#include<iostream>

using namespace std;

void max\_min(int\* a,int i,int j,int &max,int &min)

{

if(i==j){

max=a[i];

min=a[i];

}

else if(i==j-1){

max=(a[i]>a[j]?a[i]:a[j]);

min=(a[i]<a[j]?a[i]:a[j]);

}

else{

int mid =i+(j-i)/2;

int max1,min1;

max\_min(a,i,mid,max,min);

max\_min(a,mid+1,j,max1,min1);

if(max<max1) max=max1;

if(min>min1)min=min1;

}

}

int main()

{

int a[20]={1,2,3,4,5,6,7,8,34,23,56,78,32,12,13,15,36,45,65,89};

int max,min;

max\_min(a,0,19,max,min);

cout<<"max is:"<<max<<" min is:"<<min<<endl;

}

int **Partition** (Type a[], int m, int r)

{

int i = m, j = r+1; //考虑这里为什么+1？

Type x=a[m];

// 将< x的元素交换到左边区域

// 将> x的元素交换到右边区域

while (true)

{

while (a[++i] <x);

while (a[- -j] >x);

if (i >= j) break; //退出循环

Swap(a[i], a[j]);//交换两个元素

}

a[m] = a[j];

a[j] = x;

return j;

}

void **QuickSort** (Type a[], int p, int r)

{

if (p<r) {

int q=Partition(a,p,r);

QuickSort (a,p,q-1); //对左半段排序

QuickSort (a,q+1,r); //对右半段排序

}

}

删除字符串s中出现的所有字符c。

/\* squeeze: delete all c from s \*/

void squeeze(char s[], int c)

{

int i, j;

for (i = j = 0; s[i] != '\0'; i++)

if (s[i] != c)

s[j++] = s[i];

s[j] = '\0';

}

删除连续的重复字符：

#include<iostream>

using namespace std;

void remove\_elem(char\* s)

{

int i=0,j=0;

while(s[i]!='\0'){

if(s[i]==s[j])

s[j]=s[i++];

else{

++j;

s[j]=s[i++];

}

}

s[++j]='\0';

}

int main()

{

char a[17]="kooooooosddddedd";

remove\_elem(a);

int i=0;

while(a[i]!='\0')

cout<<a[i++]<<' ';

cout<<endl;

}

速度快的memcpy：

void \*my\_memcpy(void \*dst, void\* src, int size)  
{  
    int len = sizeof(long);  
    long \*pdst = (long\*)dst;  
    long \*psrc = (long\*)src;  
    for (; size >= len; ++pdst, ++psrc, size -= len)  
    {     
        \*pdst = \*psrc;  
    }     
    for (len = size, size = 0; size < len; ++size)  
    {     
        \*((char\*)pdst + size) = \*((char\*)psrc + size);  
    }     
    return dst;  
}  
int main()  
{  
    char src[] = "hello", dst[10] = {'\0'};  
    my\_memcpy(dst, src, 5);   
    printf("dst = %s\n", dst);  
    return 0;  
}

将整型数组的元素左移m位，不能申请数组空间，不允许采用每次左移：

#include<iostream>

using namespace std;

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

{

int x;

x=\*a;

\*a=\*b;

\*b=x;

}

void move\_array(int data[],int n,int m)

{

int i,j;

m=m%n;

if(m==0) return;

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

{

for(j=i;j<i+m&&j<n-m;j++)

{

swap(&data[j],&data[j+m]);

}

}

move\_array(data+n-m,m,m-n%m);

}

[#面试题#](http://huati.weibo.com/k/é¢è¯é¢?from=501)给定一个无序的整数数组，怎么找到第一个大于0，并且不在此数组的最小整数。比如[1,2,0] 返回 3, [3,4,-1,1] 返回 2。最好能O(1)空间和O(n)时间。

来源[http://weibo.com/lirenchen](http://weibo.com/lirenchen?from=otherprofile&wvr=4&loc=infweihao)。

解法：

直接考虑这个问题是比较困难的。不如先换个简单的问题。

**问题1**

给定一个数组，长度为n，除a[0]以外，其他元素都是a[i] == i。那么请找出第一个大于0，且不在此数组中的最小整数。

答案就是:  n + (a[0] == n);

**问题2**

给定一个数组，长度为n，某几个位置的元素满足 a[x] < 0 或者 a[x] > n，余下的元素都满足a[i] == i。那么请找出第一个大于0，且不在此数组中的最小整数。

