**Compression**

**Background**

**Compression Technique:** Huffman Encoding

**Aim:** Compressing the data in the XML/JSON file, simply we reduce file size

**input:** input File that we want to compress it

**Output:** Compressed File that its size is smaller than the original file

Main Idea

Asweknow that every character has 8 bit or 1 byte size, The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters in the file.

In a way that make

🡪 **The most frequent character gets the smallest code**

🡪 **The least frequent character gets the largest code**

so, if we replace every input character code by its new code, we can reduce the total size of file.

**- Note:** the codes assigned to input characters are **Prefix Codes**, means the codes are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. so, no errors happen during decoding.

Main steps in Huffman Compression

1) Build a Huffman Tree from input characters.

2) Traverse the Huffman Tree and assign codes to characters.

3) Encode every character in input file using new codes to compressed file

**Steps to build Huffman Tree**

1. Get frequency of occurrences of every input character
2. Create a leaf node for each unique character and build a min heap (used as priority queue) of all leaf nodes, The value of frequency field is used to compare two nodes in min heap
3. Extract two nodes with the minimum frequency from the min heap
4. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
5. Repeat steps 3 and 4 until the heap contains only one node. The remaining node is the root node, and the tree is complete.

**Steps to assign codes to characters from Huffman Tree**

1. Traverse the Huffman tree starting from the root.
2. Maintain an auxiliary string.
3. While moving to the left child, string = string + 0
4. While moving to the right child, string = string + 1
5. When encounter with leaf node (input character node), set the code filed with the string
6. We will do that for every leaf node

**Steps to encode every character to compressed file**

Compressed file will consist of 2 parts (**header**, **actual data**)

Header will contain: 1) number of unique input character

2) every input character + its code

Actual data: compressed data (replacement of every character in file to its code)

**- Note:** we addHeader to compressed file that will be used to rebuild Huffman tree so we can do Decompression  
  
**Implementation details and Complexity**

there are many auxiliary functions, but I will focus on main functions



1. **build\_priority\_queue()**

/\*

gets frequency of each character in file and set the frequency of character node to it

then add the node of character that exists in the file to priority queue

if any character of 128 character in the Ascii Table does not exists in the file

then its character nodes will not add to priority queue

\*/

Time Complexity: **O(n)** as n is the number of characters in input file, as I visit every character in the file till end of the input file

Space Complexity: **O(n)** as n isnumber of nodes that added to priority queue

1. **build\_huffman\_tree()**

/\*

build the huffman tree ,making characters nodes as leafs

* the most frequency character will be leaf node near to root   
  (smallest depth) (small huffman code)
* the least frequency character will be leaf node far from root   
  (largest depth) (big huffman code)

\*/

Time Complexity: **O(nlog(n))** as iteration or repetition (as described above in section of (Steps to build Huffman Tree)) we be dependent on the size of priority queue O(n)  
and time for extract minimum nodes from priority queue(min heap) will be O(log(n))

Space Complexity: **O(n)** as I copy the main priority queue in temporary priority queue to execute steps described above in section of (Steps to build Huffman Tree)

1. **calculate\_huffman\_codes()**

/\* this the function only calls the auxiliary function called **traverse ()** in this form

**Traverse (root, "");**

So, I will focus on it \*/

* **void traverse (node\_ptr, string);**

/\*  
 i traverse the huffman tree to get huffman code for a character

save the huffman code in code data member of node of every character   
as described above in section of (Steps to assign codes to characters from Huffman Tree)

\*/

Time Complexity: **O(n)** as I visit every node in tree until reach leaf node and set its code to it

Space Complexity: **O(1)** as extra space needed

1. **compression\_saving\_to\_compressed\_file()**/\*

creating the Compressed File that will consist of 2 main parts

first we create header to use it in decompression  
then encode actual data

\*/

Time Complexity: **O(n^2)** as I visit every character in input file until end of file to encode every character and save it in compressed file and while visiting I used auxiliary function called **binary\_to\_decimal()** whichits time complexity is O(n) so the final is O(n^2)

Space Complexity: **O(n)** as I copy the main priority queue in temporary priority queue and I used string variable that its size depend on the frequency and number of characters in the input file