B树

B树的定义（为什么需要B树）

B树是一类树，也称为了 **平衡的多路查找树。**包括B树、B+树、B*树等，是一棵 **自平衡的搜索树**，它类似普通的平衡二叉树，不同的一点是B树允许每个结点有更多的子结点。B树是 专门为外部存储器设计的，如磁盘，**它对于读取和写入大块数据有良好的性能**，所以一般被用在 **文件系统** 及 **数据库** 中。

传统用来搜索的平衡二叉树有很多，如 AVL 树，红黑树等。这些树在一般情况下查询性能非常好，但当数据非常大的时候它们就无能为力了。

原因是 当数据量非常大时，内存不够用，大部分数据只能存放在磁盘上，只有需要的数据才加载到内存中。一般而言内存访问的时间约为 50 ns，而磁盘在 10 ms 左右。速度相差了近 5 个数量级，磁盘读取时间远远超过了数据在内存中比较的时间。这说明程序大部分时间会阻塞在磁盘 IO 上。

那么我们如何提高程序性能？减少磁盘 IO 次数，像 AVL 树，红黑树这类平衡二叉树从设计上无法“迎合”磁盘。

B树的特点

索引的效率依赖于磁盘 IO 的次数，快速索引需要有效的减少磁盘 IO 次数，如何快速索引呢？**索引的原理** 其实是不断的缩小查找范围，就如我们平时用字典查单词一样，先找首字母缩小范围，再第二个字母等等。

平衡二叉树是每次将范围分割为两个区间。为了更快，B树每次将范围分割为多个区间，区间越多，定位数据越快越精确。

那么如果结点为区间范围，每个结点就较大了。所以新建结点时，直接申请页大小的空间（磁盘是按页分的，一般为 512 Byte（字节）。磁盘 IO 一次读取若干个页，具体大小和操作系统有关，一般为 4k，8k 或 16k），计算机内存分配是按页对齐的，这样就实现了一个结点只需要一次 IO。

具体来说，拿一棵m阶B树为例，**满足如下性质**

1. 书中每个结点最多有m棵子树（即最多含有m-1个关键字）

> **解释：** m 阶B树，它结点的指针域最多有 m 个，每个指针域指向一棵子树；两个指针域之间必须由一个 key 值将两个指针域隔开

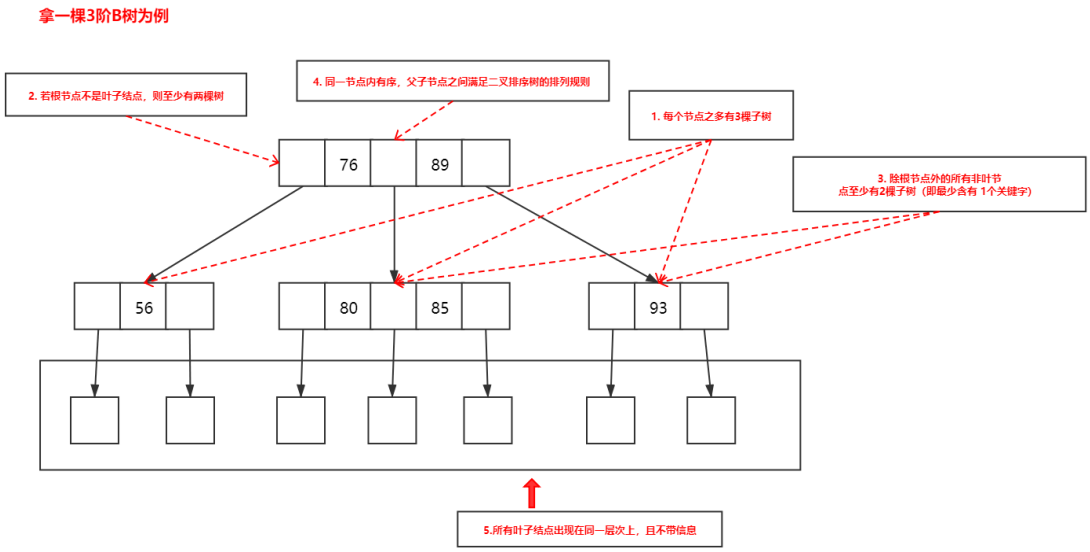
2. 若根结点不是叶子结点，则至少有两棵树，根结点关键字数量$[1, m-1]$，子树数量$[2, m]$

> **解释：** 根结点如果不是叶子结点（在B树中，叶子结点是没有信息的，也就是null），那么根结点至少有两个指针域（换句话说，至少有两棵子树）。如果没有两棵子树怎么办（比如，只有两个key），那么，将这两个key合并为一个结点（如果无法合并，比如m=2，那就无法构成B树）。B树的根结点至少有一个关键字，最多有m-1个关键字（也就是有m个指针域，m棵子树）

