实验报告：B 树

涂奕腾 20201018

一、需求分析

实现 B 树来维护一些数，其中需要提供以下操作：

1.插入 x 数
2.删除 x 数(若有多个相同的数，因只删除一个)
3.查询 x 数的排名(排名定义为比当前数小的数的个数 +1)
4.查询排名为 x 的数
5.求 x 的前驱(前驱定义为小于 x，且最大的数)
6.求 x 的后继(后继定义为大于 x，且最小的数)

二、实现设计

1. B 树整体设计

定义 M 为 B 树的最小度数，故除了根节点每个节点至少有 $M-1$ 个关键字，M 个儿子节点；每个节点至多有 $2M-1$ 个关键字和 $2M$ 个子节点。

2. B 树节点的设计

val 键值对记录每个关键字的值和出现次数，Node*ch[]和 fa 分别用于记录子节点指针和父节点指针，keyNum 用于记录当前节点关键字个数，siz 用于记录子树大小(此处定义为子树中所有关键字的出现次数之和)，isLeaf 用于判断是否是叶节点。

```
template <typename T>
typedef pair<T, int> Type;
struct Node{
    Type val[key_max];
    Node *ch[ch_max] = {NULL}, *fa = NULL;
    int keyNum = 0, siz = 0;
    bool isLeaf = true;
    T key(int i) {return val[i].fi;}
    int &cnt(int i) {return val[i].se;}
    Node(Node *fa = NULL) : fa(fa) {}
};
```

3. 插入操作的设计

将新的关键字插入到一个叶节点上，若不是新关键字则相应节点上关键字的计数加一。当一个节点已满($2M-1$ 个关键字)时，按其中间关键字分裂为两个各含 $M-1$ 个关键字的节点，中间关键字提升到父亲节点以标识两颗新树的划分点。但如果该节点的父亲节点也是满的，就必须在插入关键字之前分裂，最终满节点的分裂会沿着树向上传播。

```
void split(Node *x, int k){//将 x 的 k 儿子 y 分裂为 2 个儿子 y,z
    Node *z = new Node(x), *y = x->ch[k];
    z->isLeaf = y->isLeaf;
    memmove(z->val, y->val + M, key_min * sizeof(Type));
    if(!y->isLeaf){
        memmove(z->ch, y->ch + M, ch_min * sizeof(Node*));
        for(int i = 0; i < ch_min; ++i) z->ch[i]->fa = z;
    }
    z->keyNum = y->keyNum = key_min;
    memmove(x->ch + k + 2, x->ch + k + 1, (x->keyNum - k) * sizeof(Node*));
    x->ch[k + 1] = z;
    memmove(x->val + k + 1, x->val + k, (x->keyNum - k) * sizeof(Type));
    x->val[k] =  y->val[key_min];
    y->siz = calsiz(y), z->siz = calsiz(z);
    ++x->keyNum;
}
```

但在实际插入时，我们并不是等到找出插入过程中实际要分裂的满节点时才分裂，实际上在沿着树向下查找新的关键字所属位置时就沿途分裂每一个遇到的满节点(包括叶子节点自身)，这样就能保证在分裂一个满节点时，其父亲节点一定是不满的。

