Chapter 13
Linked Structures - Stacks
Chapter Scope

• Object references as links
• Linked vs. array-based structures
• Managing linked lists
• Linked implementation of a stack
Linked Structures

- An alternative to array-based implementations are *linked structures*
- A linked structure uses object references to create links between objects
- Recall that an object reference variable holds the address of an object
Linked Structures

• A Person object, for instance, could contain a reference to another Person object

• A series of Person objects would make up a linked list:
Linked Structures

- Links could also be used to form more complicated, non-linear structures
Linked Lists

• There are no index values built into linked lists
• To access each node in the list you must follow the references from one node to the next

```java
Person current = first;
while (current != null)
{
    System.out.println(current);
    current = current.next;
}
```
Linked Lists

• Care must be taken to maintain the integrity of the links

• To insert a node at the front of the list, first point the new node to the front node, then reassign the front reference
Linked Lists

• To delete the first node, reassign the `front` reference accordingly.

• If the deleted node is needed elsewhere, a reference to it must be established before reassigning the `front` pointer.
Linked Lists

• So far we've assumed that the list contains nodes that are *self-referential* (*Person points to a Person*)
• But often we'll want to make lists of objects that don't contain such references
• Solution: have a separate `Node` class that forms the list and holds a reference to the objects being stored
Linked Lists

• There are many variations on the basic linked list concept
• For example, we could create a *doubly-linked list* with next and previous references in each node and a separate pointer to the rear of the list
Stacks Revisited

• In the previous chapter we developed our own array-based version of a stack, and we also used the `java.util.Stack` class from the Java API.

• The API's stack class is derived from `Vector`, which has many non-stack abilities.

• It is, therefore, not the best example of inheritance, because a stack is not a vector.

• It's up to the user to use a `Stack` object only as intended.
Stacks Revisited

• Stack characteristics can also be found by using the Deque interface from the API

• The LinkedList class implements the Deque interface

• Deque stands for double-ended queue, and will be explored further later

• For now, we will use the stack characteristics of a Deque to solve the problem of traversing a maze
Traversing a Maze

• Suppose a two-dimensional maze is represented as a grid of 1 (path) and 0 (wall)
• Goal: traverse from the upper left corner to the bottom right (no diagonal moves)
Traversing a Maze

• Using a stack, we can perform a backtracking algorithm to find a solution to the maze
• An object representing a position in the maze is pushed onto the stack when trying a path
• If a dead end is encountered, the position is popped and another path is tried
• We'll change the integers in the maze grid to represent tried-but-failed paths (2) and the successful path (3)
import java.util.*;
import java.io.*;

/**
 * Maze represents a maze of characters. The goal is to get from the
 * top left corner to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Java Foundations
 * @version 4.0
 */

public class Maze
{
    private static final int TRIED = 2;
    private static final int PATH = 3;

    private int numberRows, numberColumns;
    private int[][] grid;
/**
 * Constructor for the Maze class. Loads a maze from the given file.
 * Throws a FileNotFoundException if the given file is not found.
 * 
 * @param filename the name of the file to load
 * @throws FileNotFoundException if the given file is not found
 */

public Maze(String filename) throws FileNotFoundException {
    Scanner scan = new Scanner(new File(filename));
    numberRows = scan.nextInt();
    numberColumns = scan.nextInt();

    grid = new int[numberRows][numberColumns];
    for (int i = 0; i < numberRows; i++)
        for (int j = 0; j < numberColumns; j++)
            grid[i][j] = scan.nextInt();
}
/**
 * Marks the specified position in the maze as TRIED
 * @param row the index of the row to try
 * @param col the index of the column to try
 */
public void tryPosition(int row, int col) {
    grid[row][col] = TRIED;
}

/**
 * Return the number of rows in this maze
 * @return the number of rows in this maze
 */
public int getRows() {
    return grid.length;
}

/**
 * Return the number of columns in this maze
 * @return the number of columns in this maze
 */
public int getColumnns() {
    return grid[0].length;
}
/**
 * Marks a given position in the maze as part of the PATH
 * @param row the index of the row to mark as part of the PATH
 * @param col the index of the column to mark as part of the PATH
 */
public void markPath(int row, int col)
{
    grid[row][col] = PATH;
}

/**
 * Determines if a specific location is valid. A valid location
 * is one that is on the grid, is not blocked, and has not been TRIED.
 * @param row the row to be checked
 * @param column the column to be checked
 * @return true if the location is valid
 */
public boolean validPosition(int row, int column)
{
    boolean result = false;

    // check if cell is in the bounds of the matrix
    if (row >= 0 && row < grid.length &&
        column >= 0 && column < grid[row].length)
    {
        // check if cell is not blocked and not previously tried
        if (grid[row][column] == 1)
            result = true;

        return result;
    }
/**
 * Returns the maze as a string.
 *
 * @return a string representation of the maze
 */

public String toString()
{
    String result = "\n";

    for (int row=0; row < grid.length; row++)
    {
        for (int column=0; column < grid[row].length; column++)
            result += grid[row][column] + "\n";
    }

    return result;
}

import java.util.*;

