CS201 Project Presentation

Text Compression Using Huffmann Algorithm

Breakdown of our project

The project is aimed at implementing Huffman encoding and decoding process using Min Heap.

The project is divided in three steps:

- Implementation of Huffman tree using Min Heap, in which all leaf nodes contain the character and its occuring frequency.
- Using the basic chaining techniques in hashtable it will generate two files - compressed version of input file and encoded bit table for each character.
- After encoding, we need to decrypt and decode the data which will take as input the output of step 2 and generate decoded.txt.

Basic Logic Behind our Code

1)Implementation of Huffman Encoding algorithm

Firstly we need to take the input from the user that contains the data (text file) which we have to compress using this algorithm which contains basically two steps :

- First we have to find the frequency of each character.
- Creating the Huffman Tree.

Basic Logic Behind our Code

2)Implementation of Huffman tree

- Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, The least frequent character is at root).
- Extract two nodes with the minimum frequency from the min heap.
- 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.
- Repeat steps 2.b and 2.c until the heap contains only one node. The remaining node is the root node and the tree is complete.

Basic Logic Behind our Code

3) Printing the Codes from Huffman Tree

- Traverse the tree formed starting from the root. Maintain an auxiliary array.
- While moving to the left child, write 0 to the array.
- While moving to the right child, write 1 to the array.
- Print the array when a leaf node is encountered. As we have to print encoded code in output file we maintain an array of pointers pointing to the head of linked list in which we store data of array in linked list

Basic Logic Behind our Code

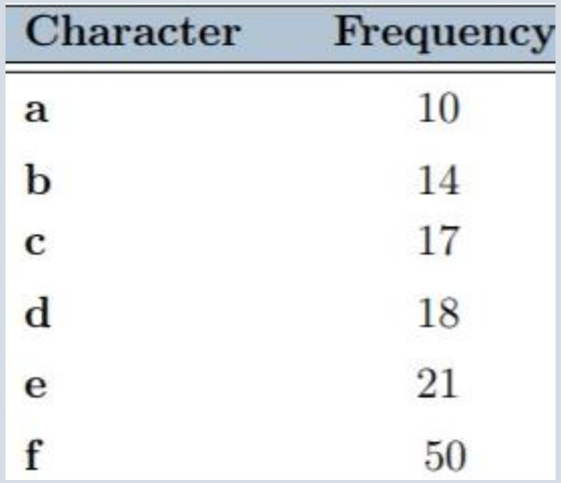
4) Implementation of Huffman Decoding algorithm

To decode the encoded data we require the Huffman tree. We iterate through the binary encoded data. To find character corresponding to current bits, we use following simple steps.

