

查找



学习目标

理解查找的基本概念

掌握常用查找算法及其性能分析

掌握树表查找（二叉排序树、平衡二叉树）

掌握B树概念（B树）

树表的提出

🕒 如何在一个大型的数据集合上进行动态查找？

📎 顺序查找：不要求元素的有序性，查找性能是 $O(n)$ ，插入、删除的性能是 $O(1)$

📎 折半查找：查找性能是 $O(\log_2 n)$
为保证元素的有序性，插入、删除要移动元素，性能是 $O(n)$

🕒 有没有一种查找结构，使得插入、删除、查找均具有较好效率？

📌 树表：将查找集合组织成树的结构 \Rightarrow 二叉排序树
(二叉查找树)



二叉排序树 (二叉查找树)

讲什么？



二叉排序树的定义



二叉排序树的类定义



二叉排序树的实现——查找



二叉排序树的实现——插入



二叉排序树的实现——删除

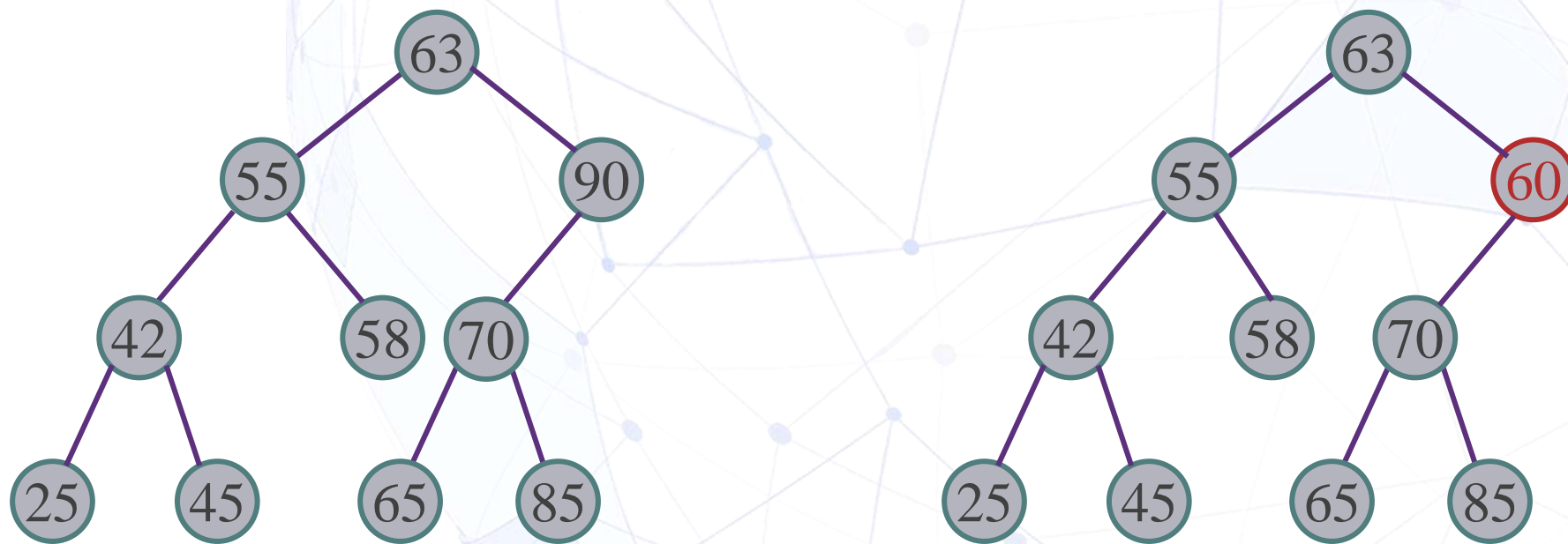


二叉排序树的性能分析

二叉排序树的定义

✚ 二叉排序树（二叉查找树）：或者是一棵空的二叉树，或者是具有下列性质的二叉树：

- (1) 若它的左子树不空，则左子树上所有结点的值均小于根结点的值
- (2) 若它的右子树不空，则右子树上所有结点的值均大于根结点的值
- (3) 它的左右子树也都是二叉排序树



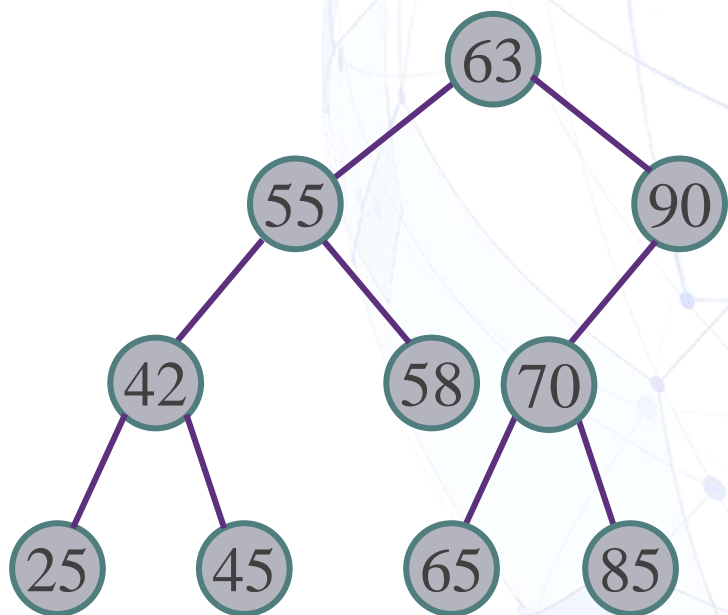
二叉排序树的定义

二叉排序树是记录之间满足某种大小关系的二叉树



写出二叉排序树的中序序列? \Rightarrow 升序序列 \Rightarrow 二叉排序树

25 42 45 55 58 63 65 70 85 90



观察一下这个中序序列，63前面一个值58处在什么位置？65呢？

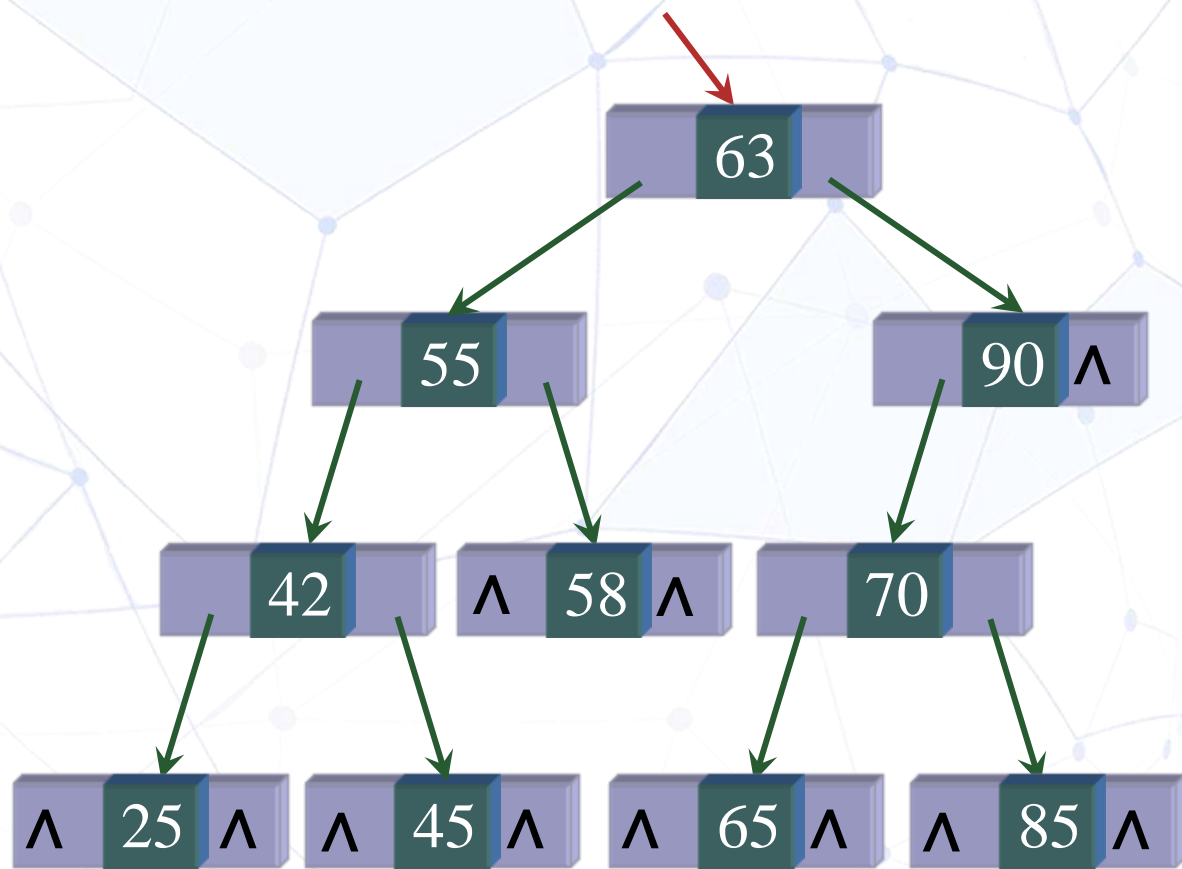
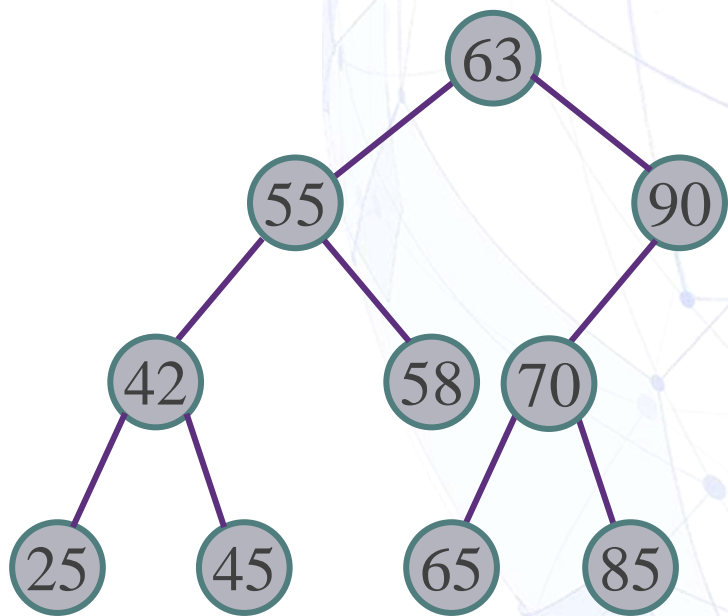
二叉排序树的存储



如何存储二叉排序树?



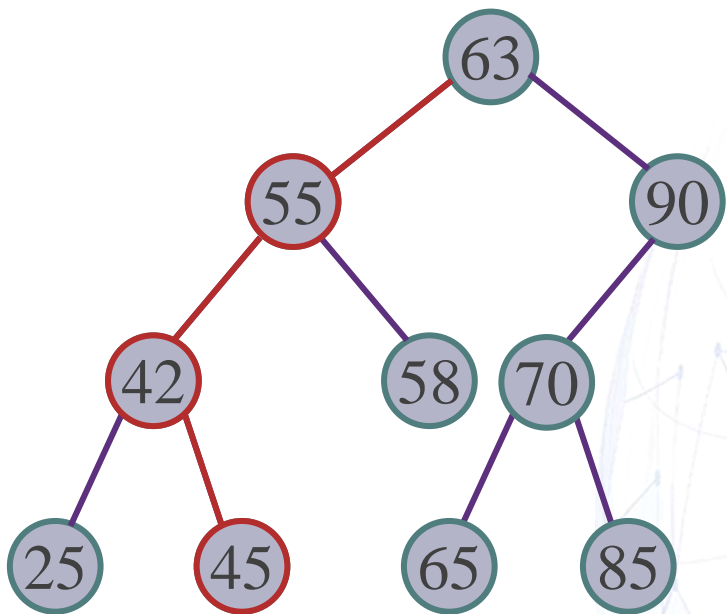
二叉链表



二叉排序树的类定义

```
class BiSortTree
{
public:
    BiSortTree(int a[ ], int n);
    ~ BiSortTree( ){Release(root);}
    BiNode<int> *InsertBST(int x) {return InsertBST(root, x);}
    void DeleteBST(BiNode<int> *p, BiNode<int> *f );
    BiNode<int> *SearchBST(int k) {return SearchBST(root, k);}
private:
    BiNode<int> *InsertBST(BiNode<int> *bt , int x);
    BiNode<int> *SearchBST(BiNode<int> *bt, int k);
    void Release(BiNode<DataType> *bt);
    BiNode<int> *root;
};
```

查找操作



在二叉排序树中查找给定值 k 的过程是：

- (1) 若root是空树，则查找失败；
- (2) 若 $k = \text{root} \rightarrow \text{data}$ ，则查找成功；
- (3) 若 $k < \text{root} \rightarrow \text{data}$ ，则在 bt 的左子树上查找；
- (4) 若 $k > \text{root} \rightarrow \text{data}$ ，则在 bt 的右子树上查找。

二叉排序树的查找效率在于
只需查找二个子树之一

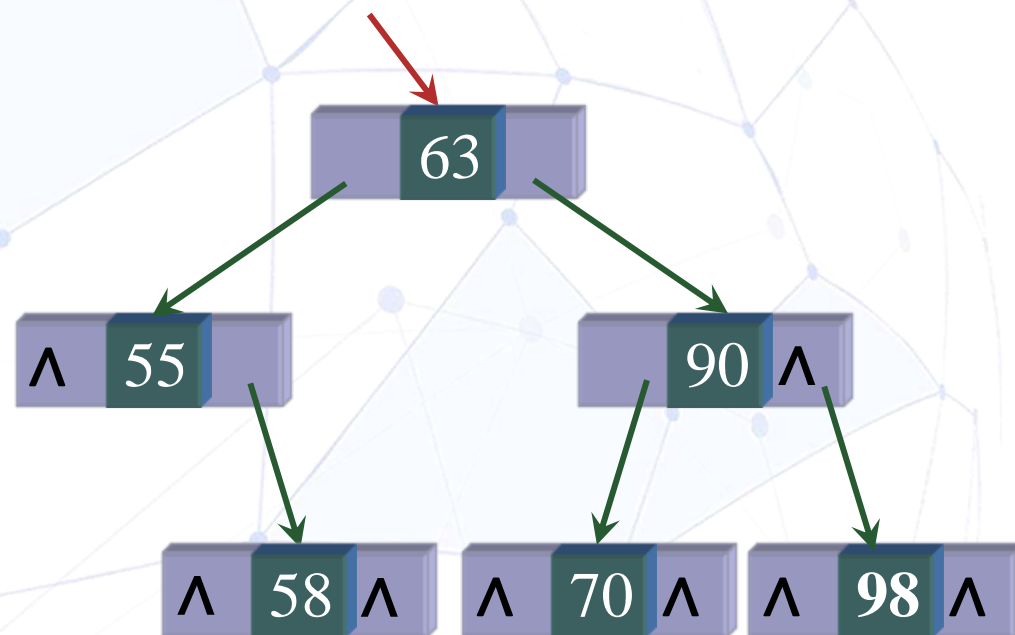
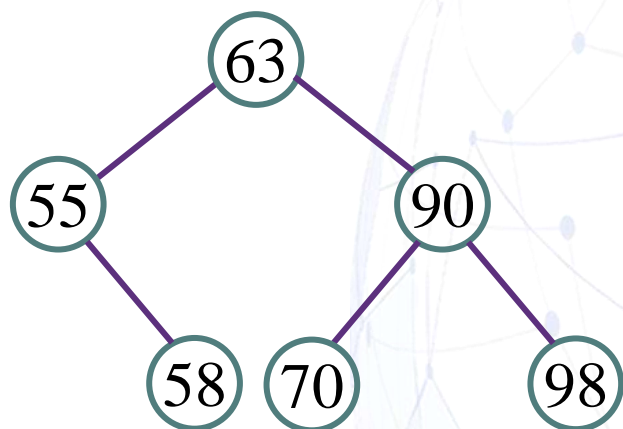
```
BiNode<int> * BiSortTree :: SearchBST(BiNode<int> *bt, int k)
{
    if (bt == nullptr) return nullptr;
    if (bt->data == k) return bt;
    else if (bt->data > k) return SearchBST(bt->lchild, k);
    else return SearchBST(bt->rchild, k);
}
```

插入操作



如何在二叉排序树中插入一个元素？例如，插入98

插入55

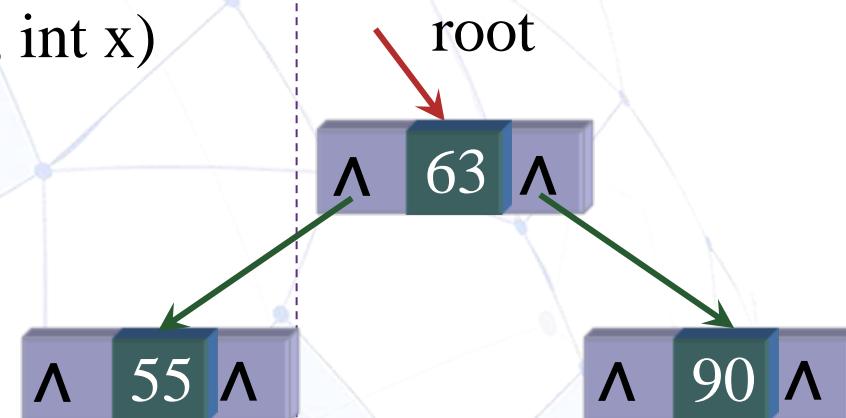


如果新插入的结点已经存在，无需插入；
否则，新插入的结点必为一个新的叶子结点，其插入位置由查找过程得到

若二叉排序树为空树，则新插入的结点为新的根结点；

插入操作

```
BiNode<int> *BiSortTree :: InsertBST(BiNode<int> *bt, int x)
{
    if (bt == nullptr) {                //找到插入位置
        BiNode<int> *s = new BiNode<int>;
        s->data = x;
        s->lchild = nullptr; s->rchild = nullptr;
        bt = s;
        return bt;
    }
    else if (bt->data == x) return bt;
        else if (bt->data > x) bt->lchild = InsertBST(bt->lchild, x);
            else bt->rchild = InsertBST(bt->rchild, x);
}
```



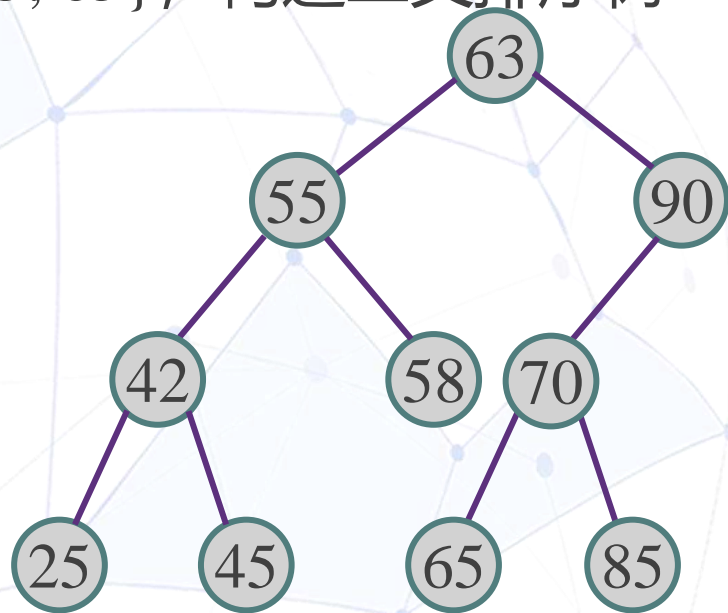
二叉排序树的构造

从空树出发，经过一系列查找和插入操作后，可以生成一棵二叉排序树

二叉排序树的构造

例如，给定查找集合{63, 55, 42, 45, 58, 90, 70, 25, 85, 65}，构造二叉排序树

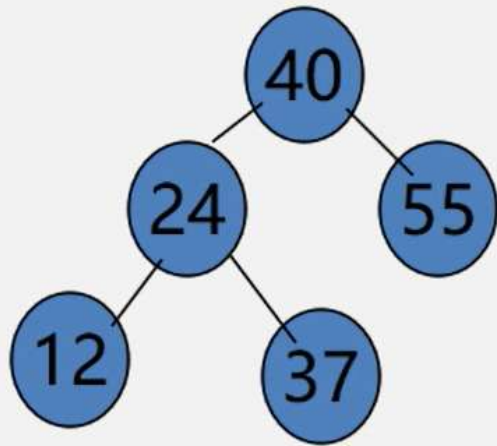
```
BiSortTree::BiSortTree(int a[ ], int n)
{
    root = nullptr;
    for (int i = 0; i < n; i++)
        root = InsertBST(root, a[i]);
}
```



