# 数据结构

数据结构是什么，数据结构其实就是对数据的存储和操作的一门学问

数据结构对个体存储研究不大，主要是对个体存储之间的研究

# 大o表示法

由于本人对大o表示法的研究还是不算太深，就只能稍作介绍，大O表示法：称一个[函数](http://baike.baidu.com/subview/15061/18968664.htm)g(n) 是O(f(n))，当且仅当存在常数c>0和n0>=1，对一切n>n0均有|g(n)|<=c|f(n)|成立，也称函数 g(n)以f(n)为界或者称g(n)囿于f(n)。记作g(n)=O(f(n))。 定义：如果一个问题的规模是n，解这一问题的某一算法所需要的时间为T(n)，它是n的某一函数。T(n)称为这一算法的“[时间复杂度](http://baike.baidu.com/subview/104946/104946.htm)”。当输入量n逐渐加大时，时间复杂度的极限情形称为算法的“渐近时间复杂度”

我们可以用大o表示法来估算程序的运行时间复杂度

主要宗旨是找程序中最大的

比如

For（i = 0;I < n;i++）

A = b + c;

这样程序a=b+c是一个时间单元，然后循环迭代是n-1个时间单元；，I = 0是一个时间单元，i++是n个时间单元

所以这个运行时间是O(n)

如果是ifelse这种选择语句就是运行时间较长的时间单元来表示运行时间复杂度

然后for的嵌套是两个参数相乘

最后像那种对半这种寻找的是logN

# 线性表之顺序存储

当我们使用对数组的增加成员或者减少成员的时候我们就需要对这个数组进行操作，我们的数组拥有下标，所以说是顺序存储但是我们如果直接操作数组的话会很麻烦，我们一般会定义一个结构体，中间成员一个是该数组，另外一个是记录它的个数的

struct nd{

datatype data\_arr[DATASIZE];

int last;

};

typedef struct nd sqlist;

我们初始化一般都是用malloc去开辟一个空间，然后将last初始化为-1，这样就代表现在里面是没有数的

sqlist \*sqlist\_create(void)

{

sqlist \*me;

me = malloc(sizeof(\*me));

if(me == NULL)

{

return NULL;

}

me->last = -1;

return me;

}

我们接下来就开始对这个数组进行操作

首先我们看看需要几个参数

1. 需要操作的结构体
2. 对数组插入的位置
3. 需要插入的数据

这样我们就开始进行操作，在这之前我们需要检测一下操作的这个位置是否已经越界以及目前被操作的数组是否已经满了

if(me->last >= DATASIZE - 1)//判断下标是否已经满了

{

return -1;

}

if(i < 0 || i > me->last+1) //判断插入的位置是否小于0以及是否大于现在数组再加一的位置，因为我们可以在原数组的最后一位添加数据所以是加一

{

return -2;

}

如果上面都没有发生的话就代表这个位置是可以操作的

for(j = me->last;i <= j;j--)

{me->data\_arr[j+1] =me->data\_arr[j];} //如果原数组不是空的话就会进行这一步操作，将原来该位置的数字向后移动

me->data\_arr[i] = \*data; //将需要插入的数字放到该位置上

me->last++;//因为插入了一个，所以个数加一

int sqlist\_insert(sqlist \*me,int i,datatype \*data)

{

int j = 0;

if(me->last >= DATASIZE - 1)

{

return -1;

}

if(i < 0 || i > me->last+1)

{

return -2;

}

for(j = me->last;i <= j;j--)

{me->data\_arr[j+1] =me->data\_arr[j];}

me->data\_arr[i] = \*data;

me->last++;

return 0;

}

有插入就有从哪个位置删除

我们看一下删除函数

我们需要的参数是

1. 需要操作的结构体
2. 需要删除的位置

同样的，我们需要先检测这个位置是否可以操作

if(i < 0 || i > me->last) //如果传入的位置小于0或者大于现有的数组的个数就报错

return -1;

我们检测完了就可以进行操作了，

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

me->data\_arr[j-1] = me->data\_arr[j]; //因为我们要从中间某个位置删除，所以之后的位置都要向前移动所以是后面的给前一个赋值

me->last--;//因为删除了一个所以减一

int sqlist\_delete(sqlist \*me,int i)

{

int j;

if(i < 0 || i > me->last)

return -1;

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

me->data\_arr[j-1] = me->data\_arr[j];

me->last--;

return 0;

}

我们有几个检测函数

int sqlist\_getnum(sqlist \*me) //获取这个数组一共有多少个元素

{

return (me->last+1);

}

int sqlist\_setempty(sqlist \*me) //将数组置空

{

me->last = -1;

return 0;

}

int sqlist\_isempty(sqlist \*me) //查看这个数组是否为空

{

if(me->last == -1)

return 0;

return -1;

}

int sqlist\_destory(sqlist \*me) //注销这个结构体

{

free(me);

return 0;

}

in

我们有查找函数

有两个参数，一个是操作的结构体另外一个是需要查找的数字

我们还是要先检测，我们先看看这个数组是否为空

if(sqlist\_isempty(me) == 0) //如果返回0就代表是空

return -1;

然后就开始逐个对比

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

if(me->data\_arr[i] == \*data) //如果在循环内找到与data相同的数字就返回现在的下标如果在循环内没有就返回-1报错

return i;

return -1;

int sqlist\_find(sqlist \*me,datatype \*data)

{

int i;

if(sqlist\_isempty(me) == 0)

return -1;

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

if(me->data\_arr[i] == \*data)

return i;

return -1;

}

我们也可以将两个数组合并，如果有一样的就略过，有不一样的就加上、

这个函数没有什么预备工作，我们就直接开始对比

int i = 0;

for(i = 0;i < me2->last;i++)//在第二个数组的范围内

{

if(sqlist\_find(me1,&me2->data\_arr[i]) < 0)//如果有两个不一样的数

{

sqlist\_insert(me1,0,&me2->data\_arr[i]);//将第二个数组内的数加到第一个数组中

}

}

int sqlist\_union(sqlist \*me1,sqlist \*me2)

{

int i = 0;

for(i = 0;i < me2->last;i++)

{

if(sqlist\_find(me1,&me2->data\_arr[i]) < 0)

{

sqlist\_insert(me1,0,&me2->data\_arr[i]);

}

}

return 0;

}

**不管在写什么程序的时候，在我们写一个模型的时候有必要把一个模型的完整性体现出来，也就是说当一个模型只有一个元素的时候我们也要保证它的完整性**

# 链表

当我们写链表的时候一定要注意我们现在操作的指针是否为NULL

无论何时只要你确定一个指向，那么你就必须抱着你这个指针不是NULL

还有就是我们声明出来的结构体指针并不是分配了一个结构体，它只是一个指针，但是内存会给它一个足够大的空间让它去指向，一般都是和这个结构体的大小是相等的

当我们写完一个链表时如果我们的链表没有插入或者删除的话我们的malloc的次数应该和表的大小是相等的

当我们写一个比较需要空内存很大的一个东西的时候，如果系统没有地方来存放了这时候就会返回NULL指针

Malloc函数分配的东西并不是这个结构体只是跟这个结构体所占的空间大小一样罢了

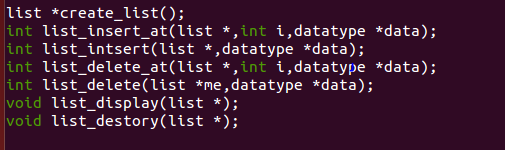
## 单向有头链表

首先我们要说一下链表的概念，首先链表的存储形式是链式存储，它的特点就是存取容易方便

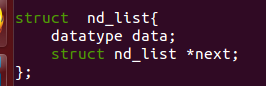
链表是基于结构体之上的，然后在结构体的成员中再加上一个该结构体的指针作为下一个同样的结构体的入口，这样一个接着一个就像一个链子连接着，所以叫链表

链表分好多种，现在这种是单向有头链表，单向，有头

我们接下来看一下单向有头链表的一些基本函数

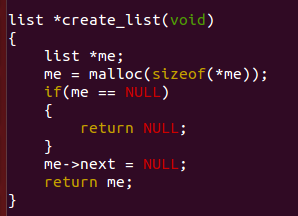


我们先看一下链表的定义



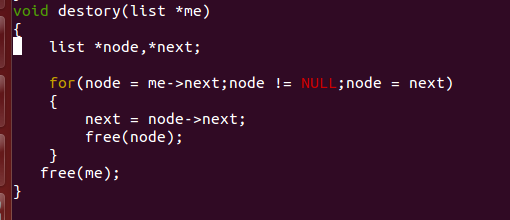
data是我们的链表中一个节点的成员，然后next是下一个链表节点的入口

我们先来看看创建一个链表，创建一个链表跟上面的创建一个顺序存储的结构体相差不大



利用malloc分配一个分区，然后暂时地将next置空，最后返回一个地址值

然后删除和顺序存储有点差别，我们需要把每个节点都需要释放掉（free）



首先利用for循环寻找next节点，找到一个就释放一个，最后将这个链表的头部释放

然后是插入

当我们需要将一个数据插入到这个链表的时候，我们就首先要分配一个同样的结构体然后将需要插入的位置的前一个位置的next地址给新分配的这个结构体，然后将这个结构体的地址赋给上一个结构体的next节点，这样就构成了插入，这里要提醒一下的是，这里是单向链表，也就是说走到后面一个结构体的时候只能找到下面一个结构体的地址，找不到上面一个的地址，如果要找只能重来，如果我们插入的时候找到的是需要插入的下一个地址，那么我们嵌入的时候就会找不到上一个节点的地址，这样的话链表就会断了

int list\_insert\_at(list \*me,int i,datatype \*data)

{

int j = 0;

list \*node = me,\*newnode = NULL;

while(j < i && node != NULL)

{

node = node->next;

j++;

}

if(node)

{

newnode = malloc(sizeof(\*newnode));

if(newnode == NULL)

{

return -1;

}

newnode->data = \*data;

newnode->next = node->next;

node->next = newnode;

return 0;

}

else

return -2;

}

这个是在制定的位置插入链表的，我们定义一个整型变量用来计数，然后到了那个位置的节点就开始进行嵌入链表

int list\_intsert(list \*me,datatype \*data)

{

list \*p = me,\*q;

while(p->next && p->next->data < \*data)

{

p = p->next;

}

q = malloc(sizeof(\*q));

q->next = p->next;

p->next = q;

q->data = \*data;

return 0;

}

这个是顺序插入，也就是我们比较一下这个要插入的数据跟原有链表中的数据的大小，直到碰到比它大的就在它之前插入

int list\_isempty(list \*me)

{

if(me->next == NULL)

return 0;

return 1;

}

void list\_display(list \*me)

