中山大学数据科学与计算机学院本科生实验报告

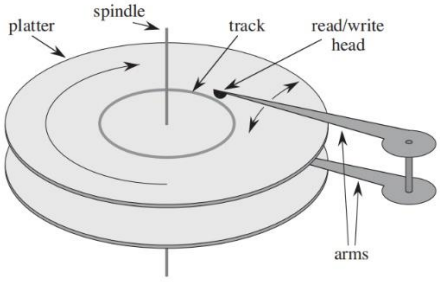
（2019-2020 学年第一学期）

课程名称：数据结构与算法分析　任课教师：乔海燕

年级&班级	18 级计科 8 班	专业(方向)	计算机类
学号	18340208	姓名	张洪宾
学号	18340215	姓名	张天祎

一、 引言

B+树是一种在文件系统和数据库中常用的存储结构。《算法导论》在介绍 B 树的时候介绍了磁盘的结构：



磁盘上数据必须用一个三维地址唯一标示：柱面号、盘面号、块号(磁道上的盘块)。如上图，在磁盘执行读/写功能的时候，磁盘不断转动，读写装置固定不动，当磁道在读/写头下通过时，就可以进行数据的读 / 写了。

读/写磁盘上某一指定数据需要下面 3 个步骤：

(1) 首先移动臂根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找 。

(2) 根据盘面号来确定指定盘面上的磁道。

(3) 盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

而算法导论也给出写作时的磁盘转动速度，大致是 7200RPM，即旋转一圈需要 8.33 毫秒。平均来讲我们访存的时候需要旋转半圈，仅考虑访存的第一阶段，

相比内存的处理来说也是一个巨大的开销。也就是说，如果访存次数过多，那在内存的处理时间可以近似地忽略掉。那我们对算法时间复杂度的追求就失去了没有意义。为了解决这个问题，我们需要采取一种比较合理的方法，使得访问存储器的数量尽可能地少。于是我们便用 B 树的一种变形 B+树的结构来对数据进行存储。

二、解决方法

问题 0. 关于描述 B+树的概念和 B+树的逻辑结构

B+树的定义有很多，我们参考了很多书，最终决定采用清华大学出版社严蔚敏老师的数据结构教材上定义的 B+树。书上的定义如下：

4. B^+ 树

B^+ 树是应文件系统所需而出的一种 B-树的变型树①。一棵 m 阶的 B^+ 树和 m 阶的 B-树的差异在于：

(1) 有 n 棵子树的结点中含有 n 个关键字。

(2) 所有的叶子结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，且叶子结点本身依关键字的大小自小而大顺序链接。

(3) 所有的非终端结点可以看成是索引部分，结点中仅含有其子树(根结点)中的最大(或最小)关键字。

书上也给出了一个具体的例子：

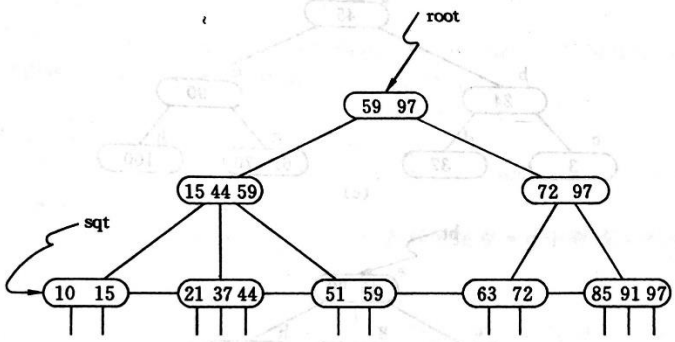
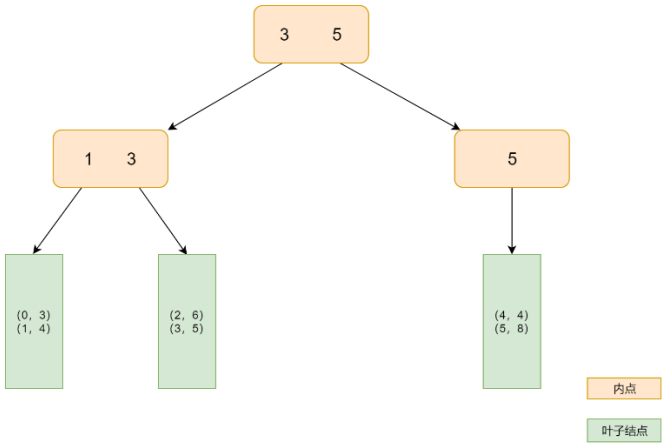


