

**系统能力综合培养实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学 院： | 计算机学院 | | |
| 专 业： | 计算机科学与技术 | | |
| 班 级： |  | | |
| 学 号： |  |  |  |
| 姓 名： |  |  |  |
| 指导教师： |  | | |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2020年 1月 6日

目 录

[1 实验任务概述 1](#_Toc29308057)

[2 索引管理模块 3](#_Toc29308058)

[2.1 模块概述 3](#_Toc29308059)

[2.2 模块设计 3](#_Toc29308060)

[2.3 模块实现 5](#_Toc29308061)

[2.4 实验过程 17](#_Toc29308062)

[3 实验总结 27](#_Toc29308063)

[3.1 工作总结 27](#_Toc29308064)

[3.2 改进方案 27](#_Toc29308065)

# 实验任务概述

基于HustBase系统的总体设计架构，根据预先给定的系统框架、部分模块和接口要求，设计并实现系统中的其余模块功能，完成一个具有基本数据定义、数据操纵和数据查询功能的单用户关系数据库管理系统。

HustBase是一个用于实践教学的简化版数据库管理系统，其设计目标是支持简单的SQL语句，提供基本的数据定义、数据操纵、数据查询和查询优化功能。为了便于学生在为期四周的实验周期内掌握系统框架并完成实验内容，与实际的商用DBMS相比，HustBase做了大量的功能简化，舍弃了如多用户、事务、故障恢复等常见特性，支持的SQL语法也非常有限。

HustBase的体系架构如图3.1所示。系统由页面管理、记录管理、索引管理、系统管理、查询处理、语法分析和用户界面等模块构成。各模块功能如下：

* 1. 页面管理：数据库中所有的数据、元数据和索引数据均以文件的形式存储在磁盘上。Windows操作系统支持的文件结构为流文件，而数据库中的数据是以记录为单位来进行存取的，为了实现对磁盘文件的高效I/O，需要将流文件转换为记录文件。这个过程分为两步：首先，将连续的流文件划分为由若干个固定大小的页面，将其抽象为分页文件；其次，将分页文件的页面划分为若干个固定大小的记录插槽或索引项插槽，将其抽象为记录文件或索引文件。本模块提供第一步所需功能，即将流文件抽象为分页文件，并提供以页面为单位的文件读写接口，具体包括：创建、销毁、打开和关闭分页文件；遍历指定文件中的所有页面；从指定文件中读取一个特定页面；在指定文件中添加、删除及修改页面等。
  2. 记录管理：系统表和数据表中的数据均以记录为单位进行存取。本模块在页面管理模块的基础上，将文件中每个页面又划分为若干个固定大小的记录插槽，实现记录的插入、删除、修改和查找，为上层的数据操纵及数据查询功能提供支持。具体包括：创建、销毁、打开和关闭记录文件；插入、删除、修改记录；查找符合指定条件的记录等。
  3. 索引管理：利用索引，系统可以为查询提供快速访问路径。本模块在页面管理模块的基础上，将文件中每个页面又划分为若干个固定大小的索引项插槽，提供对索引项的管理。索引文件与记录文件的不同之处在于，索引采用B+树结构来组织索引项，以便提高关键字查找的效率。具体包括：创建、销毁、打开和关闭索引文件；索引项的插入、删除、修改以及查找等。



图1.1 HustBase系统体系架构图

* 1. 系统管理：本模块提供对数据定义（CREATE/DROP TABLE，CREATE/DROP INDEX）和数据操纵（INSERT、DELETE、UPDATE）功能的支持。系统管理模块的实现依赖于记录管理模块和索引管理模块。
  2. 查询处理：本模块用于提供对数据查询（SELECT）功能的支持，并向客户端返回最终的查询结果集。查询处理模块的功能依赖于记录管理模块和索引管理模块。
  3. 语法分析：本模块提供对SQL命令的语法分析，并将分析结果以语法树的形式提供给系统管理模块和查询处理模块，以便其做进一步的分析和处理。
  4. 用户界面：本模块为用户提供操作界面，接收用户输入的SQL命令，并向用户输出SQL命令的执行结果。

在图3.1显示的各模块中，蓝色模块的功能代码已经提供，红色模块则需要学生自行设计实现。学生根据自己的实际能力，也可以对蓝色模块的代码进行一定程度的修改以扩展系统功能，但原则上不应修改本文档中规定的模块接口。

任务分工：

组员A：范明展，负责记录管理模块。

组员B：吴剑锋， 负责系统管理和查询处理模块。

组员C：孟嵩淼，负责索引管理模块。

# 索引管理模块

## 模块概述

本模块用于管理存储在索引文件中的索引信息，索引文件也是在页面文件的基础上改造的，它对上层模块提供以索引项为单位的文件读写接口。索引管理类似于记录管理，但不同于记录管理模块中的顺序存储记录的做法，索引管理采用B+树结构来组织索引项。利用B+树索引，查询时可以快速定位某条记录所在的位置，从而避免对数据表做全面扫描。

## 模块设计

1. B+Tree 定义

B+树是一种多路平衡树，树中的每个节点包含一组有序的关键字和指针。非叶子节点可包含的最大分支数称为该B+树的序数（order）。如果一个B+树的序数为m，则B+树应满足下列的性质：

1. 根节点只有一个，分支数范围为[2, m]
2. 除根以外的每个内部节点的分支数范围为[m/2, m]；
3. 叶子节点包含的关键字个数范围为[m/2, m]；
4. 所有节点中的关键字按升序排列；
5. 非叶子节点中的指针P[i]指向关键字值属于[K[i-1], K[i])的子树；
6. 所有的叶子节点都在同一层；
7. 叶子节点中包含了全部关键字，及指向包含这些关键字记录的指针，且叶子节点本身依关键字的大小自小而大顺序链接。
8. 索引文件

一个B+树索引存储为一个索引文件，文件由若干页面构成，每个索引节点占用一个页面。



图2.1 索引文件结构示意图

类似于记录文件，索引文件的第0个页面也是存放页面控制信息。与记录文件不同的是，索引文件从第1页就开始存放索引数据。第1页与后续的索引页略有不同，该页从pData[0]开始，还存放了一个描述索引控制信息的数据结构：

typedef struct{

int attrLength; //建立索引的属性值的长度

int keyLength; //B+树中关键字的长度

AttrType attrType; //建立索引的属性值的类型

PageNum rootPage; //B+树根节点的页面号

PageNum first\_leaf; //B+树第一个叶子节点的页面号

int order; //B+树的序数

int height; //B+Tree的高度

}IX\_FileHeader;

除第1页外，其它索引数据页面的这片区域为空白区域，不存储有效信息。索引控制信息结构在索引建立的时候初始化，并且随着索引文件的变化，其中的rootPage会发生变化。

紧跟在索引控制信息结构之后，从pData[sizeof(IX\_FileHeader)]开始，存放当前页面所存储节点的节点控制信息，对应的数据结构为：

typedef struct{

int is\_leaf; //该节点是否为叶子节点

int keynum; //该节点实际包含的关键字个数

PageNum parent; //指向父节点所在的页面号

PageNum prev; //指向左兄弟节点所在的页面号

PageNum next; //指向右兄弟节点所在的页面号

char \*keys; //指向关键字区的指针

RID \*rids; //指向指针区的指针

}IX\_Node;

紧接IX\_Node结构之后，从pData[sizeof(IX\_FileHeader)+ sizeof(IX\_Node)]开始，存放B+树节点信息。每个节点中包含一组关键字和一组指针，关键字和指针的数量相等。对于非叶子节点来说，指针指向的是下层子树的一个根节点；对于叶子节点来说，指针指向的则是包含关键字的那条记录。

一个B+树节点占据一个页面，页面的大小固定，但是由于不同的索引关键字的大小不同，因此不同索引的B+树节点能够容纳的关键字的最大数量也就不同，即不同索引的B+树序数m不同。这是在索引创建时通过给定的属性值的大小来进行计算的。

计算公式为：

m=(PF\_PAGE\_SIZE-控制信息长度)/(2\*sizeof(RID)+atrrLength)

## 模块实现

1. RC CreateIndex (const char \*fileName, AttrType attrType,int attrLength)

此函数创建一个名为fileName的索引。attrType描述被索引属性的类型，attrLength描述被索引属性的长度。

实现：

创建一个名为fileName的分页文件并打开，分配一个page作为B+Tree的根节点（页号为1），对根节点中的各项信息进行初始化如下：

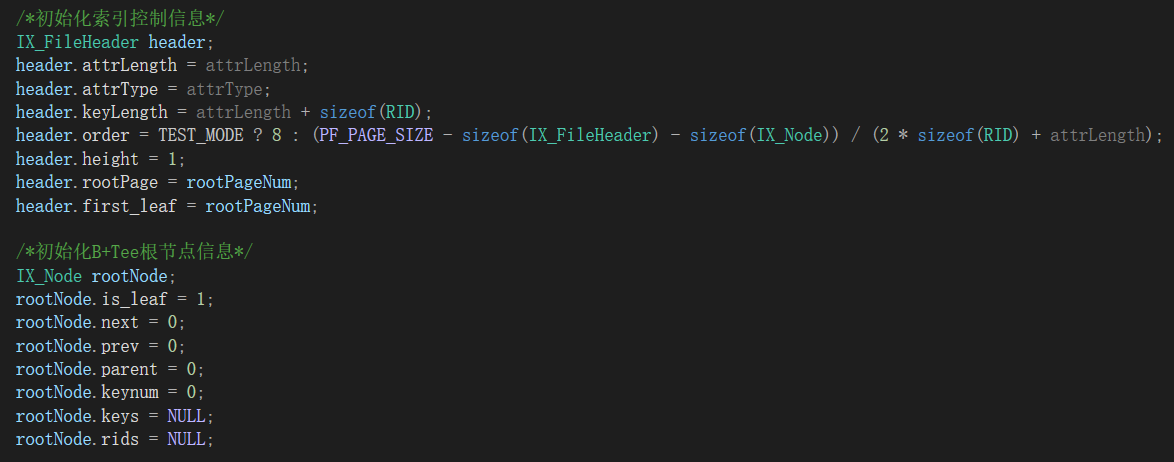


图2.2 B+Tree初始化

1. RC map(IX\_IndexHandle\* indexHandle, IX\_DataNode\* dataNode, PageNum pageNum)

RC unmap(IX\_IndexHandle\* indexHandle, IX\_DataNode\* dataNode, PageNum pageNum)

