Semester Project

Submitted to:

Ma’am Mariam Hida

Submitted by:

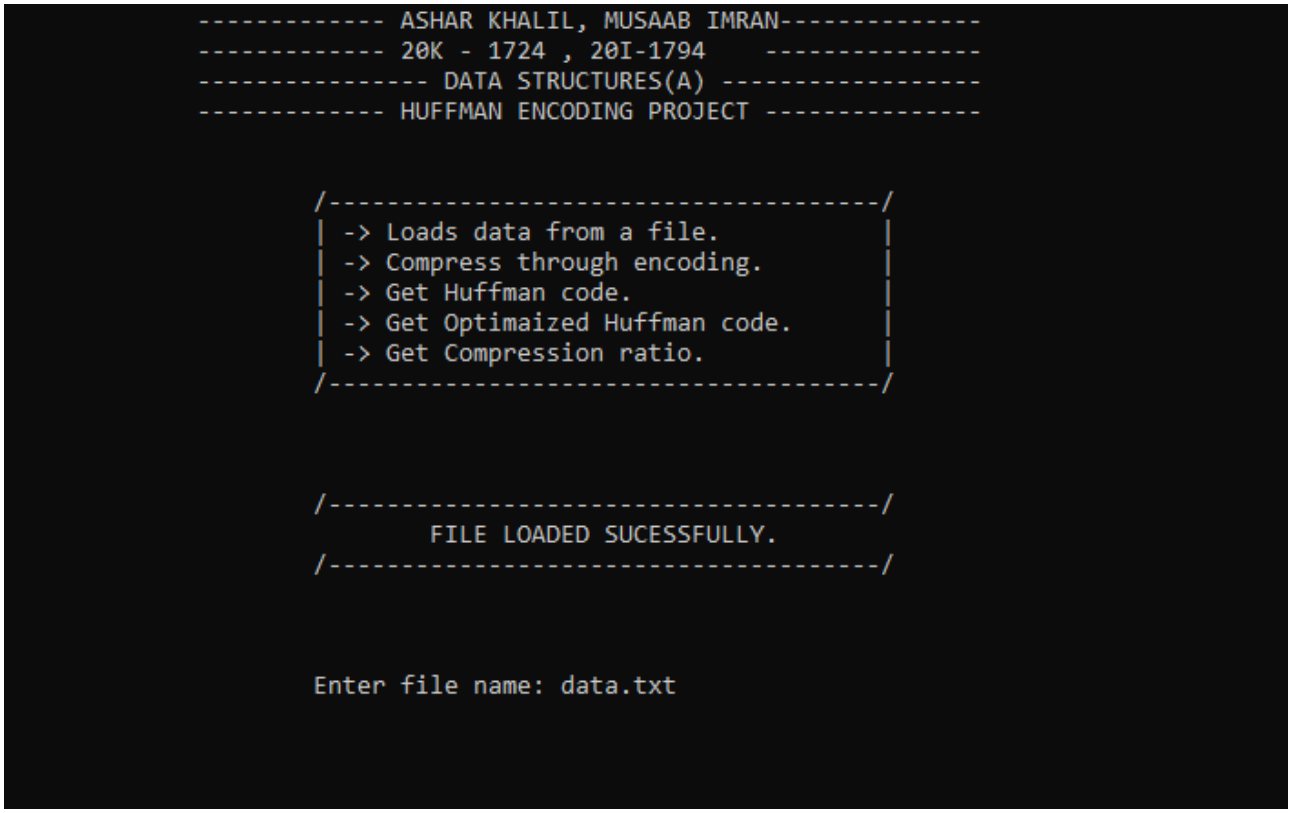
Musaab Imran (20i-1794)

Ashar Khalil (20k-1724)

