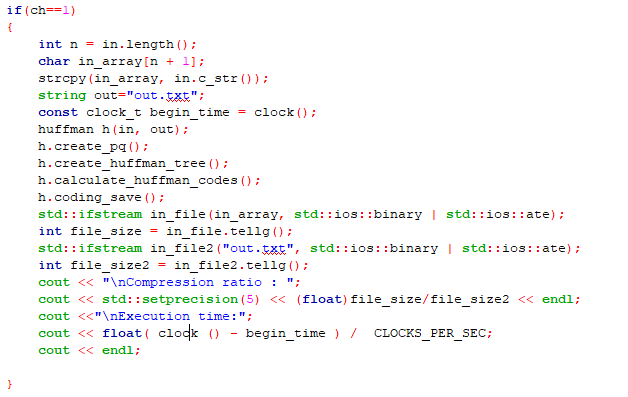
**Algorithms: Huffman assignment**

**Monique Ehab 4928  
Fagr Ahmed 4814  
Cloudia Kamal 4948**

* **Objective:**-Compression and decompression of each of text files, images and folders using Huffman encoding.
* **Discussion of Implementation:**

1. **Text Files:**

**Main:**  
-User chooses File.  
-User chooses (1) for compression and (2) for decompression.  
-*Compression*:

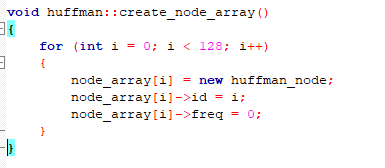


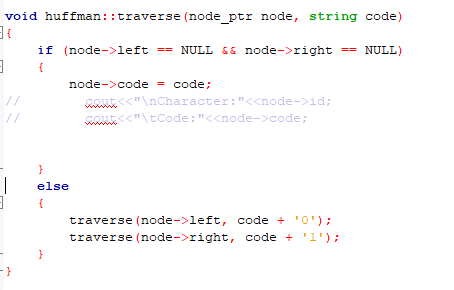
NB: functions will be explained in details in “Functions” section.

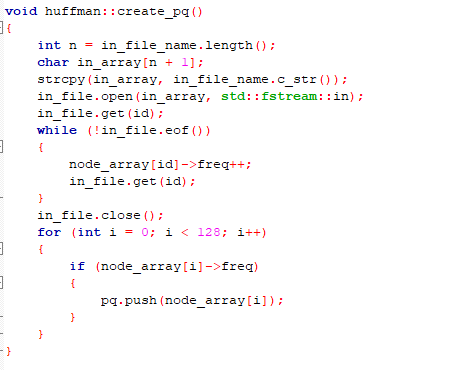
\*create a priority queue.  
\*create Huffman tree based on the priority queue created.  
\*Assign codes to the characters (Huffman code).  
\*Save coded characters into a compressed file.  
\*Calculate Compression Ratio= size of file/ size of output file.  
\*calculate execution time.

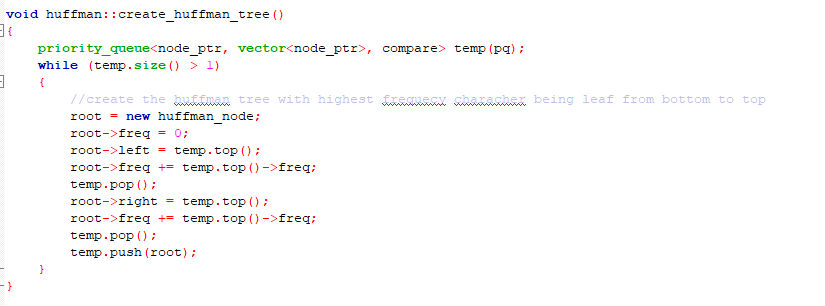
**Functions:**

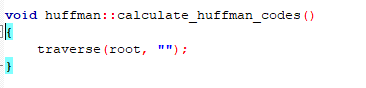
1-Create\_node\_array:  
-creating Huffman nodes.  
-assigning their Id a value from 1-128 (Ascii).  
-set their frequency with 0.