图 9.18 一棵 3 阶的 B^+ 树

但是在具体的作业中，因为并未要求线性查找，不需要将叶子结点本身依关键字大小自小而大顺序连接，所以我们将这一个性质去掉了，并不影响题目要求的性质。所以我们可以得到，一棵 m 阶 B+树最多有 m 个结点，至少有$\lceil m/2 \rceil$

个结点。而我们选择让所有非终端结点中的 key 值是子树（根结点）中的最大关键字。基于以上分析，可以这样规定：叶子结点中的括号是记录文件，即 key 和 value 组成键值对，可以通过 key 查找到 value。内点便是按上述结构层层递推。如下图就是这样一棵 B+树，M = 2，L = 2，符合上述定义。



问题 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）

我们要做的事情是模拟文件在读写的过程中读写外部存储块的过程，而老师将外部存储块的大小限制到 40Bytes，所以我们的目标是找到合适的 M 值和 L 值，使得在这个 40Bytes 的外部存储块中可以存储有关关键字、地址和记录信息数据。

无论是叶子结点还是内点，我们都需要能够根据该结点访问其父结点。这就意味着我们需要存储其双亲结点的地址。为了便于后面的设计，我们使用了一个 node 代表内点或者叶子结点，使用我们需要用一个 bool 值来判断它是否为叶子结点。

而为了准确计算 M 和 L 值，我们需要先设计出类的具体结构，然后再用类的具体结构来计算 M 和 L 值使得在 40 个 Bytes 内完成目标。我们最后的 M = 4，L = 6，计算过程在问题 2 中详细描述。

问题 2. 设计存储上述 B+树的数据结构设计 （程序设计语言描述）

我们为了让记录文件与关键字形成对应关系，我们用了一个 record 结构体来表示键值关系，并将关键字和记录文件作为整体插入 B+树中。

```
struct record {
    keytype key;//记录包括 key
    valuetype value;
    record(keytype key = 0, valuetype value = 0)
    {
        this->key = key;
        this->value = value;
    }
};
```

然后我们就要开始设计结点：首先为了简化程序的设计，我们决定让叶结点和内点共用一个结构体。这样就带来了一个问题，我们需要用一个 bool 值来判断它是不是叶结点。然后如果它是叶结点，我们就用一个数组存储 record 类型变量，这个数组最多由 L 个元素，并且我们需要知道这个数组有多少个元素。而如果它是内点，就相对会复杂一些，它要存储 n 个关键字和 n 个地址(n <= m)，同样的，我们需要找到这个 n 的值。所以在两种情况下我们的结点都需要再给结点定义一个计数变量。然后我们为了方便各种操作我们需要找到该结点的双亲结点，因为我们在插入，删除的时候都会回溯到双亲结点进行操作。所以我们定义的结点结构体如下：

```
struct node {
    bool leaf;//判断是否为叶节点
    unsigned char number;//有效的 key 数
    //多的一个数组空间仅在分裂时使用，最大存放数仍为 m，l
    keytype key[m+1];//存放 key
    record block[l+1];//存放含 record 的块
    node* child[m+1];//孩子指针
```