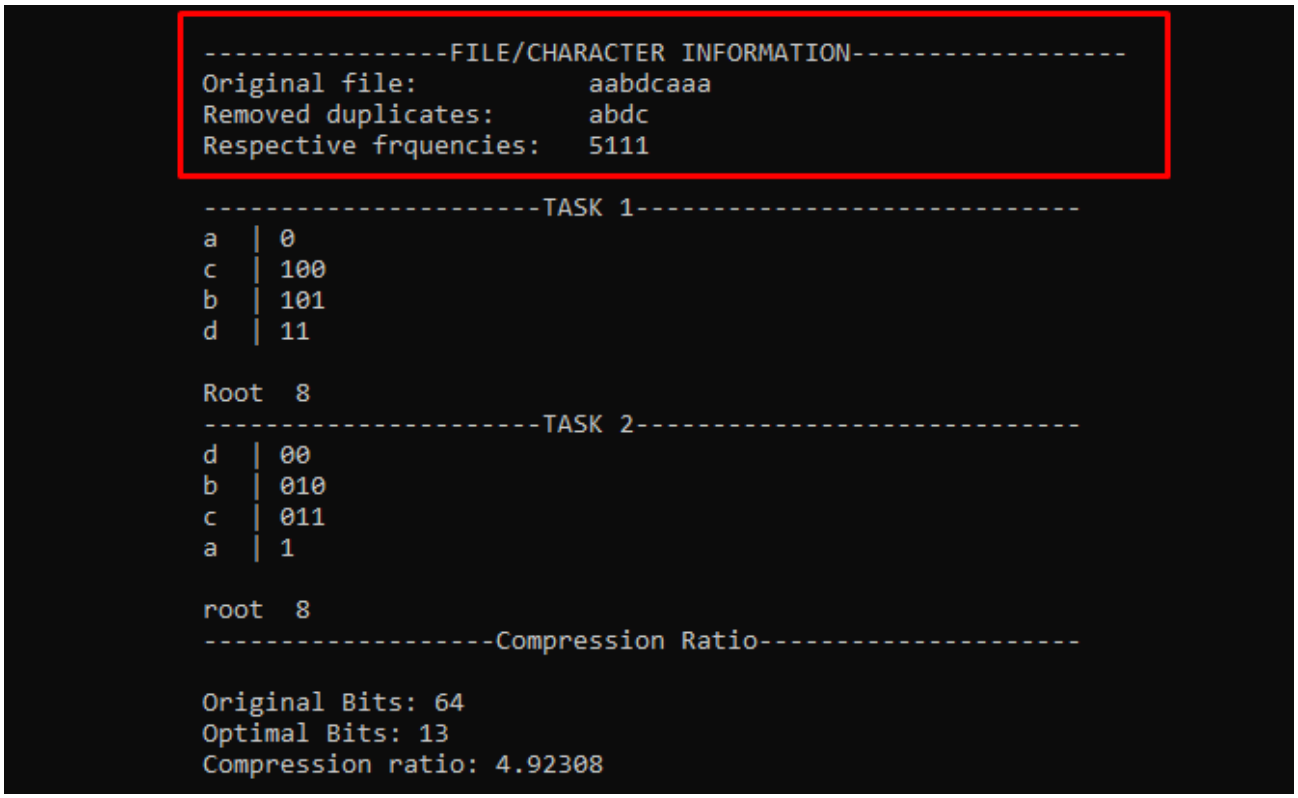
CS – A

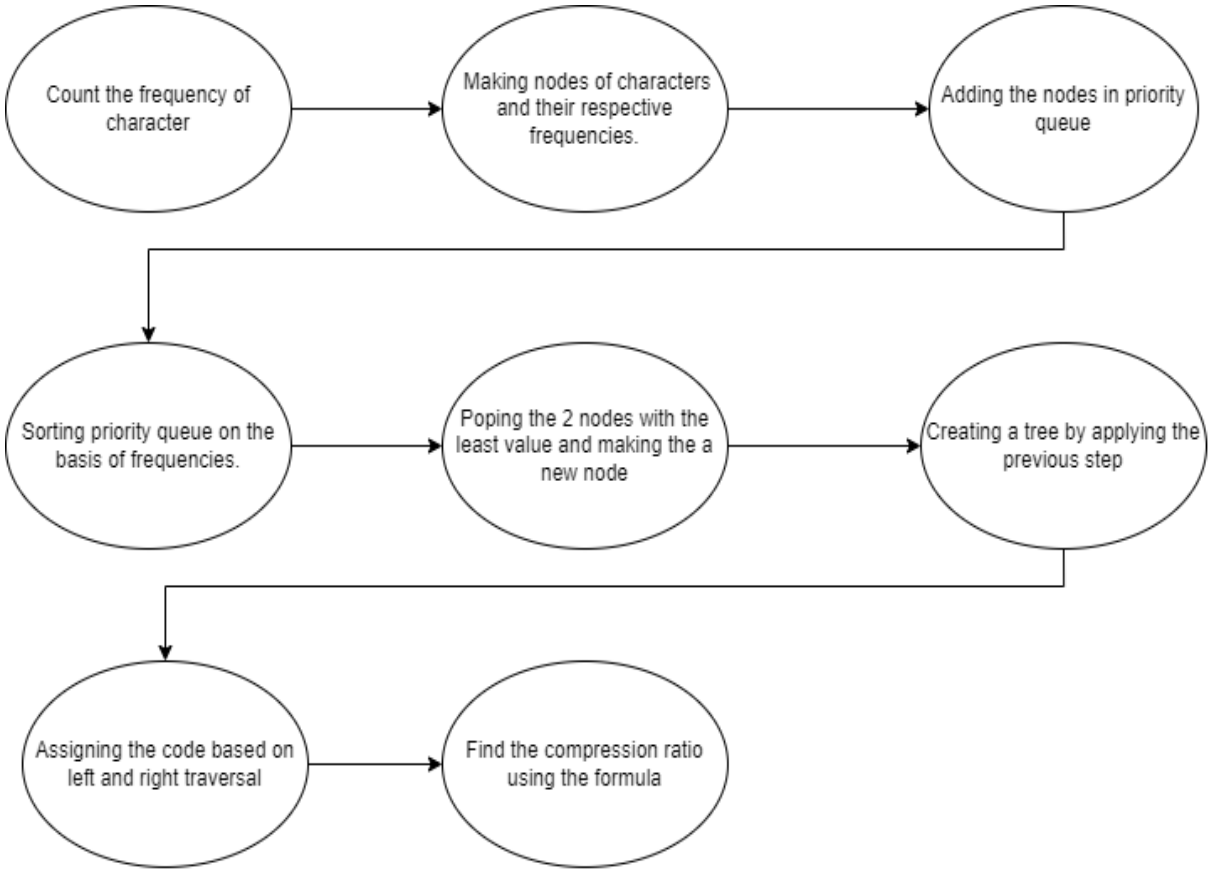
Contents

- The screenshot is showing the main menu of the project.



- Screenshot showing the contents of the string, unique characters, and their respective frequencies. As frequencies are of prime importance in HUFFMAN encoding.





- Basic structure/flow of the implemented code.

TASK 01: Huffman Tree Implementation (without Priority Queue)

```
cout << "\n\t\t\t\t\t----------------------TASK 1----------------------------\n";
vector<hnode> v;
for (int i = 0; i < arr.size(); i++) {
    hnode q;
    q.character = arr[i];
    q.frequecy = freq[i];
    v.push_back(q);
}

vector<hnode> s;   //making copy of vector for task 2
s = v;

htree ht1;
sort(v.begin(), v.end(), compareStuff());//sorting the vector

for (int i = 0; i < arr.length() - 1; i++)//making tree of task1 in this function/ passed the vector by reference
    ht1.insert_task_one(v);

const int cv = arr.length();
int* arrr = new int[cv];
int  top = 0;
ht1.printcodes(ht1.root, arrr, top);//printing huffman codes
cout << "\n\t\t\t\t\tRoot  " << ht1.root->frequecy;
```

- Screenshot showing the calling of the function of question 01.

