FACULTY OF ENGINEERING AND APPLIED SCIENCE

Department of Electrical, Computer and Software Engineering

SOFE 2715 Data Structure / Group Project (20%)

Group 10 Members

- Massimo Albanese - 100616057
- Clarissa Branje - 100716458
- Haider Sarmad - 100622306
- Tegveer Singh -100730432

<u>Table of Contents</u>

Project Description

Our team collectively agreed to take on the challenge of the Project 10 Galton Board Simulation. By doing so, not only did the group gain a better understanding of the importance of Pretransersing trees/ patterns, but our partnership was also able to display the phenomenon of the galton board. The way the board works is a defined number of balls will fall upon a peg and will bounce to the left or right. This process continues as the balls fall through several rows of peggs and the balls will accumulate into one of n bins. The interesting outcome to the board is that the balls will appear to display the Central Limit Theorem pattern; this states that given enough independent random variables the total aggregate outcome of their results converges to an approximately normal distribution.

To fulfill the objective, the program must be able to simulate six different numbers of balls to be dropped, as well as show each additional random point being shown (not the movement of the balls but the creation of the distribution). The project was designed with the software resources of Eclipse and IntelliJ IDEA but a very key software that was also frequently used was Discord as code was sent to each member with this application. To add, the app was especially useful for team meetings from home as screen share was available for members to see the code being developed at the same time.

Project Background

In order to remain fair and organised, the group would each propose their best approach to the problem. From there the project was divided into stages for each member to be the leader of that stage; where the leader would be coding the program at that stage and the other members must follow up after a major update. To focus on each member's contribution, Clarissa highly participated in the early stages of sample solutions that first simulated the project. See Figure A1 and A2 in Appendix A for early staged development.

While the program did provide the necessary information (eg, the slot each ball fell in and the final output), the output becomes messy and unclear after inputs that are bigger than 100 let alone 10 000 or 100 000 (see figure A3). This is where Massimo led the division of the project into several classes, and shaped the program into its final form; see Appendix B for the final code and Project Breakdown structure for more detailed information on the final approach. There were many struggles that had to be faced in relation to the data structure; it had to be fully declared, filled, and indexed before computation.

At this stage there were many complications; for example, the structure must be able to be easily built at any given size and allow easy transversal. To also aid the obstacle of disorganised transversal, potential paths were already determined instead of relying on random generation. While recursive traversal was in mind, the group settled on a graph that is similar to a binary tree with each node having two parents and two children (a peg could direct a ball to the left or right and could have been hit from the left or right side). The data structure is then stored in an array of node classes with each class holding 4 values: the index of the node in the array, the level where the node is situated (See figure D1), and the index of the nodes left and right child.

To preserve a true Galton Board Simulation with alternating pins, the nodes in the inner layers share 1 child with each of its neighbors. It was also important to note that the final layer of nodes had no children but a “bin” for the ball to land into. These “bins” are node objects and hold a separate index for where the balls are summed instead of having references to children in an array of integers with a size of n. Within the first attempts to compute the output in early recursive trials for each ball in each transversal, there were stack overflows at 500 balls. In the second attempt, there was an iterative approach for each ball with a recursion method to calculate the graph transversal. This combination allowed us to have a big O of O(n), with the computational time scaling linearly with the amount of balls dropped see figure D3.

Tegveer and Haider, were heavily involved with the graphical interface as well as suggesting early solutions and breaking up methods into classes with Massimo. Clarissa also led the documentation portion of diagrams with no issues faced as the group actively participated in each stage giving suggestions for potential better changes/stages.

Several troubles occurred during the implementation of the animated portion of the code while using a window builder software (eclipse’s windowbuilder, intellij idea’s formbuilderMultiple methods for graphical representation). It was hard to understand how to create JTables with coloured boxes as well as rendering and building a coloured Jpanel that had specified heights.

To face these challenges, Massimo decided to create a graphical user interface from scratch, as well as an animated JPanel, using the highly useful Graphics2D. The graphics portion sets the background to the color desired; for this project white was selected. Next, the graphical interface unit iterates through the bin array, it then fills each red rectangle proportional to the percentage of balls in the slot compared to the sum of balls. Lastly, the GUI draws a black frame around the red rectangle while drawing each text.The GUI has the capability of displaying the amount of balls in each spot. The output can be found in Appendix C.

Project Breakdown Structure

Node Class

Each Node class has two parameters in common: **index** and **level**. The **index** parameter holds the index of the node in the array, and the **level** holds the level of the node in the graph (Figure D2). Every level of nodes in the graph save the last has two other parameters: **left** and **right**. These parameters store the indexes of the left and right children respectively. Through method overloading, the final level of nodes in the graph have a unique parameter, **bin**. This holds the index of the corresponding 'bin' the balls fall into, which is an integer array of the same size as the amount of nodes on the final level.

PinGraph Class

`private Node[] fill(int size)`

This function generates the 'pins' the balls fall through. Each 'pin' is a **node** class. When inputted a size **n**, it will generate a graph starting with one root node of index 0 on the first level. Each level generated after has one more node than the last, with the final level having **n** nodes (Figure D2). Each node generated is given an index, a level, and the indexes of its children. The children's indexes are determined by the sum of the index of the node and the level for the left, and the same plus 1 for the right. For example, if the graph is of size 6, the children of node 4 will be of index 7 and 8 (Figure D2). Every node is stored in an array, which enables quick access of parameters. The size of the array is computed recursively in the **sum** function. **Fill** returns a filled array of nodes.

`private int sum(int size, int sum)`

This function computes the size of the array recursively. It sums the size of each level a PinGraph of size **n** would have, then returns an integer value of the size of the array. For example, if the PinGraph has a size of 6, it will have 21 nodes, so an array of size 21.

GaltonBoard Class (main class)

`private static int[] compute(PinGraph pins, int size, int balls)`

The **compute** function iteratively goes through each ball dropped and calls the **findBin** function for each one. It creates an array of integers of size **n** (same size as the final level of the **PinGraph**), and will add 1 to the corresponding bin the **findBin** function returns for each ball. **Compute** returns a filled array of integers, with each integer in that array representing the amount of balls dropped in that bin.

`static int findBin(int index, PinGraph pins)`

The **findBin** function is a recursive function that travels the **PinGraph** until it reaches the final level. For each **node** it encounters in the **Pingraph**, it generates a random number between 1 and 2, which calls **findBin** with the index of the left or right child respectively. The base case is only the nodes on the final level that have a **bin** parameter, so when that value is not 0, it returns the **bin** value of the final node.

GaltonBoardGUI Class (main class for GUI)