{

list \*node = me->next;

if(list\_isempty(me) == 0)

return ;

while(node != NULL)

{

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

node = node->next;

}

printf("\n");

// return 0;

}

上面的那个函数是检测链表是否为空，如果为空就返回0，不为空就返回1

下面那个函数是显示函数，就是把每个节点的东西都打印出来

接下来是删除某个节点

int list\_delete\_at(list \*me,int i,datatype \*data)

{

int j = 0;

list \*p = me,\*q;

// data = NULL;

if(i < 0)

{

return -1;

}

while(p->next && j < i)

{

j++;

p = p->next;

}

if(p)

{

q = p->next;

p->next = q->next;

\*data = q->data;

free(q);

q = NULL;

return 0;

}

else

return -2;

}

这是删除某个节点的函数，性质跟插入某个节点差不多，轮着走然后找到那个节点然后释放那个空间然后把节点和后面的接上

int list\_delete(list \*me,datatype \*data)

{

list \*p = me,\*q;

while(p->next && p->next->data != \*data)

{

p = p->next;

}

if(p->next == NULL)

{return -1;}

else

{

q = p->next;

p->next = q->next;

free(q);

q = NULL;

}

}

这是删除某个数据，一样的，就是轮着找如果有这个数据就删掉如果没有就报错

## 单向无头链表

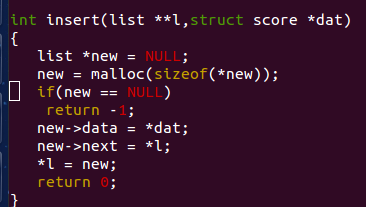
我们先来体会一下无头链表的意义，无头链表也就是第一个结构体直接放入了我们要存储的数据，而有头链表是第一个不放，这样各有各的好处

有头链表虽然比无头的多占用了一个空间，但是操作简单，也就是对第一个节点的操作无需特殊处理，跟后面的节点一样处理就好了，但是无头就不行了，因为我们要在第一个节点增减的话就需要单独处理，因为如果跟后面的一样处理的话那么这个链表就会断

但是无头链表可以轻松地将两个链表连接到一块，因为它没有头，可以直接当做中间的节点

我们来看一下它的一些简单代码的实现

首先，因为是无头的，从第一个节点开始就是有效数据，所以我们不必向有头链表似的需要首先创建一个头，所以没有create函数，我们直接就有插入函数，创建一个无头链表仅仅需要定义一个该结构体的指针就好了

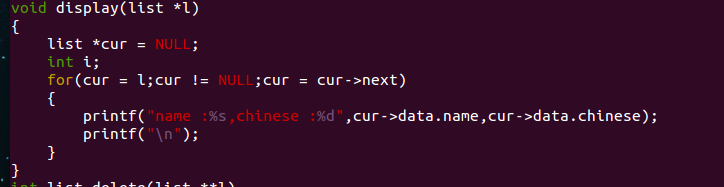


这里我们用一个二级指针来接收这个链表的结构体，这样的好处是操作的直接就是这个结构体的地址，并不是像一级指针的形参一样需要返回指针值，这样我们的返回值可以用作判断是否执行成功了

我们还需要定义一个结构体指针，因为我们需要插入一个结构体，所以首先需要创建一个结构体，这里从头部插入有所不同，也就是说我们之前的是把需要插入的位置的前面一个节点的next给到现在需要插入的结构体的next中，然后把这个结构体的地址赋值给前面一个节点的next

而现在的这个无头链表如果需要在第一个节点插入的时候前面是没有节点的如果按照之前的方法的话我们就会找不到后面的节点了，我们就需要把后面的地址首先赋值给新的next，然后再把新的地址赋值给原来的指针，这样就是将新来的节点当做第一个了，如果先把新结构体的地址赋值给原来的指针的话，那么后面的链表就丢了

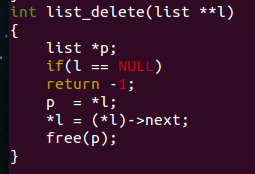
接下来是显示结构体的函数



没什么好说的，就是从头到尾地遍历就好了

然后是删除

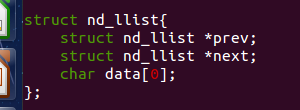
我们删除需要将需要删除的那个结构体也就是free掉，但是我们不能直接free，如果直接free的话就会将现在的第一个节点的地址free掉，这样的话我们就找不到这个链表了，所以我们需要先将地址赋值给别的指针，然后再将原来我们的链表指向的这个指针指向它的next，然后我们free掉那个指针就好了



## 双向循环链表

我们单向链表会了，双向链表就不难了，说白了是将结构体中加入了一个prev指针，它的类型跟next是一样的，只不过就是上一个节点的入口，由于这个入口的增加它的性能会增加很多

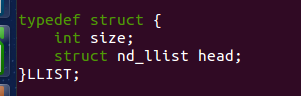
首先我们看一下它是怎么定义的



这里这个char data[0]需要解释一下

实际上这么回事，这样做的话其实就是设置了一个可伸缩的结构体，因为我们定义的char data[0]其实是不占用空间的，但是这么设置的话就会将这个结构体后面的字节数标记了起来，当我们给它赋值的时候就不用先malloc了，因为我们是定义的是一段空间，我们没有必要用动态存储，如果想用，就可以将data设置为void \*data，这样我们就需要malloc来给它分配空间了，当然，在初始化的时候我们的头节点的data需要将它置空，也就是NULL，然后释放或者删除的时候也需要将data这段空间首先释放掉

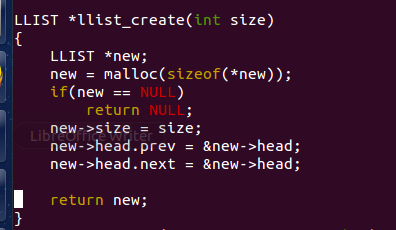
然后我们来看一下它的头结构体怎么定义的



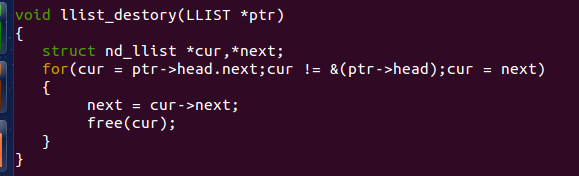
首先这个size是给用户的，因为我们做好的头文件是不知道用户的结构体的大小的，这需要用户来提供信息来让我们知道用户的结构体的大小，下面是我们上面的链表的结构体的定义，这就是我们的头节点

然后我们看一下怎么创建链表的，我们第一次创建肯定就只是创建一个节点，也就是头节点，因为是循环链表，所以要创建的时候需要将它们的前驱指针以及后继指针都指向第一个节点的地址，但是我们需要注意一点，这里并不是指向头节点，而是指向头节点中的head结构体，这样才能够保证链表的完整

我们首先要给它分配一段空间



然后对应创建就会有删除



这里就是需要将这个链表遍历一遍然后每走到一个就要释放一个

然后是插入函数

我们这里插入跟单向链表有所不同，因为我们不仅是将next指针操作，我们还需要将需要插入的前驱和后继的prev指针也需要连接上，如果这要是按照我们原来的思想多定义几个指针的话很有可能就会被绕晕，我们这里就用另外一个方法，这个办法也是体现了双向链表的特点，我们不需要定义很多个指针，因为我们的双向链表的任意一个节点都可以找到任何一个位置，因为它的前面节点的地址以及后面节点的地址都有

我们插入一个节点时应该需要知道它插入的节点的前驱节点或者是后继节点，我们是首尾插入所以首先要把这个前驱节点的next指针给到头节点然后把这个新的节点的地址给头节点的next，再将新的节点的prev赋值头节点的地址，然后就是体现双向链表特性的时候了，我们可以将新的节点的next入口的prev赋值成自己的地址这样就可以将后面的节点连接到自己上，然后再将自己的prev入口的next指向自己也是同样的道理，反过来插入到尾部也是一样的





这样就可以了

int llist\_insert(LLIST \*me,const void \*data,int mode)

{

struct nd\_llist \*newnode = NULL;

LLIST \*p = NULL;

p = me;

newnode = malloc(sizeof(struct nd\_llist) + me->size);

if(newnode == NULL)

return -1;

memcpy(newnode->data,data,me->size);

if(mode == FORWARD)

{

newnode->next = p->head.next;

newnode->prev = &p->head;

}

else if(mode == BEHIND)

{

newnode->prev = p->head.prev;

newnode->next = &p->head;

}

else

{

return -3;

}

newnode->next->prev = newnode;

newnode->prev->next = newnode;

return 0;

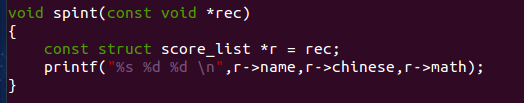
}

接下来我们就要写一个显示函数，也就是说显示整个链表，这里跟以前写的有所不同，我们需要写一个回调函数

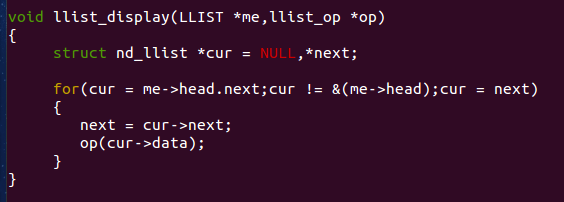
因为我们并不知道用户要用的结构体的样子，我们并不知道用户结构体中的成员是什么，所以我们需要让用户来提供成员是什么，也就是说用一个回调函数，让用户写好的printf函数传到我们这里，然后我们调用这个回调函数就可以了，我们需要typedef的一个这个函数的类型



