**Decompression**

**Background**

**Decompression Technique:** Huffman Decoding

**Aim:** Decompressing the data in .huf file, simply we restore the original data

**input:** compressed File which generated from compression operation

**Output:** Original File that its size is bigger than the compressed file

**Main Idea**

To decompress the compressed data, we need to rebuild the Huffman tree. So, we will need to know every character and its Huffman code.

In compression operation I store this required data in header section of compressed file so I will read it first to rebuild Huffman tree  
  
We go through the binary encoded data. To find character corresponding to current bits, we use following simple steps.

1. start from root and do following until a leaf is found.
2. If current bit is 0, we move to left node of the tree.
3. If the bit is 1, we move to right node of the tree.
4. If during traversal, we encounter a leaf node, we print character of that leaf node and then again continue the iteration of the encoded data starting from step 1.

No errors happen during the decompression operation as the codes are prefix codes

**Implementation details and Complexity**



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

/\*

reading the compressed file and read the header part to access character + its code

pass character and its code to auxiliary function called **build\_tree\_paths()**

we do that specific number of times equal the number of unique characters in the original file (simply the size of priority queue)   
with these we can rebuild the Huffman tree

\*/

* **build\_tree\_paths(string& huffman\_code, char character)**

/\*

As traversing we loop on every character in the Huffman code  
if it is ‘0’ go to left child node, if there is not left child node, we create it

If it is ‘1’ go to right child node, if there is not right child node, we create it

\* We continue looping until reaching the end of string, the latest appended node is the character node (leaf node)

\*/

Time Complexity: **O(n)** as it depends on the length of code string

Space Complexity: **O(k\*M)** as K is the number of nodes created to build character node path, M is the size of Node

So the complexity of the rebuild\_huffman\_tree() function

Time Complexity: **O(n^2)** as it depends on the number of unique characters in the original file(first byte in the compressed file) O(n) and after getting character and its code we use build\_tree\_paths() function O(n)

Space Complexity: **O(k\*M)** as K is the total number of nodes created to build Huffman tree, M is the size of Node

1. **decompression\_saving\_to\_decompressed\_file()**

/\*

reading the compressed file from the part of actual data (after header)

decoding this encoded data to original data and save it in the decoded file

as described above in section of (Main Idea)

\*/

Time Complexity: **O(n)** as n is number of characters to decompress

Space Complexity: **O(n)** as n is number of characters to decompress