`public class GaltonBoardGUI extends JFrame`

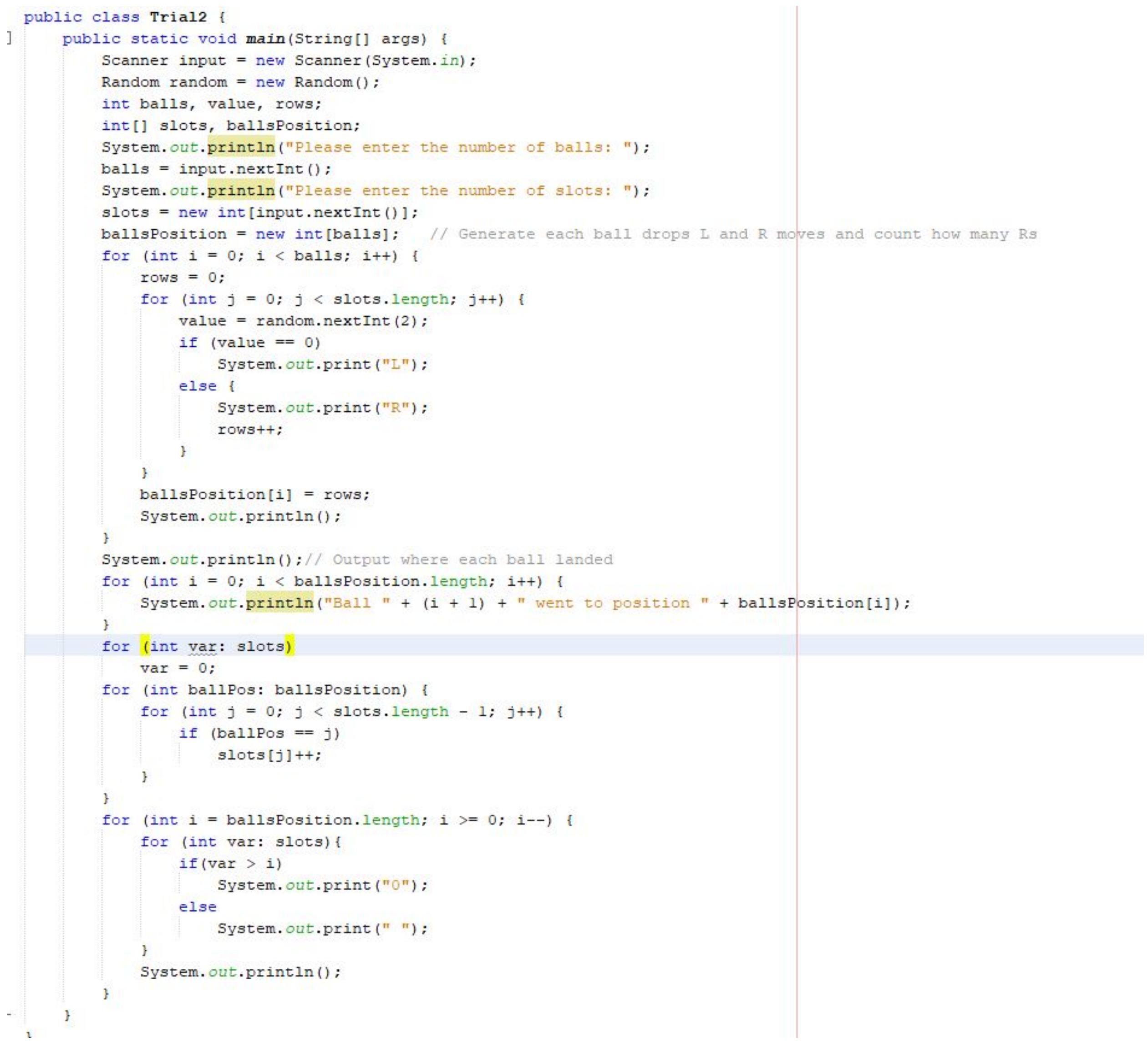
The GaltonBoardGUI class implements the Interactive Graphical User Interface. The constructor of this class initializes a GUI Frame. This frame holds the contentPane, which holds the UIPanel JPanel and the paintArea JComponent. The UIPanel has the user inputs in it, in which the user enters the values for number of balls and number of slots or bins. These are text inputs with default values set to 1000 and 11 respectively. UIPanel also has a Start button with an event Listener associated with it. Once the Start button is pressed, the action listener calls the **draw** function. The paintArea JComponent displays a bar chart based on the entered values

`private static void draw(int[] bins, int balls)`

The **draw()** method takes the input values (bins, balls) as arguments and uses the bins/balls ratio to determine height of the rectangles. The graphics2D object is used to draw the rectangles and the borders around them for each bin, and displays the number of balls in that particular bin. A for-each loop goes through each bin, and calls the graphics2D object to draw the bar associated with that bin.