对页面管理模块提供的接口进行封装，得到一对读写函数，分别对分页文件中的某一页进行读取和写入。

IX\_DataNode定义如下，它为B+Tree节点在内存中的存在形式，node\_info存放节点相关信息，buffer存放关键字数组和RID数组：

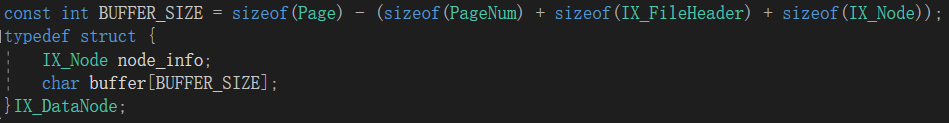


图2.3 IX\_DataNode定义

map负责读取硬盘中编号为pageNum的页，并写入到dataNode。map函数在调用页面管理模块的接口得到该page的数据后，需要对node\_info中的keys和rids进行初始化，使它们指向正确的位置。

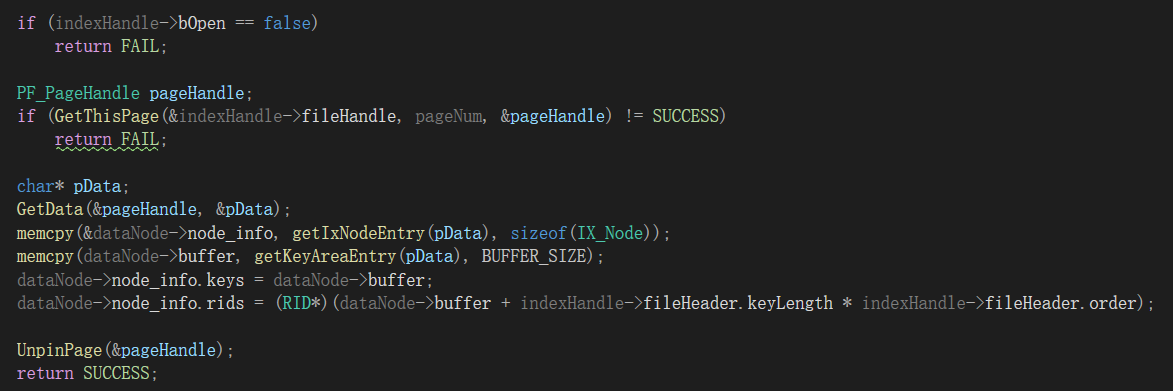


图2.3 map实现

unmap负责将dataNode中的节点内容写入到硬盘中编号为pageNum的页。再写入信息后，需要将该页标记为dirty，同时解除该页面，以确保更新后的信息会从缓存中写入到磁盘内。

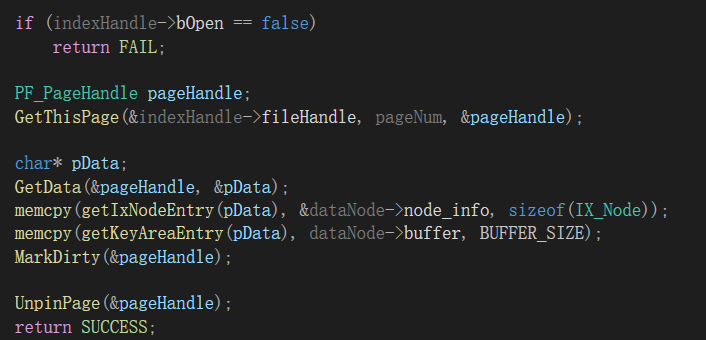


图2.4 unmap实现

1. RC OpenIndex (const char \*fileName, IX\_IndexHandle \*indexHandle)

此函数打开名为fileName的索引文件。如果方法调用成功，则indexHandle为指向被打开的索引句柄的指针。索引句柄用于在索引中插入或删除索引项，也可用于索引的扫描。

实现：

打开名为fileName的分页文件，并将文件句柄保存在indexHandle->fileHandle中，同时获得编号为1的page读入内存中，将它保存索引控制信息写入到indexHandle->fileHeader中。

1. RC CloseIndex(IX\_IndexHandle \*indexHandle)

此函数关闭句柄indexHandle对应的索引文件。

实现：

该函数会将索引控制信息写回磁盘，之后关闭indexhandle->fileHandle对应的分页文件。

1. int lower\_bound(IX\_IndexHandle\* indexHandle, IX\_DataNode\* dataNode, const char\* key, int begin, int end)

目标数组为dataNode中keys[begin, end), 返回首个大于等于key的元素的指针偏移，若没有则返回尾指针偏移，它的实现参考了C++ std::lower\_bound，使用了二分查找法。

int upper\_bound(IX\_IndexHandle\* indexHandle, IX\_DataNode\* dataNode, const char\* key, int begin, int end)

目标数组为dataNode中keys[begin, end), 返回首个大于key的元素的指针偏移，若没有则返回尾指针偏移, C++ std::upper\_bound，使用了二分查找法。

1. PageNum search\_index(IX\_IndexHandle\* indexHandle, const char\* key)

该函数负责找到关键字key所在叶子节点的父节点的编号。它从索引控制信息获得B+Tree的高度height，因此非叶子节点的高度为height-1，从根节点，循环height-2次，根据之前的定义，P[i]指向的分支关键字范围是[K[i-1], K[i])，每次调用upper\_bound找到第一个大于key的关键字，否则返回最后一个分支，也就得到了key所在的分支。

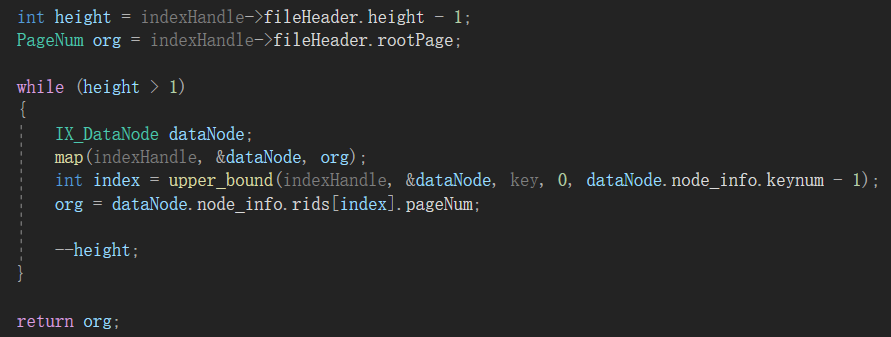


图2.5 search\_index实现

PageNum search\_leaf(IX\_IndexHandle\* indexHandle, const char\* key)

该函数负责找到关键字key所在的叶子节点的编号，它调用了search\_index函数找到目标叶子节点父节点的编号，之后在父节点内调用upper\_bound函数找到key所在的叶子节点。

1. RC InsertEntry (IX\_IndexHandle \*indexHandle,void \*pData, const RID \*rid)

此函数向IX\_IndexHandle对应的索引中插入一个索引项。参数pData指向要插入的属性值，参数rid标识该索引项对应的元组，即向索引中插入一个值为（\*pData，rid）的键值对。

实现：

根据pData和rid得到关键字key，调用search\_leaf函数找到待插入关键字所在的叶子节点编号，并将其读取到内存中。之后使用lower\_bound在叶子节点中查找，确定无重复关键字，代码如下：

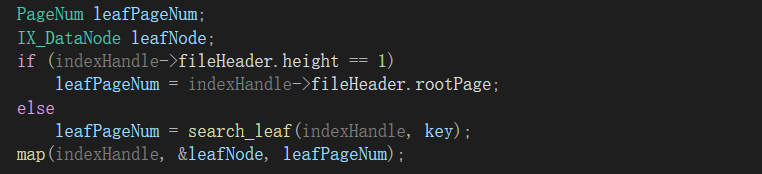


图2.6

若叶子节点关键字个数小于order，则无需分裂节点，直接插入关键字到叶子节点。否则，分裂叶节点得到左右两个叶节点，分割点为keynum / 2，之后根据key的大小，确定key插入到左节点还是右节点，根据结果，调整分割点以确保左右叶子节点的关键字个数之差的绝对值小于等于1，代码如下：



