Project 4: Huffman Tree

CS 0445 — Data Structure

Due Sunday December 7, 2014 at Midnight

Introduction to Huffman Tree

In the past, computer storage was very limited. A diskette could only hold data in kilobyte. One way to allow a diskette to store a file that is larger than its capacity is to compress the file into a smaller file. One way to do so is to use a Huffman tree. In this project, we are going to focus on compression and decompression of a text using Huffman tree.

A text file consist of a series of characters. In ASCII format, a character can be represented by one byte (8 bits). To support international characters, each character must be represented by two bytes (16 bits) called Unicode character. Suppose our Java use Unicode characters. To compress a text file, what we need to do is to represent each character by something smaller than 16 bits. For example, if a text file contains only 60 different unique characters, there is no reason why we need to use 16 bits for each characters. Since there are only 60 different unique characters, 6 bits for each character is enough. Note that 6 bit give us 64 different values from 0 to 63. Thus, we can assign the characters `'a'` to 000000, `'b'` to 000001, and so on. Note that to be able to decompress the file back to the original text file, we also need to store this conversion table into the compressed file. Thus, a compressed file will consist of two parts; (1) a data structure representing the compression table of the file, and (2) a series of 0s and 1s that represent a series of character.

In previous method, a character will acquire a fix size number of bits. To compress a file with high compression ratio, we can use variable size representation instead. The idea is to represent often used characters with shorter number of bits and hardly used characters with longer number of bits. For example, vowels and space are often used in a text file, they may be represented by 4 or 5 bits but character `'w'` is hardly used, we can represented it using 9 or 10 bits. In doing so, a large text file can further compress down with higher compression ratio.

There are various way to represent conversion table. One way is to use a Huffman tree which is suitable for variable size representations. A Huffman tree is a binary tree of characters where inner nodes contain no characters or null character and each leaf node contains a unique character. An example of a Huffman tree is shown in Figure 1.

To compress a text file using the Huffman tree shown in Figure 1, we need to replace each character in the text file by a series of 0s and 1s and treat them as bits, not character. A series of 0s and 1s for each character is the path from the root node to the character. 0 represents going to the left child and 1 represents going to the right child from the current node. For example, the character `'a'` will be represented by 0111 since you need to go

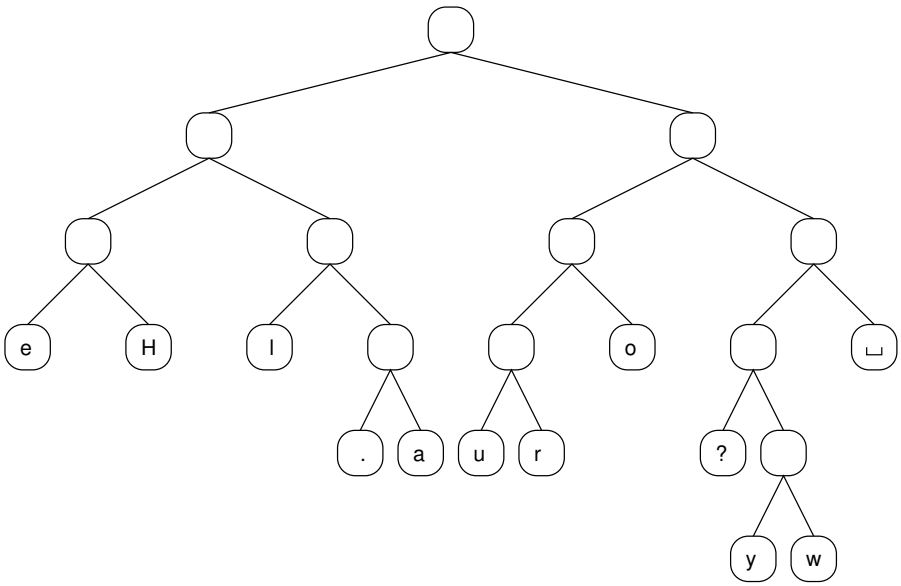

Figure 1: An Example of a Huffman Tree

left-right-right-right from the root node to reach the node containing the character `'a'`. Another example is the character `'r'` which will be represented by 1001 since you need to go right-left-left-right from the root node to reach the node containing the character `'r'`. Note that the character ␣ in Figure 1 represents the space character. For a larger example, consider a text file is as follows:

```
Hello. How are you?
```

After the above file is compressed using the Huffman tree shown in Figure 1, the result is as follows:

```
00100001001010101101110011011101111101111001000111110101010110001100
```

Note that the text file contains 19 characters (19 bytes for ASCII or 38 bytes for Unicode). After compression, the whole file is compressed into only 9 bytes (66 bits in the above example). However, the data structure of the Huffman tree must be included into the compressed file for decompression. This may result in larger file size compare to the original file. This is the reason why we generally compress large files, not small files.