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

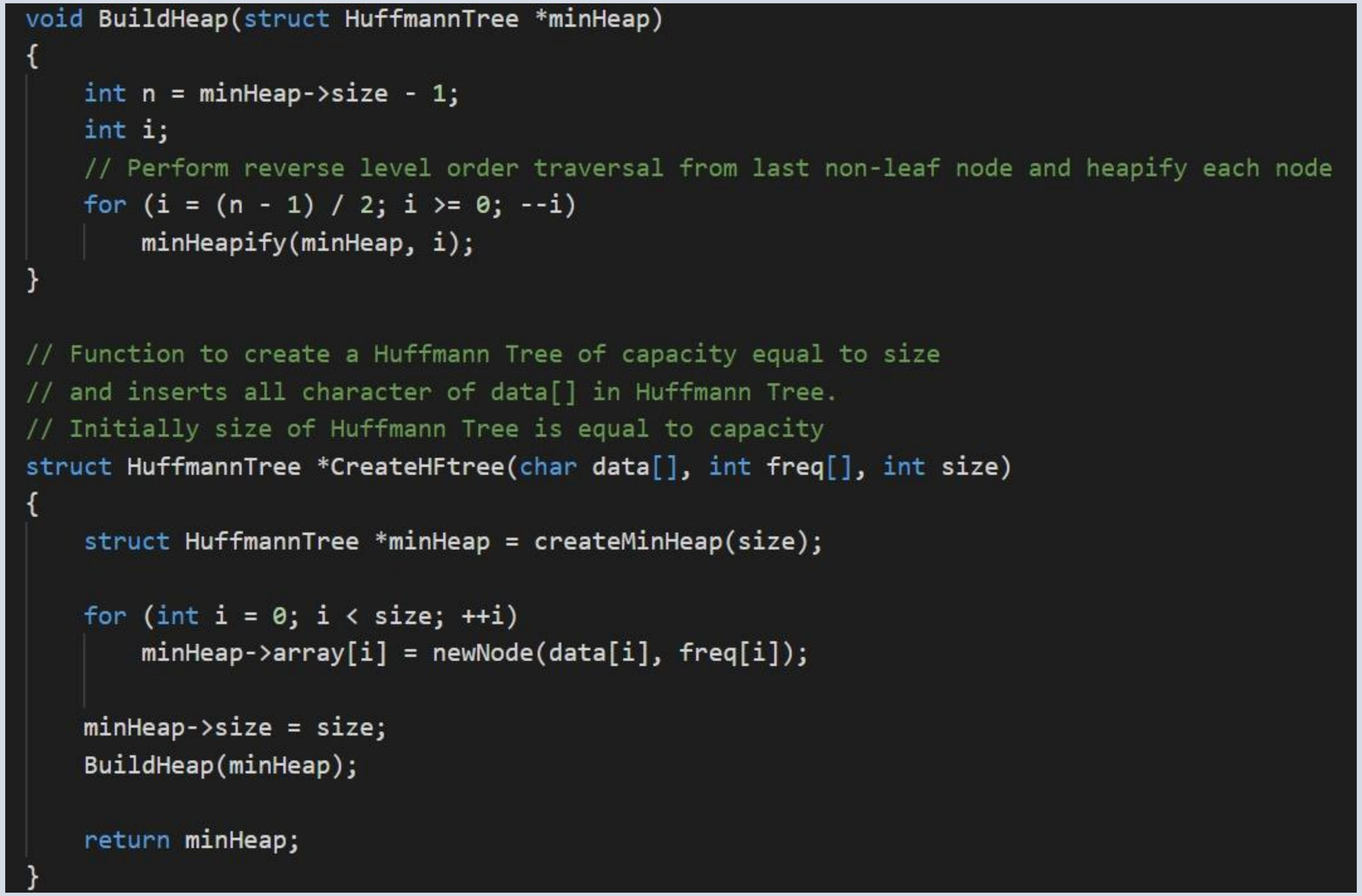
Working Example

- Let Input file contain following character with their respective frequency



Character	Frequency
a	10
b	14
c	17
d	18
e	21
f	50

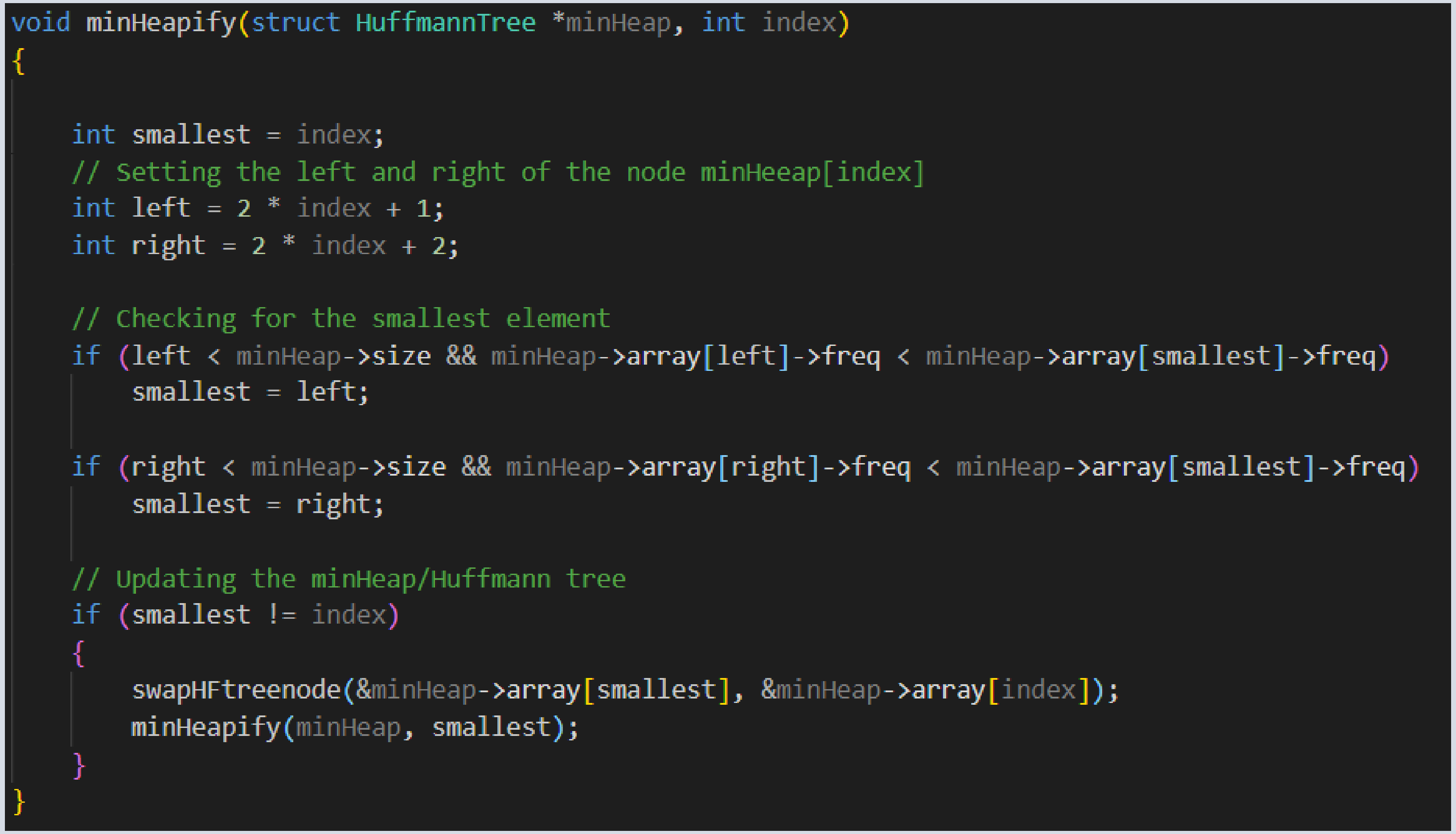
- Build a min heap that contains 6 nodes, where each node represents root of a tree with single node



```c
void BuildHeap(struct HuffmannTree *minHeap)
{
    int n = minHeap->size - 1;
    int i;
    // Perform reverse level order traversal from last non-leaf node and heapify each node
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// Function to create a Huffmann Tree of capacity equal to size
// and inserts all character of data[] in Huffmann Tree.
// Initially size of Huffmann Tree is equal to capacity
struct HuffmannTree *CreateHFtree(char data[], int freq[], int size)
{
    struct HuffmannTree *minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);

    minHeap->size = size;
    BuildHeap(minHeap);

    return minHeap;
}
```

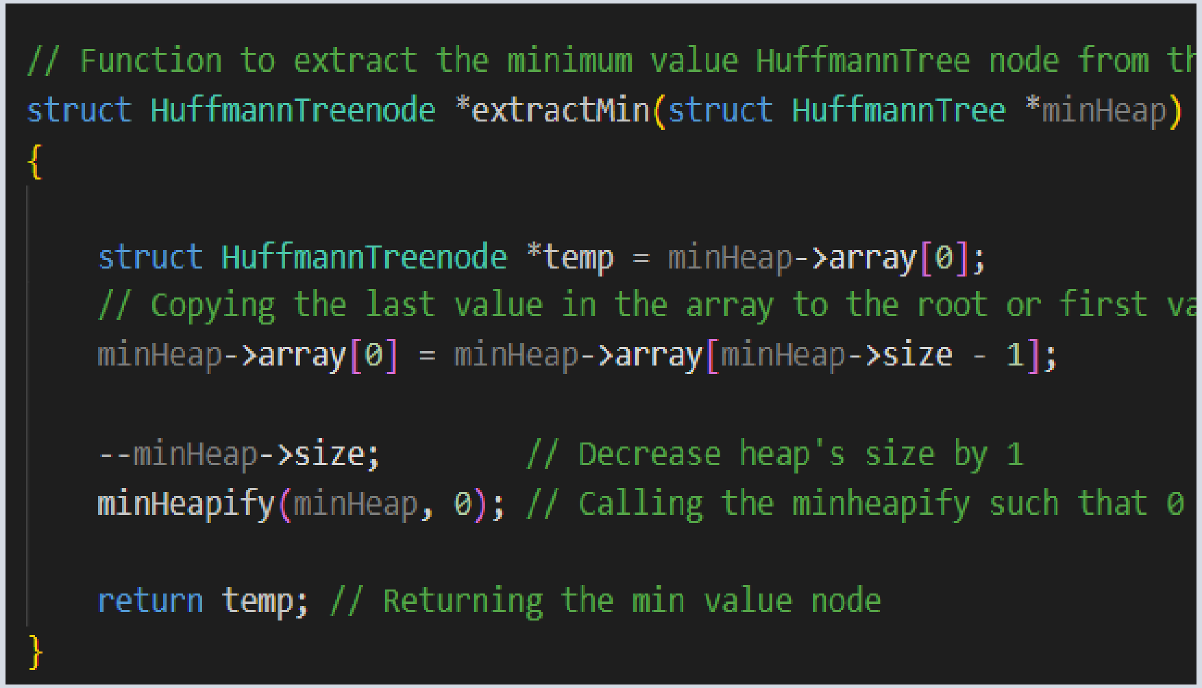
- Minheapify Function-

This Function makes sure that minheap property is satisfied in the array using indexing property.

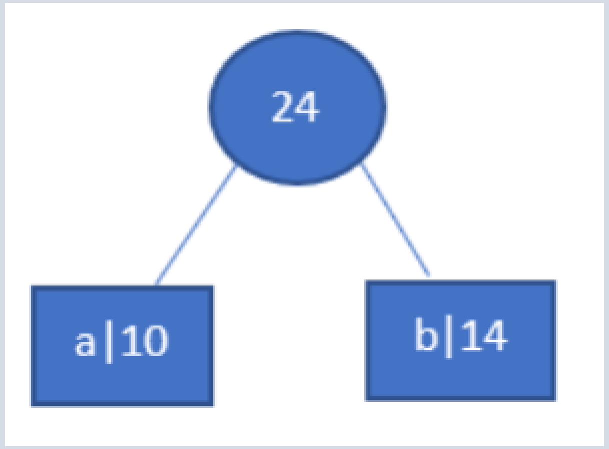


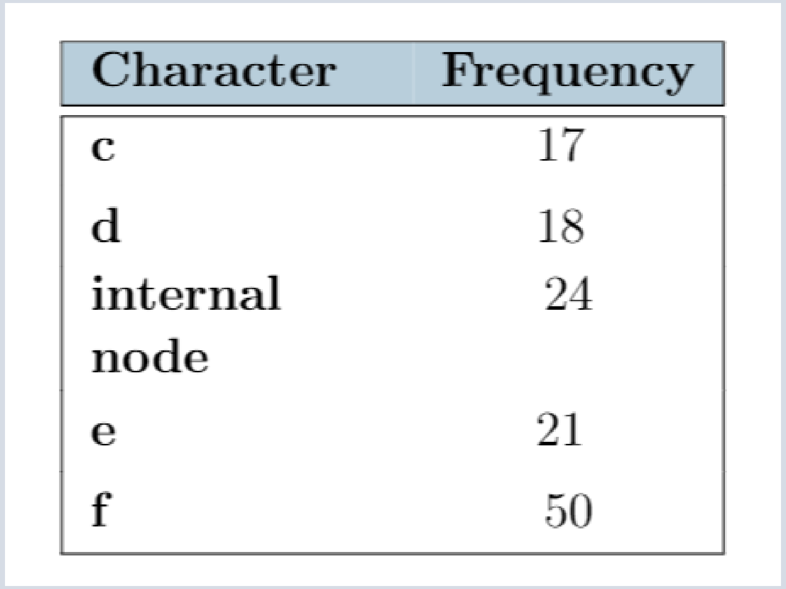
```c
void minHeapify(struct HuffmannTree *minHeap, int index)
{

    int smallest = index;
    // Setting the left and right of the node minHeeap[index]
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    // Checking for the smallest element
    if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
        smallest = right;

    // Updating the minHeap/Huffmann tree
    if (smallest != index)
    {
        swapHFtreenode(&minHeap->array[smallest], &minHeap->array[index]);
        minHeapify(minHeap, smallest);
    }
}
```

- Extract two minimum frequency nodes from min heap. Add new internal node with frequency .

