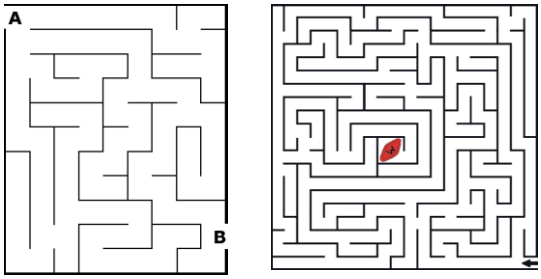
Chapter 4

Stacks and Queues

Intro: How to exit a maze

To proceed by trial and error ➔Stack: a way to exit amaze



Objectives

Discuss the following topics:

- Stacks
- Queues
- Priority Queues
- Case Study: Exiting a Maze

➔They are restricted list:

- Restricting their number of elements and/or
- Restricting operations on them

Stacks

- A **stack** is a linear data structure that can be accessed only at one of its ends for storing and retrieving data
- A stack is called an **LIFO** structure: last in/first out

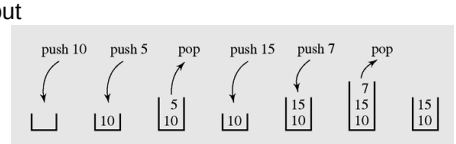
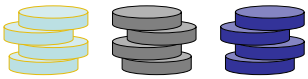


Figure 4-1 A series of operations executed on a stack

Stacks (continued)

- The following operations are needed to properly manage a stack:
  - *clear()* — Clear the stack
  - *isEmpty()* — Check to see if the stack is empty
  - *push(el)* — Put the element *el* on the top of the stack
  - *pop()* — Take the topmost element from the stack
  - *topEl()* — Return the topmost element in the stack without removing it

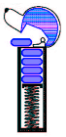
Stacks Overview



Applications of Stacks

- Direct applications
  - Delimiter matching
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

The Stack ADT (§4.2)



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```
public interface Stack {
  public int size();
  public boolean isEmpty();
  public Object top()
      throws EmptyStackException;
  public void push(Object o);
  public Object pop()
      throws EmptyStackException;
}
```


Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm *size*()
  return $t + 1$

Algorithm *pop*()
  if *isEmpty*() **then**
    throw *EmptyStackException*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$

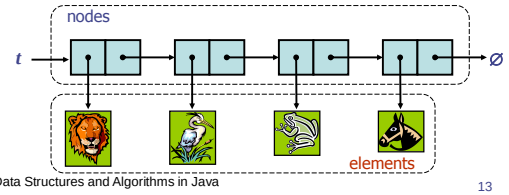
Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    throw *FullStackException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

Stack using a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



LAB 1

- Convert radix system
  - Write class to convert the number in radix 10 to any radix system

LAB 2

- Viết chương trình nhập chuỗi và cho phép undo các kí tự được xóa

LAB 3

- Matching delimiter

Stacks (continued)

Matching delimiters- A way to use stacks

```
a = b + (c – d ) * (e – f);
g[10] = h[i[9]] + (j + k) * l;
while (m < (n[8] + o)) { p = 7; r = 6; }
```

Matched

```
a = b + (c – d) * (e – f));
g[10] = h[i[9]] + j + k) * l;
while (m < (n[8] + o]) { p = 7; r = 6; }
```

Mismatched