/**
 * MazeSolver attempts to traverse a Maze using a stack. The goal is to get from the
 * given starting position to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Java Foundations
 * @version 4.0
 */

public class MazeSolver
{
    private Maze maze;

    /**
     * Constructor for the MazeSolver class.
     */
    public MazeSolver(Maze maze)
    {
        this.maze = maze;
    }
}
/**
 * Attempts to traverse the maze using a stack. Inserts special
 * characters indicating locations that have been TRIED and that
 * eventually become part of the solution PATH.
 *
 * @param row row index of current location
 * @param column column index of current location
 * @return true if the maze has been solved
 */

public boolean traverse()
{
    boolean done = false;
    int row, column;
    Position pos = new Position();
    Deque<Position> stack = new LinkedList<Position>();
    stack.push(pos);

    while (!(done) && !stack.isEmpty())
    {
        pos = stack.pop();
        maze.tryPosition(pos.getx(), pos.gety()); // this cell has been tried
        if (pos.getx() == maze.getRows()-1 && pos.gety() == maze.getColumns()-1)
            done = true; // the maze is solved
        else
        {
            push_new_pos(pos.getx() - 1, pos.gety(), stack);
            push_new_pos(pos.getx() + 1, pos.gety(), stack);
            push_new_pos(pos.getx(), pos.gety() - 1, stack);
            push_new_pos(pos.getx(), pos.gety() + 1, stack);
        }
    }

    return done;
}
/**
 * Push a new attempted move onto the stack
 * @param x represents x coordinate
 * @param y represents y coordinate
 * @param stack the working stack of moves within the grid
 * @return stack of moves within the grid
 */

private void push_new_pos(int x, int y,
                            Deque<Position> stack)
{
    Position npos = new Position();
    npos.setx(x);
    npos.sety(y);
    if (maze.validPosition(x, y))
        stack.push(npos);
}

import java.util.*;
import java.io.*;

/**
 * MazeTester determines if a maze can be traversed.
 * @author Java Foundations
 * @version 4.0
 */
public class MazeTester
{
    /**
     * Creates a new maze, prints its original form, attempts to
     * solve it, and prints out its final form.
     */
    public static void main(String[] args) throws FileNotFoundException
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter the name of the file containing the maze: ");
        String filename = scan.nextLine();

        Maze labyrinth = new Maze(filename);
        System.out.println(labyrinth);
        MazeSolver solver = new MazeSolver(labyrinth);

        if (solver.traverse())
            System.out.println("The maze was successfully traversed!");
        else
            System.out.println("There is no possible path.");

        System.out.println(labyrinth);
    }
}
Implementing a Stack using Links

• Let's now implement our own version of a stack that uses a linked list to hold the elements

• Our `LinkedStack<T>` class stores a generic type `T` and implements the same `StackADT<T>` interface used previously

• A separate `LinearNode<T>` class forms the list and hold a reference to the element stored

• An integer `count` will store how many elements are currently in the stack
Implementing a Stack using Links

- Since all activity on a stack happens on one end, a single reference to the front of the list will represent the top of the stack.
Implementing a Stack using Links

• The stack after A, B, C, and D are pushed, in that order:
Implementing a Stack using Links

- After E is pushed onto the stack:
package jsjf;

/**
 * Represents a node in a linked list.
 *
 * @author Java Foundations
 * @version 4.0
 */

public class LinearNode<T> {
    private LinearNode<T> next;
    private T element;

    /**
     * Creates an empty node.
     */
    public LinearNode() {
        next = null;
        element = null;
    }

    /**
     * Creates a node storing the specified element.
     * @param elem element to be stored
     */
    public LinearNode(T elem) {
        next = null;
        element = elem;
    }
}
/**
 * Returns the node that follows this one.
 * @return reference to next node
 */
public LinearNode<T> getNext()
{
    return next;
}

/**
 * Sets the node that follows this one.
 * @param node node to follow this one
 */
public void setNext(LinearNode<T> node)
{
    next = node;
}

/**
 * Returns the element stored in this node.
 * @return element stored at the node
 */
public T getElement()
{
    return element;
}

/**
 * Sets the element stored in this node.
 * @param elem element to be stored at this node
 */
public void setElement(T elem)
{
    element = elem;
}
package jsjf;

import jsjf.exceptions.*;
import java.util.Iterator;

/**
 * Represents a linked implementation of a stack.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class LinkedStack<T> implements StackADT<T> {
    private int count;
    private LinearNode<T> top;

    /**
     * Creates an empty stack.
     */
    public LinkedStack() {
        count = 0;
        top = null;
    }
/**
 * Adds the specified element to the top of this stack.
 * @param element element to be pushed on stack
 */
public void push(T element)
{
    LinearNode<T> temp = new LinearNode<T>(element);
    temp.setNext(top);
    top = temp;
    count++;
}

/**
 * Removes the element at the top of this stack and returns a
 * reference to it.
 * @return element from top of stack
 * @throws EmptyCollectionException if the stack is empty
 */
public T pop() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("stack");
    T result = top.getElement();
    top = top.getNext();
    count--;
    return result;
}
Implementing a Stack using Links

Adding a Node to the Front of a Linked List

Set the new node’s next reference to the front of the list

Temp. setNext (top);

top = temp;

reset the front of the list