哈尔滨工业大学

**<<数据库系统>>**

**实验报告**

**(2020年度春季学期)**

|  |  |
| --- | --- |
| **姓名：** | **史纪元** |
| **学号：** | **1173300919** |
| **学院：** | **计算机科学与技术学院** |
| **教师：** | **史建焘** |

实验二 数据库索引及查询算法实现

## 一、实验目的

掌握B树索引查找算法，多路归并排序算法，并用高级语言实现

## 二、实验环境

Windows 10，Python 3.6，Pycharm

## 三、实验过程及结果

**1. 生成数据**

使用random.sample()函数在0~0xFFFFFFFF中随机生成10^6个整数，每个整数后接一个长度为12个字符的随机生成的字符串，生成字符串的函数为：

def random\_string(string\_length):  
 letters = string.ascii\_lowercase  
 return ''.join(random.choice(letters) for k in range(string\_length))

然后以每个条目4B整数+12B字符串的格式二进制写入data文件中，完整代码为：

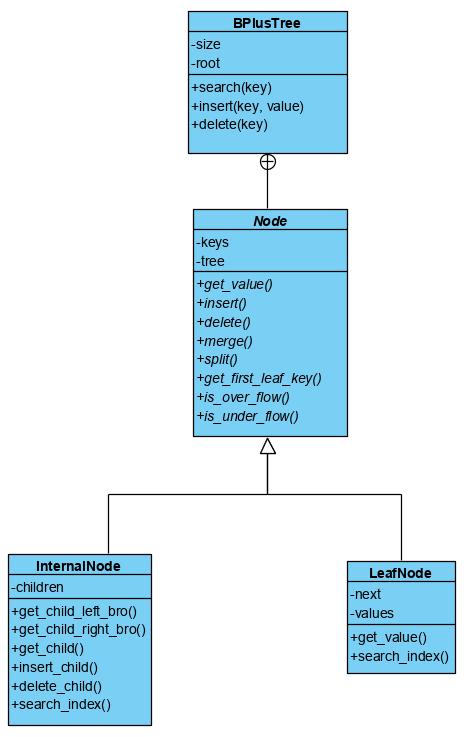
number = random.sample(range(0xFFFFFFFF), 10 \*\* 6)  
 items = []  
 for i in range(10 \*\* 6):  
 items.append((number[i], random\_string(12)))  
 with open('data', 'wb') as f:  
 for item in items:  
 f.write(item[0].to\_bytes(4, byteorder='big'))  
 f.write(item[1].encode(encoding='utf-8'))

得到16B \* 10^6大小的文本文件，用WinHex查看：

**2.B+树索引**

本次实验我选用的是B+树索引。

因为B+树的叶结点比非叶结点多了一个指向下一个叶结点的属性，因此我分别定义了B+树类BPlusTree、结点抽象类Node、内部节点类InternalNode和叶结点类LeafNode。各类的属性、方法和关系如下：



B+树初始化时，首先初始化阶数size，然后初始化一个叶结点，并将根节点置为该叶结点。B+树类的search、insert和delete方法委托给了InternalNode和LeafNode来实现。

抽象类Node定义了叶结点和内部节点通用的属性keys、tree（当前节点所属的B+树）和insert、delete、get\_value等方法。

**2.1 InternalNode（非叶节点）类**

内部结点类继承Node抽象类，并新增了children属性，该属性用来保存一个内部节点的子结点。

函数介绍：

search\_index(key)找到self.keys中能够插入key的下标并返回该下标。

def search\_index(self, key):  
 i = 0  
 for k in self.keys:  
 if k > key:  
 break  
 i += 1  
 return i

get\_child(key)找到可能包含key的子结点并返回，由于B+树的非叶结点不保存具体的value，因此该方法一定能找到一个可能包含key的子结点，但具体能不能找到key对应的value未必，要看叶结点的情况。

def get\_child(self, key):  
 i = self.search\_index(key)  
 return self.children[i]

get\_value(key)方法根据输入key返回B+树中key对应的value。具体实现为首先找到包含该key的子结点，然后再递归调用get\_value(key)方法。

def get\_value(self, key):  
 return self.get\_child(key).get\_value(key)

