**郝斌数据结构自学笔记**

**--知识点+程序源代码**

**2015.12 By-HZM**

1\_什么叫做数据结构

数据结构概述

定义

我们如何把现实中大量而复杂的问题以特定的数据类型和特定的存储结构保存到主存储器（内存）中，以及在此基础上为实现某个功能（比如查找某个元素，删除某个元素，对所有元素进行排序）而执行的相应操作，这个相应的操作也叫算法。

数据结构=个体的存储+个体的关系存储

算法=对存储数据的操作

2\_衡量算法的标准

算法

解题的方法和步骤

衡量算法的标准

1）时间复杂度：大概程序执行的次数，而非执行的时间

2）空间复杂度：算法执行过程中大概所占用的最大内存

3）难易程度

4）健壮性

3\_数据结构的特点

数据结构的地位

数据结构是软件中最核心的课程

程序=数据的存储+数据的操作+可以被计算机执行的语言

4\_预备知识\_指针\_1

5\_预备知识\_指针\_2

指针的重要性：

指针是C语言的灵魂

定义：

地址：

地址是内存单元的编号,从0开始的非负整数，范围：0-FFFFFFFF【0-4G-1】

CPU=====地址线，控制线，数据线=====内存

指针：

指针就是地址，地址就是指针。

指针变量是存放内存单元地址的变量。

指针的本质是一个操作受限的非负整数。

分类：

1.基本类型的指针

2.指针和数组的关系

变量并不一定连续分配，随机分配内存。

内存：

内存是多字节组成的线性一维存储空间。

内存的基本划分单位是字节。

每个字节含有8位，每一位存放1个0或1个1.

内存和编号是一一对应的。

软件在运行前需要向操作系统申请存储空间。在软件运行期间，该软件所占空间不再分配给其他软件。当软件运行完毕后，操作系统将回收该内存空间（操作系统并不清空该内存空间中遗留下来的数据）。

NOTE：1）指针变量也是变量，普通变量前不能加\*，常亮和表达式前不能加&。

2）局部变量只在本函数内部使用。

如何通过被调函数修改主调函数中普通变量的值。

1）实参为相关变量的地址；

2）形参为以该变量的类型为类型的指针变量；

3）在被调函数中通过 \*形参变量名的形式 的形式就可以修改主函数。

CASE 1

#include<stdio.h>

int main(void)

{

int \*p; //p是个变量名字，int\*表示该p变量只能存储int类型变量的地址

int i=10;

int j;

// j=\*p;

// printf("%d\n",j); //error,p未指定

// char ch='A';

// p=&ch; //error，类型不一致

p=&i; //p保存i的地址，p指向i；修改p的值不影响i的值，修改i的值不影响p的值；任何场合下，\*p和i可以互换。\*p等价于i。

//p=10; //error

j=\*p;//等价于j=i;

printf("i=%d,j=%d,\*p=%d\n",i,j,\*p);

return 0;

}

CASE 2

#include<stdio.h>

void f(int \* i)//不是定义了一个名字叫做\*i的形参，而是定义了一个形参，该形参名字叫做i，它的类型是int\*

{

\*i=100;

}

int main(void)

{

int i=9;

f(&i); //局部变量只在本函数内部使用。

printf("i=%d\n",i);

}

指针和数字

数组名：一维数组名是个指针变量，它存放的是一维数组第一个元素的地址，它的值不能被改变，一维数组名指向的是数组的第一个元素。

CASE 1

a[3]==\*(3+a); 3[a] ==\*(a+3)==\*(3+a);

int a[5]={1,2,3,4,5};

Show\_Aarry(a,5);//a等价于&a[0],&a[0]本身就是int\*类型

void Show\_Array(int \* p,int len)

{

Int I;

//P[2]=-1;// p[0]=\*p ; p[2]==\*(p+2)==\*(a+2)==a[2] ; p[i]就是主函数的a[i]

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

printf(“%d\n”,p[i]);

}

指针变量的运算

指针变量不能相加，不能相乘，不能相除。

如果两指针变量属于同一数组，则可以相减。

指针变量可以加减一整数，前提是最终结果不能超过指针变量

p+i的值是p+i\*(p所指向的变量所占的字节数)

p-i的值是p-i\*(p所指向的变量所占的字节数)

p++<==>p+1 p--<==>P-1

6\_所有的指针变量只占4个子节 用第一个字节的地址表示整个变量的地址

CASE 1

double \*p;

double x=66.6; //一个double占8个字节

p=&x;//x占8个字节，1个字节是8位，1个字节一个地址，p内只存放了一个地址，通常是字节的首地址

double arr[3]={1.1,2.2,3.3};

double \*q;

q=&arr[0];

printf(“%p\n”,q); //%p实际就是以十六进制输出

q=&arr[1];

q=printf(“%p\n”,q); //p,q相差8

无论指针指向的变量占多少个字节，指针变量统一都只占4个字节

7\_如何通过函数修改实参的值

发送地址

CASE 1 修改指针变量的值，只能修改地址

void f(int \*\*);

int main(void)

{

int i=9;

int \*p=&i;// \*p;p=&i;

printf(“%p\n”,p);

f(&p);

printf(“%p\n”,p);

return 0;

}

//void f(int \*q)

//{

// q=(int \*)0xffffffff; //错误，不会改变p的值

//}

void f(int \*\* q)

{

\*q=(int \*)0xffffffff;

}

8\_结构体的使用概述

结构体

为什么会出现结构体：

为了表示一些复杂的数据，而普通的基本类型变量无法满足要求

什么叫做结构体：

结构体是用户根据实际需要自己定义的复合数据类型

如何使用结构体：

两种方式——

struct Student st={1000,”zhagnsan”,20};

struct Student\*pst=&st;

1)通过结构体变量名来实现

st.sid

2)通过指向结构体变量的指针来实现【重点】

pst->sid

pst所指向的结构体变量中的sid这个成员

CASE 1

#include<stdio.h>

#include <string.h>

struct Student

{

int sid；

char name[200];

int age;

}; //分号不能省

Int main(void)

{

struct Student st={1000,”zhagnsan”,20};

printf(“%d,%s%d\n,”,st.sid,st.name,st.age);

printf(“%d,%s%d\n,”,st); //error

st.sid=99; //第一种

//st.name=”lisi”; //error

strcpy(st.name,”lisi”);

st.age=22;

struct Student\*pst;

pst=&st; //第二种

pst->sid=99; //pst->等价于(\*pst).sid，而(\*pst).sid等价于st.sid，所以pst->sid等价于st.sid

Return 0;

}

注意事项：

结构体变量不能加减乘除，但可以相互赋值

普通结构体变量和结构体指针变量作为函数传参的问题

CASE 2

#include<stdio.h>

struct Student

{

int sid；

char name[200];

int age;

};

void f(struct Student \*pst);

void g(struct Student st);

void g2(struct Student \*pst);

int main (void)

{

struct Student st; //已经为st分配好了内存

f(&st);

//g(st);

g2(&st);

// printf(“%d %s %d\n”,st.sid,st.name,st.age); //输出方法一

return 0;

}

void g(struct Student st) //整体变量赋值//输出方法二，速度慢，耗空间，耗内存，不推荐

{

printf(“%d %s %d\n”,st.sid,st.name,st.age);

}

void g2(struct Student \*pst)

{

printf(“%d %s %d\n”,pst->sid,pst->name,pst->age);

}

void f(struct Student \*pst)

{

(\*pst).sid=99;

strcpy(pst->name,”zhagnsan”);

pst->age=22;

}

9\_malloc()动态分配内存概述

动态内存的分配和释放

CASE 1

#icclude<stdio.h>

#include<malloc.h>

int main(void)

{

int a[5]={1,2,3,4,5}; //静态数组

int len;

printf(“请输入你需要分配的数组长度：len=”);

scanf(“%d”,&len);

int \*pArr=(int \*)malloc(sizeof(int)\*len); //(int \*)为强制转换，强制使pArr指向前四个字节。可以将pArr当做数组名来操作

//\*pArr=4;//类似于a[0]=4;

// pArr[1]=10;//类似于a[1]=10;

// printf(“%d %d\n”,\*pArr,pArr[1]);

//我们可以把pArr当做一个普通数组来使用

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

scanf(“%d”,&pArr);

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

printf(“%d\n”,\*(pArr+i));

free(pArr); //把pArr所代表的的动态分配的20个字节的内存释放

return 0;

}

10\_跨函数使用内存讲解及其示例

CASE 1

#include<stdio.h>

int f();

int main(void)

{

int i=10;

i=f();

printf(“i=%d\n”,i);

for(i=0;i<2000;++i)

f();

return 0;

}

int f()

{

int j=20;

return j;

}

CASE 2

main ()

{

int \*p;

fun(&p);

...

}

int fun (int \*\*q)

{

int s; //s为局部变量。调用完毕后s就没有了，最终p没有指向一个合法的整型单元

\*q=&s;

}

CASE 3

main()

{

int \*p;

fun(&p);

...

}

int fun(int \*\*q)

{

\*q=(int \*)malloc(4); //返回4个字节，只取第1个字节地址赋给\*q，\*q==p。执行完后，因为没有free()，内存没有释放。如果没有free()，整个程序彻底终止时才能释放

}

程序内部类定义方法

A aa=new A();

A \*pa=(A\*)malloc(sizeof(A));

CASE 4

#include<stdio.h>

#include<malloc.h>

struct Student

{

int sid;

int age;

}

struct Student \* CreatStudent(void);

void ShowStudent(struct Student \*);

int main(void)

{

struct Student \*ps;

ps=CreatStudent();

ShowStudent(ps);

return 0;

}

struct Student \* CreatStudent(void)

{

struct Student \*P=(struct Student \*)malloc(sizeof(struct Student));

p->sid=99;

p->age=88;

return p;

}

void ShowStudent(struct Student \*pst)

{

printf(“%d %d\n”,pst->sid,pst->age);

}

11\_复习

12\_连续存储数组的算法演示\_1

13\_连续存储数组的算法演示\_2

模块一：线性结构【把所有的结点用一根直线穿起来】

1）连续存储[数组]

2）离散存储[链表]

线性结构的两种常见应用之一 栈

线性结构的两种常见应用之二 队列

专题：递归

1. 1=2+3+4+...100的和

2. 求阶乘

3. 汉诺塔

4. 走迷宫

模块二：非线性结构

树

图

连续存储[数组]

