Huffman Codes

Group 18

Date:2020-04-30

CONTENT

Chapter 1 Introduction

1.1 Background

Huffman coding is a type of variable word length coding (VLC). Huffman proposed the coding method in 1952. This method completely constructs the codeword with the shortest average length of different prexes based on the occurrence probability of characters. It is sometimes called the best coding, And it is generally called Huffman coding. The following cites a theorem, which ensures that the code length is allocated according to the occurrence probability of characters, so that the average code length is the shortest.

There are diverse variable word length coding for one array of characters, while obviously not all of them are correct. Only some specic VLC can be called as correct Huffman codes and
the detailed strategies to judge them are described in this report.

1.2 Problem Description

In this project, for a specic array of characters with dened frequency, we need to find whether the given codes are correct or not. Note that the optimal solution is not necessarily generated by Huffman algorithm. Any prex code with code length being optimal is considered correct. So we need to generate a variety of codes strategies for our tests and use our program to check their correctness.

Chapter 2 Algorithm Specification

2.1 Sketch and Data Structure

Here is our program sketch:

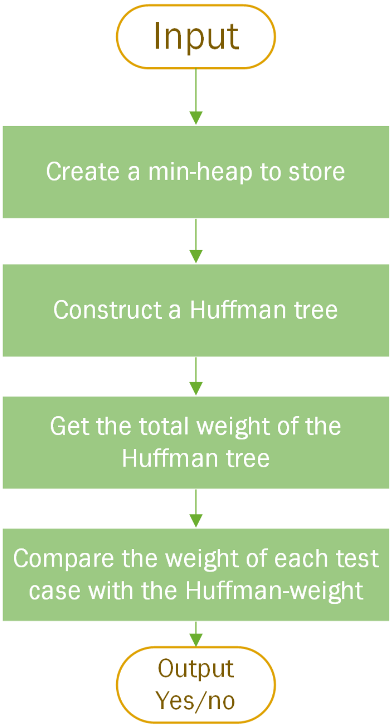

Figure1: Program Sketch

2.2 Function Description

There are several functions used in the program, complementing our main algorithm. We hereby list some functions which are acting as auxiliary in the program to make our report more completed.

2.2.1 Minimum-Heap

2.2.1.1 Insert

The sketch of the function `heap_insert` is as following:

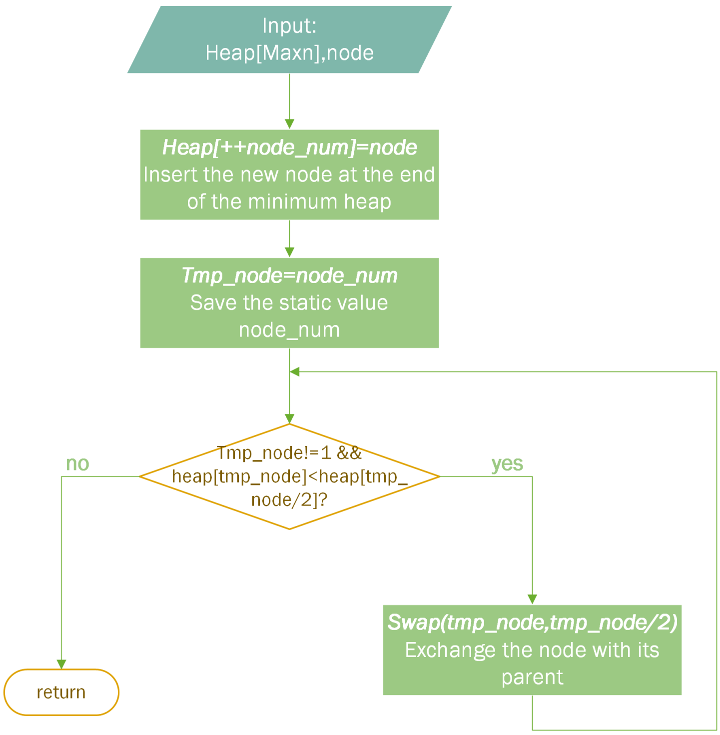

Figure2: Program Sketch of min-heap insertion

The pseudocode to insert elements in to a minimum heap is as following:

```
function heap_insert(ptr_huffman_node node)
    heap[++node_num]<-node
    this<-node_num
    while (this != 1 and heap[this]->frequency < heap[this/2]->frequency)
        do swap(this, this/2);
        this /= 2;
    end
return
```

