**Purpose**

To learn another Binary Tree ADT in addition to the BST and AVL.

To learn how to use the Map ADT.

**Background**

A map ADT is a set of pairs. Each pair is composed of a key and a (mapped) value. Keys within a map must be unique (no duplicates). The (mapped) values may be duplicates.

**Requirements**

Develop an encoding/decoding program for any number of characters that can perform all of the operations described below. This is completed by creating a Huffman tree (see section 8.6 of the textbook).

To complete this lab, you must create a class called HuffmanTree that implements the HuffmanInterface as well as two more classes that inherit from TreeInterface and NodeInterface respectively, similar to Lab 6. You will not be able to pass off this lab if you do not implement the interfaces or if you name your class differently. These interfaces will allow us to automate and expedite the pass-off process. You may use any additional methods or classes as you see fit and this is encouraged. Do not modify the interfaces in any way.

Part1 – Create the Tree (10 points)

* Implement the createTree and getTree methods of Huffman Interface.
* Determine the frequency of occurrence of each character, including punctuation marks and whitespace characters, by scanning some sample text. Sample text is a string that ispassed in as a parameter.
* Follow the “Rules for Your Tree” in the Requirement Notes below when constructing the tree. This test guarantees that your tree conforms to these requirements.

Part 2 – Make the Encoding (6 points)

* Implement the getEncoding method of the HuffmanInterface .
* Determine a minimal Huffman encoding using the “Rules for Your Tree” in the Requirement Notes below.
* Use a Map data structure to store the encodings with the character as the Key and the encoding as the value mapped to by the key.

Part 3 – Encode (6 points)

* Implement the encodeMessage method of the HuffmanInterface. Your encoded messages should contain 1's and 0's and nothing else.

Part 4 – Decode (6 points)

* Implement the decodeMessage method of the HuffmanInterface.

UML design document. (2 points)

Refer to page 113 in the book for an example.

Please bring when asking TAs questions.

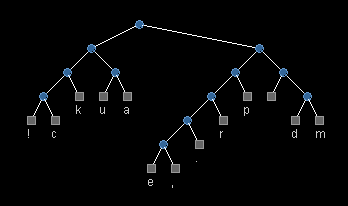
**Requirement Notes**

**General**

* You are required to create a separate '.cpp' and '.h' for each class you implement for this lab.
* After creation of the tree, use a map to store the encoding that your tree generates with the character as the key and the encoding as the mapped value
* Your program needs to be “bullet-proofed”. IT must accommodate all possible input, even input which is not specifically mentioned in the specifications. This requires you as the programmer to anticipate what problems could arise from invalid input and to handle them appropriately.

**Rules for Your Tree**

* Here is a sample text (brackets not included): [muckduck! muuuud draumak pmdap dp ucmr, paumeu dpapkam. mka. rpkuakdap pamk aprckpku.arp pruak, dcd ckap! r! p! .!mrp]. Here is a visual tree created from the sample text above (the node that appears blank on the right side of the tree is for the space character):



* Several different strings will be used to test your solution. For your convenience when testing, we have provided three files: the sample text used above “sample.txt”, a message “message.txt” and the encoding of that message “encMessage.txt” using the tree generated from“sample.txt”.
* For any given text, there could be multiple Huffman trees that are all optimal. This could lead to problems if two parties want to exchange messages that are encoded with trees generated using the same sample text. If each party generates a different, but still optimal, tree, they would end up with nonsensical decoded messages. To avoid this problem for your solution, we will use a set of arbitrary, but necessary, rules that will specify how various aspects of the tree should be handled.
  + The left branch of a node will use a 0 in the encoding and the right branch will use a 1.
  + When combining two nodes to make a new node in your tree, the node with the lower frequency should be placed on the left and the node with the higher frequency should be on the right.
* In situations where there are duplicate frequencies, a few rules need to be used to create the ordering of your priority queue. Original characters should come before combined nodes. When ordering two original characters, the ordering of the ASCII character set should be used. When ordering two combined nodes, the older node should come first.