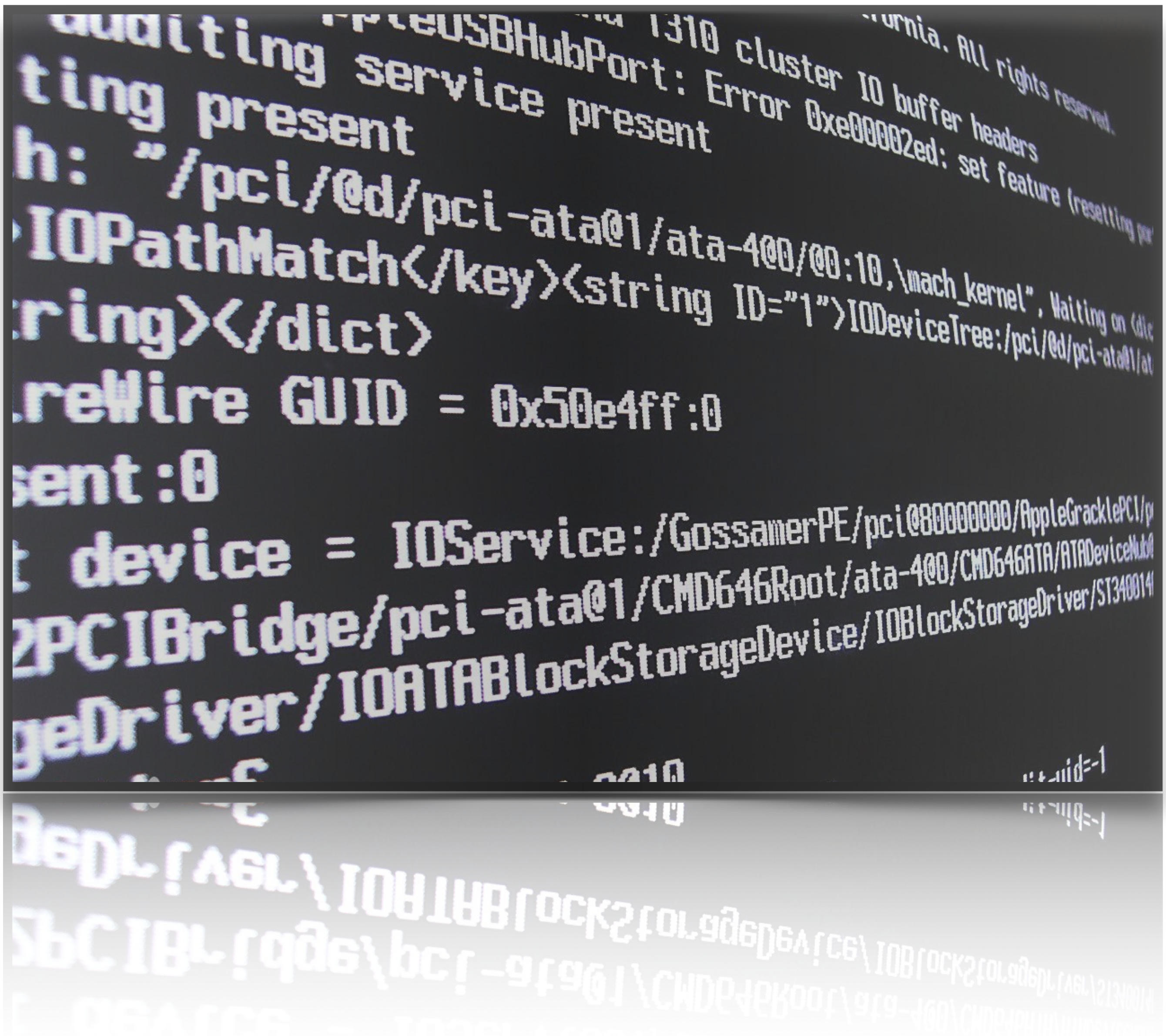
Project4 Huffman Codes

Group 19

Research - December 29, 2014

Group members :

Huang Zhao-yang(3130000696), Wang Chao-qi(3130102015), Zhang Han-yi(3130102345)



Chapter 1 Introduction

Huffman code is an optimal prefix code found using the algorithm developed by David A. Huffman while he was a Ph.D student at MIT, and published in the 1952 paper"A Method for the Construction of Minimum-Redundancy Codes".[1]