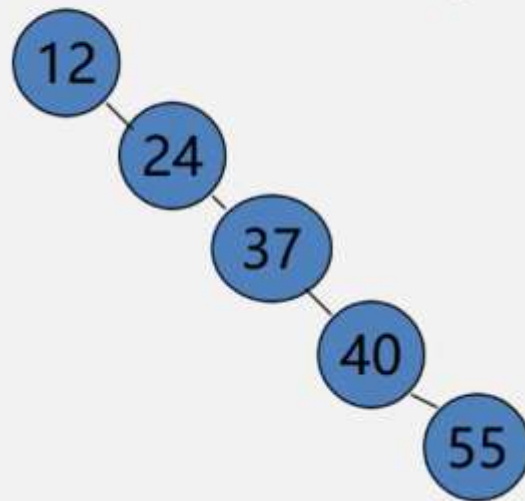
- (1) 每次插入的新结点都是二叉排序树上新的叶子结点;
- (2) 找到插入位置后，不必移动其它结点，**仅需修改某个结点的指针**;
- (3) 在左子树/右子树的查找过程与在整棵树上查找过程相同;
- (4) 新插入的结点**没有破坏**原有结点之间的**关系**。

关键字的输入顺序不同，建立的不同二叉排序树。

40, 24, 12, 37, 55



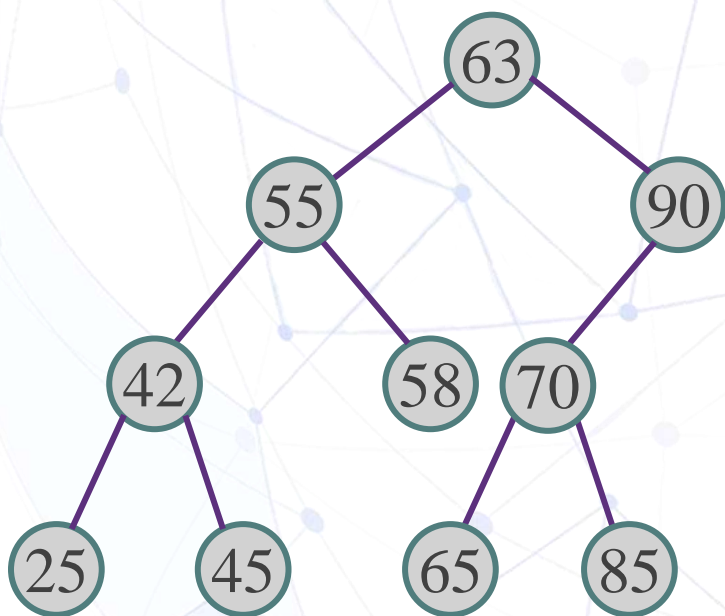
12, 24, 37, 40, 55



删除操作

🕒 如何在二叉排序树中删除一个元素？

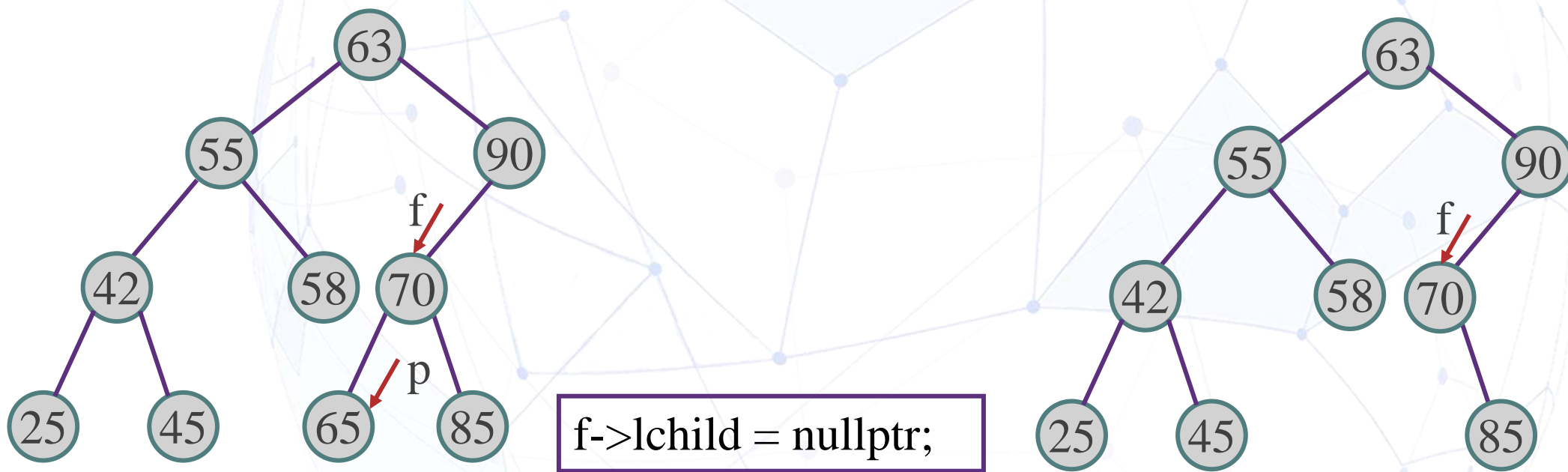
- (1) 从二叉排序树中删除一个结点之后，要仍然保持二叉排序树的特性；
- (2) 当删除分支结点时破坏了二叉排序树中原有结点之间的链接关系，需要重新修改指针，使得删除结点后仍为一棵二叉排序树。



删除操作

不失一般性，设待删除结点为 p ，其双亲结点为 f ，且 p 是 f 的左孩子，

📎 情况 1——被删除的结点是叶子结点

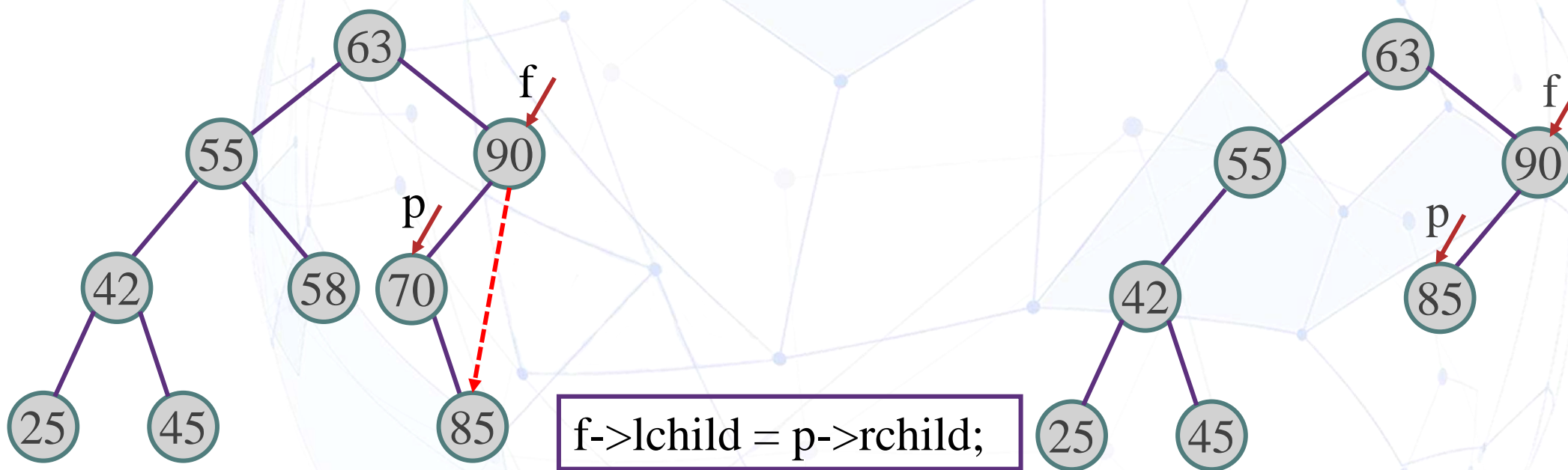


操作：将双亲结点中相应指针域的值改为空

删除操作

不失一般性，设待删除结点为 p ，其双亲结点为 f ，且 p 是 f 的左孩子，

📎 情况 2——被删除的结点只有左子树或者只有右子树

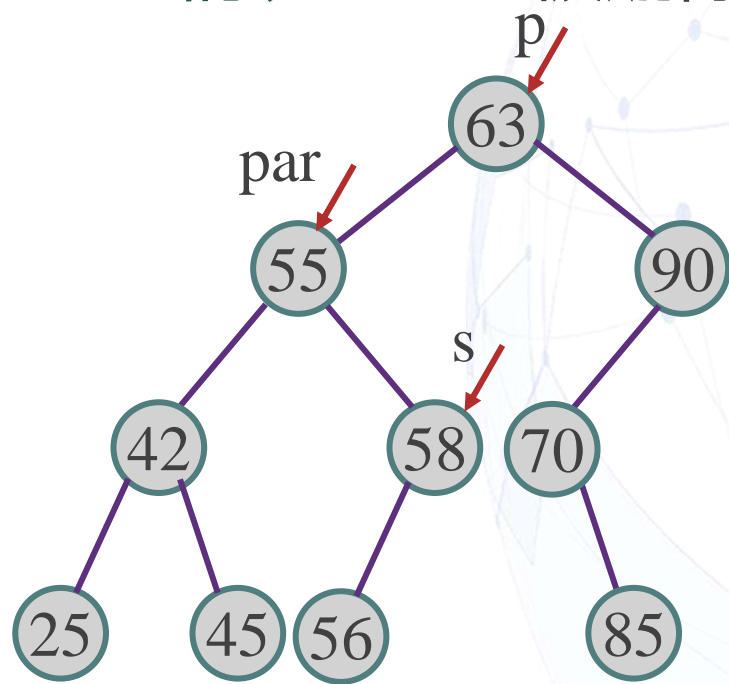


操作：将双亲结点中相应指针域指向被删除结点的左子树（或右子树）

删除操作

不失一般性，设待删除结点为 p ，其双亲结点为 f ，且 p 是 f 的左孩子，

📎 情况 3——被删除的结点既有左子树也有右子树



```
BiNode *par = p, *s = p->lchild;
while (s->rchild != NULL)
```

```
{
```

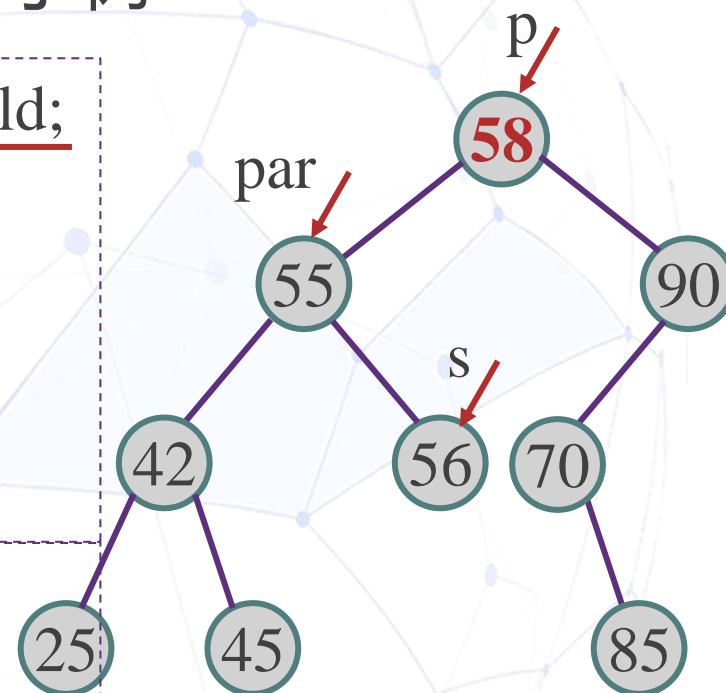
```
    par = s;
```

```
    s = s->rchild;
```

```
}
```

```
p->data = s->data;
```

```
par->rchild = s->lchild;
```

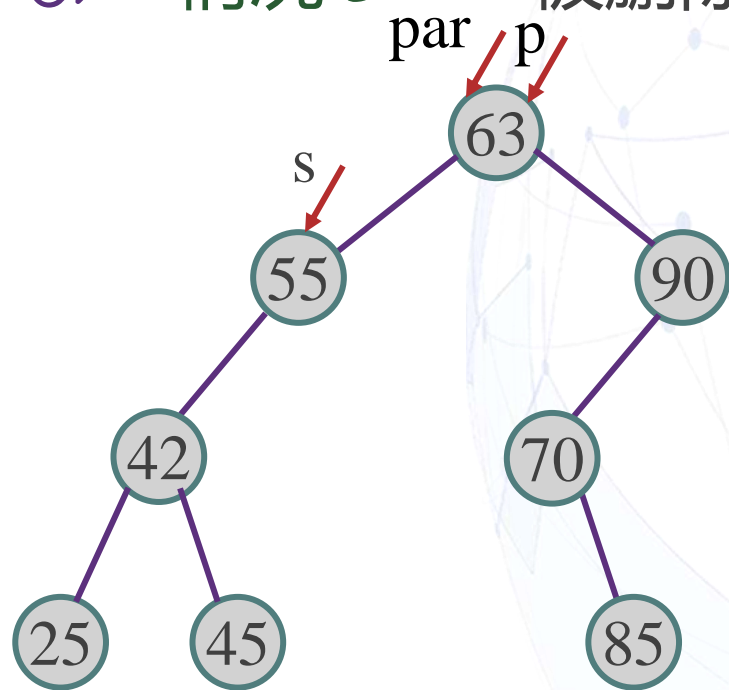


操作：以其左子树中的最大值（？）结点替换之，然后再删除该结点

删除操作

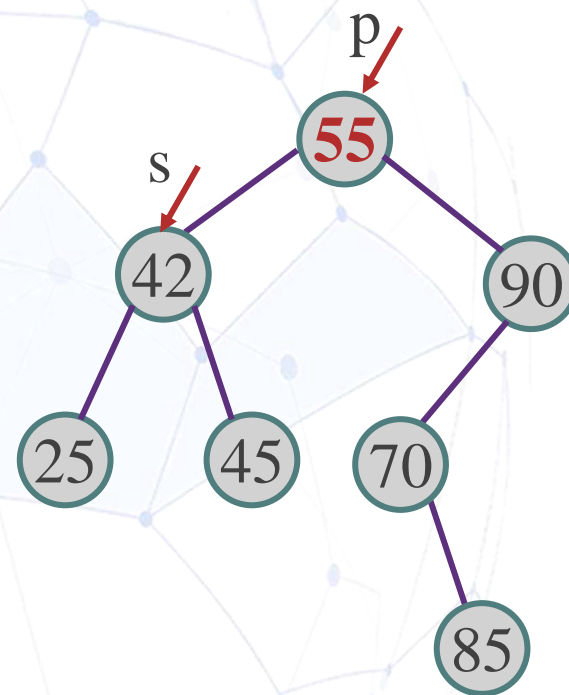
不失一般性，设待删除结点为 p ，其双亲结点为 f ，且 p 是 f 的左孩子，

📎 情况 3——被删除的结点既有左子树也有右子树



特殊情况：左子树中的最大值
结点是被删结点的孩子

```
if (p == par)
    par->lchild = s->lchild;
```



操作：以其左子树中的最大值结点替换之，然后再删除该结点

删除操作



请总结并描述二叉排序树的删除算法



能否以其右子树中的最小值结点替换之，然后再删除该结点？

```
BiNode *par = p, *s = p->lchild;
while (s->rchild != nullptr)
{
    par = s;
    s = s->rchild;
}
p->data = s->data;
if (par == p) par->lchild = s->lchild;
else par->rchild = s->lchild;
delete s;
```

```
/*查找右子树的最左下结点*/
BiNode *par = p, *s = p->rchild;
while (s->lchild != nullptr)
{
    par = s;
    s = s->lchild;
}
p->data = s->data;
if (par == p) par->rchild = s->rchild;
else par->lchild = s->rchild;
delete s;
```

