# B树

B树也叫B-树或是平衡多路查找树，这几个名字说的是同一个东西

B树是一颗平衡树，它不会像普通BST树那样变成一条链子

B树不是二叉树而是多叉树，一个节点可以有很多个儿子，所以它相当于是多叉平衡查找树

## 性质

对于一个m(m>=3)阶B树有如下性质

1. 每个节点最多m-1个关键字，最多m个儿子，可以说几阶B树就最多几个儿子
2. 每个非根非叶子节点至少m/2个儿子，根节点至少2个儿子。
3. 所以的NULL节点都在同一层
4. 有j个儿子的节点有j-1个关键字，关键码按照递增排序
5. 满足搜索树性质，关键字keylist[i]和keylist[i+1]之间的节点以及它的所有子节点的数都介于区间(keylist[i],keylist[i+1])

keylist[0]之前的节点以及它的所有子节点的数都介于区间(-oo,keylist[0])

keylist[sz-1]之后的节点以及它的所有子节点的数都介于区间(keylist[sz-1],+oo)

B树也可以像红黑树等其他平衡树那样，在每个节点储存<key,value>对key建树，用来索引对应value,在这片文章中，为了方便看，B树每个节点省略了value

还有是，不论B树还是红黑树，有些地方把空节点看成叶子节点(因为红黑树里空节点算黑色节点)，但我习惯于把空节点就叫空节点，叶子结点指最底下的节点 。

**其他性质**

## 节点数据

keylist[] //

vallist[] //

son[] //

p //父节点

为了方便，sz表示节点keylist的元素个数，由B树性质，sz+1∈[m/2,m]

## 查找算法

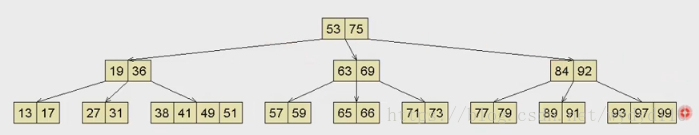
方法是从从根节点开始，每遇到一个节点，就遍历它的keylist, 若找到就返回true

但若带查找的k出现：

* k<keylist[0],说明要找的数据可能存在于son[0]里，往son[0]走继续找
* keylist[i]<k<keylist[i+1],说明要找的数据可能存在于son[i+1]里，往son[i+1]走继续找
* k>keylist[sz-1],说明要找的数据可能存在于son[sz]里，往son[sz]走继续找

可见B数的keylist元素永远是升序的

按照上述步骤，找到叶子也没找到，说明这个元素不存在，再插入算法里，这个位置就是要插入的位置。



如果查找75

先对根节点进行一次顺序查找，正确命中75。

如果查找69

先对根节点进行一次顺序查找，这次查找以失败终止于介乎53和75之间的这个引用。对63和69所在的这个节点进行一次顺序查找，成功找到69。

如果查找45

 先对根节点进行一次顺序查找，这次查找以失败终止53左侧的引用。将19和36这个节点通过IO载入内存，并做一次顺序查找，这次查找以失败终止于36右侧的引用，......最终查找失败于41与49之间的引用，底下是节点了，说明查找

## 插入算法

B树插入算法，首先初始化一个root,初始里面没有数据，对于每个数据，需要确认插入的位置，通过查找算法找到要插入的叶子结点的相应位置，之后只需记住核心思想是”分裂”：

1. 直接插入在相应位置，如果该节点<m，则什么都不做，插入结束。否则进入下一步
2. 此时节点关键字个数超过了m-1，要”分裂”：把keylist最中间的元素keymid拿出来，按照：keymid左边的元素，keymid本身，keymid右边的元素。分成3部分，把keymid本身插入到父亲节点里，放到合适的位置，把当前节点按第1和3部分分开成2个节点，放在父节点的keymid两侧。

如果当前节点是root怎么办，它没有父亲，那就创造个父亲，令它是root.

