Experiment No. : 4

Title: Implement Huffman Algorithm using Greedy approach

Batch: A2 **Roll No.: 16010421059** **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)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Explanation and Working of Variable Length Huffman Algorithm:
Construct a Huffman tree by using these nodes.

Value	A	B	C	D	E	F
Frequency	5	25	7	15	4	12

Solution:

Step 1: According to the Huffman coding we arrange all the elements (values) in ascending order of the frequencies.

Value	E	A	C	F	D	B
Frequency	4	5	7	12	15	25

Step 2: Insert first two elements which have smaller frequency.

Value	C	EA	F	D	B
Frequency	7	9	12	15	25

Step 3: Taking next smaller number and insert it at correct place.

Value	F	D	CEA	B
Frequency	12	15	16	25

Step 4: Next elements are F and D so we construct another subtree for F and

Value	CEA	B	FD
Frequency	16	25	27

Step 5: Taking next value having smaller frequency then add it with CEA and insert it at correct place.

Value	FD	CEAB
Frequency	27	41

Step 6: We have only two values hence we can combined by adding them.

Huffman Tree

Value	FDCEAB
Frequency	68

Now the list contains only one element i.e. FDCEAB having frequency 68 and this element (value) becomes the root of the Huffman tree.

Value	CEA	B	FD
Frequency	16	25	27

Step 5: Taking next value having smaller frequency then add it with CEA and insert it at correct place.

Value	FD	CEAB
Frequency	27	41

Step 6: We have only two values hence we can combined by adding them.

Huffman Tree

Value	FDCEAB
Frequency	68

Now the list contains only one element i.e. FDCEAB having frequency 68 and this element (value) becomes the root of the Huffman tree.

Derivation of Huffman Algorithm:

Time complexity Analysis

The time complexity for encoding each unique character based on its frequency is O(nlog n).

Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is O(nlog n).

Program(s) of Huffman Algorithm:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TREE_HT 100

struct MinHeapNode {

    char data;
```

```
    unsigned freq;
    struct MinHeapNode *left, *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(o) of Huffman Algorithm:

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Post Lab Questions: - Differentiate between Fixed length and Variable length Coding with suitable example.

In a fixed-length code each codeword has the same length. In a variable-length code codewords may have different lengths. Here are examples of fixed and variable legth codes for our problem (note that a fixed length code must have at least 3 bits per codeword).

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
a fixed-length	000	001	010	011	100	101
a variable-length	0	101	100	111	1101	1100

Conclusion: (Based on the observations):
This experiment helped in understanding and implementing Huffman Algorithm using Greedy approach.

Outcome: CO2 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. 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.