**12、排序算法有哪些？快速排序是怎么实现的？最好时间复杂度，平均时间复杂度？**

快速排序

void qsort(int a[],int low,int high)

{

if(low>high)

return;

int key=a[low]; //数组中第一个作为枢轴

int front=low;

int last=high;

while(front<last)

{

while(front<last&&a[last]>=key)

{

last--;

}

a[front]=a[last]; //将比第一个小的移到低端

while(front<last&&a[front]<=key)

{

front++;

}

a[last]=a[front]; //将比第一个大的移到高端

}

a[front]=key; //枢轴记录到位

qsort(a,low,front-1);

qsort(a,front+1,high);

}

平均时间复杂度O(nlogn)，最坏时间复杂度O(n2),空间复杂度平均O(logn)最坏O(n)

归并排序

void Merge(int a[],int start,int mid,int end,int temp[])

{

int i=start,j=mid+1,k=start;

// std::cout<<start<<" "<<mid<<" "<<end<<std::endl;

while(i!=mid+1&&j!=end+1)

{

if(a[i]>a[j])

{

temp[k++]=a[j++];

}

else

{

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

}

}

while(i!=mid+1)

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

while(j!=end+1)

temp[k++]=a[j++];

for(i=start;i<=end;i++)

{

a[i]=temp[i];

}

}

void MergeSort(int a[],int start,int end,int temp[])

{

int mid;

if(start<end)

{

mid=(start+end)/2;

MergeSort(a,start,mid,temp);

MergeSort(a,mid+1,end,temp);

Merge(a,start,mid,end,temp);

}

}

时间复杂度O(nlogn)，空间复杂度O(n)

**14、计算二进制的一的个数，这个算法叫做平行算法。**

int BitCount(unsigned int n)

{

    n = (n &0x55555555) + ((n >>1) &0x55555555) ;

    n = (n &0x33333333) + ((n >>2) &0x33333333) ;

    n = (n &0x0f0f0f0f) + ((n >>4) &0x0f0f0f0f) ;

    n = (n &0x00ff00ff) + ((n >>8) &0x00ff00ff) ;

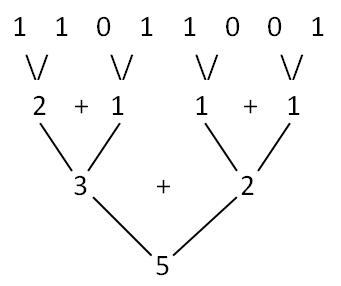
    n = (n &0x0000ffff) + ((n >>16) &0x0000ffff) ;

    return n ;

}

速度不一定最快，但是想法绝对巧妙。 说一下其中奥妙，其实很简单，先将n写成二进制形式，然后相邻位相加，重复这个过程，直到只剩下一位。

以217（11011001）为例，有图有真相，下面的图足以说明一切了。217的二进制表示中有5个1



**19、单链表反转**

/\*

链表

1.带头结点的：head里面存放链表长度（或其他信息），head->next指向第一个实际节点；

2.不带头结点的：head即第一个实际节点

\*/

typedef struct Node

{

int data;

Node \* next;

};

void reverseList(Node \*head)

{

if ((head==NULL)||((head->next)==NULL)) return ; //如果head为空，或者头结点指向空节点（链表长度为0），则退出。

Node \*pre,\*cur,\*next; //cur 记录当前位置，pre记录上一个位置，为cur->next的值；next记录下一个位置，反转后cur不等于cur->next

cur=head->next;

pre=NULL; //逆转之后，头结点变为尾节点，其next为Null

while (cur!=NULL)

{

next=cur->next; //反转后不能再用cur->next，所以先记录下这个节点

cur->next=pre;

pre=cur;

cur=next;

}

head->next=pre; //cur已经为空，所以pre为尾节点。head->next指向它。

return ;