As we can see that, the Huffman codes are not unique(For instance, given a string "aaaxuaxz", we can observe that the frequencies of the of the characters 'a', 'x', 'u' and 'z' are 4, 2, 1 and 1, respectively. We may either encode the symbols as {'a'=0, 'x'=10, 'u'=110, 'z'=111}, or in another way as {'a'=1, 'x'=01, 'u'=001, 'z'=000}, both compress the string into 14 bits.). So, there comes the problem that if someone gives you the codes, and he/she says it's a type of Huffman codes, how can you make sure what he/she says is right or wrong.

It's a waste of time and difficult for people to judge the code whether it is Huffman code or not if it can be implemented by a program. So, we decide to write a program to make the task easier.

Chapter 2 Data Structure/Algorithm specification

2.1 IMPLEMENTATIO

First, we have to make sure what properties does Huffman code have.

(Define: **Wij** the weight of the node in **i**th level, and **j**th element on this level from left to right; And each node's weight is the sum of its children' weight, leaf node's weight is equal to the frequency of the character which is represented by it. e.g: the root's weight is W00).

1.We can represent the code as a tree.

2.If we define: the root is on the level 0, and its child is on the level 1, and each node has a weight $\mathbf{W_{ij}}$; Now, if there exist two nodes **A,B,** whose weights are $\mathbf{W_{ij}}$ and $\mathbf{W_{mn}}$, ($\mathbf{i > m}$), then $\mathbf{W_{mn} >= W_{ij}}$;

3.After we adjust each node(NULL represents its weight is 0) to ensure that its left-child's weight is not smaller than its right-child's weight, if we travel the tree level by level(from top(root) to bottom), and on each level we travel from left to right, then we can get a sequence which is **NON INCREASING**.

Below is the proof of these three properties.(1-3).

PREMISE: Huffman Codes are optimum code.

PROPERTY-1: it's obvious.

PROPERTY-2: If not, that is $\mathbf{W_{mn} < W_{ij}}$ ($\mathbf{i > m}$), then if exchange this two nodes's position, the total cost will be decreased by $\mathbf{W_{ij}}$ - $\mathbf{W_{mn}}$. So, the original code is not optimum, CONTRADICTION!

PROPERTY-3:Proof by induction:

3.1: on the level **1**. At most two nodes, so, after adjustment, we can ensure that left > right.

3.2: assume on level **N-1** (N>=2), all the elements from left to right is NON INCREASING.

3.3

3.3.1 :On level **N** (N>=2), if exist 2 nodes:A,B(left to right and near to each other), if A >= B is wrong, then A < B.

3.3.2 :Obviously, A,B do not share the same parent.

3.3.3 :A is on the left of B, so A's parent's weight is bigger than B's parent's(from 3.2)(and each node has zero of two children).

3.3.4 So, A and B must have brother/sister, So, this case is C > B > A > D, ([A,C],[B,D] share the same parent). However, when we build the Huffman code, each time we choose two smallest nodes. So, here, it's [A,D],[B,C] share the same parent instead of [A,C],[B,D]. CONTRADICTION!

3.4:FROM 3.1, 3.2 ,3.3, property-3 is proofed.

Below are the data structure and algorithm specifications.

DATA STRUCTURE	

Data Structure I	`struct TreeNode`
	`{`
	`    TreeNode *left,*right;`
	`    int data;`
	`    int symbol;`
	`};`

Specification I	Name: TreeNode 1.Left and Right are pointers pointing to the two children of the Node. 2.Data is to store the Weight of the node. 3.Symbol is to store the character's ASCII number.

Data Structure II	`struct QueNode`
	`{`
	`    int data,symbol;`
	`};`

Specification II	Name: QueNode This structure is to store the inputs characters and their frequency. 1.Data is to store the frequency. 2.Symbol is to store the character's ASCII number.

Below is the Algorithm Specification.