插入k完成后，如果有value应该把value也插入相应位置

1. 步骤二进行之后，可能父节点也不满足B树性质，即关键字个数超过了m-1，应该把它在分裂。所以B树插入是个自下而上的递归过程,直到分裂完root为止：for(now=叶子;now!=NULL&&now-> keylist.size()<m;now=now->parent){}

<https://www.01hai.com/note/av124483>

## 查找前驱后继

前驱：比某个数小的最大的数；

查找前驱：和BST树类似，对于查找node中的某个数keylist[i]的前驱，先向

keylist[i-1]到keylist[i]之间的分叉走(如果是keylist[0]直接走son[0])，也就是沿着son[i]走一个节点，然后一直沿着son[sz]走，直到走到叶子节点，这个节点的keylist[sz-1]就是前驱。

后继:比某个数大的最小的数

查找前驱：和BST树类似，对于查找node中的某个数keylist[i]的后继，先向

keylist[i]到keylist[i+1]之间的分叉走(如果是keylist[sz-1]直接走son[sz])，也就是沿着son[i+1]走一个节点，然后一直沿着son[0]走，直到走到叶子节点，这个节点的keylist[0]就是后继。

B树的NULL节点都在同一层，也就是叶子节点下面，且如果son[sz]存在，则son[0]，son[1]…..son[sz]全都存在，如果son[sz]不存在，则son[0]，son[1]…..son[sz]全都不存在，不会出现像bst树那种，前驱后继不是叶子节点的情况。

## 删除算法

删除操作类似BST树，要先看是不是在叶子，如果不在叶子，找到这个元素的后继或者前驱，替换它，而后继或者前驱一定在叶子节点。

那么问题变成了如何删除叶子的某个值，定义里说B树节点的值个数不能小于m/2，所以删除节点可能会破坏这个性质。步骤是

1. 直接删除，如果keylist.size()>=m/2什么都不做，否则继续下面步骤
2. 此时keylist.size()<m/2，首先试图从兄弟那里索取一个数，(左兄弟的keylist[sz-1]或者右兄弟的keylist[0])。

兄弟节点的keylist.size()必须满足keylist.size()>=m/2+1,因为得要完了也满足B树性质才可以借。如果可以索取，也不是直接拿过来，因为直接拿不满足搜索树的性质(B树这两个节点应该，左边的所有数都要比右边的小)

假设被拿的数是a，这两个节点的父亲的夹值是keymid，先把keymid放到当前缺少元素的节点，再把a放到原本keymid的位置

此后B树满足性质，删除的调整操作结束。

1. 若不能从兄弟那里索取一个数，此时说明兄弟节点的keylist.size()==m/2,而且自己本身的keylist.size()<m/2 ，应合并自己和兄弟节点,同时把这两个节点的父亲的夹值是keymid放到合并后的节点里。但此时由于从父亲拿了一个数，父亲可能出现keylist.size()<m/2的情况，要继续检查父亲节点，并且调整，

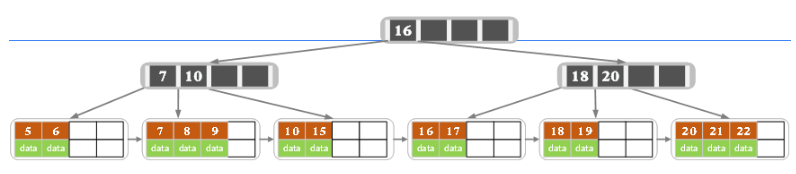
所以B树删除是个自下而上的递归过程,直到检查合格或者到root为止。

for(now=叶子;now!=root&&now-> keylist.size()>=m/2;now=now->parent){}

# B+树

B+树是B树的变体，B+树的空间消耗量比B树更大，它是为了做<key,value>这种索引而设计的，在索引的扩展性和效率上比B树好。

B+树和B树区别在于，B+树节点分为两种，叶子节点和内部节点，内部节点只储存key,不出村value,而叶子结点储存key和value，叶子结点用链表连起来



## 性质

对于一个m(m>=3)阶B+树有如下性质

1. 每个节点最多m-1个关键字，最多m个儿子，可以说几阶B+树就最多几个儿子
2. 每个非根非叶子节点至少m/2个儿子，根节点至少2个儿子。
3. 所以的NULL节点都在同一层
4. 有j个儿子的节点有j-1个关键字，关键码按照递增排序
5. 满足搜索树性质，关键字keylist[i]和keylist[i+1]之间的节点以及它的所有子节点的数都介于区间[keylist[i],keylist[i+1])

keylist[0]之前的节点以及它的所有子节点的数都介于区间(-oo,keylist[0])

keylist[sz-1]之后的节点以及它的所有子节点的数都介于区间[keylist[sz-1],+oo)

