B-tree-详解

阅读更多

1 B树的定义

1.1 节点

节点的属性

1. n：关键字个数
2. key：关键字数组
3. c：孩子数组
4. leaf：是否为叶节点

每个节点具有以下性质

1. x.n：当前存储在节点x中的关键字个数
2. x.n个关键字本身$x.key_1$，$x.key_2$，…，$x.key_{x.n}$，以非降序存放，使得$x.key_1 \le x.key_2 \le \ldots \le x.key_{x.n}$
3. x.leaf：一个布尔值，如果x是叶节点，则为TRUE，如果x为内部节点，则为FALSE
4. 每个内部节点x还包含x.n+1个指向其孩子的指针，$x.c_1$，$x.c_2$，…，$x.c_{x.n+1}$，叶节点没有孩子，所以他们的c属性没有定义
5. 关键字$x.key_i$对存储在各子树中的关键字范围加以分割：如果k_i为任意一个存储在以$x.c_i$为根的子树中的关键字，那么$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \ldots \le x.key_{x.n} \le k_{x.n+1}$
6. **每个叶节点都具有相同的深度，即树的高度h**
7. **每个节点所包含的关键字个数有上界和下界，用一个被称为B数的最小度数(minimum degree)的固定整数$t \ge 2$来表示这些界**

- **除了根节点以外的每个节点必须至少有t-1个关键字，因此除了根节点以外的每个内部节点至少有t个孩子，如果树非空，根节点至少含有一个关键字**
- **每个节点至多可包含2t-1个关键字，因此，一个内部节点最多可有2t个孩子，当一个节点恰好有2t-1个关键字时，称该节点是满的**
- t=2时的B树是最简单的，在实际中，t的值越大，B树的高度就越小

1.2 树

属性

1. t：B树的度
2. root：B树的根节点

性质

1. 对于节点x ，关键字$x.key_i$与子树指针$x.c_i$的索引相同，就说$x.c_i$是关键字$x.key_i$对应的子树指针
2. 子树$x.c_i$的元素介于$x.key_{i-1}$~$x.key_i$之间 $1\le i\le x.n+1$，为保持一致性，记$x.key_0= -\infty$，$x.key_{x.n+1}=+\infty$

2 伪代码

2.1 Split

分裂给定节点，分裂操作会产生一个新的节点，该新节点会插入到父节点当中，并含有分裂前一半的关键字数量以及孩子数量(非叶子节点的分裂才需要考虑孩子)

1. 要分裂的节点必须是满节点，即关键字数目为2t-1
2. 要分裂的节点的父节点必须是非满节点

```
1  B-TREE-SPLIT-CHILD(x,i)//x.ci是满节点，x是非满节点
2  z=ALLOCATE-NODE()//z是由y的一半分裂得到
3  y=x.c[i]
4  z.leaf=y.leaf
5  z.n=t-1
6  for j=1 to t-1
7      z.key[j]=y.key[j+t] //将y中[t+1…2t-1]总共t-1个关键字复制到节点z中作为[1…t-1
8  if not y.leaf//如果y不是叶节点，那么y还有t个指针需要复制到z中
9      for j=1 to t
10         z.c[j]=y.c[j+t]
11 y.n=t-1
12 for j=x.n+1 downto i+1//指针y和z必然是相邻的，并且他们所夹的关键字就是原来y中第t个
13     x.c[j+1]=x.c[j]
14 x.c[i+1]=z
15 for j=x.n downto i
16     x.key[j+1]=x.key[j]
17 x.key[i]=y.key[t]
18 x.n=x.n+1
19 DISK-WRITE(y)
20 DISK-WRITE(z)
21 DISK-WRITE(x)
```

2.2 Merge

合并两个节点

1. 合并的两个节点，其关键字数量必须是t-1
2. 合并节点的父节点含有的关键字数目必须大于t-1

```
1  B-TREE-MERGE(x,i,y,z)
2  y.n=2t-1
3  for j=t+1 to 2t-1
4      y.key[j]=z.key[j-t]
5  y.key[t]=x.key[i] //the key from node x merge to node y as the tth key
6  if not y.leaf
7      for j=t+1 to 2t
8          y.c[j]=z.c[j-t]
9  for j=i+1 to x.n
10     x.key[j-1]=x.key[j]
11     x.c[j]=x.c[j+1]
12 x.n=x.n-1
13 Free(z)
```

