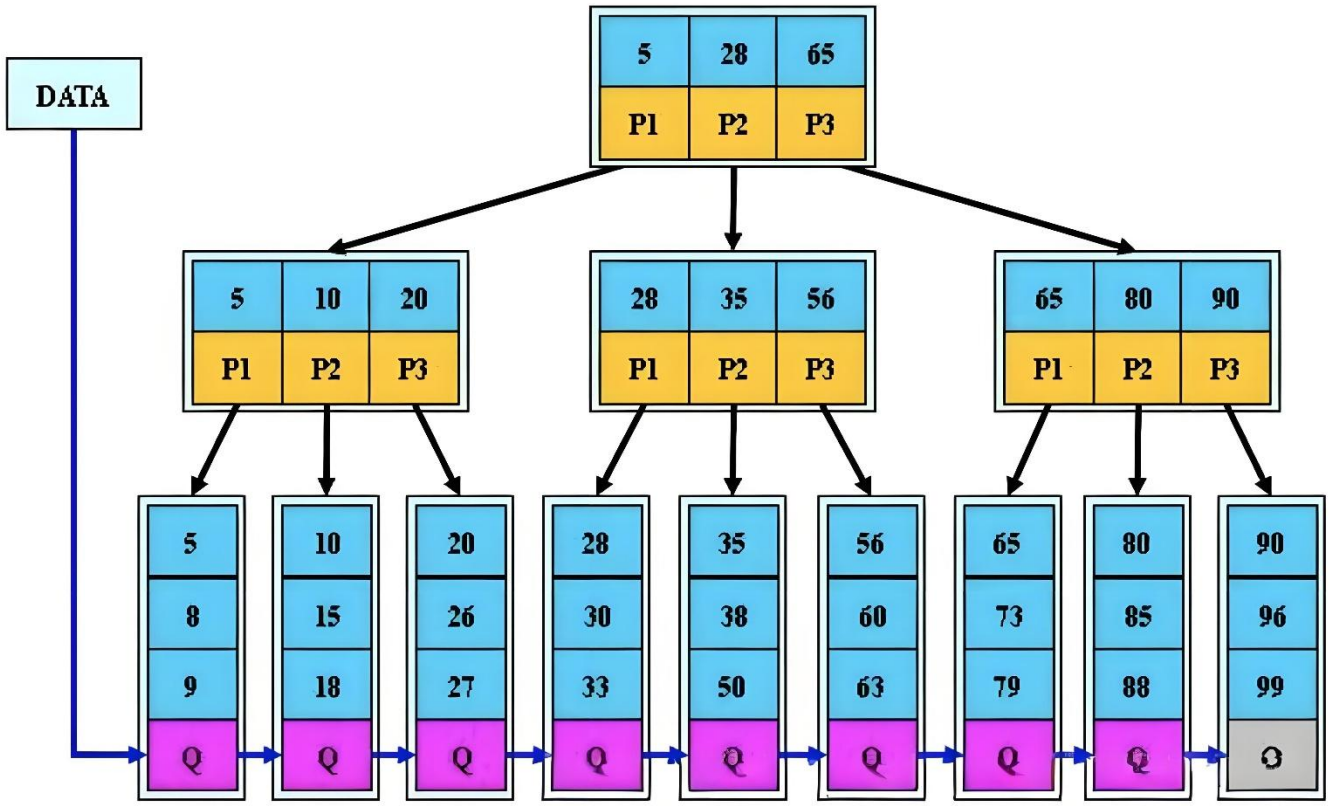
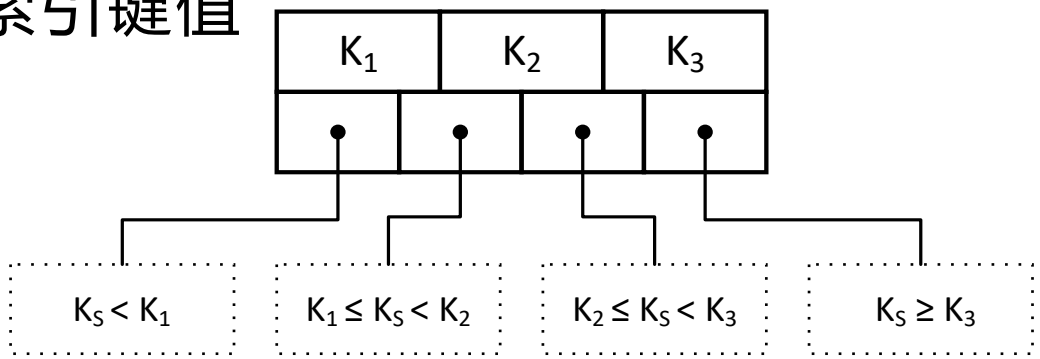
B+树


