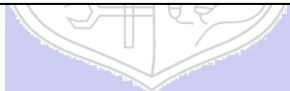




Experiment No. : 4

Title: Implement Huffman Algorithm using Greedy approach



Experiment No.: 04

Aim: To Implement Huffman Algorithm using Greedy approach and analyse its time complexity.

Algorithm of Huffman Algorithm:

```

HUFFMAN( $C$ )
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$     // return the root of the tree

```

Explanation and Working of Variable Length Huffman Algorithm:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters; lengths of the assigned codes are based on the frequencies of corresponding characters.

The variable-length codes assigned to input characters are Prefix Codes, meaning the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of the frequency field is used to compare two nodes in the min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents the root of a tree with a single node.

Step 2 Extract two minimum frequency nodes from the min heap. Add a new internal node with frequency $5 + 9 = 14$.

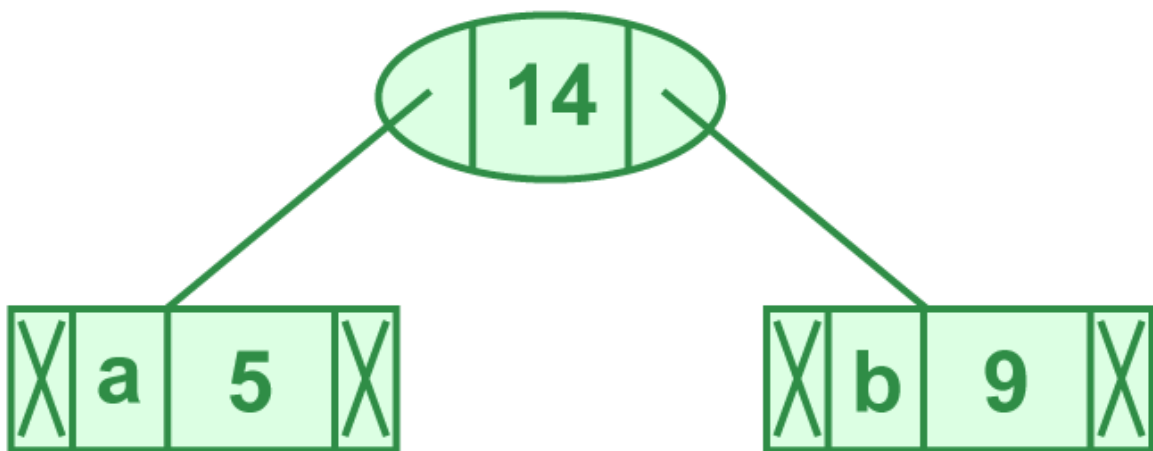


Illustration of step 2

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency $12 + 13 = 25$

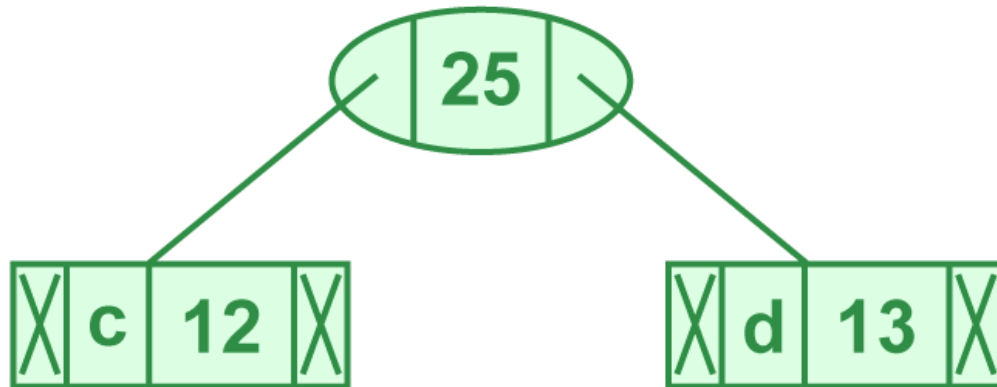


Illustration of step 3

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency $14 + 16 = 30$

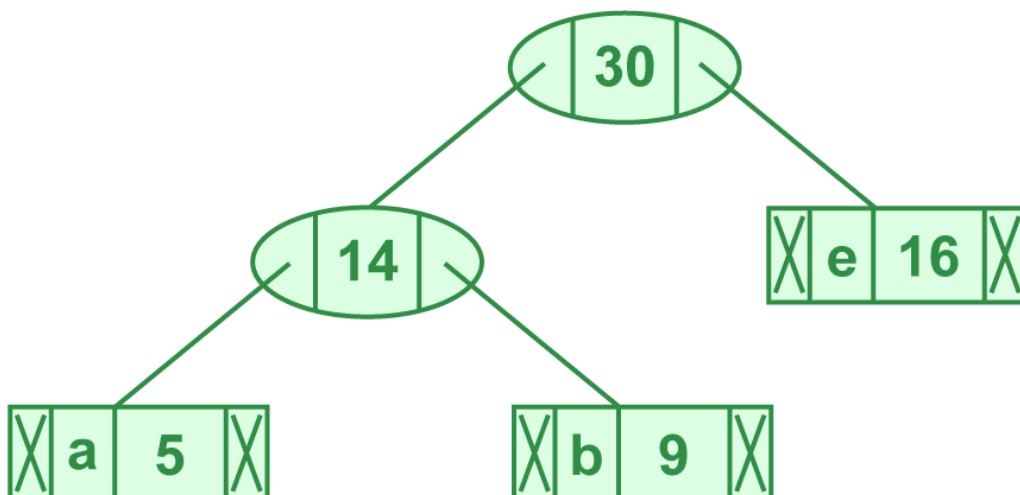


Illustration of step 4

Now the min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency $25 + 30 = 55$

