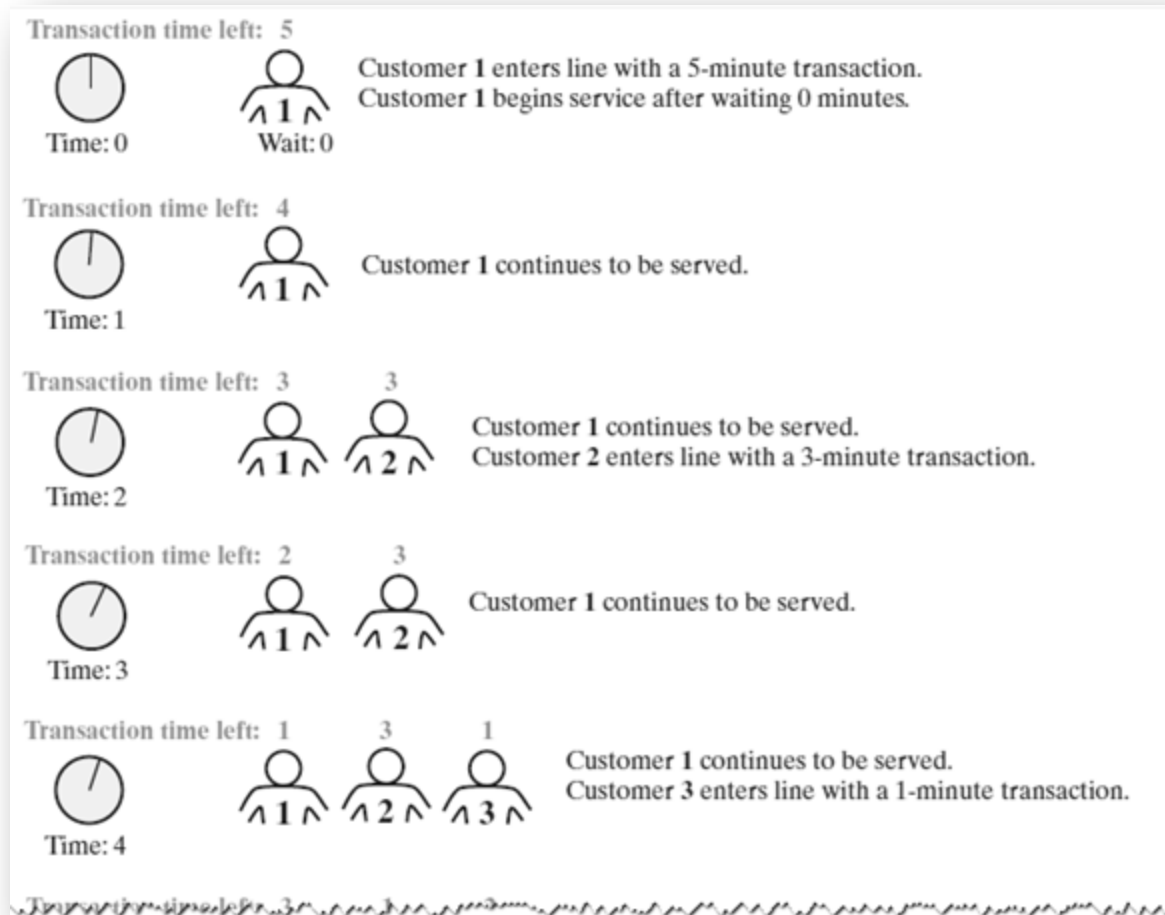


FIGURE 10-6 A simulated waiting line

Simulating a Waiting Line

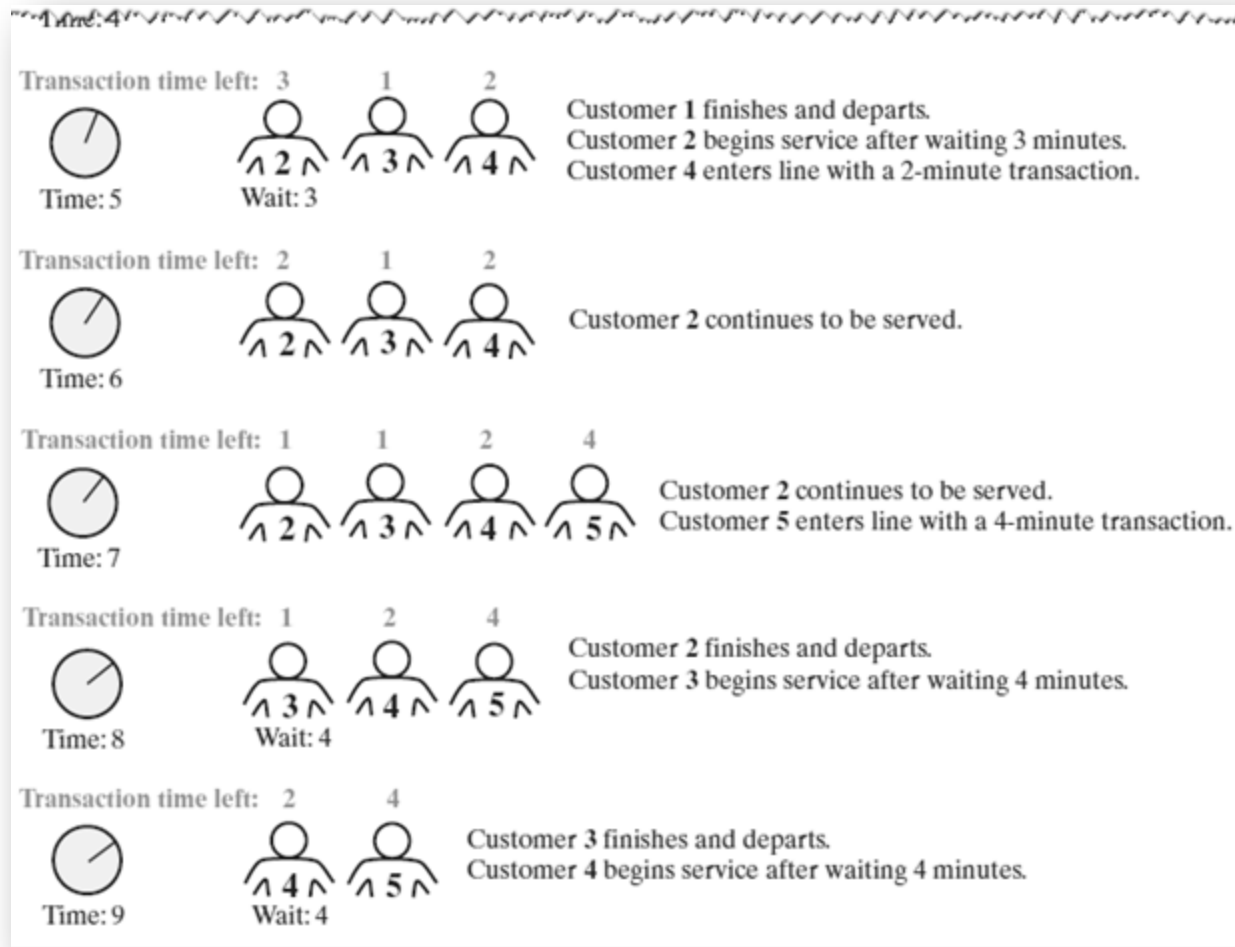


FIGURE 10-6 A simulated waiting line

Simulating a Waiting Line

```
1  /** Simulates a waiting line. */
2  public class WaitLine
3  {
4      private QueueInterface<Customer> line;
5      private int numberOfArrivals;
6      private int numberServed;
7      private int totalTimeWaited;
8
9      public WaitLine()
10     {
11         line = new LinkedQueue<>();
12         reset();
13     } // end default constructor
14
15     /** Simulates a waiting line with one serving agent.
16         @param duration The number of simulated minutes.
17         @param arrivalProbability A real number between 0 and 1, and the
18             probability that a customer arrives at
19             a given time.
20         @param maxTransactionTime The longest transaction time for a
21             customer. */
22     public void simulate(int duration, double arrivalProbability,
23                          int maxTransactionTime)
```

LISTING 10-2 The class **WaitLine**

Simulating a Waiting Line

```
20 // ...
21 // ...
22 // ...
23 // ...
24 // ...
25 // ...
26 // ...
27 // ...
28 // ...
29 // ...
30 // ...
31 // ...
32 // ...
33 // ...
34 // ...
35 // ...
36 // ...
37 // ...
38 // ...
39 // ...
40 // ...
41 // ...
42 // ...
43 // ...

    customer. */
    public void simulate(int duration, double arrivalProbability,
        int maxTransactionTime)
    {
        int transactionTimeLeft = 0;

        for (int clock = 0; clock < duration; clock++)
        {
            if (Math.random() < arrivalProbability)
            {
                numberOfArrivals++;
                int transactionTime = (int)(Math.random()
                    * maxTransactionTime + 1);
                Customer nextArrival = new Customer(clock, transactionTime,
                    numberOfArrivals);
                line.enqueue(nextArrival);
                System.out.println("Customer " + numberOfArrivals
                    + " enters line at time " + clock
                    + ". Transaction time is "
                    + transactionTime);
            } // end if

            if (transactionTimeLeft > 0)
```

