

动态变量的应用—动态数组

- ▶ 对输入的若干个数进行排序，如果输入时先输入数的个数，然后再输入各个数，可用动态数组表示这些数：

```
int n;  
int *p;  
cin >> n;  
p = new int[n];  
for (int i=0; i<n; i++)  
    cin >> p[i];  
sort(p,n);  
.....  
delete []p;
```



- 对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1），这时，可以按以下方式实现：

```
const int INCREMENT=10;
int max_len=20,count=0,n,*p=new int[max_len];
cin >> n;
while (n != -1)
{
    if (count >= max_len)
    {
        max_len += INCREMENT;
        int *q=new int[max_len];
        for (int i=0; i<count; i++) q[i] = p[i];
        delete []p;
        p = q;
    }
    p[count] = n;
    count++;
    cin >> n;
}
sort(p,count);
.....
delete []p;
p= NULL;
```

开始不知道数据规模，因此预设一定量，
但当预设的数值仍不能满足要求时☹，
需要对数组“扩长”，牵涉“数据转移”☹...
如何解决该问题？

6 复杂数据的描述—构造数据类型

6.6 链 表

郭延文

2019级计算机科学与技术系

▶ 以上例题对于“**不确定个数**”输入数据排序的实现方法虽然可行，但是：

▶ 当数组空间不够时，需要重新申请空间、进行数据转移以及释放原有的空间（切记！），这样做比较麻烦并且效率有时不高

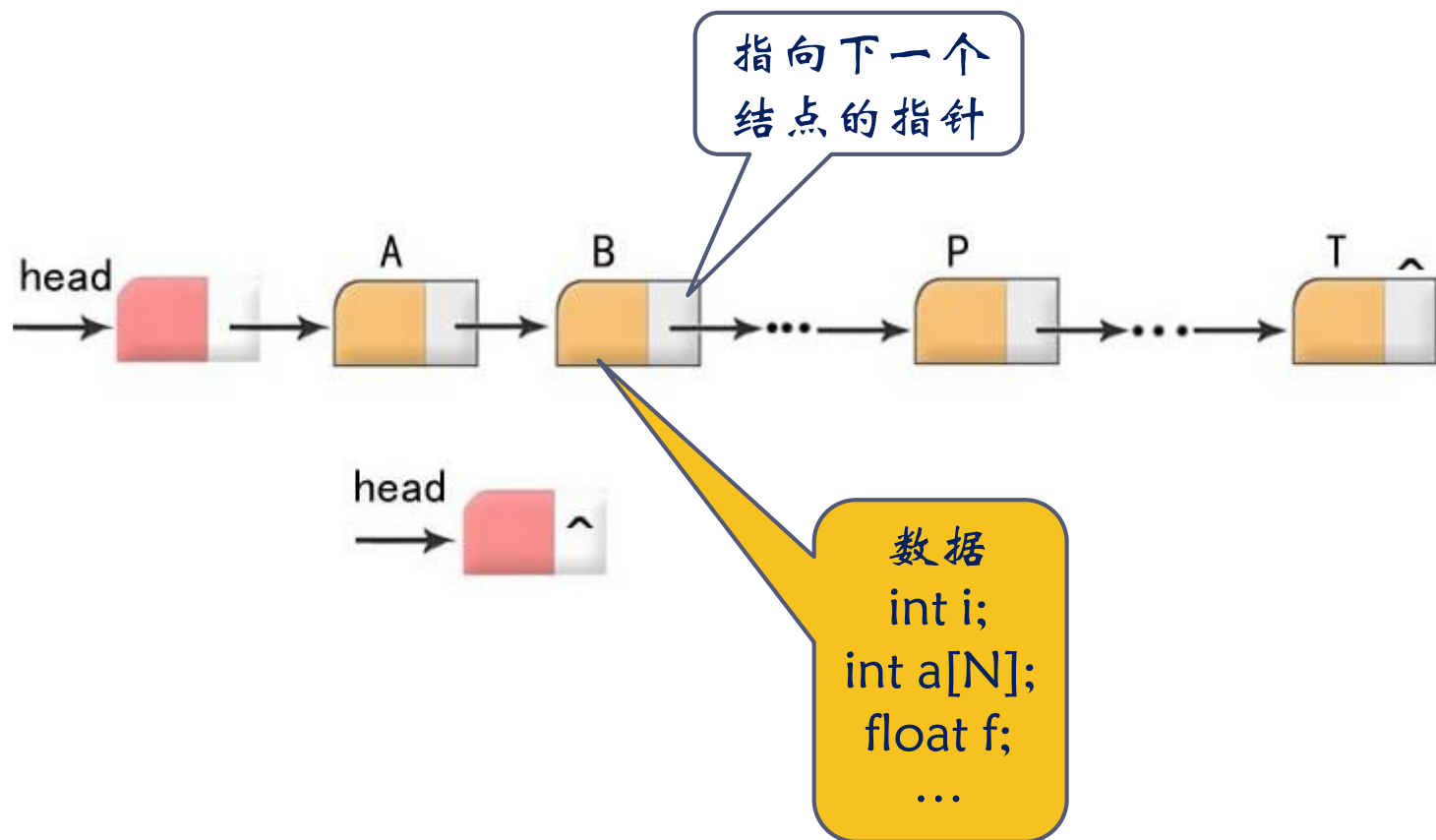
▶ 当需要在数组中增加或删除元素时，还将会面临数组元素的大量移动问题。

▶ **链表**可以避免数组的上述问题

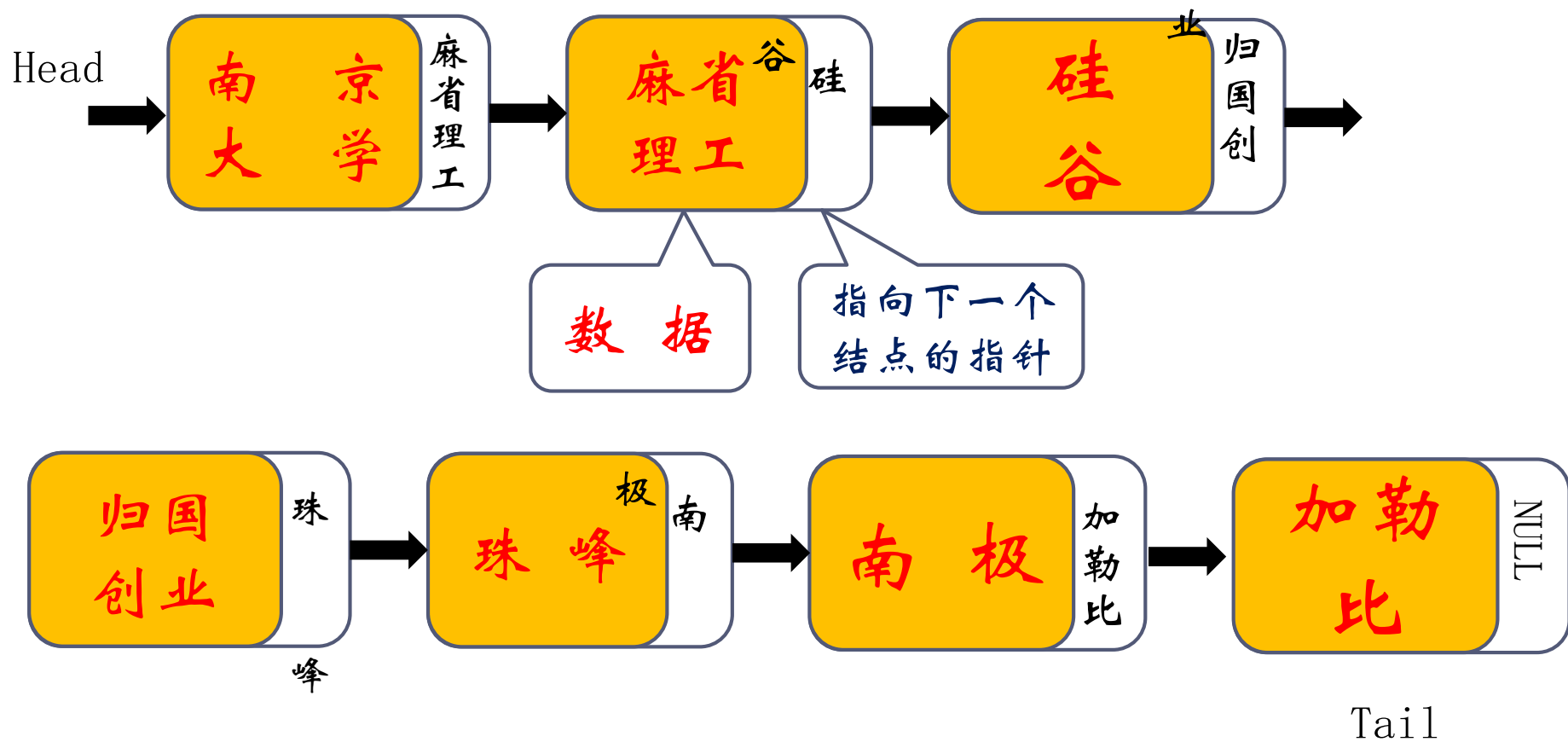
