https://cloud.tencent.com/developer/article/1826525

单链表的冒泡，快排，选择，插入，归并5种排序算法详解(多图+代码实现)

文章被收录于专栏：[嵌入式与Linux那些事](https://cloud.tencent.com/developer/column/91634" \t "_blank)

上节介绍了链表的基本操作史上最全单链表的增删改查反转等操作汇总以及5种排序算法(C语言)

[https://cloud.tencent.com/developer/article/1826524](https://cloud.tencent.com/developer/article/1826524?from_column=20421&from=20421) 这节介绍链表的5种排序算法。

**0.稳定排序和原地排序的定义**

**稳定排序**：

  假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的**相对次序保持不变**，即在原序列中，ri=rj，且ri在rj之前，而在排序后的序列中，ri仍在rj之前，则称这种排序算法是稳定的；否则称为不稳定的。像**冒泡排序**，**插入排序**，**基数排序**，**归并排序**等都是稳定排序

**原地排序**：

  基本上**不需要额外辅助的的空间，允许少量额外的辅助变量**进行的排序。就是在原来的排序数组中比较和交换的排序。像**选择排序**，**插入排序**，**希尔排序**，**快速排序**，**堆排序**等都会有一项比较且交换操作(swap(i,j))的逻辑在其中，因此他们都是属于原地(原址、就地)排序，而合并排序，计数排序，基数排序等不是原地排序

**1.冒泡排序**

**基本思想:**

  把第一个元素与第二个元素比较，如果第一个比第二个大，则交换他们的位置。接着继续比较第二个与第三个元素，如果第二个比第三个大，则交换他们的位置…   我们对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样一趟比较交换下来之后，排在最右的元素就会 是最大的数。除去最右的元素，我们对剩余的元素做同样的工作，如此重复下去，直到排序完成。

**具体步骤**：

  1.比较相邻的元素。如果第一个比第二个大，就交换他们两个。

  2.对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

  3.针对所有的元素重复以上的步骤，除了最后一个。

  4.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。

**时间复杂度**：O(N2) **空间复杂度**：O(1) **稳定排序**:是 **原地排序**:是

代码语言：javascript

复制

Node \*BubbleSort(Node \*phead)

{

Node \* p = phead;

Node \* q = phead->next;

/\*有几个数据就-1；比如x 个i<x-1\*/

for(int i=0;i<5;i++)

{

while((q!=NULL)&&(p!=NULL))

{

if(p->data>q->data)

{

/\*头结点和下一节点的交换，要特殊处理，更新新的头head\*/

if (p == phead)

{

p->next = q->next;

q->next = p;

head = q;

phead = q;

/\*这里切记要把p,q换回来，正常的话q应该在p的前面，进行的是p,q的比较

\*但是经过指针域交换之后就变成q,p.再次进行下一次比较时，

\*就会变成q,p的数据域比较。假如原本p->data > q->data,则进行交换。变成q->data和p->data比较，

\*不会进行交换，所以排序就会错误。有兴趣的可以调试下。

\*/

Node\*temp=p;

p=q;

q=temp;

}

/\*处理中间过程，其他数据的交换情况，要寻找前驱节点if (p != phead)\*/

else

{

/\*p，q的那个在前，那个在后。指针域的连接画图好理解一点\*/

if (p->next == q)

{

/\*寻找p的前驱节点\*/

Node \*ppre = FindPreNode(p);

/\*将p的下一个指向q的下一个\*/

p->next = q->next;

/\*此时q为头结点，让q的下一个指向p，连接起来\*/

q->next = p;

/\*将原来p的前驱节点指向现在的q，现在的q为头结点\*/

ppre->next = q;

Node\*temp=p;

p=q;

q=temp;

}

else if (q->next == p)

{

Node \*qpre = FindPreNode(q);

q->next = p->next;

p->next = q;

qpre->next = p;

Node\*temp=p;

p=q;

q=temp;

}

}

}

/\*地址移动\*/

p = p->next;

q = q->next;

}

/\*进行完一轮比较后，从头开始进行第二轮\*/

p = phead;

q = phead->next;

}

head = phead;

return head;

