# B+树原理以及Java代码实现

最初查找[二叉树](https://so.csdn.net/so/search?q=%E4%BA%8C%E5%8F%89%E6%A0%91&spm=1001.2101.3001.7020)，由于树的高度会随着有序序列输入而急剧增长，后来出现平衡二叉树，红黑树。B树可以海量数据的快速查询检索，B树主要分为B树（B-树），B+树，B\*树等。

**B树（B-树）**

M路搜索树，参数M定义节点的分支个数；

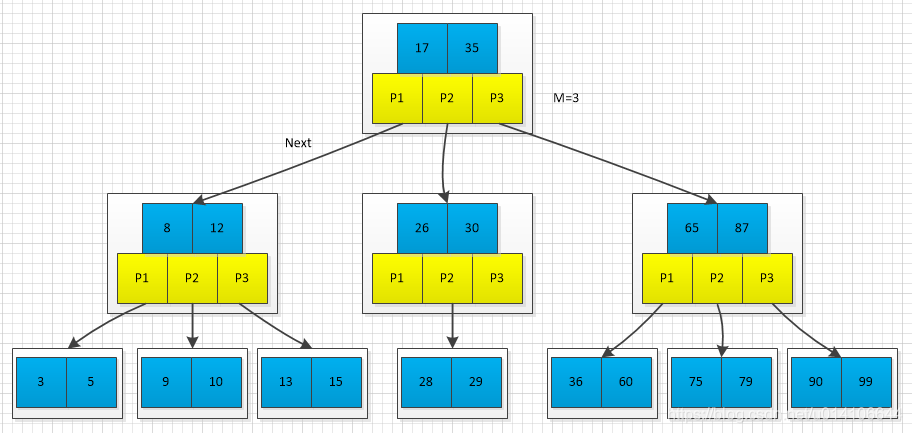
对于根节点孩子数目为[2,M]，对于其余节点孩子数目为[M/2,M];

每个节点含有关键字属性，至少M/2-1  至少M-1;关键字个数=孩子个数-1；

节点有两种类型：

叶子节点    处于同一层

非叶子结点   关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；即关键字时有序的；孩子指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于**(K[i-1], K[i])**的子树；



关键字集合分布在整颗树中；任何一个关键字出现且只出现在一个结点中；搜索有可能在非叶子结点结束；其搜索性能等价于在关键字全集内做一次二分查找；

**B+树**

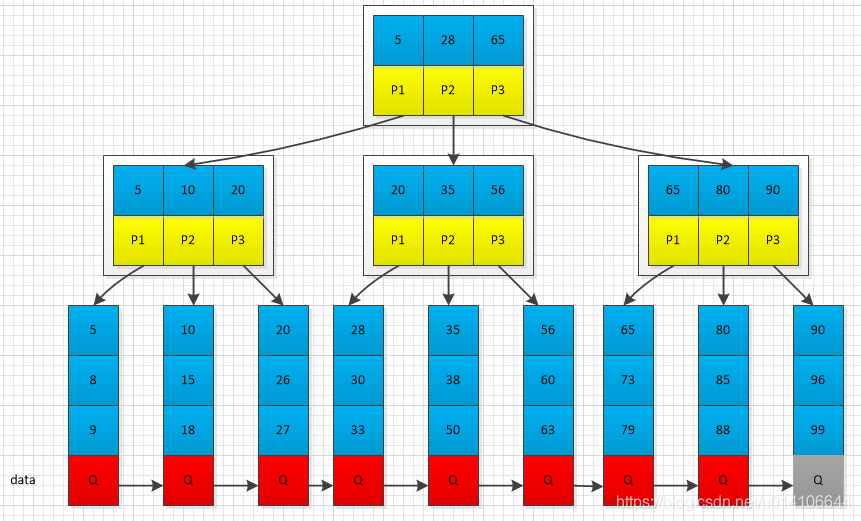
对于B树的改进，每个节点具有关键字以及孩子指针属性：

非叶子结点的子树指针与关键字个数相同；

非叶子结点的子树指针P[i]，指向关键字值属于**[K[i], K[i+1])**的子树（B-树是开区间）；

为所有叶子结点增加一个链指针；

所有关键字都在叶子结点出现；

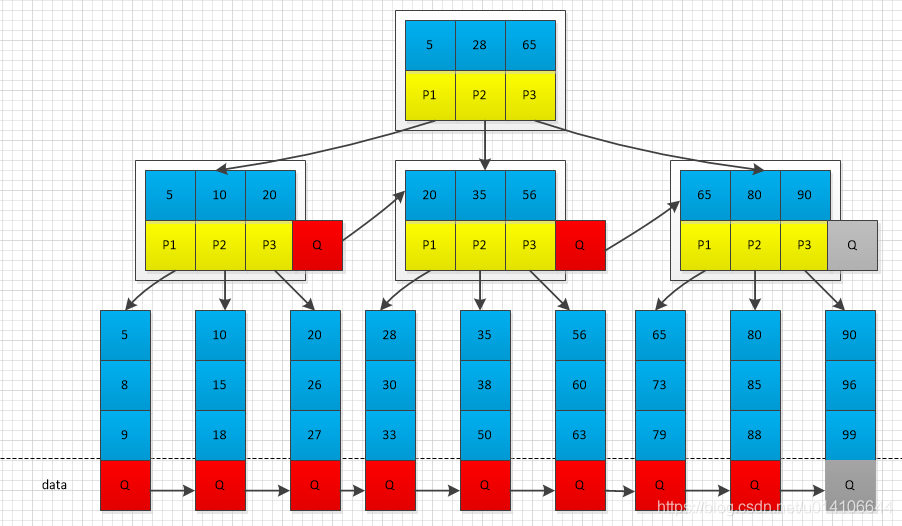


所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；不可能在非叶子结点命中；非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；更适合文件索引系统；

