


# SimpleDB Lab3 实验报告

## commit记录

07 May, 2022 1 commit		
 Lab3 extra exercise Finish Test TTATT authored 2 hours ago	37e346ce	 
04 May, 2022 1 commit		
 Lab3 Finish Test TTATT authored 2 days ago	83f83b29	 
03 May, 2022 1 commit		
 Lab3 exercise3  TTATT authored 3 days ago	41724d49	 
02 May, 2022 1 commit		
 Lab3 exercise2 finish test TTATT authored 4 days ago	3bf4b492	 
01 May, 2022 1 commit		
 Lab3 exercise1 finish test TTATT authored 5 days ago	082933ae	 

## 实验设计思路





















### exercise1

在exercise1中要求在 `BTreeFile.java` 这个类中实现 `findLeafPage()` 这个方法。这个方法的作用就是在以B+树的结构管理文件中的页时，能够查找到含有指定 `Field` 的页面。

由于整个B+树的结构是按照 `Field` 有序构建的，因此在查找的过程中可以通过比较每一个内部叶节点中的 `Field` 和给定的 `Field` 之间的大小关系来确定下一个将要查找的内部叶节点是谁。

这个方法整体的设计思路是，首先根据提供的 `BTreePageId` 获取到当前的page，然后去判断这个page是否位叶节点，如果是叶节点则通过数据库的缓冲池来获取这个page然后作为结果返回。如果不是叶节点，则在当前的page中去遍历Entry，来比较当前遍历到的Entry的 `Field` 和给定的 `Field` 之间的大小关系，当找到一个比给定的 `Field` 大的Entry之后，就继续遍历这个Entry的左子树。直到找到含有给定 `Field` 的叶节点为止。

特别要注意的是，如果给定的 `Field` 为空的话，则默认查找最左侧的叶节点，因此在递归搜索的过程中，每次只需要通过迭代器获取第一个Entry然后去递归这个Entry的左子树即可。

	          	
	✓ BTreeFileReadTest (simpledb)	730毫秒
	✓ getld	624毫秒
	✓ getTupleDesc	11毫秒
	✓ readPage	7毫秒
	✓ indexIterator	38毫秒
	✓ numPages	7毫秒
	✓ testIteratorBasic	18毫秒
	✓ testIteratorClose	25毫秒

