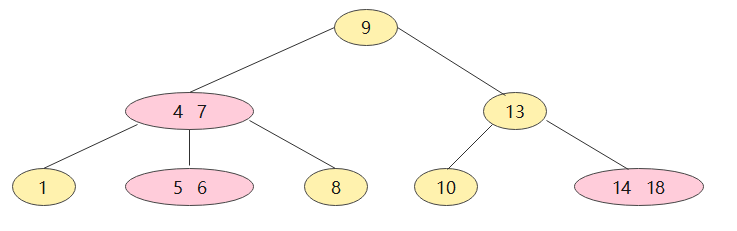
设计思想：

1. 关于B树和B+树

|  |  |
| --- | --- |
| B树 | 多路搜索树，每个结点存储M/2到M个关键字，非叶子结点存储指向关键字范围的子结点；所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中； |
| B+树 | 在B树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中； |

B树其实最开始源于的是二叉树，二叉树是只有左右孩子的树，当数据量越大的时候，二叉树的节点越多，那么当从根节点搜索的时候，影响查询效率。所以如果这些节点存储在外存储器中的话，每访问一个节点，相当于进行了一次I/O操作。

B树的特点：（m是树的高度，k是树的阶）



1.根结点至少有两个子节点。

2.每个中间节点都包含k-1个元素和k个孩子，其中 m/2 <= k <= m 。

3.每一个叶子节点都包含k-1个元素，其中 m/2 <= k <= m。

4.所有的叶子结点都位于同一层。

5.每个节点中的元素从小到大排列，节点当中k-1个元素正好是k个孩子包含的元素的值域分划。

B+树的优点：

B+树的磁盘读写代价更低。B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了；

B+树查询效率更加稳定。由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；

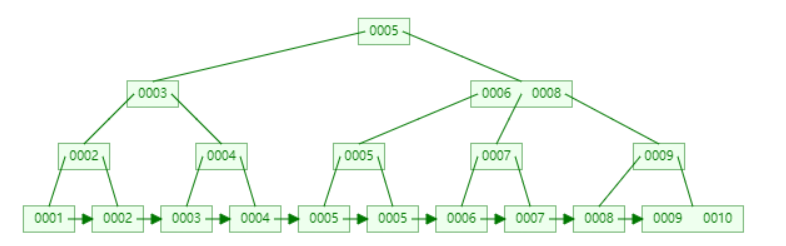
B+树便于范围查询（最重要的原因，范围查找是数据库的常态）。B树在提高了IO性能的同时并没有解决元素遍历的时候效率低下的问题，正是为了解决这个问题，B+树应用而生。B+树只需要去遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作或者说效率太低；

B+树的缺点：

主键不是有序递增的，导致每次插入数据产生大量的数据迁移和空间碎片；

即使主键是有序递增的，大量写请求的分布仍是随机的。

B+树样式：



（B+树可视化地址：<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>）

实现方案：

数据结构：（采用java实现）

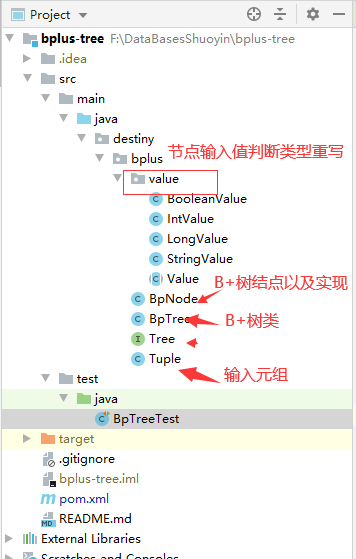
|  |
| --- |
| import java.util.ArrayList; import java.util.List; public class BpNode {   */\*\*  \* 是否为叶子节点  \*/* boolean isLeaf;   */\*\*  \* 是否为跟节点  \*/* boolean isRoot;   */\*\*  \* 父节点  \*/* BpNode parent;   */\*\*  \* 叶节点的前节点  \*/* BpNode previous;   */\*\*  \* 叶节点的后节点  \*/* BpNode next;   */\*\*  \* 节点的关键字列表  \*/* List<Tuple> entries;   */\*\*  \* 节点的指针列表  \*/* List<BpNode> children;   */\*\*  \* 节点指针的最大值  \*/* final int maxLength = 5;   public BpNode getParent() {  return parent;  }   public void setParent(BpNode parent) {  this.parent = parent;  }   public BpNode getNext() {  return next;  }   public List<Tuple> getEntries() {  return entries;  }   public List<BpNode> getChildren() {  return children;  } |