Firstly, vectors were used, and then the vectors were sorted using the operator overloading. The nodes are added, and then the vector is sorted. This way we have the vector which is sorted and then we apply the rest of the steps of Huffman coding.

```
// insertion of question 01
void insert_task_one(vector<hnode>& l)
{
    // first node val
    auto it1 = l.begin();   //extracting first element of vector
    hnode* f1 = new hnode(*it1);
    l.erase(l.begin());//deleteing it


    // second node
    auto it = l.begin(); //extracting second element of vector
    hnode* f2 = new hnode(*it);
    l.erase(l.begin()); //deleteing it

    // making of the new node
    hnode* temp1 = new hnode;      //saving in temp node to be inserted as a new root
    temp1->frequecy = f1->frequecy + f2->frequecy;
    temp1->right = f1;
    temp1->left = f2;
    cout << temp1->character;
    root = temp1;


    l.push_back(*root);//insering the new root node in vector
    std::sort(l.begin(), l.end(), compareStuff());   //sorting once again
}
```

- Inserting values in the vector which is node-based. First, we pop the 2 nodes which the least frequency and make a new node using them and add them to the vector.

```
int isLeaf(hnode* root) {
    return !(root->left) && !(root->right);
}

// printing of codes
void printcodes(hnode* root, int arr[], int top)
{
    double count = 0;

    // left traversal
    if (root->left)
    {
        arr[top] = 0;
        printcodes(root->left, arr, top + 1);
    }

    // right traversal
    if (root->right) {
        arr[top] = 1;
        printcodes(root->right, arr, top + 1);
    }

    // printing of leaf nodes
    if (isLeaf(root)) {
        cout << "\t\t\t\t\t" << root->character << "  | ";
        for (int i = 0; i < top; ++i) {
            count++;
            cout << arr[i];
        }

        // counting optimal number of bits
        root->comp = count * root->frequecy;
        cout << "\n";
    }
}
```

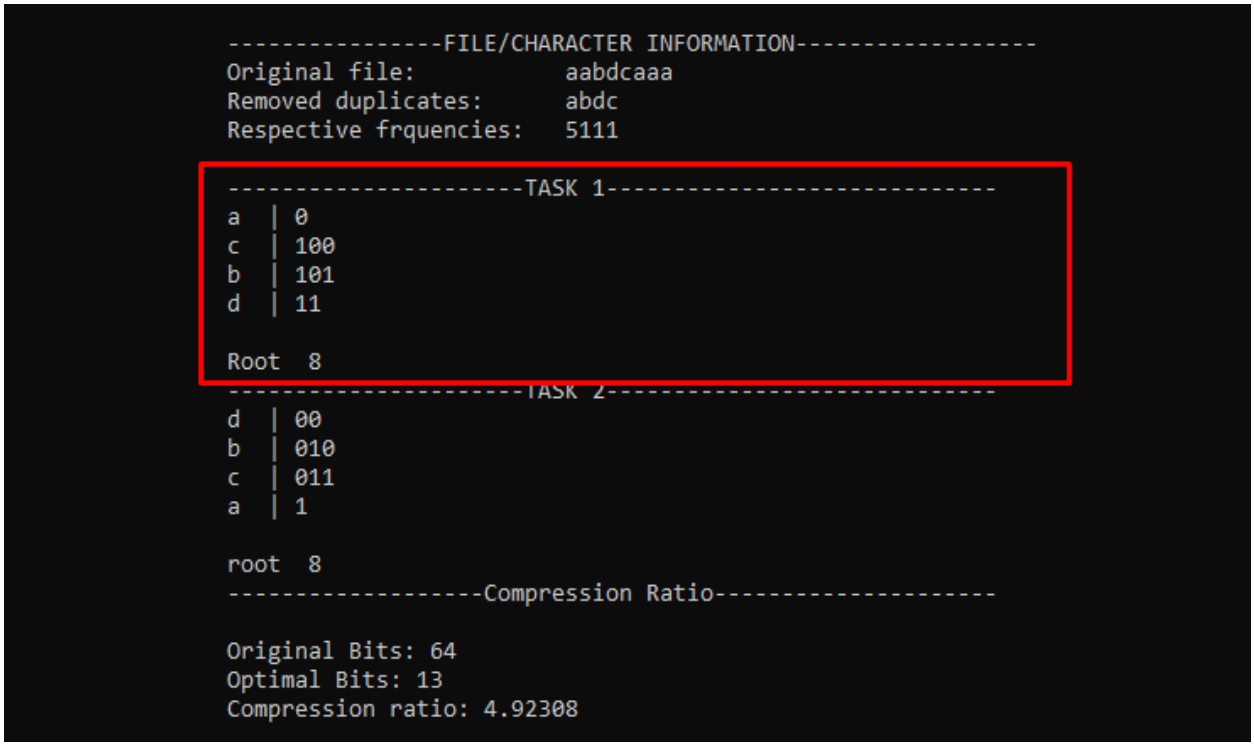
- Screenshot of printing the leaf nodes and assigning codes(bits) to the character nodes.

