Ruslan Khait

Project 1

The first project about Huffman encoding was executed by doing 5 steps. The first step was to get the frequency of each character from the poem. The second step was to use the frequency of characters I got from the poem and create a priority queue to help organizes the frequency from least to greatest. The third step was to use the information I put in the priority queue and use the algorithm for Huffman coding to create an optimal binary tree. The fourth step was to traverse the tree to get the prefix codes for the characters. Finally, the last step was to compress the file and replace each character with its binary number counterpart.

This program was designed in Java. Java has a method called Files.readAllBytes(Paths.get())) that could read a string of the location where the file you want Java to find. I created a method call readFile that takes a String that shows the directory of a file and returns a String of the text inside the file. For this project, the poem was copied and pasted to a text file. The directory of that text file was used to the readFile method. The text inside the file was turned into a String. I used a hash map to count the frequency or the amount of characters the poem contains. Java has a built-in hash map class that I used because it contains a key and a value. This is useful to count the frequency of the characters in the poem. The key was used to hold a specific character and the value of the hash map would hold the frequency of the character. There is a for loop that goes through the entire String that contains the text of the poem. It counts the frequency of each character. If the character was never counted in the loop, the character goes to an empty key in the map and gets the value of 1 because the loop counted only the character once. If there is a key already there for the character than it will add the value that was presented there by one. You repeat this for all the characters until it reaches the end. I created two separate collections that would store two arrays. One for all the characters, and the other for all the Frequencies.

Since trees are used to design the algorithm for the project. It was necessary to create a node. A data type for this project called huffmanNode was created. It contains an integer called frequency, a character type called c, and two huffmanNode types called left and right. The datatype huffmanNode contains a member method that’s a Boolean called isLeaf. In a tree, it checks if a node is a leaf, by checking if the node’s left and right are pointing to null. I also overridden the compare method for the comparator class. The compare method would now compare to different nodes that and both pointing to a frequency and arrange the order from least to greatest. Furthermore, the point of the Huffman algorithm is to check for the minimum values since it is a greedy method. A priority queue can check each frequency and arrange them from smallest to biggest. I created a priority queue called pq where the key stores the character and the value would store the frequency. I created a for loop where a node would loop through both arrays that I created. The character array would go to the character node and the frequency node would go with the frequency node. Both nodes would be added to the priority queue and the priority queue would arrange them in order.

Now it’s time for the Huffman algorithm. The Huffman algorithm is a method where you add two of the smallest frequency then arrange the other frequencies with the sum of the value of the frequencies that you just added. Repeat this method by making a tree where they are no more frequencies to add together. The node that is pointing left would count as a zero and the right would count as a one. From the root going to each character would be the prefix code for each character. I created a tree that does this. First, I created a node called root that points to nothing. While the size of the priority queue doesn’t equal to one. One node called x would point to the smallest frequency then delete that frequency, the other node called y would point to the next smallest frequency and that frequency will also get delete. A new node called z would be created that would equal to the frequency of the sum of the frequency of x and frequency of y. The node z points left to node x and node z points right to node y also. To save the data that I got from z, I put it into the root node. Followed by adding z into the priority queue. This keeps repeating until the priority queue’s size is one, which is the sum of all the frequencies.

Finally, I created a hash map that stores the character and its prefix code. I also created a method that helps find the prefix code for each character. I called this method binaryCode because the prefix code turns into binary numbers. The method goes through the tree in a preorder way. First it starts in the root then goes left then right. The method has a hash map that I mentioned and an empty string as its parameter. If the method goes left a zero is added to the string, if it goes right a one is added to the string. If the method reaches a leaf, save that character that it reached to the map with the string as its value. It does this for all the characters in the tree. There is also another method that would change the characters from the poem to their prefix code. This method is called textToBinary. When it finds a character in the poem it would look through the hash map that contains the prefix code for that character and return the prefix code for each character.

I have a few errors in my program, one being that the compiler outputs the prefix code blank, but if you copy and paste the blank spots to notepad you can see the prefix code. Another problem I was having was that my text file for the prefix code was a lot bigger in memory than my text file with the poem. Maybe it was because I treated the prefix code has a string rather than as a binary number. I thought that both files would have about the same amount of memory.

The size of the tree was 11 and the time complexity was O(nlogn), n is the number of characters that are being searched. The priority queue arranges the numbers from least to greatest, in other words means finds the minimum value which is O(logn). So together it’s O(nlogn).

