顺序表

2023年11月19日 15:51

结构体：有三个内容，第一个内容是指针指向数组的内容，第二个是数组的大小，第三个内容是容量

```
typedef struct Seqlist {
    int* a;        // a是一个指针，类似数组的首元素的地址
    int size;    //有效数据
    int capacity;    //空间容量
}SL;
```

顺序表的本质就是数组：

初始化顺序表

将数组指针指向空
size设置成0
capacity设置成0

```
void SQInit(SQ* psl) {
    assert(psl->a != NULL);

    psl->a = NULL;
    psl->size = 0;
    psl->capacity = 0;
}
```

删除顺序表

断言结构体的内容不能是NULL，如果是就不需要destroy

如果psl->a等于NULL就不用做任何操作

1. 下面把psl->a 指向空
2. 把size和capacity设置成0

```
void SQDestroy(SQ* psl) {
    assert(psl);
    if (psl->a != NULL) {
        psl->a = NULL;
        free(psl->a);
        psl->size = 0;
        psl->capacity = 0;

    }
}
```

打印顺序表

顺序表的本质就是数组，所以直接打印数组就可以了

```
void SQCheckCapacity(SQ* psl) {
    assert(psl);
    for (int i = 0; i < psl->size; i++) {
        printf("%d ", psl->a[i]);
    }
    printf("\n");

}
```

检查容量

如果size和capacity相等，那么就需要扩容了

1. 检查psl->capacity是否是0，如果是0就把capacity的大小设定为4
2. 如果不是0；就开出之前容量的一倍的大小
3. 在数组a之后扩展一个新的内存块，用realloc的好处是

```
void SQCheckCapacity(SQ* psl) {
    assert(psl);
```

1. 检查psl->capacity是否是0，如果是0就把capacity的大小设定为4
2. 如果不是0；就开出之前容量的一倍的大小
3. 在数组a之后扩展一个新的内存块，用realloc的好处是当内存开辟失败的时候，会在堆上重新找一块完整的地方，为了防止内容丢失，所以将开辟后的内存放置在一个临时的结构体中，然后将内容赋值到psl->a的内容中

```
void SQCheckCapacity(SQ* psl) {
    assert(psl);
    if (psl->size == psl->capacity)
    {
        int newcapacity = psl->capacity == 0 ? 4 : psl->capacity * 2;
        SLDataType* tmp = (SLDataType*)realloc(psl->a, sizeof(SLDataType) * newcapacity);
        if (tmp == NULL) {
            perror("realloc fail");
            return;
        }
        psl->a = tmp;
        psl->capacity = newcapacity;
    }

}
```

尾插

尾插就是在最后的内容中直接插入到数组的最后然后++size

```
void SQPushBack(SQ* psl, SLDataType x) {
    assert(psl);      保证传来的不是一个空指针或者是野指针

    SQCheckCapacity(psl);      检查SQ的内容是否够

    psl->a[psl->size] = x;
    psl->size++;

}
```

头插

头插的本质是：

1. 将有的有数组都右移一个
   a. 右移的过程中不能从前往后移，不然会产生覆盖
   b. 需要从后往前看，先找到到最后的内容，然后--
2. 将要插入的数插到第一个

```
void SQPushFront(SQ* psl, SLDataType x) {
    assert(psl);
    SQCheckCapacity(psl);

    int end = psl->size - 1;
    while (end >= 0) {
        psl->a[end + 1] = psl->a[end];
        end--;
    }
    psl->a[0] = x;
    psl->size++;
}
```

尾删

直接将size--，在打印的时候就直接不用打印了

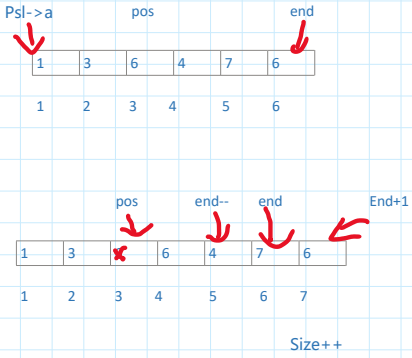
```
void SQPopBack(SQ* psl) {
    assert(psl);
    assert(psl->size > 0);

    psl->size--;
}
```

头删

将数组的内容向前移动一格然后覆盖掉第一个元素
再将size--

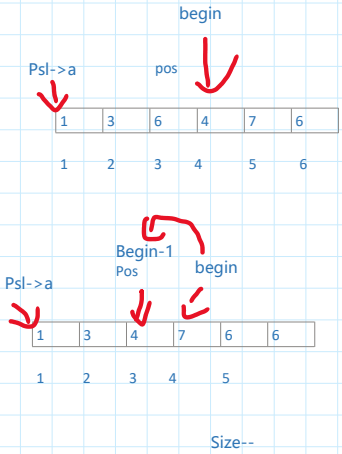
```
void SQPopFront(SQ* psl) {
    assert(psl);
    assert(psl->size > 0);
    int begin = 1;

    while (begin < psl->size) {
        psl->a[begin - 1] = psl->a[begin];
        begin++;
    }
    psl->size--;
}
```

任意位置插入




```
void SQInsert(SQ* psl, int pos, SLDataType x) {
    assert(psl);

    SQCheckCapacity(psl);

    int end = psl->size-1;
    while (end < pos) {
        psl->a[end + 1] = psl->a[end];
        end--;
    }
    psl->a[pos] = x;
    psl->size++;

}
```

任意位置删除



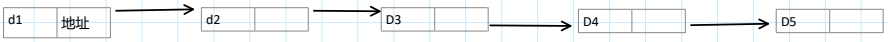

```
void SQErase(SQ* psl, int pos) {
    assert(psl);
    assert(psl->size > 1);
    int begin = pos+1;
    while (begin < psl->size) {
        psl->a[begin - 1] = psl->a[begin];
        begin++;
    }
    psl->size--;
}
```

单向链表的实现

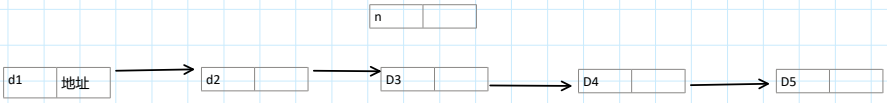
2023年11月14日 6:00

链表是什么

链表的主要是写一个结构体，结构体的内容有两个，第一个储存的是内容，也就是我们需要保存的数据内容，第二个保存的是下一个的地址



结构体的实现



SLNDataType:讲内容int改一个命名的好处是方便以后更改链表里的元素类型

```c
#pragma once
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef int SLNDataType;

typedef struct SListNode {
    SLNDataType val;
    struct SListNode* next;
    //结构体不能嵌套结构体，这里的指针是指向下一个结点的

}SLNode;

void SLTPrint(SLNode* phead);
void SLTPushBack(SLNode** pphead, SLNDataType x);
void SLTPushFront(SLNode** pphead, SLNDataType x);

void SLTPopBack(SLNode** pphead);
void SLTPopFront(SLNode** pphead);

SLNode* SLTFind(SLNode* pphead, SLNDataType x);

//在pos的前面插入

void SLTInsert(SLNode** pphead,SLNode* pos, SLNDataType x);

//删除pos位置
SLNode* SLTFind(SLNode* phead, SLNDataType x);

//删除节点
void SLTErase(SLNode** pphead, SLNode* pos);


//摧毁SLT
SLNode* SLTDestroy(SLNode** pphead);

//在后pos后加入
//后面插入后面删除
void SLTInsertAfter( SLNode* pos, SLNDataType x);

void SLTEraseAfter( SLNode* pos);
```

结构体的嵌套不能直接传结构体，因为结构体的大小取决于结构体内的变量的大小，但是如果嵌套结构体的话，结构体的大小就无法确定，但是如果是下一个结构体的指针，那结构体的大小就可以确定了

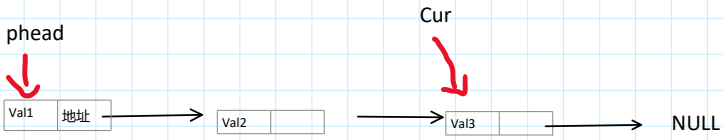
创建新的结点

```c
SLNode* CreateNode(SLNDataType x) {
    SLNode* newnode =(SLNode*) malloc(sizeof(SLNode));
    if (newnode == NULL) {
        perror("malloc fail");
        exit(-1);
    }
    newnode->val = x;
    newnode->next = NULL;
    return newnode;
}
```

给新的节点找开辟一个新的空间

把形参方进这个结构体的val参数中
产生的结点的 的内容一开始为空

打印链表



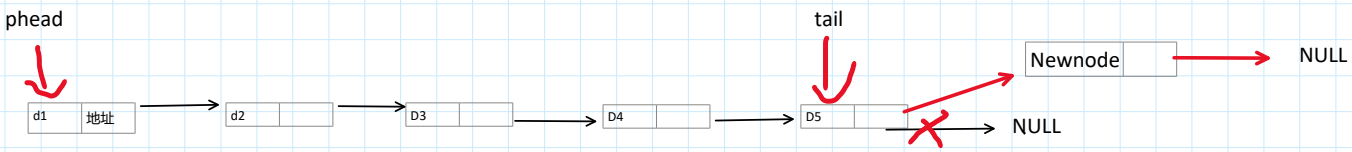

```c
void SLTPrint(SLNode* phead) {
    SLNode* cur = phead;
    while (cur != NULL) {
        printf("%d->", cur->val);
        cur = cur->next;
    }
    printf("NULL\n");
}
```

传第一个结构体的指针

创建一个结构体=

指向Val的值进行打印

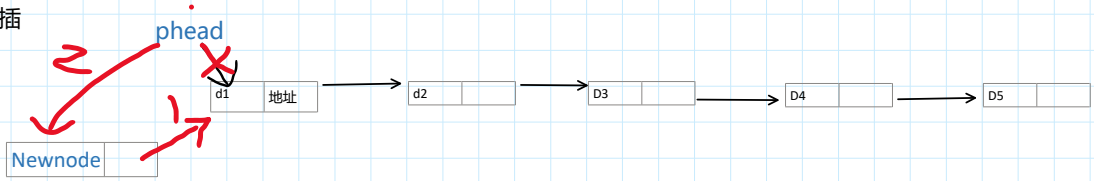
尾插



用一个tail来依次遍历每个地址，找到最后的一个变量，直到找到NULL就代表最后一个的数

```c
void SLTPushBack(SLNode** pphead, SLNDataType x) {
    assert(pphead);
    SLNode* newnode = CreateNode(x);
    if (*pphead == NULL) {//改变的结构体的指针，所以用结构体的指针的地址
        *pphead = newnode;//但这些变量都是局部变量，
        //这个函数结束就会销毁，哪怕他是指针，存储的是地址，但是是局部变量的地址，所以要在函数内取二级指针
    }
    else {
        SLNode* tail = *pphead;//先找一个尾部的指针，指向开头，往后寻找尾部
        //先要找到最后的结点
        //下面改变的是结构体的成员
        while (tail->next != NULL) {//不能tail! =NULL，是这个tail的地址不是空，但是我们需要的是tail->next的地址不为空
            tail = tail->next;
        }


        tail->next = newnode;
    }

}
```

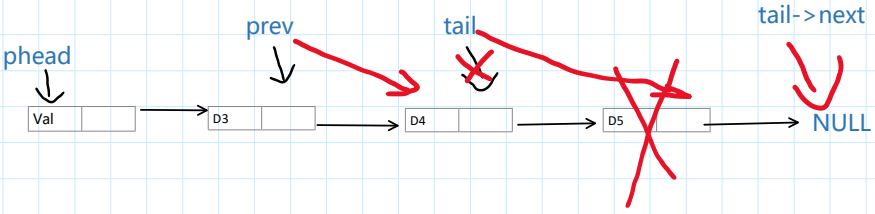
头插




```c
void SLTPushFront(SLNode** pphead, SLNDataType x) {
    assert(pphead);
    SLNode* newnode = CreateNode(x);

    newnode->next = *pphead;
    *pphead = newnode;
}
```

先把newnode的内容变成phead的地址
如果先赋值phead里的内容就找不到newnode了
只能先创建一个结点，然后将这个结点指向头地址，pphead是储存head地址的指针

尾删

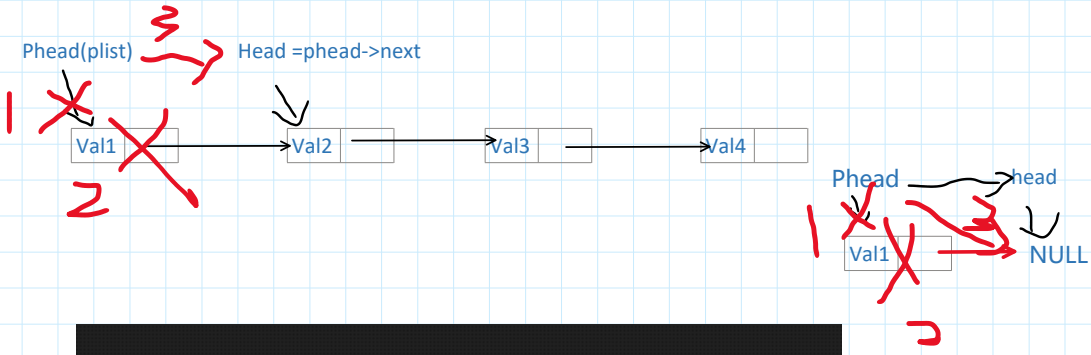


```c
void SLTPopBack(SLNode** pphead) {
    assert(pphead);
    //if (*pphead == NULL)
    //  return;

    assert(*pphead);
    if ((*pphead)->next == NULL) {
        free(*pphead);
        *pphead = NULL;
    }
    else {
        SLNode* prev = NULL;
        SLNode* tail = *pphead;//先找一个尾部的指针，指向开头，往后寻找尾部
    //先要找到最后的结点
    //下面改变的是结构体的成员
        while (tail->next != NULL) {//不能tail!=NULL，是这个tail的地址不是空，但是我们需要的是tail->next的地址不为空
            prev = tail;
            tail = tail->next;
        }
        free(tail);
        tail = NULL;
        prev->next = NULL;
    }
}
```

如果找到phead里下一个地址直接指向的是NULL，
直接可以free这个指针

当tail的next指向的值为NULL的时候，代表了这个tail已经
走到了最后的值了，这个时候在把tail的内容释放，在把
prev的next指向NULL保证链表的完整就可以了

头删

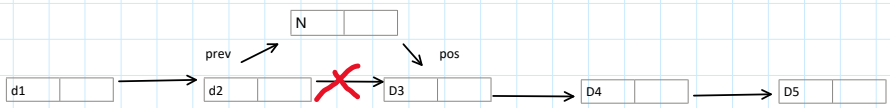



```c
void SLTPopFront(SLNode** pphead) {
    assert(pphead);
    assert(*pphead);
    SLNode* head = (*pphead)->next;
    free(*pphead);
    (*pphead) = head;
}
```

哪怕只有一个节点也可以实现

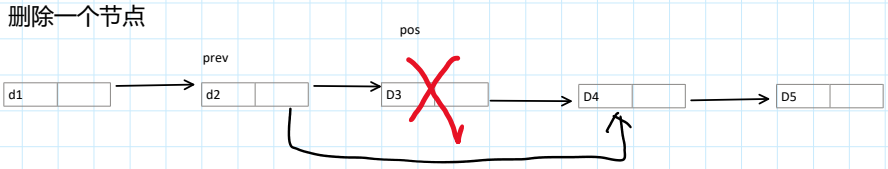
不能先free再赋值
free完就不存在了

在pos前插入一个节点



```c
void SLTInsert(SLNode** pphead, SLNode* pos, SLNDataType x) {

    //严格限定pos一定是链表里面的一个有效节点
    assert(pphead);
    assert(pos);
    assert(*pphead);

    //较为宽松
    //要么都是空，要么都不是空
    //assert((!pos && !(*pphead))||(pos && *pphead));


    if (*pphead == pos) {
        //头插                    如果直接在第一个位置进行插入就等于头插
        SLTPushFront(pphead, x);
    }
    else {
        SLNode* prev = *pphead;
        while (prev->next != pos) {
            prev = prev->next;     创建一个prev，prev的next先遍历到pos的点
        }
        SLNode* newnode = CreateNode(x);
        prev->next = newnode;                  然后创建一个新的结点
        newnode->next = pos;
    }
}
```

删除一个节点



```c
void SLTErase(SLNode** pphead, SLNode* pos) {
    assert(pphead);
    assert(pos);
    assert(*pphead);

    if (*pphead == pos) {
        SLTPopFront(pphead);
    }
    else {
        SLNode* prev = *pphead;
        while (prev->next != pos) {
            prev = prev->next;
        }

        prev->next = pos->next;
        free(pos);
        pos = NULL;
    }

}
```

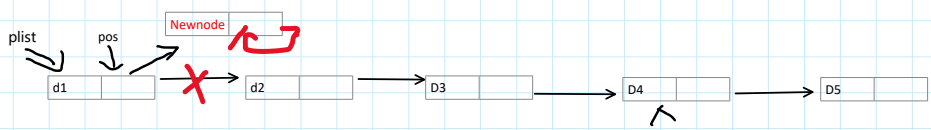
先用一个prev的next找到pos的点

然后把prev->next指向pos下一个数

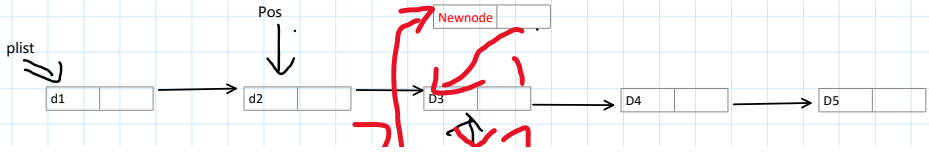
然后free 目标地址

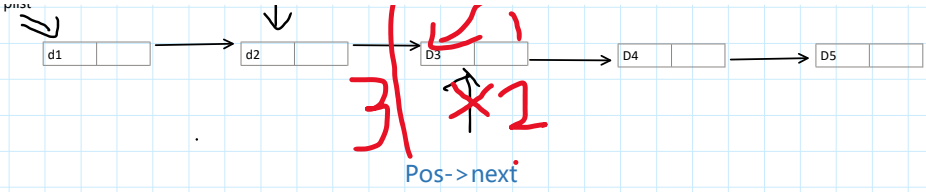
加入到pos后面的节点

错误的方式




```c
void SLTInsertAfter(SLNode* pos, SLNDataType x) {
    assert(pos);

    SLNode* newnode = CreatNode(x);

    pos->next = newnode;
    newnode->next = pos->next;
}
```






```
void SLTInsertAfter(SLNode* pos, SLNDataType x) {
    assert(pos);

    SLNode* newnode = CreatNode(x);
    newnode->next = pos->next;
    pos->next = newnode;
}
```

Newnode->next指向了原来的pos的内容

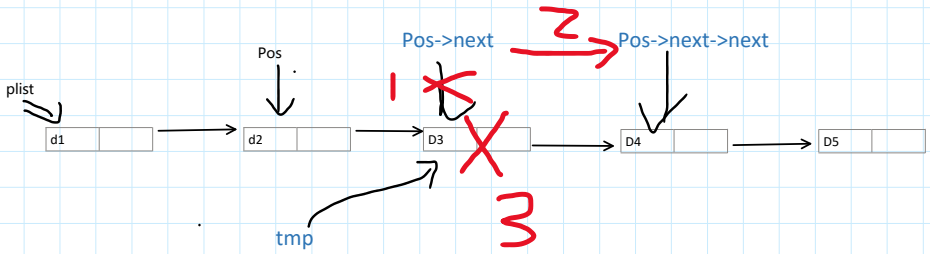
Pos->next直接连向了newnode的地址

最终变成

pos， pos->next指向newnode

newnode，newnode->next指向原本pos->next 的位置

删除pos后




```
void SLTEraseAfter(SLNode* pos) {
    assert(pos->next);
    SLNode* tmp = pos->next;
    pos->next = pos->next->next;
    free(tmp);
    tmp = NULL;
}
```

先建立一个临时结构体指向pos后的内容

然后把pos->next的地址直接换成pos->next->next的地址，然后free掉临时结构体就完成了，因为这都是临时指针在函数结束后，名字就不重要了

链表OJ题

2023年11月14日 22:58

876. 链表的中间结点

简单

相关企业

给你单链表的头结点 head ，请你找出并返回链表的中间结点。

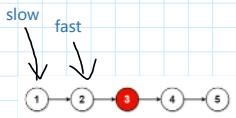
如果有两个中间结点，则返回第二个中间结点。

示例 1：

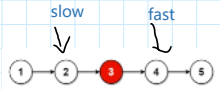
输入： head = [1,2,3,4,5]
输出： [3,4,5]
解释： 链表只有一个中间结点，值为 3 。

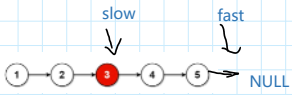
示例 2：

输入： head = [1,2,3,4,5,6]
输出： [4,5,6]
解释： 这链表有两个中间结点，值分别为 3 和 4 ，返回第二个结点。

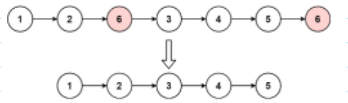


slow走一格fast走两格
然后返回slow的地址就是中间数

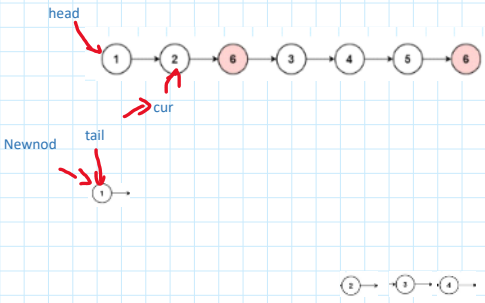




oj题目



遍历原链表
把不是val的节点，尾插到新链表




```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* removeElements(struct ListNode* head, int val) {
    struct ListNode* newhead = NULL,* tail = NULL;
    struct ListNode* cur = head;
    while(cur){

        if (cur->val != val) {
            if (tail == NULL) {
                newhead = tail = cur;
            }
            else {
                tail->next = cur;
                tail = tail->next;
            }
            cur = cur->next;
        }
        else {
            struct ListNode* tmp = cur;
            cur = cur->next;
            free(tmp);
        }
    }
    if(tail)
        tail->next =NULL;

    return newhead;
}
```

CM11 链表分割

[编程题]链表分割

时间限制：C/C++ 3秒，其他语言6秒　空间限制：C/C++ 32M，其他语言64M　热度指数：100303

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。

说明：本题已包含复杂数据结构ListNode，点此查看相关信息

36123787 5

31236787

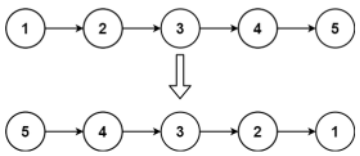
把<val的尾插到第一个链表
把>=val的尾插到第二个链表

206. 反转链表

简单 相关标签 相关企业 Aa

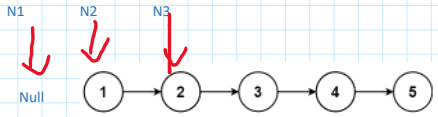
给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。

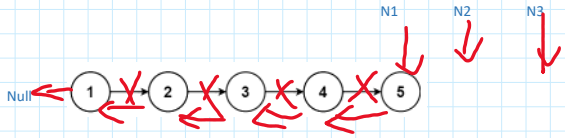
示例 1:



把指针掉头就可以了

```
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){

        n2->next = n1;
        n1 = n2;
        n2 = n3;
        if(n3)
            n3 = n3->next;
    }
    return n1;
}
```

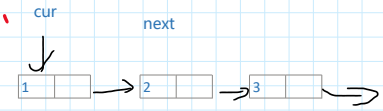




方法二

头插

1、

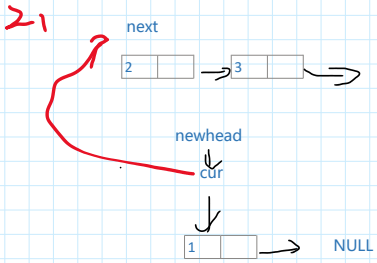


newhead

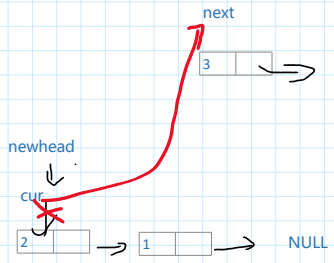
NULL

```
struct ListNode* reverseList(struct ListNode* head) {
    struct ListNode* cur = head;
    struct ListNOde* newhead = NULL;
    while(cur){
        struct Listnode * next = cur->next;
        cur->next = newhead;
        newhead = cur;
        cur = next;
    }
    return newhead;
}
```

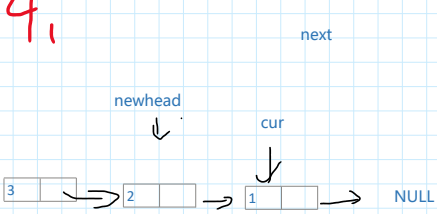
2、



3、



4、



链表中倒数第k个结点

题目　题解(112)　讨论(1k)　排行

时间限制：1秒　空间限制：64M

知识点：链表　双指针

描述

输入一个链表，输出该链表中倒数第k个结点。

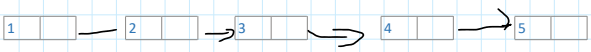
示例1

输入：　1,{1,2,3,4,5}

返回值：{5}

```c
/**
 * struct ListNode {
 *  int val;
 *  struct ListNode *next;
 * };
 */

/**
 *
 * @param pListHead ListNode类
 * @param k int整型
 * @return ListNode类
 */
struct ListNode* FindKthToTail(struct ListNode* pListHead, int k ) {
    struct ListNode* slow = pListHead, *fast = pListHead;
    while(k--){
        if(fast == NULL)
            return NULL;
        fast = fast -> next;
    }
    while (fast){
        slow =slow-> next;
        fast = fast-> next;
    }

    return slow;
    // write code here
}
```

slow

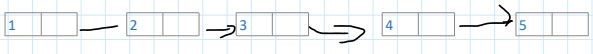


fast

k==3

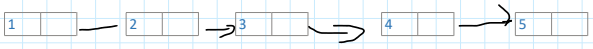
两种都可以

slow



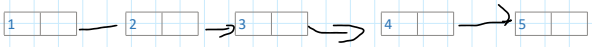
fast

slow



fast

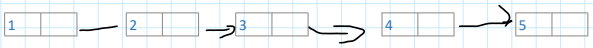
slow



fast

先走k步

slow



fast

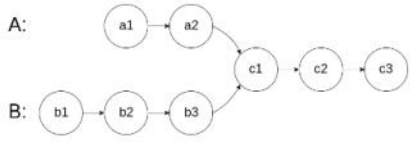
先走k-1步

160. 相交链表

简单　相关标签　相关企业　Aa

给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null 。

图示两个链表在节点 c1 开始相交：



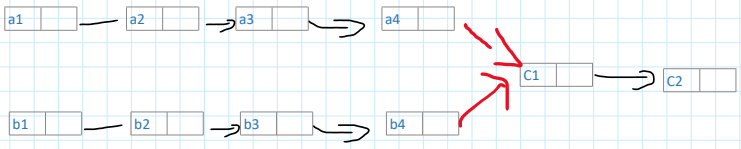
思路一：暴力求解

A链表的所有结点依次取b链表找一遍

用结点的地址作比较

时间复杂度O(N^2)

1. 判断是否相交
2. 找相交点



思路二 优化->时间复杂度到O(N)

分别找到尾结点

如果尾结点相同就相交

如果尾结点不同就不相交

找到相交的结点，就是将全部都走一遍找到每个链表的结点

```c
struct ListNode *getIntersectionNode(struct ListNode *headA, struct ListNode *headB) {
    struct ListNode* curA= headA, *curB= headB;
    int lenA = 1;
    int lenB = 1;

    //找尾结点，顺便找差值
    while(curA->next){
        lenA++;
        curA=curA->next;


    }
    while(curB->next){
        lenB++;
        curB=curB->next;


    }
    if(curA != curB){
        return NULL;
    }

    int n = abs(lenA-lenB);
    struct ListNode* longlist = headA, *shortlist =headB;
    if(lenB > lenA){
        longlist =headB;
        shortlist = headA;
    }

    //长的先走差距步
    while(n--){
        longlist = longlist->next;

    }
    while(longlist !=shortlist){
            longlist =longlist->next;
            shortlist = shortlist->next;
    }
```

如果尾结点相同就相交
如果尾结点不同就不相交

找到相交的结点，就是将全部都走一遍找到每个链表的结点
的数量，然后找到差值，然后让长的先走，然后再一起走

```
    while(longlist !=shortlist){
        longlist =longlist->next;
        shortlist = shortlist->next;
    }

    return longlist;
}
```

141. 环形链表

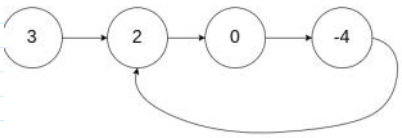
简单 相关标签 相关企业 Aa

给你一个链表的头节点 head ，判断链表中是否有环。

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，评测系统内部使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。**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。

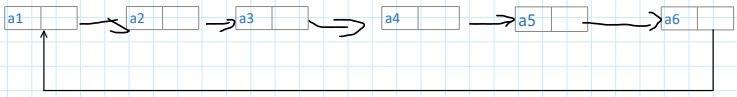
如果链表中存在环，则返回 true 。否则，返回 false 。

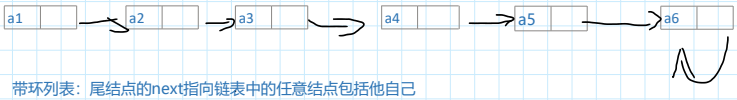
示例 1：



输入： head = [3,2,0,-4], pos = 1
输出： true
解释： 链表中有一个环，其尾部连接到第二个节点。

```
bool hasCycle(struct ListNode *head) {
    struct ListNode* fast = head, *slow = head;
    while(fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;

            if(slow==fast){
                return true;
            }

    }
    return false;
}
```





带环列表：尾结点的next指向链表中的任意结点包括他自己

追击问题：快慢指针，一个一次走一步，一个一次走两步，
然后当快指针走进循环了，慢指针才走了一般
当慢指针走进循环，但是快指针已经再循环内了
总有会追上，如果相等就代表有环

1.slow一次走一步，fast一次走两步

假设slow进环时
fast和slow之间的距离为N

slow进环后，fast和slow的距离变化，每次追击距离缩小1
距离变化， n->n-1->n-2...3->2->1->0

2.slow一次走一步，fast一次走三步

假设slow进环时
fast和slow之间的距离为N

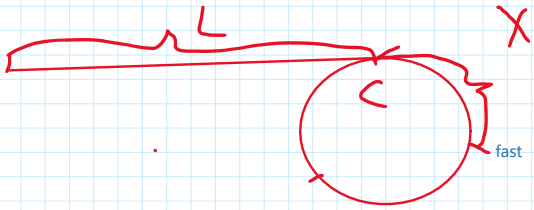
slow进环后，fast和slow的距离变化，每次追击距离缩小2
距离变化，如果N时偶数 n>n-2->n-4...4->2->0
如果N时奇数 n>n-2->n-4...3->1->-1 ->-3 ->-5

距离是-1代表，开始新的一轮追击，这个距离是C-1（假设c环的长度）
如果C-1是奇数，死循环
如果C-1是偶数，第二轮会追上

如果N是奇数，c是偶数时就追不上，这个条件是否成立？

3.slow一次走n步，fast一次走m步一定能相遇？ m>n>1

距离缩小m-n（>=1的整数）
N%（m-n）==0



起点到环入口点：L
入口点到相遇点：x
环的长度：C

从开始相遇时slow走的距离: L+X
从开始到相遇时fast走的距离: L+n*C+X (slow进环前，fast已经在环里转了n圈)

```
struct ListNode *detectCycle(struct ListNode *head) {
    struct ListNode* fast = head, *slow = head;
    while(fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;

            if(slow==fast){
                struct ListNode* meet = slow;
                while(head != meet){
```

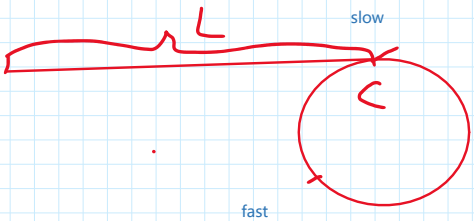
fast路程 = slow路程*2
2*(L + X) = L+ n*C + X
L + X = n*C
L = n*C -X
L = (n-1) *C -(C-X)

结论：一个指针从相遇点走，他们会在入口点相遇

```c
struct ListNode *detectCycle(struct ListNode *head) {
    struct ListNode* fast = head, *slow = head;
    while(fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;

            if(slow==fast){
                struct ListNode* meet = slow;
                while(head != meet){
                    head = head->next;
                    meet = meet->next;
                }
                return meet;
            }

    }
    return false;
}
```

2*(L + X) = L+ n*C + X
L + X = n*C
L = n*C -X
L = (n-1) *C -(C-X)

结论：一个指针从相遇点走，他们会在入口点相遇



起点到环入口点：L
slow入环的时候， fast-slow的距离是N
环的长度：C

Slow入环的时候
慢指针走的路程：L
快指针走的路程：L + n*C - N (n>=1)

快指针走的路程是慢指针的3倍
3L=L + n*C - N
2L=n*C - N

2L是偶数
n*C是偶数
N是就不可能是奇数

如果N是奇数，c是偶数时就追不上，这个条件是否成立?

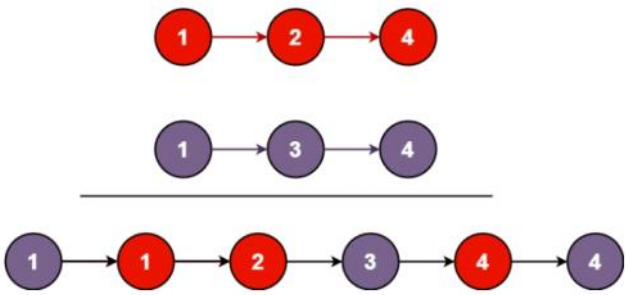
只能选一个

21. 合并两个有序链表

简单 相关标签 相关企业 Aa

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。

示例 1：



输入： l1 = [1,2,4], l2 = [1,3,4]
输出： [1,1,2,3,4,4]

取小的尾插

OR36 链表的回文结构

题目 题解(47) 讨论(172) 排行

较难 通过率：21.68% 时间限制：3秒 空间限制：32M

知识点： 链表 栈

描述

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。
测试样例：

```
1->2->2->1
```

返回：true

1->2->2->1
1 2
1->2->2->1

找到中间结点
然后把后半段逆置
然后比对

```cpp
class PalindromeList {
public:

    struct ListNode* reverseList(struct ListNode* head) {
        struct ListNode* cur = head;
        struct ListNode* newnode = NULL;
        while (cur) {
            struct ListNode* next = cur -> next;

            cur -> next = newnode;
            newnode = cur;
            cur = next;
        }
        return newnode;
    }

    struct ListNode* middleNode(struct ListNode* head) {
        struct ListNode* slow = head, * fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }
    bool chkPalindrome(ListNode* head) {
        struct ListNode* mid = middleNode(head);
        struct ListNode* rhead = reverseList(mid);

        while(head && rhead){
            if(head->val != rhead ->val)
                return false;
            head = head->next;
            rhead=rhead->next;

        }
        return true;
    }
};
```

138. 随机链表的复制

中等 相关标签 相关企业 提示 Aa

给你一个长度为 n 的链表，每个节点包含一个额外增加的随机指针 random ，该指针可以指向链表中的任何节点或空节点。

构造这个链表的 深拷贝。 深拷贝应该正好由 n 个 全新 节点组成，其中每个新节点的值都设为其对应的原节点的值。新节点的 next 指针和 random 指针也都应指向复制链表中的新节点，并使原链表和复制链表中的这些指针能够表示相同的链表状态。**复制链表中的指针都不应指向原链表中的节点 。**

例如，如果原链表中有 X 和 Y 两个节点，其中 X.random --> Y 。那么在复制链表中对应的两个节点 x 和 y ，同样有 x.random --> y 。

返回复制链表的头节点。

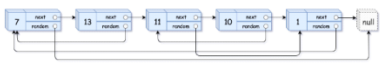
用一个由 n 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 [val, random_index] 表示：

- val：一个表示 Node.val 的整数。
- random_index：随机指针指向的节点索引（范围从 0 到 n-1）；如果不指向任何节点，则为 null 。

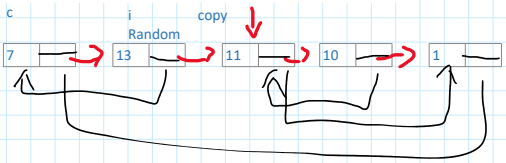
你的代码 只 接受原链表的头节点 head 作为传入参数。

单链表叠加一个随机指针
随机指向链表任意结点或者空

cur

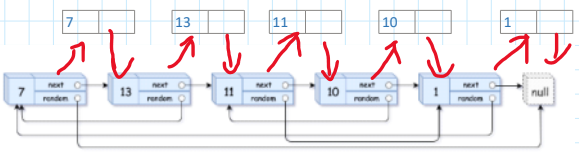


random是原链表第i个

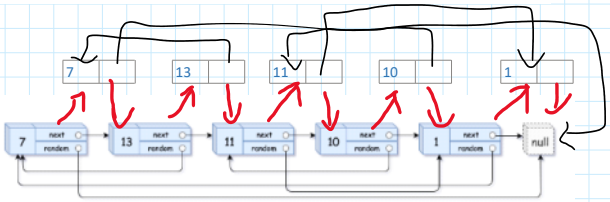


找copy链表中的第i个

找每个random都是N
合计O(N^2)

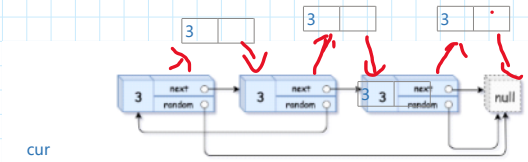


拷贝结点都在原结点的后面

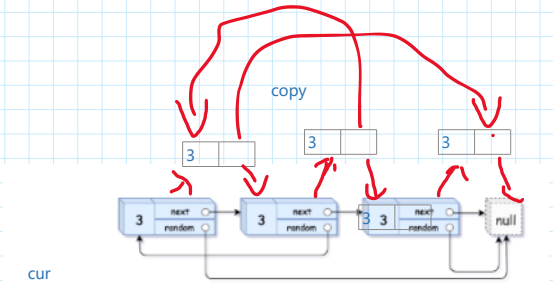


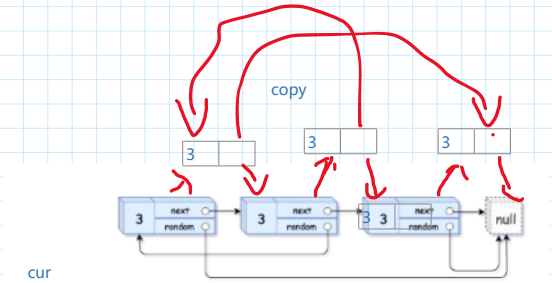
copy

1.把拷贝节点插入在原结点的后面




```
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->next = cur->next;
      cur->next = copy;

      cur = cur->next->next;

    }
}
```

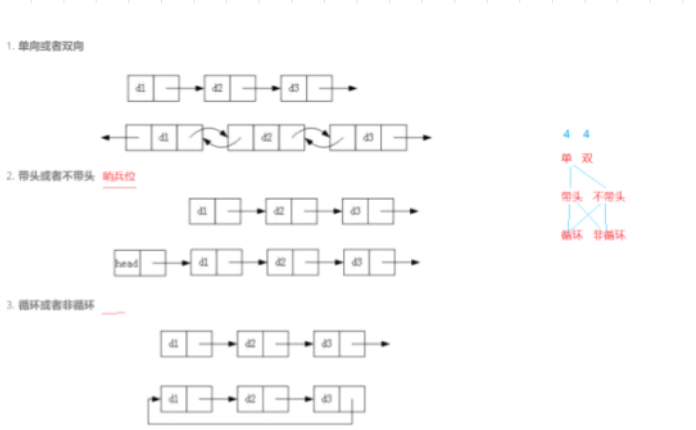


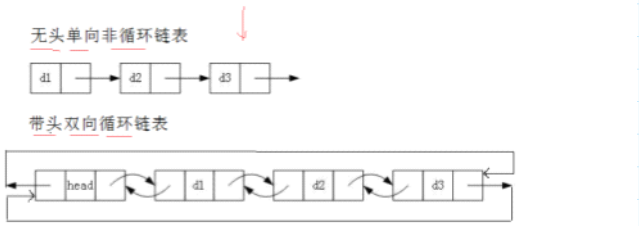


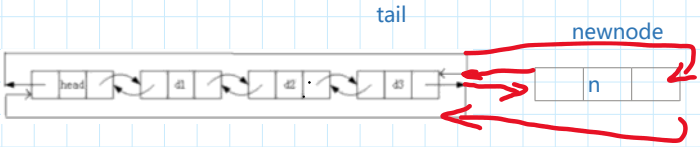
3. copy结点解下来尾插

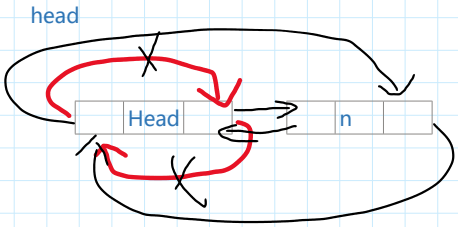
双链表

2023年11月17日 18:45

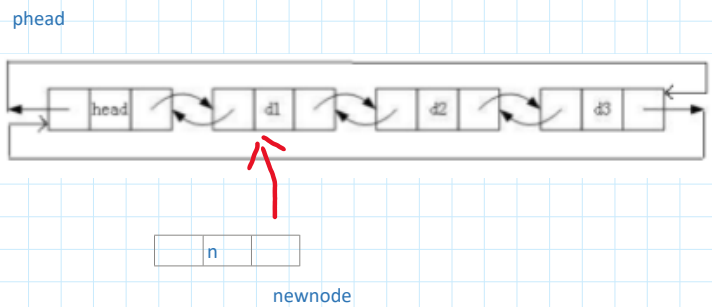






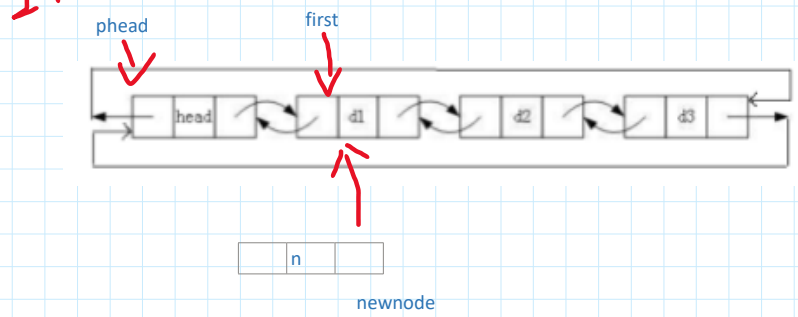


1.



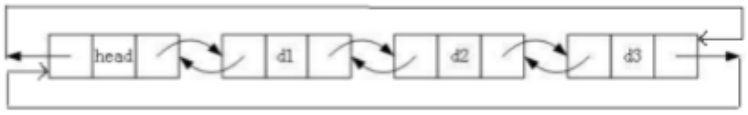

```
void LTPushFront(LTNode* phead, LTDataType x) {
    assert(phead);
    LTNode* newnode = CreatLTNode(x);

    newnode->next = phead->next;
    phead->next->prev = newnode;

    phead->next = newnode;
    newnode->prev = phead;
}
```

1、

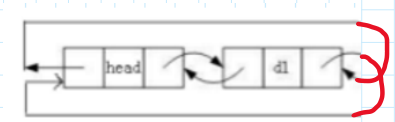



```
void LTPushFront(LTNode* phead, LTDataType x) {
    assert(phead);
    LTNode* newnode = CreatLTNode(x);
    LTNode* first = phead->next;

    phead->next = newnode;
    newnode->next = first;
    newnode->prev = phead;
    first->prev = newnode;
}
```

popfront

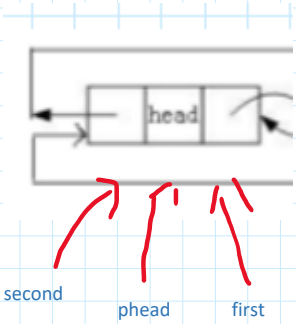


phead first sencond



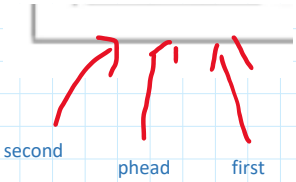
second phead first

```
void LTPopFront(LTNode* phead) {
    assert(phead);
    LTNode* first = phead->next;
    LTNode* second = first->next;
    phead->next = second;
    second->prev = phead;
    free(first);
    first = NULL;
}
```

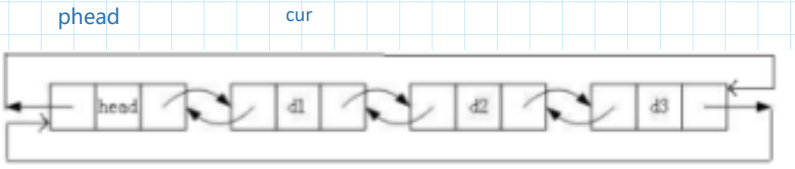


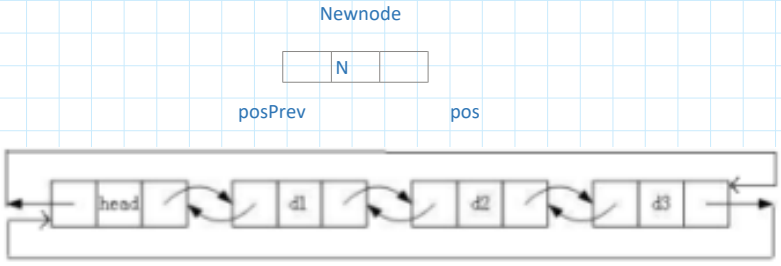
当只有哨兵位的时候他们都指向一个结构体，如果free这个结构体，phead就会变成野指针
所以要加一个assert

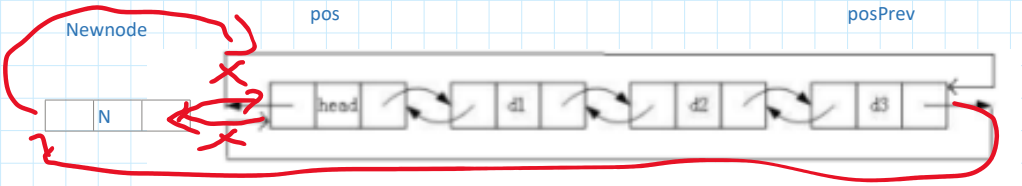
```
void LTPopFront(LTNode* phead) {
    assert(phead);

    assert(phead->next != phead);
    LTNode* first = phead->next;
    LTNode* second = first->next;
    phead->next = second;
    second->prev = phead;
    free(first);
    first = NULL;

}
```

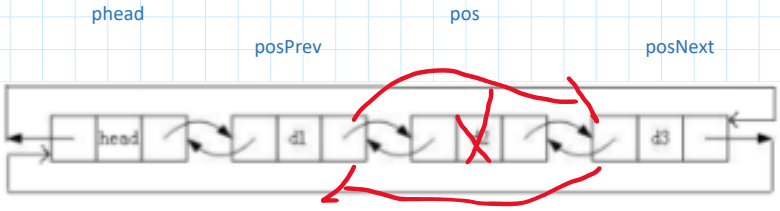



```c
void LTPopFront(LTNode* phead) {
    assert(phead);

    assert(phead->next != phead);
    LTNode* first = phead->next;
    LTNode* second = first->next;
    phead->next = second;
    second->prev = phead;
    free(first);
    first = NULL;

}
```









链表（双向）优势：

1. 任意位置插入删除都是O(1)
2. 按需申请释放，合理利用空间、不存在浪费

问题：

1. 下标随机访问不方便，O(N)

顺序表问题：

1. 头部或者中间插入删除效率低，要挪动数据。O(N)
2. 空间不够需要扩容，扩容一定的消耗，且可能存在一定的空间浪费
3. 只适合尾插尾删

优势（物理内存延续）

1. 支持下标的随机访问O(N)

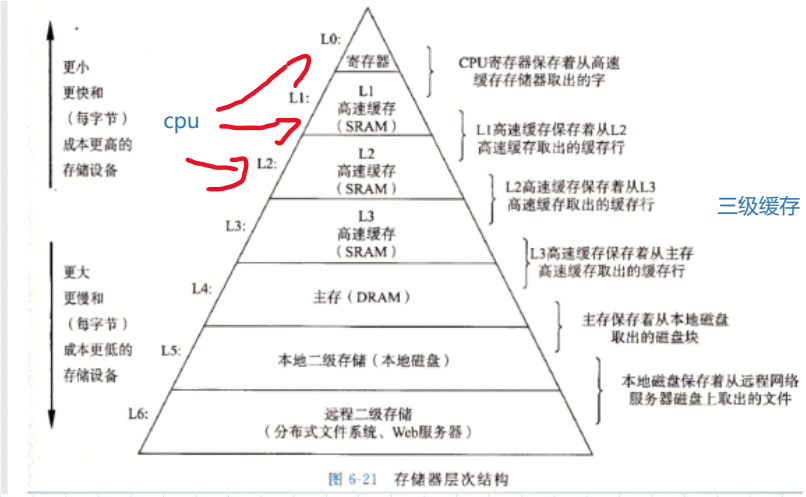


图 6-21 存储器层次结构

cpu

三级缓存

存储 内存 硬盘 不带电存储（读写速度较慢）

硬盘：磁盘和固态

带电存储（读写速度快）

运算 CPU GPU 内存

栈

2023年11月18日 14:18

栈底　　栈顶　　栈一般使用数组栈

如何实现栈?
1.数组栈
2.链式栈

1	2	3	4	5

双向链表，栈顶可以是尾，也可以是头
一般用单链表实现

1 → 2 → 3 → 4

栈顶

top

空的
Top == -1

Push
++top;
a[top] = x

Top

1

一个数据
Top==0
指向栈顶元素

top

空的
Top == 0

Push
a[top] = x
++top;

Top

1

一个数据
Top==1
指向栈顶元素的下一个位置

20. 有效的括号

简单 相关标签 相关企业 提示 Aa

给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串 s ，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

示例 1：

> **输入：** s = "()"
> **输出：** true

示例 2：

> **输入：** s = "()[]{}"
> **输出：** true

示例 3：

> **输入：** s = "(]"
> **输出：** false

1.左括号入栈
2.右括号，取栈顶左括号匹配

1. 数量匹配
2. 顺序匹配

```c
bool isValid(char* s) {
    ST st;
    STInit(&st);
    while(*s){
      if(*s == '['|| *s == '('||*s == '{')
      {
        STPush(&st, *s);
      }
      else
      {
        //右括号比左括号多，数量匹配问题
        if(STEmpty(&st)){
          STDestroy(&st);
          return false;
        }

        char top = STTop(&st);
        STPop(&st);
        //顺序不匹配
        if((*s == ']' && top != '[')
          ||(*s == '}' && top != '{')
          ||(*s == ')' && top !='('))
          {
            STDestroy(&st);
            return false;
          }

      }
      ++s;
    }
    //栈为空，返回真，说明数量为真,zuo
    bool ret = STEmpty(&st);

    STDestroy(&st);
    return ret;
}
```

1. 创建一个栈叫st
2. 将栈初始化
3. 遍历栈
4. 将所有的{{(都压入栈内
5. 将栈顶的内容拿出来和s比对，如果都不一样，就代表失败，表示不符合（在这个比对前要保证在右括号比左括号多）
6. 如果栈都走光了就代表数量一样

注意事项：

1. 所有的函数里都是传的结构体的地址
2. 每次取完TOP的值以后，记得pop一个
3. 是拿}))去和栈里的内容比，存左括号的原因也是因为第一个肯定是左括号，不然直接错误了
4. 记得在return之前删除栈
5. 记得加括号，在判定的时候，这个每个是不同的

```
((*s == ']' && top != '[')
||(*s == '}' && top != '{')
||(*s == ')' && top !='('))
```

232. 用栈实现队列

简单 相关标签 相关企业 A文

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（`push`、`pop`、`peek`、`empty`）：

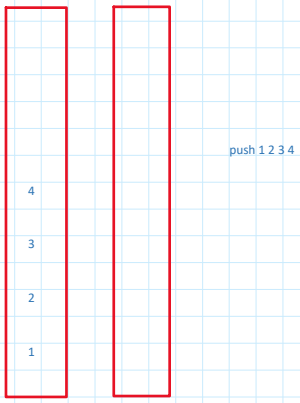
实现 `MyQueue` 类：

- `void push(int x)` 将元素 x 推到队列的末尾
- `int pop()` 从队列的开头移除并返回元素
- `int peek()` 返回队列开头的元素
- `boolean empty()` 如果队列为空，返回 `true` ；否则，返回 `false`

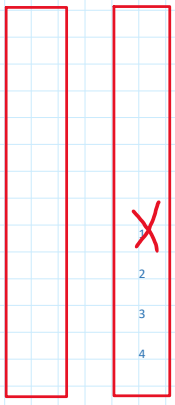
说明：

- 你 **只能** 使用标准的栈操作 —— 也就是只有 `push to top`, `peek/pop from top`, `size`, 和 `is empty` 操作是合法的。
- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。

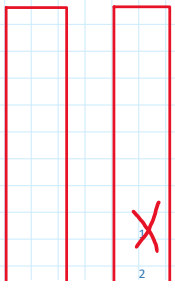
1. 创建两个栈，第一个用于输出，第二个用于插入
第一个栈用于pop， 一个栈用于push



push 1 2 3 4



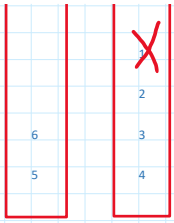
pop->1

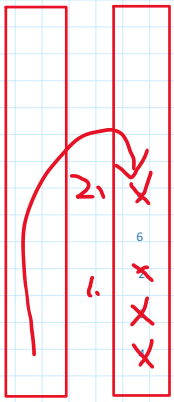


push 5 6

```
93  typedef struct {
94    ST Sin;
95    ST Sout;
96  } MyQueue;
97
98
99  MyQueue* myQueueCreate() {
100   MyQueue* pq = (MyQueue*)malloc(sizeof(MyQueue));
101   STInit(&pq->Sin);
102   STInit(&pq->Sout);
103
104   return pq;
105
106 }
107
108 void myQueuePush(MyQueue* obj, int x) {
109
110   STPush(&obj->Sin, x);
111
112 }
113
114 int myQueuePeek(MyQueue* obj) {
115     if(STEmpty(&obj->Sout)){
116       //导入数据
117       while(!STEmpty(&obj->Sin)){
118         STPush(&obj->Sout,STTop(&obj->Sin));
119         STPop(&obj->Sin);
120
121       }
122     }
123     return STTop(&obj->Sout);
124 }
125
126 int myQueuePop(MyQueue* obj) {
127   int front = myQueuePeek(obj);
128   STPop(&obj->Sout);
129   return front;
130 }
131
132
133
134 bool myQueueEmpty(MyQueue* obj) {
135   return STEmpty(&obj->Sin) && STEmpty(&obj->Sout);
136
137 }
138
139 void myQueueFree(MyQueue* obj) {
140   STDestroy(&obj->Sin);
141   STDestroy(&obj->Sout);
142   free(obj);
143
144 }
145
146
```

```
144  }
145
146
```





pop->2
pop->3
pop->4
pop->5

出数据出左边的，入数据入右边的

622. 设计循环队列

中等 相关标签 相关企业 Aa

设计你的循环队列实现。 循环队列是一种线性数据结构，其操作表现基于 FIFO（先进先出）原则并且队尾被连接在队首之后以形成一个循环。它也被称为“环形缓冲器”。

循环队列的一个好处是我们可以利用这个队列之前用过的空间。在一个普通队列里，一旦一个队列满了，我们就不能插入下一个元素，即使在队列前面仍有空间。但是使用循环队列，我们能使用这些空间去存储新的值。

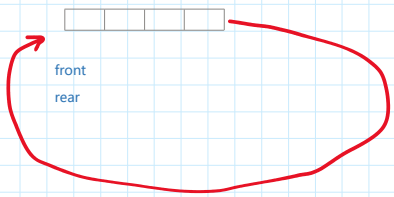
你的实现应该支持如下操作：

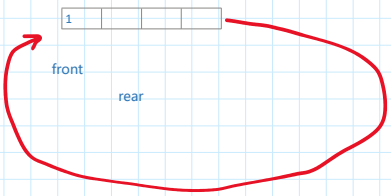
定长

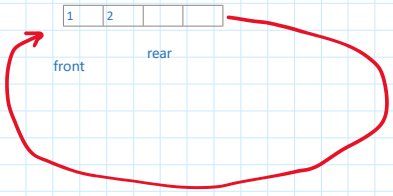
- `MyCircularQueue(k)`：构造器，设置队列长度为 k 。
- `Front`：从队首获取元素。如果队列为空，返回 -1 。
- `Rear`：获取队尾元素。如果队列为空，返回 -1 。
- `enQueue(value)`：向循环队列插入一个元素。如果成功插入则返回真。
- `deQueue()`：从循环队列中删除一个元素。如果成功删除则返回真。
- `isEmpty()`：检查循环队列是否为空。
- `isFull()`：检查循环队列是否已满。

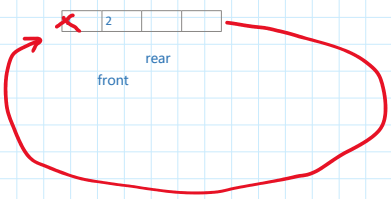
用数组比较好一点

空



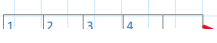






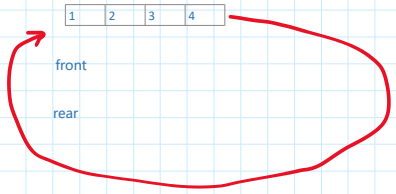
k==4

满



front == rear 空
rear的下一个是front 满

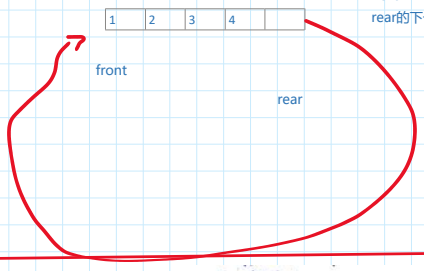
满



k==4

front == rear 空

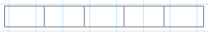
rear的下一个是front 满



k == 4

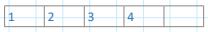
多开一个空间

front
rear



空

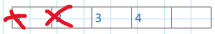
front rear



Push
1 2 3 4

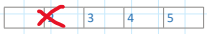
满
(rear+1)%(k+1) == front

front rear



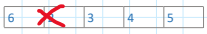
pop->1
pop->2

rear front



push
5 6 7

rear front



front
rear

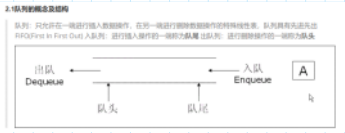
pop->3
pop->4
pop->5
pop->6

为空

```c
  int * a;
  int front;
  int rear;
  int k;

} MyCircularQueue;


MyCircularQueue* myCircularQueueCreate(int k) {
    MyCircularQueue* obj = (MyCircularQueue*)malloc(sizeof(MyCircularQueue));
    //多开一个方便区分空和满
    obj->a =(int*)malloc(sizeof(int)*(k+1));
    obj->front = obj->rear = 0;
    obj->k = k;

    return obj;
}

bool myCircularQueueIsEmpty(MyCircularQueue* obj) {
    return obj->front == obj->rear;
}

bool myCircularQueueIsFull(MyCircularQueue* obj) {
    return (obj->rear+1)%(obj->k+1) == obj->front;
}


bool myCircularQueueEnQueue(MyCircularQueue* obj, int value) {
    if(myCircularQueueIsFull(obj))
      return false;
    obj->a[obj->rear] = value;
    obj->rear++;
    obj-> rear %= (obj->k+1);

    return true;
}

bool myCircularQueueDeQueue(MyCircularQueue* obj) {
    if(myCircularQueueIsEmpty(obj)){
      return false;
    }
    ++obj->front;
     obj-> front %= (obj->k+1);

     return true;

}

int myCircularQueueFront(MyCircularQueue* obj) {
    if(myCircularQueueIsEmpty(obj)){
      return -1;
    }
    else
      return obj->a[obj->front];
}

int myCircularQueueRear(MyCircularQueue* obj) {
    if(myCircularQueueIsEmpty(obj)){
      return -1;
    }
    else
      return obj->a[(obj->rear+obj->k) % (obj->k+1)];
}


void myCircularQueueFree(MyCircularQueue* obj) {
    free(obj->a);
    free(obj);

}

/**
 * Your MyCircularQueue struct will be instantiated and called as such:
 * MyCircularQueue* obj = myCircularQueueCreate(k);
```

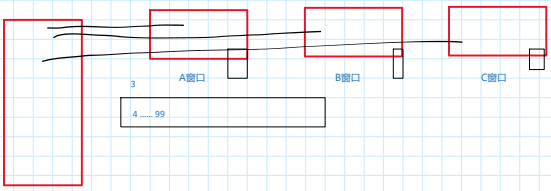
队列

2023年11月19日 11:46

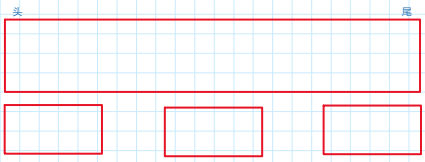


入队：1 2 3 4 5

出队：1 2 3 4 5

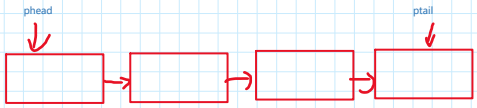


抽号机



单向列表

双向列表



初始化队列

```c
// 初始化队列
void QueueInit(Queue* q) {
    assert(q);
    q->_front = q->_rear = NULL;
    q->size = 0;
}
```

队尾入队列

1. 先断言确定pq的内容不是野指针
2. 创造一个新的结点用malloc
3. 如果这是第一个结点，把head和tail同时指向这个新的结点
4. 如果已经有这个结点了，就把这个结点放在tail的后面然后让tail指向它

```c
void QueuePush(Queue* q, QDataType data) {
    assert(q);
    QNode* newnode = (QNode*)malloc(sizeof(QNode));
    if (newnode == NULL) {
        perror("malloc failed");
        exit(-1);
    }
    if (q->_rear == NULL) {
        q->_front = q->_rear = newnode;
    }
    else {
        q->_rear->_next = newnode;
        q->_rear = newnode;
    }
    q->size++;
}
```

队头出队列

1. 先断言队列内有内容
2. 创一个del表示要删除的地方
3. del指向head，把head等于head->next
4. 然后free（del）

但是如果p->head ==NULL,表示tail也指向了NULL，所以要记得释tail

记得减掉size

```c
void QueuePop(Queue* q) {
    assert(q);
    assert(q->_front);
    QNode* del = q->_front;
    q->_front = q->_front->_next;
    free(del);
    if (q->_front == NULL)
        q->_rear == NULL;
    q->size--;
}
```

销毁队列

1. 断言队列内是否有内容
2. 创一个结构体指向最开始，然后遍历整个链表
3. 在遍历的过程中，创另一个结构体next表示cur下一个结构体
4. 然后free cur，同时让cur指向next；
5. 把head和tail指向NULL
6. 把size变成0

```c
// 销毁队列
void QueueDestroy(Queue* q) {
    assert(q);
    QNode* cur = q->_front;
    while (cur) {
        QNode* next = cur->_next;
        free(cur);
        cur = next;
    }
    q->_front = q->_rear = NULL;
    q->size = 0;
}
```

简单的检测是否为空，元素个数，队头元素和队尾元素

```c
// 获取队列头部元素
QDataType QueueFront(Queue* q) {
```

```c

typedef int QDataType;

typedef struct QueueNode
{
  QDataType val;
  struct QueueNode* next;
}QNode;


typedef struct Queue {
  QNode* phead;
  QNode* ptail;
  int size;
}Queue;

void QueueInit(Queue* pq);
```

简单的检测是否为空，元素个数，队头元素和队尾元素

```c
// 获取队列头部元素
QDataType QueueFront(Queue* q) {
    assert(q);
    assert(q->_front);
    return q->_front->_data;
}
// 获取队列队尾元素
QDataType QueueBack(Queue* q) {
    assert(q);
    assert(q->_rear);
    return q->_rear->_data;
}
// 获取队列中有效元素个数
int QueueSize(Queue* q) {
    assert(q);
    return q->size;
}
// 检测队列是否为空，如果为空返回非零结果，如果非空返回0
int QueueEmpty(Queue* q) {
    assert(q);
    return q->_front == NULL;
}
```

225. 用队列实现栈

简单 相关标签 相关企业 A⁂

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（`push`、`top`、`pop` 和 `empty`）。

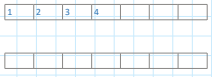
实现 `MyStack` 类：

- `void push(int x)` 将元素 x 压入栈顶。
- `int pop()` 移除并返回栈顶元素。
- `int top()` 返回栈顶元素。
- `boolean empty()` 如果栈是空的，返回 `true` ；否则，返回 `false` 。

注意：

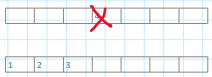
- 你只能使用队列的基本操作 —— 也就是 `push to back`、`peek/pop from front`、`size` 和 `is empty` 这些操作。
- 你所使用的语言也许不支持队列。 你可以使用 list （列表）或者 deque（双端队列）来模拟一个队列 , 只要是标准的队列操作即可。

Push:1 2 3 4
Pop: 4

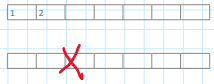


1. 一个队列存数据
2. 另一个队列用来出数据时，导数据

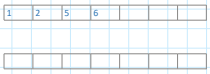
Push:1 2 3 4
Pop: 4



Push:1 2 3
Pop: 3



Push:1 2 3
Push: 5 6



1. 我们需要两个队列来组成栈，一个用于存数据，另一个用于导出数据

1. 栈的初始化：先为栈初始化一块地方里面存放两个栈的头
2. 将栈底指向两个队列的开头

压栈

1. 如果哪个队列是空的，就把数据压在这个队列中

```c
  QNode* phead;
  QNode* ptail;
  int size;
}Queue;

void QueueInit(Queue* pq);
void QueueDetroy(Queue* pq);


void QueuePush(Queue* pq, QDataType x);

void QueuePop(Queue* pq);

QDataType QueueFront(Queue* pq);
QDataType QueueBack(Queue* pq);
bool QueueEmpty(Queue* pq);
int QueueSize(Queue* pq);


void QueueInit(Queue* pq) {
  assert(pq);
  pq->phead = pq->ptail = NULL;
  pq->size = 0;

}
void QueueDetroy(Queue* pq) {
  assert(pq);

  QNode* cur = pq->phead;
  while (cur)
  {
    QNode* next = cur->next;
    free(cur);
    cur = next;
  }
  pq->phead = pq->ptail = NULL;
  pq->size = 0;
}


void QueuePush(Queue* pq, QDataType x) {
  assert(pq);

  QNode* newnode = (QNode*)malloc(sizeof(QNode));
  if (newnode == NULL) {
    perror("malloc fail");
    return;
  }
  newnode->val = x;
  newnode->next = NULL;
  if (pq->ptail == NULL) {
    pq->ptail = pq->phead = newnode;
  }
  else {
    pq->ptail->next = newnode;
    pq->ptail = newnode;
  }
  pq->size++;
}

void QueuePop(Queue* pq) {
  assert(pq);
  assert(pq->phead);
  QNode* del = pq->phead;
  pq->phead = pq->phead->next;
  free(del);
  del = NULL;
  if (pq->phead == NULL)
    pq->ptail = NULL;

  pq->size--;
}

QDataType QueueFront(Queue* pq) {
  assert(pq);
  assert(pq->phead);

  return pq->phead->val;
}
QDataType QueueBack(Queue* pq) {
  assert(pq);
  assert(pq->ptail);

  return pq->ptail->val;
}
bool QueueEmpty(Queue* pq) {
  assert(pq);

  return pq->phead == NULL;

}
int QueueSize(Queue* pq) {
  assert(pq);

  return pq->size;
}



typedef struct {
    Queue q1;
    Queue q2;
} MyStack;


MyStack* myStackCreate() {
    MyStack* pst = (MyStack*)malloc(sizeof(MyStack
    QueueInit(&pst->q1);
    QueueInit(&pst->q2);

    return pst;

}

void myStackPush(MyStack* obj, int x) {
    if(!QueueEmpty(&obj->q1)){
      QueuePush(&obj->q1,x);
    }
    else
    {
      QueuePush(&obj->q2,x);
    }
```

出栈：

1. 判断哪个队列是空的
2. 然后把把不是空的队列的数据存到空的队列数据中，保证最后不是空的队列中剩下一个数据
3. 记录不是空的队列中的最后一个数，再pop的最后一个数，然后返回这个数

```
}
}

int myStackPop(MyStack* obj) {
  Queue* emptyq = &obj->q1;
  Queue* nonemptyq = &obj->q2;
  if(!QueueEmpty(&obj->q1)){
      emptyq = &obj->q2;
      nonemptyq = &obj->q1;
  }
  while(QueueSize(nonemptyq)>1){
      QueuePush(emptyq, QueueFront(nonemptyq));
      QueuePop(nonemptyq);
  }
  int top = QueueFront(nonemptyq);
  QueuePop(nonemptyq);
  return top;

}
```

这边是创一个结构体，要把结构体的指针的地址传过去

一定要记住这里的OBJ是一个临时变量
如果你只是单纯的使用这个形参不需要取地址
但是如果想要改变这个结构体的指针指向的变量
就要传这个指针的地址

因为这里面是结构体的指针的地址，就不需要传取地址符了

判断栈顶数据：
1.直接判断queueback是什么
两个都判断，找到那个不是空的队列

```
int myStackTop(MyStack* obj) {
    if(!QueueEmpty(&obj->q1)){
      return QueueBack(&obj->q1);
    }
    else
    {
      return QueueBack(&obj->q2);
    }
}

bool myStackEmpty(MyStack* obj) {
  return QueueEmpty(&obj->q1)&&QueueEmpty(&obj->q2
}
```

直接删除两个queque

```
void myStackFree(MyStack* obj) {
  QueueDetroy(&obj->q1);
  QueueDetroy(&obj->q2);

  free(obj);
}

/**
 * Your MyStack struct will be instantiated and ca
 * MyStack* obj = myStackCreate();
 * myStackPush(obj, x);

 * int param_2 = myStackPop(obj);

 * int param_3 = myStackTop(obj);
```

```
int myStackTop(MyStack* obj) {
    int top;
    if (!QueueEmpty(&obj->q1)) {
        top = QueueBack(&obj->q1);
    }
    else {
        top = QueueBack(&obj->q1);
    }
    return top;
}
```

直接return可以减少一个函数的定义