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:	杨富祥
学号:	1171800323
学院:	计算机学院
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、插入和删除算法
- 掌握多路归并排序算法
- 用高级语言实现。

二、实验环境

Windows10 + Python3.7

三、实验过程及结果

1．数据生成

实验要求生成 100 万条记录，每条记录由属性 A（4 字节整数）和属性 B（12 字节字符串）组成。代码如下：

```
total = 1000000
output = ''
for index in range(0, total):
    A = random.randint(1, total)
    B = ''
    for i in range(0, 12):
        B += chr(random.randint(97, 122))
    output += str(A) + ',' + B + '\n'
```

生成数据保存在 data.csv 中：

```
key,value
450574,teuvqillbkhs
452912,kwmrtnuiugcp
278503,lkzcdwhrjmsy
112154,mwvalykibwui
979139,zedbpqmmbrjs
473089,nxaeikejcaud
374142,siuyplmdvapl
671983,jyimwwrgqqvy
707110,lztttlqumwze
260289,jmokptutmfmf
812401,xndgyiwyuics
796462,kjnejplnjdcl
164383,zytdrfqrpnif
```

2．B+树索引算法

1）数据结构

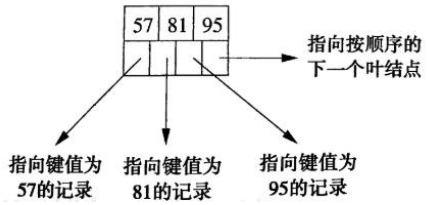
KeyValue：对每个记录 A，B 存储到 KeyValue 中：

```
class KeyValue:
    def __init__(self, key, value):
        self.key = key
        self.value = value

    def __str__(self):
        return str((self.key, self.value))
```

将所有数据读到 keyValueList 中：

```
def read_data(filename='../data.csv'):
    keyValueList = []
    data = pd.read_csv(filename, sep=',')
    for number, keyValue in data.iterrows():
        key = keyValue["key"]
        value = keyValue["value"]
        keyValueList.append(KeyValue(key, value))
    return keyValueList
```

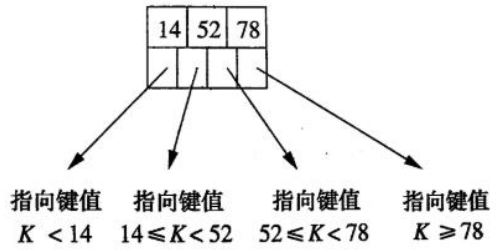
B+树的叶结点，以下图为例，order 为 4，keyValueList 存放{57:“x”，81：“x”，95：“x”}，brother 为下一个叶结点，parent 为一个内结点。




```
class LeafNode:
    def __init__(self, order):
        self.__order = order
        self.keyValueList = []
        self.brother = None
        self.parent = None

    @staticmethod
    def isLeaf():
        return True

    def isFull(self):
        return len(self.keyValueList) > self.__order - 1

    def isLessThanHalf(self):
        return len(self.keyValueList) < (self.__order - 1) / 2
```

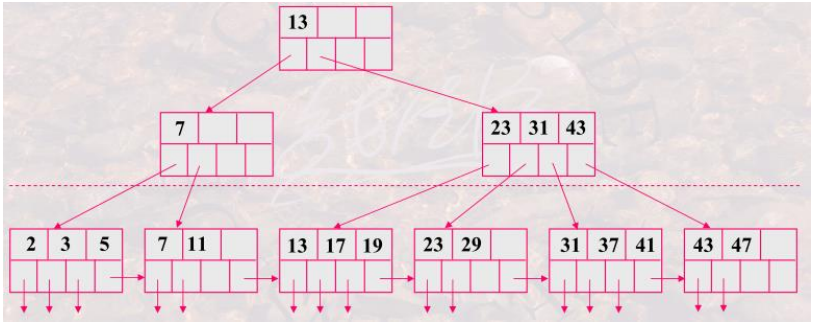
B+树的内结点，以下图为例，order 为 4，indexValueList 为[14,52,78]，

pointerList 存放内结点或者叶结点，parent 为内结点或 None。




```
class InterNode:
    def __init__(self, order):
        self.__order = order
        self.indexValueList = []  # 索引值
        self.pointerList = []  # 指向某一个结点的指针
        self.parent = None

    @staticmethod
    def isLeaf():
        return False

    def isFull(self):
        return len(self.indexValueList) > self.__order - 1

    def isLessThanHalf(self):
        return len(self.pointerList) < self.__order / 2
```

B+树，以下图为例，order 为 4，root 初始为叶结点，之后会被修改为内结点（13 所在结点），从根节点可抵达任意子结点。leaf 为 2，3，4 所在结点，可以根据 brother 遍历整个索引。




```
class BplusTree:
    def __init__(self, order):
        self.__order = order
        self.__root = LeafNode(order)
        self.__leaf = self.__root
```