In this function, we traverse the tree and assign bits based on the left and right positions of each node. Then we are printing the code when we check whether the node is a leaf or not. We are also printing the value of the root node. The last 2 lines of code show the calculation of the optimal number of bits for the compression ratio.

```
// vector overloading
struct compareStuff
{
    bool operator()(const hnode& rhs, const hnode& lhs)//operator overloading for vector sort of type node
    {
        if (lhs.frequecy == rhs.frequecy)
            return lhs.character > rhs.character;
        return lhs.frequecy > rhs.frequecy;
    }
};
```

- Overloading of vector (node type).

As we wanted to sort our based on the frequency, so we did operator overloading and sorted the nodes based on their respective frequencies.



- Screenshot showing the output of question 01 based on character and their respective frequency information.

TASK 02: Huffman Tree Implementation (with Priority Queue)

```
cout << "\n\t\t\t\t\t\t----------TASK 2----------\n";
// using of priority queue
priority_queue<hnode> pq;
for (auto it = s.cbegin(); it != s.cend(); it++)
    pq.push(*it);

htree ht;
ht.makeHuff(&pq);    //making huffman tree for task 2....passing priority queue by reference

arrr = new int[cv];
top = 0;
ht.printcodes(ht.root, arrr, top);//printing huffman codes
cout << "\n\t\t\t\t\troot  " << ht.root->frequecy; // root printing
```

- Screenshot showing the calling of the function of question 02.

Firstly, the object of the inbuilt priority queue was declared and we passed it in the function of **makeHuff** and then we made the tree in that function.

```
void makeHuff(priority_queue<hnode>* p)
{
    int count = 0;
    while (!p->empty())
    {

        // popping of nodes
        if (count == 0)  //for initial node
        {
            hnode* min1 = NULL;
            hnode* min2 = NULL;
            min1 = new hnode(p->top().character, p->top().frequecy);
            p->pop();
            min2 = new hnode(p->top().character, p->top().frequecy);
            p->pop();
            insert(min1, min2, *p);
            count++;
            if (p->size() == 1 || p->size() == 0)
                return;
        }

        // using the popped nodes and making the third node
        else if (!p->empty() && count != 0)// for next nodes
        {
            hnode* h1 = new hnode;
            *h1 = p->top();
            p->pop();
            hnode* h2 = new hnode;
            if (!p->empty())
            {
                *h2 = p->top();
                p->pop();
            }
            insert(h1, h2, *p);
            if (p->size() == 1 || p->size() == 0)
                return;
        }
    }
}
```

Firstly, we make the list of all the characters and their respective frequencies and make a node of each value and then we insert the nodes in the priority queue and then apply sorting by operator overloading.

```
int isLeaf(hnode* root) {
    return !(root->left) && !(root->right);
}

// printing of codes
void printcodes(hnode* root, int arr[], int top)
{
    double count = 0;

    // left traversal
    if (root->left)
    {
        arr[top] = 0;
        printcodes(root->left, arr, top + 1);
    }

    // right traversal
    if (root->right) {
        arr[top] = 1;
        printcodes(root->right, arr, top + 1);
    }

    // printing of leaf nodes
    if (isLeaf(root)) {
        cout << "\t\t\t\t\t" << root->character << "  | ";
        for (int i = 0; i < top; ++i) {
            count++;
            cout << arr[i];
        }

        // counting optimal number of bits
        root->comp = count * root->frequecy;
        cout << "\n";
    }
}
```

- Screenshot of printing the leaf nodes and assigning codes(bits) to the character nodes.

