Experiment No.8
Implementation Huffman encoding(Tree) using Linked List
Name: Tejashree Anand Karekar
Roll No:20
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:

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.

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;
}
```

```
// 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->node->frequency < minHeap->head->node->frequency) {
            node->next = minHeap->head;
            minHeap->head = node;
        } else {
            struct MinHeapNode* current = minHeap->head;
            while (current->next != NULL && current->next->node->frequency < node->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;
    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]);
    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 is a widely used data compression algorithm that is preferred in several real-world applications due to its efficiency in reducing data size. Here are some real-world applications of Huffman coding and why it is preferred in each:

1. **Data Compression**: Huffman coding is extensively used in data compression algorithms, including popular file compression formats like ZIP and GZIP. It efficiently represents and encodes data, reducing file sizes significantly. This makes it suitable for data storage, file transfer, and saving bandwidth.

2. **Text Compression**: In applications such as text messaging, email, and web browsing, where text data is transmitted, Huffman coding is used to compress text messages and documents. It helps reduce the amount of data transferred over networks and improves communication efficiency.

3. **Image Compression**: Huffman coding is used in image compression formats like JPEG. By encoding image data more efficiently, it reduces the size of image files while maintaining acceptable quality. This is crucial for applications like web image display and storage of large image databases.

4. **Voice and Audio Compression**: In voice communication, music streaming, and audio recording, Huffman coding is used to compress audio data. It helps reduce the size of audio files without significant loss of audio quality, making it suitable for streaming and storage of audio content.

5. **Video Compression**: In video streaming and video conferencing applications, Huffman coding is employed in video compression algorithms like H.264. It plays a key role in reducing the size of video files and real-time video transmission over networks.

6. **Network Data Transfer**: Huffman coding can be used in network protocols to compress data before transmission. This is particularly useful in scenarios where bandwidth is limited or costly, such as satellite communications and IoT devices.

7. **Error Correction Codes**: Huffman coding is also used in error-correcting codes to improve data reliability during transmission. By encoding data with Huffman codes, error detection and correction can be more efficient and reliable, benefiting applications like wireless communication and data storage.

Huffman coding is preferred in these applications due to its ability to achieve near-optimal compression by assigning shorter codes to more frequent symbols, which minimizes the overall encoded data length. This results in reduced storage requirements, faster data transmission, and improved efficiency, making it a valuable tool in various data compression and encoding scenarios.

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

ANS :- 1. Variable-length codes: Huffman coding generates variable-length codes, which can lead to complications in encoding and decoding, requiring additional bookkeeping.

2. Inefficiency with small symbol sets: For small symbol sets, Huffman coding may not provide significant compression benefits, and its overhead can even outweigh the compression gains.

3. Lack of adaptability: Huffman coding relies on fixed probabilities for symbol frequencies, making it less adaptive to changes in data characteristics.

4. Not always optimal: While Huffman coding provides a near-optimal compression for a static data source, it may not perform optimally for all types of data.

5. Compression overhead: The need to transmit the Huffman tree structure can introduce some overhead, especially for very short messages or in situations where the tree structure must be sent along with the compressed data.