南開大學
Nankai University

计算机学院

SimpleDB 实验报告

Lab3

姓名：朱霞洋

学号：2113301

专业：计算机科学与技术

2023 年 4 月 28 日

摘要

lab3 实现 BTree File 以及一些基本的页面的分裂合并、插入删除的方法，与 B+ 树这种数据结构有关。

代码链接：朱霞洋的 Gitlab 仓库

目录

1 提交历史

2 设计思路

在此前的 File 实现中用了 HeapFile 堆文件来简单顺序存贮数据，然而当数据规模很大时，这样顺序存储和搜索的效率是低下的，因此在此次 lab 中将实验一个基于 B+ 树的 BTreeFile 来存储数据，达到更高的搜索性能。

BTreeFile 中包含一颗 B+ 树，这是一个高度平衡的多路搜索树，所有 Tuples 存放在叶结点上，而非叶结点上存储着索引，通过索引找到相应的叶子节点进行查询。

BTreeFile 中有四种不同的 Page，四种页面使用 PageId 中的 pgcateg 区分，分别是：

1. 根节点页面：一个 B+ 树的根节点，在 SimpleDB 中实现为 BTreeRootPtrPage.java;
2. . 头部节点页面：用于记录整个 B+ 树中的一个页面的使用情况，在 SimpleDB 中实现为 BTree-HeaderPage；
3. 内部节点页面：除去根节点和叶节点外的节点，即 BTreeInternalPage，每个 BTreeInternalPage 由一个一个的 entry 组成，每个 entry 有其索引值 key，以及左右子页面结点；
4. 叶子节点页面：不存放 entry，而是存储 tuples，在 SimpleDB 中实现为 BTreeLeafPage

2.1 Exercise1 Search

Exercise1 中要实现 findLeafPage 的递归函数，这个函数会从当前 PageId 为 pid 的页面结点开始，在 B+ 树中查找可能包含字段 Field f 的叶节点，用只读权限锁定叶节点路径上的所有内部节点，并使用权限 perm 锁定叶节点。如果 f 为 null，它将查找最左边的叶节点。主要实现思路为：

1. 获取数据页类型并判断该数据页是否为叶子节点，如果是则递归结束，将该页面返回，不是的话进行以下步骤；
2. 调用 getPage 函数，通过 pid，以 READ_ONLY 只读权限找到当前页面；
3. 获取当前内部节点的迭代器，对内部节点的 entry 进行迭代，这里要主要 field 是空的处理，如果是空直接继续从迭代器首元素的左页面指针向下查询；
4. . 找到第一个大于（或等于）filed 的 entry，然后递归其左子页面结点，如果到了最后一个页面，则递归其右子页面结点；

该部分较为简单，需要注意和理解的是，参数中的 dirtypages 是一个需要更新的脏页的 list，在之后的函数调用中会不断用到；此外需要注意 pageId 中的 pgcateg() 方法用于获取页面类型，getPage 方法用于通过 pid 获取页面，读取当内部页面结点时要用只读权限，而递归寻找时用 perm 权限。

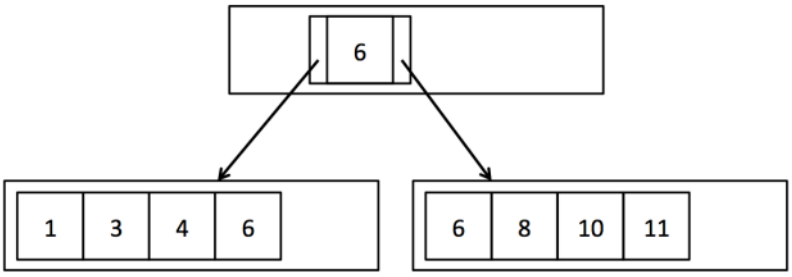

Figure 1: A simple B+ Tree with duplicate keys

图 2.1: 按 key 查找

2.2 Exercise2 Insert

通过 findLeafPage 函数可以找到应该插入 Tuple 的页面，如果页面有空 Slot 则可直接放入，然而如果页面已满，此时要进行分裂叶结点操作：页面中的 Tuples 会平均分开，后半部分放入一个新的叶结点中，同时会在父内部节点中新增一个 Entry，左右子结点指针指向分裂后的叶结点，Entry 中的 key 是第二个页面结点的第一个 Tuple 的 keyField。

