Huffman Codes

Advance Data Structure

Project 5

Yizhou Chen Zewei Cai Jinze Wu

2019-5-5

Content

Chapter 1 Introduction

1.1 Background Information

Huffman coding is an application of Huffman tree，which is widely used, such as JPEG in the application of the Huffman coding. First of all, Let us talk about Huffman tree. Huffman tree is also called optimal binary tree, which is a binary tree with the **shortest** weighted path length(**WPL**). WPL is that the weight of all the leaf nodes in the tree is multiplied by the path length from the tree to the root node.

1.2 Problem Description

● General Description

The problem is that we need to judge whether the Huffman codes submitted by students are correct or incorrect.

● Solving steps

- Firstly, we input the basic data with characters and frequency of the characters to generate a Huffman tree.
- Then, we calculate the weight path length of generated Huffman tree and the tree generated by students' Huffman code.
- Finally, we judge the student' s submission by comparing the length of two weighted paths and checking the correctness of the tree.

- **Sample**

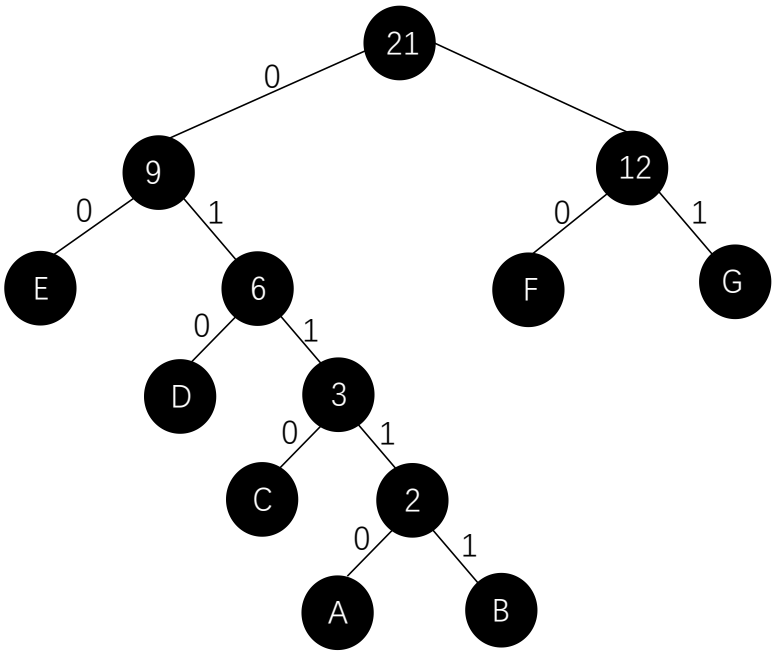
INPUT:

```
7          (N distinct  characters)
A  1  B  1  C  1  D  3  E  3  F  6  G  6          (c[1]  f[1]  c[2]  f[2]  ...  c[N]  f[N])
1          (M student  submissions)
A  00000
B  00001
C  0001
D  001
E  01
F  10
G  11
```

OUTPUT

```
Yes        (The  answer)
```

Huffman Tree Generated by Program



From the tree, we can get the WPL is equal to 53 which is same as the case1.
(WPL_Case1=2*6+2*6+2*3+3*3+4+5+5=53)

Chapter 2 Algorithm Specification

2.1 Statement

We group come up with three solutions for this project. In fact, these are two ways to build the Huffman tree and two ways to check for prefix collisions. But for every solution, we all build the correct Huffman tree with the given input, and then check the prefix conflict of all samples one by one.

2.2 Structure

The following structure is used for storing inputs from the users. We will use the structure to set up a Huffman tree.

```
1:  typedef struct HuffNode {
2:      char c;         // the character
3:      int f;          // the frequency
4:      HuffNode *Left, *Right;   // the left and right son
5:  }*HuffNodePtr;   // the pointer
6:  typedef HuffNodePtr HuffTree;
```

Besides, another structure is used in PR2 to help check prefix collisions. For this algorithm, we would insert nodes, which need to be checked, one by one to test whether it will cause conflict.

```
1:  typedef struct TrieNode {
2:      bool isLeaf, isVisited;      //attribute of nodes
3:      TrieNode *Left,*Right;     // the left and right son
4:  } *TrieNodePtr;
5:  typedef TrieNodePtr TrieTree;
```

PR3 uses heap structure to help build a heap which is convenient to build a Huffman tree.

```
1:  typedef struct HeapStruct {
2:      int size,capacity;      //Current size and capacity of a heap
3:      HuffNodePtr *Elements;      //Huffman tree nodes
4:  } *Heap;
```

2.3 Main sketch of solutions

2.3.1 PR1

2.3.1.1 Specification

This algorithm uses queue combination method to generate a Huffman tree and check prefix conflict by comparing the strings. Details will be introduced in later part.

2.3.1.2 Pseudo code:

Main Function

```
1: function main ( )
2:  Input and initialize Q_leaf : n integers, n characters and n freq
3:      sort by the frequency
4:      GenerateHuffmanTree(Q_leaf,n);
5:      Calculate wpl;
6:      input samples
9:          Calculate the current wpl
10:         If(wpl != current wpl) then output   "No"
11:         else sort code by the length
12:         Checking by comparing the strings
17: End function
```

2.3.2 PR2

2.3.2.1 Specification

This algorithm uses queue combination method to generate a Huffman tree and check prefix conflict by building a tree method. Details will be introduced in later part.

