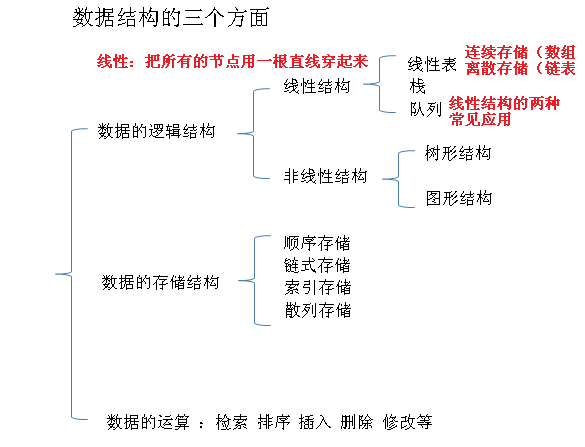
**优化版郝斌数据结构自学笔记**

**by wangdehua** [**wanggong\_mail@qq.com**](mailto:wanggong_mail@qq.com)

**对郝斌老师的代码进行了优化，显得更加专业**

# 数据结构定义



# 连续存储-->数组

数组的增删改查程序

**#include** <stdio.h>

**#include** <stdlib.h>

**#define** N 100

**typedef** **int** datatype; // typedef有分号！！

//这样如果想定义char型数组，就把int改成char就行了

**typedef** **struct**

{

datatype data[N];

**int** last; //last存放[最后]一个元素的下标

} sqlist; //定义了sqlist的结构体类型

sqlist \* **list\_create**();

**int** **list\_insert**(sqlist \*L, datatype value);

**int** **list\_insert\_pos**(sqlist \*L, datatype value, **int** pos);

**int** **list\_full**(sqlist \*L);

// 1-------满 0-------不满

**int** **list\_empty**(sqlist \*L);

// 1-------空 0-------非空

**int** **list\_delete**(sqlist \*L);

**int** **list\_delete\_pos**(sqlist \*L, **int** pos);

**void** **list\_show**(sqlist \*L);

**int** **list\_replace**(sqlist \*L, datatype value, **int** pos);

**int** **list\_replace\_value**(sqlist \*L, datatype old, datatype new);

**int** **list\_search**(sqlist \*L, datatype value);

//找到-----1 找不到------0

**void** **list\_purge**(sqlist \*L);

**void** **list\_union**(sqlist \*La, sqlist \*Lb);

**int** **main**(**void**)

{

sqlist \*L = NULL;

sqlist \*La = NULL;

sqlist \*Lb = NULL;

**int** search;

**if** (NULL == (L = list\_create()))

{//create函数失败的返回值是NULL

//这里将L指向动态存储区,注意(L = list\_create())要打上括号，否则

//"=="的优先级高于"=",语句会出错

**printf**("list\_create failed\n");

**return** -1;

}

**if** ((La = list\_create()) == NULL) //create函数失败返回值是NULL

{//这里将La指向动态存储区

**printf**("list\_create failed\n");

**return** -1;

}

**if** ((Lb = list\_create()) == NULL) //create函数失败返回值是NULL

{//这里将Lb指向动态存储区

**printf**("list\_create failed\n");

**return** -1;

}

list\_insert(L, 5);

list\_insert(L, 6);

list\_insert(L, 1);

list\_insert(L, 4);

list\_insert(L, 8);

list\_insert(L, 8);

list\_insert(L, 9);

list\_insert(L, 9);

list\_show(L);

list\_insert\_pos(L, 111, 0);

list\_insert\_pos(L, 222, 5);

list\_insert\_pos(L, 333, 20);

list\_show(L);

list\_delete(L);

list\_delete(L);

list\_show(L);

list\_delete\_pos(L, 5);

list\_delete\_pos(L, 31);

list\_show(L);

list\_purge(L);

list\_show(L);

list\_replace(L, 13, 2);

list\_replace(L, 14, 3);

list\_show(L);

list\_replace\_value(L, 13, 100);

list\_replace\_value(L, 1, 100);

list\_show(L);

search = list\_search(L,5);

**printf**("%d\n", search);

search = list\_search(L,999);

**printf**("%d\n", search);

list\_insert(La, 1);

list\_insert(La, 2);

list\_insert(La, 3);

list\_insert(La, 4);

list\_insert(Lb, 5);

list\_insert(Lb, 6);

list\_insert(Lb, 7);

list\_union(La, Lb);

**free**(L);

**free**(La);

**free**(Lb);

**return** 0;

}

sqlist \***list\_create**()

{

sqlist \*L = NULL;

L = **malloc**(**sizeof**(sqlist));

//malloc前可以不加类型转换了，自动转换

**if** (NULL == L)

{//申请不上那L没指向新空间还指向NULL

**printf**("malloc failed\n");

**return** NULL;//返回值也是NULL

}

L->last = -1;

//数组元素从0开始，有一个元素时last是0，没有时是-1

**return** L;

}

**int** **list\_insert**(sqlist \*L, datatype value)

{//在表的末尾插入

**if** (list\_full(L))

{

**printf**("list is full!\n");

**return** -1;

}

L->data[L->last + 1] = value;

L->last++;

**return** 0;

}

**int** **list\_full**(sqlist \*L)

{//判断表的空间是不是满了

**return** L->last == (N - 1) ? 1 : 0;

}//定义N个元素，下标就是0到N-1

**int** **list\_empty**(sqlist \*L)

{

**return** L->last == -1;

}

**void** **list\_show**(sqlist \*L)

{

**int** i;

**for** (i = 0; i <= L->last; i++)

{

**printf**("%d ", L->data[i]);

}

putchar(10);

**return**;

}

**int** **list\_insert\_pos**(sqlist \*L, datatype value, **int** pos)

{//在表的任意位置插入

**int** i;

**if** (list\_full(L))

{

**printf**("list is full!\n");

**return** -1;

}

**if** (pos < 0 || pos > L->last + 1)//要保证插入后位置是连续的

{//不是>=,可以在last也就是最后一个位置上面插入，仍然是连续的

**printf**("error pos\n");

**return** -1;

}

**for** (i = L->last; i >= pos; i--)

{//在pos处插入，要把pos处原值及以上值全部上移，空出pos位置

L->data[i + 1] = L->data[i];

}

L->data[pos] = value;//把值插在pos空位

L->last++;

**return** 0;

}

**int** **list\_delete**(sqlist \*L) //删除末尾

{

**if** (list\_empty(L))

{//empty空了返回1，if是真

**printf**("list is empty\n");

**return** -1;

}

L->last--;//多的那个数等着被覆盖就行了

**return** 0;

}

**int** **list\_delete\_pos**(sqlist \*L, **int** pos)

{//删除任意位置元素

**int** i;

**if** (list\_empty(L))

{

**printf**("list is empty\n");

**return** -1;

}

**if** (pos < 0 || pos > L->last)

{

**printf**("error pos\n");

**return** -1;

}

**for** (i = pos; i < L->last; i++)

{//i是<,因为下面是data[i+1],执行到last前一个就把

//last值给它了。因为元素数少了1个，最后last值没意义，

//加上等号多走一步当然没有错误

L->data[i] = L->data[i + 1];

}//删除pos，就把pos以上的元素依次往下覆盖就行

L->last--;

**return** 0;

}

**int** **list\_replace**(sqlist \*L, datatype value, **int** pos)

{//替换任意位置元素

**if** (list\_empty(L))

{

**printf**("list is empty\n");

**return** -1;

}

**if** (pos < 0 || pos > L->last)

{

**printf**("error pos\n");

**return** -1;

}

L->data[pos] = value;

**return** 0;

}

**int** **list\_replace\_value**(sqlist \*L, datatype old, datatype new)

{//替换某个具体数值

**int** i;

**for** (i = 0; i <= L->last; i++)

{

**if** (L->data[i] == old)

{

L->data[i] = new;

**return** 0;

}

}

**printf**("no found old\n");

**return** -1;

}

**int** **list\_search**(sqlist \*L, datatype value)

{//查找

**int** i;

**for** (i = 0; i <= L->last; i++)

{

**if** (L->data[i] == value)

{

**return** 1;

}

}

**return** 0;

}

**void** **list\_purge**(sqlist \*L)

{//清除相同的元素

**int** i, j;

**for** (i = 0; i < L->last; i++)

{//因为下面是j=i+1,所以<

**for** (j = i + 1; j <= L->last; j++)

{

**if** (L->data[i] == L->data[j])

{

list\_delete\_pos(L, j);

j--;

}

}

}

}

/\*

j--解析：如果元素是1[2]34[2]567，那么检测到两个2相同，就把后面那个2删去，

也就是567往前覆盖。没有没有j--，j++就到了6的位置，那么5被忽视了。

\*/

**void** **list\_union**(sqlist \*La, sqlist \*Lb)

{//将数组b添加到数组a的后面

**int** i;

**for** (i = 0; i <= Lb->last; i++)

{

La->data[La->last + 1] = Lb->data[i];

La->last++;

}

list\_show(La);

**printf**("L->last = %d\n", La->last);

}

/\*list\_union方法2

void list\_union(sqlist \*La,sqlist \*Lb)

{

int i;

for (i = 0;i <= Lb->last;i++)

{

La->data[La->last+1+i] = Lb->data[i];

}

La->last = La->last + Lb->last + 1;

//数组b有10个数，Lb->last为9，所以最后加1

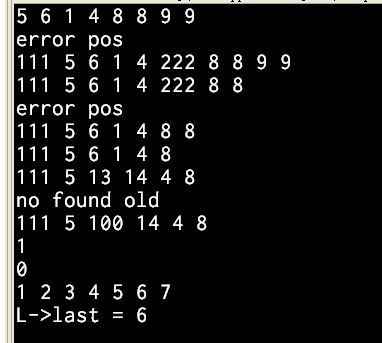
list\_show(La);

printf("La->last = %d\n",La->last);

}

\*/

运行结果:



# 离散存储->链表

1. 链表的定义

n个节点离散分配，彼此通过指针相连，每个节点只有**1个**后续节点，第一个节点没有前驱

节点，尾节点没有后续节点。

1. 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：头节点是第一个有效节点之前的那个节点，头节点并**不存放有效**数据，也没有存放

链表中有效节点的个数。加头节点的目主要是为了方便对链表的操作

**链表为空**时就是里面只有一个头结点，不是什么都没有！

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

1. 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

1. 每一个链表节点的数据类型该如何表示

typedef struct Node

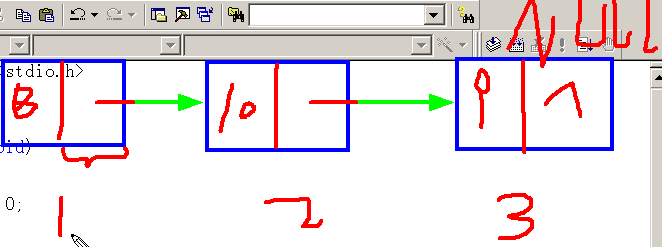
{

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 \*！！！**



1. 链表的分类

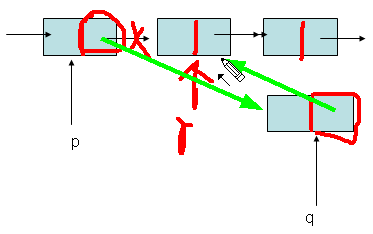
单链表：每个链表的指针域只能指向后面的节点

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：

1. 非循环单链表插入节点图解



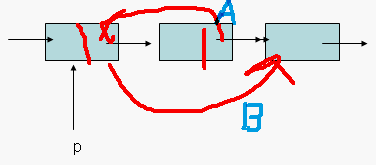
插入算法1 推荐 q->Next=p->pNext; p->Next=q;

**注意: p，q不是节点，是指针变量！这两行的顺序不能倒过来！**

插入算法2 r=p->pNext; p->Next=q; q->pNext=r;