```
while (m < (n[8] + o))
```

Nested matched

Algorithm for delimiter matching : refer to the page 142.

Stacks (continued)

Push to the stack an open delimiter and pop it from the stack when appropriate close delemiter is detected in input string

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));

Figure 4-2 Processing the statement `s=t[5]+u/(v*(w+y));` with the algorithm `delimiterMatching()`

Stacks (continued)

[(	(	v * (w + y));
[(	v	* (w + y));
[(	*	(w + y));
[[(	(	w + y));
[[(	w	+y));
[[(	+	y));
[[(	y	));
[(	)	);
empty	)	;
empty	;	

Figure 4-2 Processing the statement `s=t[5]+u/(v*(w+y));` with the algorithm `delimiterMatching()`

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([[ ]])}
  - correct: ((( )(( )){([[ ]])}
  - incorrect:)(( )){([[ ]])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

Algorithm ParenMatch(X,n):
Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in X match
Let S be an empty stack
for i=0 to n-1 **do**
    if $X[i]$ is an opening grouping symbol **then**
        S.push($X[i]$)
    else if $X[i]$ is a closing grouping symbol **then**
        if S.isEmpty() **then**
            return false *{nothing to match with}*
        if S.pop() does not match the type of $X[i]$ **then**
            return false *{wrong type}*
if S.isEmpty() **then**
    return true *{every symbol matched}*
else
    return false *{some symbols were never matched}*

LAB 4

- Calculating the very big number

Stacks (continued)

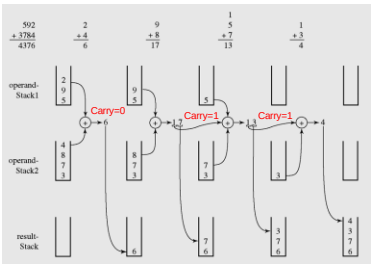


Figure 4-3 An example of adding string numbers 592 and 3,784 using stacks

Project

Project #1. Input.txt
2*(4+6)-{[4*(1+9)-(9-9)] + [2*(7-9)]}

→Output.txt: -24

Project #2
- Viết chương trình cho phép Undo và Redo

Project Maze

How to Implementation a Stack

- Select a linear storage → array/ linked list
  - Inplement all basic operations:
    clear() , isEmpty() , push(el) , pop() , topEl()
- The following slides will depict stacks using an ArrayList and a linked list as a pool for storing its elements.

Stacks (continued)

```
public class Stack {
    private java.util.ArrayList pool = new java.util.ArrayList();
    public Stack() {
    }
    public Stack(int n) {
        pool.ensureCapacity(n);
    }
    public void clear() {
        pool.clear();
    }
    public boolean isEmpty() {
        return pool.isEmpty();
    }
    public Object topEl() {
        if (isEmpty())
            throw new java.util.EmptyStackException();
        return pool.lastElement();
```

An array is used to store elements
→The active end of the stack is the end of the array
→ The end of the array is the top of the stack

Figure 4-4 Array list implementation of a stack

Stacks (continued)

```
    }
    public Object pop() {
        if (isEmpty())
            throw new java.util.EmptyStackException();
        return pool.remove(pool.size()-1);
    }
    public void push(Object el) {
        pool.add(el);
    }
    public String toString() {
        return pool.toString();
    }
}
```

pool.size()-1 is the index of the top element

Figure 4-4 Array list implementation of a stack (continued)

Stacks (continued)

```
public class LLStack {
    private java.util.LinkedList list = new java.util.LinkedList();
    public LLStack() {
    }
    public void clear() {
        list.clear();
    }
    public boolean isEmpty() {
        return list.isEmpty();
    }
    public Object topEl() {
        if (isEmpty())
            throw new java.util.EmptyStackException();
        return list.getLast();
```

We select the end of the stack is the active end of the stack.

Figure 4-5 Implementing a stack as a linked list

Stacks (continued)