寻宝游戏



链表



链表



好好学习 程序设计!

动态变量的应用——链表

- ▶ 链表用于表示由若干个同类型的元素所构成的具有线性结构的复合数据。
 - ▶ 链表元素在内存中不必存放在连续的空间内。
 - ▶ 链表中的每一个元素除了本身的数据外，还包含一个（或多个）指针，它（们）指向链表中下一个（前一个或其它）元素。
 - ▶ 如果每个元素只包含一个指针，则称为单链表，否则称为多链表。
-

联合 (union) 类型

```
union myType  
{  
    int i;  
    char c;  
    double d;  
};  
myType v;
```

- ◆ 在程序中可以分时操作其中不同数据类型的成员

对联合变量的分时操作

- ▶ 对于上述联合变量v，在程序中可以分时操作其中不同数据类型的成员。比如，

```
v.i = 12;           //以下只操作变量v的成员i
```

.....

```
v.c = 'X';         //以下只操作变量v的成员c
```

.....

```
v.d = 12.95;       //以下只操作变量v的成员d
```

.....

- ▶ 当给一个联合变量的某成员赋值后，再访问该变量的另外一个成员，将得不到原来的值。比如，

```
v.i = 12;  
printf("%lf", v.d);           //不会输出12.95
```

- ▶ 即可以分时把v当作不同类型的变量来使用，但不可以同时把v当作不同类型的变量来使用。

借助联合实现“多态性”

- ▶ 联合类型使程序呈现出某种程度的多态性。这种多态性的好处是在提高程序可读性的同时可以实现多种数据共享内存空间。比如，

```
union Array
```

```
{
```

```
    int int_a[100];
```

```
    double dbl_a[100];
```

```
};
```

```
Array buffer;
```

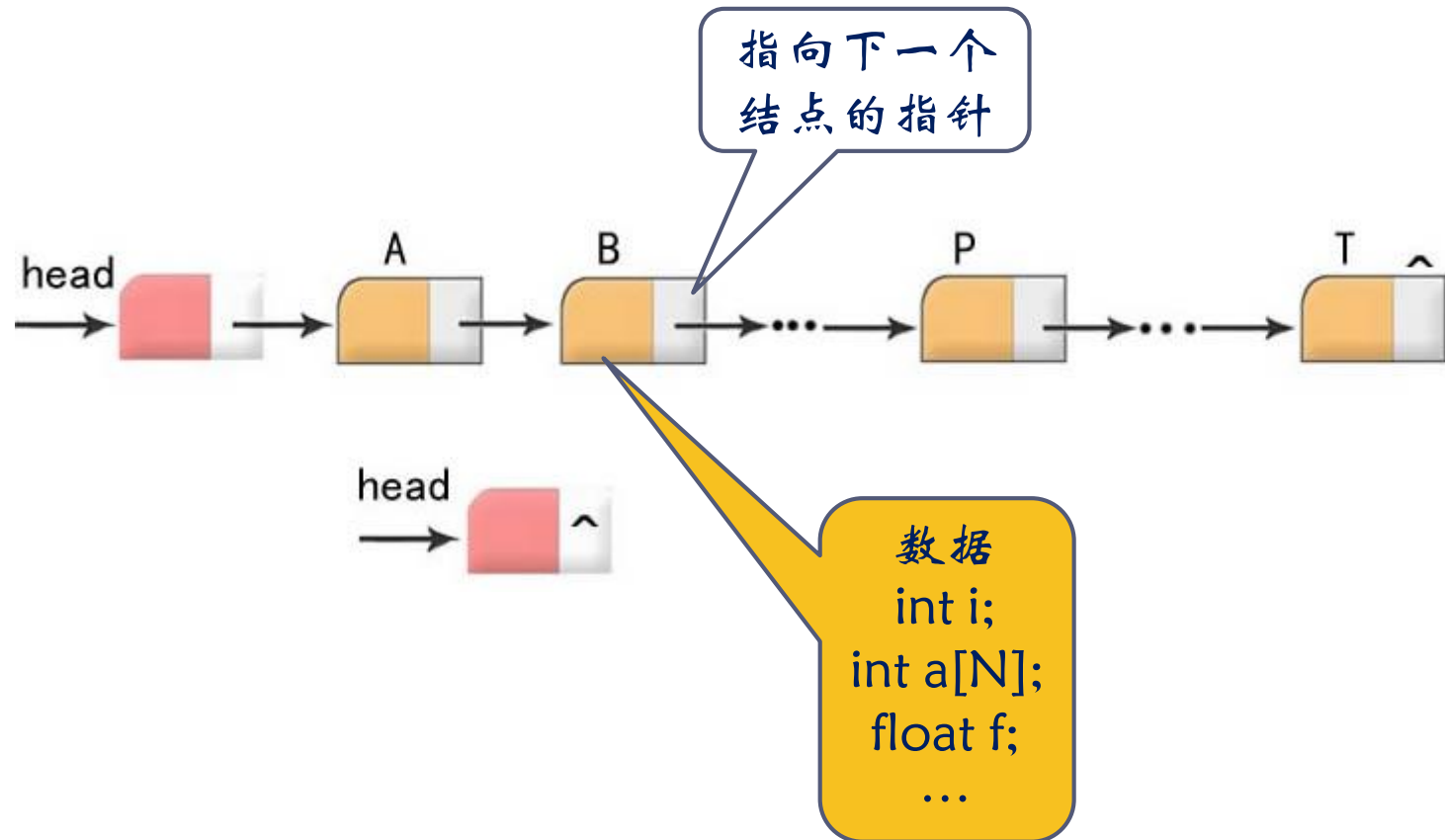
```
... buffer.int_a ... //使用数组int_a，有一半内存空间闲置
```

```
.....
```

