**1017: 进制转换问题**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 48  Solved: 40  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1017)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1017)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1017)]

**Description**

**利用栈类，实现将一个十进制整数转化为二至九进制之间的任一进制输出。函数原型void Convert(int num,int d)，将十进制num转换为d进制，在函数体内实现输出即可。**

若num为正整数，则算法规则如下：

循环（直至num为0）

{   将num%d压栈；

   num/=d;

}

循环（栈非空）

{

    栈顶元素弹栈，输出（进制的高位）

}

要求程序能够处理正、负整数（含0）

**Input**

输入：6 2

输出:110

输入：90 8

输出：132

**Output**

**Sample Input**

6 2

**Sample Output**

110

#include <iostream>

using namespace std;

class jz

{

private:

int data[20];

int top;

public:

jz(){top=-1;}

~jz(){}

bool Empty();

void Ennum(int num,int d);

int Denum();

};

void jz::Ennum(int num,int d)

{

while(num)

{

data[++top]=num%d;

num/=d;

}

}

bool jz::Empty()

{

return top==-1;

}

int jz::Denum()

{

int x;

x=data[top--];

return x;

}

void Convert(int num,int d)

{

jz a;

int x;

a.Ennum(num,d);

while(!a.Empty())

{

x=a.Denum();

cout<<x;

}

}

int main()

{

int num,d;

cin>>num>>d;

Convert(num,d);

}

**1018: 表达式求值**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 23  Solved: 18  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1018)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1018)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1018)]

**Description**

**要求输入一个以字符串方式存储的表达式，利用栈，能够计算表达式的值。表达式中允许的运算符为+，-，\*，/，%，(，)。**

**函数原型： int CalValue(char \*s); s存储表达式，函数返回值为表达式的值，表达式中的操作数为正整数。**

输入： 15+(3-5)\*7

输出：1

**Input**

**Output**

**Sample Input**

5+7\*(25-12)/2-3

**Sample Output**

47

#include<iostream>

#include<cmath>

using namespace std;

char pre[7][7]={

/\* + - \* / ( ) % \*/

{'>','<','<','<','>','>','>'},

{'<','>','<','<','>','>','>'},

{'>','>','<','<','>','>','<'},

{'>','>','<','<','>','>','<'},

{'>','>','>','>','>','<','>'},

{'<','<','<','<','<','>','<'},

{'>','>','<','<','>','0','='}

}; /\*运算符之间的优先级制作成一张表格\*/

class bds

{

private:

char a[1000];

double b[1000];

int len\_a,len\_b;

public:

bds();

~bds(){};

void Push\_a(char s);

void Push\_b(double num);

char Pop\_a();

double Pop\_b();

char Get\_a();

double Get\_b();

void Display();

};

bds::bds()

{

len\_a=0;

len\_b=0;

a[len\_a++]='+';

b[len\_b++]=0;

b[len\_b++]=0;

}

void bds::Push\_a(char s)

{

if(len\_a<1000)

a[len\_a++]=s;

// cout<<"Pu: "<<a[len\_a-1]<<" ";}

else throw "Over flow";

}

void bds::Push\_b(double num)

{

if(len\_b<1000)

b[len\_b++]=num;

else throw "Over flow";

}

char bds::Pop\_a()

{

char x;

if(len\_a==0) throw "Empty";

x=a[len\_a-1];

len\_a--;

return x;

}

double bds::Pop\_b()

{

double x;

if(len\_b==0) throw "Empty";

x=b[len\_b-1];

len\_b--;

return x;

}

char bds::Get\_a()

{

char x;

x=a[len\_a-1];

return x;

}

double bds::Get\_b()

{

double x;

x=b[len\_b-1];

return x;

}

void bds::Display()

{

double x,y,num;

char s;

int x1,y1;

while(len\_b>2)

{

x=Pop\_b();

y=Pop\_b();

s=Pop\_a();

switch(s)

{

case '+': num=x+y;break;

case '-': num=y-x;break;

case '/': num=y/x;break;

case '\*': num=y\*x;break;

case '%': {y1=y;x1=x;num=y1%x1;break;}

case '(': s=Pop\_a();

case ')': num=y-x;break;

}

Push\_b(num);

}

x1=Pop\_b();

cout<<x1<<endl;

}

char youxianji(char a, char b)

{

int j,i;

switch(a){

case '+': i=0; break;

case '-': i=1; break;

case '\*': i=2; break;

case '/': i=3; break;

case '(': i=4; break;

case ')': i=5; break;

case '%': i=6; break;

}

switch(b){

case '+': j=0; break;

case '-': j=1; break;

case '\*': j=2; break;

case '/': j=3; break;

case '(': j=4; break;

case ')': j=5; break;

case '%': j=6; break;

}

return pre[i][j];

}

int main()

{

char s[1000],s1;

int i,x1,y1,k,mark;

double num,x,y;

while(cin>>s)

{

bds A;

for( i=0 ; s[i]!=0 ; )

{

if(s[i]<48||s[i]>57)

{

if(youxianji(s[i],A.Get\_a())=='<')

{

x=A.Pop\_b();

y=A.Get\_b();

s1=A.Get\_a();

switch(s1)

{

case '+': num=x+y;A.Pop\_b();A.Pop\_a();break;

case '-': num=y-x;A.Pop\_b();A.Pop\_a();break;

case '/': num=y/x;A.Pop\_b();A.Pop\_a();break;

case '\*': num=y\*x;A.Pop\_b();A.Pop\_a();break;

case '%': {y1=y;x1=x;num=y1%x1;A.Pop\_b();A.Pop\_a();break;}

case '(': A.Push\_b(x);break;

case ')': A.Push\_b(x);break;

}

if(s1!='(')

A.Push\_b(num);

}

if(s[i]!=')')

A.Push\_a(s[i++]);

if(s[i]==')'&&s1=='(')

{ i++;A.Pop\_a();}

}

else

{

for(num=0,k=1,mark=0; s[i]>=48&&s[i]<=57||s[i]=='.'; i++)

if(s[i]=='.') mark=1;

else

{ if(mark==1)

num=num+(s[i]-48)/pow(10,k++);

else

num=num\*10+s[i]-48;

}

A.Push\_b(num);

}

}

A.Display();

}

return 0;

}

**1019: 顺序存储二叉树**

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 64  Solved: 34  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1019)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1019)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1019)]

**Description**

**顺序存储的二叉树，依次实现二叉树的三种递归遍历：先序、中序、后序；二叉树的先序非递归、层次遍历；计算二叉树的叶子个数、高度。部分代码已知，请勿改动，将其它代码补充完整，达到相应的输出结果。**

 //循环队列类  
const int QueueSize=100;  //循环队列的最大长度  
template <class T>        //定义模板类CirQueue  
class CirQueue  
{  
public:  
    CirQueue( );                 //构造函数，置空队  
    ~ CirQueue( );               //析构函数  
    void EnQueue(T x);           //将元素x入队  
    T DeQueue( );                //将队头元素出队  
    T GetQueue( );               //取队头元素（并不删除）  
    bool Empty( );               //判断队列是否为空  
 bool Full();  
private:  
    T data[QueueSize];           //存放队列元素的数组  
    int front, rear;    //队头和队尾指针，分别指向队头元素的前一个位置和队尾元素的位置  
};

 //功    能：初始化空队列  
 template <class T>  
CirQueue<T>::CirQueue( )   
{  
 front=rear=0;  
}

// 功    能：销毁队列  
template <class T>  
CirQueue<T>::~CirQueue( )  
{  
}

//功    能：元素x入队  
template <class T>   
void CirQueue<T>::EnQueue(T x)  
{  
    if (Full()) throw "Overflow";  
    rear=(rear+1) % QueueSize;   //队尾指针在循环意义下加1  
    data[rear]=x;                //在队尾处插入元素  
}

