栈和队列

栈和队列仍然属于线性结构，他们的逻辑结构和线性表相同，具有线性结构的共同特征。栈和队列的基本操作是线性表操作的子集，限定插入和删除元素的操作只能在线性表的一端进行。栈按照“先进后出”的规则进行操作，队列按照“先进先出”的规则进行操作。

栈

栈的类型定义

栈是只允许在表的一端进行插入、删除操作的线性表，具有后进先出(LIFO, Last In First Out)，也称为先进后出(FILO, Fist In Last Out)的特点。后进先出表示最晚入栈的元素最先被删除，先进后出表示最先入栈的元素最后被删除。

栈的术语说明如下：

- 栈顶(Top)：允许进行插入和删除操作的一端称为栈顶。
- 栈底(Bottom)：表的另一段被称为栈底。
- 进栈(Push)：在栈位置插入元素的操作叫进栈，也叫入栈、压栈。
- 出栈(Pop)：删除栈顶元素的操作叫出栈，也叫弹栈、退栈。
- 空栈：不含元素的空表称为空栈。
- 栈溢出：当栈满时，若再有元素进栈，则发生上溢；当栈空时，若再出栈，则发生下溢。

栈的抽象数据类型定义如下：

```
template <class T>
class Stack{
public:
    virtual bool empty() const = 0; // 判空
    vietual int size() const = 0; // 求栈中元素的个数
    virtual void push(const T& x) = 0; // 压栈，插入元素x为新的栈顶元素
    virtual T pop() = 0; // 弹栈，若栈非空，则删除栈顶元素
    virtual T getTop() const = 0; // 取栈顶元素，返回栈顶元素但不弹出
    virtual void clear() = 0; // 清空栈
    virtual ~Stack(){}
};
```

自定义异常处理类：

```
class outOfRange:publid exception{
public:
    const char* what()const throw(){
        return "ERROR! OUT OF RANGE.\n";
    }
};
class badSize:public exception{
public:
    const char* what()const throw(){
        return "ERROR! BAD SIZE.\n";
    }
};
```

顺序栈的表示和实现

利用顺序存储结构实现的栈称为顺序栈。类似于顺序表，栈中的元素用一个一维数组来存储，栈底位置可以设置在数组的任意一个端点处，通常设在小下标的一端，栈顶是随着插入和删除操作而变化的。为方便操作，用一个整型变量top存放栈顶元素的位置（下标），top称为栈顶指针。初始时，`top = -1`，表示栈为空；元素进栈，top加1，然后将数据写入top所指向的单元中；退栈时，top减1。

顺序栈的类型描述如下：

```
template <class T>
class seqStack:public Stack<T>{
    T* data; // 存放栈中元素的数组
    int top; // 栈顶指针，指向栈顶元素
    int maxSize; // 栈的大小
    void resize(); // 扩大栈空间
public:
    seqStack(int initSize = 100);
    ~seqStack(){delete [] data;}
    void clear() {top = -1;} // 清空栈
    bool empty() const {return top == -1;} // 判空
    int size() const {return top + 1;} // 求长度
    void push(const T& value); // 进栈
    T pop(); // 出栈
    T getTop() const; // 取栈顶元素
}
```

将数组的小下标端设为栈底，栈空时，栈顶指针`top = -1`；入栈时，栈顶指针加1，即`++top`；出栈时，栈顶指针减1，即`--top`。

下面介绍栈的基本操作在顺序栈中的实现。

构造函数

初始化一个空的顺序栈，置栈顶指针top为-1.

```
template <class T>
seqStack<T>::seqStack(int initSize = 100){
    if (initSize <= 0) throw badSize();
    data = new T[initSize];
    maxSize = initSize;
    top = -1;
}
```

进栈

```
template <class T>
void seqStack<T>::push(const T& value){
    if (top == maxSize - 1) resize(); // 检查顺序栈是否已满
    data[++top] = value; // 修改栈顶指针，新元素入栈
}
```

出栈

```
template <class T>
T seqStack<T>::pop(){
    if(empty()) throw outOfRange(); // 空栈无法弹栈
    return data[top--]; // 修改栈顶指针，返回栈顶元素
}
```

实际上“出栈”并不一定要带回元素，带回元素是“副产品”，出栈的主要目的是下移指针。

取栈顶元素

```
template <class T>
T seqStack<T>::getTop() const{
    if(empty()) throw outOfRange();
    return data[top];
}
```

说明：若允许修改返回的栈顶元素，则考虑使用引用作为函数的返回值。

扩大栈空间

```
template <class T>
void seqStack<T>::resize(){
    T *tmp = data;
    data = new T[2 * maxSize];
    for (int i = 0; i < maxSize; ++i)
        data[i] = tmp[i];
    maxSize *= 2;
    delete [] tmp;
}
```

注意，对于顺序栈，入栈时应先判断栈是否已满，栈满的条件为`top == maxSize - 1`。若栈是满的，则需要重新申请空间，否则将出现上溢出错误。执行出栈和读栈顶元素操作时，应先判断栈是否为空，在栈为空时不能操作，否则会产生下溢出错误。栈空常常作为一种控制转移的条件。

栈的基本操作中，除了进栈和扩大栈空间以外，所有的操作的时间复杂度都是$O(1)$。当栈满时，这是最坏情况，需要调用`resize()`扩大栈空间才能进栈，时间复杂度是$O(n)$；当栈不满时，进栈的时间复杂度为$O(1)$。

实际应用中，往往在一个程序中会出现使用多个栈的情况。这需要给每个栈都分配一个大小合适的空间。这很难做到，因为难以估计各个栈的最大实际空间。而且，各个栈的最大实际空间是变化的，有的栈已经满了，有的栈还是空的。因此，如果多个栈能够共享内存空间将是很理想的，这将减少发生上溢出的可能性。在这种模式下，每个栈都设栈顶和栈底指针。当一个栈满后，可以向左、右栈借用空间，只有当所有的栈都满了，才会发生溢出。但是多栈共享空间的算法比较复杂，特别是当临近溢出的时候，为了找到一个可用单元会需要移动很多元素。

假设程序中需要两个栈，我们可以让两个栈共享一个数组空间。利用“栈底位置不变”的性质，将两个栈的栈底分别设置在数组的两端，入栈时两个栈相向延伸。只有当两个栈的栈顶相遇时才发生溢出。

描述两个栈共享向量的结构定义如下：

```
template <class T>
class dSeqStack:public Stack<T>{
    int maxSize; // 栈中最多可存放的元素个数
    T* data; // 存放栈中元素的数组
     int top[2]; // 两个栈顶指针
    void resize(); // 扩大栈空间
public:
    dSeqStack(int size); // 构造函数
    ~dSeqStack(); // 析构函数
    void clear(int num); // 清空栈，参数num用于区分操作哪个栈
    bool empty(int num); // 判空，参数num用于区分操作哪个栈
    int size(int num); // 求栈中元素个数，参数num用于区分操作哪个栈
    void push(const T& value, int num); // 压栈，参数num用于区分操作哪个栈
    T pop(T* item, int num); // 弹栈，参数num用于区分操作哪个栈
    T getTop(int num) const; // 取栈顶元素，参数num用于区分操作哪个栈
}
```

其中，`top[0]`是第一个栈的栈顶指针，`top[1]`是第二个栈的栈顶指针。初始化时，`top[0] = -1`，`top[1] = maxSize`。当`top[1]-top[0] = 1`时，栈满。

链栈的表示和实现

用链式存储结构实现的栈称为链栈。链栈中结点的结构和单链表中结点的结构相同。链栈由栈顶指针top唯一确定。因为栈的主要操作是在栈顶进行插入和删除操作，所以链栈通常不带头结点，top指针直接指向栈顶元素。当`top == nullptr`时为空栈。

链栈的类型定义以及基本操作如下：

```
template <class T>
class linkStack:public Stack<T>{
    struct Node{
        T data;
        Node* next;
        Node(){next = nullptr;}
        Node(const T& value, Node* p = nullptr){data = value; next = p;}
    };
    Node* top; // 栈顶指针
public:
    linkStack(){top = nullptr;} // 构造空栈
    ~linkStack(){clear();}
    void clear(); // 清空
    bool empty() const {return top == nullptr;} // 判空
```

```
    int size() const; // 求长度
    void push(const T& value); // 压栈
    T pop(); // 弹栈
    T getTop() const; // 取栈顶元素
}
```

清空栈

```
template <class T>
void linkStack<T>::clear(){
    Node* p;
    while (top != nullptr){
        p = top; // p指向当前栈顶元素
        top = top->next; // top指针移向次栈顶元素
        delete p; // 释放p指向的当前栈顶元素
    }
}
```

求栈中元素个数

```
template <class T>
int linkStack<T>::size() const{
    Node* p = top;
    int count = 0;
    while (p){
        count++;
        p = p->next;
    }
    return count;
}
```

进栈

```
template <class T>
void linkStack<T>::push(const T& value){
    Node* p = new Node(value, top); // 在栈顶插入元素
    top = p; // p称为新的栈顶元素
}
```

出栈

```
template <class T>
T linkStack<T>::pop(){
    if(empty()) throw outOfRange();
    Node* p = top;
    T value = p->data; // value保存栈顶元素的值
    top = top->next; // top指针向后移动
    delete p; // 删除栈顶元素
    return value;
}
```

取栈顶元素

```
template <class T>
T linkStack<T>::getTop() const{
    if(empty()) throw outOfRange();
    return top->data;
}
```

链表的入栈、出栈、取栈顶元素、判空等等操作都是在栈顶进行的，与栈中元素的个数无关，这些操作的时间复杂度均为$O(1)$。链栈的入栈操作不需要判定栈是否已满，但是出栈操作仍然需要判定栈是否为空。

链栈的清空和求长度操作需要遍历链栈中所有元素，因此时间复杂度为$O(n)$。若在链栈类型定义中增加整型变量cueLength，用于记录当前栈中元素个数，这样求长度操作的时间复杂度就可以降为$O(1)$。

栈的应用举例

十进制数转化为其他进制数

在计算中，经常碰到十进制数N和其他d进制数的转换问题，实现转换的方法的方法很多，其中一种方法基于以下公式：

$$N = (N/d) * d + N\%d$$

其中/表示整除，而%表示取模。

下面以十进制数N转换为十六进制数为例说明转换过程：当$N \neq 0$时，首先计算$N\%16$得到的余数为十六进制数的最低位；然后给N赋值$N/16$，再次计算$N\%16$的余数；持续这个过程，直到$N = 0$，停止转换；最后得到的余数是十六进制数的最高位。这样我们很容易想到运用栈来存储计算过程中产生的余数。

十进制数$(1000)_{10}$转换为十六进制数是$(3E8)_{16}$，其运算过程如下：

N	$N/16$	$N\%16$
1000	62	8
62	3	14(即十六进制数E)
3	0	3

若用一个栈存放每次除法的余数，则只要$N \neq 0$，就将$N\%16$的结果进栈，然后修改$N = N/16$。

```
void convert(){
    linkStack<int> S; // 初始化栈S
    int N, e;
    cin >> N; // 读入十进制数
    while (0 != N){
        S.push(N%16); // 将得到的十六进制数压入栈
        N = N/16; // 数N除以16作为新的被除数
    }
    while(!S.empty()){
        e = S.pop();
        if (e > 9) cout << char(e-10+'A'); // 10～15的十六进制数为A～F
        else cout << e;
    }
    cout << endl;
}
```

表达式中括号的匹配检查

假定表达式中的括号有一下三对：‘(’和‘)’，‘[’和‘]’，‘{’和‘}’，我们可以使用栈来检查表达式中括号是否正确匹配。规则如下：如果为左括号，则入栈。如果为右括号，若栈顶是其对应的左括号，则退栈；若栈顶不是其对应的左括号，则结论为括号不匹配。当表达式结束后，若栈为空，则表明表达式中括号配对；否则表明表达式中括号不配对。

```
bool match(){
    linkStack<char> T;
    char item;
    int i = 0;
    string expression;
    getlinke(cin, expression); // 输入字符串
    while (i < expression.size()){ // 扫描字符串
        switch (expression[i]){
            case '(':T.push(expression[i]); break;
            case '[':T.push(expression[i]); break;
            case '{':T.push(expression[i]); break;
            case ')':
                if (T.empty() || (item = T.getTop()) != '('){
                    cout << "mismatched" << endl;
                    return false;
                }
                else T.pop();
                break;
            case ']':
                if (T.empty() || (item = T.getTop()) != '['){
                    cout << "mismatched" << endl;
                    return false;
                }
                else T.pop();
                break;
            case '}':
                if (T.empty() || (item = T.getTop()) != '{'){
                    cout << "mismatched" << endl;
                    return false;
                }
                else T.pop();
                break;
        };
        i++;
    }
    if(T.empty()) {cout << "matched\n" << endl; return true;}
    else{cout << "mismatched" << endl; return false;} // 栈非空，不配对
}
```

利用栈消除递归

栈的一个重要应用是在程序设计语言中实现递归。所谓递归是指：在一个定义内部，直接（或间接）出现定义本身的应用，则称它们是递归的，或者递归定义的。递归定义由以下两部分组成：

1. 递归规则，将规模较大的原问题分解成一个或多个规模更小，且具有与原问题类似特性的子问题，求解原问题的方法同样可用来求解这些子问题。
2. 递归出口，无须分解可直接求解的最小子问题，也称为递归的终止条件。

递归的本质是函数调用，在系统内部，函数调用是用栈来实现的。（该死的运行时栈，不做赘述）

以下三种情况，常常使用递归的方法求解问题

1. 定义是递归的：在数学上有许多问题都是递归定义的，例如阶乘、斐波那契数列
2. 数据结构是递归的：某些数据结构本身就是递归的，它们的操作可递归地实现。例如链表结点Node由数据域data和指针域next组成，而next为指向Node的指针
3. 问题的解法是递归的：典型例子是汉诺塔问题

队列

队列的类型定义

队列是一种只允许在表的一端插入，在另一段删除的，操作受限的线性表。像排队一样，入队时排在队尾，到达越早的结点离开得越早。所以队列的特点是先进先出(FIFO, Fist In First Out)。

允许插入的一端称为队尾(记为rear)，允许删除的一端称为队头(记为front)。当队列中没有元素时称为空队列。

队列的抽象数据类型定义如下：

```
template <class T>
class Queue{
public:
    virtual bool empty() const = 0; // 判空
    virtual void clear() = 0; // 清空队列
    virtual int size() const = 0; // 求队列长度
    virtual void enQueue(const T& x) = 0; // 入队
    virtual T deQueue() = 0; // 出队
    virtual T getHead() const = 0; // 读队头元素
    virtual ~Queue(){} // 虚析构函数
};
```

循环队列——队列的顺序表现和实现

在队列的顺序存储结构中除了一个能容纳数据元素的数组空间以外，还需要两个指针分别指向队列的前端和尾端。我们约定，队头指针指向队头元素的前一个位置，队尾指针指向队尾指针。顺序队列的类型定义如下：

```
template <class T>
class seqQueue:publid Queue<T>{
    T* data;
    int maxSize;
    int front, rear;
    void resize();
public:
    seqQueue(int initSize = 100);
    ~seqQueue(){delete [] data;}
    void clear(){front = rear = -1;} // 清空队列
    bool empty() const {return front == rear;} // 判空
    bool full() const {return (rear + 1) % maxSize == front;} // 判满
    int size() const {return (rear - front + maxSize) % maxSize;} // 队列长度
    void enQueue(const T& x);
```

```
    T deQueue();
    T getHead() const;
};
```

下面分析顺序队列基本操作的实现：

1. 根据约定，front指向队头元素的前一位置，rear指向队尾元素，则清空队列的操作为`front = rear = -1;`。
2. 在不考虑溢出的情况下，入队时队尾指针`rear++`，指向新的队尾位置后，新元素`item`入队，操作为`rear++; data[rear] = item;`。
3. 在不考虑队空的情况下，出队时头指针`front++`，表示队头元素出队，操作为`front++; item = data[front];`。
4. 队中元素个数为`m = rear - front`。当队满时，`m == maxSize`，当队空时，`m == 0`。

按照上述思想来看，随着出队、入队的进行，会使得整个队列整体向后移动，这样就会出现：队尾指针已经移到了最后`rear = maxSize - 1`，若再有元素入队就会溢出。但是事实上，此时并未真的队满，队伍的前端还有许多空着位置可用，这种现象称为“假溢出”。出现这种现象的原因是：队列限制为“队头出队，队尾入队”。那么，怎么解决“假溢出”溢出呢?

将队列的数据区`data[0,maxSize-1]`视为头尾相连的循环结构，从逻辑上认为单元0就是单元`maxSize - 1`。我们称这种队列为循环队列。但是这样又出现了一个问题：在“队满”和“队空”的情况下，`rear == front`这个条件都成立，此时如何区分“队满”和“队空”成为了必须要解决的问题。常用的解决方法如下：

1. 设置一个标志位以区分队列是空还是非空。
2. 设置计数器`count`统计堂前队列中元素个数，若`count == 0`则队列空，若`count == maxSize`则队列满。
3. 牺牲一个存储空间用于区分队列空和满，规定`front`指向的单元不能存储队列元素，起到标识作用，此时队满的状态是：`(rear + 1) % maxSize == front`，即：将队尾指针加1后除以maxSize取模等于队头指针作为队列满的标志。

下面给出循环链表基本操作的实现：

初始化一个空链表

```
template <class T>
seqQueue<T>::seqQueue(int initSize){
    if(initSize <= 0) throw badSize();
    data = new T[initSize];
    maxSize = initSize;
    front = rear = - 1;
}
```

入队

```
template <class T>
void seqQueue<T>::enQueue(const T& x){
    if ((rear + 1) % maxSize == front) resize(); // 若队列满，则扩大空间
    rear = (rear + 1) % maxSize; // 移动队尾指针
    data[rear] = x; // x入队
}
```

出队

```
template <class T>
T seqQueue<T>::deQueue(){
    if(empty()) throw outOfRange(); // 若队列空，抛出异常
    front = (front + 1) % maxSize; // 移动队首指针
    return data[front]; // 返回队首元素
}
```

取队首元素

```
template <class T>
T seqQueue<T>::getHead() const{
    if(empty()) throw outOfRange();
    return data[(front + 1) % maxSize]; // 返回队首元素，不移动队首指针
}
```

扩大队列空间

```
template <class T>
void seqQueue<T>::resize(){
    T* p = data;
    data = new T[2 * maxSize];
    for (int i = 1; i < size(); ++i)
        data[i] = p[(front + i) % maxSize]; // 复制元素
    front = 0; rear = size(); // 设置队首和队尾指针
    maxSize *= 2;
    delete p;
}
```

链队列——队列的链式表示和实现

用链式结构存储结构表示的队列简称为链队列。用无头结点的单链表表示队列，表头为队头，表尾为队尾。一个链队列需要两个指针front和rear分别指示队头元素和队尾元素（分别称为头指针和尾指针），这样既方便在队首删除元素，又方便在队尾插入元素。链队列不会出现队列满的情况，但队列为空的情况依然存在。当队列为空时，单链表中没有结点，即头尾指针都为空指针。

下面给出链队列的类型定义：

```
template <class T>
class linkQueue:public Queue<T>{
    struct node{
        T data;
        node* next;
        node(const T& x, node& N = nullptr){data = x; next = N;}
        node():next(nullptr){}
        ~node(){};
    };
    node *front, *rear; // 队头指针，队尾指针
public:
    linkQueue(){front = rear = nullptr;}
    ~linkQueue(){clear();}
    void clear(); // 清空队列
    bool empty() const {return front == nullptr;} // 判空
    int size() const; // 队列长度
```

```
    void enQueue(const T& x); // 入队
    T deQueue(); // 出队
    T getHead() const; // 取队首元素
};
```

链队列的基本操作实现如下：

清空队列

```
template <class T>
void linkQueue<T>::clear(){
    node* p;
    while (front != nullptr){ // 释放队列中所有结点
        p = front;
        front = front->next;
        delete p;
    }
    rear = nullptr; // 修改尾指针
}
```

求队列长度

```
template <class T>
int linkQueue<T>::size() const{
    node* p = front;
    int count = 0;
    while (p){
        count++;
        p = p->next;
    }
    return count;
}
```

入队

```
template <class T>
void linkQueue<T>::enQueue(const T& x){
    if (rear == nullptr) // 原队列为空
        front = rear = new node(x); // 入队元素即是队首又是队尾
    else{
        rear->next = new node(x); // 在队尾入队
        rear = rear->next; // 修改队尾指针
    }
}
```

出队

```
template <class T>
T linkQueue<T>::deQueue(){
    if(empty()) throw outOfRange(); // 队列空，抛出异常
    node* p = front;
    T value = front->data; // 保存队首元素
    front = front->next; // 在队首出队
    if (front == nullptr) // 如果原来只有一个元素，出队后队列为空
        rear = nullptr; // 修改队尾指针
    delete p;
    return value;
}
```

取队首元素

```
template <class T>
T linkQueue<T>::getHead() const{
    if(empty()) throw outOfRange(); // 队列空，抛出异常
    return front->data; // 返回队首元素
}
```

采用带有头尾指针的单链表存储队列时，队列基本操作的实现非常简单，入队、出队和取队首元素的时间复杂度都是$O(1)$。清空队列和求队列长度都需要遍历队列，时间复杂度为$O(n)$，也可以在链队列类型定义中增加整型变量用于记录当前队列的长度。