

课程实验报告

2023 － 2024 学年第 二 学期

|  |  |
| --- | --- |
| 专业名称 | 软件工程 |
| 课程名称 | 数据结构实验 |
| 授课班级 | 23软件 |
| 学 号 | 23090200150 |
| 姓 名 | 丁士程 |

信息工程学院（大数据学院）

2024 年 6 月 28 日

**线性表的基本操作及应用**

**实验目的**

1．掌握线性表的链式存贮结构及基本操作，深入了解链表的基本特性，以便在实际问题背景下灵活运用它们。

2．巩固该存贮结构的构造方法，深入理解和灵活掌握链表的插入、删除等操作。

1. **实验环境**

⒈ 硬件：每个学生需配备计算机一台。操作系统：Windows；

⒉ 软件：Windows操作系统+C\C++等；

1. **实验要求**

1．定义各链表类型，并定义带有头结点的单链表。

2．将教材中链表的建立、初始化、插入等函数实现。

3．由主函数按照用户要求对各个链表操作访问。

5．特别注意：一元多项式的相加可参见课本实例（此做为样例），并且自己需设计3-5组测试数据，要包含一些如一个多项式为空等边界条件及复杂度极限输入数据！

6.每次操作之前要有明确的说明，操作后要输出操作结果。

1. **实验内容**

一元多项式相加

1. **源代码及调试**

以一元多项式相加为例

此代码中包含单链表的一些基本操作：

1. 建立链表

2. 初始化链表

3. 首插法、尾插法、按位插入

4. 链表输出

5. 链表元素遍历

6. 查找元素：按位查找、按元素查找

7. 获得链表长度

8. 删除节点：按元素内容删除、按位删除

其他操作：

1. 元素交换：节点地址不变，交换元素内容。并以此推出单链表的冒泡排序法（此代码中并未使用）

2. 两链表的集合操作：并集、交集（此代码中并未使用）

3. 一元多项式相加

**五、代码如下**

#include<bits/stdc++.h>

using namespace std;

typedef int elemtype;

**定义元素类型**

**此处定义为int类型**

// typedef string elemtype;

// typedef char elemtype;

typedef struct Lnode

{

**定义单链表每个节点中所包含的元素：系数data，次数power**

    elemtype data;

    int power;

    struct Lnode \*next;

}Lnode,\*linklist;

void init\_linklist(linklist &l)//初始化链表

{

    l=(Lnode \*)malloc(sizeof(linklist));

    l->next=NULL;

}

Lnode \*getelem\_locate(linklist l,int i)//按位查找

{

    int j=1;

    Lnode \*p=l->next;

    if(i==0)

    {

        return l;

    }

    while(p!=NULL&&j<i)

**指针循环到所查找的位置**

    {

        p=p->next;

        j++;

    }

    return p;

}

void insert\_linklist(linklist &l,int k,elemtype a)//在第k个后面插入元素

{

    Lnode \*p=getelem\_locate(l,k);

    Lnode \*s=(Lnode \*)malloc(sizeof(Lnode));

    s->data=a;

    s->next=p->next;

    p->next=s;

}

int Length(linklist L) //获得链表长度

{

    Lnode \*p = L->next;

    int len = 0;

    while(p){

        len++;

        p = p->next;

    }

    return len;

}

void printlinklist(linklist l)//链表元素挨个输出

{

    Lnode \*p=l;

    while(p->next!=NULL)

    {

        p=p->next;

        cout<<p->data<<"->";

    }

    cout<<endl<<endl;

}

void head\_insert(linklist &l,elemtype n)//首插法

{

    Lnode \*s=(Lnode \*)malloc(sizeof(Lnode));

    s->data=n;

    s->next=l->next;

    l->next=s;

}

Lnode \*getelem\_data(linklist l,elemtype n)//按值查找

{

    Lnode \*p=l->next;

    while(p!=NULL)

**指针循环到所查找的元素的位置**

    {

        if(p->data==n)

        {

            return p;

            cout<<p->data;

            return 0;

        }

        p=p->next;

    }

    return p;

**输出并结束循环**

}

void tail\_insert(linklist &l,elemtype a)//尾插法

{

    insert\_linklist(l,Length(l),a);

}

void delete\_locate\_lnode(linklist &l,int k)//按位删除元素

{

    Lnode \*p=getelem\_locate(l,k-1);

    p->next=p->next->next;

}

void delete\_data\_lnode(linklist &l,elemtype n)//按数据删除元素

{

    Lnode \*p=getelem\_data(l,n);

    p->next=p->next->next;

}

void swap\_linklist(linklist &l,int k,int i)//第k和第i个元素交换数值

{

    Lnode \*p=getelem\_locate(l,k);

    Lnode \*q=getelem\_locate(l,i);

    elemtype temp;

    temp=q->data;

    q->data=p->data;

    p->data=temp;

}

bool sort\_linklist(linklist &l) //类似冒泡排序的链表元素排序方法

{

    for(int i=1;i<=Length(l);i++)

    {

        for(int j=1;j<=Length(l);j++)

        {

            Lnode \*p=getelem\_locate(l,j);

            Lnode \*q=getelem\_locate(l,j+1);

            if(q->data<q->data)

            {

                swap\_linklist(l,j,j+1);

            }

        }

    }

}

void merge(linklist la,linklist lb,linklist &lc)//两个链表取交集

{

    Lnode \*pa,\*pb,\*pc;

    pa=la->next;

    pb=lb->next;

    pc=lc;

    while(pa!=NULL&&pb!=NULL)

    {

        if(pa->data<=pb->data)

        {

            pc->next=pa;

            pc=pc->next;

            pa=pa->next;

        }

        else

        {

            pc->next=pb;

            pc=pc->next;

            pb=pb->next;

        }

    }

    if(pa!=NULL)

    {

        pc->next=pa;

    }

    else

    {

        pc->next=pb;

    }

}

void union\_linklist(linklist la,linklist lb,linklist &lc)//两个链表取并集

{

    Lnode \*pa,\*pb,\*pc;

    pa=la->next;

    pb=lb->next;

    pc=lc;

    while(pa!=NULL&&pb!=NULL)

    {

        if(pa->data<pb->data)

        {

            pc->next=pa;

            pc=pc->next;

            pa=pa->next;

        }

        else if(pa->data==pb->data)

        {

            pc->next=pa;

            pc=pc->next;

            pa=pa->next;

            pb=pb->next;

        }

        else

        {

            pc->next=pb;

            pc=pc->next;

            pb=pb->next;

        }

    }

    if(pa!=NULL)

    {

        pc->next=pa;

    }

    else

    {

        pc->next=pb;

    }

}

void add(linklist LA,linklist LB,linklist LC) //两个一元多项式相加的操作函数

{

    linklist p=LA->next,q=LB->next,s,sr=LC;

    while(p&&q)

    {

        s=(linklist)malloc(sizeof(Lnode));

        if(p->power<q->power)

        {

            s->data=p->data;

            s->power=p->power;

            s->next=NULL;

            sr->next=s;

            sr=s;

            p=p->next;

        }

        else if(p->power==q->power)

        {

            s->data=p->data+q->data;

            s->power=p->power;

            s->next=NULL;

            sr->next=s;

            sr=s;

            p=p->next;

            q=q->next;

        }

        else

        {

            s->data=q->data;

            s->power=q->power;

            s->next=NULL;

            sr->next=s;

            sr=s;

            q=q->next;

        }

    }

    if(p){

        s->next=p;

    }

    else{

        s->next=q;

    }

}

void traverse(linklist LC) //遍历操作的链表，并输出

{

    linklist p=LC->next;

    if(p)

    {

        if(p->next==NULL)

        {

            cout<<0;

            return;

        }

        cout<<p->data<<"x^"<<p->power;

    }

    p=p->next;

    while(p)

    {

        if(p->data>0)

            cout<<'+'<<p->data<<"x^"<<p->power;

        else cout<<p->data<<"x^"<<p->power;

        p=p->next;

    }

}

bool Create\_List\_Tail(linklist &L,int n) //尾插法创建链表

{

    int i;

    linklist p,r;

    r=L;

    for(i=1;i<=n;i++)

    {

        p=new Lnode;

        cin>>p->data>>p->power;

        p->next=NULL;

        r->next=p;

        r=p;

    }

    return true;

}

int main()

{

    int n,m;

    linklist(LA);

    linklist(LB);

    linklist(LC);

    init\_linklist(LA);

    init\_linklist(LB);

    init\_linklist(LC);  //初始化LA,LB,LC；

    cin>>n;

    Create\_List\_Tail(LA,n);  //建表LA

    cin>>m;

    Create\_List\_Tail(LB,m);  //建表LB

    add(LA,LB,LC); //求解

    traverse(LC);  //遍历

}

/\*

5

**测试数据1**

2 1 3 2 4 3 5 4 6 5

3

1 2 2 3 4 5

\*/

/\*

**期望输出答案1**

2x^1+4x^2+6x^3+5x^4+10x^5

\*/

/\*

1

1 1

**测试数据2**

1

1 1

\*/

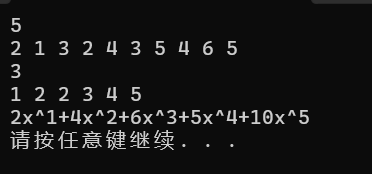
/\*

**期望输出答案2**

0

\*/

**六、运行结果及分析**

1、运行该代码结果1如下：

分析：

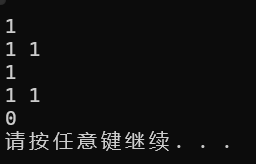
第二行输入多项式A：得到A=2x^1+3x^2+4x^3+5x^4+6x^5

第四行输入多项式B：得到B=1x^2+2x^3+4x^5

两式相加得到多项式C即最终答案：

C= 2x^1+4x^2+6x^3+5x^4+10x^5

2、运行该代码结果2如下：



分析：

第二行输入多项式A：得到A=0

第四行输入多项式B：得到B=0

两式相加得到多项式C即最终答案：

C= 0

**七、实验总结与体会**

通过此次实验掌握了单链表的基本操作（建立链表、初始化链表、首插法、尾插法、按位插入、链表输出、链表元素遍历、查找元素：（按位查找、按元素查找）、删除节点：（按元素内容删除、按位删除）、获得链表长度）。