For example, if we want to insert the element `0` into the minimum tree as following:

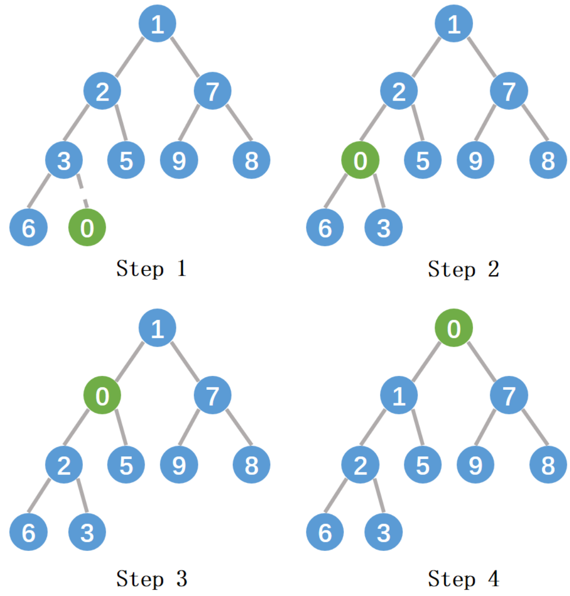

Figure3: Sample of inserting in min-heap

2.2.1.1 Delete_Min

The sketch of the function `delete_min` is as following:

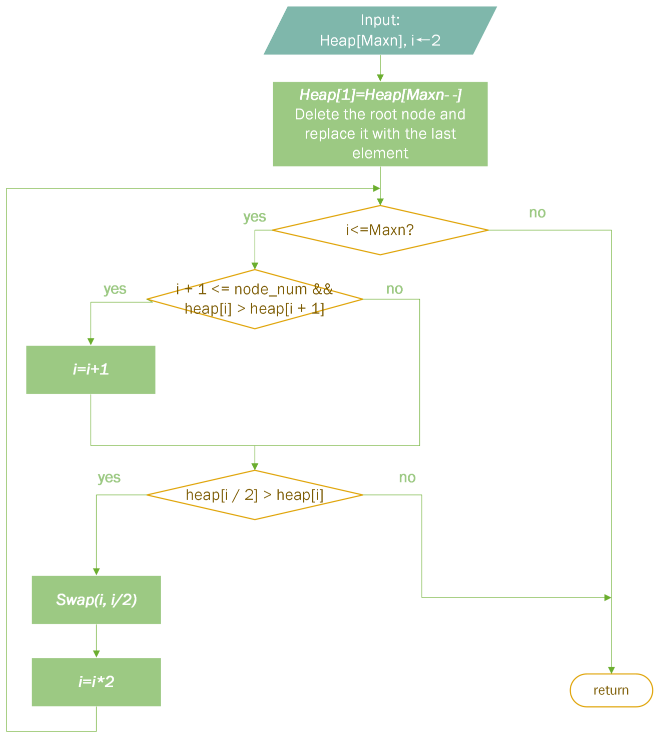

Figure4: Program Sketch of min-heap deletion

The pseudocode to delete the minimum element in a minimum heap is as following:

```
function heap_delete()
    heap[1]<-heap[node_num--]
    for i<-2 to node_num
        do
        if(i + 1 <= node_num && heap[i] > heap[i + 1])
            do i<-i+1
        end
        if(heap[i / 2] > heap[i])
            do swap(i,i/2)
        else
            do break
        end
        i<-i*2
    end
return
```

For example, if we want to delete the minimum element `0` in the minimum tree as following:

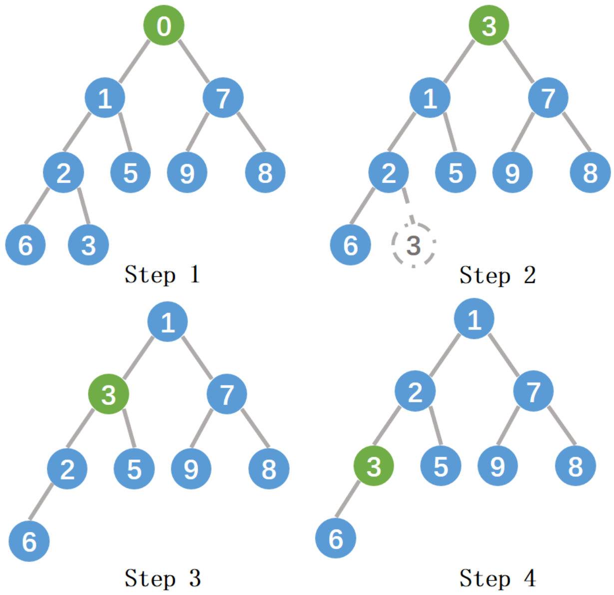

Figure5: Sample of deleting the min element

2.2.2 Construct a Huffman tree

The basic idea of Huffman coding is to construct a Huffman tree with the use frequency as the weight, and then use Huffman tree to code the characters. To construct a Huffman tree, the character to be encoded is taken as the leaf node, and the frequency of using the character in the file is taken as the weight of the leaf node. In a bottom-up way, a tree is constructed after $n - 1$ `merging` operations. The core idea is that the larger the weight, the closer the leaf is to the root.

The form of Huffman structure is like:

```c
typedef struct huffman_node* ptr_huffman_node ;
struct huffman_node {
    char chr;                   // The letter of A-Z,a-z,_
    int frequency;              // The number of this letter's appearances
    ptr_huffman_node left;      // The pointer to the left child
    ptr_huffman_node right;     // The pointer to the right child
};
```

The sketch of the function `construct_huffman` is as following:

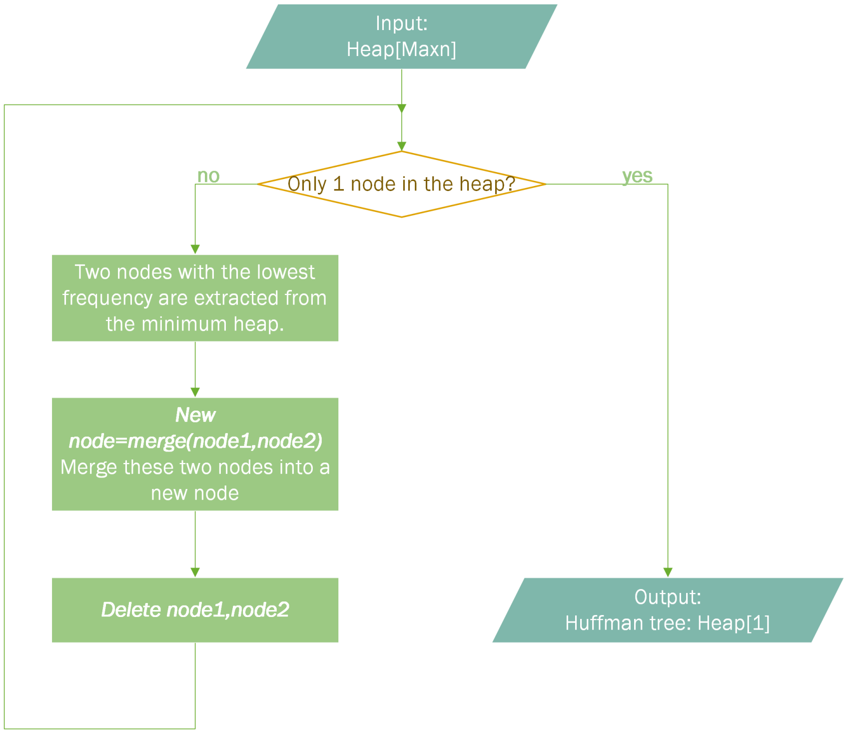

Figure6: Program Sketch of Huffman tree construction

The pseudocode to construct a Huffman tree is as following:

```
function construct_huffman()
    while (there are more than 1 node in heap)
        do
        node1<-Heap[1]
        delete_min()
        node2<-Heap[1]
        delete_min()
        insert(merge(node1,node2))
    end
return
```

For example, suppose that the input min-heap is as following:

Heap	**1**	**2**	**3**	**4**	**5**
char	A	B	C	D	E
frequency	1	2	4	3	7

Obviously, the two nodes with the least frequency are A and B. So we merge them and insert the new node into the min-heap.

Heap	**1**	**2**	**3**	**4**
char	D	A+B	C	E
frequency	3	3	4	7

Then, the two nodes with the least frequency are A+B and D. So we merge them and insert the new node into the min-heap.

