



泛型，数据结构，List接口， Set接口



黑马程序员™
www.itheima.com

传智播客旗下
高端IT教育品牌



目录 Contents

- ◆ 泛型
- ◆ 数据结构
- ◆ List集合
- ◆ Set集合

目录 Contents

- ◆ 泛型介绍
- ◆ 自定义泛型类
- ◆ 自定义泛型接口
- ◆ 自定义泛型方法
- ◆ 泛型通配符
- ◆ 受限泛型

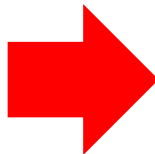
目标 TARGET

- ◆ 理解泛型的含义
- ◆ 理解泛型的好处

1 泛型的含义

泛型是一种类型参数，专门用来保存类型用的。

最早接触泛型是在`ArrayList<E>`，这个E就是所谓的泛型了。使用`ArrayList`时，只要给E指定某一个类型，里面所有用到泛型的地方都会被指定对应的类型。



```
ArrayList<String> list1 = new ArrayList<>(); // E = String
ArrayList<Integer> list2 = new ArrayList<>(); // E = Integer
ArrayList list3 = new ArrayList(); // E = Object
```

如果没有给泛型变量设定一个类型，默认表示Object。

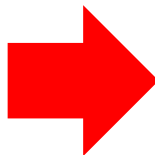
2 使用泛型的好处

不用泛型带来的问题：

集合若不指定泛型，默认就是Object。存储的元素类型自动提升为Object类型。获取元素时得到的都是Object，若要调用特有方法需要转型，给我们编程带来麻烦。

使用泛型带来的好处：

可以在编译时就对类型做判断，避免不必要的类型转换操作，精简代码，也避免了因为类型转换导致的错误。



```
// 泛型没有指定类型，默认就是Object
ArrayList list = new ArrayList();
list.add("Hello");
list.add("World");
list.add(100);
list.add(false);
// 集合中的数据就比较混乱，会给获取数据带来麻烦
for (Object obj : list) {
    String str = (String) obj;
    // 当遍历到非String类型数据，就会报异常出错
    System.out.println(str + "长度为：" + str.length());
}
```

3 注意

泛型在代码运行时，泛型会被擦除。后面学习反射的时候，可以实现在代码运行的过程中添加其他类型的数据到集合。

泛型是什么？

泛型是一种类型参数，专门用来保存类型用的
用来限定某一种数据类型的规范，泛型中存放引用数据类型

总结

泛型有什么好处？

在编译时期会做类型的检查，可以有效避免在运行时类型强转的异常，对于程序员来讲不用额外的类型强转操作，简化代码。

泛型在运行时有什么特点？

泛型在运行的时候，就会被擦除。

目标 TARGET

- ◆ 能够定义含有泛型的类
- ◆ 能够使用含有泛型的类，指定泛型类型

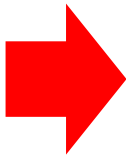
自定义泛型类

当在一个类中定义属性的时候，不确定属性具体是什么类型时，就可以使用泛型表示该属性的类型。

1 定义格式

在类型名后面加上一对尖括号，里面定义泛型。一般使用一个英文大写字母表示，如果有多个泛型使用逗号分隔。

```
public class 类名<泛型名>{  
    //类型内部，就可以把泛型名当做是某一种类型使用了。  
}
```

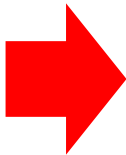


```
public class Student<X,Y>{  
    X xObj;  
}
```

2 泛型的确定

当创建对象时，确定泛型类中泛型的数据类型

举例：ArrayList<String> list = new ArrayList<>();



```
Student<String,Integer> stu = new Student<>();
```

3 代码实践

定义一个人类，定义一个属性表示爱好，但是具体爱好是什么不清楚，可能是游泳，乒乓，篮球。



代码实践

自定义泛型类

定义类时，什么时候可以使用泛型（使用场景）？

类中定义属性不知道具体类型时，就可以使用泛型。

总结

类中如何定义格式？

```
public class 类名<泛型名>{  
}
```

泛型使用时，如何确定具体类型？

创建对象的时候可以指定具体泛型

目录 Contents

- ◆ 能够定义含有泛型的接口
- ◆ 能够使用含有泛型的接口，给泛型指定具体类型

自定义泛型接口

当定义接口时，内部方法中其参数类型，返回值类型不确定时，就可以使用泛型替代了。

1 定义格式

在接口名后面加上一对尖括号，里面定义泛型。一般使用一个英文大写字母表示，如果有多个泛型使用逗号分隔。

```
public interface 接口名<泛型名>{  
    //类型内部，就可以把泛型名当做是某一种类型使用了。  
}
```



```
public interface Collection<E>{  
  
    public boolean add(E e);  
  
}
```

2 泛型的确定

可以在实现类实现接口时，确定接口中的泛型的类型

如果实现类和接口不指定具体的类型，继续使用泛型指定
变成含有泛型的类使用。

举例：public class Test<E> implements Collection<E>{}



```
public class CollectionImp implements  
Collection<String>{  
  
    public boolean add(String e);  
  
}
```

3 代码实践

模拟一个Collection接口，表示集合，集合操作的数据不确定。

定义一个接口MyCollection具体表示。

代码实践

什么时候使用泛型？

定义一个接口的时候，如果内部需要使用某一个数据不确定具体类型，就可以使用泛型。

总结

定义接口泛型的格式？

接口名后面加上尖括号，里面定义泛型名

怎么给泛型接口确定泛型？

子类如果可以确定类型，在实现接口的时候，直接确定类型
子类如果不确定类型，继续使用泛型指定，回到泛型类的使用

目标 TARGET

- ◆ 能够定义含有泛型的方法，并使用

自定义泛型方法

当定义方法时，方法中参数类型，返回值类型不确定时，就可以使用泛型替代了。

1 定义格式

在方法的返回值类型前，加上泛型

修饰符 <泛型名> 返回值类型 方法名(参数类别){
}



```
public interface Collection<E> extends Iterable<E> {  
    // ...  
    <T> T[] toArray(T[] a);    //将集合变成数组  
}
```

2 泛型的指定

调用含有泛型的方法时，传入的数据其类型就是泛型的类型

3 代码实践

定义存储字符串的ArrayList集合，将字符串的集合转换为字符串数组。

代码实践

自定义泛型方法

泛型方法其泛型的定义格式？

返回值前面定义泛型

什么时候可以确定方法中泛型的类型？

调用方法时传入数据，该数据是什么类型，泛型就是什么类型

总结

目标 TARGET

- ◆ 理解泛型通配符的意义
- ◆ 能够使用受限泛型

1 泛型通配符介绍

当我们对泛型的类型确定不了，而想要表达的可以是任意类型，可以使用泛型通配符给定。

符号就是一个问号：**?** 表示任意类型，用来给泛型指定的一种通配值。如下：

```
public static void shuffle(List<?> list){  
    //...  
}
```

说明：该方法时来自工具类Collections中的一个方法，用来对存储任意类型数据的List集合进行乱序。

2 泛型通配符使用

泛型通配符搭配集合使用一般在方法的参数中比较常见

在集合中泛型是不支持多态的，如果为了匹配任意类型，我们就会使用泛型通配符了。

但是 `ArrayList<E>` 和 `ArrayList<?>` 都可以接收任意类型,那么通配符存在的意义是什么？

受限泛型

受限泛型是指,在使用通配符的过程中,对泛型做了约束,给泛型指定类型时,只能是某个类型父类型或者子类型。

1. 泛型的下限

<? super 类型> 只能是某一类型, 及其父类型, 其他类型不支持

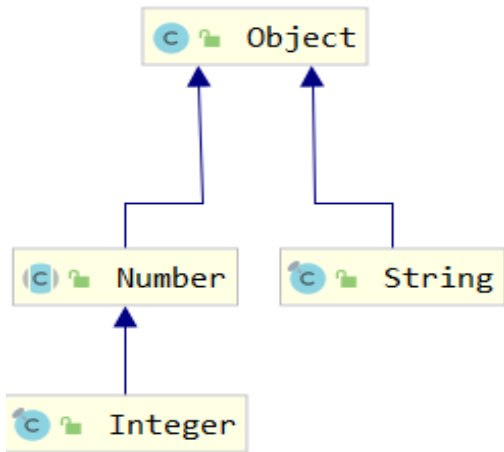
2. 泛型的上限

<? extends 类型> 只能是某一个类型, 及其子类型, 其他类型不支持

需求:

show1方法, 参数只接收元素类型是Number或者其父类型的集合

show2方法, 参数只接收元素类型是Number或者其子类型的集合



泛型的上限格式和含义是什么？

`<? extends 类型>` 只能是本类型或者子类型

总结

泛型的下限格式和含义是什么？

`<? super 类型>` 只能是本类型或者父类型，到顶了只能是Object

目录 Contents

- ◆ 数据结构的认识
- ◆ 栈
- ◆ 队列
- ◆ 链表
- ◆ 树
- ◆ 平衡二叉树
- ◆ 红黑树

数据结构(栈,队列,数组,链表,二叉树)

目标 TARGET

- ◆ 理解数据结构的含义
- ◆ 知道常用的数据结构有哪些
- ◆ 理解栈结构的特点
- ◆ 理解队列结构的特点
- ◆ 理解数组结构的特点
- ◆ 理解链表结构的特点

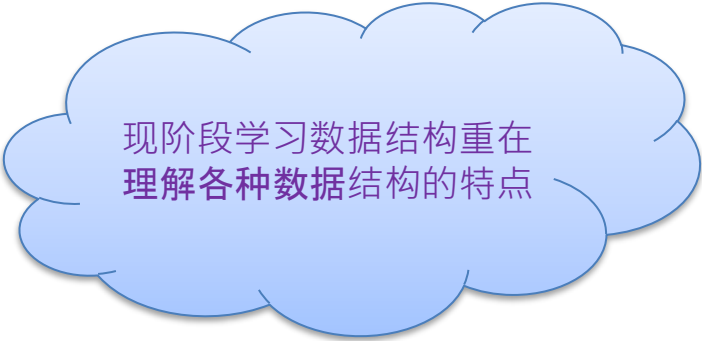
1 数据结构介绍

数据结构是计算机存储、组织数据的方式。通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。数据结构往往同高效的检索算法和索引技术有关。