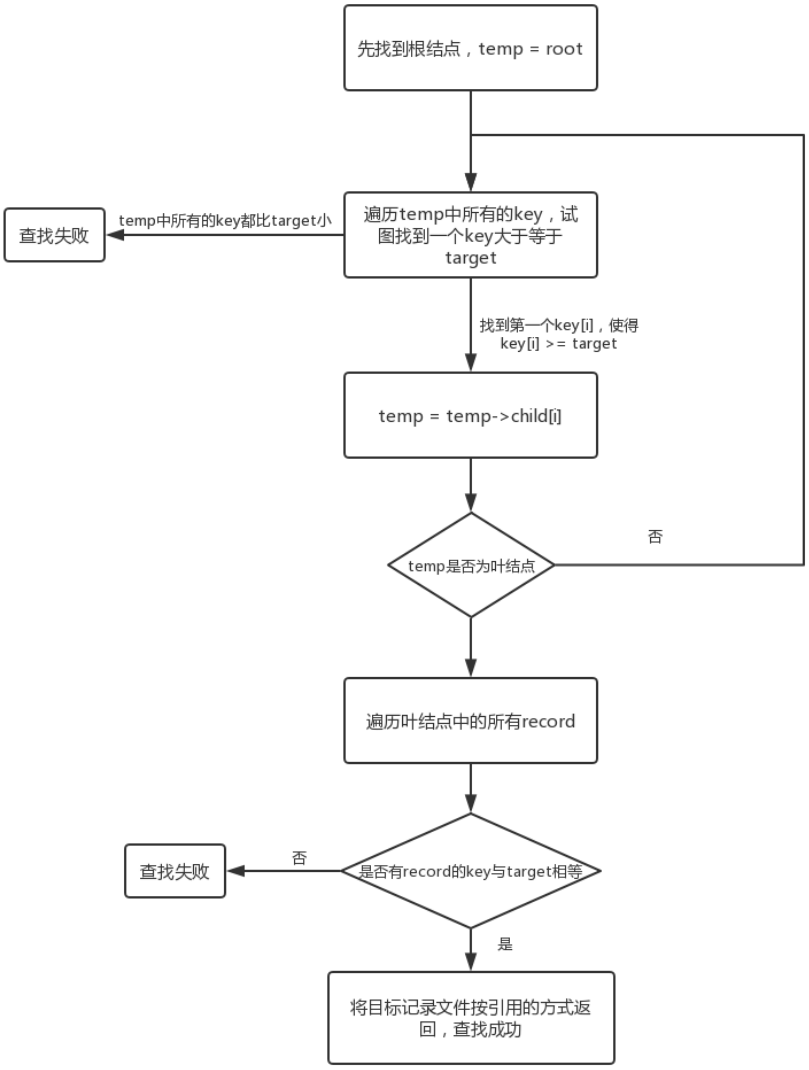
```
    node* parents;//父指针
    node()
    {
        leaf = 0;
        number = 0;
        parents = nullptr;
        for (int i = 0; i < m+1; i++)
        {
            key[i] = 0;
            child[i] = nullptr;
        }
        for (int i = 0; i < l+1; i++)
        {
            block[i] = record();
        }
    }
};
```

可以看到，我们为了节省空间，并且考虑到 M 和 L 值不会太大，我们用一个字节的 unsigned char 来表示结点的计数变量。为了让叶结点存储比较合理的数量的记录文件，我们决定使用两个外部存储块来存储叶结点。对于叶结点，它含 leaf、number、block、parents，可以推导出$L = \frac{80-1-1-4}{8+4} = 6$。对于非叶节点，它含 leaf、number、key、child、parents，可以推导出$M = \frac{40-1-1-4}{4+4} = 4$，满足题目要求，于是我们的 M = 4，L = 6。

