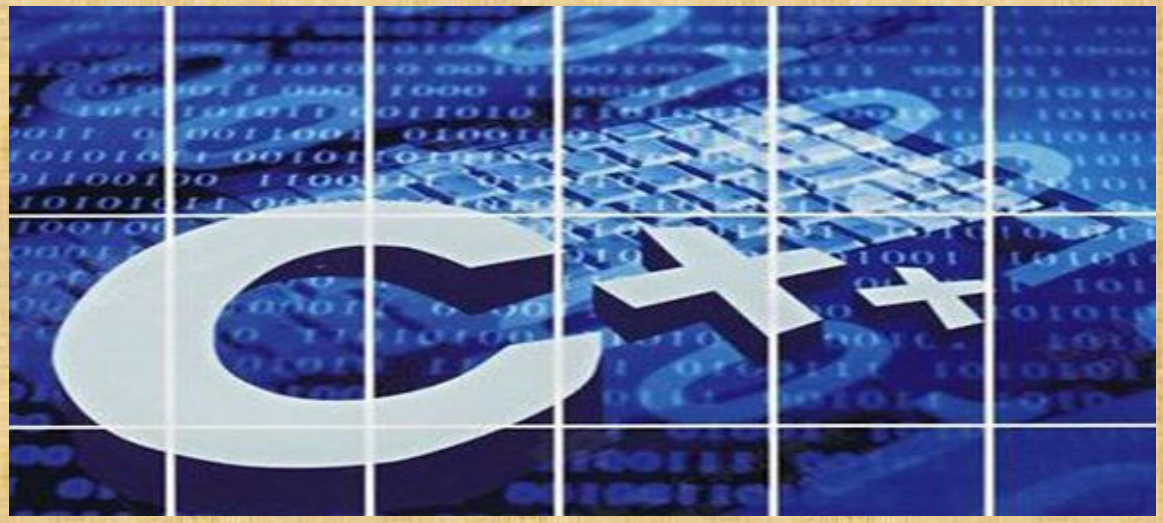
第七章 自定义数据类型



本章主要内容

7.1 结构体类型、变量定义与访问

7.2 结构体与函数

7.3 链表

7.4 共用体

7.5 枚举类型

7.3 链表

- 链表的概念
- 链表的创建
- 链表的应用

7.3.1 链表的概念

程序运行时需要内存可以通过动态内存分配来实现。

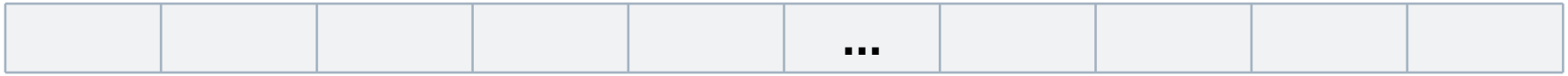
为了提高内存的使用效率，每次不能过多申请内存，否则容易造成内存浪费，策略是，在程序运行过程中，根据实时的需要逐次根据当前的需要动态分配内存。如何将多次动态申请的内存组织管理起来，方便进行数据增加、删除、修改等？

7.3.1 链表的概念

编写一个能够管理5000名职工信息的系统。

解决方案1：分配数组

分配具有5000个Employee变量存储空间的数组。



- 不能处理任意多的职工信息；
- 删除时出现中间空位，需要移动元素；
- 插入元素时需要移动元素；
- 如果职工数量少，会出现内存浪费；
- 操作较复杂、耗时；

