BST 树

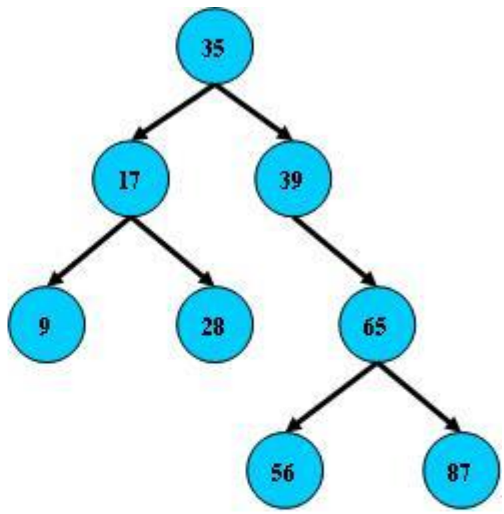
即二叉搜索树：

1.所有非叶子结点至多拥有两个儿子（Left 和 Right）；

2.所有结点存储一个关键字；

3.非叶子结点的左指针指向小于其关键字的子树，右指针指向大于其关键字的子树；

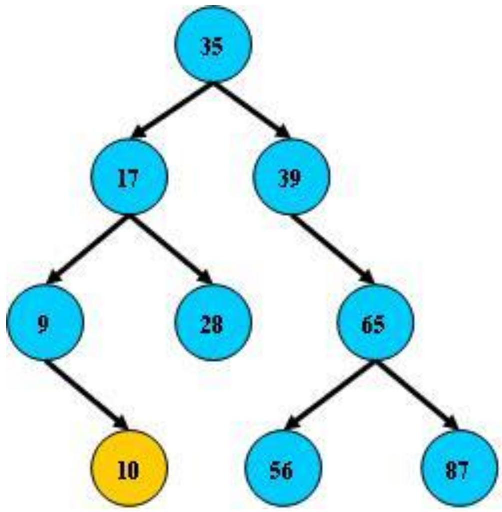
如：

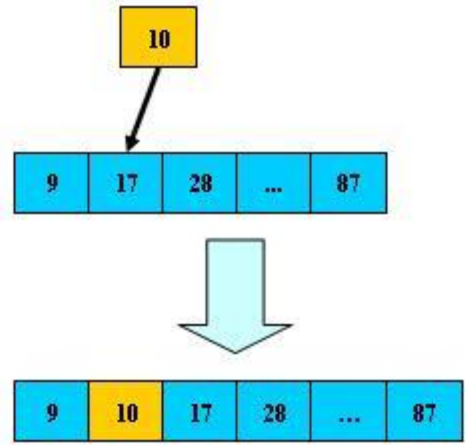


BST 树的搜索，从根结点开始，如果查询的关键字与结点的关键字相等，那么就命中；
否则，如果查询关键字比结点关键字小，就进入左儿子；如果比结点关键字大，就进入右儿子；如果左儿子或右儿子的指针为空，则报告找不到相应的关键字；

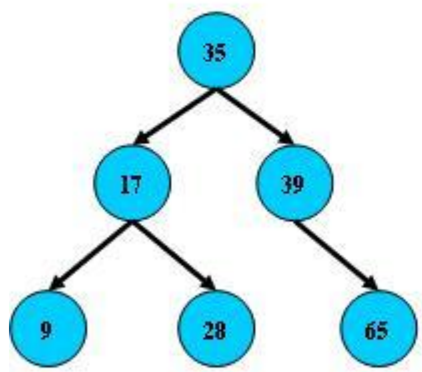
如果 BST 树的所有非叶子结点的左右子树的结点数目均保持差不多（平衡），那么 B 树
的搜索性能逼近二分查找；但它比连续内存空间的二分查找的优点是，改变 BST 树结构（插入与删除结点）不需要移动大段的内存数据，甚至通常是常数开销；

