Huffman Code Generator

COP 4530 Programming Project 4

Instructions

For Programming Project 4, you will be generating Huffman codes to compress a given string. A Huffman code uses a set of prefix code to compress the string with no loss of data (lossless). David Huffman developed this algorithm in the paper “A Method for the Construction of Minimum-Redundancy Codes” (http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf)

A program can generate Huffman codes from a string using the following steps:

- Generate a list of the frequency in which characters appear in the string using a map
- Inserting the characters and their frequencies into a priority queue (sorted first by the lowest frequency and then lexicographically)
- Until there is one element left in the priority queue
  - Remove two characters/frequencies pairs from the priority queue
  - Turn them into leaf nodes on a binary tree
  - Create an intermediate node to be their parent using the sum of the frequencies for those children
  - Put that intermediate node back in the priority queue
- The last pair in the priority queue is the root node of the tree
- Using this new tree, encode the characters in the string using a map with their prefix code by traversing the tree to find where the character’s leaf is. When traversal goes left, add a 0 to the code, when it goes right, add a 1 to the code
- With this encoding, replace the characters in the string with their new variable-length prefix codes

In addition to the compress string, you will need to be able to serialize the tree. Without the serialized version of the Huffman tree, you will not be able to decompress the Huffman codes. Tree serialization will organize the characters associated with the nodes using post order. During the post order when you visit a node,

- if it the node is a leaf (external node) then you add a L plus the character to the serialize tree string
- if it is a branch (internal node) then you add a B to the serialize tree string

For decompression, two input arguments will be needed. The Huffman Code that was generated by your compress method and the serialized tree string from your serializeTree method. Your Huffman tree will have to be built by deserializing the tree string by using the leaves and branches indicators. After you have your tree back, you can decompress the Huffman Code by tracing the tree to figure out what variable length codes represent actual characters from the original string.

So, for example, if we are compressing the string “if a machine is expected to be infallible it cannot also be intelligent”:

Our compress algorithm would generate the following codes for the characters:

Character	Frequency	Prefix Code
(space)	12	00
a	5	1011
b	3	10101
c	3	11000
d	1	010000
e	9	100
f	2	01011
g	1	010001
h	1	010010
i	8	011
l	6	1101
m	1	010011
n	6	1110
o	3	11001
p	1	010100
s	2	10100
t	6	1111
x	1	010101

Our code would be:
01101011001011000100111011110000100100111110100000111010000100010101010101001001
100011111000100000001111110010010101100000011111001011101111011101011101011101100
000111111001100010111111011101100111110010111101101001100100101011000001111101111
1001101110101101000110011101111

And our serialize tree would look like:
L LdLgBLhLmBBLpLxBLfBBLiBBLeLsLbBLaBBLcLoBLlBLnLtBBBB

You will need to create one class for this project: HuffmanTree for the compression, decompression, and serialization that uses a linked binary tree. You are given a Heap-based

Priority Queue for the sorting. You are allowed to use the STL map, vector, and stack, but not the STL priority queue.

Abstract Class Methods

std::string compress(const std::string inputStr)
Compress the input string using the method explained above. Note: Typically we would be returning a number of bits to represent the code, but for this project we are returning a string

std::string serializeTree() const
Serialize the tree using the above method. We do not need the frequency values to rebuild the tree, just the characters on the leaves and where the branches are in the post order.

std::string decompress(const std::string inputCode, const std::string serializedTree)
Given a string created with the compress method and a serialized version of the tree, return the decompressed original string

Other things in Huffman hpp/cpp

To simplify the process, I have given the full interface and implementation for a class called HuffmanNode. This class has all the basics for a tree node (leaf, branch, root, data members for linking, accessor) and also includes a comparator class for use with the heap. You should not need to alter any of the code for this node.

Examples

Below are some examples of how your code will run

```
HuffmanTree t;

string test = "It is time to unmask the computing community as a
Secret Society for the Creation and Preservation of Artificial
Complexity”;

/*
1000101011110000101001100110000010111011001110111101111100011001001011
0100100001111001110010011101101111010110010101010111110011000001110000
1011011110101100100010111110001100001111111111001011010011001011101001
1111101111001001110011110100111101111110000111001111111111010101110110
1001100111001001110110100110010011100101011000101100111100101001110000
0111010000000100111010100111001001000110010101100010110011110101110101
1110100010001000110001010110001111000001011001011101001101011001010101
010010111101000111000011111111 */
string code = t.compress(test);
```

```
/* LiLmLnBBLrLaBLtBBLPLdBLgLkBBLALIBLvLxBBBLhLlBLCLSBBBLsLpLfBBLoBBL
LeLcLuLyBBBBBB */
string tree = t.serializeTree();

/* It is time to unmask the computing community as a Secret Society
for the Creation and Preservation of Artificial Complexity */
string orig = t.decompress(code, tree);
```

Deliverables

Please submit complete projects as zipped folders. The zipped folder should contain:

- Huffman.cpp (Your written HuffmanTree class)
- Huffman.hpp (Your written HuffmanTree class)
- HeapQueue.cpp (The given Heap Priority Queue using an array/vector class)
- HeapQueue.hpp (The given Heap Priority Queue using an array/vector class)
- HuffmanBase.cpp (The provided base class and helper class)
- HuffmanBase.hpp (The provided base class and helper class)
- PP4Test.cpp (Test file)
- TestStrings.hpp (Test file)
- catch.hpp (Catch2 Header)

And any additional source and header files needed for your project.

Hints

For the encoding step where you translate characters using your Huffman Tree, this is essentially a preordering of the tree and can be done recursively.

Remember, when you are deserializing, you are going from post ordering back to the full tree. This is very similar to the postfix to infix conversion you did in PP2, but now building a tree instead of an expression.

For decoding the characters, you just follow the tree down the branches until you hit the leaf with the character, adding a zero for a left move and adding a 1 for a right move.

I suggest implementing a recursive method to destroy nodes for your destructor.

For the branching nodes, I suggest using the null character just to hold a spot since this should not be popping up in text you are compressing.

Additional resources:
https://en.wikipedia.org/wiki/Huffman_coding
https://www.youtube.com/watch?v=0kNXhFIEd_w

Extra Credit

There is one opportunity for extra credit on this project.

You will need to use a map in various parts of this project. You are welcome to use the STL map provided by C++, but for 10 points of extra credit on this assignment you can implement your own map class in separate files from your code. You must create a Hash-based Map that can be given a size N for the size of the table and uses a division compression method (y mod N) and uses open addressing for collisions.

Rubric

Any code that does not compile will receive a zero for this project.

Criteria	Points
Should compress the turing string	3
Should serialize the tree for turing	3
Should decompress the turing string	3
Should compress the dijkstra string	3
Should serialize the tree for dijkstra	3
Should decompress the dijkstra string	3
Should compress the wikipedia string	3
Should serialize the tree for wikipedia	3
Should decompress the wikipedia string	3
Should compress the constitution string	3
Should serialize the tree for constitution	3
Should decompress the constitution string	3
Code uses object oriented design principles (Separate headers and sources, where applicable)	2
Code is well documented	2
Total Points	**40**

Extra Credit

Criteria	Points
Successfully implemented a Hash Map	10
Total Points	**10**