专题二 链表

专题二 链表

- 基础知识回顾
- 链表的概念 (11.3.2)
- 单向链表的常用操作 (11.3.3)
- 链表的应用 (11.3.1)

基础知识回顾

- 自定义类型typedef (12.1)
  - 定义变量 int num[10]
  - 变量名 → 新类型名 num → IntArray
  - 加上typedef typedef int IntArray[10]
  - 用新类型名定义变量 IntArray a ←→ int a[10]

```
struct {   // 无类型名的结构定义
    …
} file;
struct {
    …
} FILE;
typedef struct {
    …
} FILE; (stdio.h)
```

```
FILE *fp; // 无类型名的结构指针定义

struct {    // 无类型名的结构指针定义
    …
} *fp;

(*fp).fd;   // 通过结构变量访问结构成员
fp->fd;     // 通过结构指针访问结构成员
```

基础知识回顾

- 动态内存分配malloc和free
  - 虽然存储块是动态分配的，但它的大小在分配后也是确定的，不要越界使用

```
typedef struct student {
    int num;
    char name[20];
    int score;
} Student;
```


```
struct student s;
Student s;
struct student *ps = (struct student *) malloc(sizeof(struct student));
Student *ps = (Student *) malloc(sizeof(Student));

free(ps);
```

专题二 链表

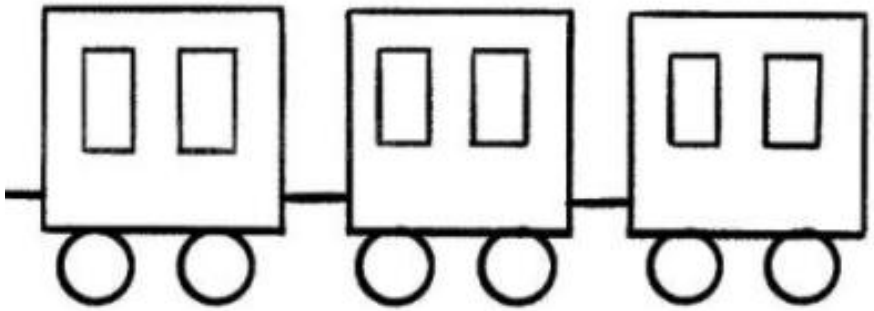
- 基础知识回顾
- 链表的概念 (11.3.2)
- 单向链表的常用操作 (11.3.3)
- 链表的应用 (11.3.1)

链表的概念

- 链表是一种常见且重要的动态存储分布的数据结构
  - 由若干个同一结构类型的“结点”依次串接而成
  - 结构 = 结点内容 + 指向下一个结点的指针

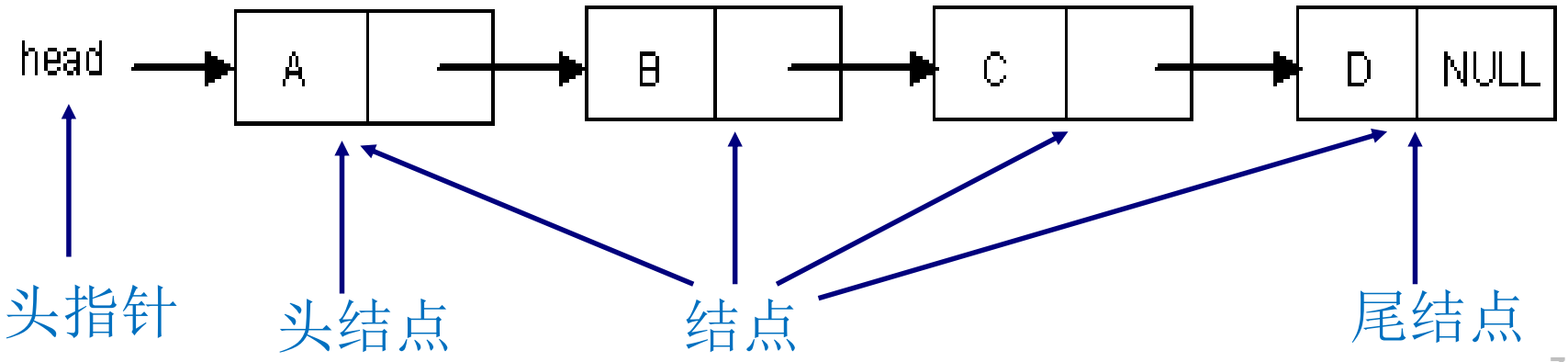
```
typedef struct node {
    ElementType data;          // 结点内容
    struct node *next;         // 指向下一个结点的指针
} Node;
```

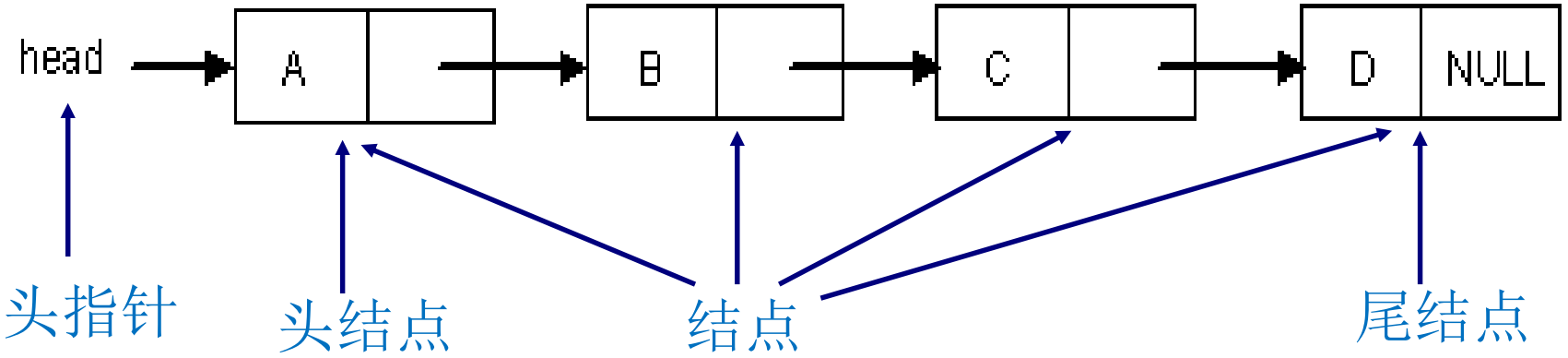
结构的递归定义



链表的概念

- 链表是由若干个同一结构类型的“结点”依次串接而成
  - 单向链表和双向链表
  - 数组　：数组名 = 链表：头指针
  - 数组　：a + i　= 链表：通过next指针间接访问
  - 字符串：'\0'　= 链表：next == NULL






```
typedef struct node {
    char data;
    struct node *next;
} Node;

Node *head = (Node *) malloc(sizeof(Node));
head->data = 'A';
head->next = (Node *) malloc(sizeof(Node));
head->next->data = 'B';
head->next->next = (Node *) malloc(sizeof(Node));
head->next->next->data = 'C';
head->next->next->next = (Node *) malloc(sizeof(Node));
head->next->next->next->data = 'D';
head->next->next->next->next = NULL;  // 尾结点的指针为NULL
```

数组 vs. 链表

- 数组
  - 事先定义固定长度的数组 (编译前，或运行时首次使用)
  - 在数组元素个数不确定时，可能会发生浪费内存空间的情况 (连续内存空间)
  - 插入和删除元素需要大量的数据移动
  - 优点是随机存取：存取任一元素只需要少量时间

- 链表
  - 动态存储分配的数据结构，长度可以动态增长
  - 根据需要动态开辟内存空间，比较方便地插入和删除元素/结点 (非连续内存空间)
  - 使用链表可以节省内存，操作效率高
  - 缺点是不能随机存取：访问序号为i的元素需要遍历整表

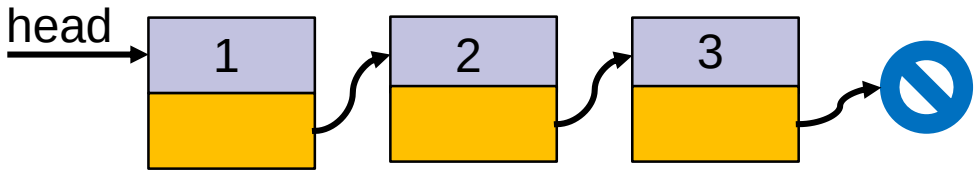
专题二 链表

- 基础知识回顾
- 链表的概念 (11.3.2)
- 单向链表的常用操作 (11.3.3)
  - 链表的建立
  - 链表的遍历
  - 结点插入
  - 结点删除
  - 链表的释放
- 链表的应用 (11.3.1)

单向链表的常用操作

- 链表不要求逻辑上相邻的两个数据元素在物理内存上也相邻，通过“链”建立起数据元素之间的逻辑关系
- 对链表的插入、删除不需要移动数据元素，只需要修改“链”
  - 数组的插入、删除需要移动数据元素
- 常用操作
  - 链表的创建和释放
  - 链表的插入和删除
  - 链表的遍历