这个就是我们的回调函数的类型，然后我们让用户按照这个来写一个回调函数



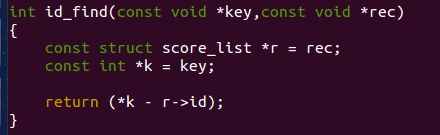
这样就可以不需要我们知道结构体是什么就可以通过用户打印出来内容了



然后是查找函数

这里查找函数也同样的需要一个回调函数，也就是说需要用户来告诉我们那里面有什么元素，因为我们并不知道用户的结构体是什么样子的

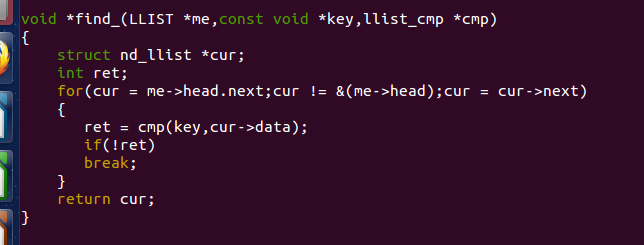
我们来看一下用户的回调函数是怎么写的



这样就可知找到结构体了，然后我们也需要tyoedef一个抽象的函数类型

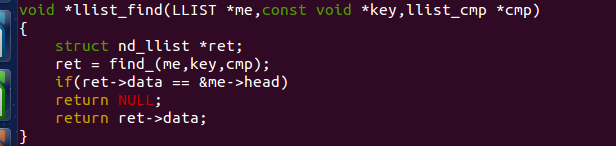


这样就可以用这个当做这个函数的类型名来传递这个函数的指针了

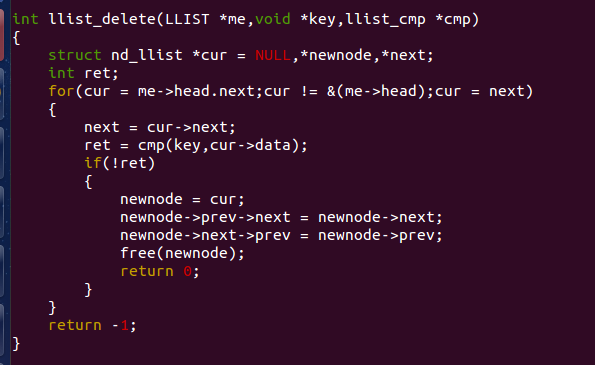


这样就表示如果找到了就为0，

然后

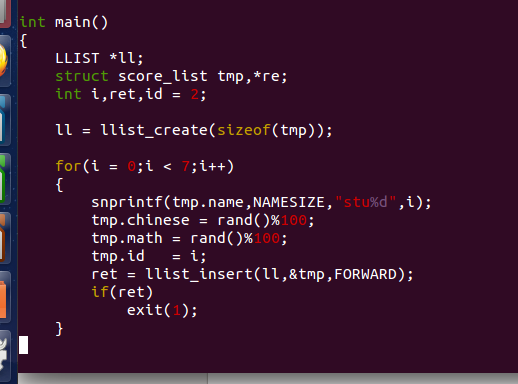


，接下来是我们的删除指定元素的函数



使用for循环遍历一下然后如果找到了就把这个节点拿出去释放掉就可以了

这样我们的函数就大致写完了



# 栈的使用

栈就是一个限制插入和删除只能在一个位置的表，因为栈是一个表，所以任何一个实现表的方法都可以实现栈

## 栈的顺序存储

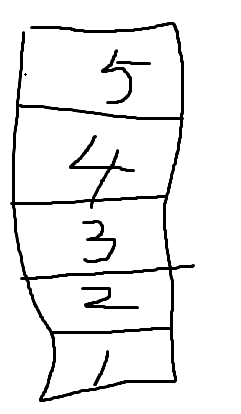
我们首先要了解一下栈是怎么用的，栈作为一种[数据结构](http://baike.baidu.com/view/9900.htm)，是一种只能在一端进行插入和删除操作的特殊[线性表](http://baike.baidu.com/view/178622.htm)。它按照先进后出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。栈具有记忆作用，对栈的插入与删除操作中，不需要改变栈底[指针](http://baike.baidu.com/view/159417.htm)。

我们也可以通过栈来保存现场，比如说我们上面使用到的递归问题就是利用栈的保存现场来做的

它还可以传递参数

我们这次利用栈的存储特性来对栈进行研究

首先我们要知道栈的存储特性，实行先进后出的存储方式

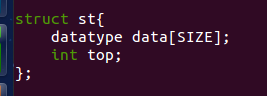


如图所示，首先存入的放在最底部，然后一个一个往上加，然后出栈的时候是从5开始出，最后出1，所以说是先进后出

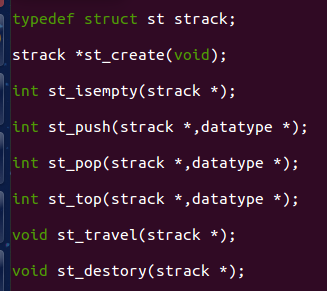
我们可以做一个顺序存储的例子来进一步地了解栈

我们首先跟原来一样创建一个c文件一个h文件一个main.c

首先我们应该定义一个栈的结构体

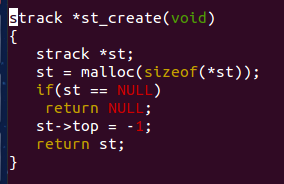


其中data数组是我们存储数据的地方，然后top代表现在栈的位置，一般都是赋值为-1

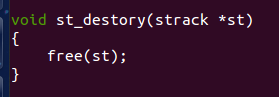


这些是我们需要实现的功能函数，然后我们在.c文件中实现一下

首先是创建函数，这个就是malloc一个结构体大小的空间，然后对top进行初始化

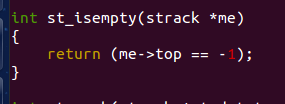


然后是删除这个结构体，这个比较简单直接free掉结构体就行了



然后是检测是否是空栈函数

直接返回一个top是否等于-1的一个判断语句



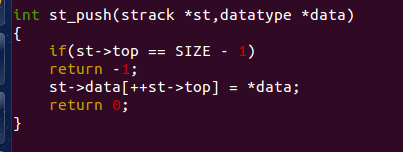
这样返回判断的如果是空栈top就等于-1，这样就为真，要么就是假

然后是我们的插入和删除，在栈中比较习惯用push和pop

我们就先来看插入是怎么实现的push

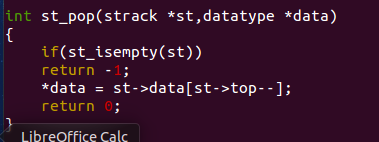
其实并不难，只不过是首先要判断是否是满栈，如果是就return不是就将数据

存入栈中并++top

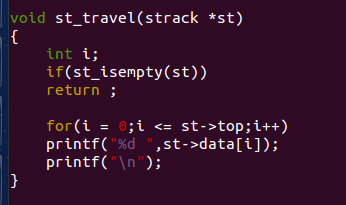


我们再来看pop函数

也不是很难就是将它递减就可以了



最后还有个显示函数就是来一个for循环



然后主函数进行插入删除等测试操作

int main()

{

datatype arr[] = {12,13,0,45,323};

strack \*st;

int i,ret;

datatype tmp;

st = st\_create();

if(st == NULL)

exit(1);

for(i = 0;i < sizeof(arr)/sizeof(\*arr);i++)

{

st\_push(st,&arr[i]);

}

st\_travel(st);

while(st\_pop(st,&tmp) == 0)

{

printf("pop:%d",tmp);

}

printf("\n");

/\*

if(ret != 0)

printf("push filed\n");

else

st\_travel(st);

\*/

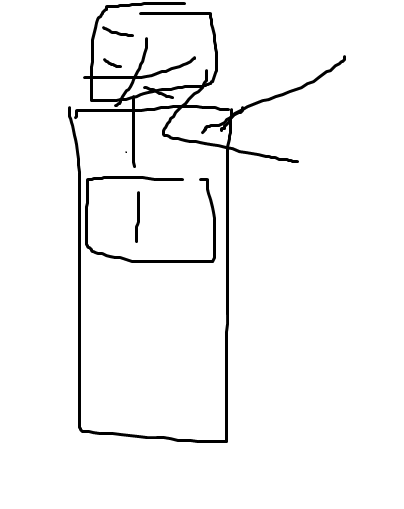
st\_destory(st);

exit(0);

}

## 栈的链式存储

上面我们说了栈的顺序存储，这回我们说说栈的链式存储，我们上面说的顺序存储有一些缺陷，就是它的占用空间有可能有限还有可能出现浪费，这是顺序存储的通病，这回我们来看一下链式存储

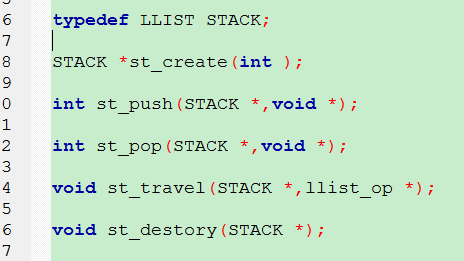


这个图是我们的双线有头循环链表，外面画的框代表着栈，跟链表一样，入栈就是将头节点跟头节点下面一个节点插入一个新的节点，出栈就是将头节点后的第一个节点删除

我们来看一下代码

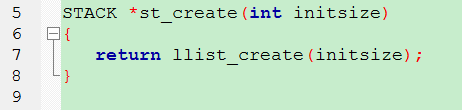
我们这里需要使用之前的双向链表的代码进行二次封装

我们创建一个.c和.h和main.c文件还有复制之前的链表的代码，然后我们先看一下.h的代码



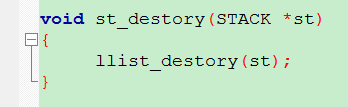
我们需要实现这几个代码，我们这里就是调用双向链表的函数

首先是创建一个栈结构体

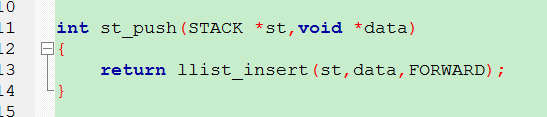


实际上其实就是创建一个双向环链 一个头节点，所以我们可以直接调用双向链表的创建函数

紧接着是删除函数



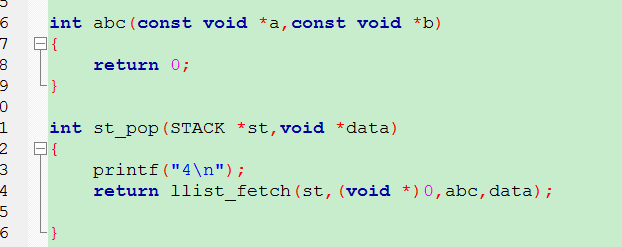
然后是push函数



因为是在头节点后面插入，这跟我们之前写的双向链表插入函数中断FORWORD模式是一样的，所以我们直接调用insert函数就可以了模式直接设置成FORWARD就行了

然后是pop函数，我们要求的是删除栈顶元素需要知道删除的数值

我们之前写的fetch函数是这个功能，但是fetch函数是删除选定的位置的节点，所以我们要在参数那边做点手脚，我们可以这么写



这样就是删除头节点之后第一个节点的元素了

然后下面是主函数代码

#include "strack.h"

#define NAMESIZE 24

struct nd\_score{

char name[NAMESIZE];

int id;

int math;

int chinese;

};

static void print\_s(void \*nd)

{

struct nd\_score \*n = nd;

printf("id:%d name:%s chinese:%d math:%d",n->id,n->name,n->chinese,n->math);

}

int main()

{

struct nd\_score nd;