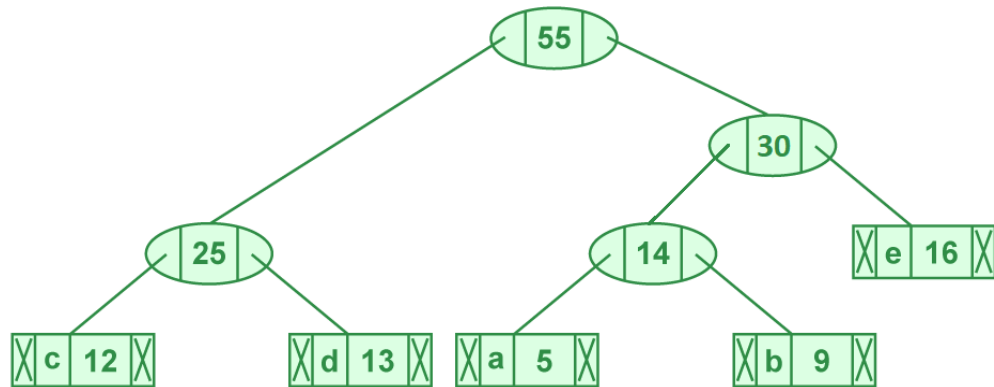


Illustration of step 5

Now the min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55



Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency $45 + 55 = 100$

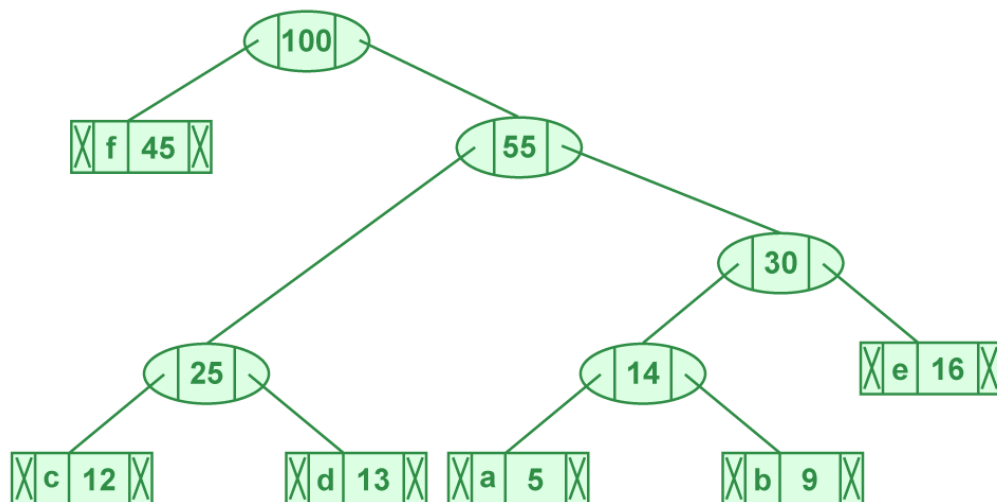


Illustration of step 6

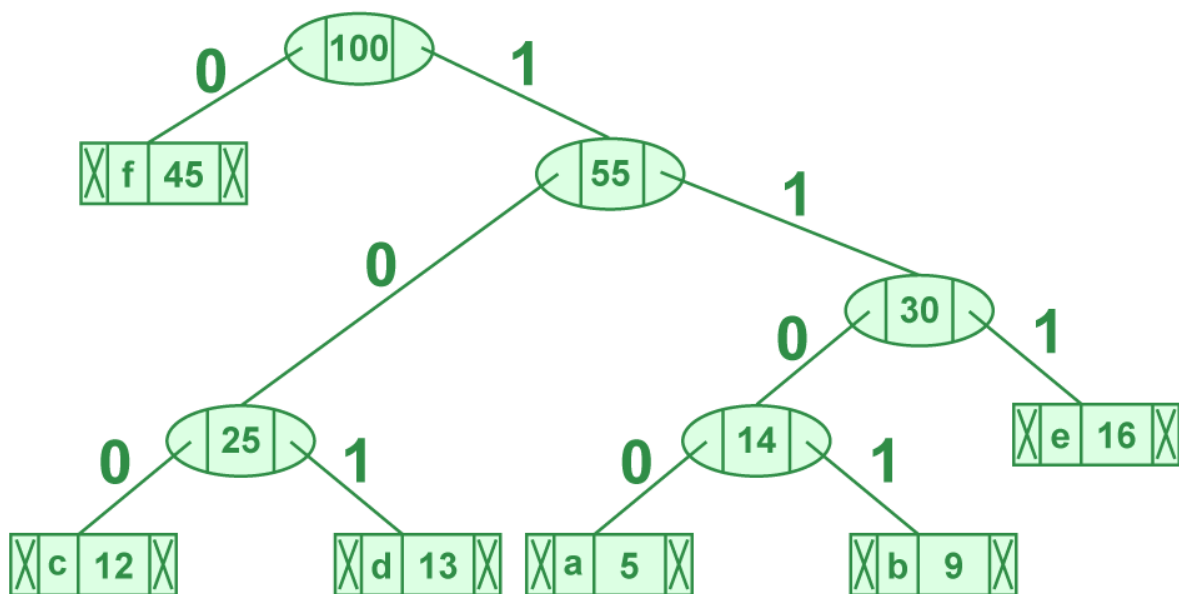
Now min heap contains only one node.

character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



Steps to print code from HuffmanTree

The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Derivation of Huffman Algorithm:

Time complexity Analysis

The Huffman Algorithm is a popular method used in data compression, employing a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm builds the code table based on the estimated probability or frequency of occurrence for each possible value of the source symbol. The time complexity of the Huffman Algorithm is

primarily determined by the process of building the Huffman tree, which is used to generate the codes.

Steps in Huffman Algorithm:

1. Create a leaf node for each symbol and add it to a priority queue or min-heap based on its frequency. Complexity: $O(n \log n)$ for inserting all nodes into the heap, where n is the number of unique symbols.

2. Build the Huffman Tree by repeating the following steps until there is only one node in the heap:

- Remove the two nodes of the lowest frequency from the heap. Complexity per operation: $O(\log n)$
- Create a new node with a frequency equal to the sum of the two nodes' frequencies. Add this new node to the heap. Complexity per operation: $O(\log n)$
- This step is repeated $n - 1$ times, where n is the number of unique symbols.

Derivation of Time Complexity:

- **Insertion of n nodes into a heap:** Initially, you insert n nodes into the heap, which takes $O(n \log n)$ time.
- **Building the Huffman tree:**
 - Each iteration of removing two nodes and inserting one node takes $O(\log n)$ time because each operation on the heap (insertion or deletion) takes $O(\log n)$ time, where n is the number of elements in the heap.
 - Since you combine two nodes at each step and the number of steps required is $n - 1$, the total time for building the Huffman tree is $O(n \log n)$.