DATA
5
28
65
P1
P2
P3
5
10
20
P1
P2
P3
28
35
56
P1
P2
P3
65
80
90
P1
P2
P3
5
8
9
Q
10
15
18
Q
20
26
27
Q
28
30
33
Q
35
38
50
Q
56
60
63
Q
65
73
79
Q
80
85
88
Q
90
96
99
Q

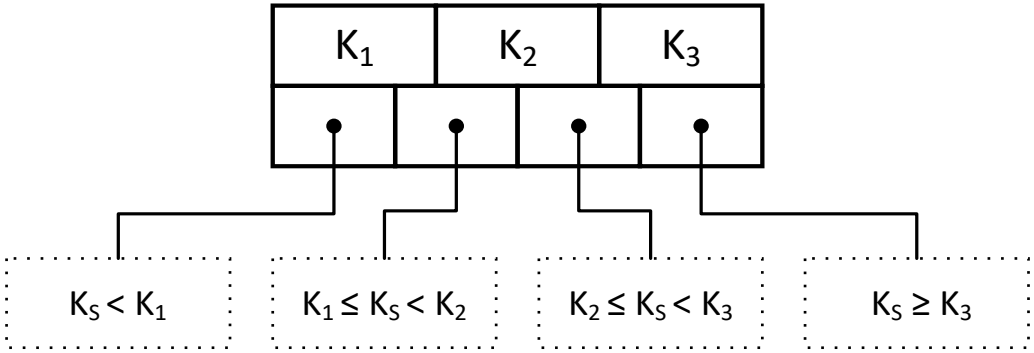
B+树

- B+树可以定义一个m值作为预定范围，即m路(阶)B+树。
- 一个节点的结构是：（K1，P1，K2，P2，...Km-1,Pm)，其中P为指向子节点（子树）的指针，K为索引键值。
- 对于根节点，如果它本身不是叶子节点，则至少拥有1个关键字，即至少有2个子树
- 除了根节点，非叶子节点至少拥有 $\lceil m/2 \rceil$ 个指针；叶子节点最少 $\lceil m/2 \rceil$ 个索引键值
- 除了根节点，所有节点最多包含 m 个子树（指针），即 $m-1$ 个索引键值



