CS1027a Computer Science Fundamentals II

Assignment 3 - Summer 2019

Learning Outcomes

By completing this assignment, you will gain skills in:

- using a Binary Tree ADT, an Ordered List ADT, and an Unordered List ADT
- understanding an important application of a Binary Tree
- understanding the linked implementation of the Binary Tree ADT

Background on Huffman Coding

Text compression is an important technique in computer science. It can be used to reduce the size of files stored on disk, making better use of disk storage. It also makes it practical to send large amounts of data, in compressed form, across networks. There are many techniques for text compression; one such technique is called Huffman encoding, invented by D.A. Huffman at MIT in 1952. It is an encoding at the basis of the Unix compress utility and part of the JPEG encoding process.

In the character encodings with which we are most familiar, the same number of bits are used to represent each character: 8 bits for ASCII, 16 bits for Unicode. For example, the 8-bit ASCII code for the character A is 01000001, for the character B is 01000010, etc. Unicode, which is used by Java, encodes A as 0000000001000001. Clearly there is a waste of space with the encoding of these characters in Unicode. But even with ASCII code, the same number of bits is used for each character, whether it is used frequently or not. One way to save space is by reducing the number of bits for some characters, specifically the most common ones such as A and E, with a longer bit pattern for the less common letters such as Q and X.

Huffman coding compresses text by coding the component symbols of a file (letters of the alphabet, spaces, etc.) based on the frequency of their occurrence. The more often a symbol occurs, the shorter its code. One issue with variable length codes is how to know where a code ends and where the next one starts. This is not a problem if no symbol's code is the start of the code for another symbol, and the Huffman code has this property.

The way that Huffman coding works for a set of symbols is that it builds a binary tree (called the Huffman tree) based on the frequencies of the symbols, and the codes can then be read off the tree, as explained below. We will first demonstrate how a Huffman tree is built with an example:

consider the following set of characters and their frequency of use in some file, sorted into ascending order by frequency:

Symbol	Frequency
A	5
B	7
C	10
D	15
E	20
F	45

We will build a binary tree made up of nodes that have storage for two data items: a symbol and the frequency of the node. It will turn out that the symbols themselves will be stored only at the leaf nodes (the symbol field has no meaning in the interior nodes).

We first make a Huffman tree consisting of just a root containing a Huffman pair for every symbol/frequency pair in our set and add them to an ordered list of nodes such that the nodes are in ascending order by frequency. The ordered list will look like this:

A, 5 B, 7 C, 10 D, 15 E, 20 F, 45

We start building the Huffman tree by removing the two nodes with the lowest frequencies from the ordered list and constructing a binary tree with these nodes as leaf nodes and a parent node with a frequency that is the sum of the two lower nodes' frequencies. (The symbol field in an interior node has no meaning and will be denoted by the character "*" in this example.) The resulting binary tree looks like this:

```
     *, 12
      /   \
  A,5     B,7
```

We now insert the new parent node, with frequency 12, into the ordered list *at the proper location so that the list is still ordered by frequency.* So now the list contains:

C, 10 *, 12 D, 15 E, 20 F, 45

Repeat the previous steps, combining the two lowest-frequency nodes into a binary tree with a new root node.

This results in the binary tree:

```
          *, 22
          /    \
     C, 10      *, 12
                /    \
             A,5    B, 7
```

and the list now contains:

```
D, 15     E, 20     *, 22     F, 45
```

Repeat until there is only one element left in the list. The binary trees and the ordered list are shown at each step:

```
     *, 35
     /    \
D, 15   E, 20

List:    *, 22    *, 35    F, 45
```

(note that we temporarily have a second binary tree)

```
           *, 57
       ____/   \____
      /             \
   *, 22             *, 35
   /    \            /    \
C, 10   *, 12     D, 15   E, 20
        /    \
     A,5    B, 7

List:    F, 45    *, 57
```

(the second tree of the step above has now become part of the main tree)

```
                         *, 102
         _______________/   \_______________
        /                                   \
     F, 45                                  *, 57
                                        _____/  \_____
                                       /              \
                                    *, 22              *, 35
                                    /    \             /    \
                                 C, 10   *, 12      D, 15   E, 20
                                         /    \
                                      A,5    B, 7

List:    *, 102
```

Since the list now only contains one node, we are done. This node is the root of the Huffman tree. **Note that the symbols are all in the leaf nodes of the tree, and that the symbols with the lowest frequencies are furthest from the root of the tree.**

To get the binary Huffman encoding for a symbol:

- start with an empty encoding, at the root of the tree
- traverse the tree to the node with the symbol, appending a 0 to the encoding every time you take a branch to the left, and a 1 every time you take a branch to the right. For example, the encoding for the symbol E is 111 and the encoding for the symbol F is 0. Because each symbol is at a leaf, the encoding for one symbol cannot be the start of the encoding for another symbol; therefore, the Huffman encoding is unambiguous.