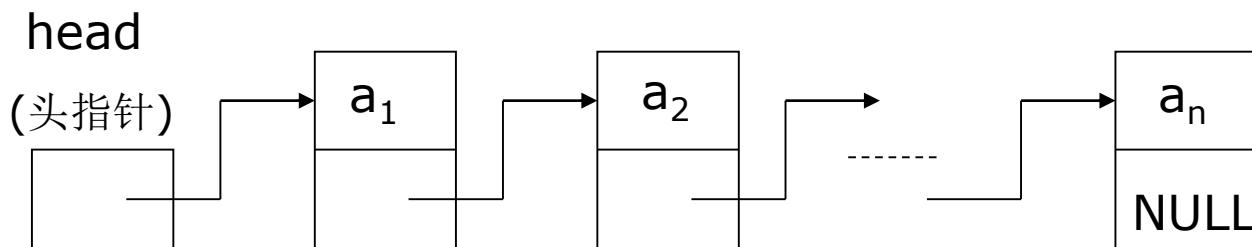
```
... buffer.dbl_a ... //使用数组dbl_a，没有内存空间闲置
```

```
.....
```

链表



单链表



- ▶ 单链表的每个元素只包含一个“指向相邻节点”的指针。
- ▶ 需要一个**头指针**，指向第一个元素。
- ▶ 单链表中的结点类型和表头指针变量可定义如下：

```
struct Node           //结点的类型定义
{
    int content;       //代表结点的数据
    Node *next;        //代表后一个结点的地址
};
```

```
Node *head=NULL; //头指针变量定义，初始状态下
```

▶ //为空值。NULL在cstdio中定义为0

链表操作

基本操作：

- ▶ 链表的建立
- ▶ 链表的遍历和元素定位（头、尾、特定位置和特定值）
- ▶ 链表中节点的插入（头、尾、特定位置）与删除
- ▶ 链表的输出与删除

衍生操作：

- ▶ 链表的反转
- ▶ 基于链表的排序
- ▶ 特定元素检索程序
- ▶ ...



基本操作：

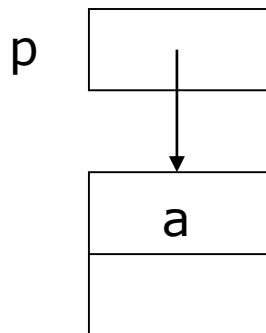
在链表中插入一个结点

- ▶ 首先产生一个新结点：

`Node *p=new Node;` // 产生一个动态变量来表示新结点

`p->content = a;` // 把a赋给新结点中表示结点值的成员

- ▶ 图示为：



该结点插入在什么位置？

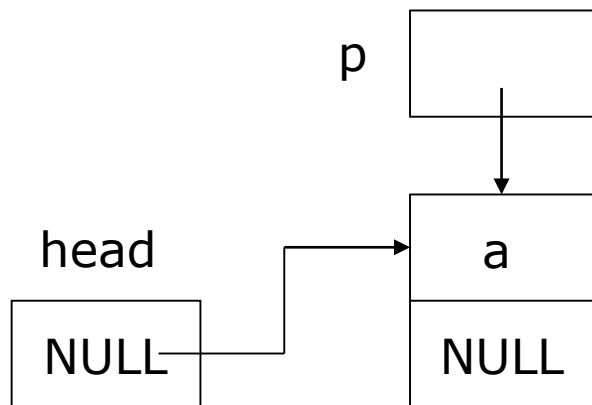
- ▶ 表头（链表头部）？
- ▶ 表尾（链表尾部）？
- ▶ 第 i 个结点的后面
- ▶ 该结点是否是链表的第一个结点（此前为空）？



- ▶ 如果链表为空（创建第一个结点时），则进行下面的操作：

```
if (head == NULL)           // 表头指针为空
{
    head = p;                // 头指针指向新结点。
    p->next = NULL;          // 等价于 head->next = NULL;
                              // 把新结点的next成员置为NULL。
}
```

