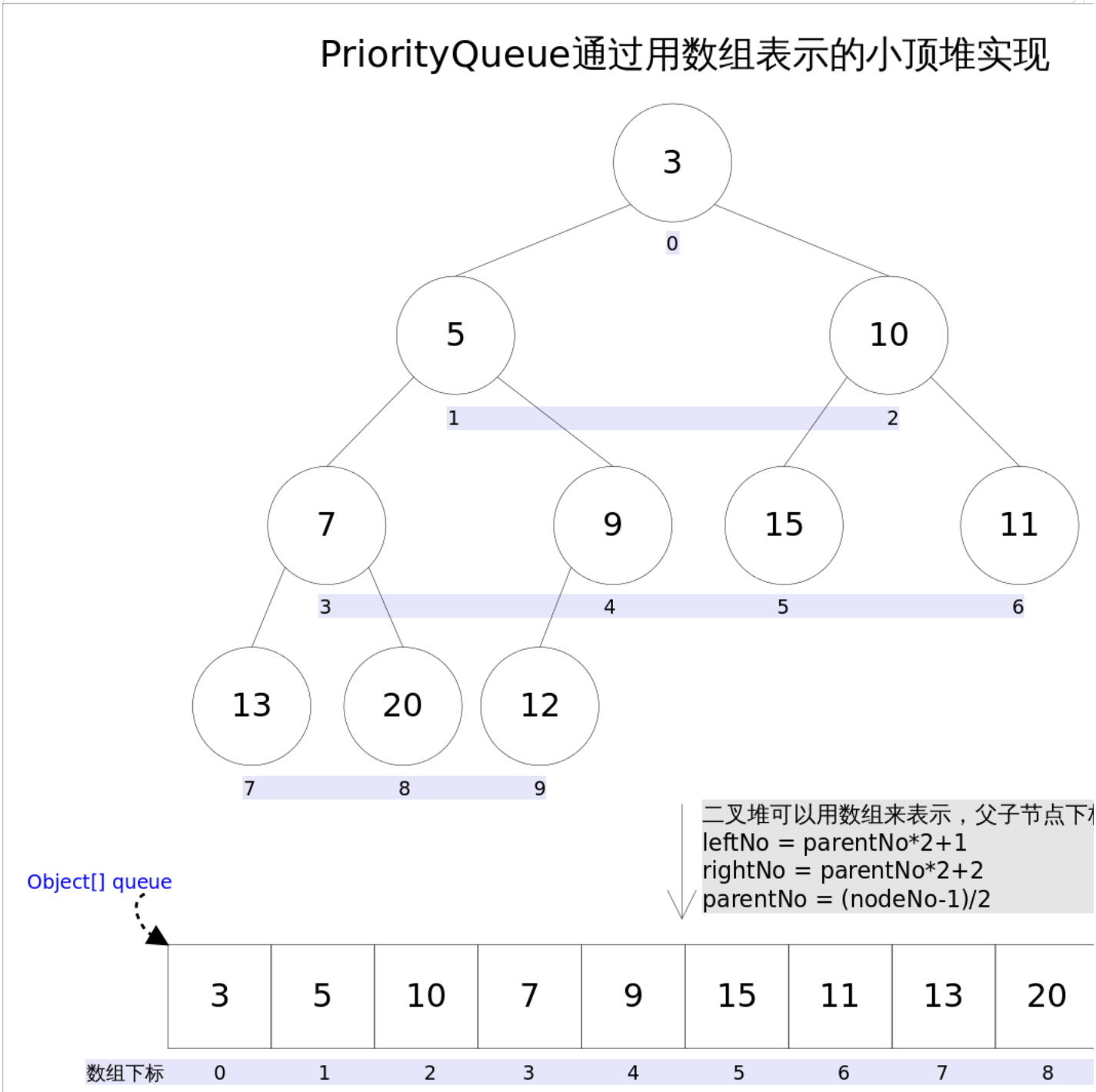
1.阻塞队列的实现

lock + 多个条件（condition）阻塞控制
实现原理：通过可重入锁ReenTrantLock+Condition 来实现多线程之间的同步效果
入队过程：
add方法：插入成功返回true；插入失败抛异常
put方法：插入元素到尾部，如果失败则调用Condition.await()方法进行阻塞等待，直到被唤醒；
offer方法：插入元素到尾部，如果失败则直接返回false，
offer(timeout)：插入元素到尾部，如果失败则调用Condition.await（timeout）方法进行阻塞等待指定时间，直到被唤醒或阻塞超时，还是失败就返回false
而一旦插入成功，就会唤醒出队的等待操作，执行出队的Condition的signal()方法
出队过程：
主要方法为：poll（）、take（）、remove（）
基本上和入队过程类似，出队结束会唤醒入队的等待操作，执行入队的Condition的signal()方法
而不管是入队操作还是出队操作，都会通过ReentrantLock来控制同步效果，通过两个Condition来控制线程之间的通信效果

2.延迟队列的实现

DelayQueue主要也是通过ReentrantLock+Condition来保证线程安全，而内部还采用了ProrityQueue来保证队列的优先级，实际就是按延时的时间来进行排序，延迟时间最短的排在队列的头部，
所以每次从头部获取的元素都是最先会过期的数据。

ProrityQueue的实现原理是使用二叉树小顶堆来实现的
https://blog.csdn.net/u010623927/article/details/87179364
Condition介绍：
https://blog.csdn.net/xh13007612005/article/details/89678117



3.LRU本地缓存的实现

双向链表+hashMap实现

```
class Solution {
```

```
    private LinkedList<Integer> linkedList;
    private Map<Integer, Integer> map;

    private int max_size;
    private int cur_size = 0;

    public Solution(int capacity) {
        linkedList = new LinkedList<>();
        map = new HashMap<>();
        this.max_size = capacity;
    }

    public int get(int key) {
        if(!map.containsKey(key)){
            return -1;
        }
        int val = map.get(key);
        Object o = key;
        linkedList.remove(o);
        linkedList.addLast(key);
        return val;
    }

    public void put(int key, int value) {
        if(map.containsKey(key)){
            // 这个put不能省略，即时key存在，若新添加的value更新了，那刚好就将value更新，如果省略，则value更新不了
            map.put(key, value);
            Object o = key;
            linkedList.remove(o);
            linkedList.addLast(key);
        }else{
            map.put(key, value);
            cur_size++;
            linkedList.addLast(key);
            if(cur_size>max_size){
                int tmp = linkedList.removeFirst();
                map.remove(tmp);
                cur_size--;
            }
        }
    }
}

package code.fragment;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
public class LRUCache<V> {
    /**
     * 容量
     */
    private int capacity = 1024;
    /**
     * Node记录表
     */
    private Map<String, ListNode<String, V>> table = new ConcurrentHashMap<>();
    /**
     * 双向链表头部
     */
    private ListNode<String, V> head;
    /**
     * 双向链表尾部
     */
    private ListNode<String, V> tail;

    public LRUCache(int capacity) {
        this();
        this.capacity = capacity;
    }

    public LRUCache() {
        head = new ListNode<>();
        tail = new ListNode<>();
        head.next = tail;
        head.prev = null;
        tail.prev = head;
        tail.next = null;
    }

    public V get(String key) {
        ListNode<String, V> node = table.get(key);
        //如果Node不在表中，代表缓存中并没有
        if (node == null) {
            return null;
        }
        //如果存在，则需要移动Node节点到表头
        //截断链表，node.prev -> node  -> node.next ====> node.prev -> node.next
        //         node.prev <- node <- node.next  ====>  node.prev <- node.next
        node.prev.next = node.next;
        node.next.prev = node.prev;

        //移动节点到表头
        node.next = head.next;
        head.next.prev = node;
        node.prev = head;
        head.next = node;
        //存在缓存表
        table.put(key, node);
```

```
        return node.value;
    }

    public void put(String key, V value) {
        ListNode<String, V> node = table.get(key);
        //如果Node不在表中，代表缓存中并没有
        if (node == null) {
            if (table.size() == capacity) {
                //超过容量了 ,首先移除尾部的节点
                table.remove(tail.prev.key);
                tail.prev = tail.next;
                tail.next = null;
                tail = tail.prev;
            }
            node = new ListNode<>();
            node.key = key;
            node.value = value;
            table.put(key, node);
        }
        //如果存在，则需要移动Node节点到表头
        node.next = head.next;
        head.next.prev = node;
        node.prev = head;
        head.next = node;
    }

    /**
     * 双向链表内部类
     */
    public static class ListNode<K, V> {
        private K key;
        private V value;
        ListNode<K, V> prev;
        ListNode<K, V> next;

        public ListNode(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public ListNode() {
        }
    }

    public static void main(String[] args) {
        LRUCache<ListNode> cache = new LRUCache<>(4);
        ListNode<String, Integer> node1 = new ListNode<>("key1", 1);
        ListNode<String, Integer> node2 = new ListNode<>("key2", 2);
        ListNode<String, Integer> node3 = new ListNode<>("key3", 3);
        ListNode<String, Integer> node4 = new ListNode<>("key4", 4);
        ListNode<String, Integer> node5 = new ListNode<>("key5", 5);
        cache.put("key1", node1);
        cache.put("key2", node2);
        cache.put("key3", node3);
        cache.put("key4", node4);
        cache.get("key2");
        cache.put("key5", node5);
        cache.get("key2");
    }
}
```

4.数组三个数和为指定目标数值的三个数

```
/**
* 双指针法
* @param nums
* @param target
*/
public static void finger(int[] nums, int target) {
    if (nums == null) {
        return;
    }

    // 记录数组的长度
    int lengths = nums.length-1;

    // 先固定一个值，要么固定最左边的值，要么固定最右边的值
    for (int i = 0; i < lengths; i++) {
        int left = (i + 1); // 左指针
        int right = lengths; // 右指针

        int low = (i + 1); // 临时左指针
        int high = lengths; // 临时右指针

        // 当左指针小于右指针的时候，就不需要循环了
        while (left < right) {
            // 比较临时指针
            while (low < high) {
                if ((nums[i] + nums[low] + nums[high]) == target) {
                    System.out.println("三个数之和等于" + target + "的分别是: " + nums[i] +"、" + nums[low] + "和" + nums[high]);
                }
                high--; // 相当于从右往左一直找数
            }

            high = right--; // 循环找完一遍数之后，把临时右指针还原，右指针减1
            low = ++left; // 因为临时左指针第一个元素已经比较完，所有临时左指针加1，左指针也加1

            while (low < high) {
```

```
                if ((nums[i] + nums[low] + nums[high]) == target) {
                    System.out.println("三个数之和等于" + target + "的分别是: " + nums[i] +"、" + nums[low] + "和" + nums[high]);
                }
                low++; // 相当于从左往右一直找数
            }
            low = left; // 循环找完一遍数之后，把临时左指针还原，下次循环使用
                high--; // 临时右指针也减1，下次循环使用
        }
    }
}
```

5.topN

```
    public class TopN {
    public static int N = 10;           //Top10
    public static int LEN = 100000000; //1亿个整数
    public static int arrs[] =  new int[LEN];
    public static int arr[] = new int[N];
    //数组长度
    public static int len = arr.length;
    //堆中元素的有效元素 heapSize<=len
    public static int heapSize = len;
    public static void main(String[] args) {
        //生成随机数组
       for(int i = 0;i<LEN;i++){
           arrs[i] = new  Random().nextInt(999999999);
       }
       //构建初始堆
       for(int i =  0;i<N;i++){
           arr[i] = arrs[i];
       }
       //构建小顶堆
        long start =System.currentTimeMillis();
       buildMinHeap();
       for(int i = N;i<LEN;i++){
           if(arrs[i] > arr[0]){
               arr[0] = arrs[i];
               minHeap(0);
           }
       }
        System.out.println(LEN+"个数，求Top"+N+"，耗时"+(System.currentTimeMillis()-start)+"毫秒");
       print();
    }
    /**
     * 自底向上构建小堆
     */
    public static void buildMinHeap(){
        int size = len / 2;
        for(int i = size;i>=0;i--){
            minHeap(i);
        }
    }
    /**
     * i节点为根及子树是一个小堆
     * @param i
     */
    public static void minHeap(int i){
        int l = left(i);
        int r = right(i);
        int index = i;
        if(l<heapSize && arr[l]<arr[index]){
            index = l;
        }
        if(r<heapSize && arr[r]<arr[index]){
            index = r;
        }
        if(index != i){
            int t = arr[index];
            arr[index] = arr[i];
            arr[i] = t;
            //递归向下构建堆
            minHeap(index);
        }
    }
    /**
     * 返回i节点的左孩子
     * @param i
     * @return
     */
    public static int left(int i){
        return 2*i;
    }
    /**
     * 返回i节点的右孩子
     * @param i
     * @return
     */
    public static int right(int i){
        return 2*i+1;
    }
    /**
     * 打印
     */
    public  static void print(){
        for(int a:arr){
            System.out.print(a+",");
        }
        System.out.println();
```

```
    }
```

6.排序算法的一种
https://www.cnblogs.com/fnlingnzb-learner/p/9083552.html

最基础的四个算法：冒泡、选择、插入、快排中，快排的时间复杂度最小O

排序法	平均时间	最差情形	稳定度	额外
冒泡	O(n2)	O(n2)	稳定	O(1)
选择	O(n2)	O(n2)	不稳定	O(1)
插入	O(n2)	O(n2)	稳定	O(1)
基数	O(logRB)	O(logRB)	稳定	O(n)
Shell	O(nlogn)	O(ns) 1<s<2	不稳定	O(1)
快速	O(nlogn)	O(n2)	不稳定	O(nl
归并	O(nlogn)	O(nlogn)	稳定	O(1)
堆	O(nlogn)	O(nlogn)	不稳定	O(1)

```
package QuickSort;
public class QuickSort
{
    public static void main(String[] args)
    {
        int[] a = { 49, 38, 65, 97, 76, 13, 27, 49, 78, 34, 12, 64, 1, 8 };
        System.out.println("排序之前：");
        for (int i = 0; i < a.length; i++)
        {
            System.out.print(a[i] + " ");
        }
        // 快速排序
        quick(a);
        System.out.println();
        System.out.println("排序之后：");
        for (int i = 0; i < a.length; i++)
        {
            System.out.print(a[i] + " ");
        }
    }
    private static void quick(int[] a)
    {
        if (a.length > 0)
        {
            quickSort(a, 0, a.length - 1);
        }
    }
    private static void quickSort(int[] a, int low, int high)
    {
        if (low < high)
        { // 如果不加这个判断递归会无法退出导致堆栈溢出异常
            int middle = getMiddle(a, low, high);
            quickSort(a, 0, middle - 1);
            quickSort(a, middle + 1, high);
        }
    }
    private static int getMiddle(int[] a, int low, int high)
    {
        int temp = a[low];// 基准元素
        while (low < high)
        {
```

```
            // 找到比基准元素小的元素位置
            while (low < high && a[high] >= temp)
            {
                high--;
            }
            a[low] = a[high];
            while (low < high && a[low] <= temp)
            {
                low++;
            }
            a[high] = a[low];
        }
        a[low] = temp;
        return low;
    }
}
```

7.一万以内的质数

```
package com.huaxin;
public class TestPrimeNum {
    public static void main(String[] args){
        //方法三
        boolean flag03 = false;
        long start03 = System.currentTimeMillis();
        for(int i = 2; i < 100000; i++){
            for(int j = 2; j <= Math.sqrt(i); j++){
                if(i % j == 0){
                    flag03 = true;
                    break;
                }
            }
            if(flag03 == false){
                System.out.println(i);
            }
        }
        long end03 = System.currentTimeMillis();
        System.out.println("经历的时间为: " + (end03 - start03));
    }
}
```

8.二叉树遍历

1.先序遍历（根>左>右）

递归的方式

```
    /**
     * 二叉树前序遍历    根-> 左-> 右
     * @param node    二叉树节点
     */
    public static void preOrderTraveral(TreeNode node){
        if(node == null){
            return;
        }
        System.out.print(node.data+" ");
        preOrderTraveral(node.leftChild);
        preOrderTraveral(node.rightChild);
    }
```

非递归的方式

```
    public static void preOrderTraveralWithStack(TreeNode node){
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode treeNode = node;
        while(treeNode!=null || !stack.isEmpty()){
            //迭代访问节点的左孩子，并入栈
            while(treeNode != null){
                System.out.print(treeNode.data+" ");
                stack.push(treeNode);
                treeNode = treeNode.leftChild;
            }
            //如果节点没有左孩子，则弹出栈顶节点，访问节点右孩子
            if(!stack.isEmpty()){
                treeNode = stack.pop();
                treeNode = treeNode.rightChild;
            }
        }
    }
```

2.中序遍历（左>根>右）

递归的方式

```
    /**
     * 二叉树中序遍历    左-> 根-> 右
     * @param node    二叉树节点
     */
    public static void inOrderTraveral(TreeNode node){
        if(node == null){
            return;
        }
        inOrderTraveral(node.leftChild);
        System.out.print(node.data+" ");
        inOrderTraveral(node.rightChild);
    }
```

非递归的方式

```
    public static void inOrderTraveralWithStack(TreeNode node){
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode treeNode = node;
        while(treeNode!=null || !stack.isEmpty()){
            while(treeNode != null){
                stack.push(treeNode);
                treeNode = treeNode.leftChild;
            }
```

```
            if(!stack.isEmpty()){
                treeNode = stack.pop();
                System.out.print(treeNode.data+" ");
                treeNode = treeNode.rightChild;
            }


        }
    }
```

3.后续遍历（左>右>根）

递归的方式

```
    /**
     * 二叉树后序遍历    左-> 右-> 根
     * @param node     二叉树节点
     */
    public static void postOrderTraveral(TreeNode node){
        if(node == null){
            return;
        }
        postOrderTraveral(node.leftChild);
        postOrderTraveral(node.rightChild);
        System.out.print(node.data+" ");
    }
```

非递归的方式

```
    public static void postOrderTraveralWithStack(TreeNode node){
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode treeNode = node;
        TreeNode lastVisit = null;   //标记每次遍历最后一次访问的节点
        while(treeNode!=null || !stack.isEmpty()){//节点不为空，结点入栈，并且指向下一个左孩子
            while(treeNode!=null){
                stack.push(treeNode);
                treeNode = treeNode.leftChild;
            }
            //栈不为空
            if(!stack.isEmpty()){
                //出栈
                treeNode = stack.pop();
                /**
                 * 这块就是判断treeNode是否有右孩子，
                 * 如果没有输出treeNode.data，让lastVisit指向treeNode，并让treeNode为空
                 * 如果有右孩子，将当前节点继续入栈，treeNode指向它的右孩子,继续重复循环
                 */
                if(treeNode.rightChild == null || treeNode.rightChild == lastVisit) {
                    System.out.print(treeNode.data + " ");
                    lastVisit = treeNode;
                    treeNode  = null;
                }else{
                    stack.push(treeNode);
                    treeNode = treeNode.rightChild;
                }
            }
        }
    }
```

4.层序遍历

```
    public static void levelOrder(TreeNode root){
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while(!queue.isEmpty()){
            root = queue.pop();
            System.out.print(root.data+" ");
            if(root.leftChild!=null) queue.add(root.leftChild);
            if(root.rightChild!=null) queue.add(root.rightChild);
        }
    }
```

9.单链表翻转

反转-循环法

1.先指向前一个

2.把前一个赋值为当前本身

3.当前本身赋值为下一个

反转-递归发

```
public ListNode ReverseList(ListNode head) {
        if (head == null || head.nextNode == null) {
            return head;
        }
        ListNode next = head.nextNode;
        head.nextNode = null;
        ListNode newHead = ReverseList(next);
        next.nextNode = head;
        return newHead;
    }

public static Node reverseList(Node head){
    if (head == null || head.getNext() == null){
        return head;
    }
    Node next = head.getNext();
    head.setNext(null);
    Node newNext = reverseList(next);
    next.setNext(head);
    return newNext;
}
```

10.二分查找

```
int BinarySearch(int array[], int n, int value)
```

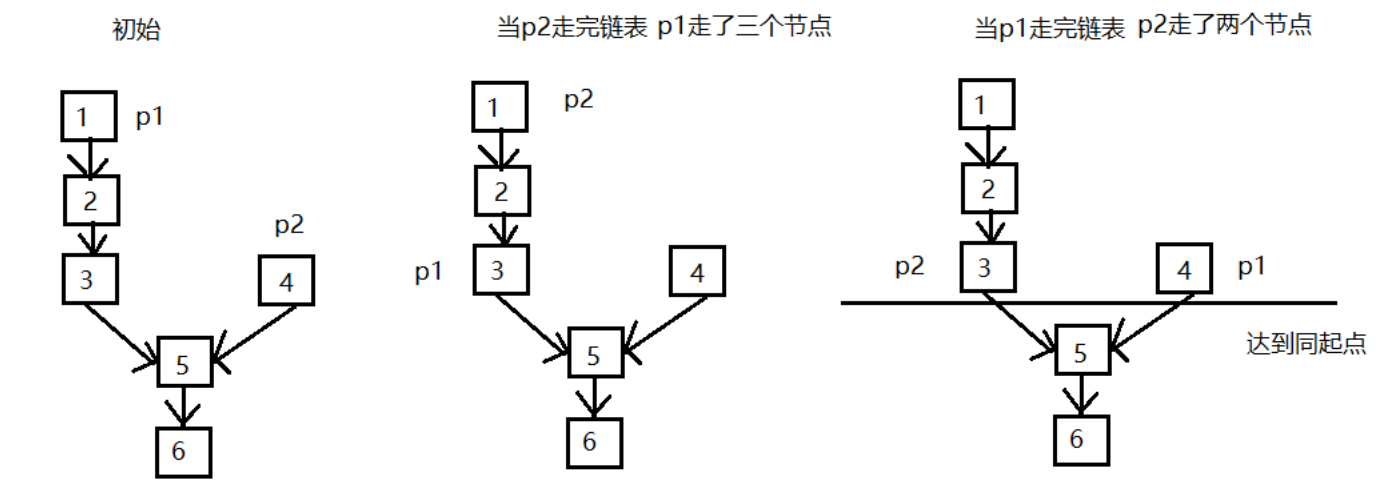
```
{
    int left = 0;
    int right = n - 1;
    //如果这里是int right = n 的话，那么下面有两处地方需要修改，以保证一一对应：
    //1、下面循环的条件则是while(left < right)
    //2、循环内当 array[middle] > value 的时候，right = mid

    while (left <= right)  //循环条件，适时而变
    {
        int middle = left + ((right - left) >> 1);  //防止溢出，移位也更高效。同时，每次循环都需要更新。

        if (array[middle] > value)
        {
            right = middle - 1;  //right赋值，适时而变
        }
        else if(array[middle] < value)
        {
            left = middle + 1;
        }
        else
            return middle;
        //可能会有读者认为刚开始时就要判断相等，但毕竟数组中不相等的情况更多
        //如果每次循环都判断一下是否相等，将耗费时间
    }
    return -1;
}


public static Integer search(Integer [] array, int k){
    if (array == null || array.length == 0){
        return null;
    }
    int left = 0;
    int right = array.length - 1;
    while (left <= right){
        int middle = (left + right) / 2;
        if (array[middle] == k){
            return middle;
        }else if (k > array[middle]){
            left = middle + 1;
        }else if (k < array[middle]){
            right = middle - 1;
        }
    }
    return null;
}
```

11.两个链表的公共节点



https://blog.csdn.net/da

```
package com.codinginterviews.list;
import java.util.Stack;
/*
* 题目：
* 两个链表的第一个公共结点 -- newcoder 剑指Offer 36
*
* 题目描述：
* 输入两个链表，找出它们的第一个公共结点。
*/
public class FindFirstCommonNode {
    static class ListNode{
        private int val;

        private ListNode next;

        public ListNode (int val) {
            this.val = val;
        }

        @Override
        public String toString() {
            if (this.next == null) {
```

```
            return String.valueOf(this.val);
        }
        return this.val + "->" + this.next.toString();
    }

}

/*
 * 获取单链表的第一个交点
 * 如果两个链表相交，因为链表的指针只有一个，所以相交后，后面的节点必定重合，因此相交只能是Y型，而不是X型
 *
 * node0->node1->node5->node6
 * node2->node3->node5->node6
 *
 * 思路:
 * 分别把两个链表压入栈中，弹出元素，最后一个相同的元素即为第一个交点
 *
 * 分析：此方式简单易行，需要遍历两次(两个链表+一次栈寻找);但是需要借助额外空间，需要优化
 */
public static ListNode findFirstCommonNode(ListNode pHead1, ListNode pHead2) {
    if(pHead1 == null || pHead2 == null) {
        return null;
    }
    Stack<ListNode> stack1 = new Stack<>();
    while (pHead1 != null) {
        stack1.push(pHead1);
        pHead1 = pHead1.next;
    }

    Stack<ListNode> stack2 = new Stack<>();
    while (pHead2 != null) {
        stack2.push(pHead2);
        pHead2 = pHead2.next;
    }

    ListNode commonNode = null;

    while (!stack1.empty() && !stack2.empty()) {
        ListNode pop1 = stack1.pop();
        ListNode pop2 = stack2.pop();
        if ( pop1 == pop2) {
            commonNode = pop1;
        } else {
            break;
        }
    }

    return commonNode;
}

/*
 * 获取单链表的第一个交点
 * 如果两个链表相交，因为链表的指针只有一个，所以相交后，后面的节点必定重合，因此相交只能是Y型，而不是X型
 *
 * node0->node1->node5->node6
 * node2->node5->node6
 *
 * 思路:
 * 1、分别读取两个链表长度，然后用两个指针分别指向两个链表
 * 2、长的先走lenMax-lenMin步，元素相等时，则找到第一个相交节点
 *
 * 分析：此方式简单易行，需要遍历两次(两个链表 + 一次寻找)
 */
public static ListNode findFirstCommonNodeII(ListNode pHead1, ListNode pHead2) {
    if (pHead1 == null || pHead2 == null) {
        return null;
    }

    // 分别遍历获取长度
    int len1 = 0;
    ListNode tmpNode1 = pHead1;
    while (tmpNode1 != null) {
        len1++;
        tmpNode1 = tmpNode1.next;
    }

    int len2 = 0;
    ListNode tmpNode2 = pHead2;
    while (tmpNode2 != null) {
        len2++;
        tmpNode2 = tmpNode2.next;
    }

    // 赋值快慢指针
    ListNode fast = null;
    ListNode low = null;
    int num = 0;
    if (len1 >= len2) {
        fast = pHead1;
        low = pHead2;
        num = len1 - len2;
    } else {
        fast = pHead2;
        low = pHead1;
        num = len2 - len1;
    }
```

```
        // fast 指针 先走num步
        for (int i=0; i< num; i++) {
            fast = fast.next;
        }

        while (fast != null && low != null) {
            if (fast == low) {
                return fast;
            }
            fast = fast.next;
            low = low.next;
        }

        return null;
    }

    /**
     * 思路(摘自牛客网):
     * 1、长度相同有公共结点，第一次就遍历到；没有公共结点，走到尾部NULL相遇，返回NULL
     * 2、长度不同有公共结点，第一遍差值就出来了，第二遍一起到公共结点；没有公共，一起到结尾NULL。
     */
    public static ListNode findFirstCommonNodeIII(ListNode pHead1, ListNode pHead2) {
        if (pHead1 == null || pHead2 == null) {
            return null;
        }

        ListNode p1 = pHead1;
        ListNode p2 = pHead2;

        while (p1 != p2) {
            p1 = p1 == null ? pHead2 : p1.next;
            p2 = p2 == null ? pHead1 : p2.next;
        }

        return p1;
    }

    public static void main(String args[]) {

        ListNode node1 = new ListNode(11);
        ListNode node2 = new ListNode(12);
        node1.next = node2;

        ListNode head1 = createTestLinkedList(7, node1);
        ListNode head2 = createTestLinkedList(8, node1);


        // 链表相交
        System.out.println("link common node: " + findFirstCommonNode(head1, head2));
        // 链表相交
        System.out.println("link common node: " + findFirstCommonNodeII(head1, head2));
        // 链表相交
        System.out.println("link common node: " + findFirstCommonNodeIII(head1, head2));

    }

    private static ListNode createTestLinkedList(int n, ListNode addNode) {
        ListNode head = new ListNode(0);
        ListNode curNode = head;
        for (int i = 1; i < n; i++) {
            curNode.next = new ListNode(i);
            curNode = curNode.next;
        }
        curNode.next = addNode;
        return head;
    }
}
```

12.字符串查找

KMP

```
/**
 * KMPSearch 算法
 *
 * @author stecai
 */
public class KMPSearch {
    /**
     * 获得字符串的next函数值
     *
     * @param str
     * @return next函数值
     */
    private static int[] calculateNext(String str) {
        int i = -1;
        int j = 0;
        int length = str.length();
        int next[] = new int[length];
        next[0] = -1;

        while (j < length - 1) {
            if (i == -1 || str.charAt(i) == str.charAt(j))
                i++;
                j++;
                next[j] = i;
            } else {
                i = next[i];
            }
        }
```

```
29.
30.          return next;
31.      }
32.
33.      /**
34.       * KMP匹配字符串
35.       *
36.       * @param source 目标字符串
37.       * @param pattern 指定字符串
38.       * @return 若匹配成功，返回下标，否则返回-1
39.       */
40.      public static int match(String source, String pattern) {
41.          int i = 0;
42.          int j = 0;
43.          int input_len = source.length();
44.          int kw_len = pattern.length();
45.          int[] next = calculateNext(pattern);
46.
47.          while ((i < input_len) && (j < kw_len)) {
48.              // 如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
49.              if (j == -1 || source.charAt(i) == pattern.charAt(j)) {
50.                  j++;
51.                  i++;
52.              } else {
53.                  // 如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j],
54.                  // next[j]即为j所对应的next值
55.                  j = next[j];
56.              }
57.          }
58.
59.          if (j == kw_len) {
60.              return i - kw_len;
61.          } else {
62.              return -1;
63.          }
64.      }
65.  }
```

13.无重复字符的最长子串

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        Set<Character> set = new HashSet<>();
        int ans = 0, i = 0, j = 0;
        while (i < n && j < n) {
            // try to extend the range [i, j]
            if (!set.contains(s.charAt(j))){
                set.add(s.charAt(j++));
                ans = Math.max(ans, j - i);
            }
            else {
                set.remove(s.charAt(i++));
            }
        }
        return ans;
    }
}
```