Binary Sheep^{TM} - A Gamification Concept

CSCI338302 Final Project - Final Report

Erin Ballengee, Andrew Lim, Kamil Piskorz

Introduction

Binary heaps and tree manipulation are an often-used concept to find the k^{th} largest or smallest element in an array in a variety of applications. By creating a game revolving around the core concept of swapping nodes in trees, we create an alternative means of teaching this idea in a practical, educational, and entertaining manner as a supplement to traditional classroom education.

The game revolves around ordering a set of sheep that has fallen out of order. Players are instructed to order the sheep in the fashion that they would become a maximum binary heap. A graphic interface using a Java canvas allowed visual representation of sheep and other gameplay elements.

Design - Github Link

https://github.com/andrewplim98/AlgorithmsProjectCSCI3383

Design - Overview

Preliminary designs, as well as backend prototypes, implied that we would want to use Java in conjunction with HTML, using JSON and Javascript to bridge the two platforms. However, early on in the development of the actual game itself, we realized that using HTML and Javascript would incur a significant amount of difficulty in data conversion for little to no benefit. As a result, our game was built entirely in Java. The game consists of an interlinked structure comprised of an underlying heap management system, and a visual interface that users manipulate. This interface was geared to call certain backend functions such as swap functions to control the contents of the heap. Because heaps can be represented in array notation, this meant that the underlying heap controlled both whether the game was won/lost and what was displayed to the user. The interface would also read the contents of the underlying heap in addition to manipulating them, so as to determine what to display to the user.

Design - Backend Heap Control

Our design largely revolved around referencing and managing a single underlying array. Defined as sheepHeap, this array was initialized with 15 random integers between 1-100. This would serve as the basis for canvas display and heap value control. Because the array would require swap operations to arrange into a maximum binary heap, we implemented a counter for how many times the swap function was called. If the counter exceeded a certain predetermined number, the game would end in defeat. This number was set to 12 for testing purposes, with 12 being derived from the maximum number of swaps to build a heap from an array of 15 integers. We also created a utility function to check the array to see if it was in the form of a valid binary max heap after every swap - if it was, the game would end in victory.

Design - Game Display and Variable Management

The code in our project was segmented into two categories, each composed of its own Java classes that worked together with each other to achieve the final product. The first category, which was mentioned previously, was the binary heap which functioned as the underlying game mechanic. The second category was the game itself. It included all the files that interacted with each other so the game elements could be visualized as objects, updated, and rendered onto a Java JFrame. Our project folder included the following core files:

Window.java

The window class, along with our other files, was composed of the Java Abstract Window Toolkit library which extended Java's Canvas class. The purpose and functionality of this class were to set up a JFrame for which our 2D game could run in. We set up the window size, resizability, and visibility, and finished off by mapping our game to begin when the window was opened.

```java
import javax.swing.*;
import java.awt.Canvas;
import java.awt.Dimension;

public class Window extends Canvas{

  private static final long serialVersionUID = 421;

  public Window(int width, int height, String title, Game game){
    JFrame frame = new JFrame(title);

    frame.setPreferredSize(new Dimension(width, height));
    frame.setMaximumSize(new Dimension(width, height));
    frame.setMinimumSize(new Dimension(width, height));

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.setLocationRelativeTo(null);
    frame.add(game);
    frame.setVisible(true);
    game.start();
  }
}
```

Figure 1. Window.java

<u>Game.java</u>

The Game class became the foundation for the rest of our files. Essentially, through this file, we started the game and executed all other functions that existed within other classes. The main component of this file was the game loop located in the run() function. This was the “heartbeat” of the game, so to speak. Essentially, this function would loop 60 times in one second, update variables stored within the game objects, and render them onto the canvas. This game loop was configured to run in a single thread.

Each class has a tick and a render function which would be called from within the Game class, and would be called each time the run() function ran. The tick function would update the values of our game objects while the render function would draw them onto the canvas in the correct positions.

