查找单链表的中间结点

slow = fast = head;

while (fast&&fast->next)

{

fast = fast->next->next;

slow = slow->next;

}

 不带头结点的单链表相关算法

/\*一定要看重点低级错误

插入和删除会影响头指针

标志位frist在第一次操作完后要改变它的值！

着重看懂删除函数的每一行代码

分析如何前驱指针始终伴随扫描指针\*/

typedef struct node

{

int data;

struct node \*next;

}Node;

typedef Node\* linklist;

Status init\_list(linklist \*pHead)

{

\*pHead = NULL;

return ok;

}

Status InputData(linklist \*pHead)

{/\*牢记标志位frist的使用方法\*/

int DATA, frist = 0;

Node \*p, \*q; p = q = NULL;

while (scanf("%d", &DATA), DATA != 0)

{

p = (Node\*)malloc(sizeof(Node));

if (!p) return error;

p->data = DATA;

p->next = NULL;

if (!frist)

{

\*pHead = p;

q = p;

frist = 1;

}

else

{

q->next = p;

q = p;

}

}

}

Status AddToTail(linklist \*pHead, int e)

{

Node \*p = (Node\*)malloc(sizeof(Node));

if (!p) return error;

p->data = e;

p->next = NULL;

if (\*pHead == NULL)

{

\*pHead = p;

return ok;

}

else

{

Node \*q = \*pHead;

while (q->next != NULL)/\*【重点低级错误】寻找最后一个节点，while应该是终止于倒数第二个节点\*/

q = q->next;

q->next = p;

return ok;

}

}

Status Traverse(linklist pHead)

{

if (pHead == NULL)

return error;

while (pHead != NULL)

{

printf("%d ", pHead->data);

pHead = pHead->next;

}

printf("\n");

}

Status RemoveNode(linklist \*pHead, int value)

{

if (\*pHead == NULL)

return error;

Node \*p = \*pHead, \*q = NULL;

while (p != NULL&&p->data != value)

{

q = p;

p = p->next;

}

if (p == \*pHead)

{

\*pHead = p->next;

return ok;

}

else

{

q->next = p->next;

return ok;

}

}

Status FromTailPrintf1(linklist pHead)

{

if (pHead == NULL)

return error;

Node\* stack[100];

int top = -1;

while (pHead != NULL)

{

if (top<100)

{

top++;

stack[top] = pHead;

}

else return error;

pHead = pHead->next;

}

while (top != -1)

printf("%d ", stack[top--]->data);

}

递归逆序输出单链表

void FromTailPrintf2(linklist pHead)

{

if (pHead != NULL)

{

FromTailPrintf(pHead->next);

printf("%d ", pHead->data);

}

}

单链表的前驱指针始终伴随扫描指针写法

//至少含有一个元素，即不为空

p = frist; q = NULL;

while (p != NULL)

{

    //在此通过p可以操作所有的元素

    q = p;/\*这两个语句是不可分割的，是一个整体，相当于原语\*/

    p = p->next;/\*在这个整体前后，q一定指向p的前驱\*/

    //在此p：2 to last；q：1 to last - 1

}

/\*出循环后，q仍旧指向p的前驱，只要p不指向空\*/

/\*适用于查找value，并进行牵扯到其前驱的操作\*/

头插法

void InputData(LinkList \*pHead)

{

char Data;

Node \*p = NULL;

if (\*pHead == NULL)

return;

while (Data = getchar(), Data != '#')

{

p = (Node\*)malloc(sizeof(Node));

if (!p) exit(-1);

p->ch = Data;

p->next = NULL;

p->next = (\*pHead)->next;

(\*pHead)->next = p;

}

}

尾插法

void CreateList2(LinkList \*L, int n)

{

int i;

LinkList p, q;

\*L = (LinkList)malloc(sizeof(struct LNode)); // 生成头结点

(\*L)->next = NULL;

q = \*L;//q指向表尾

printf("请输入%d个数据\n", n);

for (i = 1; i <= n; i++)

{

p = (LinkList)malloc(sizeof(struct LNode));

scanf("%d", &p->data);

q->next = p;

q = q->next;//q始终指向表尾

}

p->next = NULL;

}

合并两个有序链表

void MergeList(LinkList La, LinkList \*Lb, LinkList \*Lc)

{

LinkList pa = La->next, pb = (\*Lb)->next, pc;

\*Lc = pc = La;

while (pa&&pb)

if (pa->data <= pb->data)

{

pc->next = pa;

pc = pa;

pa = pa->next;

}

else

{

pc->next = pb;

pc = pb;

pb = pb->next;

}

pc->next = pa ? pa : pb;

free(\*Lb);

Lb = NULL;

}

逆置单链表

void reverse(LinkList L)

{

if (L == NULL)

return;

LinkNode \*q, \*p;

p = L->next;

L->next = NULL;

while (p != NULL)

{

q = p->next;

p->next = L->next;

L->next = p;

p = q;

}

}

查找倒数第k个元素

采用两个指针P1,P2，P1先前移K步，然后P1、P2同时移动，当p1移动到尾部时，P2所指位置的元素即倒数第k个元素 。

// 查找单链表中倒数第K个结点

ListNode \* RGetKthNode(ListNode \* pHead, unsigned int k) // 函数名前面的R代表反向

{

if (k == 0 || pHead == NULL) // 这里k的计数是从1开始的，若k为0或链表为空返回NULL

return NULL;

ListNode \* pAhead = pHead;

ListNode \* pBehind = pHead;

while (k > 1 && pAhead != NULL) // 前面的指针先走到正向第k个结点

{

pAhead = pAhead->m\_pNext;

k--;

}

if (k > 1 || pAhead == NULL) // 结点个数小于k，返回NULL

return NULL;

while (pAhead->m\_pNext != NULL) // 前后两个指针一起向前走，直到前面的指针指向最后一个结点

{

pBehind = pBehind->m\_pNext;

pAhead = pAhead->m\_pNext;

}

return pBehind; // 后面的指针所指结点就是倒数第k个结点

}

判断单链表(不带头结点)是否有环

如果有环，两步快指针会和一步慢指针相遇，就像操场跑步。没环，快指针会扫描到表尾结束

bool Ring(LinkList L) {

LinkList pFast, pSlow;

if (!L)

return false;

pFast = L;

pSlow = L;

while (pFast && pFast->next) {

pSlow = pSlow->next;

pFast = pFast->next->next;

if (pFast == pSlow)

return true;

}

return false;

}

从尾到头打印单链表

void fromTailPrint(int \*a, int n) {

if (a == NULL)

return;

if (n >= 1) {

fromTailPrint(a + 1, n - 1);

printf("%d ", a[0]);

}

}

O(1)时间内删除结点

void Delete(ListNode \* pHead, ListNode \* pToBeDeleted){

if (pToBeDeleted == NULL)

return;

if (pToBeDeleted->m\_pNext != NULL){

pToBeDeleted->m\_nKey = pToBeDeleted->m\_pNext->m\_nKey; // 将下一个节点的数据复制到本节点，然后删除下一个节点

ListNode \* temp = pToBeDeleted->m\_pNext;

pToBeDeleted->m\_pNext = pToBeDeleted->m\_pNext->m\_pNext;

delete temp;

}

else { // 要删除的是最后一个节点

if (pHead == pToBeDeleted){ // 链表中只有一个节点的情况

pHead = NULL;

delete pToBeDeleted;

}

else {

ListNode \* pNode = pHead;

while (pNode->m\_pNext != pToBeDeleted) // 找到倒数第二个节点

pNode = pNode->m\_pNext;

pNode->m\_pNext = NULL;

delete pToBeDeleted;

}

}

}