目录

[**综合实验一** 3](#_Toc106052817)

[**综合实验二** 9](#_Toc106052818)

[**综合实验三** 14](#_Toc106052819)

[**综合实验四** 20](#_Toc106052820)

[**综合实验五** 24](#_Toc106052821)

[**综合实验六** 28](#_Toc106052822)

[**综合实验七** 37](#_Toc106052823)

[**综合实验八** 41](#_Toc106052824)

[**综合实验九** 45](#_Toc106052825)

[**综合实验十** 48](#_Toc106052826)

[**综合实验十一** 51](#_Toc106052827)

[**综合实验十二** 53](#_Toc106052828)

[**综合实验十三** 56](#_Toc106052829)

[**综合实验十四** 58](#_Toc106052830)

**综合实验一**

1. 实验内容

链表及其多项式相加

1. 算法说明

两个链表根据系数相加，然后存到第三个链表里。

1. 程序清单

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

typedef int ElemType;

typedef int Status;

const int maxn=3000;

const ll INF=1e18;

//---------------------------------------结构体定义区--------------------------------

typedef struct{

   float coef;//系数

   int expn;//指数

}term;

typedef struct Lnode//定义结点

{

    term data;//数据域

    struct Lnode \*next;//指针域

}LNode,\*LinkList;//将这个类型命名为LNode,并定义一个指向这个类型的指针

//----------------------------------------函数定义区---------------------------------

void CreateList\_L(LinkList &L,int n);//头插法创建链表

void creatlist(LinkList &L,int n);//尾插法创建链表

void Creatpolyn(LinkList &L, int a);//创建多项式

void MergeList\_L(LinkList &La,LinkList &Lb,LinkList &Lc);//链表的合并

void Orderinsertmerge(LinkList &L, term e);//按顺序插入项

void printpolyn(LinkList p);//打印多项式

void add(LinkList L1, LinkList L2, LinkList &L3);//加法，结果以L3返回

void List(LinkList L); //打印链表

Status GetElem\_L(LinkList L,int i,ElemType &e);//查找第i个元素并返回

Status ListInsert\_L(LinkList&L,int i,ElemType e);//前插法：在带头结点的单链线性表L中第i个位置之前插入元素e

Status ListDelete\_L(LinkList &L,int i,ElemType &e);//在带头结点的单链线性表L中，删除第i个元素，并由e返回其值

//----------------------------------------函数区--------------------------------------

void Creatpolyn(LinkList &L, int a)//创建多项式

{

    L = (LinkList)malloc(sizeof(Lnode));

    L->next = NULL;//建立头结点

    term e;

    for (int i = 1; i <= a; i++)

    {

        scanf("%f%d", &e.coef, &e.expn);

        Orderinsertmerge(L, e);//将所读得的数据插入链表

    }

}

void Orderinsertmerge(LinkList &L, term e)//按顺序插入项,每次插到表尾

{

    LinkList p, q;//q为操作指针，p指针指向q的下一位以方便排序

    p = L;//每次p都从头开始

    LinkList o = (LinkList)malloc(sizeof(Lnode));//o为临时节点

    while (p)//如果p=NULL,则跳出循环，在顺序表末尾添加结点

    {

        q = p;//通过循环移动指针q的位置

        p = p->next;//将p指向NULL

        if (e.coef == 0) return;//系数为0则直接返回该函数

        if (q->data.expn == e.expn)//次数相等则系数相加

        {

            q->data.coef += e.coef;

            return;

        }

        if (p && p->data.expn > e.expn)

        {

            /\*如果下一个节点所包含的次数大于想要插入的项的次数，

            就把想插入的项插入到当前节点的下一项以实现排序\*/

            o->data = e;//要插入的项赋值给临时节点

            q->next = o;//将临时结点插入到q的后面，p的前面

            o->next = p;

            return;

        }

    }

    o->data = e;

    q->next = o;

    o->next = NULL;//临时节点的指针域指向null，成为末尾节点

}

void add(LinkList L1, LinkList L2, LinkList &L3)//加法，结果以L3返回

{

    L3 = (LinkList)malloc(sizeof(Lnode));//建立头结点

    L3->next = NULL;

    LinkList q = L1->next;//指向L1的头结点的下一节点

    while (q)

    {

        Orderinsertmerge(L3, q->data);//插入至L3

        q = q->next;

    }

    q = L2->next;//指向L2的头结点的下一节点

    while (q)

    {

        Orderinsertmerge(L3, q->data);

        q = q->next;

    }

}

void printpolyn(LinkList p)//输出多项式

{

    LinkList q = p->next;//从头结点的下一结点开始

    while (q)

    {

        if (q->data.coef == 0)//如果系数为0，则不打印该项（加法会导致存在系数为零的项，不打印比删除方便）

        {

            q = q->next;

            continue;

        }

        printf("%.0f", q->data.coef);//先输出系数

        if (q->data.expn != 0)//如果次数不为零再输出，防止输出x^0

            printf(" %d\n", q->data.expn);

        q = q->next;//指向下一项

    }

}

//--------------------------------------主函数区--------------------------------------

int main(){

    LinkList L1, L2, L3;

    int n;scanf("%d", &n);

    Creatpolyn(L1, n);

    scanf("%d", &n);

    Creatpolyn(L2, n);

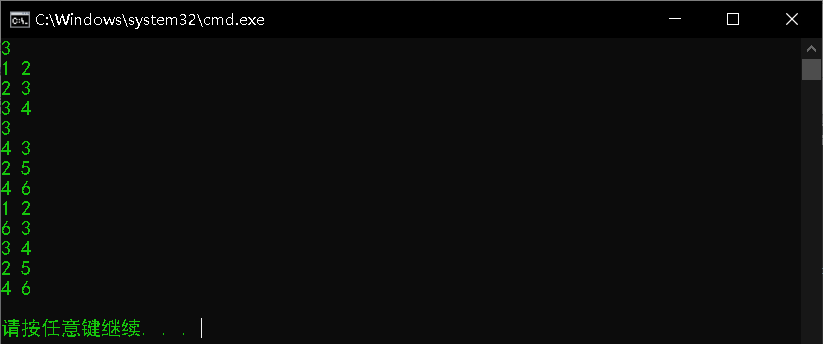
    add(L1, L2, L3);

    printpolyn(L3);

    return 0;