举例：集合的体系是非常庞大的，不同集合采用的数据结构是不同，导致了不同集合的特点是不同，选择正确的集合去使用，会给我带来更高的代码执行效率

2 常见的数据结构

1. 栈
2. 队列
3. 数组
4. 链表
5. 树（二叉树，二叉平衡树，红黑树）
6. 哈希表（数组+链表+红黑树）



现阶段学习数据结构重在
理解各种数据结构的特点

目标 TARGET

- ◆ 能够说出栈结构的特点

常见数据结构之栈

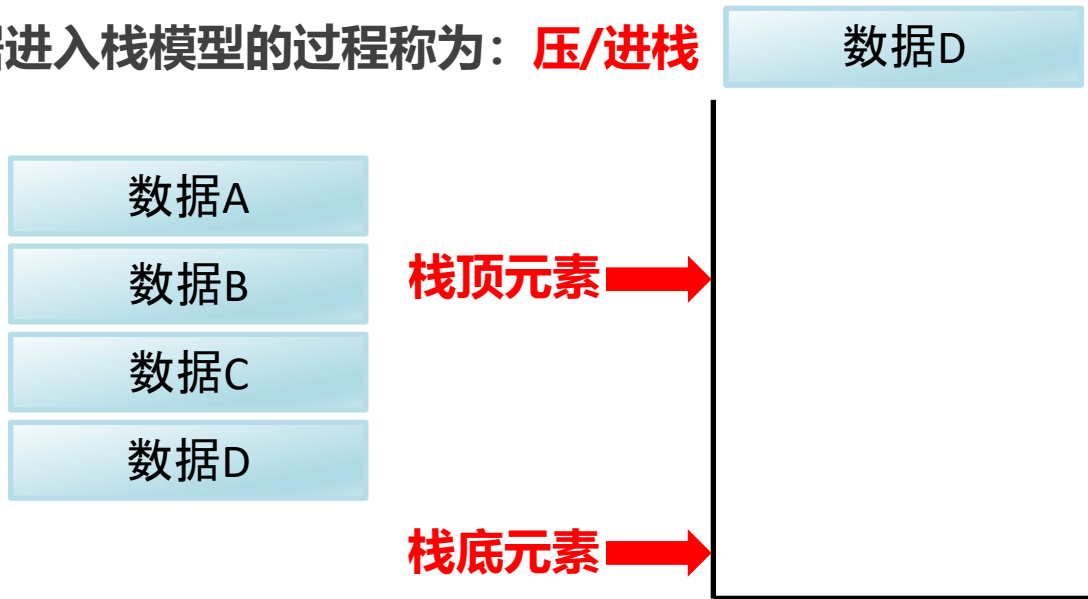


一端开口 栈顶

一端封闭 栈底

常见数据结构之栈

数据进入栈模型的过程称为：**压/进栈**



常见数据结构之栈

数据进入栈模型的过程称为：**压/进栈**

数据离开栈模型的过程称为：**弹/出栈**

栈顶元素 →

数据D

数据C

数据B

栈底元素 →

数据A

常见数据结构之栈

栈结构的特点：先进后出



栈顶元素 →

数据D

数据C

数据B

栈底元素 →

数据A

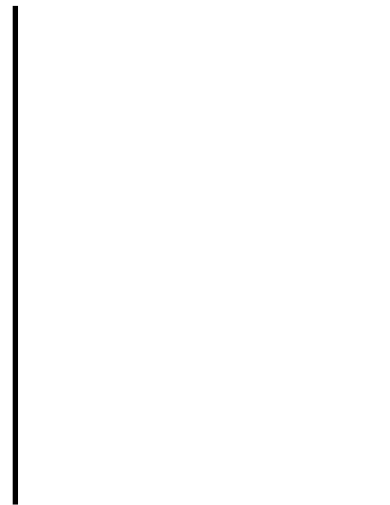
目标 TARGET

- ◆ 能够说出队列结构的特点

常见数据结构之队列



一端开口 后端



一端开头 前端

常见数据结构之队列

数据从**后端**进入队列模型的过程称为：**入队列**



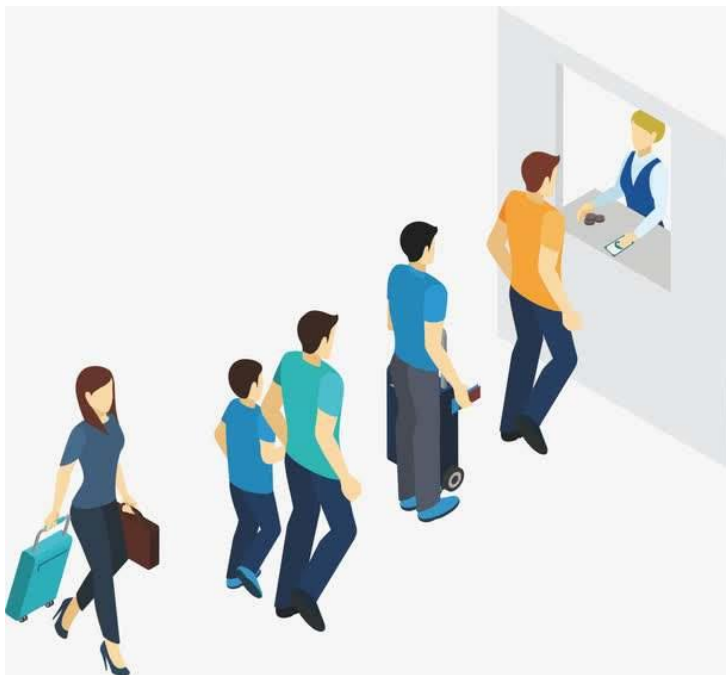
常见数据结构之队列

数据从**后端**进入队列模型的过程称为：**入队列**
数据从**前端**离开队列模型的过程称为：**出队列**



队列特点：先进先出

常见数据结构之队列



入队列方向



后端

数据D

数据C

数据B

数据A

前端

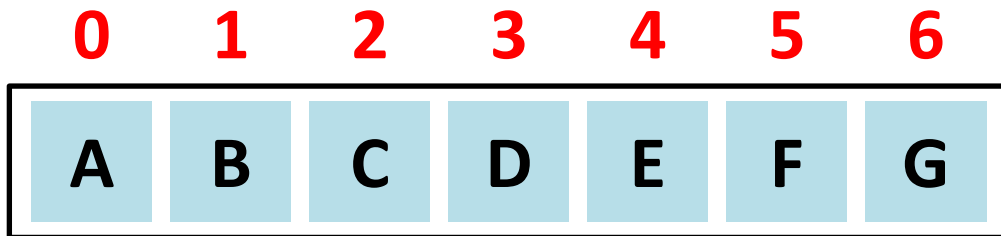
出队列方向



目标 TARGET

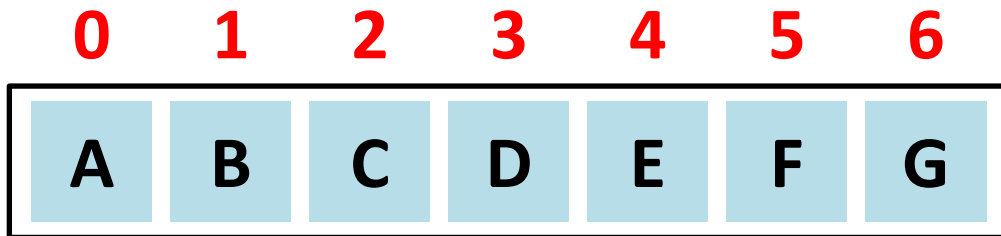
- ◆ 能够说出数组结构的特点

常见数据结构之数组



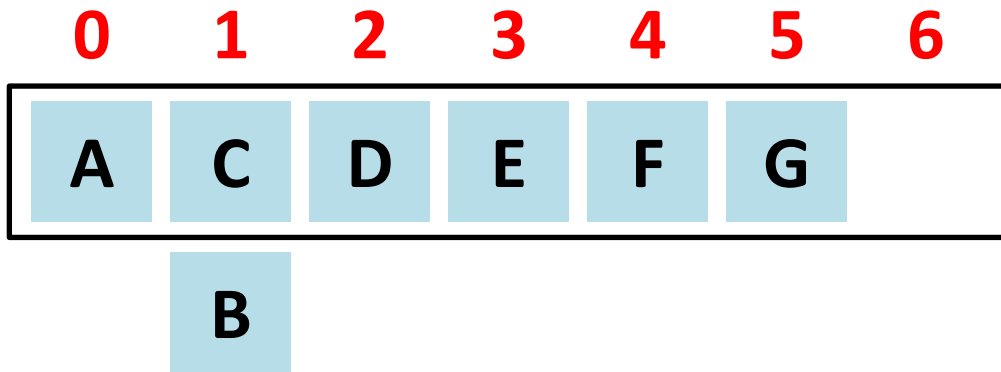
查询数据通过地址值和索引定位，查询任意数据耗时相同，**查询速度快**

常见数据结构之数组



查询数据通过地址值和索引定位，查询任意数据耗时相同，**查询速度快**
删除数据时，要将原始数据删除，同时后面每个数据前移，**删除效率低**

常见数据结构之数组



数组是一种**查询快**，**增删慢**的模型

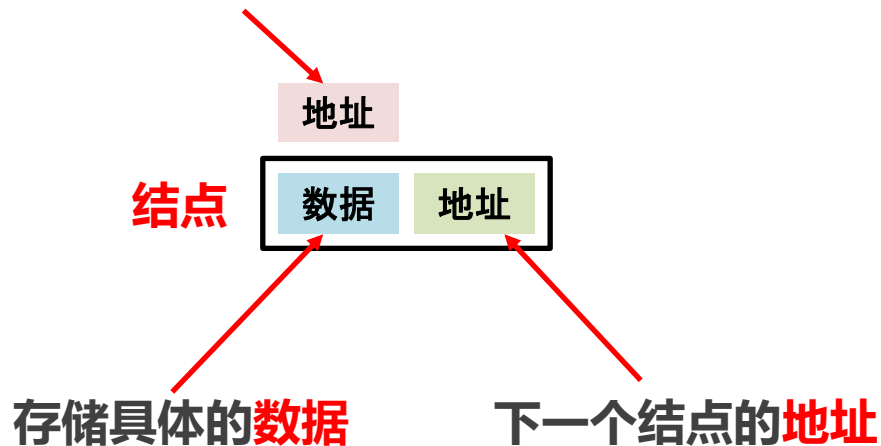
查询数据通过地址值和索引定位，查询任意数据耗时相同，**查询速度快**
删除数据时，要将原始数据删除，同时后面每个数据前移，**删除效率低**
添加数据时，添加位置后的每个数据后移，再添加元素，**添加效率极低**

目标 TARGET

- ◆ 能够说出链表结构的特点

常见数据结构之链表

结点的存储位置 (地址)

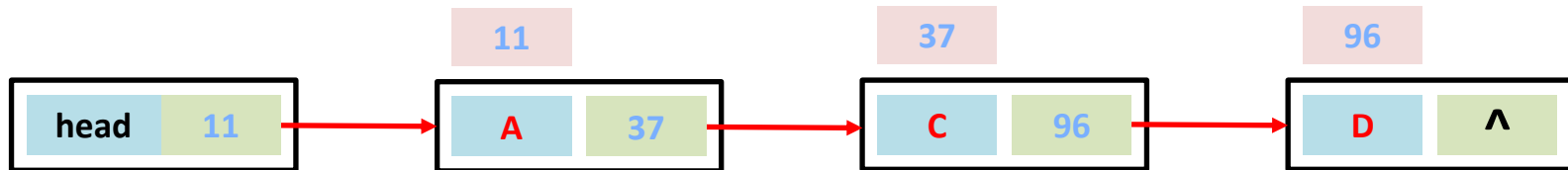


头结点



结点指向空地址
(表示结束)

常见数据结构之链表

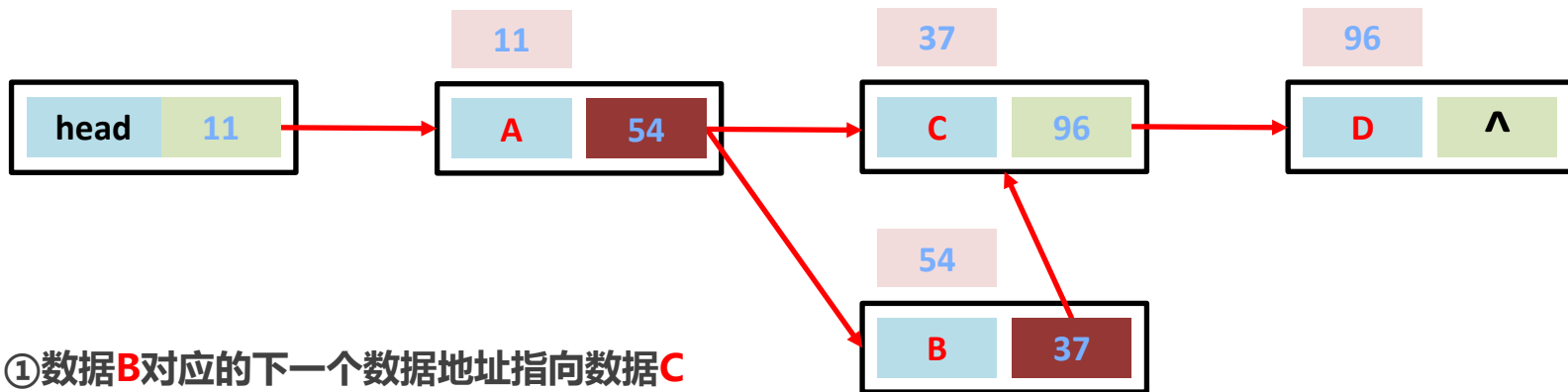


添加一个数据**A**

再添加一个数据**C**

再添加一个数据**D**

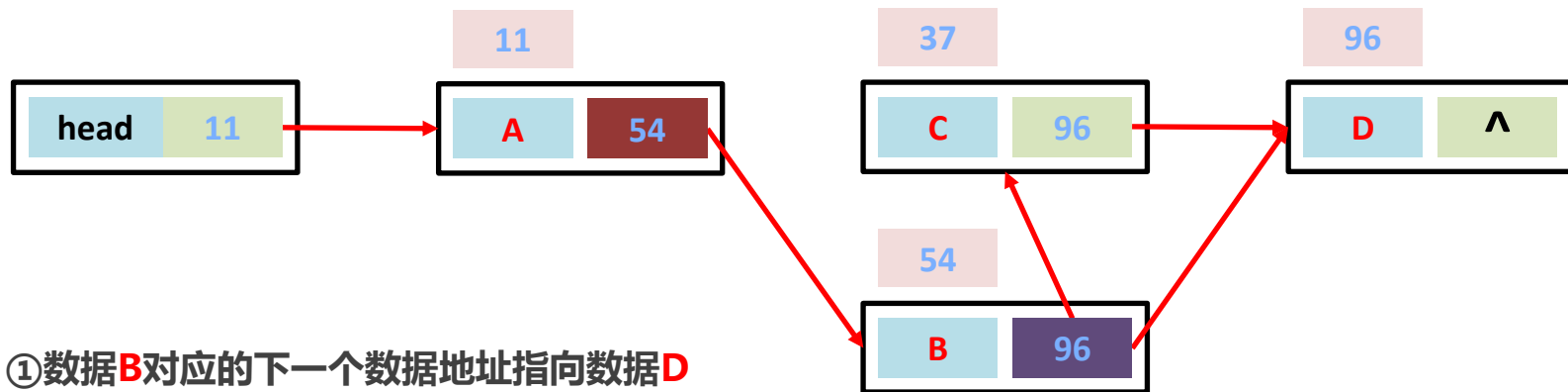
常见数据结构之链表



- ①数据B对应的下一个数据地址指向数据C
- ②数据A对应的下一个数据地址指向数据B

在数据AC之间添加一个数据B

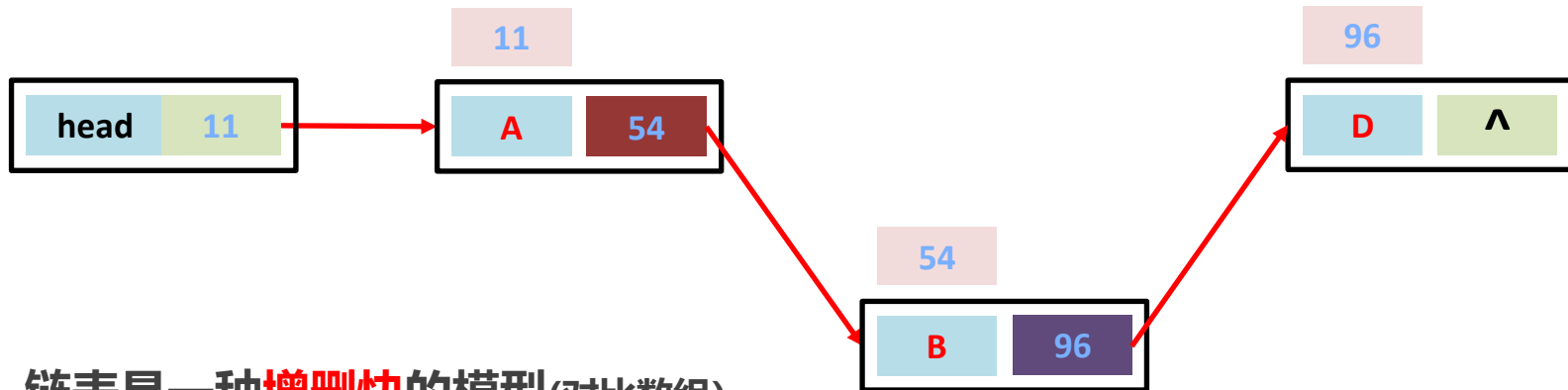
常见数据结构之链表



- ①数据B对应的下一个数据地址指向数据D
- ②数据C删除

在数据AC之间添加一个数据B
删除数据BD之间的数据C

常见数据结构之链表



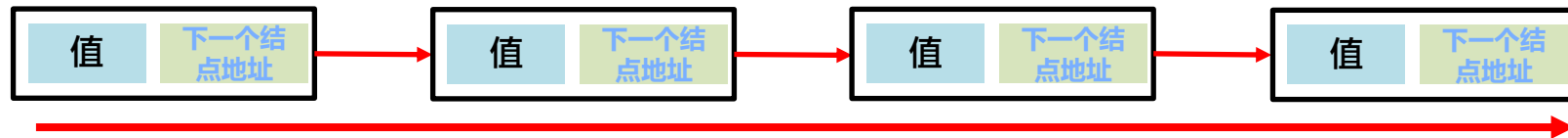
链表是一种**增删快**的模型(对比数组)

链表是一种**查询慢**的模型(对比数组)

