### B+树的模拟

组队完成

**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

1. **引言**
2. **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。

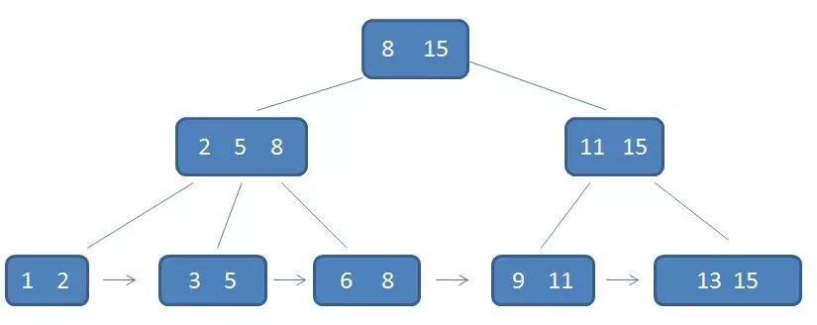
1. **解决方法**
2. **描述B+树的概念和B+树的逻辑结构**

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：

1. 有n棵子树的节点中含有n个关键字
2. 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。
3. 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。

1. **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

1. **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；

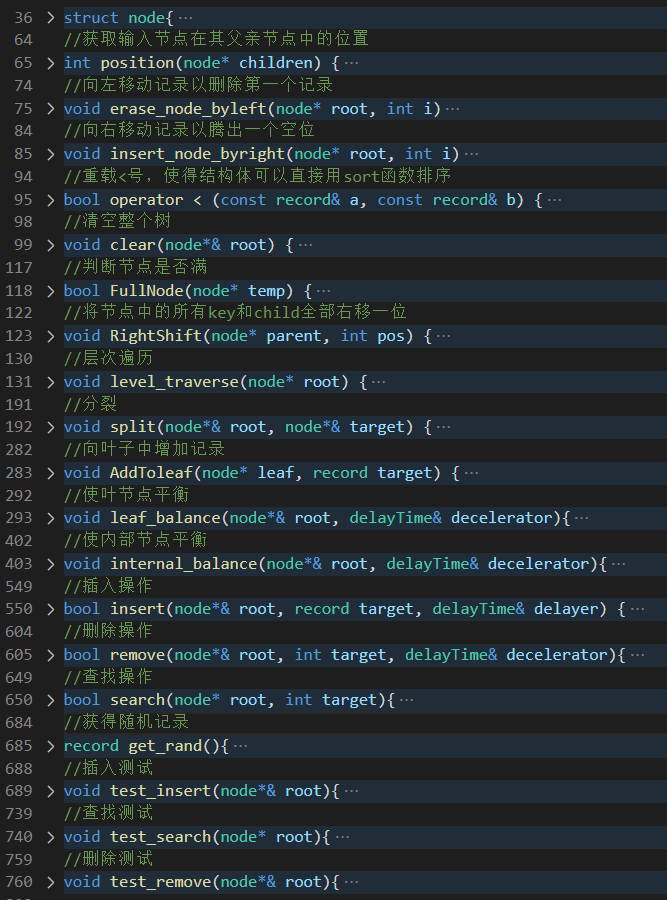
对于叶子节点：其重要存储leaf、number、parent、block，

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。

1. **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。

1. **列出源代码各个模块命名清单（不需要代码清单）**



1. **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

1. **运行结果分析，包括外部存取延时统计**

详见第三部分

1. **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。

1. **解决方法**
2. **输入输出的形式**

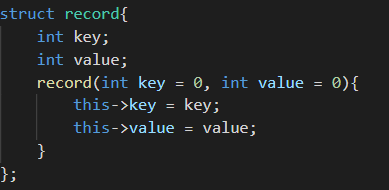
**输出：**插入、查找、删除操作的结果反馈，以及每做完一项操作之后对全树的层次遍历。

**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
4. **使用的数据结构**
5. **结构体**

我们共设计了两个结构体。

1. 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。



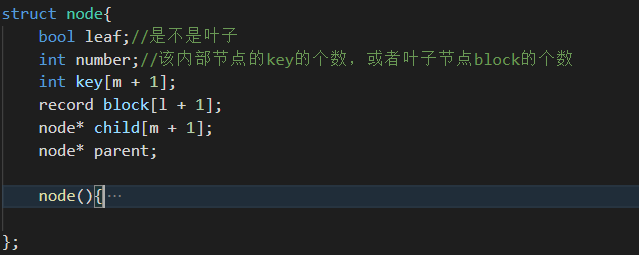
1. 第二个结构体node，即表示B+树中的每个节点。

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。

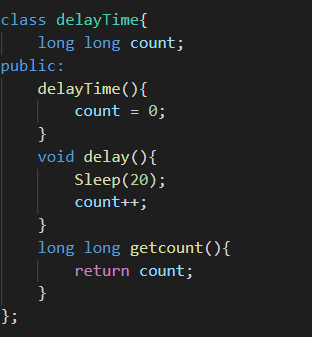
对于叶节点，number中存储record的个数；有block 存储记录；parent指向其父亲；数组 key 无效，无child。

