
step further

专题

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与封装（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、**指针**、**结构**）

归纳与推广（程序设计的本质）

问题的提出

- (1) 对输入的10个整数进行排序，可以用数组存储这10个整数；
- (2) 对输入的若干个整数进行排序（先输入整数的个数 n ，后输入 n 个整数）
 - 1) 定义长度为 n 的数组来存储这 n 个整数（支持新版C语言标准的IDE）；
 - 2) 创建长度为 n 的动态数组来存储这 n 个整数；

```
int *pda = (int *)malloc(sizeof(int)*n); // new int[n]
```

- (3) 对输入的若干个正整数进行排序（先输入各个正整数，后输入一个结束标志 -1）
 - 1) 可以不断创建动态数组来存储这些整数；

扩容：max_len +=

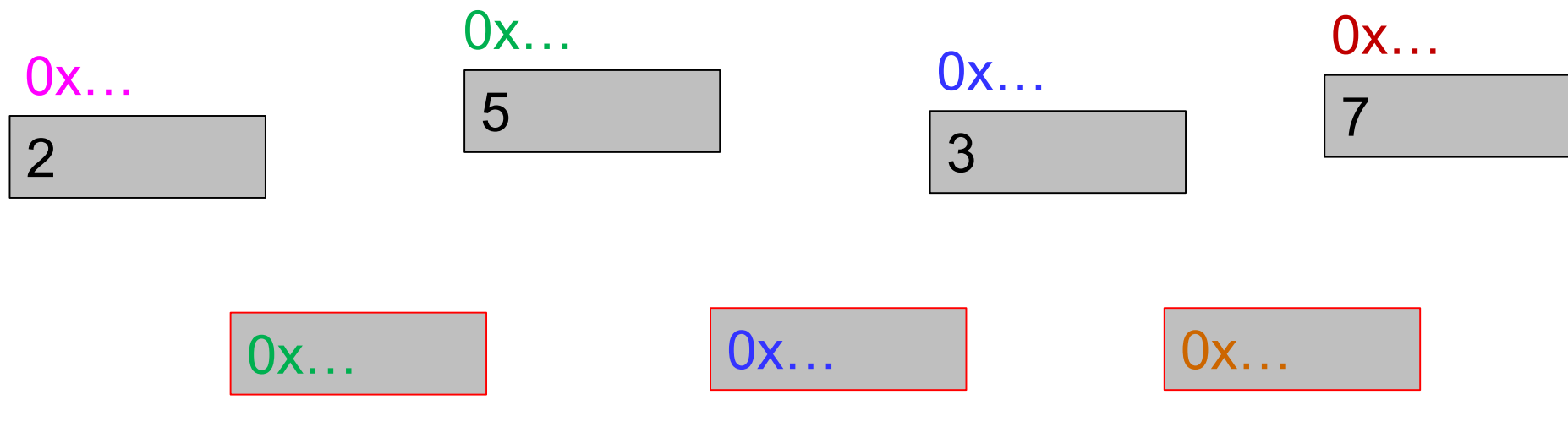
2) ? ? ? ?

创建时需要大量数据搬迁？
创建时找不到足够的连续空间？

输入一个数，插入到已经排好序的若干个数当中，保持原序？
删除一个数，保持原序？

解决办法

- 用指针变量把若干个分散的动态变量的地址存储起来

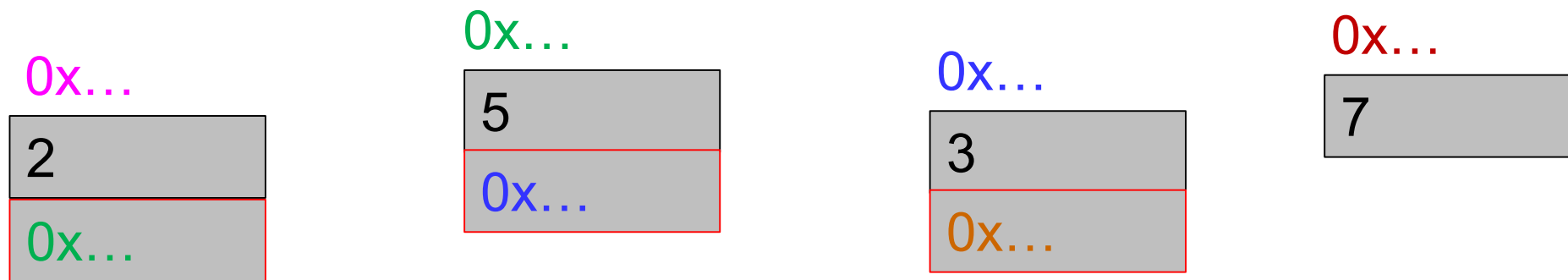


解决办法

```
typedef struct Node Node;
```

```
struct Node  
{  
    int data;  
    Node *next;  
};
```

- 用指针变量把若干个分散的动态变量的地址存储起来
 - 把下一个指针变量和上一个动态变量“捆绑”起来
- 动态的（整型变量 + 指针变量） → 动态的（结构体）



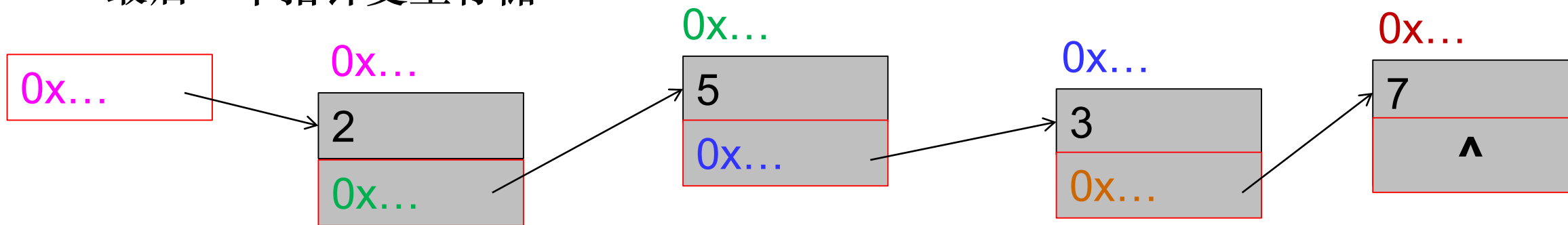
0x...

解决办法

- 用指针变量把若干个分散的动态变量的地址存储起来
 - 把下一个指针变量和上一个动态变量“捆绑”起来
- 动态的（整型变量 + 指针变量） → 动态的（结构体）

● 形成链表

- ◆ 第一个指针变量 通过 定义 放栈区
- ◆ 最后一个指针变量存储 NULL



- 内存的栈区：存放**定义的**基本类型变量、数组，指针变量、数组，结构体、形参…
- 内存的堆区（零星的空间）：存放**创建的**动态变量、动态数组

● 数组的缺陷:


- ◆ 数组的长度一般在写代码定义之前或程序执行创建之前就得确定，所占内存空间在其存储期内一般保持不变，数组元素有序且连续排列，定位和查询比较方便，但删除一个元素可能会引起大量数据的移动而降低效率，插入一个元素不仅可能会引起大量数据的移动，还可能会受到数组长度的制约。

● 链表

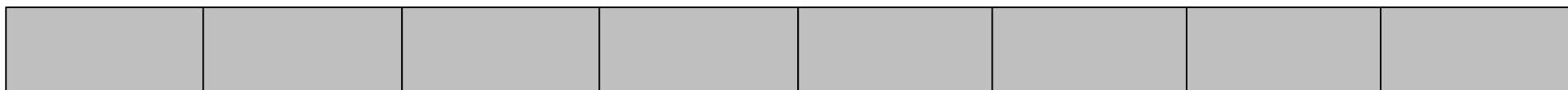
- ◆ 在程序运行期间，动态结构体（节点）的个数可以随机增加或减少、所占内存空间大小可以动态变化、数据在逻辑上有序地连续排列，而物理上并不需要占用连续的存储空间。
- ◆ 不过，链表不是一种数据类型，不是定义一下或创建一次就能形成，其创建、撤销代码比数组要复杂

数组 (array)

```
int a[10]
```



```
(int *)malloc(sizeof(int)*10); //new int[10];
```



链表 (list)



创建链表（单向链表）

- 链表这种数据结构，不是一种数据类型，不是定义一下或创建一次就能形成；需要**通过基于循环流程编写代码**将若干个同类型的数据（节点）链接起来。
- 链表中的每个节点通常是一个动态结构体，结构体中至少有一个指针类型成员。
- 单向链表只有一个指针类型成员，例如：

```
struct Node
{
    int data; //存储数据
    Node *next; //存储下一个节点的地址
};
```


创建链表

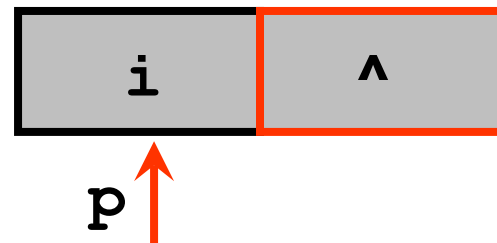
```
typedef struct Node Node;
```

• 创建一个节点

```
struct Node  
{  
    int data;  
    Node *next;  
};
```

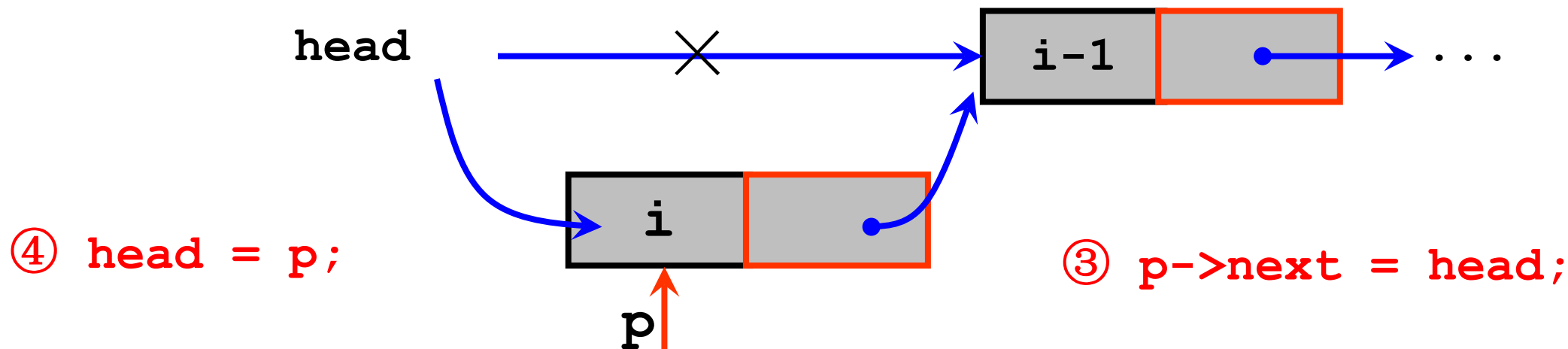
```
Node *p = (Node *)malloc(sizeof(Node)); // Node *p = new Node;
```

```
scanf("%d", &p->data); // p->data = i;  
p->next = NULL;
```



头部插入节点的方式创建链表

① `for(i=0; i<n; ++i)`



② `p = (Node *)malloc(sizeof(Node)); //p = new Node;`
`p->data = i;`

指针变量 赋值 地址



head

0x2000

```
Node *InsCreate(int n)
```

0x2000

0
^

```
{
```

```
Node *head = NULL;
```

```
for(int i = 0; i < n; ++i)
```

p

0x2000

```
{
```

```
Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;
```

```
p -> data = i; //也可给新节点的数据成员输入值
```

```
p -> next = head; //head的值赋给新节点的指针成员
```

```
head = p; //将p这个指针变量的值赋给指针变量head
```

```
}
```

```
return head;
```

```
}
```

head

0x2000

```
Node *InsCreate(int n)
```

```
{
```

```
    Node *head = NULL;
```

```
    for(int i = 0; i < n; ++i)
```

```
    {
```

```
        Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;
```

```
        p -> data = i; //也可给新节点的数据成员输入值
```

```
        p -> next = head; //head的值赋给新节点的指针成员
```

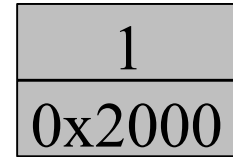
```
        head = p; //将p这个指针变量的值赋给指针变量head
```

```
    }
```

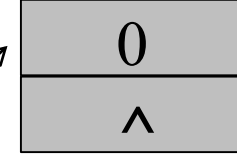
```
    return head;
```

```
}
```

0x3000



0x2000



p

0x3000

head

0x3000

```
Node *InsCreate(int n)
```

```
{
```

```
Node *head = NULL;
```

```
for(int i = 0; i < n; ++i)
```

```
{
```

```
Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;
```

```
p -> data = i; //也可给新节点的数据成员输入值
```

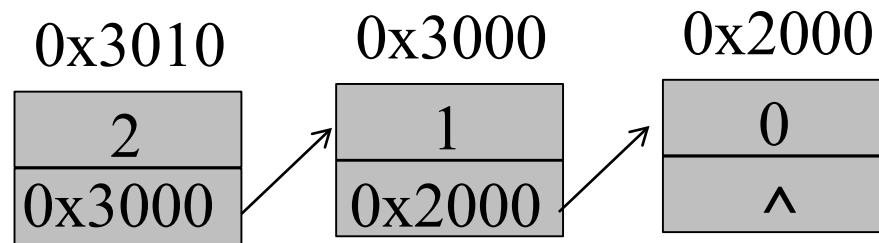
```
p -> next = head; //head的值赋给新节点的指针成员
```

```
head = p; //将p这个指针变量的值赋给指针变量head
```

```
}
```

```
return head;
```

```
}
```



p

0x3010

head
0x3010

```
Node *InsCreate(int n)
```

```
{
```

```
Node *head = NULL;
```

```
for(int i = 0; i < n; ++i)
```

```
{
```

```
Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;
```

```
p -> data = i; //也可给新节点的数据成员输入值
```

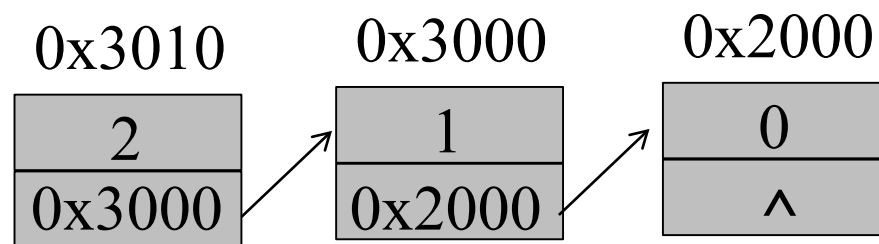
```
p -> next = head; //head的值赋给新节点的指针成员
```

```
head = p; //将p这个指针变量的值赋给指针变量head
```

```
}
```

```
return head;
```

```
}
```



head

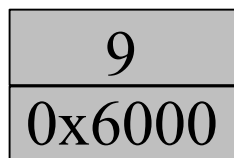
0x7000

```
Node *InsCreate(int n)
```

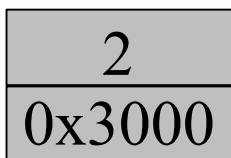
```
{
```

```
Node *head = NULL;
```

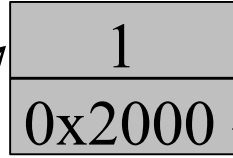
0x7000



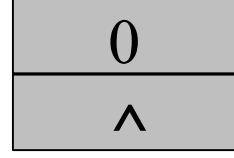
0x3010



0x3000



0x2000



```
for(int i = 0; i < n; ++i)
```

```
{
```

```
Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;
```

```
p -> data = i; //也可给新节点的数据成员输入值
```

```
p -> next = head; //head的值赋给新节点的指针成员
```

```
head = p; //将p这个指针变量的值赋给指针变量head
```

```
}
```

```
return head;
```

```
} // 倒序
```

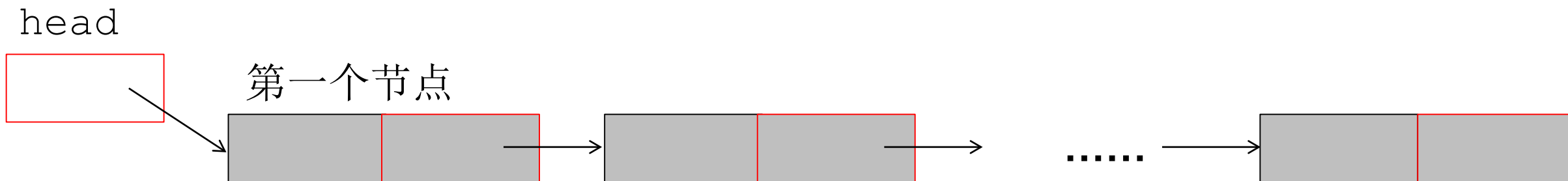
```
Node *InsCreate(int n)
{
    Node *head = NULL;

    for(int i = 0; i < n; ++i)
    {
        Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;
        p->data = i; //也可给新节点的数据成员输入值
        p->next = head; //head的值赋给新节点的指针成员
        head = p; //将p这个指针变量的值赋给指针变量head
    }
    return head;
}
```

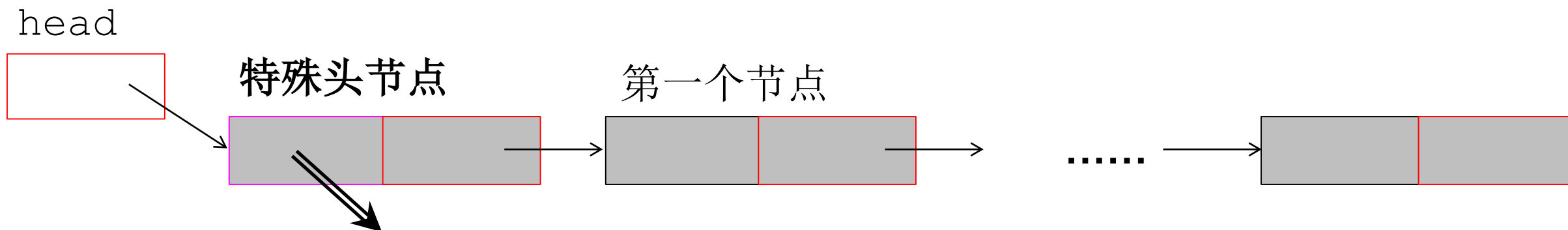
这样，第十个节点成为链表的头节点，只要知道头节点的地址（存于head中），就可以访问链表中的所有节点。

特殊头节点

无特殊头节点



有特殊头节点



不存数据，一般可以用来存链表的长度
这样，对头节点的删除或之前的插入操作 不用做特殊化处理

指针类型返回值：一般用来返回一组数据

返回链表（头节点的地址）

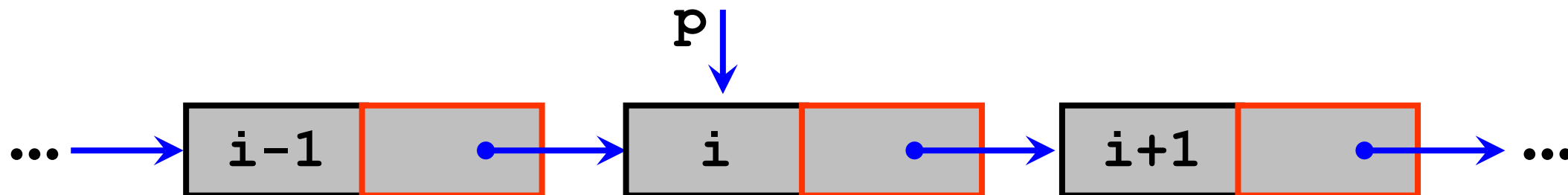
```
int main( )
{
    int n;
    scanf("%d", &n);
    Node *h = InsCreate(n);
    PrintList(h);
    .....
```

整个链表的输出（链表的遍历）

等价于 `while (p != NULL)`

① `while (p)`

③ `p = p->next;`



② `printf("%d ", p->data);`

```
void PrintList(Node *head)
{
    if( !head )        //等价于 if(head == NULL)
        printf("List is empty. \n");
    else
    {
        如果写成 while(head -> next)
        while(head)      //等价于 while(head != NULL)
        {
            printf("%d, ", head -> data);
            head = head -> next;
        }
        printf("\n");
    }
}
```

```
int main( )
{
    Node *h = InsCreate( );
    PrintList(h);
    .....
}
```

```
void PrintList(Node *head)
```

```
{
```

```
    if( !head )        //等价于 if(head == NULL)
```

```
        printf("List is empty. \n");
```

```
else
```

```
{
```

如果写成 **while(head -> next)** 则不能输出尾节点！！

```
    while(head)        //等价于 while(head != NULL)
```

```
    {        printf("%d, ", head -> data);
```

```
        head = head -> next;
```

```
    }
```

```
    printf("\n");
```

```
}
```

```
}
```

```
int main( )
```

```
{
```

```
    Node *h = InsCreate( );
```

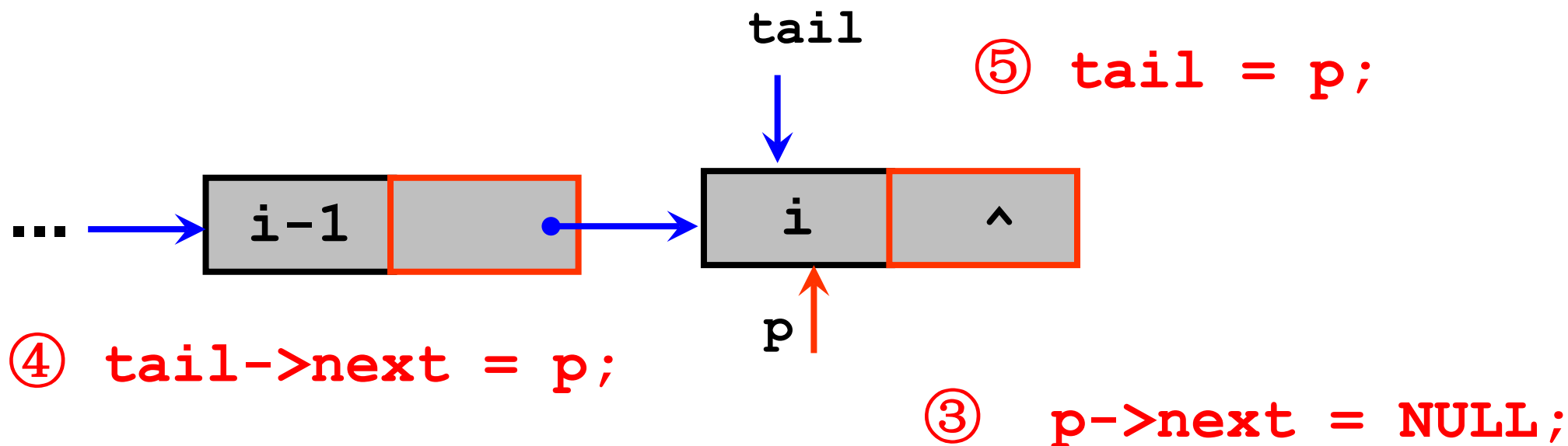
```
    PrintList(h);
```

```
    .....
```

创建链表

尾部追加节点的方式创建链表

① `for(i=0; i<n; ++i)`



② `Node *p = (Node *)malloc(sizeof(Node)); // Node *p = new Node;`
`p->data = i;`

```
Node *AppCreate(int n)
{
```

```
Node *head = NULL, *tail = NULL;
for(int i = 0; i < n; ++i)
{
```

```
//Node *p = new Node;
```

```
Node *p = (Node *)malloc(sizeof(Node)); //创建新节点
```

```
p -> data = i; //也可给新节点的数据成员输入值
```

```
p -> next = NULL; //给新节点的指针成员赋值
```

```
if(head == NULL) //链表起初没有节点时，处理创建的第一个节点
```

```
head = p;
```

```
else //处理后续创建的节点
```

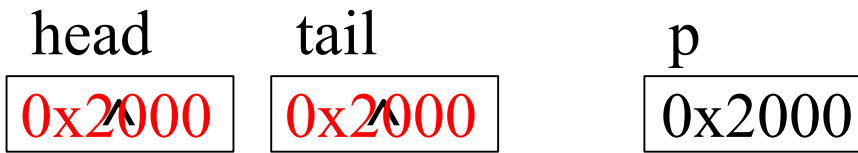
```
tail -> next = p;
```

```
tail = p;
```

```
}
```

```
return head;
```

```
}
```



```
Node *AppCreate(int n)
{
    0x2000
    0
    ^
}
```

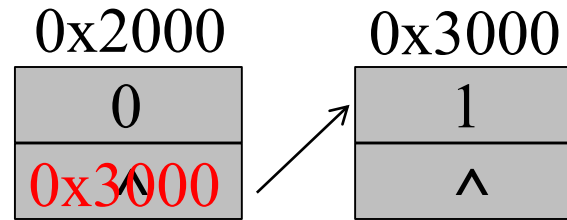
```
Node *head = NULL, *tail = NULL, *p;
for(int i = 0; i < n; ++i)
{
```

```
//Node *p = new Node;
```

```
    p = (Node *)malloc(sizeof(Node)); //创建新节点
    p -> data = i; //也可给新节点的数据成员输入值
    p -> next = NULL; //给新节点的指针成员赋值
    if(head == NULL) //链表起初没有节点时，处理创建的第一个节点
        head = p;
    else
        tail -> next = p;
    tail = p;
}
return head;
}
```




```
Node *AppCreate(int n)
{
```



```
Node *head = NULL, *tail = NULL, *p;
for(int i = 0; i < n; ++i)
{
```

```
//Node *p = new Node;
```

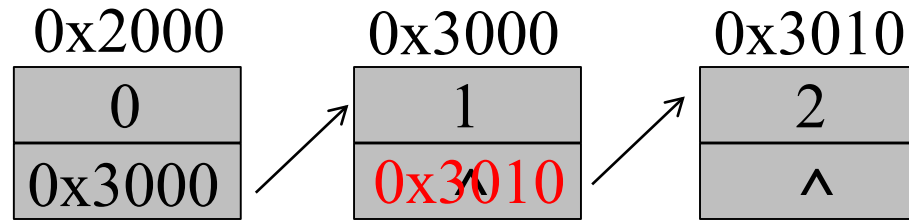
```
    p = (Node *)malloc(sizeof(Node)); //创建新节点
    p -> data = i; //也可给新节点的数据成员输入值
    p -> next = NULL; //给新节点的指针成员赋值
    if(head == NULL)
        head = p;
    else //处理后续创建的节点
        tail -> next = p;
    tail = p;
```

```
}
return head;
```

```
}
```

head	tail	p
0x2000	0x3000	0x3000

```
Node *AppCreate(int n)
{
```



```
Node *head = NULL, *tail = NULL, *p;
for(int i = 0; i < n; ++i)
{
```

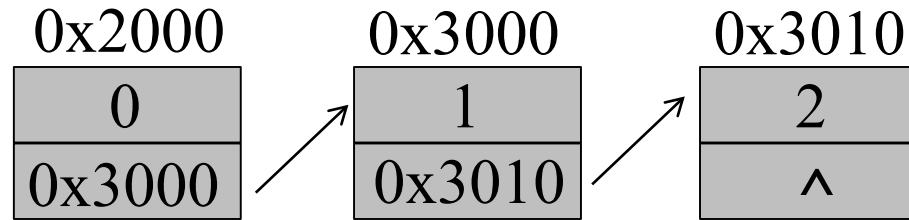
```
//Node *p = new Node;
```

```
    p = (Node *)malloc(sizeof(Node)); //创建新节点
    p -> data = i; //也可给新节点的数据成员输入值
    p -> next = NULL; //给新节点的指针成员赋值
    if(head == NULL)
        head = p;
    else //处理后续创建的节点
        tail -> next = p;
    tail = p;
```

```
}
return head;
```

head	tail	p
0x2000	0x3010	0x3010

```
Node *AppCreate(int n)
{
```



```
Node *head = NULL, *tail = NULL, *p;
for(int i = 0; i < n; ++i)
{
```

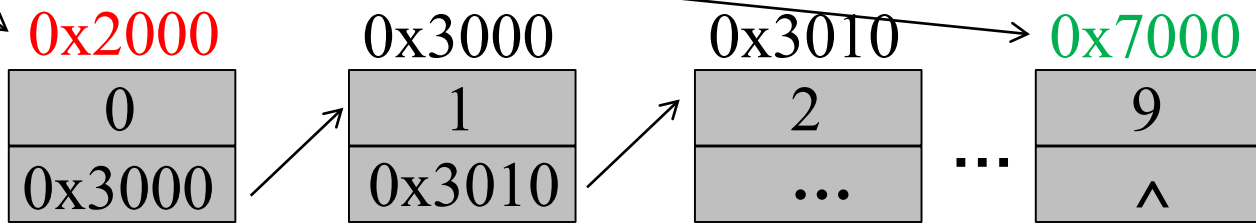
```
//Node *p = new Node;
```

```
    p = (Node *)malloc(sizeof(Node)); //创建新节点
    p -> data = i;                     //也可给新节点的数据成员输入值
    p -> next = NULL;                 //给新节点的指针成员赋值
    if(head == NULL)
        head = p;
    else
        tail -> next = p;             //处理后续创建的节点
    tail = p;
```

```
}
return head;
```

head tail p

0x2000 0x7000 0x7000



```
Node *AppCreate(int n)
{
```

```
Node *head = NULL, *tail = NULL, *p;
for(int i = 0; i < n; ++i)
{
```

//Node *p = new Node;

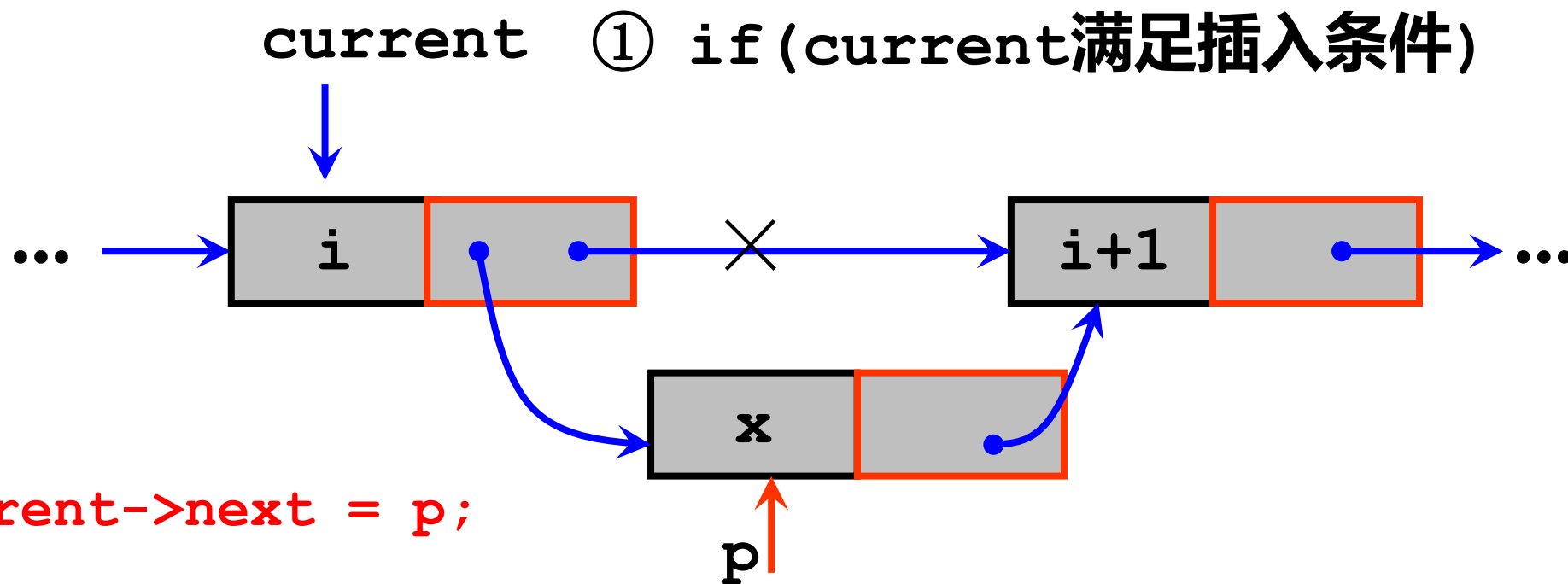
```
    p = (Node *)malloc(sizeof(Node)); //创建新节点
    p -> data = i;                     //也可给新节点的数据成员输入值
    p -> next = NULL;                 //给新节点的指针成员赋值
    if(head == NULL)
        head = p;
    else
        tail -> next = p;
    tail = p;
```

//处理后续创建的节点

```
}
return head;
}
```

链表中插入节点

- 链表中的各个节点在物理上并非存储于连续的内存空间，所以在链表中插入一个节点不会引起其它节点的移动。下面假设原链表首节点的地址存于head中，在第 i ($i > 0$) 个节点后插入一个节点。



④ `current->next = p;`

② `Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;`
`p->data = x;`

③ `p->next = current->next;`

```
void InsertNode(Node *head, int i)
{
    Node *current = head;    // current指向第一个节点
    int j = 1;
    while(j < i && current -> next != NULL)    //查找第i个节点
    {
        ++j;
        current = current -> next;
    } //循环结束时, current指向第i个节点或最后一个节点 (节点数不够i时)

    ...
}
```

```
void InsertNode(Node *head, int i)
```

```
{
```

```
...
```

```
if(j == i)    // current指向第i个节点
```

//Node *p = new Node;

```
{ Node *p = (Node *)malloc(sizeof(Node)); //创建新节点
```

```
    scanf("%d", &p -> data);
```

```
    p -> next = current ->next;
```

```
                //让第i+1个节点链接在新节点之后
```

```
    current -> next = p; //让新节点链接在第i个节点之后
```

```
}
```

```
else    //链表中没有第i个节点
```

```
    printf("没有节点: %d \n", i);
```

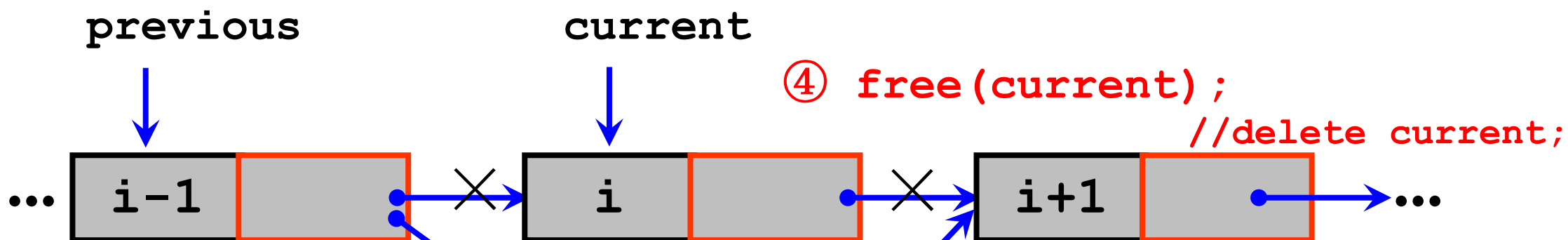
```
}
```

链表中删除节点

- 链表中的各个节点在物理上并非存储于连续的内存空间，所以在链表中删除一个节点不会引起其它节点的移动。下面假设原链表首节点的地址存于head中，删除第 i ($i > 0$) 个节点。

① `if (previous->next` 满足删除条件)

② `current = previous->next;`



③ `previous->next = current->next;`

```
Node *DeleteNode(Node *head, int i)
{
    if(i == 1)    //删除头节点
    {
        Node *current = head;    // current指向头节点
        head = head->next;    // head指向新的头节点
        free(current); // delete current;释放删除节点的空间
    }
    else
    {
        .....
    }
    return head;
}
```

```
else
```

```
{ Node *previous = head;    // previous指向头节点
```

```
    int j = 1;
```

```
    while(j < i-1 && previous -> next != NULL)
```

```
    {    ++j;
```

```
        previous = previous -> next;
```

```
    }    // 查找第i-1个节点
```

```
    if(previous -> next != NULL)    // 链表中存在第i个节点
```

```
    {    Node *current = previous -> next;
```

```
        // current指向第i个节点
```

```
        previous -> next = current -> next;
```

```
        // 让待删除节点的前后两个节点相链接
```

```
        free(current); // delete current; 释放第i个节点的空间
```

```
    }
```

```
else // 链表中没有第i个节点
```

```
    printf("没有节点: %d \n", i); }
```

整个链表的删除

- 链表中的每个节点都是动态变量，所以在链表处理完后，最好用程序释放整个链表所占空间，即删除链表。
- 假设原链表首节点的地址存于head中，则删除整个链表的程序为：

```
void DeleteList(Node *head)
{
    while(head)
        //遍历链表，如果写成while(head -> next)则不能删除尾节点！！
        {
            Node *current = head;
            head = head -> next;
            free(current); // delete current;
        }
}
```

- 上述程序中的形参head与实参（即使变量名也是head）是不同的指针变量，形参的值有可能发生改变，所以要通过return语句返回给调用者。如果利用函数的副作用返回其值，则形参需定义成二级指针！！

参数为指针的传值调用

```
int main()
```

```
{
```

```
Node *h = (Node *)malloc(sizeof(Node)); //Node *h = new Node;
```

```
h->data = 1;
```

```
h->next = NULL;
```

```
InsOneNode(h);
```

```
...
```

```
return 0;
```

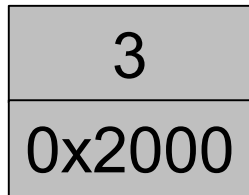
```
}
```

head 0x3000

h 0x2000

0x3000

0x00002000



已有链表头部插入1个结点?

void

```
Node *InsOneNode(Node *head)
```

```
h = InsOneNode(h);
```

```
Node *p = new Node;
```

```
p -> data = 3;
```

```
p -> next = head; //并未取值
```

```
head = p; //并未取值
```

```
return head;
```

```
}
```

改为传址调用

```
int main()
```

```
{
```

```
Node *h = (Node *)malloc(sizeof(Node)); //Node *h = new Node;
```

```
h->data = 1;
```

```
h->next = NULL;
```

```
InsOneNode(&h);
```

```
...
```

```
return 0;
```

```
}
```

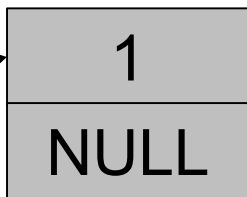
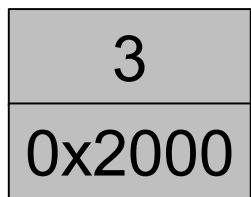
0x7000

h 0x2000

head 0x2000

0x3000

0x2000



已有链表头部插入1个结点?

void

```
Node *InsOneNode(Node **head)
```

```
{
```

```
Node *p = new Node;
```

```
p -> data = 3;
```

```
p -> next = *head;
```

```
*head = p;
```

```
return head;
```

```
}
```

改为引用 (C++)

```
int main()
{
    Node *h = new Node;
    h-> data = 1;
    h-> next = NULL;
    InsOneNode(h) ;
    ...

    return 0;
}
```

已有链表头部插入1个结点?

```
void
Node *InsOneNode(Node *&head)
{
    Node *p = new Node;
    p -> data = 3;
    p -> next = head;
    head = p;

    return head;
}
```

```

Node *InsertBeforeKeyNode(Node *h, int key)
{
    Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;
    scanf("%d", &p -> data);
    if(h != NULL)
    {
        Node *current = h;
        Node *previous = NULL;
        while(current != NULL && current -> data != key )
        {
            previous = current;
            current = current -> next;
        }
        if(current != NULL && previous != NULL)
        {
            p -> next = current;
            previous -> next = p;
        }
        else if(current != NULL && previous == NULL)
        {
            p -> next = current;
            h = p;
        } //头部插入
    }
    return h;
}

```

又比如，在某节点前插入新节点


```

void InsertBeforeKeyNode(Node **h, int key)
{
    Node *p = (Node *)malloc(sizeof(Node)); //Node *p = new Node;
    scanf("%d", &p -> data);
    if(*h != NULL)
    {
        Node *current = *h;
        Node *previous = NULL;
        while(current != NULL && current -> data != key )
        {
            previous = current;
            current = current -> next;
        }
        if(current != NULL && previous != NULL)
        {
            p -> next = current;
            previous -> next = p;
        }
        else if(current != NULL && previous == NULL)
        {
            p -> next = current;
            *h = p;
        } //头部插入
    }
}

```

又比如，在某节点前插入新节点

顺序！
短路规则

```
void InsertNode(Node *head, int i)
{
    Node *current = head;    // current指向第一个节点
    int j = 1; while(current -> next != NULL && j < i)
while(j < i && current -> next != NULL)    //查找第i个节点
{    ++j;
    current = current -> next;
} //循环结束时, current指向第i个节点或最后一个节点 (节点数不够i时)

...

}
```

再比如

```
Node *DeleteNode(Node *head) //删除头节点
{
    Node *current = head;    // current指向头节点
    head = head->next;        // head指向新的头节点
    free(current);            // delete current; 释放删除节点的空间
    return head;
}
```

```
void DeleteNode(Node **head) //删除头节点
{
    Node *current = *head;    // current指向头节点
    *head = *head->next;      // *head指向新的头节点
    free(current);            // delete current; 释放删除节点的空间
    //return head;
}
```

基于链表的排序

- 基于链表的排序一般会涉及两个节点数据成员的比较和交换操作，以及节点的插入等操作。

例9.1 用链表实现N个数的插入法排序。

...

```
const int N = 10;
```

```
typedef struct Node Node;
```

```
struct Node
```

```
{ int data;
```

```
    Node *next;
```

```
} extern Node *AppCreate( );
```

```
extern Node *SortList(Node *head);
```

```
extern void PrintList(Node *head);
```

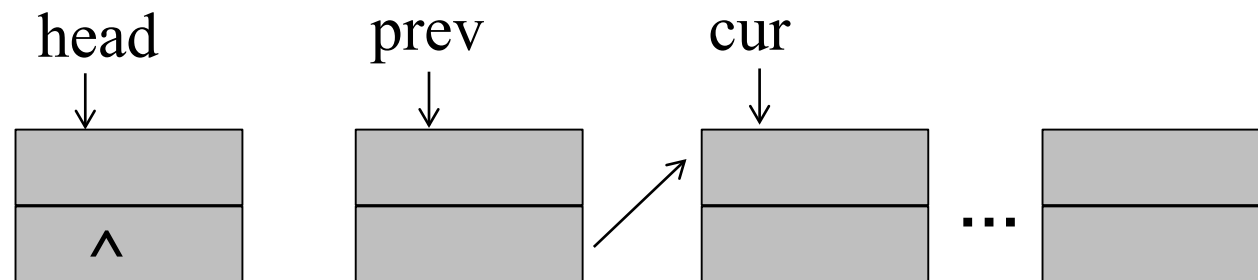
```
extern void DeleteList(Node *head);
```

```
int main( )
{
    Node *head = AppCreate( );           //建立链表，程序略
    PrintList(head);                     //输出链表，程序略
    head = SortList(head);
    PrintList(head);                     //输出排序之后的链表
    DeleteList(head);                    //删除链表，程序略
    return 0;
}
```

Node *InsertIn(Node *head, Node *prev); // 插入一个节点

Node *SortList(Node *head) //插入法排序函数

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



```
Node *cur = head -> next;
```

```
head -> next = NULL;
```

//将头节点脱离下来，作为已排序队列
//将后面的节点依次插入已排序队列

```
while(cur)
```

```
{
```

```
    Node *prev = cur;
```

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

```
    head = InsertIn(head, prev) ;
```

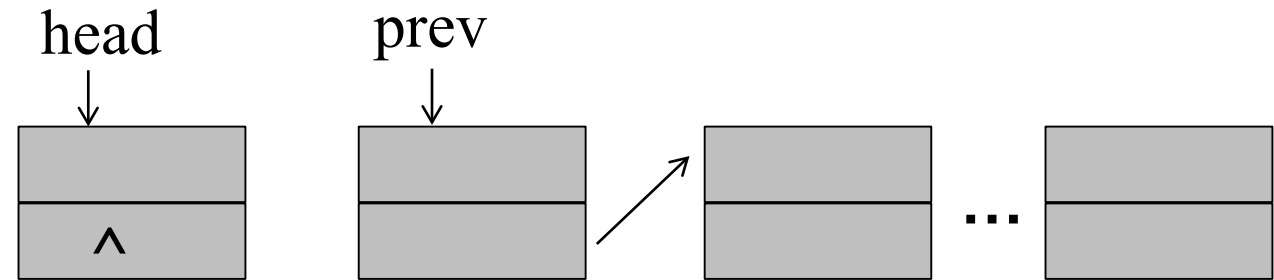
```
}
```

```
return head;
```

```
}
```

Node *InsertIn(Node *head, Node *prev) //插入一个节点

```
{  
    if (prev -> data < head -> data)  
    {  
        prev -> next = head;  
        head = prev;  
        return head;  
    } //插入头部
```



Node *InsertIn(Node *head, Node *prev) //插入一个节点

{

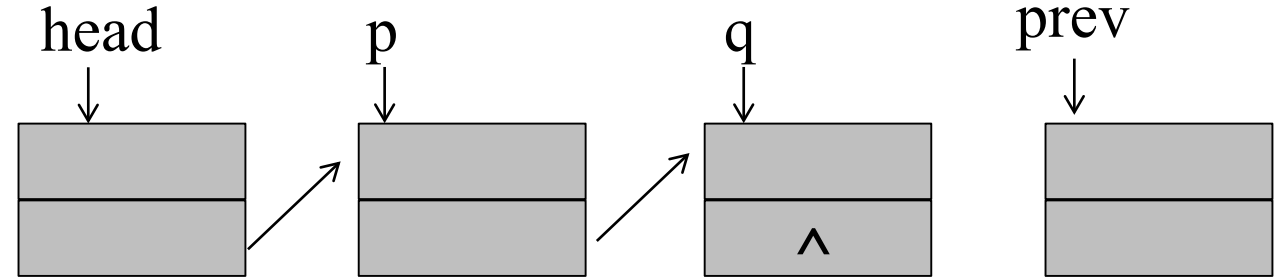
if (prev -> data < head -> data)

{ prev -> next = head;

head = prev;

return head;

} //插入头部



Node *q = head;

Node *p = head; //用q和p操纵已排序队

while (q)

{ if (prev -> data < q -> data)

break;

p = q;

q = q -> next;

} //查找合适的位置，在p后插入

prev -> next = p -> next;

p -> next = prev;

return head;

}

基于链表的信息检索

❁ 一般不适合用折半查找法。

例9.2 基于链表的顺序查找程序。

...

```
typedef struct NodeStu NodeStu;
```

```
struct NodeStu
```

```
{ int id;           //学号
```

```
  float score; //成绩
```

```
  NodeStu *next;
```

```
};
```

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

```
extern float ListSearch(NodeStu *head);
```

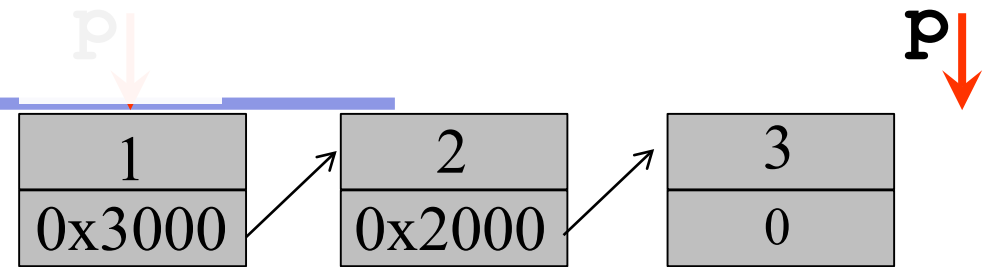
```
extern void DeleteList(NodeStu *head);
```

```
int main( )
{ NodeStu *head = AppCreateStu( ); //建立链表
  int x = 201220999;
  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! \n");
  else
      printf("%s同学的成绩为: %f \n", x, y);

  return 0;
}
```

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

链表操作的注意事项



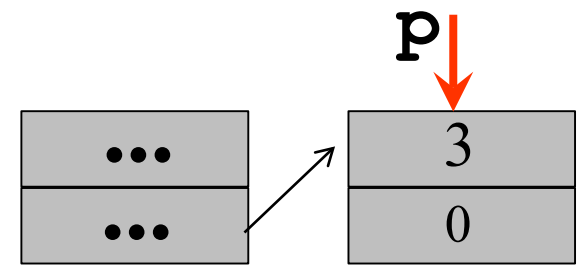
最后一个节点的 next 成员一般要置为 NULL

```
while (p)
或
while (p != NULL)
```

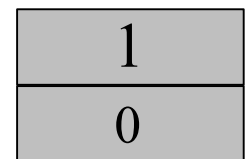
访问链表的指针变量已经指向最后一个节点的 next 成员 p -> ?

空链表 (head == NULL) 的处理

对最后一个节点 (p -> next == NULL) 的访问



对只有一个节点的链表 (head -> next == NULL) 的访问



小结

- 基于结构类型和指针类型的数据结构——链表
- 基于链表的排序和检索算法的程序实现方法
- 要求：
 - ◆ 掌握链表的特征及其创建、删除、插入节点、删除节点等方法
 - 一个程序代码量 \approx 200行
 - ◆ 继续保持良好的编程习惯
 - 删除动态空间...

Thanks!

