Linked Lists/Priority Queues/Binary Trees

75 pts Extra Credit

You may work with a partner, or you may work alone.

This lab involves using Linked Lists, Priority Queues, and Binary Trees in order to create a Huffman code for a text file. It compresses a text file using the Huffman code you created, and then compares that to the ascii version of the file (loosely compared).

Some definitions:

Linked List *– you should know what that is at this point. For this lab you'll be modifying the linked list you created in class. I modified the doubly linked list, but I can see no reason not to use the singly linked list.*

Queue*. A queue is a list in which nodes can only be added to the end, and removed from the beginning. The best analogy to a queue is a line at a bank/grocery store/etc. People get in line at the end, and get served and leave the line from the beginning. If you've got a queue as a linked list, you'd only need the remFirst() and push() methods.*

Priority Queue: *A variant on the queue is the priority queue. In a priority queue, each node has a priority, so those nodes with the highest priority are always at the front of the list. This means that, for our purposes, when a new node is inserted into the list, it is inserted in order based on its priority.*

Binary Tree: *a data structure with each node having at least 2 links: one to the left "child" and one to the right "child". Usually each node has a link to its parent as well. You may already be familiar with a binary search tree, in which, for each node, the data in the left child is less than the data in the node, and the data in the right child is greater than the data in the node. Key factors to know about a binary tree are that each node is, on its own, a binary tree, that each node has at most 2 children (and if it has no children, then it is known as a leaf). Each node has at most 1 parent. And the tree has only one root node, from which everything else branches.*

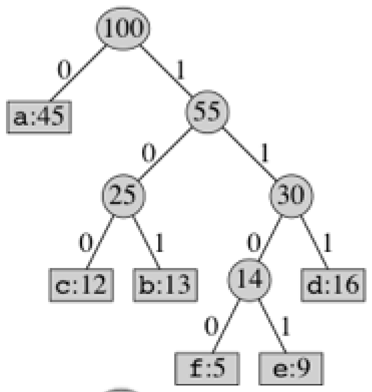
Huffman Code: *a method used for compressing files, often image files (e.g., jpg files) in which data that occurs most frequently within a file has the shortest representational code, and data that occurs less frequently has a longer representational code. The Huffman code is lossless, meaning that when we compress a file using the Huffman code, no data is lost or compromised.*

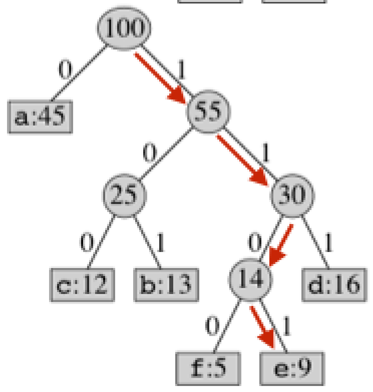
Quoting http://www.cs.dartmouth.edu/~traviswp/cs10/lab/lab4/lab4.html*:*

A problem with variable-length encodings is figuring out where a code word ends. For instance, if we used '1' for 'E', '10' for 'F' and '00' for 'X' we would be in trouble. When decoding the file we would not know whether to interpret '1' as 'E', or as the start of the code for 'F'. Huffman Encoding removes this ambiguity by producing *prefix-free* codes. This means that for any given code word, adding bits to the end cannot produce another code word. Hence no code word is a prefix of another. When the computer observes a series of bits that corresponds to a code word, it knows that it cannot be part of a larger code word, and can safely interpret the series of bits as its corresponding character.

At this point the purpose and value or Huffman Encoding should be fairly clear. So how do we do it? The task is to generate a set of prefix-free codes whose lengths are inversely correlated with the frequency of the encoded character. There are two clever parts of the algorithm, the use of a binary tree to generate the codes, and the construction of the binary tree using a priority queue. Specifically we will construct a tree such that each character is a leaf and the path from the root to that character gives that character's code word, where a left child is interpreted as a 0 and a right child as a 1.

For example, in this tree, the codeword for e is 1101:





Assignment:

You are going to create a Huffman code for the characters in a text file, print out the file using the Huffman code, and then compare its size to a file created using the binary representation of each character in the file.

In order to create a Huffman code for a particular document or file, we first need to figure out the frequency counts of characters in a file. For this you'll need your linked list, with some modifications. Your linked list nodes will hold a character instead of an int, and it will also hold a count of the number of occurrences of that character. (Your node will also have a few other fields: an **STRING** field, which will eventually hold the code generated for that character, and since you'll be changing this linked list into a binary tree, the nodes will also need a left and a right pointer (as well as a next and a prev pointer, for the doubly linked list).

