PostgreSQL-BTREE Index

理论基础

索引的基本概念

索引的目的：能够更快地**搜索**到满足条件的记录

如何实现一个数据库索引?

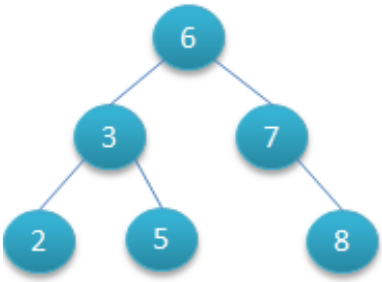
PG数据库存储引擎是基于页面Page(8K bytes)去管理磁盘数据，实现索引，选择哪种数据结构更好?

链表

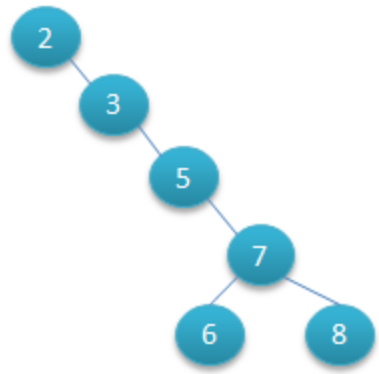
将所有的key-data排序，依次放在页面上，然后使用指针把页面串成一个链表
缺点很明显，插入、查询的时间复杂度都是O(n)

二叉搜索树

左子树的key小于根的key，右子树的key大于根的key
插入、查询最多只需要读取树高个节点，明显，时间复杂度都是O(log(n))



缺点：
1.每个节点对应一个页面，那么空间会有浪费
2.插入、删除可能会导致树不平衡，最坏的情况下会退化成链表，时间复杂度自然就退化成O(n)



平衡二叉树AVL

在二叉搜索树的基础上，通过自旋转保证任何节点的子树高度差最大不超过1
插入、查询的时间复杂度O(log(n))
并且根据自旋转流程，能够保证不退化成链表

缺点：
1.自旋转逻辑复杂
2.也不能解决存储空间浪费的问题

BTREE

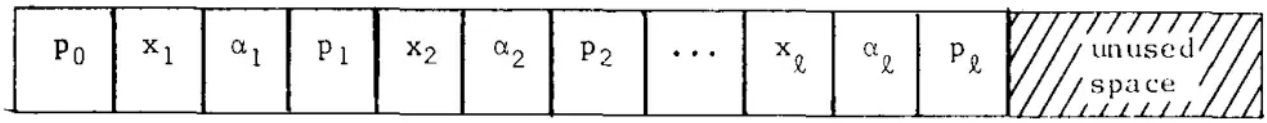
什么是BTREE?

"一般化的二叉搜索树"