Heap	**1**	**2**	**3**
char	C	E	D+(A+B)
frequency	4	7	6

Now, the two nodes with the least frequency are C and D+(A+B). So we merge them and insert the new node into the min-heap.

Heap	**1**	**2**
char	E	C+(D+(A+B))
frequency	7	10

Finally, merge the remain two nodes, we can get the Huffman tree as following:

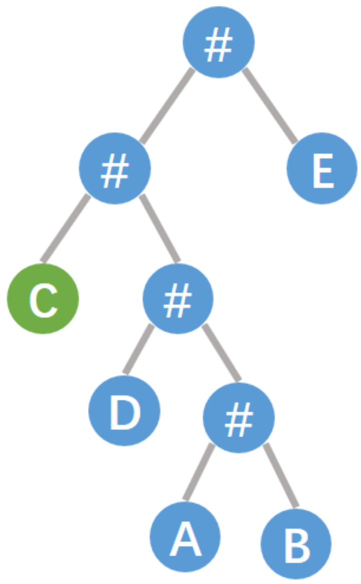

Figure6: Sample of Huffman tree

2.2.3 Compare

To judge whether the student's submission is correct, we just need to compare the weight of Huffman tree we construct before with the weight of the solution that student submits.

The pseudocode to calculate the weight of Huffman tree is as following:

```
function calculate_weight(ptr_huffman_node node, int level)
    if (node == NULL) return;
    calculate_weight(node->left,level + 1)
    calculate_weight(node->right,level + 1)
    if (node->chr != '#')
        do weight += node->frequency * level
        // Since I use '#' to mark it non-leaf
    end
end
```

The weight of the sample showed in `Figure 6` is 36.
If one of the solutions submitted is as following:

char	**A**	**B**	**C**	**D**	**E**
frequency	1	2	4	3	7
code	0110	0111	010	00	1

The weight of it can be calculated as:

$$weight = 1 \times 4 + 2 \times 4 + 4 \times 3 + 3 \times 2 + 7 \times 1 = 36$$

So the weight of the submission is equal to the Huffman tree. In order to judge the correctness, then we also need to check whether there is an element's code is the foreword or subsequence of another element.

```
function check_prefix(int test_groups_num)
    for i<-0 to test_groups_num step 1
        do for j<-0 to test_groups_num step1
            if i=j continue
            result<-strstr(test_table[i],test_table[j])
            if result==NULL
                do continue
            if result=test_table[i]
                return 0
        end
    end
    return 1
end
```

Chapter 3 Testing Results

We partly use the c++ testing case generating program in Appendix II to generate our test case.

3.1 Test case 1

Purpose of this case

Juxtaposition, multi branches; length wrong; length right but prefix wrong; only uppercase characters

Input(sample)

```
7
A 1 B 1 C 1 D 3 E 3 F 6 G 6
4
A 00000
B 00001
C 0001
D 001
E 01
F 10
G 11
A 01010
B 01011
C 0100
D 011
E 10
F 11
G 00
A 000
B 001
C 010
D 011
E 100
F 101
G 110
A 00000
B 00001
C 0001
D 001
E 00
F 10
G 11
```

Output

```
Yes
Yes
No
No
```

Result

Correct

3.2 Test case 2

Purpose of this case

Lowercase letters with 0, 1 reversed and 2 points swapping; 2 points overlap

Input

```
3
a 13 b 3 c 78
1
a 01
b 00
c 1
```

Output

```
Yes
```

Result

Correct

3.3 Test case 3

Purpose of this case

Unequal length of codes but all right; equal length but wrong prefix; length of codes greater than N

Input

```
4
A 4 B 4 C 6 D 6
1
A 111
B 1
C 0
D 001
```

Output

```
No
```

Result

Correct

3.4 Test case 4

Purpose of this case

Maximum N = 63 & M = 1000

Input

Refer to `max_in.txt` in the `code` folder.

Output

Refer to `max_out.txt` in the `code` folder.

Result

Correct

3.5 Test case 5

Purpose of this case

Minimum N = 2 & M = 1

Input

```
2
A 0 B 1
1
A 1
B 0
```

Output

```
Yes
```

Result

Correct

3.6 Test case 6

Purpose of this case

The number of characters to be encoded is double, while the length to be submitted is equal.

