**CS271: Data Structures**

**Spring 2024**

**Project 5**

**Binary Trees**

**for**

**Huffman Coding**

## Overview

The main objective of this project is to implement a Binary Tree data structure and then use it to implement a Huffman Coding scheme for letters a .. z.

## Parsing Text for Letter Frequency

Download the file WarAndPeace.txt. Write a small program that will count the number of instances of letters. First convert all characters to lower case. Then count the number of occurrences of 'a', of 'b', and so on up to 'z'. Print a table of the letter frequencies found in this novel. You can use I/O redirection to save this data into a text file.

You do not need to submit the source code for this small program.

Please do submit your datafile of frequency counts. Name this file frequency.txt.

## BT (Binary Tree) Class

You are to create a BT class in c++ which implements a Binary Tree. This class should hold templated data values at each node (see below). The Binary Tree has the following property:

* The value at the root must be larger than the value of either child.
* Each node can have 0, 1 or 2 children.

Note, this is similar to the Heap property except that we relax the fullness requirement; our BT data structure can have sparse branches. Because of this, you cannot implement it using an array. You must use a pointer type structure with pointers to left, right children and the parent.

I will leave it up to you to decide which methods need to be part of this tree structure.

## Template Type

I suggest creating a small class for the template type that will be stored at each node in the tree. Each node will store two pieces of information:

- a count (integer)

- a character (char) which will be one of a..z or 0 (to indicate no character)

We will need the ability to compare two binary trees and decide which is larger. A larger BT has a larger count field at the root.

You will need to implement the six comparison operators in your template type ( >, <, >=, <=, ==, != ) which should compare the count fields at the root and return the appropriate boolean result.

## Huffman Coding

Huffman coding is a way to encode data using an efficient format. The Huffman code, when built, can be shown to be the minimal length representation of the data. We use the data frequency to build the code. The more frequent data are coded with shorter vectors while less frequent data is coded with longer vectors.

Read Section 16.3 in your textbook to understand how to use a BT to build and store a Huffman coding scheme.

Let's illustrate the construction of a Huffman code on a small example. Suppose we have only 6 letters (instead of 26) and the letters have the following frequencies:

char: a e c d b j

frequency: 25 21 15 13 7 2



Notice we use the convention of putting the larger node to the left when we combine two BTs into a single BT. Internal nodes have no character (or character 0) while leaf nodes have characters. The count at a leaf node is the frequency of that character. The count of an internal node is the sum of counts for its two children.

When we "read" the tree to form a code, we assign 0 to a left turn and 1 to a right turn. Thus the 'e' is encoded as a 10 (first a right turn, then a left turn). This results in the following encodings:

char: a e c d b j

frequency: 25 21 15 13 7 2

code: 00 10 11 010 0110 0111

This results in an average of 2.375 bits per character. This is smaller than the 3 bits per character of before.

When we receive a decoding, we use the tree to read bits until we get to a leaf node. For example, 01110001010 is parsed as:

left-right-right-right or a 'j', 0111

then left-left for an 'a, 00

then left-right-left for a 'd', 010

then right-left for an 'e'. 10

This spells jade.

## Priority Queue

You will need to use a priority queue to implement the Huffman Coding scheme. The items in the queue will all be BT objects. Thus we need a comparison operator (> and < minimally) so we can decide priority in the queue. The normal operation is to remove the two smallest BTs in the queue, combine them into a new single BT, and then re-insert it into the queue.

Notice you will need to be able to build a new BT out of two existing BTs. Be sure you include this kind of functionality in your BT class.

If you want, you can go back and grab your PQueue class from cs173. I suggest though that it may be easier to use the STL library priority queue class that is built in. You will have to do a little research to understand how to use it.

## Application

Your overall application is to build a Huffman Coding scheme based on the letter frequencies from the War and Peace novel. This involves several steps:

* Building a small object for a node in a BT
* Building the BT class (templated for holding the above object)
* Use a priority queue (templated for holding BTs)
* Write an application to construct a Huffman coding scheme. This should output a file called code.txt which has the following format:
* Find the song lyrics to your group's favorite song. Store them in a plain text file after you strip out all the non-letters, and convert all the remaining letters to lowercase. Write a small application called encode.cpp which reads the code.txt file and the song lyrics (from stdin), encodes the song lyrics and prints the binary string to stdout.
* Write a small application decode.cpp which reads the encoded song lyrics and decodes them back to plaintext (obviously without punctuation). This application should read the code.txt file and create (re-create) a binary tree structure from this code. Use the binary tree to parse code strings back into lower case letters.

Submit all files in a tar/zip file. Include a makefile that compiles multiple executable targets. Document very very very well. Maybe include a readme.txt file which helps decipher all the parts.