**B\*树**

B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针；

B\*树定义非叶子结点关键字个数至少为(2/3)\*M，即块的最低使用率为2/3（代替B+树的1/2）；



B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

B\*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针；B\*树分配新结点的概率比B+树要低，空间使用率更高；

**B+树Java代码实现**

实现B+的构造，插入以及查询方法

**定义B+树节点关键字以及孩子指针数据结构**

定义B+树参数M

对于每个节点  孩子指针利用Entry[M]来表示，m表示关键字的个数；Entry表示具体的关键字信息  Key  Value以及Next指针

1. public class BTree<Key extends Comparable<Key>, Value> {
3. //参数M 定义每个节点的链接数
4. private static final int M = 4;
6. private Node root;
8. //树的高度 最底层为0 表示外部节点 根具有最大高度
9. private int height;
11. //键值对总数
12. private int n;
14. //节点数据结构定义
15. private static final class Node{
16. //值的数量
17. private int m;
18. private Entry[] children = new Entry[M];
19. private Node(int k){
20. m = k;
21. }
22. }
24. //节点内部每个数据项定义
25. private static class Entry{
26. private Comparable key;
27. private final Object val;
28. //下一个节点
29. private Node next;
30. public Entry(Comparable key, Object val, Node next){
31. this.key = key;
32. this.val = val;
33. this.next = next;
34. }
35. @Override
36. public String toString() {
37. StringBuilder builder = new StringBuilder();
38. builder.append("Entry [key=");
39. builder.append(key);
40. builder.append("]");
41. return builder.toString();
42. }
44. }



**查询方法**

根据B+树定义，非叶子结点存储索引信息，叶子结点存储数据信息，利用height来判断当前是哪一层，如果height=0说明，当前节点为叶子结点，对于叶子结点，则执行顺序查找即可；如果当前节点为非叶子节点，则查找key在当前节点的哪一个孩子指针中，如果key大于当前节点所有关键字即j==m-1     如果key小于某个关键字j+1  则说明key位于j孩子指针中。

1. public Value get(Key key){
2. return search(root, key, height);
3. }
5. private Value search(Node x, Key key, int ht) {
6. Entry[] children = x.children;
7. //外部节点 这里使用顺序查找
8. //如果M很大 可以改成二分查找
9. if(ht == 0){
10. for(int j=0; j<x.m; j++){
11. if(equal(key, x.children[j].key))
12. return (Value)children[j].val;
13. }
14. }
15. //内部节点 寻找下一个节点
16. else{
17. for(int j=0; j<x.m; j++){
18. //最后一个节点 或者 插入值小于某个孩子节点值
19. if(j+1==x.m || less(key, x.children[j+1].key))
20. return search(x.children[j].next, key, ht-1);
21. }
22. }
23. return null;
24. }



**插入方法**

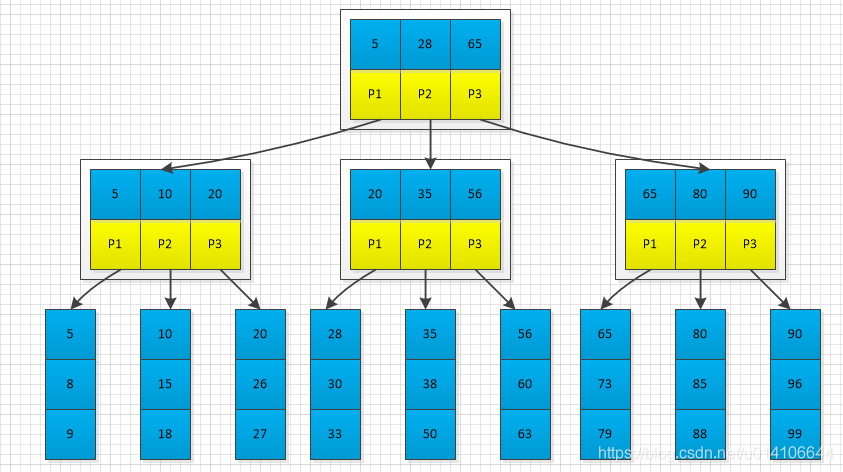
利用递归思想，insert实现在某个节点中插入数据，如果当前节点为叶子结点，则找到待插入的位置j，将j以后数据后移一位，将当前数据插入,插入之后如果当前节点已满，则将当前节点的后M/2个元素产生一个新节点返回；如果当前节点为非叶子节点，找到待插入数据位于哪一个孩子节点中，递归调用insert方法，如果返回值非空，则说明下一层插入时，出现节点分裂，将新节点指向父节点；

1. public void put(Key key, Value val){
2. //插入后的节点 如果节点分裂，则返回分裂后产生的新节点
3. Node u = insert(root, key, val, height);
4. n++;
5. //根节点没有分裂 直接返回
6. if(u == null) return;
7. //分裂根节点
8. Node t = new Node(2);
9. //旧根节点第一个孩子 新分裂节点
10. t.children[0] = new Entry(root.children[0].key, null, root);
11. t.children[1] = new Entry(u.children[0].key, null, u);
12. root = t;
13. height++;
14. }
16. private Node insert(Node h, Key key, Value val, int ht) {
17. int j;
18. //新建待插入数据数据项
19. Entry t = new Entry(key, val, null);
20. // 外部节点 找到带插入的节点位置j
21. if(ht == 0){
22. for(j=0; j<h.m; j++){
23. if(less(key, h.children[j].key))
24. break;
25. }
26. }else{
27. //内部节点 找到合适的分支节点
28. for(j=0; j<h.m; j++){
29. if(j+1==h.m || less(key, h.children[j+1].key)){
30. Node u = insert(h.children[j++].next, key, val, ht-1);
31. if(u == null) return null;
32. t.key = u.children[0].key;
33. t.next = u;
34. break;
35. }
36. }
37. }
38. //System.out.println(j + t.toString());
39. //j为带插入位置，将顺序数组j位置以后后移一位 将t插入
40. for(int i=h.m; i>j; i++){
41. h.children[i] = h.children[i-1];
42. }
43. System.out.println(j + t.toString());
44. h.children[j] = t;
45. h.m++;
46. if(h.m < M) return null;
47. //如果节点已满 则执行分类操作
48. else return split(h);
49. }
51. private Node split(Node h) {
52. //将已满节点h的后一般M/2节点分裂到新节点并返回
53. Node t = new Node(M/2);
54. h.m = M / 2;
55. for(int j=0; j<M/2; j++)
56. t.children[j] = h.children[M/2+j];
57. return t;
58. }