2.3.2.2 Pseudo code:

Main Function

```
1: function main ( )
2:    Input and initialize Q_leaf: n integers, n characters and n freq
3:        sort by the frequency
4:        GenerateHuffmanTree(Q_leaf,n);
5:        Calculate wpl;
6:        input samples
7:            Calculate the current wpl
8:            If(wpl != current wpl) then output   "No"
9:            else
10:               Trieroot = createTrieNode();
11:               For i = 0 to n-1
12:                   if (!insert(Trieroot,code[i]))
13:                       then output   "No"
14:           output   "Yes"
15:           deleteTrieTree(Trieroot);
16: End function
```

2.3.3 PR3

Specification

This algorithm uses min-heap method to generate a Huffman tree and check prefix conflict by building a tree method. Details will be introduced in later part. The

main() function is the same as PR2’ s. the difference between them is function specific implementation. Thus, we won’ t paste the pseudo code here and details will be introduced in later part.

2.4 Two Ways to Generate a Huffman Tree

2.4.1 Queue Combination Method

2.4.1.1 Specification

As we know, we have n raw nodes stored in a queue at first. And, p means the number of discrete nodes we haven’ t deal with, q means number of roots which set up by the nodes, i means index of the current node we should deal with next and j means index of the current roots. What’ s need to be done is to combine the two smallest nodes into one new node, which should be pushed into another queue until get a whole tree.

2.4.1.2 Pseudo code:

Important Function

```
1: function GenerateHuffmanTree(HuffNodePtr *Q_leaf, int n)
2:    Initialize: p=n, q=0, i=0, j=0;
3:        while (p != 0 || q != 0)
4:            if (p == 1 && q == 0) then root = Q_leaf[i];    break;
5:            if (p == 0 && q == 1) then root = Q_nonleaf[j];    break;
6:            if (q >= 2 && (p == 0 || Q_leaf[i]->f > Q_nonleaf[j+1]->f))
7:                then combine Q_nonleaf[j] and Q_nonleaf[j+1]
8:            else if (p >= 2 && (q == 0 || Q_leaf[i+1]->f < Q_nonleaf[j]->f))
9:                then combine Q_leaf[i+1] and Q_nonleaf[j]
10:           else combine Q_leaf[i] and Q_nonleaf[j]
11:           Return root
12: End function
```

2.4.2 Min-Heap Method

2.4.2.1 Specification

PR3 uses min-heap method to build the Huffman tree. At first, we can directly build min-heap using array structure. After that, we delete two minimum nodes at a time and add up the frequencies of the two nodes as their parent, and insert the parent node back into the min-heap. Finally, a Huffman tree can be gotten. The insertion and deletion of the heap are not covered here for we have learned it in DS course.

2.4.2.2 Pseudo code:

Important Function

```
1: function GenerateHuffmanTree(Heap H, int n)
2:         for i = 0 to H->size-2
3:             x = H->Elements[1];
4:             deleteMin(H);
5:             y = H->Elements[1];
6:             deleteMin(H);
7:             temp = new HuffNode;
8:             temp->f = x->f + y->f;
9:             temp->Left = x;
10:            temp->Right = y;
11:            insert(temp,H);
12:        return H->Elements[1];
13: End function
```

2.5 Two Ways to Check Prefix Conflict

2.5.1 Comparing strings Method

2.5.1.1 Specification

we sort the sample letters based on the given code length and directly check whether the short string is included at the beginning of long string. If so, it causes conflicts.

Main Function

```
1: function Check1 ( )
2:          sort code by the length
3:          flag =true;
4:          For i = 0 to n-2
5:               For j = i+1 to n-1
6:                    if (code[i] == code[j].substr(0,code[i].length()))
7:                         flag =false;
8:                         break;
9:          if(flag) then output   "Yes"
10:         else output   "No"
11: End function
```

2.5.2 Building a Tree Method

2.5.2.1 Specification

For every sample, the code of every node is given. What' s need to be done is to build a tree based on the code. If the current element is '0', we set up one left child; If the current element is '1', we set up one right child. Besides, the nodes in the path of one letter would be marked(T->isVisited = true) and the letter where it is will be marked (T->isLeaf = true). Next, if the path of one node is contained in the path of another node, 'No' would be output.

2.5.2.2 Pseudo code:

Important Function

```
1: function insert(TrieTree root, string s)
2:    Initialize: T = root;
3:        For i = 0 to s.length()-1
4:            if (s[i]=='0')
5:                if ( T->Left == NULL)
6:                    then T->Left = createTrieNode();
7:                T = T->Left;
8:            else
9:                if (T->Right == NULL)
10:                   then T->Right = createTrieNode();
11:               T = T->Right;
12:           if (T->isLeaf) then return false;
13:           if (i == s.length()-1 && T->isVisited) then return false;
14:           if (i == s.length()-1) then T->isLeaf = true;
15:           T->isVisited = true;
16:       return true;
17: End function
```

2.6 Correctness of Algorithm

- Core Algorithm

Firstly, Find out the two smallest nodes, calculate the sum of the two weights, and then establish a parent node. Then, delete the two smallest nodes and incorporate the parent node into the new tree.

- Prove

According to the algorithm, a set of N leaf nodes takes two minimum nodes at a time and adds one node until only the last node is left. Therefore, the number of non-leaf nodes must be n-1. We can find WPL= the sum of non-leaf nodes' value. Therefore, in order to ensure the minimum of WPL, we just need to ensure the minimum of non-leaf nodes' value(generated nodes' value). A new node is

generated through two smallest nodes each time, which can ensure that each new node is the smallest and the non-leaf nodes' value is minimum.

Chapter 3 Testing Results

3.1 Purpose

Our purpose is to test the correctness and efficiency of the algorithm.

3.2 Preparation

3.2.1 Correctness Test

Special Data Test (Huffman Tree)

Firstly, we test the algorithm with ordinary sample and extreme data, including the given sample, the characters with same frequency, only one character inputted, two character inputted and 63(maximum) characters inputted.

Massive Data Test (Students' Submission)

Secondly, we test the algorithm with some random different student' s submission data with 30 characters.

In this project, we write 3 different program and use the first program to generate **correct but different** submission data by changing the position of the characters in the input data because it will change the order of selecting two minimum value characters.