1. 删除非循环单链表节点图解

算法1（**不采用**）：



p->pNext=p->pNext->pNext; //代码是A步骤，把第2个节点的指向给了第1个节点的指向

//经过A步骤就实现了B步骤

为什么不行:

如图，这样做了之后指向第2个节点的指针就没了，就是说我就找不到第2个节点了。这样

我就没有释放第2个节点的内存了！

算法2：算法1加一步即可，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);

1. 链表的反转(华三笔试题)

用头插法来创建链表，插入的顺序是123456，但是输出的顺序就是654321，

现在的挑战是怎么让输入和输出的顺序一致？

void list\_reverse(linklist \*H)

{

linklist \*p = H->next,\*q = NULL;

H->next = NULL;

while(p != NULL)

{

q = p;

p = p->next;

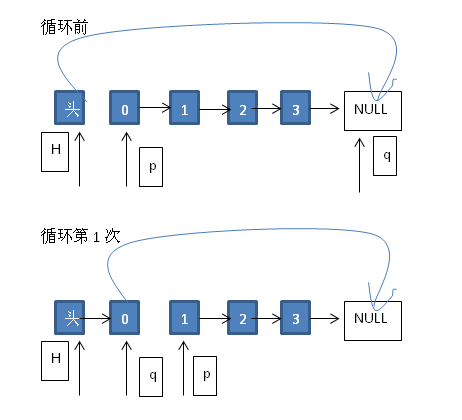
q->next = H->next;

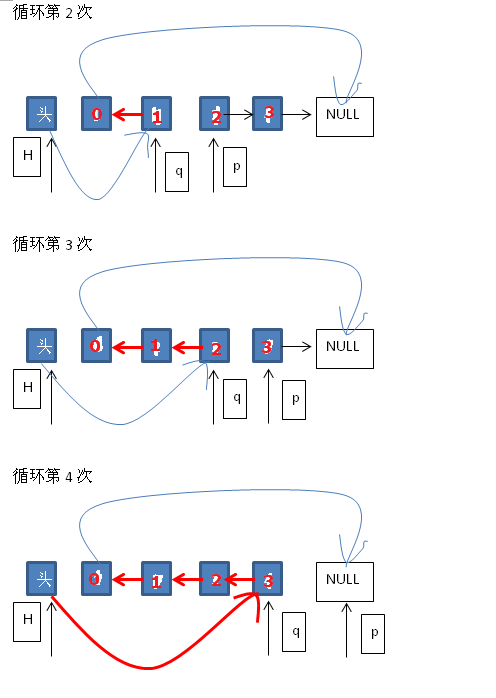
H->next = q;

}

}

解析:





1. 程序如下:

#include <stdio.h>

#include <malloc.h>

typedef int datatype;

struct node

{

datatype data;

struct node \* next;

};

typedef struct node linklist;

linklist \* list\_create();

int head\_insert(linklist \*H,datatype value);

int head\_delete(linklist \*H);

void list\_show(linklist \*H);

int list\_empty(linklist \*H);

int list\_insert\_pos(linklist \*H,datatype value,int pos);

int list\_delete\_value(linklist \*H,datatype value);

int list\_replace(linklist \*H,datatype old,datatype new);

int list\_search(linklist \*H,datatype value);

void list\_reverse(linklist \*H);

int main(void)

{

linklist \* H = NULL;

if (NULL == (H = list\_create()))

{

printf("list\_create failes\n");

return -1;

}

printf("没有反转输出如下:\n");

head\_insert(H,1);

head\_insert(H,2);

head\_insert(H,3);

head\_insert(H,4);

head\_insert(H,5);

head\_insert(H,6);

list\_show(H);

printf("头插入如下:\n");

head\_insert(H,111);

head\_insert(H,222);

list\_show(H);

printf("任意合法位置插入如下:\n");

list\_insert\_pos(H,333,0);

list\_insert\_pos(H,555,2);

list\_show(H);

printf("头删如下:\n");

head\_delete(H);

list\_show(H);

printf("删除具体值如下:\n");

list\_delete\_value(H,6);

list\_delete\_value(H,10);

list\_show(H);

printf("替换如下:\n");

list\_replace(H,5,20);

list\_show(H);

list\_replace(H,100,20);

list\_show(H);

printf("查找如下:\n");

printf("pos:%d\n",list\_search(H,555));

printf("pos:%d\n",list\_search(H,100));

printf("反转输出如下:\n");

list\_reverse(H);

list\_show(H);

return 0;

}

linklist \*list\_create()

{//(H = malloc(sizeof(linklist))别忘了打括号

//因为“==”优先级高于“=”，不打括号语句错误

linklist \* H = NULL;

if (NULL == (H = malloc(sizeof(linklist))))

{

printf("malloc failed\n");

return NULL;

}

H->next = NULL;

return H;

}

int head\_insert(linklist \*H, datatype value)

{//在头结点的后面插入

linklist \* p = NULL;

if (NULL == (p = malloc(sizeof(linklist))))

{//p指向要插入的元素的空间

printf("malloc failed\n");

return -1;

}

p->data = value;

p->next = H->next;

H->next = p;

return 0;

}

void list\_show(linklist \* H)

{

while (H->next != NULL)

{

printf("%d ", H->next->data);//注意是H->next->data!!!

//头结点不存放内容，所以是输出下一个的内容!!!

H = H->next;

//指向头结点的指针不断移向下一个，当H指向的节点中的next指针指向

//NULL时，就没有下一个内容可以输出了。H指向尾时，上一次已经把尾的

//内容输出了

}

putchar(10);

}

int list\_empty(linklist \* H)

{//1是空 0是非空

if (H->next != NULL)

{

return 0;

}

else

{

return 1;

}

}

int head\_delete(linklist \* H)

{//头删，删除头结点后面的那个，直接把头结点指向后面的后面那个

//再把头结点后面的释放

if (list\_empty(H))

{

printf("list is empty\n");

return -1;

}

linklist \* p = H->next;

H->next = p->next;

free(p);

p = NULL;//防止野指针！！

return 0;

}

int list\_insert\_pos(linklist \* H, datatype value, int pos)

{//在任意合法位置插入

/\*原理:头节点之后的首节点及以后的位置分别为0,1,2,3...注意头结点不算位置!

\* 0位置是首节点! 我想在3处插入，pos = 3,那我将头指针移动到2处，

\* 然后就是在头结点的后面插入\*/

int i = 0;//注意i从0开始

linklist \* p = H, \*q = NULL; //p是头指针，q指向pos

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

{

p = p->next;

i++;

}

/\*正确情况下，有0-5个位置，pos=3,那我要将p指向位置2。

\* i = 0时p指向0位置，i++,i=1;

\* 1 = 1时p指向1位置，i++,i=2;

\* 1 = 2时p指向2位置，i++,i=3,不满足小于pos，退出循环;

\* 再往后就是头插法了

\*

\*如果要在0位置插入，pos = 0,不满足循环条件直接执行下面的头插法代码

\*

\*错误情况: 有0-5个位置,5里指针指向NULL,我最大可以在6处插入，我将p指向5

\*但是我在7,8,9...处插入就错误了，假设我pos = 7

\*i=5时p指向5， i++,i=6;

\*i=6时p指向NULL， i++,i=7退出循环;

\*往下执行头插法代码时，p指向NULL了，p->next就错误了!

\*所以while循环里要求p != NULL,就是用来限制pos不能超过6的！

\*/

if (NULL == p)

{

printf("error pos\n");

return -1;

}

if (NULL == (q = malloc(sizeof(linklist))))

{

printf("malloc failed\n");

return -1;

}

//下面就是同在头结点后面插入

q->data = value;

q->next = p->next;

p->next = q;

return 0;

}

int list\_delete\_value(linklist \*H,datatype value)

{//查找并删除某个确定值

linklist \* p = H,\* q = NULL;//q指向要删除的位置

while(p->next != NULL)

{//第1次循环排除空链表的情况

if (value == p->next->data)

{//头节点不放内容，所以从下一个开始找!

q = p->next;//要删除的位置给q

p->next = q->next;//把q指向的下一个交给p，这样q就没用了

free(q);

q = NULL;//预防野指针

return 0;

}

else

p = p->next;

//p一步步往下走直到指向尾节点时就跳出循环了,p指向尾巴时上一次已经判断尾巴里的内容了

}

printf("no value\n");

return -1;

}

int list\_replace(linklist \* H,datatype old,datatype new)

{

linklist \* p = H;

while(p->next != NULL)

{

if (p->next->data == old)

{

p->next->data = new;

return 0;

}

else

{

p = p->next;

}

}

printf("no old\n");

return -1;

}

int list\_search(linklist \* H,datatype value)

{

int pos = 0;

linklist \* p = H;

while(p->next != NULL)

{

if (value == p->next->data)

{

return pos;

}

else

{

p = p->next;

pos++;

}

}

return -1;

}

void list\_reverse(linklist \*H) //链表的反转

{//输入123456，输出是654321，我们要让输入输出一样

linklist \* p = H->next,\* q = NULL;

H->next = NULL;

while(p != NULL)

{

q = p;

p = p->next;

q->next = H->next;

H->next = q;

}

}

/\*统计节点中最大值，最小值和累加值，自己写的，华三笔试题\*/

void statistic(linklist \*H)

{

datatype Min,Max,t,Total = 0;

Min = Max = H->next->data;

while(H->next != NULL)

{

t = H->next->data;

if (t > Max)

{

Max = t;

}

if (t < Min)

{

Min = t;

}

H = H->next;

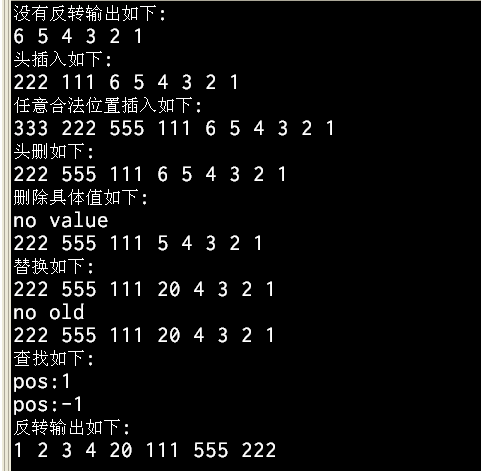
Total = Total + t;

}

printf("Min = %d,Max = %d,Total = %d\n",Min,Max,Total);

}

运行结果如图:



1. 小算法: 如果两个链表la,lb本身已经从小到大排好序，要求将lb插入la的后面，然后从小到大排好序输出。注意: 前提是la,lb本身已经从小到大排好了！

**#include**<stdio.h>

**#include**<stdlib.h>

**typedef** **int** datatype;

struct node

{

datatype data;

struct node \* next;

};

typedef struct node linknode;

linknode \* list\_create();

int head\_insert(linknode \* H,datatype value);

**void** **show**(linknode \*H);

**void** **Merge**(linknode \*Ha,linknode \*Hb);

**int** **main**()

{

linknode \* ha = NULL, \*hb = NULL;

**if** ((ha = list\_create()) == NULL)

{

**printf**("create list failed\n");

**return** -1;

}

**if** ((hb = list\_create()) == NULL)

{

**printf**("create list failed\n");

**return** -1;

}

//要求两个链表本身已经是从小到大输出，这个前提要满足!

