奈学教育

Java语言核心---集合

孙玖祥 Samuel

自我介绍

- ◆ 证券行业从业8年
- ➢ 原知名证券信息服务商任项目经理，负责行情系统、资讯系统两条业务线
- ➢ 曾在国内三大商品交易所之一任期货策略交易平台负责人
- ➢ 原知名互金行业的创业公司任架构师。负责日活50万级模拟炒股软件架构设计、主持消息中心平台建设、主导策略交易炒股机器人产品；
- ◆ 连续创业经验
- ➢ 2010年创建外卖平台smartchef
- ➢ 薪航向科技有限公司创始人
- ◆ 讲师行业3年
- ➢ 某培训业上市公司金牌讲师，负责项目实训和就业指导阶段授课。学员月内综合就业率达96+%，毕业学员逾千人
- ➢ 入驻某在线教育平台，目前Java学科类热门讲师排名前五

数组的特点

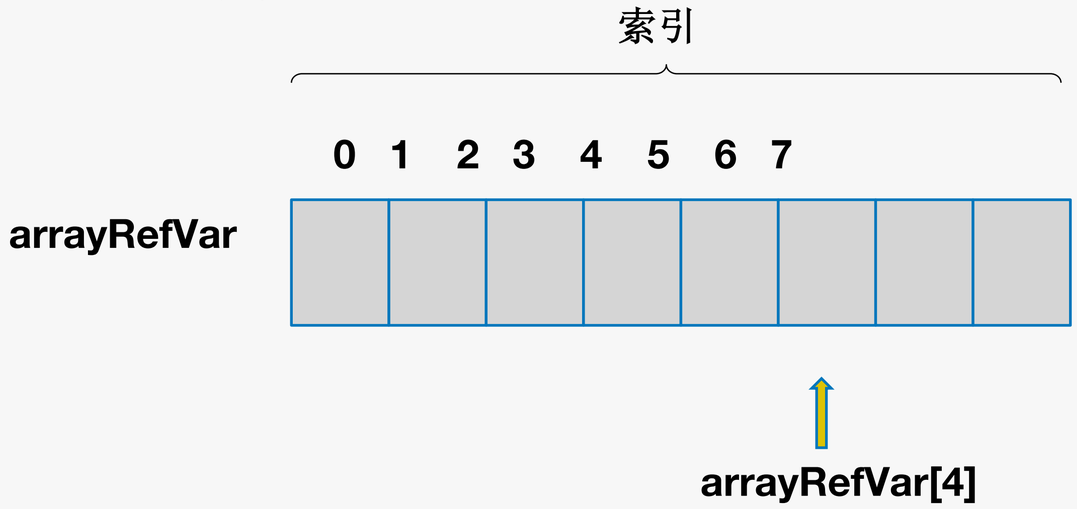
◆Java 语言中提供的数组是用来存储固定大小的同类型元素

◆Java中可以使用两种方式来声明数组

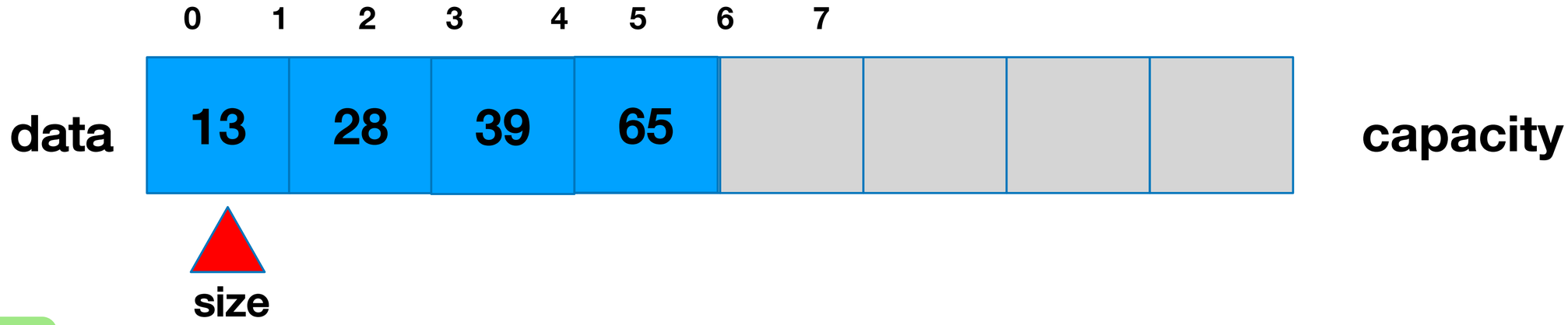
- dataType[] arrayRefVar
- dataType arrayRefVar[]

◆Java中数组的创建方式同样有两种

- arrayRefVar = new dataType[arraySize]
- dataType[] arrayRefVar = {value0, value1, ..., valuek}



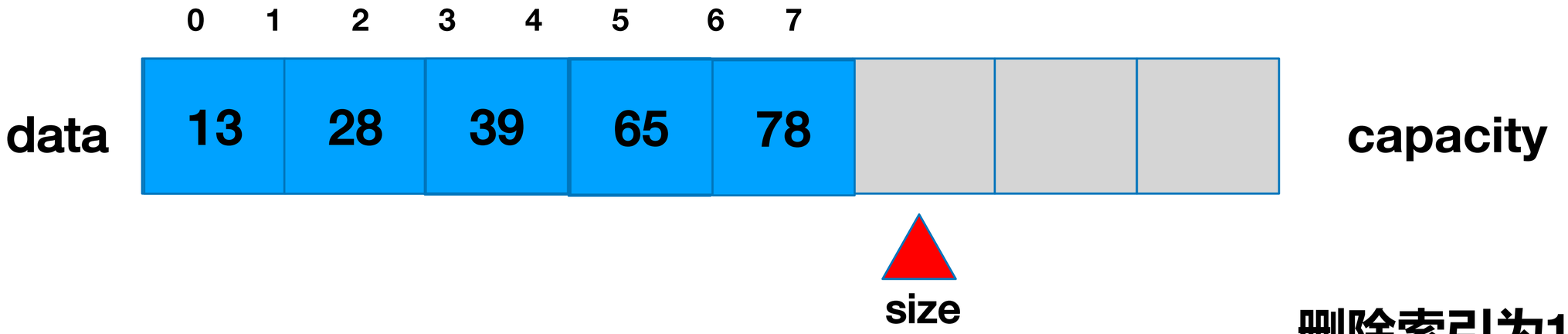
数组添加元素



数组查询和修改元素

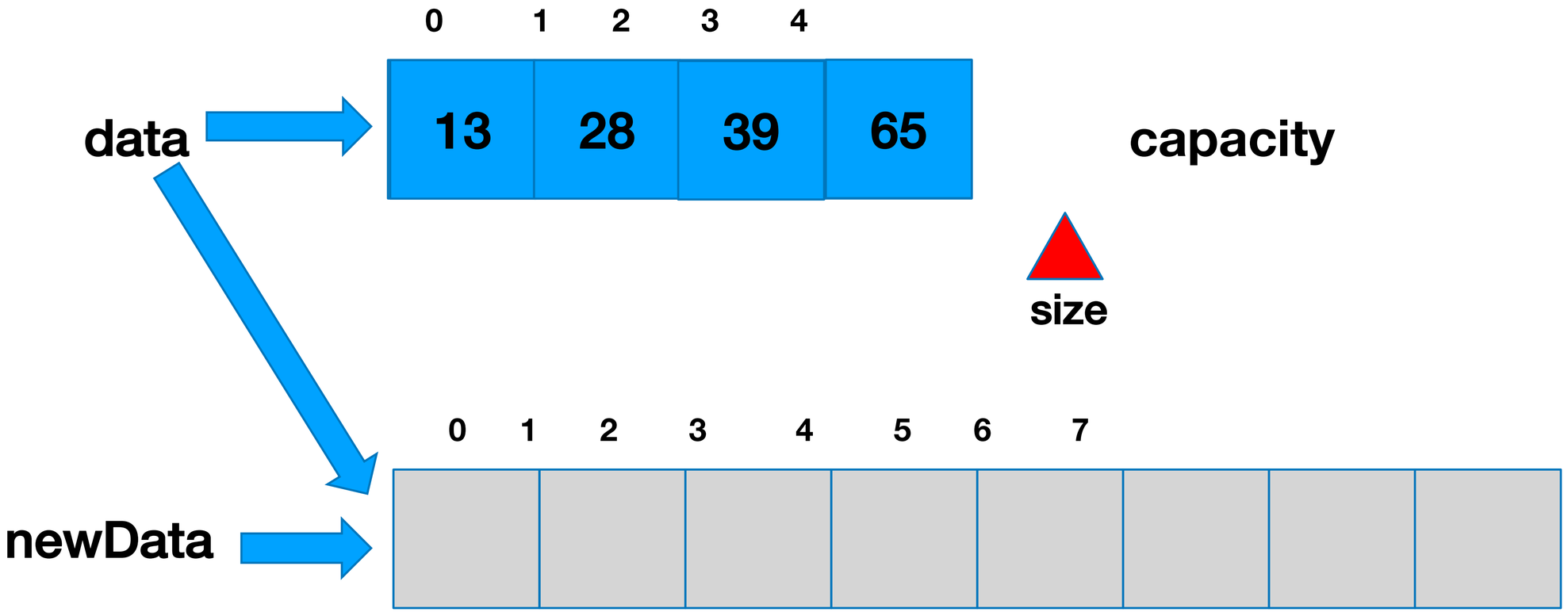
- 给定索引获得到元素
- 设置某个索引位置上的元素为e
- contains和find

数组删除元素



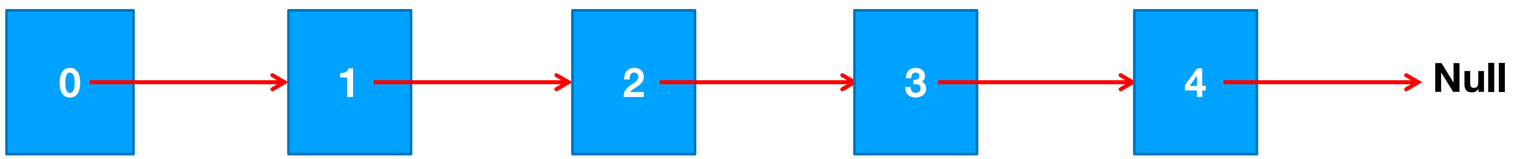
删除索引为1的元素

动态数组

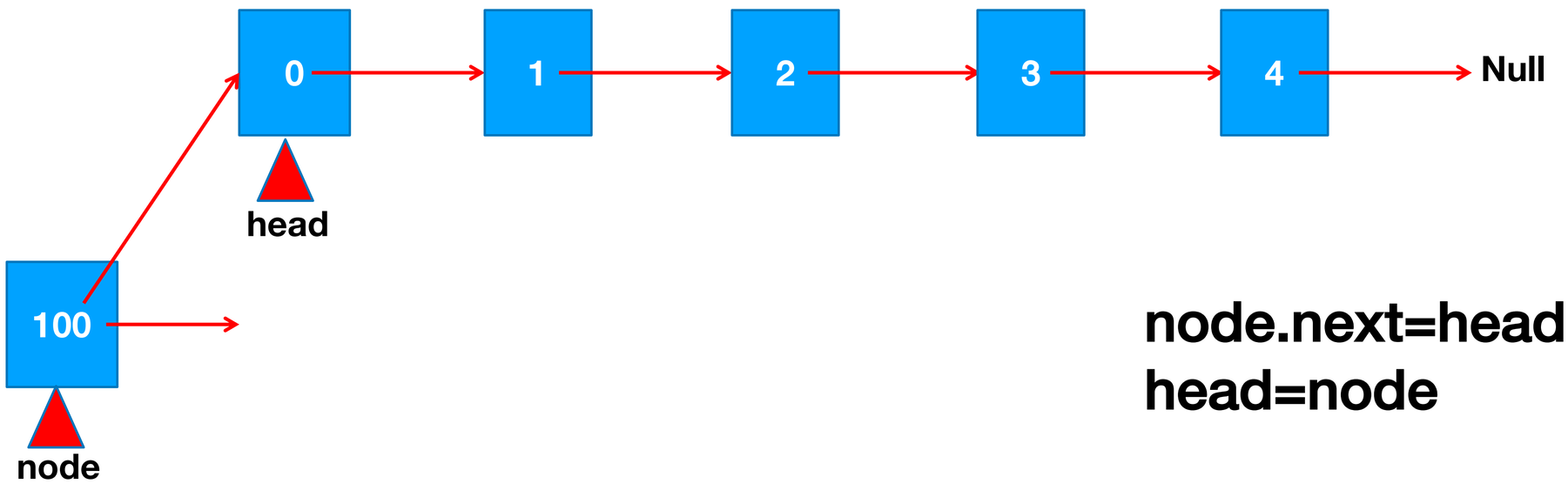


链表的特征

- 链表（Linked list）是一种真正的动态的数据结构
- 链表是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针
- 使用链表结构可以克服数组需要预先知道数据大小的缺点，但增加了结点的指针域，空间开销比较大
- 链表允许插入和移除链表上任意位置上的节点，但是不允许随机存取
- 链表有很多种不同的类型：单向链表，双向链表以及循环链表