}

1. 运行结果



5、分析与思考

在代码中划分不同的区域，可以增强代码的可读性。

**综合实验二**

1. 实验内容

括号匹配判断算法

1. 算法说明

利用数据结构中的栈来实现括号匹配。

3、程序清单

#include<bits/stdc++.h>

#define STACK\_INIT\_SIZE 100//栈的长度

#define STACKINCREMENT 10//每次增加的长度

#define OK 1

#define OVERFLOW -2

#define ERROR 0

typedef char SElemType;

typedef int Status;

//---------------------------------------结构体定义区--------------------------------

typedef struct {

    SElemType \*base;//栈底指针，栈构造前和销毁后都为空

    SElemType \*top;//栈顶指针，指向栈顶元素的下一位置

    int stacksize;//当前分配的栈的存储空间数

}SqStack;

//----------------------------------------函数定义区---------------------------------

Status GetTop(SqStack S,SElemType &e)

{

    if(S.top==S.base) return ERROR;

    e=\*(S.top-1);

    return OK;

}

Status InitStack(SqStack &S)//初始化函数

{

    S.base = (SElemType \*)malloc(STACK\_INIT\_SIZE\*sizeof(SElemType));

    if(!S.base) exit(OVERFLOW);//代码健壮性检验

    S.top= S.base;//赋初值——栈顶=栈尾

    S.stacksize = STACK\_INIT\_SIZE;//栈长初始化

    return OK;

}

Status StackEmpty(SqStack S)//栈空函数

{

    return S.top == S.base;//栈空时候头为尾

}

Status Push(SqStack &S, SElemType e)//压栈函数

{

    if(S.top - S.base>=S.stacksize)

        return ERROR;

    \*S.top++ = e;//向top指针指向的空间赋值，赋完后指针top后移一位

    return OK;

}

Status Pop(SqStack &S, SElemType &e)//弹出函数

{

    if(S.base == S.top) return ERROR;

    e = \*--S.top;

    return OK;

}

Status Match(char \*st)//匹配括号函数

{

    SqStack S;

    InitStack(S);

    SElemType e;

    int flag = 1,i = 0;

    while(st[i])

    {

        switch(st[i])//字符串有值

        {

            case '(':

            case '[':

            case '{':

                Push(S,st[i]);//只要是左括号就全压入栈

                break;

            case ')':

            case '}':

            case ']':

                if(StackEmpty(S))//如果栈是空的话（右括号先进来的情况），flag=0

                {

                    flag = 0;

                    break;

                }

                Pop(S,e);

                if(!(e == '('&&st[i] == ')'||e == '['&&st[i] == ']'||e == '{'&&st[i] == '}'))

                {//左右不匹配的情况

                    flag = 0;

                    break;

                }

        }

        if(!flag) break;

        i++;

    }

    if(flag && StackEmpty(S))//标志为True且所有括号匹配完

        return true;

    else

        return false;

}

//--------------------------------------主函数区---------------------------------------

int main(){

    char st[1000];

    while(gets(st))

    {

        if(Match(st)) printf("yes\n");

        else printf("no\n");

    }

    return 0;

}

4、运行结果



5、分析与思考

主要是要根据左右括号来做出不同的操作。

**综合实验三**

1. 实验内容

编译器中的表达式求值问题

1. 算法说明

利用栈来解决编译器中的表达式求值问题。

3、程序清单

#include<bits/stdc++.h>

using namespace std;

int size = 1005;

typedef struct{

    int\* top;

    int\* base;

}SqStack;

char op[][8] = {//不同符号的优先级,为了方便所以用二维数组的形式来存储

        //分别为+-\*/()#

        {">><<<>>"},

        {">><<<>>"},

        {">>>><>>"},

        {">>>><>>"},

        {"<<<<<=>"},

        {">>>>>>>"},

        {"<<<<<<="}

};

char oop[8] = "+-\*/()#";//符号查询对应的数字

char Precede(char a, char b){//比较函数,比较两数的大小,返回>或<等

    int temp1 = 0;

    int temp2 = 0;

    for(int i = 0;i < 8;i++){

        if(a == oop[i])temp1 = i;//取出对应符号的数字,来获取大小优先关系

        if(b == oop[i])temp2 = i;

    }

    return op[temp1][temp2];

}

int Operate(int a, char theta, int b){//运算函数,很简单

    if (theta == '+'){

        return a+b;

    }else if(theta == '-'){

        return a-b;

    }else if(theta == '\*'){

        return a\*b;

    }else if(theta == '/'){

        return a/b;

    }

}

int InitStack(SqStack &s){//初始化栈函数

    s.base = (int\*) malloc(size\*sizeof(int));//开辟空间来给栈,这里先随便来一个1005,应该够用了,小了可能会Runtimeerror

    s.top = s.base;//top和base初始设为一样的

    return 1;

}

int GetTop(SqStack &s){//获取栈顶的元素

    if(s.top == s.base)return 0;//这就说明栈是空的

    return \*(s.top - 1);//因为栈的top是不存在的数,所以需要往下-1

}

int Push(SqStack &s,int e){//将数字推入栈

    \*s.top ++ = e;//此处先将e的值存储栈,然后将top++,先后顺序注意

    return 1;

}

int Pop(SqStack &s,int &e){//弹出栈顶的数字元素同时将那元素存入e中

    if(s.top == s.base)return 0;

    e = (int)\*(--s.top);//因为s中存的是char型,所以要用int强转一下

    return 1;

