**CSC 464 Program #1**

**Heap Implementation and Huffman Codes**

**DUE : Midnight - Thursday October 6th, 2022**

**Overview**

I would like you to implement a Heap Data Structure (see below) and use it to assist in the process of producing a Huffman Code for text documents. Huffman codes are intended to provide text-compression of documents. Please see the slides on Huffman codes posted to Canvas.

**IMPORTANT NOTES**

Please note that you ***MAY NOT use Java’s built-in PriorityQueue*** class to implement your heap. I am requiring you to implement a ***MINheap*** as defined later in this document. You are free to use other built-in classes as you feel necessary but your heap implementation must be your own creation and all methods described in this document must be implemented.

You MAY USE the Java HashMap class (in fact, I would recommend it) or a similar HashMap-like structure which allows you to place an item into a collection using a character for the index and an integer for the value. For example, in Java

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

Please also note that, in the “real world”, you would obviously often choose to use all built-in classes to implement your program but unfortunately, ***I am requiring you to implement your own Heap class as part of the assignment.***

This variable ***map*** will be used to count the frequencies of each character in a text file.

**Program Description**

Before you even begin to design your program or put fingers to keyboard, you need to be 100% sure that you understand the algorithm that we discussed for Huffman Encoding. Understand how we build a Huffman Tree in a Bottom-up fashion. If you do not understand this, then you will most likely fail to complete this assignment. That said, - go READ THE SLIDES if you have not already done so.

Your program will prompt the user for a path to an input file. You may assume that the file is a standard text file. Read the file line by line and walk through each line character by character. If a character has not been seen before, then it should be inserted into the HashMap with a value of 1. If the value has previously been seen, then its value in the HashMap should be incremented by 1. Once the entire file has been read, the data input is complete. **Close the file!!** The HashMap now contains each unique character in the file along with a count for how many times it occurred.

**NOTE about Reading File line by Line for Grad Students**

Most programming languages strip off the end-of-line marker(s) when you read a line of text from an input file. Since we know that, we can insert and end-of-line symbol into our hashmap and increment it for each line of input we read. It shouldn’t matter what symbol you use – I would use either LineFeed (ASCII value decimal 10) or CR (ASCII value decimal 13) just be consistent. When it comes time to print out the end-of-line simply use something equivalent to

**System.out.println(“”);**

In other words, print an empty line. This will use the correct end of line for whatever operating system is in use.

**HeapNode Class**

You will need to define and use the following HeapNode class. These objects will be what is actually stored into the Heap that you will be using.

![Table

Description automatically generated]()

**public class HeapNode**

**{**

**Character ch;**

**Integer freq;**

**HeapNode left = null;**

**HeapNode right = null;**

**public HeapNode(Character ch, Integer freq)**

**{**

**this.ch = ch;**

**this.freq = freq;**

**}**

**public HeapNode(Character ch, Integer freq, HeapNode left, HeapNode right)**

**{**

**this.ch = ch;**

**this.freq = freq;**

**this.left = left;**

**this.right = right;**

**}**

**}**

Your next task will be to create a HeapNode for each of the characters that were found in the file. How you accomplish this is pretty much up to you but I will show you a way that I often access all of the key/value pairs of a HashMap. Your code may differ but I wanted to give you an idea of how to access all key/value pairs in a HashMap.

**for ( Map.Entry <Charatcer,Integer> entry : map.entrySet() ) {**

**Character k = entry.getKey() ;**

**Integer v = entry.getValue();**

**// now I can create a HeapNode with character c and frequency count v and**

**// insert it into the Heap.**

**}**

The node should be created using the first HeapNode constructor and be passed a character and its corresponding count as found int the HashMap.

Remember that your Heap will be a MinHeap and contain HeapNodes that will be ordered based upon the frequency count field. Thus, the lowest frequency character will be stored at the root of the tree, in the form of a HeapNode.

Once all of the character-frequency HeapNodes have been inserted into the Heap, you can begin to produce the Huffman codes for each character. This process consists of deleting the minimum item of the Heap and storing it into a variable ***left*** (i.e. the root) and then deleting the new minimum item from the Heap and storing it into a variable ***right***. At this point, you should create a new HeapNode using the second HeapNode constructor. The constructor should be passed ***null*** for the ch field, ***(left.freq + right.freq)*** for the freq field, and ***left*** and ***right*** for the left and right fields respectively.  ***Once constructed, the new node should be inserted into the heap.***

This process should continue until there is only one single HeapNode remaining in the Heap. At that point, the Huffman tree has been completely built. The Huffman tree is rooted at the single HeapNode that remained in the Heap.

![Diagram

Description automatically generated]()

The next step is to assign an actual Huffman code to each character in the Huffman tree. We do this by performing a Preorder traversal of the tree and assuming that each left edge we follow adds a 0 to the code and each right edge we follow adds a 1 to the code. You should note that all characters are leaves in the tree.