查询数据**D**是否存在, 必须从头(**head**)开始查询
查询第**N**个数据, 必须从头(**head**)开始查询

常见数据结构之链表

单
向
链
表



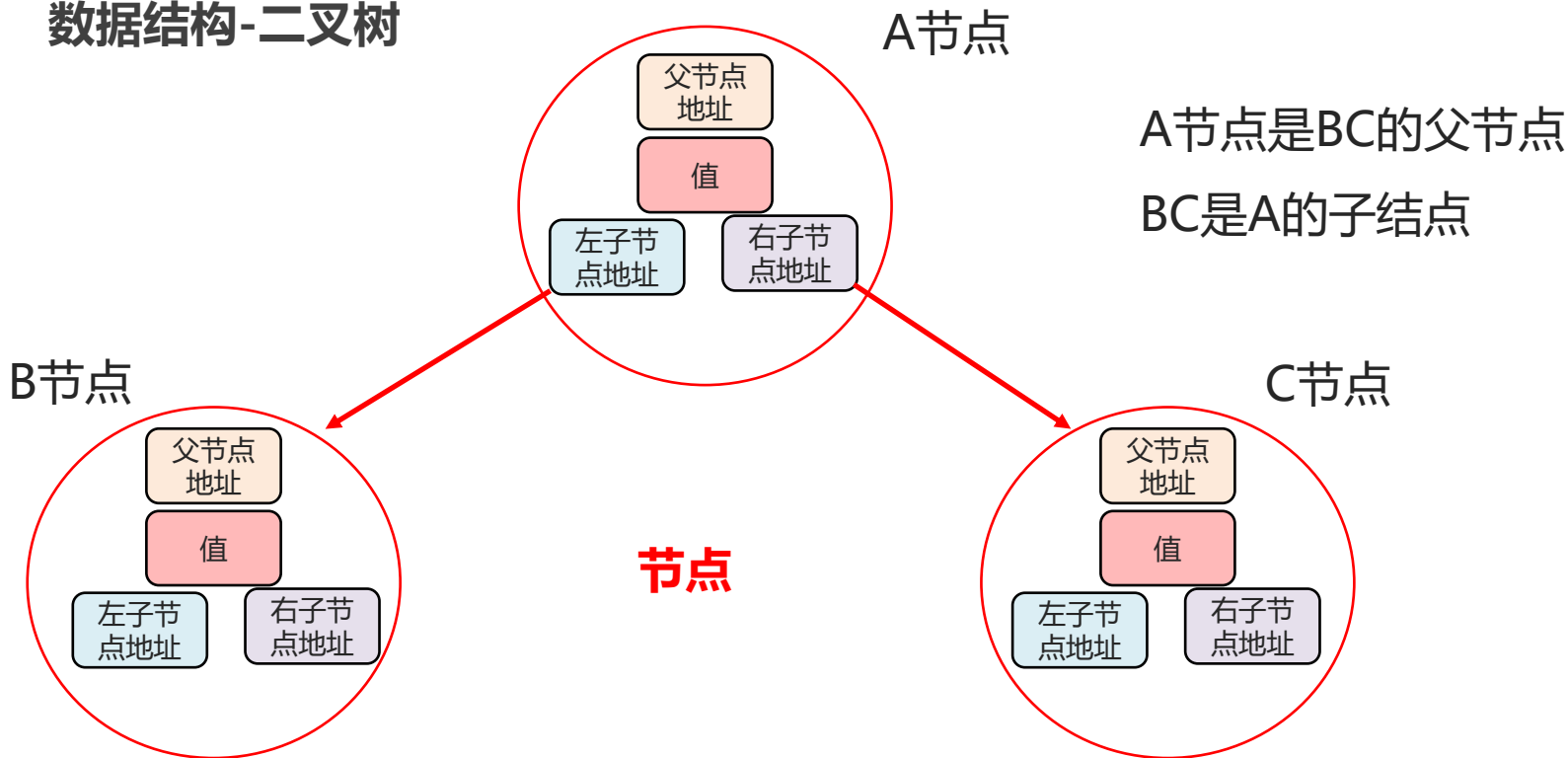
双
向
链
表



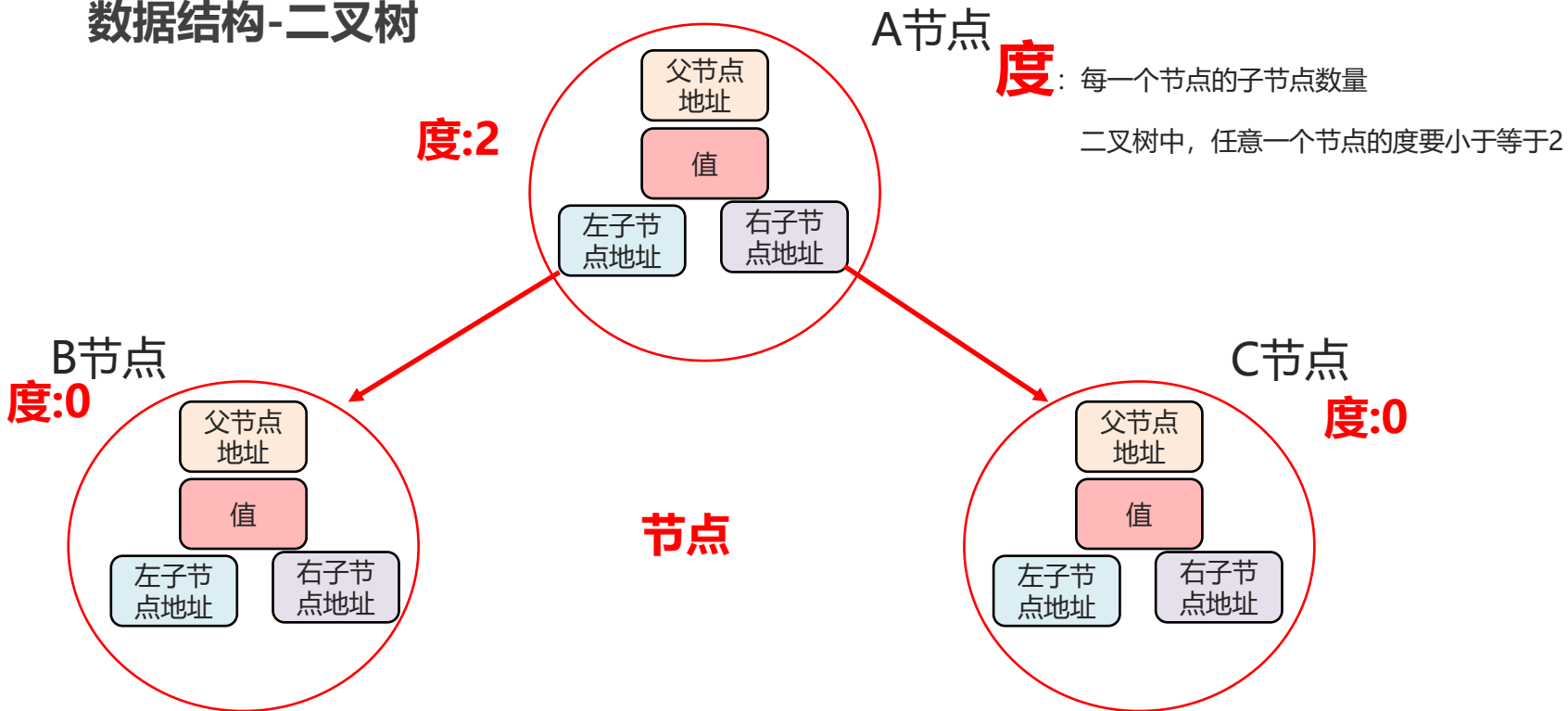
目标 TARGET

- ◆ 理解树结构相关术语
- ◆ 理解二叉查找树的特点
- ◆ 理解二叉平衡树的特点
- ◆ 理解红黑树的特点

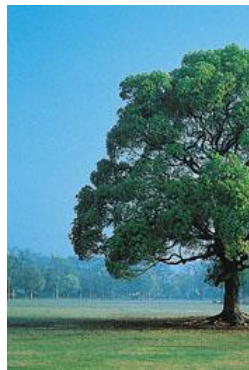
数据结构-二叉树



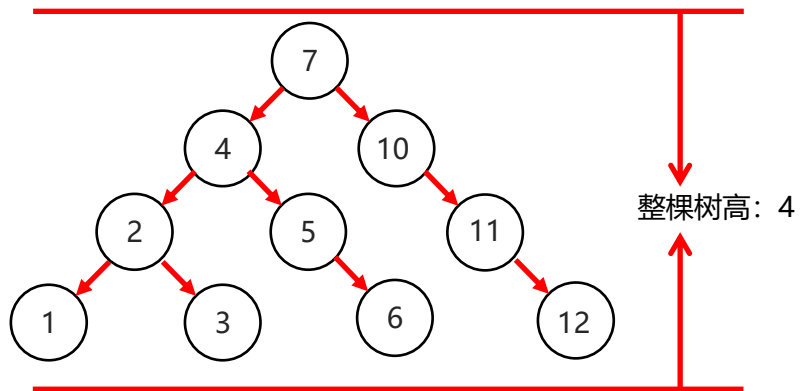
数据结构-二叉树



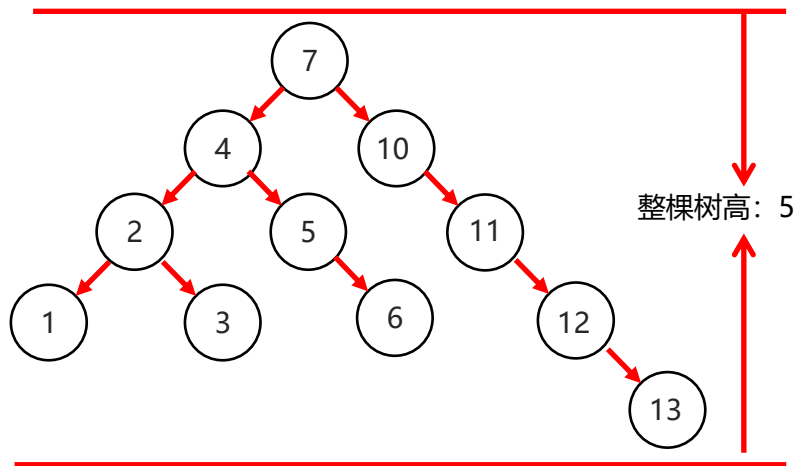
数据



数据结构-二叉树

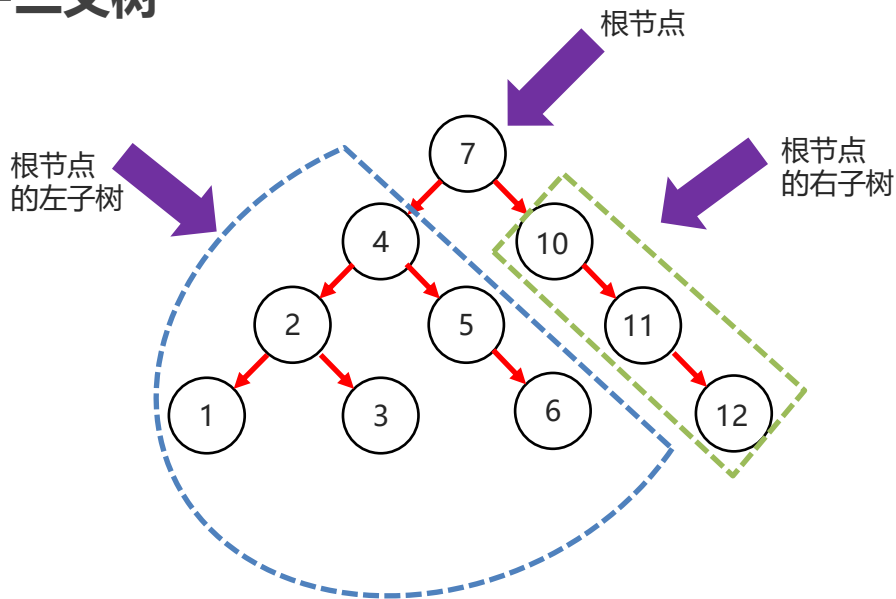


数据结构-二叉树



二叉树：每个节点最多有两个子节点

数据结构-二叉树



最顶层的为：根节点

4节点为7节点的左子节点

10节点为7节点的右子节点

蓝色虚线：根节点的左子树

根节点的左子树高：3

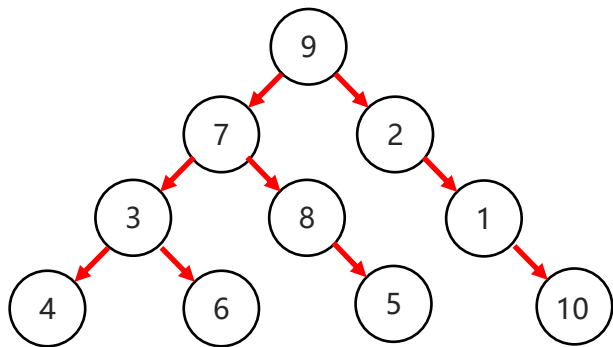
绿色虚线：根节点的右子树

根节点的右子树高：3

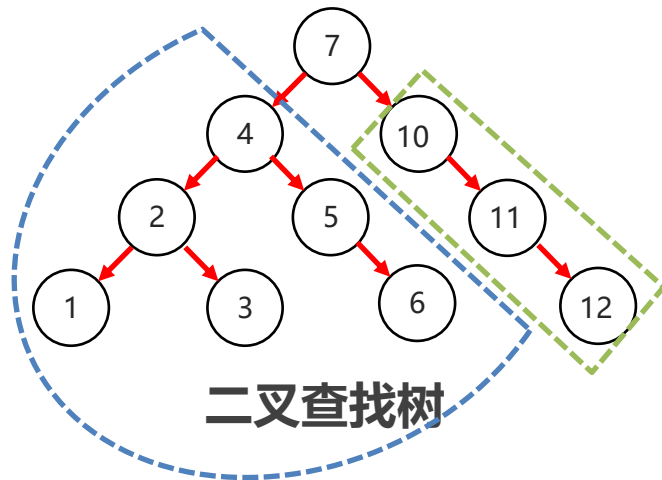
目标 TARGET

- ◆ 理解二叉查找树的特点
- ◆ 理解二叉查找树的添加特点

二叉查找树



普通的二叉树



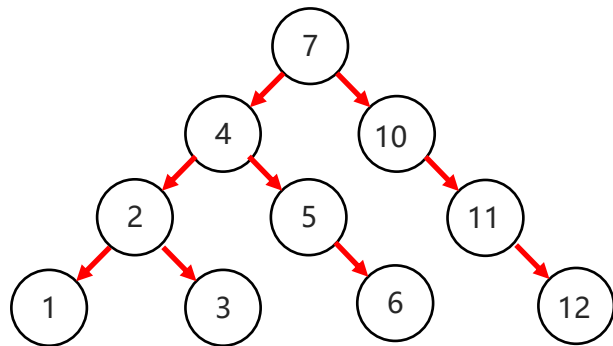
二叉查找树

二叉查找树

二叉查找树，又称二叉排序树或者二叉搜索树。

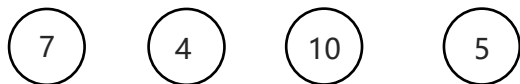
特点：

- 1, 每一个节点上最多有两个子节点
- 2, 每个节点的左子节点比当前节点小 右子节点比当前节点大

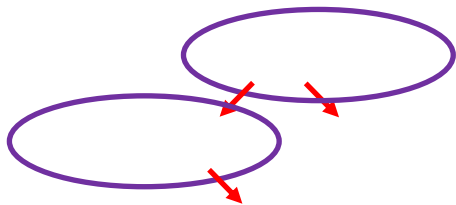
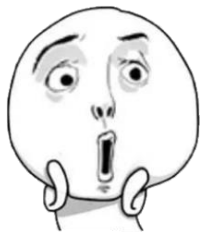


二叉查找树

二叉树查找树添节点



将上面的节点按照二叉查找树的规则存入

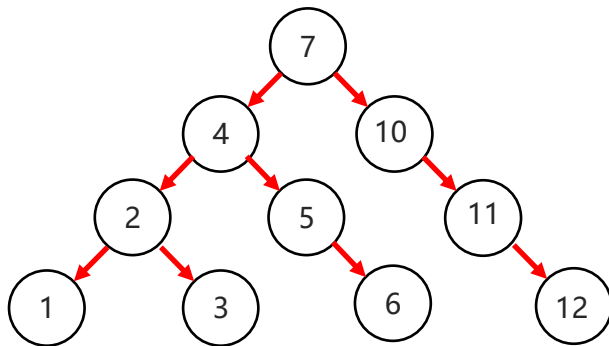


规则:

小的存左边

大的存右边

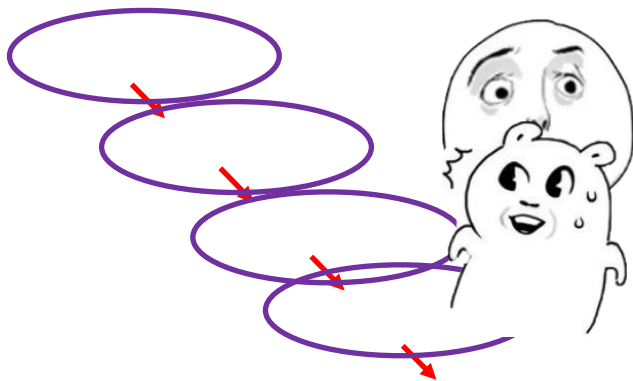
一样的不存



二叉树查找树添节点



将上面的节点按照二叉查找树的规则存入

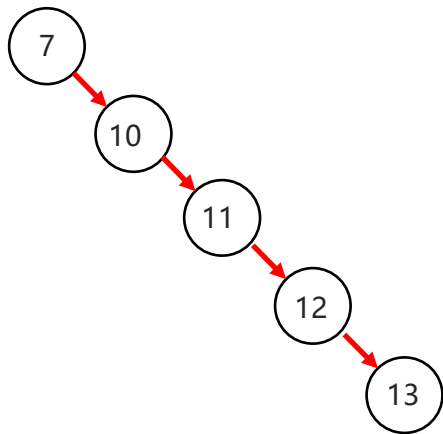


目标 TARGET

- ◆ 理解平衡二叉树的特点
- ◆ 理解平衡二叉树的旋转

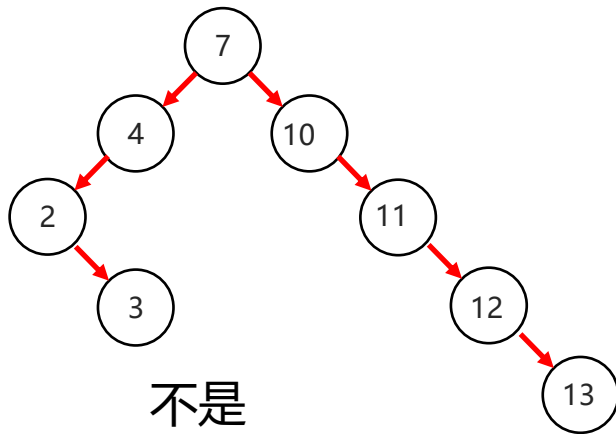
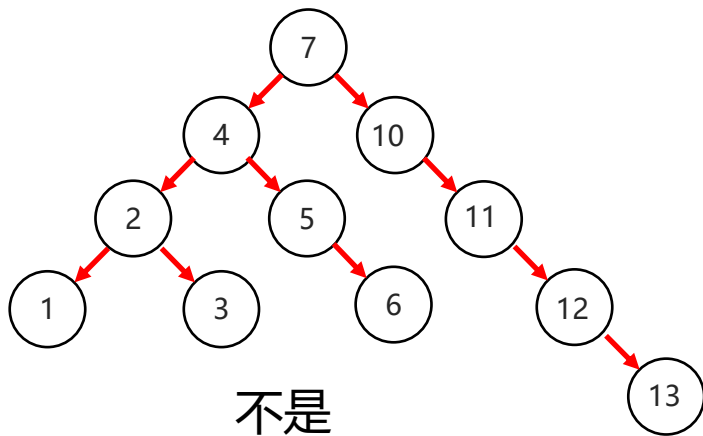
■ 二叉树-平衡二叉树

数据结构-平衡二叉树



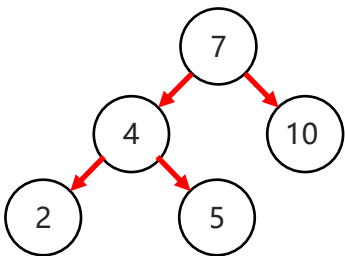
数据结构-平衡二叉树

- 二叉树左右两个子树的高度差不超过1
- 任意节点的左右两个子树都是一颗平衡二叉树

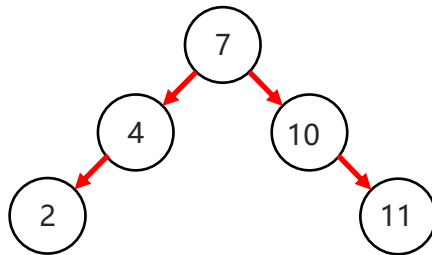


数据结构-平衡二叉树

- 二叉树左右两个子树的高度差不超过1
- 任意节点的左右两个子树都是一颗平衡二叉树



是的



是的

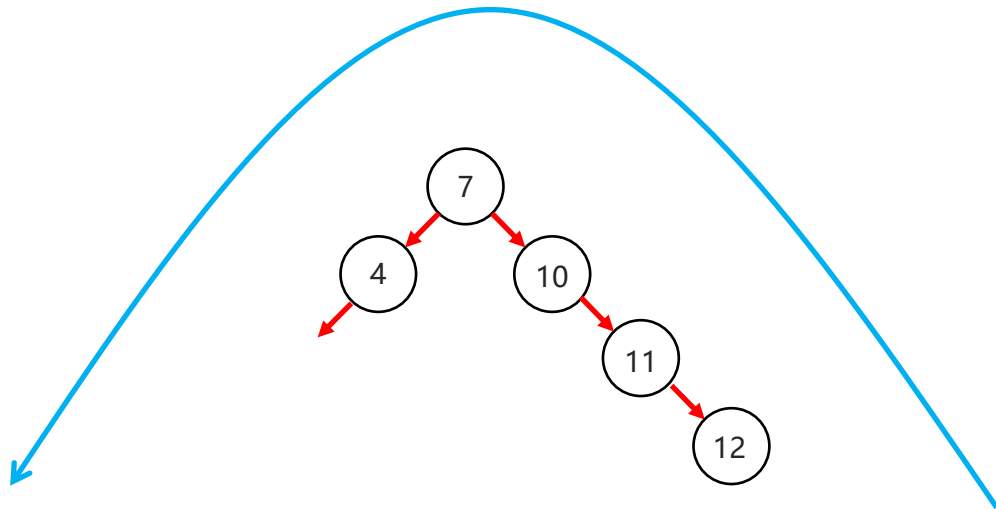
平衡二叉树-旋转

- 左旋
- 右旋

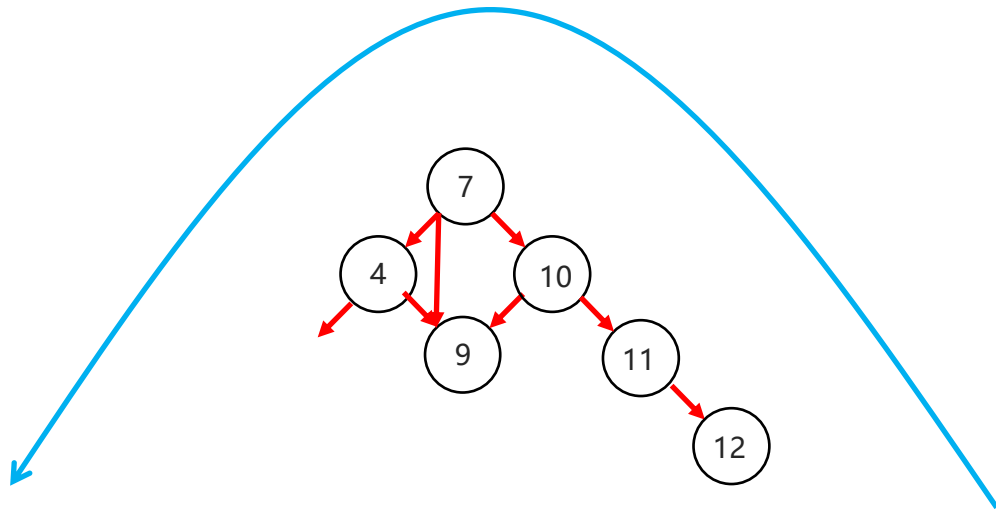
触发时机:

当添加一个节点之后, 该树不再是一颗平衡二叉树

平衡二叉树-左旋

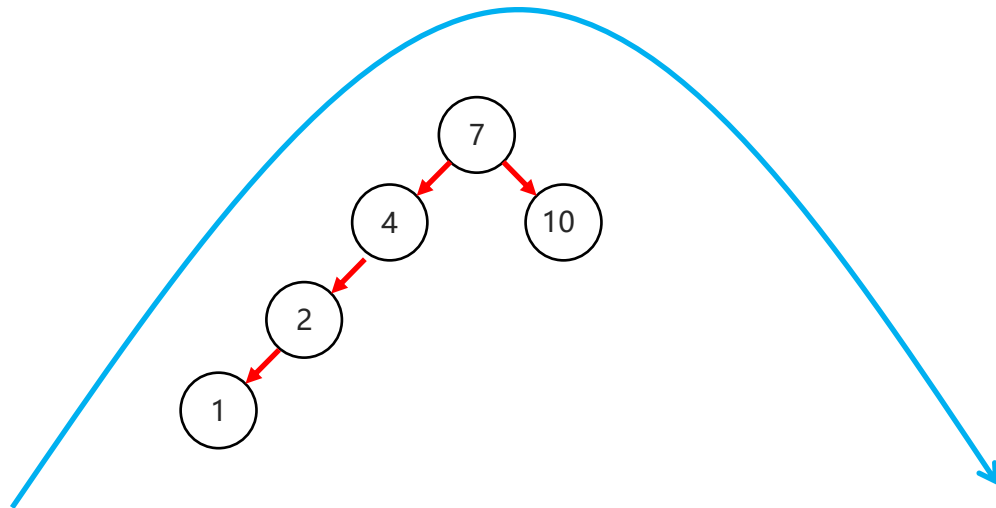


平衡二叉树-左旋

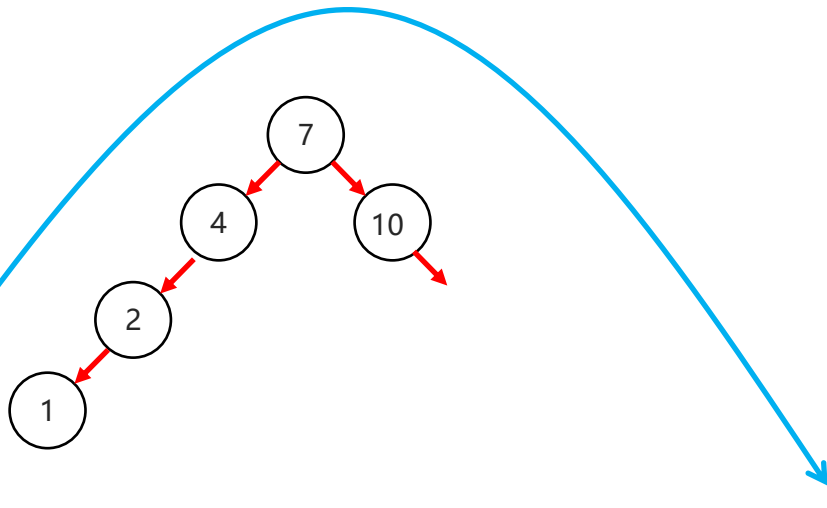


左旋：就是将根节点的右侧往左拉，原先的右子节点变成新的父节点，并把多余的左子节点出让，给已经降级的根节点当右子节点

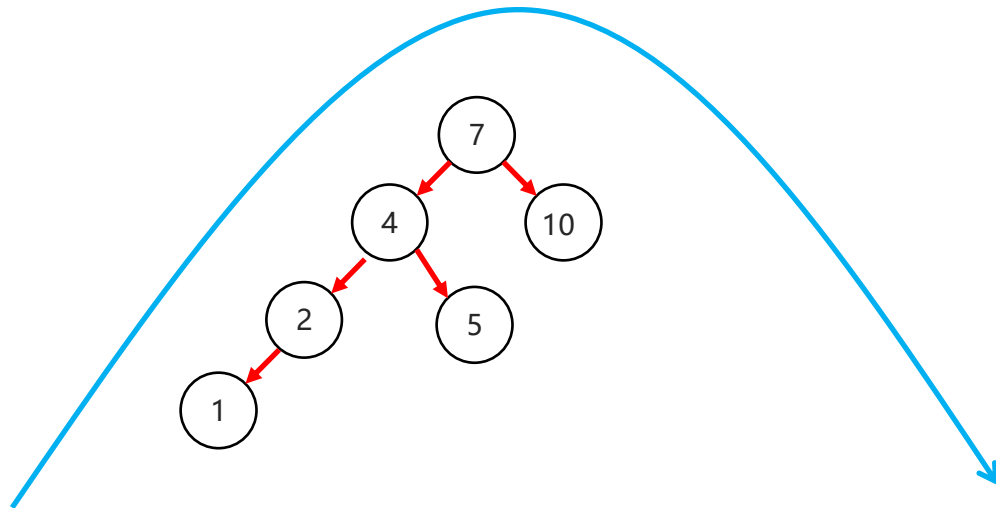
平衡二叉树-右旋



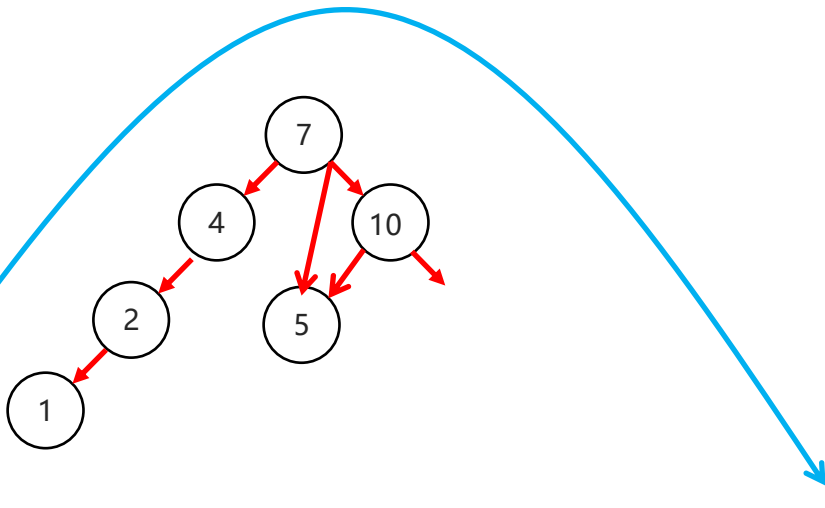
平衡二叉树-右旋



平衡二叉树-右旋

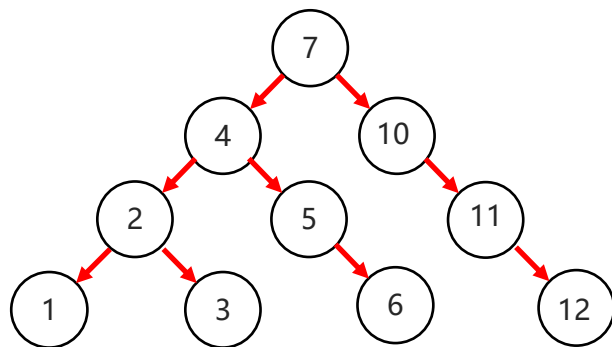


平衡二叉树-右旋



右旋：将根节点的左侧往右拉，左子节点变成了新的父节点，并把多余的右子节点出让，给已经降级根节点当左子节点

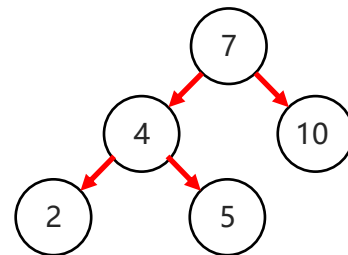
小结



二叉查找树



利用左旋和右旋机制保证树的平衡



平衡二叉树



二叉树-平衡二叉树

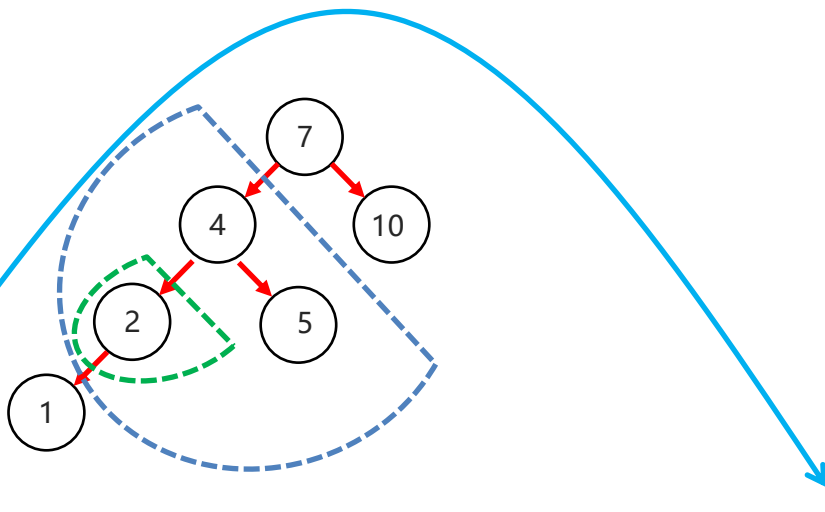
平衡二叉树-旋转的四种情况

- 左左
- 左右
- 右右
- 右左

平衡二叉树-左左

- 左左

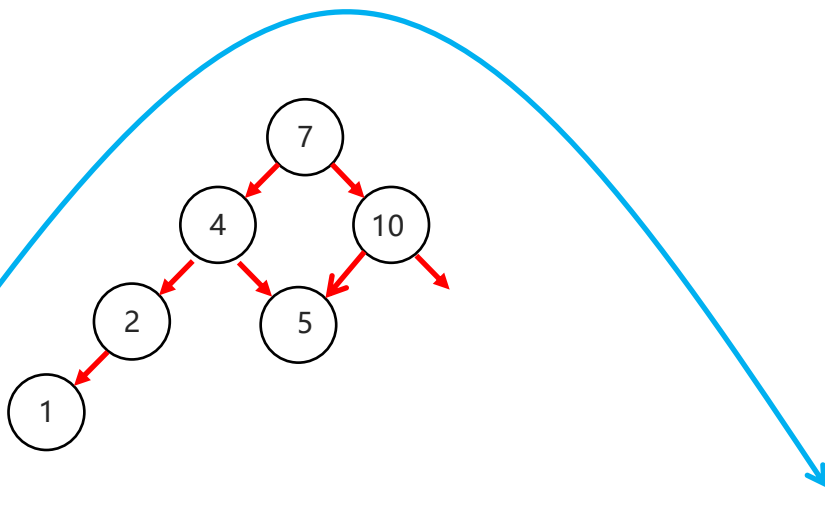
当根节点左子树的左子树有节点插入，导致二叉树不平衡



平衡二叉树-左左

- 左左

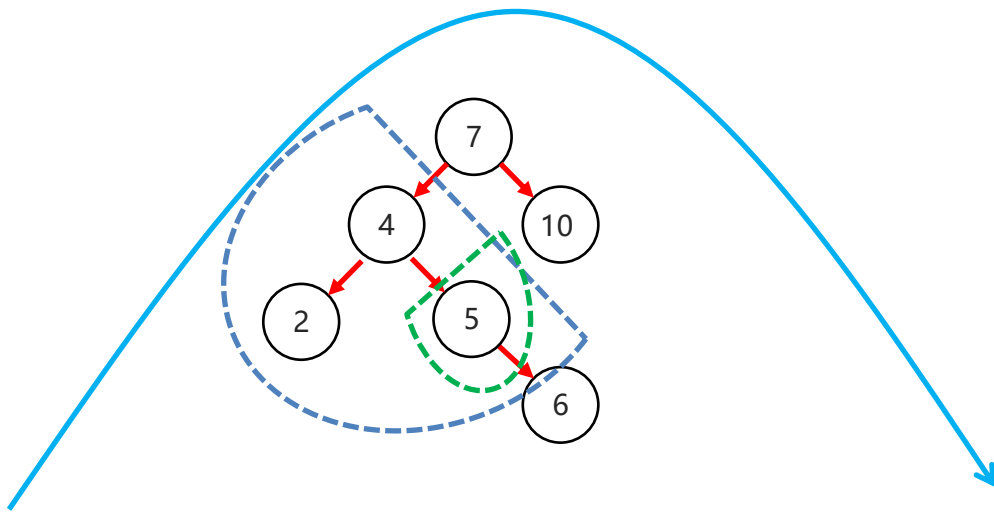
当根节点左子树的左子树有节点插入，导致二叉树不平衡



平衡二叉树-左右

- 左右

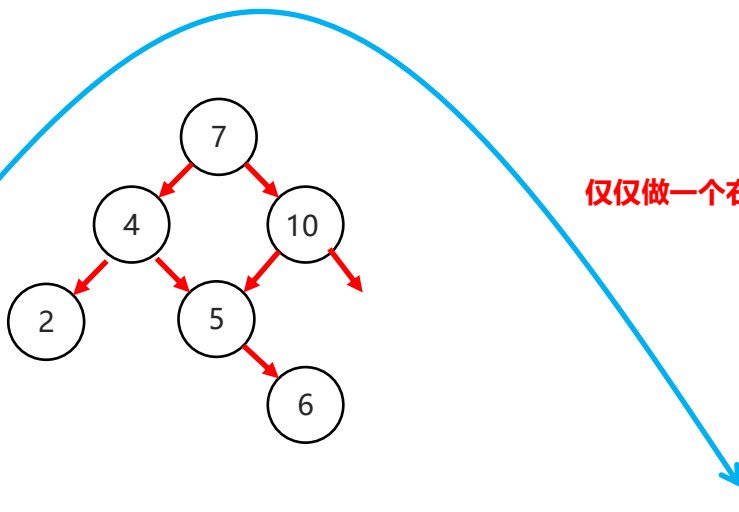
当根节点左子树的右子树有节点插入，导致二叉树不平衡



平衡二叉树-左右

- 左右

当根节点左子树的右子树有节点插入，导致二叉树不平衡

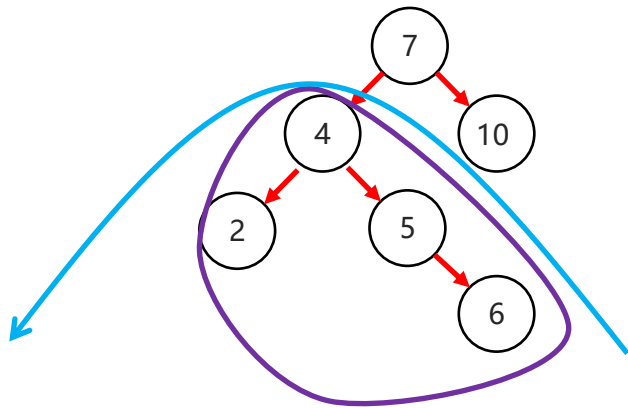


仅仅做一个右旋还是不行

平衡二叉树-左右

- 左右

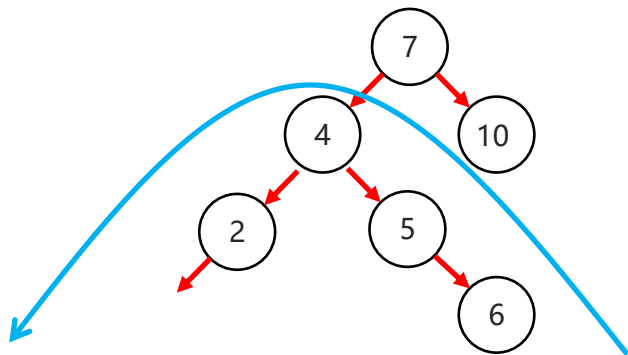
当根节点左子树的右子树有节点插入，导致二叉树不平衡



平衡二叉树-左右

- 左右

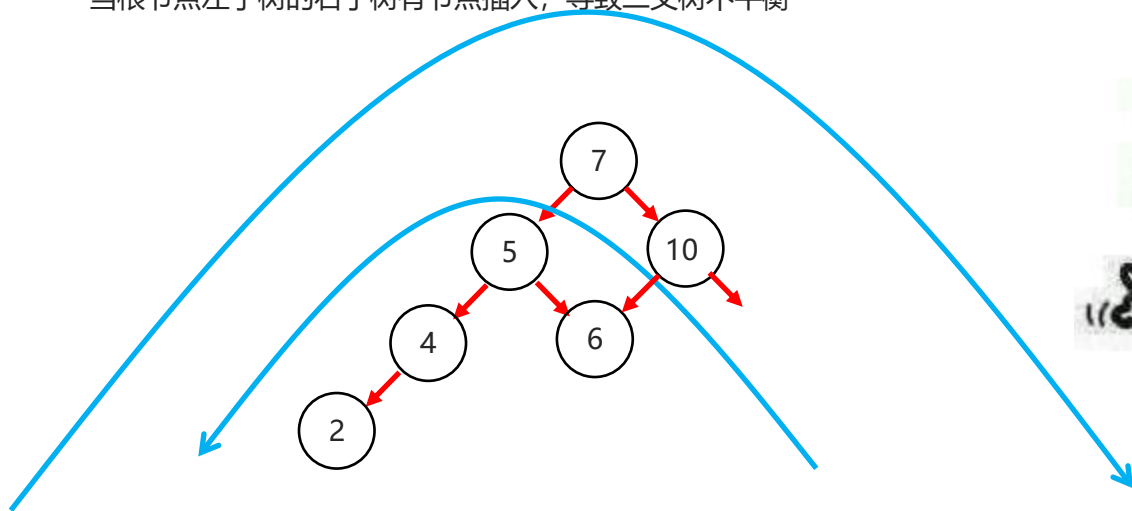
当根节点左子树的右子树有节点插入，导致二叉树不平衡



平衡二叉树-左右

- 左右

当根节点左子树的右子树有节点插入，导致二叉树不平衡



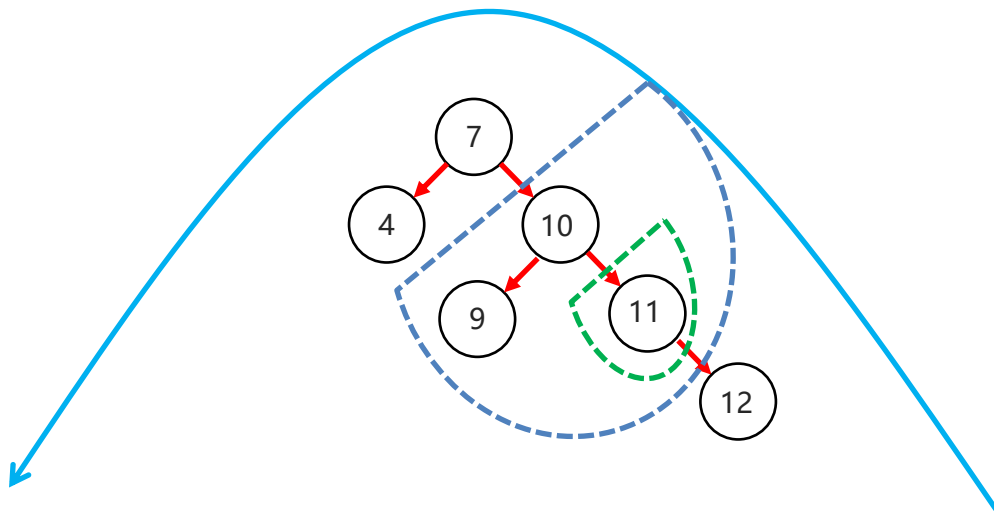
完美!



平衡二叉树-右右

