```
/*************************
VGP310 Example code to build a Huffman tree from character frequencies
in min priority queue of the code symbols with character and freq.
W. Dobson   1-22-2018

This queue structure automatically sorts the Csym elements
in ascending order (least first)

1-22 modified to create encode table using strings of 1/0 characters
to represent bit stream.
************************/
//#include "stdafx.h"
#include <iostream>
#include <vector>
#include <queue>
#include <string>

using namespace std;

// Code symbol data structure (or node)
struct CsymNode {
      char c;
      unsigned int f;
      CsymNode *left, *right;

      // overloaded initialize constructor for this structure
      CsymNode(char c, unsigned f) {
            left = NULL;
            right = NULL;
            this->c = c;
            this->f = f;
      }
};

// Comparator boolean operator for use by the priority queue
struct compare_Csym {
      bool operator() (const CsymNode *a, const CsymNode *b) const
      {
            return a->f > b->f;
      }  // Note the > forces ascending order
};
```

```
// struct for encoder table nodes
struct EncoderNode {
      char c;
      string bits;  // now uses string for bits for convenience
};

/******************
Function to scan a string counting frequency of characters then putting them
into a priority queue.  Returns the number of codesymbols discovered.
******************/
unsigned freqcount(string& str, priority_queue<CsymNode*, vector<CsymNode*>,
compare_Csym>& pq) {
      unsigned symcnt = 0;
      vector<unsigned> freq(256, 0);

      // Loop scan the input string incrementing the count using the ASCII code
      // for the character as an address for the counter value.
      for (unsigned i = 0; i < str.size(); i++) {
            ++freq[(unsigned)str[i]];
      }

      // Loop scans the vector of freq counts and pushes any non zero
      // into a priority queue using the ASCII char equal to the index.
      for (unsigned i = 0; i < 255; i++) {
            if (freq[i] > 0) {
                  CsymNode *newnode = new CsymNode((char)i, freq[i]);
                  //cout << "debug: " << newnode->c << " " << newnode->f << endl;
                  pq.push(newnode);
                  ++symcnt;
            }
      }

      return symcnt;
}
```

```
/************
Function to build a Huffman coding tree
*************/
CsymNode *huffmantree(priority_queue<CsymNode*, vector<CsymNode*>,
compare_Csym>& pq) {
      CsymNode *left, *right, *top;

      top = NULL;
      while (pq.size() > 1) {
            left = new CsymNode((char)255, 0);
            right = new CsymNode((char)255, 0);
            left = pq.top();
            pq.pop();
            right = pq.top();
            pq.pop();
            // Create new parent node with freq being the sum
            // of the two child node freqs. We use the hex 0xff
            // for all non leaf nodes.
            top = new CsymNode((char)255, left->f + right->f);
            top->left = left;
            top->right = right;
            // Push the new node into the queue and the process repeats
            // with one fewer nodes in the queue each time.
            pq.push(top);
      }

      return top;  // returns last top node which is the root of the tree
}
```

```
/*************
Recursive function to traverse Huffman tree and build an encoder table
(dictionary)
**************/
void huffencodetable(CsymNode *root, vector<EncoderNode> *etable, string str) {
      EncoderNode n1;

      if (!root)       // terminates the recursion
            return;

      if ((unsigned)root->c < 255) {
            n1.c = root->c;
            n1.bits = str;

            etable->push_back(n1);
      }

      huffencodetable(root->left, etable, str + "0");
      huffencodetable(root->right, etable, str + "1");
}

/*************
Huffman encoder function creates a bitstream string of 0 and 1 char representing
the binary encoded data (using a string out of convenience).
*************/
void huffmanencoder(string& str, vector<EncoderNode>& etable, string& bitstream)
{
      bitstream.clear();  // start with a clean slate
      for (unsigned i = 0; i < str.size(); i++) {
            // look up each char in string and add code to bitstream string
            for (unsigned j = 0; j < etable.size(); j++){
                  if (str[i] ==  etable[j].c)
                        bitstream = bitstream + etable[j].bits;
            }
      }
}
```

```
/*************
Huffman decoder function takes a bitstream string input and decodes
using decoder tree referenced by root ptr.
***************/
void huffmandecoder(CsymNode *root, string& bitstream, string& decodestr){
    CsymNode *ptr;
    ptr = root;
    decodestr.clear();

    for(unsigned i = 0; i<bitstream.size(); i++){
        // if no char c=data in node traverse left/right as bitstream dictates
        if(bitstream[i] == '0' && (unsigned char)ptr->c >= 255 ){
            ptr = ptr->left;
        }
        else if(bitstream[i] == '1' && (unsigned char)ptr->c >= 255 ){
            ptr = ptr->right;
        }

        // if node contains data concatenate to str
        if((unsigned)ptr->c < 255){
            decodestr = decodestr + ptr->c;
            ptr = root;
        }

    }
}

/************
Recursive function to traverse Huffman tree and print contents
**********/
void printtree(CsymNode *root, string str) {
      if (!root)       // terminates the recursion
            return;
      //cout << hex << (short)root->c <<endl;
      if ((unsigned)root->c < 255) {     // I used 0xff as the null character
            cout << root->c << ": " << str.c_str() << endl;
      }

      printtree(root->left, str + "0");
      printtree(root->right, str + "1");
}
```

```
/************
Main program body
*************/
int main()
{
      cout << "Huffman Encode/Decode Builder" << endl;

      // Instantiation of priority queue object.
      // Note: vector parameter is required along with the compare operator
      priority_queue<CsymNode*, vector<CsymNode*>, compare_Csym> pq;
      queue<CsymNode *> q2;

      CsymNode *root;
      vector<EncoderNode> encodetable;
      string str, bstr, bitstream, decodestr;
      unsigned symcnt = 0;

      cout << "Enter a string : ";
      cin >> str;

      cout << "String length = " << str.size() << endl;

      // Call function to scan string and create freq list in priority queue
      symcnt = freqcount(str, pq);

      cout << "Symbol cnt = " << symcnt << "   pq size = " << pq.size() << endl;
      cout  << endl;
      cout << "Build Huffman tree:" << endl;
      root = huffmantree(pq);
      cout << "Root freq cnt = " << root->f << endl << endl;

      bstr.clear();  // be sure to clear string before each use
      // generate a huffman encoding lookup table
      huffencodetable(root, &encodetable, bstr);

      cout << "Encoder table contents:" << endl;
      for (unsigned i = 0; i < encodetable.size(); i++){
            if((unsigned)encodetable[i].c > 32 &&
                  (unsigned)encodetable[i].c != 127) // if printable char
                  cout << "char: " << encodetable[i].c << "  code bits: " <<
                        encodetable[i].bits << endl;
```

```
            else   // print hex codes
                  cout << "char: [" << (unsigned)encodetable[i].c <<
                        "]  code bits: " << hex << encodetable[i].bits << endl;
      }

      cout << endl << "Huffman tree contents:" << endl;
      bstr.clear();
      printtree(root, bstr);

      huffmanencoder(str, encodetable, bitstream);

      cout << endl << "Encoded bit cnt = " << (int)bitstream.size() << endl;
      cout << "Bits: " << bitstream << endl;

      huffmandecoder(root, bitstream, decodestr);

      cout << "Decoded bitstream: size = " << decodestr.size() << endl
            << decodestr << endl;
      return 0;
}
```