Data Structures:

The data structures I used were and std::map, an std::priority\_queue, and a Node struct that I implemented myself. The Node structure was appropriate because it contained all the information the program needed to build the tree. The only problem with it was that only my leaf nodes needed a character value, so every node that wasn’t a leaf had an extra unnecessary character value. However, this is only a byte of memory so it is not an important problem. Additionally, I don’t know if it would be possible to implement the tree using a leaf node struct and a parent node struct, so it may not be a problem at all. Thus, I would not choose a different structure again if given the chance. The priority\_queue was also a good choice, although I had to create a separate comparison class to compare the nodes. Having a structure that sorted itself whenever I had to insert or remove something was very useful. Finally, the map was also a natural data structure to use because I could assign each character its unique frequency and its unique encoded Huffman value (in two different maps). Thus, when encoding a file, I need only check the map to see if it exists, and if it did, return the Huffman encoding of it. This was much more computationally efficient than traversing the tree for a value that may or may not exist.

Hardest Part:

The hardest part of this assignment was attempting to decode a bit spring. I didn’t understand that no Huffman code is a prefix of another. Once I realized this, I determined that all I had to do was traverse the tree according to the code, and if I reached a character, to add it to the string, and if I reached a nullptr, to return “”.

Computation Complexity:

The computational complexity of encoding was O(n), where n is the number of nodes in the tree. I only had to do this once, because I placed the Huffman code of each character into a map, thus when encoding a character I just checked that the character was in the map, and if it was I returned the code and if not I returned an empty string.

The computational complexity of decoding a string was O(n), where n is the length of the string to be decoded. Although I used a double loop to decode the string, I only had one value I was incrementing in both loops, with that value being the index used to traverse the string. I needed a double loop because I needed to start back at the top of the tree after finding each character.

The computational complexity of traversing the tree to get to a single character is

O(log n), while the complexity of traversing the entire tree is O(n), where n is the number of nodes in the tree.