1. 什么叫数组

元素类型相同，大小相同

2. 数组的优缺点：

CASE 1

#include<stdio.h>

#include<malloc.h> //包含了malloc函数

#include<stdlib.h> // 包含了exit函数

//定义了一个数据类型，该数据类型的名字叫做struct Arr,该数据类型含有3个成员，分别为pBase，len,cnt

struct Arr

{

int \*pBase;//存储的是数组第一个元素的地址

int len;//数组所能容纳的最大元素的个数

int cnt;//当前数组有效元素的个数

//int increment;//自动增长因子

};

void init\_arr(struct Arr \*pArr,int length);//初始化,使pBase指向一个有效的数组，而不再是垃圾数字

bool append\_arr(struct Arr \*pArr,int val);//追加，可能成功，可能失败

bool insert\_arr(struct Arr \*pArr,int pos,int val); //pos的值从1开始

bool delete\_arr(struct Arr \*pArr,int pos,int \*pVal);

int get();

bool is\_empty(struct Arr \*pArr);//是否已满

bool is\_full(struct Arr \*pAr);//是否为空

void sort\_arr(struct Arr \*pArr);//排序

void show\_arr(struct Arr \*pArr);//显示,分号不能省

void innversion\_arr(struct Arr \*pArr);//倒置

int main (void)

{

struct Arr arr; //只定义没初始化时，内部三个变量里都是垃圾数字

int val;

int posi=2;

int len=6;

//init\_arr(arr);//会输出垃圾数字，并不能改变arr的值

// printf("%d\n",arr.len);

init\_arr(&arr,len);//会输出垃圾数字，并不能改变arr的值

show\_arr(&arr);

append\_arr(&arr,1);

append\_arr(&arr,-3);

append\_arr(&arr,6);

append\_arr(&arr,45);

append\_arr(&arr,13);

if(append\_arr(&arr,34))

{

printf("追加成功！\n");

}

else

{

printf("追加失败！\n");

}

printf("追加之后的数组内容是:\n");

show\_arr(&arr);

if(delete\_arr(&arr,posi,&val))

{

printf("删除成功！\n");

printf("删除的元素是第%d个元素\n",posi);

printf("删除的元素是：%d\n",val);

}

else

{

printf("删除失败！\n");

}

/\*

append\_arr(&arr,1);

append\_arr(&arr,2);

append\_arr(&arr,3);

append\_arr(&arr,4);

append\_arr(&arr,5);

insert\_arr(&arr,1,99); //pos的值从1开始

\*/

/\* append\_arr(&arr,6);

append\_arr(&arr,7);

show\_arr(&arr);

if(append\_arr(&arr,8))

{

printf("追加成功！\n");

}

else

{

printf("追加失败！\n");

}

\*/

printf("删除之后的数组内容是:\n");

show\_arr(&arr);

innversion\_arr(&arr);//倒置

printf("倒置之后的数组内容是:\n");

show\_arr(&arr);

sort\_arr(&arr);

printf("排序之后的数组内容是:\n");

show\_arr(&arr);

return 0;

}

void init\_arr(struct Arr \*pArr,int length)

{

//(\*pArr).len=99;

pArr->pBase = (int\*)malloc(sizeof(int)\*length);

if(NULL==pArr->pBase)

{

printf("动态内存分配失败！\n");

exit(-1);//终止整个程序

}

else

{

pArr->len=length;

pArr->cnt=0;

}

return;

}

bool is\_empty(struct Arr \*pArr)//是否已满

{

if(0==pArr->cnt)

return true;

else

return false;

}

void show\_arr(struct Arr \*pArr)//显示

{

// if(数组为空)

// 提示用户数组为空

// else

// 输出数组有效内容

if(is\_empty(pArr)) //

{

printf("数组为空!\n");

}

else

{

for(int i=0;i<pArr->cnt;i++)

printf("%d ",pArr->pBase[i]);

printf("\n");

}

}

bool is\_full(struct Arr \*pArr)//是否为空

{

if(pArr->cnt==pArr->len)

return true;

else

return false;

}

bool append\_arr(struct Arr \*pArr,int val)//追加，可能成功，可能失败

{

//满时返回false

if(is\_full(pArr))

return false;

//不满时追加

pArr->pBase[pArr->cnt]=val;

(pArr->cnt)++;

return true;

}

bool insert\_arr(struct Arr \*pArr,int pos,int val) //pos的值从1开始

{

int i;

if(is\_full(pArr))

return false;

if(pos<1||pos>pArr->cnt+1) //

return false;

for (i=pArr->cnt-1;i>=pos-1;--i)

{

pArr->pBase[i+1]=pArr->pBase[i]; //i赋给i+1

}

pArr->pBase[pos-1]=val;

pArr->cnt++;

return true;

}

bool delete\_arr(struct Arr \*pArr,int pos,int \*pVal)

{

int i;

if(is\_empty(pArr))

return false;

if(pos<1||pos>pArr->cnt)

return false;

\*pVal=pArr->pBase[pos-1];

for(i=pos;i<pArr->cnt;++i)

{

pArr->pBase[i-1]=pArr->pBase[i];

}

pArr->cnt--;

return true;

}

void innversion\_arr(struct Arr \*pArr)//倒置

{

int i=0;

int j=pArr->cnt-1;

int t;

while(i<j)

{

t=pArr->pBase[i];

pArr->pBase[i]=pArr->pBase[j];

pArr->pBase[j]=t;

++i;

--j;

}

return;

}

void sort\_arr(struct Arr \*pArr)//排序

{

int i,j,t;

for(i=0;i<pArr->cnt;++i)

{

for(j=i+1;j<pArr->cnt;++j)

{

if(pArr->pBase[i]>pArr->pBase[j])

{

t=pArr->pBase[i];

pArr->pBase[i]=pArr->pBase[j];

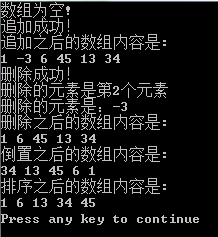
pArr->pBase[j]=t;

}

}

}

}



14\_链表的重要性

15\_typedef的用法

CASE 1

#include<stdio.h>

typedef int ZHAGNSAN;//为int再重新多取一个名字，ZHAGNSAN等价于int

typedef struct Student

{

int sid;

char name[100];

char sex;

}ST;

int main(void)

{

int i=10;//等价于ZHANGSAN i=10;

// ZHAGNSAN j=20;

// printf("%d\n",j);

struct Student st;//等价于ST st；

struct Student \*ps=&st;//等价于ST \*ps;

ST st2;

st2.sid=200;

printf("%d\n",st2.sid);

return 0;

}

CASE 2

#include<stdio.h>

typedef int ZHAGNSAN;//为int再重新多取一个名字，ZHAGNSAN等价于int

typedef struct Student

{

int sid;

char name[100];

char sex;

}\*PST; //PST等价于struct Student \*

int main(void)

{

struct Student st;//等价于ST st；

PST ps=&st;

ps->sid=99;

printf("%d\n",ps->sid);

return 0;

}

CASE 3

#include<stdio.h>

typedef int ZHAGNSAN;//为int再重新多取一个名字，ZHAGNSAN等价于int

typedef struct Student

{

int sid;

char name[100];

char sex;

}\*PSTU,STU; //PSTU等价于struct Student \*,STU代表了struct Student

int main(void)

{

STU st;//相当于struct Srudent st;

PSTU ps=&st;//相当于struct Srudent \*ps=&st;

ps->sid=99;

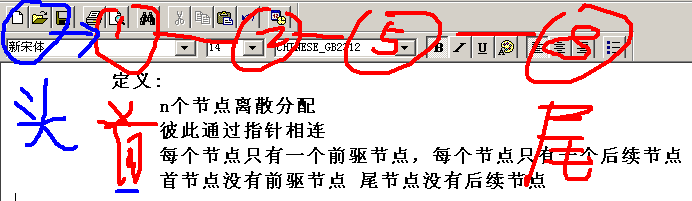
printf("%d\n",ps->sid);

return 0;

}

16\_链表的定义

定义：



n个节点离散分配；彼此通过指针相连；每个节点只有一个后续节点，首节点没有前驱节点，尾节点没有后续节点。

专业术语：

首节点：第一个有效节点

尾节点：最后一个有效节点

头节点：头节点的数据类型和首节点的数据类型相同。第一个有效节点之前的那个节点；头节点并不存放存放有效数据；加头节点的目主要是为了方便对链表的操作。

头指针：指向头节点的指针变量

尾指针：指向尾节点的指针变量

头节点-首节点。。。。。。尾节点【头节点并没有存储有效数据，也没有存放链表中有效节点的个数。首节点开始存放有效数据。在链表前边加一个没有实际意义的头节点，可以方便对链表的操作。头节点于之后节点的数据类型相同】

分类：

算法：

链表的优缺点：

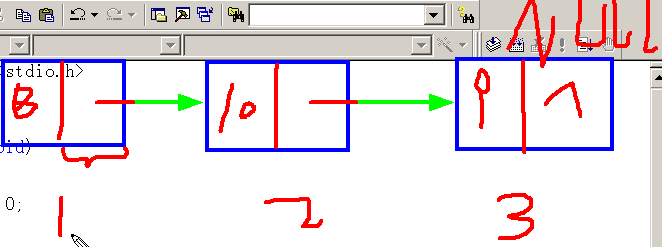
17\_如果希望通过一个函数来对链表进行处理，我们至少需要接受链表的哪些参数

如果希望通过一个函数来对链表进行处理，我们至少需要接受链表的哪些参数

只需要一个参数：头指针

因为我们通过头指针可以推算出链表的其他所有参数

18\_每一个链表节点的数据类型该如何表示的问题



CASE 1

#include<stdio.h>

typedef struct Node

{

int data;//数据域

struct Node \* pNext;//指针域

}NODE,\*PNODE;//NODE等价于struct Node，PNODE等价于struct Node \*

int main(void)

{

return 0;

}

19\_链表的分类

分类：

单链表：每个链表的指针域只能指向后面的节点

双链表：每一个节点有两个指针域

循环链表：能通过任何一个节点找到其他所有的结点

非循环链表：

20\_非循环单链表插入节点伪算法讲解

算法：

遍历

查找

清空

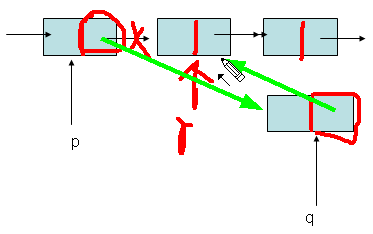
销毁

求长度

排序

删除节点

插入节点



插入算法1)r=p->pNext;p->Next=q;q->pNext=r;

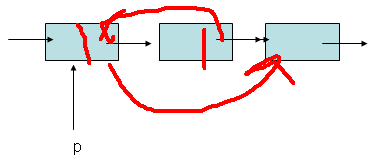
插入算法2)q->Next=p->pNext;p->Next=q;【p，q不是节点，是指针变量】

