CS 215 - Fundamentals of Programming II
Spring 2020 – Programming Project 6
30 points

Out: April 15, 2020
Due: April 27, 2020

This project is to build a Huffman coding tree and use it to encode and decode messages.

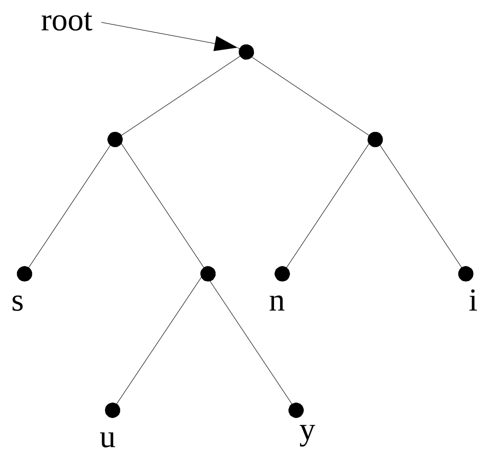
Huffman Codes

As you should know, characters are encoded in a computer in ASCII format. The ASCII format assigns 8 bits to each character. This makes it easy to determine where the bits for one character start and end in a file. Although ASCII is convenient, it also is wasteful of space. If we look at a file of English text, we can see there are far fewer than 256 distinct characters. (256 is the maximum number of characters 8 bits can encode.) To save space, it is possible to compress a textfile by encoding each character with fewer than 8 bits. The assignment of such an encoding can be done in various different ways.

One way is to design a Huffman code. A Huffman code is a variable-length encoding scheme (i.e., the code for each character is not necessarily the same length) and has the prefix property (no sequence of bits that represents a character is a prefix of a longer sequence for some other character). This allows the code to be represented using a binary tree (called a codetree) with the following properties:

- Every node is either a leaf or has exactly 2 children.
- Letters appear only at the leaves of the tree.
- Each letter appears at most once in the codetree; thus there is a unique root-to-leaf path encoding for each letter.

For example, consider the following codetree:



The code for each letter is a string formed by appending a `'0'` when taking a left branch and a `'1'` for a right branch when traversing the root-to-leaf path. In the codetree above, the code for `'u'` is `"010"` (left, right, left), the code for `'s'` is `"00"`, and the code for `'n'` is `"10"`. A message is encoded by appending the codes for letters in the message together. For example, the code for the message `"sun"` is `"0001010"`, which is formed by appending the codes for `'s'`, `'u'`, and `'n'` together.

The development of a Huffman code is left as a research exercise for the reader. It is based on symbol frequency such that the shorter length codes are assigned to symbols with higher frequency, resulting in an optimal (i.e., fewest bits) encoding for texts exhibiting the same frequency distribution. The code construction algorithm builds the codetree from these frequencies. (There is a discussion of this in the zyBook, Chapter 10.1.)

For this project, you will be given codefiles containing a Huffman code. Your assignment is to write a HuffmanTree class (specification given below) that can read a codefile and construct a codetree for use in encoding and decoding messages, and write a (main) program to encode and decode messages using a HuffmanTree object.

Specification of Node struct

This project uses a Node struct similar to the one in `bst.h`. (It does not have a parent pointer and is not in a template.) The definition should be included in a private section of the HuffmanTree class definition file (`huffman.h`) as follows:

This definition should be included in a private section of the the HuffmanTree class definition file (`huffman.h`) as follows:

```
class HuffmanTree
{
   private:
      // Node struct definition
      // Attribute declarations
   public:
      // Operation prototypes
   private:
      // Helper function prototypes
};
```

The Node struct is used to implement a node of Huffman codetree.

Data Attributes

Object	Type	Name
letter	char	data
link to the left child	Node *	lchild
link to the right child	Node *	rchild

Operations
Explicit-value constructor with default arguments – initializes the attributes of the Node struct. This function is to be in-lined in the struct definition.

- Analysis

Objects	Default argument	Type	Movement	Name
letter	'*'	char	received	initial_letter
link to the left child	nullptr	Node *	received	initial_lchild
link to the right child	nullptr	Node *	received	initial_rchild

Specifications for HuffmanTree Class

Attributes
There is only one attribute in this class, a single Node pointer variable that points to the root of the constructed codetree.

Operations
(Note: there is no default constructor to this class. It does not make sense to be able to create an empty codetree.)

Explicit-value constructor - builds the codetree from an input stream connected to a codefile.