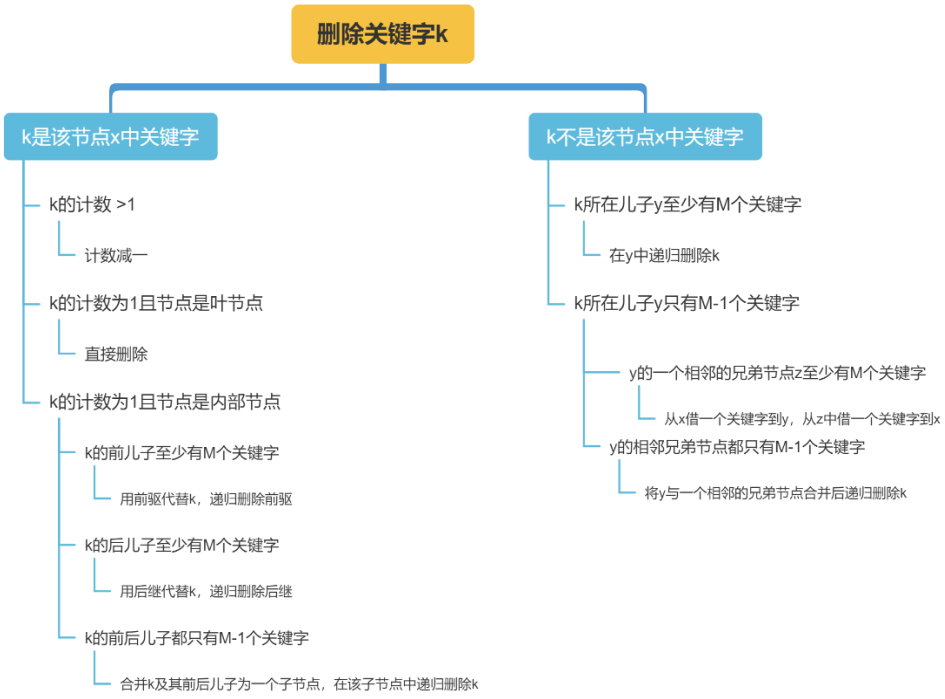
```
void Insert(T val){
    if(rt->keyNum == key_max){//根节点已满，修改根节点
        Node *x = new Node;
        x->isLeaf = false, x->ch[0] = rt, x->siz = rt->siz;
        rt->fa = x; rt = x;
        split(x, 0);
    }
    insert_Nonfull(rt, val);//插入到已经存在的叶节点上
}

void insert_Nonfull(Node *x, T val){
    while(1){
        int i = lower_bound(x->val, x->val + x->keyNum, val, [](const Type &a,
const T &b) {return a.fi < b;}) - x->val;//二分查找插入位置
        if(i != x->keyNum && val == x->val[i].fi){//关键字已经被记录则增加计数
            ++x->cnt(i);
            while(x) ++x->siz, x = x->fa;//维护各级子树大小
            return;
        }
        if(x->isLeaf){//如果是叶节点，则直接插入
            memmove(x->val + i + 1, x->val + i, (x->keyNum - i) *
sizeof(Type));
            x->val[i] = mp(val, 1);
            ++x->keyNum;
            while(x) ++x->siz, x = x->fa;
            return;
```

```
        }
        if(x->ch[i]->keyNum == key_max){//查找新关键字的位置时就沿途分裂遇到的每个满节点，保证分裂每个满节点时其父亲必不满
            split(x, i);
            if(x->key(i) < val) ++i;
            else if (val == x->key(i)){
                ++x->cnt(i);
                while(x) ++x->siz, x = x->fa;
                return;
            }
        }
        x = x->ch[i];
    }
}
```

时间复杂度：二分查找$O(logM)$，树深度$O(logN)$，总复杂度$O(logMlogN)$

4. 删除操作的设计

删除节点的思路如下图所示：



合并的代码如下：

```
void merge(Node *x, int k){//将 x 的 k 儿子与 k+1 儿子合并，同时合并 x 中第 k 个关键字，两个孩子原本都只有 M-1 个关键字，合并后有 2*M-1 个
    Node *y = x->ch[k], *z = x->ch[k + 1];
    y->keyNum = key_max;
    y->val[key_min] = x->val[k];
    memmove(y->val + M, z->val, key_min * sizeof(Type));
    if(!y->isLeaf){
```

```
        memmove(y->ch + M, z->ch, ch_min * sizeof(Node*));
        for(int i = ch_min; i < ch_max; ++i) y->ch[i]->fa = y;
    }
    memmove(x->val + k, x->val + k + 1, (x->keyNum -k - 1) * sizeof(Type));
    memmove(x->ch + k + 1, x->ch + k + 2, (x->keyNum -k - 1) * sizeof(Node*));
    --x->keyNum, y->siz = calsiz(y);
    free(z);
}
```