}

int Cpop(SqStack &s,char &e){//弹出栈顶的字符元素同时将那元素存入e中

    if(s.top == s.base)return 0;

    e = \*(--s.top);//注意这里的顺序是先执行--再是取top的值

    return 1;

}

void Evaluate(char ch){//转化函数,将中序表达式转化成后序表达式

    SqStack opnd,optr,rear;//opnd栈储存操作数,optr栈存储操作符,rear存后序表达式

    InitStack(opnd);//初始化栈

    InitStack(optr);

    InitStack(rear);

    int a,b;

    char x;

    char theta;

    Push(optr,'#');//先把#退入optr中,作为判断栈空的条件

    while(ch!='#'||GetTop(optr)!='#'){//当读入的字符不为#同时操作符栈顶元素不是#时

        if((ch >= '0') && (ch <= '9')){//

            Push(opnd,ch - '0');//因为ch是字符型所以需要减去'0'来转化成数字型存入opnd

            Push(rear,ch);//如果是数字就直接推入后续表达式中

            ch = getchar();//获取下一个字符

            if(ch == '\n')ch = '#';//由题目知,只有一行所以当读到的字符是回车时判断输入结束,将之替换成#

        }

        else

            switch(Precede(GetTop(optr),ch)){//判断ch和操作符栈顶的元素之间的优先关系

                case '=':

                    Cpop(optr,x);//当优先级相同时也就意味着是一对括号,所以直接取出optr中的(就好了,所以就直接弹出

                    ch = getchar();

                    if(ch == '\n')ch = '#';

                    break;

                case '>'://如果栈顶的优先级高就进行运算

                    Cpop(optr,theta);//弹出栈顶的操作符

                    Pop(opnd, b);//这里有先后顺序,先弹出b再是a

                    Pop(opnd, a);

                    Push(rear, theta);//将即将进行运算的操作符存入后续表达式中

                    Push(opnd, Operate(a, theta, b));//将运算后的结果再存入操作数中

                    break;

                case '<':

                    Push(optr,ch);//如果优先级高于栈顶元素就推入栈

                    ch = getchar();

                    break;

            }

    }

    opnd.top = opnd.base;//将opnd栈置空,这里将top和base相等也可以达到这个效果

    int \*p = rear.base;//新定义一个p指针来获取后缀表达式

    while(p!=rear.top)//意思是当rear栈不为空时

    {

        cout << (char)\*p << " ";//因为是int型所以需要强转成char

        if(\*p >= '0' && \*p <= '9')

            Push(opnd,\*p - '0');//是数字就直接推入操作数栈

        else//是操作符就直接进行运算

        {

            Pop(opnd,b);

            Pop(opnd,a);

            Push(opnd, Operate(a,\*p,b));

        }

        p++;//取下一个字符

    }

    cout << endl << GetTop(opnd) << endl;//此处的opnd中只有一个数,也就是我们需要求的结果

}

int main(){

    char ch;

    while(cin >> ch){

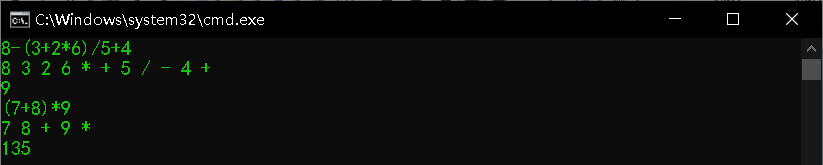
        if(ch != '\n')Evaluate(ch);//当读入数据不是回车符时,进行转化函数

    }

    return 0;

}

1. 运行结果



5、分析与思考

要注意输入数据和输入数据的格式。

**综合实验四**

1. 实验内容

第3章栈与队列——循环队列插入与删除

2、算法说明

利用循环队列来实现插入和删除操作

3、程序清单

#include <bits/stdc++.h>

using namespace std;

typedef int QElemType;

typedef int Status;

#define OK 1

#define ERROR 0

#define OVERFLOW -2

#define MAXQSIZE 100001//最大队列长度

int head=0,tail=0,flag=0;//flag=0表示未满

int a[MAXQSIZE];

char s[20];

//---------------------------------------结构体定义区--------------------------------

typedef struct QNode{

    QElemType data;

    struct QNode \*next;

}QNode,\*QueuePtr;

typedef struct{

    QueuePtr front;//对头指针

    QueuePtr rear;//队尾指针

}LinkQueue;

typedef struct{

    QElemType \*base;//初始化的动态分配内存空间

    int front;//头指针，若队列不空，则指向队头元素

    int rear;//尾指针，若队列不空，指向队列尾元素的下一个位置

}SqQueue;

//---------------------------------------函数声明区--------------------------------

Status InitQueue(LinkQueue &Q);//构造一个空队列Q

Status QueueLength(SqQueue Q);//返回Q的元素个数，即队列的长度

Status DestroyQueue(LinkQueue &Q);//销毁队列Q

Status EnQueue(LinkQueue &Q,QElemType e);//插入元素e为Q的新的队尾元素

Status EnQueue(int x);//入队

Status DeQueue(LinkQueue &Q,QElemType &e);//队头元素出队

Status DeQueue();//队头元素出队

//---------------------------------------函数定义区--------------------------------

Status InitQueue(SqQueue &Q)//构造一个循环空队列Q

{

    Q.base=(QElemType\*)malloc(MAXQSIZE\*sizeof(QElemType));

    if(!Q.base)exit(OVERFLOW);

    Q.front=Q.rear=0;

    return OK;

}

Status QueueLength(SqQueue Q)//返回Q的元素个数，即队列的长度

{

    return (Q.rear-Q.front+MAXQSIZE)%MAXQSIZE;

}

Status EnQueue(int x)//入队

{

    if(tail==head&&flag==1) return 0;//队满

    tail=(tail+1)%MAXQSIZE;

    a[tail]=x;

    if(tail==head) flag=1;

    return 0;

}

Status DeQueue()//队头元素出队

{

    if(head==tail&&flag==0) return -1;

    int e;

    head=(head+1)%MAXQSIZE;

    e=a[head];

    if(head==tail) flag=0;

    return e;

