Quadtrees in Image Processing

Sarah Hamdan, 202243500, F-06, 11th May 2024.

Introduction

In this project, 2 main classes were created, ImgQuadTree and ImgQuadTreeFileCreator. The QTNode class was also created for the nodes. The first class takes the pre-order representation of a quad tree from a file and creates a quad tree using it. This quad tree represents a compressed image. In this class, a 2D array can be created which represents the uncompressed version of the image. In the second class, we are given a file which includes the 2D array representation of an image, meaning it is the uncompressed version, and this class creates a file which includes the compressed version. This file can be later used by the ImgQuadTreeDisplayer class to generate images.

In this report, a general explanation of how each class and method works is given, but most of the focus is on explaining the thought process behind creating each of them, since this is what the project description said was required. You can find more details on how each method works written in the comments of the java files. Problem-solving strategies and challenges in working on this project are detailed in this report, although some sections include more details than others, since certain things have a simpler and more straightforward design that others.

ImgQuadTree Class

QTNode Class

```
161    /*
162     The QTNode class is how each ImgQuadTree node is created. Each node has 5 attributes:
163     4 children (which are also QTNodes) + 1 intensity value.
164     */
165    private class QTNode {
166
167        //instance vars:
168        private QTNode child_1;
169        private QTNode child_2;
170        private QTNode child_3;
171        private QTNode child_4;
172        private int intensity;
173
174        //constructor:
175        QTNode(){
176
177            this.intensity = -1; //The intensity is initially == -1 and can be changed later.
178            this.child_1 = null; //Each child is initially set to null.
179            this.child_2 = null;
180            this.child_3 = null;
181            this.child_4 = null;
182
183        }
184
185    }
```

This inner class creates the ImgQuadTree's nodes. Each node has 4 children and an intensity. The constructor of this class simply gives all the instance variables default values, which can be accessed and changed later.

Constructor & instance variables

```
8   public class ImgQuadTree {
9
10      //instance vars:
11      private QTNode root = new QTNode();
12      private int nodes_num = 1; //initially we only have the root
13      private int leaves_num = 0;
14
15      //constructor:
16      public ImgQuadTree(String filename){
17
18          File image = new File(filename);
19          try{
20              Scanner input = new Scanner(image);
21              this.root.intensity = input.nextInt();
22              add(root, input); //The constructor creates the tree through the add method
23          }
24          catch(FileNotFoundException ex){
25              System.out.println(ex.getMessage());
26          }
27
28      }
```

The ImgQuadTree has 3 instance variables: root, nodes_num, and leaves_num. root is a QTNode. nodes_num and leaves_num were added for efficiency. These two variables are incremented as the ImgQuadTree is created. Later, they can be retrieved easily with getters. This is faster than having to traverse the ImgQuadTree each time to fetch the number of nodes and leaves.

The constructor of this class creates a File object, then a Scanner object which reads from this file. A FileNotFoundException is thrown in case this file is not found, and an error message is printed to the terminal. Otherwise, the first int in the file is first assigned as the intensity of the root. Next, the add method is called, and this is where the rest of the ImgQuadTree is built.