B+树

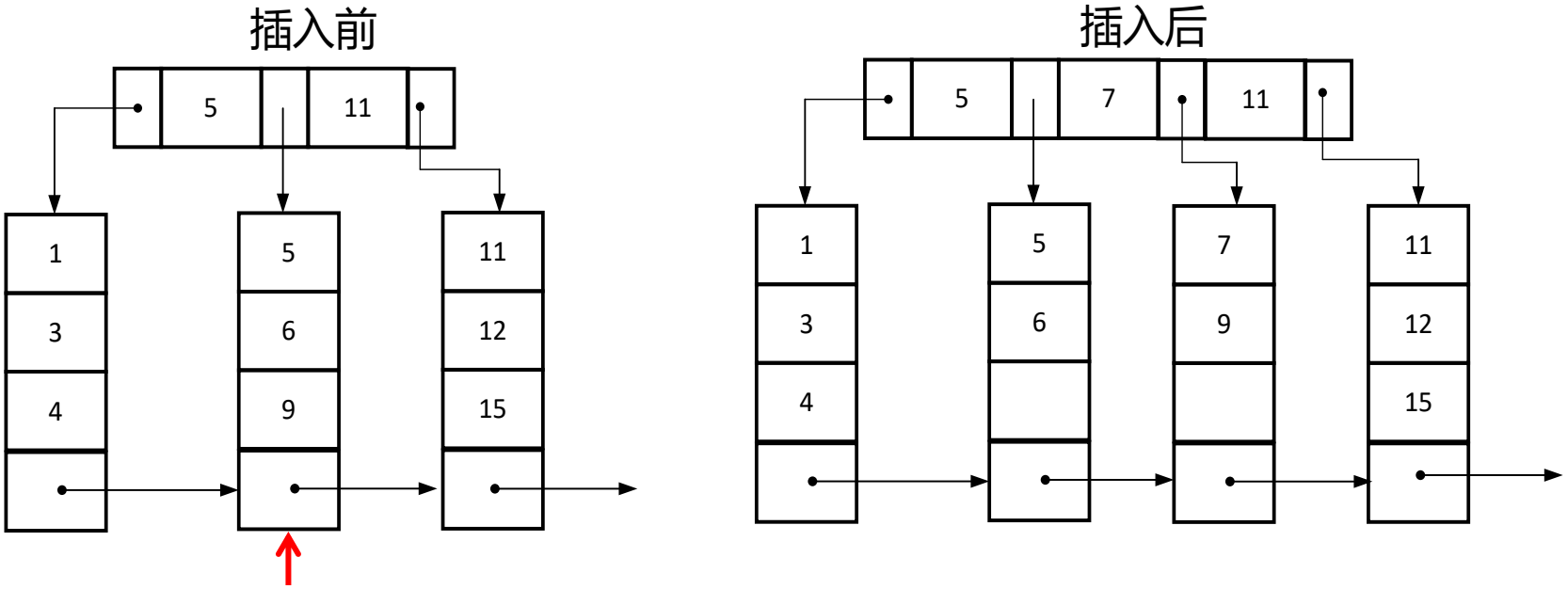
- 非叶子节点不保存数据，只保存关键字用作索引，所有数据都保存在叶子节点中。
- 非叶子节点有若干子树指针，如果非叶子节点索引键值为k1,k2,…km-1，那么第一个子树索引键值判断条件为小于k1，第二个为大于等于k1而小于k2，以此类推，最后一个为大于等于km-1，总共可以划分出m个区间，即可以有m个分支。