- 右右

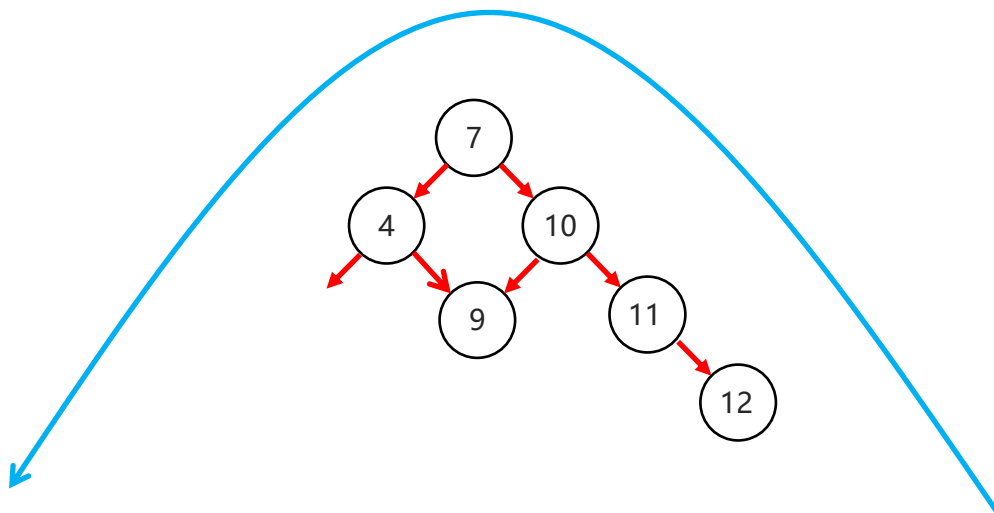
当根节点右子树的右子树有节点插入，导致二叉树不平衡



平衡二叉树-右右

- 右右

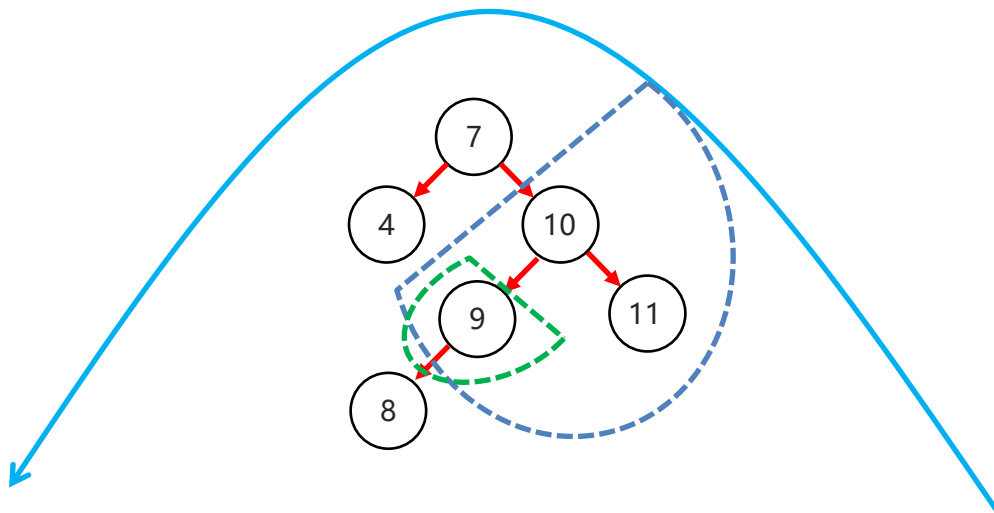
当根节点右子树的右子树有节点插入，导致二叉树不平衡



平衡二叉树-右左

- 右左

当根节点右子树的左子树有节点插入，导致二叉树不平衡

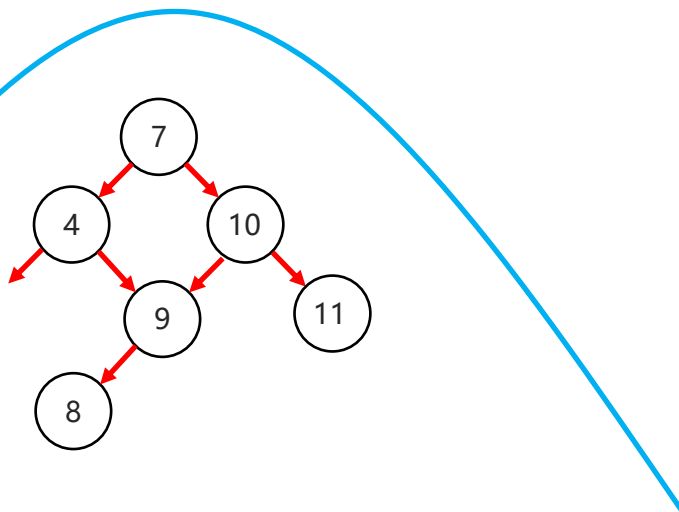


平衡二叉树-右左

- 右左

当根节点右子树的左子树有节点插入，导致二叉树不平衡

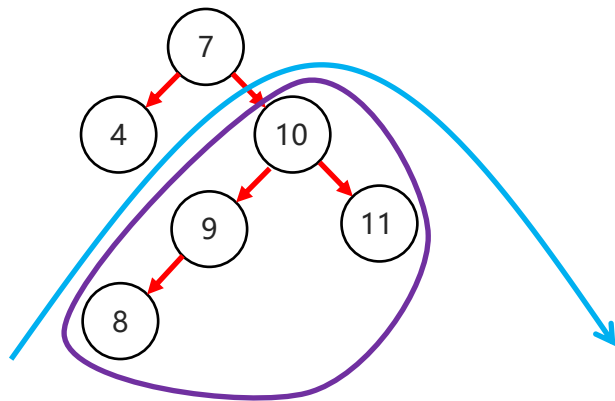
仅仅做一个左旋还是不行



平衡二叉树-右左

- 右左

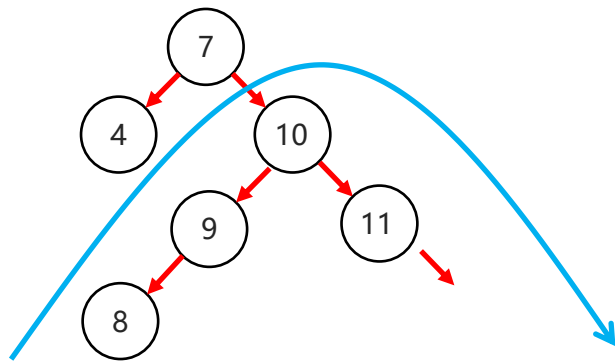
当根节点右子树的左子树有节点插入，导致二叉树不平衡



平衡二叉树-右左

- 右左

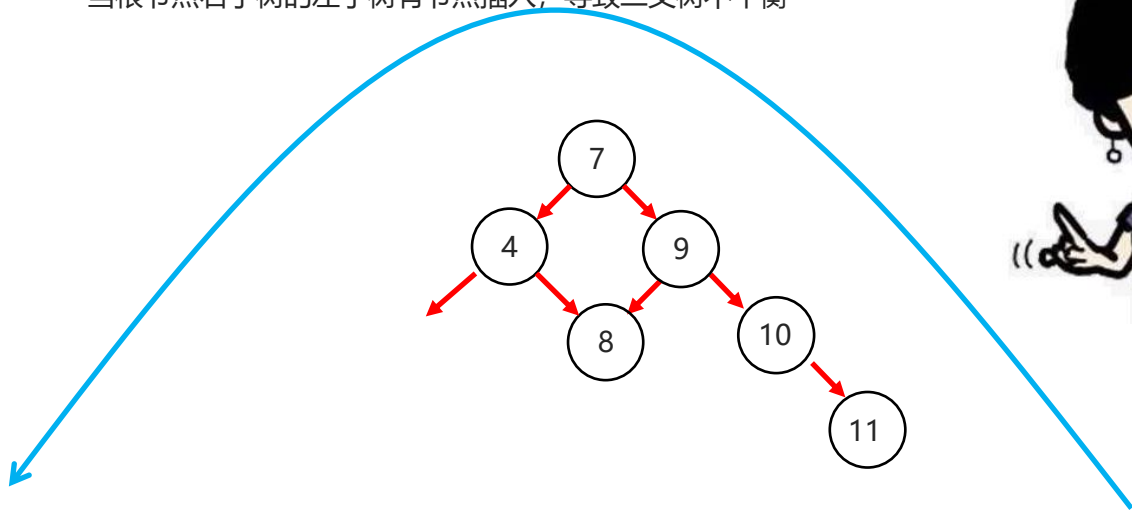
当根节点右子树的左子树有节点插入，导致二叉树不平衡



平衡二叉树-右左

- 右左

当根节点右子树的左子树有节点插入，导致二叉树不平衡



完美!



目标 TARGET

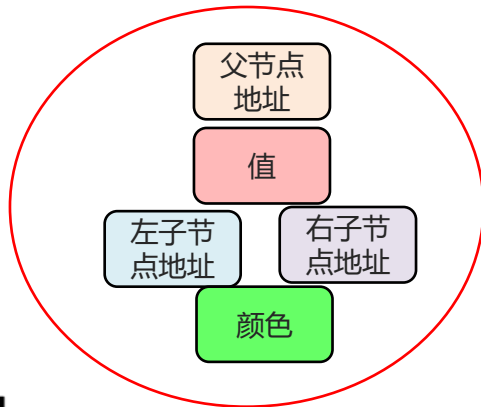
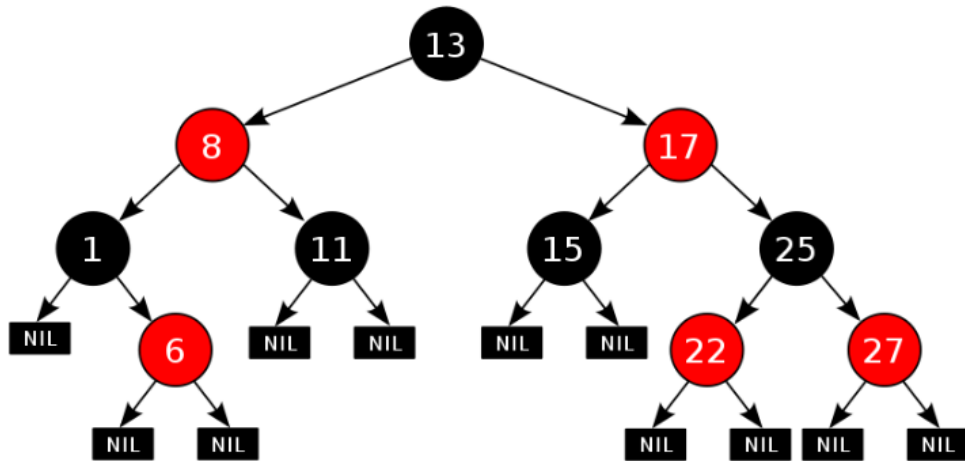
- ◆ 红黑树的特点
- ◆ 红黑的规则

红黑树

- 平衡二叉B树
- 每一个节点可以是红或者黑
- 红黑树不是高度平衡的，它的平衡是通过“自己的红黑规则”进行实现的

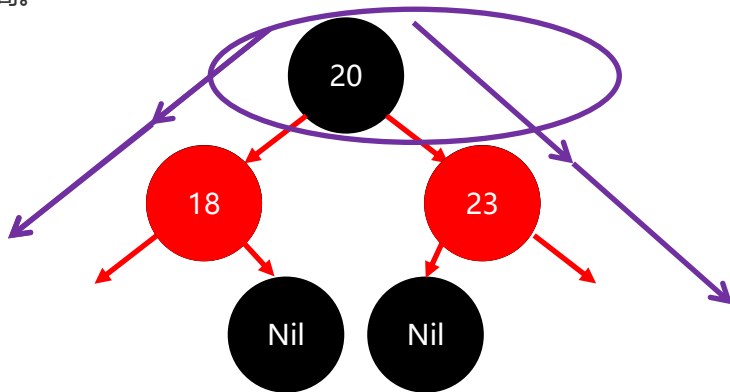
红黑规则

1. 每一个节点或是红色的，或者是黑色的。
2. 根节点必须是黑色
3. 如果一个节点没有子节点或者父节点，则该节点相应的指针属性值为Nil，这些Nil视为叶节点，每个叶节点(Nil)是黑色的；
4. 如果某一个节点是红色，那么它的子节点必须是黑色(不能出现两个红色节点相连的情况)
5. 对每一个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点；



添加节点

- 添加的节点的颜色，可以是红色的，也可以是黑色的。
- 红色效率高。

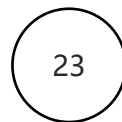
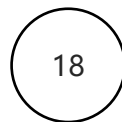
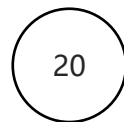
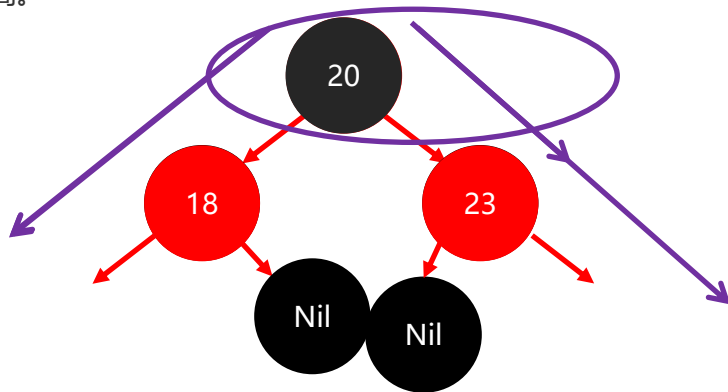