}

**2.快速排序**

**基本思想**

  我们从数组中选择一个元素，我们把这个元素称之为中轴元素吧，然后把数组中所有小于中轴元素的元素放在其左边， 所有大于或等于中轴元素的元素放在其右边，显然，此时中轴元素所处的位置的是有序的。也就是说，我们无需再移动中轴 元素的位置。

  从中轴元素那里开始把大的数组切割成两个小的数组(两个数组都不包含中轴元素)，接着我们通过递归的方式，让中轴元素 左边的数组和右边的数组也重复同样的操作，直到数组的大小为1，此时每个元素都处于有序的位置。

**具体步骤**：

  1.从数列中挑出一个元素，称为 “基准”（pivot）;

  2.重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

  3.递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；

**时间复杂度**：O(NlogN) **空间复杂度**：O(logN) **稳定排序**：否 **原地排序**：是

代码语言：javascript

复制

int \*QuickSort(Node\* pBegin, Node\* pEnd)

{

if(pBegin == NULL || pEnd == NULL || pBegin == pEnd)

return 0;

//定义两个指针

Node\* p1 = pBegin;

Node\* p2 = pBegin->next;

int pivot = pBegin->data;

//每次只比较小的，把小的放在前面。经过一轮比较后，被分成左右两部分。其中p1指向中值处，pbegin为pivot。

while(p2 != NULL)/\* && p2 != pEnd->next \*/

{

if(p2->data < pivot)

{

p1 = p1->next;

if(p1 != p2)

{

SwapData(&p1->data, &p2->data);

}

}

p2 = p2->next;

}

/\*此时pivot并不在中间，我们要把他放到中间，以他为基准，把数据分为左右两边\*/

SwapData(&p1->data, &pBegin->data);

//此时p1是中值节点

//if(p1->data >pBegin->data)

QuickSort(pBegin, p1);

//if(p1->data < pEnd->data)

QuickSort(p1->next, pEnd);

}

**3.插入排序**

**基本思想**:每一步将一个待排序的记录，插入到前面已经排好序的有序序列中去，直到插完所有元素为止。

**具体步骤**：

  1.将待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列；

  2.取出下一个元素，在已经排序的元素序列中从后向前扫描；

  3.如果该元素（已排序）大于新元素，将该元素移到下一位置；

  4.重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

  5.将新元素插入到该位置后；

  6.重复步骤2~5。

**时间复杂度**：O(N2) **空间复杂度**：O(1) **稳定排序**：是 **原地排序**：是

代码语言：javascript

复制

/\*不好理解可以调试下看下具体过程\*/

Node \*InsertSort(Node \*phead)

{

/\*为原链表剩下用于直接插入排序的节点头指针\*/

Node \*unsort;

/\*临时指针变量：插入节点\*/

Node \*t;

/\*临时指针变量\*/

Node \*p;

/\*临时指针变量\*/

Node \*sort;

/\*原链表剩下用于直接插入排序的节点链表：可根据图12来理解。\*/

unsort = phead->next;

/\*只含有一个节点的链表的有序链表：可根据图11来理解。\*/

head->next = NULL;

/\*遍历剩下无序的链表\*/

while (unsort != NULL)

{

/\*注意：这里for语句就是体现直接插入排序思想的地方\*/

/\*无序节点在有序链表中找插入的位置\*/

/\*跳出for循环的条件：

\*1.sort为空，此时，sort->data < t->data，p存下位置，应该放在有序链表的后面

\*2.sort->data > t->data ，跳出循环时，t->data放在有序链表sort的前面

\*3.sort为空 sort->data > t->data，也插入在sort前面的位置

\*/

/\*q为有序链表\*/

for (t = unsort, sort = phead; ((sort != NULL) && (sort->data < t->data)); p = sort, sort = sort->next);

/\*退出for循环，就是找到了插入的位置插入位置要么在前面，要么在后面\*/

/\*注意：按道理来说，这句话可以放到下面注释了的那个位置也应该对的，但是就是不能。原因：你若理解了上面的第3条，就知道了。\*/

/\*无序链表中的第一个节点离开，以便它插入到有序链表中。\*/

unsort = unsort->next;

/\*插在第一个节点之前\*/

/\*sort->data > t->data\*/

/\*sort为空 sort->data > t->data\*/

if (sort == phead)

{

/\*整个无序链表给phead\*/

phead = t;