链表的创建



- 输入n个整数，创建链表

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

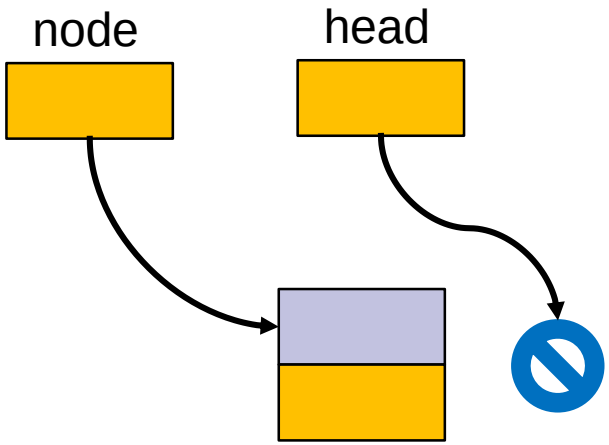
动态内存分配需检查是否成功
由于篇幅，课件代码省略检查
作业务必增加检查是否成功

```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        if (node == NULL) {
            printf("Memory Allocation Failed.");
            exit(0);
        }
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```

链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

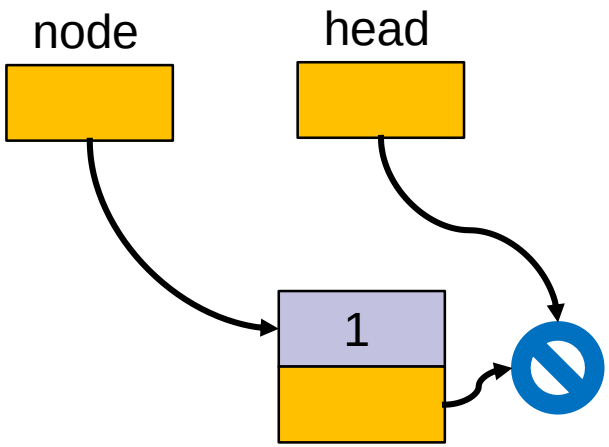
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

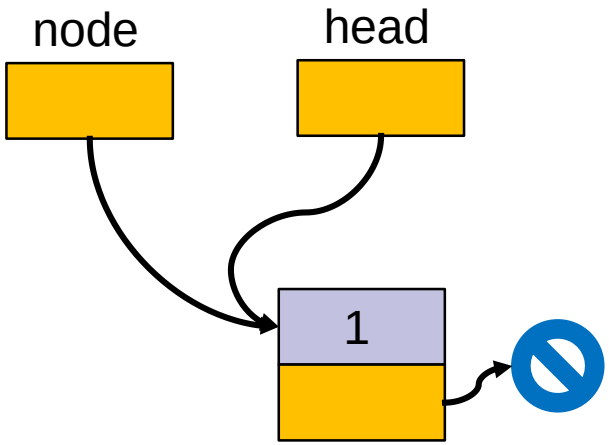
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

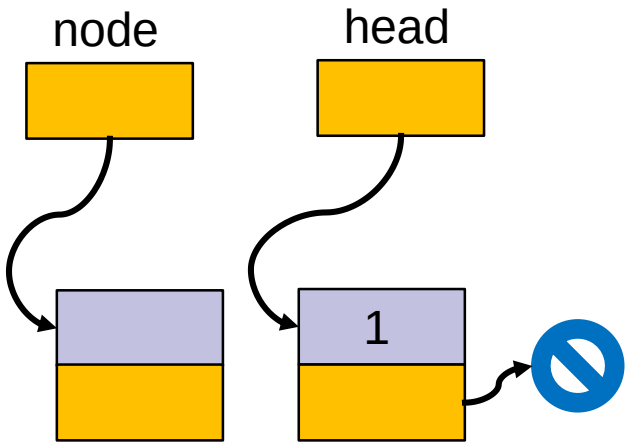
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

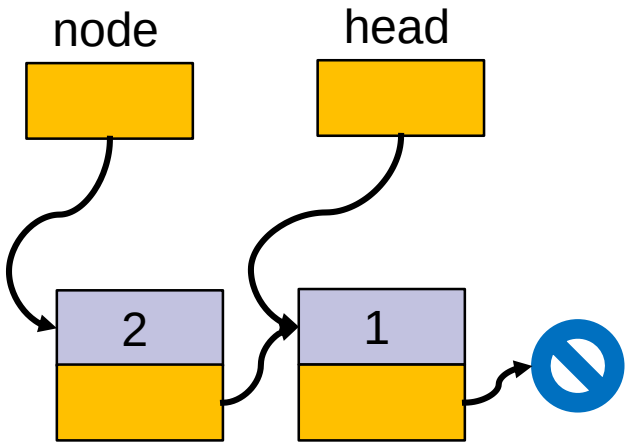
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

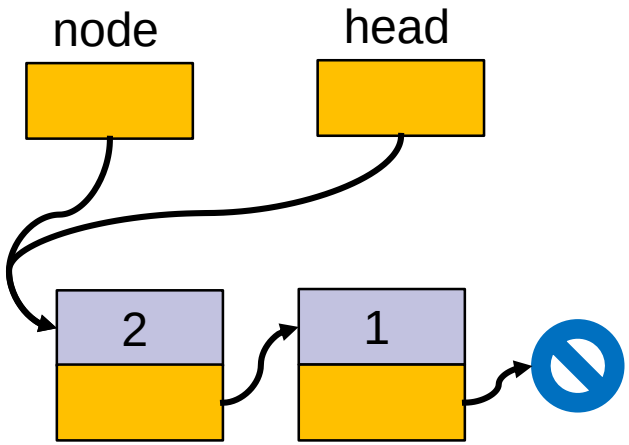
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

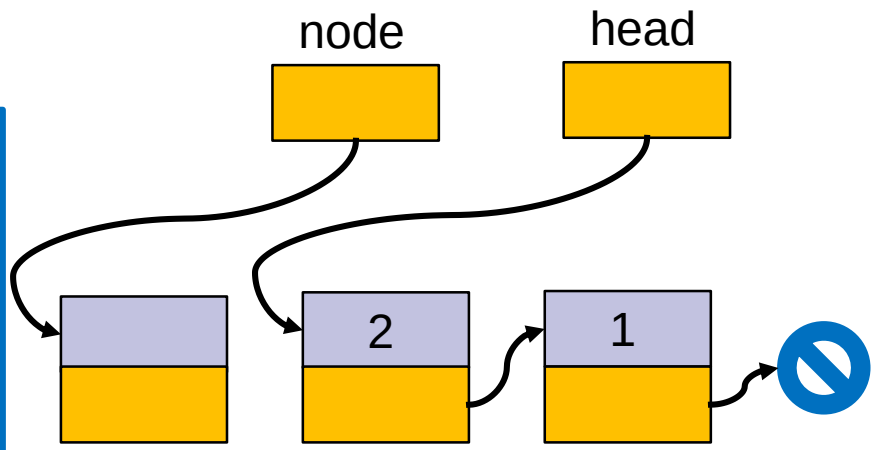
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

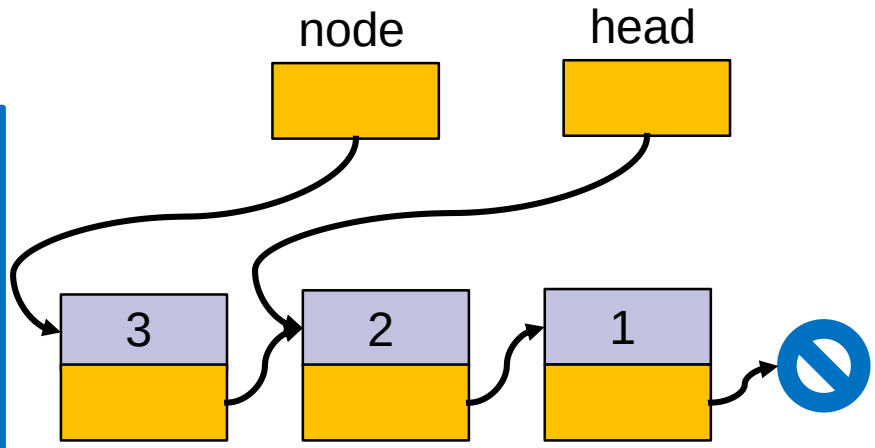
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

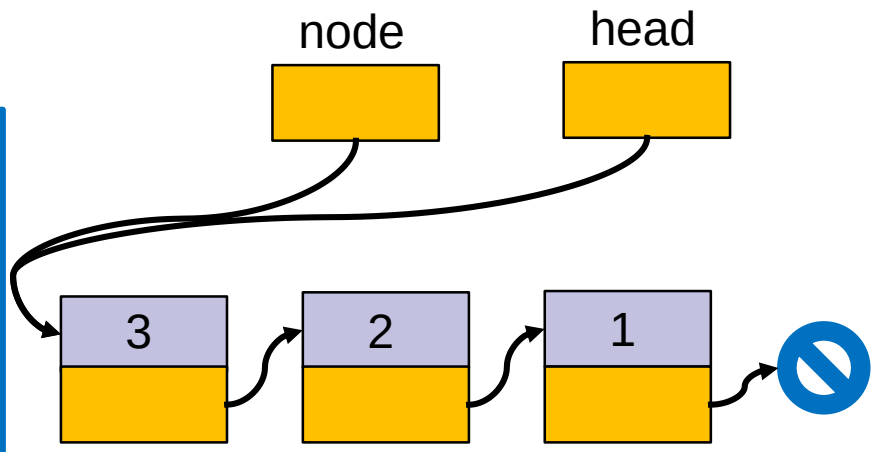
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，创建链表
  - 3 1 2 3

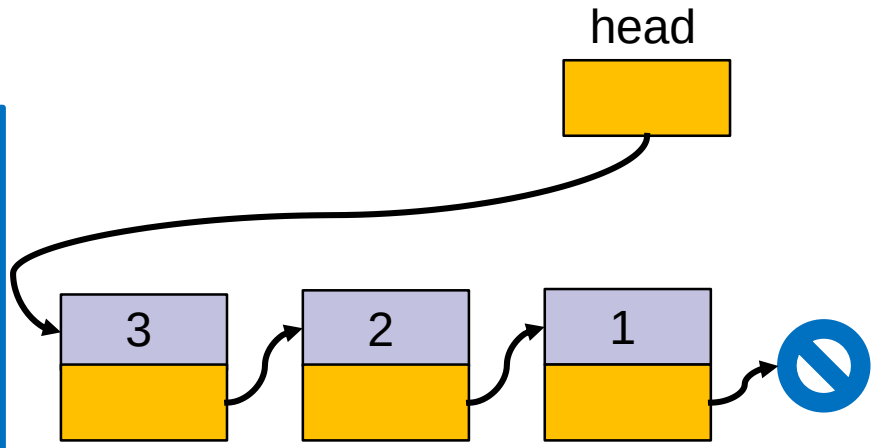
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，逆序创建链表
  - 3 1 2 3

```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```



链表的创建

- 输入n个整数，顺序创建链表
  - 思路1：每次插入链尾

```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        Node *tail = head;
        while (tail->next != NULL)
            tail = tail->next;
        tail->next = node;
    }
    return head;
}
```

指针需先判断后使用

空链表需要特殊处理

链表的创建

- 输入n个整数，顺序
  - 思路1：每次插入链尾
    - 每次插入都需要遍历链表
  - 思路2：记录尾结点

```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)   // 空链表
            head = node;
        else {                      // 非空链表
            Node *tail = head;
            while (tail->next != NULL)
                tail = tail->next;
            tail->next = node;
        }
    }
    return head;
}
```

链表的创建

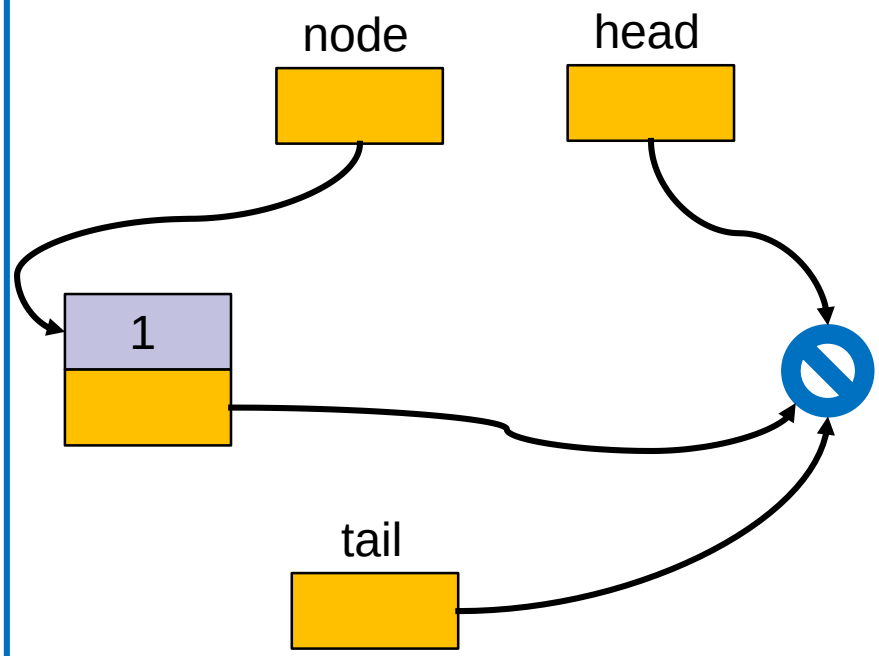
- 输入n个整数，顺序创建链表
  - 思路1：每次插入链尾
    - 每次插入都需要遍历链表
  - 思路2：记录尾结点
    - 使用变量记录额外信息

```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)  // 空链表
            head = node;
        else                        // 非空链表
            tail->next = node;
        tail = node;            // 更新尾结点
    }
    return head;
}
```

链表的创建

- 输入n个整数，顺序创建链表 (3 1 2 3)

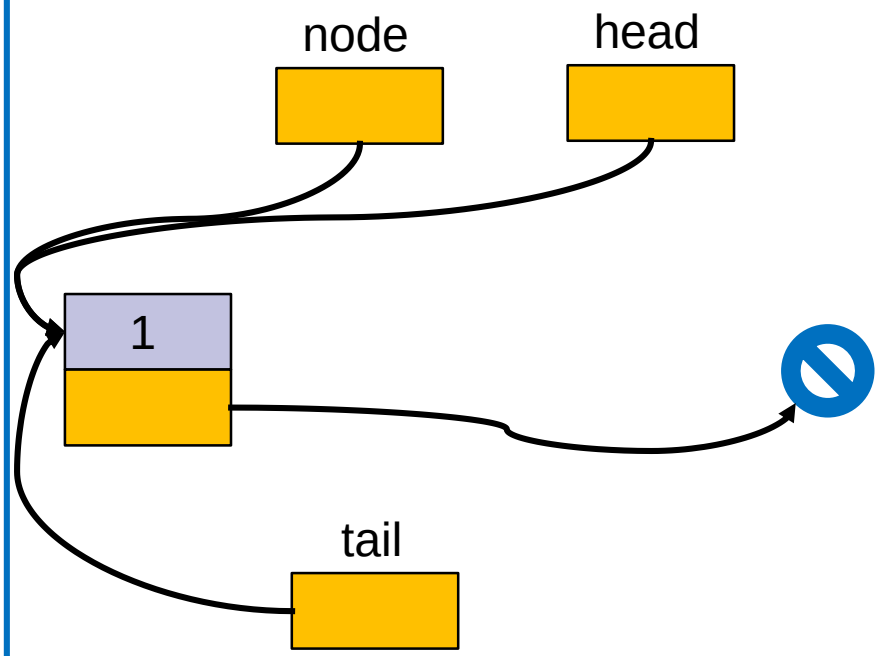
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)  // 空链表
            head = node;
        else                        // 非空链表
            tail->next = node;
        tail = node;              // 更新尾结点
    }
    return head;
}
```



链表的创建

- 输入n个整数，顺序创建链表 (3 1 2 3)

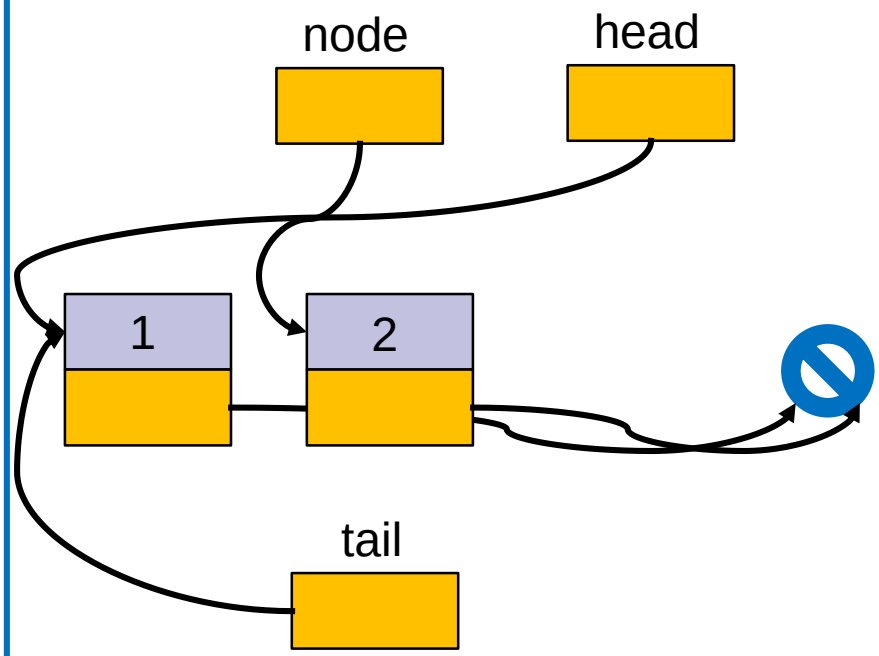
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)  // 空链表
            head = node;
        else                        // 非空链表
            tail->next = node;
        tail = node;                // 更新尾结点
    }
    return head;
}
```