To decode a Huffman encoding:

- as you get bits from an input stream, traverse the tree beginning at the root, taking the left-hand path if the bit is 0 and the right-hand path if the bit is 1. When you encounter a leaf, you have found the symbol that was encoded.

Assignment 3 Provided Classes:

The file **Assmt3Files.zip** contains all the classes that are provided with this assignment. Use the classes from this provided zip file. You are not allowed to make any change to any of these provided classes. We proceed to briefly describe them, one by one:

- **Class ArrayIterator.java**: Unchanged from course website
- **Class ArrayList.java**: A method public T getElement(int i) to get the i-th element from the list was added
- **Class ArrayOrderedList.java**: Unchanged from course website
- **Class ArrayUnorderedList.java**: Unchanged from course website
- **Interface BinaryTreeADT.java**: modified to mirror LinkedBinaryTree.java
- **Class BinaryTreeNode.java**: modified such that the attributes are private with their getter and setter methods implemented. A method public Boolean isLeaf() has been added. It returns true if a node is a leaf and false otherwise.
- **Class CompressedFile.java**: explanation inside the file
- **Class ElementNotFoundException.java**: Unchanged from course website
- **Class EmptyCollectionException.java**: Unchanged from course website
- **Class Huffman.java**: new, provided class containing the main method for the assignment (explanations to be found below)
- **Class HuffmanPair.java**: new, provided class (explanations to be found below)
- **Class LinkedBinaryTree.java**: Modified such that the attributes are private with their getter and setter method implemented
- **Interface ListADT.java**: Unchanged from course website

- **Class OrderedListADT.java**: Unchanged from course website
- **Class TextFile.java:** explanation inside the file
- **Class UnorderedListADT.java**: Unchanged from course website
- **Class TestEncodingData**: This class is provided to you such that it tests your EncodingData class
- **Class TestHuffmanTree**: This class is provided to you such that it tests your HuffmanTree class
- **Class TestHuffmanEncoder**: This class is provided to you such that it tests your HuffmanEncoder class

Note that the tests performed by these last three classes are **not exhaustive**. In other words, even though a class may pass its tests, it still may contain errors.

Assignment Tasks

Write a program according to the specifications found below, that:

- compresses a text file into a Huffman-encoded file, generating the symbol frequency file for that particular text file
- decodes a Huffman-encoded file into a text file, using the symbol frequency file for that particular file

Functional Specifications

Your implementation will contain the following classes, according to the specifications given in this document:

- EncodingData.java
- HuffmanTree.java
- HuffmanCoder.java

Class HuffmanPair.java:

This class is provided to you and represents a single pair of the data items used in building a Huffman tree, i.e. a symbol and its frequency. This class contains:

Attributes:

private char symbol: a symbol that is to be encoded
private int frequency: the frequency of the symbol

Constructors:

public HuffmanPair(char symbol, int frequency)
public HuffmanPair(int frequency)to be used when the symbol is irrelevant. When the symbol is irrelevant, then it must be set to (char) 0.

Public methods:

- getCharacter(), getFrequency(), setCharacter(), setFrequency() setter and getter methods for both attributes
- incrementFrequency() method
- toString() method

- equals(Object obj) method, that works on the symbol
- attribute (note how it is implemented)
- compareTo(HuffmanPair otherPair) method, that compares two HuffmanPair objects based on the frequency attribute. This will be used by the ordered list of Huffman pairs, such that the list orders the Huffman pairs by increasing frequency.