//功    能：队头元素出队，返回值为出队元素  
template <class T>   
T CirQueue<T>::DeQueue( )  
{  
    if (Empty()) throw "Downflow";   
    front=(front+1) % QueueSize;    //队头指针在循环意义下加1  
    return data[front];             //读取并返回出队前的队头元素，注意队头指针  
}  
                                   //指向队头元素的前一个位置  
// 功    能：获取队头元素  
template <class T>  
T CirQueue<T>::GetQueue( )  
{     
    int i;  
    if (Empty()) throw "Downflow";   
    i=(front+1) % QueueSize;  //注意不要给队头指针赋值  
    return data[i];  
}

// 功    能：判断队列是否为空,若空返回true，否则返回false  
template <class T>   
bool CirQueue<T>::Empty( )   
{  
    return front==rear;   
}  
//功   能：判断队列是否满，若满返回true，否则false  
template <class T>  
bool CirQueue<T>::Full()  
{  
 return (rear+1)%QueueSize==front;  
}

//栈类  
const int StackSize=100;  //10只是示例性的数据，可以根据实际问题具体定义  
template <class T>       //定义模板类SeqStack  
class SeqStack  
{  
public:  
    SeqStack( ) ;            //构造函数，栈的初始化  
 ~SeqStack( );            //析构函数  
    void Push(T x);          //将元素x入栈  
    T Pop( );                //将栈顶元素弹出  
    T GetTop( );          //取栈顶元素（并不删除）  
 bool Empty( );           //判断栈是否为空  
 bool Full();             //判断栈是否为满  
private:  
    T data[StackSize];      //存放栈元素的数组  
    int top;                //栈顶指针，指示栈顶元素在数组中的下标  
};

//初始化空栈  
template <class T>  
SeqStack<T>::SeqStack( )  
{  
 top=-1;  
}

//功    能：销毁栈  
template <class T>  
SeqStack<T>::~SeqStack( )  
{  
}

//功    能：元素x压栈  
template <class T>   
void SeqStack<T>::Push(T x)  
{  
    if (Full()) throw "Overflow";  
    top++;  
    data[top]=x;  
}  
// 功    能：栈顶元素弹栈  
template <class T>  
T SeqStack<T>::Pop( )  
{   
    T x;  
    if (Empty()) throw "Downflow";  
    x=data[top--];  
    return x;  
}

// 功    能：读取当前的栈顶元素  
template <class T>   
T SeqStack<T>::GetTop( )  
{  
 if (Empty())throw"Downflow";    
    return data[top];  
}

//功    能：判断栈是否为空  
template <class T>   
bool SeqStack<T>::Empty( )  
{  
 if(top==-1) return 1;  
 else return 0;  
}

//功    能：判断栈是否为满  
template <class T>   
bool SeqStack<T>::Full( )  
{  
 return top==StackSize-1;  
}

//顺序二叉树类  
const int TreeSize=100;  //最大长度  
template <class T>  
class BiTree{  
public:                 
 BiTree(T \*); //构造函数，初始化一棵二叉树，其前序序列由键盘输入  
    void PreOrder(int pos);     //前序遍历二叉树  
    void InOrder(int pos);      //中序遍历二叉树  
    void PostOrder(int pos);    //后序遍历二叉树  
    int CountLeaf(int pos);     //计算叶子结点总数  
 int Depth(int pos);         //计算高度  
 void PreOrder();            //非递归先序遍历    
 void LevelOrder();          //层次遍历  
private:  
    T data[TreeSize];

    //存储结点的元素值，从1下标开始,双亲下标为i，则左右孩子依次为2i，2i+1  
};

//constructor  
template <class T>  
BiTree<T>::BiTree(T \*str)  
{  
 strcpy(data+1,str);   
}

{  
 char str[80];  
 cin>>str;  
 BiTree<char> bt(str);  
 bt.PreOrder(0);  
 cout<<endl;  
 bt.InOrder(0);  
 cout<<endl;  
 bt.PostOrder(0);  
 cout<<endl;  
 bt.PreOrder();  
 cout<<endl;  
 bt.LevelOrder();  
 cout<<endl;  
 cout<<bt.CountLeaf(0)<<" "<<bt.Depth(0)<<endl;  
 return 0;  
}

**Input**

输入数据为ABD#C####EF############，表示数据在数组中的存储次序，#表示该结点为空结点

**Output**

输出数据各行依次为先序、中序、后序递归遍历，先序非递归（借助栈实现），层次遍历，叶子个数，高度

A B C E F D    
B E C F A D   
E F C B D A   
A B C E F D   
A B D C E F   
3 4

**Sample Input**

ABD#C####EF############

**Sample Output**

A B C E F D

B E C F A D

E F C B D A

A B C E F D

A B D C E F

3 4

#include<iostream>

#include<cstring>

using namespace std;

static int count=0;

static int c=0;

const int QueueSize=100; //循环队列的最大长度

template <class T> //定义模板类CirQueue

class CirQueue

{

public:

CirQueue( ); //构造函数，置空队

~ CirQueue( ); //析构函数

void EnQueue(T x); //将元素x入队

T DeQueue( ); //将队头元素出队

T GetQueue( ); //取队头元素（并不删除）

bool Empty( ); //判断队列是否为空

bool Full();

private:

T data[QueueSize]; //存放队列元素的数组

int front, rear; //队头和队尾指针，分别指向队头元素的前一个位置和队尾元素的位置

};

//功 能：初始化空队列

template <class T>

CirQueue<T>::CirQueue( )

{

front=rear=0;

}

// 功 能：销毁队列

template <class T>

CirQueue<T>::~CirQueue( )

{

}

//功 能：元素x入队

template <class T>

void CirQueue<T>::EnQueue(T x)

{

if (Full()) throw "Overflow";

rear=(rear+1) % QueueSize; //队尾指针在循环意义下加1

data[rear]=x; //在队尾处插入元素

}

//功 能：队头元素出队，返回值为出队元素

template <class T>

T CirQueue<T>::DeQueue( )

{

if (Empty()) throw "Downflow";

front=(front+1) % QueueSize; //队头指针在循环意义下加1

return data[front]; //读取并返回出队前的队头元素，注意队头指针

}

//指向队头元素的前一个位置

// 功 能：获取队头元素

template <class T>

T CirQueue<T>::GetQueue( )

{

int i;

if (Empty()) throw "Downflow";

i=(front+1) % QueueSize; //注意不要给队头指针赋值

return data[i];

}

// 功 能：判断队列是否为空,若空返回true，否则返回false

template <class T>

bool CirQueue<T>::Empty( )

{

return front==rear;

}

//功 能：判断队列是否满，若满返回true，否则false

template <class T>

bool CirQueue<T>::Full()

{

return (rear+1)%QueueSize==front;

}

//栈类

const int StackSize=100; //10只是示例性的数据，可以根据实际问题具体定义

template <class T> //定义模板类SeqStack

class SeqStack

{

public:

SeqStack( ) ; //构造函数，栈的初始化

~SeqStack( ); //析构函数

void Push(T x); //将元素x入栈

T Pop( ); //将栈顶元素弹出

T GetTop( ); //取栈顶元素（并不删除）

bool Empty( ); //判断栈是否为空

bool Full(); //判断栈是否为满

private:

T data[StackSize]; //存放栈元素的数组

int top; //栈顶指针，指示栈顶元素在数组中的下标

};

//初始化空栈

template <class T>

SeqStack<T>::SeqStack( )

{

top=-1;

}

//功 能：销毁栈

template <class T>

SeqStack<T>::~SeqStack( )