}

Status DestroyQueue(LinkQueue &Q)//销毁队列Q

{

    while(Q.front)

    {

        Q.rear=Q.front->next;

        free(Q.front);

        Q.front=Q.rear;

    }

    return OK;

}

//---------------------------------------主函数区-----------------------------------

int main() {

    int n,x,t;scanf("%d",&n);

    while(n--)

    {

        scanf("%s",&s);

       //printf("%s\n",a);

       if(s[0]=='e')

       {

            scanf("%d",&x);

            EnQueue(x);

       }

       else if(s[0]=='d')

       {

           t=DeQueue();

           printf("%d\n",t);

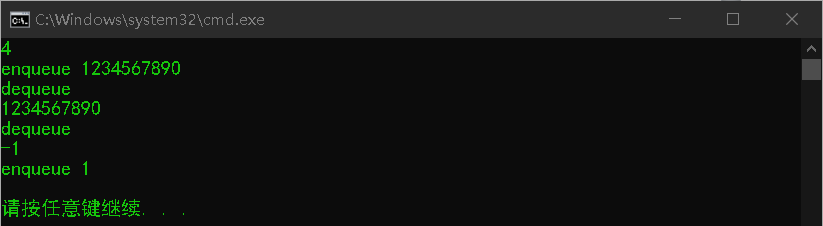
       }

    }

    return 0;

}

4、运行结果



5、分析与思考

循环队列可以很好的利用空间。

**综合实验五**

1. 实验内容

实验三 KMP算法

1. 算法说明

利用KMP算法来进行模式匹配。

3、程序清单

#include <bits/stdc++.h>

using namespace std;

char S[100005],T[100005];

int nextt[100005];

//-----------------------------------函数声明区------------------------------------------

int index(char S[],char T[],int pos);//朴素的模式匹配算法，时间复杂度为O(m\*n)

void Get\_next(char T[]);//获取模式串的next数组

int kmp(char S[],char T[]);//KMP算法,时间复杂度为O(m+n)

//-----------------------------------函数定义区------------------------------------------

int index(char S[],char T[],int pos)//朴素的模式匹配算法，时间复杂度为O(m\*n)

{

    int i=pos,j=0;

    int length1=strlen(S);

    int length2=strlen(T);

    while(i<=length1-1&&j<=length2-1)

    {

        if(S[i]==T[j]) i++,j++;

        else i=i-j+1,j=0;

    }

        //if(j>=length2) return i-length2;

        //printf("%d\n",j);

        if(j>=length2) return 1;

        else return 0;

}

void Get\_next(char T[])//获取模式串的next数组

{

    int i=0,j=-1;

    int length2=strlen(T);

    nextt[0]=-1;

    while(i<length2)

    {

        if(j==-1||T[i]==T[j]) nextt[++i]=++j;

        else j=nextt[j];

    }

}

int kmp(char S[],char T[])//KMP算法

{

    int i=0,j=0;

    Get\_next(T);

    int length1=strlen(S);

    int length2=strlen(T);

    while(i<length1)

    {

        if(j==-1||S[i]==T[j]) i++,j++;

        else j=nextt[j];

        //if(j>=length2) return i-length2;

        if(j>=length2) return 1;

    }

    return 0;

}

//-----------------------------------主函数区------------------------------------------

int main()

{

    while(scanf("%s",S)!=EOF)

    {

        int n;scanf("%d",&n);

        while(n--)

        {

            scanf("%s",T);

            //int flag=kmp(S,T);

            int flag=index(S,T,0);

            if(flag) printf("yes\n");

            else printf("no\n");

        }

    }

    return 0;

}

1. 运行结果



5、分析与思考

要理解KMP算法中的next数组，并且还要知道可以通过引入nextval数组来进一步优化KMP算法。

**综合实验六**

1. 实验内容

由先序遍历和中序遍历序列建立二叉树并进行后序遍历

1. 算法说明

根据先序遍历和中序遍历序列建立二叉树并进行后序遍历

3、程序清单

#include<bits/stdc++.h>

#define MAX\_TREE\_SIZE 100//二叉树的最大结点数

#define OK 1

#define OVERFLOW -2

#define ERROR 0

using namespace std;

typedef char TElemType;

typedef int Status;

typedef struct BiTNode{

    TElemType data;

    struct BiTNode \*lchild,\*rchild,\*parent;

    int deep;//结点的深度

}BiTNode,\*BiTree;

typedef enum PointerTag{Link,Thread};//定义一个枚举类型

typedef struct BiThrNode{//线索二叉树的结点

    TElemType data;

    struct BiThrNode \*lchild,\*rchild;//左右指针

    PointerTag LTag,RTag;//左右标志

}BiThrNode,\*BiThrTree;

typedef struct BPTNode{

    TElemType data;

    int parent;

    char LRTag;

}BPTNode;

typedef struct BPTree{

    BPTNode nodes[MAX\_TREE\_SIZE];

    int num\_node;

    int root;

}BPTree;

Status PrintElement(TElemType e)

{   //最简单的Visit()函数

    //调用实例：PreOrderTraverse(T,PrintElement)

    printf("%d ",e);

    return OK;

}

Status Visit(TElemType e)

{

    printf("%c",e);

    return OK;

}

Status CreatBiTree(BiTree &T)//按先序次序构建二叉树

{

    TElemType ch;

    scanf("%c",&ch);

    if(ch=='#'||ch=='\n') T=NULL;

    else{

        if(!(T=(BiTNode\*)malloc(sizeof(BiTNode)))) exit(OVERFLOW);

        T->data=ch;

        CreatBiTree(T->lchild);

        CreatBiTree(T->rchild);

    }

    /\* if(!(T=(BiTNode\*)malloc(sizeof(BiTNode)))) exit(OVERFLOW);

    T->data=ch;//生成根据结点

    CreatBiTree(T->lchild);//构造左子树

    CreatBiTree(T->rchild);//构造右子树 \*/

    return OK;