Part 1:

Your linked list (priority queue) LLPQ class will need (along with a first field and a size field), the following methods:

```
	LLPQ();
	~LLPQ();
	void printLLPQ();
	// prints out the linked list/ priority queue
	void addFirst(char x, string co="");
	//adds the very first character node to the linked list, along with an
	//original default code value set to -1.

	void addAtFirst(char x, string co="");
	// add a new node to the beginning of the linked list (modifying the first
	//pointer and the size, and setting the code field to co, (default = -1).
	LLNode *remFirst();
	// removes the first node from the list (to be used in creating the
	//Huffman code
	string findCode(char k);
	// goes through the linked list, finds the character k, and returns the code
associated with that node – used to translate a file once you have the code (Note
that if we had studied hash tables/maps, this would be so much easier using them

	void sortLL();
	// sorts your linked list, based on the counts in the nodes (so the character
that occurred the least frequently will be at the beginning of the list, and the
character that occurred most frequently will be at the end of the list.
	void insertUnique(char c);
	// inserts only unique characters into the linked list.  If the character is
already in the linked list, it increases the count of that character
	void insertInOrder(LLNode *n);

	//inserts the node n into the linked list in order of its count value – this
will be used in the creation of the Huffman code.
```

So to start creating your priority queue, you will have 4 files: LLNode.hpp, LLNode.cpp, LLPQ.hpp, and LLPQ.cpp.

You will read in a file, and create a linked list, with only unique characters in the list, keeping track of the number of times each character occurs.

You will then sort the list. Now you have a priority queue, with the character nodes that occurred the least at the front of the list, and the ones occurring most at the end of the list.

Part 2: Making the Huffman Code:

In order to make the Huffman code, you will be creating a binary tree out of your priority queue linked list. Literally using the exact same nodes, because the nodes have been designed to work as both linked list nodes and as tree nodes. This makes creating a Huffman code easily. A Huffman code is created as follows:

Take the first two nodes from the priority queue (those with the lowest counts) and remove them from the queue (but don't delete them). Create a new node. The new node should have a bogus character (I used ‘*’ for my inner nodes) and a count that is the count of the first node + the count of the second node. This new node’s left child should be the first node you removed from the queue, and the new node’s right child should be the second node you removed from the queue.

Now insert into the priority queue in order this new node you just created. Yep, you’ve got one node you are inserting, with the count being the count of old node 1 + count of old node 2, and this node just happens to have two children, so it is also a tree.

Once you have inserted the new node into the priority queue, you will remove from the beginning of the priority queue, make a new node, count being the sum of the 2 nodes’ you’ve just removed counts, the new node’s left being the first node you removed, and the new node’s right being the second node you just removed, and now insert this new node (in order) into your priority queue.

You will continue doing this until there is only one node left in the priority queue. This will be your root.

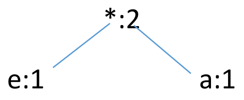
So, for instance, if your linked list starts as:

```
e:1->t:2->a:1-> :3->s:3->i:2->h:1->T:1->
```

```
sorted, you will get the following priority queue:
```

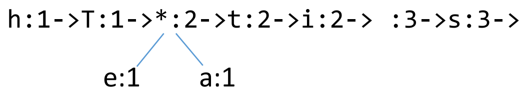
```
e:1->a:1->h:1->T:1->t:2->i:2-> :3->s:3->
```

```
Now you will combine the first two nodes into a new tree:
```



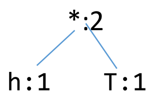
```
h:1->T:1->t:2->i:2-> :3->s:3->
```

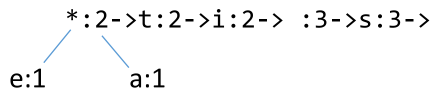
You’ll insert in order this new node into the linked list:



```
h:1->T:1->*:2->t:2->i:2-> :3->s:3->
```

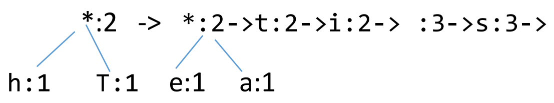
Repeat:





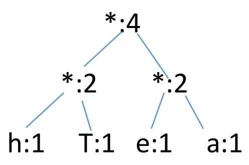
```
*:2->t:2->i:2-> :3->s:3->
```

Insert (in order) the new node into the priority queue:



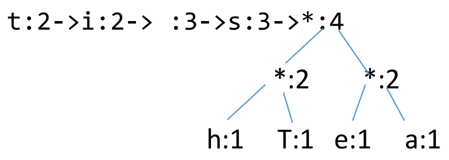
```
*:2 -> *:2->t:2->i:2-> :3->s:3->
```

Remove and combine first two nodes again:

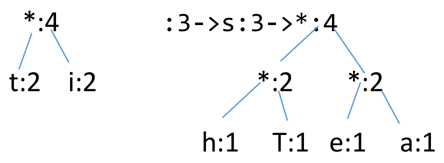


```
t:2->i:2-> :3->s:3->
```

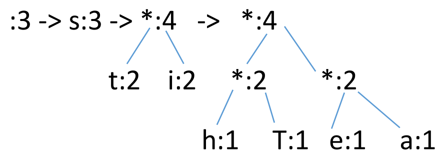
Insert *:4



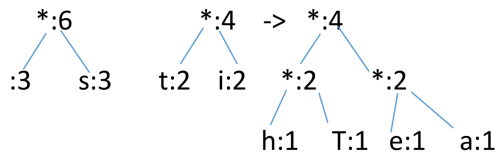
Repeat:



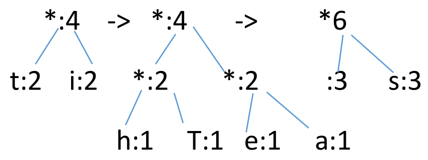
Insert:



Repeat:

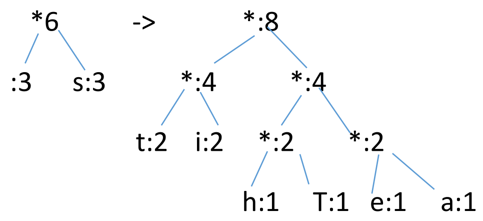


Insert:

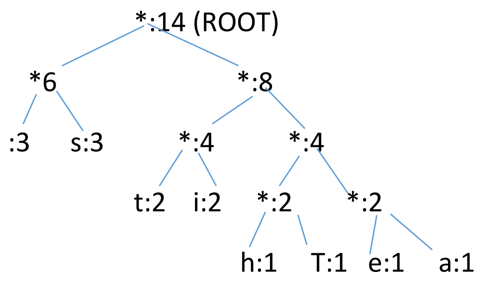


(Note that this row is still a linked list priority queue)

Repeat (with *4 and *4) and insert:



And finally, combine the final two nodes in the list and make the new node the root of the Huffman tree:



Now the root is 1, and every left branch adds a 0, every right branch adds a 1. So the code generated by this tree would be:

:00 //this is the space key
s:01
t:100
i:101
h:1100
T:1101
e:1110
a:1111

So for the Huffman code, you will need to create the following: LLHuff.hpp and LLHuff.cpp, with LLHuff.hpp looking like:

```
class LLHuff {
	string file;
	LLNode *root;
	int size;
public:
	LLPQ *pq;  //priority queue list of character nodes, made in first part
	LLPQ *ascii;  //linked list of characters and their corresponding asci codes, 
read in from the asci.txt file
	LLHuff(string f);  //f is the name of the file you'll be creating a Huffman 
code for
	~LLHuff();
	void MakeHuff();//takes the priority queue and makes the Huffman tree out of 
it as shown above, setting the root to the last remaining node created.
	void FindCode(LLNode *root, string path);  //finds the code for each leaf node 
in the tree.  A few notes about this one:  first, the path starts at 1, and is 
multiplied by 10 each iteration through to get an accurate integer representation of 
the path.  I wrote this function as a recursive one, and I STRONGLY recommend you do
```

as well. All codes for every leaf can be found in one pass of the tree with recursion.
One final note: I made a new linked list (assigned to pq, since I was done with the old pq, with each leaf node and its code being added to the linked list. This linked list of characters and codes will be used to create the compressed version of the file, in which each character is replaced with its corresponding code. This would be a lot easier with a hash table/map, but we haven't worked with them yet. If you want to make a new field and create a hash table for this part, I am fine with that.

```
	void ReadFile();
	//Opens, reads in a file character by character, and creates the priority
queue
	void compressFile();
	//Opens the original file for reading and a new, compressed file for writing.
Looks up each character read in from the original file and writes out its
corresponding code in the compressed file.
};
	void ReadAscii() ;
//reads in the asci code from the asci.txt file, and then takes the regular file and
converts it to the ascii code and saves that file.
```

I'm including the ReadFile method (which should be in your LLHuff.cpp file) below:

```
void LLHuff::ReadFile() {
	ifstream infile(file.c_str(),ios::in);     // open file
	char k;
	while (infile.get(k)) {          // loop getting single characters
		pq->insertUnique(k);// inserting the character k into the priority queue
	}
	infile.close();
}
```

You can use this file as your guideline for reading in the file in order to compress it.