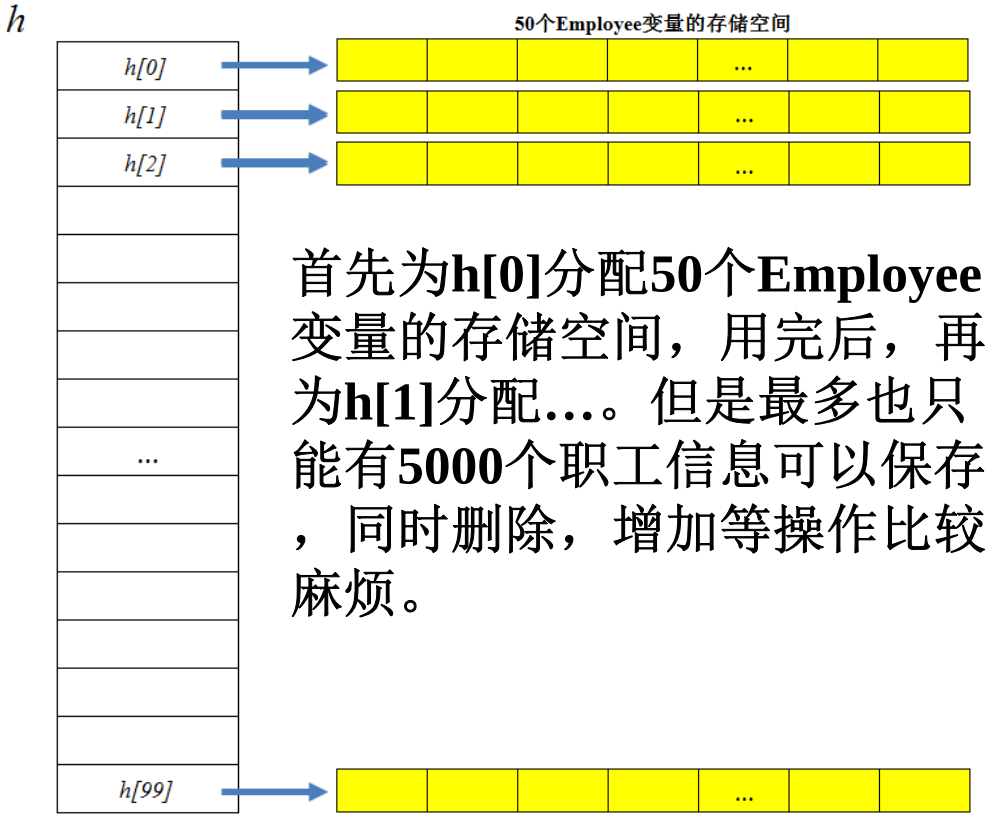
7.3.1 链表的概念

- 不能处理任意多的职工信息；
- 删除时出现中间空位，需移动元素；
- 插入时需要移动元素；
- 操作复杂、耗时；

解决方案2：

```
#include <iostream>
using namespace std;

struct Employee{
        //...
};
int main(){
   struct Employee h[100];
   h[0] = new struct Employee[50];
   h[1] = new struct Employee[50];
   //...
   h[99] = new struct Employee[50];
    return 0;
}
```

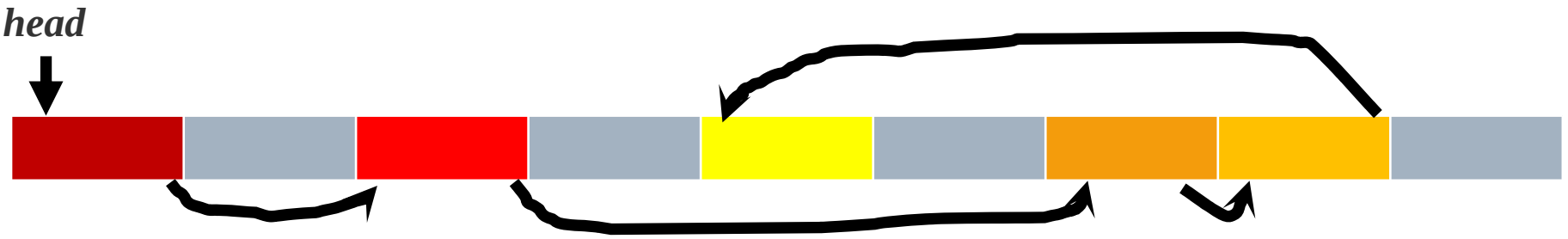


首先为h[0]分配50个Employee变量的存储空间，用完后，再为h[1]分配…。但是最多也只能有5000个职工信息可以保存，同时删除，增加等操作比较麻烦。

7.3.1 链表的概念

链表是一种数据存储方式（结构），它将相同类型的数据元素在内存中通过指针首尾串接起来，形成一个链。

利用链表存储数据元素时，每次只分配存储一个数据元素（如职工信息）的空间，然后利用指针把每次分配的空间串联起来。利用一个指针***head***存储第一个变量的地址。

