0.1 Priority Queue

Compare to the priority_queue_heap.h that we learned from class, the root of pqueue.h starts from at 0 instead of 1, so one less space wasted; it has a second typename enable it to compare based on different value (which is later used in huffman); and it is based on a dynamic vector instead of a fix array. For implementation, in percolate down the comparison is done by the cmp, second typename function, and in percolate up it's done by the same comparison function but with position of parameters reversed to perform the opposite comparison; in Push, the item is inserted by push_back before increment of cur_size; in pop, the item does not only get moved to the end but also pop-ed out the vector by pop_back.

0.2 Bstream

For BinaryInputStream: GetChar initializes a char with 0 for all of its bits. Then it reads a bit, and change the char's bit accordingly. Doing it for 8 times, a char is read and returned. GetInt has the same reading strategy as Getchar, but loops a different number of times. The number of iteration is depended on how the machine represent a Int, whether it's 4 bytes or 2 bytes. The program get this information by sizeof(int) and then times 8 to get the number of iterations. For BinaryOutputStream: PutChar takes in a char from user, and initialized a Boolean to store the output bit. It gets each bit of the char by shifting the char a number of position to left, this number is start from the length of the binary string (8 bits in this case) -1 and iterative decrement to 0. Thus the number of iteration is based on the length of the binary string. So, for PutInt the only difference is number of iteration it based on sizeof int. GetBit and PutBit are two fundamental function that are provided. GetBit reads a bit from input stream and returns a boolean, PutBit takes a boolean and puts a bit. One note, the close() in BinaryOutputStream merely flush the bits stored in buffer into file, not close ofstream. So, in latter usage, BinaryOutputStream is closed by call to compression and fstreams is closed in zap.cc and unzap.cc.

0.3 Huffman Tree

In our very last data structure, we had to implement a Huffman tree. First, there are two parts which we had to approach sequentially which is Compression. Starting with creating the frequency table, we read in a file and count every unique character that is present in the ASCII table and up its frequency in a map container that mapped a character to an integer frequency. We then push the map's contents into a priority queue that contains Huffman Nodes. We then copy the logic from the HW4 file that outlines how to create the Huffman Tree. We then will output the structure of the tree in the binary file. The logic for it includes using pre-order traversal which when you hit a '0' node, you would call PutBit(0) and when you hit a leaf node, which we would find out using the **IsLeaf() function, probably the most important function in this entire program**. If it went inside the IsLeaf() condition, we would then PutBit(1) and as well put the character associated with the node so we would then do PutChar(n-≥data()). Once we do that, we use the benefit of the previous stated algorithm in making the Huffman tree to find out that the total amount of characters that is in the file is represented by the .Top() of the priority queue so we would then PutInt() the frequency of root node into the zap file which puts the frequency of all total characters in the file. The hardest part of compression is making generating the CodeTable, which states that whether you travel right, a bit string will be abstractly held such that whenever you travel down a right edge that there is a "1" value associated with. This is repeated if you travel left but instead replaced with a "0" value. Without loss of generality, when it hits a leaf it will then grab all the bits associated with the edges that caused you to travel towards it from the root node to that specific leaf node and then we'd insert the character associated with the leaf and then insert into the coding_table. We then use the coding_table to then convert all the characters in the input file to it's respective bit in the code_table. We output it into a zap file for compression.

For the second part of the code, decompression, it's must more harder in in terms of logic and utilizes alot of the recursion that we used to traverse the tree and make the tree. We must now unzap a compressed zap file. To begin, we must analyze the structure of the input file, the zap file. To begin, we must create a top-node that recursively recreates the Huffman-tree from the zapped file's structure. To begin, I have a recursive function that takes in the input of bits to check whether they are internal nodes or leaf nodes I used it to create a Huffman node to take the place of the left and right child's place in our program. The RecreateTree()'s first recursive call function's logic checks whether if the bit is 1, which in that case it will create a leaf node, and then we know that in our output_tree function, a 1 is followed by a leaf nodes character associated with it so we GetChar() and return a new Huffman node with all the old characteristics. If the bit isn't 0, a SUPER IMPORTANT thing we need to realize is that in a Huffman tree that if a node's character is '0', it implicitly implies it will always have 2 children so we call the recursive function again such that it will return another Huffman node that calls the recursive function again for it's left and right child and checks again if the bit is a 0 or 1 and repeats this logic until we are done. This is done to both sides of the left and right side of the root node so that we get the whole picture. We then have to get the encoded characters from the input, and this is where our Huffman structure tree as well as code_table comes in handy. When we first read in the bit from the encoded characters, we check if it's either 0 or 1, and then we check if from that point do we reach a leaf node by using isLeaf() because only leaf nodes contain characters, and if not don't then we keep going down the input string. This is very important because if we don't that means that there is no character associated with the bits 0 or 1, so it will keep on going and recursively check 11, 10, 01, 00 and keep going on depending on the structure of the Huffman tree. This is handy as it scales up from having simple tree's that consist of only 1, 01, and 00 as bit's pertaining to characters but once we have somewhat close to the maximum amount of characters allowed in the ASCII table, we start having bits that look like 010010 and have no way of knowing when to stop unless at the end of the left turn of the last character of the "010010" bit we hit a leaf node which means that we found the character that is encoded to "010010".

If the zap-file starts with a 1, that implies that we only have one node that is associated with a character and not 0. We simply just take in the character, and create one top node that has very basic parameters and can use GetCharacter() function that is in our program to output the character again.