Tree树类接口：

|  |
| --- |
| package destiny.bplus; public interface Tree {   Tuple find(Tuple key);   boolean remove(Tuple key);   void insert(Tuple key); } |

B+树类：

|  |
| --- |
| package destiny.bplus; public class BpTree implements Tree {   */\*\*  \* B+树根节点  \*/* BpNode root;   */\*\*  \* B+树叶子节点的头结点  \*/* BpNode head;   public BpTree() {  root = new BpNode(true, true);  head = root;  }   public BpNode getHead() {  return head;  }   public BpNode getRoot() {  return root;  }   @Override  public Tuple find(Tuple key) {  return root.get(key);  }   @Override  public void insert(Tuple key) {  System.*out*.println("insert " + key.getValues()[0]);  root.insert(key, this);  }   @Override  public boolean remove(Tuple key) {  return root.remove(key, this);  }   */\*\*  \* 验证树本身是否符合B+树规范  \*/* public boolean validate() {  return root.validate();  } } |



文件结构：

实现细节：

查询元组节点：

|  |
| --- |
| BpNode root;  @Override public Tuple find(Tuple key) {  return root.get(key); }  public Tuple get(Tuple key) {  if (isLeaf) {  for (Tuple tuple : entries) {  if (key.compare(tuple) == 0) {  return tuple;  }  }  return null;  } else {  *// 小于首节点* if (key.compare(entries.get(0)) < 0) {  return children.get(0).get(key);  } else if (key.compare(entries.get(entries.size() - 1)) >= 0) {  return children.get(children.size() - 1).get(key);  } else {  *// TODO 后续改为二分查找* for (int i = 0; i < (entries.size() - 1); i++) {  if (key.compare(entries.get(i)) >= 0 && key.compare(entries.get(i + 1)) < 0) {  return children.get(i + 1).get(key);  }  }  }  }  return null; } |

