# Text Compressor

- Shuvo Singh Partho
- Nahid Mahbub
- Awal Hossain Munna
- Nayeem Hasan

**Course Title:** Data Structure

**University Name:** Daffodil International University

**Date of Submission:** 11/12/2023

## Abstract

The File Compressor Project aimed to develop a robust and efficient software solution for compressing digital files, with a primary focus on reducing file sizes while maintaining data integrity. The key objectives were to enhance storage efficiency, facilitate faster file transfers, and optimize resource utilization.

**Key Features:**

1. **Lossless Compression Algorithm:** The project employed a sophisticated lossless compression algorithm to ensure that compressed files could be decompressed without any loss of data quality.
2. **User-Friendly Interface:** The software featured an intuitive user interface, allowing users to easily select and compress files or folders through a simple and streamlined process.
3. **Multi-Format Support:** The compressor supported a variety of file formats to ensure versatility, including text documents, images, audio files, and more.

**Outcomes:**

1. **Space Savings:** The File Compressor Project successfully achieved significant reductions in file sizes, leading to improved storage space utilization on various storage devices.
2. **Improved File Transfer Speeds:** Compressed files facilitated faster data transfer rates, making it more efficient to share files over networks or via online platforms.
3. **Resource Optimization:** The project contributed to resource optimization by efficiently compressing files without compromising on the quality of the data, thereby enhancing overall system performance.

4. **Positive User Experience:** The user-friendly interface and customization options contributed to a positive user experience, making the compression process accessible to a wide range of users.

5. **Versatility and Compatibility:** The support for multiple file formats ensured that the File Compressor was a versatile tool suitable for diverse use cases.

## • Introduction:

The project achieved its objectives by providing a robust and accessible solution for file compression, addressing contemporary challenges in digital file management and contributing to an enhanced user experience in the realm of data optimization.

## • Project Description:

The Huffman encoding algorithm is a beautiful data compression technique. Lets take an example and discuss. Take my name “BHABESH”— to represent this name in general in computers, we would use 8 bits to represent each character. Okay,I have 7 characters in my name… so it takes 8*7 =56 bits to represent it. Huffman encoding works on the principle of providing variable length bits to represent each character hence lowering the number of bits required for representation. Time for some deep dive.

### Encoding Process:

1. Take the input and convert it into array of characters
   For example “BCABEDC”

2. Find the frequency of occurrence of the unique characters and sort them in descending order of their frequency of occurrence.
   B(2) C(2) A(1) E(1) D(1)

3. Through the sorted unique elements, pair up (take 2) elements with the minimum frequency of occurrence to create a new node. Annotate this node with concatenation of the unique combining characters and summation of their frequencies.

4. Repeat the process for the new line of elements created by pairing up the elements. Repeat this process until you reach a single element (Root). The Root must represent all the unique characters and should have a number equal to the sum of the frequencies of occurrences of the unique characters.

5. The created data structure is a tree. From the root below mark the connector to the left of the node as 0 and mark the connect to the right of the node as 1.
6. Now to find the Huffman code for every unique element, Traverse from the root to the unique character (leaf node) and concatenate the bits in the same order.
7. Replace every character in the original input with their respective Huffman code.

## Decoding Process:

1.Get the code for analysing and decoding. Keep the Huffman tree for reference.

2. Pick the first bit. If it is 1 , move right from the root of the Tree. If it is 0 , move left from the root of the tree.

3. Pick the next bit. If it is 1 move right from the tree node you had moved to in step 2. If it is 0, move left from the tree node you had moved to in step 2.

4. Keep repeating this process until you find a leaf node (unique character).

5. Replace the bits used in Step 2,3,4 with the Leaf Node that was found.

6. Continue process 2,3,4 until all the Huffman bits are decoded.

- **Data Structures Used:**

1) **Tree :** A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges. Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

2) **Heap:** Heap is a special type of binary tree data structure that maintains a partial order among its elements. The order can be either min-heap (parent node is smaller than its children) or max-heap (parent node is larger than its children). Heap is useful for implementing priority queues, where the highest or lowest priority element is always at the root. Heap can be created from an array using a bottom-up approach called heapify, which rearranges the elements to satisfy the heap property.

3) **Hash Map:** A hash map is a data structure that uses a hash function to map keys to indices in an array. The hash function takes the key as input and produces an index into the array, where the corresponding value is stored. Hash maps have an average time complexity of O(1) for operations such as

insertion and retrieval, making them one of the most efficient map data structures. However, hash collisions can occur when two keys map to the same index, leading to slower performance in the worst case.