In order to get incorrect submission data, we randomly change "0 into 1 or 1 into 0" in some Huffman codes. Finally, we get all of the submission data and input them into program3 to check the correctness.

3.2.2 Time Consumption test

- Generate Huffman Tree

We randomly generate a sequence with number of characters from 10 to 60 and test the time consumption of generating tree by two kinds of algorithms.

- Judge the Prefix Conflict

We randomly generate some input data with prefix conflict to test the time consumption of two different kinds of algorithms which are used to check the conflict of prefix conflict.

3.3 Test Data

3.3.1 Special data test

- Sample of PTA

```
7
A 1 B 1 C 1 D 3 E 3 F 6 G 6
```

```
4
```

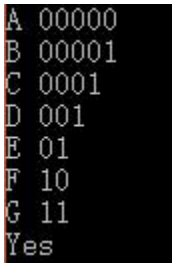
```
A 00000
B 00001
C 0001
D 001
E 01
F 10
G 11
```

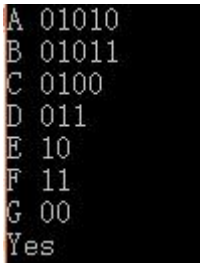
```
A 01010
B 01011
C 0100
D 011
E 10
F 11
G 00
```

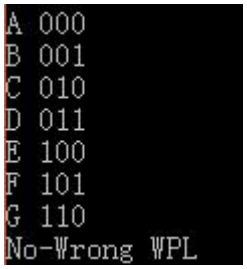
```
A 000
B 001
C 010
D 011
E 100
F 101
G 110
```

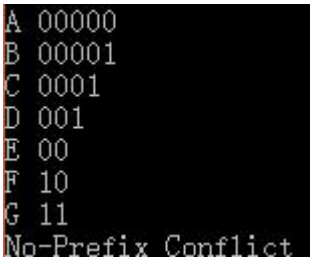
```
A 00000
B 00001
C 0001
D 001
E 00
F 10
G 11
```

Result:









- **Sample of characters with same frequency**

```
6
0 1 1 1 2 1 3 1 4 1 5 1
```

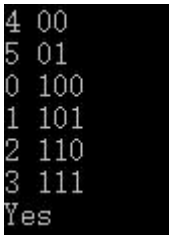
```
3
```

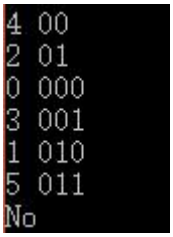
```
4 00
5 01
0 100
1 101
2 110
3 111
```

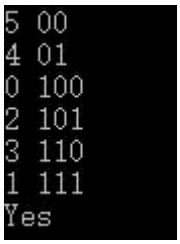
```
4 00
2 01
0 000
3 001
1 010
5 011
```

```
5 00
4 01
0 100
2 101
3 110
1 111
```

Result:







- **Sample of 2 characters**

Input

```
2
A 1 B 1
3
A 0
B 1
A 1
B 1
A 1
```

Output

- **Sample of 63 characters**

```
63
o 1 f 1 a 2 1 3 z 3 w 3 V 3 K 3 6 6 5 6 G 6 4 7 t 7 - 8 S 8 8 8 h 9 W 9 A 9 y
9 x 10 g 10 n 12 U 13 c 14 J 15 O 15 L 16 j 16 I 17 s 18 k 18 B 19 X 20 Q 20
i 21 F 21 m 22 C 22 E 22 p 23 0 23 r 23 Y 23 7 24 d 24 M 24 b 24 l 25 N 26 Z
27 H 27 e 29 u 29 2 29 3 30 R 30 D 30 q 31 T 31 v 31 P 31 9 31
```

```
2
```

```
M 00000
l 00001
N 00010
U 000110
G 0001110
K 00011110
a 000111110
o 0001111110
f 0001111111
Z 00100
H 00101
e 00110
c 001110
4 0011110
t 0011111
u 01000
2 01001
3 01010
J 010110
O 010111
R 01100
D 01101
q 01110
T 01111
v 10000
P 10001
9 10010
L 100110
- 1001110
S 1001111
j 101000
8 1010010
h 1010011
I 101010
W 1010110
A 1010111
s 101100
k 101101
B 101110
y 1011110
x 1011111
X 110000
Q 110001
i 110010
F 110011
m 110100
g 1101010
6 11010110
1 110101110
z 110101111
C 110110
E 110111
p 111000
0 111001
r 111010
Y 111011
d 111100
n 1111010
5 11110110
w 111101110
V 111101111
7 111110
b 111111
```

Result:

```
V 111101111
7 111110
b 111111
Yes
```

```
7 00000                 T 01110                 i 010011
l 00001                 q 01111                 m 010100
N 00010                 v 00000                 x 0101010
U 000110                P 00001                 6 01010110
5 0001110               9 00010                 V 010101110
K 00011110              j 000110                z 010101111
a 000111110             S 0001110               C 010110
o 0001111110            8 0001111               E 010111
f 0001111111            L 001000                Y 011000
Z 00100                 - 0010010               r 011001
H 00101                 W 0010011               0 011010
u 00110                 I 001010                p 011011
c 001110                y 0010110               d 011100
t 0011110               A 0010111               n 0111010
4 0011111               s 001100                G 01110110
e 01000                 k 001101                1 011101110
2 01001                 B 001110                w 011101111
D 01010                 h 0011110               M 011110
O 010110                g 0011111               b 011111
J 010111                Q 010000
3 01100                 X 010001
R 01101                 F 010010
```

Result:

```
1 011101110
w 011101111
M 011110
b 011111
No
```

3.3.2 Mass data test

Input

50
j 1 f 2 b 2 A 3 L 4 a 4 K 6 X 6 d 8 i 9 E 9 J 10 B 10 6 10 N 11 n 11 8 12 T 13 l 13 P 13 2 13 I 15 7 15 C 15 W 15 1 15 V 17 M 17 F 18 G 18 5 18 Q 19 k 19 R 20 0 20 Y 20 O 21 4 22 H 22 c 23 e 23 g 23 h 23 D 23 U 23 m 23 S 24 9 25 Z 25 3 25

