Authors: Jamie Shi (A12678538) and Sergelenbayar Tsogtbaatar (A12949154)
Date: February 15, 2017
Overview: Contains analysis for compression
Assignment #3
File Name: Checkpoint.pdf

When we use “set number” in Vim, we get the following results (with the first line being line 1). To save space, we did not include any frequencies of 0. To clarify, the last line (line 257) contains the binary encoding using the compress program.

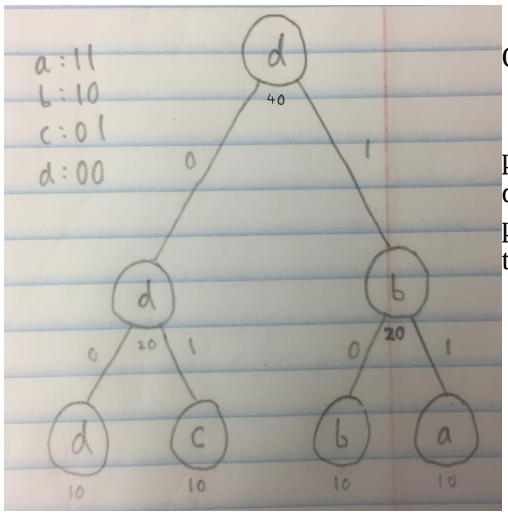
./compress checkpoint1.txt out.txt
Encoded checkpoint1.txt

Line number	Output
98	10
99	10
100	10
101	10
257	11100100111001001110010011100100111001001110010011100100111001001110010011100100

./compress checkpoint2.txt out.txt
Encoded checkpoint2.txt

Line number	Output
98	4
99	8
100	16
101	32
257	00000000100100100101010101010101011111111111111111111111111111111010101010101010100100100100100000000

For our Huffman algorithm, to maintain deterministic structure, nodes with the lowest frequency would have the highest priority. In addition, if two nodes have the same frequency, the one with the higher char (i.e. higher ASCII value) would have the higher priority. We set the parent node's symbol to the symbol of its left child, and the parent node's frequency to the sum of the frequencies of its two children. In our drawings, the symbol of each node is represent inside the circles, and the frequency of each node is shown right below it.

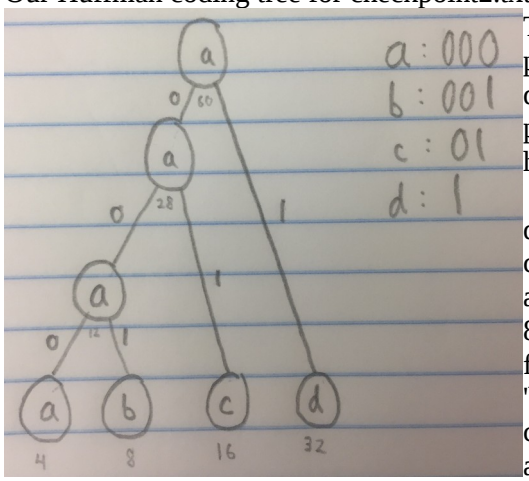


Our Huffman coding tree for checkpoint1.txt is shown on the left.

To build the tree above, we first create a priority queue, and push all the nodes with nonzero frequencies into this priority queue. (Nodes with the lowest frequencies have the highest priority, and cases of ties, nodes with the highest ASCII value have the highest priority.)

After these insertions, we pop two nodes from the priority queue, and these two nodes would be 'd' and 'c' since all nodes have the same frequency. The parent node is created and its symbol is set to the left child 'd', and its frequency is the sum of the frequency of its left and right child combined. Then we push the parent node, and we pop another two nodes from the priority queue, and we get 'b' and 'a' respectively since they both have a frequency of 10 (but node b's ASCII value is greater, so it's popped first and set to node1 whereas node a is set to node 2) while the parent node we just pushed has a frequency of 20. Then we create a parent node and assign its symbol to the left child 'b' and set its frequency to the sum of both frequencies of children nodes 'b' and 'a'; we push this parent node onto the queue. Then we pop another two nodes from the priority queue and get the parent nodes 'd' and 'b' with frequencies of 20. Then we create another parent of the parent node's and set its symbol to the left child which is 'd' and set its frequencies to the sum of both frequencies of its left and right child. Thus the queue is left with one node which we set be the root of the Huffman tree.

Our Huffman coding tree for checkpoint2.txt is shown below.



To build the tree to the left, we first create a priority queue, and push all the nodes with nonzero frequencies into this priority queue. (Nodes with the lowest frequencies have the highest priority, and cases of ties, nodes with the highest ASCII value have the highest priority.)

After these insertions, we would pop twice from the priority queue, giving us the nodes with the lowest frequencies. In this case, 'a' is popped first because it has the lowest frequency of 4, and 'b' is popped next because it has the next lowest frequency of 8. We then create a new parent node for node 'a' and 'b', with its frequency set to 12 (because this is the sum of node 'a' and node 'b"s frequencies), and its symbol set to 'a' because 'a' is the left child. We then push the parent into the priority queue, and pop another two nodes from the queue.

Popping these next two nodes would give us node 'a' (assigned to node1) and then node 'c' (assigned to node2) because these are the next two lowest frequency nodes. We create a new parent node for these two children, with its frequency set to 28 (the sum of its children's frequencies), and its symbol set to that of its left child, which is 'a' in this case. Then we push this parent node with frequency 28 and symbol 'a' onto the priority queue.

We again pop twice from the priority queue. The first node removed is node 'a' because it has the next lowest frequency of 28, and the next node removed is node 'd' because it has the next lowest frequency of 32. We create a parent node for these two nodes with frequency 60 and symbol 'a', because node 'a' is its left child. We then push this parent node onto the priority queue.