}

Status CreatBiTree\_depth(BiTree &T,int d)//按先序次序构建二叉树,并定义计算每个结点的深度

{

    TElemType ch;

    scanf("%c",&ch);

    if(ch=='#'||ch=='\n') T=NULL;

    else{

        if(!(T=(BiTNode\*)malloc(sizeof(BiTNode)))) exit(OVERFLOW);

        T->data=ch;T->deep=d;

        CreatBiTree\_depth(T->lchild,d+1);

        CreatBiTree\_depth(T->rchild,d+1);

    }

    /\* if(!(T=(BiTNode\*)malloc(sizeof(BiTNode)))) exit(OVERFLOW);

    T->data=ch;//生成根据结点

    CreatBiTree(T->lchild);//构造左子树

    CreatBiTree(T->rchild);//构造右子树 \*/

    return OK;

}

void CreatBiTreeP\_I(char \*pre,char \*in,int n)//由先序遍历和中序遍历序列建立二叉树

{

    if(n<=0) return ;

    int len=strchr(in,pre[0])-in;

    CreatBiTreeP\_I(pre+1,in,len);

    CreatBiTreeP\_I(pre+len+1,in+len+1,n-len-1);

    printf("%c",pre[0]);

}

Status BitreeDepth(BiTree T)//求二叉树的深度

{

    int depth,ldepth,rdepth;

    if(T==NULL) depth=0;

    else{

        ldepth=BitreeDepth(T->lchild);

        rdepth=BitreeDepth(T->rchild);

        depth=1+(ldepth>rdepth?ldepth:rdepth);

    }

    return depth;

}

Status InOrderTraverse\_Thr(BiThrTree T,Status(\*Visit)(TElemType e))//以双向线索链表为存储结构时对二叉树进行遍历的算法

{

    BiThrNode \*p;

    p=T->lchild;//p指向根节点

    while(p!=T)

    {

        while(p->LTag==Link) p=p->lchild;

        if(!Visit(p->data)) return ERROR;

        while(p->RTag==Thread&&p->rchild!=T)

        {

            p=p->rchild;Visit(p->data);//访问后继结点

        }

        p=p->rchild;

    }

    return OK;

}

void InThreading(BiThrTree p)//二叉树先线索化函数

{

    BiThrNode \*pre;

    if(p)

    {

        InThreading(p->lchild);//左子树线索化

        if(!p->lchild)//前驱线索

        {

            p->LTag=Thread;

            p->lchild=pre;

        }

        if(!pre->rchild)//后继线索

        {

            pre->RTag=Thread;

            p->lchild=p;

        }

        pre=p;//保持pre指向p的前驱

        InThreading(p->rchild);//右子树线索化

    }

}

Status InorderThreading(BiThrTree&Thrt,BiThrTree T)//遍历线索化链表

{

    BiThrNode \*pre;

    if(!(Thrt=(BiThrTree)malloc(sizeof(BiThrNode)))) exit(OVERFLOW);

    Thrt->LTag=Link;Thrt->RTag=Thread;

    Thrt->rchild=Thrt;

    if(!T) Thrt->lchild=Thrt;

    else{

        Thrt->lchild=T;pre=Thrt;

        InThreading(T);

        pre->rchild=Thrt;pre->RTag=Thread;

        Thrt->rchild=pre;

    }

    return OK;

}

Status PreOrderTraversal(BiTree T)//先序遍历的递归算法

{

    if(T)

    {

        printf("%c(%d)",T->data,T->deep);

        PreOrderTraversal(T->lchild);

        PreOrderTraversal(T->rchild);

    }

    return OK;

}

Status InOrderTraversal(BiTree T)

{

    if(T)

    {

        InOrderTraversal(T->lchild);

        Visit(T->data);

        InOrderTraversal(T->rchild);

    }

    return OK;

}

Status PostOrderTraversal(BiTree T)

{

    if(T)

    {

        PostOrderTraversal(T->lchild);

        PostOrderTraversal(T->rchild);

        Visit(T->data);

    }

    return OK;

}

void CountLeaf(BiThrTree T,int &count)//统计叶子结点个数

{

    if(T)

    {

        if((!T->lchild)&&(!T->rchild)) count++;

        CountLeaf(T->lchild,count);

        CountLeaf(T->rchild,count);

    }

}

Status BitreeLeafCount(BiTree T)//统计叶子结点个数

{

    if(T==NULL) return 0;

    else{

        if(T->lchild==NULL&&T->rchild==NULL) return 1;

        else return BitreeLeafCount(T->lchild)+BitreeLeafCount(T->rchild);

    }

}

Status BitreeCount(BiTree T)//求结点个数

{

    if(T==NULL) return 0;

    else return BitreeCount(T->lchild)+BitreeCount(T->rchild);

}

Status ChangeNode(BiTree bt)//交换左右子树

{

    BiTNode\*temp;

    if(bt==NULL) return 0;

    ChangeNode(bt->lchild);

    ChangeNode(bt->rchild);

    temp=bt->lchild;

    bt->lchild=bt->rchild;

    bt->rchild=temp;

}

char a[100005],b[100005];

int main(){

    while(~scanf("%s%s",a,b))

    {

        CreatBiTreeP\_I(a,b,strlen(a));

    }

    return 0;

}

4、运行结果



5、分析与思考

可以根据两个不同方式的遍历序列来建立二叉树。

**综合实验七**

1. 实验内容

二叉树的建立、中序遍历、叶子结点及深度

1. 算法说明

二叉树的建立、中序遍历、叶子结点及深度

3、程序清单

#include <bits/stdc++.h>

#define LEN sizeof(struct Bitnode)//求结构体的大小

 struct Bitnode

{

    char data;                  //存储数据

    struct Bitnode \*lchild;     //左孩子

    struct Bitnode \*rchild;     //右孩子

};

/\* 以前序遍历的方式创建二叉树 \*/

int creatBitree(struct Bitnode \*(\*T))