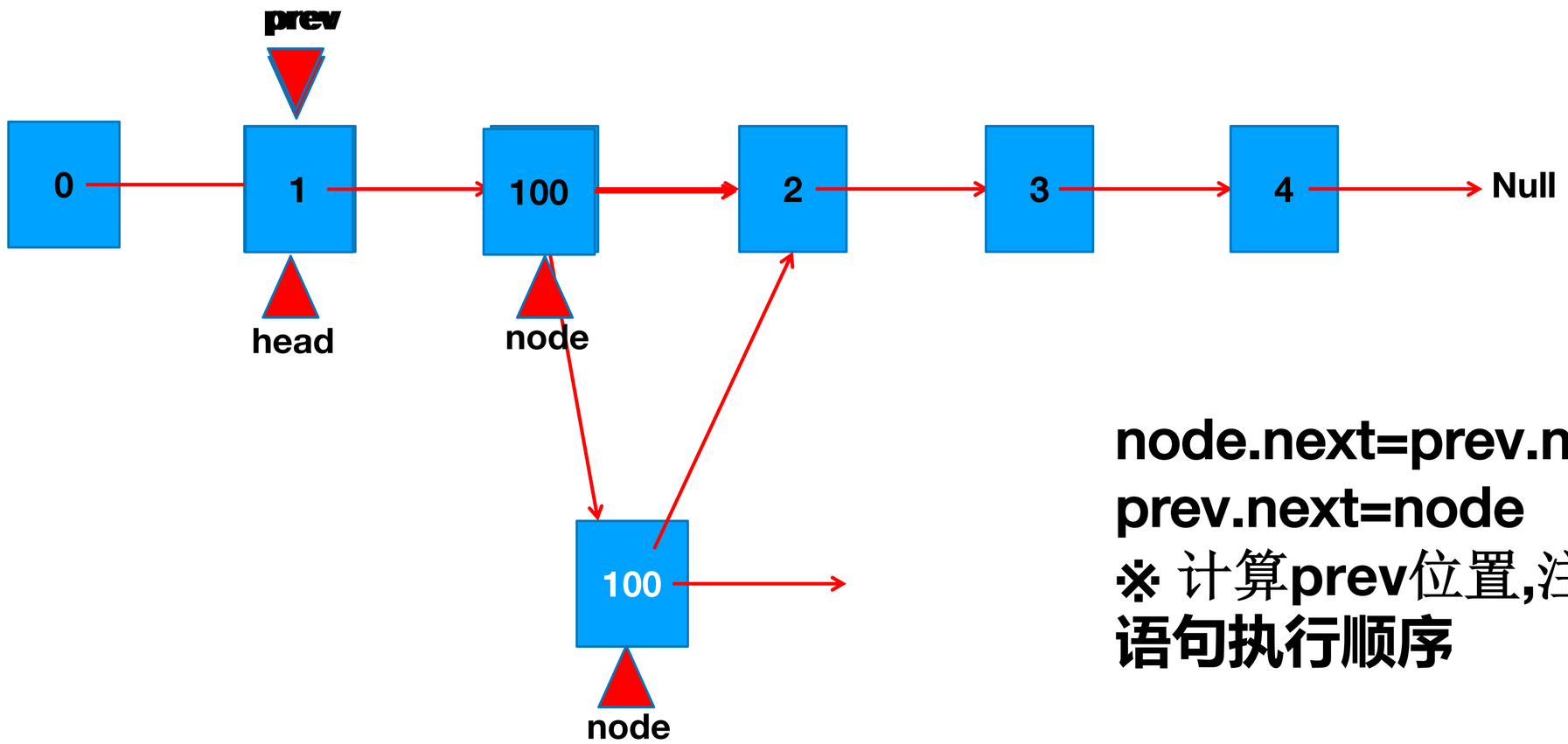


单链表的头添加元素



单链表的中间添加元素

在索引2的位置插入100



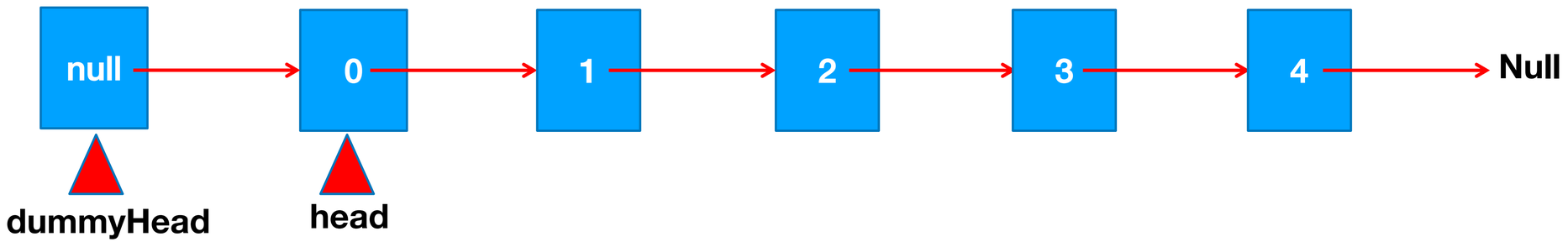
node.next=prev.next

prev.next=node

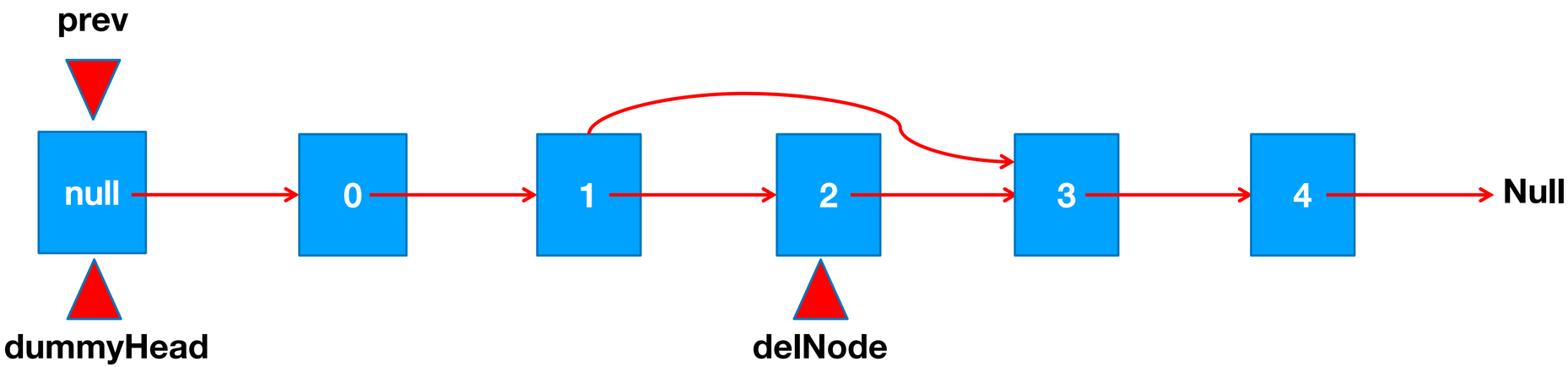
※ 计算prev位置,注意索引为0的情况

语句执行顺序

单链表的虚拟头结点

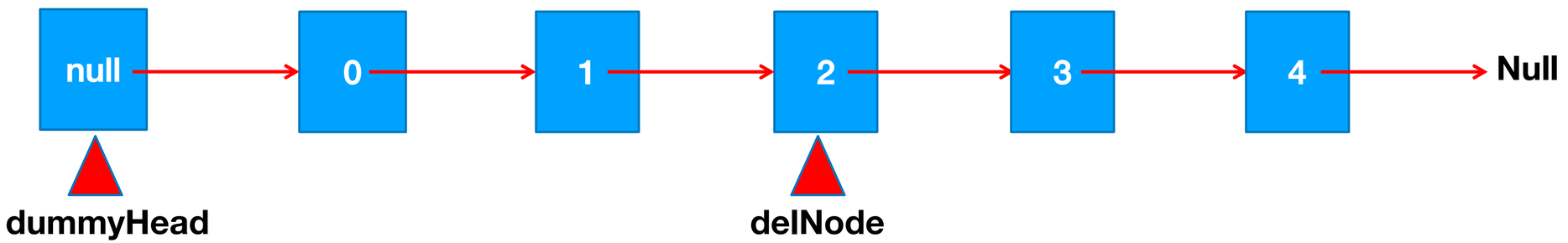


单链表的删除节点

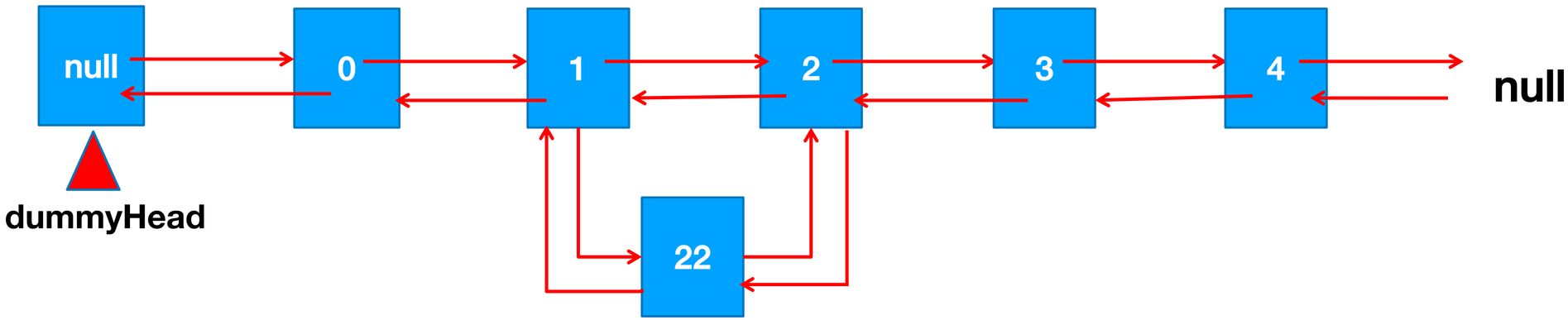


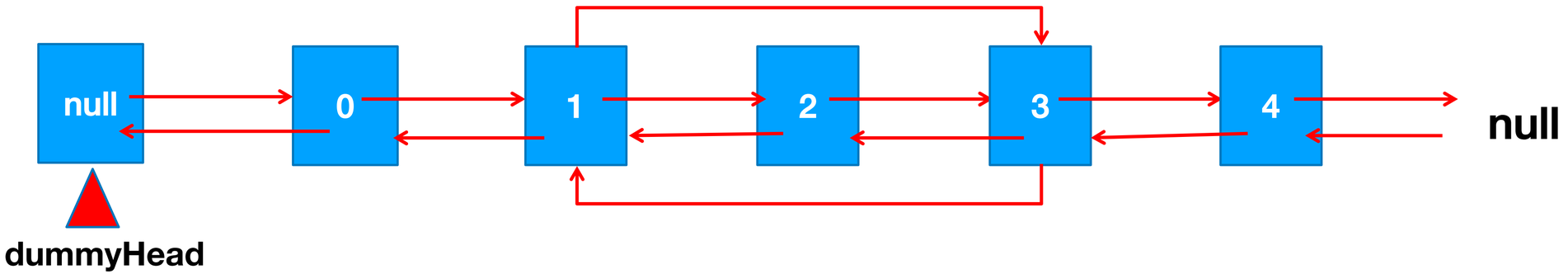
删除索引为2的节点

prev.next = delNode.next
delNode = null
※ 不能使用delNode=delNode.next

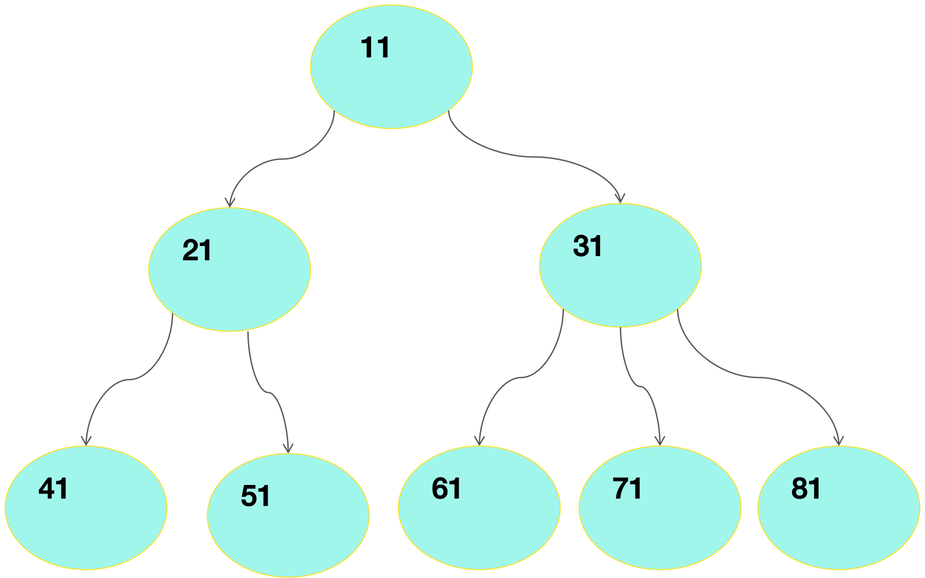


双链表





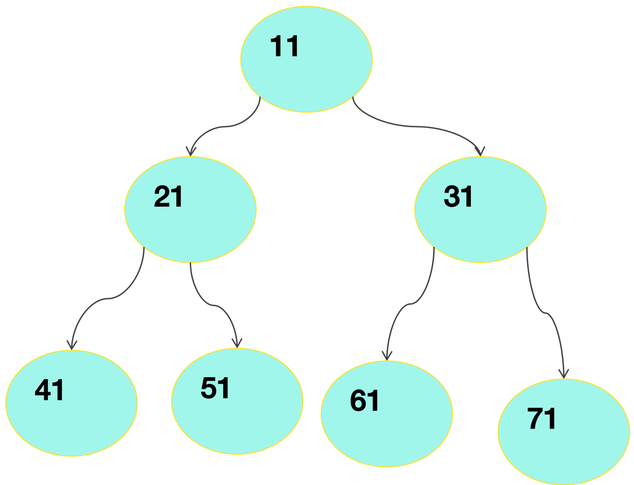
树的常用术语

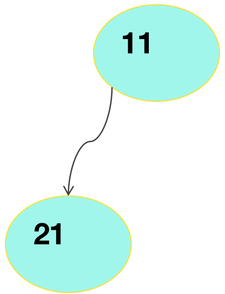


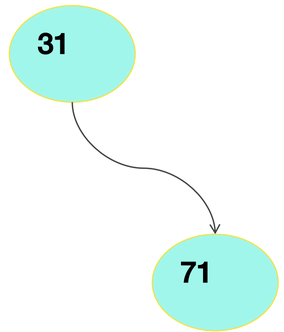
I. 节点
II. 根节点
III. 父节点
IV. 子节点
V. 叶子节点
VI. 节点的权
VII. 度
VIII. 路径
IX. 层
X. 子树
XI. 树的高度
XII. 森林

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

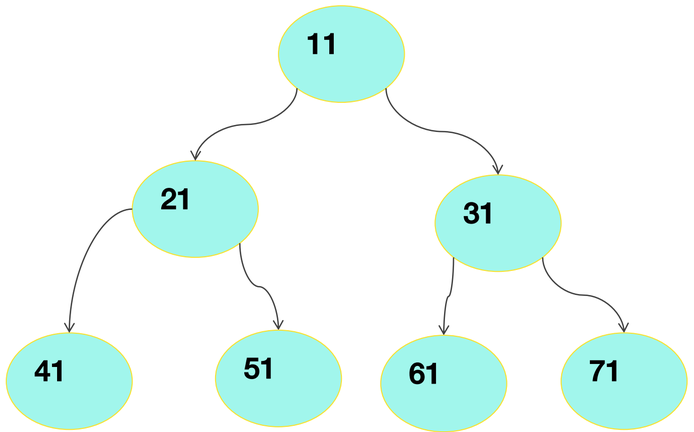






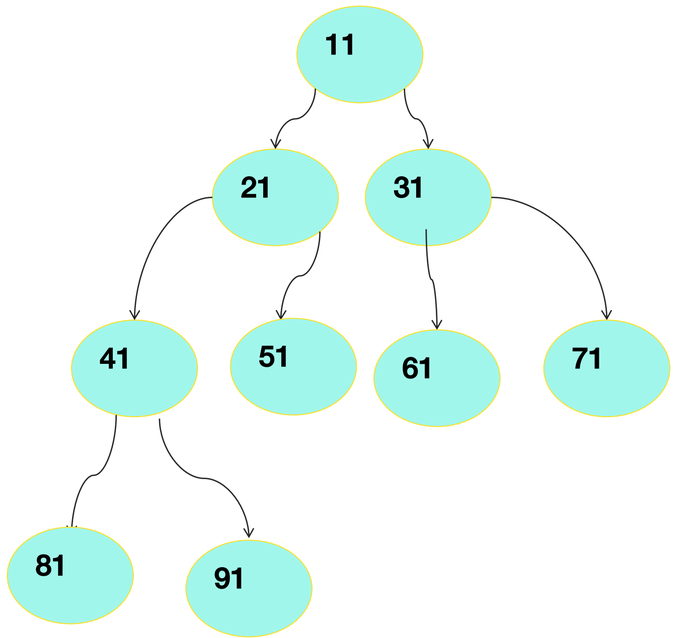
满二叉树

- 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



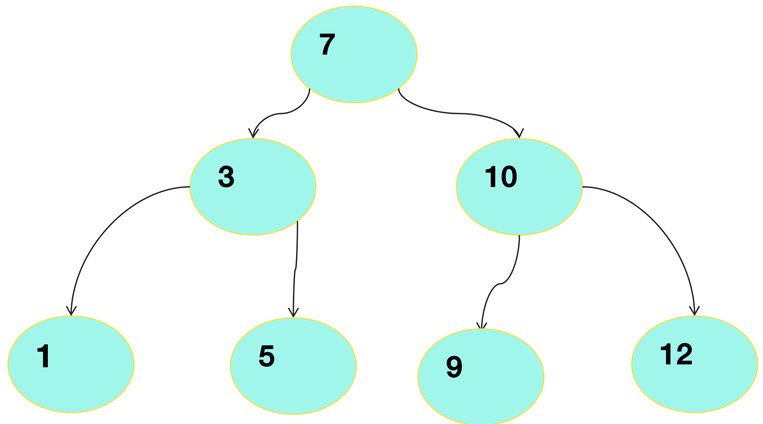
完全二叉树

- 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



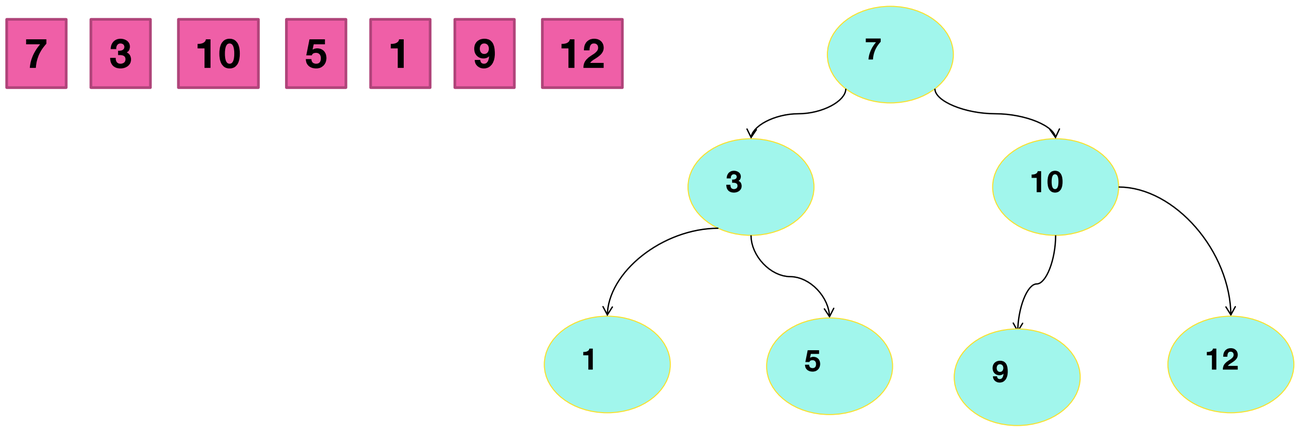
二叉搜索树概念

- 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



二叉搜索树的创建和查找

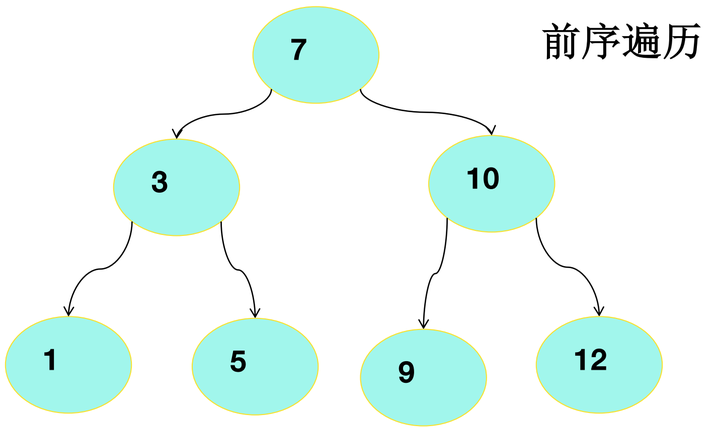
◆ 数据 (7, 3, 10, 5, 1, 9，12)，对应的二叉排序树创建过程如下



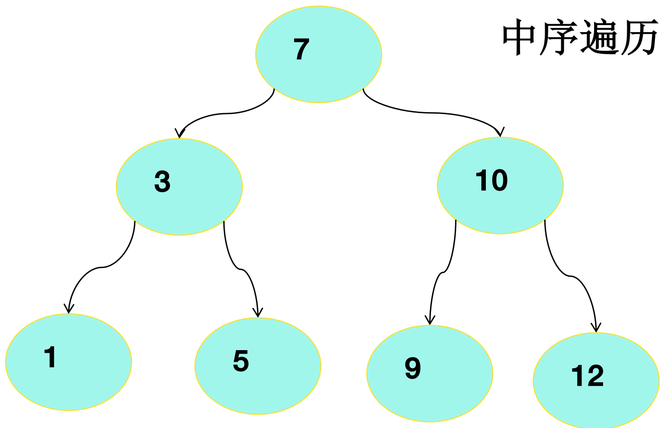
◆ 查找某一具体元素，可以使用的递归的方式

二叉搜索树的深度优先遍历

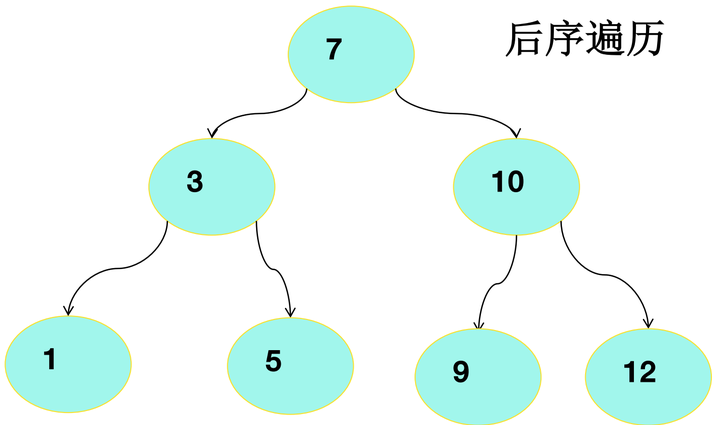
- 二叉树的深度遍历分为：前序遍历、中序遍历、后序遍历
- 前序遍历: 先输出父节点，再遍历左子树和右子树
- 中序遍历: 先遍历左子树，再输出父节点，再遍历右子树，中序遍历的结果是有序的
- 后序遍历: 先遍历左子树，再遍历右子树，最后输出父节点