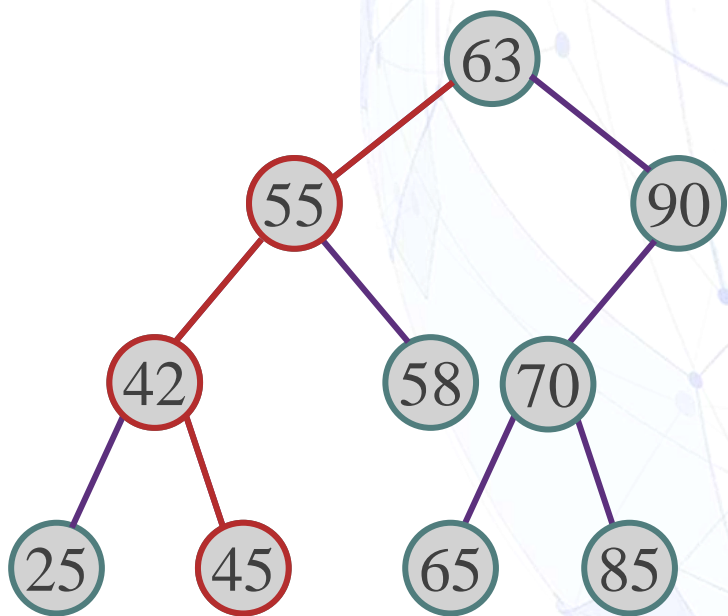
性能分析

🕒 二叉排序树的查找性能取决于什么？

比较次数不超过树的深度

二叉排序树的查找效率在于
只需查找二个子树之一

在二叉排序树中执行插入和删除操作



查找插入和删除的位置

修改相应指针

插入、删除、查找的时间复杂度相同

性能分析

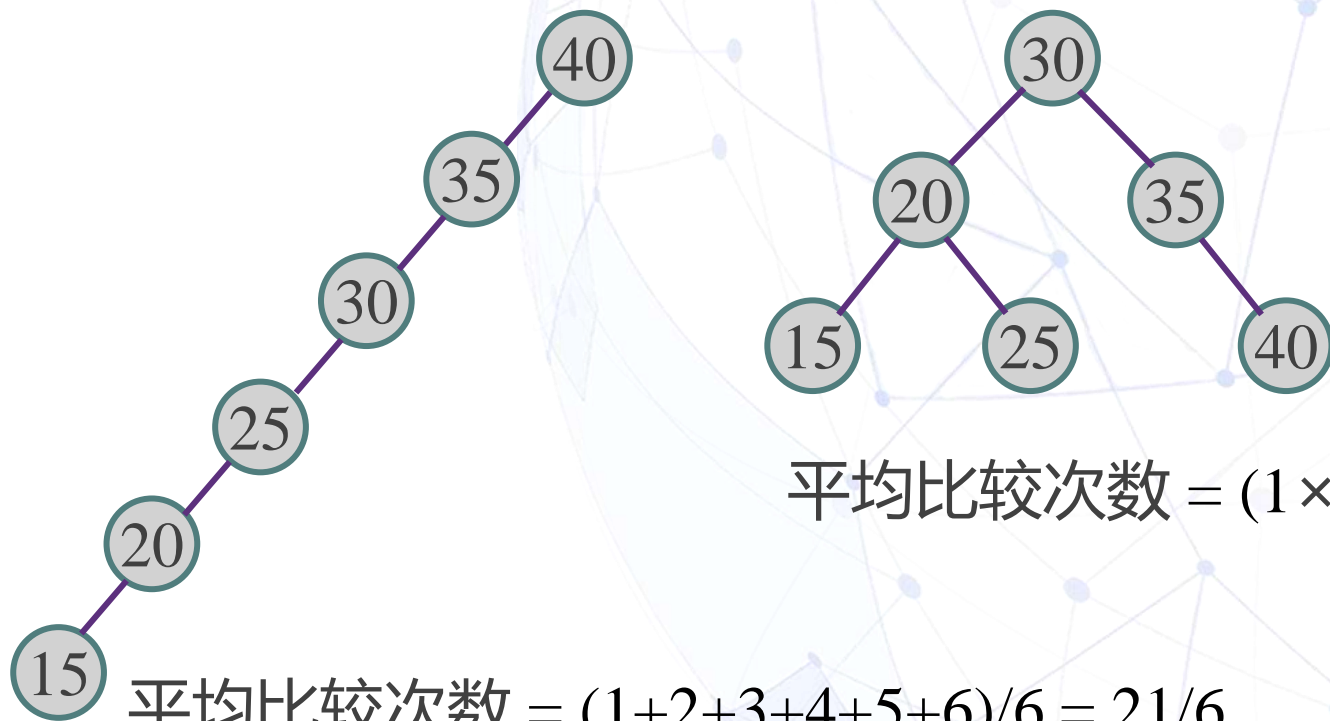


二叉排序树的深度是多少？取决于什么？

查找集合的初始排列

例如，给定查找集合{40, 35, 30, 25, 20, 15}，构造的二叉排序树深度为 n

例如，给定查找集合{15, 20, 25, 30, 35, 40}，构造的二叉排序树深度为 $\lfloor \log_2 n \rfloor + 1$



📎 最坏情况：退化为线性查找

📎 最好情况：相当于折半查找

📎 平均情况： $O(n) \sim O(\log_2 n)$

$$\text{平均比较次数} = (1 \times 1 + 2 \times 2 + 3 \times 3) / 6 = 14/6$$

$$\text{平均比较次数} = (1 + 2 + 3 + 4 + 5 + 6) / 6 = 21/6$$

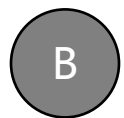


1. 二叉排序树的充要条件是任一结点的值均大于其左孩子的值, 小于其右孩子的值。



A

正确



B

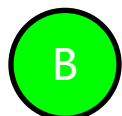
错误

提交

2. 若二叉排序树中关键码互不相同，则其中最小元素和最大元素一定是叶子结点。



正确



错误

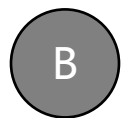
提交

3. 二叉排序树可以实现在大型的数据集合上进行动态查找。



A

正确



B

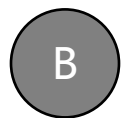
错误

提交

4. 二叉排序树中关键码互不相同，对二叉排序树进行中序遍历，其结果一定是升序序列。



正确



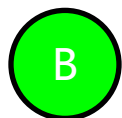
错误

提交

5. 在二叉排序树中，新插入的结点一定是叶子结点，因此，一定会增加二叉排序树的高度。



正确



错误

提交

6. 二叉排序树的查找和折半查找的时间性能相同。

☐ A 正确

☒ B 错误

提交



7-3-2 平衡二叉树

讲什么？



平衡二叉树的定义



最小不平衡子树的定义



平衡二叉树的平衡调整——LL型、RR型



平衡二叉树的平衡调整——LR型、RL型

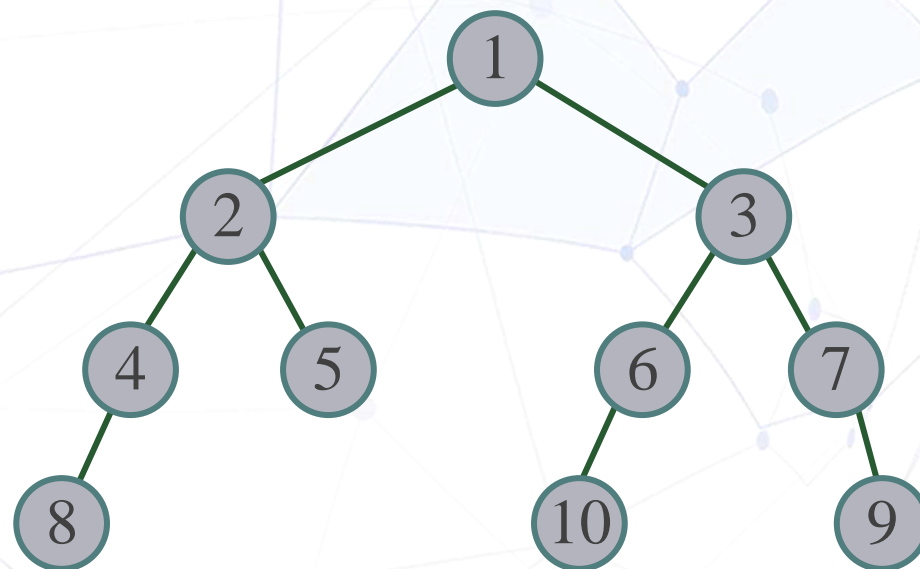
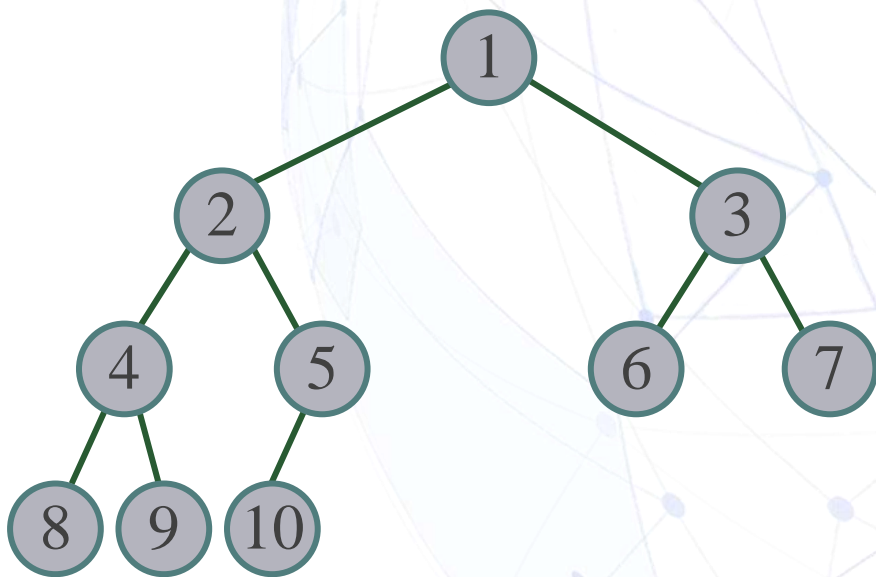
平衡二叉树的提出

📎 二叉排序树的深度取决于给定查找集合的排列，即结点的插入顺序

🕒 具有 n 个结点的二叉排序树，最小深度是多少？有什么特征？

完全二叉树的深度： $\lfloor \log_2 n \rfloor + 1$

左右子树的深度相同



平衡二叉树的提出

📎 二叉排序树的深度取决于给定查找集合的排列，即结点的插入顺序

🕒 具有 n 个结点的二叉排序树，最小深度是多少？有什么特征？

完全二叉树的深度： $\lfloor \log_2 n \rfloor + 1$



平衡二叉树的深度： $1.44 \log_2 (n + 2) - 1.328$

左右子树的深度相同



左右子树的深度相差 1

📎 维护最小深度的二叉排序树的代价太大

🕒 对于查找集合的任意排列，如何得到一棵深度尽可能小的二叉排序树？

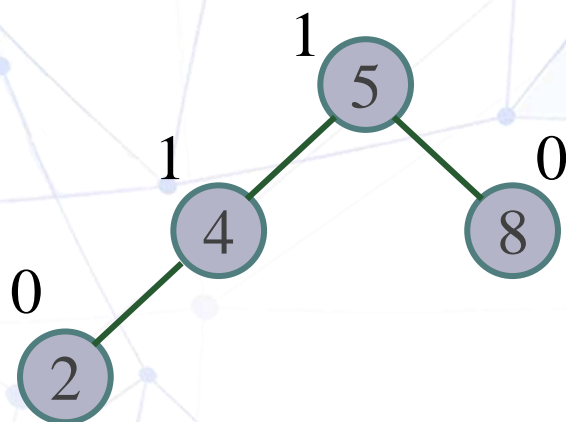
平衡二叉树的定义

✚ 平衡因子：该结点的左子树的深度（高度）减去右子树的深度（高度）

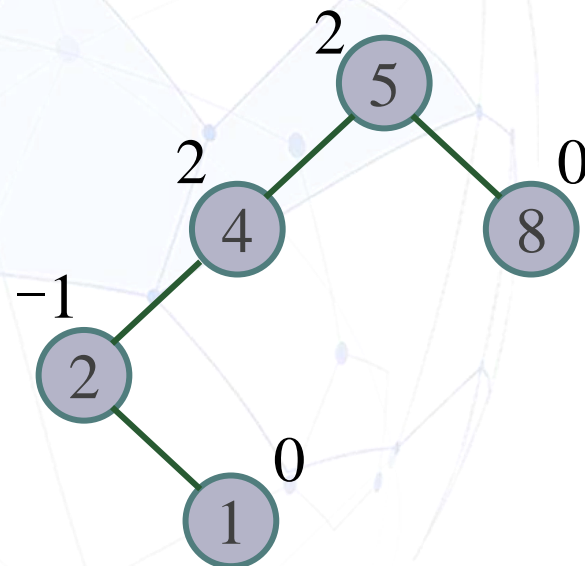
✚ 平衡二叉树：或者是一棵空的二叉排序树，或者是具有下列性质的二叉排序树：

- (1) 根结点的左子树和右子树的深度最多相差1；
- (2) 根结点的左子树和右子树也都是平衡二叉树

在平衡二叉树中，
结点的平衡因子是1、0或-1



平衡二叉树



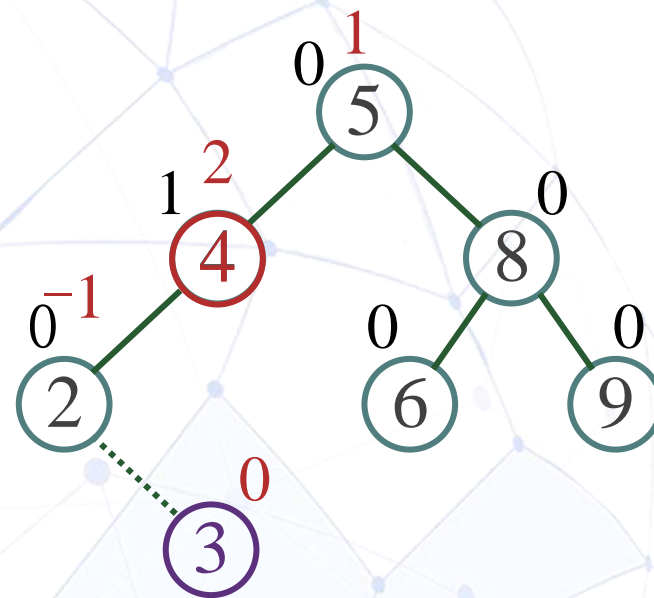
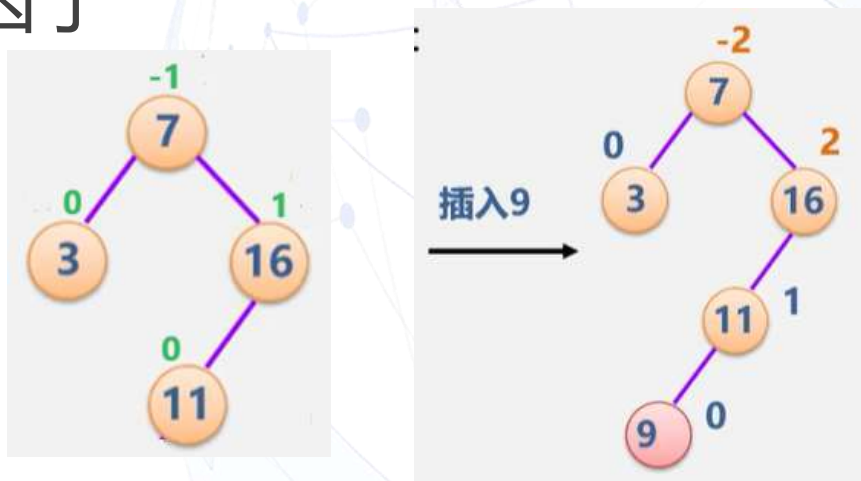
非平衡二叉树

对于一棵有 n 个结点的AVL树，其高度保持在 $O(\log_2 n)$ 数量级，

最小不平衡子树

🕒 插入结点3会影响哪些结点的平衡因子？

🕒 插入结点有时会产生多个绝对值大于1的平衡因子



📌 最小不平衡子树：以距离插入结点最近的、且平衡因子的绝对值大于 1 的结点为根的子树

📌 最小不平衡点：最小不平衡子树的根结点（关键点）

最小不平衡子树

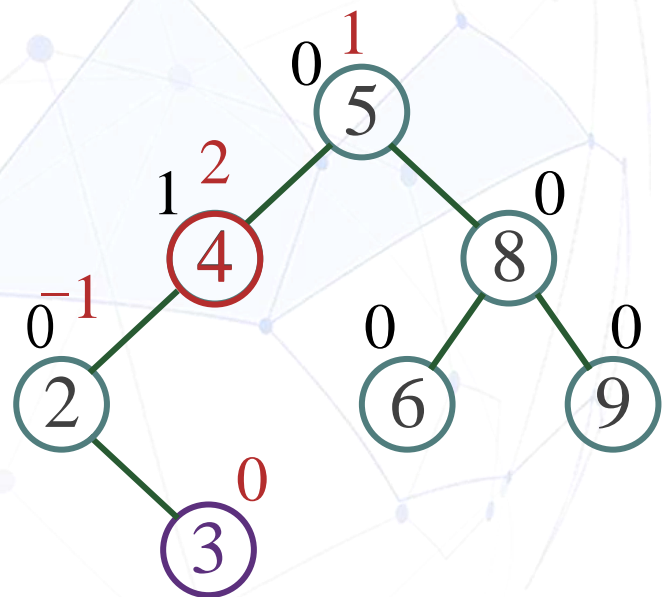
📌 最小不平衡点：最小不平衡子树的根结点（关键点）

且插入且判断，一旦失衡立即调整



只调整最小不平衡子树，并且不影响其他结点

🕒 如何调整最小不平衡子树呢？



平衡调整算法

算法：平衡调整

输入：平衡二叉树，新插入结点A

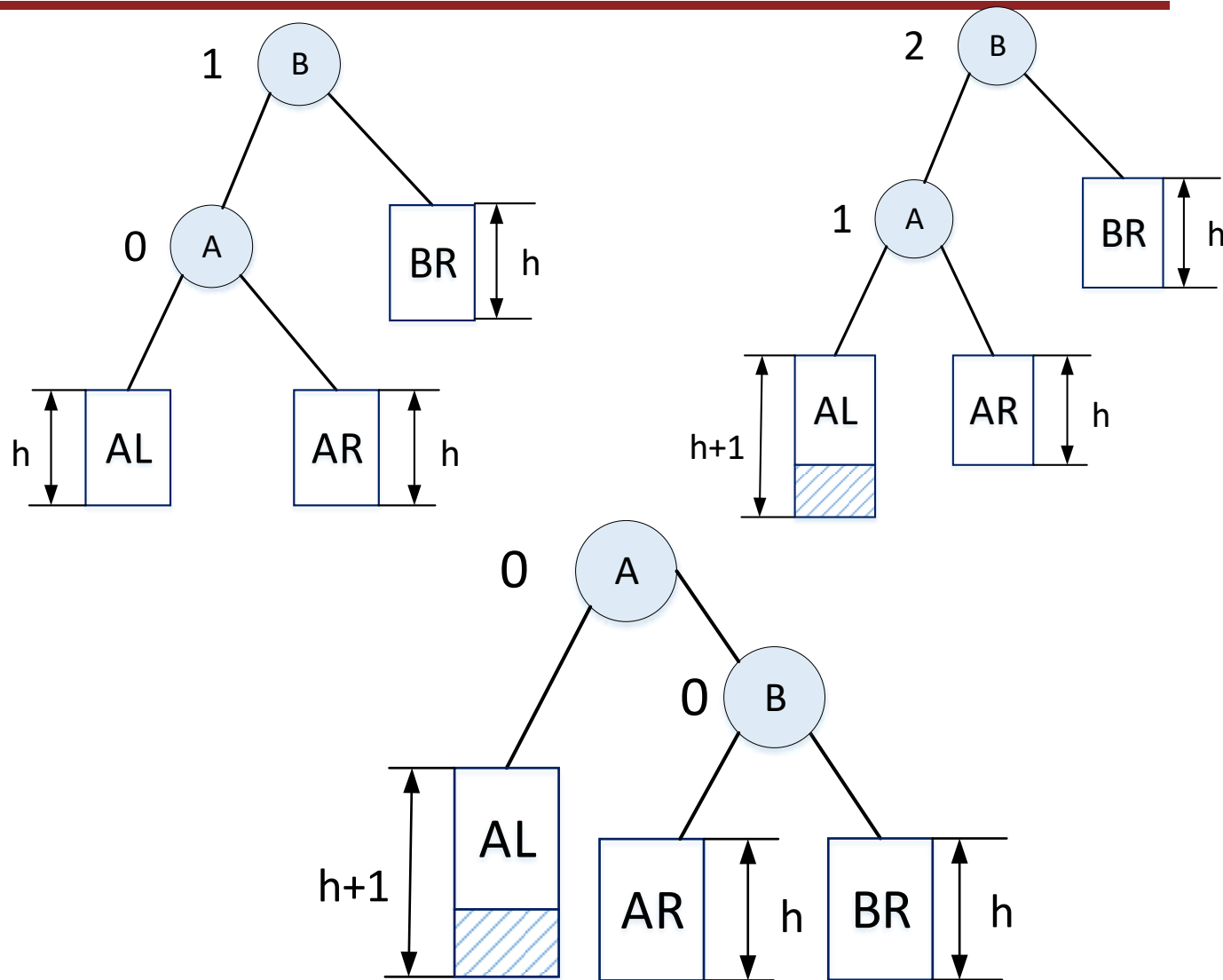
输出：新的平衡二叉树

1. 找到最小不平衡点 D;
2. 根据结点A和结点D之间的关系，判断调整类型;
3. 根据不同类型进行相应调整
 - (1) LL型、RR型：调整一次;
 - (2) LR型、RL型：调整两次;



LL型，在左孩子的左子树上插入

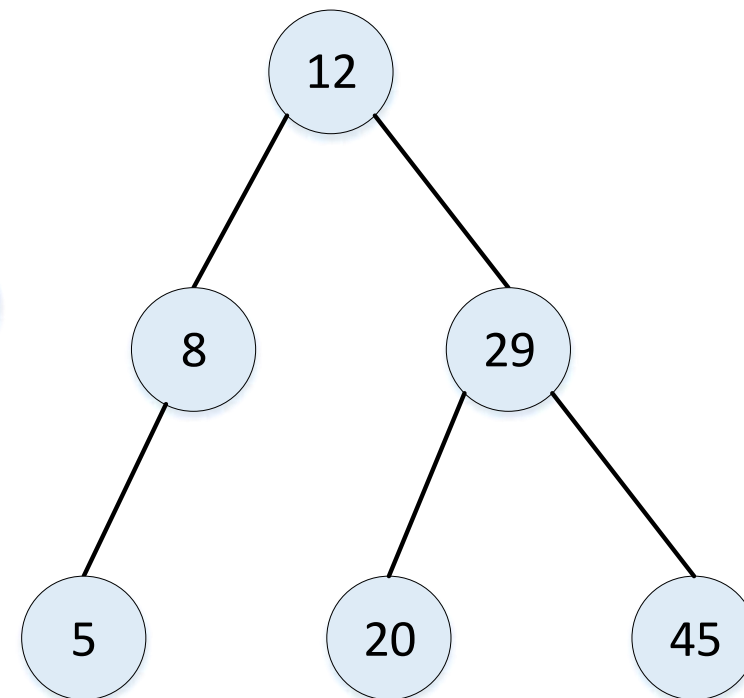
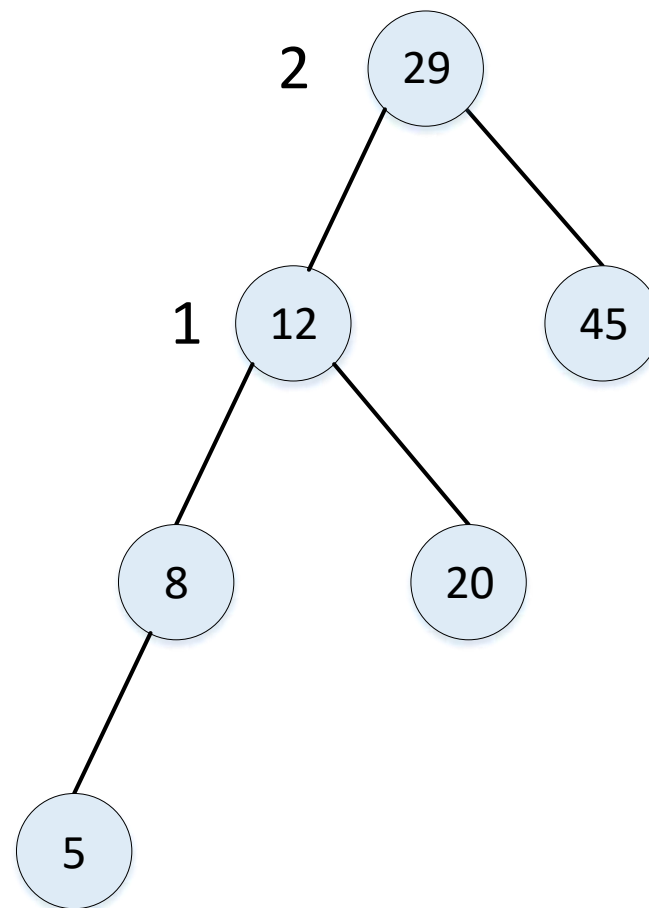
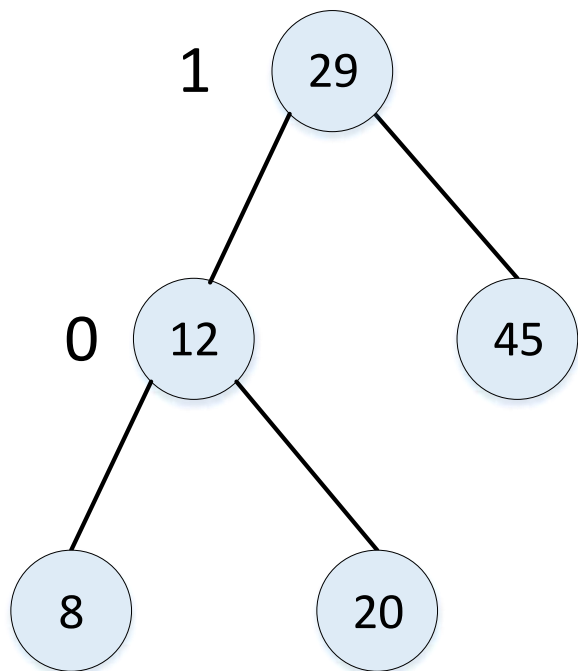
- 假设B的右子树高度和A的左、右子树高度都为 h ,
- 在B结点的左孩子A的左子树上插入了一个结点，使得B结点的平衡因子变为2，失去了平衡。
- 将B连接到A的分支以A为轴心作顺时针旋转，即A成为新的根，B降为A的右孩子，且A原来的右子树AR部分变为B的左子树。