1. 叶子结点通过左到右的指针连接，除了叶子结点，其他节点只存key,不存value

## 节点数据

内部节点：

keylist[] //

son[] //

p //父节点

叶子节点：

keylist[] //

vallist[] //

next //右节点指针

p //父节点

为了方便，sz表示节点keylist的元素个数，由B+树性质，sz+1∈[m/2,m]

## 查找算法

由于B+树叶子节点储存value,所以查找数据一定要找到叶子节点。

方法是从从根节点开始，每遇到一个节点，就遍历它的keylist,

但若带查找的k出现：

* k<keylist[0],说明要找的数据可能存在于son[0]里，往son[0]走继续找
* keylist[i]<=k<keylist[i+1],说明要找的数据可能存在于son[i+1]里，往son[i+1]走继续找
* k>=keylist[sz-1],说明要找的数据可能存在于son[sz]里，往son[sz]走继续找

按照上述步骤，一直走到叶子节点，若叶子节点的keylist没找到，说明这个元素不存在，再插入算法里，这个位置就是要插入的位置。如果找到说明存在，返回true.

## 查找前驱后继

## 插入算法

B+树插入算法，首先初始化一个root，且作为叶子结点,初始里面没有数据，对于每个数据，需要确认插入的位置，通过查找算法找到要插入的叶子结点的相应位置。B+树的插入的核心思想是”分裂”：

1. 直接插入在相应位置，如果该节点<m，则什么都不做，插入结束。否则进入下一步
2. 此时节点关键字个数超过了m-1，要”分裂”：把keylist最中间的元素keymid拿出来，按照：keymid左边的元素，keymid本身，keymid右边的元素。分成3部分，把当前节点分开成2个节点，keymid左边的元素在左边的节点，keymid本身和keymid右边的元素在右边的节点，拆开后它们还是叶子节点，通过next指针相连。同时把keymid的数值也加入到父亲节点的合适位置。

如果当前节点是root怎么办，它没有父亲，那就创造个父亲，令它是root，且这里创造的节点都是内部节点。

插入k完成后，如果有value应该把value也插入相应位置

1. 步骤二进行之后，可能父节点也不满足B+树性质，即关键字个数超过了m-1，应该把它在分裂。所以B+树插入是个自下而上的递归过程,直到分裂完root为止：for(now=叶子;now!=NULL&&now-> keylist.size()<m;now=now->parent){}

<https://www.01hai.com/note/av124483>

## 删除算法

B+树所有数据都存在叶子，所以待删除节点一定是叶子

那么问题变成了如何删除叶子的某个值，定义里说B+树节点的值个数不能小于m/2，所以删除节点可能会破坏这个性质。步骤是

1. 直接删除，如果keylist.size()>=m/2什么都不做，否则继续下面步骤
2. 此时keylist.size()<m/2，首先试图从兄弟那里索取一个数

(左兄弟的keylist[sz-1]或者右兄弟的keylist[0])。

兄弟节点的keylist.size()必须满足keylist.size()>=m/2+1,因为得要完了也满足B+树性质才可以借。如果可以索取，直接把它拿来：如果拿的是左兄弟的keylist[sz-1]，就把它放在当前节点最左边；如果拿的是右兄弟的keylist[0]，就把它放在当前节点最右边；

假设被拿的数是a，这两个节点的父亲的夹值是keymid，用a替换父亲节点原本的keymid

此后B+树满足性质，删除的调整操作结束。

1. 若不能从兄弟那里索取一个数，此时说明兄弟节点的keylist.size()==m/2,而且自己本身的keylist.size()<m/2 ，应合并自己和兄弟节点,

设靠左的节点是left,靠右的是right,它们的keylist顺序拼合即可。

同时对父亲节点删除数据，left和right的夹值。由于从父亲删除了一个数，父亲可能出现keylist.size()<m/2的情况，要继续检查父亲节点，并且调整，

所以B+树删除是个自下而上的递归过程,直到检查合格或者到root为止。

for(now=叶子;now!=root&&now-> keylist.size()>=m/2;now=now->parent){}