Huffman Codes

Authors:XXX

Date: 2022/05/04

Chapter 1: Introduction

Huffman code is an excellent way to encode, because this encoding method takes both the frequency of character occurrence and the frequency of module characters into account , so that the final overall encoding length is very short. And the encoding is not ambiguous, at the same time, the encoding process is also very simple - just building a binary tree based on each character's frequencies.

However, through Huffman's encoding process, it is not difficult to find that when some characters or combined character blocks are encountered with equal frequencies, the Huffman encoding process cannot ensure that the final result is unique, that is, the subtrees interchange positions of the same frequency are also in line with the Huffman encoding method, and the result obtained is also a Huffman code. At the same time, there will be some codes that are not Huffman codes, but still meet the requirements of the prefix code, that is, there is no ambiguity, and the overall code length is also very reasonable, which is also worth affirming, and can be considered as "right" codes.

Going back to the topic, given a list of characters with frequencies corresponding to each character, and then listing a series of encodings corresponding to those characters, all you have to do is to distinguish whether those codes are "right" codes, that is, it may be obtained by correct Huffman encoding way, but may not be, but the judgment criteria are only the following two rules:

> Take the Huffman encoding length as the standard, and meet the overall encoding length optimization.
>
> Meet the prefix code requirements, no ambiguity.

For the input forms:

- The first line gives the number of characters that need to be encoded.
- The second line gives each character's frequencies in character + frequencies.
- The third line is the number M of segments encoding the whole.
- Subsequently, there is an M-segment encoding of each character, and the format is the character to be encoded + the corresponding encoding. Make sure that the order should be the **same** with the second line given.

The output needs to detect the M-segment codes to determine whether each segment is a "right" code, if is, then output "Yes", otherwise output "No".

Chapter 2: Data Structure / Algorithm Specification

Data Structure

The Build of the Tree

Minheap

The most common way to build a Huffman Tree is through minimal heap. Because of the featureof minimal heap, it is very easy to get the smallest element (in this case, the least weighted node). And we can use this feature to easily merge the two smallest nodes.

The struct of the minheap:

```
typedef struct MinHeap
{
    WeightedTree** data;
    int size;
    int capacity;
} MinHeap;
```

Other relevant operation of the minheap:

```
void PercDown(MinHeap* heap, int p);// Adjust pth node in heap to deeper to make it a MinHeap
void Insert(MinHeap* heap, WeightedTree* tree);// Insert a WeightedTree to heap
WeightedTree* DeleteMin(MinHeap* heap)// Delete the min node in heap
void FreeHeap(MinHeap* heap);// Release memory
```

The Build of the Tree

For **weighted tree**:

We build the weighted tree according to the code given.

Firstly, we set the weight to INF to denote that the path to it is not a code. And then the binary code word for a character is represented by a simple path from the root node to the leaf of the character, where 0 means "turn to the left child" and 1 means "turn to the right child"

For **Huffman tree**:

We find the two nodes with the least weight by DeleteMin twice and take them as the two children of the new node. The weight of the new node is equal to the sum of the weights of the two children.

Finally, the new node is inserted into the minimum heap.

The main building function of the trees:

```
// Build Tree according to the code given
WeightedTree* BuildWeightedTree(char code[MAX_N][MAX_LEN], int N, int* f);
// Use Huffman algorithm to build huffman tree
WeightedTree* BuildHuffmanTree(MinHeap* heap)
```

Other relevant operation of the tree:

```
void FreeWeightedTree(WeightedTree* tree)// Release memory iteratively
```

The Queue

The function of the queue is to store the node and for computing WPL in level order.

The struct of the queue and queue node:

```
typedef struct Queue
{
    QueueNode* head;
    QueueNode* tail;
}Queue;

typedef struct QueueNode
{
    WeightedTree* data;
    struct QueueNode* next;
} QueueNode;
```

Other relevant operation of the queue:

```
Queue* CreateQueue();// Create an empty queue
void Enqueue(Queue* queue, WeightedTree* tree);// Enqueue WeightedTree
WeightedTree* Dequeue(Queue* queue);// Dequeue WeightedTree
void FreeQueue(Queue* queue);// Release memory
```

Algorithm

First, use standard huffman algorithm to build the tree and get the optimal code length.