add Method

```
30      //methods:
31
32      /*The add method takes a node "root" and input from the file, if the nextInt was != -1,
33      this means the int is a child of the node (this only applies to the first four integers,
34      past these 4, the next int is the child of the node's parent). If the nextInt == -1, this method
35      goes recursively with this node as the parent node of the following integers. With each parent
36      node this method checks for 4 children only, then it stops, this is basically the condition that
37      allows the recursion to end.
38      Note: the number itself is not the child but rather the intensity of the child, therefore
39      a node is created each time and is given this intensity.
40      */
41      public void add(QTNode root, Scanner input){
42
43          int num_1 = input.nextInt();
44          nodes_num ++; //each number from the input becomes a node, so this counter is incremented.
45          if(num_1 == -1){
46              QTNode c = new QTNode();
47              root.child_1 = c;
48              root.child_1.intensity = num_1;
49              add(root.child_1, input);
50          }
51          else{
52              leaves_num ++; //if it's != -1 then it is a leaf, so this counter is incremented.
53              QTNode c = new QTNode();
54              root.child_1 = c;
55              root.child_1.intensity = num_1;
56          }
57
58          int num_2 = input.nextInt();
59          nodes_num ++;
60          if(num_2 == -1){
61              QTNode c = new QTNode();
62              root.child_2 = c;
63              root.child_2.intensity = num_2;
64              add(root.child_2, input);
65          }
66          else{
67              leaves_num ++;
68              QTNode c = new QTNode();
69              root.child_2 = c;
70              root.child_2.intensity = num_2;
71          }
72
73          int num_3 = input.nextInt();
74          nodes_num ++;
75          if(num_3 == -1){
76              QTNode c = new QTNode();
77              root.child_3 = c;
78              root.child_3.intensity = num_3;
79              add(root.child_3, input);
80          }
81          else{
82              leaves_num ++;
83              QTNode c = new QTNode();
84              root.child_3 = c;
85              root.child_3.intensity = num_3;
86          }
87
88          int num_4 = input.nextInt();
89          nodes_num ++;
90          if(num_4 == -1){
91              QTNode c = new QTNode();
92              root.child_4 = c;
93              root.child_4.intensity = num_4;
94              add(root.child_4, input);
95          }
96          else{
97              leaves_num ++;
98              QTNode c = new QTNode();
99              root.child_4 = c;
100             root.child_4.intensity = num_4;
101         }
102     }
```

Problem-solving strategy

Since the file includes the pre-order traversal of an ImgQuadTree, this means that the first node we find is the root, the next one is its first child. If the first child's intensity is == -1, that means that the following node is going to be its child, if not, then the following node is the second child of the root, this goes for the four children of the root node, and the same goes for the children if their value is == -1, since they would also have four children. This is why recursion is used, when a node has an intensity of -1, it is sent to the add method as the "root" of its subtree.

Note that each time a value is read from the file, the nodes_num variable is incremented, and each time this value is != -1, leaves_num is incremented.

Challenges

It was difficult at first to figure out how to stop the recursion, I drew sketches of the tree and understood how the pre-order traversal of it would work. I had first thought of creating a loop which repeats four times, once for each child, but I ended up writing code for each of the children without a loop. At one point I had an issue since I wrote code for all four children and I also had the loop, so the method was being called more times than it needed to be and that caused an error, but then after debugging I removed the loop and the method worked.

getNumNodes & getNumLeaves Methods

```
104     /*
105     since nodes_num and leaves_num are instance variables incremented in the add method, their getters
106     only have to return them. There is no need for traversal here.
107     */
108     public int getNumNodes() {
109         return nodes_num;
110     }
111
112     public int getNumLeaves() {
113         return leaves_num;
114     }
```

These two getters simply return the value of the instance variables nodes_num and leaves_num, which were incremented by the add method previously.

Problem-solving strategy

I first thought of traversing through the ImgQuadTree to get the number of nodes and leaves, but I realized that this would be redundant since these values can be found at once when the ImgQuadTree is created. Making nodes_num and leaves_num instance variables makes it so there is no need for traversal through the ImgQuadTree each time their values need to be fetched. If a delete method was to be added in the future for example, these instance variables could also be decremented; so finding the number of nodes and leaves is simple and efficient using this method.

