# 目录


**目录**
**经典算法**
- Manacher算法
  - 原始问题
  - 进阶问题
- BFPRT算法
- morris遍历二叉树
  - 遍历规则
  - 先序、中序序列
  - 后序序列
  - 时间复杂度分析
- 求和为aim的最长子数组长度
  - 拓展
    - 求奇数个数和偶数个数相同的最长子数组长度
    - 求数值为1的个数和数值为2的个数相同的最长子数组（数组只含0、1、2三种元素）
  - 进阶
    - 求任意划分数组的方案中，划分后，异或和为0的子数组最多有多少个
- 高度套路的二叉树信息收集问题
  - 求一棵二叉树的最大搜索二叉子树的结点个数
  - 求一棵二叉树的最远距离
  - 舞会最大活跃度
- 求一个数学表达式的值
- 求异或和最大的子数组
  - 暴力解
  - 优化暴力解
  - 最优解
- 求和为aim的最长子数组（都大于0）
- 求和小于等于aim的最长子数组（有正有负有0）
- 环形单链表的约瑟夫问题

**经典结构**
- 窗口最大值更新结构
  - 最大值更新结构
  - 例题
    - 窗口移动
    - 求达标的子数组个数
- 单调栈结构
  - 原始问题
  - 给你一些数，创建一棵大根堆二叉树
  - 找出矩阵中一片1相连的最大矩形
  - 烽火相望
    - 1、数组中无重复的数
    - 2、数组中可能有重复的数
- 搜索二叉树
  - 平衡二叉树/AVL树
    - 平衡性
  - 典型搜索二叉树——AVL树、红黑树、SBT树的原理
    - AVL树

红黑树
SBT树
旋转——Rebalance
Java中红黑树的使用
案例
The Skyline Problem
跳表
添加数据
查找数据
删除数据
遍历数据
**从暴力尝试到动态规划**
换钱的方法数
暴力尝试
缓存每个状态的结果，以免重复计算
确定依赖关系，寻找最优解
排成一条线的纸牌博弈问题
暴力尝试
改动态规划
机器人走路问题
字符串正则匹配问题
暴力尝试
动态规划
**缓存结构的设计**
设计可以变更的缓存结构（LRU）
LFU
**附录**
手写二叉搜索树
AbstractBinarySearchTree
AbstractSelfBalancingBinarySearchTree
AVLTree
RedBlackTree


# 经典算法

## Manacher算法

### 原始问题

Manacher算法是由题目“求字符串中最长回文子串的长度”而来。比如`abcdcb`的最长回文子串为`bcdcb`，其长度为5。

我们可以遍历字符串中的每个字符，当遍历到某个字符时就比较一下其左边相邻的字符和其右边相邻的字符是否相同，如果相同则继续比较其右边的右边和其左边的左边是否相同，如果相同则继续比较……，我们暂且称这个过程为向外“扩”。当“扩”不动时，经过的所有字符组成的子串就是以当前遍历字符为中心的最长回文子串。

我们每次遍历都能得到一个最长回文子串的长度，使用一个全局变量保存最大的那个，遍历完后就能得到此题的解。但分析这种方法的时间复杂度：当来到第一个字符时，只能扩其本身即1个；来到第二个字符时，最多扩两个；……；来到字符串中间那个字符时，最多扩 `(n-1)/2+1` 个；因此时间复杂度为 `1+2+……+(n-1)/2+1` 即 `O(N^2)` 。但Manacher算法却能做到 `O(N)` 。

Manacher算法中定义了如下几个概念：

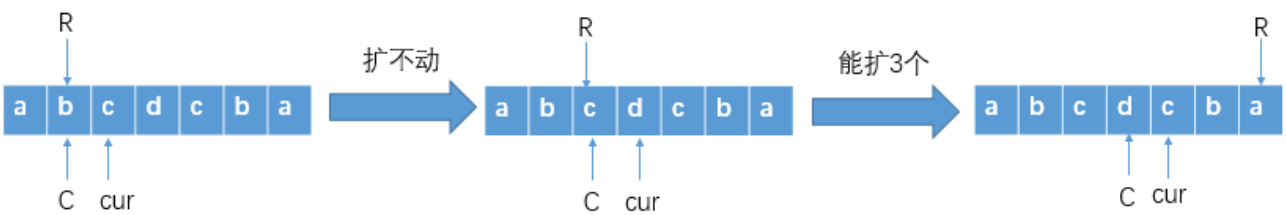
- 回文半径：串中某个字符最多能向外扩的字符个数称为该字符的回文半径。比如 `abcdcb` 中字符 `d` ，能扩一个 `c` ，还能再扩一个 `b` ，再扩就到字符串右边界了，再算上字符本身，字符 `d` 的回文半径是3。
- 回文半径数组 `pArr` ：长度和字符串长度一样，保存串中每个字符的回文半径。比如 `charArr="abcdcb"` ，其中 `charArr[0]='a'` 一个都扩不了，但算上其本身有 `pArr[0]=1` ；而 `charArr[3]='d'` 最多扩2个，算上其本身有 `pArr[3]=3` 。
- 最右回文右边界 `R` ：遍历过程中，“扩”这一操作扩到的最右的字符的下标。比如 `charArr="abcdcb"` ，当遍历到 `a` 时，只能扩 `a` 本身，向外扩不动，所以 `R=0` ；当遍历到 `b` 时，也只能扩 `b` 本身，所以更新 `R=1` ；但当遍历到 `d` 时，能向外扩两个字符到 `charArr[5]=b` ，所以 `R` 更新为5。
- 最右回文右边界对应的回文中心 `C` ： `C` 与 `R` 是对应的、同时更新的。比如 `abcdcb` 遍历到 `d` 时， `R=5` ， `C` 就是 `charArr[3]='d'` 的下标 `3` 。

处理回文子串长度为偶数的问题：上面拿 `abcdcb` 来举例，其中 `bcdcb` 属于一个回文子串，但如果回文子串长度为偶数呢？像 `cabbac` ，按照上面定义的“扩”的逻辑岂不是每个字符的回文半径都是0，但事实上 `cabbac` 的最长回文子串的长度是6。因为我们上面“扩”的逻辑默认是将回文子串当做奇数长度的串来看的，因此我们在使用Manacher算法之前还需要将字符串处理一下，这里有一个小技巧，那就是将字符串的首尾和每个字符之间加上一个特殊符号，这样就能将输入的串统一转为奇数长度的串了。比如 `abba` 处理过后为 `#a#b#b#a` ，这样的话就有 `charArr[4]='#'` 的回文半径为4，也即原串的最大回文子串长度为4。相应代码如下：

```
public static char[] manacherString(String str){
  char[] source = str.toCharArray();
  char chs[] = new char[str.length() * 2 + 1];
  for (int i = 0; i < chs.length; i++) {
    chs[i] = i % 2 == 0 ? '#' : source[i / 2];
  }
  return chs;
}
```

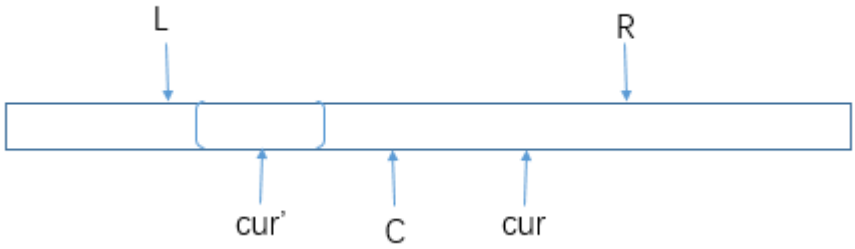
接下来分析，BFPRT算法是如何利用遍历过程中计算的 `pArr` 、 `R` 、 `C` 来为后续字符的回文半径的求解加速的。

首先，情况1是，遍历到的字符下标 `cur` 在 `R` 的右边（起初另 `R=-1` ），这种情况下该字符的最大回文半径 `pArr[cur]` 的求解无法加速，只能一步步向外扩来求解。

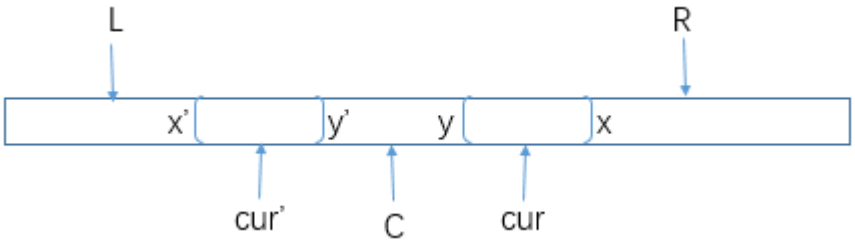



情况2是，遍历到的字符下标 `cur` 在 `R` 的左边，这时 `pArr[cur]` 的求解过程可以利用之前遍历的字符回文半径信息来加速。分别做 `cur` 、 `R` 关于 `C` 的对称点 `cur'` 和 `L` ：

- 如果从 `cur'` 向外扩的最大范围的左边界没有超过 `L` ，那么 `pArr[cur]=pArr[cur']` 。

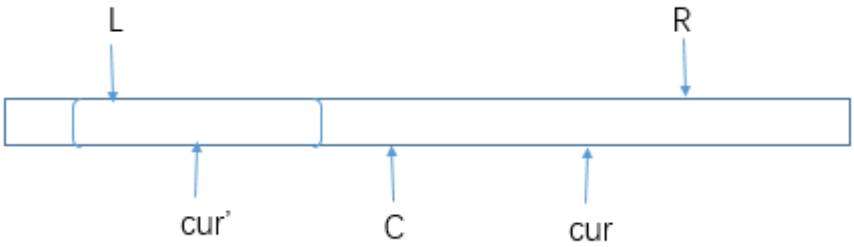



证明如下：

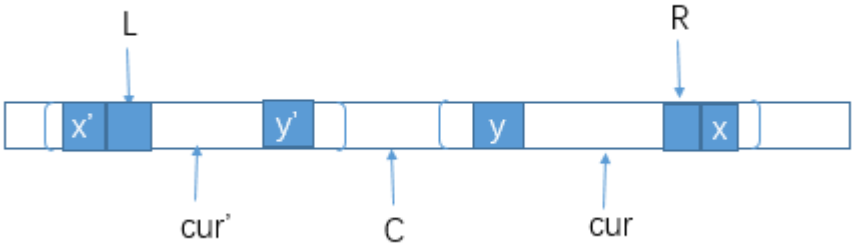



由于之前遍历过`cur'`位置上的字符，所以该位置上能扩的步数我们是有记录的（`pArr[cur']`），也就是说`cur'+pArr[cur']`处的字符`y'`是不等于`cur'-pArr[cur']`处的字符`x'`的。根据`R`和`C`的定义，整个`L`到`R`范围的字符是关于`C`对称的，也就是说`cur`能扩出的最大回文子串和`cur'`能扩出的最大回文子串相同，因此可以直接得出`pArr[cur]=pArr[cur']`。

- 如果从`cur'`向外扩的最大范围的左边界超过了`L`，那么`pArr[cur]=R-cur+1`。

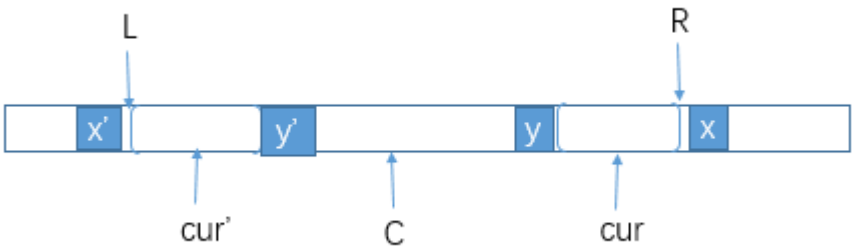



证明如下：




`R`右边一个字符`x`，`x`关于`cur`对称的字符`y`，`x,y`关于`C`对称的字符`x',y'`。根据`C,R`的定义有`x!=x'`；由于`x',y'`在以`cur'`为中心的回文子串内且关于`cur'`对称，所以有`x'=y'`，可推出`x!=y'`；又`y,y'`关于`C`对称，且在`L,R`内，所以有`y=y'`。综上所述，有`x!=y`，因此`cur`的回文半径为`R-cur+1`。

- 以`cur'`为中心向外扩的最大范围的左边界正好是`L`，那么`pArr[cur] >= (R-cur+1)`




这种情况下，`cur'`能扩的范围是`cur'-L`，因此对应有`cur`能扩的范围是`R-cur`。但`cur`能否扩的更大则取决于`x`和`y`是否相等。而我们所能得到的前提条件只有`x!=x'`、`y=y'`、`x'!=y'`，无法推导出`x,y`的关系，只知道`cur`的回文半径最小为`R-cur+1`（算上其本身），需要继续尝试向外扩以求解`pArr[cur]`。

综上所述，`pArr[cur]`的计算有四种情况：暴力扩、等于`pArr[cur']`、等于`R-cur+1`、从`R-cur+1`继续向外扩。使用此算法求解原始问题的过程就是遍历串中的每个字符，每个字符都尝试向外扩到最大并更新`R`（只增不减），每次`R`增加的量就是此次能扩的字符个数，而`R`到达串尾时问题的解就能确定了，因此时间复杂度就是每次扩操作检查的次数总和，也就是`R`的变化范围（`-1~2N`，因为处理串时向串中添加了`N+1`个`#`字符），即`O(1+2N)=O(N)`。

整体代码如下：

```java
public static int maxPalindromeLength(String str) {
  char charArr[] = manacherString(str);
  int pArr[] = new int[charArr.length];
  int R = -1, C = -1;
  int max = Integer.MIN_VALUE;
  for (int i = 0; i < charArr.length; i++) {
    pArr[i] = i > R ? 1 : Math.min(pArr[C * 2 - i], R - i);
    while (i + pArr[i] < charArr.length && i - pArr[i] > -1) {
      if (charArr[i + pArr[i]] == charArr[i - pArr[i]]) {
        pArr[i]++;
      } else {
        break;
      }
    }
    if (R < i + pArr[i]) {
      R = i + pArr[i]-1;
      C = i;
    }
    max = Math.max(max, pArr[i]);
  }
  return max-1;
}

public static void main(String[] args) {
  System.out.println(maxPalindromeLength("zxabcdcbayq"));
}
```

上述代码将四种情况的分支处理浓缩到了`7~14`行。其中第`7`行是确定加速信息：如果当前遍历字符在`R`右边，先算上其本身有`pArr[i]=1`，后面检查如果能扩再直接`pArr[i]++`即可；否则，当前字符的`pArr[i]`要么是`pArr[i']`（`i`关于`C`对称的下标`i'`的推导公式为`2*C-i`），要么是`R-i+1`，要么是`>=R-i+1`，可以先将`pArr[i]`的值置为这三种情况中最小的那一个，后面再检查如果能扩再直接`pArr[i]++`即可。

最后得到的`max`是处理之后的串（`length=2N+1`）的最长回文子串的半径，`max-1`刚好为原串中最长回文子串的长度。

## 进阶问题

给你一个字符串，要求添加尽可能少的字符使其成为一个回文字符串。

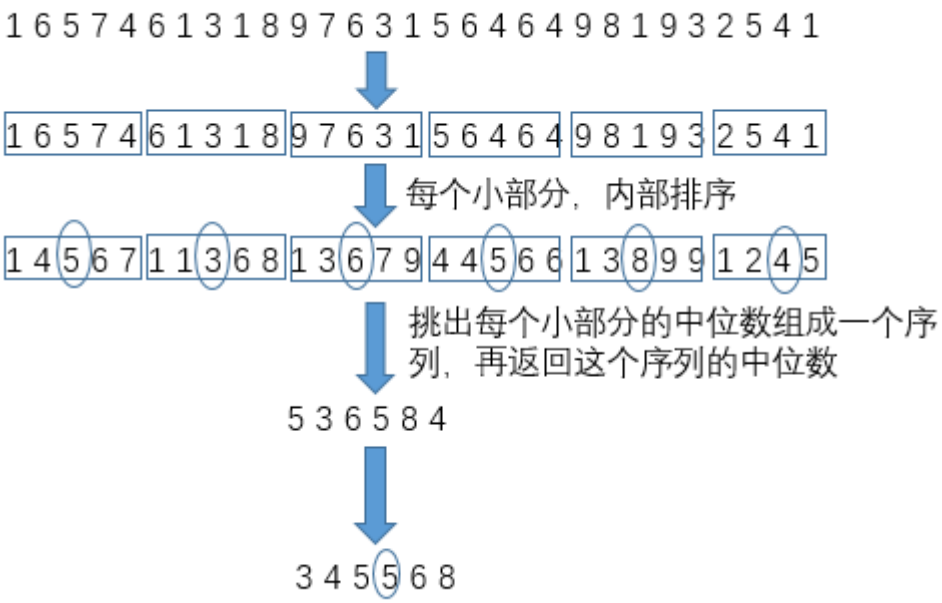
> 思路：当`R`第一次到达串尾时，做`R`关于`C`的对称点`L`，将`L`之前的字符串逆序就是结果。

# BFPRT算法

题目：给你一个整型数组，返回其中第K小的数。

这道题可以利用荷兰国旗改进的`partition`和随机快排的思想：随机选出一个数，将数组以该数作比较划分为`<,=,>`三个部分，则`=`部分的数是数组中第几小的数不难得知，接着对`<`（如果第K小的数在`<`部分）或`>`（如果第K小的数在`>`部分）部分的数递归该过程，直到`=`部分的数正好是整个数组中第K小的数。这种做法不难求得时间复杂度的数学期望为`O(NlogN)`（以2为底）。但这毕竟是数学期望，在实际工程中的表现可能会有偏差，而BFPRT算法能够做到时间复杂度就是`O(NlogN)`。

BFPRT算法首先将数组按5个元素一组划分成`N/5`个小部分（最后不足5个元素自成一个部分），再这些小部分的内部进行排序，然后将每个小部分的中位数取出来再排序得到中位数：




BFPRT求解此题的步骤和开头所说的步骤大体类似，但是“随机选出一个的作为比较的那个数”这一步替换为上图所示最终选出来的那个数。

`O(NlogN)`的证明，为什么每一轮`partition`中的随机选数改为BFPRT定义的选数逻辑之后，此题的时间复杂度就彻底变为`O(NlogN)`了呢？下面分析一下这个算法的步骤：

BFPRT算法，接收一个数组和一个K值，返回数组中的一个数

1. 数组被划分为了`N/5`个小部分，每个部分的5个数排序需要`O(1)`，所有部分排完需要`O(N/5)=O(N)`
2. 取出每个小部分的中位数，一共有`N/5`个，递归调用BFPRT算法得到这些数中第`(N/5)/2`小的数（即这些数的中位数），记为`pivot`
3. 以`pivot`作为比较，将整个数组划分为`<pivot , =pivot , >pivot`三个区域
4. 判断第K小的数在哪个区域，如果在`=`区域则直接返回`pivot`，如果在`<`或`>`区域，则将这个区域的数递归调用BFPRT算法
5. `base case`：在某次递归调用BFPRT算法时发现这个区域只有一个数，那么这个数就是我们要找的数。

代码示例：

```java
public static int getMinKthNum(int[] arr, int K) {
  if (arr == null || K > arr.length) {
    return Integer.MIN_VALUE;
  }
  int[] copyArr = Arrays.copyOf(arr, arr.length);
  return bfprt(copyArr, 0, arr.length - 1, K - 1);
}

public static int bfprt(int[] arr, int begin, int end, int i) {
  if (begin == end) {
    return arr[begin];
  }
  int pivot = medianOfMedians(arr, begin, end);
  int[] pivotRange = partition(arr, begin, end, pivot);
  if (i >= pivotRange[0] && i <= pivotRange[1]) {
    return arr[i];
  } else if (i < pivotRange[0]) {
    return bfprt(arr, begin, pivotRange[0] - 1, i);
  } else {
    return bfprt(arr, pivotRange[1] + 1, end, i);
  }
}

public static int medianOfMedians(int[] arr, int begin, int end) {
  int num = end - begin + 1;
  int offset = num % 5 == 0 ? 0 : 1;
  int[] medians = new int[num / 5 + offset];
  for (int i = 0; i < medians.length; i++) {
    int beginI = begin + i * 5;
    int endI = beginI + 4;
    medians[i] = getMedian(arr, beginI, Math.min(endI, end));
  }
  return bfprt(medians, 0, medians.length - 1, medians.length / 2);
}

public static int getMedian(int[] arr, int begin, int end) {
  insertionSort(arr, begin, end);
  int sum = end + begin;
  int mid = (sum / 2) + (sum % 2);
  return arr[mid];
}
```

```
42
43  public static void insertionSort(int[] arr, int begin, int end) {
44    if (begin >= end) {
45      return;
46    }
47    for (int i = begin + 1; i <= end; i++) {
48      for (int j = i; j > begin; j--) {
49        if (arr[j] < arr[j - 1]) {
50          swap(arr, j, j - 1);
51        } else {
52          break;
53        }
54      }
55    }
56  }
57
58  public static int[] partition(int[] arr, int begin, int end, int pivot) {
59    int L = begin - 1;
60    int R = end + 1;
61    int cur = begin;
62    while (cur != R) {
63      if (arr[cur] > pivot) {
64        swap(arr, cur, --R);
65      } else if (arr[cur] < pivot) {
66        swap(arr, cur++, ++L);
67      } else {
68        cur++;
69      }
70    }
71    return new int[]{L + 1, R - 1};
72  }
73
74  public static void swap(int[] arr, int i, int j) {
75    int tmp = arr[i];
76    arr[i] = arr[j];
77    arr[j] = tmp;
78  }
79
80  public static void main(String[] args) {
81    int[] arr = {6, 9, 1, 3, 1, 2, 2, 5, 6, 1, 3, 5, 9, 7, 2, 5, 6, 1, 9};
82    System.out.println(getMinKthNum(arr,13));
83  }
```

时间复杂度为`O(NlogN)`（底数为2）的证明，分析`bfprt`的执行步骤（假设`bfprt`的时间复杂度为`T(N)`）：

1. 首先数组5个5个一小组并内部排序，对5个数排序为`O(1)`，所有小组排好序为`O(N/5)=O(N)`
2. 由步骤1的每个小组抽出中位数组成一个中位数小组，共有`N/5`个数，递归调用`bfprt`求出这`N/5`个数中第`(N/5)/2`小的数（即中位数）为`T(N/5)`，记为`pivot`
3. 对步骤2求出的`pivot`作为比较将数组分为小于、等于、大于三个区域，由于`pivot`是中位数小组中的中位数，所以中位数小组中有`N/5/2=N/10`个数比`pivot`小，这`N/10`个数分别又是步骤1中某小组的中位数，可推导出至少有`3N/10`个数比`pivot`小，也即最多有`7N/10`个数比`pivot`大。也就是说，大于区域（或小

于）最大包含`7N/10`个数、最少包含`3N/10`个数，那么如果第`i`大的数不在等于区域时，无论是递归`bfprt`处理小于区域还是大于区域，最坏情况下子过程的规模最大为`7N/10`，即`T(7N/10)`

综上所述，`bfprt`的`T(N)`存在推导公式：`T(N/5)+T(7N/10)+O(N)`。根据 **基础篇** 中所介绍的Master公式可以求得`bfprt`的时间复杂度就是`O(NlogN)`（以2为底）。

# morris遍历二叉树

> 关于二叉树先序、中序、后序遍历的递归和非递归版本在【直通BAT算法（基础篇）】中有讲到，但这6种遍历算法的时间复杂度都需要`O(H)`（其中`H`为树高）的额外空间复杂度，因为二叉树遍历过程中只能向下查找孩子节点而无法回溯父结点，因此这些算法借助栈来保存要回溯的父节点（递归的实质是系统帮我们压栈），并且栈要保证至少能容纳下`H`个元素（比如遍历到叶子结点时回溯父节点，要保证其所有父节点在栈中）。而morris遍历则能做到时间复杂度仍为`O(N)`的情况下额外空间复杂度只需`O(1)`。

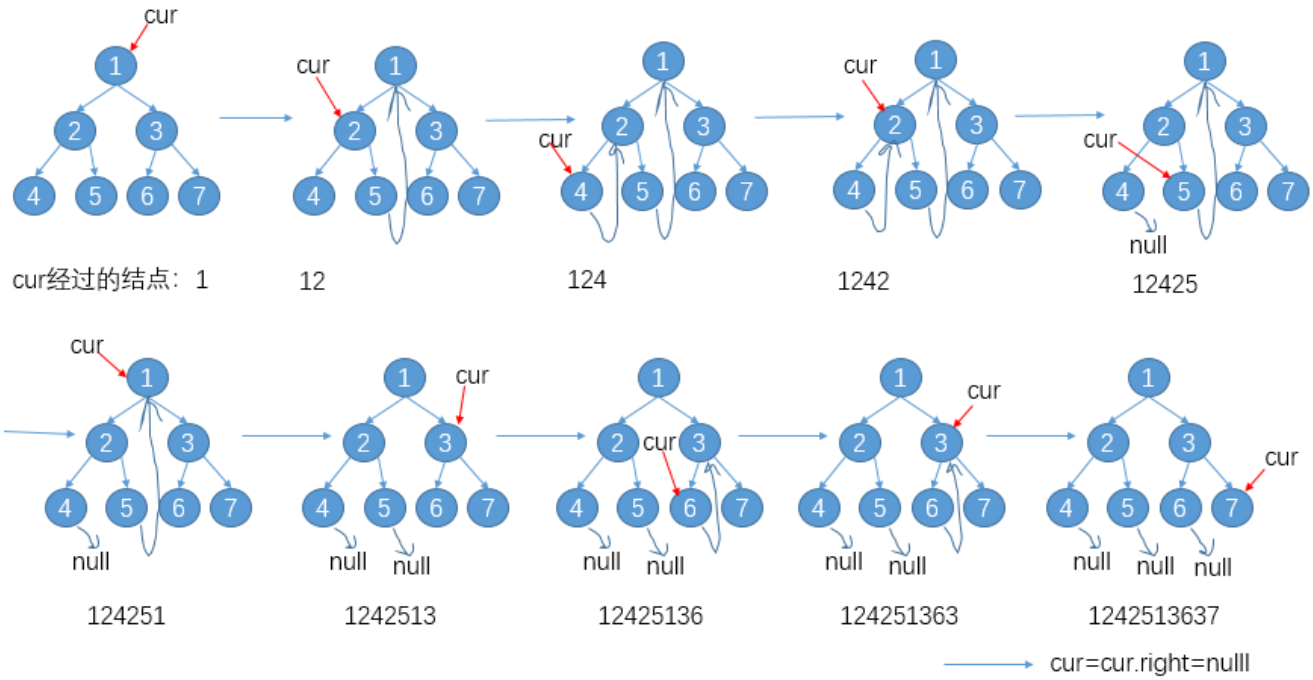
## 遍历规则

首先在介绍morris遍历之前，我们先把先序、中序、后序定义的规则抛之脑后，比如先序遍历在拿到一棵树之后先遍历头结点然后是左子树最后是右子树，并且在遍历过程中对于子树的遍历仍是这样。

忘掉这些遍历规则之后，我们来看一下morris遍历定义的标准：

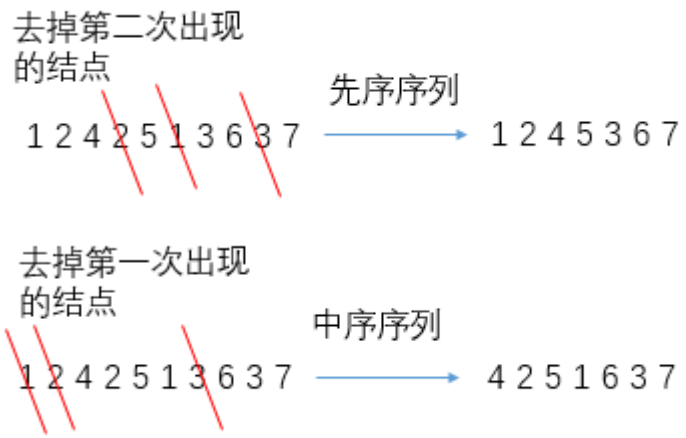
1. 定义一个遍历指针`cur`，该指针首先指向头结点
2. 判断`cur`的左子树是否存在
    - 如果`cur`的左孩子为空，说明`cur`的左子树不存在，那么`cur`右移来到`cur.right`
    - 如果`cur`的左孩子不为空，说明`cur`的左子树存在，找出该左子树的最右结点，记为`mostRight`
        - 如果，`mostRight`的右孩子为空，那就让其指向`cur`（`mostRight.right=cur`），并左移`cur`（`cur=cur.left`）
        - 如果`mostRight`的右孩子不空，那么让`cur`右移（`cur=cur.right`），并将`mostRight`的右孩子置空
3. 经过步骤2之后，如果`cur`不为空，那么继续对`cur`进行步骤2，否则遍历结束。

下图所示举例演示morris遍历的整个过程：




## 先序、中序序列

遍历完成后对`cur`进过的节点序列稍作处理就很容易得到该二叉树的先序、中序序列：




示例代码：

```java
public static class Node {
    int data;
    Node left;
    Node right;
    public Node(int data) {
        this.data = data;
    }
}

public static void preOrderByMorris(Node root) {
    if (root == null) {
        return;
    }
```

```
    Node cur = root;
    while (cur != null) {
        if (cur.left == null) {
            System.out.print(cur.data+" ");
            cur = cur.right;
        } else {
            Node mostRight = cur.left;
            while (mostRight.right != null && mostRight.right != cur) {
                mostRight = mostRight.right;
            }
            if (mostRight.right == null) {
                System.out.print(cur.data+" ");
                mostRight.right = cur;
                cur = cur.left;
            } else {
                cur = cur.right;
                mostRight.right = null;
            }
        }
    }
    System.out.println();
}

public static void mediumOrderByMorris(Node root) {
    if (root == null) {
        return;
    }
    Node cur = root;
    while (cur != null) {
        if (cur.left == null) {
            System.out.print(cur.data+" ");
            cur = cur.right;
        } else {
            Node mostRight = cur.left;
            while (mostRight.right != null && mostRight.right != cur) {
                mostRight = mostRight.right;
            }
            if (mostRight.right == null) {
                mostRight.right = cur;
                cur = cur.left;
            } else {
                System.out.print(cur.data+" ");
                cur = cur.right;
                mostRight.right = null;
            }
        }
    }
    System.out.println();
}

public static void main(String[] args) {
    Node root = new Node(1);
    root.left = new Node(2);
```

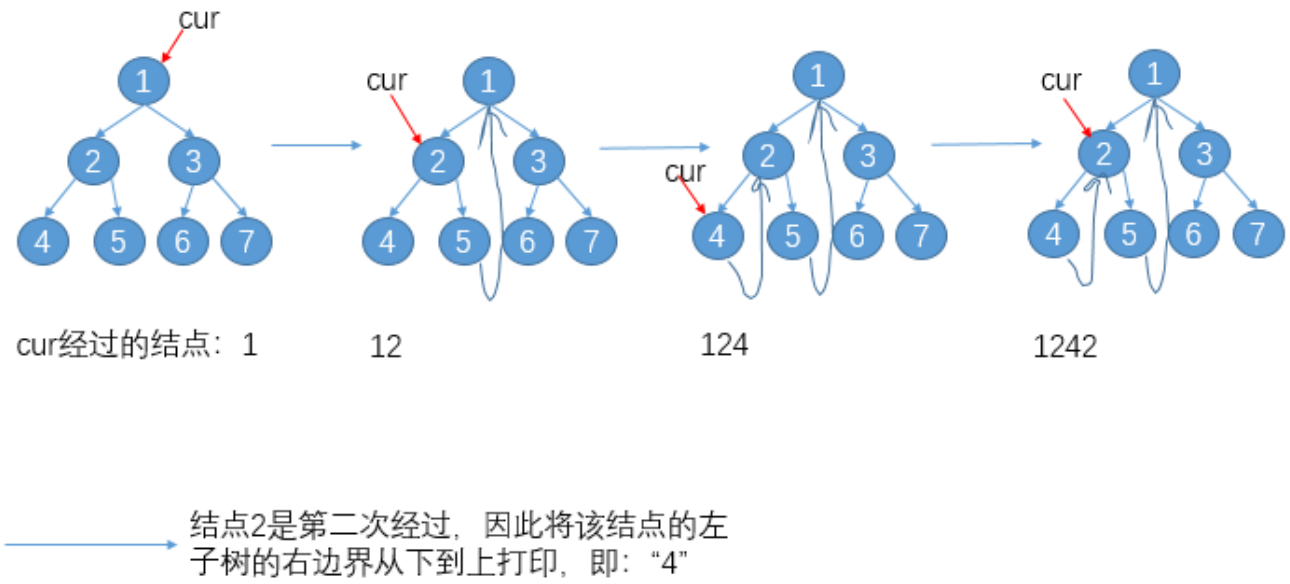
```
67      root.right = new Node(3);
68      root.left.left = new Node(4);
69      root.left.right = new Node(5);
70      root.right.left = new Node(6);
71      root.right.right = new Node(7);
72      preOrderByMorris(root);
73      mediumOrderByMorris(root);
74
75  }
```

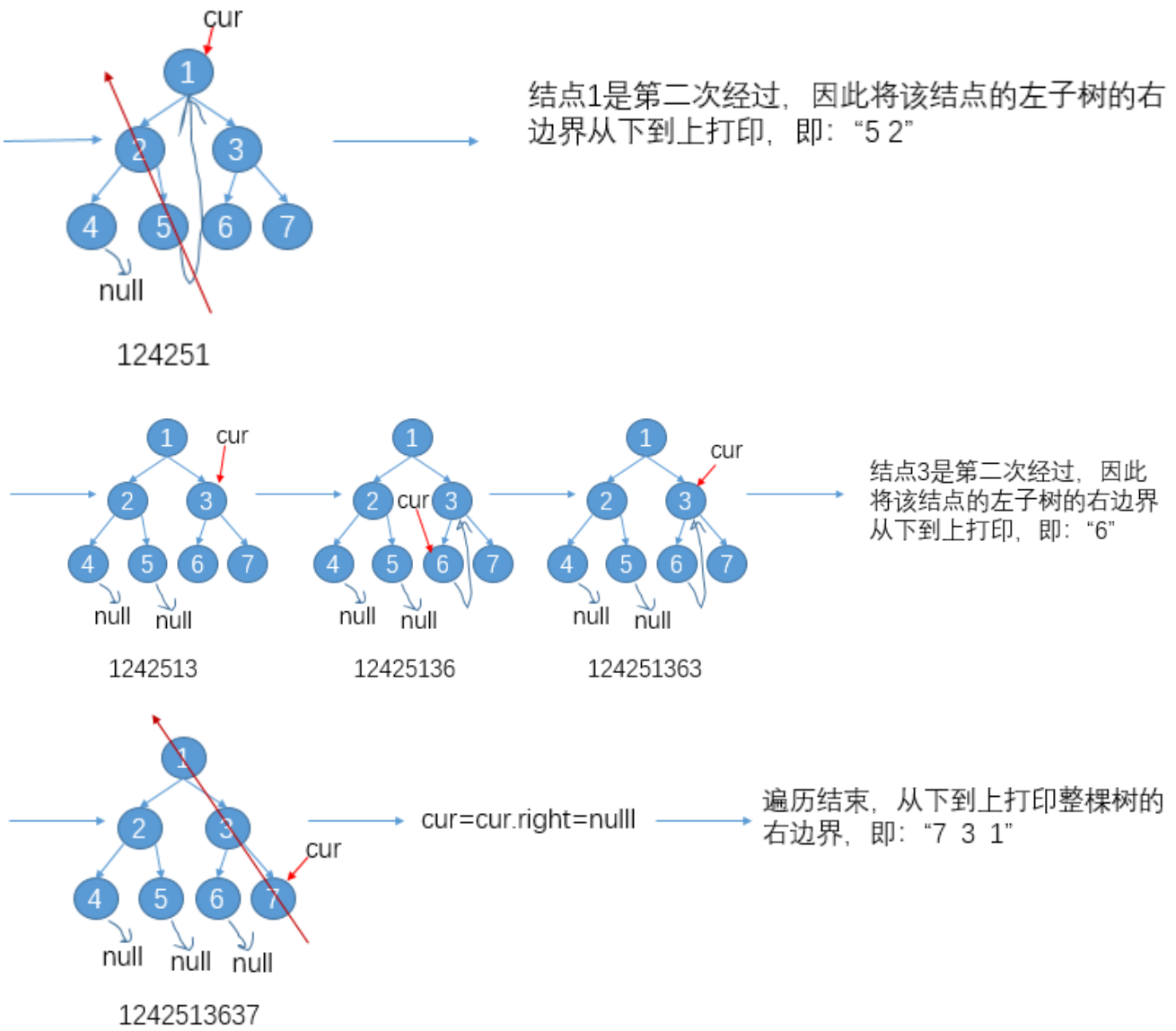
这里值得注意的是：**morris遍历会来到一个左孩子不为空的结点两次**，而其它结点只会经过一次。因此使用morris遍历打印先序序列时，如果来到的结点无左孩子，那么直接打印即可（这种结点只会经过一次），否则如果来到的结点的左子树的最右结点的右孩子为空才打印（这是第一次来到该结点的时机），这样也就忽略了`cur`经过的结点序列中第二次出现的结点；而使用morris遍历打印中序序列时，如果来到的结点无左孩子，那么直接打印（这种结点只会经过一次，左中右，没了左，直接打印中），否则如果来到的结点的左子树的最右结点不为空时才打印（这是第二次来到该结点的时机），这样也就忽略了`cur`经过的结点序列中第一次出现的重复结点。

## 后序序列

使用morris遍历得到二叉树的后序序列就没那么容易了，因为对于树种的非叶结点，morris遍历最多会经过它两次，而我们后序遍历实在第三次来到该结点时打印该结点的。因此要想得到后序序列，仅仅改变在morris遍历时打印结点的时机是无法做到的。

但其实，在morris遍历过程中，如果在每次遇到第二次经过的结点时，将该结点的左子树的右边界上的结点从下到上打印，最后再将整个树的右边界从下到上打印，最终就是这个数的后序序列：





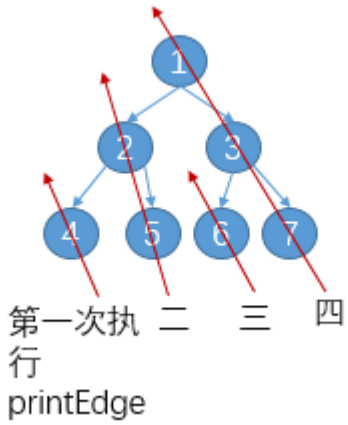

其中无非就是在morris遍历中在第二次经过的结点的时机执行一下打印操作。而从下到上打印一棵树的右边界，可以将该右边界上的结点看做以 `right` 指针为后继指针的链表，将其反转 `reverse` 然后打印，最后恢复成原始结构即可。示例代码如下（其中容易犯错的地方是 `18` 行和 `19` 行的代码不能调换）：

```
1  public static void posOrderByMorris(Node root) {
2      if (root == null) {
3          return;
4      }
5      Node cur = root;
6      while (cur != null) {
7          if (cur.left == null) {
8              cur = cur.right;
9          } else {
10             Node mostRight = cur.left;
11             while (mostRight.right != null && mostRight.right != cur) {
12                 mostRight = mostRight.right;
13             }
14             if (mostRight.right == null) {
15                 mostRight.right = cur;
```

```
                cur = cur.left;
            } else {
                mostRight.right = null;
                printRightEdge(cur.left);
                cur = cur.right;
            }
        }
    }
    printRightEdge(root);
}

private static void printRightEdge(Node root) {
    if (root == null) {
        return;
    }
    //reverse the right edge
    Node cur = root;
    Node pre = null;
    while (cur != null) {
        Node next = cur.right;
        cur.right = pre;
        pre = cur;
        cur = next;
    }
    //print
    cur = pre;
    while (cur != null) {
        System.out.print(cur.data + " ");
        cur = cur.right;
    }
    //recover
    cur = pre;
    pre = null;
    while (cur != null) {
        Node next = cur.right;
        cur.right = pre;
        pre = cur;
        cur = next;
    }
}

public static void main(String[] args) {
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.left = new Node(6);
    root.right.right = new Node(7);
    posOrderByMorris(root);
}
```

## 时间复杂度分析

因为morris遍历中，只有左孩子非空的结点才会经过两次而其它结点只会经过一次，也就是说遍历的次数小于`2N`，因此使用morris遍历得到先序、中序序列的时间复杂度自然也是`O(1)`；但产生后序序列的时间复杂度还要算上`printRightEdge`的时间复杂度，但是你会发现整个遍历的过程中，所有的`printRightEdge`加起来也只是遍历并打印了`N`个结点：




因此时间复杂度仍然为`O(N)`。

> morris遍历结点的顺序不是先序、中序、后序，而是按照自己的一套标准来决定接下来要遍历哪个结点。
>
> morris遍历的独特之处就是充分利用了叶子结点的无效引用（引用指向的是空，但该引用变量仍然占内存），从而实现了`O(1)`的时间复杂度。

# 求和为aim的最长子数组长度

举例：数组`[7,3,2,1,1,7,-6,-1,7]`中，和为`7`的最长子数组长度为4。（子数组：数组中任意个连续的数组成的数组）

大前提：如果我们求出以数组中每个数结尾的所有子数组中和为aim的子数组，那么答案一定就在其中。

规律：对于数组`[i,……,k,k+1,……,j]`，如果要求aim为800，而我们知道从`i`累加到`j`的累加和为2000，那么从`i`开始向后累加，如果累加到`k`时累加和才达到1200，那么`k+1~j`就是整个数组中累加和为800的最长子数组。

步骤：以`[7,3,2,1,1,7,-6,-3,7]`、`aim=7`为例，

- 首先将`(0,-1)`放入`HashMap`中，代表0这个累加和在还没有遍历时就出现了。`->(0,-1)`
- 接着每遍历一个数就将该位置形成的累加和存入`HashMap`，比如`arr[0]=7`，0位置上形成的累加和为前一个位置形成的累加和`0`加上本位置上的`7`，因此将`(7,0)`放入`HashMap`中表示0位置上第一次形成累加和为`7`，然后将该位置上的累加和减去`aim`，即`7-7=0`，找第一次形成累加和为0的位置，即`-1`，因此以下标为0结尾的子数组中和为aim的最长子数组为`0~0`，即`7`一个元素，记最大长度`maxLength=1`。`->(7,0)`
- 接着来到`arr[1]=3`，1位置上形成的累加和为`7+3=10`，`HashMap`中没有`key`为`10`的记录，因此放入`(10,1)`表示1位置上最早形成累加和为10，然后将该位置上的累加和减去`aim`即`10-7=3`，到`HashMap`中找有没有`key`为`3`的记录（有没有哪个位置最早形成累加和为3），发现没有，因此以下标为1结尾的子数组中没有累加和为`aim`的。`->(10,1)`
- 接着来到`arr[2]=2`，2位置上形成的累加和为`10+2=12`，`HashMap`中没有`key`为`12`的记录，因此放入`(12,2)`，`sum-aim=12-7=5`，到`HashMap`中找有没有`key`为`5`的记录，发现没有，因此以下标为2结尾的子数组中没有累加和为`aim`的。`->(12,2)`
- 来到`arr[3]=1`，放入`(13,3)`，`sum-aim=5`，以下标为3结尾的子数组没有累加和为aim的。`->(13,3)`

- 来到`arr[4]=1`，放入`(14,4)`，`sum-aim=7`，发现`HashMap`中有`key=7`的记录`(7,0)`，即在0位置上累加和就能达到7了，因此`1~4`是以下标为4结尾的子数组中累积和为`7`的最长子数组，更新`maxLength=4`。`->(14,4)`
- 来到`arr[5]=7`，放入`(21,5)`，`sum-aim=14`，`HashMap`中有`(14,4)`，因此`5~5`是本轮的最长子数组，但`maxLength=4>1`，因此不更新。`->(21,5)`
- 来到`arr[6]=-6`，放入`15,6`，没有符合的子数组。`->(15,6)`
- 来到`arr[7]=-1`，累加和为`15+(-1)=14`，但`HashMap`中有`key=14`的记录，因此不放入`(14,7)`（`HashMap`中保存的是某累加和第一次出现的位置，而14这个了累加和最早在4下标上就出现了）。`sum-aim=7`，`HashMap`中有`(7,0)`，因此本轮最长子数组为`1~7`，因此更新`maxLength=7`。
- 来到`arr[8]=7`，累加和为21，存在key为21的记录，因此不放入（21，7）。`sum-aim=14`，本轮最长子数组为`5~8`，长度为4，不更新`maxLength`。

示例代码：

```
public static int maxLength(int[] arr,int aim) {
    //key->accumulate sum   value->index
    HashMap<Integer, Integer> hashMap = new HashMap<>();
    hashMap.put(0, -1);
    int curSum = 0;
    int maxLength = 0;
    for (int i = 0; i < arr.length; i++) {
        curSum += arr[i];
        if (!hashMap.containsKey(curSum)) {
            hashMap.put(curSum, i);
        }
        int gap = curSum - aim;
        if (hashMap.containsKey(gap)) {
            int index = hashMap.get(gap);
            maxLength = Math.max(maxLength, i - index);
        }
    }
    return maxLength;
}

public static void main(String[] args) {
    int arr[] = {7, 3, 2, 1, 1, 7, -6, -1, 7};
    int aim = 7;
    System.out.println(maxLength(arr, aim));//7
}
```

# 拓展

## 求奇数个数和偶数个数相同的最长子数组长度

> 将奇数置为1，偶数置为-1，就转化成了求和为0的最长子数组长度

## 求数值为1的个数和数值为2的个数相同的最长子数组（数组只含0、1、2三种元素）

> 将2置为-1，就转化成了求和为0的最长子数组长度

# 进阶

## 求任意划分数组的方案中，划分后，异或和为0的子数组最多有多少个

举例：给你一个数组`[1,2,3,0,2,3,1,0]`，你应该划分为`[1,2,3],[0],[2,3,1],[0]`，答案是4。

**大前提**：如果我们求出了以数组中每个数为结尾的所有子数组中，任意划分后，异或和为0的子数组最多有多少个，那么答案一定就在其中。

**规律**：异或运算符合交换律和结合律。`0^N=N`，`N^N=0`。

**可能性分析**：对于一个数组`[i, ……, j, m, ……, n,k]`，假设进行符合题意的最优划分后形成多个子数组后，k作为整个数组的末尾元素必定也是最后一个子数组的末尾元素。最后一个子数组只会有两种情况：异或和不为0、异或和为0。

- 如果是前者，那么最后一个子数组即使去掉k这个元素，其异或和也不会为0，否则最优划分会将最后一个子数组划分为两个子数组，其中k单独为一个子数组。比如最后一个子数组是`indexOf(m)~indexOf(k)`，其异或和不为0，那么`dp[indexOf(k)]=dp[indexOf(k)-1]`，表示数组`0~indexOf(k)`的解和其子数组`0~(indexOf(k)-1)`的解是一样的。`->case 1`
- 如果是后者，那么最后一个子数组中不可能存在以k为结尾的更小的异或和为0的子数组。比如最后一个子数组是`indexOf(m)~indexOf(k)`，其异或和为0，那么`dp[indexOf(k)]=dp[indexOf(m)-1]+1`，表示数组`0~indexOf(k)`的解=子数组`0~(indexOf(m)-1)`的解+1。`->case 2`

示例代码：

```
1  public static int maxSubArrs(int[] arr) {
2      if (arr == null) {
3          return 0;
4      }
5      HashMap<Integer, Integer> map = new HashMap();
6      map.put(0, -1);
7      int curXorSum = 0;
8      int res = 0;
9      int[] dp = new int[arr.length];
10     for (int i = 0; i < arr.length; i++) {
11         curXorSum ^= arr[i];
12         //case 1，之前没有出现过这个异或和，那么该位置上的dp等于前一个位置的dp
13         if (!map.containsKey(curXorSum)) {
14             dp[i] = i > 0 ? dp[i - 1] : 0;
15         } else {
16             //case 2，之前出现过这个异或和，那么之前这个异或和出现的位置到当前位置形成的子数组异
   或和为0
17             int index = map.get(curXorSum);
18             dp[i] = index > 0 ? dp[index] + 1 : 1;
19         }
20         //把最近出现的异或和都记录下来，因为要划分出最多的异或和为0的子数组
21         map.put(curXorSum, i);
22     }
23     //最后一个位置的dp就是整个问题的解
24     return dp[dp.length -1];
25 }
26
27 public static void main(String[] args) {
```

```
    int arr[] = {1, 2, 3, 0, 2, 3, 1, 0,4,1,3,2};
    System.out.println(maxSubArrs(arr));
}
```

# 高度套路的二叉树信息收集问题

## 求一棵二叉树的最大搜索二叉子树的结点个数

> 最大搜索二叉子树指该二叉树的子树中，是搜索二叉树且结点个数最多的。

这类题一般都有一个**大前提**：**假设对于以树中的任意结点为头结点的子树，我们都能求得其最大搜索二叉子树的结点个数，那么答案一定就在其中。**

而对于以任意结点为头结点的子树，其最大搜索二叉子树的求解分为三种情况（**列出可能性**）：

- 整棵树的最大搜索二叉子树存在于左子树中。这要求其左子树中存在最大搜索二叉子树，而其右子树不存在。
- 整棵树的最大搜索二叉子树存在于右子树中。这要求其右子树中存在最大搜索二叉子树，而其左子树不存在。
- 最整棵二叉树的最大搜索二叉子树就是其本身。这需要其左子树就是一棵搜索二叉子树且左子树的最大值结点比头结点小、其右子树就是一棵搜索二叉子树且右子树的最小值结点比头结点大。

要想区分这三种情况，我们需要收集的信息：

- 子树中是否存在最大搜索二叉树
- 子树的头结点
- 子树的最大值结点
- 子树的最小值结点

因此我们就可以开始我们的高度套路了：

1. 将要从子树收集的信息封装成一个`ReturnData`，代表处理完这一棵子树要向上级返回的信息。
2. 假设我利用子过程收集到了子树的信息，接下来根据子树的信息和分析问题时列出的情况加工出当前这棵树要为上级提供的所有信息，并返回给上级（**整合信息**）。
3. 确定`base case`，子过程到子树为空时，停。

根据上面高度套路的分析，可以写出解决这类问题高度相似的代码：

```
public static class Node{
    int data;
    Node left;
    Node right;
    public Node(int data) {
        this.data = data;
    }
}

public static class ReturnData {
    int size;
    Node head;
    int max;
```

```
    int min;
    public ReturnData(int size, Node head, int max, int min) {
        this.size = size;
        this.head = head;
        this.max = max;
        this.min = min;
    }
}

public static ReturnData process(Node root) {
    if (root == null) {
        return new ReturnData(0, null, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    ReturnData leftInfo = process(root.left);
    ReturnData rightInfo = process(root.right);

    //case 1
    int leftSize = leftInfo.size;
    //case 2
    int rightSize = rightInfo.size;
    int selfSize = 0;
    if (leftInfo.head == root.left && rightInfo.head == root.right
        && leftInfo.max < root.data && rightInfo.min > root.data) {
        //case 3
        selfSize = leftInfo.size + rightInfo.size + 1;
    }
    int maxSize = Math.max(Math.max(leftSize, rightSize), selfSize);
    Node maxHead = leftSize > rightSize ? leftInfo.head :
                    selfSize > rightSize ? root : rightInfo.head;

    return new ReturnData(maxSize, maxHead,
                          Math.max(Math.max(leftInfo.max, rightInfo.max),
root.data),
                          Math.min(Math.min(leftInfo.min, rightInfo.min),
root.data));
}

public static void main(String[] args) {
    Node root = new Node(0);
    root.left = new Node(5);
    root.right = new Node(1);
    root.left.left = new Node(3);
    root.left.left.left = new Node(2);
    root.left.left.right = new Node(4);
    System.out.println(process(root).size);//4
}
```

## 求一棵二叉树的最远距离

如果在二叉树中，小明从结点A出发，既可以往上走到达它的父结点，又可以往下走到达它的子结点，那么小明从结点A走到结点B最少要经过的结点个数（包括A和B）叫做A到B的距离，任意两结点所形成的距离中，最大的叫做树的最大距离。

**高度套路化**：

大前提：如果对于以该树的任意结点作为头结点的子树中，如果我们能够求得所有这些子树的最大距离，那么答案就在其中。

对于该树的任意子树，其最大距离的求解分为以下三种情况：

- 该树的最大距离是左子树的最大距离。
- 该树的最大距离是右子树的最大距离。
- 该树的最大距离是从左子树的最深的那个结点经过该树的头结点走到右子树的最深的那个结点。

要从子树收集的信息：

- 子树的最大距离
- 子树的深度

示例代码：

```
public static class Node{
    int data;
    Node left;
    Node right;
    public Node(int data) {
        this.data = data;
    }
}

public static class ReturnData{
    int maxDistance;
    int height;
    public ReturnData(int maxDistance, int height) {
        this.maxDistance = maxDistance;
        this.height = height;
    }
}

public static ReturnData process(Node root){
    if (root == null) {
        return new ReturnData(0, 0);
    }
    ReturnData leftInfo = process(root.left);
    ReturnData rightInfo = process(root.right);

    //case 1
    int leftMaxDistance = leftInfo.maxDistance;
    //case 2
    int rightMaxDistance = rightInfo.maxDistance;
    //case 3
    int includeHeadDistance = leftInfo.height + 1 + rightInfo.height;

```

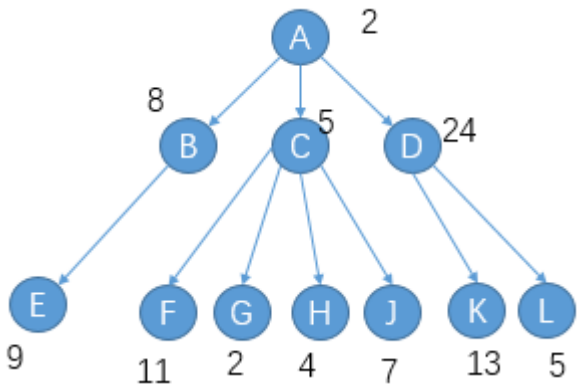
```
    int max = Math.max(Math.max(leftMaxDistance, rightMaxDistance),
includeHeadDistance);
    return new ReturnData(max, Math.max(leftInfo.height, rightInfo.height) + 1);
}

public static void main(String[] args) {
    Node root = new Node(0);
    root.left = new Node(5);
    root.right = new Node(1);
    root.right.right = new Node(6);
    root.left.left = new Node(3);
    root.left.left.left = new Node(2);
    root.left.left.right = new Node(4);
    System.out.println(process(root).maxDistance);
}
```

> 高度套路化：列出可能性->从子过程收集的信息中整合出本过程要返回的信息->返回

## 舞会最大活跃度

一个公司的上下级关系是一棵多叉树，这个公司要举办晚会，你作为组织者已经摸清了大家的心理：**一个员工的直接上级如果到场，这个员工肯定不会来。**每个员工都有一个活跃度的值（值越大，晚会上越活跃），**你可以给某个员工发邀请函以决定谁来**，怎么让舞会的气氛最活跃？返回最大的活跃值。

举例：




如果邀请A来，那么其直接下属BCD一定不会来，你可以邀请EFGHJKL中的任意几个来，如果都邀请，那么舞会最大活跃度为`A(2)+E(9)+F(11)+G(2)+H(4)+J(7)+K(13)+L(5)`；但如果选择不邀请A来，那么你可以邀请其直接下属BCD中任意几个来，比如邀请B而不邀请CD，那么B的直接下属E一定不回来，但CD的直接下属你可以选择性邀请。

**大前提**：如果你知道每个员工来舞会或不来舞会对舞会活跃值的影响，那么舞会最大活跃值就容易得知了。比如是否邀请A来取决于：B来或不来两种情况中选择对舞会活跃值增益最大的那个+C来或不来两种情况中选择对舞会活跃值增益最大的那个+D来或不来两种情况中选择对舞会活跃值增益最大的那个；同理，对于任意一名员工，是否邀请他来都是用此种决策。

**列出可能性**：来或不来。

**子过程要收集的信息**：返回子员工来对舞会活跃值的增益值和不来对舞会的增益值中的较大值。

示例代码：

```java
public static class Node{
    int happy;
    List<Node> subs;
    public Node(int happy) {
        this.happy = happy;
        this.subs = new ArrayList<>();
    }
}

public static class ReturnData {
    int maxHappy;
    public ReturnData(int maxHappy) {
        this.maxHappy = maxHappy;
    }
}

public static ReturnData process(Node root) {
    if (root.subs.size() == 0) {
        return new ReturnData(root.happy);
    }
    //case 1:go
    int go_Happy = root.happy;
    //case 2:don't go
    int unGo_Happy = 0;
    for (Node sub : root.subs) {
        unGo_Happy += process(sub).maxHappy;
    }
    return new ReturnData(Math.max(go_Happy, unGo_Happy));
}

public static int maxPartyHappy(Node root) {
    if (root == null) {
        return 0;
    }
    return process(root).maxHappy;
}

public static void main(String[] args) {
    Node A = new Node(2);
    Node B = new Node(8);
    Node C = new Node(5);
    Node D = new Node(24);
    B.subs.add(new Node(9));
    C.subs.addAll(Arrays.asList(new Node(11),new Node(2),new Node(4),new Node(7)));
    D.subs.addAll(Arrays.asList(new Node(13), new Node(5)));
    A.subs.addAll(Arrays.asList(B, C, D));
    System.out.println(maxPartyHappy(A));//57
}
```

# 求一个数学表达式的值

给定一个字符串str，str表示一个公式，公式里可能有整数、加减乘除符号和左右括号，返回公式的计算结果。

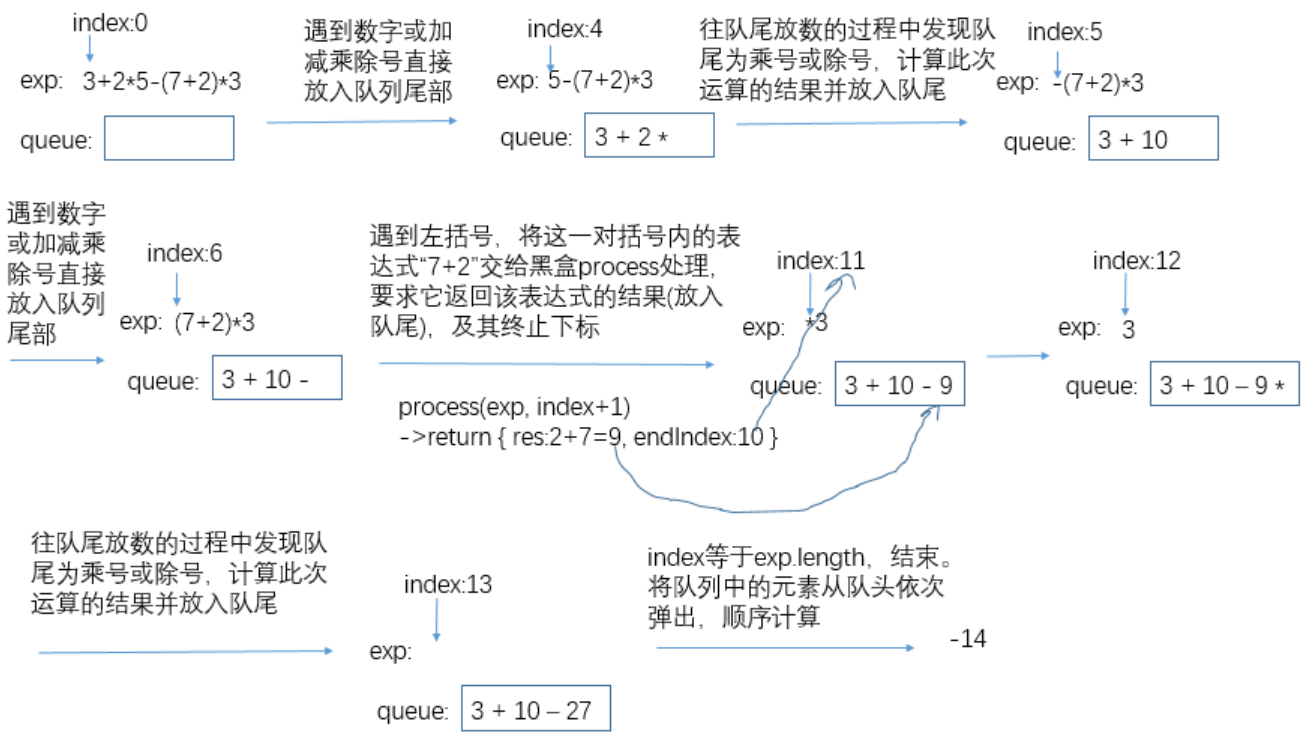
举例：`str="48*((70-65)-43)+8*1"`，返回-1816。`str="3+1*4"`，返回7。 `str="3+(1*4)"`，返回7。

说明：

1. 可以认为给定的字符串一定是正确的公式，即不需要对str做公式有效性检查。
2. 如果是负数，就需要用括号括起来，比如`"4*(-3)"`。但如果负数作为公式的开头或括号部分的开头，则可以没有括号，比如`"-3*4"`和`"(-3*4)"`都是合法的。
3. 不用考虑计算过程中会发生溢出的情况

最优解分析：此题的难度在于如何处理表达式中的括号，可以借助一个栈。但如果仅仅靠一个栈，代码量会显得纷多繁杂。如果我们将式中包含左右括号的子表达式的计算单独抽出来作为一个过程（记为`process`），那么该过程可以被复用，如果我们将整个表达式中所有包含左右括号的子表达式当做一个数值，那么原始问题就转化为计算不含括号的表达式了。

以表达式`3+2*5-(7+2)*3`为例分析解题步骤：




示例代码：

```java
public static int getValue(String exp){
    return process(exp.toCharArray(), 0)[0];
}

/**
     * @param exp   expression
     * @param index the start index of expression
     * @return int[], include two elements:the result and the endIndex
     */
public static int[] process(char[] exp, int index) {

    LinkedList que = new LinkedList();
    //下一个要往队尾放的数
```

```
    int num = 0;
    //黑盒process返回的结果
    int sub[];

    while (index < exp.length && exp[index] != ')') {

        if (exp[index] >= '0' && exp[index] <= '9') {
            num = num * 10 + exp[index] - '0';
            index++;
        } else if (exp[index] != '(') {
            // +、-、*、/
            addNum(num, que);
            num = 0;
            que.addLast(String.valueOf(exp[index]));
            index++;
        } else {
            // '('
            sub = process(exp, index + 1);
            num = sub[0];
            index = sub[1] + 1;
        }
    }

    addNum(num, que);

    return new int[]{getSum(que), index};
}

private static int getSum(LinkedList<String> que) {
    int res = 0;
    boolean add = true;
    while (!que.isEmpty()) {
        int num = Integer.valueOf(que.pollFirst());
        res += add ? num : -num;
        if (!que.isEmpty()) {
            add = que.pollFirst().equals("+") ? true : false;
        }
    }
    return res;
}

private static void addNum(int num, LinkedList<String> que) {
    if (!que.isEmpty()) {
        String element = que.pollLast();
        if (element.equals("+") || element.equals("-")) {
            que.addLast(element);
        } else{
            // * or /
            Integer preNum = Integer.valueOf(que.pollLast());
            num = element.equals("*") ? (preNum * num) : (preNum / num);
        }
    }
    que.addLast(String.valueOf(num));
```

```
}

public static void main(String[] args) {
    String exp = "48*((70-65)-43)+8*1";
    System.out.println(getValue(exp));
    System.out.println(-48*38+8);
}
```

# 求异或和最大的子数组

给你一个数组，让你找出所有子数组的异或和中，最大的是多少。

## 暴力解

遍历数组中的每个数，求出以该数结尾所有子数组的异或和。

```
public static class NumTrie{
    TrieNode root;

    public NumTrie() {
        root = new TrieNode();
    }

    class TrieNode{
        TrieNode[] nexts;
        public TrieNode(){
            nexts = new TrieNode[2];
        }
    }

    public void addNum(int num) {
        TrieNode cur = root;
        for (int i = 31; i >= 0; i--) {
            int path = (num >> i) & 1;
            if (cur.nexts[path] == null) {
                cur.nexts[path] = new TrieNode();
            }
            cur = cur.nexts[path];
        }
    }

    /**
     * find the max value of xor(0,k-1)^xor(0,i)-> the max value of xor(k,i)
     * @param num -> xor(0,i)
     * @return
     */
    public int maxXor(int num) {
        TrieNode cur = root;
        int res = 0;
        for (int i = 31; i >= 0; i--) {
            int path = (num >> i) & 1;
            //如果是符号位，那么尽量和它相同（这样异或出来就是正数），如果是数值位那么尽量和它相反
```

```
            int bestPath = i == 31 ? path : (path ^ 1);
            //如果贪心路径不存在，就只能走另一条路
            bestPath = cur.nexts[bestPath] != null ? bestPath : (bestPath ^ 1);
            //记录该位上异或的结果
            res |= (bestPath ^ path) << i;

            cur = cur.nexts[bestPath];
        }
        return res;
    }
}

public static int maxXorSubArray(int arr[]) {
    int maxXorSum = Integer.MIN_VALUE;
    NumTrie numTrie = new NumTrie();
    //没有数时异或和为0，这个也要加到前缀数中，否则第一次到前缀树找bestPath会报空指针
    numTrie.addNum(0);
    int xorZeroToI = 0;
    for (int i = 0; i < arr.length; i++) {
        xorZeroToI ^= arr[i];
        maxXorSum = Math.max(maxXorSum, numTrie.maxXor(xorZeroToI));
        numTrie.addNum(xorZeroToI);
    }
    return maxXorSum;
}


public static void main(String[] args) {
    int[] arr = {1, 2, 3, 4, 1, 2, -7};
    System.out.println(maxXorSubArray(arr));
}
```

时间复杂度为`O(N^3)`

## 优化暴力解

观察暴力解，以`{1, 2, 3, 4, 1, 2, 0}`为例，当我计算以`4`结尾的所有子数组的异或和时，我会先计算子数组`{4}`的，然后计算`{3,4}`的，然后计算`{2,3,4}`的，也就是说每次都是从头异或到尾，之前的计算的结果并没有为之后的计算过程加速。于是，我想着，当我计算`{3,4}`的时候，将`3^4`的结果临时保存一下，在下次的`{2,3,4}`的计算时复用一下，再保存一下`2^3^4`的结果，在下次的`{1,2,3,4}`的计算又可以复用一下。于是暴力解就被优化成了下面这个样子：

```
public static int solution2(int[] arr) {
    int res = 0;
    int temp=0;
    for (int i = 0; i < arr.length; i++) {
        //以i结尾的最大异或和
        int maxXorSum = 0;
        for (int j = i; j >= 0; j--) {
            temp ^= arr[j];
            maxXorSum = Math.max(maxXorSum, temp);
        }
```

```
            //整体的最大异或和
            res = Math.max(res, maxXorSum);
        }
        return res;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 1, 2, 0};
        System.out.println(solution2(arr));//7
    }
```

这时时间复杂度降为了 `O(N^2)`

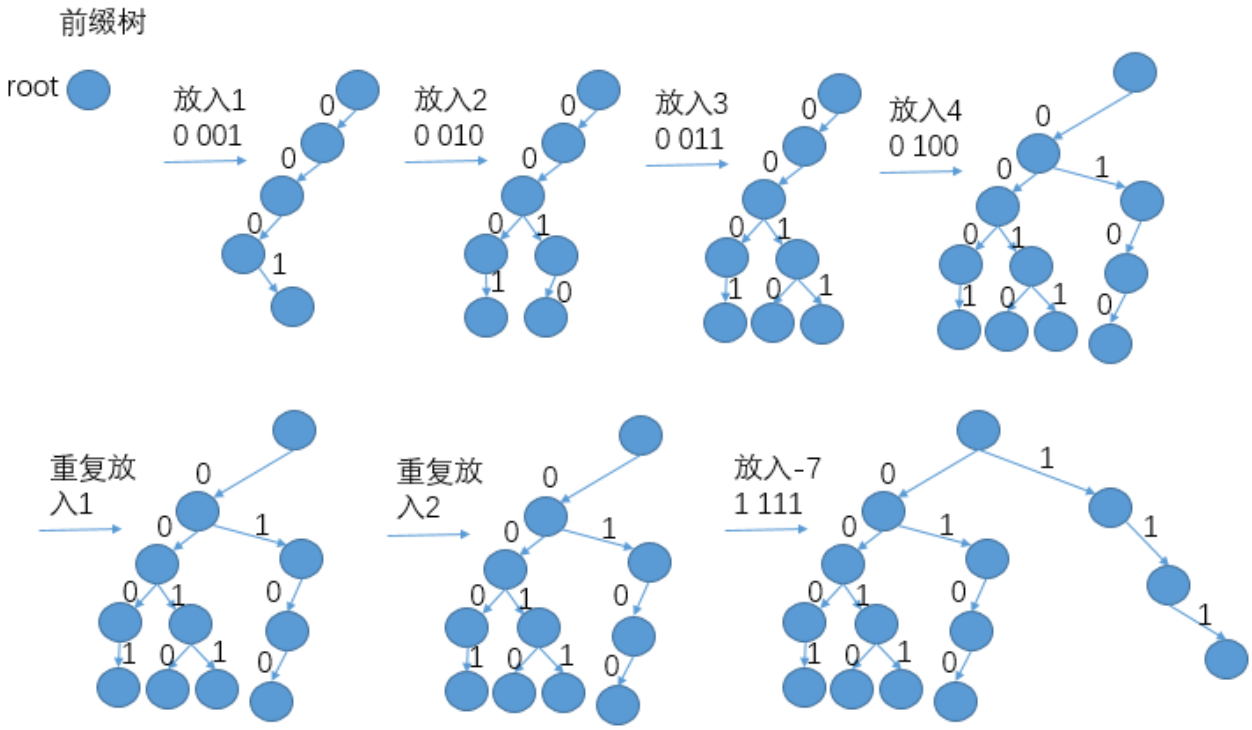
## 最优解

然而使用前缀树结构能够做到时间复杂度 `O(N)`。

解题思路：将以 `i` 结尾的所有子数组的最大异或和的求解限制在 `O(1)`。

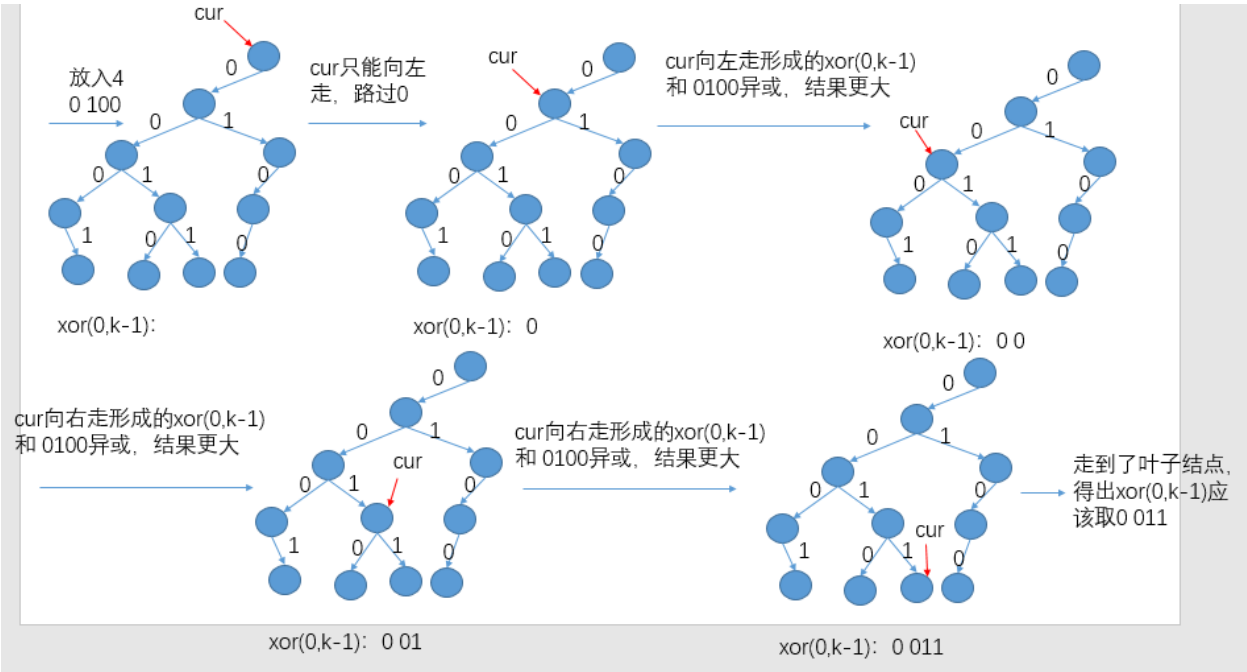
解题技巧：

1. 对于子数组 `0~i`（i是合法下标）和 `0~i` 之间的下标 `k`(k大于等于0，小于等于i)，`k~i` 的异或和 `xor(k,i)`、`0~i` 的异或和 `xor(0,i)`、`0~k-1` 之间的异或和 `xor(0,k-1)` 三者之间存在如下关系：`xor(k,i)=xor(0,i) ^ xor(o,k-1)`（`A^B=C -> B=C^A`），因此求 `xor(k,i)` 的最大值可以转化成求 `xor(0,i) ^ xor(o,k-1)` 的最大值（**这个思路很重要**，后续步骤就是根据这个来的）。
2. 遍历数组，将以首元素开头，以当前遍历元素结尾的子数组的异或和的32位二进制数放入前缀树结构中（每一位作为一个字符，且字符非0即1）。遍历结束后，所有 `0~i` 的异或和就存放在前缀树中了。比如：遍历 `{1, 2, 3, 4, 1, 2, 0}` 形成的前缀树如下：



3. 假设遍历数组建立前缀树的过程中，遍历到 `4` 这个数来了，将 `0 100` 放入其中，由于之前还遍历过 `1,2,3`，所以 `xor(0,0)`、`xor(0,1)`、`xor(0,2)` 也是在前缀树中的。如果此时要求 `xor(k,3)` 的最大值（k在下标0和3之间且包括0和3），可以将其转化为求 `xor(0,3) ^ xor(0,k-1)`，而我们已知 `xor(0,3)=0 100`，所以 `xor(0,k-1)` 的求解就变成了关键。

   `xor(0,k-1)` 的求解：此时游标 `cur` 从前缀树的根结点走向叶子结点，`cur` 沿途经过的二进制位连在一起就是 `xor(0,k-1)`，要求每次选择要经过哪个二进制位时，尽可能使之与 `xor(0,3)` 的异或结果更大：




这个求解过程就是在**贪心**（如果是符号位，那么尽可能让异或结果为0，如果是数值位，那么尽可能让异或结果为1），前缀树里只放着 `xor(0,0)、xor(0,1)、xor(0,2)、xor(0,3)`，而 `xor(0,k-1)` 只能从中取值，这个从根节点一步步试探走到叶子结点的过程就是在贪，哪一条路径对应的 `xor` 使得 `xor ^ xor(0,3)` 最大。

示例代码：

```java
public static class NumTrie{
    TrieNode root;

    public NumTrie() {
        root = new TrieNode();
    }

    class TrieNode{
        TrieNode[] nexts;
        public TrieNode(){
            nexts = new TrieNode[2];
        }
    }

    public void addNum(int num) {
        TrieNode cur = root;
        for (int i = 31; i >= 0; i--) {
            int path = (num >> i) & 1;
            if (cur.nexts[path] == null) {
```

```
                cur.nexts[path] = new TrieNode();
            }
            cur = cur.nexts[path];
        }
    }

    /**
     * find the max value of xor(0,k-1)^xor(0,i)-> the max value of xor(k,i)
     * @param num -> xor(0,i)
     * @return
     */
    public int maxXor(int num) {
        TrieNode cur = root;
        int res = 0;
        for (int i = 31; i >= 0; i--) {
            int path = (num >> i) & 1;
            //如果是符号位，那么尽量和它相同（这样异或出来就是正数），如果是数值位那么尽量和它相反
            int bestPath = i == 31 ? path : (path ^ 1);
            //如果贪心路径不存在，就只能走另一条路
            bestPath = cur.nexts[bestPath] != null ? bestPath : (bestPath ^ 1);
            //记录该位上异或的结果
            res |= (bestPath ^ path) << i;

            cur = cur.nexts[bestPath];
        }
        return res;
    }
}

public static int maxXorSubArray(int arr[]) {
    int maxXorSum = 0;
    NumTrie numTrie = new NumTrie();
    //一个数自己异或自己异或和为0，这个也要加到前缀数中，否则第一次到前缀树找bestPath会报空指针
    numTrie.addNum(0);
    int xorZeroToI = 0;
    for (int i = 0; i < arr.length; i++) {
        xorZeroToI ^= arr[i];
        maxXorSum = Math.max(maxXorSum, numTrie.maxXor(xorZeroToI));
        numTrie.addNum(xorZeroToI);
    }
    return maxXorSum;
}


public static void main(String[] args) {
    int[] arr = {1, 2, 3, 4, 1, 2, -7};
    System.out.println(maxXorSubArray(arr));//7
}
```

# 求和为aim的最长子数组（都大于0）

基础篇中有过相同的题，只不过这里的数组元素值为正数，而基础篇中的可正可负可0。

基础篇中的做法是用一个哈希表记录子数组和出现的最早的位置。而此题由于数据特殊性（都是正数）可以在额外空间复杂度`O(1)`，时间复杂度`O(N)`内完成。

使用一个窗口，用L表示窗口的左边界、R表示窗口的右边界，用sum表示窗口内元素之和（初始为0）。起初，L和R都停在-1位置上，接下来每次都要将L向右扩一步或将R向右扩一步，具体扩哪个视情况而定：

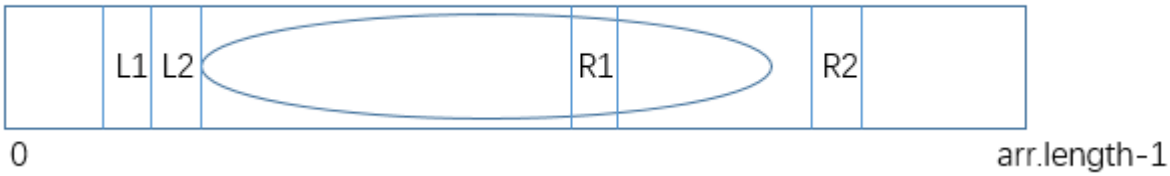
- 如果`sum<aim`，那么R往右边扩
- 如果`sum=aim`，那么记录窗口内元素个数，L往右边扩
- 如果`sum>aim`，那么L往右边扩

直到R扩到`arr.length`越界，那么此时窗口内元素之和必定小于aim，整个过程可以结束。答案就是所有`sum=aim`情况下窗口内元素最多时的个数。

示例代码：

```
/**
     * 数组元素均为正数，求和为aim的最长子数组的长度
     * @param arr
     * @return
     */
public static int aimMaxSubArray(int arr[],int aim) {
    int L=-1;
    int R= -1;
    int sum = 0;
    int len=0;
    while (R != arr.length) {
        if (sum < aim) {
            R++;
            if (R < arr.length) {
                sum += arr[R];
            } else {
                break;
            }
        } else if (sum == aim) {
            len = Math.max(len, R - L);
            sum -= arr[++L];
        } else {
            sum -= arr[++L];
        }
    }
    return len;
}

public static void main(String[] args) {
    int arr[] = {1, 2, 3, 5, 1, 1, 1, 1, 1, 1, 9};
    System.out.println(aimMaxSubArray(arr,6));
}
```

> 思考：为什么这个流程得到的答案是正确的呢？也就是说，为什么窗口向右滑动的过程中，不会错过和为aim的最长子数组？我们可以来证明一下：




假设，椭圆区域就是和为aim的最长子数组，如果L来到了椭圆区域的左边界L2，那么R的位置有两种情况：在椭圆区域内比如R1，在椭圆区域外比如R2。如果是前者，由于窗口 `L2~R1` 是肯定小于 `aim` 的（元素都是正数），因此在R从R1右移到椭圆区域右边界过程中，L是始终在L2上的，显然不会错过正确答案；如果是后者，窗口 `L2~R2` 的 `sum` 明显超过了 `aim`，因此这种情况是不可能存在的。而L在L2左边的位置上，比如L1时，R更不可能越过椭圆区域来到了R2，因为窗口是始终保持 `sum<=aim` 的。

# 求和小于等于aim的最长子数组（有正有负有0）

如果使用暴力枚举，枚举出以每个元素开头的子数组，那么答案一定就在其中（ `O(N^3)` ）。但这里介绍一种时间复杂度 `O(N)` 的解。

首先从尾到头遍历一遍数组，生成两个辅助数组 `min_sum` 和 `min_sum_index` 作为求解时的辅助信息。 `min_sum` 表示以某个元素开头的所有子数组中和最小为多少， `min_sum_index` 则对应保存该最小和子数组的结束下标。

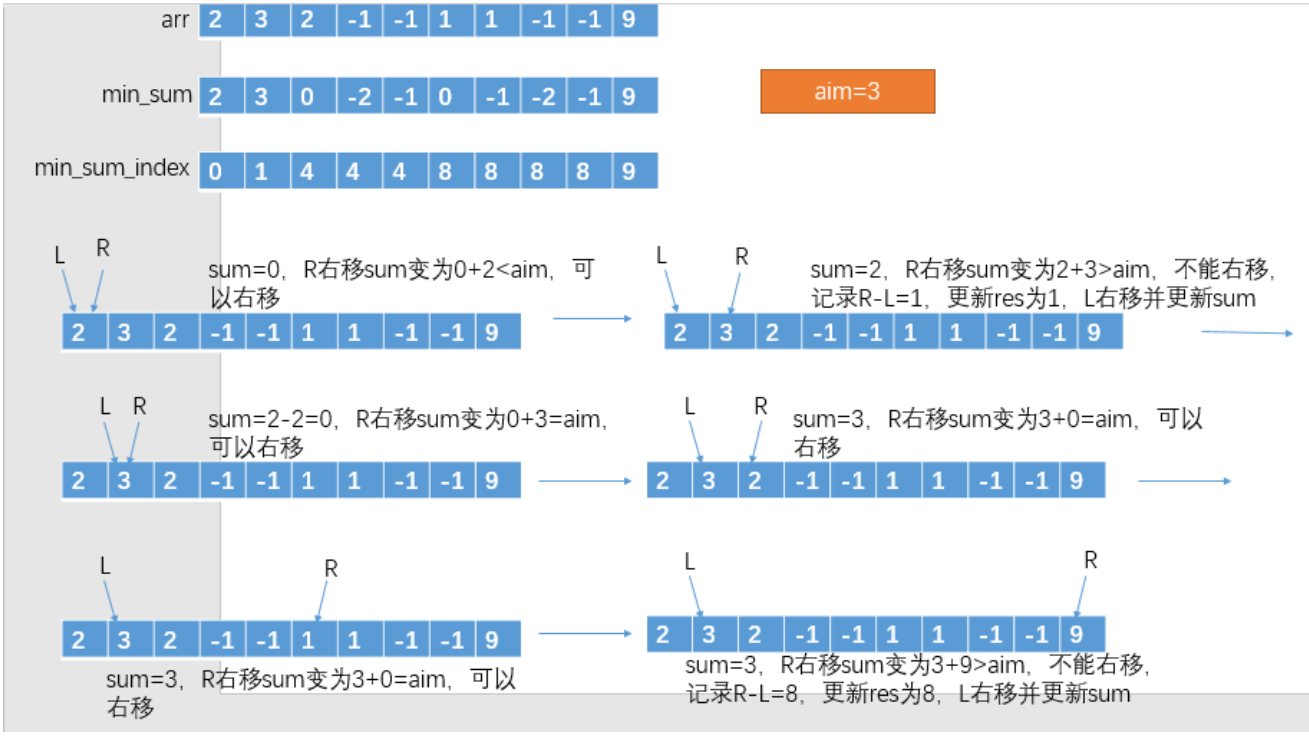
举例：对于 `[100,200,7,-6]` 。

1. 首先遍历3位置上的 `-6`，以 `-6` 开头的子数组只有 `[-6]`，因此 `min_sum[3] = -6, min_sum_index[3] = 3` （ `[-6]` 的尾元素 `-6` 在原数组中的下标是 `3` ）。
2. 接着遍历到2位置上的 `7`，以 `7` 开头的最小和子数组是 `[7,-6]`，因此 `min_sum[2] = 7-6 = 1, min_sum_index[2]=3` 。（ `[7,-6]` 的尾元素 `-6` 在原数组中的下标是 `3` ）。
3. 接着遍历到1位置上的 `200`，有 `min_sum[1] = 200, min_sum_index[1] = 1` 。
4. 接着遍历到0位置上的 `100`，有 `min_sum[0] = 100, min_sum_index[0] = 0` 。

那么遍历完数组，生成两个辅助数组之后，就可以开始正式的求解流程了：

使用一个窗口，L表示窗口的左边界，R表示窗口的右边界， `sum` 表示窗口内元素之和。

- L从头到尾依次来到数组中的每个元素，每次L来到其中一个元素上时，都尝试向右扩R，R扩到不能扩时，窗口大小 `R-L` 即为以该元素开头的、和小于等于aim的最长子数组的长度。
- L起初来到首元素，R起初也停在首元素， `sum=0` 。
- R向右扩一次的逻辑是：如果 `sum + min_sum[L] <= aim`，那么R就扩到 `min_sum_index[L] + 1` 的位置，并更新 `sum` 。
- R扩到不能扩时，记录 `R-L`，L去往下一个元素，并更新 `sum` 。
- 如果L来到一个元素后， `sum > aim`，说明以该元素开头的、和小于等于aim的最长子数组的长度，比当前的窗口大小 `R-L` 还要小，那么以该元素开头的子数组不在正确答案的考虑范围之内（因为上一个元素形成的最大窗口大于当前元素能形成的最大窗口，并且前者已经被记录过了），L直接去往一下个元素并更新 `sum` 。




示例代码：

```java
public static int lessOrEqualAim(int arr[], int aim) {
    int min_sum[] = new int[arr.length];
    int min_sum_index[] = new int[arr.length];
    min_sum[arr.length-1] = arr[arr.length - 1];
    min_sum_index[arr.length-1] = arr.length - 1;
    for (int i = arr.length - 2; i >= 0; i--) {
        if (min_sum[i + 1] < 0) {
            min_sum[i] = arr[i] + min_sum[i + 1];
            min_sum_index[i] = min_sum_index[i + 1];
        } else {
            min_sum[i] = arr[i];
            min_sum_index[i] = i;
        }
    }

    int R = 0;
    int sum = 0;
    int maxLen = 0;
    for (int L = 0; L < arr.length; L++) {
        while (R < arr.length && sum + min_sum[R] <= aim) {
            sum += min_sum[R];
            R = min_sum_index[R] + 1;
        }
        maxLen = Math.max(maxLen, R - L);
        sum -= R == L ? 0 : arr[L];
        R = Math.max(R, L + 1);
    }
    return maxLen;
}

```

```
31  public static void main(String[] args) {
32      int arr[] = {1, 2, 3, 2, -1, -1, 1, 1, -1, -1, 9};
33      System.out.println(lessOrEqualAim(arr,3));//8
34  }
```

19-27 行是实现的难点，首先19行是L从头到尾来到数组中的每个元素，然后 20-23 的 while 是尝试让R扩直到R扩不动为止， 24 行当R扩不动时就可以记录以当前L位置上的元素开头的、和小于等于aim的最长子数组长度，最后在进入下一次 for 循环、L右移一步之前， sum 的更新有两种情况：

1. 29 行的 while 执行了， R 扩出去了，因此 sum 直接减去当前L上的元素即可。
2. 29 行的 while 压根就没执行， R 一步都没扩出去且和 L 在同一位置上，也就是说此刻窗口内没有元素（只有当R>L时，窗口才包含从L开始到R之前的元素）， sum=0 ，L和R应该同时来到下一个元素， sum 仍为0，所以 sum 不必减去 arr[L] （只有当L右移导致一个元素从窗口出去时才需要减 arr[L] ）。

最后 26 行也是为了保证如果L在右移的过程中，R一直都扩不出去，那么在L右移到R上R仍旧扩不出去时，接下来R应该和L同时右移一个位置。

> 此方法能够做到 O(N) 时间复杂度的关键点是：舍去无效情况。比如L在右移一步更新 sum 之后，如果发现 sum > aim ，显然以当前L开头的、和小于等于aim的最长子数组肯定小于当前的 R-L ，而在上一步就记录了 R-(L-1) ，以当前L开头的满足条件的子数组可以忽略掉（因为一定小于 R-(L-1) ），而不必让R回退到当前L重新来扩R。
>
> 这样L和R都只右移而不回退，所以时间复杂度就是遍历了一遍数组。

# 环形单链表的约瑟夫问题

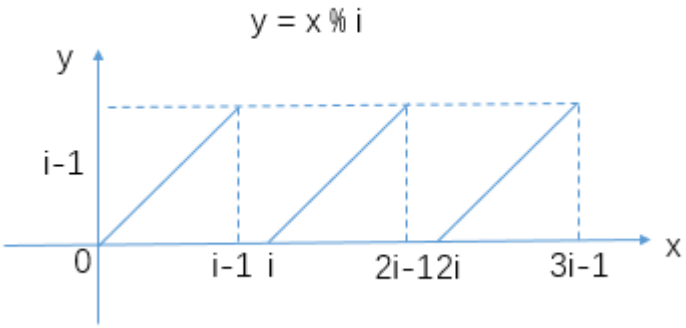
据说著名犹太历史学家Josephus有过以下故事：在罗马人占领乔塔帕特后，39个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式，41个人排成一个圆圈，由第1个人开始报数，报数到3的人就自杀，然后再由下一个人重新报1，报数到3的人再自杀，这样依次下去，直到剩下最后一个人时，那个人可以自由选择自己的命运。这就是著名的约瑟夫问题。现在请用单向环形链表描述该结构并呈现整个自杀过程。

> **输入**：一个环形单向链表的头节点head和报数的值m。
>
> **返回**：最后生存下来的节点，且这个节点自己组成环形单向链表，其他节点都删掉。
>
> **进阶**：如果链表节点数为N，想在时间复杂度为O(N)时完成原问题的要求，该怎么实现？

暴力方法：从头结点开始数，从1数到m，数到m时删除结点，再从下一个结点开始数……如此要删除（n-1）个结点，并且每次删除之前要数m个数，因此时间复杂度为 O(NxM)

这里介绍一种 O(N) 的方法。

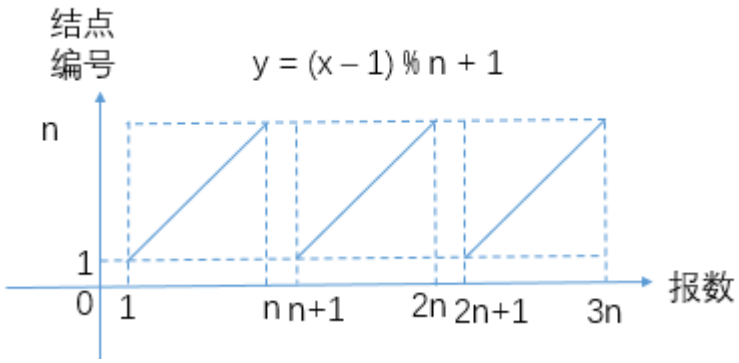
首先介绍一个函数：




如果从头结点开始，为每个结点依次编号1、2、3、……，比如环形链表有3个结点，每次报数到7时杀人：

| 结点编号 | 报数 |
| --- | --- |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 1 | 4 |
| 2 | 5 |
| 3 | 6 |
| 1 | 杀人 |

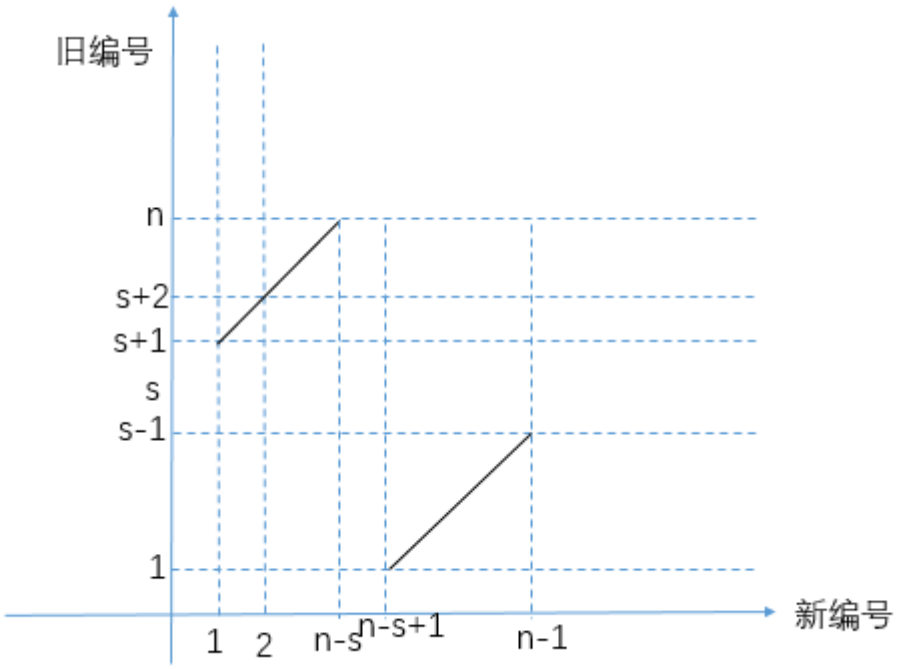
那么在杀人之前，结点编号和报数有如下对应关系（x轴代表此刻报数报到哪儿了，y轴则对应是几号结点报的，n是结点数量）：



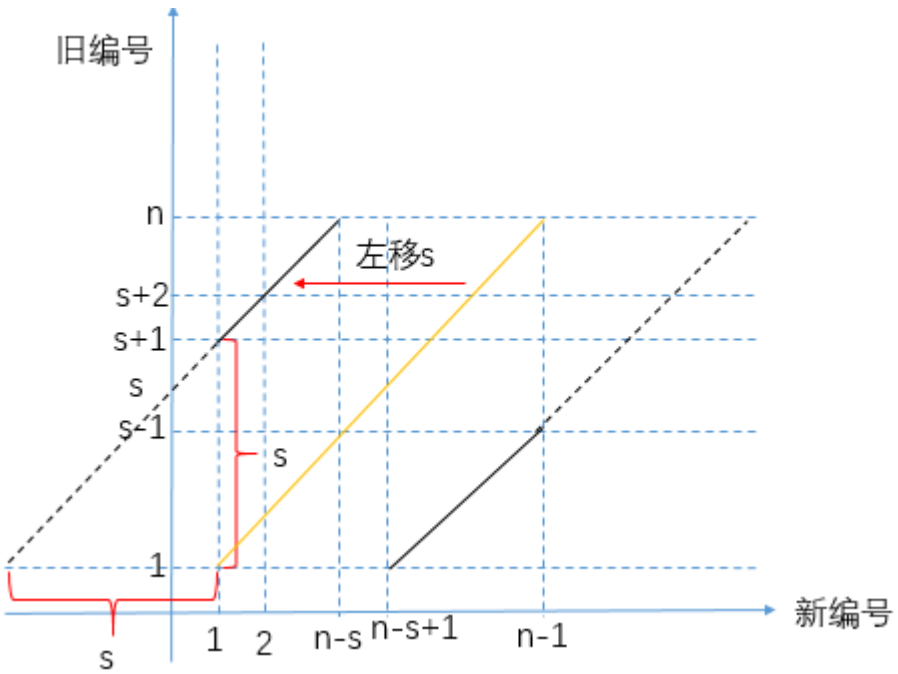

假设每次杀人后，都从下一结点重新编号、重新报数，比如环形链表有9个结点，报数到7就杀人，那么杀人之前结点的旧编号和杀人重新编号后结点的新编号有如下关系：

| 旧编号 | 新编号 |
| --- | --- |
| 1 | 3 |
| 2 | 4 |
| 3 | 5 |
| 4 | 6 |
| 5 | 7 |
| 6 | 8 |
| 7 | 被杀，从下一结点开始重新编号 |
| 8 | 1 |
| 9 | 2 |

如果链表结点数为n，报数到m杀人，那么结点的新旧编号对应关系如下（其中`s`为报数为m的结点编号）：




这个图也可以由基本函数`y = (x - 1) % n + 1`向左平移s个单位长度变换而来：




即 `y = (x - 1 + s) % n + 1`。

现在我们有了如下两个公式：

1. `结点编号 = (报数 - 1) % n + 1`
2. `旧编号 = (新编号 - 1 + s) % n +1`，其中 `s` 为报数为m的结点编号

由1式可得 `s = (m - 1) % n + 1`，带入2式可得

3. `旧编号 = (新编号 - 1 + (m - 1) % n + 1) % n + 1 = (新编号 + m - 1) % n + 1`，其中 `m` 和 `n` 由输入参数决定。

现在我们有了等式3，就可以在已知一个结点在另一个结点被杀之后的新编号的情况下，求出该结点的旧编号。也就是说，假设现在杀到了第 `n-1` 个结点，杀完之后只剩下最后一个结点了（天选结点），重新编号后天选结点肯定是1号，那么第 `n-1` 个被杀结点被杀之前天选结点的编号我们就可以通过等式3求出来，通过这个结果我们又能求得天选结点在第 `n-2` 个被杀结点被杀之前的编号，……，依次往回推就能还原一个结点都没死时天选结点的编号，这样我们就能从输入的链表中找到该结点，直接将其后继指针指向自己然后返回即可。

示例代码：

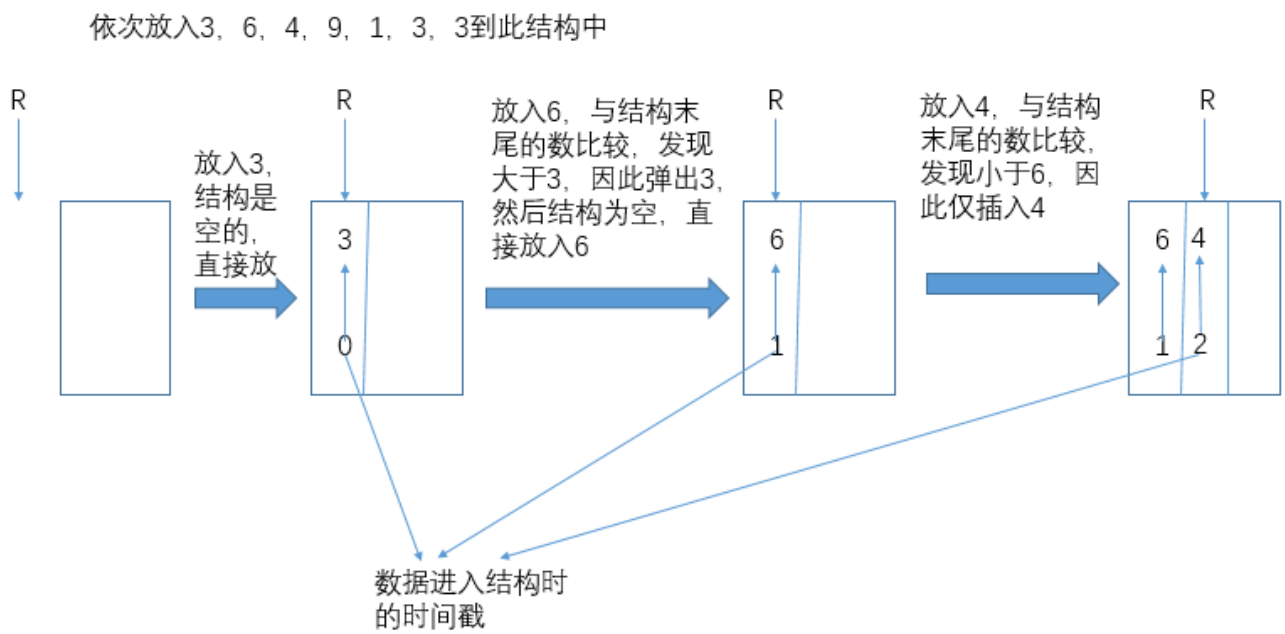
```
static class Node {
    char data;
    Node next;

    public Node(char data) {
        this.data = data;
    }
}

public static Node aliveNode(Node head, int m) {
    if (head == null) {
        return null;
    }
    int tmp = 1;
    Node cur = head.next;
```

```
    while (cur != head) {
        tmp++;
        cur = cur.next;
    }

    //第n-1次杀人前还有两个结点，杀完之后天选结点的新编号为1
    //通过递归调用getAlive推出所有结点存活时，天选结点的编号
    int nodeNumber = getAlive(1, m, 2, tmp);

    cur = head;
    tmp = 1;
    while (tmp != nodeNumber) {
        cur = cur.next;
        tmp++;
    }
    cur.next = cur;
    return cur;
}

/**
     * 旧编号 = (新编号 + m - 1) % n + 1
     *
     * @param newNumber 新编号
     * @param m
     * @param n         旧编号对应的存活的结点个数
     * @param len       结点总个数
     * @return
     */
public static int getAlive(int newNumber, int m, int n, int len) {
    if (n == len) {
        return (newNumber + m - 1) % n + 1;
    }
    //计算出新编号对应的旧编号，将该旧编号作为下一次计算的新编号
    return getAlive((newNumber + m - 1) % n + 1, m, n + 1, len);
}

public static void main(String[] args) {
    Node head = new Node('a');
    head.next = new Node('b');
    head.next.next = new Node('c');
    head.next.next.next = new Node('d');
    head.next.next.next.next = new Node('e');
    head.next.next.next.next.next = head;

    System.out.println(aliveNode(head, 3).data);//d
}
```

# 经典结构

# 窗口最大值更新结构

## 最大值更新结构




当向此结构放数据时会检查一下结构中的已有数据，从时间戳最大的开始检查，如果检查过程中发现该数据小于即将放入的数据则将其弹出并检查下一个，直到即将放入的数据小于正在检查的数据或者结构中的数据都被弹出了为止，再将要放入的数据放入结构中并盖上时间戳。如此每次从该结构取数据时，都会返回结构中时间戳最小的数据，也是目前为止进入过此结构的所有数据中最大的那一个。

此结构可以使用一个双端队列来实现，一端只用来放数据（放数据之前的检查过程可能会弹出其他数据），另一端用来获取目前为止出现过的最大值。

示例如下：

```java
package top.zhenganwen.structure;

import java.util.LinkedList;

public class MaxValueWindow {

  private LinkedList<Integer> queue;
  public MaxValueWindow() {
    this.queue = new LinkedList();
  }

  //更新窗口最大值
  public void add(int i){
    while (!queue.isEmpty() && queue.getLast() <= i) {
      queue.pollLast();
    }
    queue.add(i);
  }

  //获取窗口最大值
```

```
  public int getMax() {
    if (!queue.isEmpty()) {
      return queue.peek();
    }
    return Integer.MIN_VALUE;
  }

  //使窗口最大值过期
  public void expireMaxValue() {
    if (!queue.isEmpty()) {
      queue.poll();
    }
  }

  public static void main(String[] args) {
    MaxValueWindow window = new MaxValueWindow();
    window.add(6);
    window.add(4);
    window.add(9);
    window.add(8);
    System.out.println(window.getMax());//9
    window.expireMaxValue();
    System.out.println(window.getMax());//8
  }
}
```

# 例题

## 窗口移动

给你一个长度为`N`的整型数组和大小为`W`的窗口，用一个长度为`N-W+1`的数组记录窗口从数组由左向右移动过程中窗口内最大值。

对于数组`[1,2,3,4,5,6,7]`和窗口大小为`3`，窗口由左向右移动时有：

- `[1,2,3],4,5,6,7`，窗口起始下标为0时，框住的数是`1,2,3`，最大值是3
- `1,[2,3,4],5,6,7`，最大值是4
- `1,2,[3,4,5],6,7`，最大值是5
- ……

因此所求数组是`[3,4,5,6,7]`。

> 思路：前面介绍的窗口最大值更新结构的特性是，先前放入的数如果还存在于结构中，那么该数一定比后放入的数都大。此题窗口移动的过程就是从窗口中减一个数和增一个数的过程。拿`[1,2,3],4`到`1,[2,3,4]`这一过程分析：首先`[1,2,3],4`状态下的窗口应该只有一个值`3`（因为先加了1，加2之前弹出了1，加3之前弹出了2）；转变为`1,[2,3,4]`的过程就是向窗口先减一个数`1`再加一个数`4`的过程，因为窗口中不含`1`所以直接加一个数`4`（弹出窗口中的`3`，加一个数`4`）。

代码示例：

```
public static void add(int arr[], int index, LinkedList<Integer> queue) {
  if (queue == null) {
    return;
```

```
  }
  while (!queue.isEmpty() && arr[queue.getLast()] < arr[index]) {
    queue.pollLast();
  }
  queue.add(index);
}

public static void expireIndex(int index, LinkedList<Integer> queue) {
  if (queue == null) {
    return;
  }
  if (!queue.isEmpty() && queue.peek() == index) {
    queue.pollFirst();
  }
}

public static int[] maxValues(int[] arr, int w) {
  int[] res = new int[arr.length - w + 1];
  LinkedList<Integer> queue = new LinkedList();
  for (int i = 0; i < w; i++) {
    add(arr, i, queue);
  }
  for (int i = 0; i < res.length; i++) {
    res[i] = queue.peek();
    if (i + w <= arr.length - 1) {
      expireIndex(i, queue);
      add(arr, i + w, queue);
    }
  }
  for (int i = 0; i < res.length; i++) {
    res[i] = arr[res[i]];
  }
  return res;
}

public static void main(String[] args) {
  int[] arr = {3, 2, 1, 5, 6, 2, 7, 8, 10, 6};
  System.out.println(Arrays.toString(maxValues(arr,3)));//[3, 5, 6, 6, 7, 8, 10, 10]
}
```

这里需要的注意的是，针对这道题将窗口最大值更新结构的`add`和`expire`方法做了改进（结构中存的是值对应的下标）。例如`[2,1,2],-1->2,[1,2,-1]`，应当翻译为`[2,1,2],-1`状态下的窗口最大值为2下标上的数`2`，变为`2,[1,2,-1]`时应当翻译为下标为0的数从窗口过期了，而不应该是数据`2`从窗口过期了（这样会误删窗口中下标为2的最大值2）。

## 求达标的子数组个数

给你一个整型数组，判断其所有子数组中最大值和最小值的差值不超过`num`（如果满足则称该数组达标）的个数。（子数组指原数组中任意个连续下标上的元素组成的数组）

暴力解：遍历每个元素，再遍历以当前元素为首的所有子数组，再遍历子数组找到其中的最大值和最小值以判断其是否达标。很显然这种方法的时间复杂度为`o(N^3)`，但如果使用最大值更新结构，则能实现`O(N)`级别的解。

如果使用 L 和 R 两个指针指向数组的两个下标，且 L 在 R 的左边。当 L~R 这一子数组达标时，可以推导出以 L 开头的长度不超过 R-L+1 的所有子数组都达标；当 L~R 这一子数组不达标时，无论 L 向左扩多少个位置或者 R 向右扩多少个位置， L~R 还是不达标。

O(N) 的解对应的算法是： L 和 R 都从0开始， R 先向右移动， R 每右移一个位置就使用最大值更新结构和最小值更新结构记录一下 L~R 之间的最大值和最小值的下标，当 R 移动到如果再右移一个位置 L~R 就不达标了时停止，这时以当前 L 开头的长度不超过 R-L+1 的子数组都达标；然后 L 右移一个位置，同时更新一下最大值、最小值更新结构（ L-1 下标过期了），再右移 R 至 R 如果右移一个位置 L~R 就不达标了停止（每右移 R 一次也更新最大、小值更新结构）……；直到 L 到达数组尾元素为止。将每次 R 停止时， R-L+1 的数量累加起来就是 O(N) 的解，因为 L 和 R 都只向右移动，并且每次 R 停止时，以 L 开头的达标子串的数量直接通过 R-L+1 计算，所以时间复杂度就是将数组遍历了一遍即 O(N) 。

示例代码：

```
public static int getComplianceChildArr(int arr[], int num) {
  //最大值、最小值更新结构
  LinkedList<Integer> maxq = new LinkedList();
  LinkedList<Integer> minq = new LinkedList<>();
  int L = 0;
  int R = 0;
  maxq.add(0);
  minq.add(0);
  int res = 0;
  while (L < arr.length) {
    while (R < arr.length - 1) {
      while (!maxq.isEmpty() && arr[maxq.getLast()] <= arr[R + 1]) {
        maxq.pollLast();
      }
      maxq.add(R + 1);
      while (!minq.isEmpty() && arr[minq.getLast()] >= arr[R + 1]) {
        minq.pollLast();
      }
      minq.add(R + 1);
      if (arr[maxq.peekFirst()] - arr[minq.peekFirst()] > num) {
        break;
      }
      R++;
    }
    res += (R - L + 1);
    if (maxq.peekFirst() == L) {
      maxq.pollFirst();
    }
    if (minq.peekFirst() == L) {
      minq.pollFirst();
    }
    L++;
  }
  return res;
}

public static void main(String[] args) {
  int[] arr = {1, 2, 3, 5};
```

```
  System.out.println(getComplianceChildArr(arr, 3));//9
}
```

# 单调栈结构

## 原始问题

给你一个数组，找出数组中每个数左边离它最近的比它大的数和右边离它最近的比它大的数。

> 思路：使用一个栈，要求每次元素进栈后要维持栈中从栈底到栈顶元素值是从大到小排列的约定。将数组中的元素依次进栈，如果某次元素进栈后会违反了上述的约定（即该进栈元素比栈顶元素大），就先弹出栈顶元素，并记录该栈顶元素的信息：
>
> - 该元素左边离它最近的比它大的是该元素出栈后的栈顶元素，如果出栈后栈空，那么该元素左边没有比它大的数
> - 该元素右边离它最近的比它大的是进栈元素
>
> 然后再尝试将进栈元素进栈，如果进栈后还会违反约定那就重复操作"弹出栈顶元素并记录该元素信息"，直到符合约定或栈中元素全部弹出时再将该进栈元素进栈。当数组所有元素都进栈之后，栈势必不为空，弹出栈顶元素并记录信息：
>
> - 该元素右边没有比它大的数
> - 该元素左边离它最近的比它大的数是该元素从栈弹出后的栈顶元素，如果该元素弹出后栈为空，那么该元素左边没有比它大的数
>
> 由于每个元素仅进栈一次、出栈一次，且出栈时能得到题目所求信息，因此时间复杂度为`O(N)`

示例代码：

```
public static void findLeftAndRightBigger(int arr[]){
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < arr.length; i++) {
        //check the agreement before push the index of element
        while (!stack.empty() && arr[stack.peek()] < arr[i]) {
            //pop and record the info(print or save)
            int index = stack.pop();
            System.out.print("index:" + index + ",element:" + arr[index] + ",right
bigger is:" + arr[i]);
            if (stack.empty()) {
                System.out.print(",hasn't left bigger\n");
            } else {
                System.out.println(",left bigger is:" + arr[stack.peek()]+"\n");
            }
        }
        //push
        stack.push(i);
    }
    while (!stack.empty()) {
        int index = stack.pop();
        System.out.print("index:" + index + ",element:" + arr[index] + ",hasn't
right bigger");
        if (stack.empty()) {
            System.out.print(",hasn't left bigger\n");
```

```
        } else {
            System.out.println(",left bigger is:" + arr[stack.peek()]+"\n");
        }
    }
}

public static void main(String[] args) {
    int[] arr = {2, 1, 7, 4, 5, 9, 3};
    findLeftAndRightBigger(arr);
}
```

## 给你一些数，创建一棵大根堆二叉树

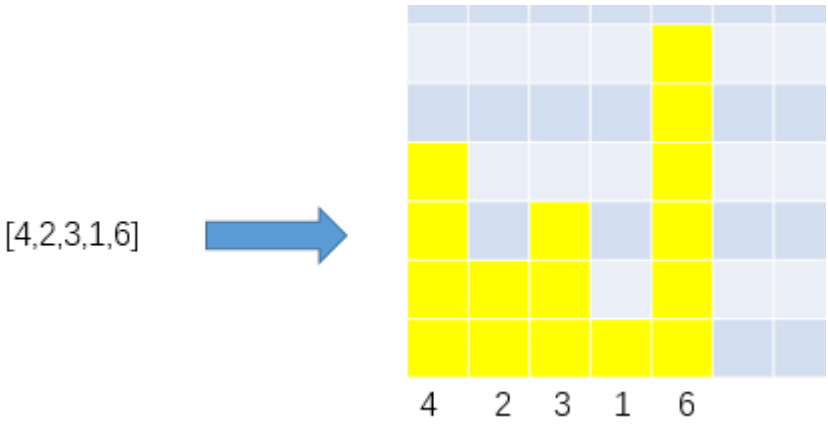
> 思路：使用一个栈底到栈顶单调递减的单调栈，将这些数`arr[]`依次入栈，记录每个数左边离它最近的比它大的数，保存在`left[]`中（下标和`arr[]`一一对应），记录每个数右边离它最近的比它大的数，保存在`right[]`中。
>
> 遍历`arr[]`建树：`left[i]`和`right[i]`都不存在的，说明`arr[i]`是最大的数，将其作为根节点；对于其他任何一个数`arr[i]`，`left[i]`和`right[i]`必有一个存在，如果都存在则将`arr[i]`作为`Math.min(left[i],right[i])`的孩子节点，如果只有一个存在（如`left[i]`）那就将`arr[i]`作为`left[i]`的孩子节点

> 思考：这样建出的树会不会是森林，会不会不是二叉树？

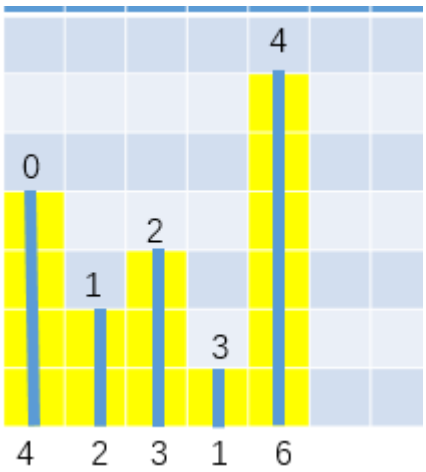
## 找出矩阵中一片1相连的最大矩形

矩阵中的数只会是0或1，求矩阵中一片1形成的最大长方形区域的面积。

此题可借鉴在直方图中找最大矩形的方法。首先一个数组可以对应一个直方图，如下所示：




接着，遍历数组，以当前遍历元素值为杆子的高并尝试向左右移动这根杆子（约定杆子不能出黄色区域）：




如上图，0号杆子向左右移动一格都会使杆子出界（黄色区域），因此0号杆子的活动面积是`4x1=4`（杆长x能活动的格子数）；1号杆子向左、向右都只能移动一格，因此其活动面积是`2x3=6`；2号杆子的活动面积是`3x1=3`；3号杆子的活动面积是`1x5=5`；4号杆子的活动面积是`6x1=6`。因此该直方图中最大矩形面积就是所有杆子的活动面积中最大的那个，即6。

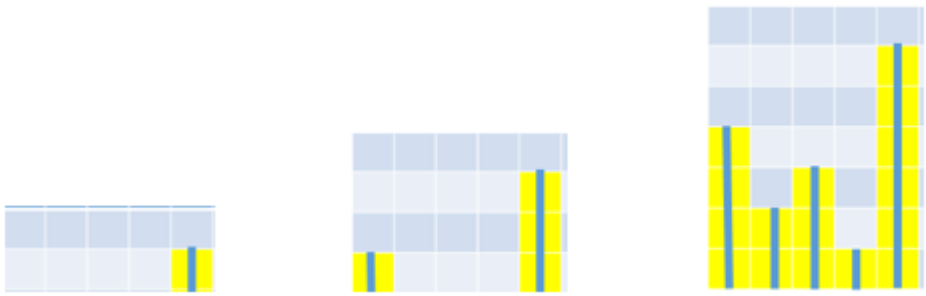
如果现在给你一个矩阵，比如

```
0 0 0 0 1
0 0 0 0 1
1 0 0 0 1
1 0 1 0 1
1 1 1 0 1
1 1 1 1 1
```

你能否将其中相连的一片1看成直方图中的黄色区域，如此的话求矩阵由一片1形成的最大矩形区域就是求直方图中最大矩形面积了。

所以对于输入的矩形，我们只要遍历每一行，以该行作为直方图的x轴，求出直方图的最大矩形面积，再比较所有行对应的最大矩形面积就能得出整个矩阵的一片1形成的最大矩形区域了。

以上面的矩阵为例，第一行、第三行、最后一行对应的直方图如下所示：



分别可以用数组`[0,0,0,0,1]`、`[1,0,0,0,3]`、`[4,2,3,1,6]`来表示，那么此题关键的点就是遍历每一行并求出以该行为x轴的直方图的数组表示之后，如何得出此直方图的最大矩形面积。下面就使用单调栈来解决此问题：

以`[4,2,3,1,6]`的求解过程为例，使用一个栈底到栈顶单调递增的栈将数组中的数的下标作为该数的代表依次压栈（数的下标->数值），首先能压的是`0->4`，接着准备压`1->2`，发现`2`比栈顶的`4`小，压人后会违反栈底到栈顶单调递增的约定，因此弹出`0->4`并记录0号杆子的活动面积（`0->4`弹出后栈为空，说明0号杆子左移到x轴的-1就跑出黄色区域了，由于是`1->2`让它弹出的，所以0号杆子右移到x轴的1就出界了，因此0号杆子只能在x轴上的0

位置上活动，活动面积是`4x1=4`，称这个记录的过程为**结算**）。由于弹出`0->4`之后栈空了，所以可以压入`1->2`、`2->3`，接着准备压`3->1`时发现`1`比栈顶`3`小，因此结算`2->3`（由于弹出`2->3`之后栈顶为`1->2`，因此2号杆子左移到x轴1位置时出界了，由于是`3->1`让其弹出的，所以2号杆子右移到x轴3位置就出界了，因此2号杆子的活动面积是`3x1=3`）。接着再准备压`3->1`，发现`1`比栈顶`1->2`的`2`小，因此结算`1->2`（弹出`1->2`后栈空，因此1号杆子左移到x轴-1时才出界，`3->1`让其出界的，因此右移到3时才出界，活动面积为`2x3=6`）……

所有数压完之后，栈肯定不为空，那么栈中剩下的还需要结算，因此依次弹出栈顶进行结算，比如`[4,2,3,1,6]`压完之后，栈中还剩`3->1,4->6`，因此弹出`4->6`并结算（由于`4->6`不是因为一个比`6`小的数要进来而让它弹出的，所以4号杆子右移到x轴`arr.length=5`位置才出界，由于弹出后栈不空且栈顶为`3->1`，所以左移到x轴的3位置上才出界的，所以活动面积为`6x1=6`；同样的方法结算`3->1`……直到栈中的都被结算完，整个过程结束。

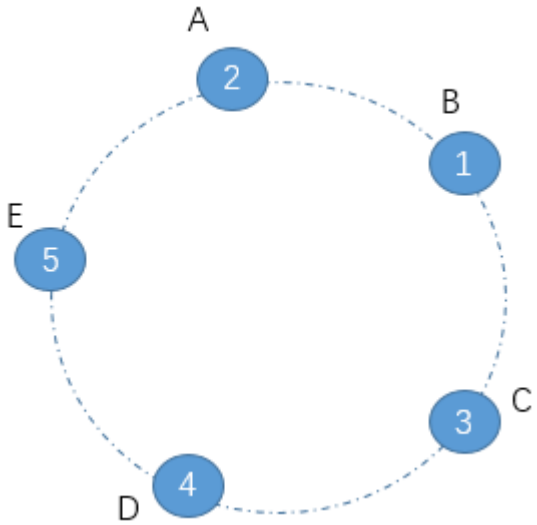
示例代码：

```
public static int maxRectangleArea(int matrix[][]){
    int arr[] = new int[matrix[0].length];
    int maxArea = Integer.MIN_VALUE;
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            arr[j] = matrix[i][j] == 1 ? arr[j]+1 : 0;
        }
        System.out.println(Arrays.toString(arr));
        maxArea = Math.max(maxArea, maxRecAreaOfThRow(arr));
    }
    return maxArea;
}

public static int maxRecAreaOfThRow(int arr[]){
    int maxArea = Integer.MIN_VALUE;
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < arr.length; i++) {
        while (!stack.empty() && arr[i] < arr[stack.peek()]) {
            int index = stack.pop();
            int leftBorder = stack.empty() ? -1 : stack.peek();
            maxArea = Math.max(maxArea, arr[index] * (i - leftBorder - 1));
        }
        stack.push(i);
    }
    while (!stack.empty()) {
        int index = stack.pop();
        int rightBorder = arr.length;
        int leftBorder = stack.empty() ? -1 : stack.peek();
        maxArea = Math.max(maxArea, arr[index] * (rightBorder - leftBorder - 1));
    }
    return maxArea;
}

public static void main(String[] args) {
    int matrix[][] = {
        {0, 0, 0, 0, 1},
        {0, 0, 0, 0, 1},
        {1, 0, 0, 0, 1},
        {1, 0, 1, 0, 1},
```

```
        {1, 1, 1, 0, 1},
        {1, 1, 1, 1, 1}
    };
    System.out.println(maxRectangleArea(matrix));//6
}
```

## 烽火相望

【网易原题】给你一个数组，数组中的每个数代表一座山的高度，这个数组代表将数组中的数从头到尾连接而成的环形山脉。比如数组`[2,1,3,4,5]`形成的环形山脉如下：




其中蓝色的圆圈就代表一座山，圈中的数字代表这座山的高度。现在在每座山的山顶都点燃烽火，假设你处在其中的一个山峰上，要想看到另一座山峰的烽火需满足以下两个条件中的一个：

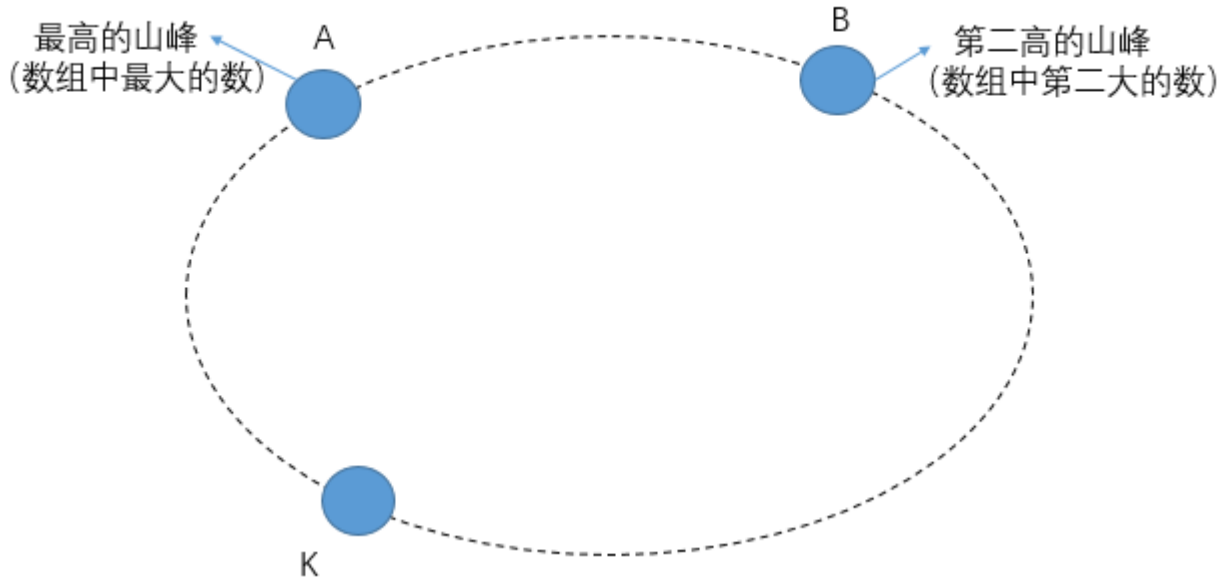
- 你想看的山峰在环形路径上与你所在的山峰相邻。比如你在山峰A上，那么你能够看到B和E上的烽火。
- 如果你想看的山峰和你所在的山峰不相邻，那么你可以沿环形路径顺时针看这座山也可以沿环形路径逆时针看这座山，只要你放眼望去沿途经过的山峰高度小于你所在的山峰和目标山峰，那么也能看到。比如C想看E，那么可以通过C->B->A->E的方式看，也可以通过C->D->E的方式看。前者由于经过的山峰的高度1和2比C的高度3和E的高度5都小，因此能看到；但后者经过的山峰D的高度4大于C的高度3，因此C在通过C->D->E这个方向看E的时候视线就被山峰D给挡住了。

问：所有山峰中，能互相看到烽火的两两山峰的对数。以`[2,1,3,4,5]`为例，能互相看见的有：`2,1`，`1,3`，`3,4`，`4,5`，`5,2`，`2,3`，`3,5`，共7对。

此题分一下两种情况

### 1、数组中无重复的数

这种情况下，答案可以直接通过公式`2*N-3`可以求得（其中`N`为数组长度），证明如下：



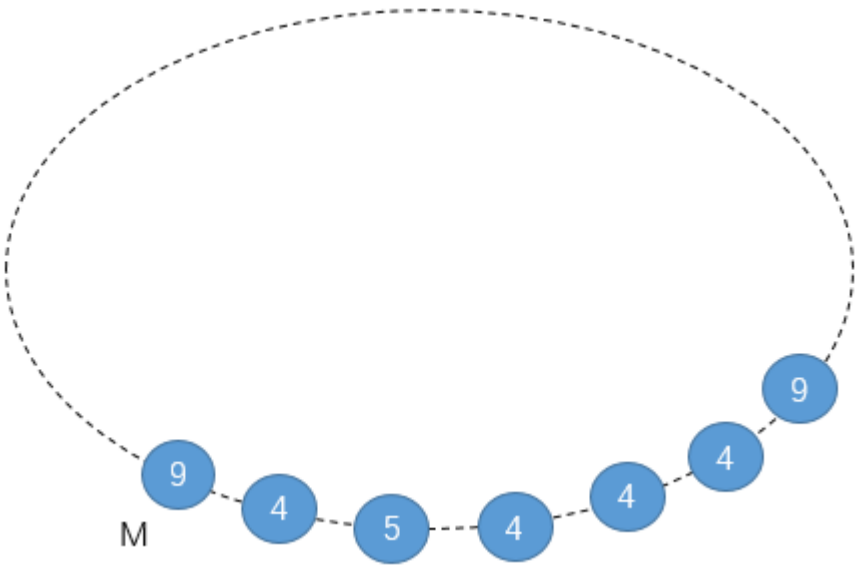

假设A是在山峰中最高，B在所有山峰中第二高。那么环形路径上介于A和B之间的任意一座山峰（比如K），逆时针方向在到达A之前或到达A时一定会遇到第一座比它高的山峰，记这座山峰和K是一对；顺时针方向，在到达B之前或到达B时，一定会遇到第一个比K高的山峰，记这座山峰和K是一对。也就是说对于除A,B之外的所有山峰，都能找到两对符合标准的，这算下来就是`(N-2)*2`了，最后AB也算一对，总数是`(N-2)*2+1=2N-3`。

但如果数组中有重复的数就不能采用上述的方法了

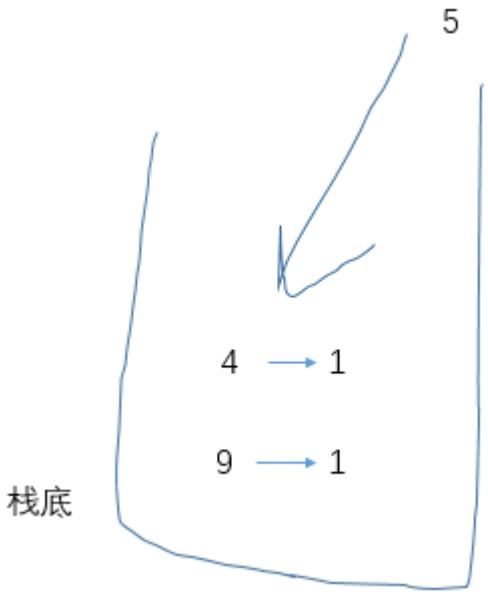
## 2、数组中可能有重复的数

> 利用单调栈

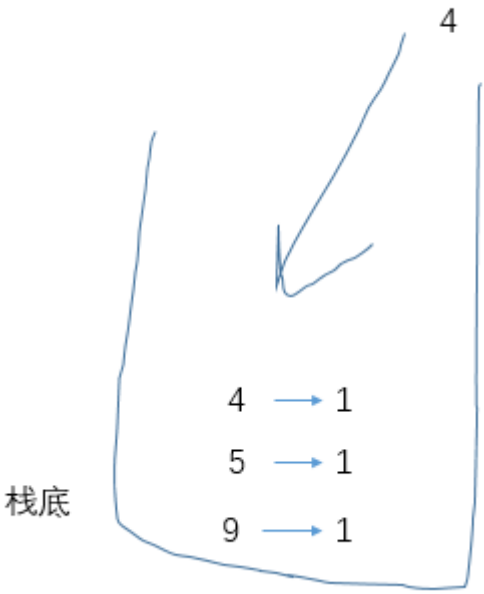
首先找出数组中最大数第一次出现的位置，记为`M`。从这个数开始遍历数组并依次压栈（栈底到栈底从大到小的单调栈），以如下的环形山脉为例：



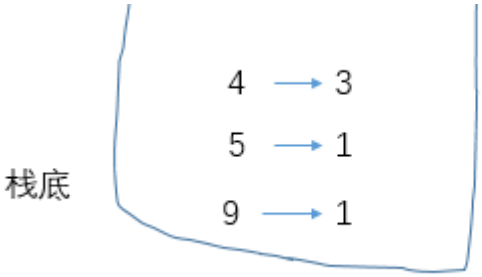

从M开始压栈，同时附带一个计数器：




当压入5时，违反单调栈约定因此结算4（4左边第一个比它高的是9，右边第一个比它高的是5，因此能和4配对的有两对）；接着再压入5、压入4，重点来了：连续两次再压入4该如何处理：



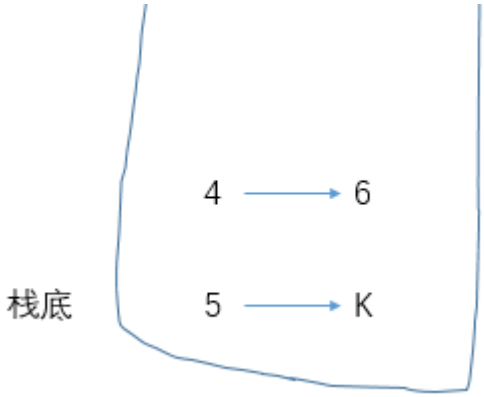

这是数组中有重复的数时，如何使用单调栈解决此题的关键：如果压入的元素与栈顶元素相同，将栈顶元素的**计数器加1**，那么再压入两个4之后栈中情况：



然后压入9，导致弹出并结算4。那么**如何结算计数器大于1的数据**呢？首先，这3座高度相同的山峰两两配对能够组成`C(3,2)=3`对，此外其中的每座山峰左边离它最近的比它高的是5、右边离它近的比它大的是9，因此这3座山峰每座都能和`5、9`配对，即`3*2=6`，因此结算结果为`3+6=9`.....

如果数据压完了，那就从栈顶弹出数据进行结算，直到结算栈底上一个元素之前（栈底元素是最大值），弹出数据的结算逻辑都是`C(K,2)+K*2`（其中K是该数据的计数器数值）。

倒数第二条数据的结算逻辑有点复杂，如图，以结算4为例：




如果K的数值大于1，那么这6座高度为4的山峰结算逻辑还是上述公式。但如果K为1，那么结算公式就是`C(K,2)+K*1`了。

最后对于最大值M的结算，假设其计数器的值为K，如果K=1，那么结算结果为0；如果K>1，那么结算结果为`C(K,2)`。

示例代码：

```
public static class Record{
    int value;
    int times;
    public Record(int value) {
        this.value = value;
        this.times = 1;
    }
}

public static int comunications(int[] arr) {
    //index of first max value
    int maxIndex = 0;
    for (int i = 0; i < arr.length; i++) {
        maxIndex = arr[maxIndex] < arr[i] ? i : maxIndex;
    }

    Stack<Record> stack = new Stack<>();
    stack.push(new Record(arr[maxIndex]));

    int res = 0;
    int index = nextIndex(arr, maxIndex);
    while (index != maxIndex) {
        while (!stack.empty() && arr[index] > stack.peek().value) {
            Record record = stack.pop();
            res += getInternalPairs(record.times) + record.times * 2;
```

```
        }
        if (arr[index] == stack.peek().value) {
            stack.peek().times++;
        } else {
            stack.push(new Record(arr[index]));
        }
        index = nextIndex(arr, index);
    }

    while (!stack.empty()) {
        Record record = stack.pop();
        res += getInternalPairs(record.times);
        if (!stack.empty()) {
            res += record.times;
            if (stack.size() > 1) {
                res += record.times;
            } else {
                res += stack.peek().times > 1 ? record.times : 0;
            }
        }
    }
    return res;
}

//C(K,2)
public static int getInternalPairs(int times){
    return (times * (times - 1)) / 2;
}

public static int nextIndex(int[] arr, int index) {
    return index < arr.length - 1 ? index + 1 : 0;
}

public static void main(String[] args) {
    int[] arr = {9, 4, 5, 4, 4, 4, 9,1};
    System.out.println(comunications(arr));
}
```

# 搜索二叉树

搜索二叉树的定义：对于一棵二叉树中的任意子树，其左子树上的所有数值小于头结点的数值，其右子树上所有的数值大于头结点的数值，并且树中不存在数值相同的结点。也称二叉查找树。

## 平衡二叉树/AVL树

### 平衡性

经典的平衡二叉树结构：在满足搜索二叉树的前提条件下，对于一棵二叉树中的任意子树，其左子树和其右子树的高度相差不超过1。

## 典型搜索二叉树——AVL树、红黑树、SBT树的原理

### AVL树

AVL树是一种具有严苛平衡性的搜索二叉树。什么叫做严苛平衡性呢？那就是**所有子树的左子树和右子树的高度相差不超过1。**弊端是，每次发现因为插入、删除操作破坏了这种严苛的平衡性之后，都需要作出相应的调整以使其恢复平衡，调整较为频繁。

### 红黑树

红黑树是每个节点都带有颜色属性的搜索二叉树，颜色或红色或黑色。在搜索二叉树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:

- 性质1. 节点是红色或黑色。
- 性质2. 根节点是黑色。
- 性质3 每个叶节点（NIL节点，空节点）是黑色的。
- 性质4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
- 性质5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

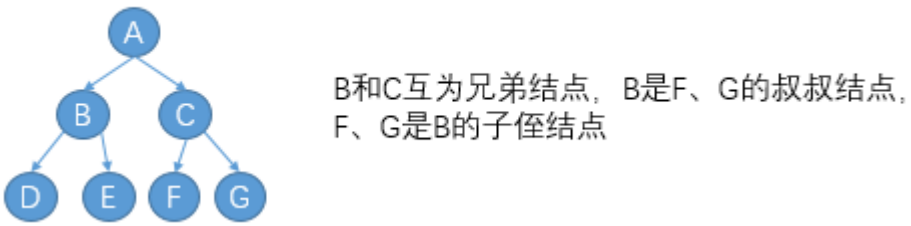
这些约束强制了红黑树的关键性质: **从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。**结果是这个树**大致上是平衡**的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

要知道为什么这些特性确保了这个结果，注意到**性质4导致了路径不能有两个毗连的红色节点**就足够了。**最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。**因为根据性质5所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

### SBT树

它是由中国广东中山纪念中学的陈启峰发明的。陈启峰于2006年底完成论文《Size Balanced Tree》，并在2007年的全国青少年信息学奥林匹克竞赛冬令营中发表。**相比红黑树、AVL树等自平衡二叉查找树，SBT更易于实现。据陈启峰在论文中称，SBT是"目前为止速度最快的高级二叉搜索树"。SBT能在O(log n)的时间内完成所有二叉搜索树(BST)的相关操作**，而与普通二叉搜索树相比，SBT仅仅加入了简洁的核心操作Maintain。由于SBT赖以保持平衡的是size域而不是其他"无用"的域，它可以很方便地实现动态顺序统计中的select和rank操作。

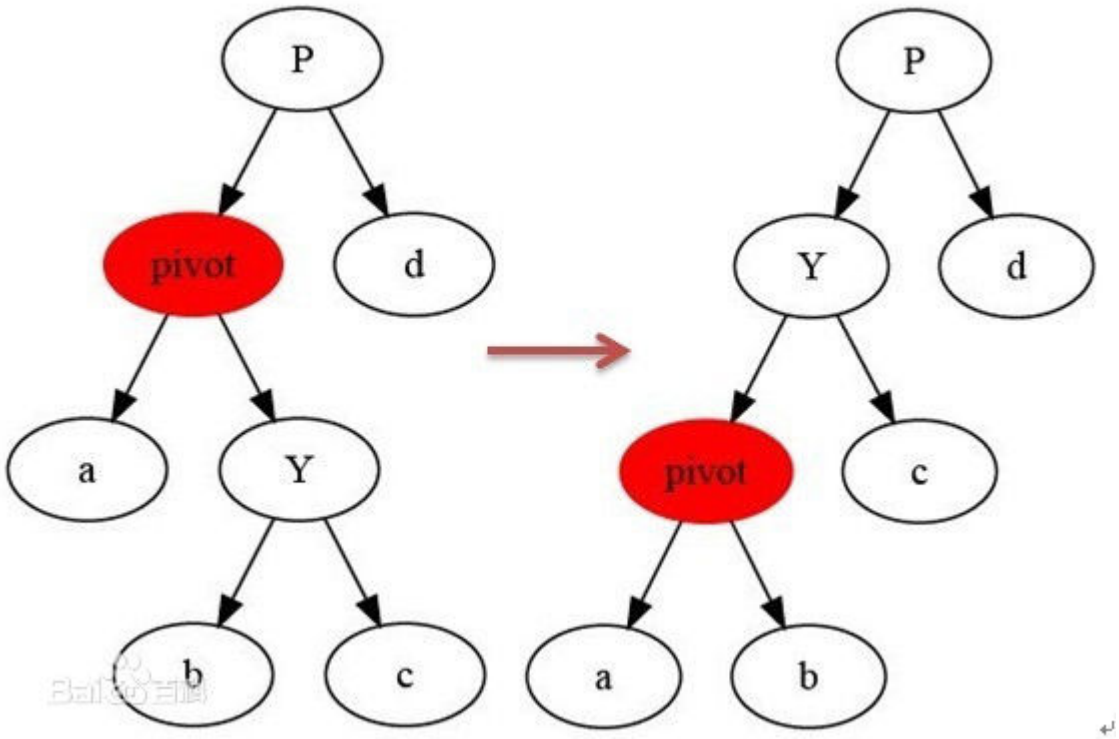
**SBT树的性质**是：对于数中任意结点，以该结点为根节点的子树的结点个数不能比以该结点的叔叔结点为根节点的子树的结点个数大。



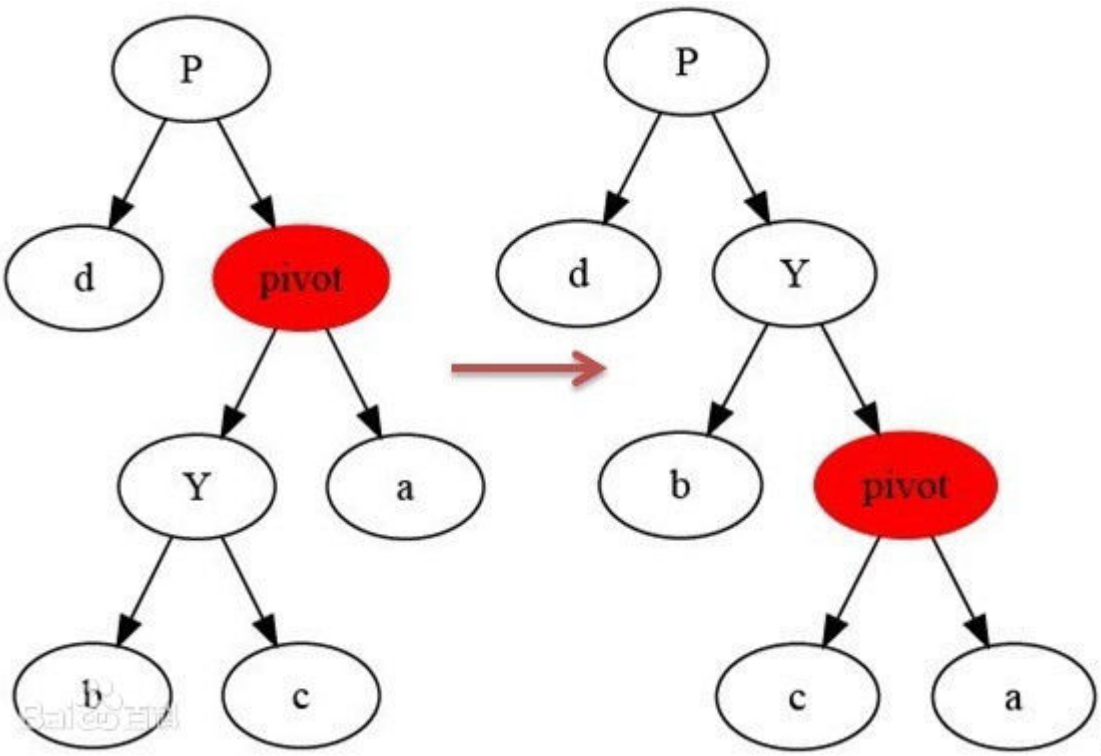

> 由于红黑树的实现较为复杂，因此现在工程中大多使用SBT树作为平衡二叉树的实现。

## 旋转——Rebalance

左旋:

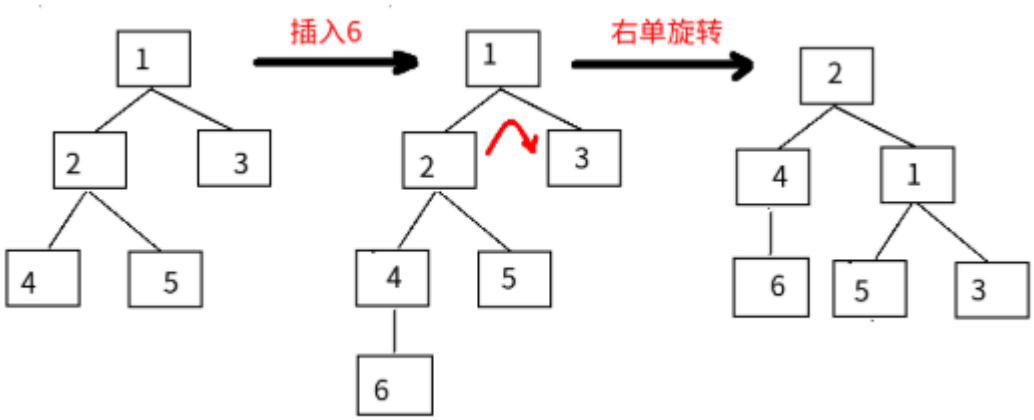



右旋：

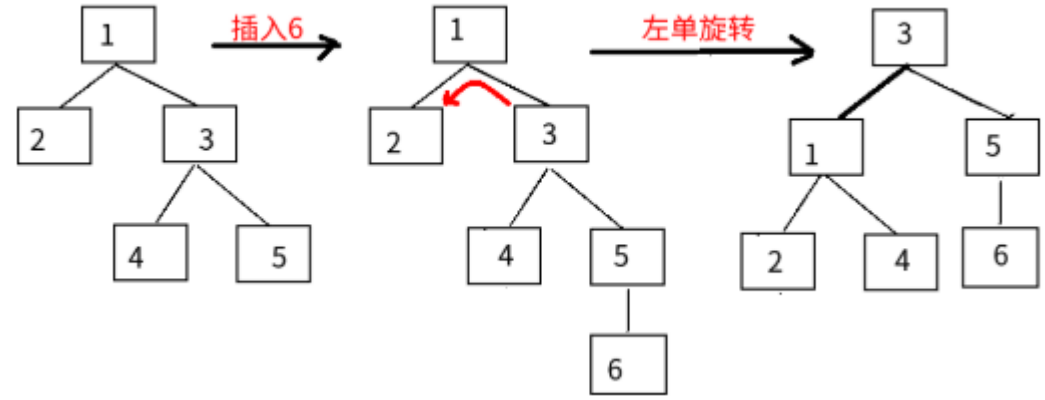



每种平衡二叉树都有自己的一套在插入、删除等操作改变树结构而破坏既定平衡性时的应对措施（但都是左旋操作和右旋操作的组合），以AVL数为例（有四种平衡调整操作，其中的数字只是结点代号而非结点数值）：

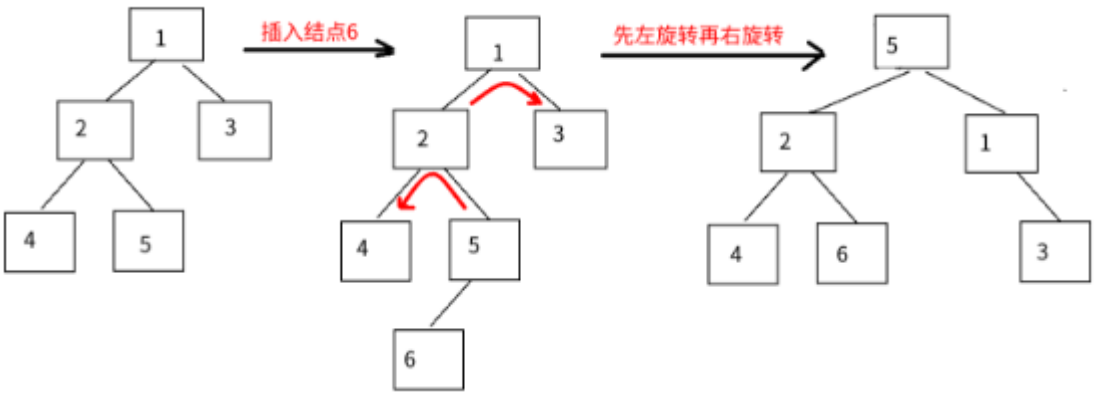
- `LL` 调整：2号结点的左孩子的左孩子导致整个树不平衡，2号结点右旋一次



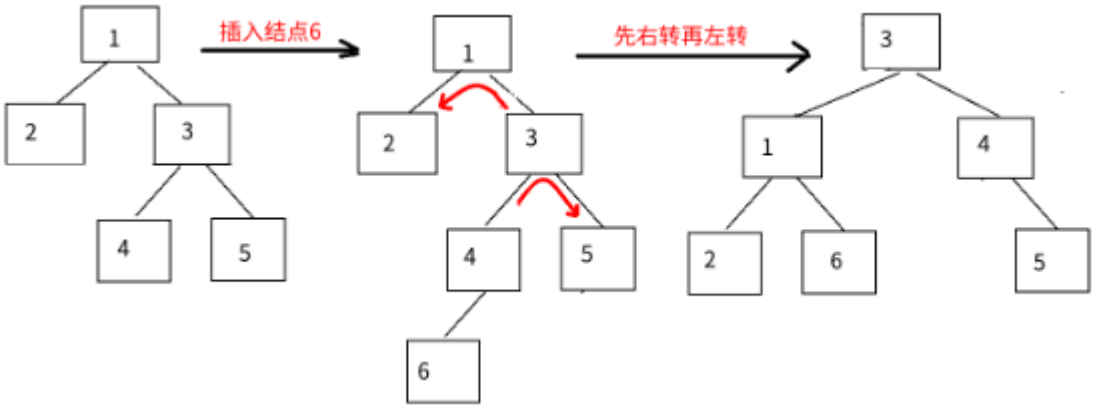

- `RR` 调整：3号结点的右孩子的右孩子导致树不平衡，3号结点左旋一次：




- `LR` 调整：先左后右




- `RL` 调整：先右后左：




> 红黑树的调整也是类似的，只不过调整方案更多。面试中一般不会让你手写红黑树（若有兴趣可参见文末附录），但我们一定能说清这些查找二叉树的性质，以及调整平衡的基本操作，再就是这些结构的使用。

## Java中红黑树的使用

Java中红黑树的实现有`TreeSet`和`TreeMap`，前者结点存储的是单一数据，而后者存储的是`<key,value>`的形式。

```
public static void main(String[] args) {
    TreeMap<Integer,String> treeMap = new TreeMap();
    treeMap.put(5, "tom");
    treeMap.put(11, "jack");
    treeMap.put(30,"tony");
    treeMap.put(18, "alice");
    treeMap.put(25, "jerry");

    //红黑树中最右边的结点
    System.out.println(treeMap.lastEntry());
    System.out.println(treeMap.lastKey());
    //红黑树最左边的结点
    System.out.println(treeMap.firstKey());
    //如果有13这个key，那么返回这条记录，否则返回树中比13大的key中最小的那一个
    System.out.println(treeMap.ceilingEntry(13));
    //如果有21这个key，那么返回这条记录，否则返回树中比21小的key中最大的那一个
    System.out.println(treeMap.floorEntry(21));
    //比11大的key中，最小的那一个
    System.out.println(treeMap.higherKey(11));
    //比25小的key中，最大的那一个
    System.out.println(treeMap.lowerKey(25));
    //遍历红黑树，是按key有序遍历的
    for (Map.Entry<Integer, String> record : treeMap.entrySet()) {
        System.out.println("age:"+record.getKey()+",name:"+record.getValue());
    }
}
```

`TreeMap`的优势是`key`在其中是有序组织的，因此增加、删除、查找`key`的时间复杂度均为`log(2,N)`。

## 案例

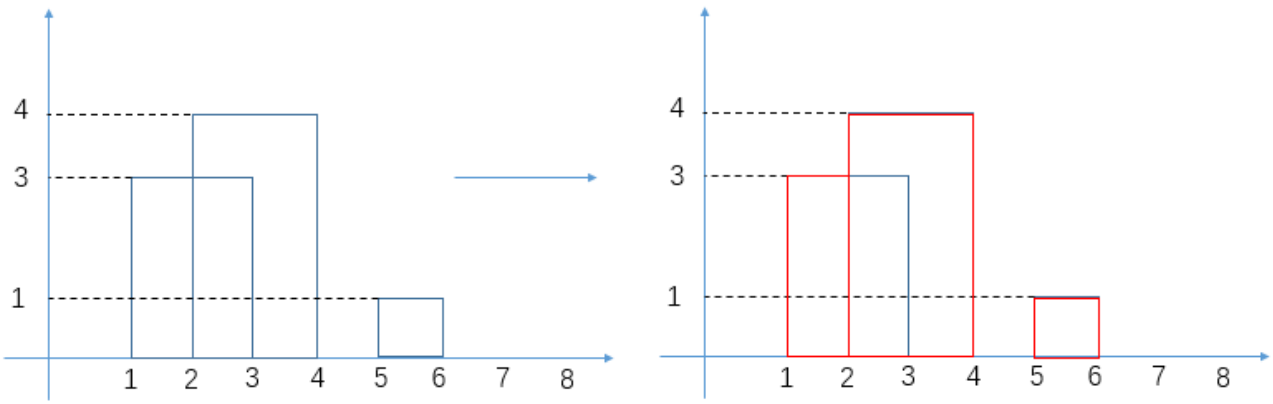
## The Skyline Problem

水平面上有 *N* 座大楼，每座大楼都是矩阵的形状，可以用一个三元组表示 `(start, end, height)`，分别代表其在x轴上的起点，终点和高度。大楼之间从远处看可能会重叠，求出 *N* 座大楼的外轮廓线。

外轮廓线的表示方法为若干三元组，每个三元组包含三个数字 (start, end, height)，代表这段轮廓的起始位置，终止位置和高度。

给出三座大楼：

```
[
  [1, 3, 3],
  [2, 4, 4],
  [5, 6, 1]
]
```




外轮廓线为：

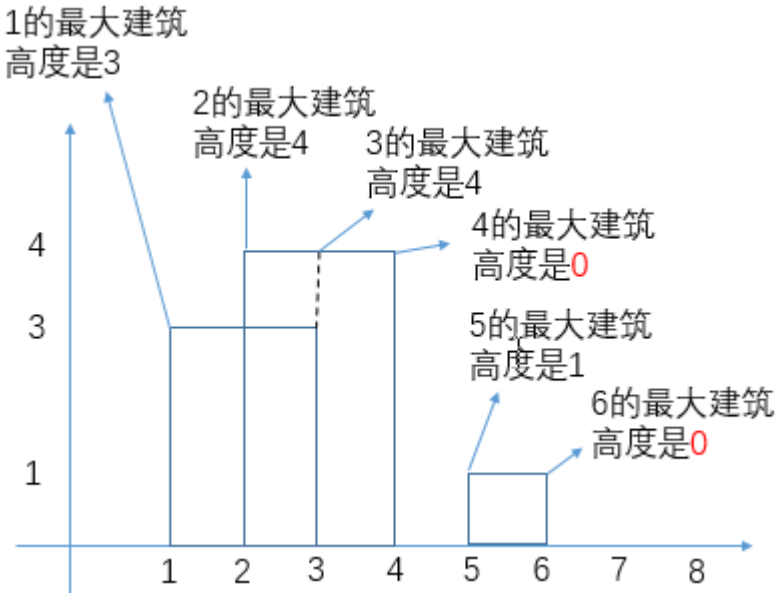
```
[
  [1, 2, 3],
  [2, 4, 4],
  [5, 6, 1]
]
```

**解析**：

1. 将一座楼的表示 `[start,end,height]` 拆分成左右两个边界（边界包含：所处下标、边界高度、是楼的左边界还是右边界），比如 `[1,3,3]` 就可以拆分成 `[1,3,true]` 和 `[3,3,false]` 的形式（`true` 代表左边界、`false` 代表右边界）。
2. 将每座楼都拆分成两个边界，然后对边界按照边界所处的下标进行排序。比如 `[[1,3,3],[2,4,4],[5,6,1]` 拆分之后为 `[[1,3,true],[3,3,false],[2,4,true],[,4,4,false],[5,1,true],[6,1,false]]`，排序后为 `[[1,3,true],[2,4,true],[3,3,false],[4,4,false],[5,1,true],[6,1,false]]`
3. 将边界排序后，遍历每个边界的高度并依次加入到一棵 `TreeMap` 红黑树中（记为 `countOfH`），以该高度出现的次数作为键值（第一次添加的高度键值为1），如果遍历过程中有重复的边界高度添加，要判断它是左边界还是右边界，前者直接将该高度在红黑树中的键值加1，后者则减1。以步骤2中排序后的边界数组为例，首先判断 `countOfH` 是否添加过边界 `[1,3,true]` 的高度 `3`，发现没有，于是 `put(3,1)`；接着对 `[2,4,true]`，`put[4,1]`；然后尝试添加 `[3,3,false]` 的 `3`，发现 `countOfH` 中添加过 `3`，而

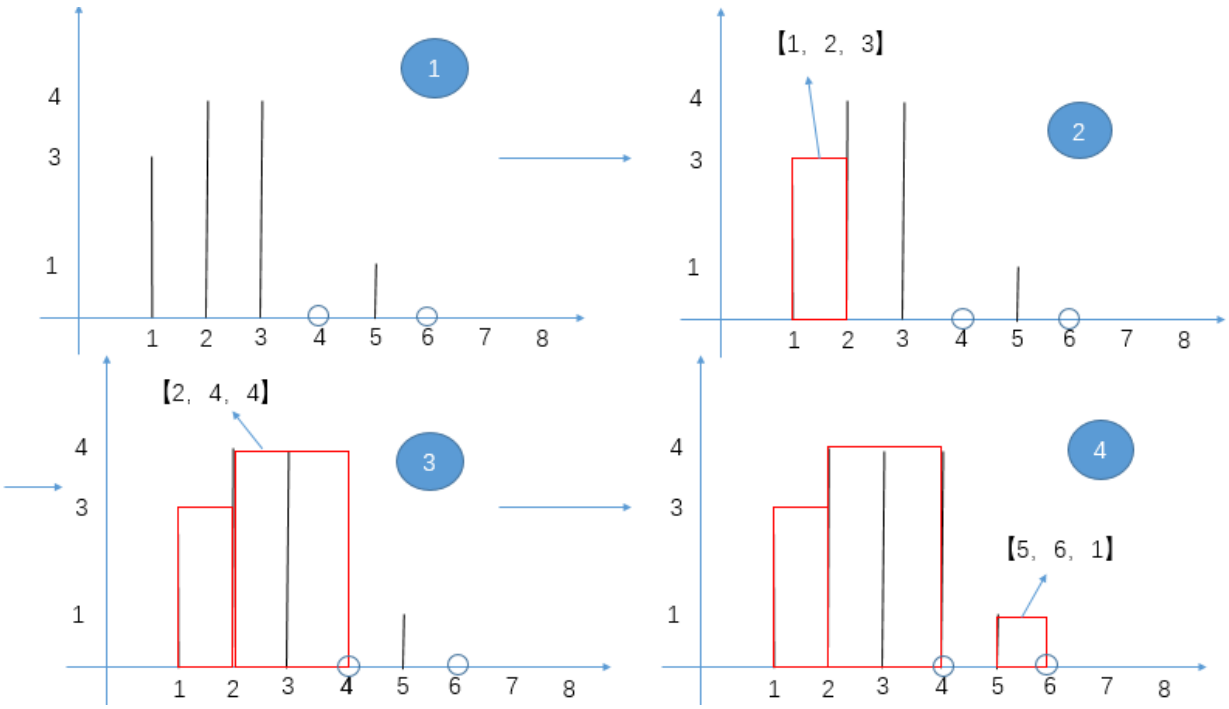
`[3,3,false]`是右边界，因此将`countOfH.get(3)`的次数减1，当`countOfH`中的记录的键值为0时直接移除，于是移除高度为3的这一条记录；……

对于遍历过程经过的每一个边界，我们还需要一棵`TreeMap`红黑树（记为`maxHOfPos`）来记录对我们后续求外轮廓线有用的信息，也就是每个边界所处下标的最大建筑高度：




这里有个细节要注意一下，那就是如果添加某个边界之后，`countOfH`树为空了，那么该边界所处下标的建筑高度要记为0，表示一片相邻建筑的结束，比如上图中下标为4和6的边界。这也是为了后续求外轮廓线提供判断的依据。

4. 遍历`maxHOfPos`中的记录，构造整个外轮廓线数组：




起初没有遍历边界时，记`start=0,height=0`，接着遍历边界，如果边界高度`curHeight!=height`如上图中的`1->2: height=0,curHeight=3`，那么记`start=1,height=3`表示第一条组外轮廓线的`start`和`height`，接下来就是确定它的`end`了。确定了一条轮廓线的`start`和`height`之后会有两种情况：下一组轮廓线和这一组是挨着的（如上图`2->3`）、下一组轮廓线和这一组是相隔的（如上图中`3->4`）。因此在遍历到边界`[index:2,H:4]`时，发现`curHeight=4 != height=3`，于是可以确定轮廓线

`start:1,heigth:3` 的 `end:2`。确定一条轮廓线后就要更新一下 `start=2,heigth=4` 表示下一组轮廓线的起始下标和高度，接着遍历到边界 `[index:3,H:4]`，发现 `curHeight=4=height` 于是跳过；接着遍历到边界 `[index:4,H:0]`，发现 `curHeight=0`，根据步骤3中的逻辑可知一片相邻的建筑到此结束了，因此轮廓线 `start:2,height:4` 的 `end=4`。

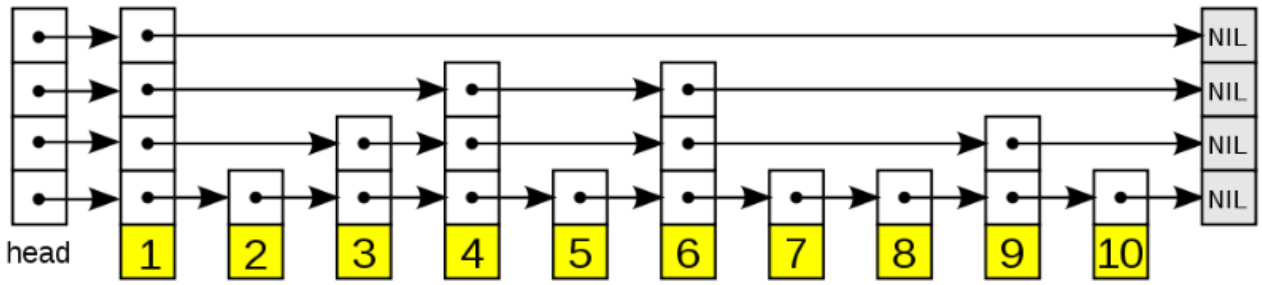
示例代码：

```
package top.zhenganwen.lintcode;

import java.util.*;

public class T131_The_SkylineProblem {

    public class Border implements Comparable<Border> {
        public int index;
        public int height;
        public boolean isLeft;

        public Border(int index, int height, boolean isLeft) {
            this.index = index;
            this.height = height;
            this.isLeft = isLeft;
        }

        @Override
        public int compareTo(Border border) {
            if (this.index != border.index) {
                return this.index - border.index;
            }
            if (this.isLeft != border.isLeft) {
                return this.isLeft ? -1 : 1;
            }
            return 0;
        }
    }

    /**
     * @param buildings: A list of lists of integers
     * @return: Find the outline of those buildings
     */
    public List<List<Integer>> buildingOutline(int[][] buildings) {
        //1、split one building to two borders and sort by border's index
        Border[] borders = new Border[buildings.length * 2];
        for (int i = 0; i < buildings.length; i++) {
            int[] oneBuilding = buildings[i];
            borders[i * 2] = new Border(oneBuilding[0], oneBuilding[2], true);
            borders[i * 2 + 1] = new Border(oneBuilding[1], oneBuilding[2],
false);
        }
        Arrays.sort(borders);

        //2、traversal borders and record the max height of each index

```

```java
        //key->height   value->the count of the height
        TreeMap<Integer, Integer> countOfH = new TreeMap<>();
        //key->index    value->the max height of the index
        TreeMap<Integer, Integer> maxHOfPos = new TreeMap<>();
        for (int i = 0; i < borders.length; i++) {
            int height = borders[i].height;
            if (!countOfH.containsKey(height)) {
                countOfH.put(height, 1);
            }else {
                int count = countOfH.get(height);
                if (borders[i].isLeft) {
                    countOfH.put(height, count + 1);
                } else {
                    countOfH.put(height, count - 1);
                    if (countOfH.get(height) == 0) {
                        countOfH.remove(height);
                    }
                }
            }

            if (countOfH.isEmpty()) {
                maxHOfPos.put(borders[i].index, 0);
            } else {
                //lastKey() return the maxHeight in countOfH RedBlackTree->log(2,N)
                maxHOfPos.put(borders[i].index, countOfH.lastKey());
            }
        }

        //3、draw the buildings outline according to the maxHOfPos
        int start = 0;
        int height = 0;
        List<List<Integer>> res = new ArrayList<>();
        for (Map.Entry<Integer, Integer> entry : maxHOfPos.entrySet()) {
            int curPosition = entry.getKey();
            int curMaxHeight = entry.getValue();
            if (height != curMaxHeight) {
                //if the height don't be reset to 0, the curPosition is the end
                if (height != 0) {
                    List<Integer> record = new ArrayList<>();
                    record.add(start);
                    record.add(curPosition);//end
                    record.add(height);

                    res.add(record);
                }
                //reset the height and start
                height = curMaxHeight;
                start = curPosition;
            }
        }
        return res;
    }
```

```
98
99      public static void main(String[] args) {
100         int[][] buildings = {
101                 {1, 3, 3},
102                 {2, 4, 4},
103                 {5, 6, 1}
104         };
105         System.out.println(new
    T131_The_SkylineProblem().buildingOutline(buildings));
106
107     }
108 }
```

# 跳表

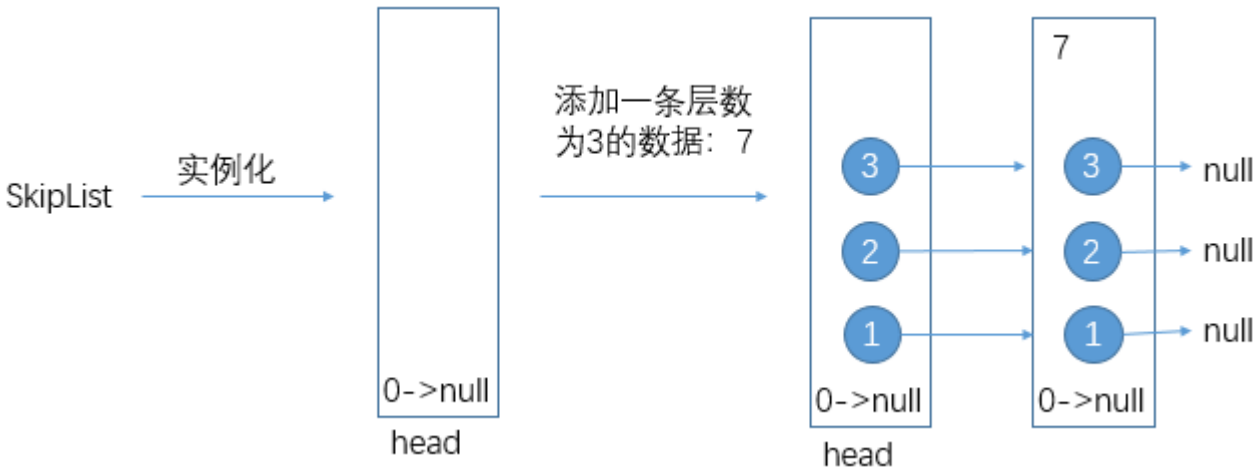



跳表有着和红黑树、SBT树相同的功能，都能实现在`O(log(2,N))`内实现对数据的增删改查操作。但跳表不是以二叉树为原型的，其设计细节如下：

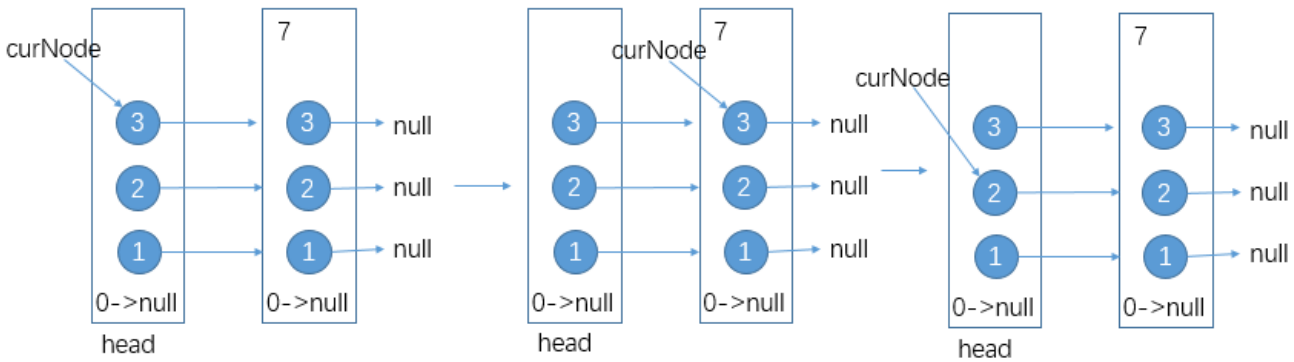
记该结构为`SkipList`，该结构中可以包含有很多结点（`SkipListNode`），每个结点代表一个被添加到该结构的数据项。当实例化`SkipList`时，该对象就会自带一个`SkipListNode`（不代表任何数据项的头结点）。

## 添加数据

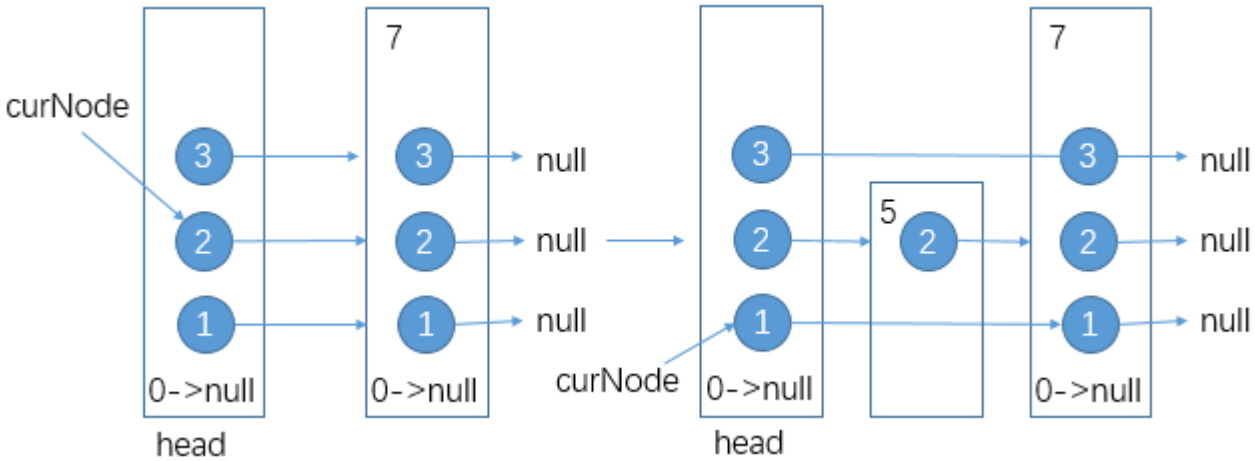
当你向其中添加数据之前，首先会抛硬币，将第一次出现正面朝上时硬币被抛出的次数作为该数据的层数（`level`，**最小为1**），接着将数据和其层数封装成一个`SkipListNode`添加到`SkipList`中。结构初始化时，其头结点的层数为0，但每次添加数据后都会更新头结点的层数为所添数据中层数最大的。比如实例化一个`SkipList`后向其中添加一条层数为`3`的数据`7`：



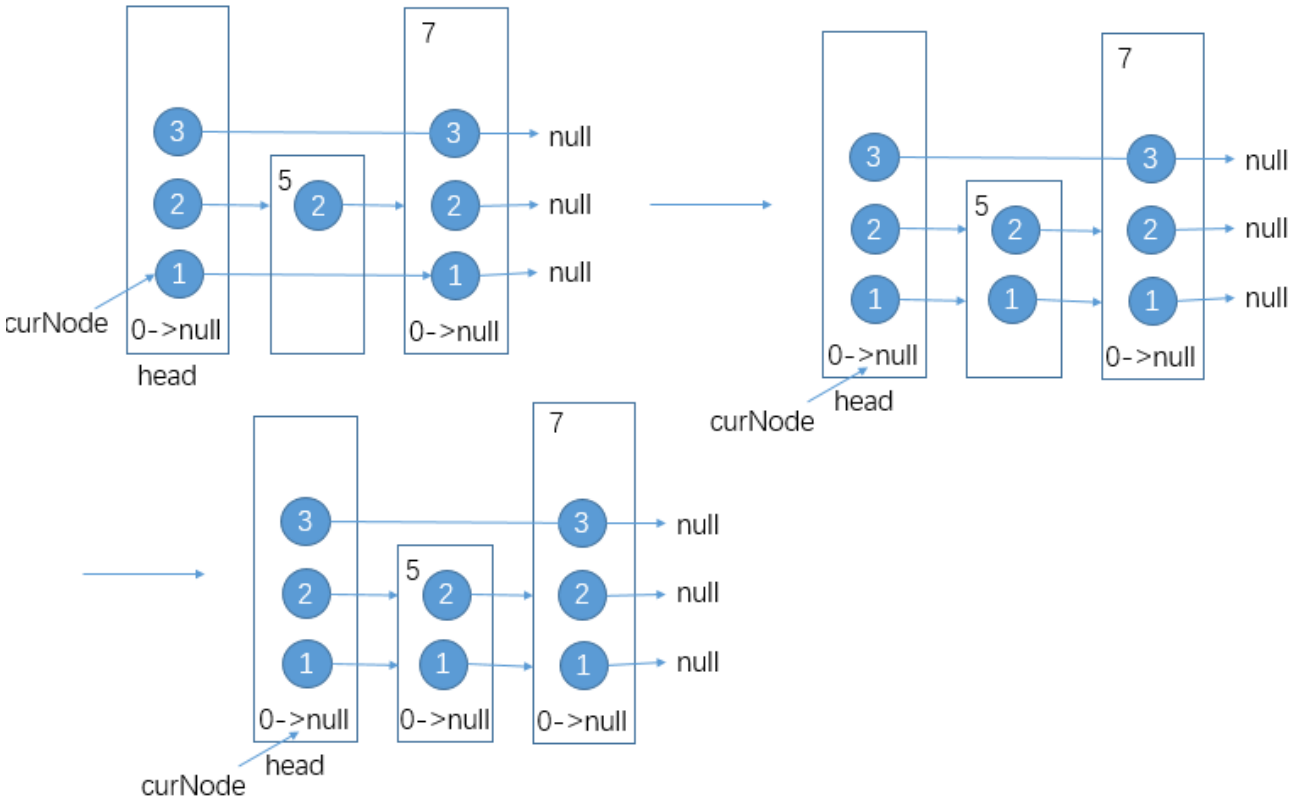

这时如果再添加一条层数为 `2` 的数据 `5` 呢？首先游标 `curNode` 会从 `head` 的最高层出发往右走，走到数据项为7的结点，发现 `7>5`，于是又退回来走向下一层：




接着再尝试往右走，还是发现 `7>5`，于是还是准备走向下一层，但此时发现 `curNode` 所在层数 `2` 是数据项 `5` 的最高层，于是先建出数据项 `5` 的第二层，`curNode` 再走向下一层：

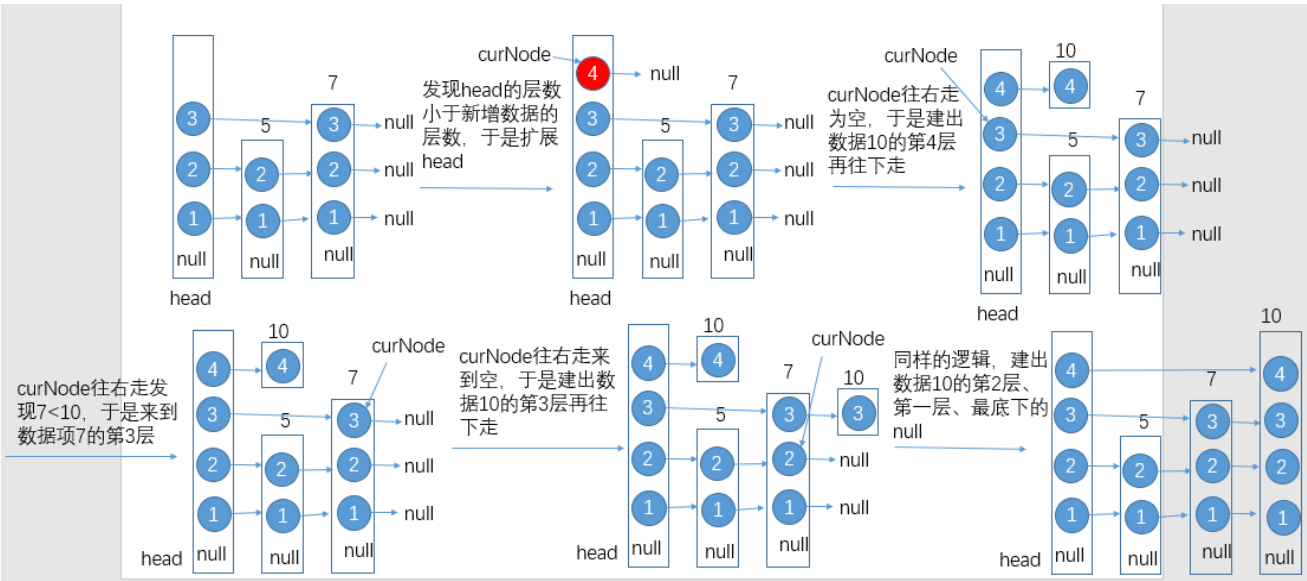



同样的，`curNode` 尝试往右走，但发现 `7>5`，`curNode` 所在层为1，但数据 `5` 的第一层还没建，于是建出，`curNode` 再往下走。当 `curNode` 走到 `null` 时，建出数据 `5` 根部的 `null`：




至此层数为2的数据项 `5` 的添加操作完毕。

那如果添加一个层数较高的数据项该如何处理呢？以添加层数为4的数据 `10` 为例：




添加操作对应的代码示例：

```
import java.util.ArrayList;

/**
 * A stored structure.Its add,delete,update,find operation are log(2,N)
 *
 * @author zhenganwen
 */
```

```java
public class SkipList {
    private SkipListNode head;
    private int maxLevel;
    private int size;
    public static final double PROBABILITY = 0.5;

    public SkipList() {
        this.head = new SkipListNode(Integer.MIN_VALUE);
        /**
         * the 0th level of each SkipListNode is null
         */
        this.head.nextNodes.add(null);
        this.maxLevel = 0;
        this.size = 0;
    }

    private class SkipListNode {
        int value;
        /**
         * nextNodes represent the all levels of a SkipListNode the element on
         * one index represent the successor SkipListNode on the indexth level
         */
        ArrayList<SkipListNode> nextNodes;

        public SkipListNode(int newValue) {
            this.value = newValue;
            this.nextNodes = new ArrayList<SkipListNode>();
        }
    }

    /**
     * put a new data into the structure->log(2,N)
     *
     * @param newValue
     */
    public void add(int newValue) {
        if (!contains(newValue)) {

            // generate the level
            int level = 1;
            while (Math.random() < PROBABILITY) {
                level++;
            }
            // update max level
            if (level > maxLevel) {
                int increment = level - maxLevel;
                while (increment-- > 0) {
                    this.head.nextNodes.add(null);
                }
                maxLevel = level;
            }
            // encapsulate value
            SkipListNode newNode = new SkipListNode(newValue);
```

```
            // build all the levels of new node
            SkipListNode cur = findInsertionOfTopLevel(newValue, level);
            while (level > 0) {
                if (cur.nextNodes.get(level) != null) {
                    newNode.nextNodes.add(0, cur.nextNodes.get(level));
                } else {
                    newNode.nextNodes.add(0, null);
                }
                cur.nextNodes.set(level, newNode);
                level--;
                cur = findNextInsertion(cur, newValue, level);
            }
            newNode.nextNodes.add(0, null);
            size++;
        }
    }

    /**
     * find the insertion point of the newNode's top level from head's maxLevel
     * by going right or down
     *
     * @param newValue newNode's value
     * @param level    newNode's top level
     * @return
     */
    private SkipListNode findInsertionOfTopLevel(int newValue, int level) {
        int curLevel = this.maxLevel;
        SkipListNode cur = head;
        while (curLevel >= level) {
            if (cur.nextNodes.get(curLevel) != null
                    && cur.nextNodes.get(curLevel).value < newValue) {
                // go right
                cur = cur.nextNodes.get(curLevel);
            } else {
                // go down
                curLevel--;
            }
        }
        return cur;
    }

    /**
     * find the next insertion from cur node by going right on the level
     *
     * @param cur
     * @param newValue
     * @param level
     * @return
     */
    private SkipListNode findNextInsertion(SkipListNode cur, int newValue,
                                           int level) {
        while (cur.nextNodes.get(level) != null
                && cur.nextNodes.get(level).value < newValue) {
```

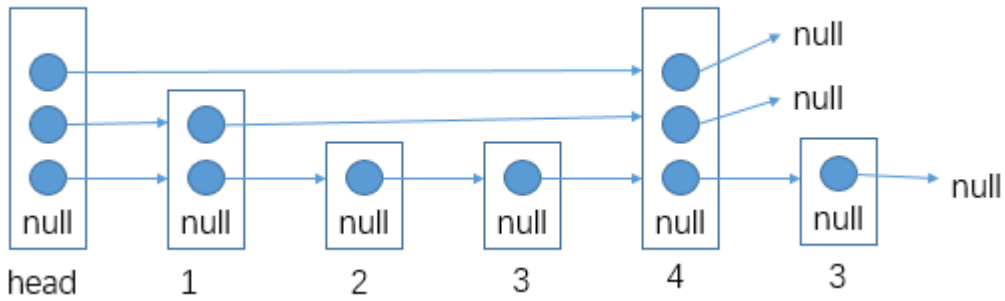
```
            cur = cur.nextNodes.get(level);
        }
        return cur;
    }

    /**
     * check whether a value exists->log(2,N)
     *
     * @param value
     * @return
     */
    public boolean contains(int value) {
        if (this.size == 0) {
            return false;
        }
        SkipListNode cur = head;
        int curLevel = maxLevel;
        while (curLevel > 0) {
            if (cur.nextNodes.get(curLevel) != null) {
                if (cur.nextNodes.get(curLevel).value == value) {
                    return true;
                } else if (cur.nextNodes.get(curLevel).value < value) {
                    cur = cur.nextNodes.get(curLevel);
                } else {
                    curLevel--;
                }
            } else {
                curLevel--;
            }
        }

        return false;
    }

    public static void main(String[] args) {
        SkipList skipList = new SkipList();
        skipList.add(1);
        skipList.add(2);
        skipList.add(3);
        skipList.add(4);
        skipList.add(5);
        //mark a break point here to check the memory structure of skipList
        System.out.println(skipList);
    }

}
```

## 查找数据

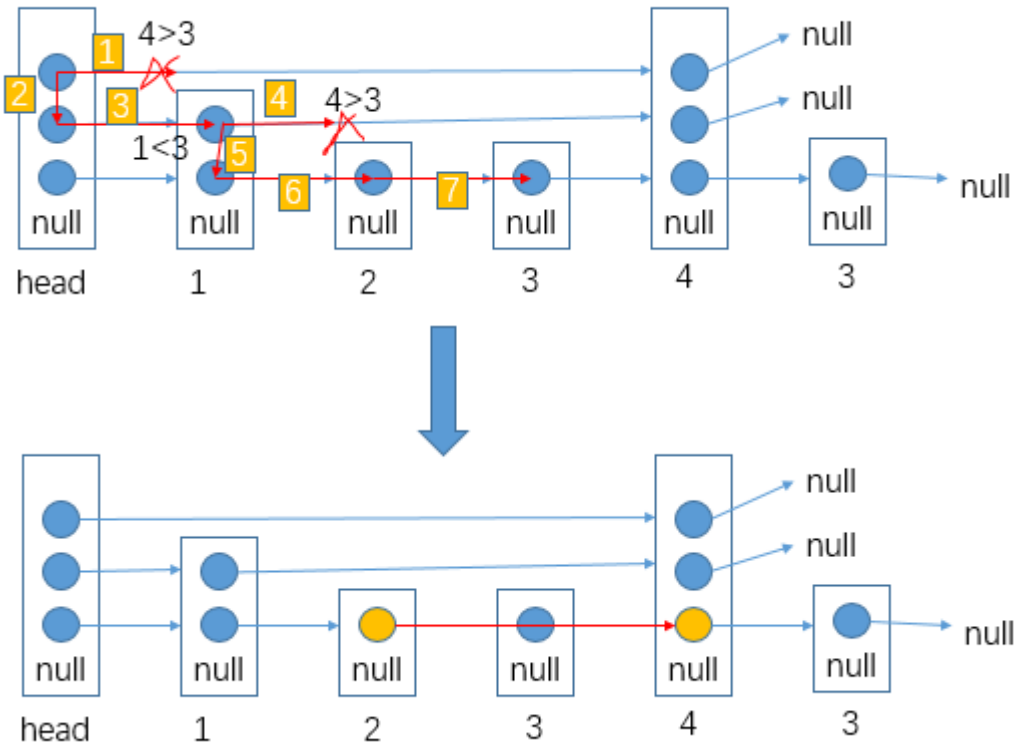
查找数据项的操作和添加数据项的步骤类似，也是游标`curNode`从`head`的最高层出发，每次先尝试向右走来到`nextNode`，如果`nextNode`封装的数据大于查找的目标`target`或`nextNode`为空，那么`curNode`回退并向下走；如果`nextNode`封装的数据小于`target`，那么`curNode`继续向右走，直到`curNode`走到的结点数据与`target`相同表示找到了，否则`curNode`走到了某一结点的根部`null`，那么说明结构中不存在该数据。`->contains()`

## 删除数据

了解添加数据的过程之后，删除数据其实就是将逻辑倒过来：解除该数据结点的前后引用关系。下图是我在写好上述`add()`方法后，向其中放入`1、2、3、4、5`后形成的结构：




如果此时删除数据`3`：




首先应该从`head`的最高层出发，通过向右或向下找到数据3的最高层（如图`2->3->5->6->7`），将该层移除整体结构并处理好该层上，其前后结点的关系。同样的逻辑，将数据`3`剩下的层移除。

示例代码：

```java
/**
     * delete skipListNode by the value
     *
     * @param value
     */
public void delete(int value) {
    //if exists
    if (contains(value)) {
        //find the node and its level
        SkipListNode deletedNode = head;
        int deletedLevels = maxLevel;
        //because exists,so must can find
        while (deletedLevels > 0) {
            if (deletedNode.nextNodes.get(deletedLevels) != null) {
                if (deletedNode.nextNodes.get(deletedLevels).value == value) {
                    deletedNode = deletedNode.nextNodes.get(deletedLevels);
                    break;
                } else if (deletedNode.nextNodes.get(deletedLevels).value < value) {
                    deletedNode = deletedNode.nextNodes.get(deletedLevels);
                } else {
                    deletedLevels--;
                }
            } else {
                deletedLevels--;
            }
        }
        //release the node and adjust the reference
        while (deletedLevels > 0) {
            SkipListNode pre = findInsertionOfTopLevel(value, deletedLevels);
            if (deletedNode.nextNodes.get(deletedLevels) != null) {
                pre.nextNodes.set(deletedLevels,
deletedNode.nextNodes.get(deletedLevels));
            } else {
                pre.nextNodes.set(deletedLevels, null);
            }
            deletedLevels--;
        }

        size--;
    }
}

public static void main(String[] args) {
    SkipList skipList = new SkipList();
    skipList.add(1);
    skipList.add(2);
    skipList.add(3);
    skipList.add(4);
    skipList.add(5);
    //mark a break point here to check the memory structure of skipList
    skipList.delete(3);
    System.out.println(skipList);
}
```

## 遍历数据

需要遍历跳表中的数据时，我们可以根据每个数据的层数至少为1的特点（每个结点的第一层引用的是比该结点数据大的结点中数据最小的结点）。

示例代码：

```
class SkipListIterator implements Iterator<Integer> {
    private SkipListNode cur;
    public SkipListIterator(SkipList skipList) {
        this.cur = skipList.head;
    }

    @Override
    public boolean hasNext() {
        return cur.nextNodes.get(1) != null;
    }

    @Override
    public Integer next() {
        int value = cur.nextNodes.get(1).value;
        cur = cur.nextNodes.get(1);
        return value;
    }
}

@Override
public String toString() {
    SkipListIterator iterator = new SkipListIterator(this);
    String res = "[ ";
    while (iterator.hasNext()) {
        res += iterator.next()+" ";
    }
    res += "]";
    System.out.println();
    return res;
}

public static void main(String[] args) {
    SkipList skipList = new SkipList();
    skipList.add(1);
    skipList.add(2);
    skipList.add(3);
    skipList.add(4);
    skipList.add(5);
    System.out.println(skipList);
    skipList.delete(3);
    System.out.println(skipList);
}
```

# 从暴力尝试到动态规划

动态规划不是玄学，也无需去记那些所谓的刻板的“公式”（例如状态转换表达式等），其实动态规划是从暴力递归而来。并不是说一个可以动态规划的题一上来就可以写出动态规划的求解步骤，我们只需要能够写出暴力递归版本，然后对重复计算的子过程结果做一个缓存，最后分析状态依赖寻求最优解，即衍生成了动态规划。本节将以多个例题示例，展示求解过程是如何从暴力尝试，一步步到动态规划的。

## 换钱的方法数

**题目**：给定数组arr，arr中所有的值都为正数且不重复。每个值代表一种面值的货币，每种面值的货币可以使用任意张，再给定一个整数aim代表要找的钱数，求换钱有多少种方法。

举例：`arr=[5,10,25,1], aim=0`：成0元的方法有1种，就是所有面值的货币都不用。所以返回1。`arr=[5,10,25,1], aim=15`：组成15元的方法有6种，分别为3张5元、1张10元+1张5元、1张10元+5张1元、10张1元+1张5元、2张5元+5张1元和15张1元。所以返回6。`arr=[3,5], aim=2`：任何方法都无法组成2元。所以返回0。

### 暴力尝试

我们可以将该题要求解的问题定义成一个过程：对于下标`index`，`arr`中在`index`及其之后的所有面值不限张数任意组合，该过程最终返回所有有效的组合方案。因此该过程可以描述为`int process(int arr[],int index,int aim)`，题目的解就是调用`process(arr,0,aim)`。那么函数内部具体该如何解决此问题呢?

其实所有面值不限张数的任意组合就是对每一个面值需要多少张的一个**决策**，那我们不妨从碰到的第一个面值开始决策，比如`arr=[5,10,25,1], aim=15`时，（选0张5元之后剩下的面值不限张数组合成15元的方法数 + 选1张5元之后剩下的面值不限张数组合成10元方法数 + 选2张5元之后剩下的面值不限张数组合成5元方法数 + 选3张5元之后剩下的面值不限张数组合成0元方法数）就是所给参数对应的解，其中“剩下的面值不限张数组合成一定的钱数”又是同类问题，可以使用相同的过程求解，因此有了如下的暴力递归：

```java
/**
     * arr中的每个元素代表一个货币面值，使用数组index及其之后的面值（不限张数）
     * 拼凑成钱数为aim的方法有多少种，返回种数
     * @param arr
     * @param index
     * @param aim
     * @return
     */
public static int process(int arr[], int index, int aim) {
    if (index == arr.length) {
        return aim == 0 ? 1 : 0;
    }
    int res = 0;
    //index位置面值的决策，从0张开始
    for (int zhangshu = 0; arr[index] * zhangshu <= aim; zhangshu++) {
        res += process(arr, index + 1, aim - (arr[index] * zhangshu));
    }
    return res;
}

public static int swapMoneyMethods(int arr[], int aim) {
```

```
    if (arr == null) {
        return 0;
    }
    return process(arr, 0, aim);
}

public static void main(String[] args) {
    int arr[] = {5, 10, 25, 1};
    System.out.println(swapMoneyMethods(arr, 15));
}
```

## 缓存每个状态的结果，以免重复计算

上述的暴力递归是极其暴力的，比如对于参数 `arr=[5, 3, 1, 30, 15, 20, 10], aim=100` 来说，如果已经决策了 `3张5元+0张3元+0张1元` 的接着会调子过程 `process(arr, 3, 85)`；如果已经决策了 `0张5元+5张3元+0张1元` 接着也会调子过程 `process(arr, 3, 85)`；如果已经决策了 `0张5元+0张3元+15张1元` 接着还是会调子过程 `process(arr, 3, 85)`。

你会发现，这个已知面额种类和要凑的钱数，求凑钱的方法的解是固定的。也就是说不管之前的决策是3张5元的，还是5张3元的，又或是15张1元的，对后续子过程的 `[30, 15, 20, 10]` 凑成 `85` 这个问题的解是不影响的，这个解该是多少还是多少。这也是**无后效性问题**。无后效性问题就是某一状态的求解不依赖其他状态，比如著名的N皇后问题就是有后效性问题。

因此，我们不妨再求解一个状态之后，将该状态对应的解做个缓存，在后续的状态求解时先到缓存中找是否有该状态的解，有则直接使用，没有再求解并放入缓存，这样就不会有重复计算的情况了：

```
public static int swapMoneyMethods(int arr[], int aim) {
    if (arr == null) {
        return 0;
    }
    return process2(arr, 0, aim);
}

/**
* 使用哈希表左缓存容器
* key是某个状态的代号，value是该状态对应的解
*/
static HashMap<String,Integer> map = new HashMap();

public static int process2(int arr[], int index, int aim) {
    if (index == arr.length) {
        return aim == 0 ? 1 : 0;
    }
    int res = 0;
    for (int zhangshu = 0; arr[index] * zhangshu <= aim; zhangshu++) {
        //使用index及其之后的面值拼凑成aim的方法数这个状态的代号：index_aim
        String key = String.valueOf(index) + "_" + String.valueOf(aim);
        if (map.containsKey(key)) {
            res += map.get(key);
        } else {
            int value = process(arr, index + 1, aim - (arr[index] * zhangshu));
```

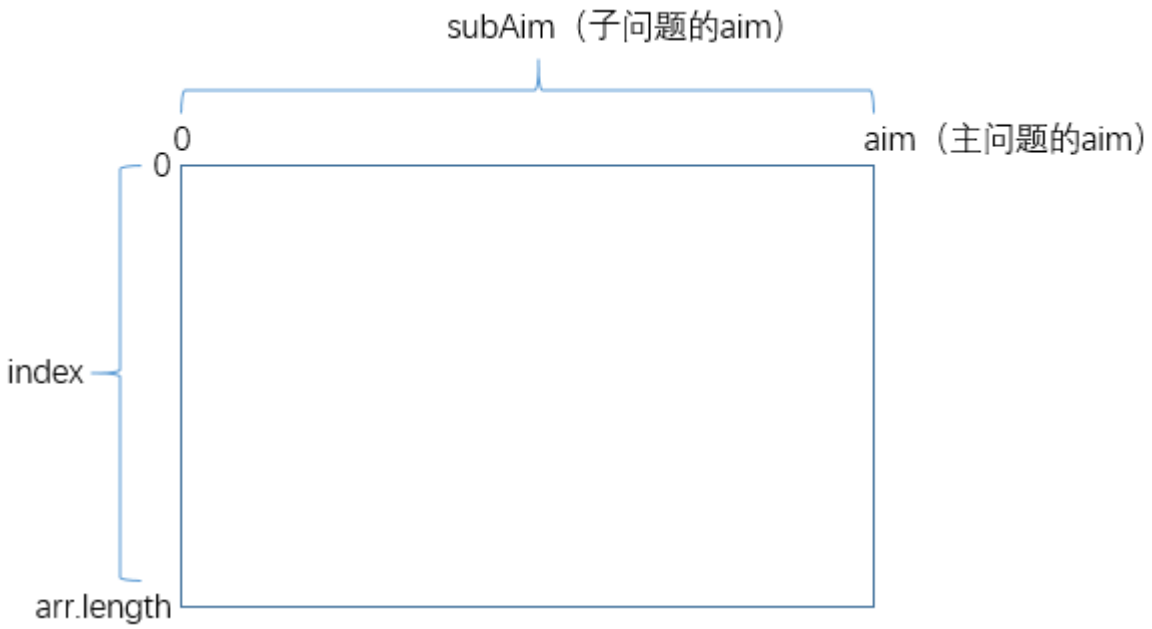
```
            key = String.valueOf(index + 1) + "_" + String.valueOf(aim - (arr[index]
* zhangshu));
            map.put(key, value);
            res += value;
        }
    }
    return res;
}

public static void main(String[] args) {
    int arr[] = {5, 10, 25, 1};
    System.out.println(swapMoneyMethods(arr, 15));
}
```

## 确定依赖关系，寻找最优解

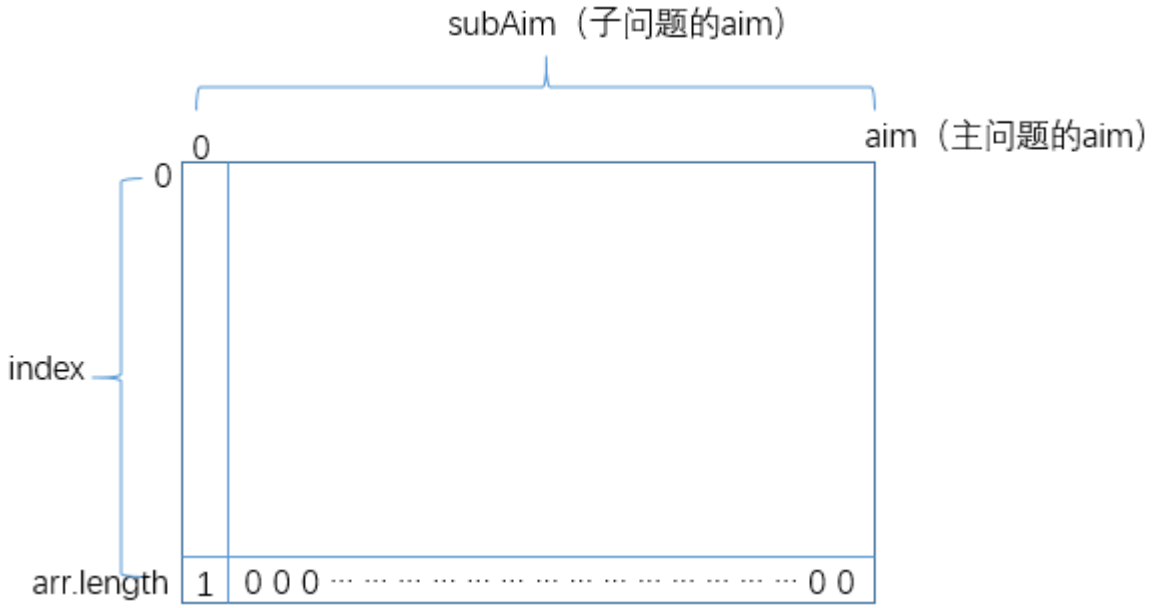
当然，借助缓存已经将暴力递归的时间复杂度拉低了很多，但这还不是最优解。下面我们将以寻求最优解为引导，挖掘出动态规划中的状态转换。

从暴力尝试到动态规划，我们只需观察暴力尝试版本的代码，甚至可以忘却题目，按照下面高度套路化的步骤，就可以轻易改出动态规划：

1. 首先每个状态都有两个参数 `index` 和 `aim` （ `arr` 作为输入参数是不变的），因此可以对应两个变量的变化范围建立一张二维表：



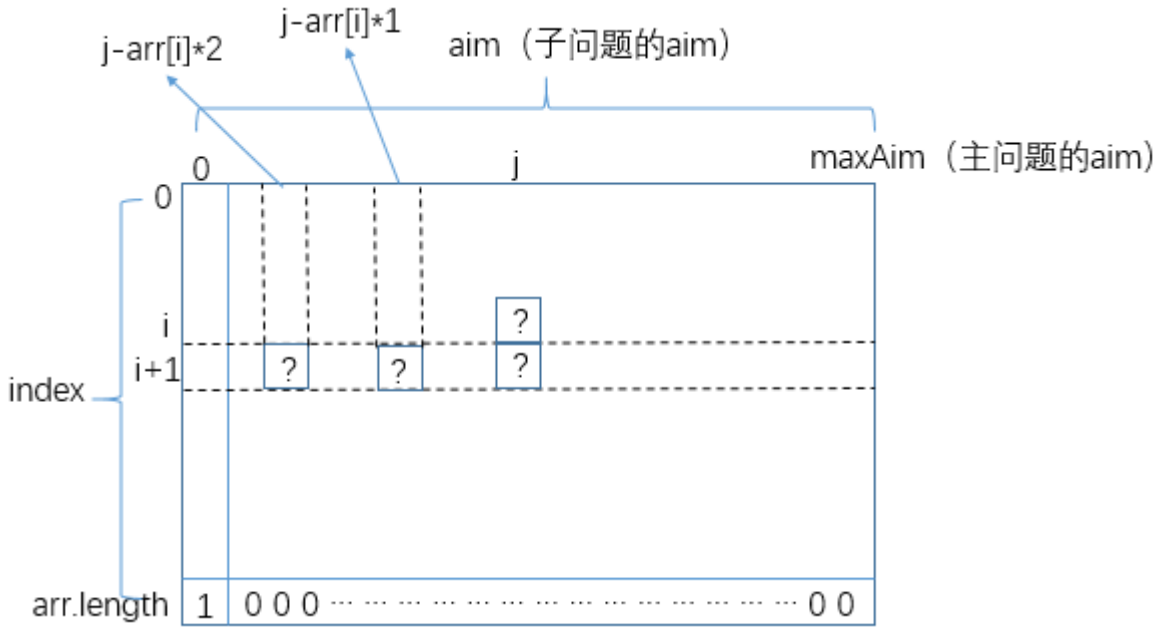

2. 从 `base case` 中找出特殊位置的解。比如 `if(index==arr.length) return aim==0?1:0` ，那么上述二维表的最后一行对应的所有状态可以直接求解：




3. 从暴力递归中找出普遍位置对应的状态所依赖的其他状态。比如：

```
for (int zhangshu = 0; arr[index] * zhangshu <= aim; zhangshu++) {
    res += process(arr, index + 1, aim - (arr[index] * zhangshu));
}
```

那么对于二维表中的一个普遍位置`(i,j)`，它所依赖的状态如下所示：



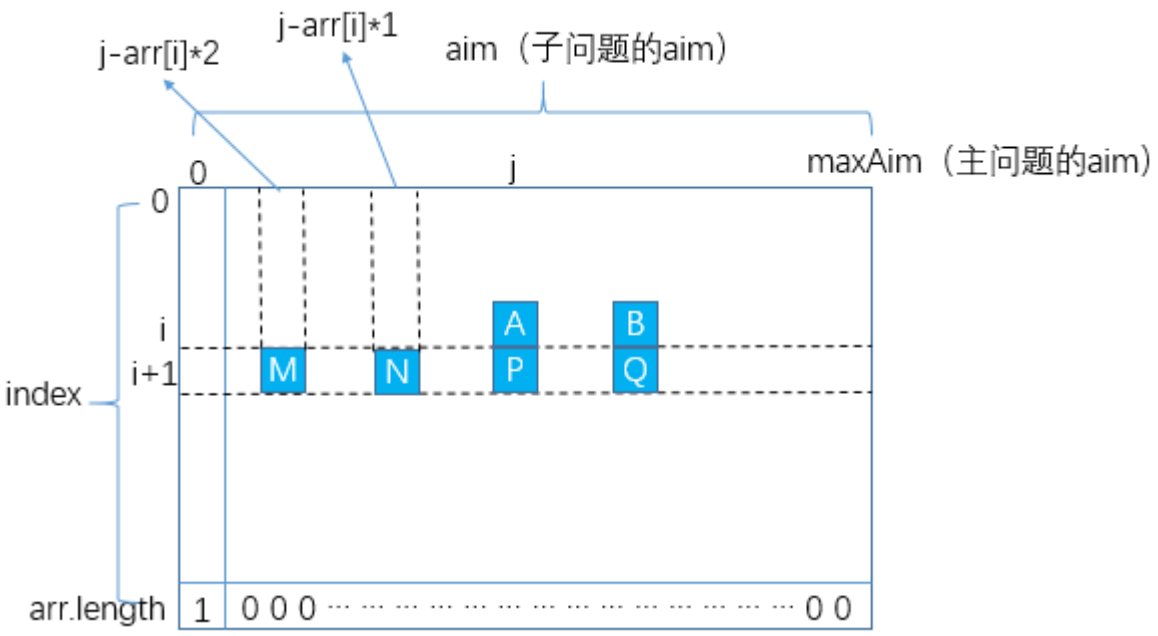

也就是说一个普遍位置的状态依赖它的下一行的几个位置上的状态。那么我们已经知道了最后一行所有位置上的状态，当然可以根据这个依赖关系推出倒数第二行的，继而推出倒数第三行的……整个二维表的所有位置上的状态都能推出来。

4. 找出主问题对应二维表的哪个状态（`(0,maxAim)`），那个状态的值就是问题的解。

示例代码：

```java
public static int maxMethodsDp(int arr[], int aim) {
    //二维表
    int dp[][] = new int[arr.length + 1][aim + 1];
    //base case
    dp[arr.length][0] = 1;
    //从倒数第二行开始推，推出整个二维表每个位置的状态
    for (int i = arr.length - 1; i >= 0; i--) {
        for (int j = 0; j <= aim; j++) {
            //i对应的面值取0张
            dp[i][j] = dp[i + 1][j];
            //i对应的面值取1张、2张、3张……
            for (int subAim = j - arr[i]; subAim >= 0; subAim = subAim - arr[i]) {
                dp[i][j] += dp[i + 1][subAim];
            }
        }
    }

    return dp[0][aim];
}

public static void main(String[] args) {
    int arr[] = {5, 10, 25, 1};
    System.out.println(maxMethodsDp(arr, 15));
}
```

到这里也许你会送一口气，终于找到了最优解，其实不然，因为如果你再分析一下每个状态的求解过程，仍然存在瑕疵：




比如你在求解状态A时，可能会将其依赖的状态M,N,P的值累加起来；然后在求解状态B时，有需要将其依赖的状态M,N,P,Q累加起来，你会发现在这个过程中`M+N+P`的计算是重复的，因此还可以有如下优化：

```java
for (int i = arr.length - 1; i >= 0; i--) {
    for (int j = 0; j <= aim; j++) {
        dp[i][j] = dp[i + 1][j];
        if (j - arr[i] >= 0) {
            dp[i][j] += dp[i][j - arr[i]];
        }
    }
}
```

至此，此题最优解的求解完毕。

# 排成一条线的纸牌博弈问题

**题目：**给定一个整型数组arr，代表分数不同的纸牌排成一条线。玩家A和玩家B依次拿走每张纸牌，规定玩家A先拿，玩家B后拿，但是每个玩家每次只能拿走最左或最右的纸牌，玩家A和玩家B都绝顶聪明。请返回最后获胜者的分数。

**举例：**`arr=[1,2,100,4]`。开始时玩家A只能拿走1或4。如果玩家A拿走1，则排列变为`[2,100,4]`，接下来玩家B可以拿走2或4，然后继续轮到玩家A。如果开始时玩家A拿走4，则排列变为`[1,2,100]`，接下来玩家B可以拿走1或100，然后继续轮到玩家A。玩家A作为绝顶聪明的人不会先拿4，因为拿4之后，玩家B将拿走100。所以玩家A会先拿1，让排列变为`[2,100,4]`，接下来玩家B不管怎么选，100都会被玩家A拿走。玩家A会获胜，分数为101。所以返回101。`arr=[1,100,2]`。开始时玩家A不管拿1还是2，玩家B作为绝顶聪明的人，都会把100拿走。玩家B会获胜，分数为100。所以返回100。

> 动态规划的题难就难在暴力尝试这个“试”法，只要能够试出了暴力版本，那改为动态规划就是高度套路的。

## 暴力尝试

```java
public static int maxScoreOfWinner(int arr[]) {
    if (arr == null) {
        return 0;
    }
    return Math.max(
        f(arr, 0, arr.length-1),
        s(arr, 0, arr.length-1));
}

public static int f(int arr[], int beginIndex, int endIndex) {
    if (beginIndex == endIndex) {
        return arr[beginIndex];
    }
    return Math.max(
        arr[beginIndex] + s(arr, beginIndex + 1, endIndex),
        arr[endIndex] + s(arr, beginIndex, endIndex - 1));
}

public static int s(int arr[], int beginIndex, int endIndex) {
    if (beginIndex == endIndex) {
        return 0;
    }
    return Math.min(
```

```
        f(arr, beginIndex + 1, endIndex),
        f(arr, beginIndex, endIndex - 1));
}

public static void main(String[] args) {
    int arr[] = {1, 2, 100, 4};
    System.out.println(maxScoreOfWinner(arr));//101
}
```

这个题的试法其实很不容易，笔者直接看别人写出的暴力尝试版本表示根本看不懂，最后还是搜了博文才弄懂。其中`f()`和`s()`就是整个尝试中的思路，与以往穷举法的暴力递归不同，这里是两个函数相互递归调用。

`f(int arr[],int begin,int end)`表示如果纸牌只剩下标在`begin~end`之间的几个了，那么作为先拿者，纸牌被拿完后，先拿者能达到的最大分数；而`s(int arr[],int begin,int end)`表示如果纸牌只剩下标在`begin~end`之间的几个了，那么作为后拿者，纸牌被拿完后，后拿者能达到的最大分数。

在`f()`中，如果只有一张纸牌，那么该纸牌分数就是先拿者能达到的最大分数，直接返回，无需决策。否则先拿者A的第一次决策只有两种情况：

- 先拿最左边的`arr[beginIndex]`，那么在A拿完这一张之后就会作为后拿者参与到剩下的`(begin+1)~end`之间的纸牌的决策了，这一过程可以交给`s()`来做。
- 先拿最右边的`arr[endIndex]`，那么在A拿完这一张之后就会作为后拿者参与到剩下的`begin~(end-1)`之间的纸牌的决策了，这一过程可以交给`s()`来做。

最后返回两种情况中，**结果较大**的那种。

在`s()`中，如果只有一张纸牌，那么作为后拿者没有纸牌可拿，分数为0，直接返回。否则以假设的方式巧妙的将问题递归了下去：

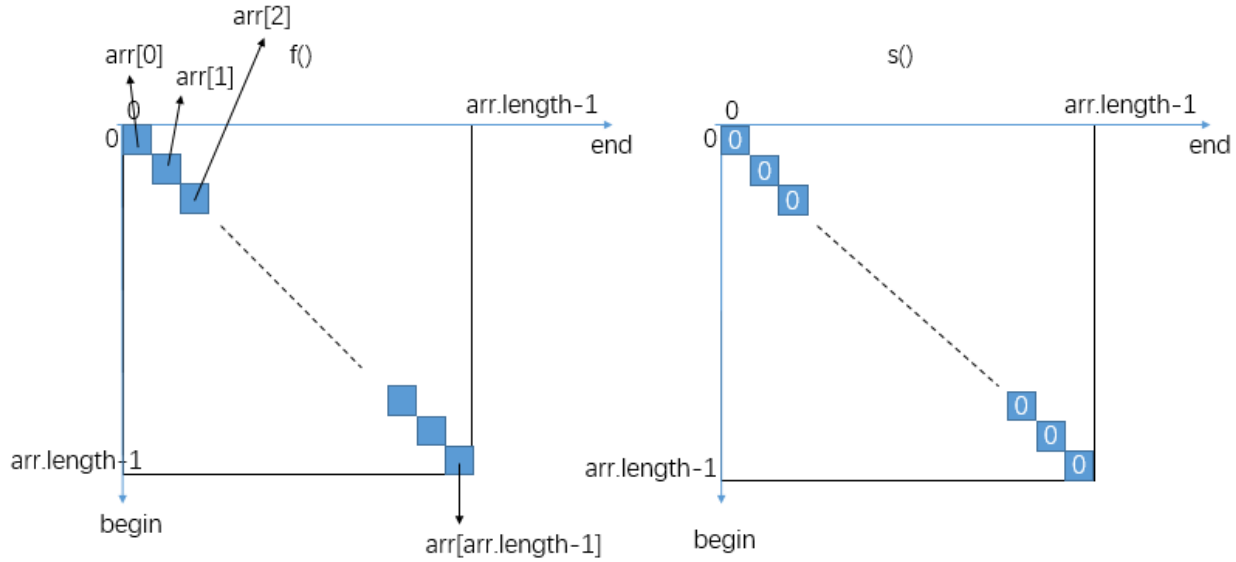
- 假设先拿者A拿到了`arr[beginIndex]`，那么去掉该纸牌后，对于剩下的`(begin+1)~end`之间的纸牌，后拿者B就转变身份成了先拿者，这一过程可以交给`f()`来处理。
- 假设先拿者A拿到了`arr[endIndex]`，那么去掉该纸牌后，对于剩下的`begin~(end-1)`之间的纸牌，后拿者B就转变身份成了先拿者，这一过程可以交给`f()`来处理。

这里取两种情况中**结果较小**的一种，是因为这两种情况是我们假设的，但先拿者A绝顶聪明，他的选择肯定会让后拿者尽可能拿到更小的分数。比如`arr=[1,2,100,4]`，虽然我们的假设有先拿者拿`1`和拿`4`两种情况，对应`f(arr,1,3)`和`f(arr,0,2)`，但实际上先拿者不会让后拿者拿到`100`，因此取两种情况中结果较小的一种。

## 改动态规划

这里是两个函数相互递归，每个函数的参数列表又都是`beginIndex`和`endIndex`是可变的，因此需要两张二维表保存`(begin,end)`确定时，`f()`和`s()`的状态值。

1. 确定`base case`对应的特殊位置上的状态值：




可以发现两张表的对角线位置上的状态值都是可以确定的，`begin<=end`，因此对角线左下方的区域不用管。

2. 由递归调用逻辑找出状态依赖。

`f()`依赖的状态：

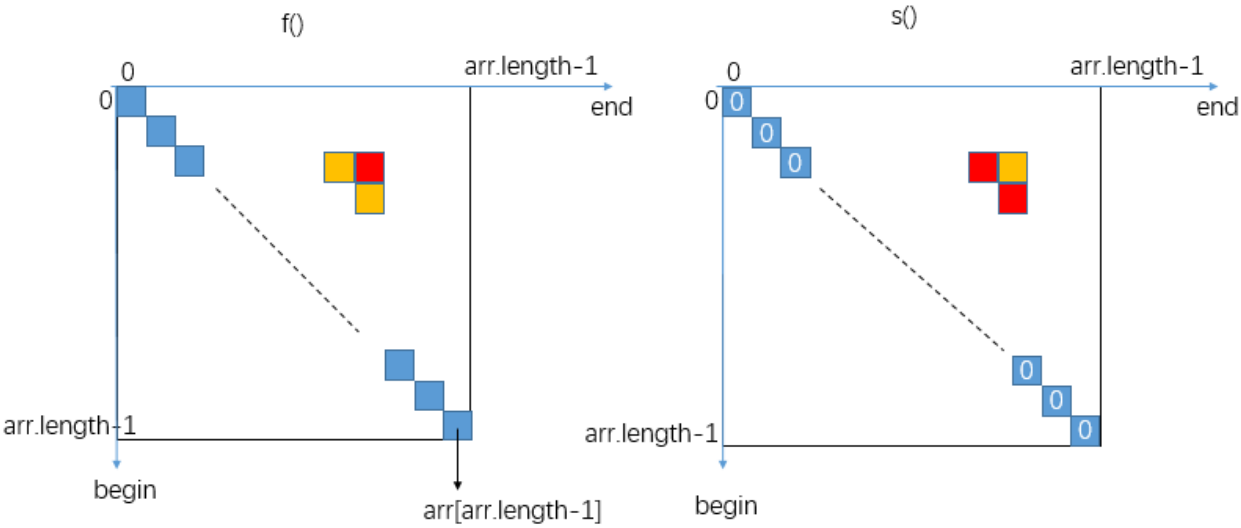
```
return Math.max(
                arr[beginIndex] + s(arr, beginIndex + 1, endIndex),
                arr[endIndex] + s(arr, beginIndex, endIndex - 1));
```

F表的`(begin,end)`依赖S表`(begin+1,end)`和`(begin,end-1)`。

`s()`依赖的状态：

```
return Math.min(
                f(arr, beginIndex + 1, endIndex),
                f(arr, beginIndex, endIndex - 1));
```

S表的`(begin,end)`依赖F表的`(begin+1,end)`和`(begin,end-1)`。



如此的话，对于对角线的右上区域，对角线位置上的状态能推出倒数第二长对角线位置上的状态，进而推出倒数第三长位置上的状态……右上区域每个位置的状态都能推出。

3. 确定主问题对应的状态：

```
return Math.max(
                f(arr, 0, arr.length-1),
                s(arr, 0, arr.length-1));
```

示例代码：

```
public static int maxScoreOfWinnerDp(int arr[]) {
    if (arr == null || arr.length == 0) {
        return 0;
    }

    int F[][] = new int[arr.length][arr.length];
    int S[][] = new int[arr.length][arr.length];
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
            if (i == j) {
                F[i][i] = arr[i];
            }
        }
    }
    //依次推出每条对角线，一共n-1条
    for (int i = 1; i < arr.length; i++) {
        for (int row = 0; row < arr.length - i; row++) {
            int col = row + i;
            F[row][col] = Math.max(arr[row] + S[row + 1][col], arr[col] + S[row][col - 1]);
            S[row][col] = Math.min(F[row + 1][col], F[row][col - 1]);
        }
    }

    return Math.max(F[0][arr.length - 1], S[0][arr.length - 1]);
}

public static void main(String[] args) {
    int arr[] = {1, 2, 100, 4};
    System.out.println(maxScoreOfWinnerDp(arr));
}
```

代码优化：

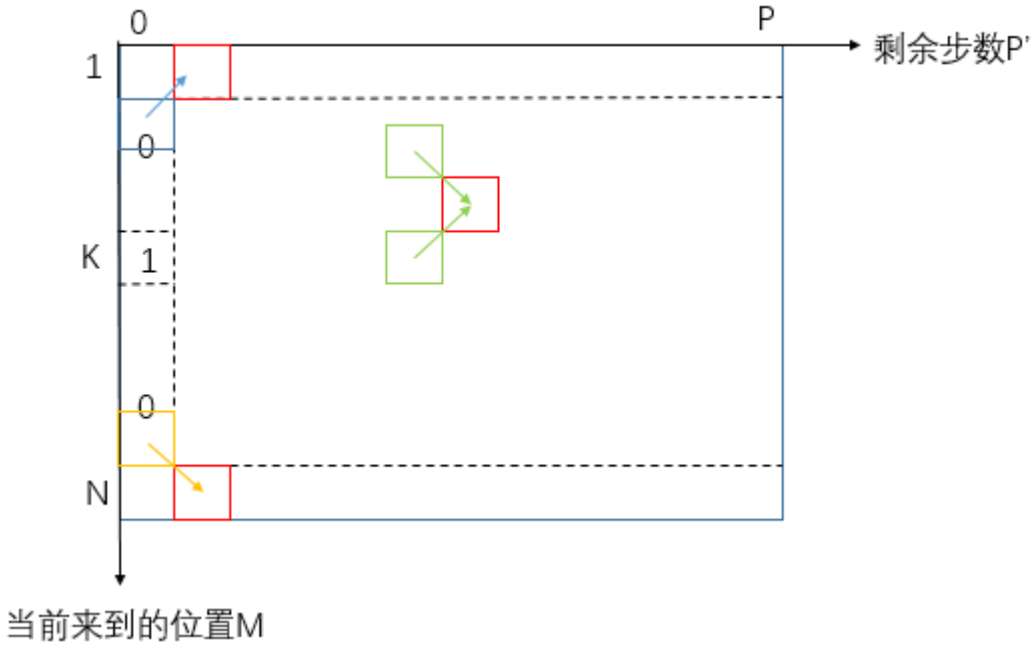
```
if (arr == null || arr.length == 0) {
    return 0;
}
int[][] f = new int[arr.length][arr.length];
int[][] s = new int[arr.length][arr.length];
for (int j = 0; j < arr.length; j++) {
    f[j][j] = arr[j];
    for (int i = j - 1; i >= 0; i--) {
        f[i][j] = Math.max(arr[i] + s[i + 1][j], arr[j] + s[i][j - 1]);
        s[i][j] = Math.min(f[i + 1][j], f[i][j - 1]);
    }
}
return Math.max(f[0][arr.length - 1], s[0][arr.length - 1]);
```

# 机器人走路问题

给你标号为1、2、3、……、N的N个位置，机器人初始停在M位置上，走P步后停在K位置上的走法有多少种。注：机器人在1位置上时只能向右走，在N位置上时只能向左走，其它位置既可向右又可向左。

```
public static int process(int N, int M, int P, int K) {
    if (P == 0) {
        return M == K ? 1 : 0;
    }
    if (M == 1) {
        return process(N, M + 1, P - 1, K);
    } else if (M == N) {
        return process(N, M - 1, P - 1, K);
    }
    return process(N, M + 1, P - 1, K) + process(N, M - 1, P - 1, K);
}

public static void main(String[] args) {
    System.out.println(process(5, 2, 3, 3));
}
```

这里暴力递归参数列表的可变变量有`M`和`P`，根据`base case`和其它特殊情况画出二维表：




动态规划示例代码：

```
public static int robotWalkWaysDp(int N, int M, int P, int K) {
    int dp[][] = new int[N + 1][P + 1];
    dp[K][0] = 1;
    for (int j = 1; j <= P; j++) {
        for (int i = 1; i <= N; i++) {
            if (i - 1 < 1) {
                dp[i][j] = dp[i + 1][j - 1];
            } else if (i + 1 > N) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = dp[i + 1][j - 1] + dp[i - 1][j - 1];
            }
        }
    }
    return dp[M][P];
}

public static void main(String[] args) {
    System.out.println(robotWalkWaysDp(5, 2, 3, 3));
}
```

# 字符串正则匹配问题

给定字符串`str`，其中绝对不含有字符`'.'`和`'*'`。再给定字符串`exp`，其中可以含有`'.'`或`'*'`，`'*'`字符不能是`exp`的首字符，并且任意两个`'*'`字符不相邻。exp中的`'.'`代表任何一个字符，exp中的`'*'`表示`'*'`的前一个字符可以有0个或者多个。请写一个函数，判断`str`是否能被`exp`匹配。

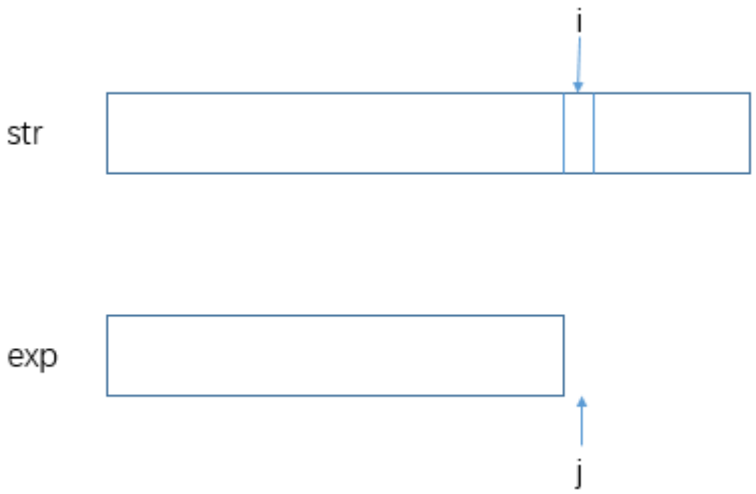
举例：

- `str="abc", exp="abc"`，返回`true`。`str="abc", exp="a.c"`，exp中单个`'.'`可以代表任意字符，所以返回`true`。
- `str="abcd", exp=".*"`。exp中`'*'`的前一个字符是`'.'`，所以可表示任意数量的`'.'`字符，当exp是`"...."`时与`"abcd"`匹配，返回`true`。
- `str="", exp="..*"`。exp中`'*'`的前一个字符是`'.'`，可表示任意数量的`'.'`字符，但是`".*"`之前还有一个`'.'`字符，该字符不受`'*'`的影响，所以`str`起码有一个字符才能被`exp`匹配。所以返回`false`。

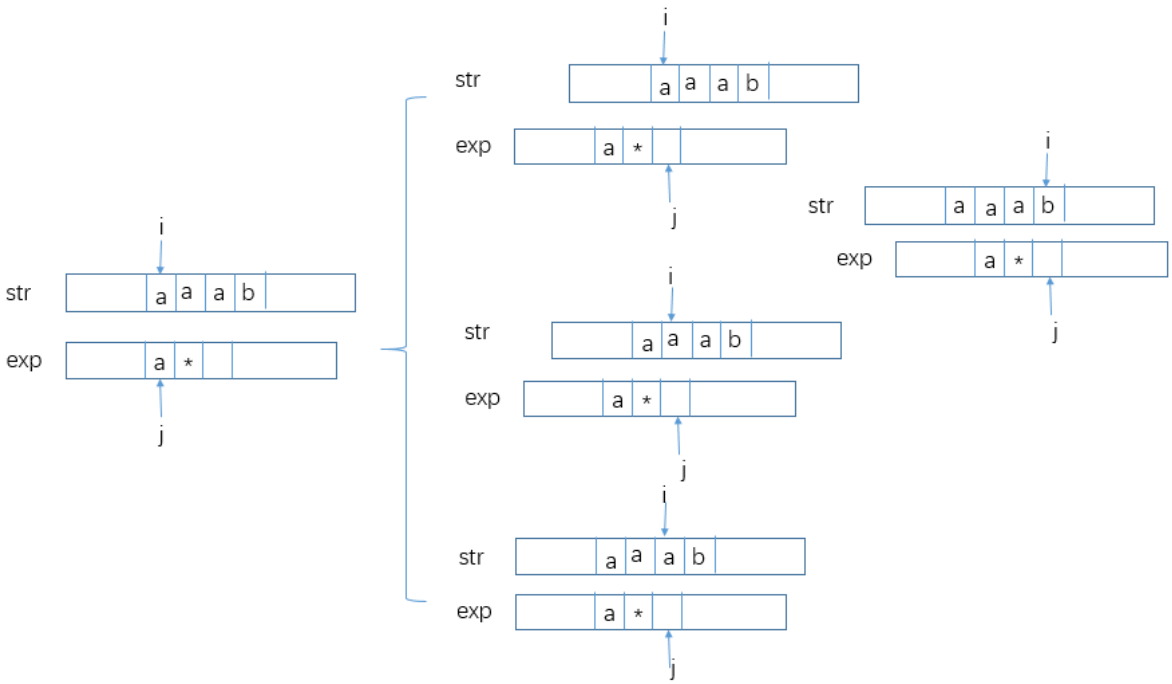
## 暴力尝试

定义一个方法`bool match(char[] str, int i, char[] exp, int j)`，表示`str`的下标`i ~ str.length`部分能否和`exp`的下标`j ~ exp.length`部分匹配，分情况讨论如下：

1. 如果`j`到了`exp.length`而`i`还没到`str.length`，返回`false`，否则返回`true`

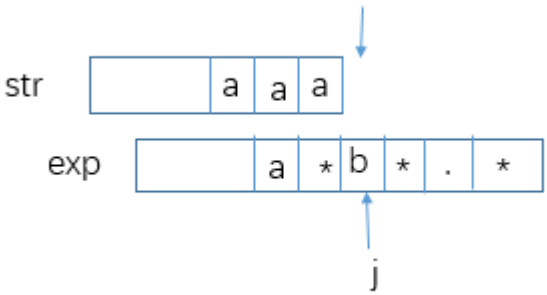



2. 如果`i`和`j`都没到右边界，并且`j`的后一个字符不是`*`或者越界，那么只有当`str[i]=exp[j]`或`exp[j]='.'`时，`i`和`j`才同时右移继续比较`match(str, i+1, exp, j+1)`，否则返回`false`
3. 如果`i`和`j`都没到右边界，并且`j`后一个字符是`*`，这时右有两种情况：
    1. `str[i] = exp[j]`或`exp[j]='.'`。比如`a*`可以匹配空串也可以匹配一个`a`，如果`str[i]`之后还有连续的相同字符，那么`a*`还可以匹配多个，不管是哪种情况，将匹配后右移的`i`和`j`交给子过程`match`




2. `str[i] != exp[j]` 且 `exp[j] != '.'` ，那么 `exp[j]*` 只能选择匹配空串。


4. 如果 `i` 到了 `str.length` 而 `j` 还没到 `exp.length` ，那么 `j` 之后的字符只能是 `a*b*c*.*` 的形式，也就是一个字符后必须跟一个 `*` 的形式，这个检验过程同样可以交给 `match` 来做




示例代码：

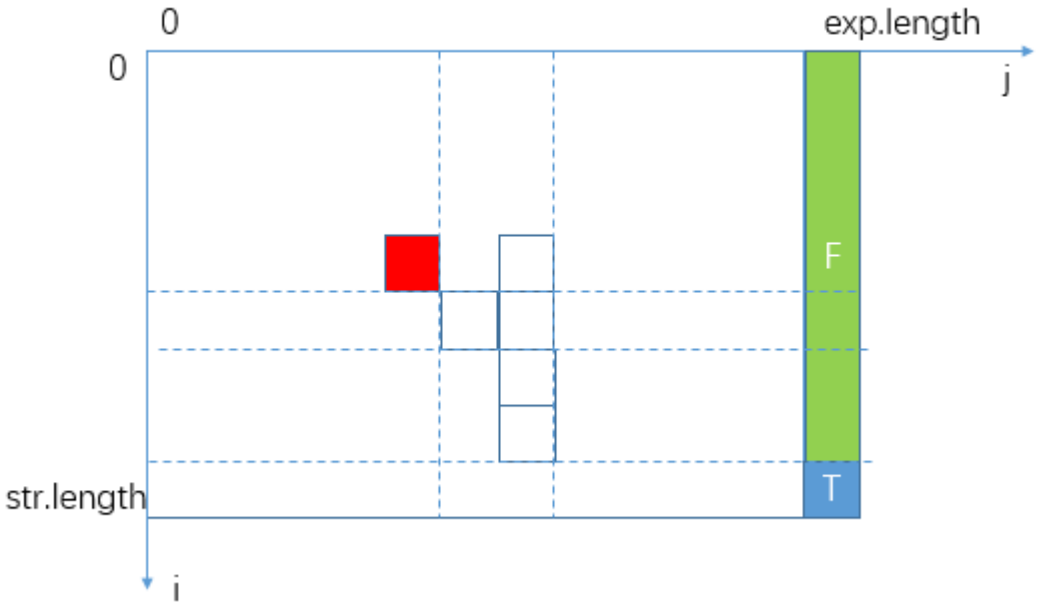
```java
public static boolean match(char[] s, int i, char[] e, int j) {
    if (j == e.length) {
        return i == s.length;
    }
    //j下一个越界或者j下一个不是*
    if (j + 1 == e.length || e[j + 1] != '*') {
        if (i != s.length && s[i] == e[j] || e[j] == '.') {
            return match(s, i + 1, e, j + 1);
```

```
        }
        return false;
    }
    //j下一个不越界并且j下一个是*
    while (i != s.length && s[i] == e[j] || e[j] == '.') {
        if (match(s, i, e, j + 2)) {
            return true;
        }
        i++;
    }
    //如果上面的while是因为 s[i]!=e[j] 而停止的
    return match(s, i, e, j + 2);
}

public static boolean isMatch(String str, String exp) {
    if (str == null || exp == null) {
        return false;
    }
    char[] s = str.toCharArray();
    char[] e = exp.toCharArray();
    return match(s, 0, e, 0);
}

public static void main(String[] args) {
    System.out.println(isMatch("abbbbc","a.*b*c"));//T
    System.out.println(isMatch("abbbbc","a.*bbc"));//T
    System.out.println(isMatch("abbbbc","a.bbc"));//F
    System.out.println(isMatch("abbbbc","a.bbbc"));//T
}
```

## 动态规划

`match` 的参数列表中只有 `i` 和 `j` 是变化的，也就是说只要确定了 `i` 和 `j` 就能对应确定一个 `match` 的状态，画出二维表并将 `base case` 对应位置状态值标注出来：

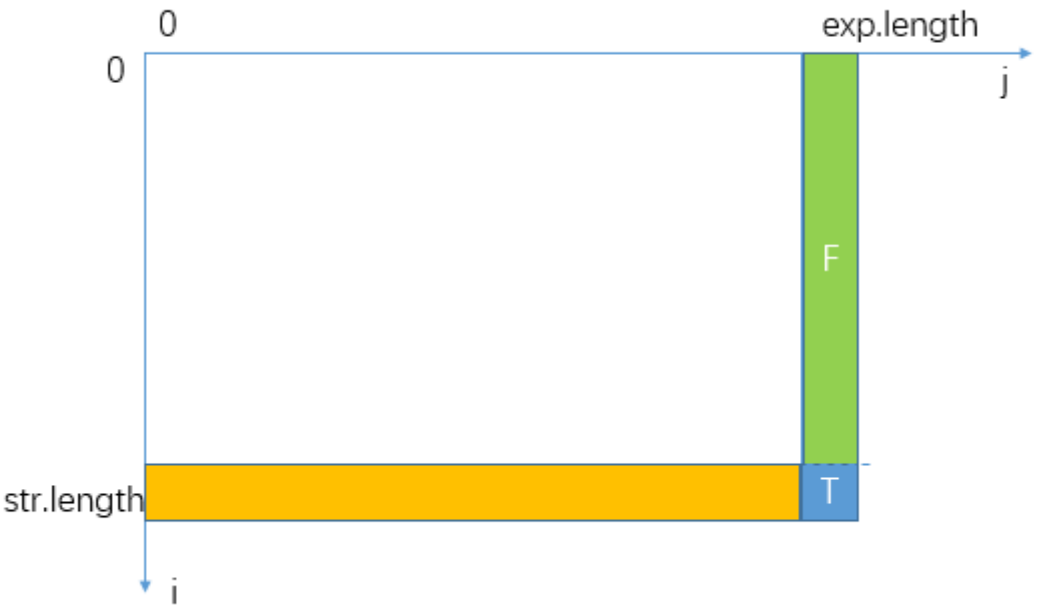
再看普遍位置 `(i,j)` 的依赖，第 `6` 行的 `if` 表明 `(i,j)` 可能依赖 `(i+1, j+1)`，第 `13` 行的 `while` 表明 `(i,j)` 可能依赖 `(i, j+2)`、`(i+1, j+2)`、`(i+2, j+2)`、……、`(s.length-1, j+2)`：



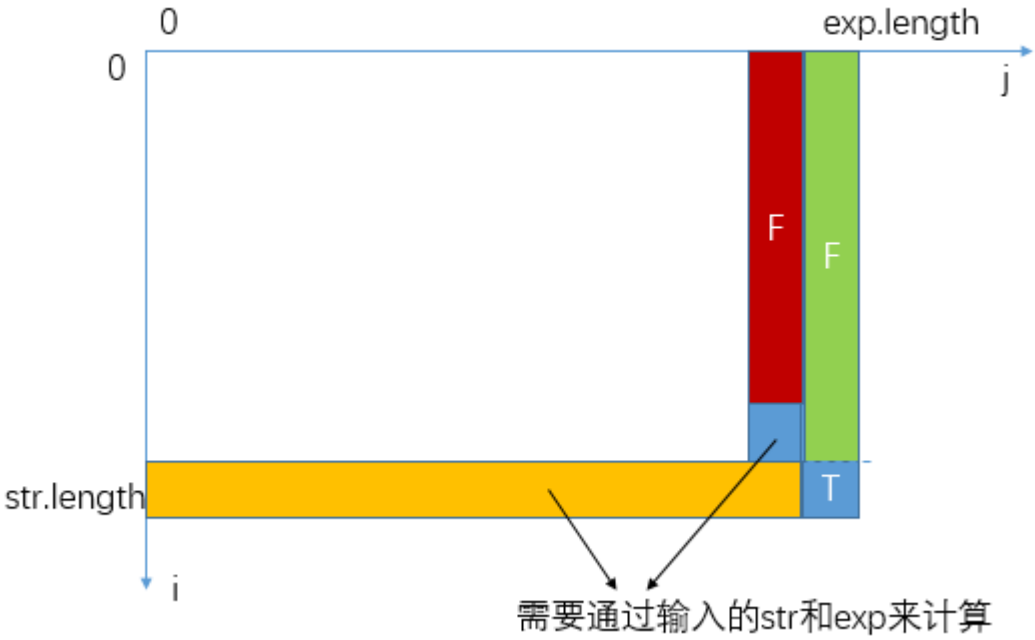

你会发现 `(i,j)` 依赖它下面一行和右边相邻两列的状态，也就是说要想推出普遍位置的状态值，起码需要最后一行、最后一列和倒数第二列上的状态值。而 `base case` 仅为我们提供了最后一列的状态值，主过程 `match(e, 0, s, 0)` 对应 `(0,0)` 位置的状态值，我们需要推出整张表所有位置的状态值才行。

这时就要回归题意了，看倒数第二列和最后一行上的状态有什么特殊含义。

首先最后一行表示 `i` 到了 `str.length`，此时如果 `j` 还没走完 `exp` 的话，从 `j` 开始到末尾的字符必须满足 `字符*字符*字符*` 的范式才返回 `true`。因此最后一行状态值易求：




而对于倒数第二列，表示 `j` 来到了 `exp` 的末尾字符，此时如果 `i` 如果在 `str` 末尾字符之前，那么也是直接返回 `false` 的：




那么接下来就只剩下 `(str.length-1, exp.length-1)` 这个位置的状态值了，该位置标明 `i` 来到了 `str` 的末尾字符，`j` 来到了 `exp` 的末尾字符，只有当这两个字符相等或 `exp` 的末尾字符为 `.` 才返回 `true` 否则 `false`，也就是说该状态可以直接通过输入参数 `str` 和 `exp` 计算，它不依赖其他状态。二维表的初始化至此全部完成。

示例代码：

```java
public static boolean isMatch(String str, String exp) {
    if (str == null || exp == null) {
        return false;
    }
    return matchDp(str, exp);
}

public static boolean matchDp(String str, String exp) {
    if (str == null || exp == null) {
        return false;
    }
    char s[] = str.toCharArray();
    char e[] = exp.toCharArray();
    boolean[][] dpMap = initDpMap(s, e);

    //从倒数第二行开始推，每一行从右向左推
    for (int i = s.length - 1; i > -1; i--) {
        for (int j = e.length - 2; j > -1; j--) {
            if (e[j + 1] != '*') {
                dpMap[i][j] = (s[i] == e[j] || e[j] == '.') && dpMap[i + 1][j + 1];
            } else {
                int tmp = i;
                while (tmp != s.length && (s[tmp] == e[j] || e[j] == '.')) {
                    if (dpMap[tmp][j + 2]) {
                        dpMap[i][j] = true;
                        break;
```

```
                    }
                    tmp++;
                }
                if (dpMap[i][j] != true) {
                    dpMap[i][j] = dpMap[i][j + 2];
                }
            }
        }
    }
    return dpMap[0][0];
}

public static boolean[][] initDpMap(char[] s, char[] e) {
    boolean[][] dpMap = new boolean[s.length + 1][e.length + 1];
    //last column
    dpMap[s.length][e.length] = true;
    //last row -> i=s.length-1
    for (int j = e.length - 2; j >= 0; j = j - 2) {
        if (e[j] != '*' && e[j + 1] == '*') {
            dpMap[s.length - 1][j] = true;
        } else {
            break;
        }
    }
    //(str.length-1, e.length-1)
    if (s[s.length - 1] == e[e.length - 1] || e[e.length - 1] == '.') {
        dpMap[s.length - 1][e.length - 1] = true;
    }
    return dpMap;
}
```

# 缓存结构的设计

## 设计可以变更的缓存结构（LRU）

设计一种缓存结构，该结构在构造时确定大小，假设大小为K，并有两个功能：`set(key,value)`：将记录`(key,value)`插入该结构。`get(key)`：返回key对应的value值。

【要求】

- set和get方法的时间复杂度为O(1)。
- 某个key的set或get操作一旦发生，认为这个key的记录成了最经常使用的。
- 当缓存的大小超过K时，移除最不经常使用的记录，即set或get最久远的。

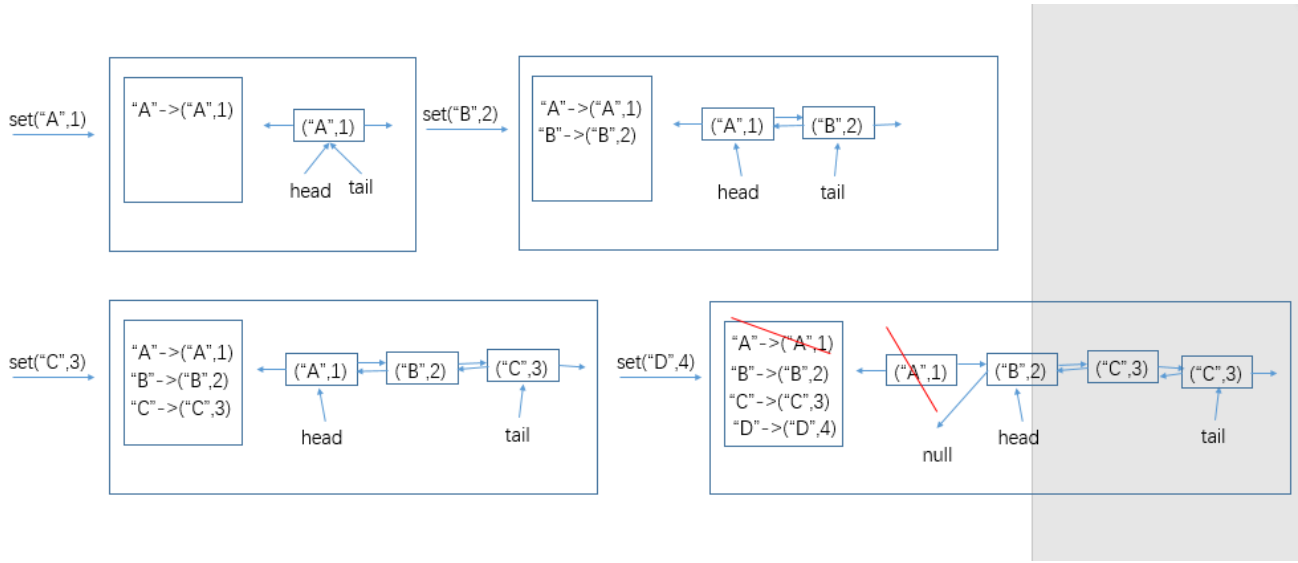
【举例】

假设缓存结构的实例是cache，大小为3，并依次发生如下行为：

1. cache.set("A",1)。最经常使用的记录为("A",1)。
2. cache.set("B",2)。最经常使用的记录为("B",2)，("A",1)变为最不经常的。
3. cache.set("C",3)。最经常使用的记录为("C",2)，("A",1)还是最不经常的。
4. cache.get("A")。最经常使用的记录为("A",1)，("B",2)变为最不经常的。

5. cache.set("D",4)。大小超过了3，所以移除此时最不经常使用的记录("B",2)，加入记录 ("D",4)，并且为最经常使用的记录，然后("C",2)变为最不经常使用的记录

> 设计思路：使用一个哈希表和双向链表




示例代码：

```java
package top.zhenganwen.structure;

import java.util.HashMap;

public class LRU {

    public static class Node<K,V>{
        K key;
        V value;
        Node<K,V> prev;
        Node<K, V> next;

        public Node(K key, V value) {
            this.key = key;
            this.value = value;
            this.prev = null;
            this.next = null;
        }
    }

    /**
     * the head is the oldest record and the tail is the newest record
     *
     * the add() will append the record to tail
     *
     * @param <K> key
     * @param <V> value
     */
    public static class DoubleLinkedList<K,V>{
        Node<K,V> head;
```

```
        Node<K, V> tail;
        public DoubleLinkedList() {
            this.head = null;
            this.tail = null;
        }

        public void add(Node<K,V> node){
            if (node == null) {
                return;
            }
            if (this.head == null) {
                this.head = node;
                this.tail = node;
            } else {
                this.tail.next = node;
                node.prev = this.tail;
                this.tail = node;
            }
        }

        public void moveToTail(Node<K,V> node){
            if (node == this.tail) {
                return;
            }
            if (node == this.head) {
                Node<K, V> newHead = node.next;
                newHead.prev = null;
                this.head = newHead;

                node.next = null;
                node.prev = this.tail;
                this.tail.next = node;
                this.tail = node;
            } else {
                node.prev.next = node.next;
                node.next.prev = node.prev;

                node.next=null;
                node.prev=this.tail;
                this.tail.next = node;
                this.tail = node;
            }
        }

        public K removeHead() {
            if (this.head != null) {
                K deletedK = this.head.key;
                if (this.head == this.tail) {
                    this.head = null;
                    this.tail = null;
                } else {
                    Node<K, V> newHead = this.head.next;
                    newHead.prev = null;
```

```
                    this.head = newHead;
                }
                return deletedK;
            }
            return null;
        }
    }

    public static class MyCache<K,V>{
        HashMap<K, Node<K, V>> map = new HashMap<>();
        DoubleLinkedList list = new DoubleLinkedList();
        int capacity;
        public MyCache(int capacity) {
            this.capacity = capacity;
        }

        public void set(K key, V value) {
            if (map.containsKey(key)) {
                //swap value

                //update map
                Node<K, V> node = map.get(key);
                node.value = value;
                map.put(key, node);
                //update list
                list.moveToTail(node);

            } else {
                //save record

                //if full,remove the oldest first and then save
                if (map.size() == this.capacity) {
                    K deletedK = (K) list.removeHead();
                    map.remove(deletedK);
                }
                Node<K, V> record = new Node<>(key, value);
                map.put(key, record);
                list.add(record);
            }
        }

        public V get(K key) {
            if (map.containsKey(key)) {
                Node<K, V> target = map.get(key);
                list.moveToTail(target);
                return target.value;
            } else {
                return null;
            }
        }
    }

    public static void main(String[] args) {
```

```
        MyCache<String, Integer> myCache = new MyCache<>(3);
        myCache.set("A", 1);
        myCache.set("B", 2);
        myCache.set("C", 3);
        System.out.println(myCache.get("A"));
        myCache.set("D", 4);
        System.out.println(myCache.get("B"));
    }
}
```

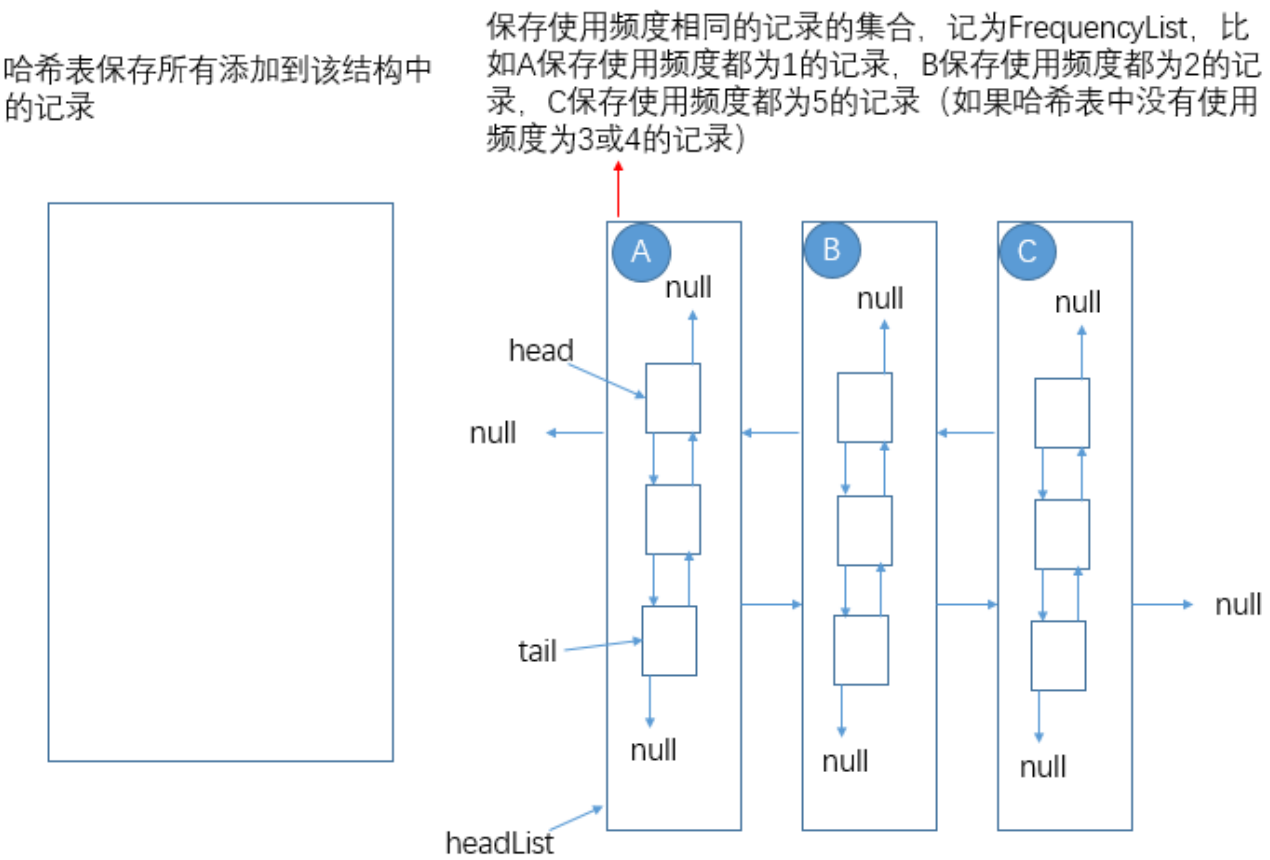
> 面试技巧：
>
> 在刷题时，如果感觉这个题明显在30分钟内解不出来就放弃，因为面试给出的题目一般会让你在30分内解出。
>
> 在面试时如果碰到自己遇到过的题也要装作没遇到过，假装一番苦思冥想、和面试官沟通细节，然后突然想通了的样子。

# LFU

LFU也是一种经典的缓存结构，只不过它是以`key`的访问频度作为缓存替换依据的。

举例：`set("A",Data)`将会在LFU结构中放入一条key为"A"的记录，并将该记录的使用频度置为1，后续的`set("A",newData)`或`get("A")`都会将该key对应的记录的使用频度加1；当该结构容量已满还尝试往里添加记录时，会先将结构中使用频度最少的记录删除，再将新的记录添加进去。

> 设计思路：使用一个哈希表和一个二维双向链表（链表中包含链表）




示例代码：

```java
import java.util.HashMap;

public class LFUCache<K,V>{

    /**
     * Save all record
     */
    private HashMap<K, Record<K,V>> recordMap;
    /**
     * The reference of the FrequencyList whose frequency is the lowest
     */
    private FrequencyList headList;
    /**
     * Save what FrequencyList a record belongs to
     */
    private HashMap<Record,FrequencyList> listOfRecord;
    /**
     * How many recordMap the LFUCache can contain
     */
    private int capacity;
    /**
     * how many recordMap has been saved
     */
    private int size;

    public LFUCache(int capacity) {
        this.recordMap = new HashMap();
        this.listOfRecord = new HashMap<>();
        this.headList = null;
        this.capacity = capacity;
        this.size = 0;
    }

    /**
     * add or update a record
     * @param key
     * @param value
     */
    public void set(K key, V value) {
        //update
        if (this.recordMap.containsKey(key)) {
            //update value and frequency
            Record<K, V> record = recordMap.get(key);
            record.value = value;
            record.frequency++;
            //adjust the record's position in FrequencyList
            adjust(record, listOfRecord.get(record));
        } else {
            //add
            if (size == capacity) {
                //delete
                recordMap.remove(headList.tail.key);
                headList.deleteRecord(headList.tail);
```

```
                size--;
                modifyFrequencyList(headList);
            }
            Record<K, V> newRecord = new Record<>(key, value);
            recordMap.put(key, newRecord);
            size++;
            if (headList == null) {
                headList = new FrequencyList(newRecord);
            } else if (headList.head.frequency != 1) {
                FrequencyList frequencyList = new FrequencyList(newRecord);
                headList.prev = frequencyList;
                frequencyList.next = headList;
                frequencyList.prev = null;
                headList = frequencyList;
            } else {
                headList.addRecordToHead(newRecord);
            }
            listOfRecord.put(newRecord, headList);
        }
    }

    /**
     * get a record by a key,return null if not exists
     * @param key
     * @return
     */
    public V get(K key) {
        if (!recordMap.containsKey(key)) {
            return null;
        }
        Record<K, V> record = recordMap.get(key);
        record.frequency++;
        adjust(record, listOfRecord.get(record));
        return record.value;
    }

    /**
     * When the record's frequency changed,split it from its current
     * FrequencyList and insert to another one
     *
     * @param record
     * @param frequencyList
     */
    private void adjust(Record<K, V> record, FrequencyList frequencyList) {
        //split
        frequencyList.deleteRecord(record);
        boolean deleted = modifyFrequencyList(frequencyList);
        //insert to anther one
        FrequencyList prevList = frequencyList.prev;
        FrequencyList nextList = frequencyList.next;
        if (nextList != null && record.frequency == nextList.head.frequency) {
            nextList.addRecordToHead(record);
            listOfRecord.put(record, nextList);
```

```
        } else {
            FrequencyList newList = new FrequencyList(record);
            if (prevList == null) {
                if (nextList != null) {
                    nextList.prev = newList;
                }
                newList.next = nextList;
                newList.prev = null;
                headList = newList;
            } else if (nextList == null) {
                prevList.next = newList;
                newList.prev = prevList;
                newList.next = null;
            } else {
                prevList.next = newList;
                newList.prev = prevList;
                newList.next = nextList;
                nextList.prev = newList;
            }
            listOfRecord.put(record, newList);
        }
    }

    /**
     * return whether the frequencyList is deleted
     * @param frequencyList
     * @return
     */
    private boolean modifyFrequencyList(FrequencyList frequencyList) {
        if (!frequencyList.isEmpty()) {
            return false;
        }
        if (frequencyList.prev == null) {
            headList = frequencyList.next;
            if (headList != null) {
                headList.prev = null;
            }
        } else if (frequencyList.next == null) {
            frequencyList.prev.next = null;
        } else {
            frequencyList.prev.next = frequencyList.next;
            frequencyList.next.prev = frequencyList.prev;
        }
        return true;
    }

    /**
     * The Record can be design to Record<K,V> or Record<V> used
     * to encapsulate data
     * @param <K> key
     * @param <V> value
     */
    private class Record<K,V> {
```

```
        K key;
        V value;
        /**
         * up->the predecessor pointer
         * down->the successor pointer
         */
        Record<K, V> up;
        Record<K, V> down;
        /**
         * the frequency of use
         */
        int frequency;

        /**
         * when the record was created , set the frequency to 1
         *
         * @param key
         * @param value
         */
        public Record(K key, V value) {
            this.key = key;
            this.value = value;
            this.frequency = 1;
        }
    }

    /**
     * The FrequencyList save a series of Records that
     * has the same frequency
     */
    private class FrequencyList {

        /**
         * prev->the predecessor pointer
         * next->the successor pointer
         */
        FrequencyList prev;
        FrequencyList next;
        /**
         * The reference of the internal RecordList's head and tail
         */
        Record<K,V> head;
        Record<K,V> tail;

        public FrequencyList(Record<K, V> record) {
            this.head = record;
            this.tail = record;
        }

        public void addRecordToHead(Record<K, V> record) {
            head.up = record;
            record.down = head;
            head = record;
```

```
        }

        public boolean isEmpty() {
            return head == null;
        }

        public void deleteRecord(Record<K,V> record) {
            if (head == tail) {
                head = null;
                tail = null;
            } else if (record == head) {
                head=head.down;
                head.up = null;
            } else if (record == tail) {
                tail = tail.up;
                tail.down = null;
            } else {
                record.up.down = record.down;
                record.down.up = record.up;
            }
        }
    }

    public static void main(String[] args) {
        LFUCache<String, Integer> cache = new LFUCache<>(3);
        cache.set("A", 1);
        cache.set("A", 1);
        cache.set("A", 1);
        cache.set("B", 2);
        cache.set("B", 2);
        cache.set("C", 3);
        cache.set("D", 4);
        System.out.println("break point");
    }
}
```

# 附录

## 手写二叉搜索树

> 下列代码来源于`github`

### AbstractBinarySearchTree

```
/**
 * Not implemented by zhenganwen
 *
```

```java
 * Abstract binary search tree implementation. Its basically fully implemented
 * binary search tree, just template method is provided for creating Node (other
 * trees can have slightly different nodes with more info). This way some code
 * from standart binary search tree can be reused for other kinds of binary
 * trees.
 *
 * @author Ignas Lelys
 * @created Jun 29, 2011
 *
 */
public class AbstractBinarySearchTree {

    /** Root node where whole tree starts. */
    public Node root;

    /** Tree size. */
    protected int size;

    /**
     * Because this is abstract class and various trees have different
     * additional information on different nodes subclasses uses this abstract
     * method to create nodes (maybe of class {@link Node} or maybe some
     * different node sub class).
     *
     * @param value
     *            Value that node will have.
     * @param parent
     *            Node's parent.
     * @param left
     *            Node's left child.
     * @param right
     *            Node's right child.
     * @return Created node instance.
     */
    protected Node createNode(int value, Node parent, Node left, Node right) {
        return new Node(value, parent, left, right);
    }

    /**
     * Finds a node with concrete value. If it is not found then null is
     * returned.
     *
     * @param element
     *            Element value.
     * @return Node with value provided, or null if not found.
     */
    public Node search(int element) {
        Node node = root;
        while (node != null && node.value != null && node.value != element) {
            if (element < node.value) {
                node = node.left;
            } else {
                node = node.right;
```

```
            }
        }
        return node;
    }

    /**
     * Insert new element to tree.
     *
     * @param element
     *            Element to insert.
     */
    public Node insert(int element) {
        if (root == null) {
            root = createNode(element, null, null, null);
            size++;
            return root;
        }

        Node insertParentNode = null;
        Node searchTempNode = root;
        while (searchTempNode != null && searchTempNode.value != null) {
            insertParentNode = searchTempNode;
            if (element < searchTempNode.value) {
                searchTempNode = searchTempNode.left;
            } else {
                searchTempNode = searchTempNode.right;
            }
        }

        Node newNode = createNode(element, insertParentNode, null, null);
        if (insertParentNode.value > newNode.value) {
            insertParentNode.left = newNode;
        } else {
            insertParentNode.right = newNode;
        }

        size++;
        return newNode;
    }

    /**
     * Removes element if node with such value exists.
     *
     * @param element
     *            Element value to remove.
     *
     * @return New node that is in place of deleted node. Or null if element for
     *         delete was not found.
     */
    public Node delete(int element) {
        Node deleteNode = search(element);
        if (deleteNode != null) {
            return delete(deleteNode);
```

```
110         } else {
111             return null;
112         }
113     }
114 
115     /**
116      * Delete logic when node is already found.
117      *
118      * @param deleteNode
119      *            Node that needs to be deleted.
120      *
121      * @return New node that is in place of deleted node. Or null if element for
122      *         delete was not found.
123      */
124     protected Node delete(Node deleteNode) {
125         if (deleteNode != null) {
126             Node nodeToReturn = null;
127             if (deleteNode != null) {
128                 if (deleteNode.left == null) {
129                     nodeToReturn = transplant(deleteNode, deleteNode.right);
130                 } else if (deleteNode.right == null) {
131                     nodeToReturn = transplant(deleteNode, deleteNode.left);
132                 } else {
133                     Node successorNode = getMinimum(deleteNode.right);
134                     if (successorNode.parent != deleteNode) {
135                         transplant(successorNode, successorNode.right);
136                         successorNode.right = deleteNode.right;
137                         successorNode.right.parent = successorNode;
138                     }
139                     transplant(deleteNode, successorNode);
140                     successorNode.left = deleteNode.left;
141                     successorNode.left.parent = successorNode;
142                     nodeToReturn = successorNode;
143                 }
144                 size--;
145             }
146             return nodeToReturn;
147         }
148         return null;
149     }
150 
151     /**
152      * Put one node from tree (newNode) to the place of another (nodeToReplace).
153      *
154      * @param nodeToReplace
155      *            Node which is replaced by newNode and removed from tree.
156      * @param newNode
157      *            New node.
158      *
159      * @return New replaced node.
160      */
161     private Node transplant(Node nodeToReplace, Node newNode) {
162         if (nodeToReplace.parent == null) {
```

```
163             this.root = newNode;
164         } else if (nodeToReplace == nodeToReplace.parent.left) {
165             nodeToReplace.parent.left = newNode;
166         } else {
167             nodeToReplace.parent.right = newNode;
168         }
169         if (newNode != null) {
170             newNode.parent = nodeToReplace.parent;
171         }
172         return newNode;
173     }
174 
175     /**
176      * @param element
177      * @return true if tree contains element.
178      */
179     public boolean contains(int element) {
180         return search(element) != null;
181     }
182 
183     /**
184      * @return Minimum element in tree.
185      */
186     public int getMinimum() {
187         return getMinimum(root).value;
188     }
189 
190     /**
191      * @return Maximum element in tree.
192      */
193     public int getMaximum() {
194         return getMaximum(root).value;
195     }
196 
197     /**
198      * Get next element element who is bigger than provided element.
199      *
200      * @param element
201      *            Element for whom descendand element is searched
202      * @return Successor value.
203      */
204     // TODO Predecessor
205     public int getSuccessor(int element) {
206         return getSuccessor(search(element)).value;
207     }
208 
209     /**
210      * @return Number of elements in the tree.
211      */
212     public int getSize() {
213         return size;
214     }
215 
```

```
216     /**
217      * Tree traversal with printing element values. In order method.
218      */
219     public void printTreeInOrder() {
220         printTreeInOrder(root);
221     }
222 
223     /**
224      * Tree traversal with printing element values. Pre order method.
225      */
226     public void printTreePreOrder() {
227         printTreePreOrder(root);
228     }
229 
230     /**
231      * Tree traversal with printing element values. Post order method.
232      */
233     public void printTreePostOrder() {
234         printTreePostOrder(root);
235     }
236 
237     /*-------------------PRIVATE HELPER METHODS-------------------*/
238 
239     private void printTreeInOrder(Node entry) {
240         if (entry != null) {
241             printTreeInOrder(entry.left);
242             if (entry.value != null) {
243                 System.out.println(entry.value);
244             }
245             printTreeInOrder(entry.right);
246         }
247     }
248 
249     private void printTreePreOrder(Node entry) {
250         if (entry != null) {
251             if (entry.value != null) {
252                 System.out.println(entry.value);
253             }
254             printTreeInOrder(entry.left);
255             printTreeInOrder(entry.right);
256         }
257     }
258 
259     private void printTreePostOrder(Node entry) {
260         if (entry != null) {
261             printTreeInOrder(entry.left);
262             printTreeInOrder(entry.right);
263             if (entry.value != null) {
264                 System.out.println(entry.value);
265             }
266         }
267     }
268 
```

```
269     protected Node getMinimum(Node node) {
270         while (node.left != null) {
271             node = node.left;
272         }
273         return node;
274     }
275 
276     protected Node getMaximum(Node node) {
277         while (node.right != null) {
278             node = node.right;
279         }
280         return node;
281     }
282 
283     protected Node getSuccessor(Node node) {
284         // if there is right branch, then successor is leftmost node of that
285         // subtree
286         if (node.right != null) {
287             return getMinimum(node.right);
288         } else { // otherwise it is a lowest ancestor whose left child is also
289             // ancestor of node
290             Node currentNode = node;
291             Node parentNode = node.parent;
292             while (parentNode != null && currentNode == parentNode.right) {
293                 // go up until we find parent that currentNode is not in right
294                 // subtree.
295                 currentNode = parentNode;
296                 parentNode = parentNode.parent;
297             }
298             return parentNode;
299         }
300     }
301 
302     // ------------------------------- TREE PRINTING
303     // -----------------------------------
304 
305     public void printTree() {
306         printSubtree(root);
307     }
308 
309     public void printSubtree(Node node) {
310         if (node.right != null) {
311             printTree(node.right, true, "");
312         }
313         printNodeValue(node);
314         if (node.left != null) {
315             printTree(node.left, false, "");
316         }
317     }
318 
319     private void printNodeValue(Node node) {
320         if (node.value == null) {
321             System.out.print("<null>");
```

```
322         } else {
323             System.out.print(node.value.toString());
324         }
325         System.out.println();
326     }
327 
328     private void printTree(Node node, boolean isRight, String indent) {
329         if (node.right != null) {
330             printTree(node.right, true, indent + (isRight ? "        " : " |
"));
331         }
332         System.out.print(indent);
333         if (isRight) {
334             System.out.print(" /");
335         } else {
336             System.out.print(" \\");
337         }
338         System.out.print("----- ");
339         printNodeValue(node);
340         if (node.left != null) {
341             printTree(node.left, false, indent + (isRight ? " |      " : "
"));
342         }
343     }
344 
345     public static class Node {
346         public Node(Integer value, Node parent, Node left, Node right) {
347             super();
348             this.value = value;
349             this.parent = parent;
350             this.left = left;
351             this.right = right;
352         }
353 
354         public Integer value;
355         public Node parent;
356         public Node left;
357         public Node right;
358 
359         public boolean isLeaf() {
360             return left == null && right == null;
361         }
362 
363         @Override
364         public int hashCode() {
365             final int prime = 31;
366             int result = 1;
367             result = prime * result + ((value == null) ? 0 : value.hashCode());
368             return result;
369         }
370 
371         @Override
372         public boolean equals(Object obj) {
```

```
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Node other = (Node) obj;
            if (value == null) {
                if (other.value != null)
                    return false;
            } else if (!value.equals(other.value))
                return false;
            return true;
        }

    }
}
```

## AbstractSelfBalancingBinarySearchTree

```
package advanced_class_03;

/**
 * Not implemented by zhenganwen
 *
 * Abstract class for self balancing binary search trees. Contains some methods
 * that is used for self balancing trees.
 *
 * @author Ignas Lelys
 * @created Jul 24, 2011
 *
 */
public abstract class AbstractSelfBalancingBinarySearchTree extends
AbstractBinarySearchTree {

    /**
     * Rotate to the left.
     *
     * @param node Node on which to rotate.
     * @return Node that is in place of provided node after rotation.
     */
    protected Node rotateLeft(Node node) {
        Node temp = node.right;
        temp.parent = node.parent;

        node.right = temp.left;
        if (node.right != null) {
            node.right.parent = node;
        }

        temp.left = node;
        node.parent = temp;

```

```
        // temp took over node's place so now its parent should point to temp
        if (temp.parent != null) {
            if (node == temp.parent.left) {
                temp.parent.left = temp;
            } else {
                temp.parent.right = temp;
            }
        } else {
            root = temp;
        }

        return temp;
    }

    /**
     * Rotate to the right.
     *
     * @param node Node on which to rotate.
     * @return Node that is in place of provided node after rotation.
     */
    protected Node rotateRight(Node node) {
        Node temp = node.left;
        temp.parent = node.parent;

        node.left = temp.right;
        if (node.left != null) {
            node.left.parent = node;
        }

        temp.right = node;
        node.parent = temp;

        // temp took over node's place so now its parent should point to temp
        if (temp.parent != null) {
            if (node == temp.parent.left) {
                temp.parent.left = temp;
            } else {
                temp.parent.right = temp;
            }
        } else {
            root = temp;
        }

        return temp;
    }

}
```

## AVLTree

```
/**
 * Not implemented by zhenganwen
 *
```

```java
 * AVL tree implementation.
 *
 * In computer science, an AVL tree is a self-balancing binary search tree, and
 * it was the first such data structure to be invented.[1] In an AVL tree, the
 * heights of the two child subtrees of any node differ by at most one. Lookup,
 * insertion, and deletion all take O(log n) time in both the average and worst
 * cases, where n is the number of nodes in the tree prior to the operation.
 * Insertions and deletions may require the tree to be rebalanced by one or more
 * tree rotations.
 *
 * @author Ignas Lelys
 * @created Jun 28, 2011
 *
 */
public class AVLTree extends AbstractSelfBalancingBinarySearchTree {

    /**
     * @see trees.AbstractBinarySearchTree#insert(int)
     *
     *      AVL tree insert method also balances tree if needed. Additional
     *      height parameter on node is used to track if one subtree is higher
     *      than other by more than one, if so AVL tree rotations is performed
     *      to regain balance of the tree.
     */
    @Override
    public Node insert(int element) {
        Node newNode = super.insert(element);
        rebalance((AVLNode)newNode);
        return newNode;
    }

    /**
     * @see trees.AbstractBinarySearchTree#delete(int)
     */
    @Override
    public Node delete(int element) {
        Node deleteNode = super.search(element);
        if (deleteNode != null) {
            Node successorNode = super.delete(deleteNode);
            if (successorNode != null) {
                // if replaced from getMinimum(deleteNode.right) then come back
there and update heights
                AVLNode minimum = successorNode.right != null ?
(AVLNode)getMinimum(successorNode.right) : (AVLNode)successorNode;
                recomputeHeight(minimum);
                rebalance((AVLNode)minimum);
            } else {
                recomputeHeight((AVLNode)deleteNode.parent);
                rebalance((AVLNode)deleteNode.parent);
            }
            return successorNode;
        }
        return null;
```

```
55      }
56
57      /**
58       * @see trees.AbstractBinarySearchTree#createNode(int,
        trees.AbstractBinarySearchTree.Node, trees.AbstractBinarySearchTree.Node,
        trees.AbstractBinarySearchTree.Node)
59       */
60      @Override
61      protected Node createNode(int value, Node parent, Node left, Node right) {
62          return new AVLNode(value, parent, left, right);
63      }
64
65      /**
66       * Go up from inserted node, and update height and balance informations if
        needed.
67       * If some node balance reaches 2 or -2 that means that subtree must be
        rebalanced.
68       *
69       * @param node Inserted Node.
70       */
71      private void rebalance(AVLNode node) {
72          while (node != null) {
73
74              Node parent = node.parent;
75
76              int leftHeight = (node.left == null) ? -1 : ((AVLNode)
                node.left).height;
77              int rightHeight = (node.right == null) ? -1 : ((AVLNode)
                node.right).height;
78              int nodeBalance = rightHeight - leftHeight;
79              // rebalance (-2 means left subtree outgrow, 2 means right subtree)
80              if (nodeBalance == 2) {
81                  if (node.right.right != null) {
82                      node = (AVLNode)avlRotateLeft(node);
83                      break;
84                  } else {
85                      node = (AVLNode)doubleRotateRightLeft(node);
86                      break;
87                  }
88              } else if (nodeBalance == -2) {
89                  if (node.left.left != null) {
90                      node = (AVLNode)avlRotateRight(node);
91                      break;
92                  } else {
93                      node = (AVLNode)doubleRotateLeftRight(node);
94                      break;
95                  }
96              } else {
97                  updateHeight(node);
98              }
99
100             node = (AVLNode)parent;
101         }
```

```
102     }
103
104     /**
105      * Rotates to left side.
106      */
107     private Node avlRotateLeft(Node node) {
108         Node temp = super.rotateLeft(node);
109
110         updateHeight((AVLNode)temp.left);
111         updateHeight((AVLNode)temp);
112         return temp;
113     }
114
115     /**
116      * Rotates to right side.
117      */
118     private Node avlRotateRight(Node node) {
119         Node temp = super.rotateRight(node);
120
121         updateHeight((AVLNode)temp.right);
122         updateHeight((AVLNode)temp);
123         return temp;
124     }
125
126     /**
127      * Take right child and rotate it to the right side first and then rotate
128      * node to the left side.
129      */
130     protected Node doubleRotateRightLeft(Node node) {
131         node.right = avlRotateRight(node.right);
132         return avlRotateLeft(node);
133     }
134
135     /**
136      * Take right child and rotate it to the right side first and then rotate
137      * node to the left side.
138      */
139     protected Node doubleRotateLeftRight(Node node) {
140         node.left = avlRotateLeft(node.left);
141         return avlRotateRight(node);
142     }
143
144     /**
145      * Recomputes height information from the node and up for all of parents. It
    needs to be done after delete.
146      */
147     private void recomputeHeight(AVLNode node) {
148         while (node != null) {
149             node.height = maxHeight((AVLNode)node.left, (AVLNode)node.right) + 1;
150             node = (AVLNode)node.parent;
151         }
152     }
153
```

```
    /**
     * Returns higher height of 2 nodes.
     */
    private int maxHeight(AVLNode node1, AVLNode node2) {
        if (node1 != null && node2 != null) {
            return node1.height > node2.height ? node1.height : node2.height;
        } else if (node1 == null) {
            return node2 != null ? node2.height : -1;
        } else if (node2 == null) {
            return node1 != null ? node1.height : -1;
        }
        return -1;
    }

    /**
     * Updates height and balance of the node.
     *
     * @param node Node for which height and balance must be updated.
     */
    private static final void updateHeight(AVLNode node) {
        int leftHeight = (node.left == null) ? -1 : ((AVLNode) node.left).height;
        int rightHeight = (node.right == null) ? -1 : ((AVLNode)
node.right).height;
        node.height = 1 + Math.max(leftHeight, rightHeight);
    }

    /**
     * Node of AVL tree has height and balance additional properties. If balance
     * equals 2 (or -2) that node needs to be re balanced. (Height is height of
     * the subtree starting with this node, and balance is difference between
     * left and right nodes heights).
     *
     * @author Ignas Lelys
     * @created Jun 30, 2011
     *
     */
    protected static class AVLNode extends Node {
        public int height;

        public AVLNode(int value, Node parent, Node left, Node right) {
            super(value, parent, left, right);
        }
    }

}
```

## RedBlackTree

```
/**
 * Not implemented by zhenganwen
 *
 * Red-Black tree implementation. From Introduction to Algorithms 3rd edition.
 *
```

```
 * @author Ignas Lelys
 * @created May 6, 2011
 *
 */
public class RedBlackTree extends AbstractSelfBalancingBinarySearchTree {

    protected enum ColorEnum {
        RED,
        BLACK
    };

    protected static final RedBlackNode nilNode = new RedBlackNode(null, null, null, null, ColorEnum.BLACK);

    /**
     * @see trees.AbstractBinarySearchTree#insert(int)
     */
    @Override
    public Node insert(int element) {
        Node newNode = super.insert(element);
        newNode.left = nilNode;
        newNode.right = nilNode;
        root.parent = nilNode;
        insertRBFixup((RedBlackNode) newNode);
        return newNode;
    }

    /**
     * Slightly modified delete routine for red-black tree.
     *
     * {@inheritDoc}
     */
    @Override
    protected Node delete(Node deleteNode) {
        Node replaceNode = null; // track node that replaces removedOrMovedNode
        if (deleteNode != null && deleteNode != nilNode) {
            Node removedOrMovedNode = deleteNode; // same as deleteNode if it has only one child, and otherwise it replaces deleteNode
            ColorEnum removedOrMovedNodeColor = ((RedBlackNode)removedOrMovedNode).color;

            if (deleteNode.left == nilNode) {
                replaceNode = deleteNode.right;
                rbTreeTransplant(deleteNode, deleteNode.right);
            } else if (deleteNode.right == nilNode) {
                replaceNode = deleteNode.left;
                rbTreeTransplant(deleteNode, deleteNode.left);
            } else {
                removedOrMovedNode = getMinimum(deleteNode.right);
                removedOrMovedNodeColor = ((RedBlackNode)removedOrMovedNode).color;
                replaceNode = removedOrMovedNode.right;
                if (removedOrMovedNode.parent == deleteNode) {
```

```
                    replaceNode.parent = removedOrMovedNode;
                } else {
                    rbTreeTransplant(removedOrMovedNode,
removedOrMovedNode.right);
                    removedOrMovedNode.right = deleteNode.right;
                    removedOrMovedNode.right.parent = removedOrMovedNode;
                }
                rbTreeTransplant(deleteNode, removedOrMovedNode);
                removedOrMovedNode.left = deleteNode.left;
                removedOrMovedNode.left.parent = removedOrMovedNode;
                ((RedBlackNode)removedOrMovedNode).color =
((RedBlackNode)deleteNode).color;
            }

            size--;
            if (removedOrMovedNodeColor == ColorEnum.BLACK) {
                deleteRBFixup((RedBlackNode)replaceNode);
            }
        }

        return replaceNode;
    }

    /**
     * @see trees.AbstractBinarySearchTree#createNode(int,
trees.AbstractBinarySearchTree.Node, trees.AbstractBinarySearchTree.Node,
trees.AbstractBinarySearchTree.Node)
     */
    @Override
    protected Node createNode(int value, Node parent, Node left, Node right) {
        return new RedBlackNode(value, parent, left, right, ColorEnum.RED);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected Node getMinimum(Node node) {
        while (node.left != nilNode) {
            node = node.left;
        }
        return node;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected Node getMaximum(Node node) {
        while (node.right != nilNode) {
            node = node.right;
        }
        return node;
```

```
104     }
105
106     /**
107      * {@inheritDoc}
108      */
109     @Override
110     protected Node rotateLeft(Node node) {
111         Node temp = node.right;
112         temp.parent = node.parent;
113
114         node.right = temp.left;
115         if (node.right != nilNode) {
116             node.right.parent = node;
117         }
118
119         temp.left = node;
120         node.parent = temp;
121
122         // temp took over node's place so now its parent should point to temp
123         if (temp.parent != nilNode) {
124             if (node == temp.parent.left) {
125                 temp.parent.left = temp;
126             } else {
127                 temp.parent.right = temp;
128             }
129         } else {
130             root = temp;
131         }
132
133         return temp;
134     }
135
136     /**
137      * {@inheritDoc}
138      */
139     @Override
140     protected Node rotateRight(Node node) {
141         Node temp = node.left;
142         temp.parent = node.parent;
143
144         node.left = temp.right;
145         if (node.left != nilNode) {
146             node.left.parent = node;
147         }
148
149         temp.right = node;
150         node.parent = temp;
151
152         // temp took over node's place so now its parent should point to temp
153         if (temp.parent != nilNode) {
154             if (node == temp.parent.left) {
155                 temp.parent.left = temp;
156             } else {
```

```
                temp.parent.right = temp;
            }
        } else {
            root = temp;
        }

        return temp;
    }


    /**
     * Similar to original transplant() method in BST but uses nilNode instead of null.
     */
    private Node rbTreeTransplant(Node nodeToReplace, Node newNode) {
        if (nodeToReplace.parent == nilNode) {
            this.root = newNode;
        } else if (nodeToReplace == nodeToReplace.parent.left) {
            nodeToReplace.parent.left = newNode;
        } else {
            nodeToReplace.parent.right = newNode;
        }
        newNode.parent = nodeToReplace.parent;
        return newNode;
    }

    /**
     * Restores Red-Black tree properties after delete if needed.
     */
    private void deleteRBFixup(RedBlackNode x) {
        while (x != root && isBlack(x)) {

            if (x == x.parent.left) {
                RedBlackNode w = (RedBlackNode)x.parent.right;
                if (isRed(w)) { // case 1 - sibling is red
                    w.color = ColorEnum.BLACK;
                    ((RedBlackNode)x.parent).color = ColorEnum.RED;
                    rotateLeft(x.parent);
                    w = (RedBlackNode)x.parent.right; // converted to case 2, 3 or 4
                }
                // case 2 sibling is black and both of its children are black
                if (isBlack(w.left) && isBlack(w.right)) {
                    w.color = ColorEnum.RED;
                    x = (RedBlackNode)x.parent;
                } else if (w != nilNode) {
                    if (isBlack(w.right)) { // case 3 sibling is black and its left child is red and right child is black
                        ((RedBlackNode)w.left).color = ColorEnum.BLACK;
                        w.color = ColorEnum.RED;
                        rotateRight(w);
                        w = (RedBlackNode)x.parent.right;
                    }
```

```
                    w.color = ((RedBlackNode)x.parent).color; // case 4 sibling is black and right child is red
                    ((RedBlackNode)x.parent).color = ColorEnum.BLACK;
                    ((RedBlackNode)w.right).color = ColorEnum.BLACK;
                    rotateLeft(x.parent);
                    x = (RedBlackNode)root;
                } else {
                    x.color = ColorEnum.BLACK;
                    x = (RedBlackNode)x.parent;
                }
            } else {
                RedBlackNode w = (RedBlackNode)x.parent.left;
                if (isRed(w)) { // case 1 - sibling is red
                    w.color = ColorEnum.BLACK;
                    ((RedBlackNode)x.parent).color = ColorEnum.RED;
                    rotateRight(x.parent);
                    w = (RedBlackNode)x.parent.left; // converted to case 2, 3 or 4
                }
                // case 2 sibling is black and both of its children are black
                if (isBlack(w.left) && isBlack(w.right)) {
                    w.color = ColorEnum.RED;
                    x = (RedBlackNode)x.parent;
                } else if (w != nilNode) {
                    if (isBlack(w.left)) { // case 3 sibling is black and its right child is red and left child is black
                        ((RedBlackNode)w.right).color = ColorEnum.BLACK;
                        w.color = ColorEnum.RED;
                        rotateLeft(w);
                        w = (RedBlackNode)x.parent.left;
                    }
                    w.color = ((RedBlackNode)x.parent).color; // case 4 sibling is black and left child is red
                    ((RedBlackNode)x.parent).color = ColorEnum.BLACK;
                    ((RedBlackNode)w.left).color = ColorEnum.BLACK;
                    rotateRight(x.parent);
                    x = (RedBlackNode)root;
                } else {
                    x.color = ColorEnum.BLACK;
                    x = (RedBlackNode)x.parent;
                }
            }

        }
    }

    private boolean isBlack(Node node) {
        return node != null ? ((RedBlackNode)node).color == ColorEnum.BLACK : false;
    }

    private boolean isRed(Node node) {
        return node != null ? ((RedBlackNode)node).color == ColorEnum.RED : false;
```

```
255     }
256 
257     /**
258      * Restores Red-Black tree properties after insert if needed. Insert can
259      * break only 2 properties: root is red or if node is red then children must
260      * be black.
261      */
262     private void insertRBFixup(RedBlackNode currentNode) {
263         // current node is always RED, so if its parent is red it breaks
264         // Red-Black property, otherwise no fixup needed and loop can terminate
265         while (currentNode.parent != root && ((RedBlackNode)
currentNode.parent).color == ColorEnum.RED) {
266             RedBlackNode parent = (RedBlackNode) currentNode.parent;
267             RedBlackNode grandParent = (RedBlackNode) parent.parent;
268             if (parent == grandParent.left) {
269                 RedBlackNode uncle = (RedBlackNode) grandParent.right;
270                 // case1 - uncle and parent are both red
271                 // re color both of them to black
272                 if (((RedBlackNode) uncle).color == ColorEnum.RED) {
273                     parent.color = ColorEnum.BLACK;
274                     uncle.color = ColorEnum.BLACK;
275                     grandParent.color = ColorEnum.RED;
276                     // grandparent was recolored to red, so in next iteration we
277                     // check if it does not break Red-Black property
278                     currentNode = grandParent;
279                 }
280                 // case 2/3 uncle is black - then we perform rotations
281                 else {
282                     if (currentNode == parent.right) { // case 2, first rotate
left
283                         currentNode = parent;
284                         rotateLeft(currentNode);
285                     }
286                     // do not use parent
287                     parent.color = ColorEnum.BLACK; // case 3
288                     grandParent.color = ColorEnum.RED;
289                     rotateRight(grandParent);
290                 }
291             } else if (parent == grandParent.right) {
292                 RedBlackNode uncle = (RedBlackNode) grandParent.left;
293                 // case1 - uncle and parent are both red
294                 // re color both of them to black
295                 if (((RedBlackNode) uncle).color == ColorEnum.RED) {
296                     parent.color = ColorEnum.BLACK;
297                     uncle.color = ColorEnum.BLACK;
298                     grandParent.color = ColorEnum.RED;
299                     // grandparent was recolored to red, so in next iteration we
300                     // check if it does not break Red-Black property
301                     currentNode = grandParent;
302                 }
303                 // case 2/3 uncle is black - then we perform rotations
304                 else {
```

```
305                     if (currentNode == parent.left) { // case 2, first rotate
    right
306                         currentNode = parent;
307                         rotateRight(currentNode);
308                     }
309                     // do not use parent
310                     parent.color = ColorEnum.BLACK; // case 3
311                     grandParent.color = ColorEnum.RED;
312                     rotateLeft(grandParent);
313                 }
314             }
315 
316         }
317         // ensure root is black in case it was colored red in fixup
318         ((RedBlackNode) root).color = ColorEnum.BLACK;
319     }
320 
321     protected static class RedBlackNode extends Node {
322         public ColorEnum color;
323 
324         public RedBlackNode(Integer value, Node parent, Node left, Node right,
    ColorEnum color) {
325             super(value, parent, left, right);
326             this.color = color;
327         }
328     }
329 
330 }
```

**未完待续。。。**