

Huffman Encoder

CS21120 Assignment



May 12, 2016

Jakub Janas – jaj48

Contents

[1. Introduction 2](#_Toc468873646)

[1.1. Huffman encoding scheme 2](#_Toc468873647)

[2. System design 3](#_Toc468873648)

[3. System implementation and operation 4](#_Toc468873649)

[3.1. FileReader class 4](#_Toc468873653)

[3.2. characterCount() 5](#_Toc468873655)

[3.3. sortingPQ() 5](#_Toc468873656)

[3.4. HuffmanNode class vol.1 6](#_Toc468873657)

[3.5. constructATree() 6](#_Toc468873664)

[4. Time and space complexity 7](#_Toc468873665)

[5. Conclusion 7](#_Toc468873666)

# Introduction

In this assignment we were to implement the Huffman encoding scheme in Java programming language. This scheme is a very efficient encoding scheme which is about 66% more efficient than the most popular American Standard Code for Information Interchange (ASCII).

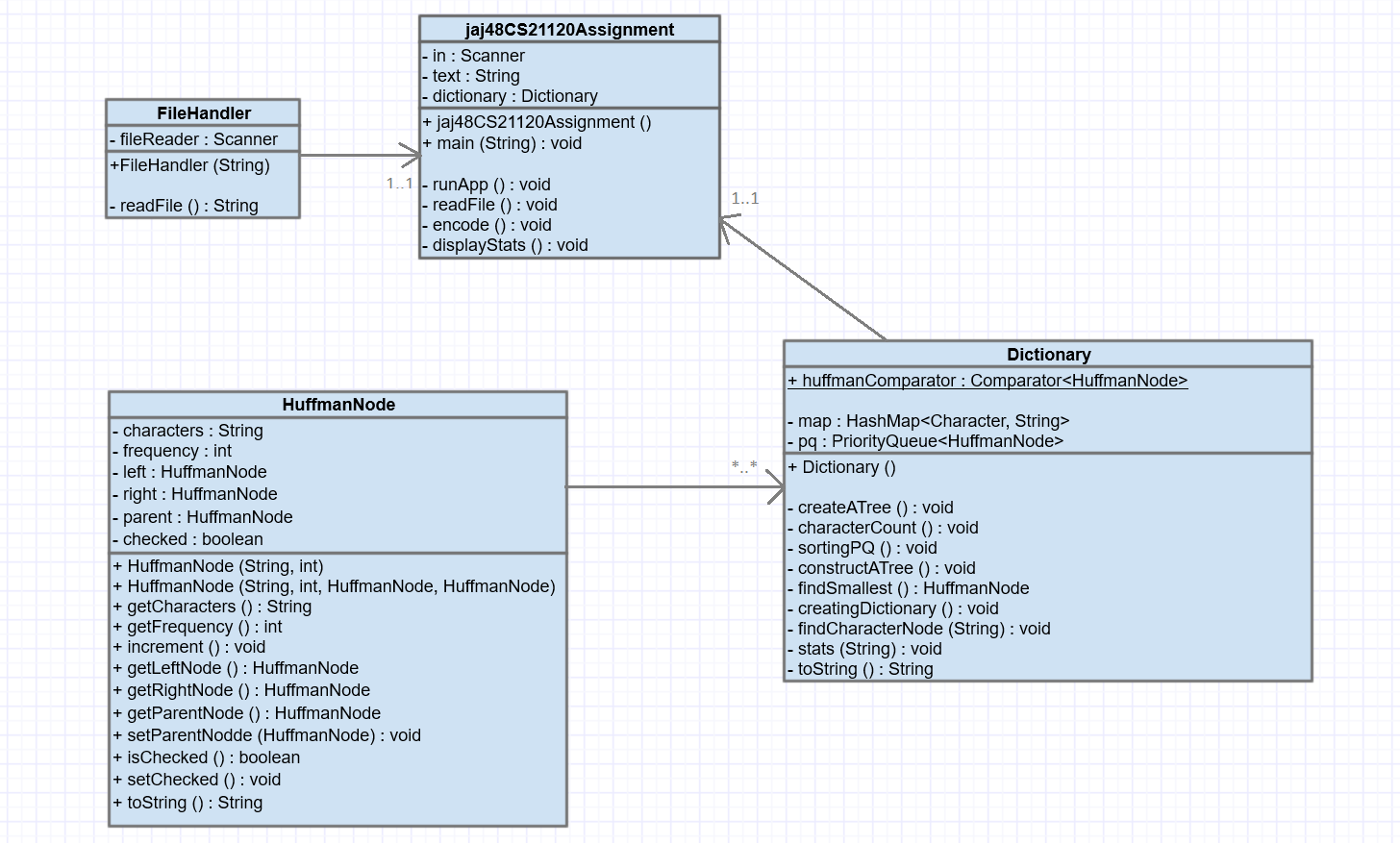
## Huffman encoding scheme

Every piece of data is encoded in a different way. It is based on the idea that the most common character should have the shortest code. In this way we can shorten the overall code by about 66%, meaning there will be only 33% of the characters of the usual ASCII encoding as a product of this encoding scheme. There are four stages to create a “dictionary” for every character:

* Count the frequency of every character that appears in the data
* Sort it in descending order
* Build the binary tree: the bottom row are the basic characters that you already have from the previous step, find 2 smallest frequencies and add them together creating a new element, repeat that step until you run out of elements to add together
* Traverse the tree for every character and generate the code by adding 0s and 1s: 0 if the lower element in the path is on the left side and 1 if the element is on the right size

After that you have a “dictionary” ready to encode the data. In my opinion I have succeeded in implementing that system correctly and my program provides all the specified information.

# System design



My program consists of four classes. Jaj48CS21120Assign is my application class. It has 3 instance variables: the scanner for taking input (just for getting the file path), string for holding the contents of the file and dictionary which is an object that encodes and holds the dictionary for the file. The application is also instantiating a FileHandler object which open a FileReader stream and reads in the contents of the file into a single string. It is then returned to application and the passed to the Dictionary object for encoding. The encoding is carried out in four steps:

* Counting the frequency of the characters and putting it into the hashmap