B+树节点插入：

|  |
| --- |
| @Override public void insert(Tuple key) {  System.*out*.println("insert " + key.getValues()[0]);  root.insert(key, this); }  public void insert(Tuple key, BpTree tree) {  if (isLeaf) {  if (!isLeafToSplit()) {  *// 叶节点无需分裂* System.*out*.println("直接插入叶节点");  insertInLeaf(key);  } else {  System.*out*.println("插入叶节点,且叶节点分裂");  *//需要分裂为左右两个节点* BpNode left = new BpNode(true);  BpNode right = new BpNode(true);  if (previous != null) {  left.previous = previous;  previous.next = left;  } else {  tree.head = left;  }  if (next != null) {  right.next = next;  next.previous = right;  }  left.next = right;  right.previous = left;  *// for GC* previous = null;  next = null;  *// 插入后再分裂* insertInLeaf(key);   int leftSize = getUpper(entries.size(), 2);  int rightSize = entries.size() - leftSize;  System.*out*.printf("leaf key left:%d right:%d\n", leftSize, rightSize);  *// 左右节点拷贝* for (int i = 0; i < leftSize; i++) {  left.entries.add(entries.get(i));  }  for (int i = 0; i < rightSize; i++) {  right.entries.add(entries.get(leftSize + i));  }  *// 不是根节点* if (!isRoot) {  *// 调整父子节点关系  // 寻找当前节点在父节点的位置* System.*out*.println("parent children is null:" + (parent.children == null));   int index = parent.children.indexOf(this); *// System.out.println("parent children size:" + parent.children.size()); // System.out.println("index:" + index);   // 删除当前指针* parent.children.remove(this);  left.setParent(parent);  right.setParent(parent);  *// 将分裂后节点的指针添加到父节点* parent.children.add(index, left);  parent.children.add(index + 1, right);  *// for GC* entries = null;  children = null;   *// 父节点[非叶子节点]中插入关键字* parent.insertInParent(right.entries.get(0));  System.*out*.println("父节点插入key");  parent.updateNode(tree);  *// for GC* parent = null;  } else {  *// 是根节点* System.*out*.println("生成新的根节点");  isRoot = false;  BpNode rootNode = new BpNode(false, true);  tree.root = rootNode;  left.parent = rootNode;  right.parent = rootNode;  rootNode.children.add(left);  rootNode.children.add(right);  *// for GC* entries = null;  children = null;  *// 根节点插入关键字* rootNode.insertInParent(right.entries.get(0));  }  }  } else {  *// 如果不是叶子节点,沿着指针向下搜索* if (isRoot) {  System.*out*.println("(1)跟节点,向下搜索");  }  if (key.compare(entries.get(0)) < 0) {  System.*out*.println("中间节点,向下搜索");  children.get(0).insert(key, tree);  } else if (key.compare(entries.get(entries.size() - 1)) >= 0) {  System.*out*.println("中间节点,向下搜索");  children.get(children.size() - 1).insert(key, tree);  } else {  *// TODO 二分查找  // 遍历比较* System.*out*.println("中间节点,向下搜索");  for (int i = 0; i < (entries.size() - 1); i++) {  if (key.compare(entries.get(i)) >= 0 && key.compare(entries.get(i + 1)) < 0) {  children.get(i + 1).insert(key, tree);  break;  }  }  }  }  } |

