哈尔滨工业大学

**<<数据库系统>>**

**实验报告**

**(2020年度春季学期)**

|  |  |
| --- | --- |
| **姓名：** | **吴昊** |
| **学号：** | **1170300527** |
| **学院：** | **计算机** |
| **教师：** | **史建焘** |

目录

[实验二 数据库索引及查询算法实现 3](#_Toc39517208)

[一、 实验目的 3](#_Toc39517209)

[二、 实验环境 3](#_Toc39517210)

[三、 实验过程 3](#_Toc39517211)

[1. 数据操纵 3](#_Toc39517212)

[2. B树 4](#_Toc39517213)

[3. 多路归并排序 6](#_Toc39517214)

[四、 实验结果 8](#_Toc39517215)

[1. 数据操纵 8](#_Toc39517216)

[2. B树 10](#_Toc39517217)

[3. 多路归并排序 12](#_Toc39517218)

[五、 效率分析 12](#_Toc39517219)

[六、 实验心得 13](#_Toc39517220)

实验二 数据库索引及查询算法实现

## 实验目的

掌握B树索引查找算法，多路归并排序算法，并用高级语言实现

## 实验环境

* 系统：Windows 10
* 开发语言：JAVA
* IDE：Intellij IDEA

## 实验过程

### 数据操纵

1. 数据结构：结构由id和info构成，这里使用了泛型并实现了Comparable接口，方便在排序比较时使用，可以根据实际情况选择实际类型，由于要求根据属性A进行排序，所以要求其实现了Comparable接口，

public class Record<K extends Comparable<K>, V>

implements Comparable<Record<K, V>> {

private K id;

private V info;

}

1. 数据生成：这里先使用了set存储生成的所有关键字，保证其不重复，接下来在生成生成的字符串均为ThisIs[记录编号]，这样在查找时可以直观的看到结果是否正确

Set<Integer> ids = new HashSet<>();

do {  
 ids.add(rand.nextInt());  
} while (ids.size() < number);  
int i = 0;  
for (Integer id : ids) {  
 Record<Integer, String> record = new Record<>();  
 record.setId(id);  
 record.setInfo("ThisIs" + String.*format*("%06d", i) + "");  
 records.add(record);  
 i++;  
}

1. 文件读写：

记录和索引的写比较类似，写记录时只需要将记录中的int转化位byte数组，string通过getBytes方法直接获取byte数组，然后通过ouputStream写入文件即可

byte[] bytes = *int2Bytes*(record.getId());

outputStream.write(bytes);  
outputStream.write(record.getInfo().getBytes());

读取文件比较简单，每次读取记录长度的字节数，然后将其按字节还原回属性AB即可：

byte[] bytes = new byte[16];

while (inputStream.read(bytes) != -1) {  
 records.add(*bytes2Record*(bytes));  
}

在file中start（已读的记录条数所有skip时\*16）开始读取最多number条记录，用于多路归并排序

byte[] bytes = new byte[16];

if (inputStream.skip(start \* 16) != start \* 16)  
 return records;  
while (inputStream.read(bytes) != -1) {  
 records.add(*bytes2Record*(bytes));  
 all++;  
 if (all >= number)  
 break;  
}

还实现了其他的辅助方法，int与byte数组的互转，byte转Record等。其他方法如通过文件获取索引页类似，只需要获取当前文件的偏移量然后构造record，由于使用了泛型，实际没什么区别这里不再说明了

1. 通过记录偏移量查找：通过偏移量读取文件内容：index即为偏移量，通过skip直接跳到记录位置，然后读取记录

byte[] bytes = new byte[16];

if (inputStream.skip(index) == index && inputStream.read(bytes) != -1) {  
 return *bytes2Record*(bytes);  
}

### B树

B树应该是整个实验最难的地方了，也是写实验时花费时间最久的部分，不过写完之后收获也是最大的。这树建的真是太好了，这里算法完全按照算法导论第18章B树完成。

1. B树结构：T为每个节点最小子节点个数(根节点除外>=2)，root为根节点

private static final int *T* = 4;

private BTreeNode root;