而如果 InternalPage 也满了，则也会进行分裂操作，会将当前页面的后一半 Entry 放入新页。而与叶结点分裂不同，叶结点分裂会复制第二个页面的第一个 key 放入 parentPage 中，而内部页面分裂会将最中间 Entry “推” 到 parentPage 中，并将这个 entry 从当前内部页面删去。推动该 Entry 时，要将其左右子结点指针更改为分裂后的两个内部页面结点。

具体实现思路如下：

2.2.1 splitLeafPage

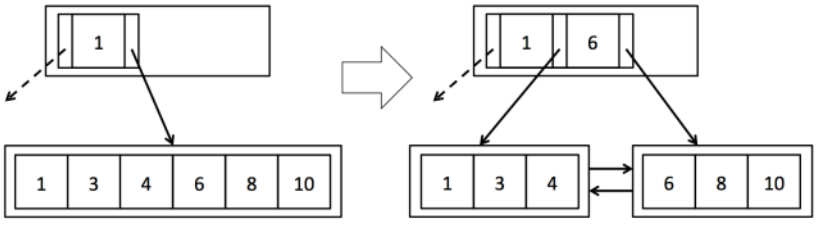

图 2.2: 叶结点分裂

1. 新建一个 leafPage，作为新的页面，将满页面的后半部分 Tuples 复制到新页面，边复制边删除。此处用到 reverseIterator，从当前叶结点尾部开始遍历；
2. 检查之前的满页面是否有右兄弟，如有需要更新兄弟指针，如同在双向链表中插入结点，同时要连接旧页与新页；

3. 找出父节点并创建 entry 进行插入，entry 左右子结点指针指向两个叶结点,entry 的 key 是第二个页面的第一个 Tuple 的 keyField；
4. 更新脏页, 并且调用 updateParentPointer 将两个叶结点的父结点设置为 parent;
5. 根据 field 找出要插入 Tuple 后所在的页面并返回；

2.2.2 splitInternalPage

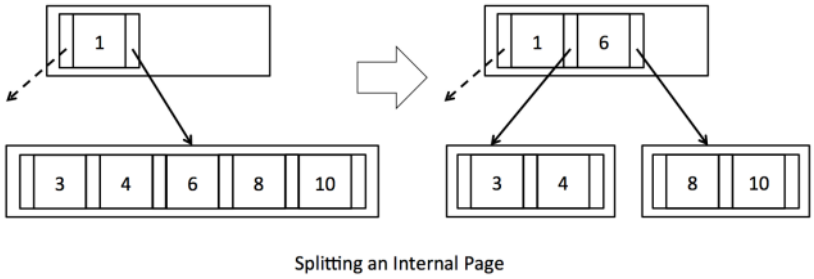

图 2.3: 内部结点分裂

1. 新建一个 internalPage，将满页面的后半部分 entry 复制到新页面，边复制边删除;
2. 将最中间 Entry “推” 到 parentPage 中，并将这个 entry 从当前内部页面删去。推动该 Entry 时，要将其左右子结点指针更改为分裂后的两个内部页面结点;
3. 调用 updateParentPointers，将分裂后的两个内部页面的所有子页面（通过遍历 Entry）的父页面设置为该内部页面；
4. 更新脏页，并根据 field 找出要插入后所在的页面并返回；

2.3 Exercise3 Delete

在删去 Tuples 时,如果兄弟页面比较满,自己却在删除一些 tuple 或者 entry 后比较空,这时可以平衡元素,从兄弟页面取一些元素过来,这样兄弟页面可以不用过早去分裂页面,即 stealFromLeafPage 方法和 stealFromLeft(Right)InternalPage 方法；而如果兄弟页面和删除页面都比较空，元素数小于 m/2，这个时候需要将两个页面合并成一个，避免空间浪费，即 mergeLeafPages 和 mergeInternalPages 方法

