



Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

Experiment No.8
Implementation Huffman encoding(Tree) using Linked List
Name: Jaffari Mohammed Ali Sayyed Naqi Ali
Roll No: 16
Date of Performance:
Date of Submission:
Marks:
Sign:

Experiment No. 8: Huffman encoding (Tree) using Linked list

Aim: Implementation Huffman encoding (Tree) using Linked list

Objective:

The objective of this experiment is to implement and evaluate the Huffman coding algorithm using a linked list data structure to assess its efficiency in data compression.

Theory:

Huffman Coding is a widely used data compression algorithm that assigns variable-length codes to symbols in a dataset. It is a lossless compression technique that is based on the frequency of symbols in the data. The key idea behind Huffman Coding is to assign shorter codes to more frequent symbols and longer codes to less frequent symbols, thus optimizing the compression ratio.

Traditional Huffman Coding:



Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

1. **Frequency Calculation:** In traditional Huffman Coding, the first step is to calculate the frequency of each symbol in the dataset.
2. **Huffman Tree Construction:** Next, a binary tree called the Huffman tree is constructed. This tree is built using a greedy algorithm, where the two least frequent symbols are merged into a new node, and this process continues until a single tree is formed.
3. **Code Assignment:** The codes are assigned to symbols by traversing the Huffman tree. Left branches are assigned the binary digit "0," and right branches are assigned the binary digit "1." The codes are assigned such that no code is a prefix of another, ensuring unambiguous decoding.
4. **Compression:** The original data is then encoded using the Huffman codes, resulting in a compressed representation of the data.

Adaptive Huffman Coding:

1. **Initial Tree:** In Adaptive Huffman Coding, an initial tree is created with a predefined structure that includes a special symbol for escape. The escape symbol is used to signal that a new symbol is being introduced.
2. **Tree Update :**As symbols are encountered in the data, the tree is updated dynamically. When a symbol is encountered for the first time, it is added as a leaf node to the tree, and the escape symbol is used to navigate to its parent. This process ensures that new symbols can be encoded even if they were not present when the tree was initially created.
3. **Code Assignment:** The codes are assigned dynamically as the tree changes. More frequent symbols have shorter codes, and less frequent symbols have longer codes.
4. **Compression:** The data is encoded using the dynamically updated Huffman tree, resulting in compressed data. The tree is updated as the data is processed.



Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

Algorithm

Adaptive Huffman Coding algorithm using the FGK (Faller-Gallager-Knuth) variant:

Step1:- Initialization:

- Create an initial tree with the escape symbol and any predefined symbols.
- Set a pointer to the escape symbol.

Step 2:- Data Processing:

- Start processing symbols from the input data stream.
- If a symbol is encountered for the first time, add it as a new leaf node to the tree. Update the tree structure as needed to maintain the prefix property.
- Use the escape symbol as a way to navigate to the parent node of the new symbol.
- After each symbol is processed, the tree is adjusted to maintain the prefix property and optimal code lengths.

Step 3:- Code Assignment:

- Traverse the tree to assign variable-length codes to the symbols dynamically.
- More frequent symbols have shorter codes, and less frequent symbols have longer codes.

Step 4:- Compression:

- Encode the input data using the dynamically updated Huffman tree.
- The compressed data consists of the variable-length codes for each symbol.

Adaptive Huffman Coding adapts to the data as it is processed, allowing for efficient encoding of symbols even if their frequencies change over time. This adaptability makes it a suitable choice for scenarios where the data distribution is not known in advance or may change dynamically.

Code:

```
#include <stdio.h>
```



```
#include <stdlib.h>
```

```
// Define the structure for a Huffman Tree Node
```

```
struct HuffmanNode {  
    char data;  
    unsigned frequency;  
    struct HuffmanNode* left;  
    struct HuffmanNode* right;  
};
```

```
// Define a structure for a Min Heap Node
```

```
struct MinHeapNode {  
    struct HuffmanNode* node;  
    struct MinHeapNode* next;  
};
```

```
// Define a structure for a Min Heap
```

```
struct MinHeap {  
    struct MinHeapNode* head;  
};
```

```
// Function to create a new Min Heap Node
```

```
struct MinHeapNode* createMinHeapNode(struct HuffmanNode* node) {  
    struct MinHeapNode* newNode = (struct MinHeapNode*)malloc(sizeof(struct  
MinHeapNode));  
    newNode->node = node;  
    newNode->next = NULL;  
    return newNode;  
}
```



Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

// Function to create a new Min Heap

```
struct MinHeap* createMinHeap() {  
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));  
    minHeap->head = NULL;  
    return minHeap;  
}
```

// Function to insert a Min Heap Node

```
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* node) {  
    if (minHeap->head == NULL) {  
        minHeap->head = node;  
    } else {  
        if (node->frequency < minHeap->head->frequency) {  
            node->next = minHeap->head;  
            minHeap->head = node;  
        } else {  
            struct MinHeapNode* current = minHeap->head;  
            while (current->next != NULL && current->next->frequency < node->frequency) {  
                current = current->next;  
            }  
            node->next = current->next;  
            current->next = node;  
        }  
    }  
}
```

// Function to extract the minimum node from the Min Heap

```
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {  
    struct MinHeapNode* temp = minHeap->head;
```



Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

```
minHeap->head = minHeap->head->next;

return temp;
}

// Function to build the Huffman Tree
struct HuffmanNode* buildHuffmanTree(char data[], int frequency[], int n) {
    struct HuffmanNode *left, *right, *top;

    // Create a Min Heap and insert all characters into it
    struct MinHeap* minHeap = createMinHeap();
    for (int i = 0; i < n; ++i) {
        struct HuffmanNode* node = (struct HuffmanNode*)malloc(sizeof(struct HuffmanNode));
        node->data = data[i];
        node->frequency = frequency[i];
        node->left = node->right = NULL;
        insertMinHeap(minHeap, createMinHeapNode(node));
    }

    // Build the Huffman Tree
    while (minHeap->head != NULL) {
        left = extractMin(minHeap)->node;
        right = extractMin(minHeap)->node;

        top = (struct HuffmanNode*)malloc(sizeof(struct HuffmanNode));
        top->data = '\0';
        top->frequency = left->frequency + right->frequency;
        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, createMinHeapNode(top));
    }
}
```



```
}  
    return extractMin(minHeap)->node;  
}  
  
// Function to print the Huffman codes for each character  
void printHuffmanCodes(struct HuffmanNode* root, int arr[], int top) {  
    if (root->left) {  
        arr[top] = 0;  
        printHuffmanCodes(root->left, arr, top + 1);  
    }  
  
    if (root->right) {  
        arr[top] = 1;  
        printHuffmanCodes(root->right, arr, top + 1);  
    }  
  
    if (root->data) {  
        printf("%c: ", root->data);  
        for (int i = 0; i < top; i++) {  
            printf("%d", arr[i]);  
        }  
        printf("\n");  
    }  
}  
  
int main() {  
    char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};  
    int frequency[] = {5, 9, 12, 13, 16, 45};  
    int n = sizeof(data) / sizeof(data[0]);
```



Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

```
struct HuffmanNode* root = buildHuffmanTree(data, frequency, n);

int arr[100], top = 0;
printf("Huffman Codes:\n");
printHuffmanCodes(root, arr, top);

return 0;
}
```

Output:

Huffman Codes:

a: 1100

c: 1101

b: 111

f: 0

e: 10

d: 111

Conclusion:

1) What are some real-world applications of Huffman coding, and why it is preferred in those applications?

ANS:- Huffman coding, a popular data compression algorithm, finds application in various real-world scenarios, including data compression (ZIP, JPEG), network communication (HTTP), and storage devices (hard drives). Its efficiency in reducing data size without significant loss in quality makes it suitable for diverse data types and distributions. Huffman coding is preferred because it generates variable-length codes, with more frequent symbols represented by shorter codes, optimizing compression for data with skewed distributions like text and images. Additionally, it strikes a balance between compression efficiency and encoding/decoding complexity, making it a practical choice for a wide range of applications.



Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

2) What are the Limitations and potential drawbacks of using Huffman coding in practical data compression scenarios?

- ANS :- Inefficiency with small alphabets or uniformly distributed data.
- Variable-length codes may impact decoding speed.
- Lack of error resilience and security features.
- Complexity in maintaining and transmitting the codebook.
- Data compression consistency relies on input data statistics.
- Overhead introduced by the codebook and tree structure.
- Complexity of implementation, especially in dynamic versions.
- Potential patent and licensing issues.