Input

```
4
A 3 B 5 C 6 D 7
1
A 00
B 01
C 10
D 11
```

Output

```
Yes
```

Result

Correct

3.7 Test case 7

Purpose of this case

Not Huffman coded, but correct

Input

```
3
A 0 B 0 C 8
```

Output

```
Yes
```

Result

Correct

Chapter 4 Analysis and Comments

4.1 Analysis on Time Complexity

We can divide the program into three parts.

The first part is the input part. There is only one loop in this function, whose time complexity is **O(N)**.

The second part is to calculate the total weight of the huffman tree. When analyzing time complexity, Function ***calculate_ weight*** is mainly about a two-layer loop, which is used to find the two smallest in the array of frequency, sum them up to get a new number and delete those two original numbers. Note that we use minimum heap to implement this design, so the time complexity is reduced to **O(NlogN)**.

The third part is to judge which of those M sets of codes is correct. It's also a two-layer loop. The outer loop repeats M times while the inner loop N times, so we can conclude that the time complexity of this part is **O(M*N)** as we must traverse all M sets of codes and each of them consists of N lines .

In conclusion, when M is much greater than N, the time complexity of this program is **O(M*N).** If not, then it's **O(NlogN)**.

4.2 Analysis on Space Complexity

Although a lot of memory is allocated dynamically in the construction of Huffman tree, most of this memory is only a linear function of N. For example, we use array to store frequencies and characters, both of which use N spaces. In conclusion, the total space complexity is **O(N)**.

4.3 Comments

The basic idea of Huffman coding is to construct a Huffman tree with the use frequency as the weight, and then use Huffman tree to code the characters. To construct a Huffman tree, the character to be encoded is taken as the leaf node, and the frequency of using the character in the file is taken as the weight of the leaf node.

In a bottom-up way, a tree is constructed after `n-1`"merging" operations. The core idea is that the larger the weight, the closer the leaf is to the root. The greedy strategy adopted by Huffman algorithm is to take two trees without parents and with the least weight from the set of trees as left and right subtrees, which gives us a further understanding of greedy algorithm.

Appendix

Appendix I Source Code in C

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

#define MAX_CODING_LENGTH 65    // Slightly larger than 63, which is the maximal length of input
#define ASCII 128               // Use ascii code for indexing, reducing time

// This structure describes a huffman node in the huffman tree
typedef struct huffman_node* ptr_huffman_node ;
struct huffman_node {
    char chr;                   // The letter of A-Z,a-z,_
    int frequency;              // The number of this letter's appearances
    ptr_huffman_node left;      // The pointer to the left child
    ptr_huffman_node right;     // The pointer to the right child
};

int weight = 0;                                             // The weight of coding using huffman tree
int node_num = 0;                                           // The number of nodes in the heap
int frequency[ASCII];                                       // Helps in finding the frequency quickly
ptr_huffman_node heap[MAX_CODING_LENGTH] = {NULL};          // The heap that stores huffman nodes
char test_table[MAX_CODING_LENGTH][MAX_CODING_LENGTH];      // The input case in each test

/**
 * @Author      : C01dkit
 * @Date        : 21:10   2020/4/25
 * @Param       : node
 * @return      : void
 * @Output      : none
 * @Description : This function will insert a pointer in the heap
 */
void heap_insert(ptr_huffman_node node);

/**
 * @Author      : C01dkit
 * @Date        : 21:35   2020/4/25
 * @Param       : smaller_index   -------- The index of the node whose frequency is smaller in the heap
 *                larger_index    -------- The index of the node whose frequency is larger in the heap
 * @return      : void
 * @Output      : node
 * @Description : This function will swap two pointers in the heap using their index.
 */
void swap(int smaller_index, int larger_index);

/**
 * @Author      : C01dkit
 * @Date        : 21:41   2020/4/25
 * @Param       : none
 * @return      : void
 * @Output      : none
 * @Description : This function will delete heap[1] and return nothing
 */
void heap_delete();

/**
 * @Author      : C01dkit
 * @Date        : 22:11   2020/4/25
 * @Param       : node    -------- The pointer to the root of huffman tree
 *                level   -------- Current depth of the tree, start at 0
 * @return      : void
 * @Output      : none
 * @Description : This function will do a post-order traversal and calculate total weight of the huffman tree
 */
void calculate_weight(ptr_huffman_node node,int level);

/**
 * @Author      : C01dkit
 * @Date        : 22:39   2020/4/25
 * @Param       : test_groups_num     -------- The number of strings going to check
 * @return      : int
 * @Output      : none
 * @Description : This function will check current group of test.
 *                If it obeys prefix rule, then the function will return 1, otherwise 0.
```