Therefore, the overall time complexity of the Huffman Algorithm is dominated by the heap operations, leading to a time complexity of $O(n \log n)$.

This complexity assumes that the heap operations (insert and delete min) are the most expensive operations, which is generally true for a heap implemented with an array or a binary tree. If you use a more sophisticated data structure that can insert and delete in faster than $\log n$ time, the complexity might differ, but in practice, a binary heap is commonly used, making $O(n \log n)$ the typical time complexity for Huffman coding.

Program(s) of Huffman Algorithm:

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

```
// A Huffman tree node
struct MinHeapNode {
    char data; // One of the input characters
    unsigned freq; // Frequency of the character
    struct MinHeapNode *left, *right; // Left and right child
};
```

```
// A Min Heap: Collection of min-heap (or Huffman tree) nodes
struct MinHeap {
```

```

unsigned size; // Current size of min heap
unsigned capacity; // Capacity of min heap
struct MinHeapNode** array; // Array of minheap node pointers
};

// A utility function to create a new min heap node
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;
}

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    minHeap->size = 0; // current size is 0
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct
MinHeapNode*));
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

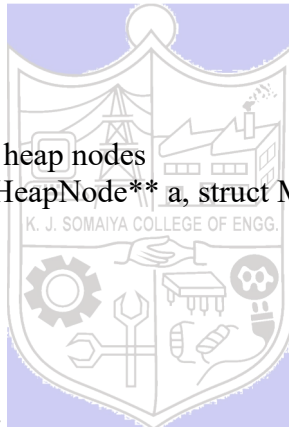
// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    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);
    }
}

```




```

// A utility function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap) {
    return (minHeap->size == 1);
}

// A standard function to extract minimum value node from heap
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;
}

// A utility function to insert a new node to Min Heap
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;
}

// A standard function to build min heap
void buildMinHeap(struct MinHeap* minHeap) {
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// A utility function to print an array of size n
void printArr(int arr[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}

// Utility function to check if this node is leaf
int isLeaf(struct MinHeapNode* root) {
    return !(root->left) && !(root->right);
}

// Creates a min heap of capacity equal to size and inserts all character of data[] in min heap.
Initially size of min heap is equal to capacity
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;
}

// The main function that builds Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
    struct MinHeapNode *left, *right, *top;

    // Step 1: Create a min heap of capacity equal to size. Initially, there are nodes equal to size.
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap)) {
        // Step 2: Extract the two minimum freq items from min heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Step 3: Create a new internal node with frequency equal to the sum of the two nodes
        // frequencies. Make the two extracted node as left and right children of this new node. Add this
        // node to the min heap
        // '$' is a special value for internal nodes, not used
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    // Step 4: The remaining node is the root node and the tree is complete.
    return extractMin(minHeap);
}

// Prints Huffman codes using the Huffman tree built above
void printCodes(struct MinHeapNode* root, int arr[], int top) {
    // Assign 0 to left edge and recur
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then print the character and its code from arr[]

```

```

if (isLeaf(root)) {
    printf("%c: ", root->data);
    printArr(arr, top);
}
}

