Experiment No. : 4

Title: Implement Huffman Algorithm using Greedy approach

Batch: B2 **Roll No.: 16010421119**
Experiment No.: 4

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

Algorithm of Huffman Algorithm: Refer Coreman for Explaination

```
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 = $ EXTRACT-MIN($Q$)
6      $z.right = y = $ EXTRACT-MIN($Q$)
7      $z.freq = x.freq + y.freq$
8      INSERT($Q, z$)
9  return 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 used for encoding information. It is widely used in digital communication systems, including fax machines, modems, and file compression software.

The Huffman coding algorithm works by assigning variable-length codes to characters based on their frequency of occurrence in the input data. The more frequently a character appears in the input, the shorter its code will be.

Here are the steps involved in the Huffman coding algorithm:

1. **Calculate the frequency of occurrence of each character in the input data.**
2. **Sort the characters by their frequency of occurrence, with the most frequent characters appearing first.**
3. **Combine the two least frequent characters into a single node, and assign the sum of their frequencies as the frequency of the new node. Repeat this process until all the nodes are combined into a single tree.**
4. **Assign a '0' to each left branch and a '1' to each right branch of the tree.**
5. **Traverse the tree to generate a code for each character in the input data, using the path from the root to the character's leaf node. Concatenate the '0's and '1's along this path to form the code for that character.**
6. **Encode the input data using the generated codes.**

Derivation of Huffman Algorithm:

Time complexity Analysis

The time complexity of step 1 is O(n), as we need to count the frequency of each symbol in the input data. This can be done efficiently using a hash table or an array.

The time complexity of step 2 is O(n), as we need to create a node for each symbol and its frequency.

The time complexity of step 3 is O(n log n), as we need to merge nodes repeatedly until we have a single root node. This can be done efficiently using a priority queue or a heap.

The time complexity of step 4 is O(n), as we need to traverse the tree once to assign bit codes to each symbol.

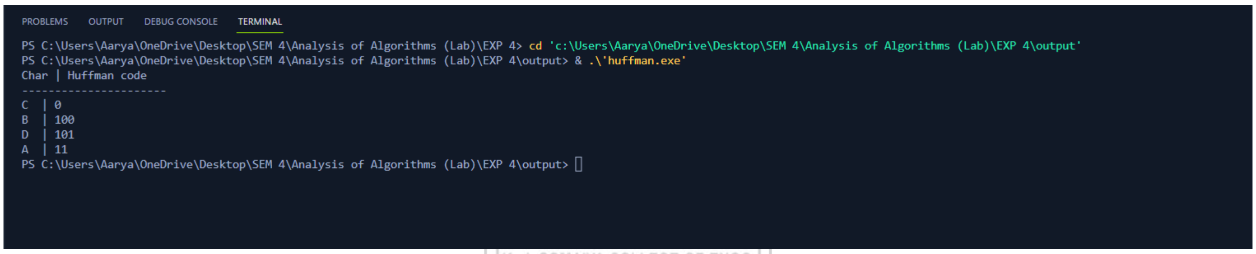
Therefore, the overall time complexity of the Huffman coding algorithm is O(n log n), dominated by step 3. This makes the algorithm efficient for large inputs, as it scales well with the number of symbols.

Program(s) of Huffman Algorithm:

```
#include <iostream>
using namespace std;

#define MAX_TREE_HT 50

struct node
{
    int frequency;
    char data;
    struct node *l, *r;
};

struct tree
{
    int size;
    int capacity;
    struct node **array;
};

struct node *newNode(char data, int frequency)
{
    struct node *temp = (struct node *)malloc(sizeof(struct
node));

    temp->l = temp->r = NULL;
    temp->data = data;
    temp->frequency = frequency;
    return temp;
}
struct tree *createtree(int capacity)
{
    struct tree *treeeap = (struct tree *)malloc(sizeof(struct
tree));
    treeeap->size = 0;
    treeeap->capacity = capacity;
    treeeap->array = (struct node **)malloc(treeeap->capacity *
sizeof(struct node *));
    return treeeap;
}

void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        cout << arr[i];

    cout << "\n";
}

void swap(struct node **a, struct node **b)
{
    struct node *t = *a;
    *a = *b;
    *b = t;
}

void treeeapify(struct tree *treeeap, int idx)
{
    int smallest = idx;
    int l = 2 * idx + 1;
    int r = 2 * idx + 2;

    if (l < treeeap->size && treeeap->array[l]->frequency <
treeeap->array[smallest]->frequency)
        smallest = l;

    if (r < treeeap->size && treeeap->array[r]->frequency <
treeeap->array[smallest]->frequency)
        smallest = r;

    if (smallest != idx)
    {
        swap(&treeeap->array[smallest], &treeeap->array[idx]);
        treeeapify(treeeap, smallest);
    }
}
int checkSizeOne(struct tree *treeeap)
{
    return (treeeap->size == 1);
}


struct node *extractMin(struct tree *treeeap)
{
    struct node *temp = treeeap->array[0];
    treeeap->array[0] = treeeap->array[treeeap->size - 1];