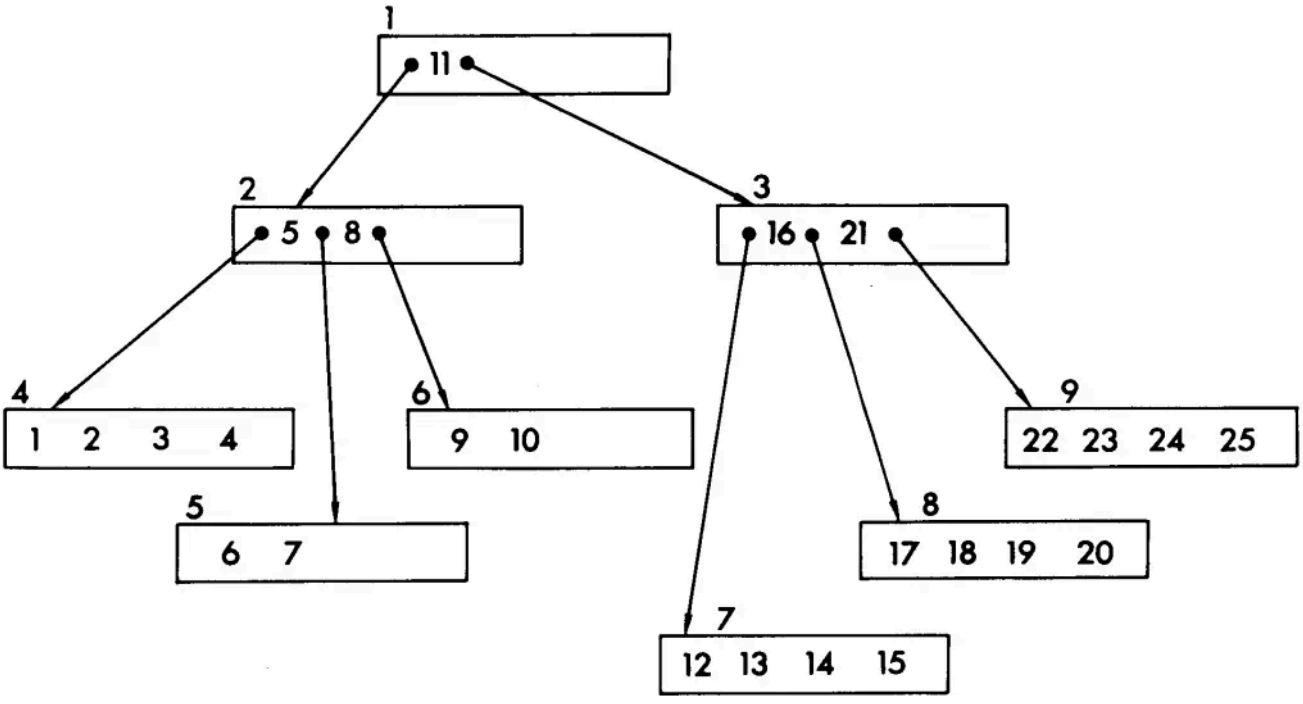
BTREE定义

1.从根节点到任意叶子节点的距离相等，为h，称为树高
2.每个节点（除了根节点）至少有k+1个儿子。根节点不是叶子节点，至少有2个儿子
3.每个节点至多有 2k+1 个儿子

节点结构：
p代表指针，x代表key，α代表value



2,3树，k=2, h =3



优势

1.树高被明显降低，搜索所需要的页面读取次数明显减少，时间复杂度O($\log_m n$)
2.也不需要像AVL树一样频繁进行复杂地平衡，因为一个节点可以拥有多个子节点，普通插入场景新加一个子节点就可以了。

B*TREE

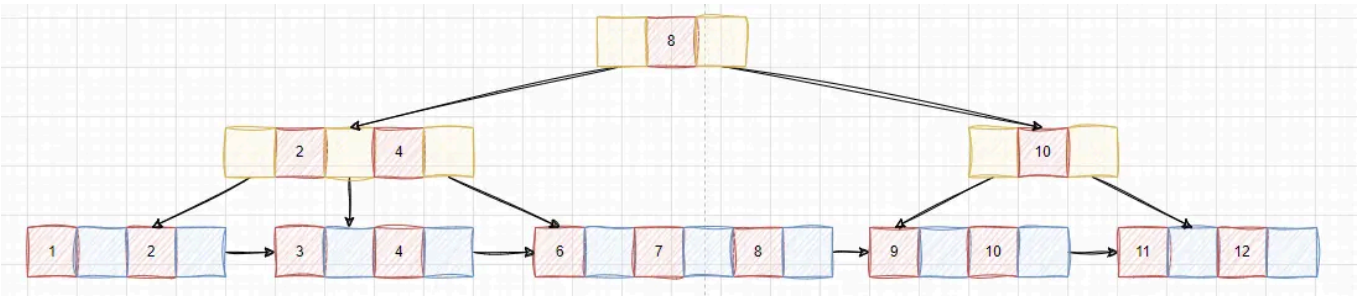
B*TREE是基于BTREE的变种

核心优化：
所有的data都放在叶子节点中，非叶子节点只存储key的信息

BTREE的实现，data可以存放在非叶子节点，如果data非常大，会导致非叶子节点存储的key变少，树高会增大，进而影响搜索效率，并且搜索性能也不稳定。
B*TREE的实现最大化了内部节点的子节点数目，树高相比BTREE更低，搜索会更快。