The key step of the check function is to create the weight tree according to the given codes, and then compare it with the standard Huffman tree. This can easily check whether the codes are prefix codes. And as for checking the length, we do this at the same time, that is when checking prefix codes, we also calculate standard length and the given codes length by using the fomular:

$$Len = \sum_{i=0}^{N-1} strlen(code[i]) * f(i)$$

And we just compare this two Len to deside whether the codes meet the length requirement of the "good" codes.

The Calculation of WPL

We use iterative and recursive approachs to calculate WPL respectively.

Recursive version

If it is not a leaf node, the sum of WPL of its left and right nodes is recursively obtained.

If it is a leaf node, the product of depth and weight is returned.

```
int GetWPL(Weighted Tree* tree, int depth)
{
    if (!tree->left && !tree->right)
        return depth * tree->weight;
    else
        return GetWPL(tree->left, depth + 1) + GetWPL(tree->right, depth + 1);
}
```

Iterative version

In order to achieve iterative method, we use queue to traverse the weightedtree in level order. Diffrent from common level traverse, we have to obtain the depth of the current node when traversing. To implement this, we introduce a level node representing the end of one level when doing level traversal whose depth is set to -1. Every time when we do dequeue operation and get the level node, it means we have reached the end of this level, so the depth should plus one. And then enqueue the level node back since the node of the next level has all been enqueued to the queue. What should be paid attention to is that when there is only one level node in the queue, it means the tree has been traversed, so just dequeue it withou enqueue it again.

Check the Prefix Code

Use queue to traverse the tree in level order. We store the test code in the queue and compare them in pairs. When the first different code element is found and one of the codes has been iterated, the prefix code is not satisfied.

If there is still node, we check the prefix code the compute the WPL. If the wpl of this tree is the same with optimalWpl got by Huffman algorithm, this should be an optimal encoding way as we have set wpl of code not satisfying prefix rule to INF.

Chapter 3: Testing Results

- Case 1: The standard sample
    - Input: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

- Expected Result:

  Yes

  Yes

  No

  No

- Actual Result:

  Iterative version:

```
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
Yes
A 01010
B 01011
C 0100
D 011
E 10
F 11
G 00
Yes
A 000
B 001
C 010
D 011
E 100
F 101
G 110
No
A 00000
B 00001
C 0001
D 001
E 00
F 10
G 11
No
time=5.185000
```

Average time：4.5638（Each test was performed five times and averaged, same as below）

Recursive version:

```
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
Yes
A 01010
B 01011
C 0100
D 011
E 10
F 11
G 00
Yes
A 000
B 001
C 010
D 011
E 100
F 101
G 110
No
A 00000
B 00001
C 0001
D 001
E 00
F 10
G 11
No
time=2.360000
```

Average time：2.3996

- Status: Pass

- Case 2: case that each character is not a prefix encoding for any other character
  - Input:

    5

    A 3 B 5 C 3 D 1 E 1

    1

    A 00

    B 01

    C 10

    D 110

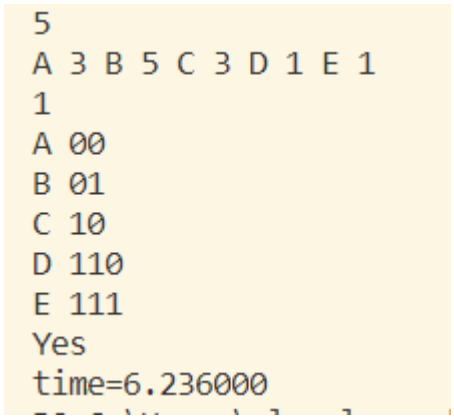
    E 111

- Expected Result:

  Yes

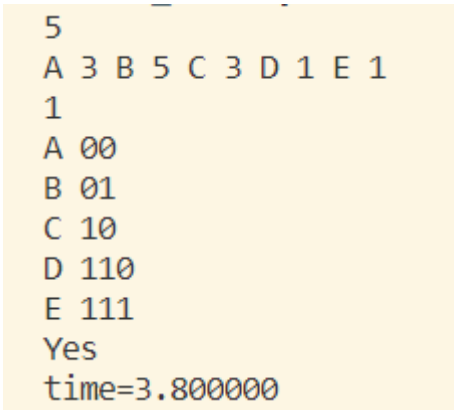
- Actual Result:

  Iterative version:

  Average time：3.6543

  Recursive version:

  Average time：2.6954

- Status: Pass

- Case 3: case that unsatisfied the requirement of prefix
  - Input:

    5

    A 3 B 5 C 3 D 1 E 1

    1

    A 00

    B 01

    C 11

    D 011

    E 100

  - Expected Result:

    No

  - Actual Result:

    Iterative version:

```
5
A 3 B 5 C 3 D 1 E 1
1
A 00
B 01
C 11
D 011
E 100
No
time=2.777000
```

Average time：2.608

Recursive version:

```
5
A 3 B 5 C 3 D 1 E 1
1
A 00
B 01
C 11
D 011
E 100
No
time=2.864000
```

Average time：2.677

- Status: Pass

The comparison of the time performance

The overhead of recursive calls is mainly in stack maintenance. The number of parameters less than or equal to 6 in a 64-bit program is passed through the register. It is usually necessary to reserve the original value through the stack before changing the value of the register. Thus, only one additional register is required to reserve through the stack (rbp).

```
0x55555555589a <_Z6GetWPLP12WeightedTreei+4>          push   rbp
0x55555555589b <_Z6GetWPLP12WeightedTreei+5>          mov    rbp,rsp
0x55555555589e <_Z6GetWPLP12WeightedTreei+8>          push   rbx
0x55555555589f <_Z6GetWPLP12WeightedTreei+9>          sub    rsp,0x18
0x5555555558a3 <_Z6GetWPLP12WeightedTreei+13>         mov    QWORD PTR [rbp-0x18],rdi
0x5555555558a7 <_Z6GetWPLP12WeightedTreei+17>         mov    DWORD PTR [rbp-0x1c],esi
```

In addition, the stack is maintained only function return address, stack frame pointer and local variable definition. Meanwhile the recursive algorithm does not involve repeated calculation and the overhead is not large in the case of small tree depth. In this case, the largest N is 63, so the recursive algorithm is not very expensive.

However, the iterative algorithm uses queues, and the maintenance of queues is more expensive than the maintenance of the recursive stack when N is small, so its performance is worse than that of the recursive algorithm.

While, because the number N is not big enough, the difference in speed of two methods is not very obvious.

Chapter 4: Analysis and Comments

Analysis

Time Complexity: O(W), where W is the number of the whole bits that the codes have.

For each block of codes that need to be checked, we need to build a weight tree according to the given codes, and then we check the weight tree with our standard tree, that is Huffman tree. If the weight tree meet all requirements, we said the codes is "right". When get the codes, we need time complexity O(N), where N is the number of characters. And in building weight tree, we need to consider each bit of each code. That is O(N*A), where A is the average length of the codes in a block. Then we get the third process, that is checking the weight tree with the standard Huffman tree. We use a queue to check, and the number of loop is the number of nodes of the weight tree, which is O(N*A). Then we have checked one block. As there are totally M blocks, So the total time complexity is O(M*N*A), while the A of each block is different, but N*A is the total bits of the codes in one block, then M*N*A is the total bits of the codes of all the blocks, let it be W, then the time complexity is O(W), where W is the number of the whole bits that the codes have.

Space Complexity: O(MAX_N*MAX_LEN), where MAX_N is the biggest number of characters, and MAX_LEN is the biggest length of the code of character.

The code[MAX_N][MAX_LEN] is used to store the bits of one block. And for another block, we just used the array "code" again. And for the space of weight tree and queue as well as the standard tree, they are all N*A, where N is the number of characters, and A is the average length of the code of character. also there are N bits to store the frequencies. So the whole space complexity is MAX_N*MAX_LEN + 3*N*A + N = O(MAX_N*MAX_LEN), where MAX_N is the biggest number of characters, and MAX_LEN is the biggest length of the code of character.

Comments

Cases with obvious characteristics that can not be optimal code should be pruned before we actually calculate its WPL, which may save us some time doing meaningless computing.