Huffman Tree Data Structure

Huffman tree can be represented by a series of character. For example, the Huffman tree shown in Figure 1 can be represented using a series of characters as follows:

```
IIILeLHILlIL.LaIIILuLrLoIIL?ILyLwL_
```

In the above series of characters, there are a lot of characters `'I'` and `'L'` and the rest are unique characters. A character `'I'` represents an interior node (non-leaf node) and a character `'L'` represents a leaf node. The above series of character is generated according to **Preorder Traversal** as follows:

- Visit the root note, if it is a leaf node, append the result by the character `'L'` follows by the character in that node. Otherwise, append the result by the character `'I'`.
- Visit all nodes in the root left subtree.
- Visit all nodes in the root right subtree.

If we put the order of visiting in Figure 1 according to preorder traversal, we will get the result shown in Figure 2.

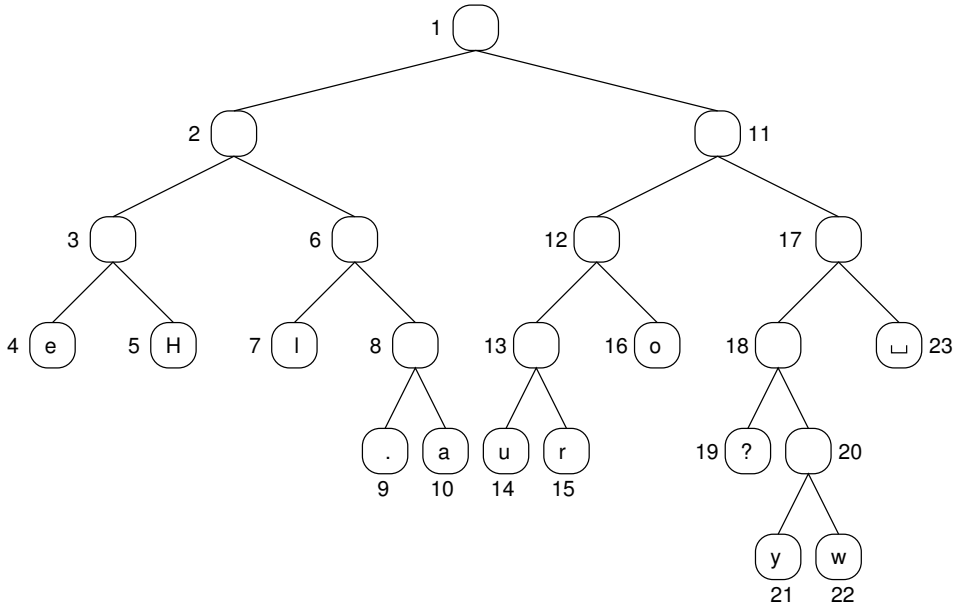

Figure 2: Preorder Traversal

Note that the Huffman tree will be unique for each text file. Since we want to achieve high compression ratio, we need to consider the frequency of each character in mind before generating the Huffman tree for a text file. The path to frequently used character should be short and the path to hardly used character should be long. For example, if we have a text file as follows:

```
aaaaaaaabbbbbbccccdde
```

The text representation of the Huffman tree for this text file will be

```
ILaILbIILeLdLc
```

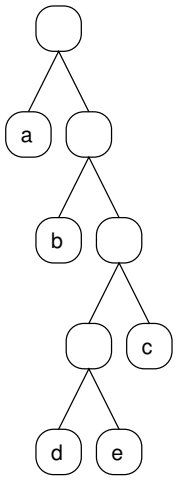

Figure 3: The Huffman Tree for `aaaaaaaabbbbbbccccdde`

which corresponds to the Huffman tree shown in Figure 3.

Since the frequency of the character `'a'` is the highest, the path to the character `'a'` should be the shortest. In the above tree the character `'a'` will be represented by one bit, 0. The frequency of the character `'b'` is high but not as high as the character `'a'`. Thus, the path to character `'b'` should be relatively short. In the above tree the character `'b'` will be represented by two bits, 10.

