B+树的模拟

李钰 19335112
小组成员 林雁纯

摘要 本程序实现了一个 B+树的插入、查找、删除操作，并模拟了存取的时间延迟，更具有真实性

一. 引言

1. 解决问题

设计 B+树用于记录查找、插入和删除的算法，包含一个自行设计的 20ms 延时器模拟一次外部存取的时间延迟。

2. 解决方法

(1) 描述 B+树的概念和 B+树的逻辑结构

B+树是应文件系统所需而出的一种 B 树的变型树。一棵 m 阶的 B+树和 m 阶的 B 树的差异在于：

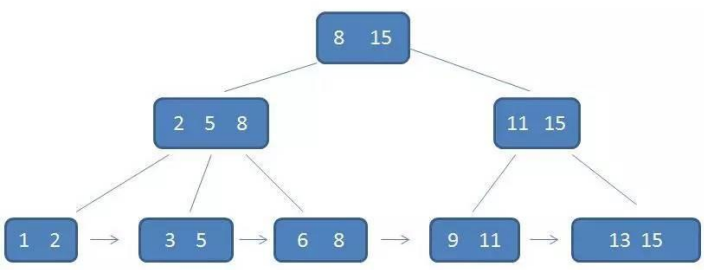
a. 有 n 棵子树的节点中含有 n 个关键字

b. 所有的叶子节点中包含了全部关键字的信息，及指向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。

c. 所有的非终端节点可以看成是索引部分，节点中仅含有其子树（根节点）中的最大（或最小）关键字。

注： 来源 清华大学出版社严蔚敏 《数据结构》

图例： 来源：https://www.jianshu.com/p/71700a464e97



- 根据题目要求，一棵 m 阶的 B+树最多有 m 个节点，至少有 m / 2（向上取整）个节点，我们选择让所有非终端节点中的 key 值是子树中的最大关键字。
- B+树存储记录，即 key 与 value 组成的键值对。我们利用 key 值充当索引，利用搜索 key 来实现查找并返回其 value、删除记录等操作。

(2) 设计存储上述 B+树的数据结构设计 （程序设计语言描述）

详见第二部分 2.使用的数据结构

(3) 用一个大小为 40Bytes 的内存单元模拟一个外部存储块，规定关键字大小为 4Bytes，地址大小为 4Bytes，记录信息数据大小为 8Bytes。确定上述 B+树的 M 值（用于内部 M-路搜索树）和 L 值（用于每个叶子块存储的记录数目）

对于内部节点：其中要存储 leaf、 number、 key、child、 parent，

1 + 1 + 4 * m + 4 * m + 4 = 40 解得 m= 4；

对于叶子节点：其重要存储 leaf、number、parent、block，

1 + 1 + 4 * l + 8 * l = 40, 解得 l = 3 为了提升存储效率，我们也可以用两个外部存储块来存叶子节点，这样叶节点可存储的记录增至 6。

(4) 设计 B+树用于记录查找、插入和删除的算法（用伪代码描述），包含一个自行设计的 20ms 延时器模拟一次外部存取的时间延迟

查找、插入以及删除算法详见第二部分 3 算法的描述，延时器的模拟详见第二部分 2.使用的数据结构，在此不再赘述。

(5) 列出源代码各个模块命名清单（不需要代码清单）

```
36  > struct node{···
64    //获取输入节点在其父亲节点中的位置
65  > int position(node* children) {···
74    //向左移动记录以删除第一个记录
75  > void erase_node_byleft(node* root, int i)···
84    //向右移动记录以腾出一个空位
85  > void insert_node_byright(node* root, int i)···
94    //重载<号，使得结构体可以直接用sort函数排序
95  > bool operator < (const record& a, const record& b) {···
98    //清空整个树
99  > void clear(node*& root) {···
117   //判断节点是否满
118 > bool FullNode(node* temp) {···
122   //将节点中的所有key和child全部右移一位
123 > void RightShift(node* parent, int pos) {···
130   //层次遍历
131 > void level_traverse(node* root) {···
191   //分裂
192 > void split(node*& root, node*& target) {···
282   //向叶子中增加记录
283 > void AddToleaf(node* leaf, record target) {···
292   //使叶节点平衡
293 > void leaf_balance(node*& root, delayTime& decelerator){···
402   //使内部节点平衡
403 > void internal_balance(node*& root, delayTime& decelerator){···
549   //插入操作
550 > bool insert(node*& root, record target, delayTime& delayer) {···
604   //删除操作
605 > bool remove(node*& root, int target, delayTime& decelerator){···
649   //查找操作
650 > bool search(node* root, int target){···
684   //获得随机记录
685 > record get_rand(){···
688   //插入测试
689 > void test_insert(node*& root){···
739   //查找测试
740 > void test_search(node* root){···
759   //删除测试
760 > void test_remove(node*& root){···
```