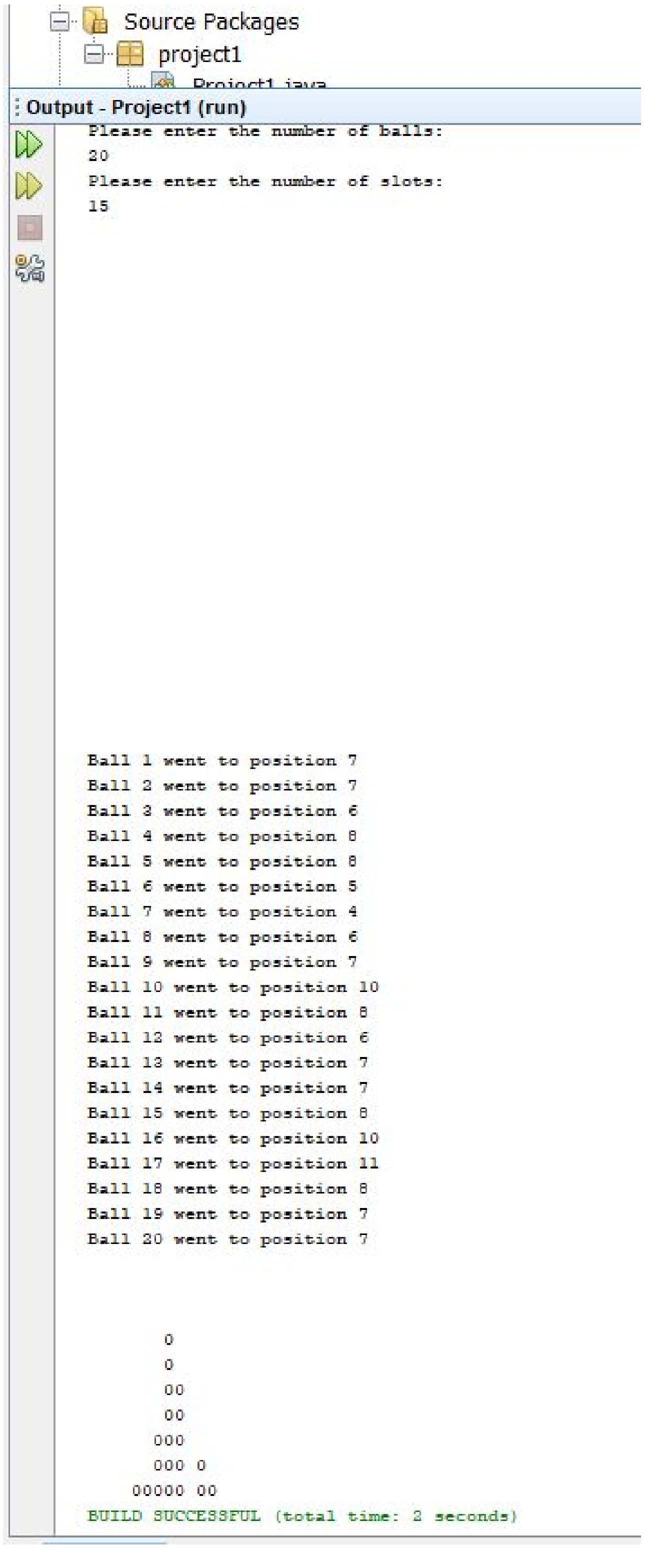
Appendix A: Early program development

Figure A1: Source Code



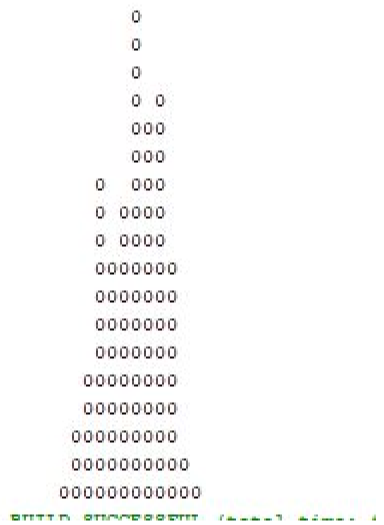
```java
public class Trial2 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        Random random = new Random();
        int balls, value, rows;
        int[] slots, ballsPosition;
        System.out.println("Please enter the number of balls: ");
        balls = input.nextInt();
        System.out.println("Please enter the number of slots: ");
        slots = new int[input.nextInt()];
        ballsPosition = new int[balls];   // Generate each ball drops L and R moves and count how many Rs
        for (int i = 0; i < balls; i++) {
            rows = 0;
            for (int j = 0; j < slots.length; j++) {
                value = random.nextInt(2);
                if (value == 0)
                    System.out.print("L");
                else {
                    System.out.print("R");
                    rows++;
                }
            }
            ballsPosition[i] = rows;
            System.out.println();
        }
        System.out.println();// Output where each ball landed
        for (int i = 0; i < ballsPosition.length; i++) {
            System.out.println("Ball " + (i + 1) + " went to position " + ballsPosition[i]);
        }
        for (int var: slots)
            var = 0;
        for (int ballPos: ballsPosition) {
            for (int j = 0; j < slots.length - 1; j++) {
                if (ballPos == j)
                    slots[j]++;
            }
        }
        for (int i = ballsPosition.length; i >= 0; i--) {
            for (int var: slots){
                if(var > i)
                    System.out.print("O");
                else
                    System.out.print(" ");
            }
            System.out.println();
        }
    }
}
```

Figure A2: Output



```
Please enter the number of balls:
20
Please enter the number of slots:
15
















Ball 1 went to position 7
Ball 2 went to position 7
Ball 3 went to position 6
Ball 4 went to position 8
Ball 5 went to position 8
Ball 6 went to position 5
Ball 7 went to position 4
Ball 8 went to position 6
Ball 9 went to position 7
Ball 10 went to position 10
Ball 11 went to position 8
Ball 12 went to position 6
Ball 13 went to position 7
Ball 14 went to position 7
Ball 15 went to position 8
Ball 16 went to position 10
Ball 17 went to position 11
Ball 18 went to position 8
Ball 19 went to position 7
Ball 20 went to position 7



       0
       0
       00
       00
      000
      000 0
    00000 00
BUILD SUCCESSFUL (total time: 2 seconds)
```

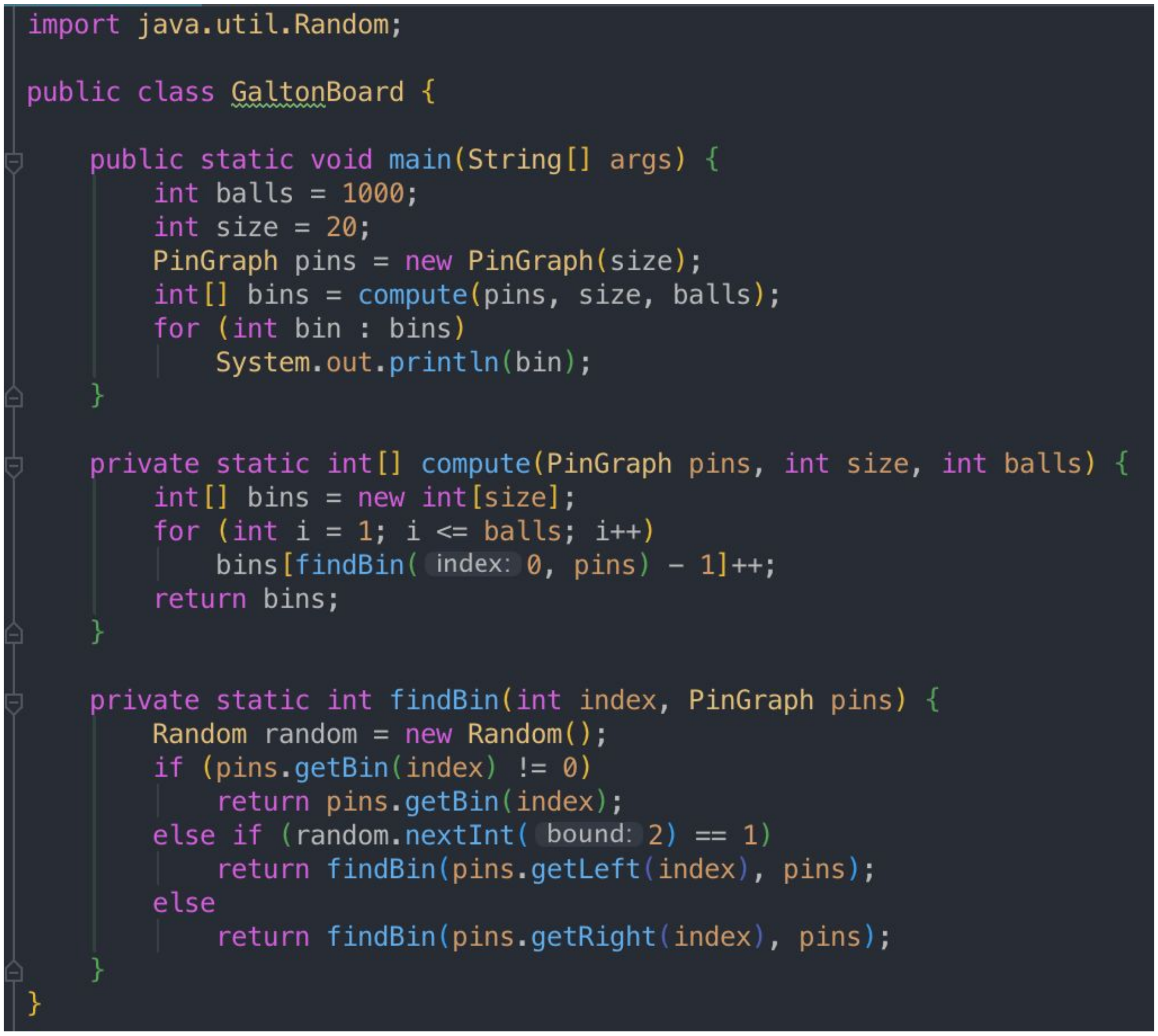
Figure A3: Demonstration of cluttered Output



```
          0
          0
          0
          0 0
          000
          000
       0  000
       0 0000
       0 0000
       0000000
       0000000
       0000000
       0000000
      00000000
      00000000
     000000000
     0000000000
    000000000000
BUILD SUCCESSFUL (total time:
```