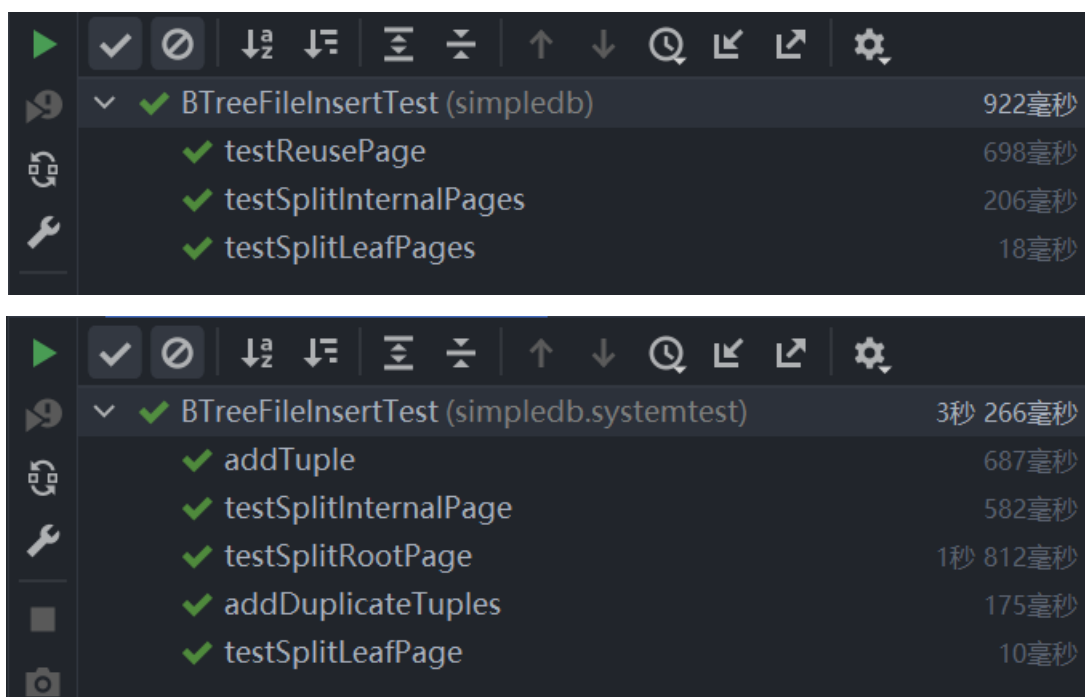
## exercise2

在exercise2中要求在 `BTreeFile.java` 这个类中实现方法 `splitLeafPage()` 和 `splitInternalPage()`。这两个方法的作用是，当插入一个新的 `Field` 的时候，可能要插入的叶节点已经满了，这时候需要将叶节点分裂成两部分，然后将右边叶节点的第一个 `Tuple` 的 `keyField` 作为索引加入到其父节点中，这个过程就需要调用 `splitLeafPage()` 这个方法。而在将新的索引加入到内部父节点的过程中，有可能内部节点也是满的，因此内部节点也要发生分裂，这时候需要调用 `splitInternalPage()` 这个方法。

`splitLeafPage()` 和 `splitInternalPage()` 这两个方法整体的实现思路一致，只不过需要注意的是，在分裂完成后向上一级父节点增加索引的时候，两个方法的实现稍有差异。

这两个方法的实现思路如下，首先要调用 `getEmptyPage()` 这个方法获取到一个全新的页面，需要注意的是，要通过传入合适的参数来确定获取的叶节点页还是内部节点页。然后通过当前的 `page` 的迭代器去遍历 `page` 中的所有内容，当遍历数量超过一半的时候，就需要将剩余的部分从 `page` 中删除，并插入到新的 `page` 中。这里需要注意，要先执行 `deleteTuple()` 方法，再执行 `insertTuple()` 方法，这是因为，在 `deleteTuple()` 方法中会将当前 `Tuple` 的 `RecordId` 置为 `null`，而后在执行 `insertTuple()` 方法的时候会重新设置 `RecordId`，这样才能保证这个 `Tuple` 有对应的 `page` 来存储。同理在实现 `splitInternalPage()` 只需要调用对应的 `deleteEntry()` 和 `insertEntry()` 方法即可。随后要注意更改 `page` 的左右兄弟指针。

在随后向父节点增加索引的过程中需要注意，对于 `splitLeafPage()` 方法，只需要将新 `page` 中的第一个 `Tuple` 的 `keyField` 作为 `Entry` 的 `Field`，然后 `page` 作为 `Entry` 的左子树，`newPage` 作为 `Entry` 的右子树，将这个新生成的 `Entry` 插入到父节点中，并设置 `newpage` 的 `parentId`。而对于 `splitInternalPage()` 方法，则需要将 `newPage` 的第一个 `Entry` 从 `newPage` 中删除，并依次作为 `Entry` 的 `Field`，`page` 作为 `Entry` 的左子树，`newPage` 作为 `Entry` 的右子树，插入到父节点中，并对父节点，`page` 和 `newPage` 都调用 `updateParentPointers()` 方法，更新这些节点中的孩子节点的父指针。



▶	✓	⊘	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	🔍	↶	↷	⚙️	
🔍	✓	✓	BTreeFileInsertTest (simplifiedb)										922毫秒
🔄	✓		testReusePage										698毫秒
🔧	✓		testSplitInternalPages										206毫秒
	✓		testSplitLeafPages										18毫秒

▶	✓	⊘	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	🔍	↶	↷	⚙️	
🔍	✓	✓	BTreeFileInsertTest (simplifiedb.systemtest)										3秒 266毫秒
🔄	✓		addTuple										687毫秒
🔧	✓		testSplitInternalPage										582毫秒
	✓		testSplitRootPage										1秒 812毫秒
■	✓		addDuplicateTuples										175毫秒
📷	✓		testSplitLeafPage										10毫秒