图2.7

根据分割点将数据拷贝到分裂得到的右节点，之后将关键字插入到目标节点，最终将左右节点写回磁盘，代码如下：

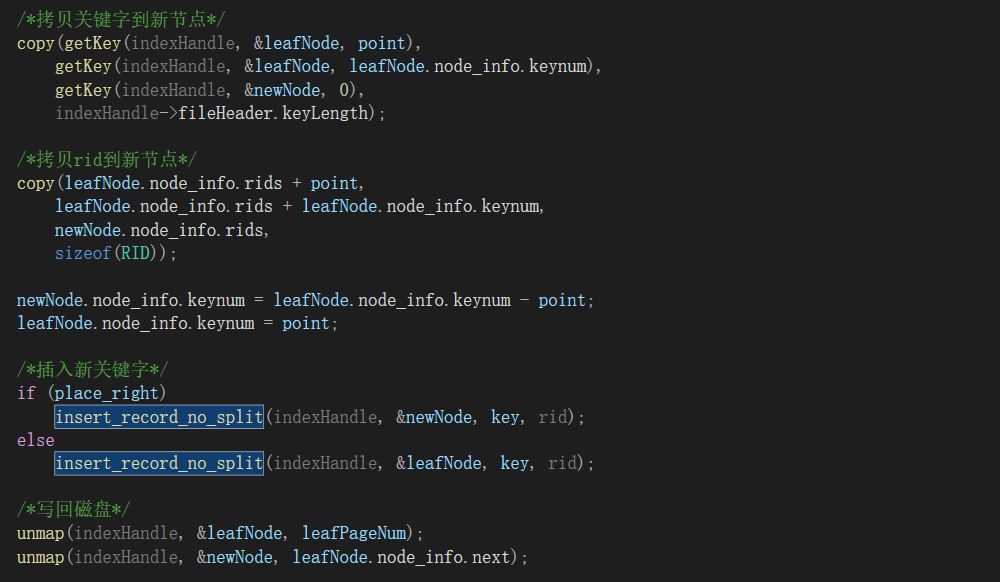


图2.8

最后一步，需要将分裂得到的新节点信息插入到父节点中。根据B+Tree的定义，左节点的上界即为右节点的第一个元素，因此将newNode.keys[0]插入到父节点，代码如下：

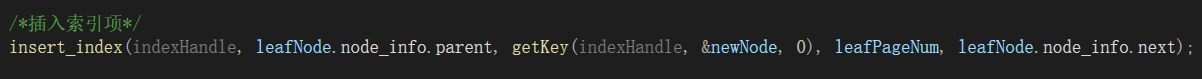


图2.9

void insert\_index(IX\_IndexHandle\* indexHandle, PageNum target, const char\* key, PageNum old, PageNum after)

该函数负责将关键字key插入到编号为target的内部节点,old和after分别指向分裂得到的两个节点。

实现：

若target为0，表明当前根节点被分裂，因此需要创建新的根节点，并将分裂后的两个节点作为新根节点的两路分支，代码如下：

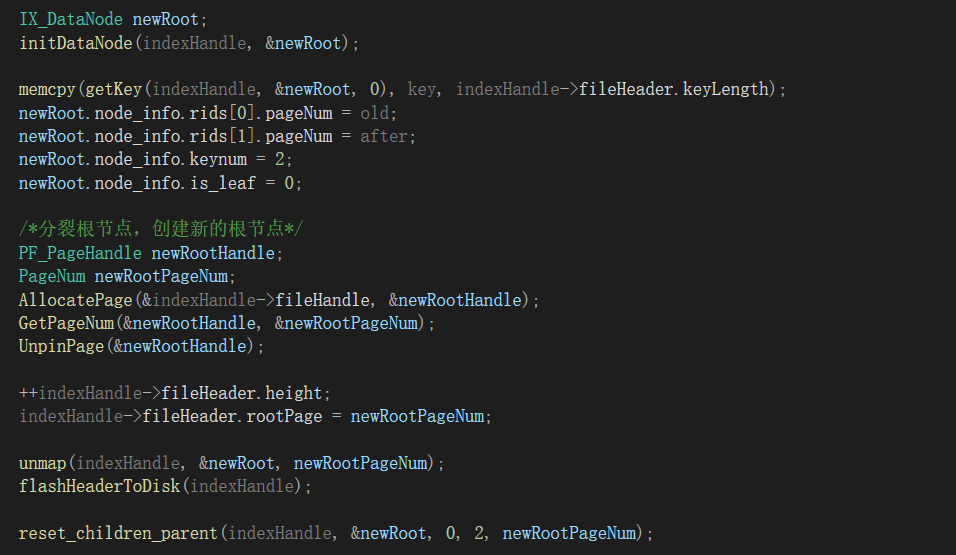


图2.10

若target不为0.则表明被分裂的是非根内部节点。与插入叶子节点类似，若当前节点非满节点，则直接插入，无需分裂，代码如下：

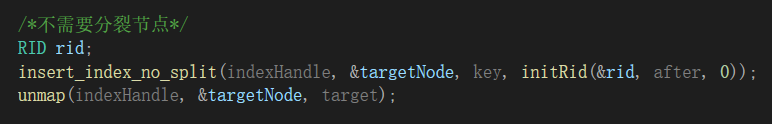


图2.11

否则，需要对内部节点进行分裂。以中点作为分割点，判断待插入关键字的位置，之后调整分割点保持树的平衡，同时需要避免插入项与它的右兄弟被分割到两个节点，原因在于需要交换待插入位置与它的右分支的指向，代码如下：

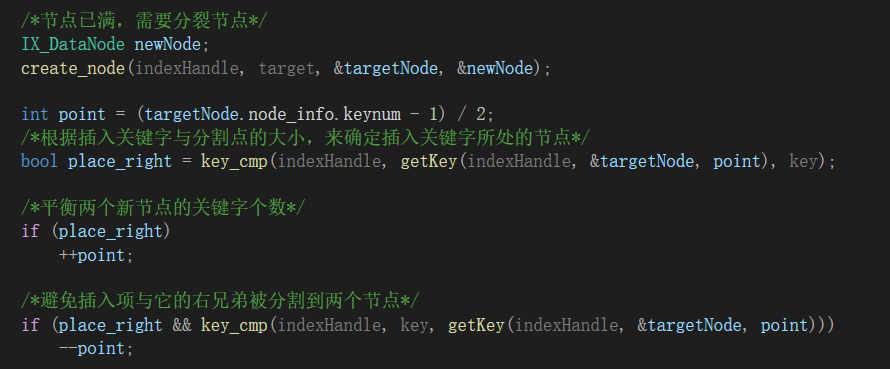


图2.12

将关键字组和RID组拷贝到新节点，代码如下：

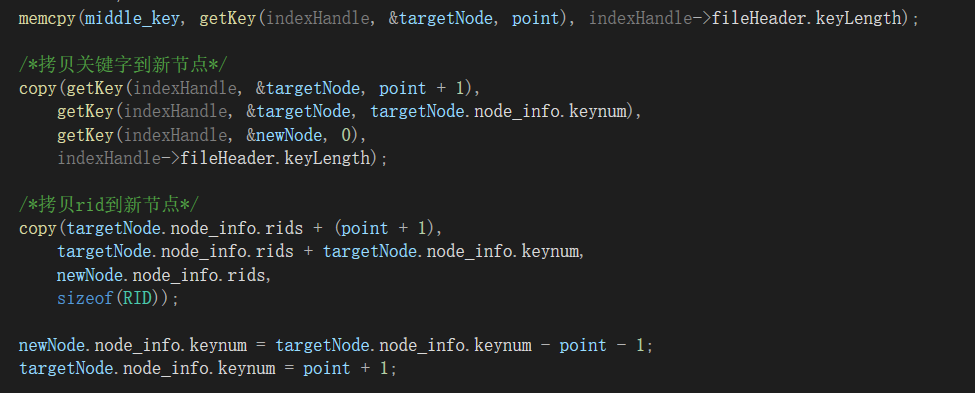


图2.13

插入关键字，并将两个节点写回磁盘，同时对于分裂得到的新节点，它的孩子节点中的父节点信息应被修改为新节点的页号。最终，调用insert\_index，递归插入关键字到当前内部节点的父节点，代码如下：

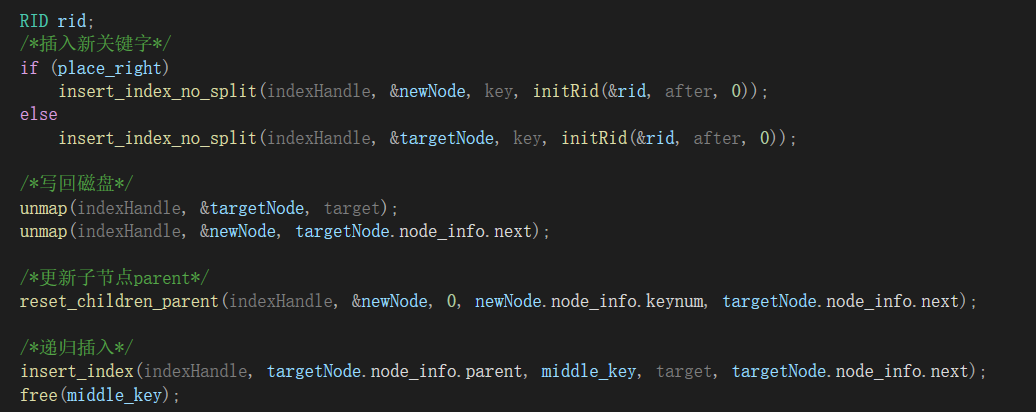


图2.14

1. RC DeleteEntry (IX\_IndexHandle \*indexHandle,void \*pData, const RID \*rid)

此函数从IX\_IndexHandle句柄对应的索引中删除一个值为（\*pData，rid）的索引项。

将pData和rid组合成关键字key，当B+Tree只有一层时，直接删除关键字即可。当B+Tree高度大于1时，调用search\_index找到目标关键字所在叶子节点的父节点页号，得到父节点内容并找到目标分支在key数组中的偏移，在得到叶子节点后，寻找第一个大于等于key的关键字，根据查找结果判断待删除关键字是否存在，代码如下：