STACK \*st;

int i;

st = st\_create(sizeof(struct nd\_score));

if(st == NULL)

exit(1);

printf("1\n");

for(i = 0;i < 6;i++)

{

nd.id = i;

snprintf(nd.name,NAMESIZE,"stu%d",i);

nd.math = rand()%100;

nd.chinese = rand()%100;

st\_push(st,&nd);

}

int ret;

printf("2\n");

while(1)

{

ret = st\_pop(st,&nd);

if(ret == -1)

break;

printf("3\n");

print\_s(&nd);

}

// st\_travel(STACK \*,llist\_op \*);

st\_destory(st);

exit(0);

}

# 队列

## 队列的顺序存储

首先我们应该理解队列的意思，队列其实在我们的生活中就有很多个例子，比如我们排队我们先排的先到

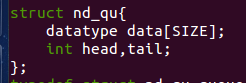
这就是队列跟栈不一样的地方，就是栈是先进后出，也就是1234进去的出来就是4321，但是队列就是先进先出，就是1234进去出来也是1234

在队列的顺序存储我们需要两个量，也就是头和尾两个变量，最开始的时候是两个变量都是表示的同一个位置，然后插入的时候尾变量就向后移动一位，取出的时候头变量就向后移动一位，但是我们这么做会有麻烦，就是如何判断队列的空和满，所以我们规定头指定的位置不能存放数据，这样就能判断是否为空是否为满了，也就是说当尾等于头部的时候就是空，如果尾的下一个位置就是头那么就代表已经满了

我们来看一下代码

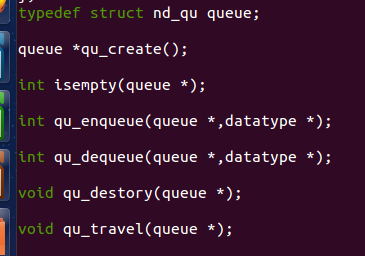
我们先创建一个.c文件和一个.h文件和一个mian.c

然后我们看一下.h文件



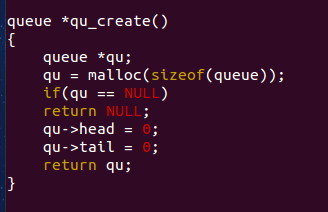
这是我们队列使用的结构体，一个头一个尾，还有一个存储数据的数组

然后接下来是我们要实现的一些函数

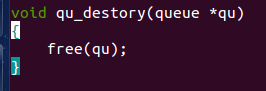


我们回到.c文件中来一个一个看程序

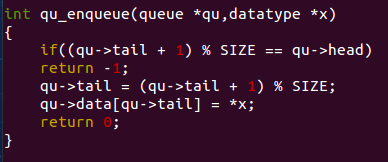
我们先把head和tail都初始化为0



然后是删除

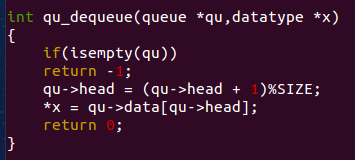


接下来是插入，插入的最开始需要检测一下这个队列是否满员，就是将tail+1检测是否等于头就可以了

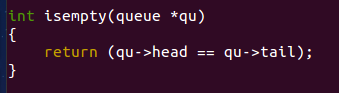


然后我们的tail增加不能只是单单的tail++，要检测是否已经加到最后了，如果加到最后就需要跳到第一个

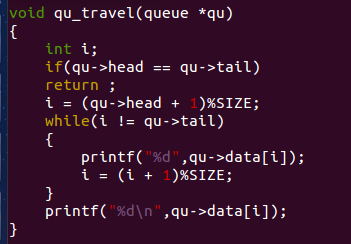
然后是取出数据，这个取出数据是要操作head的因为它是从头开始的



在删除之前还要检测一下是否为空



还有一个就是显示函数了



最后是main.c

#include "queue.h"

int main()

{

queue \*qu;

datatype arr[] = {1,2,3,4};

int i,ret;

qu = qu\_create();

if(qu == NULL)

exit(1);

for(i = 0;i < sizeof(arr)/sizeof(\*arr);i++)

{

qu\_enqueue(qu,&arr[i]);

}

datatype tmp = 1;

/\*

ret = qu\_enqueue(qu,&tmp);

if(ret == -1)

printf("insert failed\n");

else

\*/

qu\_travel(qu);

qu\_dequeue(qu,&tmp);

printf("DELETE : %d\n",tmp);

qu\_travel(qu);

qu\_destory(qu);

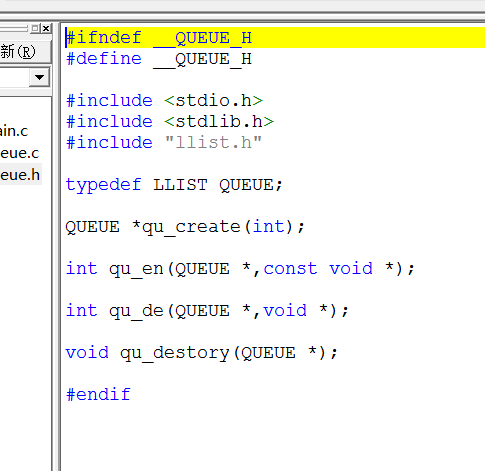
exit(0);

}

## 队列的链式存储

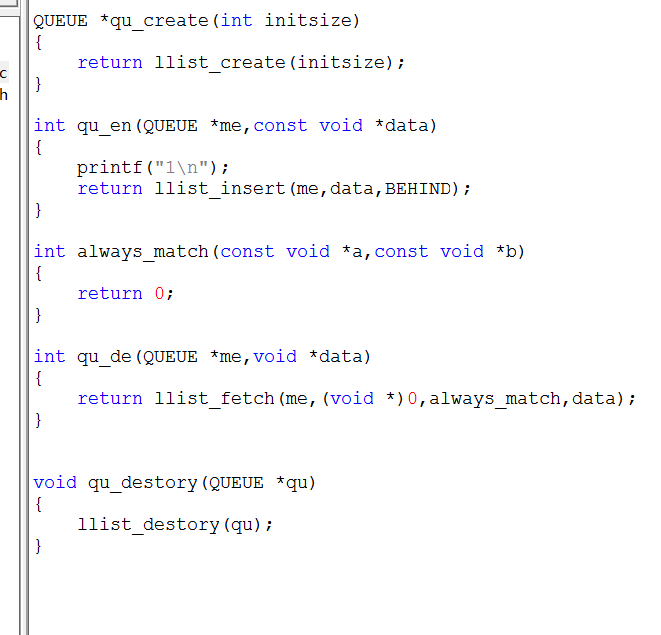
我们来说一下队列的链式存储，链式存储跟上面的栈是一样的，唯一的不同的就是这次链表的插入是在尾部插入的仅此而已

我们来看一下代码



首先这是头文件

然后我们写c文件来完成头文件的那些功能



这些就是c函数，跟栈的链式存储大致的写法是一样的，在此就不再赘述了

下面是主函数的代码

#include "queue.h"

#define NAMESIZE 24

struct nd\_score{

int id;

char name[NAMESIZE];

int math;

int chinese;

};

static void print\_s(void \*qu)

{

struct nd\_score \*r = qu;

printf("id:%d,name:%s,math:%d,chinese:%d\n",r->id,r->name,r->math,r->chinese);

}

int main()

{

struct nd\_score tmp;

QUEUE \*qu;

int i,ret;

qu = qu\_create(sizeof(\*qu));

if(qu == NULL)

exit(1);

for(i = 0;i < 6;i++)

{

tmp.id = i;

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

tmp.math = rand()%100;

tmp.chinese = rand()%100;

snprintf(tmp.name,NAMESIZE,"stu%d",i);

qu\_en(qu,&tmp);

}

while(1)

{

ret = qu\_de(qu,&tmp);

if(ret == -1)

break;

print\_s(&tmp);

}

qu\_destory(qu);

exit(0);

}

# KMP算法

Kmp算法是一种字符串匹配的一种高效率的算法，它是由三个外国人发现的，它能比我们正常使用的回溯要快很多

首先我们来说一下正常回溯是怎么干的

正常回溯就是将对比的字符串和模式串同时向后移动，如果有不一样的就会重新回到之前的下一个位置继续比，这样效率会很低，因为产生了很多不必要的开销，

这些开销是很恐怖的

而kmp算法是将比较的字符串固定不动，将模式串移动进行比较，如果有不一样的会有一个计算方式来算出next数组来记录模式串需要移动多少位，接下来我们来谈谈next数组是怎么算出来的，首先我们需要了解一下前缀和后缀的意义，前缀就是除了字符串的最后一个字符之外的组合，后缀就是除了第一个字符

之外所有字符的组合

以abcd为例，前缀就是a，ab，abc，后缀 bcd，cd，d

然后我们需要根据前缀和后缀算出部分匹配值，

这个我们得一个一个算，比如说第一个字符的前缀和后缀都是什么，第一个字符的前缀前面没有东西，所以为空，后缀是bcd，cd，d，因为没有重合的前后缀，所以a为0

然后是b，前缀a，后缀cd，d也没有所以是0

C：前缀a，ab，后缀d，0

D，无后缀，所以也为0

所以应该是0000

然后移动的位置是根据已经匹配的字符个数 – 该位的部分匹配值就是它需要向后移动的个数

#include<stdio.h>

#include<string.h>

char x[1005],y[10005];

int lenx,leny,next[1005];

void get\_next()//找到特征数组

{

int i=0,j=-1;

next[0]=-1;

while(i<lenx)

{

if(j==-1||x[i]==x[j])

{

++i;++j;

next[i]=j;

}

else

{

j=next[j];

}

}

}

void kmp()//调用函数

{

get\_next();

int i=0,j=0,cnt=0;

while(i<leny)

{

if(j==-1||x[j]==y[i])

{

++i;++j;

if(j==lenx)//记录出现次数

{

++cnt;

}

}

else

{

j=next[j];

}

}

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

}

int main()

{

int t;

scanf("%d",&t);

getchar();

while(t--)

{

gets(x);gets(y);

lenx=strlen(x);leny=strlen(y);//长度比较好用

kmp();