* Sorting by putting the characters and frequency into a priority queue, with a custom comparator that sorts in descending order, elements are of class type HuffmanNode
* Creating a binary tree, by adding the two smallest frequencies and creating a new HuffmanNode with a string of the characters from the children and sum of frequencies, repeating until the smallest frequency is the root
* Traversing the binary tree from bottom to the top, generating a binary code for each character by adding 0 if the current node is on the left of the parent node and 1 if on the right

While traversing a tree I am also storing different depths of the tree. This is necessary to calculate the average depth of the tree. After that the only thing that is left to do are the statistics. The uncompressed size is just the number of characters multiplied by 8 and the compressed size is each character’s frequency multiplied by the number of characters in it’s binary code and added together. Then the height which is the biggest depth of the tree, and number of nodes which is just size of the binary tree.

# System implementation and operation

My application class jaj48CS21120Assign is rather short and has as less as possible. I tried to make it concise because I wanted it to just “overlook” the encoding process. The main() method is just instantiating an object of class CS21120Assign and calling its runApp() method. The constructor for that class is instantiating a Scanner and object of class type Dictionary. It is also defining a new empty String for holding the contents of the file text. Going back to the runApp() it is firstly calling a readFile() method in order to read in a file.



## FileReader class

In readFile() the program will prompt you for the path to the file.

**while** (text == "")

{

System.***out***.println ("Specify the path to the file:"); String fileName = in.next();

**try**

{

FileHandler file = **new** FileHandler(fileName);

text = file.readFile();

}

**catch** (FileNotFoundException fnf)

{

System.***out***.println("File not found. ");

}

}

As you can see in above excerpt from the code this method is safe and insured for bad user input. Try/catch is surrounding instantiation of the FileHandler object which can return an exception if the file is not found. Additionally the while loop will run until a file will be read in. Inside the FileHandler class the file is read in using a Scanner with FileReader. Then when the method readFile() is called I am using a delimiter() with parameter “\\Z” to ensure that the Scanner will only stop at the end of the file.

d



## characterCount()

The next method called in encode() is dictionary.encode(). It is a general method in Dictionary class. In there everything is called in specified order. The first is characterCount().

