CS130A Project 2 : Hoffman Encoding
Due: Friday February 12, 2016 at 11:59pm

January 31, 2016

Introduction

Standard character encoding (like ASCII, Extended-ASCII, and UTF-8) are used to encode characters in standard in-memory format that is widely understandable by different programs and machines. These standards give each character a code (sequence of 0's and 1's) that represents this character internally in memory. We use these standards to be able to exchange text files and web pages without the need to send the encoding scheme between machines. The drawback of using standard representation is that each character is represented by the same number of bits which might be larger than needed. In order to understand this concept, let's discuss an example. Assume we have a text file that is written in English and it has only lower case characters (26 characters), spaces, '.', and new lines '\n'. So there are only 29 different characters that are used and they require only 5 bits to represent each character. 5 bits give us 32 different possibilities starting from 00000 to 11111. A character set of size C requires **Ceil(log(C))** bits to represent them. **why?**

So, why do we use ASCII or UTF-8 then? Because we have to agree on a standard encoding or the encoding scheme has to be sent alongside with every document to be able to decode it.

Huffman encoding algorithm is used to compress files based on the frequency of individual letters in these files. The idea is that we can use different number of bits to represent characters in a file based on their frequencies (i.e. using fewer bits to represent the more frequent characters than the least frequent ones). The general idea behind Huffman encoding is to allow a variable code length for different characters such that the most frequent character is encoded with the shortest code and vice versa. If all the characters occur with the same frequency, saving is more unlike.

In order to do the encoding, we must do the following steps:

1. **Count frequencies**: in this step, we count the frequency of each character in the file.
2. **Construct the encoding tree**: in this step, we represent the frequencies found in step 1 in the form of tree where characters are represented by leaves of the tree. For

more details about the construction of the tree, check Huffmans Algorithm section on page 456 on the textbook.

3. **Construct the encoding map**: by traversing the tree constructed in step 2, we can know the code of every character and construct a map.
4. **Encode File Characters**: in this step, we represent every character with its binary code and save that in the output file.

In the **decoding phase**, we are given a binary string i.e 001010011101101001010010 that represents the output file of the encoding and we want to return it back to the original file. We start reading the binary string bit by bit and traverse the encoding tree we constructed in the encoding phase. We start from root and for every 0 we move right and for every 1 we move left till we reach a leaf node. Upon reaching a leaf node, we replace this part of the binary string with the character at this leaf node and repeat till the end of encoded (compressed) file.

Your task in this assignment is to do the following:

1. Given a text file (plain.input), you have to implement a function to read the file, and count the character frequency of all the characters in this file.
2. Construct the encoding tree (check appendix A for a complete example). You must use a minheap to get the smallest 2 nodes (sub-trees) to be merged. Do not forget to insert the resulting sub-tree with its new weight to the heap to be merged in later steps. Usually we save the resulting tree on file beside the encoded (compressed) file to be able to load it later and decompress it. For simplicity, we just want you to keep the encoding tree in memory and use it in encoding and decoding.
3. Construct the encoding map.
4. Encode the text of the input file based on the map and print the encoded string on stdout.
5. Implement a decoding function that uses the encoding tree and decodes the lines of another input file (encoded.input) using this tree. The output should be printed in stdout. You are only required to print the output of the decoding in stdout.

Assumptions

1. You can work in this assignment in groups of 2 students.
2. If you work in a group of two, only one of you should submit the assignment on gauchospace. You should submit one file named `asg2_perm1_perm2.zip`. If you are working alone, your file should be named `asg2_perm.zip` i.e. `asg2_1234567.zip`.
3. For easiness and without loss of generality, you can assume that the first input file (plain.input) contains only lower case characters [a-z] and spaces, so in the frequency count step, do not forget to count the spaces.

A Example

In order to have the same output, we should agree on a unified way to build the trie. Assume our input file (plain.input) has this text ”i like apples”. It has to have only lower case characters and spaces. In order to count the frequencies, we create an array of size 27 (26 buckets for the lower case characters [a-z] and 1 bucket for the space character) initialized into zero as shown in Table 1.

Char	a	b	c	....	z	sp
Frequency	0	0	0	0	0	0

Table 1: Initial frequency array

Then you should iterate over the text of the input file, character by character, and increment the corresponding bucket in the array. Also you should keep track of the number of unique characters.

After doing the frequency count, the array should look like Table 2. The number of unique characters in this case is 8.