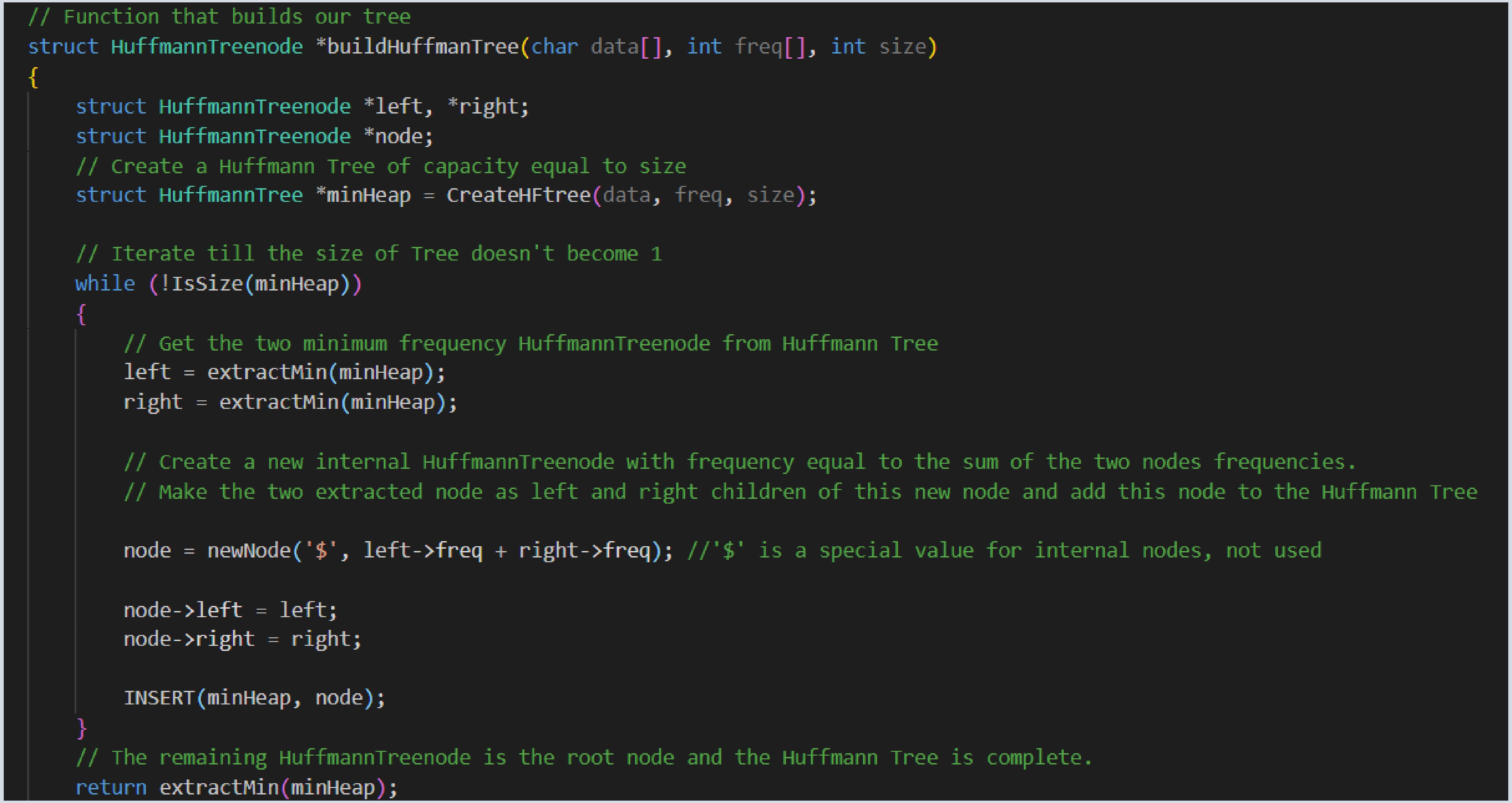


```
// Function to extract the minimum value HuffmannTree node from th
struct HuffmannTreenode *extractMin(struct HuffmannTree *minHeap)
{

    struct HuffmannTreenode *temp = minHeap->array[0];
    // Copying the last value in the array to the root or first va
    minHeap->array[0] = minHeap->array[minHeap->size - 1];

    --minHeap->size;        // Decrease heap's size by 1
    minHeapify(minHeap, 0); // Calling the minheapify such that 0

    return temp; // Returning the min value node
}
```



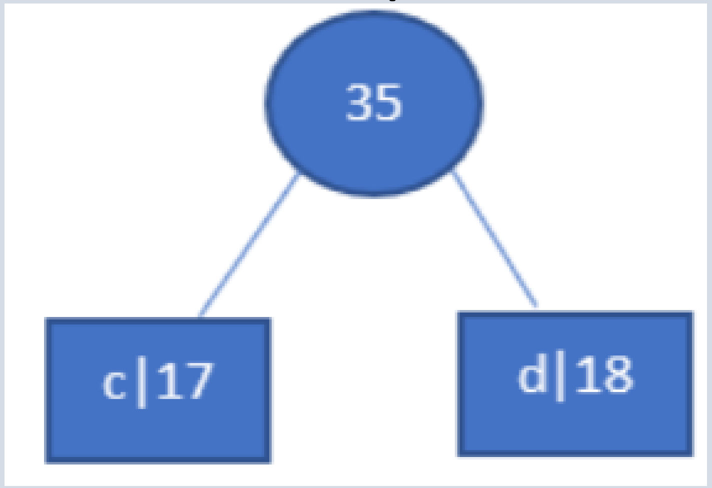


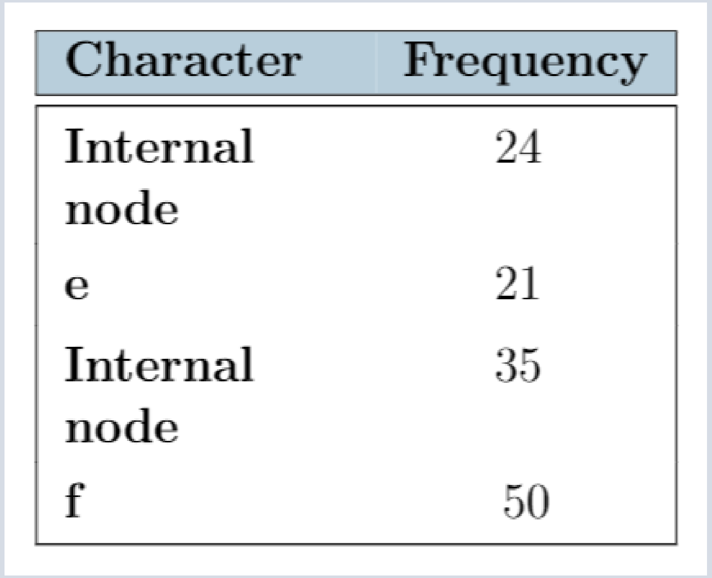
Character	Frequency
c	17
d	18
internal node	24
e	21
f	50



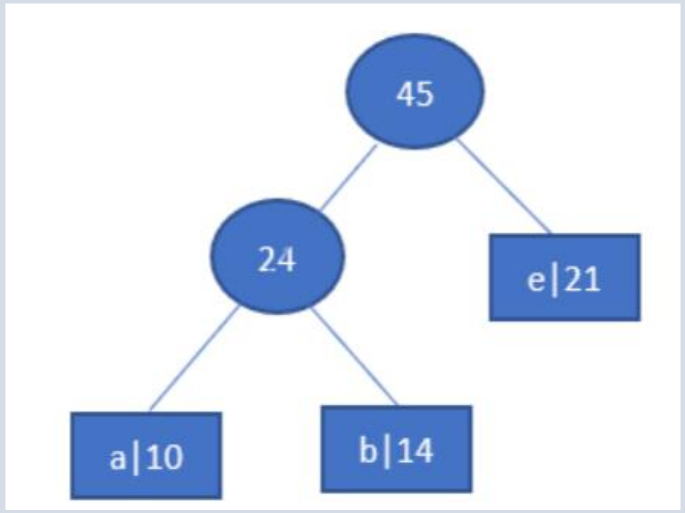
```c
// Function that builds our tree
struct HuffmannTreenode *buildHuffmanTree(char data[], int freq[], int size)
{
    struct HuffmannTreenode *left, *right;
    struct HuffmannTreenode *node;
    // Create a Huffmann Tree of capacity equal to size
    struct HuffmannTree *minHeap = CreateHFtree(data, freq, size);

    // Iterate till the size of Tree doesn't become 1
    while (!IsSize(minHeap))
    {
        // Get the two minimum frequency HuffmannTreenode from Huffmann Tree
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Create a new internal HuffmannTreenode with frequency equal to the sum of the two nodes frequencies.
        // Make the two extracted node as left and right children of this new node and add this node to the Huffmann Tree

        node = newNode('$', left->freq + right->freq); //'$' is a special value for internal nodes, not used

        node->left = left;
        node->right = right;

        INSERT(minHeap, node);
    }
    // The remaining HuffmannTreenode is the root node and the Huffmann Tree is complete.
    return extractMin(minHeap);
```