在插入和删除结点导致结点分裂的情况下，需要获得当前key的最左子结点插入到self.children中。因此定义了get\_first\_leaf\_key方法，非叶结点的该方法通过递归调用最终委托给了叶结点：

def get\_first\_leaf\_key(self):  
 return self.children[0].get\_first\_leaf\_key()

在合并结点时，需要分别获得一个结点的左右兄弟，以判断向左还是向右合并，因此定义了获得一个child的左右兄弟的方法，若某结点存在左/右兄弟，则方法会返回该兄弟结点，否则返回None。

def get\_child\_left\_bro(self, key): # 返回key对应子结点的左兄弟结点  
 i = self.search\_index(key)  
 if i == 0:  
 return None # 无左兄弟返回None  
 return self.children[i - 1]  
  
def get\_child\_right\_bro(self, key): # 返回key对应子结点的右兄弟  
 i = self.search\_index(key)  
 if i == ORDER:  
 return None # 无右兄弟返回None  
 return self.children[i + 1]

**插入操作(insert)**的参数为(key, value)，首先找到key应该插入到的子节点，然后调用该子节点的insert方法，即递归调用，最终插入到叶结点的value中。叶结点的insert方法将在LeafNode类部分讨论。

child = self.get\_child(key)  
 child.insert(key, value) # 递归,最终插入到叶结点

然后从叶节点开始判断是否有溢出情况，若有结点的key值多于B+树的阶数，则需要分裂该结点，然后获得分裂得到的新结点的最小的key（最左边），并将该key和新结点插入到当前结点的children列表中。由于insert方法是递归调用的，因此分裂节点造成的影响也是递归的，若一次插入导致了多次分裂则可以通过递归层层向上进行。

if child.is\_over\_flow(): # 若已满，需分裂  
 split\_node = child.split()  
 self.insert\_child(split\_node.get\_first\_leaf\_key(), split\_node) # 分裂出的结点作为child插入到非叶结点

分裂函数将当前节点的key和children分出一半放到一个新的非叶结点中，然后返回该新结点。

# 分裂当前节点 返回分裂出的新结点  
def split(self):  
 new\_node = BPlusTree.InternalNode(self.tree)  
 \_from = len(self.keys) // 2 + 1 # 从中间分裂 //为向下取整  
 \_to = len(self.keys)  
 new\_node.keys.extend(self.keys[\_from:\_to])  
 new\_node.children.extend(self.children[\_from:\_to + 1])  
  
 self.keys = self.keys[:\_from-1]  
 self.children = self.children[:\_from]  
  
 return new\_node

**删除操作(delete)**功能为删除key对应的value，该方法同样先找到key对应的子结点，然后递归地调用子结点的delete方法，叶结点的delete方法同样在LeafNode类部分讨论。

def delete(self, key):  
 child = self.get\_child(key)  
 child.delete(key)

删除一个值后，子结点（叶结点）可能出现key不足B+树阶数的一半的情况，这个时候需要合并结点，首先获得该结点的左右兄弟，若存在左兄弟，则将左兄弟与当前结点合并（优先向左合并），并删除被合并的结点，若左兄弟合并后溢出了，则还需要再分裂；若不存在左兄弟，则向右合并、删除被合并的结点、判断合并后是否溢出。

判断合并的代码如下：

if child.is\_under\_flow(): # 指针少于规定数目，需合并  
 left\_bro = self.get\_child\_left\_bro(key)  
 right\_bro = self.get\_child\_right\_bro(key)  
 if left\_bro: # 优先向左合并  
 left\_bro.merge(child)  
 self.delete\_child(key) # 将被合并的结点删除  
 if left\_bro.is\_over\_flow(): # 若溢出则还需分裂  
 split\_node = left\_bro.split()  
 self.insert\_child(split\_node.get\_first\_leaf\_key(), split\_node)  
 else: # 没有左兄弟，向右合并  
 child.merge(right\_bro)  
 self.delete\_child(key)  
 if child.is\_over\_flow():  
 split\_node = child.split()  
 self.insert\_child(split\_node.get\_first\_leaf\_key(), split\_node)