Algorithm(Cnt)	
OVERALL SOLUTION	**Firstly**, we read in the inputs and store them into the QUE. **Secondly**, insert each character to the TREE the path is decided by their code. **Thirdly**, after we insert one character to the TREE, judge whether the node in the end is a leaf node or not, if not, return FALSE.(check prefix) **Then**, we adjust each node's children, to ensure that the left child's weight is bigger than the right child's weight. **Finally,** we travel the tree from level by level. If find the weight is NON INCREASING, than return TRUE, else return FALSE.
Insertion	(see code below)

```
//left 0,  right 1
int InsertTreeNode(TreeNode *node,int sym,char
num[],int depth,int aimdepth)
{
    int ret=0;
    if(arrived the destination)
    {
        if(the node is occupied)
            return 0;
        else
            Insert the character into the node;
        return 1;
    }
    if(Not arrived the destination)
    switch(num[depth])
    {
        case '0':
            if(node's left-child is NULL)
                Malloc left-child;
            ret = InsertTreeNode(node-
                >left,sym,num,depth+1,aimdepth);
            break;
        case '1':
            if(node's right-child is NULL)
                Malloc right-child;
            ret = InsertTreeNode(node-
                >right,sym,num,depth+1,aimdepth);
            break;
    }
    Mark the Node;
    //the Node is occupied.
    return ret;
}
```

Algorithm(Cnt)	
Specification	We insert the character recursively. 1.Depth is the current depth. 2.AimDepth is the destination’s depth. 3.If Depth == Aimdepth, then insert it into the node, if the node is occupied, return FALSE. 4.If Depth < Aimdepth, then check the next bits, if it’s 0, then insert it into the left, else insert it into the right. 5. Mark the node we go through, which is convenient for us to judge whether there exist two codes A,B and A is the prefix of B.
Adjust_Tree_By_Travelling	(code below)
Specification	This algorithm is to compute the weight of the node(by DFS) and adjust the two children according to the their weights. Two Operations: 1. NODE->DATA = LEFTCHILD->DATA + RIGHTCHILD->DATA; 2. If(LEFTCHILD->DATA < RIGHTCHILD->DATA) SwapTwoChildren();

Adjust_Tree_By_Travelling

```
int TravelTree(TreeNode *node,int depth)
{
    Count[depth]++;
    //Count is used to calculate amount of nodes in
      each depths
    if(exist prefix condition)
        return 0;
    if(node->left!=NULL && node->right!=NULL)
    {
        if(TravelTree(node->left,depth+1)==0)
            return 0;
        if(TravelTree(node->right,depth+1)==0)
            return 0;
        node->data=node->left->data+node->right->data;
        if(left-child's weight < right-child's
            weight)
            SwapTwoChild();
        return 1;
    }
    if(it's single child)
        return 0;
    else
        return 1;
}
```

Algorithm(Cnt)

JudgeTree

```
int JudgeTree(TreeNode *Root,int &QueLen)
{
    int i;
    Compute NumberOfNodes on each level;
    for(i=2;i<=MaxDepth;i++)
        Count[i]=Count[i-1]+Count[i];
    //Compute the prefix sum of Count to get the array
    GetTheSequence_ARRAY();
    for(i=1;i<QueLen*2-1;i++)
        if(a[i]<a[i+1]) return 0;
    //judge the array if the data is increasing or
equal
    return 1;
}
```