}

}

其中核心部分代码主要是get\_next函数那个函数就是这个算法的核心部分，就是求next数组的

A b d d a

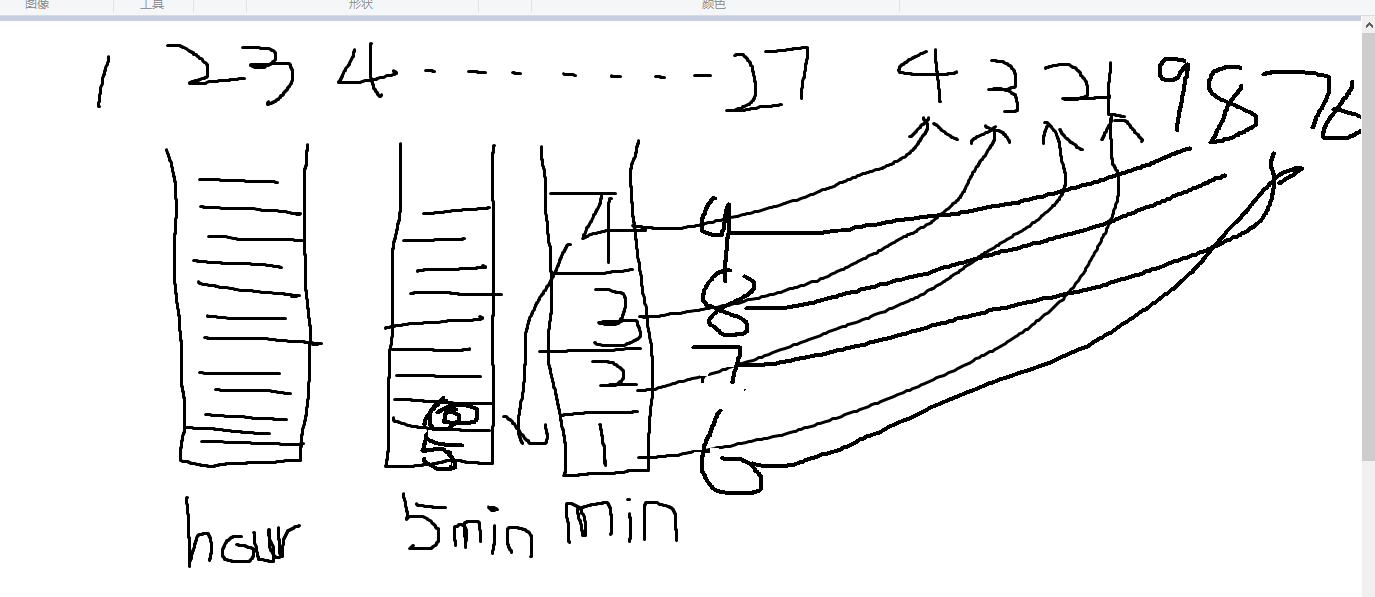
-1 0 0 1

Next数组的前两位分别是-1，和0，然后如果后面一位跟前一位一样那么就是0，如果不一样然后到d，这时j等于0，然后第二个d跟前面的一个d一样，那么j加一，然后d为1，然后到a，它跟d不一样，所以跳到d对应的数值的位置上，也就是1，也就是b，然后它跟b还是不一样，所以a就等于0

# 求中算法

这个是我们运用队列以及栈来灵活运用的一个算法，比如说现在有一个时间，从0开始计数，然后到12小时之后就清零，我们分三个容量槽，一个是1分钟的一个是5分钟的，还有一个是一小时的

我们往里面放入一些球，然后一分钟的槽放到第四个就清空溢出，然后5分钟进一个球，然后5分钟的槽进满12个就进到小时位然后这些清空的球的顺序就放进一个序列中，然后序列应该是乱的，但是直到有一个时间点是从1到27的，我们就是要求出这个的时间，这个就是求中算法



过程就如上图所示

然后我们看一下程序，这个程序是由三个栈以及一个队列组成的

我们用栈和队列的顺序存储



这个程序的思想就是如果min栈没有满那么就把球压栈否则满了那么就min栈出栈进队列然后再继续判断下一个5min栈是否满，没有满就把现在这个球压栈进去如果满了5min出栈进队列然后判断hour栈是否满如果没满压栈否则就出栈进队列在这个时候判断队列的顺序是否是1到27，如果是就打印时间，如果不是就继续循环这个过程下面是main.c的源代码，之后那几个的代码在上面都有写到

#include "queue.h"

#include "stack.h"

#define NUM\_BALL 27

static int checkqu(QUEUE \*qu)

{

int i = (qu->head + 1)%SIZE;

while(i != qu->tail)

{

if(qu->data[i] > qu->data[(i + 1)%SIZE])

return 0;

i = (i + 1)%SIZE;

}

return 1;

}

int main()

{

int i,time = 0;

QUEUE \*qu;

STACK \*st\_min,\*st\_fivemin,\*st\_hour;

type t;

datatype value;

qu = qu\_create();

if(qu == NULL)

{

printf("qu = NULL!\n");

exit(1);

}

st\_min = st\_create();

if(st\_min == NULL)

{

printf("st\_min = NULL!\n");

exit(1);

}

st\_fivemin = st\_create();

if(st\_fivemin == NULL)

{

printf("st\_fivemin = NULL!\n");

exit(1);

}

st\_hour = st\_create();

if(st\_hour == NULL)

{

printf("st\_hour = NULL!\n");

exit(1);

}

for(i = 1;i <= NUM\_BALL;i++)

enqueue(qu,&i);

qu\_travel(qu);

while(1)

{

dequeue(qu,&t);

time++;

if(st\_min->top != 3)

{

st\_push(st\_min,&t);

}

Else

{

while(!st\_isempty(st\_min))

{

st\_pop(st\_min,&value);

enqueue(qu,&value);

}

if(st\_fivemin->top != 10)

{

st\_push(st\_fivemin,&t);

}

else

{

while(!st\_isempty(st\_fivemin))

{

st\_pop(st\_fivemin,&value);

enqueue(qu,&value);

}

if(st\_hour->top != 10)

{

st\_push(st\_hour,&t);

}

Else

{

while(!st\_isempty(st\_hour))

{

st\_pop(st\_hour,&value);

enqueue(qu,&value);

}

enqueue(qu,&t);

if(checkqu(qu))

break;

}

}

}

}

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

qu\_travel(qu);

qu\_destory(qu);

st\_destory(st\_min);

st\_destory(st\_fivemin);

st\_destory(st\_hour);

exit(0);

}

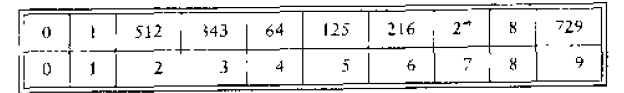
# 基数算法

基数算法是基于木桶算法的一个快速排序的算法

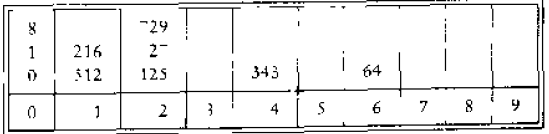
我们先了解一下这个算法

这个算法就是将一组数的每个位分别进行比较

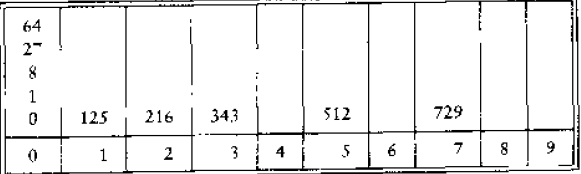
首先比一下每个数的个位数，然后将大小将之排列



这样我们的桶就出来了，然后我们开始比第二位数



因为这里面最大的位数是百位，所以我们需要再比一下第三位，如果没有第三位就当做零



这样我们就可以按顺序取出然后排列了

下面是源码

#include <stdio.h>

#include <stdlib.h>

int maximum;

void radix\_sort(int \*a,int size)

{

int temp[10][size];

int order[10] = {0};

int n;

int i,j,k;

int p;

n=1;

while(n <= 100)

{

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

{

k = (a[i]/n)%10;

temp[k][order[k]] = a[i];

order[k]++;

}

p = 0;

for(j = 0;j < 10;j++)

{

if(order[j] != 0)

{

for(i = 0;i < order[j];i++)

{

a[p] = temp[j][i];

p++;

}

order[j] = 0;

}

}

n = n\*10;

}

}

int main()

{

int i;

int a[10] = {100,112,59,60,1,0,456,789,234,684};

int \*b;

maximum = 100;

radix\_sort(a,10);

for(i = 0;i < 10;i++)

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

printf("\n");

exit(0);

}

## 快速排序

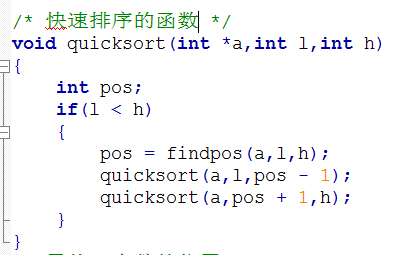
快速排序简称快排，也就是说它能非常快速的排序，快速排序也分很多种，但是大同小异，我在这里介绍一个比较快的快排

我们可以在一组数上面定义两个指针，一个指向开头，另外一个指向最后面，首先我们需要找一个数，首先要确定这个数的位置，然后把这组数分成两半，然后继续按照这种方法排序

找位置的方法就是首先让指向最后一个数做出判断，如果比我们最先定位的数大或者等于的话就向前移动，如果比这数小的话就和指向前面的给它赋值，赋值完之后就换成前面的指针移动，前面的指针是和后面的指针是相反的，如果碰到比基准数大的数就给后面指针所指向的位置赋值，如果比基准数小就向后移动，直到两个指针指向一个相同的地方的时候，我们把这个基准数的数值放到这个位置