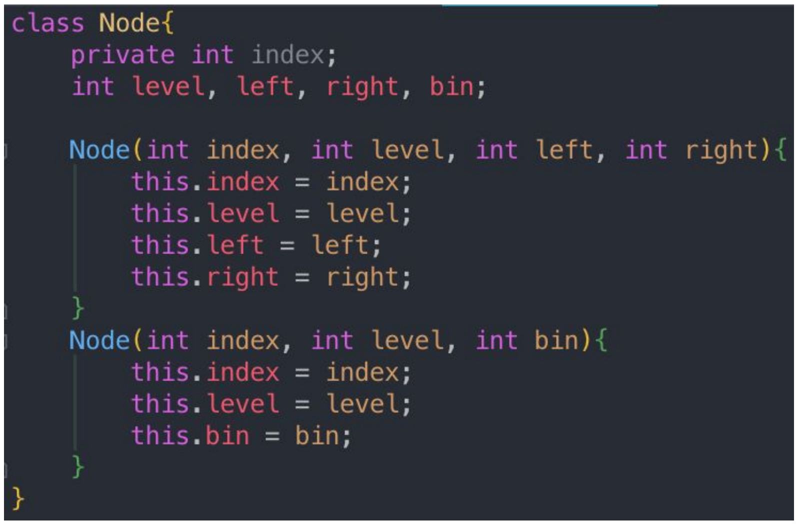
Appendix B: Final Code

Figure B1: GaltonBoard Class



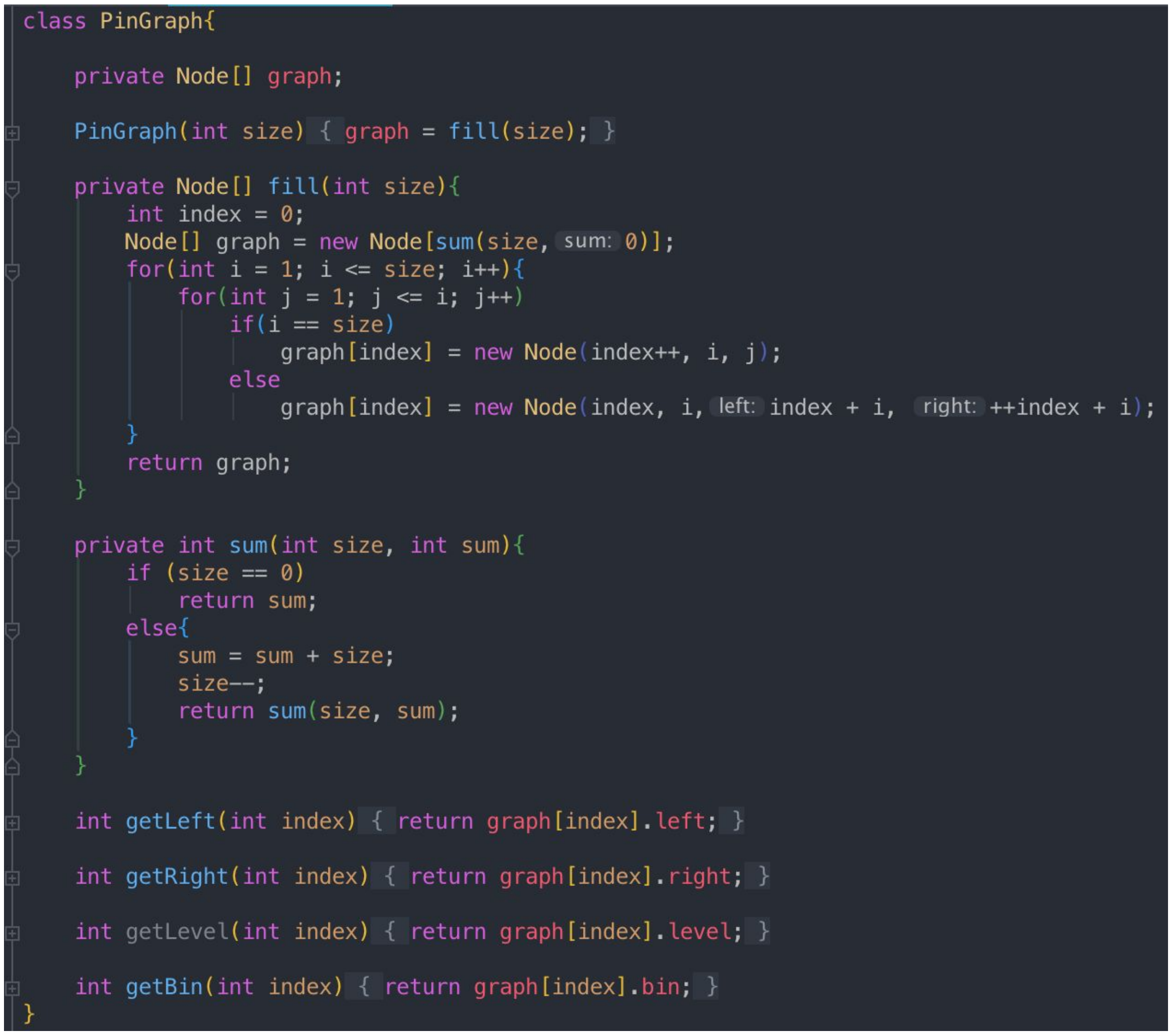
```java
import java.util.Random;

public class GaltonBoard {

    public static void main(String[] args) {
        int balls = 1000;
        int size = 20;
        PinGraph pins = new PinGraph(size);
        int[] bins = compute(pins, size, balls);
        for (int bin : bins)
            System.out.println(bin);
    }

    private static int[] compute(PinGraph pins, int size, int balls) {
        int[] bins = new int[size];
        for (int i = 1; i <= balls; i++)
            bins[findBin( index: 0, pins) - 1]++;
        return bins;
    }

    private static int findBin(int index, PinGraph pins) {
        Random random = new Random();
        if (pins.getBin(index) != 0)
            return pins.getBin(index);
        else if (random.nextInt( bound: 2) == 1)
            return findBin(pins.getLeft(index), pins);
        else
            return findBin(pins.getRight(index), pins);
    }
}
```

Figure B2: Node class



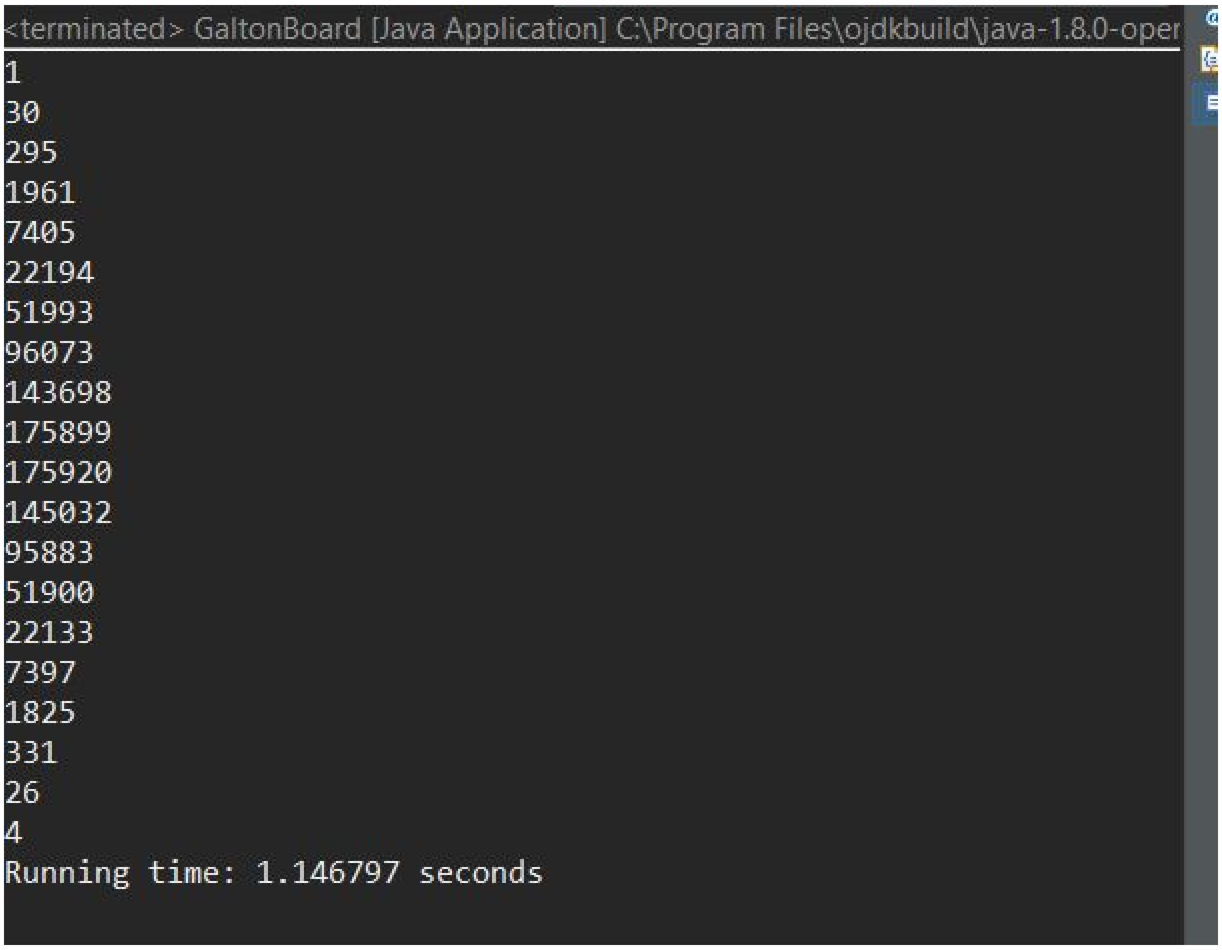
```java
class Node{
    private int index;
    int level, left, right, bin;

    Node(int index, int level, int left, int right){
        this.index = index;
        this.level = level;
        this.left = left;
        this.right = right;
    }
    Node(int index, int level, int bin){
        this.index = index;
        this.level = level;
        this.bin = bin;
    }
}
```

