Experiment No. : 4

Title: Implement Huffman Algorithm using Greedy approach

Batch: A1 Roll No.: 16010422013
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:

Variable Length Huffman Algorithm is a data compression technique where variable-length codes are assigned to different symbols based on their frequencies. In short:

1. Symbols with higher frequencies get shorter codes; lower frequencies get longer codes.
2. It dynamically adjusts code lengths during encoding to optimize compression.
3. A tree structure is built, and codes are assigned based on the path from the root to each symbol.
4. Variable length helps achieve better compression for frequently occurring symbols.
5. It is widely used in lossless compression, like in image and video compression.
6. Efficient decoding is facilitated by ensuring no code is a prefix of another.

Derivation of Huffman Algorithm:

Time complexity Analysis

The Huffman Algorithm derivation involves:

1. Frequency Counting: Calculate symbol frequencies in the input data, typically done in O(n) time (linear complexity).

2. Priority Queue Construction: Create initial nodes in a priority queue based on symbol frequencies. The construction of the priority queue has a time complexity of O(n log n).

3. Huffman Tree Construction: Repeatedly remove and combine nodes with the lowest frequencies until a single node remains, forming the Huffman tree. The time complexity for building the Huffman tree is O(n log n).

4. Tree Traversal to Assign Codes: Traverse the Huffman tree to assign binary codes to each symbol. The traversal has a time complexity of O(n) as each node is visited once.

Overall, the time complexity of the Huffman Algorithm is dominated by constructing the priority queue and the Huffman tree, resulting in O(n log n) time complexity.

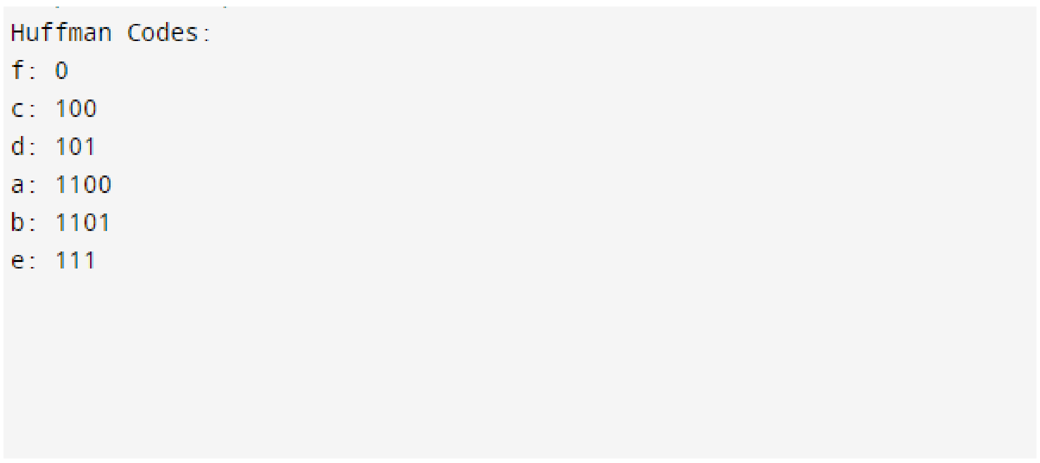
Program(s) of Huffman Algorithm:

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

struct Node {
    char data;
    unsigned frequency;
    struct Node *left, *right;
};

struct MinHeap {
    unsigned size;
    unsigned capacity;
    struct Node** array;
};

struct Node* newNode(char data, unsigned frequency) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->frequency = frequency;
    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 Node**)malloc(minHeap->capacity * sizeof(struct Node*));
    return minHeap;
}
```

```
void swapNode(struct Node** a, struct Node** b) {
   struct Node* 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]->frequency < minHeap->array[smallest]-
>frequency)
      smallest = left;

   if (right < minHeap->size && minHeap->array[right]->frequency < minHeap-
>array[smallest]->frequency)
      smallest = right;

   if (smallest != idx) {
      swapNode(&minHeap->array[smallest], &minHeap->array[idx]);
      minHeapify(minHeap, smallest);
   }
}

int isSizeOne(struct MinHeap* minHeap) {
   return (minHeap->size == 1);
}

struct Node* extractMin(struct MinHeap* minHeap) {
   struct Node* 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 Node* node) {
   ++minHeap->size;
   int i = minHeap->size - 1;
   while (i && node->frequency < minHeap->array[(i - 1) / 2]->frequency) {
      minHeap->array[i] = minHeap->array[(i - 1) / 2];
      i = (i - 1) / 2;
   }
   minHeap->array[i] = node;
}

struct Node* buildHuffmanTree(char data[], unsigned frequency[], int size) {
   struct Node *left, *right, *top;
   struct MinHeap* minHeap = createMinHeap(size);
```

```
    for (int i = 0; i < size; ++i)
        insertMinHeap(minHeap, newNode(data[i], frequency[i]));

    while (!isSizeOne(minHeap)) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        top = newNode('$', left->frequency + right->frequency);
        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    return extractMin(minHeap);
}

void printCodes(struct Node* 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 (!(root->left) && !(root->right)) {
        printf("%c: ", root->data);
        for (int i = 0; i < top; ++i) {
            printf("%d", arr[i]);
        }
        printf("\n");
    }
}

void huffmanEncode(char data[], unsigned frequency[], int size) {
    struct Node* root = buildHuffmanTree(data, frequency, size);
    int arr[100], top = 0;
    printf("Huffman Codes:\n");
    printCodes(root, arr, top);
}

int main() {
    char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    unsigned frequency[] = { 5, 9, 12, 13, 16, 45 };
    int size = sizeof(data) / sizeof(data[0]);
    huffmanEncode(data, frequency, size);

    return 0;
}
```

Output(o) of Huffman Algorithm:



```
Huffman Codes:
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.

Fixed-length coding and variable-length coding are contrasting methods employed in data compression. In fixed-length coding, a predetermined number of bits is assigned to each symbol, regardless of its frequency. This results in a uniform representation, exemplified by the ASCII encoding where each character is allocated a fixed 8 bits. On the other hand, variable-length coding adapts to the frequency of symbols, assigning shorter codes to more frequently occurring ones. Huffman coding serves as a prime example of variable-length coding, efficiently utilizing fewer bits for common symbols. While fixed-length coding is straightforward and suitable for scenarios where symbols have equal likelihood, variable-length coding offers higher compression ratios and adaptability to varying symbol frequencies. The trade-off lies in the simplicity of fixed-length coding versus the efficiency and adaptability of variable-length coding, with the choice dependent on specific use cases and requirements.

Conclusion: (Based on the observations): Hereby, we successfully implemented Huffman Tree and the output shows the binary encoding for each symbol.

Outcome: Implement Greedy and Dynamic Programming Problems.

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.