- ▶ 图示为：

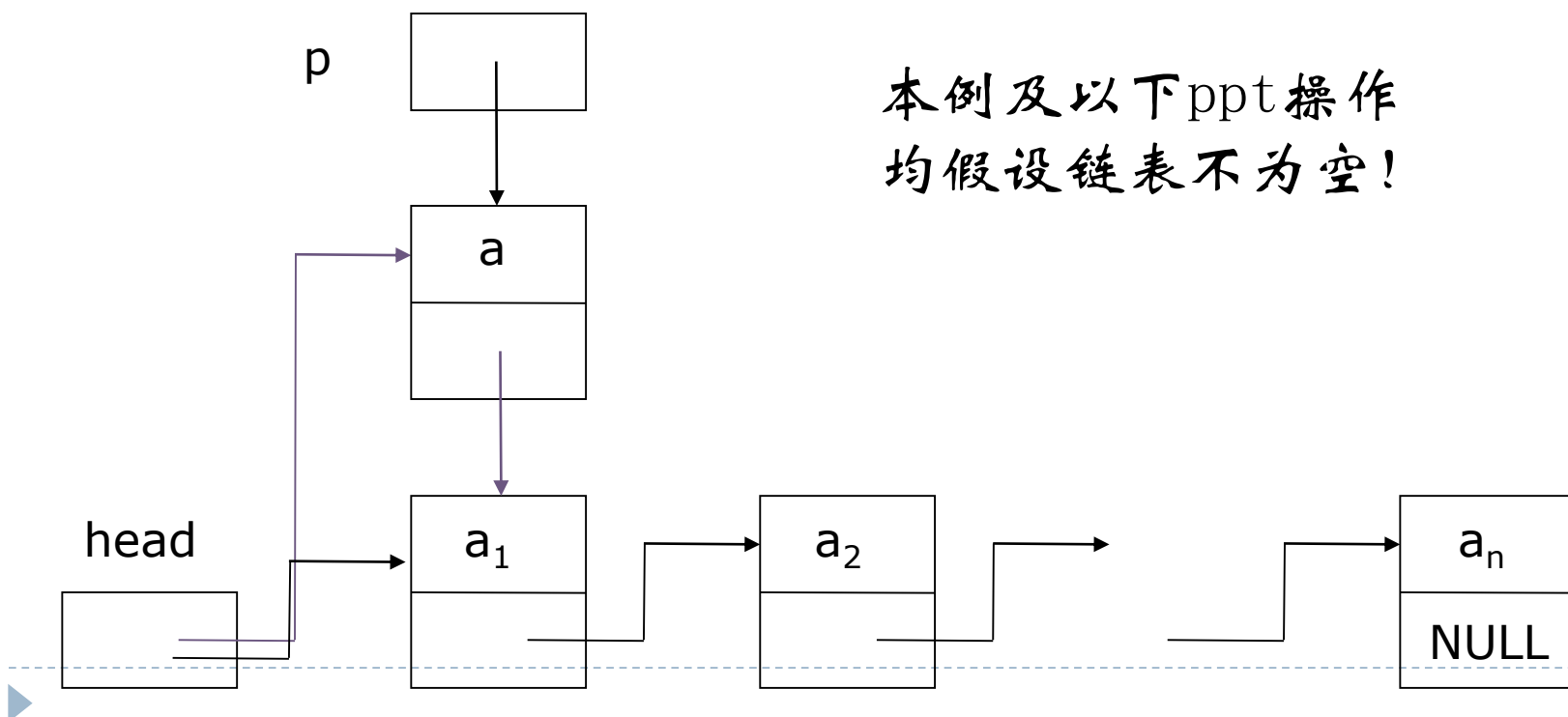


- ▶ 如果新结点插在表头，则进行下面的操作：

$p \rightarrow \text{next} = \text{head};$ // 把新结点的下一个结点指定为
// 链表原来的第一个结点。

$\text{head} = p;$ // 表头指针指向新结点

- ▶ 图示为：



- ▶ 如果新结点插在表尾，则进行下面的操作：

Node *q=head; //q指向第一个结点

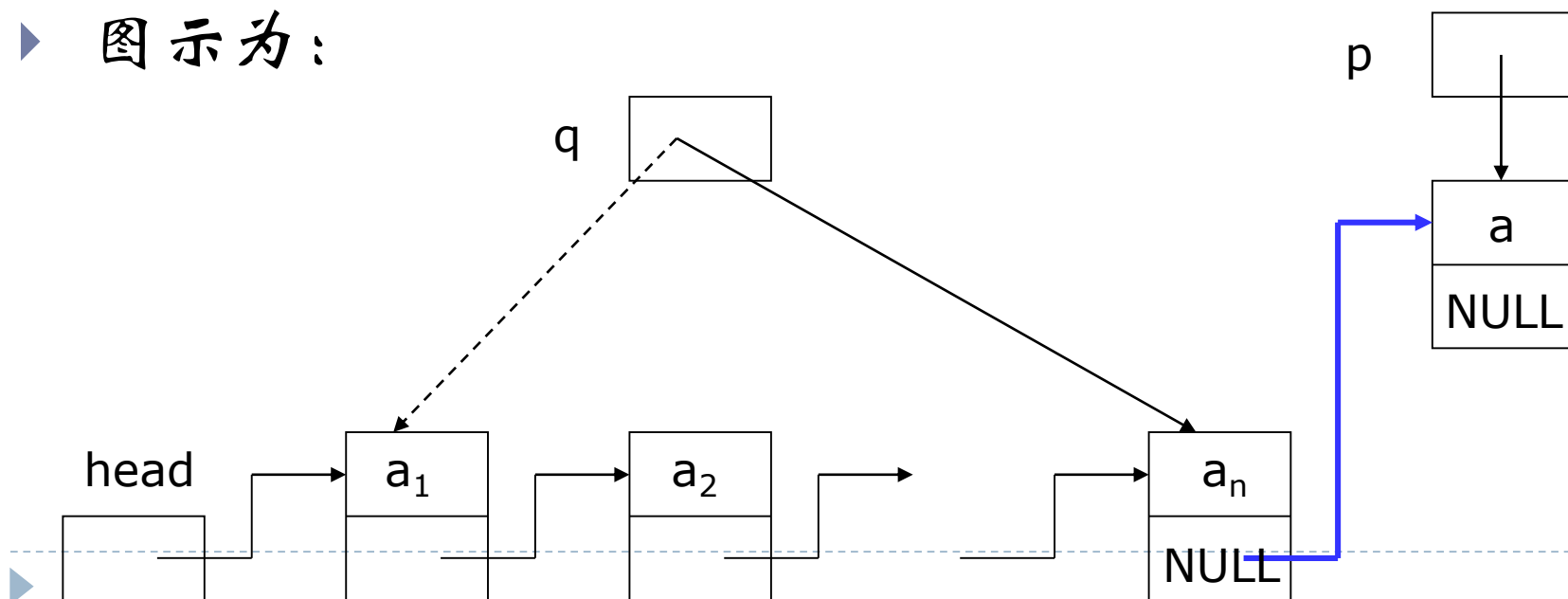
while (q->next != NULL) //循环查找最后一个结点

q = q->next; //循环结束后，q指向链表最后一个结点

q->next = p; //把新结点加到链表的尾部

p->next = NULL; //把新结点的next成员置为NULL

- ▶ 图示为：



- ▶ 如果新结点插在链表中第 i ($i > 0$) 个结点 (a_i) 的后面, 则进行下面的操作:

```
Node *q=head; //q指向第一个结点。
```

```
int j=1; //当前结点的序号, 初始化为1
```

```
while (j < i && q->next != NULL) //循环查找第i个结点
```

```
{   q = q->next;                //q指向下一个结点
```

```
    j++;                        //结点序号增加1
```

```
}
```

```
//循环结束时, q或者指向第i个结点, 或者指向最后一个结点 (结点数不够i时)。
```

```
if (j == i) //q指向第i个结点。
```

