Assignment Objective: Demonstrate the synthesis of the semester skills through the development of a Huffman encoding program.

Requirements:

* Develop a Huffman encoding program, huffman, that prints the resulting codes for each character in an input text and produces a total byte count of the input text and its encoded counterpart. Developing a compressed output file is not within this project's scope.
* There will be two cpp files to be developed, huffman.cpp and PQ.cpp. The huffmanMain.cpp file is provided.
* PQ.cpp:
  + This will largely be a copy of the Priority Queue Assignment’s PQ, but modified as follows:
  + PQ.h – change the type definition of KEY\_VALUE to a keyValue class with the following members:

public:

int key;

char value;

keyValue \*left;

keyValue \*right;

keyValue(int key, char value,

keyValue \*left = NULL, keyValue \*right = NULL);

And change all uses of KEY\_VALUE to keyValue

The keys array type is changed from KEY\_VALUE \*keys to keyValue \*\*keys;

* + PQ.cpp – change the following:
    - Change this from a Max Priority Queue to a Min Priority Queue.
    - Change the swap() function to work with parameters of type keyValue \*&x and \*&y.
    - Change the structure usage to reference usage, e.g., kv.key to kv->key
    - Change all use of KEY\_VALUE to keyValue.
    - There are a few other changes, but they are all dealing with the fact that the PQ is working with pointers instead of values.
* huffman.h and Huffman.cpp:
  + Class encoding: // This is a helper class for building up an encoding during a Huffman tree traversal
    - Public members:
      * Int patCount; // This is the length of a bit encoding
      * Unsigned int pat; // This is the bit encoding
  + Class huffman:
    - Private members:
      * static const int FCOUNT = 128; // this is the range of characters that will be managed by this program; note, this is the printable ASCII character set. This program is not meant for arbitrary binary input files; if one wanted to do that, this would be set to 256.
      * PQ \*pq; // the queue to be used
      * int \*freqs; // an array of frequencies; this will be allocated in the constructor; it will be sized according to the number of possible distinct characters from the input file … FCOUNT.
      * int fileSize; // this is where the character count of the input file will be remembered.
      * keyValue \*root; // this is the root of the Huffman tree, once it is fully built.
      * void clear(keyValue \*p); // the private, recursive version of clear; causes the given node and all its children to be deleted.
      * void getEncodings(keyValue \*p, encoding code, encoding \*rv) const; // this is the private version of the recursive encoding traversal. This will fill the entries of the rv array that gets passed in. At each leaf, it saves the encoding in rv, indexed by the value of the keyValue.
      * void printPattern(char val, int freq, encoding code) const; // this is a helper function for printIt() and dumpTree() . It prints the val and the %frequency for that val, and then the binary encoding. See the p13correctOutput.txt file for the proper format.
      * void printIt(keyValue \*p, encoding code) const; // this is the private recursive extension of the Huffman tree traversal that results in the printing of each leaf via the printPattern() function.
      * void dumpTree(keyValue \*p, encoding code) const; // this is a traditional in-order tree traversal that prints every node, including interior nodes. The printing is done by printPattern(). This function is good for debugging.
    - Public members:
      * huffman(); // the constructor; it will cause, among other things, the PQ to be created with FCOUNT capacity.
      * ~huffman(); // the destructor; lots of details to be managed, but it can manage all that in three lines.
      * bool importFile(std::string fname); // this reads the input file, character by character. For each character it keeps count in freqs[], which is indexed by the character just read in. It also keeps track of the input file’s size.
      * void buildTree(); // This is used after the input file has been read in and processed. For each non-zero entry in freqs[], create a keyValue for that character and enter it into the PQ.
      * void clear(); // calls the private version passing the root of the tree.
      * encoding \*getEncodings() const; // creates the array rv[] with FCOUNT entries and then calls the private version to do the traversal and get all the encodings.
      * int getOutFileSizeBits() const; // These two functions return what their name says they would return.
      * int getInFileSizeBytes() const;
      * void printIt() const; // these are the public starters of the recursive traversals for printing the Huffman tree.
      * void dumpTree() const;
* There is a huffmanBreadCrumbs file that has some of the functions already implemented.
* You must not use any other data structure, whether built-in or otherwise.
* **Demonstrate your code works by doing the following:**
  + Create an appropriate Makefile
  + Compile your program as follows:

make

* + Run your program as follows, with the provided BibleNIV.txt.

./huffman BibleNIV.txt > huffmanOutput.txt

* **Deliverables:**
  + Zip the following files into a single zip file: PQ.h, PQ.cpp, huffman.h, Huffman.cpp, Makefile, and huffmanOutput.txt.
    - DO NOT put a project into D2L
  + Turned into class: a hardcopy of PQ.h, PQ.cpp, huffman.h, huffman.cpp, and huffmanOutput.txt, in that order.