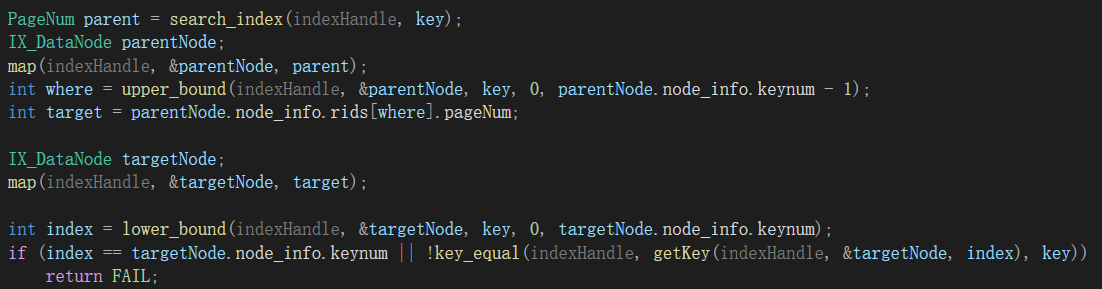


图2.15

若关键字存在，删除关键字后，判断节点内的关键字个数是否满足B+Tree条件，若满足，则直接写回磁盘即可；若不满足，首先考虑该节点向它的右兄弟或左兄弟借用一个关键字，要求它的兄弟节点存在，且借出一个关键字后，个数依然满足条件，若不满足，则借用失败，代码如下：

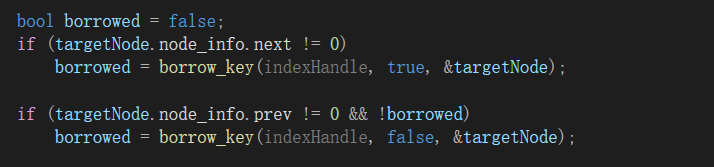


图2.16

若借用操作成功，则写回磁盘；否则，需要合并节点。根据该节点在父节点中的位置选择与它合并的兄弟节点。合并完成后，需要调用remove\_from\_index函数从内部节点进行删除操作，这与插入关键字是类似的，代码如下：



图2.17

void remove\_from\_index(IX\_IndexHandle\* indexHandle, PageNum target, IX\_DataNode\* targetNode, const char\* key)

该函数用于从内部节点targetNode删除关键字key所在的分支。

首先调用upper\_bound找到key所在的分支，之后删除该分支，并更新分支指向；当targetNode为根节点且删除后根节点分支数仅为1，则需要删除该根节点，并置它唯一的孩子节点为新的根节点，代码如下：

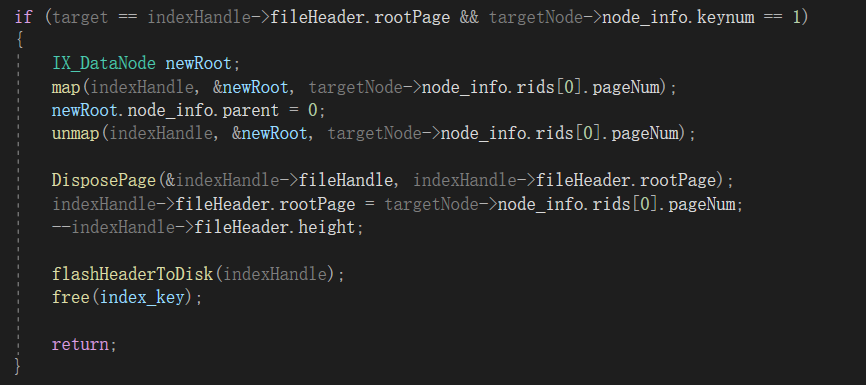


图2.18

与从叶子节点删除类似，当内部节点的关键字数目不满足条件时，需要借用关键字，或者进行兄弟间的合并。所不同的是借用操作只会与当前节点的同父母兄弟进行借用；而若选择合并节点，在合并后需要更改孩子节点的父节点信息。最后递归调用remove\_from\_index继续沿父节点向上删除。

1. RC OpenIndexScan(IX\_IndexScan \*indexScan, IX\_IndexHandle \*indexhandle, CompOp compOp, char \*value);

此函数用于在indexHandle对应的索引上初始化一个基于条件的扫描。compOp和\*value指定比较符和比较值，indexScan为初始化后的索引扫描结构指针。

为了处理非唯一索引的重复键值，将索引列的值和记录的ID值组合成索引关键字。在B+Tree的关键字比较中，依次比较属性值，pageNum，slotNum；因此在B+Tree的叶子节点序列中，属性值相同的项是连续放置的序列。若要查找属性值为value的项，使用value初始化待查找key，同时将待查找key的pageNum和slotNum初始化为最大值，则可以定位到B+Tree中第一个大于该属性值的项，value序列的尾部；若将待查找key的pageNum和slotNum初始化为最小值，则可以定位到第一个等于该属性值的项，value序列的头部；如下两种初始化方式：





对于“>=”, “=”比较符，需要将扫描的初始位置定位到value序列的头部；对于”>”比较符，需要将初始位置定位到value序列的尾部；对于”<>”操作符，扫描的初始位置定位到B+Tree叶节点序列的头部，但需要额外保存value序列的尾部，以作跳转使用。对于”<”，”<=”，”NO\_OP”比较符，扫描的初始位置定位到叶子节点序列的头部；

1. RC IX\_GetNextEntry(IX\_IndexScan \*indexScan, RID \*rid);

此函数用于继续IX\_IndexScan句柄对应的索引扫描，获得下一个满足条件的索引项，并返回该索引项对应的记录的ID。

对于”<>”比较符，将扫描放置到下一个位置，在第一次遇到不满足条件的项时，根据额外保存的信息，将扫描位置定位到value序列的尾部，跳过不满足条件的value序列，之后继续向后扫描，直到到达叶子序列的尾部；

对于其它操作符，将当前扫描的位置向后移动一位，直到遇到不满足条件的项或叶节点序列尾部结束；

向后移动操作如下，在叶节点内部时直接递增ridIx，当到达叶子节点尾部时，根据保存的右兄弟信息，继续向后扫描右兄弟，代码如下：

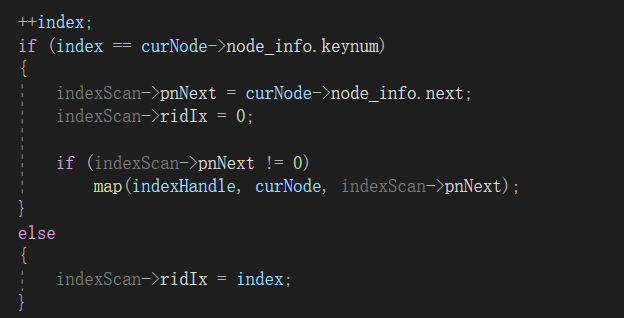


图2.19

1. RC CloseIndexScan(IX\_IndexScan \*indexScan);

关闭一个索引扫描，释放相应的资源。

1. RC GetIndexTree(char \*fileName, Tree \*index);

获取由fileName指定的B+树索引内容，返回指向B+树的指针。此函数提供给测试程序调用，用于检查B+树索引内容的正确性。以下为相关数据结构定义：

typedef struct{

int keyNum; //节点中包含的关键字（属性值）个数

char \*\*keys; //节点中包含的关键字（属性值）数组

Tree\_Node \*parent; //父节点

Tree\_Node \*sibling; //右边的兄弟节点

Tree\_Node \*firstChild; //最左边的孩子节点

}Tree\_Node; //节点数据结构

typedef struct{

AttrType attrType; //B+树对应属性的数据类型

Int attrLength; //B+树对应属性值的长度

int order; //B+树的序数

Tree\_Node \*root; //B+树的根节点

}Tree;

该函数的实现使用递归的方式遍历B+Tree，并将其全部读入内存；递归函数如下：

Tree\_Node\* createTree(IX\_IndexHandle\* indexHandle, PageNum root, Tree\_Node\* parent, Tree\_Node\* sibling)

该函数接受一个打开的索引句柄，以root为根节点，将它对应的子树读入内存，由于Tree\_Node需要保存父节点和右兄弟节点的信息，所以同时需要给出root节点的父节点和右兄弟节点的指针；因此磁盘中B+Tree的扫描和内存中B+Tree的构建是从上到下，从右到左，以确保在构建当前节点时，它的父节点和右兄弟节点已被创建；