B+树

1.插入操作

阶数为4

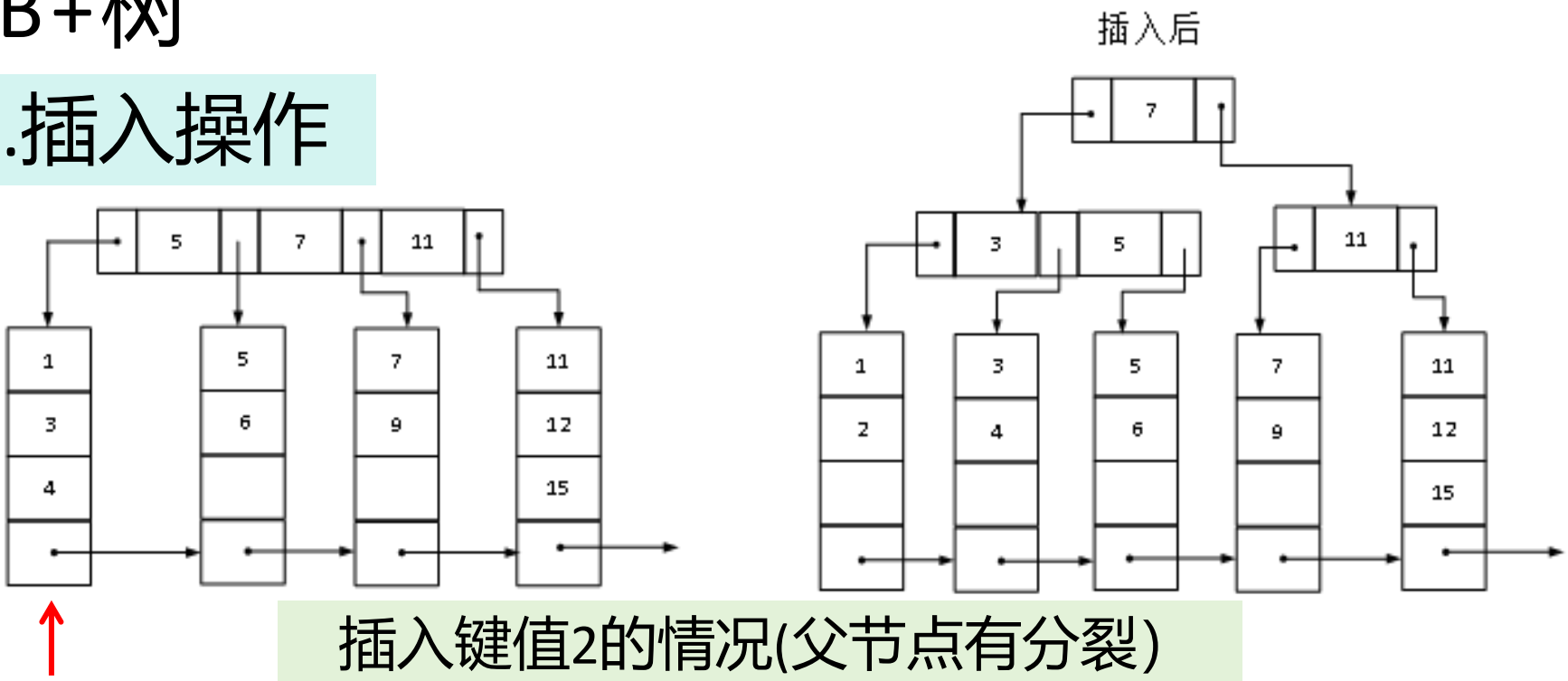


插入键值7的情况（父节点无分裂）

- 图中给出了一个节点的分裂情况：向树中插入键值7。首先通过查找算法定位插入的位置，在叶节点的6和9两个键值之间插入7。设当前叶节点的阶数为4，每个叶节点最多可以存放3个键值，因此需要进行节点的分裂操作，右图所示。以位置为m/2位置的键值，即6为分裂点，5和6保留在当前节点，7和9传输到新创建的节点中，同时将键值7提升至父节点中，增加指向父节点的指针和同级节点的指针。

B+树

1.插入操作



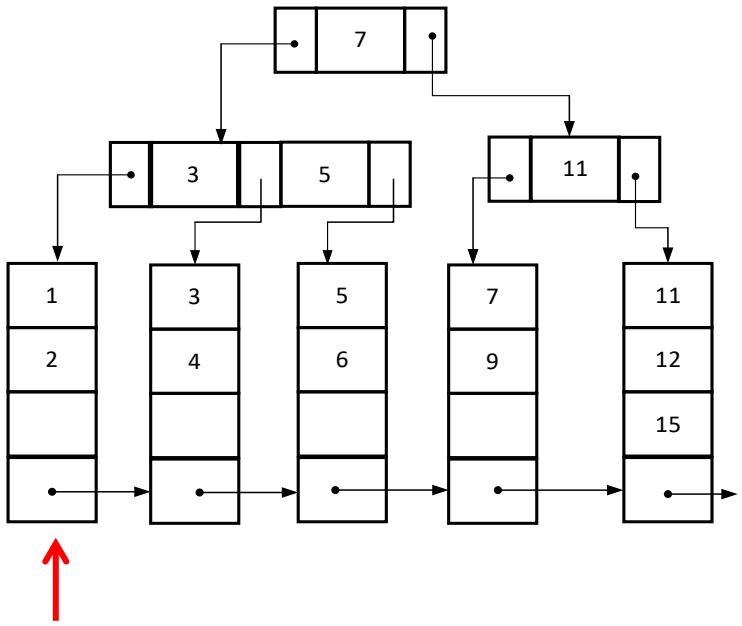
插入键值2的情况(父节点有分裂)