Class EncodingData.java:
This class is for you to write and represents a single pair of the data used to encode a text message into its binary Huffman code (a symbol and its corresponding binary Huffman code). It will be used for encoding a text file. Your implementation of the EncodingData class must include the following, with the method headers as specified:

Attributes:
private char symbol: a symbol that is to be encoded private String encoding: the binary Huffman code of the symbol (i.e. a string of 0's and 1's)
Constructor:
public EncodingData(char symbol, String encoding)
Public methods:

- getSymbol(), getEncoding(), setEncoding(): getter and setter methods for both attributes
- public boolean equals(Object obj): method that determines if two EncodingData objects are equal based on the symbol attribute (implement it in the same fashion as the equals(Object obj) in the class HuffmanPair)
- public String toString(): method that gives a string representation of the symbol and its Huffman code

Class HuffmanTree.java:
This class is for you to write and represents a Huffman tree, and will extend the LinkedBinaryTree<T> class. It will also implement the Comparable interface. Your implementation of the class public class HuffmanTree extends LinkedBinaryTree<HuffmanPair> implements Comparable<HuffmanTree> must follow the specifications below, with the method headers as specified.

Constructors to write:
The HuffmanTree class will have 4 constructors and the first three will call the appropriate constructor from the superclass (in this case, the LinkedBinaryTree class):

- public HuffmanTree(): creates an empty Huffman tree
- public HuffmanTree(HuffmanPair element): creates a Huffman tree with one Huffman pair at the root
- public HuffmanTree(HuffmanPair element, HuffmanTree leftSubtree, HuffmanTree rightSubtree) creates a Huffman tree rooted at a node containing

element, where the roots of the left subtree and right subtree are its left child and right child, respectively

The 4th constructor is:

- public HuffmanTree(ArrayOrderedList<HuffmanPair> pairsList)

The pairsList parameter is an ordered list of Huffman pairs in ascending order by frequency (the file containing the symbols and their frequencies for a text file is built for you by the Huffman class (provided) when the file is encoded. The same frequency file is used when the text file is decoded). This constructor will build the Huffman tree from this ordered list of Huffman pairs.

In this constructor, the Huffman tree will be built using the following algorithm. Wherever the algorithm refers to "trees", it means Huffman trees:

1. For each HuffmanPair in pairsList, make a HuffmanTree consisting only of a root node containing the HuffmanPair, and add each tree to a temporary ordered list buildList.

2. While there is more than one item in buildList

    - Remove the 2 trees with the lowest frequencies in their root node from buildList(**Hint**: they will be the first 2 trees in the list)
    - Construct a new tree such that the first tree that was removed from the list is the left subtree, the second tree removed is the right subtree, and the root node of the new tree contains a new Huffman pair whose frequency is the sum of the frequencies in the Huffman pairs of the two children. (**Hint:** use the third constructor for a Huffman tree)
    - Add the new tree into the ordered list buildListsuch that the frequencies are maintained in ascending order (**Hint**: the add method will insert it in the proper place)

3. The tree left in buildList is the final Huffman tree

Pay attention to the case when there is only one HuffmanPair. This happens when a text file contains one or more instances of the same character. In this case, the resulting Huffman tree has only one root node and does not provide any decoding path. Find a simple way to deal with this problem.

Methods to write:

1. public int compareTo(HuffmanTree otherTree): This is the compareTo method specified in the Comparable interface, so that you will need to have the

HuffmanTree class implement the Comparable interface. It will compare the frequencies in the root node of the trees, so that the add method of ArrayOrderedList can put that node in its correct place in the ordered list buildList in the algorithm to build a Huffman tree.

2. public String toString() : This method will return a string representation of a Huffman tree by doing a preorder traversal of the tree. It overrides the toString method of the LinkedBinaryTree class

Class HuffmanCoder.java:

This class uses a Huffman tree for encoding a character and decoding a code string. Your implementation of the HuffmanCoder class must follow the specifications below, with the method headers as specified.

Attributes:

- private HuffmanTree huffTree: the Huffman tree
- private ArrayUnorderedList<EncodingData> encodingList: an unordered list of encoding data that will be used for encoding a text file into a Huffman-coded compressed file

Constructor to write:

- public HuffmanCoder(ArrayOrderedList<HuffmanPair> pairsList): this constructor will create the huffTree, using the 4th Huffman tree constructor. It will also call the private helper method buildEncodingList(BinaryTreeNode<HuffmanPair> node, String encoding) which will build the list of symbols and their encodings from the Huffman tree huffTree. See below for further instructions.

