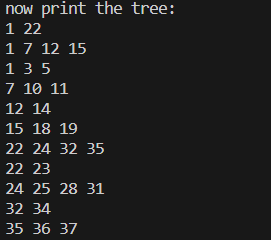
B+树的基本操作及其序列化与反序列化

## 实验目的

通过对B+树增删查改的基本操作进行代码的实现，以及实现B+树的序列化和反序列化来使自己对于B+树这一数据结构的掌握更加深刻并提高自己的编程能力。

## 实验内容

1. B+树的查找

对于 B+树中数据的查找，根据需要查询的key值对B+树从根结点向下查找。非叶子结点存储着其子结点中的最小值以及子结点的指针，以此来确定查询的key值位于那个叶子结点当中。当从根结点递归到叶子结点以后，在叶子结点中进行值的查找。叶子结点中没有值与查询的key值相匹配，则查找失败。有值相匹配则查找成功，就会显示出key值及其对于的value值。如下图：  


这是将一棵B+树以类似于中序遍历的方式打印出来，就是先打印根结点，然后是根结点的左孩子，接着右孩子的方式。如果我们要查询的key值为1，那么B+树会递归到包含1的叶子结点，也就是第三行。叶子结点中存储的key值有1，3，5 。程序会输出key所对应的value值。如下图：



如果查询不存在的值，例如查询key=4，同样B+树会递归到存有值1，3，5的叶子结点中，但叶子结点中没有key=4，则查询失败。如下图：

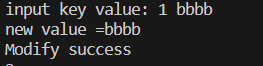


1. B+树中value值的修改

对于B+树中key值对应的value值进行修改，先找出叶子结点中key值对应的value值的指针，再修改其中内容即可。若叶子结点中无对应的key值，则会修改失败。

如下图修改key=1的value值：







修改后查询key=1的value值时，value值就从aaaa变成了bbbb。

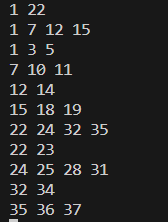
若要修改不存在的key值对应的value值，如下图：



B+树中不存在key=4的值，故程序报错，显示B+树上没有此key值。

1. B+树数据的添加

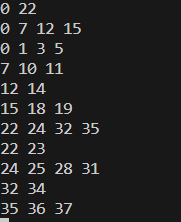
B+树数据的添加以键值对（key，value）的方式进行添加，在叶子结点上寻找key值存在的合适位置进行添加。需要注意的是如果添加的key值使叶子结点中的最小key值发生改变，则需要对应的修改其父结点中的key值。如下图添加key=0的键值对：



上图为原先B+树的各结点值，当插入key=0时：



对应的叶子结点及其父结点存储的值也会发生相应的改变如下：

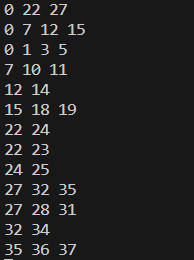


可见插入key=0后，根结点的第一个key由1变为了更小的0，根结点下面的子结点也发生了相应的改变。

实验中构建的 B+树阶数为5阶，也就是结点存储的key值最多为4个，若有叶子结点存储key值超过5个，则会进行分裂。由插入key=0后构成的B+树中，我们可以发现有一个叶子结点存储了4个key值，为24，25，28，31.现在我们插入key=27，如下图：



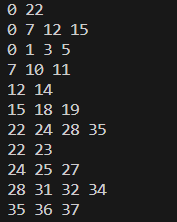
B+树结构改变如下：



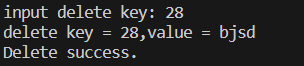
可见插入key=27时，叶子结点进行分裂，同时将中间的值key=27传递给了父结点，父结点原先存储值为22，24，32，35 。在key=27插入后再次分裂，然后将key=27传入了根节点。

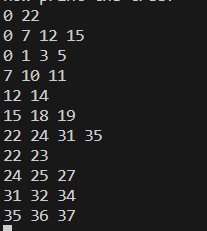
1. B+树的删除操作

对key值进行删除时，若删除的是一个叶子结点中最小的key值时，需要修改其对应父结点中的key值。如下图中的B+树，我们删除key=28。



删除后：

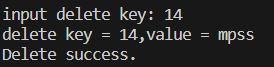


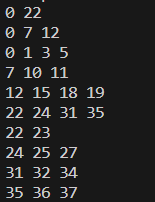


可见原先的叶子结点key值为28，31，32，34 。对应的变为了31，32，34。并且父结点中的key值由22，24，28，35变为了22，24，31，35。其中的28变为了31。

若删除key值后对应的叶子结点存储的key值数量不符合要求，则会向左右孩子进行解调，若左右孩子因此借调后也无法满足要求，则会进行融合。

如上图的B+树结构所示，若删除key=14的结点，则叶子结点中的值12，14只剩12，与右兄弟融合后也不会超过5阶 B+树的限制，故融合。





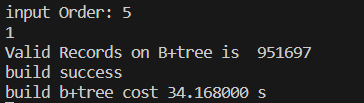
叶子结点与右兄弟融合，变为12，15，18，19。其父结点的值由0，7，12，15变为了0，7，12。