//另外从小到大输出，那就要从大到小输入!

head\_insert(ha, 60);

head\_insert(ha, 50);

head\_insert(hb, 70);

head\_insert(hb, 60);

head\_insert(hb, 3);

show(ha);

show(hb);

Merge(ha, hb);

show(ha);

**return** 0;

}

linknode \*

**list\_create**()

{

linknode \* h = NULL;//头指针

**if** ((h = **malloc**(**sizeof**(linknode))) == NULL)

{

**printf**("malloc failed\n");

**return** NULL;

}

h->next = NULL;

**return** h;

}

**int** **head\_insert**(linknode \* h, datatype value)

{

linknode \*p = NULL;

**if** ((p = **malloc**(**sizeof**(linknode))) == NULL)

{

**printf**("malloc node failed\n");

**return** -1;

}

p->data = value;

p->next = h->next;

h->next = p;

**return** 0;

}

**void** **show**(linknode \*h)

{

**while** (h->next != NULL)

{

**printf**("%d ", h->next->data);

h = h->next;

}

putchar(10);

}

**void** **Merge**(linknode \* ha, linknode \*hb)

{

linknode \* p = ha->next, \*q = hb->next, \*r = ha;

**free**(hb);

//hb插入ha的后面，hb的头结点就不需要了,这一步别忽略了!

**while** (p && q)

{

**if** (p->data <= q->data)

{

r->next = p;

r = p;

p = p->next;

}

**else**

{

r->next = q;

r = q;

q = q->next;

}

}

**if** (NULL == p)

{

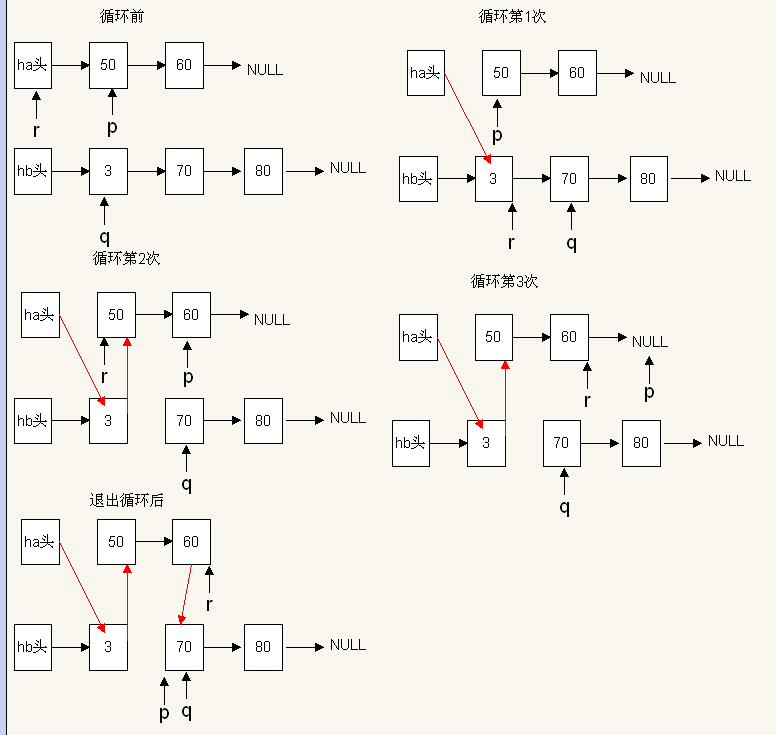
p = q;

}

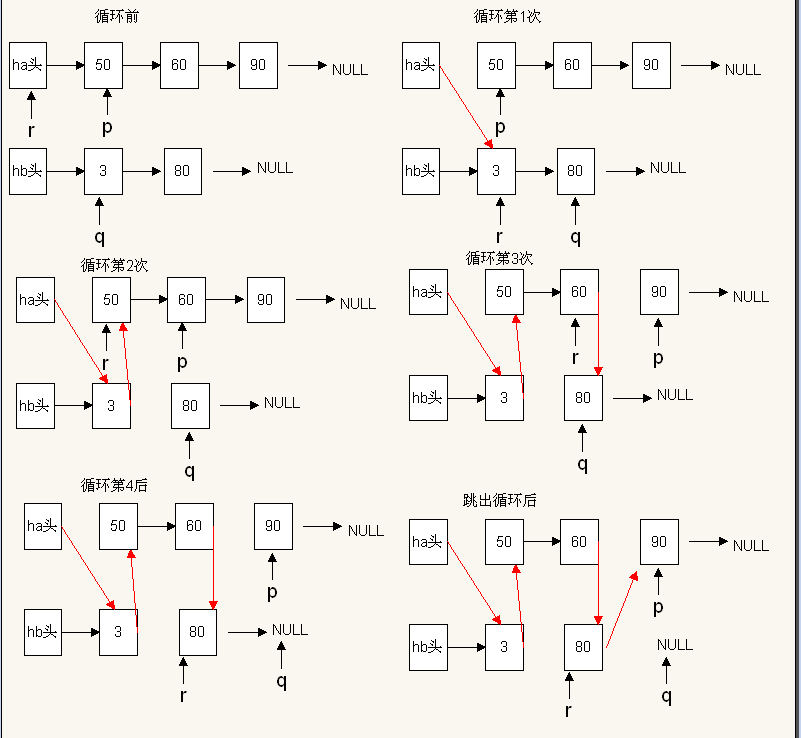
r->next = p;

}

解析1: ha的节点数少于hb的情况



解析2: ha的节点数多于hb的情况



1. 什么是泛型

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：

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。

1. 数组和链表的优缺点比较

数组优点: 存取速度很快

数组缺点:

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

链表优点:

空间没有限制

插入删除元素很快

链表缺点:

存取速度很慢

# 双向循环链表

程序如下:

**#include** <stdio.h>

**#include** <stdlib.h>

**typedef** **int** datatype;

**typedef** **struct** node

{

datatype data;

**struct** node \*next;

**struct** node \*prior;

} linklist;

linklist \* **list\_create**();

**int** **list\_insert**(linklist \*H, datatype value);

**int** **list\_delete**(linklist \*H);

**void** **list\_show(**linklist \*H);

**int** **main**(**void**)

{

linklist \*H = NULL;

**if** ((H = list\_create()) == NULL)

{

**printf**("list\_create failed\n");

**return** -1;

}

list\_insert(H, 2);

list\_insert(H, 3);

list\_insert(H, 4);

list\_insert(H, 5);

**list\_show(H);**

list\_delete(H);

**list\_show(H);**

list\_delete(H);

**list\_show(H);**

**return** 0;

}

linklist \***list\_create**()

{

linklist \*H = NULL;

**if** (NULL == (H = **malloc**(**sizeof**(linklist))))

{

**printf**("malloc failed\n");

**return** NULL;

}

H->data = 1;//这里头结点就不是不存数据了!!

H->next = H;

H->prior = H;

**return** H;

}

**int** **list\_insert**(linklist \*H, datatype value)

{//头插，插在头结点的后面。看笔记的解析，画个图就清楚了

linklist \*p = NULL;

**if** ((p = **malloc**(**sizeof**(linklist))) == NULL)

{

**printf**("malloc failed\n");

**return** -1;

}

p->data = value;

p->next = H->next;

p->next->prior = p;

H->next = p;

p->prior = H;

**return** 0;

}

void **list\_show**(linklist \*H)

{

linklist \*p = H;

**while** (p->next != H)

{//如图，如果只有一个节点p->next指向H，这时候不用循环，直接把这个

//节点的值输出就行了。若不是一个节点，进入循环，当p->next再次指向H时

//就证明链表遍历一次结束了，这时候就退出循环，把H指向的前一个节点值输出就行了

//这样就从H节点值开始，遍历一圈，到H前一个节点值结束

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

p = p->next;

}

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

}

**int** **list\_delete**(linklist \*H)

{//删除H->next指向的节点，看笔记解析，画个图就明白了

linklist \*p = H->next;

H->next = p->next;

p->next->prior = H;

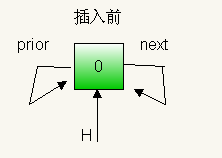
**free**(p);

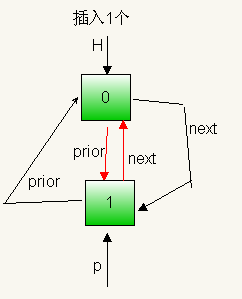
p = NULL;

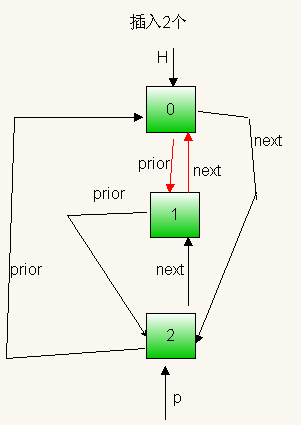
}

程序解析:

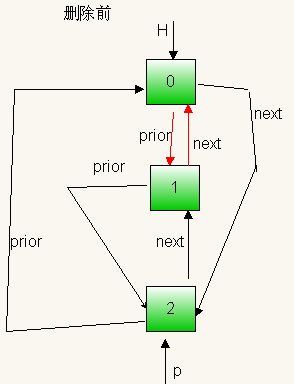
插入函数解析

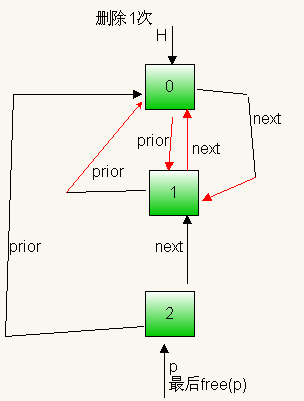


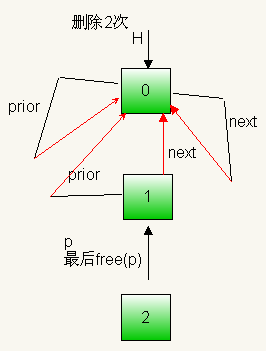




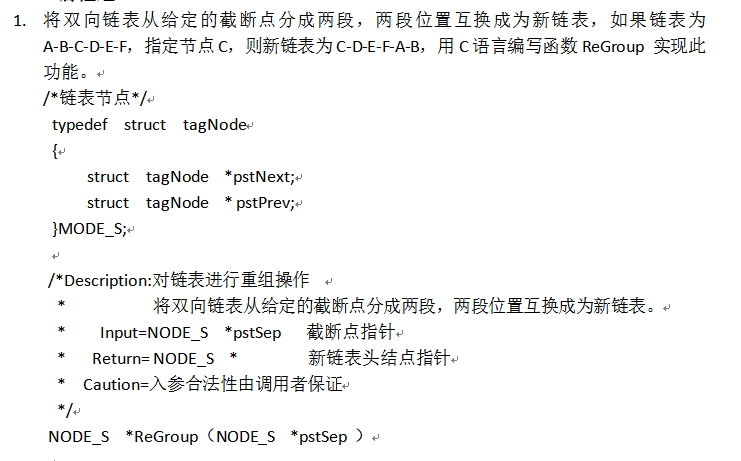
删除函数解析:



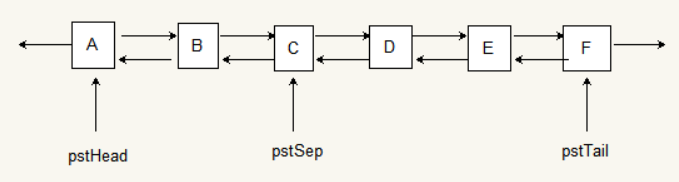




1. 华三笔试题



解:



NODE\_S \*ReGroup（NODE\_S \*pstSep ）

{

NODE\_S \*pstHead = pstSep->pstPrev->pstPrev;

NODE\_S \*pstTail = pstSep->->pstNext->pstNext->pstNext;

//先假设截断，将AB加到F的后面

pstHead-> pstPrev = pstTail;

pstTail-> pstNext = pstHead;

//截断

pstSep-> pstPrev->pstNext = NULL;

pstSep-> pstPrev = NULL;

}

# 连续栈

1. 栈的定义: 一种可以实现“**先进后出**”的存储结构

**栈类似于箱子，只允许在一端进行操作**

1. 栈分类: 静态栈(连续栈) 动态栈(链式栈)
2. 操作: 入栈 出栈
3. 连续栈程序:

**#include** <stdio.h>

**#include** <stdlib.h>

**#define** N 64

**typedef** **int** datatype;

**typedef** **struct**

{

datatype data[N];