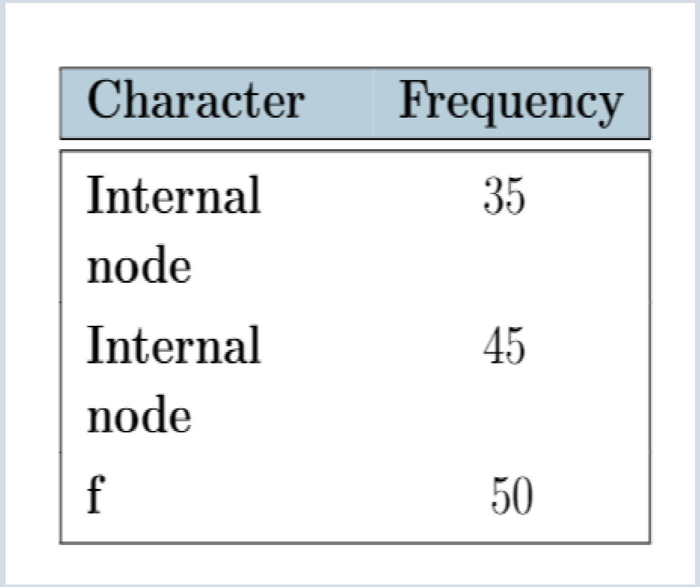
- Repeating the above steps till minheap has size 1



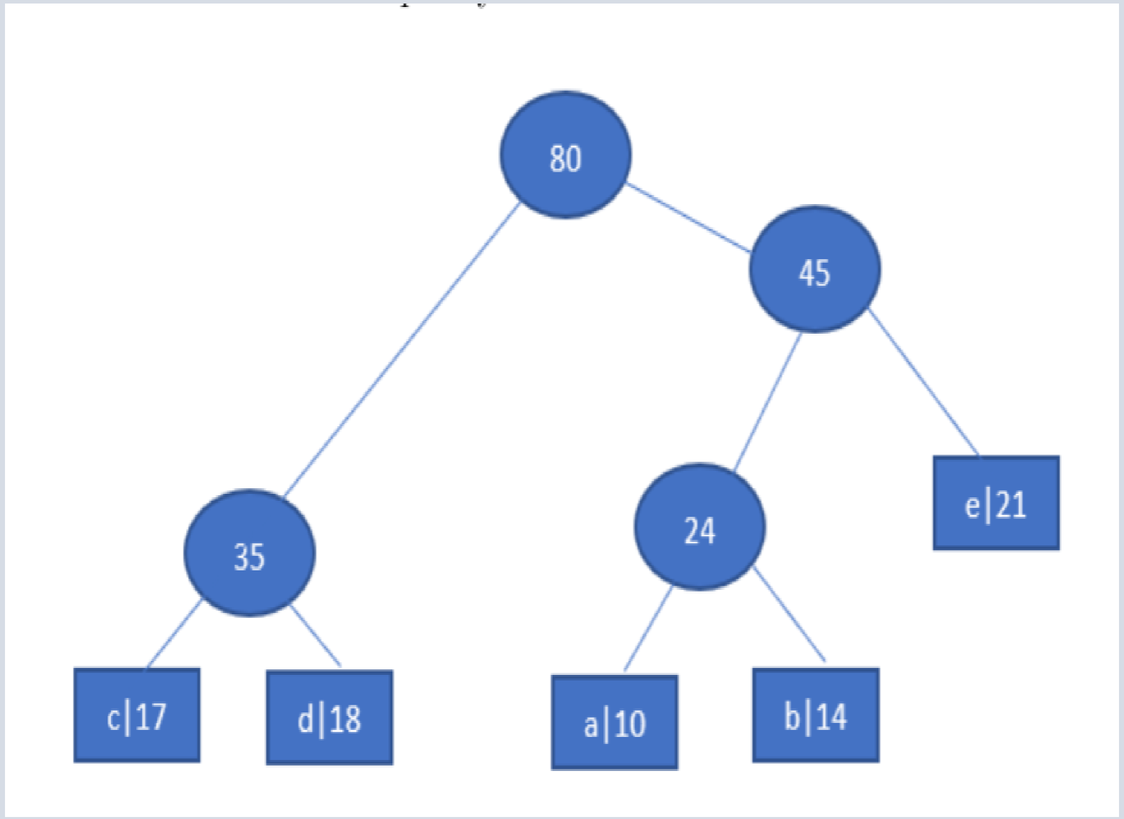


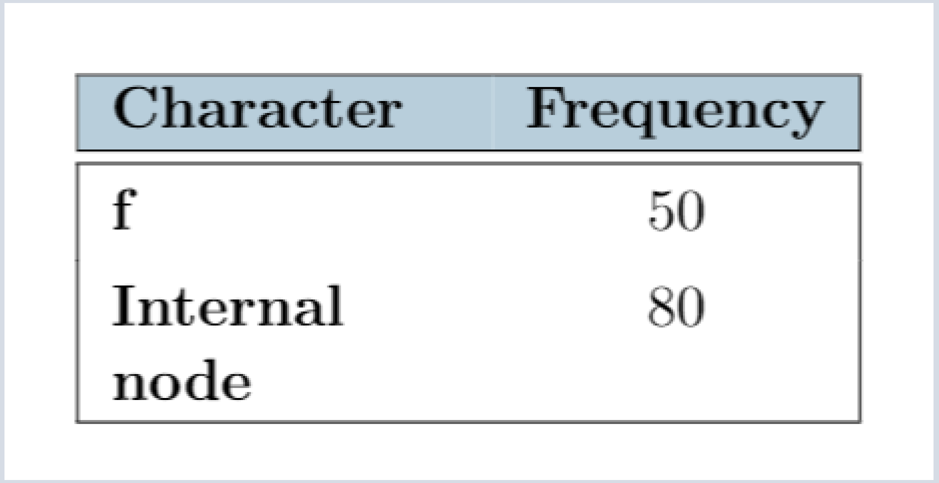
Character	Frequency
Internal node	24
e	21
Internal node	35
f	50



