第二章 栈、队列、堆

文章目录

栈、队列、堆

基础知识

（1）Stack（栈）

方法	功能
Stack stack = new Stack()	创建栈
empty()	测试堆栈是否为空
peek()	查看堆栈顶部的对象，但不从堆栈中移除它
pop()	移除堆栈顶部的对象，并作为此函数的值返回该对象
push(item)	把项压入堆栈顶部

（2）Queue（队列）

方法	功能
Queue queue=new LinkedList()	创建队列
peek()	获取但不移除此队列的头；如果此队列为空，则返回 null
poll()	获取并移除此队列的头，如果此队列为空，则返回 null
offer(item)	将指定的元素插入此队列（如果立即可行且不会违反容量限制），当使用有容量限制的队列时，无法插入元素

（3）Heap（堆）

优先队列PriorityQueue——二叉堆，最小（大）值先出

1. 优先级队列（priority queue）中的元素可以按照任意的顺序输入，却总是按照排序的顺序进行检索。也就是说，无论何时调用 poll 方法，总会获得当前优先级队列中最小的元素。
2. 优先级队列可实现堆 (heap)数据结构 。执行 offer 和 poll 操作，可以让最小的元素移动到队首，而不必花费时间对元素进行排序。
3. 一个优先级队列既可以保存实现了 Comparable 接口的类对象，也可以保存在构造器中提供的 Comparator 对象。

```
Queue<int> priorityQueue = new PriorityQueue<int>();//默认的队列 int按照从大至小
// 根据Comparator方法，按照人口大小排序
Comparator<Person> cmp = new Comparator<Person>() {
            public int compare(Person arg0, Person arg1) {
                // TODO Auto-generated method stub
                int num_a = arg0.getPopulation();
                int num_b = arg1.getPopulation();
                if(num_a < num_b) return 1;
                else if(num_a > num_b) return -1;
                else return 0;

            }
        };
        Queue<Person> priorityQueue = new PriorityQueue<Person>(11,cmp);
```

（4）Deque（双端队列）

官方推荐Deques实现堆栈，Deque的12种方法总结如下：

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

- Deque接口用作队列时，将得到 FIFO（先进先出）行为。将元素添加到双端队列的末尾，从双端队列的开头移除元素。从Queue 接口继承的方法完全等效于 Deque 方法，如下表所示：

	Deque	Queue
获取队头元素	peekFirst()	peek()
移除队头	pollFirst()	poll()
插入队尾	offerLast(item)	offer(item)

- Deque也可以被用作LIFO的栈，此时的接口应该严格参照Stack类的实现。当deque被用作栈时，元素在deque的head端push/pop。栈的方法等价于Deque中的一些方法，如下表：

	Deque	Stack
获取队头元素	peekFirst()	peek()
移除栈首	pollFirst()	pop()
插入栈首	offerFirst(item)	push(item)

leetcode

例1 使用队列实现栈（225）

题目描述
使用队列实现栈的下列操作：
push(x) – 元素 x 入栈
pop() – 移除栈顶元素
top() – 获取栈顶元素
empty() – 返回栈是否为空
算法思路
使用队列实现栈
在STACK push元素时，利用临时队列调换元素次序
方法：

1. 将新元素 push 进入临时队列 temp_queue
2. 将原队列内容 push 进入临时队列 temp_queue
3. 将临时队列元素 push 进入数据队列 data_queue
4. 得到数据队列结果

程序代码

```
public static class MyStack {
    // 使用队列实现栈
    // 在STACK push元素时，利用临时队列调换元素次序
    // 方法：
    // 1. 将新元素 push 进入临时队列 temp_queue
    // 2. 将原队列内容 push 进入临时队列 temp_queue
    // 3. 将临时队列元素 push 进入数据队列 data_queue
    // 4. 得到数据队列结果
    Queue<Integer> data_queue;  // 数据队列
    Queue<Integer> temp_queue;  // 临时队列
    /** Initialize your data structure here. */
    public MyStack() {
        data_queue = new LinkedList<Integer>();
        temp_queue = new LinkedList<Integer>();
    }

    public void push(int x) {
        temp_queue.offer(x);
        while(!data_queue.isEmpty()) {temp_queue.offer(data_queue.poll());}
        while(!temp_queue.isEmpty())data_queue.offer(temp_queue.poll());
    }

    public int pop() {
        return data_queue.poll();
    }

    public int top() {
            return data_queue.peek();
```

```
        }

        /** Returns whether the stack is empty. */
        public boolean empty() {
            return data_queue.isEmpty();
        }

        public void print() {
            for(Iterator<Integer> iter=data_queue.iterator();iter.hasNext();)
            {
                Integer temp = iter.next();
                System.out.println(temp+" ");
            }
            System.out.print("\n");
        }
    }
```

例2 使用栈实现队列（232）

题目描述
使用栈实现队列的下列操作：

push(x) – 将一个元素放入队列的尾部。
pop() – 从队列首部移除元素。
peek() – 返回队列首部的元素。
empty() – 返回队列是否为空。
算法思路
用栈实现队列
在队列 push 元素时，利用 临时栈 调换元素次序

1. 将 原数据栈 data_stack 内容 push 进入 临时栈 temp_stack
2. 将新数据 push 进入临时栈 temp_stack
3. 将临时栈 temp_stack 中的元素 push 进入数据栈 data_stack
4. 得到数据栈 data_stack

程序代码

```
    public static class MyQueue {
        // 用栈实现队列
        // 在队列 push 元素时，利用 临时栈 调换元素次序
        // 1. 将 原数据栈 data_stack 内容 push 进入 临时栈 temp_stack
        // 2. 将新数据 push 进入临时栈 temp_stack
        // 3. 将临时栈 temp_stack 中的元素 push 进入数据栈 data_stack
        // 4. 得到数据栈 data_stack
        Stack<Integer> data_stack;  // 数据栈
        Stack<Integer> temp_stack;  // 临时栈

        public MyQueue() {
                data_stack = new Stack<Integer>();
                temp_stack = new Stack<Integer>();
        }

        public void push(int x) {
            while(!data_stack.isEmpty()) temp_stack.push(data_stack.pop());
            temp_stack.push(x);
            while(!temp_stack.isEmpty())data_stack.push(temp_stack.pop());
        }

        public int pop() {
            return data_stack.pop();
        }

        public int peek() {
                return data_stack.peek();
        }

        public boolean empty() {
            return data_stack.isEmpty();
        }
    }
```

例3 包含min函数的栈（155）

题目描述
设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。

push(x) – 将元素 x 推入栈中。
pop() – 删除栈顶的元素。
top() – 获取栈顶元素。
getMin() – 检索栈中的最小元素。
算法思路
定义一个最小值栈，存储各个状态下的最小值
程序代码

```
    class MinStack {
        // 155.最小栈
        // 设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。
        // 定义一个最小值栈，存储各个状态下的最小值
        Stack<Integer> data_stack;
        Stack<Integer> min_stack;
        public MinStack() {
            data_stack = new Stack<Integer>();
            min_stack = new Stack<Integer>();   // 最小值栈，存储各个状态下最小值
        }

        public void push(int x) {
```

```
            data_stack.push(x);
            if(min_stack.isEmpty()) {min_stack.push(x);}
            else if( x < min_stack.peek())
                    min_stack.push(x);
            else min_stack.push(min_stack.peek());
        }

        public void pop() {
            data_stack.pop();
            min_stack.pop();
        }

        public int top() {
            return data_stack.peek();
        }

        public int getMin() {
            return min_stack.peek();
        }
    }
```

例4 合法的出栈队列（946）

题目描述

给定 pushed 和 popped 两个序列，只有当它们可能是在最初空栈上进行的推入 push 和弹出 pop 操作序列的结果时，返回 true；否则，返回 false 。

算法思路

采用队列 & 栈 模拟

1. 出栈结果存储在队列order中
2. 按元素顺序，将元素push进入栈
3. 每push一个元素，即检查是否与队列首部元素相同，若相同则弹出队首元素，弹出栈顶元素，直到两个元素不同结束
4. 若最终栈为空，说明序列合法，否则不合法

程序代码

```
    public boolean validateStackSequences(int[] pushed, int[] popped) {
            // 采用队列 & 栈 模拟
        // 1. 出栈结果存储在队列order中
            // 2. 按元素顺序，将元素push进入栈
            // 3. 每push一个元素，即检查是否与队列首部元素相同，若相同则弹出队首元素，弹出栈顶元素，直到两个元素不同结束
            // 4. 若最终栈为空，说明序列合法，否则不合法
            if (pushed.length != popped.length)
             return false;

            Stack<Integer> stack = new Stack<Integer>();    // stack为模拟栈
            Queue<Integer> queue = new LinkedList<Integer>();       // queue为存储结果队列
            // 将出栈结果存入队列
            for(int i=0;i<popped.length;i++)queue.offer(popped[i]);
            for(int i=0;i<pushed.length;i++) {
                stack.push(pushed[i]);  // 将入栈队列顺序入栈
                while(!stack.isEmpty() && stack.peek() == queue.peek()) {
                    stack.pop();
                    queue.poll();
                }
            }
            if(!stack.isEmpty())return false;
            return true;
    }
```

例5 简单的计算器（224）

题目描述

实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式可以包含左括号（，右括号），加号 + ，减号 -，非负整数和空格 。

算法思路

程序代码

```
    // 224. 基本计算器
    // 实现一个基本的计算器来计算一个简单的字符串表达式的值。
    // 字符串表达式可以包含左括号 ( ，右括号 )，加号 + ，减号 -，非负整数和空格  。
    public int calculate(String s) {
        final int STATE_BEGIN = 0;
        final int NUMBER_STATE = 1;
        final int OPERATION_STATE = 2;

        Stack<Integer> number_stack = new Stack<Integer>();
        Stack<Character> operation_stack = new Stack<Character>();
        int number = 0;
        int STATE = STATE_BEGIN;
        int compuate_flag = 0;
        for(int i=0;i<s.length();i++) {
            char c = s.charAt(i);
            if(c == ' ')continue;

            switch(STATE) {
            case STATE_BEGIN:
                if(isNumber(c))STATE = NUMBER_STATE;
                else STATE = OPERATION_STATE;
                i--;
                break;
            case NUMBER_STATE:
                if(isNumber(c))number = number*10 + c - '0';
                else {
                    number_stack.push(number);
                    if(compuate_flag == 1)compute(number_stack,operation_stack);
                    number = 0;
                    i--;
```

```
                    STATE = OPERATION_STATE;
                }
                break;
            case OPERATION_STATE:
                if(c == '+' || c == '-') {
                    operation_stack.push(c);
                    compuate_flag = 1;
                }else if (c == '(') {
                    STATE = NUMBER_STATE;
                    compuate_flag = 0;
                } else if (isNumber(c)) {
                    STATE = NUMBER_STATE;
                    i--;
                } else if(c == ')') compute(number_stack,operation_stack);
                break;
            }
        }
        if(number != 0) {
                number_stack.push(number);
                compute(number_stack, operation_stack);
        }
        if(number == 0 && number_stack.isEmpty())return 0;

        return number_stack.peek();
    }

    public boolean isNumber(Character c) {
            if(c >= '0' && c<= '9')return true;
            else return false;
    }

    public void compute(Stack<Integer> number_stack, Stack<Character> operation_stack) {
            if(number_stack.size()<2)return ;
            int num2 = number_stack.pop();  // 操作数2
            int num1 = number_stack.pop();  // 操作数1

            // 处理操作符
            if(operation_stack.peek() == '+')number_stack.push(num1 + num2);
            if(operation_stack.peek() == '-')number_stack.push(num1 - num2);

            operation_stack.pop();
    }
```

例6 数组中第K大的数（215）

题目描述

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

算法思路

维护一个K大小的最小堆，堆中元素小于K时，新元素直接入堆；否则，当堆顶小于新元素时，弹出堆顶，将新元素加入堆。

由于堆是最小堆，堆顶是堆中最小元素，新元素都会保证比堆顶小（否则新元素替换堆顶），故堆中第K个元素是已扫描的元素里最大的K个；堆顶即为第K大的数。

程序代码

```
    // 215. 数组中的第K个最大元素
    // 在未排序的数组中找到第 k 个最大的元素。
    // 请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
    public int findKthLargest(int[] nums, int k) {
        // 维护一个K大小的最小堆，堆中元素小于K时，新元素直接入堆；
            // 否则，当堆顶小于新元素时，弹出堆顶，将新元素加入堆
            // 由于堆是最小堆，堆顶是堆中最小元素，新元素都会保证比堆顶小（否则新元素替换堆顶）
            // 故堆中第K个元素是已扫描的元素里最大的K个；堆顶即为第K大的数
            PriorityQueue<Integer> queue = new PriorityQueue<Integer>();// 默认最小堆
            for(int i=0;i<nums.length;i++) {
                if(queue.size()<k)queue.offer(nums[i]);
                else {
                    // 若当前元素大于堆顶元素，则替换堆顶元素
                    if(nums[i] > queue.peek()){
                        queue.poll();
                        queue.offer(nums[i]);
                    }
                }
            }
            return queue.peek();
    }
```

例7 寻找中位数（295）

题目描述

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 (2 + 3) / 2 = 2.5

设计一个支持以下两种操作的数据结构：

void addNum(int num) - 从数据流中添加一个整数到数据结构中。

double findMedian() - 返回目前所有元素的中位数。

算法思路

算法思路——巧用堆的性质

动态维护一个最大堆与一个最小堆，最大堆存储一半较小数据，最小堆存储一半较大数据

维持最大堆的堆顶比最小堆堆顶小

- 情况1：最大堆与最小堆元素相同时
  添加元素：若新元素>最大堆堆顶，则将新元素插入最小堆；否则插入最大堆
  取中位值：(最大堆堆顶 + 最小堆堆顶) /2
- 情况2：最大堆元素 = 最小堆元素 + 1
  添加元素：新元素>最大堆堆顶，新元素插入最小堆；否则最大堆堆顶插入最小堆，新元素插入最大堆
  取中位数：最大堆堆顶

- 情况3：最大堆元素 = 最大堆元素 - 1
  添加元素：新元素<最小堆堆顶，新元素插入最大堆；否则最小堆堆顶插入最大堆，新元素插入最小堆
  取中位数：最小堆堆顶

程序代码

```
// 295. 数据流的中位数
 // 中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。
 static class MedianFinder {
        // 算法思路——巧用堆的性质
        // 动态维护一个最大堆与一个最小堆，最大堆存储一半较小数据，最小堆存储一半较大数据
        // 维持最大堆的堆顶比最小堆堆顶小
        // 情况1：最大堆与最小堆元素相同时
        // 添加元素：若新元素>最大堆堆顶，则将新元素插入最小堆；否则插入最大堆
        // 取中位值：(最大堆堆顶 + 最小堆堆顶) /2
        // 情况2：最大堆元素 = 最小堆元素 + 1
        // 添加元素：新元素>最大堆堆顶，新元素插入最小堆；否则最大堆堆顶插入最小堆，新元素插入最大堆
        // 取中位数：最大堆堆顶
        // 情况3：最大堆元素 = 最大堆元素 - 1
        // 添加元素：新元素<最小堆堆顶，新元素插入最大堆；否则最小堆堆顶插入最大堆，新元素插入最小堆
        // 取中位数：最小堆堆顶

        PriorityQueue<Integer> max_heap;    // 最大堆，存储一半较小元素
        PriorityQueue<Integer> min_heap; // 最小堆，存储一半较大元素

    public MedianFinder() {
        min_heap = new PriorityQueue<Integer>();    // 默认实现最小堆
        max_heap = new PriorityQueue<Integer>(new Comparator<Integer>(){    // 重写Comparator实现最大堆
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2-o1;
            }
        });
    }

    public void addNum(int num) {
            if(max_heap.isEmpty()) {max_heap.offer(num);return;}    // 初始元素插入最大堆堆顶
        if(max_heap.size() == min_heap.size()) {
            // 若新元素>最大堆堆顶，则将新元素插入最小堆；否则插入最大堆
                if(num > max_heap.peek())min_heap.offer(num);
                else max_heap.offer(num);
        }
        else if(max_heap.size() == min_heap.size()+1) {
            // 新元素>最大堆堆顶，新元素插入最小堆；否则最大堆堆顶插入最小堆，新元素插入最大堆
                if(num > max_heap.peek())min_heap.offer(num);
                else {
                    min_heap.offer(max_heap.poll());
                    max_heap.offer(num);
                }
        }else {//max_heap.size() == min_heap.size()-1
            // 新元素<最小堆堆顶，新元素插入最大堆；否则最小堆堆顶插入最大堆，新元素插入最小堆
                if(num < min_heap.peek())max_heap.offer(num);
                else {
                    max_heap.offer(min_heap.poll());
                    min_heap.offer(num);
                }
        }
    }

    public double findMedian() {
        if(max_heap.size() == min_heap.size()) {
        //  (最大堆堆顶 + 最小堆堆顶) /2
                return ((double)min_heap.peek() + (double)max_heap.peek())/2;
        }
        else if(max_heap.size() == min_heap.size()+1) {
            // 最大堆堆顶
                return max_heap.peek();
        }else {//max_heap.size() == min_heap.size()-1
            // 最小堆堆顶
                return min_heap.peek();
        }
    }
}
```

剑指offer

例1：用两个栈实现队列（5）

题目描述
用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。
程序代码

```
// 5. 用两个栈实现队列
Stack<Integer> stack1 = new Stack<Integer>();   // 数据栈
Stack<Integer> stack2 = new Stack<Integer>();// 临时栈

public void push(int node) {
    // 1. 先将 数据栈stack1 的元素压入 临时栈stack2
        while(!stack1.isEmpty())stack2.push(stack1.pop());
        // 2. 将元素压入
        stack2.push(node);
        // 3. 将 临时栈stack2 的元素压入 数据栈stack1
        while(!stack2.isEmpty())stack1.push(stack2.pop());
}

public int pop() {
        return stack1.pop();        //弹出数据栈中元素
}
```

例2：包含min函数的栈（20）

题目描述

定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））。

程序代码

```
public class MinStack {
        // 定义2个栈，一个栈记录数据，另一个栈记录记录最小值
        // 根据栈的现场存储性质，每次元素入栈同时记录此时数据栈的最小值

        Stack<Integer> data_stack = new Stack<Integer>();
        Stack<Integer> min_stack = new Stack<Integer>();

    public void push(int node) {
            // 若入栈元素小于最小栈栈顶，则将该元素加入最小栈。否则将栈顶元素加入最小栈
            if(min_stack.isEmpty()) {
                min_stack.push(node);
            }
            else {
                int current_min = min_stack.peek();
            if(node <= current_min)current_min = node;
            min_stack.push(current_min);
            }

        data_stack.push(node);
    }

    public void pop() {
        data_stack.pop();
        min_stack.pop();
    }

    public int top() {
        return data_stack.peek();
    }

    public int min() {
        return min_stack.peek();
    }
}
```

例3：栈的压入、弹出序列（21）

题目描述

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否可能为该栈的弹出顺序。假设压入栈的所有数字均不相等例如序列1,2,3,4,5是某栈的压入顺序，序列4,5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）

程序代码

```
public boolean IsPopOrder(int [] pushA,int [] popA) {
        // 定义一个栈stack模拟入栈过程
        // 遍历出栈序列。对于出栈序列每个元素 e
        // if(stack.isNotExists(e)) 该元素再栈中存在
        //      {则将压入顺序中到该元素位置的元素依此压入模拟栈（表示该元素之前的元素按压入顺序压入栈）}
        // else 该元素在栈中存在
        //      {如果栈顶元素与该元素不相同 stack.peek != e
        //      说明该弹出顺序与压入顺序不符，返回false}
        // 将该元素弹出
        // 如果最终stack中所有元素均弹出if(stack.isEmpty)return true，说明符合

        Stack<Integer> stack = new Stack<Integer>();
        Queue<Integer> queue = new LinkedList<Integer>();

        for(int i=0;i<pushA.length;i++)queue.offer(pushA[i]);

        for(int i=0;i<popA.length;i++) {
            int e = popA[i];
            if(!stack.contains(e)) {
                // 栈不包括该元素 == 该元素未加入栈
                while(!queue.isEmpty() && queue.peek()!=e)stack.push(queue.poll());
                if(!queue.isEmpty())stack.push(queue.poll());
            }
            // 栈首元素 若不等于当前访问元素，则不符合入栈规律
            if(stack.peek() != e)return false;
            else stack.pop();
        }

        if(stack.isEmpty())return true;
        return false;
    }
```