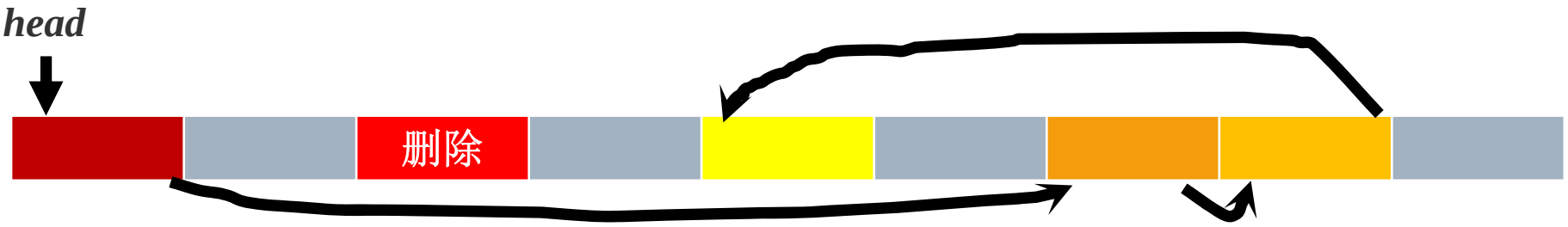


每个数据元素称为链表的结点；head称为首指针；最后一个结点的指针不指向任何结点。

7.3.1 链表的概念

链表是一种数据存储方式（结构），它将相同类型的数据元素在内存中通过指针首尾串接起来，形成一个链。

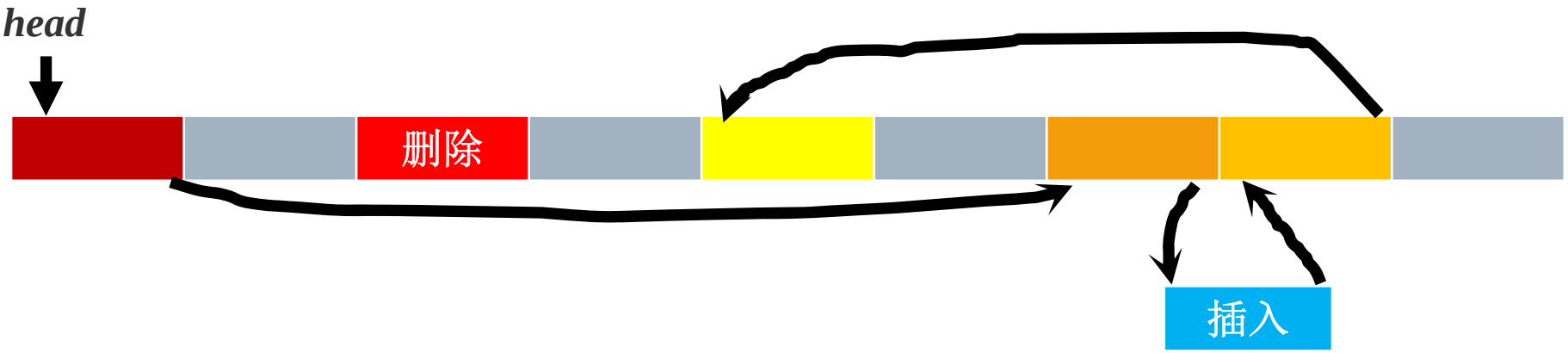
利用链表存储数据元素时，每次只分配存储一个数据元素（如职工信息）的空间，然后利用指针把每次分配的空间串联起来。利用一个指针***head***存储第一个变量的地址。



7.3.1 链表的概念

链表是一种数据存储方式（结构），它将相同类型的数据元素在内存中通过指针首尾串接起来，形成一个链。

利用链表存储数据元素时，每次只分配存储一个数据元素（如职工信息）的空间，然后利用指针把每次分配的空间串联起来。利用一个指针*head*存储第一个变量的地址。



7.3.1 链表的概念（单向链表）　结点和头指针

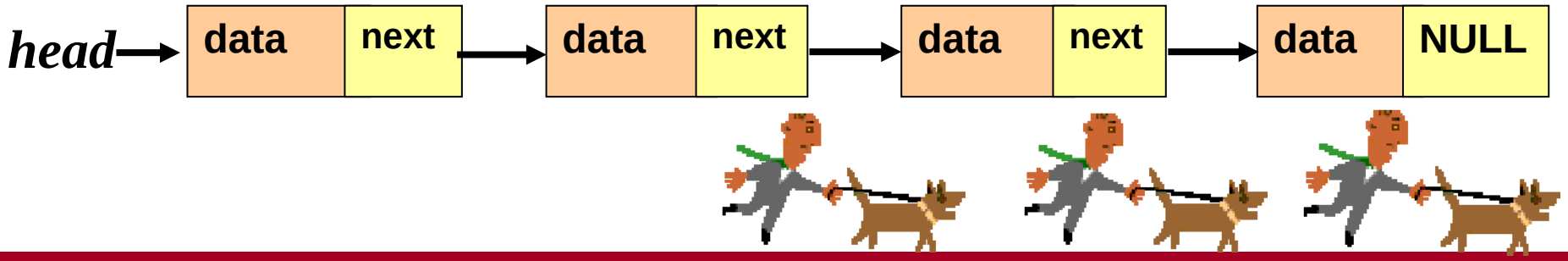
链表中的每个结点为一个结构体类型变量，该结构体类型成员包括两个：

（1）**数据域**。用来存放需要的信息（如职工的属性）；

（2）**指针域**。用来存放相邻结点的地址。

```
struct Link
{
    int  data;  //数据域
    struct Link *next;//指针域
};
```

struct Link* head;



