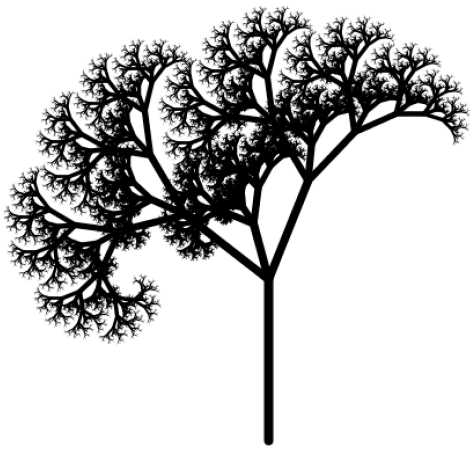
CMSC 132, Object-Oriented Programming II **Summer 2020**

Lecture 11:

Instructor: Anwar Mamat

11.1 Recursion Examples

11.1.1 Recursive Shapes: Tree



Listing 1: Recursive Shapes: Tree

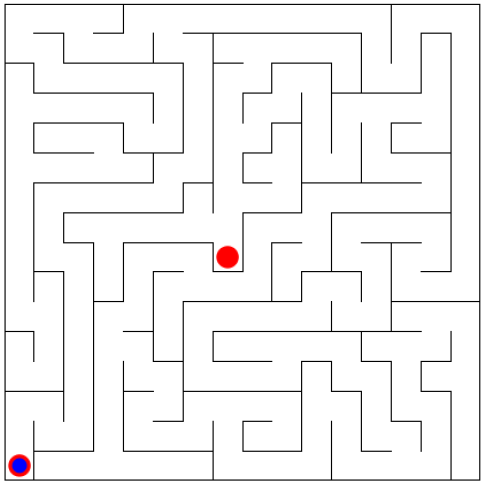
```
1  /*************************************************************************
2  *   Copyright   2000?2010, Robert Sedgewick and Kevin Wayne.
3   *  Compilation:  javac Tree.java
4   *  Execution:    java Tree N
5   *  Dependencies: StdDraw.java
6   *
7   *  Plot a tree fractal.
8   *
9   *  % java Tree 9
10  *
11  *************************************************************************/
12
13 public class Tree {
14
15     public static void tree(int n, double x, double y, double a, double branchRadius) {
16         double bendAngle   = Math.toRadians(15);
17         double branchAngle = Math.toRadians(37);
18         double branchRatio = .65;
19
20         double cx = x + Math.cos(a) * branchRadius;
21         double cy = y + Math.sin(a) * branchRadius;
22         StdDraw.setPenRadius(.001 * Math.pow(n, 1.2));
```

```
        StdDraw.line(x, y, cx, cy);
        if (n == 0) return;

        tree(n-1, cx, cy, a + bendAngle - branchAngle, branchRadius * branchRatio);
        tree(n-1, cx, cy, a + bendAngle + branchAngle, branchRadius * branchRatio);
        tree(n-1, cx, cy, a + bendAngle,               branchRadius * (1 - branchRatio));
  }

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        StdDraw.show(0);
        tree(N, .5, 0, Math.PI/2, .3);
        StdDraw.show(0);
    }
}
```