2.3 Shift

shift方法用于删除操作时，为了保证递归的节点关键字数量大于t-1，要从左边或者右边挪一个节点到当前节点，这两个方法就是执行这个操作，当且仅当左右节点的关键字数量均为t-1时(即没有多余的关键字可以挪给其他节点了)，才执行merge操作

```
1  B-TREE-SHIFT-TO-LEFT-CHILD(x,i,y,z)
2  y.n=y.n+1
3  y.key[y.n]=x.key[i]
4  x.key[i]=z.key[1]
5  z.n=z.n-1
6  j=1
7  while j≤z.n
8      z.key[j]=z.key[j+1]
9      j=j+1
10 if not z.leaf
11     y.c[y.n+1]=z.c[1]
12     j=1
13     while j≤z.n+1
14         z.c[j]=z.c[j+1]
15         j++
16 DISK-WRITE(y)
17 DISK-WRITE(z)
18 DISK-WRITE(x)
```

```
1  B-TREE-SHIFT-TO-RIGHT-CHILD(x,i,y,z)
2  z.n=z.n+1
```

```
3  j=z.n
4  while j>1
5      z.key[j]=z.key[j-1]
6      j--
7  z.key[1]=x.key[i]
8  x.key[i]=y.key[y.n]
9  if not z.leaf
10     j=z.n
11     while j>0
12         z.c[j+1]=z.c[j]
13         j--
14     z.c[1]=y.c[y.n+1]
15 y.n=y.n-1
16 DISK-WRITE(y)
17 DISK-WRITE(z)
18 DISK-WRITE(x)
19
```

2.4 插入

B树的插入操作从本质上来说是自底向上的

1. 首先将关键字插入到叶节点
2. 如果叶节点在插入之前就是满的，那么需要进行分裂操作，而分裂操作又会产生一个新节点插入到父节点中，如果父节点此时也是满的，那么首先需要分裂父节点…递归向上…

这种做法存在一个问题，因为需要访问父节点，如果持有一个父节点的指针那么会导致空间浪费，如果不持有父节点的指针，那么父节点的查找又会比较耗时。而且这种做法复杂度相对较高，实现较繁琐

因此采用了一种自顶向下**预分裂**的做法

1. 进行关键字插入操作时，会有一条从根节点到叶节点的访问路径
2. 在该条访问路径上，一旦某个节点已经满了，那么预先进行一次分裂操作(需要区分根节点与其他节点，如果根节点满了，则树高需要增加1)
3. 在进行分裂操作时，由于上一条规则可以保证，进行分裂操作的节点的父节点必定不为满节点，因此不会触发递归向上的分裂操作

下面的伪代码就是自顶向下的**预分裂**

根节点需要单独讨论

```
1  B-TREE-INSERT(T,k)
```

```
2  r=T.root
3  if r.n==2t-1 //需要处理根节点，若满了，则进行一次分裂，这是树增高的唯一方式
4      s=ALLOCATE-NODE()//分配一个节点作为根节点
5      T.root=s
6      s.leaf=FLASE//显然由分裂生成的根必然是内部节点
7      s.n=0
8      s.c[1]=r//之前的根节点作为新根节点的第一个孩子
9      B-TREE-SPLIT-CHILD(s,1)
10     B-TREE-INSERT-NONFULL(s,k)
11 else B-TREE-INSERT-NONFULL(r,k)
```

以下是非根节点的递归插入操作

1. 参数x必定是非满节点

```
1  B-TREE-INSERT-NONFULL(x,k)
2  i=x.n
3  if x.leaf //如果是叶节点，保证是非满的，找到适当的位置插入即可
4      while i ≥1 and k<x.key[i]
5          x.key[i+1]=x.key[i]
6          i=i-1
7      x.key[i+1]=k
8      x.n=x.n+1
9      DISK-WRITE(x)
10 else while i ≥ 1 and k<x.key[i]
11         i=i-1
12     i=i+1//转到对应的指针坐标
13     DISK-READ(x.c[i])
14     if x.c[i.n]==2t-1
15         B-TREE-SPLIT-CHILD(x,i)
16         if k>x.key[i]  //原来在i位置的关键字现在在i+1位置上，i位置上是y.key[t]
17             i=i+1
18     B-TREE-INSERT-NONFULL(x.c[i],k)
```