(6) 测试用例设计（初始化至少包含 50 个记录数据）

详见第三部分

(7) 运行结果分析，包括外部存取延时统计

详见第三部分

3. 实验目的

本实验旨在更深层次理解和掌握 B+树的概念和逻辑结构；明白其插入、删除以及查找的规则并掌握算法。加深理解 B+树相比于 B 树在文件存储与管理方面的优势所在。

二． 解决方法

1. 输入输出的形式

输出：插入、查找、删除操作的结果反馈，以及每做完一项操作之后对全树的层次遍历。

输入：

① 选择输入 0/1：在进行插入/删除操作时，用户可自主选择是否要插入/删除给定的记录。若输入 0，则表示用户不进行输入，程序继续运行；若输入 1，则表示用户要插入/删除自己给定的记录。

② 若选择输入了 1，用户要继续进行输入：想要进行插入、查找、删除操作的次数，以及想要插入的记录、查找的关键字、删除记录的关键字。

③ 在查找测试时，不需要步骤①

2. 使用的数据结构

① **结构体**

我们共设计了两个结构体。

a. 第一个结构体 record，其中有我们要存储的记录关键字和值信息。

通过该结构体，我们可以将 key 与 value 形成键值对的关系。

```
struct record{
    int key;
    int value;
    record(int key = 0, int value = 0){
        this->key = key;
        this->value = value;
    }
};
```

b. 第二个结构体 node，即表示 B+树中的每个节点。

因为叶节点中存储的是记录，内部节点中存储的是 key 值，为方便起见我们将两种节点都定义到一个结构体中，用一个布尔值来判断该节点是不是叶子。

对于非叶节点，其中 number 存储该节点中 key 的个数；数组 key 存储每一个 key 值；child 以及 parent 指向其对应的孩子节点和父亲节点；block 无效。

对于叶节点，number 中存储 record 的个数；有 block 存储记录；parent 指向其父亲；数组 key 无效，无 child。

除了数据的定义，该节点中还有一个 node 的初始化函数。

```
struct node{
    bool leaf;//是不是叶子
    int number;//该内部节点的key的个数，或者叶子节点block的个数
    int key[m + 1];
    record block[l + 1];
    node* child[m + 1];
    node* parent;

    node(){…

};
```

c. 在 node 结构体中，我们还定义了 record 的结构体数组，因为一个叶子节点中要储存的不止一个记录。

② 类

本程序只有一个类，即我们的模拟延时器。结构很简单，其中数据成员为延时次数的统计，成员函数包括初始函数、实现延时函数、以及返回延时次数的函数。其中 Sleep（20）表示延时 20ms，是 Windows 系统下，<Windows.h>头文件中的函数。

```
class delayTime{
    long long count;
public:
    delayTime(){
        count = 0;
    }
    void delay(){
        Sleep(20);
        count++;
    }
    long long getcount(){
        return count;
    }
};
```

③ 队列

在层次遍历中用到了队列这一数据结构。为了有好的输出效果，这里我们用到了三个队列：其中一个 hold 队列暂存 B+数中每一层的节点，通过循环其 size 次来控制显示“第 x 层”的输出，temp 队列则用于其每层每个节点的输出，因为发现<queue.h> 头文件中无队列清空函数，所以这里我们增设 toZero 队列，一直为空，通过将它赋值给其他队列来达到清空其他队列的目的（这里也可以用循环，pop）。