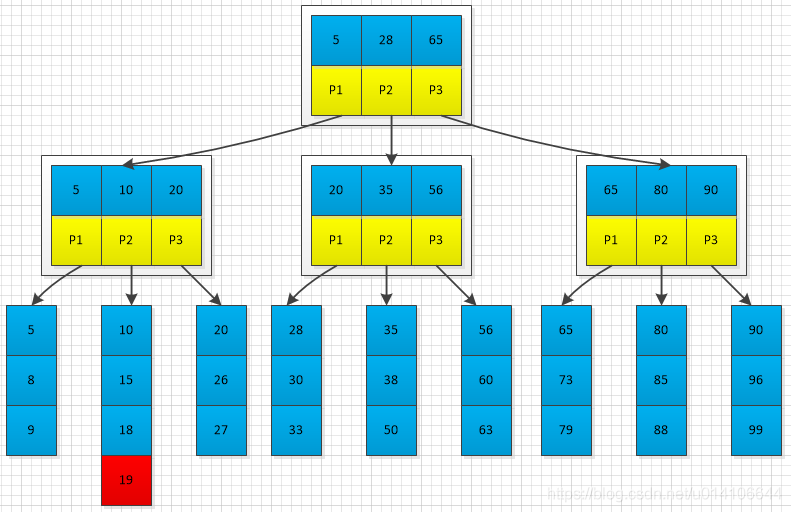


**节点分裂过程**

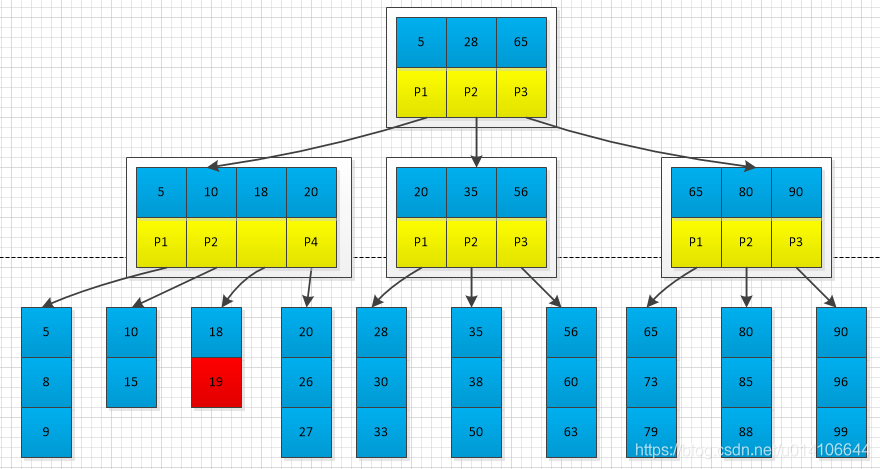
在下图M=3中插入19，首先找到19的插入位置，插入；



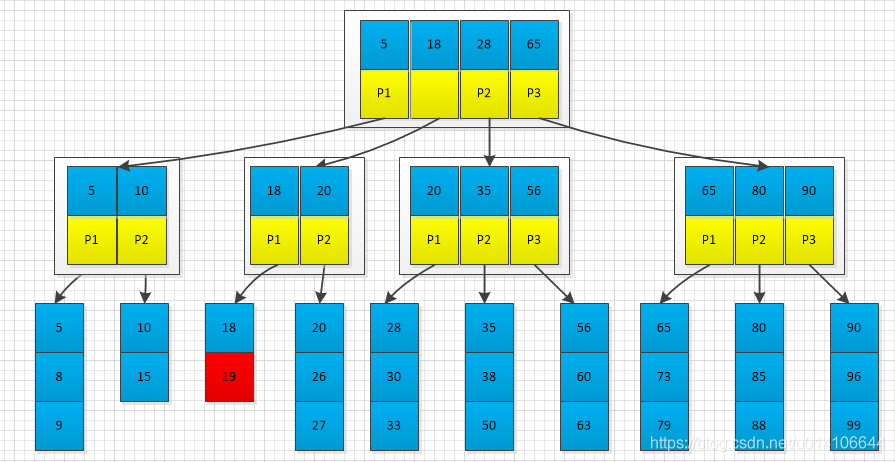
插入之后，出现节点已满



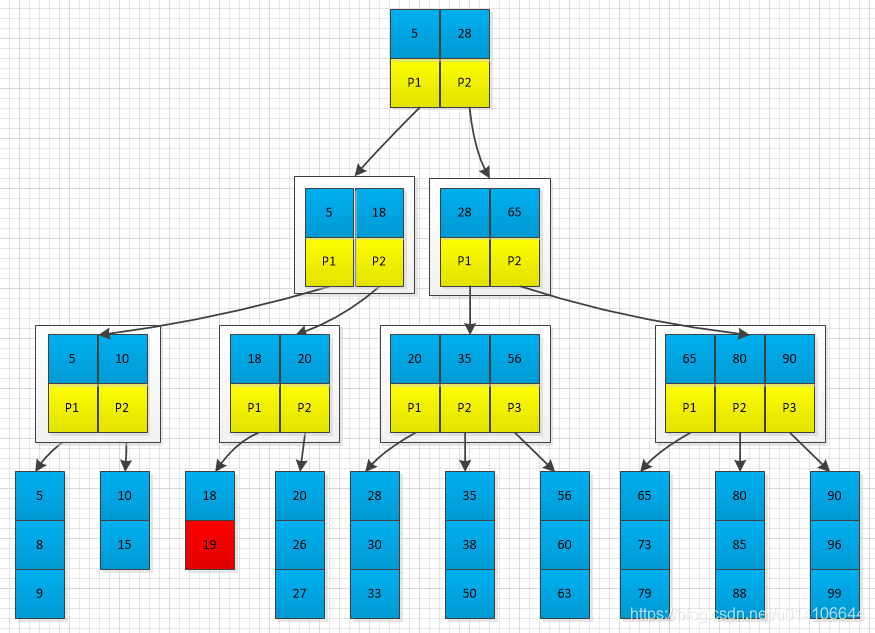
将已满节点进行分裂，将已满节点后M/2节点生成一个新节点，将新节点的第一个元素指向父节点；