链表的创建

- 输入n个整数，顺序创建链表 (3 1 2 3)

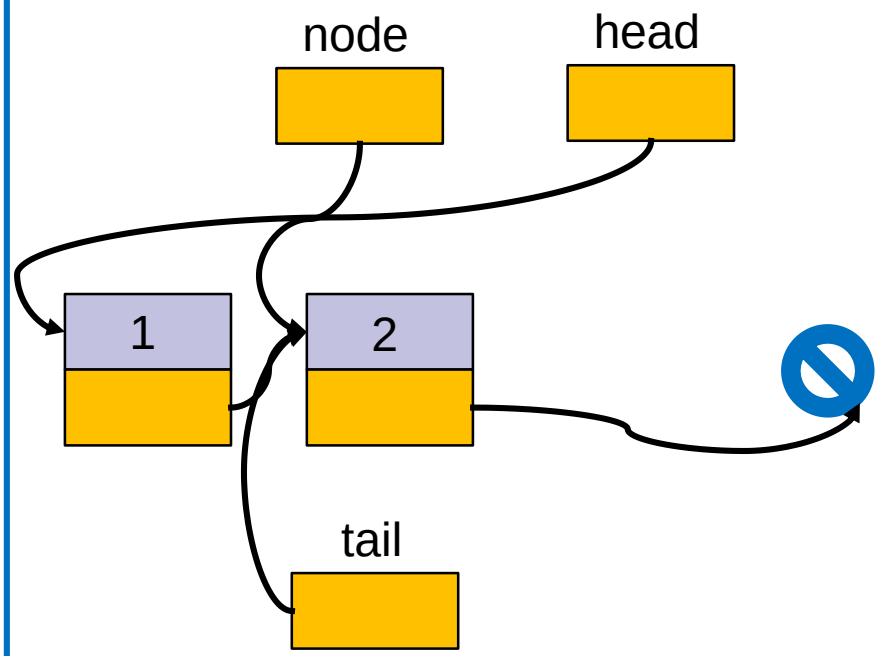
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)  // 空链表
            head = node;
        else                        // 非空链表
            tail->next = node;
        tail = node;              // 更新尾结点
    }
    return head;
}
```



链表的创建

- 输入n个整数，顺序创建链表 (3 1 2 3)

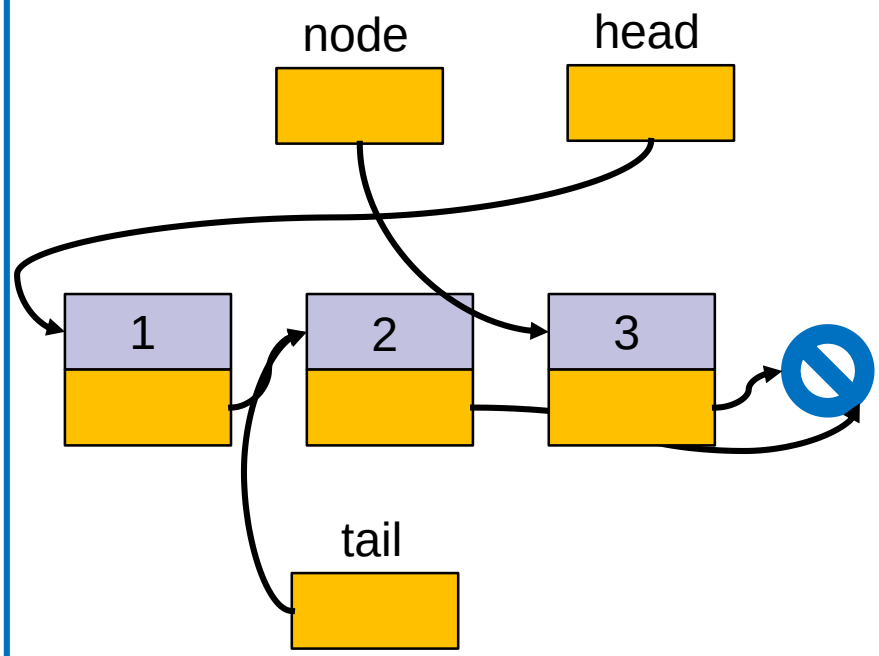
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)  // 空链表
            head = node;
        else                        // 非空链表
            tail->next = node;
        tail = node;              // 更新尾结点
    }
    return head;
}
```



链表的创建

- 输入n个整数，顺序创建链表 (3 1 2 3)

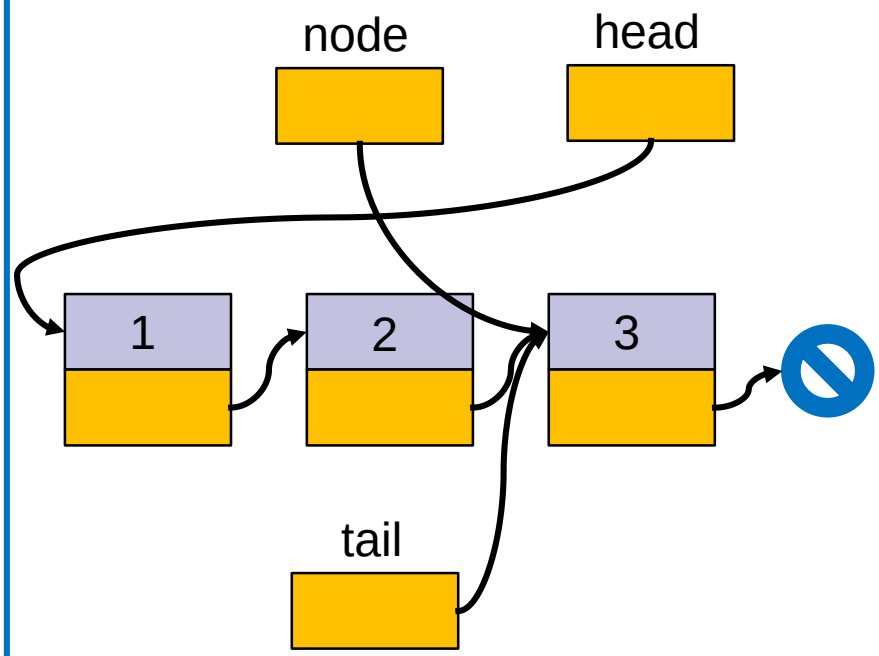
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)  // 空链表
            head = node;
        else                        // 非空链表
            tail->next = node;
        tail = node;              // 更新尾结点
    }
    return head;
}
```



链表的创建

- 输入n个整数，顺序创建链表 (3 1 2 3)

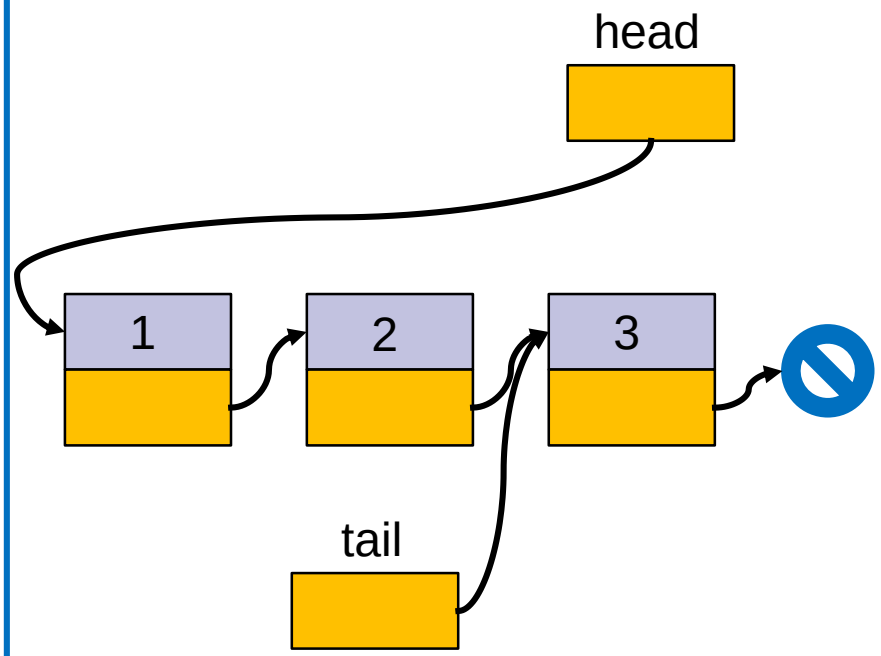
```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)  // 空链表
            head = node;
        else                        // 非空链表
            tail->next = node;
        tail = node;                // 更新尾结点
    }
    return head;
}
```



链表的创建

- 输入n个整数，顺序创建链表 (3 1 2 3)

```
Node * inputList() {
    int i, n, data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)  // 空链表
            head = node;
        else                        // 非空链表
            tail->next = node;
        tail = node;              // 更新尾结点
    }
    return head;
}
```



链表的创建

- 以-1结尾的一串整数(不包含-1)，创建链表

```
Node * inputList() {
    Node *node = NULL;
    Node *head = (Node *) malloc(sizeof(Node));
    scanf("%d", &head->data);
    head->next = NULL;
    if (head->data == -1)
        return NULL;
    while (1) {
        node = (Node *) malloc(sizeof(Node));
        scanf("%d", &node->data);
        if (node->data == -1)
            break;
        node->next = head;
        head = node;
    }
    return head;
}
```

逆序 vs. 顺序？是否有问题？

存在内存泄漏

考虑多种测试情况

- 空链表
- 单个元素链表
- 多个元素链表

链表的创建

- 以-1结尾的一串整数(不包含-1)，创建链表

```
Node * inputList() {
    int data;
    Node *head = NULL, *node = NULL;
    while (1) {
        scanf("%d", &data);
        if (data == -1)
            break;
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = head;
        head = node;
    }
    return head;
}
```

逆序

```
Node * inputList() {
    int data;
    Node *head = NULL, *tail = NULL;
    Node *node = NULL;
    while (1) {
        scanf("%d", &data);
        if (data == -1) break;
        node = (Node *) malloc(sizeof(Node));
        node->data = data;
        node->next = NULL;
        if (head == NULL)
            head = node;
        else
            tail->next = node;
        tail = node;
    }
    return head;
}
```

顺序

链表的遍历

- 链表的遍历
  - 查找元素
  - 计算链表的长度
  - 打印链表
  - 复制链表
  - ……

```
void traversalList(Node *head) {
    while (head != NULL) {
        head->data;
        head = head->next;
    }
}
```

修改head，会影响主调函数的头指针？

while (head != NULL) → while (head)

```
Node * searchList(Node *head, int value) {
    while (head) {
        if (head->data == value)
            break;
        head = head->next;
    }
    return head;
}
```