答：

很明显：当x的值限定为0时。就变成了问题1。

除此之外，这个问题的解也容易求得。

情况1： 首先从1~n开始扫描，当发现i != a[i]时，直接返回i。此时的i必定是最小的未出现的整数。

情况2： 当扫描完1~n之后，那么就回归至问题1了。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ju136/article/details/8153274)

1. for (i = 1; i < n; ++i) {
2. if (i != a[i]) return i;
3. }
4. return (n + (n == a[0]));

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

if (i != a[i]) return i;

}

return (n + (n == a[0]));

**问题3**

当给出问题2之后，就需要考虑如何把原题目转换成为问题2了。

实际上要完成的任务就变成了，如何把一个数组中的元素**元神归位**。也就是让a[i] == i。如果能让这些元素元神归位，那么就转换成为问题2了。就很容易求解了。

算法如下：

       我们从后往前扫描，i = n - 1 to 0;

                  step1如果发现a[i] < 0 || a[i] > n; 则 continue;

                  step2 如果发现a[i] == a[a[i]]; 则continue;

                  step3 如果发现0 < a[i] < n

                          说明需要将a[i]元神归位。也就是放到a[a[i]]上去。

                           swap(a[i], a[a[i]]).

                           再跳转至step2。

Note: 需要注意一种有重复数的情况，比如a[11] = 2, a[2] = 2。这时候，就不用进行交换了。直接处理下一个元素。

可以给出代码：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ju136/article/details/8153274)

1. **int** i = n, t, temp;
2. if (!a || n <= 0) return -1;
4. while ((--i) >= 0) {
5. while (0 < a[i] && a[i] < n && i != a[i]) {
6. t = a[i];
7. if (a[i] == a[t]) break;
8. temp = a[i];
9. a[i] = a[t];
10. a[t] = temp;
11. }
12. }

int i = n, t, temp;

if (!a || n <= 0) return -1;

while ((--i) >= 0) {

while (0 < a[i] && a[i] < n && i != a[i]) {

t = a[i];

if (a[i] == a[t]) break;

temp = a[i];

a[i] = a[t];

a[t] = temp;

}

}

好吧，到现在为止，原题已经变得很简单了。可以直接给出代码了。

**原题解答**

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ju136/article/details/8153274)