7 3 1 5 10 9 12



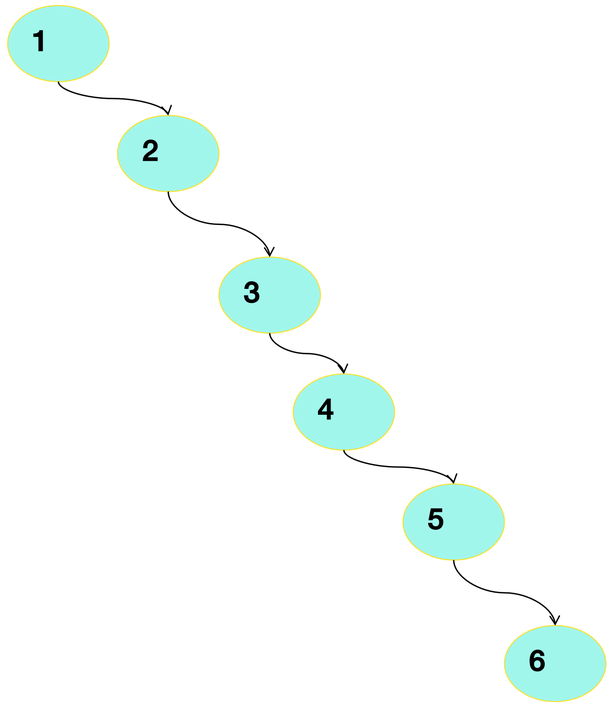
1 3 5 7 9 10 12



1 5 3 9 12 10 7

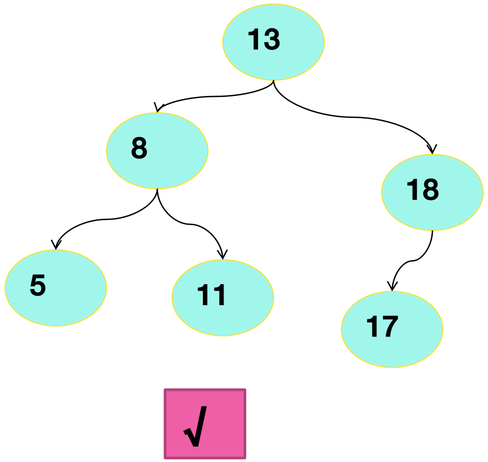
二叉搜索树的问题

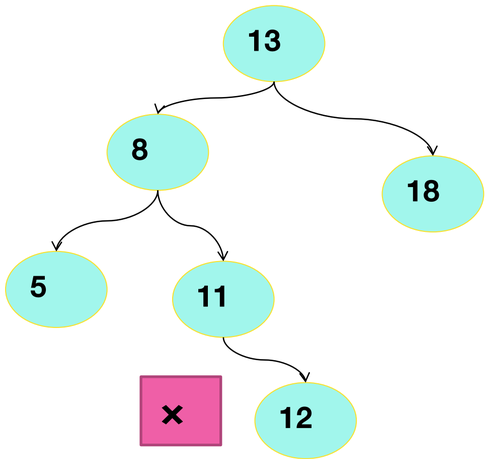
- 数据 (1,2,3,4,5,6) ,创建一棵BST
- 左子树全部为空，从形式上看，更像一个单链表
- 插入速度没有影响
- 查询速度明显降低
- 解决方案-平衡二叉树

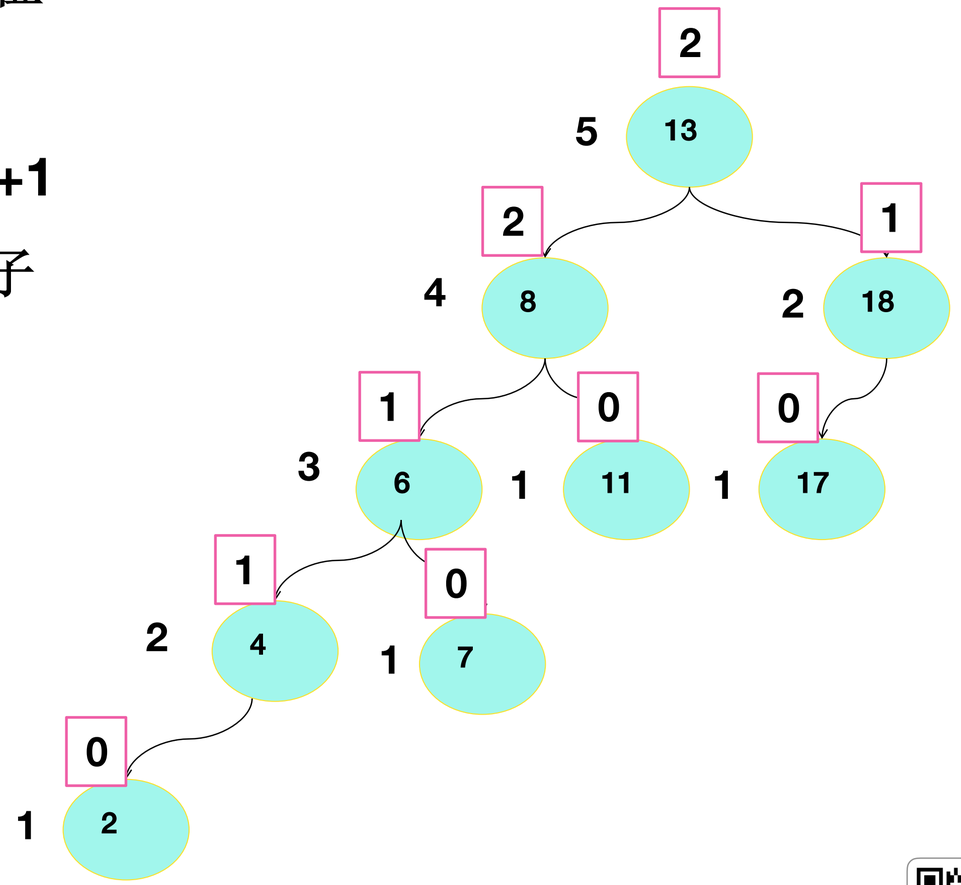


平衡二叉树(Self-balancing binary search tree)

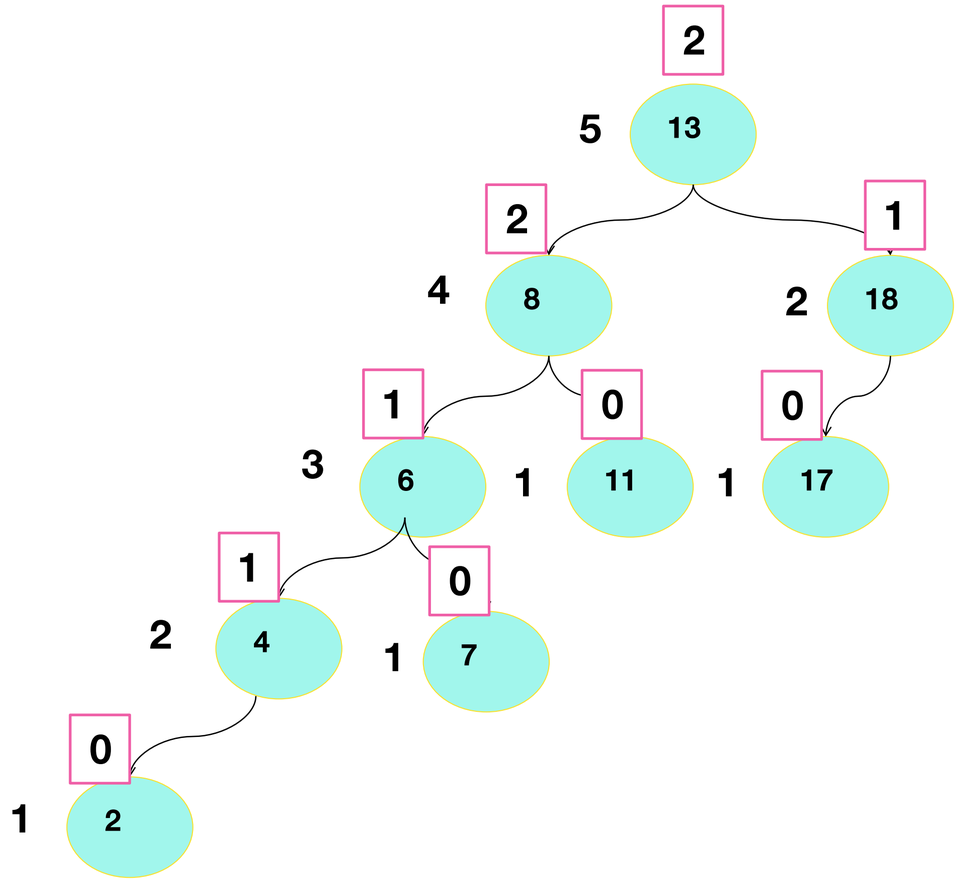
- 平衡二叉树也叫平衡二叉搜索树、需要满足BST的特征
- 任意一个节点，平衡因子的绝对值不超过1
  - 某节点的高度值=max(左子树高度，右子树高度)+1
  - 每个节点的左子树和右子树的高度差叫做平衡因子
- 平衡二叉树的高度和节点数的关系是O(logn)