合并函数将参数结点的key和children加入到当前节点的key和children中，并且为了平衡children数和key数，需要获得被合并结点的最左叶结点的最小key值，加入到当前结点的keys中（在合并之初）。合并结点的代码为：

# 合并当前节点和bro\_node  
def merge(self, bro\_node):  
 self.keys.append(bro\_node.get\_first\_leaf\_key())  
 self.keys.extend(bro\_node.keys)  
 self.children.extend(bro\_node.children)

**2.2 LeafNode（叶节点）类**

叶结点类除了Node类定义的属性外还有用来保存数据的values属性和指向下一个叶结点的next属性。

叶结点的get\_value(key)方法判断参数key是否在自己的keys中出现，若出现，则返回对应的value，否则返回None

def get\_value(self, key):  
 index = self.search\_index(key) - 1  
 if self.keys[index] == key:  
 return self.values[index]  
 else:  
 return None

这里的search\_index找到的是应该插入key的位置，因此减1才是正好保存key的位置。search\_index的代码如下：

def search\_index(self, key):  
 index = 0  
 for k in self.keys:  
 if k > key:  
 break  
 index += 1  
 return index

叶结点的插入方法insert(key, value)先找到应该插入的列表下标，然后判断该下标是否在self.keys的范围内并且keys中已包含key，若是，则直接在对应位置插入value；否则在对应位置分别插入key和value。与非叶结点不同的是，叶结点不需要判断子结点是否需要分裂，只需在根节点key溢出时进行判断并分裂，设置新的根即可（B+树第一次分裂）。叶结点的插入方法代码如下：

def insert(self, key, value):  
 index = self.search\_index(key)  
 if index < len(self.keys) and self.keys[index] == key:  
 self.values.insert(index, value)  
 else:  
 self.keys.insert(index, key)  
 self.values.insert(index, value)  
 if self.tree.root.is\_over\_flow(): # 第一次分裂  
 split\_node = self.split()  
 new\_root = BPlusTree.InternalNode(self.tree)  
 new\_root.keys.append(split\_node.get\_first\_leaf\_key())  
 new\_root.children.append(self)  
 new\_root.children.append(split\_node)  
 self.tree.root = new\_root