1000 different student submissions

Output

```
0: No
1: Yes
2: No
3: Yes
4: No
5: No
6: No
7: Yes
8: No
9: No
10: No
11: Yes
12: Yes
13: No
14: No
15: Yes
16: No
17: No
18: Yes
19: Yes
20: No
21: No
22: Yes
23: No
24: No
25: Yes
26: Yes
27: Yes
28: Yes
29: No
30: Yes
31: Yes
32: No
33: No
34: No
35: Yes
36: No
37: Yes
38: No
39: No
40: No
```

```
957: Yes
958: No
959: Yes
960: Yes
961: Yes
962: Yes
963: Yes
964: No
965: No
966: No
967: No
968: No
969: Yes
970: Yes
971: Yes
972: No
973: No
974: No
975: No
976: No
977: No
978: Yes
979: No
980: No
981: Yes
982: No
983: Yes
984: Yes
985: Yes
986: Yes
987: No
988: No
989: No
990: No
991: No
992: Yes
993: No
994: No
995: Yes
996: No
997: No
998: Yes
999: No
```

3.3.2 Time Consumption test

- **Generate Huffman Tree**

In fact, two methods of building a Huffman tree are used here, which queue combination method and min-heap method. Thus, we group want to test time consumption of these two part. But before that, we have analyzed the time complexity of the two algorithms, which are O(N) and O(NlogN) respectively. Now we want to use test results to verify our thoughts. Here is duration time consumption of two algorithms in different sizes.

Test 100000 iteration for the Generate Tree						
Number of Characters	10	20	30	40	50	60
Queue Combination	0.295	0.424	0.554	0.698	0.823	0.953
Heap Method	0.101	0.222	0.36	0.507	0.664	0.821

- **Judge the Prefix Conflict**

Time Consumption of Prefix Conflict with 1000 submissions						
Number of Characters	10	20	30	40	50	60
PR1-O(n2)	0.012	0.044	0.093	0.152	0.254	0.368
PR2-TrieTree	0.002	0.008	0.018	0.021	0.027	0.035

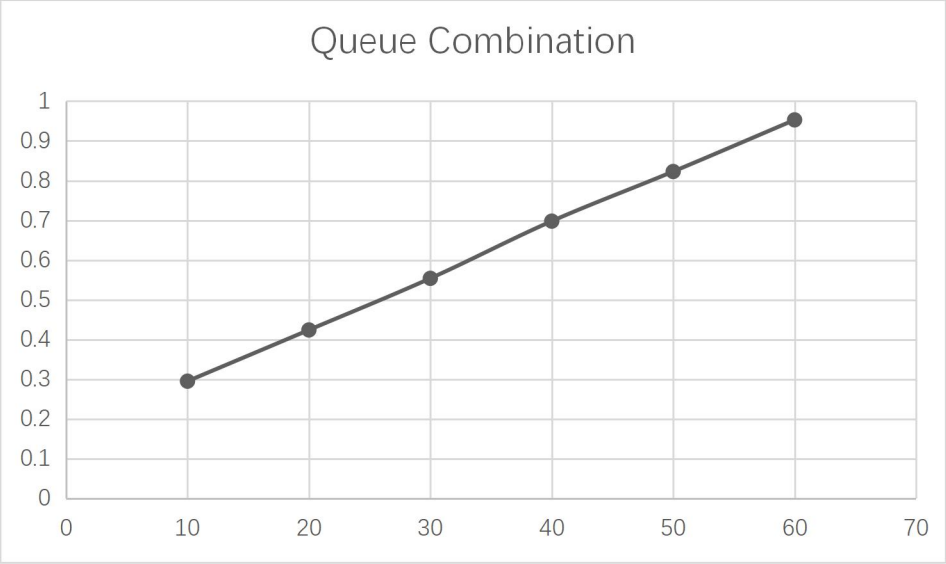
Chapter 4 Analysis and Comments

4.1 Analysis and Comments of the Testing Data

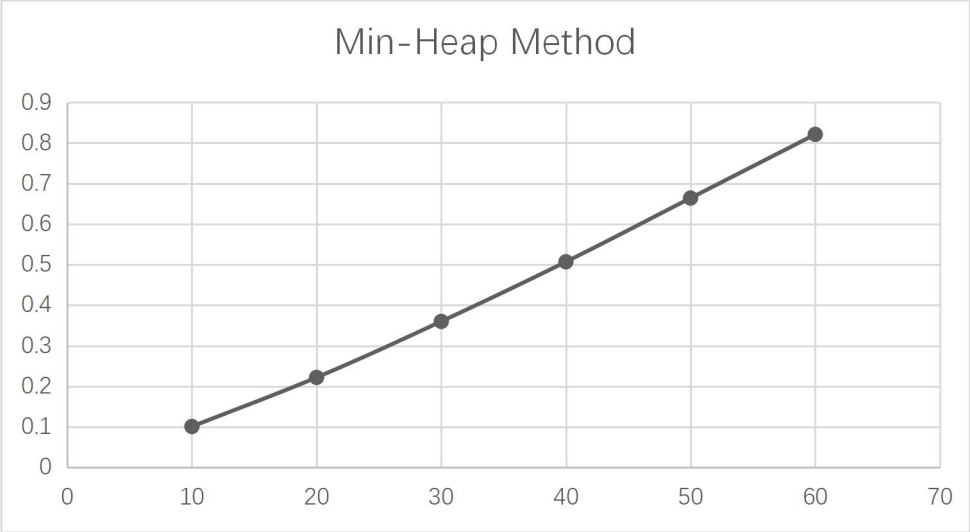
- **Correctness Analysis**

First of all, in the samples of chapter3, we can find that the output data of sample PTA is same as the result of PTA. As for the samples we create, we can theoretically prove that the algorithm of PR1(common algorithm) must be correct correct(the prove part in Chapter2), then we use PR1 to get the input data and put it into PR3(different algorithm), then we check the output data. we find that the output data is same as what we expect.