。。。。。。

21\_删除非循环单链表节点伪算法的讲解

删除

算法1（不采用）：



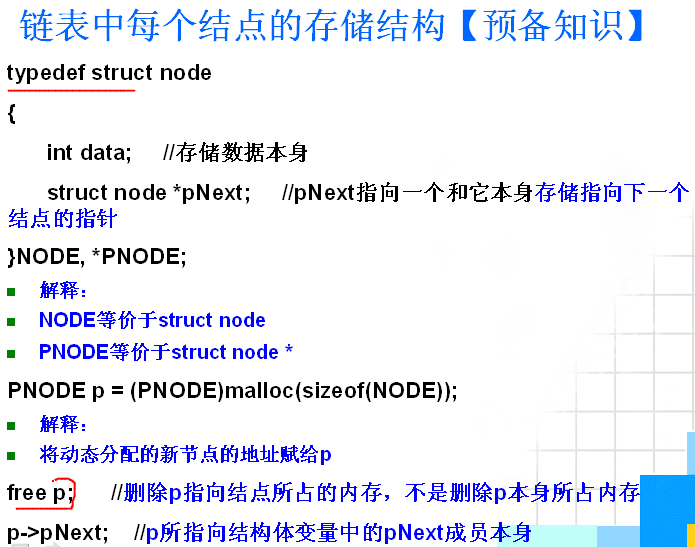
p->pNext=p->pNext->pNext;//容易导致内存泄露，没有释放内存

算法2：先临时定义一个指向p后面节点的指针r

r=p->pNext;

p->pNext=p->pNext->pNext;

free(r);



22\_学习数据结构的目的和要达到的要求

23\_复习

24\_链表创建和链表遍历算法的演示

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

typedef struct Node

{

int data;//数据域

struct Node \* pNext;//指针域

}NODE,\*PNODE;//NODE等价于struct Node，PNODE等价于struct Node \*

//函数声明

PNODE create\_list(void);

void traverse\_list(PNODE pHead);

int main(void)

{

PNODE pHead=NULL;//等价于struct Node \*pHead=NULL；

pHead=create\_list();//creat\_list()功能：创建一个非循环单链表,并将该链表的头节点的地址付给pHead

traverse\_list(pHead);

return 0;

}

PNODE create\_list(void)

{

int len;//用来存放有效节点的个数

int i;

int val;//用来临时存放用户输入的节点的值

//分配了一个不存放有效数据的头节点

PNODE pHead=(PNODE)malloc(sizeof(NODE));

if(NULL==pHead)

{

printf("分配失败，程序终止！\n");

exit(-1);

}

PNODE pTail=pHead;

pTail->pNext=NULL;

printf("请输入您需要生成的链表节点的个数：len=");

scanf("%d",&len);

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

{

printf("请输入第%d个节点的值:",i+1);

scanf("%d",&val);

PNODE pNew=(PNODE)malloc(sizeof(NODE));

if(NULL==pNew)

{

printf("分配失败，程序终止！\n");

exit(-1);

}

pNew->data=val;//挂

pTail->pNext=pNew;

pNew->pNext=NULL;

pTail=pNew;

}

return pHead;

}

void traverse\_list(PNODE pHead)

{

PNODE p=pHead->pNext;

while(NULL!=p)

{

printf("%d ",p->data);

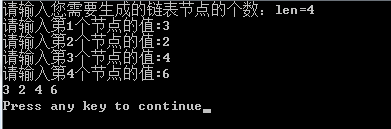
p=p->pNext;//不连续，不能用p++

}

printf("\n");

return;

}



25\_判断链表是否为空 和 求链表长度 算法的演示

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

typedef struct Node

{

int data;//数据域

struct Node \* pNext;//指针域

}NODE,\*PNODE;//NODE等价于struct Node，PNODE等价于struct Node \*

//函数声明

PNODE create\_list(void);

void traverse\_list(PNODE pHead);

bool is\_empty(PNODE pHead);

int length\_list(PNODE);

bool insert\_list(PNODE,int,int);

bool delete\_list(PNODE,int,int\*);

void sort\_list(PNODE);

int main(void)

{

PNODE pHead=NULL;//等价于struct Node \*pHead=NULL；

pHead=create\_list();//creat\_list()功能：创建一个非循环单链表,并将该链表的头节点的地址付给pHead

traverse\_list(pHead);

int len=length\_list(pHead);

printf("链表长度是%d\n",len);

if(is\_empty(pHead))

printf("链表为空！\n");

else

printf("链表不空！\n");

return 0;

}

PNODE create\_list(void)

{

int len;//用来存放有效节点的个数

int i;

int val;//用来临时存放用户输入的节点的值

//分配了一个不存放有效数据的头节点

PNODE pHead=(PNODE)malloc(sizeof(NODE));

if(NULL==pHead)

{

printf("分配失败，程序终止！\n");

exit(-1);

}

PNODE pTail=pHead;

pTail->pNext=NULL;

printf("请输入您需要生成的链表节点的个数：len=");

scanf("%d",&len);

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

{

printf("请输入第%d个节点的值:",i+1);

scanf("%d",&val);

PNODE pNew=(PNODE)malloc(sizeof(NODE));

if(NULL==pNew)

{

printf("分配失败，程序终止！\n");

exit(-1);

}

pNew->data=val;//挂

pTail->pNext=pNew;

pNew->pNext=NULL;

pTail=pNew;

}

return pHead;

}

void traverse\_list(PNODE pHead)

{

PNODE p=pHead->pNext;

while(NULL!=p)

{

printf("%d ",p->data);

p=p->pNext;//不连续，不能用p++

}

printf("\n");

return;

}

bool is\_empty(PNODE pHead)

{

if(pHead->pNext==NULL)

return true;

else

return false;

}

int length\_list(PNODE pHead)

{

PNODE p=pHead->pNext;

int len=0;

while(NULL!=p)

{

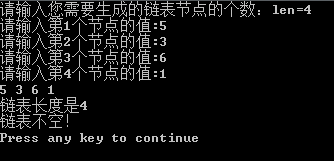
len++;

p=p->pNext;

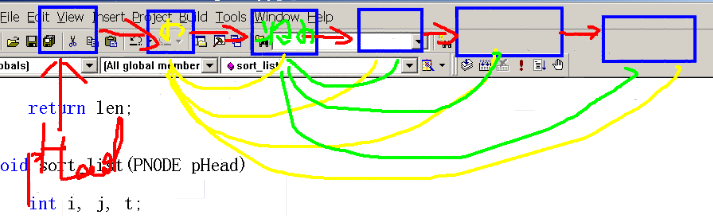
}

return len;

}



26\_通过链表排序算法的演示 再次详细讨论到底什么是算法以及到底什么是泛型【重点】



**算法：**

**狭义的算法是与数据的存储方式密切相关**

**广义的算法是与数据的存储方式无关**

**泛型：**

**利用某种技术达到的效果就是：不同的存储方式，执行的操作是一样的**

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

typedef struct Node

{

int data;//数据域

struct Node \* pNext;//指针域

}NODE,\*PNODE;//NODE等价于struct Node，PNODE等价于struct Node \*

//函数声明

PNODE create\_list(void);

void traverse\_list(PNODE pHead);

bool is\_empty(PNODE pHead);

int length\_list(PNODE);

bool insert\_list(PNODE,int,int);

bool delete\_list(PNODE,int,int\*);

void sort\_list(PNODE);

int main(void)

{

PNODE pHead=NULL;//等价于struct Node \*pHead=NULL；

pHead=create\_list();//creat\_list()功能：创建一个非循环单链表,并将该链表的头节点的地址付给pHead

traverse\_list(pHead);

int len=length\_list(pHead);

printf("链表长度是%d\n",len);

if(is\_empty(pHead))

printf("链表为空！\n");

else

printf("链表不空！\n");

sort\_list(pHead);

traverse\_list(pHead);

return 0;

}

PNODE create\_list(void)

{

int len;//用来存放有效节点的个数

int i;

int val;//用来临时存放用户输入的节点的值

//分配了一个不存放有效数据的头节点

PNODE pHead=(PNODE)malloc(sizeof(NODE));

if(NULL==pHead)

{

printf("分配失败，程序终止！\n");

exit(-1);

}

PNODE pTail=pHead;

pTail->pNext=NULL;

printf("请输入您需要生成的链表节点的个数：len=");

scanf("%d",&len);

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

{

printf("请输入第%d个节点的值:",i+1);

scanf("%d",&val);

PNODE pNew=(PNODE)malloc(sizeof(NODE));

if(NULL==pNew)

{

printf("分配失败，程序终止！\n");

exit(-1);

}

pNew->data=val;//挂

pTail->pNext=pNew;

pNew->pNext=NULL;

pTail=pNew;

}

return pHead;

}

void traverse\_list(PNODE pHead)

{

PNODE p=pHead->pNext;

while(NULL!=p)

{

printf("%d ",p->data);

p=p->pNext;//不连续，不能用p++

}

printf("\n");

return;

}

bool is\_empty(PNODE pHead)

{

if(pHead->pNext==NULL)

return true;

else

return false;

}

int length\_list(PNODE pHead)

{

PNODE p=pHead->pNext;

int len=0;

while(NULL!=p)

{

len++;

p=p->pNext;

}

return len;

}

void sort\_list(PNODE pHead)

{

int i,j,t;

int len=length\_list(pHead);

PNODE p,q;

for(i=0,p=pHead->pNext;i<len-1;++i,p=p->pNext)

{

for(j=i+1,q=p->pNext;j<len;++j,q=q->pNext)

{

if(p->data>q->data)//类似于数组中的：a[i]>a[j]

{

t=p->data;//类似于数组中的： t=a[i];

p->data=q->data;//类似于数组中的： a[i]=a[j];

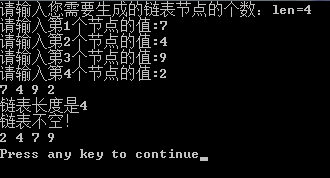
q->data=t;//类似于数组中的： a[j]=t;

}

}

}

}



27\_如何学习算法自己的一些感想

28\_链表插入和删除算法的演示

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

typedef struct Node

{

int data;//数据域

struct Node \* pNext;//指针域

}NODE,\*PNODE;//NODE等价于struct Node，PNODE等价于struct Node \*

//函数声明

PNODE create\_list(void);

void traverse\_list(PNODE pHead);

bool is\_empty(PNODE pHead);

int length\_list(PNODE);

bool insert\_list(PNODE,int,int);

bool delete\_list(PNODE,int,int\*);

void sort\_list(PNODE);

int main(void)