}

**20、单链表查找倒数第k个节点**

题目：输入一个单向链表，输出该链表中倒数第k个结点。链表的倒数第0个结点为链表的尾指针。

分析：为了得到倒数第k个结点，很自然的想法是先走到链表的尾端，再从尾端回溯k步。可是输入的是单向链表，只有从前往后的指针而没有从后往前的指针。因此我们需要打开我们的思路。既然不能从尾结点开始遍历这个链表，我们还是把思路回到头结点上来。假设整个链表有n个结点，那么倒数第k个结点是从头结点开始的第n-k-1个结点（从0开始计数）。如果我们能够得到链表中结点的个数n，那我们只要从头结点开始往后走n-k-1步就可以了。如何得到结点数n？这个不难，只需要从头开始遍历链表，每经过一个结点，计数器加一就行了。这种思路的时间复杂度是O(n)，但需要遍历链表两次。第一次得到链表中结点个数n，第二次得到从头结点开始的第n­-k-1个结点即倒数第k个结点。如 果链表的结点数不多，这是一种很好的方法。但如果输入的链表的结点个数很多，有可能不能一次性把整个链表都从硬盘读入物理内存，那么遍历两遍意味着一个结 点需要两次从硬盘读入到物理内存。我们知道把数据从硬盘读入到内存是非常耗时间的操作。我们能不能把链表遍历的次数减少到1？如果可以，将能有效地提高代码执行的时间效率。如果我们在遍历时维持两个指针，第一个指针从链表的头指针开始遍历，在第k-1步之前，第二个指针保持不动；在第k-1步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在k-1，当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后面的）指针正好是倒数第k个结点。这种思路只需要遍历链表一次。对于很长的链表，只需要把每个结点从硬盘导入到内存一次。因此这一方法的时间效率前面的方法要高。

**24、判断单链表中有没有环**

方法：快慢指针

核心思想是设定两个指针p,q，其中p每次向前移动一步，q每次向前移动两步。如果单链表中存在环则p和q最终会相遇，否则q将会首先遇到null。（可以参考在操场上跑步绕圈，快的会把慢的套圈）