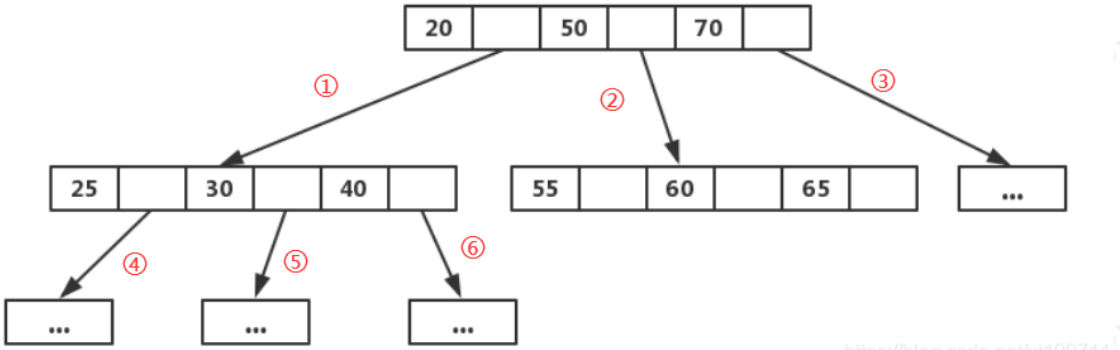
3. 除根结点外的所有非叶结点至少有 $\frac{m}{2}(向上取整)$ 棵子树（即最少含有 $\frac{m}{2}(向上取整)-1$ 个关键字）,除根结点外的所有非叶结点关键字数量$[\frac{m}{2}(向上取整)-1, m-1]$，子树数量$[\frac{m}{2}(向上取整), m]$

> **解释：** 多有除了根结点的非叶子结点，子树数量的下限是被控制的，不像根结点一样，最少可以有2棵子树，而是最少要有 $\frac{m}{2}(向上取整)$ 棵子树

4. 同一结点内有序，父子结点之间满足二叉排序树的排列规则
5. 所有叶子结点出现在同一层次上，且不带信息



图示为一棵B树。如果要寻找 `30` 的话，比较根结点 `{20, 50, 70}`，因为 $20 \leq 30 \leq 50$，走①号线；得到的子结点 `{25, 30, 40}`，找到了值为 `30` 的结点



综合来说： B树在提高效率就像二叉排序树一样，不同的是，面对不同阶的树，其结点上的值可能不止一个

那么，每棵B树的结点多少个是如何决定的呢?

比如说，对于任意一棵包含n（n≥1）个关键字、高度为h、阶数为m的B树：

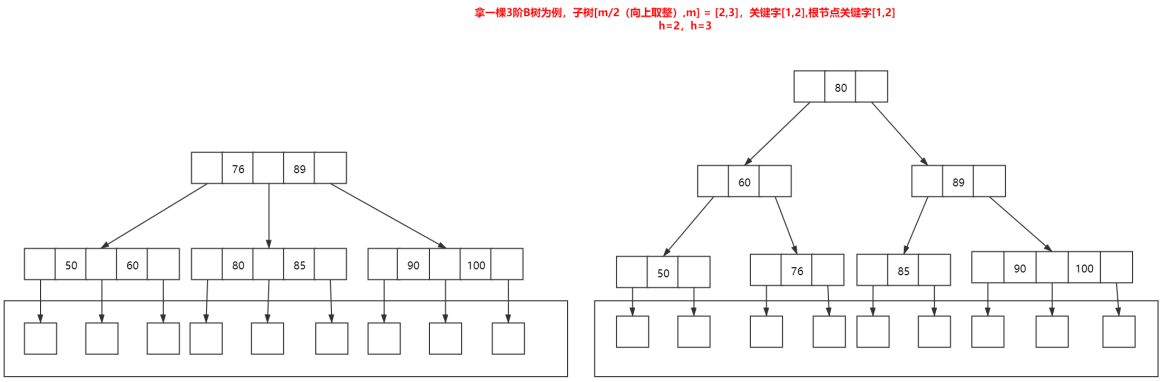
也就是说，这一棵B树的结点中的值的个数在 $[1, m-1]$

一棵B树的高的范围为

$$h \in [log_m(n+1), log_{\frac{m}{2}}(\frac{n+1}{2})+1](\frac{m}{2}向上取整) \quad (1)$$

例如：一棵3阶B树，有8个结点，那么它的 h∈[2,3.17]

、



做一个思考，把100挪到90以下可以吗？为什么？

（答案显然是不可以的，首先，不满足第3条，即非根结点需要的子树是$[\frac{m}{2}(\text{取上限}),m]$，在本示例中为$[2,3]$），把100挪到90以下,90只有一棵子树了；其次不满足第5条，即叶子结点在同一层)

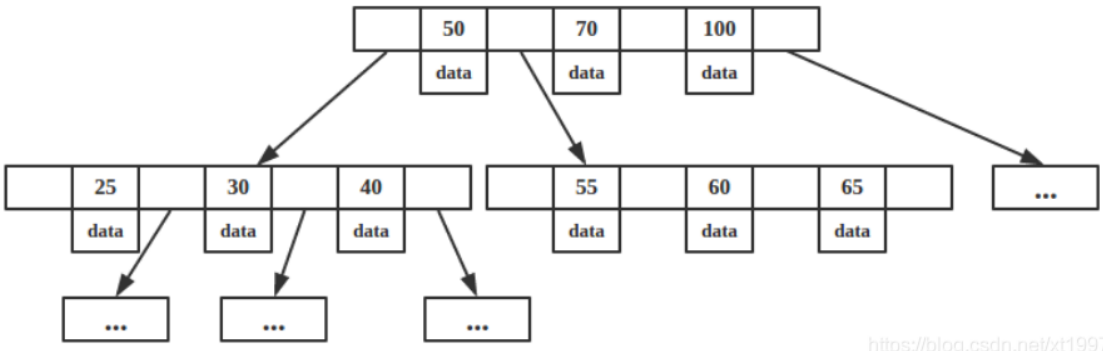
B树的查找

接下来让我们来看看B树的查找。

总的来说B树的查找分为两个基本操作：

1. 在B树中寻找结点
2. 在结点内找关键字

下面，假设每个结点有 n 个 key值，被分割为 n+1 个区间，注意，每个 key 值紧跟着 data 域，这说明B树的 key 和 data 是聚合在一起的。这里给出一棵B树