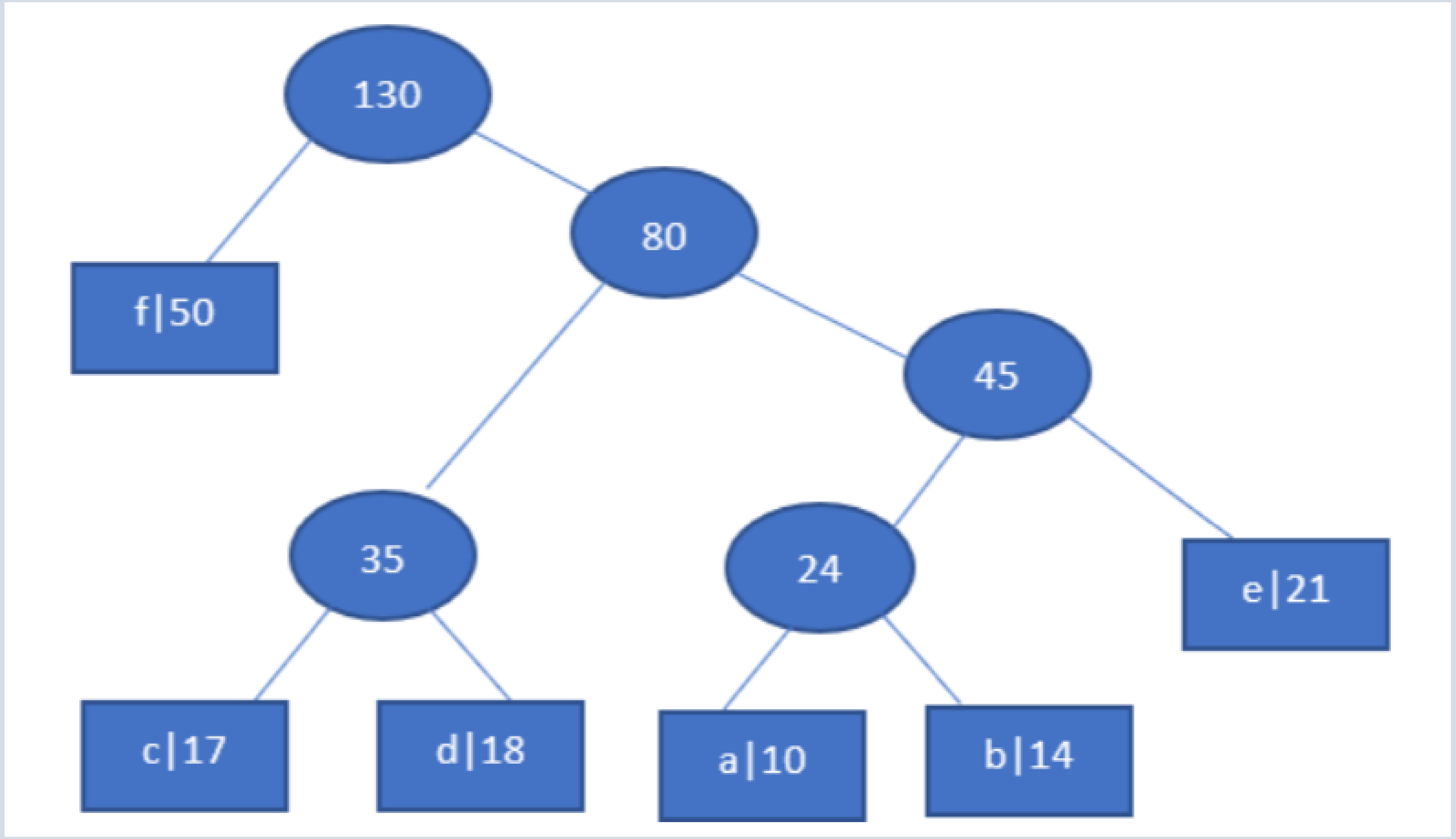


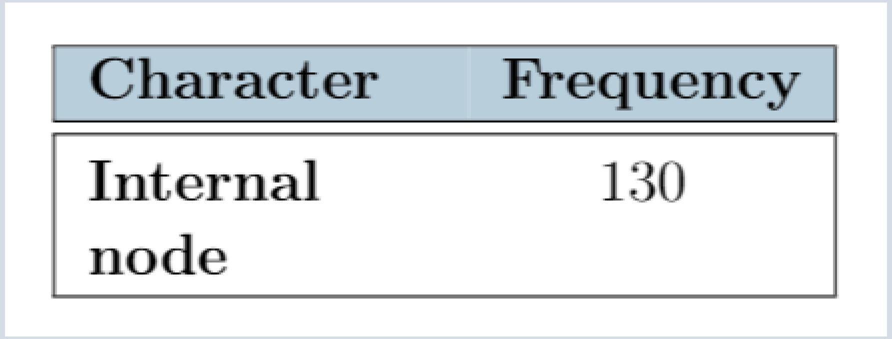
Character	Frequency
Internal node	35
Internal node	45
f	50