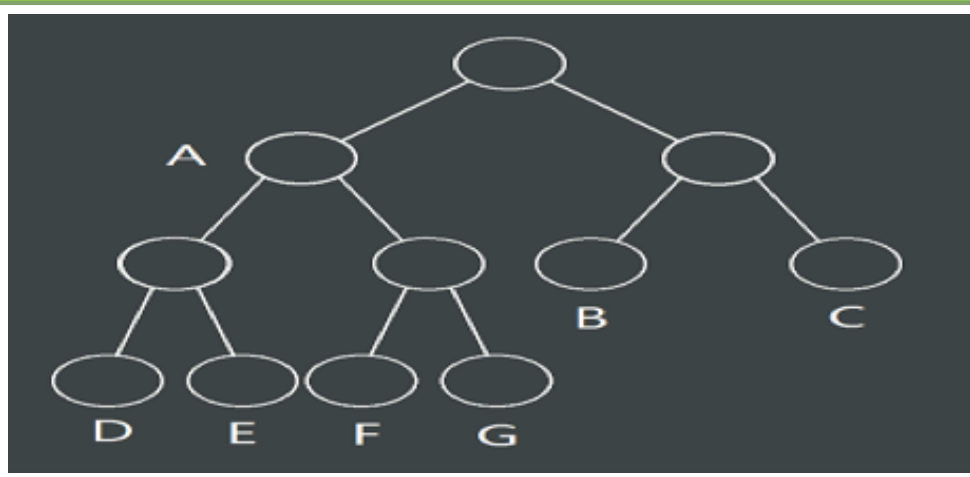
Specification

1.Compute the number of nodes on each level and store the data in array count[];
2.Compute the prefix sum (Fi = Fi + Fi-1, include it self);
3.After we know the sum of nodes on each level, we build an array to store the weight of the nodes(from top level(root) to bottom level, on each level, from left to right).
4.Travel this array, and judge whether a[i] >= a[i+1], if not, then return FALSE.

Chapter 3 Testing Results

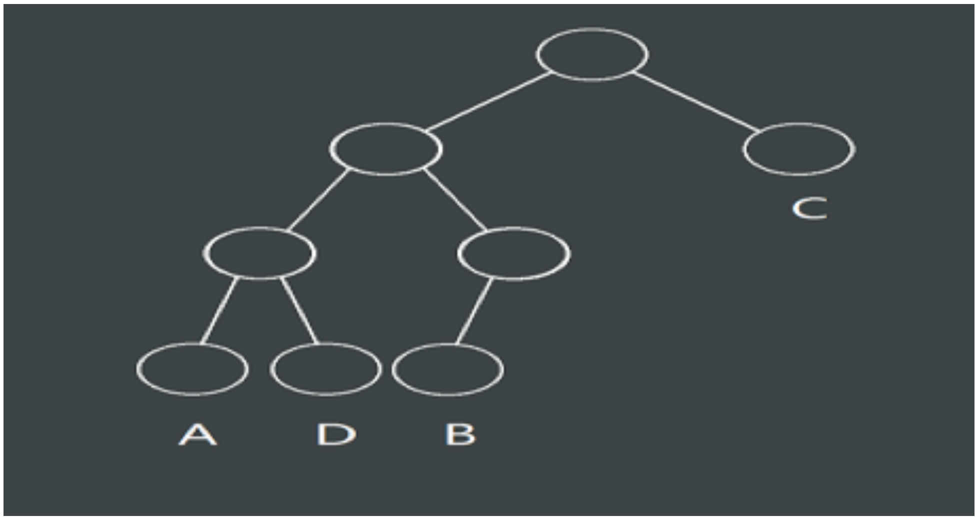
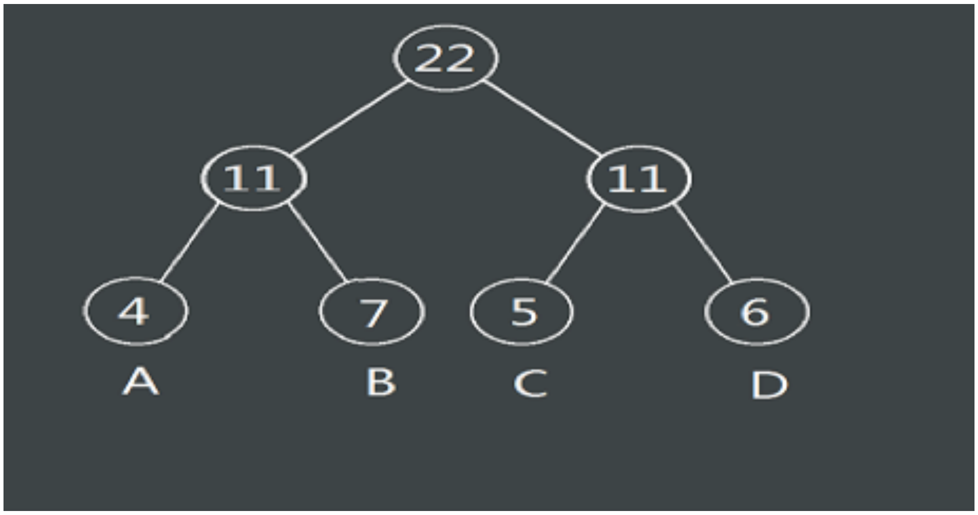
TESTING CASES AND SPECIFICATION