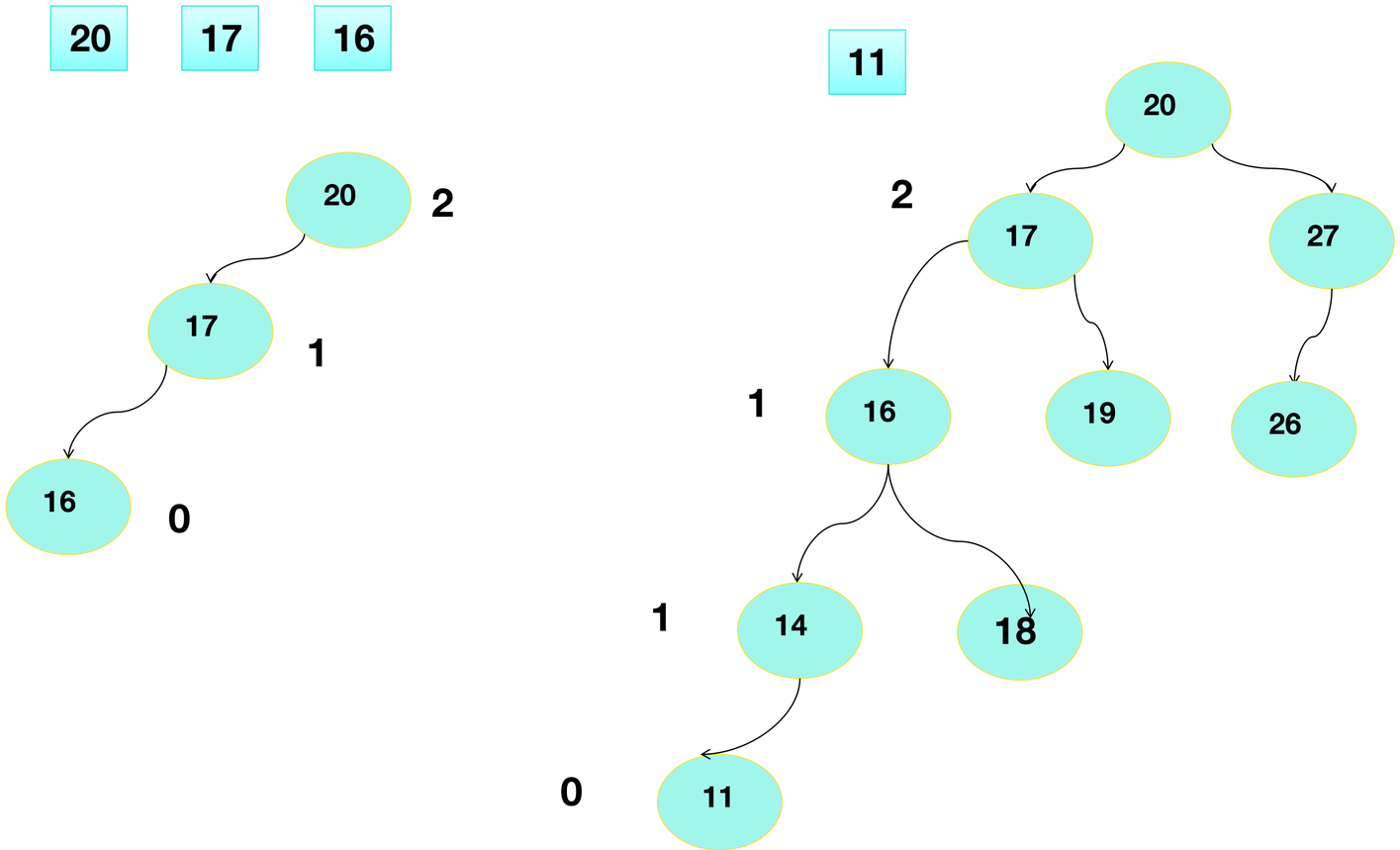


何时需要维护平衡？



- 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

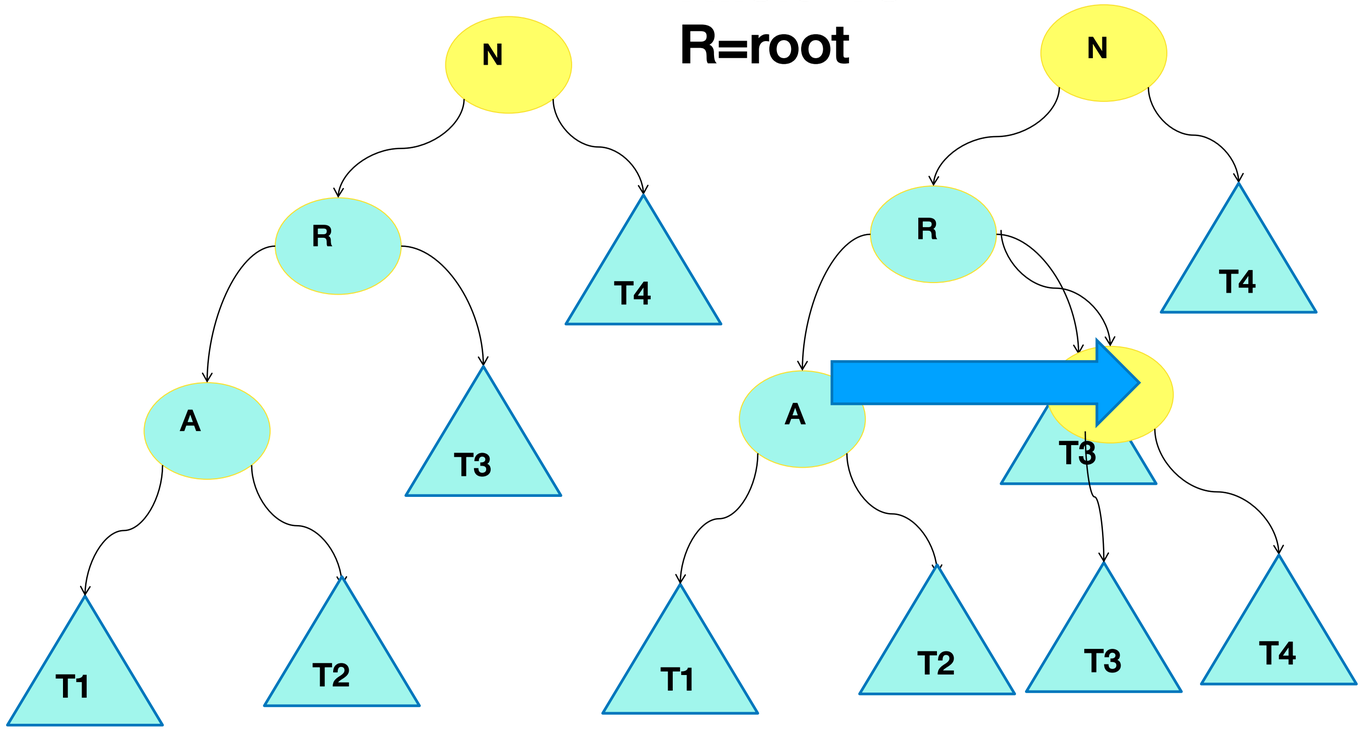
右旋产生条件



- 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

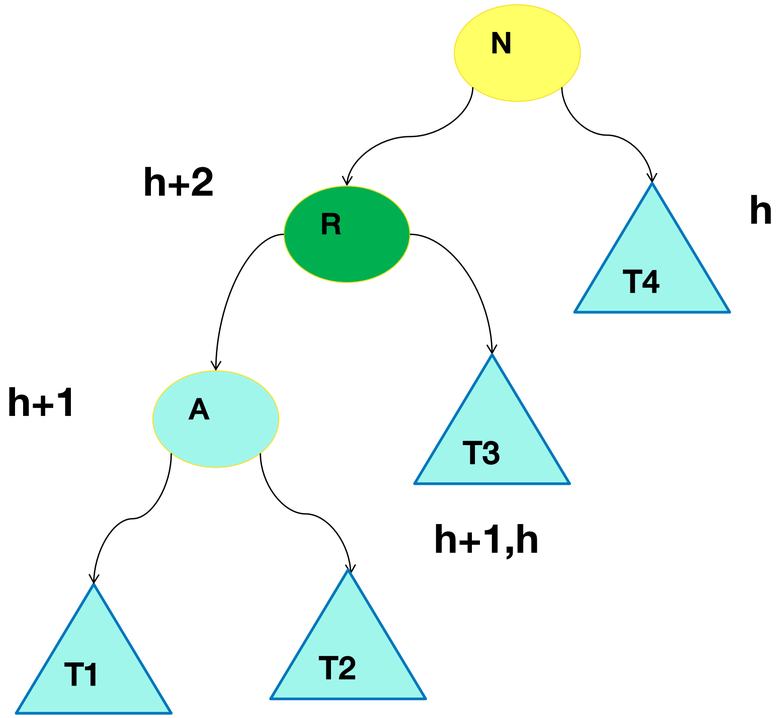
右旋过程

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

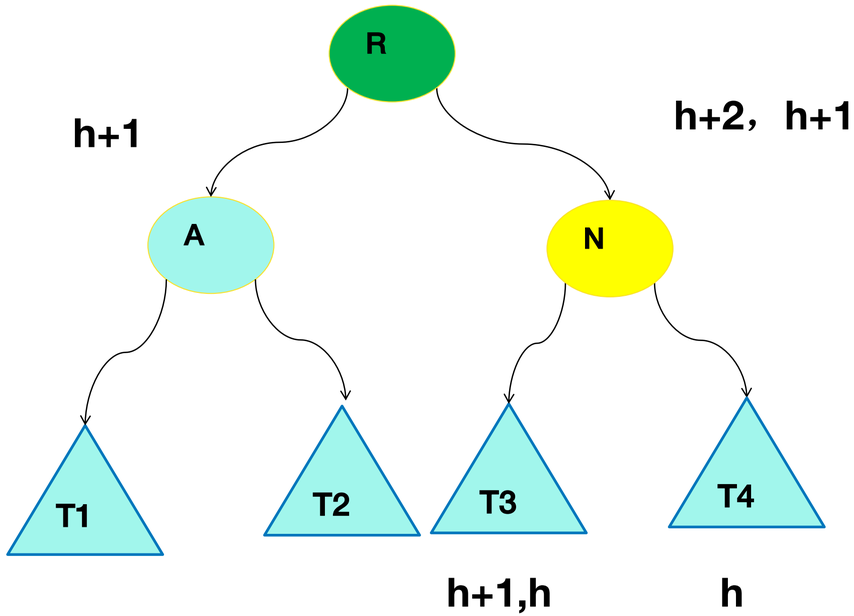


T1<A<T2<R<T3<N<T4

右旋后保持平衡

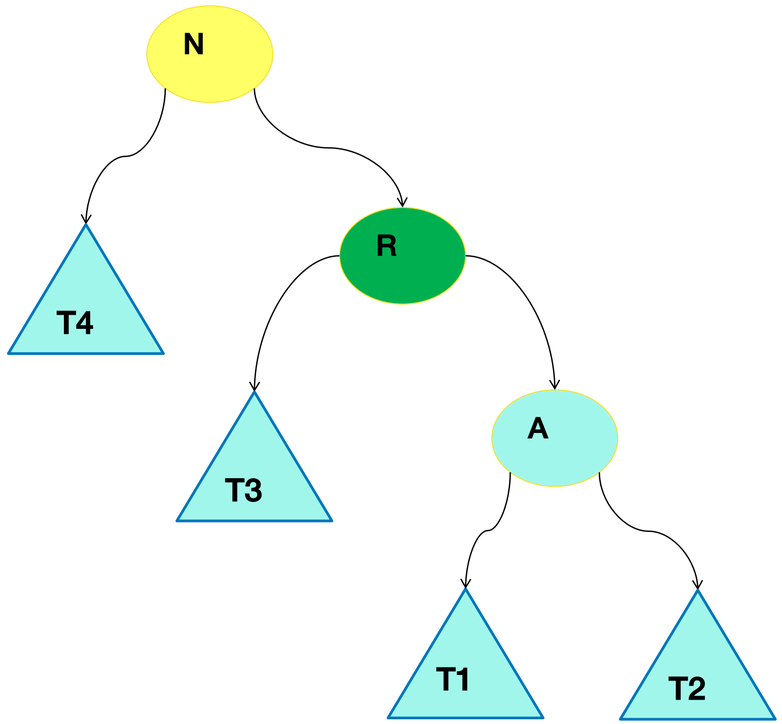


R.right=N
N.left=T3

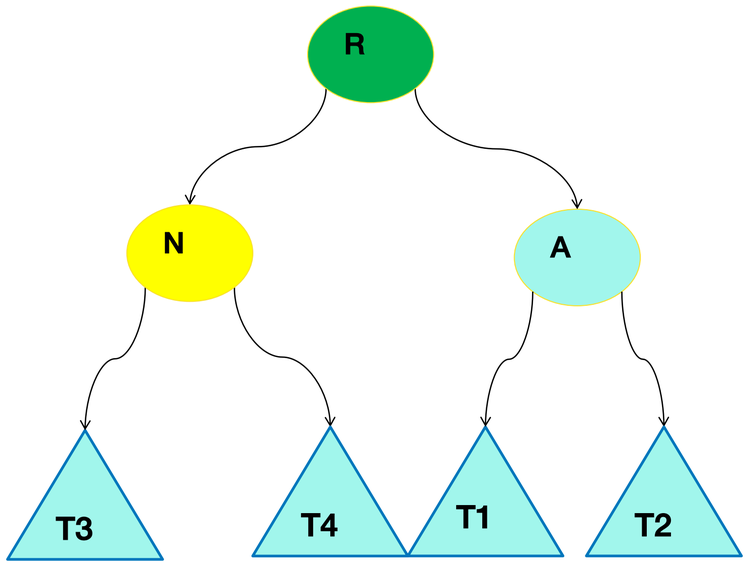


T1<A<T2<R<T3<N<T4

左旋

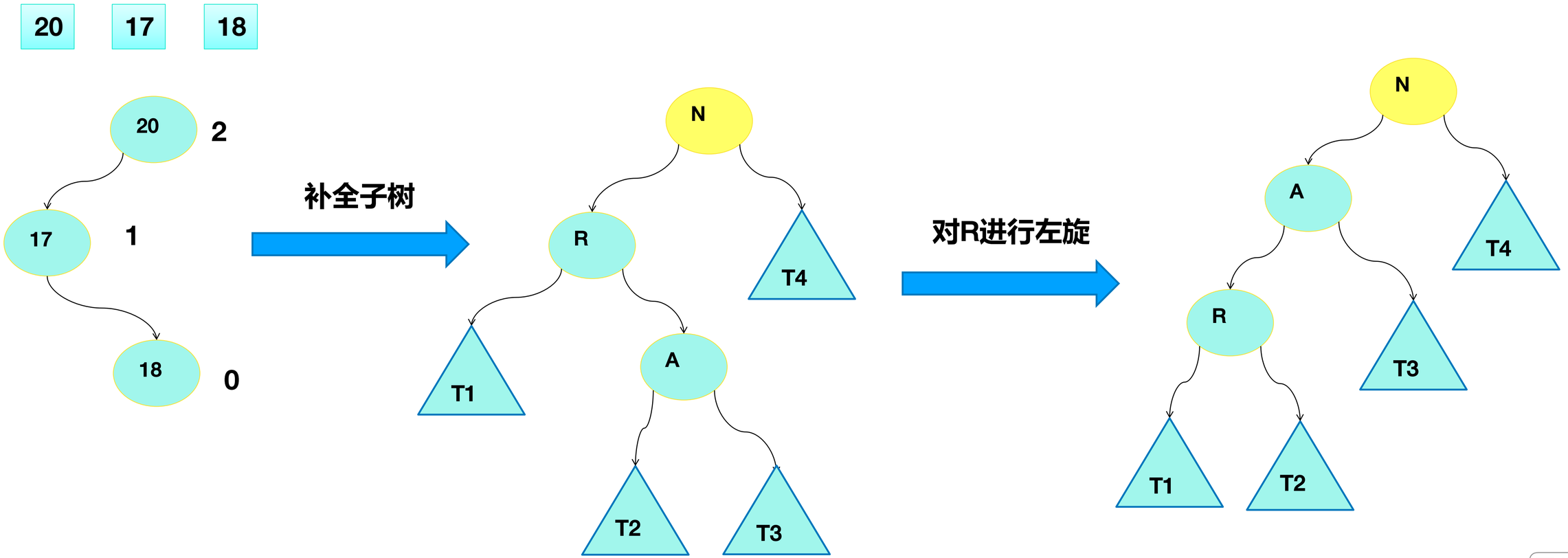


R.left=N
N.right =T3

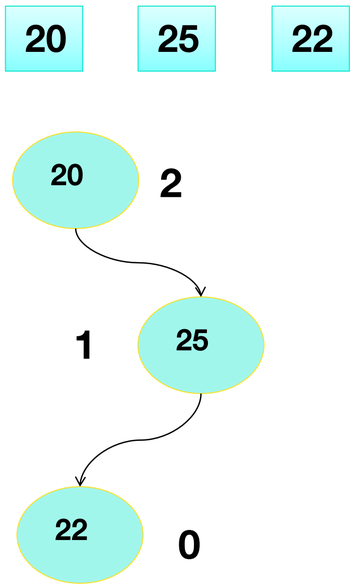


T4<N<T3<R<T1<A<T2

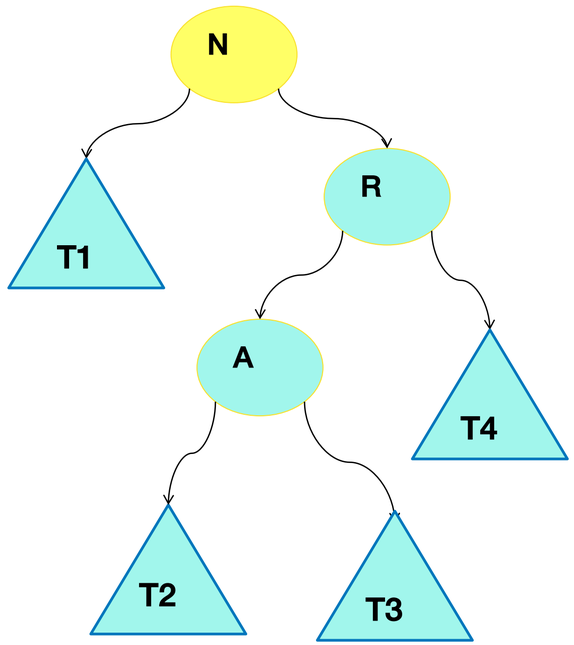
LR出现不平衡


20
17
18
20
2
17
1
18
0
补全子树
N
R
T4
T1
A
T2
T3
对R进行左旋
N
A
T4
R
T3
T1
T2
转换成了LL的情况

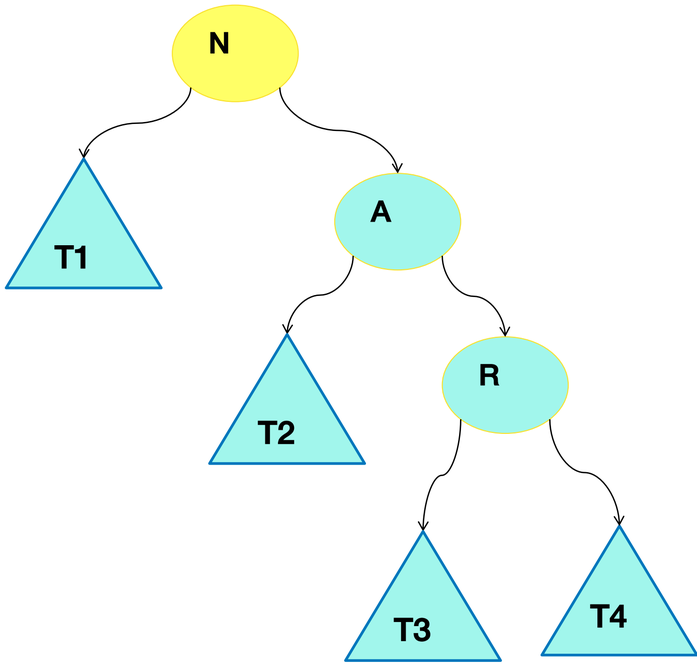
RL出现不平衡



补全子树



对R进行右旋



转换成了RR的情况

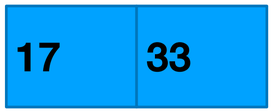
红黑树定义

- 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）
- 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

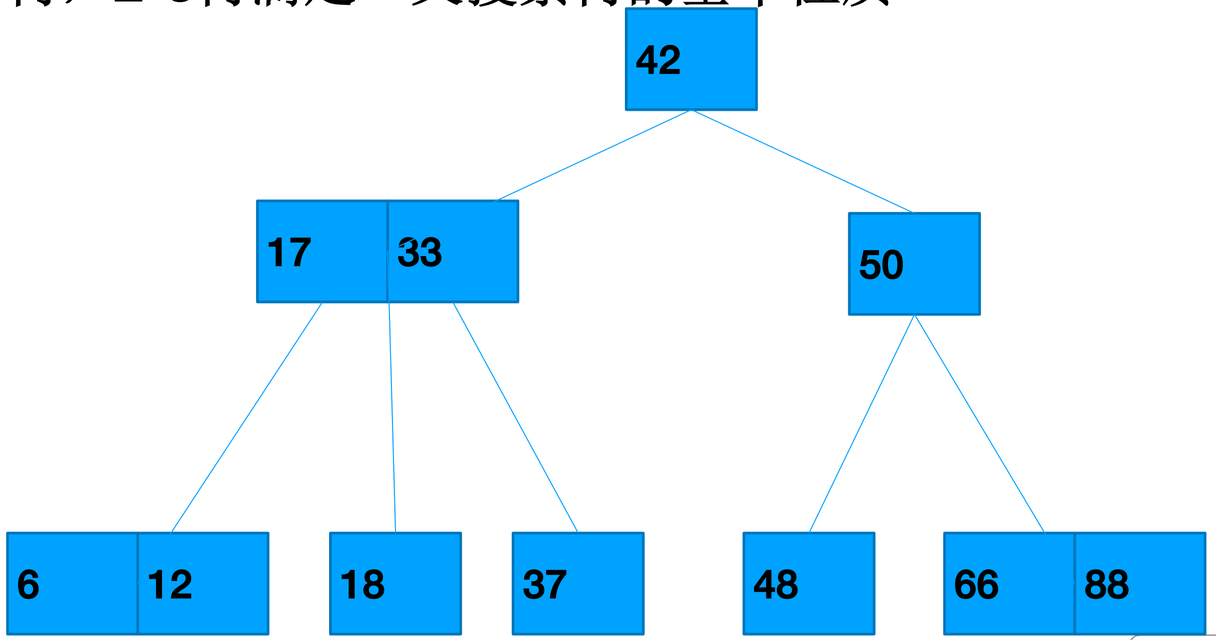
2-3树的特征

- 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

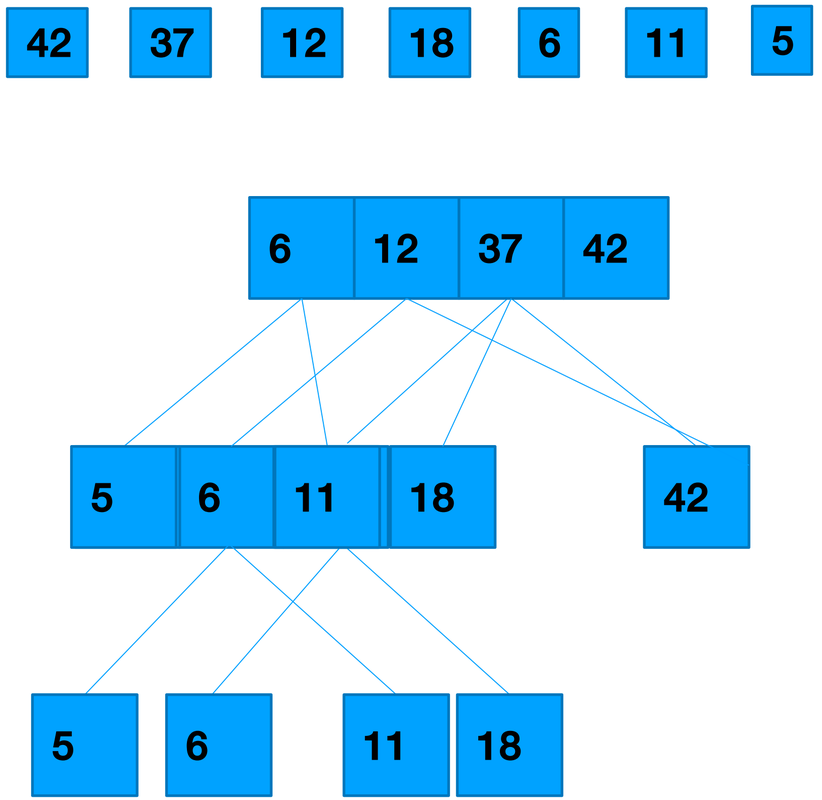


2-3树

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

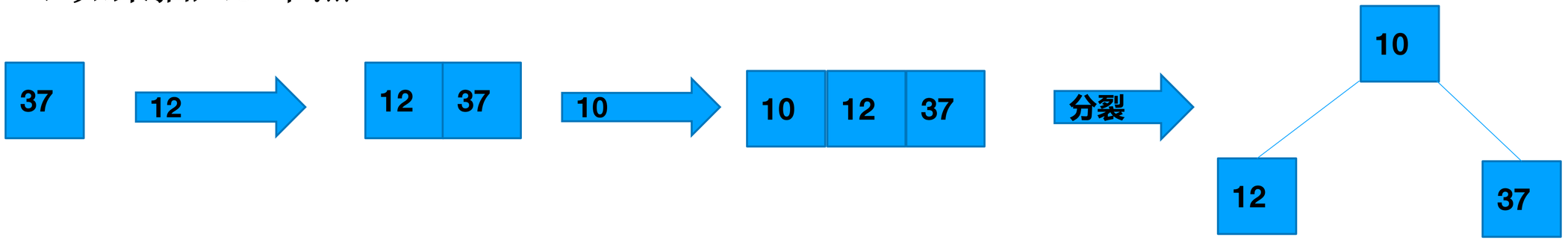
- 满足二叉搜索树的特征
- 维持绝对平衡
- 不能往**null**节点插入数据