1. 内部节点：btree的每个节点：keys为节点的关键字，children为子节点，leaf为该节点是否是叶子节点

private List<Record<K, V>> keys;

private List<BTreeNode> children;  
private boolean leaf;

1. 查询：从根节点开始找，如果根节点里有，则返回；否则找到对应的下标去子节点递归搜索，如果到了叶子节点还没找到，那就找不到。

int i = 0;

while (i < node.size() && key.compareTo(node.getKeys().get(i).getId()) > 0)  
 i++;  
if (i < node.size() && key.compareTo(node.getKeys().get(i).getId()) == 0)  
 return node.getKeys().get(i).getInfo();  
else if (node.isLeaf())  
 return null;  
else return search(node.getChildren().get(i), key);

1. 添加：添加算法比普通二叉树要复杂不少，当一个节点满时要进行分离，同样使用递归，要注意这里并不是在添加之后在调整，而是在从上向下递归的过程中如果遇到已满的节点就对其分裂，这样可以保证子节点分裂时父节点一定不是满的。

这里用到了三个函数，insert，splitNode，insertNotFull

splitNode：分裂节点这里较为简单不放代码了，输入为父节点和要分裂的子节点的下标，将子节点从中分裂成两份，中间节点提出到父节点，两侧作为两个新的子节点，由于从上至下递归，所以父节点一定不满，可以添加新分裂出的节点

insert：首先判断是否为根节点，如果是根节点要对其分裂并设置新节点为跟然后调用insertNotFull

insertNotFull：较为核心的方法，如果是根节点，那么很好直接插入，这里不用担心叶子节点是否已满，在寻找到该节点时如果已满就已经分裂。如果是内部节点，则需要判断其是否已满，如果已满则对其分裂，并对与新关键字比较之后递归的插入新关键字

int i = node.getKeys().size();

if (node.isLeaf()) {  
 while (i > 0 && key.getId().compareTo(node.getKeys().get(i-1).getId()) < 0)  
 i--;  
 node.getKeys().add(i, key);  
} else {  
 while (i > 0 && key.getId().compareTo(node.getKeys().get(i-1).getId()) < 0)  
 i--;  
 if (node.getChildren().get(i).size() == 2 \* *T* - 1) {  
 splitChild(node, i);  
 if (key.getId().compareTo(node.getKeys().get(i).getId()) > 0) {  
 i++;  
 }  
 }  
 insertNotFull(node.getChildren().get(i), key);  
}

1. 删除：删除应该是b树中最复杂的算法，由于删除后要对树进行新的调整使其一直满足b树的限定条件，分的情况也是最多的，都要分别处理所以比较麻烦，这里要保证删除后满足节点最少包含t个关键字而不是t-1个，分如下三种情况，代码较长不贴了。
2. 首先：如果是叶子节点，则直接删除
3. 如果查找到是内部节点：
4. 前子节点有至少包含t个关键字，就在他的前子节点中找到他的前驱节点，并替换他，然后在前子节点中递归删除前驱节点，注意这里不是简单的前子节点的最后一个孩子，而是要递归的找前驱节点
5. 如果前子节点只有t-1个，后子节点至少含有t个，与a类似，用后继节点替换，并递归的删除后继节点。
6. 前后都只有t-1个，那么就将该节点与前后子节点合并成一个新的子节点
7. 如果未查找到，并且当前节点只有t-1个关键字就需要调整到最少t个之后再递归的删除：
8. 如果前或后子节点包含t个节点，那么就向其借来一个节点作为新的关键字，并将关键字降到要删除的子节点中。
9. 如果兄弟节点都只有t-1个，那么就和其中一个合并
10. 输出：简单的使用队列进行遍历，从root开始遍历，首先加入root节点，然后再加入null做一个分层标记，每次读到null说明到达下一层，把它再加入到队尾再重新poll一个