{

    char data;

   // data = getchar();

    scanf("%c",&data);          //输入你要建立的树

    if(data == '#')             //用'#'标志是空节点

    {

        \*T = NULL;              //当出现'#'时，返回上一层函数，该分支创建完成

        return 0;

    }

    else

    {

        \*T = (struct Bitnode \*)malloc(LEN); //申请空间

        if(T == NULL)

        {

            printf("申请空间失败\n");

            return 0;

        }

        else

        {

            (\*T)->data = data;      //把输入的数据，存储到树的相应节点中

            creatBitree(&((\*T)->lchild));//创建左子树

            creatBitree(&((\*T)->rchild));//创建右子树

        }

    }

}

/\*中序遍历 \*/

void middleBitree(struct Bitnode \*T)

{

    if(T == NULL)

    {

        return;

    }

    else

    {

        middleBitree(T->lchild);//递归遍历左子树

        printf("%c", T->data);//打印节点数据

        middleBitree(T->rchild);//递归遍历右子树

    }

}

/\* 求取树的深度 \*/

int treeDeep(struct Bitnode \*T)

{

    int deep = 0;

    if(T != NULL)//当树不为空时

    {

        int leftdeep;

        int rightdeep;

        leftdeep = treeDeep(T->lchild);//求左子树的深度

        rightdeep = treeDeep(T->rchild);//右子树的深度

        deep = leftdeep >= rightdeep?leftdeep+1:rightdeep+1;//如果左子树的深度大于等于右子树的深度则左子树的深度+1,否则右子树+1

    }

    return deep;//返回数的深度

}

/\* 求取树的叶子节点数 \*/

int treeLeaf(struct Bitnode \*T)

{

    static int count;//定义静态变量count,未初始化存储在.bss段,作用域为当前程序，相当于全局变量

    if(T != NULL)

    {

        if((T->lchild) == NULL && (T->rchild) == NULL)//如果左子树和右子树都为空

        {

            count++;

        }

        treeLeaf(T->lchild);//递归遍历左子树

        treeLeaf(T->rchild);//递归遍历右子树

    }

    return count;//返回叶子节点数

}

int main()

{

    struct Bitnode \*T;

    int depth = 0;//树的深度

    int leafconut = 0;//树的叶子节点数

    creatBitree(&T);//创建树

    printf("中序遍历: ");

    middleBitree(T);

    printf("\n");

    depth = treeDeep(T);

    leafconut = treeLeaf(T);

    printf("叶子结点个数:%d\n", leafconut);

    printf("树的深度为：%d\n",depth);

    return 0;

}

1. 运行结果



1. 分析与思考

把不同的功能封装到不同的函数里，可以让主函数变得简洁易懂。

**综合实验八**

1. 实验内容

二叉树按中序遍历输出，并在遍历时求出叶子结点的个数与深度。

1. 算法说明

二叉树按中序遍历输出，并在遍历时求出叶子结点的个数与深度。

3、程序清单

#include <bits/stdc++.h>

#define LEN sizeof(struct Bitnode)//求结构体的大小

 struct Bitnode

{

    char data;                  //存储数据

    struct Bitnode \*lchild;     //左孩子

    struct Bitnode \*rchild;     //右孩子

};

/\* 以前序遍历的方式创建二叉树 \*/

int creatBitree(struct Bitnode \*(\*T))

{

    char data;

   // data = getchar();

    scanf("%c",&data);          //输入你要建立的树

    if(data == '#')             //用'#'标志是空节点

    {

        \*T = NULL;              //当出现'#'时，返回上一层函数，该分支创建完成

        return 0;

    }

    else

    {

        \*T = (struct Bitnode \*)malloc(LEN); //申请空间

        if(T == NULL)

        {

            printf("申请空间失败\n");

            return 0;

        }

        else

        {

            (\*T)->data = data;      //把输入的数据，存储到树的相应节点中

            creatBitree(&((\*T)->lchild));//创建左子树

            creatBitree(&((\*T)->rchild));//创建右子树

        }

    }

}

/\*中序遍历 \*/

void middleBitree(struct Bitnode \*T)

{

    if(T == NULL)

    {

        return;

    }

    else

    {

        middleBitree(T->lchild);//递归遍历左子树

        printf("%c", T->data);//打印节点数据

        middleBitree(T->rchild);//递归遍历右子树

    }

}

/\* 求取树的深度 \*/

int treeDeep(struct Bitnode \*T)

{

    int deep = 0;

    if(T != NULL)//当树不为空时

    {

        int leftdeep;

        int rightdeep;

        leftdeep = treeDeep(T->lchild);//求左子树的深度

        rightdeep = treeDeep(T->rchild);//右子树的深度

        deep = leftdeep >= rightdeep?leftdeep+1:rightdeep+1;//如果左子树的深度大于等于右子树的深度则左子树的深度+1,否则右子树+1

    }

    return deep;//返回数的深度

}

/\* 求取树的叶子节点数 \*/

int treeLeaf(struct Bitnode \*T)

{

    static int count;//定义静态变量count,未初始化存储在.bss段,作用域为当前程序，相当于全局变量

    if(T != NULL)

    {

        if((T->lchild) == NULL && (T->rchild) == NULL)//如果左子树和右子树都为空

        {

            count++;

        }

        treeLeaf(T->lchild);//递归遍历左子树

        treeLeaf(T->rchild);//递归遍历右子树

    }

    return count;//返回叶子节点数

}

int main()

{

    struct Bitnode \*T;

    int depth = 0;//树的深度

    int leafconut = 0;//树的叶子节点数

    creatBitree(&T);//创建树

    printf("中序遍历: ");

    middleBitree(T);

    printf("\n");

    depth = treeDeep(T);

    leafconut = treeLeaf(T);

    printf("叶子结点个数:%d\n", leafconut);

    printf("树的深度为：%d\n",depth);

    return 0;

