实验题第 1 题

(1) 设计思路

本题要求设计共享栈的类定义，实现判断栈空、入栈和出栈的函数。此时，两个栈的栈底分别设置在数组的两端，入栈时栈顶向数组中间移动，当两个栈的栈顶位置相遇时才是栈满状态。为了实现这一目的，可以在一个存储空间里设置两个栈顶指针，分别向后与向前移动。可以利用指针数值与容量的代数关系，找到栈满时两指针满足的条件，从而实现判空、入栈、出栈等操作。

(2) 源代码

SqStack(int size = DEFAULT_SIZE); //构造函数

```
template<class DataType>
SqStack<DataType>::SqStack(int size)     //构造一个容量为 size 的共享栈
{
  elems = new DataType[size];    //开辟存储空间，创建共享栈
  maxSize = size;
  top1 = -1;                     //构造栈 1
  top2 = size;                   //构造栈 2
}
```

Status Push(int id, const DataType &e); //入栈

```
template<class DataType>
Status SqStack<DataType>::Push(int id, const DataType &e)
{
  if(id == 1)
  {                          //如果 id 为 1，对栈 1 执行入栈
    if (top1 + (maxSize - top2 + 1) == maxSize - 1)      //判断栈 1 是否已满
      return OVER_FLOW;      //栈已满，返回上溢信息
    else                     //栈未满，执行入栈
    {
      elems[++top1] = e;     //top1+1 即为新的栈顶位置，将元素 e 放入该位置
      return SUCCESS;        //返回成功信息
    }
  }
  else
  {                          //如果 id 不为 1，对栈 2 执行入栈
    if (top1 + (maxSize - top2 + 1) == maxSize - 1)      //判断栈 2 是否已满
      return OVER_FLOW;      //栈已满，返回上溢信息
    else                     //栈未满，执行入栈
    {
      elems[--top2] = e;     //top2-1 即为新的栈顶位置，将元素 e 放入该位置
      return SUCCESS;        //返回成功信息
    }
  }
}
```

（下见次页）

Status Pop(int id, DataType &e); //出栈

```
template<class DataType>
Status SqStack<DataType>::Pop(int id, DataType &e)  //若出栈成功，用 e 返回栈顶元素的值
{
  if (id == 1)                //判断对栈 1 还是栈 2 进行出栈
  {
    if (IsEmpty(id))          //判断栈 1 是否为空
      return UNDER_FLOW;      //栈空，返回下溢信息
    else {                    //栈不为空，执行出栈
      e = elems[top1--];      //获取当前栈顶元素的值，并删除该元素，栈顶指针前移
      return SUCCESS;         //返回成功信息
    }
  }
  else
  {
    if (IsEmpty(id))          //判断栈 2 是否为空
      return UNDER_FLOW;      //栈空，返回下溢信息
    else {                    //栈不为空，执行出栈
      e = elems[top2++];      //获取当前栈顶元素的值，并删除该元素，栈顶指针后移
      return SUCCESS;         //返回成功信息
    }
  }
}
```

bool IsEmpty(int id) const; //判断栈空

```
template<class DataType>
bool SqStack<DataType>::IsEmpty(int id) const   //利用共享栈两边为栈顶的特性，分别判断栈空
{
  if(id == 1)                 //判断对栈 1 还是栈 2 进行操作
  {
    return top1 == -1;        //如果栈顶指针 1 指向-1，则栈 1 空
  }
  return top2 == maxSize;     //如果栈顶指针 2 指向 maxSize，则栈 2 空
}
```

(3) 说明

上述代码可以应用测试程序进行测试并得到正确结果。

实验题第 2 题

(1) 设计思路

本题要求**利用两个栈 s1 和 s2 的操作模拟一个队列的操作，写出使用两个栈实现入队、出队和判队列空的函数**。为了达成这一目的，需要保证队列“FIFO”（先进先出）的特性。而栈的特性是“FILO”（先进后出），那么就需要想办法利用元素在两个内部栈之间的转移，实现 FILO 到 FIFO 的转变。如果将栈顶元素出栈后立刻入栈到一个空栈，那么就会成为后者的栈底元素，以此类推，原栈的栈底元素将成为新栈的栈顶元素，所有元素次序实现完全倒换。根据这个原理，可将其中一个栈作为入队栈，另一个作为出队栈。每次执行出队/入队操作时，先将所有元素推入相应的栈，另一个栈清空，就可以确保出队/入队时，非空栈的栈顶永远是需要操作的元素或位置，模拟了队列的 FIFO 特性。

(2) 源代码

Queue.h　　//定义队列类

```
template <class DataType>
class Queue
{
    SqStack<DataType> *stk1, *stk2;          //两个构成队列的内部栈
    DataType tmp;                            //缓存变量
public:
    Queue(int size = DEFAULT_SIZE);          //构造函数
    virtual ~Queue();                        //构造函数
    bool IsEmpty() const;                    //判断队空
    Status EnQueue(const DataType &e);       //入队
    Status DeQueue(DataType &e);             //出队
};
```

Queue(int size = DEFAULT_SIZE);　　//构造函数

```
template<class DataType>
Queue<DataType>::Queue(int size)             //构造容量为 size 的队列
{
    stk1 = new SqStack<DataType>(size/2);    //取 size 的一半作为 stk1（栈 1）的容量，作入队栈
    stk2 = new SqStack<DataType>(size/2);    //取 size 的一半作为 stk2 的容量，作出队栈
}
```

bool IsEmpty() const;　　//判断队列是否为空

```
template<class DataType>
bool Queue<DataType>::IsEmpty() const        //判断队列是否为空
{
    return (stk1->IsEmpty() && stk2->IsEmpty());  //两栈皆空，则队列空
}
```

(下见次页)

Status EnQueue(const DataType &e); //入队

```
template<class DataType>
Status Queue<DataType>::EnQueue(const DataType &e)  //入队
{
    for (int i = stk2->top; i >= 0 ; i--)  //先把 stk2 的元素全部出栈，入栈到 stk1 以改序
    {
        stk2->Pop(tmp);                    //stk2 栈顶元素出栈，存入 tmp
        stk1->Push(tmp);                   //tmp 入栈 stk1，栈顶转换为栈底
    }
    stk1->Push(e);                         //再将要入队的元素 push 到入队栈 stk1
    return SUCCESS;                        //返回成功信息
}
```

Status DeQueue(DataType &e); //出队

```
template<class DataType>
Status Queue<DataType>::DeQueue(DataType &e)  //出队
{
    for (int i = stk1->top; i >= 0 ; i--)  //先把 stk1 的元素全部出栈，入栈到 stk2 以改序
    {
        stk1->Pop(tmp);                    //stk1 栈顶元素出栈，存入 tmp
        stk2->Push(tmp);                   //tmp 入栈 stk2，栈顶转换为栈底
    }
    stk2->Pop(e);                          //再将要出队的元素从出队栈 stk1 Pop 出去
    return SUCCESS;                        //返回成功信息
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

实验题第 3 题

(1) 设计思路

本题要求修改有关循环队列的设计，取消队尾指针 rear，以 front 和 length 分别表示循环队列中的队头位置和队列中所含元素的个数。试完成循环队列判断队空、入队和出队函数的实现。为了达成这一目的，需要在用 length 和 front 来表达队列数据信息的同时，能够完成对队列元素的循环操作。粗略来看，top+length 就可以表示队尾元素的位置，而 length 可以直接用于判断是否队空。需要特别注意的是，为了兼顾指针的循环性，队尾指针需要用 front 对 maxSize 取余后加上 length 再次取余来表示，即 (front%maxSize+length)%maxSize. 其余操作部分作适当修改即可。

(2) 源代码

SqQueue(int size = DEFAULT_SIZE); //构造函数

```
template<class DataType>
SqQueue<DataType>::SqQueue(int size)  //构造一个容量为 size 的空队列
{
  elems = new DataType[size];         //创建数组 elems 作为队列容器存放变量
  maxSize = size;                     //将形参 size 值传递给内部成员变量 maxsize
  front = length = 0;                 //队头指针、长度初始化为 0
}
```

bool IsEmpty() const; //判断队列是否为空

```
template<class DataType>
bool SqQueue<DataType>::IsEmpty() const       //定义布尔类型函数，判断队列是否为空
{
  return length == 0;                         //如果 length 为 0，返回 true，否则返回 false
}
```

Status EnQueue(const DataType &e); //入队

```
template<class DataType>
Status SqQueue<DataType>::EnQueue(const DataType &e)
{
  if(length == maxSize)                          //判断队列是否已满
    return OVER_FLOW;                            //队列已满，返回上溢信息
  else {                                         //队列未满
    elems[(front%maxSize+length)%maxSize] = e;   //将 e 放入队尾位置
    length++;                                    //长度+1
    return SUCCESS;                              //返回入队成功信息
  }
}
```

（下见次页）

Status DelQueue(DataType &e); //出队

```
template<class DataType>
Status SqQueue<DataType>::DelQueue(DataType &e)
{
  if(IsEmpty())                          //判断队列是否为空
    return UNDER_FLOW;                   //队列为空，返回下溢信息
  else {                                 //队列不为空
    e = elems[front];                    //获取队头元素的值
    front = (front + 1) % maxSize;       //队头指针后移 1 位即删除队头元素
    length--;                            //长度-1
    return SUCCESS;                      //返回出队成功信息
  }
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

实验题第 4 题

(1) 设计思路

本题要求**写出利用队列实现输出杨辉三角形前 n 行的算法。对于第 i（i=1, 2, 3, 4, 5）行，首先输出 n-i 个空格；然后输出该行数据，数据之间用 1 个空格间隔；最后一个数据的后面有 1 个空格**。为了达成这一目的，可以按照**出队**、**相加**、**再入队**的步骤逐行输出杨辉三角的元素。为此，我选择建立一个嵌套循环，外层执行每一行的遍历；内层循环用于对当前行的元素进行输出，并计算产生下一行的元素。每一次内层循环，**首先**取队头元素执行删除并输出，保证输出元素不重复；**然后**进行加和计算，计算过程中，可以利用两个中间变量 t1 和 t2 来滑动存储每一对出队元素的信息，每一次内层循环都会将**本次出队**和**上次出队**（循环开始前记为 0）的元素进行加和处理，随后重新从队尾入队。这样就实现了队列的更新。只需要注意，按照题目要求，在每轮外层循环先输出 n-i 个空格即可。

(2) 源代码

Status YangHui(int n){ } **//输出 n 阶杨辉三角**

```
Status YangHui(int n){
  LinkQueue<int> queue;  //创建链队列
  queue.EnQueue(1);      //初始化队列(入队两个 1, 作为杨辉三角第一行)
  queue.EnQueue(1);
  int t1 = 0, t2 = 0;    //创建中间变量 t1 和 t2

  for(int i = 1; i <= n; i++)
  //对从 1 到 n 的每一行进行操作
  {
    t2 = 0;                          //每行开始操作时重置 t2 的值 (t2 为上一个出队元素的值)
    cout << string(n - i, ' ');      //首先输出 n-i 个空格

    //操作 part1: 前 i+1 个元素的出队、相加和重新入队
    for(int j = 0;j < i+1; j++)
    {                                //第 i 行进行 i+1 次出队
      queue.DelQueue(t1);            //出队, 储存到 t1 中
      cout << t1 << ' ';             //输出 t1 (首元素)
      queue.EnQueue(t1+t2);          //t1+t2 入队 (上一次出队元素的值加刚出队元素的值)
      t2 = t1;                       //把 t1 (刚出队元素) 的值存放到 t2 中
    }

    //操作 part2: 最后一个元素 1 的入队
    queue.EnQueue(1);                //加法部分结束, 入队 1
    cout << endl;                    //换行
  }

  return SUCCESS;   //返回成功信息
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。