**int** top; //top是int型的!!

} sqstack;

sqstack \***stack\_create**();

**void** **stack\_show**(sqstack \*s);

**int** **push**(sqstack \*s,datatype value);

datatype **pop**(sqstack \*s);

**int** **stack\_full**(sqstack \*s);

**int** **stack\_empty**(sqstack \*s);

datatype **stack\_top**(sqstack \*s);

**void** **stack\_clear**(sqstack \*s);

**void** **stack\_free**(sqstack \*s);

**int** **main**(**void**)

{

sqstack \*s = NULL;

**if** ((s = stack\_create()) == NULL)

{

**printf**("stack\_create failed\n");

**return** -1;

}

push(s,10);

push(s,20);

push(s,30);

push(s,40);

stack\_show(s);

**printf**("返回栈顶元素:%d\n",stack\_top(s));

**printf**("出栈的元素:%d\n",pop(s));

**printf**("出栈的元素:%d\n",pop(s));

stack\_show(s);

**printf**("stack\_clear后show:\n");

stack\_clear(s);

stack\_show(s);

**printf**("出栈的元素:%d\n",pop(s));

**printf**("出栈的元素:%d\n",pop(s));

stack\_free(s);

**return** 0;

}

sqstack \***stack\_create**()

{//创建的第一个元素不存放数值!!

sqstack \*s = NULL;

**if** (NULL == (s = **malloc**(**sizeof**(sqstack))))

{

**printf**("malloc failed\n");

}

s->top = -1;

**return** s;

}

**int** **push**(sqstack \*s, datatype value)

{

**if** (stack\_full(s))

{

**printf**("stack is full\n");

**return** -1;

}

s->top++;

s->data[s->top] = value;

**return** 0;

}

datatype **pop**(sqstack \*s)

{//出站是因为要返回出站的数值以判断正确，所以

//函数类型是datatype 入站就定义成int就行了

**if** (stack\_empty(s))

{

**printf**("stack is empty\n");

**return** -1;

}

s->top--;

**return** s->data[s->top + 1];

}

**int** **stack\_full**(sqstack \*s)

{

**return** s->top == N - 1;

//注意是“= =”，如果相等就证明满了就返回真，否则就返回假

}

**int** **stack\_empty**(sqstack \*s)

{

**return** s->top == -1 ? 1 : 0;

//或者直接写成return s->stop == -1;

}

datatype **stack\_top**(sqstack \*s)

{//返回栈顶元素

**return** s->data[s->top];

}

**void** **stack\_show**(sqstack \*s)

{

**int** i;

**for**(i = 0;i <= s->top;i++)

{

**printf**("%d ",s->data[i]);

}

putchar(10);

}

**void** **stack\_clear**(sqstack \*s)

{//清空栈，只留第一个不放数值的元素

s->top = -1;

}

**void** **stack\_free**(sqstack \*s)

{

**while** (!stack\_empty(s))

{

**printf**("%d ", pop(s));

}

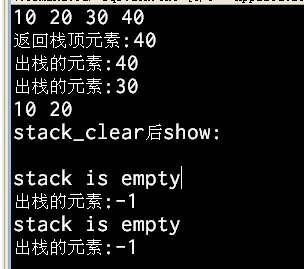
**printf**("\n");

**free**(s);

s = NULL;

}

运行结果:



# 链式栈

这次把整个程序写在三个文件中，看起来非常清楚(∩\_∩)

1. linkstack.h程序

**#ifndef** LINKSTACK\_H\_

**#define** LINKSTACK\_H\_

**typedef** **int** datatype;

**typedef** **struct** node

{

datatype data;

**struct** node \*next;

}stack;

stack \***stack\_create**();

**int** **push**(stack \*s,datatype value);

datatype **pop**(stack \*s);

**void** **stack\_show**(stack \*s);

**int** **stack\_empty**(stack \*s);

datatype **stack\_top**(stack \*s);

**void** **stack\_clear**(stack \*s);

**void** **stack\_free**(stack \*s);

**#endif** /\* LINKSTACK\_H\_ \*/

1. linkstack.c程序

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** "linkstack.h"

stack \***stack\_create**()

{//头结点不存放数据，栈空了这个节点也存在!!

stack \*s = NULL;

**if** (NULL == (s = **malloc**(**sizeof**(stack))))

{

**printf**("malloc failed\n");

**return** NULL;

}

s->next = NULL;

**return** s;

}

**int** **push**(stack \*s,datatype value)

{//这个入栈就是链表的在头结点的后面插入!!

stack \*p = NULL;

**if**(NULL == (p = **malloc**(**sizeof**(stack))))

{

**printf**("malloc node failed\n");

**return** -1;

}

p->data = value;

p->next = s->next;

s->next = p;

**return** 0;

}

datatype **pop**(stack \*s)

{//这个出栈就是链表的在删除头节点后面的节点。链表的特点就是先插入的节点

//离头结点远，头结点后边那个是最近插入的。这个特点正好用在了链式栈上，体现了

//先进后出的原则

stack \*p = NULL;//p指向要弹出的数值

datatype t;//存放弹出来的数值

**if** (stack\_empty(s))

{

**printf**("stack is empty\n");

**return** -1;

}

p = s->next;

t = p->data;

s->next = p->next;

**free**(p);

p = NULL;

**return** t;//让我们知道哪个栈顶元素出栈了

}

**int** **stack\_empty**(stack \*s)

{//栈空了的含义不是一个节点也没有，是只有一个头节点!

**return** s->next == NULL;

//注意这里是“==”!这写法很精练啊!

}

datatype **stack\_top**(stack \*s)

{//返回栈顶元素 要明白栈顶节点就是离头结点最近的节点，也就是头结点

//后面的节点，这里注意不要写成s->data!头节点不存放内容!

**return** s->next->data;

}

**void** **stack\_show**(stack \*s)

{//同链表的遍历输出是一样的

**while**(s->next != NULL)

{

**printf**("%d ",s->next->data);

//注意是s->next->data!

s = s->next;

}

putchar('\n');

}

**void** **stack\_clear**(stack \*s)

{//就是让栈空了只剩一个头节点

**while**(!stack\_empty(s))

{

**printf**("clear弹出的:%d ",pop(s));

}

**puts**(""); //这也是一种输出换行符的方法

}

**void** **stack\_free**(stack \*s)

{//完全干掉栈，连头结点也干掉了

stack\_clear(s);

**free**(s);

s = NULL;

}

1. main.c程序

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** "linkstack.h"

**int** **main**(**void**)

{

stack \*s = NULL;

**if** (NULL == (s = stack\_create()))

{

**printf**("stack\_create failed\n");

**return** -1;

}

push(s, 10);

push(s, 20);

push(s, 30);

push(s, 40);

stack\_show(s);

**printf**("栈顶元素是:%d\n", stack\_top(s));

**printf**("出栈元素是:%d\n", pop(s));

**printf**("出栈元素是:%d\n", pop(s));

stack\_show(s);

**printf**("stack\_clear后:\n");

stack\_clear(s);

stack\_show(s);

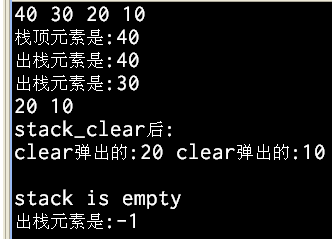
**printf**("出栈元素是:%d\n", pop(s));

stack\_free(s);

**return** 0;

}

运行结果:



# 队列

1. 定义: 一种可以实现“**先进先出**”的存储结构

**栈只允许在一端进行操作，而队列在两端操作，队尾进行存入，队头进行删除**

1. 静态队列(连续队列) 动态队列(链式队列)

**静态队列通常必须是循环队列**

1. 静态循环队列要搞清楚的问题:
2. 静态队列为什么必须是循环队列

防止浪费空间

1. 循环队列需要几个参数来确定

需要2个参数来确定: 队头指针front和队尾指针rear。队头删除，队尾存入

注意: 两个参数不同场合有不同的含义，这个先记住，慢慢体会

场合1 队列初始化 front和rear的值都是零

场合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，是**下一个！！！**

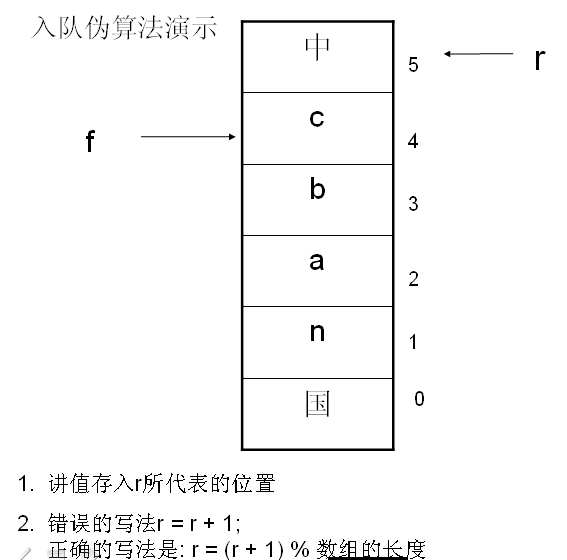
场合3 队列空 front和rear的值相等，但不一定是零

1. 循环队列入队伪算法讲解

第一步: 将值存入rear所代表的位置

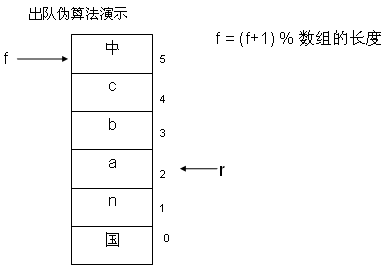
第二步: **错误**写法 rear=rear+1; (＞﹏＜)

正确写法rear=（rear+1）%数组的长度 (∩\_∩)



1. 循环队列出队伪算法讲解

front=（front+1）%数组的长度



1. 如何判断循环队列是否为空(后面有图解)

如果front于rear的值相等，则该队列就一定为空

(这句话不对，但是看完如何判断满就理解了)

1. 如何判断循环队列是否已满(后面有图解)

预备知识1: 当front = rear时，这个循环队列是满的还是空的？这个没有办法判断。为了 区分空和满，规定front指向队头元素的位置，rear指向队尾元素的下一个位置

(**rear指向的必须为空**)，所以满队元素个数比数组元素个数少一个

**也就是说牺牲一个存储位置，换来了不再纠结空和满的判断(∩\_∩)**

预备知识2: front的值可能比rear大，也可能比rear小，也可能两者相等

那么如何判断队列已满？就是如果f和r的值紧挨着，则队列已满

用C语言伪算法表示就是:

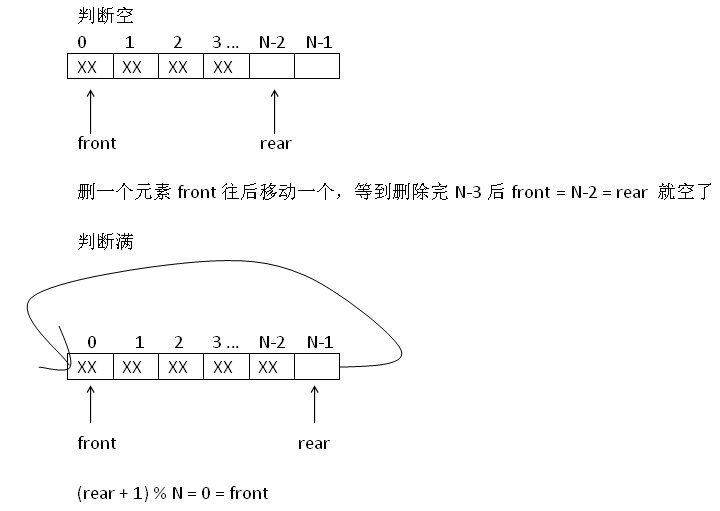
if（（r+1）%数组长度== f）

已满

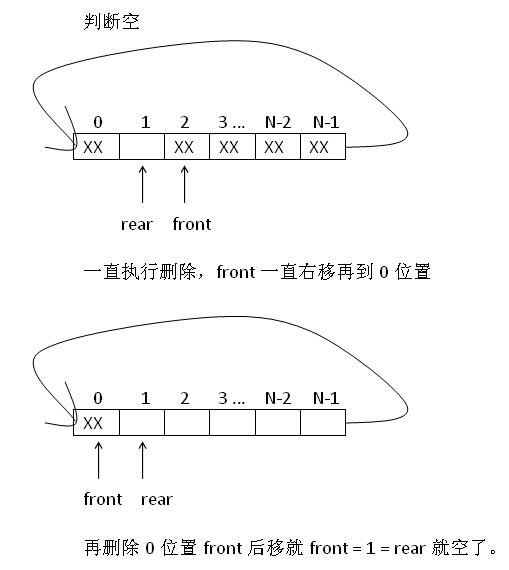
else

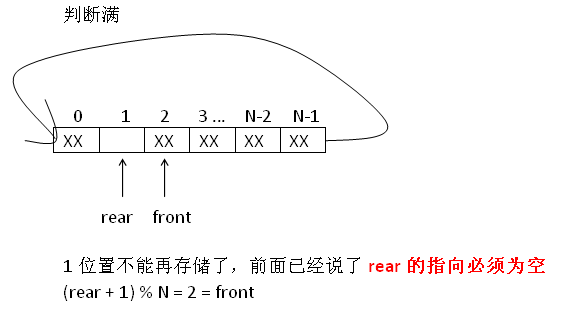
不满

1. 关于上述E和F的图解
   1. 数组有N个元素，没有进入循环时



* 1. 数组有N个元素，进入循环后





1. 静态循环队列程序分三个文件

sqqueue.h文件:

**#ifndef** SQQUEUE\_H\_

**#define** SQQUEUE\_H\_

**#define** N 5

**typedef** **int** datatype;

**typedef** **struct**

{

datatype data[N];

**int** front;

**int** rear;

}sqqueue;

sqqueue \***queue\_create**();

**int** **enqueue**(sqqueue \*sq,datatype value);

**int** **queue\_full**(sqqueue \*sq);

datatype **dequeue**(sqqueue \*sq);

**int** **queue\_empty**(sqqueue \*sq);

**void** **queue\_show**(sqquee \*sq);

**void** **queue\_clear**(sqqueue \*sq);

**#endif** /\* SQQUEUE\_H\_ \*/

sqqueue.c文件:

**#include** <stdlib.h>

**#include** <stdio.h>

**#include** "sqqueue.h"

sqqueue \***queue\_create**()

{

sqqueue \*sq = NULL;

**if** (NULL == (sq = **malloc**(**sizeof**(sqqueue))))

{

**printf**("malloc failed\n");

**return** NULL;

}

sq->front = 0;

sq->rear = 0;

**return** sq;

}

**int** **enqueue**(sqqueue \*sq,datatype value)

{

**if** (queue\_full(sq))

{

**printf**("queue is full\n");

**return** -1;

}

sq->data[sq->rear] = value;

sq->rear = (sq->rear + 1) % N;

**return** 0;

}

**int** **queue\_full**(sqqueue \*sq)

{

**if** (sq->front == (sq->rear + 1) % N)

{

**return** 1;

}

**else**

{

**return** 0;

}

}//可以简写成return = sq->front == (sq->rear + 1) % N

datatype **dequeue**(sqqueue \*sq)

{

**int** t;

**if** (queue\_empty(sq))

{

**printf**("queue is empty\n");

**return** -1;

}

t = sq->data[sq->front];

sq->front = (sq->front + 1) % N;

**return** t;

}

**int** **queue\_empty**(sqqueue \*sq)

{

**return** (sq->front == sq->rear) ? 1 : 0;

}

**void** **queue\_show**()

{

**int** i = sq->front;

**while** (i != sq->rear)

{

**printf**("%d ",sq->data[i]);

i = (i + 1) % N;

}

putchar(10);

}

**void** **queue\_clear**(sqqueue \*sq)

{

sq->front = sq->rear = 0;

}

main.c文件:

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** "sqqueue.h"

**int** **main**(**int** argc, **char** \*\*argv)

{

sqqueue \*sq = NULL;

**if** (NULL == (sq = queue\_create()))

{

**printf**("queue\_creat failed\n");

**return** -1;

}

//定义的数组N是5，只能存4个数!!我存5个数最后一个应该放不进才对!!

enqueue(sq,10);

enqueue(sq,20);

enqueue(sq,30);

enqueue(sq,40);

enqueue(sq,50);

queue\_show(sq);

**printf**("出队的值是:%d\n",dequeue(sq));

**printf**("出队的值是:%d\n",dequeue(sq));

queue\_show(sq);

queue\_clear(sq);

enqueue(sq,10);

enqueue(sq,20);

enqueue(sq,30);

enqueue(sq,40);

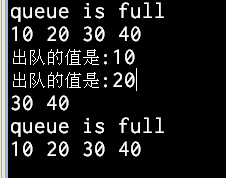
enqueue(sq,50);

queue\_show(sq);

**return** 0;

}

运行结果:



# 链式队列

链式队列程序，分三部分

linkqueue.h程序:

**#ifndef** LINKQUEUE\_H\_

**#define** LINKQUEUE\_H\_

**typedef** **int** datatype;

**typedef** **struct** node

{

datatype data;

**struct** node \*next;

}linknode;

**typedef** **struct**

{

**struct** node \*front;

**struct** node \*rear;

}queue;

queue \* **queue\_create**();

**int** **enqueue**(queue \*q,datatype value);

datatype **dequeue**(queue \*q);

**int** **queue\_empty**(queue \*q);

**void** **queue\_show**(queue \*q);

**void** **queue\_clear**(queue \*q);

**void** **queue\_free**(queue \*q);

**#endif** /\* LINKQUEUE\_H\_ \*/

linkqueue.c程序:

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** "linkqueue.h"

queue\* **queue\_create**()

{

linknode \*p = NULL;

queue \*q = NULL;

**if** ((p = **malloc**(**sizeof**(linknode))) == NULL)

{//创建的头结点不存放数据!!

**printf**("malloc node failed\n");

**return** NULL;

}

**if** ((q = **malloc**(**sizeof**(queue))) == NULL)

{

**printf**("malloc queue failed\n");

**return** NULL;

}

q->front = p;

q->rear = p;

**return** q;

}

**int** **enqueue**(queue \*q, datatype value)

{//从队尾插入

linknode \* p = NULL;

**if** ((p = **malloc**(**sizeof**(linknode))) == NULL)

{

**printf**("malloc node failed\n");

**return** -1;

}

p->data = value;

p->next = NULL;

q->rear->next = p;

q->rear = p;

**return** 0;

}

datatype **dequeue**(queue \*q)

{//从队头删除

linknode \*p = NULL;

datatype t;

**if** (queue\_empty(q))

{

**printf**("queue is empty");

**return** -1;

}

p = q->front;

q->front = p->next;

t = q->front->data;

/\*注意:t = q->front->data;不能写成t = p->data!!因为

\* 开始时front指向头结点，但头不存放数据。比如第1次删除，虽然

\* 删除的是头结点，但返回的却是头结点的下一个节点值，我们就认为

\* 删除了第一个存放数值的节点，往后也是这样。

\* 而show输出时也是从front->next开始输出，所以虽然程序实际执行的

\* 操作并不是我们认为的操作，但是执行结果却和我们预想的一致!

\*/

**free**(p);

p = NULL;

**return** t;

}

**int** **queue\_empty**(queue \*q)

{//个人理解:由dequeue函数分析，当front指向rear前一个节点时，再执行删除，

//释放的是rear前一个节点，但是返回的却是rear的值，所以front = rear时，

//我们就认为空了

**return** q->front == q->rear;

}

**void** **queue\_show**(queue \*q)

{//自己写的。。。

linknode \* p = q->front;

**while** (p->next != NULL)

{//因为头结点不存放数值，所以是p->next->data!

//另外是p->next != NULL，不是p != NULL，因为指向

//最后一个节点时上一次已经把最后一个节点值输出

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

p = p->next;

}

**puts**("");

//当队列是空时，p->next = NULL，不执行任何操作

}

**void** **queue\_clear**(queue \*q)

{

**printf**("clear:");

**while** (!queue\_empty(q))

{

**printf**("%d ", dequeue(q));

}

**puts**("");

}

**void** **queue\_free**(queue \*q)

{

queue\_clear(q);

**free**(q->front);

**free**(q);//盛放头，尾指针的节点也要释放掉!

}

main.c程序:

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** "linkqueue.h"

**int** **main**(**int** argc, **const** **char** \*argv[])

{

queue \*q;

**if**((q = queue\_create()) == NULL)

{

**printf**("queue\_create failed\n");

**return** -1;

}

enqueue(q,10);

enqueue(q,20);

enqueue(q,30);

enqueue(q,40);

queue\_show(q);

**printf**("%d \n",dequeue(q));

**printf**("%d \n",dequeue(q));

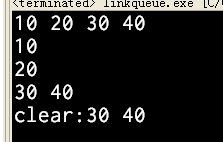
queue\_show(q);

queue\_free(q);

**return** 0;

}

运行结果:



# 递归

1. 定义

**一个函数自己直接或间接调用自己**

1. 举例

**#include** <stdio.h>

**#include** <stdlib.h>

**void** **f**(**int** n)

{

**if** (1 == n)

{

**printf**("哈哈\n");

}

**else**

{

f(n-1);

}

}

**int** **main**(**void**)

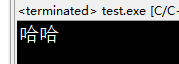
{

f(7);

**return** 0;

}

运行结果:



1. n的阶乘用递归来实现
   1. 如果不用递归来实现

**#include** <stdio.h>

**#include** <stdlib.h>

**int** **main**(**void**)

{

**int** i,n,mul = 1;

**printf**("请输入n的值:");

**scanf**("%d",&n);

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

{

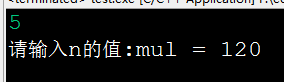
mul = mul \* i;

}

**printf**("mul = %d\n",mul);

**return** 0;

}



* 1. 如果用递归来实现

#include <stdio.h>

#include <stdlib.h>

int f(int n)

{

if(1 == n)

{

return 1;

}

else

{

return f(n-1) \* n;

}

}

int main(void)

{

int n;

printf("请输入n的值:");

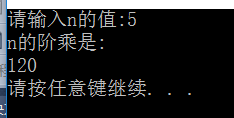
scanf("%d",&n);

printf("n的阶乘是:\n%d\n",f(n));

system("pause");

return 0;

}



1. 1+2+3+....+100之和用递归来实现
   1. 不用递归实现
   2. 用递归实现

#include <stdio.h>

#include <stdlib.h>

int f(int n)

{

if(1 == n)

{

return 1;

}

else

{

return f(n-1) + n;

}

}

int main(void)

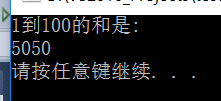
{

printf("1到100的和是:\n%d\n",f(100));

system("pause");

return 0;

}



1. 一个函数为什么可以自己调用自己

函数的调用:

当在一个函数的运行期间调用另一个函数时，在运行被调函数之前，系统需要完成三件事:

1. 将所有的实际参数，返回地址等信息传递给被调函数保存
2. 为被调函数的局部变量（包括形参）分配存储空间
3. 将控制转移到被调函数的入口

从被调函数返回主调函数之前，系统也要完成三件事:

1. 保存被调函数的返回结果
2. 释放被调函数所占的存储空间
3. 依照被调函数保存的返回地址将控制转移到调用函数

当有多个函数相互调用时，按照“后调用先返回”的原则，上述函数之间信息传递和控制转移必须借助“栈”来实现，即系统将整个程序运行时所需的数据空间安排在一个栈中，每当调用一个函数时，就在栈顶分配一个存储区，进行压栈操作，每当一个函数退出时，就释放它的存储区，进行出栈操作，当前运行的函数永远在栈顶位置。