{

PNODE pHead=NULL;//等价于struct Node \*pHead=NULL；

int val;

pHead=create\_list();//creat\_list()功能：创建一个非循环单链表,并将该链表的头节点的地址付给pHead

traverse\_list(pHead);

int len=length\_list(pHead);

printf("链表长度是%d\n",len);

if(is\_empty(pHead))

printf("链表为空！\n");

else

printf("链表不空！\n");

sort\_list(pHead);

traverse\_list(pHead);

insert\_list(pHead,4,33);

traverse\_list(pHead);

if (delete\_list(pHead,4,&val))

{

printf("删除成功，您删除的元素是：%d\n",val);

}

else

{

printf("删除失败！您删除的元素不存在!\n");

}

traverse\_list(pHead);

return 0;

}

PNODE create\_list(void)

{

int len;//用来存放有效节点的个数

int i;

int val;//用来临时存放用户输入的节点的值

//分配了一个不存放有效数据的头节点

PNODE pHead=(PNODE)malloc(sizeof(NODE));

if(NULL==pHead)

{

printf("分配失败，程序终止！\n");

exit(-1);

}

PNODE pTail=pHead;

pTail->pNext=NULL;

printf("请输入您需要生成的链表节点的个数：len=");

scanf("%d",&len);

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

{

printf("请输入第%d个节点的值:",i+1);

scanf("%d",&val);

PNODE pNew=(PNODE)malloc(sizeof(NODE));

if(NULL==pNew)

{

printf("分配失败，程序终止！\n");

exit(-1);

}

pNew->data=val;//挂

pTail->pNext=pNew;

pNew->pNext=NULL;

pTail=pNew;

}

return pHead;

}

void traverse\_list(PNODE pHead)

{

PNODE p=pHead->pNext;

while(NULL!=p)

{

printf("%d ",p->data);

p=p->pNext;//不连续，不能用p++

}

printf("\n");

return;

}

bool is\_empty(PNODE pHead)

{

if(pHead->pNext==NULL)

return true;

else

return false;

}

int length\_list(PNODE pHead)

{

PNODE p=pHead->pNext;

int len=0;

while(NULL!=p)

{

len++;

p=p->pNext;

}

return len;

}

void sort\_list(PNODE pHead)

{

int i,j,t;

int len=length\_list(pHead);

PNODE p,q;

for(i=0,p=pHead->pNext;i<len-1;++i,p=p->pNext)

{

for(j=i+1,q=p->pNext;j<len;++j,q=q->pNext)

{

if(p->data>q->data)//类似于数组中的：a[i]>a[j]

{

t=p->data;//类似于数组中的： t=a[i];

p->data=q->data;//类似于数组中的： a[i]=a[j];

q->data=t;//类似于数组中的： a[j]=t;

}

}

}

}

//在pHead所指向链表的第pos个节点的前面插入一个新的节点，新节点的值是val，并且pos的值是从1开始

bool insert\_list(PNODE pHead,int pos,int val)

{

int i=0;

PNODE p=pHead;

while(NULL!=p&&i<pos-1)

{

p=p->pNext;

++i;

}

if(i>pos-1||NULL==p)

return false;

PNODE pNew=(PNODE)malloc(sizeof(NODE));

if(NULL==pNew)

{

printf("动态分配内存失败！\n");

exit(-1);

}

pNew->data=val;

PNODE q=p->pNext;

p->pNext=pNew;

pNew->pNext=q;

return true;

}

bool delete\_list(PNODE pHead,int pos,int\* pVal)

{

int i=0;

PNODE p=pHead;

while(NULL!=p->pNext&&i<pos-1)

{

p=p->pNext;

++i;

}

if(i>pos-1||NULL==p->pNext)

return false;

PNODE q=p->pNext;

\*pVal=q->data;

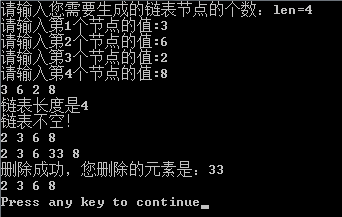
//删除p节点后面的节点

p->pNext=p->pNext->pNext;

free(q);

q=NULL;

}



29\_复习

数据结构

狭义：

数据结构是专门研究数据存储的问题

数据的存储包含两方面：个体的存储+个体关系的存储

广义：

数据结构既包括数据的存储也包括数据的操作

对存储数据的操作就是算法

算法

狭义：

算法是和数据的存储方式密切相关

广义：

算法和数据的存储方式无关

这就是泛型思想

数据的存储方式有几种

线性

连续存储【数组】

优点

存取速度很快

缺点

事先需要知道数组的长度

插入删除元素很慢

空间通常有限制

需要大块连续的内存块

离散存储【链表】

优点

空间没有限制

插入删除元素很快

缺点

存取速度很慢

线性结构的应用--栈

线性结构的应用--队列

非线性

树

图

30\_栈的定义

定义

一种可以实现“先进后出”的存储结构

栈类似于箱子

分类

算法

应用

31\_栈的分类

分类

静态栈

动态栈

32\_栈可以执行哪些操作

算法

出栈

压栈

#include<stdio.h>

#include<malloc.h>

void f(int k)

{

int m;

double \*q=(double \*)malloc(200);

}

int main(void)

{

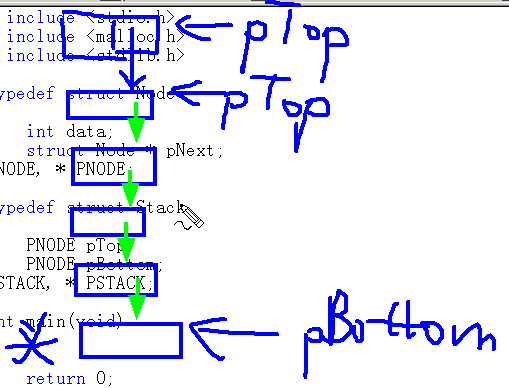
int i=10;

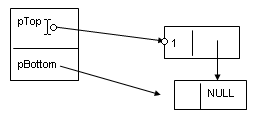
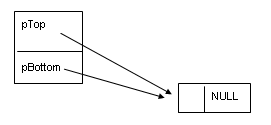
int \*p=(int \*)malloc(100);//静态的或局部变量分配在栈中分配，动态分配在堆中分配

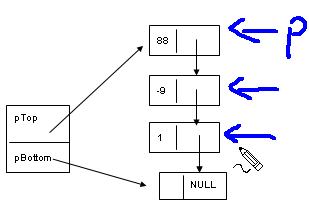
return 0;

}

33\_栈程序演示







#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

typedef struct Node

{

int data;

struct Node \*pNext;

}NODE,\*PNODE;

typedef struct Stack

{

PNODE pTop;

PNODE pBottom;

}STACK,\*PSTACK;//PSTACK等价于struct STACK\*

void init(PSTACK);

void push(PSTACK,int);

void traverse(PSTACK);

bool pop(PSTACK,int \*);

bool empty(PSTACK);

void clear(PSTACK);

int main(void)

{

STACK S;//STACK等价于struct Stack

int val;

init(&S);//目的是造出一个空栈

push(&S,1);

push(&S,2);

push(&S,3);

push(&S,4);

push(&S,5);

push(&S,6);

traverse(&S);//目的是便于输出

if(pop(&S,&val))

{

printf("出栈成功，出栈的元素是%d\n",val);

}

else

{

printf("出栈失败！");

}

traverse(&S);//目的是便于输出

clear(&S);

traverse(&S);//目的是便于输出

return 0;

}

void init(PSTACK pS)

{

pS->pTop=(PNODE)malloc(sizeof(NODE));

if(NULL==pS->pTop)

{

printf("动态内存分配失败！\n");

exit(-1);

}

else

{

pS->pBottom=pS->pTop;

pS->pTop->pNext=NULL;//pS->pBottom->pNext=NULL;

}

}

void push(PSTACK pS,int val)

{

PNODE pNew=(PNODE)malloc(sizeof(NODE));

pNew->data=val;

pNew->pNext=pS->pTop; // pS->pTop不能改成pS->pBottom

pS->pTop=pNew;

}

void traverse(PSTACK pS)

{

PNODE p=pS->pTop;

while(p!=pS->pBottom)

{

printf("%d ",p->data);

p=p->pNext;

}

printf("\n");

return;

}

bool empty(PSTACK pS)

{

if(pS->pTop==pS->pBottom)

{

return true;

}

else

{

return false;

}

}

//把pS所指向的栈出栈一次，并把出栈的元素存入pVal形参所指向的变量中，如果出栈失败，返回false，否则返回true

bool pop(PSTACK pS,int \* pVal)

{

if(empty(pS))//pS本身存放的就是S的地址

{

return false;

}

else

{

PNODE r=pS->pTop;

\*pVal=r->data;

pS->pTop=r->pNext;

free(r); //内存释放，否则容易造成内存泄露

r=NULL;

return true;

}

}

//clear清空

void clear(PSTACK pS)

{

if(empty(pS))//pS本身存放的就是S的地址

{

return ;

}

else

{

PNODE p=pS->pTop;

PNODE q=NULL;

while(p!=pS->pBottom)

{

q=p->pNext;

free(p);

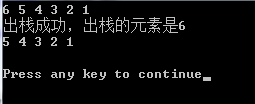
p=q;

}

pS->pTop=pS->pBottom;