    --treeeap->size;
    treeeapify(treeeap, 0);

    return temp;
}

void inserttreeeap(struct tree *treeeap, struct node
*treeeapNode)
{
    ++treeeap->size;
    int i = treeeap->size - 1;

    while (i && treeeapNode->frequency < treeeap->array[(i - 1)
/ 2]->frequency)
    {
        treeeap->array[i] = treeeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }

    treeeap->array[i] = treeeapNode;
}

void buildtreeeap(struct tree *treeeap)
{
    int n = treeeap->size - 1;
    int i;

    for (i = (n - 1) / 2; i >= 0; --i)
        treeeapify(treeeap, i);
}

int isLeaf(struct node *root)
{
    return !(root->l) && !(root->r);
}

struct tree *createAndBuildtreeeap(char data[], int frequency[],
int size)
{
    struct tree *treeeap = createtree(size);

    for (int i = 0; i < size; ++i)
```

```
        treeeap->array[i] = newNode(data[i], frequency[i]);

    treeeap->size = size;
    buildtreeeap(treeeap);

    return treeeap;
}

struct node *buildHfTree(char data[], int frequency[], int size)
{
    struct node *l, *r, *top;
    struct tree *treeeap = createAndBuildtreeeap(data,
frequency, size);

    while (!checkSizeOne(treeeap))
    {
        l = extractMin(treeeap);
        r = extractMin(treeeap);

        top = newNode('$', l->frequency + r->frequency);

        top->l = l;
        top->r = r;

        inserttreeeap(treeeap, top);
    }
    return extractMin(treeeap);
}

void printHCodes(struct node *root, int arr[], int top)
{
    if (root->l)
    {
        arr[top] = 0;
        printHCodes(root->l, arr, top + 1);
    }

    if (root->r)
    {
        arr[top] = 1;
        printHCodes(root->r, arr, top + 1);
    }
    if (isLeaf(root))
    {
        cout << root->data << "  | ";
        printArray(arr, top);
    }
}

void HuffmanCodes(char data[], int frequency[], int size)
{
    struct node *root = buildHfTree(data, frequency, size);
    int arr[MAX_TREE_HT], top = 0;
    printHCodes(root, arr, top);
}

int main()
{
    char arr[] = {'A', 'B', 'C', 'D'};
    int frequency[] = {5, 1, 6, 3};
    int size = sizeof(arr) / sizeof(arr[0]);
    cout << "Char | Huffman code ";
    cout << "\n----------------------\n";
    HuffmanCodes(arr, frequency, size);
}
```

<u>Output(o) of Huffman Algorithm:</u>



```
PS C:\Users\Aarya\OneDrive\Desktop\SEM 4\Analysis of Algorithms (Lab)\EXP 4> cd 'c:\Users\Aarya\OneDrive\Desktop\SEM 4\Analysis of Algorithms (Lab)\EXP 4\output'
PS C:\Users\Aarya\OneDrive\Desktop\SEM 4\Analysis of Algorithms (Lab)\EXP 4\output> & .\'huffman.exe'
Char | Huffman code
----------------------
C  | 0
B  | 100
D  | 101
A  | 11
PS C:\Users\Aarya\OneDrive\Desktop\SEM 4\Analysis of Algorithms (Lab)\EXP 4\output> 
```

Post Lab Questions:-

1. **Differentiate between Fixed length and Variable length Coding with suitable example.**

Ans : Fixed length encoding means that each thing you encode ends up the same length. Variable length encoding means that it may not.

For example, you might want to encode Unicode characters. UTF-8 is one way to encode them: it uses 8 bits to store any of the values that are on the ASCII table. But there are way more than 256 characters altogether, so clearly some of them won't fit in 8 bits. So some characters must have more bits than that: this is a variable length encoding.

UTF-32 uses 4 bytes (32 bits) for each character, every time. It's a fixed length encoding.

Sometimes there are cases where a fixed length encoding can store some values in less space than a variable length encoding. For instance, if it were true that UTF-8 stored characters in some number of bits from 8 to 40, UTF-32 would store those longest ones in less space.

In the actual case for UTF-8/UTF-32, though, this isn't true; the longest UTF-8 characters are 32 bits. Pretty much the only way in which UTF-32 may be better is its predictability: given a stream of bytes, we know that the nth character starts at the (n-1)*4th byte. For UTF-8, you can't make any such kind of claim.

Conclusion: (Based on the observations):

We can conclude that we have learnt about Huffman Coding Algorithm

---

Outcome:

CO 3 : Implement Backtracking and Branch-and-bound 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. Coreman ,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.