}

/\*p是sort的前驱，这样说不太确切，当sort到最后时，for里面有个sort = sort->next，

\*就会把sort置空，所以要用p暂存上一次sort的值。而且执行判断sort->data < t->data时，用的也是上一次的sort

\*/

/\*sort后面插入\*/

/\*sort遍历到了最后，此时，sort->data < t->data,sort和p都为最后一个元素。\*/

else

{

p->next = t;

}

/\*if处理之后，t为无序链表，因为要在phead前插入。这里先把t赋值给phead，再把t的next指向sort，

\*就完成了在sort之前插入小的元素，很巧妙的一种方法

\*else处理完之后，sort存放的是sort的下一次，真正的sort存放在p中。不满足条件跳出循环时，判断的是下一次的sort，

但是我们要用的插入的位置为上一次的sort，所以要记录下sort上一次的位置

\*/

/\*完成插入动作\*/

/\*当sort遍历完成为空时，t->next就是断开后面的元素（sort为空）\*/

/\*当sort不为空时，sort->data > t->data，sort存放的元素比t要大，放在后面，t->next就是再链接起来sort\*/

t->next = sort;

/\*unsort = unsort->next;\*/

}

head = phead;

return phead;

}

**4.选择排序**

**基本思想**：首先，找到数组中最小的那个元素，其次，将它和数组的第一个元素交换位置(如果第一个元素就是最小元素那么它就和自己交换)。其次，在剩下的元素中找到最小的元素，将它与数组的第二个元素交换位置。如此往复，直到将整个数组排序。这种方法我们称之为选择排序。

**具体步骤**：

  1.首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。

  2.再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。

  3.重复第二步，直到所有元素均排序完毕。

**时间复杂度**：O(N2) **空间复杂度**：O(1) **稳定排序**：否 **原地排序**：是

代码语言：javascript

复制

Node\* SelectSort(Node\* phead)

{

Node \*temp;

Node \*temphead = phead;

/\*将第一次的最大值设置为头结点\*/

int max = phead->data;

/\*交换变量\*/

for(int i = 0;i<LengthList(phead);i++)

{

/\*每次遍历开始前都要把最大值设置为头结点\*/

max = phead->data;

while (temphead->next !=NULL)

{

/\*寻找最大值\*/

if(max < temphead->next->data)

{

max = temphead->next->data;

}

/\*移动指针位置\*/

temphead = temphead->next;

}

/\*找到最大值的位置\*/

temp = FindList(max);

/\*判断最大值是否和头节点相同\*/

if(phead != temp)

{

SwapNode(phead,temp);

}

/\*更新下一次遍历的头结点\*/

temphead = temp->next;

phead = temphead;

}

}

  SwapNode相关代码如下。

代码语言：javascript

复制

/\*交换相邻节点\*/

void Swanext(Node \*p,Node \*q)

{

/\*中间相邻节点\*/

if ((p != head)&&(q != head))

{

// /\*p为前一个节点，q的前驱为p\*/

// /\*寻找p的前驱结点\*/

// Node \*ppre = FindPreNode(p);

// Node \*temp;

// /\*暂存p节点的后继结点，指向q\*/

// temp = p->next;

// /\*将q节点的后继节点赋值给p的后继结点，即将p节点放到了q位置（此时q的前驱节点的next指针还指向的是q）\*/

// p->next = q->next;

// /\*将p节点给q的next，即将完成了q与p的重新连接\*/

// q->next =p;

// /\*找到原来p的前驱节点，指向q，即完成了原来p的前驱结点和q节点的连接\*/

// ppre->next =q;

if (p->next == q)

{

Node \*ppre = FindPreNode(p);

p->next = q->next;

q->next = p;

ppre->next = q;

// PrintList(head);

}

else if (q->next == p)

{

Node \*qpre = FindPreNode(q);

q->next = p->next;

p->next = q;

qpre->next = p;

}

}

/\*头结点相邻的交换\*/

else

{

if(p == head)

{

p->next = q->next;

q->next = p;

head = q;

}

else

{

q->next = p->next;

p->next = q;

head = p;

}

}

}

/\*交换头结点和任意节点(除尾节点外)\*/

void SwapHeadAnother(Node \*tmphead,Node \*p)

{

/\*寻找p的前驱节点\*/

Node \*ppre = FindPreNode(p);

Node \*temp;

if(p!=tmphead->next)

