@[TOC]

链表总结

单链表

结构体表示：

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

1、不带头结点的单链表，递归删除所有结点值为x的链表

注意必须加引用，**不加引用会断链**

```
void deletenode(list &head, int x){          //必须加引用
    struct node *p;
    if(head == NULL) return;
    else if(head->data == x){
        p = head;
        head = head->next;                   //这里不会断链 head->next = head->next->next
        free(p);
        deletenode(head, x);
    }
    else{
        deletenode(head->next, x);
    }
}
```

2、链表的排序（默认带头结点）

（1）直接插入排序

思路：

- 需要三个指针
  - 一个工作指针p
  - 一个指针nextp 保存p的next指针用于后序遍历
  - 一个遍历指针q 找到插入位置，每次找的时候，指向head，比较head->next

代码：

```
void  InsertSort_Node(list &head){
    struct node *p = head->next,*nextp, *q;
    head->next = NULL;
    while(p!=NULL){
        nextp = p->next;
```

```
        q = head;
        while(q->next!=NULL && q->next->data < p->data){//找到插入位置
            q = q->next;
        }
        p->next = q->next;                                  //插入
        q->next = p;
        p = nextp;
    }
}
```

（2）冒泡排序

思路：

- 设置四个指针
  - q指针用来每一层的遍历
  - pre指针，指向q的前一个节点方便指针的交换
  - q指针，用来比较
  - tail指针，指向未排好序的最后一个节点
- 实现方式：
  - 模拟冒泡排序，每次pre和p都重新指向头结点，以及下一个节点
  - 如果前一个数比后一个数大，交换指针
  - 不是的话，先让p走一步
  - 随后pre往后走一步，**q指向p的下一个节点（这里保证不管交换还是没交换q节点都可以指向下一个比较的节点）**
  - 注意结束后，要更新tail指针，双重while循环也是与tail指针比较的

代码：

```
void BubbleSort_Node(list &head){
    struct node *pre,*p,*q;
    struct node *tail = NULL;           //尾指针，每次冒泡排序之后，最后一个元素都已经确定顺序
    while(head->next!= tail){           //冒泡次数
        pre = head;
        p = head->next;
        q = p->next;
        while(p->next!=tail){           //冒泡长度
            if(p->data > q->data){      //如果前一个数比后一个数大，交换指针
                pre->next = q;
                p->next = q->next;
                q->next =  p;
            }
            else{
                p = p->next;        //不用交换指针，继续往下找
            }
            pre = pre->next;        //p指针已经在上面的if和else中移动了，无需再移动
            q = p->next;
        }
        tail = p;                   //经过一次冒泡，当前最后一个元素已经排好序，tail前移
    }
```

```
}
```

(3)二路归并排序 时间复杂度NlogN 在链表排序中属于较好的一种排序方式

思路:

1. 利用快慢指针找到中点，将链表分为左右两个链表
2. 递归找到左右链表的中点
3. 当递归中只剩下一个节点时，合并链表节点

代码:

```
list Merge_Node(list head1, list head2){            //合并两个有序链表
    list newhead = (list)malloc(sizeof(struct node));
    list p = newhead;                               //工作指针
    while(head1 != NULL && head2 !=NULL){
        if(head1->data > head2->data){
            p->next = head2;
            head2 = head2->next;
        }
        else{
            p->next = head1;
            head1 = head1->next;
        }
        p = p->next;
    }
    if(head1!=NULL) p->next = head1;
    if(head2!=NULL) p->next = head2;
    return newhead->next;
}

list TwoRoadMergeSort_Node(list head){
    if(head == NULL || head->next == NULL) return head;
    list fast = head;
    list slow = head;
    while(fast->next != NULL && fast->next->next != NULL){
        fast = fast->next->next;
        slow = slow->next;
    }
    list mid = slow->next;                          //找到中点
    slow->next = NULL;
    list left =  TwoRoadMergeSort_Node(head);       //递归找中点
    list right = TwoRoadMergeSort_Node(mid);
    return Merge_Node(left, right);                 //归并
}
```

3、给两个单链表，找到公共节点

思路:

- 找到长链表与短链表，长的链表先把长的走完，然后一起走，遇到相同的节点就返回，否则继续走

代码：

```
int Length_Node(list head){//不带头结点  求链表长度
    int num = 0;
    list p = head;
    while(p!=NULL){
        num++;
        p = p->next;
    }
    return num;
}

list FindSameNode(list head1, list head2){
    int len1 = Length_Node(head1);
    int len2 = Length_Node(head2);
    list longlist, shortlist;
    int dist;
    if(len1>len2){                  //找到长的链表与短的链表
        dist = len1 - len2;
        longlist = head1;
        shortlist = head2;
    }
    else{
        dist = len2 - len1;
        longlist = head2;
        shortlist = head1;
    }

    while(dist){                    //长的链表先走完更长的那部分
        longlist = longlist->next;
        dist--;
    }
    while(longlist != NULL){        //找公共节点
        if(longlist == shortlist){
            return longlist;
        }
        else{
            longlist = longlist->next;
            shortlist = shortlist->next;
        }
    }
    return NULL;                    //未找到
}
```

4、将一个链表带头结点的链表分解为两个链表，一个含有奇数序列，一个含有偶数序列

思路：

- 创建一个节点head2，用来当第二个链表的头结点
- p指针指向所给链表head1的next，然后将head1->next置为空，方便后序插入
- 遍历p节点
- 添加计数器num

- 如果是奇数就将p节点添加到head1的工作指针p1中
- 如果是偶数就将p节点添加到head2的工作指针p2中

代码：

```
list DivlistToTwo(list &head1){
    list p = head1->next,q,p1,p2;    //p1,p2为head1，head2工作指针
    list head2 = (list)malloc(sizeof(struct node));
    head1->next = NULL;
    p1 = head1;                      //工作指针指向对应头结点
    p2 = head2;
    int num = 1;                     //添加计数器
    while(p!=NULL){                  //遍历p节点
        if(num%2==1){                //奇数节点
            p1->next = p;
            p1 = p1->next;
        }
        else{                        //偶数节点
            p2->next = p;
            p2 = p2->next;
        }
        num++;                       //继续遍历
        p = p->next;
    }
    if(p1) p1->next =NULL;           //尾部置空
    if(p2) p2->next =NULL;
    return head2;                    //返回另外一个节点
}
```

5、设A，B为两个单链表（带头结点），元素递增有序。设计一个算法，求A与B的公共元素，并产生单链表C，要求不破坏A,B的节点

思路：

- 两个工作指针，pa，pb，创建新的头结点C，另外设一个新的工作指针pc
- pa，pb指针一起走，哪个指针小，哪个指针先走，直到相同，相同创建节点加入pc中

代码：

```
list FindPublicNode(list A, list B){
    list pa = A->next, pb = B->next,pc,temp;        //pa，pb,pc为A,B,C的工作指
针,temp申请新的节点并入C中
    list C = (list)malloc(sizeof(struct node));
    pc = C;
    while(pa != NULL && pb != NULL){
        if(pa->data < pb->data){             //pa小，pa往前走
            pa = pa->next;
        }
        else if(pb->data < pa->data){        //pb小，pb往前走
            pb = pb->next;
        }
        else{                                //两个节点相同
            temp = (struct node *)malloc(sizeof(struct node));
            temp->data = pa->data;           //加入C中
```

```
            pc->next = temp;
            pc = pc->next;                    //工作指针继续往后找
            pa = pa->next;
            pb = pb->next;
        }
    }
    pc->next = NULL;
    return C;
}
```

6、已知两个带头结点链表A，B分别表示两个集合，元素递增排序，求A与B的交集，并存放于A链表中

思路：

- 和上一题基本一致，存放于A链表中说明得释放A中多余节点，当pa较小时，删除pa节点，**这里涉及删除节点，所以使用pa->next来遍历可以省略一个节点保存pa的前驱节点**
- **同时在遍历完之后，如果pa指针不为空，记得需要删除后序的节点**

代码：

```
void InterNode(list &A, list B){
    list pa = A, pb = B->next,temp;          //两个工作指针
    while(pa->next!=NULL && pb !=NULL){
        if(pa->next->data < pb->data){       //pa更小，删除节点pa->next
            temp = pa->next;
            pa->next = pa->next->next;
            free(temp);
        }
        else if(pb->data < pa->next->data){//pb更小，继续往后找，看看有无更大的
            pb = pb->next;
        }
        else{                                //相同了，就继续往后找
            pb = pb->next;
            pa = pa->next;
        }
    }
    while(pa->next!=NULL){                   //后面还有节点，继续删除
        temp = pa->next;
        pa->next = pa->next->next;
        free(temp);
    }
}
```

7、链表的KMP算法 已知整数序列 A = a1, a2,...,am。 B = b1, b2,...,bn,存入两个单链表，判断B是否为A的连续子序列

思路:

- 链表的朴素模式匹配算法
- 双指针pa，pb分别用来遍历，p指针用来记录下一次pa的指针指向
- 当两个节点值相同时，继续比较，不同pb重头开始比较，pa从下一个前面记录的下一个节点开始比较

代码：

```
bool IsPartNode(list A, list B){
    list pa = A->next, pb = B->next,p = A->next;          //pa，pb为比较节点，  p为遍历节点
    while(pa != NULL && pb != NULL){                      //遍历节点
        if(pa->data == pb->data){                         //相同就继续比较
            pa = pa->next;
            pb = pb->next;
        }
        else{                                             //不同的话，pb从头开始比较，pa从下一个节点开始比较
            p = p->next;
            pb = B->next;
            pa  = p;
        }
    }
    if(pb == NULL) return true;                           //pb遍历完，说明找到了
    else return false;                                    //pb没遍历完，说明没找到
}
```

8、 编写函数成反转链表，给定一个链表，从表头开始每两个相邻的节点为一对，反转每一对链表的两个节点，要求交换节点位置，不是交换节点的值，整体顺序不变末尾若是单个节点，则该节点位置不变，如12345，反转后为21435

思路：

- 反转两个链表需要三个指针prepre，pre，p
- 如果是不带头结点，需要注意**第一次涉及到第一个节点的交换有一些不一样**
- 在下一次的指针分配时**需要保证有后序指针才可以交换**，如果已经交换完，或者只剩最后一个节点，直接break就可以了

代码：

```
void ReVerSeTwoNode(list &head){
    int flag = 1;
    if(head == NULL ||head->next == NULL) return;         //只有一个节点,或者没有节点，直接返回
    list p = head->next, pre = head,prepre=head;          //交换两个节点，需要保存三个节点
    while(p!=NULL){
        if(flag){                                         //不带头结点的时候，第一次交换和后序交换有所区别
            pre->next = p->next;
            p->next = pre;
            head = p;
            flag = 0;
        }
        else{                                             //重新构建三个节点的指向
            pre->next = p->next;
            p->next = pre;
            prepre->next = p;
```

```
        }
        if(pre->next ==NULL ||pre->next->next == NULL) break;    //已经反转完或者只剩下一个节点，不用交换了
        prepre = pre;                                            //重新分配三个节点
        pre = pre->next;
        p = pre->next;
    }
}
```

9、链表的合并，将两个链表合并为一个升序链表（尾插法）（降序：头插法）

思路：

- 申请一个新的头结点用于存储结果
- 三个工作指针p1,p2,p
- 判断两个链表，每次将较小的节点加入p中，然后较小的指针往后走一步，如果两个都相同，则一起走一步
- 最后再将剩余的节点加入p指针中

代码：

```
list MergeTwoList(list head1, list head2){//默认不带头结点
    list newhead = (struct node *)malloc(sizeof(struct node));
    list p1 = head1, p2 = head2,p = newhead;
    while(p1 != NULL && p2 != NULL){//合并两个链表
        if(p1->data > p2->data){    //p1大于p2
            p->next = p2;
            p = p->next;
            p2 = p2->next;
        }
        else if(p1->data < p2->data){//p2大于p1
            p->next = p1;
            p = p->next;
            p1 = p1->next;
        }
        else{                        //p1等于p2
            p->next = p1;
            p = p->next;
            p1 = p1->next;
            p2 = p2->next;
        }
    }
    if(p1!=NULL){                //处理剩余节点
        p->next = p1;
    }
    else{
        p->next = p2;
    }
    return newhead->next;        //返回不带头结点的单链表
}
```

10、查找链表的倒数第k个节点(不带头结点)

思路:

- 只需要遍历一次即可，在未遍历到第k节点之前，链表不存在倒数第k元素，计数器i加1
- 当计数器遍历到了第k个元素之后，第二个指针p2开始走，p1结束的时候p2所指的节点就是倒数第k个节点

代码:

```
bool FindResKNode(list head, int k){
    int i =0;
    list p1 = head, p2 = head;
    while(p1!=NULL){
        if(i<k){            //没到p1继续走，计数器加1
            i++;
        }
        else{               //到了，两个指针一起走
            p2 = p2->next;
        }
        p1 = p1->next;
    }
    if(i == k){                 //找到了，输出节点 返回1
        printf("%d ",p2->data);
        return 1;
    }
    else{                       //没找到，返回0
        return 0;
    }
}
```

11、找到链表中最小的节点并放到最前面

思路:

- 四个指针pre,p,minpre,minp
- pre,p用来遍历 minp，minpre用来保存最小值节点
- **找到后需要判断是否是头结点**，如果是无需操作
- 不是的话，进行断链与接链，把最小值节点链接到表头

代码:

```
void FindMinNode(list &head){
    list pre=head,p=head,minp = head,minpre = head;
    while(p!=NULL){
        if(p->data < minp->data){   //找到最小节点
            minpre = pre;
            minp = p;
        }
        pre = p;            //不断遍历
        p = p->next;
    }
    if(minp == head) return;        //如果最小节点就是头结点，不用操作
    else{
```

```
        minpre->next = minp->next;  //链接到最前面
        minp->next = head;
        head = minp;
    }
    visitnode(head);
}
```

12、 判断单链表是否有环，如果有的话，返回环的起始位置

思路:

- 设置快慢指针，如果存在环，快指针一定能够追上慢指针，如果相遇，则说明存在环
- 设置两个指针，一个p1指向头结点，一个p2指向相遇节点，然后一直走，走到相遇位置便是环的起始位置（具体证明可以看王道，由于得画图，这里就不详细说明了）

代码:

```
list IsLoopAndFindStart(list head){
    list slow = head, fast = head;  //快慢指针
    while(fast!=NULL && fast->next!=NULL){
        slow = slow->next;
        fast = fast->next->next;
        if(slow==fast){          //存在环
            list p1 = head;
            list p2 = slow;
            while(p1!=p2){       //两个指针找到起始节点
                p1 = p1->next;
                p2 = p2->next;
            }
            return p1;           //返回起始位置节点
        }
    }
}
```

13、单链表保存m个整数，|data|<n(n为正整数），设计一个时间复杂度尽量低的算法，对于链表中绝对值相等的节点，仅保存第一次出现的节点而删除其他绝对值相等的节点

思路:

- 由于给了数据域的范围，所以可以采用hash数组来统计个数，为链表与hash考点的结合
- 申请一个hash数组，下标代表值，数据域保存出现个数
- 遍历链表，当数据域的值未出现时，hash表对应下标值+1
- 当hash已经个数已经大于等于1时，删除该节点，（删除节点需要保存前驱节点）

代码:

```
void DelAbsSameNode(list &head, int n){
    int hash[n+1] = {0};
    list p = head,pre;
    while(p!=NULL){
        if(hash[abs(p->data)] < 1){          //如果只出现一次，继续往下找，hash表对应的数
+1
```

```
            hash[abs(p->data)]++;
            pre = p;
            p = p->next;
        }
        else{                                   //否则删除p节点，然后继续p重新指向
            pre->next = p->next;
            free(p);
            p = pre->next;
        }
    }
}
```

14、设带头结点的单链表存储线性表 L = （a1,a2,....,an)设计空间复杂度为O(1)的算法，实现重新排列L中元素为L' = （a1,an,a2,an-1,a3,an-2....)

思路:

- 观察两个链表之间的差别，可以发现，新的排列是由a1，an交替组成，因此可以先将L分为两个链表，同时后半段链表需要逆序
- 所以，实现方式为：找到中点，断链，后半段逆序，然后后半段链表插入到前半段链表中

代码:

```
void ResAndInsert(list &L){
    list p,nextp,mid,slow = L->next,fast = L->next,nextmid;

    while(fast!=NULL && fast->next!=NULL){      //找到中点
        slow = slow->next;
        fast = fast->next->next;
    }
    mid = slow->next;
    slow->next = NULL;                          //保存中点，然后断链
    p = mid->next;                              //p为工作节点
    mid->next = NULL;                           //头插法到mid中
    while(p!=NULL){
        nextp = p->next;
        p->next = mid;
        mid = p;
        p = nextp;
    }
    p = L->next;                                //准备插入
    while(p!=NULL && mid!=NULL){
        nextmid = mid->next;                    //保存下一个节点
        mid->next = p->next;                    //插入
        p->next = mid;
        p = p->next->next;                      //p后移
        mid = nextmid;                          //mid后移
    }
}
```

特殊链表

循环单链表

1、两个循环单链表h1，h2，编写函数实现将链表h2链接到h1之后，要求链接后的链表仍然保持循环链表形式

思路：

- 分别找到对应的尾结点，然后断链、接链

代码：

```
void LinkTwoNode(list &h1, list h2){
    list tail1 = h1, tail2 = h2;
    while(tail1->next!=h1){          //找到 h1尾结点
        tail1 = tail1->next;
    }
    while(tail2->next!=h2){          //找到h2尾结点
        tail2 = tail2->next;
    }
    tail1->next = h2;                //接链
    tail2->next = h1;
}
```

2、带头结点的循环单链表，结点值均为正数，设计算法，找到结点值中最小的节点并输出，然后删除，直到单链表为空

思路：

- 找到单链表最小值，然后删除，然后继续找，循环结束条件，L->next!=L

代码：

```
void FindMinAndOut(list &head){
    list pre = head,p = head->next,minp =head->next,minpre;
    while(head->next!=head){         //只有一个节点的时候跳出循环
        pre = head;p = head->next;
        minpre= head; minp =head->next;
        while(p != head){            //循环找到最小元素
            if(p->data < minp->data){
                minp = p;            //有更小，进行更新
                minpre = pre;
            }
            pre = p;                 //继续往下找
            p = p->next;
        }
        printf("当前最小值为%d\n",minp->data);
        minpre->next = minp->next;   //删除最小值
        free(minp);
    }
}
```

双链表

(难）1、改造表头为T的双向链表，使其right域保持原来的位置关系，并且left域根据key值大小，从大到小排序

思路：

- 实质就是排序算法
- 先清除所有左指针
- 然后再遍历链表，找到最小值，将其左指针按照头插法链接到head中，得到从大到小的序列
- **注意当左指针存在时，链表已经排好序，结束标志是min最小值未进行更新**

代码：

```
typedef struct dnode{
    int data;
    int freq;
    struct dnode *llink, *rlink;
}*dlist;

int LeverUp(dlist &head){
    int min;
    dlist minp, p = head->right, tail = head;

    while(p!=NULL){                  //先清除所有左指针
        p->left = NULL;
        p = p->right;
    }

    while(true){                     //每次找最小
        p = head->right;
        min = MAX;
        while(p->right != NULL){
            if(p->left!=NULL)  p = p->right;         //p左指针不为空，说明已经在序列
中，不用排序
            else if(p->data < min){                  //记录更小值
                min = p->data;
                minp = p;
                p = p->right;
            }
            else p = p->right;                       //不是最小值继续找
        }
        if(min == MAX){
            return 1;                                //遍历一轮没变化，说明已经排好序
        }
        minp->left = tail;                           //头插法插入左指针
        tail = minp;
    }
}
```

(难）2、带头结点的双链表l，每个节点有4个数据成员，前驱节点llink，后继节点rlink，数据成员data，访问频度freq，且已知双向链表L中节点一直按访问频度递减的顺序排列且频繁访问的节点一直靠近表头，初始状态l中的所有节点freq的节点都为0，对双链表的locate操作，每操作一次，将数据值x的节点访问频度+1，设计一个算法，对双链表l的locate操作，要求操作后L中的节点仍然按照频度的递减顺序排列

思路：

- 先找到结点值为x的节点，频度加1，然后断链
  - 如果是该节点前一个节点是头结点，不用操作
- 然后找到需要插入的节点，**分情况讨论**
  - 插在最后一个，只需接三条链
  - 不是插在最后一个，需要接四条链

代码：

```
typedef struct dnode{
    int data;
    int freq;
    struct dnode *llink, *rlink;
}*dlist;

void Locate(dlist head, int x){
    dlist p = head->rlink,q;
    while(p != NULL && p->data != x) p = p->rlink;          //找到节点值为x的节点
    p->freq++;                                              //访问频度加1
    if(p!=NULL){                                            //找到节点
        p->freq++;                                          //频度加1
        if(p->llink == head) return;                        //第一个节点，不用调整
        q = p;                                              //保存需要操作的节点

        p->llink->rlink = p->rlink;                         //断开p左右节点的链并接上
        if(p->rlink!=NULL)                                  //右链得保证存在才能接
            p->rlink->llink = p->llink;

        p = q->llink;                                       //从q的前一个结点找到q的
freq大的节点
        while(p != head && p->freq < q->freq) p = p->llink; //找到需要插入的节点
        if(p->rlink==NULL){                                 //q插到p节点前 ,注意这里
的插法
            q->llink = p;
            q->rlink = p->rlink;
            p->rlink = q;
        }
        else{                                               //不是插到最后一个节点
            p->rlink->llink = q;
            q->llink = p;
            q->rlink = p->rlink;
            p->rlink = q;
        }
    }
}
```

循环双链表

1、带头结点的双向循环链表L(a1,a2,a3,...an),转化为L’（a1,a3,...an,...a4,a2)

思路:

- 添加计数器，如果是偶数则将该节点移出，并利用头插法，头插到最后一个节点的前面

代码:

```
dlist DNodeChange(dlist head){
    dlist tail = head->left,p=head->right;
    dlist q;
    int count = 1;                              //从第一个节点开始计数
    while(p!=tail){
        if(count%2==0){
            q = p->right;                       //q保存下一个节点

            p->left->right = p->right;          //先断链
            p->right->left = p->left;

            p->right = tail->right;             //在尾指针进行头插法
            tail->right = p;
            p->right->left = p;
            p->left = tail;

            p = q;                              //p指向下一个节点
        }
        else{
            p = p->right;
        }
        count++;
    }
    return head;
}
```