问题 3. 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）B+树查找算法：

先从根出发，类似于二叉查找树的查找方式，只是每次处理一个结点的时候需要先找到下一个儿子结点对应的指针，然后重复查找内点直至找到对应的叶结点。具体的流程图如下：



查找算法流程图

2）B+树的插入算法：

插入算法需要分成三种情况讨论，我们先用与查找相同的方法找到插入的目标结点后，再进行接下来的操作：

（1） 插入记录文件的目标叶结点未满

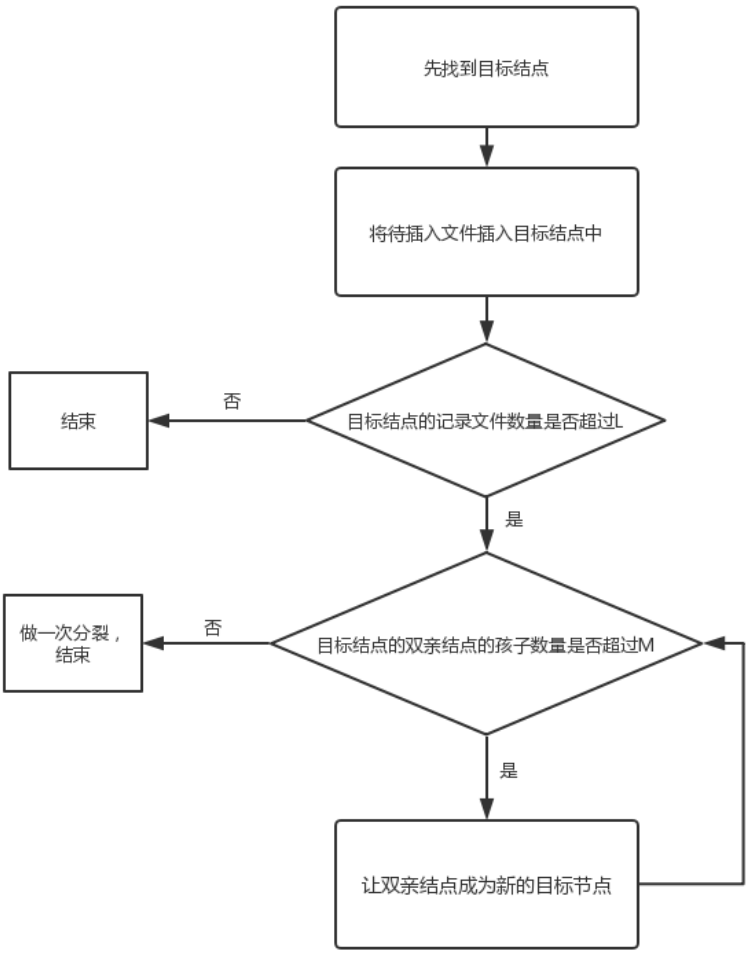
（2） 插入记录文件的目标叶结点已满而目标叶结点的双亲结点未满

（3） 插入记录文件的目标叶结点已满且目标叶结点的双亲结点已满

对于第（1）种情况，非常简单，只需要直接将待插入的记录文件插入目标叶结

点并重新排序即可。对于第（2）种情况，只需要将记录文件插入目标叶结点后做一次分裂就可以了。对于第（3）种情况，我们需要分裂叶结点之后再分裂双亲结点，并不断进行下去直到我们处理的结点不需要分裂为止，即向上处理直至重新满足B+树的定义才停止。

