Experiment No.8
Implementation Huffman encoding(Tree) using Linked List
Name:Ayush Gupta
Roll No:14
Date of Performance:
Date of Submission:
Marks:
Sign:

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

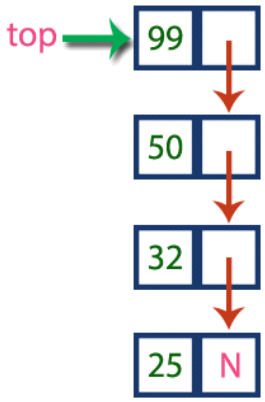
Aim: Implementation Huffman encoding (Tree) using Linked list

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'.

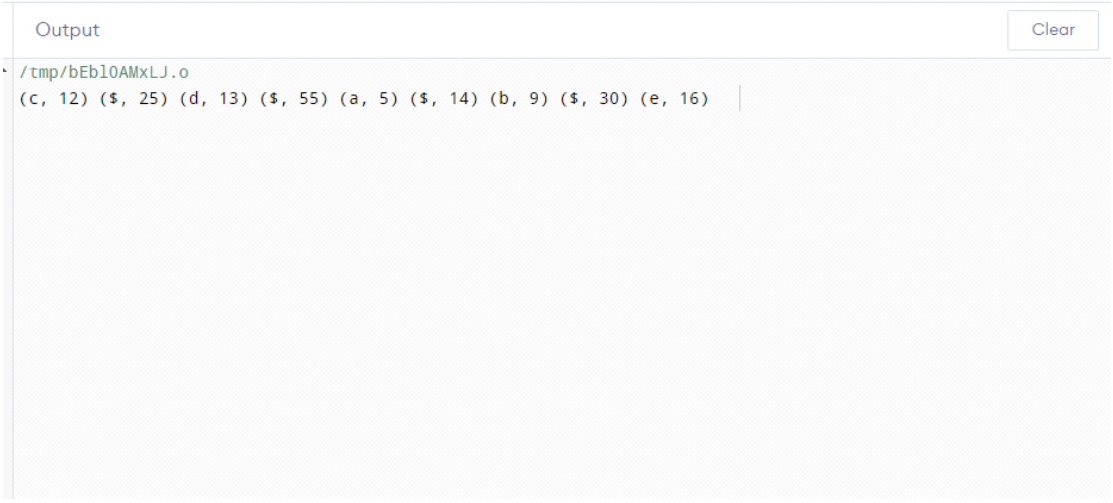
Code:

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure for a node in the Huffman tree
struct Node {
    char data;
    int frequency;
    struct Node* left;
    struct Node* right;
    struct Node* next; // For linked list
};
// Function to create a new node
struct Node* createNode(char data, int frequency) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
```

```
    newNode->frequency = frequency;
    newNode->left = newNode->right = newNode->next = NULL;
    return newNode;
}
// Function to insert a node into a sorted linked list by frequency
void insertSorted(struct Node** list, struct Node* newNode) {
    if (*list == NULL || (*list)->frequency >= newNode->frequency) {
        newNode->next = *list;
        *list = newNode;
    } else {
        struct Node* current = *list;
        while (current->next != NULL && current->next->frequency < newNode->frequency) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}
// Function to build a Huffman tree from a sorted linked list
struct Node* buildHuffmanTree(struct Node* list) {
    while (list->next != NULL) {
        struct Node* newNode = createNode('$', list->frequency + list->next->frequency);
        newNode->left = list;
        newNode->right = list->next;
        list = list->next->next;
        insertSorted(&list, newNode);
    }
    return list; // The last node in the list is the root of the Huffman tree
```

```
}
// Function to display the Huffman tree (inorder traversal)
void displayHuffmanTree(struct Node* root) {
    if (root != NULL) {
        displayHuffmanTree(root->left);
        printf("(%c, %d) ", root->data, root->frequency);
        displayHuffmanTree(root->right);
    }
}
int main() {
    // Sample character frequencies
    char characters[] = {'a', 'b', 'c', 'd', 'e'};
    int frequencies[] = {5, 9, 12, 13, 16};
    int n = sizeof(characters) / sizeof(characters[0]);
    // Build a sorted linked list of nodes
    struct Node* list = NULL;
    for (int i = 0; i < n; i++) {
        insertSorted(&list, createNode(characters[i], frequencies[i]));
    }
    // Build the Huffman tree
    struct Node* huffmanTree = buildHuffmanTree(list);
    // Display the Huffman tree
    displayHuffmanTree(huffmanTree);
    return 0;
}
```

Output:



Conclusion:

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

Huffman coding is widely used in various real-world applications due to its efficiency in data compression. Here are some common applications:

1. Data Compression: Huffman coding is used in data compression algorithms for text, images, audio, and video files. It is preferred because it produces compact representations of data by assigning shorter codes to more frequently occurring symbols, leading to reduced storage requirements and faster data transmission.

2. Image Compression (JPEG): In image compression standards like JPEG (Joint Photographic Experts Group), Huffman coding is employed to compress the quantized coefficients in the Discrete Cosine Transform (DCT) step. Huffman codes are used to represent the frequent and infrequent coefficients efficiently.

3. Audio Compression (MP3): In audio compression formats like MP3, Huffman coding is used to encode the quantized values of spectral components (e.g., frequency, amplitude) in the transformed audio signal. This results in smaller file sizes without significant loss of audio quality.

4. Text Compression (ZIP): File compression tools like ZIP and GZIP use Huffman coding to reduce the size of text files. Frequently used characters are assigned shorter codes, leading to effective compression of text documents.
5. Network Data Transmission: In network protocols, Huffman coding is applied for efficient data transmission. For example, it is used in HTTP for encoding headers and content compression, reducing the amount of data transferred over the internet.
6. Error Correction Codes: Huffman coding is used in error correction codes, such as Huffman codes used in the Reed-Solomon error correction algorithm. These codes are vital for data integrity in storage and communication systems.
7. Dictionaries in Data Structures: Huffman trees (trees used to construct Huffman codes) are employed in various data structures like dictionaries, where quick access to frequently used elements is essential.
8. DNA Sequence Compression: Huffman coding can be applied to compress DNA sequences, which often contain repetitive patterns. This helps reduce the storage requirements for large-scale genomic data.
9. Telecommunications: In telecommunication systems, Huffman coding can be used to compress voice and data transmission, making more efficient use of bandwidth and reducing costs.
10. Resource-Constrained Devices: In embedded systems and IoT (Internet of Things) devices with limited memory and processing power, Huffman coding can be used to compress data before storage or transmission, optimizing resource usage.

Huffman coding is preferred in these applications due to its ability to provide lossless compression while ensuring that frequently occurring symbols are represented using shorter codes. This results in significant data reduction without losing information, making it a valuable tool for various data compression and optimization tasks in real-world scenarios.

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

Huffman coding limitations:

1. Primarily for lossless compression.
2. Variable-length codes can be complex.
3. Encoding and decoding can be computationally intensive.
4. Overhead due to codebook transmission/storage.
5. Inefficient for small data.
6. Not optimal for mixed data types.
7. Less effective for equal symbol frequencies.
8. Limited compression compared to newer methods.
9. Efficiency can degrade in adaptive Huffman coding.

CSL303: Data Structure