**顺序栈的验证与改造**

（2）修改push

当顺序表满时，建一个更长的顺序表，把原顺序表的元素依次放入新顺序表，最后清除掉原顺序表

template<class ElemType>

Status SeqStack<ElemType>::Push(const ElemType e)

// 操作结果：将元素e追加到栈顶,如成功则返加SUCCESS,如栈已满将返回OVER\_FLOW

{

if (top == maxSize - 1) // 栈已满

{

Stackfull();

}

else { // 操作成功

elems[++top] = e; // 将元素e追加到栈顶

return SUCCESS;

}

}

template<class ElemType>

Status SeqStack<ElemType>::Stackfull()

{

maxSize=2\*maxSize;

ElemType newelems[maxSize];

for(int i=0;i<top;i++)

{

newelems[i]=elems[i];

}

delete[]elems;

elems=newelems;

}

**链式栈的验证与改造**

增加一个Car类

class Car

{

string id;

int year,month,day,hour,minute,second;

public:

Car(){};

Car(string i,int y,int mon,int d,int h,int m,int s):id(i),year(y),month(mon),day(d),hour(h),minute(m),second(s){}

scan(){cout<<id<<" "<<year<<"y"<<month<<"m"<<day<<"d"<<hour<<"h"<<minute<<"m"<<second<<"s"<<endl;}

};

其中scan函数用来输出

在LinkStack类中增加一个语句

Node<ElemType>\* getTop(){return top;}

用来返回调用top

主函数如下

int main(void)

{

LinkStack<Car> sa;

Car car1("ferrari",2017,11,21,5,12,11);

Car car2("lamborgini",2017,12,27,22,8,12);

sa.Push(car1);

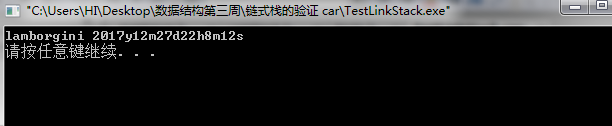
sa.Push(car2);

sa.getTop()->data.scan();

system("PAUSE");

return 0;

}



（2）对链式栈修改如下

template<class ElemType>

class LinkStack

{

protected:

// 链栈的数据成员:

Node<ElemType> \*top; // 栈顶指针

public:

// 链栈的函数成员:

LinkStack(); // 无参数的构造函数

virtual ~LinkStack(); // 析构函数

int GetLength() const; // 求栈长度

bool IsEmpty() const; // 判断栈是否为空

void Clear(); // 将栈清空

void Traverse(void (\*Visit)(const ElemType &)) const ; // 遍历栈

Status Push(const ElemType e); // 入栈

Status Top(ElemType &e) const; // 返回栈顶元素

Status Pop(ElemType &e); // 出栈

LinkStack(const LinkStack<ElemType> &s); // 复制构造函数

LinkStack<ElemType> &operator =(const LinkStack<ElemType> &s); // 赋值语句重载

};

// 链栈类的实现部分

template<class ElemType>

LinkStack<ElemType>::LinkStack()

// 操作结果：构造一个空栈表

{

top = new Node<ElemType>();

}

template<class ElemType>

LinkStack<ElemType>::~LinkStack()

// 操作结果：销毁栈

{

Clear();

}

template <class ElemType>

int LinkStack<ElemType>::GetLength() const

// 操作结果：返回栈元素个数

{

int count = 0; // 计数器

Node<ElemType> \*p;

for (p = top->next; p != NULL; p = p->next) // 用p依次搜寻每个元素

count++; // 统计链栈中结点数

return count;

}

template<class ElemType>

bool LinkStack<ElemType>::IsEmpty() const

// 操作结果：如栈为空，则返回true，否则返回false

{

return top->next == NULL;

}

template<class ElemType>

void LinkStack<ElemType>::Clear()

// 操作结果：清空栈

{

Node<ElemType> \*p;

while (top->next != NULL) {

p = top->next;

top->next = p->next;

delete p;

}

}

template <class ElemType>

void LinkStack<ElemType>::Traverse(void (\*Visit)(const ElemType &)) const

// 操作结果：从栈顶到栈底依次对栈的每个元素调用函数(\*visit)访问

{

Node<ElemType> \*p;

for (p = top->next; p != NULL; p = p->next) // 用p依次搜寻当前栈的每个元素

(\*Visit)(p->data); // 对p所指向的元素调用函数(\*visit)访问

}

template<class ElemType>

Status LinkStack<ElemType>::Push(const ElemType e)

// 操作结果：将元素e追加到栈顶,如成功则返加SUCCESS,否则如动态内存已耗尽

// 将返回OVER\_FLOW

{

Node<ElemType> \*p = new Node<ElemType>(e, top->next);

if (p == NULL) // 系统内存耗尽

return OVER\_FLOW;

else { // 操作成功

top->next = p;

return SUCCESS;

}

}

template<class ElemType>

Status LinkStack<ElemType>::Top(ElemType &e) const

// 操作结果：如栈非空,用e返回栈顶元素,函数返回SUCCESS,否则函数返回UNDER\_FLOW

{

if(IsEmpty()) // 栈空

return UNDER\_FLOW;

else { // 栈非空,操作成功

e = top->next->data; // 用e返回栈顶元素

return SUCCESS;

}

}

template<class ElemType>

Status LinkStack<ElemType>::Pop(ElemType &e)

// 操作结果：如栈非空,删除栈顶元素,并用e返回栈顶元素,函数返回SUCCESS,否则

// 函数返回UNDER\_FLOW

{

if (IsEmpty()) // 栈空

return UNDER\_FLOW;

else { // 操作成功

Node<ElemType> \*p = top->next; // 保留原栈顶

e = top->next->data; // 用e返回栈顶元素

top->next = top->next->next; // 修改栈顶

delete p; // 删除原栈顶结点

return SUCCESS;

}

}

template<class ElemType>

LinkStack<ElemType>::LinkStack(const LinkStack<ElemType> &s)

// 操作结果：由栈s构造新栈--复制构造函数

{

if (s.IsEmpty()) // s为空

top = NULL; // 构造一空栈

else { // s非空,复制栈

top = new Node<ElemType>(); // 生成当前栈项

Node<ElemType> \*q = top; // 设置当前栈底指针

for (Node<ElemType> \*p = s.top->next; p != NULL; p = p->next){

q->next = new Node<ElemType>(p->data); // 向栈底追加元素

q = q->next; // 修改栈底指针

}

}

}

template<class ElemType>

LinkStack<ElemType> &LinkStack<ElemType>::operator = (const LinkStack<ElemType> &s)

// 操作结果：将栈s赋值给当前栈--赋值语句重载

{

if (&s != this) {

Clear(); // 清空当前栈

if (!s.IsEmpty()) { // s非空,复制栈

top = new Node<ElemType>(); // 生成当前栈项

Node<ElemType> \*q = top; // 设置当前栈底指针

for (Node<ElemType> \*p = s.top->next; p != NULL; p = p->next){

q->next = new Node<ElemType>(p->data); // 向栈底追加元素

q = q->next; // 修改栈底指针

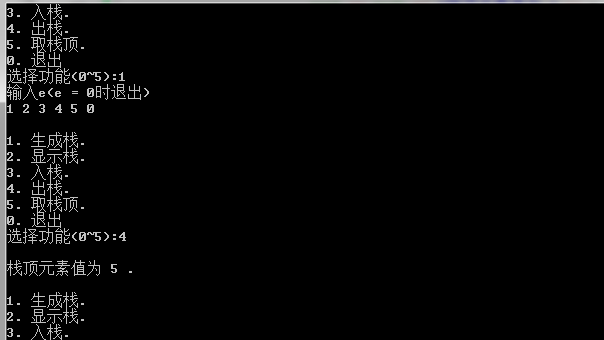
}

}

}

return \*this;

}



**设计性实验**

修改SeqStack如下

// 顺序栈模板类

template<class ElemType>

class SeqStack

{

protected:

// 顺序栈的数据成员:

int top1,top2; // 栈顶指针

int maxSize; // 栈的最大容量

ElemType \*elems; // 元素存储空间

public:

// 顺序栈的方法声明及重载编译系统默认方法声明:

SeqStack(int size = DEFAULT\_SIZE); // 构造函数

virtual ~SeqStack(); // 析构函数

int GetLength(int k) const; // 求栈的长度

bool IsEmpty(int k) const; // 判断栈是否为空

void Clear(int k); // 将栈清空

void Traverse(int k,void (\*Visit)(const ElemType &)) const; // 遍历栈

Status Push(int k,const ElemType e); // 入栈

Status Top(int k,ElemType &e) const; // 取顶元素

Status Pop(int k,ElemType &e); // 出栈

SeqStack(const SeqStack<ElemType> &s); // 复制构造函数

SeqStack<ElemType> &operator =(const SeqStack<ElemType> &s); // 赋值语句重载

};

// 顺序栈类的实现部分

template<class ElemType>

SeqStack<ElemType>::SeqStack(int size)

// 操作结果：构造一个最大容量为size的空栈

{

maxSize = size; // 栈的最大容量

if (elems != NULL) delete []elems; // 释放已有存储空间

elems = new ElemType[maxSize]; // 分配新的存储空间

top1= -1;

top2= maxSize;

}

template<class ElemType>

SeqStack<ElemType>::~SeqStack()

// 操作结果：销毁栈

{

delete []elems; // 释放栈的存储空间

}

template <class ElemType>

int SeqStack<ElemType>::GetLength(int k) const

// 操作结果：返回栈中元素个数

{

if(k==1){return top1 + 1;}

if(k==2){return maxSize-top2;}

}

template<class ElemType>

bool SeqStack<ElemType>::IsEmpty(int k) const

// 操作结果：如栈为空，则返回true，否则返回false

{

if(k==1){return top1== -1;}

if(k==2){return top2==maxSize;}

}

template<class ElemType>

void SeqStack<ElemType>::Clear(int k)

// 操作结果：清空栈

{

if(k==1){top1 = -1;}

if(k==2){top2=maxSize;}

}

template <class ElemType>

void SeqStack<ElemType>::Traverse(int k,void (\*Visit)(const ElemType &)) const

// 操作结果：从栈顶到栈底依次对栈的每个元素调用函数(\*visit)访问

{

if(k==1)

{

for (int i = top1; i >=0 ; i--)

(\*Visit)(elems[i]);

}

if(k==2)

{

for(int i = top2;i<maxSize;i++)

(\*Visit)(elems[i]);

}

}

template<class ElemType>

Status SeqStack<ElemType>::Push(int k,const ElemType e)

// 操作结果：将元素e追加到栈顶,如成功则返加SUCCESS,如栈已满将返回OVER\_FLOW

{

if (top2-top1==1) // 栈已满

return OVER\_FLOW;

else

{ // 操作成功

if(k==1)

{

elems[++top1] = e; // 将元素e追加到栈顶

return SUCCESS;

}

if(k==2)

{

elems[--top2] = e;

return SUCCESS;

}

}

}

template<class ElemType>

Status SeqStack<ElemType>::Top(int k,ElemType &e) const

// 操作结果：如栈非空,用e返回栈顶元素,函数返回SUCCESS,否则函数返回UNDER\_FLOW

{

if(IsEmpty(k)) // 栈空

return UNDER\_FLOW;

else

{ // 栈非空,操作成功

if(k==1){e = elems[top1];} // 用e返回栈顶元素

if(k==2){e = elems[top2];}

return SUCCESS;

}

}

template<class ElemType>

Status SeqStack<ElemType>::Pop(int k,ElemType &e)

// 操作结果：如栈非空,删除栈顶元素,并用e返回栈顶元素,函数返回SUCCESS,否则

// 函数返回UNDER\_FLOW

{

if (IsEmpty(k)) // 栈空

return UNDER\_FLOW;

else { // 操作成功

if(k==1){e = elems[top1--]; }

if(k==2){e = elems[top2++];}// 用e返回栈顶元素

return SUCCESS;

}

}

template<class ElemType>

SeqStack<ElemType>::SeqStack(const SeqStack<ElemType> &s)

// 操作结果：由栈s构造新栈--复制构造函数

{

maxSize = s.maxSize; // 栈的最大容量

if (elems != NULL) delete []elems; // 释放已有存储空间

elems = new ElemType[maxSize]; // 分配存储空间

top1 = s.top1;

top2 = s.top2; // 复制栈顶指针

for (int i = 0; i <= top1; i++) // 从栈底到栈顶对栈s的每个元素进行复制

elems[i] = s.elems[i];

for (int i=maxSize-1;i>=top2;i--)

elems[i] = s.elems[i];

}

template<class ElemType>

SeqStack<ElemType> &SeqStack<ElemType>::operator = (const SeqStack<ElemType> &s)

// 操作结果：将栈s赋值给当前栈--赋值语句重载

{

if (&s != this) {

maxSize = s.maxSize; // 栈的最大容量

if (elems != NULL) delete []elems; // 释放已有存储空间

elems = new ElemType[maxSize]; // 分配存储空间

top1 = s.top1;

top2 = s.top2; // 复制栈顶指针

for (int i = 0; i <= top1; i++) // 从栈底到栈顶对栈s的每个元素进行复制

elems[i] = s.elems[i];

for (int i=maxSize-1;i>=top2;i--)

elems[i] = s.elems[i];

}

return \*this;

}

如图为所示的主函数以及测试结果