{

/\*暂存p节点\*/

temp = p->next;

/\*将tmphead节点的后继节点赋值给p的后继结点，即将tmphead节点放到了p位置（此时p的前驱节点的next指针还未断开）\*/

p->next = tmphead->next;

/\*将p的后继结点赋值给tmphead的后继结点，同时连接p的前驱和tmphead\*/

tmphead->next = temp;

ppre->next =tmphead;

/\*新的头结点返回给全局head\*/

head = p;

}

else

{

/\*头结点和下一节点\*/

tmphead->next = p->next;

p->next = tmphead;

head = p;

}

}

/\*交换尾结点和任意节点(除头节点外)\*/

void SwapEndAnother(Node \*tmpend,Node \*p)

{

/\*寻找p的前驱节点\*/

Node \*ppre = FindPreNode(p);

Node \*endpre = FindPreNode(tmpend);

Node \*temp;

if((tmpend==end)&&(p!=tmpend))

{

/\*暂存p节点\*/

temp = p->next;

/\*将tmpend节点的后继节点赋值给p的后继结点，即将tmpend节点放到了p位置（此时p的前驱节点的next指针还未断开）\*/

p->next = tmpend->next;

endpre->next = p;

/\*将p的后继结点赋值给tmpend的后继结点，同时连接p的前驱和tmpend（断开了之前的连接）\*/

tmpend->next = temp;

ppre->next =tmpend;

/\*新的头结点返回给全局head\*/

end = p;

}

else

{

p->next = tmpend->next;

tmpend->next = p;

end = p;

}

}

/\*交换头结点和尾节点\*/

void SwapHeadEnd(Node \*tmphead,Node \*tmpend)

{

/\*寻找tmpend的前驱节点\*/

Node \*endpre = FindPreNode(tmpend);

Node \*temp;

/\*暂存tmpend节点\*/

temp = tmpend->next;

/\*将tmphead节点的后继节点赋值给tmpend的后继结点，即将tmpend节点放到了tmphead位置（此时tmpend的前驱节点的next指针还未断开）\*/

tmpend->next = tmphead->next;

/\*将p的后继结点赋值给tmpend的后继结点，同时连接p的前驱和tmpend（断开了之前的连接）\*/

tmphead->next = temp;

endpre->next =tmphead;

/\*新的头结点返回给全局head\*/

head = tmpend;

end = tmphead;

// PrintList(tmpend);

}

void SwapRandom(Node \*p,Node \*q)

{

/\*除了首尾节点，中间不相邻的两个节点\*/

if((p->next != q)||(q->next != p))

{

/\*寻找前驱结点\*/

Node \*ppre = FindPreNode(p);

Node \*qpre = FindPreNode(q);

/\*借助一个中间节点传递数据域\*/

Node \*temp;

temp = p->next;

/\*交换p和q\*/

/\*2、p的新后继结点要变成q的原后继结点\*/

p->next = q->next;

/\*3、q的原前趋结点（qpre）的新后继结点要变成p\*/

qpre->next = p;

/\*4、q的新后继结点要变成p的原后继结点（p->next）\*/

q->next = temp;

/\*1、p的原前趋结点（ppre）的新后继结点要变成q\*/

ppre->next = q;

}

/\*中间相邻节点的处理\*/

else if (p->next == q)

{

Node \*ppre = FindPreNode(p);

p->next = q->next;

q->next = p;

ppre->next = q;

}

else if (q->next == p)

{

Node \*qpre = FindPreNode(q);

q->next = p->next;

p->next = q;

qpre->next = p;

}

}

/\*交换任意两个节点\*/

void SwapNode(Node\*p, Node\*q)

{

// if(LengthList(head)<2)

// printf("Can not swap!The Length of list is:%d\r\n ",LengthList(head));

/\*检查是否是头尾节点\*/

/\*对于头尾节点有四种情况

\*1.p头节点和q为中间节点

\*2.p尾节点和q为中间节点

\*3.q头节点和p为中间节点

\*4.q尾节点和p为中间节点

\*5.p头结点和q尾节点

\*6.q头结点和p尾节点

\*7.其他中间交换的情况

\*/

/\*2.两个节点是否相邻 除去头结点和下一节点相邻的情况，放在headanother处理\*/

if((p->next == q)&&(p !=head)&&(q !=head))

