单向链表操作演示

Created by mac on 15/9/26.

Copyright (c) 2015年 mac. All rights reserved.

#include "stdlib.h"

#include "stdio.h"

//#define NULL 0

#define LEN sizeof(struct student)

struct student

{

int num; //学号

float score; //分数，其他信息可以继续在下面增加字段

struct student \*next; //指向下一节点的指针

};

int n; //节点总数

/\*

==========================

功能：创建n个节点的链表

返回：指向链表表头的指针

==========================

\*/

struct student \*Create()

{

struct student \*head; //头节点

struct student \*p1 = NULL; //p1保存创建的新节点的地址

struct student \*p2 = NULL; //p2保存原链表最后一个节点的地址

n = 0; //创建前链表的节点总数为0：空链表

p1 = (struct student \*) malloc (LEN); //开辟一个新节点

p2 = p1; //如果节点开辟成功，则p2先把它的指针保存下来以备后用

if(p1==NULL) //节点开辟不成功

{

printf ("\nCann't create it, try it again in a moment!\n");

return NULL;

}

else //节点开辟成功

{

head = NULL; //开始head指向NULL

printf ("Please input %d node -- num,score: ", n + 1);

scanf ("%d %f", &(p1->num), &(p1->score)); //录入数据

}

while(p1->num != 0) //只要学号不为0，就继续录入下一个节点

{

n += 1; //节点总数增加1个

if(n == 1) //如果节点总数是1，则head指向刚创建的节点p1

{

head = p1;

p2->next = NULL; //此时的p2就是p1,也就是p1->next指向NULL。

}

else

{

p2->next = p1; //指向上次下面刚刚开辟的新节点

}

p2 = p1; //把p1的地址给p2保留，然后p1产生新的节点

p1 = (struct student \*) malloc (LEN);

printf ("Please input %d node -- num,score: ", n + 1);

scanf ("%d %f", &(p1->num), &(p1->score));

}

p2->next = NULL; //此句就是根据单向链表的最后一个节点要指向NULL

free(p1); //p1->num为0的时候跳出了while循环，并且释放p1

p1 = NULL; //特别不要忘记把释放的变量清空置为NULL,否则就变成"野指针"，即地址不确定的指针

return head; //返回创建链表的头指针

}

/\*

===========================

功能：输出节点

返回： void

===========================

\*/

void Print(struct student \*head)

{

struct student \*p;

printf ("\nNow , These %d records are:\n", n);

p = head;

if(head != NULL) //只要不是空链表，就输出链表中所有节点

{

printf("head is %x\n", head); //输出头指针指向的地址

do

{

/\*

输出相应的值：当前节点地址、各字段值、当前节点的下一节点地址。

这样输出便于读者形象看到一个单向链表在计算机中的存储结构，和我们

设计的图示是一模一样的。

\*/

printf ("%x %d %5.1f %x\n", p, p->num, p->score, p->next);

p = p->next; //移到下一个节点

}

while (p != NULL);

}

}

/\*

==========================

功能：删除指定节点

(此例中是删除指定学号的节点)

返回：指向链表表头的指针

==========================

\*/

struct student \*Del (struct student \*head, int num)

{

struct student \*p1; //p1保存当前需要检查的节点的地址

struct student \*p2; //p2保存当前检查过的节点的地址

if (head == NULL) //是空链表（结合图3理解）

{

printf ("\nList is null!\n");

return head;

}

//定位要删除的节点

p1 = head;

while (p1->num != num && p1->next != NULL) //p1指向的节点不是所要查找的，并且它不是最后一个节点，就继续往下找

{

p2 = p1; //保存当前节点的地址

p1 = p1->next; //后移一个节点

}

if(p1->num==num) //找到了。（结合图4、5理解）

{

if (p1 == head) //如果要删除的节点是第一个节点

{

head = p1->next; //头指针指向第一个节点的后一个节点，也就是第二个节点。这样第一个节点就不在链表中，即删除

}

else //如果是其它节点，则让原来指向当前节点的指针，指向它的下一个节点，完成删除

{

p2->next = p1->next;

}

free (p1); //释放当前节点

p1 = NULL;

printf ("\ndelete %ld success!\n", num);

n -= 1; //节点总数减1个

}

else //没有找到

{

printf ("\n%ld not been found!\n", num);

}

return head;

}

//销毁链表

int DestroyList(struct student \*head)

{

struct student \*p;

if(head==NULL)

return 0;

while(head)

{

p=head->next;

free(head);

head=p;

}

return 1;

}

/\*

==========================

功能：插入指定节点的后面

(此例中是指定学号的节点)

返回：指向链表表头的指针

==========================

\*/

struct student \*Insert (struct student \*head, int num, struct student \*node)

{

struct student \*p1; //p1保存当前需要检查的节点的地址

if (head == NULL) //（结合图示7理解）

{

head = node;

node->next = NULL;

n += 1;

return head;

}

p1 = head;

while(p1->num != num && p1->next != NULL) //p1指向的节点不是所要查找的，并且它不是最后一个节点，继续往下找

{

p1 = p1->next; //后移一个节点

}

if (p1->num==num) //找到了（结合图示8理解）

{

node->next = p1->next; //显然node的下一节点是原p1的next

p1->next = node; //插入后，原p1的下一节点就是要插入的node

n += 1; //节点总数增加1个

}

else

{

printf ("\n%ld not been found!\n", num);

}

return head;

}

/\*

==========================

功能：反序节点

(链表的头变成链表的尾，链表的尾变成头)

返回：指向链表表头的指针

==========================

\*/

struct student \*Reverse (struct student \*head)

{

struct student \*p; //临时存储

struct student \*p1; //存储返回结果

struct student \*p2; //源结果节点一个一个取

p1 = NULL; //开始颠倒时，已颠倒的部分为空

p2 = head; //p2指向链表的头节点

while(p2 != NULL)

{

p = p2->next;

p2->next = p1;

p1 = p2;

p2 = p;

}

head = p1;

return head;

}

/\*

==========================

功能：选择排序(由小到大)

返回：指向链表表头的指针

==========================

\*/

struct student \*SelectSort (struct student \*head)

{

struct student \*first; //排列后有序链的表头指针

struct student \*tail; //排列后有序链的表尾指针

struct student \*p\_min; //保留键值更小的节点的前驱节点的指针

struct student \*min; //存储最小节点

struct student \*p; //当前比较的节点

first = NULL;

while(head != NULL) //在链表中找键值最小的节点

{

//注意：这里for语句就是体现选择排序思想的地方

for (p = head, min = head; p->next != NULL; p = p->next) //循环遍历链表中的节点，找出此时最小的节点

{

if (p->next->num < min->num) //找到一个比当前min小的节点

{

p\_min = p; //保存找到节点的前驱节点：显然p->next的前驱节点是p

min = p->next; //保存键值更小的节点

}

}

//上面for语句结束后，就要做两件事；一是把它放入有序链表中；二是根据相应的条件判断，安排它离开原来的链表

//第一件事

if (first == NULL) //如果有序链表目前还是一个空链表

{

first = min; //第一次找到键值最小的节点

tail = min; //注意：尾指针让它指向最后的一个节点

}

else //有序链表中已经有节点

{

tail->next = min; //把刚找到的最小节点放到最后，即让尾指针的next指向它

tail = min; //尾指针也要指向它

}

//第二件事

if (min == head) //如果找到的最小节点就是第一个节点

{

head = head->next; //显然让head指向原head->next,即第二个节点，就OK

}

else //如果不是第一个节点

{

p\_min->next = min->next; //前次最小节点的next指向当前min的next,这样就让min离开了原链表

}

}

if (first != NULL) //循环结束得到有序链表first

{

tail->next = NULL; //单向链表的最后一个节点的next应该指向NULL

}

head = first;

return head;

}

/\*

==========================

功能：直接插入排序(由小到大)

返回：指向链表表头的指针

==========================

\*/

struct student \*InsertSort (struct student \*head)

{

struct student \*first; //为原链表剩下用于直接插入排序的节点头指针

struct student \*t; //临时指针变量：插入节点

struct student \*p,\*q; //临时指针变量

first = head->next; //原链表剩下用于直接插入排序的节点链表：可根据图12来理解

head->next = NULL; //只含有一个节点的链表的有序链表：可根据图11来理解

while(first != NULL) //遍历剩下无序的链表

{

//注意：这里for语句就是体现直接插入排序思想的地方

for (t = first, q = head; ((q != NULL) && (q->num < t->num)); p = q, q = q->next); //无序节点在有序链表中找插入的位置

//退出for循环，就是找到了插入的位置，应该将t节点插入到p节点之后，q节点之前

//注意：按道理来说，这句话可以放到下面注释了的那个位置也应该对的，但是就是不能。原因：你若理解了上面的第3条，就知道了

//下面的插入就是将t节点即是first节点插入到p节点之后，已经改变了first节点，所以first节点应该在被修改之前往后移动，不能放到下面注释的位置上去

first = first->next; //无序链表中的节点离开，以便它插入到有序链表中

if (q == head) //插在第一个节点之前

{

head = t;

}

else //p是q的前驱

{

p->next = t;

}

t->next = q; //完成插入动作

//first = first->next;

}

return head;

}

/\*

==========================

功能：冒泡排序(由小到大)

返回：指向链表表头的指针

==========================

\*/

struct student \*BubbleSort (struct student \*head)

{

struct student \*endpt; //控制循环比较

struct student \*p; //临时指针变量

struct student \*p1,\*p2;

p1 = (struct student \*) malloc (LEN);

p1->next = head; //注意理解：我们增加一个节点，放在第一个节点的前面，主要是为了便于比较。因为第一个节点没有前驱，我们不能交换地址

head = p1; //让head指向p1节点，排序完成后，我们再把p1节点释放掉

for (endpt = NULL; endpt != head; endpt = p) //结合第6点理解

{

for (p = p1 = head; p1->next->next != endpt; p1 = p1->next)

{

if (p1->next->num > p1->next->next->num) //如果前面的节点键值比后面节点的键值大，则交换

{

p2 = p1->next->next; //结合第1点理解

p1->next->next = p2->next; //结合第2点理解

p2->next = p1->next; //结合第3点理解

p1->next = p2; //结合第4点理解

p = p1->next->next; //结合第6点理解

}

}

}

p1 = head; //把p1的信息去掉

head = head->next; //让head指向排序后的第一个节点

free (p1); //释放p1

p1 = NULL; //p1置为NULL，保证不产生“野指针”，即地址不确定的指针变量

return head;

}

/\*

==========================

功能：插入有序链表的某个节点的后面(从小到大)

返回：指向链表表头的指针

==========================

\*/

struct student \*SortInsert (struct student \*head, struct student \*node)

{

struct student \*p; //p保存当前需要检查的节点的地址

struct student \*t; //临时指针变量

if (head == NULL) //处理空的有序链表

{

head = node;

node->next = NULL;

n += 1; //插入完毕，节点总数加

return head;

}

p = head; //有序链表不为空

while(p->num < node->num && p != NULL) //p指向的节点的学号比插入节点的学号小，并且它不等于NULL

{

t = p; //保存当前节点的前驱，以便后面判断后处理

p = p->next; //后移一个节点

}

if (p == head) //刚好插入第一个节点之前

{

node->next = p;

head = node;

}

else //插入其它节点之后

{

t->next = node; //把node节点加进去

node->next = p;

}

n += 1; //插入完毕，节点总数加1

return head;

}

/\*

以上函数的测试程序：

提示：根据测试函数的不同注释相应的程序段，这也是一种测试方法。

\*/

int main(void)

{

struct student \*head;

struct student \*stu;

int thenumber;

// 测试Create()、Print()

head = Create();

Print(head);

//测试Del()

printf("\nWhich one delete: ");

scanf("%d",&thenumber);

head = Del(head,thenumber);

Print(head);

//测试Insert()

stu = (struct student \*)malloc(LEN);

printf("\nPlease input insert node -- num,score: ");

scanf("%d %f",&stu->num,&stu->score);

printf("\nInsert behind num: ");

scanf("%d",&thenumber);

head = Insert(head,thenumber,stu);

Print(head);

//测试Reverse()

printf("\nReverse the LinkList: \n");

head = Reverse(head);

Print(head);

//测试SelectSort()

printf("\nSelectSort the LinkList: \n");

head = SelectSort(head);

Print(head);

//测试InsertSort()

printf("\nInsertSort the LinkList: \n");

head = InsertSort(head);

Print(head);

//测试BubbleSort()

printf("\nBubbleSort the LinkList: \n");

head = BubbleSort(head);

Print(head);

printf("\nSortInsert the LinkList: \n");

//测试SortInsert():上面创建链表，输入节点时请注意学号num从小到大的顺序

stu = (struct student \*)malloc(LEN);

printf("\nPlease input insert node -- num,score: ");

scanf("%d %f",&stu->num,&stu->score);

head = SortInsert(head,stu);

Print(head);

//销毁链表

DestroyList(head);

printf ("\n");

system ("pause");

}