通过自己思考，以数组的类比，实现了交换元素的操作，并在此基础之上完成了对单链表的冒泡排序。

通过此次实验，增强了我对于类比思想的理解，巩固了该存贮结构的构造方法，深入理解和灵活掌握链表的插入、删除等操作，体会到C++语言的魅力。

**堆栈及队列的应用**

**一、实验目的**

1．掌握栈和队列的抽象数据类型。

2．掌握实现栈与环队的各种操作的算法。

3．理解栈与递归的关系、环队应用。

**二、实验环境**

1.硬件：每个学生需配备计算机一台。

2.软件：windows 操作系统+ C、C++等。

**三、实验内容要求**

（一）栈表达式求值：

1．描述栈的每种操作在顺序栈或链栈上的实现。

2.将建栈、初始化栈、判断栈是否非空、求栈的长度、输出从栈顶到栈底的元素分别定义为5个子函数，通过主函数实现对上述子函数的调用。

3. 输入数据：数据域（data）设定为整型。

4.实现栈的表达式求值式的计算。

  （二）环队应用：

 1．用Ｃ或C++描述循环队列每种操作在顺序循环队列或链队上的实现；

2.将建队、初始化队列、判断循环队列空满、求队列的长度、实现入队出队操作分别定义为子函数，通过主函数实现对上述子函数的调用。

3．用Ｃ或C++描述每种操作在循环队列或者链队列上的实现。

4．将建队列、初始化队列、判断队列是否非空、求队列的长度、输出队列的元素分别定义为5 个子函数，通过主函数实现对上述子函数的调用；

   5．自选择：实现日常生活中环队的一个应用实例。比如圆桌会议安排及其他应用等。

**四、实验内容**

1．在自己的U盘的“姓名+学号”文件夹中创建“实验3”文件夹，本次实验的所有程序和数据都要求存储到本文件夹中。

2．定义两个堆栈：数据栈和操作栈。

3．完成一个表达式求值算法（输入一个复杂表达式，利用数据栈和操作栈进行表达式求值）。

4．具体实验原理参考教材中的表达式求值部分:要求实现多个实例输入输出测试数据。

5．将建队列、初始化队列、判断队列是否非空、求队列的长度、输出队列的元素分别定义为5 个子函数，通过主函数实现对上述子函数的调用；

6．实现日常生活中环队的一个应用实例；题目：约瑟夫环要求有多组输入输出具有不同规模或者特殊数据测试。

**五、代码如下：**

1、堆栈表达式求值

#include<bits/stdc++.h>

using namespace std;

typedef int DataType;

typedef struct

{

    DataType data[20];

    int top;

}SqStack;

int InitStack(SqStack &S) // 初始化堆栈

{

    S.top=-1;

    return 1;

}

int StackEmpty(SqStack S) // 判断堆栈是否为空

{

    return(S.top==-1?1:0);

}

int StackFull(SqStack S) // 判断堆栈是否满了

{

    return(S.top==20-1?1:0);

}

int Push(SqStack &S,DataType e) // 将元素e插入到栈中，作为新栈顶

{

    if(StackFull(S))

    return 0; // 栈满

    S.top++; // 栈顶位置上移

    S.data[S.top]=e;

    return 1;

}

int Pop(SqStack &S,DataType &e) // 若栈不为空，则删除栈顶元素

{

    if(StackEmpty(S))

    return 0; // 栈空

    e=S.data[S.top];

    S.top--;

    return 1;

}

DataType GetTop(SqStack S) // 若栈不为空，则取栈顶元素

{

    DataType e;

    if(StackEmpty(S))

    return 0; // 栈空

    e=S.data[S.top]; // 取出数据，top不变

    return e;

}

char Precede(char a,char b) // 比较两个算符的优先级

{

    char z;

    if((b=='+')||(b=='-')||(b=='\*')||(b=='/')||(b=='(')||(b==')')||(b=='='))

    switch(a)

    {

        case '+':

        case '-':

            if((b=='\*')||(b=='/')||(b=='('))

                z='<';

            else

                z='>';break;

        case '\*':

        case '/':

            if(b=='(')

                z='<';

            else

                z='>';break;

        case '(':

            if (b=='=')

                z='E';

            else if(b==')')

                z='=';

            else z='<';

            break;

        case ')':

            if(b=='(')

                z='E';

            else

                z='>';

            break;

        case '=':

            if(b=='=')

                z='=';

            else if(b==')')

                z='E';

            else z='<';break;

    }

    else z='E';

    return(z);

}

int In(char ch) // 判断字符ch是否为算符

{

    int i,flag=0;

    char op[7]={'+','-','\*','/','(',')','='};

    for(i=0;i<7;i++)

    {

        if(ch==op[i])

        {

            flag=1;break;

        }

    }

    return flag;

}

DataType Operate(DataType a,char theta,DataType b)

{

    DataType z;

    switch(theta)

    {

        case '+':z=a+b;break;

        case '-':z=a-b;break;

        case '\*':z=a\*b;break;

        case '/':z=a/b;break;

    }

    return(z);

}

int CaculateExpression()

{

    SqStack optr,opnd;

    DataType x,theta,a,b,s;

    char c;

    c=getchar();

    InitStack(optr);

    Push(optr,(DataType)'=');

    InitStack(opnd);

    while(c!='=' || (char)GetTop(optr)!='=')

    {

        if(!In(c))

        {

            s=c-'0';

            Push(opnd,(DataType)s);

            c=getchar();

        }

        else

        switch(Precede((char)GetTop(optr),c))

        {

            case '<': // 栈顶算符优先级低

                Push(optr,(DataType)c);

                c=getchar();

                break;

            case '=': // 优先级相同，脱去括号

                Pop(optr,x);

                c=getchar();

                break;

            case '>': // 栈顶算符优先级高，退栈并将运算

                Pop(optr,theta);

                Pop(opnd,b);

                Pop(opnd,a);

                Push(opnd,Operate(a,(char)theta,b));

                break;

            case 'E':printf("表达式中括号不匹配！");exit(1);

        }

    }

    return GetTop(opnd);

}

int main()

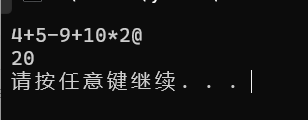
{

    printf("%d\n",CaculateExpression());

    return 0;

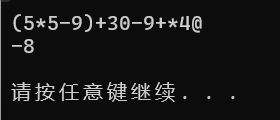
}

测试用例1及结果



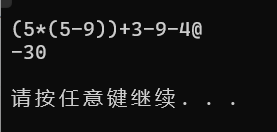
正确

测试用例2及结果



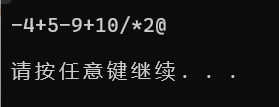
错误，该用例应无正确结果

测试用例3及结果



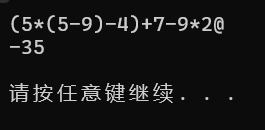
正确

测试用例4及结果



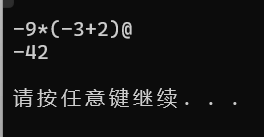
正确（少输出“NO”）

测试用例5及结果



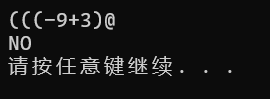
正确

测试用例6及结果



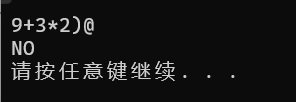
正确

测试用例7及结果



正确

测试用例8及结果



正确

2、环形队列

#include<bits/stdc++.h>

using namespace std;

typedef struct {

    //第一种实现:

    int\* \_data;

    //第一个元素的位置: 队头位置

    int \_front;

    //最后一个元素的下一个位置

    int \_rear;

    int \_k;

} MyCircularQueue;

MyCircularQueue\* myCircularQueueCreate(int k) {

    //多开一个元素的空间

    MyCircularQueue\* mq = (MyCircularQueue\*)malloc(sizeof(MyCircularQueue));

    mq->\_data = (int\*)malloc(sizeof(int) \* (k + 1));

    mq->\_front = mq->\_rear = 0;

    mq->\_k = k;

    return mq;

}

bool myCircularQueueIsEmpty(MyCircularQueue\* obj) {

    if(obj->\_front == obj->\_rear)

        return true;

    else

        return false;

}

bool myCircularQueueIsFull(MyCircularQueue\* obj) {

    if((obj->\_rear + 1) % (obj->\_k + 1) == obj->\_front)

        return true;

    else

        return false;

}

bool myCircularQueueEnQueue(MyCircularQueue\* obj, int value) {

    //判断队列是否已满

    if(myCircularQueueIsFull(obj))

        return false;

    obj->\_data[obj->\_rear++] = value;

    //判断队尾是否越界

    if(obj->\_rear > obj->\_k)

        obj->\_rear = 0;

    return true;

}

bool myCircularQueueDeQueue(MyCircularQueue\* obj) {

    //判断队列是否为空

    if(myCircularQueueIsEmpty(obj))

        return false;

    //出队

    obj->\_front++;

    //判断队头是否越界

    if(obj->\_front > obj->\_k)

        obj->\_front = 0;

    return true;

}

int myCircularQueueFront(MyCircularQueue\* obj) {

    if(myCircularQueueIsEmpty(obj))

        return -1;

    else

        return obj->\_data[obj->\_front];

}

int myCircularQueueRear(MyCircularQueue\* obj) {

    if(myCircularQueueIsEmpty(obj))

        return -1;

    //返回队尾rear的前一个位置的元素

    //判断rear是否为0;

    if(obj->\_rear != 0)

        return obj->\_data[obj->\_rear - 1];

    else

        return obj->\_data[obj->\_k];

}

void myCircularQueueFree(MyCircularQueue\* obj) {

    free(obj->\_data);

    free(obj);

}

**六、实验总结与体会：**

通过这次实训，我收获了很多，一方面学习到了许多以前没学过的专业知识与知识的应用，另一方面还提高了自己动手做项目的能力。本次实训，是对我能力的进一步锻炼，也是一种考验。从中获得的诸多收获，也是很可贵的，是非常有意义的。

但是在学习中，仍然存在一些难以解决的问题，只能通过借鉴别人的代码学习。

在实训中我学到了许多新的知识。是一个让我把书本上的理论知识运用于实践中的好机会，原来，学的时候感叹学的内容太难懂，现在想来，有些其实并不难，关键在于理解。

**二叉树建立及遍历操作**

**一、实验目的**

1．熟悉二叉树的存贮结构及遍历方式，掌握有关算法的实现。

2．能够利用二叉树解决具体问题。

**二、实验环境**

1.硬件：每个学生需配备计算机一台。

2.软件：windows 操作系统+ C、C++等。

**三、实验内容要求**

1.要求采用二叉链表作为存贮结构，完成二叉树的建立、先序、中序和后序遍历的操作。其中先序遍历和后序遍历采 用递归算法，中序遍历采用非递归算法。

2.输入数据：树中每个结点的数据类型设定为字符型。

3.要求测试多个二叉树样例，有原图。

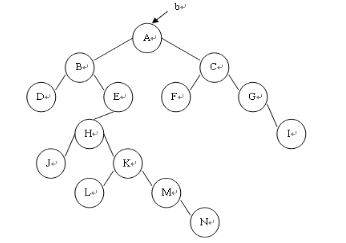
**四、实验内容**

1．在自己的U盘的“姓名+学号”文件夹中创建“实验5”文件夹，本次实验的所有程序和数据都要求存储到本文件夹中。

2．实现如下二叉树处理函数：

1）要求采用二叉链表作为存贮结构，完成二叉树的建立、先序、中序后序遍历的操作。其中先序遍历和后序遍历采用递归算法，中序遍历采用非递归算法。

2）输入数据：树中每个结点的数据类型设定为字符型。



3）实现如下二叉树处理函数：

建树子函数

先序遍历子函数

中序遍历子函数（非递归）

后序遍历子函数

**五、代码如下：**

#include<bits/stdc++.h>

using namespace std;

typedef char ElemType;

typedef struct Node/\*整棵树和结点名称\*/

{

    ElemType data;                    /\*数据域\*/

    struct Node \*lchild, \*rchild; /\*左子树和右子树\*/

} \* BiTree, BiNode;

void CreateBiTree(BiTree &T)/\*先需创建二叉树\*/

{

    char ch;

    cin >> ch;

    if (ch == '#')