Figure B3: PinGraph Class



```java
class PinGraph{

    private Node[] graph;

    PinGraph(int size) { graph = fill(size); }

    private Node[] fill(int size){
        int index = 0;
        Node[] graph = new Node[sum(size, sum: 0)];
        for(int i = 1; i <= size; i++){
            for(int j = 1; j <= i; j++)
                if(i == size)
                    graph[index] = new Node(index++, i, j);
                else
                    graph[index] = new Node(index, i, left: index + i,  right: ++index + i);
        }
        return graph;
    }

    private int sum(int size, int sum){
        if (size == 0)
            return sum;
        else{
            sum = sum + size;
            size--;
            return sum(size, sum);
        }
    }

    int getLeft(int index) { return graph[index].left; }

    int getRight(int index) { return graph[index].right; }

    int getLevel(int index) { return graph[index].level; }

    int getBin(int index) { return graph[index].bin; }
}
```

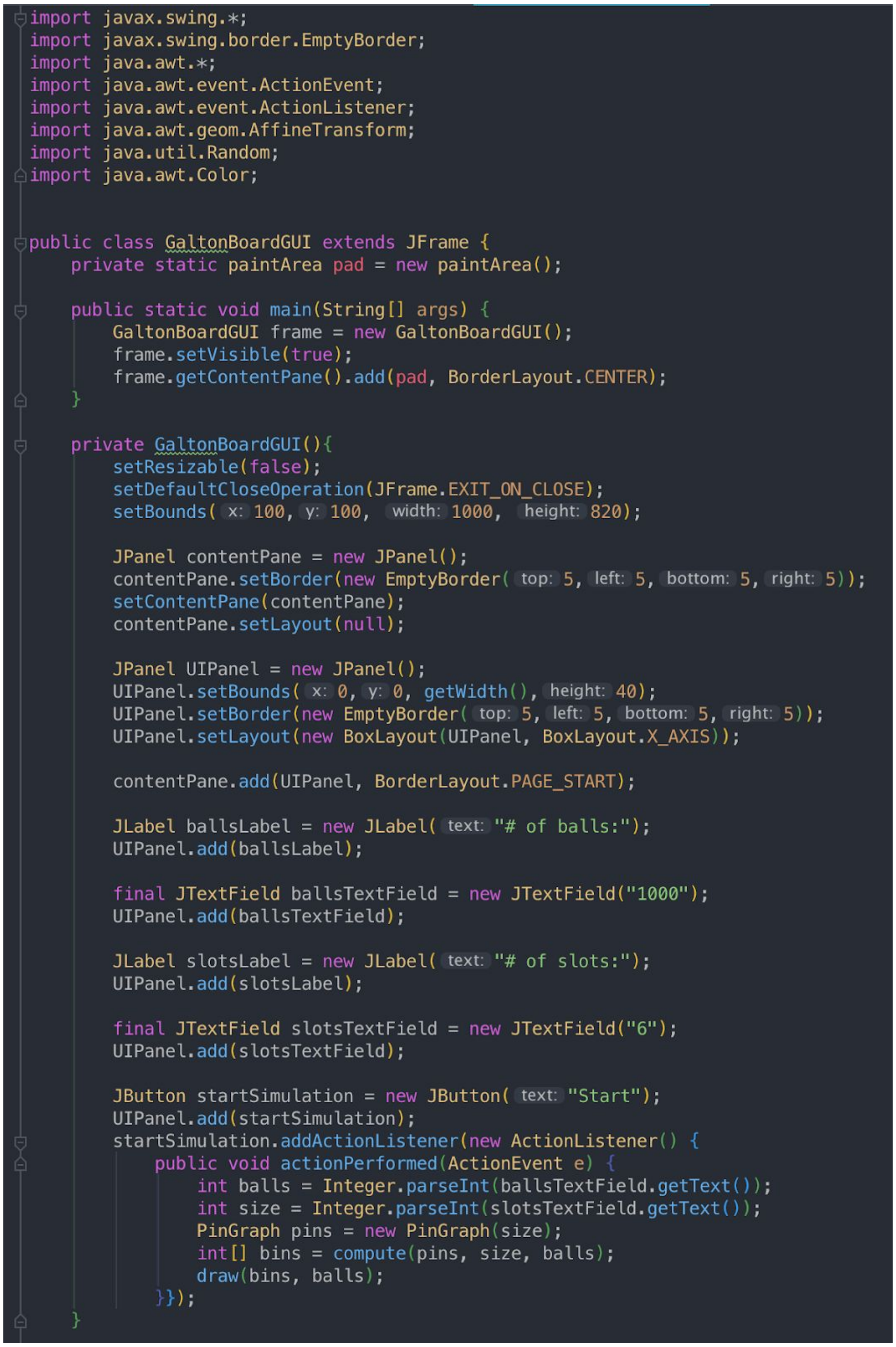
Figure B4: Output



```
<terminated> GaltonBoard [Java Application] C:\Program Files\ojdkbuild\java-1.8.0-oper
1
30
295
1961
7405
22194
51993
96073
143698
175899
175920
145032
95883
51900
22133
7397
1825
331
26
4
Running time: 1.146797 seconds
```