11.1.2 Maze



Listing 2: Maze Solver

```
/*************************************************************************
 *  Compilation:  javac Maze.java
 *  Execution:    java Maze.java N
 *  Dependecies:  StdDraw.java
 *
 *  Generates a perfect N-by-N maze using depth-first search with a stack.
 *  % java Maze 62
 *  % java Maze 61
 *
 *  Note: this program generalizes nicely to finding a random tree
 *        in a graph.
 *
 *      Copyright   2002?2010, Robert Sedgewick and Kevin Wayne.
 *
 *************************************************************************/

public class Maze {
    private int N;                 // dimension of maze
    private boolean[][] north;     // is there a wall to north of cell i, j
    private boolean[][] east;
    private boolean[][] south;
```

```
22      private boolean[][] west;
23      private boolean[][] visited;
24      private boolean done = false;
25
26      public Maze(int N) {
27          this.N = N;
28          StdDraw.setXscale(0, N+2);
29          StdDraw.setYscale(0, N+2);
30          init();
31          generate();
32      }
33
34      private void init() {
35          // initialize border cells as already visited
36          visited = new boolean[N+2][N+2];
37          for (int x = 0; x < N+2; x++) visited[x][0] = visited[x][N+1] = true;
38          for (int y = 0; y < N+2; y++) visited[0][y] = visited[N+1][y] = true;
39
40
41          // initialze all walls as present
42          north = new boolean[N+2][N+2];
43          east  = new boolean[N+2][N+2];
44          south = new boolean[N+2][N+2];
45          west  = new boolean[N+2][N+2];
46          for (int x = 0; x < N+2; x++)
47              for (int y = 0; y < N+2; y++)
48                  north[x][y] = east[x][y] = south[x][y] = west[x][y] = true;
49      }
50
51
52      // generate the maze
53      private void generate(int x, int y) {
54          visited[x][y] = true;
55
56          // while there is an unvisited neighbor
57          while (!visited[x][y+1] || !visited[x+1][y] || !visited[x][y-1] || !visited[x-1][y]) {
58
59              // pick random neighbor (could use Knuth's trick instead)
60              while (true) {
61                  double r = Math.random();
62                  if (r < 0.25 && !visited[x][y+1]) {
63                      north[x][y] = south[x][y+1] = false;
64                      generate(x, y + 1);
65                      break;
66                  }
67                  else if (r >= 0.25 && r < 0.50 && !visited[x+1][y]) {
68                      east[x][y] = west[x+1][y] = false;
69                      generate(x+1, y);
70                      break;
71                  }
72                  else if (r >= 0.5 && r < 0.75 && !visited[x][y-1]) {
73                      south[x][y] = north[x][y-1] = false;
74                      generate(x, y-1);
75                      break;
76                  }
77                  else if (r >= 0.75 && r < 1.00 && !visited[x-1][y]) {
78                      west[x][y] = east[x-1][y] = false;
79                      generate(x-1, y);
80                      break;
81                  }
82              }
83          }
84      }
85
86      // generate the maze starting from lower left
87      private void generate() {
88          generate(1, 1);
89
```

```
90  /*
91          // delete some random walls
92          for (int i = 0; i < N; i++) {
93              int x = (int) (1 + Math.random() * (N-1));
94              int y = (int) (1 + Math.random() * (N-1));
95              north[x][y] = south[x][y+1] = false;
96          }
97
98          // add some random walls
99          for (int i = 0; i < 10; i++) {
100             int x = (int) (N / 2 + Math.random() * (N / 2));
101             int y = (int) (N / 2 + Math.random() * (N / 2));
102             east[x][y] = west[x+1][y] = true;
103         }*/
104
105     }
106
107
108
109     // solve the maze using depth-first search
110     private void solve(int x, int y) {
111         if (x == 0 || y == 0 || x == N+1 || y == N+1) return;
112         if (done || visited[x][y]) return;
113         visited[x][y] = true;
114
115         StdDraw.setPenColor(StdDraw.BLUE);
116         StdDraw.filledCircle(x + 0.5, y + 0.5, 0.25);
117         StdDraw.show(0);
118         pressAnyKeyToContinue();
119         // reached middle
120         if (x == N/2 && y == N/2) done = true;
121         //solve(x+1, y + 1); //goes diagonally
122         if (!north[x][y]) solve(x, y + 1);
123         if (!east[x][y])  solve(x + 1, y);
124         if (!south[x][y]) solve(x, y - 1);
125         if (!west[x][y])  solve(x - 1, y);
126
127         if (done) return;
128
129         StdDraw.setPenColor(StdDraw.PINK);
130         StdDraw.filledCircle(x + 0.5, y + 0.5, 0.25);
131         StdDraw.show(0);
132         pressAnyKeyToContinue();
133     }
134
135     // solve the maze starting from the start state
136     public void solve() {
137         for (int x = 1; x <= N; x++)
138             for (int y = 1; y <= N; y++)
139                 visited[x][y] = false;
140         done = false;
141         solve(1, 1);
142         if(done) {
143                         System.out.println("Solved!");
144         }else{
145                         System.out.println("cannot␣solve!");
146         }
147     }
148
149     // draw the maze
150     public void draw() {
151         StdDraw.setPenColor(StdDraw.RED);
152         StdDraw.filledCircle(N/2.0 + 0.5, N/2.0 + 0.5, 0.375);
153         StdDraw.filledCircle(1.5, 1.5, 0.375);
154
155         StdDraw.setPenColor(StdDraw.BLACK);
156         for (int x = 1; x <= N; x++) {
157             for (int y = 1; y <= N; y++) {
```

```
158                 if (south[x][y]) StdDraw.line(x, y, x + 1, y);
159                 if (north[x][y]) StdDraw.line(x, y + 1, x + 1, y + 1);
160                 if (west[x][y])  StdDraw.line(x, y, x, y + 1);
161                 if (east[x][y])  StdDraw.line(x + 1, y, x + 1, y + 1);
162             }
163         }
164         StdDraw.show(1000);
165     }
166 
167 
168     private void pressAnyKeyToContinue()
169  {
170         System.out.println("Press␣any␣key␣to␣continue...");
171         try
172         {
173             System.in.read();
174         }
175         catch(Exception e)
176         {}
177  }
178 
179     // a test client
180     public static void main(String[] args) {
181         int N = 16;   //Integer.parseInt(args[0]);
182         Maze maze = new Maze(N);
183         StdDraw.show(0);
184         maze.draw();
185         maze.solve();
186     }
187 
188 }
```