**实验一 线性结构**

**1、采用链式结构存储一元多项式并对其求导**

#include<iostream>

using namespace std;

typedef struct PolyNode \* PtrToPolyNode;

struct PolyNode{

int Coef; //系数

int Expon; //指数

PtrToPolyNode Next;

};

typedef PtrToPolyNode Polynomial;

Polynomial CreatP() //创建链表表示多项式

{

Polynomial P, Pfront, temp;

P = (Polynomial)malloc(sizeof(struct PolyNode));

P->Next = NULL;

Pfront = P; //记录多项式链表头结点

for(int i=0;i<4;i++) //假定输入的多项式有4项

{

temp = (Polynomial)malloc(sizeof(struct PolyNode));

cin>>temp->Coef>>temp->Expon;

temp->Next = NULL;

P->Next = temp;

P = P->Next;

}

return Pfront;

}

void derivative(Polynomial P) //多项式求导

{

while(P->Next)

{

P = P->Next;

if(P->Expon > 0)

{

P->Coef = P->Coef \* P->Expon;

P->Expon = P->Expon - 1;

cout<<P->Coef<<" "<<P->Expon<<endl;

}

}

}

int main()

{

cout<<"输入样例："<<endl;

Polynomial P = CreatP();//创建链表表示多项式

cout<<"输出样例："<<endl;

derivative(P); //求导并输出

return 0;

}

/\*

输入样例：

3 4

-5 2

6 1

-2 0

输出样例：

12 3

-10 1

6 0

\*/

**2、将两个升序链表合并为一个升序链表**

#include<iostream>

using namespace std;

typedef struct Node \* PtrToNode; //定义结构体

struct Node{

int Data; //数据

PtrToNode Next;

};

typedef PtrToNode List;

List CreatL() **//创建新链表**

{

List L, Lfront, temp;

L = (List)malloc(sizeof(struct Node));

L->Next = NULL;

Lfront = L; //记录链表头结点

while(true) //创建任意长度的链表

{

temp = (List)malloc(sizeof(struct Node));

cin>>temp->Data;

temp->Next = NULL;

L->Next = temp;

L = L->Next;

if(getchar()!=' ') //当按回车键（不再接空格）时，链表创建结束并返回

return Lfront;

}

}

void AddL(List L1, List L2) **//合并链表**

{

List L3 = L1; //结果链表在L1基础上改造

int d1, d2;

L1 = L1->Next;

L2 = L2->Next;

while(L1&&L2)

{

if(L1->Data >= L2->Data) // L2的值偏小

{

L3->Next = L2;

L3 = L3->Next;

L2 = L2->Next;

}

else // L1的值偏小

{

L3->Next = L1;

L3 = L3->Next;

L1 = L1->Next;

}

}

if(L1)

L3->Next = L1;

else

L3->Next = L2;

}

int main()

{

cout<<"输入样例："<<endl;

List L1 = CreatL(); //创建链表L1

List L2 = CreatL(); //创建链表L2

List Lfront = L1; //记录L1头结点

cout<<"输出样例："<<endl;

AddL(L1,L2); //合并L1、L2到L1

while(Lfront->Next) //输出合并的升序链表

{

Lfront = Lfront->Next;

cout<<Lfront->Data<<" ";

}

return 0;

}

/\*

输入：1 4 5

1 3 6

输出：1 1 3 4 5 6

\*/

**3、计算中缀表达式的结果（栈结构表示）**

**先将中缀表达式转化为后缀表达式，再运用新栈进行单次计算**

#include<iostream>

#include<stack>

#include<string>

using namespace std;

struct Node{

double num; //操作数

char op; //操作符

};

typedef struct Node expnode;

stack<expnode>s;

int ex\_c(char c) //将操作符转换成对应的优先级

{

int m;

switch(c)

{

case '+': m = 1; break;

case '-': m = 1; break;

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

case '/': m = 2; break;

default: break;

}

return m;

}

double answer\_count(char c, double n1, double n2) //操作符出栈时就进行“单次”运算

{

double answer0;

switch(c)

{

case '+': answer0 = n1+n2; break;

case '-': answer0 = n1-n2; break;

case '\*': answer0 = n1\*n2; break;

case '/': answer0 = n1/n2; break;

default: break;

}

return answer0;

}

void Expression\_Calculate(string str) //**将中缀表达式转化为后缀表达式**，不输出，直接计算{（数字入栈，符号判断）

stack<double>numS; //存后缀表达式中的操作数的栈

expnode temp; //每次存放表达式中的一个字符

double k; //用于处理小数的0.1,0.01

double n1,n2; //记录每次参加运算的两个数

double answer; //记录每次运算的结果

temp.op = '#';

s.push(temp); //栈底元素为#

for(int i=0;i<str.length();) //注意不用自动加1

{

if(str[i]>='0' && str[i]<='9') //判断是否为数字

{

temp.num = str[i] - '0'; //将字符转换成数字储存

i++;

while(i<str.length() && str[i]>='0'&&str[i]<='9') //整数部分出超过一位数字

{

temp.num = temp.num\*10 + (str[i] - '0');

i++;

}

if(str[i]=='.')