In this function, we traverse the tree and assign bits based on the left and right positions of each node. Then we are printing the code when we check whether the node is a leaf or not. We are also printing the value of the root node. The last 2 lines of code show the calculation of the optimal number of bits of each character for the compression ratio.

```
bool operator <(const hnode& rhs) const  //operator overloading for priority_queue of type hnode
{
    if (frequecy == rhs.frequecy)
        return character > rhs.character;
    return frequecy > rhs.frequecy;
}
```

- Priority queue overloading for sorting the values based on frequencies.

```
----------------FILE/CHARACTER INFORMATION------------------
Original file:          aabdcaaa
Removed duplicates:     abdc
Respective frquencies:  5111

----------------------TASK 1-----------------------------
a  | 0
c  | 100
b  | 101
d  | 11

Root  8
----------------------TASK 2-----------------------------
d  | 00
b  | 010
c  | 011
a  | 1

root  8
-------------------Compression Ratio---------------------

Original Bits: 64
Optimal Bits: 13
Compression ratio: 4.92308
```

- Screenshot showing the output of question 02 based on character and their respective frequency information.

TASK 03: File Compression

```
    // calculation of compression ratio
    cout << "\n\t\t\t\t\t----------Compression Ratio----------\n";
    cout << "\n\t\t\t\t\tOriginal Bits: " << (array.length() * 8);
    float val = ht.showcompression(ht.root);
    cout << "\n\t\t\t\t\tOptimal Bits: " << val;

    // printing the compression ratio
    cout << "\n\t\t\t\t\tCompression ratio: " << (array.length() * 8) / val;
    cout << endl;
}
```

- Calling of function from main for question 03

The array length is the list of all the characters present in the file and by multiplying it by 8 we get the total number of bits that would be initially used without the Huffman encoding.

Then we find the number of bits used after applying the Huffman encoding algorithm using the function **showcompression** by sending the root.

```
// compression function
double showcompression(hnode* root)
{
    static double compresion = 0;

    // sending the compression variable
    if (root == NULL)
        return compresion;

    // recursive traversals
    showcompression(root->left);
    if (isLeaf(root))
        compresion += root->comp;
    showcompression(root->right);
}
```

The compression function adds the values of variable **comp** which is the number of bits of each character multiplied by their respective frequency. The **comp** has the total number of bits used by each character. Then we add all the values and return them.

```
    // counting optimal number of bits
    root->comp = count * root->frequecy;
    cout << "\n";
}
```

- The 2 lines of code show the calculation of the optimal number of bits of each character for the compression ratio.

Calculations:

- **Calculation of original number of bits**

Characters: abcd

Frequency: 5111

Total characters: 5+1+1+1= 8

Bits for each character: 8

Original bits: 8×8 =64 bits

- **Calculation of optimal number of bits**

a => 1 = 2 bits

b =>100 = 3 bits

c =>101 = 3 bits

d =>01 = 2 bits

Optimal bits: 2×5+3×1+3×1+2×1= 13

- **Calculation of optimal number of bits**

Compression ratio: Original bits / Optimal bits

Compression ratio: 64/13 = **4.92**

```
----------------FILE/CHARACTER INFORMATION-----------------
Original file:          aabdcaaa
Removed duplicates:     abdc
Respective frquencies:  5111

----------------------TASK 1----------------------------
a  | 0
c  | 100
b  | 101
d  | 11

Root  8
----------------------TASK 2----------------------------
d  | 00
b  | 010
c  | 011
a  | 1

root  8
------------------Compression Ratio--------------------

Original Bits: 64
Optimal Bits: 13
Compression ratio: 4.92308
```