求环的入口点？

记录p和q第二次相遇的位置就是环的入口点

求环长？

第二问中已经知道了环的入口点，利用计数器再跑一圈就可以了

**42、知道数组名，求数组的长度**

不考虑函数中值传递数组传递为指针的情况

可以写一个模板来进行通用化

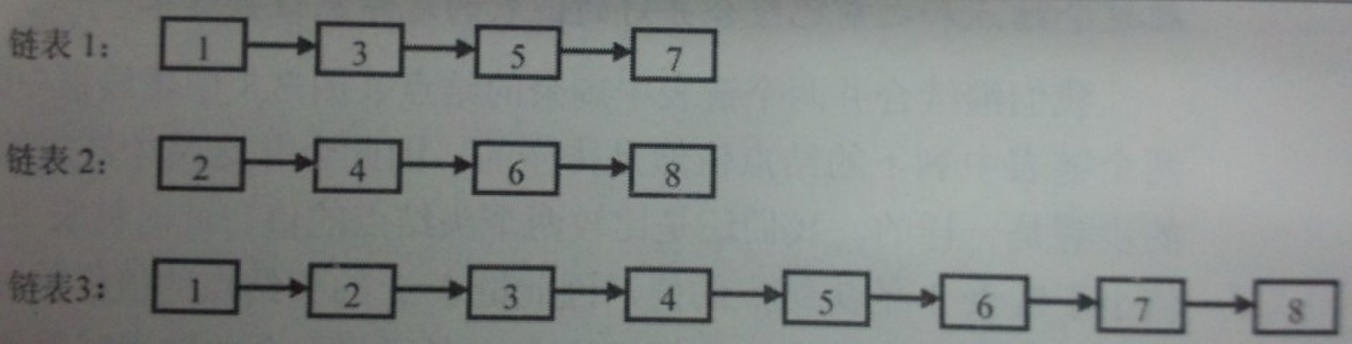
Template<class T>

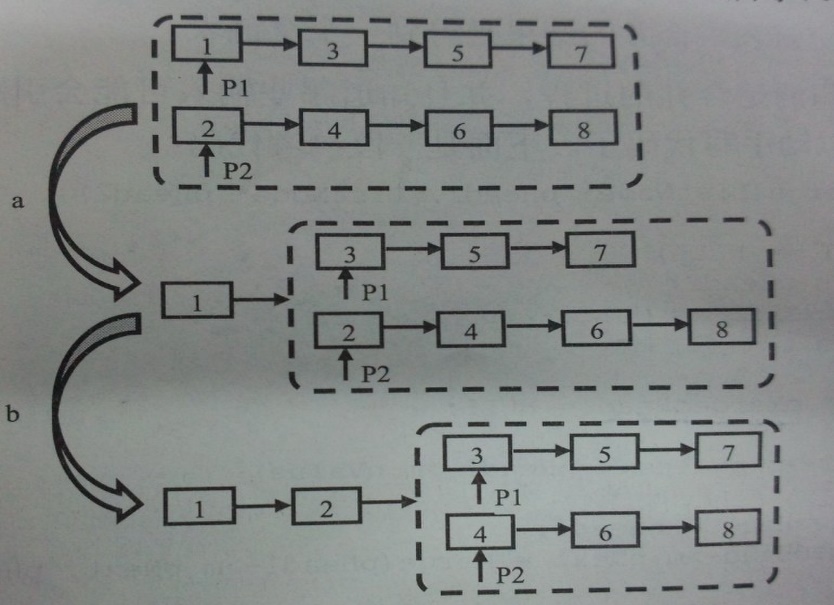
Int getArrayLen(T& array)

{

return (sizeof(array)/sizeof(array[0]));

**48、合并有序链表**





首先给出一个递归实现，他理解起来非常简单

Struct node

{

int data;

node\* next;

}

node\* Merge(node\* pHead1,node\* pHead2)

{

if(pHead1==NULL)

reutrn pHead2;

else if(pHead2==NULL)

return pHead1;

node\* pMergeHead=NULL;

if(pHead1->data<pHead2->data)

{

pMergeHead=pHead1;

pMergeHead->next=Merge(pHead1->next,pHead2);

}

else

{

pMergeHead=pHead2;

pMergeHead->next=Merge(pHead1,pHead2->next);

}

return pMergeHead;

}

此方法可以说极为的清晰易懂，缺点是，他需要链表长度的栈空间，当链表很长的时候，递归的效率将会比较低。

那么接下来给出一个非递归的版本

node\* Merge2(node \*lsit1,node \*list2)

{

if(list1==NULL)

return list2;

if(list2==NULL)

return list1;

node \*head=NULL;

node \*p=NULL;

if(list1->data<=list2->data)

{

head=list1;

list1=list1->next;

}

else

{

head=list2;

list2=list2->next;

}

p=head;

while(list1!=NULL&&list2!=NULL)

{

if(list1->data<=list2->data)

{

p->next=list1;

list1=list1->next;

p=p->next;

}

else

{

p->next=list2;

list2=list2->next;

p=p->next;

}

}

if(list1!=NULL)

{

p->next=list1;

}

else

{

p->next=list2;

}

return head;

}

**49、二叉搜索树转双向链表**

二叉搜索树的中序遍历是一个顺序结构，转换的链表也是一个顺序结构

10 
12 
16 

可以看到在遍历到树（或者子树）的根节点之后，将它左子树的最大节点和右子树的最小节点连在一起，然后处理每个叶子节点的连接关系，即可完成转换

void converNode(BinaryTreeNode\*,BinaryTreeNode\*\*);

BinaryTreeNode\* convert(BinaryTreeNode\* pRoot)

{

BinaryTreeNode \*pLastNode=NULL;

convertNode(pRoot,&pLastNode);