- 图中给出了一个非叶节点的分裂情况：若向图中的B+树中插入键值2时，在包含（1，3，4）的叶子节点中插入该值以后，该节点超过了最大键值个数3，因此需要进行分裂。选择键值2处作为分裂点，将键值3提升至父节点，并将（1，3，4）节点分裂成两部分，1和2保留在原始节点中，3和4放在新创建的节点中。之后，由于非叶节点（5，7，11）在插入键值3后，超过了最大键值个数，需要再次进行分裂，提升键值7，创建新的根节点，树的高度由2变为3，此处因为是非叶节点分裂，所以新创建的节点（11）中不用重复保留键值7。最后，增加新节点到父节点的指针

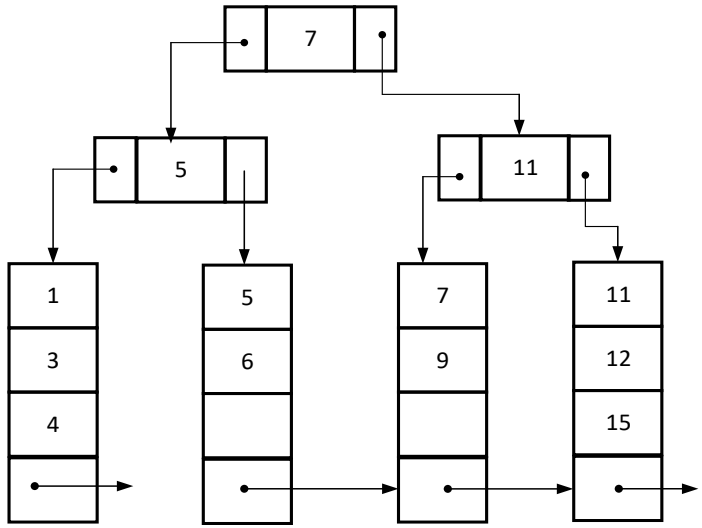
B+树

2.删除操作

删除前



删除后

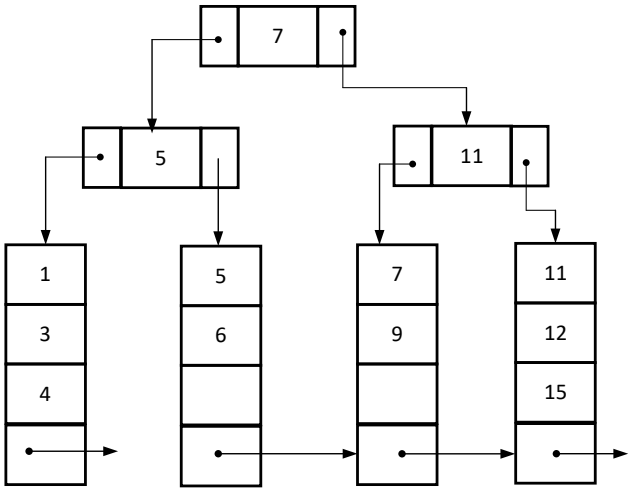