首先递归的结束条件是到达叶子节点，此时从磁盘中读出叶子节点的信息，并将指定内容保存至创建的Tree\_Node中，最后返回叶子节点的指针，代码如下：

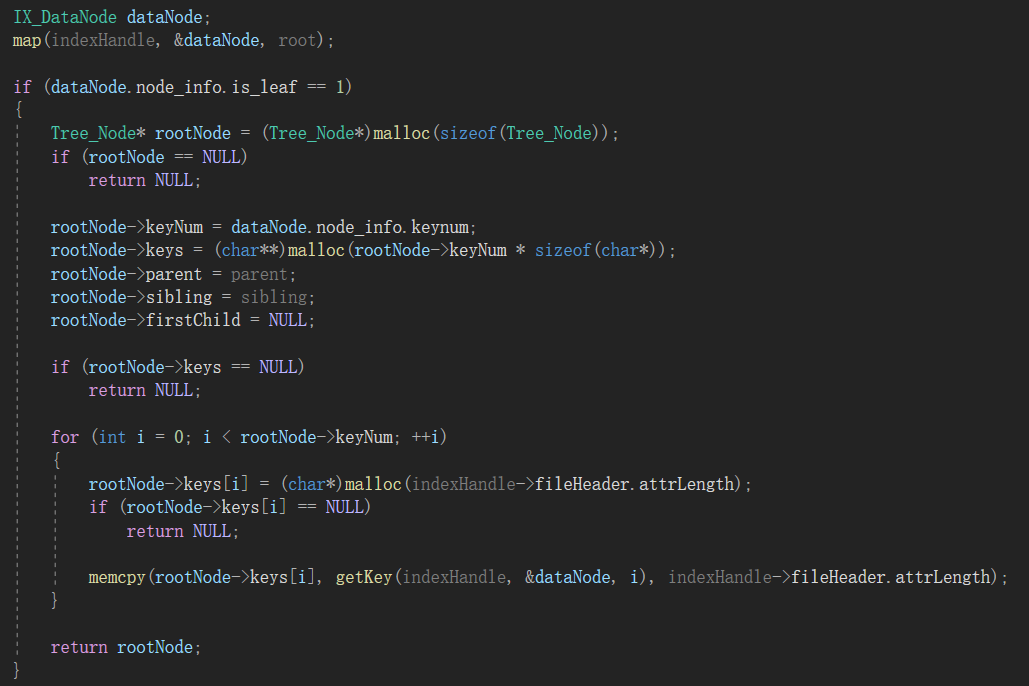


图2.20

对于内部节点，除了将指定的信息保存至创建的Tree\_Node中之外，需要从右到左，对它的孩子节点递归调用createTree函数创建子树，它的最右孩子的右兄弟即为它的右兄弟的第一个孩子，它的其它孩子的右兄弟，即为上一次递归调用创建的节点，最后一个创建的孩子节点保存至firstChild中去，最后返回rootNode，代码如下：

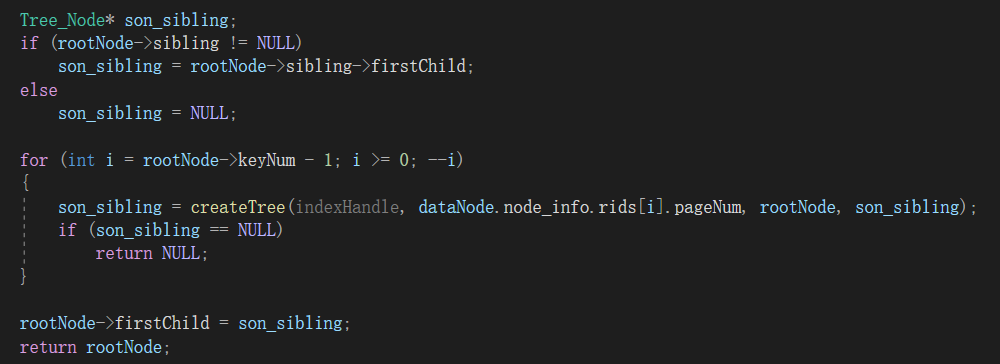


图2.21

因此最终要获得完整的B+Tree，对整棵树的根节点调用createTree即可，代码如下：

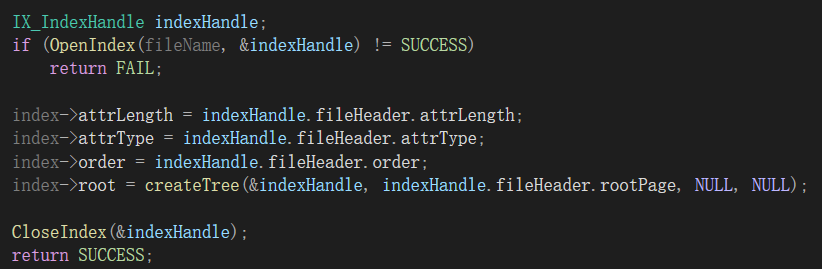


图2.22

## 实验过程

1. 调试过程
2. 调试1
3. 问题描述：插入关键字后，查找出错
4. 调试分析：在未分裂情况下插入记录到节点时忘记递增关键字数目；
5. 调试2
6. 问题描述：搜索关键字时，有些正确显示，有些则乱码
7. 调试分析：在OpenIndexScan函数中，将类型为IX\_DataNode的局部变量直接赋值给indexScan->currentNode，问题在于indexScan->currentNode中保存的指针keys和rids实际指向的是局部变量的关键字区和指针区，因此退出函数后，局部变量在栈中的内存被释放，它可能再次被覆盖，也可能没有，因此一部分结果正确，一部分错误。直接对indexScan->currentNode进行初始化使得指针指向正确即可解决；
8. 调试3
9. 问题描述：删除索引项时，有时会产生内存错误
10. 调试分析：在删除内部节点的索引项并进行节点合并时，忘记修改被合并的内部节点的孩子节点的父节点信息，因此再次删除索引项，并向上递归删除时，会使用过时的父节点信息，因此所访问的page已被释放或者被分配给了其它节点；
11. 单元测试
12. 插入删除测试：

将程序设定为测试模式，这样B+Tree的序数被固定为8，方便打印叶子节点的内容。

测试用例：属性列字段为char类型，长度16，内容为[1, 100000, 100]中的整数的16进制字符串，每个重复10次，插入到B+Tree中，最终每个属性值删除8个，代码如下：

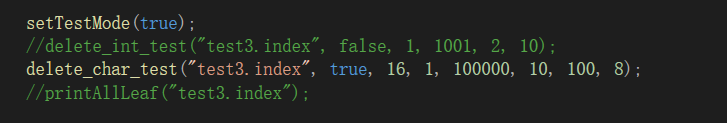


图2.23

测试结果：

插入后，图中每一行打印一个叶子节点的属性值内容：



图2.24 插入后

删除操作后：

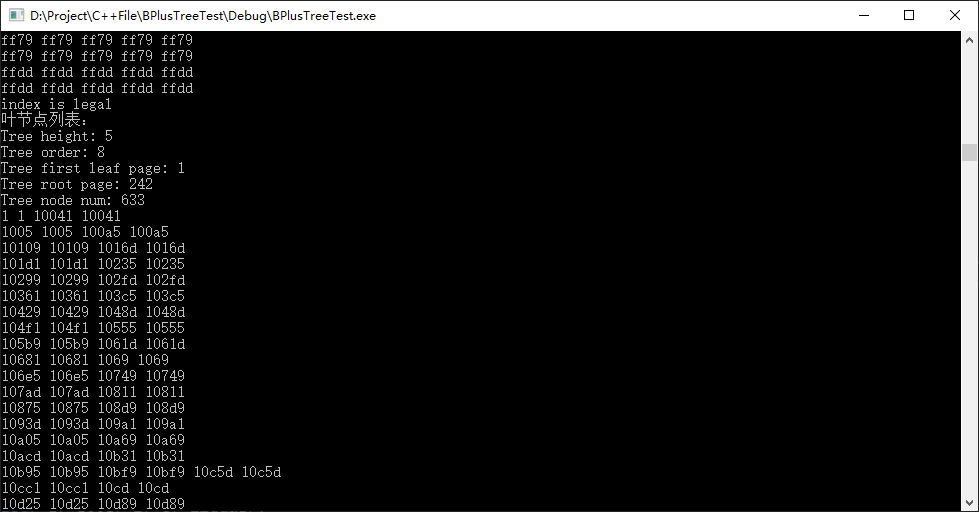


图2.25 删除后

1. 插入查询测试：

测试用例：

将程序设置为非测试模式，这样B+Tree的序数不受限制；



插入属性值序列[begin, end, interval)的整数，每个值重复num次，RID值中pageNum与属性值相同，重复属性值的slotNum在pageNum的基础上递增，之后在几个测试点进行查询；测试样例：