{

i++;

k=0.1;

while(i<str.length() && str[i]>='0'&&str[i]<='9')

{

temp.num = temp.num + (str[i] - '0')\*k;

i++;

k=k\*0.1;

}

}

numS.push(temp.num);//操作数按后缀顺序入栈

}

else if(str[i]=='(') //判断是否为左括号

{

temp.op = str[i];

s.push(temp);

i++;

}

else if(str[i]==')') //判断是否为右括号

{

while(!s.empty() && s.top().op!='(')

{

n2 = numS.top(); //取操作数栈栈顶元素

numS.pop(); //取出后就出栈

n1 = numS.top(); //再取一个栈顶元素

numS.pop(); //取出后就出栈

answer = answer\_count(s.top().op, n1, n2); //调用单次运算函数计算

numS.push(answer); //计算得到的新的值入操作数的栈

s.pop(); //已经运算的操作符出栈

}

s.pop(); //弹出左括号

i++;

}

else if(str[i]=='=') //判断是否为等号

{

break;

}

else //判断是否为运算符

{

while(!s.empty()&&s.top().op!='(' && s.top().op!='#'&&ex\_c(str[i])<=ex\_c(s.top().op))

{

n2 = numS.top();

numS.pop();

n1 = numS.top();

numS.pop();

answer = answer\_count(s.top().op, n1, n2); //调用函数计算

numS.push(answer); //计算得到的新的值入操作数的栈

s.pop();

}

temp.op = str[i];

s.push(temp);

i++;

}

}

while(!s.empty()) //如果符号栈不为空

{

if(s.top().op!='#')

{

n2 = numS.top();

numS.pop();

n1 = numS.top();

numS.pop();

answer = answer\_count(s.top().op, n1, n2); //调用函数计算

numS.push(answer); //计算得到的新的值入操作数的栈

s.pop();

}

else

{

s.pop(); //当只剩栈底元素“#”时，将其出栈

}

}

printf("%.2f",numS.top()); //输出最终结果，即操作数栈中的最后一个值

}

int main()

{

string str\_M; //中缀表达式串儿

cout<<"请输入一个中缀表达式："<<endl;

cin>>str\_M;

cout<<"表达式计算的结果为："<<endl;

Expression\_Calculate(str\_M);//表达式计算

return 0;

}

/\*

输入样例：20\*(4.5–3)=

输出结果：30.00

\*/

**4、输入队列的元素个数，以及队列中的元素，将队列中元素从小到大排列**

#include<iostream>

#include<queue>

#include<string>

using namespace std;

queue<int>q;

void sort\_q(queue<int>q, int n) //排序

{

int a,b,c,t;

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

{

a=q.front();

q.pop();

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

{

b=q.front();

q.pop();

if(a<b)

{

q.push(a);

a=b;

}

else

q.push(b);

}

q.push(a);

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

{

c=q.front();

q.pop();

q.push(c);

}

}

for(int i=0; i<n; i++) //输出

{

t=q.front();

q.pop();

cout<<t<<" ";

}

}

int main()

{

int n;

int k;

cout<<"输入样例："<<endl;

cin>>n;

for(int i=0; i<n; i++) //输入

{

cin>>k;

q.push(k);

}

cout<<"输出样例："<<endl;

sort\_q(q,n); //排序与输出

return 0;

}

/\*

输入样例：10

9 4 6 1 8 3 7 0 2 5

输出样例：

0 1 2 3 4 5 6 7 8 9

\*/

**实验二 树**

**1、判断两棵二叉树是否同构**

#include<iostream>

#include <string.h>

using namespace std;

struct TNode{

char data; //数据类型为字符

int left; //左孩子对应编号

int right; //右孩子对应编号

};

TNode BT1[50],BT2[50]; //定义输入的两棵二叉树

int flag = 0; //判断两棵树是否同构，=0目前同构，=1不同构

//根据左右孩子结点的编号找到左右结点的data，并传回

//将无孩子结点的-1都改为字符‘0’，更方便直接比较

char getBT1\_childData(int num){

if(num != -1)

return BT1[num].data;

else

return '0';

}

char getBT2\_childData(int num){

if(num != -1)

return BT2[num].data;

else

return '0';

}

//比较两棵树相同的结点的左右孩子结点的data是否相等

void compare(char L1, char R1, char L2, char R2){

if(L1 == L2) //1左=2左

{

if(R1 != R2) //1右!=2右

flag = 1;

}

else if(L1 == R2) //1左=2右

{

if(R1 != L2) //1右!=2左

flag = 1;

}

else //1左和2的左右都不等

flag = 1;

}

int main()

{

int N1,N2; //分别为两棵树的节点数

cout<<"输入样例："<<endl;

cin>>N1; //输入BT1

for(int i=0;i<N1;i++)

cin>>BT1[i].data>>BT1[i].left>>BT1[i].right;

cin>>N2; //输入BT2

for(int i=0;i<N2;i++)

cin>>BT2[i].data>>BT2[i].left>>BT2[i].right;

if(N1!=N2) flag = 1;//结点数不等肯定不同构

if(flag == 0) //如果结点数相同，继续判断

{

for(int i=0;i<N1;i++)

{

int j=0;

for(j=0;j<N2;j++)

{

if(BT1[i].data == BT2[j].data) //找到了BT2中与BT1[i]对应的结点data

{

//比较它们俩的左右孩子结点的data

compare(getBT1\_childData(BT1[i].left), getBT1\_childData(BT1[i].right), getBT2\_childData(BT2[j].left) , getBT2\_childData(BT2[j].right));

break;

}

}

if(j==N2 || flag){

//如果BT2中没有BT1[i]对应的那个结点，或者已经检测出flag为1了，就跳出循环

flag = 1;break;

}

}

}

cout<<"输出样例："<<endl;

if(flag)

cout<<"False"<<endl;

else

cout<<"True"<<endl;

return 0;

}

/\*

输入样例

8

A 1 2

B 3 4

C 5 -1

D -1 -1

E 6 -1

G 7 -1

F -1 -1

H -1 -1

8

G -1 4

B 7 6

F -1 -1

A 5 1

H -1 -1

C 0 -1

D -1 -1

E 2 -1

\*/

**2、将二叉搜索树中序遍历，形成递增序列，并生成新树（无左结点）**

#include<iostream>

#include<queue>

using namespace std;

typedef struct TNode \*Position;

typedef Position BinTree;

struct TNode{

int Data; //假设数据类型为整型

BinTree Left;

BinTree Right;