Lastly, the game class would cycle between game states and determine which functions should run during each game state. This will be explained in the next section.

```java
public class Game extends Canvas implements Runnable{

  private static final long serialVersionUID = 421;

  public static final int WIDTH = 1280, HEIGHT = 720;


  private Image gameScreen;
  private Thread thread;
  private boolean running = false;
  private Handler handler;
  private Menu menu;
  private Instructions instructions;
  private Lose lose;
  private Win win;

  public STATE gameState = STATE.Menu;

  public Game(){
    handler = new Handler();
    menu = new Menu(this, handler);

    instructions = new Instructions(this, handler);
    this.addMouseListener(menu);
    this.addMouseListener(instructions);

    lose = new Lose(this, handler);
    this.addMouseListener(lose);
    new Window(WIDTH, HEIGHT, "Binary Sheep", this);

    win = new Win(this, handler);
    this.addMouseListener(lose);

    // if(gameState == STATE.Game){
    //   //need to add 15 sheep to make a tree
    // }

  }
```

Figure 2. Game Class

```java
public void run(){
  long lastTime = System.nanoTime();
  double amountOfTicks = 60.0;
  double ns = 1000000000 / amountOfTicks;
  double delta = 0;
  long timer = System.currentTimeMillis();
  int frames = 0;

  while(running){
    long now = System.nanoTime();
    delta += (now - lastTime) / ns;
    lastTime = now;
    while(delta >= 1){
      tick();
      delta--;
    }
    if(running)
      render();
    frames++;

    if(System.currentTimeMillis() - timer > 1000){
      timer += 1000;
      // System.out.println("FPS: " + frames);
      frames = 0;
    }
  }
  stop();
}

private void tick(){


  if(gameState == STATE.Game){
    handler.tick();
  }else if(gameState == STATE.Menu){
    menu.tick();
  }else if(gameState == STATE.Lose){
    lose.tick();
  }else if(gameState == STATE.Win){
    win.tick();
  }
}
```

Figure 3. Game Loop and Tick Function

<u>STATE.java</u>

The STATE class is a helper class that lists all the game's states. All the states within the game were Game, Menu, Instructions, Win, and Lose. Whenever the game state was changed within the Game class, Menu class, or the Instructions class, the corresponding functions would be called and all objects appearing in that particular game state would be rendered and ticked. An example of this would be when the Game state was changed to the Instruction state, the instructions screen would be rendered and ticked while all other state objects would cease to be rendered and ticked. Figure 3 illustrates what would be ticked if the state was changed.

```java
public enum STATE{
  Menu,
  Instructions,
  Lose,
  Win,
  Game
};
```

Figure 4. STATE.java

<u>Handler.java</u>

The Handler class stores all playing objects within a linked list. It is responsible for looping through all these objects, ticking each one, and rendering each one accordingly to the display on each loop of the game's run() function. By "playing" objects, we are referring to objects that exist only when the game is in its game state, so this means only our sheep objects. If we were to later add another object to the game such as a wolf or a farmer that only exists in the game state, they would be stored in the handler's linked list.

```java
//loops through all objects in the game, renders them, and updates them to screen
public class Handler{

  LinkedList<GameObject> object = new LinkedList<GameObject>();

  //tick through all of our game objects
  public void tick(){
    for(int i = 0; i < object.size(); i++){
      GameObject tempObject = object.get(i);

      tempObject.tick();
    }
  }

  //render all of our game objects
  public void render(Graphics g){
    for(int i = 0; i < object.size(); i++){
      GameObject tempObject = object.get(i);

      tempObject.render(g);
    }
  }

  //handles adding and removing object from list
  public void addObject(GameObject object){
    this.object.add(object);
  }
  public void removeObject(GameObject object){
    this.object.remove(object);
  }
}
```