}

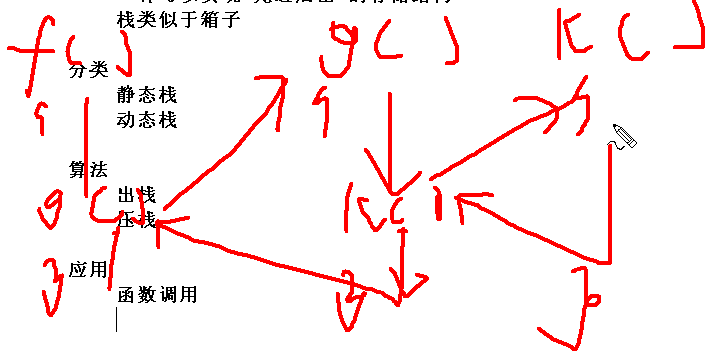
}



34\_栈的日常具体应用

应用

函数调用



中断

表达式求值

内存分配

缓冲处理

迷宫

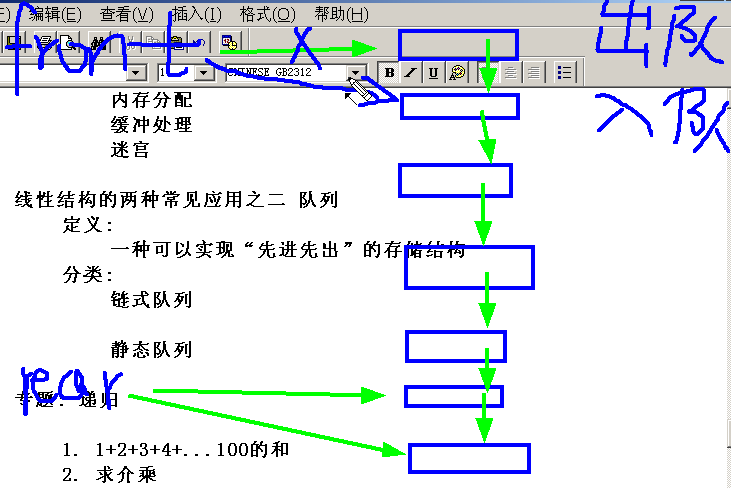
35 \_ 队列1 \_ 什么是队列

线性结构的两种常见应用之二 队列

定义：

一种可以实现“先进先出”的存储结构

36 \_ 队列2 \_ 队列的分类 和 链式队列伪算法的讲解



分类：

链式队列：用链表实现

静态队列：用数组实现

37 \_ 队列3 \_ 学习循环队列必须要弄清楚的7个问题概述

静态队列

静态队列通常都必须是循环队列

循环队列的讲解：

1）静态队列为什么必须是循环队列

2）循环队列需要几个参数来确定

3）循环队列各个参数的含义

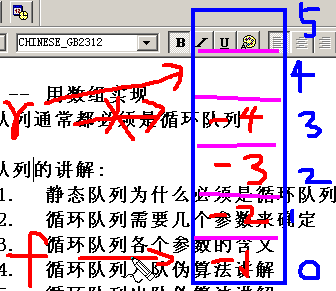
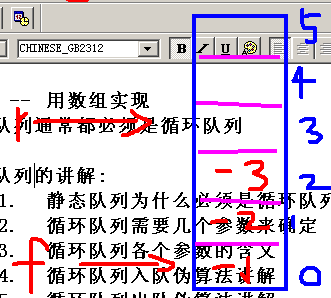
4）循环队列入队伪算法讲解

5）循环队列出队伪算法讲解

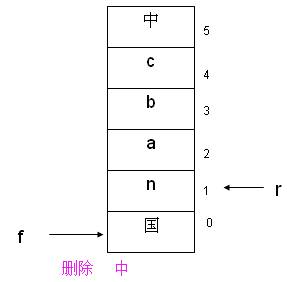
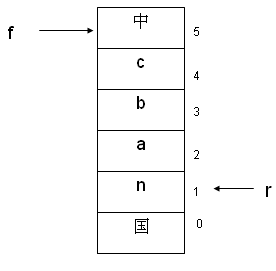
6）如何判断循环队列是否为空

7）如何判断循环队列是否已满

38 \_ 队列4 \_ 静态队列为什么必须是循环队列



如果要删除元素（出队），f只能加；如果要增加元素（入队），r只能加。（按照一般数组的方法）。r指向当前队列的下一个位置。



39 \_ 队列5 \_ 循环队列需要几个参数来确定 及其含义的讲解

需要2个参数来确定

front

rear

两个参数不同场合有不同的含义

建议初学者先记住，然后慢慢体会

1）队列初始化

front和rear的值都是零

2）队列非空

front代表的是队列的第一个元素，rear代表的是队列的最后一个有效元素的下一个元素

3）队列空

front和rear的值相等，但不一定是零

40 \_ 队列6 \_ 循环队列各个参数的含义

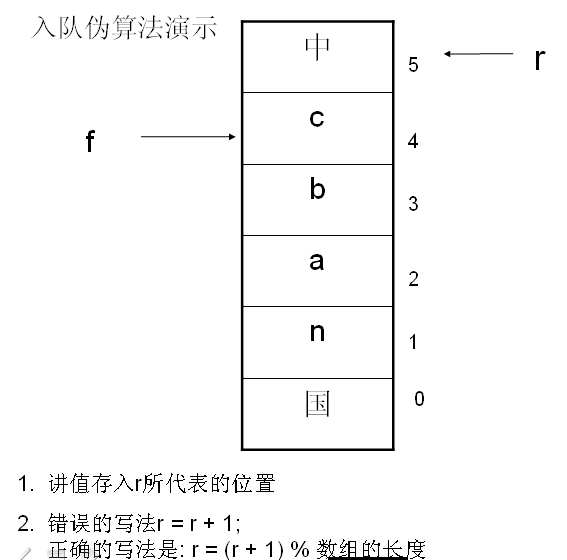
41 \_ 队列7 \_ 循环队列入队伪算法讲解

两步完成：

1.将值存入r所代表的位置

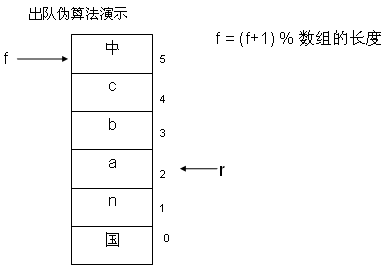
2.错误的写法：r=r+1;

正确写法是：r=（r+1）%数组的长度



42 \_ 队列8 \_ 循环队列出队伪算法讲解

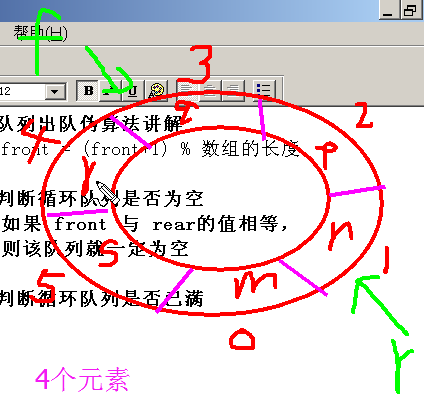
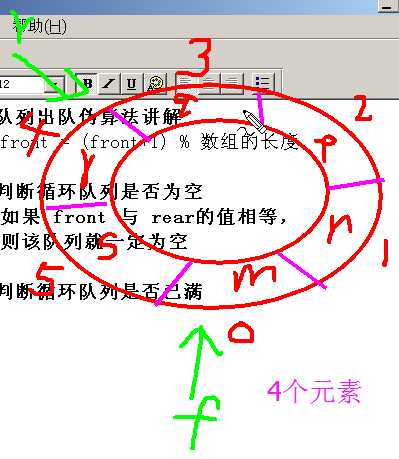
f=（f+1）%数组的长度

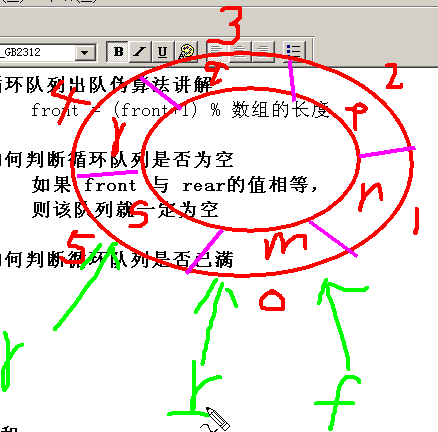
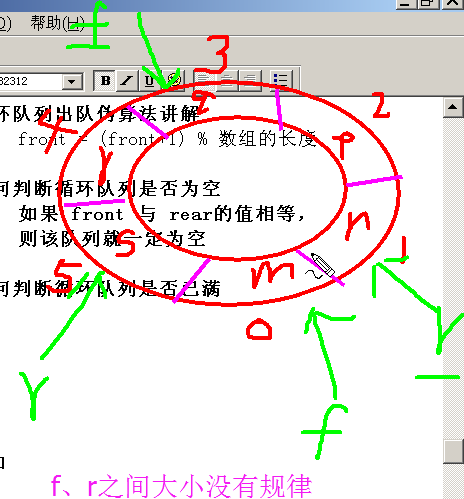


43 \_ 队列9 \_ 如何判断循环队列是否为空

如果front于rear的值相等，则该队列就一定为空。

44 \_ 队列10 \_ 如何判断循环队列是否已满





若f、r相等，不知道队列到底是空还是满。

预备知识：

front的值可能比rear大，也可能比rear小，也可能两者相等。

如何判断循环队列是否已满

两种方式：

1.多增加一个标志参数

2.少用一个元素【通常使用第二种方式】

如何判断队列已满：如果f和r的值紧挨着，则队列已满

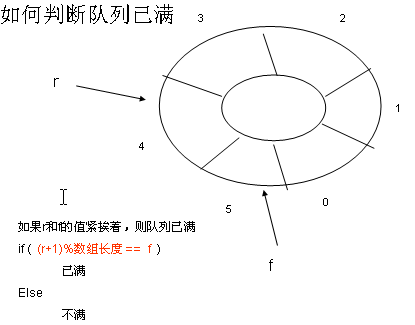
用C语言伪算法表示就是：

if（（r+1）%数组长度==f）

已满

else

不满



45 \_ 复习 \_ 求链表的长度

46 \_ 复习上节课队列知识

队列

定义：一种可以实现先进先出的存储结构

分类：

静态队列

链式队列

47 \_ 循环队列程序演示

队列算法：

入队

出队

#include<stdio.h>

#include<malloc.h>

typedef struct Queue

{

int \*pBase;

int front;

int rear;

}QUEUE;

void init(QUEUE \*);

bool en\_queue(QUEUE \*,int);//入队

void traverse\_queue(QUEUE \*);

bool full\_queue(QUEUE \*);

bool out\_queue(QUEUE \*,int \*);

bool empty\_queue(QUEUE \*);

int main(void)

{

QUEUE Q;

int val;

init(&Q);

en\_queue(&Q,1);

en\_queue(&Q,2);

en\_queue(&Q,3);

en\_queue(&Q,4);

en\_queue(&Q,5);

en\_queue(&Q,6);

en\_queue(&Q,7);

traverse\_queue(&Q);

if(out\_queue(&Q,&val))

{

printf("出队成功，队列出队的元素是%d\n",val);

}

else

{

printf("出队失败！\n");

}

traverse\_queue(&Q);

return 0;

}

void init(QUEUE \*pQ)

{

pQ->pBase=(int \*)malloc(sizeof(int)\*6);//6个元素

pQ->front=0;

pQ->rear=0;

}

bool full\_queue(QUEUE \* pQ)

{

if((pQ->rear+1)%6==pQ->front)

return true;

else

return false;

}

bool en\_queue(QUEUE \* pQ,int val)

{

if(full\_queue(pQ))

{

return false;

}

else

{

pQ->pBase[pQ->rear]=val;

pQ->rear=(pQ->rear+1)%6;

return true;

}

}

void traverse\_queue(QUEUE \* pQ)

{

int i=pQ->front;

while(i!=pQ->rear)

{

printf("%d ",pQ->pBase[i]);

i=(i+1)%6;

}

printf("\n");

}

bool empty\_queue(QUEUE \* pQ)

{

if(pQ->front==pQ->rear)

return true;

else

return false;

}

bool out\_queue(QUEUE \* pQ,int \* pVal)