```
int lengthOfList(Node *head) {
    int length = 0;
    while (head) {
        length++;
        head = head->next;
    }
    return length;
}
```

链表的遍历

- 链表的遍历
  - 查找元素
  - 计算链表的长度
  - 打印链表
  - 复制链表
  - ……

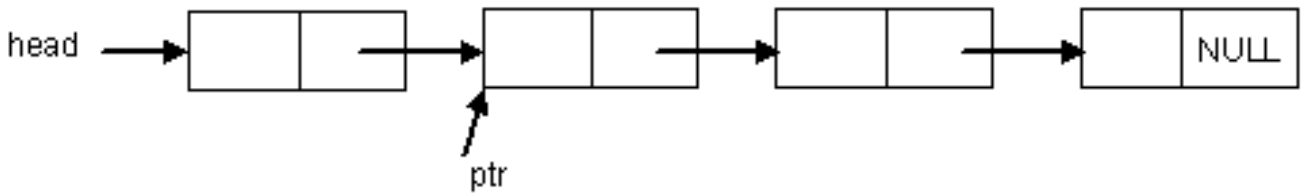
```
void printList(Node *head) {
    while (head) {
        printf("%d ",  head->data);
        head = head->next;
    }
}
```

链表复制 = 旧链表遍历 + 新链表创建

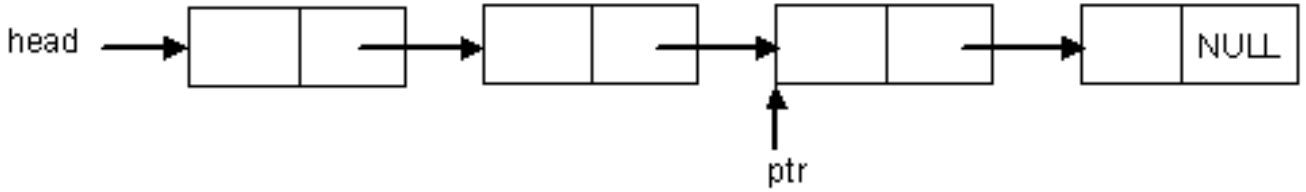
```
Node * copyList(Node *head) {
    Node *another = NULL, *tail = NULL;
    Node *node = NULL;
    while (head) {
        node = (Node *) malloc(sizeof(Node));
        node->data = head->data;
        node->next = NULL;
        if (another == NULL)
            another = node;
        else
            tail->next = node;
        tail = node;
        head = head->next;
    }
    return another;
}
```

链表的遍历 (图11.12)

```
for(ptr = head; ptr != NULL; ptr = ptr->next)
    printf("%d\t%s\t%d\n", ptr->num, ptr->name, ptr->score);
```



(a) 执行 ptr = ptr->next 前



(b) 执行 ptr = ptr->next 后

结点插入

- 无序链表的插入

插入到链表头

```
Node * insertList(Node *head, int value) {
    Node *node = NULL;
    node = (Node *) malloc(sizeof(Node));
    if (node == NULL) {
        printf("Memory Allocation Failed.\n");
        exit(0);
    }
    node->data = value;
    node->next = head;
    return node;
}
```

动态内存分配需检查是否成功
由于篇幅，课件代码省略检查

插入到链表尾

```
Node * insertList(Node *head, int value) {
    Node *node = NULL;
    node = (Node *) malloc(sizeof(Node));
    node->data = value;
    node->next = NULL;

    while (head->next != NULL)
        head = head->next;
    head->next = node;
    return head;
}
```

空链表需要特殊处理
指针使用前需先判断

结点插入

- 无序链表的插入

插入到链表头

```
Node * insertList(Node *head, int value) {
    Node *node = NULL;
    node = (Node *) malloc(sizeof(Node));
    if (node == NULL) {
        printf("Memory Allocation Failed.\n");
        exit(0);
    }
    node->data = value;
    node->next = head;
    return node;
}
```

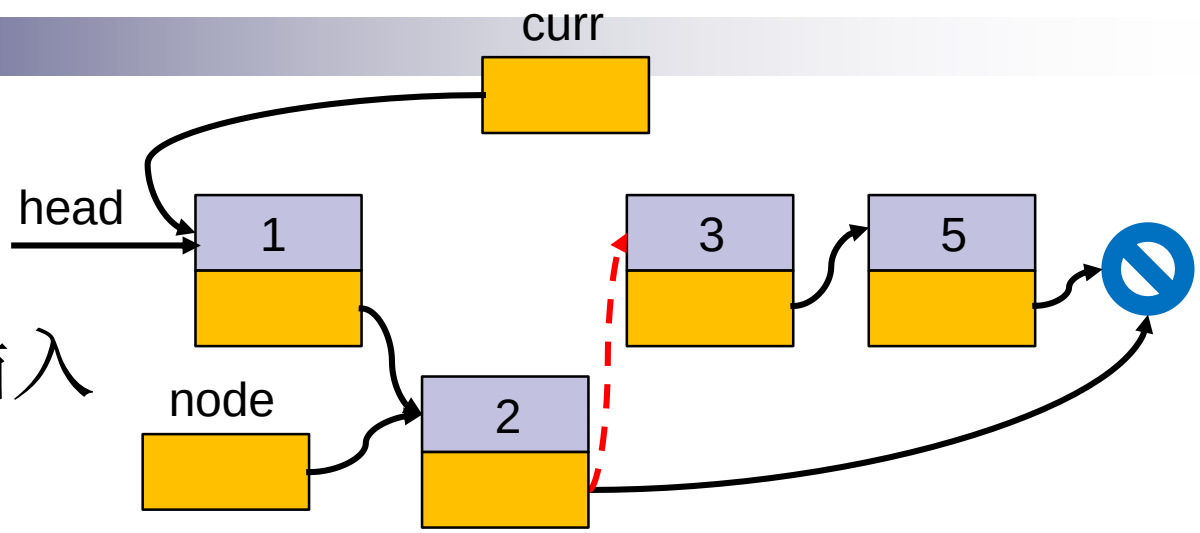
插入到链表尾

```
Node * insertList(Node *head, int value) {
    Node *node = NULL;
    node = (Node *) malloc(sizeof(Node));
    node->data = value;
    node->next = NULL;

    if (head == NULL) // 空链表
        head = node;
    else {                    // 非空链表
        Node *tail = head;
        while (tail->next != NULL)
            tail = tail->next;
        tail->next = node;
    }
    return head;
}
```

结点插入

- 有序链表的插入
  - 插入2

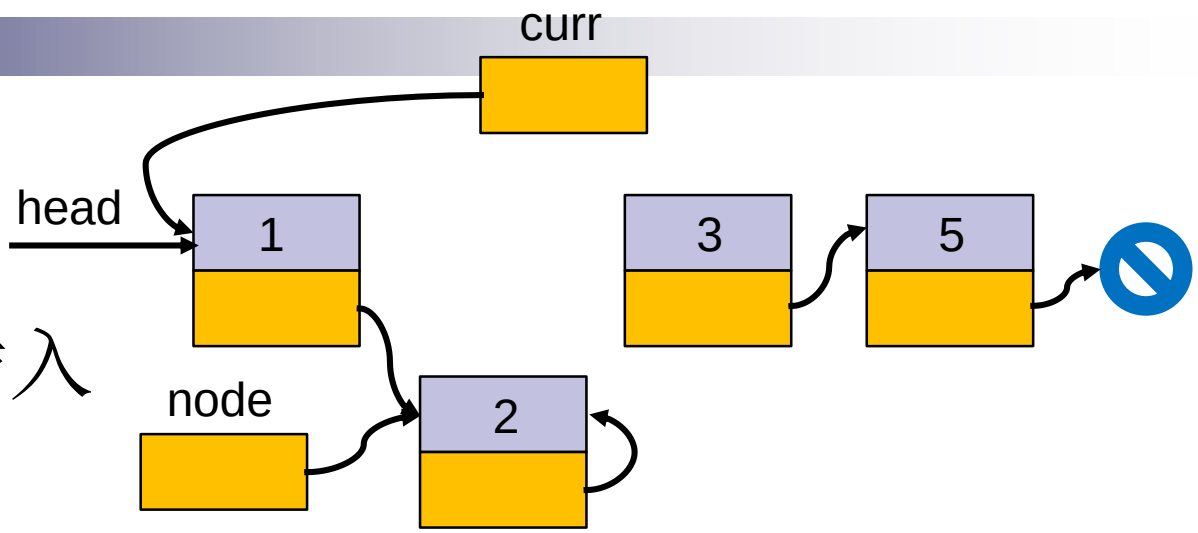



```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)    // 空链表
        head = node;
    else {                    // 非空链表
        while (curr->next->data < value && curr->next != NULL)
            curr = curr->next;
        curr->next = node;
    }
    return head;
}
```

结点插入

- 有序链表的插入
  - 插入2



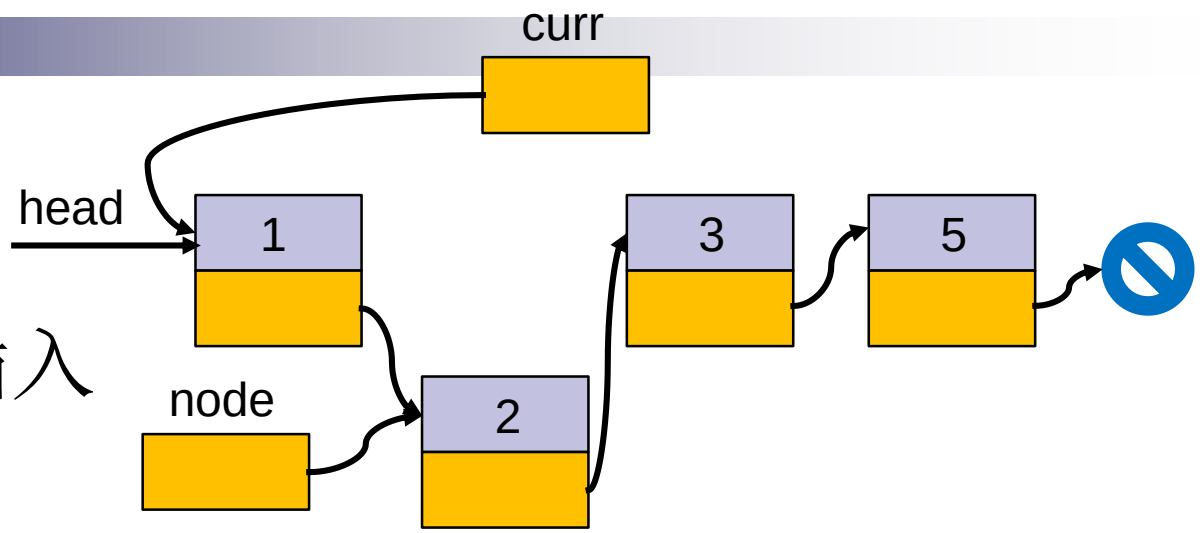

```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)    // 空链表
        head = node;
    else {                    // 非空链表
        while (curr->next->data < value && curr->next != NULL)
            curr = curr->next;
        curr->next = node;
        node->next = curr->next;
    }
    return head;
}
```

✗ 先断后连

结点插入

- 有序链表的插入
  - 插入2

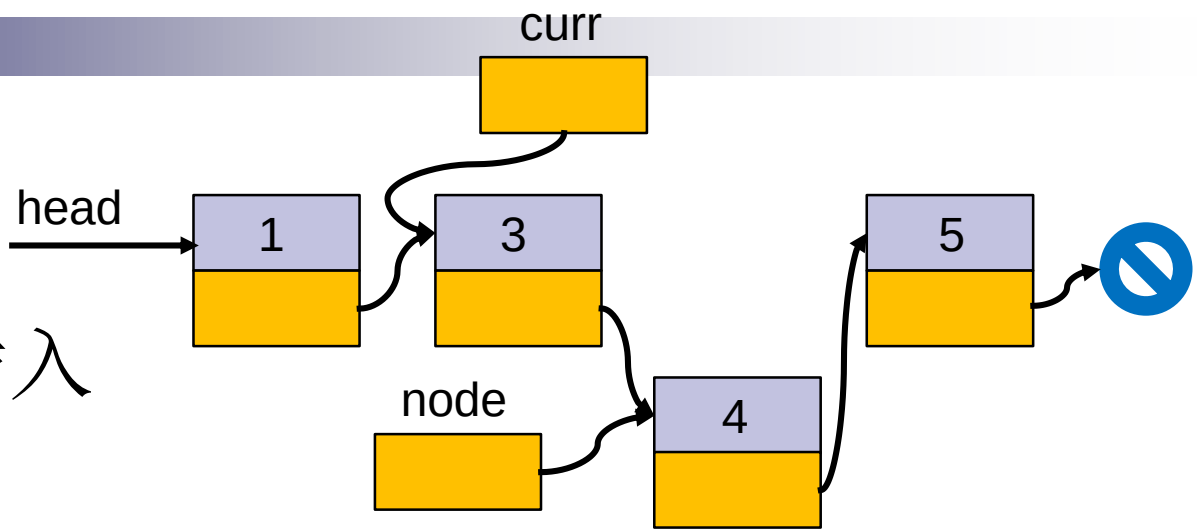