额外的，当所有数据放在叶子节点上，那么就可以在叶子节点层添加向右的指针，将所有叶子节点串起来，这样对于顺序查询有很大的提升。

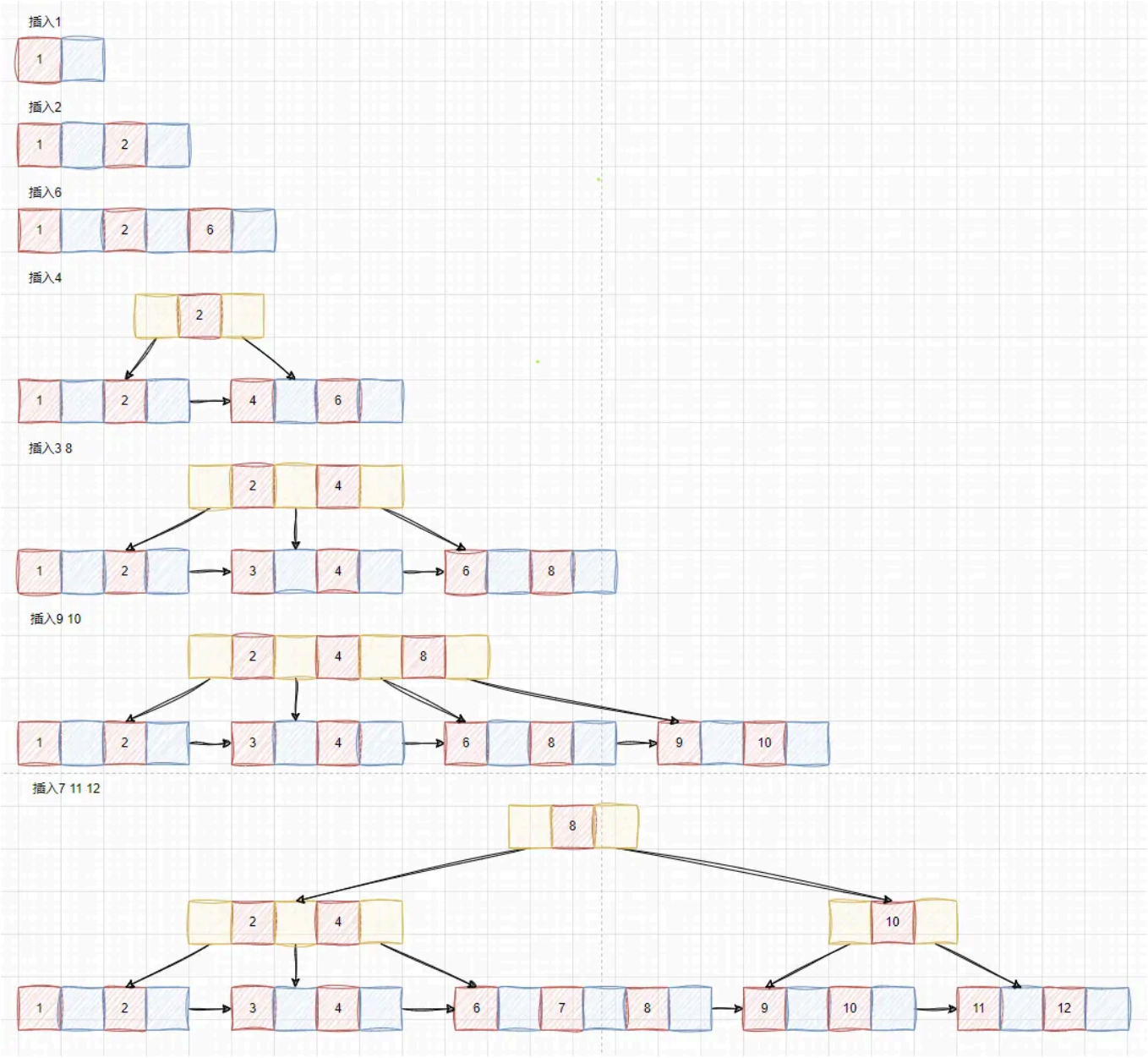
红色是key
黄色是指向下层的指针
蓝色是data



查询流程

从root节点开始，对key进行比较，跟随对应的下降指针，直到下降到叶子节点即可。

插入流程



B-link-tree

进一步思考性能问题

当前的数据结构能支持并发操作吗，是否会存在问题？

并发问题样例

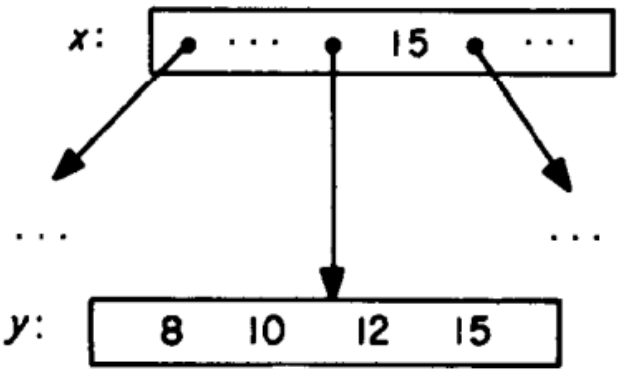
操作定义

小写字母x,y,current,定义为变量，指向磁盘上的页面
大写字母A,B,C定义为内存当中的block

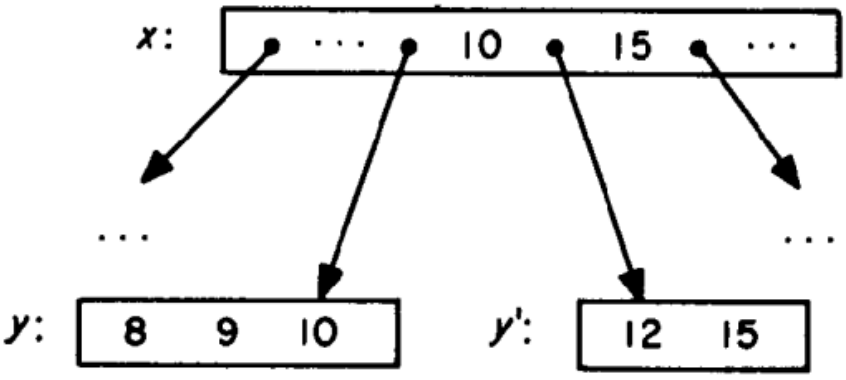
lock(x) 表示对于x指向的磁盘页面加互斥锁
unlock(x)表示对x指向的磁盘页面解锁
A<-get/read(x)表示读取x指向的页面到A内存块中