At this point, the size of the priority queue is not greater than 1, so we exit the while loop, and set the root to the remaining node in the priority queue, which in this case is node 'a' with frequency 60.

Our Huffman tree is constructed in such a way that if you go the left child, it's represented as 0, and if you go to the right child, it's represented as a 1. You can see this in our image we drew.

Explanation for encoding the string in checkpoint1.txt:

cat checkpoint1.txt = abcdabcdabcdabcdabcdabcdabcdabcdabcdabcd

We use the Huffman tree we drew to encode. When we encode a symbol, we first start at a leaf node whose symbol matches the symbol to encode. We then traverse up a level, and during this process, append '0' to a string if the node is its parent's left child, and append '1' to a string if the node is its parent's right child. We repeat this process of traversal and appending '0' or '1' accordingly until we reach the point where the parent node is null. After this, we have the string of the path encoded in reverse because we traversed up the tree, so we just reverse this string and write it to the ofstream.

For example, to encode 'a', we start at the leaf node whose symbol is 'a'. We then traverse up a level, and because the original node 'a' is the right child of its parent, we append a '1' to the string. We then traverse up another level, and because this next node is a left child of its parent, we append another '1' to the string. We then stop traversing up because we've reached the root node (whose parent node is null), so we just output the string “11” in reverse, which is still 11. Thus, the encoding for 'a' is 11.

To encode 'b', we start at the root node whose symbol is 'b'. We traverse up a level, and because the

original node 'b' is its parent's left child, we append a '0' to the string. We then traverse up another level, and because this next node is a right child of its parent, we append a '1' to the string. We then stop traversing up because we've reached the root node (whose parent node is null), so we just output the string “01” in reverse, which is 10. Thus, the encoding for 'b' is 10.

To encode 'c', we use the same process described above. The leaf node with symbol 'c' is the right child of its parent, so we first append a '1' to the string. We then append a '0' to the string because the next node is a left child of its parent. Traversal then stops because we have reached a point where the current node's parent node is null. Thus we output the string “10” in reverse, which is 01. Therefore, the encoding for 'c' is 01.

The encoding for 'd' is 00 because the leaf node with symbol d is the left child of its parent (so we append a '0' to the string), and the leaf 'd' node's parent is also the left child of its parent (so we append another '0' to the string). Then we have reached a point where the current node's parent is null, so we output the string “00” in reverse, which is still 00.

Therefore, to encode the input for checkpoint1.txt, we go through the encoding process explained above for each character. To explicate, the encoding for “a” is 11, so the encoding for “ab” is 1110 (11 represents the encoding for a, and 10 represents the encoding for b). The encoding for “abc” is 111001 (where the last two digits, 01 represents the encoding for c), and likewise the encoding for “abcd” is 11100100. Using this logic, our encoding by hand is
11100100111001001110010011100100111001001110010011100100111001001110010011100100
Our hand-coded encoded message matches that of the compressor output.

Explanation for encoding the string in checkpoint2.txt:
cat checkpoint2.txt = aabbbbccccccccddddddddddddddddddddddddddddddddddddddddcccccccccbbbbaa

To encode 'a', we start at the leaf node whose symbol is 'a'. We then traverse up a level, and because the original node 'a' is the left child of its parent, we append a '0' to the string. We then traverse up another level, and because this next node is a left child of its parent, we append another '0' to the string. We go up an additional level, and because this next node is also a left child of its parent, we append another '0' to the string. We then stop traversing up because we've reached the root node (whose parent node is null), so we just output the string “000” in reverse, which is still 000. Thus, the encoding for 'a' is 000.

To encode 'b', we start at the root node whose symbol is 'b'. We traverse up a level, and because the original node 'b' is its parent's right child, we append a '1' to the string. We then traverse up another level, and because this next node is a left child of its parent, we append a '0' to the string. We then go up another level, and because this next node is a left child of its parent, we append a '0' to the string. We then stop traversing up because we've reached the root node (whose parent node is null), so we just output the string “100” in reverse, which is 001. Thus, the encoding for 'b' is 001.

To encode 'c', we use the same process described above. The leaf node with symbol 'c' is the right child of its parent, so we first append a '1' to the string. We then append a '0' to the string because the next node is a left child of its parent. Traversal then stops because we have reached a point where the current node's parent node is null. Thus we output the string “10” in reverse, which is 01. Therefore, the encoding for 'c' is 01.

The encoding for 'd' is 1 because the leaf node with symbol d is the right child of its parent (so we append a '1' to the string). Then we stop traversing because the current node's parent is null, so we output the string “1” in reverse, which is still 1.

Therefore, to encode the input for checkpoint2.txt, we go through the encoding process explained above for each character. To explicate, the encoding for “a” is 000, so the encoding for “aa” is 000000 (000 represents the encoding for a, and 000 represents the encoding for the next a). The encoding for “aab” is 000000001 (where the last two digits, 001 represents the encoding for b), and likewise the encoding for “aabb” is 000000001001. Using this logic, our encoding by hand is

2*(000) + 4*(001) + 8*(01) + 16*(1) + 16*(1) + 8*(01) + 4*(001) + 2*(000) =
000000001001001001010101010101010111111111111111111111111111111101010101010101010010010
01001000000

Just to clarify: the first 2 letters are A's, so we have two instances of 000 – thus we do 2*(000). The next 4 letters are B's, so we have four instances of 001 – thus we do 4*(001). The next 8 letters are C's, so we have 8 instances of 01 – thus we do 8*(01), and so on. We then “append” all these together in order to get the encoded message.

Our hand-coded message did not differ from the compressor output.