2.5 删除

B树的删除操作本质上来说是自底向上的

1. 首先找到要删除关键字的节点
2. 如果该节点的关键字数量为t-1，则需要进行shift或者merge操作
3. 如果执行了merge操作会使得父节点的关键字数量减少1，如果父节点的关键字数量也是t-1，则父节点可能首先要进行一次merge…递归向上…

这种做法存在一个问题，因为需要访问父节点，如果持有一个父节点的指针那么会导致空间浪费，如果不持有父节点的指针，那么父节点的查找又会比较耗时。而且这种做法复杂度相对较高，实现较繁琐

因此采用了一种自顶向下**预合并**的做法

1. 进行关键字删除操作时，会有一条从根节点到被删除的关键字所在节点的访问路径
2. 在该条访问路径上，一旦某个节点的关键字数量为t-1，那么预先进行一次合并操作(需要区分根节点与其他节点，如果根节点关键字数量为1，则树高需要减少1)
3. 在进行合并操作时，由于上一条规则可以保证，进行合并操作的节点的父节点的关键字数量必定大于t-1，因此不会触发递归向上的合并操作

下面的伪代码就是自顶向下的**预合并**

根节点需要单独讨论

```
1  B-TREE-DELETE(T,k) //以下都是delete会用到的函数
2  r=T.root
3  if r.n==1
4      DISK-READ(r.c[1])
5      DISK-READ(r.c[2])
6      y=r.c[1]
7      z=r.c[2]
8      if not r.leaf and y.n==z.n==t-1
9          B-TREE-MERGE-CHILD(r,1,y,z)
10         T.root=y
11         FREE-NODE(r)
12         B-TREE-DELETE-NOTNONE(y,k)
13     else B-TREE-DELETE-NOTNONE(r,k)
14 else B-TREE-DELETE-NOTNONE(r,k)
```

以下是非根节点的递归删除操作

1. 参数x的关键字数量必定大于t-1

```
1  B-TREE-DELETE-NOTNONE(x,k)
2  i=1
3  if x.leaf
4      while i ≤ x.n and k>x.key[i]
5          i=i+1
6      if k==x.key[i]
7          for j=i+1 to x.n
8              x.key[j-1]=x.key[j]
9          x.n=x.n-1
10         DISK-WRITE(x)
11     else error:”the key does not exist”
12 else
13     while i ≤ x.n and k>x.key[i]
14         i=i+1
15     DISK-READ(x.c[i])
```

```
    y=x.c[i]
    if i ≤ x.n
        DISK-READ(x.c[i+1])
        z=x.c[i+1]
    if i ≤ x.n and k==x.key[i]       //Cases 2
        if y.n>t-1    //Cases 2a
            k’=B-TREE-MIMIMUM(y)
            B-TREE-DELETE-NOTNONE(y,k’)
            x.key[i]=k’
        elseif z.n>t-1  //Case 2b
            k’=B-TREE-MAXIMUM(z)
            B-TREE-DELETE-NOTNONE(z,k’)
            x.key[i]=k’
        else B-TREE-MERGE-CHILD(x,i,y,z) //Cases 2c
            B-TREE-DELETE-NOTNONE(y,k)
    else   //Cases3
        if i>1
            DISK-READ(x.c[i-1])
            p=x.c[i-1]
        if y.n==t-1
            if i>1 and p.n>t-1  //Cases 3a
                B-TREE-SHIFT-TO-RIGHT-CHILD(x,i-1,p,y)
            elseif i ≤ x.n and z.n>t-1
                B-TREE-SHIFT-TO-LEFT-CHILD(x,i,y,z)
            elseif i>1  //Cases 3b
                B-TREE-MERGE-CHILD(x,i-1,p,y)
                y=p
            else B-TREE-MERGE-CHILD(x,i,y,z) //Cases 3c
        B-TREE-DELETE-NOTNONE(y,k)
```