        T = NULL;

    else

    {

        T = new BiNode; /\*创建一个新节点\*/

        T->data = ch;

        CreateBiTree(T->lchild);

        CreateBiTree(T->rchild);

    }

    /\*递归创建\*/

}

void InOrderTraverse(BiTree T) /\*中序遍历  递归\*/

{

    if (T)

    {

        InOrderTraverse(T->lchild);

        cout << T->data;

        InOrderTraverse(T->rchild);

    }

}

void InOrderTraverse\_non(BiTree t)/\*中序遍历  非递归\*/

{

  struct BiTNode\* St[Depth(t)], \*p;

  int top = 0;          //置空栈

  if (t != NULL){

      St[top++] = t;

    while(top){

       p = St[--top]; printf("%c ", p->data);

       if(p->rchild != NULL)

          St[top++] = p->rchild;

       if(p->lchild != NULL)

          St[top++] = p->lchild;

    }

    printf("\n");

  }

}

void PreOrderTraverse(BiTree T)/\*先序遍历\*/

{

    if (T)

    {

        cout << T->data;

        PreOrderTraverse(T->lchild);

        PreOrderTraverse(T->rchild);

    }

}

void PostOrderTraverse(BiTree T)/\*后序遍历\*/

{

    if (T)

    {

        PostOrderTraverse(T->lchild);

        PostOrderTraverse(T->rchild);

        cout << T->data;

    }

}

int NodeCount(BiTree T)/\*统计二叉树中结点的个数\*/

{

    if (T == NULL)

        return 0;

    else

        return NodeCount(T->lchild) + NodeCount(T->rchild) + 1;

}

int Depth(BiTree T)/\*求树的深度\*/

{

    if (T == NULL)

        return 0;

    else

    {

        int i = Depth(T->lchild);

        int j = Depth(T->rchild);

        return i > j ? i + 1 : j + 1;

    }

}

void Copy(BiTree T, BiTree &NewT)/\*复制二叉树\*/

{

    if (T = NULL)

    {

        NewT = NULL;

        return;

    }

    else

    {

        NewT = new BiNode;

        NewT->data = T->data;

        Copy(T->lchild, NewT->lchild);

        Copy(T->rchild, NewT->rchild);

    }

}

int LeafCount(BiTree T)/\*统计二叉树中叶子结点的个数\*/

{

    if (!T)

        return 0;

    if (!T->lchild && !T->rchild)

        return 1;

    /\*如果二叉树左子树和右子树皆为空，说明该二叉树根节点为叶子结点，结果为1\*/

    else

        return LeafCount(T->lchild) + LeafCount(T->rchild);

}

void PrintAllPath(BiTree T, char path[], int pathlen)/\*二叉树中从每个叶子结点到跟结点的路径\*/

{

    int i;

    if (T != NULL)

    {

        path[pathlen] = T->data; /\*将当前结点放入路径中\*/

        if (T->lchild == NULL && T->rchild == NULL)

        {

            /\*若这个节点是叶子结点\*/

            for (i = pathlen; i >= 0; i--)

                cout << path[i] << "　";

            cout << "\n";

        }

        else

        {

            PrintAllPath(T->lchild, path, pathlen + 1);

            PrintAllPath(T->rchild, path, pathlen + 1);

        }

    }

}

int BiTree\_empty(BiTree T)/\*判断二叉树是否为空\*/

{

    if (T)

        return 1;

    else

        return 0;

}

int main()

{

    BiTree T;

    CreateBiTree(T);

    cout << endl;

    InOrderTraverse\_non(T)；/\*中序遍历  非递归\*/

    cout << endl;

    InOrderTraverse(T);/\*中序遍历  递归\*/

    cout << endl;

    PreOrderTraverse(T);/\*先序遍历\*/

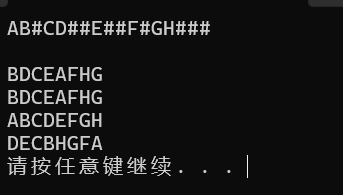
    cout << endl;

    PostOrderTraverse(T);/\*后序遍历\*/

    return 0;

}

//测试数据AB#CD##E##F#GH###

**使用测试用例运行结果如下：**

**六、实验总结与体会：**

通过这次实训，我收获了很多，一方面学习到了许多以前没学过的专业知识与知识的应用，另一方面还提高了自己动手做项目的能力。本次实训，是对我能力的进一步锻炼，也是一种考验。从中获得的诸多收获，也是很可贵的，是非常有意义的。

但是在学习中，仍然存在一些难以解决的问题，只能通过借鉴别人的代码学习。

在实训中我学到了许多新的知识。是一个让我把书本上的理论知识运用于实践中的好机会，原来，学的时候感叹学的内容太难懂，现在想来，有些其实并不难，关键在于理解。

**霍夫曼编码及应用**

**一、实验目的**

1．进一步掌握二叉树的存储结构和相应算法。

2．掌握霍夫曼树的创建和霍夫曼编码。

**二、实验环境**

1.硬件：每个学生需配备计算机一台。

2.软件：windows 操作系统+ C、C++等。

**三、实验内容要求**

1．要求采用二叉链表作为存贮结构，完成霍夫曼树的创建。

2．输出对应数据的霍夫曼编码，并求出平均编码长度。

**四、实验内容**

1．在自己的U盘的“姓名+学号”文件夹中创建“实验4”文件夹，本次实验的所有程序和数据都要求存储到本文件夹中。

2．现在某电报公司假设有10字符进行编码，这10个字符的使用频率如下表所示，请创建霍夫曼树（也可结合日常生活及社会应用实例，不限于本实例）。

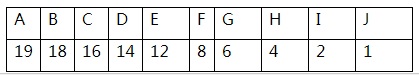
1）要求采用二叉链表作为存贮结构，完成霍夫曼树的创建；

2）输出对应字符的霍夫曼编码，并求出平均编码长度。

3\*):建树编码原则：

左子树编码0，右子树1；

建树过程中：权值小为左孩子；有同权值的，优先选择树深浅的子树。



3、同时用哈夫曼算法思路完成如下题目：（测试数据请参见“附件”或“答案解析”）：

要求针对每一条输出相应输出答案，运行结果抓图：答案输出形如：

test01:\*\*\* test2:\*\*\* 。

**五、代码如下：**

#include <bits/stdc++.h>

using namespace std;

typedef struct{

   int weight;

   int parent;

   int lChild;

   int rChild;

} Node, \*HuffmanTree;

typedef char \*HuffmanCode;

void select(HuffmanTree \*huffmanTree, int n, int \*s1, int \*s2)

{

    int i=0,min;

    for(i = 1; i <= n; i++){

        if((\*huffmanTree)[i].parent == 0){

            min = i;

            break;

        }

    }

    for(i = 1; i <= n; i++)

      if((\*huffmanTree)[i].parent == 0)

        if((\*huffmanTree)[i].weight < (\*huffmanTree)[min].weight)

          min = i;

    \*s1 = min;

    for(i = 1; i <= n; i++){

        if((\*huffmanTree)[i].parent == 0 && i != (\*s1)){

            min = i;

            break;

        }

    }

    for(i = 1; i <= n; i++)

      if((\*huffmanTree)[i].parent == 0 && i != (\*s1))

        if((\*huffmanTree)[i].weight < (\*huffmanTree)[min].weight)

          min = i;

    \*s2 = min;

}

void createHuffmanTree(HuffmanTree \*huffmanTree, int w[], int n)

{

    int m = 2 \* n - 1;

    int s1,s2,i;

    \*huffmanTree = (HuffmanTree)malloc((m + 1) \* sizeof(Node));

    for(i = 1; i <= n; i++){

        (\*huffmanTree)[i].weight = w[i];

        (\*huffmanTree)[i].lChild = 0;

        (\*huffmanTree)[i].parent = 0;

        (\*huffmanTree)[i].rChild = 0;

    }

    for(i = n + 1; i <= m; i++){

        (\*huffmanTree)[i].weight = 0;

        (\*huffmanTree)[i].lChild = 0;

        (\*huffmanTree)[i].parent = 0;

        (\*huffmanTree)[i].rChild = 0;

    }

    for(i = n + 1; i <= m; i++){

        select(huffmanTree, i-1, &s1, &s2);

        (\*huffmanTree)[s1].parent = i;

        (\*huffmanTree)[s2].parent = i;

        (\*huffmanTree)[i].lChild = s1;

        (\*huffmanTree)[i].rChild = s2;

        (\*huffmanTree)[i].weight = (\*huffmanTree)[s1].weight + (\*huffmanTree)[s2].weight;

    }

    cout<<(\*huffmanTree)[m].weight;

}

int main(void)

{

    HuffmanTree HT;

    int \*w,i,n,wei,m;

    scanf("%d",&n);

    w=(int \*)malloc((n+1)\*sizeof(int));

    for(i=1; i<=n; i++){

        fflush(stdin);

        scanf("%d",&wei);

        w[i]=wei;

    }

    createHuffmanTree(&HT, w, n);

    return 0;

}

Test 1

Input:

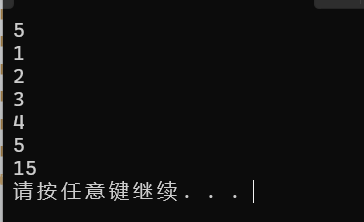
5

1 2 3 4 5

Output:

15

运行结果截图：



Test 1

Input:

10

66 4 3 2 44 6 7 45 3 5

Output:

185

运行结果截图：



**六、实验总结与体会：**

这次的哈夫曼树也是结合了课本上的理论知识以及网络上的视频讲解才初步了解。

至于最终的实现，借鉴了网络上的大量大佬的代码和解析才得到最终的代码。

前些时间在动态规划的学习中也接触到“合并糖果”这类题，在自己又回顾先前的代码，发现动态规划与树的结构有些类似，奈何本人脑子不灵光，只能得知二者在结构上的相似，至于其原理，目前并未理解透。