具体实现思路为：

2.3.1 stealFromLeafPage

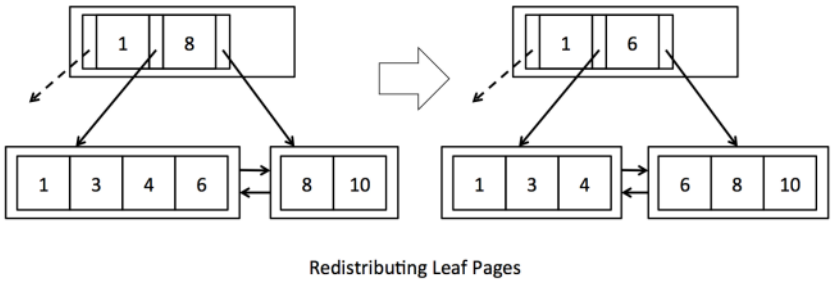

图 2.4: 叶结点 Tuple 转移

1. 通过该页面和兄弟页面的 Tuples 数目计算均值，然后从兄弟页面转移 Tuples 至当前页面;

```
int move = siblingNum - distributeNum;
  while(move > 0){
     Tuple tp = it.next();
     sibling.deleteTuple(tp);
     page.insertTuple(tp);
     move--;
  }              //transport the touples to the page
```

2. 更新父节点中的 entry，要将其 key 值设置为第二个界面的第一个 keyField

```
    Field key =null;
  if(isRightSibling)
     key = it.next().getField(sibling.keyField);     //if it is the right sibling, use its first touple's
         keyFiled to set the entry
  else
     key = page.iterator().next().getField(page.keyField);  //if it is the left sibling , use the page

  entry.setKey(key);
  parent.updateEntry(entry);
```

2.3.2 stealFromLeft(Right)InternalPage

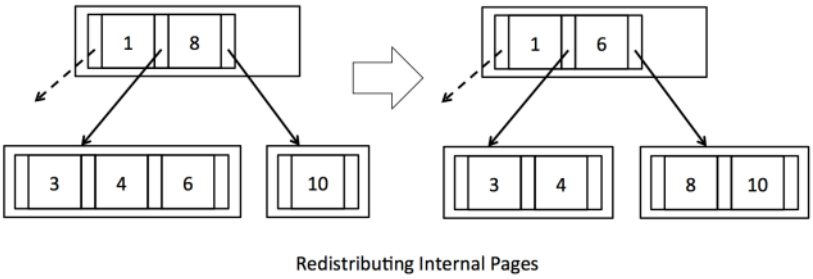

图 2.5: 内部结点 Entry 转移

1. 平均两个内部页面的 Entry 数目，具体操作与叶结点的分配相似，可见仓库; 因为内部节点与其父节点中的 key 值没有重复，迁移 key 的时候也需要将父节点中对应的 Entry 移动到 page 中，并且对其左右子结点指针进行设置；

```
Iterator<BTreeEntry> it = leftSibling.reverseIterator();
BTreeEntry right = page.iterator().next();
BTreeEntry left = it.next();
BTreeEntry entry = new BTreeEntry(parentEntry.getKey(),left.getRightChild(),right.getLeftChild());
page.insertEntry(entry);     //adjust the entry from the parent to the page
```

2. 分配之后，将左侧页面节点中最大的 key “推到” 父节点中，并 page 页面中的 Entry 的父指针，以及设置脏页；

```
dirtypages.put(page.getId(),page);
dirtypages.put(leftSibling.getId(), leftSibling);
dirtypages.put(parent.getId(),parent);        //put the modified pages into dirtypages
updateParentPointers(tid,dirtypages,page);
```

2.3.3 mergeLeagPages

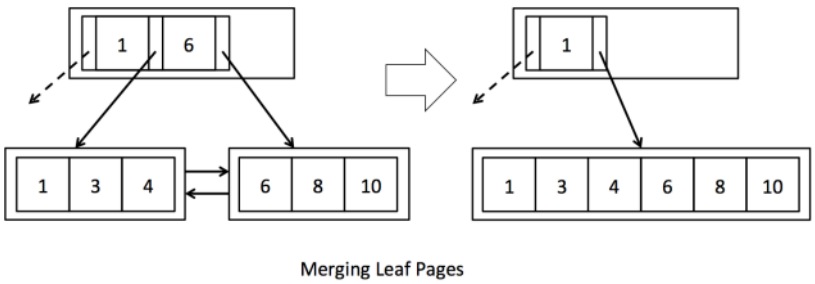

图 2.6: 叶结点合并

1. 将 rightPage 中的所有 tuple 添加到 leftPage 中, 调用 rightPage 中的迭代器即可实现；

2. 判断 rightPage 是否有右兄弟，如果没有 leftPage 的右兄弟为空，如果有 leftPage 的右兄弟指向 rightPage 的右兄弟;
3. 回收页面，将右页设置为空页，调用 setEmptyPage 方法;
4. 调用 deleteParentEntry 方法，从父结点中删除左右孩子指针指向 leftPage 和 rightPage 的 entry;

```
1 deleteParentEntry(tid,dirtypages,leftPage,parent,parentEntry);    // no need for the parent entry
```