getImageArray & its helper Method

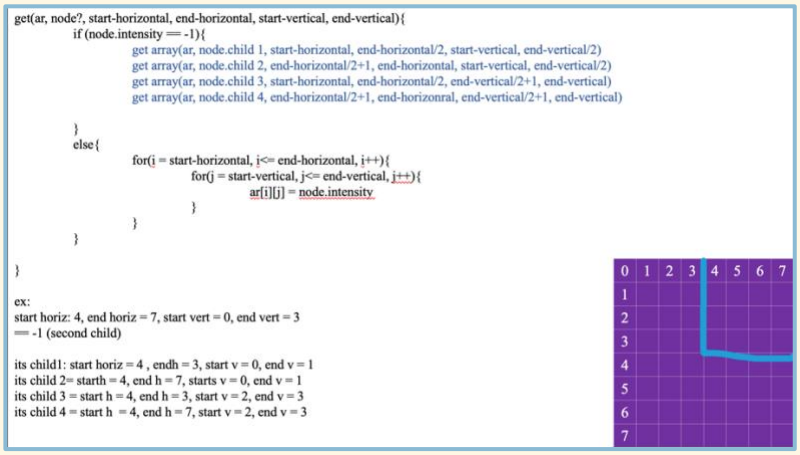
```java
//the getImageArray method:
/*
this getImageArray method basically sets up the recursion, most of the work is done in
its helper method...
*/
public int[][] getImageArray(){
    int dimention = 256; //dimention can be changed depending on the number of pixels.
    int[][] image_array = new int[dimention][dimention];
    getImageArray(image_array, this.root, start_horiz:0, dimention-1, start_vert:0, dimention-1);
    return image_array;
}

//its helper method:
/*
Once we know we hit a leaf and we know which part of the array it should be in, filling the array
is simply done in a nested for loop. The trick is in changing the array's indices which we are
searching for leaves in. We start with the root node and each of its children represents a quarter
of the array, and the indices are changed accordingly. The same adjustment of the indices is done
with each child node recursively if its intensity == -1.
*/
public void getImageArray(int[][] array, QTNode node, int start_horiz, int end_horiz, int start_vert, int end_vert){
    if(node.intensity == -1){
        getImageArray(array, node.child_1, start_horiz, (start_horiz + end_horiz)/2, start_vert,(start_vert+end_vert)/2);
        getImageArray(array, node.child_2, (start_horiz + end_horiz)/2+1, end_horiz, start_vert, (start_vert+end_vert)/2);
        getImageArray(array, node.child_3, start_horiz, (start_horiz + end_horiz)/2, (start_vert+end_vert)/2+1, end_vert);
        getImageArray(array, node.child_4, (start_horiz + end_horiz)/2+1, end_horiz, (start_vert+end_vert)/2+1, end_vert);
    }
    else{

        for(int i = start_horiz; i <= end_horiz; i++){
            for(int j = start_vert; j <= end_vert; j++){
                array[j][i] = node.intensity;
            }
        }
    }

}
```

Problem-solving strategy & challenges



The recursion in this method relies on changing the indices which are used to fill the array in the nested for loop once a leaf is found (AKA if the node.intensity != -1). It took me a while to figure out how to change these indices depending on each quadrant, but using an example (as seen in the screenshot above) and seeing what the starting and ending index for each quadrant of the array would be helped me find the pattern eventually. A friend helped me realize that I need to send the horizontal and vertical starting and ending indices separately to the method. Of course, the whole array starts with the root node whose intensity is == -1, and then the first quadrant is its first child, the second quadrant is its second child, and so on... Each one of these children could then be split up into four quadrants as well if their intensity value was == -1 too, and this is why recursion was used. As done in the add method, the getImageArray method is called recursively 4 times in each method call, once for each child. If a node's intensity