插入操作的流程图如下：

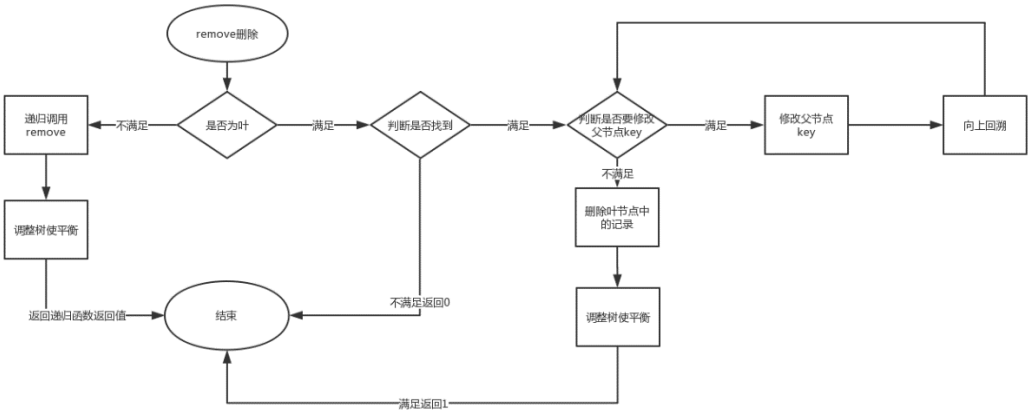


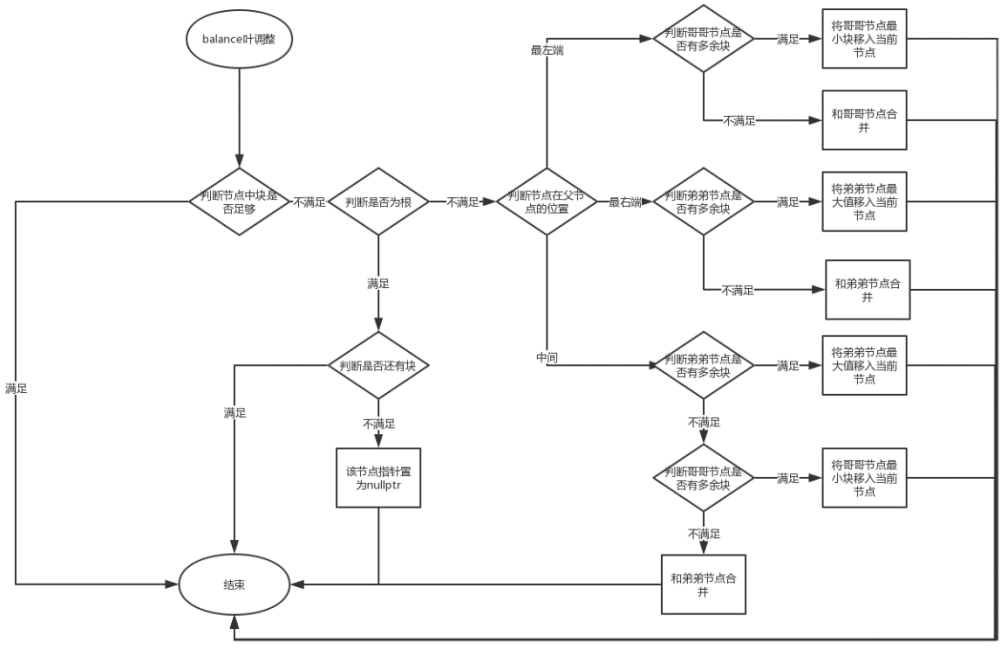
插入算法流程图

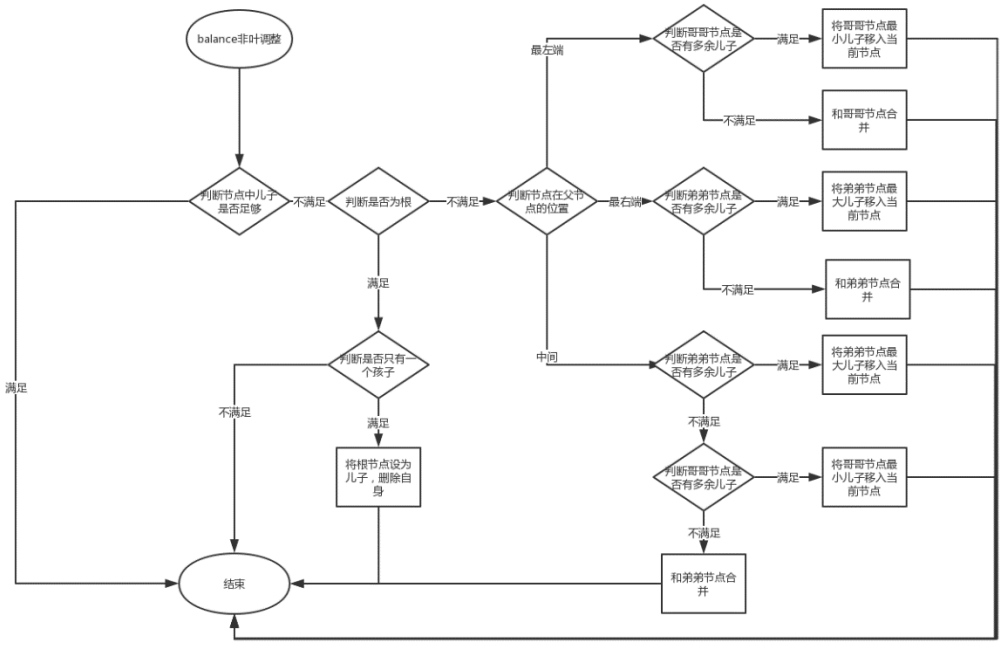
3）B+树的删除算法：

本程序使用 bottom-up 方法，先通过树寻找 key 所在的叶，再删除 key 对应的记录并修改需要修改的所有父节点中的 key 值。再看此时的节点是否满足 B+