?

```
else //链表中没有第i个结点。
```

```
    cout << "没有第" << i << "个结点\n";
```

$p \rightarrow next = q \rightarrow next;$

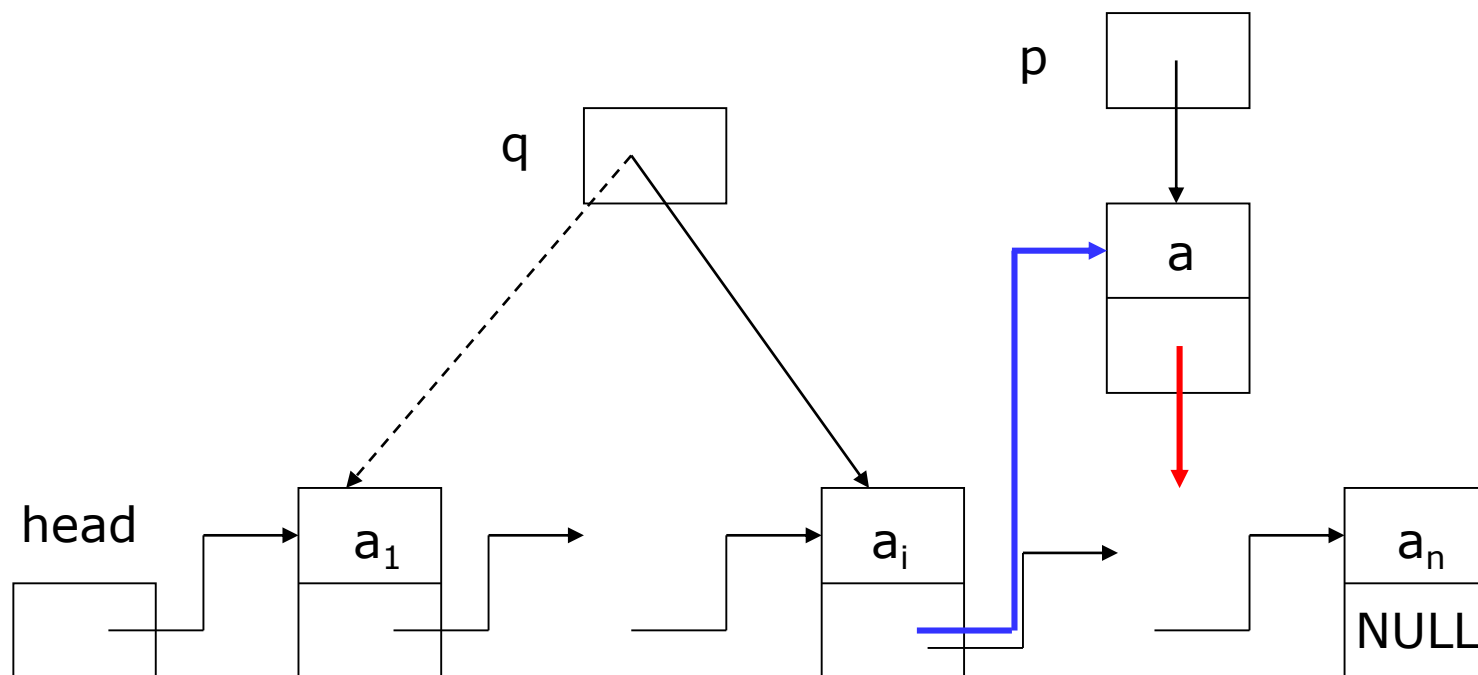
// 把q所指向结点的下一个结点指定为

// 新结点的下一个结点

$q \rightarrow next = p;$

// 把新结点指定为q所指向结点的下一个结点

► 图示为：



- ▶ 如果新结点插在链表中第 i ($i > 0$) 个结点 (a_i) 的后面, 则进行下面的操作:

Node *q=head; //q指向第一个结点。

```
int j=1; //当前结点的序号, 初始化为1
```

```
while (j < i && q->next != NULL) //循环查找第i个结点
```

```
{   q = q->next;                //q指向下一个结点
```

```
    j++;                        //结点序号增加1
```

```
}
```

//循环结束时, q或者指向第 i 个结点, 或者指向最后一个结点 (结点数不够 i 时)。

```
if (j == i) //q指向第i个结点。
```

```
{   p->next = q->next; //把q所指向结点的下一个结点指定为
```

```
    //新结点的下一个结点
```

```
    q->next = p;        //把新结点指定为q所指向结点的下一个结点
```

```
}
```

```
else //链表中没有第 $i$ 个结点。
```

```
    cout << "没有第" << i << "个结点\n";
```

基本操作：

在链表中删除一个结点

下面的操作假设链表不为空，即：head \neq NULL

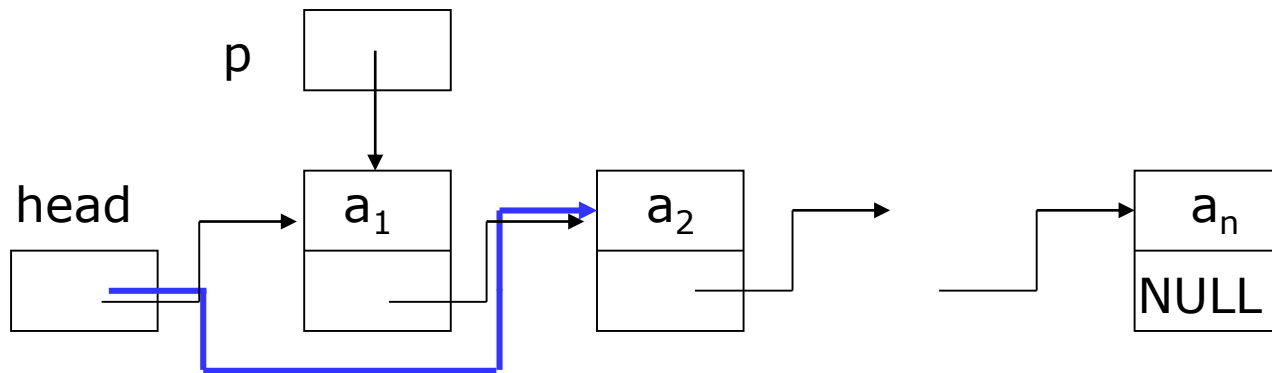
- ▶ 如果删除链表中第一个结点，则进行下面的操作：

Node *p=head; //p指向第一个结点。

head = head->next; //头指针指向第一个结点的下一个结点。

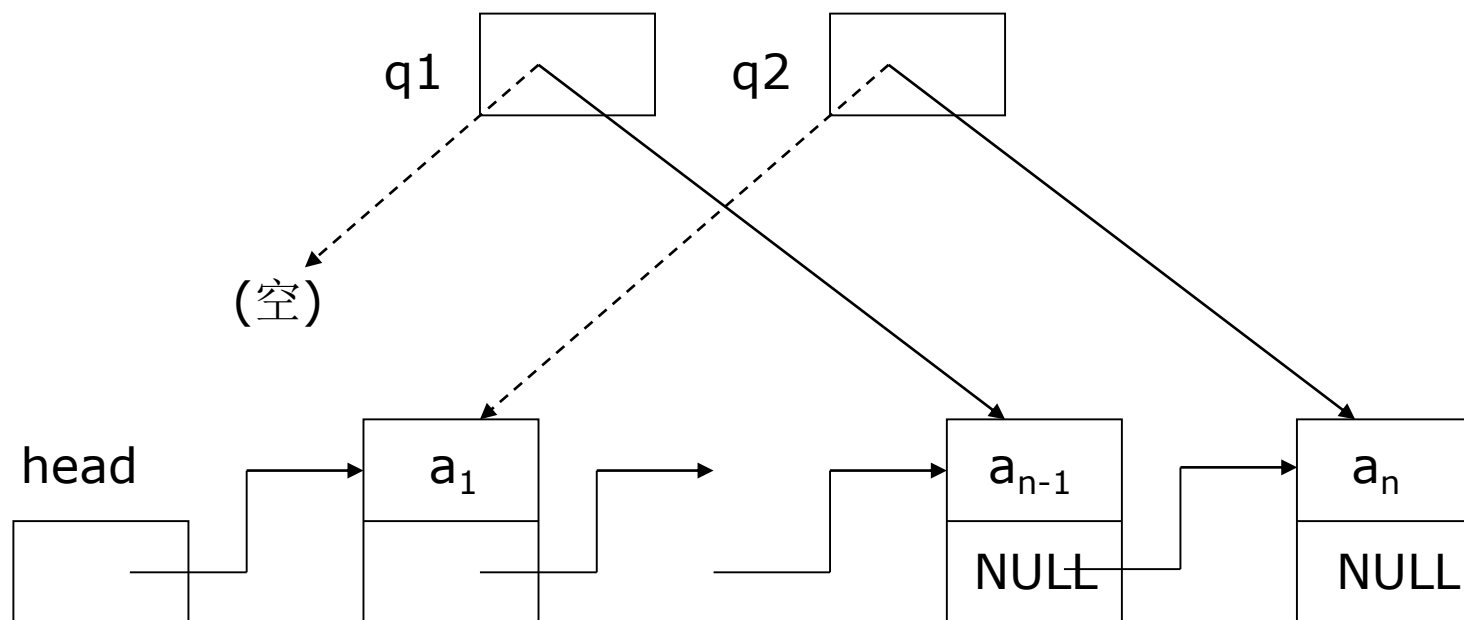
delete p; //归还删除结点的空间。

- ▶ 图示为：



如果删除链表的最后一个结点

► 图示为：



- ▶ 如果删除链表的最后一个结点，则进行下面的操作：

```
Node *q1=NULL,*q2=head;
```

```
//循环查找最后一个结点，找到后，q2指向它，q1指向它的前一个结点。
```

```
while (q2->next != NULL)
```

```
{ q1 = q2;
```

```
  q2 = q2->next;
```

```
}
```

```
if (q1 == NULL)    //链表中只有一个结点。
```

```
  head = NULL;    //把头指针置为NULL。
```

```
else //存在倒数第二个结点。
```

```
  q1->next = NULL; //把倒数第二个结点的next置为NULL。
```

```
delete q2;        //归还删除结点的空间。
```

▶ 如果删除链表中第 i ($i > 0$) 个结点 a_i , 则进行下面的操作:

if ($i == 1$) //要删除的结点是链表的第一个结点。

{

Node *p=head; //p指向第一个结点。

head = head->next; //head指向第一个结点

 //的下一个结点。

delete p; //归还删除结点的空间。

}

```
else //要删除的结点不是链表的第一个结点。
{ Node *p=head; //p指向第一个结点。
  int j=1; //当前结点的序号，初始化为1
  while (j < i-1) //循环查找第i-1个结点。
  { if (p->next == NULL)
    { break; //当没有下一个结点时，退出循环
      p = p->next; //p指向下一个结点
      j++; //结点序号加1
    }
    if (p->next != NULL) //链表中存在第i个结点。
    { Node *q=p->next; //q指向第i个结点。
      p->next = q->next; //把第i-1个结点的next
                        //改成第i个结点的next
      delete q; //归还第i个结点的空间。
    }
  }
  else //链表中没有第i个结点。
    cout << "没有第" << i << "个结点\n";
}
```

在链表中检索某个值a

```
int index=0;    //用于记住结点的序号，初始化为0
```

```
//从第一个结点开始遍历链表的每个结点查找值为a的结点。
```

```
for (Node *p = head; p != NULL; p = p->next)
```

```
{    index++;    //记住结点的序号，下面输出时需要。
```

```
    if (p->content == a)
```

```
        break;
```

```
}
```

```
if (p != NULL) //找到了
```

```
    cout << "第" << index << "个结点的值为：" << a << endl;
```

```
else //未找到
```

```
▶ cout << "没有找到值为" << a << "的结点\n";
```

例：基于链表的顺序查找程序

```
#include <stdio.h>
```

```
struct NodeStu
```

```
{  int id;
```

```
    float score;
```

```
    NodeStu *next;
```

```
};
```

```
extern NodeStu * AppCreate( );
```

```
extern float ListSearch(NodeStu *head, int x);
```

```
extern void DeleteList(NodeStu * head);
```

基于链表的信息检索

```
int main()  
{  NodeStu *head = AppCreateStu();           //建立链表  
    int x = 1001;  
    float y = ListSearch(head, x);           //在链表中查找指定id对应的score值  
    if(y < 0)  
        printf("没有找到学号为%d的同学的成绩.\n", x);  
    else  
        printf("学号为%d的同学的成绩为%f.\n", x, y);  
    DeleteList(head); //删除链表, 程序略  
    return 0;  
}
```

在链表中查找元素

```
float ListSearch(NodeStu *head, int x)
{
    NodeStu *p;
    for(p = head; p != NULL; p = p->next) //遍历链表，查找id为x的节点
        if(p->id == x)
            break;
    if(p != NULL) //找到了
        return p->score;
    else
        return -1.0;
}
```

删除整个链表

链表中的每个节点都是动态变量，所以在链表处理完(不再使用)后，需用程序释放整个链表（的每个结点）所占空间，即删除链表

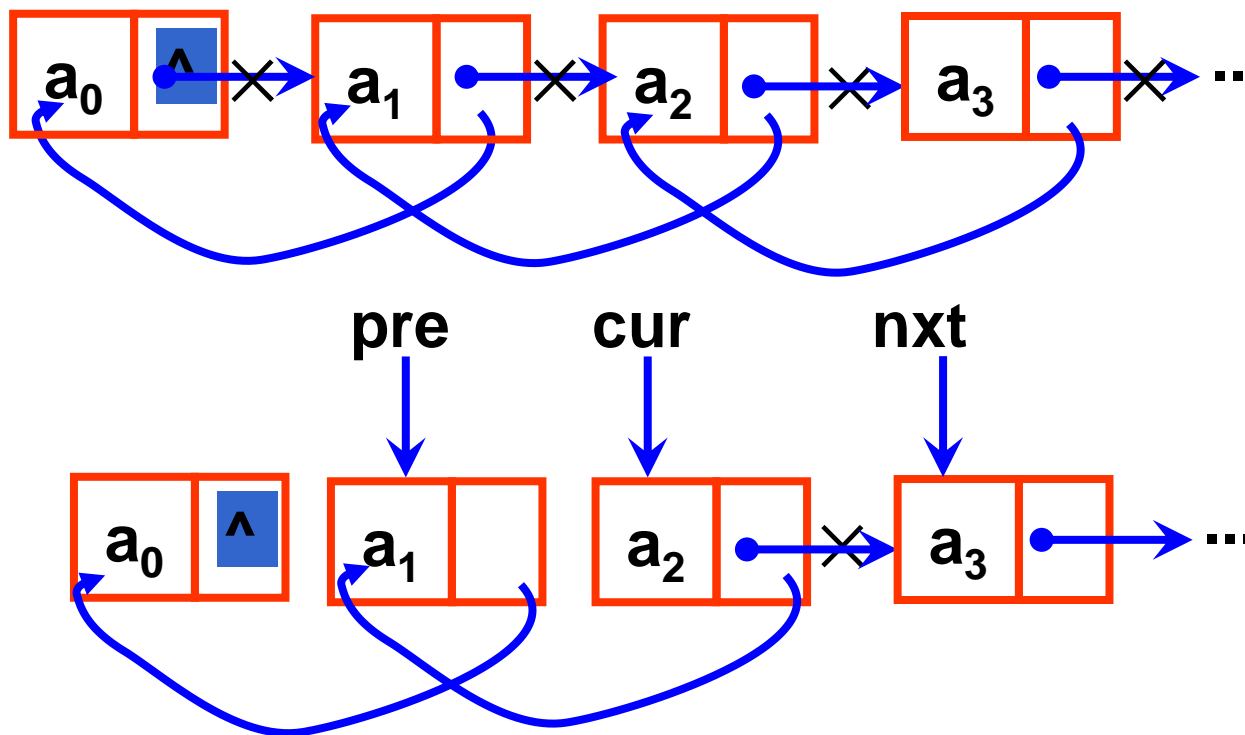
```
void DeleteList(NodeStu * head)
{
    while(head)
    {
        Node *current = head;
        head = head -> next;
        delete current;
    }
}
```



链表的反转

- ▶ 链表的反转指的是将链表的前后关系颠倒，头节点变成尾节点，尾节点变成头节点。

链表反转示意



反转的基本思想

- ▶ 解决这一问题的思路可以是：**依次将每个当前节点的前一个节点链接在当前节点的后面**
- ▶ 事先要将**当前节点**、**前一个节点**、**下一个节点**的地址存于三个指针变量中，以免被覆盖
- ▶ 其中头节点的前一个节点和尾节点的下一个节点都看作空地址

```
Node * Reverse(Node *head)
```

```
{
```

```
Node *pre = NULL;
```

```
Node *cur = NULL;
```

```
Node *nxt = head;
```

```
while(nxt != NULL)
```

```
{
```

```
pre = cur;
```

```
cur = nxt;
```

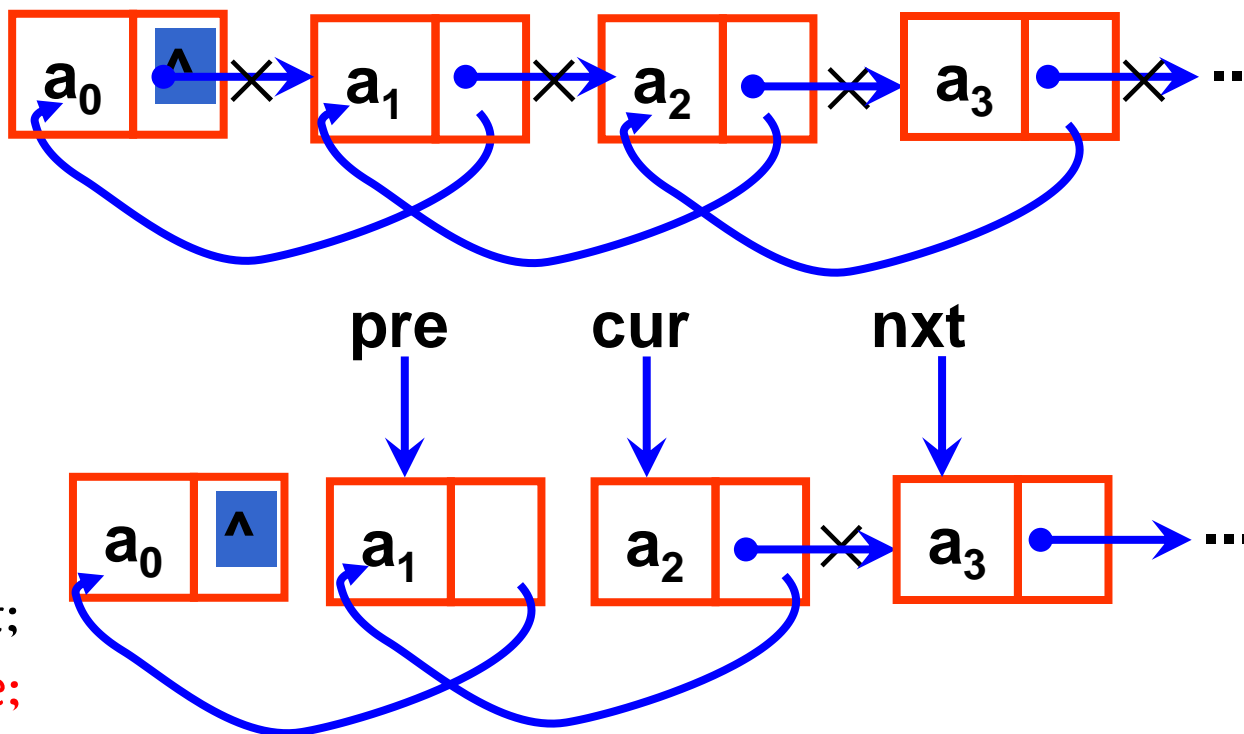
```
nxt = cur -> next;
```

```
cur -> next = pre;
```

```
}
```

```
return cur;
```


```
}
```



其实，用双向链表，易于实现链表的反转！


例：综合应用：对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1）

```
struct Node
{
    int content; //代表结点的数据
    Node *next; //代表后一个结点的地址
};
extern Node *input();           //输入数据，建立链表，返回链表的头指针
extern void sort(Node *h);      //排序
extern void output(Node *h);    //输出数据
extern void remove(Node *h);    //删除链表
int main()
{
    Node *head;
    head = input();
    sort(head);
    output(head);
    remove(head);
    return 0;
}
```



```
#include <iostream>
#include <cstdio>
using namespace std;
Node *input() //从表尾插入数据
{ Node *head=NULL, //头指针
  *tail=NULL; //尾指针

  int x;
  cin >> x;
  while (x != -1)
  { Node *p=new Node;
    p->content = x;
    p->next = NULL;
    if (head == NULL)
      head = p;
    else
      tail->next = p;
    tail = p;
    cin >> x;
  }
  return head;
}
```



```
#include <iostream>
#include <cstdio>
using namespace std;
Node *input() //从表头插入数据
{ Node *head=NULL; //头指针
  int x;
  cin >> x;
  while (x != -1)
  { Node *p=new Node;
    p->content = x;
    p->next = head;
    head = p;
    cin >> x;
  }
  return head;
}
```

特别注意!

建立链表，头指针作为形参

```
.....  
void input(Node *&h) //从表头插入数据，建立链表，h返回头指针  
{  
    int x;  
    cin >> x;  
    while (x != -1)  
    {  
        Node *p=new Node;  
        p->content = x;  
        p->next = h;  
        h = p;  
        cin >> x;  
    }  
}  
int main()  
{  
    Node *head=NULL;  
    input( head);  
    .....  
}
```

*&h

用链表实现选择排序法

回顾选择排序法

- 从N个数中选择最大者，与第N个数交换位置；然后从剩余的N-1个数中再选择最大者，与第N-1个数交换位置；...；直到只剩下一个数为止
- N个数选择N-1趟，每趟内比较的次数随趟数递减，且不是每次比较都进行元素交换操作。

5	2
4	4
0	0
2	5

2	2
4	0
0	4
5	5

2	0
0	2
4	4
5	5

if(max != i-1)

输入N个数给a[0]到a[N-1]

for(i=N; i>1; i--)

max=0

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

a[max]<a[j]