- **Time Complexity Analysis**
  - **Generate Tree**

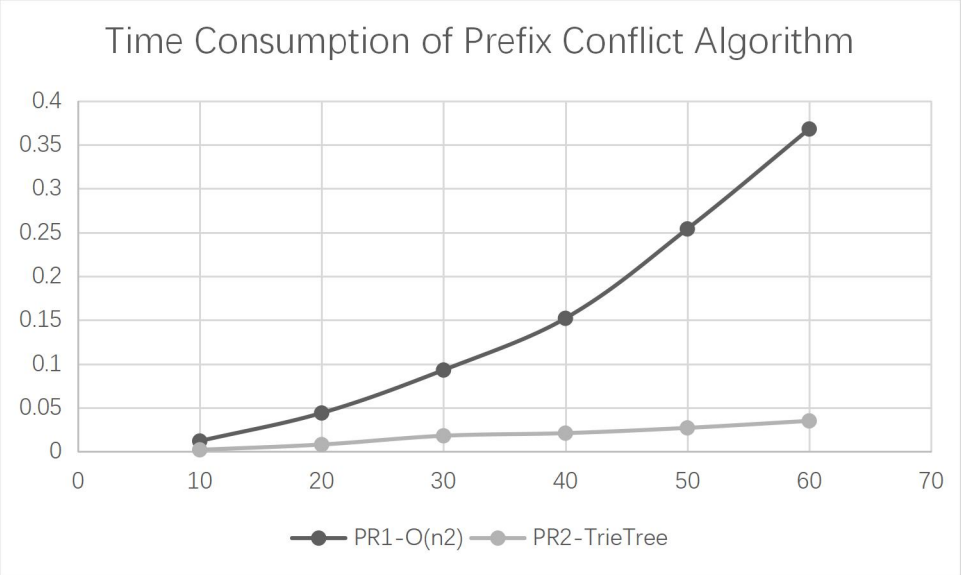


We can clearly see that the resulting graph of time consumption of queue combination method is a straight line. This is in line with our expectations. However, we think the size is too small to make a decision prematurely. But it’ s a good point for us to further analysis of time complexity.



Also, we test consumption result of min-heap method. We also get the approximate line graph. But this is a little crooked be comparison. Thus, we still need to do further analysis.

- **Prefix Conflict**



we test time consumption result of two algorithm used to deal with prefix conflict. After visualizing the test data, we find that using TrieTree can greatly improve the efficiency of checking prefix conflict.

4.2 Pre-analysis about Time Complexity

- PR1

In this part, we use two queue to help combine nodes and build a Huffman tree. For each combination, it costs O(1). And it needs to deal with all nodes once. Thus, the time complexity of building a Huffman tree is O(N). But the first queue needs to be sorted first, which at least costs O(NlogN). Besides, when we calculate the wpl, we traverse all nodes. Therefore, the time complexity of calculating wpl is O(N). For checking, we use two loop to compare one code string to other rest code string to confirm whether there is a conflict. Thus, to check one sample would cost $O(N^2)$.

- PR2

The method of building the Huffman tree is the same as PR1. Thus, it costs O(N) for building, O(N) for calculating wpl and O(NlogN) for sorting. However, the method of checking is to use the code of each node to build a Huffman tree for testing the prefix conflict. Thus, it depends on the length of nodes. The worst time complexity must be $O(\sum L_i)$ and L_i means the length of one letter' s given code.

- PR3

In this algorithm, we build a min-heap first using bottom-up method, which costs O(N). Then we use a loop for all nodes. Every time, we delete two minimum nodes from heap and insert one new node until there is only one node in the heap. For one deletion and one insertion both costs O(logN). Thus, the total time complexity is O(NlogN). Also, calculating wpl costs O(N). And the method of checking is the same as PR1' s, which costs $O(N^2)$.

4.3 Pre-analysis about Space Complexity

- PR1

In this algorithm, we use two queues to combine nodes. But, every two nodes would generate one new parent node, the number of which is N-1. thus, it needs 2N-1 spaces. Thus, the space complexity is O(N). For checking, we compare the code of two nodes directly, So no extra space is required.

- PR2

The building method is the same as PR1, therefore, space complexity is O(N). For checking, it applies space based on the length of the string. Thus, in the worst case, the space complexity is $O(\sum L_i)$ and L_i means the length of one letter’ s given code.

- PR3

Minimum-heap is used in PR3. To build a heap needs N spaces and combine nodes needs another N-1 spaces. Thus, the space complexity of building Huffman tree is O(N). For checking, no extra space is required for the same reason as PR1’ s.