};

int TreeMdata[50]; //记录中序遍历得到的新树的每个结点的data值（空为-1）；

int j=0; //统计TreeMdata中有多少元素

//层序遍历，**创建二叉搜索树**/生成新的递增顺序搜索树

BinTree CreatBinTree(int Treedata[])

{

int Data;

int i=0;

BinTree BT, T;

queue<BinTree>Q;

//建立第一个结点，即根结点

if(Treedata[i]==-1)i++; //跳过中序遍历得到的第一个空-1

Data = Treedata[i];

i++;

if(Data != -1)

{

//分配根节点单元，并将结点地址入队

BT = (BinTree)malloc(sizeof(struct TNode));

BT->Data = Data;

BT->Left = BT->Right = NULL;

Q.push(BT);

}

else return NULL; //否则返回树为空

while(!Q.empty()&&Treedata[i])

{

T = Q.front();

Q.pop();

Data = Treedata[i]; //读入T的左孩子

i++;

if(Data == -1)

T->Left = NULL;

else

{

//分配新结点，作为出队结点的左孩子；再将新结点入队

T->Left = (BinTree)malloc(sizeof(struct TNode));

T->Left->Data = Data;

T->Left->Left = T->Left->Right = NULL;

Q.push(T->Left);

}

Data = Treedata[i]; //读入T的右孩子

i++;

if(Data == -1)

T->Right = NULL;

else

{

//分配新结点，作为出队结点的右孩子；再将新结点入队

T->Right = (BinTree)malloc(sizeof(struct TNode));

T->Right->Data = Data;

T->Right->Left = T->Right->Right = NULL;

Q.push(T->Right);

}

}

return BT;

}

/**/中序遍历二叉搜索树**，并将data存进数组Treedata[50]中

void InorderTraversal(BinTree BT){

if(BT){

InorderTraversal(BT->Left);

TreeMdata[j] = BT->Data;

j++;

InorderTraversal(BT->Right);

}

else{

TreeMdata[j] = -1;

j++;

}

}

int main()

{

cout<<"输入样例："<<endl;

int TreeNode[50]; //记录输入的二叉搜索树每个结点的data值（空为-1）；

int i=0;

while(true)

{

cin>>TreeNode[i];

i++;

if(getchar()!=' ') //当按回车键（不再是空格）时，输入结束

break;

}

BinTree BST; //二叉搜索树

BinTree BSTM; //中序遍历得到的新的递增序列搜索树

BST = CreatBinTree(TreeNode); //创建输入的二叉搜索树

InorderTraversal(BST); //中序遍历，并生成新的递增顺序搜索树

cout<<"输出样例："<<endl;

TreeMdata[j-1]=0; //跳过中序遍历得到的最后一个空-1

BSTM = CreatBinTree(TreeMdata); //创建新的递增排序搜索树

for(i=1;i<j-1;i++)

cout<<TreeMdata[i]<<" ";

return 0;

}

/\*

输入样例：5 3 6 2 4 -1 8 1 -1 -1 -1 7 9

输出样例：1 null 2 null 3 null 4 null 5 null 6 null 7 null 8 null 9

注：样例里面的null在实验代码中均由“-1”代替

\*/

**3、判断一颗二叉树是否为 二叉搜索树**

#include<iostream>

#include<queue>

using namespace std;

typedef struct TNode \*Position;

typedef Position BinTree;

struct TNode{

int Data; //假设数据类型为整型

BinTree Left;

BinTree Right;

};

int TreeMdata[50]; //记录中序遍历得到的每个结点的data值（空为-1）；

int j=0; //统计TreeMdata中有多少元素

//层序遍历，**创建**二叉树

BinTree CreatBinTree(int Treedata[])

{

int Data;

int i=0;

BinTree BT, T;

queue<BinTree>Q;

//建立第一个结点，即根结点

if(Treedata[i]==-1) i++; //跳过中序遍历得到的第一个空-1

Data = Treedata[i];

i++;

if(Data != -1)

{ //分配根节点单元，并将结点地址入队

BT = (BinTree)malloc(sizeof(struct TNode));

BT->Data = Data;

BT->Left = BT->Right = NULL;

Q.push(BT);

}

else return NULL; //否则返回树为空

while(!Q.empty()&&Treedata[i])

{

T = Q.front();

Q.pop();

Data = Treedata[i]; //读入T的左孩子

i++;

if(Data == -1)

T->Left = NULL;

else{ //分配新结点，作为出队结点的左孩子；再将新结点入队

T->Left = (BinTree)malloc(sizeof(struct TNode));

T->Left->Data = Data;

T->Left->Left = T->Left->Right = NULL;

Q.push(T->Left);

}

Data = Treedata[i]; //读入T的右孩子

i++;

if(Data == -1)

T->Right = NULL;

else{ //分配新结点，作为出队结点的右孩子；再将新结点入队

T->Right = (BinTree)malloc(sizeof(struct TNode));

T->Right->Data = Data;

T->Right->Left = T->Right->Right = NULL;

Q.push(T->Right);

}

}

return BT;

}

//**中序遍历二叉树**，并将data存进数组Treedata[50]中

void InorderTraversal(BinTree BT)

{

if(BT){

InorderTraversal(BT->Left);

TreeMdata[j] = BT->Data;

j++;

InorderTraversal(BT->Right);

}

}

//**判断**得到的TreeMdata[]中的值**是否为升序序列**，由此判断是否为二叉搜索树，

void IsBST(int Treedata[])

{

int k=0;

while(Treedata[k+1])

{

if(Treedata[k] >= Treedata[k+1]) //不是升序

{

cout<<"False"<<endl;

break;

}

k++;

}

if(k+1 == j) //全部是升序

cout<<"True"<<endl;

}

int main()

{

cout<<"输入样例："<<endl;

int TreeNode[50]; //记录输入的二叉树每个结点的data值（空为-1）；

int i=0;

while(true)

