File zipping using GREEDY Huffman encoder algorithm

DESIGN AND ANALYSIS OF ALGORITHMS MINI-PROJECT REPORT

Team Members :

IMMADISETTY DURGANJANEYULU(RA2011030010176)

KAKARAPARTHI SAI SEETHARAM(RA2011030010174)

PAVULURI SAHITHI KRISHNA(RA2011030010178)

CONTENT PG NO

CONTRIBUTION TABLE

S/NO	NAME	REG NO	CONTRIBUTION
1.	DURGANJANEYULU	RA2011030010176	Algorithm analysis
2.	SEETHARAM	RA2011030010174	Complexity analysis
3.	SAHITHI KRISHNA	RA2011030010178	Intro, problem statement

PROBLEM DEFINITION

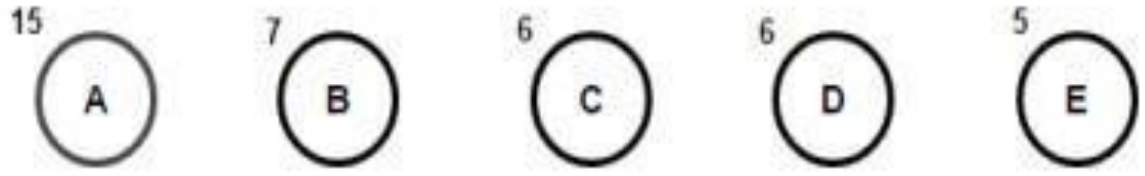
A ZIP file is a file in the ZIP format which implements lossless compression in order to reduce the volume of saved files.

In Windows, you work with zipped files and folders in the same way that you work with uncompressed files and folders.

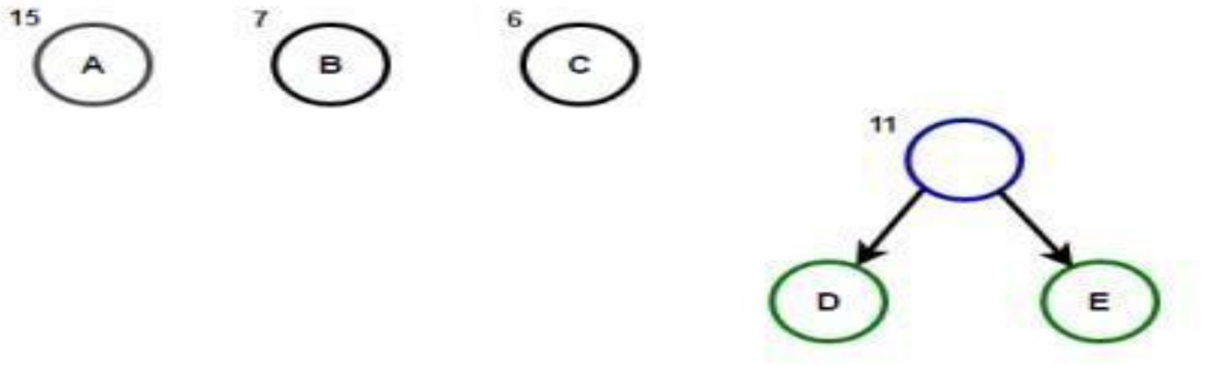
PROBLEM EXPLANATION

In the project ,we use Huffman's algorithm to construct a tree that is used for data compression. We assume that each character has an associated weight equal to the number of times the character occurs in a file, When compressing a file we'll need to calculate these weights.

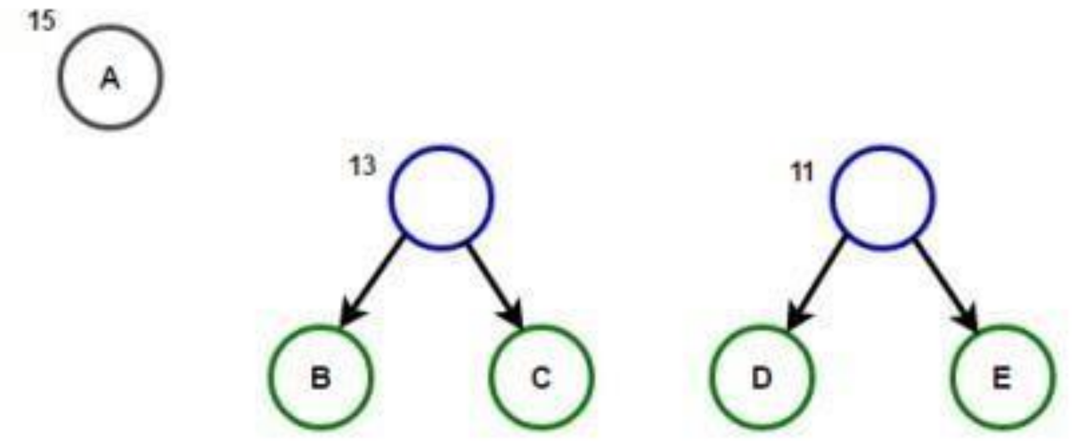
Step 1:



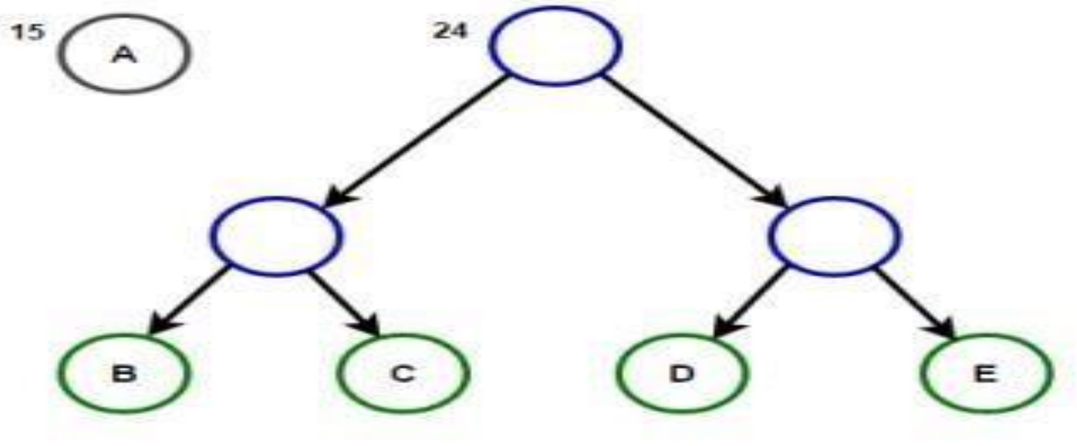
Step 2:



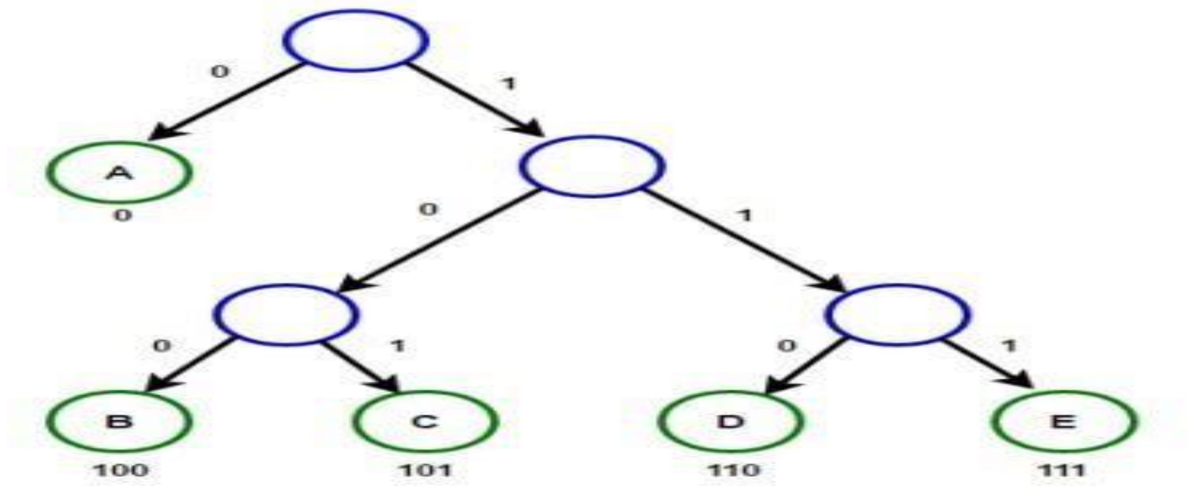
Step 3:



Step 4:



Step 5:



DESIGN TECHNIQUE USED

Huffman's algorithm assumes that we're building a single tree from a group (or forest) of trees. Initially, all the trees have a single node with a character and the character's weight.