Char	a	...	e	...	i	j	k	l	...	p	...	s	...	sp
Frequency	1	0	2	0	2	0	1	2	0	2	0	1	0	2

Table 2: Frequency count

Then you should create a min-heap having only the non-zero frequency character as in Table 3.

	a	e	i	k	l	p	s	sp
	1	2	2	1	2	2	1	2

Table 3: Initial heap

Then we call buildHeap() to build the min-heap. The invariant is satisfied by applying the following rules:

Rules

1. The invariant for the heap is the parent is $\leq$ the children. Hence, when percolating up and the node frequency equals to the parent’s frequency or when percolation down and the node frequency equals to children’s frequency, percolation stops.
2. While percolating down, if both children have the same weight (and this weight is smaller than parent’s weight), the parent is swapped by the right child.

3. When merging 2 nodes to create a trie, the first deleted node (from heap) is merged as the right child and the second deleted node is merged as the left child.

Before building the heap, it should look like Figure 1

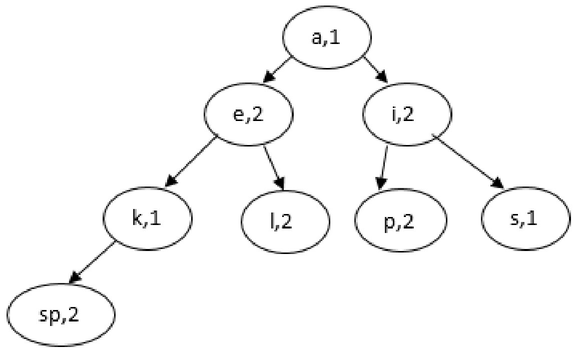

Figure 1: The heap before calling buildHeap()

After calling buildHeap(), it should look like Figure 2

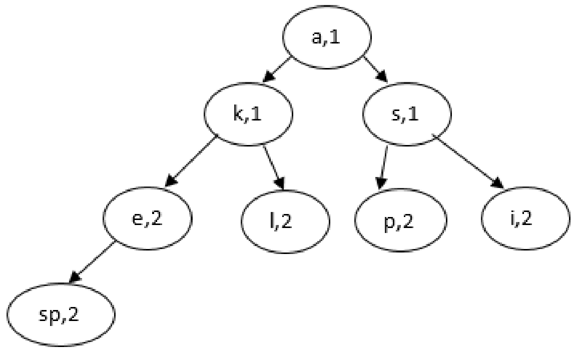

Figure 2: Initial heap

Now, start constructing the trie.

1. deleteMin twice from the heap and merge the deleted nodes into one node. The first deleted node goes right and the second deleted node goes left.
   Delete min (a,1) creates a hole at the roots as shown in Figure 3

   We fill this hole with the right most leaf node as shown in Figure 4

   Percolate down and swap the parent with the right child when the 2 children have the same weight as in Figure 5.

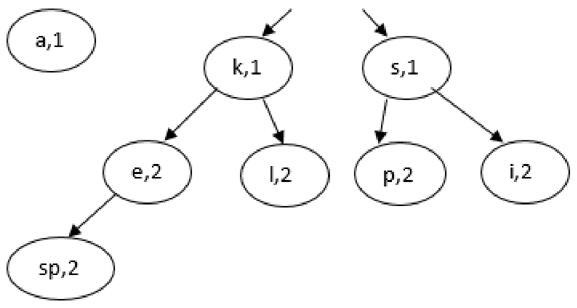

Figure 3: Create hole at the root

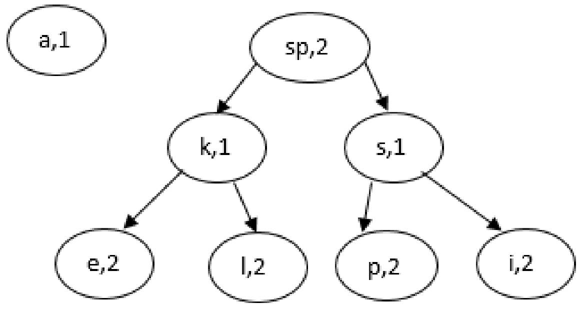

Figure 4: Delete the right most leaf and insert it at the root

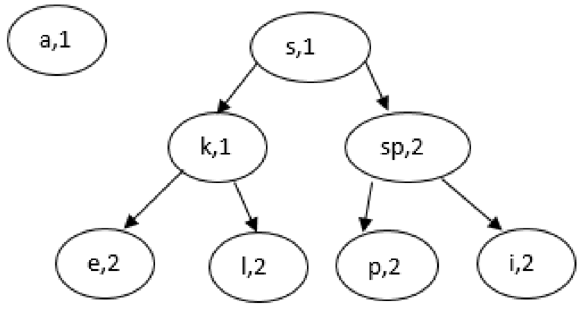

Figure 5: Swap parent with the right child when both children have the same weight

Now, percolation stops as the weight for sp equals to the weight of both of its children.

