CSI 2110 Computer Science Fall 2016 University of Ottawa

Assignment #2 Programming (75 points, weight 7.5%); Due: Sat, October 15, 11:59PM

The assignment must be uploaded on blackboard; follow the submission instructions.

Late submission is only accepted from 1 min - 24 hs late for 30%off (i.e. your mark *0.7).

Huffman Compression

Lossless compression is a type of data compression that allows the original data to be recovered from the compressed data. Huffman encoding is a compression technique that reduces the number of bits needed to store a message based on the idea that more frequent letters should have a shorter bit representation and less frequent ones should have a longer bit representation. Text compression is useful when we wish to reduce bandwidth for digital communications, so as to minimize the time required to transmit a text. Text compression is also used to store large documents more efficiently, for instance allowing your hard drive to contain as many documents as possible.

In this assignment, you will implement the Huffman compression and decompression algorithm.

Objectives: Practice implementing tree-based data structures. Practice analyzing different alternative implementations for auxiliary data structures (priority queues). Appreciation of how data structures can be used for practical purposes. Having fun while learning.

1 How the Huffman code works

Suppose we have 4 possible letters that appear on a message, string or file: A,C,T,G. We could encode these letters with a fixed length code using 2 bits for each letter:

letter	A	C	G	T
fixed length encoding	00	01	10	11

In this way, message: AACGTAAATAATGAAC that has 16 letters can be encoded with 32 bits as 00000110110000001100001110000001

Huffman encoding would instead choose a variable length code to take advantage of difference in letter frequencies:

letter	A	C	G	T
Frequency in text	9	2	2	3
Huffman code	1	010	011	00

With Huffman encoding the same message can be encoded with 27 bits (saving 15% space)

110100110011100110001111010

Now imagine a more drastic scenario where a text message uses a standard 8-byte encoding for each character (like in UTF-8 character encoding) able to support 256 different characters. However, our specific message is the same as in the example above, so that for the 4 characters we have the same frequencies as shown in the table above but the other 252 other

characters have frequency 0. Now the message above would be encoded in UTF-8 using 16*8=128 bits or 16 bytes. Now comparing our Huffman encoding above that has 27 bits, which is a bit less than 4 bytes we can now save 75% space.
Now if you are not impressed about saving 12 bytes of storage, suppose you want to store the entire human genome composed of a sequence of 3 billion nucleotide base pairs. This can be represented by a sequence of 3 billion characters each being one of A (adenine), T (thymine), G (guanine), C (cytosine), which could be stored using 3 billion bytes using UTF-8 encoding which is about 3GB. Huffman encoding for such a file would depend on the relative frequency between these four letters, but assuming about 25% frequency of each of these 4 letters (and of course absence of any other of the 252 character), Huffman would encode each letter with 2 bits yielding a compressed file of 750MB.

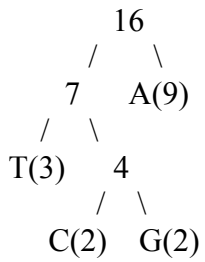
How to build the Huffman code from the letters in a text and their frequencies?

The Huffman code is always a **prefix code**, that is, no codeword is a prefix of any other codeword. This generates no ambiguity. When decoding our example

110100110011100110001111010

we have no doubt that the first word is an A since no other codeword starts with a 1; similarly we decode the next A. Then the next bits will be read one by one until we conclude unequivocally that the next codeword can only possibly be 010 which is a C. Try continuing this example to see how the original text can be decoded using the Huffman code given in the table in the previous page.

Huffman's algorithm produces an optimal variable-length prefix code for a given string X. It is based on the construction of a binary tree T that represents the code.
Here we quote the textbook by Goodrich et al. 6th edition page 595:
Each edge in T represents a bit in a codeword, with an edge to the left child representing a 0 and an edge to the right child representing a 1. Each leaf v is associated with a specific character and the codeword for that character is defined by the sequence of bits associated with the edges in the path from the root of T to v. Each leaf v has a frequency, f(v), which is simply the frequency in the string X of the character associated with v. In addition, we give each internal node v in T a frequency, f(v), that is the sum of the frequencies of all the leaves in the subtree rooted at v.
For the previous example, where X="AACGTAAATAATGAAC", and the frequencies and codewords given in the previous table, we have the following Huffman tree:



In order to understand how the tree can be used for decoding, use the tree to decode the encoded string 110100110011100110001111010 to obtain back X. You can find another example in page 596 of the textbook. Inspect that to better understand Huffman trees.