5. 设置脏页;

2.3.4 mergeInternalPage

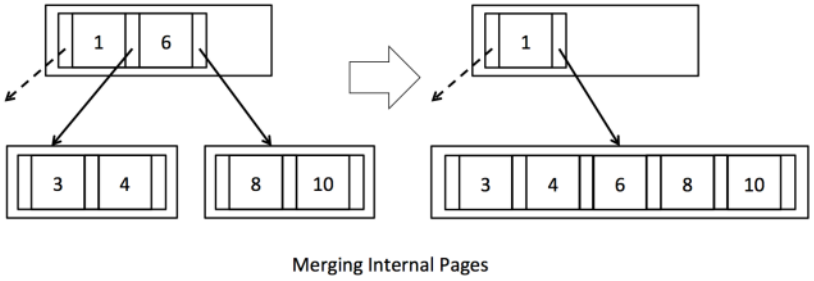

图 2.7: 内部结点合并

1. 先将父结点中的指向 leftPage 和 rightPage 的 entry 添加到 leftPage 中;
2. 将 rightPage 中的 entry 转移到 leftPage 中，调用迭代器实现;
3. 调用 updateParentPointers(tid,dirtypages,leftPage)，将 leftPage 指向的所有子叶的父结点设置为 leftPage;
4. 回收页面，将右页设置为空页，调用 setEmptyPage 方法;
5. 调用 deleteParentEntry 方法，从父结点中删除左右孩子指针指向 leftPage 和 rightPage 的 entry;
6. 设置脏页

3 重难点

3.1 LeafPage 和 InternalPage 的区别

在分裂结点时，叶结点将全部 Tuple 平均分开即可，并将父节点的对应 Entry 的 Key 改为分裂后第二个页面的首个 key，而内部结点需要将最中间的 Entry 推至上一级;

在合并结点时，叶结点简单的全部合并即可，并删去父结点对应的 Entry；而内部节点需要先将上一级中的 Entry 移回 page，然后再进行两页的合并;

元素转移时也类似，内部结点需要多进行一步将上一级的 Entry 移回 Page，再进行合并，最后重新设置 parentEntry 的 Key;

在细节操作上二者有很大差别，例如还有叶结点的兄弟指针设置等等，要谨慎对待以免出错;

3.2 一些方法的调用和理解

此次实现 lab3 调用了一些已有的方法，对于这些方法的作用的理解也是重点之一。

1. updateParentPointer 与 updateParentPointers：前者用于设置单独一页的父指针，而后者则是遍历某一内部节点的所有 Entry，对每一个子页面调用 updateParentPointer 方法，从而达到将所有子节点的父指针全部设置为当前页面的目的，在更改页面元素时需要用到；

2. getParentWithEmptySlots: 用以找到一个可以接收新 Entries 的父结点，在这个方法中有可能会创建新的根结点，也有可能调用 splitInternalPage 方法分裂内部节点从而留出空位；

3. updateEntry: 检查要更新的 Entry 的 Key 值是否合法，并根据 Entry 的 key 和子节点指针，设置子页面数组 children[] 和 key 数组 keys[];

4. deleteParentEntry：会删除一个指定的 Entry 和其右子结点指针，这个方法还会处理当删除了根结点的 Entry 时的情况以及删除 Entry 后 Entry 数小于最低界限的情况，此时会进行页面的合并或是新建根页面。

4 改动部分

此次主要是对 BufferPool 进行了一些改动，由于此前使用的是 HeapFile，因此在 BufferPool 的 getPage 函数以及 DeleteTuple 和 InsertTuple 等函数中，将 File 类型统一改为 DbFile，内存页的类型也统一改为 Page，否则会出现 BTreeLeafPage 或 BTreeInternalPage 无法转换成 HeapPage 的错误。

```
1   DbFile table = Database.getCatalog().getDatabaseFile(tableId); //modified in lab5, use DbFile instead of
        HeapFile
2       ArrayList<Page> affectedPages = table.insertTuple(tid,t);
3       for(Page page : affectedPages){
4           page.markDirty(true, tid);
5           if (idToPage.size() == numPages) {
6               evictPage();            //when bufferpool is full, evict page
7           }
8           idToPage.put(page.getId(), page);
9       }
10      ...
```