{

cin>>TreeNode[i];

i++;

if(getchar()!=' ')//当按回车键（不再是空格）时，输入结束

break;

}

cout<<"输出样例："<<endl;

BinTree BT; //二叉树

BT = CreatBinTree(TreeNode); //创建输入的二叉树

InorderTraversal(BT); //中序遍历，得到遍历结果，存入数组TreeMdata[j]

IsBST(TreeMdata); //判断中序遍历结果是否为升序

return 0;

}

//2 1 3 true

//5 1 4 -1 -1 3 6 false

**4、寻找任意两个结点最近的祖先节点，并输出**

#include<iostream>

#include <math.h>

using namespace std;

struct Node{

int data; //数据类型为整型

int num; //节点编号

int floor; //记录数据所在树的层数

};

Node BT[50]; //定义输入的二叉树

int i=0; //树的节点编号

void Ancestors(Node BT[], int a, int b) //在BT中寻找a,b的(最近的)公共祖先

{

Node x,y;

x.data = a;

y.data = b;

for(int j=0;j<i;j++) //先找到a,b在树中的位置

{

if(BT[j].data == x.data)

{

x.floor = BT[j].floor;

x.num = j+1;

}

if(BT[j].data == y.data)

{

y.floor = BT[j].floor;

y.num = j+1;

}

}

while(x.data != y.data) //再通过自下而上比较父节点的方式寻找最近的祖先节点

{

if(x.floor > y.floor)

{

x.floor = x.floor - 1;

x.num = x.num/2;

x.data = BT[x.num-1].data;

}

else if(x.floor < y.floor)

{

y.floor = y.floor - 1;

y.num = y.num/2;

y.data = BT[y.num-1].data;

}

else

{

x.floor = x.floor - 1;

y.floor = y.floor - 1;

x.num = x.num/2;

y.num = y.num/2;

x.data = BT[x.num-1].data;

y.data = BT[y.num-1].data;

}

}

cout<<x.data<<endl;

}

int main()

{

cout<<"输入样例："<<endl;

while(true)

{

cin>>BT[i].data; //数据

BT[i].num = i+1; //编号

BT[i].floor = int(log(BT[i].num)/log(2))+1; //层数

i++;

if(getchar()!=' ') //当按回车键（不再是空格）时，输入结束

break;

}

int a,b; //记录两个子节点数据

cin>>a; cin>>b;

cout<<"输出样例："<<endl;

Ancestors(BT, a, b); //在BT中寻找a,b的(最近的)公共祖先

return 0;

}

/\*

3 5 1 6 2 0 8 -1 -1 7 4

5

1

输出：3

3 5 1 6 2 0 8 -1 -1 7 4

5

4

输出：5

\*/

**实验三 图**

**1、创建并输出带权图的邻接矩阵**

#include<iostream>

using namespace std;

#define MaxVertexNum 100 //最大顶点数为100

#define Infinity 0 //邻接矩阵中无边用0代替

//**图结点的定义**

typedef struct GNode \*PtrToGNode;

struct GNode{

int Nv; //顶点数

int Ne; //边数

int G[MaxVertexNum][MaxVertexNum]; //邻接矩阵

};

typedef PtrToGNode MGraph; //以邻接矩阵的方式存储的图类型

//**边的定义**

typedef struct ENode \*PtrToENode;

struct ENode{

int v1,v2; //边的顶点

int w; //权重

};

typedef PtrToENode Edge;

//**初始化** 一个有VertexNum个顶点但没有边的图

MGraph CreateGraph(int VertexNum){

MGraph Graph;

Graph = (MGraph)malloc(sizeof(struct GNode));

Graph->Nv = VertexNum;

Graph->Ne = 0;

//初始化邻接矩阵，顶点编号从0开始到Nv-1

for(int i=0; i<Graph->Nv; i++)

for(int j=0; j<Graph->Nv; j++)

Graph->G[i][j] = Infinity;//初始化矩阵为0

return Graph;

}

//**插入边**

void InsertEdge(MGraph Graph, Edge E){

//因为是有向图，一次只需插入一条边

Graph->G[E->v1][E->v2] = E->w;

}

//**构建图**

MGraph BuildGraph(){

int Nv;

cin>>Nv;

MGraph Graph;

Graph = CreateGraph(Nv);

Edge E;

E = (Edge)malloc(sizeof(struct ENode));

for(int i=0; i<10; i++) //10条边

{

//读入边，顺序为起点、终点、权重，插入邻接矩阵

cin>>E->v1>>E->v2>>E->w;

InsertEdge(Graph, E); //插入边

Graph->Ne++;

}

return Graph;

}

int main()

{

cout<<"输入样例："<<endl;

MGraph Graph;

Graph = BuildGraph();

cout<<"输出样例："<<endl;

for(int i=0; i<Graph->Nv; i++) //输出邻接矩阵

{

for(int j=0; j<Graph->Nv; j++)

cout<<Graph->G[i][j]<<" ";

cout<<endl;

}

return 0;

}

/\*

输入样例：

6

0 1 5

0 3 7

1 2 4

2 0 8

2 5 9

3 2 5

3 5 6

4 3 5

5 0 3

5 4 1

\*/

**2、利用邻接表存储有向图，并对图进行深度优先搜索**

#include<iostream>

using namespace std;

#define MaxVertexNum 100 //最大顶点数为100

bool Visited[MaxVertexNum] = {false}; //记录该顶点是否已经访问过了

**//边的定义**

typedef struct ENode \*PtrToENode;

struct ENode{

int v1,v2; //边的顶点，无权重

};

typedef PtrToENode Edge;

**//邻接点的定义**

typedef struct AdjVNode \*PtrToAdjVNode;

struct AdjVNode{

int AdjV; //邻接点下标，无边权重

PtrToAdjVNode Next; //指向下一个邻接点的指针

};