删除操作大致上可以分为三类

1. 删除的关键字在叶节点上，删除即可
2. 删除的关键字位于某个中间节点，在左子树中找最大值或者右子树中找最小值代替当前的值，然后递归删除这个最小或者最大值
3. 继续在子树中查找被删除的节点，必须保证递归时的节点关键字大于t-1，当关键字为t-1时，需要执行shift或者merge操作

3 Java代码

3.1 节点

```
1  public class BTreeNode {
2      int n;
```

```
3      int[] keys;
4      BTreeNode[] children;
5      boolean isLeaf;
6
7      BTreeNode(int t) {
8          n = 0;
9          keys = new int[2 * t - 1];
10         children = new BTreeNode[2 * t];
11         isLeaf = false;
12     }
13 }
```

3.2 B树

```
1   package org.liuyehcf.algorithm.datastructure.tree.btree;
2
3   import java.util.*;
4
5   /**
6    * Created by HCF on 2017/4/5.
7    */
8
9   public class BTree {
10      private int t;
11
12      private BTreeNode root;
13
14      public BTree(int t) {
15          this.t = t;
16          this.root = createNode();
17          this.root.isLeaf = true;
18      }
19
20      private BTreeNode createNode() {
21          return new BTreeNode(t);
22      }
23
24      public void insert(int k) {
25          if (root.n == 2 * t - 1) {
26              BTreeNode s = createNode();
27              s.isLeaf = false;
28              s.children[0] = root;
29              root = s;
30              split(root, 0);
31          }
32          insertNotFull(root, k);
```

```
33          if (!check())
34              throw new RuntimeException();
35      }
36
37      private void split(BTreeNode x, int i) {
38          BTreeNode z = createNode();
39          BTreeNode y = x.children[i];
40          for (int j = 0; j < t - 1; j++) {
41              z.keys[j] = y.keys[j + t];
42          }
43          if (!y.isLeaf) {
44              for (int j = 0; j < t; j++) {
45                  z.children[j] = y.children[j + t];
46              }
47          }
48          for (int j = x.n; j > i; j--) {
49              x.keys[j] = x.keys[j - 1];
50              x.children[j + 1] = x.children[j];
51          }
52          x.keys[i] = y.keys[t - 1];
53          x.children[i + 1] = z;
54          x.n++;
55
56          z.n = y.n = t - 1;
57          z.isLeaf = y.isLeaf;
58      }
59
60      private void insertNotFull(BTreeNode x, int k) {
61          int i = x.n - 1;
62          if (x.isLeaf) {
63              while (i >= 0 && x.keys[i] >= k) {
64                  x.keys[i + 1] = x.keys[i];
65                  i--;
66              }
67              i++;
68              x.keys[i] = k;
69              x.n++;
70          } else {
71              while (i >= 0 && x.keys[i] >= k) {
72                  i--;
73              }
74              i++;
75              if (x.children[i].n == 2 * t - 1) {
76                  split(x, i);
77                  if (k > x.keys[i]) {
78                      i++;
79                  }
```

```
80              }
81              insertNotFull(x.children[i], k);
82          }
83      }
84
85      private boolean check() {
86          return checkN(root);
87      }
88
89      private boolean checkN(BTreeNode x) {
90          if (x.isLeaf) {
91              return (x == root) || (x.n >= t - 1 && x.n <= 2 * t - 1);
92          } else {
93              boolean flag = (x == root) || (x.n >= t - 1 && x.n <= 2 * t - 1
94              for (int i = 0; i <= x.n; i++) {
95                  flag = flag && checkN(x.children[i]);
96              }
97              return flag;
98          }
99      }
100
101     public void insert(int[] keys) {
102         for (int key : keys) {
103             insert(key);
104         }
105     }
106
107     public void delete(int k) {
108         if (root.n == 1) {
109             if (!root.isLeaf && root.children[0].n == t - 1 && root.childre
110                 merge(root, 0);
111                 root = root.children[0];
112             }
113         }
114         deleteNotNone(root, k);
115         if (!check())
116             throw new RuntimeException();
117     }
118
119     private void merge(BTreeNode x, int i) {
120         BTreeNode y = x.children[i];
121         BTreeNode z = x.children[i + 1];
122         for (int j = 0; j < t - 1; j++) {
123             y.keys[j + t] = z.keys[j];
124         }
125         if (!y.isLeaf) {
126             for (int j = 0; j < t; j++) {
```

```
