Recursive Data Structures

- Computer scientists often encounter data structures that are defined recursively
  - Trees (Chapter 8) are defined recursively
- Linked list can be described as a recursive data structure
- Recursive methods provide a very natural mechanism for processing recursive data structures
- The first language developed for artificial intelligence research was a recursive language called LISP
Recursive Definition of a Linked List

• A non-empty linked list is a collection of nodes such that each node references another linked list consisting of the nodes that follow it in the list
• The last node references an empty list
• A linked list is empty, or it contains a node, called the list head, that stores data and a reference to a linked list
Recursive Size Method

/** Finds the size of a list.  
   * @param head The head of the current list 
   * @return The size of the current list 
   */
private int size(Node<E> head) {
    if (head == null)
        return 0;
    else
        return 1 + size(head.next);
}

/** Wrapper method for finding the size of a list. 
   * @return The size of the list 
   */
public int size() {
    return size(head);
}
Recursive toString Method

/**
 * Returns the string representation of a list.
 * @param head The head of the current list
 * @return The state of the current list
 */
private String toString(Node<E> head) {
    if (head == null)
        return "";
    else
        return head.data + "\n" + toString(head.next);
}

/**
 * Wrapper method for returning the string representation of a list.
 * @return The string representation of the list
 */
public String toString() {
    return toString(head);
}
Recursive Replace Method

```java
/**
 * Replaces all occurrences of oldObj with newObj.
 * post: Each occurrence of oldObj has been replaced by newObj.
 * @param head The head of the current list
 * @param oldObj The object being removed
 * @param newObj The object being inserted
 */
private void replace(Node<E> head, E oldObj, E newObj) {
    if (head != null) {
        if (oldObj.equals(head.data))
            head.data = newObj;
        replace(head.next, oldObj, newObj);
    }
}

/**
 * Wrapper method for replacing oldObj with newObj.
 * post: Each occurrence of oldObj has been replaced by newObj.
 * @param oldObj The object being removed
 * @param newObj The object being inserted
 */
public void replace(E oldObj, E newObj) {
    replace(head, oldObj, newObj);
}
```
Recursive Add Method

/** Adds a new node to the end of a list. 
 * @param head The head of the current list 
 * @param data The data for the new node 
 */
private void add(Node<E> head, E data) {
    // If the list has just one element, add to it.
    if (head.next == null)
        head.next = new Node<E>(data);
    else
        add(head.next, data); // Add to rest of list.
}

/** Wrapper method for adding a new node to the end of a list. 
 * @param data The data for the new node 
 */
public void add(E data) {
    if (head == null)
        head = new Node<E>(data); // List has 1 node.
    else
        add(head, data);
}
/** Removes a node from a list.
   post: The first occurrence of outData is removed.
   @param head The head of the current list
   @param pred The predecessor of the list head
   @param outData The data to be removed
   @return true if the item is removed
            and false otherwise
*/
private boolean remove(Node<E> head, Node<E> pred, E outData) {
    if (head == null) // Base case - empty list.
        return false;
    else if (head.data.equals(outData)) { // 2nd base case.
        pred.next = head.next; // Remove head.
        return true;
    } else
        return remove(head.next, head, outData);
}
/** Wrapper method for removing a node (in LinkedListRec). 
   post: The first occurrence of outData is removed. 
   @param outData The data to be removed 
   @return true if the item is removed, 
            and false otherwise 
*/
public boolean remove(E outData) {
    if (head == null)
        return false;
    else if (head.data.equals(outData)) {
        head = head.next;
        return true;
    } else
        return remove(head.next, head, outData);
}
Problem Solving with Recursion

• Will look at two problems
  • Towers of Hanoi
  • Counting cells in a blob

![Figure 7.11: Children's Version of Towers of Hanoi](image1)

![Figure 7.16: A Sample Grid for Counting Cells in a Blob](image2)
Towers of Hanoi

FIGURE 7.11
Children’s Version of Towers of Hanoi
Towers of Hanoi (continued)

**TABLE 7.1**
Inputs and Outputs for Towers of Hanoi Problem

<table>
<thead>
<tr>
<th>Problem Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of disks (an integer)</td>
</tr>
<tr>
<td>Letter of starting peg: L (left), M (middle), or R (right)</td>
</tr>
<tr>
<td>Letter of destination peg (L, M, or R), but different from starting peg</td>
</tr>
<tr>
<td>Letter of temporary peg (L, M, or R), but different from starting peg and destination peg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A list of moves</td>
</tr>
</tbody>
</table>
Algorithm for Towers of Hanoi

Solution to 3-Disk Problem: Move 3 Disks from Peg L to Peg R
1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.