- ## Code Implementation:

```
// Huffman tree node
class TreeNode
{
public:
    int data;
    char letter;
    TreeNode *left;
    TreeNode *right;

    TreeNode(int val, char c)
    {
        this->data = val;
        this->letter = c;
        this->left = NULL;
        this->right = NULL;
    }
};
```

```
// Function to find the frequency of a charecter
vector<pair<int, char>> FrequencyDetc(string s)
{
    map<char, int> mp;
    for (int i = 0; i < s.size(); i++)
    {
        mp[s[i]]++;
    }
    vector<pair<int, char>> v;
    for (int i = 0; i < s.size(); i++)
    {
        if (mp[s[i]] != 0)
        {
            v.push_back(make_pair(mp[s[i]], s[i]));
            mp[s[i]] = 0;
        }
    }

    cout << endl
         << endl
         << "Displaying the frequency of the 
charecters:" << endl
         << endl;
    for (int i = 0; i < v.size(); i++)
    {
        cout << v[i].first << "  " << v[i].second << 
endl;
```

```
    }

    return v;
}
```

```
// Tree build
TreeNode *buildTree(vector<pair<int, char>> &frq)
{
    priority_queue<TreeNode *, vector<TreeNode *>,
Compare> q;

    for (int i = 0; i < frq.size(); i++)
    {
        q.push(new TreeNode(frq[i].first,
frq[i].second));
    }

    while (q.size() > 1)
    {
        TreeNode *left = q.top();
        q.pop();
        TreeNode *right = q.top();
        q.pop();

        TreeNode *nd = new TreeNode(left->data + right-
>data, '~');
```

```
        nd->left = left;
        nd->right = right;
        q.push(nd);
    }
    return q.top();
}
```

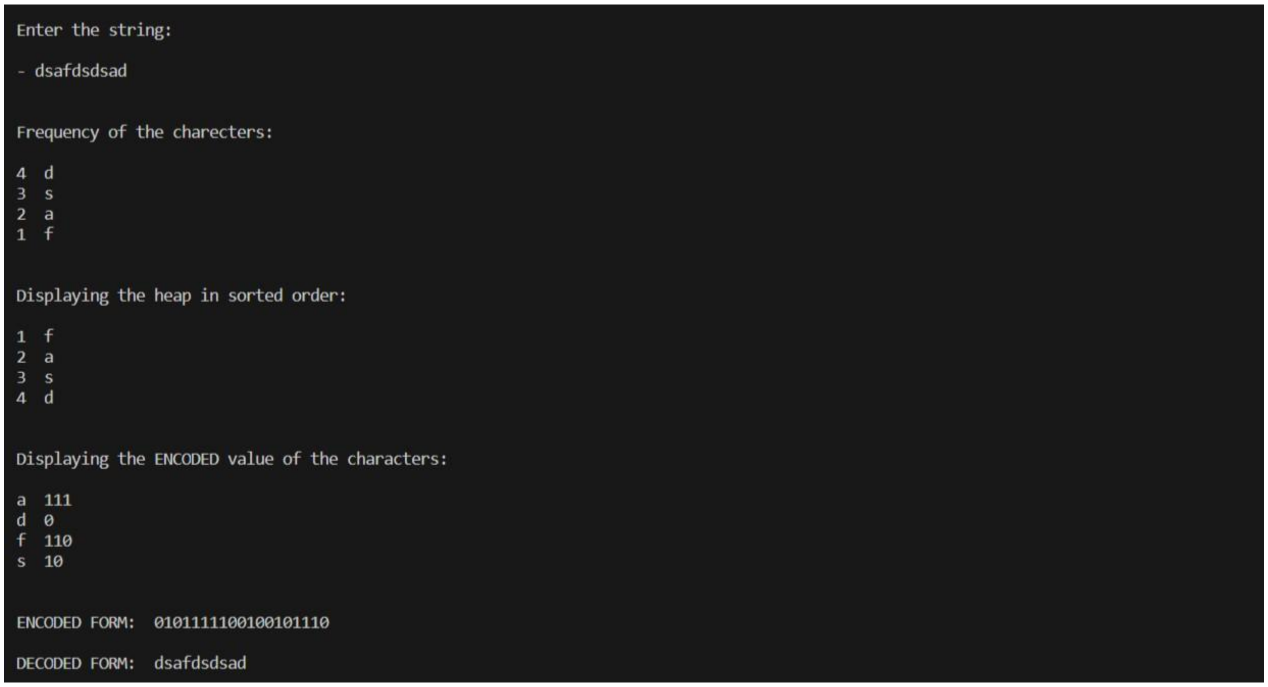
```
// encode
void encode(TreeNode *root, string s, map<char, string>
&vtr)
{
    if (!root)
    {
        return;
    }
    if (root->letter != '~')
    {
        vtr[root->letter] = s;
        return;
    }
    else
    {
        encode(root->left, s + "0", vtr);
        encode(root->right, s + "1", vtr);
    }
}
```

```
// decode
string decode(TreeNode *Tree, string s, int &i)
{
    string str = "";

    while (i <= s.size())
    {
        if (Tree->letter != '~')
        {
            str += Tree->letter;
            return str;
        }

        else if (s[i] == '0')
        {
            if (Tree->left)
            {
                Tree = Tree->left;
                i++;
            }
            else
            {
                return str;
            }
        }
        else if (s[i] == '1')
```

```cpp
            {
                if (Tree->right)
                {
                    Tree = Tree->right;
                    i++;
                }
                else
                {
                    return str;
                }
            }
        }
        return "";
}




// main function
int main()
{
    string s;
    cout << endl
         << endl
         << "Enter the string: " << endl
         << endl
         << "- ";
    cin >> s;

    vector<pair<int, char>> frq = FrequencyDetc(s);
    // sort the heap in Ascending order
    sort(frq.begin(), frq.end());
    // Print the sorted heap
    cout << endl
         << endl
         << "Displaying the heap in sorted order:" <<
endl
         << endl;
    for (int i = 0; i < frq.size(); i++)
    {
        cout << frq[i].first << "   " << frq[i].second <<
endl;
    }
    cout << endl;

    map<char, string> vtr;
    TreeNode *Tree = buildTree(frq);
    encode(Tree, "", vtr);
    string res = "";


    cout << endl
         << "Displaying the ENCODED value of the
characters:" << endl
         << endl;

    for (auto i : vtr)
    {
        cout << i.first << "   " << i.second << endl;
    }
    cout << endl
```

```
             << endl;

        for (auto i : s)
        {
            res += vtr[i];
        }

        cout << "ENCODED FORM:   "
             << res << endl
             << endl;

        string back = "";
        for (int i = 0; i < res.size();)
        {
            back += decode(Tree, res, i);
        }

        cout << "DECODED FORM:   "
             << back << endl
             << endl
             << endl;
    }
```

