链表(Linked List)详解

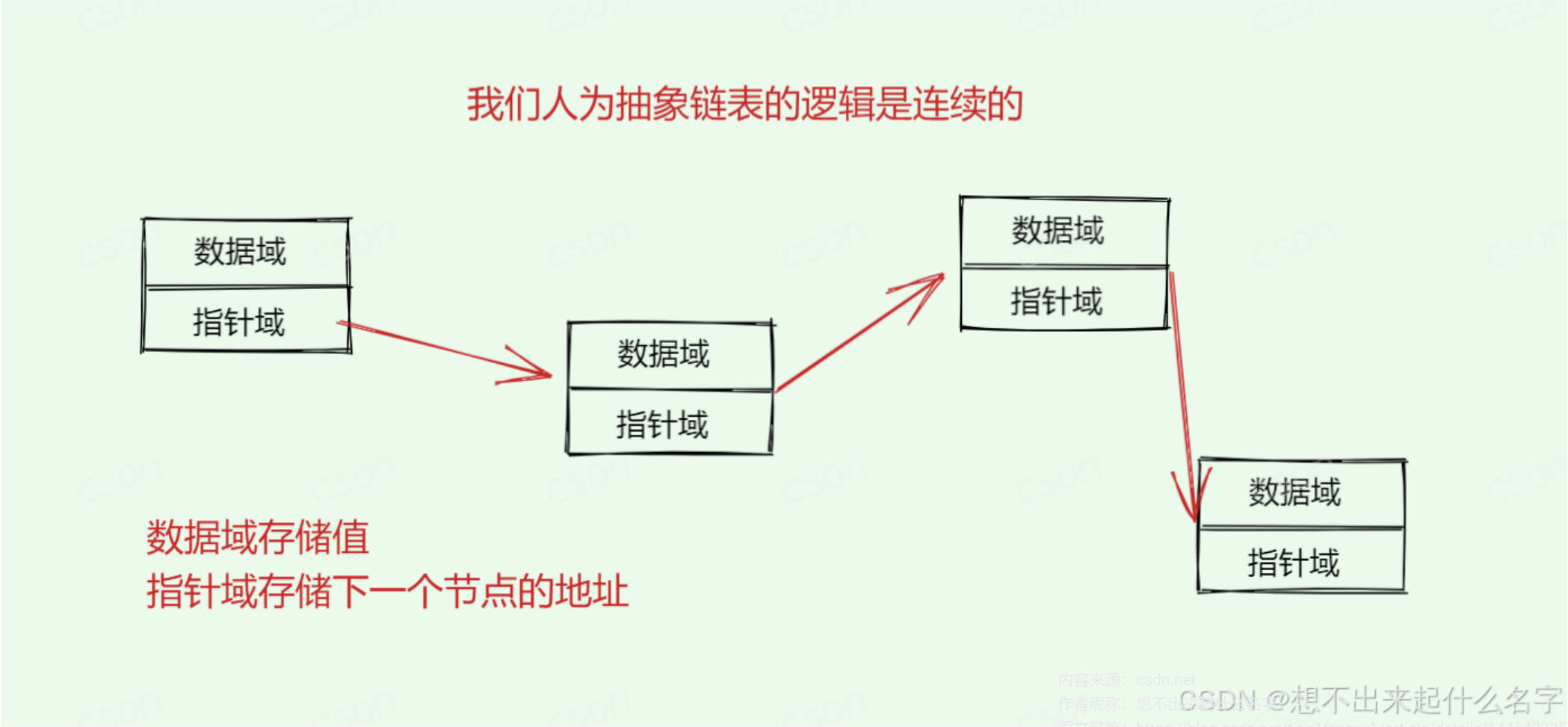
想不出来起什么名字 已于 2025-01-03 21:09:39 修改

引言

上一篇我们提到 顺序表 这个 数据结构，它的底层是数组

而这一篇，我们学习一个新的数据结构——链表

和顺序表不同，链表在**逻辑上是连续的，但是在物理上（内存分布）可能是不连续的**



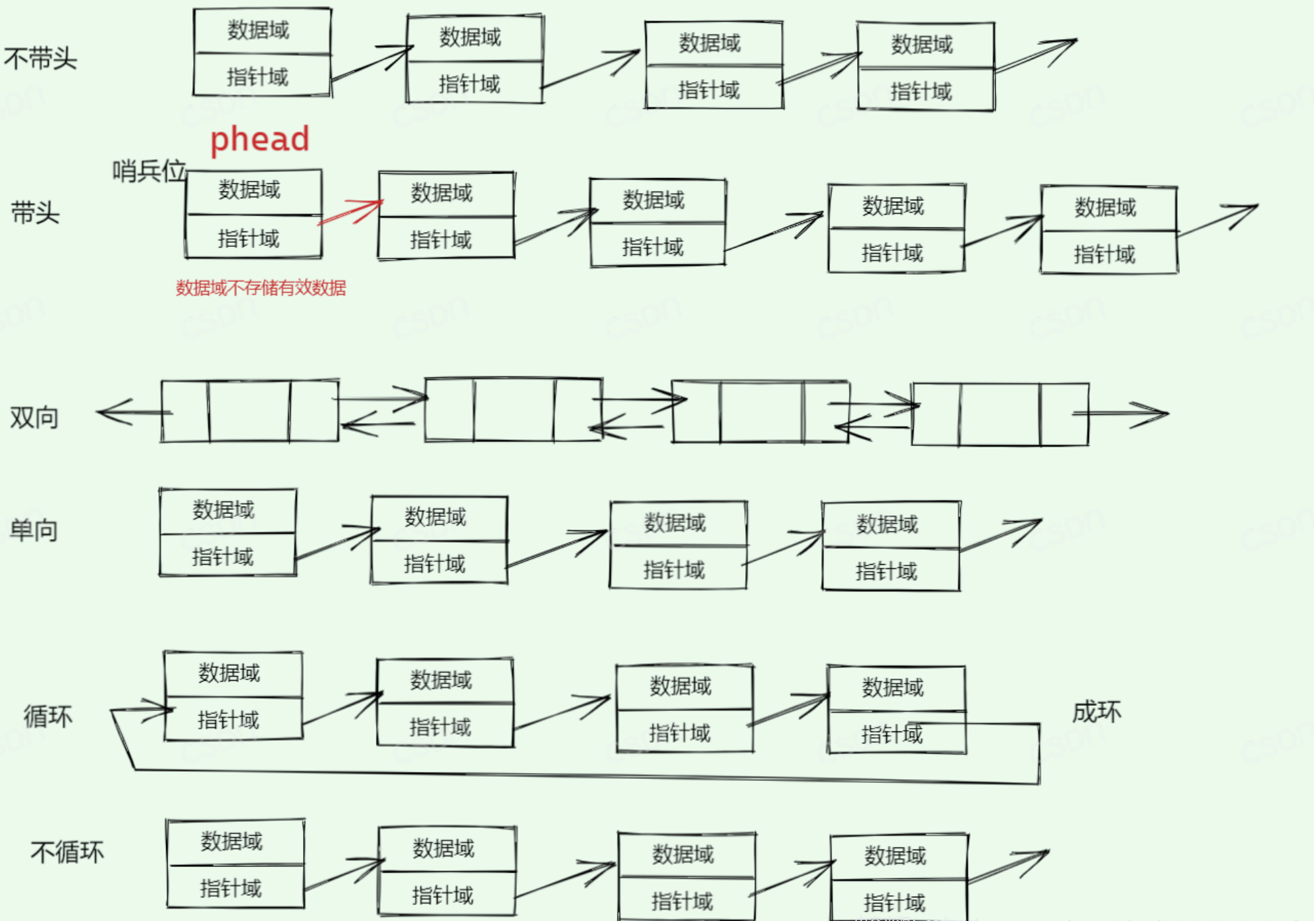
我们可以将链表想象成火车，一节一节车厢就是链表的节点，而车厢里的内容就是数据域，链接每个车厢的车钩就相当于指针

链表的每个节点是我们自己**在堆上开辟**的，在堆上开辟的空间，操作系统是按照一定的策略开辟的，两次开辟的空间地址可能连续，也可能不连续，所以在物理结构上是不连续的

这里就可以总结出链表最大的特点：**数据做岛屿，指针做桥梁**，这样分散的数据就有了联系

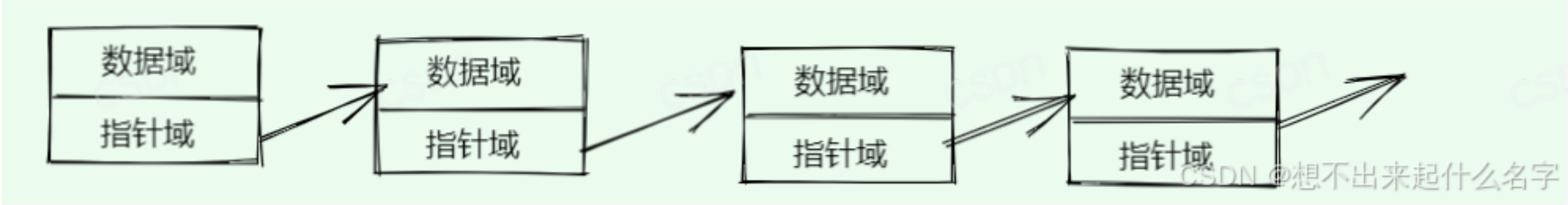
链表的分类

基于带头与否，循环与否，双向与否，我们可以组合出$2^3 = 8$种链表

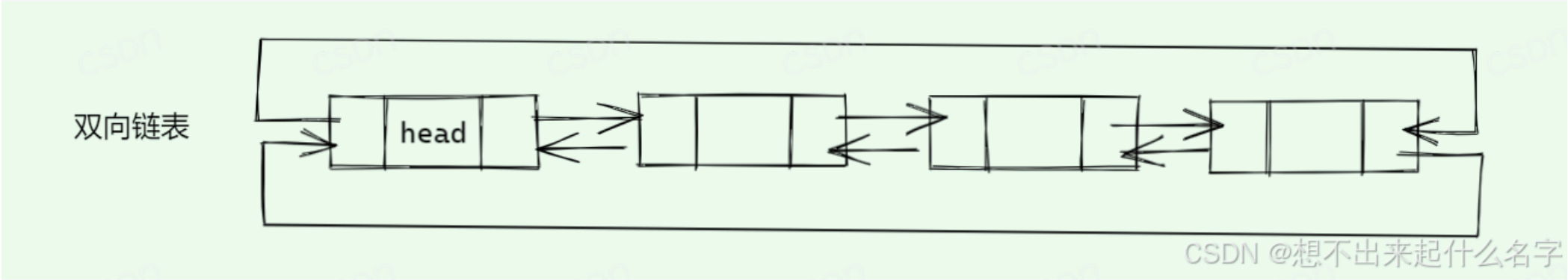


虽然有这么多的链表的结构，但是我们实际中最常用还是两种结构：

不带头单向不循环链表：即**单链表**



带头双向循环链表：即**双向链表**



单链表：结构简单，一般不会单独用来存数据。实际中更多是作为其他数据结构的子结构

双向链表：结构最复杂，一般用在单独存储数据。实际中使用的 链表数据结构，都是带头双向循环链表

单链表的实现

定义一个单链表

```
//不带头不循环单向链表
typedef int SLDataType;
typedef struct SListNdoe {
    SLDataType data;//数据域
    struct Listnode* next;//指针域
}SLTNode;
```

代码解读

定义接口

```
//链表的销毁
void SLTDestory(SLTNode** pplist);

//插入操作
void SLTPushHead(SLTNode** pplist, SLDataType input);
void SLTPushBack(SLTNode** pplist, SLDataType input);
//指定位置之前插入元素
void SLTPushPosFront(SLTNode** pplist, SLTNode* pos,SLDataType input);
//指定位置之后插入元素
void SLTPushPosAfter(SLTNode* pos,SLDataType input);

//删除操作
void SLTPopBack(SLTNode** pplist);
void SLTPoplist(SLTNode** pplist);
//删除指定位置之后的一个元素
void SLTPopPosAfter(SLTNode* pos);
//删除指定位置元素
void SLTPopPos(SLTNode** pplist, SLTNode* pos);

//查找元素
SLTNode* SLTFind(SLTNode* plist, SLDataType target);

//链表打印
void SLTPrint(SLTNode* plist);

//创建新的节点
SLTNode* SLTBuyNode(SLDataType input);
```

代码解读

插入操作

在插入节点之前需要定义新的接口，开辟一个节点的空间

```
SLTNode* SLTBuyNode(SLDataType input) {
```

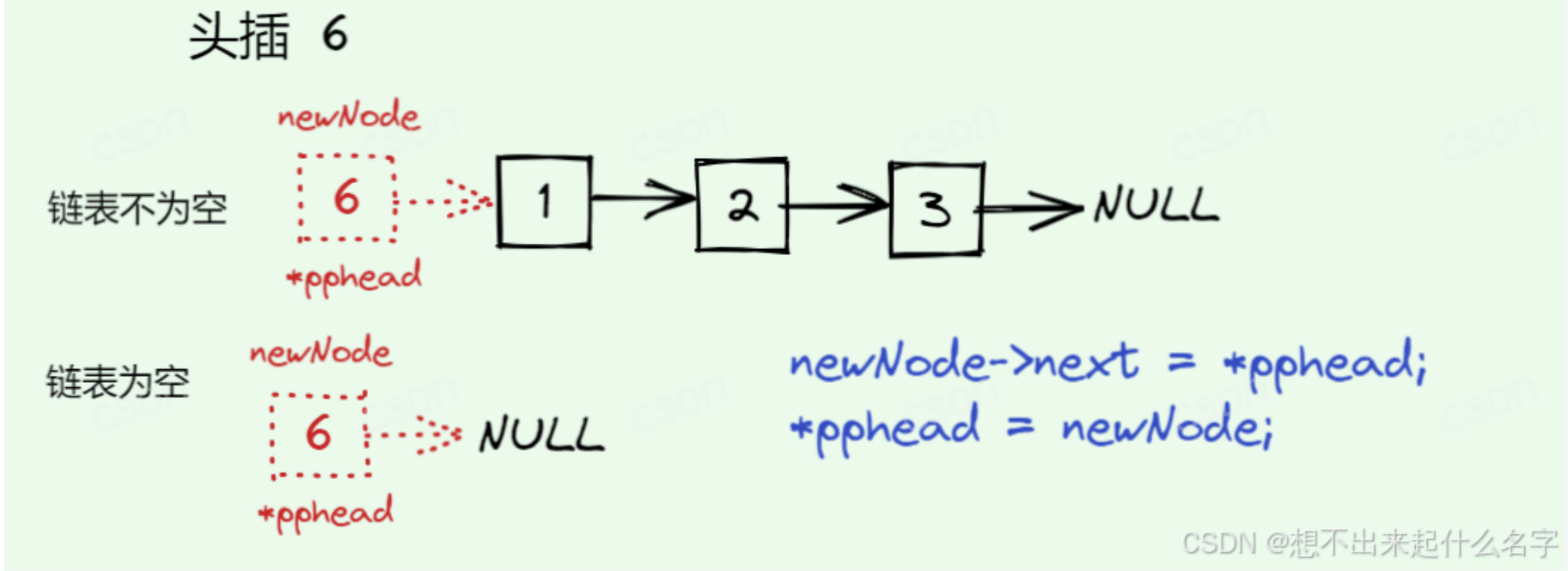
```
    SLTNode* newNode = (SLTNode*)malloc(sizeof(SLTNode));
    if (newNode == NULL) {
        perror("malloc() err!");
        return;
    }
    newNode->data = input;
    newNode->next = NULL;

    return newNode;
}
```

代码解读

头插

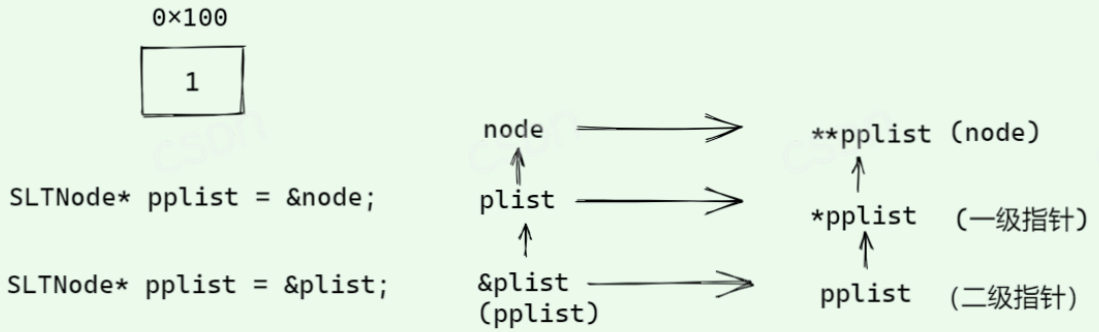
```
void SLTPushHead(SLTNode** pplist, SLDataType input) {
    //检查二级指针是否有效
    assert(pplist);

    SLTNode* newNode = SLTBuyNode(input);

    newNode->next = *pplist;
    *pplist = newNode;
}
```

代码解读



为什么要定义二级指针?

如果函数参数定义成一级指针，那么应改成SLTPushHead(plist，1)了，这时候就是传值操作，SLTPushHead()函数栈帧结束后，值也随之销毁



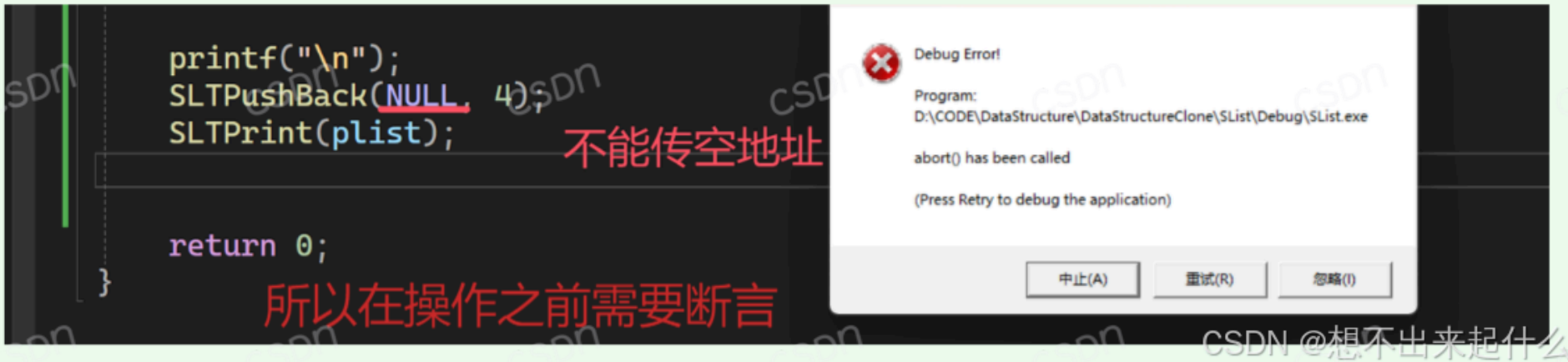
一级指针的地址 二级指针来接收

```c
int main() {
    SLTNode* plist = NULL;
    SLTPushHead(&plist,1);
    SLTPushHead(&plist,2);
    SLTPushHead(&plist,3);
    SLTPushHead(&plist,4);
```

```c
void SLTPushHead(SLTNode** pplist, SLDataType input) {
    //检查二级指针是否有效
    assert(pplist);

    SLTNode* newNode = SLTBuyNode(input);

    newNode->next = *pplist;
    *pplist = newNode;
}
```

如果函数参数定义成一级指针，那么应改成SLTPushHead(plist, 1)了，
这时候就是传值操作，SLTPushHead()函数栈帧结束后，值也随之销毁

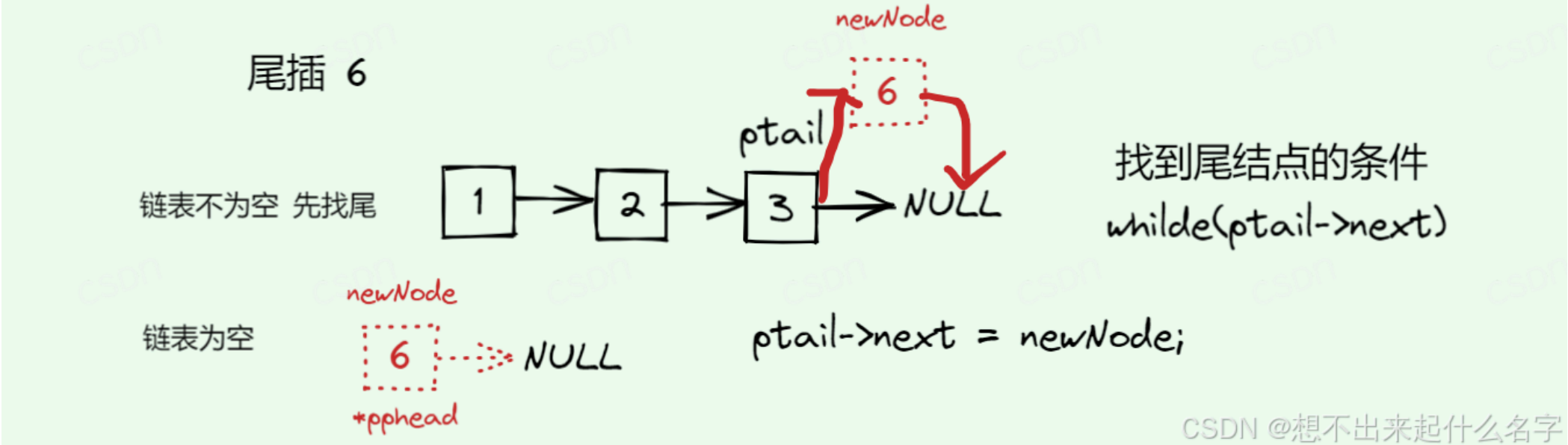
还有需要注意的地方，**我们需要操作指针时，先要验证指针是否有效，不然会出现内存泄漏的情况**



尾插

```
1  //尾插
2  void SLTPushBack(SLTNode** pplist, SLDataType input) {
3      //检查二级指针是否有效
4      assert(pplist);
5
6      SLTNode* newNode = SLTBuyNode(input);
7      SLTNode* ptail = *pplist;
8      //链表为空 新节点就是尾节点
9      if (*pplist == NULL) {
10         ptail = newNode;
11         return;
12     }
13     //链表不为空 找尾节点
14     while (ptail->next) {
15         ptail = ptail->next;
16     }
17     ptail->next = newNode;
18 }
```

代码解读



指定位置需要查找 查找指定节点

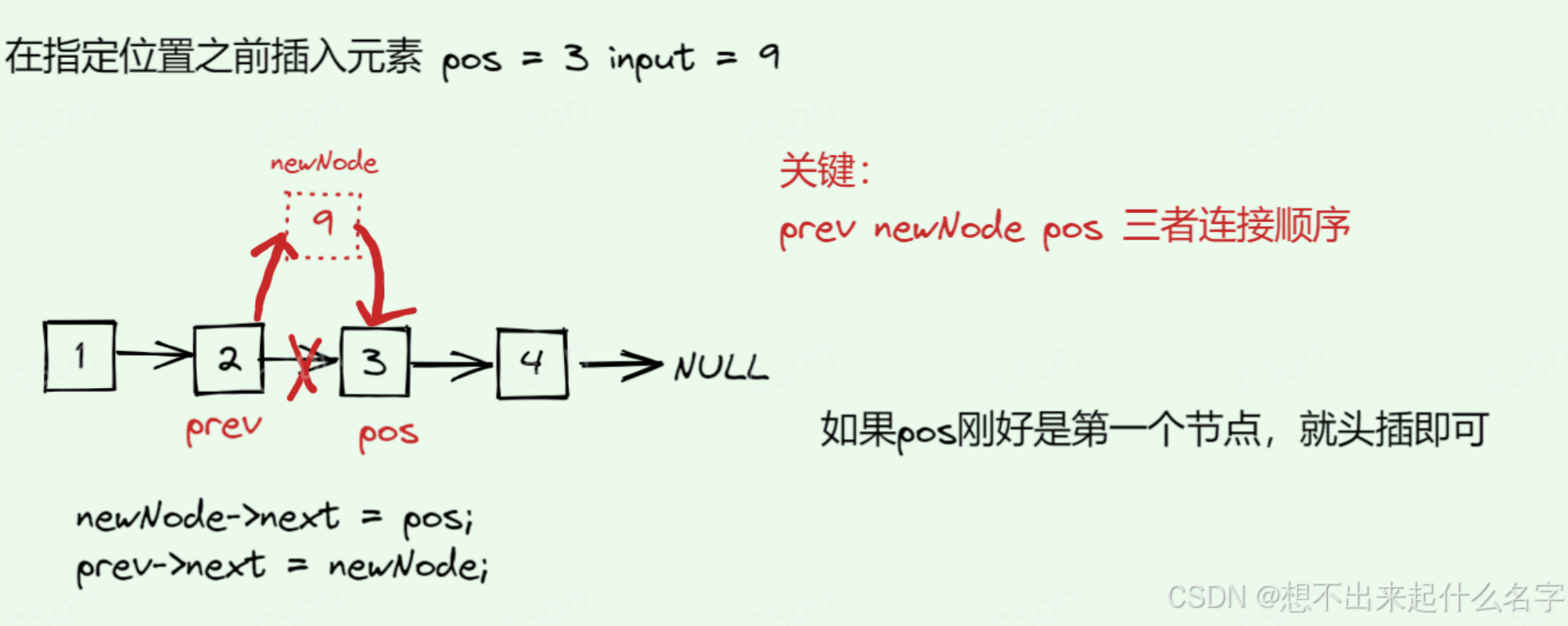
```
SLTNode* SLTFind(SLTNode* plist, SLDataType target) {
    assert(plist);
    SLTNode* pcur = plist;
    while (pcur) {
        if (pcur->data == target) {
            return pcur;
        }
        pcur = pcur->next;
    }
    return NULL;
}
```

代码解读

指定位置之前插入节点

```c
//指定位置之前插入元素
void SLTPushPosFront(SLTNode** pplist, SLTNode* pos, SLDataType input) {
    assert(pplist);
    assert(pos);
    assert(*pplist);

    //pos刚好是第一个节点 头插
    if (pos == *pplist) {
        SLTPushHead(pplist, input);
        return;
    }

    //pos不是第一个节点 找pos的前驱节点
    SLTNode* newNode = SLTBuyNode(input);
    SLTNode* prev = *pplist;
    while (prev != NULL && prev->next != pos) {
        prev = prev->next;
    }
//添加判断，看是否找到了pos的前驱节点
    if (prev == NULL) {
//说明没找到pos，可能pos不属于该链表，可以根据具体需求进行错误处理，这里简单释放刚申请的节点并返回
        free(newNode);
        return;
    }
    newNode->next = pos;
    prev->next = newNode;
}
```

代码解读

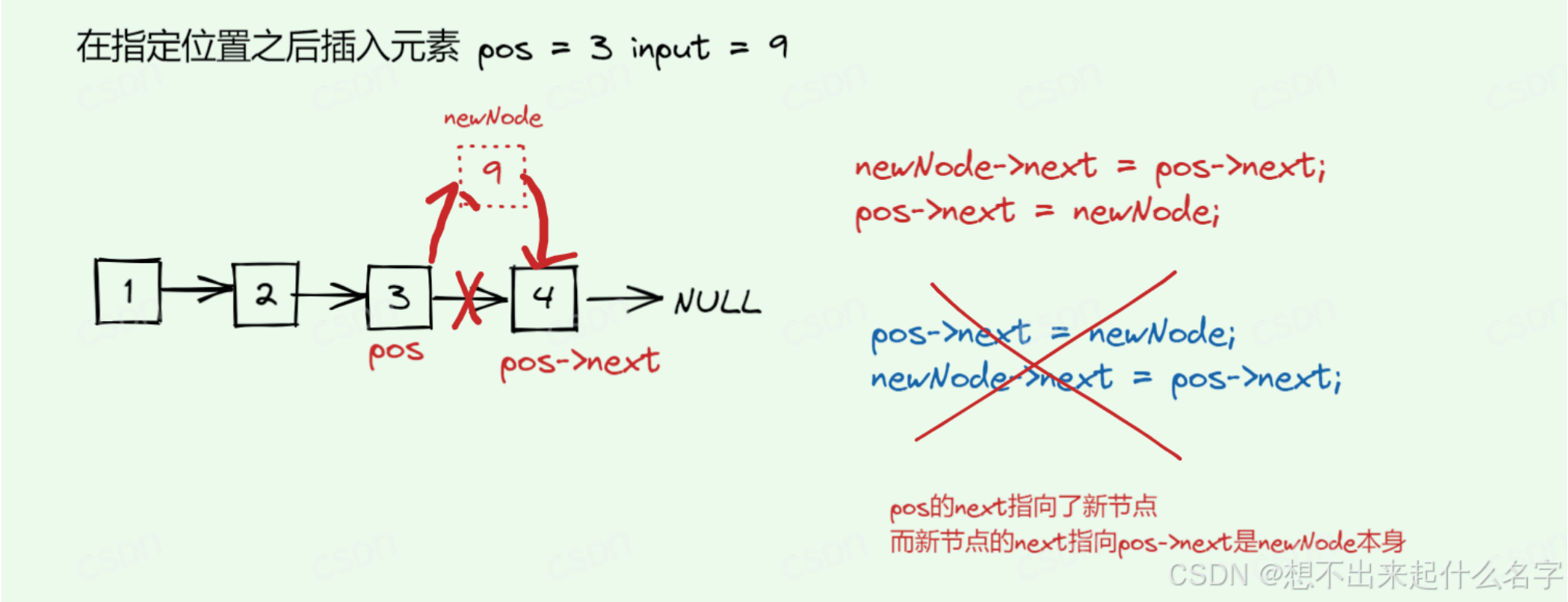


指定位置之后插入节点

一定要注意指针链接顺序！！！！！

```
1  //指定位置之后插入元素
2  void SLTPushPosAfter(SLTNode* pos, SLDataType input) {
3      //只需要找到pos的位置即可 pos之前的节点无需考虑
4      assert(pos);
5      SLTNode* newNode = SLTBuyNode(input);
6
7      newNode->next = pos->next;
8      pos->next = newNode;
9  }
```

代码解读



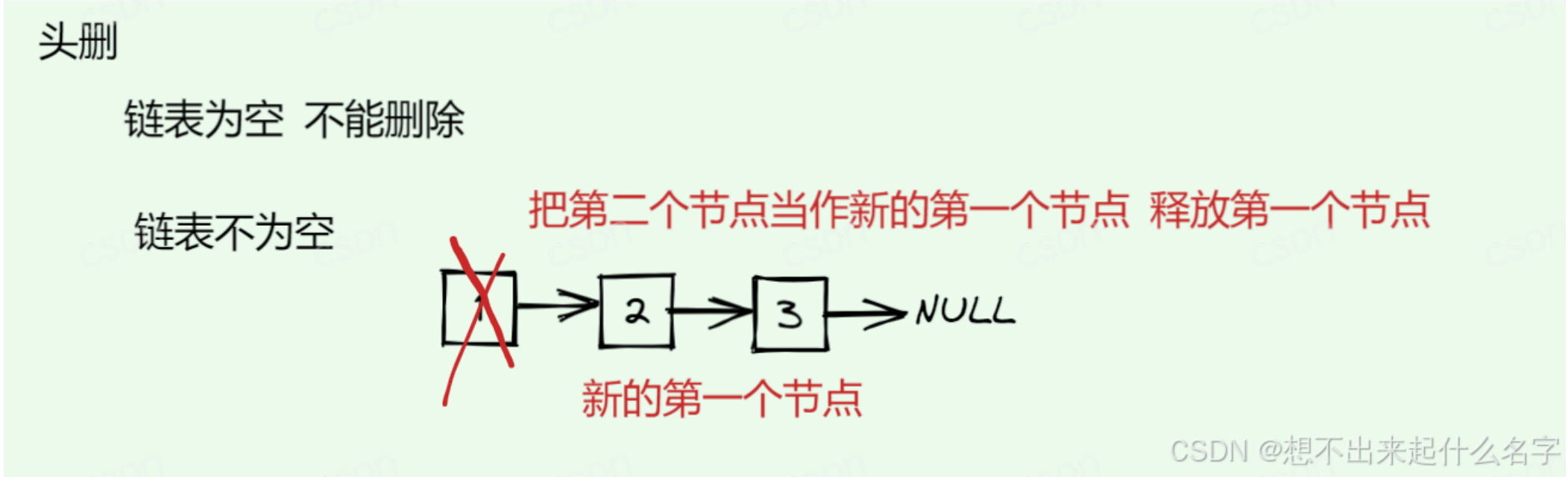
删除操作

头删

```
//头删
void SLTPoplist(SLTNode** pplist) {
    //检查二级指针是否有效
    assert(pplist);
    //链表不能为空 空链表不能执行删除操作
    assert(*pplist);
    //把第二个节点当作第一个节点 释放第一个节点
    SLTNode* next = (*pplist)->next;
    free(*pplist);
    *pplist = next;
```

```
}
```

代码解读



尾删

```c
//尾删
void SLTPopBack(SLTNode** pplist) {
    //检查二级指针是否有效
    assert(pplist);
    //链表不能为空 空链表不能执行删除操作
    assert(*pplist);

    //链表不为空
    // 只有一个节点
    if ((*pplist)->next == NULL) {
        free(*pplist);
        *pplist = NULL;
        return;
    }
    //有多个节点
```

```
    //找尾 保存前驱节点
    SLTNode* ptail = *pplist;
    SLTNode* prev = NULL;
    while (ptail->next) {
        prev = ptail;
        ptail = ptail->next;
    }
    //销毁尾节点
    prev->next = NULL;
    free(ptail);
    ptail = NULL;
}
```

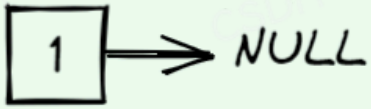
代码解读

尾删

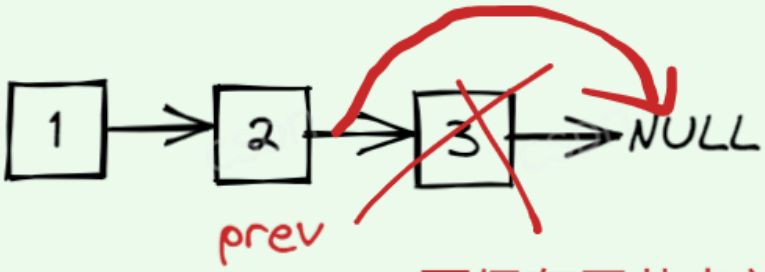
链表为空 不能删除

链表不为空

链表只有一个节点



链表有多个节点



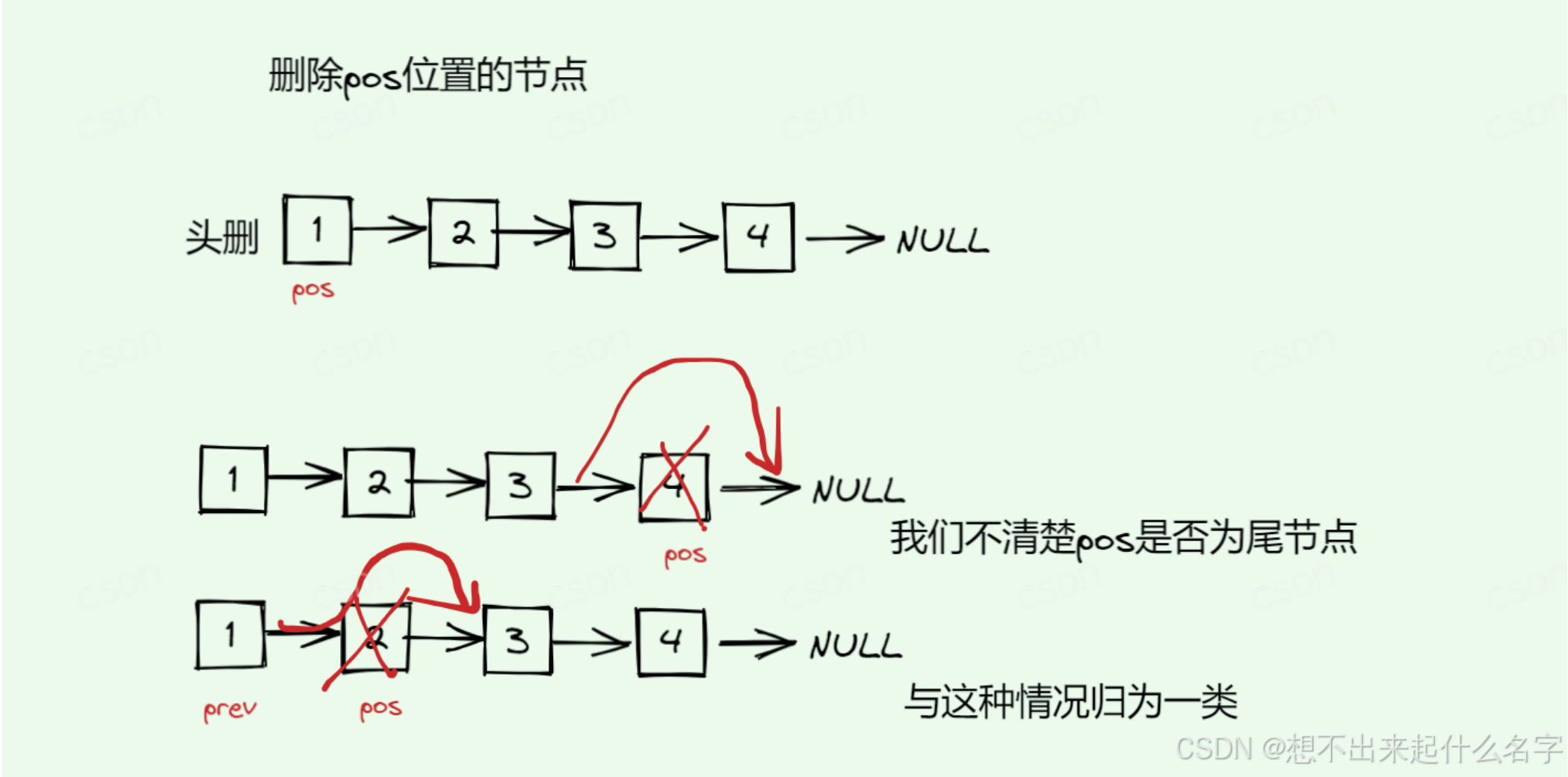
要保存尾节点前的前驱节点

删除指定位置节点

```c
//删除指定位置元素
void SLTPopPos(SLTNode** pplist, SLTNode* pos) {
    assert(pplist);
    assert(pos);
    assert(*pplist);

    //pos刚好在第一个节点位置
    if (pos == *pplist) {
        //头删
```

```
        SLTPoplist(pplist);11         return;
    }
    SLTNode* prev = *pplist;
    while (prev != NULL && prev->next != pos) {
        prev = prev->next;
    }
    prev->next = pos->next;
    free(pos);
    pos = NULL;
```

代码解读

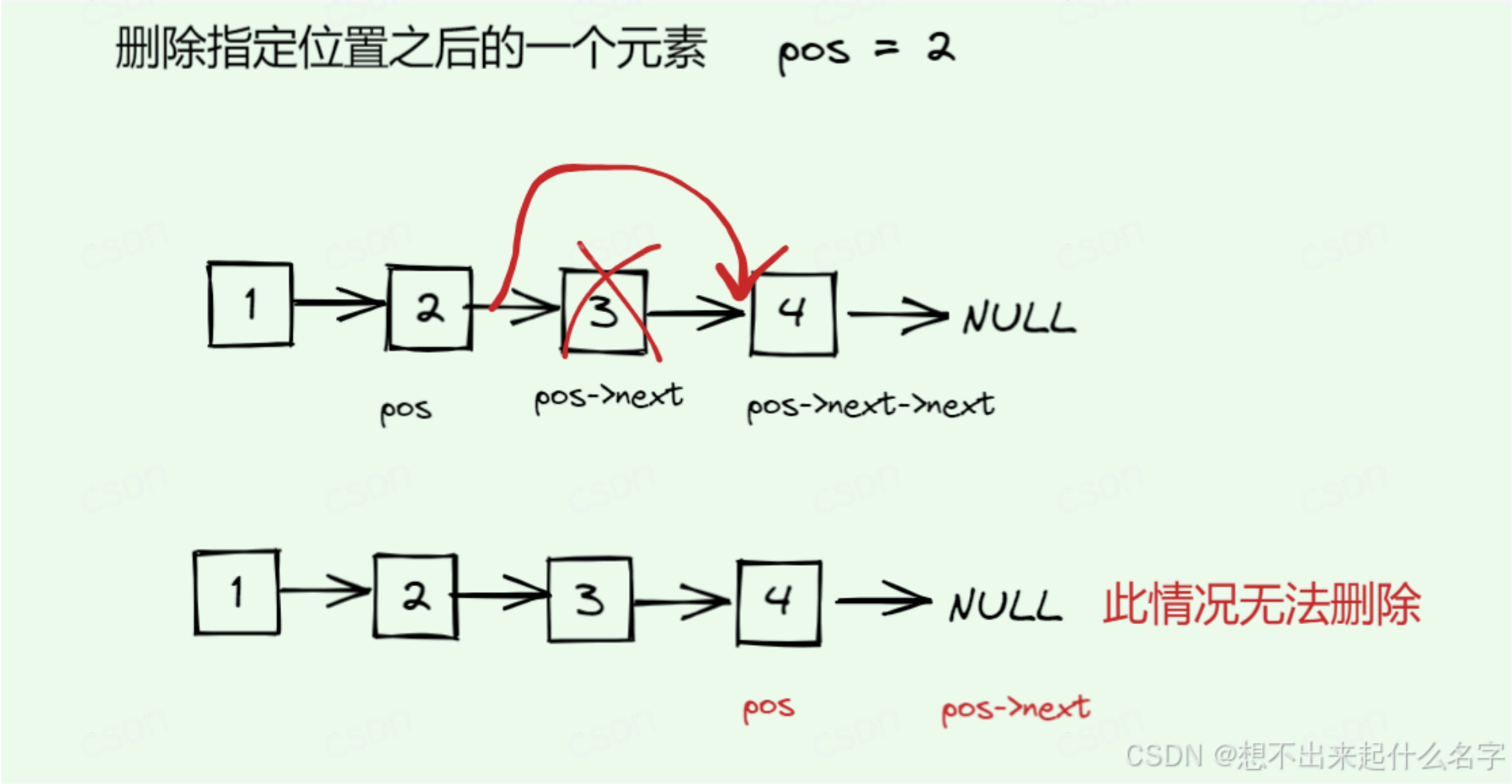


删除指定位置之后的节点

```c
//删除指定位置之后的一个元素
void SLTPopPosAfter(SLTNode* pos) {
    assert(pos);
    assert(pos->next);

    SLTNode* del = pos->next;
    pos->next =del->next;
    free(del);
```

```
    del = NULL;
}
```

代码解读



其他操作

销毁链表

```
//链表的销毁
void SLTDestory(SLTNode** pplist) {
```

```
    assert(pplist); 4      assert(*pplist);

    SLTNode* del = *pplist;
    while (del) {
        SLTNode* pcur = del;
        free(pcur);
        del = del->next;
    }
    *pplist = NULL;
}
```

代码解读

打印链表

```
void SLTPrint(SLTNode* plist) {
    SLTNode* pcur = plist;
    while (pcur)
    {
        printf("%d->", pcur->data);
        pcur = pcur->next;
    }
    printf("NULL\n");
}
```

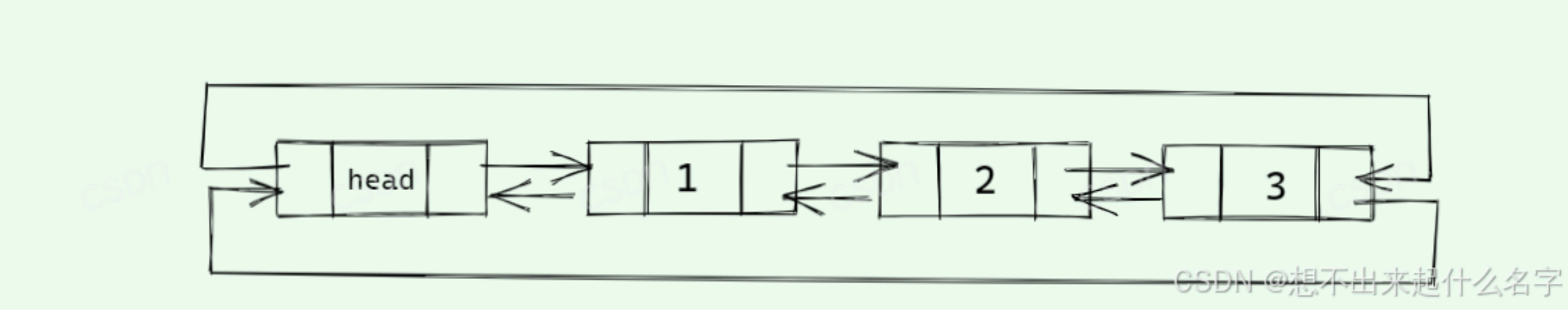
代码解读

双向链表

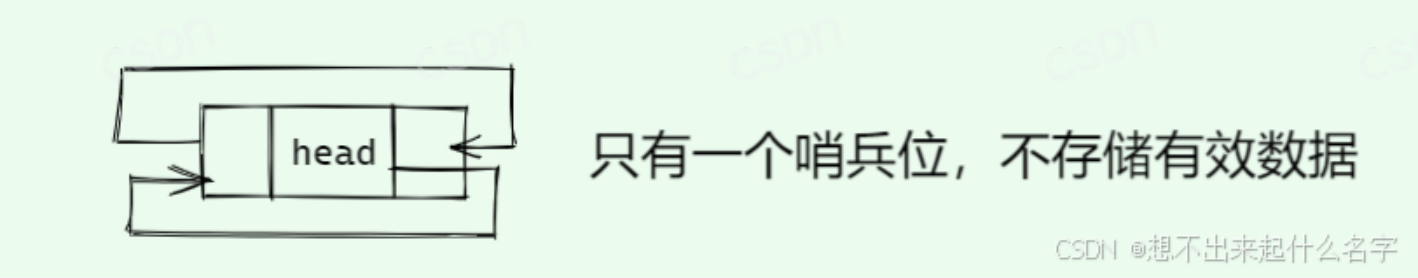
相关定义

> **先定义一个双向链表:**
> **typedef int LTDataType;**
> **typedef struct List {**
> **LTDataType data;**

```
    struct List* next;//指向后继节点
    struct List* prev;//指向前驱节点
}LTNode;
```



双向链表为空的情况——



定义接口

参数设计，要一级指针还是二级指针?
一级指针，因为我们现在有一个哨兵位可以找到链表所有节点
二级指针是为了解引用一次操作链表，没有哨兵位，就需要解引用二级指针来操作链表
而有了哨兵位，有了天然的可操作链表的优势

```
//创建新的节点
ListNode* ListNewNode(LTDataType input);
// 初始化链表
ListNode* ListInit();
```

```
// 双向链表销毁
void ListDestory(ListNode* plist);
// 双向链表打印
void ListPrint(ListNode* plist);
// 双向链表尾插
void ListPushBack(ListNode* plist, LTDataType input);
// 双向链表尾删
void ListPopBack(ListNode* plist);
// 双向链表头插
void ListPushFront(ListNode* plist, LTDataType input);
// 双向链表头删
void ListPopFront(ListNode* plist);
// 双向链表查找
ListNode* ListFind(ListNode* plist, LTDataType input);
// 双向链表在pos的前面进行插入
void ListInsert(ListNode* pos, LTDataType iinput);
// 双向链表删除pos位置的结点
void ListErase(ListNode* pos);
```

代码解读

插入操作

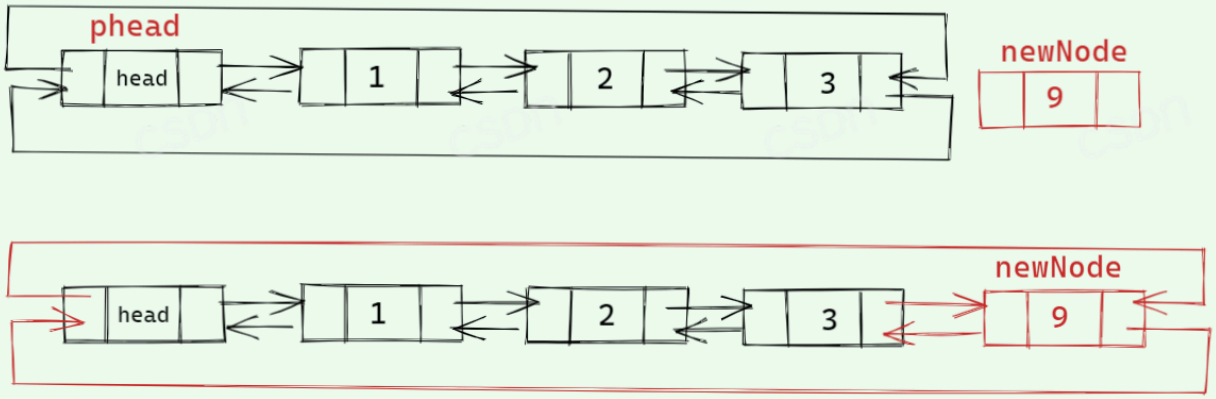
尾插

```
void ListPushBack(ListNode* plist, LTDataType input) {
    assert(plist);
    ListNode* newNode = ListNewNode(input);
    ListNode* pcur = plist->next;

    newNode->next = plist;
    newNode->prev = plist->prev;

    plist->prev->next = newNode;
    plist->prev = newNode;

```

```
}
```

代码解读

尾插9



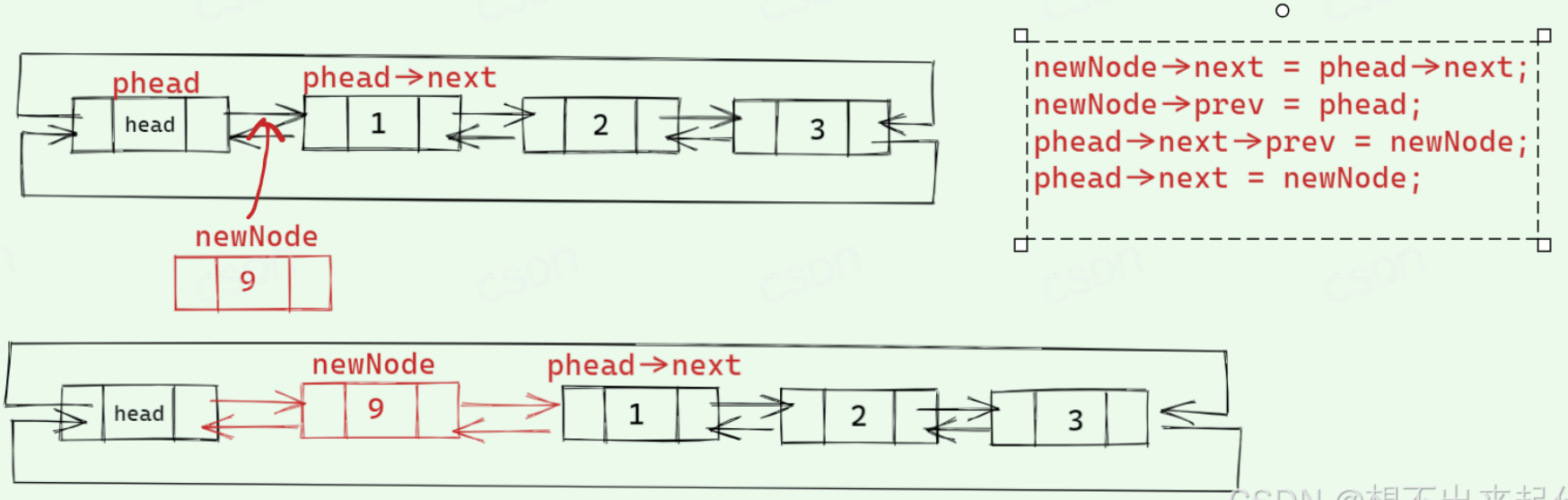

```
newNode->next = phead;
newNode->prev = phead->prev;
phead->prev->next = newNode;
phead->prev = newNode;
```


头插

```
// 双向链表头插
void ListPushFront(ListNode* plist, LTDataType input) {
    assert(plist);

    ListNode* newNode = ListNewNode(input);

    newNode->next = plist->next;
    newNode->prev = plist;

    plist->next->prev = newNode;
    plist->next = newNode;
}
```

代码解读

头插9 在哨兵位之后插入新节点

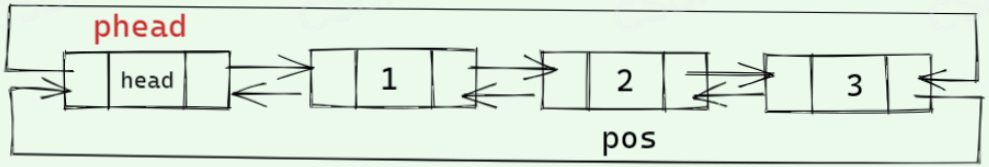


指定位置之前插入（先要找到pos的位置）

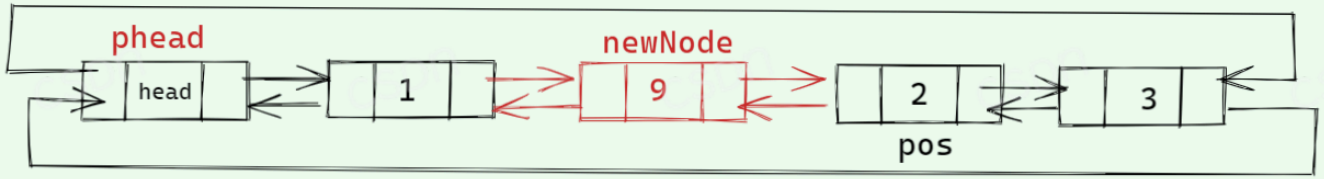
```
// 双向链表在pos的前面进行插入
void ListInsert(ListNode* pos, LTDataType input) {
    assert(pos);

    ListNode* newNode = ListNewNode(input);

    newNode->next = pos;
    newNode->prev = pos->prev;

    pos->prev->next = newNode;//这两个顺序不能错
    pos->prev = newNode;
}
```

代码解读

在pos之前插入9




```
newNode->next = pos;
newNode->prev = pos->prev;

pos->prev->next = newNode;//这两个顺序不能错
pos->prev = newNode;
```



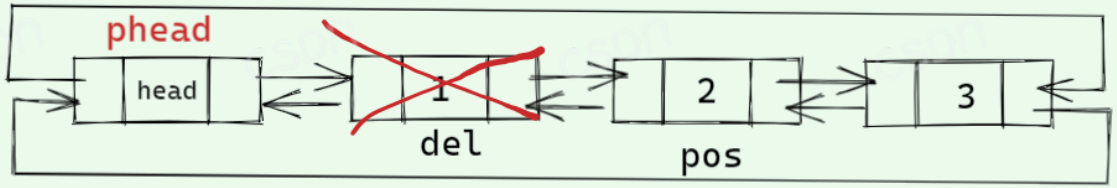
删除操作

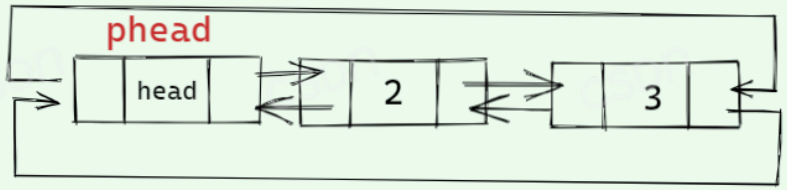
头删

```c
void ListPopFront(ListNode* plist) {
    assert(plist->next != plist);//链表不能为空

    ListNode* del = plist->next;
    ListNode* next = del->next;

    next->prev = plist;
    plist->next = next;//顺序不能错

    free(del);
    del = NULL;
}
```

代码解读

头删





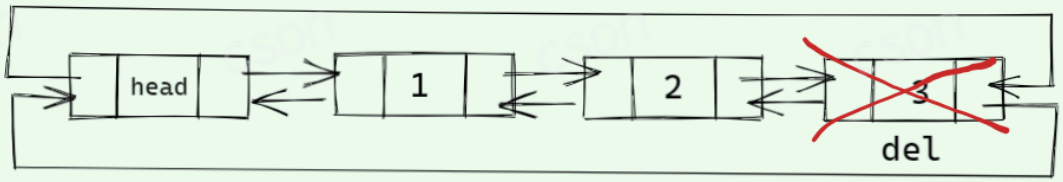

```
ListNode* del = plist->next;
ListNode* next = del->next;

next->prev = plist;
plist->next = next;//顺序不能错
```

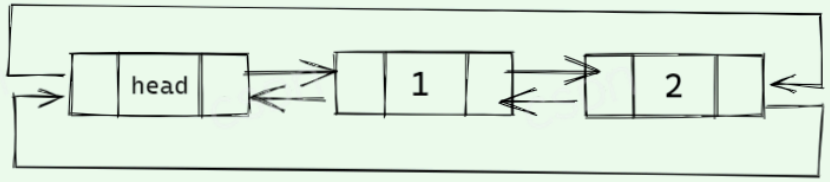

尾删

```
1  // 双向链表尾删
2  void ListPopBack(ListNode* plist) {
3      assert(plist->prev != plist);
4
5      ListNode* del = plist->prev;
6      ListNode* prev = del->prev;
7
8      prev->next = plist;
9      plist->prev = prev;
10
11     free(del);
12     del = NULL;
13 }
```

代码解读

尾删





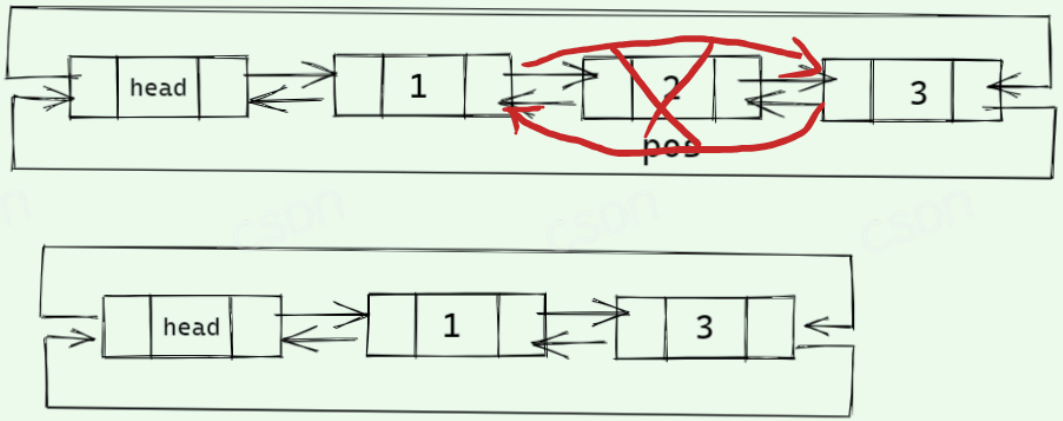

```
ListNode* del = plist->prev;
ListNode* prev = del->prev;

prev->next = plist;
plist->prev = prev;//顺序不能错
```


删除指定位置（先要找到pos的位置）

```
1  // 双向链表删除pos位置的结点
2  void ListErase(ListNode* pos) {
3      assert(pos);
4
5      pos->prev->next = pos->next;
6      pos->next->prev = pos->prev;
7
8      free(pos);
9      pos = NULL;
10 }
```

代码解读

删除指定位置




```
pos->prev->next = pos->next;
 pos->next->prev = pos->prev;
```


其他操作

创建新节点

```
1  //创建新的节点
2  ListNode* ListNewNode(LTDataType input) {
3      ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
4      if (newNode == NULL) {
5          perror("malloc() err!");
6          return 1;
7      }
8      newNode->data = input;
9      newNode->next = newNode->prev = newNode;
10 }
```

代码解读

查找节点

```
// 双向链表查找
ListNode* ListFind(ListNode* plist, LTDataType target) {
    assert(plist);

    ListNode* pcur = plist->next;

    while (pcur != plist) {
        if (pcur->data == target) {
            return pcur;
        }
        pcur = pcur->next;
    }
    return NULL;
}
```

代码解读

打印链表

```
// 双向链表打印
void ListPrint(ListNode* plist) {
    assert(plist);

    ListNode* pcur = plist->next;
    while (pcur != plist) {
        printf("%d->", pcur->data);
        pcur = pcur->next;
    }
    printf("\n");
}
```

代码解读

销毁链表

```
// 双向链表销毁
void ListDestory(ListNode* plist) {
    assert(plist);
```

```
    ListNode* pcur = plist->next;
    while (pcur != plist) {
        ListNode* next = pcur->next;
        free(pcur);
        pcur = next;
    }
    //只剩下哨兵位了
    free(plist);
    plist = NULL;
}
```

代码解读

链表和顺序表的对比

	顺序表	链表
存储空间	数组，连续存储	结构体，不一定连续存储
是否支持随机访问	支持，访问效率：O(1)	不支持：O(n)
任意位置插入或删除节点	可能需要挪动数据，效率低O(n)	只需修改指针指向
应用场景	元素高效存储+频繁访问	任意位置插入和删除频繁
缓存利用率	高	低

总结

- 理解链表的现实意义
- 链表的具体分类
- 单链表的增删查改
- 双向链表的增删查改
- 顺序表和链表的区别具体应用场景