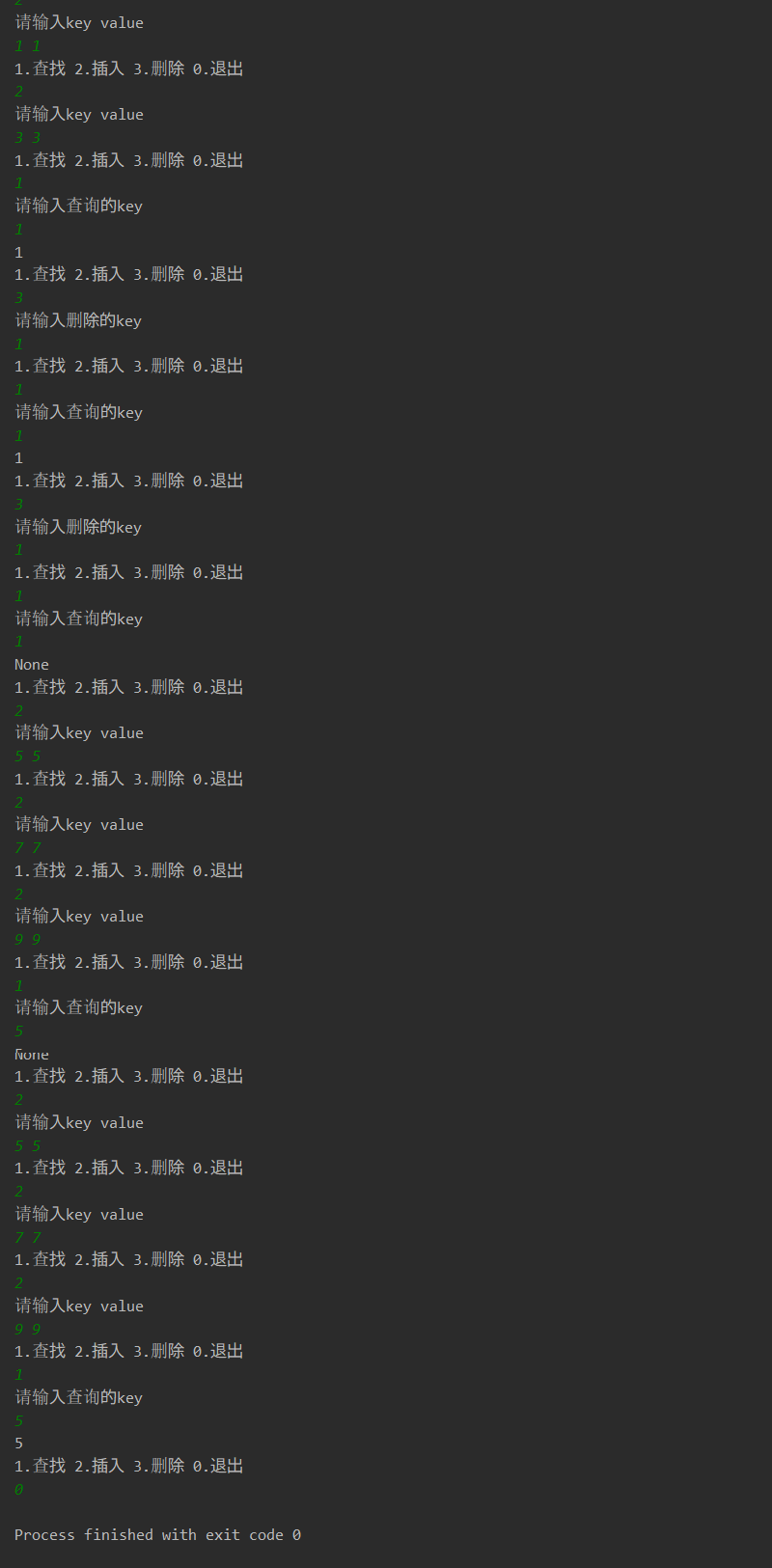
叶结点的删除方法直接在自己的keys中找到要删除的key，并删除它对应的value，若key不在自己的keys中则不执行操作。

def delete(self, key):  
 index = self.search\_index(key) - 1  
 if self.keys[index] == key:  
 del self.keys[index]  
 del self.values[index]

叶结点的分裂和合并方法需要设置self.next即指向下一叶结点的指针，其他与非叶结点大致相同。

**2.3 测试功能**

定义了B+树类后，测试它的查找、插入、删除功能：

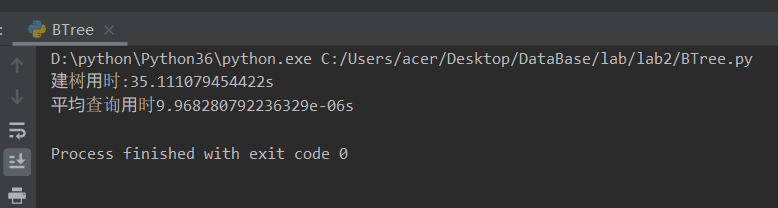


**2.4 从文件建立B+树索引**

从第一步创建的文件建立B+树索引。首先初始化B+树，然后每次从文件中读取16字节数据，前4字节转换为整型作为key，后12字节转换为字符串型value，插入到B+树中，直到读完文件插入结束。然后用文件的前100个记录进行查询操作。分别记录建树和查询操作的用时。建树的代码如下：

def build\_tree\_from\_file():  
 a = time.time()  
 b\_plus\_tree = BPlusTree()  
 test\_search = []  
  
 with open('data', 'rb') as f:  
 while True:  
 chunk = f.read(4)  
 if not chunk:  
 break  
 key = int.from\_bytes(chunk, byteorder='big')  
 value = f.read(12).decode(encoding='utf-8')  
 b\_plus\_tree.insert(key, value)  
  
 b = time.time()  
 print('建树用时:' + str(b - a) + 's')  
  
 with open('data', 'rb') as f:  
 for i in range(100):  
 tmp = f.read(4)  
 test\_search.append(int.from\_bytes(tmp, byteorder='big')) # 待查询列表 100个  
 f.read(12) # skip values  
  
 c = time.time()  
 for s in test\_search:  
 b\_plus\_tree.search(s)  
 d = time.time()  
 avg = (d - c) / 100  
 print('平均查询用时' + str(avg) + 's')