Figure 5. Handler.java

GameObject.java

This class contains a series of function calls pertaining to playing objects to affect and alter various parameters such as position, vertical/horizontal velocity, and ID. This is the superclass for all of our playing objects, which later get more specified parameters in their own classes. The perk of having a superclass for all playing objects is the benefit of eventual scalability. Each playing object will contain the parameters in this superclass so we do not need to repeat them for every other subclass.

```java
public abstract class GameObject{

  protected int x, y;
  protected ID id;
  protected int index;
  protected int value;
  protected int velX, velY;

  //game object constructor --> used for all game objects
  public GameObject(int x, int y, ID id, int index, int value){
    this.x = x;
    this.y = y;
    this.id = id;
    this.index = index;
    this.value = value;
  }

  public abstract void tick();
  public abstract void render(Graphics g);

  public void setX(int x){
    this.x = x;
  }
  public void setY(int y){
    this.y = y;
  }
  public int getX(){
    return x;
  }
  public int getY(){
    return y;
  }
  public void setVelX(int velX){
    this.velX = velX;
  }
  public void setVelY(int velY){
    this.velY = velY;
  }
  public int getVelX(){
    return velX;
  }
  public int getVelY(){
    return velY;
```

Figure 6. Portion of GameObject Superclass

<u>Sheep.java</u>

This file contains the data of sheep objects, including position, displayed value, and underlying sprite.

```java
public class Sheep extends GameObject{

  private Image sheepie;
  private Font font;

  public Sheep(int x, int y, ID id, int index, int value){
    super(x, y, id, index, value);

    velX = 1;
    velY = 0;
  }

  public void tick(){
    // x += velX;
    value = Menu.sheepHeap[index];
    // value = value;
  }

  public void render(Graphics g){
    try{
      sheepie = ImageIO.read(getClass().getResourceAsStream("/Images/sheepie.png"));
    } catch(IOException ex){
      ex.printStackTrace();
    }
    g.drawImage(sheepie, x, y, 100,100, null);
    g.setFont(new Font("Corbel", Font.PLAIN, 18));

    g.setColor(Color.red);
    g.drawString(Integer.toString(index), x + 48, y + 100);

    g.setFont(new Font("Corbel", Font.PLAIN, 24));
    g.setColor(Color.black);
    if(value > 9){
      g.drawString(Integer.toString(value), x + 40 , y + 60);
    }else{
      g.drawString(Integer.toString(value), x + 45 , y + 60);
    }
    // g.setColor(Color.white);
    // g.fillRect(x,y,32,32);
  }
```

Figure 8. Sheep.java

<u>ID.java</u>

This class designates more specific parameters, such as which class an object will belong to. This allows the GameObject superclass to recognize which subclasses it will be dealing with so that the correct object's parameters can be accessed.

```java
public enum ID{
  Sheep();
}
```

Figure 9. ID.java

<u>Menu.java</u>

Initializes canvas and adds sheep objects to the linked list when the game is started. This file also initializes the sheepHeap, which serves as the underlying logical heap from which all values were drawn from, as well as initializes the swap function that is used to manage the heap. Because the menu navigated with the click of a mouse, we needed to extend the java class MouseAdapter.

```java
public class Menu extends MouseAdapter{

  private Win win;
  private Lose lose;
  public TextBox t;
  public static int[] sheepHeap;
  private Image menuScreen;
  private Game game;
  private Instructions instructions;
  public Handler handler;
  public static final int WIDTH = 1280, HEIGHT = 720;

  public Menu(Game game, Handler handler){
    this.game = game;
    this.handler = handler;
  }

  public static void array_swap(int[] arr, int index1, int index2){
    int temp = arr[index1];
    arr[index1] = arr[index2];
    arr[index2] = temp;
  }

  public int[] createHeap(){
    int[] sheepHeap = new int[15];
    int arrlen = sheepHeap.length;
    //Generates 15 Random Numbers in the range 1 -100
    for(int i = 0; i < arrlen; i++) {
      sheepHeap[i] = (int)(Math.random()*100 + 1);
    }
    return sheepHeap;
  }
  public void mousePressed(MouseEvent e){
    int mx = e.getX();
    int my = e.getY();

    //if mouse is over play button
    if(game.gameState == STATE.Menu && mouseOver(mx, my, 460, 240, 360, 105)){
      game.gameState = STATE.Game;


      sheepHeap = createHeap();

