Lab 8

Due No Due Date **Points** 1

You are suggested to use the teaching servers burrow.soic.indiana.edu or hulk.soic.indiana.edu or tank.soic.indiana.edu for practicing C programs.

Lab 8: Huffman code and data compression

For this lab, all students must have a good grasp on priority queue, pointer and tree traversing.

Huffman code

Given a fixed-length encoded text file containing only letters a, b, c, d, e, f with the following frequencies:

a, 45000 times, 000
b, 13000 times, 001
c, 12000 times, 010
d, 16000 times, 011
e, 9000 times, 100
f, 5000 times, 101

The file will take space of (45000+13000+12000+16000+9000+5000)*3 = 300000 bits

Rather than putting a fixed 3 bits for each character, we can allocate fewer bits to frequently used character character to compress the file. Huffman algorithm effectively generates prefix code for the given characters frequency. The prefix codes are bit codes such that no codeword can be prefix of another codeword.

In the example above, if we run Huffman algorithm, we will get the following prefix codes:
a - 0, 1 bit
b - 101, 3 bits
c - 100, 3 bits
d - 111, 3 bits
e - 1101, 4 bits
f - 1100, 4 bits

The total size = 1 * 45000 + 3 * 13000 + 3 * 12000 + 3* 16000 + 4 * 9000 + 4 * 5000 = 224000 bits

Here the size of the file will be much lower. No codeword is a prefix of another codeword, for that reason the codewords such as 1, 10, 11 etc are not used. This makes decoding easy because this way you do not need to look around the context to decide which character one piece of code corresponds to. For example, if the compressed binary data is 01111011100, we can easily know that the decompressed data was adbf (0-111-101-1100).

In this lab, you will be given character frequencies and will be required to generate the prefix code of the characters using Huffman algorithm.

The Huffman algorithm is given below:

```
HUFFMAN(C: list of characters with frequency)
        n = length(C)
        PriorityQueue PQ = C
        for i = 1 to n-1
                create a new node z
                z.left = x = EXTRACT_MIN(PQ)
                z.right = y = EXTRACT_MIN(PQ)
                z.freq = x.freq + y.freq
                INSERT(PQ, z)
        return EXTRACT_MIN(PQ)  // return the root of the tree
```

This algorithm builds a Huffman coding tree. The code of each character corresponds to the path traversing to each character node. Starting from the root, append '0' if you goes to the left, '1' if you go to the right. The Huffman algorithm can be seen as a greedy algorithm. Each step it chooses 2 nodes with minimum frequncy and combine them into one node with frequncy of the sum. You can think about why this leads to a optimal prefix code tree. (which means the size of compressed file with the given frequency is the smallest)

The tree node can be initiated like this

```
typedef struct node_struct
{
        char symbol;
        int freq;
        struct node_struct *left;
        struct node_struct *right;
} Node;
```

To maintain the priority queue, we can use C++ priority_queue, a sample code with priority queue for this Node structure given below, students are suggested to be familiar with working with priority queue with structure.

```
        #include <stdio.h>
        #include <queue>

        using namespace std;

        typedef struct node_struct
        {
                char symbol;
                int freq;
                struct node_struct *left;
                struct node_struct *right;
        } Node;

        struct NodeCompare
        {
                bool operator()(const Node &n1, const Node &n2) const
                {
                        return n1.freq > n2.freq;
                }
        };
```

```
int main()
{
        int n = 6;
        Node nodeList[6] = {
                                {'a', 45, NULL, NULL},
                                {'b', 13, NULL, NULL},
                                {'c', 12, NULL, NULL},
                                {'d', 16, NULL, NULL},
                                {'e', 9, NULL, NULL},
                                {'f', 5, NULL, NULL}
                        };

        std::priority_queue<Node, vector<Node>, NodeCompare> PQ;

        for(int i = 0; i < n; ++i)
                PQ.push(nodeList[i]);

        for(int i = 0; i < n; ++i)
        {
                Node t;
                t = PQ.top();                   // Getting the min item, the item still stays in th
e Queue
                PQ.pop();                       // Popping out this item

                printf("%c %d\n", t.symbol, t.freq);
        }
        return 0;
}
```

The output of this sample program is:
f 5
e 9
c 12
b 13
d 16
a 45

The node was extracted according to there minimum freq value as we defined in the NodeCompare structure. We can extract a node **t** and put the values in our own created node **x** the following way, it is always a good idea not to work with a priority_queue item once it has been popped out. We should copy the popped out item values and forget it.

```
// Our created new Node, we could also use malloc() to create this
Node *x = new Node;

Node t;
t = PQ.top();                   // Getting the min item, the item still stays in the Queue

PQ.pop();                       // Popping out this item

// Copying t value into our created x node, we will not work anymore with t after this point
x->symbol = t.symbol;
x->freq = t.freq;
x->left = t.left;
x->right = t.right;
```

Finally after the tree has been built we can print out the codewords the following way:

```
CALL printTree(Root, 0);

global character array codeword
printTree(T, depth)
{
        if (T.symbol is not 0)
        {
                print "Codeword for <T.symbol> is <codeword>"
                // print out codeword up to depth.
                Return
        }

        codeword[depth] := "0"
        printTree(Left of T, depth+1)

        codeword[depth] := "1"
        printTree(Right of T, depth+1)
}
```