**//顶点表头结点的定义**

typedef struct Vnode{

PtrToAdjVNode FirstEdge; //边表头指针

char Data; //存顶点的数据

}AdjList[MaxVertexNum];

**//图结点的定义**

typedef struct GNode \*PtrToGNode;

struct GNode{

int Nv; //顶点数

int Ne; //边数

AdjList G; //邻接表

};

typedef PtrToGNode LGraph; //以邻接表的方式存储的图类型

//**初始化**一个有VertexNum个顶点但没有边的图

LGraph CreateGraph(int VertexNum)

{

LGraph Graph;

Graph = (LGraph)malloc(sizeof(struct GNode));

Graph->Nv = VertexNum;

Graph->Ne = 0;

//初始化邻接表头指针，顶点编号从0开始到Nv-1

for(int i=0; i<Graph->Nv; i++)

Graph->G[i].FirstEdge = NULL;

return Graph;

}

/**/插入边**

void InsertEdge(LGraph Graph, Edge E)

{

PtrToAdjVNode NewNode;

//插入<v1,v2>，为v2建立新的邻接点

NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjVNode));

NewNode->AdjV = E->v2;

//将v2插入v1的表头

NewNode->Next = Graph->G[E->v1].FirstEdge;

Graph->G[E->v1].FirstEdge = NewNode;

}

//**构建图**

LGraph BuildGraph()

{

int Nv;

cin>>Nv;

//创建空图

LGraph Graph;

Graph = CreateGraph(Nv);

//读入顶点的数据（char）

for(int i=0; i<Graph->Nv; i++)

cin>>Graph->G[i].Data;

//创建边

Edge E;

E = (Edge)malloc(sizeof(struct ENode));

char Vdata[2];

for(int i=0; i<7; i++) //7条边

{

//读入边，顺序为起点、终点，无权重，插入邻接表

cin>>Vdata[0]>>Vdata[1];

for(int j=0; j<Graph->Nv; j++) //找到起点终点data对应的顶点下标

{

if(Graph->G[j].Data == Vdata[0])

E->v1 = j;

if(Graph->G[j].Data == Vdata[1])

E->v2 = j;

}

InsertEdge(Graph, E); //插入边

Graph->Ne++;

}

return Graph;

}

//以第v个顶点为出发点，对领接表存储的图Graph进行**深度优先搜索**

void DFS(LGraph Graph, int v)

{

PtrToAdjVNode f;

Visited[v] = true; //标记顶点v已经访问过了为TRUE

cout<<Graph->G[v].Data<<" "; //输出正在访问下标为v的顶点的data

for(f = Graph->G[v].FirstEdge; f; f = f->Next) //对于v的每一个邻接点f->AdjV

if(!Visited[f->AdjV]) //如果f指向的结点还没有被访问过

DFS(Graph, f->AdjV); //则递归访问它

}

int main()

{

cout<<"输入样例："<<endl;

LGraph Graph;

Graph = BuildGraph();

cout<<"输出样例："<<endl;

DFS(Graph, 0); //深度优先搜索，选择从下标为0的顶点开始

return 0;

}

/\*

输入样例：

6

a b c d e f

a b

a c

a e

b e

e d

d f

e f

\*/

**3、给一个社交网络图，对每个节点计算符合“六度空间”理论的结点占结点总数的百分比**

//总体思路是：采用 邻接表存储图 + 广度优先搜索 + 队列的实现过程

#include<iostream>

#include<queue>

using namespace std;

#define MaxVertexNum 100//最大顶点数为100

bool Visited[MaxVertexNum] = {false};//记录该顶点是否已经访问过了

**//边的定义**

typedef struct ENode \*PtrToENode;

struct ENode{

int v1,v2; //边的顶点，无权重

};

typedef PtrToENode Edge;

**//邻接点的定义**

typedef struct AdjVNode \*PtrToAdjVNode;

struct AdjVNode{

int AdjV; //邻接点下标，无边权重

PtrToAdjVNode Next; //指向下一个邻接点的指针

};

**//顶点表头结点的定义**

typedef struct Vnode{

PtrToAdjVNode FirstEdge; //边表头指针，无顶点数据

}AdjList[MaxVertexNum];

**//图结点的定义**

typedef struct GNode \*PtrToGNode;

struct GNode{

int Nv; //顶点数

int Ne; //边数

AdjList G; //邻接表

};

typedef PtrToGNode LGraph; //以邻接表的方式存储的图类型

**//初始化**一个有VertexNum个顶点但没有边的图

LGraph CreateGraph(int VertexNum)

{

LGraph Graph;

Graph = (LGraph)malloc(sizeof(struct GNode));

Graph->Nv = VertexNum;

Graph->Ne = 0;

//初始化邻接表头指针，顶点编号从0开始到Nv-1

for(int i=0; i<Graph->Nv; i++)

Graph->G[i].FirstEdge = NULL;

return Graph;

}

**//插入边**（注意：社交网络图为无向图）

void InsertEdge(LGraph Graph, Edge E)

{

PtrToAdjVNode NewNode;

//插入<v1,v2>，为v2建立新的邻接点

NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjVNode));

NewNode->AdjV = E->v2;

//将v2插入v1的表头

NewNode->Next = Graph->G[E->v1].FirstEdge;

Graph->G[E->v1].FirstEdge = NewNode;

//插入<v2,v1>，为v1建立新的邻接点

NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjVNode));

NewNode->AdjV = E->v1;

//将v1插入v2的表头

NewNode->Next = Graph->G[E->v2].FirstEdge;

Graph->G[E->v2].FirstEdge = NewNode;

}

**//构建图**

LGraph BuildGraph()

{

int Nv,Ne,Vdata[2];

cin>>Nv>>Ne;

//创建空图

LGraph Graph;

