Wright Kim
wdk7bj
postlab10

The very first thing I did was creating a vector of struct to store the unique characters and the number of frequencies of those characters. The only reason I created vector of struct and store the values there instead of storing directly to the heap was because I thought that it can be too complicated to store the values to the heap directly with my programming skill. And since the running time for inserting to the vector is constant, I thought it will not affect the performance of the program at all. Although the inserting operation's running time is constant, the program needed n running time to check whether the character is in the vector or not. The only way to check this is check every element in vector one by one, so the worst case is n although it will not be that bad in practice. The space complexity is 5n bytes, 1 byte for char, 4 bytes for int data type for frequency of unique char, and there are n stored in the vector.

Once the unique characters and the frequency associated to each character is stored in the vector, I created a heap called cheap so that I can insert the node, which carries the character and frequency informations, to the heap, which is based on vector. The time to copy the values to the heap is constant, since I can copy from vector index of 1 (since index 0 is left empty intentionally), and inserting the node into the heap is also constant time. There is a node in each heap, which stores a value for a character with data type char, frequency value with an integer, and two NULL pointers, which will be used later on to point left or right child of the node. Therefore, the space complexity will be an addition of 1 byte for char, 4 bytes for int, and 8 bytes for each pointers, which is 21 bytes for one node. Since there are n nodes in heap, the total space complexity will be 21n bytes.

Then I rewinded and re-read the file to find the total number of bits to use it later when print various things as instructed. This is simple constant running time program, since all it needs to do is increment a variable by one when it reads in valid char. Therefore, the Sspace complexity will be 4 bytes.

The last step before printing out results were creating the huffman tree. Since the nodes are stored in the heap, where the first index is empty, I used while loop to apply the algorithm of huffman tree to create one until the size of the heap is 2, where the first index is empty and the second index is huffman tree. The algorithm to create the huffman tree is as following. Inside the while loop, first create a new node called huffman and create a node pointer temp. Then assigned the minimum node inside the heap to the temp by calling deleteMin function. This will be the left child of huffman node and the frequency of the huffman node is the frequency of the temp node. Then, pop another node using deleteMin and made that node a right child of huffman node then added the frequency of temp node to the huffman node and pushed it back to the heap using insert, which will automatically place the huffman node in the right place inside the heap. This operation was done until the size of the heap was two. Calling deleteMin function on heap requires worst case n running time at first, but since the size of the heap decreases as the operation proceeds, the actual running time will be log n. The space complexity will be the bytes per node time the total number of nodes, which is 21n bytes, since 1 byte for char, 4 bytes for int, and 16 bytes for two pointers.

The last step of the huffman encoding lab was printing out each unique character with its prefix code and the encoded version of the text file. Since all of the unique char and the frequency values are at the leaf of the huffman tree, I created a method called 'prefix', which navigates from the root of the huffman tree to the leaf recursively. If the left child is not NULL called prefix method again by passing the left child as a new root and string 0 added to the prefix code. If right child is not NULL called prefix method by passing the right child as a new root with string 1 added to the prefix code. If the left and right child are both null, then I know I am at the leaf so printed out the character and the prefix code. Before printing the values, stored them using STL map, to print the text file in prefix code when re-read the text file. The running time for navigating the huffman tree is log n since it is a binary tree, and the space complexity is 5 bytes times the number indices of the map, which is 5n bytes.

After printing out the unique characters and its prefix codes, re-read the text file to print the text file in prefix codes. This can be done by using the map, which stored the values needed in previous step. For each character in the text file, it requires to find the map array, so the worst case running time is n, and since the values are already stored in previous step, there is no addition space required for this operation.

For the decoding part of the lab, I also used the vector of struct to store the character and prefix code values. The reason I chose to use this data structure was because since there will not be a huge text file input, it was permitted to use vector and it was familiar to use since it was once used in the encoding operation. The first part of given code reads in the unique character and its prefix codes. Therefore, first stored the character in the vector after creating a space, stored the prefix code at the same index to pair with the character. The insert operation of vector takes constant time and the space complexity is 16 bytes since there are two string values stored in the vector of struct.

Then, I created a huffman tree using the prefix codes obtained in previous step. First created a new node with allocated memory, and for the size of the vector storing the unique character and prefix code, I called the huffmanDec method which will create a huffman tree for me. If the first character of the prefix code is 0, then created a new node called leftChild, and if the root’s left pointer is NULL, made leftChild left child of the parent node. If it is not null, replaced the root’s left child with the new node after copying the values of it. Then removed the first prefix character and checked if the length of the prefix code is 0. If it is, then I am at the leaf so assigned the values to the leaf node properly. Same operation was done when the first character of the prefix code was 1, except every thing is to the right. Called this method recursively to create the huffman tree. The run time is log n because all I am doing here is check the left or right pointer and the length of the prefix code inside the binary tree. But the space complexity will be vary by the input file, but it will be basically 32n bytes since each node has 32 bytes, two strings and two pointers.

Finally, after creating huffman tree from previous operation, next step was to read all bits and decoded into a readable file. I created a method which can do this for me, by passing two huffman root nodes, and a string that has all bits stored. The reason passing two root nodes are because to pass the root node to the method since it is being called recursively, while the other root node will be used to move on to the left or right child. The worst case running time for this is log n since it is navigating huffman tree which is a binary tree and the in terms of space complexity, it is using the values that are already created so no new space added to perform the operation.