

如何解决好动态统计问题

广东省中山市第一中学 余江伟

wintokk@gmail.com

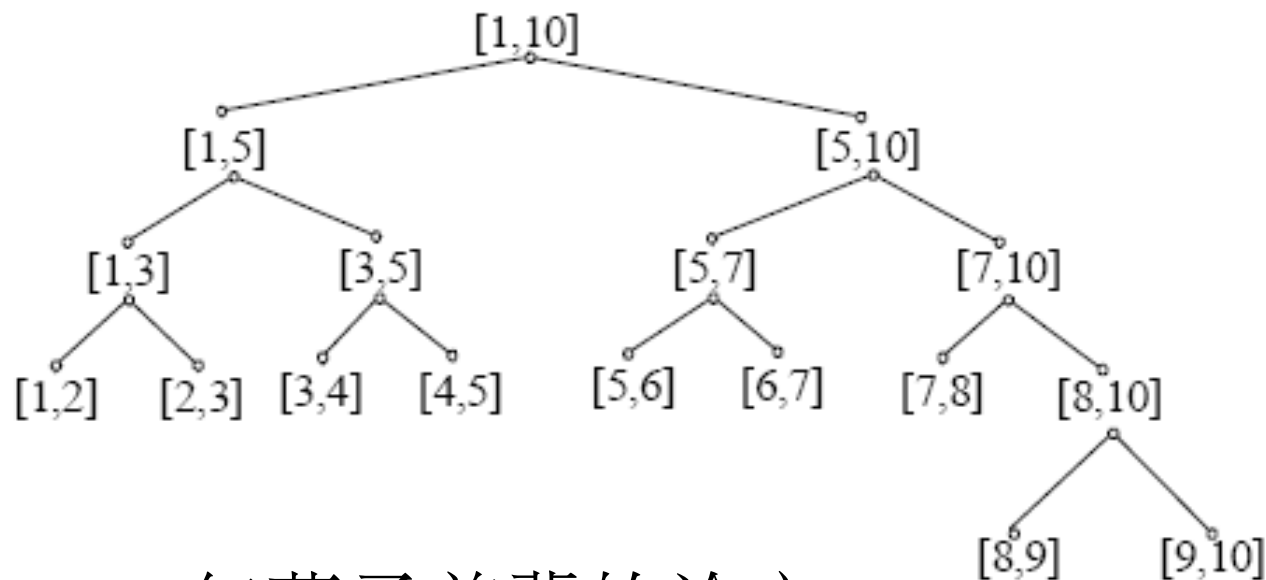
【引言】

- 在信息学竞赛中，统计问题十分常见。请看一个例子：
 - 在长度为 N ($2 \leq N \leq 10^6$) 的序列上进行 M 次以下操作：

操作		说明
增加	<i>Increase (i, j, c)</i>	第 i 项到第 j 项的值同时加上 c
询问	<i>Query (i, j)</i>	查询第 i 项到第 j 项的最大值和最小值

【引言】

- 利用线段树，可以轻松设计出时间复杂度 $O(M\log N)$ 、空间复杂度 $O(N)$ 的算法。



- 详见 2004 年薛矛前辈的论文

【引言】

- 线段树在本题取得成功的原因
 - 高效的组织结构
 - 很好地支持区间操作
 - 前提条件——本题中，序列项与项之间隐含着严格不变的次序关系
- 当统计对象次序发生大规模变化，线段树就显得力不从心了，必须寻找更优秀的解法

【例一】维护序列 (NOI2005)

- 写一个程序维护一个序列，支持 6 种操作：
 - **INSERT** $a \{c_n\}$
 - 在序列第 a 项后插入长度为 n 序列
 - **DELETE** $a b$
 - 删除序列的第 a 项到第 b 项
 - **MAKE-SAME** $a b c$
 - 把序列的第 a 项到第 b 项的值统一改为 c
 - **REVERSE** $a b$
 - 把序列的第 a 项到第 b 项首尾翻转后放回原位
 - **GET-SUM** $a b$
 - 输出序列的第 a 项到第 b 项的和
 - **MAX-SUM**
 - 求序列中和最大的一段非空子列，并输出最大和

【例一】维护序列 (NOI2005)

- 写一个程序维护一个序列
 - **INSERT** $a \{c_k\}$
 - **DELETE** $a b$
 - **MAKE-SAME** $a b c$
 - **REVERSE** $a b$
 - **GET-SUM** $a b$
 - **MAX-SUM**
- 任何时刻序列长度
 $1 \leq N \leq 500,000$
- 操作总数
 $1 \leq M \leq 20,000$
- 插入数的总数
 $1 \leq K \leq 4,000,000$

【例一】维护序列（NOI2005）

■ 初步分析

- 本题需要模拟一个序列的变化过程并随时统计相关求和信息
- 具有操作种类多、规模大的特点
- 朴素算法
 - 数组 / 链表模拟，只能拿到部分分数
- 更优秀的算法
 - 块状链表（参考解答）
 - 树形结构

【例一】维护序列 (NOI2005)

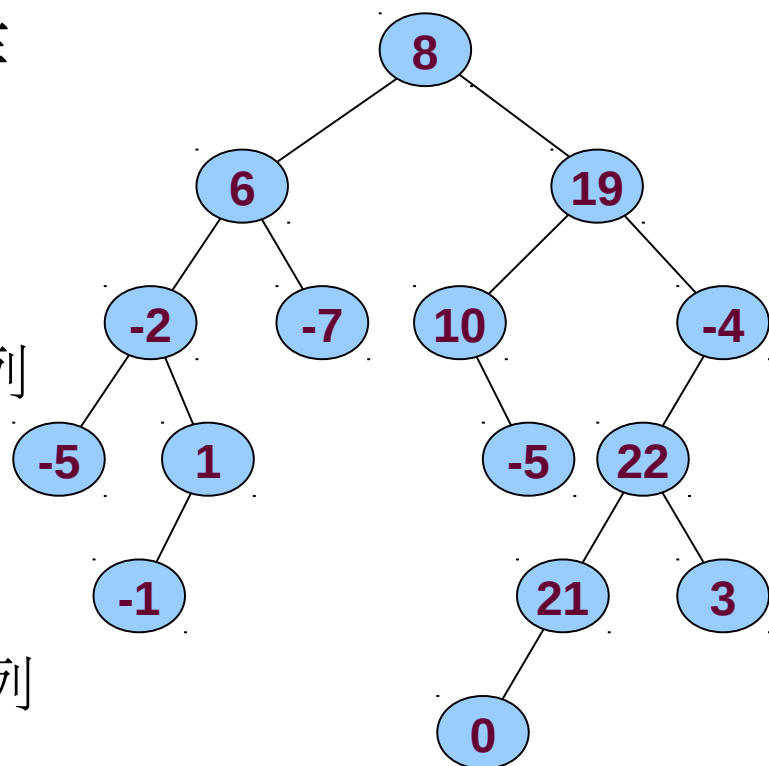
- 关键问题——表示 & 操作

【例一】维护序列 (NOI2005)

■ 关键问题——表示 & 操作

□ 如何表示

- 二叉查找树 (BST) 表示序列
- 每个节点记录一个数
- BST 中序遍历结果为原序列



一棵表示 $(-5, -2, -1, 1, 6, -7, 8, 10, -5, 19, 0, 21, 22, 3, -4)$ 的 BST

【例一】维护序列 (NOI2005)

- 关键问题——表示 & 操作
 - 如何操作
 - 不难发现，大多数操作都是围绕某个“连续段”进行的
 - “连续段”在 **BST** 中可能比较分散，我们希望把这些节点聚集起来
 - 伸展树

【例一】伸展树简介

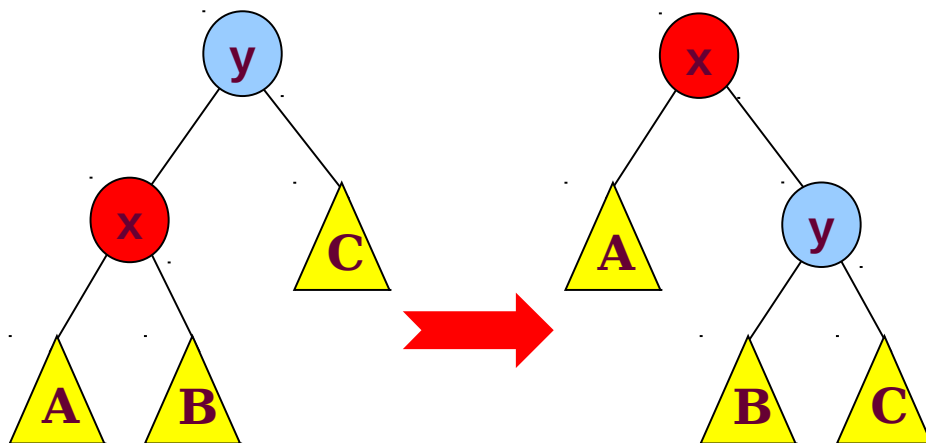
- 伸展树是一种自适应 (Self-Adjusting) 的 BST 。
具体地说，每次访问一个节点后，按照一定规则进行旋转，将其调整为树的根。

【例一】伸展树简介

- 伸展树是一种自适应 (Self-Adjusting) 的 BST 。
具体地说，每次访问一个节点后，按照一定规则进行旋转，将其调整为树的根。
- 伸展树的旋转规则
 - Zig/Zag
 - Zig-Zig/Zag-Zag
 - Zig-Zag/Zag-Zig

【例一】伸展树简介

- 伸展树是一种自适应 (Self-Adjusting) 的 BST 。
具体地说，每次访问一个节点后，按照一定规则进行旋转，将其调整为树的根。
- 伸展树的旋转规则
 - Zig/Zag
 - Zig-Zig/Zag-Zag
 - Zig-Zag/Zag-Zig

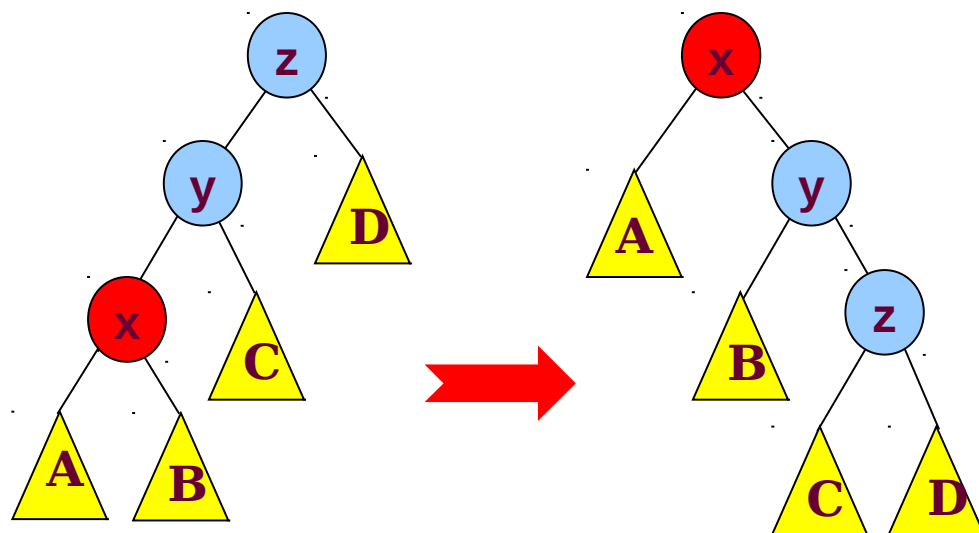


【例一】伸展树简介

- 伸展树是一种自适应 (Self-Adjusting) 的 BST 。
具体地说，每次访问一个节点后，按照一定规则进行旋转，将其调整为树的根。

- 伸展树的旋转规则

- Zig/Zag
- Zig-Zig/Zag-Zag
- Zig-Zag/Zag-Zig

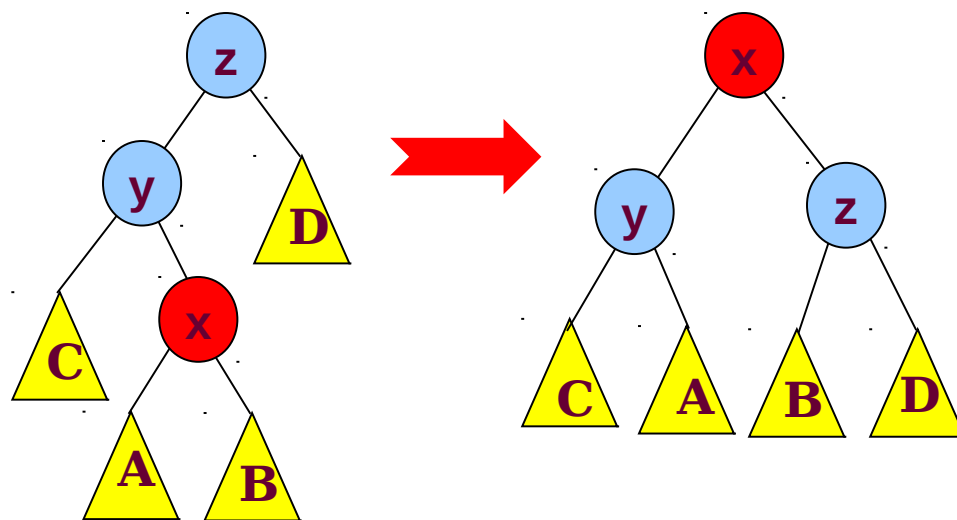


【例一】伸展树简介

- 伸展树是一种自适应 (Self-Adjusting) 的 BST 。
具体地说，每次访问一个节点后，按照一定规则进行旋转，将其调整为树的根。

- 伸展树的旋转规则

- Zig/Zag
- Zig-Zig/Zag-Zag
- Zig-Zag/Zag-Zig





【例一】伸展树简介

- 按照以上规则将节点调整到根的过程称为伸展操作。可以证明，伸展操作的平摊复杂度为 $O(\log N)$ 。
- 利用伸展操作，可以完成所有 **BST** 的基本操作。
- 针对本题，在节点上记录子树的大小 (**Size**)，可以实现第 **K** 个节点的查找功能 (**SplayKth**)。这也是解决本题的核心过程。

【例一】核心过程伪代码 (1)

SplayKth(p, kth)

// 把以 p 为根的子树下第 kth 个节点提到子树的根，并返回节点编号

```
1  if Size[Left[p]]+1 = kth then return p
2  if Size[Left[p]] ≥ kth
3  then x ← Left[p]
4      if Size[Left[x]]+1 = kth then return Zig( x, p )
5   if Size[Left[x]] ≥ kth then
6      return Zig-Zig( SplayKth(Left[x], kth), x, p )
7   kth ← kth - Size[Left[x]] - 1
8      return Zig-Zag( SplayKth(Right[x], kth), x, p )
9  else... // 目标节点在右子树的情况可类似处理
```

【例一】操作分析——插入

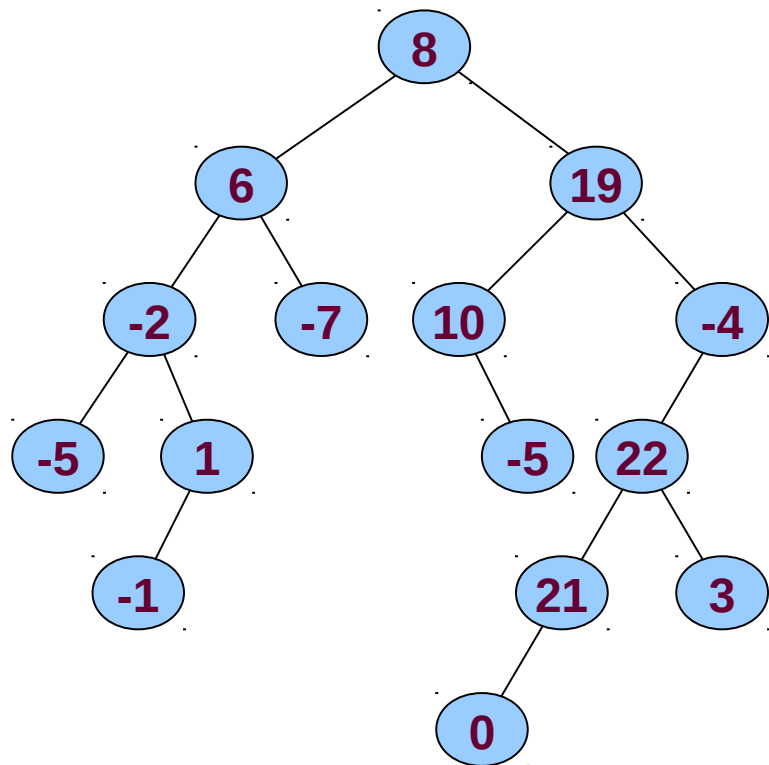
- 对一棵有 N 个节点、具有树根 $Root$ 的伸展树执行一次

$Root \leftarrow \text{SplayKth}(Root, K)$

所查询节点位于树根，前 $K-1$ 个节点被聚集在根的左子树上，后 $N-K$ 个节点被聚集在根的右子树上。

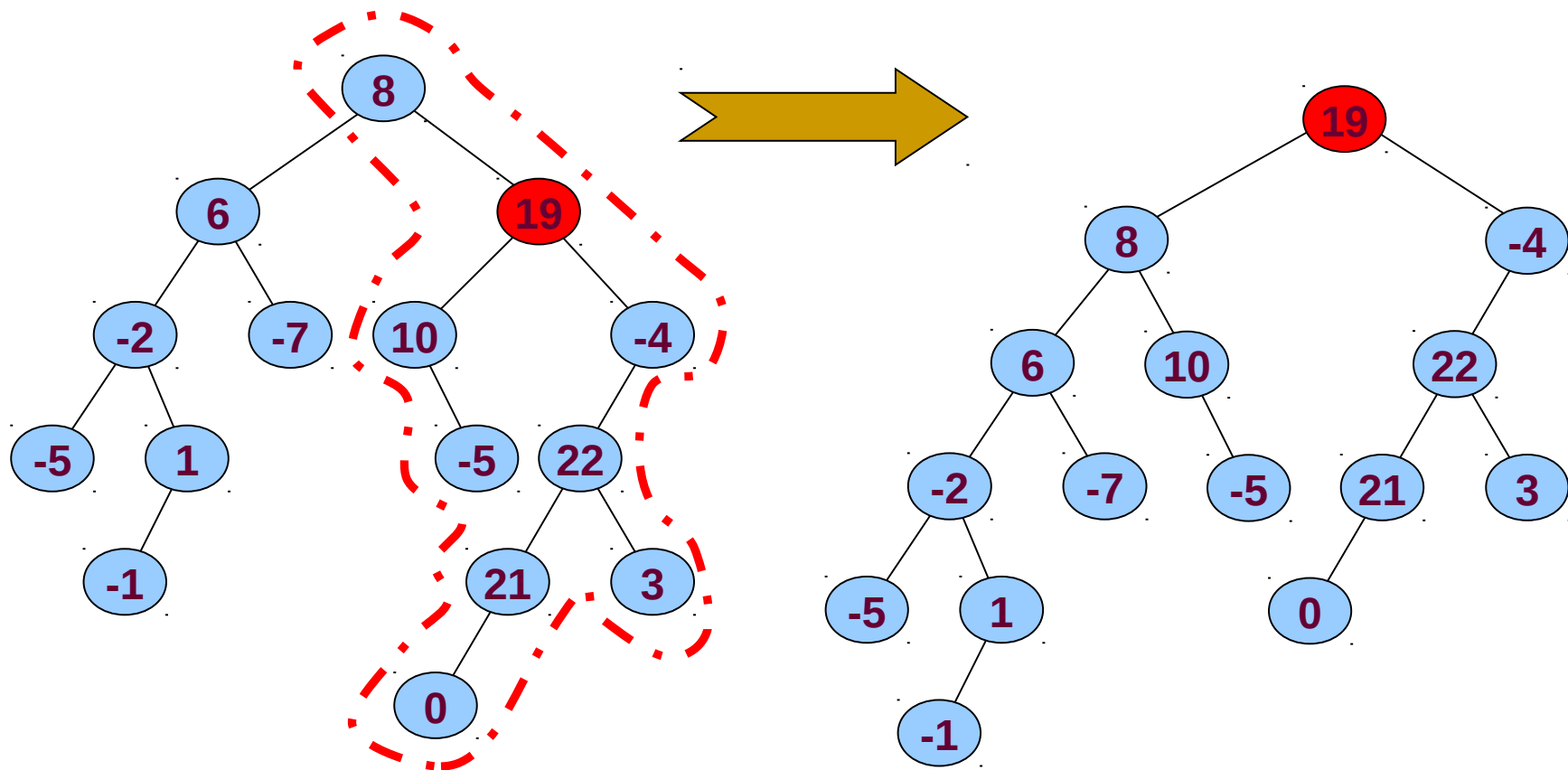
- 利用这个性质，可以实现对“连续段”的插入

【例一】操作分析——插入



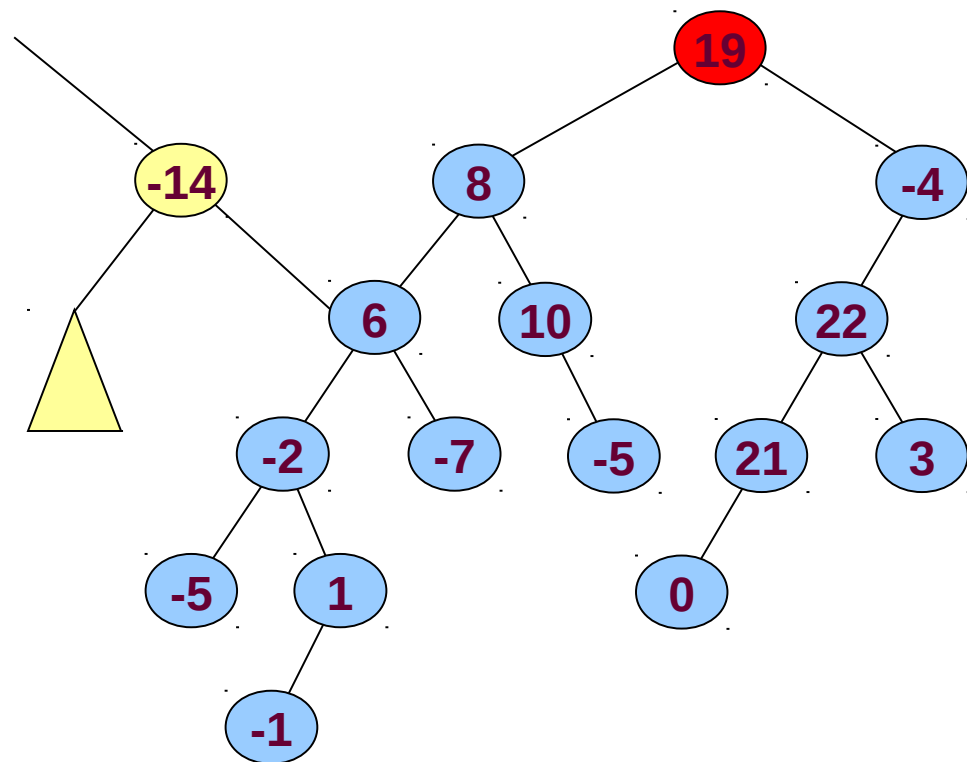
在第 **8** 项后插入一个“连续段”

【例一】操作分析——插入



找到第 8 项：红色节点 **19**

【例一】操作分析——插入



在藍色三角形樹之間插入“連續段”

【例一】操作分析——提取

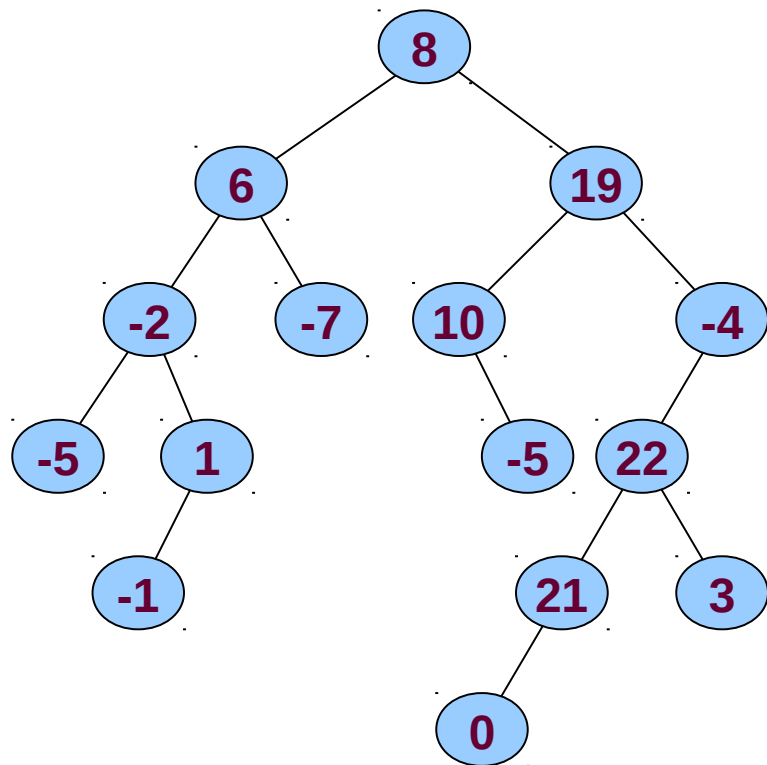
- 对一棵有 N 个节点、具有树根 $Root$ 的伸展树执行一次

$Root \leftarrow \text{SplayKth}(Root, K)$

所查询节点位于树根，前 $K-1$ 个节点被聚集在根的左子树上，后 $N-K$ 个节点被聚集在根的右子树上。

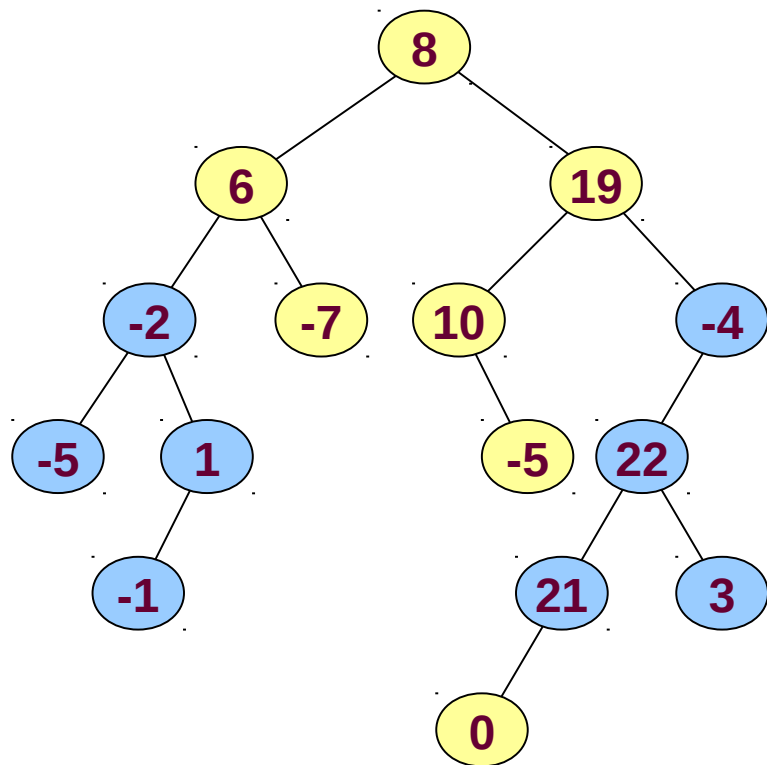
- 进一步地，在 $Root$ 的左子树上，再次利用这个性质，可以实现对“连续段”的提取

【例一】操作分析——提取



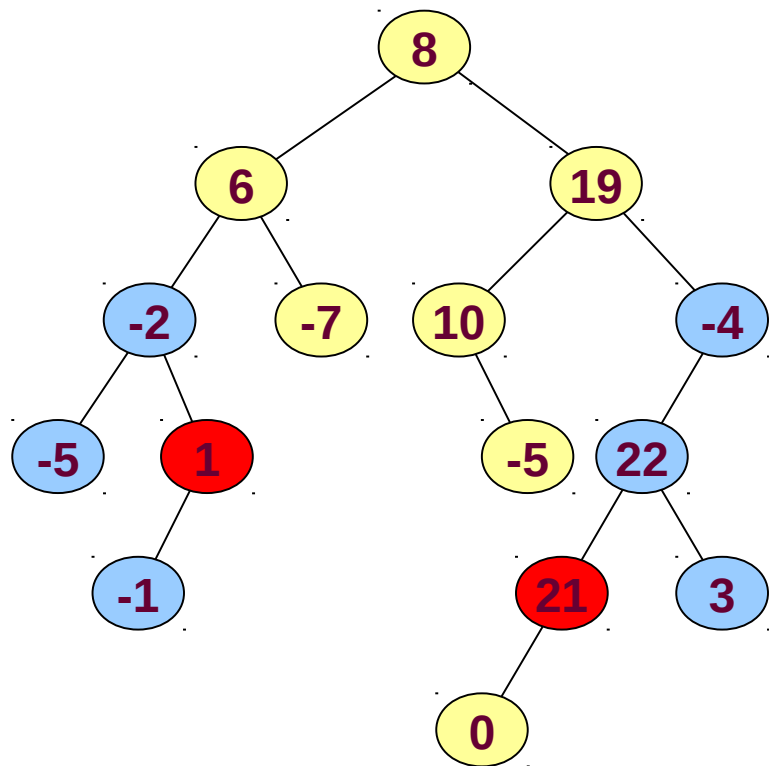
提取第 **5** ~ **11** 项

【例一】操作分析——提取



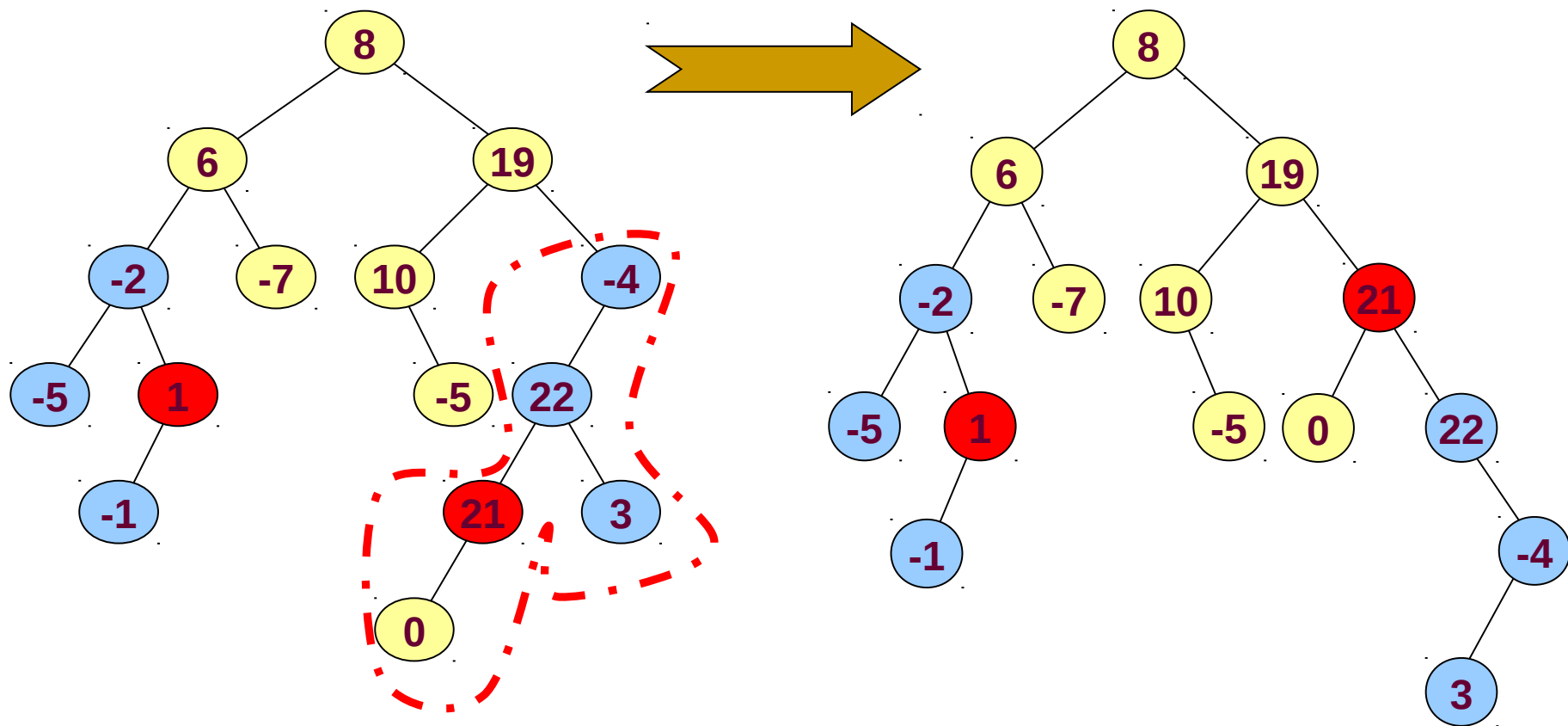
黄色节点为所求“连续段”

【例一】操作分析——提取



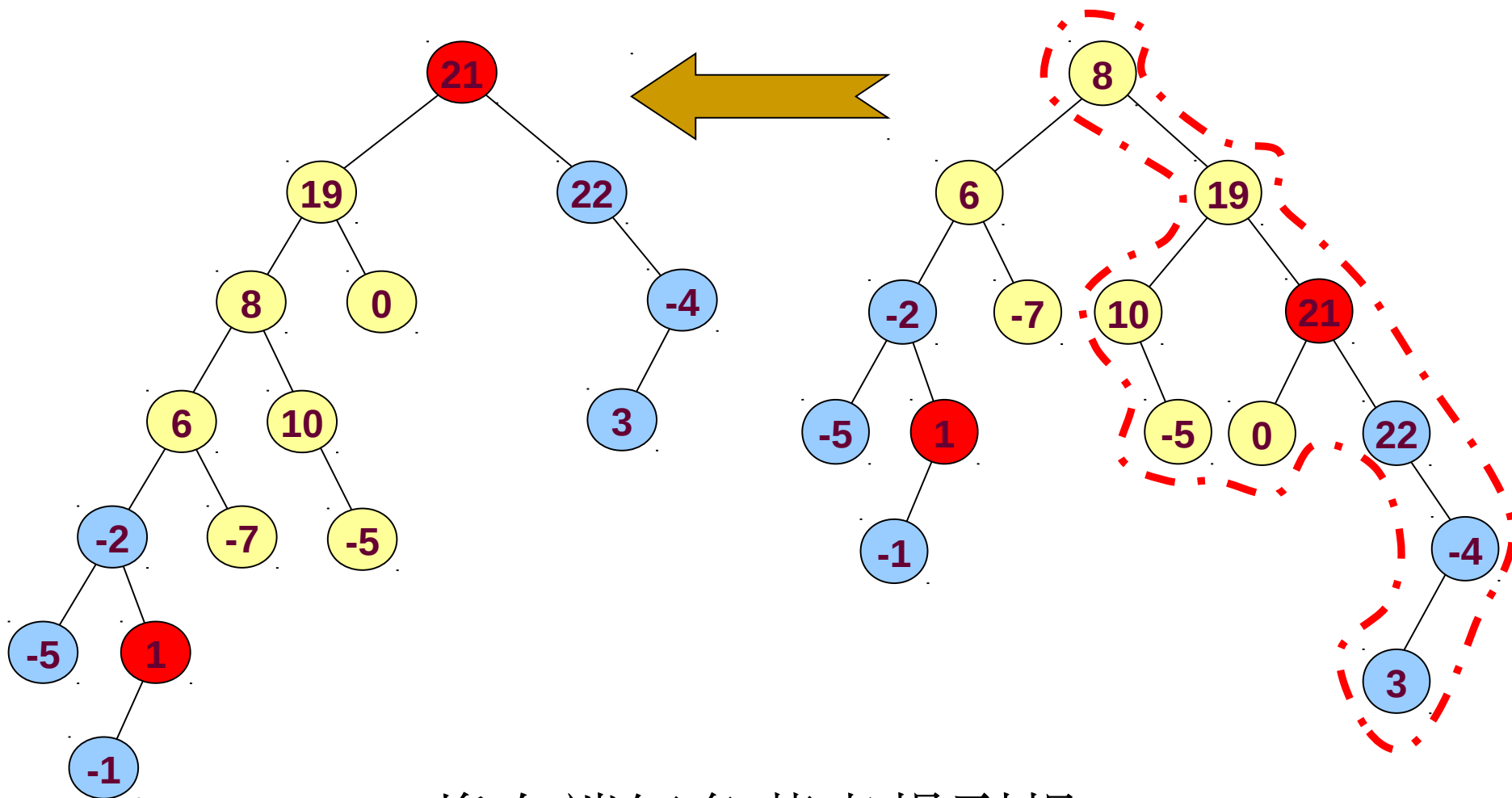
红色节点代表与“连续段”相邻的左右两项

【例一】操作分析——提取



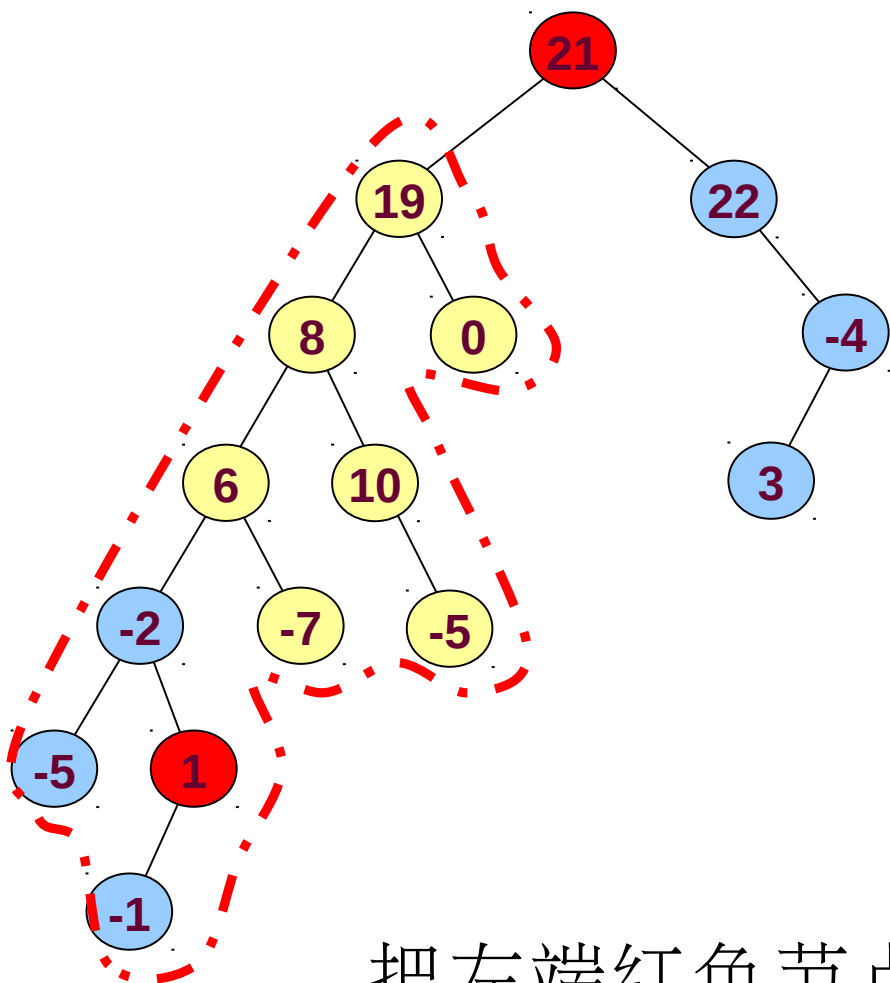
将右端红色节点提到根

【例一】操作分析——提取



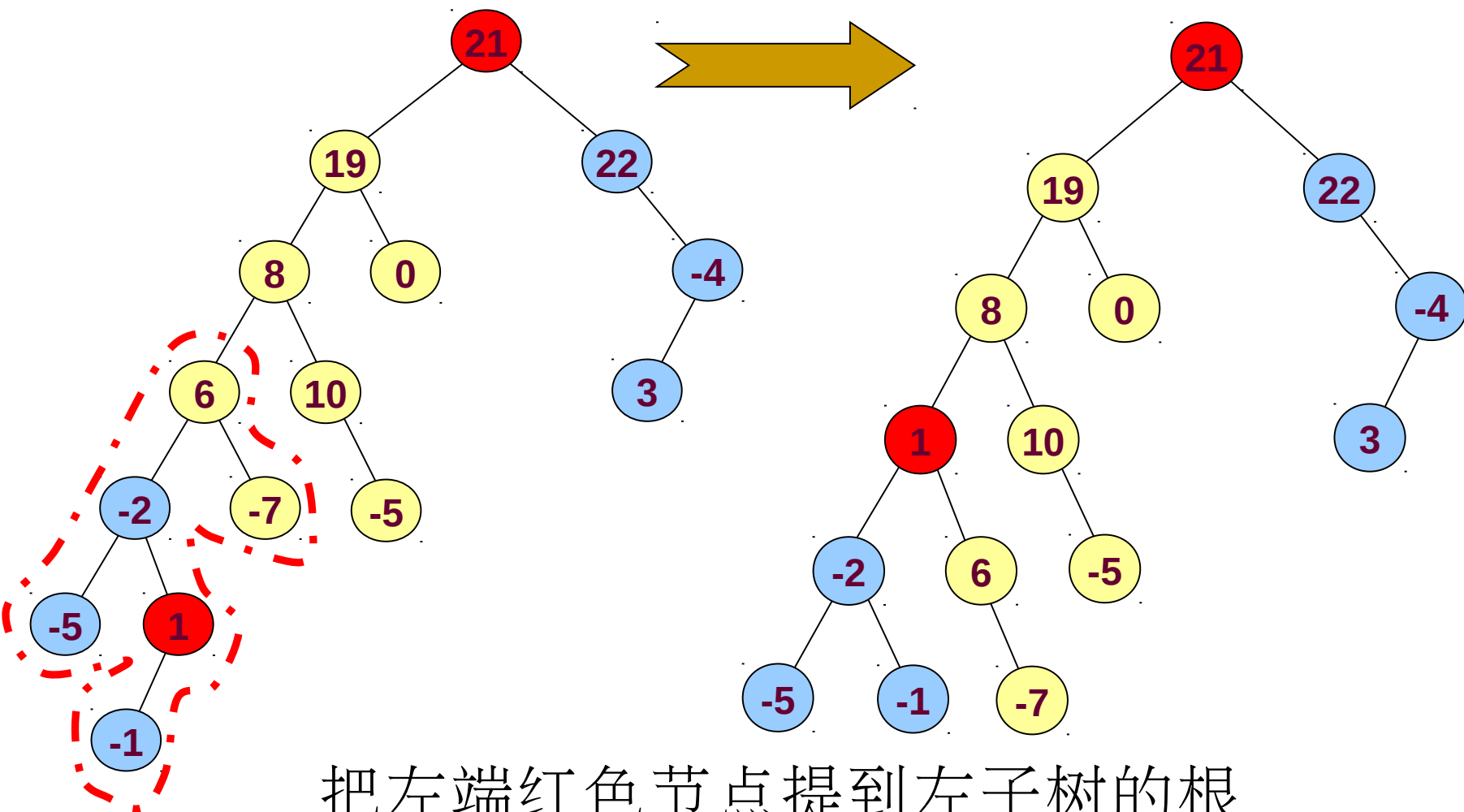
将右端红色节点提到根

【例一】操作分析——提取



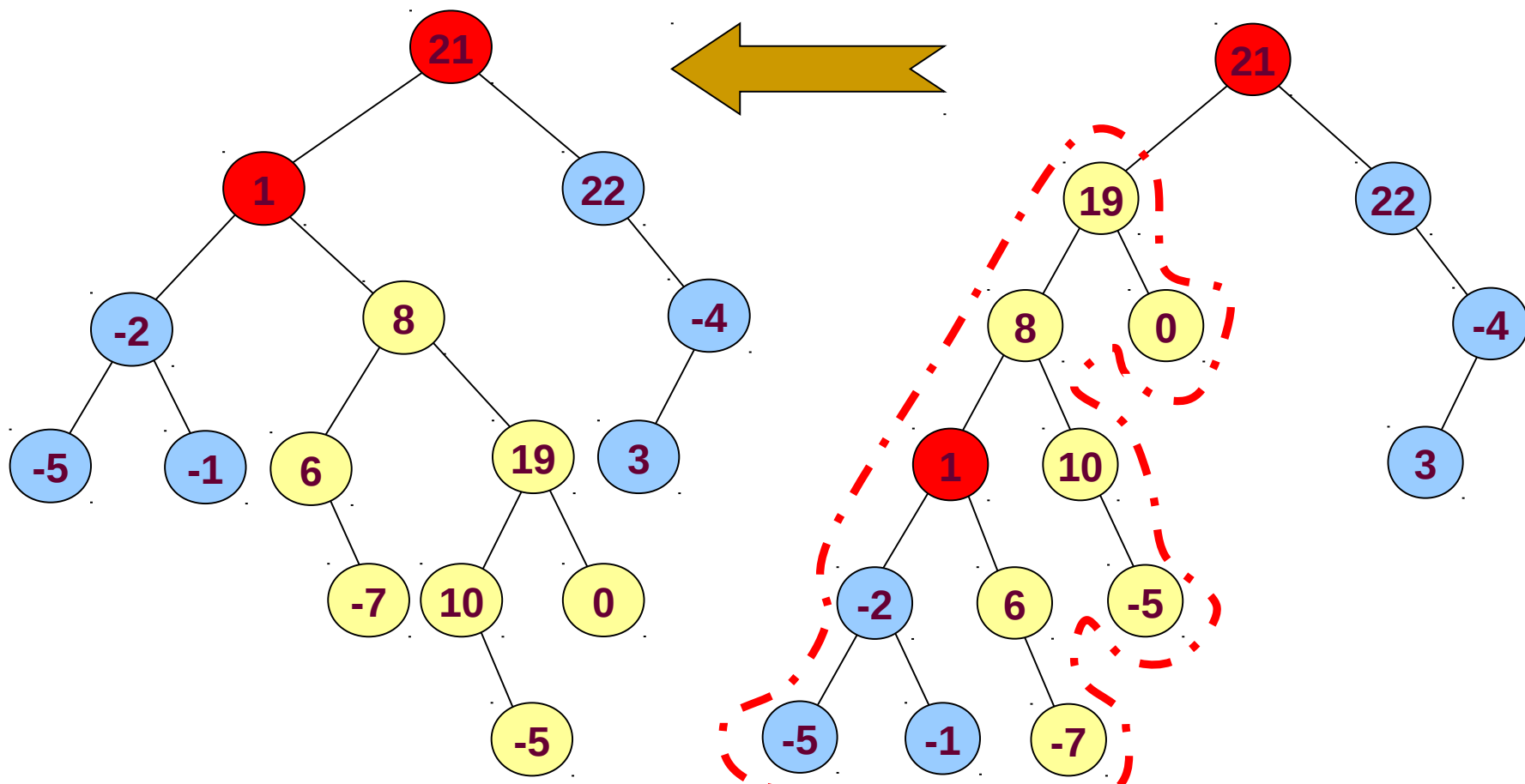
把左端红色节点提到左子树的根

【例一】操作分析——提取



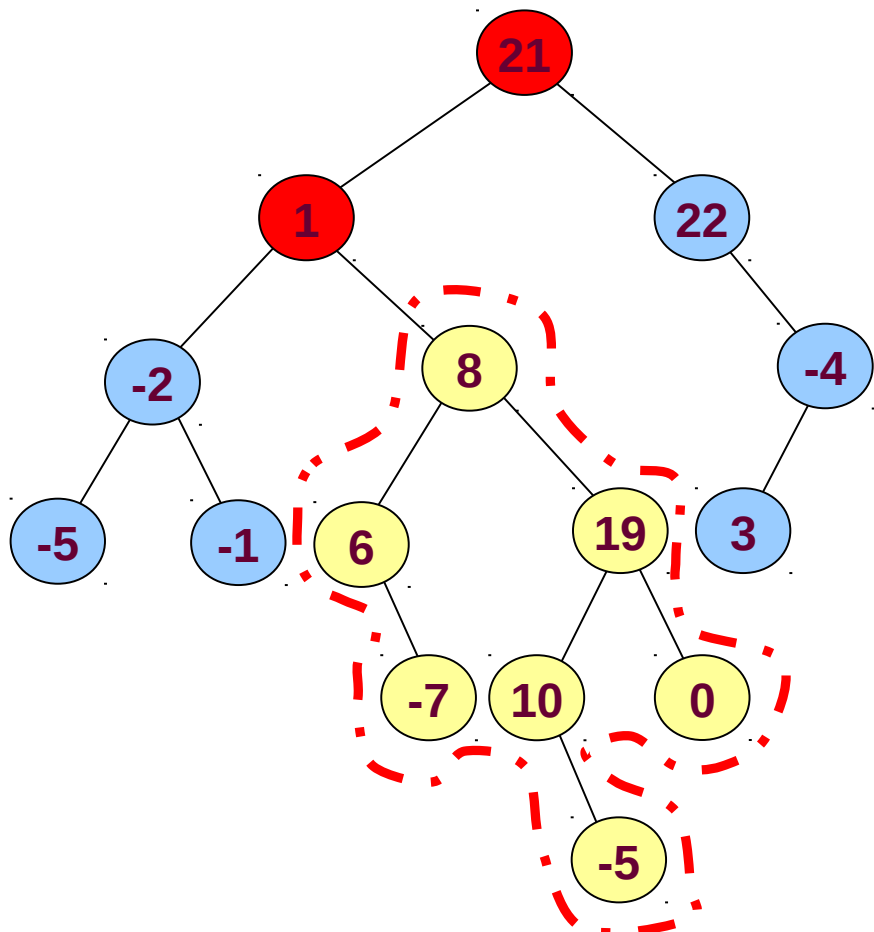
把左端红色节点提到左子树的根

【例一】操作分析——提取



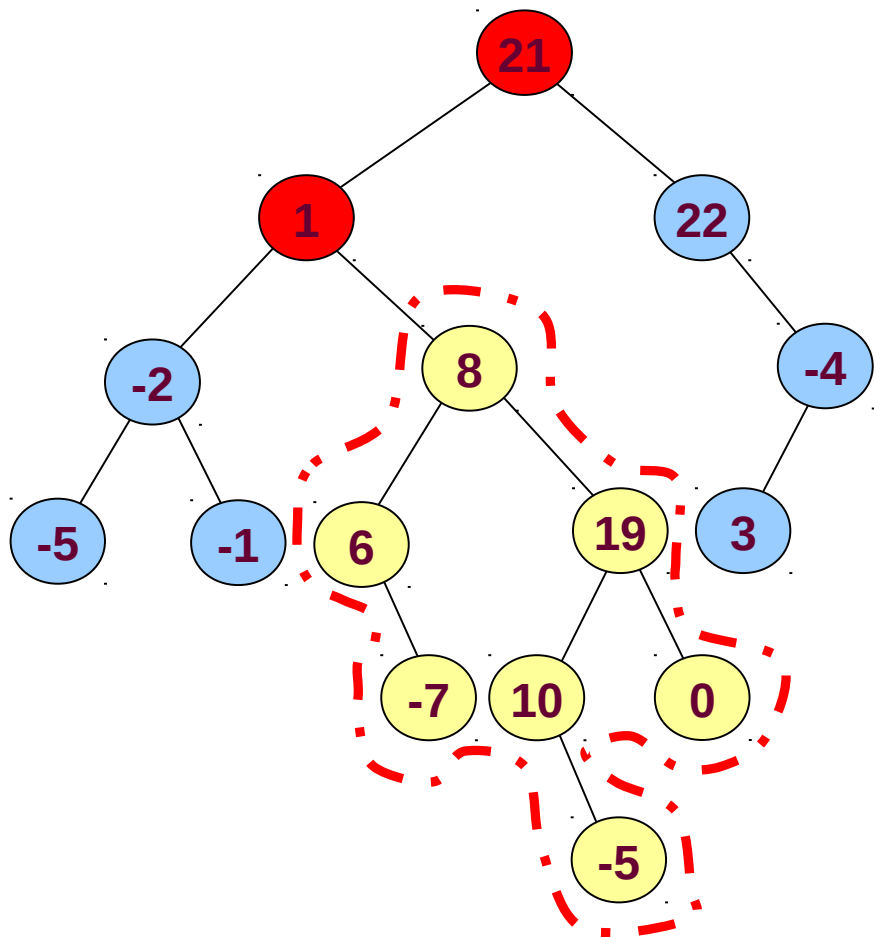
把左端红色节点提到左子树的根

【例一】操作分析——提取



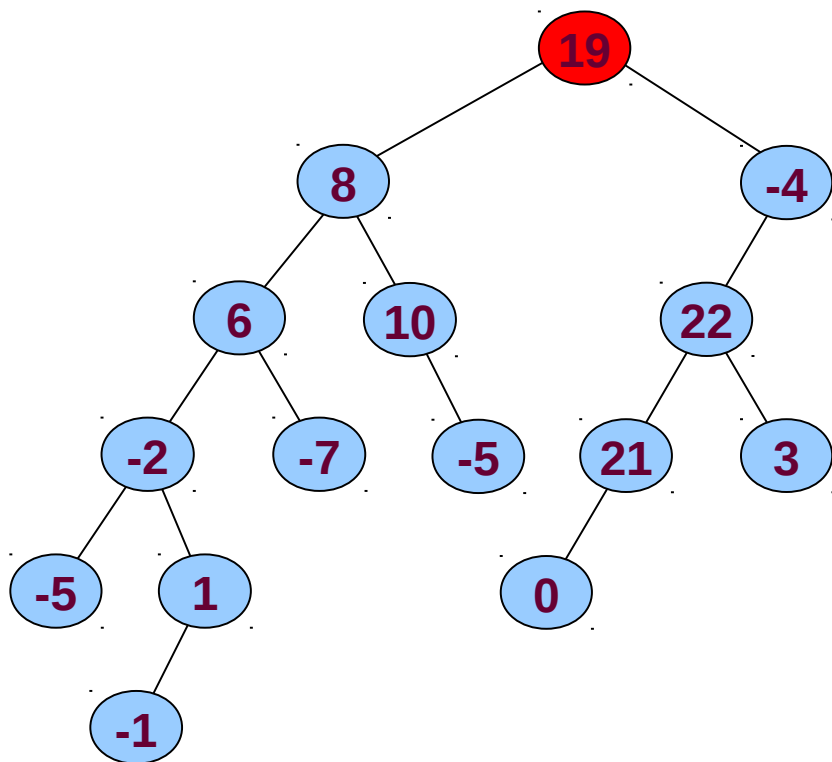
成功提取第 **5** ~ **11** 项！

【例一】操作分析



- 对“连续段”进行操作
 - Delete
 - 直接删除子树
 - Get-Sum
 - 维护每个节点的子树所赋值 (Value) 的和 (Sum) 即可
 - Reverse & Make-Same
 - 分别设置标记，访问节点前处理并向子树传递

【例一】操作分析



■ Max-Sum

- 维护信息
 - 子树内最大子列和 (AllMax)
 - 子树左 / 右起最大和 (LMax, RMax)
- 动态规划求解
- 直接输出根节点的 AllMax 值

【例一】细节处理

■ 注意事项

- 随着节点附加信息增多，需要适当修改核心过程 SplayKth
- 提取“连续段” $[a, b]$ 时，用到以下代码：

$$\begin{aligned} \text{Root} &\leftarrow \text{SplayKth}(\text{Root}, b+1) \\ \text{Left}[\text{Root}] &\leftarrow \text{SplayKth}(\text{Left}[\text{Root}], a-1) \end{aligned}$$

为避免边界情况 ($b=\text{Size}[\text{Root}]$ 或 $a=1$) 的讨论，在首尾增加两个空白节点，并把输入数据的位置 x 替换为 $x'=x+1$

【例一】核心过程伪代码 (2)

SplayKth(p, kth) // 把以 **p** 为根的子树下第 **kth** 个节点提到子树的根

1 **处理** *Left[p]* 和 *Right[p]* 的标记

2 if Size[Left[p]]+1 = kth then return p

3 if Size[Left[p]] ≥ kth

4 then x ← Left[p]

改变树的形态
后维护最大和

5 **处理** *Left[x]* 和 *Right[x]* 的标记

6 if Size[Left[x]]+1 = kth then return Zig(x, p)

7 if Size[Left[x]] ≥ kth then

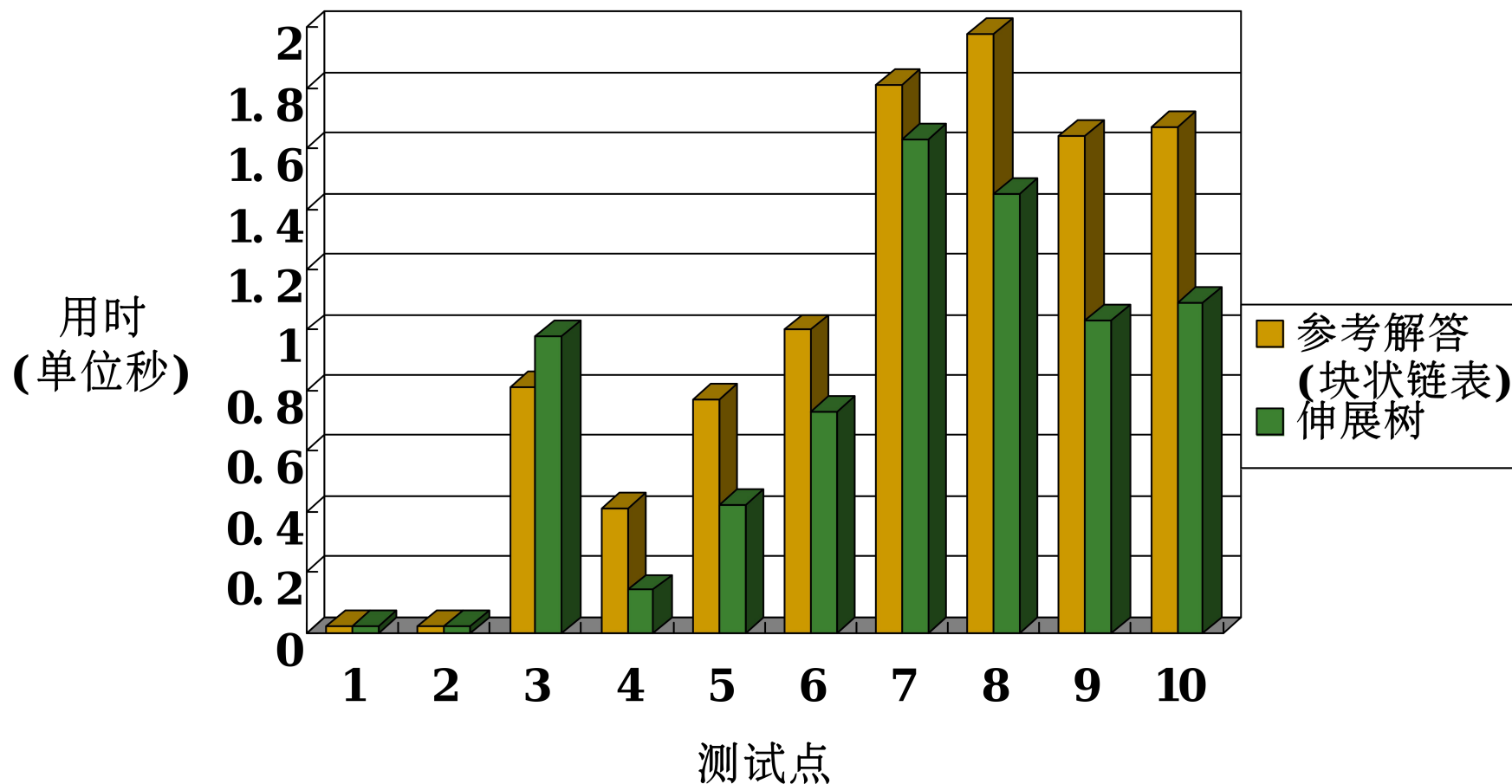
8 return Zig-Zig(SplayKth(Left[x], kth), x, p)

9 kth ← kth - Size[Left[x]] - 1

10 return Zig-Zag(SplayKth(Right[x], kth), x, p)

11 else... // 目标节点在右子树的情况可类似处理

【例一】性能比较



- 面对大规模的统计问题，对边界范围离散化，是减小问题规模的有效手段。
- 习惯上，离散化的过程在主算法之前执行。这是统计对象之间次序的固定所带来的便利。
- 如果统计对象次序发生变化，该如何应对？

【例二】文本编辑器 (NOI2003)

- 模拟一个字符串的变化过程。可能的操作有：
 - **INSERT**: 在光标处插入字符串
 - **DELETE**: 删除光标后 n 个字符
 - **GET**: 输出光标后 n 个字符，正确的输出总字符数不超过 3M
 - **MOVE**: 将光标移动到当前第 k 个字符之后
 - **PREV / NEXT**: 光标向前 / 后移动一位
- 数据范围
 - 插入的总字符数不超过 2^{21}
 - 插入 (Insert) 和删除 (Delete) 的总次数不超过 4000

【例二】文本编辑器 (NOI2003)

■ 模型分析

把光标移动的操作累积起来，需要时再定位，问题转化为【例一】的简化版：

- **INSERT**
- **DELETE**
- **GET**



总共不超过 4000 次

$$4000 \ll 2^{21}$$

虽然我们可以沿用【例一】提到的解法，但本题有更简单的算法。突破口就在于增删操作次数上限与插入字符总数上限之间的巨大落差！

【例二】文本编辑器 (NOI2003)

■ 感性认识

- 两个数值之间的差距意味着：在执行少量增删指令后，字符串的长度可能已经达到一个较大的数目，但字符间的次序关系并未被大规模打乱。

■ 理性分析

- 如果把每次插入的字符串看作一个具有连续属性的区间，那么我们所说的“打乱”就是指某次增删操作对区间的划分和排列。随着增删操作的进行，区间的划分也越来越细。这一过程即为离散化。

【例二】文本编辑器 (NOI2003)

■ 举例说明

操作
INSERT 0 "Balanced_tree"

区间划分与排列情况 & 输出字符串

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

【例二】文本编辑器 (NOI2003)

■ 举例说明

操作
INSERT 0 "Balanced_tree"

区间划分与排列情况 & 输出字符串
[1, 13]

B	a	l	a	n	c	e	d	_	T	r	e	e							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

【例二】文本编辑器 (NOI2003)

■ 举例说明

操作
INSERT 0 "Balanced_tree"
DELETE 3 7

区间划分与排列情况 & 输出字符串
[1, 13]
[1, 2] → [8, 13]

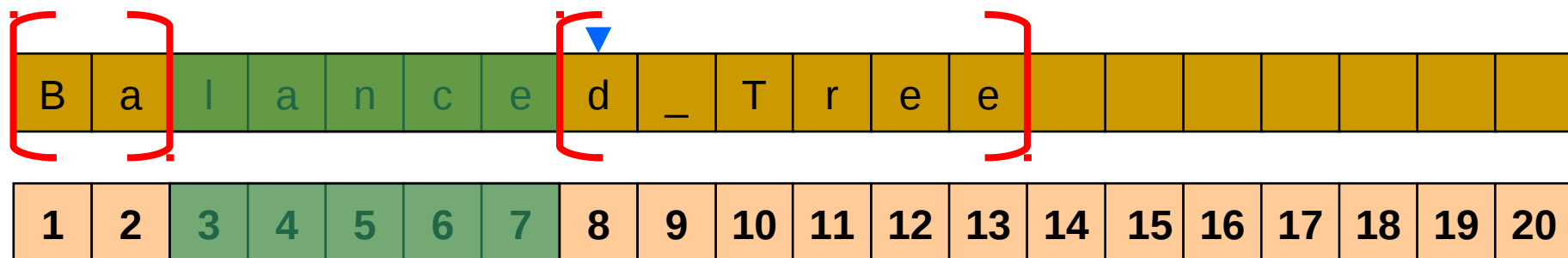
B	a	l	a	n	c	e	d	_	T	r	e	e							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

【例二】文本编辑器 (NOI2003)

■ 举例说明

操作
INSERT 0 "Balanced_tree"
DELETE 3 7
INSERT 3 "_editor"

区间划分与排列情况 & 输出字符串
[1, 13]
[1, 2] → [8, 13]

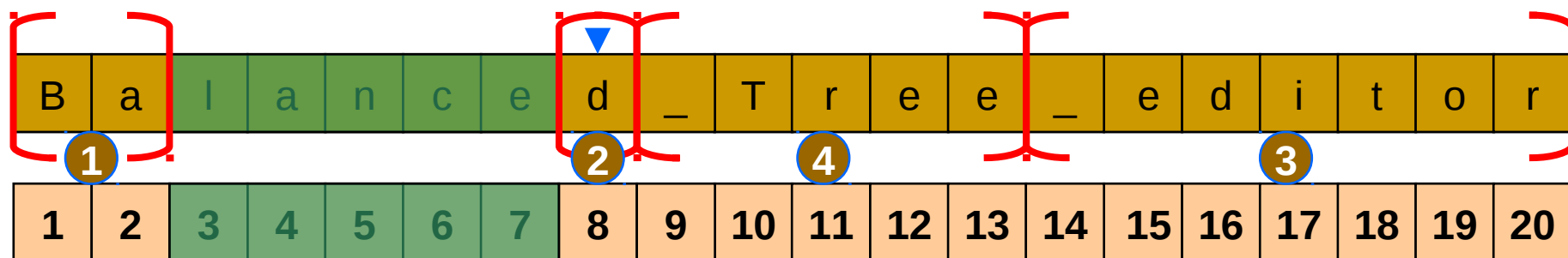


【例二】文本编辑器 (NOI2003)

■ 举例说明

操作
INSERT 0 "Balanced_tree"
DELETE 3 7
INSERT 3 "_editor"

区间划分与排列情况 & 输出字符串
[1, 13]
[1, 2] → [8, 13]
[1, 2] → [8, 8] → [14, 20] → [9, 13]

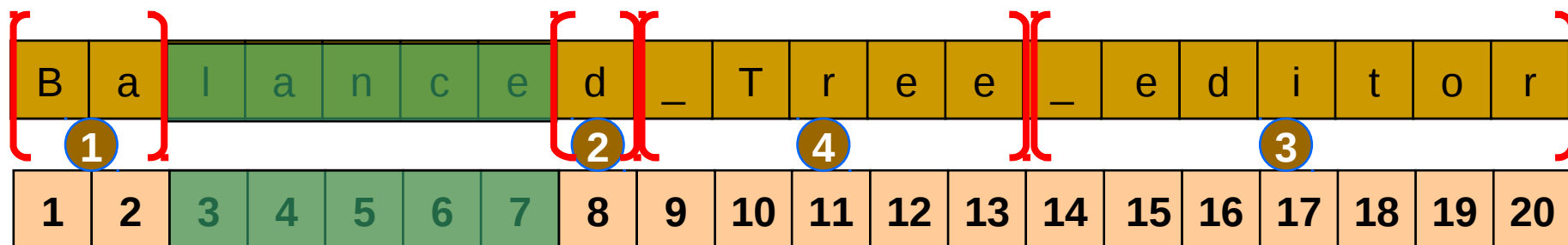


【例二】文本编辑器 (NOI2003)

■ 举例说明

操作
INSERT 0 "Balanced_tree"
DELETE 3 7
INSERT 3 "_editor"
GET [1, 16]

区间划分与排列情况 & 输出字符串
[1, 13]
[1, 2] → [8, 13]
[1, 2] → [8, 8] → [14, 20] → [9, 13]
"Bad_editor_Tree"



【例二】文本编辑器 (NOI2003)

■ 算法分析

- 把读入的字符串按顺序存储在大数组里。
- 在整个过程中，变化的只是各区间的端点与排列顺序，无需对字符串作任何改动。
- 每次增删操作新划分出的区间数不超过 2 个，而增删操作总数不超过 4000，所以总区间数不超过 8000。

离散化思想

字符串操作

问题规模大大降低

区间操作

低

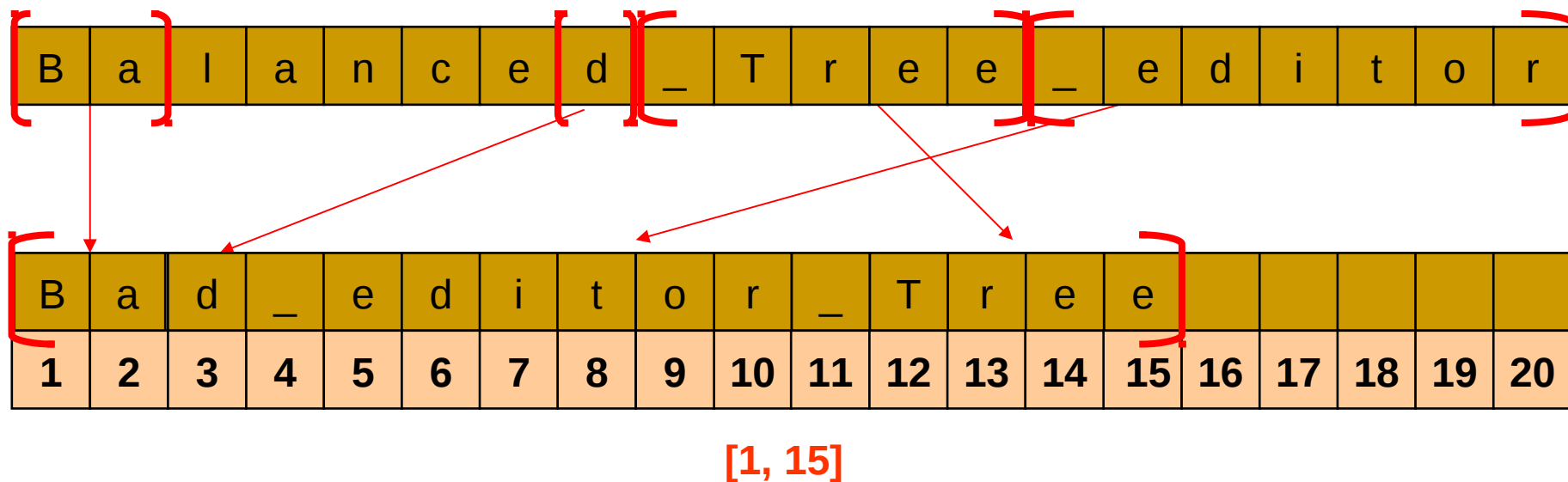
如何解决好动态统计问题
中山一中 余江伟

【例二】文本编辑器 (NOI2003)

■ 算法的优化

- 区间划分越来越细——区间总数越来越多
- 当区间总数达到一定数量 (*Limit*) 后，按顺序将多个区间合并成一个

[1, 2] → [8, 8] → [14, 20] → [9, 13]



【例二】文本编辑器 (NOI2003)

■ 算法的优化

- 一次整理的复杂度高达 $O(C)$ （ C 为字符串长度）
整理少了区间数量太多
整理多了反而成为瓶颈
- 利用几何平均思想，平衡两者的矛盾
- 理论上，在满足以下等式情况下复杂度最低：

$$2C = Limit^2$$

- 实际中，把区间上限设置为 1000 左右较好

【总结】

- 纵观整个过程，要解决好动态统计问题，需要灵活运用数据结构，并融入各种深刻的算法思想，从而高效地解决问题
- 面对纷繁复杂的问题，要经过冷静的分析，抓住共同点，写出紧凑的代码
- 遇到简化版的问题，不要轻易跳过，尝试一题多解，拓宽思路

纸上得来终觉浅，绝知此事要躬行

谢谢大家！