{

}

//功 能：元素x压栈

template <class T>

void SeqStack<T>::Push(T x)

{

if (Full()) throw "Overflow";

top++;

data[top]=x;

}

// 功 能：栈顶元素弹栈

template <class T>

T SeqStack<T>::Pop( )

{

T x;

if (Empty()) throw "Downflow";

x=data[top--];

return x;

}

// 功 能：读取当前的栈顶元素

template <class T>

T SeqStack<T>::GetTop( )

{

if (Empty())throw"Downflow";

return data[top];

}

//功 能：判断栈是否为空

template <class T>

bool SeqStack<T>::Empty( )

{

if(top==-1) return 1;

else return 0;

}

//功 能：判断栈是否为满

template <class T>

bool SeqStack<T>::Full( )

{

return top==StackSize-1;

}

//顺序二叉树类

const int TreeSize=100; //最大长度

template <class T>

class BiTree{

public:

BiTree(T \*str); //构造函数，初始化一棵二叉树，其前序序列由键盘输入

void PreOrder(int pos); //前序遍历二叉树

void InOrder(int pos); //中序遍历二叉树

void PostOrder(int pos); //后序遍历二叉树

int CountLeaf(int pos); //计算叶子结点总数

int Depth(int pos); //计算高度

void PreOrder(); //非递归先序遍历

void LevelOrder(); //层次遍历

private:

T data[TreeSize];

//存储结点的元素值，根结点存储在1下标，双亲下标为i，则左右孩子依次为2i，2i+1

};

template <class T>

void BiTree<T>::PreOrder(int pos) //前序遍历二叉树

{

if(pos<TreeSize&&data[pos]!='#')

{

cout<<data[pos]<<" ";

PreOrder(2\*pos);

PreOrder(2\*pos+1);

}

}

template <class T>

void BiTree<T>::InOrder(int pos) //中序遍历二叉树

{

if(pos<TreeSize&&data[pos]!='#')

{

InOrder(2\*pos);

cout<<data[pos]<<" ";

InOrder(2\*pos+1);

}

}

template <class T>

void BiTree<T>::PostOrder(int pos) //后序遍历二叉树

{

if(pos<TreeSize&&data[pos]!='#')

{

PostOrder(2\*pos);

PostOrder(2\*pos+1);

cout<<data[pos]<<" ";

}

}

template <class T>

int BiTree<T>::CountLeaf(int pos) //计算叶子结点总数

{

if(pos<=TreeSize&&data[pos]!='#')

{

count++;

if((2\*pos<=TreeSize&&data[2\*pos]!='#')||(2\*pos+1<=TreeSize&&data[2\*pos+1]!='#'))

count--;

CountLeaf(2\*pos);

CountLeaf(2\*pos+1);

}

return count;

}

template <class T>

int BiTree<T>::Depth(int pos) //计算高度

{

int l,r;

if(pos<=TreeSize&&data[pos]!='#')

{

l=Depth(2\*pos);

r=Depth(2\*pos+1);

return c=l+1>r+1?l+1:r+1;

}

return 0;

}

template <class T>

void BiTree<T>::PreOrder() //非递归先序遍历

{

SeqStack<T> s;

int x=1;

s.Push(x); //将元素root入栈

while(!s.Empty())

{

x=s.GetTop(); //取栈顶元素（并不删除）

cout<<data[x]<<" ";

s.Pop( ); //将栈顶元素弹出

if(x\*2+1<=TreeSize&&data[x\*2+1]!='#') s.Push(x\*2+1);

if(x\*2<=TreeSize&&data[x\*2]!='#') s.Push(x\*2);

}

}

template <class T>

void BiTree<T>::LevelOrder() //层次遍历

{

CirQueue<T> s;

int x=1;

s.EnQueue(x);

while(!s.Empty())

{

x=s.GetQueue();

if(x\*2<=TreeSize&&data[x\*2]!='#') s.EnQueue(x\*2);

if(x\*2+1<=TreeSize&&data[x\*2+1]!='#') s.EnQueue(x\*2+1);

cout<<data[x]<<" ";

s.DeQueue();

}

}

//constructor

template <class T>

BiTree<T>::BiTree(T \*str)

{

strcpy(data+1,str);

}

int main()

{

char str[80];

cin>>str;

BiTree<char> bt(str);

bt.PreOrder(1);

cout<<endl;

bt.InOrder(1);

cout<<endl;

bt.PostOrder(1);

cout<<endl;

bt.PreOrder();

cout<<endl;

bt.LevelOrder();

cout<<endl;

cout<<bt.CountLeaf(1)<<" "<<bt.Depth(1)<<endl;

return 0;

}

## 1020: 二叉链表（1）

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 80  Solved: 61  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1020)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1020)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1020)]

## Description

**已知二叉链表类，请实现二叉树的先序、中序、后序递归遍历。**

#include<iostream>  
#include<string>  
using namespace std;

//二叉树类，实现二叉树的递归算法：三种遍历  
template <class T>  
struct BiNode   //二叉树的结点结构  
{  
  T data;         
  BiNode<T> \*lchild, \*rchild;  
};

template <class T>  
class BiTree{  
public:                 
 BiTree(); //构造函数，初始化一棵二叉树，其前序序列由键盘输入  
    ~BiTree();         //析构函数，释放二叉链表中各结点的存储空间  
    void PreOrder(void)  
 {  
  PreOrder(root);  
 }  
 void InOrder(void)  
 {  
  InOrder(root);  
 }  
 void PostOrder(void)  
 {  
  PostOrder(root);  
 }  
private:  
    BiNode<T> \*root;         //指向根结点的头指针        
 void Creat(BiNode<T> \*&root);//被构造函数调用，递归方式生成一颗二叉树  
    void Release(BiNode<T> \*&root);     //被析构函数调用   
 void PreOrder(BiNode<T> \*root);     //前序遍历二叉树  
    void InOrder(BiNode<T> \*root);      //中序遍历二叉树  
    void PostOrder(BiNode<T> \*root);    //后序遍历二叉树  
};

//定义类中的成员函数

//构造函数:Creat利用创建二叉树  
template<class T>  
BiTree<T>::BiTree()   
{  
    Creat(root);  
}

//功    能：递归方法创建一棵二叉树,由构造函数调用  
template <class T>  
void BiTree<T>::Creat(BiNode<T> \* &root)  
{  
 T ch;   
 cin>>ch;  
    if (ch=="#") root = NULL;  //创建结点值为字符串的二叉树  
    else{   
      root = new BiNode<T>;   //生成一个结点  
         root->data=ch;  
         Creat(root->lchild);    //递归建立左子树  
         Creat(root->rchild);    //递归建立右子树  
    }   
}  
//功    能：析构函数，释放二叉链表中各结点的存储空间  
template<class T>   
BiTree<T>::~BiTree() //析构函数不能带参数  
{  
 Release(root);  
}

//功    能：释放二叉树的存储空间，析构函数调用  
template<class T>  
void BiTree<T>::Release(BiNode<T>\*&root)  
{  
  if (root != NULL){                    
   Release(root->lchild);   //释放左子树  
      Release(root->rchild);   //释放右子树  
      delete root;  
  }    
}

int main()  
{  
 BiTree<string> bt;  
 bt.PreOrder();  
 cout<<endl;  
 bt.InOrder();  
 cout<<endl;  
 bt.PostOrder();  
 cout<<endl;  
  return 0;  
}

**Input**

**Output**

**Sample Input**

Li Sun # Zhao Zhou # # Wang # # Qian # #

**Sample Output**

Li Sun Zhao Zhou Wang Qian

Sun Zhou Zhao Wang Li Qian

Zhou Wang Zhao Sun Qian Li

#include<iostream>

#include<string>

using namespace std;