父节点出现已满，将父节点继续分裂



一直分裂，如果根节点已满，则需要分类根节点，此时树的高度增加



对于根节点分裂，新建一个节点  将旧根节点第一个元素以及分类节点第一个元素组合作为新的根节点。

1. public void put(Key key, Value val){
2. //插入后的节点 如果节点分裂，则返回分裂后产生的新节点
3. Node u = insert(root, key, val, height);
4. n++;
5. //根节点没有分裂 直接返回
6. if(u == null) return;
7. //分裂根节点
8. Node t = new Node(2);
9. //旧根节点第一个孩子 新分裂节点第一个孩子组成新节点作为根
10. t.children[0] = new Entry(root.children[0].key, null, root);
11. t.children[1] = new Entry(u.children[0].key, null, u);
12. root = t;
13. height++;
14. }

性能：

含有N个元素的M阶B+树，一次查找或者插入的时间复杂度为log(M)N~log(M/2)N，当M较大时，该值基本为常数；

代码位置：<https://github.com/ChenWenKaiVN/TreeSWT/blob/master/src/com/tree/bt/BTree.java>

参考链接：

<https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/BTree.java.html>

<https://www.2cto.com/net/201808/773535.html>

<https://blog.csdn.net/qq_35008624/article/details/81947773>

<https://www.cnblogs.com/vincently/p/4526560.html>

# [B+树Java代码实现以及测试](https://www.cnblogs.com/jing99/p/11741685.html)

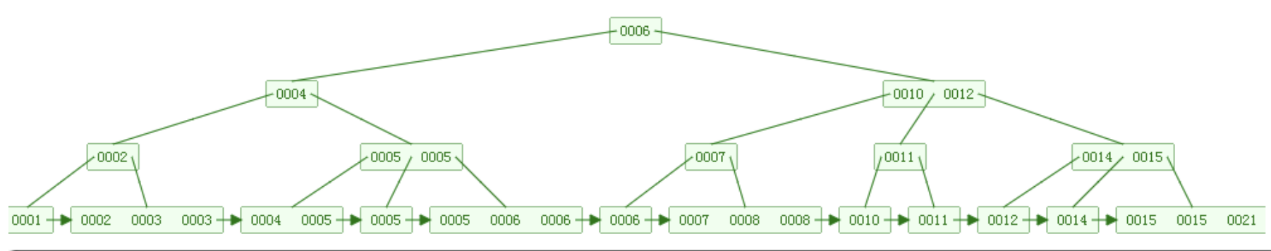
M阶B+树的定义：

1. 任意非叶子结点最多有M个子节点；且M>2；
2. 除根结点以外的非叶子结点至少有 M/2个子节点；
3. 根结点至少有2个子节点；
4. 除根节点外每个结点存放至少M/2和至多M个关键字；（至少2个关键字）
5. 非叶子结点的子树指针与关键字个数相同；
6. 所有结点的关键字：K[1], K[2], …, K[M]；且K[i] < K[i+1]；
7. 非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树；
8. 所有叶子结点位于同一层；
9. 为所有叶子结点增加一个链指针；
10. 所有关键字都在叶子结点出现；

**B+**树是对B树的一种变形树，它与B树的差异在于：

* 有k个子结点的结点必然有k个关键码；
* 非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中。
* 树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

　　如下图，是一个B+树:



 　　下图是B+树的插入动画：



代码如下：

　　BPlusTree定义如下：

[复制代码](javascript:void(0);)

/\*\*

\* B+树的定义：

\* 1.任意非叶子结点最多有M个子节点；且M>2；M为B+树的阶数

\* 2.除根结点以外的非叶子结点至少有 (M+1)/2个子节点；

\* 3.根结点至少有2个子节点；

\* 4.除根节点外每个结点存放至少（M-1）/2和至多M-1个关键字；（至少1个关键字）

\* 5.非叶子结点的子树指针比关键字多1个；

\* 6.非叶子节点的所有key按升序存放，假设节点的关键字分别为K[0], K[1] … K[M-2],指向子女的指针分别为P[0], P[1]…P[M-1]。则有：

\* 　　　P[0] < K[0] <= P[1] < K[1] …..< K[M-2] <= P[M-1]

\* 7.所有叶子结点位于同一层；

\* 8.为所有叶子结点增加一个链指针；

\* 9.所有关键字都在叶子结点出现

\*/

@SuppressWarnings("all")

public class BPlusTree<K extends Comparable<K>, V> {

// 根节点

protected BPlusNode<K, V> root;

// 阶数，M值

protected int order;

// 叶子节点的链表头

protected BPlusNode<K, V> head;

// 树高

protected int height = 0;

public BPlusNode<K, V> getHead() {

return head;

}

public void setHead(BPlusNode<K, V> head) {

this.head = head;

}

public BPlusNode<K, V> getRoot() {

return root;

}

public void setRoot(BPlusNode<K, V> root) {

this.root = root;

}

public int getOrder() {

return order;

}

public void setOrder(int order) {

this.order = order;

}

public void setHeight(int height) {

this.height = height;

}

public int getHeight() {

return height;