1. **int** find(**int** \*a, **int** n) {
2. **int** i = n, t, temp;
3. if (!a || n <= 0) return -1;
5. while ((--i) >= 0) {
6. while (0 < a[i] && a[i] < n && i != a[i]) {
7. t = a[i];
8. if (a[i] == a[t]) break;
9. temp = a[i];
10. a[i] = a[t];
11. a[t] = temp;
12. }
13. }
15. for (i = 1; i < n; ++i)
16. if (a[i] != i) return i;
18. return (n + (a[0] == n));
19. }
20. 散列基本思想：1确定一个函数关系。2以节点的关键码k为自变量。3函数值h(k)作为节点的存储地址。4检索时也是根据这个函数计算其存储位置。
21. 通常散列表的存储空间是一个一维数组，散列地址是数组的下标。
22. 冲突：某个散列函数对于不相等的关键码计算出了相同的散列地址。
23. 负载因子：空间m,节点数n,q=n/m。
24. 同义词：发生冲突的两个关键码。
25. 散列函数的选取方法：除余法，平方取中法，折叠法。
26. 除余法：用关键码x 除以M（往往取散列表长度，一般选择小于表长的最大质数），并取余数作为散列地址。
27. 平方取中法：先通过求关键码的平方来扩大差别，再取其中的几位来作为散列地址。
28. 变换基数：将十进制数看成13进制，然后转成10进制，然后取后几位存储。
29. 折叠法：图书号是0-42-20586-4 用5864+4220+04 取最后几位得到0088作存储地址。或者分界叠加：5864+0224+04.
30. ELFhash字符串散列函数。可用于搜索引擎URL的散列。
31. int ELFhash(char\* key)
32. {
33. unsigned long h=0;
34. while(\*key){
35. h=(h<<4)+\*key++;
36. unsigned long g=h&0xF000000L;
37. if(g) h^=g>>24;
38. h&=~g;
39. }
40. return h%M;
41. }
42. 碰撞的处理：
43. 开散列方法，也称拉链法，把发生冲突的关键码存储在散列主表之外。
44. 闭散列法：也成开地址法。
45. 散列的搜索时间一般是<2的。
46. 如果整个散列表存储在内存中，开散列方法比较容易实现。
47. 如果散列表存储在磁盘中，用开散列不合适，因为同义词元素可能会存储在不同的磁盘页中，会引起多次的磁盘访问，从而增加检索时间，可用桶式散列方法（把一个文件的记录分为若干个存储桶，每个存储桶包含一个或多个页块，一个存储桶内的各页块用指针连接起来，每个页块包含若干记录。散列函数h(k)表示具有关键码值k的记录所在的存储桶号）。
48. 闭散列方法：当冲突发生时，使用某种方法为关键码值K生成一个散列地址序列d0,d1,d2,…dm-1。其中d0=h(k)称为K的基地址，所有di(0<i<m)是后继散列地址。形成探查的方法不同，所得到的解决冲突的方法也不同。
49. 线性探查：依次探查下述地址单元：d+1,d+2,d+3…0,1,…d-1。
50. 改进线性探测法：每次跳过c个槽。
51. 二次探测：探测序列依次为1^2,-1^2,2^2,-2^2,
52. 伪随机数探测：产生一个伪随机序列，每次都是用同样的序列。
53. void permute(int\* array,int n)
54. {
55. for(int i=1;i<=n;i++)
56. swap(array[i-1],array[Random(i)]);
57. }
58. 双散列探查法：使用两个散列函数h1 和h2。若在地址h1(key)=d发生冲突，则再计算h2(key)得到的探查序列为：(d+h2(key))%M, (d+2h2(key))%M,(d+3h2(key))%M。
59. 散列表中删除元素：开散列可以真正删除，闭散列方法要做标记，不能真正删除，删除后会使探查序列断掉。
60. 索引是把一个关键码与它对应的数据记录的位置相关联的过程。索引文件用于记录这种联系的文件组织结构。索引文件的记录：关键码和指针对。指针指向主要数据库文件中的完整记录。二级索引是一级索引的索引。
61. B树是一种平衡的多分树，有5个特点：
62. 1每个节点至多有m个子节点。
63. 2除根节点和叶节点外，其他每个节点至少有m/2取上整个子节点。
64. 3根节点至少有两个子节点。唯一例外的是根节点就是叶节点时，没有子节点，此时B树只包含一个节点。
65. 4所有的叶节点在同一层。
66. 5有K个子节点的非根节点恰好包含k-1个关键码。
67. B树的性质：1树高平衡，所有节点都在同一层。2关键码没有重复，父节点中的关键码是其子节点的分界。3B树把（值接近）相关记录放在同一个磁盘页中，从而利用了访问局部性原理。4B树保证树中至少有一定比例的节点是满的，这样能够改进空间的利用率，能减少检索和更新操作的磁盘读取数目。
68. B树的查找：在两个关键码之间有一个指针，指向关键码值介于这两个关键码之间的关键码值。每一个关键码都包含一个指针，这个指针指向关键码在主文件中的位置。
69. B树的插入：注意保持性质，特别是等高和阶的限制，1找到最底层，插入。2若溢出，则节点分裂，中间关键码连同新指针插入父节点。3若父节点也溢出，则继续分裂。4分裂过程可能传达到根节点（则树升高一层）。
70. B树的删除:删除的关键码不在叶节点层，先把此关键码与它在B树里的后继兑换位置(找左边子树的最大值或右边子树的最小值交换)，然后再删除该关键码。
71. 如果删除的关键码在叶节点层：
72. 删除后关键码个数不小于m/2-1，则直接删除。
73. 如果关键码值小于m/2-1，则：
74. 如果兄弟节点关键码值个数不等于m/2-1,则从兄弟节点中移若干个关键码到该节点中来，父节点中的一个关键码要做相应的变化。
75. 如果兄弟节点关键码个数等于m/2-1，则合并。同时将父节点中的关键码拉下来。
76. B+树： 1每个节点至多有m个子节点，2每个节点（除根外）至少有m/2个子节点。3根节点至少有两个子节点。4有k个子节点的节点必有k个关键码。在叶节点层是所有关键码的全集。节点中的关键码是子节点中最大(最小)关键码的复写。
77. B+树的查找：应该到叶节点层，在上层已找到待查的关键码，并不停止，而是继续沿指针向下一直查到叶节点层的这个关键码。B+树的叶节点一般链接起来，形成一个双链表。B+树的应用更广泛。B+树的插入：插入到底层，如果溢出，就分裂，然后将分裂后节点的最大元素写入父节点，如果父节点也溢出，则继续分裂。B+树的删除：从父节点找到大于要删除节点的最小关键码，然后一直查找到叶节点。然后删除，如果要删除的节点在父节点中的有副本，则不必删除副本。动态索引使用B树和B+树。
78. B树的性能分析：包含N个关键码的B树有N+1个外部空指针。
79. 中科院出的切词包，能将一段话分成词语，ICTCLAS。将源代码中的六个文件夹中的内容添加到自己的工程中，运行后将错误提示的头文件包含进来。直到没有错误为止。
80. Trie结构主要应用于信息检索，用来存储英文字符串，尤其是大规模的英文字典。
81. Trie结构是非平衡的，PATRICIA结构是平衡的，每个内部节点都代表一个位的比较，必然产生两个子节点，所以它是满二叉树。进行一次检索，最多只需要关键码位数次的比较即可。
82. AVL树：如果T是一棵AVL树，则它的左右子树也是AVL树，并且|hl-hr|<=1。左右子树的高度之差是平衡因子，平衡因子是0,1，-1。