Delete min (s,1) creates a hole at the roots as shown in Figure 6

We fill this hole with the right most leaf node as shown in Figure 7

Now, percolate (l,2) down as shown in Figure 8

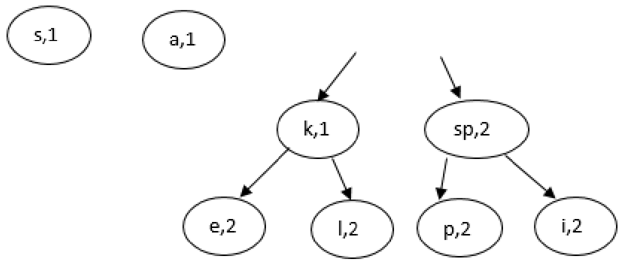

Figure 6: Create hole at the root

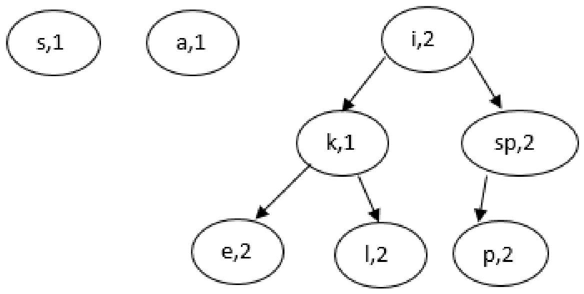

Figure 7: Delete the right most leaf and insert it at the root

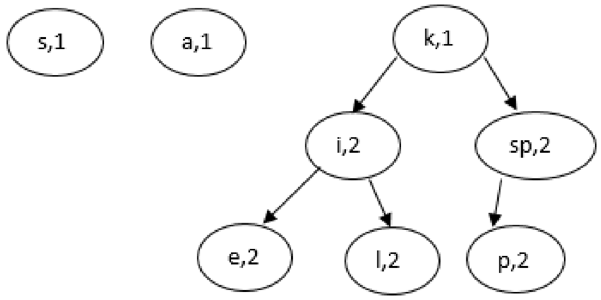

Figure 8: Percolate(l,2) down

Then percolation stops.

2. Now merge s,1 an a,1 (a,1 goes to the right because it is firstly deleted from the heap) as shown in Figure 9

3. Now, insert `t_1` into the heap as shown in Figure 10. Notice that `t_1` represents the

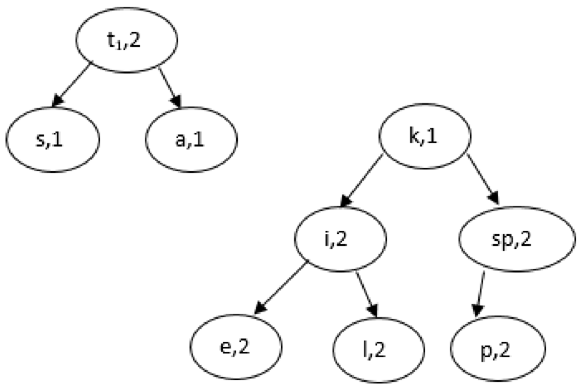

Figure 9: Merge a,1 and s,1 into t_1,2 and insert it into the heap

2 deleted nodes s,1 and a,1. That's why it weighs 2.

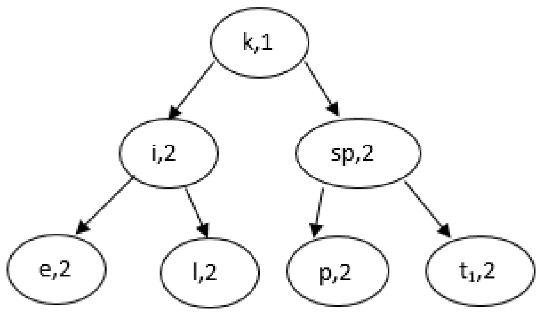

Figure 10: Insert `t_1` into the heap.

In this situation, t_1 is not percolated up because its parent weight equals to its weight. Percolation stops here.

4. Repeat step 1-3 till we have only one node which is the root of the trie.

   After that, we have to generate codes for our letters. The rule is, whenever you go right, put ZERO and whenever you go left put ONE. The code of a letter is the sequence of zeros and ones from the root till this character which is a leaf node