Appendix: Source Code

- Recursion Version

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

#define MAX_N 63
#define MAX_LEN 64
#define INF 1001


typedef struct WeightedTree
{
    int weight;
    struct WeightedTree* left, * right;
} WeightedTree;

typedef struct MinHeap
{
    WeightedTree** data;
    int size;
    int capacity;
} MinHeap;
```

```c
// Node of the queue
typedef struct QueueNode
{
    WeightedTree* data;
    struct QueueNode* next;
} QueueNode;

// For prefix checking in level order
typedef struct Queue
{
    QueueNode* head;
    QueueNode* tail;
}Queue;

// Queue
Queue* CreateQueue();
void Enqueue(Queue* queue, WeightedTree* tree);
WeightedTree* Dequeue(Queue* queue);
void FreeQueue(Queue* queue);

// Heap
MinHeap* CreateMinHeap(int n);
void PercDown(MinHeap* heap, int p);
WeightedTree* DeleteMin(MinHeap* heap);
void Insert(MinHeap* heap, WeightedTree* tree);
void FreeHeap(MinHeap* heap);

int GetWPL(WeightedTree* tree, int depth);
WeightedTree* BuildHuffmanTree(MinHeap* heap);
WeightedTree* BuildWeightedTree(char code[MAX_N][MAX_LEN], int N, int* f);
void FreeWeightedTree(WeightedTree* tree);
int CheckPrefixAndOptimal(WeightedTree* tree, int optimalWpl);

int main()
{
    int N, M, i, j;
    int f[MAX_N] = { 0 };
    char code[MAX_N][MAX_LEN];

    scanf("%d", &N);

    MinHeap* heap = CreateMinHeap(N);   // Initialize an empty heap

    for (i = 0; i < N; ++i)
    {
        scanf("%*s %d", &f[i]);

        //Fill the heap content simultaneously, though it will not be a MinHeap
yet
        heap->data[++heap->size] = (WeightedTree*)malloc(sizeof(WeightedTree));
        heap->data[heap->size]->weight = f[i];
        heap->data[heap->size]->left = heap->data[heap->size]->right = NULL;
    }

    // Adjust array data to MinHeap with time complexity of O(N)
    for (i = N / 2; i >= 1; i--)
        PercDown(heap, i);
```

```
    //Use huffman algorithm to generate a optimal length
    WeightedTree* huffmanTree = BuildHuffmanTree(heap);

    int huffman = GetWPL(huffmanTree, 0);

    scanf("%d", &M);

    //Compare the length of students given code and check prefix
    for (i = 0; i < M; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            scanf("%*s %s", code[j]);
        }

        // Build WeightedTree according to the code input
        WeightedTree* tree = BuildWeightedTree(code, N, f);

        if (CheckPrefixAndOptimal(tree, huffman))   // Check if the code satisfy
optimal and prefix requirement
        {
            printf("Yes\n");
        }
        else
        {
            printf("No\n");
        }

        // Free the memory that is mallocated
        FreeWeightedTree(tree);
    }

    // Free the memory that is mallocated
    FreeHeap(heap);
    FreeWeightedTree(huffmanTree);
    return 0;
}

int CheckPrefixAndOptimal(WeightedTree* tree, int optimalWpl)
{
    WeightedTree* tmp;
    Queue* queue = CreateQueue();
    Enqueue(queue, tree);
    int wpl = 0;

    while (queue->tail)//queue is not empty
    {
        tmp = Dequeue(queue);

        //Check isPrefix
        if (tmp->weight != INF) // The path to this node is a code
        {
            if (tmp->left || tmp->right) // isPrefix
            {
                wpl = INF;
                break;
            }
```

```
        }

        if (tmp->left)
            Enqueue(queue, tmp->left);
        if (tmp->right)
            Enqueue(queue, tmp->right);
    }

    if (wpl != INF)
        wpl = GetWPL(tree, 0);
    FreeQueue(queue);

    return wpl == optimalWpl;
}


// Build Tree according to the code given
WeightedTree* BuildWeightedTree(char code[MAX_N][MAX_LEN], int N, int* f)
{
    WeightedTree* tree = (WeightedTree*)malloc(sizeof(WeightedTree));
    tree->left = tree->right = NULL;
    tree->weight = INF; // Set the weight to INF to denote that the path to it
is not a code