添加三个元素，
一共需要调整两次

对每一个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点；

添加节点

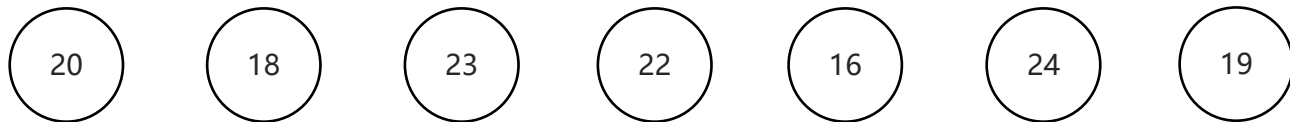
- 添加的节点的颜色，可以是红色的，也可以是黑色的。
- 红色效率高。



添加三个元素，
一共需要调整一次
所以，添加节点时，
默认为红色，效率高。

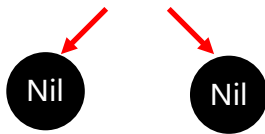
根节点必须是黑色

二叉树-红黑树



红黑树

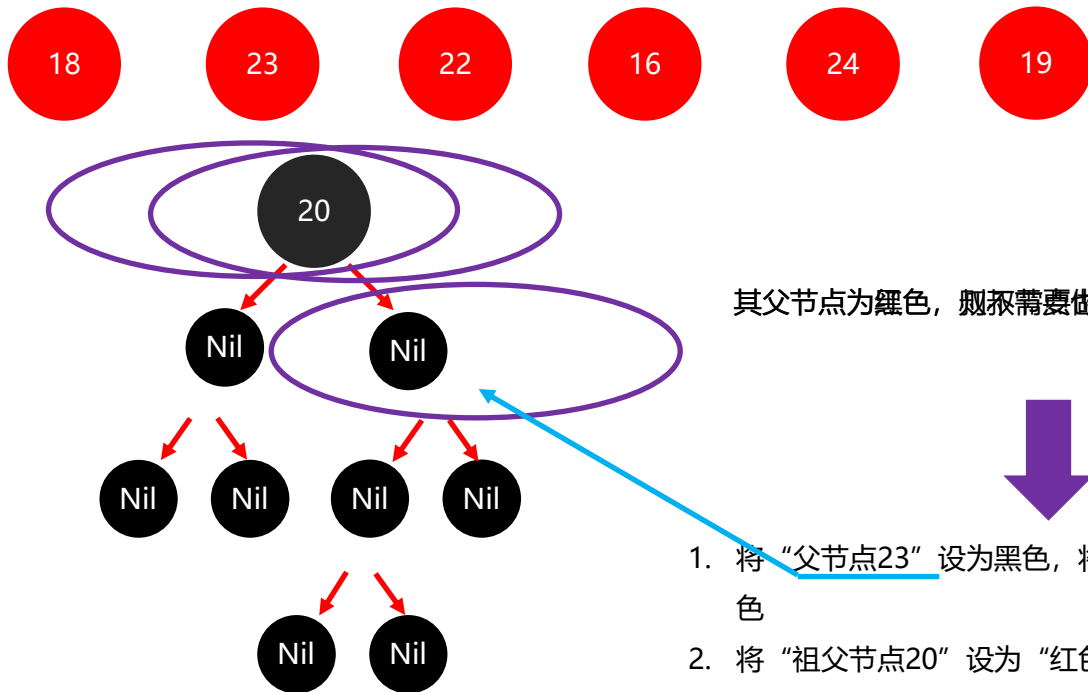
- 每一个节点是红色或者黑色，根节点必须是黑色
- 每个叶节点(Nil)是黑色的；
- 不能出现两个红色节点相连
- 对每一个节点，到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点；



当添加的节点为根节点时，
直接变成黑色就可以了

红黑树

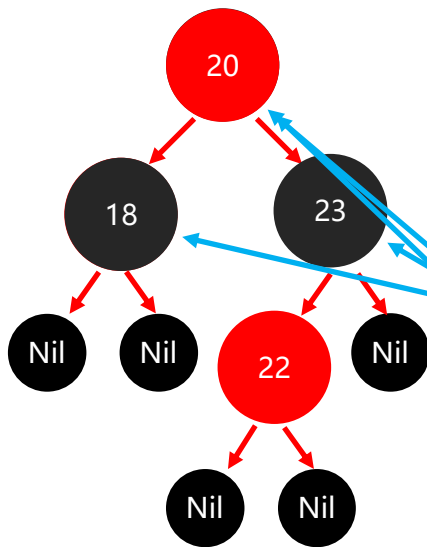
- 每一个节点是红色或者黑色，根节点必须是黑色
- 每个叶节点(Nil)是黑色的；
- 不能出现两个红色节点相连
- 对每一个节点，到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点；



1. 将“父节点23”设为黑色，将“叔叔节点18”设为黑色
2. 将“祖父节点20”设为“红色”。
3. 如果祖父节点为根节点，则将根节点再次变成黑色。

红黑树

- 每一个节点是红色或者黑色，根节点必须是黑色
- 每个叶节点(Nil)是黑色的；
- 不能出现两个红色节点相连
- 对每一个节点，到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点；



其父节点为红色，叔叔节点也是红色



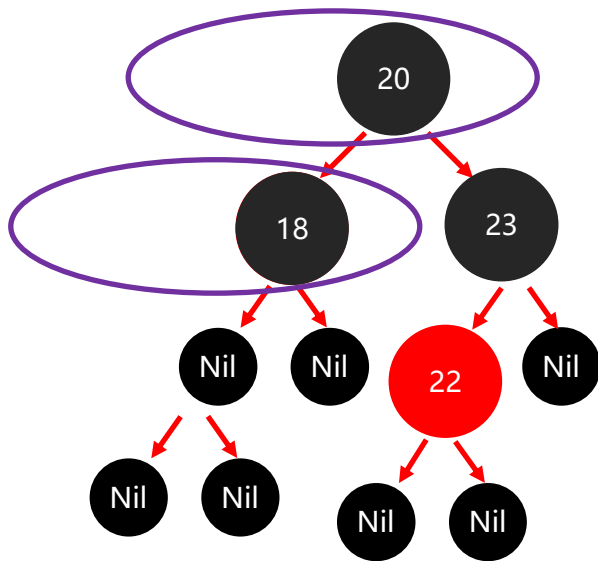
1. 将“父节点23”设为黑色，将“叔叔节点18”设为黑色
2. 将“祖父节点20”设为“红色”。
3. 如果祖父节点为根节点，则将根节点再次变成黑色。

- 每一个节点是红色或者黑色，根节点必须是黑色
- 每个叶节点(Nil)是黑色的；
- 不能出现两个红色节点相连
- 对每一个节点，到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点；

16

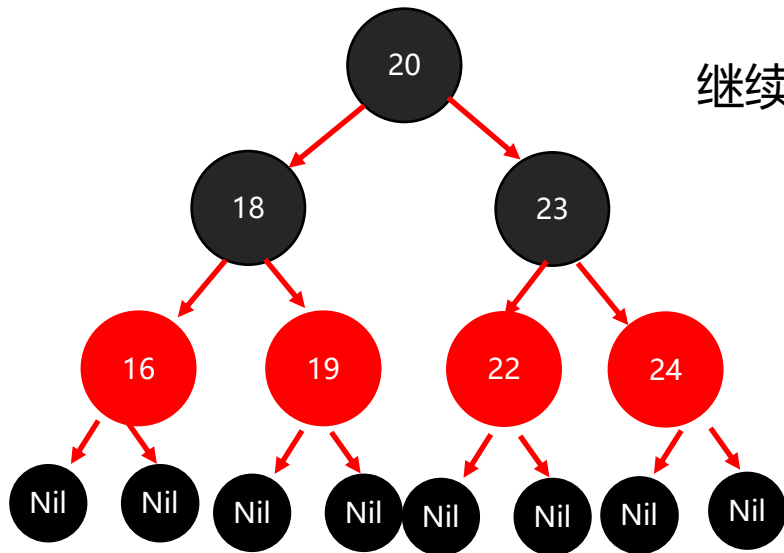
24

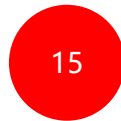
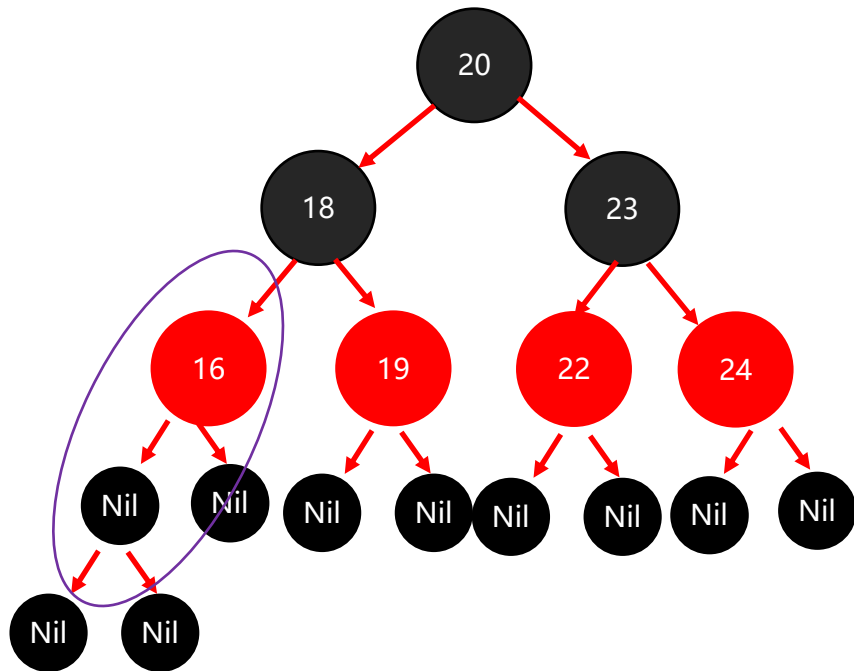
19



其父节点为黑色，则不需要做任何操作。

继续添加数据

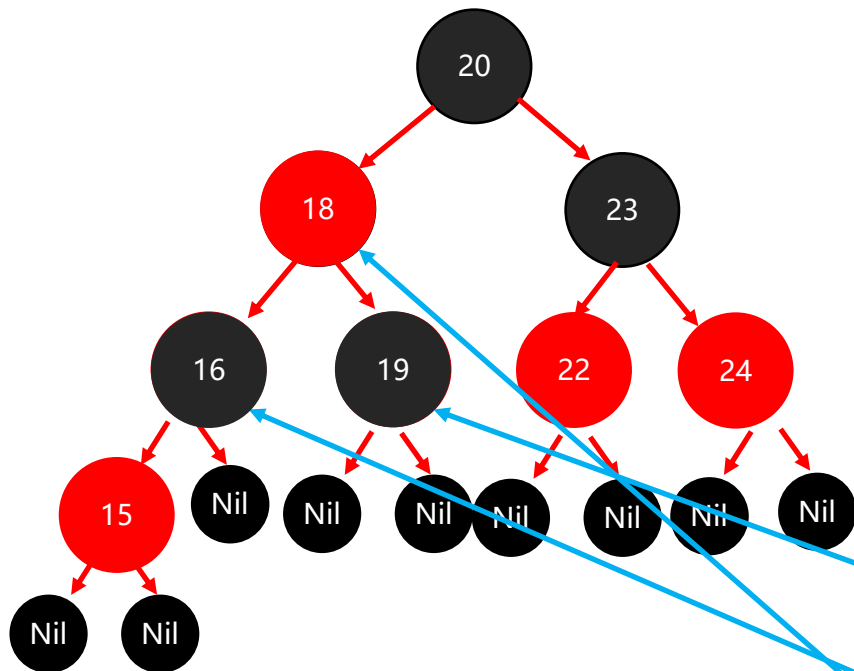




其父节点为红色，叔叔节点也是红色

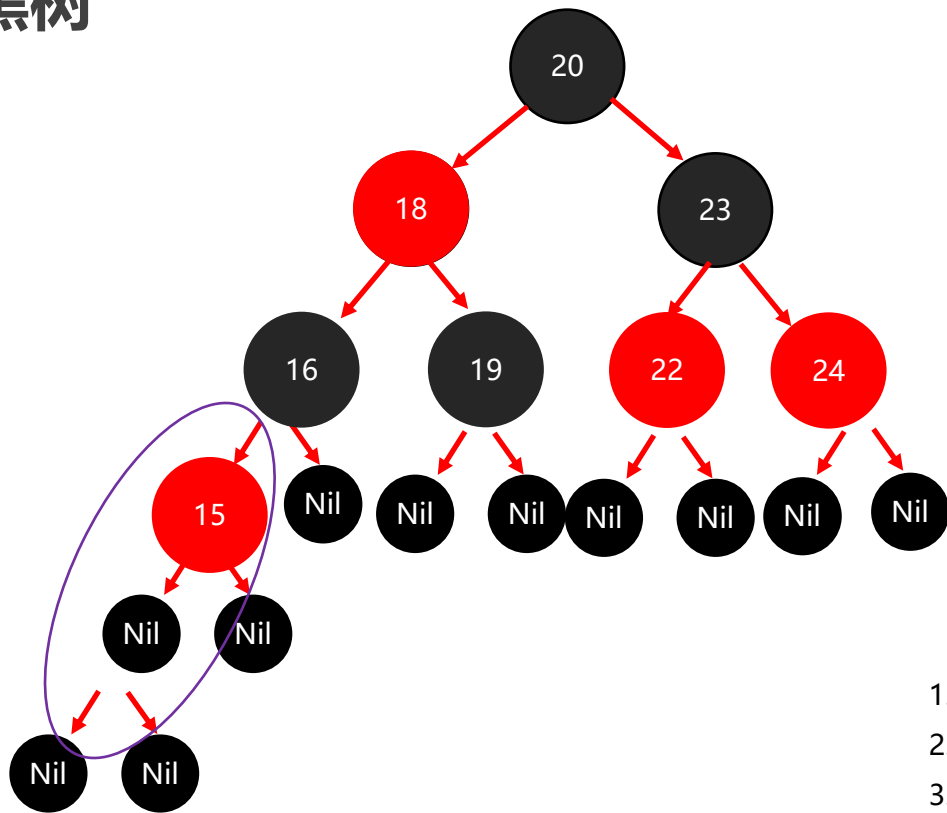


1. 将“父节点16”设为黑色，将“叔叔节点19”设为黑色
2. 将“祖父节点18”设为“红色”。
3. 如果祖父节点为根节点，则将根节点再次变成黑色。



其父节点为红色，叔叔节点也是红色

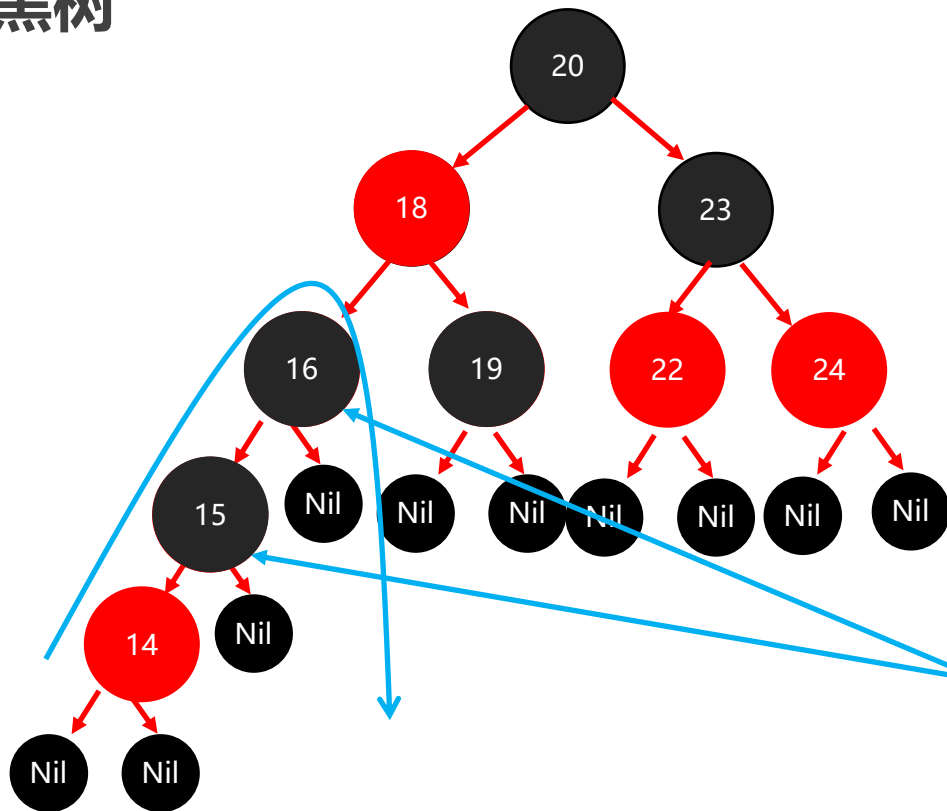
1. 将“父节点16”设为黑色，将“叔叔节点19”设为黑色
2. 将“祖父节点18”设为“红色”。
3. 如果祖父节点为根节点，则将根节点再次变成黑色。