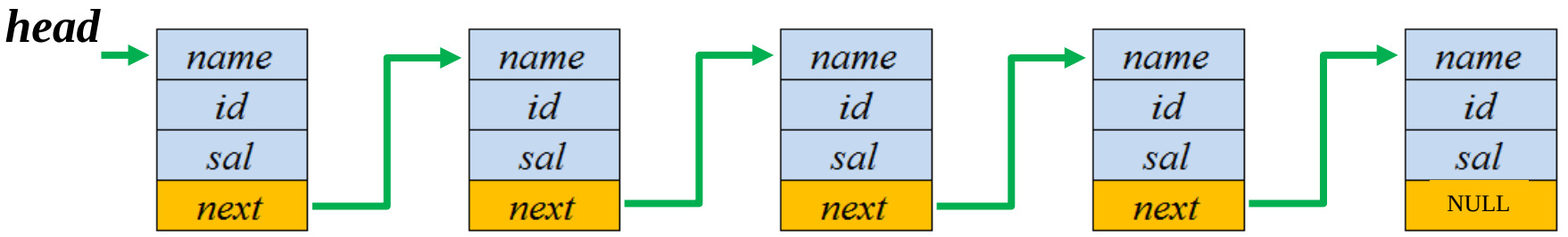
7.3.1 链表的概念（单向链表）

结点和头指针

```
struct Employee
{
    char name[20];
    char id[11];          //数据域
    int sal;
    struct Employee *next;//指针域
};

struct Employee*  head;
```

每个结点的存储空间通过new struct Employee分配。

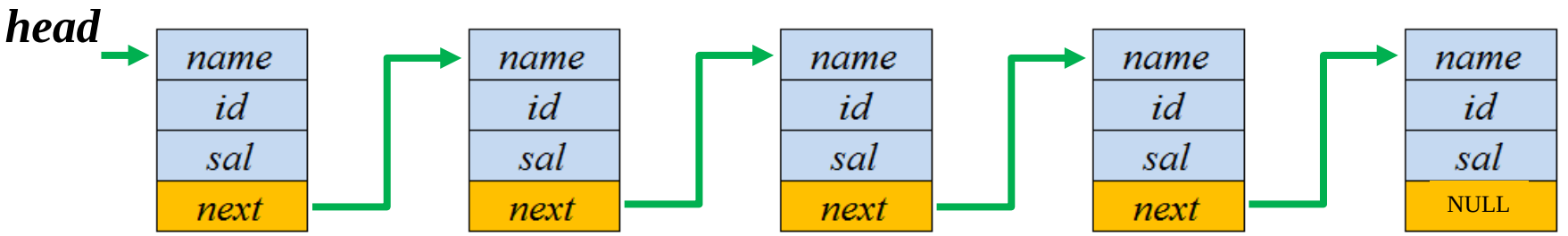


7.3.1 链表的概念（单向链表）

结点和头指针

```
struct Employee
{
    char name[20];
    char id[11];            //数据域
    int sal;
    struct Employee *next;//指针域
};

struct Employee*  head;
```



7.3.2 链表的操作

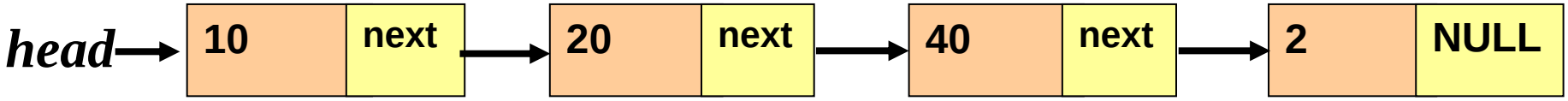
链表的操作包括：

- 插入。向链表中按照规则加入一个结点。
- 删除。删除某个结点。
- 检索。在链表中查找满足条件的结点。
- 遍历。按照顺序访问链表中的所有结点。

7.3.2 链表的操作

插入

例。将用户输入的多个整数按照输入的顺序采用单向链表存储，并实现增加，删除和查询功能。如用户输入**10,20,40,2,**则创建的链表如下。



7.3.2 链表的操作 插入

头指针（首指针）：是一个指针变量，用于存放第一个结点的地址。

```
struct Link* head = NULL; //头指针
struct Link* p = NULL, pr = NULL;//辅助的指针
```