Here in this case we are using Greedy approach .This property says that the globally optimal solution can be obtained by making a locally optimal solution (Greedy). The choice made by a Greedy algorithm may depend on earlier choices but not on the future. It iteratively makes one Greedy choice after another and reduces the given problem to a smaller one.
A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy

algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

Advantages of Greedy Approach:

- The algorithm is easier to describe.
- This algorithm can perform better than other algorithms (but, not in all cases).
- Typically have less time complexities.
- Greedy algorithms can be used for optimization purposes or finding close to optimization in case of NP Hard problems.

ALGORITHM OF THE PROBLEM

Huffman Coding Algorithm

1. Create a priority queue Q consisting of each unique character.
2. Sort then in ascending order of their frequencies.
3. for all the unique characters:
4. create a new Node
5. extract minimum value from Q and assign it to left Child of new Node
6. extract minimum value from Q and assign it to right Child of new Node

7. calculate the sum of these two minimum values and assign it to the value of new Node
8. insert this new Node into the tree
9. return root Node

PSEUDO CODE

Data Stíuctuíe used: Píioíity queue = Q   Huffman

```
(c)
n = |c|
Q = c foí i =1 to n-1 do z =
Allocate-Node () x = left[z] =
EXTRACT_MIN(Q)   y = right[z]
= EXTRACT_MIN(Q)
f[z] = f[x] + f[y]
INSERT (Q, z) return
EXTRACT_MIN(Q)
```

TIME COMPLEXITY

- Q implemented as a binary heap.
- line 2 can be performed by using BUILD-HEAP in O(n) time.
- FOR loop executed |n| - 1 times and since each heap operation requires O(lg n) time.
- ð the FOR loop contributes (|n| - 1) O(lg n)ð O(n lg n)
- Thus the total running time of Huffman on the set of n characters is O(n lg n).

IMPLEMENTATION OF CODE

```
#include <iostream>

#include <string>

#include <queue>

#include <unordered_map> using

namespace std;


#define EMPTY_STRING ""


// A Tree node
```

```
struct Node
{
    char      ch;
int freq;
    Node *left, *right;
};

// Function to allocate a new tree node
Node* getNode(char ch, int freq, Node* left, Node* right)
{
    Node* node = new Node();

    node->ch = ch;     node-
>freq = freq;     node->left =
left;     node->right = right;

    return node;
}

// Comparison object to be used to order the heap
```

```
struct comp
{
    bool operator()(const Node* l, const Node* r) const
    {
        // the highest priority item has the lowest frequency
return l->freq > r->freq;
    }
};

// Utility function to check if Huffman Tree contains only a single
node
bool isLeaf(Node* root) {
    return root->left == nullptr && root->right == nullptr; }

// Traverse the Huffman Tree and store Huffman Codes in a map.
void encode(Node* root, string str, unordered_map<char, string>
&huffmanCode)
{
    if (root == nullptr) {
return;}
```