127                 y.children[j + t] = z.children[j];
128             }
129         }
130 
131         y.keys[t - 1] = x.keys[i];
132         for (int j = i; j < x.n - 1; j++) {
133             x.keys[j] = x.keys[j + 1];
134             x.children[j + 1] = x.children[j + 2];
135         }
136         x.n--;
137         y.n = 2 * t - 1;
138     }
139 
140     private void deleteNotNone(BTreeNode x, int k) {
141         int i = 0;
142         if (x.isLeaf) {
143             while (i < x.n && k > x.keys[i]) {
144                 i++;
145             }
146             if (x.keys[i] != k) throw new RuntimeException("no such an elem
147             while (i < x.n - 1) {
148                 x.keys[i] = x.keys[i + 1];
149                 i++;
150             }
151             x.n--;
152         } else {
153             while (i < x.n && k > x.keys[i]) {
154                 i++;
155             }
156             BTreeNode y = x.children[i];
157             BTreeNode z = null;
158             if (i < x.n) {
159                 z = x.children[i + 1];
160             }
161             if (i < x.n && x.keys[i] == k) {
162                 if (y.n > t - 1) {
163                     int kk = maximum(x.children[i]);
164                     deleteNotNone(x.children[i], kk);
165                     x.keys[i] = kk;
166                 } else if (z.n > t - 1) {
167                     int kk = minimum(x.children[i + 1]);
168                     deleteNotNone(x.children[i + 1], kk);
169                     x.keys[i] = kk;
170                 } else {
171                     merge(x, i);
172                     deleteNotNone(x.children[i], k);
173                 }
```

```
174             } else {
175                 BTreeNode p = null;
176                 if (i > 0) {
177                     p = x.children[i - 1];
178                 }
179                 if (y.n == t - 1) {
180                     if (p != null && p.n > t - 1) {
181                         shiftToRight(x, i - 1, p, y);
182                     } else if (z != null && z.n > t - 1) {
183                         shiftToLeft(x, i, y, z);
184                     } else if (p != null) {
185                         merge(x, i - 1);
186                         y = p;
187                     } else {
188                         merge(x, i);
189                     }
190                 }
191                 deleteNotNone(y, k);
192             }
193         }
194     }
195 
196     private int maximum(BTreeNode x) {
197         while (!x.isLeaf) {
198             x = x.children[x.n];
199         }
200         return x.keys[x.n - 1];
201     }
202 
203     private int minimum(BTreeNode x) {
204         while (!x.isLeaf) {
205             x = x.children[0];
206         }
207         return x.keys[0];
208     }
209 
210     private void shiftToRight(BTreeNode x, int i, BTreeNode p, BTreeNode y)
211         for (int j = y.n; j > 0; j--) {
212             y.keys[j] = y.keys[j - 1];
213         }
214         y.keys[0] = x.keys[i];
215         x.keys[i] = p.keys[p.n - 1];
216 
217         if (!y.isLeaf) {
218             for (int j = y.n + 1; j > 0; j--) {
219                 y.children[j] = y.children[j - 1];
220             }
```

```
221             y.children[0] = p.children[p.n];
222         }
223 
224         y.n++;
225         p.n--;
226     }
227 
228     private void shiftToLeft(BTreeNode x, int i, BTreeNode y, BTreeNode z)
229         y.keys[y.n] = x.keys[i];
230         x.keys[i] = z.keys[0];
231         for (int j = 0; j < z.n - 1; j++) {
232             z.keys[j] = z.keys[j + 1];
233         }
234         if (!y.isLeaf) {
235             y.children[y.n + 1] = z.children[0];
236             for (int j = 0; j < z.n; j++) {
237                 z.children[j] = z.children[j + 1];
238             }
239         }
240         y.n++;
241         z.n--;
242     }
243 
244     public void inOrderTraverse() {
245         inOrderTraverse(root);
246         System.out.println();
247     }
248 
249     private void inOrderTraverse(BTreeNode x) {
250         if (x.isLeaf) {
251             for (int i = 0; i < x.n; i++) {
252                 System.out.print(x.keys[i] + ", ");
253             }
