顺序表的问题及思考

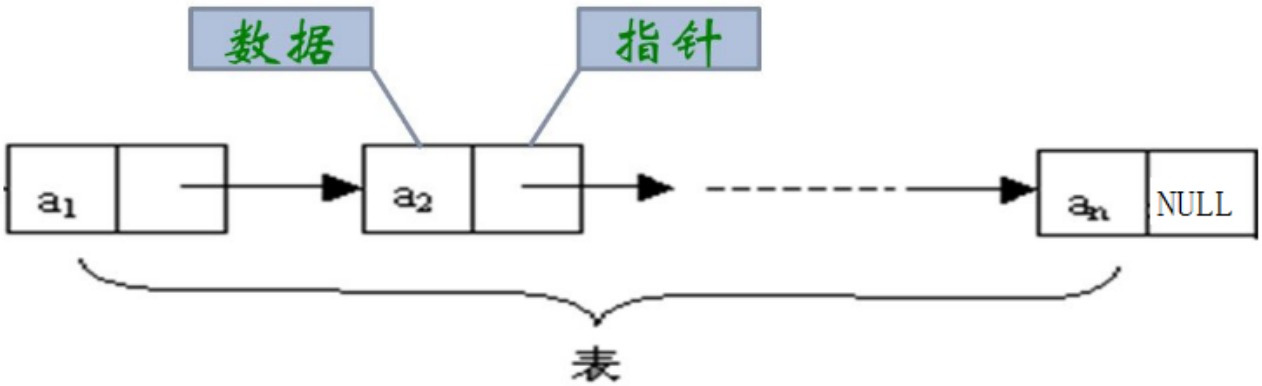
问题：

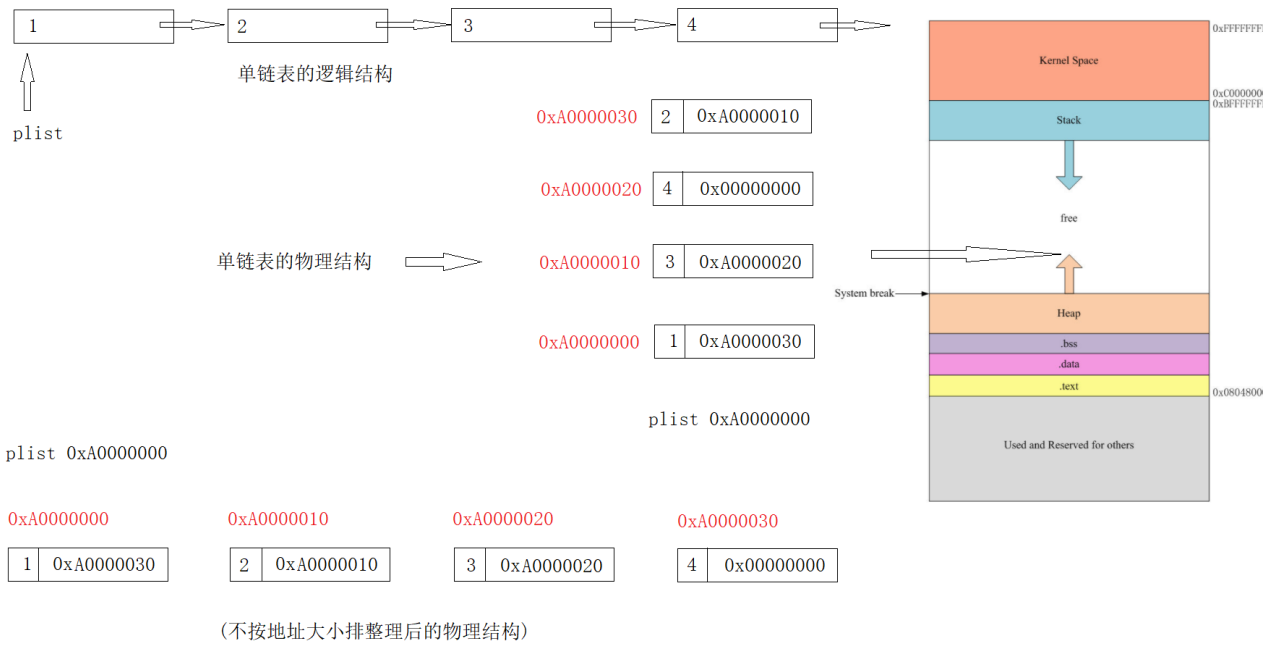
1. 中间/头部的插入删除，时间复杂度为O(N)
2. 增容需要申请新空间，拷贝数据，释放旧空间。会有不小的消耗。
3. 增容一般是呈2倍的增长，势必会有一定的空间浪费。例如当前容量为100，满了以后增容到200，我们再继续插入了5个数据，后面没有数据插入了，那么就浪费了95个数据空间。

思考：

如何解决以上问题呢？下面给出了链表的结构来看看。

概念：链表是一种**物理存储结构上非连续**、非顺序的存储结构，数据元素的**逻辑顺序**是通过链表中的**指针链接**次序实现的 。





（不按地址大小排整理后的物理结构）

实际中要实现的链表的结构非常多样，以下情况组合起来就有8种链表结构：

1. 单向、双向

单链表的实现：

```
// 1、无头+单向+非循环链表增删查改实现
typedef int SLTDateType;
typedef struct SListNode
{
    SLTDateType data;
    struct SListNode* next;
}SListNode;
// 动态申请一个节点
SListNode* BuySListNode(SLTDateType x);
// 单链表打印
void SListPrint(SListNode* plist);
// 单链表尾插
void SListPushBack(SListNode** pplist, SLTDateType x);
// 单链表的头插
void SListPushFront(SListNode** pplist, SLTDateType x);
// 单链表的尾删
void SListPopBack(SListNode** pplist);
// 单链表头删
void SListPopFront(SListNode** pplist);
// 单链表查找
SListNode* SListFind(SListNode* plist, SLTDateType x);
// 单链表在pos位置之前插入x
void SListInsert(SListNode** pplist, SListNode* pos, SLTDateType x);
// 单链表删除pos位置的值
void SListErase(SListNode** pplist, SListNode* pos);
```

创建结点结构

```
typedef int SLDataType;

typedef struct SListNode
{
    int data;
    struct SListNode* next;
}SLTNode;
```

建立新结点

```
SLTNode* BuyListNode(SLDataType x)
{
    SLTNode* newnode = (SLTNode*)malloc(sizeof(SLTNode));
    if (newnode == NULL)
    {
        printf("malloc fail\n");
        exit(-1);
    }
    newnode->data = x;
    newnode->next = NULL;
    return newnode;
}
```

尾插法建立单链表

尾插法要考虑两个方面：

1.如果链表为空，那么新结点就是第一个结点

2.注意要修改的实参是指针变量，因此要改变指针变量的值，函数体内部应该传递二级指针

pphead是二级指针，不能为NULL，加上断言更加安全

```
//尾插建立
void SListPushBack(SLTNode** pphead, SLDataType x)
{
    assert(pphead);

    //建立新结点
    SLTNode* newnode = BuyListNode(x);
    if (*pphead == NULL)
    {
        *pphead = newnode;
    }
    else
    {
        //找到尾结点
        SLTNode* tail = *pphead;
        while (tail->next != NULL)
```

```
        {
            tail = tail->next;
        }
        tail->next = newnode;
    }
}
```

头插法建立单链表

总是第一个元素， 不需要考虑链表为空的情况

```
//头插建立
void SListPushFront(SLTNode** pphead, SLDataType x)
{
    assert(pphead);

    //建立新结点
    SLTNode* newnode = BuyListNode(x);
    newnode->next = *pphead;//就是newnode->next = plist
    *pphead = newnode;
}
```

尾删法

尾删法要考虑两个方面：

1.不能只把最后一个元素的空间释放，还要将其前驱结点的指针置空，避免出现野指针问题

2.表中只有一个元素时的处理

```
void SListPopBack(SLTNode** pphead)//尾删
{
    assert(pphead);

    //while(tail->next!=NULL)

    //温柔处理方式
    if (*pphead == NULL)
    {
        return;
    }
```

```
    //暴力方式
    //assert(*pphead != NULL);
    //表中只有1个元素
    if ((*pphead)->next == NULL)
    {
        free(*pphead);
        *pphead = NULL;
    }
    //表中大于1个元素
    else
    {
        SLTNode* pre = NULL;//前驱结点
        SLTNode* tail = *pphead;
        while (tail->next)
        {
            pre = tail;
            tail = tail->next;
        }
        free(tail);
        tail = NULL;
        pre->next = NULL;
    }
}
```

不定义前驱结点的尾删法

```
void SListPopBack2(SLTNode** pphead)//尾删
{
    assert(pphead);

    //温柔处理方式
    if (*pphead == NULL)
    {
        return;
    }
    //暴力方式
    //assert(*pphead != NULL);

    //表中只有1个元素
```

```
    if ((*pphead)->next == NULL)
    {
        free(*pphead);
        *pphead = NULL;
    }
    //表中大于1个元素
    else
    {
        SLTNode* tail = *pphead;
        //while(tail->next!=NULL)
        while (tail->next->next)
        {
            tail = tail->next;
        }
        free(tail->next);
        tail->next = NULL;
    }

}
```

头删法

```
void SListPopFront(SLTNode** pphead)//头删
{
    assert(pphead);
    //空链表，温柔处理方式
    if (*pphead == NULL)
    {
        return;
    }
    //assert(*pphead!=NULL);
    SLTNode* next = (*pphead)->next;//plist的下一个结点
    free(*pphead);
    *pphead = next;//plist的下一个结点作为plist
}

```

查找--同时具有修改的作用

```c
SLTNode* SListFind(SLTNode* phead, SLDataType x)
{
    SLTNode* cur = phead;
    //while (cur != NULL)
    while (cur)
    {
        if (cur->data == x)
        {
            return cur;
        }
        cur = cur->next;
    }
    return NULL;
}
```

任意位置插入在pos前插入

在pos前插入，由于要找前驱，时间复杂度是O(n)

要考虑在第一个位置插入的情况 -- 头插

要插入的位置pos不能是NULL，加上断言更加安全

```c
void SListInsert(SLTNode** pphead, SLTNode* pos, SLDataType x)
{
    assert(pphead);
    assert(pos);

    //考虑在第一个位置插入
    if (pos == *pphead)
    {
        SListPushFront(pphead, x);
    }
    else
    {
        SLTNode* newnode = BuySListNode(x);
        SLTNode* pre = *pphead;//从头开始
    while (pre->next != pos)
    {
        pre = pre->next;//找到pos的前驱
```

```
    }
    pre->next = newnode;//pre不在指向pos，而是newnode
    newnode->next = pos;
    }
}

void SListInsert2(SLTNode** pphead, SLTNode* pos, SLDataType x)
{
    assert(pos);

    SLTNode* newnode = BuySListNode(x);
    //考虑在第一个位置插入
    if (pos == *pphead)
    {
        newnode->next = *pphead;
        *pphead = newnode;
    }
    else
    {
        SLTNode* pre = *pphead;//从头开始
        while (pre->next != pos)
        {
            pre = pre->next;//找到pos的前驱
        }
        pre->next = newnode;
        newnode->next = pos;
    }
}

```

任意位置插入 -- 在pos后插入，这个更合适也更简单

```
void SListInsertAfter(SLTNode* pos, SLDataType x)
{
    assert(pos);
    SLTNode* newnode = BuyListNode(x);
    newnode->next = pos->next;//这两句代码不可以调换顺序
    pos->next = newnode;
}
```

任意位置删除

由于要找前驱，时间复杂度是O(n)

```
// 删除pos位置的值
void SListErase(SLTNode** pphead, SLTNode* pos)
{
    assert(pphead);
    if (pos == *pphead)
    {
        //SListPopFront(pphead);
        *pphead = pos->next;
        free(pos);
        pos = NULL;
    }
    else
    {
        SLTNode* pre = *pphead;
        while (pre->next != pos)
        {
            pre = pre->next;
        }
        pre->next = pos->next;
        free(pos);
    }
}
```

删除pos位置的后一个值这个更合适也更简单

```
void SListEraseAfter(SLTNode* pos)
{
    assert(pos->next != NULL);
    SLTNode* next = pos->next;
    pos->next = next->next;
    free(next);
    next = NULL;
}
```

释放空间

```c
//释放空间
void SListDestory(SLTNode** pphead)
{
    assert(pphead);
    SLTNode* cur = *pphead;
    while (cur)
    {
        SLTNode* next = cur->next;
        free(cur);
        cur = next;
    }
    *pphead = NULL;
 }
```

打印链表

```c
void SListPrint(SLTNode* phead)
{
    assert(pphead);

    SLTNode* cur = phead;
    while (cur != NULL)
    {
        printf("%d->", cur->data);
        cur = cur->next;
    }
    printf("NULL\n");
}
```

OJ题

1.给你一个链表的头节点 head 和一个整数 val，请你删除链表中所有满足Node.val= val 的节点，并返回新的头节点

```c
struct ListNode* removeElements(struct ListNode* head, int val)
```

```
{
    struct ListNode* pre = NULL, * cur = head;
    while (cur)
    {
        if (cur->val == val)
        {
            //头删
            if (cur == head)
            {
                head = cur->next;
                free(cur);
                cur = head;
            }
            else
            {
                //删除
                pre->next = cur->next;
                free(cur);
                cur = pre->next;
            }
        }
        else
        {
            //迭代往后走
            pre = cur;
            cur = cur->next;
        }
    }
    return head;
}
```

2.给你单链表的头节点，请你反转链表，并返回反转后的链表。 head

思路一：

```
struct ListNode* reverseList(struct ListNode* head)
{
    if (head == NULL)
```

```
    {
        return NULL;
    }
    struct ListNode* n1, *n2, *n3;
    n1 = NULL;
    n2 = head;
    n3 = head->next;
    while (n2)
    {
        //翻转
        n2->next = n1;
        //迭代
        n1 = n2;
        n2 = n3;
        if (n3)
        {
            n3 = n3->next;
        }
    }
    return n1;
}
```

思路二：取原链表中结点，头插到newhead新链表中

```
struct ListNode* reverseList(struct ListNode* head)
{
    struct ListNode* cur = head;
    struct ListNode* newhead = NULL;
    while (cur != NULL)//可以处理没有节点的情况
    {
        struct ListNode* next = cur->next;
        //头插
        cur->next = newhead;
        newhead = cur;
        //迭代
        cur = next;
    }
    return newhead;
}
```

3.返回中间结点

给定一个头结点为 head 的非空单链表，返回链表的中间结点如果有两个中间结点，则返回第二个中间结点.

```c
//快慢指针
struct ListNode* middleNode(struct ListNode* head)
{
    struct ListNode* slow,*fast;
    slow = fast=head;
    while(fast!=NULL && fast->next!=NULL)
    {
        slow=slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

4.输入一个链表，输出该链表中倒数第k个结点

1.fast先走k步

2.slow和fast再一起走，fast == NULL时，slow就是倒数第k个

```c
struct ListNode* FindKthToTail(struct ListNode* pListHead,int k)
{
    struct ListNode* fast, * slow;
    slow = fast = pListHead;
    while (k--)
    {
        //k大于链表长度
        if (fast == NULL)
        {
            return NULL;
        }
        fast = fast->next;
    }
    while (fast)
```

```
    {
        slow = slow->next;
        fast = fast->next;
    }
    return slow;
}
```

5.合并链表

将两个升序链表合并为一个新的 **升序** 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

1.依此比较链表中的结点，每次取小的结点，尾插到新链表即可

```
struct ListNode* mergeTwoLists(struct ListNode* l1,struct ListNode* l2)
{
    //如果其中一个链表为空，就返回另一个
    if (l1 == NULL)
        return l2;
    if (l2 == NULL)
        return l1;
    struct ListNode* head = NULL, * tail = NULL;
    while (l1 && l2)
    {
        if (l1->val < l2->val)
        {
            if (head == NULL)
            {
                head = tail = l1;
            }
            else
            {
                tail->next = l1;
                tail = tail->next;
            }
            l1 = l1->next;
        }
        else
        {
            if (head == NULL)
```

```
            {
                head = tail = l2;
            }
            else
            {
                tail->next = l2;
                tail = tail->next;
            }
            l2 = l2->next;
        }
    }
    if (l1)
    {
        tail->next = l1;
    }
    if (l2)
    {
        tail->next = l2;
    }
    return head;
}
```

6.链表分割_牛客题霸_牛客网 (nowcoder.com)

现有一链表的头指针 ListNode* **pHead**，给一定值x，编写一段代码将所有小于x的结点排在其余结点之前，且不能改变原来的数据顺序，返回重新排列后的链表的头指针。

```

#include <cstddef>
#include <cstdlib>
// struct ListNode {
//     int val;
//     struct ListNode *next;
//     ListNode(int x) : val(x), next(NULL) {}
// };
class Partition
{
public:
    ListNode* partition(ListNode* pHead, int x)
```

```
    {
        struct ListNode* lessHead,*lessTail,*greaterHead,*greaterTail;
        //开一个哨兵位的头节点，方便尾插
        lessHead=lessTail=(struct ListNode*)malloc(sizeof(struct ListNode));
        lessTail->next=NULL;
        greaterHead=greaterTail=(struct ListNode*)malloc(sizeof(struct ListNode));
        greaterTail->next=NULL;
        struct ListNode * cur=pHead;
        while (cur)
        {
            if (cur->val<x)
            {
                lessTail->next=cur;
                lessTail=cur;
            }
            else
            {
                greaterTail->next=cur;
                greaterTail=cur;
            }
            cur=cur->next;
        }
        lessTail->next=greaterHead->next;
        greaterTail->next=NULL;//一定要考虑成环的情况，要把原链表的尾结点置空

        struct ListNode* newHead = lessHead->next;
        free(lessHead);
        free(greaterHead);
        return newHead;
    }
};
```

7.链表的回文结构

https://www.nowcoder.com/practice/d281619e4b3e4a60a2cc66ea32855bfa?tpId=182&tqI

对于一个链表，请设计一个时间复杂度为O(n),额外空间复杂度为O(1)的算法，判断其是否为回文结构。

给定一个链表的头指针**A**，请返回一个bool值，代表其是否为回文结构。保证链表长度小于等于900。

```
1
2  // struct ListNode {
3  //     int val;
4  //     struct ListNode *next;
5  //     ListNode(int x) : val(x), next(NULL) {}
6  // };
7
8  struct ListNode* middleNode(struct ListNode* head)
9  {
10     struct ListNode* slow,*fast;
11     slow = fast=head;
12     while(fast!=NULL && fast->next!=NULL)
13     {
14         slow=slow->next;
15         fast = fast->next->next;
16     }
17     return slow;
18 }
19
20 struct ListNode* reverseList(struct ListNode* head)
21 {
22     struct ListNode* cur = head;
23     struct ListNode* newhead = NULL;
24     while (cur != NULL)//可以处理没有节点的情况
25     {
26         struct ListNode* next = cur->next;
27         //头插
28         cur->next = newhead;
29         newhead = cur;
30         //迭代
31         cur = next;
32     }
33     return newhead;
34 }
35
36 class PalindromeList {
37 public:
38     bool chkPalindrome(ListNode* A)
```

```
    {
        struct ListNode* mid=middleNode(A);
        struct ListNode* rHead = reverseList(mid);

        struct ListNode* curA=A;
        struct ListNode* curR=rHead;
        while(curA && curR)
        {
            if(curA->val != curR->val)
            {
                return  false;
            }
            else
            {
                curA=curA->next;
                curR=curR->next;
            }
        }
        return true;
    }
};
```

8.相交链表

给你两个单链表的头节点 headA 和 headeB，请你提出并返回两个单表相交的起始节点。如果两个链表没有交点返回 null

思路1：暴力求解 -- 穷举 O(n^2)

依此取A链表中的每个结点跟B链表中的所有结点比较，如果有地址相同的结点，就是相交，第一个相同的交点

思路2:优化到O(n)

1.尾结点相同就是相交，否则就是不相交

2.求交点：长的链表先走长度差步，再同时走，第一个相同就是交点

```
struct ListNode* getIntersectionNode(struct ListNode* headA, struct ListNode* headB)
{
    struct ListNode* tailA = headA;
    struct ListNode* tailB = headB;
    int lenA = 1;
```

```
    while (tailA->next)
    {
        ++lenA;
        tailA = tailA->next;
    }
    int lenB = 1;

    while (tailB->next)
    {
        lenB++;
        tailB->next;
    }
    if (tailA != tailB)
    {
        return NULL;
    }
    //距离差
    int gap = abs(lenA - lenB);
    //长的先走差距步，再同时走找交点
    int lenA = 1;
    struct ListNode* longList = headA;//假设A长
    struct ListNode* shortList = headB;
    if (lenA < lenB)//如果A短，就把A给shortList
    {
        shortList = headA;
        longList = headB;
    }
    //长的先走差距步
    while (gap--)
    {
        longList = longList->next;
    }

    while (longList!=shortList)
    {
        longList = longList->next;
        shortList = shortList->next;
    }
    return longList;
}
```

9.给定一个链表，判断链表中是否有环

链表带环（循环链表的一种），尾结点->next指向自己 的元素

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置(索引从0 始)。如果 pos是-1，则在该链表中没有环。注意: pos不作为参数进行传递，仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 true 。否则，返回 false 。

快慢指针：slow和fast指向链表的开始，slow一次走一步，fast一次走两步，不带环，fast就会为空；带环，fast就会在环里面追上slow

```
bool hasCycle(struct ListNode* head)
{
    struct ListNode* slow = head, * fast = head;
    while (fast && fast->next)
    {
        slow = slow->next;
        fast = fast->next->next;
        if (slow = fast)//相遇，就是环
        {
            return true;
        }
    }
    return false;
}
```

延伸问题：

1.为什么slow和fast一定会在环中相遇?会不会在环里面错过，永远遇不上？结论：一定会相遇

假设进环之后slow和fast距离为N，slow每走一步fast每走两步他们之间的距离就会减1，他们的距离每次减1，最终会减到0，所以他们一定会相遇。

2.为什么slow走一步，fast走两步呢？能不能fast走3、4、5...n步呢？结论：n>2不一定会相遇

以此类推。

***10.求环形链表的入口点?**

```
struct ListNode* detectCycle(struct ListNode* head)
```

```
{
    struct ListNode* slow = head, *fast = head;
    while (fast && fast->next)
    {
        slow = slow->next;
        fast = fast->next->next;
        if (slow = fast)//相遇，带环
        {
            struct ListNode* meet = slow;
            //公式证明的L = C - X
            while (slow != head)
            {
                meet = meet->next;
                head = head->next;
            }
            return meet;
        }
    }
    return NULL;//不带环
}
```

思路2：

*11.复杂链表的复制

剑指 Offer 35. 复杂链表的复制 - 力扣（Leetcode）

请实现 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 指针指向下一个节点，还有一个 指针指向链表中的任意节点或者

```
struct Node* cur = head;
while (cur)
{
    struct Node* copy = (struct Node*)malloc(sizeof(struct Node));
    copy->val = cur->val;//copy结点的val = 原结点的val
    //插入copy结点
    copy->next = cur->next;
    cur->next = copy;
    cur = copy->next;
```

```
}
```

```
cur = head;
while (cur)
{
    struct Node* copy = cur->next;
    if (cur->random == NULL)
    {
        copy->random = NULL;
    }
    else
    {
        copy->random = cur->random->next;
    }
    cur = copy->next;
}
```

```
struct Node* copyHead = NULL, * copyTail = NULL;
cur = head;
while (cur)
{
    struct Node* copy = cur->next;
    struct Node* next = copy->next;
    if (copyTail == NULL)
    {
        copyHead = copyTail;
    }
    else
    {
        copyTail->next = copy;
        copyTail = copy;
    }
    //恢复原链表
    cur->next = next;
    cur = next;
```

```
}
```

完整代码

```
struct Node* copyRandomList(struct Node* head)
{
    //
    struct Node* cur = head;
    while (cur)
    {
        struct Node* copy = (struct Node*)malloc(sizeof(struct Node));
        copy->val = cur->val;//copy结点的val = 原结点的val
        //插入copy结点
        copy->next = cur->next;
        cur->next = copy;
        cur = copy->next;
    }
    //
    cur = head;
    while (cur)
    {
        struct Node* copy = cur->next;
        if (cur->random == NULL)
        {
            copy->random = NULL;
        }
        else
        {
            copy->random = cur->random->next;
        }
        cur = copy->next;
    }
    //3.把拷贝结点解下来，链接成新链表，同时恢复原链表
    struct Node* copyHead = NULL, * copyTail = NULL;
    cur = head;
    while (cur)
    {
        struct Node* copy = cur->next;
        struct Node* next = copy->next;
```

```
        if (copyTail == NULL)
        {
            copyHead = copyTail = copy;
        }
        else
        {
            copyTail->next = copy;
            copyTail = copy;
        }
        //恢复原链表
        cur->next = next;
        cur = next;
    }
    return copyHead;
}
```

2. 带头、不带头

3. 循环、非循环

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：

1. 无头单向非循环链表：**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**，如哈希桶、图的邻接表等等。另外这种结构在**笔试面试**中出现很多。
2. 带头双向循环链表：**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。另外这个结构虽然结构复杂，但是使用代码实现以后会发现结构会带来很多优势，实现反而简单了，后面我们代码实现了就知道了。

循环双向链表的实现

```
// 2、带头+双向+循环链表增删查改实现
typedef int LTDataType;
typedef struct ListNode
{
```

```
    LTDataType _data;
    struct ListNode* _next;
    struct ListNode* _prev;
}ListNode;
// 创建返回链表的头结点.
ListNode* ListCreate();
// 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 x);
// 双向链表尾删
void ListPopBack(ListNode* plist);
// 双向链表头插
void ListPushFront(ListNode* plist, LTDataType x);
// 双向链表头删
void ListPopFront(ListNode* plist);
// 双向链表查找
ListNode* ListFind(ListNode* plist, LTDataType x);
// 双向链表在pos的前面进行插入
void ListInsert(ListNode* pos, LTDataType x);
// 双向链表删除pos位置的节点
void ListErase(ListNode* pos);
```

初始化链表

```
LTNode* ListInit()//初始化
{
    //作为哨兵位的头结点,不用给值，随机值，存啥不重要
    LTNode* phead = (LTNode*)malloc(sizeof(LTNode));
    phead->next = phead;
    phead->pre = phead;
    return phead;
}
```

建立新结点

```c
//建立新结点
LTNode* BuyLisyNode(SLDataType x)
{
    LTNode* newnode = (LTNode*)malloc(sizeof(LTNode));
    newnode->data = x;
    newnode->next = NULL;
    newnode->pre = NULL;
    return newnode;
}
```

尾插法

```c
void ListPushBack(LTNode* phead, SLDataType x)//尾插
{
    assert(phead);
    LTNode* tail = phead->pre;
    LTNode* newnode = (LTNode*)malloc(sizeof(LTNode));
    newnode->data = x;
    //LTNode* tail = phead->pre;
    //LTNode* newnode = BuyLisyNode(x);
    tail->next = newnode;
    newnode->pre = tail;
    newnode->next = phead;
    phead->pre = newnode;
}
```

尾删法

phead -> next == phead,表示链表为空，不能删除了

不能提前释放空间，会出现野指针，要Tailpre记录一下尾结点的地址

```c
void ListPopBack(LTNode* phead)//尾删
{
    //不能删除哨兵位
    assert(phead);
    assert(phead->next != phead);
```

```
    LTNode* tail = phead->pre;
    LTNode* tailpre = tail->pre;
    free(phead->pre);//free(tail);

    //记录为指针，避免出现野指针
    tailpre->next = phead;
    phead->pre = tailpre;
}
```

另一种写法：

```
void ListPopBack2(LTNode* phead)//尾删
{
    //不能删除哨兵位
    assert(phead);
    assert(phead->next != phead);
    LTNode* tail = phead->pre;

    phead->pre = tail->pre;
    tail->pre->next = phead;
    free(tail);
}
```

头插法

```
void ListPushFront(LTNode* phead, SLDataType x)//头插
{
    assert(phead);
    LTNode* newnode = BuyLisyNode(x);
    LTNode* next = phead->next;

    //空链表也不会有问题
    phead->next = newnode;
    newnode->pre = phead;
    newnode->next = next;
    next->pre = newnode;
}
```

头删法

```
void ListPopFront(LTNode* phead)//头删
{
    assert(phead);
    assert(phead->next != phead);//链表空

    LTNode* next = phead->next;
    LTNode* nextNext = next->next;

    phead->next = nextNext;
    nextNext->pre = phead;
    free(next);
}
```

查找

```
LTNode* ListFind(LTNode* phead, SLDataType x)//查找
{
    assert(phead);
    LTNode* cur = phead->next;
    while (cur != phead)
    {
        if (cur->data == x)
        {
            return cur;
        }
        cur = cur->next;
    }
    return NULL;
}
```

任意位置插入

```
//在pos位置前插入
LTNode* ListInsert(LTNode* pos, SLDataType x)//任意位置插入
{
    assert(pos);
```

```
    LTNode* posPre = pos->pre;
    LTNode* newnode = BuyLisyNode(x);

    //posPre   newnode   pos
    posPre->next = newnode;
    newnode->pre = posPre;
    newnode->next = pos;
    pos->pre = newnode;
}
```

头插尾插的改造

```
void ListPushBack2(LTNode* phead, SLDataType x)//尾插
{
    assert(phead);
    ListInsert(phead, x);
}
void ListPushFront2(LTNode* phead, SLDataType x)//头插
{

    assert(phead);
    ListInsert(phead->next, x);
}
```

任意位置删除

```
LTNode* ListErase(LTNode* pos)//任意位置删除
{
    assert(pos);
    LTNode* posPre = pos->pre;
    LTNode* posNext = pos->next;

    posPre->next = posNext;
    posNext->pre = posPre;
    free(pos);
    pos = NULL;
}
```

头删尾删的改造

```c
void ListPopFront3(LTNode* phead)//头删
{
    assert(phead);
    assert(phead->next != phead);//链表空
    ListErase(phead->next);

}

void ListPopBack3(LTNode* phead)//尾删
{
    //不能删除哨兵位
    assert(phead);
    assert(phead->next != phead);
    ListErase(phead->pre);
}
```

打印链表

```c
void ListPrint(LTNode* phead)
{
    assert(phead);
    LTNode* cur = phead->next;
    while (cur != phead)
    {
        printf("%d ", cur->data);
        cur = cur->next;
    }
    printf("\n");
}
```

销毁空间

//最后释放头结点

free(phead);这种写法错误

但是释放的是形参，而不是实参，plist并不会改变，因此应该穿二级指针，但是为了保持接口的一致性，如果还要传一级指针的话，哨兵位就放在函数体外释放

```c
void ListDestroy(LTNode* phead)//销毁空间
{
    assert(phead);
    LTNode* cur = phead;
    while (cur!=phead)
    {
        //释放一个就找不到他的下一个节点了，因此要保存下来下一个节点
        LTNode* next = cur->next;
        free(cur);
        cur = next;
    }

}
```