Now, we just need to explain the algorithm to build the Huffman tree: the Huffman coding algorithm begins with each of the characters of the string X being the root node of a single-node binary tree. Each iteration of the algorithm takes the two binary trees with the smallest frequencies and merges them into a single binary tree creating a root of the new binary tree having the sum of he frequencies of the two subtrees. This process is repeated until only one tree is left.
The construction of the tree in the example above is illustrated next:

Start: A(9), T(3) , C(2), G(2)

Iteration 1: A(9), T(3) , 4

```
          /   \
       C(2)   G(2)
```

Iteration 2:) A(9), 7

```
          /   \
       T(3)    4
             /   \
          C(2)   G(2)
```

Iteration 3: 16

```
          /     \
        7       A(9)
      /   \
   T(3)    4
         /   \
      C(2)   G(2)
```

At each iteration, we need to remove 2 trees with minimum frequency and insert 1 combined tree with the sum of the subtree's frequencies in the list of trees. These two operations indicate we need to use a priority queue. The algorithm to build a tree is given next:
Algorithm Huffman(X):

Input: String X of length n with d distinct characters
Output: Huffman tree for X

1. Compute the frequency f(c) of each character c of X.
2. Initialize a priority queue Q
3. For each character c in X do {
4. Create a single-node binary tree T storing c
5. Insert T with key f(c) into Q }
6. While Q.size() > 1 do {
7. e1= Q.removeMin()
8. e2= Q.removeMin()
9. Create new binary tree newT with left subtree e1.tree and right subtree e2.tree
10. newFreq= e1.freq+e2.freq
11. Insert newT with key newFreq into Q
}
12. e= Q.removeMin()
13. return e.tree

Proving that the Huffman tree algorithm produces an optimal prefix code is beyond the scope of this assignment, but this can be found in other more advanced textbooks. This is an example where the ``greedy method'' is successful in finding the optimal solution for a problem.

2 Implementation details and what you must do

Compression Algorithm (Huffman encoding)

When using Huffman encoding for compressing a file, the compressed file must include information in order to reconstruct the Huffman tree necessary in the decoding; that is, we need somehow to inform the character frequencies so that the decoder constructs the tree in exactly the same way as the encoder (breaking ties for equal frequencies in a consistent manner). In this assignment this information is stored in **class LetterFrequencies**. If we were generating a file, we would need to write this information in the header of the file (for example, the first bytes of the file would contain a sequence of pairs (letter, frequency). Another issue when creating a stream of bits to write to a file is that these bits must be written as bytes, and as it can be seen in our first example, the number of bits might not be a multiple of 8. Thus the last byte will need to be filled with extra bits that are not part of the sequence. So that this does not produce an incorrect decoding of these extra bits, we will introduce a special symbol that we call **EndOfText** for us to know when the sequence ends ignoring any trailing bits used to complete a byte.
While the compression of files is a motivating application, in this assignment we will simply focus on the Huffman encoding and decoding parts. The bits generated will be stored as characters of a String for easiness of visualization and we will not be concerned with the part of packing the bits generated into bytes to be written to a file. (The interested student could take on this task of extending the current program to do file manipulation for their own interest).

Huffman Compression Algorithm (steps are in TestHuffmanWithStrings.testStringEncoding())

1) Read characters of the input text recording their frequencies by using the constructor of class LetterFrequencies:

```
LetterFrequencies lf = new LetterFrequencies(inputText);
```

2) Build the Huffman tree using the frequencies of letters of the input text. This must be implemented by you in the following method that is called by the constructor of the HuffmanTree class:

```
private HuffmanNode BuildTree(int[] frequencies,char[] letter)
```

This method must build the Huffman tree and return the root of such tree. The HuffmanTree is a binary tree with nodes as members of the class HuffmanNode. The pseudocode of this algorithm has been given in the previous page under the name Algorithm Huffman(X). As seen there, this will use a priority queue; methods to build a priority queue have been provided in class `HeapPriorityQueue` that implements interface `PriorityQueue`.

In addition to the tree, class HuffmanTree will also keep track of the leaves where each letter has been placed by using the leafWhereLetterIs array indexed by the character unicode (16 bits) which is a number between 0 and 2^{16} -1, plus the extra symbol we created "EndOfText" which will be coded by our convention as character 2^{16}. This array will be useful when encoding characters.

You have been given the initial portion of method HuffmanTree.BuildTree (Steps 1-5 have been implemented). You need to complete its implementation with steps 6-13 of Algorithm Huffman(X).

Note that printCodeTable() has been implemented for you, and will do an inorder traversal of the Huffman tree and print the codeword for each letter. This will help to verify correctness of your method HuffmanTree.BuildTree.