LL示例

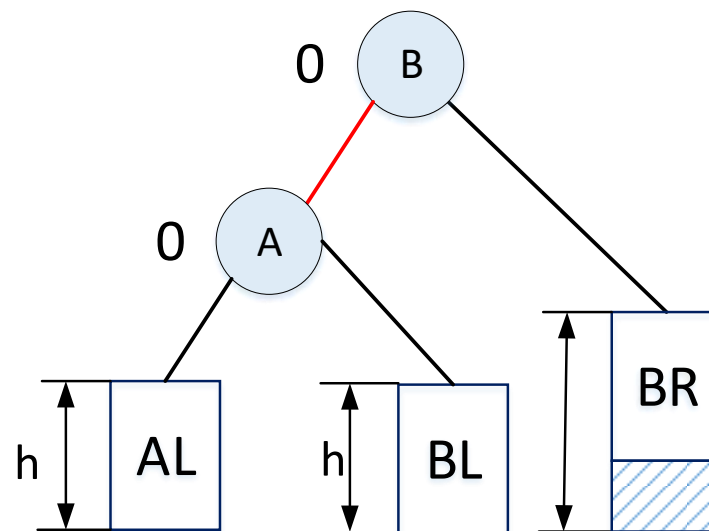
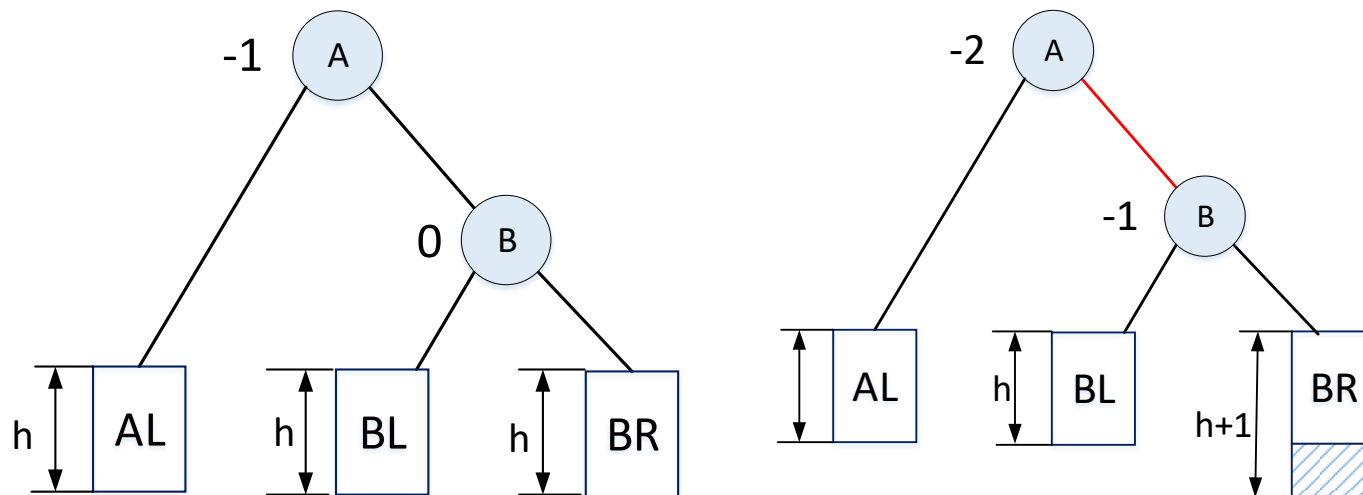
插入5





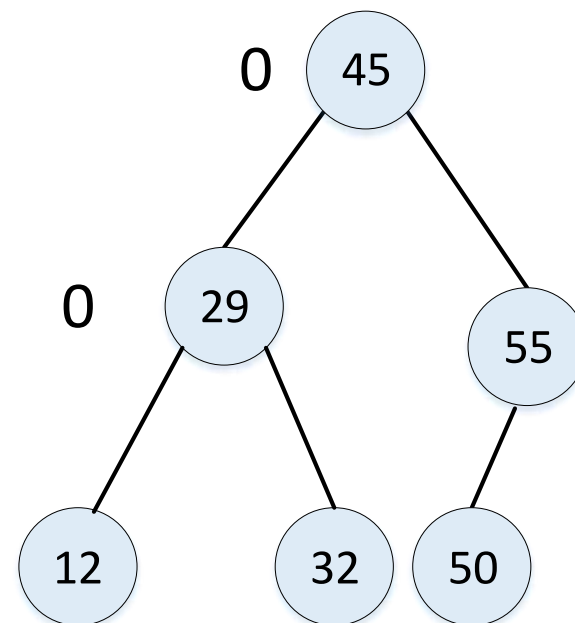
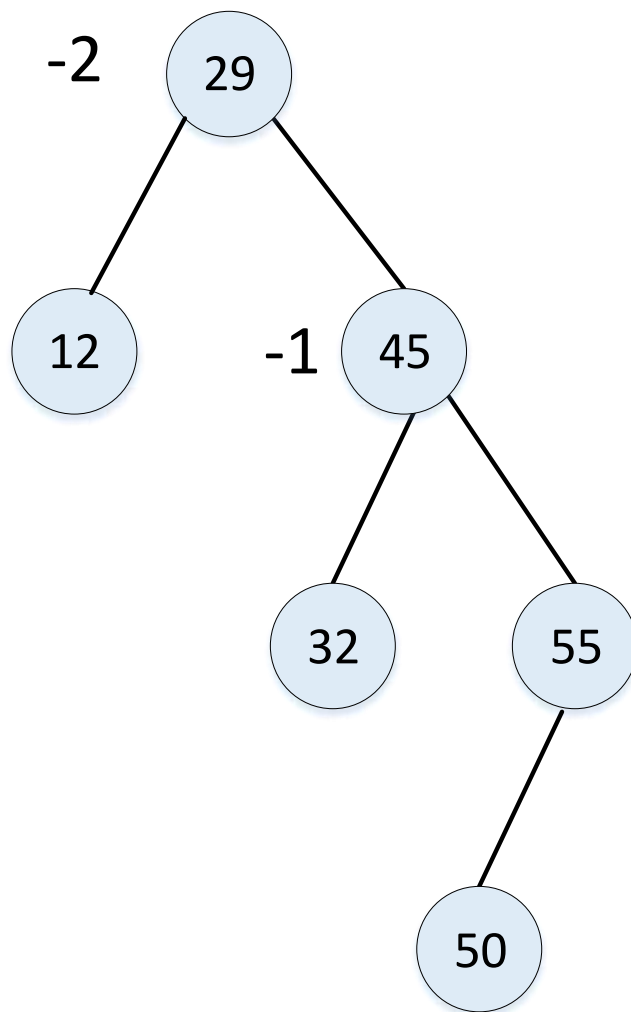
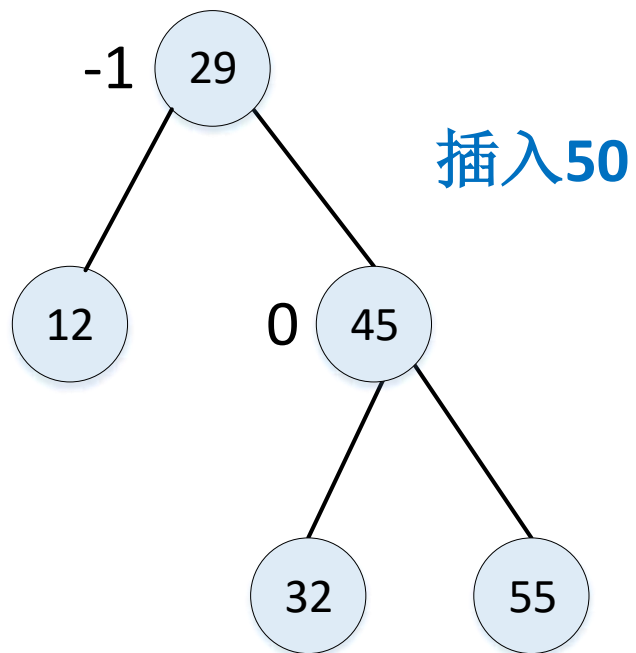
RR型，在右孩子的右子树上插入

- 在A结点的右孩子的右子树上插入了一个结点，使得A结点的平衡因子变为-2，失去平衡
- 将B连接到A的分支以B为轴心作**逆时针**旋转，即B成为新的根，A降为B的左孩子，且B原来的左子树BL部分变为A的右子树。





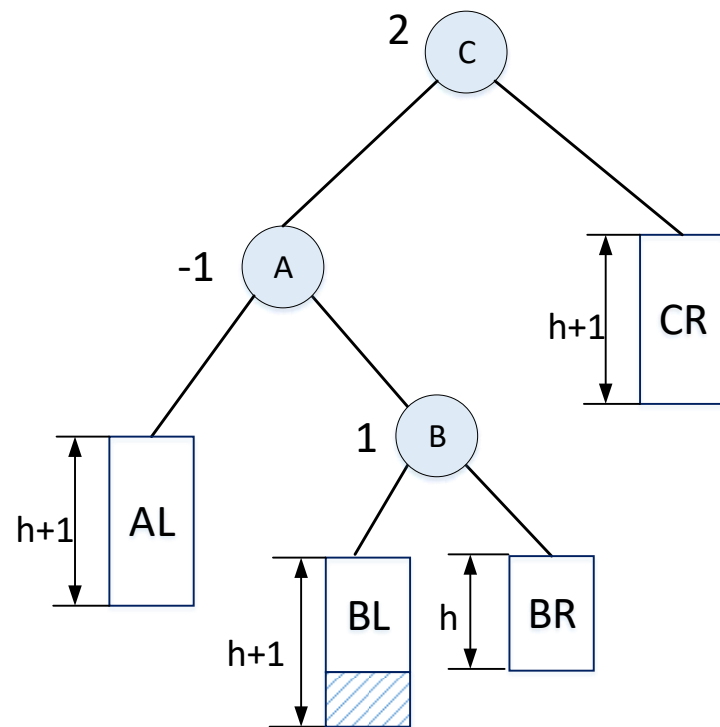
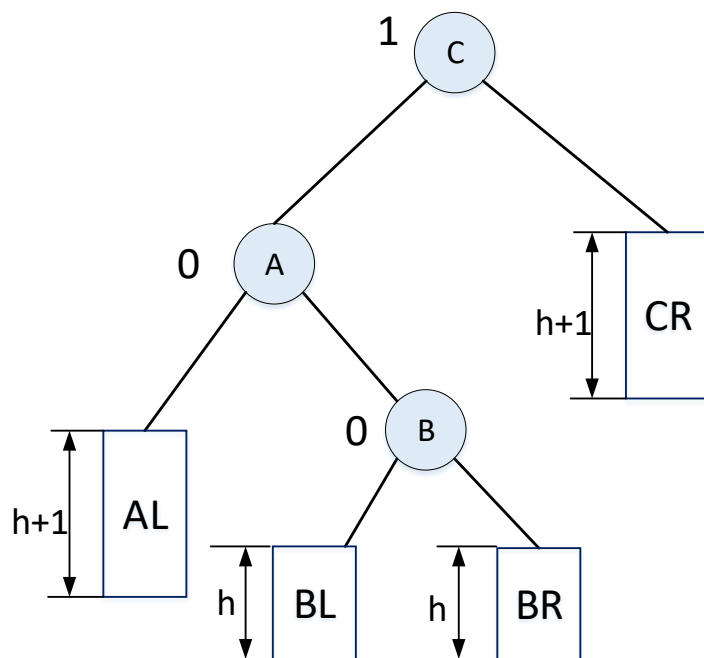
RR示例





LR型，在左孩子的右子树上插入

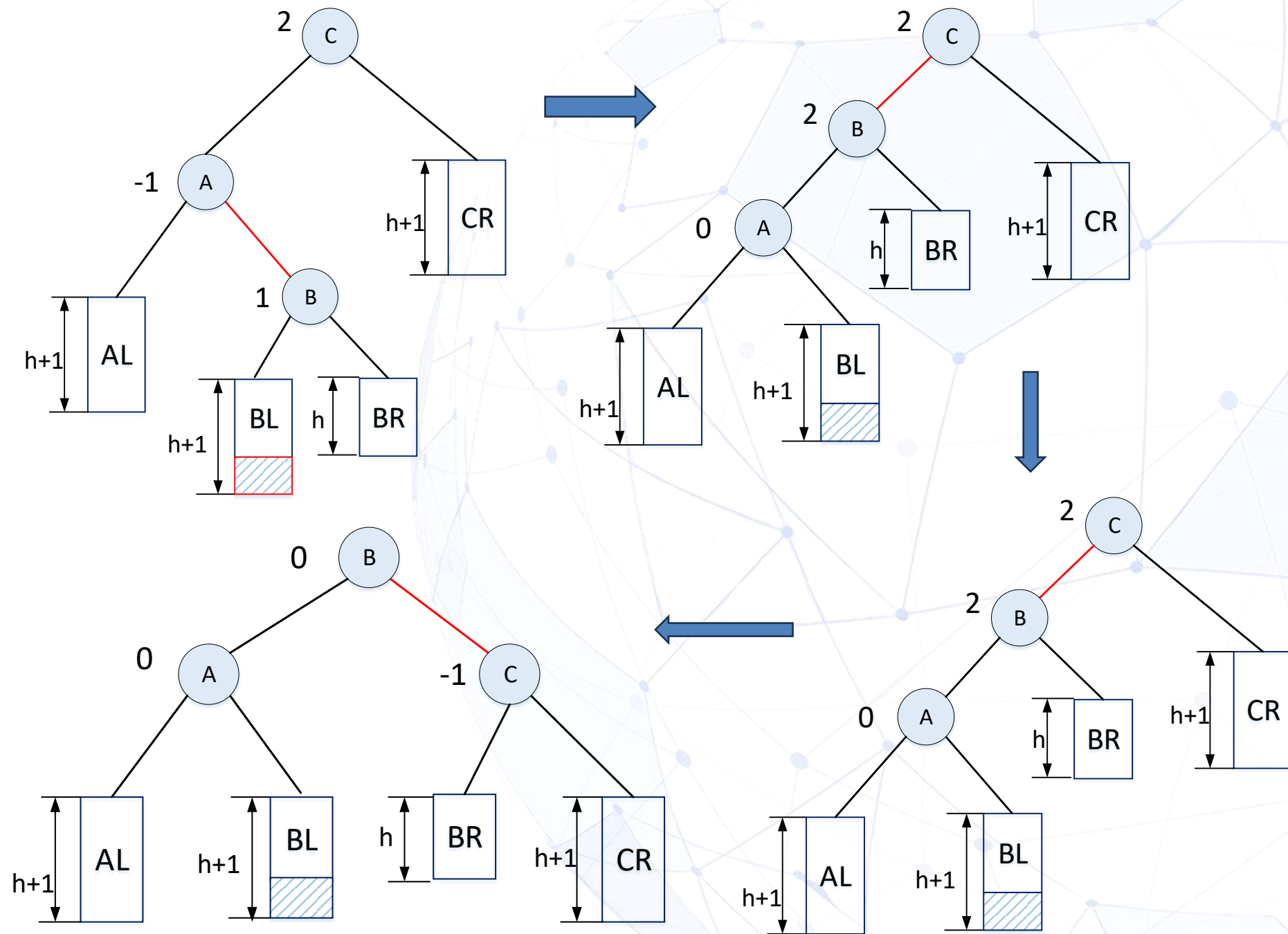
- 结点C的左孩子A的右子树上插入了一个结点，使得C结点的平衡因子变为2。
- 需作2次旋转，先逆再顺



LR型

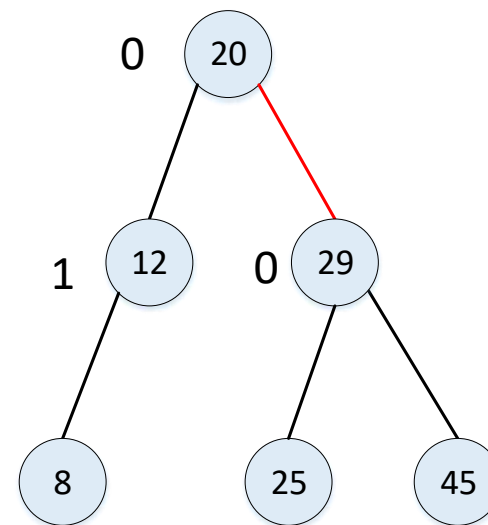
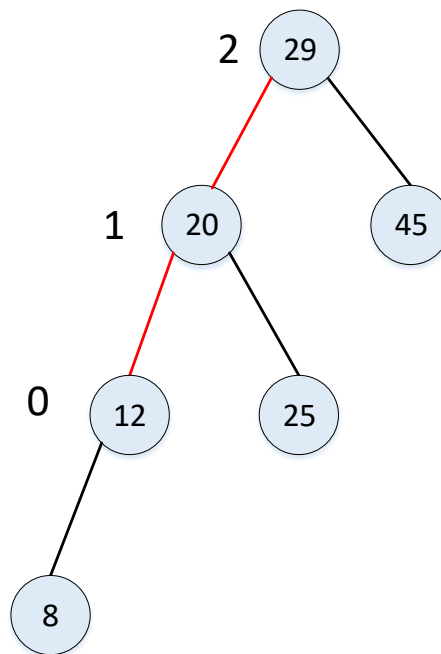
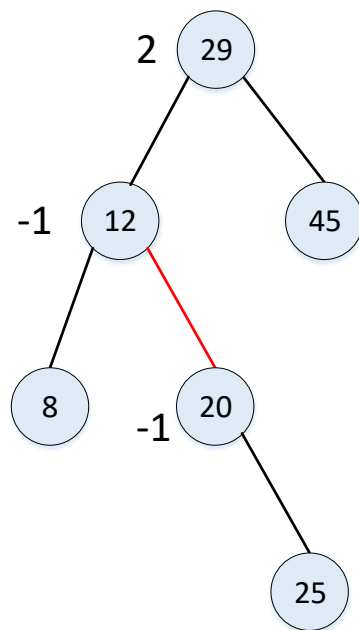
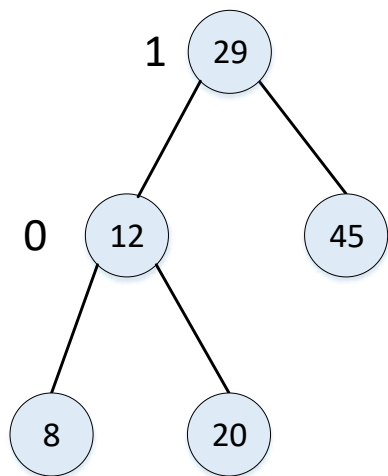
1) 将A连接到B的分支以A为轴心作**逆时针**旋转，即B成为A的双亲，B原来的左子树BL部分变为A的右子树。

2) 将B连接到C的分支以B为轴心作**顺时针**旋转，即B成为新的根结点，C成为B的右孩子，B原来的右子树BR部分变为C的左子树。





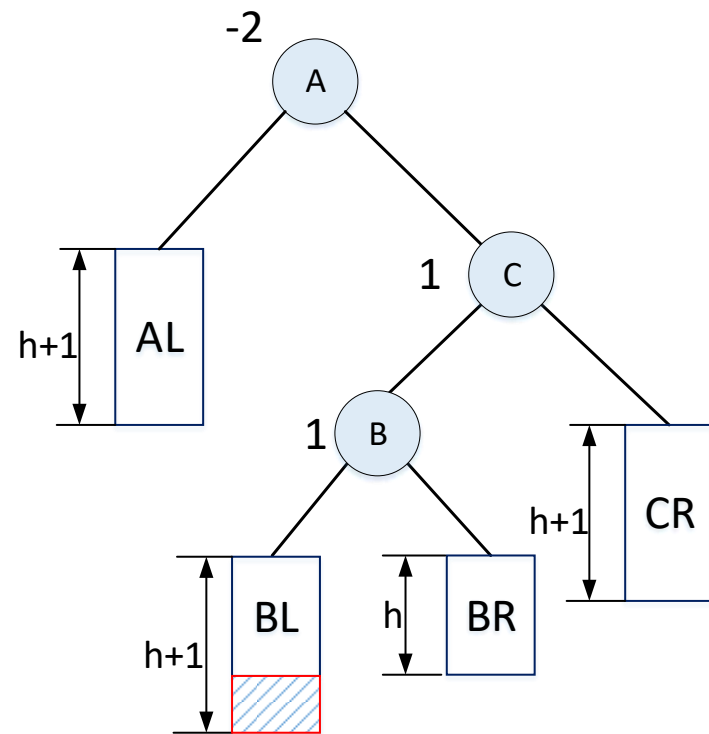
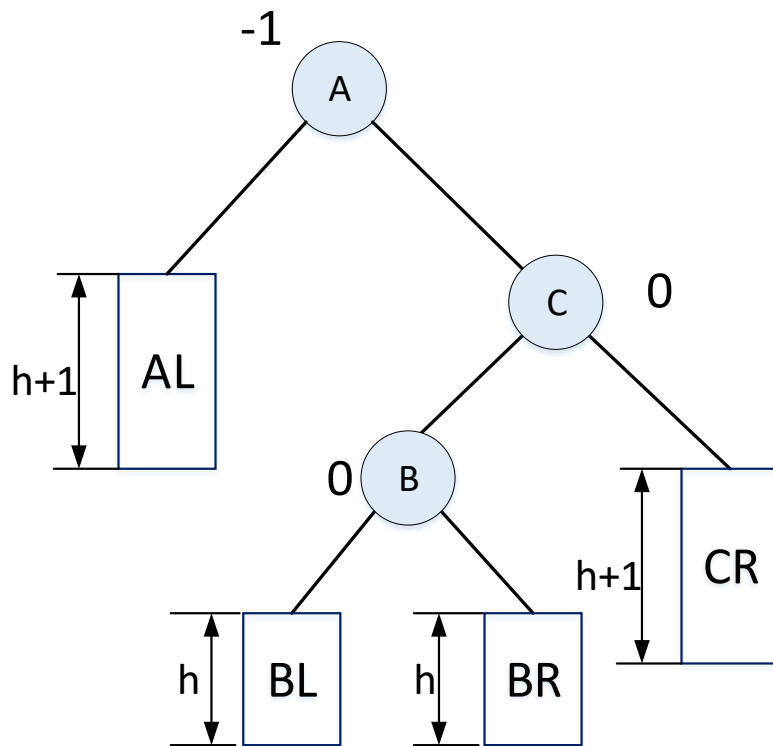
LR示例