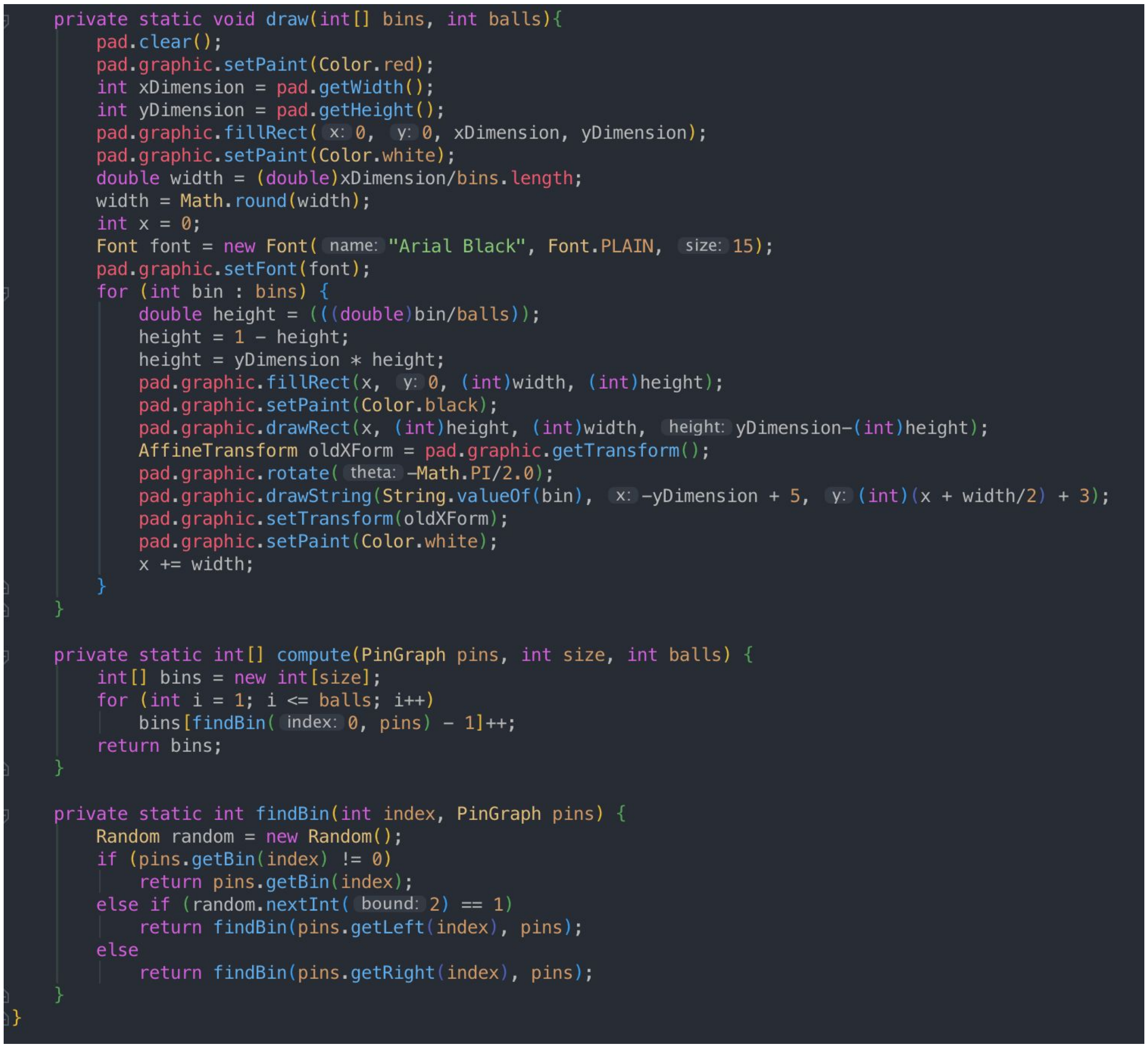
Appendix C: Graphic Interface

Figure C1: GaltonBoardGUI class, part 1



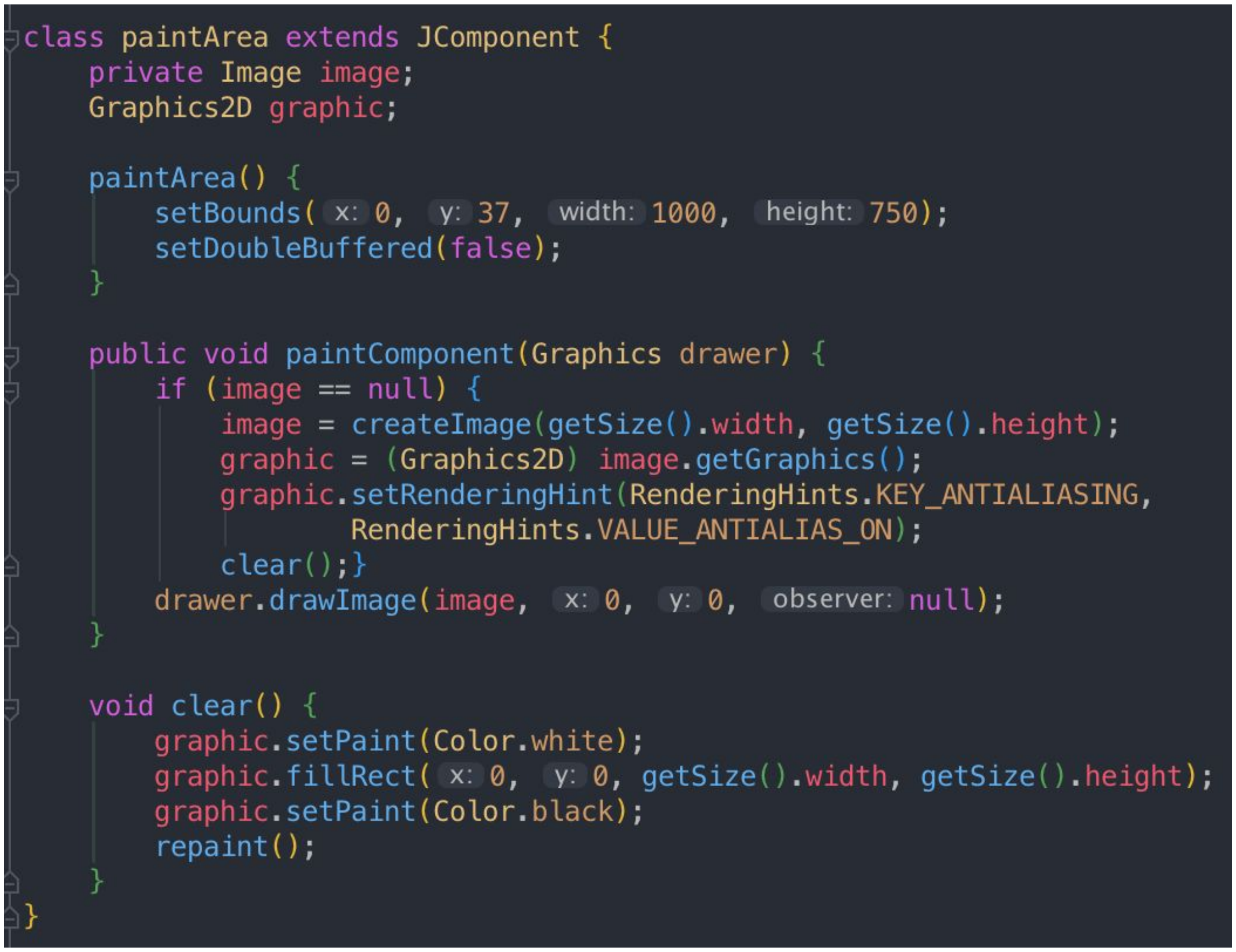
```java
import javax.swing.*;
import javax.swing.border.EmptyBorder;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.AffineTransform;
import java.util.Random;
import java.awt.Color;


public class GaltonBoardGUI extends JFrame {
    private static paintArea pad = new paintArea();

    public static void main(String[] args) {
        GaltonBoardGUI frame = new GaltonBoardGUI();
        frame.setVisible(true);
        frame.getContentPane().add(pad, BorderLayout.CENTER);
    }

    private GaltonBoardGUI(){
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds( x: 100, y: 100,  width: 1000,  height: 820);

        JPanel contentPane = new JPanel();
        contentPane.setBorder(new EmptyBorder( top: 5, left: 5, bottom: 5, right: 5));
        setContentPane(contentPane);
        contentPane.setLayout(null);

        JPanel UIPanel = new JPanel();
        UIPanel.setBounds( x: 0, y: 0, getWidth(), height: 40);
        UIPanel.setBorder(new EmptyBorder( top: 5, left: 5, bottom: 5, right: 5));
        UIPanel.setLayout(new BoxLayout(UIPanel, BoxLayout.X_AXIS));

        contentPane.add(UIPanel, BorderLayout.PAGE_START);

        JLabel ballsLabel = new JLabel( text: "# of balls:");
        UIPanel.add(ballsLabel);

        final JTextField ballsTextField = new JTextField("1000");
        UIPanel.add(ballsTextField);

        JLabel slotsLabel = new JLabel( text: "# of slots:");
        UIPanel.add(slotsLabel);

        final JTextField slotsTextField = new JTextField("6");
        UIPanel.add(slotsTextField);

        JButton startSimulation = new JButton( text: "Start");
        UIPanel.add(startSimulation);
        startSimulation.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int balls = Integer.parseInt(ballsTextField.getText());
                int size = Integer.parseInt(slotsTextField.getText());
                PinGraph pins = new PinGraph(size);
                int[] bins = compute(pins, size, balls);
                draw(bins, balls);
            }});
    }
```

