**Programmer Manual**

**Class Tree for Huffman Code**

1. **Problem Description**

A Tree consists of a TreeNode that is templated. Its purposes is to allow for a tree construction or deconstruction of any size, in order to create an encoded table and message for any input message(following specific input restrictions).

1. **Class Tree**

The Tree class is a user-defined class, utlizing the stl database, allowing the use of a list to create an organized representation of character frequency in a file. Tree is templated, allowing for flexibility of the type of TreeNode used to comprise the Tree.

For example, in the user’s program

Tree<TreeNode<BaseData>> tree\_obj; //declares a Tree object called tree\_obj of type Tree.

Public data members: (note: only newly created functions are listed, all given functions are omitted)

void make\_tree(list<TreeNode<BaseData>> mylist) - creates a tree from a list of TreeNodes made of info. Prints Preorder traversal to display Huffman Tree after completion.

list<TreeNode<BaseData>> getlist() - read in from user input file while character isn't a \* symbol. adds letter read in to std::list<info>. increments weight of letter with each successive add. returns mylist for use elsewhere.

void print\_Huff\_table(list<TreeNode<BaseData>> mylist) – displays the Huffman Table

void encode(TreeNode<BaseData>\* p, string code) - Encodes a message by following the Huffman tree - "1" is left, "0" is right

void store\_table() - asks user to input a filename to store Huffman table in, stores table.

void store\_msg() - asks user to input a filename to store Encoded message in, stores encoded message.

void decode() - Asks user to input two files - one for the Huffman table, and one for the encoded message. Creates a temp array of info to hold new Huffman table info from user file. Reads through encoded message, using given Huffman Table.

1. **High Level Program Solution**

make\_tree(list<TreeNode<BaseData>> mylist)

IN: list<TreeNode<BaseData>> mylist – list used to organize and sort relevant data to make Huff Table/tree

Declare required data type variables.

WHILE the size of the list is not 1

Declare dynamic temp1 values

Set temp values to front of list values in all members related

Pop front two items from the list and hold in temp value, used to create a new tree

Set sum of new node to sum of each temp value’s weight

Get new node composed of temp1,temp2, and it’s weight

Push new node to back of the list

Sort the list

Set Root to the front of the list

Encode the tree

Display the Huffman tree in an Inorder traversal

list<TreeNode<BaseData>> getlist()

IN: none

Declare required data type variables

Prompt user for a text file name

Open filename

If Filename cannot be opened

Exit program and display termination message

Read in character

WHILE character does not equal a ‘\*’ symbol

IF character is not a ‘space’

Set temp to char value, append temp to message member

FOR beginning of list to end of list

IF current list character is equal to read in character

Increment weight of current list index, set inlist variable to true

IF inlist is set to false

Increment weight of character read in, and push to back of list

Read in new character

Reset character weight to 0

Reset inlist to false value

FOR entire length of message string

Display current index of message

Display ending ‘\*’ symbol for formatting

Close input file

Add all letters from alphabet that were unused to the list

SORT the list

RETURN the list

void print\_Huff\_table(list<TreeNode<BaseData>> mylist)

IN: list<TreeNode<BaseData>> mylist – list to pass in so it can be displayed

Display output message

FOR list beginning to list end

Display current list character as well as current list weight

void encode(TreeNode<BaseData>\* p, string code)

IN: TreeNode<BaseData>\* p, string code – to hold the node, and the code for that node

IF p does not equal nullpointer

Call encode recursively, passing in p->left, and code + ‘1’ because left convention adds a 1 to the code for that item

IF p->letter does not equal a ‘\*’ symbol then set current p code to passed in code value

Call encode recursively, passing in p->right, and code + ‘0’ because right convention adds a 0 to the code for that item

void store\_table()

IN: none

Declare required data type variables

Display message asking for user to enter file name

Input filename

FOR 26 iterations

Display to output file current list character as well as current list weight

Close output file

void store\_msg()

IN: none

Declare required data type variables

Display message asking for user to enter file name

Input filename

FOR 26 iterations

Display to output file current list character as well as current list weight

Close output file

void decode()

IN: none

Declare required data type variables

Ask user to enter a Huffman table input file and then read in

IF input file can’t be opened

Display output termination message

Read in letter

WHILE letter does not equal ‘\*’

Ignore 5 spaces or until a ‘ ‘

Get the input code for a letter from the first input file

increment index

read in next letter

close input file 1

display decompression message

display question asking user to input file name for the compressed message

read in

open second input file

IF input file 2 can’t be opened

Display output termination message

Read in letter

WHILE character does not equal ‘\*’

Set temp equal to temp plus the new character

FOR 26 iterations

IF read in code is equal to temp

IF read in letter does not equal ‘\*’

Display read in letter

Set temp to empty string

Read in next character

Close input file 2

1. **Limitations and Suggestions**

The limitations of this class are that the input files required for all the encoding functions must be specifically formatted. The input file for the unencoded message must be all lower case and must end with a ‘\*’ character. The input file for the Huffman table must be in a specific order. The encoded message must also end with a ‘\*’ character.