其父节点为红色，叔叔节点也是红色



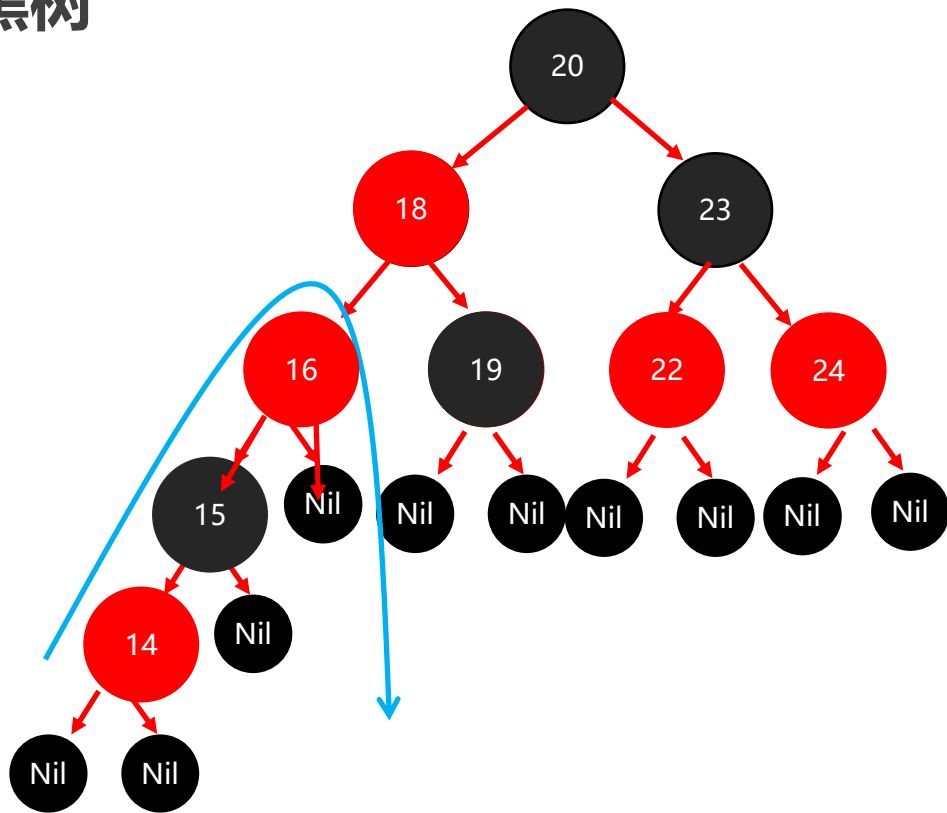
1. 将“父节点16”设为黑色，将“叔叔节点19”设为黑色
2. 将“祖父节点18”设为“红色”。
3. 如果祖父节点为根节点，则将根节点再次变成黑色。



其父节点为红色，叔叔节点也是黑色



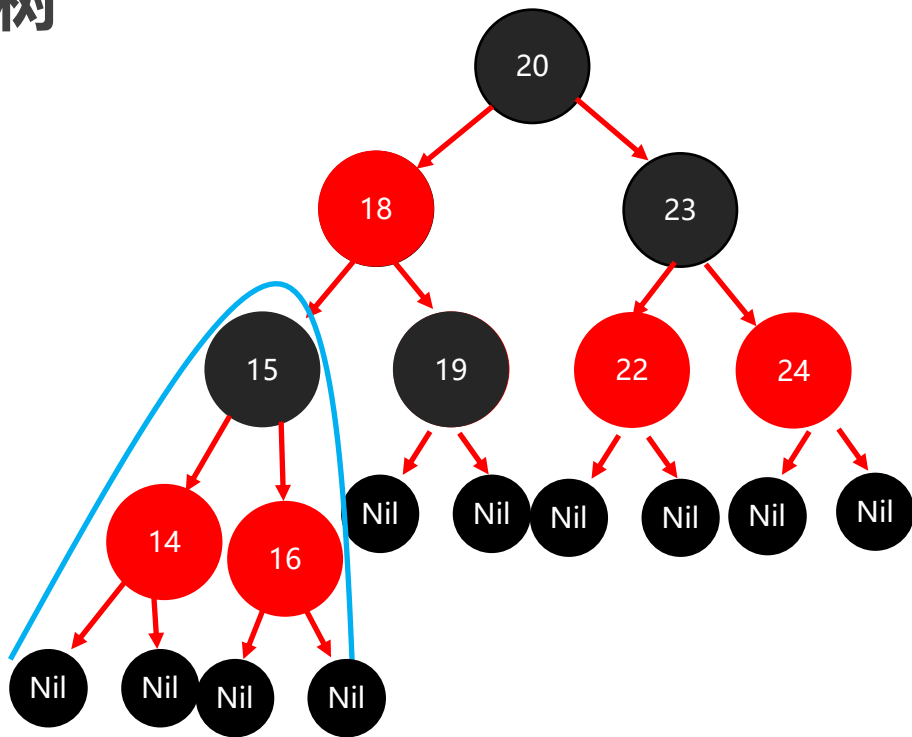
1. 将“父节点15”设为“黑色”
2. 将“祖父节点16”设为“红色”。
3. 以祖父节点为支点进行旋转



其父节点为红色，叔叔节点也是黑色



1. 将“父节点15”设为“黑色”
2. 将“祖父节点16”设为“红色”。
3. 以祖父节点为支点进行旋转



其父节点为红色，叔叔节点也是黑色



1. 将“父节点15”设为“黑色”
2. 将“祖父节点16”设为“红色”。
3. 以祖父节点为支点进行旋转

红黑树小结

红黑树不是高度平衡的，它的平衡是通过"红黑规则"进行实现的

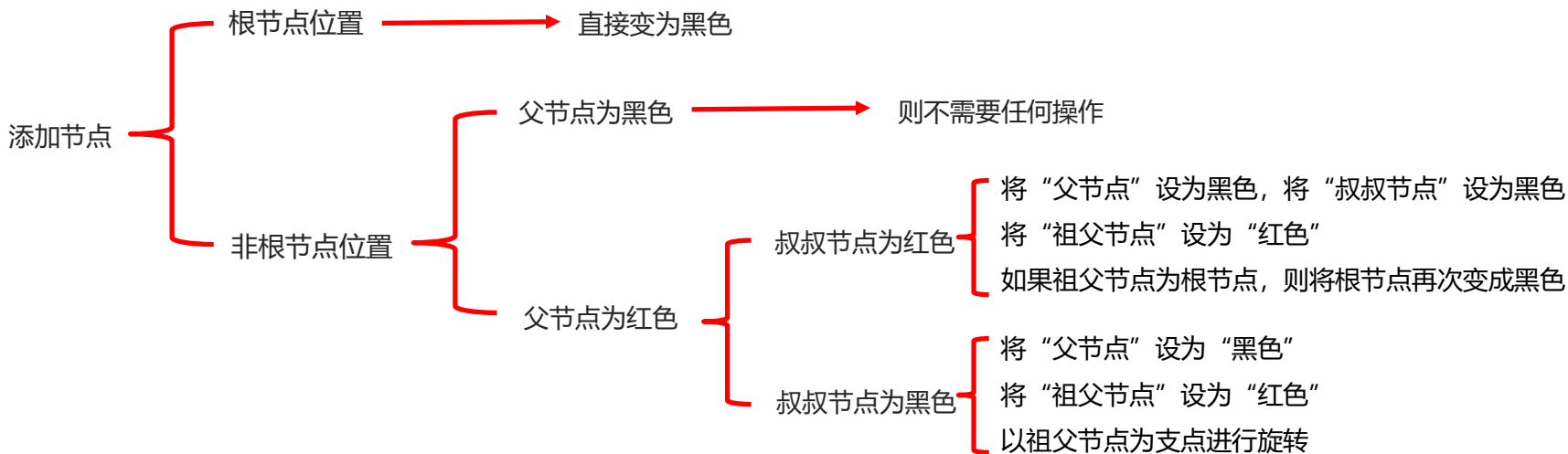
规则如下：

1. 每一个节点或是红色的，或者是黑色的。
2. 根节点必须是黑色
3. 如果一个节点没有子节点或者父节点，则该节点相应的指针属性值为Nil，这些Nil视为叶节点，每个叶节点(Nil)是黑色的；
4. 不能出现两个红色节点相连的情况
5. 对每一个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点；

红黑树小结

红黑树在添加节点的时候：

添加的节点默认是红色的。



第三章 List

目录 Contents

- ◆ List接口介绍
- ◆ LinkedList集合

目标 TARGET

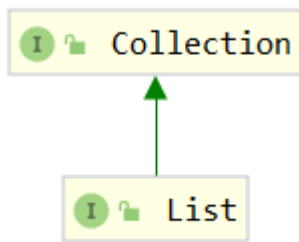
- ◆ 熟悉List集合的特点
- ◆ 熟悉List接口常用特有方法
- ◆ 熟悉List集合的常见实现类及其底层数据结构

1 List集合特点

List集合是Collection集合子类型，继承了所有Collection中功能，同时List增加了带索引的功能

特点如下：

1. 元素的存取是有序的【有序】
2. 元素具备索引【有索引】
3. 元素可以重复存储【可重复】

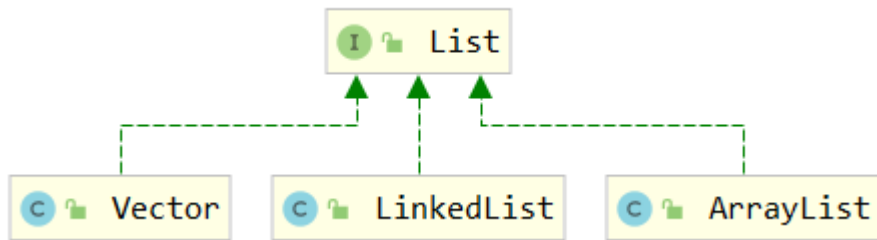


2 常用子类

ArrayList: 底层结构就是数组【查询快，增删慢】

Vector: 底层结构也是数组（线程安全，同步安全的，低效，用的就少）


LinkedList: 底层是链表结构（双向链表）【查询慢，增删快】



3 List中特有的方法

List继承了Collection中所有方法，元素具备索引特性，因此新增了一些含有索引的特有方法，如下：

- public void **add**(int index, E element): 将指定的元素，添加到该集合中的指定位置上。
- public E **get**(int index): 返回集合中指定位置的元素。
- public E **remove**(int index): 移除列表中指定位置的元素, 返回的是被移除的元素。
- public E **set**(int index, E element): 用指定元素替换集合中指定位置的元素, 返回值的更新前的元素。



代码实践

List集合有什么特点？

1. 有序 2. 可重复的 3. 有索引

总结

List集合有哪些常用的子类及底层数据结构是啥？

1. ArrayList: 数组结构
2. LinkedList: 双向链表
3. Vector: 数组结构

List集合有哪些常用的特有方法？

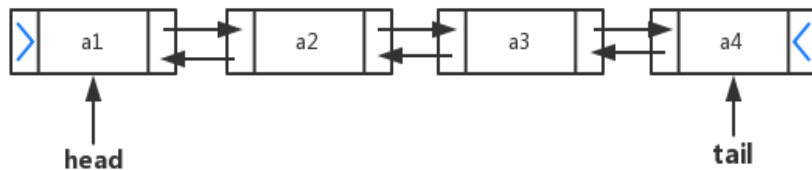
add remove set get

目标 TARGET

- ◆ 熟悉LinkedList首尾操作方法
- ◆ 能够自己查阅理解add, get等方法的

1 LinkedList的介绍

LinkedList底层结构是双向链表。每个节点有三个部分的数据，一个是保存元素数据，一个是保存前一个节点的地址，一个是保存后一个节点的地址。可以双向查询，效率会比单向链表高。



2 LinkedList特有方法

public void addFirst(E e):将指定元素插入此列表的开头。

public void addLast(E e):将指定元素添加到此列表的结尾。

public E getFirst():返回此列表的第一个元素。

public E getLast():返回此列表的最后一个元素。

public E removeFirst():移除并返回此列表的第一个元素。

public E removeLast():移除并返回此列表的最后一个元素。

第四章 Set

目录 Contents

- ◆ Set接口介绍
- ◆ HashSet集合
- ◆ 哈希表结构
- ◆ LinkedHashSet集合

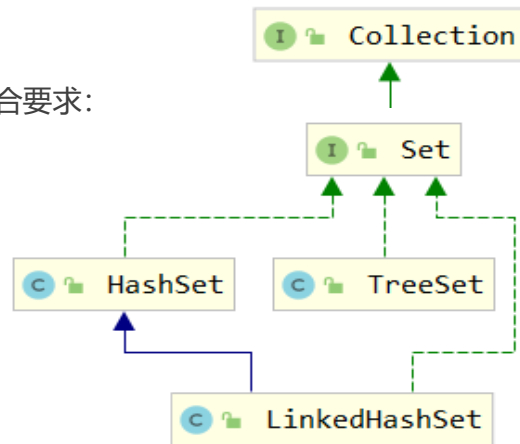
目标 TARGET

- ◆ 能够熟悉Set集合的特点
- ◆ 熟悉常见的Set实现类

1 Set集合介绍

Set集合也是Collection集合的子类型，**没有特有方法**。Set比Collection定义更严谨，Set集合要求：

1. 元素不能保证添加和取出顺序（无序）
2. 元素是没有索引的(无索引)
3. 元素唯一(元素唯一)



2 Set常用子类

HashSet: 底层由HashMap，
底层结构哈希表结构。
去重，无索引，无序。
哈希表结构的集合，操作效率会
非常高。

LinkedHashSet:底层结构链表
加哈希表结构。
具有哈希表表结构的特点，也
具有链表的特点。

TreeSet: 底层是有TreeMap，
底层数据结构 红黑树。
去重，让存入的元素具有排序
(升序排序)

Set集合特点？

无序
无索引
去重

总结

Set集合有哪些常用的子类，底层结构是什么？

HashSet 哈希表结构
LinkedHashSet 链表+哈希表结构
TreeSet 红黑树

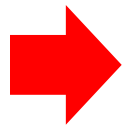
目标 TARGET

- ◆ 使用使用HashSet集合对数据进行去重
- ◆ 理解哈希表结构

1 HashSet概述

java.util.HashSet是Set接口的实现类，没有特有方法。底层是哈希表结构，具有去重特点。

```
HashSet<Integer> set = new HashSet<>();  
set.add(10);  
set.add(10);  
set.add(20);  
set.add(30);  
System.out.println(set);
```



[20, 10, 30]

练习：使用HashSet集合存储字符串并遍历

练习：使用HashSet集合存储自定义对象并遍历

目标 TARGET

- ◆ 能够说出哈希表结构的组成及特点

哈希值

哈希值：是JDK根据对象的**地址**或者**对象的属性**算出来的int类型的**数值**

Object类中有一个方法可以获取**对象的哈希值**

- `public int hashCode()`：返回对象的哈希码值

对象的哈希值特点

- 同一个对象多次调用`hashCode()`方法返回的哈希值是相同的
- 默认情况下，不同对象的哈希值是不同的。而重写`hashCode()`方法，不同对象的哈希值有可能相同



常见数据结构之哈希表

哈希表

- JDK8之前，底层采用**数组+链表**实现
- JDK8以后，底层采用数组 + 链表/红黑树实现

HashSet1.7版本原理解析

table[]

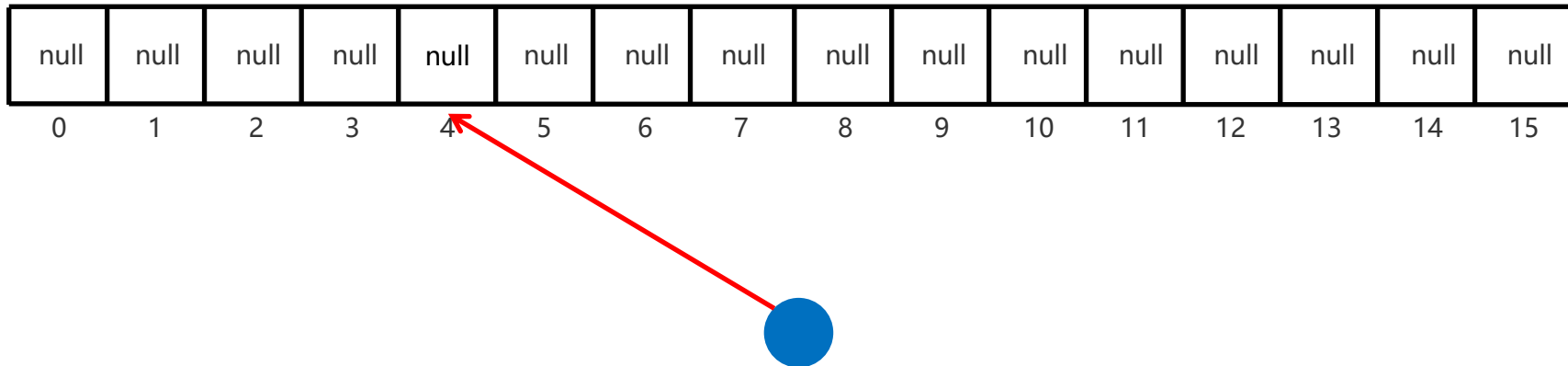
null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

```
HashSet<String> hm = new HashSet<>();
```

1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table

HashSet1.7版本原理解析

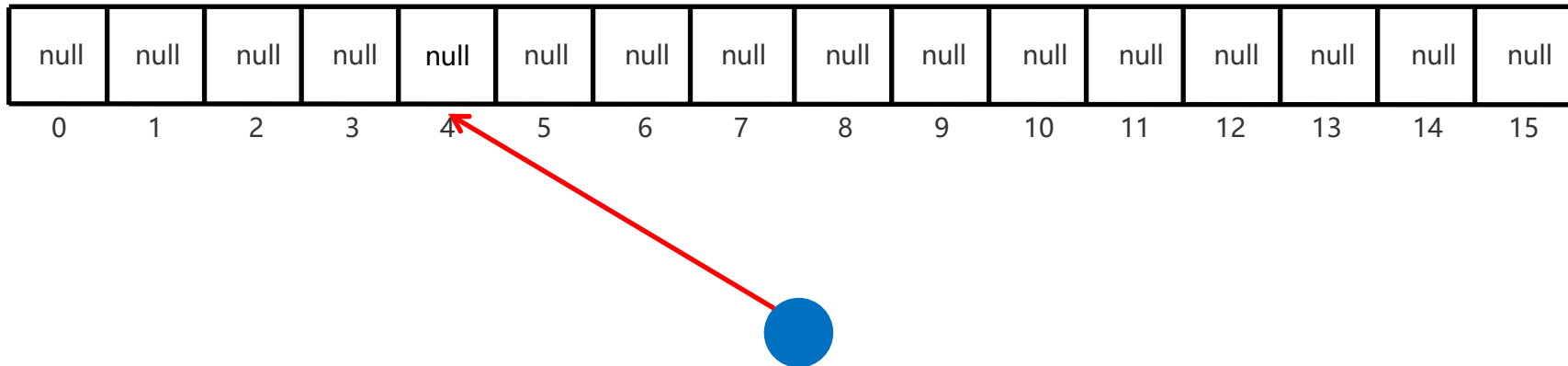
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置

