COP 4530
Project 3
Spring 2024

Instructions-

For Programming Project 3, you will be generating Huffman codes to compress a given string. A Huffman code uses a set of prefix codes to compress the string without losing 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:

1. Generate a list of the frequency in which characters appear in the string using a map
2. Inserting the characters and their frequencies into a priority queue (sorted first by the lowest frequency and then lexicographically). See **Priority Queue formation** section
3. Until there is one element left in the priority queue
4. Remove two characters/frequency pairs from the priority queue
5. Turn them into leaf nodes on a binary tree
6. Create an intermediate node to be their parent using the sum of the frequencies for those children
7. Put that intermediate node back in the priority queue
8. The last pair in the priority queue is the root node of the tree
9. 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
10. With this encoding, replace the characters in the string with their new variable-length prefix codes

In addition to the compressed 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,

1. If the node is a leaf (external node) then you add an `L` plus the `<character>` to the serialized tree string

2. If it is a branch (internal node) then you add a `B` plus `$` to the serialized tree string plus the `<character>` to the serialized tree string

3. For decompression, two input arguments will be needed- 1)The Huffman Code generated by your compress method and 2)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 compression algorithm would generate the following Huffman codes for the characters:

```
n: 1111
t: 1110
b: 11011
c: 11010
l: 1100
o: 10111
d: 101101
g: 101100
x : 011100
a: 1010
<space> : 00
e: 100
p: 011101
m: 011110
h: 011111
s: 01100
i: 010
f: 01101
```

Our code would be:

```
0100110100101000011110101011010011111010111110000010011000010001110001110110011010111010010110100111010111001101110000010111101101101011001100010110111100100000101110001101010101111111110111111000101011000110010111001101110000010111111101001100100010101100100111111110
```

Our serialized tree would look like this:
`L<space>LiLsLfB$LxLpB$LmLhB$B$B$B$B$LeLaLgLdB$LoB$B$B$LlLcLbB$B$LtLnB$B$B$B$`

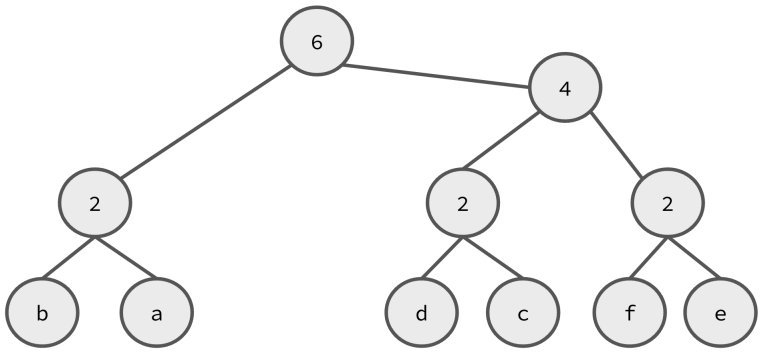
You are given a Heap-based Priority Queue for the sorting. **You are allowed to use the STL map, vector, and stack, but not the STL priority queue**

Priority queue formation

Priority queue should be formed with the Huffman nodes so that the comparison between nodes follows the logic-

```
function compare(l as HuffmanNode, r as HuffmanNode):
    if freq(l) == freq(r):
          // If frequencies are equal, compare
     characters lexicographically
        return l.data < r.data
    else:
        // Otherwise, compare frequencies
        return freq(l) > freq(r)
```

As a simple example, if we compress the string “`abcdef`”, it will result in the following Huffman tree-



The generated codes will be as follows-

```
e : 111
c : 101
d : 100
f : 110
a : 01
b : 00
```

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 several 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

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

/*
10001000101101111100001100101111001111101100101010110011000
10001101111000010001101100101001001110110101111010001110011
01100010111100101000111110101111010001100110110000101111010
00001110011110000110011111010010111101011100011110010110100
11010101111111111001000001110011011010000111001010010011101
10101100000111100111010111110100010110011100100000001100000
00011110100001110000110110110111010111110100010110101001101
10100010100010101111011011111101111111011110011111010110010
00011100110100111111010110101111010000 */
string code = t.compress(test);

/*
LdLPB$LyB$LrB$LnLmB$B$LtLuLfB$LaB$B$B$LsLILAB$LkLgB$B$B$LhLS
B$LpLlB$B$B$LoLCLxLvB$B$LcB$B$B$L LeLiB$B$B$B$ */
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:

`HeapQueue.hpp`
`HuffmanBase.cpp`
`HuffmanBase.hpp`
`PP3Test.cpp`
`TestStrings.hpp`
`catch.hpp`

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

Testing-

See `PP3Test.cpp` for details. Remember you are free to use any development environment but your code must runnable on the student cluster with the commands specified in `PP3Test.cpp`

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 the text you are compressing.

- Additional resources:

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

Rubric-

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

Criteria	Points
Should compress the turing string	7.5
Should serialize the tree for turing string	7.5
Should decompress the turing string	7.5
Should compress the dijkstra string	7.5
Should serialize the tree for dijkstra string	7.5
Should decompress the dijkstra string	7.5
Should compress the wikipedia string	7.5
Should serialize the tree for wikipedia string	7.5
Should decompress the wikipedia string	7.5
Should compress the constitution string	7.5
Should serialize the tree for constitution string	7.5
Should decompress the constitution string	7.5
The code is well documented	10
Total Points	**100**