树的记录数要求，若不满足则根据叶在父节点中的位置考虑借或合并的情况，还需要特别考虑叶为根的情况。非叶节点的调整同理，通过递归逐层向上，根的儿子若为一个，则收缩根，树高下降。流程图如下：







4）20ms 延时器模拟一次外部存取的时间延迟：

在引言部分我们也看到，我们做一次访存需要比较长的时间，于是我们设计了一个 20ms 延时器来模拟访存的过程。我们定义了一个类，用于调用系统自带的延时函数。在这里我和我的队友产生了分歧，我想要使用 Mac OS 下的 Unix 内核下的系统调用，这个调用也可以被 Linux 使用，我的队友坚持使用 Visual Studio 开发，这就需要调用 Windows 操作系统的系统调用。这两个系统下实现这一部分的代码略有不同，于是我写了两个版本，一个 Linux 版，一个 Windows 版，以 Linux 版为例：

```
//定义的时间延迟类
class delay_count{
    long long int count;
public:
    delay_count(){
        count = 0;
    }
    void delay(){
        usleep(20000);//Linux 系统调用，暂停 20000 微秒，即 20ms
        count++;    //对暂停的次数进行计数，便于最后统计暂停的总时间
    }
    long long int get_count(){
```

```
        return count;
    }
};
```

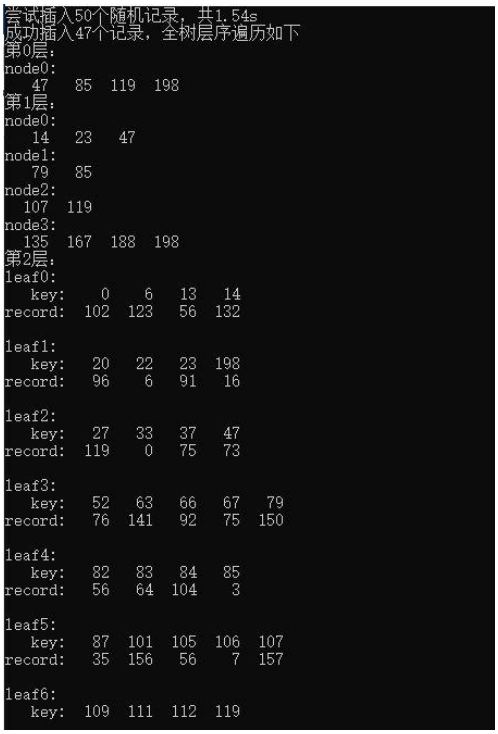
然后我们就在插入和删除的函数中加入一个 delay_count 类型的引用，然后再需要访存的地方调用该对象的 delay 函数,就可以实现延时和对延时的计数，然后最后取出该对象的 count 值再乘以 20ms 即可计算出访问的延时。

问题 4. 列出源代码各个主模块命名清单（不需要代码清单）

```
bool search(node*root,keytype the_key ,record & target);
bool insert(node*& root, record target, delay_count& retarder);
bool remove(node* &root, keytype target,delay_count&retarder);
```

问题 5、6. 测试用例设计（初始化至少包含 50 个记录数据）、运行结果分析，包括外部存取延时统计

在 main 函数中调用 test 函数。首先通过随机数生成 50 个 record，然后依次插入（如果 key 重复则插入失败），再显示该操作的时间并打印树（层序遍历）。如下：



可以看到成功插入 47 个，耗时 1.54s。叶中 record 按 key 由小到大排序，下层最大 key 出现在上层 key 中。

```
leaf6:
   key:  109  111  112  119
record:  185  144  178   54

leaf7:
   key:  120  123  131  135
record:   57  169   17  172

leaf8:
   key:  139  148  154  156  164  167
record:  143  104  108  143  147  171

leaf9:
   key:  169  174  175  188
record:  108  164   17   13

leaf10:
   key:  189  197  198
record:   79   35  170
```