除了数据的定义，该节点中还有一个node的初始化函数。



1. 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。



1. **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）。

1. **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。

1. **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。

1. **算法的描述**
2. **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**

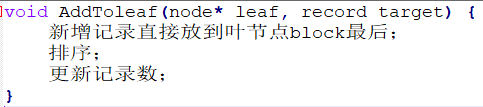
**Input: root<指向根结点指针的地址>, target<记录>, delayer<延时器>**

**Output: true<插入成功>/false<插入失败>**



* **Add To leaf函数**

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



* **Split函数**

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

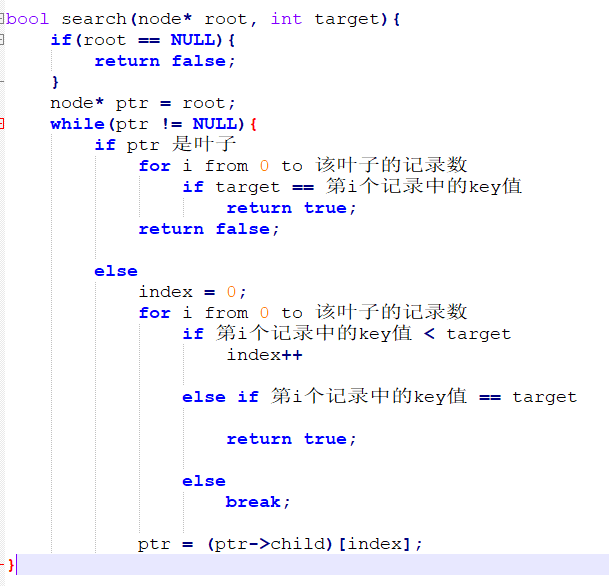


1. **查找算法**
2. 若根为空，直接返回false。
3. 若根不为空，且要搜索的节点是叶子，则从第一个record开始遍历直到找到与待寻找的记录的key值相同的key值，返回true，否则返回false；
4. 若根不为空，且要搜索的节点不是叶子，则在要搜索的节点key中找到第一个大于待搜索的key，并进入该key所对应的子树，再进行上述操作，做循环。若在该过程中遇到了相同的key值，则返回true。若一直找到了叶子节点，则进行操作②。