Figure 7.12
Towers of Hanoi After the First Two Steps in Solution of the Three-Disk Problem
Algorithm for Towers of Hanoi (continued)

Solution to 2-Disk Problem: Move Top 2 Disks from Peg M to Peg R
1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.

In Figure 7.13 we show the pegs after steps 1 and 2. When step 3 is completed, the three pegs will be on peg R.

Figure 7.13
Towers of Hanoi After First Two Steps in Solution of Two-Disk Problem
Algorithm for Towers of Hanoi (continued)

Solution to 4-Disk Problem: Move 4 Disks from Peg L to Peg R
1. Move the top three disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top three disks from peg M to peg R.

Figure 7.14
Towers of Hanoi After the First Two Steps in Solution of the Four-Disk Problem
Recursive Algorithm for $n$-Disk Problem: Move $n$ Disks from the Starting Peg to the Destination Peg

1. if $n$ is 1
2. Move disk 1 (the smallest disk) from the starting peg to the destination peg.
3. else
4. Move the top $n - 1$ disks from the starting peg to the temporary peg (neither starting nor destination peg).
5. Move disk $n$ (the disk at the bottom) from the starting peg to the destination peg.
6. Move the top $n - 1$ disks from the temporary peg to the destination peg.
Implementation of Recursive Towers of Hanoi

```java
/** Class that solves Towers of Hanoi problem. */
public class TowersOfHanoi {
  /** Recursive method for "moving" disks.
   * pre: startPeg, destPeg, tempPeg are different.
   * @param n is the number of disks
   * @param startPeg is the starting peg
   * @param destPeg is the destination peg
   * @param tempPeg is the temporary peg
   * @return A string with all the required disk moves
   */
  public static String showMoves(int n, char startPeg, char destPeg, char tempPeg) {
    if (n == 1) {
      return "Move disk 1 from peg " + startPeg + " to peg " + destPeg + "\n";
    } else { // Recursive step
      return showMoves(n - 1, startPeg, tempPeg, destPeg) + "Move disk " + n + " from peg " + startPeg + " to peg " + destPeg + "\n" + showMoves(n - 1, tempPeg, destPeg, startPeg);
    }
  }
}
```
Counting Cells in a Blob

- Consider how we might process an image that is presented as a two-dimensional array of color values
- Information in the image may come from
  - X-Ray
  - MRI
  - Satellite imagery
  - Etc.
- Goal is to determine the size of any area in the image that is considered abnormal because of its color values
Counting Cells in a Blob (continued)

**Table 7.3**
Class TwoDimGrid

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>void recolor(int x, int y, Color aColor)</td>
<td>Resets the color of the cell at position (x, y) to aColor.</td>
</tr>
<tr>
<td>Color getColor(int x, int y)</td>
<td>Retrieves the color of the cell at position (x, y).</td>
</tr>
<tr>
<td>int getNROWS()</td>
<td>Returns the number of cells in the y-axis.</td>
</tr>
<tr>
<td>int getNCOLS()</td>
<td>Returns the number of cells in the x-axis.</td>
</tr>
</tbody>
</table>

**Table 7.4**
Class Blob

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>int countCells(int x, int y)</td>
<td>Returns the number of cells in the blob at (x, y).</td>
</tr>
</tbody>
</table>
Counting Cells in a Blob (continued)

Algorithm for `countCells(x, y)`

1. if the cell at (x, y) is outside the grid
2. The result is 0.
   else if the color of the cell at (x, y) is not the abnormal color
3. The result is 0.
   else
4. Set the color of the cell at (x, y) to a temporary color.
5. The result is 1 plus the number of cells in each piece of the blob that includes a nearest neighbor.
import java.awt.*;

/** Class that solves problem of counting abnormal cells. */
public class Blob implements GridColors {

 /** The grid */
 private TwoDimGrid grid;

 /** Constructors */
 public Blob(TwoDimGrid grid) {
   this.grid = grid;
 }
}
Implementation (continued)

```java
/**
  * Finds the number of cells in the blob at (x,y).
  * <pre>
  * pre: Abnormal cells are in ABNORMAL color;
  *     Other cells are in BACKGROUND color.
  * post: All cells in the blob are in the TEMPORARY color.
  * @param x  The x-coordinate of a blob cell
  * @param y  The y-coordinate of a blob cell
  * @return  The number of cells in the blob that contains (x, y)
  */
  public int countCells(int x, int y) {
    int result;

    if (x < 0 || x >= grid.getNCols() || y < 0 || y >= grid.getNRows())
      return 0;
    else if (!grid.getColor(x, y).equals(ABNORMAL))
      return 0;
    else {
      grid.recolor(x, y, TEMPORARY);
      return 1
        + countCells(x - 1, y + 1) + countCells(x, y + 1)
        + countCells(x + 1, y + 1) + countCells(x - 1, y)
        + countCells(x + 1, y) + countCells(x - 1, y - 1)
        + countCells(x, y - 1) + countCells(x + 1, y - 1);
    }
  }
  }
```
Counting Cells in a Blob (continued)
Backtracking

