

LABORATORY REPORT

**Algorithm Laboratory (CS-39001)**

**B.Tech Program in ECS**

Submitted By

**Name:- SANNIDHI DEB**

**Roll No: 2330044**



**Kalinga Institute of Industrial Technology  
(Deemed to be University) Bhubaneswar, India**

Autumn, 2025

## Table of Contents

| <b>Experiment Number</b>  | 6.1   |          |    |    |   |   |   |   |                     |    |    |    |    |   |   |          |   |   |   |   |   |   |          |  |  |  |  |  |  |         |  |  |  |  |  |  |
|---------------------------|---|----------|----|----|---|---|---|---|---------------------|----|----|----|----|---|---|----------|---|---|---|---|---|---|----------|--|--|--|--|--|--|---------|--|--|--|--|--|--|
| <b>Experiment Title</b>   | <p>Huffman coding assigns variable length code words to fixed length input characters based on their frequencies or probabilities of occurrence.</p> <p>Given a set of characters along with their frequency of occurrences, write a C program to construct a Huffman tree.</p> <p>Given:</p> <table border="1"> <thead> <tr> <th>Alphabet</th><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th><th>f</th></tr> </thead> <tbody> <tr> <td>Frequency (in 1000)</td><td>45</td><td>13</td><td>12</td><td>16</td><td>9</td><td>5</td></tr> </tbody> </table> <p>Output (to be shown on the console):</p> <table border="1"> <thead> <tr> <th>Alphabet</th><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th><th>f</th></tr> </thead> <tbody> <tr> <td>Codeword</td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr> <td>BitSize</td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </tbody> </table> <p>Based on the frequency given, print the total number of bits needed to represent a file with 100,000 characters.</p> | Alphabet | a  | b  | c | d | e | f | Frequency (in 1000) | 45 | 13 | 12 | 16 | 9 | 5 | Alphabet | a | b | c | d | e | f | Codeword |  |  |  |  |  |  | BitSize |  |  |  |  |  |  |
| Alphabet                  | a   | b        | c  | d  | e | f |   |   |                     |    |    |    |    |   |   |          |   |   |   |   |   |   |          |  |  |  |  |  |  |         |  |  |  |  |  |  |
| Frequency (in 1000)       | 45  | 13       | 12 | 16 | 9 | 5 |   |   |                     |    |    |    |    |   |   |          |   |   |   |   |   |   |          |  |  |  |  |  |  |         |  |  |  |  |  |  |
| Alphabet                  | a   | b        | c  | d  | e | f |   |   |                     |    |    |    |    |   |   |          |   |   |   |   |   |   |          |  |  |  |  |  |  |         |  |  |  |  |  |  |
| Codeword                  |   |          |    |    |   |   |   |   |                     |    |    |    |    |   |   |          |   |   |   |   |   |   |          |  |  |  |  |  |  |         |  |  |  |  |  |  |
| BitSize                   |   |          |    |    |   |   |   |   |                     |    |    |    |    |   |   |          |   |   |   |   |   |   |          |  |  |  |  |  |  |         |  |  |  |  |  |  |
| <b>Date of Experiment</b> | 01/10/2025  |          |    |    |   |   |   |   |                     |    |    |    |    |   |   |          |   |   |   |   |   |   |          |  |  |  |  |  |  |         |  |  |  |  |  |  |
| <b>Date of Submission</b> | 10/10/2025  |          |    |    |   |   |   |   |                     |    |    |    |    |   |   |          |   |   |   |   |   |   |          |  |  |  |  |  |  |         |  |  |  |  |  |  |

## 1. Algorithm:-

START

samidhi Deb, 2330044

1. Create leaf node for each symbol and add priority queue (min-heap)
2. While there is more than one node in queue:  
    Remove two nodes with smallest freq.  
    Create new internal node with freq = sum of two nodes.  
    Set two nodes as left and right children  
    Insert new node back in queue.
3. Remaining node is root of Tree.
4. Traverse and assign "0" to left and "1" to right edge.
5. Record codeword for each leaf.
6. Total bits =  $\sum (\text{freq} \times \text{code length})$

END

## 2. Code:-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_TREE_HT 100
typedef struct MinHeapNode {
    char data;
    unsigned freq;
    struct MinHeapNode *left, *right;
} MinHeapNode;

typedef struct MinHeap {
    unsigned size;
    unsigned capacity;
    MinHeapNode** array;
}
```

```

MinHeap;
MinHeapNode* newNode(char data, unsigned freq) {
    MinHeapNode* temp = (MinHeapNode*)malloc(sizeof(MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

MinHeap* createMinHeap(unsigned capacity) {
    MinHeap* minHeap = (MinHeap*)malloc(sizeof(MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (MinHeapNode**)malloc(capacity * sizeof(MinHeapNode*));
    return minHeap;
}
void swapMinHeapNode(MinHeapNode** a, MinHeapNode** b) {
    MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}
void minHeapify(MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    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;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}
MinHeapNode* extractMin(MinHeap* minHeap) {
    MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[--minHeap->size];
    minHeapify(minHeap, 0);
    return temp;
}

```

```

void insertMinHeap(MinHeap* minHeap, MinHeapNode* node) {
    int i = minHeap->size++;
    while (i && node->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    minHeap->array[i] = node;
}

void buildMinHeap(MinHeap* minHeap) {
    for (int i = (minHeap->size - 1) / 2; i >= 0; i--)
        minHeapify(minHeap, i);
}

int isLeaf(MinHeapNode* node) {
    return !(node->left) && !(node->right);
}

MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
    MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; i++)
        minHeap->array[i] = newNode(data[i], freq[i]);
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
    MinHeapNode *left, *right, *top;
    MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    while (minHeap->size > 1) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }
    return extractMin(minHeap);
}

```