向链表中插入结点时，首先确定节点按照什么逻辑插入到链表中，同时考虑链表自身的两种情况

- 链表为空 ***(head = NULL)***
- 链表不为空 ***(head != NULL)***

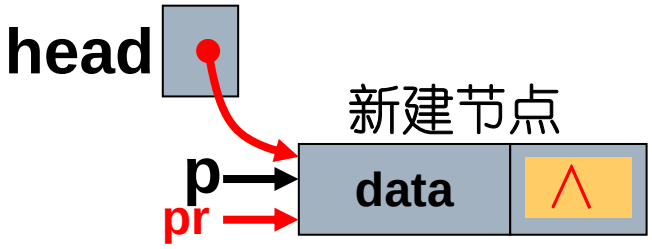
7.3.2 链表的操作

插入

- 若原链表为空表(head == NULL)，则将新建结点p置为头结点。pr始终指向最后一个结点。

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

(1)head = p



(2) pr = p

(3) pr->next = NULL

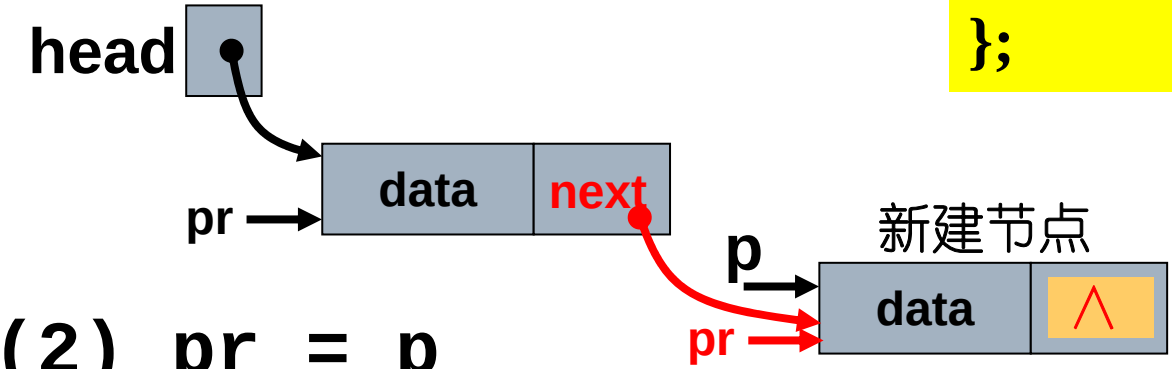
7.3.2 链表的操作

插入

- 若原链表为非空，则将新建节点p添加到表尾。pr始终指向最后一个节点。

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

(1) pr->next = p



(2) pr = p

(3) pr->next = NULL

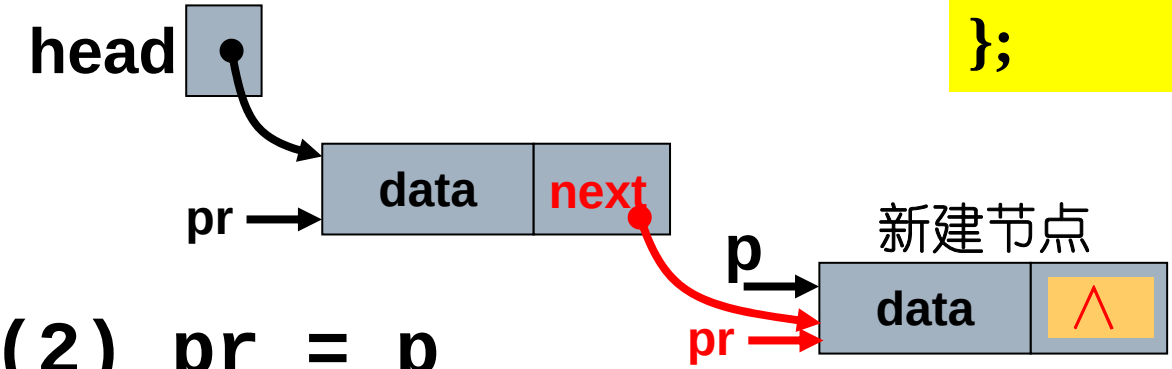
7.3.2 链表的操作

插入

- 若原链表为非空，则将新建节点p添加到表尾。pr始终指向最后一个节点。

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

(1) pr->next = p



(2) pr = p

(3) pr->next = NULL

7.3.2 链表的操作

遍历

- 访问每个结点。

```
p = head;
while(p != NULL){
    printf("%d", p->data);
    p=p->next;
}
```

```
#include <iostream>
using namespace std;
struct Link{
        int data;
        struct Link* next;
};
int main(){
        struct Link* head = NULL;
        struct Link* p=NULL, pr=NULL;
        //...假设已经创建链表
        p = head;
        while(p != NULL){
                printf("%d\n", p->data );
                p=p->next;
        }
        return 0;
}
```