删除键值2的情况（合并）

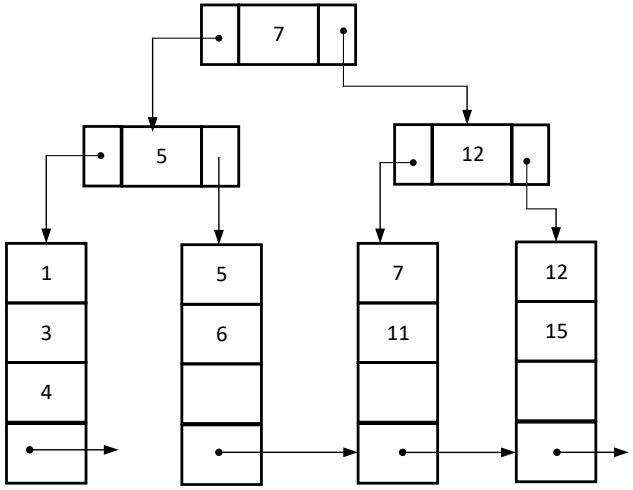
B+树

2.删除操作

删除前



删除后



删除键值9的情况（重新分配）

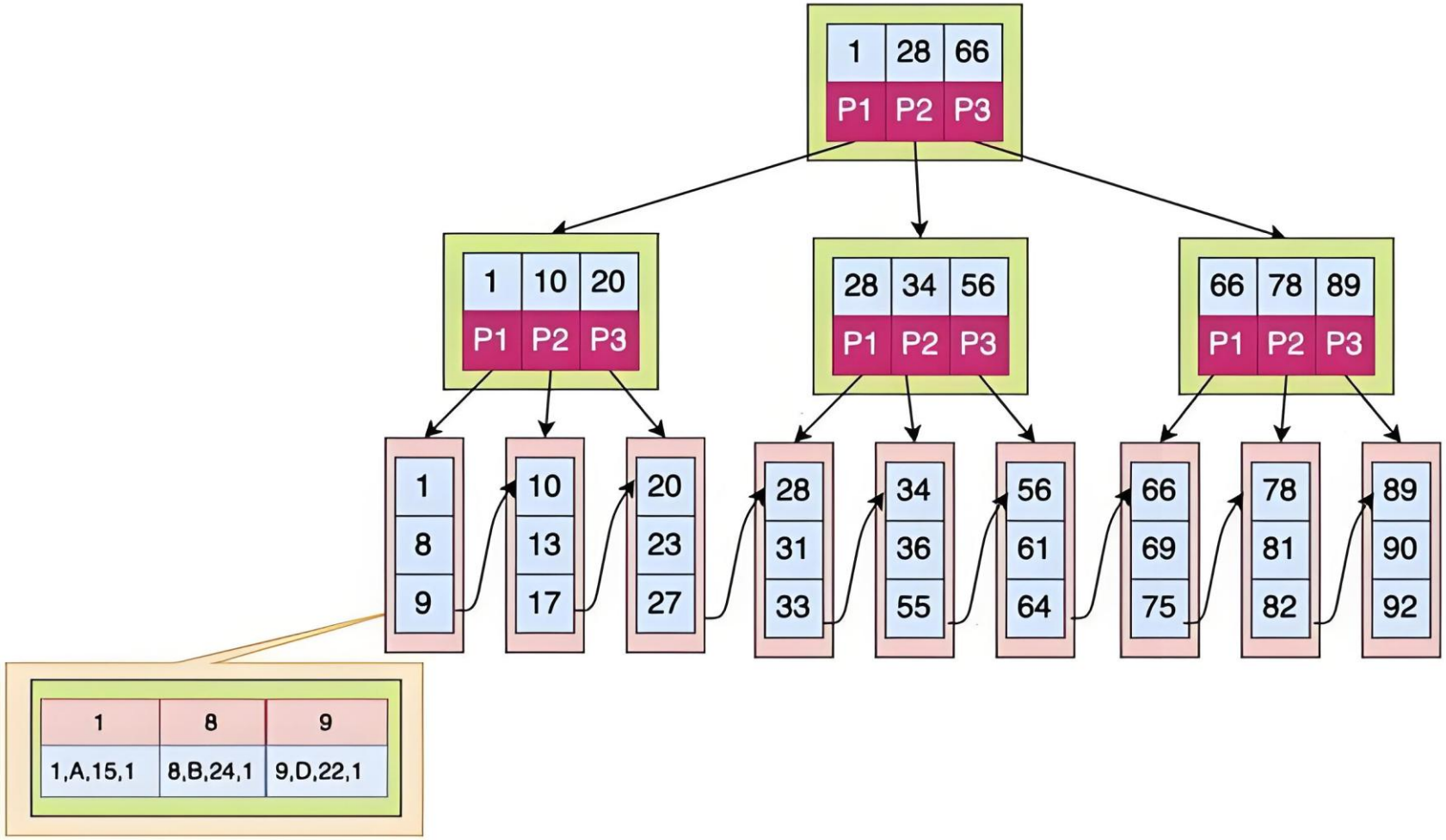
聚集索引 VS 非聚集索引

- 在 MySQL 中，B+ 树索引按照存储方式的不同分为聚集索引和非聚集索引
- 聚集索引（聚簇索引）：以 InnoDB 作为存储引擎的表，表中的数据都会有一个主键，即使你不创建主键，系统也会帮你创建一个隐式的主键。这是因为 InnoDB 是把数据存放在 B+ 树中的，而 B+ 树的键值就是主键，在 B+ 树的叶子节点中，存储了表中所有的数据
- 这种以主键作为 B+ 树索引的键值而构建的 B+ 树索引，我们称之为聚集索引