Appendix

Appendix A Source code

- **PR5_1**

```
#include<iostream>
#include<vector>
#include<map>
#include<algorithm>

using namespace std;

typedef struct HuffNode {
    char c;  // the character
    int f;  // the frequency and the weight
    HuffNode *Left, *Right;  // the left and right son
} *HuffNodePtr;  // the pointer
typedef HuffNodePtr HuffTree;

long long wpl = 0;  // the minimal weighted path length

/* the compare function for the frequency */
bool cmpQ(HuffNodePtr x,HuffNodePtr y) {
    return x->f < y->f;  // increasing order
}

/* the compare function for the code */
bool cmpS(string x,string y) {
    return x.length()<y.length();  // increasing order
}

/* This function is used to generate a huffman tree
 * paramater Q_leaf: the initial nodes stored in increasing order
 * paramater n: number of nodes
 * return: the root of the generated huffman tree
 */
HuffTree GenerateHuffmanTree(HuffNodePtr *Q_leaf, int n) {
    vector<HuffNodePtr> Q_nonleaf;  // generalized nodes as non-leaves
    int p = n, q = 0;  // p - number of nodes in Q_leaves q - number of nodes in Q_numleaf
    int i = 0, j = 0; // i - index of the current element in Q_leaf j - index of the current element in Q_nonleaf
```

```
    HuffTree root;
    HuffNodePtr temp;
    while (p != 0 || q != 0) {
        /* if only 1 node left */
        if (p == 1 && q == 0) {
            root = Q_leaf[i];
            break;
        }
        if (p == 0 && q == 1) {
            root = Q_nonleaf[j];
            break;
        }
        /* pick 2 from the non-leaf nodes and add them up to make new node */
        if (q >= 2 && (p == 0 || Q_leaf[i]->f > Q_nonleaf[j+1]->f)) {
            temp = new HuffNode;
            temp->f = Q_nonleaf[j]->f + Q_nonleaf[j+1]->f;
            temp->Left = Q_nonleaf[j];
            temp->Right = Q_nonleaf[j+1];
            Q_nonleaf.push_back(temp);
            j+=2; q--;
        }
        /* pick 2 from the leaf nodes and add them up to make new node */
        else if (p >= 2 && (q == 0 || Q_leaf[i+1]->f < Q_nonleaf[j]->f)) {
            temp = new HuffNode;
            temp->f = Q_leaf[i]->f + Q_leaf[i+1]->f;
            temp->Left = Q_leaf[i];
            temp->Right = Q_leaf[i+1];
            Q_nonleaf.push_back(temp);
            i+=2; p-=2; q++;
        }
        /* pick 1 from leaf nodes and and 1 from non-leaf nodes */
        else {
            temp = new HuffNode;
            temp->f = Q_leaf[i]->f + Q_nonleaf[j]->f;
            temp->Left = Q_leaf[i];
            temp->Right = Q_nonleaf[j];
            Q_nonleaf.push_back(temp);
            i++; j++; p--;
        }
    }
    return root;
}

/* This function is used to calculated the weight of all nodes */
/* paramater root: the curent Huffnode pointer */
void WPL(HuffTree root) {
```

```
    if (root == NULL) return;
    wpl += root->f;
    WPL(root->Left);
    WPL(root->Right);
    delete root; // release the space
}

int main(void) {
    int n,m;

    cin>>n;  // number of nodes

    HuffNodePtr* Q_leaf = new HuffNodePtr[n];  // initial nodes as leaves
    map<char,int> freq;  // the map using characters as keys and frequency as va
lues

    /* input and initialize */
    for (int i = 0; i < n; i++) {
        Q_leaf[i] = new HuffNode;
         cin >> Q_leaf[i]->c >> Q_leaf[i]->f;
         freq[Q_leaf[i]->c] = Q_leaf[i]->f;  // use a map for the afterwords ma
tch
         Q_leaf[i]->Left = Q_leaf[i]->Right = NULL;
     }

    /* sort by the frequency */
    sort(Q_leaf,Q_leaf+n,cmpQ);

    /* generate huffman tree */
    HuffTree root = GenerateHuffmanTree(Q_leaf,n);

    /* encode and calulate the wpl*/
    WPL(root);
    wpl -= root->f;  // the root->f should be excluded

    /* judge */
    char ci;
    string* code = new string[n];
    int pseudo_wpl;
    cin>>m; // number of cases
    for (int r = 0; r < m; r++) {
        /* calculate the weighted path length of the case */
        pseudo_wpl = 0;
        for (int i = 0; i < n; i++) {
            cin>>ci>>code[i];
            pseudo_wpl += freq[ci]*code[i].length();
        }
```

```
        /* if not equal to the calculated minimal wpl */
        if (pseudo_wpl != wpl) {
            cout<<"No"<<endl;
            continue;
        }
        else {
            /* judge if all the nodes are the leaf-nodes */
            sort(code,code+n,cmpS);  // sort the codes in increasing length
            bool flag = true;
            for (int i = 0; i < n-1; i++)
                for (int j = i+1; j < n; j++) {
                        /* if one code is the prefix of another then it can 鈥
    樋 be a leaf node*/
                    if (code[i] == code[j].substr(0,code[i].length())) {
                        flag = false;
                        break;
                    }
                }
            if (flag) cout<<"Yes"<<endl;
            else cout<<"No"<<endl;
        }
    }
    delete []code;

    return 0;
}
```

- **PR5_2**

```
#include<iostream>
#include<vector>
#include<map>
#include<algorithm>

using namespace std;

typedef struct HuffNode {
    char c;  // the character
    int f;  // the frequency and the weight
    HuffNode *Left, *Right;  // the left and right children
} *HuffNodePtr;  // the pointer
typedef HuffNodePtr HuffTree;

typedef struct TrieNode {
    bool isLeaf,isVisited;  // marks if the node is a leaf node or is visited
    TrieNode *Left,*Right;  // the left and right children
} *TrieNodePtr;  // the pointer
```

```
typedef TrieNodePtr TrieTree;

long long wpl = 0;  // the minimal weighted path length

/* This function is used to create a new node for trie tree
 * return the TrieNode pointer
 */
TrieNodePtr createTrieNode() {
    TrieNodePtr T = new TrieNode;
    T->Left=T->Right = NULL;
    T->isLeaf = T->isVisited = false;  // intialize
    return T;
}

/* This function is used to insert a new code to trie tree and judge if it is il
legal
 * paramater root: the root of the trie tree
 * paramater s: the code to be inserted
 * return true for legal false for illegal
 */
bool insert(TrieTree root, string s) {
    TrieNodePtr T = root;
    for (int i = 0; i < s.length(); i++) {
        /* if 0 insert to the left subtree */
        if (s[i]=='0') {
            if ( T->Left == NULL)
                T->Left = createTrieNode();
            T = T->Left;
        }
        /* if 1 insert to the right subtree */
        else {
            if (T->Right == NULL)
                T->Right = createTrieNode();
            T = T->Right;
        }
        /* if the curent node is the leaf node of another code */
        if (T->isLeaf) return false;
        /* if current node reaches its end but the node is already visited */
        if (i == s.length()-1 && T->isVisited) return false;
        /* set the leaf mark */
        if (i == s.length()-1) T->isLeaf = true;
        /* set the visited mark */
        T->isVisited = true;
    }
    return true;
}

