Experiment No.8
Implementation Huffman encoding (Tree) using Linked List
Name: AARYAN CHANDRAKANT GOLE
Roll No: 12
Date of Performance:
Date of Submission:
Marks:
Sign:

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

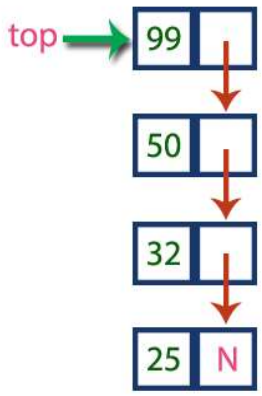
Aim: Implementation Huffman encoding (Tree) using Linked list

Objective:

Stack can be implemented using linked list for dynamic allocation. Linked list implementation gives flexibility and better performance to the stack.

Theory:

A stack implemented using an array has a limitation in that it can only handle a fixed number of data values, and this size must be defined at the outset. This limitation makes it unsuitable for cases where the data size is unknown. On the other hand, a stack implemented using a linked list is more flexible and can accommodate an unlimited number of data values, making it suitable for variable-sized data. In a linked list-based stack, each new element becomes the 'top' element, and removal is achieved by updating 'top' to point to the previous node, effectively popping the element. The first element's "next" field should always be NULL to indicate the end of the list.



Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.

Step 2 - Define a 'Node' structure with two members data and next.

Step 3 - Define a Node pointer 'top' and set it to NULL.

Step 4 - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

push(value) - Inserting an element into the Stack

Step 1 - Create a newNode with given value.

Step 2 - Check whether stack is Empty (top == NULL)

Step 3 - If it is Empty, then set newNode → next = NULL.

Step 4 - If it is Not Empty, then set newNode → next = top.

Step 5 - Finally, set top = newNode.

pop() - Deleting an Element from a Stack

Step 1 - Check whether the stack is Empty (top == NULL).

Step 2 - If it is Empty, then display "Stack is Empty!!!

Step 3 - If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.

Step 4 - Then set 'top = top → next'.

Step 5 - Finally, delete 'temp'. (free(temp)).

display() - Displaying stack of elements

Step 1 - Check whether stack is Empty (top == NULL).

Step 2 - If it is Empty, then display 'Stack is Empty!!!' and terminate the function.

Step 3 - If it is Not Empty, then define a Node pointer 'temp' and initialize with top.

Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack. (temp → next != NULL).

Step 5 - Finally! Display 'temp → data ---> NULL'.

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?

- **Data Compression:**
  Huffman coding is extensively used in data compression algorithms like ZIP, GZIP, and DEFLATE. It's efficient for compressing text, images, and other data types.
- **Image and Video Compression:**
  JPEG (Joint Photographic Experts Group) uses Huffman coding to compress images. MPEG (Moving Picture Experts Group) standards also employ Huffman coding for video compression.
- **Text Compression:**
  Huffman coding is used in text editors, search engines, and document storage systems to compress and store text data efficiently.
- **Data Encryption:**
  - Huffman codes are employed in various encryption and data encoding schemes.
  - They assist in reducing the size of encrypted data while preserving data security.

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

1. **Not Suitable for All Data Types:** Huffman coding is most effective for data with non-uniform symbol frequencies, where some symbols are more common than others. For data with a relatively uniform distribution, other compression methods like Run-Length Encoding (RLE) may be more efficient.
2. **Variable-Length Codes:** Huffman coding generates variable-length codes, which can complicate storage and transmission. Decoding variable-length codes efficiently requires additional bookkeeping, making it less suitable for certain real-time applications.

3. **Limited Compression for Small Data Sets:** For very small data sets, Huffman coding might not provide significant compression benefits due to the overhead of maintaining the codebook and encoding/decoding data.
4. **Complexity and Overhead:** Creating and maintaining the Huffman tree, especially in real-time applications, can add complexity and computational overhead. It may not be the best choice for scenarios where speed is of the essence.
5. **Lossless Compression Only:** Huffman coding is primarily used for lossless compression, which preserves data integrity. It is not suitable for applications where lossy compression is acceptable or preferred, such as image or audio compression.
6. **Lack of Security:** Huffman coding does not provide any security or encryption of the data. It is not designed to protect data from unauthorized access, and additional encryption methods may be required when security is a concern.