Graph = CreateGraph(Nv);

//创建边

Edge E;

E = (Edge)malloc(sizeof(struct ENode));

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

{

//读入边，顺序为起点、终点，无权重，插入邻接表

cin>>Vdata[0]>>Vdata[1]; //把结点编号转化为下标，-1即可

E->v1 = Vdata[0]-1;

E->v2 = Vdata[1]-1;

InsertEdge(Graph, E);//插入边

}

return Graph;

}

//以S为出发点，对图Graph进行6层的**BFS广度优先搜索**

int SDS\_BFS(LGraph Graph, int S)

{

queue<int>Q;

int V,Last,Tail;

PtrToAdjVNode F; //定义一个指向邻接表结点的指针

int Count, Level;

Visited[S] = true; //标记顶点v已经访问过了为TRUE

Count = 1; //统计符合“六度空间”理论的人数，从1开始

Level = 0; //起始点定义为第0层

Last = S; //该层只有S一个顶点，是该层被访问的最后一个顶点

Q.push(S); //将S入队列

while(!Q.empty())

{

V = Q.front();

Q.pop();

for(F = Graph->G[V].FirstEdge; F; F = F->Next)//对于V的每一个邻接点F->AdjV

{

if(!Visited[F->AdjV]) //如果F指向的结点还没有被访问

{

Visited[F->AdjV] = true; //标记F->AdjV已被访问

Count++; //人数加1

Tail = F->AdjV; //改变层尾

Q.push(F->AdjV); //将F->AdjV入队列

}

}

if(V == Last) //如果上一层的最后一个顶点弹出了

{

Level++; //层数递增

Last = Tail; //更新当前层尾为该层被访问的最后一个顶点

}

if(Level == 6) break; //如果6层遍历结束，退出搜索

}

if(!Q.empty()) Q.pop(); //释放队列所有元素

return Count; //返回统计的人数

}

//一个顶点的六度空间搜索完毕以后，要重新**初始化某些顶点的Visited[]值**

void InitializeVisited(int Nv)

{

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

Visited[i] = false;

}

//**用邻接表存储图，对每个顶点检验六度空间理论**

void Six\_Degrees\_of\_Separation(LGraph Graph)

{

int Count;

for(int i=0; i<Graph->Nv; i++) //对图中的每个顶点都检验一遍六度空间理论

{

InitializeVisited(Graph->Nv); //初始化Visited[]值

Count = SDS\_BFS(Graph, i); //BFS广度优先搜索，检验六度空间

printf("%d：%.2f%%\n", i+1, 100.0\*(double)Count/(double)Graph->Nv);

}

}

int main()

{

cout<<"输入样例："<<endl;

LGraph Graph; //用邻接表存储图

Graph = BuildGraph();

cout<<"输出样例："<<endl;

Six\_Degrees\_of\_Separation(Graph); //对图的每个顶点检验六度空间理论

return 0;

}

/\*

输入样例

10 9

1 2

2 3

3 4

4 5

5 6

6 7

7 8

8 9

9 10

\*/

**实验四 排序**

**1、利用四种排序方法进行升序排序**

#include<iostream>

using namespace std;

void PercDown(int A[], int p, int N) //将N个元素的数组中，以A[p]为根的子堆调整为最大堆

{

int Parent, Child;

int X;

X = A[p]; //取出根结点存放的值

for(Parent = p; (2\*Parent+1)<N; Parent = Child)

{

Child = 2\*Parent+1;

if( (Child!=N-1) && (A[Child]<A[Child+1]) )

Child++; //Child指向左右子结点中的较大者

if(X >= A[Child]) break; //即找到了合适的位置

else

A[Parent] = A[Child]; //下滤

}

A[Parent] = X;

}

void Swap(int \*a, int \*b) //交换两个元素的函数

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void HeapSort(int A[], int N) **//堆排序**

{

int i;

for(i=N/2-1; i>=0; i--) //建立最大堆

PercDown(A, i, N);

for(i=N-1; i>0; i--)

{

Swap(&A[0], &A[i]); //将根结点与最后一个结点交换

PercDown(A, 0, i); //将新的堆重新调整为最大堆

}

}

void BubbleSort(int A[], int N) **//冒泡排序**

{

int P, i;

bool flag;

for(P=N-1; P>=0; P--)

{

flag = false; //标记该次循环中是否发生交换，若无，则说明整个序列有序

for(i=0; i<P; i++) //此为一趟冒泡

{ //每次循环找出一个最大元素，被交换到最右端

if(A[i]>A[i+1])

{

Swap(&A[i], &A[i+1]);

flag = true; //标记发生了交换

}

}

if(flag == false) break; //若全程无交换，则跳出循环

}

}

void InsertionSort(int A[], int N) **//直接插入排序**

{

int P, i;

int temp;

for(P=1; P<N; P++)

{

temp = A[P]; //取出未排序元素中的第一个元素

for(i=P; i>0&&A[i-1]>temp; i--)

A[i] = A[i-1]; //依次与已排序序列中的元素比较并右移

A[i] = temp; //找到了合适的位置，放进去

}

}

void ShellSort(int A[], int N) **//希尔排序**

{

int Si, D, P, i;

int temp;

int Sedgewick[] = {5, 3, 1, 0}; //列出一部分增量

for(Si=0; Sedgewick[Si]>=N; Si++)

; //初始的增量Sedgewick[Si]不能超过待排序的序列长度N

for(D=Sedgewick[Si]; D>0; D=Sedgewick[++Si])

for(P=D; P<N; P++) //插入序列

{

temp = A[P];

for(i=P; i>=D && A[i-D]>temp; i-=D)

A[i] = A[i-D];

A[i] = temp;

}

}

int main()

{

cout<<"输入样例："<<endl;

int Numb; //选择的算法编号

int N; //待排序元素的个数

cin>>Numb;

cin>>N;

int A[N];