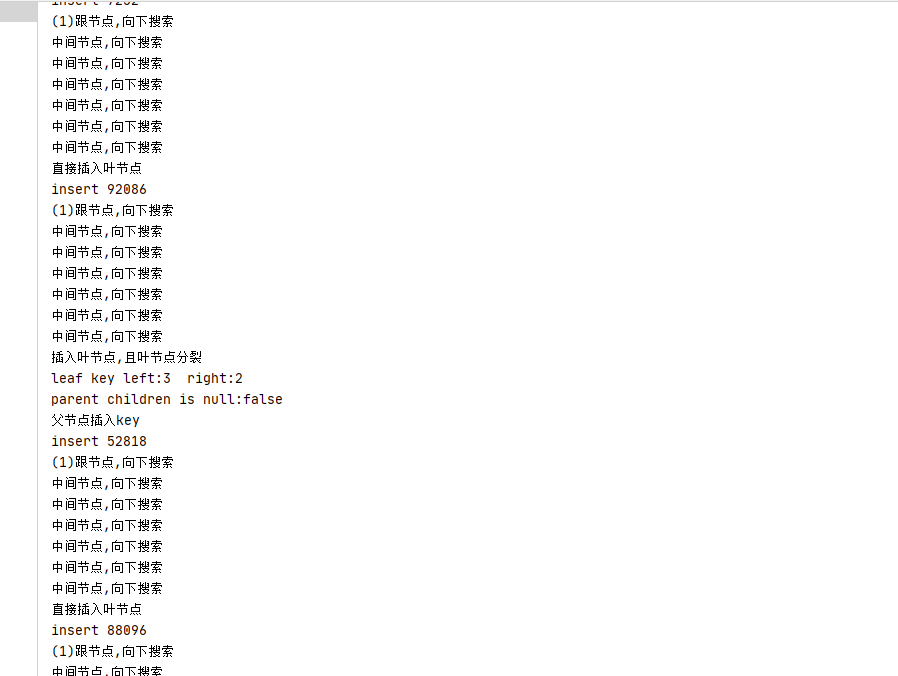
B+树删除：

|  |
| --- |
| @Override public boolean remove(Tuple key) {  return root.remove(key, this); }  public boolean remove(Tuple key, BpTree tree) {  boolean isFound = false;  if (isLeaf) {  *// 如果是叶子节点* if (!contains(key)) {  *// 不包含关键字* return false;  }  *// 是 叶子节点 且是 根节点,直接删除* if (isRoot) {  if (removeInLeaf(key)) {  isFound = true;  }  } else {  if (canRemoveDirectInLeaf()) {  *// 可以在叶节点中直接删除* if (removeInLeaf(key)) {  isFound = true;  }  } else {  *// 如果当前关键字不够,并且前节点有足够的关键字,从前节点借* if (leafCanBorrow(previous)) {  if (removeInLeaf(key)) {  borrowLeafPrevious();  isFound = true;  }  } else if (leafCanBorrow(next)) {  if (removeInLeaf(key)) {  borrowLeafNext();  isFound = true;  }  *// 从后兄弟节点借* } else {  *// 合并叶子节点, 先合并后删除* BpNode tmpParent = this.parent;  if (leafCanMerge(previous)) {  *// 和前叶子节点合并* mergeToPreLeaf(this.previous, this);  if (previous.removeInLeaf(key)) {  isFound = true;  }  *// 删除在父节点中的key* int parentKeyIdx = getMiddleKeyIdxInParent(this);  parent.entries.remove(parentKeyIdx);  *// 删除在父节点中的指针* parent.children.remove(this);  *// for GC* parent = null;  entries = null;  *// 更新 叶节点链表* if (this.next != null) {  BpNode tmp = this;  tmp.previous.next = tmp.next;  tmp.next.previous = tmp.previous;  tmp.previous = null;  tmp.next = null;  } else {  this.previous.next = null;  this.previous = null;  }  } else if (leafCanMerge(next)) {  *// 和后叶子节点合并* mergeToPreLeaf(this, this.next);  if (removeInLeaf(key)) {  isFound = true;  }  *// 删除在父节点中的key* int parentKeyIdx = getMiddleKeyIdxInParent(this.next);  parent.entries.remove(parentKeyIdx);  *// 删除在父节点中的指针* parent.children.remove(this.next);  *// for GC* next.parent = null;  next.entries = null;  *// 更新 叶节点链表* if (this.next.next != null) {  BpNode tmp = this.next;  this.next = tmp.next;  tmp.next.previous = this;  tmp.previous = null;  tmp.next = null;  } else {  this.next.previous = null;  this.next = null;  }  }  tmpParent.updateRemove(tree);  }  }  }  } else {  *// 非叶子节点,继续向下搜索* if (key.compare(entries.get(0)) < 0) {  if (children.get(0).remove(key, tree)) {  isFound = true;  }  } else if (key.compare(entries.get(entries.size() - 1)) >= 0) {  if (children.get(children.size() - 1).remove(key, tree)) {  isFound = true;  }  } else {  for (int i = 0; i < (entries.size() - 1); i++) {  if (key.compare(entries.get(i)) >= 0 && key.compare(entries.get(i + 1)) < 0) {  if (children.get(i + 1).remove(key, tree)) {  isFound = true;  }  }  break;  }  }  }  return isFound; } |

测试代码1：

|  |
| --- |
| @Test public void testRandom() {  BpTree tree = new BpTree();  int n = 100000;  Random random = new Random();  int i;  for (i = 0; i < n; i++) {  int x = random.nextInt(n);  Tuple tuple = *createTuple*(x);  tree.insert(tuple);  boolean isValid = tree.validate();  if (!isValid) {  break;  }  }  System.*out*.println(i); }  //插入一万次 |