一般而言，根结点都在内存中，B树以每个结点为一次磁盘 IO，比如上图中，若搜索 key 为 25 结点的 data，首先在根结点进行二分查找（因为 keys 有序，二分最快），判断 key 25 小于 key 50，所以定位到最左侧的结点，此时进行一次磁盘 IO，将该结点从磁盘读入内存，接着继续进行上述过程，直到找到该 key 为止。

参考代码

```
Data* BTreeSearch(Root *node, Key key)
{
    Data* data;
    if(root == NULL)
        return NULL;
    // 二分法查找目标key
    data = BinarySearch(node);
    if(data->key == key)
    {
        return data;
    }else{
        // 若没有找到，对磁盘进行一次IO操作
        node = ReadDisk(data->next);
        BTreeSearch(node, key);
    }
}
```

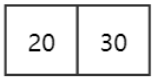
B树的插入

与二叉查找树相比，B树的插入就复杂得多了。因为B树的定义和限制就决定了这一点。

下面用图片的方式，给大家介绍B树插入过程中需要注意的问题

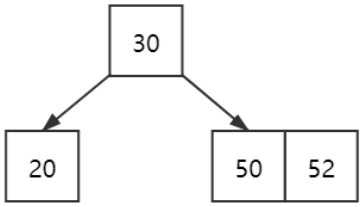
这是一棵3阶B树，给出一组关键字{20, 30, 50, 52, 60, 68, 70}，根据B树的性质，根结点的关键字数量调整范围为[1,2]，其他非叶结点关键字数量调整范围为[1,2]

step1： B树为空，20插入，作为根结点，此时，根结点仅1个关键字；然后30插入，由于比20大，所以插入20的右边，此时，根结点关键字个数为2个，满足B树要求

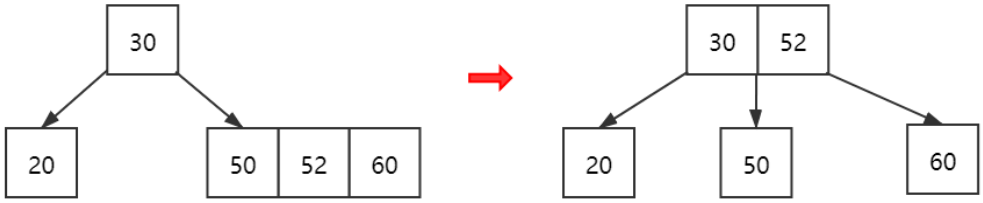


step2： 50插入，作为根结点，此时，根结点有3关键字，不满足B树的定义要求，所以需要分裂；按B树的分裂方法，应将中间的数提出，作为左右关键字的父结点，左右关键字作为其子结点，如下图。

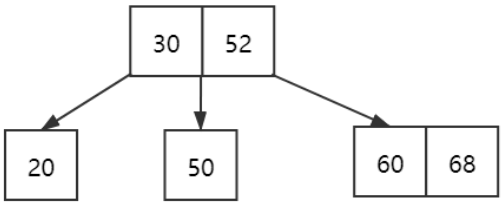
step3： 52插入，52比30大，来到右边的子树，52有比50大，所以插入到50右边。此时这个结点的关键字数量为2，满足B树定义要求。