```

}

void printCodes(MinHeapNode* root, int arr[], int top, int* totalBits) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1, totalBits);
    }
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1, totalBits);
    }

    if (isLeaf(root)) {
        printf("Character: %c\n", root->data);
        printf("Codeword: ");
        for (int i = 0; i < top; i++)
            printf("%d", arr[i]);
        printf("\nBitSize: %d\n\n", top);
        *totalBits += root->freq * top;
    }
}

void HuffmanCodes(char data[], int freq[], int size) {
    MinHeapNode* root = buildHuffmanTree(data, freq, size);
    int arr[MAX_TREE_HT], top = 0, totalBits = 0;
    printCodes(root, arr, top, &totalBits);
    printf("Total bits needed for 100,000 characters: %d\n", totalBits);
}

int main() {
    int n;
    printf("Enter number of characters:\n");
    scanf("%d", &n);
    char data[n];
    int freq[n];
    for (int i = 0; i < n; i++) {
        printf("Enter character %d:\n", i + 1);
        scanf(" %c", &data[i]);
        printf("Enter frequency for '%c':\n", data[i]);
        scanf("%d", &freq[i]);
    }
    HuffmanCodes(data, freq, n);
}

```

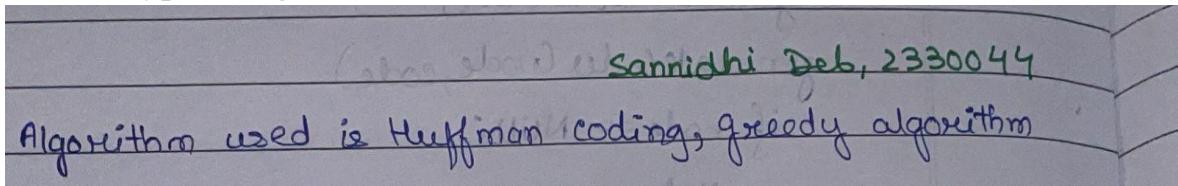
```
    printf("\nSannidhi Deb\nRoll No: 2330044\n");
    return 0;
}
```

### **3.Results/Output:- Entire Screen Shot including Date & Time:-**

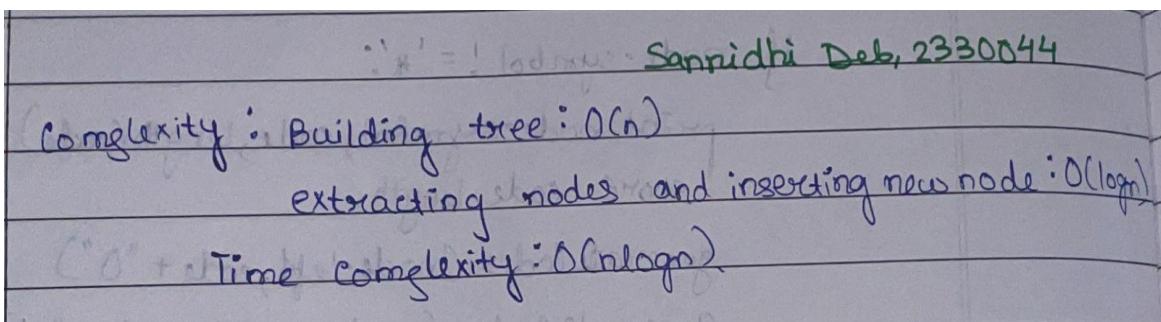
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 5 JavaSE-23 + ⋮  
C:\Users\debsa\OneDrive\Desktop\AL_Lab_044>a.exe  
Enter number of characters: 6  
Enter character 1: a  
Enter frequency for 'a': 45  
Enter character 2: b  
Enter frequency for 'b': 13  
Enter character 3: c  
Enter frequency for 'c': 12  
Enter character 4: d  
Enter frequency for 'd': 16  
Enter character 5: e  
Enter frequency for 'e': 9  
Enter character 6: f  
Enter frequency for 'f': 5  
Character: a | Codeword: 0 | BitSize: 1  
Character: c | Codeword: 100 | BitSize: 3  
Character: b | Codeword: 101 | BitSize: 3  
Character: f | Codeword: 1100 | BitSize: 4  
Character: e | Codeword: 1101 | BitSize: 4  
Character: d | Codeword: 111 | BitSize: 3  
  
Total bits needed for 100,000 characters: 224000  
  
Sannidhi Deb  
2330044
```

#### **4. Remarks:-**

- ### 1. What type of algorithm is used?



2. Analyze the complexity of your algorithm.



### 3. Any other observations?

Sannidhi Deb, 2330044

Total characters =  $45 + 13 + 12 + 16 + 9 + 5 = 100$   
and freq. is given in thousands, hence,  
Bits :  $(45000 \times 1) + (13000 \times 3) + (12000 \times 3) + (16000 \times 3) + (9000 \times 4) + (5000 \times 4)$   
Gives = 224000 bits

### 5. Conclusion:-

Huffman coding efficiently compresses data by assigning shorter codes to more frequent characters and longer codes to less frequent ones. It minimizes the total number of bits required for encoding, making it a powerful application of greedy algorithms in lossless data compression.

Sannidhi Deb  
(2330044)

Signature of the FIC

Sannidhi Deb

(Name of the FIC)