```
    }
    public Object pop() {
        if (isEmpty())
            throw new java.util.EmptyStackException();
        return list.removeLast();
    }
    public void push(Object el) {
        list.addLast(el);
    }
    public String toString() {
        return list.toString();
    }
}
```

Figure 4-5 Implementing a stack as a linked list (continued)

Stacks (continued)

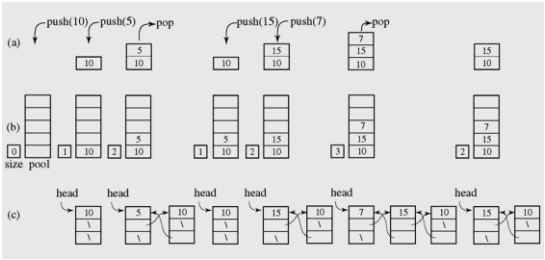


Figure 4-6 A series of operations executed on an abstract stack (a) and the stack implemented with an array (b) and with a linked list (c)

Stacks in `java.util`

Method	Operation
`boolean empty()`	Return `true` if the stack includes no element and `false` otherwise.
`Object peek()`	Return the top element on the stack; throw `EmptyStackException` for empty stack.
`Object pop()`	Remove the top element of the stack and return it; throw `EmptyStackException` for empty stack.
`Object push(Object el)`	Insert `el` at the top of the stack and return it.
`int search(Object el)`	Return the position of `el` on the stack (the first position is at the top; –1 in case of failure).
`Stack()`	Create an empty stack.

Figure 4-7 A list of methods in `java.util.Stack`; all methods from `Vector` are inherited

Queues

- A **queue** is a waiting line that grows by adding elements to its end and shrinks by taking elements from its front
- A queue is a structure in which both ends are used:
  - One for adding new elements
  - One for removing them
- A queue is an **FIFO** structure: first in/first out

Queues (continued)

- The following operations are needed to properly manage a queue:
  - *clear()* — Clear the queue
  - *isEmpty()* — Check to see if the queue is empty
  - *enqueue(el)* — Put the element *el* at the end of the queue
  - *dequeue()* — Take the first element from the queue
  - *firstEl()* — Return the first element in the queue without removing it

Queues (continued)

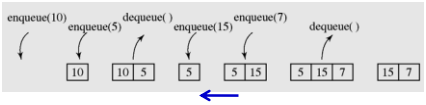


Figure 4-8 A series of operations executed on a queue

How to Implementation a Queue

- Select a linear storage → array/ linked list
  - If an array is used: To avoid shift elements when an element is pick out, use circular mechanism for pushing in and picking out an element.
  - If a linked list is used:
    - Enqueue: Add to the last
    - Dequeue: Remove first
- Inplement all basic operations:

clear() , isEmpty() , enqueue(el) , dequeue() , firstEl()

- The follwing slides will depict queues using a circular array and a linked list as a pool for storing its elements.

Queues (continued)

If an array is used to store elements of a queue the 2 indexes must be used to mark the beginning and the last positions.

If we do not want to shift elements when an element is pick out the queue then a **circular mechanism** is used.
→ What configuration will describe the queue is full?

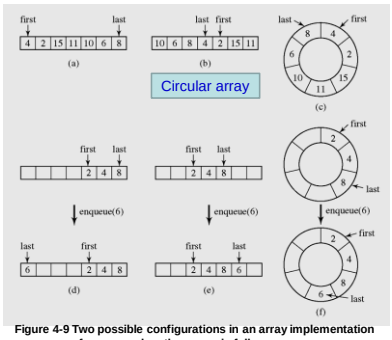


Figure 4-9 Two possible configurations in an array implementation of a queue when the queue is full

Queues (continued)

```
public class ArrayQueue {
    private int first, last, size;
    private Object[] storage;
    public ArrayQueue() {
        this(100);
    }
    public ArrayQueue(int n) {
        size = n;
        storage = new Object[size];
        first = last = -1;
    }
    public boolean isFull()  {
        return first == 0 && last == size-1 || first == last + 1;
    }
    public boolean isEmpty() {
        return first == -1;
```

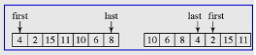


Figure 4-10 Array implementation of a queue

Queues (continued)

```
    }
    public void enqueue(Object el) {
        if (last == size-1 || last == -1) {
            storage[0] = el;
            last = 0;
            if (first == -1)
                first = 0;
        }
        else storage[++last] = el;
    }
```

The queue is full or empty → Add to the position 0, circular adding

Figure 4-10 Array implementation of a queue (continued)

Queues (continued)

```
    public Object dequeue() {
        Object tmp = storage[first];
        if (first == last)
            last = first = -1;
        else if (first == size-1)
            first = 0;
        else first++;
        return tmp;
    }
    public void printAll() {
        for (int i = 0; i < size; i++)
            System.out.print(storage[i] + " ");
    }
}
```

In case of only one element

In case of first in the end of array → circular increaseing

Normal case

Figure 4-10 Array implementation of a queue (continued)

Queues (continued)

```
public class Queue {
    private java.util.LinkedList list = new java.util.LinkedList();
    public Queue() {
    }
    public void clear() {
        list.clear();
    }
    public boolean isEmpty() {
        return list.isEmpty();
    }
    public Object firstEl() {
        return list.getFirst();
```

Figure 4-11 Linked list implementation of a queue

Queues (continued)