1. 相同数据不同阶数B+树基本操作花费时间比较

随机生成了100万对键对值来进行不同阶数B+树进行基本操作时花费的时间比较。

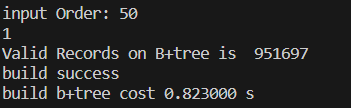
1. 5阶B+树和50阶B+树构建花费时间的比较

将100万对键对值用5阶B+树来存储，花费时间如下图：



因为随机生成的数据有一部分key值是重复的，实际上只插入了951697对键对值。花费时间为34.168000 s。

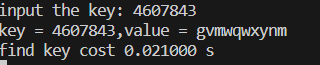
将100万对键对值用50阶B+树来存储，花费时间如下图：



花费时间仅为0.823000 s。可见存储大数据文件时，B+树阶数越高，构建B+树时间就会大大缩短。

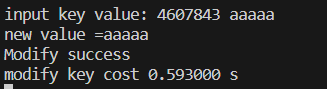
1. 5阶B+树和50阶B+树相同数据进行基本操作花费时间的比较

下图为5阶B+树查询key=4607843，此为随机生成数据文件的最后一行数据。



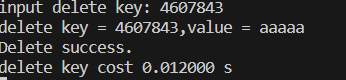
花费时间为0.021000 s。

修改其value值：



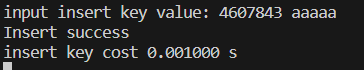
花费0.593000 s。

对该值进下删除操作：



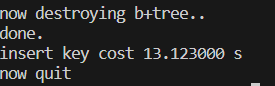
花费0.012000 s。

进行插入操作：



花费时间为0.001000 s。

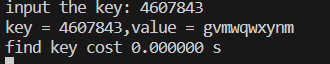
销毁树：



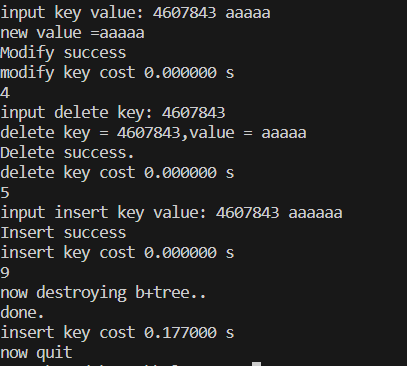
花费13.123000 s。

接着为50阶B+树进行相同操作时的时间：

查找：



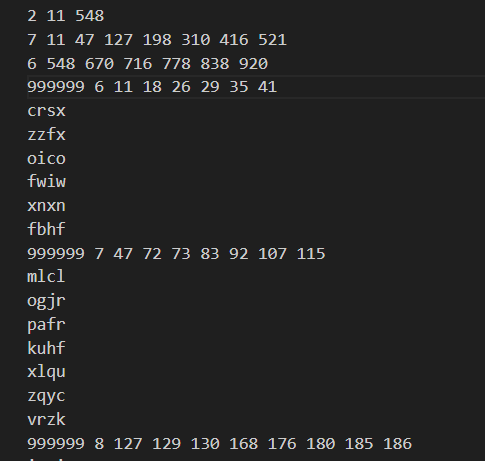
貌似耗时太短无法得出时间。下列操作为修改，删除，插入以及树的删除。



时间耗时都太短了，故显示为0.000000s 。树的销毁花费了0.177000s。可当我采用1000万数据时，在构建5阶B+树时则要出现系统崩溃。显然面对大数据文件，阶数越高就越能减少基本操作需要花费的时间。

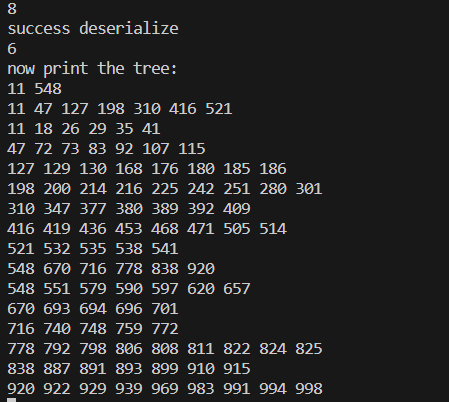
1. B+树的序列化和反序列化

不知道自己理解的对不对，我是把B+树的序列化理解为将B+树用一个文件存储起来。反序列化则是将文件还原成B+树。在我自己设计的序列化算法中，我使每一个非叶子结点的key值占据文件中的一行位置。而对于叶子结点，则会有一行存储key值，另外会有几行存储value值。下图为一个100个数据的10阶B+树的序列化部分文件：



遍历B+树的结点，然后每个结点为一行。如果存储结点key值的第一个数据为999999，则表明是叶子结点。反之则是该节点含的key值个数。后面则是key值。若文件的一行第一个数据为999999，则表明此行对应的结点为叶子结点。则第2个数据为结点key值个数，后面依次为key值，然后下面几行为key值对应的value值。

反序列化则是每读一行，若知道该行不是叶子结点，则每读一个key值就开辟一个结点，结点中key对应的value就负责存储结点的地址。将结点放入队列中等待遍历，叶子结点则不需要开辟空间。



上图是将文件反序列化后打印出来的B+树，和初始树一样，增删查改也能够进行。

## 实验收获

实现了B+树的基本操作与序列化和反序列化，提高了代码的编写能力。明白了面对大数据存储时，提高B+树的阶数能够有效的缩短建树以及增删查改需要花费的时间。也第一次尝试了序列化与反序列化。明白了序列化后B+树阶数就固定了，反序列化后仍是固定的阶数。