    WeightedTree* tmp = tree;
    int i, j;
    for (i = 0; i < N; ++i)
    {
        tmp = tree;
        for (j = 0; code[i][j] != '\0'; ++j)    // While it is not the end of
the code
        {
            if (code[i][j] == '0')  // '0' means a left child
            {
                if (!tmp->left) // If child not exists, create it
                {
                    tmp->left = (WeightedTree*)malloc(sizeof(WeightedTree));
                    tmp->left->left = NULL;
                    tmp->left->right = NULL;
                    tmp->left->weight = INF;
                }
                tmp = tmp->left;
            }
            else // '1' means a right child
            {
                if (!tmp->right)    // If child not exists, create it
                {
                    tmp->right = (WeightedTree*)malloc(sizeof(WeightedTree));
                    tmp->right->left = NULL;
                    tmp->right->right = NULL;
                    tmp->right->weight = INF;
                }
                tmp = tmp->right;
            }
        }
        tmp->weight = f[i]; // Set the weight of the code node to its frequency
    }
```

```
    return tree;
}

// Use Huffman algorithm to build huffman tree
WeightedTree* BuildHuffmanTree(MinHeap* heap)
{
    WeightedTree* node;

    while (heap->size > 1)
    {
        node = (WeightedTree*)malloc(sizeof(WeightedTree));
        node->left = DeleteMin(heap);
        node->right = DeleteMin(heap);
        node->weight = node->left->weight + node->right->weight;
        Insert(heap, node);
    }

    return DeleteMin(heap); // return the root of the huffman tree
}

// Release memory recursively
void FreeWeightedTree(WeightedTree* tree)
{
    if (!tree) return;
    FreeWeightedTree(tree->left);
    FreeWeightedTree(tree->right);
    free(tree);
}

// Compute WPL recursively for huffman tree
int GetWPL(WeightedTree* tree, int depth)
{
    if (!tree)
        return INF;

    if (!tree->left && !tree->right)
        return depth * tree->weight;
    else
        return GetWPL(tree->left, depth + 1) + GetWPL(tree->right, depth + 1);
}

//MinHeap

// Create an empty MinHeap
MinHeap* CreateMinHeap(int n)
{
    MinHeap* heap;
    heap = (MinHeap*)malloc(sizeof(MinHeap));
    heap->data = (WeightedTree**)malloc(sizeof(WeightedTree*) * (MAX_N + 1));
    heap->capacity = MAX_N;
    heap->size = 0;

    heap->data[0] = (WeightedTree*)malloc(sizeof(WeightedTree));
    heap->data[0]->weight = -1; // sentinel
    heap->data[0]->left = heap->data[0]->right = NULL;

    return heap;
}
```

```
// Adjust pth node in heap to deeper to make it a MinHeap
void PercDown(MinHeap* heap, int p)
{
    int parent, child;
    WeightedTree* tmp = heap->data[p];
    int n = heap->size;
    for (parent = p; 2 * parent <= n; parent = child)
    {
        child = 2 * parent;
        if (child + 1 <= n && heap->data[child + 1]->weight < heap->data[child]-
>weight)    // Find the smaller Node
            child++;

        if (tmp->weight > heap->data[child]->weight)    // Move node to higher
to make space for the tmp
        {
            heap->data[parent] = heap->data[child];
        }
        else // Find the proper place
        {
            break;
        }
    }
    heap->data[parent] = tmp;
}

// Insert a WeightedTree to heap
void Insert(MinHeap* heap, WeightedTree* tree)
{
    int i = ++heap->size;
    if (i <= heap->capacity)
    {
        for (; heap->data[i / 2]->weight > tree->weight; i /= 2)    // Find
proper place to insert it
        {
            heap->data[i] = heap->data[i / 2];  // move the original nodes to
make space for new node
        }
        heap->data[i] = tree;
    }
}

// Delete the min node in heap
WeightedTree* DeleteMin(MinHeap* heap)
{
    WeightedTree* minTree = NULL;
    if (heap->size)
    {
        minTree = heap->data[1];    // The min is at the 1th place
        heap->data[1] = heap->data[heap->size--];   // Move the last node to 1th
node
        PercDown(heap, 1);  // Adjust the 1th node
    }

    return minTree;
}
```

```
// Release memory
void FreeHeap(MinHeap* heap)
{
    free(heap->data);
    free(heap);
}

// Queue

// Create an empty queue
Queue* CreateQueue()
{
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->head = queue->tail = NULL;
    return queue;
}