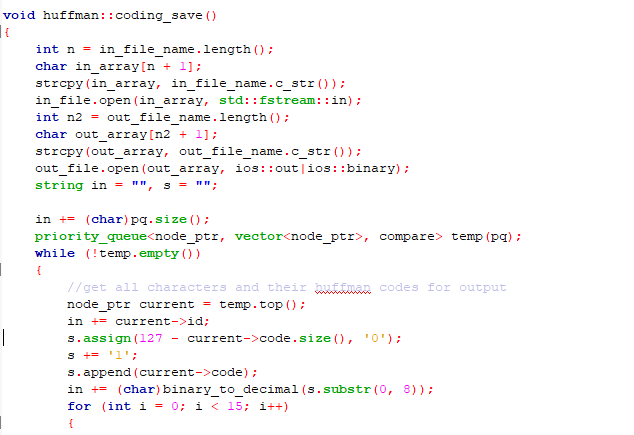


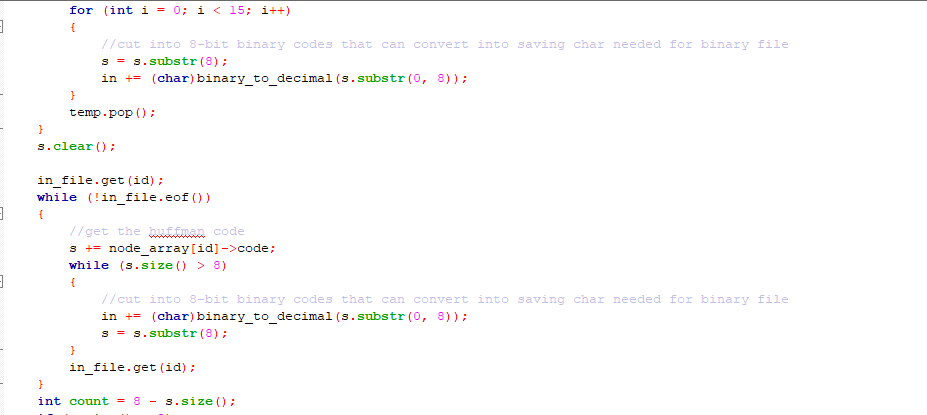
2-traverse:  
-sets the right nodes with 0   
-sets the left nodes with 1  


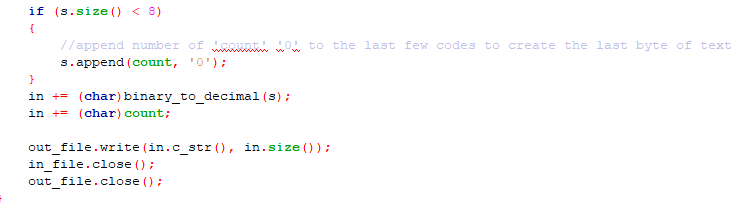
3-create priority queue:  
-reads file character by character.  
-adds frequency to nodes based on number of appearance.  
-adds them into new vector.  


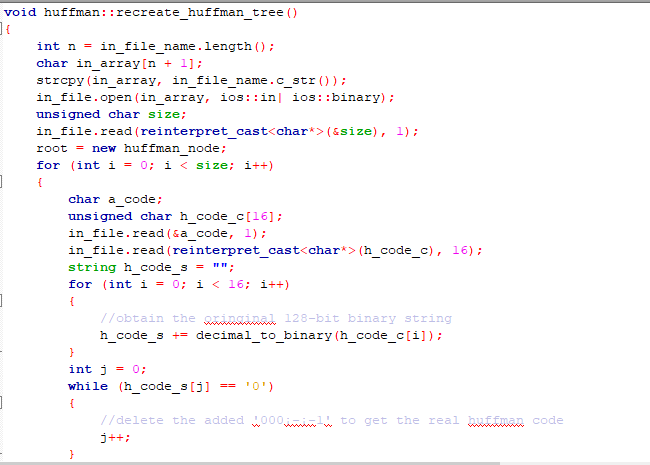
4-create Huffman tree:  
-create a new node with freq 0.  
-insert the top of the pq into the left node then pop it from queue.  
-insert the 2nd top of pq to the right node then pop it from queue.  
-push the sum of their frequencies into the pq again.  