RL型，在右孩子的左子树上插入

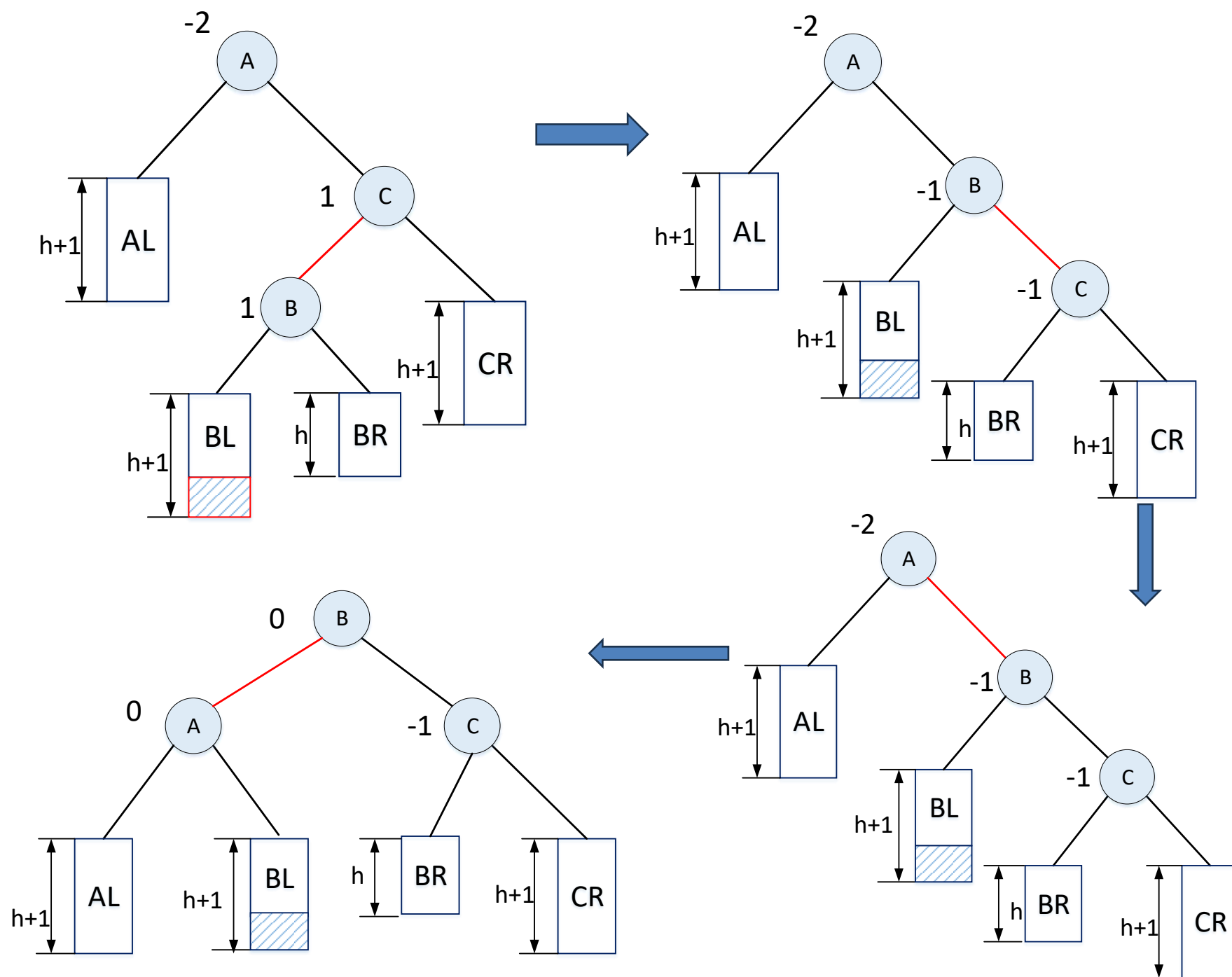
- 在根结点A的右孩子C的左子树上插入了一个结点，使得A结点的平衡因子变为-2；
- 需要作2次旋转，先顺再逆



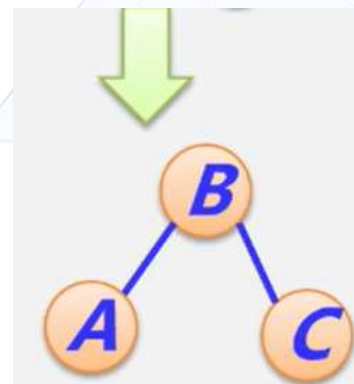
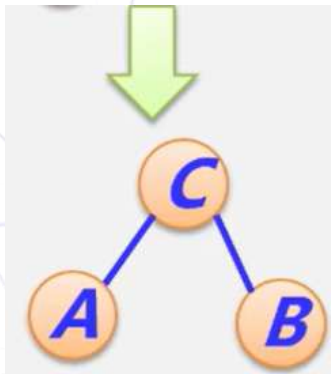
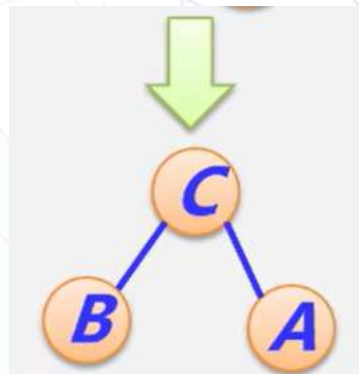
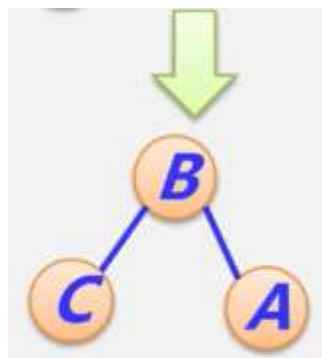
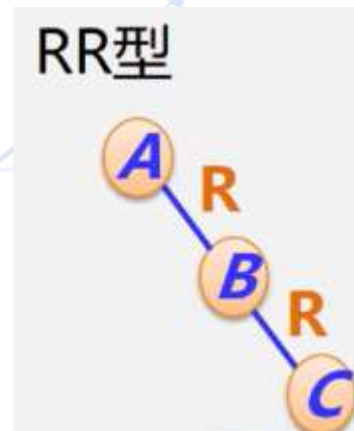
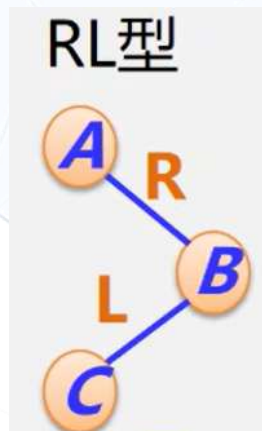
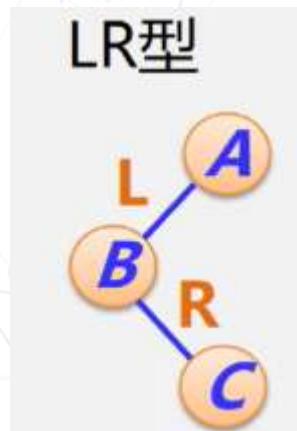
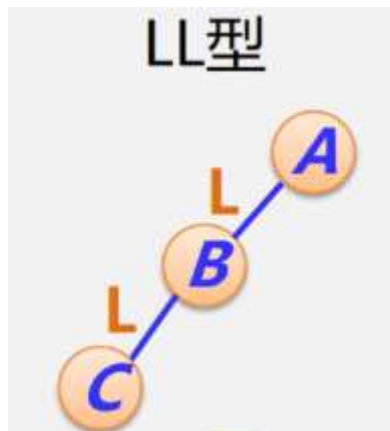
RL型

1) 先看A的子树部分，将C连接到B的分支以C为轴心作**顺时针**旋转，即B成为C的双亲，B原来的右子树BR部分变为C的左子树。

2) 再将B连接到A的分支以B为轴心作**逆时针**旋转，即B成为新的根结点，C成为B的右孩子，B原来的右子树BR部分变为C的右子树。



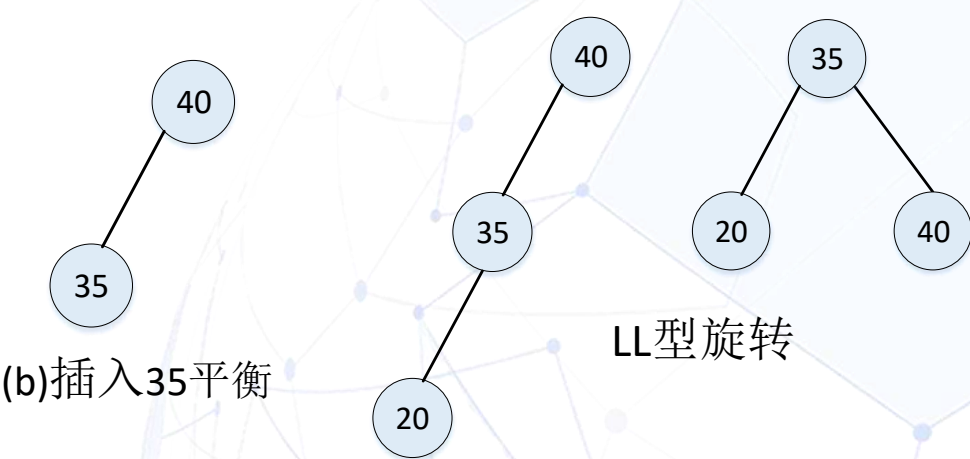
平衡调整的4种类型（假设C是待插入结点）



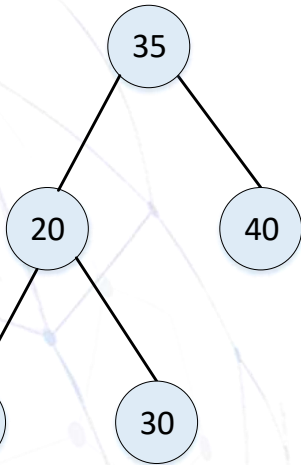
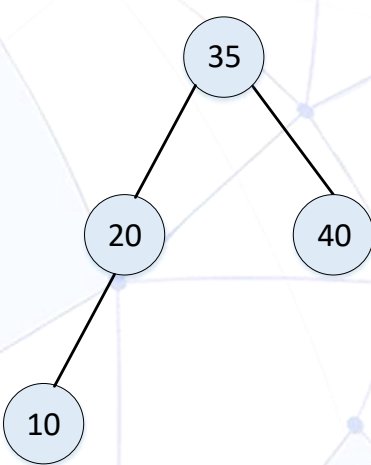
调整原则：1) 降低高度 2) 保持二叉排序树性质

中间值

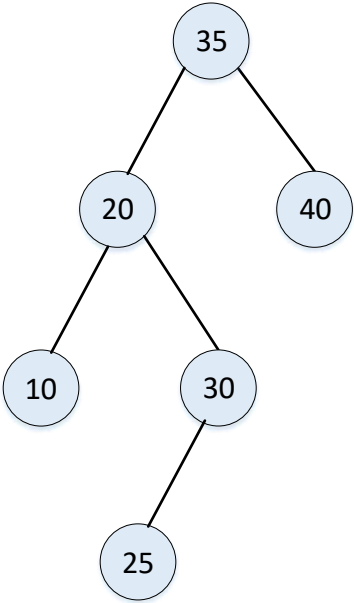
设有关键字序列(40, 35, 20, 10, 30, 25, 38)，给出平衡二叉树的构造过程。



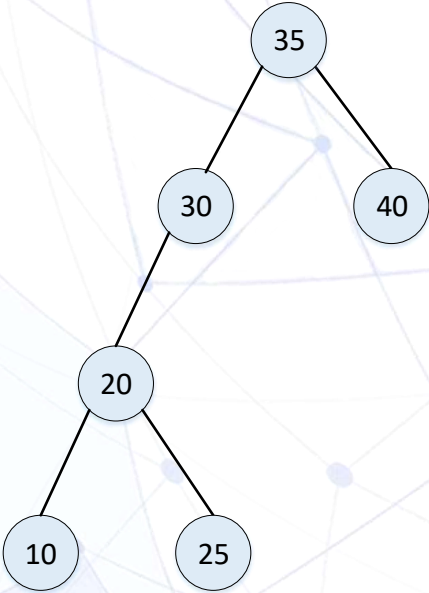
(c)插入20不平衡



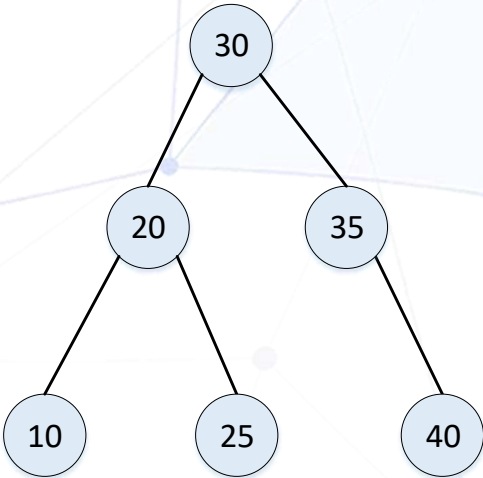
(e)插入30平衡



(f)插入25



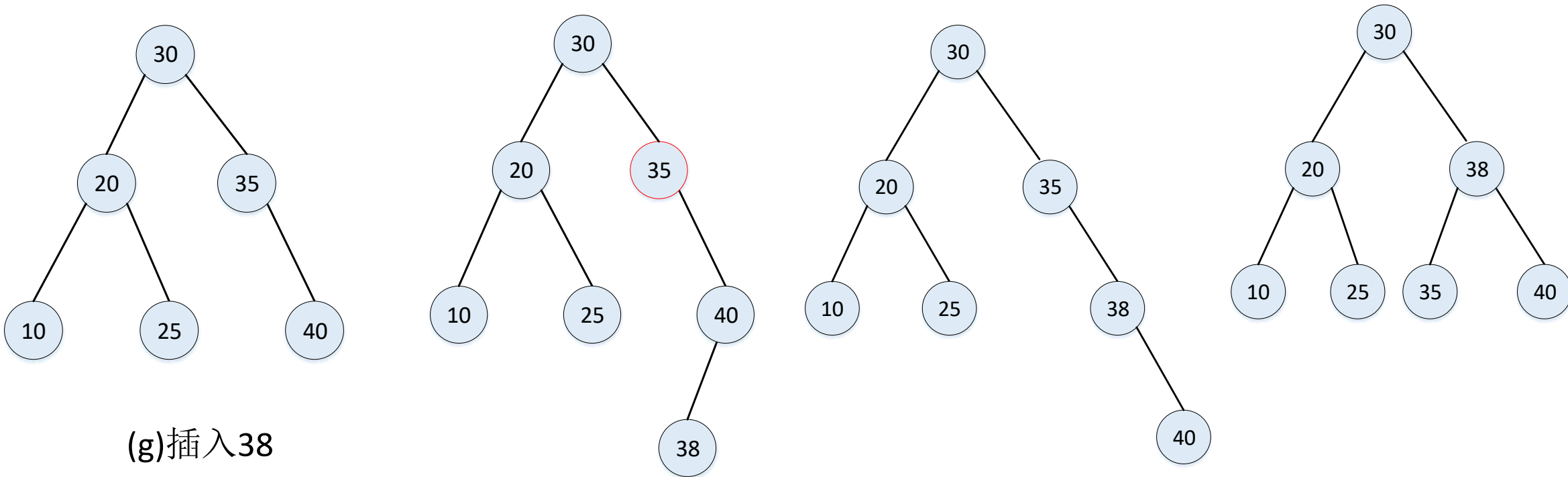
LR型旋转，
在35的左孩子的
右子树上插入





AVL树构造示例

设有关键字序列(40, 35, 20, 10, 30, 25, **38**), 给出了平衡二叉树的构造过程。


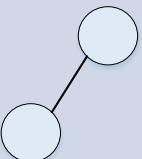
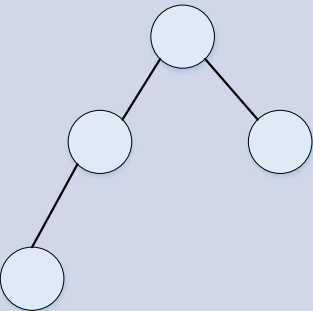
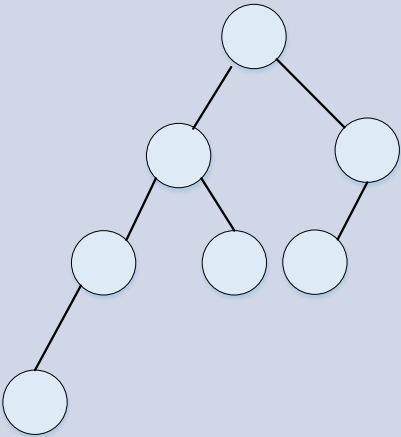
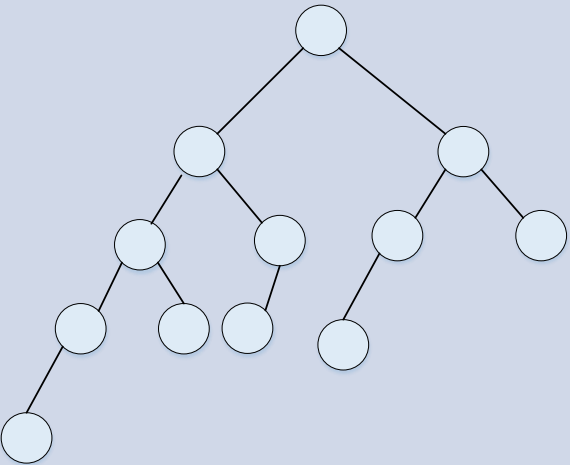


即在35的右孩子的左子树上插入, RL型旋转



AVL树结点数的最大高度

- 一棵最差的AVL树中非叶子结点的平衡因子都为1或者-1;
- 高度1至5, 非叶子结点平衡因子都为1的“左重”AVL树及对应结点数, 见下表;
- 设高度为 $h(h \geq 3)$ 的AVL树总结点为 N_h , 则存在递归公式 $N_h = N_{h-1} + N_{h-2} + 1$, 类似于斐波那契数列第 h 个数的计算, 可以得到 $h = 1.44 \log_2 N_h$ 。因此, **AVL树下的查找时间复杂度为 $O(\log_2 n)$ 。**

高度 h	1	2	3	4	5
左重平衡 二叉树					
结点数 N_h	1	2	4	7	12



不同结构查找表下算法性能比较

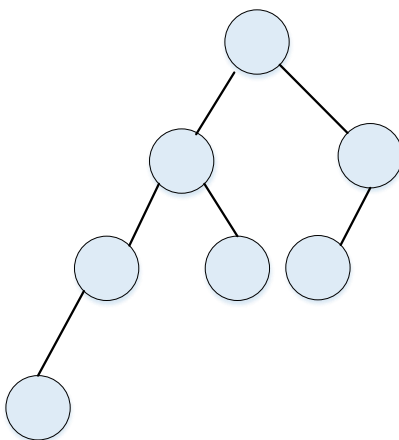
	无序顺序表	有序顺序表	二叉查找树	AVL树
查找	$O(n)$	$O(\log_2 n)$	$O(\log_2 n) \rightarrow O(n)$	$O(\log_2 n)$
插入	$O(1)$	$O(n)$	$O(\log_2 n) \rightarrow O(n)$	$O(\log_2 n)$
删除	$O(1)/O(n)$	$O(n)$	$O(\log_2 n) \rightarrow O(n)$	$O(\log_2 n)$