另外原本想通过简单的数组解决“合并糖果”问题，后来发现自己想的太简单了，也就不了了之，最终呈现上如上代码。

**图的最小生成树算法的实现及应用**

**实验预备知识：**

1．理解图最小生成树的意义和相应算法。

2．掌握带权图的存储结构

**一、实验目的**

1．使学生熟悉最小生成树的算法实现及应用。

2．掌握带权图的存储结构和处理方法。

**二、实验环境**

1.硬件：每个学生需配备计算机一台。

2.软件：windows 操作系统+ C、C++等。

**三、实验内容要求**

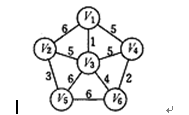
1．能够独立完成带权图的存储和最小生成树的生成；

2、并能灵活把最小生成树应用于解决现实生活中的问题。

**四、实验内容**

1．在自己的U盘的“姓名+学号”文件夹中创建“实验5”文件夹，本次实验的所有程序和数据都要求存储到本文件夹中。

2．现在某电信公司要对如下图的几个城市之间进行光纤连接布线，请用合适的存储结构将下图存储到计算机中方便进行处理。



3．现在公司想以最小的代价将所有城市连通，方便所有城市间通信，请用普里姆算法和克鲁斯卡尔算法实现本图的最小生成树（此类算法为贪心算法）。

4、同时所有同学需要掌握无负边无自环的图的并查集的应用。

5、完成如下实际应用问题（输入数据参见附件）：最小方差生成树题目描述：

给定带权无向图，求出一颗方差最小的生成树。

输入格式：

输入多组测试数据。第一行为N,M，依次是点数和边数。接下来M行，每行三个整数U,V,W，代表连接U,V的边，和权值W。保证图连通。n=m=0标志着测试文件的结束。

输出格式：

对于每组数据，输出最小方差，四舍五入到0.01。输出格式按照样例。

样例输入

4 5

1 2 1

2 3 2

3 4 2

4 1 1

2 4 3

4 6

1 2 1

2 3 2

3 4 3

4 1 1

2 4 3

1 3 3

0 0

样例输出

Case 1: 0.22

Case 2: 0.00

数据规模与约定

1<=U,V<=N<=50,N-1<=M<=1000,0<=W<=50。

数据输入参见附件！

\*输出请依照输入依次输出（抓图）：

如INPUT1输出为：\*\*\*\*

**五、代码如下：**

#include <bits/stdc++.h>

using namespace std;

double const MAX = 10000000000000.0;

int n, m, tmp[1005], fa[55];

double ans;

struct Edge

{

    int u, v;

    double w, val;

}e[1005];

bool cmp(Edge a, Edge b)

{

    return a.w < b.w;

}

void UF\_set(int n)

{

    for(int i = 1; i <= n; i++)

        fa[i] = i;

}

int Find(int x)

{

    return x == fa[x] ? x : fa[x] = Find(fa[x]);

}

void Union(int a, int b)

{

    int r1 = Find(a);

    int r2 = Find(b);

    if(r1 != r2)

        fa[r2] = r1;

}

void Kruskal(int sum)

克鲁斯卡尔算法

{

    UF\_set(n);

    int cnt = 0;

    double f\_all = 0;

    double all = 0;

    double ave = sum \* 1.0 / (n - 1);

    for(int i = 0; i < m; i++)

        e[i].w = (e[i].val - ave) \* (e[i].val - ave);

    sort(e, e + m, cmp);

    for(int i = 0; i < m; i++)

    {

        int u = e[i].u;

        int v = e[i].v;

        if(Find(u) != Find(v))

        {

            Union(u, v);

            f\_all += e[i].w;

            all += e[i].val;

            cnt ++;

        }

        if(cnt == n - 1)

            break;

    }

    if((int)all == sum)

        ans = min(ans, f\_all);

}

// int prim(int pos) {

普利姆算法

//     w[pos] = 0;

//     for(int i = 1; i <= n; i ++) {

//         int cur = -1;

//         for(int j = 1; j <= n; j ++) {

//             if(!v[j] && (cur == -1 || w[j] < w[cur])) {

//                 cur = j;

//             }

//         }

//         if(w[cur] >= MAX) return MAX;

//         ans += w[cur];

//         v[cur] = 1;

//         for(int k = 1; k <= n; k ++) {

//             if(!v[k]) w[k] = min(w[k],a[cur][k]);

//         }

//     }

//     return ans;

// }

int main()

{

    int ca = 1;

    while(scanf("%d %d", &n, &m) != EOF && (m + n))

    {

        // if(n == 1 || n == 2)

        // {

        //     printf("0.00\n");

        //     continue;

        // }

        int minv = 0;

        int maxv = 0;

        ans = MAX;

        for(int i = 0; i < m; i++)

        {

            scanf("%d %d %lf", &e[i].u, &e[i].v, &e[i].val);

            tmp[i] = e[i].val;

        }

        sort(tmp, tmp + m);

        for(int i = 0; i < n - 1; i++)

            minv += tmp[i];

        for(int i = m - 1; i > m - n; i--)

            maxv += tmp[i];

        for(int i = minv; i <= maxv; i++)

            Kruskal(i);

        // cout<<ans;

        ans = ans / (n - 1);

        cout<<"Case "<<ca<<": "<<fixed<<setprecision(2)<<ans<<endl;

        ca++;

}

    for(int i=1;i<=ca;i++)

    {

        cout<<"Case "<<i<<": "<<fixed<<setprecision(2)<<aaa[i]<<endl;

    }

**方便后续抓图**

}

**测试用例及结果**

样例输入1

3 3

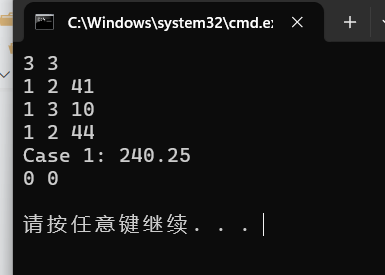
1 2 41

1 3 10

1 2 44

0 0

抓图



样例输入2

3 5

1 2 33

2 3 35

1 2 44

2 3 16

1 1 8

3 5

1 2 6

2 3 26

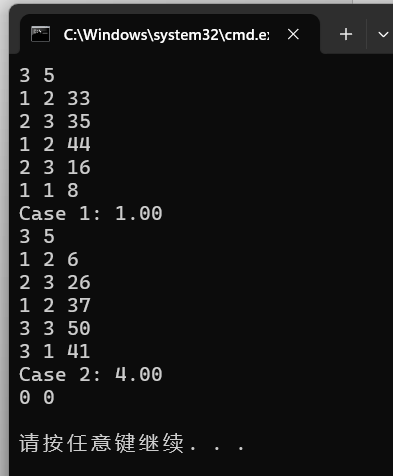
1 2 37

3 3 50

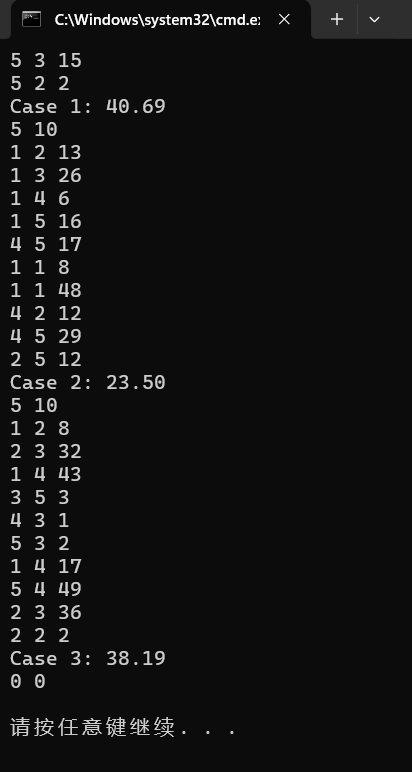
3 1 41

0 0

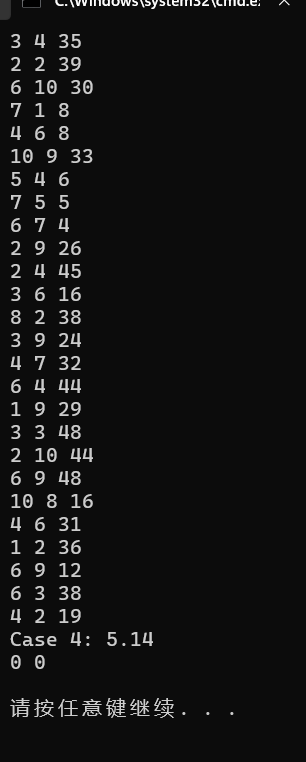
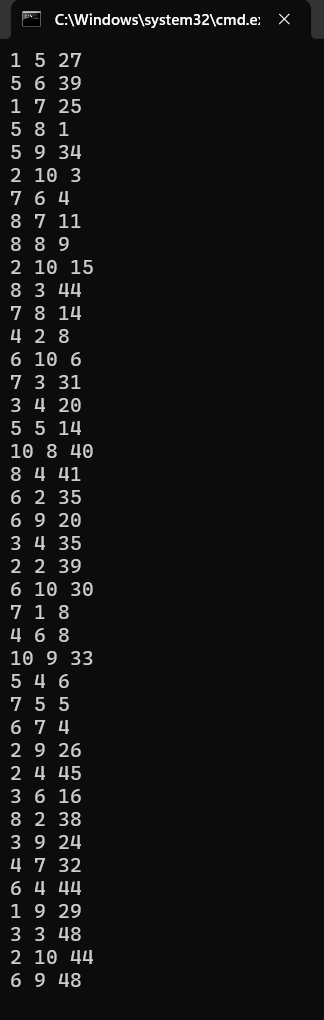
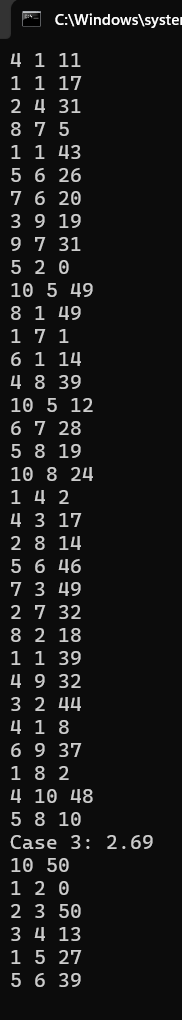
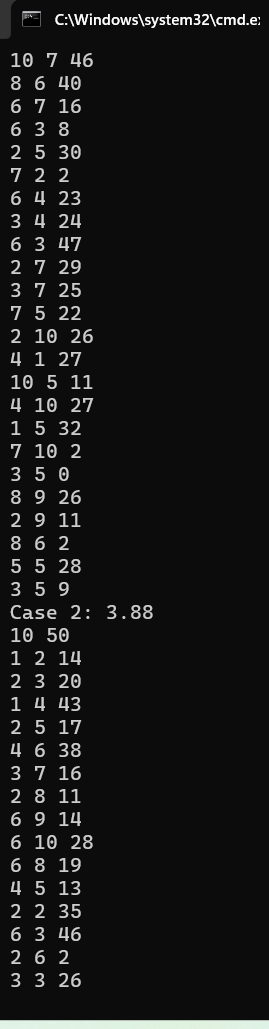
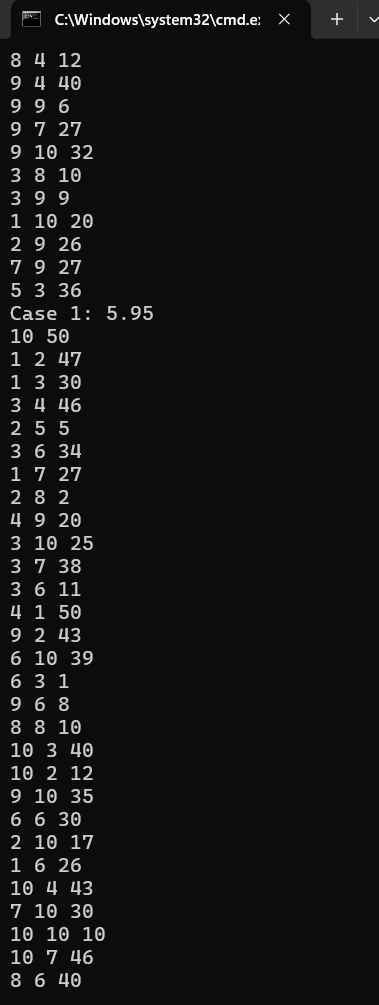
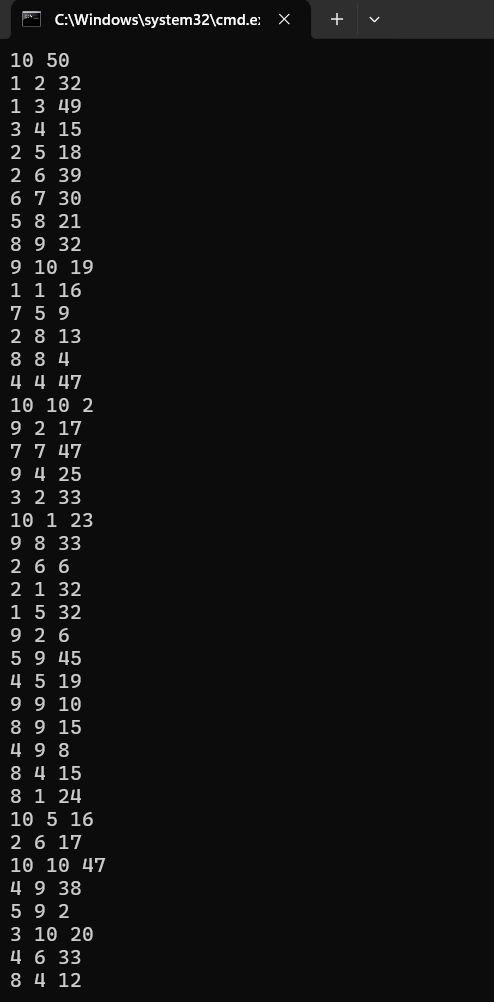
抓图



