Design Analysis and Algorithm – Lab Work

Week 8

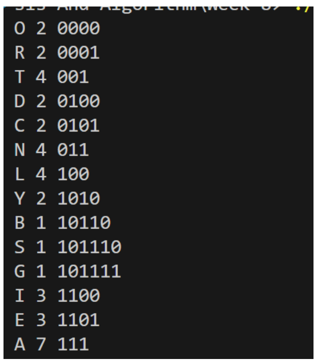
Question 1:Write a Program to perform Huffman Coding on "DATAANALYTICSANDINTELLIGENCELABORATORY":

Code:

```
#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(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

void swapMinHeapNode(struct MinHeapNode** a,
                     struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
```

```c
    *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);
    }
}

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;
}

struct MinHeap* buildMinHeap(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;

    for (int i = (size - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);

    return minHeap;
}

int isLeaf(struct MinHeapNode* root) {
    return !(root->left) && !(root->right);
}

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 %d ", root->data, root->freq);
        for (int i = 0; i < top; ++i)
            printf("%d", arr[i]);
        printf("\n");
    }
}

void HuffmanCodes(char data[],
                  int freq[],
                  int size) {

    struct MinHeapNode *left, *right, *top;

    struct MinHeap* minHeap =
        buildMinHeap(data, freq, size);

    while (minHeap->size != 1) {

        left = extractMin(minHeap);
        right = extractMin(minHeap);

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

        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    int arr[MAX_TREE_HT], topIndex = 0;
    printCodes(extractMin(minHeap), arr, topIndex);
}

int main() {

    char arr[] = {'A','D','T','N','L','I','E','C','S','G','B','O','R','Y'};
    int freq[] = {7,2,4,4,4,3,3,2,1,1,1,2,2,2};
    int size = sizeof(arr)/sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;
}
```

Output:



```
sis And Algorithm\Week 6> ./
O 2 0000
R 2 0001
T 4 001
D 2 0100
C 2 0101
N 4 011
L 4 100
Y 2 1010
B 1 10110
S 1 101110
G 1 101111
I 3 1100
E 3 1101
A 7 111
```

Time Complexity

The time complexity of Huffman Coding mainly depends on the number of **unique characters (n)**. First, we build a min-heap using all unique characters, which takes **O(n)** time. Then, we repeatedly extract the two minimum-frequency nodes and insert their combined node back into the heap. Each extraction and insertion operation takes **O(log n)** time because the heap must maintain its ordering property. Since this merging process happens **(n − 1)** times, the total time required for tree construction becomes **O(n log n)**. Finally, printing the Huffman codes requires a traversal of the tree, which takes **O(n)** time. Therefore, the overall time complexity of Huffman Coding is **O(n log n)**.

Space Complexity

The space complexity is determined by the storage used for the min-heap and the Huffman tree. The min-heap stores **n nodes**, requiring **O(n)** space. The Huffman tree itself contains **2n − 1 nodes** (including internal and leaf nodes), which is also proportional to **O(n)**. Additionally, the recursion stack used during tree traversal (while printing codes) takes at most **O(n)** space in the worst case (when the tree becomes skewed). Since all these space requirements grow linearly with the number of unique characters, the overall space complexity of Huffman Coding is **O(n)**.