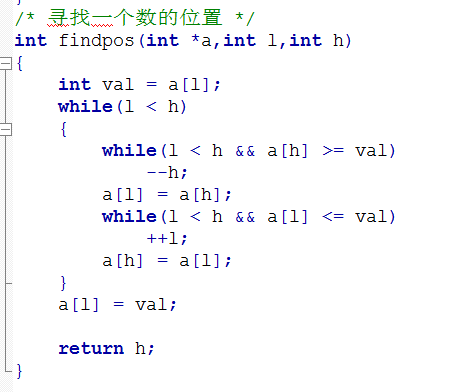
我们一般基准数就是第一个数

我们来看看程序是怎么实现的



首先这个是快速排序的函数，这里是一个递归，就是首先找到一个基准数据的位置，然后再去找这个位置的左边和右边。

我们来看一下这个找位置的这个函数



程序已经写的很明确了，就是跟上面说的是一样的

# 树

首先一棵树我们需要知道以下知识

一棵树是一些节点的集合，这个集合可以是空集

每一颗子树的根叫做根的儿子，r是每一颗子树的根的父亲

一棵树是N个节点和N – 1条边的集合，其中一个节点叫做根

除了根节点其它节点都有它的父亲

没有儿子的节点叫做树叶

深度：该节点到根的唯一路径的长，所以，根的深度为0

高：该节点到一片树叶的最长距离，也就是说树叶的高度为0，一棵树的高等于根的高，一棵树的深度等于这棵树的最长树叶的深度，这个深度等于这棵树的高

非终端节点就是非叶子节点

度：子节点的个数成为度，也就是说它有多少个孩子

一课数的度就是它的子节点最多的个数

如果存在n1到n2的一条路径那么n1是n2的祖先，n2是n1的后裔，如果n1 不等于n2那么n1就是n2的一位真祖先，n2就是n1的一个真后裔

一般树的存储有四种方法

孩子表示法

首先将树的这些节点存到数组里面去，其中每个节点的内容都存储它的父节点的下标，跟节点存为-1

双亲表示法

同样放在数组里面去，然后把每个节点的后代写成链表存进去

双亲孩子表示法

就是上述两个表示法融合

二叉树表示法

将一个普通树转换成二叉树的规则：左指针指向它的第一个孩子右指针指向它的兄弟

将一般树转换成二叉树是没有左子树的

森林的存储

一般森林的存储是转化成为二叉树，森林转换二叉树的方法跟一般树转化为二叉树的方法是差不多的，只不过就是将几个互不相交的数的根与根之间作为了兄弟

比如A树和B树

它们都有跟，要变成二叉树就要把A的第一个孩子变成左子树，但是A是根它没有兄弟所以这时候B就是它的兄弟，剩下的转化方式就跟一般树转化为二叉树是一样的了

## 二叉树

满二叉树就是二叉树的所有子数全满了，数目是2^k – 1 k代表的是二叉树的层数，每个二叉树的子树的分支的序号是左：2n右：2n + 1 n是这个分支的序号

### 树的三种遍历

树的三种遍历分别为先序遍历、中序遍历、后序遍历三种，如果我们把其中一种弄明白后面的两种就好说了

**先序遍历**

首先访问这颗树的跟节点

再先序遍历左子树

最后现需遍历右子树

首先是遍历根，然后再遍历跟的左子树，然而根的左子树本身也是一颗树，然后继续按照之前说的先去遍历它的根，遍历完成之后再去遍历这颗树的左子树，以此类推

如果遍历到最后一个左子树这个没有孩子了那么就会遍历这个孩子的父亲的右子树，右子树本身也是一个树然后遍历右子树的根，然后是这个右子树的左子树

直到这个右子树遍历完然后退回到上一个级别的右子树继续遍历，之后的过程就是一样的了

这个过程根递归是非常相像的

**中序遍历**

先中序遍历左子树

再访问根节点

最后访问右子树

中序遍历根先序遍历基本是差不多的，首先要中序遍历这棵树的左子树，也就是说到左子树之后还要找到左子树的左子树直到左子树为空，然后访问这棵树的

根，然后访问这棵树的右子树，这个小树访问完然后退出来访问上一级的根，然后继续访问这一级的右子树，以此类推，最后到真正这棵树的根之后，访问根

，然后用同样的方法来访问右子树就可以了

将一个树进行了一次中序遍历之后，我们得到的是排过序的结果

**后序遍历**

后序访问左子树

后序访问右子树

最后访问跟节点

后序遍历跟上面两个遍历方法是差不多的

我们如果想通过两种遍历方法还原出原来二叉树的样子，我们需要知道两种遍历方式而且必须是其中一个必须是中序遍历才可以退出来

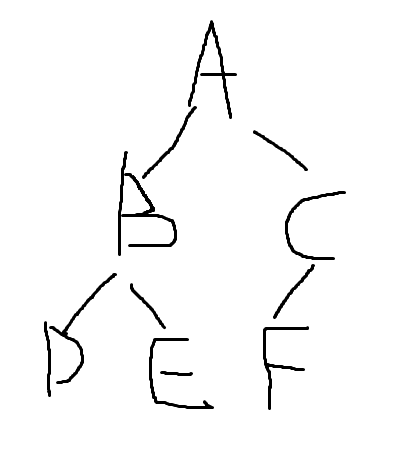
首先我们按照这些原则来推测就好了，但是经过很多年的研究表明没有中序是推测不出来的

我们先来看一下给你前序遍历和中序遍历来写这个树的后序遍历

前：ABDECF

中：DBEAFC

首先我们要看前序遍历的第一个，A在第一位也就是说A是这棵树的根然后我们看第二个元素，也就是B 前序遍历的原则就是先根然后左子树，后右子树，所以这个B肯定是这个树的左子树的根，然后我们再看中序遍历的B，它的左边是D，中序遍历的特点是先左然后跟然后右，所以D是B的左子树，然而D的前面已经什么都没有了，也就是说D是叶子分支了，然后我们看B的右边的E，我们再结合前序遍历来看的话E在D的后面所以E是B字数的右子树，然后我们整个树的左子树已经遍历完成了，然后我们继续看这颗树的右子树，我们再看先序遍历，首先先序遍历已经推出了ABDE，所以后面剩下C F根据先序遍历的规则，C应该是这颗树的右子树的根，然后F根据中序遍历看就是C的左子树



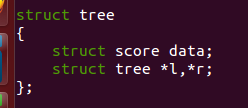
这颗树大致就应该是这个样子的，所以它的后序遍历就为DEBFCA

## 二叉树的程序实现

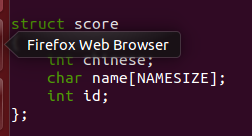
我们可以来写一个简单的树

将一个普通树转换成二叉树的规则：左指针指向它的第一个孩子右指针指向它的兄弟

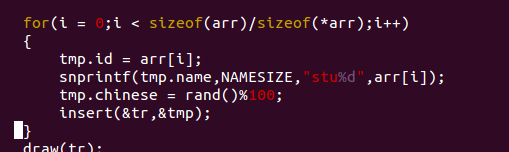
首先我们应该创建一个树的结构体，一个二叉树有两个分支所以我们需要两个入口



我们假设一个树的节点时用来存放一个学生的id 名字 以及他的成绩的

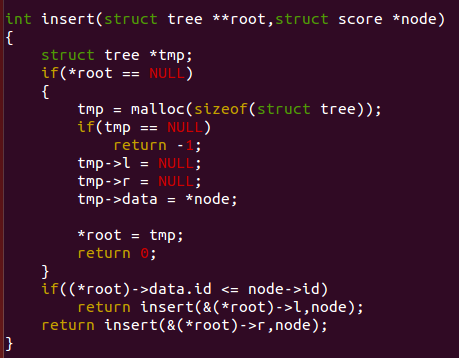


我们首先创建一个随机数组，我们在这里规定，如果数比上一个节点的数小就放在左分支，如果比上一个数大就放在右分支上

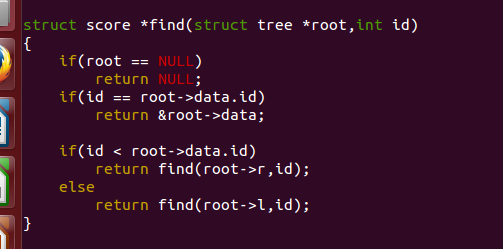


我们用一个for循环将数组中的内容全部放到树里面去，我们再看一下insert函数是怎么写的

这里面运用了一个递归的方法



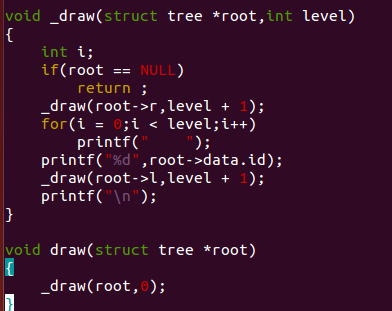
我们插入函数写出来了就不难写出查找函数



同样是运用了递归的原理

这样我们的简单的二叉树就已经写完了

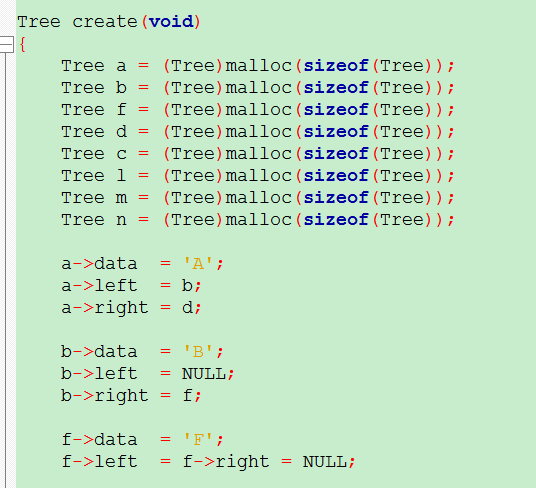
我们也可以画一个二叉树

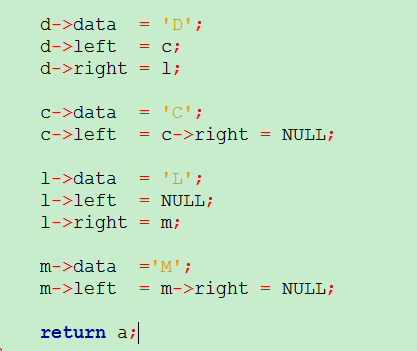


## 树的三种遍历的实现

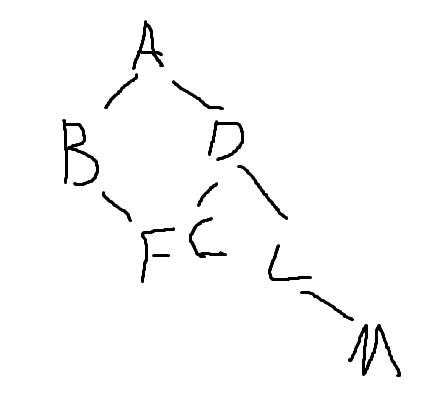
我们来写一个程序来简单实现一下树的三种遍历方式，树的三种遍历方式在之前说到概念的时候就已经可以看出来类似是递归，没错，我们就可以按照递归来写