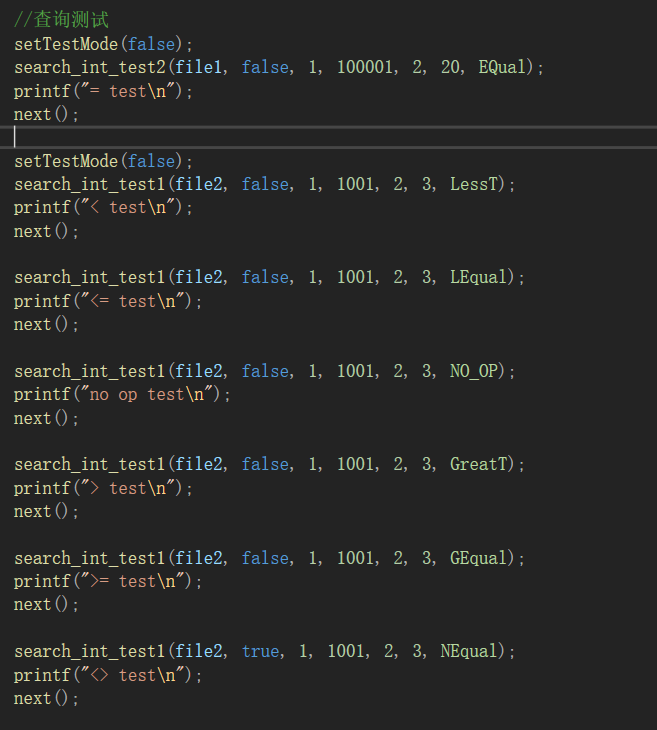


图2.26

1. “=”测试

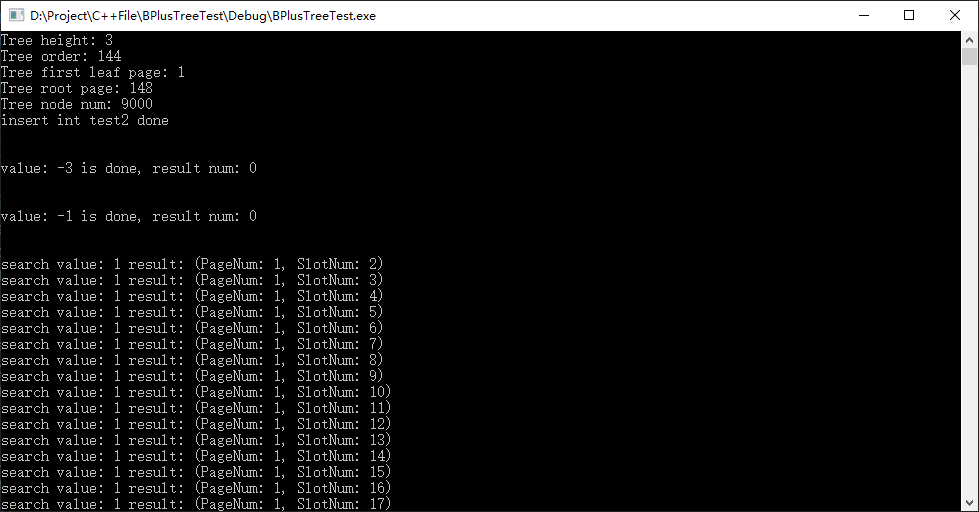


图2.27

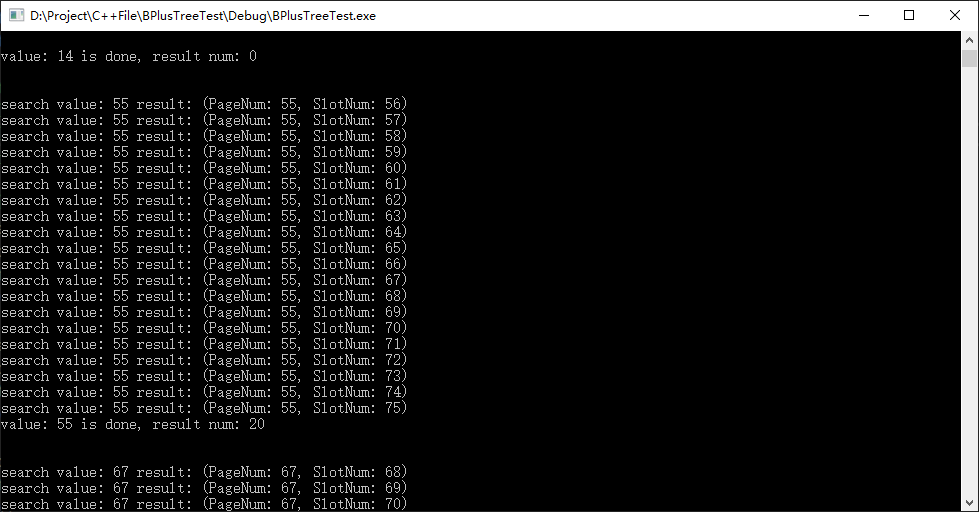


图2.28

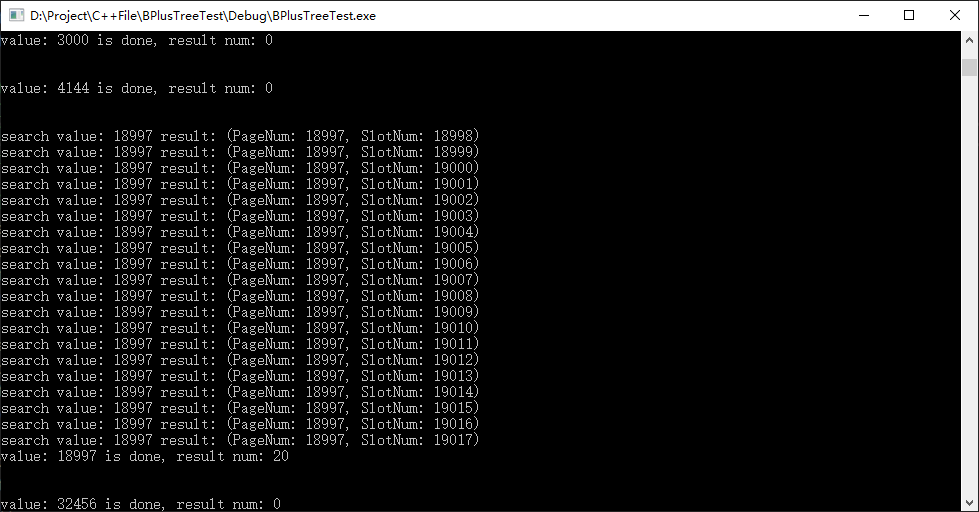


图2.29

1. “<” 测试

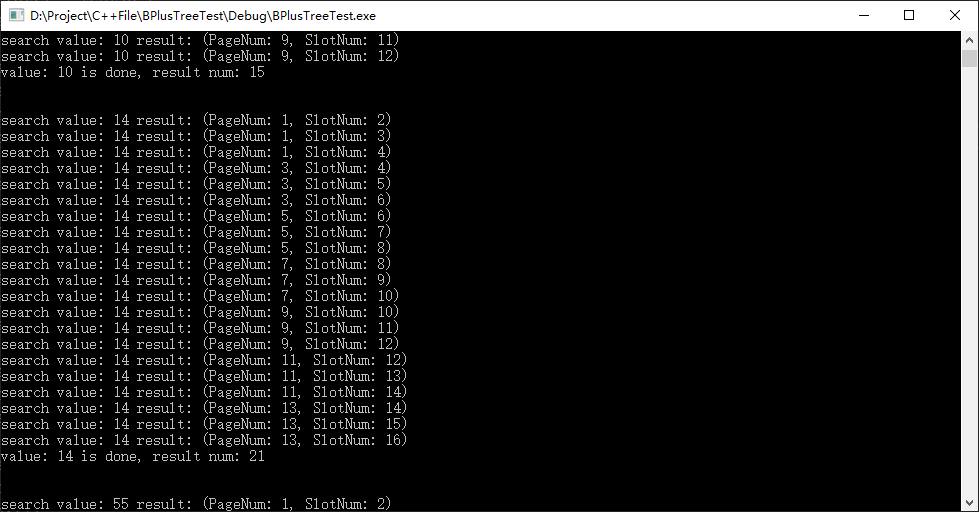


图2.30

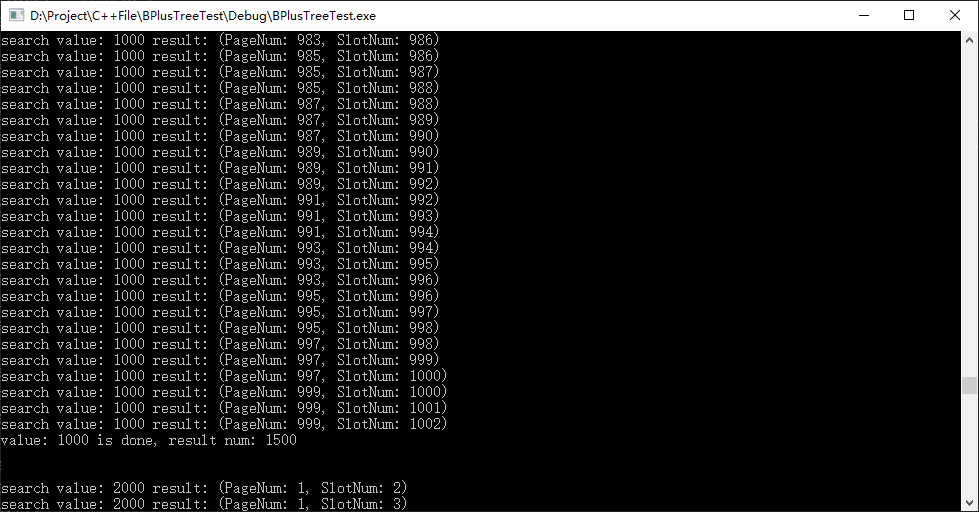


图2.31

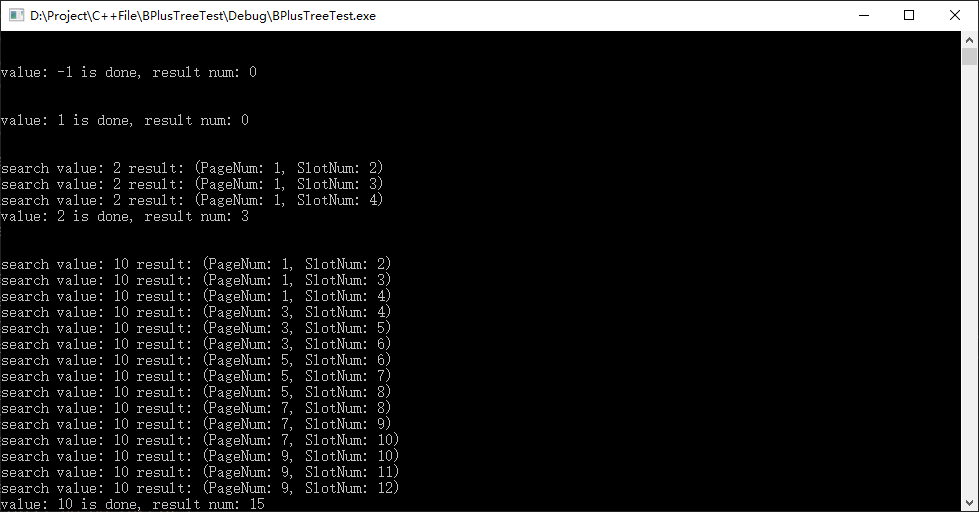


图2.32

1. “<=” 测试

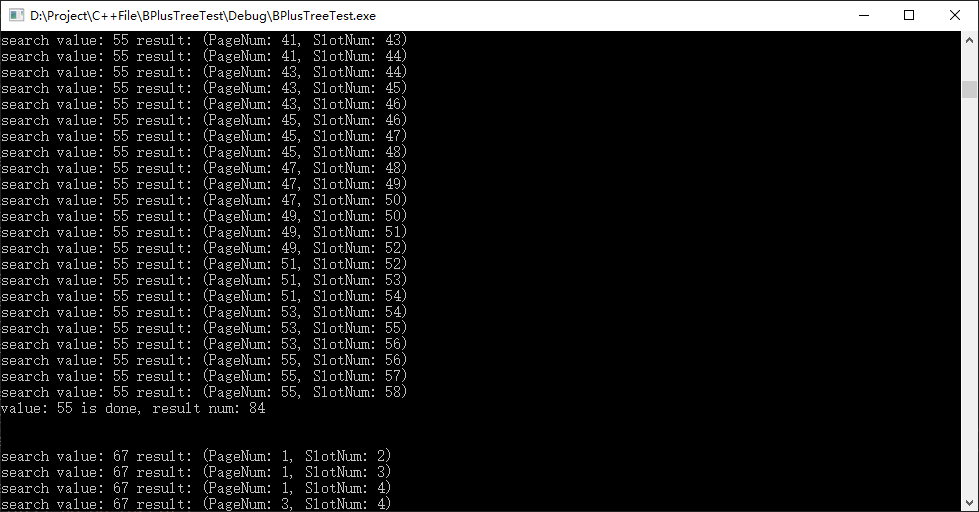


图2.33

1. “no\_op” 测试

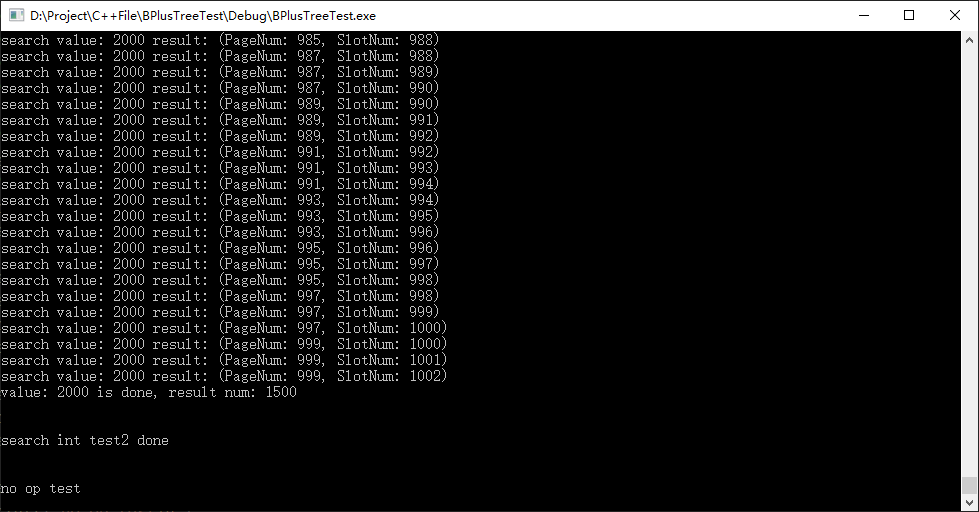


图2.34

1. “>” 测试

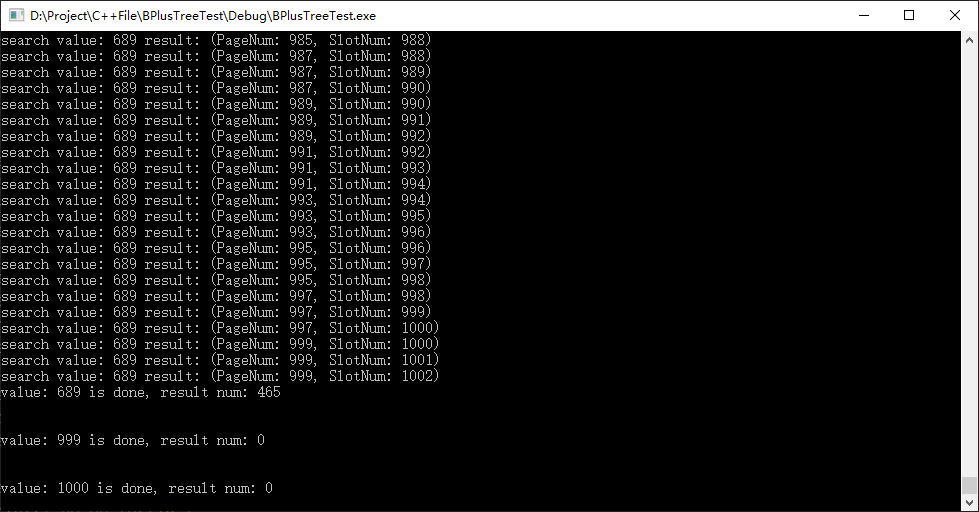


图2.35

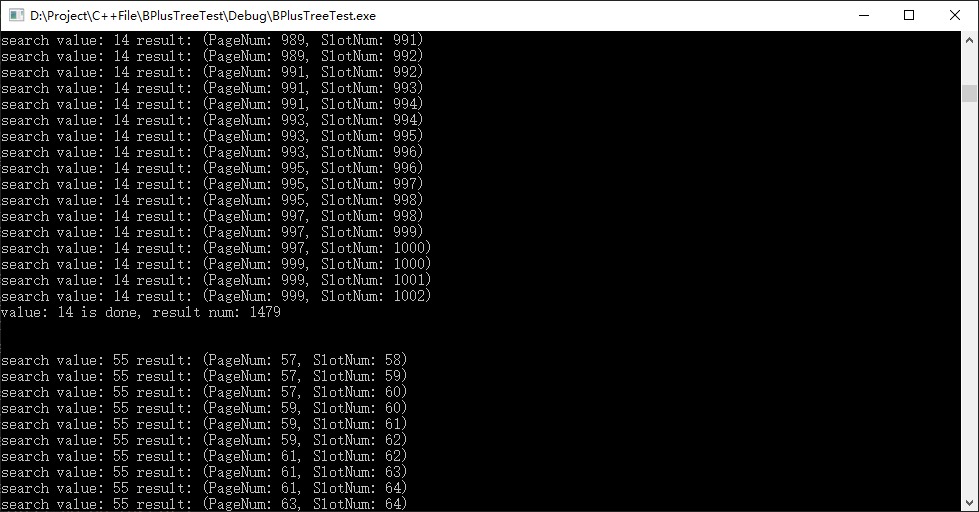