7.3.2 链表的操作 检索

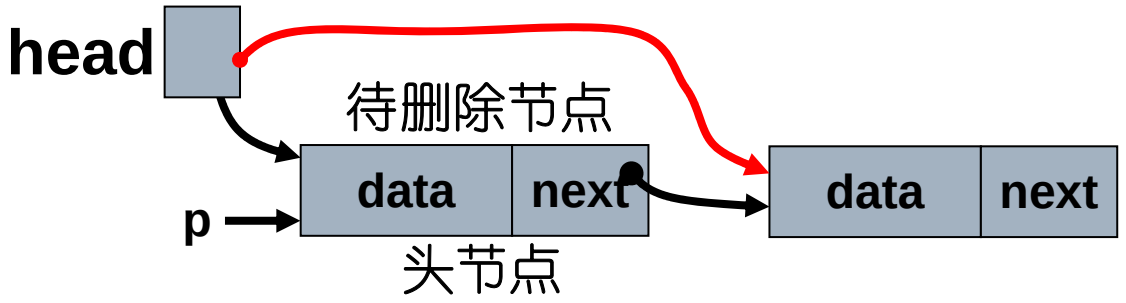
- 查找满足某个条件的结点。

```
#include <iostream>
using namespace std;
struct Link{
        int data;
        struct Link* next;
};
int main(){
        struct Link* head = NULL;
        struct Link* p=NULL, pr=NULL;
        //. . . 假设已经创建链表
        p = head;
        while(p != NULL){
                if (p->data==10) break;
                p=p->next;
        }
        //p为NULL，没有找到。否则找到
        return 0;
}
```

7.3.2 链表的操作 删除结点

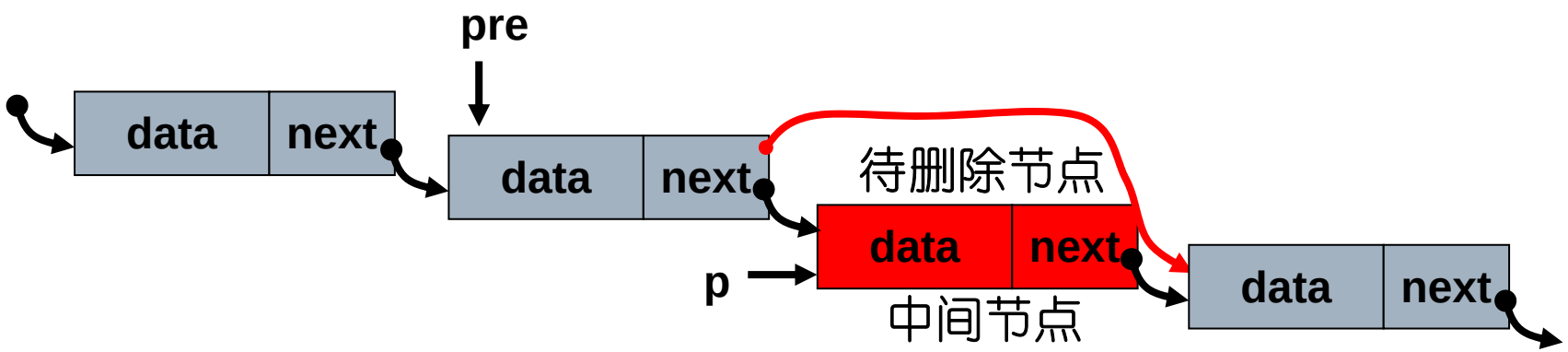
- 若原链表为空表(head==NULL)，则退出程序;
- 若待删除节点是头结点，则将head指向当前结点的下一个结点即可删除当前结点.

```
(1) head = p->next
(2) delete p
```



7.3.2 链表的操作　　删除结点

- 若待删除节点不是头节点，则将前一节点的指针域指向当前节点的下一节点即可删除当前节点，需要前后指针。




```
pre->next = p->next;
delete p;
```

7.3.2 链表的操作

删除结点

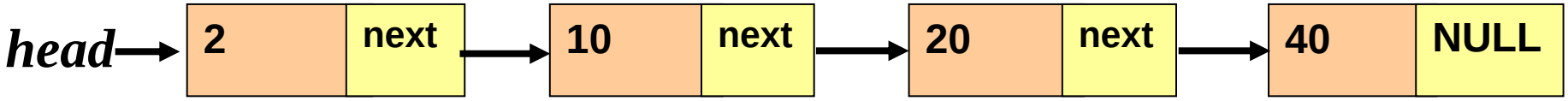
- 若待删除节点不是头节点，则将前一节点的指针域指向当前节点的下一节点即可删除当前节点，需要前后指针。

```
struct Link* pre = NULL, *p = NULL;
pre = head; //假设已经判断过，第一个结点不需要删除
p = head->next;
while(p!=NULL && p->data…){//查找满足条件的结点
        pr = pre->next;
        p = p->next;
}
if (p!=NULL){//如果找到
        pre->next = p->next;
}
```