//二叉树类，实现二叉树的递归算法：三种遍历

template <class T>

struct BiNode   //二叉树的结点结构

{

  T data;

  BiNode<T> \*lchild, \*rchild;

};

template <class T>

class BiTree{

public:

 BiTree(); //构造函数，初始化一棵二叉树，其前序序列由键盘输入

    ~BiTree();         //析构函数，释放二叉链表中各结点的存储空间

    void PreOrder(void)

                         {

                          PreOrder(root);

                         }

 void InOrder(void)

                             {

                              InOrder(root);

                             }

 void PostOrder(void)

                             {

                              PostOrder(root);

                             }

private:

    BiNode<T> \*root;         //指向根结点的头指针

 void Creat(BiNode<T> \*&root);//被构造函数调用，递归方式生成一颗二叉树

    void Release(BiNode<T> \*&root);     //被析构函数调用

 void PreOrder(BiNode<T> \*root);     //前序遍历二叉树

    void InOrder(BiNode<T> \*root);      //中序遍历二叉树

    void PostOrder(BiNode<T> \*root);    //后序遍历二叉树

};

//定义类中的成员函数

//构造函数:Creat利用创建二叉树

template<class T>

BiTree<T>::BiTree()

{

    Creat(root);

}

//功    能：递归方法创建一棵二叉树,由构造函数调用

template <class T>

void BiTree<T>::Creat(BiNode<T> \* &root)

{

 T ch;

 cin>>ch;

    if (ch=="#") root = NULL;  //创建结点值为字符串的二叉树

    else{

      root = new BiNode<T>;   //生成一个结点

         root->data=ch;

         Creat(root->lchild);    //递归建立左子树

         Creat(root->rchild);    //递归建立右子树

    }

}

//功    能：析构函数，释放二叉链表中各结点的存储空间

template<class T>

BiTree<T>::~BiTree() //析构函数不能带参数

{

 Release(root);

}

//功    能：释放二叉树的存储空间，析构函数调用

template<class T>

void BiTree<T>::Release(BiNode<T>\*&root)

{

  if (root != NULL){

   Release(root->lchild);   //释放左子树

      Release(root->rchild);   //释放右子树

      delete root;

  }

}

template<class T>

void BiTree<T>::PreOrder(BiNode<T> \*root)    //前序遍历二叉树

{

    if(root)

    {

        cout<<root->data<<" ";

        PreOrder(root->lchild);

        PreOrder(root->rchild);

    }

}

template<class T>

void BiTree<T>::InOrder(BiNode<T> \*root)    //中序遍历二叉树

{

    if(root)

    {

        InOrder(root->lchild);

        cout<<root->data<<" ";

        InOrder(root->rchild);

    }

}

template<class T>

void BiTree<T>::PostOrder(BiNode<T> \*root)    //后序遍历二叉树

{

    if(root)

    {

        PostOrder(root->lchild);

        PostOrder(root->rchild);

        cout<<root->data<<" ";

    }

}

int main()

{

 BiTree<string> bt;

 bt.PreOrder();

 cout<<endl;

 bt.InOrder();

 cout<<endl;

 bt.PostOrder();

 cout<<endl;

  return 0;

}

## 1021: 二叉链表（2）

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 44  Solved: 39  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1021)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1021)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1021)]

## Description

**递归算法计算二叉树的叶子个数和高度。请完成如下两个算法：**

int CountLeaf(BiNode<T> \*);       //递归算法计算叶子数  
int Depth(BiNode<T> \*);           //递归算法计算高度

其余代码已知如下：

#include<iostream>  
#include<string>  
using namespace std;

//二叉树类，实现计算高度、叶子数  
template <class T>  
struct BiNode   //二叉树的结点结构  
{  
  T data;         
  BiNode<T> \*lchild, \*rchild;  
};

template <class T>  
class BiTree{  
public:                 
 BiTree(); //构造函数，初始化一棵二叉树，其前序序列由键盘输入  
    ~BiTree();         //析构函数，释放二叉链表中各结点的存储空间  
 BiNode<T> \*GetRoot()  
 {  
  return root;  
 }  
    int CountLeaf()  
 {  
  return CountLeaf(root);  
 }  
 int Depth()  
 {  
  return Depth(root);  
 }  
 int CountLeaf(BiNode<T> \*);       //计算叶子数  
 int Depth(BiNode<T> \*);           //计算高度  
private:  
    BiNode<T> \*root;         //指向根结点的头指针        
 void Creat(BiNode<T> \*&root);//被构造函数调用，递归方式生成一颗二叉树  
    void Release(BiNode<T> \*&root);   //被析构函数调用    
};

//定义类中的成员函数

//构造函数:Creat利用创建二叉树  
template<class T>  
BiTree<T>::BiTree()   
{  
    Creat(root);  
}

//功    能：递归方法创建一棵二叉树,由构造函数调用  
template <class T>  
void BiTree<T>::Creat(BiNode<T> \* &root)  
{  
 T ch;   
 cin>>ch;  
    if (ch=="#") root = NULL;  //创建结点值为字符串的二叉树  
    else{   
      root = new BiNode<T>;   //生成一个结点  
         root->data=ch;  
         Creat(root->lchild);    //递归建立左子树  
         Creat(root->rchild);    //递归建立右子树  
    }   
}  
//功    能：析构函数，释放二叉链表中各结点的存储空间  
template<class T>   
BiTree<T>::~BiTree() //析构函数不能带参数  
{  
 Release(root);  
}

//功    能：释放二叉树的存储空间，析构函数调用  
template<class T>  
void BiTree<T>::Release(BiNode<T>\*&root)  
{  
  if (root != NULL){                    
   Release(root->lchild);   //释放左子树  
      Release(root->rchild);   //释放右子树  
      delete root;  
  }    
}

int main()  
{  
 BiTree<string> bt;  
 cout<<bt.CountLeaf()<<" "<<bt.Depth()<<endl;  
    cout<<bt.CountLeaf(bt.GetRoot())<<" "<<bt.Depth(bt.GetRoot())<<endl;  
 return 0;  
}

**Input**

**Output**

**Sample Input**

Li Sun # Zhao Zhou # # Wang # # Qian # #

**Sample Output**

3 4

3 4

#include<iostream>

#include<string>

int i;

using namespace std;

//二叉树类，实现计算高度、叶子数

template <class T>

struct BiNode //二叉树的结点结构

{

T data;

BiNode<T> \*lchild, \*rchild;

};

template <class T>

class BiTree{

public:

BiTree(); //构造函数，初始化一棵二叉树，其前序序列由键盘输入

~BiTree(); //析构函数，释放二叉链表中各结点的存储空间

BiNode<T> \*GetRoot()

{

return root;

}

int CountLeaf()

{

return CountLeaf(root);

}

int Depth()

{

return Depth(root);

}

int CountLeaf(BiNode<T> \*root); //计算叶子数

int Depth(BiNode<T> \*root); //计算高度

private:

BiNode<T> \*root; //指向根结点的头指针

void Creat(BiNode<T> \*&root);//被构造函数调用，递归方式生成一颗二叉树

void Release(BiNode<T> \*&root); //被析构函数调用

};

//定义类中的成员函数

//构造函数:Creat利用创建二叉树

template<class T>

BiTree<T>::BiTree()

{

Creat(root);

}

//功 能：递归方法创建一棵二叉树,由构造函数调用

template <class T>

void BiTree<T>::Creat(BiNode<T> \* &root)

{

T ch;

cin>>ch;

if (ch=="#") root = NULL; //创建结点值为字符串的二叉树

else{

root = new BiNode<T>; //生成一个结点

root->data=ch;