}

1. 运行结果



1. 分析与思考

把不同的功能封装到不同的函数里，可以让主函数变得简洁易懂。

**综合实验九**

1. 实验内容

图的表示+深度和广度优先搜索

2、算法说明

图的表示+DFS和BFS

3、程序清单

#include<stdio.h>

#include<string.h>

#include<algorithm>

#include<queue>

using namespace std;

int map[110][110],vis[1100];

void bfs(int u, int n) {

    int i;

    queue<int> qu;

    qu.push(u);

    while(!qu.empty()) {

        int tmp = qu.front();

        vis[tmp]=1;

        qu.pop();

        printf("%d ", tmp);

        for(i = 0; i < n; i ++) {

            if(i == tmp) continue;

            if(map[tmp][i] && !vis[i]) {

                vis[i] = 1;

                qu.push(i);

            }

        }

    }

}

void dfs(int u, int n)

{

        int i;

        vis[u] = 1;

        printf("%d ", u);

        for(i = 0; i < n; i ++)

        {

            if(i == u)

            continue;

            if(map[u][i]>0 && !vis[i])

            {

            dfs(i, n);

            }

        }

}

int main()

{

    int n,m,a,b,i,j;

    while(~scanf("%d%d",&n,&m))

    {

     memset(map,0,sizeof(map));

     for(j=0;j<m;j++)

     {

     scanf("%d%d",&a,&b);

     map[a][b]=map[b][a]=1;

     }

     memset(vis, 0, sizeof(vis));

     for(i = 0 ; i < n; i ++)

     {

      if(!vis[i])

      dfs(i, n);

     }

      printf("\n");

     memset(vis, 0, sizeof(vis));

        for(i = 0 ; i < n; i ++) {

            if(!vis[i]) {

                bfs(i, n);

            }

        }

        printf("\n");

    }

}

4、运行结果



1. 分析与思考

要理解BFS和DFS的不同之处。

**综合实验十**

1. 实验内容

C语言Kruskal算法的实现

1. 算法说明

C语言Kruskal算法的实现

1. 程序清单

#include <bits/stdc++.h>

#define N 9   // 图中边的数量

#define P 6   // 图中顶点的数量

//构建表示边的结构体

struct edge {

    //一条边有 2 个顶点

    int initial;

    int end;

    //边的权值

    int weight;

};

//qsort排序函数中使用，使edges结构体中的边按照权值大小升序排序

int cmp(const void\* a, const void\* b) {

    return  ((struct edge\*)a)->weight - ((struct edge\*)b)->weight;

}

//克鲁斯卡尔算法寻找最小生成树，edges 存储用户输入的图的各个边，minTree 用于记录组成最小生成树的各个边

void kruskal\_MinTree(struct edge edges[], struct edge minTree[]) {

    int i, initial, end, elem, k;

    //每个顶点配置一个标记值

    int assists[P];

    int num = 0;

    //初始状态下，每个顶点的标记都不相同

    for (i = 0; i < P; i++) {

        assists[i] = i;

    }

    //根据权值，对所有边进行升序排序

    qsort(edges, N, sizeof(edges[0]), cmp);

    //遍历所有的边

    for (i = 0; i < N; i++) {

        //找到当前边的两个顶点在 assists 数组中的位置下标

        initial = edges[i].initial - 1;

        end = edges[i].end - 1;

        //如果顶点位置存在且顶点的标记不同，说明不在一个集合中，不会产生回路

        if (assists[initial] != assists[end]) {

            //记录该边，作为最小生成树的组成部分

            minTree[num] = edges[i];

            //计数+1

            num++;

            elem = assists[end];

            //将新加入生成树的顶点标记全部改为一样的

            for (k = 0; k < P; k++) {

                if (assists[k] == elem) {

                    assists[k] = assists[initial];

                }

            }

            //如果选择的边的数量和顶点数相差1，证明最小生成树已经形成，退出循环

            if (num == P - 1) {

                break;

            }

        }

    }

}

void display(struct edge minTree[]) {

    int cost = 0, i;

    printf("最小生成树为:\n");

    for (i = 0; i < P - 1; i++) {

        printf("%d-%d  权值：%d\n", minTree[i].initial, minTree[i].end, minTree[i].weight);

        cost += minTree[i].weight;

    }

    printf("总权值为：%d", cost);

}

int main() {

    int i;

    struct edge edges[N], minTree[P - 1];

    for (i = 0; i < N; i++) {

        scanf("%d %d %d", &edges[i].initial, &edges[i].end, &edges[i].weight);

    }

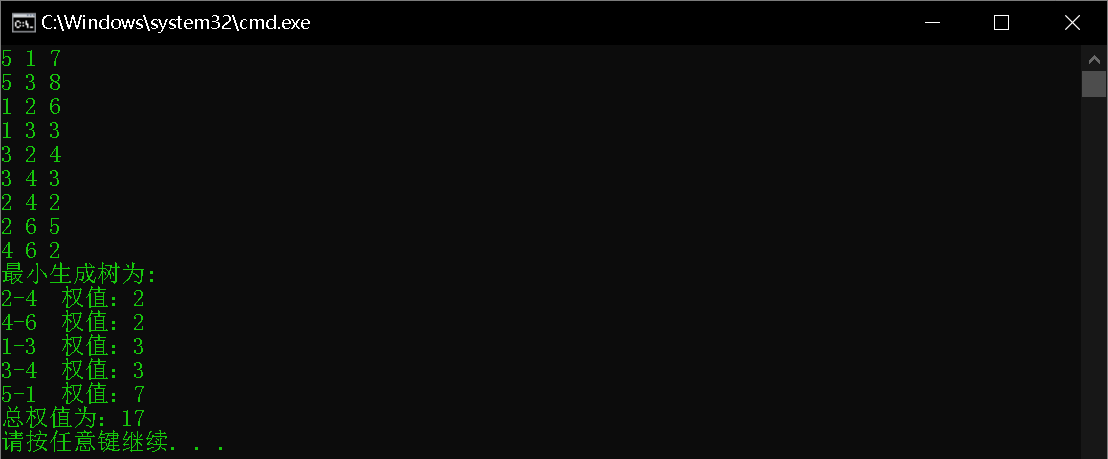
    kruskal\_MinTree(edges, minTree);

    display(minTree);

    return 0;