```
    }
    public Object dequeue() {
        return list.removeFirst();
    }
    public void enqueue(Object el) {
        list.addLast(el);
    }
    public String toString() {
        return list.toString();
    }
}
```

Figure 4-11 Linked list implementation of a queue (continued)

Queues (continued)

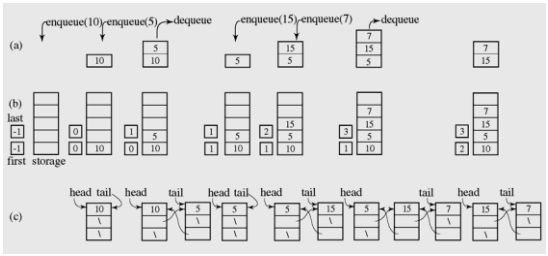


Figure 4-12 A series of operations executed on an abstract queue (a) and the stack implemented with an array (b) and with a linked list (c)

Queues (continued)

- In **queuing theory**, various scenarios are analyzed and models are built that use queues for processing requests or other information in a predetermined sequence (order)

Queues (continued)

Number of Customers per Minute	Percentage of One-Minute Intervals	Range
0	15	1–15
1	20	16–35
2	25	36–60
3	10	61–70
4	30	71–100

(a)

Amount of Time Needed for Service in Seconds	Percentage of Customers	Range
0	0	—
10	0	—
20	0	—
30	10	1–10
40	5	11–15
50	10	16–25
60	10	26–35
70	0	—
80	15	36–50
90	25	51–75
100	10	76–85
110	15	86–100

(b)

Figure 4-13 Bank One example: (a) data for number of arrived customers per one-minute interval and (b) transaction time in seconds per customer

Queues (continued)

```
import java.util.Random;

class BankSimulation {
    static Random rd = new Random();
    static int Option(int percents[]) {
        int i = 0, perc, choice = Math.abs(rd.nextInt()) % 100 + 1;
        for (perc = percents[0]; perc < choice; perc += percents[i+1], i++);
        return i;
    }
    public static void main(String args[]) {
        int[] arrivals = {15,20,25,10,30};
        int[] service = {0,0,0,10,5,10,10,0,15,25,10,15};
        int[] clerks = {0,0,0,0};
        int clerksSize = clerks.length;
        int customers, t, i, numOfMinutes = 100, x;
        double maxWait = 0.0, thereIsLine = 0.0, currWait = 0.0;
        Queue simulQ = new Queue();
```

Figure 4-14 Bank One example: implementation code

Queues (continued)

```
    System.out.print(" t = " + t);
    for (i = 0; i < clerksSize; i++)// after each minute subtract
        if (clerks[i] < 60)         // at most 60 seconds from time
            clerks[i] = 0;          // left to service the current
        else clerks[i] -= 60;       // customer by clerk i;
    customers = Option(arrivals);
    for (i = 0; i < customers; i++) { // enqueue all new customers
        x = Option(service)*10;     // (or rather service time
        simulQ.enqueue(new Integer(x));         // they require);
        currWait += x;
    }
```

Figure 4-14 Bank One example: implementation code (continued)

Queues (continued)

```
    // dequeue customers when clerks are available:
    for (i = 0; i < clerksSize && !simulQ.isEmpty(); )
        if (clerks[i] < 60) {
            x = ((Integer) simulQ.dequeue()).intValue();
                // assign more than one customer
            clerks[i] += x;         // to a clerk if service time
            currWait  -= x;         // is still below 60 sec;
        }
        else i++;
    if (!simulQ.isEmpty()) {
         thereIsLine++;
         System.out.print(" wait = " + ((long)(currWait/6.0)) / 10.0);
         if (maxWait < currWait)
```

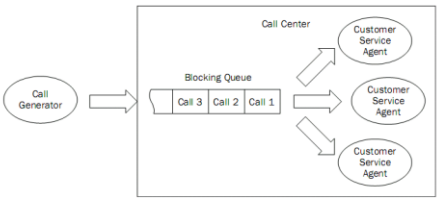
Figure 4-14 Bank One example: implementation code (continued)

Queues (continued)