## exercise3

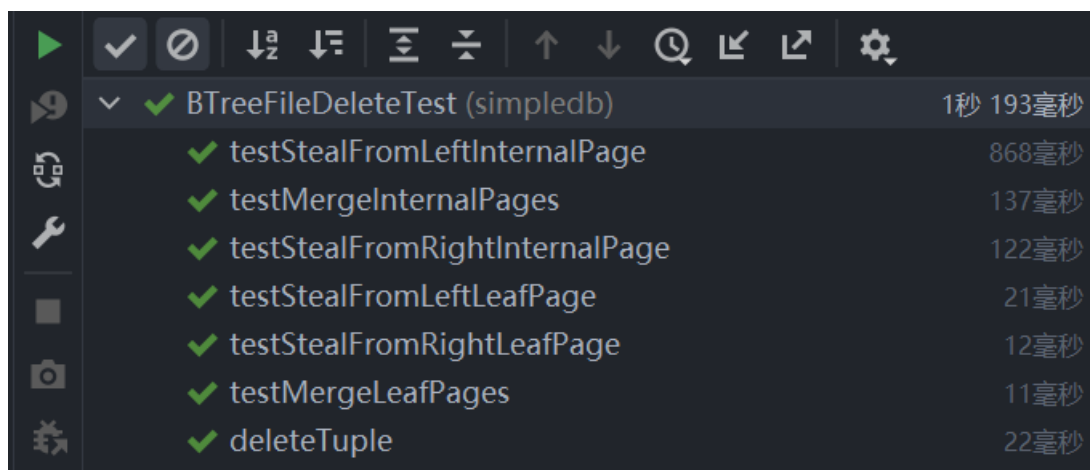
在exercise3中要求在 `BTreeFile.java` 这个类中实现方法 `stealFromLeafPage()` `stealFromLeftInternalPage()` `stealFromRightInternalPage()` `mergeLeafPages()` 和 `mergeInternalPages()`。这几个方法的作用就是，在删除某一个Field的时候，可能会导致当前page中的元素数量少于 $m/2$ ，这个时候就需要对于当前page中的元素进行补充。补充的方式有两种，一种是和左右的page去借元素，而当左右的page中也没有重组的元素的时候，就需要选择一个相邻的page和当前的page进行合并。

对于方法 `stealFromLeafPage()`，首先计算一下两个页面中一共有多少的元素，然后尽可能使得元素在两个页面之间平分。对于从右边的相邻page中借tuple的情况，需要获取到右边page的正向iterator，而对于从左边的相邻page中借tuple的情况，需要获取到左边page的反向iterator。首先将父结点中原来entry中保存值插入到page中，然后将 `iterator.next()` 获取到的元素设置到父结点中对应的entry里，并更新父节点，并从相邻节点的page中删除这个元素。

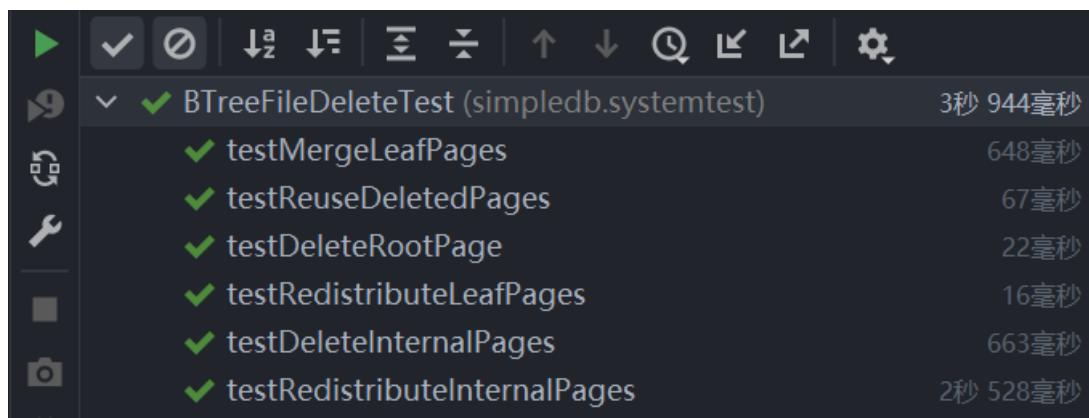
对于方法 `stealFromLeftInternalPage()`，首先计算一下两个页面中一共有多少的元素，然后尽可能使得元素在两个页面之间平分。然后从左边的page中通过反向迭代器获取最后一个Entry，从右边的page中通过正向迭代器获取第一个Entry，然后再左边的page中调用 `deleteKeyAndRightChild()` 方法，删除掉左边的左后一个元素及其右子树。然后利用 `parentEntry.getKey()` `leftEntry.getRightChild()` 和 `rightEntry.getLeftChild()` 构造一个要插入page中的新Entry。并且将parentEntry中的Key设置为leftEntry中的Key，最后对父节点调用 `updateEntry()` 方法。

最后需要注意的是，要对page调用 `updateParentPointers()`，更新其子节点的父指针。

对于方法 `mergeLeafPages()`，要实现将所有右边page的tuple移动到左边的page中。在移动完所有的tuple之后，需要更新page之间的左右指针关系，保证将rightPage删除之后，相邻的指针关系不会丢失。对于 `mergeInternalPages()` 所需要执行的操作也是类似的。