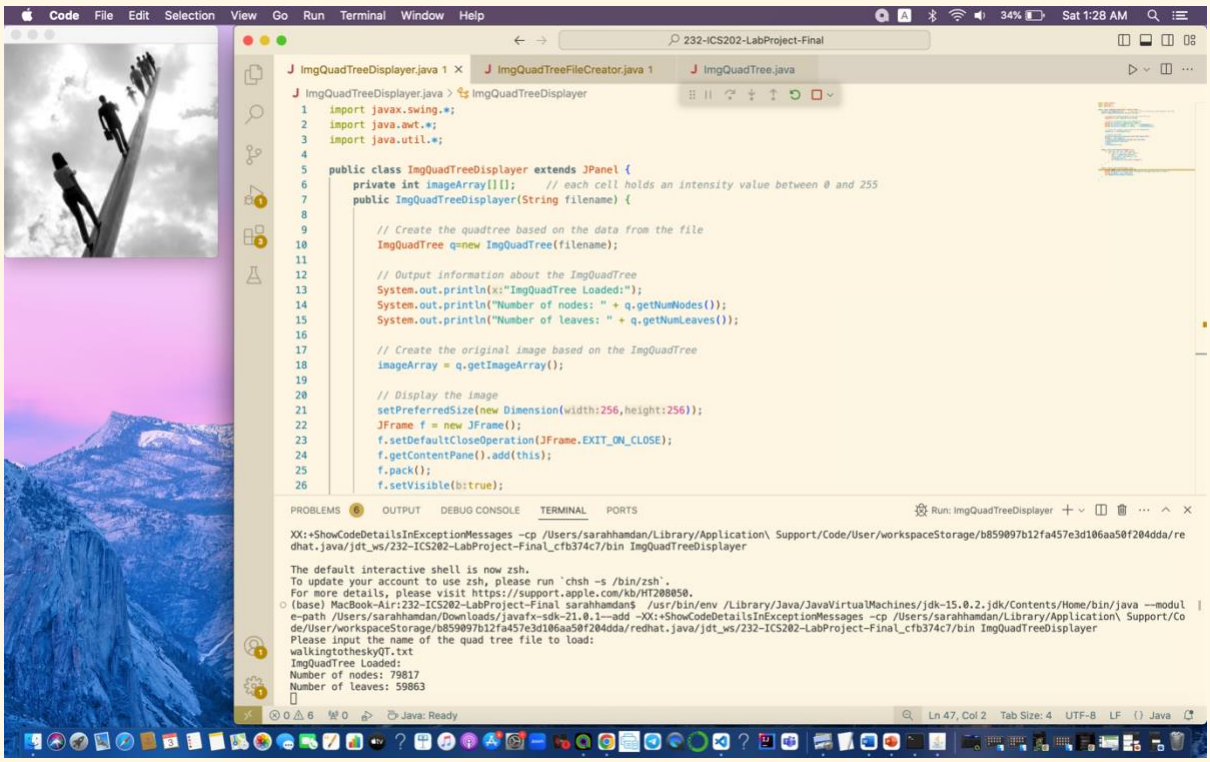
is != -1, then the portion of the array whose starting and ending indices were passed in the method's parameters is simply filled using a nested for loop.

print2DArray Method

```
154     //This is an extra method which allows the user to view the 2D array
155     public static void print2DArray(int array[][]){
156         for (int i = 0 ; i < array.length ; i++){
157             System.out.println(Arrays.toString(array[i]));
158         }
159     }
```

This is an extra method I added for testing purposes. It allows the user to print the 2D array generated by the getImageArray method. In all honesty I had to google how to print a 2D array, but this is just an extra method so that's probably fine :) .

Output (from quadtree to image)



This is the output which results from running the ImgQuadTreeDisplayer class and entering the names of the image file "walkingtotheskyQT.txt". It can be seen above that the number of nodes and leaves was found and printed to the terminal, and the image appeared at the top left part of the screen.

ImgQuadTreeFileCreator Class

Constructor

```
6   public class ImgQuadTreeFileCreator {
7
8       //constructor:
9       public ImgQuadTreeFileCreator(){
10
11          String filename = "smileyface.txt"; //can be changed for a different picture
12          File image = new File(filename);
13          int dimension = 256; //can be changed if the number of pixels was different
14          int[][] imageArray = new int[dimension][dimension];
15
16          /*This class takes input from a file which includes a 2D array's data fields in row-major
17          order. It has to create a file which includes the data of a compressed image
18          represented by an ImgQuadTree printed in that file in pre-order. But first, the array
19          represented in the first file is created in the constructor, afterwards, the compressed
20          image's file is created.
21          */
22
23          try{
24              Scanner input = new Scanner(image);
25              int[][] ar = createArray(input, imageArray); //creating the array first,
26              createFile(ar); //then creating the file using that array.
27          }
28          catch(FileNotFoundException ex){
29              System.out.println(ex.getMessage());
30          }
31
32      }
```