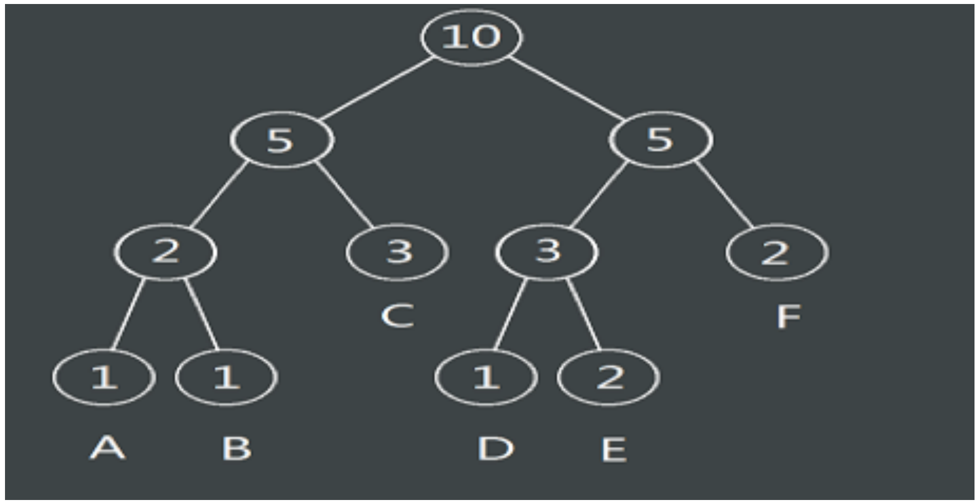
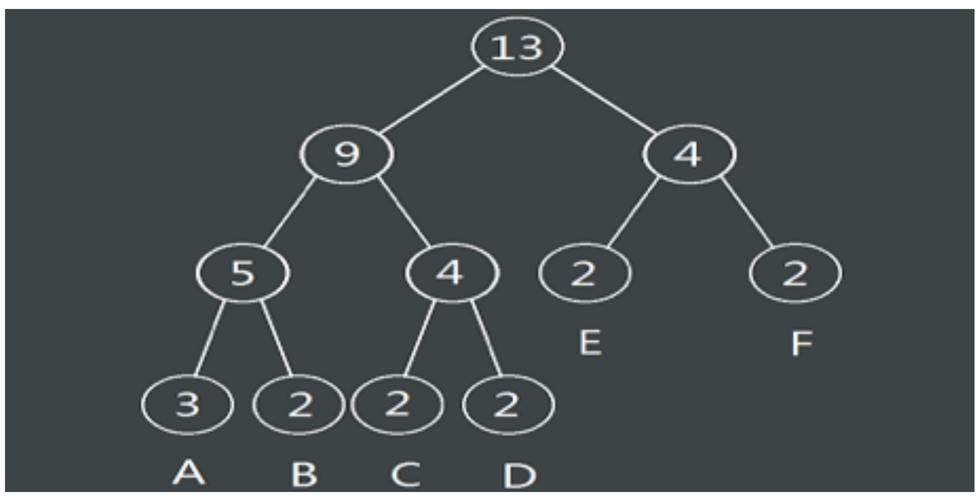
Case 1