删除的代码如下：

```
void erase(Node *x, T val){
    int i = lower_bound(x->val, x->val + x->keyNum, val, [](const Type &a,
const T &b) {return a.fi < b;}) - x->val;
    if(i != x->keyNum && val == x->val[i].fi){
        if(x->cnt(i) > 1){
            --x->cnt(i);
            while(x) --x->siz, x = x->fa;
            return;
        }
        if(x->isLeaf){//叶节点，直接删除
            memmove(x->val + i, x->val + i + 1, (--x->keyNum - i) *
sizeof(Type));
            while(x) --x->siz, x = x->fa;
        }
        else{
            if(x->ch[i]->keyNum >= M){//前驱所在儿子有足够多儿子
                Node *y = x->ch[i];
                while(!y->isLeaf) y = y->ch[y->keyNum];//找前驱
                x->val[i] = y->val[y->keyNum - 1];
                if(x->cnt(i) != 1){ //y 的对应节点 cnt 有多个，那么沿路减 siz;只有一
个单独处理
                    y->cnt(y->keyNum - 1) = 1;
                    while(y != x) y->siz -= x->cnt(i) - 1, y = y->fa;
                }
                erase(x->ch[i], x->key(i));
            }
            else if(x->ch[i + 1]->keyNum >= M){//后继所在儿子有足够多儿子
                Node *y = x->ch[i + 1];
                while(!y->isLeaf) y = y->ch[0];//找后继
                x->val[i] = y->val[0];
                if(x->cnt(i) != 1){
                    y->cnt(0) = 1;
                    while(y != x) y->siz -= x->cnt(i) - 1, y = y->fa;
                }
                erase(x->ch[i + 1], x->key(i));
            }
```

```
            else{//否则合并节点
                merge(x, i);
                if(!rt->keyNum) rt = x->ch[i], rt->fa =NULL;//没有键，但是还可能有一个儿子，根变成该儿子
                erase(x->ch[i], val);
            }
        }
    }
    else if(!x->isLeaf){
        if(x->ch[i]->keyNum == key_min){//需要合并的情况
            Node *y = x->ch[i];
            if(i >= 1 && x->ch[i - 1]->keyNum >= M){//找左兄弟借节点，把 x 的一个键移入要删的 key 所在孩子，把它的兄弟的一个 key 和孩子移入 x
                Node *z = x->ch[i - 1];
                memmove(y->val + 1, y->val, y->keyNum * sizeof(Type));
                y->val[0] = x->val[i-1];
               x->val[i - 1] = z->val[z->keyNum - 1];
                if(!y->isLeaf){
                    memmove(y->ch + 1, y->ch, (y->keyNum + 1) * sizeof(Node*));
                    y->ch[0] = z->ch[z->keyNum], y->ch[0]->fa = y;
                }
                --z->keyNum, ++y->keyNum;
                y->siz = calsiz(y), z->siz = calsiz(z);
                erase(y, val);
            }
            else if(i < x->keyNum && x->ch[i + 1]->keyNum >= M){
                Node *z = x->ch[i + 1];
                y->val[y->keyNum] = x->val[i];
                x->val[i] = z->val[0];
                if(!y->isLeaf){
                    y->ch[y->keyNum + 1] = z->ch[0], y->ch[y->keyNum + 1]->fa = y;
                    memmove(z->ch , z->ch + 1, z->keyNum * sizeof(Node*));
                }
                memmove(z->val, z->val + 1, (z->keyNum - 1) * sizeof(Type));
                --z->keyNum, ++y->keyNum;
                y->siz = calsiz(y), z->siz = calsiz(z);
                erase(y, val);
            }
            else {//两个兄弟都没有节点借,则与左右任一兄弟合并
                if(i) --i;
                y = x->ch[i];
                merge(x, i);
                if(!rt->keyNum) rt = y, rt->fa = NULL;
                erase(y, val);
            }
```

```
        }
        else erase(x->ch[i], val);
    }
    //如果是叶节点还没有找对对应的值则树中无该值
}
```

时间复杂度：二分查找$O(logM)$，树深度$O(logN)$，总复杂度$O(logMlogN)$

5. 四种查找操作的设计

(1) 查询排名

统计小于目标关键字的关键字个数即可，通过统计关键字的计数和子树大小实现。

```
int Rank(T val){
    Node *x = rt;
    int res = 0;
    while(x){
        if(x->key(x->keyNum - 1) < val){
            res += x->siz - getsiz(x->ch[x->keyNum]);
            x = x->ch[x->keyNum];
            continue;
        }
        for(int i = 0; i < x->keyNum; ++i){
            if(x->key(i) < val) res += getsiz(x->ch[i]) + x->cnt(i);
            else if(x->key(i) == val) return res + getsiz(x->ch[i]) + 1;
            else {x = x->ch[i]; break;}
        }
    }
    return res;
}
```