```
                maxWait = currWait;
            }
            else System.out.print(" wait = 0;");
        }
        System.out.println("\nFor " + clerksSize + " clerks, there was a line "
            + thereIsLine/numOfMinutes*100.0 + "% of the time;\n"
            + "maximum wait time was " + maxWait/60.0 + " min.");
    }
}
```

Figure 4-14 Bank One example: implementation code (continued)

A call center simulation



Queue Exercise

1. Implement a queue with two stacks. What is the complexity of the Enqueue and Dequeue operations ?

11.4.1. Palindromes

- Khái niệm: Một chuỗi được gọi là Palindrome nếu như đọc xuôi giống đọc ngược.
- Bài toán: Cho trước một chuỗi, kiểm tra xem chuỗi đó có phải là chuỗi palindrome hay không?
- Ví dụ về chuỗi palindrome: **Able was I ere I saw Elba**

LAB

Dùng Queue kiểm tra một chuỗi ký tự có đối xứng không?

11.4.2. Demerging

- Tổ chức dữ liệu hợp lý - **Demerging**
- **Bài toán**: Xem xét bài toán sau:
  - Giả sử, với một hệ thống quản lý nhân sự. Các bản ghi được lưu trên file.
  - Mỗi bản ghi gồm các trường: Họ tên, giới tính, ngày tháng năm sinh, ...
  - Dữ liệu trên đã được sắp theo ngày tháng năm sinh.
  - Cần tổ chức lại dữ liệu sao cho nữ được liệt kê trước nam nhưng vẫn giữ được tính đã sắp theo ngày tháng năm sinh.

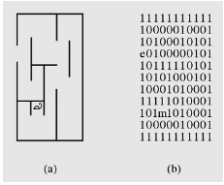
Priority Queues

- A **priority queue** can be assigned to enable a particular process, or event, to be executed out of sequence without affecting overall system operation
- In priority queues, elements are dequeued according to their priority and their current queue position

Priority Queues (continued)

- Priority queues can be represented by two variations of linked lists:
  - All elements are entry ordered based on their priorities.
  - Order is maintained by putting a new element in its proper position according to its priority (the enqueue operation must be modified to satisfy this criterion.)

Case Study: Exiting a Maze



1: wall
0: passage
e: exit
m: mouse

Figure 4-15 (a) A mouse in a maze; (b) two-dimensional character array representing this situation

Ý tưởng: Ô nào đã xét rồi thì đánh dấu để không quay trở lại nữa
-Điểm bắt đầu vào stack
-Khi chưa tìm thấy lối ra thì
-Đánh dấu điểm này là xét rồi
-Cất các ô có thể đi kề chung quanh ô hiện hành vào stack
-Nếu stack trống → Thất bại
-Ngược lại lấy trong stack ra 1 ô để làm điểm hiện hành
(Code: page 161)

Case Study: Exiting a Maze (continued)

```
Pseudocode of an algorithm for escaping a maze
exitMaze()
   initialize stack, exitCell, entryCell, currentCell
    currentCell = entryCell;
    while currentCell is not exitCell
       mark currentCell as visited;
       push onto the stack the unvisited neighbors of currentCell;
       if stack is empty
           failure;
       else pop off a cell from the stack and make it currentCell;
    success;
```


Case Study: Exiting a Maze (continued)

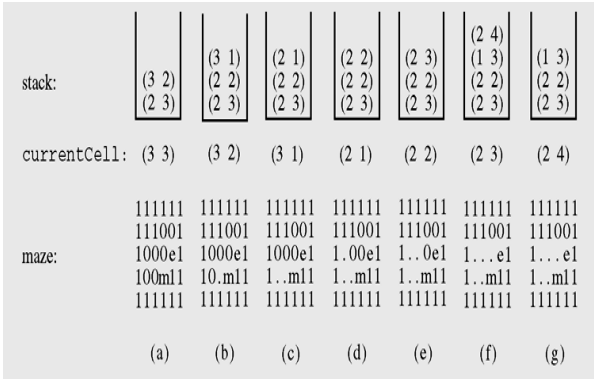


Figure 4-16 An example of processing a maze

Example 1

1	1	1
1	e	0
0	1	0
0	m	0
0	0	0

Example 2

1	1	1	1	1
1	e	1	0	1
1	1	0	1	1
1	1	0	0	0
1	0	0	m	0

Example 3

1	1	1	1	1
1	e	1	0	1
1	0	0	1	1
1	1	0	0	0
1	0	0	m	0

Example 4

1	1	1	0	e
1	1	1	0	1
1	0	0	0	1
1	1	0	0	0
1	0	0	m	0

Case Study: Exiting a Maze (continued)