**A函数调用A函数和A函数调用B函数在计算机看来是没有任何区别的，只不过用我们日常的思维方式理解比较怪异而已！**

例子: 间接调用自己举例

#include<stdio.h>

void g(int);

void f(int n)

{

g(n);

}

void g(int m)

{

f(m);

}

int main(void)

{

return 0;

}

1. **递归必须满足三个条件**
2. 递归必须得有一个明确的终止条件
3. 该函数所处理的数据规模必须在递减
4. 这个转化必须是可解的

例子: 递增的值可以增加，但处理的数据的规模需要递减

**这个例子纠结了很久问别人画图才会！**

#include<stdio.h>

int f(int n)

{

if(n>7)

printf("哈哈\n");

else

n=f(n+1);

return n;

}

int main(void)

{

int val;

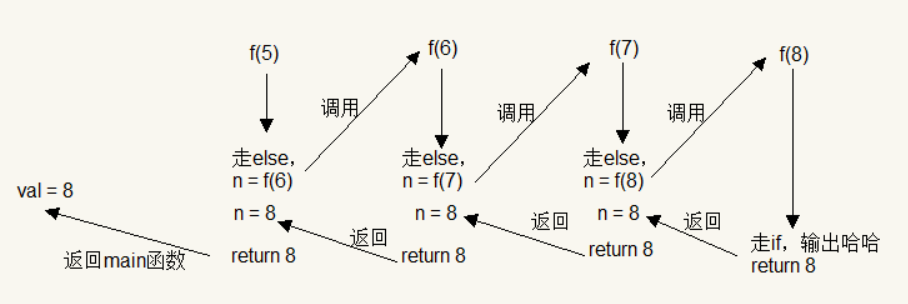
val=f(5);

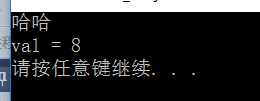
printf("val=%d\n",val);

return 0;

}

解析: 如下图





1. 循环和递归的比较

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。

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

1. 汉诺塔

汉诺塔问题是源于印度一个古老传说的益智玩具。大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，**在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。**

**可以从百度上搜汉诺塔的小游戏玩一下，就明白了！**



或许下面的解释会好理解一些:  
现在要求庙里的老和尚把这64个盘子全部移动到**第三个**柱子上。移动的时候始终只能小盘子压着大盘子，一次移动一个。

001、此时老和尚（后面我们叫他第1个和尚）觉得很难，所以他想：要是有一个人能把前63个盘子先移动到第二个柱子上，我再把最后一个盘子直接移动到第三个柱子，再让那个人把刚才的前63个盘子从第二个柱子上移动到第三个柱子上，我的任务就完成了，简单。所以他找了第2个和尚，命令：

① 你把前63个盘子移动到第二柱子上

② 等我自己把第64个盘子移到第三个柱子上后

③ 你把前63个盘子移动到第三柱子上

002、第2个和尚接了任务，也觉得很难，所以他也和第1个和尚一样想：要是有一个人能把前62个盘子先移动到第三个柱子上，我再把最后一个盘子直接移动到第二个柱子，再让那个人把刚才的前62个盘子从第三个柱子上移动到第二个柱子上，我的任务就完成了，简单。所以他也找了第3个和尚，命令：

① 你把前62个盘子移动到第三柱子上

② 等我自己把第63个盘子移到第二个柱子上后

③ 你把前62个盘子移动到第二柱子上

003、第3个和尚接了任务，又把移动前61个盘子的任务依葫芦画瓢的交给了第4个和尚，递推下去，直到把任务交给了第64个和尚为止。（**估计第64个和尚很郁闷，没机会也命令下别人，因为到他这里盘子只有一个了**）。

004、到此任务下交完成，到各司其职完成的时候了。完成后开始回推了：

第64个和尚移动第1个盘子，把它移开，然后第63个和尚移动他给自己分配的第2个盘子。第64个和尚再把第1个盘子移动到第2个盘子上。到这里第64个和尚的任务完成，第63个和尚完成了第62个和尚交给他的任务的第一步。

从上面可以看出，只有第64个和尚的任务完成了，第63个和尚的任务才能完成，只有第2到第64个和尚的任务完成后，第1个和尚的任务才能完成。这是一个典型的递归问题。

现在我们以有**3个**盘子来分析：

第1个和尚命令：

  ㈠ 第2个和尚你先把第一柱子前2个盘子移动到第二柱子。（借助第三个柱子）。

㈡第1个和尚我自己把第一柱子最后的盘子移动到第三柱子。

㈢第2个和尚你把前2个盘子从第二柱子移动到第三柱子。

很显然，第㈡步很容易实现 (∩\_∩)

其中第㈠步。第2个和尚他有2个盘子，他就命令：

       ① 第3个和尚你把第一柱子第1个盘子移动到第三柱子。（借助第二柱子）

       ② 第2个和尚我自己把第一柱子第2个盘子移动到第二柱子上。

       ③ 第3个和尚你把第1个盘子从第三柱子移动到第二柱子。

同样，第②步很容易实现，但第3个和尚他只需要移动1个盘子，所以他也不用下派任务了。

（注意: 这就是**停止递归的条件，也叫边界值**）

第㈢步可以分解为，第2个和尚还是有2个盘子，命令：

      ①第3个和尚你把第二柱子上的第1个盘子移动到第一柱子。

      ② 第2个和尚我把第2个盘子从第二柱子移动到第三柱子。

      ③第3个和尚你把第一柱子上的盘子移动到第三柱子。

分析组合起来就是：1→3   1→2    3→2    1→3   2→1  2→3   1→3共需要**7步**，也就是一共**3个盘子需要7步**。（百度搜汉诺塔小游戏玩一遍就会了(∩\_∩)）

如果是4个盘子，则第1个和尚的命令中第1步和第3步各有3个盘子，所以各需要7步，共14步，再加上第1个和尚的1步，所以4个盘子总共需要移动7+1+7=15步，同样，5个盘子需要15+1+15=31步，6个盘子需要31+1+31=63步……由此可以知道，移动n个盘子需要

(2的n次方减1)

从上面分析可知把n个盘子从一柱移到三柱的**伪算法**：

（1）把一柱上（n-1）个盘子借助三柱移到二柱。

（2）把一柱上第n个盘子移到三柱。

（3）把二柱上（n-1）个盘子借助一柱移到三柱。

汉诺塔程序:

#include <stdio.h>

#include <stdlib.h>

void hannuota(int n,int a,int b,int c)

{

/\*

如果是一个盘子

直接将一柱子上的盘子从一移到三

否则

先将一柱子上的n-1个盘子借助三移到二

直接将一柱子上的盘子从一移到三

最后将二柱子上的n-1个盘子借助一移到三

\*/

if(1 == n)

{

printf("将编号为%d的盘子直接从%d柱子移动到%d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%d柱子移动到%d柱子\n",n,a,c);

hannuota(n-1,b,a,c);

}

}

int main(void)

{

int a = 1;

int b = 2;

int c = 3;

int n;

printf("请输入盘子的个数:");

scanf("%d",&n);

hannuota(n,a,b,c);

system("pause");

return 0;

}

经过汉诺塔小游戏的验证，程序是正确的。

# 树

1. 树的定义

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；每个节点只有一个父节点但可以有多个子节点；但有一个节点没有父节点，它叫做根节点

1. 树的专业术语

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

1. 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

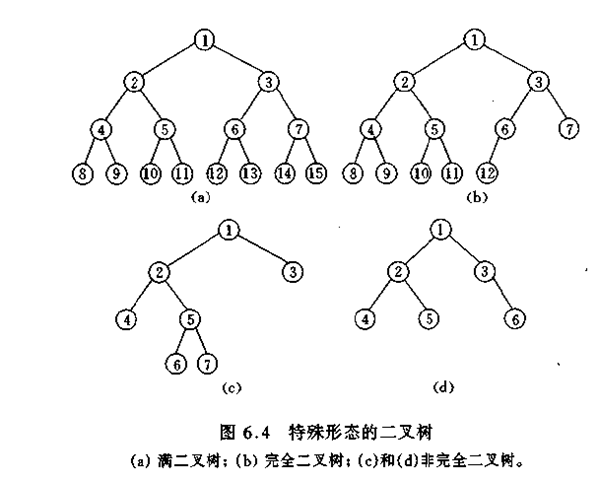
》》满二叉树

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

**满二叉树是完全二叉树的一个特例，就是完全一个都不删就是满**



》》森林：n个互不相交的树的集合

1. 二叉树的连续存储

**连续存储->必须转化成完全二叉树进行**

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

1. 二叉树的链式存储
2. 一般树的存储

》》双亲表示法

》》孩子表示法

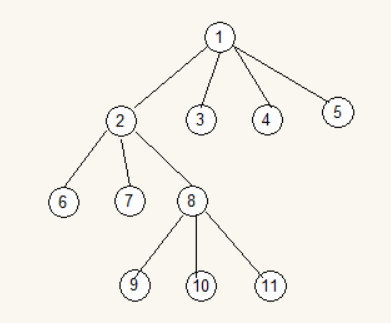
》》双亲孩子表示法

》》**二叉树表示法：**把一个普通树转化成二叉树来存储

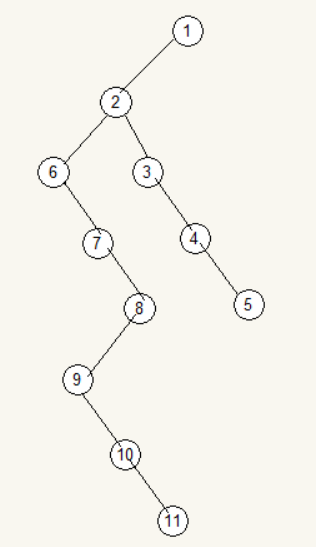
具体转换方法：设法保证任意一个节点的左指针域指向它的第一个孩子，右指针域指向它的下一个兄弟。只要能满足此条件，就可以把一个普通的胡转换成二叉树。

**一个普通树转换成的二叉树根节点一定没有右子树**

例子:



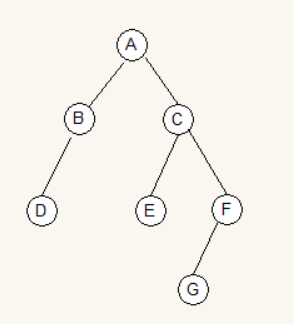
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