```

```
65.
66. /* This function is used to delete the trie tree and release the space
67.  * paramater root: the current node pointer
68.  */
69. void deleteTrieTree(TrieNodePtr root) {
70.     if (root == NULL) return;
71.     deleteTrieTree(root->Left);  // delete the left child
72.     deleteTrieTree(root->Right);  // delete thr right child
73.     delete root;
74. }
75.
76. /* the compare function for frequency */
77. bool cmpQ(HuffNodePtr x,HuffNodePtr y) {
78.     return x->f < y->f;  // increasing order
79. }
80.
81. /* This function is used to generate a huffman tree
82.  * paramater Q_leaf: the initial nodes stored in increasing order
83.  * paramater n: number of nodes
84.  * return: the root of the generated huffman tree
85.  */
86. HuffTree GenerateHuffmanTree(HuffNodePtr *Q_leaf, int n) {
87.     vector<HuffNodePtr> Q_nonleaf;  // generalized nodes as non-leaves
88.     int p = n, q = 0;  // p - number of nodes in Q_leaves q - number of nodes in
    Q_numleaf
89.     int i = 0, j = 0; // i - index of the current element in Q_leaf j - index of
    the current element in Q_nonleaf
90.     HuffTree root;
91.     HuffNodePtr temp;
92.     while (p != 0 || q != 0) {
93.         /* if only 1 node left */
94.         if (p == 1 && q == 0) {
95.             root = Q_leaf[i];
96.             break;
97.         }
98.         if (p == 0 && q == 1) {
99.             root = Q_nonleaf[j];
100.             break;
101.         }
102.         /* pick 2 from the non-leaf nodes and add them up to make new node */

103.         if (q >= 2 && (p == 0 || Q_leaf[i]->f > Q_nonleaf[j+1]->f)) {
104.             temp = new HuffNode;
105.             temp->f = Q_nonleaf[j]->f + Q_nonleaf[j+1]->f;
106.             temp->Left = Q_nonleaf[j];
107.             temp->Right = Q_nonleaf[j+1];
108.             Q_nonleaf.push_back(temp);
```

```
            j+=2; q--;
        }
        /* pick 2 from the leaf nodes and add them up to make new node */
        else if (p >= 2 && (q == 0 || Q_leaf[i+1]->f < Q_nonleaf[j]->f)) {
            temp = new HuffNode;
            temp->f = Q_leaf[i]->f + Q_leaf[i+1]->f;
            temp->Left = Q_leaf[i];
            temp->Right = Q_leaf[i+1];
            Q_nonleaf.push_back(temp);
            i+=2; p-=2; q++;
        }
        /* pick 1 from leaf nodes and and 1 from non-leaf nodes */
        else {
            temp = new HuffNode;
            temp->f = Q_leaf[i]->f + Q_nonleaf[j]->f;
            temp->Left = Q_leaf[i];
            temp->Right = Q_nonleaf[j];
            Q_nonleaf.push_back(temp);
            i++; j++; p--;
        }
    }
    return root;
}

/* This function is used to calculated the weight of all nodes */
/* paramater root: the curent Huffnode pointer */
void WPL(HuffTree root) {
    if (root == NULL) return;
    wpl += root->f;
    WPL(root->Left);
    WPL(root->Right);
    delete root; // release the space
}

int main(void) {
    int n,m;

    cin>>n;  // number of nodes

    HuffNodePtr* Q_leaf = new HuffNodePtr[n];  // initial nodes as leaves
    map<char,int> freq;

    /* input and initialize */
    for (int i = 0; i < n; i++) {
        Q_leaf[i] = new HuffNode;
        cin >> Q_leaf[i]->c >> Q_leaf[i]->f;
```

```
        freq[Q_leaf[i]->c] = Q_leaf[i]->f;  // use a map for the afterwords ma
    tch
        Q_leaf[i]->Left = Q_leaf[i]->Right = NULL;
    }

    /* sort by the frequency */
    sort(Q_leaf,Q_leaf+n,cmpQ);

    /* generate huffman tree */
    HuffTree Huffroot = GenerateHuffmanTree(Q_leaf,n);

    /* encode and calulate the wpl*/
    WPL(Huffroot);
    wpl -= Huffroot->f;  // the root->f should be excluded

    /* judge */
    char ci;
    string* code = new string[n];
    int pseudo_wpl;
    cin>>m;
    for (int r = 0; r < m; r++) {
        /* calculate the weighted path length of the case */
        pseudo_wpl = 0;
        for (int i = 0; i < n; i++) {
            cin>>ci>>code[i];
            pseudo_wpl += freq[ci]*code[i].length();
        }
        /* if not equal to the calculated minimal wpl */
        if (pseudo_wpl != wpl) {
            cout<<"No"<<endl;
            continue;
        }
        else {
            /* use trie tree to judge if all the nodes are the leaf-nodes */
            bool flag = true;
            TrieTree Trieroot = createTrieNode();  // an empty root
            for (int i = 0; i < n; i++)
                /* insert and judge if the node is a leaf-node */
                if (!insert(Trieroot,code[i])) {
                    flag = false;
                    break;
                }
            deleteTrieTree(Trieroot);
            if (flag) cout<<"Yes"<<endl;
            else cout<<"No"<<endl;
        }
    }
```

```
    delete []code;

    return 0;
}
```