}

public V get(K key) {

return root.get(key);

}

public V remove(K key) {

return root.remove(key, this);

}

public void insertOrUpdate(K key, V value) {

root.insertOrUpdate(key, value, this);

}

public BPlusTree(int order) {

if (order < 3) {

System.out.print("order must be greater than 2");

System.exit(0);

}

this.order = order;

root = new BPlusNode<K, V>(true, true);

head = root;

}

public void printBPlusTree() {

this.root.printBPlusTree(0);

}

}

[复制代码](javascript:void(0);)

　　BplusNode定义如下：

[复制代码](javascript:void(0);)

@SuppressWarnings("all")

public class BPlusNode<K extends Comparable<K>, V> {

// 是否为叶子节点

protected boolean isLeaf;

// 是否为根节点

protected boolean isRoot;

// 父节点

protected BPlusNode<K, V> parent;

// 叶节点的前节点

protected BPlusNode<K, V> previous;

// 叶节点的后节点

protected BPlusNode<K, V> next;

// 节点的关键字列表

protected List<Entry<K, V>> entries;

// 子节点列表

protected List<BPlusNode<K, V>> children;

public BPlusNode(boolean isLeaf) {

this.isLeaf = isLeaf;

entries = new ArrayList();

if (!isLeaf) {

children = new ArrayList();

}

}

public BPlusNode(boolean isLeaf, boolean isRoot) {

this(isLeaf);

this.isRoot = isRoot;

}

public V get(K key) {

//如果是叶子节点

if (isLeaf) {

int low = 0, high = entries.size() - 1, mid;

int comp;

while (low <= high) {

mid = (low + high) / 2;

comp = entries.get(mid).getKey().compareTo(key);

if (comp == 0) {

return entries.get(mid).getValue();

} else if (comp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

//未找到所要查询的对象

return null;

}

//如果不是叶子节点

//如果key小于节点最左边的key，沿第一个子节点继续搜索

if (key.compareTo(entries.get(0).getKey()) < 0) {

return children.get(0).get(key);

//如果key大于等于节点最右边的key，沿最后一个子节点继续搜索

} else if (key.compareTo(entries.get(entries.size() - 1).getKey()) >= 0) {

return children.get(children.size() - 1).get(key);

//否则沿比key大的前一个子节点继续搜索

} else {

int low = 0, high = entries.size() - 1, mid = 0;

int comp;

while (low <= high) {

mid = (low + high) / 2;

comp = entries.get(mid).getKey().compareTo(key);

if (comp == 0) {

return children.get(mid + 1).get(key);

} else if (comp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return children.get(low).get(key);

}

}

public void insertOrUpdate(K key, V value, BPlusTree<K, V> tree) {

//如果是叶子节点

if (isLeaf) {

//不需要分裂，直接插入或更新

if (contains(key) != -1 || entries.size() < tree.getOrder()) {

insertOrUpdate(key, value);

if (tree.getHeight() == 0) {

tree.setHeight(1);

}

return;

}

//需要分裂

//分裂成左右两个节点

BPlusNode<K, V> left = new BPlusNode<K, V>(true);

BPlusNode<K, V> right = new BPlusNode<K, V>(true);

//设置链接

if (previous != null) {

previous.next = left;

left.previous = previous;

}

if (next != null) {

next.previous = right;

right.next = next;

}

if (previous == null) {

tree.setHead(left);

}

left.next = right;

right.previous = left;

previous = null;

next = null;

//复制原节点关键字到分裂出来的新节点

copy2Nodes(key, value, left, right, tree);

//如果不是根节点

if (parent != null) {

//调整父子节点关系

int index = parent.children.indexOf(this);

parent.children.remove(this);

left.parent = parent;

right.parent = parent;

parent.children.add(index, left);

parent.children.add(index + 1, right);

parent.entries.add(index, right.entries.get(0));

entries = null; //删除当前节点的关键字信息

children = null; //删除当前节点的孩子节点引用

//父节点插入或更新关键字

parent.updateInsert(tree);

parent = null; //删除当前节点的父节点引用

//如果是根节点

} else {

isRoot = false;

BPlusNode<K, V> parent = new BPlusNode<K, V>(false, true);

tree.setRoot(parent);

left.parent = parent;

right.parent = parent;

parent.children.add(left);

parent.children.add(right);

parent.entries.add(right.entries.get(0));

entries = null;

children = null;

}

return;

}

//如果不是叶子节点

//如果key小于等于节点最左边的key，沿第一个子节点继续搜索

if (key.compareTo(entries.get(0).getKey()) < 0) {

children.get(0).insertOrUpdate(key, value, tree);

//如果key大于节点最右边的key，沿最后一个子节点继续搜索

} else if (key.compareTo(entries.get(entries.size() - 1).getKey()) >= 0) {

children.get(children.size() - 1).insertOrUpdate(key, value, tree);

//否则沿比key大的前一个子节点继续搜索

} else {

int low = 0, high = entries.size() - 1, mid = 0;

int comp;

while (low <= high) {

mid = (low + high) / 2;

comp = entries.get(mid).getKey().compareTo(key);

if (comp == 0) {

children.get(mid + 1).insertOrUpdate(key, value, tree);

break;

} else if (comp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

if (low > high) {

children.get(low).insertOrUpdate(key, value, tree);

}

}

}

private void copy2Nodes(K key, V value, BPlusNode<K, V> left,

BPlusNode<K, V> right, BPlusTree<K, V> tree) {

//左右两个节点关键字长度

int leftSize = (tree.getOrder() + 1) / 2 + (tree.getOrder() + 1) % 2;

boolean b = false;//用于记录新元素是否已经被插入

for (int i = 0; i < entries.size(); i++) {

if (leftSize != 0) {

leftSize--;

if (!b && entries.get(i).getKey().compareTo(key) > 0) {

left.entries.add(new SimpleEntry<K, V>(key, value));

b = true;

i--;