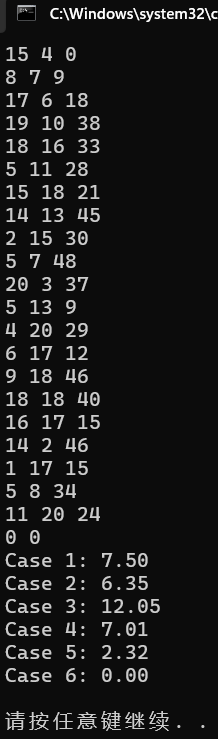
样例输入3，运行结果抓图



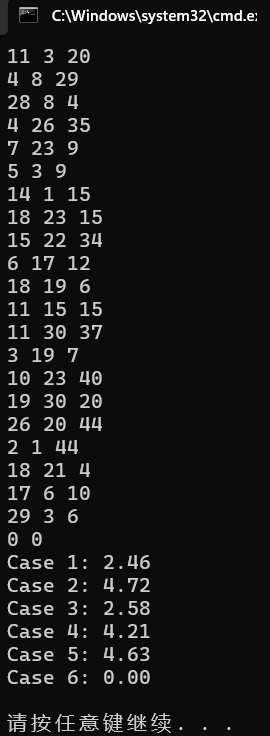
样例输入4，运行结果抓图



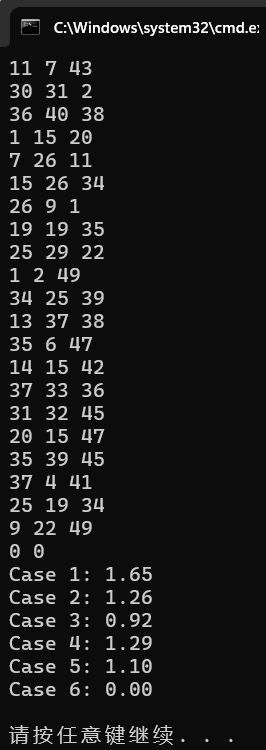
样例输入5，运行结果抓图



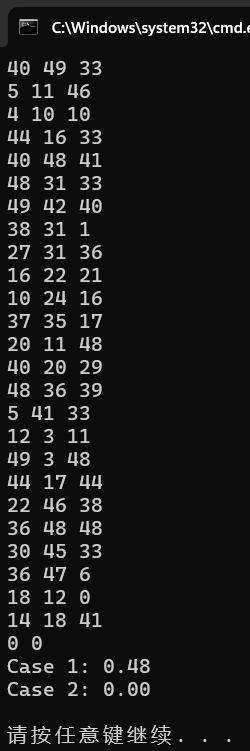
样例输入6抓图



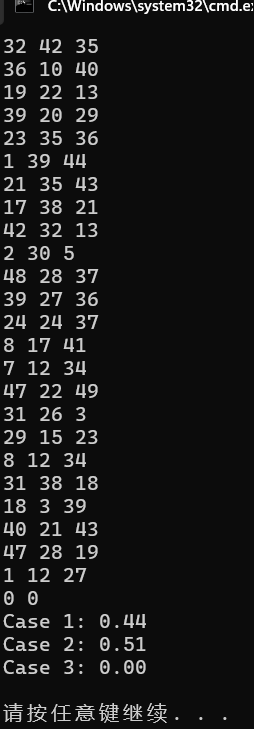
样例输入7，运行结果抓图



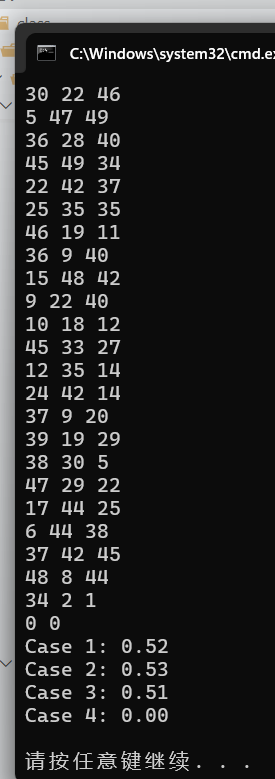
样例输入8，运行结果抓图



样例输入9，运行结果抓图



样例输入10，运行结果抓图



**六、实验总结与体会：**

通过这次实训，我收获了很多，一方面学习到了许多以前没学过的专业知识与知识的应用，另一方面还提高了自己动手做项目的能力。本次实训，是对我能力的进一步锻炼，也是一种考验。从中获得的诸多收获，也是很可贵的，是非常有意义的。

但是在学习中，仍然存在一些难以解决的问题，只能通过借鉴别人的代码学习。

此外，这次的实际用例晦涩难懂，问了同学，上网上搜索大量网页才大致搞懂题目的意思，在同学的帮助下理解了在实际生活中的运用，也是扩展了我的视野。

**图的最短路径算法实现及应用**

**实验预备知识：**

1．理解图最小生成树的意义和相应算法。

2．掌握带权图的存储结构

**一、实验目的**

1．使学生熟悉最小生成树的算法实现及应用。

2．掌握带权图的存储结构和处理方法。

**二、实验环境**

1.硬件：每个学生需配备计算机一台。

2.软件：windows 操作系统+ C、C++等。

**三、实验要求**

1．能够独立完成带权图的存储和最短路径的生成。

2、熟练掌握并实现图的最短路径Dijkstra和Floyd算法及其应用。

3、完成要求的竞赛题目（实际工程应用）：参见题目描述。

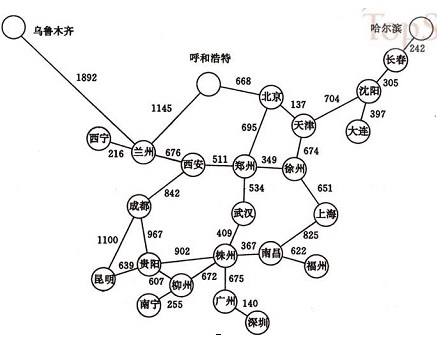
**四、实验内容**

1．在自己的U盘的“姓名+学号”文件夹中创建“实验6”文件夹，本次实验的所有程序和数据都要求存储到本文件夹中。

2．现在假设我国铁路交通图如下（权值表示距离），请用合适的存储结构将下图存储到计算机中方便进行处理。

3．现在想以最小的代价从武汉出发到达其他城市目的地，请用Dijkstra和Floyd算法分别实现要求的路径。

4.完成如下问题的程序设计思路框架及说明、代码编写、调试及运行结果抓图（抓图时不需要抓输入数据，只抓运行的输出结果！形式如：TEST1:\*\*\*\*;TEST2:\*\*\*\*）:测试数据参照附件。



4665: 畅通工程

时间限制 : 1.000 sec 内存限制 : 128 MB

题目描述

魔法世界的道路四通八达，不过路多了也不好，每次要从一个城镇到另一个城镇时，都有许多种道路方案可以选择，而某些方案要比另一些方案行走的距离要短很多，这让行人很困扰。

现在已知起点和终点，请你计算出要从起点到终点，最短需要行走多少距离。

输入

第一行包含两个正整数N和M（0＜N＜200，0＜M＜1 000），分别代表现有城镇的数目和已修建的道路的数目。城镇分别以0～N－1编号。

接下来是M行道路信息。每一行有三个整数A，B，X（0≤A，B＜N，A≠B，0＜X＜10 000），表示城镇A和城镇B之间有一条长度为X的双向道路。

再接下一行有两个整数S，T（0≤S，T＜N），分别代表起点和终点。

输出

输出最短需要行走的距离。如果不存在从S到T的路线，就输出-1。

样例输入

3 3

0 1 1

0 2 3

1 2 1

0 2

样例输出

2

**五、代码如下：**

Floyd算法：

#include<bits/stdc++.h>

using namespace std;

const int INF = 0x3f3f3f3f;

const int N = 200;

int g[N + 1][N + 1], n;

void floyd()

{

    for(int k = 0; k < n; k++)

    {

        for(int i = 0; i < n; i++)

        {

            for(int j = 0; j < n; j++)

            {

                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);  //以k为中间点，找到更小的距离

            }

        }

    }

}

int main()

{

    int m, u, v, w;

    cin>>n>>m;

    memset(g, INF, sizeof(g));//初始化，使每个点之间的距离最大（无穷）

    for (int i=1;i<=m;i++) //存图

    {

        cin>>u>>v>>w;

        if (w<g[u][v])

        {

            g[u][v]=w;

            g[v][u]=w;

        }

    }

    floyd();

    cin>>u>>v;

    if(u==v)//同一点

    {

        cout<<0;

    }

    else if(g[u][v]<INF)//两点之间没有直接通路，使用计算出的最短路径

    {

        cout<<g[u][v];

    }

    else//两点之间不通

    {

        cout<<-1;

    }

    return 0;

}

Dijkstra算法

#include <bits/stdc++.h>

using namespace std;

const int INF = ((unsigned int)(-1) >> 1);

const int MAXN = 200;

struct Edge // 边

{

    int v, cost;

