实验 3　线性表的链接存储结构实现

3.1　实 验 目 的

- 掌握线性表的链式结构的定义；
- 掌握链接存储结构结点的访问方法；
- 熟练运用链表的基本操作。

3.2　实 验 原 理

1. 链接存储结构

(1) 链接存储结构中的每个存储单元称为“结点”，结点包含一个数据域和一个指针域；数据域存放数据元素信息；指针域存放后继结点地址；数据元素之间的逻辑关系通过结点中的指针表示；访问链接存储结构通常由第一个结点开始，逐一访问所有结点。

(2) 线性表与链接存储结构的映射关系，用链式存储结构的结点存储线性表的数据元素；链接存储结构中结点指针域与后继结点的链接关系表示每个数据元素与其直接后继数据元素间的逻辑关系。

2. 带头结点的单链表(常用)

(1) 带头结点的单链表基本结构。

在链表的第一个结点之前附设一个结点，称为头结点。带头结点的单链表：头结点的引入使得单链表的头指针永远不为空，从而给插入、删除等操作带来了方便。基本结构如图 3-1 所示。

(2) 单链表(用结点指针实现)与顺序表(用数组实现)存储结构的对比如表 3-1 所示，方便理解。

3. 单链表的基本操作

以下 p、q 为指向任意结点的指针。

(1) 判空表：head->next==NULL。

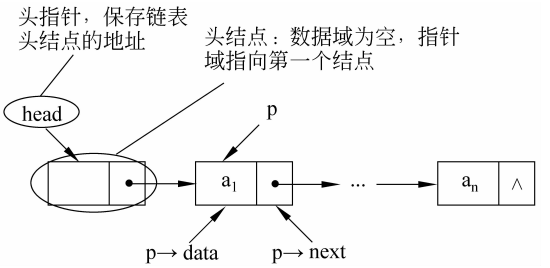
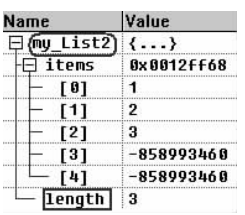
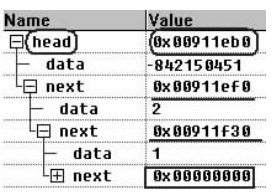


图 3-1　带头结点的单链表基本结构

表 3-1　顺序表与单链表调试下对比

顺序表调试下的存储结构	单链表调试下的存储结构
	

(2) 是否为表尾：p－＞next＝＝NULL。

(3) 指针后移：p＝p－＞next。

(4) 结点连接：p－＞next＝q(q 结点放在 p 结点之后)。

(5) 前驱：若 p－＞next＝＝q，则 p 指向 q 的前驱结点。

以上用到名称如图 3-2 所示。

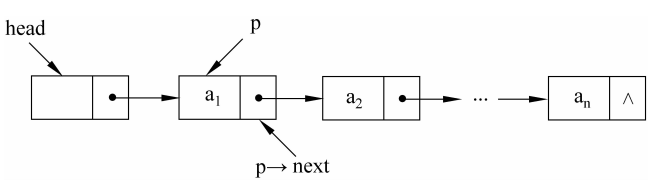


图 3-2　单链表

4. 单链表的重要基本操作

单链表的基本操作有查找结点、添加结点、删除结点。

1) 查找指定结点

设置一个跟踪链表结点的指针 p，初始时 p 指向链表中的第一个结点，然后顺着 next 域依次指向每个结点，每指向一个结点就判断其是否等于指定结点，若是，则返回该结点地址；否则继续向后搜索，直到 p 为 NULL，表示链表中无此元素，返回 NULL。

2) 链表任意位置插入法

单链表结点的插入是利用修改结点指针域的值，使其指向新的链接位置来完成插入

操作，无须移动任何元素。

假定在链表中指定结点之前插入一个新结点，要完成这种插入必须首先找到所插位置的前一个结点，再进行插入。假设指针 p 指向待插位置的前驱结点，指针 t 指向新结点。

关键语句：

```
Node *t=new Node; t->data=d;
t->next=p->next;
p->next=t;
```

3）链表表头插入法

在单链表的头结点之后第一个数据结点之前插入一个新结点，head 指向表头结点，head－＞next 表示表头的后继结点，指针 t 指向待插入结点。

关键语句：

```
Node *t=new Node; t->data=ai;
t->next=head->next;
head->next=t;
```

4）链表表尾插入法

关键语句：

```
Node *t=new Node; t->data=d; t->next=NULL;
last->next=t;
last=t;
```

5）删除指定位置的结点

（1）首先在单链表中找到被删除位置 i 的前一个结点 i－1，并用指针 p 指向 i－1，指针 t 指向被删除的结点 i。

（2）将指针 p 所指结点的指针域修改为 t 所指结点的后继。

（3）释放被删结点 t，即 delete(t)。

关键语句：

```
t=p->next;
p->next=t->next;
delete t;
```

3.3　实 验 要 求

- 理解结点的概念，理解结点中数据域与指针域各自的作用；
- 理解带头结点单链表的存储结构特点；
- 熟悉单链表的基本操作域重要操作；
- 本次实验在课堂内要提交第 1～第 3 题的文件(.cpp 和.h 文件内容)完整的相关代码。

3.4 实验内容与步骤

1. 用 chainlist.h 和 chainlist.cpp 创建工程

(1) 利用 chainlist.h 和 chainlist.cpp 文件内容创建主函数。

① chainlist.h 内容：

```
//chainlist.h
#include<iostream>
using namespace std;

typedef   char  DataType;

struct Node                          //Node 为结点类型名
{
    DataType     data;               //data 代表数据元素
    struct Node     * next;          //next 为指向下一结点的指针
};

//初始化单链表
int InitList(Node * &H);

//判表空
int ListEmpty(Node * H);

//求单链表中当前元素的个数
int ListLength(Node * H);

//遍历单链表
void TraverseList(Node * H);

//返回第一个与指定值匹配的元素位置
int Find_item(Node * H, DataType item);

//获取单链表中指定位置上的数据元素
int Find_pos(Node * H, int pos, DataType * item);

//向线性表指定位置插入一个新元素
int ListInsert (Node * H , int pos, DataType item);

//从线性表中删除第一个与指定值匹配的元素
int ListDelete (Node * H, DataType item);
```

```
//撤销单链表
void DestroyList(Node * &H);
```

② chainlist. cpp 程序清单。

```
#include "chainlist.h"

//初始化单链表
int InitList(Node * &H)
{                                          //H 为指向单链表的头指针
    H=new Node;
    if(!H)
    {
        cout<<"初始化错误"<<endl;
        return 0;
    }
    H->next=NULL;
    return 1;
}

//判表空
int ListEmpty(Node * H)
{                                          //H 为指向单链表的头指针
    if(H->next)
        return 0;
    else                                   //头结点指针域为空
        return 1;
}

//求单链表中当前元素的个数
int ListLength(Node * H)
{                                          //H 为指向单链表的头指针
    Node * p=H->next;
    int total=0;
    while(p){
        total++;                           //计数器+1
        p=p->next;                         //指针后移
    }
    return total;
}

//遍历单链表
void TraverseList(Node * H)
{                                          //H 为指向单链表的头指针
    Node * p=H->next;
```

```
    while(p)
    {
          cout<<p->data<<"  ";
        p=p->next;
    }
    cout<<endl;
}

//返回第一个与指定值匹配的元素位置
int Find_item(Node * H, DataType item)
{                                       //H 为指向单链表的头指针
    Node *p=H->next;
    int pos=0;                          //结点位序
    while(p)
    {                                   //从单链表第一个结点开始顺序查找所有结点
          pos++;
        if(p->data==item) break;
        p=p->next;
    }
    if(p) return pos;                   //返回位置编号
    else  return 0;
}

//获取单链表中指定位置上的数据元素
int Find_pos(Node * H, int pos, DataType *item)
{                                       //H 为指向单链表的头指针
    Node *p=H->next;
    int i=1;                            //结点位序
    while(p && i!=pos){
        p=p->next; i++;
    }
    if(p==NULL){                        //查找不成功,退出运行
        cout<<"位置无效"<<endl;
        return 0;
    }
    *item=p->data;
    return 1;
}

//向线性表指定位置插入一个新元素
int ListInsert (Node *H , int pos, DataType item)
{
    Node *p=H;  int i=0;
    while(p){                           //查找 pos 的前驱
```

```
        if(i+1==pos) break;
        p=p->next; i++;
    }
    if(p==NULL){                          //查找不成功,退出运行
        cout<<"插入位置无效"<<endl;
        return 0;
    }
    Node * t=new Node; t->data=item;  //a
    t->next=p->next;                  //b
    p->next=t;                        //c
    return 1;
}

//从线性表中删除第一个与指定值匹配的元素
int ListDelete (Node * H, DataType item)
{
    Node * p=H, * t;  int i=0;
    while(p->next){                   //查找 pos 的前驱
        if(p->next->data==item) break;
        p=p->next;
    }
    if(p->next==NULL){                //查找不成功,退出运行
        cout<<"删除元素不存在"<<endl;
        return 0;
    }
    t=p->next;                        //at 为被删除结点
    p->next=t->next;                  //b 删除 t 的链接关系
    delete t;                         //c 释放被删结点
    return 1;
}

//撤销单链表
void DestroyList(Node * &H)
{                                     //H 为指向单链表的头指针
    Node * p=H;
    while(H){
        p=H;
        H=H->next;
        delete p;
    }
}
```

(2) 观察线性表的各接口函数的函数名、参数表、返回值;明确接口函数的调用方法。思考若要改变线性表元素的类型应如何操作? 若添加新的接口函数,需要在哪些文件中

进行修改；

(3) 设计一个主程序完成如下功能：

① 定义管理单链表的头指针 head。

② 初始化单链表 head。

③ 在 head 的头部依次插入'a','b','c','d'元素(可思考如何在尾部插入)。

④ 若 head 非空,则输出 head 的长度及 head 中的元素。

⑤ 输出 head 的第三个元素。

⑥ 输出元素'a'的位置。

⑦ 在 head 第 3 个位置上插入元素'f',并输出 head 中元素。

⑧ 删除用户指定的任意元素,若删除成功则输出删除后 head 中的元素,删除失败给出适当的提示。

⑨ 释放单链表 head。

参考输出效果如图 3-3 所示。

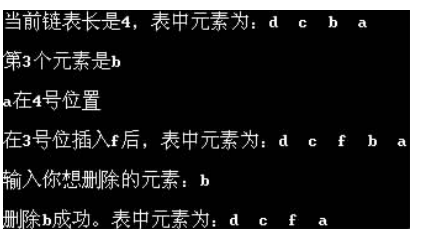


图 3-3 运行结果

如需了解程序运行过程,可在循环语句的循环体处设置断点,观察程序运行时,各变量的当前值。

(4) 总结使用单链表的一般步骤。

2. 设计接口函数

接口函数"int ListInsert_order (Node * H, DataType item);",向递增有序的单链表 H 中插入新的元素 item,插入后单链表仍然有序。

3. 课后练习

修改工程,存储学生信息{101 ,85},{103, 90.5},{104, 73},{105, 55},并且要让下面的主函数可以正常运行。

提示：

(1) 重新设计链表中的元素类型 DataType。

(2) 修改链表中的相关操作以符合新的元素类型。

(3) 设计新的接口,删除链表中学号为 number 的结点。

```
int main()
{
    Node *head;
    InitList(head);
    DataType A[]={{101 ,85},{103, 90.5},{104, 73},{105, 55}};

    for(int i=0;i<4;i++)
        ListInsert(head,i+1,A[i]);
    cout<<"学号 成绩"<<endl;
```

```
    TraverseList(head);

    ListDelete(head,103);

    cout<<"学号 成绩"<<endl;
    TraverseList(head);
    DestroyList(head);
    return 1;
}
```

3.5　实 验 参 考

1. 3.4 节第 1 题的参考

(1) 用下列代码建立主函数。

```
#include "chainlist.h"
int main()
{
    //定义管理单链表的头指针
    Node *head;
    DataType t;

    //初始化单链表 head
    InitList(head);

    //在 head 的头部依次插入 a, b, c, d 元素
    for(int i=0;i<4;i++)
        ListInsert(head,1,'a'+i);
/*
    //在 head 的尾部依次插入 a, b, c, d, e 元素
    for(int i=0;i<5;i++)
        ListInsert(head,i+1,'a'+i);
*/
    //若 head 非空,则输出 head 的长度及 head 中的元素
    if(ListEmpty(head)==1)
        cout<<"\n 单链表空!"<<endl;
    else
    {
        cout<<"\n 当前链表长是"<<ListLength(head)<<",表中元素为: ";
        TraverseList(head);              //遍历输出单链表
    }

    //输出 head 的第三个元素
```

```
    Find_pos(head,3,&t);
    cout<<"\n 第 3 个元素是"<<t<<endl;

    //输出元素 a 的位置
    cout<<"\na 在"<<Find_item(head,'a')<<"号位置"<<endl;

    //在 head 第 3 个位置上插入元素 f,并输出 head 中元素
    ListInsert(head,3,'f');              //F9 设置断点
    cout<<"\n 在 3 号位插入 f 后,表中元素为: ";
    TraverseList(head);

    //删除任意元素,删除成功则输出删除后 head 中的元素,删除失败给出提示
    cout<<"\n 输入你想删除的元素: "; cin>>t;
    if(ListDelete(head,t)==1)
    {
        cout<<"\n 删除"<<t<<"成功。表中元素为: ";
        TraverseList(head);
    }
    else
        cout<<"\n 删除\'"<<t<<"\'失败。表中元素没有变化!";

    //撤销单链表
    DestroyList(head);
    return 1;
}
```

(2) 将光标停在“ListInsert(head,3,'f');”所在行并按 F9 键设置断点,然后按 F5 键调试程序。当程序暂停时,在 watch 窗口中观察 head 如图 3-4 所示。

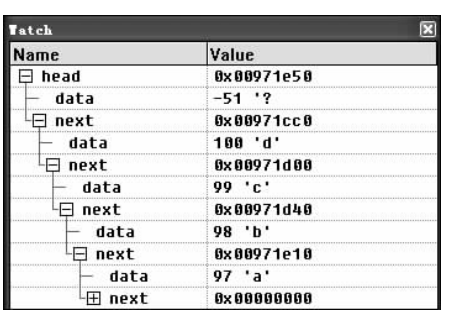


图 3-4 watch 窗口观察 head

(3) 可以观察到由 head 指针 **0x00971e50** 管理的单链表如图 3-5 所示。

(4) 按 F10 键 1 次运行程序,watch 窗口变化如图 3-6 所示。

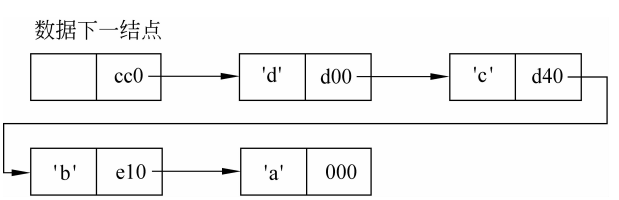


图 3-5 单链表

插入前与插入后链表的变化对比如表 3-2 所示。

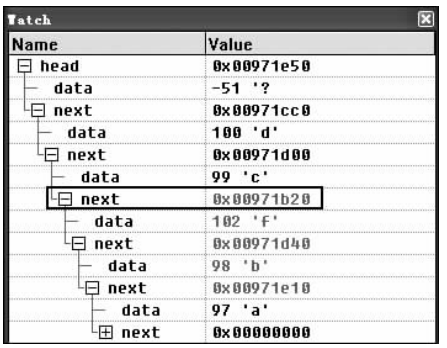


图 3-6　运行中链表的变化

表 3-2　插入前与插入后的对比

执行插入操作前				执行插入操作后			
结点序号	结点首地址	结点中的数据(data)	结点指向的后继结点(next)地址	结点序号	结点首地址	结点中的数据(data)	结点指向的后继结点(next)地址
0	e50		cc0	0	e50		cc0
1	cc0	‘d’	d00	1	cc0	‘d’	d00
2	d00	‘c’	d40	2	d00	‘c’	b20
3	d40	‘b’	e10	3	b20	‘f’	d40
4	e10	‘a’	000	4	d40	‘b’	e10
5				5	e10	‘a’	000

2. 3.4 节第 2 题的参考代码

```
int ListInsert_order (Node * H , DataType item)
{
    Node * p=H;
    //查找 item 的前驱 p
    while(p->next && item>p->next->data)
        p=p->next;
    Node * t=new Node;              //关键语句 a,b,c
    t->data=item;                   //a
    t->next=p->next;                //b
    p->next=t;                      //c
    return 1;
}
```

3. 3.4 节第 3 题的参考

在 chainlist.h 文件中课后练习：重新设计链表中的元素类型 DataType。

```
struct Grade
{
    int id;
    float score;
};
typedef  Grade  DataType;

//删除链表中学号为 number 的结点
int ListDelete (Node * H, int number);
```

在 chainlist.cpp 文件中,将 Find_item 函数注释。

```
//遍历单链表
void TraverseList(Node * H)
{  //H 为指向单链表的头指针
    Node * p=H->next;
    while(p)
    {
        cout<<p->data.id<<"  "<<p->data.score<<endl;
        p=p->next;
    }
    cout<<endl;
}
//删除链表中学号为 number 的结点
int ListDelete (Node * H, int number)
{
    Node * p=H, * t;
    while(p->next){                  //查找 number 的前驱
        if(p->next->data.id==number) break;
        p=p->next;
    }
    if(p->next==NULL){               //查找不成功,退出运行
        cout<<"删除元素不存在"<<endl;
        return 0;
    }
    //关键语句①,②,③
    t=p->next;                       //①t 为被删除结点
    p->next=t->next;                 //②删除 t 的链接关系
    delete t;                        //③释放被删结点
    return 1;
}
```

实验 4　栈的实现与应用

4.1　实 验 目 的

- 栈的顺序存储结构实现；
- 栈的链式存储结构实现；
- 理解栈的 LIFO 特征，并熟练运用栈解决一些实际算法问题。

4.2　实 验 原 理

1. 栈的特点

栈和队列是两种重要的线性结构。从数据结构角度看，栈和队列也是线性表，其特殊性在于栈和队列的基本操作是线性表操作的子集，它们是操作受限的线性表。从数据类型角度看，它们是和线性表不同的两类重要抽象数据类型。栈的特点是后进先出(Last In First Out，LIFO)。

2. 栈的定义

栈(Stack)是限定只能在表的一端进行插入和删除操作的线性表。

(1) 允许插入和删除运算的一端称作栈顶(top)。

(2) 不允许插入和删除的另一端称作栈底(bottom)。

(3) 在栈顶进行的插入操作称为入栈或进栈。

(4) 在栈顶进行的删除操作称为出栈或退栈。

3. 栈的抽象数据类型

ADT Stack。

① 数据元素集合：具有相同性质数据元素的一个有限序列，且只能在称为栈顶的一端进行插入和删除操作。

② 基本操作：

- 初始化栈(InitStack)：初始化栈。
- 入栈(Push)：在栈顶插入新的数据元素。
- 出栈(Pop)：删除栈顶数据元素。
- 取栈顶元素(GetTop)：获取栈顶的数据元素。
- 判栈空(StackEmpty)：判断栈是否为空栈。

4. 栈的顺序存储结构与基本运算的实现

(1) 用数组存储栈元素。

(2) 用数组元素之间的位置表示栈的逻辑结构。

(3) 用最后一个元素的下标指示栈的栈顶。

5. 栈的链式存储结构与基本运算的实现

(1) 用结点的数据域存储栈元素。

(2) 用结点的指针域表示栈的逻辑结构。

(3) 用单链表的第一个结点指示栈的栈顶。

4.3 实 验 要 求

- 熟悉栈的特性及操作特点；
- 理解栈的顺序存储结构特点；
- 理解栈的链式存储结构特点；
- 用本实验提供的 5 个文件(.cpp、.h)组建工程，并按以下步骤调试程序，分别验证顺序存储结构和链式存储结构栈的入栈和出栈操作。

4.4 实验内容与步骤

1. 顺序存储结构实现栈

(1) 入栈“int Push(SqStack &S, DataType item);”。

在入栈接口“Push(my_stack1,i);”处设置断点①，按 F5 键启动调试，按 F10 键逐句执行，直到数据 11～20 全部入栈。程序暂停时观察栈顶数据 my_stack1.items[my_stack1.top]和栈顶位置 my_stack1.top；栈的相关数据变化如表 4-1 所示。

表 4-1　顺序栈入栈的相关数据变化情况

入栈序号	栈中全部数据 my_stack1.items	栈顶位置 my_stack1.top	栈顶数据 my_stack1.items[my_stack1.top]
0	无	−1	无
1	11	0	11
2	11,12	1	12
3	11,12,13	2	13
4	11,12,13,14	3	14
5	11,12,13,14,15	4	15
6	11,12,13,14,15,16	5	16
7	11,12,13,14,15,16,17	6	17
8	11,12,13,14,15,16,17,18	7	18
9	11,12,13,14,15,16,17,18,19	8	19
10	11,12,13,14,15,16,17,18,19,20	9	20

进入调试可以看到入栈的过程，相关信息如图 4-1 所示。

图 4-1　顺序栈入栈相关数据变化调试监视信息

(2) 出栈“int Pop(SqStack &S, DataType &item);”。

在“Pop(my_stack1, result);”处设置断点②，按 F5 键启动调试，按 F10 键逐句执行，直到所有数据完全出栈；出栈的相关数据变化情况如表 4-2 所示。

表 4-2　顺序栈出栈的相关数据变化情况

出栈序号	栈中全部数据 my_stack1.items	栈顶位置 my_stack1.top	栈顶数据 my_stack1.items[my_stack1.top]
0	11,12,13,14,15,16,17,18,19,20	9	20
1	11,12,13,14,15,16,17,18,19	8	19
2	11,12,13,14,15,16,17,18	7	18
3	11,12,13,14,15,16,17	6	17
4	11,12,13,14,15,16	5	16
5	11,12,13,14,15	4	15
6	11,12,13,14	3	14
7	11,12,13	2	13
8	11,12	1	12
9	11	0	11
10	无	−1	无

注意：顺序存储时，出栈并不会清除原来存储的值，由 top 指向栈顶即可。只有当有新值入栈时才会覆盖旧值。出栈调试相关信息如图 4-2 所示。

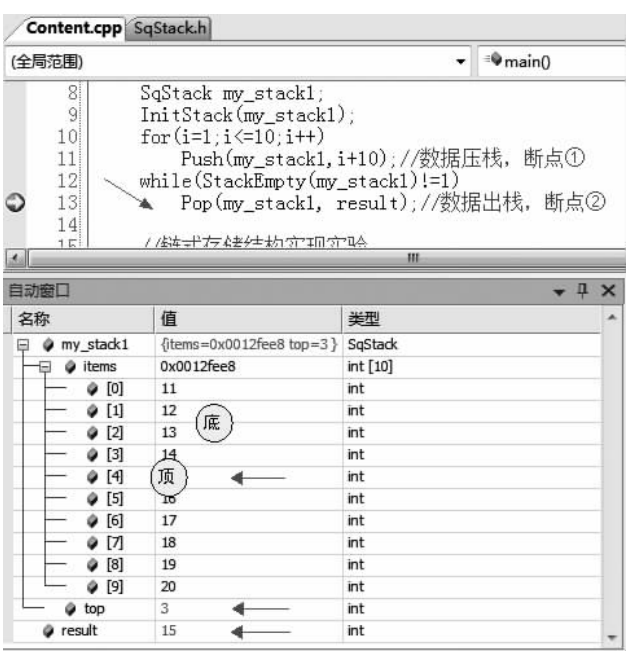


图 4-2　顺序栈出栈相关数据变化调试监视信息

2. 链式存储结构实现栈

(1) 入栈“int Push(SNode * top, DataType item);”。

在入栈接口“Push(my_stack2,i);”处设置断点③,按 F5 键启动调试,按 F10 键逐句执行,直到所有数据全部入栈。当程序暂停时观察栈中各结点的数据、栈顶指针的变化情况;入栈的相关数据变化情况如表 4-3 所示。

表 4-3　链式栈入栈的相关数据变化情况

入栈序号	栈中全部数据 my_stack2	栈顶数据 my_stack2－＞next－＞data
0	无	无
1	11	11
2	11,12	12
3	11,12,13	13
4	11,12,13,14	14
5	11,12,13,14,15	15
6	11,12,13,14,15,16	16
7	11,12,13,14,15,16,17	17
8	11,12,13,14,15,16,17,18	18
9	11,12,13,14,15,16,17,18,19	19
10	11,12,13,14,15,16,17,18,19,20	20

调试可以看到链栈的入栈过程。进入调试可以看到入栈的过程,相关信息如图 4-3 所示。

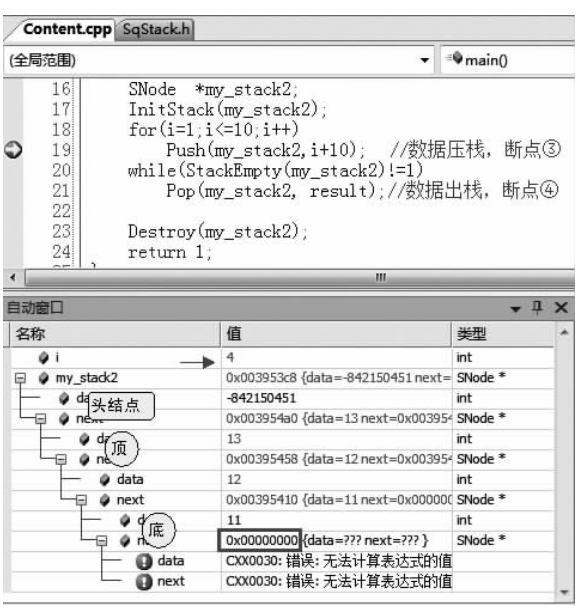


图 4-3　链式栈入栈相关数据变化调试监视信息

（2）出栈“int Pop(SNode *top, DataType &item);”。

在“Pop(my_stack2, result);”处设置断点④，按 F5 键启动调试，按 F10 键逐句执行，直到所有数据完全出栈；出栈的相关数据变化情况如表 4-4 所示。

表 4-4 链式栈出栈的相关数据变化情况

出栈序号	栈中全部数据 my_stack2	栈顶数据 my_stack2->next->data
0	11,12,13,14,15,16,17,18,19,20	20
1	11,12,13,14,15,16,17,18,19	19
2	11,12,13,14,15,16,17,18	18
3	11,12,13,14,15,16,17	17
4	11,12,13,14,15,16	16
5	11,12,13,14,15	15
6	11,12,13,14	14
7	11,12,13	13
8	11,12	12
9	11	11
10	无	无

出栈时，会清除栈顶结点，释放空间。调试可以看到链栈的出栈过程。出栈调试信息如图 4-4 所示。

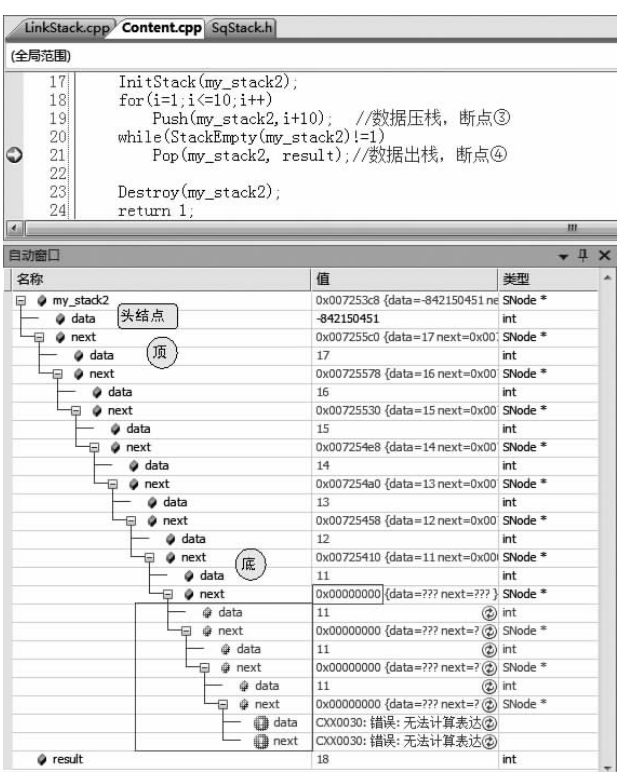


图 4-4 链式栈出栈相关数据变化调试监视信息

3. 设计新接口并在 content.cpp 文件中测试

(1) 设计顺序栈的遍历接口“int TraverseStack(SqStack &S);”,实现栈中数据按出栈的顺序输出,但保留栈中的数据。

(2) 设计链栈的遍历接口“int TraverseStack(SNode *top);”,实现栈中数据按出栈的顺序输出,但保留栈中的数据。

4. 链栈的应用

编写函数,判断给定的字符串是否回文。例如,字符串“abcba”、“abccba”均为中心对称,字符串“abcdba”不是中心对称,将中心对称的字符串称为回文。

要求:利用本实验已实现的链栈基本操作来实现,并进行测试。

函数原型:

```
int IsReverse(char *s);
```

函数功能:判断字符串 s 是否为回文,若是则返回 1,否则返回 0。

提示:先将字符串内容全部入栈,然后执行出栈操作,依次判断每次出栈的字符是否与字符串中对应正向字符相同,如果出现不相同的字符则说明字符串不是回文,全部都相同则是回文。

4.5 实 验 参 考

1. 4.4 节实验内容的第 3 题参考

(1) 设计顺序栈的遍历接口“int TraverseStack(SqStack &S);”,实现栈中数据按出栈的顺序输出,但保留栈中的数据。

```
//按出栈顺序遍历栈
int TraverseStack(SqStack &S)
{
    if(StackEmpty(S)==1)
    {
        cout<<"栈空"<<endl;
        return 0;                          //栈空,遍历失败
    }
    for(int i=S.top; i>=0; i--)
        cout<<S.items[i]<<" ";
    cout<<endl;
    return 1;                              //遍历成功
}
```

(2) 设计链栈的遍历接口“int TraverseStack(SNode *top);”,实现栈中数据按出栈的顺序输出,但保留栈中的数据。

```
//按出栈顺序遍历栈
int TraverseStack(SNode *top)
{
    if(StackEmpty(top)==1)
    {
        cout<<"空栈"<<endl;
        return 0;                          //空栈,遍历失败
    }
    SNode *p=top->next;                    //不能移动栈顶指针
    while(p)
    {
        cout<<p->data<<" ";
        p=p->next;
    }
    cout<<endl;
    return 1;                              //遍历成功
}
```

2.4.4 节实验内容的第 4 题参考

```
//判断字符串 s 是否为回文
int IsReverse(char *s)
{
    SNode *stack;                          //定义管理链栈的栈顶指针
    DataType t;
    InitStack(stack);                      //初始化链栈
    for(int i=0; i<strlen(s); i++)         //将字符串中的内容入栈
        Push(stack,s[i]);
    i=0;
    while(StackEmpty(stack)!=1)
    {
        Pop(stack,t);                      //出栈
        //比较出栈元素与字符串正向元素是否相同
        if(t!=s[i])
            return 0;                      //不是回文
        else
            i++;
    }
    return 1;                              //是回文
}
```