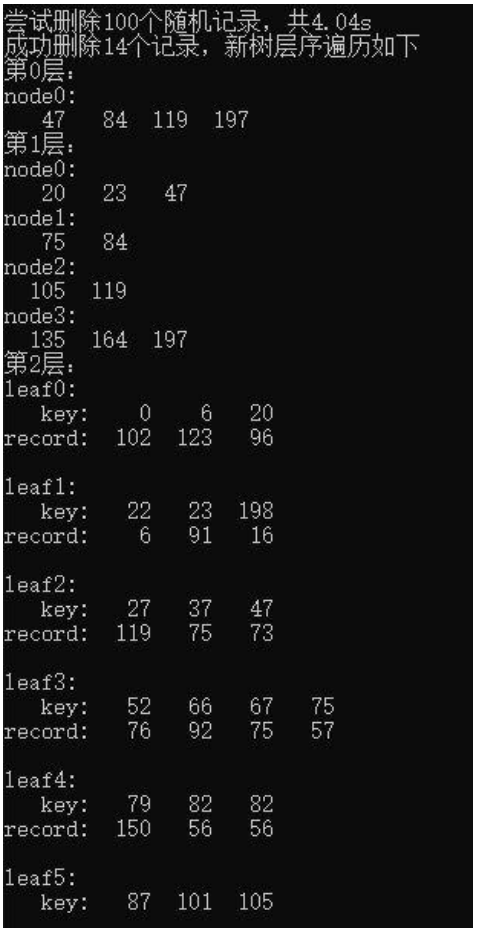
然后插入一个随机 record（循环直至成功），显示插入它的时间并打印树。

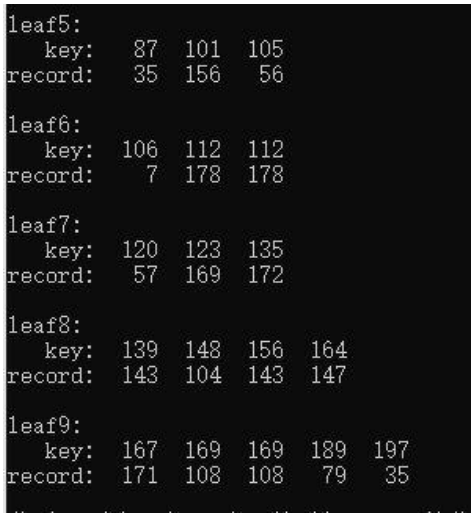
```
此时再插一个随机记录（75,57），花费0.04s
新树如下
第0层：
node0:
   47   85  119  198
第1层：
node0:
   14   23   47
node1:
   79   85
node2:
  107  119
node3:
  135  167  188  198
第2层：
leaf0:
   key:    0    6   13   14
record:  102  123   56  132

leaf1:
   key:   20   22   23  198
record:   96    6   91   16

leaf2:
   key:   27   33   37   47
record:  119    0   75   73

leaf3:
   key:   52   63   66   67   75   79
record:   76  141   92   75   57  150

leaf4:
   key:   82   83   84   85
record:   56   64  104    3

leaf5:
   key:   87  101  105  106  107
record:   35  156   56    7  157
```

```
leaf6:
   key:  109  111  112  119
record:  185  144  178   54

leaf7:
   key:  120  123  131  135
record:   57  169   17  172

leaf8:
   key:  139  148  154  156  164  167
record:  143  104  108  143  147  171

leaf9:
   key:  169  174  175  188
record:  108  164   17   13

leaf10:
   key:  189  197  198
record:   79   35  170
```