//pLast指向双向链表的尾结点

//现在需要返回头节点

BinaryTreeNode \*pHeadOfList=pLastNode;

while(pRoot!=NULL&&pHeadOfList->left!=NULL)

{

pHeadOfList=pHeadOfList->left;

}

return pHeadOfList;

}

void convertNode(BinaryTreeNode \*pNode,BinaryTreeNode \*\*pLastNode)

{

if(pNode==NULL)

return;

BianryTreeNode \*pCurrent=pNode;

if(pCurrent->left!=NULL)

convertNode(pCurrent->left,pLastNode);

pCurrent->left=\*pLastNode;

if(\*pLastNode!=NULL)

(\*pLastNode)->right=pCurrent;

\*pLastNode=pCurrent;

if(pCurrent->right!=NULL)

converNode(pCurrent->right,pLastNode);

}

**50、字符串排列**

输入一个字符串，输出该字符串的所有排列

不用C++ STL next\_permutation的方法

首先求所有可能出现在第一个位置的字符，即把第一个字符和后面的所有字符交换，然后固定第一个字符求后面的字符，后面的字符继续固定第一个字符，与后面的做交换……

这就是一个递归

(a) 
(b) 

固定第一个字符交换的情况

void Permutation(char\*);

void Permutation(char\*,char\*);

void Permutation(char \*pStr)

{

if(pStr==NULL)

return;

Permutation(pStr,pStr);

}

void Permutation(char \*pStr,char \*pBegin)

{

if(\*pBegin=='\0')

{

std::cout<<pStr<<std::endl;

}

else

{

for(char\* pch=pBegin;\*pCh!='\0';++pch)

{

char temp=\*pCh;

\*pCh=\*pBegin;

\*pBegin=temp;

Permutation(pStr,pBegin+1);

temp=\*pCh;

\*pCh=\*pBegin;

\*pBegin=temp;

}

}

}

**51、数组中出现次数超过一半的数字**

数组中有一个数字出现的次数超过数组长度的一半，也就是说他出现的次数比其他所有数字出现次数的和还要多。因此，可以考虑在遍历数组的时候保存两个值，一个是数字组中的一个数字，一个是次数。当我们遍历到下一个数字的时候 ，如果下一个数字和之前保存的数字不同，则次数减1。如果次数为0，则需要保存下一个数字，并把次数设为1，最后保留下来的一定是最后一次把次数设为1时对应的数字

int MoreThanHalfNum\_by\_Through(int \*numbers,int length)

{

if(CheckInvalidArray(numbers,length)

return 0;

int result=numbers[0];

int times=1;

for(int i=1;i<length;++i)

{

if(times==0)

{

result=numbers[i];

times=1;

}

else if(numbers[i]==result)

times++;

else

times--;

}

if(!CheckMoreThanHalf(numbers,length,result))

result=0;

return result;

}

**52、数组的第K个数**

使用Partition函数，基于数组的第K个数字来调整，使得比第k个数字小的所有数字都位于数组的左边，比第k个数大的都位于数组的右边。

void GetLeastNumbers(int \*input,int n,int k)

{

if(input==NULL||output==NULL||k>n||n<=0||k<=0)

return;

int start=0;

int end=n-1;

int index=Partition(input,n,start,end);

while(index!=k-1)

{

if(index>k-1)

{

end=index-1;

index=Partition(intput,n,start,end);

}

else

{

start=index+1;

index=Partition(input,n,start,end);

}

}

return input[k-1];

}

Partition方法

int Partition(int data[],int length,int start,int end)

{

if(data==NULL||length<=0||start<0||end>=length)

std::cerr<<"error"<<"\n";

int index=start;

swap(&data[index],data[end]);

int start=-1;

for(int index=start;index<end;++index)

{

if(data[index]<data[end]

{

++small;

if(small!=index)

swap(&data[index],&data[small]);

}

}

++small;

swap(&data[small],&data[end]);

return small;

}

另外也可以使用最小堆（堆排序）的方法

以及更加简单粗暴的STL库中的map（红黑树实现）