}

4、运行结果



1. 分析与思考

要理解Kruskal算法的实现过程，以及和Prim算法的不同之处。

**综合实验十一**

1. 实验内容

单源最短路径——Dijkstra算法

1. 算法说明

最短路径——Dijkstra算法

3、程序清单

#include<bits/stdc++.h>

const int INF = 1000000000;

int d[210],vis[210],w[210][210];

int main()

{

    int n,m,i,j,k;

    while(~scanf("%d%d", &n,&m))

    {

        memset(vis, 0, sizeof(vis));

        memset(w, 0, sizeof(w));

        int a,b,c,s,e;

        for(i=0; i<m; i++)

        {

            scanf("%d%d%d", &a,&b,&c);

            if(w[a][b] == 0) w[a][b] = w[b][a] = c;

            else if(c < w[a][b]) w[a][b] = w[b][a] = c;

        }

        scanf("%d%d", &s,&e);

        for(i=0; i<n; i++)

            d[i] = (i == s ? 0 : INF);

        for(i=0; i<n; i++)

        {

            int mmin = INF;

            for(j=0; j<n; j++)

                if(!vis[j] && d[j] <= mmin)

                    mmin = d[k=j];

            vis[k] = 1;

            for(j=0; j<n; j++)

                if(!vis[j] && w[k][j] && d[k]+w[k][j] < d[j])

                    d[j] = d[k]+w[k][j];

        }

        if(d[e] == INF) printf("-1\n");

        else  printf("%d\n", d[e]);

    }

    return 0;

}

4、运行结果



1. 分析与思考

Dijkstra算法的边权值不能为负数。

**综合实验十二**

1. 实验内容

查找——BST树的构建与应用

1. 算法说明

BST树的构建与应用

3、程序清单

#include<stdio.h>

#include<stdlib.h>

typedef struct Binode{

    int data;

    Binode \*lchild,\*rchild;

}Binode,\*Bitree;

int count=0;

int pretraverse(Bitree T)

{

    if(T)

    {

        printf("%d ",T->data);

        pretraverse(T->lchild);

        pretraverse(T->rchild);

    }

    return 1;

}

int SearchBST(Bitree T,int key,Bitree f,Bitree &p)

{

    if(!T){p=f;return 0;}

    else if(key==T->data){p=T;return 1;}

    else if(key<T->data){SearchBST(T->lchild,key,T,p);}

    else {SearchBST(T->rchild,key,T,p);}

}

int InsertBST(Bitree &T,int e)

{

    Binode \*s,\*p;

    if(!SearchBST(T,e,NULL,p)){

        s=(Bitree)malloc(sizeof(Binode));

        s->data=e;s->lchild=s->rchild=NULL;

        if(!p)T=s;

        else if(e<p->data) p->lchild=s;

        else p->rchild=s;

        count++;

        return 1;

    }

    else return -1;

}

int main()

{

    Bitree T;

    int i,n,t;

    T=NULL;

    scanf("%d",&n);

    for(i=0;i<n;i++)

    {

        scanf("%d",&t);

        InsertBST(T,t);

    }

    printf("%d\n",count);

    pretraverse(T);

    printf("\n");

}

4、运行结果



5、分析与思考

要理解二叉查找树建立的过程。

**综合实验十三**

1. 实验内容

实验八 快速排序

1. 算法说明

可以利用C++中封装好的sort()函数实现快速排序。

3、程序清单

#include<bits/stdc++.h>

using namespace std;

int n;

int a[20005];

int main(){

    while(~scanf("%d",&n))

    {

        memset(a,0,sizeof(a));

        for(int i=0;i<n;++i)

        {

            scanf("%d",&a[i]);

        }

        sort(a,a+n);

        for(int i=0;i<n;++i)

        {

            printf("%d ",a[i]);

        }

        printf("\n");

    }

    return 0;

}

4、运行结果



5、分析与思考

快速排序的时间复杂度是O(nlogn)

**综合实验十四**

1. 实验内容

堆的建立和维护，实践堆结构

1. 算法说明

将数组建成一个大根堆，每次将最后一个元素和大根元素交换，这样就使得最大的数排在最后。交换完之后，将剩下的数据进行堆化，再次建成大根堆。重复上述过程，直到得到有序数组。

3、程序清单

#include<bits/stdc++.h>

using namespace std;

void swap(int arr[], int a, int b) {

    int t = arr[a];

    arr[a] = arr[b];

    arr[b] = t;

}

void heapify(int arr[], int index, int heapSize) {

    int left = (index \* 2) + 1;

    while (left < heapSize) {

        int largest = left;

        if (left+1 < heapSize && arr[left+1] > arr[left]) {

            largest = left + 1;

        }

        if (arr[largest] <= arr[index]) {

            break;

        }

        swap(arr, largest, index);

        index = largest;

        left = index \* 2 + 1;

    }

}

void heapSort(int arr[], int length) {

    if (length < 2) {

        return;

    }

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

        heapify(arr, i, length);

    }

    int heapSize = length;

    swap(arr,0,--heapSize);

    while (heapSize > 0) {

        heapify(arr,0,heapSize);

        swap(arr,0,--heapSize);

    }

}

int main() {

    int n;

    scanf("%d", &n);

    int arr[n];

    for (int i = 0; i < n; i++) {

        scanf("%d", &arr[i]);

    }

    heapSort(arr,n);

    for (int i = 0; i < n; i++) {

        printf("%d ", arr[i]);

    }

}

1. 运行结果



5、分析与思考

本次实验学会了如何使用堆排序的方式对数据进行排序。学会堆排序的前提就是了解完全二叉树的结构如何用数组的方式表现出来。