1. 相同结点个数的平衡二叉树和完全二叉树的高度相同。

A 正确

B 错误

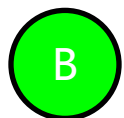


提交

2. 构造平衡二叉树的过程是，首先将查找集合构造二叉排序树，再进行整体调整。



正确



错误

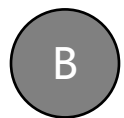
提交

3. 在平衡二叉树中进行平衡调整时，只涉及最小不平衡子树，不会涉及其他结点。



A

正确

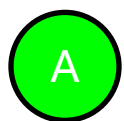


B

错误

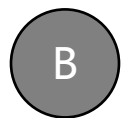
提交

4. 对于平衡二叉树的调整，LL型的对称类型是（ ）。



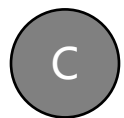
A

RR型



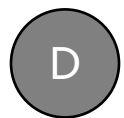
B

LR型



C

RL型



D

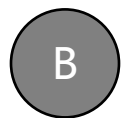
没有对称类型

提交

5. 在平衡二叉树的调整过程中，如果结点之间的关系发生冲突，则原来的关系将会被破坏。



正确



错误

提交

6. 对于查找集合{25, 12, 18, 35, 40, 30}构造平衡二叉树，写出插入每一个元素的结果。



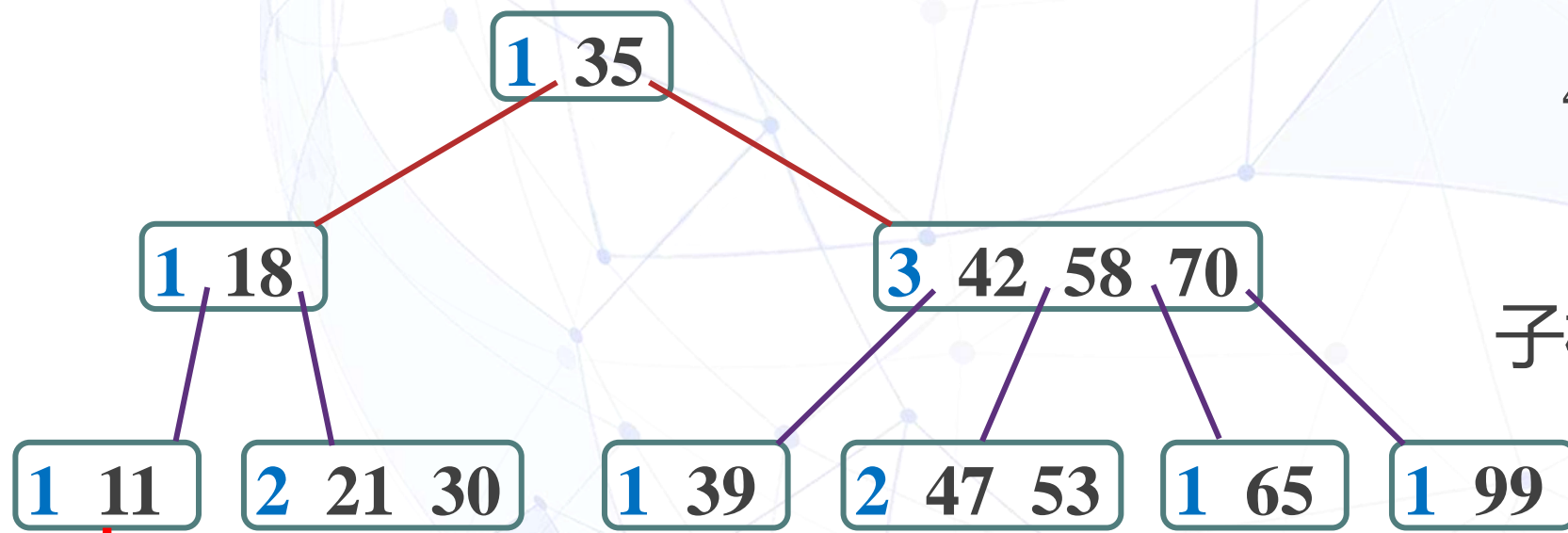
第七章 查找技术

7-3-3 B 树

B树的定义

📌 B树：一棵 m 阶的B树或者为空树，或者为满足下列特性的 m 叉树：

- (1) 每个结点至多有 m 棵子树
- (2) 根结点至少有两棵子树
- (3) 除根结点外，所有结点至少有 $\lceil m/2 \rceil$ 棵子树



4 阶 B 树



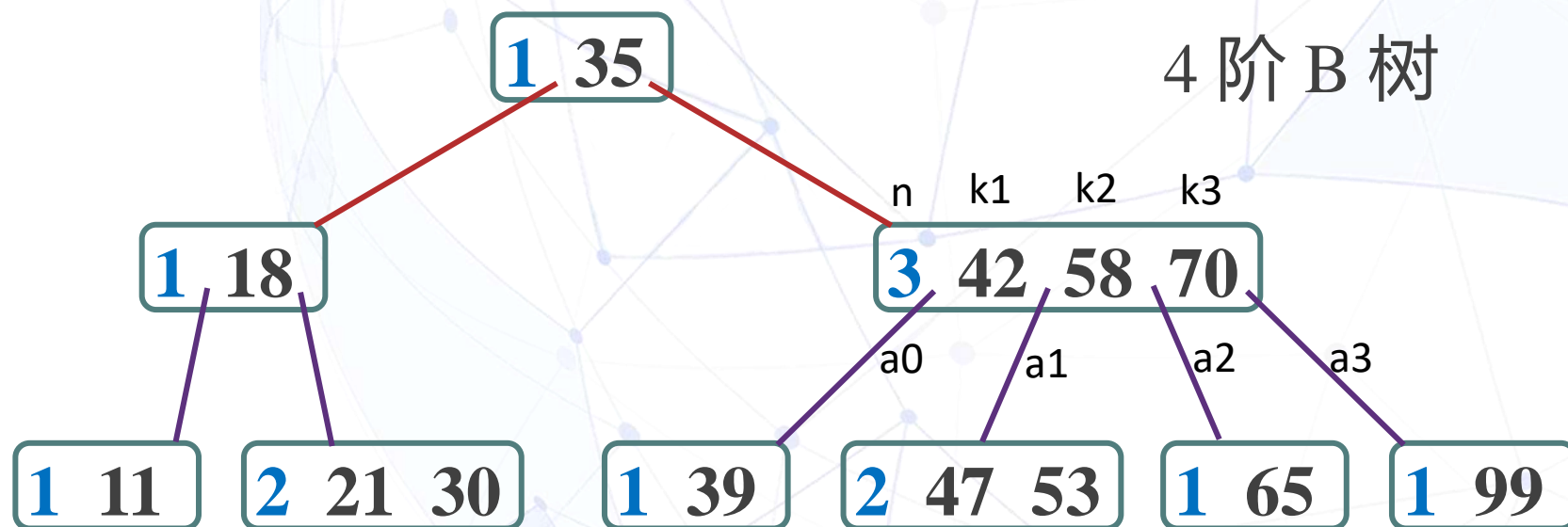
子树个数 2 ~ 4

注意这个结点有2棵子树

B树的定义

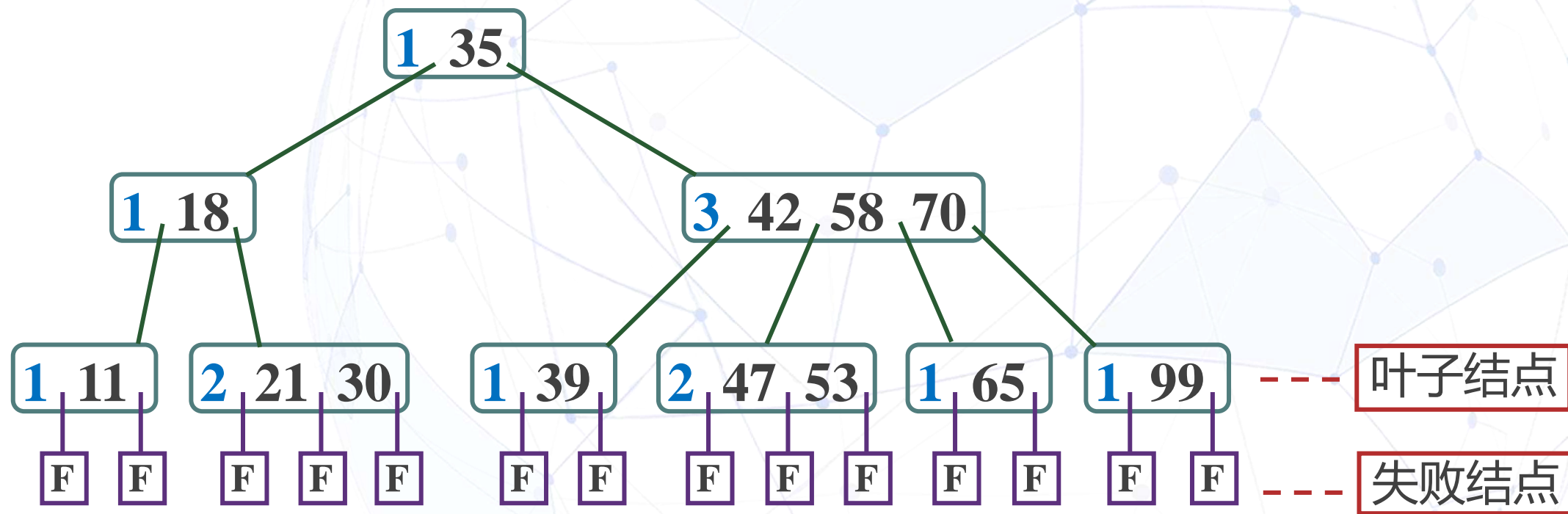
📌 B树：一棵 m 阶的B树或者为空树，或者为满足下列特性的 m 叉树：

(4) 所有结点都包含以下数据： $(n, A_0, K_1, A_1, K_2, \dots, K_n, A_n)$ ，其中， n ($\lceil m/2 \rceil - 1 \leq n \leq m - 1$) 为关键码的个数， K_i ($1 \leq i \leq n$) 为关键码，且 $K_i < K_{i+1}$ ($1 \leq i \leq n-1$)， A_i ($0 \leq i \leq n$) 为指向子树根结点的指针，且指针 A_i 所指子树中所有结点的关键码均小于 K_{i+1} 大于 K_i



B树的定义

📌 **B树**：一棵 m 阶的B树或者为空树，或者为满足下列特性的 m 叉树：
(5) 叶子结点都在同一层



叶子结点都在同一层



树高平衡



具有较高的查找效率

B树的特征

🕒 在 B 树中，结点的值有什么特征？

A_i 所指子树中所有结点的关键码均小于 K_{i+1} 大于 K_i

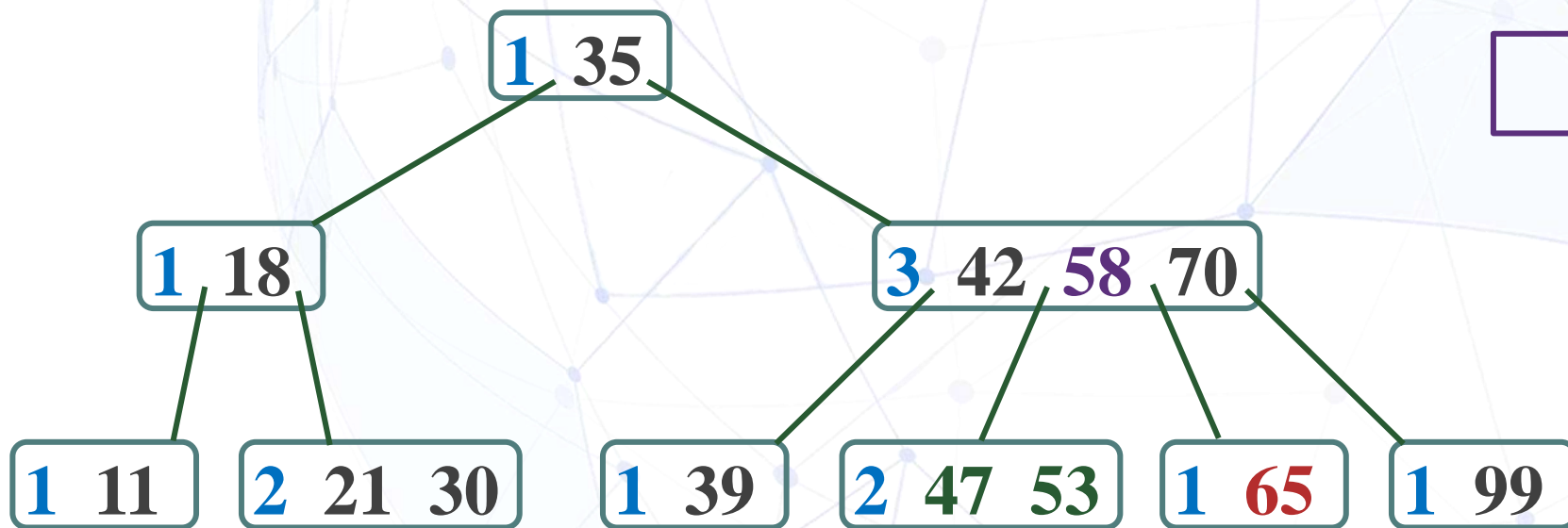
二叉排序树



2-3 树



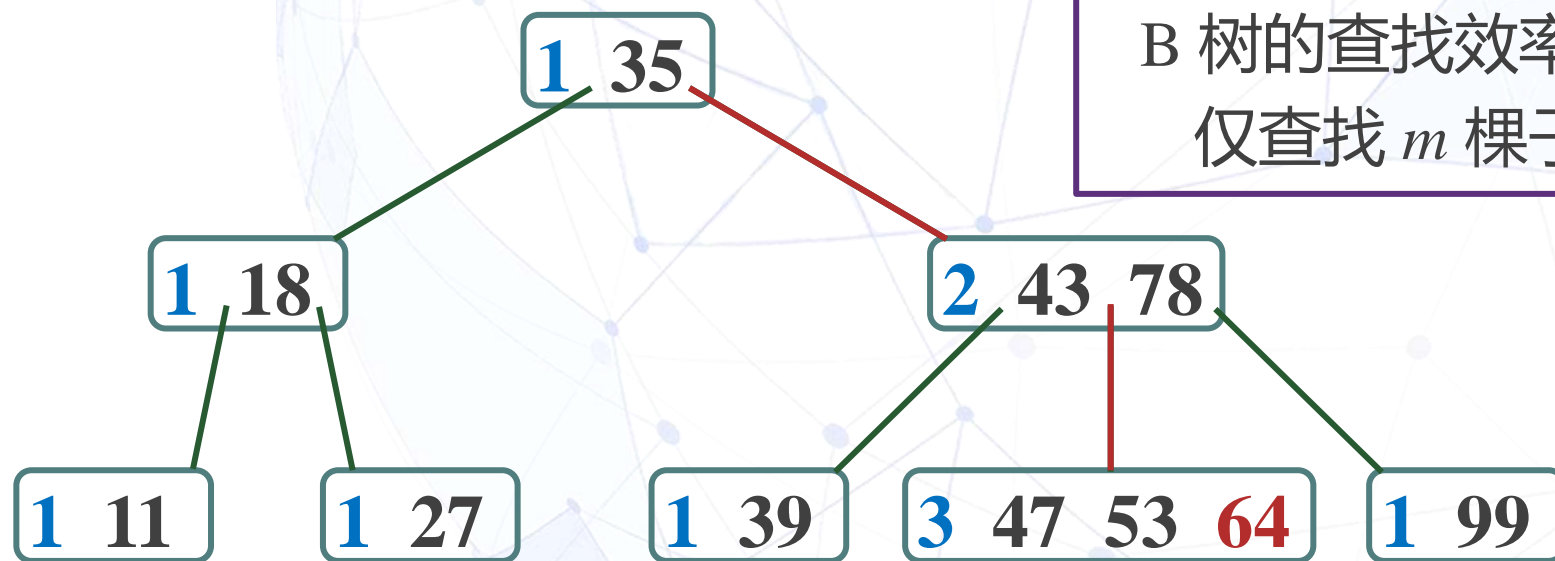
B 树



B树的查找



B 树的查找过程：

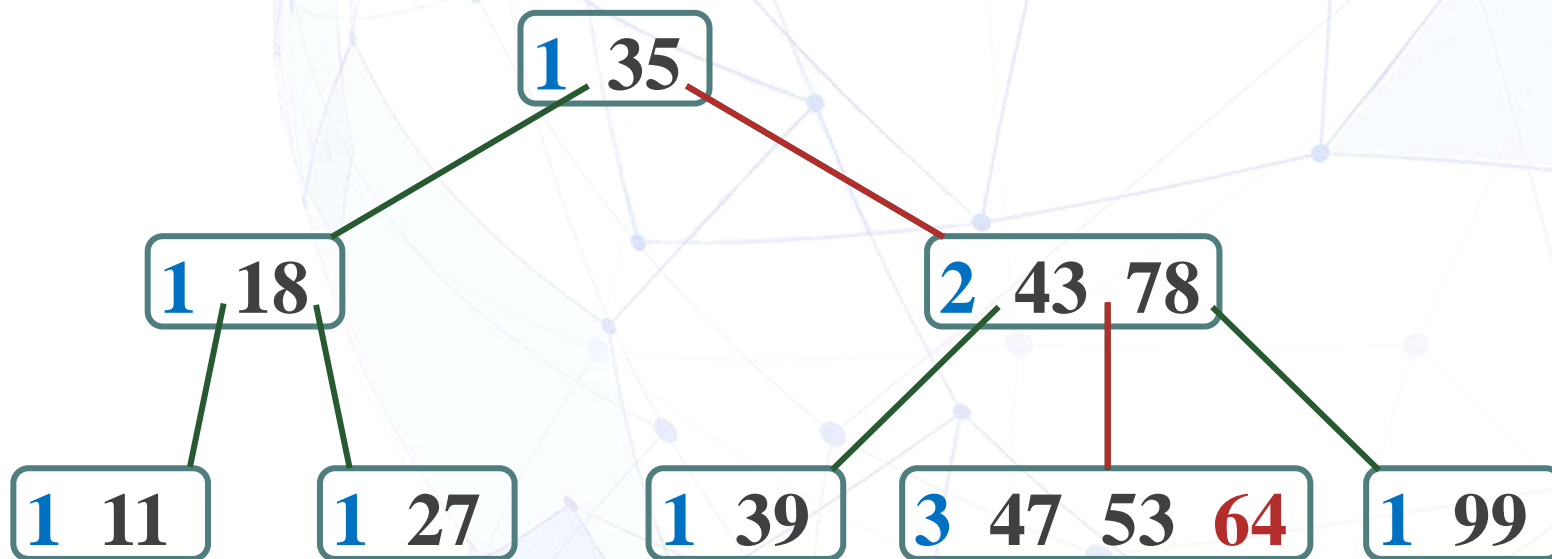
- (1) 顺指针查找结点：按照指针到相应的子树中查找；
- (2) 在结点中查找关键码：若在结点内找到，则查找成功；
- (3) 重复上述两步，到达外部结点时，查找失败。



B 树的查找效率就体现在
仅查找 m 棵子树之一

B树的查找

-  B 树通常存储在磁盘上，则顺指针查找结点是在磁盘上进行，结点内查找操作是在内存中进行，即在磁盘上找到某结点后，先将结点的信息读入内存，然后再查找等于 k 的关键码。
-  显然，在磁盘上进行一次查找比在内存中进行一次查找耗费的时间多得多，因此，在磁盘上进行查找的次数，是决定 B 树查找效率的首要因素。



B树的深度

🕒 含有 n 个关键码的 m 阶 B 树，最坏情况下的深度是多少呢？

第一层至少有1个结点 ← 根结点

第二层至少有2个结点 ← 根结点至少有两棵子树

第三层至少有 $2 \left\lceil \frac{m}{2} \right\rceil$ 个结点 ← 非终端结点至少有 $\left\lceil \frac{m}{2} \right\rceil$ 棵子树

以此类推，第 $k+1$ 层至少有 $2 \left(\left\lceil \frac{m}{2} \right\rceil \right)^{k-1}$ 个结点

若 m 阶 B 树有 n 个关键码，则第 $k+1$ 层有 $n+1$ 个结点（失败结点）

↑
第 $k+1$ 层为查找不成功的结点

$$n + 1 \geq 2 \left(\left\lceil \frac{m}{2} \right\rceil \right)^{k-1}$$

B树的深度

🕒 含有 n 个关键码的 m 阶 B 树，最坏情况下的深度是多少呢？

$$n+1 \geq 2\left(\left\lceil \frac{m}{2} \right\rceil\right)^{k-1} \Rightarrow k \leq \log_{\lceil m/2 \rceil} \left(\frac{n+1}{2}\right) + 1$$

因此，含有 n 个关键码的 m 阶 B 树的最大深度是 $\log_{\lceil m/2 \rceil} \left(\frac{n+1}{2}\right) + 1$