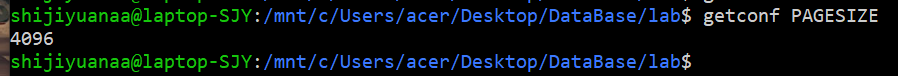
得到运行结果为：



建立有100万个条目B+树索引用时35秒，平均每次查询操作用时9.97\*10^(-6)秒，可以看出B+树索引有很强的时间性能。

**3.多路归并排序**

1MB内存 = 1024\*1024 B，而待排序文件大小=16B \* 1000000，若**分为16个子集合**，则每个子集合大小为1000\*1000B < 1MB。且我的内存页大小为4KB，



也就是说1MB内存共有256个内存页，16<256，因此分为16个子集合**满足两趟扫描算法**的要求。

**归并算法实现**

第一步，先将整个待排序文件划分为16个子文件，划分过程中对每个子文件（每个子文件10^6B<1MB）进行内排序（用快排实现）。划分子文件的代码为：

# 划分为小文件并排序  
def split\_file():  
 with open('data', 'rb') as fr:  
 for i in range(block\_number): # block\_number = 16  
 tmp = []  
 for j in range(int(part\_file\_size/16)): # 每个记录16字节  
 attr\_a = fr.read(4)  
 attr\_b = fr.read(12)  
 tmp.append((attr\_a, attr\_b))  
 quick\_sort(tmp, 0, len(tmp)-1) # 对每个子文件进行排序  
 with open(prefix+str(i), 'wb') as fw:  
 for item in tmp:  
 fw.write(item[0])  
 fw.write(item[1])

其中，快速排序的代码为：

def partition(arr, low, high):  
 i = low-1  
 pivot = arr[high][0]  
 for j in range(low, high):  
 if arr[j][0] <= pivot:  
 i = i+1  
 arr[i], arr[j] = arr[j], arr[i]  
 arr[i+1], arr[high] = arr[high], arr[i+1]  
 return i+1  
  
def quick\_sort(arr, low, high):  
 if low < high:  
 pi = partition(arr, low, high)  
 quick\_sort(arr, low, pi-1)  
 quick\_sort(arr, pi+1, high)

得到了16个已排序的子文件之后，对这些文件进行归并：

首先初始化需要用到的变量，如输入缓冲区、输出缓冲区、文件列表等。输入缓冲区是一个保存字节型数据的列表，长度为16，其中每个元素对应从一个文件中读取的数据。

file\_list = [] # 保存文件指针  
 in\_buffer = [] # 模拟输入缓冲区  
 out\_buffer = b'' # 保存字节型记录，模拟输出缓冲区  
 for i in range(block\_number):  
 file\_list.append(open((prefix + str(i)), 'rb')) # 依次打开待归并文件

然后从每个文件中读出一个内存页大小的数据，保存在输入缓冲区中：

with open('merge\_sort/result', 'wb') as fw:  
 for fr in file\_list:  
 in\_buffer.append(fr.read(page\_size)) # 保存从每个子集合中读的块

之后进入归并排序的主循环，首先获得输入缓冲区中最小的key，若存在该key，则获取该key对应的记录（16个字节）、将该记录添加到输出缓冲区并从输入缓冲区中删除，判断输出缓冲区是否达到一个内存页大小，若已达到，则写入到文件，并将输出缓冲区清空。接着判断输入缓冲区中是否有某个文件读的块已经全部用完（从输入缓冲区中移除），若存在，则再从该文件中读取一个内存页大小的数据到输入缓冲区中。若输入缓冲区中已经没有待归并数据，则结束循环，并关闭文件。

