Ethan Wang & Briana Fedkiw
ECS 36C
20 May 2022

Programming HW4 Report

Part 1: Priority Queue

This program closely follows the Professor's implementation for binary heap with some alterations. Indexing was changed to start at 0 instead of 1. Push and pop used push_back and pop_back for automatic resizing, and the indexes were adjusted to fix the new indexing scheme. Percolate up and down were modified to call CompareNodes, an added function using the provided comparator to compare nodes so that the user can choose how to build the heap.

The tester tests each of the functions in the public API using different data types. Comparison classes were made to test using MyClass pointers in different heap structures. The tester also tests scenarios where duplicate keys are pushed.

Part 2: Binary Stream

Get/PutChar both perform their operations bit by bit, while Get/PutInt call Get/PutChar to go byte by byte. GetChar/Int initializes a variable as 0 and then shifts left while masking with the read bit/byte. GetInt required an additional mask to make sure the byte read contains only the wanted bits. PutChar/Int also initializes a variable and shifts right while masking with a bit/byte of all 1s to extract the right data. CHAR_BIT and sizeof(int) were used to make sure that the program works across different architectures.

The main idea in the tests was checking that reading and writing would always be in the correct bit by bit order it is done in. This means that Get/PutChar always does 8 bits, Get/PutInt 4 bytes, and Get/PutBit 1 bit. No matter how things are written or read, it will be in the specified order, starting from the very first bit of the input file or sequence being written. The testers also check errors along with the buffer flush.

Part 3: Huffman Compression, Zap, and Unzap

Compression was broken down into 4 steps: counting frequency, building the huffman tree, making the code table, and writing to the output file. Building the tree dynamically created node pointers to avoid unnecessary copying and the pqueue used a comparator class we made for the node pointers. The encoded tree and code table were created using a recursive traversal of the tree to keep track of whether a left node or right node is visited and whether it was an internal node or character node. Writing used the string containing the file contents to determine how many characters there are and the encoding sequence to write.

Decompression was broken into two steps: rebuilding the tree and writing to the new file. The algorithm for rebuilding the tree checked whether a 0 or 1 was read. If a 0 was read, an internal node pointer with the next two nodes as its children is dynamically created. This was recursively done to keep the structure of the tree. To write the output, based on the bit read in, the tree was traversed with 0 being left and 1 right until a leaf node is hit.

Both zap and unzap opened the files in the correct mode, and after checking that the files were opened properly, compress or decompress was run respectively without creating an object of the class.