put(A,x)表示将A内存块中的数据写入x指向的磁盘页面

假定get、put操作都是不可分割的原子操作

考虑如下场景：
key=9的插入导致y分裂为y，y'，此时insert(9)和search(15)并发，查询操作可能无法看到页面已有数据



(a)



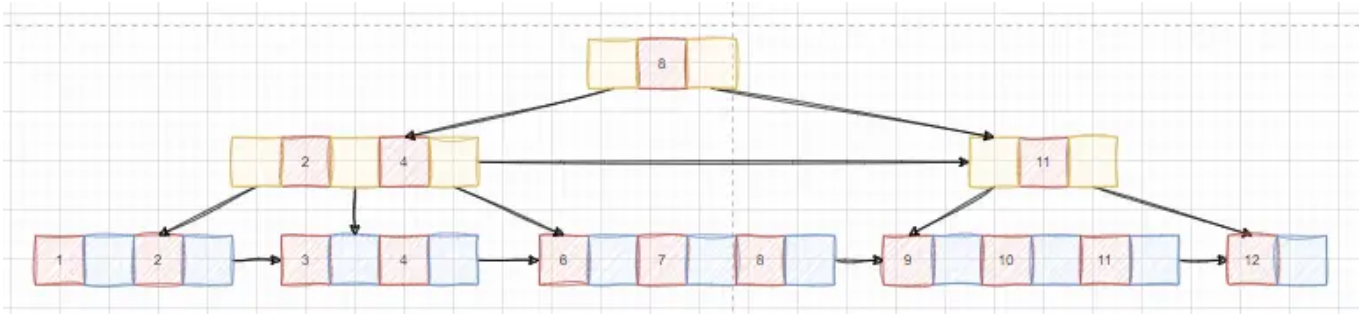
(b)

search(15)	insert(9)
1. $C \leftarrow$ read(x)	
2.	$A \leftarrow$ read(x)
3. examine C; get ptr to y	
4.	examine A; get ptr to y
5.	$A \leftarrow$ read(y)
6.	insert 9 into A; must split into A, B
7.	put(B, y')
8.	put(A, y)
9.	Add to node x a pointer to node y'.
10. $C \leftarrow$ read(y)	
11. *error*: 15 not found!	