2-3树添加节点

◆ 如果插入2-节点

◆ 如果插入3-节点



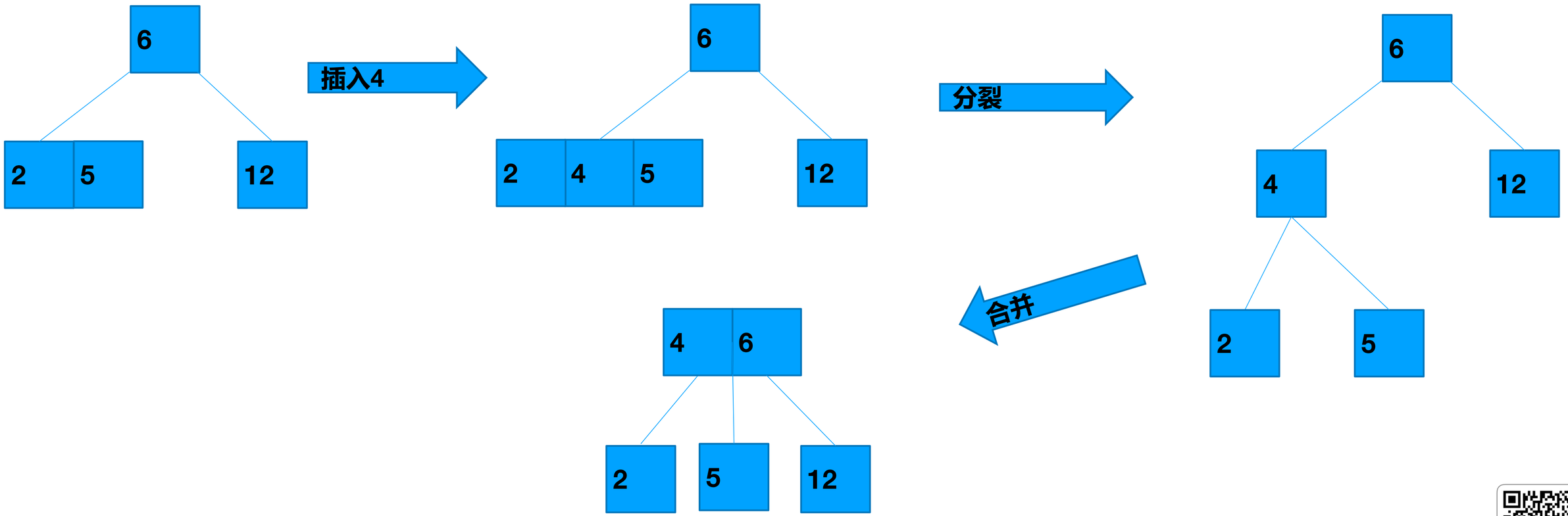
◆ 2-3树中添加一个新元素，或者添加到2-节点或者添加到3-节点

◆ 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

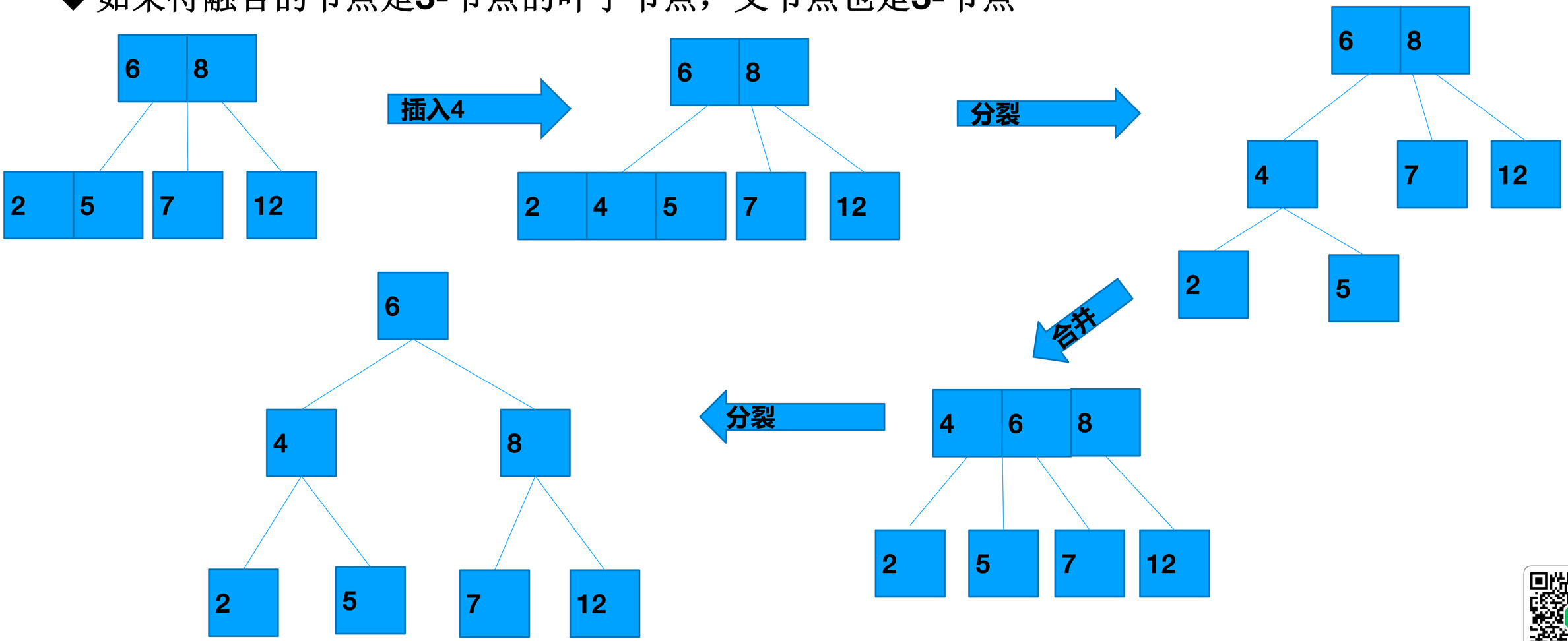
2-3树添加节点

◆ 如果待融合的节点是3-节点的叶子节点，父节点是2-节点

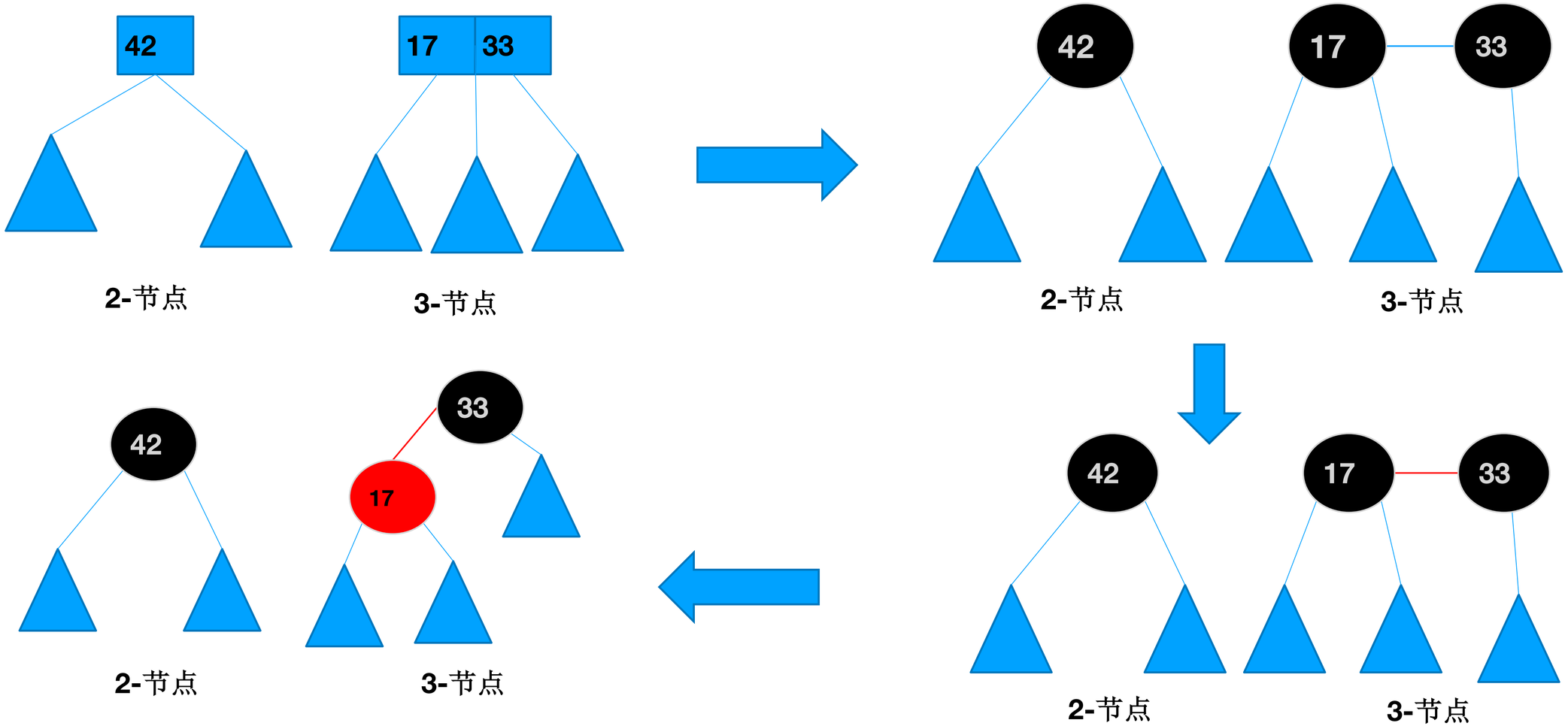


2-3树添加节点

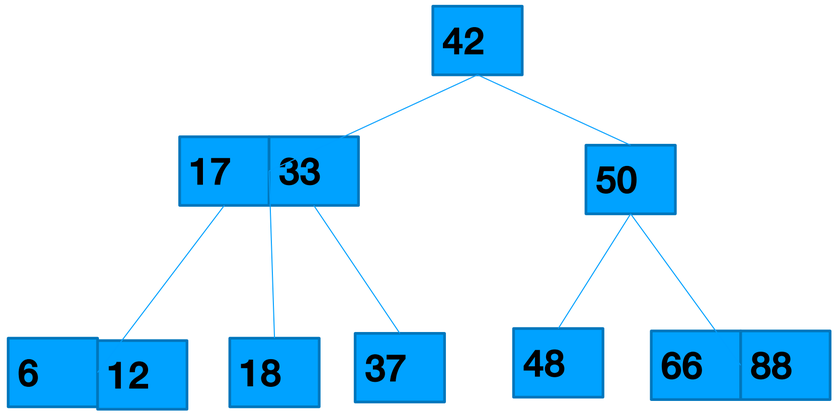
◆ 如果待融合的节点是3-节点的叶子节点，父节点也是3-节点

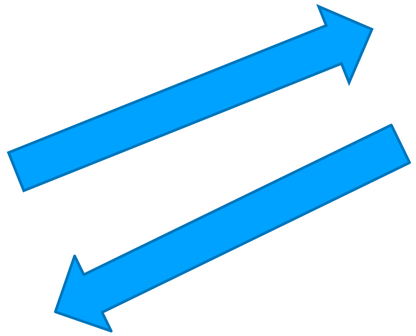


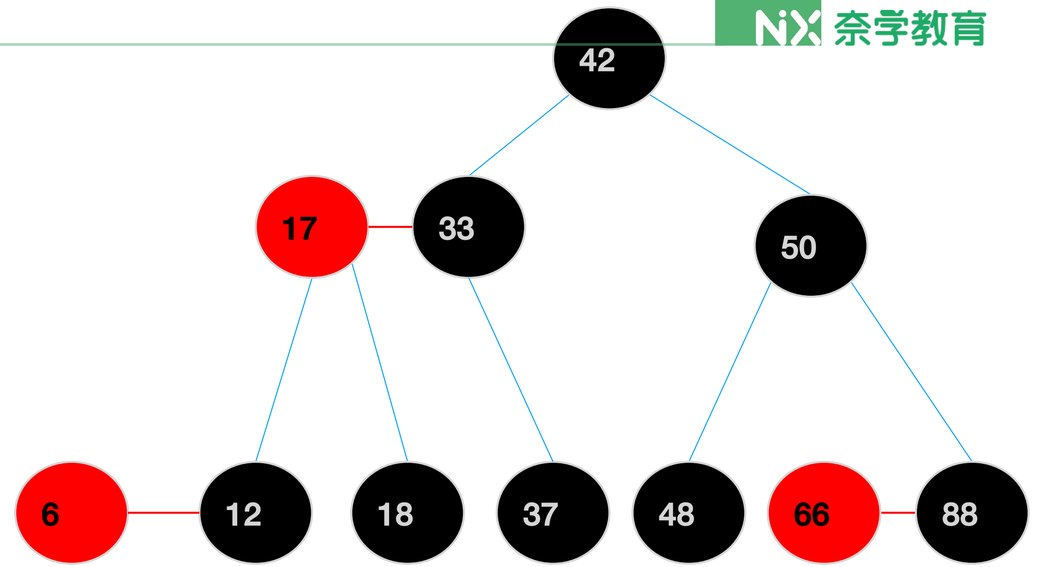
2-3树和红黑树等价性

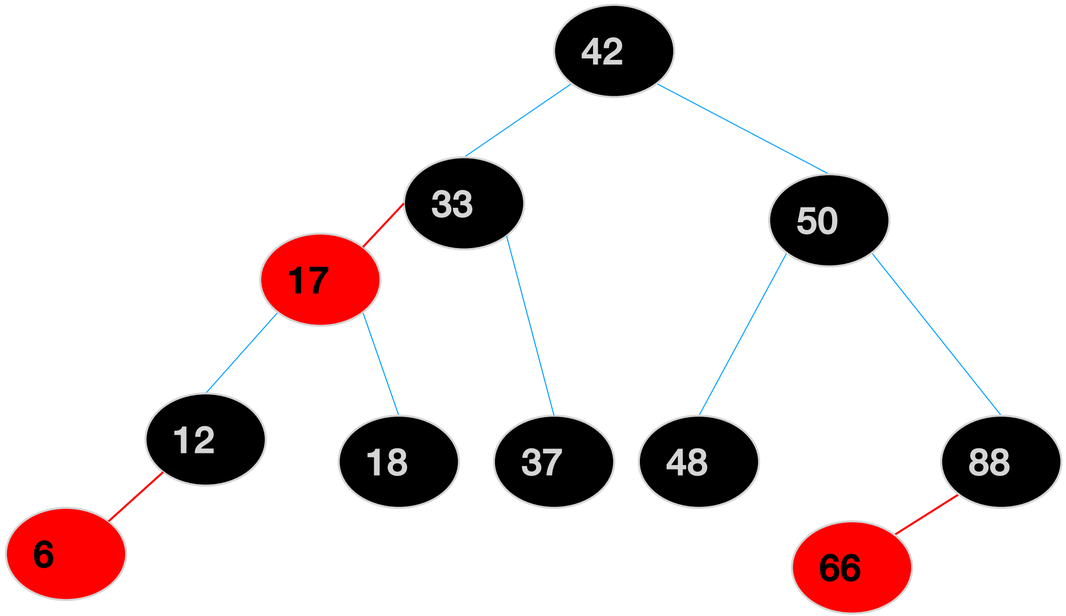


2-3树和红黑树等价性



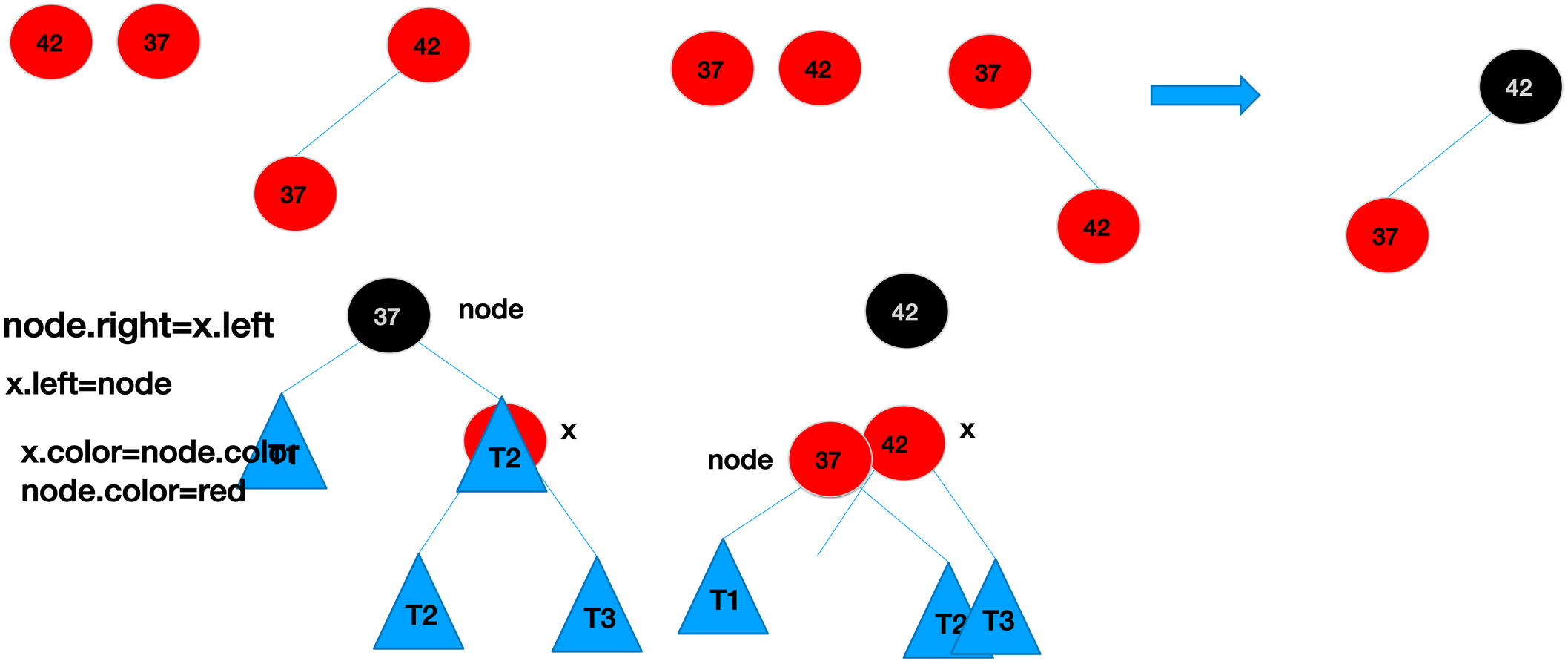




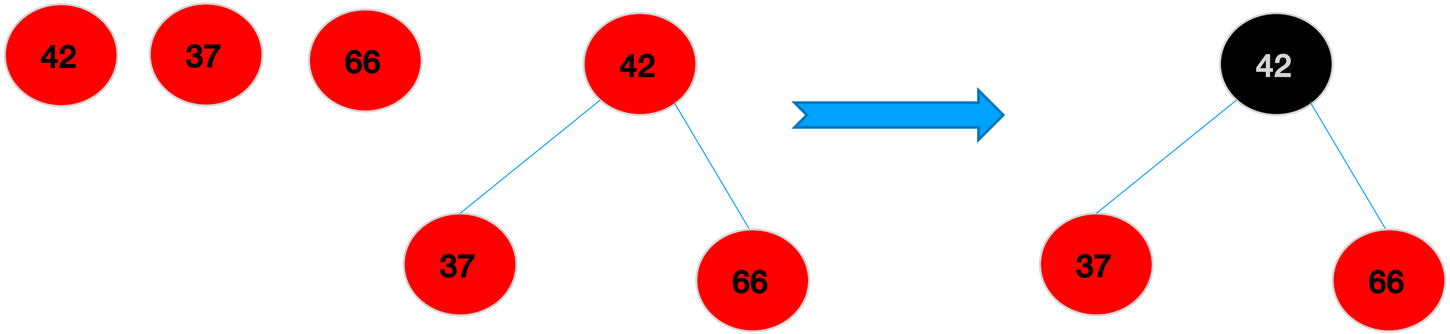


- 红黑树的每个节点或是红色或者是黑色
- 根节点是黑色的，红色节点向左倾斜
- 每个叶子节点都是黑色的
- 如果一个节点是红色的，那么他的孩子节点都是黑色的
- 从任何一个节点到叶子节点，经过的黑色节点是一样的