```
//*************************  Maze.java  ****************************
import java.io.*;

class MazeCell {
    public int x, y;
    public MazeCell() {
    }
    public MazeCell(int i, int j) {
        x = i; y = j;
    }
    public boolean equals(MazeCell cell) {
        return x == cell.x && y == cell.y;
    }
}
```

Figure 4-17 Listing of the program for maze processing

Case Study: Exiting a Maze (continued)

```
class Maze {
    private int rows = 0, cols = 0;
    private char[][] store;
    private MazeCell currentCell, exitCell = new MazeCell(), entryCell = new
    MazeCell();
    private final char exitMarker = 'e', entryMarker = 'm', visited = '.';
    private final char passage = '0', wall = '1';
    private Stack mazeStack = new Stack();
    public Maze() {
        int row = 0, col = 0;
        Stack mazeRows = new Stack();
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader buffer = new BufferedReader(isr);
        System.out.println("Enter a rectangular maze using the following "
                + "characters:\nm - entry\ne - exit\n1 - wall\n0 - passage\n"
                + "Enter one line at at time; end with Ctrl-z:");
```

Figure 4-17 Listing of the program for maze processing (continued)

Case Study: Exiting a Maze (continued)

```
try {
    String str = buffer.readLine();
    while (str != null) {
        row++;
        cols = str.length();
        str = "1" + str + "1";  // put 1s in the borderline cells;
        mazeRows.push(str);
        if (str.indexOf(exitMarker) != -1) {
            exitCell.x = row;
            exitCell.y = str.indexOf(exitMarker);
        }
        if (str.indexOf(entryMarker) != -1) {
            entryCell.x = row;
            entryCell.y = str.indexOf(entryMarker);
        }
        str = buffer.readLine();
    }
```

Figure 4-17 Listing of the program for maze processing (continued)

Case Study: Exiting a Maze (continued)

```
} catch(IOException eof) {
}
rows = row;
store = new char[rows+2][];       // create a 1D array of char arrays;
store[0] = new char[cols+2];      // a borderline row;
for ( ; !mazeRows.isEmpty(); row--)
    store[row] = ((String) mazeRows.pop()).toCharArray();
store[rows+1] = new char[cols+2]; // another borderline row;
for (col = 0; col <= cols+1; col++) {
    store[0][col] = wall;         // fill the borderline rows with 1s;
    store[rows+1][col] = wall;
}
```

Figure 4-17 Listing of the program for maze processing (continued)

Case Study: Exiting a Maze (continued)

```
}
private void display(PrintStream out) {
    for (int row = 0; row <= rows+1; row++)
        out.println(store[row]);
    out.println();
}
private void pushUnvisited(int row, int col) {
    if (store[row][col] == passage || store[row][col] == exitMarker)
        mazeStack.push(new MazeCell(row,col));
}
public void exitMaze(PrintStream out) {
    currentCell = entryCell;
    out.println();
```

Figure 4-17 Listing of the program for maze processing (continued)

Case Study: Exiting a Maze (continued)

```
while (!currentCell.equals(exitCell)) {
    int row = currentCell.x;
    int col = currentCell.y;
    display(System.out);        // print a snapshot;
    if (!currentCell.equals(entryCell))
        store[row][col] = visited;
    pushUnvisited(row-1,col);
    pushUnvisited(row+1,col);
    pushUnvisited(row,col-1);
    pushUnvisited(row,col+1);
    if (mazeStack.isEmpty()) {
        display(out);
        out.println("Failure");
        return;
```

Figure 4-17 Listing of the program for maze processing (continued)

Case Study: Exiting a Maze (continued)

```
            }
            else currentCell = (MazeCell) mazeStack.pop();
        }
        display(out);
        out.println("Success");
    }
    static public void main (String args[]) {
        (new Maze()).exitMaze(System.out);
    }
}
```

Figure 4-17 Listing of the program for maze processing (continued)

Summary

- A stack is a linear data structure that can be accessed at only one of its ends for storing and retrieving data.
- A stack is called an LIFO structure: last in/first out.
- A queue is a waiting line that grows by adding elements to its end and shrinks by taking elements from its front.
- A queue is an FIFO structure: first in/first out.

Summary (continued)

- In queuing theory, various scenarios are analyzed and models are built that use queues for processing requests or other information in a predetermined sequence (order).
- A priority queue can be assigned to enable a particular process, or event, to be executed out of sequence without affecting overall system operation.

Summary (continued)

- In priority queues, elements are dequeued according to their priority and their current queue position.