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

cin>>A[i];

cout<<"输出样例："<<endl;

switch(Numb)

{

case 1: //堆排序

HeapSort(A, N);

break;

case 2: //冒泡排序

BubbleSort(A, N);

break;

case 3: //直接插入排序

InsertionSort(A, N);

break;

case 4: //希尔排序

ShellSort(A, N);

break;

default: break;

}

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

cout<<A[i]<<" ";

return 0;

}

/\*

输入样例：

1

12

57 40 38 11 13 34 48 75 6 19 9 7

\*/

**2、用快速排序对数据进行升序排列**

#include<iostream>

using namespace std;

//交换函数

void Swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void InsertionSort(int A[], int N) **//简单排序函数**

{

int P, i;

int temp;

for(P=1; P<N; P++)

{

temp = A[P]; //取出未排序元素中的第一个元素

for(i=P; i>0&&A[i-1]>temp; i--)

A[i] = A[i-1]; //依次与已排序序列中的元素比较并右移

A[i] = temp; //找到了合适的位置，放进去

}

}

int Median3(int A[], int Left, int Right) **//确定主元函数**

{

int Center = (Left+Right)/2;

if(A[Left] > A[Center])

Swap(&A[Left], &A[Center]);

if(A[Left] > A[Right])

Swap(&A[Left], &A[Right]);

if(A[Center] > A[Right])

Swap(&A[Center], &A[Right]);

//此时已满足A[Left]<=A[Center]<=A[Right]

Swap(&A[Center], &A[Right-1]); //将基准Pivot藏到右边

//只需要考虑A[Left+1]...A[Right-2]

return A[Right-1]; //返回基准Pivot

}

void Qsort(int A[], int Left, int Right) **//核心递归函数**

{

int Pivot, Cutoff = 5; //Cutoff为阈值

int Low, High;

if(Cutoff <= Right-Left) //如果序列元素充分多，进入快速排序

{

Pivot = Median3(A, Left, Right); //选基准

Low = Left;

High = Right -1;

while(1) //将序列中比基准小的移到基准左边，大的移到右边

{

while(A[++Low] < Pivot);

while(A[--High] > Pivot);

if(Low < High)

Swap(&A[Low], &A[High]);

else

break;

}

Swap(&A[Low], &A[Right-1]); //将基准换到正确的位置

Qsort(A, Left, Low-1); //递归解决左边

Qsort(A, Low+1, Right); //递归解决右边

}

else

InsertionSort(A+Left, Right-Left+1); //元素太少，用直接插入排序（简单排序）

}

int main()

{

cout<<"输入样例："<<endl;

int N; //待排序的元素数

cin>>N;

int A[N];

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

cin>>A[i];

cout<<"输出样例："<<endl;

Qsort(A, 0, N-1); //调用快排函数

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

cout<<A[i]<<" ";

return 0;

}

/\* 输入样例：

10

49 35 68 99 70 13 25 50 111 60

输出样例：

13 25 35 49 50 60 68 70 99 111

\*/

**3、给定整数数组nums和整数k，请返回数组中第k个最大的元素**

#include<iostream>

using namespace std;

//交换函数

void Swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

//冒泡排序，当获得第k个大的元素的时候就停止

void BubbleSort(int A[], int N, int k)

{

for(int P=N-1; P>=N-k; P--) //冒泡循环k次以后停止

for(int i=0; i<P; i++) //此为一趟冒泡，每次找出一个最大元素，被交换到最右端

if(A[i] > A[i+1])

Swap(&A[i], &A[i+1]);

cout<<A[N-k]<<endl; //冒泡循环k次以后，输出A的第N-k个元素

}

int main()

{

cout<<"输入样例："<<endl;

int nums[50];

int N = 0; //待排序的元素数

int k; //第k个最大的元素

while(1)

{

cin>>nums[N];

if(nums[N]==-1) break;

N++;

}

cin>>k;

cout<<"输出样例："<<endl;

if(k <= N)

BubbleSort(nums, N, k); //冒泡排序，找到k就停止

else

cout<<"输入的k值超过了整数序列的个数！"<<endl;

return 0;

}

/\*

输入样例：

3 2 1 5 6 4 -1

2

输出结果：

5

\*/

1、L为顺序表，删除下标i到j的所有元素（算法）

void Delete\_i\_j(List L, int i, int j)

{

for(int k=1;k<=j-i+1;k++)

{

for(int t=i; t<L->last; t++)

L->Data[t]=L->Data[t+1];

L->Last--;

}

}

2、判别给定的二叉树是否为二叉搜索树（算法）

//方法一:二叉搜索树中序遍历得到升序排列（假设数据为整型）

//若函数IsBST()的返回值为1，则是二叉搜索树；为0则不是。

int TreeNode[50];

int i=0;

bool IsBST(BinTree BT)

{ //判断该二叉树是否为二叉搜索树

InorderTraversal(BT);//调用中序遍历函数

bool flag=1;

int k=0;

while(TreeNode[k+1]!=Null)

{ //判断二叉树中序遍历的序列是否为升序

if(TreeNode[k]>=TreeNode[k+1])

{

flag=0;

break;

}

k++;

}

return flag;

}

3、求n个数据中的最大值

#include<iostream>

using namespace std;

void getMax(int n,float num[])

{

float n\_max=num[0];

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

{

if(num[i]>n\_max)n\_max=num[i];

}

cout<<"这n个数据中的最大值为："<<n\_max;

}

int main()

{

int N;

cout<<"请输入数据的个数 n=";

cin>>N;

float a[N];

cout<<"请输入这n个数据："<<endl;

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

{

cin>>a[i];

}

getMax(N,a);

return 0;

}

//该算法的时间复杂度为：T(n)=O(n)