Creat(root->lchild); //递归建立左子树

Creat(root->rchild); //递归建立右子树

}

}

//功 能：析构函数，释放二叉链表中各结点的存储空间

template<class T>

BiTree<T>::~BiTree() //析构函数不能带参数

{

Release(root);

}

//功 能：释放二叉树的存储空间，析构函数调用

template<class T>

void BiTree<T>::Release(BiNode<T>\*&root)

{

if (root != NULL){

Release(root->lchild); //释放左子树

Release(root->rchild); //释放右子树

delete root;

}

}

template<class T>

int BiTree<T>::CountLeaf(BiNode<T> \*root) //递归算法计算叶子数

{

            if(root==this->root) i=0;

            if(root)

            {

                i++;

            if(root->lchild||root->rchild) i--;

            CountLeaf(root->lchild);

            CountLeaf(root->rchild);

            }

            return i;

}

template<class T>

int BiTree<T>::Depth(BiNode<T> \*root) //递归算法计算高度

{

            if(root==this->root) i=0;

            if(root)

            {

            i++;

            CountLeaf(root->lchild);

            i--;

            CountLeaf(root->rchild);

            }

            return i+1;

}

int main()

{

BiTree<string> bt;

cout<<bt.CountLeaf()<<" "<<bt.Depth()<<endl;

cout<<bt.CountLeaf(bt.GetRoot())<<" "<<bt.Depth(bt.GetRoot())<<endl;

return 0;

}

## 1022: 二叉链表（3）

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 49  Solved: 35  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1022)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1022)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1022)]

## Description

**分别利用栈和队列实现二叉树的先序非递归算法和层次遍历算法。**

**已知部分代码如下：（勿改动）**

#include<iostream>  
#include<string>  
using namespace std;

//循环队列类  
const int QueueSize=100;  //循环队列的最大长度  
template <class T>        //定义模板类CirQueue  
class CirQueue  
{  
public:  
    CirQueue( );                 //构造函数，置空队  
    ~ CirQueue( );               //析构函数  
    void EnQueue(T x);           //将元素x入队  
    T DeQueue( );                //将队头元素出队  
    T GetQueue( );               //取队头元素（并不删除）  
    bool Empty( );               //判断队列是否为空  
 bool Full();  
private:  
    T data[QueueSize];           //存放队列元素的数组  
    int front, rear;    //队头和队尾指针，分别指向队头元素的前一个位置和队尾元素的位置  
};

 //功    能：初始化空队列  
 template <class T>  
CirQueue<T>::CirQueue( )   
{  
 front=rear=0;  
}

// 功    能：销毁队列  
template <class T>  
CirQueue<T>::~CirQueue( )  
{  
}

//功    能：元素x入队  
template <class T>   
void CirQueue<T>::EnQueue(T x)  
{  
    if (Full()) throw "Overflow";  
    rear=(rear+1) % QueueSize;   //队尾指针在循环意义下加1  
    data[rear]=x;                //在队尾处插入元素  
}

//功    能：队头元素出队，返回值为出队元素  
template <class T>   
T CirQueue<T>::DeQueue( )  
{  
    if (Empty()) throw "Downflow";   
    front=(front+1) % QueueSize;    //队头指针在循环意义下加1  
    return data[front];             //读取并返回出队前的队头元素，注意队头指针  
}  
                                   //指向队头元素的前一个位置  
// 功    能：获取队头元素  
template <class T>  
T CirQueue<T>::GetQueue( )  
{     
    int i;  
    if (Empty()) throw "Downflow";   
    i=(front+1) % QueueSize;  //注意不要给队头指针赋值  
    return data[i];  
}

// 功    能：判断队列是否为空,若空返回true，否则返回false  
template <class T>   
bool CirQueue<T>::Empty( )   
{  
    return front==rear;   
}  
//功   能：判断队列是否满，若满返回true，否则false  
template <class T>  
bool CirQueue<T>::Full()  
{  
 return (rear+1)%QueueSize==front;  
}

//栈类  
const int StackSize=100;  //10只是示例性的数据，可以根据实际问题具体定义  
template <class T>       //定义模板类SeqStack  
class SeqStack  
{  
public:  
    SeqStack( ) ;            //构造函数，栈的初始化  
 ~SeqStack( );            //析构函数  
    void Push(T x);          //将元素x入栈  
    T Pop( );                //将栈顶元素弹出  
    T GetTop( );          //取栈顶元素（并不删除）  
 bool Empty( );           //判断栈是否为空  
 bool Full();             //判断栈是否为满  
private:  
    T data[StackSize];      //存放栈元素的数组  
    int top;                //栈顶指针，指示栈顶元素在数组中的下标  
};

//初始化空栈  
template <class T>  
SeqStack<T>::SeqStack( )  
{  
 top=-1;  
}

//功    能：销毁栈  
template <class T>  
SeqStack<T>::~SeqStack( )  
{  
}

//功    能：元素x压栈  
template <class T>   
void SeqStack<T>::Push(T x)  
{  
    if (Full()) throw "Overflow";  
    top++;  
    data[top]=x;  
}  
// 功    能：栈顶元素弹栈  
template <class T>  
T SeqStack<T>::Pop( )  
{   
    T x;  
    if (Empty()) throw "Downflow";  
    x=data[top--];  
    return x;  
}

// 功    能：读取当前的栈顶元素  
template <class T>   
T SeqStack<T>::GetTop( )  
{  
 if (Empty())throw"Downflow";    
    return data[top];  
}

//功    能：判断栈是否为空  
template <class T>   
bool SeqStack<T>::Empty( )  
{  
 return top==-1;  
}

//功    能：判断栈是否为满  
template <class T>   
bool SeqStack<T>::Full( )  
{  
 return top==StackSize-1;  
}

//二叉树类，实现二叉树的非递归算法：先序遍历、层次遍历  
template <class T>  
struct BiNode   //二叉树的结点结构  
{  
  T data;         
  BiNode<T> \*lchild, \*rchild;  
};  
//实现二叉树的非递归遍历  
template <class T>  
class BiTree{  
public:                 
 BiTree(); //构造函数，初始化一棵二叉树，其前序序列由键盘输入  
    ~BiTree();         //析构函数，释放二叉链表中各结点的存储空间  
    void PreOrder();     //前序遍历二叉树  
    void LevelOrder();   //层次遍历二叉树  
 //   void InOrder();      //中序遍历二叉树  
  //  void PostOrder();    //后序遍历二叉树  
private:  
    BiNode<T> \*root;         //指向根结点的头指针        
 void Creat(BiNode<T> \*&root);//被有参构造函数调用，以便生成一颗二叉树  
    void Release(BiNode<T> \*&root);   //被析构函数调用   
};

//定义类中的成员函数  
//构造函数:Creat利用创建二叉树  
template<class T>  
BiTree<T>::BiTree()   
{  
    Creat(root);  
}

//功    能：递归方法创建一棵二叉树,由构造函数调用  
template <class T>  
void BiTree<T>::Creat(BiNode<T> \* &root)  
{  
 T ch;   
 cin>>ch;  
    if (ch=="#") root = NULL;  //创建结点值为字符串的二叉树  
    else{   
      root = new BiNode<T>;   //生成一个结点  
         root->data=ch;  
         Creat(root->lchild);    //递归建立左子树  
         Creat(root->rchild);    //递归建立右子树  
    }   
}  
//功    能：析构函数，释放二叉链表中各结点的存储空间  
template<class T>   
BiTree<T>::~BiTree() //析构函数不能带参数  
{  
 Release(root);  
}