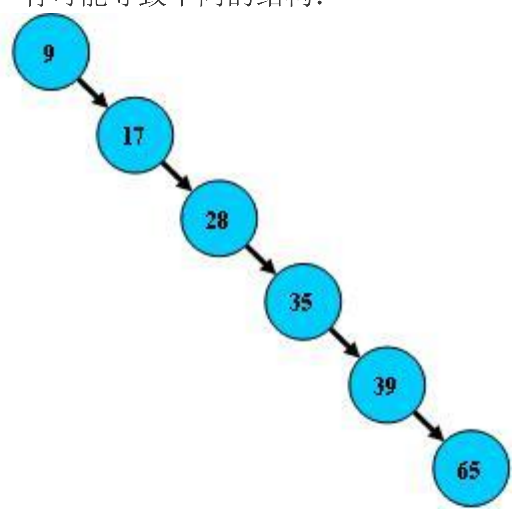
如：





但 BST 树在经过多次插入与删除后，有可能导致不同的结构：





右边也是一个 BST 树，但它的搜索性能已经是线性的了；同样的关键字集合有可能导致不同的
树结构索引；所以，使用 BST 树还要考虑尽可能让 BST 树保持左图的结构，和避免右图的结构，也就
是所谓的“平衡”问题；

AVL 平衡二叉搜索树

定义：平衡二叉树或为空树,或为如下性质的二叉排序树:

（1）左右子树深度之差的绝对值不超过 1;

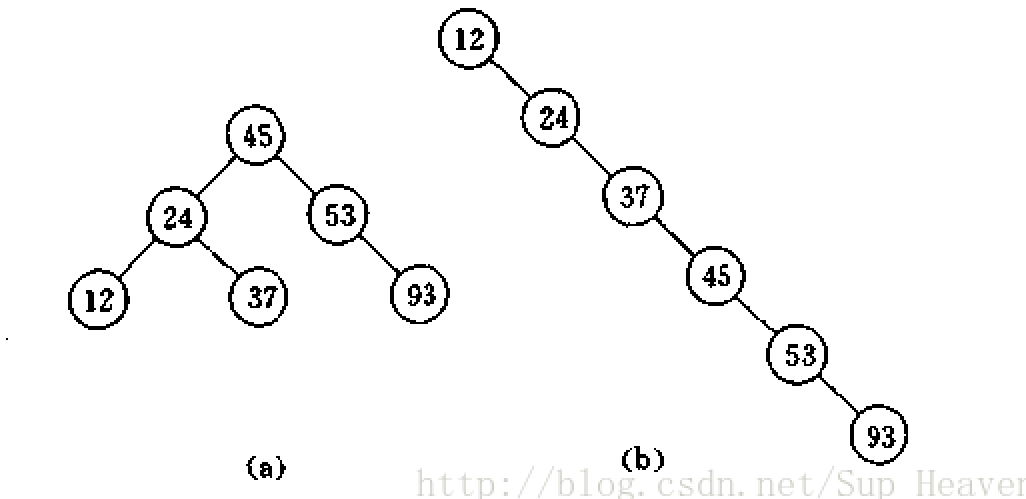
（2）左右子树仍然为平衡二叉树.

平衡因子 BF=左子树深度 - 右子树深度.