// Enqueue WeightedTree
void Enqueue(Queue* queue, WeightedTree* tree)
{
    // Create new node for WeightedTree
    QueueNode* node = (QueueNode*)malloc(sizeof(QueueNode));
    node->data = tree;
    node->next = NULL;

    if (queue->tail)    // If the queue has elements
    {
        queue->tail->next = node;
        queue->tail = node;
    }
    else // If it is an empty queue
    {
        queue->head = queue->tail = node;
    }
}

WeightedTree* Dequeue(Queue* queue)
{
    WeightedTree* result = NULL;
    QueueNode* tmp;
    if (queue->head)    // If the queue is not empty
    {
        tmp = queue->head;
        queue->head = tmp->next;
        if (!tmp->next)
            queue->tail = NULL;

        result = tmp->data;
        free(tmp);  // Release memory
    }

    return result;
}

void FreeQueue(Queue* queue)
{
    QueueNode* p = queue->head;
    QueueNode* tmp = p;
```

```
    while (p)
    {
        tmp = p->next;
        free(p);
        p = tmp;
    }
    free(queue);
}
```

- Iterative Version

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

#define MAX_N 63
#define MAX_LEN 64
#define INF 1001

typedef struct WeightedTree
{
    int weight;
    struct WeightedTree* left, * right;
} WeightedTree;

typedef struct MinHeap
{
    WeightedTree** data;
    int size;
    int capacity;
} MinHeap;

// Node of the queue
typedef struct QueueNode
{
    WeightedTree* data;
    struct QueueNode* next;
} QueueNode;

// For WPL computing in level order
typedef struct Queue
{
    QueueNode* head;
    QueueNode* tail;
}Queue;

// Queue
Queue* CreateQueue();
void Enqueue(Queue* queue, WeightedTree* tree);
WeightedTree* Dequeue(Queue* queue);
void FreeQueue(Queue* queue);

// Heap
MinHeap* CreateMinHeap(int n);
void PercDown(MinHeap* heap, int p);
WeightedTree* DeleteMin(MinHeap* heap);
```

```
void Insert(MinHeap* heap, WeightedTree* tree);
void FreeHeap(MinHeap* heap);

int GetWPL(WeightedTree* tree);
WeightedTree* BuildHuffmanTree(MinHeap* heap);
WeightedTree* BuildWeightedTree(char code[MAX_N][MAX_LEN], int N, int* f);
void FreeWeightedTree(WeightedTree* tree);
int CheckPrefixAndOptimal(WeightedTree* tree, int optimalWpl);

int main()
{
    int N, M, i, j;
    int f[MAX_N] = { 0 };
    char code[MAX_N][MAX_LEN];

    scanf("%d", &N);

    MinHeap* heap = CreateMinHeap(N);   // Initialize an empty heap

    for (i = 0; i < N; ++i)
    {
        scanf("%*s %d", &f[i]);

        //Fill the heap content simultaneously, though it will not be a MinHeap
yet
        heap->data[++heap->size] = (WeightedTree*)malloc(sizeof(WeightedTree));
        heap->data[heap->size]->weight = f[i];
        heap->data[heap->size]->left = heap->data[heap->size]->right = NULL;
    }

    // Adjust array data to MinHeap with time complexity of O(N)
    for (i = N / 2; i >= 1; i--)
        PercDown(heap, i);

    //Use huffman algorithm to generate a optimal length
    WeightedTree* huffmanTree = BuildHuffmanTree(heap);

    int huffman = GetWPL(huffmanTree);

    scanf("%d", &M);

    //Compare the length of students given code and check prefix
    for (i = 0; i < M; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            scanf("%*s %s", code[j]);
        }

        // Build WeightedTree according to the code input
        WeightedTree* tree = BuildWeightedTree(code, N, f);

        if (CheckPrefixAndOptimal(tree, huffman))   // Check if the code satisfy
optimal and prefix requirement
        {
            printf("Yes\n");
        }
        else
```

```
        {
            printf("No\n");
        }

        // Free the memory that is mallocated
        FreeWeightedTree(tree);
    }

    // Free the memory that is mallocated
    FreeHeap(heap);
    FreeWeightedTree(huffmanTree);
    return 0;
}

int CheckPrefixAndOptimal(WeightedTree* tree, int optimalWpl)
{
    WeightedTree* tmp;
    Queue* queue = CreateQueue();   // Use queue to traverse the tree in level
order
    Enqueue(queue, tree);

    WeightedTree* level = (WeightedTree*)malloc(sizeof(WeightedTree));
    level->left = level->right = NULL;
    level->weight = -1;
    Enqueue(queue, level);  // This node is used to denote the end of each level