- **PR5_3**

```
#include<iostream>
#include<vector>
#include<map>
#include<algorithm>

using namespace std;

typedef struct HuffNode {
    char c;  // the character
    int f, w;  // the frequency and the weight
    HuffNode *Left, *Right;  // the left and right son
} *HuffNodePtr;  // the pointer
typedef HuffNodePtr HuffTree;

/* the struct of binary heap used as a priority queue*/
typedef struct HeapStruct {
    int size,capacity;
    HuffNodePtr *Elements;
} *Heap;

long long wpl = 0;  // the minimal weighted path length

/* the compare function for codes */
bool cmpS(string x,string y) {
    return x.length()<y.length();
}

/* This function is used to initialize a min-heap
 * paramater m: the necessary capacity
 * paramater H: the heap pointer
 * return H: the heap pointer
 */
Heap initialize_heap(int m, Heap H) {
    H = new HeapStruct;
     H->capacity = m+1;  /* the index should be from 1 to m */
     H->size = 0;  /* there is no element in the heap currently */
     H->Elements = new HuffNodePtr[m+1]; /* allocate the memory for elements */

    if (H == NULL) {
```

```
39.         printf("Out of space.");
40.         exit(0);
41.     }
42.     H->Elements[0] = new HuffNode;
43.     H->Elements[0]->f=-1;  /* put a negative integer at elements[0] as the minim
    um value */
44.     H->Elements[0]->Left = H->Elements[0] ->Right = NULL;
45.     return H;
46. }
47.
48. /* This function is used to insert a node into the heap
49.  * paramater x: the new node pointer
50.  * paramater H: the heap pointer
51.  */
52. void insert(HuffNodePtr x, Heap H){
53.     int i;
54.     H->size++;  /* expand the size of the heap */
55.     for (i = H->size; (H->Elements[i / 2])->f > x->f; i /= 2)  /* compare the va
    lue */
56.         H->Elements[i] = H->Elements[i / 2];  /* percolate up each vertex whose
    value is less than x's value */
57.     H->Elements[i] = x;  /* place the new vertex */
58. }
59.
60. /* This function is used to delete the minimal node from the heap
61.  * paramater H: the heap pointer
62.  */
63. void deleteMin(Heap H) {
64.     int i,child;
65.     HuffNodePtr x;
66.     x = H->Elements[H->size--];  /* shrink the size of the heap and put the last
     vertex on element[1] */
67.     for (i = 1; i * 2 <= H->size; i = child) {
68.         child = i*2;
69.         if (child + 1 <= H->size && (H->Elements[child + 1])->f < (H->Elements[c
    hild])->f)
70.             child++;  /* pick the smaller children */
71.         if ((H->Elements[child])->f < x->f)
72.             H->Elements[i] = H->Elements[child];  /* percolate down each child w
    hose value is larger than x's value */
73.         else break;
74.     }
75.     H->Elements[i] = x;  /* replace the last vertex */
76. }
77.
78. /* This function is used to generate a huffman tree
79.  * paramater H: the heap pointer
80.  * paramater n: number of nodes
```

```
 * return: the root of the generated huffman tree
 */
HuffTree GenerateHuffmanTree(Heap H, int n) {
    HuffNodePtr temp;
    int ite = H->size-1; // the iteration number
    for (int i = 0; i < ite; i++) {
        /* find the two min value and delete them from the heap*/
        HuffNodePtr x = H->Elements[1];
        deleteMin(H);
        HuffNodePtr y = H->Elements[1];
        deleteMin(H);
        /* create a new node as their parent and insert it into the heap */
        temp = new HuffNode;
        temp->f = x->f + y->f;
        temp->Left = x;
        temp->Right = y;
        insert(temp,H);
    }
    /* the last element in the heap is the root of the huffman tree */
     return H->Elements[1];
}

/* This function is used to calculated the weight of all nodes */
/* paramater root: the curent Huffnode pointer */
void WPL(HuffTree root) {
    if (root == NULL) return;
    wpl += root->f;
    WPL(root->Left);
    WPL(root->Right);
    delete root; // release the space
}

int main(void) {
    int n,m;

    cin>>n;  // number of nodes

    map<char,int> freq;

    /* input and initialize */
    Heap H;
    H = initialize_heap(n,H);
    HuffNodePtr temp;
    for (int i = 0; i < n; i++) {
        temp = new HuffNode;
        cin >> temp->c >> temp->f;
```

```
        temp->Left = temp->Right = NULL;
        freq[temp->c] = temp->f;
        insert(temp,H);
    }

    /* generate huffman tree */
    HuffTree root = GenerateHuffmanTree(H,n);

    /* encode and calulate the wpl*/
    WPL(root);
    wpl -= root->f;

        /* judge */
    char ci;
    string* code = new string[n];
    int pseudo_wpl;
    cin>>m; // number of cases
    for (int r = 0; r < m; r++) {
        /* calculate the weighted path length of the case */
        pseudo_wpl = 0;
        for (int i = 0; i < n; i++) {
            cin>>ci>>code[i];
            pseudo_wpl += freq[ci]*code[i].length();
        }
        /* if not equal to the calculated minimal wpl */
        if (pseudo_wpl != wpl) {
            cout<<"No"<<endl;
            continue;
        }
        else {
            /* judge if all the nodes are the leaf-nodes */
            sort(code,code+n,cmpS);  // sort the codes in increasing length
            bool flag = true;
            for (int i = 0; i < n-1; i++)
                for (int j = i+1; j < n; j++) {
                        /* if one code is the prefix of another then it can 鈥
    槏 be a leaf node*/
                    if (code[i] == code[j].substr(0,code[i].length())) {
                        flag = false;
                        break;
                    }
                }
            if (flag) cout<<"Yes"<<endl;
            else cout<<"No"<<endl;
        }
    }
```

```
172.     delete []code;
173.
174.     return 0;
175. }
```

Appendix B Declaration

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

Signature: