EDU

CSDN学院 IT实战派

图解数据结构和算法

树

讲师：Samuel

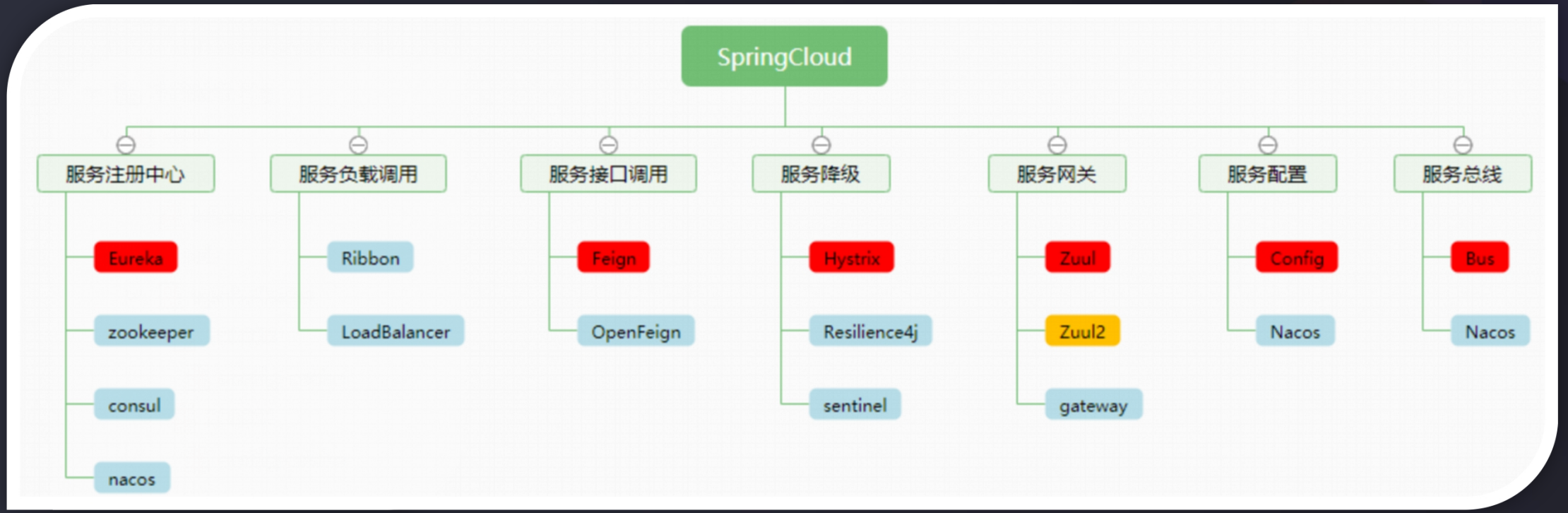
本章概述

1 为什么使用树

考虑一种使用场景：查找电脑上某个文件所的目录的逻辑 用代码实现出来

本章概述

◆树这种数据结构有着天然的使用场景....



本章概述

1 为什么使用树

- 数组存储方式的分析

  优点：通过下标方式访问元素，速度快。对于有序数组，还可使用二分查找提高检索速度。

  缺点：如果要插入或者删除值会整体移动，效率较低

- 链式存储方式的分析

  优点：在一定程度上对数组存储方式有优化

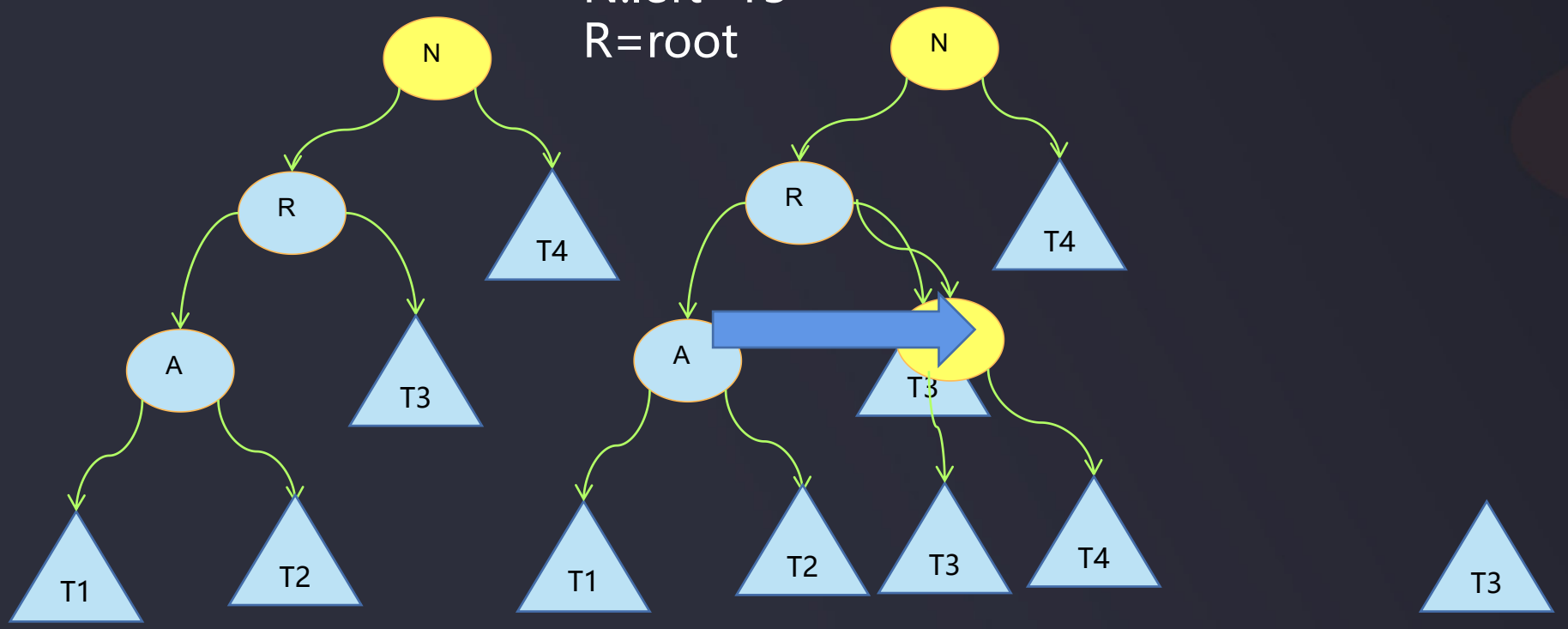
  缺点：在进行检索时，检索某个值，需要从头节点开始遍历，效率仍然较低

- 树存储方式的分析

  能提高数据存储，读取的效率，比如利用 二叉搜索树，既可以保证数据的检索速度，同时也可以保证数据的插入，删除，修改的速度

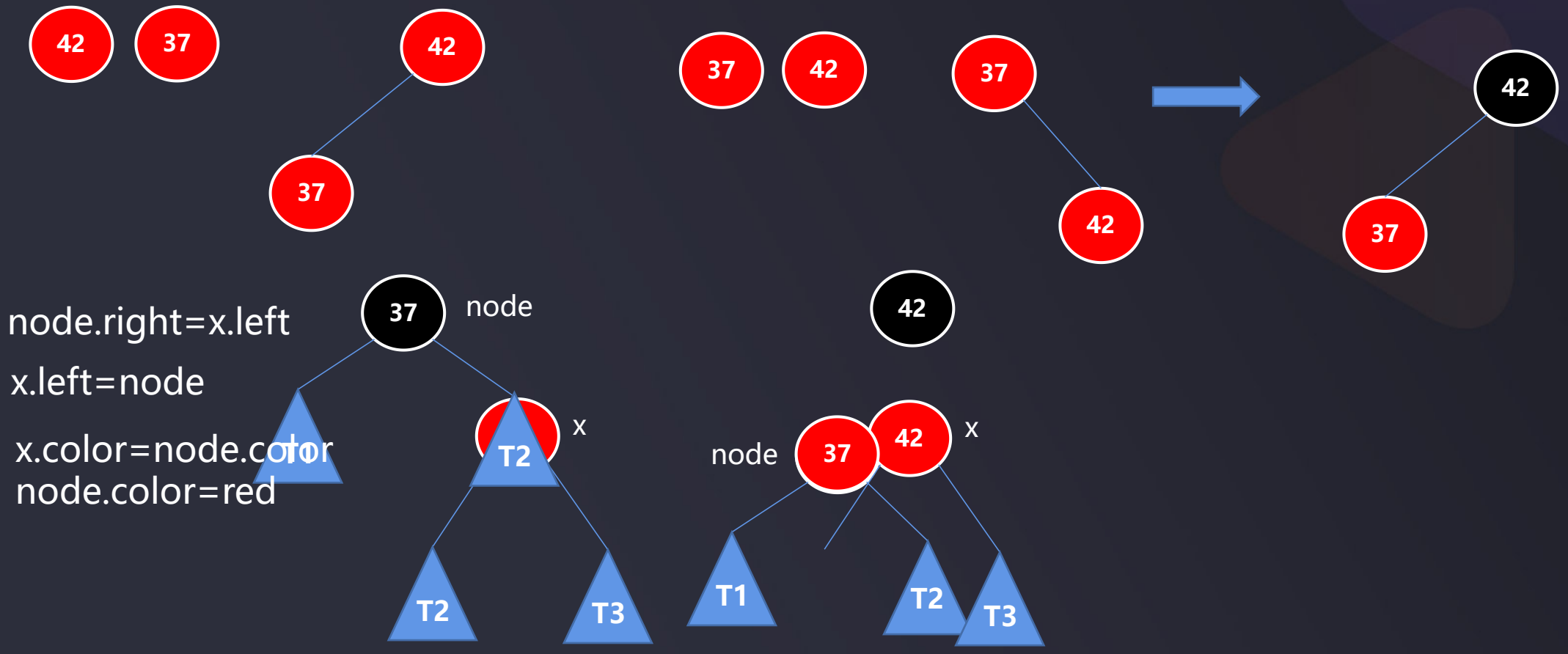
本章概述

R.right=N
N.left=T3
R=root



T1<A<T2<R<T3<N<T4

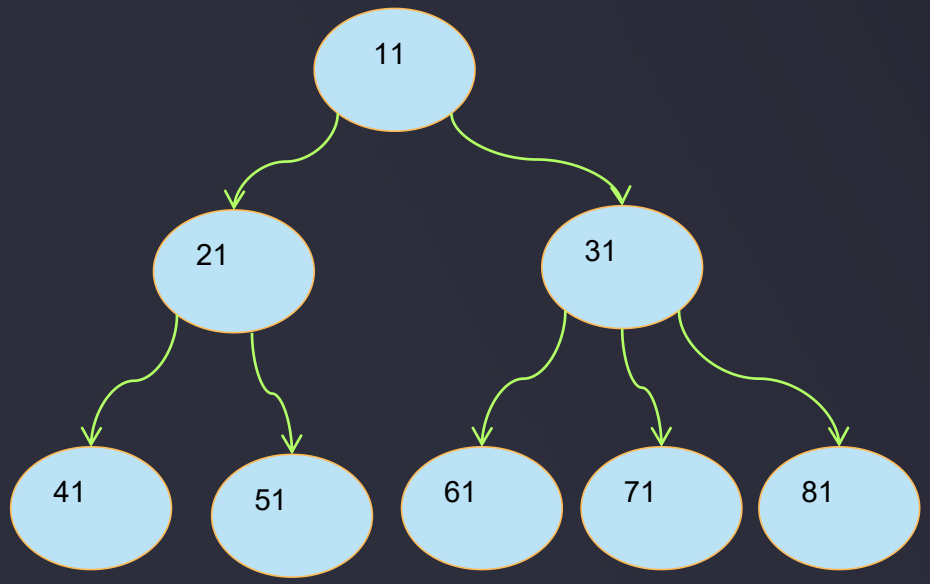
本章概述


42
37
42
37
37
42
37
42
42
37
node.right=x.left
x.left=node
x.color=node.color
node.color=red
37
node
T1
T2
x
T2
T3
42
node
37
42
x
T1
T2
T3

Content

树基本概念

2 树的常用术语

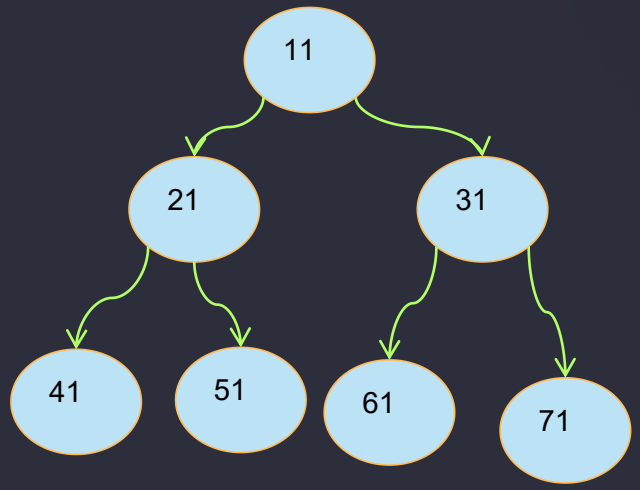


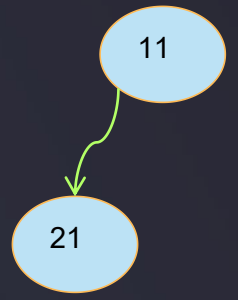
I. 节点
II. 根节点
III. 父节点
IV. 子节点
V. 叶子节点
VI. 节点的权
VII. 度
VIII. 路径
IX. 层
X. 子树
XI. 树的高度
XII. 森林

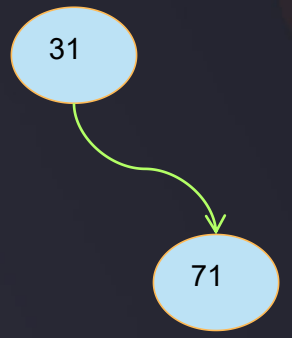
树基本概念

3 二叉树的概念

- 每个节点**最多只能有两个子节点**的一种形式的数称为二叉树
- 二叉树的子节点分为左节点和右节点



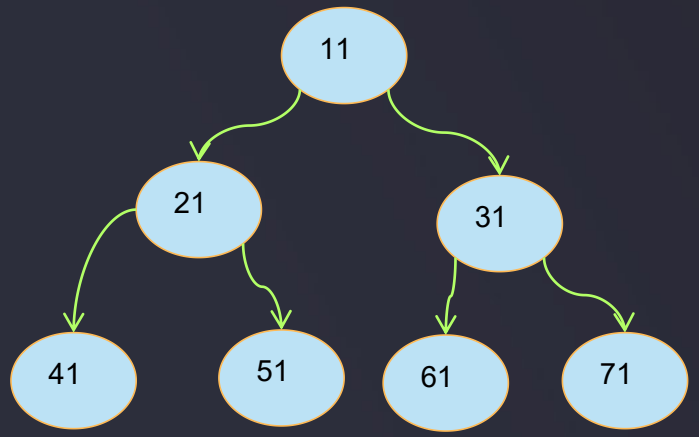




树基本概念

4 满二叉树

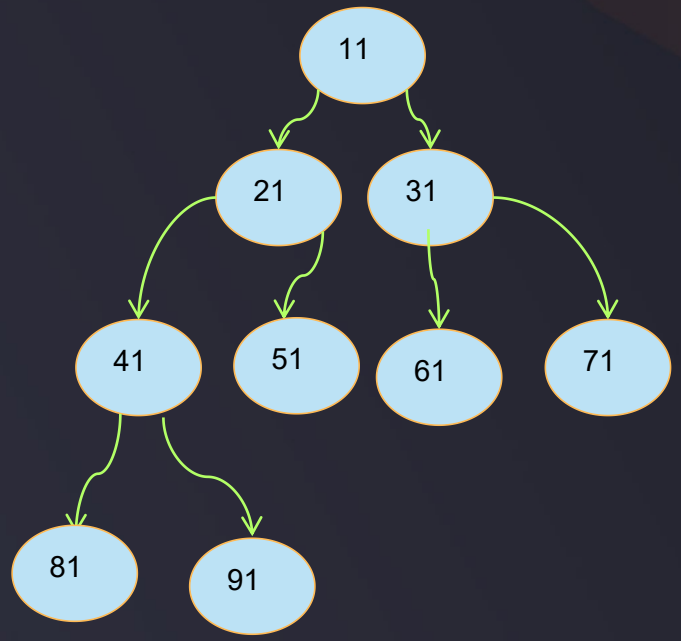
- 所有非叶子节点都存在左子树和右子树，并且所有叶子都在最后一层的二叉树为满二叉树
- 叶子节点只能在最后一层
- 非叶子节点的度一定是2
- 同样深度的二叉树中，满二叉树的节点个数最多，叶子数最多



树基本概念

5 完全二叉树

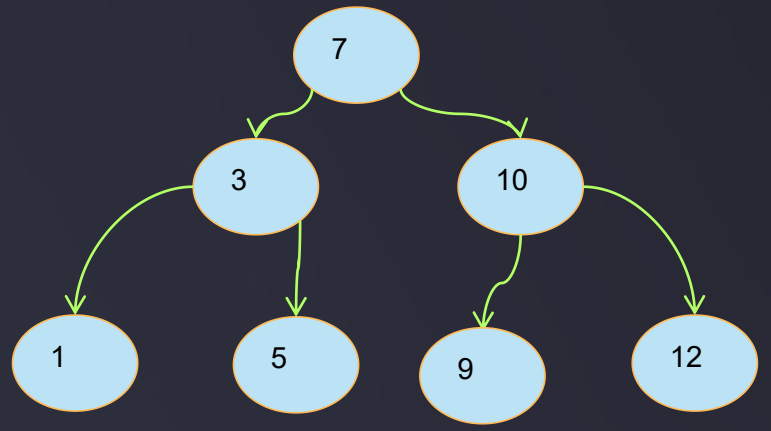
- 如果该二叉树的所有叶子节点都在最后一层或者倒数第二层，而且最后一层的叶子节点在左边连续倒数第二层的叶子节点在右边连续，我们称为完全二叉树
- 层数为n的完全二叉树，节点总数=2^n-1
- 如果节点的度是1，则该节点只有左孩子
- 同样节点数目的二叉树，完全二叉树深度最小
- 满二叉树一定是完全二叉树，反之则不一定



二叉搜索树BinarySearchTree

1 二叉搜索树概念

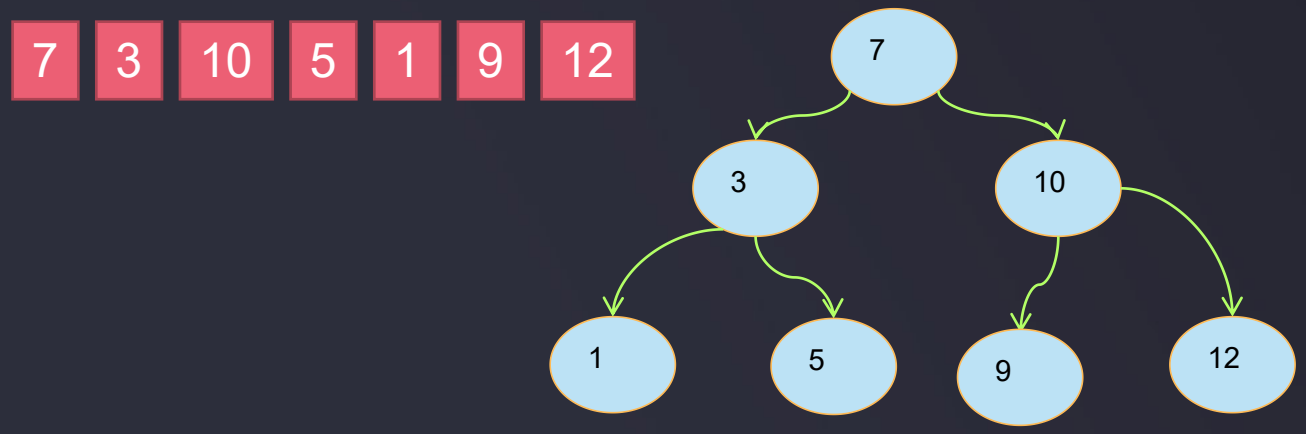
- 二叉搜索树也叫做二叉排序树，任何一个非叶子节点，要求左子节点的值比当前节点的值小，右子节点的值比当前节点的值大
- 如果有相同的值，可以将该节点放在左子节点或右子节点
- 数据 (7,3,10,12,5,1,9) ，对应的二叉排序树为



二叉搜索树BinarySearchTree

1 二叉搜索树的创建和查找

- 数据 (7, 3, 10, 5, 1, 9， 12)， 对应的二叉排序树创建过程如下



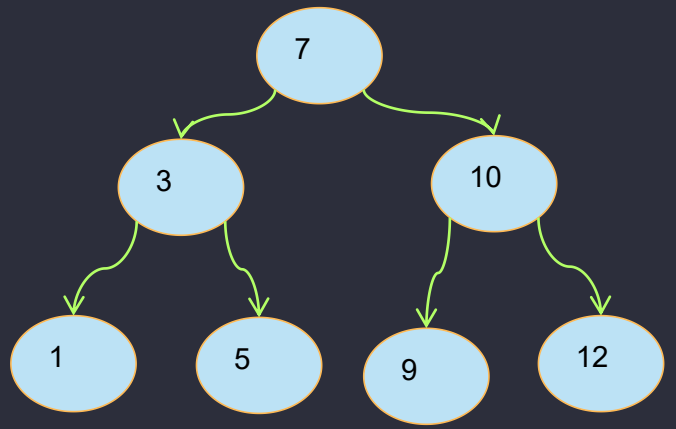
- 查找某一具体元素，可以使用的递归的方式

二叉搜索树BinarySearchTree

2 二叉搜索树的深度优先遍历

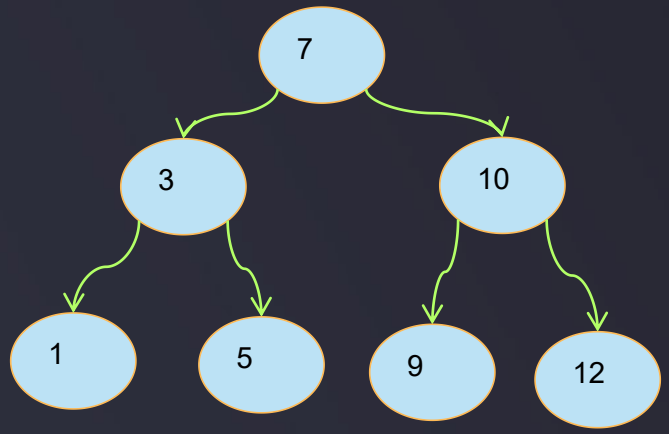
- 二叉树的深度遍历分为：前序遍历、中序遍历、后序遍历
- 前序遍历: 先输出父节点，再遍历左子树和右子树
- 中序遍历: 先遍历左子树，再输出父节点，再遍历右子树，中序遍历的结果是有序的
- 后序遍历: 先遍历左子树，再遍历右子树，最后输出父节点

前序遍历



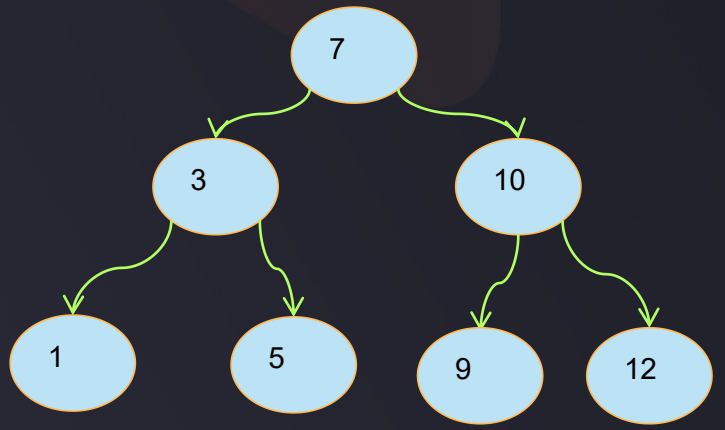
7 3 1 5 10 9 12

中序遍历



1 3 5 7 9 10 12

后序遍历

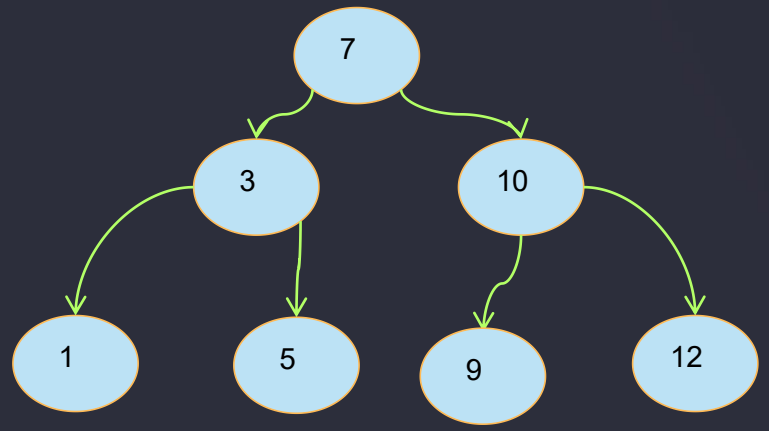


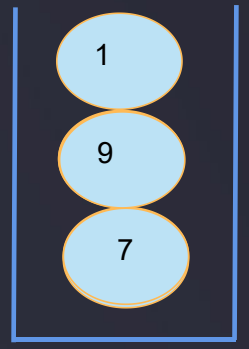
1 5 3 9 12 10 7

二叉搜索树BinarySearchTree

3 二叉搜索树非递归前序遍历

- 非递归前序遍历借助了栈
- 根压栈→根出栈→压根的右孩子和左孩子→所有孩子已入栈则栈顶出站→
  压入出栈节点的右孩子和左孩子→栈顶出站……

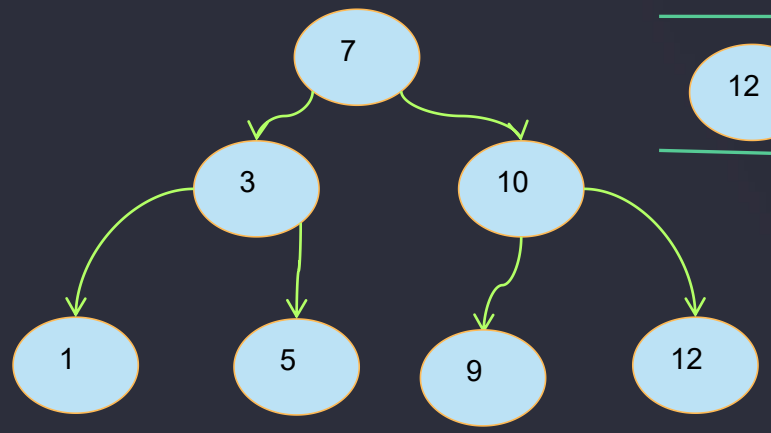


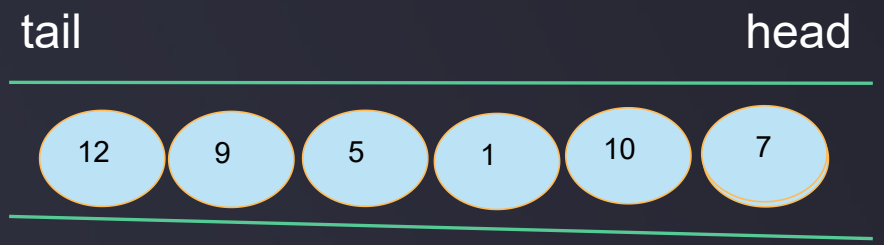


二叉搜索树BinarySearchTree

4 二叉搜索树的广度优先遍历

- 二叉树的广度优先遍历即层次遍历使用层次遍历可以更快的找到问题的解
- 常用于算法设计中，用作求最短路径
- 层次遍历借助队里来实现

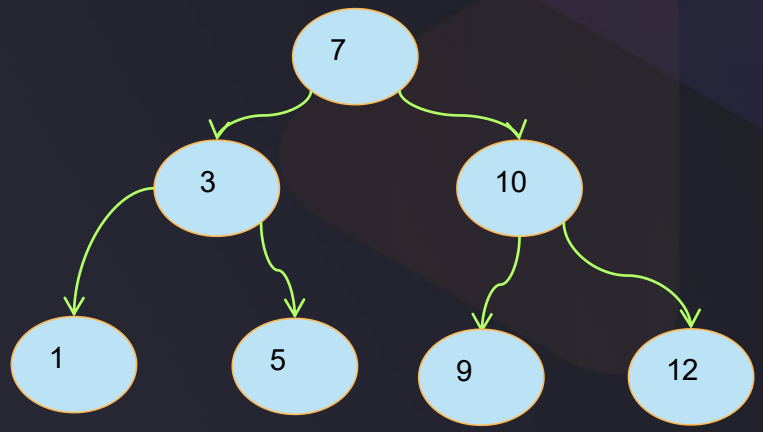




二叉搜索树BinarySearchTree

5 二叉搜索树的删除

- 数据 (7, 3, 10, 5, 1, 9,12) ， 对应的二叉排序树为
- 删除叶子节点 (比如：1, 5, 9,12)，则直接删除
- 删除只有一棵子树的节点 (比如删掉节点5后节点3)
  - 删除该节点，并将唯一的子树上移即可
- 删除有两棵子树的节点. (比如：3,7,10)
  - 该节点的左子树中最大节点替换他
  - 该节点的右子树中最小节点替换他

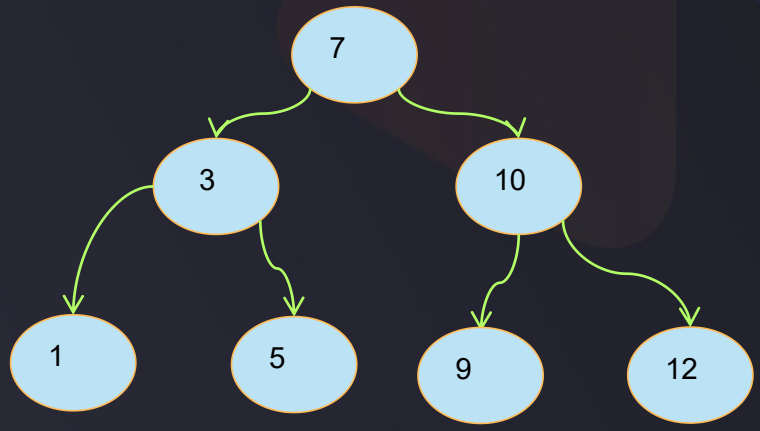


二叉搜索树BinarySearchTree

6 二叉搜索树总结

- 左子树都比root要小，右子树都要比root要大
- 两种实现添加加元素
- 查找元素所在节点以及查找元素的父节点
- 删除元素的两种实现方式
- 灵活应用：floor和ceil、rank和select

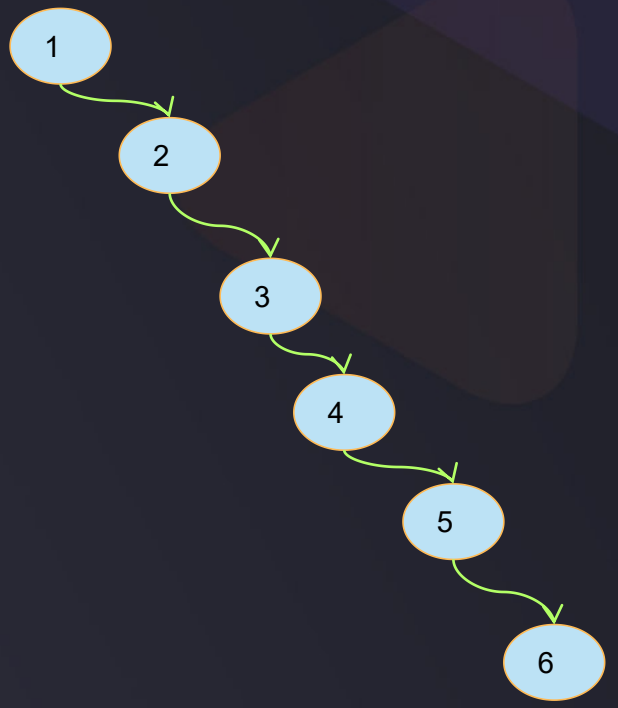
寻找8的floor和ceil



平衡二叉树AVL

1 二叉搜索树的问题

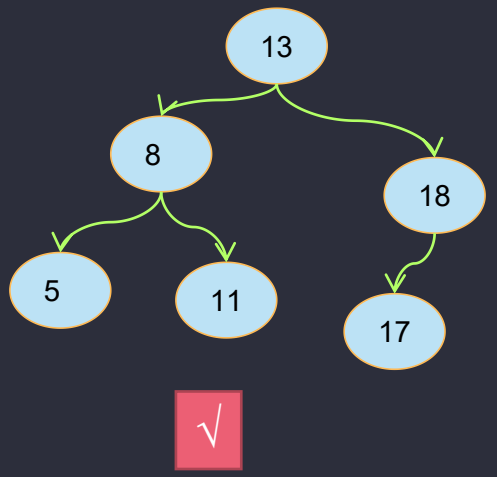
- 数据 (1,2,3,4,5,6) ,创建一棵BST
- 左子树全部为空，从形式上看，更像一个单链表
- 插入速度没有影响
- 查询速度明显降低
- 解决方案-平衡二叉树

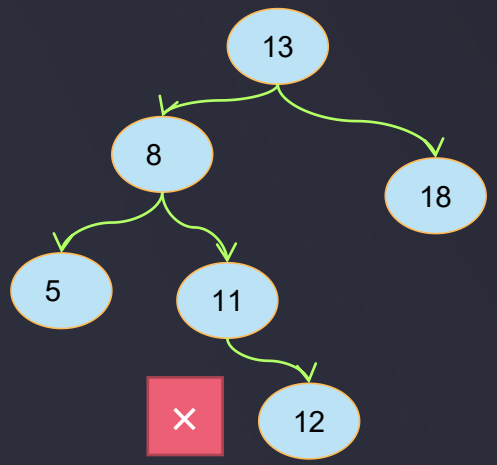


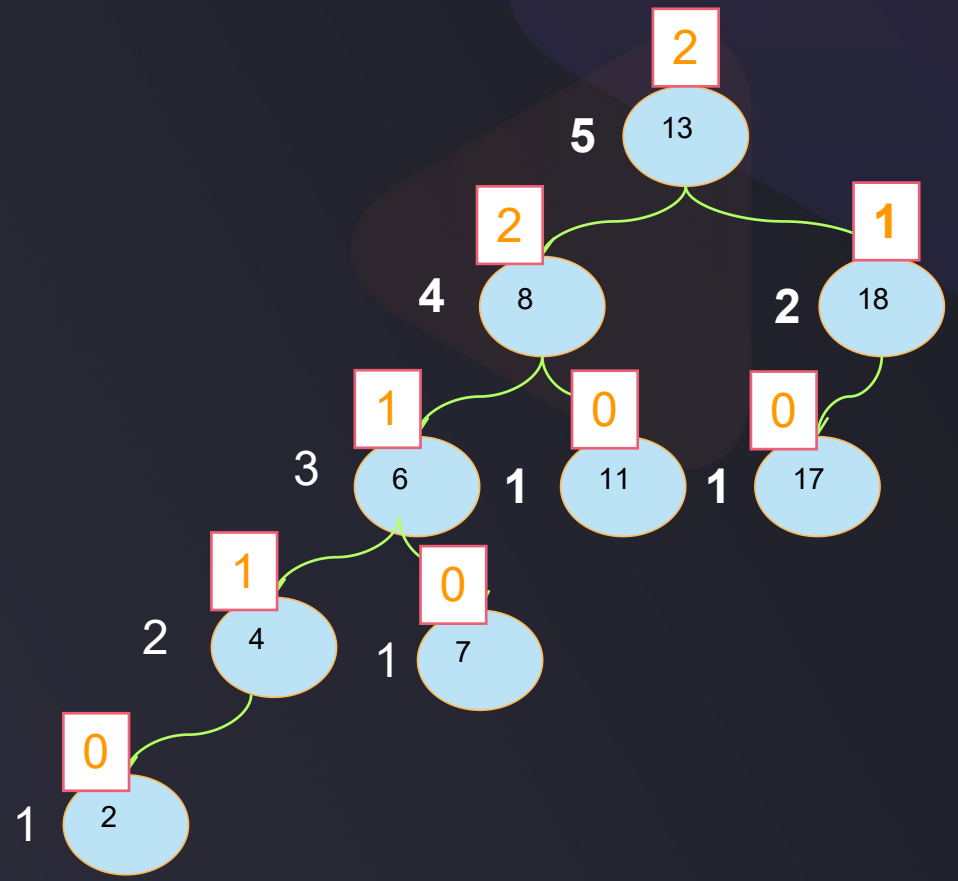
平衡二叉树AVL

2 平衡二叉树(Self-balancing binary search tree)

- 平衡二叉树也叫平衡二叉搜索树、需要满足BST的特征
- 任意一个节点，平衡因子的绝对值不超过1
  - 某节点的高度值=max(左子树高度，右子树高度)+1
  - 每个节点的左子树和右子树的高度差叫做平衡因子
- 平衡二叉树的高度和节点数的关系是O(logn)

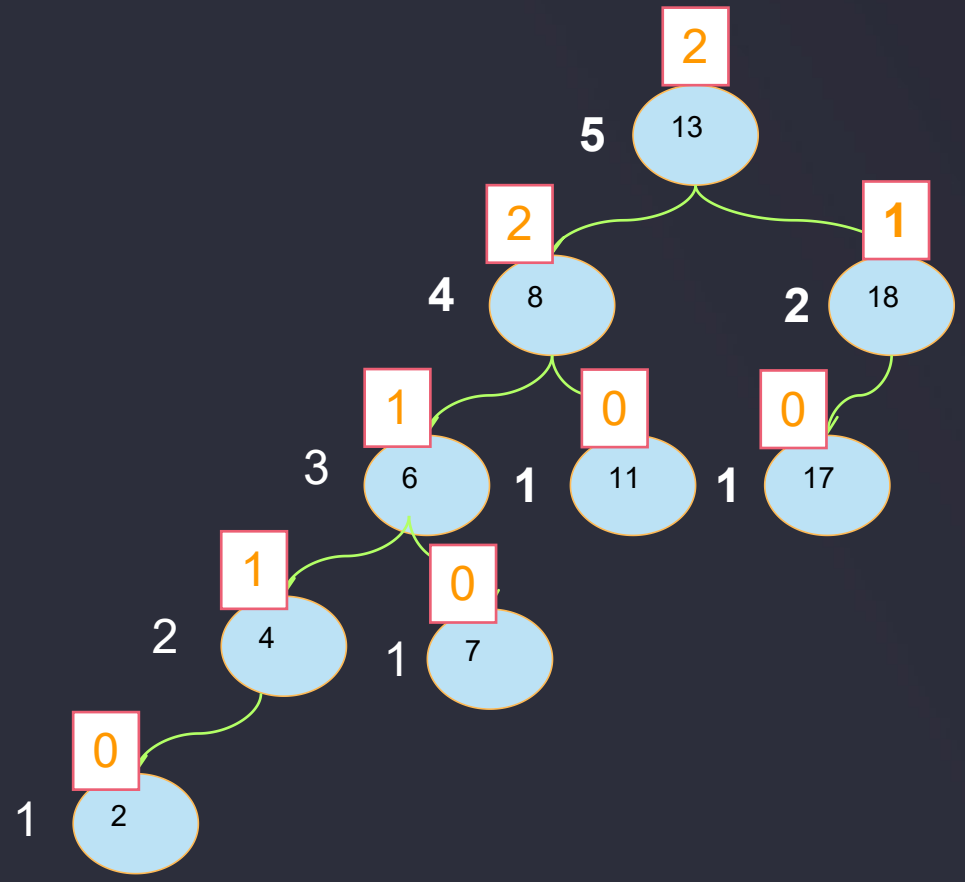






平衡二叉树AVL

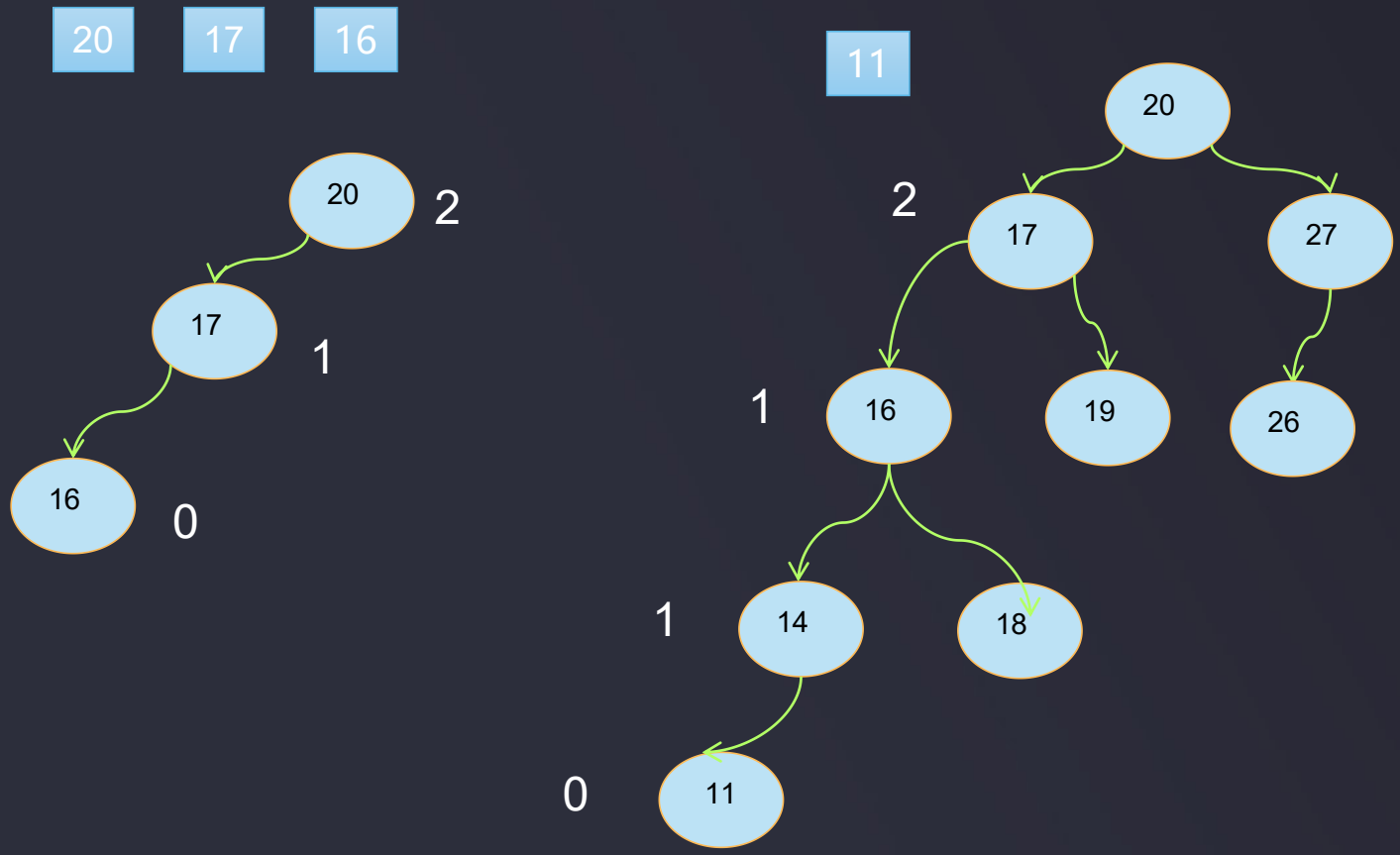
3 何时需要维护平衡?



- BST中插入新节点时从根节点一路寻找正确的位置
  该位置一定是叶子位置
- 由于新增加新的节点，才导致了BST不再平衡
  即平衡因子绝对值>1
- 导致不平衡的节点一定发生在插入路径上的某一处
- 插入是递归插入，因此能够拿到这条完整路径
  计算这条路径的每个节点的平衡因子

平衡二叉树AVL

4 右旋产生条件

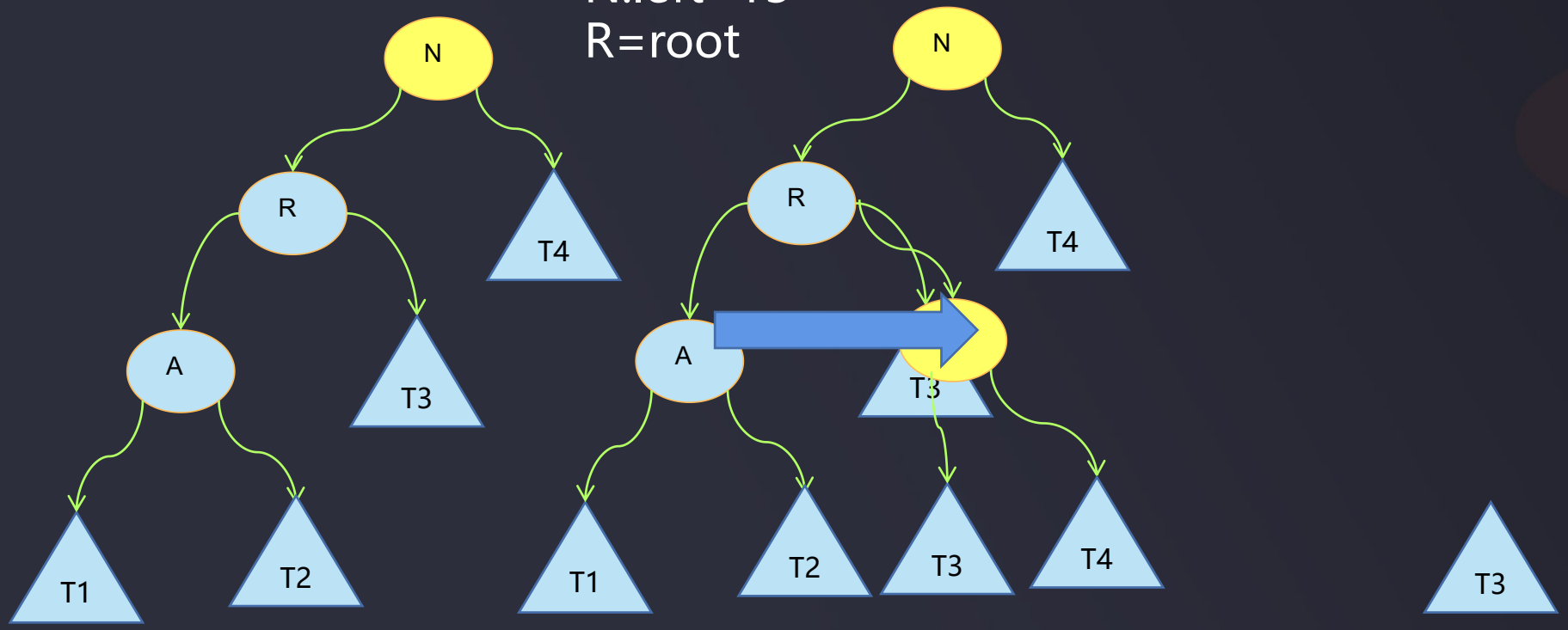


- 新插入节点导致了不平衡
- 不平衡节点在插入的路径上
- 叶子节点在不平衡节点的左侧的左侧
- 可以使用右旋来实现

平衡二叉树AVL

5 右旋过程

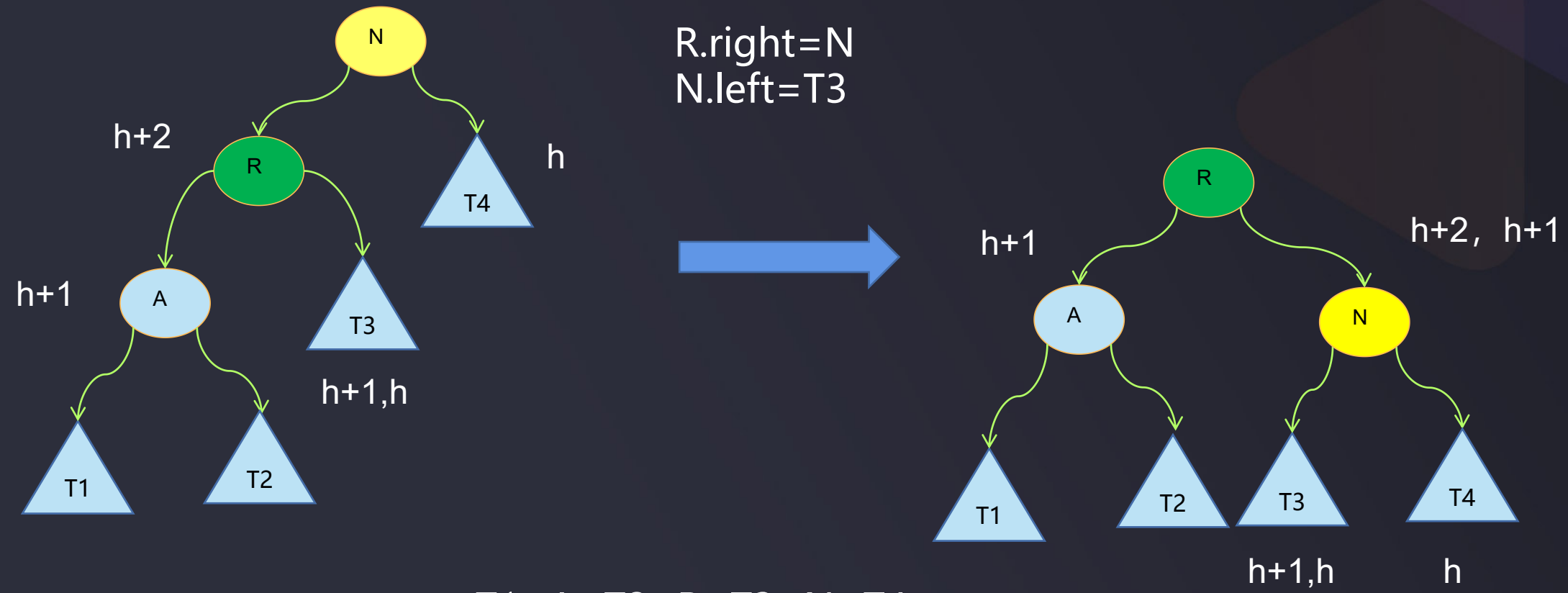
R.right=N
N.left=T3
R=root



T1<A<T2<R<T3<N<T4

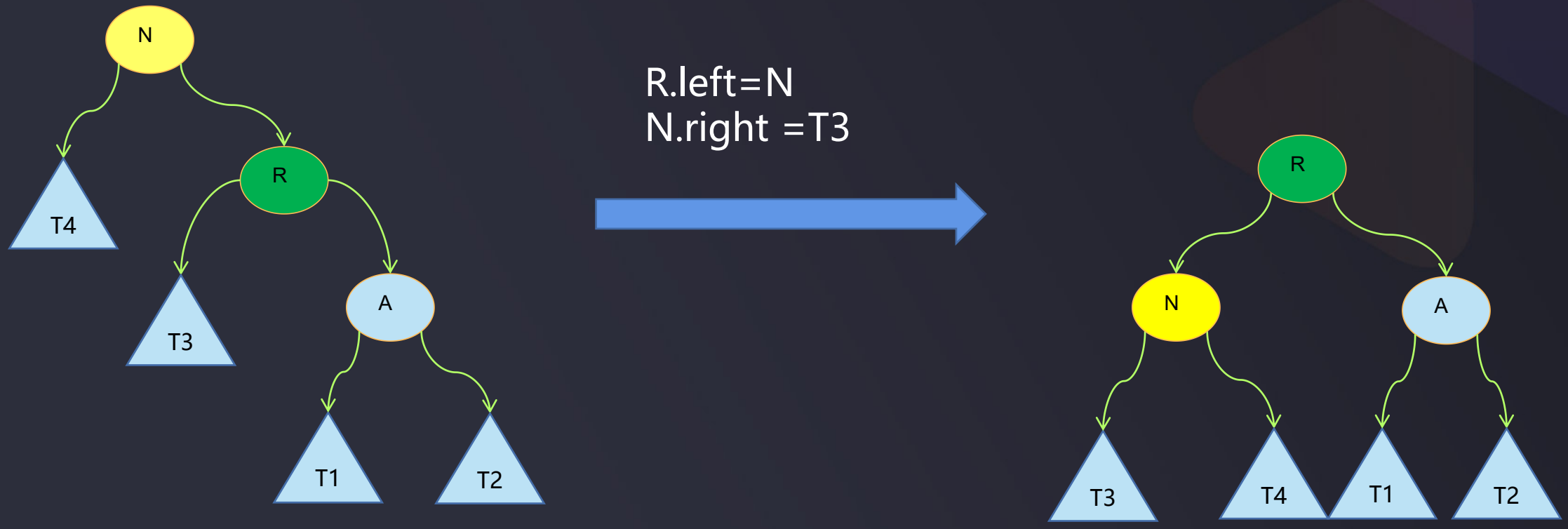
平衡二叉树AVL

6 右旋后保持平衡



平衡二叉树AVL

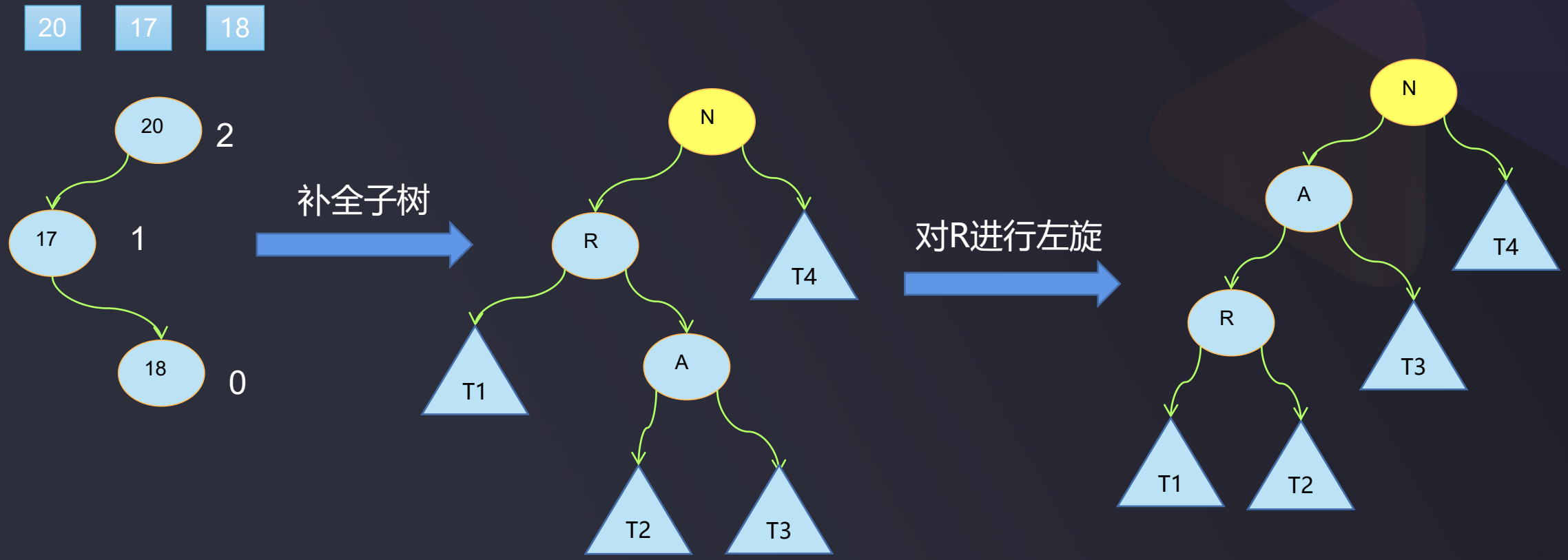
7 左旋



T4<N<T3<R<T1<A<T2

平衡二叉树AVL

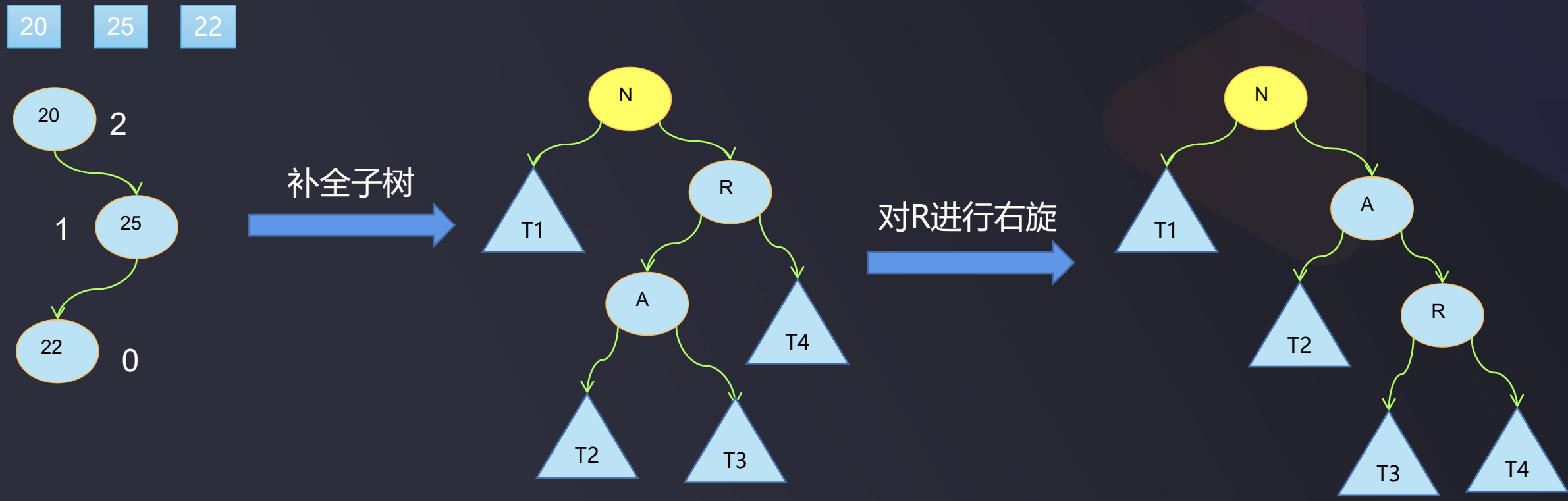
8 LR出现不平衡



转换成了LL的情况

平衡二叉树AVL

9 RL出现不平衡

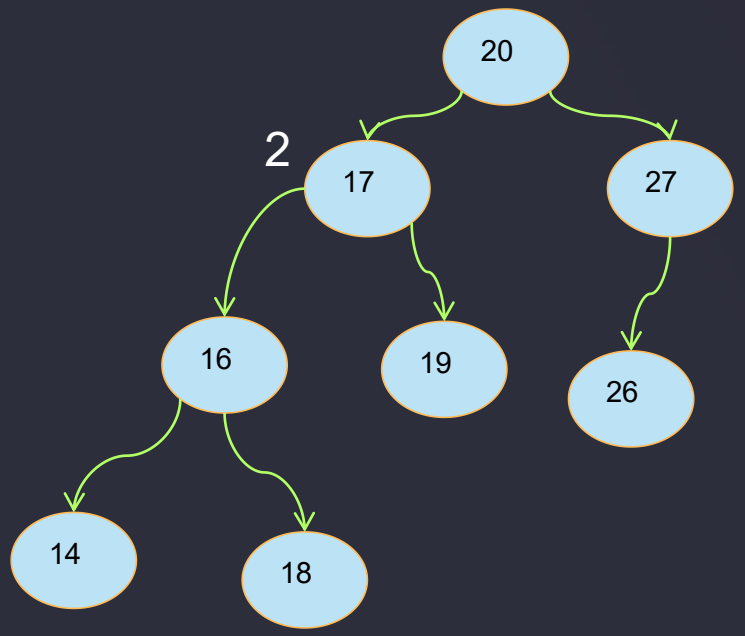


转换成了RR的情况

平衡二叉树AVL

删除节点

删除叶子19



- 由于删除了节点，才导致了BST不再平衡，即平衡因子绝对值>1
- 导致不平衡的节点发生在删除路径上的某一处
- 删除和插入一样都是递归的，因此进行平衡化的过程和插入时的逻辑一致

红黑树RBT

1 红黑树定义

- 前提：研究红黑树时叶子节点指的是最后的空节点(我们原来理解的叶子节点的孩子节点)
- 红黑树的每个节点都是有颜色的，或是红色或者是黑色
- 根节点是黑色的
- 每个叶子节点都是黑色的(红色节点向左倾斜叫做左倾红黑树，Left-Leaning Red-Black Tree)
- 如果一个节点是红色的，那么他的孩子节点都是黑色的
- 从任何一个节点到叶子节点，经过的黑色节点是一样的

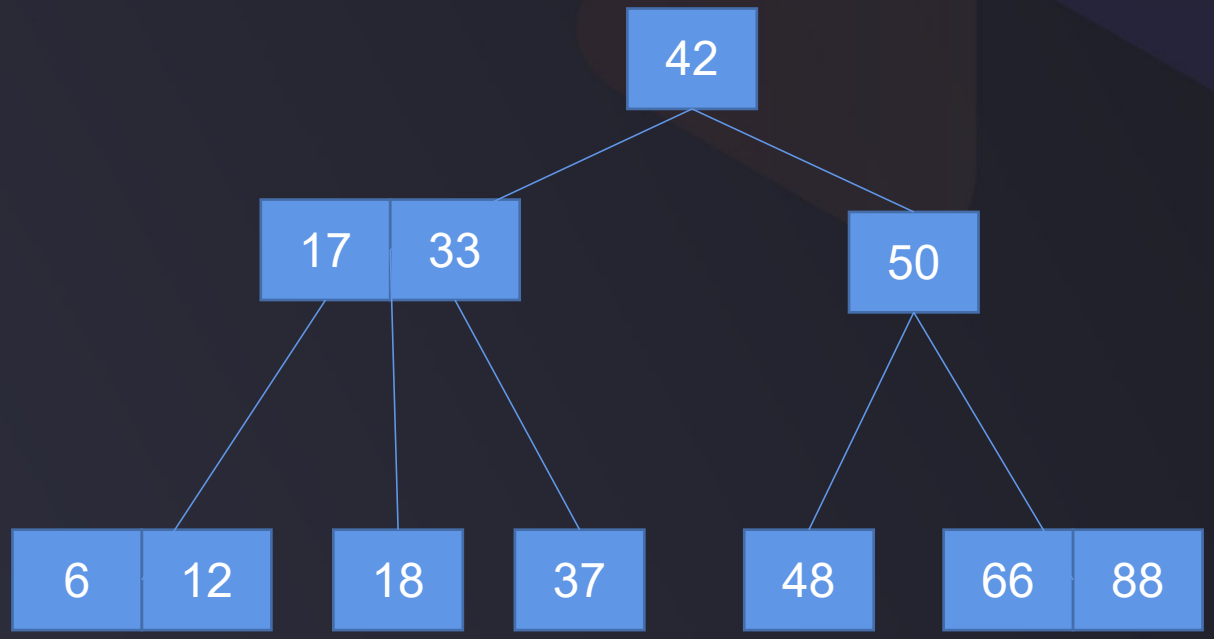
红黑树RBT

2 2-3树的特征

- 每个节点都可以存放一个元素或者两个元素
- 存放一个元素的节点称为2-节点、存放两个元素的节点叫做3-节点
- 每个节点有2个或者3个子节点的树称为2-3树，2-3树满足二叉搜索树的基本性质
- 2-3树是一个绝对平衡的树

2-节点

3-节点



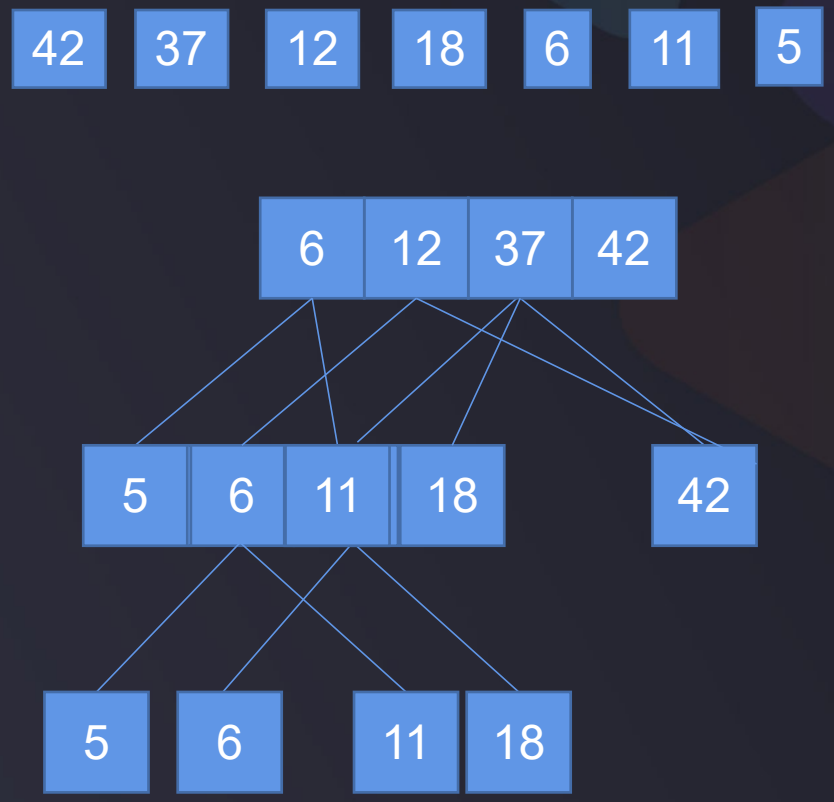
2-3树

红黑树RBT

3 2-3树添加节点维持绝对平衡

2-3树添加节点遵循三个大的前提

- 满足二叉搜索树的特征
- 维持绝对平衡
- 不能往null节点插入数据



红黑树RBT

4 2-3树添加节点

- 如果插入2-节点

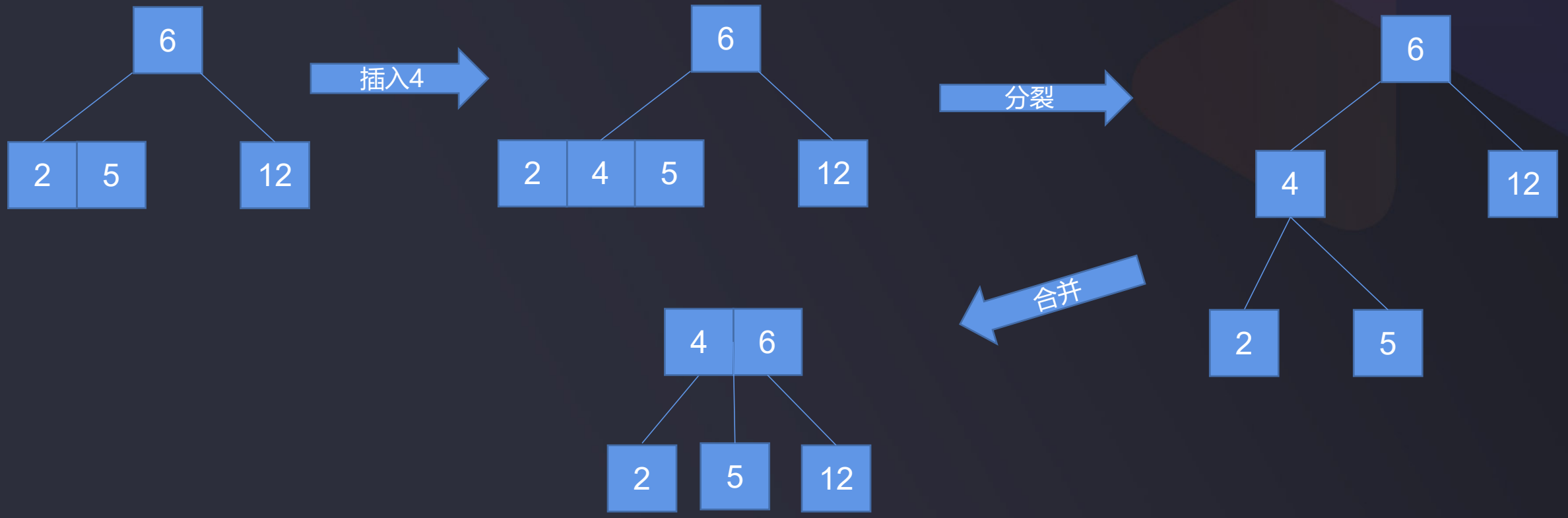
- 如果插入3-节点

- 2-3树中添加一个新元素，或者添加到2-节点或者添加到3-节点
- 添加到2-节点，形成一个3-节点
- 添加到3-节点，暂时形成一个4-节点，然后把4节点进行分裂

红黑树RBT

4 2-3树添加节点

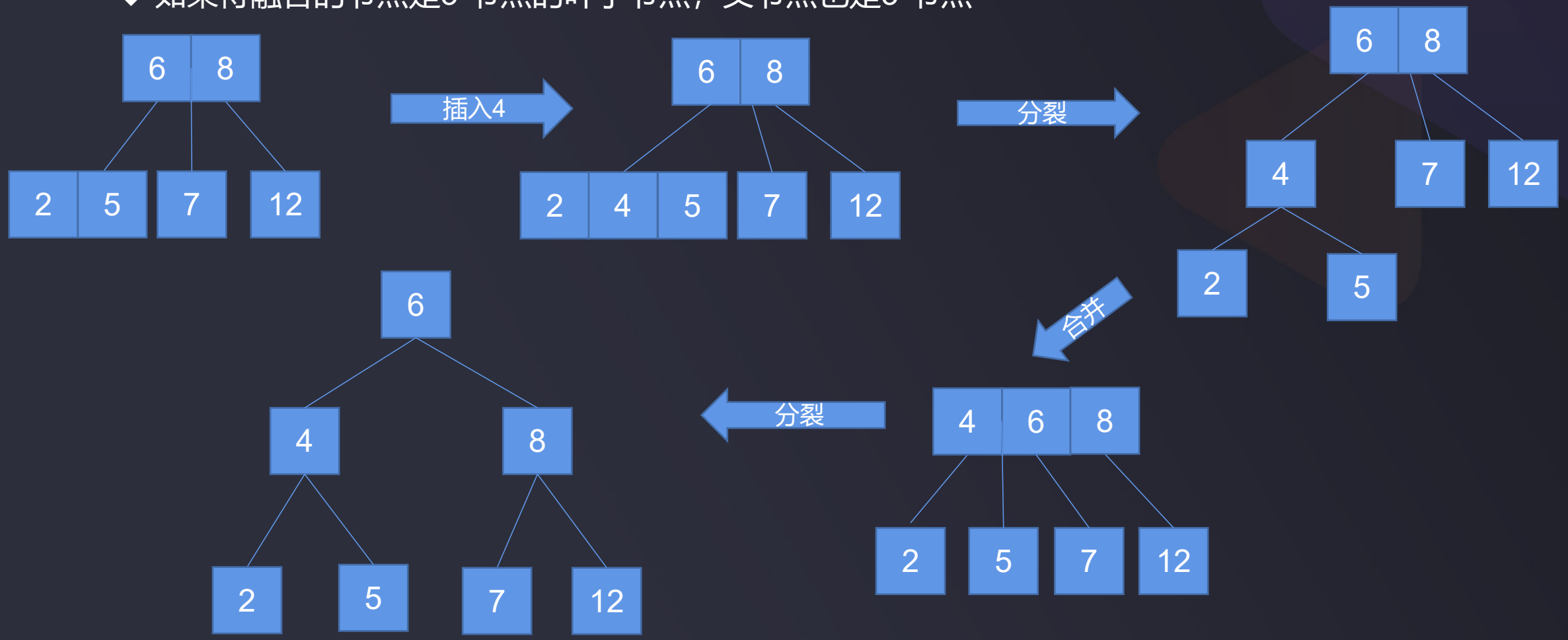
- 如果待融合的节点是3-节点的叶子节点，父节点是2-节点



红黑树RBT

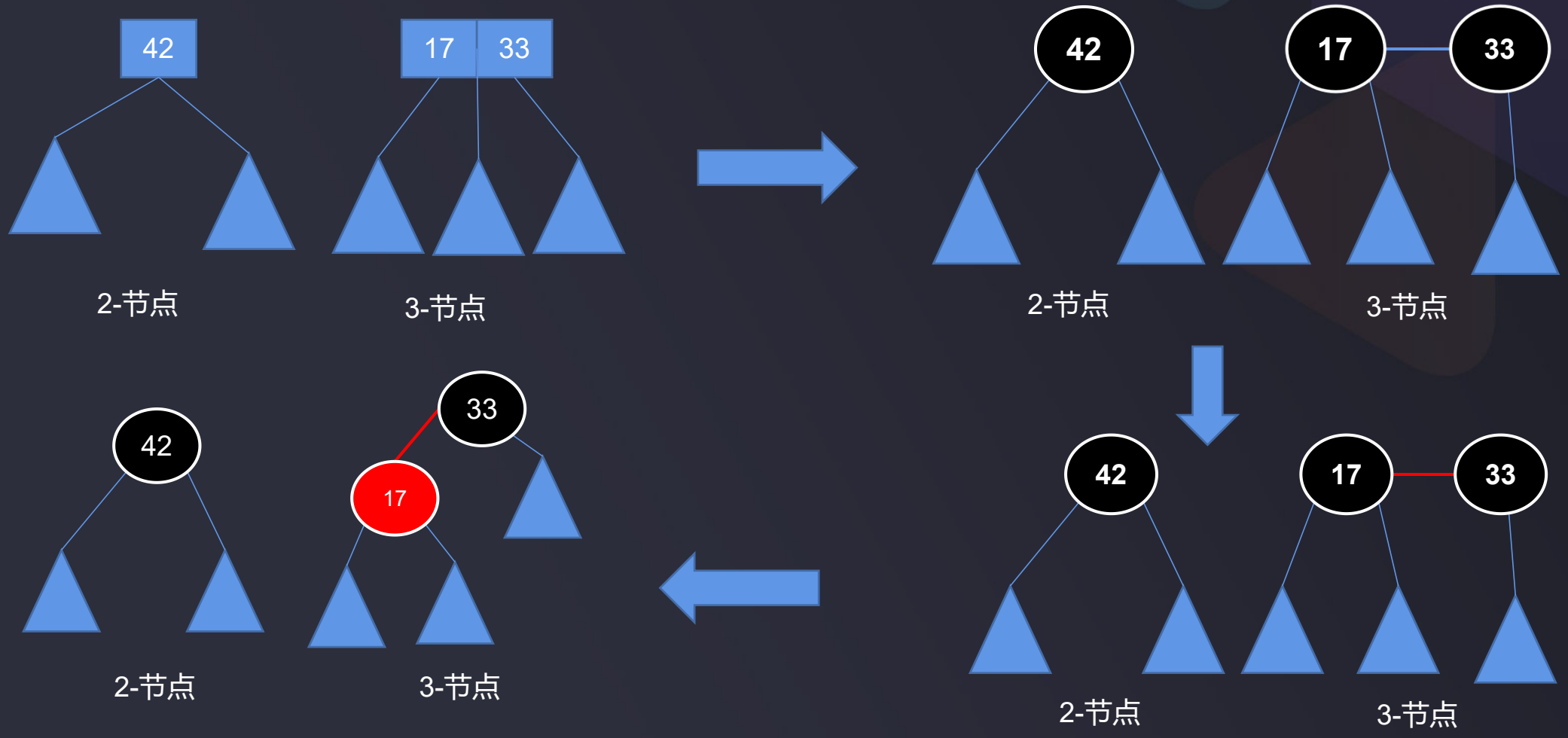
4 2-3树添加节点

◆ 如果待融合的节点是3-节点的叶子节点，父节点也是3-节点



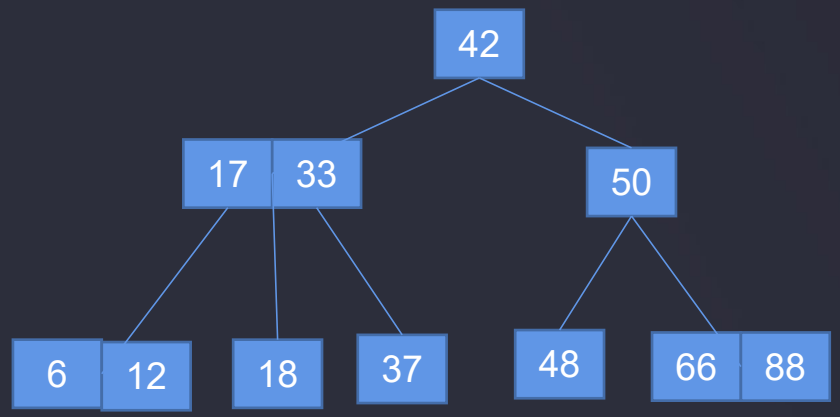
红黑树RBT

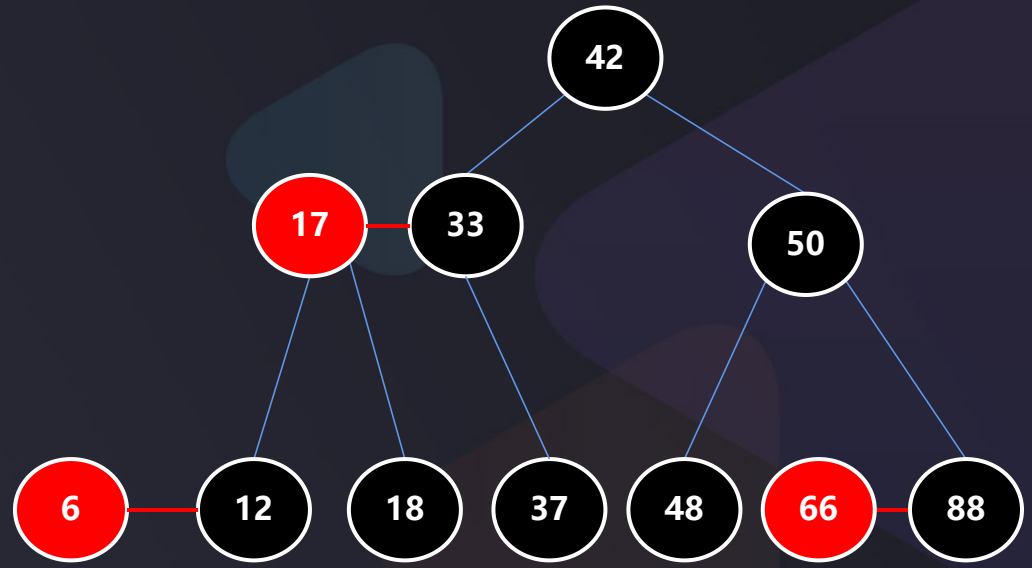
5 2-3树和红黑树等价性

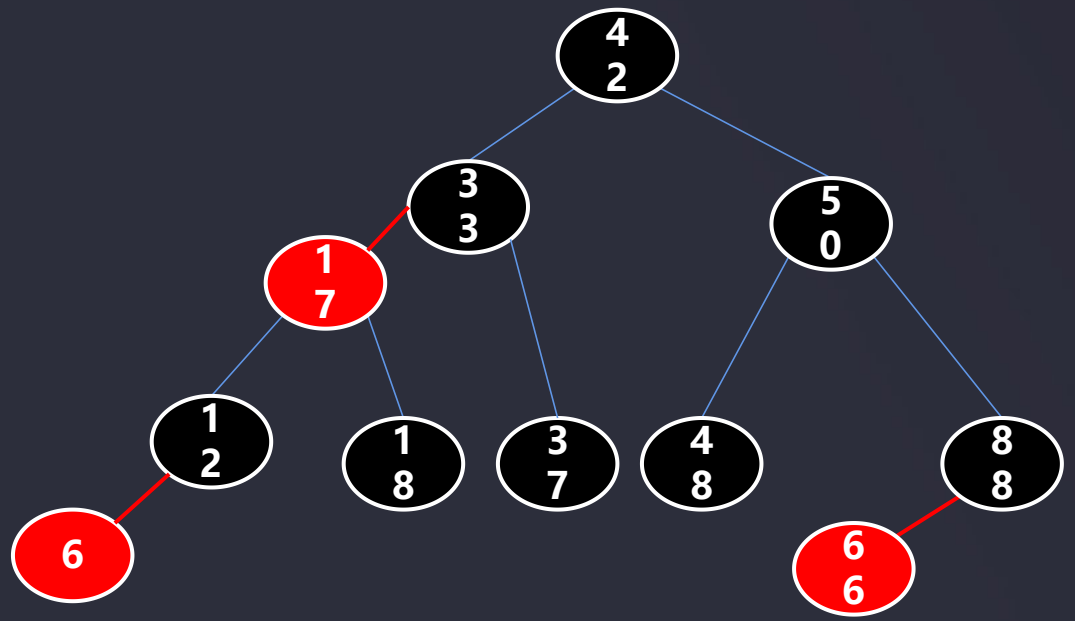


红黑树RBT

5 2-3树和红黑树等价性



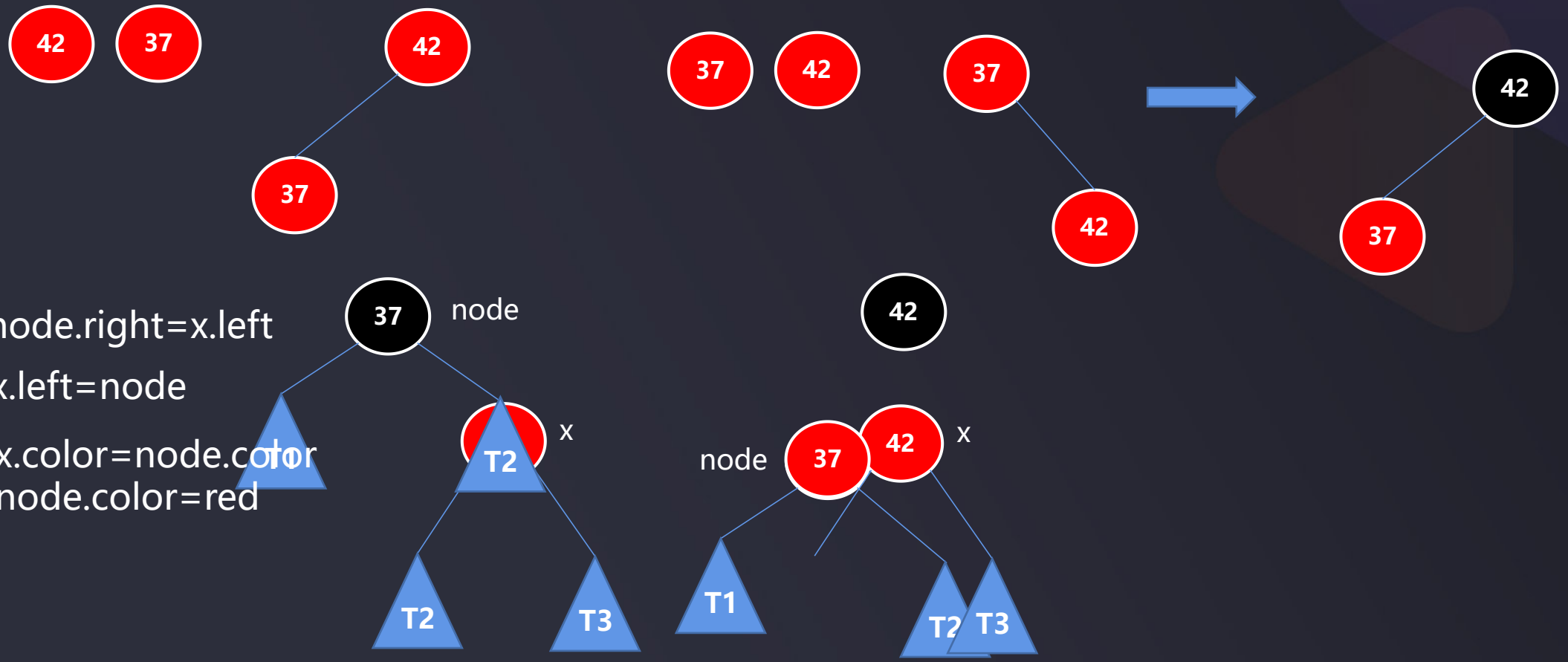




- 红黑树的每个节点或是红色或者是黑色
- 根节点是黑色的，红色节点向左倾斜
- 每个叶子节点都是黑色的
- 如果一个节点是红色的，那么他的孩子节点都是黑色的
- 从任何一个节点到叶子节点，经过的黑色节点是一样的

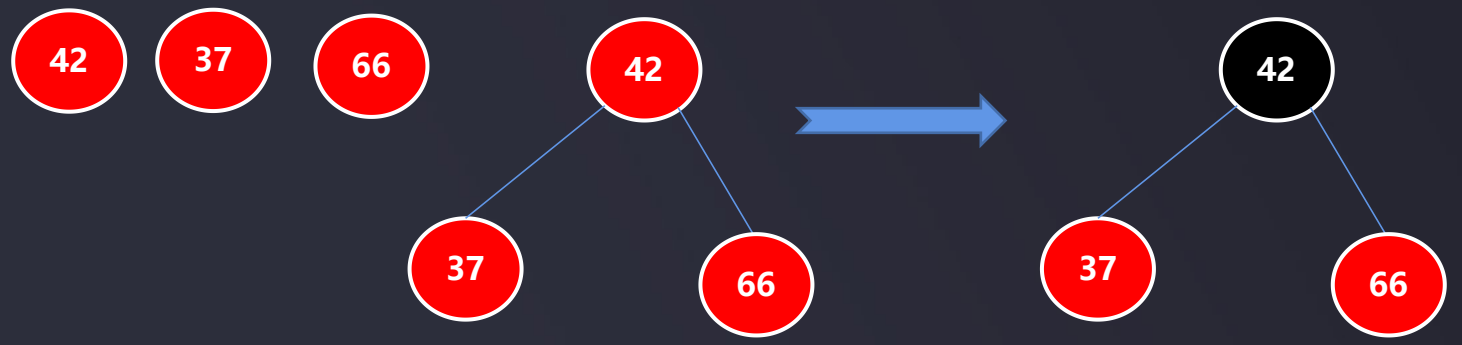
红黑树RBT

6 红黑树添加元素-保持根是黑色的、左旋转

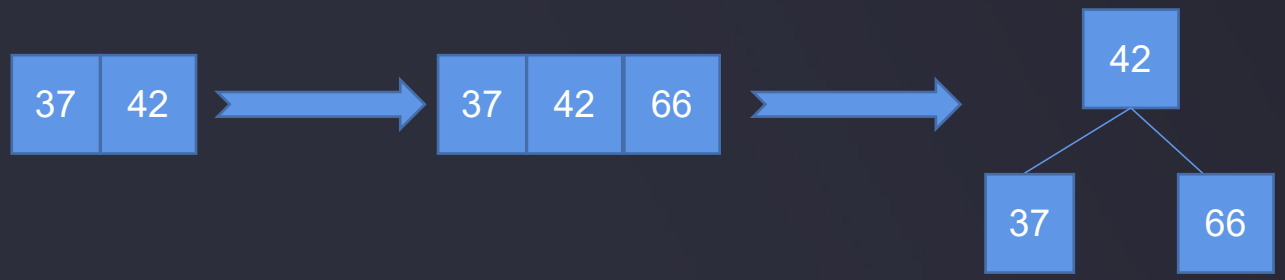


红黑树RBT

6 红黑树添加元素-颜色翻转

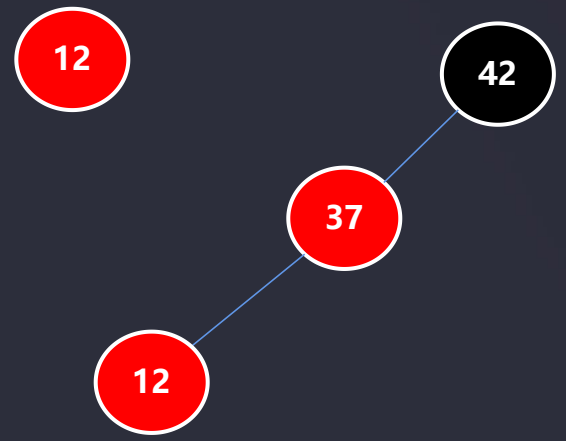


flipColors 颜色翻转

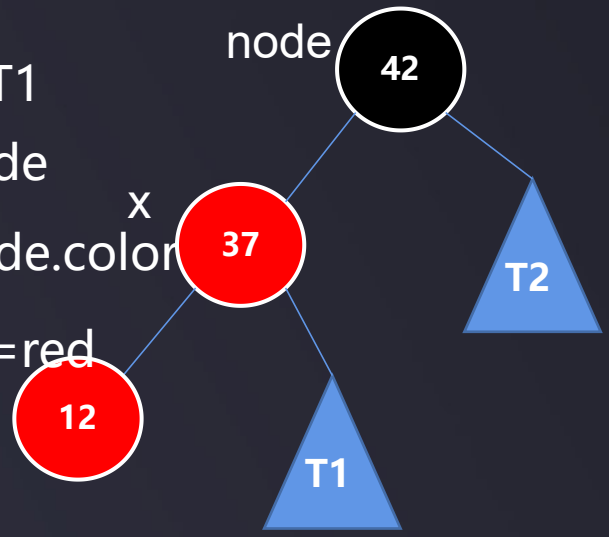


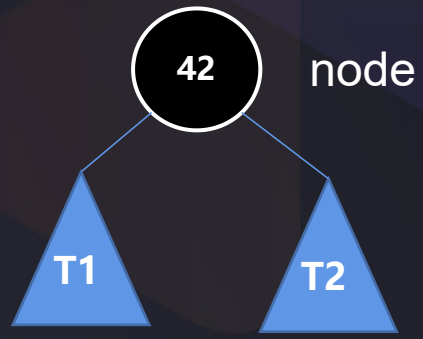
红黑树RBT

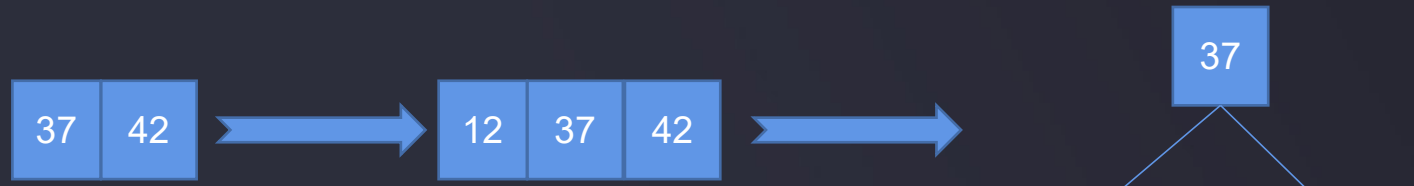
6 红黑树添加元素-右旋



- node.left=T1
- x.right=node
- x.color=node.color
- node.color=red
- flipColors

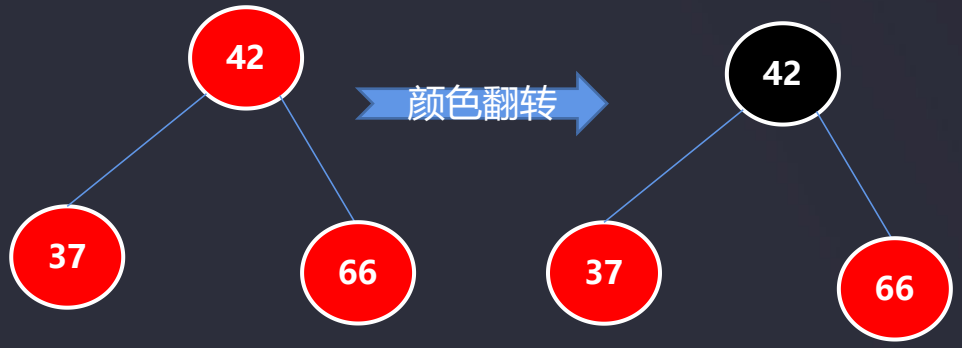


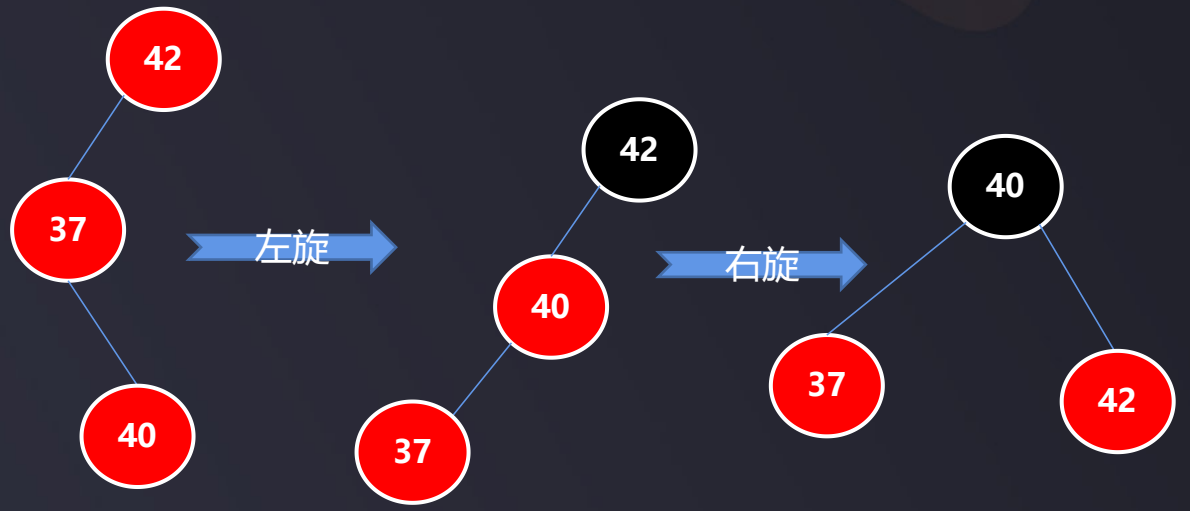




红黑树RBT

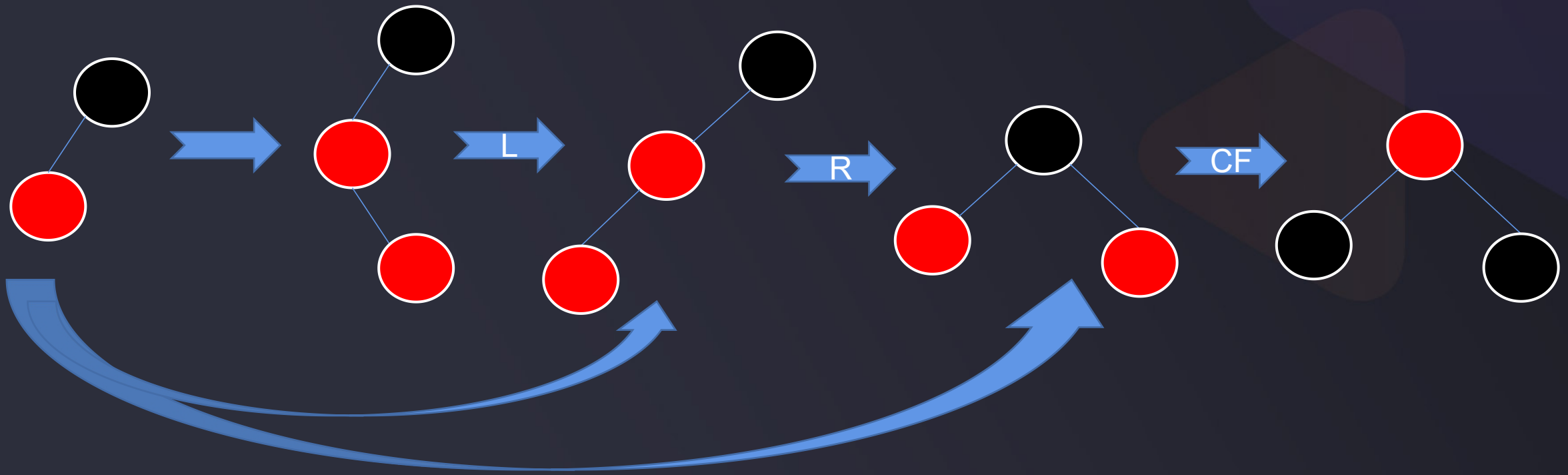
7 红黑树添加元素





红黑树RBT

7 红黑树添加元素



红黑树RBT

8 三种树的应用场景

- 对于完全随机的数据，BST不会出现一侧偏斜的情况，极端情况下会退化成链表或者非常不平衡树
- 对于查询较多的业务场景，AVL是最佳的选择
- RBT的综合性能更优
- HashMap在Java8后引入了红黑树，TreeMap、TreeSet的底层也是红黑树

EDU

CSDN学院 IT实战派

下节课再见，记得关注公众号