我们简单先创建一个二叉树

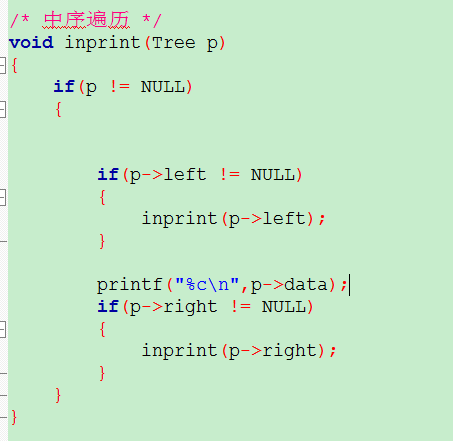




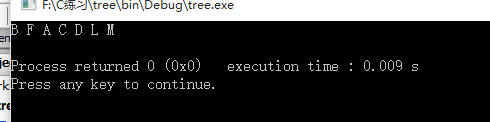
我们创建的二叉树是这样的



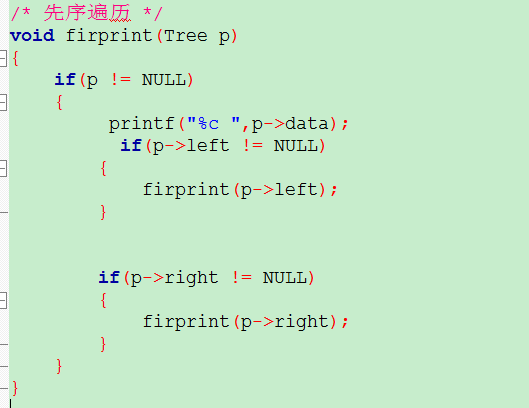
然后我们首先写一个中序遍历



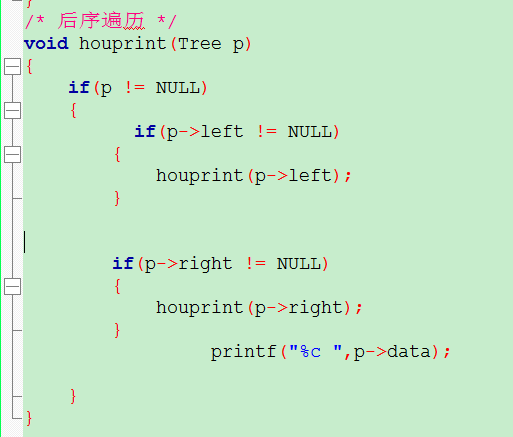
我们就是让其打印这个节点的树，然后按照中序遍历的节点的顺序来调用一下自己就好了

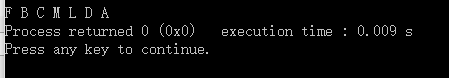


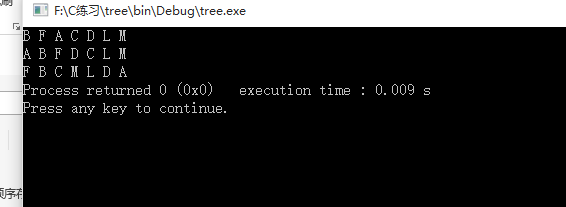
然后先序遍历也是一样











## 二叉查找树

二叉树的比较广泛的用途就是查找，我们假设现在要存放的是一些整数然后我们来写一些例程

首先是定义

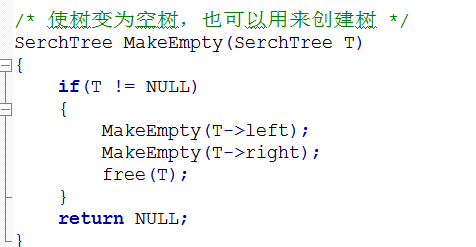
我们首先定义一下头文件



我们可以看到这些函数中的这些功能

我们来一一实现

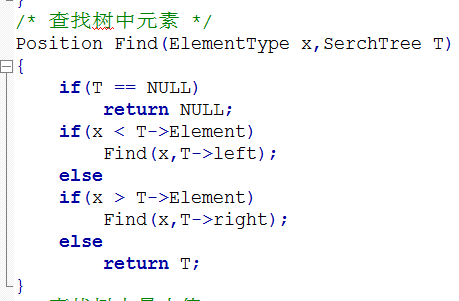
首先是MakeEmpty函数



就是将一棵树变为空树，也可以当成是初始化

接着是Find函数

我们这个树的规则是跟的左面放比根小的数右边放比跟大的数



也就是说小的数往左，大的数往右

运用尾递归

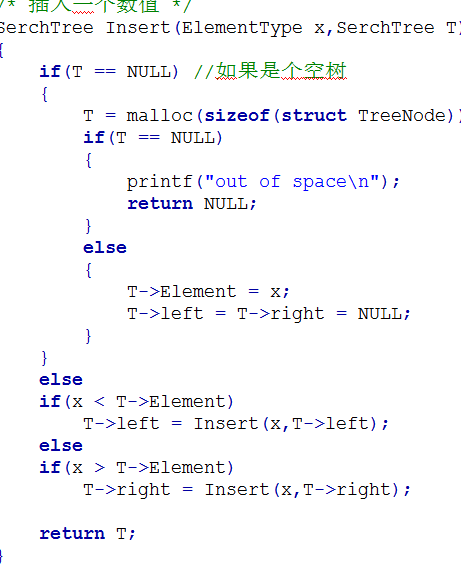
接下来是找最小值和最大值



我们再这里运用了两种方法来找最小值和最大值

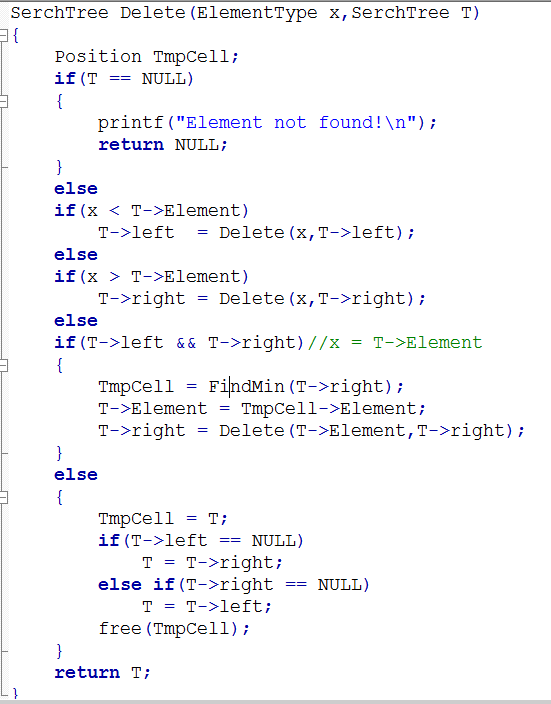
查找最小值运用的是递归函数，查找最大值用的是while

然后是插入函数，插入这个操作可以是像Find函数那样先查找如果找到跟插入一样的数据了，那么就不做操作，否则就插入到路径的最后一个点上



这里其实也是运用到了递归的用途

然后是删除，其实删除是最难的操作，我们采用这种策略：用其右子树最小的数据代替该节点的数据，然后删除原来的那个节点



还是大量用到了递归，所以总的看来树中递归的用处是非常大的

## AVL树

AVL树是带有平衡条件的二叉树，这个平衡条件必须要容易保持

最简单的概念就是左右子树具有相同的高度，另外一种平衡条件就是每个节点都必须要有相同高度的左子树和右子树

一课AVL树是其每个节点的左子树和右子树的高度最多差1的二叉查找树

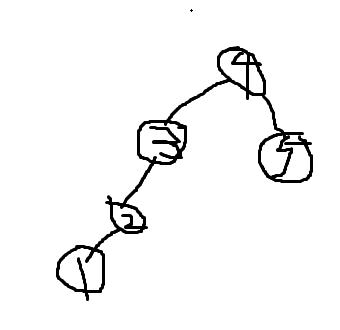
**在这里有个规律，就是一个AVL树的节点数，首先我们要知道AVL树的高度，节点数最少就是这个高度h的斐波那契数列的前h项之和**

破坏AVL树的平衡条件是有节点的左右子树的高度差2及以上

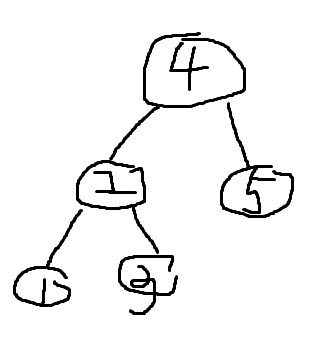
## 树的旋转问题

我们把已经不平衡的二叉树平衡起来的条件是旋转，我们通过旋转来将失去平衡的二叉树来恢复平衡

旋转分单旋转和双旋转



我们首先看一下这个树，这个数的左子树的左子树和右子树相差等于2，那么我们就需要进行一次单旋转，我们来看一下这个单旋转是怎么旋转的，我们将2替代3的位置，并将3作为2的右子树，这样就完成了一次旋转

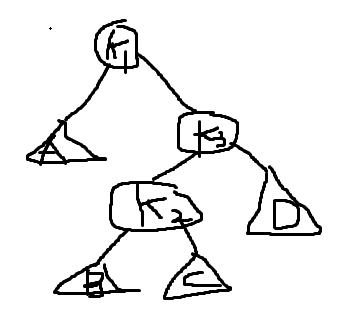


这个就是旋转之后的结果

我们再来看一下双旋转

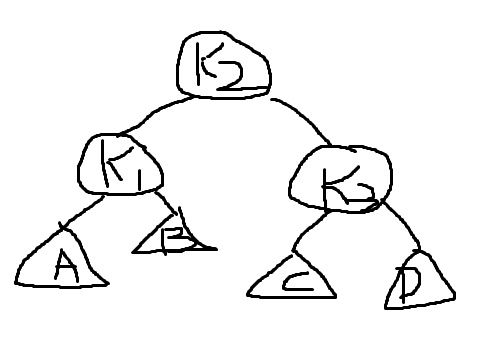
双旋转一般是在单旋转解决不了问题的时候使用的

我们看一下如何双旋转，实际上就是进行了两个单旋转



我们看到这颗树，这棵树是我们一个单旋转解决不了的，所以我们需要进行一个双旋转

我们将K2代替K1的位置，并将k1和k3作为k2的左右子树，k2的左子树变成了k1的右子树，k2的右子树变成k3的左子树



这样就又平衡了

## 伸展树

伸展树主要的功能就是类似于我们搜索时我们上次搜索出来的东西会第一个出现，伸展树就是这么一个思想，就是把这次需要找的东西放到树的根的部位，这样的算法也是类似于一种旋转的方法来实现的

## B-树

# 散列（哈希表）

散列是一种把数目多余数组的的数据简化并放到一个有限的空间中，当我们寻找的时候也会很好寻找

散列函数的三种寻常处理方式

·直接取余法：f(x):= x mod maxM ; maxM一般是不太接近 2^t 的一个质数。

·乘法取整法：f(x):=trunc((x/maxX)\*maxlongit) mod maxM，主要用于实数。

·平方取中法：f(x):=(x\*x div 1000 ) mod 1000000); 平方后取中间的，每位包含信息比较多。

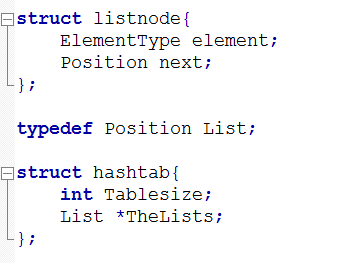
我们现在就拿最简单的直接取余法为例，它取余的一般是这个散列的大小，需要我们注意的是，如果大小是10，而且输入的数都是以0为个位的数值的话，那么就会发生分布不均匀的情况，为了避免这种情况，我们的散列的大小尽量要取到素数，也就是说如果我们需要的大小是10，那么我们就取11为这个散列的大小，而且尽量不要取2的n次方，因为我们拿2^3 = 8为例，分别取28和20，结果都是4，这时候就会不均匀，我们会看到如果用8的话那么这个数的二进制第四位不管是1还是0都是这个结果，所以我们应该取素数