**伪代码**

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

**Output: true/false**



1. **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>**

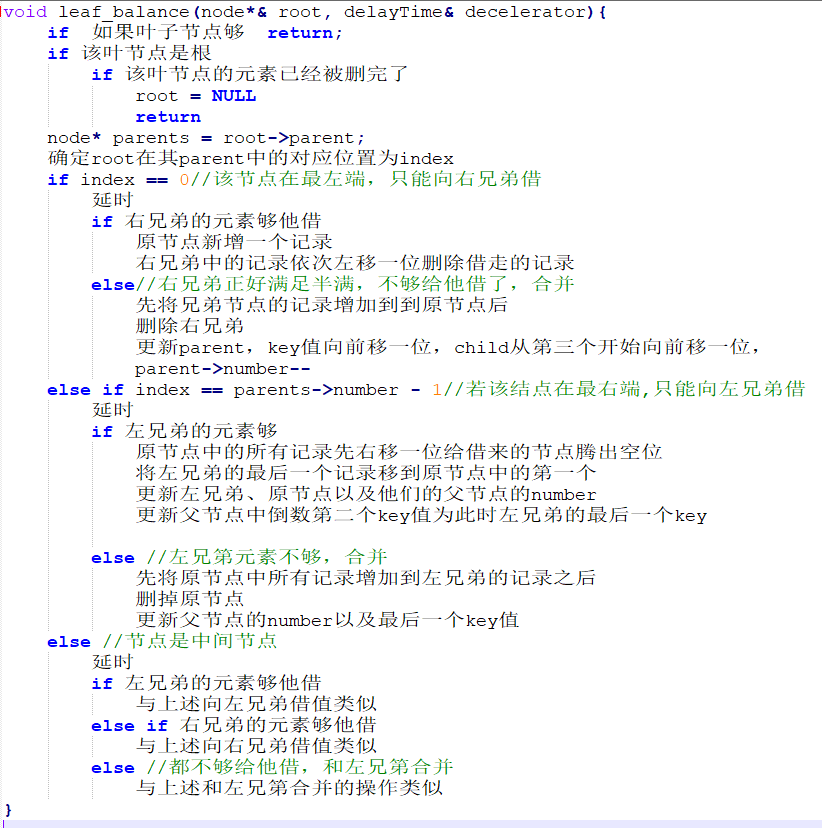
**Output：true/false**



相关函数

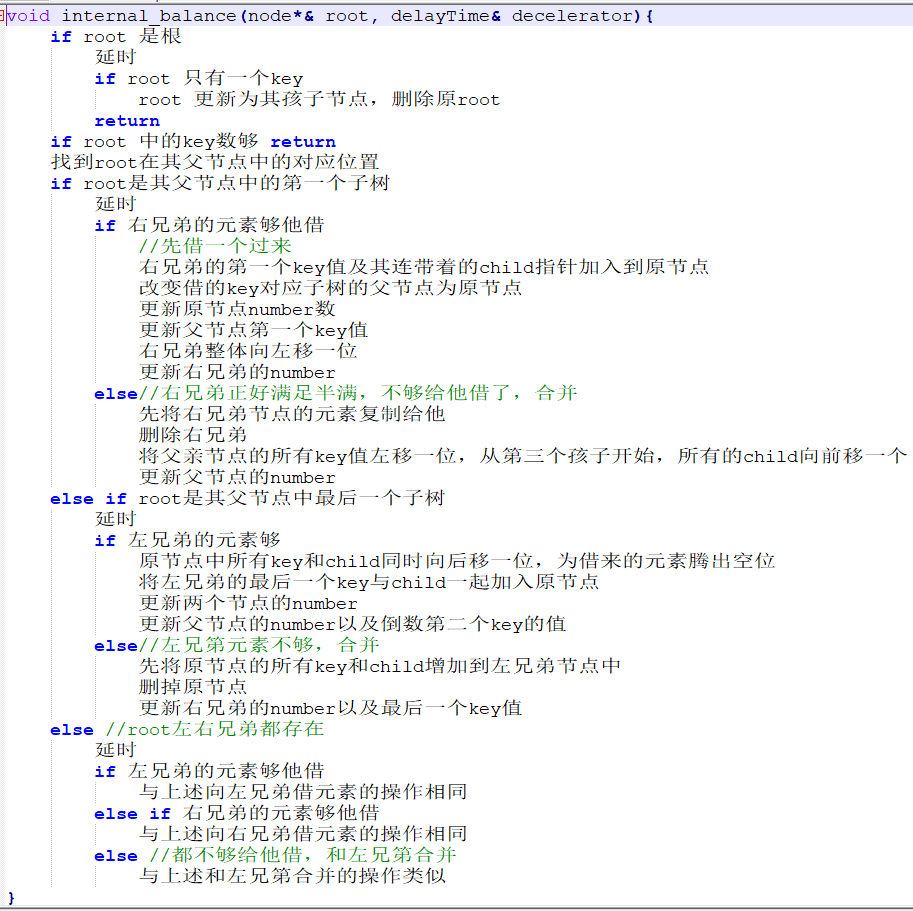
* leaf\_balance

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



* internal\_balance

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



1. **程序使用和测试说明**
2. **编译运行环境**

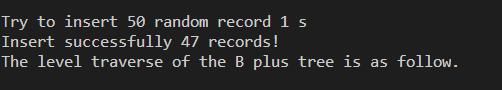
本程序必须要在Windows系统下编译运行，因为其中用到了windows系统下的Sleep函数。直接编译运行即可。

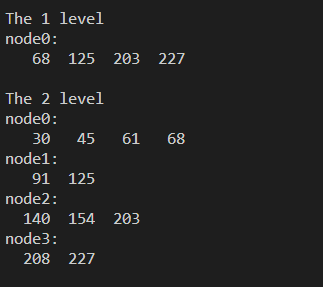
1. **测试说明**
2. **插入操作测试**

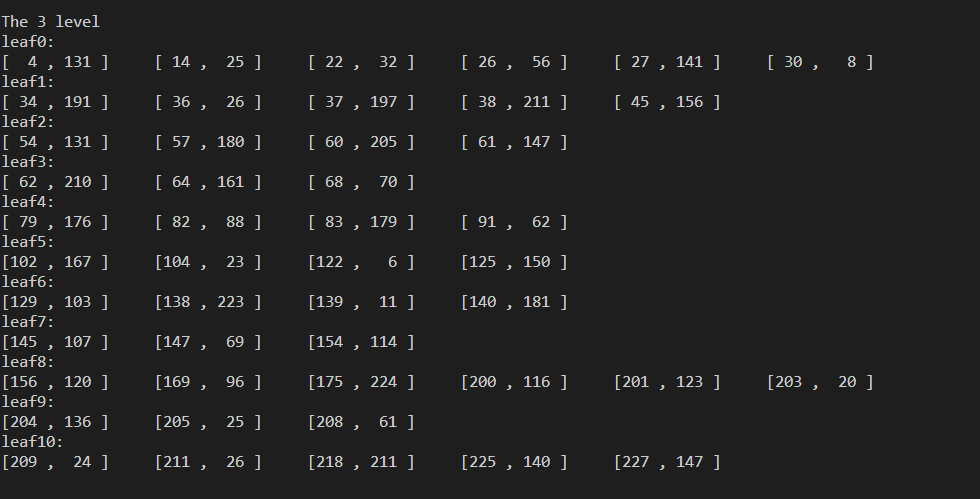
本程序分为三个test：test\_insert, test\_search, test\_remove