图2.36

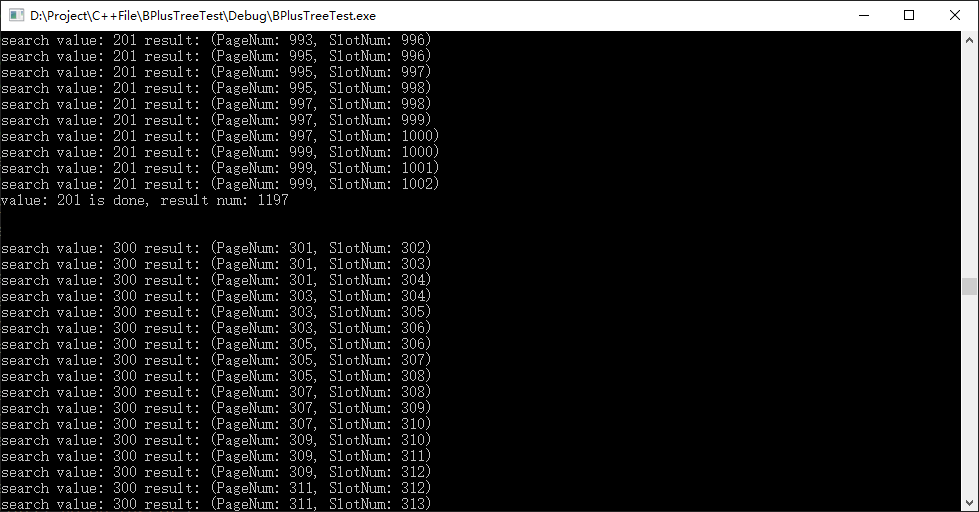


图2.37

1. “>=” 测试

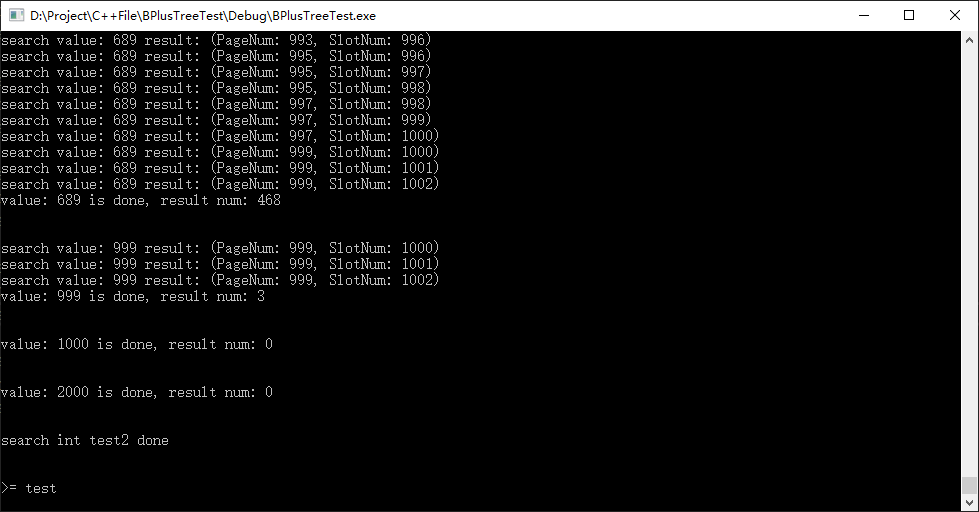


图2.38

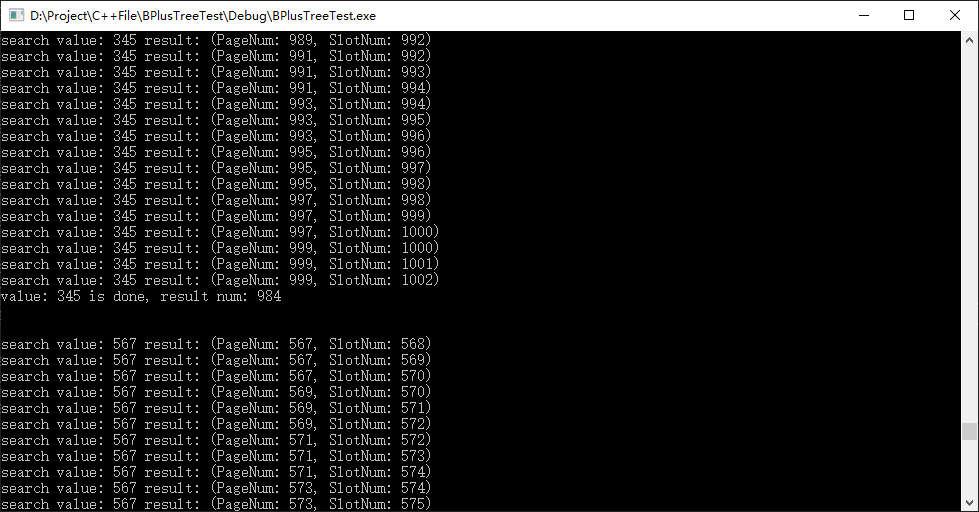


图2.39

1. “<>” 测试

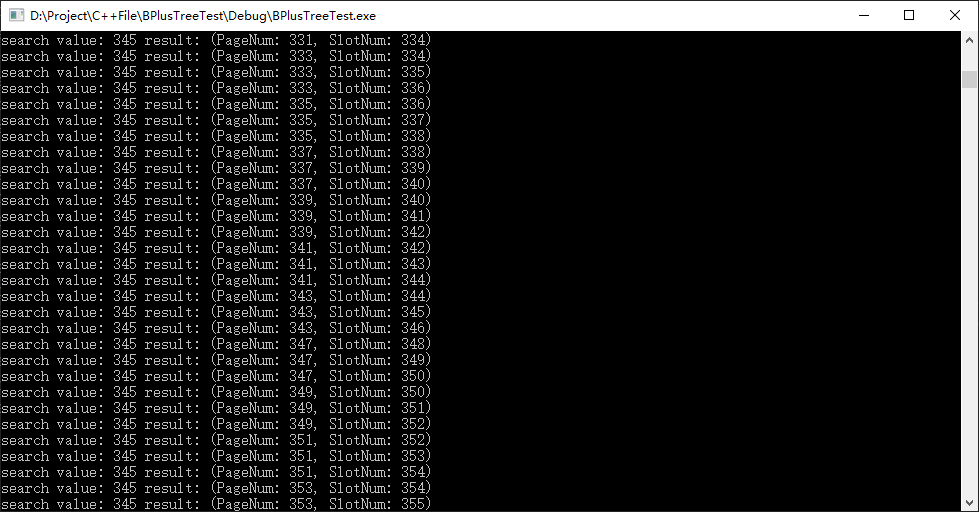


图2.40

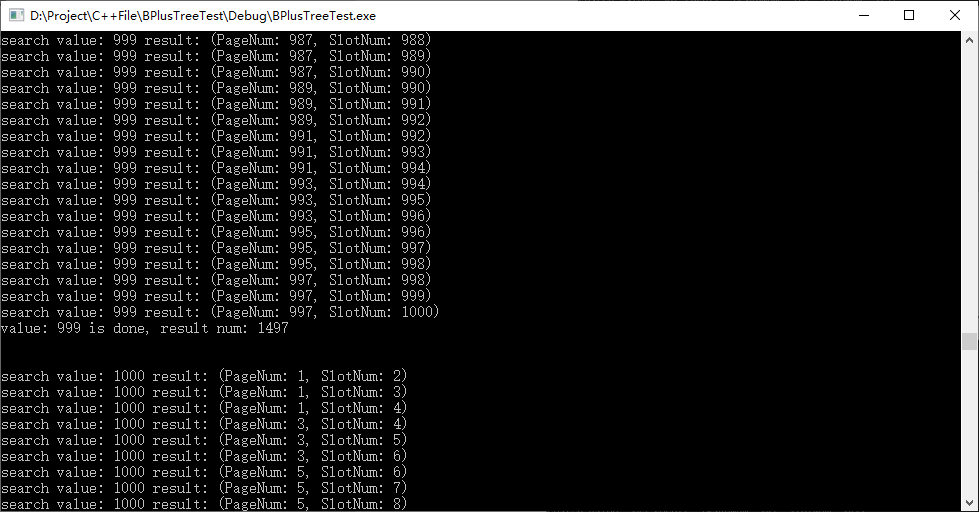


图2.41

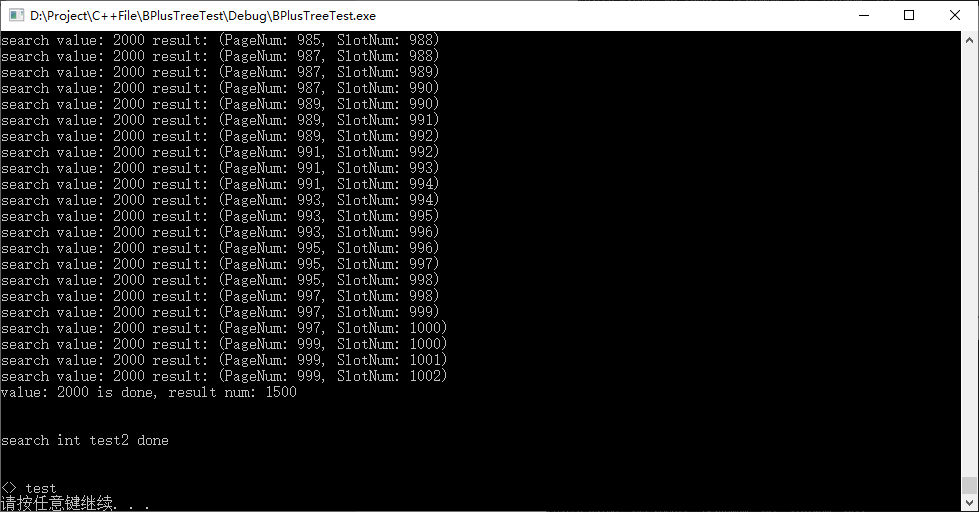


图2.42

# 实验总结

## 工作总结

本次实验我的工作是索引模块的开发，并进行独立测试，最终与其他部分进行联调；整个索引模块的核心部分是索引项的插入和删除；

对于插入操作，需要考虑当叶节点已满时需要进行的分裂操作，同时需要向上递归插入关键字到内部节点，因此将叶子节点的插入与内部节点的插入分别定义；

对于删除操作，为插入操作的逆过程，同样分为两个函数；每个节点同时保存左右两兄弟的页号，这样当进行借用操作时，可以有左右两个选择，只有当左右兄弟均无法借用时才会考虑代价较大的合并节点操作。