Character	Frequency
f	50
Internal node	80

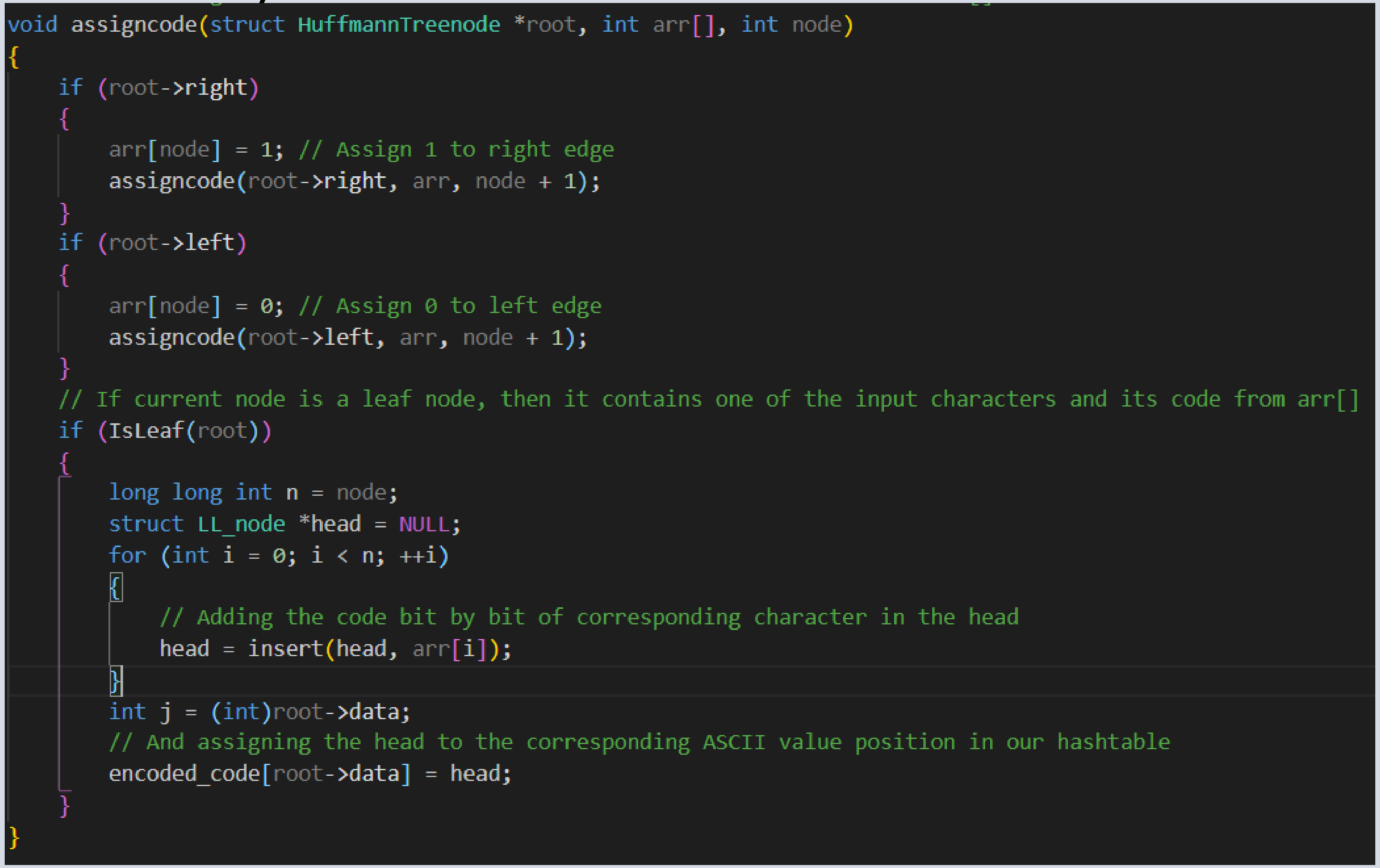




Character	Frequency
Internal node	130

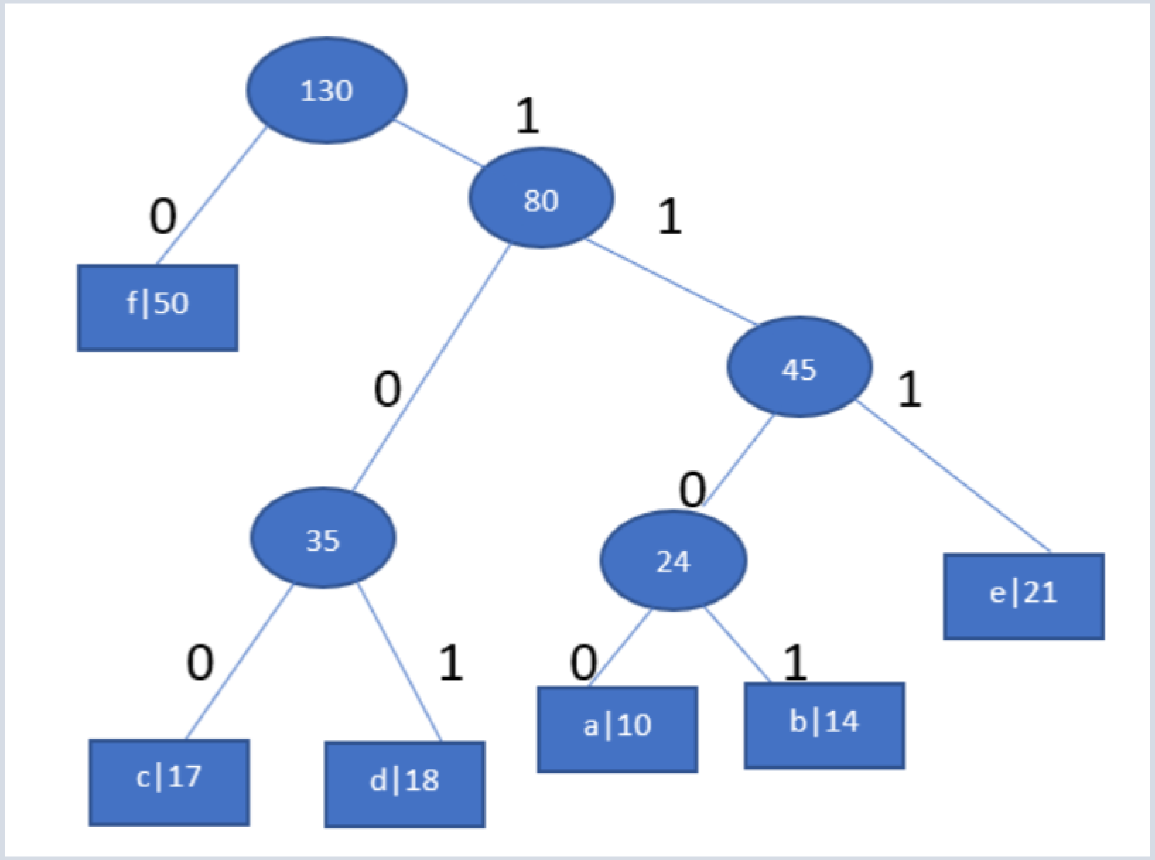
Since the heap contains only one node, the algorithm stops here.

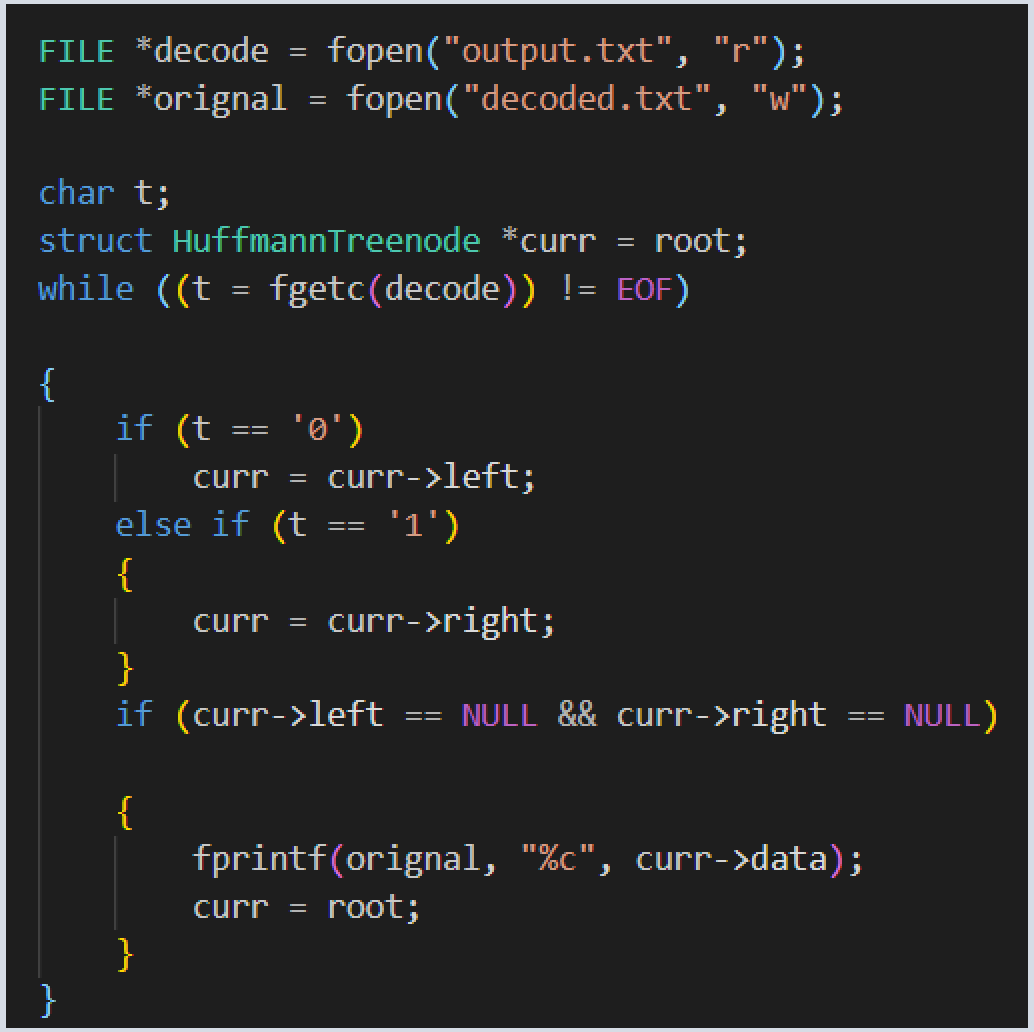
- Assigning encoded codes to character according to their corresponding frequency using hash table(Chaining with use of singly linked list)



```
void assigncode(struct HuffmannTreenode *root, int arr[], int node)
{
    if (root->right)
    {
        arr[node] = 1; // Assign 1 to right edge
        assigncode(root->right, arr, node + 1);
    }
    if (root->left)
    {
        arr[node] = 0; // Assign 0 to left edge
        assigncode(root->left, arr, node + 1);
    }
    // If current node is a leaf node, then it contains one of the input characters and its code from arr[]
    if (IsLeaf(root))
    {
        long long int n = node;
        struct LL_node *head = NULL;
        for (int i = 0; i < n; ++i)
        {
            // Adding the code bit by bit of corresponding character in the head
            head = insert(head, arr[i]);
        }
        int j = (int)root->data;
        // And assigning the head to the corresponding ASCII value position in our hashtable
        encoded_code[root->data] = head;
    }
}
```

Decoding the encoded file by traversing the Huffman tree





```
FILE *decode = fopen("output.txt", "r");
FILE *orignal = fopen("decoded.txt", "w");

char t;
struct HuffmannTreenode *curr = root;
while ((t = fgetc(decode)) != EOF)

{
    if (t == '0')
        curr = curr->left;
    else if (t == '1')
    {
        curr = curr->right;
    }
    if (curr->left == NULL && curr->right == NULL)

    {
        fprintf(orignal, "%c", curr->data);
        curr = root;
    }
}
```

How to run the Code ?

NOTE: For reference here is the link of demo run of our code made by one of our team member:
https://youtu.be/0_fCZSjUAVY

- **Program will first ask you to enter name of file which you want to compress:**
  For Example one can enter "t.txt"

- **Then Program reads file character by charcter**
  Program reads file character by character and store the count of each character