} else {

left.entries.add(entries.get(i));

}

} else {

if (!b && entries.get(i).getKey().compareTo(key) > 0) {

right.entries.add(new SimpleEntry<K, V>(key, value));

b = true;

i--;

} else {

right.entries.add(entries.get(i));

}

}

}

if (!b) {

right.entries.add(new SimpleEntry<K, V>(key, value));

}

}

/\*\*

\* 插入节点后中间节点的更新

\*/

protected void updateInsert(BPlusTree<K, V> tree) {

//如果子节点数超出阶数，则需要分裂该节点

if (children.size() > tree.getOrder()) {

//分裂成左右两个节点

BPlusNode<K, V> left = new BPlusNode<K, V>(false);

BPlusNode<K, V> right = new BPlusNode<K, V>(false);

//左右两个节点子节点的长度

int leftSize = (tree.getOrder() + 1) / 2 + (tree.getOrder() + 1) % 2;

int rightSize = (tree.getOrder() + 1) / 2;

//复制子节点到分裂出来的新节点，并更新关键字

for (int i = 0; i < leftSize; i++) {

left.children.add(children.get(i));

children.get(i).parent = left;

}

for (int i = 0; i < rightSize; i++) {

right.children.add(children.get(leftSize + i));

children.get(leftSize + i).parent = right;

}

for (int i = 0; i < leftSize - 1; i++) {

left.entries.add(entries.get(i));

}

for (int i = 0; i < rightSize - 1; i++) {

right.entries.add(entries.get(leftSize + i));

}

//如果不是根节点

if (parent != null) {

//调整父子节点关系

int index = parent.children.indexOf(this);

parent.children.remove(this);

left.parent = parent;

right.parent = parent;

parent.children.add(index, left);

parent.children.add(index + 1, right);

parent.entries.add(index, entries.get(leftSize - 1));

entries = null;

children = null;

//父节点更新关键字

parent.updateInsert(tree);

parent = null;

//如果是根节点

} else {

isRoot = false;

BPlusNode<K, V> parent = new BPlusNode<K, V>(false, true);

tree.setRoot(parent);

tree.setHeight(tree.getHeight() + 1);

left.parent = parent;

right.parent = parent;

parent.children.add(left);

parent.children.add(right);

parent.entries.add(entries.get(leftSize - 1));

entries = null;

children = null;

}

}

}

/\*\*

\* 删除节点后中间节点的更新

\*/

protected void updateRemove(BPlusTree<K, V> tree) {

// 如果子节点数小于M / 2或者小于2，则需要合并节点

if (children.size() < tree.getOrder() / 2 || children.size() < 2) {

if (isRoot) {

// 如果是根节点并且子节点数大于等于2，OK

if (children.size() >= 2) return;

// 否则与子节点合并

BPlusNode<K, V> root = children.get(0);

tree.setRoot(root);

tree.setHeight(tree.getHeight() - 1);

root.parent = null;

root.isRoot = true;

entries = null;

children = null;

return;

}

//计算前后节点

int currIdx = parent.children.indexOf(this);

int prevIdx = currIdx - 1;

int nextIdx = currIdx + 1;

BPlusNode<K, V> previous = null, next = null;

if (prevIdx >= 0) {

previous = parent.children.get(prevIdx);

}

if (nextIdx < parent.children.size()) {

next = parent.children.get(nextIdx);

}

// 如果前节点子节点数大于M / 2并且大于2，则从其处借补

if (previous != null

&& previous.children.size() > tree.getOrder() / 2

&& previous.children.size() > 2) {

//前叶子节点末尾节点添加到首位

int idx = previous.children.size() - 1;

BPlusNode<K, V> borrow = previous.children.get(idx);

previous.children.remove(idx);

borrow.parent = this;

children.add(0, borrow);

int preIndex = parent.children.indexOf(previous);

entries.add(0, parent.entries.get(preIndex));

parent.entries.set(preIndex, previous.entries.remove(idx - 1));

return;

}

// 如果后节点子节点数大于M / 2并且大于2，则从其处借补

if (next != null

&& next.children.size() > tree.getOrder() / 2

&& next.children.size() > 2) {

//后叶子节点首位添加到末尾

BPlusNode<K, V> borrow = next.children.get(0);

next.children.remove(0);

borrow.parent = this;

children.add(borrow);

int preIndex = parent.children.indexOf(this);

entries.add(parent.entries.get(preIndex));

parent.entries.set(preIndex, next.entries.remove(0));

return;

}

// 同前面节点合并

if (previous != null

&& (previous.children.size() <= tree.getOrder() / 2

|| previous.children.size() <= 2)) {

for (int i = 0; i < children.size(); i++) {

previous.children.add(children.get(i));

}

for (int i = 0; i < previous.children.size(); i++) {

previous.children.get(i).parent = this;

}

int indexPre = parent.children.indexOf(previous);

previous.entries.add(parent.entries.get(indexPre));

for (int i = 0; i < entries.size(); i++) {

previous.entries.add(entries.get(i));

}

children = previous.children;

entries = previous.entries;

//更新父节点的关键字列表

parent.children.remove(previous);

previous.parent = null;

previous.children = null;

previous.entries = null;

parent.entries.remove(parent.children.indexOf(this));

if ((!parent.isRoot

&& (parent.children.size() >= tree.getOrder() / 2

&& parent.children.size() >= 2))

|| parent.isRoot && parent.children.size() >= 2) {

return;

}

parent.updateRemove(tree);

return;

}

// 同后面节点合并

if (next != null

&& (next.children.size() <= tree.getOrder() / 2

|| next.children.size() <= 2)) {

for (int i = 0; i < next.children.size(); i++) {

BPlusNode<K, V> child = next.children.get(i);

children.add(child);

child.parent = this;

}

int index = parent.children.indexOf(this);

entries.add(parent.entries.get(index));

for (int i = 0; i < next.entries.size(); i++) {

entries.add(next.entries.get(i));