{

if(empty\_queue(pQ))

{

return false;

}

else

{

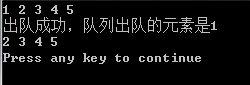
\*pVal=pQ->pBase[pQ->front];

pQ->front=(pQ->front+1)%6;

return true;

}

}



48 \_ 队列的具体应用

所有和时间有关的操作都有队列的影子

49 \_ 可以不看

50 \_ 递归1 \_ 递归的定义 和 不同函数之间相互调 程序举例

定义：

一个函数自己直接或间接调用自己

举例：

1,1+2+3+4+..+100的和

2,求阶乘

3,汉诺塔

4,走迷宫

#include<stdio.h>

void f();

void g();

void k();

int main(void)

{

f();

return 0;

}

void f()

{

printf("FFFF\n");

g();

printf("1111\n");

}

void g()

{

printf("GGGG\n");

k();

printf("2222\n");

}

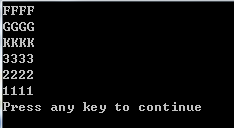
void k()

{

printf("KKKK\n");

printf("3333\n");

}



51 \_ 递归2 \_ 一个函数自己调自己 程序举例

#include<stdio.h>

void f(int n)

{

if(n==1)

printf("自己调自己\n");

else

f(n-1);

}

int main(void)

{

f(7);

return 0;

}



52 \_ 递归3 \_ 1+2+3+....+100之和用递归来实现

CASE 1

#include<stdio.h>

int main(void)

{

int val;

int i,mult=1,s;

printf("请输入一个数字：");

printf("val=");

scanf("%d",&val);

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

mult=mult\*i;

printf("%d的阶乘是：%d\n",val,mult);

return 0;

}



CASE 2

#include<stdio.h>

//假定n的值是大于或等于1的值

long f(long n)

{

if(1==n)

return 1;

else

return f(n-1)\*n;

}

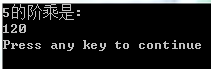
int main(void)

{

printf("5的阶乘是:\n%d\n",f(5));

return 0;

}



CASE 3

#include<stdio.h>

//假定n的值是大于或等于1的值

long sum(long n)

{

if(1==n)

return 1;

else

return sum(n-1)+n;

}

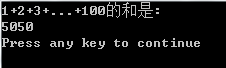
int main(void)

{

printf("1+2+3+...+100的和是:\n%d\n",sum(100));

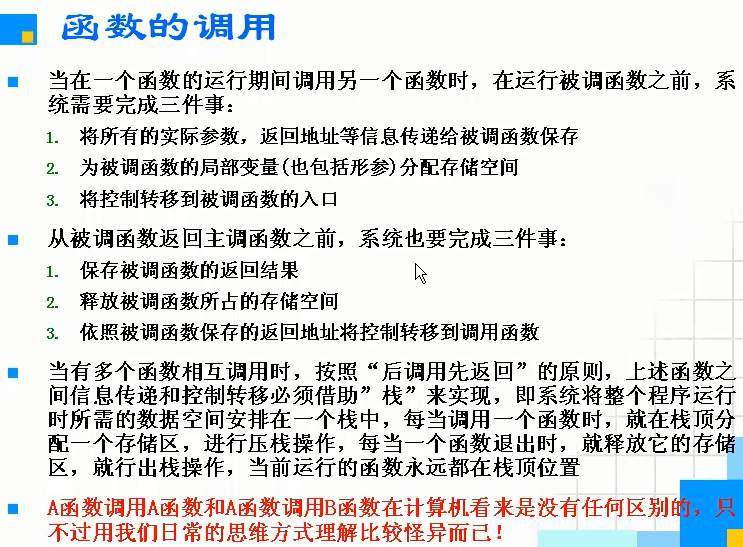
return 0;

}



53 \_ 递归4 \_ 布置作业\_汉诺塔

54 \_ 递归5 \_ 一个函数为什么可以自己调用自己



CASE 1

#include<stdio.h>

//间接调用自己举例

void f(int );

void g(int);

void f(int n)

{

g(n);

}

void g(int m)

{

f(m);

}

int main(void)

{

return 0;

}

CASE 2

#include<stdio.h>

int f(int n)

{

n+=2;//n=n+2;

return n;

}

int main(void)

{

int val;

val=f(5);

printf("val=%d\n",val);

return 0;

}



CASE 3

#include<stdio.h>

//A函数调用B函数举例

int g(int);

int f(int n)

{

n=g(n);

return n;

}

int g(int m)

{

m=m\*2;

return m;

}

int main(void)

{

int val;

val=f(5);

printf("val=%d\n",val);

return 0;

}



55 \_ 递归6 \_ 递归必须满足三个条件

递归满足三个条件：

1.递归必须得有一个明确的终止条件

2.该函数所处理的数据规模必须在递减

3.这个转化必须是可解的

CASE 1

#include<stdio.h>

//递增的值可以增加，但处理的数据的规模需要递减

int f(int n)

{

if(n>7)

printf("哈哈\n");

else

n=f(n+1);

return n;

}

int main(void)

{

int val;

val=f(5);

printf("val=%d\n",val);

return 0;

}



循环和递归的关系：

理论上，所有的循环都可以转化成递归，但是用递归能解决的问题不一定能用循环解决。

56 \_ 递归7 \_ 循环和递归的比较

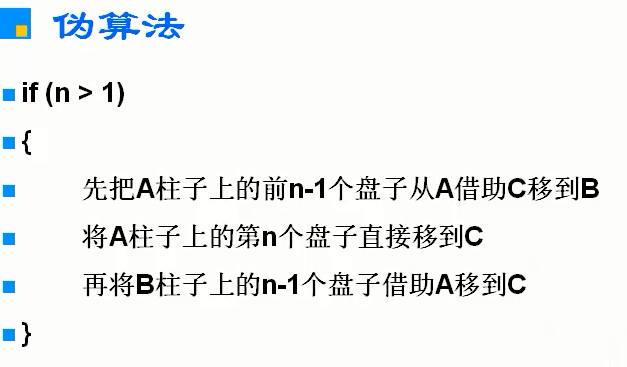
递归：

易于理解，速度慢，存储空间大

循环：

不易理解，速度快，存储空间小

57 \_ 递归8 \_ 汉诺塔



#include<stdio.h>

//汉诺塔

void hannuota(int n,char A,char B,char C)

{

/\*

如果是一个盘子

直接将A柱子上的盘子从A移到C

否则

先将A柱子上的n-1个盘子借助C移到B

直接将A柱子上的盘子从A移到C

最后将B柱子上的n-1个盘子借助A移到C

\*/

if(1==n)

{

printf("将编号为%d的盘子直接从%c柱子移到%c柱子\n",n,A,C);

}

else

{

hannuota(n-1,A,C,B);

printf("将编号为%d的盘子直接从%c柱子移到%c柱子\n",n,A,C);

hannuota(n-1,B,A,C);

}

}

int main(void)

{

char ch1='A';

char ch2='B';

char ch3='C';

int n;

printf("请输入要移动盘子的个数 :");

scanf("%d",&n);

hannuota(n,'A','B','C');

return 0;