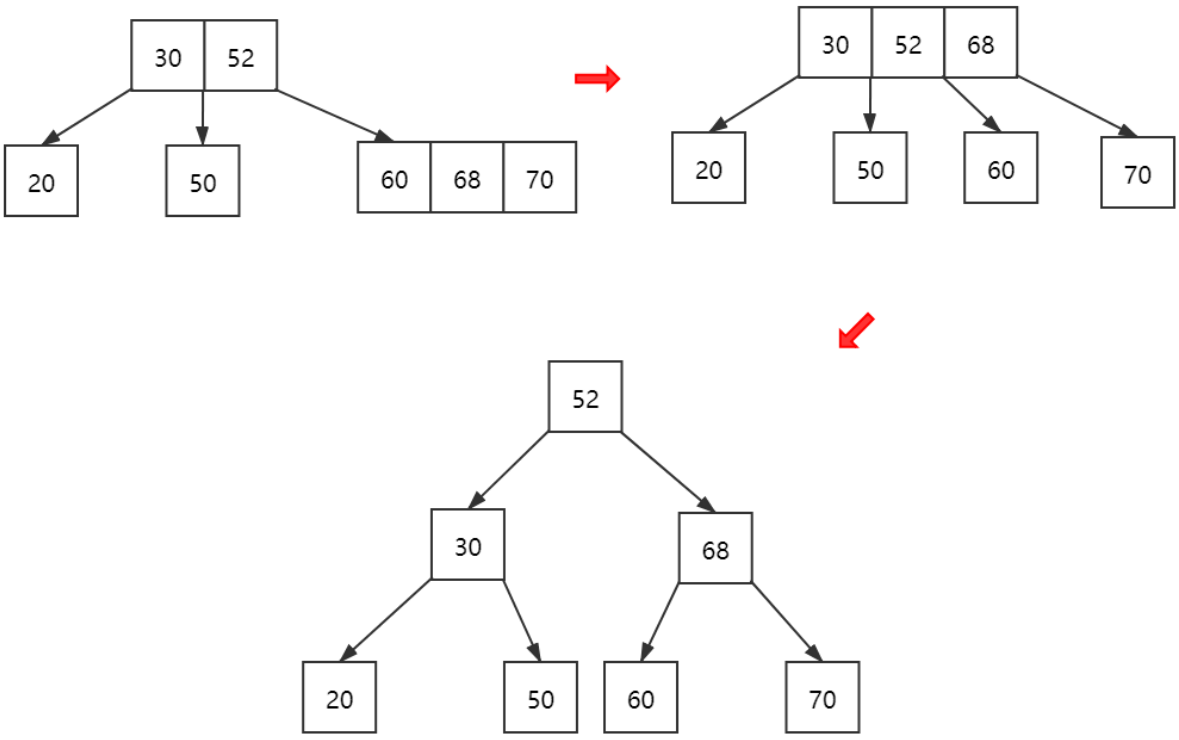
step4： 60插入，52比30大，来到右边的子树，60有比52大，所以插入到52右边。但此时，这个结点的关键字数量为3，不满足B树定义要求。按照B树分裂原则，将中间值52提出作为左右关键字的父结点，左右关键字作为52的子结点，如下图。



step5： 68插入，68比52大，来到右边的子树，68有比60大，所以插入到60右边。此时这个结点的关键字数量为2，满足B树定义要求。



step6： 最后，70插入，70比52大，来到右边的子树，70有比68大，所以插入到68右边。此时这个结点的关键字数量为3，不满足B树定义要求。按照B树分裂原则，将中间值68提出作为左右关键字的父结点，左右关键字作为68的子结点，如下图2。但此时，他们的父结点，也是根结点的关键字数量为3，不满足B树定义要求。按照B树分裂原则，将中间值52提出作为左右关键字的父结点，左右关键字作为52的子结点，如下图3 。



最后浓缩提炼总结

1. 检查插入结点后，结点关键字是否大于B树定义的结点关键字上限
2. 如果超过上限，那就采用结点分裂，将中间的结点作为左右关键字的父结点
3. 之后再次检查，提出后结点进入父结点后，该结点关键字是否大于B树定义的结点关键字上限
4. 如果超过上限，那就采用结点分裂，将中间的结点作为左右关键字的父结点（1,2循环，加一个检查）

B树的删除

相对来说，B树的删除较为复杂，不过原理都是类似的，就是满足B树的定义。也就是说，最基本的一点，要是删除后的结点中的关键字个数≥ $\frac{m}{2}(\text{向上取整})-1$，因此不可避免的，在删除的过程中，涉及结点的合并问题。

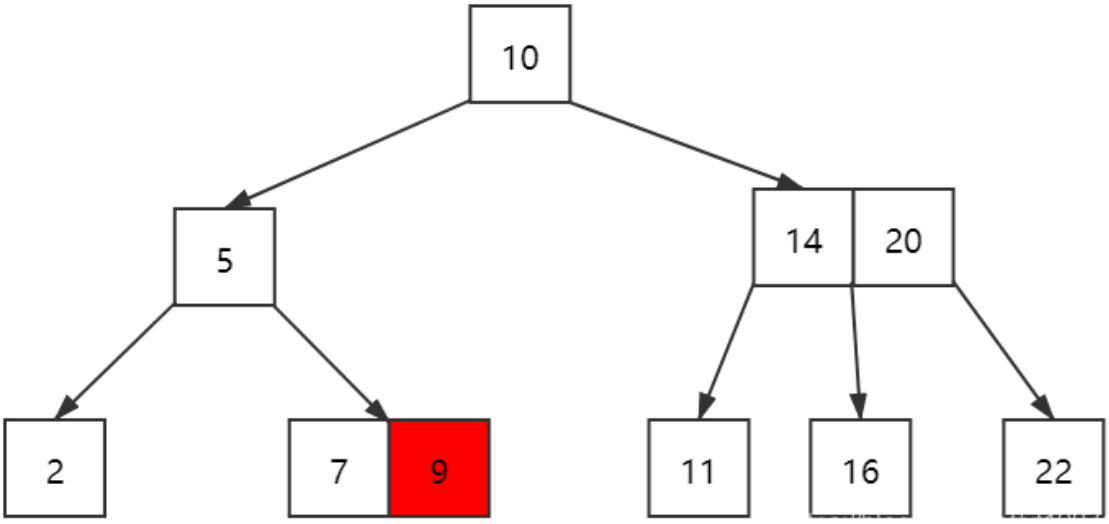
根据删除的关键字位置不同，可以分为关键字 **在终端结点上** 和 **不在终端结点上**