```

```java
      TextBox t = new TextBox(sheepHeap, handler.object, game, handler);
      // for(int i = 0; i<15;i++){
      //   System.out.print(sheepHeap[i] + " ");
      //}
    }else if(game.gameState == STATE.Menu && mouseOver(mx, my, 460, 350, 360, 105)){
      game.gameState = STATE.Instructions;
    }
  }

  public void mouseReleased(MouseEvent e){

  }

  //checks if mouse is hovering over a button
  private boolean mouseOver(int mx, int my, int x, int y, int width, int height){
    if(mx > x && mx < x + width){
      if(my > y && my < y + height){
        return true;
      }else return false;
    }else return false;
  }

  public void tick(){

  }

  public void render(Graphics g){
    try{
      menuScreen = ImageIO.read(getClass().getResourceAsStream("/Images/menuscreen.png"));
    } catch(IOException ex){
      ex.printStackTrace();
    }
    g.drawImage(menuScreen, 0, 0, null);
    // g.setColor(Color.white);
    // g.drawRect(460,350,360,105);
  }
```

Figure 10. Menu Class Functions

Figure 11. Menu Mouse Click & Rendering

Instructions.java

The Instructions gamestate contains an image that displays gameplay instructions, as well as a button to return to the previous gamestate (main menu).

```java
public class Instructions extends MouseAdapter{


  private Image instructionScreen;
  private Game game;
  private Menu menu;
  private Handler handler;
  public static final int WIDTH = 1280, HEIGHT = 720;

  public Instructions(Game game, Handler handler){
    this.game = game;
    this.handler = handler;
  }

  public void mousePressed(MouseEvent e){
    int mx = e.getX();
    int my = e.getY();

    //return button
    if(game.gameState == STATE.Instructions && mouseOver(mx, my, 403,550,475,115)){
      game.gameState = STATE.Menu;
    }
  }

  public void mouseReleased(MouseEvent e){

  }

  //checks if mouse is hovering over a button
  private boolean mouseOver(int mx, int my, int x, int y, int width, int height){
    if(mx > x && mx < x + width){
      if(my > y && my < y + height){
        return true;
      }else return false;
    }else return false;
  }

  public void tick(){

  }

  public void render(Graphics g){
    try{
      instructionScreen = ImageIO.read(getClass().getResourceAsStream("/Images/instructionscreen.png"));
    } catch(IOException ex){
      ex.printStackTrace();
    }
    g.drawImage(instructionScreen, 0, 0, null);
```

Figure 12. Instructions.java

TextBox.java

In order to swap two elements, a textbox is created that has two spots that the user can enter array indices into and click ‘Enter’, which will execute swap operations on the entered indices of the maxheap. The sheep are updated and rendered based on the current indices of the heap’s values. This file also checks the heap to see if it is a maximum binary heap, and increments the swap operation counter.

TextBox included the functions needed to play sounds at points in the code. Sheep.wav, the sound of a sheep bleating, was played after every single swap execution. Nelson.wav, the sound of The Simpsons character Nelson laughing, was played when the swap operation call counter exceeded a preset benchmark, signaling defeat. This class also contained calls to Game.java to change the gamestate to either winning or losing depending on the outcome.

```java
    public TextBox(int[] sheepHeap, LinkedList<GameObject> object, Game game, Handler handler){

      this.game = game;
      this.handler = handler;

      setTitle("Binary Sheep");
      setVisible(true);
      setSize(400,200);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      File SheepBaa = new File("Sheep.wav");
      File Nelson = new File("Nelson.wav");

      jp.add(jt1);
      jt1.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
          jt2.requestFocus();
        }
      });

      jp.add(jt2);
      jt2.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
          String input1 = jt1.getText();
          String input2 = jt2.getText();

          alpha = Integer.parseInt(input1);
          beta = Integer.parseInt(input2);

          System.out.println("Unsorted: \n");
          for (int i=0; i<15; i++){
            System.out.print(sheepHeap[i] + " ");
          }

          // charlie = alpha + beta;
          // array_swap(sheepHeap, object.get(alpha).index, object.get(beta).index, Menu.handler)
          array_swap(sheepHeap, object.get(alpha).index, object.get(beta).index);
          System.out.println("alpha");
          PlaySound(SheepBaa);

          if(isHeap(sheepHeap, 0, 14) == true){
            System.out.println("A winrar is you!");
            game.gameState = STATE.Win;
          }

          swapcallcount += 1;
          if(swapcallcount > swapcallmax){
            System.out.println("You Lose");
            game.gameState = STATE.Lose;
            PlaySound(Nelson);
```