Queue<BTreeNode> queue = new LinkedList<>();

queue.offer(root);  
queue.offer(null); //分层标记

### 多路归并排序

1. 生成临时有序文件：number为每个有序文件里面最大记录条数，生成的文件存在tmp文件夹下的datai文件。首先读取记录存放在临时的list中，当存放记录条数达到最大值时将其排序然后写入新的文件，如果有剩余写入新的文件，核心代码如下，

while (inputStream.read(bytes) != -1) {

Record<Integer, String> record = FileRecord.*bytes2Record*(bytes);  
 records.add(record);  
 if (records.size() % number == 0) {  
 records.sort(Comparator.*comparingInt*(Record::getId));  
 FileRecord.*writeRecord*("tmp/data" + i, records);  
 records.clear();  
 i++;  
 }  
}  
if (records.size() > 0) {  
 records.sort(Comparator.*comparingInt*(Record::getId));  
 FileRecord.*writeRecord*("tmp/data" + i, records);  
}

1. 败者树：这里使用败者树降低归并时的时间复杂度，并实现了泛型其中tree为树的内部节点，size为树的叶子节点数目，最小值为树中可能出现的最小值，设置为int的最小值，初始化时内部节点全部初始化为最小值

private Integer[] tree;

private int size;  
private final ArrayList<T> leaves;  
private static final Integer *MIN\_KEY* = Integer.*MIN\_VALUE*;

其中最核心的是adjust算法，其中要注意的是(s+size)/2是叶子节点的父节点，具体下面有注释

int t = (s + size) / 2;

while (t > 0) {  
 if (s >= 0 && (tree[t].equals(*MIN\_KEY*) || leaves.get(s).compareTo(leaves.get(tree[t])) > 0)) {  
 // 当前节点比它的父节点大，则把当前节点记录为败者，  
 //并把父节点作为胜者进入下一轮比较  
 int temp = s;  
 s = tree[t];  
 tree[t] = temp;  
 }  
 // tree[Integer/2] 是 tree[Integer] 的父节点  
 t /= 2;  
}  
tree[0] = s;

1. 归并排序：参数为临时文件夹路径，每次在每个文件中读取记录数，存放结果的文件名

mergeSort(String path, int number, String resultFilename)

files为该文件夹下所有文件，start记录下次在每个文件中读取记录的位置，单位是条数

File[] files = rootFile.listFiles();

int[] start = new int[files.length];

然后再每个文件中读取number条记录，算法再数据操纵中的文件读写部分已说明，并将其转化为queue，，这里主要是数据结构的转化就不贴代码了。

下面是归并的核心部分：首先初始化败者树，并获取当前最小节点的下标，根据下标可以获得当前最小值。并在该下标的队列中重新poll一个补充该叶子节点的元素，如果返回的是null，说明该队列已经没有元素，就要重新读取，这里通过start记录了每个文件中已经读取的记录数，然后从start处记录读取即可，并更新start[i]如果返回的是空，说明该文件已经被读完那么就要删除该叶子节点，同时还要删除该队列，否则后面排序时队列的下标和败者树的下标将不对应。每排序号一条记录就将其写入结果文件

//初始化败者树

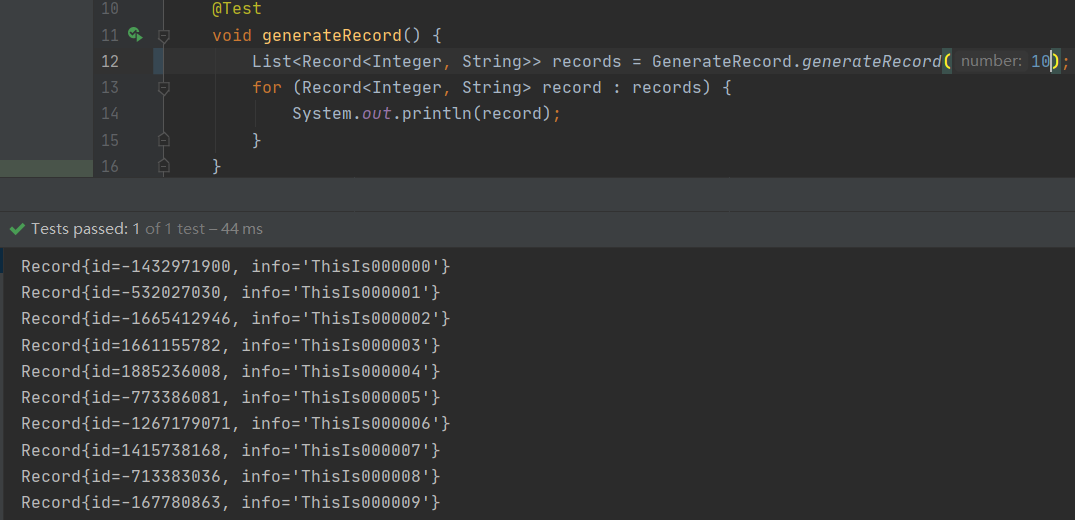
LoserTree<Record<Integer, String>> loserTree = new LoserTree<>(initValues);