- Analysis

Objects	Type	Movement	Name
input stream connected to codefile	istream	received & passed back	codefile

The codefile will consist of an integer *n*, followed by *n* lines of the form `letter code` where code is the Huffman encoding for the letter. There may be two special "letters", `%` and `$`, that represents the space character and the newline character, respectively, in the codefile (since we cannot distinguish a space or a newline as a message symbol versus a space or a newline as a delimiter between codefile items). The constructor is to read the codefile and build the codetree. For example, the codefile for the codetree above would be:

```
5
i 11
n 10
s 00
u 010
y 011
```

The basic algorithm is to read each letter (as a character) and its code (as a string) and use the code to traverse the tree (starting with the root) as one would do when decoding. If an interior node does not exist, create it and continue. When the end of the code is reached, you are at the leaf node that stores the letter as its data value. To aid in displaying the tree, the interior nodes should be constructed containing the character `'*'` for their data values. **The data value for the node containing the space character should be `' '` (a space, not `%`), and the data value for the node containing the newline character should be `'\n'` (a newline, not `$`).**

write - writes out the codetree “sideways” to an output stream using the recursive write_tree helper function.

- Analysis

Objects	Type	Movement	Name
output stream	ostream	received & passed back	out

write_tree – **a private** recursive helper function to write out a (sub)tree to an output stream using an RNL traversal (visit right child, visit node, visit left child – so that the tree looks correct when the output is displayed turned 90 degrees to the left). **Empty trees do not produce any output.** Each node value should be written on a separate line with (8*level) number of spaces before it (where the root value starts at level 0). The space character in a codetree should be written (displayed) as `%` (not as `' '`), and the newline character should be written (displayed) as **`$`** (not as **`'\n'`** or the actual newline character).

- Analysis

Objects	Type	Movement	Name
output stream	ostream	received & passed back	out
root of tree to write	Node*	received	local_root
level of node	int	received	level

decode - decodes message from an input stream, writing results to an output stream. Note: this function is not recursive. Assumes the streams are valid.

- Analysis

Objects	Type	Movement	Name
input stream connected to encoded message file	istream	received & passed back	message_file
output stream	ostream	received & passed back	out

The basic algorithm for this function is to read each character from the message file (all characters will be `0` or `1`, and there will be no spaces, so use **`>>`**) and traverse the codetree starting at the root. When a leaf is reached, write out the letter in the leaf node, then start over again at the root for the next encoding. You may assume that the encoded message file contains only 0's and 1's, and ignore any other inputs. (However, you may want to display a debugging error message if other characters are found in the input stream stating they are being ignored. Be sure to comment out any such debugging code before submitting.) **Note the output must end with a newline, so if the last decoded character is not a newline, one will needed to be output explicitly.**

encode - encodes message from an input stream, writing results to an output stream, using the recursive char_to_code helper function. Assumes the streams are valid.

- Analysis

Objects	Type	Movement	Name
input stream connected to message file	istream	received & passed back	message_file
output stream	ostream	received & passed back	out

The basic algorithm for this function is to repeatedly read a character (including spaces and newlines, so use the `message_file.get()` member function) from the input stream, encode it using helper function char_to_code, then write the encoding to the output stream. In order for the resulting file to be a proper text file, **the output must end with a an actual newline after all the input characters have been encoded.** You may assume that the message file contains only the characters in the codetree and ignore any that do not encode properly. However, you may want to display a debugging error message if no encoding is found for a read character stating that the character is being ignored. Be sure to comment out any such debugging code.

char_to_code – **a private** recursive helper function to find an encoding from the codetree.

- Analysis

Objects	Type	Movement	Name
character to encode	char	received	ch
node of tree being traversed	Node*	received	local_root
path constructed so far	string	received	path_so_far
encoding for character	string	returned	code

The basic algorithm for this function is to build a path (in path_so_far, by concatenating a `0` or `1` as appropriate) as the function traverses the codetree. When a leaf node is reached, if the leaf data value is the character being encoded, return the path; otherwise, this was not the correct path and return the empty string. Note: concatenating multiple empty strings results in an empty string.

Copy constructor - creates a copy of the source HuffmanTree using the recursive copy_tree helper function.

- Analysis

Objects	Type	Movement	Name
original tree	HuffmanTree	received	source

copy_tree – **a private** recursive helper function that copies a binary tree and returns a pointer to the root of the copy

- Analysis

Objects	**Type**	**Movement**	**Name**
root of source tree	Node*	received	local_root
root of copy	Node*	returned	copy_root

Copying a (sub)tree given a pointer to its root is a pre-order traversal: make a new root node for the copy, then make copies of the left and right subtrees and hook them up to the new root node. Coding note: since the return type comes before the HuffmanTree:: scope operator before the name of the function, the return type for this function needs to have the HuffmanTree:: scope operator before it, too, in the .cpp file. I.e.,
```
HuffmanTree::Node * HuffmanTree::copy_tree (Node *local_root)
```

Destructor – deletes all HuffmanTree nodes using the delete_tree private helper function.

- Analysis - no objects

delete_tree – **a private** recursive helper function that deletes binary tree nodes.