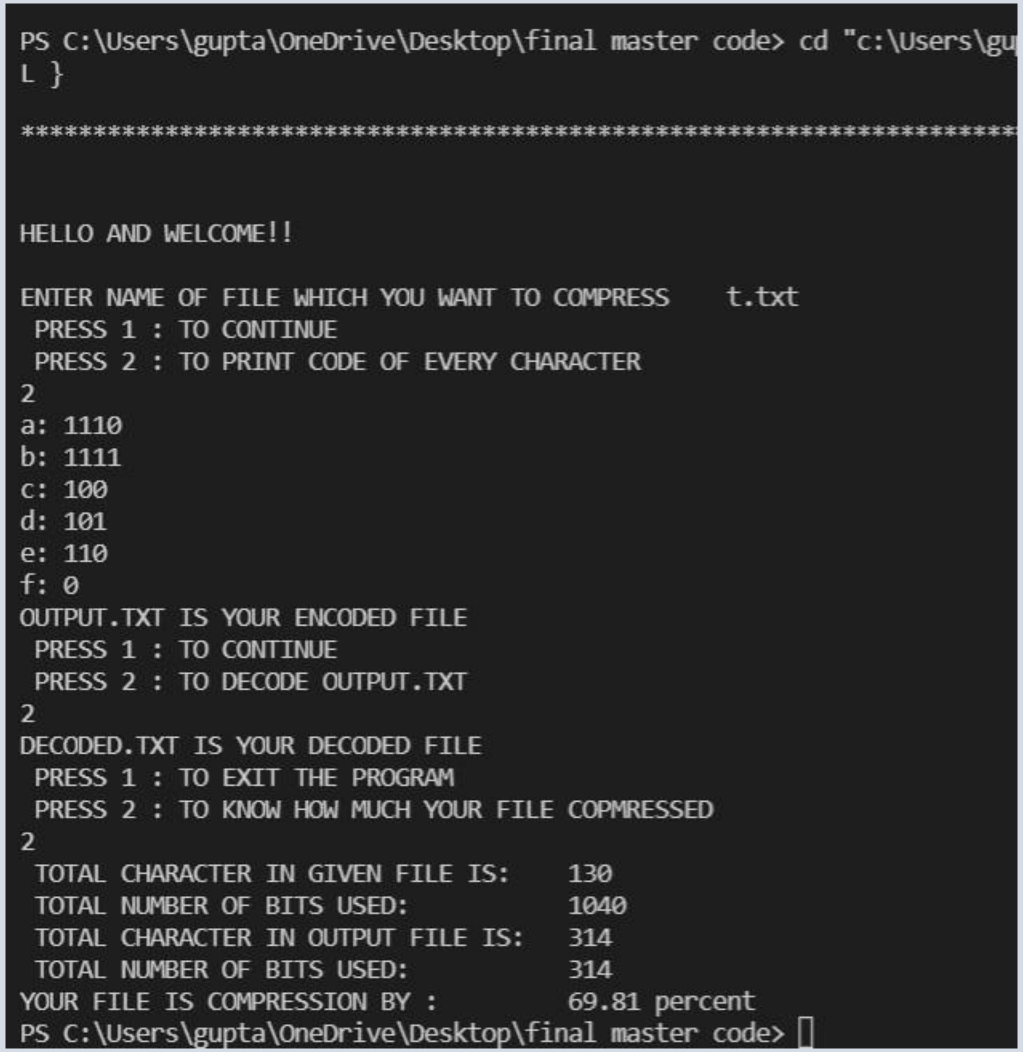
- **To print encoded code of each character press 2 else press 1 to conitnue**
  Program stores the encoded bits using hashing(chaining using linked lists)

How to run the Code ?

- **OUTPUT.TXT will be generated.**
  Output.txt will includes the encoded code of input text file
- **To decrypt encoded file press 2**
  This will decrypt the encoded file i.e output.txt
- **DECODED.TXT will be generated**
- **Press 2 to get summary about compression**
  This will provide the compression percentage, reduction of number of bits etc
- **Program will tell you by how much percentage your file get compressed.**

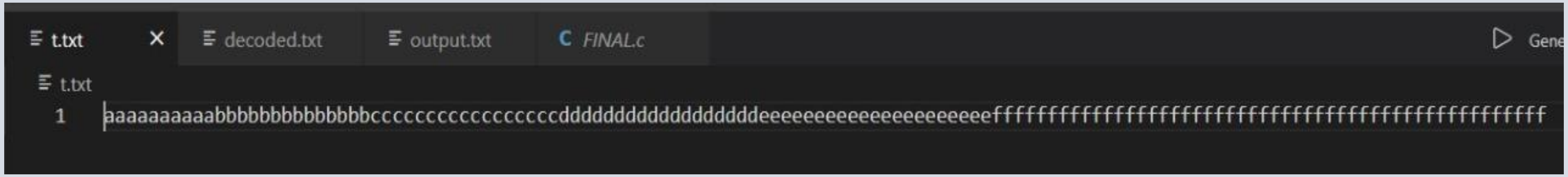
NOTE: The git repo is also attached with some random text files for reference such as "a.txt" ,"b.txt" , "t.txt".

- User interface for this particular example :



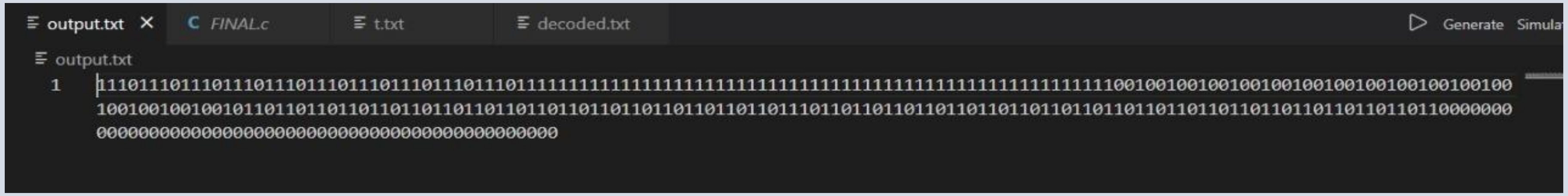
```
PS C:\Users\gupta\OneDrive\Desktop\final master code> cd "c:\Users\gu
L }

**********************************************************************


HELLO AND WELCOME!!

ENTER NAME OF FILE WHICH YOU WANT TO COMPRESS    t.txt
 PRESS 1 : TO CONTINUE
 PRESS 2 : TO PRINT CODE OF EVERY CHARACTER
2
a: 1110
b: 1111
c: 100
d: 101
e: 110
f: 0
OUTPUT.TXT IS YOUR ENCODED FILE
 PRESS 1 : TO CONTINUE
 PRESS 2 : TO DECODE OUTPUT.TXT
2
DECODED.TXT IS YOUR DECODED FILE
 PRESS 1 : TO EXIT THE PROGRAM
 PRESS 2 : TO KNOW HOW MUCH YOUR FILE COPMRESSED
2
 TOTAL CHARACTER IN GIVEN FILE IS:    130
 TOTAL NUMBER OF BITS USED:           1040
 TOTAL CHARACTER IN OUTPUT FILE IS:   314
 TOTAL NUMBER OF BITS USED:           314
YOUR FILE IS COMPRESSION BY :         69.81 percent
PS C:\Users\gupta\OneDrive\Desktop\final master code>
```

- Input/t.txt



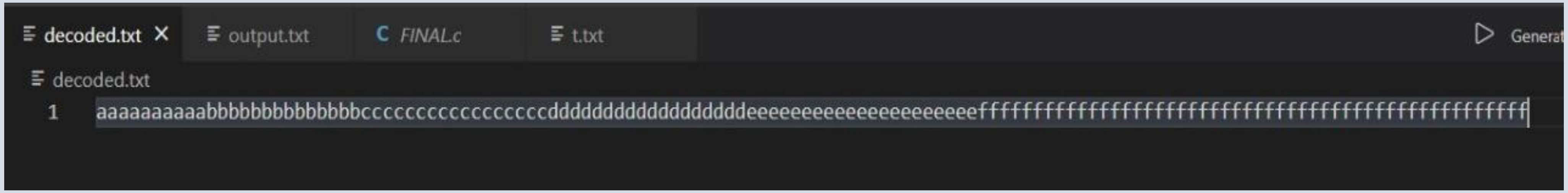
```
aaaaaaaaaabbbbbbbbbbbbbbccccccccccccccccddddddddddddddddddeeeeeeeeeeeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

- Output.txt