Problem-solving strategy

The constructor of this class has a string variable “filename” which has the name of the file which has the row-major order representation of a 2D array that represents an uncompressed image. This file name can be changed if a different file was to be used. A possible adjustment could be making the filename a parameter which this constructor takes from the user, or creating a Scanner object which asks the user for the name of the file, but this was not requested in the project. A File object is created with the filename, so a Scanner could read the information in the file. This class is supposed to take this 2D array row-major representation, and generate a new file which has the pre-order representation of an ImgQuadTree that represents a compressed version of the first image. This, however, is not done directly. First, the row-major order representation is used to create a 2D array, then this array is used to find what would be the pre-order traversal of the ImgQuadTree that would represent this image. Therefore, an array called imageArray and a Scanner object are first created, and they are passed to the method createArray. The returned array from this method is then passed to the createFile method which finally creates the required file. The constructor calls all these methods, but the detailed processes are done within each one of the methods whose details are discussed in the sections below. Since we have a Scanner object, of course we have a try catch block which allows us to print an error message to the terminal in case the file we are reading from was not found.

createArray Method

```
36      /*
37      The createArray method simply creates a 2D array from the file using a nested for loop, then it
38      returns that array.
39      */
40      public int[][] createArray(Scanner scan, int[][] array){
41
42          for(int i = 0; i <= array.length-1; i++){
43              for(int j = 0; j <= array.length-1; j++){
44                  array[i][j] = scan.nextInt();
45              }
46          }
47          return array;
48      }
```

Problem-solving strategy

This method simply creates an array using a nested loop. Since it is reading from a file which includes the row-major traversal of a 2D array, the for loop for the rows has to repeat 256 times, and then we enter a new column, so the for loop for the columns is repeated, this is straightforward.

same Method

```
50      /*
51      The method same was created because in the createFile method, if a portion of the array
52      has the same values throughout, this means it can be represneted by a leaf. This condition is
53      checked for many times in the createFile method, so it was easier to create a method which
54      checks for it for readability and reusability.
55      This method takes the indices which represent the portion of the array that will be checked,
56      and it simply traverses through that portion of the array. The veriable "val" holds the integer
57      in the first element of this portion, and then all of the array's element are compared to val
58      during the traversal. If an element's value != val, then the method same returns false. If the
59      traversal finishes without returning false, then the method returns true.
60      */
61      public boolean same(int[][] array, int start_horiz, int end_horiz, int start_vert, int end_vert){
62          int val = array[start_vert][start_horiz];
63          for(int i = start_horiz; i <= end_horiz; i++){
64              for(int j = start_vert; j <= end_vert; j++){
65                  if(array[j][i] != val){
66                      return false;
67                  }
68              }
69          }
70          return true;
71      }
```

Problem-solving strategy

This is an extra method I created for the sake of reusability mainly, and it also helps with readability. This method is used in the createFile method since I need to keep checking if a specific portion of the array has the same value throughout. If it does, that means we found a leaf and we add its intensity value to the file (more on that later). I started writing the createFile method first and I realized I was going to be checking if the values within a certain potion of an array are same many times, so I decided to create this method for reusability. This method takes the petameters start_horiz, end_horiz, start_vert and end_vert exactly like the getImageArray method does. The indices are adjusted in the createFile method, again, exactly as they were in the getImageArray method, since the createFile method is basically the inverse of it.