④ 数组

本程序中，每个非叶子节点的 key 值用数组存放；以及每个 key 对应的子树用 child 数组存放；叶子结点的记录用结构体数组存放。

⑤ 链表

整个 B+树的每个节点由链表链接，有的书中给出的 B+树定义说明其每个叶子节点也有连接关系，但题目未要求我们这里为节省空间就不加入叶子节点之间的指针了。

3. 算法的描述

1) 插入算法

本程序采用自底向上的插入方法。

若为空树，则新增两个节点。一个充当根其中记录新增节点的 key 值，另一个作为他的孩子成为叶，其中存储记录。更新相应的数据，返回 true。

若树不为空，但新增记录的 key 值大于根中所有的 key 的值，则将根中最大的 key 值替换为新增记录的 key 值，接着进入该 key 值对应的子树。若其不是叶子，则替换其最大的 key 值为新增记录的 key，再进入它的子树，以此类推，直到最后进入的子树是叶子节点。然后进行叶节点插入记录的操作。

若树不为空，且新增记录的 key 不比 root 中的 key 的最大值大，则在 root 的 key 数组中，找到第一个比新增 key 值大的位置，进入其子树，之后按此方法一直循环，直到进入叶子节点中。进行叶节点插入记录的操作。在这个过程中，如果发现内部节点或叶子节点中已经有相同 key 值存在，那么返回 false，插入失败。

在完成上述步骤后，进入分裂操作，之后返回 true，插入成功。

伪代码

- **Insert 函数**

Input: root<指向根结点指针的地址>, target<记录>, delayer<延时器>

Output: true<插入成功>/false<插入失败>

```
bool insert(node*& root, record target, delayTime& delayer) {
    if 树是空的{
        新增根节点;
        新增叶子节点;
        根节点中存储key值;
        叶子节点中存储记录;
        两个节点之间通过parent和child两个指针相连
        return true;
    }

    node* hold = root

    if target.key > root中key的最大值{
        while hold 不是叶{
            hold中最后一个key值改为target.key
            hold = hold的最后一个child
        }
    }
    else {
        while hold 不是叶 {
            找到target.key 应该存在的位置，即进入第一个大于他的key值的位置
            hold = 该位置的子树
    }

    进入AddToleaf函数
    进入split函数

    return true;
}
```

- **Add To leaf 函数**

Input: leaf<要进行插入操作的节点>， target<要插入的记录>

```
void AddToleaf(node* leaf, record target) {
    新增记录直接放到叶节点block最后;
    排序;
    更新记录数;
}
```

➢ **Split 函数**

Input：root<根节点的指针地址>, target<待分裂的结点指针地址>

```
void split(node*& root, node*& target) {
    if target 未满 return;

    node* parent = target->parent;
    if target是叶子
        找到他在父亲节点对应的位置
        父亲节点里从他的位置开始（不包括他的位置）key值以及child全部向右移一位
        新增节点
        确定新增叶子结点的record数
        更新原来叶子节点对应的父亲中的key值
        更新原来叶子节点的record数
        更新新增节点的record数
        给新增节点的record赋值
        更新父亲节点中key值
        if 父亲节点满
            split(root, target->parent);

    else target不是叶节点
        if target  是根节点
            新增根节点
            新增分裂节点

            更新原节点key的个数
            确定新增节点key的个数
            找到原节点现在的最大值给新根的第一个key赋值
            找到新增节点的最大值给新根的第二个key赋值
            更新新增节点的key、child

            新根与原节点和新增节点通过指针建立父子联系
            root = newroot;


        else 分裂的是非根内部节点
        类似于叶结点的分裂只不过做区分的是内部节点改变的是key和child，
        而叶节点是block

        这里不在赘述
}
```

2) 查找算法

① 若根为空，直接返回 false。

② 若根不为空，且要搜索的节点是叶子，则从第一个 record 开始遍历直到找到与待寻找的记录的 key 值相同的 key 值，返回 true，否则返回 false；