LISTING 10-2 The class **WaitLine**

Simulating a Waiting Line

```
34 Customer nextArrival = new Customer(clock, transactionTime,
35                                     numberOfArrivals);
36 line.enqueue(nextArrival);
37 System.out.println("Customer " + numberOfArrivals
38                   + " enters line at time " + clock
39                   + ". Transaction time is "
40                   + transactionTime);
41 } // end if
42
43 if (transactionTimeLeft > 0)
44     transactionTimeLeft--;
45 else if (!line.isEmpty())
46 {
47     Customer nextCustomer = line.dequeue();
48     transactionTimeLeft = nextCustomer.getTransactionTime() - 1;
49     int timeWaited = clock - nextCustomer.getArrivalTime();
50     totalTimeWaited = totalTimeWaited + timeWaited;
51     numberServed++;
```

LISTING 10-2 The class **WaitLine**

Simulating a Waiting Line

```
52         System.out.println("Customer " + nextCustomer.getCustomerNumber()
53                             + " begins service at time " + clock
54                             + ". Time waited is " + timeWaited);
55     } // end if
56 } // end for
57 } // end simulate
58
59 /** Displays summary results of the simulation. */
60 public void displayResults()
61 {
62     System.out.println();
63     System.out.println("Number served = " + numberServed);
64     System.out.println("Total time waited = " + totalTimeWaited);
65     double averageTimeWaited = ((double)totalTimeWaited) / numberServed;
66     System.out.println("Average time waited = " + averageTimeWaited);
67     int leftInLine = numberOfArrivals - numberServed;
68     System.out.println("Number left in line = " + leftInLine);
69 } // end displayResults
```

LISTING 10-2 The class **WaitLine**

Simulating a Waiting Line

```
64     System.out.println("Total time waited = " + totalTimeWaited);
65     double averageTimeWaited = ((double)totalTimeWaited) / numberServed;
66     System.out.println("Average time waited = " + averageTimeWaited);
67     int leftInLine = numberOfArrivals - numberServed;
68     System.out.println("Number left in line = " + leftInLine);
69 } // end displayResults
70
71 /** Initializes the simulation. */
72 public final void reset()
73 {
74     line.clear();
75     numberOfArrivals = 0;
76     numberServed = 0;
77     totalTimeWaited = 0;
78 } // end reset
79 } // end WaitLine
```

LISTING 10-2 The class **WaitLine**

Sample output

The Java statements

```
WaitLine customerLine = new WaitLine();  
customerLine.simulate(20, 0.5, 5);  
customerLine.displayResults();
```

simulate the line for 20 minutes with a 50 percent arrival probability and a 5-minute maximum transaction time. They produce the following results (next slide). Since this example uses random numbers, another execution of the Java statements likely will have different results.

Sample output

Customer 1 enters line at time 0. Transaction time is 4
Customer 1 begins service at time 0. Time waited is 0
Customer 2 enters line at time 2. Transaction time is 2

Customer 3 enters line at time 4. Transaction time is 1
Customer 2 begins service at time 4. Time waited is 2
Customer 4 enters line at time 6. Transaction time is 4
Customer 3 begins service at time 6. Time waited is 2
Customer 4 begins service at time 7. Time waited is 1
Customer 5 enters line at time 9. Transaction time is 1
Customer 6 enters line at time 10. Transaction time is 3
Customer 5 begins service at time 11. Time waited is 2
Customer 7 enters line at time 12. Transaction time is 4
Customer 6 begins service at time 12. Time waited is 2
Customer 8 enters line at time 15. Transaction time is 3
Customer 7 begins service at time 15. Time waited is 3
Customer 9 enters line at time 16. Transaction time is 3
Customer 10 enters line at time 19. Transaction time is 5
Customer 8 begins service at time 19. Time waited is 4

Number served = 8
Total time waited = 16
Average time waited = 2.0
Number left in line = 2

Evaluation of expressions using stack

Watch the following video:

- Infix, Prefix and Postfix
(<https://youtu.be/jos1Flt21is>)
- Evaluation of Prefix and Postfix expressions using stack (https://youtu.be/MeRb_1bddWg)
- Infix to Postfix
(<https://www.youtube.com/watch?v=vq-nUF0G4fl>)

Evaluation of expressions using stack

Question 2: Write a program to convert postfix expression to infix expression. Compare it with the code for postfix evaluation from lecture Week 9.

Reference

- HTML Tag Matching - Chapter 6, Michael T. Goodrich and Roberto Tamassia. 2014. Data Structures and Algorithms in Java. 6th Edition. Wiley.
- Simulating a waiting line - Chapter 10, Frank M. Carrano and Timothy Henry. 2015. Data Structures and Abstractions with Java, 4th Edition. Pearson.
- Internet resources