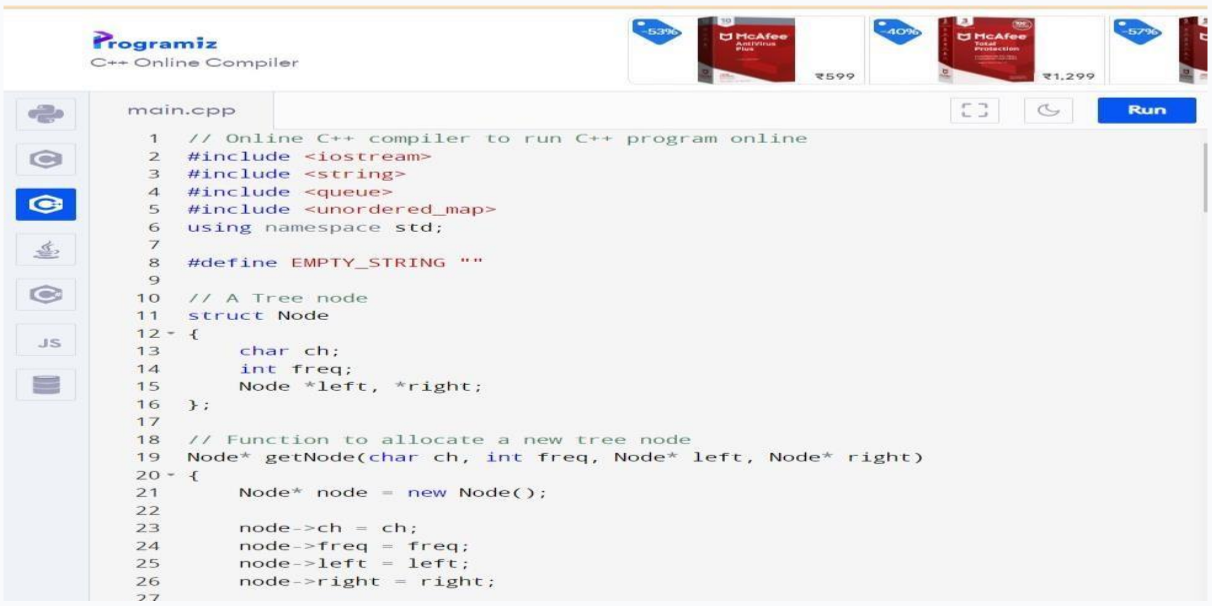
```
    // found a leaf node
if (isLeaf(root)) {
        huffmanCode[root->ch] = (str != EMPTY_STRING) ? str : "1";
    }

    encode(root->left, str + "0", huffmanCode);     encode(root-
>right, str + "1", huffmanCode);
}

// Traverse the Huffman Tree and decode the encoded string void
decode(Node* root, int &index, string str)
{
    if (root == nullptr) {
return;
    }

    // found a leaf node
if (isLeaf(root))
    {
        cout << root->ch;
   return;
   }

   index++;

   if (str[index] == '0') {
       decode(root->left, index, str);
   }
else {
       decode(root->right, index, str);
   }
}

// Builds Huffman Tree and decodes the given input text void
buildHuffmanTree(string text)
{
   // base case: empty string     if
(text == EMPTY_STRING) {
return;
   }
```

```
    // count the frequency of appearance of each character
    // and store it in a map
unordered_map<char, int> freq;
for (char ch: text) {
freq[ch]++;
    }

    // Create a priority queue to store live nodes of the Huffman tree
priority_queue<Node*, vector<Node*>, comp> pq;

    // Create a leaf node for each character and add
it     // to the priority queue.     for (auto pair: freq)
{
        pq.push(getNode(pair.first, pair.second, nullptr, nullptr));
    }

    // do till there is more than one node in the queue
while (pq.size() != 1)
    {
        // Remove the two nodes of the highest priority
        // (the lowest frequency) from the queue
        Node* left = pq.top(); pq.pop();

        Node* right = pq.top();    pq.pop();


        // create a new internal node with these two nodes as children
and

        // with a frequency equal to the sum of the two nodes'
frequencies.

        // Add the new node to the priority queue.


        int sum = left->freq + right->freq;

pq.push(getNode('\0', sum, left, right));     }


    // `root` stores pointer to the root of Huffman Tree

Node* root = pq.top();


    // Traverse the Huffman Tree and store Huffman Codes

    // in a map. Also, print them

    unordered_map<char, string> huffmanCode;

encode(root, EMPTY_STRING, huffmanCode);
```

```
  cout << "Huffman Codes are:\n" << endl;     for (auto pair:
huffmanCode) {
        cout << pair.first << " " << pair.second << endl;

    }


    cout << "\nThe original string is:\n" << text << endl;


    // Print encoded string

string str;

    for (char ch: text) {

str += huffmanCode[ch];

    }


    cout << "\nThe encoded string is:\n" << str << endl;

cout << "\nThe decoded string is:\n";


    if (isLeaf(root))

    {

        // Special case: For input like a, aa, aaa, etc.

        while (root->freq--) {

cout << root->ch;
```