时间复杂度：顺序查找$O(M)$，树深度$O(logN)$，总复杂度$O(MlogN)$

(2) 按排名查询

类似地，通过统计关键字的计数和子树大小实现，flag 判断不存在那么多数的情况。

```
T Kth(int k){
    Node *x = rt;
    while(x){
        bool flag = 0;
        for(int i = 0; i<= x->keyNum; ++i){
            int l = getsiz(x->ch[i]) + 1, r = getsiz(x->ch[i]) + (i ==
x->keyNum ? 1 : x->cnt(i));
            if(k >= l && k <= r) return x->key(i);
            if(k < l){x = x->ch[i]; flag = 1; break;}
            k -= r;
        }
```

```
        if(!flag) return 0;
    }
}
```

时间复杂度：顺序查找$O(M)$，树深度$O(logN)$，总复杂度$O(MlogN)$

(3) 查询前驱

```
T Predecessor(T val) {
    T res = -inf; Node* x = rt;
    while(x){
        int i = lower_bound(x->val, x->val + x->keyNum, val, [](const Type &a,
const T &b) {return a.fi < b;}) - x->val;
        if(i) res = x->key(i-1);
        x = x->ch[i];
    }
    return res;
}
```

时间复杂度：二分查找$O(logM)$，树深度$O(logN)$，总复杂度$O(logMlogN)$

(4) 查询后继

类似于查询前驱。

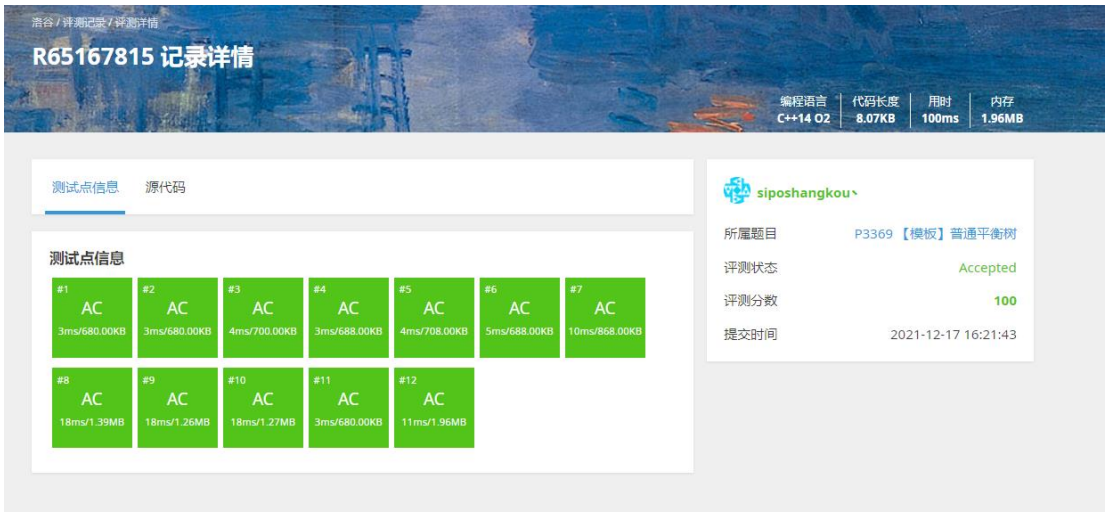
```
T Successor(T val) {
    T res = inf; Node* x = rt;
    while(x){
        int i = upper_bound(x->val, x->val + x->keyNum, val, [](const T &b,
const Type &a) {return a.fi > b;}) - x->val;
        if(i != x->keyNum) res = x->key(i);
        x = x->ch[i];
    }
    return res;
}
```