//输出胜者  
Integer s = loserTree.getWinner();  
try (OutputStream outputStream = new FileOutputStream(resultFilename)) {  
 Record<Integer, String> record = loserTree.getLeaf(s);  
 byte[] bytes = FileRecord.*int2Bytes*(record.getId());  
 outputStream.write(bytes);  
 outputStream.write(record.getInfo().getBytes());  
 while (true) {  
 //新增叶子节点  
 Record<Integer, String> newLeaf = sources.get(s).poll();  
 if (newLeaf == null) {  
 List<Record<Integer, String>> recordList = FileRecord.*readRecord*(files[s], start[s], number);  
 if (recordList.isEmpty()) { //返回空说明已经读完该文件，删除叶子  
 loserTree.del(s);  
 sources.remove((int) s);  
 }  
 else { //未读完将新的数据加入到待排序序列中  
 start[s] += recordList.size();  
 sources.set(s, new LinkedList<>(recordList));  
 newLeaf = sources.get(s).poll();  
 loserTree.add(newLeaf, s);  
 }  
 } else {  
 loserTree.add(newLeaf, s);  
 }  
  
 s = loserTree.getWinner();  
 if (s == null) {  
 break;  
 }  
 //输出胜者  
 record = loserTree.getLeaf(s);  
 bytes = FileRecord.*int2Bytes*(record.getId());  
 outputStream.write(bytes);  
 outputStream.write(record.getInfo().getBytes());  
 }  
} catch (IOException e) {  
 e.printStackTrace();  
}