7.3.2 链表的操作

插入

例。将用户输入的多个整数按照从小到大顺序采用单向链表存储。如用户输入10, 40, 20,2,则创建的链表如下。

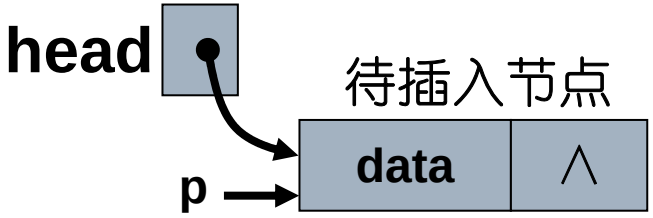


7.3.2 链表的操作

插入

- 若原链表为空表(**head = NULL**)，则将新节点p作为头节点，让head指向新节点p

(1) head = p

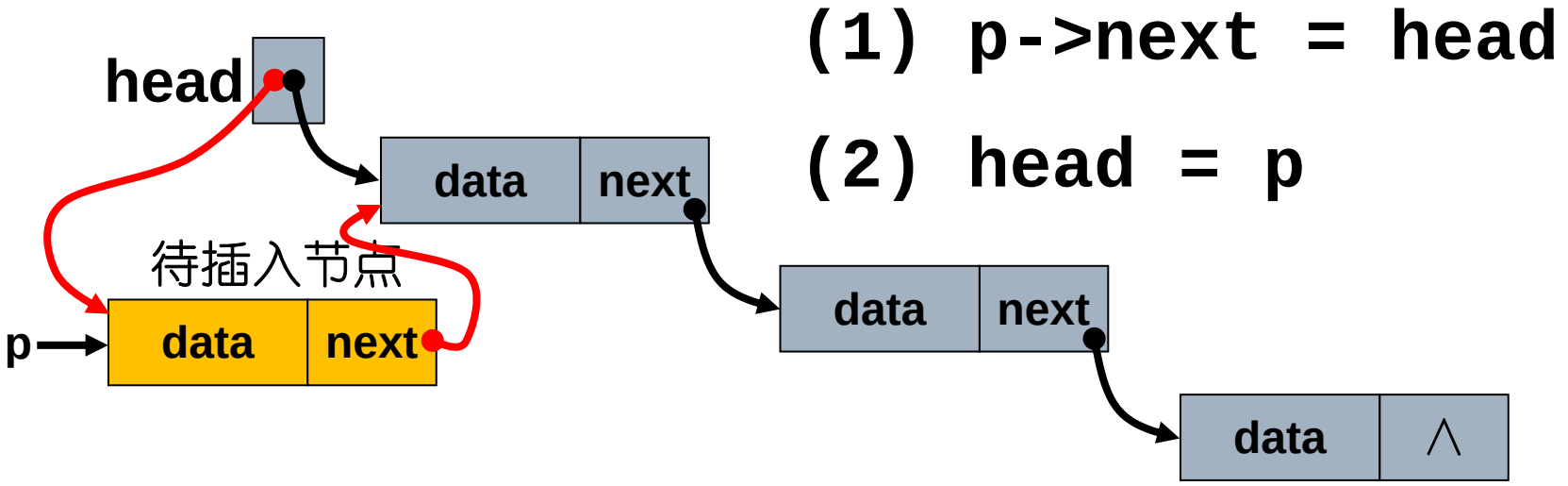



```
p = new struct link;
p->next = NULL;
p->data = nodeData;
```

7.3.2 链表的操作

插入

- 若原链表为非空，则按节点值（假设已按升序排序）的大小确定插入新节点的位置，可能的位置包括：头结点前、最后一个结点后或者链表中间。
- 若在头节点前插入新节点，则将新节点的指针域指向原链表的头节点，且让head指向新节点