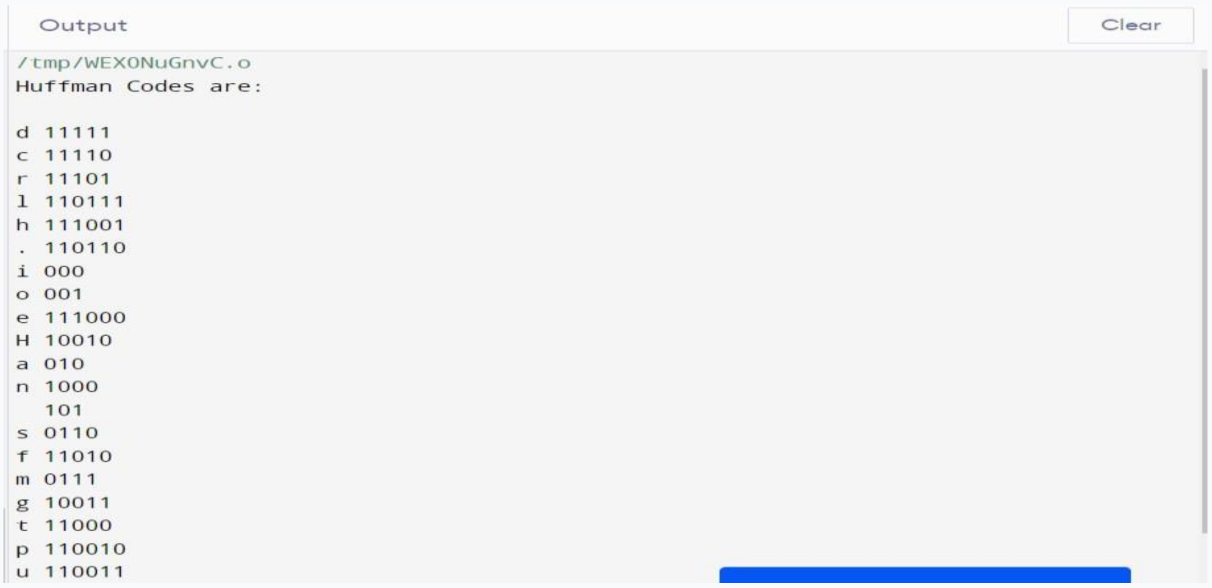
```
        }
}
else {
        // Traverse the Huffman Tree again and this time,
        // decode the encoded string
int index = -1;
        while (index < (int)str.size() - 1) {
decode(root, index, str);
        }
    }
}

// Huffman coding algorithm implementation in C++ int
main()
{
    string text = "Huffman coding is a data compression algorithm.";
buildHuffmanTree(text);

    return 0;
}
```

INPUT

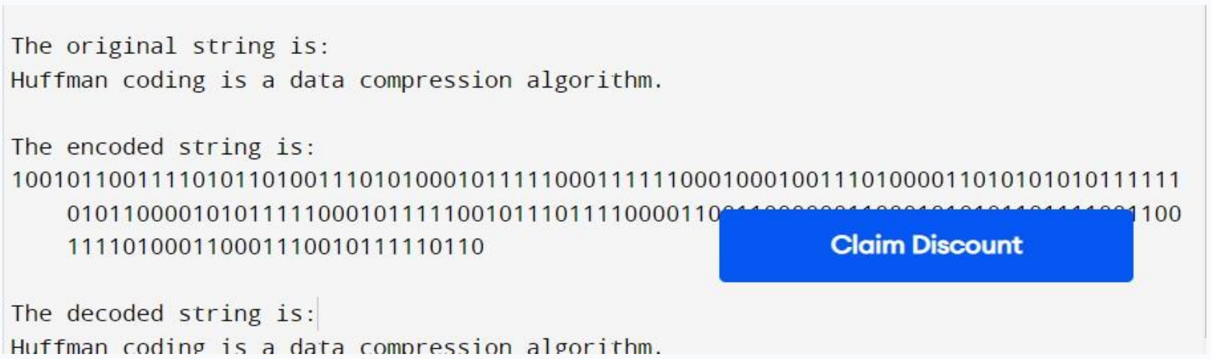


```cpp
// Online C++ compiler to run C++ program online
#include <iostream>
#include <string>
#include <queue>
#include <unordered_map>
using namespace std;

#define EMPTY_STRING ""

// A Tree node
struct Node
{
    char ch;
    int freq;
    Node *left, *right;
};

// Function to allocate a new tree node
Node* getNode(char ch, int freq, Node* left, Node* right)
{
    Node* node = new Node();

    node->ch = ch;
    node->freq = freq;
    node->left = left;
    node->right = right;
```

OUTPUT



```
/tmp/WEXONuGnvC.o
Huffman Codes are:

d 11111
c 11110
r 11101
l 110111
h 111001
. 110110
i 000
o 001
e 111000
H 10010
a 010
n 1000
  101
s 0110
f 11010
m 0111
g 10011
t 11000
p 110010
u 110011
```



```
The original string is:
Huffman coding is a data compression algorithm.

The encoded string is:
10010110011110101101001110101000101111100011111000100010011101000011010101010111111
    01011000010101111100010111110010111011110000110[illegible]1100
    11110100011000111001011111110110

The decoded string is:
Huffman coding is a data compression algorithm.
```

CONCLUSION

We can conclude that, Greedy Huffman encoding algorithm is the best suitable and efficient algorithm for data compression. And the advantage to using a greedy algorithm is that solutions to smaller instances of the problem can be straightforward and easy to understand.

REFERENCES

- Geeks for Geeks
- Javapoint
- Programiz
- f reeCodeCamp