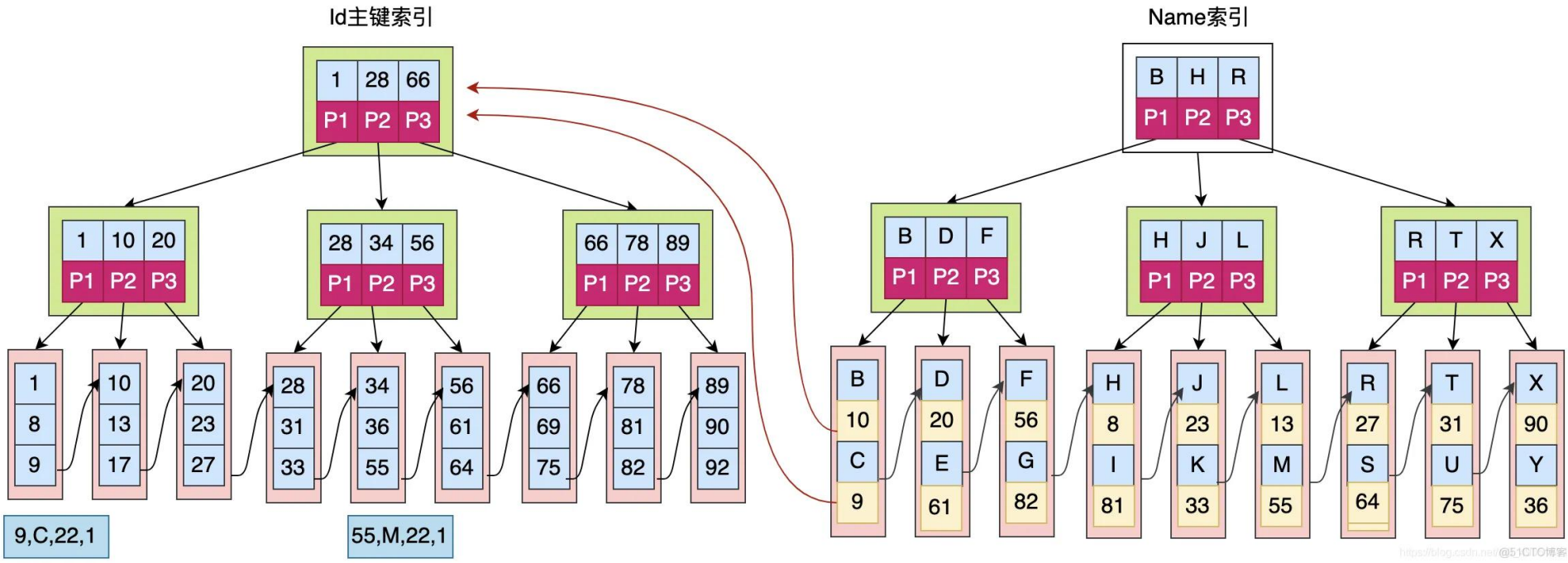
聚集索引 vs 非聚集索引

- 非聚集索引（非聚簇索引）：以主键以外的列值作为键值构建的 B+ 树索引，我们称之为非聚集索引。
- 以主键以外的列值作为键值构建的 B+ 树索引，我们称之为非聚集索引。
- 非聚集索引与聚集索引的区别在于非聚集索引的叶子节点不存储表中的数据，而是存储该列对应的主键，想要查找数据我们还需要根据主键再去聚集索引中进行查找，这个再根据聚集索引查找数据的过程，我们称为回表。

聚集索引的例子（Innodb引擎）



非聚集索引（辅助索引）


Id主键索引
Name索引
1 28 66
P1 P2 P3
B H R
P1 P2 P3
1 10 20
28 34 56
66 78 89
B D F
H J L
R T X
1 8 9
10 13 17
20 23 27
28 31 33
34 36 55
56 61 64
66 69 75
78 81 82
89 90 92
B 10 C 9
D 20 E 61
F 56 G 82
H 8 I 81
J 23 K 33
L 13 M 55
R 27 S 64
T 31 U 75
X 90 Y 36
9,C,22,1
55,M,22,1

B+树的优点

- 高效查询：
  - 范围查询：B+树的所有数据都存储在叶子节点，且叶子节点通过指针相连，适合范围查询。
  - 稳定时间复杂度：查询、插入、删除操作的时间复杂度为O(log n)，适合大规模数据。
- 磁盘I/O优化：
  - 减少磁盘访问：节点通常设计为一个磁盘块大小，减少I/O操作。
  - 高扇出：节点能存储更多键，降低树的高度，进一步减少磁盘访问。