平衡二叉树每个结点的平衡因子只能是 1，0，-1。若其绝对值超过 1，则该二叉排序树就是不

平衡的。

如图所示为平衡树和非平衡树示意图：



RBT 红黑树

AVL 是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多；

红黑是弱平衡的，用非严格的平衡来换取增删节点时候旋转次数的降低；

所以简单说，搜索的次数远远大于插入和删除，那么选择 AVL 树，如果搜索，插入删除次数几乎差不多，应该选择 RB 树。

红黑树上每个结点内含五个域，color，key，left，right，p。如果相应的指针域没有，则设为 NIL。

一般的，红黑树，满足以下性质，即只有满足以下全部性质的树，我们才称之为红黑树：

1）每个结点要么是红的，要么是黑的。

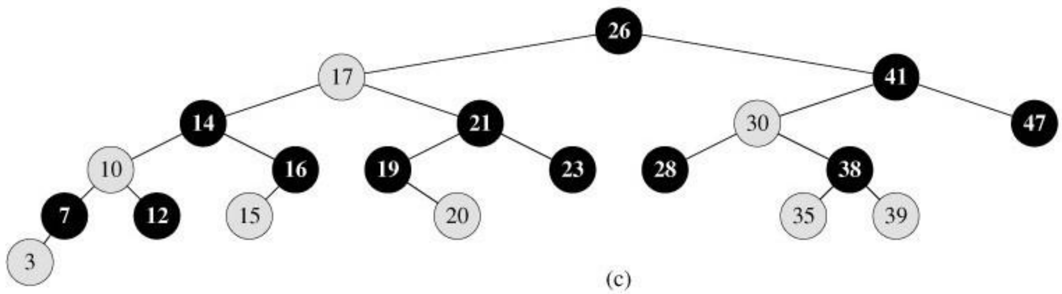
2）根结点是黑的。

3）每个叶结点，即空结点（NIL）是黑的。

4）如果一个结点是红的，那么它的俩个儿子都是黑的。

5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

下图所示，即是一颗红黑树：



(c)

B-树

是一种平衡多路搜索树（并不是二叉的）：

1.定义任意非叶子结点最多只有 M 个儿子；且 M>2；

2.根结点的儿子数为[2, M]；

3.除根结点以外的非叶子结点的儿子数为[M/2, M]；

4.每个结点存放至少 M/2-1（取上整）和至多 M-1 个关键字；（至少 2 个关键字）

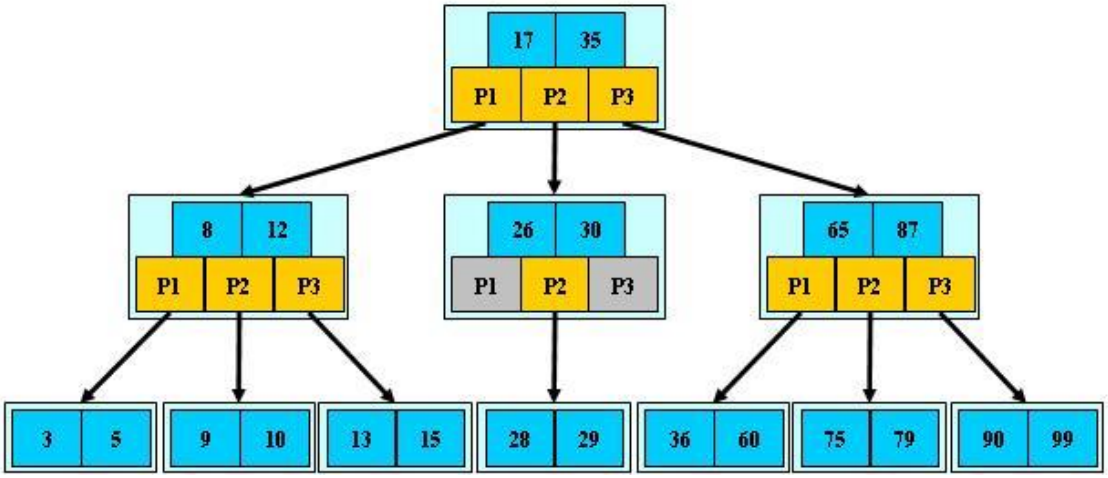
5.非叶子结点的关键字个数=指向儿子的指针个数-1；

6.非叶子结点的关键字：K[1], K[2], …, K[M-1]；且 K[i] < K[i+1]；

7.非叶子结点的指针：P[1], P[2], …, P[M]；其中 P[1]指向关键字小于 K[1]的子树，P[M]指向关键字大于 K[M-1]的子树，其它 P[i]指向关键字属于(K[i-1], K[i])的子树；