                            // With this, we can get the depth for wpl computing

    int wpl = 0;
    int depth = 0;

    while (queue->tail)//queue is not empty
    {
        tmp = Dequeue(queue);

        if (tmp->weight == -1 && queue->head)   // If this node is the level
node, that is, the end of the level
        {
            depth++;
            Enqueue(queue, level);
        }
        else if (tmp->weight == -1 && !queue->head) // If there is no more level
        {

        }
        else // If there is still some node at this level
        {
            //Check isPrefix and computing wpl
            if (tmp->weight != INF) // The path to this node is a code
            {
                if (tmp->left || tmp->right) // isPrefix
                {
                    wpl = INF;
                    break;
                }
                wpl += tmp->weight * depth; // computing ucrrent wpl
            }

            // If this node has children, enqueue to traverse them later
```

```
            if (tmp->left)
                Enqueue(queue, tmp->left);
            if (tmp->right)
                Enqueue(queue, tmp->right);
        }
    }

    // Free the memory that is mallocated
    FreeQueue(queue);
    free(level);

    // If the wpl of this tree is the same with optimalWpl got by Huffman
algorithm,
    // this should be an optimal encoding way as we have set wpl of code not
satisfying prefix rule to INF.
    return wpl == optimalWpl;
}

// Build Tree according to the code given
WeightedTree* BuildWeightedTree(char code[MAX_N][MAX_LEN], int N, int* f)
{
    WeightedTree* tree = (WeightedTree*)malloc(sizeof(WeightedTree));
    tree->left = tree->right = NULL;
    tree->weight = INF; // Set the weight to INF to denote that the path to it
is not a code

    WeightedTree* tmp = tree;
    int i, j;
    for (i = 0; i < N; ++i)
    {
        tmp = tree;
        for (j = 0; code[i][j] != '\0'; ++j)    // While it is not the end of
the code
        {
            if (code[i][j] == '0')  // '0' means a left child
            {
                if (!tmp->left) // If child not exists, create it
                {
                    tmp->left = (WeightedTree*)malloc(sizeof(WeightedTree));
                    tmp->left->left = NULL;
                    tmp->left->right = NULL;
                    tmp->left->weight = INF;
                }
                tmp = tmp->left;
            }
            else // '1' means a right child
            {
                if (!tmp->right)    // If child not exists, create it
                {
                    tmp->right = (WeightedTree*)malloc(sizeof(WeightedTree));
                    tmp->right->left = NULL;
                    tmp->right->right = NULL;
                    tmp->right->weight = INF;
                }
                tmp = tmp->right;
            }
        }
        tmp->weight = f[i]; // Set the weight of the code node to its frequency
```

```
    }

    return tree;
}

// Use Huffman algorithm to build huffman tree
WeightedTree* BuildHuffmanTree(MinHeap* heap)
{
    WeightedTree* node;

    while (heap->size > 1)
    {
        node = (WeightedTree*)malloc(sizeof(WeightedTree));
        node->left = DeleteMin(heap);
        node->right = DeleteMin(heap);
        node->weight = node->left->weight + node->right->weight;
        Insert(heap, node);
    }

    return DeleteMin(heap); // return the root of the huffman tree
}

// Release memory iteratively
void FreeWeightedTree(WeightedTree* tree)
{
    Queue* queue = CreateQueue();
    WeightedTree* tmp;
    Enqueue(queue, tree);

    while (queue->head)
    {
        tmp = Dequeue(queue);
        if (tmp->left)
            Enqueue(queue, tmp->left);
        if (tmp->right)
            Enqueue(queue, tmp->right);
        free(tmp);
    }

    FreeQueue(queue);
}

// Compute WPL iteratively for huffman tree
int GetWPL(WeightedTree* tree)
{
    WeightedTree* tmp;
    Queue* queue = CreateQueue();
    Enqueue(queue, tree);

    WeightedTree* level = (WeightedTree *)malloc(sizeof(WeightedTree)); //
Denote end of one level
    level->left = level->right = NULL;
    level->weight = -1;