//功    能：释放二叉树的存储空间，析构函数调用  
template<class T>  
void BiTree<T>::Release(BiNode<T>\*&root)  
{  
  if (root != NULL){                    
   Release(root->lchild);   //释放左子树  
      Release(root->rchild);   //释放右子树  
      delete root;  
  }    
}

int main()  
{  
 BiTree<string> bt;  
 bt.PreOrder();  
 cout<<endl;  
 bt.LevelOrder();  
 cout<<endl;  
 return 0;  
}

**Input**

**Output**

**Sample Input**

Li Sun # Zhao Zhou # # Wang # # Qian # #

**Sample Output**

Li Sun Zhao Zhou Wang Qian

Li Sun Qian Zhao Zhou Wang

#include<iostream>

#include<string>

using namespace std;

//循环队列类

const int QueueSize=100; //循环队列的最大长度

template <class T> //定义模板类CirQueue

class CirQueue

{

public:

CirQueue( ); //构造函数，置空队

~ CirQueue( ); //析构函数

void EnQueue(T x); //将元素x入队

T DeQueue( ); //将队头元素出队

T GetQueue( ); //取队头元素（并不删除）

bool Empty( ); //判断队列是否为空

bool Full();

private:

T data[QueueSize]; //存放队列元素的数组

int front, rear; //队头和队尾指针，分别指向队头元素的前一个位置和队尾元素的位置

};

//功 能：初始化空队列

template <class T>

CirQueue<T>::CirQueue( )

{

front=rear=0;

}

// 功 能：销毁队列

template <class T>

CirQueue<T>::~CirQueue( )

{

}

//功 能：元素x入队

template <class T>

void CirQueue<T>::EnQueue(T x)

{

if (Full()) throw "Overflow";

rear=(rear+1) % QueueSize; //队尾指针在循环意义下加1

data[rear]=x; //在队尾处插入元素

}

//功 能：队头元素出队，返回值为出队元素

template <class T>

T CirQueue<T>::DeQueue( )

{

if (Empty()) throw "Downflow";

front=(front+1) % QueueSize; //队头指针在循环意义下加1

return data[front]; //读取并返回出队前的队头元素，注意队头指针

}

//指向队头元素的前一个位置

// 功 能：获取队头元素

template <class T>

T CirQueue<T>::GetQueue( )

{

int i;

if (Empty()) throw "Downflow";

i=(front+1) % QueueSize; //注意不要给队头指针赋值

return data[i];

}

// 功 能：判断队列是否为空,若空返回true，否则返回false

template <class T>

bool CirQueue<T>::Empty( )

{

return front==rear;

}

//功 能：判断队列是否满，若满返回true，否则false

template <class T>

bool CirQueue<T>::Full()

{

return (rear+1)%QueueSize==front;

}

//栈类

const int StackSize=100; //10只是示例性的数据，可以根据实际问题具体定义

template <class T> //定义模板类SeqStack

class SeqStack

{

public:

SeqStack( ) ; //构造函数，栈的初始化

~SeqStack( ); //析构函数

void Push(T x); //将元素x入栈

T Pop( ); //将栈顶元素弹出

T GetTop( ); //取栈顶元素（并不删除）

bool Empty( ); //判断栈是否为空

bool Full(); //判断栈是否为满

private:

T data[StackSize]; //存放栈元素的数组

int top; //栈顶指针，指示栈顶元素在数组中的下标

};

//初始化空栈

template <class T>

SeqStack<T>::SeqStack( )

{

top=-1;

}

//功 能：销毁栈

template <class T>

SeqStack<T>::~SeqStack( )

{

}

//功 能：元素x压栈

template <class T>

void SeqStack<T>::Push(T x)

{

if (Full()) throw "Overflow";

top++;

data[top]=x;

}

// 功 能：栈顶元素弹栈

template <class T>

T SeqStack<T>::Pop( )

{

T x;

if (Empty()) throw "Downflow";

x=data[top--];

return x;

}

// 功 能：读取当前的栈顶元素

template <class T>

T SeqStack<T>::GetTop( )

{

if (Empty())throw"Downflow";

return data[top];

}

//功 能：判断栈是否为空

template <class T>

bool SeqStack<T>::Empty( )

{

return top==-1;

}

//功 能：判断栈是否为满

template <class T>

bool SeqStack<T>::Full( )

{

return top==StackSize-1;

}

//二叉树类，实现二叉树的非递归算法：先序遍历、层次遍历

template <class T>

struct BiNode //二叉树的结点结构

{

T data;

BiNode<T> \*lchild, \*rchild;

};

//实现二叉树的非递归遍历

template <class T>

class BiTree{

public:

BiTree(); //构造函数，初始化一棵二叉树，其前序序列由键盘输入

~BiTree(); //析构函数，释放二叉链表中各结点的存储空间

void PreOrder(); //前序遍历二叉树

void LevelOrder(); //层次遍历二叉树

void InOrder(); //中序遍历二叉树

// void PostOrder(); //后序遍历二叉树

private:

BiNode<T> \*root; //指向根结点的头指针

void Creat(BiNode<T> \*&root);//被有参构造函数调用，以便生成一颗二叉树

void Release(BiNode<T> \*&root); //被析构函数调用

};

//定义类中的成员函数

//构造函数:Creat利用创建二叉树

template<class T>

BiTree<T>::BiTree()

{

Creat(root);

}

//功 能：递归方法创建一棵二叉树,由构造函数调用

template <class T>

void BiTree<T>::Creat(BiNode<T> \* &root)

{

T ch;

cin>>ch;

if (ch=="#") root = NULL; //创建结点值为字符串的二叉树

else{

root = new BiNode<T>; //生成一个结点

root->data=ch;

Creat(root->lchild); //递归建立左子树

Creat(root->rchild); //递归建立右子树

}

}

//功 能：析构函数，释放二叉链表中各结点的存储空间

template<class T>

BiTree<T>::~BiTree() //析构函数不能带参数

{

Release(root);

}

//功 能：释放二叉树的存储空间，析构函数调用

template<class T>

void BiTree<T>::Release(BiNode<T>\*&root)

{

if (root != NULL){

Release(root->lchild); //释放左子树

Release(root->rchild); //释放右子树

delete root;

}

}

template<class T>

void BiTree<T>::PreOrder() //前序遍历二叉树

{

SeqStack<BiNode<T> \*> s;

BiNode<T> \* x;

s.Push(root); //将元素root入栈

while(!s.Empty())

{

x=s.GetTop(); //取栈顶元素（并不删除）

cout<<s.GetTop()->data<<" ";

s.Pop( ); //将栈顶元素弹出

if(x->rchild) s.Push(x->rchild);

if(x->lchild) s.Push(x->lchild);

}

}

template<class T>

void BiTree<T>::InOrder() //非递归中序遍历

{

SeqStack<BiNode<T> \*> s;

BiNode<T> \* x;

s.Push(root); //将元素root入栈

x=s.GetTop();

while(x || !s.Empty())

{

while(x)

{

s.Push(x);

x=x->lchild;

}

x=s.Pop();

cout<<x->data<<" ";

x=x->rchild;

}

}

template<class T>

void BiTree<T>::LevelOrder() //层次遍历

{

CirQueue<BiNode<T> \*> s;

BiNode<T> \* x;

s.EnQueue(root);

while(!s.Empty())

{

x=s.GetQueue();

if(x->lchild) s.EnQueue(x->lchild);

if(x->rchild) s.EnQueue(x->rchild);

cout<<x->data<<" ";

s.DeQueue( );

}

}

int main()

{

BiTree<string> bt;

bt.PreOrder();

cout<<endl;

bt.LevelOrder();

cout<<endl;

return 0;