③ 若根不为空，且要搜索的节点不是叶子，则在要搜索的节点 key 中找到第一个大于待搜索的 key，并进入该 key 所对应的子树，再进行上述操作，做循环。若在该过程中遇到了相同的 key 值，则返回 true。

若一直找到了叶子节点，则进行操作②。

伪代码

Input: root<B+树的根>, target<想要搜索的记录的关键字>

Output: true/false

```
bool search(node* root, int target){
    if(root == NULL){
        return false;
    }
    node* ptr = root;
    while(ptr != NULL){
        if ptr 是叶子
            for i from 0 to 该叶子的记录数
                if target == 第i个记录中的key值
                    return true;
            return false;

        else
            index = 0;
            for i from 0 to 该叶子的记录数
                if 第i个记录中的key值 < target
                    index++

                else if 第i个记录中的key值 == target

                    return true;

                else
                    break;

            ptr = (ptr->child)[index];
}
```

3) 删除算法

本程序使用自底向上的删除方法，先删除叶节点中的记录以及可能出现在内部节点的所有 key 值，再进行叶节点的平衡调整，和内部节点的平衡调整操作。

① 若要进行删除操作的节点是叶子，在其中没有找到相同的 key 值，返回 false；若找到，且相同的 key 所在的位置不是最后一个，则删除

该记录，将其他记录左移一位，调整叶节点平衡。若找到，但是所在位置恰好是最后一个且该节点 key 数不为一，则要修改其父亲节点中对应的 key 值，若父节点要修改的 key 值也符合上述条件，则要一直循环，修改它爷爷节点的 key 值（改为倒数第二大的 key），直到到达根。

② 若要进行删除操作的不是叶子，则利用递归一直向下寻找，直到找到该 key 值可能存在的叶节点，之后调整内部节点平衡，返回结果。

伪代码

Input：root<B+树的根> ,target<想要删除的记录的 key 值>,

decelerator<延时器>

Output：true/false

```
bool remove(node*& root, int target, delayTime& decelerator){

    if root是叶子
        if 找到要删除的key的位置
            if 该位置不是第一且是最后一个
                记录第二大的key
                node* cur = root;
                while cur 不是根

                    找到待删除元素在其父节点的位置
                    更新父结点中的元素值
                        if  修改的父节点的关键字在其爷爷节点中也出现
                            cur = cur->parent
                        else
                            break;


            删除root节点中的第i个,将其右边的元素同步左移一位
            进入 leaf_balance(root, decelerator)函数，调整叶节点平衡
            return true;
        return false;//没有找到待删除的元素

    else
        for i from 0 to root中key的个数
            if target <= (root->key)[i]
                延时
                bool temp = remove((root->child)[i], target, decelerator);//一直递归，直到找到叶节点
                进入internal_balance(root, decelerator)函数，调整内部节点平衡
                return temp
        return false

}
```

相关函数

- leaf_balance

Input:root<B+树根结点指针的地址>, decelerator<延时器>

```
void leaf_balance(node*& root, delayTime& decelerator){
    if  如果叶子节点够  return;
    if 该叶节点是根
        if 该叶节点的元素已经被删完了
            root = NULL
            return
    node* parents = root->parent;
    确定root在其parent中的对应位置为index
    if index == 0//该节点在最左端，只能向右兄弟借
        延时
        if 右兄弟的元素够他借
            原节点新增一个记录
            右兄弟中的记录依次左移一位删除借走的记录
        else//右兄弟正好满足半满，不够给他借了，合并
            先将兄弟节点的记录增加到到原节点后
            删除右兄弟
            更新parent，key值向前移一位，child从第三个开始向前移一位，
            parent->number--
    else if index == parents->number - 1//若该结点在最右端,只能向左兄弟借
        延时
        if 左兄弟的元素够
            原节点中的所有记录先右移一位给借来的节点腾出空位
            将左兄弟的最后一个记录移到原节点中的第一个
            更新左兄弟、原节点以及他们的父节点的number
            更新父节点中倒数第二个key值为此时左兄弟的最后一个key

        else //左兄第元素不够，合并
            先将原节点中所有记录增加到左兄弟的记录之后
            删掉原节点
            更新父节点的number以及最后一个key值
    else //节点是中间节点
        延时
        if 左兄弟的元素够他借
            与上述向左兄弟借值类似
        else if 右兄弟的元素够他借
            与上述向右兄弟借值类似
        else //都不够给他借，和左兄第合并
            与上述和左兄第合并的操作类似
}
```