    Edge(int v2, int c) {v=v2; cost=c;}

};

struct Node // 结点

{

    int u, cost;

    Node(){}

    Node(int u2, int l)

    {

        u=u2;

        cost=l;

    }

    bool operator<(const Node n) const

    {

        return cost > n.cost;

    }

};

vector<Edge> g[MAXN+1];

int dist[MAXN+1];

bool vis[MAXN+1];

void dijkstra(int start, int n)

{

    priority\_queue<Node> q;

    for (int i = 0; i <= n; i++)

    {

        dist[i] = INF;

        vis[i] = false;

    }

    dist[start] = 0;

    q.push(Node(start, 0));

    Node f;

    while (!q.empty())

    {

        f = q.top();

        q.pop();

        int u = f.u;

        if (!vis[u])

        {

            vis[u] = true;

            int len = g[u].size();

            for (int i = 0; i < len; i++)

            {

                int v2 = g[u][i].v;

                if(vis[v2])

                {

                    continue;

                }

                int tempcost = g[u][i].cost;

                int nextdist = dist[u] + tempcost;

                if (dist[v2] > nextdist)

                {

                    dist[v2] = nextdist;

                    q.push(Node(v2, dist[v2]));

                }

            }

        }

    }

}

int main()

{

    int n, m, u, v, w, s, t;

    cin>>n>>m;

    for(int i=1;i<=m;i++)

    {

        cin>>u>>v>>w;

        g[u].push\_back(Edge(v, w));

        g[v].push\_back(Edge(u, w));

    }

    cin>>s>>t;

    dijkstra(s, n);

    if(dist[t]==INF)

    {

        cout<<-1;

    }

    else

    {

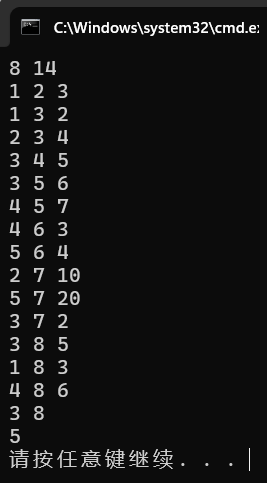
        cout<<dist[t];

    }

    return 0;

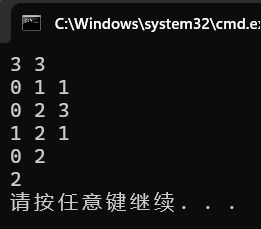
}

Test0运行结果：



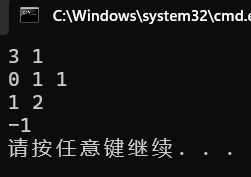
正确

Test1运行结果：



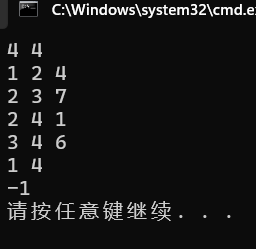
正确

Test2运行结果：



正确

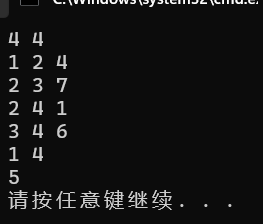
Test3运行结果：



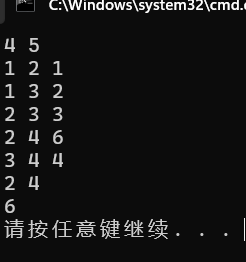
使用第一段代码错误 大概是由于超限

使用第二段Dijkstra算法正确，

不是很明白为什么

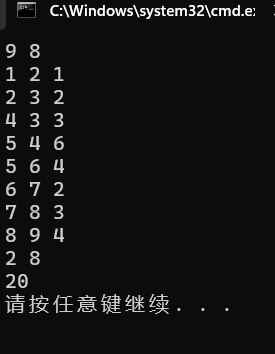


Test4运行结果：



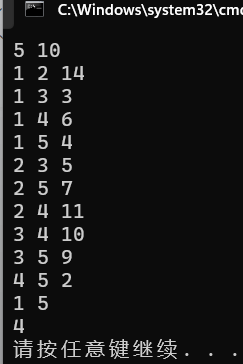
正确

Test5运行结果：



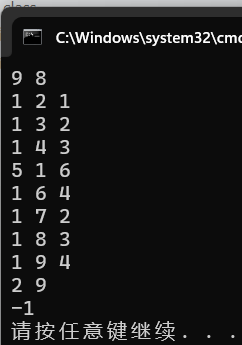
正确

Test6运行结果：

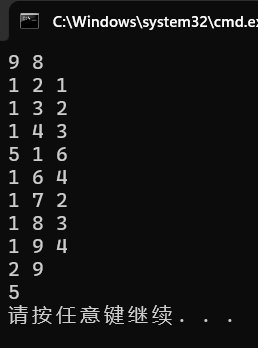


正确

Test7运行结果：

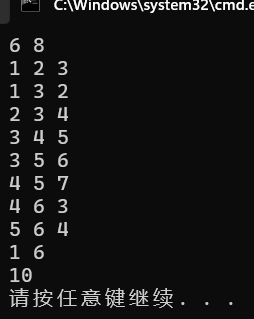


与test3类似，floyd算法错误



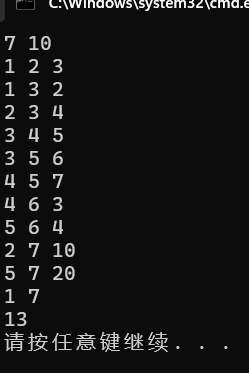
Dijkstra算法正确

Test8运行结果：



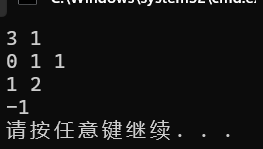
正确

Test9运行结果：



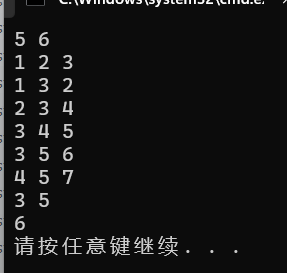
正确

Test10运行结果：



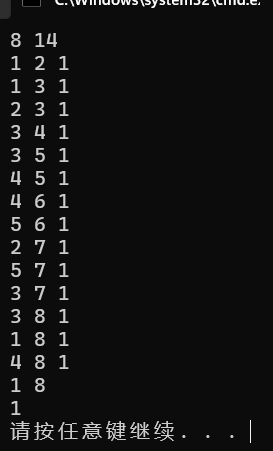
正确

Test11运行结果：



正确

Test12运行结果：



正确

在前面的“国家铁路交通图”题中，可以以上述题目作为模板，触类旁通。

|  |  |
| --- | --- |
| 北京 | 1 |
| 成都 | 2 |
| 大连 | 3 |
| 福州 | 4 |
| 广州 | 5 |
| 贵阳 | 6 |
| 哈尔滨 | 7 |
| 呼和浩特 | 8 |
| 昆明 | 9 |
| 兰州‘ | 10 |
| 柳州 | 11 |
| 南昌 | 12 |
| 南宁 | 13 |
| 上海 | 14 |
| 深圳 | 15 |
| 沈阳 | 16 |
| 天津 | 17 |
| 乌鲁木齐 | 18 |
| 武汉 | 19 |
| 西安 | 20 |
| 徐州 | 21 |
| 长春 | 22 |
| 郑州 | 23 |

因此，现将图中的城市以拼音作为排序，并赋予编号如下：

在获得这些编号后，通过数图中的边数量，顶点数量（23个城市），以及每个边（距离）的数值，即可通过上述方式获得所求最短距离。

存储结构：以二维数组可以存入少量的数据，适合铁路分布图。

**六、实验总结与体会：**

通过这次实训，我能够通过典例有所感悟，并应用到实际的生活中：铁路分布最短距离。但是有所不足的是，自己理解了两个算法的原理和具体代码，但是要从头自己手敲，并且不借鉴同学和网络上的代码仍有些距离，自己并未完全吃透这两个算法。因此我将多练类似的题目，多看网上的教学视频，以此来增强自己对这两个算法的认识，争取以最快速度做到自己完全独立完成类似题目。

有所不足的是，由于对于这两个的算法认识不够深刻，test3与test7中floyd算法存在的问题自己无法解决。

此外，以特殊扩散到普遍，从简单进入到复杂，这是我在此次实验中所学到的哲学道理，也告诉我学习的步骤，收获颇多。

**哈希查找及应用**

**实验预备知识：**

1．理解哈希函数和哈希表。

2．掌握各种哈希函数和冲突解决办法。

**一、实验目的**

1．掌握什么是哈希表和哈希函数。

2．掌握哈希表的构建和哈希查找。

**二、实验环境**

1.硬件：每个学生需配备计算机一台。

2.软件：windows 操作系统+ C、C++等。

**三、实验要求**

1．分析并设计一个哈希表。哈希函数用除留余数法构造，用线性探测再散列处理冲突。

2．学生成绩报告单。任给任一学生学号即可打印出该生的成绩报告

3．学生学号可以不按照顺序。

4.同时完成本实验中的附加应用题目。

**四、实验内容**

1、设计并完成如下哈希实验：

1）设计一个哈希表。哈希函数用除留余数法构造，用线性探测再散列处理冲突；

2）学生成绩报告单。任给任一学生学号即可打印出该生的成绩报告单；

3）某个学院有200名同学，每个同学记录包括：班级、学号、姓名和语文、数学、外语等三门课程成绩（学号为2位整数）；

4）以学号为主关键字，用除留余数法构造哈希函数，用线性探测再散列处理冲突构建哈希表；

5）输入任何一个学生学号，输出该学生的信息；

6）用链地址法在实现。

2、完成如下的哈希应用问题：（测验用数据参见附件）

相同的雪花

时间限制 : 3.000 sec 内存限制 : 128 MB

题目描述

现在你的任务是编写一个程序比较你所处的世界是否有相同的雪花，已知每朵雪花有六瓣，如果两朵雪花彼此对应的花瓣有相同的长度则说明是一样的。

输入

第一行包括一个整数n（0＜n≤100 000）表示雪花的数目。随后n行描述每一朵雪花。每朵雪花有六个整数，每个整数范围在（0~10 000 000)之间，表示雪花六个花瓣的长度。六个整数的先后出现顺序可能是雪花花瓣的顺时针顺序也可能是逆时针顺序，并且可能是从任一个花瓣位置开始的。比如说，对同一个雪花，描述的方法可能是1 2 3 4 5 6 或者 4 3 2 1 6 5。

输出

如果没有相同的雪花，输出“No two snowflakes are alike.”，否则输出“Twin snowflakes found.”。

样例输入

2

1 2 3 4 5 6

4 3 2 1 6 5

样例输出

Twin snowflakes found.