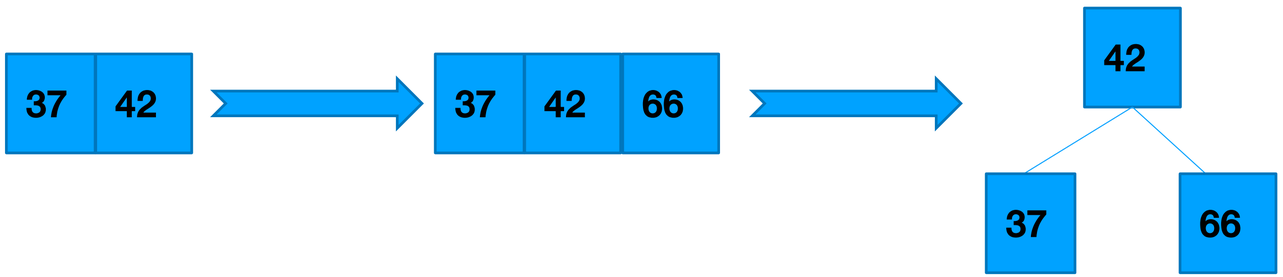
红黑树添加元素-保持根是黑色的、左旋转



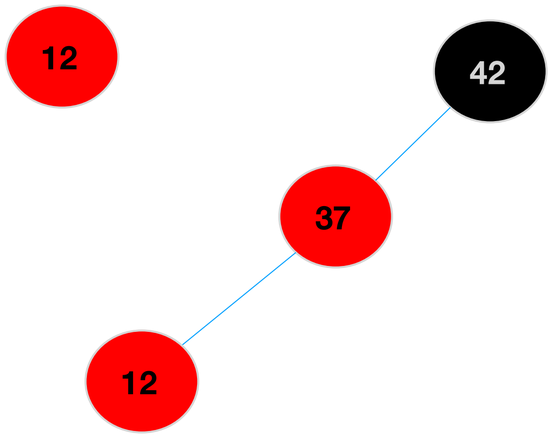
红黑树添加元素-颜色翻转



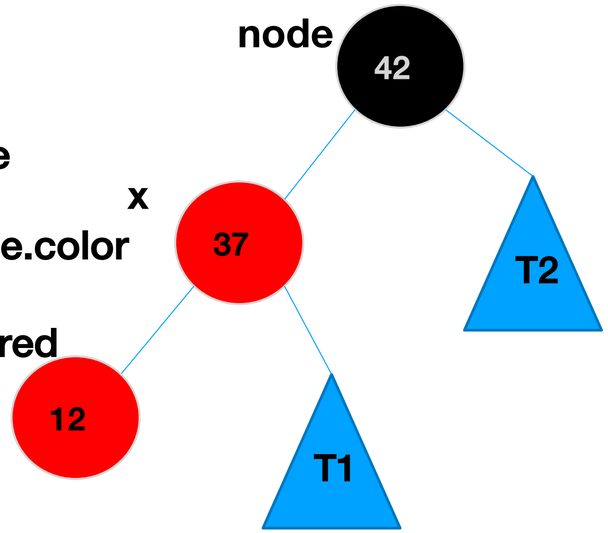
flipColors 颜色翻转

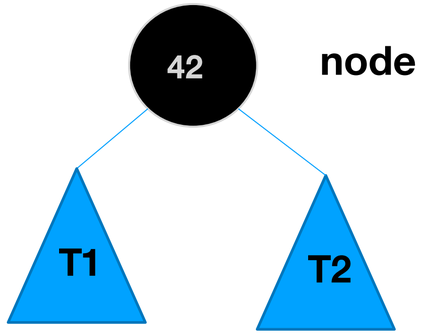


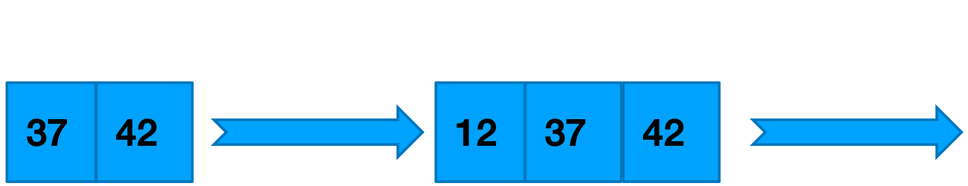
红黑树添加元素-右旋

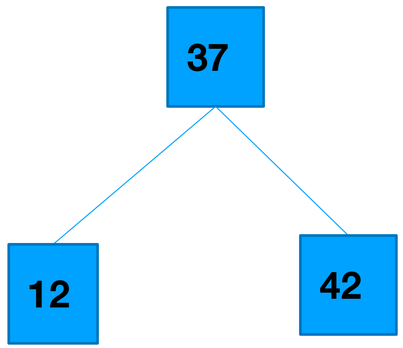


- node.left=T1
- x.right=node
- x.color=node.color
- node.color=red
- flipColors

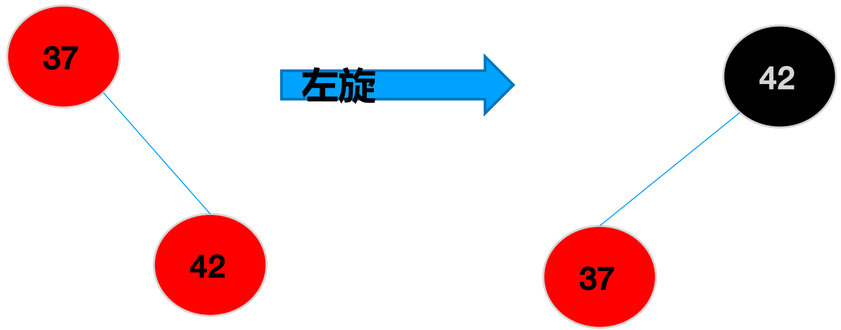


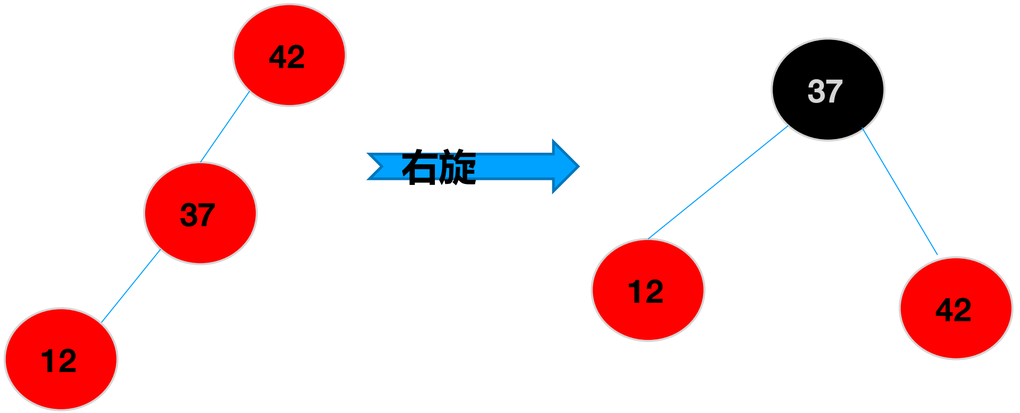


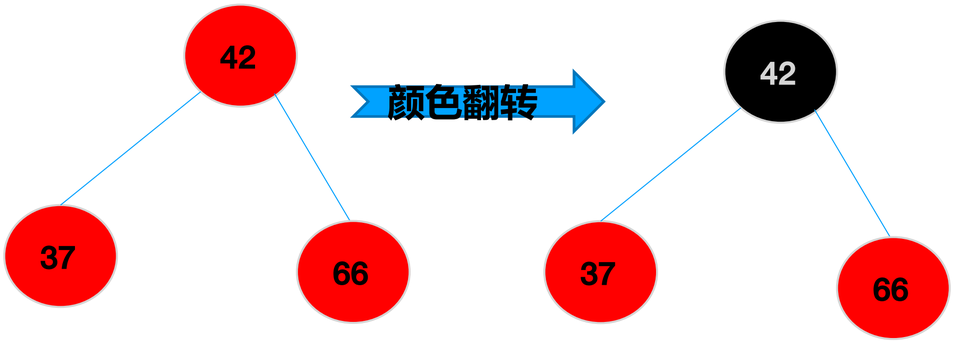


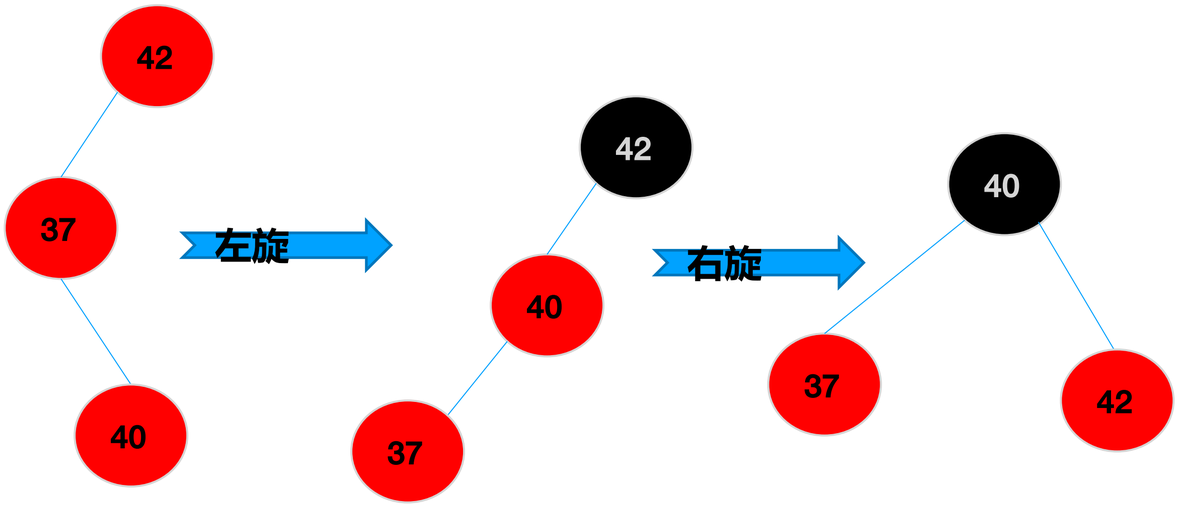


红黑树添加元素

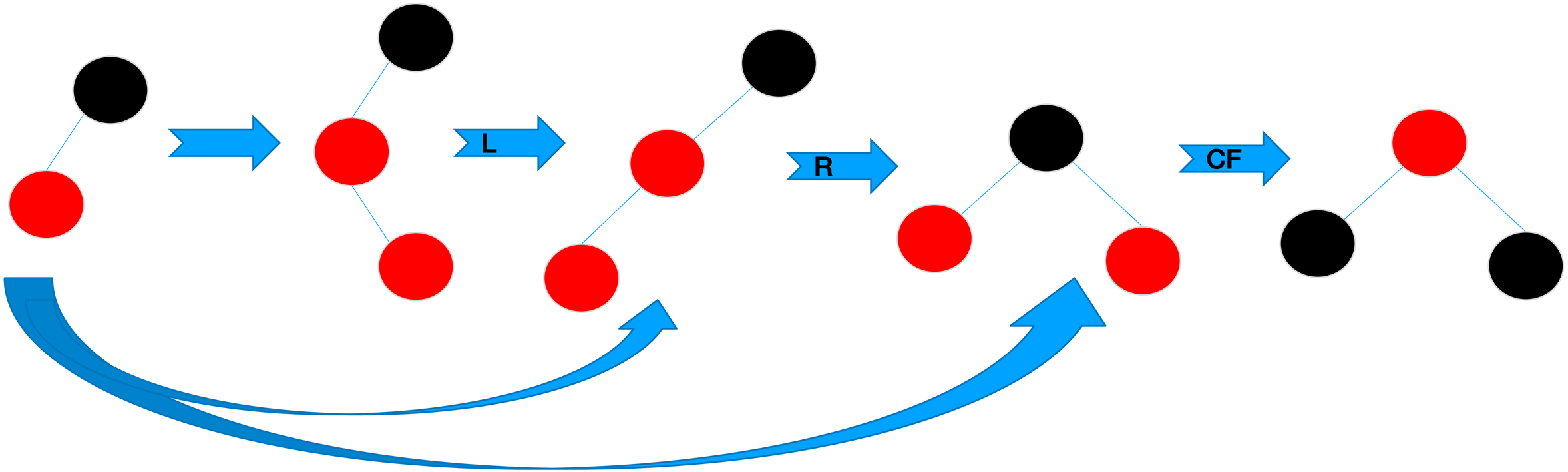

37
42
左旋
42
37


42
37
12
右旋
37
12
42


42
37
66
颜色翻转
42
37
66

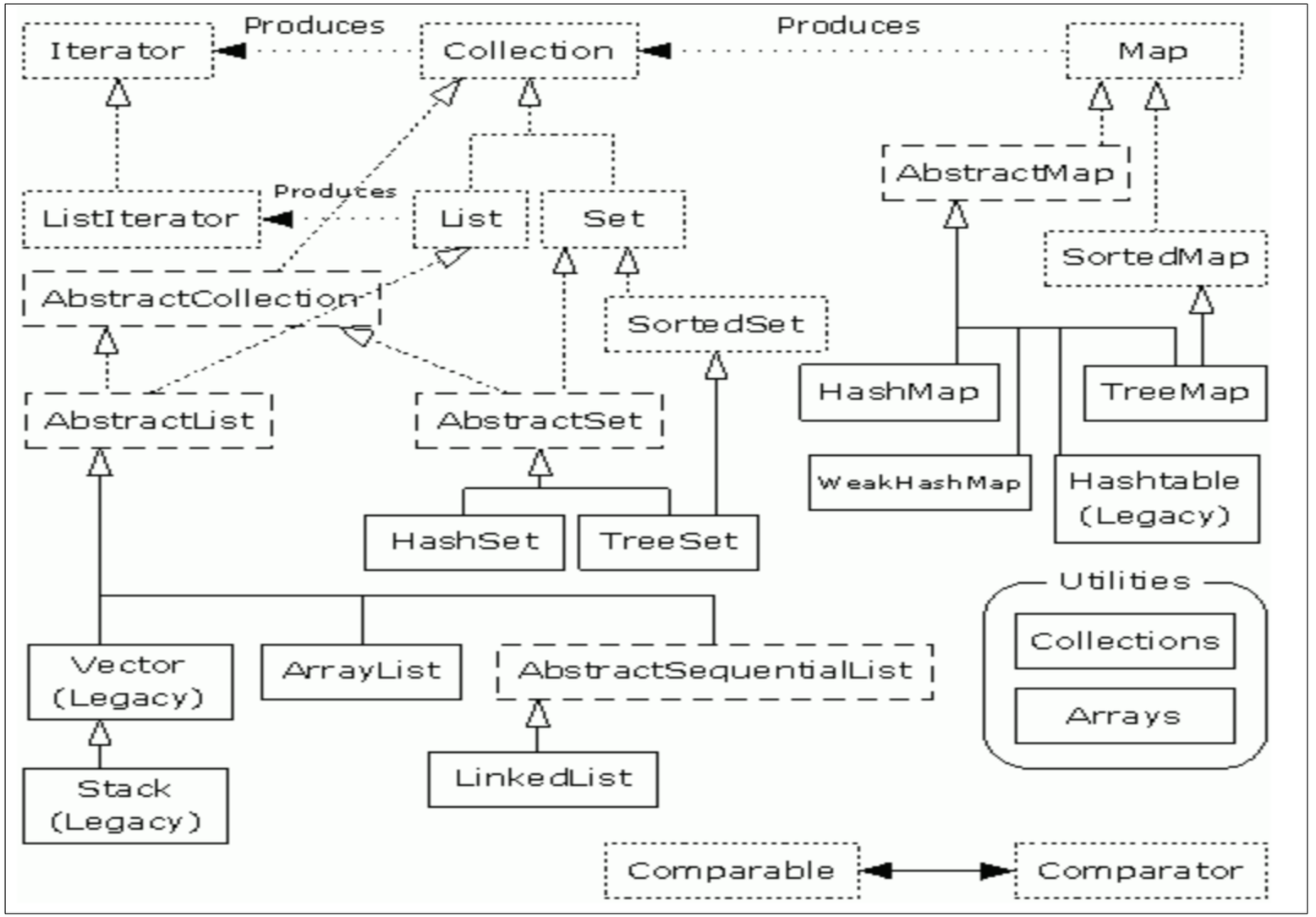

42
37
40
左旋
42
