Q`算法

1.旋转数组a[4,5,1,2,3]

求最小值： 取a[midddle]与a[right]比较,如果a[middle]值比较大，那么最小元素在a[middle]到a[right]之间，如果a[right]较大，说明最小元素在a[left]到a[middle]之间，当right-left=1时说明找到了最小的元素;

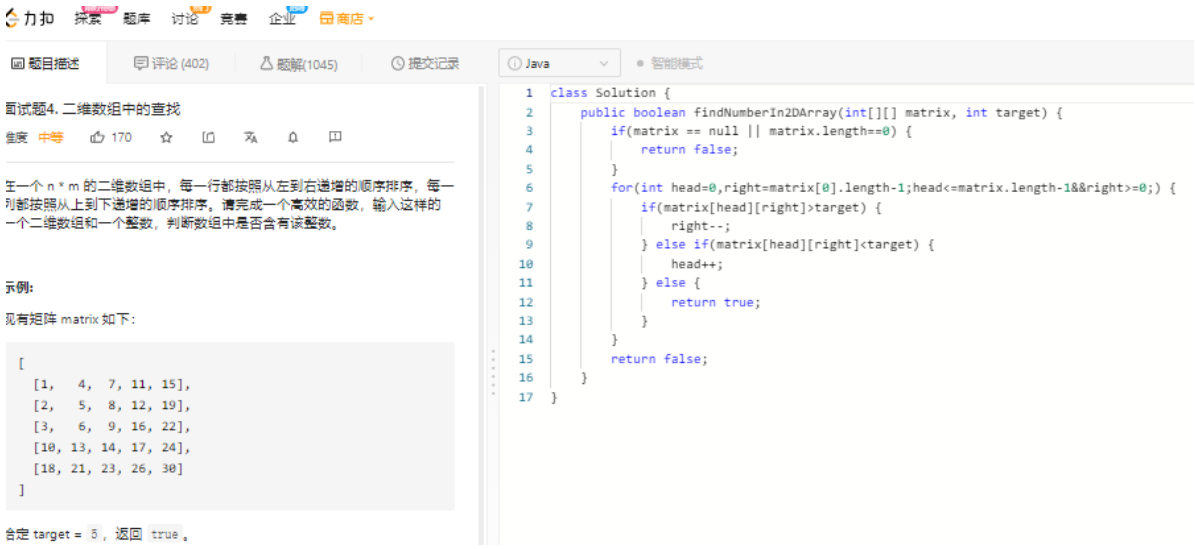
```java
class Solution {
    public int minArray(int[] numbers) {
        return search(numbers,0,numbers.length-1);
    }

    public int search(int[] numbers,int start,int end) {
        if((end-start)<=1) {
            return numbers[start]>numbers[end]?numbers[end]:numbers[start];
        }
        int middle = (start+end)/2;
        if(numbers[end]>numbers[middle]) {
            end = middle;
        } else if(numbers[end]<numbers[middle]){
            start = middle+1;
        } else{
            end--;
        }
        return search(numbers,start,end);
    }
}
```

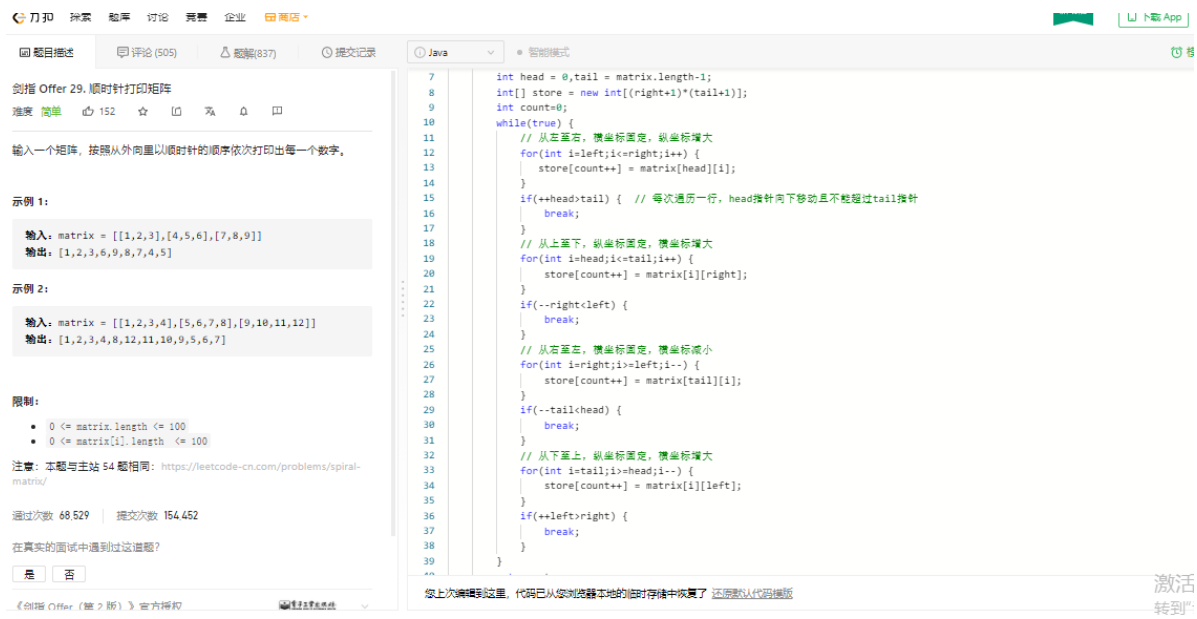
求某个值是否在数组中: 关键要理解，a[left]-a[middle]和a[middle]-a[right]这两个子数组一定有一个是有序的，先找到这个有序的数组，判断target是否在这个有序的数组，如果在有序的数组则可以对这个数组使用二分查找，否则在另一个数组递归查找。

```java
    public int start_z(int[] nums,int target,int start,int end) {
        if(start <= end) {
            int middle = (start+end)/2;
            if(nums[middle] == target) {
                return middle;
            }
            if(start == end) {
                return -1;
            }
            if(nums[start]<nums[middle]) {
                if(nums[start]<=target && nums[middle]>=target) {
                    return search(nums,target,start,middle);
                } else {
                    return start_z(nums,target,middle+1,end);
                }
            } else {
                if(nums[middle+1]<=target && target<=nums[end]) {
                    return search(nums,target,middle+1,end);
                } else{
                    return start_z(nums,target,start,middle);
                }
            }
        }
        return -1;
    }
```

2.二维数组



注意从右上或者左下查找，例如从右上开始，此时横坐标为0，纵坐标为array[0].length，如果右上角的元素小于目标元素，那么第一行都不用搜索了，横坐标直接向下，如果右上角元素大于目标元素，那么最后一列不需要搜索，纵坐标向左



主要是理解二维数组遍历的时候，向不同方向移动的时候行值，列值是如何变动的

3.大根堆/小根堆

求解第K大/小的元素，求解前K大/小的元素

```
int[] array = new int[]{3,4,6,1,2,8,9,3,5};
PriorityQueue<Integer> smallQueue = new PriorityQueue<>((n1,n2)->n1-n2);  //小根堆
PriorityQueue<Integer> bigQueue = new PriorityQueue<>((n1,n2)->n2-n1); // 大根堆
for (int i=0;i<array.length;i++) {
bigQueue.add(array[i]);
smallQueue.add(array[i]);
}
System.out.println(smallQueue.poll());  // 1
System.out.println(smallQueue.poll());  // 2
System.out.println(bigQueue.poll());  // 9
System.out.println(bigQueue.poll());  // 8
```

4.回溯算法

```
// row表示前row行已经选好皇后
public void backTrack(List<String[]> temp,int n,int row) {
    if(row == temp.size()) {
        store.add(new ArrayList<>(temp));
        return;
    }
    // 遍历row行的第一列到最后一列
    for(int i=0;i<n;i++) {  // 每一层可供选择的数
        if(!isValid(temp,i,row)) {
            continue;
        }
        String[] strings = temp.get(row);
        strings[i]="queue";
        backTrack(temp,n, row: row+1);  // 每递归调用一次，层数加1
        strings[i]=".";
    }

}
```

暴力求解，可以画出决策树

> result = []

```
public void backtrack(路径, 选择列表) {
if 满足结束条件:
result.add(路径)
return
for 选择 in 选择列表:    // 选择列表即决策树分支数目
做选择
backtrack(路径, 选择列表)
撤销选择
}
```

可以引入int[]tag数组，判断记录某个分支是否被访问过；通过对分支排序，可以实现对决策时的剪枝处理

```
private void select2(List<Integer> source, List<List<Integer>> store,
List<Integer> temp,boolean[] tag) {
if(temp.size() == source.size()) {
   store.add(new ArrayList<>(temp));
   return;
}
for(int i=0;i<source.size();i++) {
   if(tag[i]) {
       continue;
   }
   if(i>0 && source.get(i)== source.get(i-1)&& !tag[i-1]) {   // 剪枝
       continue;
   }
   temp.add(source.get(i));
   tag[i]=true;
```

```
    select2(source,store,temp,tag);
    temp.remove(temp.size()-1);
    tag[i]=false;
}
}
```

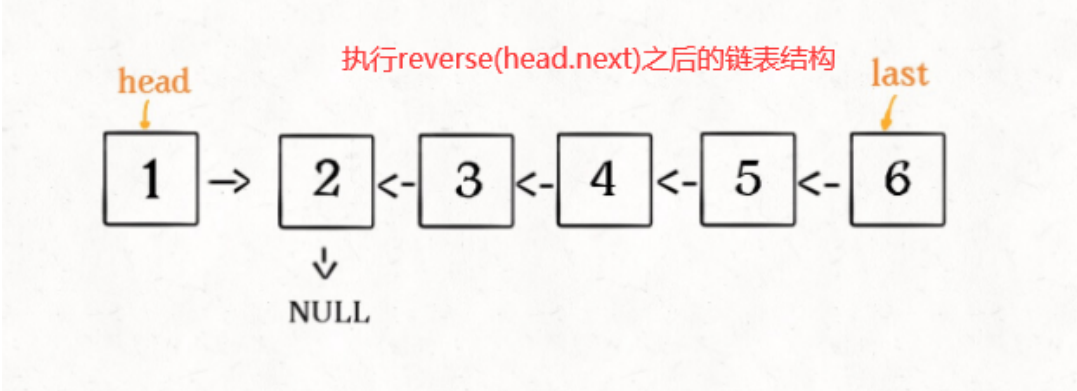
5.链表

链表反转：非递归

```
public ListNode reverseList(ListNode head) {
ListNode pre = null;
ListNode current = head;
while(current!=null) {
 // 需要记录下要反转的节点的下一个节点，不然直接反转该节点会丢失原来的关系。
 ListNode next = current.next;
 current.next = pre;    // 让待翻转的节点指向其前一个节点

// 开始翻转下一个节点，同样是要先找到该节点的前一个节点
    pre = current;
    current = next;  // 因为上面已经记录了该节点，所以直接让current指向记录的节点即可
}
return pre;  // 返回头节点
}
```

链表翻转：递归




```
public ListNode reverseList(ListNode head) {
if(head == null || head.next == null) {
  return head;
}
ListNode tail = reverseList(head.next);
head.next.next=head;
head.next=null;
return tail;
}
```

关键是明确reverseList函数的意义，即翻转从head开始到末尾节点的链表，同时返回反转后的头节点；显然reverseList(head.next)的意义就是翻转从head.next开始到末尾的链表，reverseList(head.next)执行结束则从head.next开始到结束的整个子链表已经翻转了，还剩head节点未翻转，所以有head.next.next=head; head.next=null;

翻转链表前n个节点

```
ListNode successor = null; // 后驱节点
```

```
// 反转以 head 为起点的 n 个节点，返回新的头结点
ListNode reverseN(ListNode head, int n) {
if (n == 1) {
// 记录第 n + 1 个节点
successor = head.next;
return head;
}
// 以 head.next 为起点，需要反转前 n - 1 个节点
ListNode last = reverseN(head.next, n - 1);

head.next.next = head;
// 让反转之后的 head 节点和后面的节点连起来
head.next = successor;
return last;
}
```

翻转链表m到n的节点

```
ListNode reverseBetween(ListNode head, int m, int n) {
// base case
if (m == 1) {
return reverseN(head, n);
}
// 前进到反转的起点触发 base case
head.next = reverseBetween(head.next, m - 1, n - 1);
return head;
}
```

如果 `m == 1`，就相当于反转链表开头的 `n` 个元素

按k个一组翻转

```
/** 反转区间 [a, b) 的元素，注意是左闭右开 */
ListNode reverse(ListNode a, ListNode b) {
ListNode pre, cur, nxt;
pre = null; cur = a; nxt = a;
// while 终止的条件改一下就行了
while (cur != b) {
   nxt = cur.next;
   cur.next = pre;
   pre = cur;
   cur = nxt;
}
// 返回反转后的头结点
return pre;
}

ListNode reverseKGroup(ListNode head, int k) {
if (head == null) return null;
// 区间 [a, b) 包含 k 个待反转元素
ListNode a, b;
a = b = head;
for (int i = 0; i < k; i++) {
   // 不足 k 个，不需要反转，base case
   if (b == null) return head;
   b = b.next;
```

```
}
// 反转前 k 个元素
ListNode newHead = reverse(a, b);
// 递归反转后续链表并连接起来
a.next = reverseKGroup(b, k);
return newHead;
}
```

旋转链表，先连为一个环

```
12      public ListNode rotateRight(ListNode head, int k) {
13          if(head == null || head.next == null) {
14              return head;
15          }
16          int n=0;
17          ListNode temp = head;
18          ListNode pre = head;
19          while(head != null) {
20              n++;
21              pre = head;
22              head = head.next;
23          }
24          pre.next = temp;
25          // 如果右循环一次，表示从倒数第一个节点处切割
26          for(int i=1;i<(n-k%n);i++) {  // 每循环右移n次，相当于没有右移  k%n为实际的右循环
    数
27              temp = temp.next;
28          }
29          ListNode newHead = temp.next;
30          temp.next = null;
31          return newHead;
32      }
```


6.二叉树非递归遍历 使用栈存放节点

```
// 非递归前序遍历
public List<Integer> preorderTraversal(TreeNode root) {
List<Integer> rs = new ArrayList<>();
Stack<TreeNode> stack = new Stack<>();
while (!stack.empty() || root != null) {
   while (root != null) {
       rs.add(root.val);  // 访问根节点
       stack.push(root);  // 左子树入栈
       root = root.left;
   }
   root = stack.pop();    // 弹出最后访问的左子树节点
   root = root.right;    // 遍历右子树，注意右子树可能不存在
}
return rs;
}
```

```
// 非递归中序遍历
public List<Integer> inorderTraversal(TreeNode root) {
List<Integer> rs = new ArrayList<>();
Stack<TreeNode> stack = new Stack<>();
while (!stack.empty() || root != null) {
   while (root != null) {
       stack.push(root);  // 比较前序遍历的话，在左子树入栈之前不访问根节点值
       root = root.left;
   }
```

```
    root = stack.pop();    // 弹出最后入栈的左子树节点
    rs.add(root.val);    // 访问其值
    root = root.right;    // 遍历右子树，右子树可能不存在
}
return rs;
}
```

```
/**
* 从逆后序遍历与先序遍历的关系中我们可以知道逆后序遍历序列为先序遍历交换左右子树的遍历顺序得到的，
* 所以我们得到了逆后序序列之后然后逆序就可以得到后序遍历的序列了，所以需要两个栈，第一个栈用来存储先序遍历交换左右子树的遍历的中介结果，
* 第二个是存储后序遍历的结果（逆序也就是可以理解为先进后出的意思）
* @param root
* @return
*/
// 非递归后续遍历
public List<Integer> postorderTraversal(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    LinkedList<Integer> rs = new LinkedList<>();
    if (root == null) {
        return rs;
    }
    stack.push(root);

    while (!stack.empty()) {
        TreeNode temp = stack.pop();
        rs.addFirst(temp.val);    // 每次访问都加到LinkedList的表头
        if (temp.left != null) {
            stack.push(temp.left);
        }
        if (temp.right != null) {
            stack.push(temp.right);
        }
    }
    return rs;
}
```

7.基于二叉树递归遍历重新构建二叉树关系

翻转二叉树(二叉树左子树跟右子树交换)

```
public TreeNode invertTree(TreeNode root) {
    if(root == null) {
        return null;
    }
    TreeNode right=null;
    TreeNode left = null;

    right = invertTree(root.right);  // 需要先保存下原来的root的右子节点，
    left = invertTree(root.left);    // 需要先保存下原来的root的左子节点

    root.left = right;
    root.right = left;
    return root;
```

```
}
```

二叉树展开为单链表(只有右子树且结构为root->左子树->右子树)

```
public void flatten(TreeNode root) {
   if(root == null) {
      return;
   }
   TreeNode oldLeft = root.left;  // 保存下老的的节点关系
   TreeNode oldRight = root.right;

   flatten(root.left);    // 展开子树
   flatten(root.right);

   root.left = null;        // 设置root的左侧为空
   root.right =  oldLeft;  // 将展开的左子树接到原来root的右侧

   TreeNode temp = root;
   while(temp.right != null) {    // 遍历寻找右子树应该插入的位置
       temp = temp.right;
   }
   temp.right=oldRight;
}
```

8.构建二叉树

构造最大二叉树

```
/* 主函数 */
TreeNode constructMaximumBinaryTree(int[] nums) {
return build(nums, 0, nums.length - 1);
}

/* 将 nums[lo..hi] 构造成符合条件的树，返回根节点 */
TreeNode build(int[] nums, int lo, int hi) {
// base case
if (lo > hi) {
return null;
}

// 找到数组中的最大值和对应的索引
int index = -1, maxVal = Integer.MIN_VALUE;
for (int i = lo; i <= hi; i++) {
if (maxVal < nums[i]) {
  index = i;
  maxVal = nums[i];
}
}

TreeNode root = new TreeNode(maxVal);  // 创建根节点
// 递归调用构造左右子树
root.left = build(nums, lo, index - 1);  // 创建左子树同时根节点指向左子树
root.right = build(nums, index + 1, hi);  // 创建右子树同时根节点指向右子树

return root;  //返回创建的树的根节点
}
// 根据前序以及中序来创建二叉树
```

```
TreeNode buildByPreOrderAndInOrder(int[] preOrder,int preStart,int
preEnd,int[] inOrder,int inStart,int inEnd) {
if(preOrder == null || preOrder.length == 0) {
   return null;
}
TreeNode root = new TreeNode(preOrder[preStart]);
int index = -1;
for(int i=inStart;i<=inEnd;i++) {
   if(inOrder[i]==preOrder[preStart]) {
       index = i;
        }
}
root.left =
buildByPreOrderAndInOrder(preOrder,preStart+1,?,inOrder,inStart,index-1);
root.right =
buildByPreOrderAndInOrder(preOrder,?,preEnd,inOrder,inStart+1,inEnd);
return root;
}
```

9.LRU算法与LFU算法

> LRU 算法的淘汰策略是 Least Recently Used，也就是每次淘汰那些最久没被使用的数据；而 LFU 算法的淘汰策略是 Least Frequently Used，也就是每次淘汰那些使用次数最少的数据。
>
> LRU 算法的核心数据结构是使用哈希链表 `LinkedHashMap`，首先借助链表的有序性使得链表元素维持插入顺序，同时借助哈希映射的快速访问能力使得我们可以在 O(1) 时间访问链表的任意元素。
>
> 而 LFU 算法相当于是把数据按照访问频次进行排序，这个需求恐怕没有那么简单，而且还有一种情况，如果多个数据拥有相同的访问频次，我们就得删除最早插入的那个数据。也就是说 LFU 算法是淘汰访问频次最低的数据，如果访问频次最低的数据有多条，需要淘汰最旧的数据。

10.BFS算法(广度优先遍历)

> **问题的本质就是让你在一幅「图」中找到从起点 `start` 到终点 `target` 的最近距离**

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
Queue<Node> q; // 核心数据结构
Set<Node> visited; // 避免走回头路

q.offer(start); // 将起点加入队列
visited.add(start);
int step = 0; // 记录扩散的步数

while (q not empty) {
   int sz = q.size();
   /* 将当前队列中的所有节点向四周扩散 */
   for (int i = 0; i < sz; i++) {
       Node cur = q.poll();
       /* 划重点：这里判断是否到达终点 */
       if (cur is target)
           return step;
       /* 将 cur 的相邻节点加入队列 */
       for (Node x : cur.adj())
           if (x not in visited) {
```

```
                q.offer(x);
                visited.add(x);
            }
    }
    /* 划重点：更新步数在这里 */
    step++;
}
}
```

二叉树的按层次遍历既是一种特殊的BFS

```
public List<TreeNode> levalOrderWithNoRecurse(TreeNode root) {
    if(root == null) {
        return null;
    }
List<TreeNode> list = new ArrayList<>();
Queue<TreeNode> queue  = new LinkedList<>();
queue.push(root);
while(!queue.isEmpty()) {
   int size = queue.size();
   for(int i=0;i<queue.size();i++) {
       TreeNode node = queue.poll();
       list.add(node);
       if(node.left != null) {
           queue.push(node.left);
       }
       if(node.right != null) {
           queue.push(node.right);
       }
   }
}
return list;
}
```

11.数据流求中位数

使用两个优先级队列PriorityQueue,一个大根堆一个小根堆，大根堆存储的是较小的数，小根堆存储的是较大的数，保证小根堆的数都大于大根堆的数，大根堆与小根堆size差<=1，每次添加元素先加入大根堆，然后把大根堆堆顶元素取出来放入小根堆。如果要取中位数，比较大根堆与小根堆容量大小，容量大的取出堆顶元素即可，容量相同则每一个队列取出一个元素求平均值。

```
private PriorityQueue<Integer> bigQueue = new PriorityQueue<>((a,b)->b-a);
 // 大根堆存的是较小的数
private PriorityQueue<Integer> smallQueue = new PriorityQueue<>();   // 小根堆存的是较大的数

public void addNum(int num) {
   if(bigQueue.size() > smallQueue.size()) {
       bigQueue.offer(num);
       smallQueue.offer(bigQueue.poll());
   } else {
       smallQueue.offer(num);
       bigQueue.offer(smallQueue.poll());
   }
}

public double findMedian() {
```

```
    if(bigQueue.size()>smallQueue.size()) {
        return bigQueue.peek().longValue();
    } else if(bigQueue.size()<smallQueue.size()) {
        return smallQueue.peek().longValue();
    } else {
        return
(smallQueue.peek().doubleValue()+bigQueue.peek().doubleValue())/2;
    }
}
```

12.单调栈

> 从数组尾部开始入栈，如果比栈顶元素大，则移除栈顶元素继续与新的栈顶元素比较直到找到一个栈顶元素比当前元素大，然后当前元素入栈作为新的栈顶元素。从栈顶->栈底元素依次增大

```
for(int i=nums2.length-1;i>=0;i--) {
 // 此栈为递归减小，栈顶的元素比较小，每次与栈顶元素比较，比栈顶元素大则移除栈顶元素继续比较
 while (!stack.isEmpty() && stack.peek()<=nums2[i]) {
        stack.pop();
 }
// 如果栈不为空说明栈中存在比该元素大的值，并且此时栈顶是第一个比他大的
int result = stack.isEmpty()?-1:stack.peek();
map.put(nums2[i],result);  // 记录该元素右侧第一个比他大的元素
stack.push(nums2[i]);  // 将该元素入栈，然后该元素左侧的元素比较的时候就会首先跟该元素比较
}
```

13. 单调队列

> 使用LinkedList构建，队头元素为最大值，队尾为最小值，每次新加入元素的话，从队尾开始比较如果队尾元素比较小，那么就移除队尾元素，直到找到一个元素比待加入的元素大，然后把该元素加入队列的尾部

```
static class WindowQueue {
private LinkedList<Integer> queue = new LinkedList<>();
public void push(int i) {
   while (!queue.isEmpty() && queue.getLast() < i) {
        queue.removeLast();
    }
    queue.addLast(i);
}

 public int max() {
    return queue.getFirst();
 }

 public void pop(int i) {
    if(i == queue.getFirst()) {
        queue.removeFirst();
     }
}
}
```

14.栈实现队列与队列实现栈

> 两个栈实现队列

```
public void push(int x) {
   stackA.push(x);
}
public int pop() {        // 出队的时候判断stackB是否为空，为空的话把stackA的值都加入stackB，然后将stackB的栈顶出栈即可
if(stackB.isEmpty()) {
  while (!stackA.isEmpty()) {
      stackB.push(stackA.pop());
  }
}
if(!stackB.isEmpty()) {
    return stackB.pop();
}
return -1;
}
public int peek() {
if(stackB.isEmpty()) {
   while (!stackA.isEmpty()) {
       stackB.push(stackA.pop());
   }
}
if(!stackB.isEmpty()) {
  return stackB.peek();
}
return -1;
}
public boolean empty() {
return stackA.isEmpty() && stackB.isEmpty();
}
```

队列实现栈

```
private LinkedList<Integer> linkedList = new LinkedList<>();

public void push(int x) {
   linkedList.addFirst(x);
}
public int pop() {
   return linkedList.removeFirst();
}
public int top() {
   return linkedList.getFirst();
}
public boolean empty() {
   return linkedList.isEmpty();
}
```

15.o(1)时间，查询/删除数组中任意元素

> 底层使用数组，数组地址连续，然后使用Map记录数组元素跟坐标的位置，删除元素的话跟数组末尾的元素交换再删除即可

16.Set去重

```
// Set 去重
Set<List<Integer>> ss = store.stream().map(one -> {
return
one.stream().sorted(Comparator.comparing(Integer::intValue)).collect(Collect
ors.toList());
}).collect(Collectors.toSet());
```

17.三个线程轮流打印

```
1. public class ThreadThreadp{
2.     public static String TAG = "ThreadThreadp";
3.     private int flag = 0;
4.
5.     public synchronized void printa() throws InterruptedException{
6.         while(true){
7.             if(flag==0){
8.                 Log.d(TAG,"A");
9.                 flag = 1;
10.                notifyAll();
11.            }
12.            wait();
13.        }
14.    }
15.
16.    public synchronized void printb() throws InterruptedException{
17.        while(true){
18.            if(flag==1){
19.                Log.d(TAG,"B");
20.                flag = 2;
21.                notifyAll();
22.            }
23.            wait();
24.        }
25.    }
26.
27.    public synchronized void printc() throws InterruptedException{
28.        while(true){
29.            if(flag==2){
30.                Log.d(TAG,"C");
31.                flag = 0;
32.                notifyAll();
33.            }
34.            wait();
35.        }
36.    }
37.
38.    public static void main(){
39.        ThreadThreadp t = new ThreadThreadp();
40.        PrintA printA = new PrintA(t);
41.        PrintB printB = new PrintB(t);
42.        PrintC printC = new PrintC(t);
43.
44.        Thread t1 = new Thread(printA);
```

```
        Thread t2 = new Thread(printB);
        Thread t3 = new Thread(printC);
        t1.start();
        t2.start();
        t3.start();
    }
}

class PrintA implements Runnable{
    private ThreadThreadp t;
    PrintA(ThreadThreadp t){
        this.t = t;
    }

    @Override
    public void run() {
        try {
            t.printa();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```