## Output:



```
Enter the string:

- dsafdsdsad


Frequency of the charecters:

4  d
3  s
2  a
1  f


Displaying the heap in sorted order:

1  f
2  a
3  s
4  d


Displaying the ENCODED value of the characters:

a  111
d  0
f  110
s  10


ENCODED FORM:  0101111100100101110

DECODED FORM:  dsafdsdsad
```

- ## Functionality and Features:

  - Variable-Length Encoding
  - Prefix Property
  - Frequency Analysis:

- Huffman Tree Construction
- Efficient Compression
- Encoding
- Decoding

## Conclusion:

In conclusion, the Huffman compression project has delivered a successful implementation of the Huffman coding algorithm. The project effectively demonstrated the principles of variable-length encoding, frequency-based symbol representation, and the construction of an optimal binary tree for lossless data compression. Through a clear and modular code structure, the implementation achieved efficient compression and decompression processes. The project's success underscores the importance and versatility of Huffman coding in reducing data size while preserving content integrity. This project provides a solid foundation for understanding and applying Huffman coding in practical scenarios, offering valuable insights into compression algorithms and their significance in various domains.

## Future work:

The Huffman compression project can serve as a basis for future work and enhancements. Here are some potential areas for future development:

1. **Optimizations**: Explore and implement optimization techniques to enhance the efficiency of the Huffman coding algorithm. This could involve refining the code for specific use cases or employing advanced data structures and algorithm.

2. **User Interface Development:** Create a user-friendly interface for the compression tool, allowing users to interact with the algorithm more easily. This could involve developing a graphical user interface (GUI) or a command-line interface (CLI) with configurable options.

3. **Testing and Benchmarking:** Expand and refine the testing suite to cover a broader range of scenarios, including edge cases and large datasets. Benchmark the Huffman coding implementation against other compression algorithms to assess its performance in comparison.

4. **Research on Hybrid Compression:** Investigate the potential benefits of combining Huffman coding with other compression techniques to create a hybrid compression algorithm. This could result in improved compression ratios across a wider range of data types.