1、数据结构：栈

栈是仅在仅在表尾进行插入和删除的线性表，也成为先进后出的线性表，把允许插入和删除的一端成为栈顶，另一端为栈底。

栈的应用：

1、递归：递归过程退回的顺序是它执行顺序的逆序，在退回过程中，可能需要执行某些动作，包括恢复在前行过程中保存的数据，这样的结构很适合用栈来实现；

2、四则表达式运算，中缀表达式转换为后缀，函数嵌套使用，已访问页面的历史记录。

实现：java中的Stack类，及LinkedList类，具体可查看源码

2、数据结构：队列

队列是只允许在一端进行插入操作，而在另一端进行删除操作的线性表，也叫做先进先出表。

优先队列：在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出（first in, largest out）的行为特征。通常采用堆数据结构来实现。

实现：用堆（heap）来实现，堆是一种非线性结构，（本篇随笔主要分析堆的数组实现）可以把堆看作一个数组，也可以被看作一个完全二叉树，通俗来讲堆其实就是利用完全二叉树的结构来维护的一维数组，分为大顶堆和小顶堆。

大顶堆：每个结点的值都大于或等于其左右孩子结点的值，Key[i]>=Key[2i+1]&&key>=key[2i+2]

小顶堆：每个结点的值都小于或等于其左右孩子结点的值，Key[i]<=key[2i+1]&&Key[i]<=key[2i+2]

双端队列：可以在两边添加或者删除元素的队列，在java中的实现类为linkedList

3、堆与栈的相关操作

1、用堆实现栈的效果：

```
public class statckToqueue {
private Stack<Integer> s1 = new Stack<>();
private Stack<Integer> s2 = new Stack<>();
```

```
private int front;
//方法一（使用两个栈 入队 - O(n)， 出队 - O(1)）
public void offer(Integer x){
if (s1.empty())
front = x;
while(!s1.isEmpty()){
s2.push(s1.pop());
}
s1.push(x);
while (!s2.isEmpty())
s1.push(s2.pop());
}
public void poll(){
s1.pop();
if (!s1.empty())
front = s1.peek();
}
//方法二（使用两个栈 入队 - O(1)，出队 - 摊还复杂度 O(1)
/*
* 根据栈 LIFO 的特性，s1 中第一个压入的元素在栈底。为了弹出 s1 的栈底元素，我们得把 s1 中所有的元素全部弹出，再把它们压入到另一个栈 s2 中，这个操作会让元素的入栈顺序反转过来。通过这样的方式，s1 中栈底元素就变成了 s2 的栈顶元素，这样就可以直接从 s2 将它弹出了。一旦 s2 变空了，我们只需把 s1 中的元素再一次转移到 s2 就可以了
*/
public void push(int x) {
if (s1.empty())
front = x;
s1.push(x);
}
public void pop() {
if (s2.isEmpty()) {
while (!s1.isEmpty())
s2.push(s1.pop());
}
s2.pop();
}
}
```

2、用栈实现堆：（https://leetcode-cn.com/problems/implement-stack-using-queues/solution/yong-dui-lie-shi-xian-zhan-by-leetcode/）

```
package stackAndQueue;
import java.util.LinkedList;
import java.util.Queue;
//方法一（两个队列，压入 -O(1)O(1)， 弹出 -O(n)O(n)）
public class stackToQueue {
private Queue<Integer> q1=new LinkedList(); //
private Queue<Integer> q2=new LinkedList();
private int top;
//时间复杂度：O(1)
//队列是通过链表来实现的，入队（add）操作的时间复杂度为 O(1)O(1)
//入栈
public void push(Integer value) {
q1.add(value);
top=value;
}
//时间复杂度：O(n)
//算法让 q1 中的 nn 个元素出队，让 n - 1n−1 个元素从 q2 入队，在这里 nn 是栈的大小。这个过程总共产生了 2n - 12n−1 次操作，时间复杂度为 O(n)O(n)。
//出栈
public void pop() {
while(q1.size()>1){ //将q1的除最后一个元素外的其他元素放入q2中，将q1中的最后一个元素弹出即可
top = q1.remove();
q2.add(top);
}
q1.poll();
Queue<Integer> temp = q1;
q1 = q2;
q2 = temp;//
}
/**
* 时间复杂度：O(n)
算法会让 q1 出队 n 个元素，同时入队 n + 1 个元素到 q2。这个过程会产生 2n+1 步操作，同时链表中插入 操作和 移除 操作的时间复杂度为 O(1)，因此时间复杂度为 O(n)。
空间复杂度：O(1)
```

```
* @param value
*/
// 方法二（两个队列， 压入 - O(n)， 弹出 - O(1)），将新加入的元素永远放在空的队列q2中;
public void push2(Integer value) {
q2.add(value);
top=value;
while(!q1.isEmpty()){
q2.add(q1.remove()); //q1的元素都加入q2中，这样新加入的元素就在头部
}
Queue<Integer> temp = q1;
q1 = q2;
q2 = temp;//
}
/**
* 时间复杂度：O(1)O(1)
空间复杂度：O(1)O(1)
*/
public void pop2() {
q1.poll(); //直接让q1头元素出队即可
if (!q1.isEmpty()) {
top = q1.peek(); //获取队列的头部元素
}
}
//方法三（一个队列， 压入 - O(n)， 弹出 - O(1)）
public void push3(Integer value) {
q1.add(value);
int size=q1.size();
while(size>1){
q1.add(q1.remove());
size--;
}
}
public void pop3() {
q1.remove();
// top=q1.peek();
}
public boolean isEmpty() {
```

```
return q1.isEmpty();
}
public int top(){
return q1.peek();
}
}
```

3、返回数据流中的第K大元素；(https://leetcode-cn.com/problems/kth-largest-element-in-a-stream/solution/xiao-ding-dui-by-darrenchan/)

像大小为 k 的堆中添加元素的时间复杂度为 {O}(\log n),们将重复该操作 N 次，故总时间复杂度为 {O}(N \log k)。

```
public class KthLargest {
private int k;
private int[] nums;
private PriorityQueue<Integer> queue=new PriorityQueue(); //优先队列就是小顶堆
public KthLargest(int k, int[] nums) {
super();
this.k = k;
this.nums = nums;
for (int i : nums) {
add(i);
}
}
public Integer add(Integer value){
if (queue.size()<k) {
queue.offer(value);
}else if(queue.peek()<value){
queue.poll(); //弹出栈顶元素
queue.offer(value);
}
return queue.peek();
}
}
```

4、有效的括号（https://leetcode-cn.com/problems/valid-parentheses/

时间复杂度：O(n)O(n)，因为我们一次只遍历给定的字符串中的一个字符并在栈上进行 O(1)O(1) 的推入和弹出操作。
空间复杂度：O(n)O(n)，当我们将所有的开括号都推到栈上时以及在最糟糕的情况下，我们最终要把所有括号推到栈上。例如 (((((((((。

```
public class ValidCharacter {
public boolean isValid(String s) {
if(s==null) return false;
if(s.isEmpty()) return true;
Stack<Character> stack=new Stack<>();
Map map=new HashMap();
map.put(')', '(');
map.put('}', '{');
map.put(']', '[');
if(map.containsKey(s.charAt(0))) return false;//
for (Character character : s.toCharArray()) {
// If the current character is a closing bracket
if(map.containsKey(character))
if(stack.isEmpty()) return false; //If the first character is a closing bracket
if(!stack.pop().equals(map.get(character))) {
return false;
}
}else {
stack.push(character);
}
}
return stack.isEmpty();
}
}
```
