2 顺序表和链表

1. 线性表

线性表（*linear list*）是n个具有相同特性的数据元素的有限序列。 线性表是一种在实际中广泛使 用的数据结构，常见的线性表：顺序表、链表、栈、队列、字符串...

线性表在逻辑上是线性结构，也就说是连续的一条直线。但是在物理结构上并不一定是连续的， 线性表在物理上存储时，通常以数组和链式结构的形式存储。

2. 顺序表

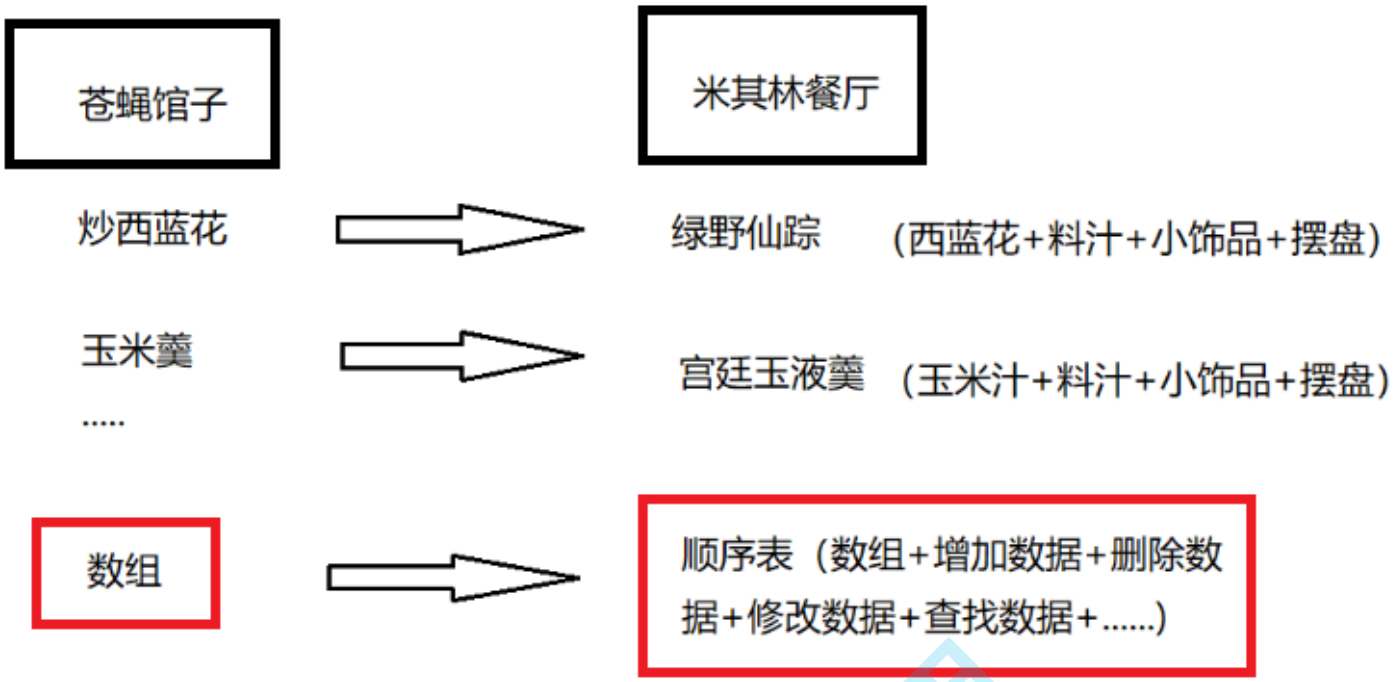
2.1 概念与结构

概念：顺序表是用一段**物理地址连续**的存储单元依次存储数据元素的线性结构，一般情况下采用数组存储。

顺序表

> 顺序表和数组的区别?
>
> 顺序表的底层结构是数组，对数组的封装，实现了常用的增删改查等接口

比特就业课

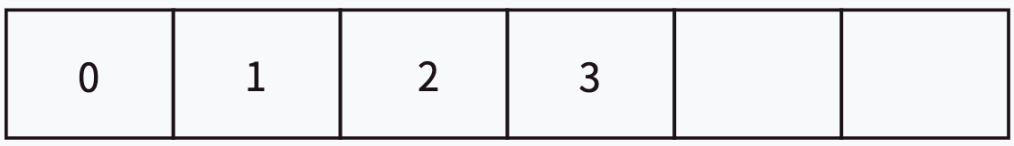


2.2 分类

2.2.1 静态顺序表

概念：使用定长数组存储元素

```
//静态顺序表
typedef int SLDataType;
#define N 7
typedef struct SeqList{
    SLDataType a[N];    //定长数组
    int size;           //有效数据个数
}SL;
```

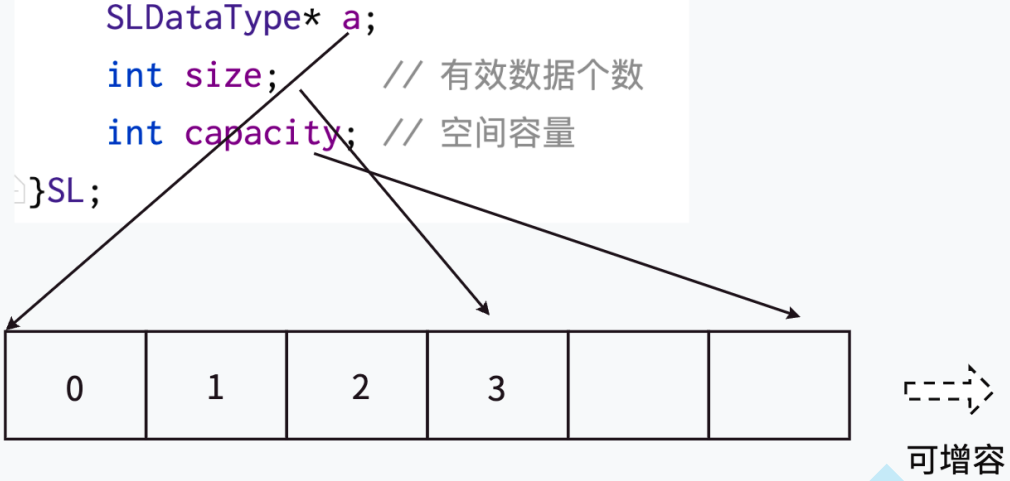


静态顺序表缺陷：空间给少了不够用，给多了造成空间浪费

2.2.2 动态顺序表

比特就业课

```
// 动态顺序表 -- 按需申请
typedef struct SeqList
{
    SLDataType* a;
    int size;       // 有效数据个数
    int capacity; // 空间容量
}SL;
```



2.3 动态顺序表的实现

`SeqList.h`

```
#define INIT_CAPACITY 4
typedef int SLDataType;
// 动态顺序表 -- 按需申请
typedef struct SeqList
{
    SLDataType* a;
    int size;     // 有效数据个数
    int capacity; // 空间容量
}SL;

//初始化和销毁
void SLInit(SL* ps);
void SLDestroy(SL* ps);
void SLPrint(SL* ps);
//扩容
void SLCheckCapacity(SL* ps);

//头部插入删除 / 尾部插入删除
void SLPushBack(SL* ps, SLDataType x);
void SLPopBack(SL* ps);
void SLPushFront(SL* ps, SLDataType x);
void SLPopFront(SL* ps);

//指定位置之前插入/删除数据
void SLInsert(SL* ps, int pos, SLDataType x);
void SLErase(SL* ps, int pos);
```

比特就业课

```
27 int SLFind(SL* ps, SLDataType x);
```

> 💡 代码小提示
>
> 编写代码过程中要勤测试，避免写出大量代码后再测试而导致出现问题，问题定位无从下手。

2.4 顺序表算法题

2.4.1 移除元素

https://leetcode.cn/problems/remove-element/description/

2.4.2 删除有序数组中的重复项

https://leetcode.cn/problems/remove-duplicates-from-sorted-array/description/

2.4.3 合并两个有序数组

https://leetcode.cn/problems/merge-sorted-array/description/

2.5 顺序表问题与思考

- 中间/头部的插入删除，时间复杂度为O(N)
- 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。
- 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。

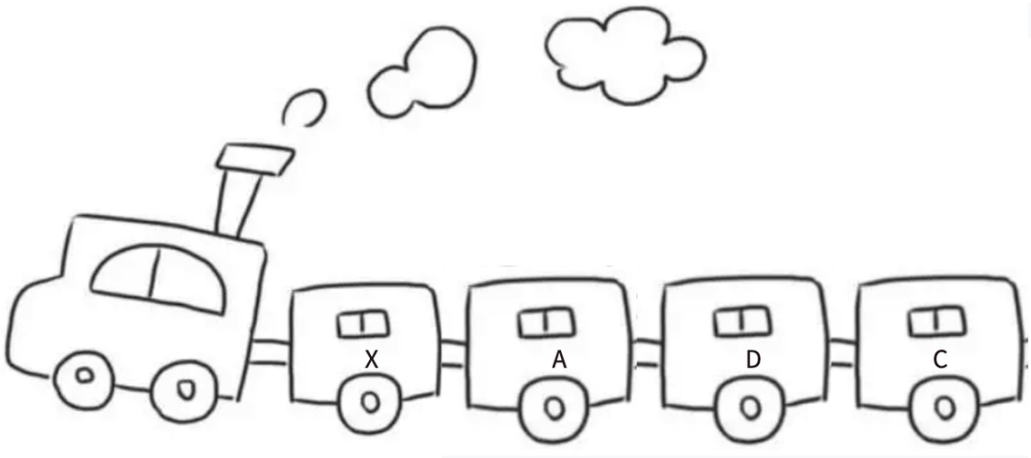
> 思考：如何解决以上问题呢?

3. 单链表

3.1 概念与结构

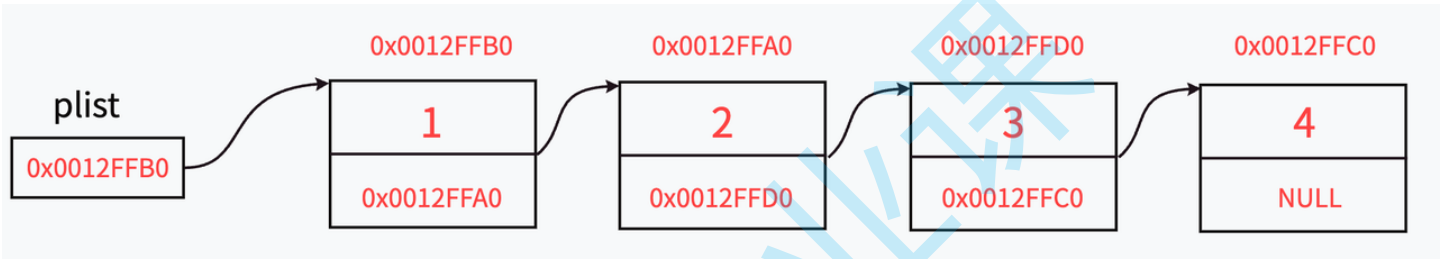
概念：链表是一种**物理存储结构上非连续**、非顺序的存储结构，数据元素的**逻辑顺序**是通过链表中的**指针链接**次序实现的。

比特就业课



淡季时车次的车厢会相应减少，旺季时车次的车厢会额外增加几节。只需要将火车里的某节车厢去掉/加上，不会影响其他车厢，每节车厢都是独立存在的。

在链表里，每节“车厢”是什么样的呢？



3.1.1 结点

与顺序表不同的是，链表里的每节"车厢"都是独立申请下来的空间，我们称之为“结点/结点”

> 结点的组成主要有两个部分：当前结点要保存的数据和保存下一个结点的地址（指针变量）。

图中指针变量 plist保存的是第一个结点的地址，我们称plist此时“指向”第一个结点，如果我们希望plist“指向”第二个结点时，只需要修改plist保存的内容为0x0012FFA0。

链表中每个结点都是独立申请的（即需要插入数据时才去申请一块结点的空间），我们需要通过指针变量来保存下一个结点位置才能从当前结点找到下一个结点。

3.1.2 链表的性质

> 1、链式机构在逻辑上是连续的，在物理结构上不一定连续
>
> 2、结点一般是从堆上申请的
>
> 3、从堆上申请来的空间，是按照一定策略分配出来的，每次申请的空间可能连续，可能不连续

结合前面学到的结构体知识，我们可以给出每个结点对应的结构体代码：

假设当前保存的结点为整型：

```
1 struct SListNode
2 {
3     int data;       //结点数据
4     struct SListNode* next; //指针变量用保存下一个结点的地址
```

比特就业课

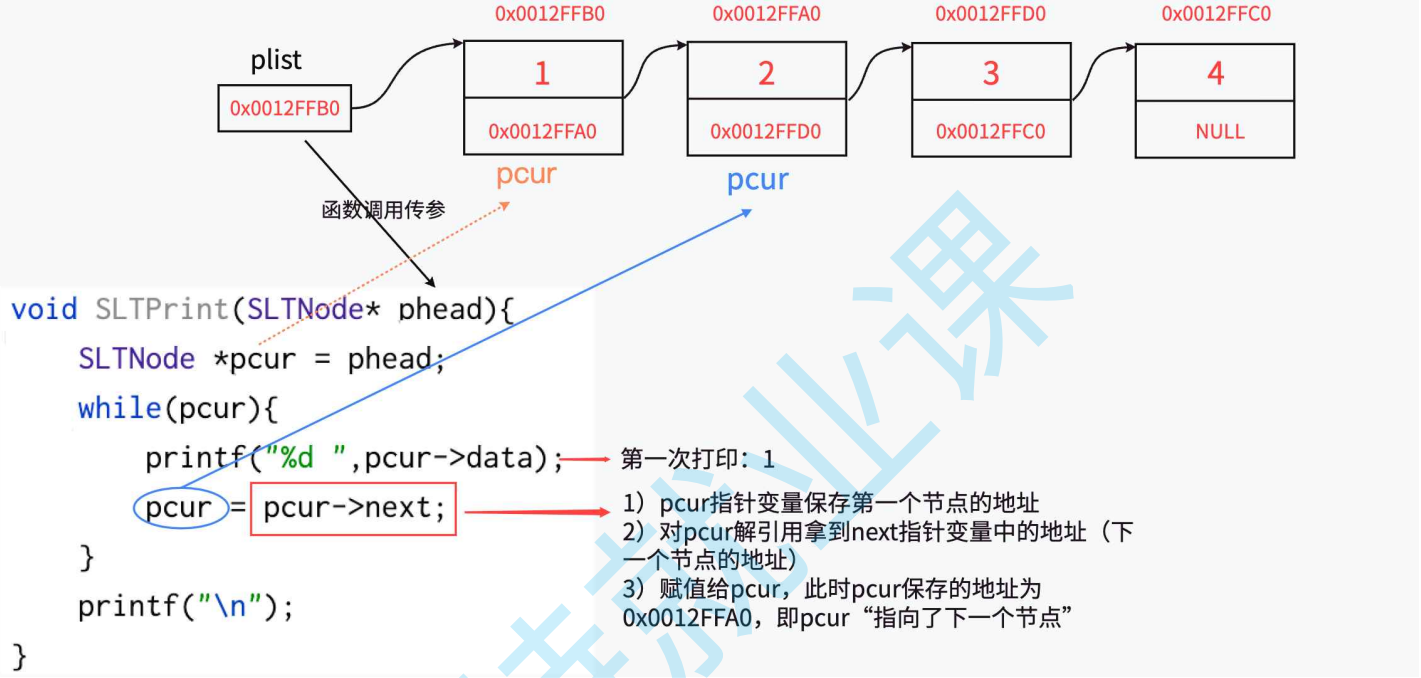
```
};
```

当我们想要保存一个整型数据时，实际是向操作系统申请了一块内存，这个内存不仅要保存整型数据，也需要保存下一个结点的地址（当下一个结点为空时保存的地址为空）。

当我们想要从第一个结点走到最后一个结点时，只需要在当前结点拿上下一个结点的地址就可以了。

3.1.3 链表的打印

给定的链表结构中，如何实现结点从头到尾的打印?



思考：当我们想保存的数据类型为字符型、浮点型或者其他自定义的类型时，该如何修改?

3.2 实现单链表

`SList.h`

```
typedef int SLTDataType;
typedef struct SListNode
{
    SLTDataType data;       //结点数据
    struct SListNode* next; //指针保存下一个结点的地址
}SLTNode;


void SLTPrint(SLTNode* phead);

//头部插入删除/尾部插入删除
void SLTPushBack(SLTNode** pphead, SLTDataType x);
void SLTPushFront(SLTNode** pphead, SLTDataType x);
```

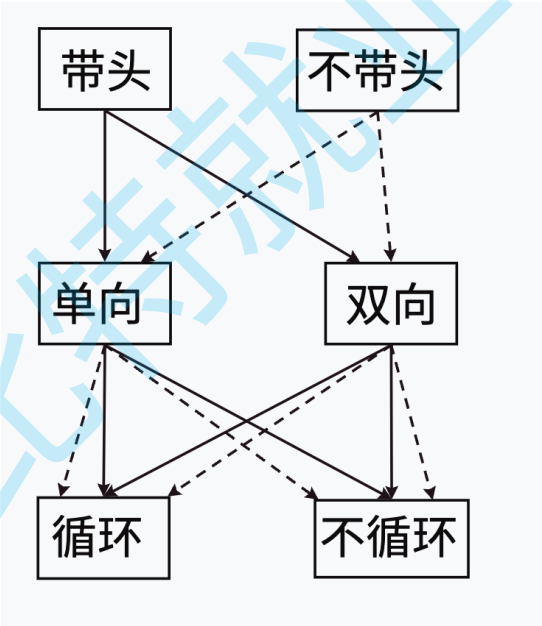
比特就业课

```
void SLTPopBack(SLTNode** pphead);
void SLTPopFront(SLTNode** pphead);

//查找
SLTNode* SLTFind(SLTNode* phead, SLTDataType x);
//在指定位置之前插入数据
void SLTInsert(SLTNode** pphead, SLTNode* pos, SLTDataType x);
//删除pos结点
void SLTErase(SLTNode** pphead, SLTNode* pos);
//在指定位置之后插入数据
void SLTInsertAfter(SLTNode* pos, SLTDataType x);
//删除pos之后的结点
void SLTEraseAfter(SLTNode* pos);
//销毁链表
void SListDestroy(SLTNode** pphead);
```

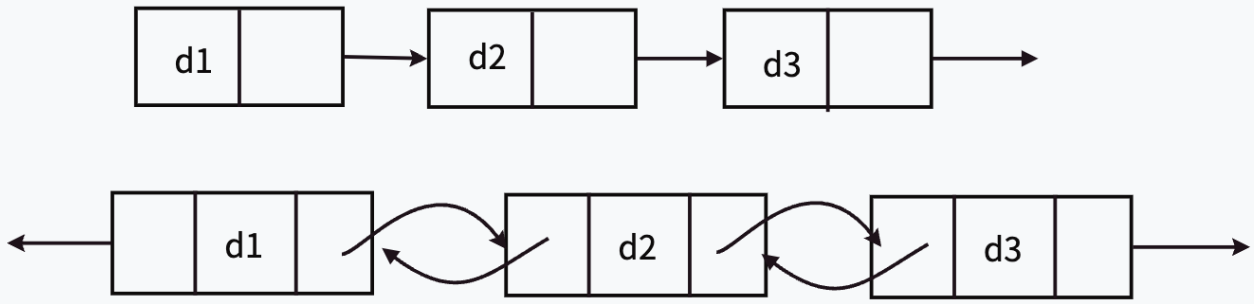
3.3 链表的分类

链表的结构非常多样，以下情况组合起来就有8种（2 x 2 x 2）链表结构：

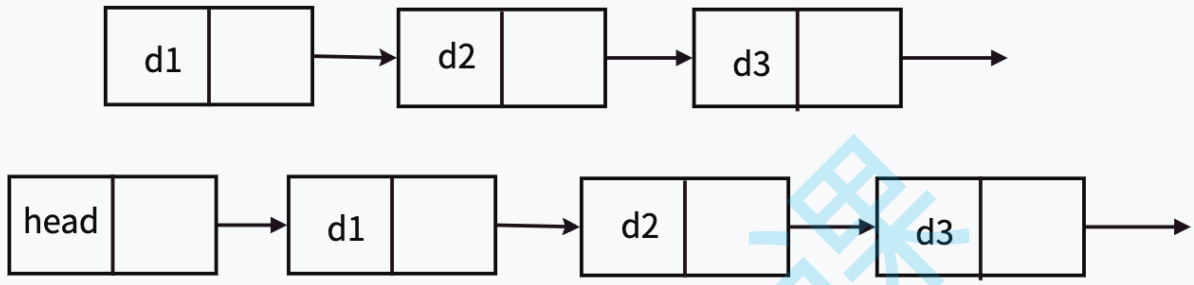


链表说明：

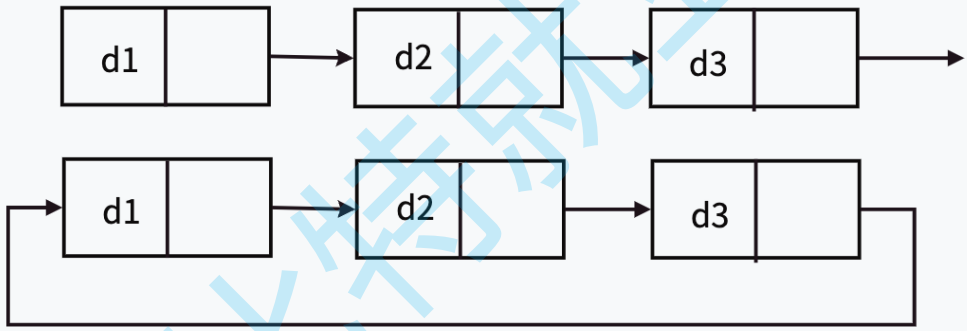
1.单向或者双向



2.带头或者不带头



3.循环或者不循环



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：单链表和双向带头循环链表

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。

3.4 单链表算法题

3.4.1 移除链表元素

https://leetcode.cn/problems/remove-linked-list-elements/description/

> OJ代码有bug怎么办？VS调试技能用起来

1）将OJ代码复制粘贴到vs上

```c
#include<stdio.h>
#include<stdlib.h>

struct ListNode {
    int val;//int val
    struct ListNode* next;
};
typedef struct ListNode LTNode;

LTNode* removeElements(LTNode* head, int val) {
    //创建一个空链表
    LTNode* newHead, * newTail;
    newHead = newTail = NULL;

    //遍历原链表
    LTNode* pcur = head;
    while (pcur)
    {
        //找值不为val的节点，尾插到新链表中
        if (pcur->val != val)
        {
            //链表为空
            if (newHead == NULL)
            {
                newHead = newTail = pcur;
            }
            else {
                //链表不为空
                newTail->next = pcur;
                newTail = newTail->next;
            }
        }
        pcur = pcur->next;
    }
    return newHead; 已用时间 <= 1ms
}
```

比特就业课

2）创建测试方法，调用本次要调试的目标方法

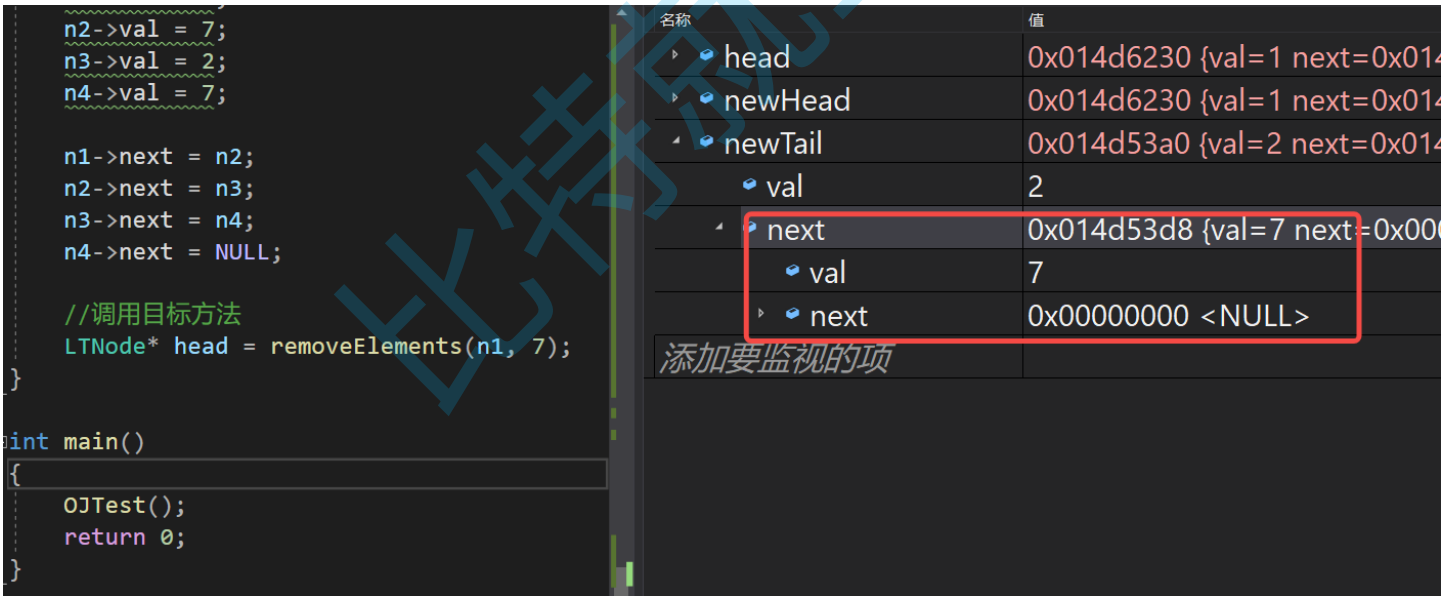
```c
void OJTest()
{
    //构建参数：链表
    LTNode* n1 = (LTNode*)malloc(sizeof(LTNode));
    LTNode* n2 = (LTNode*)malloc(sizeof(LTNode));
    LTNode* n3 = (LTNode*)malloc(sizeof(LTNode));
    LTNode* n4 = (LTNode*)malloc(sizeof(LTNode));

    n1->val = 1;
    n2->val = 7;
    n3->val = 2;
    n4->val = 7;

    n1->next = n2;
    n2->next = n3;
    n3->next = n4;
    n4->next = NULL;

    //调用目标方法
    LTNode* head = removeElements(n1, 7);
}
```

3）利用vs调试工具排查代码问题



3.4.2 反转链表

https://leetcode.cn/problems/reverse-linked-list/description/

3.4.3 链表的中间结点

https://leetcode.cn/problems/middle-of-the-linked-list/description/

3.4.4 合并两个有序链表

https://leetcode.cn/problems/merge-two-sorted-lists/description/

3.4.5 链表分割

https://www.nowcoder.com/practice/0e27e0b064de4eacac178676ef9c9d70

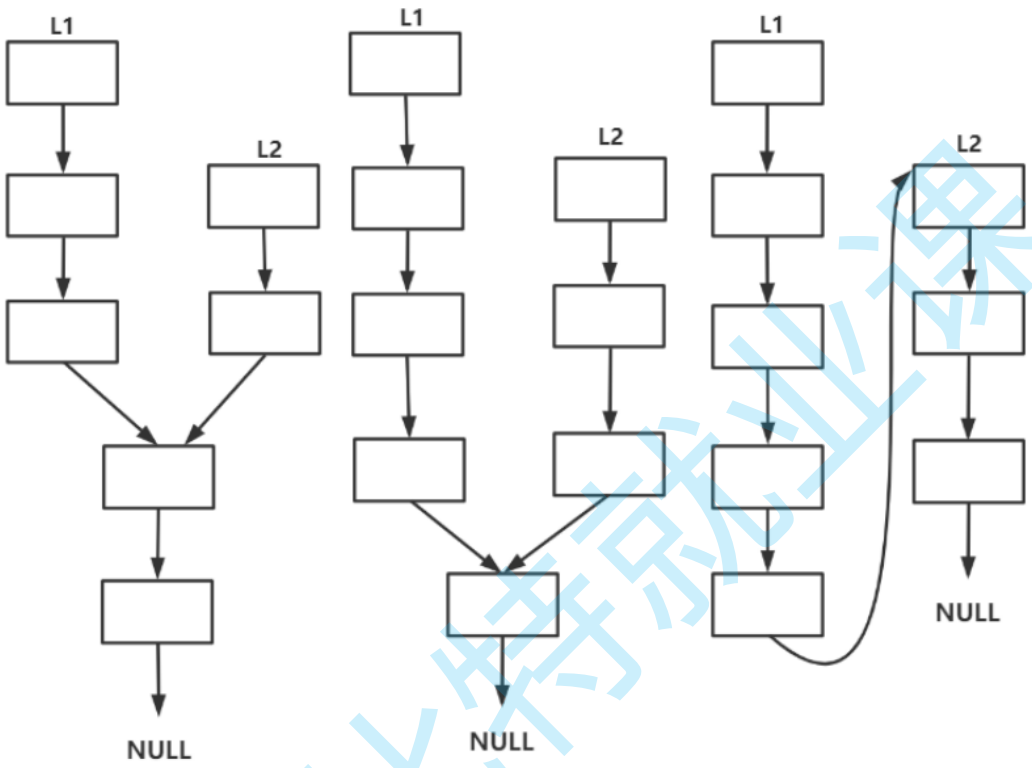
3.4.6 链表的回文结构

https://www.nowcoder.com/practice/d281619e4b3e4a60a2cc66ea32855bfa

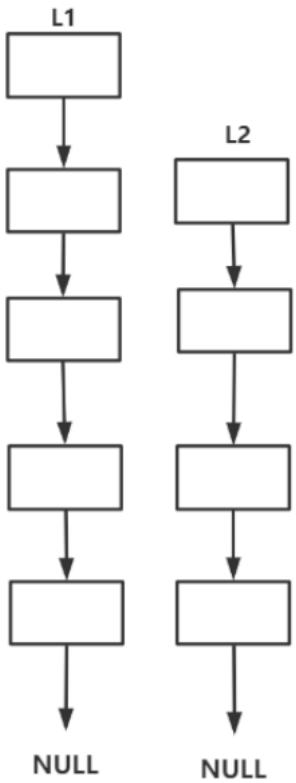
3.4.7 相交链表

https://leetcode.cn/problems/intersection-of-two-linked-lists/description/

相交：



不相交：



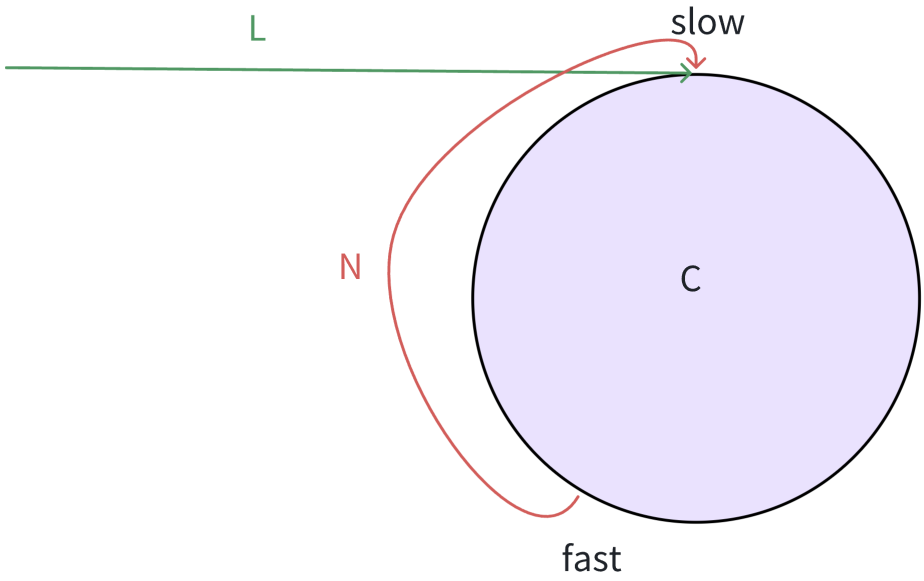
比特就业课

3.4.8 环形链表I

https://leetcode.cn/problems/linked-list-cycle/description/

> 💡 快慢指针
>
> 快慢指针，即慢指针一次走一步，快指针一次走两步，两个指针从链表起始位置开始运行，如果链表
>
> 带环则一定会在环中相遇，否则快指针率先走到链表的未尾

思考1：为什么快指针每次走两步，慢指针走一步可以相遇，有没有可能遇不上，请推理证明！



slow一次走一步，fast一次走2步，fast先进环，假设slow也走完入环前的距离，准备进环，此时fast和slow之间的距离为N，接下来的追逐过程中，每追击一次，他们之间的距离缩小1步

追击过程中fast和slow之间的距离变化：

N

N-1

N-2

N-3

.....

2

1

0 ⇨ 距离为0，即追上了

因此，在带环链表中慢指针走一步，快指针走两步最终一定会相遇。

思考2：快指针一次走3步，走4步，...n步行吗?

step1:

按照上面的分析，慢指针每次走一步，快指针每次走三步，此时快慢指针的最大距离为N，接下来的追逐过程中，每追击一次，他们之间的距离缩小2步

追击过程中fast和slow之间的距离变化：

比特就业课

	N为偶数	N为奇数
N	N	N
N-2	N-2	N-2
N-4	N-4	N-4
.....	.....	.....
	4	3
	2	1
	0	-1
	(追上)	（意味错过了,但是因为再环里面 相当于进入新一轮的追击,他们之间的 距离变成C-1(假设C是环的长度)

分析:

1、如果N是偶数，第一轮就追上了

2、如果N是奇数，第一轮追不上，快追上，错过了，距离变成-1，即C-1，进入新的一轮追击

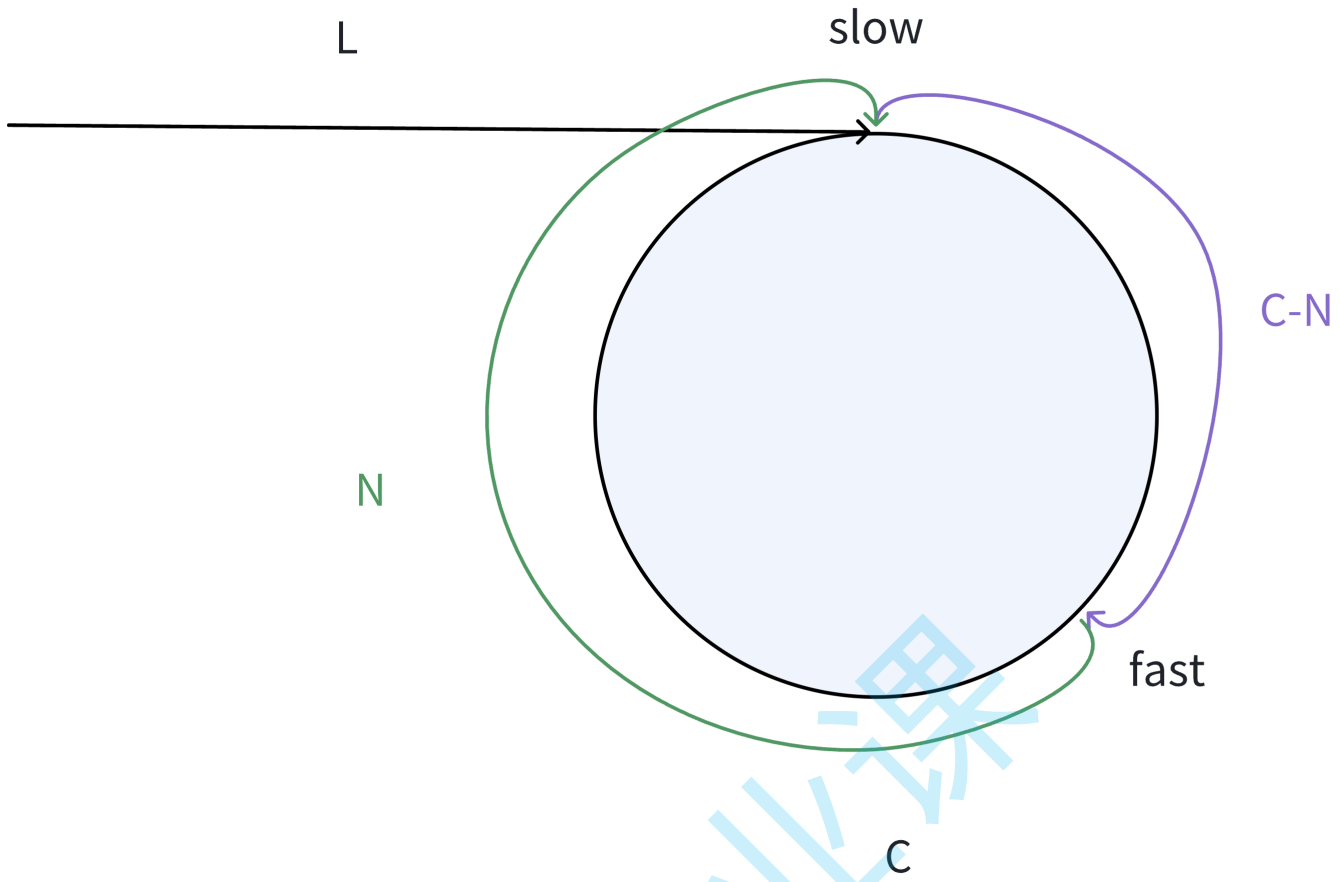
a、C-1如果是偶数，那么下一轮就追上了

b、C-1如果是奇数，那么就永远都追不上

总结一下追不上的前提条件：**N是奇数，C是偶数**

step2：

比特就业课



假设：

环的周长为C，头结点到slow结点的长度为L，slow走一步，fast走三步，当slow指针入环后，slow和fast指针在环中开始进行追逐，假设此时fast指针已经绕环x周。

在追逐过程中，快慢指针相遇时所走的路径长度：

`fast: L+xC+C-N`

`slow: L`

由于慢指针走一步，快指针要走三步，因此得出：`3 * 慢指针路程 = 快指针路程`，即：

$$3L = L + xC + C - N$$

$$2L = (x+1)C - N$$

对上述公式继续分析：由于偶数乘以任何数都为偶数，因此 $2L$ 一定为偶数，则可推导出可能得情况：

- 情况1：偶数 = 偶数 - 偶数
- 情况2：偶数 = 奇数 - 奇数

由step1中（1）得出的结论，如果N是偶数，则第一圈快慢指针就相遇了。

由step1中（2）得出的结论，如果N是奇数，则fast指针和slow指针在第一轮的时候套圈了，开始进行下一轮的追逐；当N是奇数，要满足以上的公式，则 `(x+1)C` 必须也要为奇数，即C为奇数，满足（2）a中的结论，则快慢指针会相遇

比特就业课

> 因此， step1 中的 **N是奇数，C是偶数** 不成立，既然不存在该情况，则快指针一次走3步最终一定也可以相遇。
>
> 快指针一次走4、5.....步最终也会相遇，其证明方式同上。

```c
typedef struct ListNode ListNode;
bool hasCycle(struct ListNode *head) {
    ListNode* slow,*fast;
    slow = fast = head;
    while(fast && fast->next)
    {
        slow = slow->next;
        int n=3; //fast每次走三步
        while(n--)
        {
            if(fast->next)
                fast = fast->next;
            else
                return false;
        }
        if(slow == fast)
        {
            return true;
        }
    }
    return false;
}
```

比特就业课

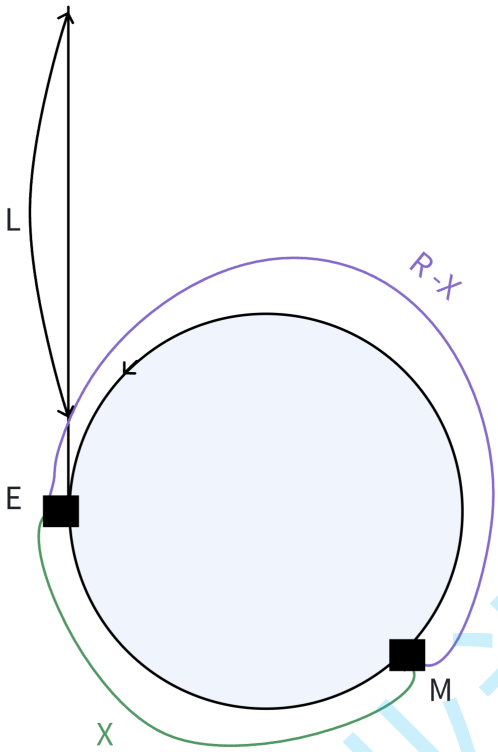
> 💡 提示
>
> 虽然已经证明了快指针不论走多少步都可以满足在带环链表中相遇，但是在编写代码的时候会有额外的步骤引入，涉及到快慢指针的算法题中通常习惯使用慢指针走一步快指针走两步的方式。

3.4.9 环形链表II

https://leetcode.cn/problems/linked-list-cycle-ii/description/

> 💡 结论
>
> 让一个指针从链表起始位置开始遍历链表,同时让一个指针从判环时相遇点的位置开始绕环运行,两个指针都是每次均走一步,最终肯定会在入口点的位置相遇。

证明以上结论



说明:

H为链表的起始点，E为环入口点，M与判环时候相遇点

设:

环的长度为R，H到E的距离为L，E到M的距离为 `X` ，则：M到E的距离为 `R-X`

在判环时,快慢指针相遇时所走的路径长度:

`fast: L+X + nR`

`slow: L+X`

注意:

1.当慢指针进入环时，快指针可能已经在环中绕了n圈了，n至少为1

因为：快指针先进环走到M的位置,，最后又在M的位置与慢指针相遇

2.慢指针进环之后，快指针肯定会在慢指针走一圈之内追上慢指针

因为：慢指针进环后,快慢指针之间的距离最多就是环的长度,而两个指针在移动时,每次它们至今的距离都缩减一步，因此在慢指针移动一圈之前快，指针肯定是可以追上慢指针的，而快指针速度是满指针的两倍，因此有如下关系是:

`2 * (L+X)=L+X+nR`

`L+X=nR`

`L=nR-X`

比特就业课

> `L = (n-1)R+(R-X)`
>
> (n为1,2,3,4.....，n的大小取决于环的大小，环越小n越大)
>
> 极端情况下，假设n=1，此时：`L=R-X`
>
> 即：一个指针从链表起始位置运行，一个指针从相遇点位置绕环，每次都走一步，两个指针最终会在入口点的位置相遇

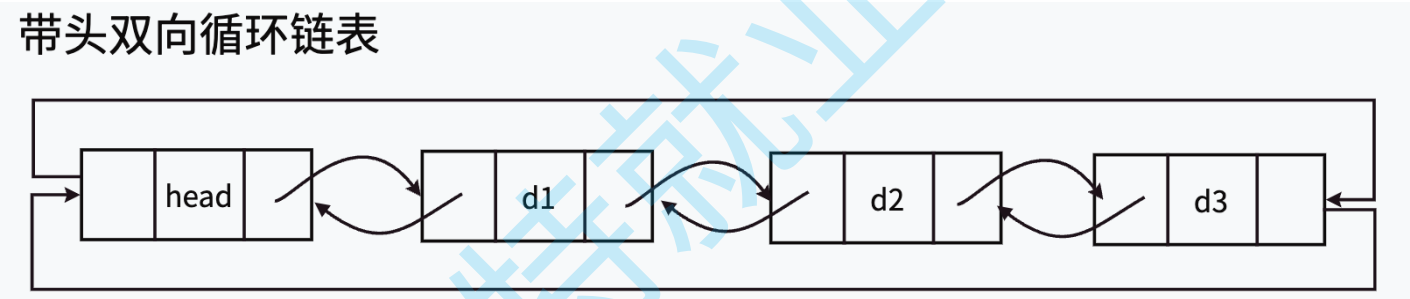
3.4.10 随机链表的复制

https://leetcode.cn/problems/copy-list-with-random-pointer/description/

> 更多链表算法刷题入口：
>
> 牛客网：https://www.nowcoder.com/exam/oj
>
> LeetCode：https://leetcode.cn/problems/copy-list-with-random-pointer/description/

4. 双向链表

4.1 概念与结构



> 注意：这里的“带头”跟前面我们说的“头结点”是两个概念，实际前面的在单链表阶段称呼不严谨，但是为了同学们更好的理解就直接称为单链表的头结点。

带头链表里的头结点，实际为“哨兵位”，哨兵位结点不存储任何有效元素，只是站在这里“放哨的”

4.2 实现双向链表

`List.c`

```
typedef int LTDataType;
typedef struct ListNode
{
    struct ListNode* next; //指针保存下一个结点的地址
    struct ListNode* prev; //指针保存前一个结点的地址
    LTDataType data;
}LTNode;

```

比特就业课

```
//void LTInit(LTNode** pphead);
LTNode* LTInit();
void LTDestroy(LTNode* phead);
void LTPrint(LTNode* phead);
bool LTEmpty(LTNode* phead);

void LTPushBack(LTNode* phead, LTDataType x);
void LTPopBack(LTNode* phead);

void LTPushFront(LTNode* phead, LTDataType x);
void LTPopFront(LTNode* phead);
//在pos位置之后插入数据
void LTInsert(LTNode* pos, LTDataType x);
void LTErase(LTNode* pos);
LTNode *LTFind(LTNode* phead,LTDataType x);
```

5. 顺序表与链表的分析

不同点	顺序表	链表（单链表）
存储空间上	物理上一定连续	逻辑上连续，但物理上不一定连续
随机访问	支持O(1)	不支持：O(N)
任意位置插入或者删除元素	可能需要搬移元素，效率低O(N)	只需修改指针指向
插入	动态顺序表，空间不够时需要扩容和空间浪费	没有容量的概念，按需申请释放，不存在空间浪费
应用场景	元素高效存储+频繁访问	任意位置高效插入和删除

比特就业课