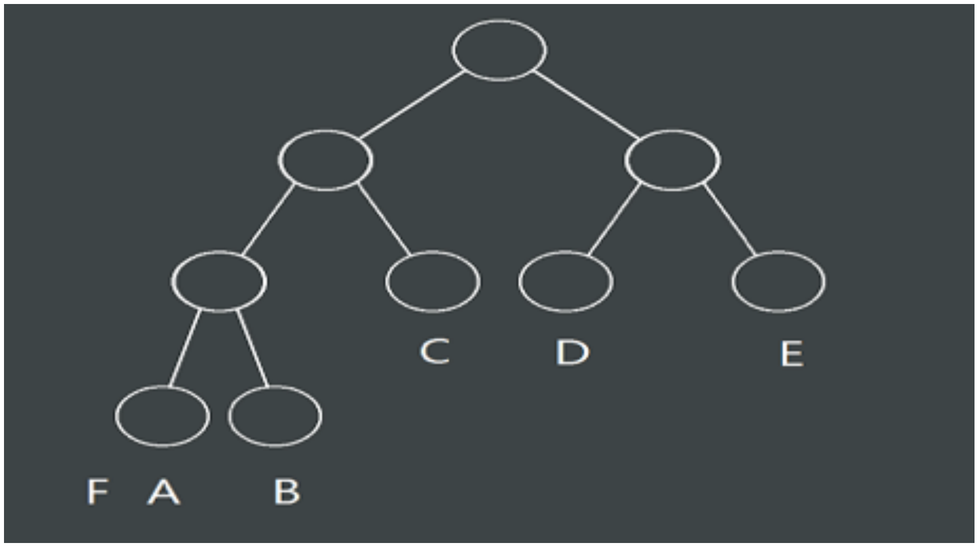
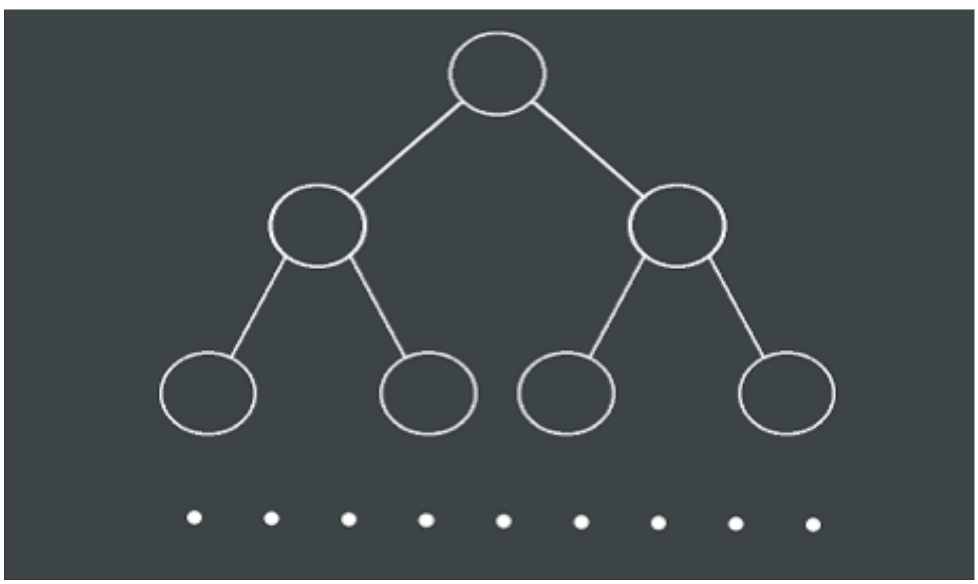


Specification

Case1:(data1.in)
one code is a prefix of another.
PASS.

TESTING CASES AND SPECIFICATION	
Case 2	
Specification	Case2:(data2.in) An invalid Huffman tree: there is a node which has exactly one child. PASS.
Case 3	
Specification	Case 3(data3.in, data5.in): It’s an optimal code but not a Huffman Code. PASS.

TESTING CASES AND SPECIFICATION	
Case 4	
Specification	Case 4:(data4.in) There is a higher frequency node in higher level in a Huffman tree. PASS.
Case 5	
Specification	Case 5(data8.in): One code is distributed to more than one character. PASS.

TESTING CASES AND SPECIFICATION	
Case 6	
Specification	Case 6:(data7.in) This is a case to test the program security. Some code is far too long and we are supposed to recognize it. Otherwise it will take too many memories. PASS.
Case 7	
Specification	Case 7(data6.in): There is one more test case to check different Huffman codes that are equivalent. PASS.

TESTING CASES AND SPECIFICATION	
Case 8	It's a a big data. PASS.

Chapter 4 Analysis and Comments

Specification:

L - sum of length of path of all nodes.

N - amount of the given characters.

Time Analysis:

We should spend O(L) time in receiving information from input and building the given tree, which can't be omitted. The following operation on the tree only need O(N) time, because the number of the nodes of the tree is exactly 2n-1, and we can judge the tree can't be Huffman Tree when the nodes overflowed.

In conclusion, the time complexity of this program is O(L).

Space Analysis:

We allocated constant space for each node in the tree, so the tree cost O(N) space. Then we picked up each nodes into an array, so the space of the array is also O(N).

In conclusion, the space complexity of this program is O(N).