HashSet1.7版本原理解析

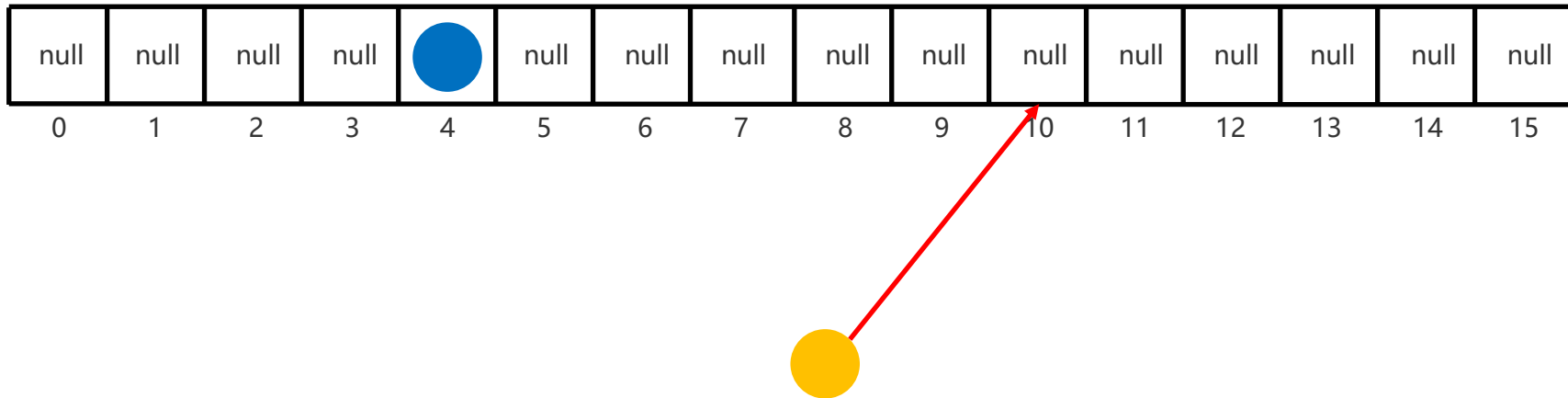
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入

HashSet1.7版本原理解析

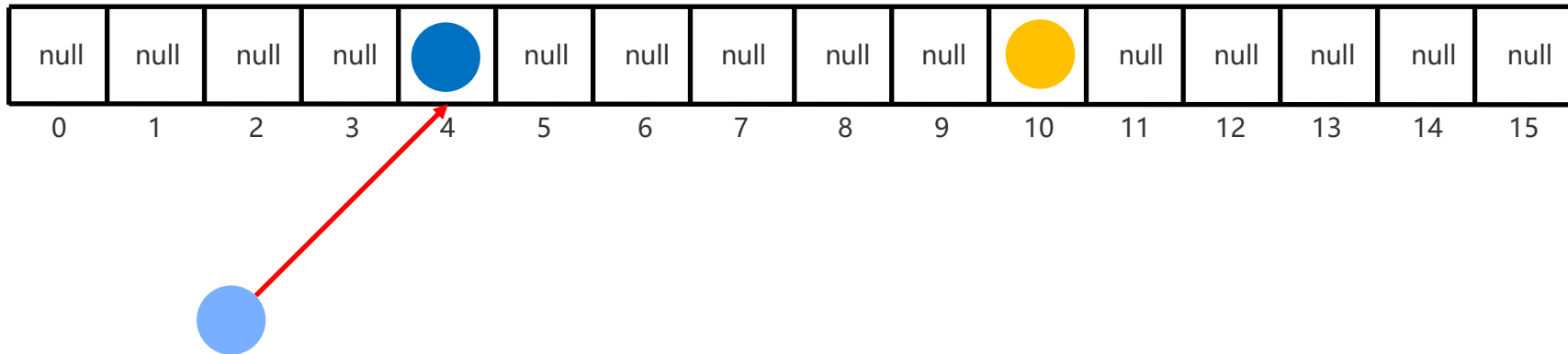
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入

HashSet1.7版本原理解析

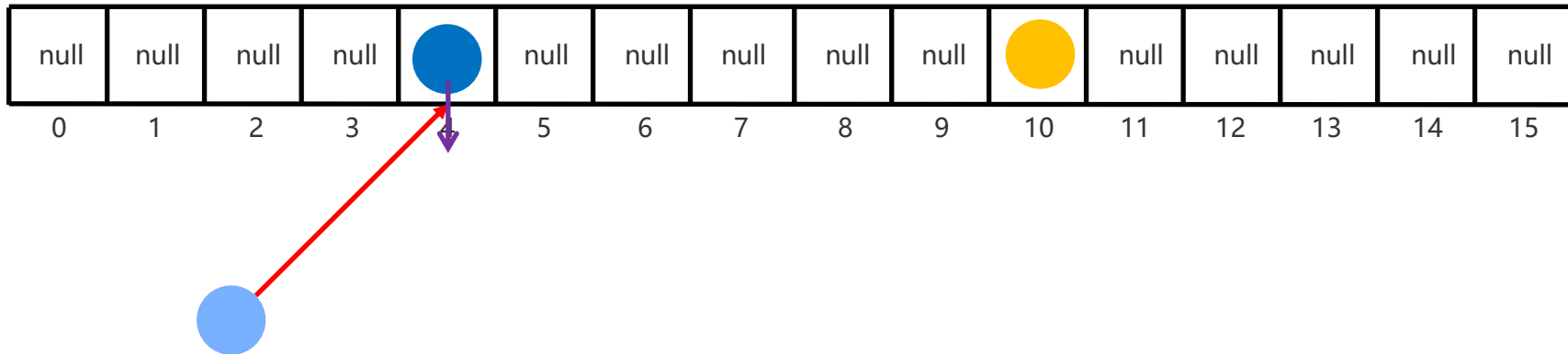
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较属性值

HashSet1.7版本原理解析

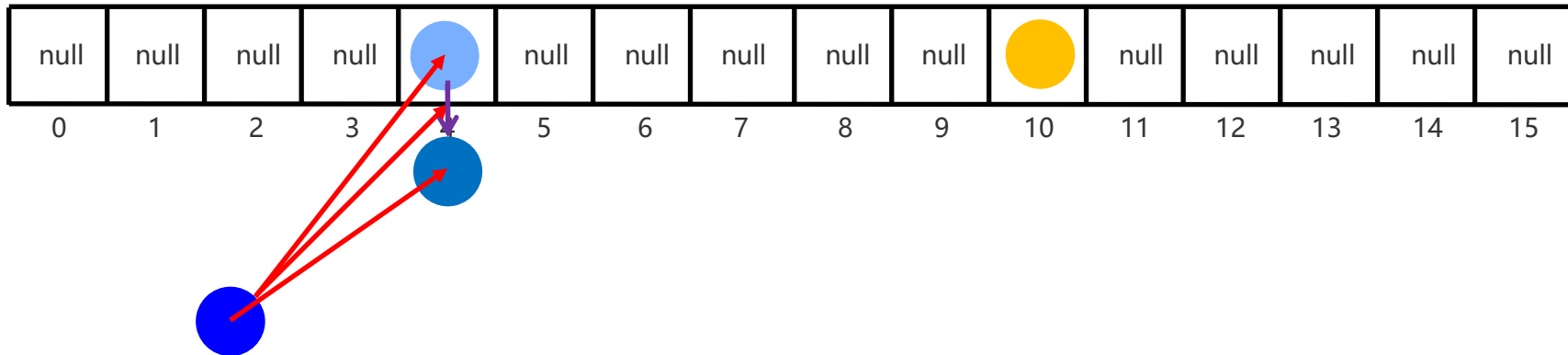
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较属性值
5. 如果一样，则不存，如果不一样，则存入数组，老元素挂在新元素下面

HashSet1.7版本原理解析

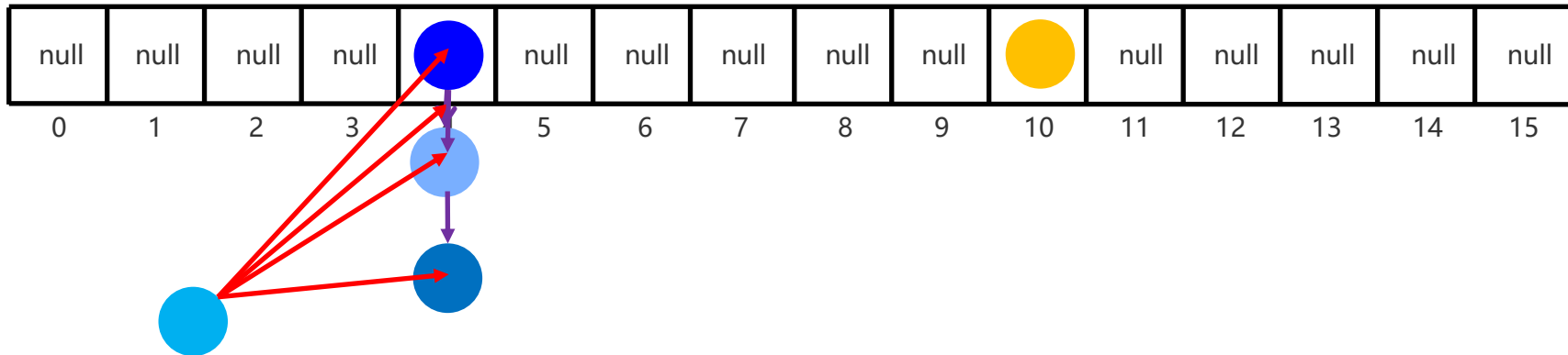
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较属性值
5. 如果一样，则不存，如果不一样，则存入数组，老元素挂在新元素下面

HashSet1.7版本原理解析

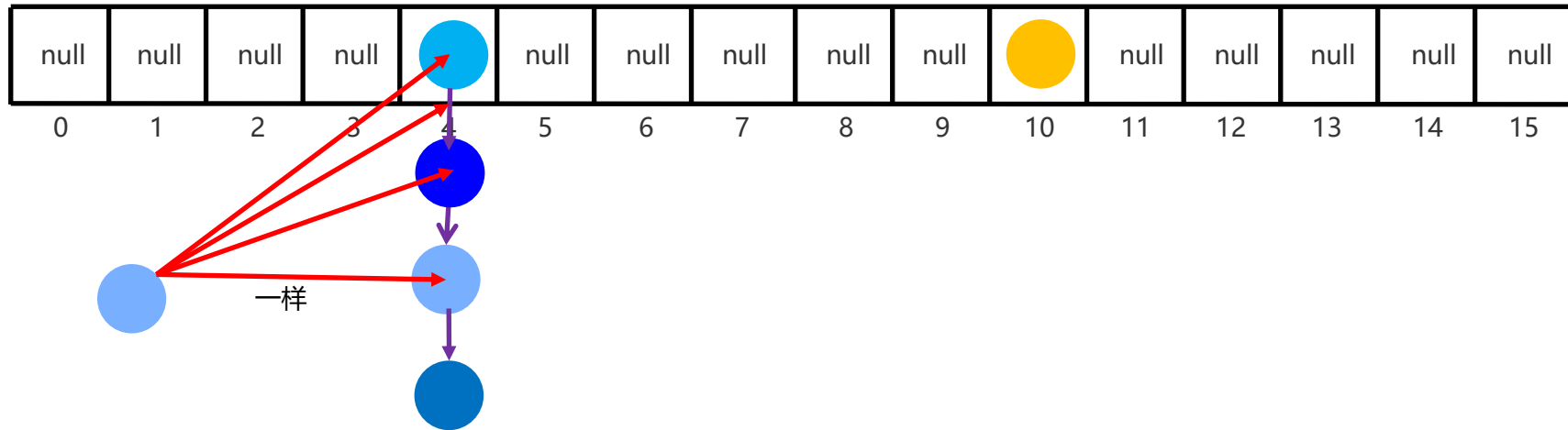
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较属性值
5. 如果一样，则不存，如果不一样，则存入数组，老元素挂在新元素下面

HashSet1.7版本原理解析

Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较属性值
5. 如果一样，则不存，如果不一样，则存入数组，老元素挂在新元素下面

HashSet1.8版本原理解析

底层结构：哈希表。（数组、链表、红黑树的结合体）。

当挂在下面的元素过多，那么不利于查询，所以在JDK8以后，
当链表长度超过8的时候，自动转换为红黑树。

存储流程不变。

HashSet1.8版本原理解析

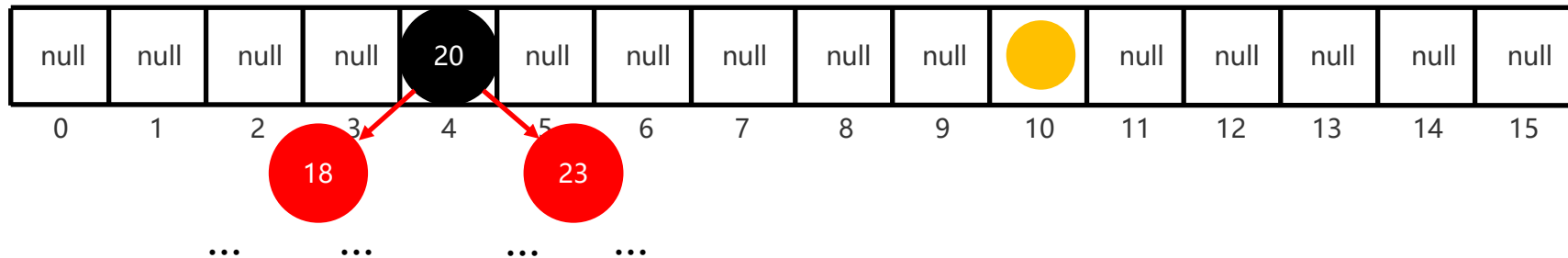
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较属性值
5. 如果一样，则不存，如果不一样，则存入数组，老元素挂在新元素下面

HashSet1.8版本原理解析

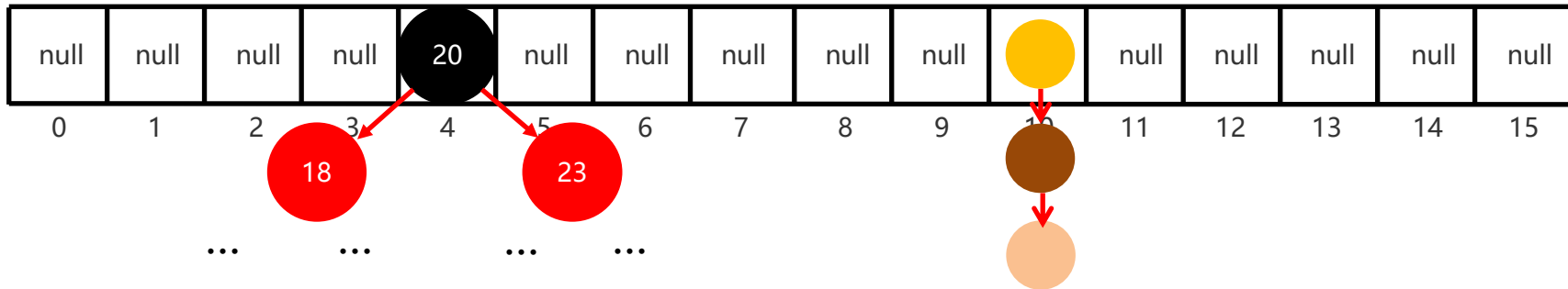
Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较属性值
5. 如果一样，则不存，如果不一样，则存入数组，老元素挂在新元素下面

HashSet1.8版本原理解析

Segment[]



1. 创建一个默认长度16，默认加载因为0.75的数组，数组名table
2. 根据元素的哈希值跟数组的长度计算出应存入的位置
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较属性值
5. 如果一样，则不存，如果不一样，则存入数组，老元素挂在新元素下面

目标 TARGET

- ◆ 能够说出底层数据结构，及其使用特点

1 LinkedHashSet 介绍

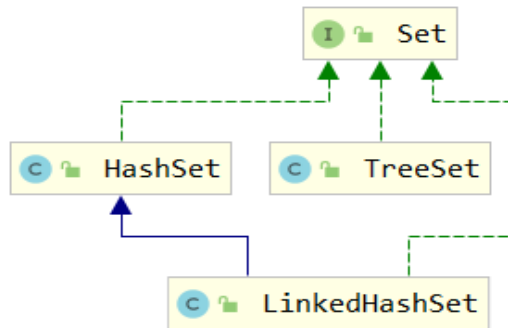
LinkedHashSet底层由链表结构和哈希表结构。

链表结构：是为了保证插入顺序

哈希表结构：是为了去重

存储元素的时候，先过哈希表，如果哈希表能够接受数据，进一步存到链表结构表结构实实现：

```
LinkedHashSet<Integer> set = new LinkedHashSet<>();  
set.add(10);  
set.add(10);  
set.add(20);  
set.add(30);  
System.out.println(set); //[10, 20, 30]
```



请说出LinkedHashSet底层数据结构，及其特点

底层使用哈希表结构和链表结构，元素存储时有去重，有序的特点

总结



传智播客旗下高端IT教育品牌