定义

引入额外的”link“，在仅锁定很少的页面情况下，解决BTREE的并发问题。
引入highkey概念：
叶子节点，表示该页面上最大的key
非叶子节点，表示所有子树的的最大key

核心操作：在所有节点上添加指向右兄弟节点的指针



查询算法

定义操作：
x <- scannode(v, A)
在内存块A中扫描，寻找对于value v的合理位置，赋值给x（结果可能是link ptr，即指向右兄弟页面的指针）

算法伪代码

```
procedure search(v)
current ← root;                                   /* Get ptr to root node */
A ← get(current);                                 /* Read node into memory */
while current is not a leaf do
begin                                             /* Scan through tree */
   current ← scannode(v, A);                      /* Find correct (maybe link) ptr */
   A ← get(current)                               /* Read node into memory */
end;
                                                  /* Now we have reached leaves. */
while t ← scannode(v, A) = link ptr of A do
                                                  /* Keep moving right if necessary */
begin
   current ← t;
   A ← get(current)                               /* Get node */
end;
                                  /* Now we have the leaf node in which v should exist. */
if v is in A then done "success" else done "failure"
```

插入算法

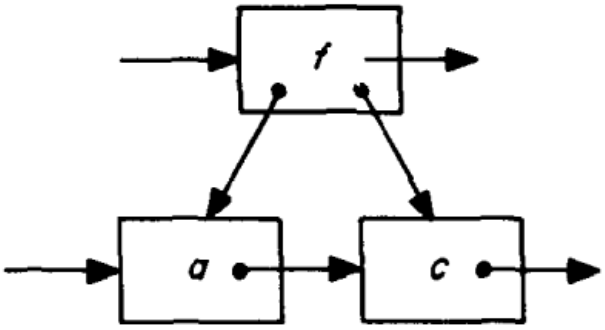
首先搜到到要插入的位置，该方式等同于上面的查询算法。
比较特殊的是在查询过程中要记录每次进行层级下降的节点（即维护一个路径stack）

当找到插入位置后，插入数据，会有2种情况：

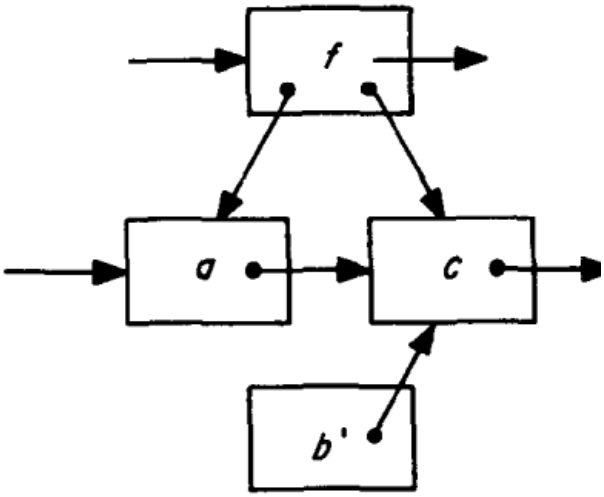
1. 页面空间足够，直接修改页面数据，直接插入即可（称为safe）
2. 页面空间不足，插入数据导致分裂（称为unsafe）
   1场景简单，无需额外讨论
   2场景详细讨论如下：
   当前已有页面结构(a)，新插入数据v，导致a页面分裂，页面分裂为a'和b'，a'和a对应同一个磁盘页面。

a'页面数据 + b’页面数据 = a页面数据+新插入数据v
b'页面指向c页面，a‘页面指向b'页面，再将a'的high key和指向b’的指针插入父节点，直到插入操作到达一个safe节点（即不再触发页面分裂）

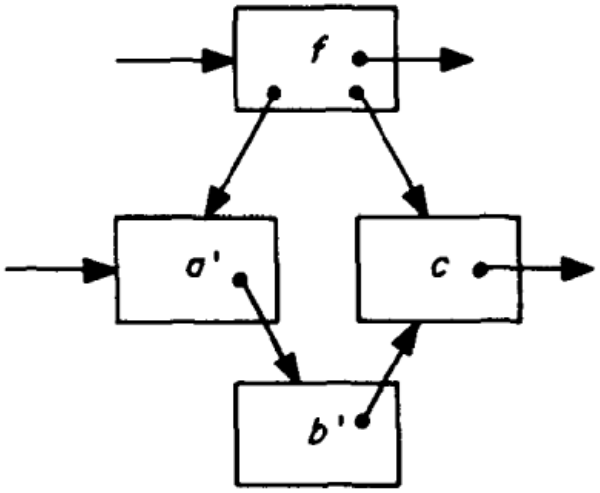
分裂示例
插入数据导致a页面分裂成a'和b'