You should store the characters and their associated Huffman codes into a new second HashMap structure which I call HuffmanCode. ***This second HashMap uses Characters as keys and Strings as values.*** Create this HashMap and pass it as an argument to the encode( ) method described below. ***The initial call to this method should pass the root of the tree, the empty string “”, and the new HuffmanCode hashmap.***

Below is a pseudo-code outline of this *recursive* **encode**( ) method.

**public static void encode(HeapNode node, String str,**

**HashMap <Character, String> HuffmanCode)**

**{**

**if (node == null) { // base case**

**return;**

**}**

**// Found a leaf node**

**if node is a leaf { // remember this is pseudo code right?**

**If the *str* passed in as an argument is not empty then**

**Insert node.ch into HuffmanCode hashmap with value str that was passed in.**

**Else insert node.ch into HuffmanCode hashmap with value “1”.**

**}**

**encode(node.left, str + '0', HuffmanCode); // recursively call on left subtree**

**encode(node.right, str + '1', HuffmanCode); // recursively call on right subtree**

**}**

Once we have recursively encoded the Huffman Tree, we have stored all of the characters into the new HuffmanCode hashmap and can now print them out using something like :

**System.out.println("Huffman Codes are: " + HuffmanCode);**

Your program should now reopen the file and again read it line by line. Build a String of the encoded text using the appropriate Huffman Code that was calculated and stored into the HuffmanCode hashmap. When all lines have been read and encoded, print out the encoded file of 0’s and 1’s.

**CSC 564 Students ONLY!!!**

I am requiring Graduate Students to also implement a **decode**( ) method which will take the encoded string which was just printed out and decode it back into its original text using the Huffman code that was produced and print out the result. This decoded text should now be printed. I will be looking to make sure you have not tried any shortcuts here.

**Heap Class**

I am requiring that you implement your own Heap class. Obviously, you should rely upon the slides and other sources to ensure that you understand the standard array implementation of a Heap (AKA Partially Ordered Tree).

**Heap( )** - You should have a constructor for your class that initializes your data members. These will consist of an array (you may assume that an array of 100 HeapNodes will suffice for any input file) and an integer counter to keep track of the number of elements in the Heap at any time.

**HeapNode** **peek**( ) - This method returns the root node but DOES NOT remove it from the Heap.

**HeapNode** **deleteMin**( ) - This method will *remove* and return the root node of the Heap. It should also use the method describe in the slides to reconfigure the Heap into a correct form.

**void**  **insert**( HeapNode node ) - This is the standard Heap insertion. Again, see the slides for how the heap insertion works.

**int** **getSize**( ) - This method returns the current size of the Heap, meaning the number of Heapnodes currently contained in the Heap.

**Sample Input**

*Huffman coding is a data compression algorithm.*

**Sample Output**

**Do NOT add a newline character entry UNLESS there is more than one line in File! Always assume that the last line has no return on it. So, if you simply count the lines that you read, either do not add an entry into the frequency count HashMap for end-of-line (in the case of a single line file) or add the end-of-line with a frequency of *(# of lines read) – 1*. This will match my sample output files.**

**I used the value of Character.valueOf( (char) 0) as my end-of-line symbol when building the frequency hashmap. I simply kept a count of the lines read and added that count -1 as the frequency for this value into the hashmap after the entire file was read.**

**It is important to remember this so that when you go to encode the file you will need to add the correct Huffman code for it at the end of each line of text.**

**Frequency counts are: { =6, a=5, c=2, d=2, e=1, f=2, g=2, H=1, h=1, i=4, l=1, m=3, n=3, .=1, o=4, p=1, r=2, s=3, t=2, u=1}**

**Huffman Codes are: { =100, a=010, c=0011, d=11001, e=110000, f=0000, g=0001, H=110001, h=110100, i=1111, l=101010, m=0110, n=0111, .=10100, o=1110, p=110101, r=0010, s=1011, t=11011, u=101011}**

**The original file is: Huffman coding is a data compression algorithm.**

**The encoded string is:**

**11000110101100000000011001001111000011111011001111101110001100111110111000101001100101011011010100001111100110110101001011000010111011111111100111100010101010000111100010111111011110100011010100**

**The decoded file is:** **Huffman coding is a data compression algorithm.**

This last part of output (shown in red) is only required of graduate students enrolled in CSC 564.

**Submitting your program**

**I want ALL CLASSES in separate files.** For example, the Heap class should be in a file named **Heap.java** (or similar), the HeapNode in **HeapNode.java** and so on. Your main class should be named Prog1 and stored in **Prog1.java**. You should have no other files/classes.

You should also run your program on the data files supplied on Canvas. There should be 5 of them. Save the output from all 5 of your runs into separate files and submit them along with your source code files.

***If you KNOW that your program has problems, either in compilation, execution or incorrect results, please provide a brief README file explaining known issues.***

Place all of your files into a folder/directory named LastName\_FirstName\_prog1 (for example, I would place all of my files into a folder named newell\_gary\_prog1) then Zip/compress your folder and submit it on canvas using the supplied link.