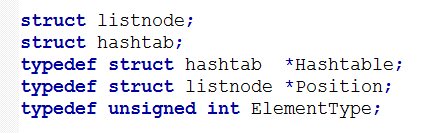
当我们遇到两个输入的位置都一样的时候那么我们成这个位“冲突”，我们处理冲突的办法有很多，这里举两个例子

## 分离连接法

这个办法是把这个散列每一个成员都变成一个链表的表头，然后如果有冲突的成员就放到同一个链表里面

我们来看一下代码





这里一个是散列表，一个是表中每个成员的链表

我们需要这么多个typedef，因为不这样下面有可能分不清谁是谁了

接下来是创建一个散列

Hashtable hash\_init(int size)

{

Hashtable h;

int i;

if(size < Minsize)

{

printf("The size is small!\n");

return NULL;

}

h = malloc(sizeof(struct hashtab));

if(h == NULL)

{

printf("out of space!\n");

return h;

}

h->Tablesize = NextPrime(size);//检验是否为素数，如果不是就变成最近的下一个素数

printf("%d\n",h->Tablesize);

h->TheLists = malloc(sizeof(List) \* h->Tablesize);

if(h->TheLists == NULL)

{

printf("Out of space!\n");

return NULL;

}

for(i = 0;i < h->Tablesize;i++)

{

h->TheLists[i] = malloc(sizeof(struct listnode));

if(h->TheLists[i] == NULL)

{

printf("Out of space\n");

return NULL;

}

else

{

h->TheLists[i]->next = NULL;

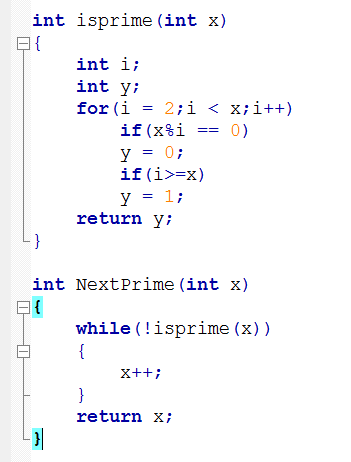
}

}

return h;

}

首先是给hashtable这个结构体分配空间，然后要把空间变成一个素数，这个函数如下

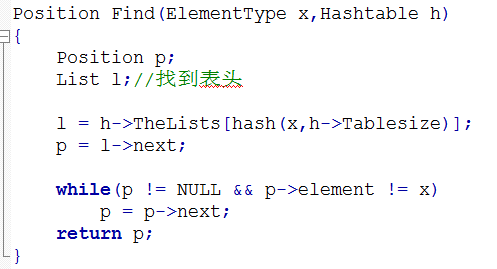


就是先判断是否是素数，如果不是就一直加到是为止

然后我们继续看创建函数，接下来我们根据刚才得到的大小分配散列中每个成员（也就是表头）的大小

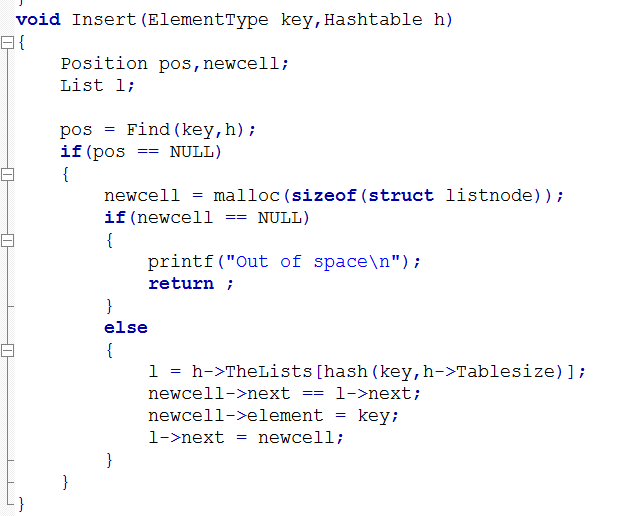
再之后就是在每个表头中分配一个链表的大小，最后返回这个散列的地址

然后是查找函数



这个跟链表的查找是一样的

然后是插入



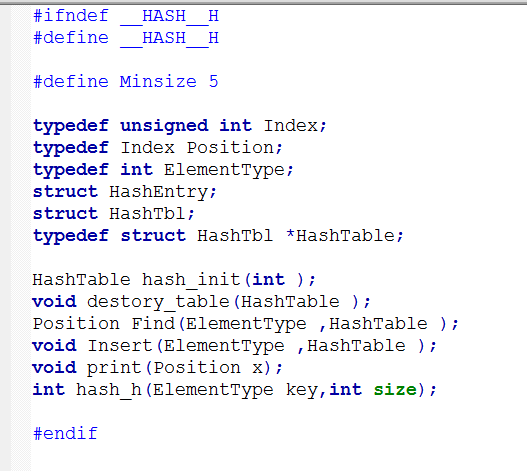
## 开放定址法

上面的方法需要用到链表，开放定址法的主要思想就是首先按照散列的算法先算出来，如果这个位置是空的就往里面放，如果遇到冲突就寻找下一个位置，直到有空位为止

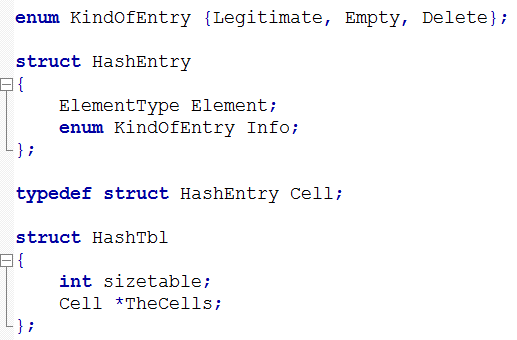
只要表足够大，总能找到一个单元存放数据

我们采用平方探测法来对这个冲突进行解决，也就是F（i）= i^2 i代表第几次，比如说我们现在第一次遇到冲突，那么I = 1，F（i）=1，那么我们就向下移动一个，i加一，之后如果还有元素那么我们就i的平方，这次I等于2，那么就向下移动两个单位，以此类推

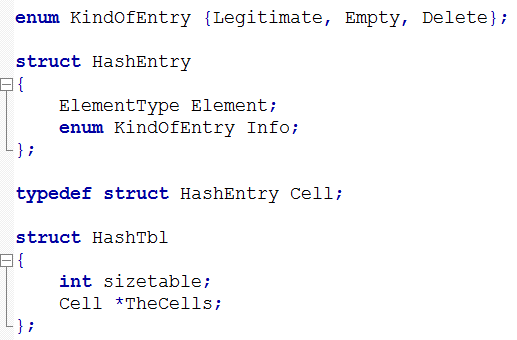
我们看一下程序的实现

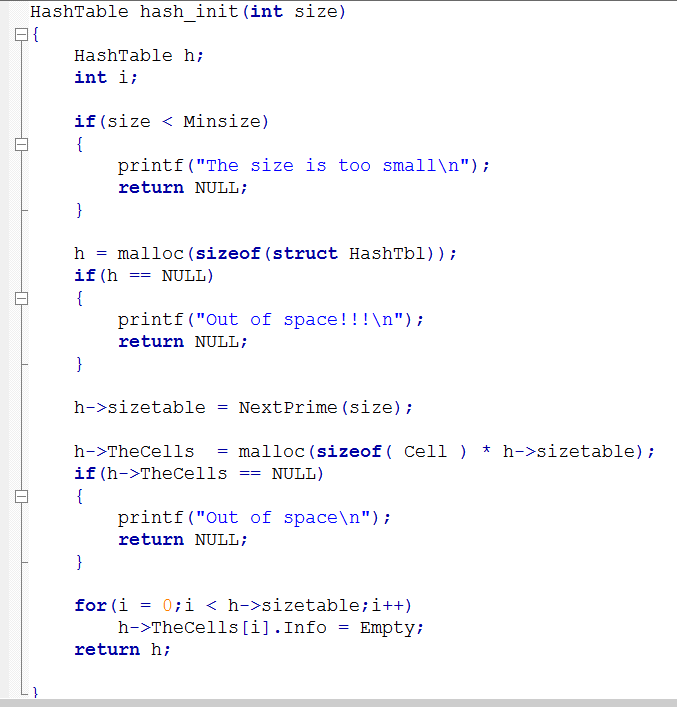


首先这是头文件

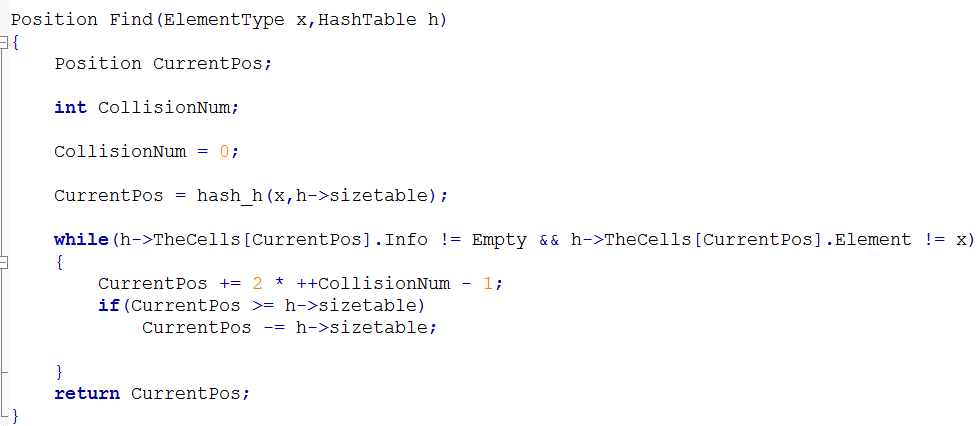


以及我们需要用到的结构体





初始化程序跟上面的差不多，只不过需要在每个空位中加上Empty标记



然后是查找函数

这个函数就是根据平方探测法写出来的，其中最主要的算法是CurrentPos += 2 \* ++CollisionNum - 1;这个的意思是把平方展开了

F（i） = F（I - 1） + 2\*I – 1

最后是插入函数也跟上面的一样