Comments:

After several discussions, we've reached our common agreement. As you can see, the input model should cost O(L) time, so we got the optimal time complexity, and to store a given tree, we must allocate no less than O(N) space, so we got the optimal space complexity. For my part, we've done what we can do, and it's an excellent algorithm.

Appendix Source code:

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

typedef struct QueNode QueNode;
typedef struct TreeNode TreeNode;
struct TreeNode
{
    TreeNode *left,*right;
    int data;
    int symbol;
};

struct QueNode
{
    int data,symbol;
};

TreeNode *Root;
QueNode Que[201];
int QueLen,BaseQueLen,MaxDepth,TotalSymbol;
int Weight[201],a[201],Count[201];


/*--------------Allocate Space---------------------------------*/
TreeNode *NewTreeNode(TreeNode *left,TreeNode *right,int data,int symbol);
/*--------------Allocate Space---------------------------------*/

/*--------------Operations About Tree--------------------------*/
int InsertTreeNode(TreeNode *node,int sym,char num[],int depth);
void FreeTree(TreeNode *node);
int TravelTree(TreeNode *node,int depth);
/*--------------Operations About Tree--------------------------*/

TreeNode *NewTreeNode(TreeNode *left,TreeNode *right,int data,int symbol)
{
    TreeNode *node=(TreeNode*)malloc(sizeof(TreeNode));
    node->left=node->right=NULL;
    node->data=data;
    node->symbol=symbol;
    return node;
```

```
}
//Allocate space with given elements

void Initialize()
{
    memset(Count,0,sizeof(Count));
    TotalSymbol=64;
    Root=NewTreeNode(NULL,NULL,0,63);
    MaxDepth=0;
}
//Initialize the corresponding structures

void FreeTree(TreeNode *node)
{
    if(node==NULL) return;
    FreeTree(node->left);
    FreeTree(node->right);
    free(node);
}
//Free the space of the node of the tree after a query

void Clear()
{
    FreeTree(Root);
}
//Clear all the redundancy after a query

int TransferCharToInt(char c)
{
    if(c<='9'&&c>='0') return c-'0';    //char of '0'~'9' will be numbered
0~9
    if(c<='z'&&c>='a') return c-'a'+10; //char of 'a'~'z' will be numbered
10~35
    if(c<='Z'&&c>='A') return c-'A'+36; //char of 'A'~'Z' will be numbered
36~61
    if(c=='_') return 62;               //char of '_'     will be numbered
62
    return 0;
}
//Transfer a char to a corresponding int

int OverallInput(QueNode *Que,int &QueLen)
{
    int i,fre,n;
    char sym;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        getchar();
        scanf("%c %d",&sym,&fre);
```

```
        Que[i].data=fre;
        Que[i].symbol=TransferCharToInt(sym);   //put the elements into Que
        Weight[Que[i].symbol]=fre;              //store the frequency to 
corresponding symbol
    }
    QueLen=n;
    scanf("%d",&n);
    getchar();
    return n;
}
//Get the frequency of each char and store the given elements into Que 
array

/
*---------------------------------------------------------------------------
*/
/*      This function will insert the given char sym into the tree, whose
*/
/*  path is represented by string num.
*/
/*      error happens when 0 has been returneed
*/
/*      node is current node that we touched
*/
/*      sym is the char that we should insert into this tree this time
*/
/*      num represent the path of the char should be inserted
*/
/*      depth is current depth of this tree.
*/
/*      aimdepth is the goal depth that we should target
*/
/
*---------------------------------------------------------------------------
*/
int InsertTreeNode(TreeNode *node,int sym,char num[],int depth,int 
aimdepth)
{
    int ret=0;
    if(depth==aimdepth)//if we hit the target, we'll put this sym in this 
node
    {
        if(node->symbol<63&&node->symbol>-1) return 0;
            //if this node was covered with a valuable sym, an error 
happens
        node->symbol=sym;
        node->data=Weight[sym];
        return 1;
            //sym has covered this node successfully
    }
```