}

parent.children.remove(next);

next.parent = null;

next.children = null;

next.entries = null;

parent.entries.remove(parent.children.indexOf(this));

if ((!parent.isRoot && (parent.children.size() >= tree.getOrder() / 2

&& parent.children.size() >= 2))

|| parent.isRoot && parent.children.size() >= 2) {

return;

}

parent.updateRemove(tree);

return;

}

}

}

public V remove(K key, BPlusTree<K, V> tree) {

//如果是叶子节点

if (isLeaf) {

//如果不包含该关键字，则直接返回

if (contains(key) == -1) {

return null;

}

//如果既是叶子节点又是根节点，直接删除

if (isRoot) {

if (entries.size() == 1) {

tree.setHeight(0);

}

return remove(key);

}

//如果关键字数大于M / 2，直接删除

if (entries.size() > tree.getOrder() / 2 && entries.size() > 2) {

return remove(key);

}

//如果自身关键字数小于M / 2，并且前节点关键字数大于M / 2，则从其处借补

if (previous != null &&

previous.parent == parent

&& previous.entries.size() > tree.getOrder() / 2

&& previous.entries.size() > 2) {

//添加到首位

int size = previous.entries.size();

entries.add(0, previous.entries.remove(size - 1));

int index = parent.children.indexOf(previous);

parent.entries.set(index, entries.get(0));

return remove(key);

}

//如果自身关键字数小于M / 2，并且后节点关键字数大于M / 2，则从其处借补

if (next != null

&& next.parent == parent

&& next.entries.size() > tree.getOrder() / 2

&& next.entries.size() > 2) {

entries.add(next.entries.remove(0));

int index = parent.children.indexOf(this);

parent.entries.set(index, next.entries.get(0));

return remove(key);

}

//同前面节点合并

if (previous != null

&& previous.parent == parent

&& (previous.entries.size() <= tree.getOrder() / 2

|| previous.entries.size() <= 2)) {

V returnValue = remove(key);

for (int i = 0; i < entries.size(); i++) {

//将当前节点的关键字添加到前节点的末尾

previous.entries.add(entries.get(i));

}

entries = previous.entries;

parent.children.remove(previous);

previous.parent = null;

previous.entries = null;

//更新链表

if (previous.previous != null) {

BPlusNode<K, V> temp = previous;

temp.previous.next = this;

previous = temp.previous;

temp.previous = null;

temp.next = null;

} else {

tree.setHead(this);

previous.next = null;

previous = null;

}

parent.entries.remove(parent.children.indexOf(this));

if ((!parent.isRoot && (parent.children.size() >= tree.getOrder() / 2

&& parent.children.size() >= 2))

|| parent.isRoot && parent.children.size() >= 2) {

return returnValue;

}

parent.updateRemove(tree);

return returnValue;

}

//同后面节点合并

if (next != null

&& next.parent == parent

&& (next.entries.size() <= tree.getOrder() / 2

|| next.entries.size() <= 2)) {

V returnValue = remove(key);

for (int i = 0; i < next.entries.size(); i++) {

//从首位开始添加到末尾

entries.add(next.entries.get(i));

}

next.parent = null;

next.entries = null;

parent.children.remove(next);

//更新链表

if (next.next != null) {

BPlusNode<K, V> temp = next;

temp.next.previous = this;

next = temp.next;

temp.previous = null;

temp.next = null;

} else {

next.previous = null;

next = null;

}

//更新父节点的关键字列表

parent.entries.remove(parent.children.indexOf(this));

if ((!parent.isRoot && (parent.children.size() >= tree.getOrder() / 2

&& parent.children.size() >= 2))

|| parent.isRoot && parent.children.size() >= 2) {

return returnValue;

}

parent.updateRemove(tree);

return returnValue;

}

}

/\*如果不是叶子节点\*/

//如果key小于等于节点最左边的key，沿第一个子节点继续搜索

if (key.compareTo(entries.get(0).getKey()) < 0) {

return children.get(0).remove(key, tree);

//如果key大于节点最右边的key，沿最后一个子节点继续搜索

} else if (key.compareTo(entries.get(entries.size() - 1).getKey()) >= 0) {

return children.get(children.size() - 1).remove(key, tree);

//否则沿比key大的前一个子节点继续搜索

} else {

int low = 0, high = entries.size() - 1, mid = 0;

int comp;

while (low <= high) {

mid = (low + high) / 2;

comp = entries.get(mid).getKey().compareTo(key);

if (comp == 0) {

return children.get(mid + 1).remove(key, tree);

} else if (comp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return children.get(low).remove(key, tree);

}

}

// 判断当前节点是否包含该关键字

protected int contains(K key) {

int low = 0, high = entries.size() - 1, mid;

int comp;

while (low <= high) {

mid = (low + high) / 2;

comp = entries.get(mid).getKey().compareTo(key);

if (comp == 0) {

return mid;

} else if (comp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return -1;

}

// 插入到当前节点的关键字中

protected void insertOrUpdate(K key, V value) {

//二叉查找，插入

int low = 0, high = entries.size() - 1, mid;

int comp;

while (low <= high) {

mid = (low + high) / 2;

comp = entries.get(mid).getKey().compareTo(key);

if (comp == 0) {

entries.get(mid).setValue(value);

break;

} else if (comp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

if (low > high) {

entries.add(low, new SimpleEntry<K, V>(key, value));

}

}

// 删除节点

protected V remove(K key) {

int low = 0, high = entries.size() - 1, mid;

int comp;

while (low <= high) {

mid = (low + high) / 2;

comp = entries.get(mid).getKey().compareTo(key);

if (comp == 0) {

return entries.remove(mid).getValue();

} else if (comp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("isRoot: ");

sb.append(isRoot);

sb.append(", ");

sb.append("isLeaf: ");

sb.append(isLeaf);

sb.append(", ");

sb.append("keys: ");

for (Entry<K, V> entry : entries) {

sb.append(entry.getKey());

sb.append(", ");