while True: # 进行归并排序  
 index = get\_min(in\_buffer) # 找到最小key值  
 if index == -1: # 说明所有文件都已读完，终止排序  
 fw.write(out\_buffer)  
 break  
 key\_value = in\_buffer[index][:16] # 取最小key对应的记录  
 in\_buffer[index] = in\_buffer[index][16:] # 从输入缓冲区中删除  
 out\_buffer += key\_value # 放到输出缓冲区  
 if len(out\_buffer) >= page\_size: # 若输出缓冲区已有一个页大小，则输出到文件  
 fw.write(out\_buffer)  
 out\_buffer = b'' # 输出缓冲区清空  
 if len(in\_buffer[index]) == 0: # 若一个文件的文件块已排序完  
 page = file\_list[index].read(page\_size)  
 if len(page) > 0: # 若该文件还有剩余，则再读一个pagesize  
 in\_buffer[index] = page  
 else: # 否则说明该文件已读完，不处理  
 pass

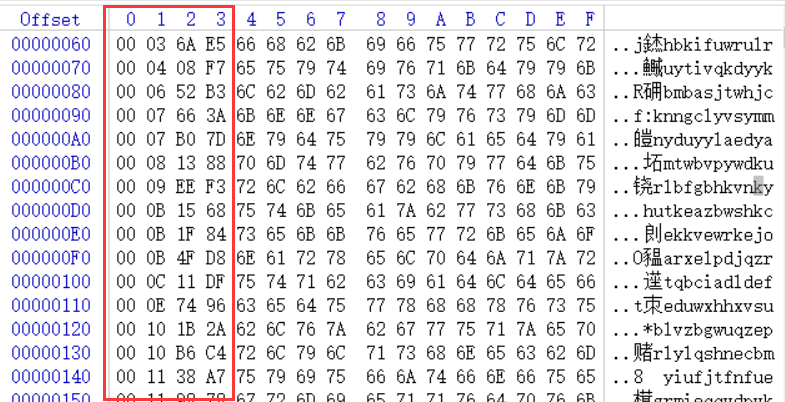
for i in range(block\_number):  
 file\_list[i].close() # 关闭文件

获得输入缓冲区中最小key对应的文件的代码为：

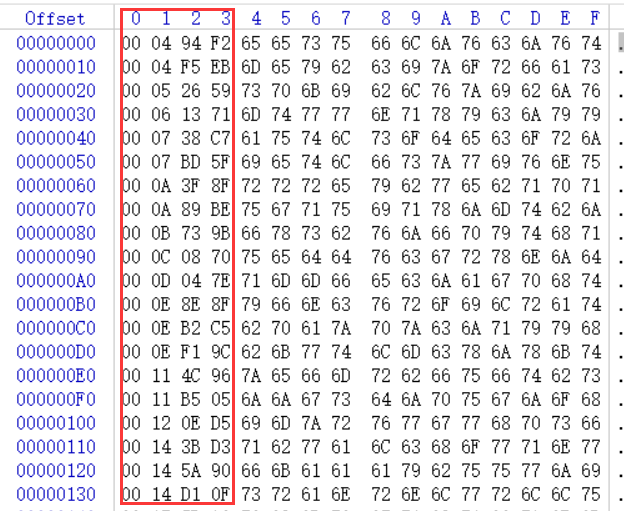
# 找到待排序块中最小的块的下表  
def get\_min(in\_buffer):  
 min\_key = 0xFFFFFFFF.to\_bytes(4, byteorder='big')  
 index = -1  
 for i in range(len(in\_buffer)):  
 if len(in\_buffer[i]) > 0: # 在还没读完的文件块中找最小的key  
 tmp = in\_buffer[i][:4]  
 if tmp <= min\_key:  
 index = i  
 min\_key = tmp  
 return index