➢ internal_balance

Input:root<B+树根结点指针的地址>, decelerator<延时器>

```
void internal_balance(node*& root, delayTime& decelerator){
    if root 是根
        延时
        if root 只有一个key
            root 更新为其孩子节点，删除原root
        return
    if root 中的key数够 return
    找到root在其父节点中的对应位置
    if root是其父节点中的第一个子树
        延时
        if 右兄弟的元素够他借
            //先借一个过来
            右兄弟的第一个key值及其连带着的child指针加入到原节点
            改变借的key对应子树的父节点为原节点
            更新原节点number数
            更新父节点第一个key值
            右兄弟整体向左移一位
            更新右兄弟的number
        else//右兄弟正好满足半满，不够给他借了，合并
            先将右兄弟节点的元素复制给他
            删除右兄弟
            将父亲节点的所有key值左移一位，从第三个孩子开始，所有的child向前移一个
            更新父节点的number
    else if root是其父节点中最后一个子树
        延时
        if 左兄弟的元素够
            原节点中所有key和child同时向后移一位，为借来的元素腾出空位
            将左兄弟的最后一个key与child一起加入原节点
            更新两个节点的number
            更新父节点的number以及倒数第二个key的值
        else//左兄第元素不够，合并
            先将原节点的所有key和child增加到左兄弟节点中
            删掉原节点
            更新右兄弟的number以及最后一个key值
    else //root左右兄弟都存在
        延时
        if 左兄弟的元素够他借
            与上述向左兄弟借元素的操作相同
        else if 右兄弟的元素够他借
            与上述向右兄弟借元素的操作相同
        else //都不够给他借，和左兄第合并
            与上述和左兄第合并的操作类似
}
```

三． 程序使用和测试说明

1. 编译运行环境

本程序必须要在 Windows 系统下编译运行，因为其中用到了 windows 系统下的 Sleep 函数。直接编译运行即可。

2. 测试说明

（1）插入操作测试

本程序分为三个 test：test_insert, test_search, test_remove

a. Test_insert

首先，程序运行后会自动输出插入 50 个随机生成的记录结果，以及插入完成后对全树的遍历；如下图

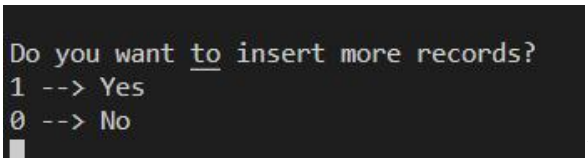
```
Try to insert 50 random record 1 s
Insert successfully 47 records!
The level traverse of the B plus tree is as follow.
```

```
The 1 level
node0:
   68  125  203  227

The 2 level
node0:
   30   45   61   68
node1:
   91  125
node2:
  140  154  203
node3:
  208  227
```

```
The 3 level
leaf0:
[  4 , 131 ]    [ 14 ,  25 ]    [ 22 ,  32 ]    [ 26 ,  56 ]    [ 27 , 141 ]    [ 30 ,   8 ]
leaf1:
[ 34 , 191 ]    [ 36 ,  26 ]    [ 37 , 197 ]    [ 38 , 211 ]    [ 45 , 156 ]
leaf2:
[ 54 , 131 ]    [ 57 , 180 ]    [ 60 , 205 ]    [ 61 , 147 ]
leaf3:
[ 62 , 210 ]    [ 64 , 161 ]    [ 68 ,  70 ]
leaf4:
[ 79 , 176 ]    [ 82 ,  88 ]    [ 83 , 179 ]    [ 91 ,  62 ]
leaf5:
[102 , 167 ]    [104 ,  23 ]    [122 ,   6 ]    [125 , 150 ]
leaf6:
[129 , 103 ]    [138 , 223 ]    [139 ,  11 ]    [140 , 181 ]
leaf7:
[145 , 107 ]    [147 ,  69 ]    [154 , 114 ]
leaf8:
[156 , 120 ]    [169 ,  96 ]    [175 , 224 ]    [200 , 116 ]    [201 , 123 ]    [203 ,  20 ]
leaf9:
[204 , 136 ]    [205 ,  25 ]    [208 ,  61 ]
leaf10:
[209 ,  24 ]    [211 ,  26 ]    [218 , 211 ]    [225 , 140 ]    [227 , 147 ]
```

