Experiment No.8
Implementation Huffman encoding(Tree) using Linked List
Name: Gautam D. Chaudhari
Roll No: 04
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.

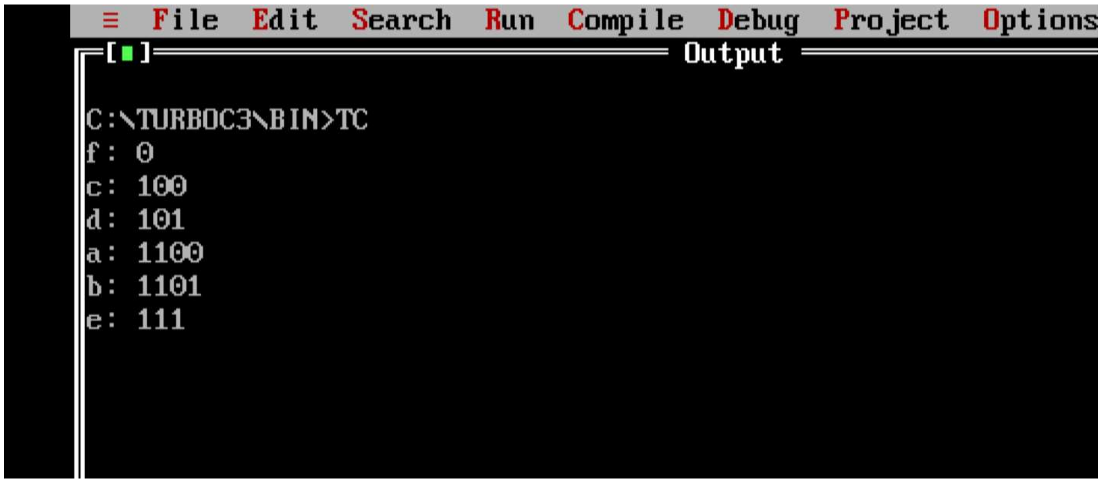
Adaptive Huffman Coding adapts to the data as it is processed, allowing for efficient encoding of symbols even if their frequencies change over time. This adaptability makes it a suitable choice for scenarios where the data distribution is not known in advance or may change dynamically.

Code:

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

#define MAX_TREE_HT 100

struct MinHeapNode {
    char data;
    unsigned freq;
    struct MinHeapNode* left;
    struct MinHeapNode* right;
};

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

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

struct MinHeap* createMinHeap(unsigned capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct
MinHeapNode*));
    return minHeap;
}

void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

void minHeapify(struct 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);
    }
}

int isSizeOne(struct MinHeap* minHeap) {
    return (minHeap->size == 1);
}

struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

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

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

void printArr(int arr[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}

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

struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
    struct 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;
}

struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
    struct MinHeapNode *left, *right, *top;
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);
    while (!isSizeOne(minHeap)) {
        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(struct MinHeapNode* root, int arr[], int top) {
   if (root->left) {
      arr[top] = 0;
      printCodes(root->left, arr, top + 1);
   }
   if (root->right) {
      arr[top] = 1;
      printCodes(root->right, arr, top + 1);
   }
   if (isLeaf(root)) {
      printf("%c: ", root->data);
      printArr(arr, top);
   }
}

void HuffmanCodes(char data[], int freq[], int size) {
   struct MinHeapNode* root = buildHuffmanTree(data, freq, size);
   int arr[MAX_TREE_HT], top = 0;
   printCodes(root, arr, top);
}

int main() {
   char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
   int freq[] = { 5, 9, 12, 13, 16, 45 };
   int size = sizeof(arr) / sizeof(arr[0]);
   HuffmanCodes(arr, freq, size);
   return 0;
}
```

Output:



```
C:\TURBOC3\BIN>TC
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Conclusion:

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

Huffman coding is used in various real-world applications due to its efficiency in data compression, which minimizes the storage or transmission requirements of information.

- Data Compression: Huffman coding reduces file sizes for efficient storage and transmission.
- Image and Video Compression: Used in formats like JPEG and MPEG for smaller images and videos.
- Text Compression: Applied in formats like ePub and PDF for compact text documents.
- Data Transmission: Reduces data size in network protocols, improving efficiency.
- Error Correction: Used with error-correcting codes for data reliability.
- Audio Compression: Employed in formats like MP3 for smaller audio files with acceptable quality.
- Archive Formats: Used in archive formats like TAR and ARJ for space-efficient file storage.
- Cloud Storage: Enhances data storage and transfer efficiency in cloud services.
- Resource-Constrained Devices: Suitable for IoT sensors and embedded systems to save energy and storage space.

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

Huffman coding is a powerful compression technique, but it also has some limitations and potential drawbacks in practical data compression scenarios:

- Variable-Length Codes: Huffman codes result in variable-length bit sequences, which can complicate random access within compressed data.
- Compression Efficiency: It may not achieve the highest compression levels compared to more advanced methods like LZW or BWT.
- Preprocessing Overhead: Building Huffman trees can be computationally expensive, especially for large datasets.
- Lossless Compression: Suitable for lossless compression, but not ideal for lossy compression scenarios like images or audio.
- Inefficiency with Small Alphabets: Less efficient for very small datasets or alphabets.
- Lack of Adaptability: Does not adapt well to changing data statistics; not ideal for dynamic data.
- Complexity: More complex compression algorithms, like DEFLATE, may outperform Huffman coding.
- Loss of Compression for Smaller Datasets: Overhead can lead to larger compressed files for very small datasets.
- Non-Uniform Frequency Distribution: Less effective if symbol frequencies are evenly distributed.