createFile & its helper Methods

```
82    public void createFile(int[][] array){
83
84       Scanner input = new Scanner(System.in);
85       System.out.println(x:"Enter the name of the file which will include the compressed image: ");
86       String filename = input.next();
87       File txtFile = new File(filename);
88       try{
89          PrintWriter output = new PrintWriter(txtFile);
90          boolean identical = same(array, start_horiz:0, array.length-1, start_vert:0, array.length-1);
91          if(identical){
92             output.println(array[0][0]);
93          }
94          else{
95             output.println(x:"-1");
96             createFile(output, array, start_horiz:0, array.length-1, start_vert:0, array.length-1); //it's going to be broken down
97                                                                                                     //in the helper method into its 4 parts
98          }
99          output.close();
100      }
101      catch(FileNotFoundException ex){
102         System.out.println(ex.getMessage());
103      }
104   }
118   public void createFile(PrintWriter output, int[][] array, int start_horiz, int end_horiz, int start_vert, int end_vert){
119
120      //Q1: the first quadrant of the array, AKA the first child of the parent node
121      boolean identical = same(array, start_horiz, (start_horiz + end_horiz)/2, start_vert,(start_vert+end_vert)/2);
122      if(identical){
123         int printable = array[start_vert][start_horiz];
124         output.println(printable);
125      }
126      else{
127         output.println(x:"-1");
128         createFile(output, array, start_horiz, (start_horiz + end_horiz)/2, start_vert,(start_vert+end_vert)/2);
129      }
130      //Q2:
131      boolean identical_2 = same(array, (start_horiz + end_horiz)/2+1, end_horiz, start_vert, (start_vert+end_vert)/2);
132      if(identical_2){
133         int printable = array[start_vert][(start_horiz + end_horiz)/2+1];
134         output.println(printable);
135      }
136      else{
137         output.println(x:"-1");
138         createFile(output, array, (start_horiz + end_horiz)/2+1, end_horiz, start_vert, (start_vert+end_vert)/2);
139      }
140      //Q3:
141      boolean identical_3 = same(array, start_horiz, (start_horiz + end_horiz)/2, (start_vert+end_vert)/2+1, end_vert);
142      if(identical_3){
143         int printable = array[(start_vert+end_vert)/2+1][start_horiz];
144         output.println(printable);
145      }
146      else{
147         output.println(x:"-1");
148         createFile(output, array, start_horiz, (start_horiz + end_horiz)/2, (start_vert+end_vert)/2+1, end_vert);
149      }
150      //Q4:
151      boolean identical_4 = same(array,  (start_horiz + end_horiz)/2+1, end_horiz, (start_vert+end_vert)/2+1, end_vert);
152      if(identical_4){
153         int printable = array[(start_vert+end_vert)/2+1][(start_horiz + end_horiz)/2+1];
154         output.println(printable);
155      }
156      else{
157         output.println(x:"-1");
158         createFile(output, array, (start_horiz + end_horiz)/2+1, end_horiz, (start_vert+end_vert)/2+1, end_vert);
159      }
160
161   }
162
163   }
```

Problem-solving strategy

This method writes the pre-order traversal of a supposed ImgQuadTree using the 2D array created earlier. This method does not actually create an ImgQuadTree, since that is not needed. It writes the pre-order traversal in a new file on the user's device, and since each user might have a file with a different name, this method prompts the user to enter the name of the file which will be written in. It creates a File object using this name and a PrintWriter object, which necessitates having try and catch blocks which will print an error message to the screen in case the file entered by the user is not found. At the end of the try block the PrintWriter object is closed, and then the information is printed into the file.

First, the same method is used to find if the entire array only has one value, which would mean printing one number into the file. Otherwise, -1 is printed and the array is sent to the helper method in which it is sectioned into 4 more quadrants. We use the same method for each one of the quadrants, if it returns true then we print the value of the intensity, if not then we go recursively within this quadrant, and the indices which represent this quadrant are adjusted in the same manner they were in the getImageArray method. This method is basically the opposite of the getImageArray

method, so this was the logic behind it. So similar to the getImageArray method and the add method, this method calls itself four times, once for each quadrant.