- 顺序访问高效：
  - 链表结构：叶子节点通过指针相连，顺序访问非常高效。
- 平衡性：
  - 自动平衡：插入和删除后，B+树会自动调整，保持平衡，确保操作效率。

B+树的缺点

- 空间开销：
  - 额外指针：内部节点和叶子节点存储大量指针，增加空间占用。
- 复杂性：
  - 实现复杂：相比二叉搜索树，B+树的实现和调试更复杂。
- 写操作开销（写放大）：
  - 分裂与合并：插入和删除可能导致节点分裂或合并，增加额外开销。

B+树的适用场景

- B+树因其高效查询和磁盘I/O优化，广泛应用于数据库、文件系统、键值存储、搜索引擎、分布式系统、大数据处理和实时系统等场景，特别适合需要频繁查询和范围查询的应用。
- B+树的不适用场景？
- 高写入负载、内存受限、完全随机访问、高并发写入、前缀查询或模糊查询

问题（举例）

- 假设有一个B+树，其阶数（即每个节点的最大子节点数）为3。初始时，B+树为空。请依次插入以下键值，并画出每次插入后的B+树结构：

假设有一个MySQL数据库表 students插入序列：

10, 20, 5, 15, 25,

假设有一个MySQL数据库表 students，存放数据如右表，假设MySQL使用B+树作为聚簇索引，且B+树的阶数为 3，请画出基于 id 列的聚簇索引B+树结构。

表中存储了以下数据：

id	name	age	grade
10	Alice	20	A
5	Bob	22	B
20	Carol	21	A
15	Dave	19	C
25	Eve	23	B

如果插入一条新记录 (12, 'Kevin', 18, 'B')，描述B+树的结构变化，并画出插入后的B+树。