What to do

For this project, you have to finish the key elements of the Compression/Decompression application (`CompressionDecompressionGUI.java`) shown in Figure 4. This application consists of four sections as follows:

1. The top-left corner allows you to type in or cut and paste a content of a text file.
2. The top-right corner shows a series of characters representing the Huffman tree data structure of the text in the top-left corner.
3. The bottom-left corner shows the result of the compression of the text in the top-left corner using the Huffman tree in the top-right corner.
4. The bottom-right corner shows the result of the decompression of the data in the bottom-left corner using the Huffman tree structure in the top-right corner.

This Compression/Decompression application use static methods in `CompressDecompress.java`. This class contains three static public methods as follows:

1. `public static String getTreeString(final BinaryNodeInterface<Character> root)`: This method takes the reference of the root node of a Huffman tree of type `BinaryNodeInterface`

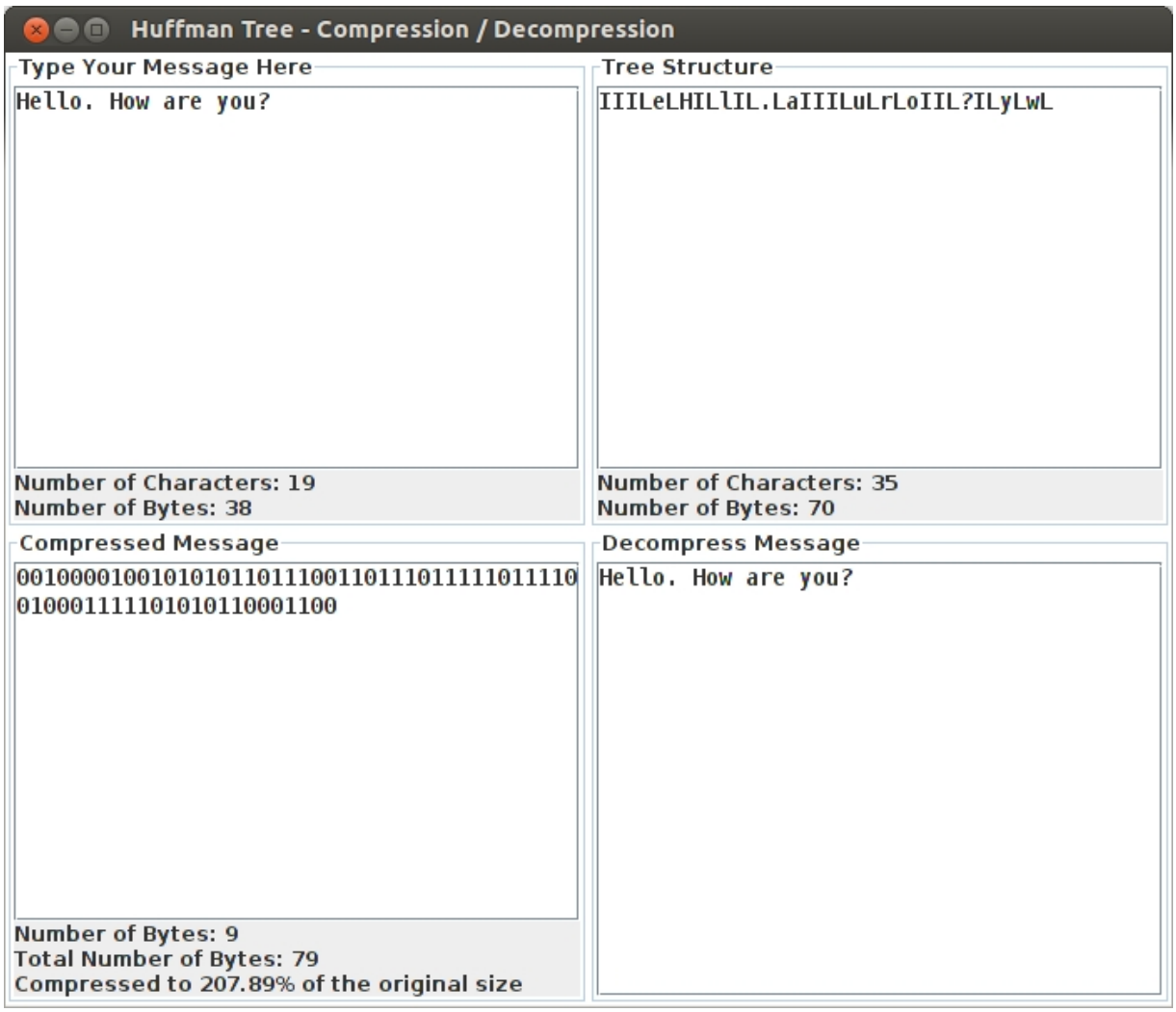

Figure 4: Compression / Decompression Application

as an argument and returns a string representation of the Huffman tree data structure. The returned string will be shown in the top-right corner of the Compression/Decompression application.