5 测试结果

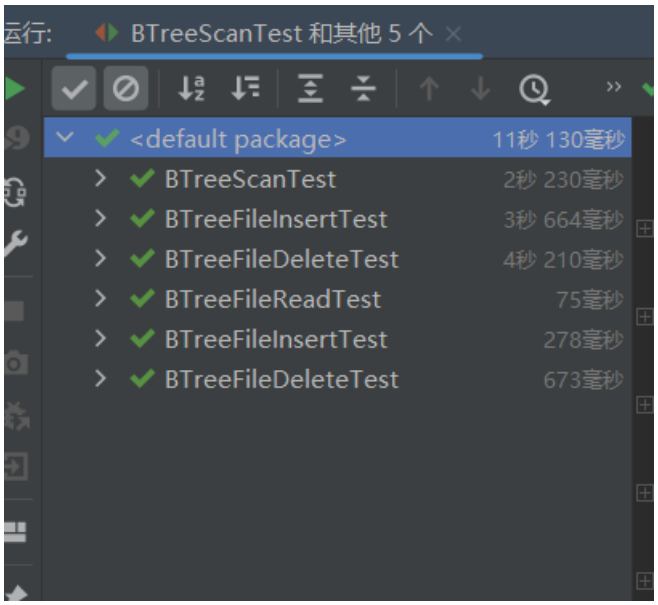

图 5.8: 测试结果

6 附加题

附加题要求实现一个 BTreeFile 的 ReverseIterator 迭代器以及一个 BTreeReverseScan 的类，反向迭代 BTreeFile 中的 Tuples，并且可能会有一个 IndexPredicate，在 IndexPredicate 中会有一个 Field 类型的 fieldvalue 和一个运算符 op，当提供 IndexPredicate 时，会返回所有与 IndexPredicate 中的 fieldvalue 满足 op 关系的 Tuple 的反向迭代器（如小于 fieldvalue 或大于等于 fieldvalue 等）。

反向迭代时,需要从最右侧页面开始,为了不影响基础的实现方法,我新建了一个 findMostRightLeafPage 方法，用于找寻最右侧的叶结点：

```
1  BTreeLeafPage findMostRightLeafPage(TransactionId tid, HashMap<PageId, Page> dirtypages,
       BTreePageId pid, Permissions perm)
2      throws DbException, TransactionAbortedException {
3    int pageType = pid.pgcateg(); //get the type of the page(internal/leaf ...)
4    if(pageType == BTreePageId.LEAF) //once we found the leaf page, return it
5      return (BTreeLeafPage) getPage(tid,dirtypages,pid,perm);
6    BTreeInternalPage iPage = (BTreeInternalPage) getPage(tid,dirtypages,pid,Permissions.READ_ONLY);
7
8    Iterator<BTreeEntry> it = iPage.iterator(); //get the entries
9    BTreeEntry entry = null;
10   while(it.hasNext())
11     entry = it.next();     //find the most right entry
12   return findMostRightLeafPage(tid,dirtypages,entry.getRightChild(),perm); //continue finding on the right
13 }
```

此后实现一个 BTreeFileReverseIterator，用于返回所有 Tuples 的反向迭代器，较为重要的 open() 和 readnext():

```
...
    public void open() throws DbException, TransactionAbortedException {
      BTreeRootPtrPage rootPtr = (BTreeRootPtrPage) Database.getBufferPool().getPage(
            tid, BTreeRootPtrPage.getId(f.getId()), Permissions.READ_ONLY);
      BTreePageId root = rootPtr.getRootId();
      curp = f.findMostRightLeafPage(tid, new HashMap<PageId, Page>(), root, Permissions.READ_ONLY);
      it = curp.reverseIterator();
    }

 protected Tuple readNext() throws TransactionAbortedException, DbException {
      if (it != null && !it.hasNext())
         it = null;

      while (it == null && curp != null) {
         BTreePageId nextp = curp.getLeftSiblingId();
         if(nextp == null) {
            curp = null;
         }
         else {
            curp = (BTreeLeafPage) Database.getBufferPool().getPage(tid,
                  nextp, Permissions.READ_ONLY);
            it = curp.reverseIterator();
            if (!it.hasNext())
               it = null;
         }
      }

      if (it == null)
         return null;
      return it.next();
    }
```