```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)    // 空链表
        head = node;
    else {                    // 非空链表
        while (curr->next->data < value && curr->next != NULL)
            curr = curr->next;
        node->next = curr->next;    // 先连后断
        curr->next = node;
    }
    return head;
}
```

结点插入

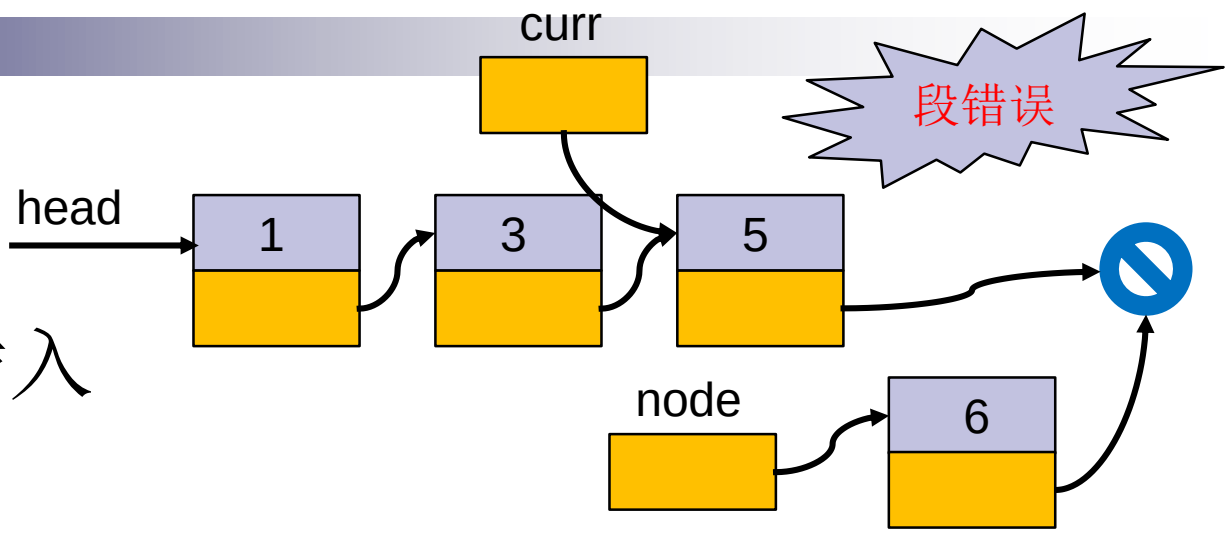
- 有序链表的插入
  - 插入4




```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)    // 空链表
        head = node;
    else {                    // 非空链表
        while (curr->next->data < value && curr->next != NULL)
            curr = curr->next;
        node->next = curr->next;    // 先连后断
        curr->next = node;
    }
    return head;
}
```

结点插入



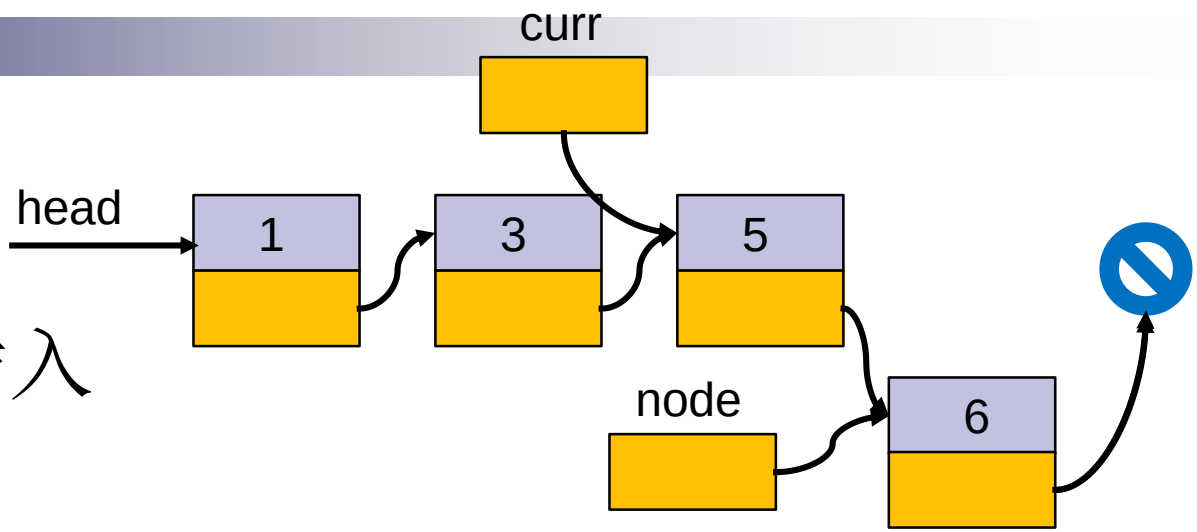
- 有序链表的插入
  - 插入6

```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)    // 空链表
        head = node;
    else {                    // 非空链表
        while (curr->next->data < value && curr->next != NUL
            curr = curr->next;
        node->next = curr->next;    // 先连后断
        curr->next = node;
    }
    return head;
}
```

指针需先判断是否有效
如果有效才能继续访问

结点插入

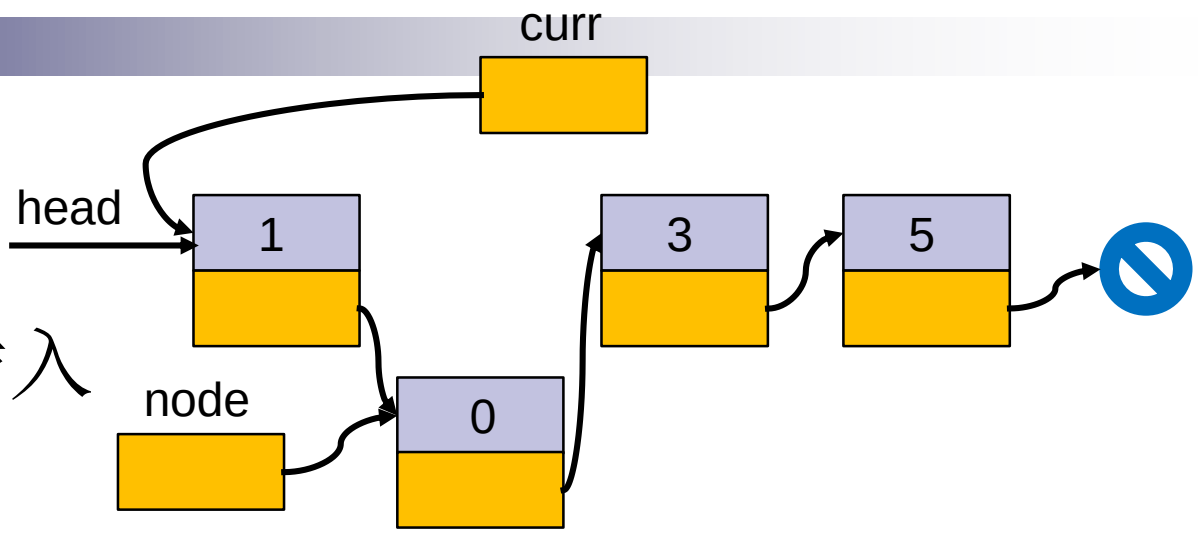


- 有序链表的插入
  - 插入6

```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)    // 空链表
        head = node;
    else {                    // 非空链表
        while (curr->next != NULL && curr->next->data < value) // 先判断后使用
            curr = curr->next;
        node->next = curr->next;   // 先连后断
        curr->next = node;
    }
    return head;
}
```

结点插入



- 有序链表的插入
  - 插入0

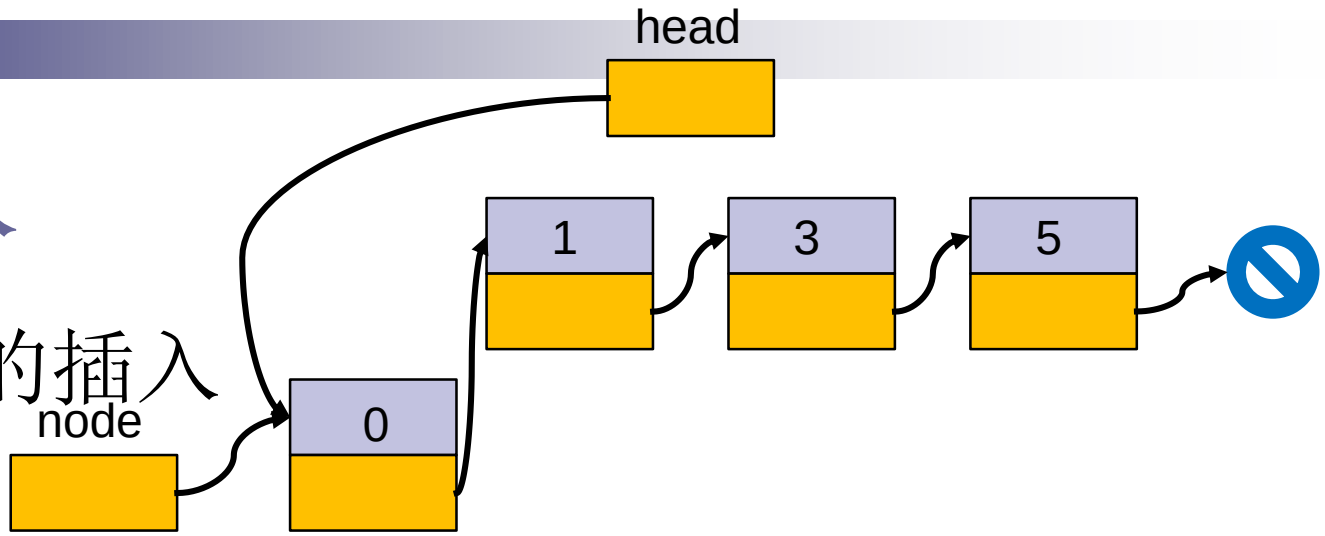
```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)    // 空链表
        head = node;
    else {                    // 非空链表
        while (curr->next != NULL && curr->next->data < value) // 先判断后使用
            curr = curr->next;
        node->next = curr->next;   // 先连后断
        curr->next = node;
    }
    return head;
}
```

✗ 跳过了head节点的判断

结点插入

- 有序链表的插入




```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)                  // 空链表
        head = node;
    else if (head->data > data) {      // 非空链表，插入链表头
        node->next = head;             // 先连后断
        head = node;
    } else {                           // 非空链表，插入链表中间或末尾
        while (curr->next != NULL && curr->next->data < value) // 先判断后使用
            curr = curr->next;
        node->next = curr->next;       // 先连后断
        curr->next = node;
    }
```

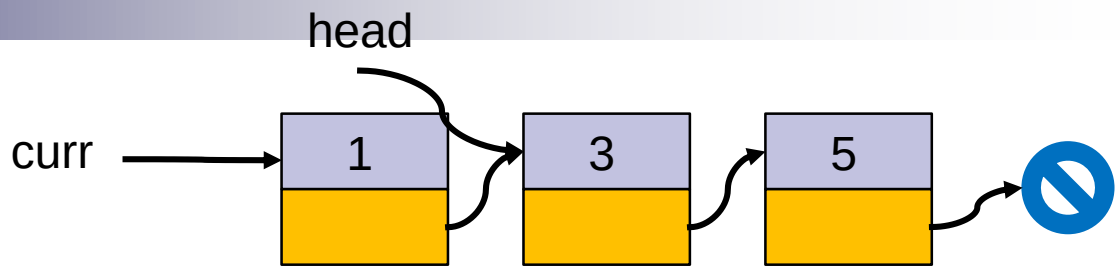
结点插入 (单指针实现)

- 有序链表的插入(空链表，插入链表头、中间、末尾)

```
Node * insertList(Node *head, int value) {
    Node *node = (Node *) malloc(sizeof(Node)), *curr = head;
    node->data = value;
    node->next = NULL;

    if (head == NULL)                  // 空链表
        head = node;
    else if (head->data > data) {  // 非空链表，插入链表头
        node->next = head;             // 先连后断
        head = node;
    } else {                               // 非空链表，插入链表中间或末尾
        while (curr->next != NULL && curr->next->data < value)  // 先判断后使用
            curr = curr->next;
        node->next = curr->next;   // 先连后断
        curr->next = node;
    }
    return head;
}
```

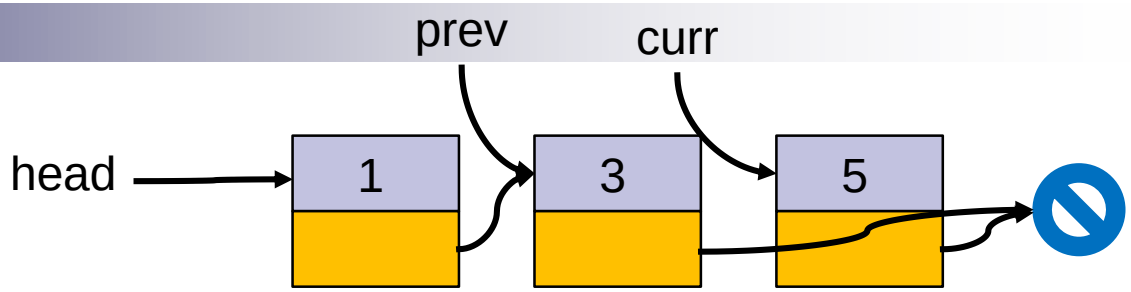
结点删除



- 空链表、头结点、其他情况(删除1)