2. `public static String compress(final BinaryNodeInterface<Character> root, final String message)`: This method takes the reference of the root node of the Huffman tree and the string (from top-left corner) as arguments and returns the string representation of compressed data (series of 0s and 1s). The returned string will be shown in the bottom-left corner of the Compression/Decompression application.

3. `public static String decompress(final String treeString, final String data)`: This method takes the string representation of a Huffman tree (from top-right corner), and a string representation of a compressed data (bottom-left corner) as arguments, and returns the string representation of a decompress data. The returned string will

be shown in the bottom-right corner of the Compression/Decompression application.

The Compression/Decompression application uses the above three methods in the method `keyReleased` in the inner class named `InputKeyEvent` as shown below:

```
private class InputKeyEvent implements KeyListener
{
   :
   public void keyReleased(KeyEvent arg0)
   {
      :
      HuffmanTree ht = new HuffmanTree(str);
      BinaryNodeInterface<Character> root = ht.getRootNode();
      :
      String treeString = CompressDecompress.getTreeString(root);
      :
      String compressedString = CompressDecompress.compress(root, str);
      :
      String decompressedString = CompressDecompress.decompress(treeString,
                                                                compressedString);
      :
   }
   :
}
```

The class `HuffmanTree` generate a Huffman tree based on the string `str` (from top-left corner). **This class is given (`HuffmanTree.java`)**. So, you do not have to generate the Huffman tree yourself. The method `getRootNode()` of the class `HuffmanTree` returns the reference of the root node of the Huffman tree. Then the application will send the reference of the root node of the Huffman tree (`root`) to the method `getTreeString()` of `CompressDecompress` class to obtain the string representation of the Huffman tree data structure. Note that this string will not be used in compression process. Just imagine that it will be used to include with the compressed file. Then the application will send the reference of the root node of the Huffman tree (`root`) and the string of the text (`str`) to the method `compress()`. You can image that the task the method `compress()` is to compress the text. So, it needs the root node of the Huffman tree and the string to be compressed. After the application get the string representation of the compressed data (`compressedString`), it will send this together with the string presentation of the Huffman tree data structure (`treeString`) to the method `decompress()` to decompress the data. Note that the `decompress()` method should not receive the root node of the Huffman tree as an argument, it **MUST** construct the Huffman tree from the Huffman tree data structure (`treeString`) and uses it for decompression.

So, your job is to finish methods `getTreeString()`, `compress()`, and `decompress()` marked `TO DO` in `CompressDecompress.java`. `BinaryNodeInterface.java` and `BinaryNode.java` are provided. **Do not modify signatures of any of those three methods. You are not allowed to have any instance/global variables in side `CompressDecompress.java` but variables inside methods are fine. You are NOT allowed to use any Java predefine classes other than `ArrayList`.**

Helpful Hints

Compression

Note that to locate a specific character in a tree is not quite easy because characters are arranged in no particular order. An easy way to do this is to create an encoding table. This can easily be done by recursive calls and backtracking. For the tree shown in Figure 1, the encoding table should some what like the following:

Character	Code
e	000
H	001
l	010
.	0110
a	0111
u	1000
r	1001
:	:

Then use the above table to compress the message.

Decompression

As you see in the example, the compressed data is a long series of 0s and 1s. There is no indicator to tell us which subseries of 0s and 1s belongs to which character. Note that all unique characters are stored in leaf nodes. Thus, what you need to do is to traverse the tree starting from the root node; if the data is 0, go to left child, and if the data is 1, go to right child. As soon as you hit a leaf node, that is your character and you restart from root node again until you run out of compressed data.

Test Class

A test class `CompressDecompressTester.java` is provided. However, the tree structure is quite complex. It is not easy to test every possible situation. You may have to write your own test class for a specific situation that you want to test.

Submission

This assignment is due on Sunday, December 7 at midnight. Late submission will not be accepted. Zip all files, `CompressionDecompress.java` and any additional classes that you created into one single file and submit it to CourseWeb. If you have only one file, simply submit the `CompressDecompress.java`.