```
 */
int check_prefix(int test_groups_num);

int main () {
    int N;
    ptr_huffman_node temp;
    scanf("%d",&N);
    getchar();
    // This loop will restore the initial data and construct a heap
    for (int i = 0; i < N; i++) {
        temp = (ptr_huffman_node)malloc(sizeof(struct huffman_node));
        if (i == 0) scanf("%c %d",&temp->chr,&temp->frequency);
        else scanf(" %c %d",&temp->chr,&temp->frequency);
        temp->left = NULL;
        temp->right = NULL;
        frequency[(int)temp->chr] = temp->frequency;
        heap_insert(temp);
    }
    // This loop will construct a huffman tree using the heap
    while (node_num > 1) {  // If the heap only contains one node, then it is the root of the huffman tree
        temp = (ptr_huffman_node)malloc(sizeof(struct huffman_node));
        temp->chr = '#';    // Marking this node a non-leaf node
        temp->frequency = heap[1]->frequency;
        temp->left = heap[1];
        heap_delete();      // Pop one min-node
        temp->frequency += heap[1]->frequency;
        temp->right = heap[1];
        heap_delete();      // Pop another min-node
        // Since the minimal 2 nodes have been popped and connected to the temp node
        // Then push the node back into the heap for the next loop
        heap_insert(temp);
    }
    calculate_weight(heap[1],0);    // Calculate the weight using huffman algorithm
    int M, test_weight;
    char test_char,test_string[MAX_CODING_LENGTH];
    scanf("%d",&M);
    // This loop will read M test cases, the answer will be given right away
    for (int i = 0; i < M; i++) {
        test_weight = 0;
        for (int j = 0; j < N; j++) {
            getchar();
            scanf("%c %s",&test_char,test_string);
            strcpy(test_table[j], test_string);    // This makes checking prefix more quickly
            test_weight += strlen(test_string) * frequency[(int)test_char];    // Calculate the weight of this case
        }
        // After input, there are two test.

        // If test result cost more, then it fails immediately.
        if (test_weight != weight) printf("No\n");

        // Otherwise, check if one string starts at the beginning of any other one
        else if (check_prefix(N) == 0) printf("No\n");

        // If this case is able to pass the previous two tests, then it is OK.
        else printf("Yes\n");
    }
    return 0;
}

int check_prefix(int test_groups_num) {
    char* str_result;
    for (int i = 0; i < test_groups_num; i++) {
        for (int j = 0; j < test_groups_num; j++) {
            if (j == i) continue;   // No need to test itself
            str_result = strstr(test_table[i], test_table[j]);  // Check1 : substring
            if (str_result == NULL) continue;  // If it is not its substring, then continue
            if (test_table[i] == str_result) { // Check2 : prefix
                return 0;  // If it starts from the same address as the other one, then it doesn't obey prefix rule
            }
        }
    }
    return 1;
}

void calculate_weight(ptr_huffman_node node, int level) {
    if (node == NULL) return;
    calculate_weight(node->left,level + 1);
    calculate_weight(node->right,level + 1);
    if (node->chr != '#') weight += node->frequency * level;    // Since I use '#' to mark it non-leaf
}

void heap_delete() {
    heap[1] = heap[node_num--];  // Move the last one to the top, node_num counts down
```

```
    // This loop will check the property of min-heap from top
    for (int i = 2; i <= node_num; i *= 2) {
        if (i + 1 <= node_num && heap[i]->frequency > heap[i + 1]->frequency) i++;
        if (heap[i / 2]->frequency > heap[i]->frequency) swap(i, i / 2);
        else break;
    }
}

void heap_insert(ptr_huffman_node node) {
    heap[++node_num] = node;   // Insert one node to the last, node_num counts up
    int this = node_num;
    // This loop will check the property of min-heap from bottom
    while (this != 1 && heap[this]->frequency < heap[this/2]->frequency) {
        swap(this, this/2);
        this /= 2;
    }
}

void swap(int smaller_index, int larger_index) {
    // Simply swap two nodes
    ptr_huffman_node temp;
    temp = heap[smaller_index];
    heap[smaller_index] = heap[larger_index];
    heap[larger_index] = temp;
}
```

Appendix II Testing Set Generating Script in C++