🕒 含有 n 个关键码的 m 阶 B 树，最好情况下的深度是多少呢？

$$\sum_{i=1}^k m^{i-1} = \lfloor \log_m n \rfloor + 1$$



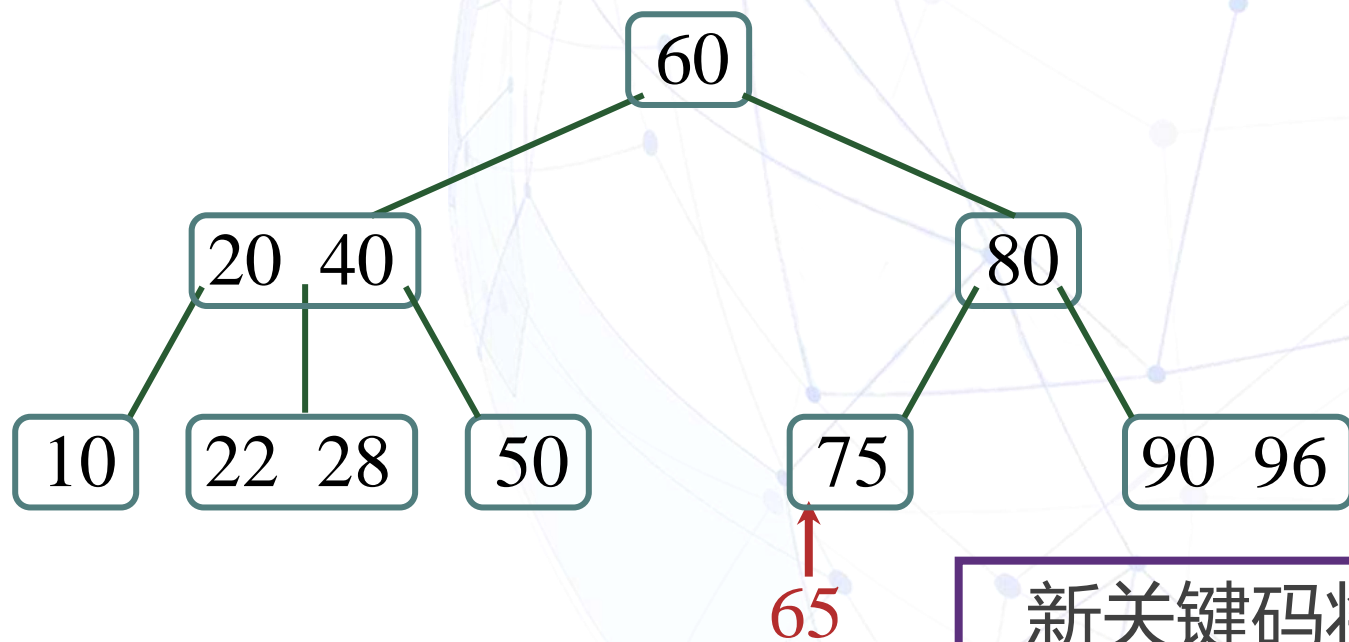
B树的插入

✈ 假定在 m 阶 B 树中插入关键码 key ，设 $n=m-1$ ，插入过程如下：

(1) **定位**：确定关键码 key 应该插入哪个叶子结点并返回该结点的指针 p 。

若 p 中的关键码个数小于 n ，则直接插入关键码 key ；

否则，结点 p 的关键码个数**溢出**，执行“分裂——提升”过程。



B 树是多少阶？



$m = 3, n = 2$

新关键码将插入到相应的**叶子**结点中

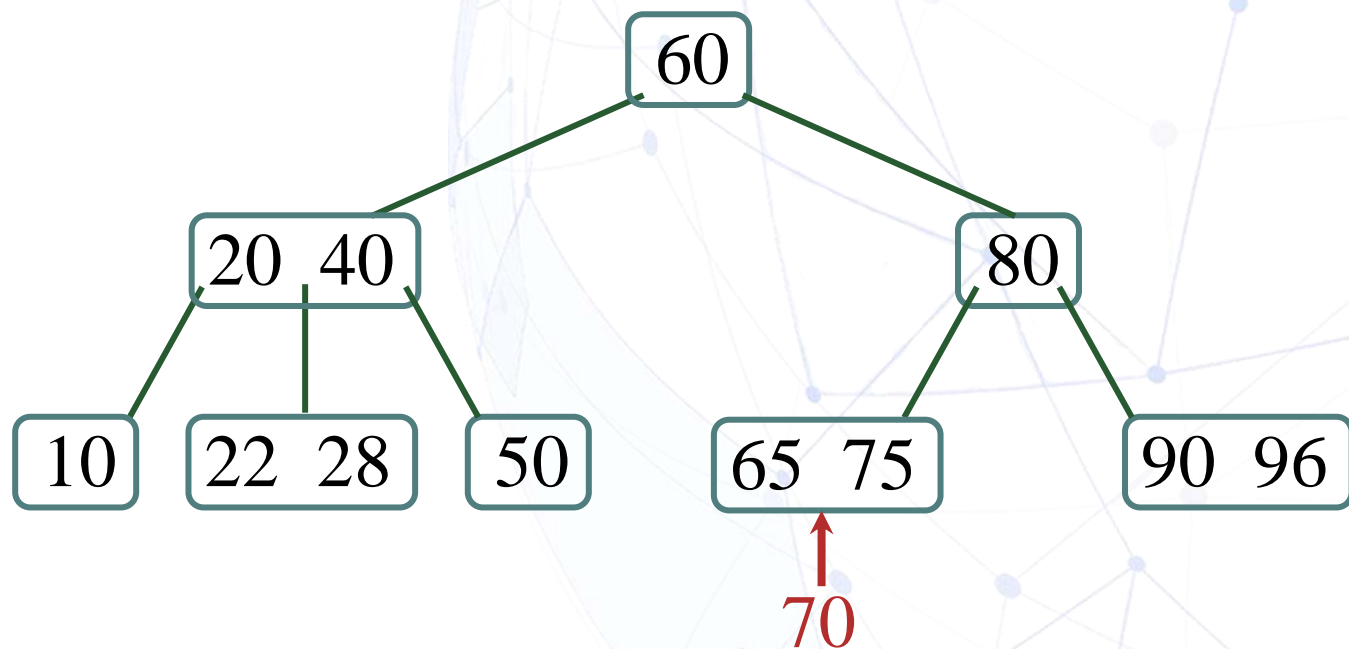
B树的插入

✈ 假定在 m 阶 B 树中插入关键码 key ，设 $n=m-1$ ，插入过程如下：

(1) **定位**：确定关键码 key 应该插入哪个终端结点并返回该结点的指针 p 。

若 p 中的关键码个数小于 n ，则直接插入关键码 key ；

否则，结点 p 的关键码个数**溢出**，执行“分裂——提升”过程。



B 树是多少阶？



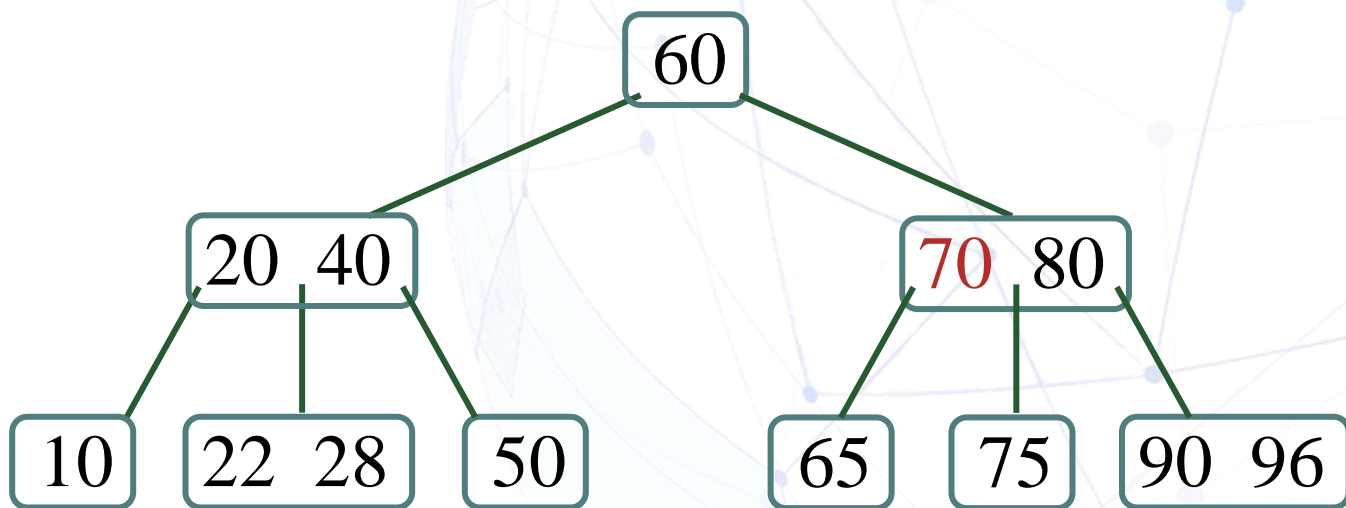
$m = 3, n = 2$

发生溢出

B树的插入

✈ 假定在 m 阶 B 树中插入关键码 key ，设 $n=m-1$ ，插入过程如下：

(2) 分裂——提升：将结点 p “分裂” 成两个结点，分别是 p_1 和 p_2 ，把中间的关键码 k “提升” 到父结点，并且 k 的左指针指向 p_1 ，右指针指向 p_2 。



B 树是多少阶？



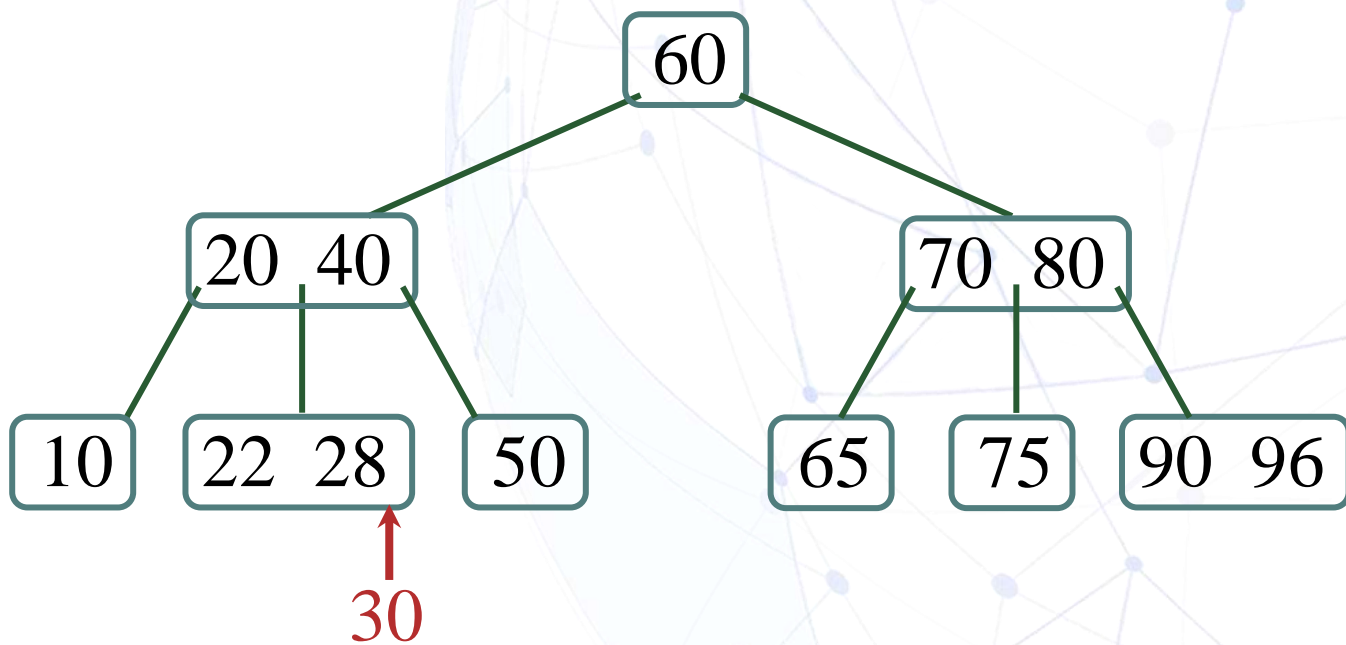
$m = 3, n = 2$

分裂——提升

B树的插入

✈ 假定在 m 阶 B 树中插入关键码 key ，设 $n=m-1$ ，插入过程如下：

(2) **分裂——提升**：如果父结点的关键码个数也溢出，则继续执行“分裂——提升”过程。显然，这种分裂可能一直上传，如果根结点也分裂了，则树的高度增加了一层。



B 树是多少阶？



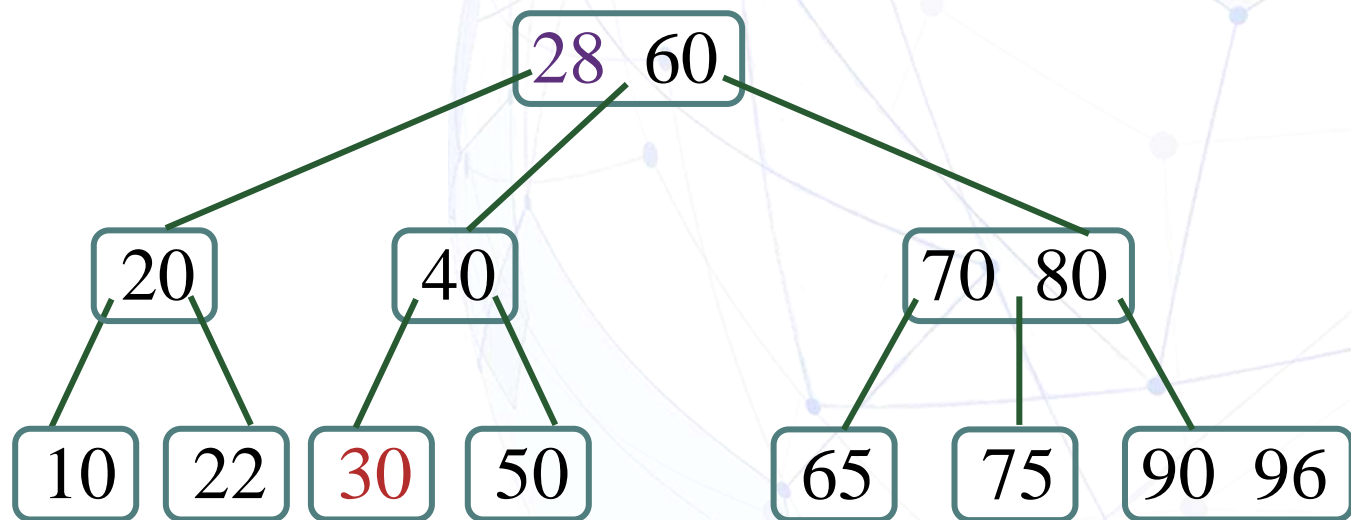
$m = 3, n = 2$

发生溢出

B树的插入

✈ 假定在 m 阶 B 树中插入关键码 key ，设 $n=m-1$ ，插入过程如下：

(2) 分裂——提升：如果父结点的关键码个数也溢出，则继续执行“分裂——提升”过程。显然，这种分裂可能一直上传，如果根结点也分裂了，则树的高度增加了一层。



B 树是多少阶？



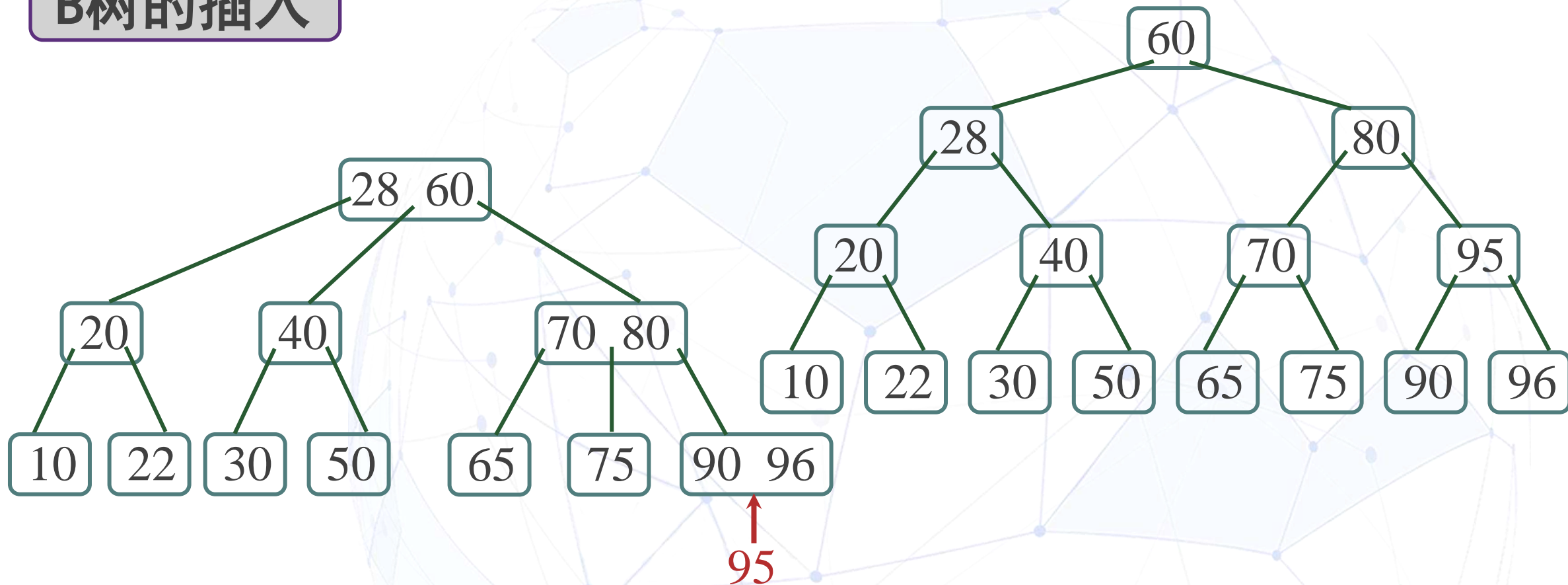
$$m = 3, n = 2$$

分裂——提升

再次发生溢出

再次分裂——提升

B树的插入



🕒 为什么 B 树的根结点最少有两棵子树？

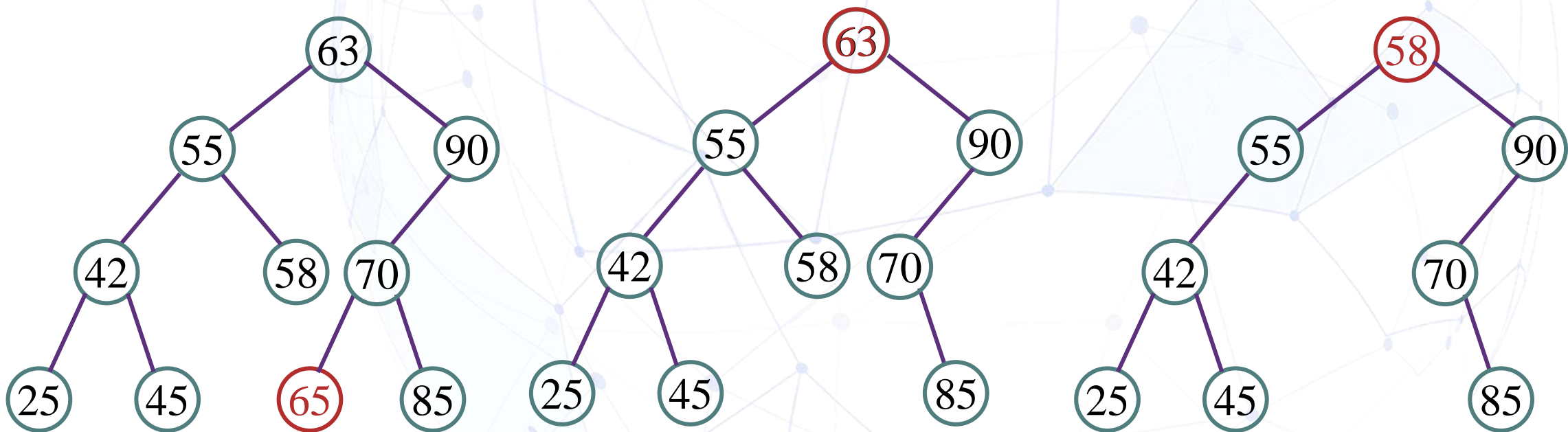
如果在 B 树中插入一个元素时导致根结点发生溢出，则 B 树产生一个新的根结点并且树高增加了一层，此时，新根只有一个关键码和两棵子树。