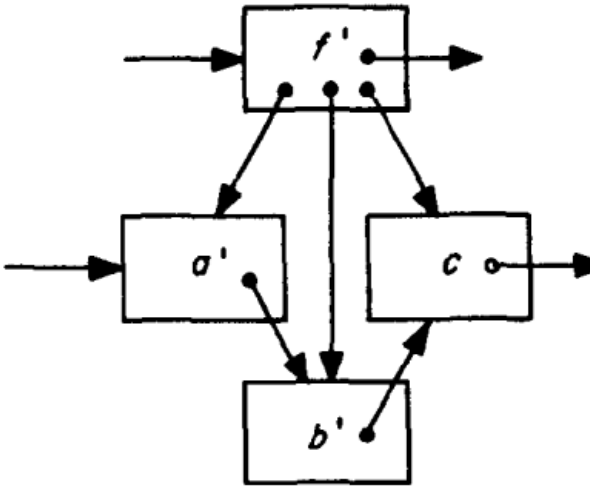
(a)



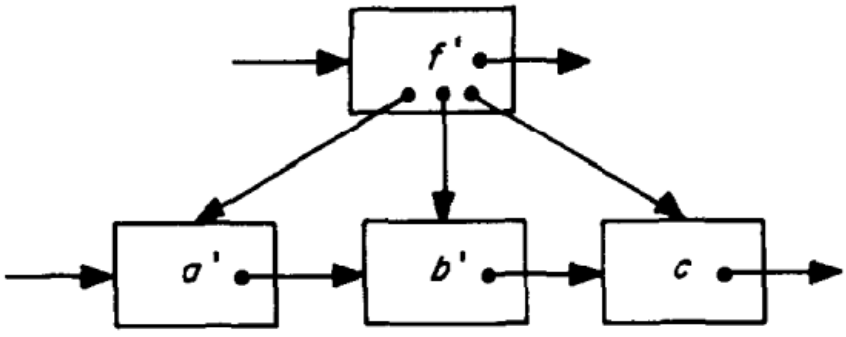
(b)



(c)



(d)



(e)

操作定义

A <- node.insert (A, w, v)

表示插入指针w和值v到内存块A中

u <- allocate(1 new page for B)

表示为内存块B申请一个新磁盘页面，赋值给指针u

A,B <- rearrange old A, adding...

表示将内存块A分裂为2个节点，内存块A、内存块B

算法伪代码

```
procedure insert(v)
initialize stack;                                        /* For remembering ancestors */
current ← root;
A ← get(current);
while current is not a leaf do
begin                                                    /* Scan down tree */
  t ← current;
  current ← scannode(v, A);
  if new current was not link pointer in A then
    push(t);                                             /* Remember node at that level */
  A ← get(current)
end;
lock(current);                                           /* We have a candidate leaf */
A ← get(current);
move.right;                                              /* If necessary */
if v is in A then stop "v already exists in tree";       /* And t points to its record */
w ← pointer to pages allocated for record associated with v;
Doinsertion:
if A is safe then
begin
  A ← node.insert(A, w, v);                              /* Exact manner depends if current is a leaf */
  put(A, current);
  unlock(current);                                       /* Success—done backtracking */
end else begin                                           /* Must split node */
  u ← allocate(1 new page for B);
  A, B ← rearrange old A, adding v and w, to make 2 nodes,
    where (link ptr of A, link ptr of B) ← (u, link ptr of old A);
  y ← max value stored in new A;                         /* For insertion into parent */
  put(B, u);                                             /* Insert B before A */
  put(A, current);                                       /* Instantaneous change of 2 nodes */
  oldnode ← current;                                     /* Now insert pointer in parent */
  v ← y;
  w ← u;
  current ← pop(stack);                                  /* Backtrack */
  lock(current);                                         /* Well ordered */
  A ← get(current);
  move.right;                                            /* If necessary */
  unlock(oldnode);
  goto Doinsertion                                       /* And repeat procedure for parent */
end
```