8.所有叶子结点位于同一层；

如：（**M=3**）



B-树的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；

B-树的特性：

1.关键字集合分布在整颗树中；

2.任何一个关键字出现且只出现在一个结点中；

3.搜索有可能在非叶子结点结束；

4.其搜索性能等价于在关键字全集内做一次二分查找；

5.自动层次控制；

由于限制了除根结点以外的非叶子结点，至少含有 M/2 个儿子，确保了结点的至少利用率，其最底搜索性能为：

$$
\begin{aligned}
& O_{Min} \\
&= O[\log_2(\lceil \tfrac{M}{2} - 1 \rceil) \times \log_{\frac{M}{2}}(\tfrac{N}{\lceil \frac{M}{2} - 1 \rceil})] \\
&= O[\log_2(\tfrac{M}{2})] \times O[\log_{\frac{M}{2}}(\tfrac{N}{\frac{M}{2}})] \\
&= O[\log_2(\tfrac{M}{2}) \times (\log_{\frac{M}{2}} N - 1)] \\
&= O[\log_2 N - \log_2(\tfrac{M}{2})] \\
&= O[\log_2 N] - O[C] \\
&= O[\log_2 N]
\end{aligned}
$$

其中，M 为设定的非叶子结点最多子树个数，N 为关键字总数；

所以 B-树的性能总是等价于二分查找（与 M 值无关），也就没有 B 树平衡的问题；

由于 M/2 的限制，在插入结点时，如果结点已满，需要将结点分裂为两个各占 M/2 的结点；删除结点时，需将两个不足 M/2 的兄弟结点合并；

B+树

B+树是 B-树的变体，也是一种多路搜索树：

1.其定义基本与 B-树同，除了：

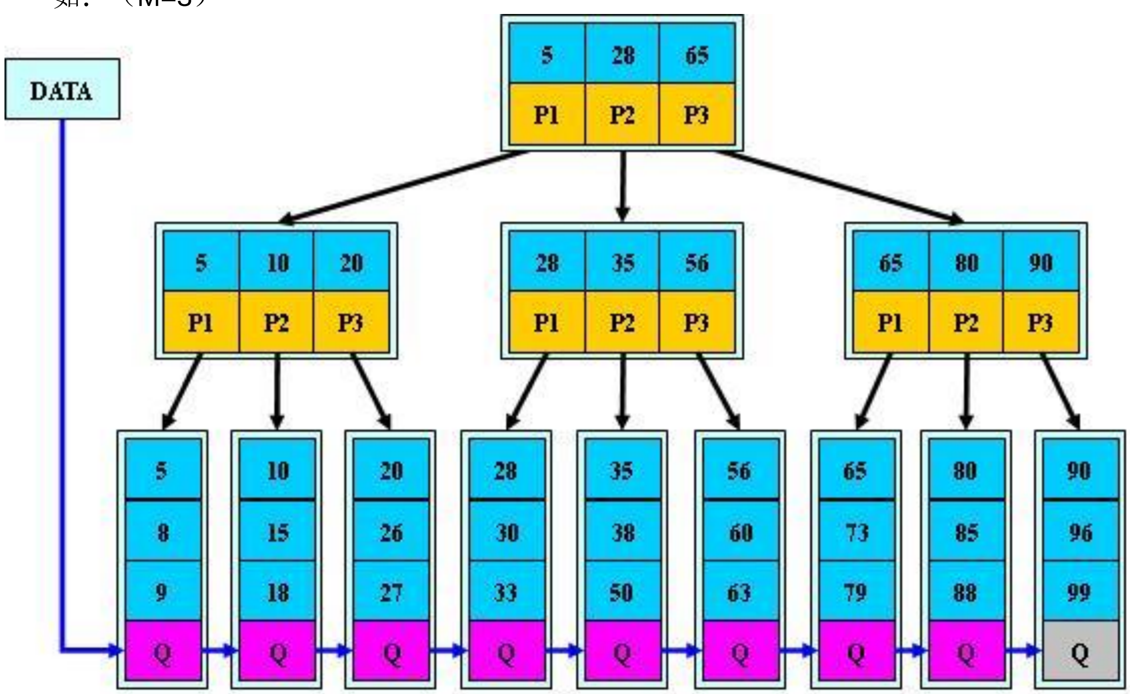
2.非叶子结点的子树指针与关键字个数相同；

3.非叶子结点的子树指针 P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

5.为所有叶子结点增加一个链指针；

6.所有关键字都在叶子结点出现；

如：（M=3）



B+的搜索与 B-树也基本相同，区别是 B+树只有达到叶子结点才命中（B-树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；