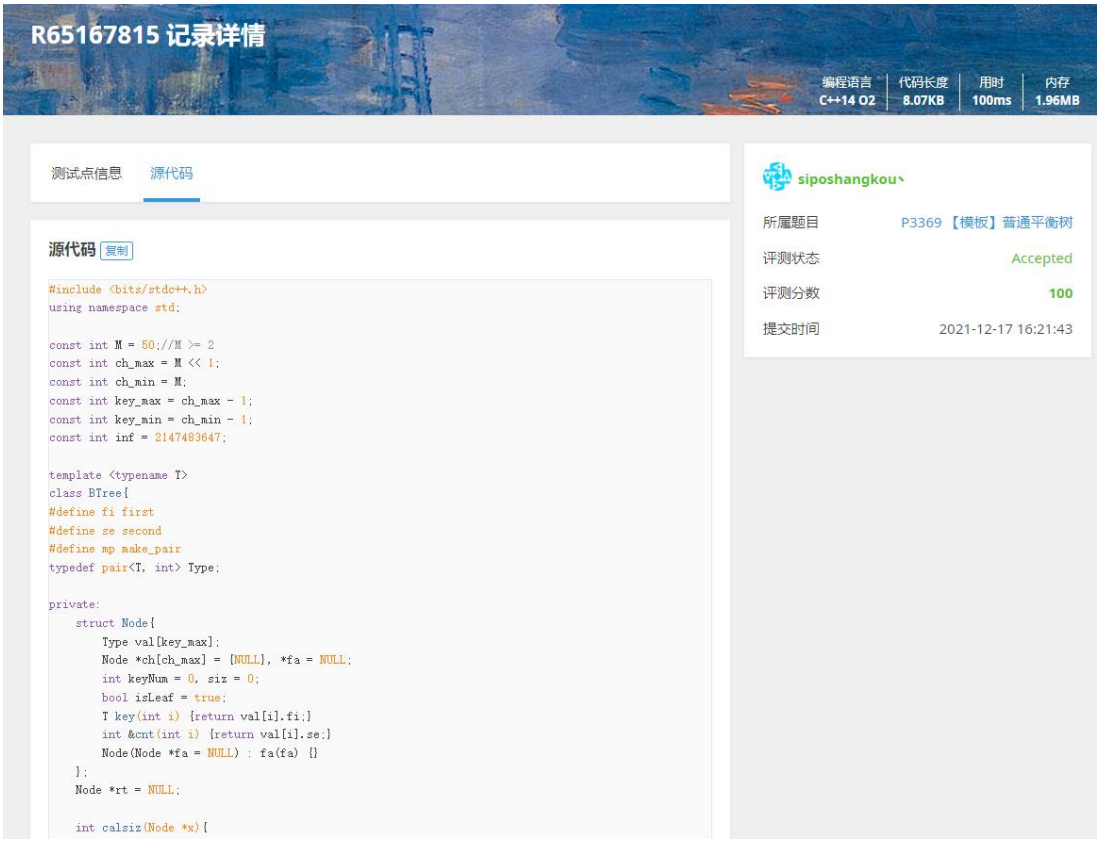
时间复杂度：二分查找$O(logM)$，树深度$O(logN)$，总复杂度$O(logMlogN)$

三、测试样例

见 in.txt, out.txt

四、AC 记录


洛谷 / 评测记录 / 评测详情
R65167815 记录详情
编程语言 C++14 O2
代码长度 8.07KB
用时 100ms
内存 1.96MB
测试点信息
源代码
测试点信息
#1 AC 3ms/680.00KB
#2 AC 3ms/680.00KB
#3 AC 4ms/700.00KB
#4 AC 3ms/688.00KB
#5 AC 4ms/708.00KB
#6 AC 5ms/688.00KB
#7 AC 10ms/868.00KB
#8 AC 18ms/1.39MB
#9 AC 18ms/1.26MB
#10 AC 18ms/1.27MB
#11 AC 3ms/680.00KB
#12 AC 11ms/1.96MB
siposhangkou
所属题目 P3369 【模板】普通平衡树
评测状态 Accepted
评测分数 100
提交时间 2021-12-17 16:21:43


R65167815 记录详情
编程语言 C++14 O2
代码长度 8.07KB
用时 100ms
内存 1.96MB
测试点信息
源代码
源代码 复制
#include <bits/stdc++.h>
using namespace std;
const int M = 50;//M >= 2
const int ch_max = M << 1;
const int ch_min = M;
const int key_max = ch_max - 1;
const int key_min = ch_min - 1;
const int inf = 2147483647;
template <typename T>
class BTree{
#define fi first
#define se second
#define mp make_pair
typedef pair<T, int> Type;
private:
struct Node{
Type val[key_max];
Node *ch[ch_max] = {NULL}, *fa = NULL;
int keyNum = 0, siz = 0;
bool isLeaf = true;
T key(int i) {return val[i].fi;}
int &cnt(int i) {return val[i].se;}
Node(Node *fa = NULL) : fa(fa) {}
};
Node *rt = NULL;
int calsiz(Node *x){
siposhangkou
所属题目 P3369 【模板】普通平衡树
评测状态 Accepted
评测分数 100
提交时间 2021-12-17 16:21:43