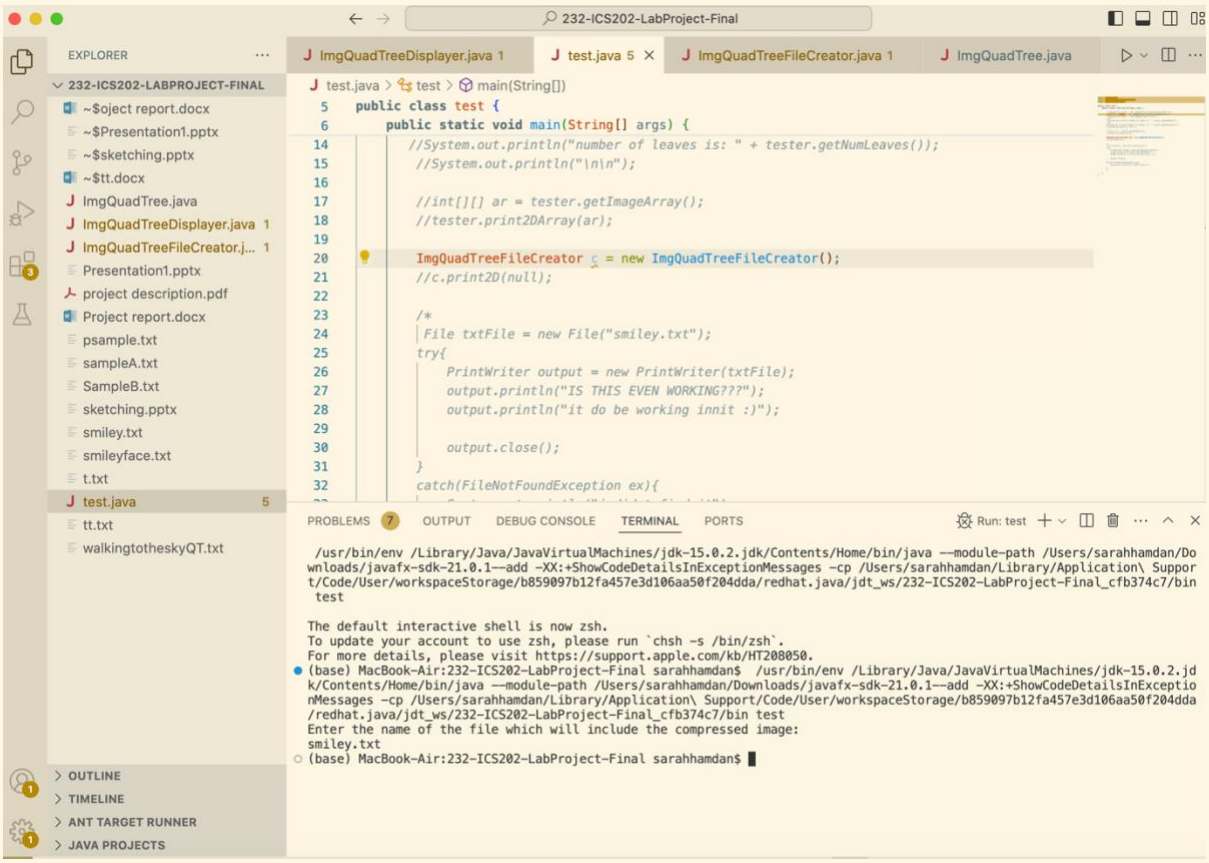
When I created the add method, I learned that the pre-order representation means the first number is the parent node (we print -1), and the next number is its first child. If this child was not a leaf then the following node is the child's child and so on... if its child is a leaf (we print its intensity value) then we move onto the second child until we reach the fourth child, and with each one of the children we go recursively if it was a parent to find its children. As explained earlier, we know whether or not it's a leaf node using the same method, if the same method returns true, then we found a leaf. As explained earlier as well, I am using the terms: node, parent, children, even though we are not actually creating an ImgQuadtree, since we can immediately print the pre-order representation.

Each time same returns true, we print the number (intensity) within that portion of the array into the file. The way we find this number is through taking the value at the first element in that portion of the array. I had first thought of making the same method return this number along with the boolean value using a data structure, but I realized that creating this data structure would probably take up more space, and it is easier and more efficient to access this value using the array's indices directly.