## 实验结果

### 数据操纵

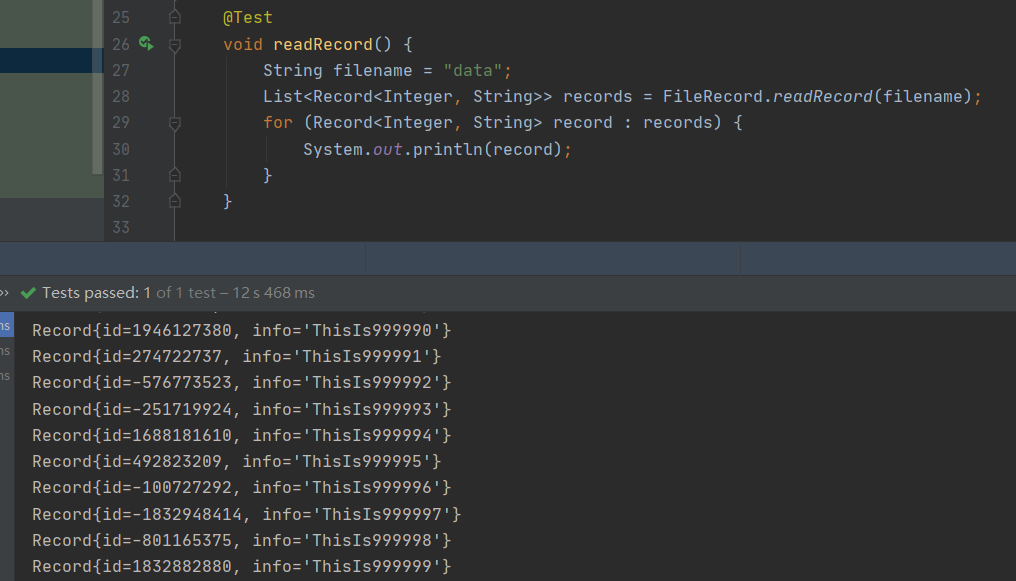
1. 生成记录：十条为例



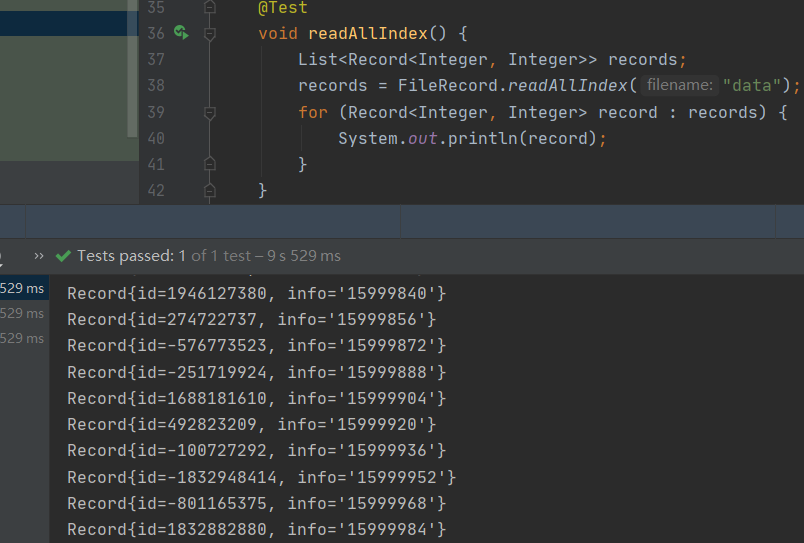
1. 生成文本记录：每条记录16字节，1000000条记录共15.2MB