执行归并排序，用WinHex查看各阶段输出结果：

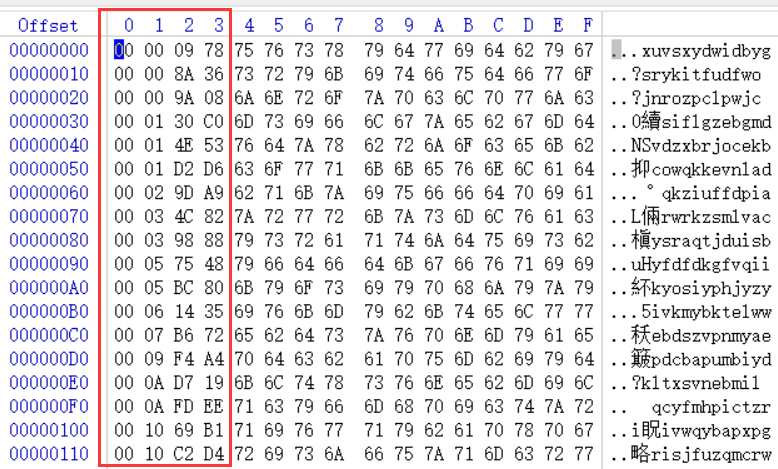
第一个子集合：



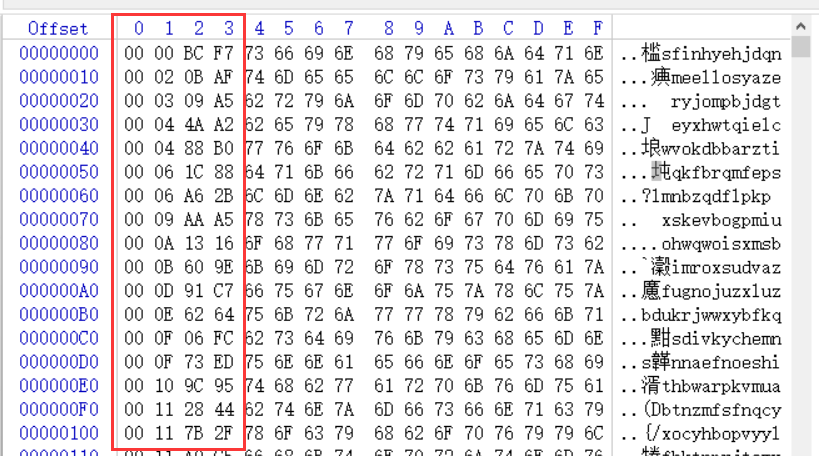
第二个子集合：



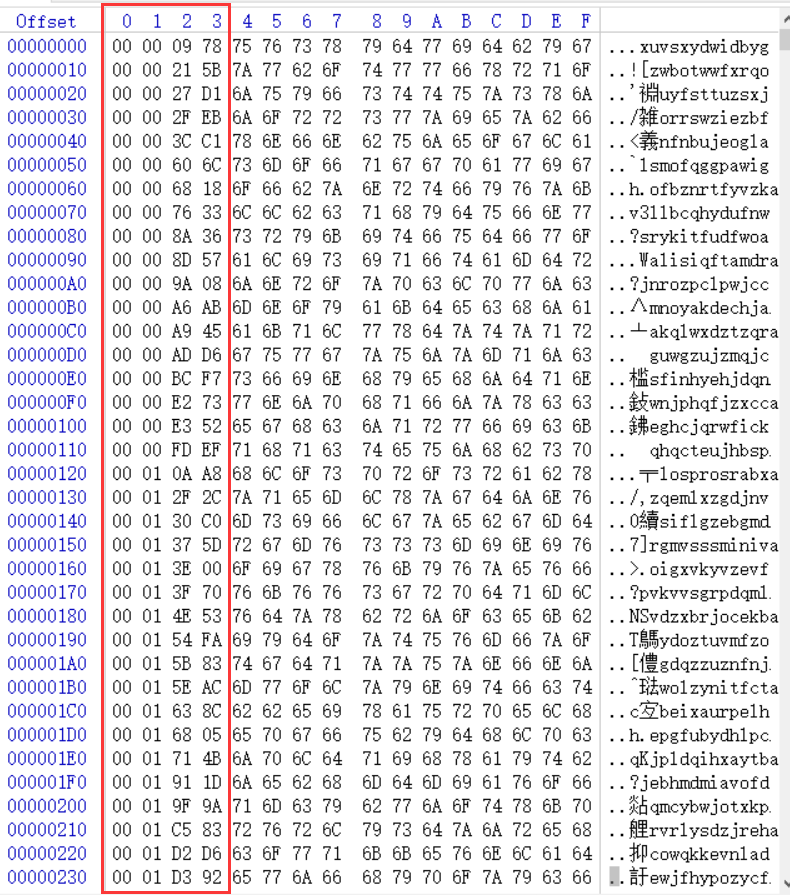
第三个子集合：

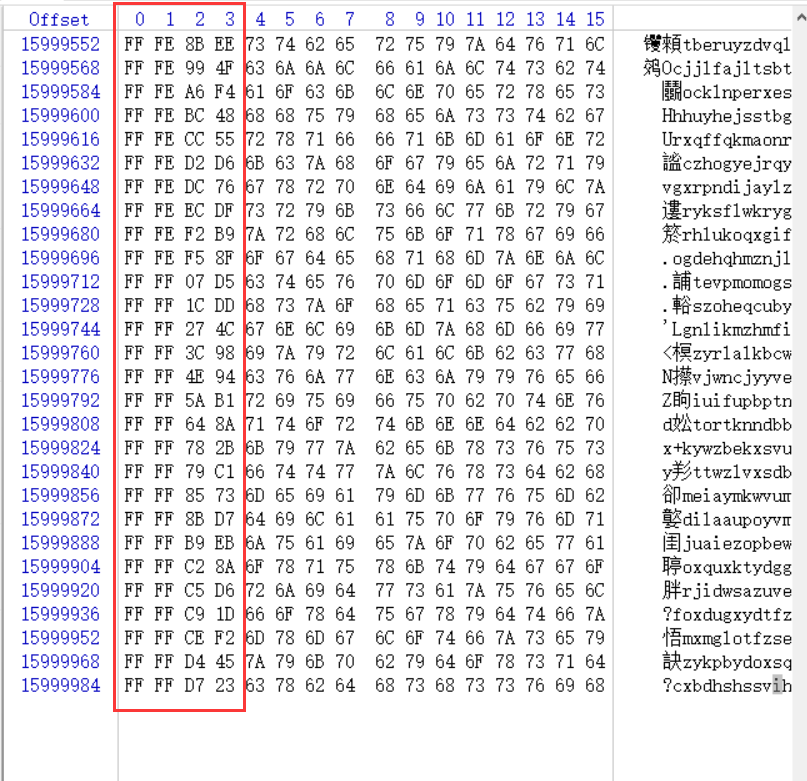


第四个子集合：



可以看到每个子集合内部已经按照属性A排序，其他12个子集合同理，不再重复展示。最后归并得到的结果文件为：

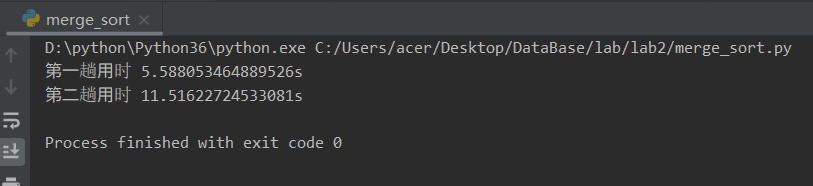




可以看到，所有数据都已按照属性A排序。

**性能分析**

在程序运行时记录消耗的时间：



由于第一趟扫描用的是快速排序，时间复杂度为O(nlogn)，因此用时很短。我一开始尝试使用冒泡排序，但是由于数据量太大（尽管是子集合但仍有10^6B的数据），用时非常长以至于很难运行完程序，在改为快速排序后速度大大提升，但是由于快速排序是递归调用，因此占用的内存空间比冒泡排序大很多。归并排序由于要处理16\*10^6B的数据，因此用时比第一趟扫描长一些。如果希望进一步提升程序运行速度可以每次仅从文件中读取一个记录，这里选择每次读取一个内存页大小是为了模拟多路归并排序的过程。

## 四、实验收获

通过在课堂上学习B+树我仅能够大致说出每个操作应该怎么做，但没有考虑过真正实现，经过本次实验一点一点实现B+树，我对B+树的结构和操作有了更深的印象。另外多路归并排序算法也让我体会到了在处理大文件时要充分利用存储的层次结构。