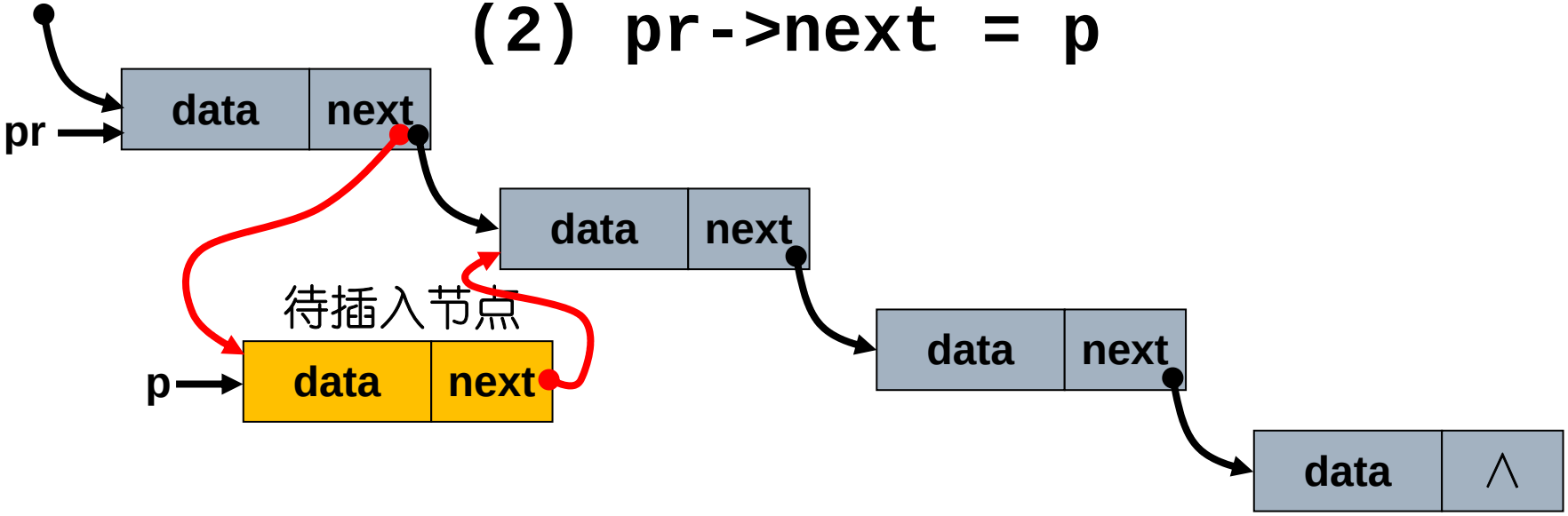


7.3.2 链表的操作

插入

- 若在链表中间插入新节点，则将新节点的指针域指向下一节点且让前一节点的指针域指向新节点

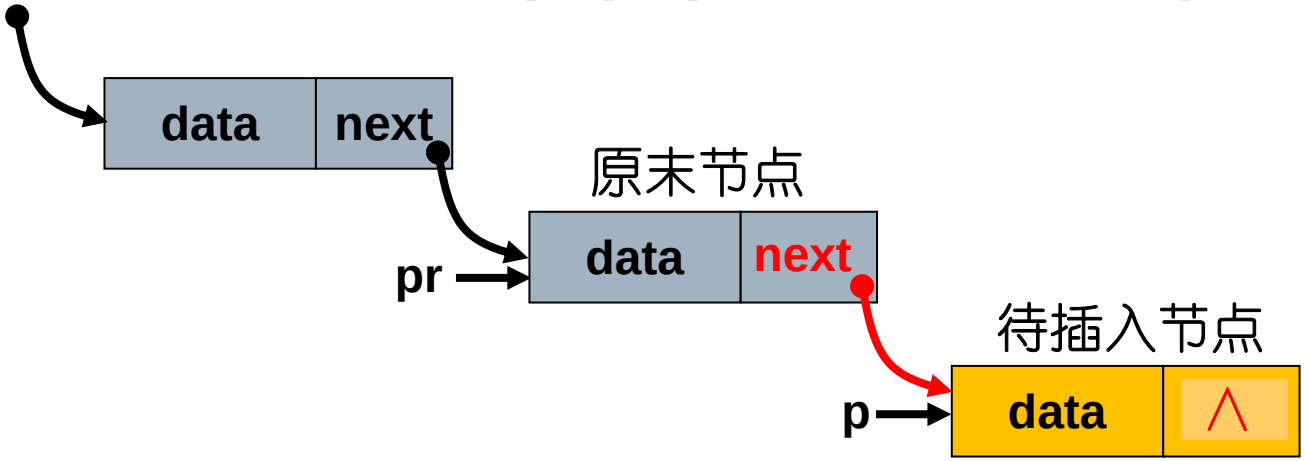
```
(1) p->next = pr->next
(2) pr->next = p
```



7.3.2 链表的操作 插入

- 若在表尾插入新节点，则末节点指针域指向新节点

(1) pr->next = p



作业

1. 创建单向链表，存储用户输入的任意个整数。（1）新增加的数据放到链表头部；（2）新增加的数据放到链表的尾部。

2. 有两个已经按照从小到大顺序存放的单向链表，编写函数将链表链接到一起，新链表中数据任然从小到大排序。

```
struct Link* merge(struct Link* ahead, struct Link* bhead);
```

3. 有17个人轮流数数，数到3的倍数的人退出，请问最后留下的一个人号码是多少。（采用循环链表）

欢 迎 交 流