Figure C2: GaltonBoardGUI class, part 2



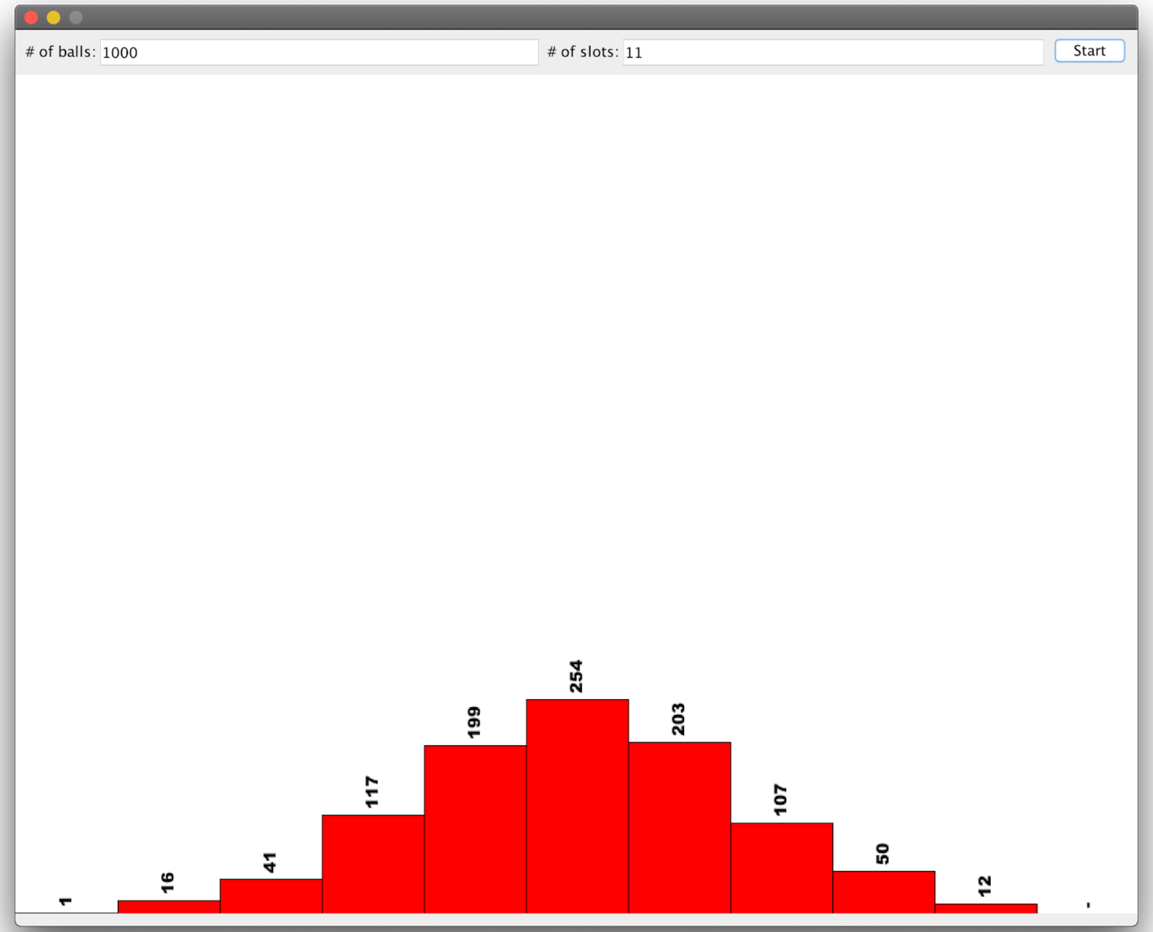
```
    private static void draw(int[] bins, int balls){
        pad.clear();
        pad.graphic.setPaint(Color.red);
        int xDimension = pad.getWidth();
        int yDimension = pad.getHeight();
        pad.graphic.fillRect( x: 0,  y: 0, xDimension, yDimension);
        pad.graphic.setPaint(Color.white);
        double width = (double)xDimension/bins.length;
        width = Math.round(width);
        int x = 0;
        Font font = new Font( name: "Arial Black", Font.PLAIN,  size: 15);
        pad.graphic.setFont(font);
        for (int bin : bins) {
            double height = (((double)bin/balls));
            height = 1 - height;
            height = yDimension * height;
            pad.graphic.fillRect(x,  y: 0, (int)width, (int)height);
            pad.graphic.setPaint(Color.black);
            pad.graphic.drawRect(x, (int)height, (int)width,  height: yDimension-(int)height);
            AffineTransform oldXForm = pad.graphic.getTransform();
            pad.graphic.rotate( theta: -Math.PI/2.0);
            pad.graphic.drawString(String.valueOf(bin),  x: -yDimension + 5,  y: (int)(x + width/2) + 3);
            pad.graphic.setTransform(oldXForm);
            pad.graphic.setPaint(Color.white);
            x += width;
        }
    }

    private static int[] compute(PinGraph pins, int size, int balls) {
        int[] bins = new int[size];
        for (int i = 1; i <= balls; i++)
            bins[findBin( index: 0, pins) - 1]++;
        return bins;
    }

    private static int findBin(int index, PinGraph pins) {
        Random random = new Random();
        if (pins.getBin(index) != 0)
            return pins.getBin(index);
        else if (random.nextInt( bound: 2) == 1)
            return findBin(pins.getLeft(index), pins);
        else
            return findBin(pins.getRight(index), pins);
    }
}
```

Figure C3: PaintArea class



```java
class paintArea extends JComponent {
    private Image image;
    Graphics2D graphic;

    paintArea() {
        setBounds( x: 0,  y: 37,  width: 1000,  height: 750);
        setDoubleBuffered(false);
    }

    public void paintComponent(Graphics drawer) {
        if (image == null) {
            image = createImage(getSize().width, getSize().height);
            graphic = (Graphics2D) image.getGraphics();
            graphic.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
            clear();}
        drawer.drawImage(image,  x: 0,  y: 0,  observer: null);
    }

    void clear() {
        graphic.setPaint(Color.white);
        graphic.fillRect( x: 0,  y: 0, getSize().width, getSize().height);
        graphic.setPaint(Color.black);
        repaint();
    }
}
```