Swanext(p,q);

else if((q->next == p)&&(p !=head)&&(q !=head))

Swanext(q,p);

/\*1.p头节点和q为中间节点\*/

else if((p == head)&&(q != end))

SwapHeadAnother(p,q);

/\*2.p尾节点和q为中间节点\*/

else if ((p == end)&&(q != head))

SwapEndAnother(p,q);

/\*3.q头节点和p为中间节点\*/

else if((q == head)&&(p != end))

SwapHeadAnother(q,p);

/\*4.q尾节点和p为中间节点\*/

else if((q == end)&&(p != head))

SwapEndAnother(q,p);

/\*5.p头结点和q尾节点\*/

else if((p == head)&&(q == end))

SwapHeadEnd(p,q);

/\*6.q头结点和p尾节点\*/

else if((q == head)&&(p == end))

SwapHeadEnd(q,p);

/\*7.其他中间交换的情况\*/

else

SwapRandom(p,q);

}

**5.归并排序**

**基本思想**：归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。 作为一种典型的分而治之思想的算法应用，归并排序的实现由两种方法： 自上而下的递归（所有递归的方法都可以用迭代重写，所以就有了第 2 种方法）； 自下而上的迭代；

**具体步骤**：

  1.申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列；

  2.设定两个指针，最初位置分别为两个已经排序序列的起始位置；

  3.比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置；

  4.重复步骤 3 直到某一指针达到序列尾；

  5.将另一序列剩下的所有元素直接复制到合并序列尾。

**时间复杂度**：O(NlogN) **空间复杂度**：O(N) **稳定排序**：是 **原地排序**：否

代码语言：javascript

复制

/\*获取链表中间元素\*/

Node \*getMiddleNode(Node \*pList)

{

if (pList == NULL)

{

return NULL;

}

Node \*pfast = pList->next;

Node \*pslow = pList;

while (pfast != NULL)

{

pfast = pfast->next;

if (pfast != NULL)

{

pfast = pfast->next;

pslow = pslow->next;

}

}

return pslow;

}

/\*合并有序链表，合并之后升序排列\*/

Node \*MergeList(Node \*p1, Node \*p2)

{

if (NULL == p1)

{

return p2;

}

if (NULL == p2)

{

return p1;

}

Node \*pLinkA = p1;

Node \*pLinkB = p2;

Node \*pTemp = NULL;

/\*较小的为头结点，pTemp存下头结点\*/

if (pLinkA->data <= pLinkB->data)

{

pTemp = pLinkA;

pLinkA = pLinkA->next;

}

else

{

pTemp = pLinkB;

pLinkB = pLinkB->next;

}

/\*初始化头结点，即头结点指向不为空的结点\*/

Node \*pHead = pTemp;

while (pLinkA && pLinkB)

{

/\*合并先放小的元素\*/

if (pLinkA->data <= pLinkB->data)

{

pTemp->next = pLinkA;

/\*保存下上一次节点。如果下一次为NULL，保存的上一次的节点就是链表最后一个元素\*/

pTemp = pLinkA;

pLinkA = pLinkA->next;

}

else

{

pTemp->next = pLinkB;

pTemp = pLinkB;

pLinkB = pLinkB->next;

}

}

/\*跳出循环时，有一个为空。把不为空的剩下的部分插入链表中\*/

pTemp->next = pLinkA ? pLinkA:pLinkB;

head = pHead;

return pHead;

}

Node \*MergeSort(Node \*pList)

{

if (pList == NULL || pList->next == NULL)

{

return pList;

}

/\*获取中间结点\*/

Node \*pMiddle = getMiddleNode(pList);

/\*链表前半部分，包括中间结点\*/

Node \*pBegin = pList;

/\*链表后半部分\*/

Node \*pEnd = pMiddle->next;

/\*必须赋值为空 相当于断开操作。pBegin--pMiddle pEnd---最后 \*/

pMiddle->next = NULL;

/\*排序前半部分数据，只有一个元素的时候停止，即有序\*/

pBegin = MergeSort(pBegin);

/\*排序后半部分数据 递归理解可以参考PrintListRecursion;\*/

pEnd = MergeSort(pEnd);

/\*合并有序链表\*/

return MergeList(pBegin, pEnd);

}