}

## 1023: haffman编码

Time Limit: 1 Sec  Memory Limit: 128 MB  
Submit: 31  Solved: 23  
[[Submit](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/submitpage.php?id=1023)][[Status](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/problemstatus.php?id=1023)][[Web Board](http://221.203.21.203:8001/rwt/USTL/http/GV6T6N3UFZ5UGLSSGM3B/OnlineJudge/bbs.php?pid=1023)]

## Description

**根据权值序列构造huffman树，并输出各个权值对应的huffman编码，并计算wpl值。**

**已知部分代码如下：**

#include <iostream>  
#include <climits>   //INT\_MAX  
#include <string>  
using namespace std;

template <class T>

//链式栈  
struct Node  
{  
    T data;  
    Node<T> \*next;    
};

template <class T>  
class LinkStack  
{  
public:  
    LinkStack( );              //构造函数，置空链栈  
    ~LinkStack( );             //析构函数，释放链栈中各结点的存储空间  
    void Push(T x);           //将元素x入栈  
    T Pop( );                  //将栈顶元素出栈  
    T GetTop( );               //取栈顶元素（并不删除）  
    bool Empty( );             //判断链栈是否为空栈  
    void VisitStack();         //遍历栈（从顶到底）  
private:  
    Node<T> \*top;             //栈顶指针即链栈的头指针  
};

/\*  
 \* 前置条件：栈不存在  
 \* 输    入：无  
 \* 功    能：栈的初始化  
 \* 输    出：无  
 \* 后置条件：构造一个空栈  
 \*/

template <class T>  
LinkStack<T>::LinkStack( )  
{  
 top=NULL;   
}  
/\*  
 \* 前置条件：栈已存在  
 \* 输    入：无  
 \* 功    能：销毁栈  
 \* 输    出：无  
 \* 后置条件：释放栈所占用的存储空间  
 \*/  
template <class T>  
LinkStack<T>::~LinkStack( )  
{  
 while (top)  
 {  
  Node<T> \*p;  
  p=top->next;  
        delete top;  
        top=p;  
 }  
}

/\*  
 \* 前置条件：栈已存在  
 \* 输    入：节点s  
 \* 功    能：在栈顶插入该节点  
 \* 输    出：无  
 \* 后置条件：如果插入成功，栈顶增加了一个元素  
 \*/

template<class T>   
void LinkStack<T>::Push(T x)  
{  
    Node<T> \*s;  
 s=new Node<T>;    
 if(!s) throw"Overflow";  
    s->data = x;     //申请一个数据域为x的结点s  
    s->next = top;   
 top=s;           //将结点s插在栈顶  
}

/\*  
 \* 前置条件：栈已存在  
 \* 输    入：无  
 \* 功    能：删除栈顶元素  
 \* 输    出：如果删除成功，返回被删元素值，否则，抛出异常  
 \* 后置条件：如果删除成功，栈顶减少了一个元素  
 \*/

template <class T> T LinkStack<T>::Pop( )  
{     
    Node<T> \*p;  
 int x;   
    if (top==NULL) throw "Downflow";  
    x=top->data;            //暂存栈顶元素  
    p=top;   
 top=top->next;         //将栈顶结点摘链  
    delete p;  
    return x;  
}

/\*  
 \* 前置条件：栈已存在  
 \* 输    入：无  
 \* 功    能：读取当前的栈顶元素  
 \* 输    出：若栈不空，返回当前的栈顶元素值  
 \* 后置条件：栈不变  
 \*/

template <class T>   
T LinkStack<T>::GetTop( )  
{  
    if (top)   
  return top->data;  
 else throw"Downflow";  
}

/\*  
 \* 前置条件：栈已存在  
 \* 输    入：无  
 \* 功    能：判断栈是否为空  
 \* 输    出：如果栈为空，返回1，否则，返回0  
 \* 后置条件：栈不变  
 \*/

template <class T>   
bool LinkStack<T>::Empty( )  
{  
    return top==NULL;   
}  
template <class T>  
void  LinkStack<T>::VisitStack()  
{  
       Node<T> \*p;;  
       for(p=top;p;p=p->next)  
           cout<<p->data;  
}

//huffman树  
const int MaxSize=100;

//huffman树结点信息（注意增加了各个权值所对应的汉字信息）  
struct HuffNode{  
 int weight;   //权值  
 string word;  //权值对应的信息  
 int parent;   //双亲所在数组下标  
 int lchild,rchild;  //左右孩子所在数组下标  
};

//huffman树类  
class HuffTree  
{  
private:  
 HuffNode hufftree[MaxSize+1];   //huffman树的存储（1~2\*n-1下标）  
 int n;  //权值个数  
public:  
    HuffTree(int w[],string str[],int n);   //initialize  
 void SelectMinW(int n,int &min); //选择最小权值的根结点所在下标  
 void VisitHuffTree();    //输出huffmantree  
    int HuffCode(LinkStack<char> S[]);  //设计haffman编码并计算wpl值，栈S[1]~S[n]存储各个权值的编码  
    void DisplayAllCoding(LinkStack<char> S[]); //输出各个权值的huffman编码  
};  
HuffTree::HuffTree(int w[],string str[],int n){  
 int i,min1,min2;  
 hufftree[0].weight=INT\_MAX;  
// hufftree[0].parent=-1;  
 for(i=1;i<=n;i++){  
  hufftree[i].weight=w[i];  
  hufftree[i].word=str[i];  
//  cout<<hufftree[i].word<<" ";  
  hufftree[i].parent=-1;  
  hufftree[i].lchild=-1;  
  hufftree[i].rchild=-1;  
 }  
 for( ;i<=2\*n-1;i++){  
  SelectMinW(i,min1);  
  SelectMinW(i,min2);  
//  cout<<"select: "<<min1<<"\t"<<min2<<endl;  
  hufftree[i].lchild=min1;  
  hufftree[i].rchild=min2;  
  hufftree[i].weight=hufftree[min1].weight+hufftree[min2].weight;  
  hufftree[i].parent=-1;  
  hufftree[min1].parent=i;  
  hufftree[min2].parent=i;  
 }  
 this->n=n;  
}

void HuffTree::SelectMinW(int n,int &min){  
 min=0;  
 for(int i=1;i<=n;i++)  
  if(hufftree[i].parent==-1&&  
      hufftree[i].weight<hufftree[min].weight) min=i;  
 hufftree[min].parent=0;//避免下一次重复被选  
}

void HuffTree::VisitHuffTree(){  
 cout<<"No\tweight\tparent\tlchild\trchild\n";  
 for(int i=1;i<=2\*n-1;i++){  
  cout<<i<<"\t"<<hufftree[i].weight<<"\t";  
  cout<<hufftree[i].parent<<"\t";  
  cout<<hufftree[i].lchild<<"\t"<<hufftree[i].rchild<<"\n";  
 }  
}  
   
int main(){  
 int w[MaxSize/2],n;//w存储权值，n存储权值个数

string inf[MaxSize];//存储各个权值对应的信息(一个汉字)  
    cin>>n;   //输入权值个数  
 int i;  
 for(i=1;i<=n;i++)  
  cin>>w[i];   //输入权值  
 for(i=1;i<=n;i++)  
  cin>>inf[i];  //输入各个权值对应的汉字  
   
 HuffTree ht(w,inf,n);   //create huffman tree  
// ht.VisitHuffTree();     //display huffman tree

 LinkStack<char> S[100];   //storage huffman coding  
 int wpl=ht.HuffCode(S);   //calculate the huffman coding for every weight and wpl  
 ht.DisplayAllCoding(S);   //display huffman coding  
 cout<<"WPL="<<wpl<<endl;  //display wpl  
 return 0;  
}