- Analysis

Objects	**Type**	**Movement**	**Name**
root of source tree	Node*	received	local_root

Deleting a tree given a pointer to its root is a post-order traversal: delete the left and right subtrees, then delete the root node.

operator= - assignment operator, make this HuffmanTree a copy of the source HuffmanTree, returns a reference to this tree. **This function must be implemented using the copy and swap technique.**

- Analysis

Objects	**Type**	**Movement**	**Name**
original tree	HuffmanTree	received	source
this tree	HuffmanTree &	returned	*this

Assignment

This assignment will be distributed by the GitKeeper submission system with two codefiles `samplecodefile.dat` and `codefile.dat`. Follow the directions in the assignment email. Be sure to commit your changes early and often. **The first commit for this project should be no later than Friday, April 17.** It is recommended that commits be done after each class operation is implemented and tested. Commit messages should be descriptive of what has been completed.

The file `samplecodefile.dat` will be the partial Huffman code used in the examples above, thus suitable for preliminary testing of your HuffmanTree class. The file `codefile.dat` will contain a full Huffman code for the lowercase Latin alphabet (plus space and newline) based on the distribution of letters in English. You will have to write your own sample input files. Note that when the encode and decode functions are working, you should be able to send the output of the encode function into the decode function and get back your original plaintext message (and vice versa). You can use the `diff` command to see if two files are the same.

(20 points) Write the implementation of the HuffmanTree class as specified above. The HuffmanTree class definition must be put in header file `huffman.h` with suitable compilation guards. The implementations of the HuffmanTree class functions must be put in source file `huffman.cpp`. Note that the recursive helper functions are free (i.e., non-private, non_member) functions. As such their prototypes belong at the beginning of the source file, not the header file.

(10 points) Write a (main) program in file `coder.cpp` that has one command-line argument, the name of a codefile. All appropriate error checking must be completed. The program should construct a HuffmanTree object using the given codefile, then display the constructed codetree to the screen. Then the program should enter a menu driven loop that asks the user if they want to encode, decode or quit. For encode and decode, the program should ask for an input file name and an output filename, then do the appropriate action reading in from the input file and sending the output to the output file.

You must submit a makefile named `Makefile` that creates an executable file named `coder` for your project. It should conform to the examples demonstrated in class.

REMINDER: Your project must compile for it to be graded. Submissions that do not compile will be returned for resubmission and assessed a late penalty. Submissions that do not substantially work also will be returned for resubmission and assessed a late penalty.

Follow the guidelines in the *C++ Programming Style Guideline* handout. As stated in the syllabus, part of the grade on a programming project depends on how well you adhere to the guidelines. The grader will look at your code listing and grade it according to the guidelines.

Side note

The encoded message file generated by this program will be larger than the original unencoded message, since our representation of a bit is a text character (8 bits). Consider that each character in the unencoded message will take up 8 bits, while each character in the encoded message will take up only 1 bit. So to compute the amount compression that would take place if each bit of the encoded message actually took up 1 bit, compare the number of characters in the unencoded message with the number of characters in the encoded message divided by 8. (The `wc` command in Unix will tell you how many lines, words, and characters are in a textfile.)

How to submit

Reminder: all public names (the class name, the operation names, and file names) must be as specified including capitalization, and all parameters must be implemented in the order listed. Internal names (private data member, private member functions, function parameters, local variables) can be whatever you wish. The autotest program also requires that the format of any created strings or output be exactly as expected including whitespace, capitalization, and spelling. Submit projects by first adding and committing changes, then pushing the project. See handout *Using GitKeeper for Submitting Assignments* for additional information.

Note: in addition to compiling and running the autotest programs, when all the autotest program pass, the last one will be run again under the valgrind memory leak detector program. This will cause the output of the autotest program to be repeated. If you wish to run this test before pushing your project, first make sure that valgrind is installed on your machine (login into an admin account and type: `sudo apt install valgrind`). Then to run a program under valgrind use:

```
$ valgrind --leak-check=full --error-exitcode=1 ./<program name>
```

Example Run

```
user@csserver:~/cs215/project6$ more samplemessage.plaintext
sunnyinus
user@csserver:~/cs215/project6$ ./coder samplecodefile.dat
                i
        *
                n
*
                        y
                *
                        u
        *
                s


Please choose one of:

   E - Encode a message
   D - Decode a message
   Q - Quit the program

Enter your choice: e
Enter the name of a plaintext message file: samplemessage.plaintext
Enter name of output file: samplemessage.encoded

Please choose one of:

   E - Encode a message
   D - Decode a message
   Q - Quit the program

Enter your choice: d
Enter the name of a compressed message file: samplemessage.encoded
Enter name of output file: samplemessage.recovered

Please choose one of:

   E - Encode a message
   D - Decode a message
   Q - Quit the program
```

```
Enter your choice: q
user@csserver:~/cs215/project6$ more samplemessage.encoded
0001010100111111001000
user@csserver:~/cs215/project6$ more samplemessage.recovered
sunnyinus
user@csserver:~/cs215/project6$
```