```
/*
Test cases generator
Input: N & M
Output:out.txt
*/

#include <bits/stdc++.h>
using namespace std;

#define MAXN 63
#define MAXM 1000

std::map<int, char> Code_dict;
char sym[MAXN];
int freq[MAXN];

int Binary(int i)
{
    if (i < 2) return i;
    else
    {
        return (Binary(i / 2) * 10 + i % 2);
    }
}

class Node {
public:
    char c;
    int frequency;
    Node* left;
    Node* right;
    Node(char _c, int f, Node* l = NULL, Node* r = NULL)
        :c(_c), frequency(f), left(l), right(r) { }
    bool operator<(const Node& node) const { // reload
        return frequency > node.frequency;
    }
};

void initNode(priority_queue<Node>& q, int N)
{
    for (int i = 0; i < N; i++)
    {
        Node node(sym[i], freq[i]);
        q.push(node);
    }
}

//construct huffmanTree
void huffmanTree(priority_queue<Node>& q) {
    while (q.size() != 1) {
        Node* left = new Node(q.top()); q.pop();
        Node* right = new Node(q.top()); q.pop();
        Node node('R', left->frequency + right->frequency, left, right);
        q.push(node);
    }
}

// print huffmanCode
void huffmanCode(Node* root, string& prefix, map<char, string>& result) {
    string m_prefix = prefix;
    if (root->left == NULL) return;
    //left subtree
    prefix += "0";
    if (root->left->left == NULL) result[root->left->c] = prefix;
    else huffmanCode(root->left, prefix, result);
    prefix = m_prefix;
    //right subtree
    prefix += "1";
    if (root->right->right == NULL) result[root->right->c] = prefix;
    else huffmanCode(root->right, prefix, result);
}

void testResult(map<char, string> result) {
    //recur
    map<char, string>::const_iterator it = result.begin();
    while (it != result.end())
    {
        cout << it->first << ' ' << it->second << endl;
```

```
        ++it;
    }
}

int main()
{
    priority_queue<Node> q;
    int N, M;
    //initialize
    cout << "Please input the number of N and M:";
    cin >> N >> M;

    streambuf* coutBuf = cout.rdbuf();
    ofstream of("out.txt");
    streambuf* fileBuf = of.rdbuf();
    cout.rdbuf(fileBuf);

    cout << N << endl;
    //i==key, c==Code_dict[i]
    int c = 48;
    int* Flag = (int*)malloc(sizeof(int) * 63);
    for (int i = 0; i < 10 + 26 * 2 + 1; i++, c++)
    {
        if (i == 10) c = 65;
        if (i == 36) c = 95;
        if (i == 37) c = 97;
        Code_dict[i] = c;
        //0 48 A 65 _ 95 a 97
        Flag[i] = 0;
    }
    //random char and frequency
    srand((unsigned)(time(NULL)));
    for (int i = 0; i < N; i++)
    {
        int a = rand() % 63;
        if (!Flag[a])
        {
            sym[i] = Code_dict[a];
            int b = rand() % 1000 + 1;
            freq[i] = b;
            cout << sym[i] << ' ' << freq[i] << ' ';
            Flag[a] = 1;
        }
        else i--;
    }
    free(Flag);
    cout << endl;
    cout << M << endl;

    //calculate the correct codes
    initNode(q, N);
    huffmanTree(q);

    Node root = q.top();
    string prefix = "";
    map<char, string> result;
    huffmanCode(&root, prefix, result);
    //test result
    for (int j = 1; j <= M / 2; j++) testResult(result);

    //calculate bits
    int num = 1, k = N;
    while ((k = k / 2) != 0)
    {
        num++;
    }
    //print codes
    for (int j = M / 2 + 1; j <= M; j++)
    {
        for (int i = 1; i <= N; i++)
        {
            int code = 0;
            code = Binary(i);
            cout << sym[i - 1] << " ";
            if (j == M && i == N)
            {
                cout.width(num);
                cout.fill('0');
                cout << code;
            }
            else
            {
                cout.width(num);
                cout.fill('0');
                cout << code << endl;
            }

```

```
        }
    }

    of.flush();
    of.close();
    cout.rdbuf(coutBuf);
    return 0;
}

```

Appendix III Declaration and Signatures

Declaration

We hereby declare that all the work done in this project titled "***Huffman Codes***" is of our independent effort as a group.

Signatures