- Backtracking is an approach to implementing systematic trial and error in a search for a solution
  - An example is finding a path through a maze
- If you are attempting to walk through a maze, you will probably walk down a path as far as you can go
  - Eventually, you will reach your destination or you won’t be able to go any farther
  - If you can’t go any farther, you will need to retrace your steps
- Backtracking is a systematic approach to trying alternative paths and eliminating them if they don’t work
Backtracking (continued)

• Never try the exact same path more than once, and you will eventually find a solution path if one exists

• Problems that are solved by backtracking can be described as a set of choices made by some method

• Recursion allows us to implement backtracking in a relatively straightforward manner
  • Each activation frame is used to remember the choice that was made at that particular decision point

• A program that plays chess may involve some kind of backtracking algorithm
Backtracking (continued)

**Figure 7.18**
Maze as Grid of Buttons Before SOLVE Is Clicked

**Figure 7.19**
Maze as Grid of Buttons After SOLVE Is Clicked
Recursive Algorithm for finding Maze Path

**Recursive Algorithm for findMazePath**(x, y)

1. if the current cell is outside the maze
2. Return *false* (you are out of bounds).
   
   **else** if the current cell is part of the barrier or has already been visited
3. Return *false* (you are off the path or in a cycle).
   
   **else** if the current cell is the maze exit
4. Recolor it to the path color and return *true* (you have successfully completed the maze).
   
   **else** // Try to find a path from the current path to the exit:
5. Mark the current cell as on the path by recoloring it to the path color.
6. **for** each neighbor of the current cell
7. if a path exists from the neighbor to the maze exit
8. Return *true*.
   
   // No neighbor of the current cell is on the path.
9. Recolor the current cell to the temporary color (visited) and return *false*.
Implementation

```java
/** Class that solves maze problems with backtracking. */
public class Maze implements GridColors {

    /** The maze */
    private TwoDimGrid maze;

    public Maze(TwoDimGrid m) {
        maze = m;
    }

    /** Wrapper method. */
    public boolean findMazePath() {
        return findMazePath(0, 0); // (0, 0) is the start point.
    }

```
/** Attempts to find a path through point (x, y).
pre: Possible path cells are in BACKGROUND color;
    barrier cells are in ABNORMAL color.
post: If a path is found, all cells on it are set to the
    PATH color; all cells that were visited but are
    not on the path are in the TEMPORARY color.
@param x The x-coordinate of current point
@param y The y-coordinate of current point
@return If a path through (x, y) is found, true;
    otherwise, false
*/
public boolean findMazePath(int x, int y) {
    if (x < 0 || y < 0
        || x >= maze.getNCols() || y >= maze.getNRows())
        return false; // Cell is out of bounds.
    else if (!maze.getColor(x, y).equals(BACKGROUND))
        return false; // Cell is on barrier or dead end.
    else if (x == maze.getNCols() - 1
        && y == maze.getNRows() - 1) {
        maze.recolor(x, y, PATH); // Cell is on path
        return true; // and is maze exit.
    } else { // Recursive case.
        maze.recolor(x, y, PATH);
        if (findMazePath(x - 1, y)
            || findMazePath(x + 1, y)
            || findMazePath(x, y - 1)
Implementation (continued)

```java
|| findMazePath(x, y + 1 ) ) {
    return true;
} else {
    maze.recolor(x, y, TEMPORARY);  // Dead end.
    return false;
}
```
Chapter Review

- A recursive method has a standard form
- To prove that a recursive algorithm is correct, you must
  - Verify that the base case is recognized and solved correctly
  - Verify that each recursive case makes progress toward the base case
  - Verify that if all smaller problems are solved correctly, then the original problem must also be solved correctly
- The run-time stack uses activation frames to keep track of argument values and return points during recursive method calls
Chapter Review (continued)

- Mathematical Sequences and formulas that are defined recursively can be implemented naturally as recursive methods.
- Recursive data structures are data structures that have a component that is the same data structure.
- Towers of Hanoi and counting cells in a blob can both be solved with recursion.
- Backtracking is a technique that enables you to write programs that can be used to explore different alternative paths in a search for a solution.