1. 森林的存储
2. 先序遍历：【先访问根节点】

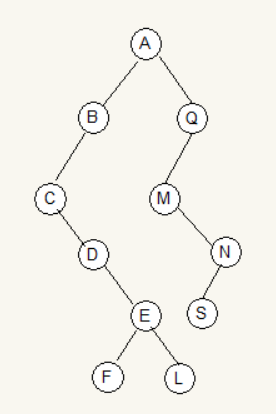
先访问根节点，再先序访问左子树，再先序访问右子树。

例子1：



先：ABDCEFG

例子2：

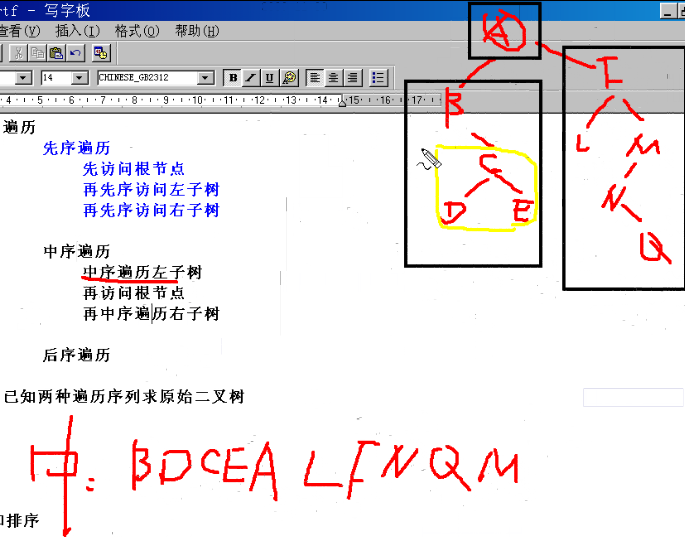


先：ABCDEFLQMNS

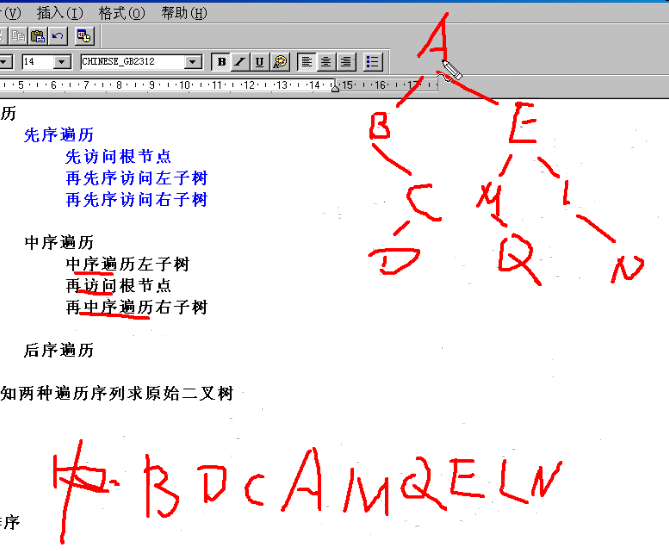
1. 二叉树的中序遍历【中间访问根节点】

中序遍历左子树，再访问根节点，再中序遍历右子树

例子1：



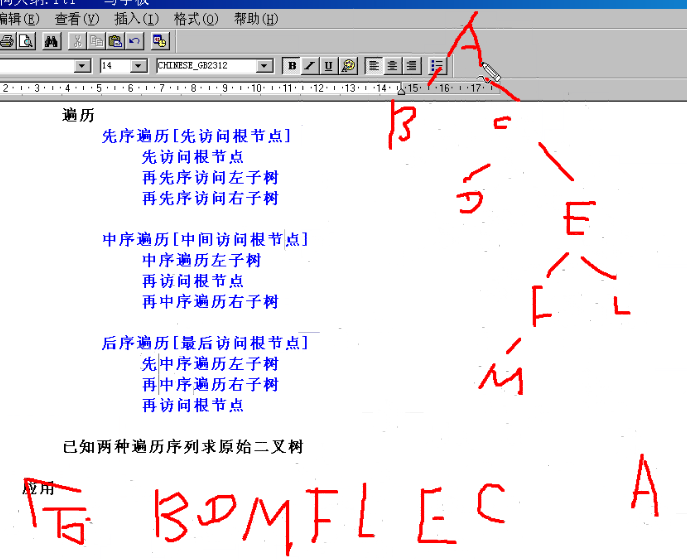
例子2：



1. 二叉树的后序遍历【最后访问根节点】

后序遍历左子树，后序遍历右子树，最后访问根节点

例子1：



例子2：



1. 已知两种遍历序列求原始二叉树概述

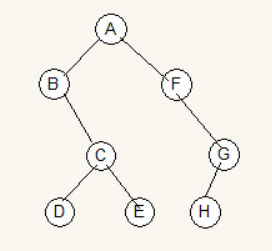
通过先序和中序或者中序和后序我们可以还原出原始的二叉树，但是通过先序和后序是无法还原出原始的二叉树的。

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

1. 已知先序和中序求后序

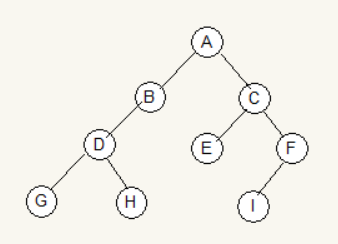
例子1：先序ABCDEFGH 中序BDCEAFHG

分析： BDCE A FHG 左边是左子树，右边是右子树

后序是DECBHGFA

例子2：先序ABDGHCEFI 中序GDHBAECIF

分析： GDHB A ECIF 左边是左子树，右边是右子树

后序：GHDBEIFCA

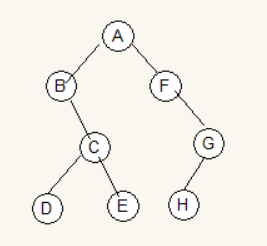
1. 已知中序和后序求先序

中：BDCEAFHG 后：DECBHGFA

分析:通过后序知道A是根，通过中序知道BDCE A FHG 左边是左子树，右边是右子树

左子树BDCE在后序中哪个最后出现，哪个就是根节点，所以B是根节点。通过中序知道B没有左子树，在后序中C在最后面，所以C是B的右子树的根。

FHG中可以F没有左子树，F在后序中最后出现，所以F是A的右子树的根。HG中G在后序中最后面，所以G是F的右子树的根。

 先序：ABCDEFGH

1. 二叉树的程序

main.c:

#include <stdio.h>

#include <stdlib.h>

#include "bitree.h"

#include "linkqueue.h"

int main(int argc, const char \*argv[])

{

bitree \*r = NULL;

if (NULL == (r = bitree\_create()))

{

printf("FILE:%s,LINE:%d,bitree\_create failed\n",\_\_FILE\_\_,\_\_LINE\_\_);

return -1;

}

preorder(r);//先序遍历

putchar(10);

inorder(r);//中序遍历

putchar(10);

postorder(r);//后序遍历

putchar(10);

noorder(r);//层序遍历,用链式队列实现

putchar(10);

return 0;

}

bitree.h:

#ifndef \_BITREE\_H\_

#define \_BITREE\_H\_

typedef char datatype\_t;

typedef struct node\_t

{

datatype\_t data;

struct node\_t \*lchild;

struct node\_t \*rchild;

}bitree;

bitree \* bitree\_create();

void preorder(bitree \*);

void inorder(bitree \*);

void postorder(bitree \*);

void noorder(bitree \*);

#endif

linkqueue.h:

#ifndef \_LINKQUEUE\_H\_

#define \_LINKQUEUE\_H\_

#include "bitree.h"

typedef bitree \* datatype;

typedef struct node

{

datatype data;

struct node \*next;

}linknode;

typedef struct

{

struct node \*front;

struct node \*rear;

}queue;

queue \*queue\_create();

int enqueue(queue \*lq,datatype value);

datatype dequeue(queue \*lq);

int queue\_empty(queue \*lq);

#endif

bitree.c:

#include <stdio.h>

#include <stdlib.h>

#include "bitree.h"

#include "linkqueue.h"

bitree \* bitree\_create()

{

char ch;

bitree \*r = NULL;

scanf("%c",&ch);

if ('#' == ch)

{

return NULL;

}//除了一开始就就输入NULL，这个NULL返回到主函数，往后进入递归，

//这个NULL返回到上一层的bitree\_create()

if (NULL == (r = malloc(sizeof(bitree))))

{

printf("FILE:%s,LINE:%d,malloc failed\n",\_\_FILE\_\_,\_\_LINE\_\_);

return NULL;

}

//创建是按照先序遍历的方式创建

r->data = ch;

r->lchild = bitree\_create();//递归

r->rchild = bitree\_create();

return r;

}

void preorder(bitree \*r)

{

if (r == NULL)

{

return;

}//这个NULL是下面的递归的preorder()中的r->l/rchild可能是NULL

printf("%c ",r->data);

preorder(r->lchild);

preorder(r->rchild);

}

void inorder(bitree \*r)

{

if (NULL == r)

{

return;

}

inorder(r->lchild);

printf("%c ",r->data);

inorder(r->rchild);

}

void postorder(bitree \*r)

{

if (NULL == r)

{

return;

}

postorder(r->lchild);

postorder(r->rchild);

printf("%c ",r->data);

}

void noorder(bitree \*r)

{

queue \*lq = NULL;

bitree \*p = NULL;

if (NULL == (lq = queue\_create()))

{

printf("FILE:%s.LINE:%d,queue\_creat failed\n",\_\_FILE\_\_,\_\_LINE\_\_);

return;

}

printf("%c ",r->data);

enqueue(lq,r);

while (!queue\_empty(lq))

{

p = dequeue(lq);

if (p->lchild != NULL)

{

printf("%c ",p->lchild->data);

enqueue(lq,p->lchild);

}

if (p->rchild != NULL)

{

printf("%c ",p->rchild->data);

enqueue(lq,p->rchild);

}

}

}

linkqueue.c:

#include <stdio.h>

#include <stdlib.h>

#include "linkqueue.h"

#include "bitree.h"

queue\* queue\_create()

{

linknode \*p = NULL;

queue \*lq = NULL;

if (NULL == (p = malloc(sizeof(linknode))))

{

printf("FILE:%s,LINE:%d,malloc node failed\n",\_\_FILE\_\_,\_\_LINE\_\_);

return NULL;

}

if (NULL == (lq = malloc(sizeof(queue))))

{

printf("FILE:%s,LINE:%d,malloc queue failed\n",\_\_FILE\_\_,\_\_LINE\_\_);

return NULL;

}

lq->front = p;

lq->rear = p;

return lq;

}

int enqueue(queue \*lq,datatype value)

{

linknode \*p = NULL;

if (NULL == (p = malloc(sizeof(linknode))))

{

printf("FILE:%s,LINE:%d,malloc node failed\n",\_\_FILE\_\_,\_\_LINE\_\_);

return -1;

}

p->data = value;

p->next = NULL;

lq->rear->next = p;

lq->rear = p;

return 0;

}

datatype dequeue(queue \*lq)

{

linknode \*p = NULL;

datatype t;

if (queue\_empty(lq))

{

printf("queue is empty\n");

return NULL;

}

p = lq->front;

lq->front = p->next;

t = lq->front->data;

free(p);

p = NULL;

return t;

}

int queue\_empty(queue \*lq)

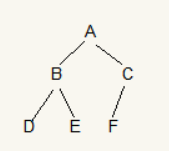
{

return lq->front == lq->rear;

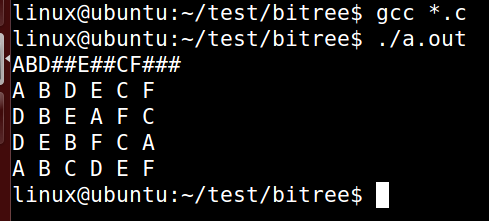
}

运行结果:

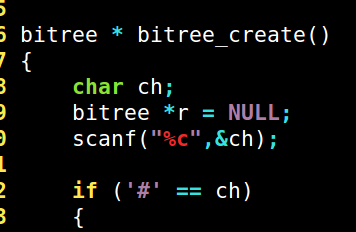
我如果想创造这样的二叉树



就这样输入

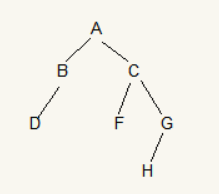


**输入的时候注意问题**:不要输入A回车B回车D……这样输入是得不出结果的，因为

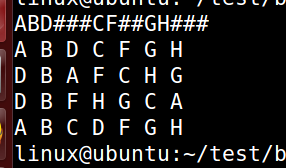


scanf()这里使用%c输入，第一次scanf()接收的是A，递归进入的scanf()接收的不是B，而是A完后的回车……scanf()使用%c输入的问题一定要注意！