40
37
右旋
40
37
42

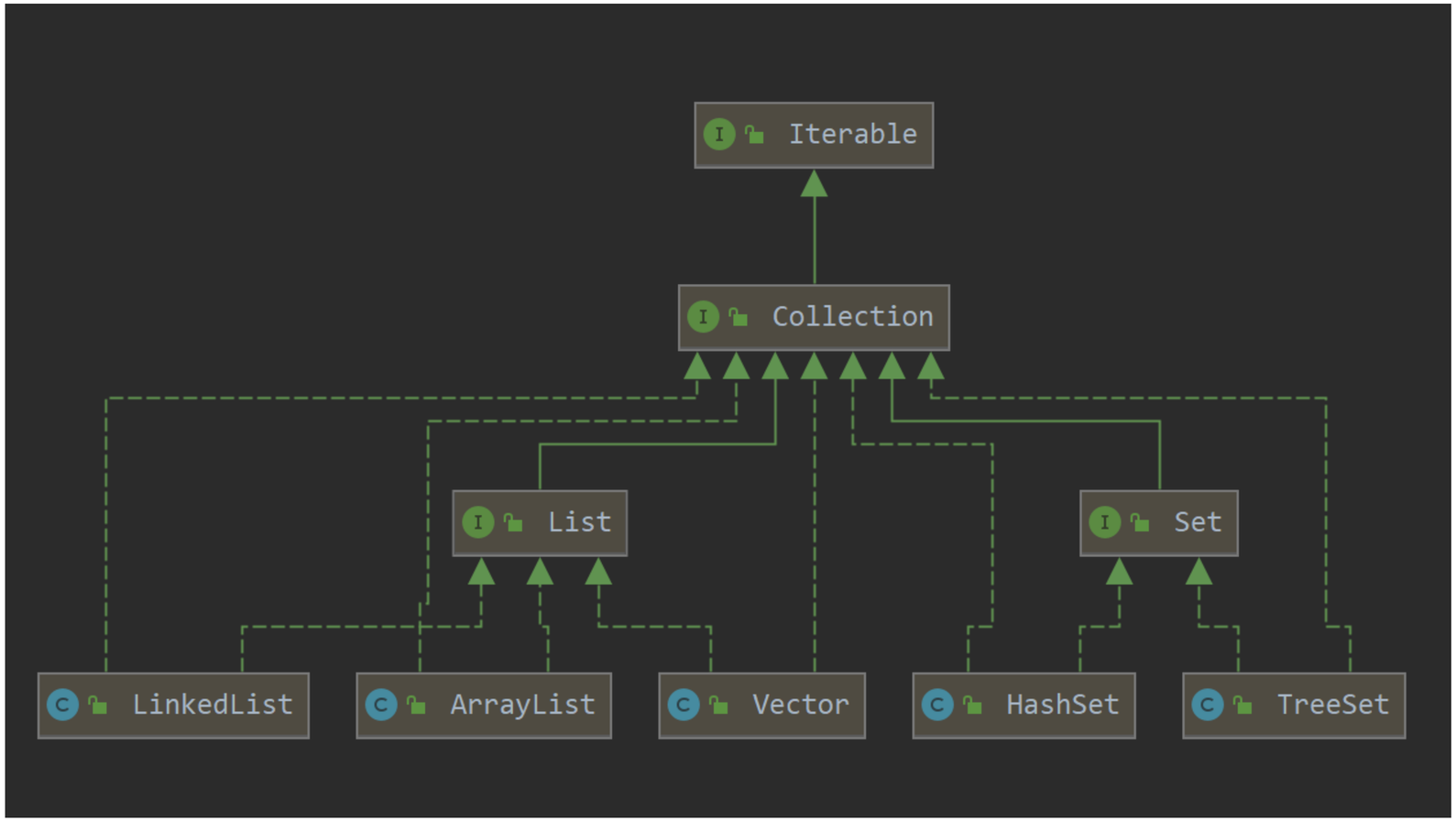
红黑树添加元素

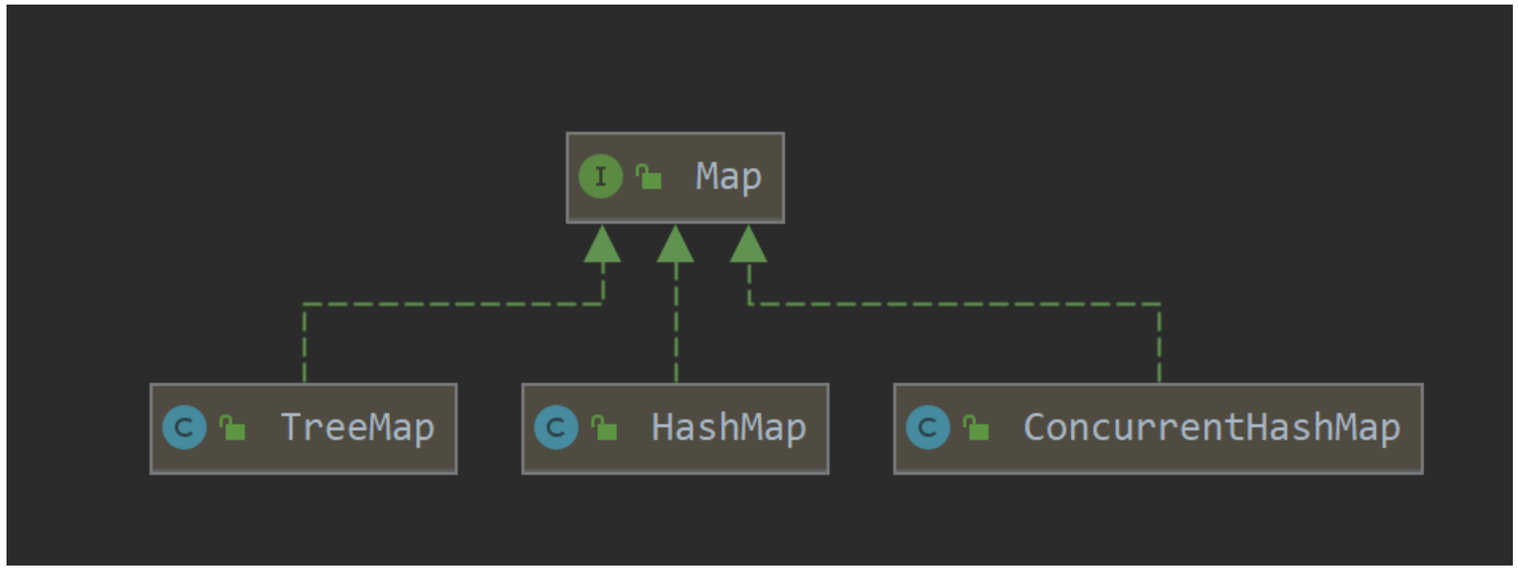


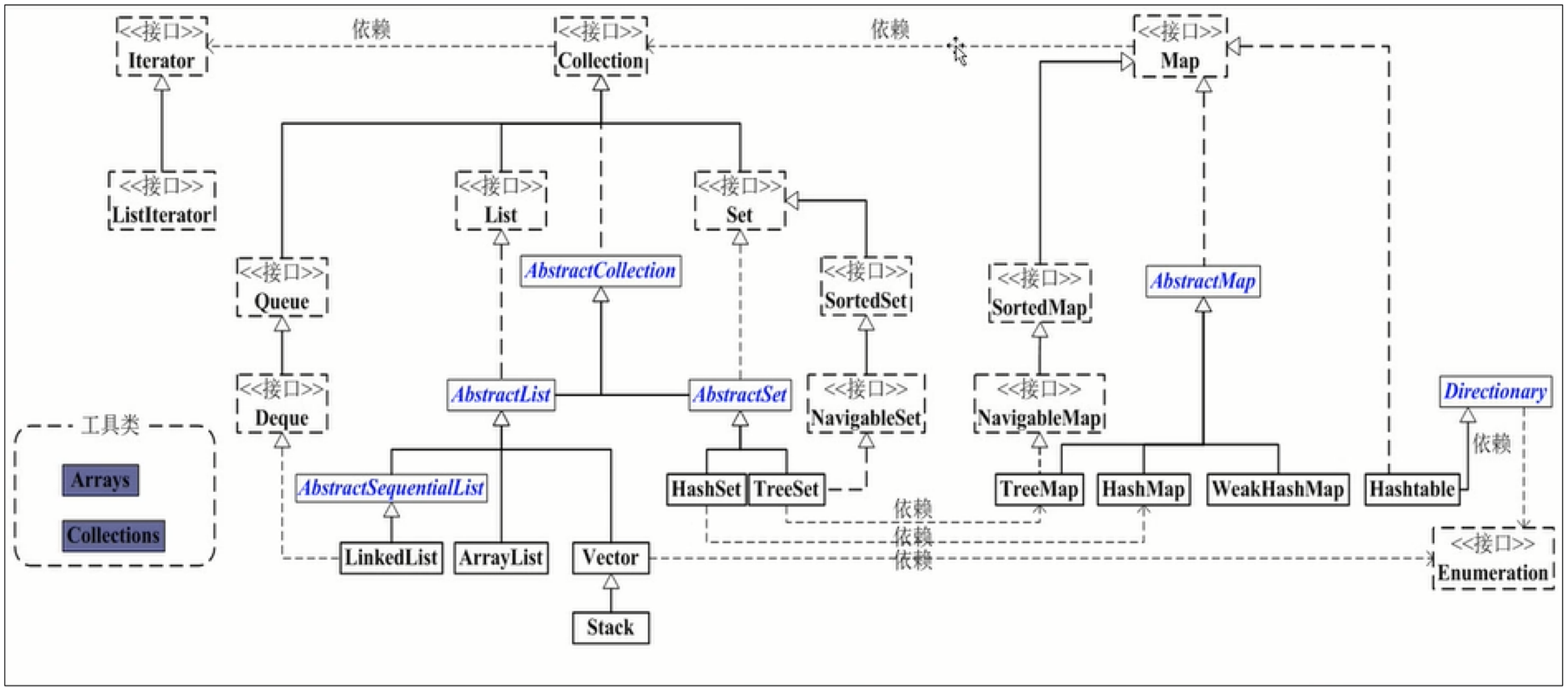
三种树的应用场景

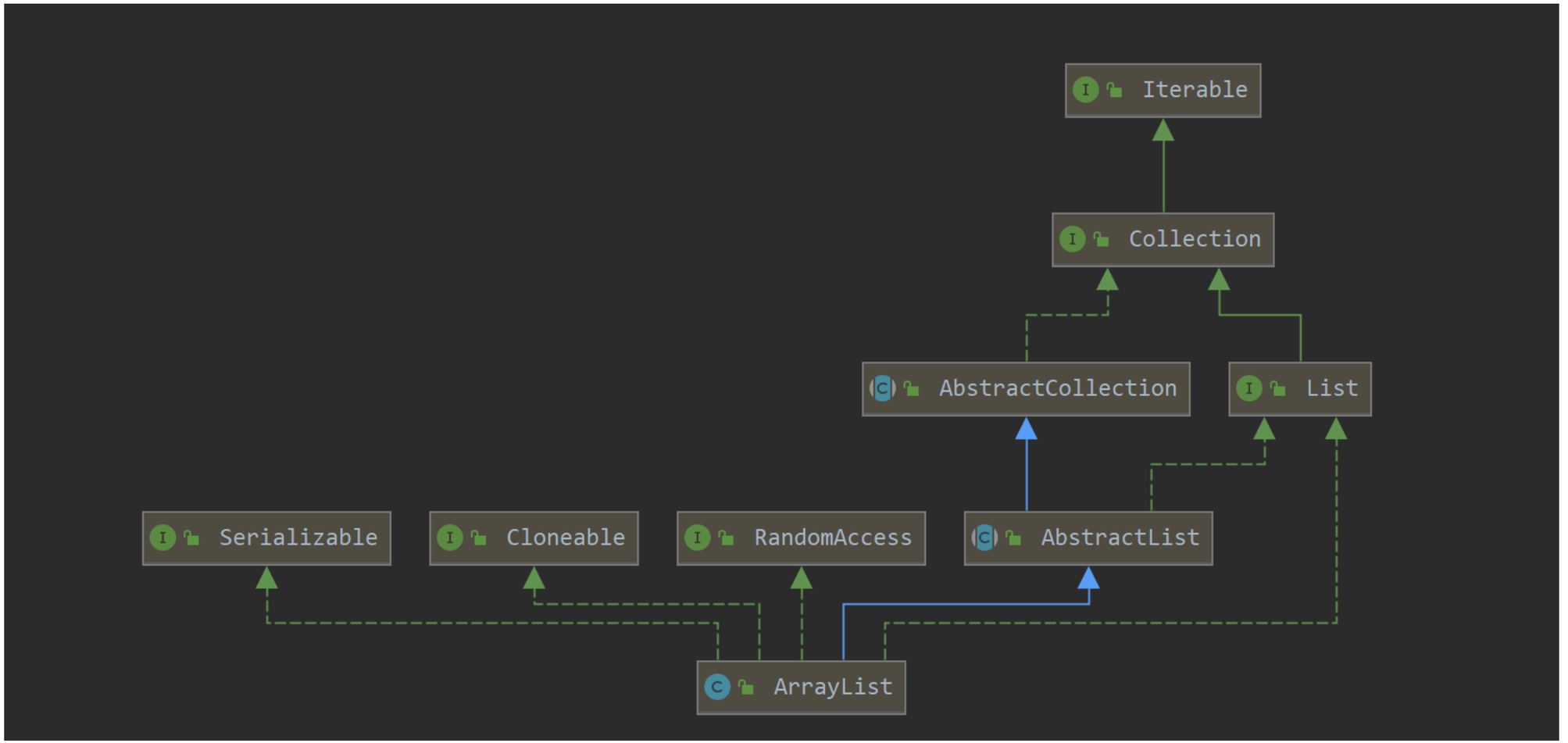
- 对于完全随机的数据，**BST**不会出现一侧偏斜的情况，极端情况下会退化成链表或者非常不平衡树
- 对于查询较多的业务场景，**AVL**是最佳的选择
- **RBT**的综合性能更优
- **HashMap**在**Java8**后引入了红黑树，**TreeMap**、**TreeSet**的底层也是红黑树

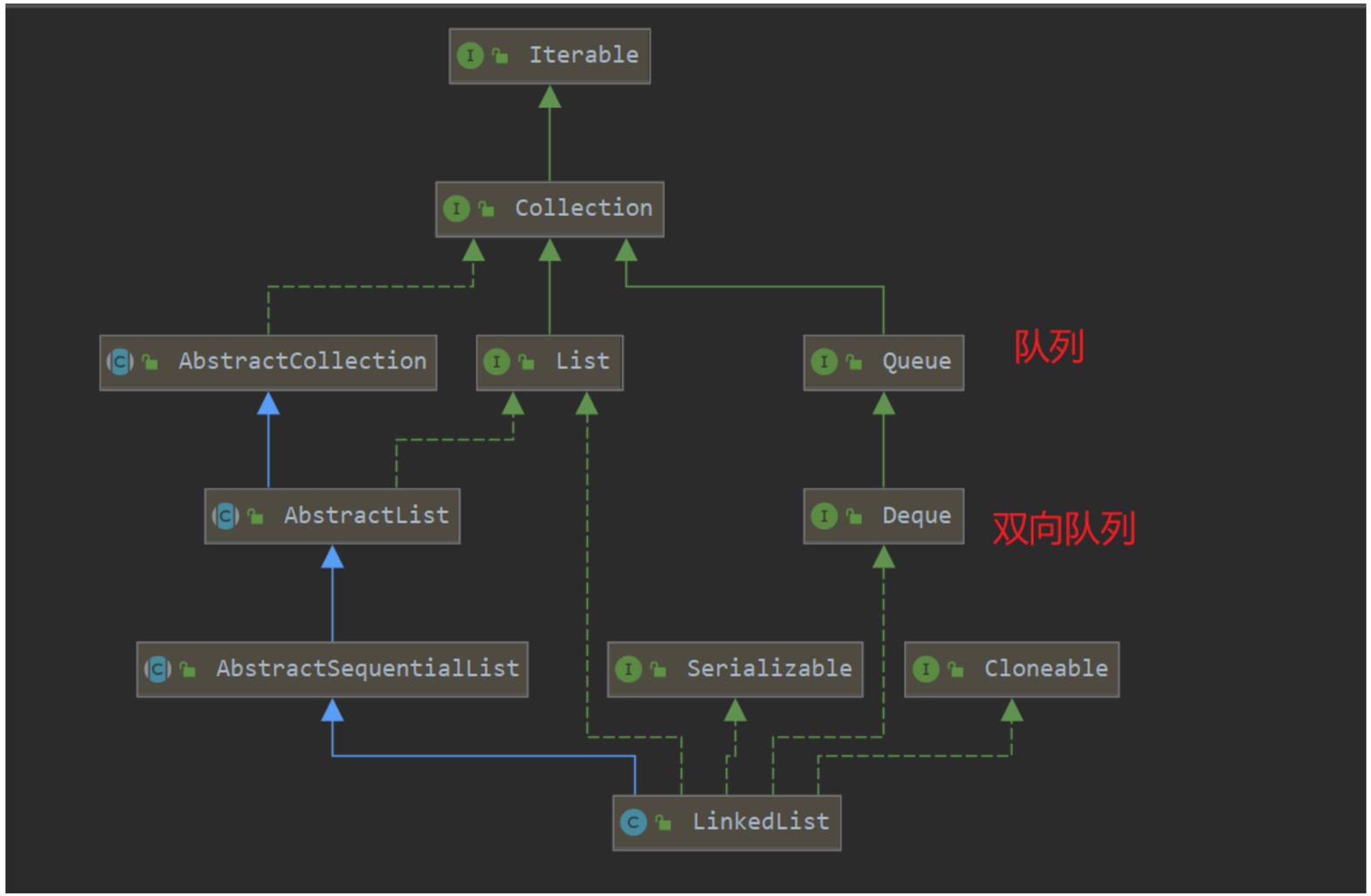

Iterator
Produces
Collection
Produces
Map
ListIterator
Produces
List
Set
AbstractMap
SortedMap
AbstractCollection
SortedSet
AbstractList
AbstractSet
HashMap
TreeMap
WeakHashMap
Hashtable
(Legacy)
HashSet
TreeSet
Utilities
Collections
Arrays
Vector
(Legacy)
ArrayList
AbstractSequentialList
Stack
(Legacy)
LinkedList
Comparable
Comparator