3) The encoding of the input text is done by traversing the input text again and encoding character by character doing individual calls to method:

```
HuffmanTree.encodeCharacter(char c );
```

This method must be implemented by you. Given a character you must determine the bits of its encoding and return them as a String. In the example, the result of

```
encodeCharacter('G')
```

will be "011".

The running time of this method should be O(L) where L is the length of the output code.

Note that the calls for appropriate methods to do the 3 tasks have been implemented in method testStringEncode(inputText) in class TestHuffmanWithStrings, which should be used as given. What you need to implement for encoding are methods BuildTree and encodeCharacter in class HuffmanTree.

Decompression Algorithm (Huffman decoding)

Huffman Decompression Algorithm:

1) Based on the letter frequency used in encoding, build the Huffman tree again. In our case, we have saved the letter frequency and we will use again. (If one was using this for file compression, we would read the character frequencies that would have been written in the header of the encoded file; this is not the case in this assignment).

2) Decode character by character from the stream of bits of the encoded text. This is done by several calls to method HuffmanTree.decodeCharacter, as it has been provided in by method TestHuffmanWithStrings.testStringDecode, which is shown next:

```
public static String testStringDecode(String codedText, LetterFrequencies lf) {
        HuffmanTree huffTree= new HuffmanTree(lf);
        BitFeedInForString seq=new BitFeedInForString(codedText);
        StringBuilder decodedText=new StringBuilder();
        while (seq.hasNext()) {
                int symbol=huffTree.decodeCharacter(seq);
                if (symbol == Integer.MAX_VALUE)
                        break;
                if (symbol!=HuffmanTree.EndOfText)
                        decodedText.append((char) symbol);
                else break;
                }
        return decodedText.toString(); // return the decoded string
        }
```

What you need to implement is the following method of class `HuffmanTree`:
`public int decodeCharacter(Iterator<Byte> bit)`

Successive calls to `bit.getNext()` will give you the bits of the codedText. Method `decodeCharacter` must use the Huffman tree with root stored in `root` and traverse it by using the bits until it reaches a leaf which corresponding to a letter; the integer code for such letter must be returned. Invoking `getLetter()` for that leave will give the integer code for the letter.
Any error encountered in this process must be indicated by returning Integer.MAX_VALUE, which is a value used by no letter or special symbol Huffman.EndOfText.
The running time of this method should be O(L) where L is the length of the code for the character found.

3 Assignment Tasks and Mark Breakdown (75 marks)

Examine the classes provided. In particular, you need to read the main program and other methods of class `TestHuffmanWithStrings` which are implementing the main steps of the encoding and decoding algorithms described above.
This class is not to be altered; additional texts can be used to test your program by yourself or your TAs by adding extra strings to array `String [] textsToTest` in the main program.

Part 1) Implement the required methods of HuffmanTree described in section 2 above, namely:

A. (25 marks) private HuffmanNode BuildTree(int[] frequencies,char[] letters)
B. (20 marks) private String encodeCharacter(int c)
C. (20 marks) public int decodeCharacter(BitFeedIn bit)

Part 2) (2 marks) Test your method with strings given. The TA may add more Strings when testing your code, so be sure to do extra tests in addition to the ones provided.
You should show the output with TestHuffmanWithStrings obtained in your program (file output.txt)

Part 3) (6 Marks) What is the running time of Algorithm Huffman(X) in terms of big-Oh as a function of the number of letters ? Answer this question for the following 3 implementations of priority queue Q. Briefly justify in each case.

A. Priority queue Q implemented using a **Sorted List**.
B. Priority queue Q implemented using a **Unsorted List**.
C. Priority queue Q implemented using a **Heap**.

Part 4) (2 marks) Submit strictly following the instructions below.

4 What to handin and submission instructions

1) Create one directory with name being the letter "s" followed by your student number.
If your student number is 564789 your directory will be called s564789

2) Inside the directory include the following:

- the source code of your program, i.e. **all the *.java classes for your program**. (these includes all files that we provided and your modified `HuffmanTree.java`)
- a file called **output.txt** showing the output obtained with the main program `TestHuffmanWithStrings`
- A file called **README.txt containing:**
  - Your answer to the questions in Part 3)
  - Optional: Any information relevant for the TA to mark your program. For example, if not all parts of the code are working you may inform the TA here which parts are working, explain known bugs, etc.

3) zip the directory containing all the files above and submit (in the example of the student number above the file submitted will be called s564789.zip) When the TA unzips the file, it should see the directory s564789 containing all files described above.