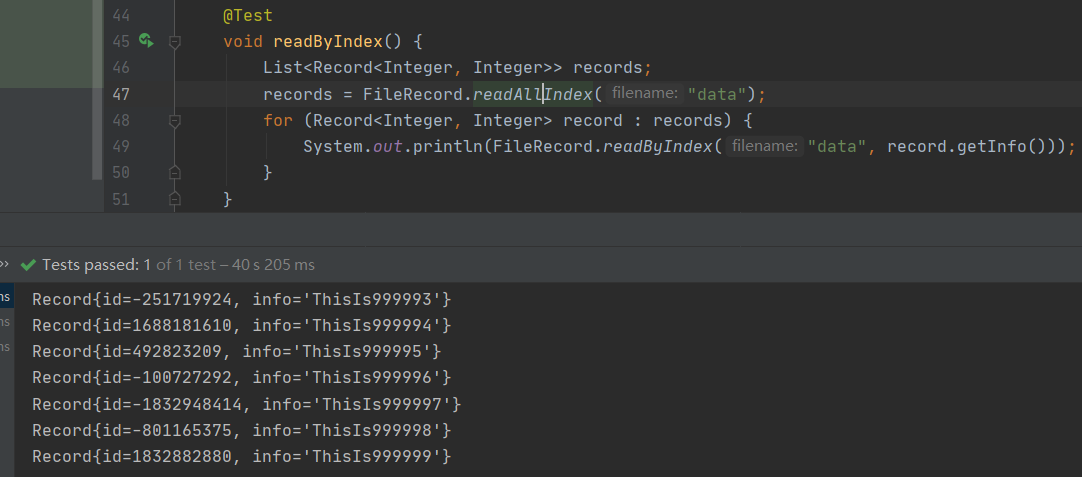
1. 数据读取：直接读取data文件



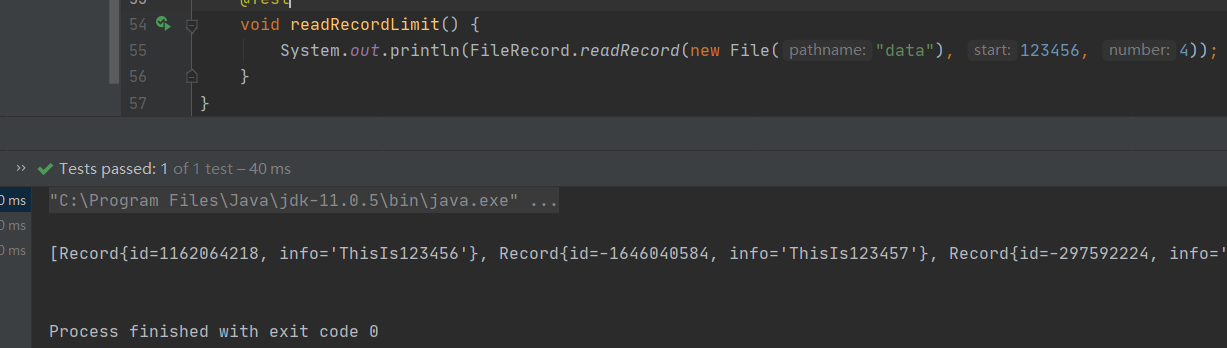
1. 索引读取：索引值为文件中的偏移量，所以都是16的倍数