Part 3: creating a compressed file and a corresponding

In the above LLHuff files, you've got functions to create a linked list of characters and code nodes (stored in pq, unless you used a hash table/map). You will need to use the compressFile function to re-read in the text file, open a file for writing (using

```
	ofstream outfile("compressed.txt",ios::out);
```

Once you've opened the file for writing, you can write to the file using:

```
	outfile << comp <<" ";  //retype the quotes – ms word changed them
```

// In the compressFile function, I looked up the character I read in from the file in the priority queue, found its code, and put the code into the int comp. I then printed out the comp into the outfile, with a space between each character to check it.

Each character also has an ascii/binary representation that is 8 bits long. In readAscii, I read in the ascii table, and created a linked list of ascii values for each character. In the compressFile function, I also opened a second file for writing. In that file, I looked up each character I read in in the ascii linked list, and printed that code out to the second file, and saved that file as well.

Once complete, look at the size difference between the two files. It should be fairly small, because the files were small to start with, but it should be slightly shorter.

I've included the readAscii method so you can compare the size of your compressed file with the size of the regular ascii file. Note that I'm putting spaces between both the Huffman code and the ascii code for easier reading. The nice thing about both the Huffman code (with variable sized codes) and the asci code (with fixed size codes) is that you don't need to include that extra space. But the lack of space does make it hard to read.

```
void LLHuff::ReadAscii() {
	cout << file << endl;
	ifstream infile("asciitable.txt",ios::in);     // open file
	char ch;
	string asciicode;
	if (!infile.is_open()) {
		return;
	}
	infile >> asciicode;
	pq->addFirst(' ',asciicode);
	infile >> asciicode;
	while (infile>>ch) {          // loop getting single characters
		cout << ch;
		infile>>asciicode; // gets entire word (code)
		cout << asciicode << endl;
		pq->addAtFirst(ch,asciicode);
	}
	cout << endl;
	infile.close();
	/////////////////////////////////////
	ifstream infile2(file.c_str(),ios::in); // open file for reading
	ofstream outfile("asciivsn.txt",ios::out);
	char k;
	string comp;
	while (infile2.get(k)) {          // loop getting single characters
		cout << k;
		comp = pq->findCode(k);
		if (comp == "") {
			cout << "ERROR WITH " << k << endl;
		}
		else {
			cout << k << ":" << comp << endl;
			outfile << comp << " ";
		}
	}
	cout << endl;
	infile2.close();
	outfile.close();
}
```

NOTE: for testing, you can use test.txt. Once you have everything working, you should test it on a larger file, like Monet.txt. I've also included the Huffman main function as well below:

```
#include <stdio.h>
#include <iostream>
#include "LLNode.hpp"
#include "LLHuff.hpp"
using namespace std;

int main() {
	cout << "reading file" << endl;
	LLHuff code("tests.txt");
	code.ReadFile();
	code.pq->printLLPQ();
	code.pq->sortLL();
	code.pq->printLLPQ();
	code.MakeHuff();
	code.compressFile();
	code.ReadAscii();
	return 0;
}
```

Below I've also included my final output files for the test.txt) (again, note that I put space between each character code. This wastes space but makes it easier to read):

Compressed.txt (final size:474 bytes):

```
00010 0000 1000 1100 01 1000 1100 01 1010 01 1001 001 1100 1001 10110 01 01 00010 0000 1000
1100 01 1000 1100 01 11011 11010 101111 00011 01 1010 01 1001 001 1100 1001 10110 01 01
1110001 1110000 01 1001 0000 1000 1100 01 0000 1010 11101 01 101110 001 001 11010 01 1010
11010 01 1010 111101 1001 111100 1010 101111 01 001 111111 001 1110011 1110010 001 11010
111101 00011 1111101 01 00011 11011 111100 1111100 11101 01 101110 001 01 11101 11011 11011
111111 001 11101 10110
```

And Asciivsn.txt (final size: 837 bytes):

```
01010100 01101000 01101001 01110011 00100000 01101001 01110011 00100000 01100001
00100000 01110100 01100101 01110011 01110100 00101110 00100000 00100000 01010100
01101000 01101001 01110011 00100000 01101001 01110011 00100000 01101111 01101110
01101100 01111001 00100000 01100001 00100000 01110100 01100101 01110011 01110100
00101110 00100000 00100000 01001001 01100110 00100000 01110100 01101000 01101001
01110011 00100000 01101000 01100001 01100100 00100000 01100010 01100101 01100101
01101110 00100000 01100001 01101110 00100000 01100001 01100011 01110100 01110101
01100001 01101100 00100000 01100101 01101101 01100101 01110010 01100111 01100101
01101110 01100011 01111001 00101100 00100000 01111001 01101111 01110101 00100111
01100100 00100000 01100010 01100101 00100000 01100100 01101111 01101111 01101101
01100101 01100100 00101110
```