}



58 \_ 递归9 \_ 递归的应用

举例：

递归的应用

59\_1线性结构总复习 2线性结构和非线性结构关系 3栈队列链表数组之间的关系【重点】

逻辑结构：

线性：

数组

链表

栈和队列是一种特殊的线性结构

非线性：

树

图

物理结构：

60\_树1\_树的定义

模块二：非线性结构

树

树定义

树分类

树存储

树操作

树应用

树定义：

专业定义：有且只有一个称为根的节点；有若干个互不相交的子树，这些树本身也是一棵树

通俗的定义：树是由节点和边组成

61\_树2\_树的专业术语解释

节点、父节点、子节点、子孙、堂兄弟

深度：从根节点到最底层节点的层数称之为深度，根节点是第一层

叶子节点：没有子节点的节点

非终端节点：实际就是非叶子节点

度；子节点的个数

62\_树3\_树的分类

一般树：任意一个节点的子节点的个数都不受限制

二叉树：任意一个节点的子节点的个数最多两个，且子节点的位置不可更改

分类：

一般二叉树

满二叉树

在不增加树的层数的前提下，无法再多添加

一个节点的二叉树就是满二叉树

完全二叉树

如果只是删除了满二叉树最底层最右边的连续若干个点，这样形成的二叉树就是完全二叉树。

森林：n个互不相交的树的集合

63\_树4\_二叉树连续存【重点】

树的存储

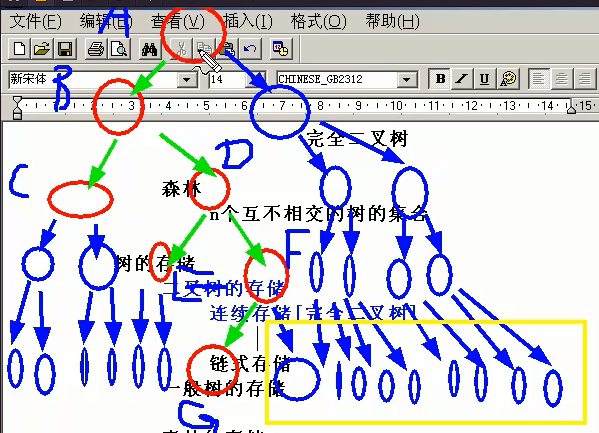
二叉树的存储

连续存储【完全二叉树】

链式存储

一般树的存储

森林的存储

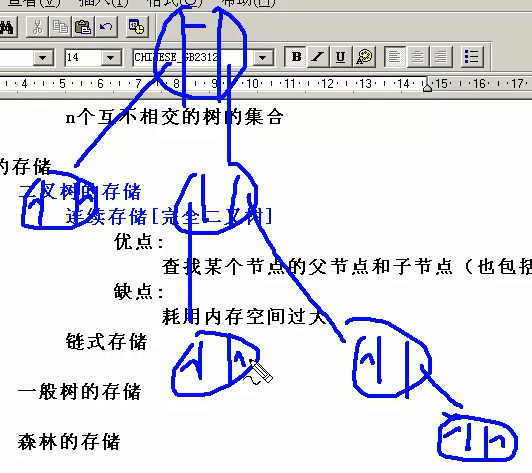


连续存储【完全二叉树】

优点：查找某个节点的父节点和子节点（也包括有没有子节点）速度很快

缺点：耗用内存空间过大

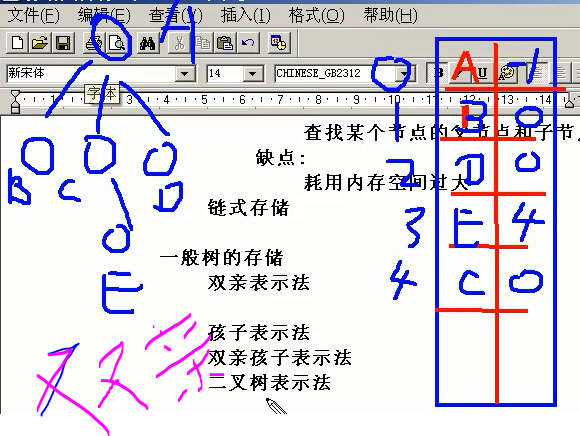
64\_树5\_二叉树的链式存储



65\_树6\_普通树的存储

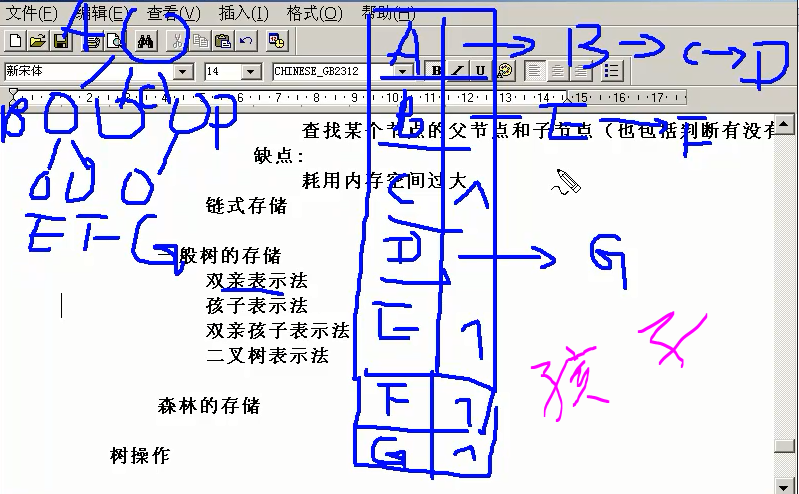
双亲表示法

求父节点方便



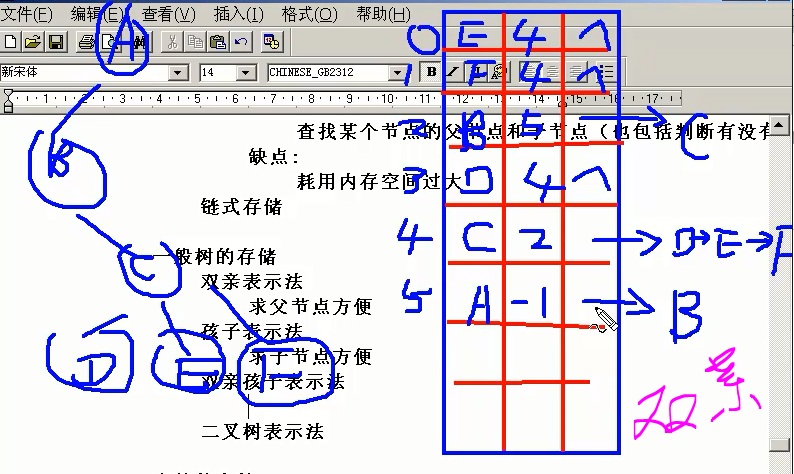
孩子表示法

求子节点方便



双亲孩子表示法

求父节点和子节点都很方便

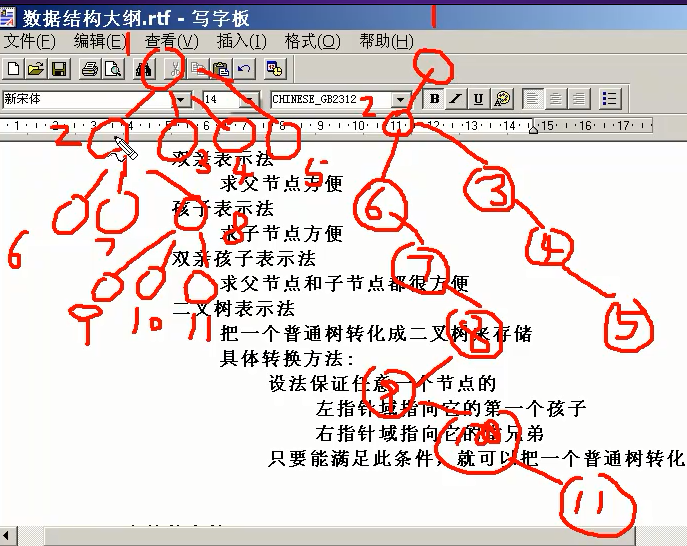


二叉树表示法

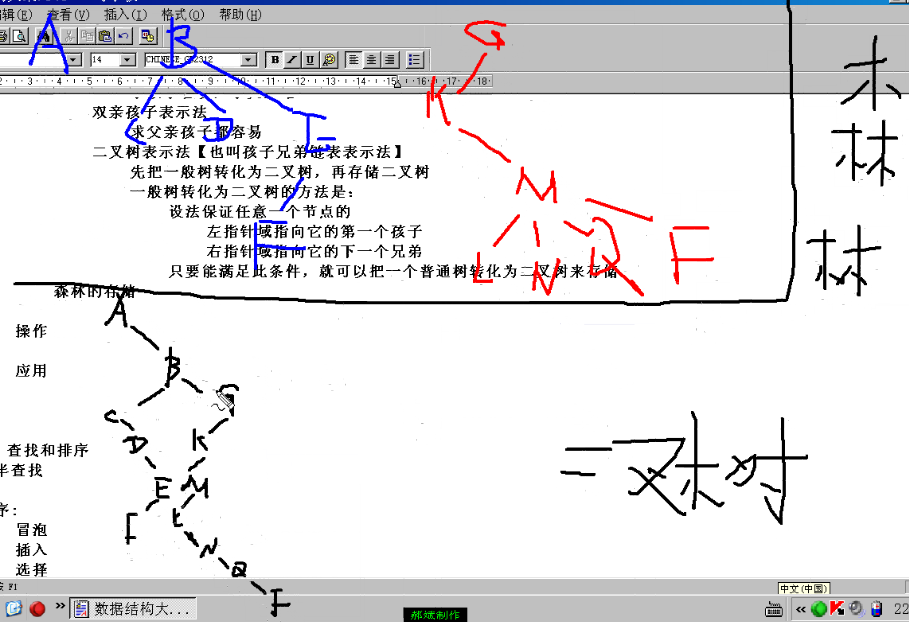
把一个普通树转化成二叉树来存储

具体转换方法：设法保证任意一个节点的左指针域指向它的第一个孩子，右指针域指向它的下一个兄弟。只要能满足此条件，就可以把一个普通的胡转换成二叉树。

一个普通树转换成的二叉树移动没有右子树



66\_树7\_森林的存储



67\_树8\_二叉树的先序遍历

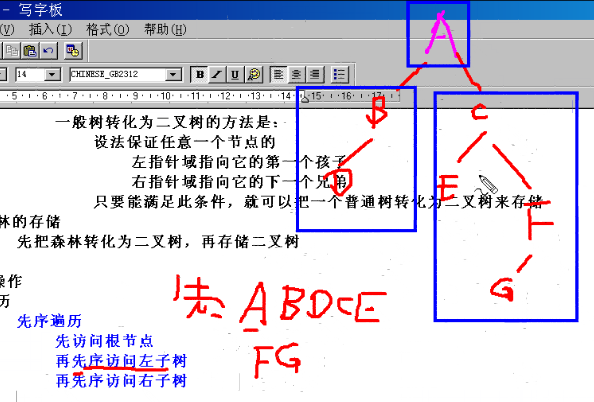
二叉树操作

1.遍历：先序遍历，中序遍历，后序遍历

2.已知两种遍历序列求原始二叉树

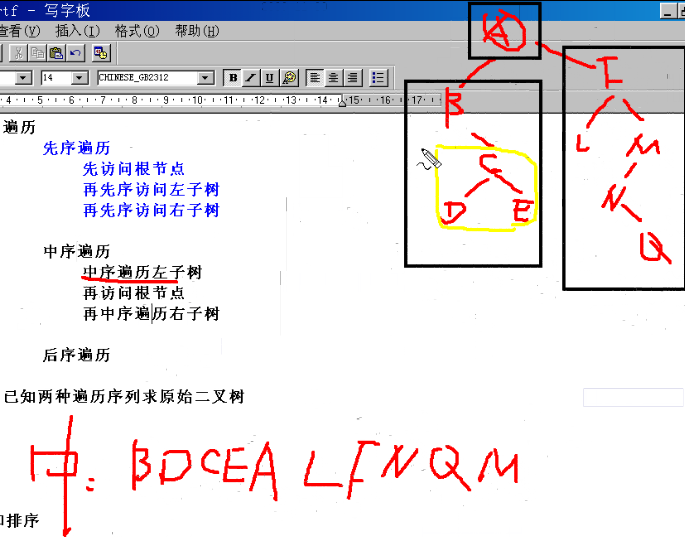
先序遍历：【先访问根节点】

先访问根节点，再先序访问左子树，再先序访问右子树。



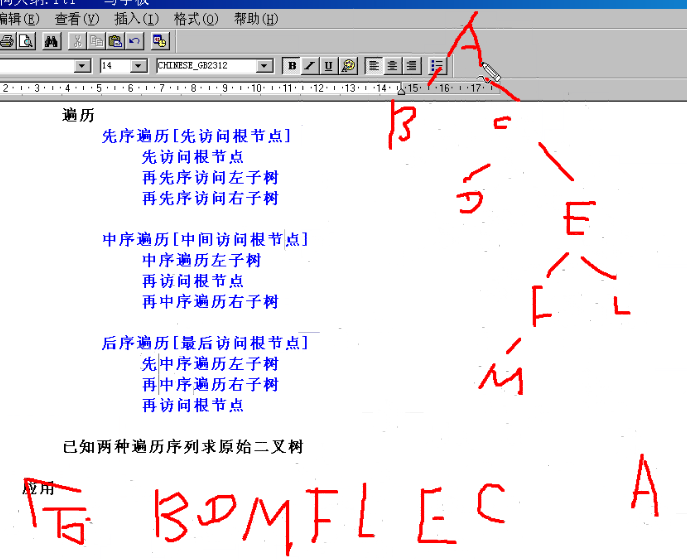
68\_树9\_二叉树的中序遍历【中间访问根节点】

中序遍历左子树，再访问根节点，再中序遍历右子树



69\_树10\_二叉树的后序遍历【最后访问根节点】

序遍历左子树，后序遍历右子树后序遍历根节点

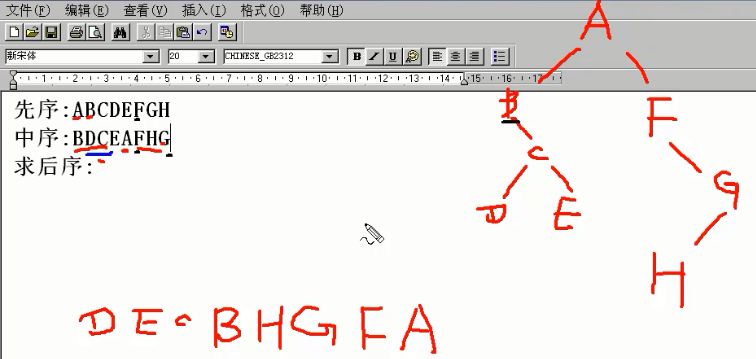


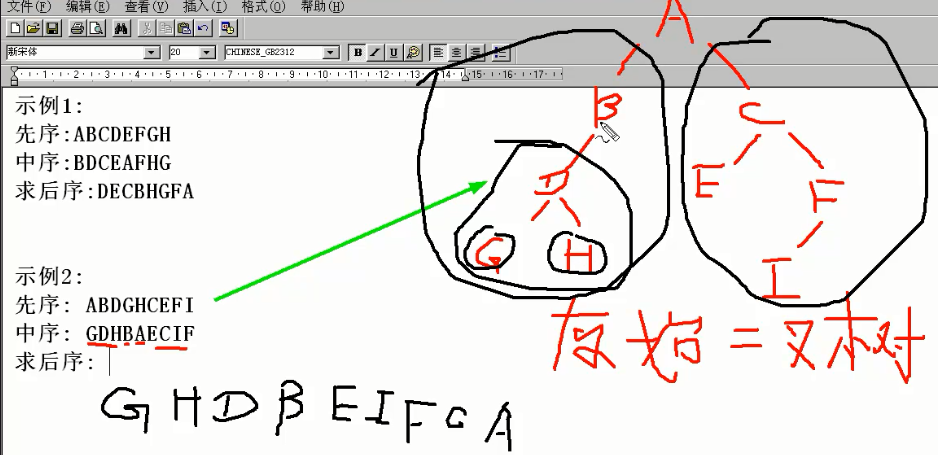
70\_树11\_已知两种遍历序列求原始二叉树概述

通过先序和中序 或者 中序和后序我们可以还原出原始的二叉树，但是通过先序和后序是无法还原出原始的二叉树的。

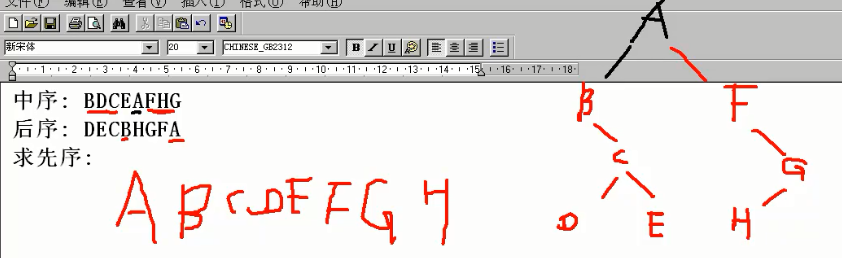
换种说法，只有通过先序和中序，或通过中序和后序，我们才能唯一的确定一个二叉树。

71\_树12\_已知先序和中序求后序





72\_树13\_已知中序和后序求先序



73\_树14\_树的应用简单介绍

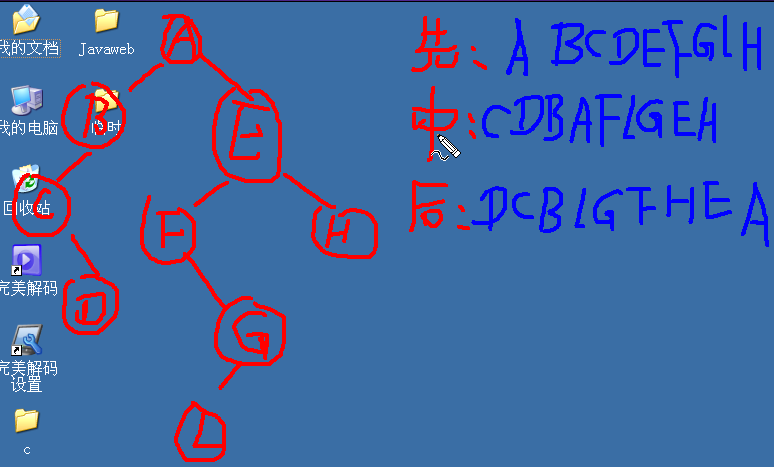
树是数据库数据组织的一种重要形式。

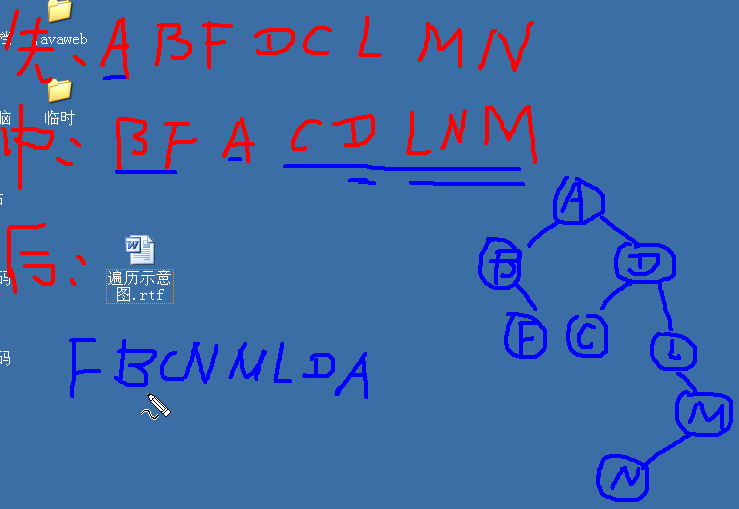
操作系统子父进程的关系本身就是一棵树。

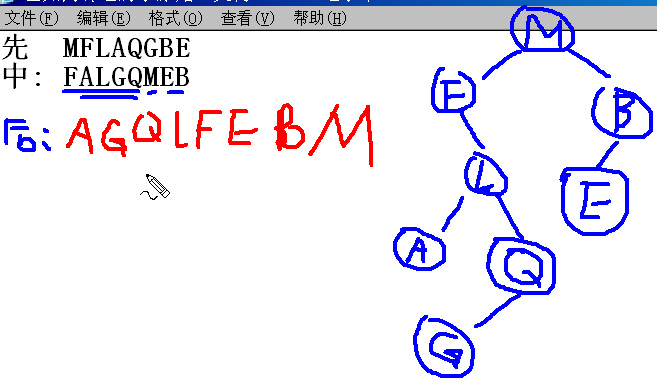
面向对象语言中类的继承关系本身就是一棵树。

赫夫曼树

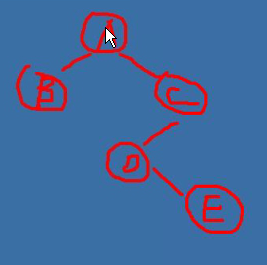
74\_树15\_复习上节课知识







75\_树16\_链式二叉树遍历具体程序演示



//程序执行有问题

#include<stdio.h>

