/*Name: Trang Thai Van Nguyen A91104612
*Name: Wei Jen Huang A99045410
*Date: 1/25/2016
*Assignment: PA2
*File name: Checkpoint.pdf
*Description: Solutions to the checkpoint requirement. Includes a handsolved compression and
*uncompress of checkpoint1.txt and checkpoint2.txt along with explanation of how it was
*solved.
*/

Checkpoint1.txt:

abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd

store it in a counter, for the code we used a vector to keep track of it, just like how we would do it by hand.

freqs[97] = 10→ 10 'a'

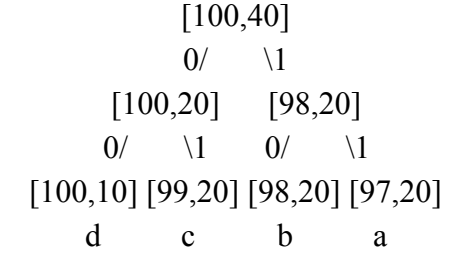
freqs[98] = 10 → 10 'b'

freqs[99] = 10 → 10 'c'

freqs[100] = 10 → 10 'd'

then I built the Huffman Tree by comparing the symbols and the count

Huffman Tree for checkpoint1.txt



Encoded by hand:

11100100111001001110010011100100111001001110010011100100111001001110010011100100111001

00

Encoded by our function:

11100100111001001110010011100100111001001110010011100100111001001110010011100100111001

00

The resulting code are the same since the Huffman Tree was built correctly. We did not encounter the situation where our hand-coded text is different from the compressor output.

Checkpoint2.txt

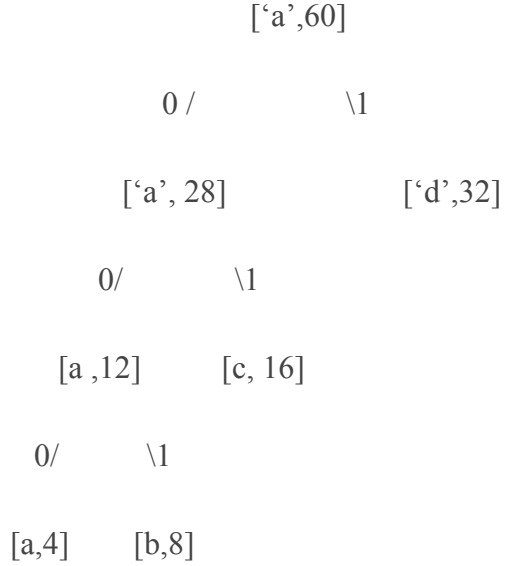
aabbbbccccccccddddddddddddddddddddddddddddddddddddcccccccccbbbbaa

encoded message by function:

00000000100100100101010101010101011111111111111111111111111111111010101010101
01010010010010010000000

frequency counts:
freqs[97] = 4 → 4 ‘a’
freqs[98] = 8 → 8 ‘b’
freqs[99] = 16 → 16 ‘c’
freqs[100] = 32 → 32 ‘d’

Manually construct a Huffman coding tree:



Manually encode the string:
00000000100100100101010101010101011111111111111111111111111111111010101010101
01010010010010010000000

Compare with my compressor output:
They match perfectly.

Description of how to build the tree and how to find the code-word of each byte:
First of all, I start by going through the frequency vector that I counted in my compress.cpp code. For each frequency count that is not 0, I create a new HCNode* with the count, and the character of the position, and its default construction (for example, freqs[97] = 5 means that there are count of 5 ‘a’ in my input file). After creating the node, I push it to my priority queue. Priority queue operator() was overloaded such that the node with the smallest count would have the highest priority. If there is a tie in count, then the bigger the ascii value of the character has

higher priority. After adding all the nodes to my queue, and while my size is > 1, I will extract the 2 highest priority value, remove it from the queue, combine their count, create a new node that takes the combined count value, the symbol of the first child (highest priority), assign c0 to the first child (highest priority) and c1 to the second child (second highest priority). Point the extracted nodes to their parents. Push the new node back to the queue, and repeat the algorithm until size == 1. When size == 1, I assign the root of the tree to the node, and remove it from the queue.

To find the code-word of each byte, first of all I find the pointer to the symbol pointed to by leaf member variables. I check if it does not have parents. If not, then I will just print out '0' and return the function. If it does have parent, then I use recursive encode, which will help me to print out the code through back-tracing and then print it out in the opposite order. So it will keep going up (check if *node is c0 or c1 of the parents) to keep traversing up the tree, and once I get to the root, it is my base case, and I output the code-word for each byte by the return of each recursion.