B+的特性：

1.所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好

是有序的；

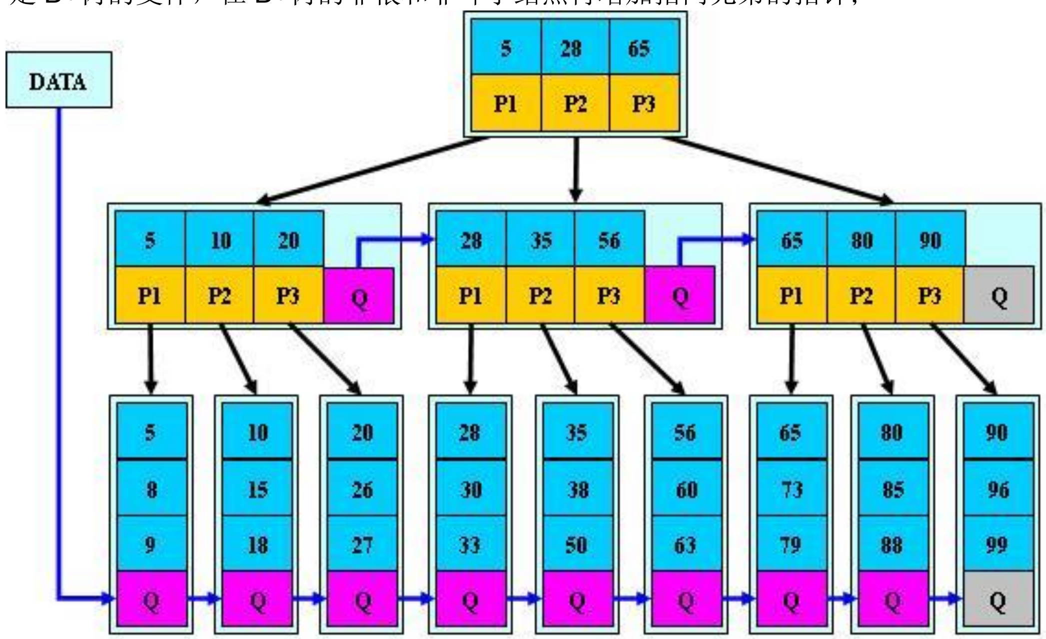
2.不可能在非叶子结点命中；

3.非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；

4.更适合文件索引系统；比如对已经建立索引的数据库记录，查找 10<=id<=20，那么只要通过根节点搜索到 id=10 的叶节点，之后只要根据叶节点的链表找到第一个大于 20 的就行了，比 B-树在查找 10 到 20 内的每一个时每次都从根节点出发查找提高了不少效率。

B*树

是 B+树的变体，在 B+树的非根和非叶子结点再增加指向兄弟的指针；



B*树定义了非叶子结点关键字个数至少为(2/3)*M，即块的最低使用率为 2/3（代替 B+树的 1/2）；

B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中 1/2 的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

B*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制 1/3 的数据到新结点，最后在父结点增加新结点的指针；

所以，B*树分配新结点的概率比 B+树要低，空间使用率更高；

小结

B 树：二叉树，每个结点只存储一个关键字，等于则命中，小于走左结点，大于走右结点；

B-树：多路搜索树，每个结点存储 M/2 到 M 个关键字，非叶子结点存储指向关键字范围的子结点；

所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；

B+树：在 B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；

B*树：在 B+树基础上，为非叶子结点也增加链表指针，将结点的最低利用率从 1/2 提高到 2/3；

B+/B*Tree 应用

数据库索引--索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。

数据库索引--表数据文件本身就是按 B+Tree 组织的一个索引结构，这棵树的叶节点 data 域保存了完整的数据记录。这个索引的 key 是数据表的主键。

倒排索引--也可以由 B 树及其变种实现但不一定非要 B 树及其变种实现，如 lucene 没有使用 B 树结构，因此 lucene 可以用二分搜索算法快速定位关键词。实现时，lucene 将下面三列分别作为词典文件（Term Dictionary）、频率文件(frequencies)、位置文件 (positions)保存。其中词典文件不仅保存有每个关键词，还保留了指向频率文件和位置文件的指针，通过指针可以找到该关键字的频率信息和位置信息。