Homework：栈和队列

3.设从键盘输入一整数的序列:a1,a2,a3.. an，试算法实现:用栈结构存储输入的整数，当ai≠-1时，将ai进栈当a=-1时，输出栈顶整数并出栈。算法应对异常情况(入栈满等)给出相应的信息。

```
//顺序栈
class stack {
#define MAXSIZE 10
public:
    stack()
        :_base(new int[MAXSIZE])
        ,_top(_base)
        ,_stacksize(MAXSIZE)
    {}
    ~stack()
    {
        delete[] _base;
        _base = _top = nullptr;
        _stacksize = 0;
    }
    void store(int* a)
    {
        for (int i = 0; i < _top - _base; i++)
        {
            if (_top == _base + _stacksize)
            {
                cout << "栈已满" << endl;
            }
            if (a[i] == -1)
            {
                if (!empty())
                {
                    pop(a[i]);

                }
                else
                {
                    cout << "栈已空" << endl;
                }
            }
            else
            {
                push(a[i]);
            }
        }
    }

private:
    bool empty()
    {
        if (_base == _top)
        {
            return true;
        }
        return false;
    }
    int pop(int n)
    {
        if (!empty())
        {
            _top--;
        }
        return *_top;
    }
    void push(int n)
    {
        *_top = n;
        _top++;
    }

private:
    int* _base;
    int* _top;
    int _stacksize;
};
```

6.假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点(注意:不设头指针)，试编写相应的置空队列、判断队列是否为空、入队和出队等算法。

```
class list {
    list()
        :_front(new listnode[1])
        ,_rear(_front)
    {
        //带头循环链表的头节点
        _front->_data = 0;
        _front->next = _front;
    }
    bool empty()
    {
        if (_front == _rear)
            return true;
        return false;
    }
    int pop()
    {
        if (!empty())
        {
            listnode* tmp = _front->next;
            _front->next = tmp->next;
            int ret = tmp->_data;
            delete[] tmp;
            tmp = nullptr;
            return ret;
        }
        else
        {
            cout << "队列已空" << endl;
        }
    }
    void push(int n)
    {
        listnode* tmp = new listnode[1];
        tmp->_data = n;
        tmp->next = _front;
        _rear->next = tmp;
    }
    list& set_empty()
    {
        while (_front != _rear)
        {
            pop();
        }
        return *this;
    }
    int get_head()
    {
        if (!empty())
        {
            return _front->next->_data;
        }
    }
    ~list()
    {
        set_empty();
        delete[] _front;
        _front = nullptr;
        _rear = nullptr;
    }
private:
    class listnode{
    public:
        int _data;
        listnode* next;
    };
    listnode* _front;
    listnode* _rear;
};
```

9.已知Ackermann函数定义 (1)写出计算 Ack(m,n)的递归算法，并根据此算法给出Ack(2,1)的计算过程 (2)写出计算Ack(m, n)的非递归算法。

```
//递归
int Ackermann_recurrence(int m, int n)
{
    if (m == 0)
        return n + 1;
    else if (m != 0 && n == 0)
        return Ackermann_recurrence(m - 1, 1);
    else
        return Ackermann_recurrence(m - 1, Ackermann_recurrence(m, n - 1));
}

//非递归
int Ackermann(int m, int n)
{
    stack<int> stack;
    stack.push(m);
    stack.push(n);
    int nn, mm;
    while (stack.size() != 1)
    {
        nn = stack.top();
        stack.pop();
        mm = stack.top();
        stack.pop();
        if (mm == 0)
        {
            nn += 1;
            stack.push(nn);
        }
        else if (mm != 0 && nn == 0)
        {
            stack.push(mm - 1);
            stack.push(1);
        }
        else
        {
            stack.push(mm - 1);
            stack.push(mm);
            stack.push(nn - 1);
        }
    }
    return nn;
}
```