Raj Garimella 919351073
Bradley Manzo 916953788

Programming Assignment 4 Report

The first program pqueue.h is an implementation of a binary heap priority queue that was heavily inspired by the implementation in class with a few notable changes. Our priority queue stores the root at index 0 instead of 1 and we adjusted the member to accommodate the change. It also takes an optional template class that is used to compare the objects. This allows the user to make a min or max queue of any comparable objects. The priority queue throws an std::underflow_error when the user tries to remove from or access an empty (because doing so causes it to underflow).

We tested pqueue.h using the program test_pqueue.cc. In addition to the provided tests, we added additional tests that interweave pushing and popping. We test min and max queues using std::greater and std::less respectively. We also tested the priority queue of pointers to MyClass class. We created 2 comparator classes (one for min and one for max) that take in 2 pointers to MyClass and compare the dereferenced objects that they store.

Our next program, bstream.h, allows us to read in and write out streams of binary code. This is achieved through two classes: BinaryInputStream, which contains Get() functions, and BinaryOutputStream, which contains Put() functions. The former works by reading in bit by bit from an std::ifstream using a buffer the width of a single byte which is refilled when empty. This function allows for the implementation of GetChar() and GetInt() which read 8 and 32 bits each. By ORing the result of GetBit() with the left-shifted the least significant bit (LSB) of our internal variable, we can construct the data bit by bit and return a complete value. PutBit() works similarly by inserting bit by bit, maintaining an internal buffer the width of a byte, and when full is flushed to the outstream. Since PutChar() and PutInt() must output the internal variable in the right order, it must be output from the most significant bit (MSB) to LSB. This is achieved by shifting right to the MSB of interest and ANDing with 1 to load the bit we are interested in into the char or int and output it (the char or int) to the ofstream.

We used test_bstream.cc to test bstream.h. We hard-coded the writing of a file and used our Get() functions to verify the proper input. Inversely, we tested our Put() functions by using them to write to a file and then hardcoded the reading of the file to verify the proper output. Furthermore, we tested the behavior of these functions when used individually as well as when interwoven together.

Our third program, huffman.h, makes use of the previous two. It contains the HuffmanNode class (provided) and the Huffman class, which contains the public Compress() and Decompress() methods. These functions are static which allows them to be called without instantiating an object. Compress() reads from an std::ifstream, counts the frequency of each character, records the number of characters, and saves the contents to a string. Then, it adds all the characters and corresponding frequencies to a min queue of HuffmanNode pointers. (To allow for this, we wrote the HuffmanNodePointerLess comparator class that functions much the same as those described in test_pqueue.h.) Next, the top 2 nodes are removed from the priority queue and a new one is added with the two nodes as its children and its frequency being the sum of the children's frequencies. This is repeated until only one node (the root of the Huffman Tree) remains in the queue. The tree is traversed with preorder traversal. When a leaf node is traversed, we use BinaryOutputStream to put a 1 bit followed by its stored character. It puts a 0 for the internal nodes. The paths to each node are recorded, with 0 representing going left and 1 going right. Next, the BinaryOutPutStream outputs the number of characters. Then it reads over the string storing the input and, for each character, outputs its corresponding path. Finally, it uses postorder traversal to delete the Huffman tree.

Decompress() starts by using the BinaryInputStream to reconstruct the HuffmanTree from the bottom up using a stack. It reads bits from the stream and pushes an empty node when it reads a 0. When it reads a 1, it gets a char and stores that in the node. Whenever the stack has 2 consecutive nodes on top that either store a char or are a parent, it pops them both off, makes them the children of a new node, and pushes the new parent to the stack. This repeats until the stack only has one node (the root). See comments for more details. Next, the function uses the BinaryInputStream to get an int (the number of characters) and repeats the following that many times: it reads bits from the BinaryInputStream while traversing the tree, going left when it reads a 0 and right when it reads a 1. When it encounters a node that stores a char, it outputs the char to the std::ofstream. Finally, the Huffman tree is deleted as described above.

The programs zap.cc and unzap.cc use the Compress() and Decompress() respectively. They take in the input and output files as the command line arguments and validate the input, printing an error and exiting if the input is wrongly formatted or the input cannot be opened. Then it performs the appropriate function call. Since the functions are static, we don't instantiate a Huffman object.