#include<malloc.h>

struct BTNode

{

int data;

struct BTNode \*pLchhid;//p是指针，L是左，child是孩子

struct BTNode \*pRchhid;

};

void PreTraverseBTree( struct BTNode \* pT);

void InTraverseBTree( struct BTNode \* pT);

void PostTraverseBTree( struct BTNode \* pT);

struct BTNode \* CreateBTree(void);

int main(void)

{

struct BTNode \* pT=CreateBTree();

printf("前序遍历：\n");

PreTraverseBTree(pT);

printf("中序遍历：\n");

InTraverseBTree(pT);

printf("后序遍历：\n");

PostTraverseBTree(pT);

return 0;

}

struct BTNode \* CreateBTree(void)

{

struct BTNode \* pA=(struct BTNode \*)malloc(sizeof(struct BTNode ));

struct BTNode \* pB=(struct BTNode \*)malloc(sizeof(struct BTNode ));

struct BTNode \* pC=(struct BTNode \*)malloc(sizeof(struct BTNode ));

struct BTNode \* pD=(struct BTNode \*)malloc(sizeof(struct BTNode ));

struct BTNode \* pE=(struct BTNode \*)malloc(sizeof(struct BTNode ));

pA->data='A';

pB->data='B';

pC->data='C';

pD->data='D';

pE->data='E';

pA->pLchhid=pB;

pA->pRchhid=pC;

pB->pLchhid=pB->pRchhid=NULL;

pC->pLchhid=pD;

pC->pRchhid=NULL;

pD->pLchhid=NULL;

pD->pRchhid=pE;

pE->pLchhid=pE->pRchhid=NULL;

return pA;

}

void PreTraverseBTree( struct BTNode \* pT)

{

if(pT!=NULL)

{

printf("%c\n",pT->data);

if(NULL!=pT->pLchhid)

{

PreTraverseBTree(pT->pLchhid);

}

if(NULL!=pT->pLchhid)

{

PreTraverseBTree(pT->pRchhid);

}

//pT->pLchhid可以代表整个左子树

}

}

void InTraverseBTree( struct BTNode \* pT)

{

if(pT!=NULL)

{

if(NULL!=pT->pLchhid)

{

InTraverseBTree(pT->pLchhid);

}

printf("%c\n",pT->data);

if(NULL!=pT->pLchhid)

{

InTraverseBTree(pT->pRchhid);

}

//pT->pLchhid可以代表整个左子树

}

}

void PostTraverseBTree( struct BTNode \* pT)

{

if(pT!=NULL)

{

if(NULL!=pT->pLchhid)

{

PostTraverseBTree(pT->pLchhid);

}

if(NULL!=pT->pLchhid)

{

PostTraverseBTree(pT->pRchhid);

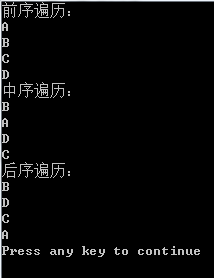
}

printf("%c\n",pT->data);

//pT->pLchhid可以代表整个左子树

}

}



76\_树17\_5种常用排序概述 和 快速排序详细讲解

排序：

冒泡

插入

选择

快速排序

归并排序

排序和查找的关系：

排序是查找的前提，排序是重点

77\_树18\_再次讨论什么是数据结构

研究的是数据的存储和数据的操作的一门学问。

数据的存储分为两类：

分体的存储

个体关系的存储

从某个角度而言，数据的最核心的就是个体关系的存储，个体的存储可以忽略不计。

78\_树19\_再次讨论到底什么是泛型

同一种逻辑结构，无论该逻辑结构物理存储是什么样子的，我们可以对它执行相同的操作。