Figure C4: Running Program



# of balls: 1000

# of slots: 11

Appendix D: Diagrams

Figure D1: Galton Tree Data Structure (graph) diagram

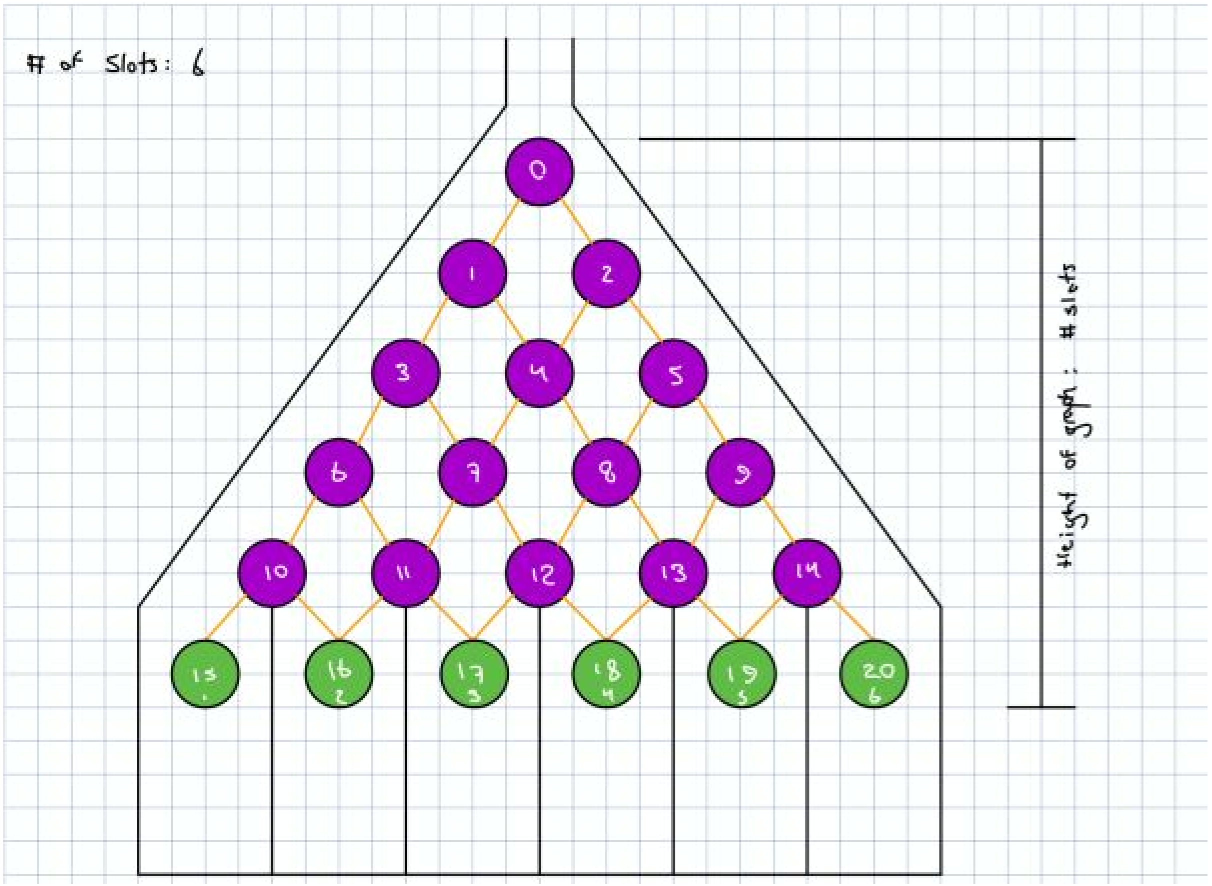


Figure D2: Pre Transverse order demonstration

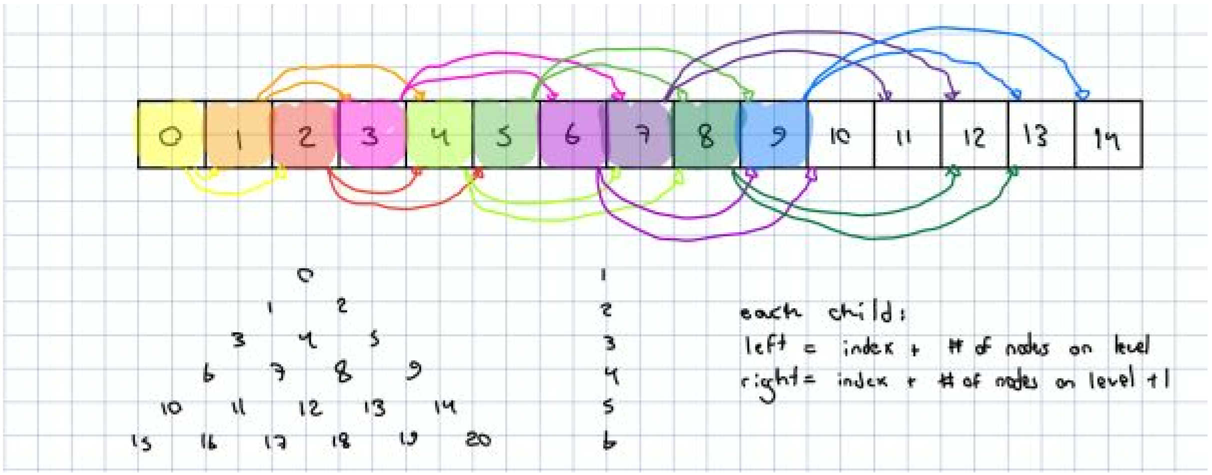
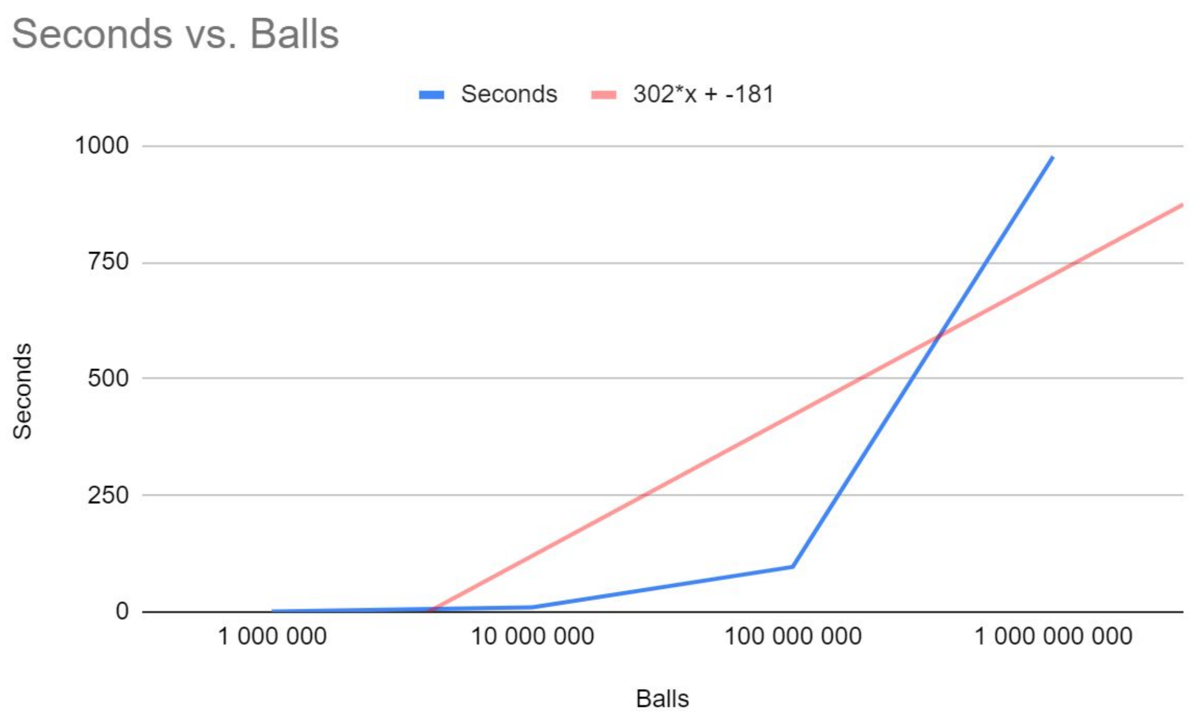


Figure D3: Graph Comparing the Runtime and Size of Array


Seconds vs. Balls
Seconds
302*x + -181
Seconds
Balls