```

// The main function that builds a Huffman Tree and print codes by traversing the built Huffman Tree

```

int main() {
    int size;
    printf("Enter the number of characters: ");
    scanf("%d", &size);

    char arr[size];
    int freq[size];

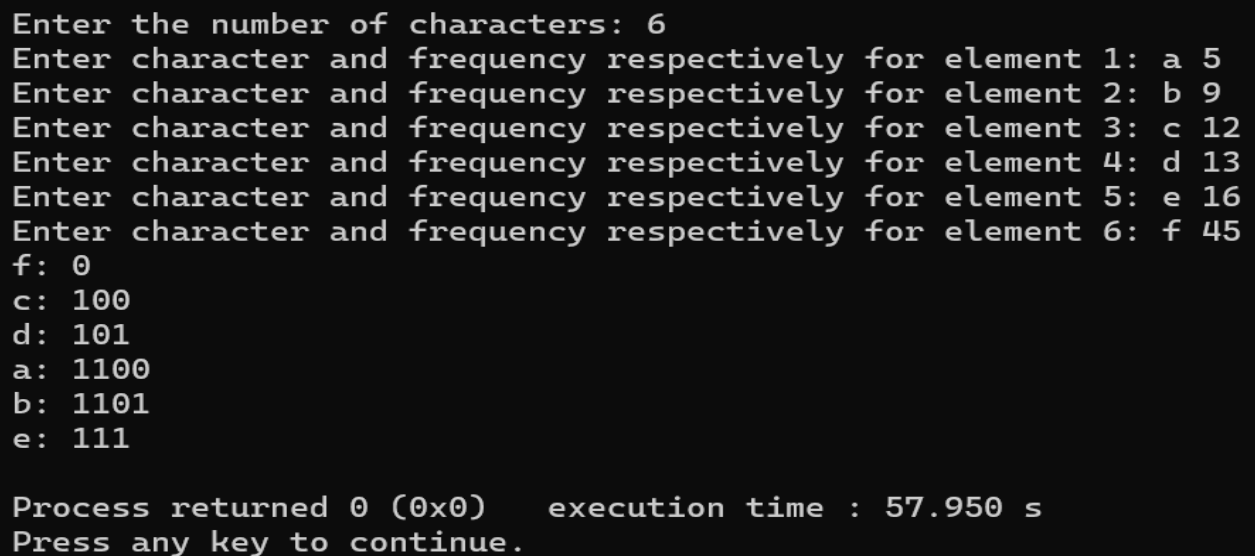
    // Taking input from user
    for (int i = 0; i < size; ++i) {
        printf("Enter character and frequency respectively for element %d: ", i + 1);
        scanf(" %c %d", &arr[i], &freq[i]);
    }

    struct MinHeapNode* root = buildHuffmanTree(arr, freq, size);

    int codes[size], top = 0;
    printCodes(root, codes, top);
    return 0;
}

```

Output(o) of Huffman Algorithm:



```

Enter the number of characters: 6
Enter character and frequency respectively for element 1: a 5
Enter character and frequency respectively for element 2: b 9
Enter character and frequency respectively for element 3: c 12
Enter character and frequency respectively for element 4: d 13
Enter character and frequency respectively for element 5: e 16
Enter character and frequency respectively for element 6: f 45
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111

Process returned 0 (0x0)    execution time : 57.950 s
Press any key to continue.

```

Post Lab Questions:

Differentiate between Fixed length and Variable length Coding with suitable examples.

Ans: Fixed-length coding and variable-length coding are both methods used in data compression to represent symbols or data elements with different lengths. Here's how they differ:

Fixed-Length Coding:

- In fixed-length coding, each symbol or data element is represented using the same number of bits.
- This method assigns a specific bit pattern to each symbol, regardless of its frequency or probability of occurrence.
- Fixed-length coding is straightforward and easy to implement, but it may not be the most efficient in terms of compression, especially if some symbols are more common than others.
- Examples:
ASCII (American Standard Code for Information Interchange) encoding: Each character is represented using 8 bits (1 byte), regardless of its frequency or usage.
Morse code: Each letter or symbol is represented using a sequence of dots and dashes, with each having a fixed length.

Variable-Length Coding:

- In variable-length coding, symbols or data elements are represented using variable numbers of bits.
- More frequently occurring symbols are assigned shorter codewords, while less frequent symbols are assigned longer codewords. This exploits the statistical properties of the data to achieve better compression.
- Variable-length coding schemes typically use prefix codes, ensuring that no codeword is a prefix of another, making it unambiguous during decoding.
- Variable-length coding is more complex to implement compared to fixed-length coding but can achieve better compression ratios, especially for data with non-uniform distributions.
- Examples:
Huffman coding: A widely used variable-length coding technique where the most frequent symbols are represented using shorter codewords, and less frequent symbols are represented using longer codewords. For instance, in a text document, common letters like 'e' and 't' may be represented by shorter bit sequences than less common letters like 'z' or 'q'.
Adaptive Huffman coding: Similar to Huffman coding but with the added capability of dynamically updating the codebook as symbols are encountered in the data stream.

Conclusion: (Based on the observations)

The experiment demonstrated the effectiveness and practical relevance of the Huffman algorithm for data compression tasks. Despite its simplicity and greedy nature, the algorithm produces optimal prefix codes and operates with a time complexity that scales reasonably well with the size of the input data, making it a valuable tool for efficient data encoding and compression.

Outcome: Implement Greedy and Dynamic Programming algorithms

References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Cormen ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.