## Input

输入格式：

权值个数

各个权值

各个权值对应的汉字信息

例如：

5  
12 35 17 7 22  
李 赵 钱 孙 周

**Output**

**Sample Input**

5

12 35 17 7 22

李 赵 钱 孙 周

**Sample Output**

Coding of 12(李):011

Coding of 35(赵):11

Coding of 17(钱):00

Coding of 7(孙):010

Coding of 22(周):10

WPL=205

#include <iostream>

#include <climits> //INT\_MAX

#include <string>

using namespace std;

template <class T>

//链式栈

struct Node

{

T data;

Node<T> \*next;

};

template <class T>

class LinkStack

{

public:

LinkStack( ); //构造函数，置空链栈

~LinkStack( ); //析构函数，释放链栈中各结点的存储空间

void Push(T x); //将元素x入栈

T Pop( ); //将栈顶元素出栈

T GetTop( ); //取栈顶元素（并不删除）

bool Empty( ); //判断链栈是否为空栈

void VisitStack(); //遍历栈（从顶到底）

private:

Node<T> \*top; //栈顶指针即链栈的头指针

};

/\*

\* 前置条件：栈不存在

\* 输 入：无

\* 功 能：栈的初始化

\* 输 出：无

\* 后置条件：构造一个空栈

\*/

template <class T>

LinkStack<T>::LinkStack( )

{

top=NULL;

}

/\*

\* 前置条件：栈已存在

\* 输 入：无

\* 功 能：销毁栈

\* 输 出：无

\* 后置条件：释放栈所占用的存储空间

\*/

template <class T>

LinkStack<T>::~LinkStack( )

{

while (top)

{

Node<T> \*p;

p=top->next;

delete top;

top=p;

}

}

/\*

\* 前置条件：栈已存在

\* 输 入：节点s

\* 功 能：在栈顶插入该节点

\* 输 出：无

\* 后置条件：如果插入成功，栈顶增加了一个元素

\*/

template<class T>

void LinkStack<T>::Push(T x)

{

Node<T> \*s;

s=new Node<T>;

if(!s) throw"Overflow";

s->data = x; //申请一个数据域为x的结点s

s->next = top;

top=s; //将结点s插在栈顶

}

/\*

\* 前置条件：栈已存在

\* 输 入：无

\* 功 能：删除栈顶元素

\* 输 出：如果删除成功，返回被删元素值，否则，抛出异常

\* 后置条件：如果删除成功，栈顶减少了一个元素

\*/

template <class T> T LinkStack<T>::Pop( )

{

Node<T> \*p;

int x;

if (top==NULL) throw "Downflow";

x=top->data; //暂存栈顶元素

p=top;

top=top->next; //将栈顶结点摘链

delete p;

return x;

}

/\*

\* 前置条件：栈已存在

\* 输 入：无

\* 功 能：读取当前的栈顶元素

\* 输 出：若栈不空，返回当前的栈顶元素值

\* 后置条件：栈不变

\*/

template <class T>

T LinkStack<T>::GetTop( )

{

if (top)

return top->data;

else throw"Downflow";

}

/\*

\* 前置条件：栈已存在

\* 输 入：无

\* 功 能：判断栈是否为空

\* 输 出：如果栈为空，返回1，否则，返回0

\* 后置条件：栈不变

\*/

template <class T>

bool LinkStack<T>::Empty( )

{

return top==NULL;

}

template <class T>

void LinkStack<T>::VisitStack()

{

Node<T> \*p;;

for(p=top;p;p=p->next)

cout<<p->data;

}

//huffman树

const int MaxSize=100;

//huffman树结点信息（注意增加了各个权值所对应的汉字信息）

struct HuffNode{

int weight; //权值

string word; //权值对应的信息

int parent; //双亲所在数组下标

int lchild,rchild; //左右孩子所在数组下标

};

//huffman树类

class HuffTree

{

private:

HuffNode hufftree[MaxSize+1]; //huffman树的存储（1~2\*n-1下标）

int n; //权值个数

public:

HuffTree(int w[],string str[],int n); //initialize

void SelectMinW(int n,int &min); //选择最小权值的根结点所在下标

void VisitHuffTree(); //输出huffmantree

int HuffCode(LinkStack<char> S[]); //设计haffman编码并计算wpl值，栈S[1]~S[n]存储各个权值的编码

void DisplayAllCoding(LinkStack<char> S[]); //输出各个权值的huffman编码

};

HuffTree::HuffTree(int w[],string str[],int n){

int i,min1,min2;

hufftree[0].weight=INT\_MAX;

// hufftree[0].parent=-1;

for(i=1;i<=n;i++){

hufftree[i].weight=w[i];

hufftree[i].word=str[i];

// cout<<hufftree[i].word<<" ";

hufftree[i].parent=-1;

hufftree[i].lchild=-1;

hufftree[i].rchild=-1;

}

for( ;i<=2\*n-1;i++){

SelectMinW(i,min1);

SelectMinW(i,min2);

// cout<<"select: "<<min1<<"\t"<<min2<<endl;

hufftree[i].lchild=min1;

hufftree[i].rchild=min2;

hufftree[i].weight=hufftree[min1].weight+hufftree[min2].weight;

hufftree[i].parent=-1;

hufftree[min1].parent=i;

hufftree[min2].parent=i;

}

this->n=n;

}

void HuffTree::SelectMinW(int n,int &min){

min=0;

for(int i=1;i<=n;i++)

if(hufftree[i].parent==-1&&

hufftree[i].weight<hufftree[min].weight) min=i;

hufftree[min].parent=0;//避免下一次重复被选

}

void HuffTree::VisitHuffTree(){

cout<<"No\tweight\tparent\tlchild\trchild\n";

for(int i=1;i<=2\*n-1;i++){

cout<<i<<"\t"<<hufftree[i].weight<<"\t";

cout<<hufftree[i].parent<<"\t";

cout<<hufftree[i].lchild<<"\t"<<hufftree[i].rchild<<"\n";

}

}

int HuffTree::HuffCode(LinkStack<char> S[]) //设计haffman编码并计算wpl值，栈S[1]~S[n]存储各个权值的编码

{

int i,wpl=0,z;

for( i = 1; i < 2\*n-1 ; i++ )

{

z=i;

while( hufftree[z].parent != -1)

{

if(hufftree[hufftree[z].parent].lchild == z)

S[i].Push(48);

else

S[i].Push(49);

z=hufftree[z].parent;

if(i<=n)

wpl+=hufftree[i].weight;

}

}

return wpl;

}

void HuffTree::DisplayAllCoding(LinkStack<char> S[]) //输出各个权值的huffman编码

{

int i;

for( i = 1; i<2\*n-1 ; i++ )

{

if(hufftree[i].lchild == -1 && hufftree[i].rchild == -1 )

{

cout<<"Coding of "<<hufftree[i].weight<<"("

<<hufftree[i].word<<")"<<":";

S[i].VisitStack();

cout<<endl;

}

}

}

int main(){

int w[MaxSize/2],n;//w存储权值，n存储权值个数

string inf[MaxSize];//存储各个权值对应的信息(一个汉字)

cin>>n; //输入权值个数

int i;

for(i=1;i<=n;i++)

cin>>w[i]; //输入权值

for(i=1;i<=n;i++)

cin>>inf[i]; //输入各个权值对应的汉字

HuffTree ht(w,inf,n); //create huffman tree

// ht.VisitHuffTree(); //display huffman tree

LinkStack<char> S[100]; //storage huffman coding

int wpl=ht.HuffCode(S); //calculate the huffman coding for every weight and wpl

ht.DisplayAllCoding(S); //display huffman coding

cout<<"WPL="<<wpl<<endl; //display wpl

return 0;

}