```
    switch(num[depth])
    {
        case '0':   //we'll go to the left child of this node when meet '0'
            if(node->left==NULL)    //if the left child of this node is
NULL, we'll create a new node
                node->left=NewTreeNode(NULL,NULL,0,-1);
            ret=InsertTreeNode(node->left,sym,num,depth+1,aimdepth);
            break;
        case '1':   //we'll go to the right child of this node when meet
'1'
            if(node->right==NULL)   //if the right child of this node is
NULL, we'll create a new node
                node->right=NewTreeNode(NULL,NULL,0,-1);
            ret=InsertTreeNode(node->right,sym,num,depth+1,aimdepth);
            break;
    }
    if(node->symbol==-1) node->symbol=TotalSymbol++;
        //if this node wasn't covered with a symbol, make a symbol and
cover it
    return ret;
}

void GetCharAndString(int &sym,char *num,int &len)
{
    char ch;
    ch=getchar();
    sym=TransferCharToInt(ch);
    getchar();
    ch=getchar();
    len=0;
    while(ch=='0'||ch=='1')
    {
        num[len++]=ch;
        ch=getchar();
    }
    num[len]='\0';
}
//this function serve under the Input function,
//get the given sym,num and count the length of the num

int Input(TreeNode *Root,int CharNum)
{
    int sym;
    char num[65];
    int len,i;
    for(i=0;i<CharNum;i++)
    {
        GetCharAndString(sym,num,len);
        if(InsertTreeNode(Root,sym,num,0,len)==0) return 0;
        if(len>MaxDepth) MaxDepth=len;
```

```
    }
    MaxDepth++;
    return 1;
}
//this function is used to get the input information
//and insert the given sym into the tree

void SwapChild(TreeNode *node)
{
    TreeNode *tmp=node->left;
    node->left=node->right;
    node->right=tmp;
}
//this function is used to swap the child of the given node

/
*--------------------------------------------------------------------------
*/
/*      This function is used to count the amount of nodes of the tree
*/
/*  in each depths, judge the single child&prefix condition, and calculate
*/
/*  data of the internal nodes.
*/
/*      depth is current depth of this tree
*/
/
*--------------------------------------------------------------------------
*/

int TravelTree(TreeNode *node,int depth)
{
    Count[depth]++;     //Count is used to calculate amount of nodes in
each depths
    if(node->symbol<63) //judge prefix condition
    {
        if(node->left!=NULL || node->right!=NULL) return 0;
        return 1;
    }
    if(node->left!=NULL && node->right!=NULL)
    {
        if(TravelTree(node->left,depth+1)==0) return 0;
        if(TravelTree(node->right,depth+1)==0) return 0;
        node->data=node->left->data+node->right->data;
            //data of this node is the sum data of its child
        if(node->left->data > node->right->data) SwapChild(node);
            //Ensure the data of the left child is larger than the right
child
        return 1;
    }
```

```
    if(node->left!=NULL || node->right!=NULL) return 0;
            //judge single child condition
    return 1;   //travel this tree successfully
}

void GetArray(TreeNode *node,int depth)
{
    if(node==NULL) return;
    a[Count[depth]--]=node->data;
    GetArray(node->left,depth+1);
    GetArray(node->right,depth+1);
}
//Put the treenode into the array

int JudgeTree(TreeNode *Root,int &QueLen)
{
    int i;
    if(TravelTree(Root,1)==0) return 0;
    //Travel this tree
    for(i=2;i<=MaxDepth;i++)
        Count[i]=Count[i-1]+Count[i];
    //get the prefix sum of Count to get the array
    GetArray(Root,1);
    for(i=1;i<QueLen*2-1;i++)
        if(a[i]<a[i+1]) return 0;
    //judge the array if the data is increasing or equal
    return 1;
}

int main()
{
    int rep;
    rep=OverallInput(Que,QueLen);
    for(;rep>0;rep--)
    {
        Initialize();                //initialize all the elements
        if(Input(Root,QueLen)==1)    //work with the algorithm
        {
            if(JudgeTree(Root,QueLen)==1)
                printf("Yes\n");
            else printf("No\n");
        } else printf("No\n");
        Clear();                     //clear the redundant elements
    }
    return 0;
}
```

References:

[1]WIKIPEDIA ,"Huffman coding","http://en.wikipedia.org/wiki/Huffman_coding#Compression", First Paragraph.

Author List:

王超奇 Reporter

黄昭阳 Programmer

张涵祎 Tester

Declaration:

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

Signatures:

王超奇 3130102015

黄昭阳 3130000696

张涵祎 3130102345