```
Node * deleteList(Node *head, int value) {
    Node *curr = head;
    if (head == NULL) return head;  // 空链表
    if (head->data == data) {       // 非空链表，删除头结点
        head = head->next;          // 先连接后释放内存
        free(curr);
    } else {                        // 非空链表，删除中间或尾结点
        Node *prev = head;
        curr = head->next;
        while (curr != NULL && curr->data != value) { // 先判断后使用
            prev = curr;
            curr = curr->next;
        }
        if (curr != NULL) {         // 找到带删除的结点
            prev->next = curr->next;  // 先连接后释放内存
            free(curr);
        }
    } return head;
```

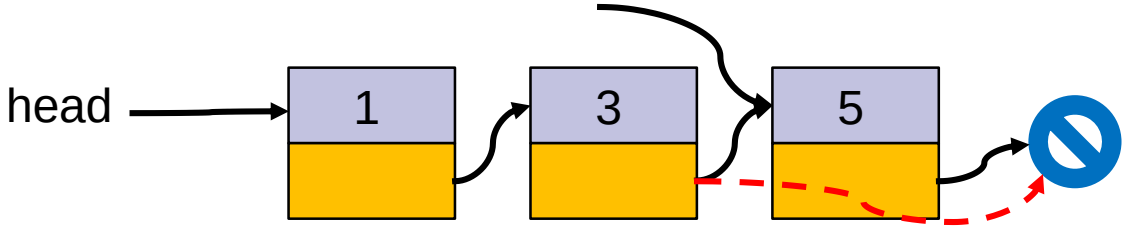
结点删除



- 空链表、头结点、其他情况(删除5)

```
Node * deleteList(Node *head, int value) {
    Node *curr = head;
    if (head == NULL) return head;  // 空链表
    if (head->data == data) {       // 非空链表，删除头结点
        head = head->next;          // 先连接后释放内存
        free(curr);
    } else {                        // 非空链表，删除中间或尾结点
        Node *prev = head;
        curr = head->next;
        while (curr != NULL && curr->data != value) { // 先判断后使用
            prev = curr;
            curr = curr->next;
        }
        if (curr != NULL) {         // 找到带删除的结点
            prev->next = curr->next;  // 先连接后释放内存
            free(curr);
        }
    } return head;
```

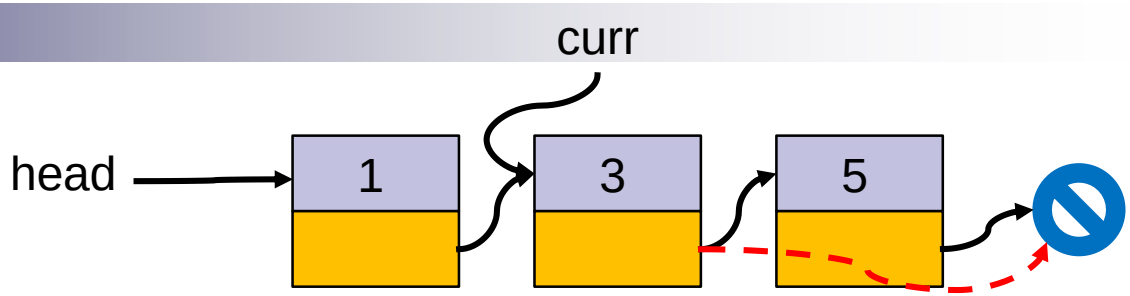
结点删除



- 空链表、头结点、其他情况(删除5)

```
Node * deleteList(Node *head, int value) {
    Node *curr = head;
    if (head == NULL) return head;  // 空链表
    if (head->data == data) {       // 非空链表，删除头结点
        head = head->next;          // 先连接后释放内存
        free(curr);
    } else {                        // 非空链表，删除中间或尾结点
        while (curr != NULL && curr->data != value) // 先判断后使用
            curr = curr->next;
        ?->next = curr->next;
        free(curr);
    }
    return head;
}
```

结点删除

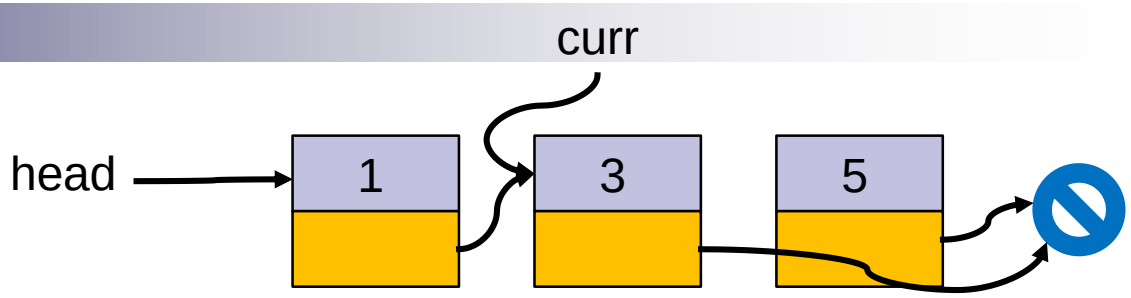


- 空链表、头结点、其他情况(删除5)

```
Node * deleteList(Node *head, int value) {
    Node *curr = head;
    if (head == NULL) return head;  // 空链表
    if (head->data == data) {       // 非空链表，删除头结点
        head = head->next;          // 先连接后释放内存
        free(curr);
    } else {                        // 非空链表，删除中间或尾结点
        while (curr->next != NULL && curr->next->data != value) // 先判断后使用
            curr = curr->next;
        free(curr->next);
        curr->next = curr->next->next;
    }
    return head;
}
```

✗ curr->next指向的结点内存已释放

结点删除



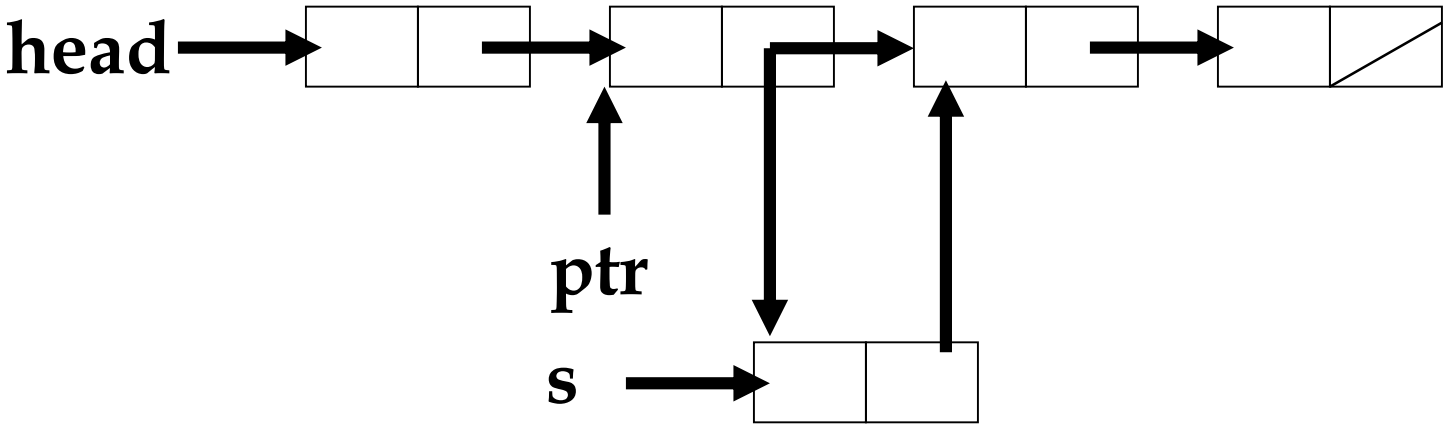
- 空链表、头结点、其他情况(删除5)

```
Node * deleteList(Node *head, int value) {
    Node *curr = head;
    if (head == NULL) return head;  // 空链表
    if (head->data == data) {       // 非空链表，删除头结点
        head = head->next;          // 先连接后释放内存
        free(curr);
    } else {                        // 非空链表，删除中间或尾结点
        while (curr->next != NULL && curr->next->data != value) // 先判断后使用
            curr = curr->next;
        curr->next = curr->next->next;
        free(curr->next);           // 先连接后释放内存
    }
    return head;
}
```

结点插入与结点删除 (双指针实现和单指针实现)

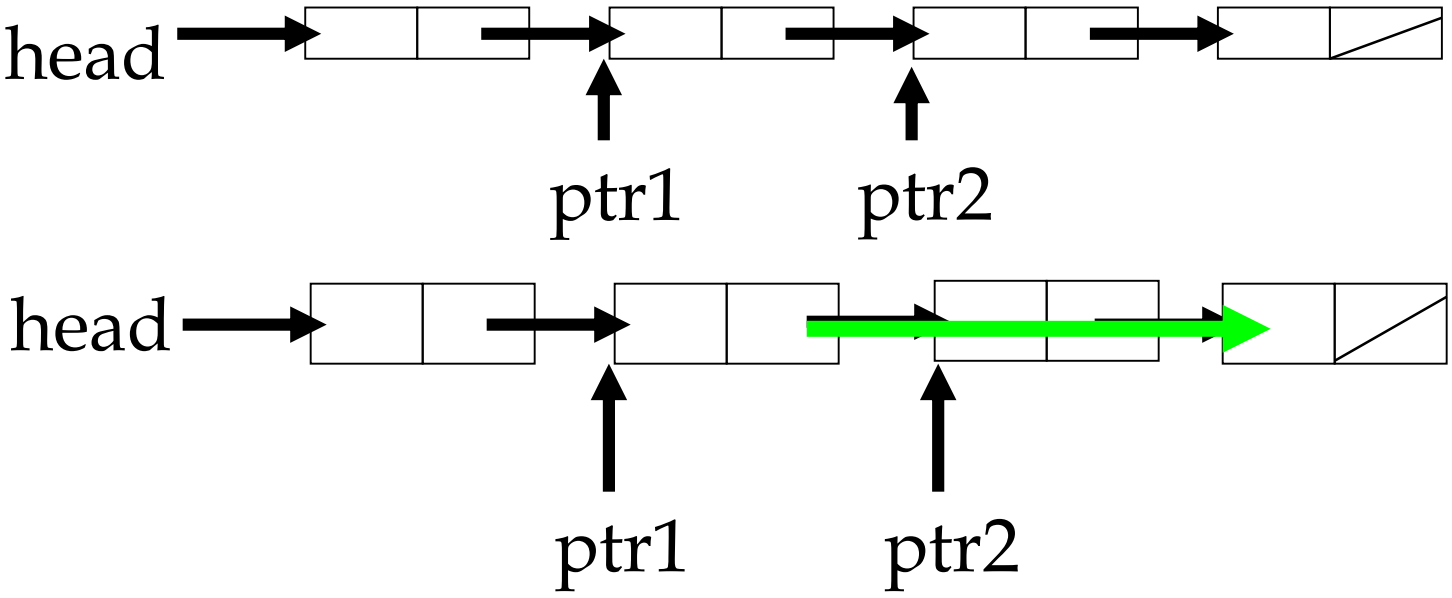
- 考虑四/五种情况/测试 (空链表，链表头、中、尾，不存在)
- 先连后断，先连后删
  (先保存信息，再删除，否则导致信息丢失)

插入结点 (图11.13)



- 先连: s->next = ptr->next;
- 后断: ptr->next = s;

删除结点 (图11.14)



- ptr2=ptr1->next;
- 先接:ptr1->next=ptr2->next;

ptr1->next = ptr1->next->next;

- 后删:free(ptr2);

链表的删除

- 链表的删除
  - free函数不会递归删除节点，需要逐个删除
  - 结点内存释放后，其内容可能失效