254         } else {
255             for (int i = 0; i < x.n; i++) {
256                 inOrderTraverse(x.children[i]);
257                 System.out.print(x.keys[i] + ", ");
258             }
259             inOrderTraverse(x.children[x.n]);
260         }
261     }
262 
263     public boolean search(int k) {
264         return search(root, k);
265     }
266 
267     private boolean search(BTreeNode x, int k) {
```

```
268         if (x.isLeaf) {
269             for (int i = 0; i < x.n; i++) {
270                 if (k == x.keys[i]) return true;
271             }
272             return false;
273         } else {
274             int i = 0;
275             while (i < x.n && k > x.keys[i]) {
276                 i++;
277             }
278             if (i < x.n && k == x.keys[i]) return true;
279             return search(x.children[i], k);
280         }
281     }
282 
283     public int successor(int k) {
284         if (!search(k)) throw new RuntimeException();
285         return successor(root, k);
286     }
287 
288     private int successor(BTreeNode x, int k) {
289         int i = 0;
290         if (x.isLeaf) {
291             while (x.keys[i] <= k) {
292                 i++;
293             }
294             //i must less than x.n
295             return x.keys[i];
296         } else {
297             while (i < x.n && x.keys[i] <= k) {
298                 i++;
299             }
300             if (k >= maximum(x.children[i])) {
301                 //i couldn't equals x.n
302                 return x.keys[i];
303             } else {
304                 return successor(x.children[i], k);
305             }
306         }
307     }
308 
309     public int precursor(int k) {
310         if (!search(k)) throw new RuntimeException();
311         return precursor(root, k);
312     }
313 
314     private int precursor(BTreeNode x, int k) {
```

```
315         int i = x.n - 1;
316         if (x.isLeaf) {
317             while (x.keys[i] >= k) {
318                 i--;
319             }
320             //i must no less than 0
321             return x.keys[i];
322         } else {
323             while (i >= 0 && x.keys[i] >= k) {
324                 i--;
325             }
326             if (k <= minimum(x.children[i + 1])) {
327                 //i must large than 0
328                 return x.keys[i];
329             } else {
330                 return precursor(x.children[i + 1], k);
331             }
332         }
333     }
334 
335     public static void main(String[] args) {
336         long start = System.currentTimeMillis();
337 
338         Random random = new Random();
339 
340         int TIMES = 10;
341 
342         while (--TIMES > 0) {
343             System.out.println("剩余测试次数: " + TIMES);
344             BTree bTree = new BTree(random.nextInt(20) + 3);
345 
346             int N = 10000;
347             int M = N / 2;
348 
349             Set<Integer> set = new HashSet<Integer>();
350             for (int i = 0; i < N; i++) {
351                 set.add(random.nextInt());
352             }
353 
354             List<Integer> list = new ArrayList<Integer>(set);
355             Collections.shuffle(list, random);
356             //插入N个数据
357             for (int i : list) {
358                 bTree.insert(i);
359             }
360 
361             //删除M个数据
```

```
362             Collections.shuffle(list, random);
363 
364             for (int i = 0; i < M; i++) {
365                 set.remove(list.get(i));
366                 bTree.delete(list.get(i));
367             }
368 
369             //再插入M个数据
370             for (int i = 0; i < M; i++) {
371                 int k = random.nextInt();
372                 set.add(k);
373                 bTree.insert(k);
374             }
375             list.clear();
376             list.addAll(set);
377             Collections.shuffle(list, random);
378 
379             //再删除所有元素
380             for (int i : list) {
381                 bTree.delete(i);
382             }
383         }
384         long end = System.currentTimeMillis();
385         System.out.println("Run time: " + (end - start) / 1000 + "s");
386     }
387 }
```

4 参考

- 《算法导论》