    Enqueue(queue, level);  // End of the first level

    int wpl = 0;
    int depth = 0;
```

```
    while (queue->tail)//queue is not empty
    {
        tmp = Dequeue(queue);

        if (tmp->weight == -1 && queue->head)  // Check if it is the end of the
level
        {
            depth++;
            Enqueue(queue, level);
        }
        else if (tmp->weight == -1 && !queue->head) //  If there is no other
node except level node
        {

        }
        else
        {
            if (!tmp->left && !tmp->right) // The path to this node is a code
            {
                wpl += tmp->weight * depth;
            }

            if (tmp->left)
                Enqueue(queue, tmp->left);
            if (tmp->right)
                Enqueue(queue, tmp->right);
        }
    }

    FreeQueue(queue);
    free(level);

    return wpl;
}

//MinHeap

// Create an empty MinHeap
MinHeap* CreateMinHeap(int n)
{
    MinHeap* heap;
    heap = (MinHeap*)malloc(sizeof(MinHeap));
    heap->data = (WeightedTree**)malloc(sizeof(WeightedTree*) * (MAX_N + 1));
    heap->capacity = MAX_N;
    heap->size = 0;

    heap->data[0] = (WeightedTree*)malloc(sizeof(WeightedTree));
    heap->data[0]->weight = -1; // sentinel
    heap->data[0]->left = heap->data[0]->right = NULL;

    return heap;
}

// Adjust pth node in heap to deeper to make it a MinHeap
void PercDown(MinHeap* heap, int p)
{
    int parent, child;
    WeightedTree* tmp = heap->data[p];
```

```
    int n = heap->size;
    for (parent = p; 2 * parent <= n; parent = child)
    {
        child = 2 * parent;
        if (child + 1 <= n && heap->data[child + 1]->weight < heap->data[child]->weight)    // Find the smaller Node
            child++;

        if (tmp->weight > heap->data[child]->weight)    // Move node to higher to make space for the tmp
        {
            heap->data[parent] = heap->data[child];
        }
        else // Find the proper place
        {
            break;
        }
    }
    heap->data[parent] = tmp;
}

// Insert a WeightedTree to heap
void Insert(MinHeap* heap, WeightedTree* tree)
{
    int i = ++heap->size;
    if (i <= heap->capacity)
    {
        for (; heap->data[i / 2]->weight > tree->weight; i /= 2)    // Find proper place to insert it
        {
            heap->data[i] = heap->data[i / 2];  // move the original nodes to make space for new node
        }
        heap->data[i] = tree;
    }
}

// Delete the min node in heap
WeightedTree* DeleteMin(MinHeap* heap)
{
    WeightedTree* minTree = NULL;
    if (heap->size)
    {
        minTree = heap->data[1];    // The min is at the 1th place
        heap->data[1] = heap->data[heap->size--];   // Move the last node to 1th node
        PercDown(heap, 1);  // Adjust the 1th node
    }

    return minTree;
}

// Release memory
void FreeHeap(MinHeap* heap)
{
    free(heap->data);
    free(heap);
}
```

```
// Queue

// Create an empty queue
Queue* CreateQueue()
{
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->head = queue->tail = NULL;
    return queue;
}

// Enqueue WeightedTree
void Enqueue(Queue* queue, WeightedTree* tree)
{
    // Create new node for WeightedTree
    QueueNode* node = (QueueNode*)malloc(sizeof(QueueNode));
    node->data = tree;
    node->next = NULL;

    if (queue->tail)    // If the queue has elements
    {
        queue->tail->next = node;
        queue->tail = node;
    }
    else // If it is an empty queue
    {
        queue->head = queue->tail = node;
    }
}

WeightedTree* Dequeue(Queue* queue)
{
    WeightedTree* result = NULL;
    QueueNode* tmp;
    if (queue->head)    // If the queue is not empty
    {
        tmp = queue->head;
        queue->head = tmp->next;
        if (!tmp->next)
            queue->tail = NULL;

        result = tmp->data;
        free(tmp);  // Release memory
    }

    return result;
}

void FreeQueue(Queue* queue)
{
    QueueNode* p = queue->head;
    QueueNode* tmp = p;

    while (p)
    {
        tmp = p->next;
        free(p);
        p = tmp;
```

```
    }
    free(queue);
}
```

References

No references.

Declaration

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