1. 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

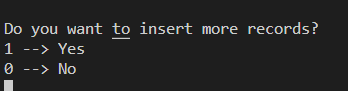






其中叶子节点中的记录以[key, value]的形式打印，从遍历结果我们可以看出，对于第一层的所有关键字，都是对应第二层的每个节点中的最后一个关键字，在叶子节点中可以看到，所有关键字都是升序排列。考虑延时问题，插入五十个数据，成功47个，耗时应该是0.94s,但这里的精确值只保留到秒，所以输出耗时1s.

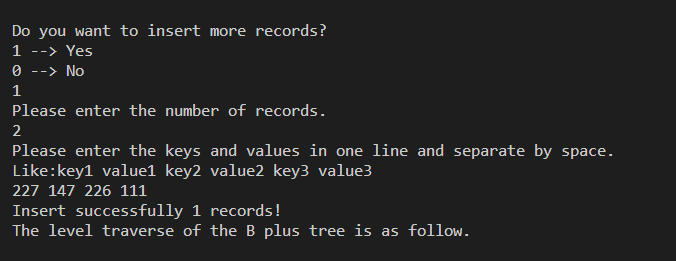
接着程序输出

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）



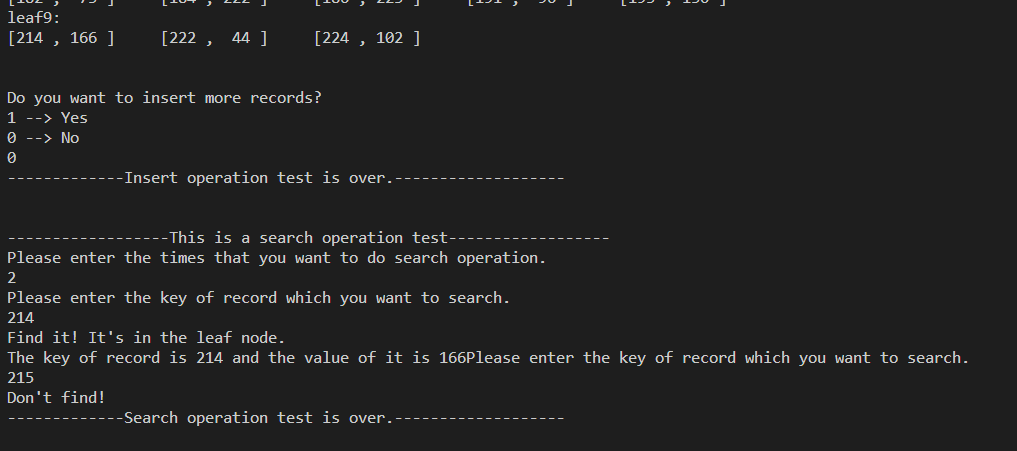
这里我们主要看第十个叶子中新增了[226, 111]这一记录，而[227, 144]未被改变



接着，插入测试结束，进入查找测试

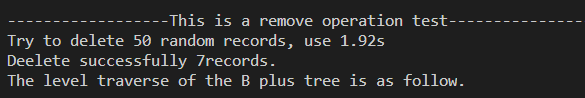
1. **查找操作测试**

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。

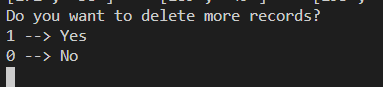


1. **删除操作测试**

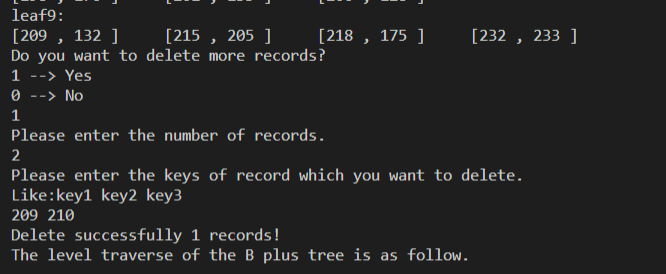
进入删除测试，程序先自动删除100个随机生成的关键字进行删除，并给出结果反馈和树的层次遍历



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



1. **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。

1. **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>