1. 通过索引读取info：先读取索引，再通过索引读取距离内容

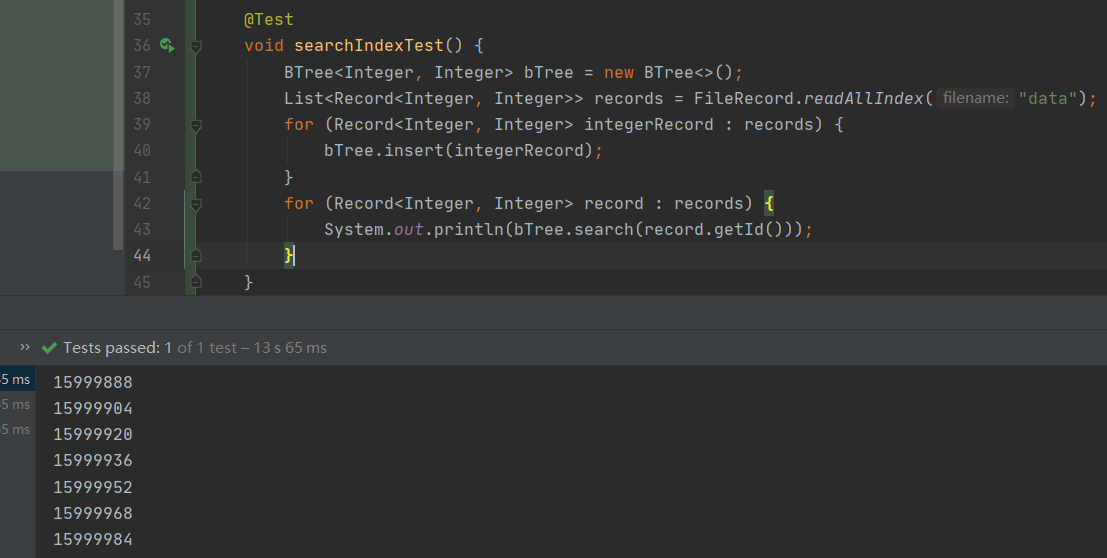


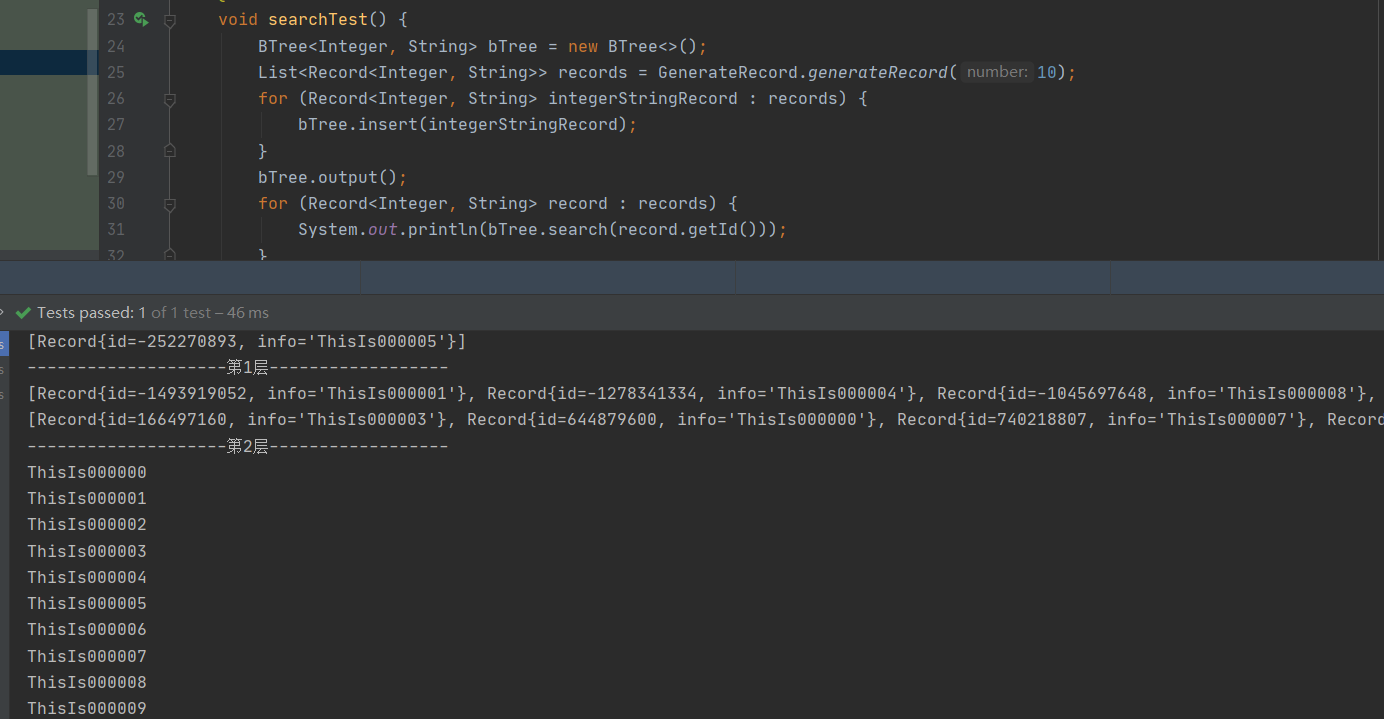
1. 从指定位置读取指定条数记录：从123456\*16的偏移量开始读取4条记录