**for** (**int** i = 0; i < text.length(); i++)

{

**char** character = text.charAt(i);

**if** (map.get(character) == **null**)

{

map.put(character, 1);

}

**else**

{

map.put(character, (map.get(character) + 1));

}

}

It is a simple method goes through every character in the string and if this is not in HashMap yet, I will add it with value 1. Otherwise it adds it to the HashMap again with incremented value.

## sortingPQ()

For this bit I have a instance variable of PriorityQueue with a custom comparator.

**public** **static** Comparator<HuffmanNode> *huffmanComparator* = **new**  
Comparator<HuffmanNode>()

{

@Override

**public** **int** compare(HuffmanNode node1, HuffmanNode node2)

{

**return** -(node1.getFrequency() - node2.getFrequency());

}

};

I had to create a new comparator since I used a HuffmanNode class as elements of this PriorityQueue and a PQ does not have an ability to compare custom objects like these on its own. I will disscuss the HuffmanNode in the next subsection. The comparator is simply taking out the frequency of the nodes and compares them, but in order to sort the PQ in descending order I have to return an inverse of the actual substraction.

The next step is to put the keys and values as single elements into the PriorityQueue. I have tried to just reclaim them from HashMap, but with the very first character came a very big problem. The first character in ASCII table is NULL (0). When I tried to use it for getting the value I have been receiving a NullExceptionPointer. My solution to this was to take out the keyset and values as a Set and Collection respectively and use iterators to iterate over them. This time it worked perfectly and for every character and frequency the HuffmanNode object is created for it. The PQ is populated.

## HuffmanNode class vol.1

The first version of HuffmanNode is used in the pq for sorting. It only needs two instance variables. I have chosen a String over a character because I want to use the same class for the binary tree later where you need a to add the characters and frequencies from two nodes together. It also needs an integer variable for frequency. I have created getter methods for these two, but I don’t want them to change them ever so I omitted setter methods.



## constructATree()

Then in the encode() a constructATree() method is called. This one actually creates all the nodes for the binary tree and puts them in the new PriorityQueue called binary tree.

**while** (!pq.isEmpty() || findSmallest() != binaryTree.peek())

{

HuffmanNode minOne = findSmallest();

minOne.setChecked();

**if** (pq.contains(minOne)) // if it is a node from pq

{

pq.remove(minOne); // remove it from there

}

HuffmanNode minTwo = findSmallest();

minTwo.setChecked();

**if** (pq.contains(minTwo))

{

pq.remove(minTwo);

}

**if** (!binaryTree.contains(minOne))

{

binaryTree.add(minOne);

}

**if** (!binaryTree.contains(minTwo))

{

binaryTree.add(minTwo);

}

String combinedChars = minOne.getCharacters() + minTwo.getCharacters();

**int** combinedFrequency = minOne.getFrequency() + minTwo.getFrequency();

HuffmanNode newNode = **new** HuffmanNode(combinedChars, combinedFrequency, minOne, minTwo);

minOne.setParentNode(newNode);

minTwo.setParentNode(newNode);

binaryTree.add(newNode);

}

So this is implemented to run until there are no nodes in the old pq and the first element of the binaryTree is the smallest of the unchecked nodes. I have a helper function called findSmallest() that searches through the pq and the binaryTree for a smallest node and returns it. When it’s returned the setChecked() method is called on it so that it will not be searched again. If it’s in pq then we have to remove it from there. The same is called again to acquire a second smallest node (setChecked() is called and it’s removed from pq if needed). Then if they are not in the binaryTree then they are added. After that calculations are made in order to create a new node. With that new node I am setting the two smallest node parent instance variable to. After that the only thing left is to add it to the binaryTree.

# Time and space complexity

# Conclusion