其中叶子节点中的记录以[key, value]的形式打印，从遍历结果我们可以看出，对于第一层的所有关键字，都是对应第二层的每个节点中的最后一个关键字，在叶子节点中可以看到，所有关键字都是升序排列。考虑延时问题，插入五十个数据，成功 47 个，耗时应该是 0.94s,但这里的精确值只保留到秒，所以输出耗时 1s.

接着程序输出

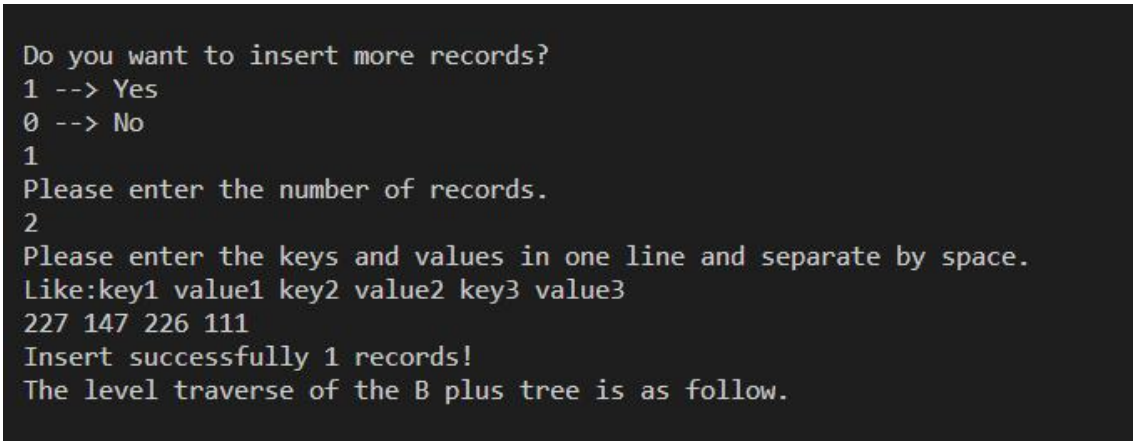


询问用户是否要尝试增添记录？

若想进行插入操作，则输入 1， 不想则输入 0；

这里输入 1，紧接着输入想要增添的记录个数，以及记录的 key 与 value

注：这里的输入要按照 key1 value1 key2 value2 的形式输入，空格间隔，回车结束



我们这里测试插入两个记录，一个是[227, 144] 在原树中存在，另一个是[226, 111]在原树中未出现，获得反馈：成功增添一个记录，符合预期，新树遍历如下(这里不全部展示，只看有改动的部分）

这里我们主要看第十个叶子中新增了[226, 111]这一记录，而[227, 144]未被改变

```
-------------Insert operation test is over.--------------------
```

接着，插入测试结束，进入查找测试

（2）查找操作测试

首先输入要查找的次数，这里进行 2 次，接着输入查找记录的 key，回车结束，这里测试了 214，出现在了叶子节点中，程序给出正确反馈，再次查找 215，原树中不存在，程序给出查找结果。

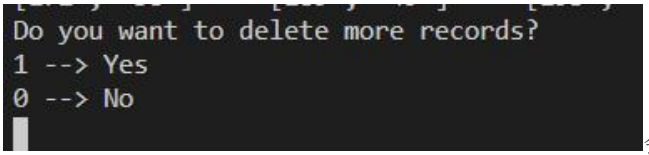
```
leaf9:
[214 , 166 ]    [222 ,  44 ]    [224 , 102 ]


Do you want to insert more records?
1 --> Yes
0 --> No
0
-------------Insert operation test is over.--------------------


------------------This is a search operation test------------------
Please enter the times that you want to do search operation.
2
Please enter the key of record which you want to search.
214
Find it! It's in the leaf node.
The key of record is 214 and the value of it is 166Please enter the key of record which you want to search.
215
Don't find!
------------Search operation test is over.-------------------
```