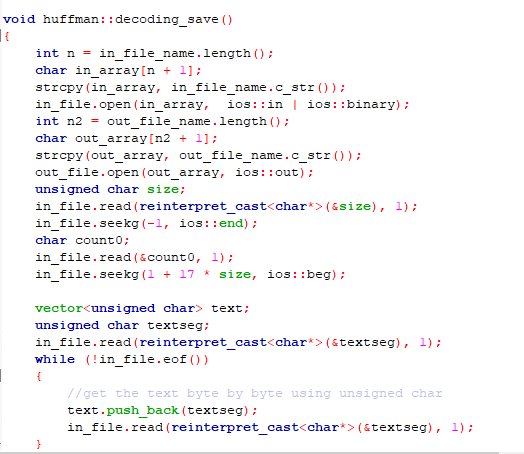
5-calculating Huffman code:  
-looping on all ids of the characters and assigning them codes using the traverse function.  


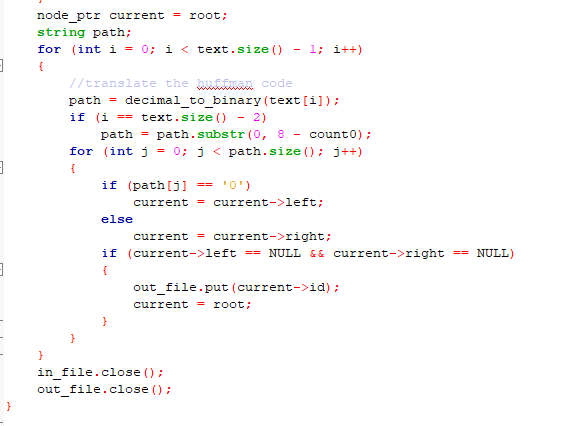
6-coding save:  
-save the size of the file in the 1st character of the file.  
-loop on number of elements in pq to get their ids and codes.  
-set the top with the id.  
-appending zeros and ending it with 1 indicating the start of the Huffman code appended.  
-each 8 characters are converted into ASCII, then popped.





7-recreating Huffman tree:  
-reads the compressed file.  
-1st character is the size of file.  
-loop on that size to get each character and its code.  
(since ASCII is 128, and each 8 bits are decoded, therefore skip 16 to read next code)  
-skip zeros till (1) which indicates start of Huffman code.  
-build tree based on code. (left is 0 and right is 1)  


7-decoding save:  
-skip header.  
-loop on size of file read character by character.  
-decode each charcter.  




1. **Image Files:**-same steps are done except for:  
   \*reads into extended ascii 256 instead of 128  
   \*in creating pq, we read file into a binary vector ifstream  
   \*in code saving, append 15 zeros (can’t extend that number)
2. **Folders:**

**-**same steps except for:  
\*in pq, when adding frequencies of characters, it reads the character in entire folder not in specific file.  
\*when saving file, adds “eof” to indicate end of file and start of new one and saves that number of eofs so that when decoding can know how many times to loop.  
\*in decoding, generates a random name for files, and knows when to separate from the number of eofs added in file.

**Complexity :**

**Compression :**

1-Building Huffman tree : O(nlogn) where n is the number of distinct characters in the file

2-Reading all the file : O( m\*n ) where m is the length of the file and using the array and incrementing frequencies through index is o(1) , therefore it is O(m)

**Total O(nlogn+m)**

**Decompression :**

1-Reading the compressed file O(k), where k is the length of the file

2-Finding the character in the Huffman tree O(nlogn)

Total O(nlogn+k)