```
void deleteList(Node *head) {
    free(head);
}
```

```
void deleteList(Node *head) {
    while (head) {
        Node *node = head->next;
        free(head);
        head = node;
    }
}
```

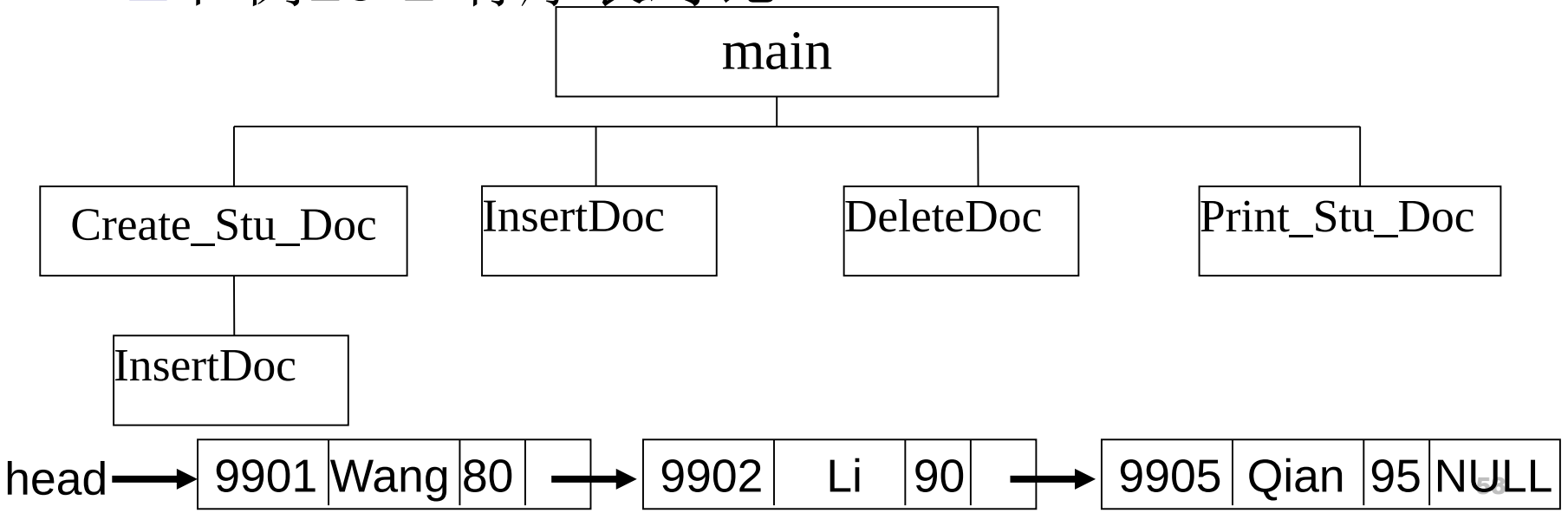
```
void deleteList(Node *head) {
    while (head) {
        free(head);
        head = head->next;
    }
}
```

专题二 链表

- 基础知识回顾
- 链表的概念 (11.3.2)
- 单向链表的常用操作 (11.3.3)
- 链表的应用 (11.3.1)
  - 学生信息成绩 (11.3.1)
  - 一元多项式及其运算 (数据结构的选择)
  - 猴子选大王 (用数组模拟链表，循环链表)

链表的应用1 - 学生成绩信息

- [例11-10] 建立一个学生成绩信息(包括学号、姓名、成绩)的单向链表，学生记录按学号由小到大顺序排列，要求实现对成绩信息的插入、修改、删除和遍历操作
  - 和例10-1 有序表对比



链表的应用1 - 学生成绩信息

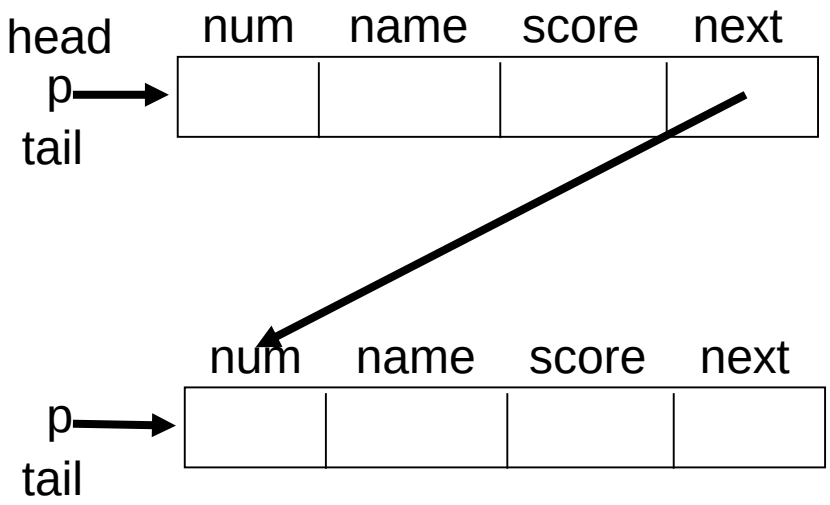
- 通常使用结构来定义单向链表结点的数据类型

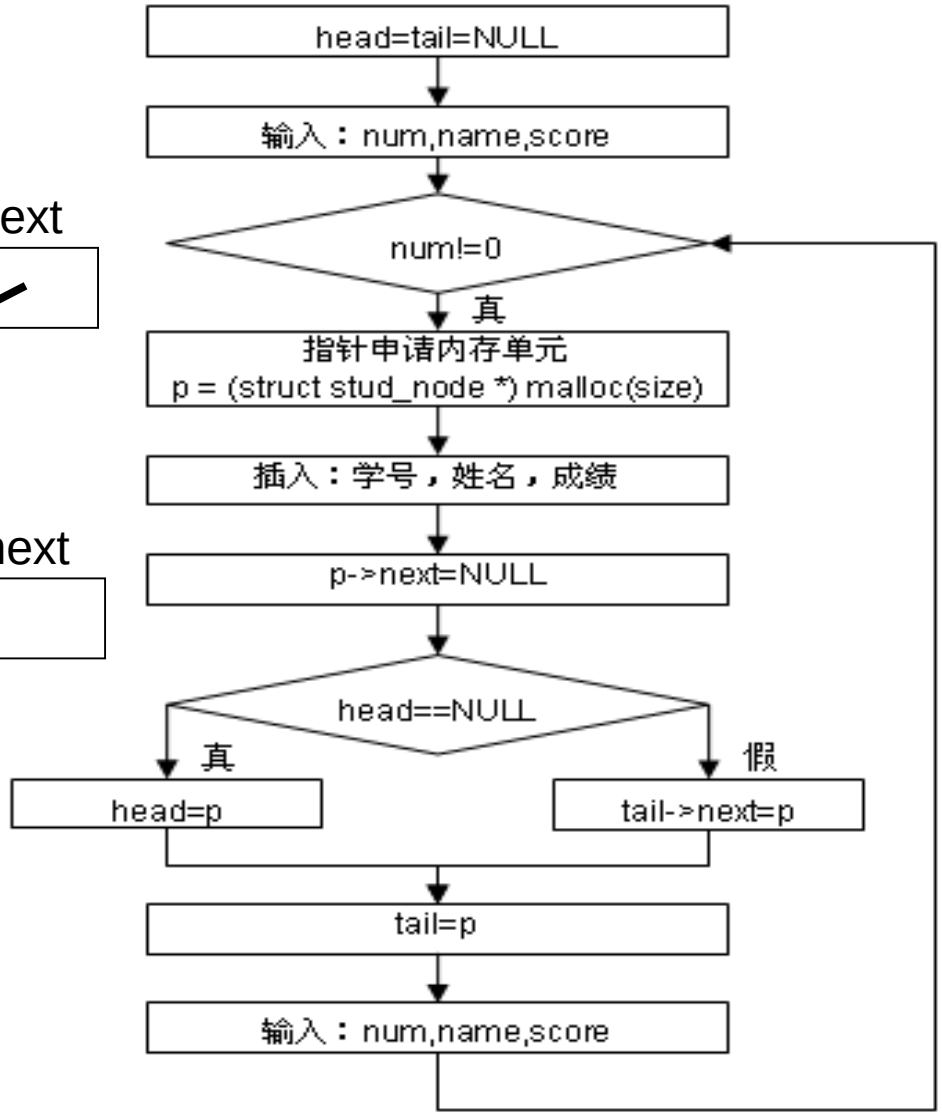
```
struct stud_node {
    int     num;
    char  name[20];
    int     score;
    struct stud_node *next;
};
```

- 功能
  - struct stud_node * Create_Stu_Doc(); /* 新建链表 */
  - struct stud_node * InsertDoc(struct stud_node * head, struct stud_node * stud); /* 插入 */
  - struct stud_node * DeleteDoc(struct stud_node * head, int num); /* 删除 */
  - void Print_Stu_Doc(struct stud_node * head); /* 遍历 */

链表的应用1 - 学生成绩信息

- 链表的建立





链表的应用1 - 学生成绩信息

- 链表的建立

```
head = tail = NULL;
scanf("%d%s%d", &num, name, &score);
while(num != 0) {
   p = (struct stud_node *) malloc(size);
   p->num = num;
   strcpy(p->name, name);
   p->score = score;
   p->next = NULL;
   if(head == NULL)
      head = p;
   else
      tail->next = p;
   tail = p;
   scanf("%d%s%d", &num, name, &score);
}
```

尾部插入

头部插入

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

例11-1 源程序

学生信息插入

```
struct stud_node *ptr ,*ptr1, *ptr2;
ptr2 = head;
ptr = stud;                  /* ptr指向待插入的新的学生记录结点 */
if(head == NULL) {           /* 原链表为空时的插入 */
    head = ptr;              /* 新插入结点成为头结点 */
    head->next = NULL;
} else {                     /* 原链表不为空时的插入 */
    while((ptr->num > ptr2->num) && (ptr2->next != NULL)) {
        ptr1 = ptr2;         /* ptr1, ptr2各后移一个结点 */
        ptr2 = ptr2->next;
    }
    if(ptr->num <= ptr2->num) { /* 在ptr1与ptr2之间插入新结点 */
        if(head == ptr2)
            head = ptr;
        else
            ptr1->next = ptr;
       ptr->next = ptr2;
    } else {                 /* 新插入结点成为尾结点 */
       ptr2->next = ptr;
       ptr->next = NULL;
    }
}
```

例11-1 源程序

学生信息删除

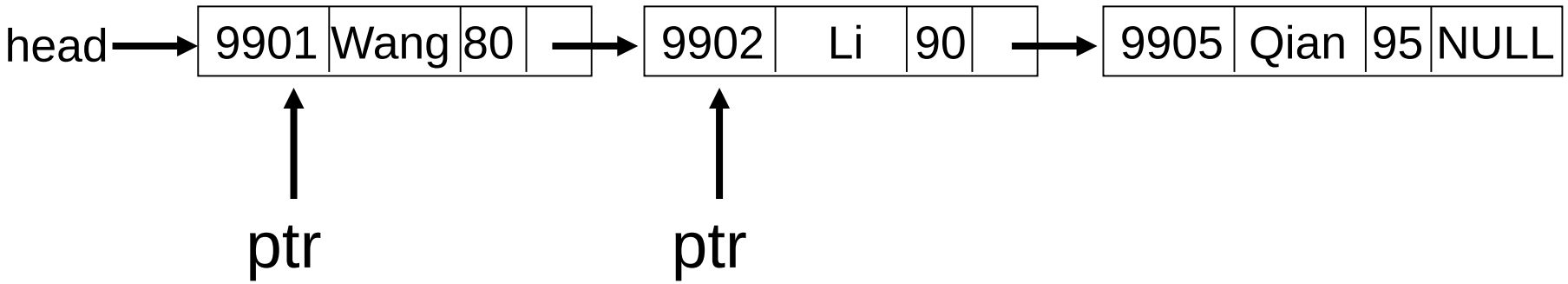
```
struct stud_node *ptr1, *ptr2;
/* 要被删除结点为表头结点 */
while(head != NULL && head->num == num) {
    ptr2 = head;
    head = head->next;
    free(ptr2);
}
if(head == NULL) /*链表空 */
    return NULL;
/* 要被删除结点为非表头结点 */
ptr1 = head;
ptr2 = head->next; /*从表头的下一个结点搜索所有符合删除要求的结点 */
while(ptr2 != NULL) {
    if(ptr2->num == num) { /* ptr2所指结点符合删除要求 */
        ptr1->next = ptr2->next;
        free(ptr2);
    } else {
        ptr1 = ptr2;          /* ptr1后移一个结点 */
    }
    ptr2 = ptr1->next;    /* ptr2指向ptr1的后一个结点 */
}
```

例11-1 源程序

学生信息打印

```
void Print_Stu_Doc(struct stud_node * head)
{
    struct stud_node * ptr;
    if(head == NULL) {
        printf("\nNo Records\n");
        return;
     }