（3）删除操作测试

进入删除测试，程序先自动删除 100 个随机生成的关键字进行删除，并给出结果反馈和树的层次遍历

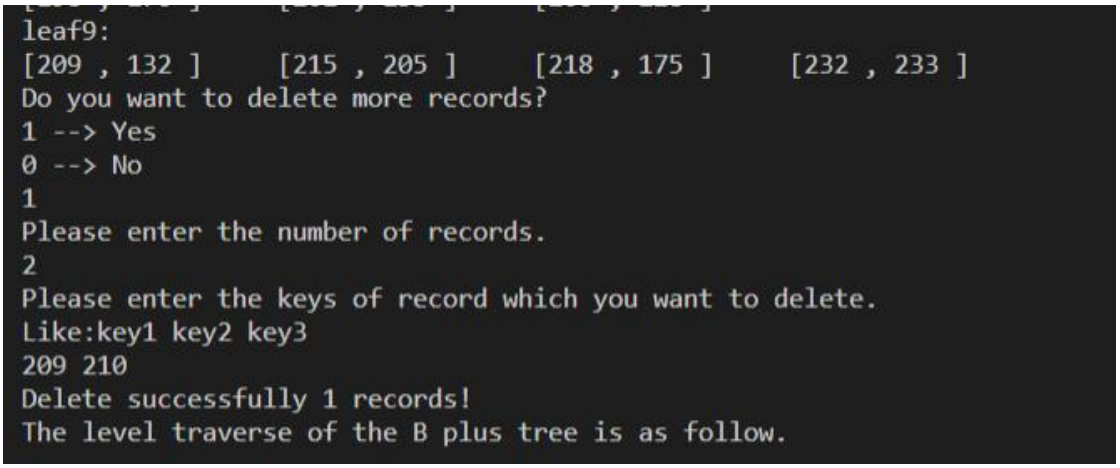
```
------------------This is a remove operation test----------------
Try to delete 50 random records, use 1.92s
Deelete successfully 7records.
The level traverse of the B plus tree is as follow.
```

接下来

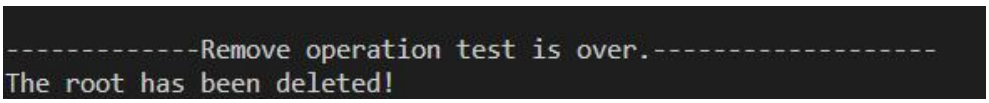


会询问你是否想自主删除记录？

若输入 1，则紧接着输入删除次数，以及你想要删除的记录的 key 值，这里测试两个，一个在树中出现，一个没有，程序返回正确



最后程序运行结束，清空树



四． 总结和讨论

在开始写程序之前，要做好充分的准备工作，充分理解 B+树的概念模型以及增删查找的具体操作。因为这个步骤没有做到位，一开始我们的插入算法写成了 234 树的自顶向下的插入方法，后来发现写错再重写浪费了很多时间。通过网上搜索资料，发现了两种 B+树的定义方法，一种是 M 阶 B+树的非根内部节点可以有 M-1 个 key 值和 M 个子树；还有一种是 M 阶的 B+树有 M 个 key 值和 M 个子树，一一对应，为方便起见，我们选择了后者对 B+树的定义。

模拟延时器时，通过上网查找资料，发现了 Windows 系统自带的可

以让程序短暂休眠的函数 Sleep，我们利用它，来模拟了系统的延时。

通过本程序，除了 B+树的节点分裂、合并等代码实现，我还学习到了随机数的生成函数，受益匪浅。

五． 参考文献

清华大学出版社严蔚敏 《数据结构》

百度百科 https://baike.baidu.com/item/B+%E6%A0%91/7845683

简书 https://www.jianshu.com/p/71700a464e97