```
111011101110111011101110111011101110111011111111111111111111111111111111111111111111111111111111100100100100100100100100100100100100
100100100100101101101101101101101101101101101101101101101101101101110110110110110110110110110110110110110110110110110110110110000000
0000000000000000000000000000000000000000000000000000
```

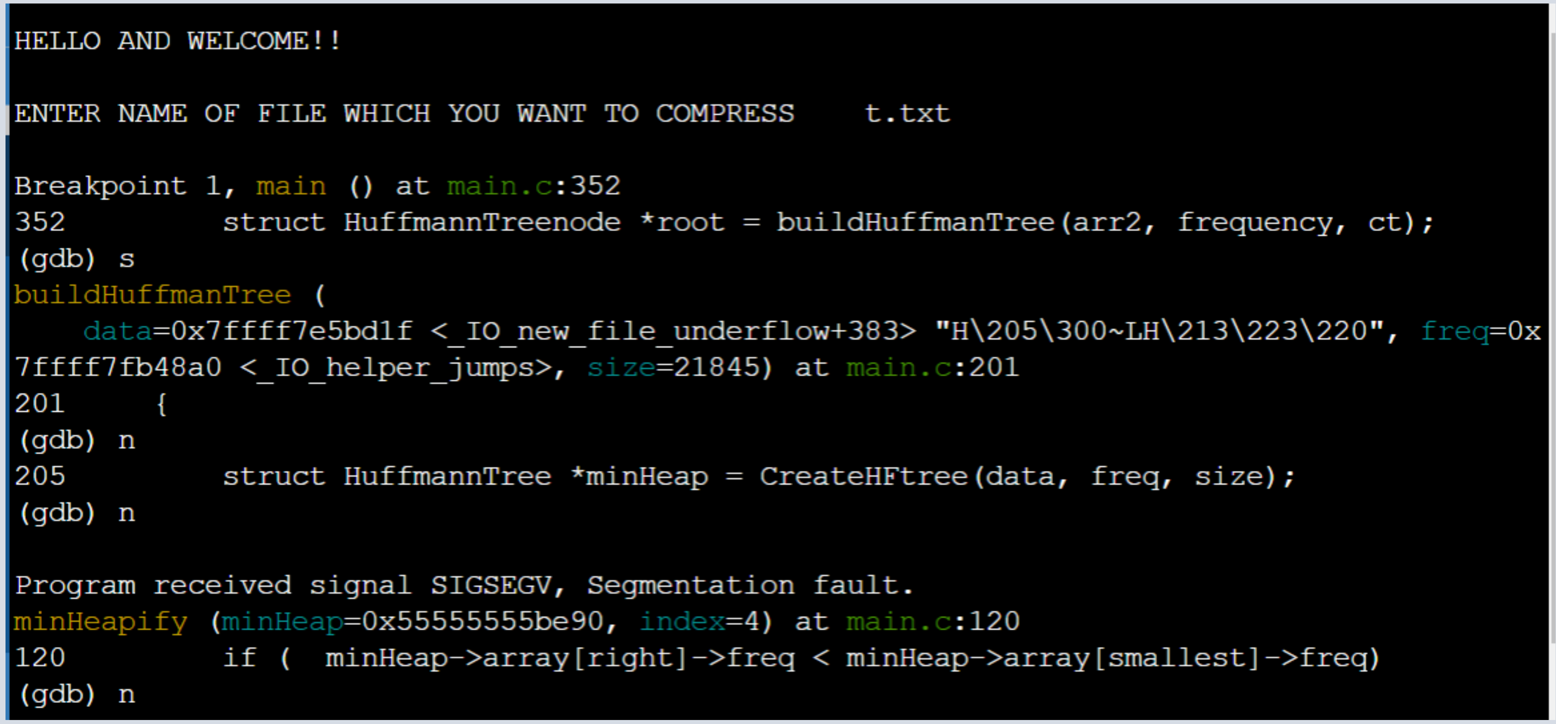
- Decoded.txt



```
aaaaaaaaaabbbbbbbbbbbbbbccccccccccccccccddddddddddddddddddeeeeeeeeeeeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

Usage of Debugger

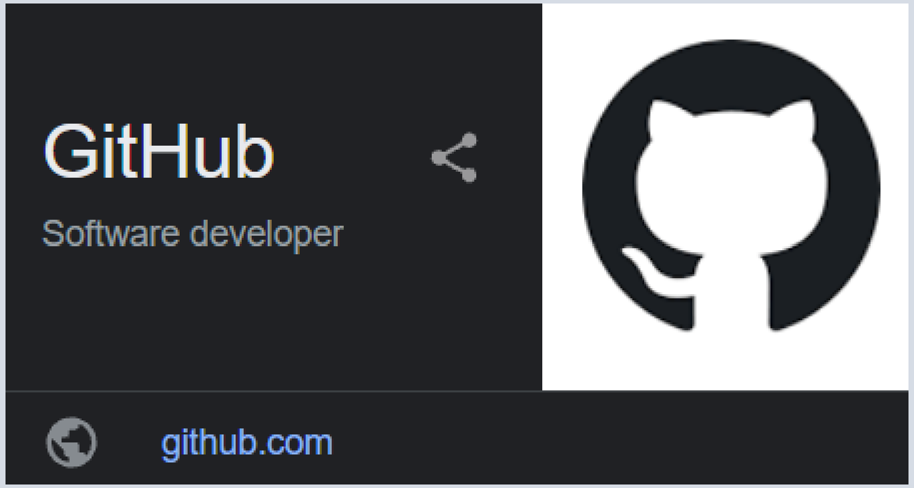
- While running the code , we get an error(segmentation fault) in heapify function at line 120

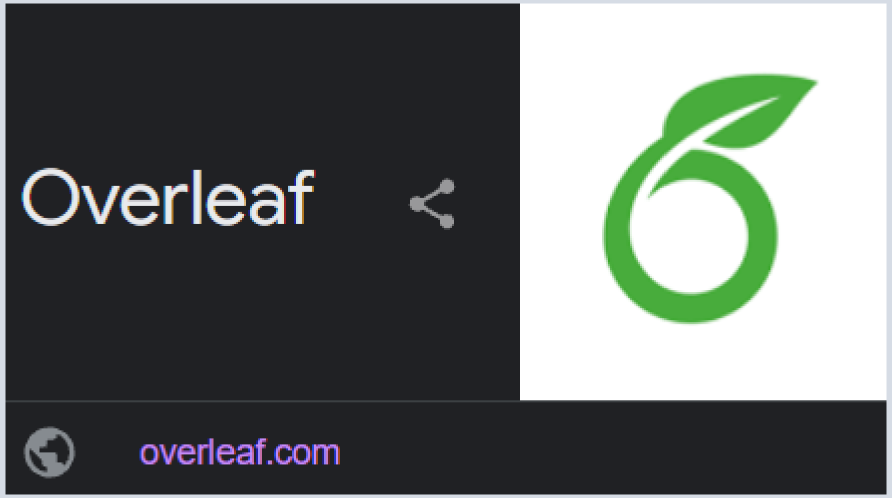


```
HELLO AND WELCOME!!

ENTER NAME OF FILE WHICH YOU WANT TO COMPRESS    t.txt

Breakpoint 1, main () at main.c:352
352          struct HuffmannTreenode *root = buildHuffmanTree(arr2, frequency, ct);
(gdb) s
buildHuffmanTree (
    data=0x7ffff7e5bd1f <_IO_new_file_underflow+383> "H\205\300~LH\213\223\220", freq=0x
7ffff7fb48a0 <_IO_helper_jumps>, size=21845) at main.c:201
201      {
(gdb) n
205          struct HuffmannTree *minHeap = CreateHFtree(data, freq, size);
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
minHeapify (minHeap=0x55555555be90, index=4) at main.c:120
120          if (  minHeap->array[right]->freq < minHeap->array[smallest]->freq)
(gdb) n
```

Usage of Github and Overleaf

- We used Github and overleaf latex to collaborate among ourselves.
- It helped us to work on the same program remotely and efficiently.





Why Data Compression is Important and How Huffmann algorithm allows us to compress the data !

- If we talk about today research says that trillions of data is producing every day. If we have to use this data for future we must store it somewhere that is the reason we have to build some data structures which can compress the size of data without loosing it. Data compression can dramatically decrease the amount of storage a file takes .
- As a result of compression, administrators spend less money and less time on storage. Compression optimizes backup storage performance and has recently shown up in primary storage data reduction

Why Data Compression is Important and How Huffmann algorithm allows us to compress the data !

- Huffman encoding is an algorithm used to compress the size of data without loosing it by assigning different code to each characters using Huffman tree . It is a statistical compression method that converts characters into variable length bit strings and produces a prefix code.
- The least frequent character gets the longest code and the character which is most frequent gets the smallest .The code is assign to character in such a way code assigned to one character is not the prefix of code assigned to any other character. This is how Huffmann Coding makes sure that there is no ambiguity when decoding the generated bits stream.

More Things that can be added on !

- To make it a Fully Furnished Text Compressor , We can also add ASCII codes of the symbols of different languages such as:

1. Greek symbols : α,β,γ,ε etc.
2. IPA extension symbols : ɯ,ɸ,æ etc.
3. Currency symbols : ₹,€,₳ etc.

More Things that can be added on !

- Based on the same algorithm , We can design an Image compressor too .The basic difference is only that

1. In Text compression we are taking input file of characters and generating codes on the basis of ASCII values of characters.
2. Whereas in Image compression we will take input image as an ordered histogram where we can treat each pixel intensity value as a symbol and treat the whole image as an 2D array ,from where we can convert them to variable codes based on their frequency values.

Contribution of Team Members

- Jatin:

File handling of the input and output files , Constructing the Huffmann Tree, Report on the overleaf

- Abhay Shukralia:

Report on the overleaf , Constructing the Huffmann Tree,Decoding the encoded file

- Tanuj Kumar:

Construction of Hashtable to store the encoded bits , Constructing the Huffmann Tree, Report on the overleaf

Thank You

Group 26 Members:

- Abhay Shukralia 2020csb1061
- Tanuj Kumar 2020csb1134
- Jatin 2020csb1090