下面同样结合图片说明B树的删除是一个怎样的过程（所举的例子都是3阶B树，也就是说，B树的根结点的关键字数量范围为$[1,2]$，非根非叶子结点的关键字数量范围为 $[1,2]$）。

1. 如果删除的关键字在终端结点上

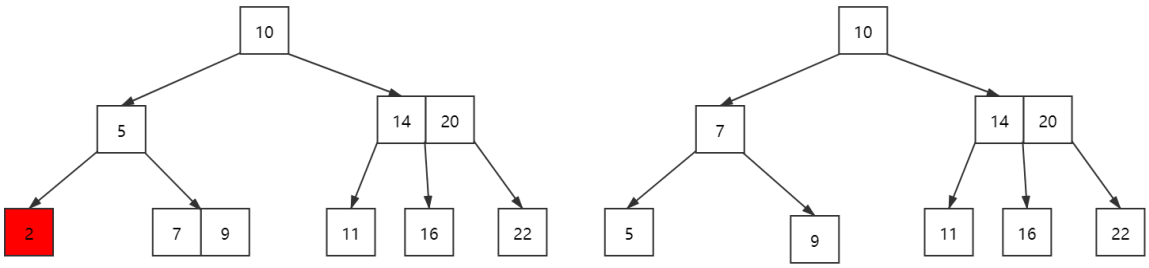
① 结点内关键字数 **大于** $\frac{m}{2}(\text{向上取整})-1$，这是删除这个关键字不会被破坏B树的定义要求，所以直接删除。

如下图，删除9，直接删除



② 结点内关键字数 **等于** $\frac{m}{2}(向上取整)-1$，并且其左右兄弟结点存在关键字数大于 $\frac{m}{2}(向上取整)-1$ 的结点，则去兄弟结点中借关键字。

如下图，删除2，由于结点中的关键字数量 **等于** $\frac{m}{2}(向上取整)-1=1$，恰好其兄弟结点{7,9}的关键字大于1，向兄弟结点借一个关键字，且在调整的时候需要按照大小顺序进行排序。

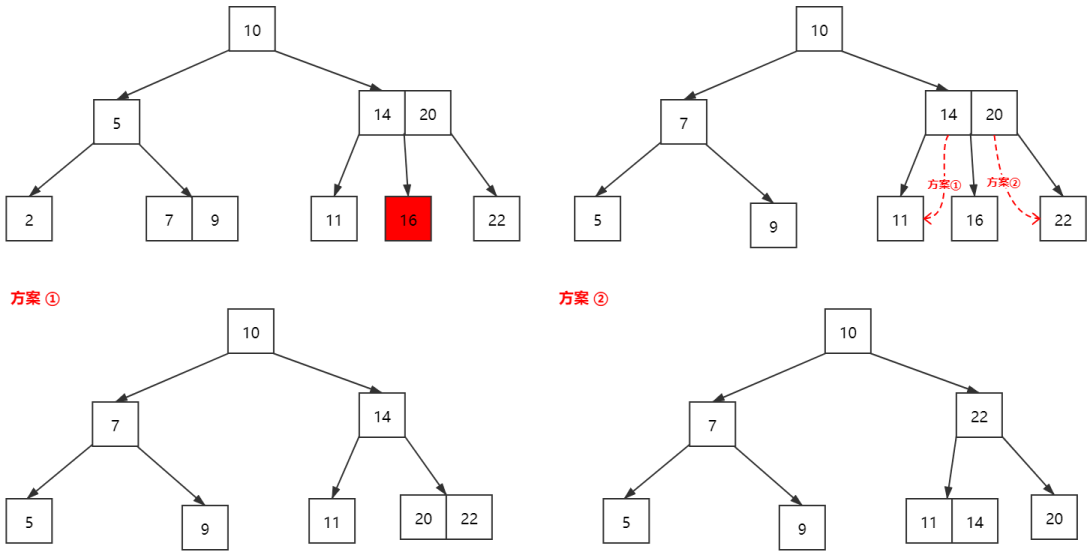


③ 结点内关键字数 **等于** $\frac{m}{2}(向上取整)-1$，并且其左右兄弟结点不存在关键字数大于 $\frac{m}{2}(向上取整)-1$ 的结点，则需要对结点进行合并。

如下图，删除16，由于结点中的关键字数量 **等于** $\frac{m}{2}(向上取整)-1=1$，但是其兄弟结点的关键字均等于1，所以不存在像兄弟结点借一个关键字这样的情况。所以需要进行结点合并。