**五、代码如下：**

#include<bits/stdc++.h>

using namespace std;

const int maxn = 100010;

const int P = 100007;

int head[P] , cnt;

int snow[maxn][6];

int n;

struct node{

    int to, next;

}e[maxn];

void init(){

    cnt = 0;

    memset(head , -1, sizeof(head));

}

void addhash(int val ,int i){ //将下标i 添加到值为val 的链表中

    e[++cnt].to = i;

    e[cnt].next = head[val];

    head[val] = cnt;

}

int cmp(int a , int b){ //比较雪花

    int i , j;

    for(int i = 0 ; i < 6; i++){

        if(snow[a][0] == snow[b][i]){

            for(j = 1 ; j < 6 ; j ++){ //顺时针

                if(snow[a][j] != snow[b][(j + i) % 6]){

                    break;

                }

            }

            if(j == 6){

                return 1;

            }

            for(j = 1; j < 6 ; j ++){ //逆时针

                if(snow[a][6 - j] != snow[b][(j + i ) % 6]){

                    break;

                }

            }

            if(j == 6){

                return 1;

            }

        }

    }

    return 0;

}

bool find(int i){

    int key , sum = 0;

    for(int j = 0 ; j < 6 ; j ++){

        sum += snow[i][j];

    }

    key = sum % P;

    for(int j = head[key] ; j ; j = e[j].next){

        if(cmp(i , e[j].to)){

            return 1;

        }

    }

    addhash(key , i);

    return 0;

}

int main(){

    scanf("%d", &n);

    int flag = 0;

    int sum, key;

    for(int i = 0 ; i < n ; i++){

        for(int j = 0 ; j < 6 ; j ++){

            scanf("%d", &snow[i][j]);

        }

        if(find(i)){

            flag = 1;

            break;  //如果多组测试用例，则continue 继续读入

        }

    }

    if(flag){

        puts("Twin snowflakes found.");

    }

    else{

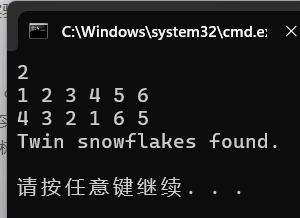
        puts("No two snowflakes are alike.");

    }

    return 0;

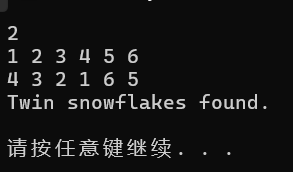
}

Sample1输出



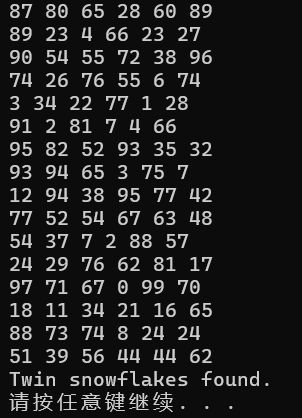
正确

Test0输出



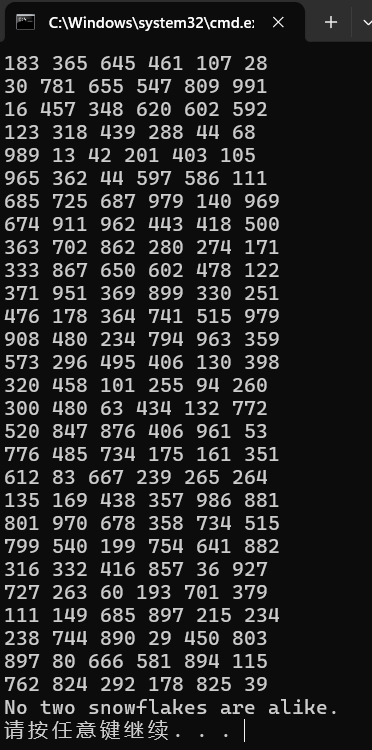
正确

Test1输出



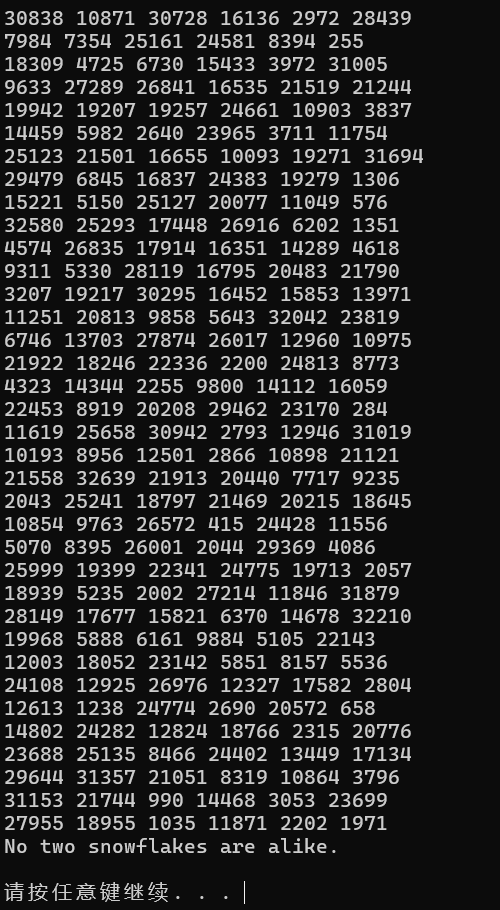
正确

Test2输出、



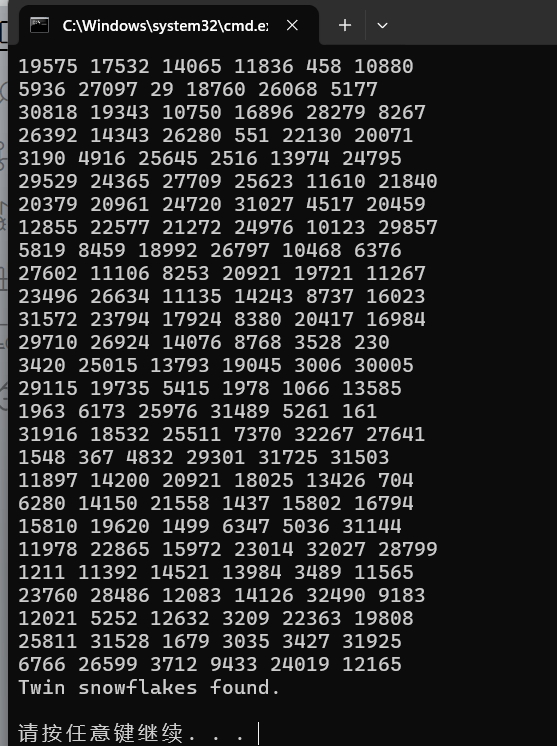
正确

Test8输出



正确

Test9输出



正确

**学生成绩报告单**

代码如下

#include<bits/stdc++.h>

using namespace std;

const int N = 100;

struct Student {

    int studentID;

    string name;

    string className;

    int chineseGrade;

    int mathGrade;

    int englishGrade;

};

Student\* hashTable[N] = {nullptr};

int hashFunction(int key) {

    return key % N;

}

void insert(Student\* student) {

    int index = hashFunction(student->studentID);

    while (hashTable[index] != nullptr) {

        index = (index + 1) % N;

    }

    hashTable[index] = student;

}

Student\* search(int studentID) {

    int index = hashFunction(studentID);

    while (hashTable[index] != nullptr && hashTable[index] -> studentID != studentID) {

        index = (index + 1) % N;

    }

    return hashTable[index];

}

int main() {

    int n;

    cout << "请输入学生个数:";

    cin >> n;

    cout << "请输入学生信息:" << endl;

    for(int i = 0 ; i < n; i++){

        Student\* stu = new Student;

        cin >> stu->studentID >> stu->name >> stu->className >> stu->chineseGrade >> stu->mathGrade >> stu->englishGrade;

        insert(stu);

    }

    cout << "请输入查询学生学号:";

    int studentID;

    cin >> studentID;

    Student\* result = search(studentID);

    if (result != nullptr) {

        cout << "学号：" << result->studentID << endl;

        cout << "姓名：" << result->name << endl;

        cout << "班级：" << result->className << endl;

        cout << "语文成绩：" << result->chineseGrade << endl;

        cout << "数学成绩：" << result->mathGrade << endl;

        cout << "外语成绩：" << result->englishGrade << endl;

    }

    else {

        cout << "学生信息未找到！" << endl;

    }

    for (int i = 0; i < N; i++){

        if (hashTable[i] != nullptr){

            delete hashTable[i];

        }

    }

    return 0;

}

/\*

测试数据

15 zhangsan class1 99 80 60

34 lisi class4 69 96 100

21 wangwu class1 70 80 85

11 laoliu class3 89 69 77

17 zhaoqi class2 88 66 70

30 ann class5 80 61 78

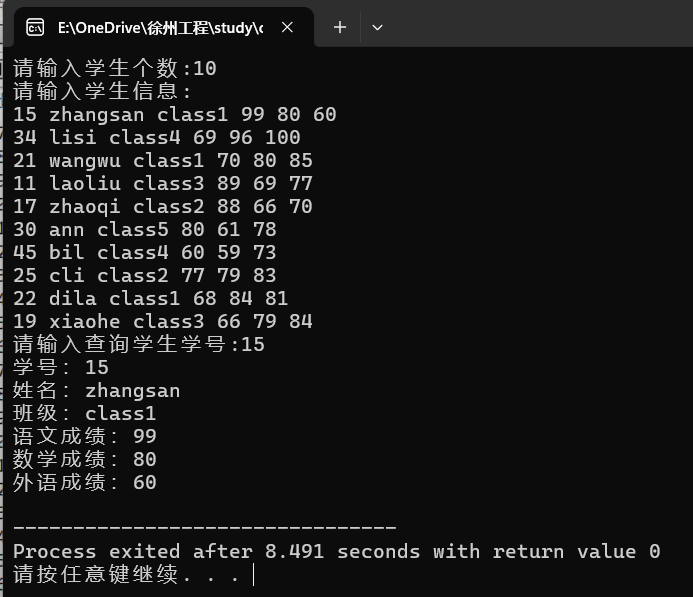
45 bil class4 60 59 73

25 cli class2 77 79 83

22 dila class1 68 84 81

19 xiaohe class3 66 79 84

\*/



**六、实验总结与体会：**

通过这次实训，我能够通过典例有所感悟，并应用到实际的生活中：铁路分布最短距离。但是有所不足的是，自己理解了两个算法的原理和具体代码，但是要从头自己手敲，并且不借鉴同学和网络上的代码仍有些距离，自己并未完全吃透这两个算法。因此我将多练类似的题目，多看网上的教学视频，以此来增强自己对这两个算法的认识，争取以最快速度做到自己完全独立完成类似题目。

有所不足的是，有的测试数据太大导致程序无法运行出来，或者得花费大量时间来运行最终结果，最后我在网上查找了一些文章解决的部分的问题，最终结果呈现如上。此外，以特殊扩散到普遍，从简单进入到复杂，这是我在此次实验中所学到的哲学道理，也告诉我学习的步骤，收获颇多。

**排序及其应用**

**实验预备知识：**

1．理解并掌握各种常用内部排序算法的基本概念、思想和方法。

2．掌握常用内部排序算法流程。

**一、实验目的**

1．掌握常用的排序方法，深刻理解排序的定义和各种排序方法的特点。

2．通过实验观察不同方法的不同之处，记录并分析各种排序方法的结果。

**二、实验环境**

1.硬件：每个学生需配备计算机一台。

2.软件：windows 操作系统+ C、C++等。

**三、实验内容要求**

在教科书中，各种内部排序算法的时间复杂度分析结果只给出了算法执行时间的阶，或大概执行时间。试通过随机数据比较各算法的关键字比较次数和关键字移动次数，以取得直观感受。要求：