我如果想创造这样的二叉树



就这样输入



# 第二篇 查找和排序

# 冒泡排序

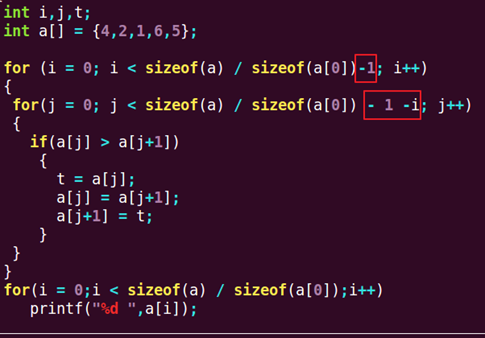
冒泡排序

比较相邻的元素。如果第一个比第二个大，就交换他们两个

对每一对相邻元素作同样的工作。第一次后，最后的元素是最大的数

针对所有的元素重复以上的步骤，除了最后一个

持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较



C:\Users\dell\AppData\Local\YNote\data\840122088@qq.com\4a07e0aa2a104f13b912f62c4fa5b2a6\clipboard.png

# 选择排序

首先通过n-1 次比较，从n个数中找出最小的，将它与第1个数交换。第一次结束，最小的数放在第1个

再通过n-2次比较，从剩余的n-1个数中找出最小的，将它与第二个数交换。

第二次结束，次小的数放在第2个

重复上述过程

区别在于：

冒泡算法，每次比较如果发现较小的元素在后面，就交换两个相邻的元素

 而选择排序算法的改进在于：

 先并不急于调换位置，先从a[0]开始逐个检查，看哪个数最小就记下该数所在的位置P。等一次     扫描完毕，再把a[P]和a[0]对调，这时a[0]到a[9]中最小的数据就换到了最前面。

所以，选择排序每扫描一次数组，只需要一次真正的交换，而冒泡可能需要很多次。但是他俩比 较的次数是一样的。

一句话总结：选择是通过下标改动来排，冒泡则直接交换。

 程序如下:

#include<stdio.h>

int main(int argc, const char \*argv[])

{

int i,j,k = 0;

int a[10] = {1,3,5,7,9,2,4,6,8,0};

for(i = 0;i < sizeof(a)/sizeof(a[0]) -1;i++)

{

        k = i;

        for(j = i + 1;j < sizeof(a)/sizeof(a[0]);j++)

        {

               if(a[j] < a[k])

               {

                      k = j; //k是最小值的下标

               }

        }

        if(i != k)

        {//例如第一次循环，a[0]如果不是最小的，就把a[0]换成最小的

               a[i] ^= a[k];

               a[k] ^= a[i];

               a[i] ^= a[k];

        }

}

for(i = 0;i < sizeof(a)/sizeof(a[0]);i++)

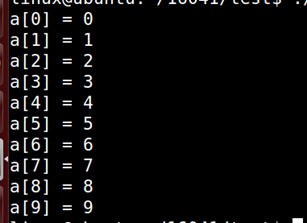
{

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

}

return 0;

}



# 快速排序

上一节的冒泡排序可以说是我们学习第一个真正的排序算法，并且解决了桶排序浪费空间的问题，但在算法的执行效率上却牺牲了很多，它的时间复杂度达到了O(N2)。假如我们的计算机每秒钟可以运行10亿次，那么对1亿个数进行排序，桶排序则只需要0.1秒，而冒泡排序则需要1千万秒，达到115天之久，是不是很吓人。那有没有既不浪费空间又可以快一点的排序算法呢？那就是“快速排序”啦！光听这个名字是不是就觉得很高端呢。

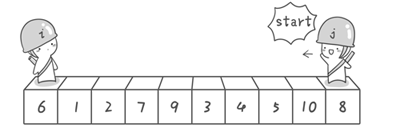
假设我们现在对“6  1  2 7  9  3  4  5 10  8”这个10个数进行排序。首先在这个序列中随便找一个数作为**基准数**（不要被这个名词吓到了，就是一个用来参照的数，待会你就知道它用来做啥的了）。为了方便，就让第一个数6作为基准数吧。接下来，需要将这个序列中所有比基准数大的数放在6的右边，比基准数小的数放在6的左边，类似下面这种排列。

      3  1  2 5  4  **6**  9 7  10  8

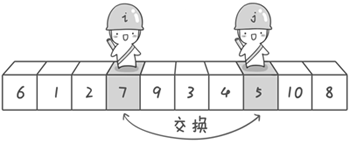
在初始状态下，数字6在序列的第1位。我们的目标是将6挪到序列中间的某个位置，假设这个位置是k。现在就需要寻找这个k，并且以第k位为分界点，左边的数都小于等于6，右边的数都大于等于6。想一想，你有办法可以做到这点吗？

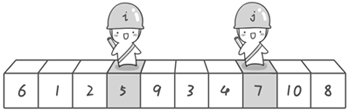
给你一个提示吧。请回忆一下冒泡排序，是如何通过“交换”，一步步让每个数归位的。此时你也可以通过“交换”的方法来达到目的。具体是如何一步步交换呢？怎样交换才既方便又节省时间呢？先别急着往下看，拿出笔来，在纸上画画看。我高中时第一次学习冒泡排序算法的时候，就觉得冒泡排序很浪费时间，每次都只能对相邻的两个数进行比较，这显然太不合理了。于是我就想了一个办法，后来才知道原来这就是“快速排序”，请允许我小小的自恋一下(^o^)。

方法其实很简单：分别从初始序列“6  1  2 7  9  3  4  5 10  8”两端开始“探测”。先从**右**往**左**找一个小于6的数，再从**左**往**右**找一个大于6的数，然后交换他们。这里可以用两个变量i和j，分别指向序列最左边和最右边。我们为这两个变量起个好听的名字“哨兵i”和“哨兵j”。刚开始的时候让哨兵i指向序列的最左边（即i=1），指向数字6。让哨兵j指向序列的最右边（即j=10），指向数字8。



首先哨兵j开始出动。因为此处设置的基准数是最左边的数，所以需要让哨兵j先出动，这一点非常重要（请自己想一想为什么）。哨兵j一步一步地向左挪动（即j--），直到找到一个小于6的数停下来。接下来哨兵i再一步一步向右挪动（即i++），直到找到一个数大于6的数停下来。最后哨兵j停在了数字5面前，哨兵i停在了数字7面前。



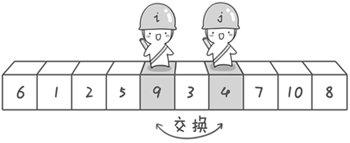


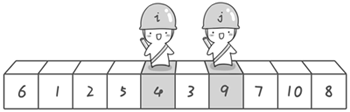
  现在交换哨兵i和哨兵j所指向的元素的值。交换之后的序列如下。

       6  1  2  **5**  9 3  4  **7**  10  8

 到此，第一次交换结束。接下来开始哨兵j继续向左挪动（再友情提醒，每次必须是哨兵j先出发）。他发现了4（比基准数6要小，满足要求）之后停了下来。哨兵i也继续向右挪动的，他发现了9（比基准数6要大，满足要求）之后停了下来。此时再次进行交换，交换之后的序列如下。

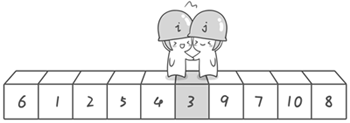
       6  1  2 5  **4**  3  **9**  7 10  8

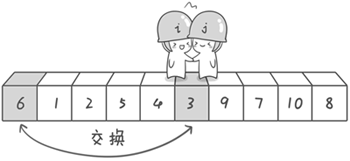


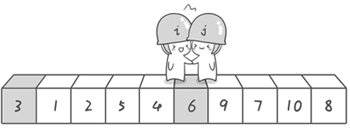


 第二次交换结束，“探测”继续。哨兵j继续向左挪动，他发现了3（比基准数6要小，满足要求）之后又停了下来。哨兵i继续向右移动，糟啦！此时哨兵i和哨兵j相遇了，哨兵i和哨兵j都走到3面前。说明此时“探测”结束。我们将基准数6和3进行交换。交换之后的序列如下。

**3**  1 2  5  4  **6**  9 7  10  8







 到此第一轮“探测”真正结束。此时以基准数6为分界点，6左边的数都小于等于6，6右边的数都大于等于6。回顾一下刚才的过程，其实哨兵j的使命就是要找小于基准数的数，而哨兵i的使命就是要找大于基准数的数，直到i和j碰头为止。

OK，解释完毕。现在基准数6已经归位，它正好处在序列的第6位。此时我们已经将原来的序列，以6为分界点拆分成了两个序列，左边的序列是“3  1 2  5  4”，右边的序列是“9  7  10  8”。接下来还需要分别处理这两个序列。因为6左边和右边的序列目前都还是很混乱的。不过不要紧，我们已经掌握了方法，接下来只要模拟刚才的方法分别处理6左边和右边的序列即可。现在先来处理6左边的序列现吧。

左边的序列是“3  1  2 5  4”。请将这个序列以3为基准数进行调整，使得3左边的数都小于等于3，3右边的数都大于等于3。好了开始动笔吧。

如果你模拟的没有错，调整完毕之后的序列的顺序应该是。

       2  1  **3**  5  4

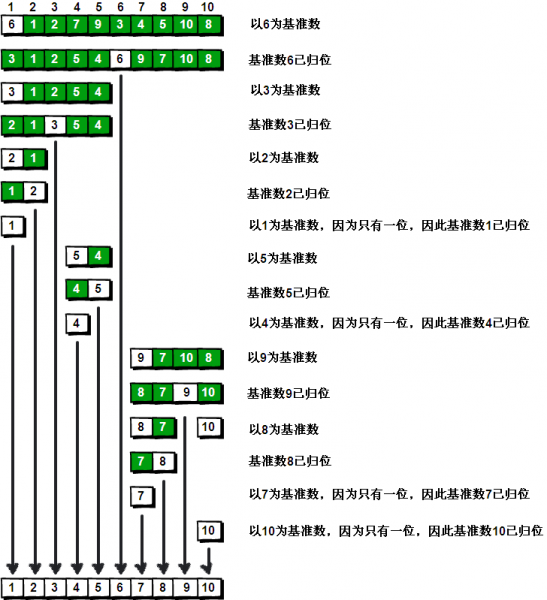
OK，现在3已经归位。接下来需要处理3左边的序列“2 1”和右边的序列“5 4”。对序列“2 1”以2为基准数进行调整，处理完毕之后的序列为“1 2”，到此2已经归位。序列“1”只有一个数，也不需要进行任何处理。至此我们对序列“2 1”已全部处理完毕，得到序列是“1 2”。序列“5 4”的处理也仿照此方法，最后得到的序列如下。

       1  2  3 4  5  6 9  7  10  8

对于序列“9  7  10  8”也模拟刚才的过程，直到不可拆分出新的子序列为止。最终将会得到这样的序列，如下。

       1  2  3 4  5  6  7  8 9  10

到此，排序完全结束。细心的同学可能已经发现，快速排序的每一轮处理其实就是将这一轮的基准数归位，直到所有的数都归位为止，排序就结束了。下面上个霸气的图来描述下整个算法的处理过程。



程序:

#include <stdio.h>

int partition(int \* a, int i, int j);

void sort(int \*a, int i, int j);

int main()

{

int a[] = {-1,-3,0,5,-88,16,28,-55}, i, n;

n = sizeof(a) / sizeof(int);

sort(a, 0, n-1);//第二个参数表示第一个元素的下标，

//第三个参数表示最后一个元素的下标

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

printf("%d ", a[i]);

printf("\n");

return 0;

}

void sort(int \*a, int i, int j)

{

int pos;

if (i >= j)

return;

pos = partition(a, i, j);

sort(a, i, pos-1);

sort(a, pos+1, j);

}

int partition(int \* a, int i, int j)

{

int key = a[i];

while (i < j)

{

while (i < j && a[j] >= key)

j--;

a[i] = a[j];

while (i < j && a[i] <= key)

i++;

a[j] = a[i];

}//终止while循环之后i和j一定是相等的

a[i] = key;

return i; //i可以改为j,但不能改为key,也不能改为a[i]、a[j]

}

结果:

