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）

抽象与联系（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-1、字符串、**结构、指针-2、链表**）

归纳与推广（程序设计的本质）

链表的构造与操作

重难点

- 链表的建立
- 整个链表的输出与删除
- 链表中节点的插入与删除
- 基于链表的排序与检索程序

1）对输入的10个整数进行排序，可以用数组来实现；

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），可以用数组来实现（新C标准）；

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），可以用动态数组来实现（C++标准、老C标准）；

3）对输入的若干个正整数进行排序（先输入各个正整数，最后输入一个结束标志 -1），可以用不断调整动态数组的大小来实现；

问题的提出

3）对输入的若干个正整数进行排序（先输入各个正整数，最后输入一个结束标志 -1），可以用不断调整动态数组的大小来实现：

可能需要大量数据搬迁

可能找不到足够的连续空间（(int *)malloc(sizeof(int)*max_len)）

```
//new int[max_len]
```

4-1）输入一个数，插入到已经排好序的若干个数当中，保持原来的大小顺序

4-2）删除一个数，保持原来的大小顺序

？

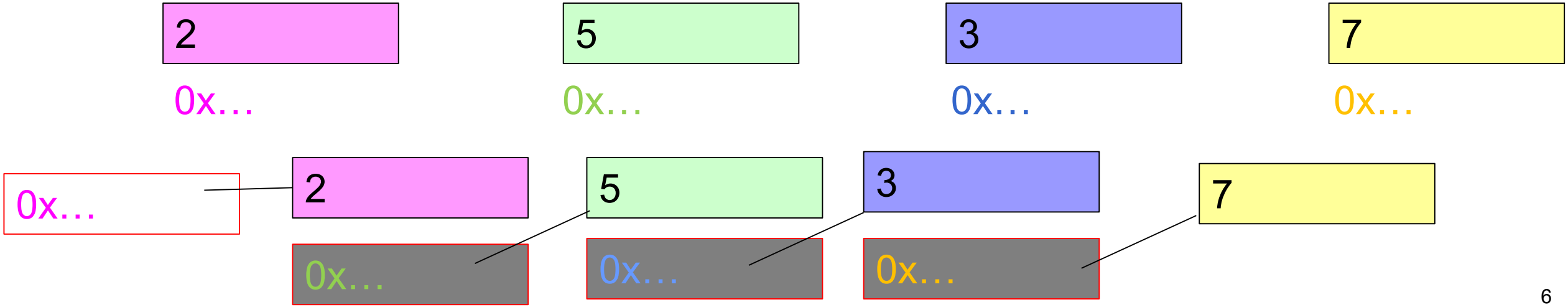
？

- **数组的缺陷：**
  - 数组的长度一般在定义前就已确定，所占内存空间在其生存期始终保持不变，即使可变，由于数组元素的有序性，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，还可能会受数组所占内存空间大小的限制。

- **链表**
  - 实际应用中，常常需要表示一种在程序运行期间元素个数可以随机增加或减少、所占内存空间大小可动态变化、数据元素在逻辑上连续排列而物理上并不占用连续存储空间的数据群体，C语言用链表（list）来表示这种数据群体。

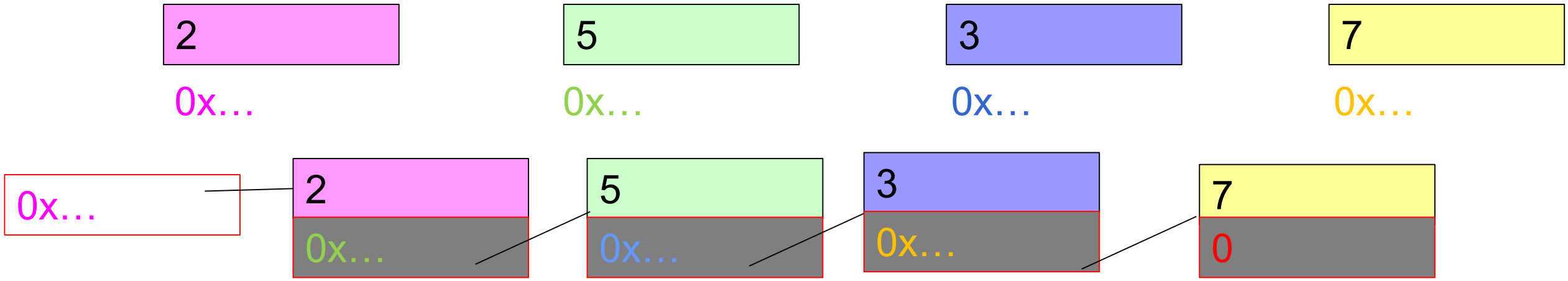
链表

- 内存
  - 栈区：存放程序中定义的 基本类型变量、数组、指针变量、指针数组、结构体、形式参数…
  - 堆区（零星的空间）：存放程序中创建的单个动态变量、动态数组（多个关联的动态变量）
- 堆区没有足够的连续空间存储动态数组时怎么办?
  - 用指针把若干个分散的动态变量链接起来
  - 动态的（变量 + 指针） → 动态的（结构体）



链表

- 内存
  - 栈区：存放程序中定义的 基本类型变量、数组、指针变量、指针数组、结构体、形式参数…
  - 堆区（零星的空间）：存放程序中创建的单个动态变量、动态数组（多个关联的动态变量）
- 堆区没有足够的连续空间存储动态数组时怎么办?
  - 用指针把若干个分散的动态变量链接起来
  - 动态的（变量 + 指针） → 动态的（结构体）



链表的建立

- 链表不是一种数据类型，而是一种通过程序将若干个同类型的数据（节点）链接起来的数据结构。链表中的每个节点是一个结构类型的动态变量，这种结构类型的若干个成员中至少有一个指针类型的成员，其构造形式为：

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

单向链表

- 注意，结构类型的成员不能是本结构类型的变量，但可以是本结构类型的指针变量。正是通过指针变量得以将若干个节点链接起来，从而完成链表的建立。可见，链表中的各节点不必存放在连续的内存空间中。

产生一个新节点

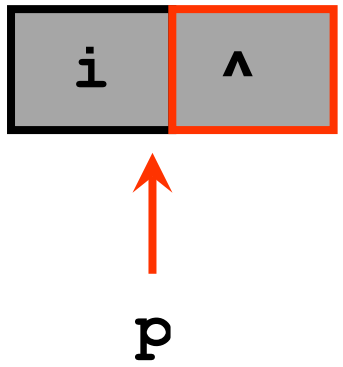
```
struct Node
{
        int data;
        Node *next;
};
```

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

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

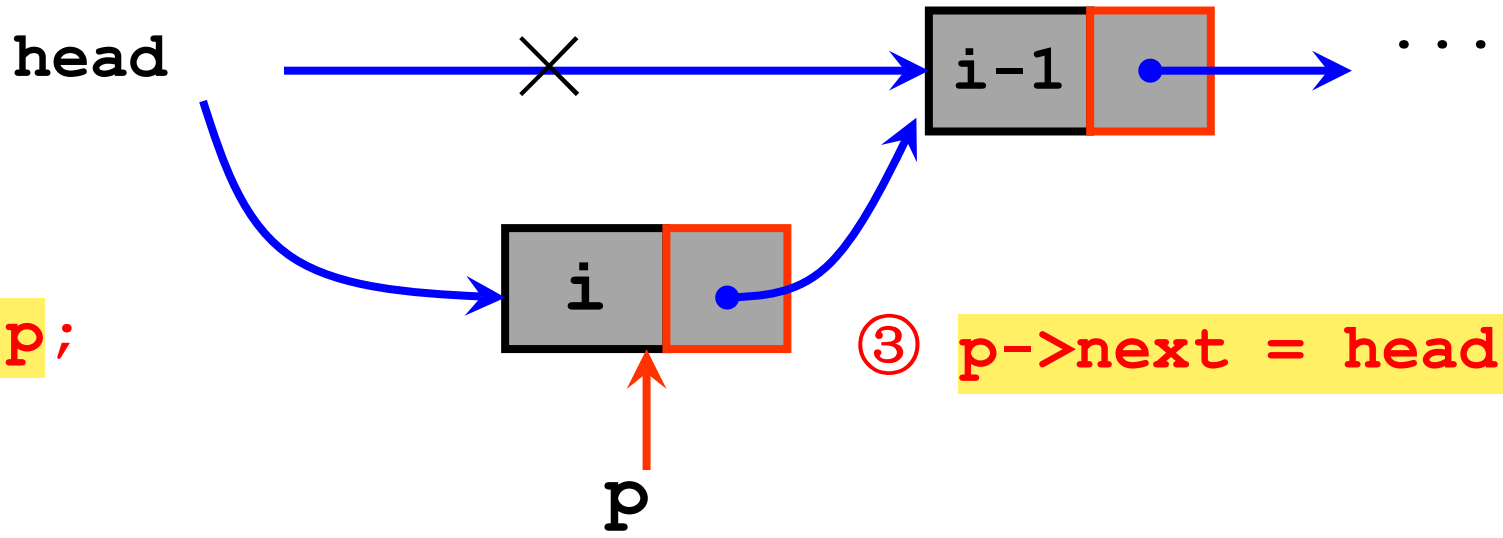
```
scanf("%d", &p->data); // p->data = i;
```

```
p->next = 0; // p->next = NULL;
```



（1）头部插入节点方式建立链表

① `for(i=0; i<n; ++i)`



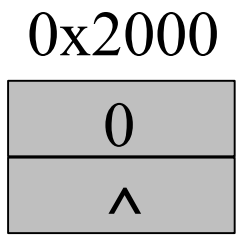
④ `head = p;`

③ `p->next = head;`

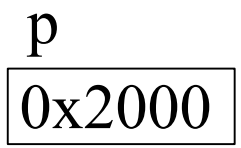
② `p = (Node *)malloc(sizeof(Node)); //new Node;`
`p->data = i;`

指针变量　赋值　地址

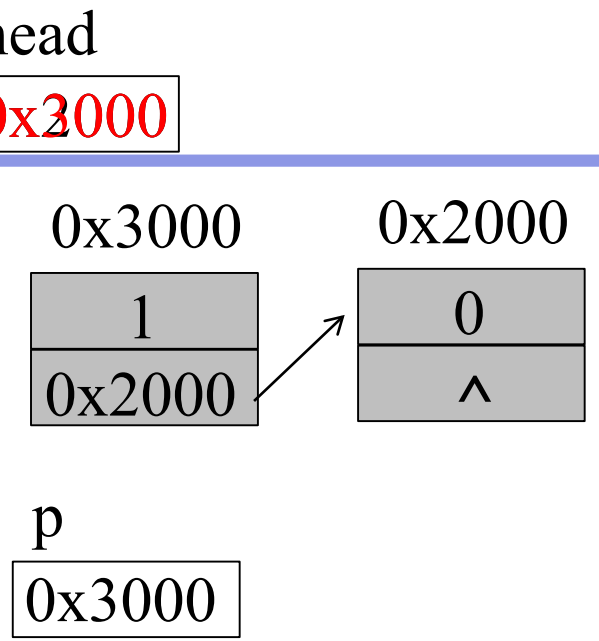

```
struct Node
{
        int data;
        Node *next;
};
```




```
#define N 10
Node *InsCreate( )
{
  Node *head = NULL;
```

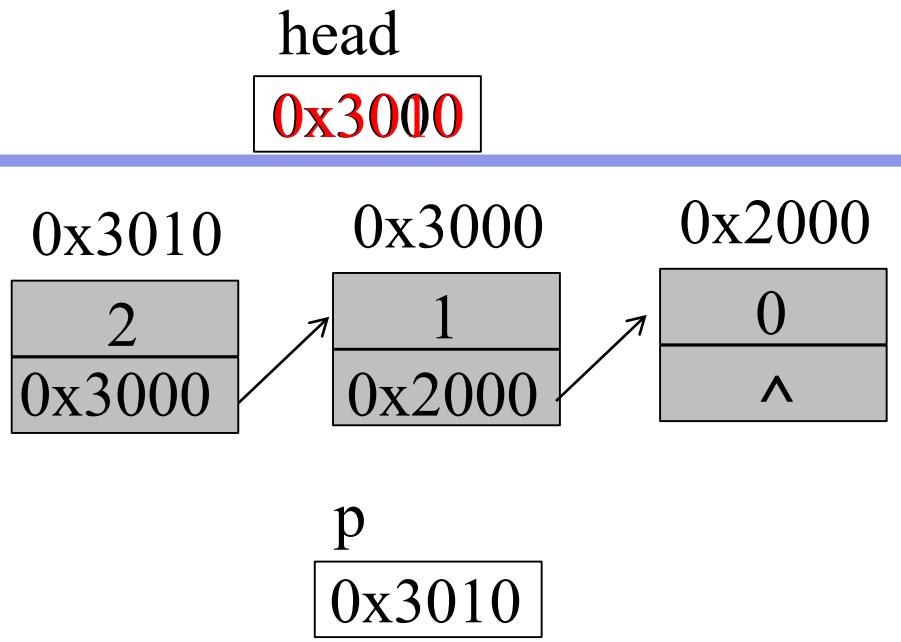



```
  for(int i = 0; i < N; ++i)
  {
     Node *p = (Node *)malloc(sizeof(Node)); //new Node;
     p -> data = i;          //也可给新节点的数据成员输入值
     p -> next = head;       //head的值赋给新节点的指针成员
     head = p;               //将p这个指针变量的值赋给指针变量head
  }
  return head;
}
```



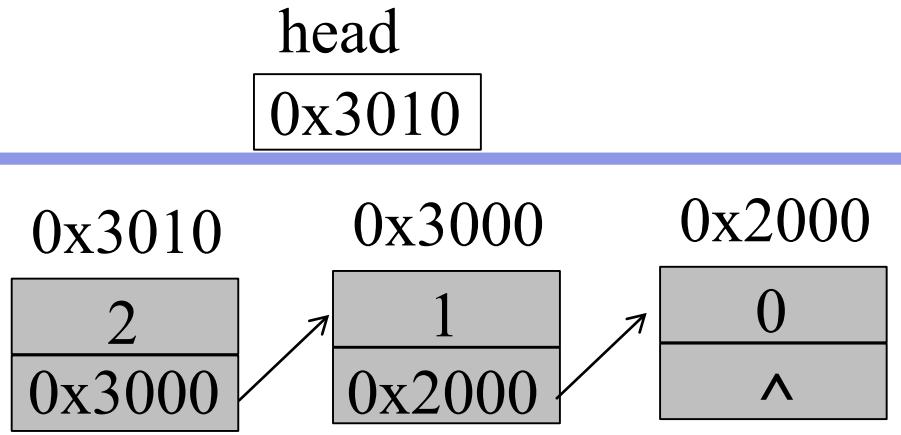

```
struct Node
{
        int data;
        Node *next;
};
```

```
#define N 10
Node *InsCreate( )
{
  Node *head = NULL;

  for(int i = 0; i < N; ++i)
  {
     Node *p = (Node *)malloc(sizeof(Node)); //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;
}
```




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

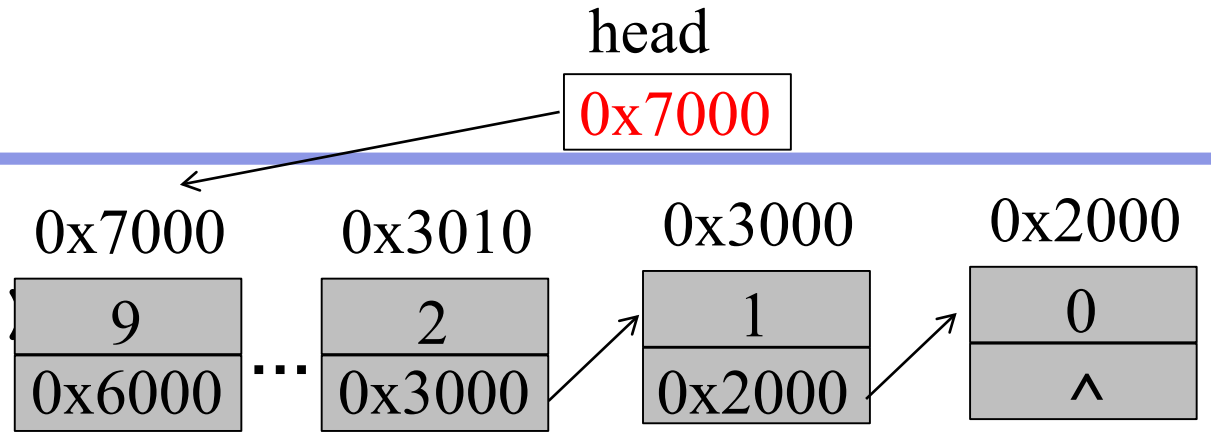
```
#define N 10
Node *InsCreate( )
{
  Node *head = NULL;

  for(int i = 0; i < N; ++i)
  {
     Node *p = (Node *)malloc(sizeof(Node)); //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;
}
```




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

```
#define N 10
Node *InsCreate( )
{
  Node *head = NULL;

  for(int i = 0; i < N; ++i)
  {
     Node *p = (Node *)malloc(sizeof(Node)); //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




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

```
#define N 10
Node *InsCreate( )
{
  Node *head = NULL;

  for(int i = 0; i < N; ++i)
  {
     Node *p = (Node *)malloc(sizeof(Node)); //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;
}
```

```
#define N 10
Node *InsCreate( )
{
  Node *head = NULL;

  for(int i = 0; i < N; ++i)
  {
     Node *p = (Node *)malloc(sizeof(Node)); //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;
}
```

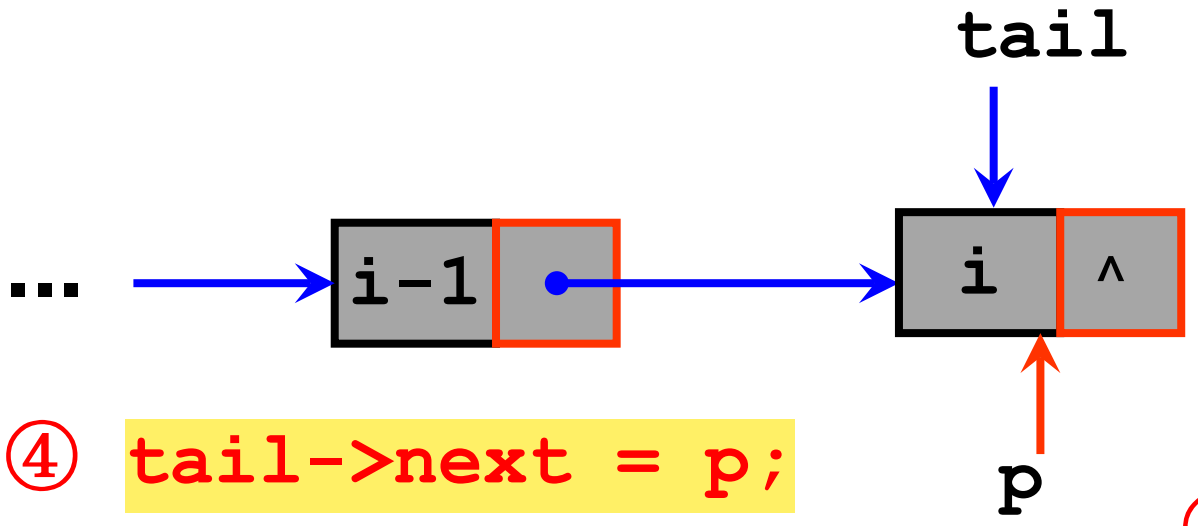
这样，第十个节点成为链表的头节点，只要知道头节点的地址（存于head中），就可以访问链表中的所有节点。

指针类型返回值：一般用来返回一组数据 链表

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

(2) 尾部追加节点方式建立链表

① `for(i=0; i<n; ++i)`



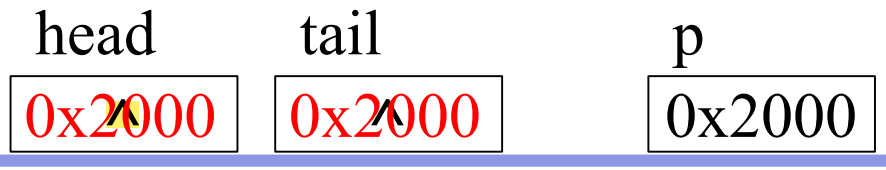
⑤ `tail = p;`

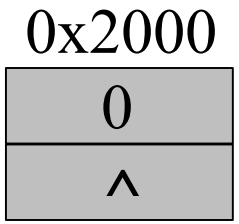
④ `tail->next = p;`

③ `p->next = NULL;`

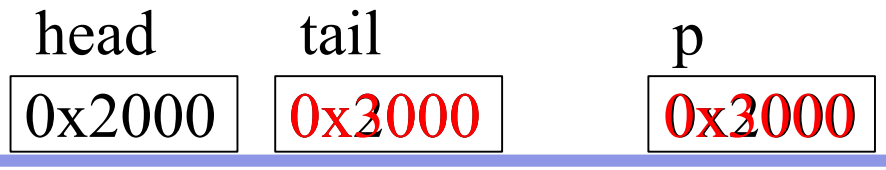
② `Node *p = (Node *)malloc(sizeof(Node));` //new Node;
`p->data = i;`

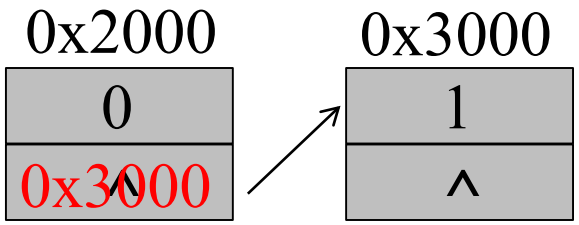
```
#define N 10
Node *AppCreate( )
{ Node *head = NULL, *tail = NULL;
  for(int i = 0; i < N; ++i)
  {   Node *p = (Node *)malloc(sizeof(Node)); //new Node;创建新节点
      p -> data = i;          //也可给新节点的数据成员输入值
      p -> next = NULL;       //给新节点的指针成员赋值
      if(head == NULL)        //链表起初没有节点时，处理创建的第一个节点
            head = p;
      else                    //处理后续创建的节点
            tail -> next = p;
      tail = p;
  }   return head; }
```



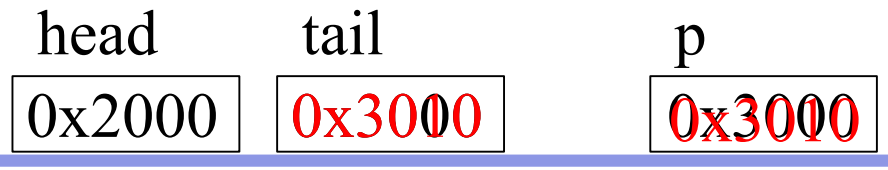


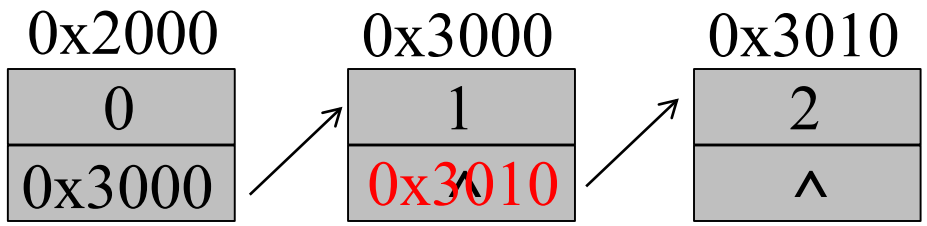

```
#define N 10
Node *AppCreate( )
{ Node *head = NULL, *tail = NULL, *p;
  for(int i = 0; i < N; ++i)
  {  p = (Node *)malloc(sizeof(Node)); //new Node;创建新节点
     p -> data = i;          //也可给新节点的数据成员输入值
     p -> next = NULL;       //给新节点的指针成员赋值
     if(head == NULL)        //链表起初没有节点时，处理创建的第一个节点
           head = p;
     else                    //处理后续创建的节点
           tail -> next = p;
     tail = p;
  }  return head; }
```



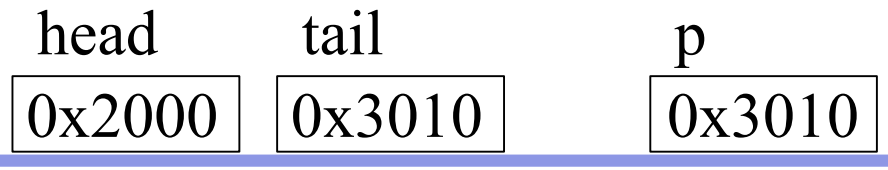


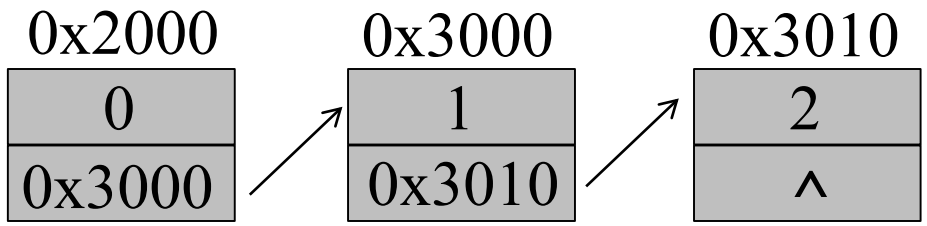

```
#define N 10
Node *AppCreate( )
{ Node *head = NULL, *tail = NULL, *p;
  for(int i = 0; i < N; ++i)
  {  p = (Node *)malloc(sizeof(Node)); //new Node;创建新节点
     p -> data = i;          //也可给新节点的数据成员输入值
     p -> next = NULL;       //给新节点的指针成员赋值
     if(head == NULL)        //链表起初没有节点时，处理创建的第一个节点
          head = p;
     else                    //处理后续创建的节点
          tail -> next = p;
     tail = p;
  }  return head; }
```



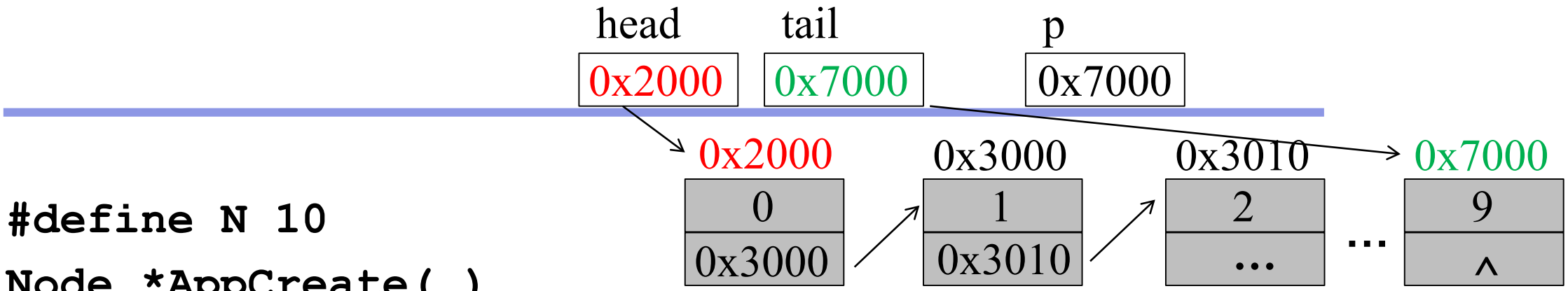



```
#define N 10
Node *AppCreate( )
{ Node *head = NULL, *tail = NULL, *p;
  for(int i = 0; i < N; ++i)
  {  p = (Node *)malloc(sizeof(Node));  //new Node;创建新节点
     p -> data = i;          //也可给新节点的数据成员输入值
     p -> next = NULL;       //给新节点的指针成员赋值
     if(head == NULL)        //链表起初没有节点时，处理创建的第一个节点
          head = p;
     else                    //处理后续创建的节点
          tail -> next = p;
     tail = p;
  }  return head; }
```






```
#define N 10
Node *AppCreate( )
{ Node *head = NULL, *tail = NULL, *p;
  for(int i = 0; i < N; ++i)
  {   p = (Node *)malloc(sizeof(Node));   //new 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; }
```



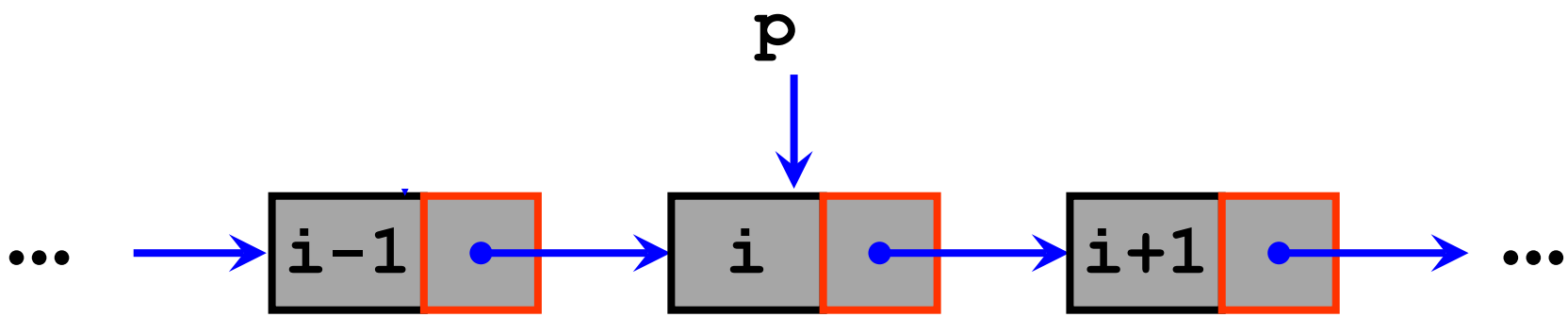

```
#define N 10
Node *AppCreate( )
{ Node *head = NULL, *tail = NULL, *p;
  for(int i = 0; i < N; ++i)
  {  p = (Node *)malloc(sizeof(Node));  //new Node;创建新节点
     p -> data = i;          //也可给新节点的数据成员输入值
     p -> next = NULL;       //给新节点的指针成员赋值
     if(head == NULL)        //链表起初没有节点时，处理创建的第一个节点
          head = p;
     else                    //处理后续创建的节点
          tail -> next = p;
     tail = p;
  }  return head; }
```

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);`

- 假设原链表首节点的地址存于head中，则输出整个链表的实现方式为：

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

如果写成 while(head -> next)

- 假设原链表首节点的地址存于head中，则输出整个链表的实现方式为：

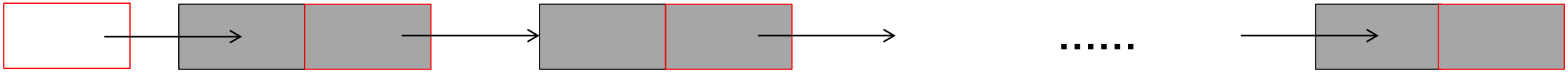
```
void PrintList(Node *head)
{ if(!head)     //如果是空链表
     printf("List is empty. \n");
  else
  {
     while(head)     //遍历链表，等价于while(head != NULL)
     {    printf("%d, ", head -> data);
          head = head -> next;
     }
     printf("\n");
  }
}
```

如果写成 while(head -> next) 则不能输出尾节点！！

- `int a[10];`

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

- 链表




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

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

```
struct Node
{     char c;
      Node *next;
};
```

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

```
struct Node *p = (struct Node *)malloc(sizeof(struct Node));
```

- **解决办法**
  - 用 typedef

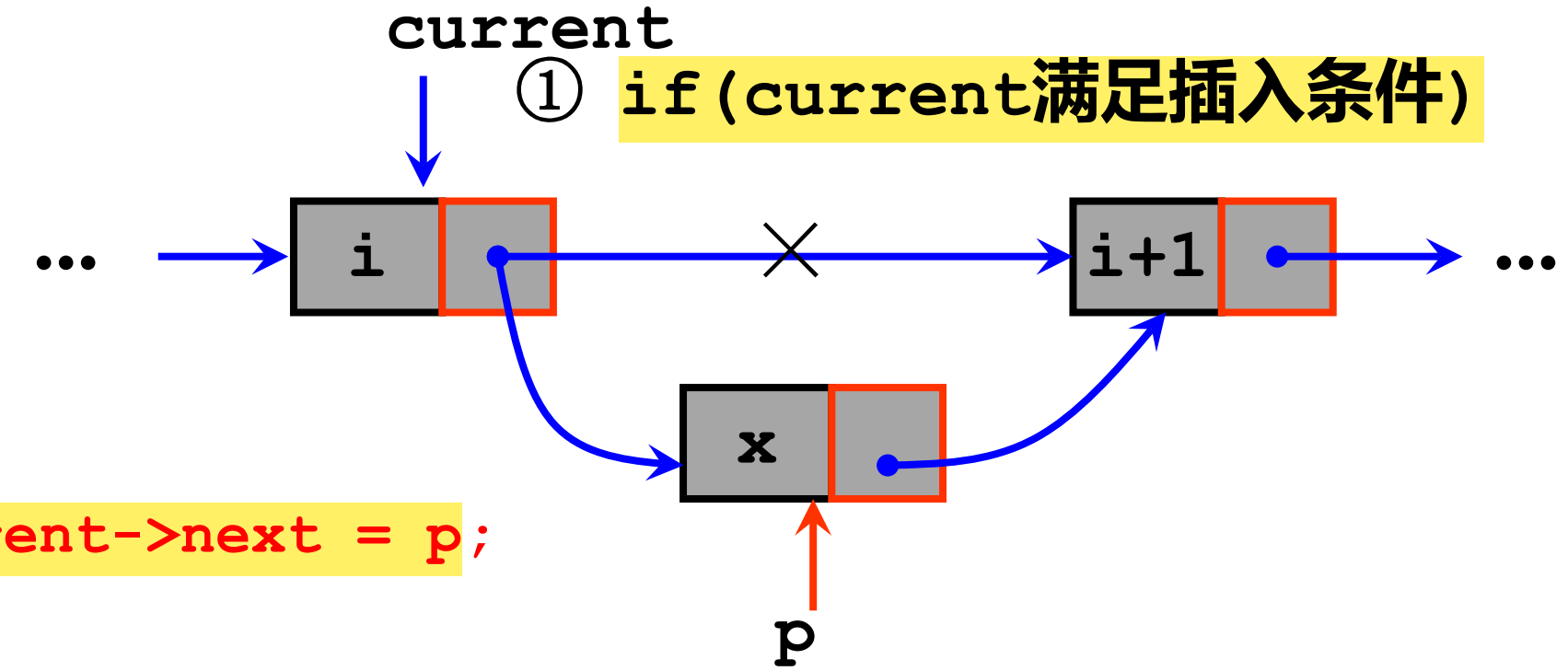
```
struct Node
{     char c;
      Node *next;
};
typedef struct Node Node;
```

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

  - 或 存储文件名 为 .cpp

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，所以在链表中插入或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，在第i（i>0）个节点后插入一个节点。



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

③ p->next = current->next;

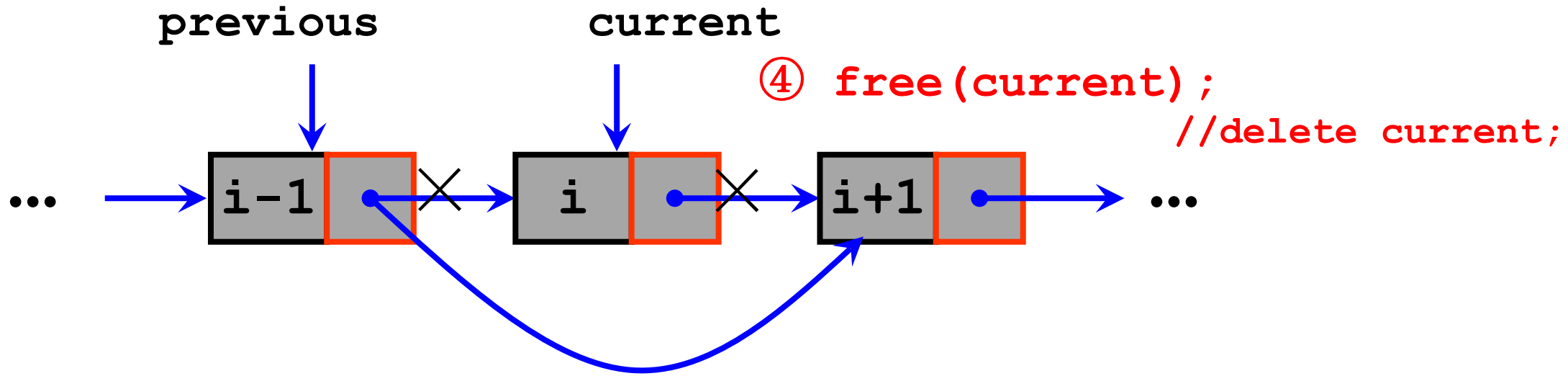
```
void InsertNode(Node *head, int i)
{
  Node *current = head;   // current指向第一个节点
  int j = 1;
  while(j < i && current -> next != NULL)      //查找第i个节点
  {   current = current -> next;
      ++j;
  } //循环结束时，current指向第i个节点或最后一个节点（节点数不够i时）

  …

}
```

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

  if(j == i)    // current指向第i个节点
  {  Node *p = (Node *)malloc(sizeof(Node)); //new 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);
}
```

删除第i（i>0）个节点

① if(previous->next满足删除条件)

② current = previous->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)
    {    previous = previous -> next;
         ++j;
    }    //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;
  }
}
```

难点2

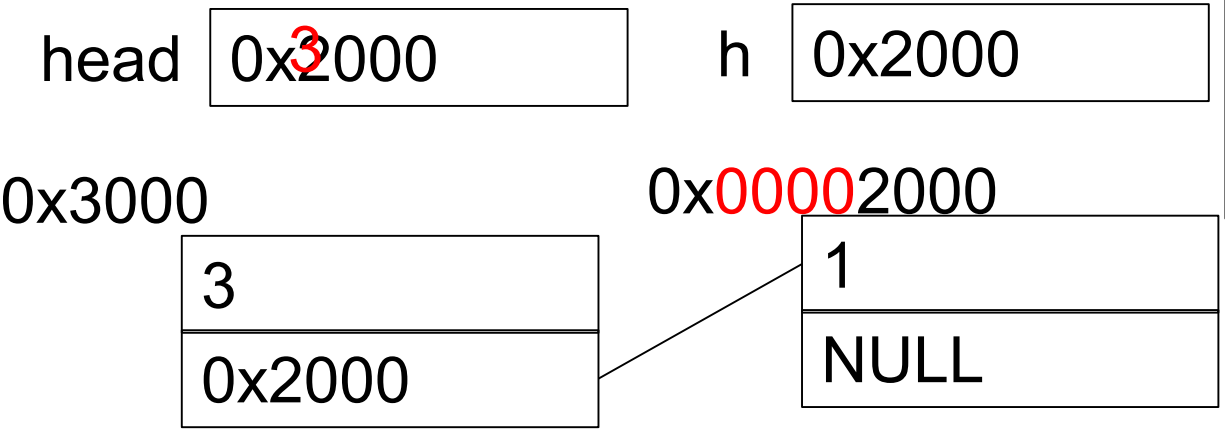
- 上述程序中的形参head与实参（即使变量名也是head）是不同的指针变量，形参的值有可能发生改变，所以要通过return语句返回给调用者。如果利用函数的副作用返回其值，则形参需定义成二级指针！！

参数为指针的传值调用

```
int main()
{
  Node *h = (Node *)malloc(sizeof(Node)); //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;//并未取值
      head = p;//并未取值

      return head;
}
```

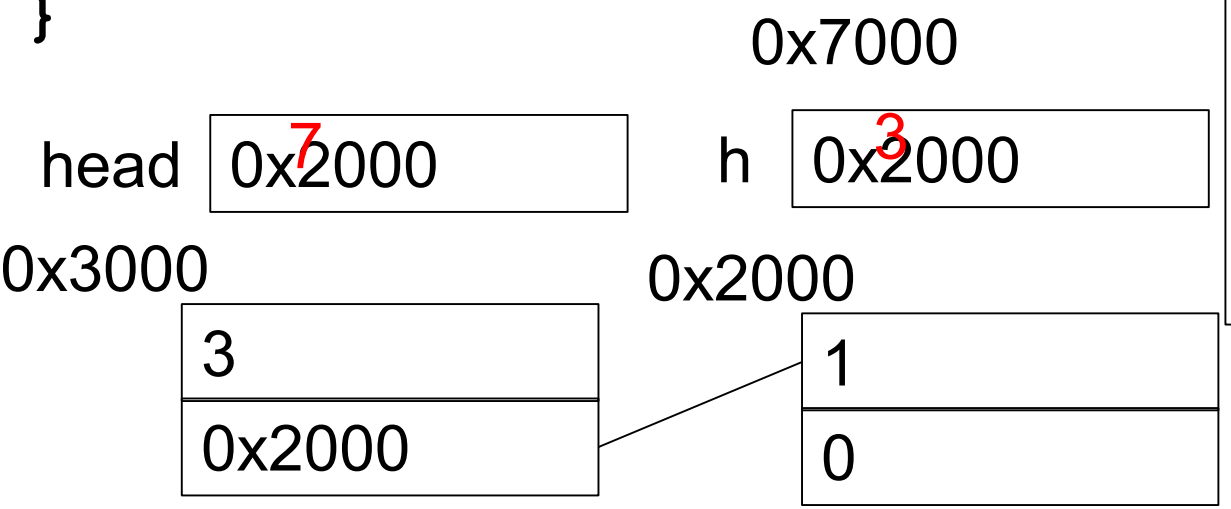


改为传址调用

```
int main()
{
  Node *h = (Node *)malloc(sizeof(Node)); //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;
}
```



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)); //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

顺序！
短路规则

```
void InsertBeforeKeyNode(Node **h, int key)
{    Node *p = (Node *)malloc(sizeof(Node)); //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;
          }
     }
```

又比如，在某节点前插入新节点

再比如

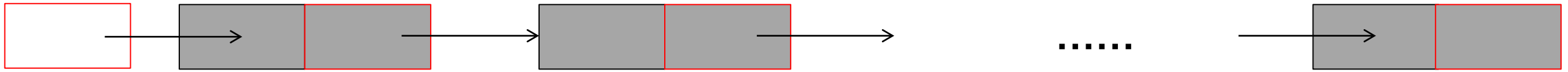
```
Node *DeleteNode(Node *head) //删除头节点
{     Node *current = head;     // current指向头节点
      head = head->next;  // head指向新的头节点
      free(current); // delete current;释放删除节点的空间
      return head;
}

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;
}
```

链表操作中需要注意的几个问题

- 注意考虑几个特殊情况下的操作
  - 链表为空表 (head==NULL)
  - 链表只有一个节点
  - 对链表的第一个节点进行操作
  - 对链表的最后一个节点进行操作
  - 最后一个节点的next指针应为NULL
  - 操控链表的指针是否已经指向了链表末尾

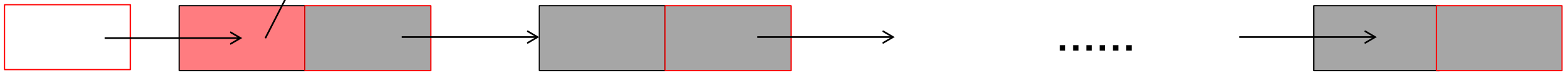
- **头指针必须有**

- **无头节点，只有头指针**



- **有头节点**

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



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 用链表实现N个数的插入法排序。

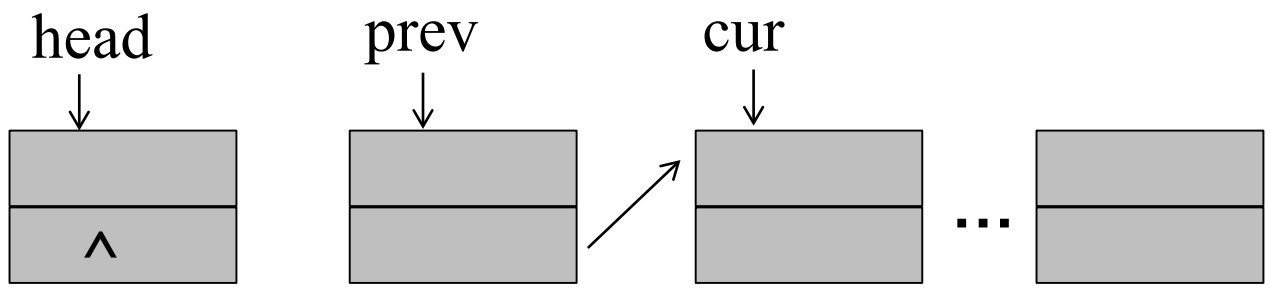
…

```
#define N 10
struct Node
{ int data;
  Node *next;
};
```

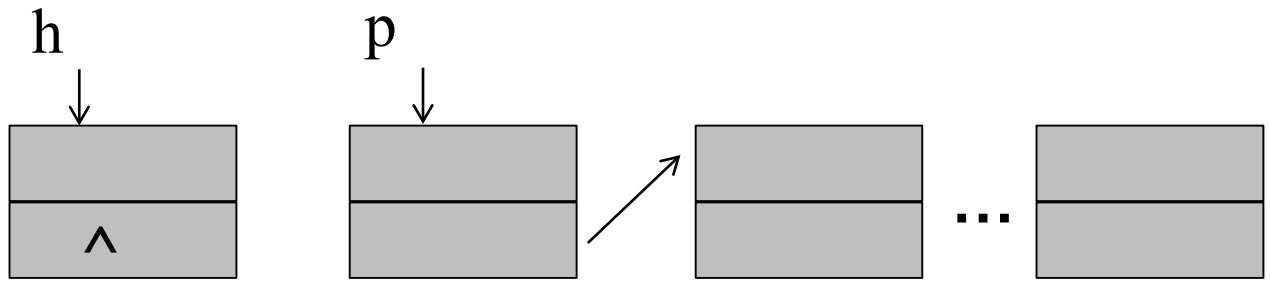
```
extern Node *AppCreate( );
extern Node *ListSort(Node *head);
extern void PrintList(Node *head);
extern void DeleteList(Node *head);
```

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

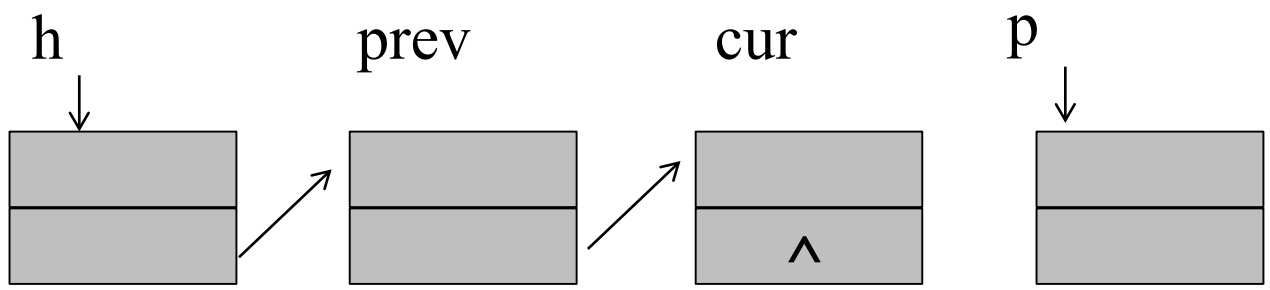
```
Node *ListSortInsert(Node *h, Node *p); // 插入一个节点
Node *ListSort(Node *head) //插入法排序函数
{   if(head == NULL)
        return NULL;
    if(head -> next == NULL)
        return head;
    Node *cur = head -> next;//指向第二个节点
    head -> next = NULL; //将头节点脱离下来，作为已排序队列
    while(cur)//将后面的节点依次插入已排序队列
    {     Node *prev = cur;
          cur = cur -> next;
          head = ListSortInsert(head, prev);
    }
    return head;          }
```



```
Node *ListSortInsert(Node *h, Node *p)//插入一个节点
{  if(p -> data < h -> data)
   {     p -> next = h;
         h = p;
         return h;
   }//插入头部
```



```
Node *ListSortInsert(Node *h, Node *p)//插入一个节点
{  if(p -> data < h -> data)
   {     p -> next = h;
         h = p;
         return h;
   }//插入头部
   Node *cur = h;
   Node *prev; //用cur和prev操纵已排序队
   while(cur)
   {     if(p -> data < cur -> data)
               break;
         prev = cur;
         cur = cur -> next;
   }//查找合适的位置，在prev后插入
         p -> next = prev -> next;
         prev -> next = p;
         return h;
   }
```



基于链表的信息检索

- 一般不适合用折半查找法。

例 基于链表的顺序查找程序。

…

```
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 = 181220999;
  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)
{
  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…
```

```
for(p = head; p && p -> id != x ; p = p->next) ;
```

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≈200行
  - 继续保持良好的编程习惯

Thanks!