合并：把上一层的结点取关键字与下一层结点合并。这种合并方式不唯一。

如方案①是把父结点的14与11合并；方案②是把父结点的20与22合并



2. 如果删除的关键字不在终端结点上

需要先转换成在终端结点上，在按照在终端结点上的情况来分别考虑对应的方法

这里介绍一个概念，就是相邻关键字。

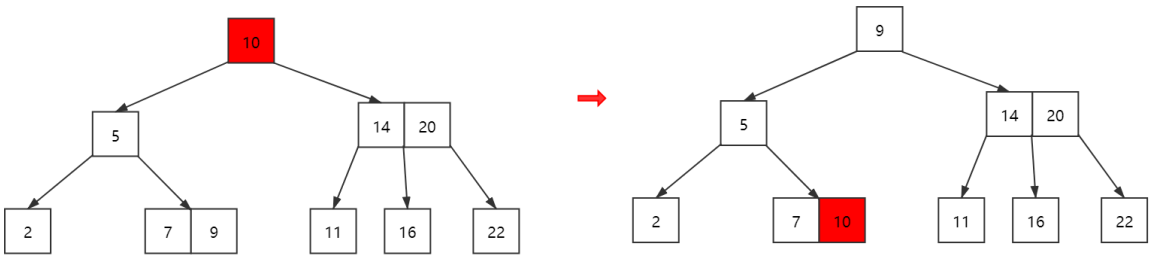
相邻关键字就是对于不在终端结点上的关键字，它的相邻关键字使其其 **左子树中最大的关键字** 或者 **右子树中最小的关键字**

第一种情况：存在关键字数量大于 $\frac{m}{2}(\text{向上取整})-1$（这里等于1） 结点的左子树和右子树，在对应子树上找到该关键字的相邻关键字，然后将相邻关键字替换待删除关键字

Step1：找出这个待删除关键字的相邻关键字，比如下图中10的相邻关键字急救室9和11，其实就是这个大小序列中该关键字的 **直接前驱或者是直接后继关键字**

Step2：将这个待删除的关键字和这个相邻关键字互换

Step3：这是就回到了删除终端结点的操作

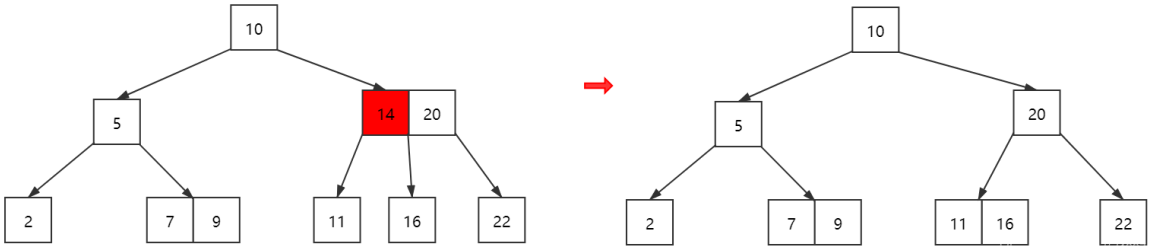


第二种情况：存在关键字数量均等于 $\frac{m}{2}(\text{向上取整})-1$ （这里等于1），则将这两个子树结点合并，然后删除待删除关键字

Step1：首先观察待删除结点14，发现它的左右子树中关键字均等于1

Step2：将14的左右子树合并为{11,6}

Step3：删除14，然后将{11,6}结点作为20的左结点

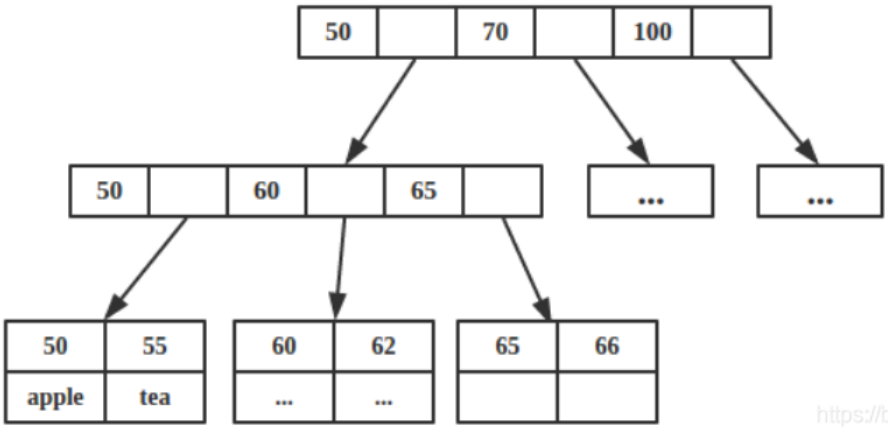


B+树

B+树的概念

B+树是B树的变种，它与B树的不同之处在于：

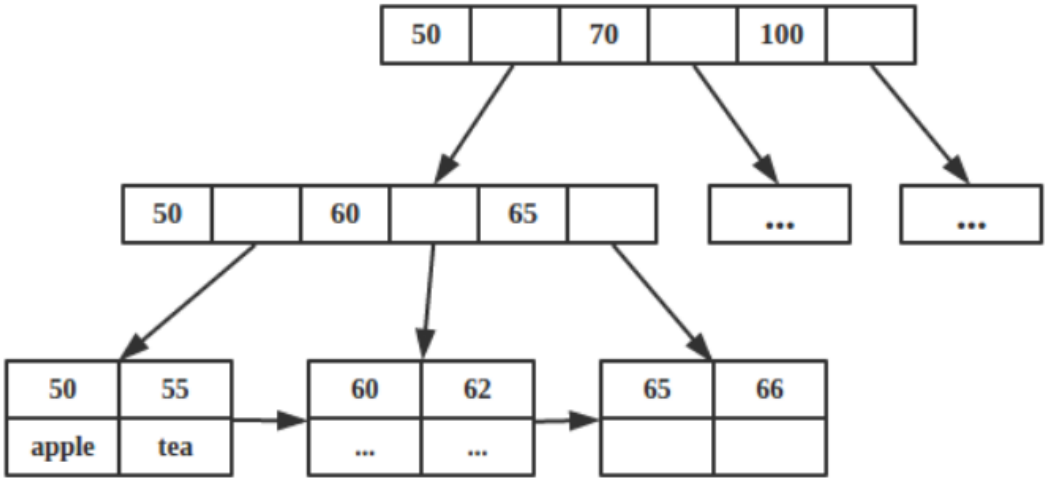
- 在B+树中，key 的副本存储在 **内部结点**，真正的 key 和 data 存储在叶子结点上 。
- n 个 key 值的结点指针域为 n 而不是 n+1。