结果：



测试代码2：

|  |
| --- |
| @Test public void test1() {  BpTree bpTree = new BpTree();  for (int i = 1; i <= 20; i++) {  bpTree.insert(*createTuple*(i));  }  bpTree.insert(*createTuple*(21));  *printLink*(bpTree.getHead());  BpNode root = bpTree.getRoot();  *printNode*(root.getChildren().get(0));  *printNode*(root.getChildren().get(1));   System.*out*.println("root=" + bpTree.getRoot().getEntries().get(0).getValues()[0]);  boolean isValid = bpTree.validate();  System.*out*.println("isValid:" + isValid);   Tuple key = bpTree.find(*createTuple*(16));  System.*out*.println("key:" + key.getValues()[0]); } |

结果：

|  |
| --- |
| "C:\Program Files\Java\jdk-15.0.1\bin\java.exe" -ea -Didea.test.cyclic.buffer.size=1048576 "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.1\lib\idea\_rt.jar=7280:C:\Program Files\JetBrains\IntelliJ IDEA 2020.1\bin" -Dfile.encoding=UTF-8 -classpath "C:\Program Files\JetBrains\IntelliJ IDEA 2020.1\lib\idea\_rt.jar;C:\Program Files\JetBrains\IntelliJ IDEA 2020.1\plugins\junit\lib\junit5-rt.jar;C:\Program Files\JetBrains\IntelliJ IDEA 2020.1\plugins\junit\lib\junit-rt.jar;F:\DataBasesShuoyin\bplus-tree\target\test-classes;F:\DataBasesShuoyin\bplus-tree\target\classes;C:\Users\123\.m2\repository\junit\junit\4.12\junit-4.12.jar;C:\Users\123\.m2\repository\org\hamcrest\hamcrest-core\1.3\hamcrest-core-1.3.jar" com.intellij.rt.junit.JUnitStarter -ideVersion5 -junit4 BpTreeTest,test1  insert 1  直接插入叶节点  insert 2  直接插入叶节点  insert 3  直接插入叶节点  insert 4  直接插入叶节点  insert 5  插入叶节点,且叶节点分裂  leaf key left:3 right:2  生成新的根节点  insert 6  (1)跟节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 7  (1)跟节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 8  (1)跟节点,向下搜索  中间节点,向下搜索  插入叶节点,且叶节点分裂  leaf key left:3 right:2  parent children is null:false  父节点插入key  insert 9  (1)跟节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 10  (1)跟节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 11  (1)跟节点,向下搜索  中间节点,向下搜索  插入叶节点,且叶节点分裂  leaf key left:3 right:2  parent children is null:false  父节点插入key  insert 12  (1)跟节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 13  (1)跟节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 14  (1)跟节点,向下搜索  中间节点,向下搜索  插入叶节点,且叶节点分裂  leaf key left:3 right:2  parent children is null:false  父节点插入key  insert 15  (1)跟节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 16  (1)跟节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 17  (1)跟节点,向下搜索  中间节点,向下搜索  插入叶节点,且叶节点分裂  leaf key left:3 right:2  parent children is null:false  父节点插入key  非叶节点插入关键字后,需要分裂  middle node p left:3 right:3  current is root:true  parent null:true  insert 18  (1)跟节点,向下搜索  中间节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 19  (1)跟节点,向下搜索  中间节点,向下搜索  中间节点,向下搜索  直接插入叶节点  insert 20  (1)跟节点,向下搜索  中间节点,向下搜索  中间节点,向下搜索  插入叶节点,且叶节点分裂  leaf key left:3 right:2  parent children is null:false  父节点插入key  insert 21  (1)跟节点,向下搜索  中间节点,向下搜索  中间节点,向下搜索  直接插入叶节点  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  4  7  13  16  19  root=10  isValid:true  key:16  Process finished with exit code 0 |