B树的删除

🕒 如何删除二叉排序树中的一个结点？

📎 情况 1——被删除的结点是叶子结点

📎 情况 3——被删除的结点既有左子树也有右子树



B树的删除

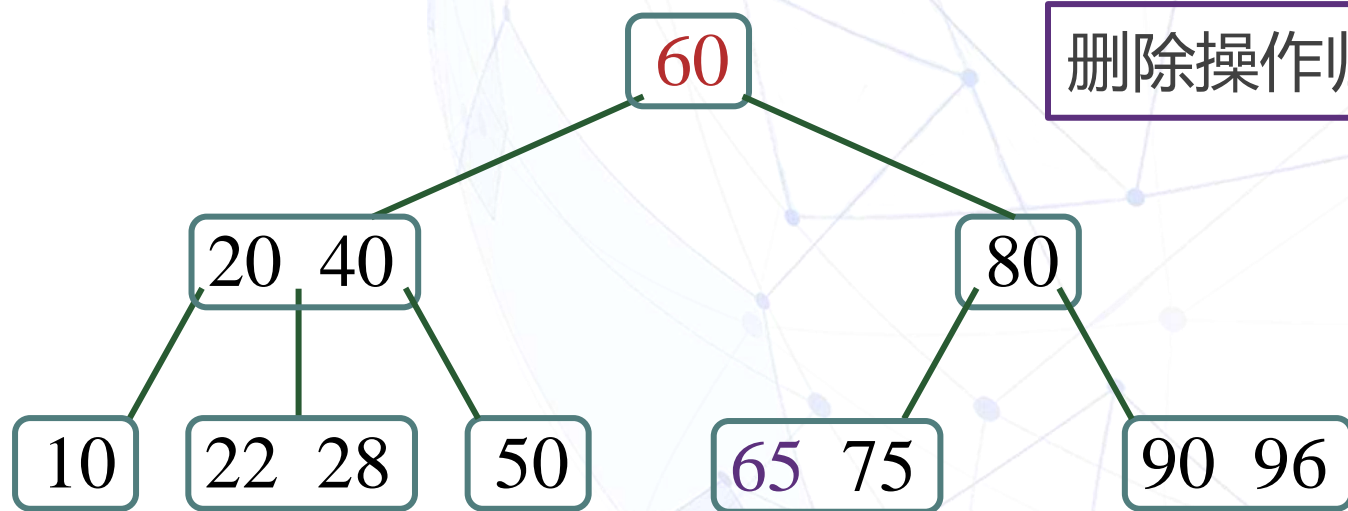
✈ 假定在 m 阶 B 树中删除关键码 key ，删除过程如下：

(1) **定位**：确定关键码 key 在哪个结点并返回该结点的指针 q 。

假定 key 是结点 q 中的第 i 个关键码 K_i ，有以下两种情况：

1.1 若结点 q 是叶子结点，则删除 key ；

1.2 若结点 q 不是叶子结点，则用 A_i 所指子树中的最小值 x 替换 K_i ；删除 x ；



删除操作归结为在叶子结点中删除关键码

B树的删除

✈ 假定在 m 阶 B 树中删除关键码 key ，删除过程如下：

(2) 判断是否下溢

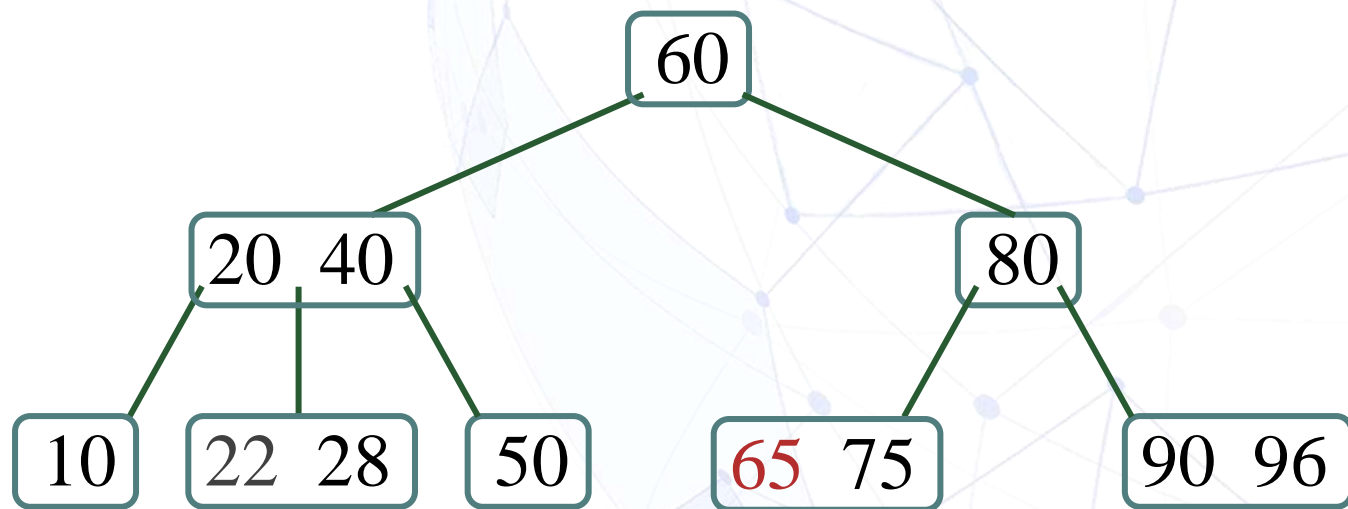
2.1 如果叶子结点中关键码的个数大于 $\left\lceil \frac{m}{2} \right\rceil - 1$ ，则直接删除；



B 树是多少阶？



$$\left\lceil \frac{m}{2} \right\rceil - 1 = 1$$



B树的删除

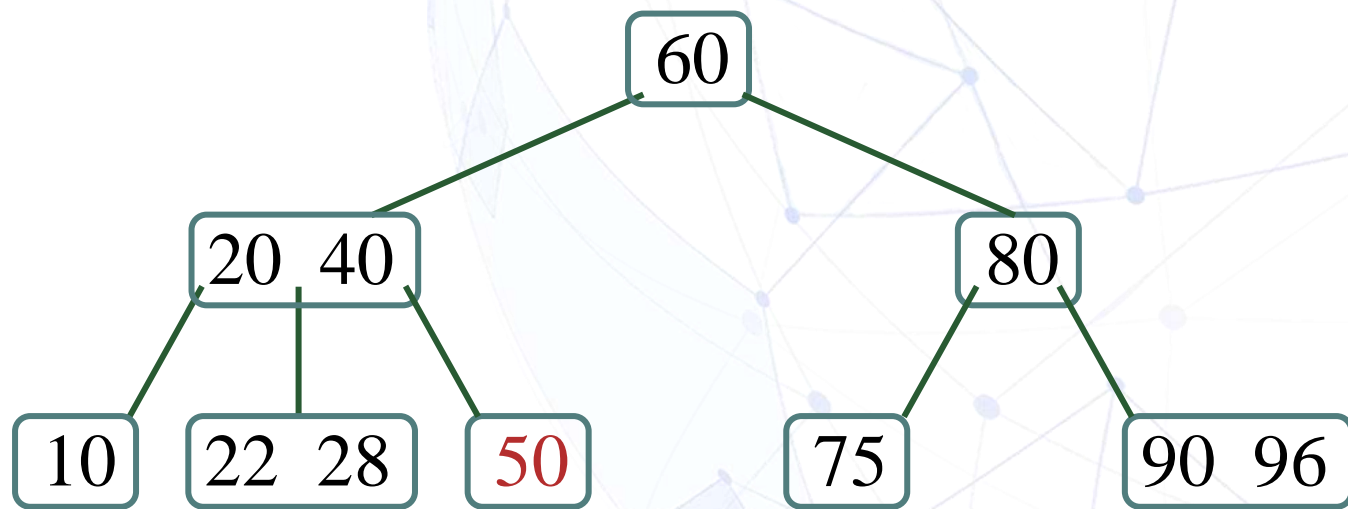
✈ 假定在 m 阶 B 树中删除关键码 key ，删除过程如下：

(2) 判断是否下溢

2.1 如果叶子结点中关键码的个数大于 $\left\lceil \frac{m}{2} \right\rceil - 1$ ，则直接删除；

2.2 否则，删除操作涉及到兄弟结点

2.2.1 兄弟结点的关键码个数大于 $\left\lceil \frac{m}{2} \right\rceil$ ，则向兄弟结点借一个关键码；



兄弟够借

B树的删除

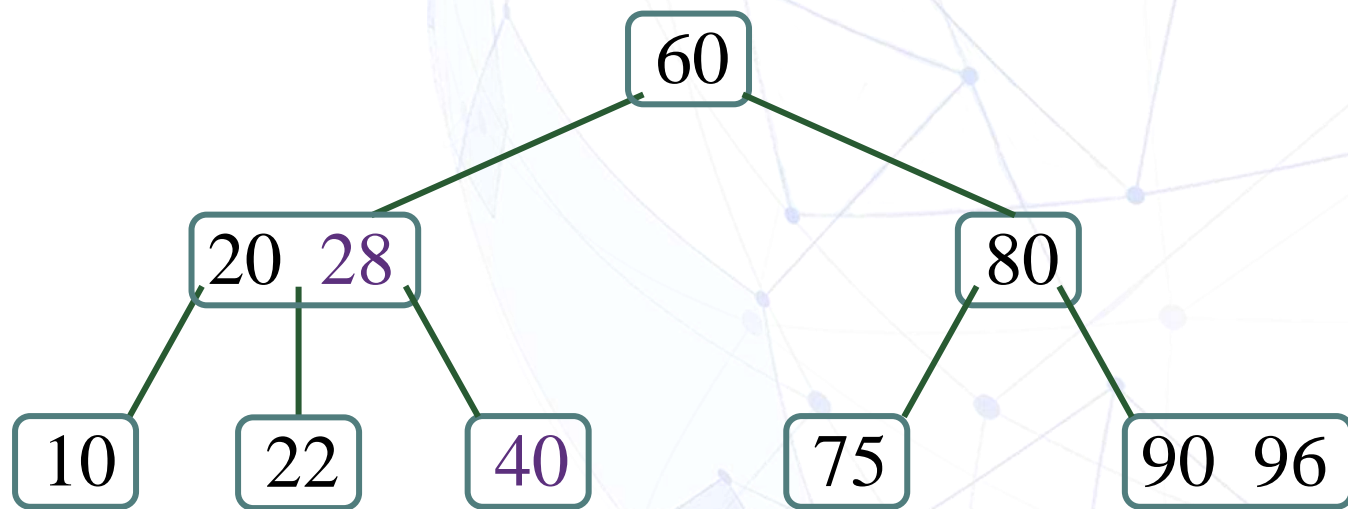
✈ 假定在 m 阶 B 树中删除关键码 key ，删除过程如下：

(2) 判断是否下溢

2.1 如果叶子结点中关键码的个数大于 $\left\lceil \frac{m}{2} \right\rceil - 1$ ，则直接删除；

2.2 否则，删除操作涉及到兄弟结点

2.2.1 兄弟结点的关键码个数大于 $\left\lceil \frac{m}{2} \right\rceil$ ，则向兄弟结点借一个关键码，并且借来的关键码“上移”到双亲结点，双亲结点相应关键码“下移”；



B树的删除

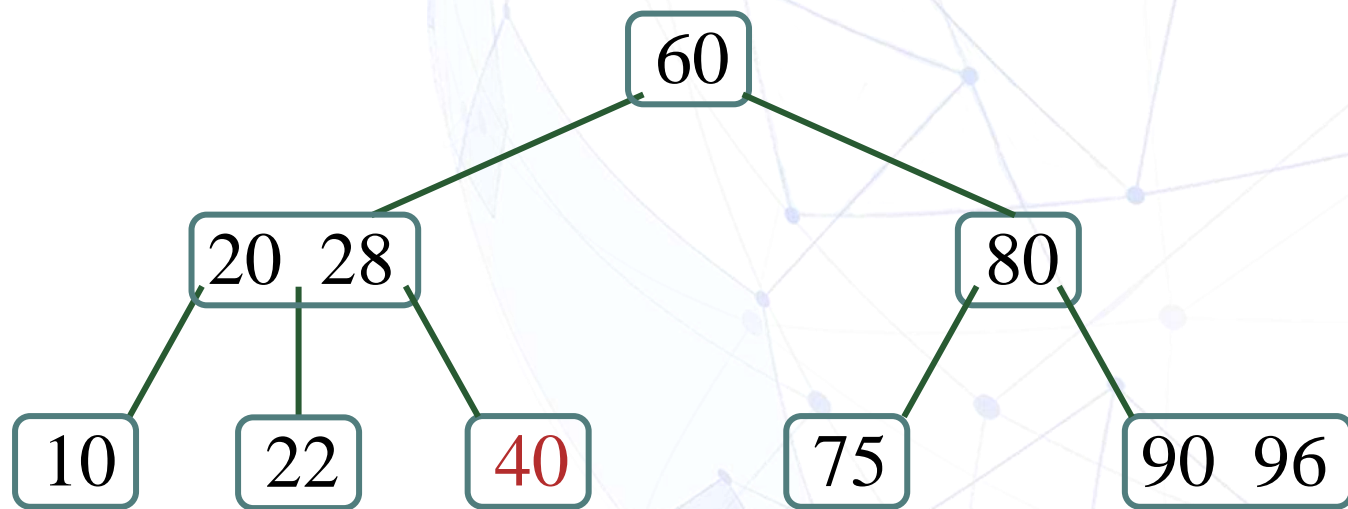
✈ 假定在 m 阶 B 树中删除关键码 key ，删除过程如下：

(2) 判断是否下溢

2.1 如果叶子结点中关键码的个数大于 $\left\lceil \frac{m}{2} \right\rceil - 1$ ，则直接删除；

2.2 否则，删除操作涉及到兄弟结点

2.2.2 兄弟结点的关键码个数 **不大于** $\left\lceil \frac{m}{2} \right\rceil$ ，则执行“合并”兄弟操作；



兄弟不够借

B树的删除

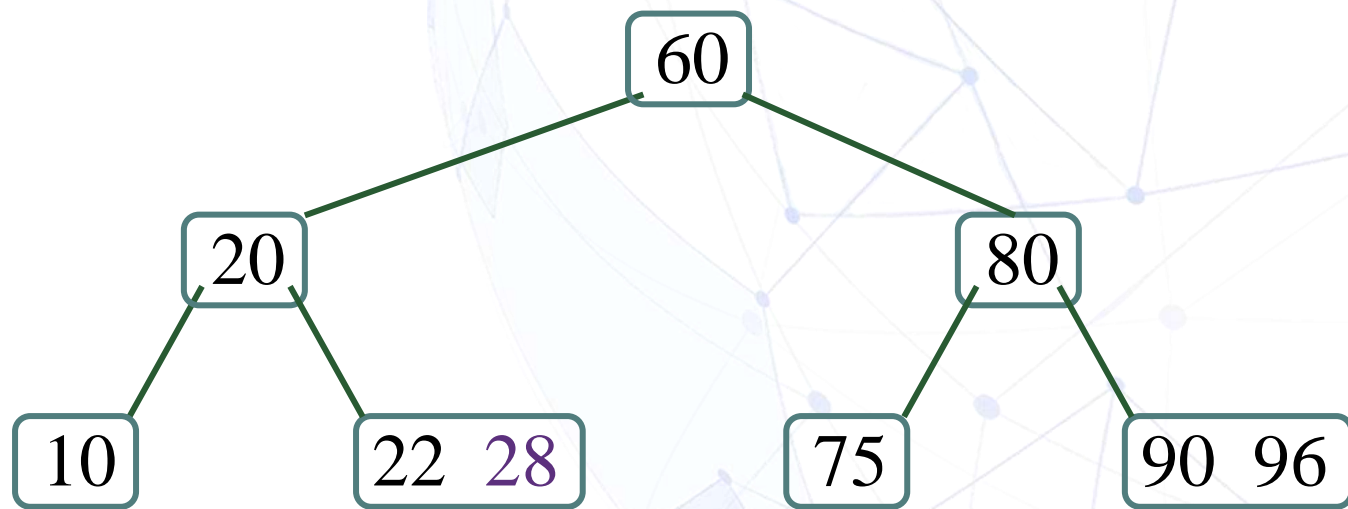
✈ 假定在 m 阶 B 树中删除关键码 key ，删除过程如下：

(2) 判断是否下溢

2.1 如果叶子结点中关键码的个数大于 $\left\lceil \frac{m}{2} \right\rceil - 1$ ，则直接删除；

2.2 否则，删除操作涉及到兄弟结点

2.2.2 兄弟结点的关键码个数 **不大于** $\left\lceil \frac{m}{2} \right\rceil$ ，则执行“合并”兄弟操作；
删除空结点，并将双亲结点中的相应关键码“下移”到合并结点中；



合并——下移

B树的删除

✈ 假定在 m 阶 B 树中删除关键码 key ，删除过程如下：

(2) 判断是否下溢

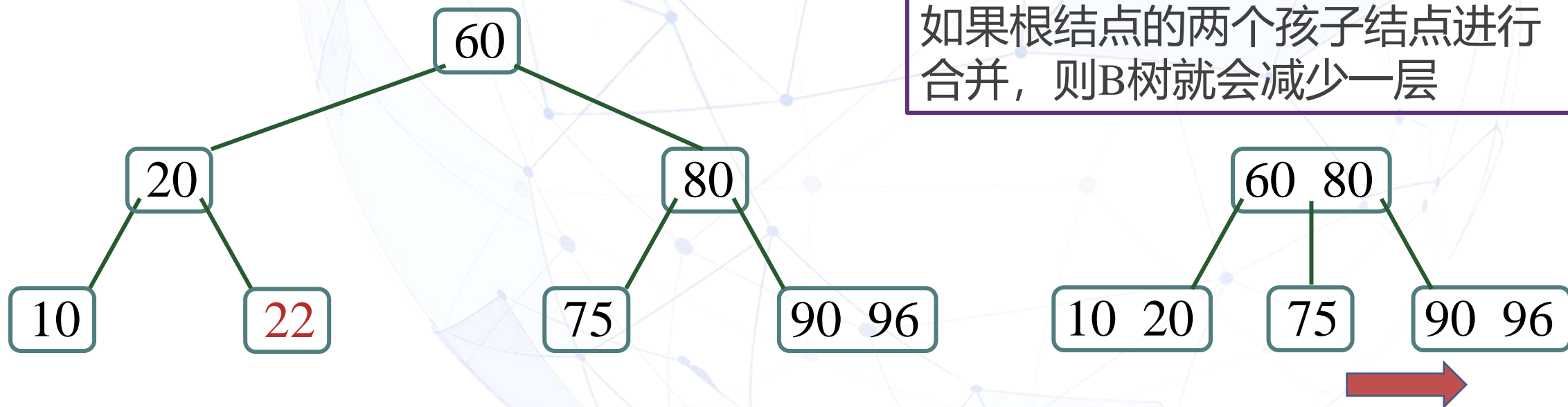
2.1 如果叶子结点中关键码的个数大于 $\left\lceil \frac{m}{2} \right\rceil - 1$ ，则直接删除；

2.2 否则，删除操作涉及到兄弟结点

2.2.2 兄弟结点的关键码个数 **不大于** $\left\lceil \frac{m}{2} \right\rceil$ ，则执行“合并”兄弟操作；

如果双亲结点的关键码个数发生下溢，则双亲结点也要进行借关键码或合并操作

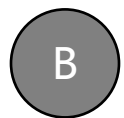
如果根结点的两个孩子结点进行合并，则B树就会减少一层



1. 在B树中，对于任何一个分支结点中的某个关键码 k ，比 k 大的最小关键码和比 k 小的最大关键码一定都在叶结点中。



正确



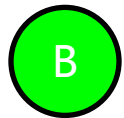
错误

提交

2. m 阶B树中每个结点的子树个数都大于或等于 $m/2$ 。



正确



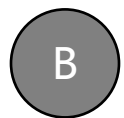
错误

提交

3. m 阶B树中任何一个结点的左右子树的高度都相等。



正确



错误

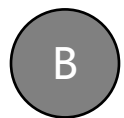
提交

4. B树是结点之间满足一定大小关系的平衡m叉树。



A

正确



B

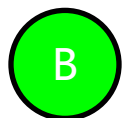
错误

提交

5. 在B树中执行插入操作，新插入的关键码一定是在叶子结点中。



正确



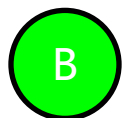
错误

提交

6. 在B树中执行删除操作，一定会涉及到被删除关键码结点的双亲结点。



正确



错误

提交

7. 在7阶B树中插入一个关键码，在执行插入操作前，当所插入结点的键码个数为（ ）时发生分裂操作。

A 7

B 6

C 4

D 3

提交

8. 在7阶B树中删除一个关键码，在执行删除操作前，当所删除结点的关键码个数为（ ）时发生借关键码的操作。

A 7

B 6

C 4

D 3

提交

对图7-4所示3阶B树，画出插入元素24的结果。

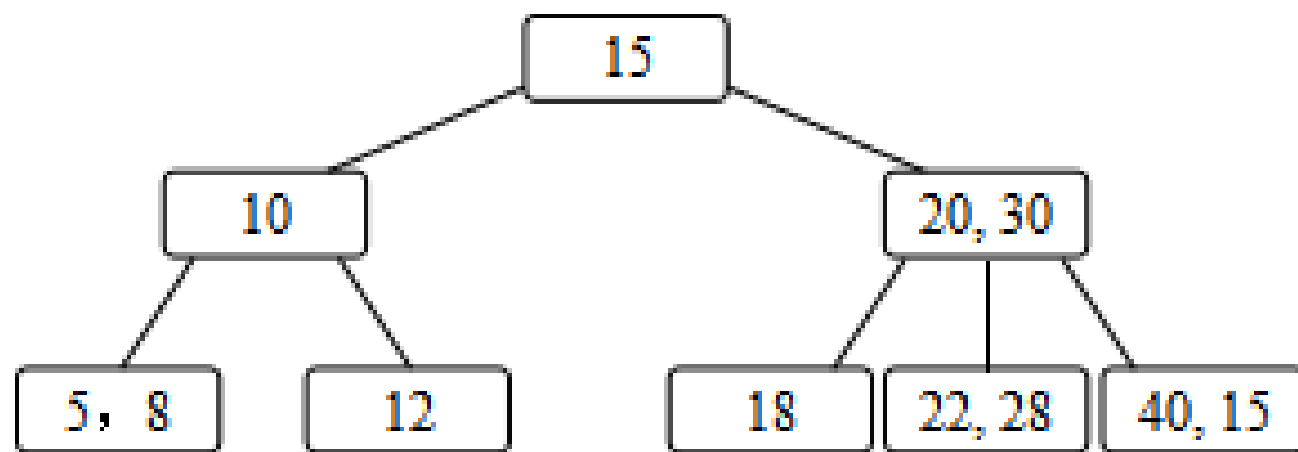


图 7-4 一棵 B 树