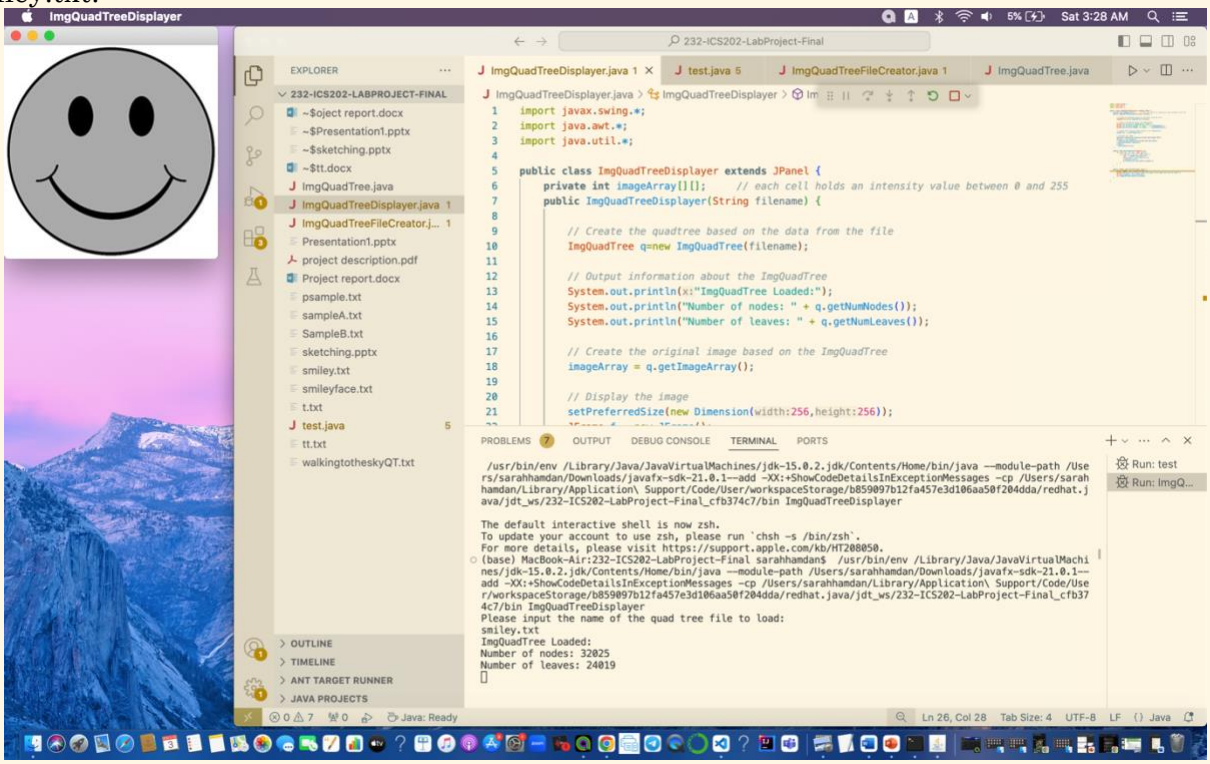
Challenges

I was facing an issue at first since the values were not being printed in the file, and I realized after a google search that I had forgotten to close the PrintWriter object, and it has to be closed in order for everything to be written into the file. The adjustment of the indices was not accurate the first time I wrote the code, so the values being printed were not in the right order, but I switched them and then the order became correct.

Output (from image to quadtree)



In a test class I created an object of ImgQuadTreeFileCreator, in the terminal I was asked to enter the name of the file which will include the compressed image, and I wrote smiley.txt.



Next, I ran the ImgQuadTreeDisplayer class and it prompted me to enter the name of the file which has the quad tree, so I wrote smiley.txt, and the image appeared at the top left portion of the screen. The number of nodes and leaves was printed in the terminal.