Figure 13. Textbox 1 & 2 Functions

```java
      jb.addActionListener(new ActionListener(){

        public void actionPerformed(ActionEvent e){
          String input1 = jt1.getText();
          String input2 = jt2.getText();

          alpha = Integer.parseInt(input1);
          beta = Integer.parseInt(input2);
          charlie = alpha + beta;

          System.out.println("Preswap: \n");
          for (int i=0; i<15; i++){
            System.out.print(sheepHeap[i] + " ");
          }
          array_swap(sheepHeap, object.get(alpha).index, object.get(be
          System.out.println("Beta");
          PlaySound(SheepBaa);

          swapcallcount += 1;
          System.out.println();
          System.out.println(swapcallcount);
          if(swapcallcount > swapcallmax){
            System.out.println("You Lose");
            game.gameState = STATE.Lose;
            PlaySound(Nelson);
          }
          if(isHeap(sheepHeap, 0, 14) == true){
            System.out.println("A winrar is you!");
            game.gameState = STATE.Win;
          }
          System.out.println("Postswap: \n");
          for (int i=0; i<15; i++){
            System.out.print(sheepHeap[i] + " ");
          }
        //  jl.setText(String.valueOf(charlie));
        }
      });

      jp.add(jb);
      //jp.add(jl);
      add(jp);

  }
  static void PlaySound(File Sound) {
```

Figure 14. Submit Button Functions

All sounds were taken from brief Youtube clips converted into .wav files.

Win.java

The victory screen is called upon a win state being reached. It tells the user “YOU WIN” in large letters in the center of the screen and contains a button to return to the main menu.

Lose.java

The defeat screen, which is called upon the lose state being reached. It tells the user "YOU LOSE" in large letters in the center of the screen and plays a menacing noise. It also contains a button to return to the main menu.

Did you meet your milestone stated in the initial report? If not, what made you fail to achieve it?

Our primary milestone was to have a fully functional game with a coherent logical system, working and interactable user elements, and some level of aesthetic design, including but not limited to sprites, sounds, and backgrounds. This milestone was met to our satisfaction.

While we accomplished our primary milestone in creating a functional game, several minor milestones were cut during development due to time constraints or project changes. In particular, a feature to configure heap size was scrapped due to how it would have impacted the display end of the program, namely having to dynamically place sheep on the canvas depending on the number of elements. This was deemed too difficult to accomplish in an elegant fashion without taking too much time from other aspects. To this end, our sheep count was hard-coded and manually placed with 15 sheep, and our logical backend was adjusted accordingly to accommodate the backend. We also did not include displays such as timers and score counters due to time constraints. Much of our focus was on having the sheep themselves display correctly, as they were the focus of user interaction.

What did you learn from this project?

From this project, we learned about the ways that Java can be utilized to create a game application. We discovered the manner in which the graphics package can be used in tandem with a basic underlying Java framework to create a visual and interactive game format. By creating this game, we furthered our knowledge about how to combine various files into one project and then having them work together. While it was difficult to figure out at first, we learned about how to incorporate visual game elements as objects, and have them be rendered to update based on any swaps or changes that have occurred.

Do you agree to share your code/report with other students, say in other classes or a future Algorithms class?

Yes, we do agree to share our code and report with other students in other classes or in future Algorithms classes. It was very helpful for us in choosing our project to be able to look at past students' work and see what they had accomplished. We think that, while it could use some improvements, our game would be helpful in having students practice examples with binary heaps, and they should be able to have any resource they need to master this topic, as it can be hard for some.

Do you have any suggestions for the final project? E.g. more help? More resources? Format of poster session? Workload?...etc. Please be specific.

We think that the final project was structured well overall, but there could be more help in the beginning to aid students in choosing their topic, either by providing more resources or examples to students. It may be because we had multiple professors, but we think that if there was more class time talking about the various project ideas, it would have been helpful in sparking ideas and creativity.