int find(int \*a, int n) {

int i = n, t, temp;

if (!a || n <= 0) return -1;

while ((--i) >= 0) {

while (0 < a[i] && a[i] < n && i != a[i]) {

t = a[i];

if (a[i] == a[t]) break;

temp = a[i];

a[i] = a[t];

a[t] = temp;

}

}

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

if (a[i] != i) return i;

return (n + (a[0] == n));

}

折半查找的迭代版

int bsearch(int \*a,int x,int y,int t){

int m;

while(x<y){

m = x+(y-x)/2;

if(a[m] == t)

return m;

else if(a[m] > t)

y=m;

else

x=m+1;

}

return -1;

}

链表检测是否有环

bool hasCycle(ListNode \*head) {

if(head == NULL)

return false;

ListNode \*pre = head;

ListNode \*cur = head;

while(cur != NULL){

pre = pre->next;

cur = cur->next;

if(cur != NULL)

cur = cur->next;

else

return false;

if(cur == pre)

return true;

}

return false;

}

检测链表环的起始位置

ListNode \*detectCycle(ListNode \*head) {

if(head == NULL)

return NULL;

ListNode \*pre = head;

ListNode \*cur = head;

while(cur != NULL){

pre = pre->next;

cur = cur->next;

if(cur != NULL)

cur = cur->next;

else

return NULL;

if(cur == pre){

cur = head;

while(pre != cur){

pre = pre->next;

cur = cur->next;

}

return cur;

}

}

return NULL;

}

合并两个有序的链表

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode \*p;

ListNode \*s;

ListNode \*pre;

ListNode \*beh;

ListNode \*result;

if(l1 == NULL)

return l2;

else if(l2 == NULL)

return l1;

else if(l1->val <= l2->val){

pre = l1;

p=l1;

s=l2;

result = l1;

beh = s->next;

}else if(l1->val > l2->val){

pre = l2;

p=l2;

s=l1;

result = l2;

beh = s->next;

}

p = p->next;

while(p !=NULL && s !=NULL){

if(p->val < s->val){

p = p->next;

pre = pre->next;

}else{

pre->next = s;

s->next = p;

pre = pre->next;

if(beh != NULL){

s = beh;

beh = beh->next;

}else

break;

}

}

if(p == NULL){

pre->next = s;

}

return result;

}

链表反转 pre是反转起始节点的前一个节点。

void reverse(ListNode \*pre,ListNode \*begin,ListNode \*end){

ListNode \*end\_next = end->next;

ListNode \*cur = begin;

ListNode \*beh = begin->next;

while(beh != end\_next ){

ListNode \*next;

if(beh->next != NULL)

next = beh->next;

else

next = NULL;

beh->next = cur;

cur = beh;

beh = next;

}

pre->next = end;

begin->next = end\_next;

}

单链表归并排序

ListNode \*sortList(ListNode \*head) {

if (head == NULL || head->next == NULL)return head;

ListNode \*fast = head, \*slow = head;

while (fast->next != NULL && fast->next->next != NULL) {

fast = fast->next->next;

slow = slow->next;

}

fast = slow;

slow = slow->next;

fast->next = NULL;

ListNode \*l1 = sortList(head);

ListNode \*l2 = sortList(slow);

return mergeTwoLists(l1, l2);

}

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode dummy(-1);

for (ListNode\* p = &dummy; l1 != nullptr || l2 != nullptr; p = p->next) {

int val1 = l1 == nullptr ? INT\_MAX : l1->val;

int val2 = l2 == nullptr ? INT\_MAX : l2->val;

if (val1 <= val2) {

p->next = l1;

l1 = l1->next;

} else {

p->next = l2;

l2 = l2->next;

}

}

return dummy.next;

}

http://www.zhihu.com/question/28431865