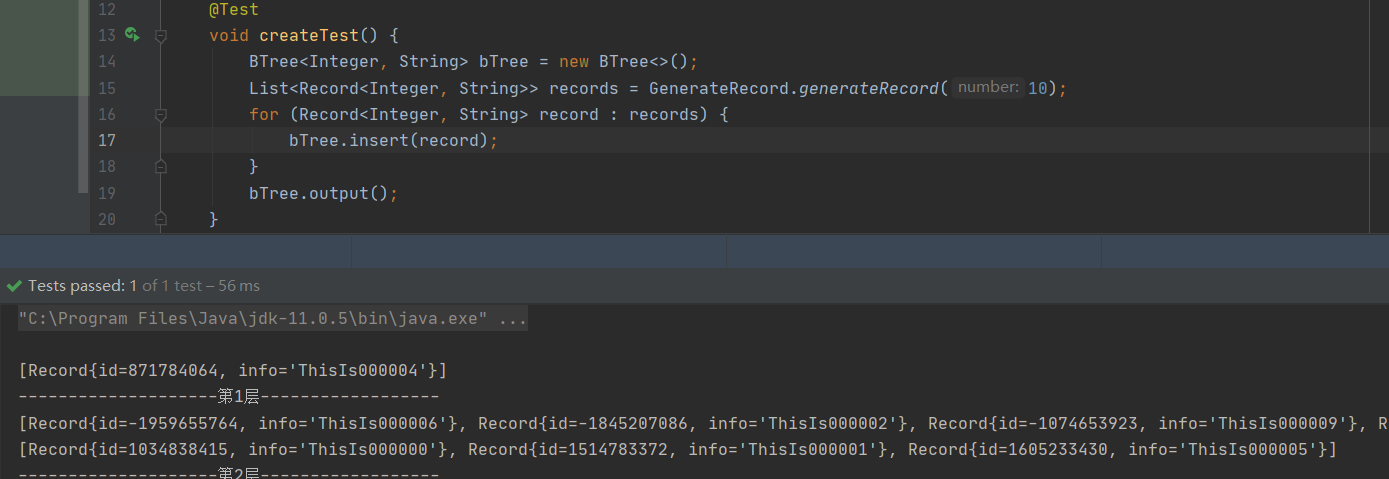
### B树

1. 查询：下面分别是按索引建树和直接按记录建树，由于使用泛型，两个没什么区别，仅用于测试的话直接用记录更直观





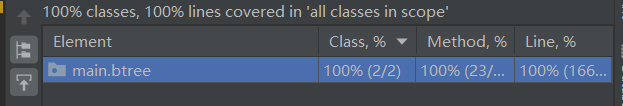
1. 添加：十条为例



1. 删除：对1000000条记录建树，并对每条记录删除一次，如果结果为空基本可以说明所有情况都遇到了且删除成功



1. 最后放一下代码覆盖率，说明bTree所有方法和情况都跑到了



### 多路归并排序

1. 生成临时有序文件：共16个文件，平均分配1000000条记录



1. 多路归并排序：采用16路归并，每次每个记录读取4096条记录。将原记录使用java自带的sort排序，与通过多路归并排序的结果进行比较



## 效率分析

1. bTree效率：

如果n>=1,那么对任意一棵包含n个关键字，高度为h、最小度数t》=2的B树T有

h≤logt(n+1)/2

B树算法的运行时间主要由他所执行的IO次数所决定，所以尽量减少IO的操作次数。因此，一个B树节点通常和一个完整的磁盘也一样大，并且磁盘页的大小限制了一个b树节点可以还有的孩子的个数

B树使我们能实现记录的查找、插入和删除，而每个文件操作只需很少的磁盘I / O。首先我们注意到，如果每个块容纳的键数n相当大，那么，分裂或合并块的情况将会很少。此外，这种操作必需时，绝大多数时候都被局限在叶结点，因此只有两个叶结点和它们的父结点受到影响。所以，我们基本上可以忽略B树重组的I / O开销。然而，每次按给定查找键值查找记录都需要我们从根结点一直访问到叶结点以找到指向记录的指针。因为我们只读B树的块，所以磁盘I / O数将是B树的层数加上一次（对查找而言）或两次（对插入或删除而言）处理记录本身的磁盘I / O

1. 归并排序效率：
   1. 生成临时有序文件：在将源文件转化为临时的有序文件时使用了Collections.sort， 这是个稳定、 时间复杂度是O(nlogn) ， 空间复杂度是O(n)
   2. 归并：使用了败者树降低复杂度，败者树可以在O(logk)的复杂度下得到最小的数，算法复杂度将为O((n-1)\*logk)， 对于外部排序这种数据量超大的排序来说，这是一个不小的提高，其中k为归并的路数，n为记录条数

## 实验心得

通过本次实验，对数据库索引的实现和多路归并排序有了新的认识，实际用的时间比预计多了不少，b算法实现是真的很优秀，看了算法导论真的很是佩服啊，果然大神搞算法，各种情况分析分类的非常清楚，通过较少的限定条件即可保证b树的成立。实现过程完全参照算法导论，做了些对java的适应。多路归并先对源文件排序成多个有序的临时文件并使用败者树完成多路归并排序。

B树感觉已经看明白了实现也都是按照书上实现的，可实际运行起来却会有很多bug，虽然耗费了很长时间，但都解决后还是觉得收获是巨大的，对b树有了非常深刻的认识。

本次实验还出乎意料的给方法都写了注释，使代码看起来更整洁

刚做的时候直接用数据建立的b树，后来发现应该改成索引，于是将其实现了泛型使得b树和败者树适配性更高了。记录一下所有commit历史