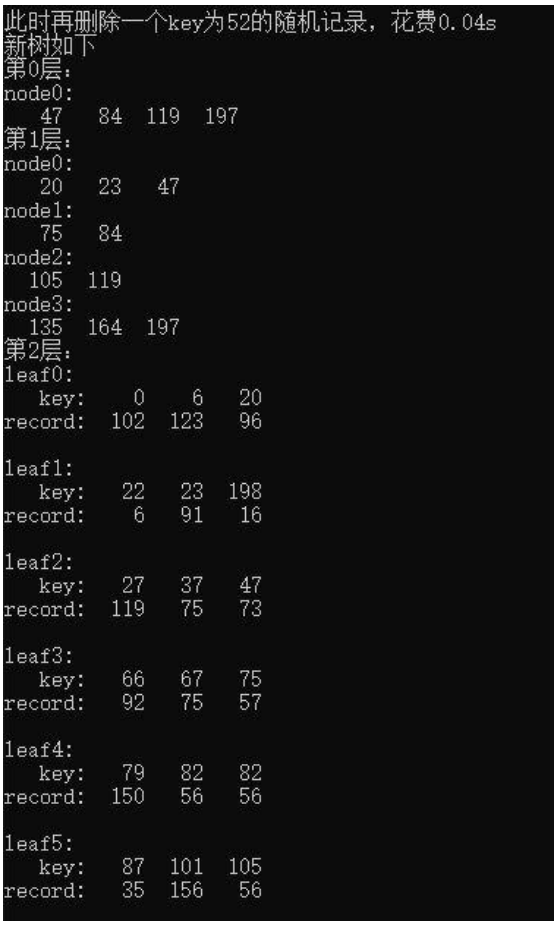
可以看到该记录被插到 leaf3 中，耗时 0.04s。

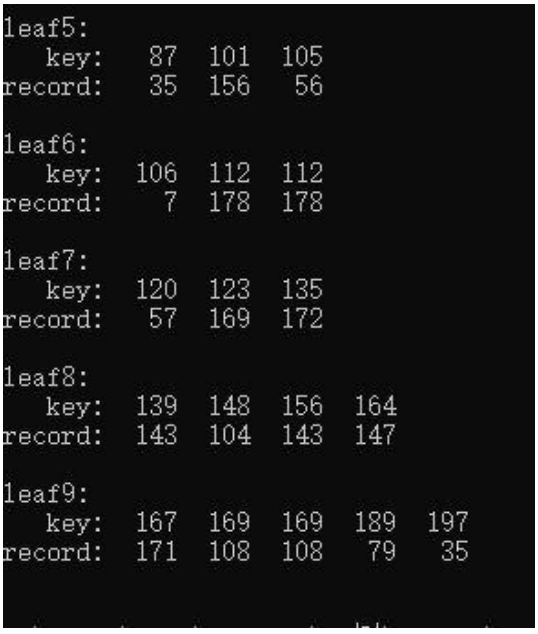
再通过随机数生成 100 个 record，然后依次删除（如果 key 不存在则删除失败），再显示该操作的时间并打印树（层序遍历）。





尝试删除 100 个记录耗时 4.04s。然后删除一个随机 record(循环直至成功)，显示插入它的时间并打印树。





可以看到 leaf3 中 key 为 52 的节点被删除了，耗时 0.02s。

三、程序使用

我们提供了两个版本，一个 Linux 版，可以再 Mac OS 和 Linux 下用命令行运行，一个是 Windows 版，基于 Visual Studio 运行。具体运行方式见各自文件夹中的 read_me 文件。

四、总结和讨论

这次实验在 B+树的实现上张洪宾负责查找和插入的实现，张天祎负责实现删除，然后其他细节上，张洪宾负责写延时，张天祎负责写测试函数，在写实验报告的时候我们则是各自写自己负责的模块再进行整合。而我们在开发环境上有一定的分歧，所以我们有两个版本的代码，主要是再系统调用上有所差异。

总体来说这次实验难度较大，花费时间很多，不过两个人再确定了树的结构之后，效率相对高了一些。今后还要继续提高团队合作能力，才能写出更好的代码。

五、参考文献

[1] 严蔚敏，吴伟民.数据结构（C 语言版）[M]. 北京：清华大学出版社，2007

[2] 数据结构与算法实验实践教程，乔海燕、蒋爱军、高集荣和刘晓铭编 著，清华大学出版社出版，2012