对于最终的查询功能，在块外部沿根节点逐层向下定位，在块内使用二分查找法查找索引项，减小了查询所需的IO次数，提高了查询速度；

为了进行测试，将索引模块以及它所引用的其它模块放置在一个单独的工程文件中，并编写测试程序进行测试，检验索引模块的功能正确性。

## 改进方案

在B+Tree设计中，为了保证索引关键字不重复，利用了RID的唯一性，将属性值和RID拼接得到唯一的索引关键字。但是缺点在于使得关键字字段有了一个固定的增量，使得B+Tree的序数减小，一个page所能够保存的关键字数目减小，当关键字数目不变时，B+Tree的高度将会增加，那么查找到一个关键字所需要的磁盘IO次数相应增加。改进方案为设计树的两种形式，若定义索引的列为非unique，则使用上述方法解决关键字重复问题，若定义索引的列为unique，则去掉索引关键字中的RID字段。

另一种解决关键字重复的方案是直接使用属性值作为索引关键字，在叶子节点的指针区，每个关键字指向一个保存着具有相同属性值的对应RID值的磁盘内链表，链表的一个节点占据一个page，但是这种方案的缺点在于如果属性值的重复次数并不多，那么大多数磁盘块仅保存了一条或几条记录，浪费了存储空间。

由于叶子节点和内部节点定义为同一种数据结构，因此内部节点的指针区使用RID中的pageNum保存页号，而slotNum和valid字段并未利用，同样造成了空间浪费，因此可以考虑额外定义内部节点，去掉未使用的字段；