因为内结点并不存储 data，所以一般B+树的叶结点和内结点大小不同，而B-树的每个结点大小一般是相同的，为一页。

为了增加 区间访问性，一般会对B+树做一些优化。

如下图带顺序访问的B+树。



带顺序访问的B+树

B+树的特点

一棵m阶的B+树需满足下列条件

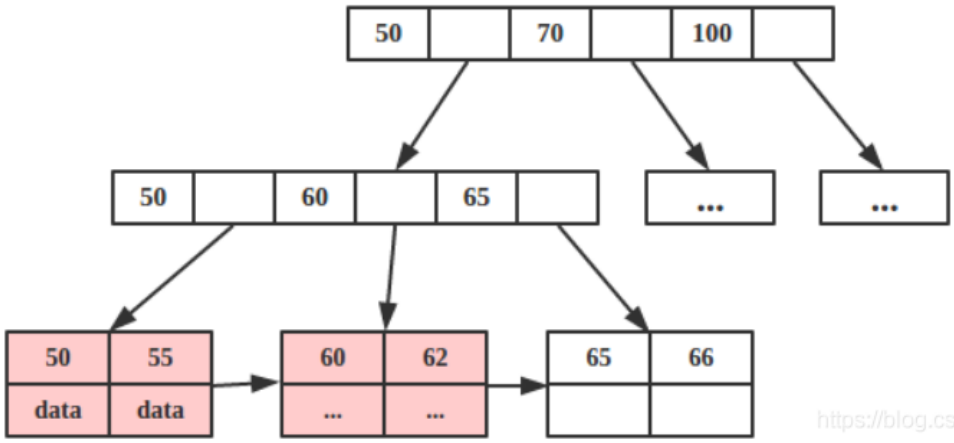
1. 每个分支结点最多有m棵子树（子结点）（与B树一样）
2. 非叶根结点至少有两棵子树，其他分支结点至少有$\frac{m}{2}$(向上取整) 棵子树（与B树一样）
3. 结点的子树个数与关键字个数相等（与B树不一样，B树的子树比关键字多1）
4. 所有叶结点包含全部关键字以及指向相应记录的指针业界点钟将关键字按大小顺序排序，并且相邻叶结点按大小顺序线连接起来

> **解释**：B+树叶结点两两相连可大大增加区间访问性，可使用在范围查询等，而B-树每个结点 key 和 data 在一起，则无法区间查找。

> 根据空间局部性原理：如果一个存储器的某个位置被访问，那么将它附近的位置也会被访问

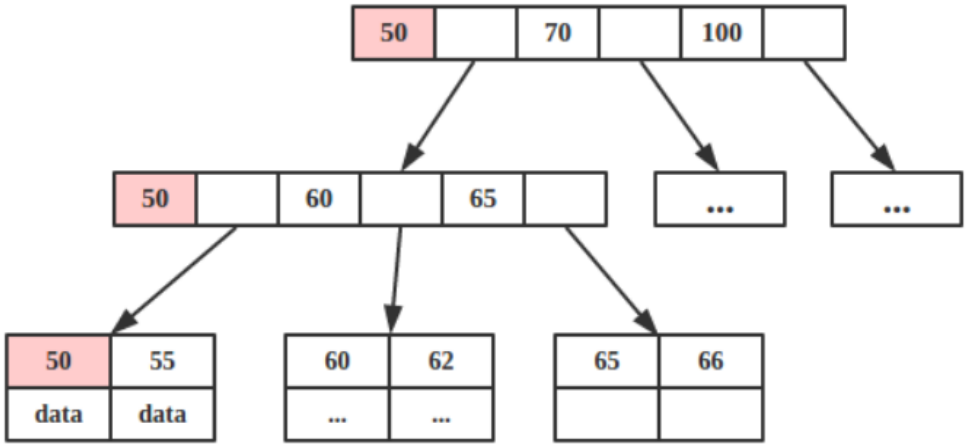
B+树可以很好的利用局部性原理，若我们访问结点 key为 50，则 key 为 55、60、62 的结点将来也可能被访问，我们可以利用磁盘预读原理提前将这些数据读入内存，**减少了磁盘 IO 的次数。**（根本目的）