Huffman.java

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.nio.file.\*;

**import** java.util.PriorityQueue;

**import** java.util.Collection;

**public** **class** Huffman {

**public** **static** String readFile(String filename)**throws** Exception

{

String poem = "";

poem = **new** String(Files.*readAllBytes*(Paths.*get*(filename)));

**return** poem;

}

**public** **static** **void** freqCounter(String s) // only used for testing to see if the hash map worked to store characters and their frequencies

{

HashMap<Character,Integer> freqCount = **new** HashMap<Character,Integer>();

**for**(**char** c : s.toCharArray())

{

**if**(freqCount.containsKey(c))

freqCount.put(c, (Integer) freqCount.get(c)+1);

**else**

freqCount.put(c, 1);

}

System.***out***.println(freqCount);

}

**public** **static** **void** printCode(huffmanNode root,String str) //only used to test the prefix code for each character

{

**if**(root.isLeaf() )

{

System.***out***.println(root.c + ": "+ str);

**return**;

}

*printCode*(root.left, str + "0");

*printCode*(root.right, str + "1");

}

**public** **static** HashMap<Character,String> binaryCode(HashMap<Character,String> map, huffmanNode root,String s)

{

**if**(root.isLeaf())

{

map.put(root.c, s);

}

**else**

{

*binaryCode*(map,root.left,s+"0");

*binaryCode*(map,root.right,s+"1");

}

**return** map;

}

**public** **static** **void** textToBinary(HashMap<Character,String>map, String s)

{

String str = "";

**for**(**char** c : s.toCharArray())

{

**if**(map.containsKey(c))

{

str += map.get(c);

}

}

System.***out***.println(str);

}

**public** **static** **void** main(String[] args) **throws** Exception {

// **TODO** Auto-generated method stub

**final** **int** SIZE = 47; // size = 47

HashMap<Character,String> map = **new** HashMap<Character,String>() ;

String poem = *readFile*("C:\\Users\\rusty\\Desktop\\Algorithms\\Projects\\Project 1\\poem.txt");

// location of the text file

HashMap<Character,Integer> freqCount = **new** HashMap<Character,Integer>();

**for**(**char** c : poem.toCharArray())

{

**if**(freqCount.containsKey(c))

freqCount.put(c, (Integer) freqCount.get(c)+1);

**else**

freqCount.put(c, 1);

}

Collection<Integer> values = freqCount.values();

Integer[] valueArray = values.toArray(**new** Integer[values.size()]);

Collection<Character> ch = freqCount.keySet();

Character[] charArray = ch.toArray(**new** Character[values.size()]);

PriorityQueue<huffmanNode> pq = **new** PriorityQueue<huffmanNode> (SIZE,**new** comparator());

**for**(**int** i = 0; i < SIZE;i++)

{

huffmanNode node = **new** huffmanNode();

node.c = charArray[i];

node.frequency = valueArray[i];

node.left = **null**;

node.right = **null**;

pq.add(node);

}

huffmanNode root = **null**;

**while** (pq.size() != 1)

{

huffmanNode x = pq.peek();

pq.poll();

huffmanNode y = pq.peek();

pq.poll();

huffmanNode z = **new** huffmanNode();

z.frequency = x.frequency + y.frequency;

z.left = x;

z.right = y;

root = z;

pq.add(z);

}

*freqCounter*(poem); // testing to see if the hash map contains the characters and their frequency

*printCode*(root,"");// test to see the prefix code of each character

*binaryCode*(map,root,"");// helping to change the characters to their prefix code

*textToBinary*(map,poem); //the size of the trie is 11. the size of the text file that contains the poem

//is about 1.47 kb while the huffman endcoded is about 6.86 kb

// I thought that the huffman code that compressed the file should have a similar

// amount of memory to the file that it is trying to compress.

// time complexity O(nlogn)

}

}

huffmanNode.java

**public** **class** huffmanNode {

**int** frequency;

**char** c;

huffmanNode left;

huffmanNode right;

**public** **boolean** isLeaf()

{

**return** left == **null** && right == **null**;

}

}

Comparator.java

**import** java.util.Comparator;

**public** **class** comparator **implements** Comparator<huffmanNode> {

@Override

**public** **int** compare(huffmanNode a,huffmanNode b)

{

**return** a.frequency - b.frequency;

}

}