Methods to write:

- public char decode(String code): This method will take the specified string of binary digits that is a Huffman encoding, and will return the original coded character. It will use the following decoding technique: as you get each character from the parameter string, traverse the tree beginning at the root, taking the left-hand path if the character is 0 and the right-hand path if the character is 1. When you encounter a leaf, you have found the symbol that was encoded. If the string of binary digits code does not lead to a leaf node, then the method must return (char) 0.
- public String encode(char c)throws ElementNotFoundException: This method will take the specified character and return the string representation of the binary Huffman encoding of that character.
- public String toString(): This method will return a string representation of the encoding list.
- private void buildEncodingList (BinaryTreeNode<HuffmanPair> node, String encoding): This method will build the unordered list encodingList (an attribute of the HuffmanCoder class) from the Huffman tree huffTree. The list encodingList will be a list of EncodingData objects. This method will add new EncodingData objects to encodingList recursively, following this algorithm:

```
if the parameter nodeis a leaf node
    add a new EncodingDataobject whose symbol is the character in the
    Huffman pair of that node and whose encoding is the second parameter
    encoding
else
    call buildEncodingListwith the left child of nodeand
encodingwith character '0' appended
    call buildEncodingListwith the right child of nodeand
encodingwith character '1' appended
```

Hint: for the first call to buildEncodingList, the parameters should be the root node of huffTree and the empty string.

Class Huffman.java:

This class is provided to you and contains the main method for this assignment. This class requires four arguments that need to be supplied to the main method. For example, if you want to encode the file text1.txt, you must execute the program with the following arguments: Huffman – c text1.fre text1.txt text1.cmp where –c means to program will encode file text1.txt into text1.cmp, while creating the symbol frequency file text1.fre for file text1.txt. After text1.txt has been compressed, it is possible to decompress it using its corresponding symbol frequency file in the following way: Huffman – d text1.fre text1.dcp text1.cmp where the decompressed file text1.dcp is produced with the help of text1.fre. Note that the contents of text1.dcp and text1.txt should be identical in every way.

Non-functional Specifications

- Assignments are to be done **individually** and **must be your own work**. Group work is not tolerated. A software tool may be used to detect plagiarism.
- All implemented programs have to perform exception handling if necessary.
- Include comments in your code in javadoc format. Add javadoc comments at the beginning of your classes indicating who the author of the code is and giving a brief description of the class. Add javadoc comments to methods and instance variables. Read information about javadoc in the second lab for this course.
- Use Java coding conventions and good programming techniques.
- Include a comment identifying yourself (name and student number, uwo email address) and the assignment number, course and year at the beginning of all your class files. For example, the professor has:

```
//   Ahmed Ibrahim 123456789                          //
//   ahmed@csd.uwo.ca or ahmed@uwo.ca                 //
//   Assignment 3, CS1027, Spring 2019                //
```

- Make sure your code runs using Eclipse's Java, even if you do not use Eclipse to write your code.

- **Zip** all the files used by your program **(project folder)**. Use your first and last names and student number and assignment number and year (with linking of the tokens in the zip filename done by underscores). For example, your professor's files would be in

  Ahmed_Ibrahim_123456789_Assignment_3_Spring_2019.zip.

- Assignment 3 due on Friday, July 26, at 5:00 pm. You need to submit all your .java files as a single zip file. And, it's your responsibility that the Zip file is not empty or damaged.
- You can submit your assignment more than once and up to three times through OWL. If you submit the assignment more than once, we assume that the latest submission is the one you want us to mark.
- It is your responsibility to submit your assignment **zip** file before the due date and in the dedicated drop-box for the assignment.
- No submission allowed through emails.

What You Will Be Marked On

Functional specifications:

- Does the program behave according to specifications?
- Does it run with the test input files provided?
- Are your classes created properly?
- Are you using appropriate data structures?
- Is the output according to specifications?

Non-functional specifications: as described above

Assignment submission: via OWL assignment submission.

Test Programs and Data

You are provided with seven text files that first need to be compressed and then decompressed. The original text files, once compressed, must decompress to their identical selves. For each class you must program, you are provided with a test class to verify its correctness. See the course assignment web page for the text files and the test programs.

Good Luck