当然B+树也能够很好的完成范围查询。比如查询 key 值在 50-70 之间的结点。



5. 所有分支结点（可视为索引的索引）中仅包含它的各个子结点（即下一级的索引快）中关键字的最大值及指向其子结点的指针

> **解释**：B+非叶结点只做索引，没有含该关键字对应的存储信息



B+树

同样的m阶B树与m阶B+树有什么区别

B树	B+树
n个关键字的结点有n+1棵子树	n个关键字的结点有n棵子树
每个结点（非根、内部）的关键字个数范围为$[\frac{m}{2}-1(\text{向上取整}),m-1]$ 根结点的关键字个数范围为[1,m-1]	每个结点（非根、内部）的关键字个数范围为$[\frac{m}{2}(\text{向上取整}),m]$ 根结点的关键字个数范围为[1,m]
每个结点的关键字都是包含全部信息的，也就是说，每个结点既包含了该关键字的data域，又包含了关键字的指针域（key）	叶结点包含信息，所有非叶结点均起索引作用，非叶结点中的每个索引只含对应字数的最大关键字和指向该子树的指针，并不含该关键字对应的存储信息。
叶结点包含的关键字与其他结点包含的关键字是不重复的	叶结点包含了全部的关键字，也就是说，在非叶结点中出现的关键字也会出现在叶结点中
时间复杂度最好是O(1)	时间复杂度固定为log_n
每个结点 key 和 data 在一起，则无法区间查找	叶结点两两相连可大大增加区间访问性，可使用在范围查询等
	更适合外部存储，由于内结点无 data 域，每个结点能索引的范围更大更精确
B树为有序数组+平衡多叉树	而B+树为有序数组链表+平衡多叉树

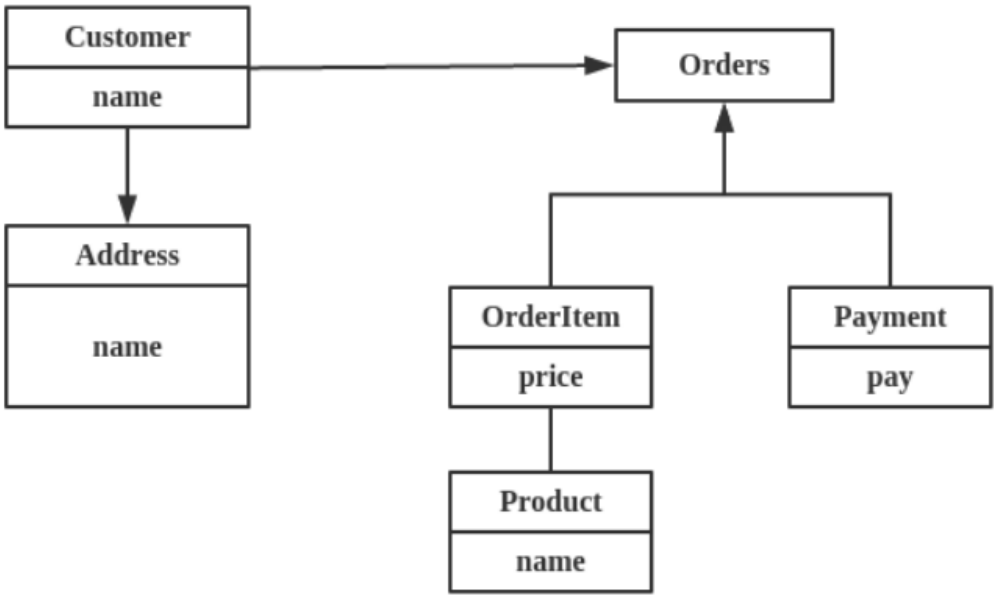
不同的数据库使用不同的树的原因在哪里

1. 为什么 MongoDB 使用B-树

MongoDB 是一种 nosql，也存储在磁盘上，被设计用在数据模型简单，性能要求高的场合。性能要求高，看看B/B+树的区别第一点：

B+树内结点不存储数据，所有 data 存储在叶结点导致查询时间复杂度固定为 log n。而B-树查询时间复杂度不固定，与 key 在树中的位置有关，最好为O(1)

我们说过，尽可能少的磁盘 IO 是提高性能的有效手段。MongoDB 是聚合型数据库，而 B-树恰好 key 和 data 域聚合在一起。



聚合型数据库

2. 为什么 Mysql 使用B+树

Mysql 是一种关系型数据库，区间访问是常见的一种情况，而 B-树并不支持区间访问（可参见上图），而B+树由于数据全部存储在叶子结点，并且通过指针串在一起，这样就很容易的进行区间遍历甚至全部遍历

1. B+树叶结点两两相连可大大增加区间访问性，可使用在范围查询等，而B-树每个结点 key 和 data 在一起，则无法区间查找
2. B+树的查询效率更加稳定，数据全部存储在叶子结点，查询时间复杂度固定为 O(log n)
3. B+树更适合外部存储。由于内结点无 data 域，每个结点能索引的范围更大更精确

B+树更适合关系型数据库

Customer	
Id	Name
1	Tom

Orders		
Id	CustomerId	ShippingAddressId
99	1	1

Product	
Id	Name
1	Book

Address	
Id	City
1	China

OrderItem			
Id	OrderId	ProductId	Prices
100	1	1	100

Payment	
Id	OrderId
1	1