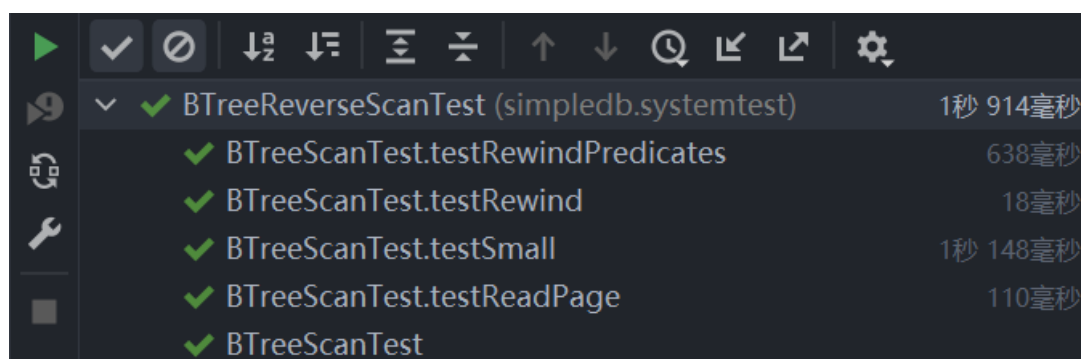
▶	✓	○
↓a	↓b	≡
↑	↓	Q
↶	↷	⚙
🔍	✓	BTreeFileDeleteTest (simplifiedb)
🔧	✓	testStealFromLeftInternalPage
🔧	✓	testMergeInternalPages
🔧	✓	testStealFromRightInternalPage
🔧	✓	testStealFromLeftLeafPage
🔧	✓	testStealFromRightLeafPage
🔧	✓	testMergeLeafPages
🔧	✓	deleteTuple



✓	BTreeFileDeleteTest (simplifiedb.systemtest)	3秒 944毫秒
✓	testMergeLeafPages	648毫秒
✓	testReuseDeletedPages	67毫秒
✓	testDeleteRootPage	22毫秒
✓	testRedistributeLeafPages	16毫秒
✓	testDeleteInternalPages	663毫秒
✓	testRedistributeInternalPages	2秒 528毫秒

## bonus exercise4

在本次的额外实验中，要求实现一个能够反向迭代遍历BTreeFile的方法。可以通过参考 `BTreeScan.java` 来实现 `BTreeReverseScan.java`，其需要做的修改很少，只需要将类中获取迭代器的方法从 `iterator` 改为 `reverseIterator`，即可获得能够反向遍历page中内容的迭代器。此外还需要在 `BTreeFile.java` 中实现 `findReverseLeafPage()`。而这里相对于 `findLeafPage()` 在获取迭代器的时候获取一个反向迭代器，然后在遍历的过程中注意反转判断逻辑即可。



✓	BTreeReverseScanTest (simplifiedb.systemtest)	1秒 914毫秒
✓	BTreeScanTest.testRewindPredicates	638毫秒
✓	BTreeScanTest.testRewind	18毫秒
✓	BTreeScanTest.testSmall	1秒 148毫秒
✓	BTreeScanTest.testReadPage	110毫秒
✓	BTreeScanTest	

## 重难点

本次实验的过程中，我主要体会到了由于项目已经初始化了一些代码，因此在实现功能，调用提供的方法的时候要特别注意方法内部的逻辑，有可能对于方法调用的先后顺序不同造成严重的错误。

例如，在从一个page中删除元素再将这个元素插入到另一个page的过程中，要注意首先需要调用delete方法，因为在这个方法内部会将当前元素的RecordId设置为NULL，并在后续调用insert方法的时候重新设置这个RecordId。如果颠倒顺序，就会导致这个元素和磁盘上文件中的内容丢失映射关系。

再比如对于所有的方法中，有可能需要操作dirtyPages，但是这个并不需要我们手动的去将发生修改的page加入到这个hashMap中，值需要在获取page的时候，通过不同的权限来获取，如果使用READ\_ONLY权限来获取，获取的page就不会被加入到dirtyPages中，如果使用READ\_WRITE权限来获取，就会将page加入到dirtyPages中。

再如在mergeInternalPage的时候，对于获取到遍历左右两侧元素的迭代器，需要在每次循环的内部去获取，因为迭代器不会自己去根据当前page的修改而更新，因此如果不重新获取，调用next方法获取到的元素就不是我们想要的元素。因此要在每次循环内部重新获取这个迭代器，或者调用rewind方法。