     printf("\nThe Students' Records Are: \n");
     printf("    Num   Name   Score\n");
     for(ptr = head; ptr!=NULL; ptr = ptr->next)
        printf("%8d %20s  %6d \n", ptr->num, ptr->name, ptr->score);
}
```



链表的应用2 - 一元多项式

- 一元多项式 $f(x)=a_0+a_1x+\cdots+a_{n-1}x^{n-1}+a_nx^n$
  - 关键数据：项数n和每一项系数a_i
  - 主要运算：相加、相减、相乘等

- 方法1：采用有序表存储
  - 例如 $f(x)=4x^5-3x^2+1$

下标i	0	1	2	3	4	5	……
a[i]	1	0	–3	0	0	4	……

链表的应用2 - 一元多项式

- 方法1：采用有序表存储所有项
  - 例如 $f(x)=4x^5-3x^2+1$

下标i	0	1	2	3	4	5	……
a[i]	1	0	–3	0	0	4	……

- 方法2：采用有序表存储非零项
  - 例如 $P_1(x)=9x^{12}+15x^8+3x^2$　$P_2(x)=26x^{19}-4x^8-13x^6+82$

数组下标i	0	1	2	……
系数	9	15	3	–
指数	12	8	2	–

(a) $P_1(x)$

数组下标i	0	1	2	3	……
系数	26	–4	–13	82	–
指数	19	8	6	0	–

(b) $P_2(x)$

链表的应用2 - 一元多项式

- 方法2：采用有序表存储非零项，相加运算
  - 比较(9, 12)和(26, 19)，将(26, 19)移到结果多项式
  - 继续比较(9, 12)和(–4, 8)，将(9, 12)移到结果多项式
  - 比较(15, 8)和(–4, 8)，15 + (–4) = 11，不为0，将新的一项(11, 8)增加到结果多项式
  - 比较(3, 2)和(–13, 6), 将(–13, 6)移到结果多项式
  - 比较(3, 2)和(82, 0), 将(3, 2)移到结果多项式
  - 将(82, 0)直接移到结果多项式
  - 最后得到的结果多项式是：((26,19), (9,12), (11, 8), (–13, 6), (3, 2), (82, 0))

$$P_1(x) = 9x^{12} + 15x^8 + 3x^2$$

数组下标i	0	1	2	……
系数	9	15	3	–
指数	12	8	2	–

(a) $P_1(x)$

$$P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$$

数组下标i	0	1	2	3	……
系数	26	–4	–13	82	–
指数	19	8	6	0	–

(b) $P_2(x)$

链表的应用2 - 一元多项式

- 方法3：采用链表存储非零项
  - 每个链表结点存储多项式中的一个非零项，包括系数和指数两个数据域以及一个指针域

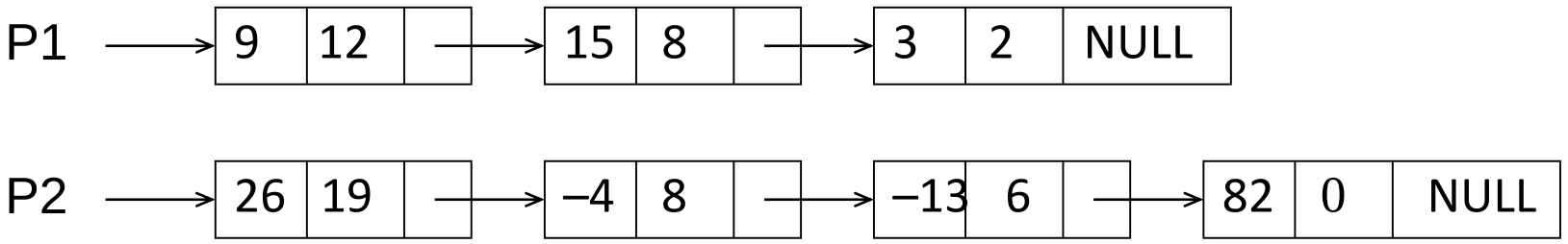
```
typedef struct PolyNode *Polynomial;
typedef struct PolyNode {
        int coef;
        int expon;
        Polynomial link;
}
```

例如：

$P_1(x) = 9x^{12} + 15x^8 + 3x^2$

$P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$

链表存储形式为：



链表的应用3 - 猴子选大王

- 一群猴子要选新猴王。新猴王的选择方法是：让n只候选猴子围成一圈，从某位置起顺序编号为1~n号。从第1号开始报数(从1到3)，凡报到3的猴子即退出圈子，接着又从紧邻的下一只猴子开始同样的报数。如此不断循环，最后剩下的一只猴子就选为猴王。请问是原来第几号猴子当选猴王？
- 输入一个正整数n (n <= 10000)，写一个程序来模拟这个过程，输出猴王的序号

序号	输入	输出
1	1	The king is monkey[1].
2	3	The king is monkey[2].
3	11	The king is monkey[7].
4	100	The king is monkey[91].

链表的应用3 - 猴子选大王

- 方法1：数组存储
  - int monkey[M]，标记是否在列

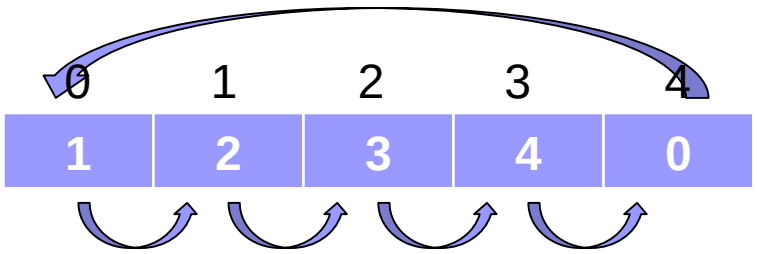
猴子选大王

```
int find_next(int index, int monkey[M], int n) {
    int next = (index + 1)%n;
    while (monkey[next] == 0)       /* 查找未出列的下一只猴子 */
        next = (next + 1)%n;
    return next;
}

int main( ) {
    int monkey[M];
    int i, n, index = 0;                                    /* index为当前报数的猴子 */
    scanf("%d", &n);                                        /* n为猴子数量 */
    for (i = 0; i < n; i++)
        monkey[i] = 1;                                      /* 所有猴子都在圈内*/
    for (i = 1; i < n; i++) {                               /* 共需出列n – 1只猴子 */
        index = find_next(index, monkey, n);                /* 报数为2的猴子 */
        index = find_next(index, monkey, n);                        /* 报数为3的猴子 */
        monkey[index] = 0;                                  /* 报数为3的猴子出列 */
        index = find_next(index, monkey, n);                /* 报数为1的猴子 */
     }
     printf("The king is monkey[%d].\n", index + 1);
     return 0;
}
```

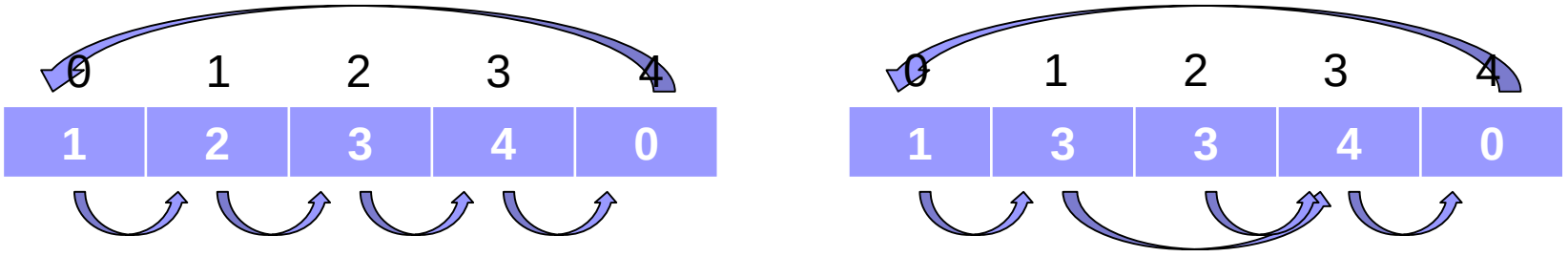
链表的应用3 - 猴子选大王

- 方法1：数组存储
  - int monkey[M]，标记是否在列
  - 与[例11-6]随机发牌的int temp[52]比较
  - find_next随着猴子不断出列，效率降低
- 方法2：数组模拟链表法
  - 把n只猴子用0~n-1编号，数组的下标表示猴子的编号，数组元素的值表示相邻下一只在圈子中的猴子编号。比如，n=5时，初始的数组



链表的应用3 - 猴子选大王

- 方法2：数组模拟链表法
  - 当2号猴子(M[2]，报数轮到3)退出圈子时，1号猴子的下一只相邻猴子就是3号猴子了，实现时只需一个赋值M[1]=M[2](即原来2号猴子的下一只相邻猴子成了1号猴子的下一只相邻猴子)



  - 优点：(1) 第i号猴子的下一只相邻猴子就是M[i]，不需要用一个循环去查找，(2) 不用当心数组M下标的访问会越界

猴子选大王

```
int find_next(int index, int monkey[], int n) {
    return monkey[index];
}

int main()  {
    int monkey[M];
    int i, n, prev, index = 0;                    /* index为当前报数的猴子 */
    scanf("%d", &n);                              /* n为猴子数量 */
    /* 下标表示猴子的编号，元素的值表示相邻下一只在圈子中的猴子编号 */
    for (i = 0; i < n - 1; i++)
        monkey[i] = i + 1;
    monkey[i] = 0;
    for (i = 1; i < n; i++) {                     /* 共需出列n – 1只猴子 */
        prev  = find_next(index, monkey, n);      /* 报数为2的猴子 */
        index = find_next(prev,  monkey, n);      /* 报数为3的猴子 */
        monkey[prev] = monkey[index];             /* 出列，更新下一猴子编号 */
        index = find_next(index, monkey, n);      /* 报数为1的猴子 */
     }
     printf("The king is monkey[%d].\n", index + 1);
     return 0;
}
```

链表的应用3 - 猴子选大王

/* 如果只需要给出猴王是谁，猴子选大王问题是经典的约瑟夫环问题。用以下递推算法： 令f[i]表示i个猴子时最后的大王，则需推出f[n]

f[1]=0;

f[i] = (f[i-1]+3)%i; /* 容易推广到报数 m */

最后输出f[n]+1 即可。

时间复杂度为n，无需数组，空间复杂度也很低，程序简单

*/

```
int main()
{
    int n, m, i, s=0;
    scanf("%d", &n);
    m = 3;
    for (i = 2; i <= n; i++)
        s = (s + m) % i;
    printf ("The King is mondey[%d].\n", s+1);
}
```

链表的应用3 - 猴子选大王改进

- 一群猴子要选新猴王。新猴王的选择方法是：让n只候选猴子围成一圈，从某位置起顺序编号为1~n号。每只猴子预先设定一个数(或称定数)，用最后一只猴子的定数d，从第一只猴子开始报数，报到d的猴子即退出圈子；当某只猴子退出时，就用它的定数决定它后面的第几只猴子将在下次退出。如此不断循环，最后剩下的一只猴子就选为猴王。请输出猴子退出圈子的次序以及当选的猴王编号

序号	输入	输出
1	1	The king is monkey[1].
2	5 3 2 1 4 3	3 4 5 1 The king is monkey[2].
3	11 3 3 3 3 3 3 3 3 3 3 3	3 6 9 1 5 10 4 11 8 2 The king is monkey[7].
4	100。。。余略	略

链表的应用3 - 猴子选大王改进

- 方法1：数组存储
  - int monkey[M]，标记是否在列
  - 与[例11-6]随机发牌的int temp[52]比较
  - find_next随着猴子不断出列，效率降低
- 方法2：数组模拟链表法
  - 把n只猴子用0~n-1编号，数组的下标表示猴子的编号，数组元素的值表示相邻下一只在圈子中的猴子编号
- 方法3：动态链表存储

猴子选大王

```
typedef struct node {      /* 这个结构类型包括三个域 */
    int number;            /* 猴子的编号 */
    int mydata;            /* 猴子的定数 */
    struct node *next;     /* 指向下一只猴子的指针 */
} linklist;
linklist *CreateCircle(int n) {
    int i;
    linklist *head,*p,*s;
    head = (linklist *) malloc(sizeof(linklist));          /*首节点创建*/
    p = head;   p->number = 1;
    scanf("%d", &p->mydata);      /* 定数(正整数)，确定下一只出局的猴子*/
    p->next = NULL;
    for (i = 1; i < n; i++) {                /* 链表创建*/
        s = (linklist*) malloc(sizeof(linklist));
        s->number = i + 1;
        scanf("%d",&s->mydata);    /* 定数(正整数)，确定下一只出局的猴子*/
        p->next = s;
        p = s;
    }
    p->next = head;                          /*链表首尾相接构成循环链表*/
    return p;        /* 返回最后一只猴子的指针，因为它指向第一只猴子 */
}
```

```
int KingOfMonkey(linklist *head, int n) {
    linklist *p = head;                  /* head指向最后一只猴子结点 */
    int i, j, steps = p->mydata;         /*用最后一只猴子的定数开始 */
    for (j = 1; j < n; j++) {            /*重复该过程n - 1次*/
        for (i = 1; i < steps; i++)
            p = p->next;          /* 将p所指的下一个节点删除, 删除前取其定数 */
        steps = p->next->mydata;
        printf("%d ", p->next->number);
        linklist *temp = p->next;        /* 删除p所指的下一个结点 */
        p->next = p->next->next;
        free(temp);
    }
    return p->number;
}
int main() {
    linklist *head;
    int n;
    printf("请输入猴子的总数和每只猴子的定数(必须是正整数)：");
    scanf("%d", &n);
    head = CreateCircle(n); /*创建单循环链表，返回最后一个结点的指针 */
    printf("\nThe king is monkey[%d].\n", KingOfMonkey(head, n));
    return 0; }
```

链表动态申请的内存是否都已释放？
能否用方法2实现该改进的问题？
如何检测链表是否存在环？

专题二 链表

辩证思维：不同的实现方案的辩证选择
数据结构的选择：数组 vs. 链表
链表创建、插入和删除的不同实现方案
链表的实现：数组模拟 vs. 动态链表

- 基础知识回顾 (结构、动态内存分配)
- 链表的概念 (11.3.2)
- 单向链表的常用操作 (11.3.3)
  - 链表的创建 (顺序/逆序)、遍历与释放
  - 结点插入 (先连后断)与结点删除 (先连后删)
    - 单指针实现 (curr->next->value)和双指针实现 (prev)
- 链表的应用 (11.3.1)
  - 学生信息成绩 (11.3.1)
  - 一元多项式及其运算 (数据结构的选择)
  - 猴子选大王 (用数组模拟链表，循环链表)