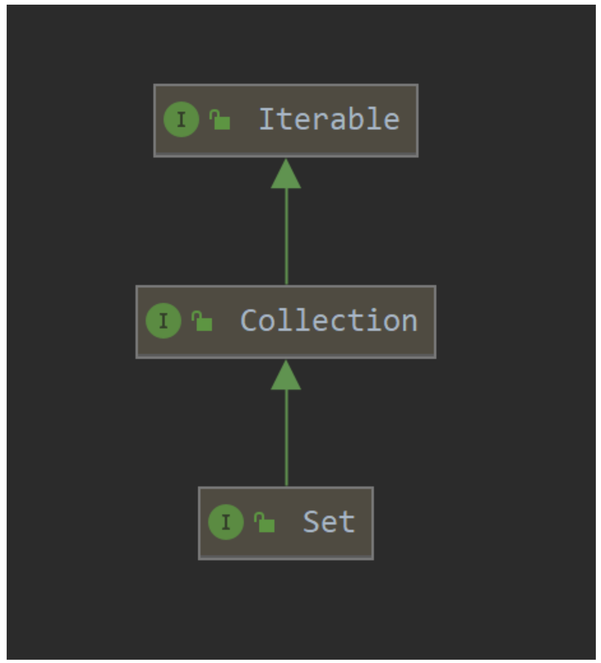

Iterable
Collection
List
Set
LinkedList
ArrayList
Vector
HashSet
TreeSet

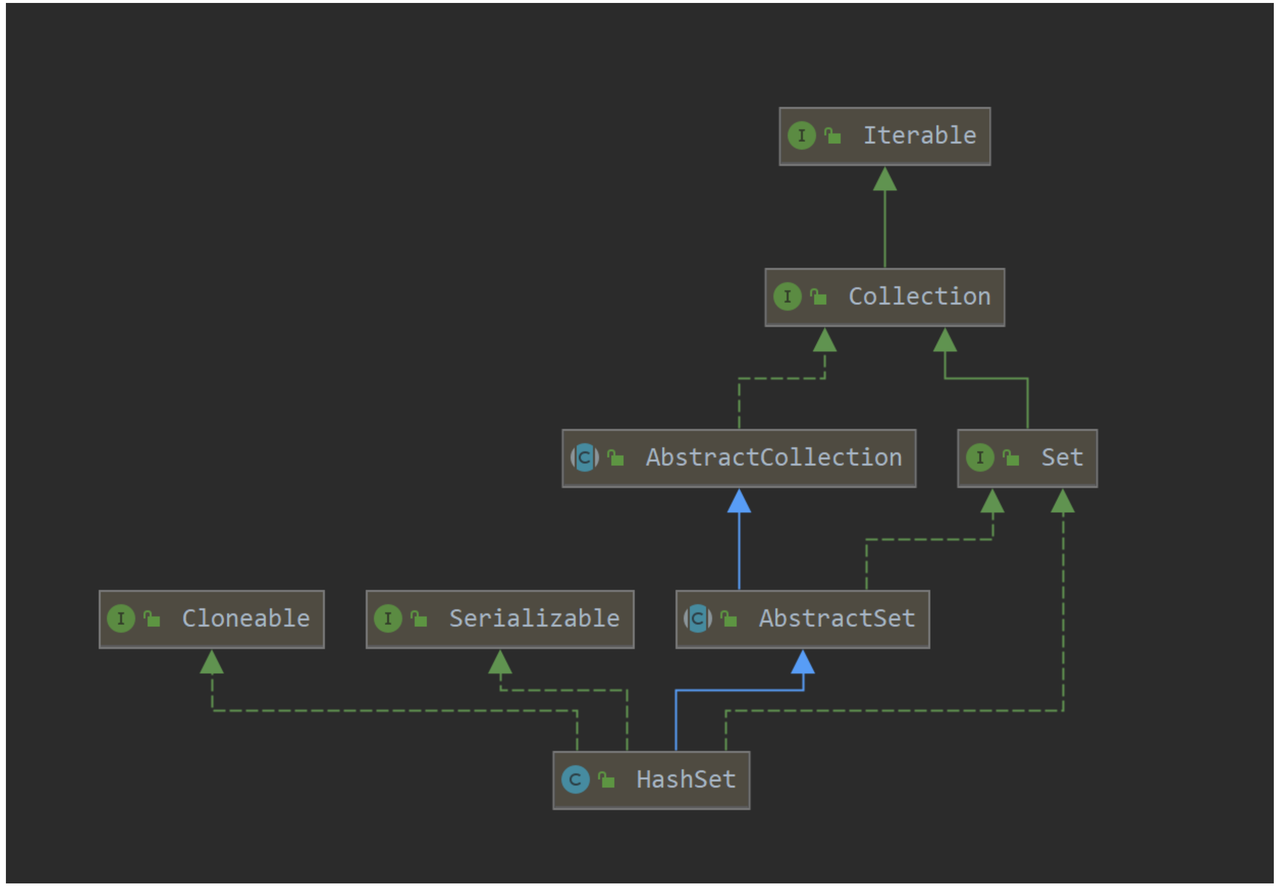

Map
TreeMap
HashMap
ConcurrentHashMap

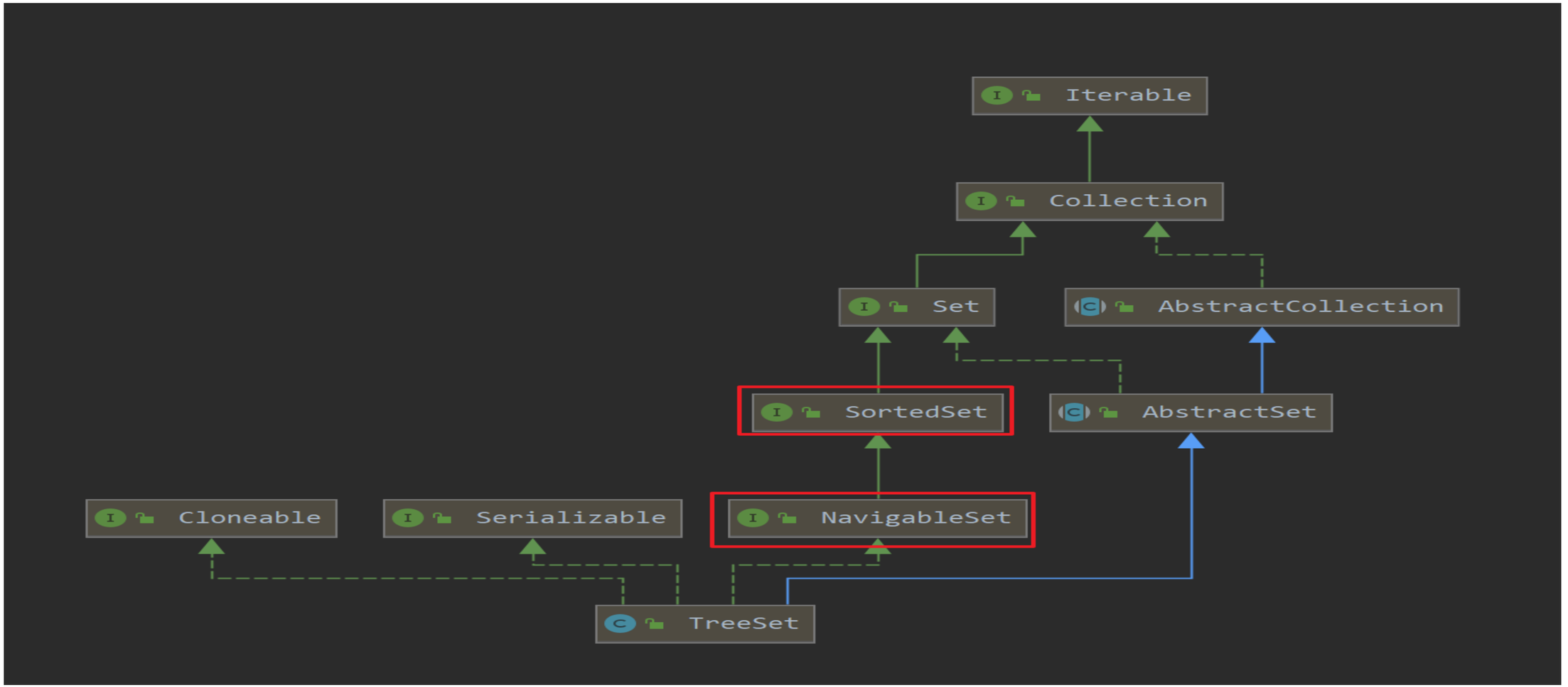

<<接口>>
Iterator
依赖
<<接口>>
Collection
依赖
<<接口>>
Map
<<接口>>
ListIterator
<<接口>>
List
<<接口>>
Set
<<接口>>
Queue
AbstractCollection
<<接口>>
SortedSet
<<接口>>
SortedMap
AbstractMap
<<接口>>
Deque
AbstractList
AbstractSet
<<接口>>
NavigableSet
<<接口>>
NavigableMap
Directionary
工具类
Arrays
Collections
AbstractSequentialList
HashSet
TreeSet
TreeMap
HashMap
WeakHashMap
Hashtable
依赖
依赖
依赖
依赖
LinkedList
ArrayList
Vector
<<接口>>
Enumeration
Stack

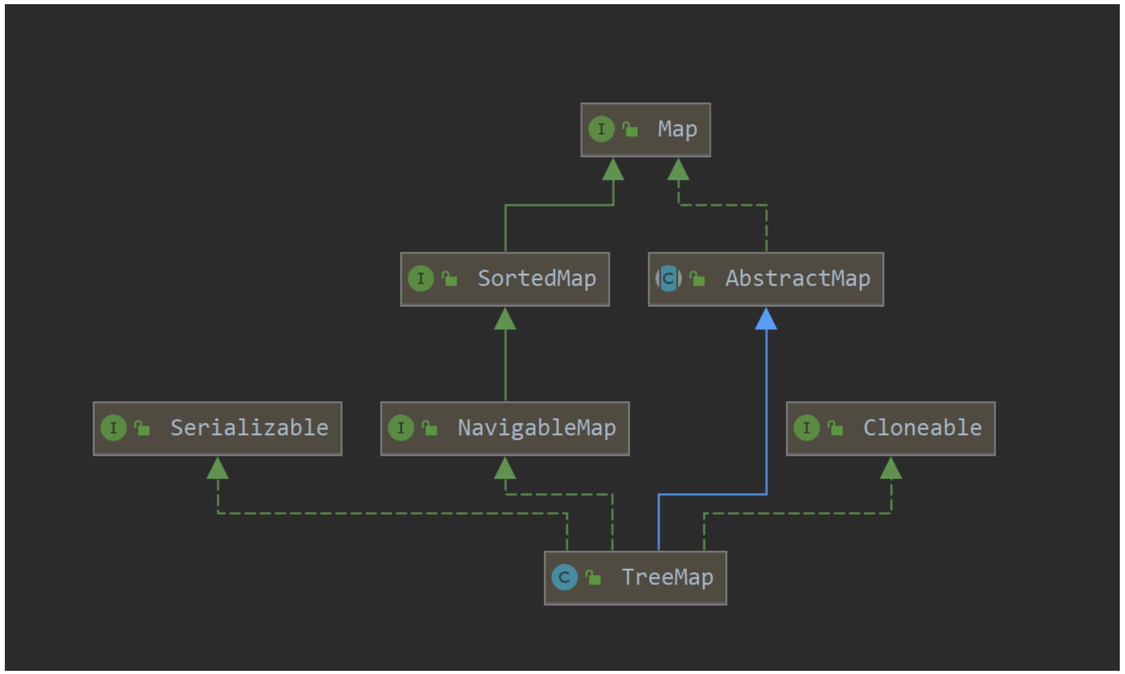

Iterable
Collection
AbstractCollection
List
Serializable
Cloneable
RandomAccess
AbstractList
ArrayList

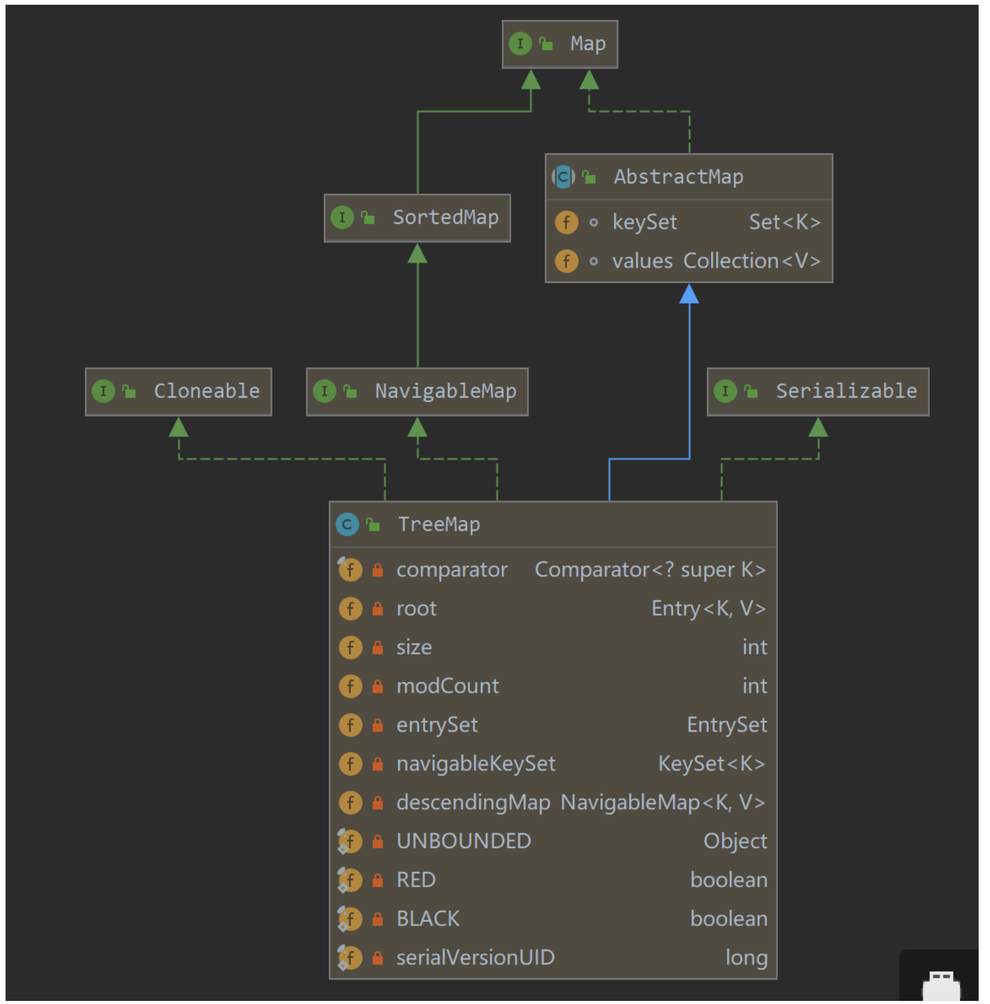

Iterable
Collection
AbstractCollection
List
Queue
队列
AbstractList
Deque
双向队列
AbstractSequentialList
Serializable
Cloneable
LinkedList

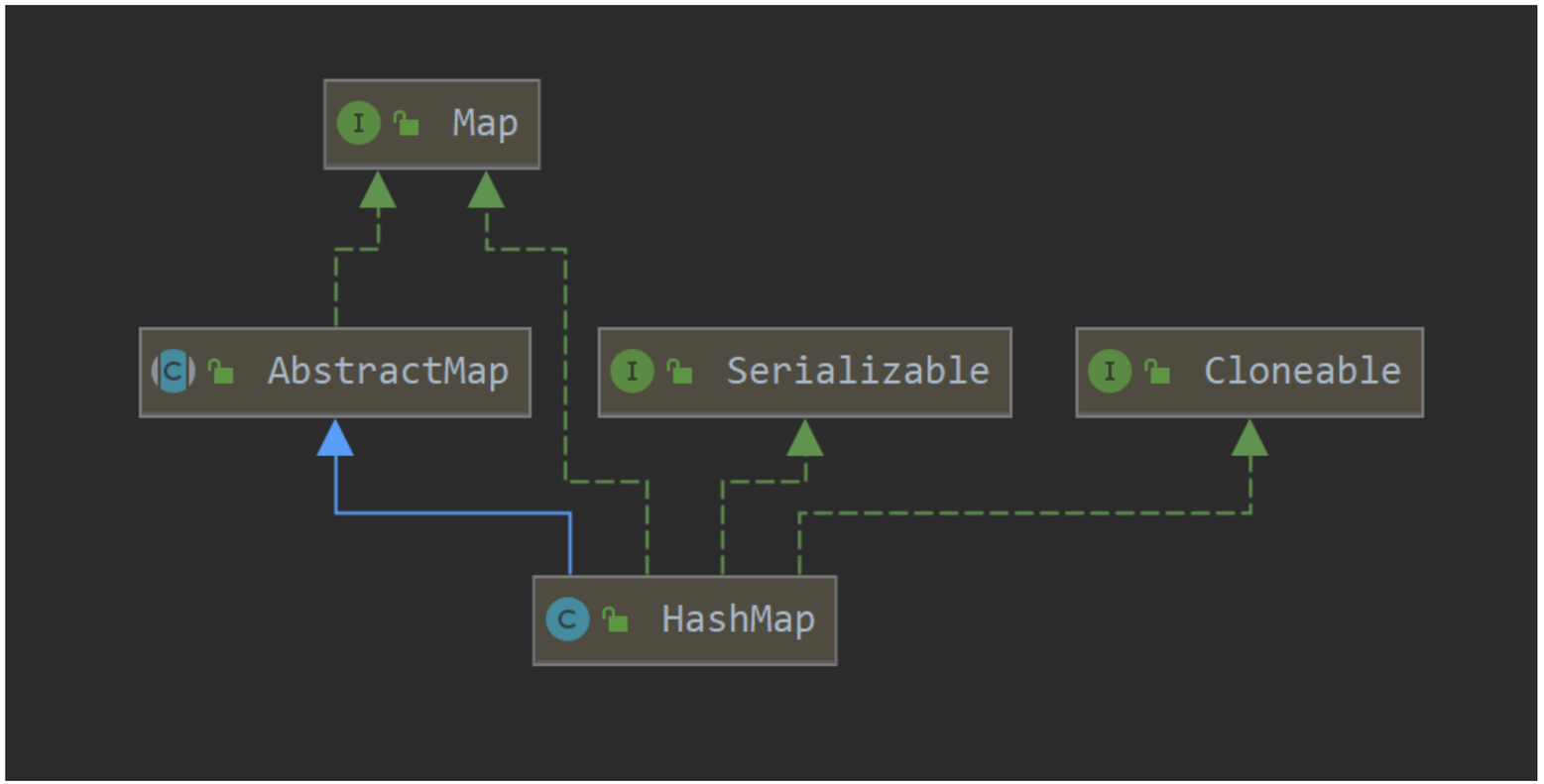

Iterable
Collection
Set

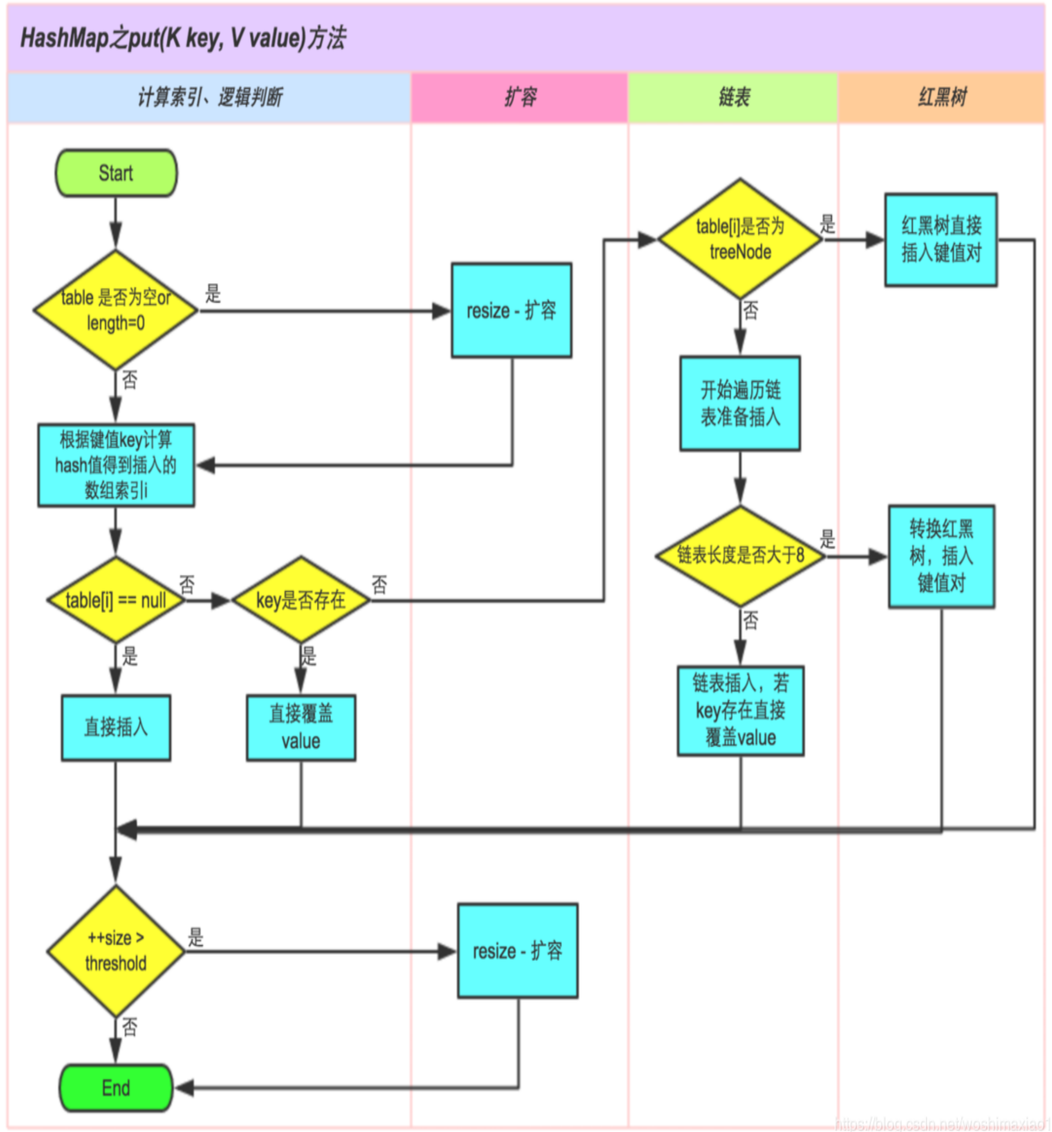

Iterable
Collection
AbstractCollection
Set
Cloneable
Serializable
AbstractSet
HashSet


Iterable
Collection
Set
AbstractCollection
SortedSet
AbstractSet
Cloneable
Serializable
NavigableSet
TreeSet


Map
SortedMap
AbstractMap
Serializable
NavigableMap
Cloneable
TreeMap


Map
SortedMap
AbstractMap
keySet Set<K>
values Collection<V>
Cloneable
NavigableMap
Serializable
TreeMap
comparator Comparator<? super K>
root Entry<K, V>
size int
modCount int
entrySet EntrySet
navigableKeySet KeySet<K>
descendingMap NavigableMap<K, V>
UNBOUNDED Object
RED boolean
BLACK boolean
serialVersionUID long


Map
AbstractMap
Serializable
Cloneable
HashMap


HashMap之put(K key, V value)方法
计算索引、逻辑判断
扩容
链表
红黑树
Start
table 是否为空or length=0
是
resize - 扩容
否
根据键值key计算hash值得到插入的数组索引i
table[i] == null
否
key是否存在
否
是
直接插入
是
直接覆盖value
table[i]是否为treeNode
是
红黑树直接插入键值对
否
开始遍历链表准备插入
链表长度是否大于8
是
转换红黑树，插入键值对
否
链表插入，若key存在直接覆盖value
++size > threshold
是
resize - 扩容
否
End