Move.right. This procedure, which is called by *insert*, follows link pointers at a given level, if necessary.

```
procedure move.right
while t ← scannode(v, A) is a link pointer of A do
begin                                                    /* Move right if necessary */
  lock(t);                                               /* Note left-to-right locking */
  unlock(current);
  current ← t;
```

正确性证明

无死锁

加锁的方向永远是向右向上，所以不会出现循环等待，不满足死锁的必要条件。

对树结构的修改正确

根据以上算法，有三处修改操作：

1. put(A, current) for safe node
2. put(B, u) for unsafe nodes
3. put(A, current) for unsafe nodes

1操作对应正常插入，2、3操作对应页面分裂操作
执行1操作时，仅修改页面数据，不会修改树结构，∴1操作不会影响树结构的正确性
执行2操作时，没有指针指向b，∴2操作不会影响树的结构正确性
执行3操作时，会修改a节点上的数据，以及将指针指向b，此时，b已经完成修改，所以对于树来讲，2、3操作可以视为一个对于树的不可分割操作，∴3操作不影响树结构的正确性

交互正确（业务并发正确）

除了修改进程外，所有的进程都可以看到一致的树

- 场景一：对于页面a，t0时刻发生写操作，对于任意的时刻t > t0，对于页面a的读操作都是正确的

基于前面的结论，put操作对于树结构的修改是正确的，所以自然成立

- 场景二：对于页面a，t0时刻发生写操作，时刻t‘ < t0，对页面a发生读操作
  第一种情况，写操作是对于safe node的插入
  (a)如果是叶子节点，插入数据不会修改树结构，查询正常
  (b)如果是非叶子节点，其插入为子层节点分裂引发，读操作获取的旧数据，无法直接访问分裂出来的右子节点，但是可以通过分裂的左子节点的link指针访问到分裂的右子节点，数据不会丢失，查询正常

第二种情况，写操作是对于unsafe node的插入
(c)如果是叶子节点，写操作使叶子节点分裂，分裂出来的两个页面数据，除了新插入的数据，等同于未分裂前节点的数据，查询正常
(d)如果是非叶子节点，子层的分裂引发父层节点插入数据，进而引起分裂。如果搜索的目标是向下，等同于(b)，查询正常，如果搜索的目标是向右，等同于(c)，查询正常

- 场景三：写写业务并发
  通过上面的总结，插入操作大致分为3步：下降查询找到leaf节点，插入数据，如果需要分裂，上升完成分裂

第一种情况，进程I写入页面a，进程I’处于写入流程中的下降查询插入位置
下降查询的操作其实等同于查询，没有差别

第二种情况，进程I写入页面a，进程I‘处于写入流程中的上升完成分裂
进程I’在下降时，记录了下降的路径（利用堆栈），当上升分裂

时，弹出栈顶节点，读取节点数据，此时这个节点可能已经被进程I分裂过，之前堆栈中记录的插入位置已经失效，但是没有问题，新的插入位置一定是在当前节点右侧，通过页面的link指针，向右寻找即可

第三种情况，进程I写入页面a，进程I‘同时写入页面a
进程I获取lock后，进程I'会被阻塞，直到进程I释放lock，依赖上述证明结论，I’对树的修改操作是正确的，所以进程I操作正常

liveLock

算法本身存在活锁的可能，即进程操作需要跟随link指针向右遍历，但永远无法找到需要的数据，因为整颗树在疯狂分裂，向右分裂的速度比起跟随link指针遍历更快
此场景非常极端，要求多个进程的速度差异巨大，才有可能出现该情况

删除

不清理非叶子节点，仅当叶子节点数据删除完成后清理叶子节点页面。
整个流程非常简单，同时只有一把锁，要删除哪个页面，锁哪个页面即可。

拓展思考

1. 一个插入流程最多加3个锁，哪三个锁，还可不可以更少
2. left-link存在要如何加锁
3. 根分裂怎么玩?

4. 如何回收中间节点?
5. fast-root?
6. 当前假设建立在磁盘提供原子接口，且同一个页面被读取到不同的内存块中，如果是基于数据库的bufferPool呢?