}

sb.append(", ");

return sb.toString();

}

public void printBPlusTree(int index) {

if (this.isLeaf) {

System.out.print("层级：" + index + ",叶子节点，keys为: ");

for (int i = 0; i < entries.size(); ++i)

System.out.print(entries.get(i) + " ");

System.out.println();

} else {

System.out.print("层级：" + index + ",非叶子节点，keys为: ");

for (int i = 0; i < entries.size(); ++i)

System.out.print(entries.get(i) + " ");

System.out.println();

for (int i = 0; i < children.size(); ++i)

children.get(i).printBPlusTree(index + 1);

}

}

}

[复制代码](javascript:void(0);)

　　测试类：

[复制代码](javascript:void(0);)

public class BPlusTreeTest {

// 测试

public static void main(String[] args) {

int size = 10;

int order = 3;

testRandomInsert(size, order);

testOrderInsert(size, order);

testRandomSearch(size, order);

testOrderSearch(size, order);

testRandomRemove(size, order);

testOrderRemove(size, order);

}

private static void testOrderRemove(int size, int order) {

BPlusTree<Integer, Integer> tree = new BPlusTree<Integer, Integer>(order);

System.out.println("\nTest order remove " + size + " datas, of order:"

+ order);

System.out.println("Begin order insert...");

for (int i = 0; i < size; i++) {

tree.insertOrUpdate(i, i);

}

tree.printBPlusTree();

System.out.println("Begin order remove...");

long current = System.currentTimeMillis();

for (int j = 0; j < size; j++) {

if (tree.remove(j) == null) {

System.err.println("得不到数据:" + j);

break;

}

}

long duration = System.currentTimeMillis() - current;

System.out.println("time elpsed for duration: " + duration);

tree.printBPlusTree();

System.out.println(tree.getHeight());

}

private static void testRandomRemove(int size, int order) {

BPlusTree<Integer, Integer> tree = new BPlusTree<Integer, Integer>(order);

System.out.println("\nTest random remove " + size + " datas, of order:"

+ order);

Random random = new Random();

boolean[] a = new boolean[size + 10];

List<Integer> list = new ArrayList<Integer>();

int randomNumber = 0;

System.out.println("Begin random insert...");

for (int i = 0; i < size; i++) {

randomNumber = random.nextInt(size);

a[randomNumber] = true;

list.add(randomNumber);

tree.insertOrUpdate(randomNumber, randomNumber);

}

tree.printBPlusTree();

System.out.println("Begin random remove...");

long current = System.currentTimeMillis();

for (int j = 0; j < size; j++) {

randomNumber = list.get(j);

if (a[randomNumber]) {

if (tree.remove(randomNumber) == null) {

System.err.println("得不到数据:" + randomNumber);

break;

} else {

a[randomNumber] = false;

}

}

}

long duration = System.currentTimeMillis() - current;

System.out.println("time elpsed for duration: " + duration);

tree.printBPlusTree();

System.out.println(tree.getHeight());

}

private static void testOrderSearch(int size, int order) {

BPlusTree<Integer, Integer> tree = new BPlusTree<Integer, Integer>(order);

System.out.println("\nTest order search " + size + " datas, of order:"

+ order);

System.out.println("Begin order insert...");

for (int i = 0; i < size; i++) {

tree.insertOrUpdate(i, i);

}

tree.printBPlusTree();

System.out.println("Begin order search...");

long current = System.currentTimeMillis();

for (int j = 0; j < size; j++) {

if (tree.get(j) == null) {

System.err.println("得不到数据:" + j);

break;

}

}

long duration = System.currentTimeMillis() - current;

System.out.println("time elpsed for duration: " + duration);

}

private static void testRandomSearch(int size, int order) {

BPlusTree<Integer, Integer> tree = new BPlusTree<Integer, Integer>(order);

System.out.println("\nTest random search " + size + " datas, of order:"

+ order);

Random random = new Random();

boolean[] a = new boolean[size + 10];

int randomNumber = 0;

System.out.println("Begin random insert...");

for (int i = 0; i < size; i++) {

randomNumber = random.nextInt(size);

a[randomNumber] = true;

tree.insertOrUpdate(randomNumber, randomNumber);

}

tree.printBPlusTree();

System.out.println("Begin random search...");

long current = System.currentTimeMillis();

for (int j = 0; j < size; j++) {

randomNumber = random.nextInt(size);

if (a[randomNumber]) {

if (tree.get(randomNumber) == null) {

System.err.println("得不到数据:" + randomNumber);

break;

}

}

}

long duration = System.currentTimeMillis() - current;

System.out.println("time elpsed for duration: " + duration);

}

private static void testRandomInsert(int size, int order) {

BPlusTree<Integer, Integer> tree = new BPlusTree<Integer, Integer>(order);

System.out.println("\nTest random insert " + size + " datas, of order:"

+ order);

Random random = new Random();

int randomNumber = 0;

long current = System.currentTimeMillis();

for (int i = 0; i < size; i++) {

randomNumber = random.nextInt(size \* 10);

System.out.print(randomNumber + " ");

tree.insertOrUpdate(randomNumber, randomNumber);

}

long duration = System.currentTimeMillis() - current;

System.out.println("time elpsed for duration: " + duration);

tree.printBPlusTree();

System.out.println(tree.getHeight());

}

private static void testOrderInsert(int size, int order) {

BPlusTree<Integer, Integer> tree = new BPlusTree<Integer, Integer>(order);

System.out.println("\nTest order insert " + size + " datas, of order:"

+ order);

long current = System.currentTimeMillis();

for (int i = 0; i < size; i++) {

tree.insertOrUpdate(i, i);

}

long duration = System.currentTimeMillis() - current;

System.out.println("time elpsed for duration: " + duration);

tree.printBPlusTree();

}

}

[复制代码](javascript:void(0);)

Stay hungry，stay foolish ！