T

F

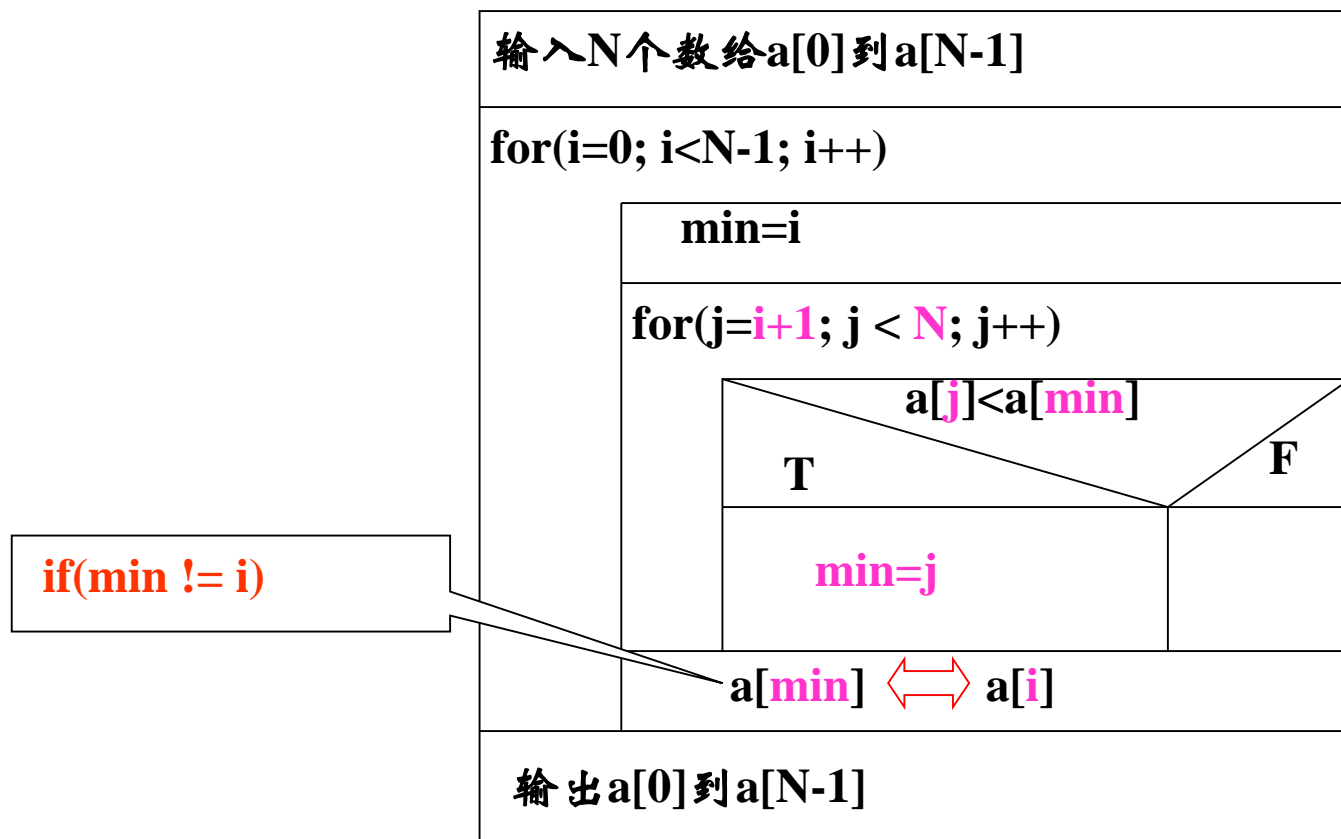
max=j

a[max] ↔ a[i-1]

输出a[0]到a[N-1]

选择排序法：小的往前放

- 从N个数中选择最小的，与第0个位置的数交换位置；然后从剩余的N-1个数中再选择最小的，与第1个数交换位置；...；直到只剩下一个数为止



```
void sort(Node *h) //采用选择排序，小的往前放
{ if (h == NULL || h->next == NULL) return;
  //从链表头开始逐步缩小链表的范围
```

```
for (Node *p1=h; p1->next != NULL; p1 = p1->next)
```

```
{
```

```
Node *p_min=p1; //p_min指向最小的结点，初始化为p1
```

```
//从p1的下一个开始与p_min进行比较
```

```
for (Node *p2=p1->next; p2 != NULL; p2=p2->next)
```

```
if (p2->content < p_min->content)
```

```
    p_min = p2;
```

```
if (p_min != p1)
```

```
{ int temp = p1->content;
```

```
  p1->content = p_min->content;
```

```
  p_min->content = temp;
```

```
}
```

```
}
```

```
}
```

只交换结构体的
content，没有改
变指针！看具体要
求是否需要改变指
针！

输入N个数给a[0]到a[N-1]

```
for(i=0; i<N-1; i++)
```

```
    min=i
```

```
    for(j=i+1; j < N; j++)
```

```
        if(a[j]<a[min])
```

```
            T
```

```
        F
```

```
            min=j
```

```
        a[min] ↔ a[i]
```

输出a[0]到a[N-1]

```
void output(Node *h)
{
    for (Node *p=h; p!=NULL; p=p->next)
        cout << p->content << ',';
    cout << endl;
}
```

```
void remove(Node *h)
{
    while (h != NULL) // 逐个元素归还空间
    {
        Node *p=h;
        h = h->next;
        delete p;
    }
}
```



再回顾：对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1）

```
struct Node
{
    int content; //代表结点的数据
    Node *next; //代表后一个结点的地址
};

extern Node *input();           //输入数据，建立链表，返回链表的头指针
extern void sort(Node *h);      //排序
extern void output(Node *h);    //输出数据
extern void remove(Node *h);    //删除链表

int main()
{
    Node *head;
    head = input();
    sort(head);
    output(head);
    remove(head);
    return 0;
}
```

更好的命名：

```
input -> create_list();
sort -> sort_data();
output -> output_list();
remove -> delete_list OR remove_data();
```

链表的构造与操作

基本操作：

- ▶ 链表的建立
- ▶ 链表的遍历和元素定位（头、尾、特定位置和特定值）
- ▶ 链表中节点的插入（头、尾、特定位置）与删除
- ▶ 链表的输出与删除

衍生操作：

- ▶ 链表的反转
- ▶ 基于链表的排序
- ▶ 特定元素检索程序

▶ ...

关键操作

- ▶ 搞清楚操作涉及的指向关系
 - ▶ 操作**前**谁（结点）指向谁（结点）
 - ▶ 操作**后**谁（结点）指向谁（结点）
 - ▶ 指针如何“赋值”、注意赋值**次序**！
- ▶ 如何定位特定（第*i*个/最后一个）元素
 - ▶ 从head开始向后循环 ($p = p \rightarrow \text{next}$)
 - ▶ 最后一个 ($p \rightarrow \text{next} == \text{NULL}$)
 - ▶ 注意特殊情况（边界条件）...



链表Coding Tips:

1. 能够确定头(head)、尾(tail)、第i个节点

(最后一个node->next为空)

2. 弄清指向关系和操作顺序!

3. 留意特殊情况!

(例如: 为空的初始情况或链表只有1个节点)

4. 注意归还空间!



调试Tips

1. 对链表调试，重要的依然是断点（cout输出做辅助）

“数据为王”，把数据（content、甚至有时还有地址）打印出来可能会好点…

当牵涉Node交换（不仅是`p->content`），地址有时也要查看

2. 程序不是单靠“瞪大眼睛”看出来的，更重要的是针对测试用例，关键步骤的输出是否符合我们的“大脑”推理、手工计算

3. （对于复杂程序和测试用例）除非你有极强的逻辑思维和记忆力，否则请拿起笔和纸来…



调试Tips

```
struct Node
{
    int content; //代表结点的数据
    Node *next; //代表后一个结点的地址
};

int main()
{
    Node *head;
    head = input();
    sort(head);
    output(head);
    remove(head);
    return 0;
}
```

发现最后结果不对：

1. (Head, hand) 记下不对的例子（测试用例）
2. 从后向前（或从前往后）排查每个函数的输出是否正确，以确定那个函数的问题
3. 发现sort不对，也有可能是input的问题
- ...



调试Tips

```
void sort(Node *h) //采用选择排序，小的往前放
```

```
{ if (h == NULL || h->next == NULL) return;
```

```
    //从链表头开始逐步缩小链表的范围
```

```
    for (Node *p1=h; p1->next != NULL; p1 = p1->next)
```

```
{
```

```
    Node *p_min=p1; //p_min指向最小的结点，初始化为p1
```

```
    //从p1的下一个开始与p_min进行比较
```

```
    for (Node *p2=p1->next; p2 != NULL; p2=p2->next)
```

```
        if (p2->content < p_min->content)
```

```
            p_min = p2;
```

```
    if (p_min != p1)
```

```
{    int temp = p1->content;
```

```
    p1->content = p_min->content;
```

```
    p_min->content = temp;
```

```
}
```

```
}
```

```
}
```

当确定一个函数

有问题后…

Q & A