4、递归法求最大公因子

#include<iostream>

using namespace std;

int getGCD(int m,int n)

{

int t,k;

if(m<n)

{

t=m;m=n;n=t;

}

k=m%n;

if(k==0)

return n;

else

{

m=n;

n=k;

return getGCD(m,n);

}

}

int main()

{

int x,y,GCD;

cin>>x>>y;

GCD=getGCD(x,y);

cout<<"最大公因子是："<<GCD;

return 0;

}

4.5、辗转相除法求最大公因子

#include<iostream>

using namespace std;

void getGCD(int m,int n)

{

int t,k;

if(m<n)

{

t=m;m=n;n=t;

}

do

{

k=m%n;

m=n;

n=k;

}while(n>0);

cout<<"最大公因子是："<<m;

}

int main()

{

int x,y;

cin>>x>>y;

getGCD(x,y);

return 0;

}

5、设计函数使带有头结点的单链表L元素递减有序（算法）

void Sort(List L)

{

Position p;

ElementType temp;

p=L; //p指向表头

int i=1; //用i++来记录链表中不为空的data元素的总个数

p=p->next; //p指向第一个元素

//先用冒泡法排序一次，统计元素i的总个数

while(p->next!=NUll)

{

if(p->data < p->next->data)

{

temp = p->data;

p->data = p->next->data;

p->next->data = temp;

}

p=p->next;

i++;

}

//继续使用冒泡法把剩下的元素排序

for(int j=i-2;j>0;j--) //j是剩下的要排序的次数

{

p=L; //p指向表头

p=p->next; //p指向第一个元素

int k=1;

while(k<=j) //k是每次排序相邻比较的次数

{

if(p->data < p->next->data)

{

temp = p->data;

p->data = p->next->data;

p->next->data = temp;

}

p=p->next;

k++;

}

}

}

6、假设二叉树采用二叉链表方式存储，输出先序遍历中第k个结点的值，k<=n。

#include<iostream>

using namespace std;

typedef struct TNode \*Position;

typedef Position BinTree;

struct TNode{

char Data;//假设数据类型为字符型

BinTree Left;

BinTree Right;

};

int i=0;

//采用先序创建的方法,根据输入的数值创建二叉树

void CreatBT(TNode \*\*Node)//此处传递的参数为二重指针

{

char data;

cin>>data;

\*Node = (BinTree)malloc(sizeof(TNode));

if(data == '#')

{

\*Node = NULL;

}

else// if ((data != '#') && (\*Node))

{

(\*Node)->Data = data;

(\*Node)->Left = NULL;

(\*Node)->Right = NULL;

CreatBT(&(\*Node)->Left);

CreatBT(&(\*Node)->Right);

}

}

//找到先序遍历中第k个值并输出

void PreTra\_k(BinTree BT, int k)

{

if(BT)

{

i++;

if(i==k)

{

cout<<BT->Data;

return;

}

if(BT->Left) PreTra\_k(BT->Left, k);

if(BT->Right) PreTra\_k(BT->Right, k);

}

}

int main()

{

BinTree BT=NULL;

cout<<"创建一棵二叉树："<<endl;

CreatBT(&BT);

int k=0;

cout<<"想要输出此二叉树第几个结点的值：";

cin>>k;

PreTra\_k(BT, k);

return 0;

}

7、给定两个两个单链表L1、L2，找出它们的公共结点，并返回（算法）

#define ERROR NULL //用空地址表示错误

Position Find\_L1\_L2(List L1, List L2)

{

Position p1=L1;

Position p2=L2;

Position p\_same[maxsize];//将公共结点的地址都存在这个指针数组中，maxsize预设，尽量足够大

i=0;

while(p1)

{

p2=L2;

while(p2)

{

if(p1==p2)

{

p\_same[i]=p1;

i++;

}

else

p2=p2->Next;

}

p1=p1->Next;

}

p\_same[i]=p1;

return p\_same;

}

8、用堆栈实现中缀表达式转后缀表达式

#include<iostream>

#include<stack>

#include<string>

using namespace std;

struct Node{

int num; //操作数

char op; //操作符

};

typedef struct Node expnode;

stack<expnode>s;

int ex\_c(char c)//将操作符转换成对应的优先级

{

int m;

switch(c)

{

case '+': m = 1; break;

case '-': m = 1; break;

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

case '/': m = 2; break;

default: break;

}

return m;

}

void exchange(string str)

{

expnode temp;//每次存放一个表达式中的字符

temp.op = '#';

s.push(temp);//栈底元素为#

for(int i=0;i<str.length();)//注意不用自动加1

{

if(str[i]>='0' && str[i]<='9')//判断是否为数字

{

temp.num = str[i] - '0';//将字符转换成数字储存

i++;

while(i<str.length() && str[i]>='0'&&str[i]<='9')//出现了超过一位的数字

{

temp.num = temp.num\*10 + (str[i] - '0');

i++;

}

cout<<temp.num;

}

else if(str[i]=='(')//判断是否为左括号

{

temp.op = str[i];

s.push(temp);

i++;

}

else if(str[i]==')')//判断是否为右括号

{

while(!s.empty() && s.top().op!='(')

{

cout<<s.top().op;

s.pop();

}

s.pop();//弹出左括号

i++;

}

else//判断是否为运算符

{

while(!s.empty() && s.top().op!='(' && s.top().op!='#' && ex\_c(str[i])<=ex\_c(s.top().op))

{

cout<<s.top().op;

s.pop();

}

temp.op = str[i];

s.push(temp);

i++;

}

}

while(!s.empty())

{

if(s.top().op!='#')

{

cout<<s.top().op;

s.pop();

}

else

{

s.pop();

}

}

}

int main()

{

string str;

cout<<"请输入一个中缀表达式："<<endl;

cin>>str;

cout<<"转换为后缀表达式为："<<endl;

exchange(str);

return 0;

}