1.理解及熟练运用直接插入排序、快速排序、堆排序和归并排序、哈希排序等内部排序算法。

2.通过计数统计各算法的关键字比较次数和关键字移动次数。

3.分析算法的时间复杂度、空间复杂度及稳定性等各项指标。

**四、实验内容**

1．在自己的U盘的“姓名+学号”文件夹中创建“实验16”文件夹，本次实验的所有程序和数据都要求存储到本文件夹中。

2．设计随机数来测试排序算法运行时间的程序，其中可以通过修改RANDNUM的值来更改测试的数据量。具体参考如下：

#define RANDNUM 10000 //随机数的个数

for(i=0;i<RANDNUM;i++)

{//产生1万个随机数

iRandNum[i]=rand()%RANDNUM;

}

3．排序算法进行测试（也和上一个实验结果进行比较），记录运行时间；把需排序的数改为20000，进行同样测试，并比较测试结果。

4. 通过随机的数据比较各算法的关键字比较次数和关键字移动次数，取得直观感受并分析算法各项性能指标。（随机数据要用伪随机数产生程序产生；至少要用5组不同的输入数据作比较；比较的指标为有关键字参加的比较次数和关键字的移动次数（关键字交换计为3次移动））

**五、代码如下：**

#include<bits/stdc++.h>

#define RANDNUM 15000 // 随机数的个数

using namespace std;

void QuickSort(int \*arr, int low, int high)//快速排序

{

    int i = low;

    int j = high;

    int k = arr[low];

    if (low > high)

        return;

    while (i < j)

    {

        while (i < j && arr[j] >= k) // 从右向左找第一个小于k的数

            j--;

        if (i < j)

            arr[i++] = arr[j];

        while (i < j && arr[i] < k) // 从左向右找第一个大于等于k的数

            i++;

        if (i < j)

            arr[j--] = arr[i];

    }

    arr[i] = k;

    // 递归调用

    QuickSort(arr, low, i - 1);  // 排序k左边

    QuickSort(arr, i + 1, high); // 排序k右边

}

void shellSort(int \*a, int len)//希尔排序

{

    int i, j, k, tmp, gap; // gap 为步长

    for (gap = len / 2; gap > 0; gap /= 2)

    { // 步长初始化为数组长度的一半，每次遍历后步长减半,

        for (i = 0; i < gap; ++i)

        { // 变量 i 为每次分组的第一个元素下标

            for (j = i + gap; j < len; j += gap)

            {                // 对步长为gap的元素进行直插排序，当gap为1时，就是直插排序

                tmp = a[j];  // 备份a[j]的值

                k = j - gap; // j初始化为i的前一个元素（与i相差gap长度）

                while (k >= 0 && a[k] > tmp)

                {

                    a[k + gap] = a[k]; // 将在a[i]前且比tmp的值大的元素向后移动一位

                    k -= gap;

                }

                a[k + gap] = tmp;

            }

        }

    }

}

void InsertSort(int \*a, int len)//插入排序

{

    for (int j=1; j<len; j++)

    {

        int key = a[j];

        int i = j-1;

        while (i>=0 && a[i]>key)

        {

            a[i+1] = a[i];

            i--;

        }

        a[i+1] = key;

    }

}

//------------------堆排序

// Heapsort.cpp : 定义控制台应用程序的入口点。

void swap(int arr[], int a, int b)      //交换元素；

{

    int temp = arr[a];

    arr[a] = arr[b];

    arr[b] = temp;

}

void adjustHeap(int arr[], int i, int length)       //调整大顶堆（仅是调整过程，建立在大顶堆已构建的基础上）

{

    int temp = arr[i];//先取出当前元素i

    for (int k = i \* 2 + 1; k<length; k = k \* 2 + 1)//从i结点的左子结点开始，也就是2i+1处开始

    {

        if (k + 1<length&&arr[k]<arr[k + 1])//如果左子结点小于右子结点，k指向右子结点

        {

            k++;

        }

        if (arr[k] >temp)//如果子节点大于父节点，将子节点值赋给父节点（不用进行交换）

        {

            arr[i] = arr[k];

            i = k;

        }

        else

        {

            break;

        }

    }

    arr[i] = temp;//将temp值放到最终的位置

}

void Heapsort(int arr[], int length)

{

    //1.构建大顶堆

    for (int i = length / 2 - 1; i >= 0; i--)

    {

        //从第一个非叶子结点从下至上，从右至左调整结构

        adjustHeap(arr, i, length);

    }

    for (int j = length - 1; j>0; j--)

    {

        swap(arr, 0, j);//将堆顶元素与末尾元素进行交换

        adjustHeap(arr, 0, j);//重新对堆进行调整

    }

}

//------------------堆排序

//----------归并排序

void merge(int\* a, int low, int mid, int hight)  //合并函数

{

    int\* b = new int[hight - low + 1];  //用 new 申请一个辅助函数

    int i = low, j = mid + 1, k = 0;    // k为 b 数组的小标

    while (i <= mid && j <= hight)

    {

        if (a[i] <= a[j])

        {

            b[k++] = a[i++];  //按从小到大存放在 b 数组里面

        }

        else

        {

            b[k++] = a[j++];

        }

    }

    while (i <= mid)  // j 序列结束，将剩余的 i 序列补充在 b 数组中

    {

        b[k++] = a[i++];

    }

    while (j <= hight)// i 序列结束，将剩余的 j 序列补充在 b 数组中

    {

        b[k++] = a[j++];

    }

    k = 0;  //从小标为 0 开始传送

    for (int i = low; i <= hight; i++)  //将 b 数组的值传递给数组 a

    {

        a[i] = b[k++];

    }

    delete[]b;     // 辅助数组用完后，将其的空间进行释放（销毁）

}

void mergesort(int\* a, int low, int hight) //归并排序

{

    if (low < hight)

    {

        int mid = (low + hight) / 2;

        mergesort(a, low, mid);          //对 a[low,mid]进行排序

        mergesort(a, mid + 1, hight);    //对 a[mid+1,hight]进行排序

        merge(a, low, mid, hight);       //进行合并操作

    }

}

//----------归并排序

int main()

{

    int numbersnum=0;

    cout<<"排序数量：";

    cin>>numbersnum;

    int iRandNum[RANDNUM]; // 存放随机数

    clock\_t first, second; // 记录开始和结束时间（以毫秒为单位）

    double t;

    int temp, a[RANDNUM], b[RANDNUM];

    int i, j, index;

    for (i = 0; i < RANDNUM; i++) // 产生1万个随机数

    {

        iRandNum[i] = rand() % 10000;

    }

    for (i = 0; i < RANDNUM; i++)

    {

        a[RANDNUM] = iRandNum[RANDNUM];

        b[RANDNUM] = iRandNum[RANDNUM];

    }

    printf("初始序列前%d个数为:\n",numbersnum);

    for (i = 0; i < numbersnum; i++)

    {

        printf("%d  ", iRandNum[i]);

        int c=sqrt(numbersnum);

        if (i %c == 0 && i > 0)

            printf("\n");

    }

    printf("\n");

    first = clock(); // 开始时间

    // 此处加入排序程序

    i = 0;

    while (i < numbersnum)

    {

        QuickSort(iRandNum, 0, RANDNUM - 1); // 快速排序

        i++;

    }

    second = clock(); // 结束时间

                      // 显示排序算法所用的时间

    t = (double)(second - first) / CLK\_TCK;

    printf("快速排序%lf seconds", t);

    printf("\n");

    first = clock();

    i = 0;

    while (i < numbersnum)

    {

        shellSort(a, RANDNUM);

        i++;

    }

    second = clock();

    t = (double)(second - first) / CLK\_TCK;

    printf("希尔排序 %lf seconds", t);

    printf("\n");

    first = clock();

    for (i = 0; i < RANDNUM - 1; i++)//----numbersnum

    {

        index = i;

        for (j = i + 1; j < RANDNUM; j++)//-----numbersnum

        {

            if (b[j] < b[index])

                index = j;

        }

        temp = b[index];

        b[index] = b[i];

        b[i] = temp;

    }

    second = clock();

    t = (double)(second - first) / CLK\_TCK;

    printf("选择排序 %lf seconds", t);

    printf("\n");

    //-----------------插入排序

    first = clock(); // 开始时间

    // 此处加入排序程序

    i = 0;

    while (i < numbersnum)

    {

        InsertSort(iRandNum, RANDNUM ); // 插入排序

        i++;

    }

    second = clock(); // 结束时间

                      // 显示排序算法所用的时间

    t = (double)(second - first) / CLK\_TCK;

    printf("插入排序%lf seconds", t);

    printf("\n");

    //-----------------堆排序

    first = clock(); // 开始时间

    // 此处加入排序程序

    i = 0;

    while (i < numbersnum)

    {

        Heapsort(iRandNum,RANDNUM ); // 堆排序

        i++;

    }

    second = clock(); // 结束时间

                      // 显示排序算法所用的时间

    t = (double)(second - first) / CLK\_TCK;

    printf("堆排序%lf seconds", t);

    printf("\n");

    //-----------------归并排序

    first = clock(); // 开始时间

    // 此处加入排序程序

    i = 0;

    while (i < numbersnum)

    {

        mergesort(iRandNum,0,RANDNUM-1 ); // 归并排序

        i++;

    }

    second = clock(); // 结束时间

                      // 显示排序算法所用的时间

    t = (double)(second - first) / CLK\_TCK;

    printf("归并排序%lf seconds", t);

    printf("\n");

}

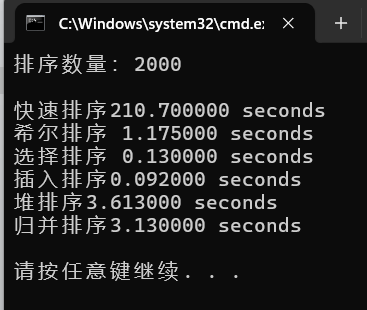
**排序20个数各个算法所需时间输出结果：**

****

**排序200个数各个算法所需时间输出结果：**

****

**排序2000个数各个算法所需时间输出结果：（由于数据量太大，只截了最终比较结果，所排的2000个数注释掉并未输出）**

****

**六、实验总结与体会：**

这次实验在网上搜索了如上几个排序算法，并将他们集中比较呈现出最终结果。

从此次输出结果中看出，快速排序最慢，选择排序较快。

通过这次实训，我收获了很多，一方面学习到了许多以前没学过的专业知识与知识的应用，另一方面还提高了自己动手做项目的能力。本次实训，是对我能力的进一步锻炼，也是一种考验。从中获得的诸多收获，也是很可贵的，是非常有意义的。

但是在学习中，仍然存在一些难以解决的问题，只能通过借鉴别人的代码学习。

在实训中我学到了许多新的知识。是一个让我把书本上的理论知识运用于实践中的好机会，原来，学的时候感叹学的内容太难懂，现在想来，有些其实并不难，关键在于理解。