然而以上没有考虑到有 IndexPredicate 限制的时候,因此还需要创建一个 BTreeReverseSearchIterator,用于按照 IndexPredicate 的条件来进行搜索并返回迭代器,同时还实现了一个 reverseIndexIterator,用以调用 BTreeReverseSearchIterator:

```
    public DbFileIterator reverseIndexIterator(TransactionId tid, IndexPredicate ipred) {
      return new BTreeReverseSearchIterator(this, tid, ipred);
    }
```

而在 BTreeReverseSearchIterator 中重要的 open() 和 readnext() 方法实现如下:

```
    public void open() throws DbException, TransactionAbortedException {
      BTreeRootPtrPage rootPtr = (BTreeRootPtrPage) Database.getBufferPool().getPage(
            tid, BTreeRootPtrPage.getId(f.getId()), Permissions.READ_ONLY);
      BTreePageId root = rootPtr.getRootId();
      if(ipred.getOp() == Op.EQUALS || ipred.getOp() == Op.LESS_THAN
            || ipred.getOp() == Op.LESS_THAN_OR_EQ) {
         curp = f.findLeafPage(tid, root, Permissions.READ_ONLY, ipred.getField());
```

```
        }
        else {
            curp = f.findMostRightLeafPage(tid, new HashMap<PageId, Page>(), root,
                Permissions.READ_ONLY);
        }
        it = curp.reverseIterator();
    }

protected Tuple readNext() throws TransactionAbortedException, DbException,
        NoSuchElementException {
        while (it != null) {

            while (it.hasNext()) {
                Tuple t = it.next();
                if (t.getField(f.keyField()).compare(ipred.getOp(), ipred.getField())) {
                    return t;
                }
                else if(ipred.getOp() == Op.GREATER_THAN || ipred.getOp() ==
                    Op.GREATER_THAN_OR_EQ) {
                    // if the predicate was not satisfied and the operation is greater than, we have
                    // hit the end, be careful we are now talking about a reverse iterator
                    return null;
                }
                else if(ipred.getOp() == Op.EQUALS &&
                    t.getField(f.keyField()).compare(Op.LESS_THAN, ipred.getField())) {
                    // if the tuple is now less than the field passed in and the operation
                    // is equals, we have reached the end
                    return null;
                }
            }

            BTreePageId nextp = curp.getLeftSiblingId();
            // if there are no more pages to the left, end the iteration
            if(nextp == null) {
                return null;
            }
            else {
                curp = (BTreeLeafPage) Database.getBufferPool().getPage(tid,
                    nextp, Permissions.READ_ONLY);
                it = curp.reverseIterator();
            }
        }

        return null;
    }
```

而在完成以上方法后，我新写了一个 BTreeReverseScanTest 对这些方法进行了检验。该类的大部分代码与 BTreeScanTest 相似，但是在几个关键部位有不同，特别是排序 Tuples 时用到的 TupleComparator

类，由于要检测反向迭代器，那么排序时也应当对 Tuples 反向排序，因此改为：

```
    public int compare(ArrayList<Integer> t1, ArrayList<Integer> t2) {
        int cmp = 0;
        if(t1.get(keyField) > t2.get(keyField)) {
            cmp = −1;
        }
        else if(t1.get(keyField) < t2.get(keyField)) {
            cmp = 1;
        }
        return cmp;
    }
```

具体代码可见朱霞洋的 Gitlab 仓库中的 BTreeReverseScanTest.java 以及 BTreeFile.java 中的 BTreeReverseSearchIterator 和 BTreeReverseItrator。

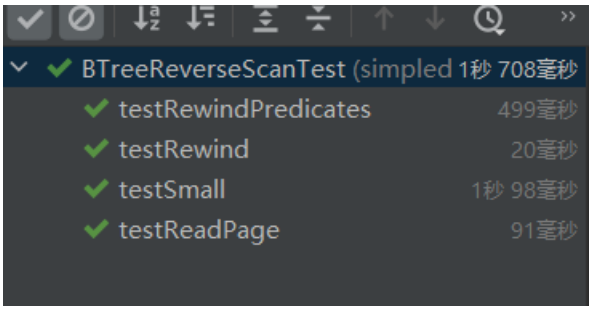

图 6.9: 附加题检测通过