2）插入

核心代码逻辑：

```
def insert_node(n, canInsert=True):
    if n.isLeaf():
        # 如果是叶结点，找到合适的位置完成插入
        if canInsert:
            sortedList = [x.key for x in n.keyValueList]
            index = binary_search_right(sortedList, keyValue.key)
            n.keyValueList.insert(index, keyValue)
        # 如果叶结点满了，分裂叶结点，并再次从父结点开始插入（假插入，为的是解决父节点可能会满的问题）
        if n.isFull():
            insert_node(split_leaf(n), False)
        else:
            return
    else:
        # 如果是内结点，满了，就分裂，从其父节点开始插入
        # 如果父节点也满了，分裂并从父节点的父节点开始插入，以此类推，确保所有结点数目合法
        if n.isFull():
            insert_node(split_inter(n), canInsert)
        else:
            if canInsert:
                # 内结点不满，寻找适合插入的子女结点进行插入
                index = binary_search_right(n.indexValueList, keyValue.key)
                insert_node(n.pointerList[index], canInsert)
```

递归插入，从根结点递归嵌套进入叶结点完成插入，插入之后再做处理（逐步向上）使结点数目合法：

- 如果插入的叶结点不满，没有分裂，直接返回；
- 如果叶结点满了，分裂，将新节点插入父节点中；
- 如果父节点满了，分裂，沿着树向上递归处理；
- 直到不再分裂或创建一个新的根结点为止。

测试用例：

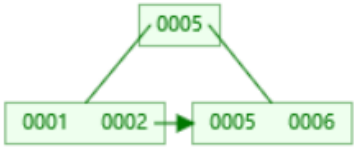
```
key,value
1  ,a
2  ,b
5  ,e
6  ,f
3  ,c
4  ,d
10 ,j
11 ,k
12, l
13, m
7  ,g
8  ,h
9  ,i
```

关键过程（order=4）：

初始根为叶结点，依次插入 1，2，5：

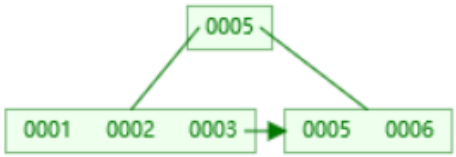

```
insert  (5, 'e')
3
B+ Tree:
[1, 2, 5] leaf height = 0 parent = None
['1a', '2b', '5e']
```

插入 6，引起叶结点分裂，将新结点的最小值 5 插入父节点：




```
insert  (6, 'f')
4
B+ Tree:
[5] inter height =  0 parent = None
[1, 2] leaf height = 1 parent =  [5]
[5, 6] leaf height = 1 parent =  [5]
['1a', '2b', '5e', '6f']
```

插入 3：

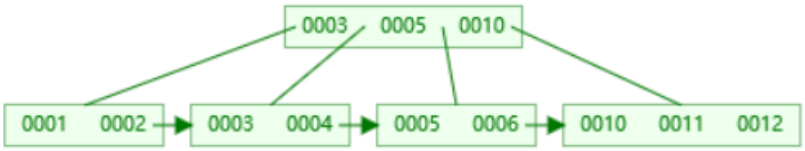



```
insert  (3, 'c')
5
B+ Tree:
[5] inter height =  0 parent = None
[1, 2, 3] leaf height = 1 parent =  [5]
[5, 6] leaf height = 1 parent =  [5]
['1a', '2b', '3c', '5e', '6f']
```

插入 4，叶结点分裂出一个新结点，将 3 插入父节点对应位置：

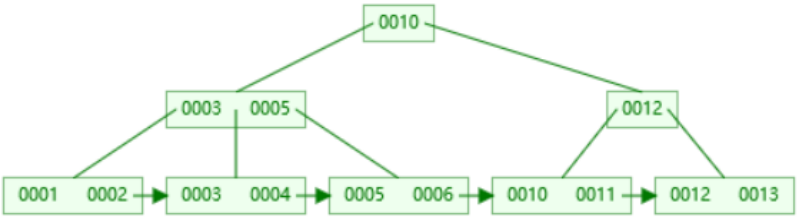
```
insert  (4, 'd')
6
B+ Tree:
[3, 5] inter height =  0 parent = None
[1, 2] leaf height = 1 parent =  [3, 5]
[3, 4] leaf height = 1 parent =  [3, 5]
[5, 6] leaf height = 1 parent =  [3, 5]
['1a', '2b', '3c', '4d', '5e', '6f']
```

插入 10，11，12：



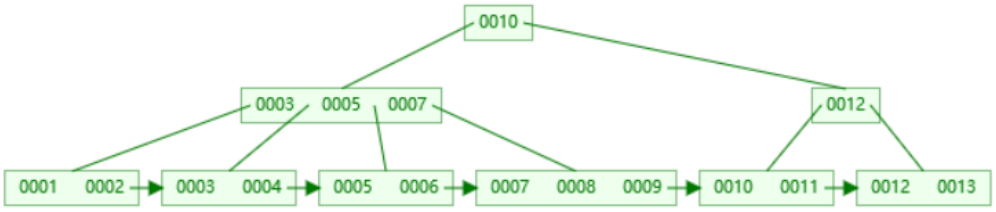

```
insert  (12, ' l')
9
B+ Tree:
[3, 5, 10] inter height =  0 parent = None
[1, 2] leaf height = 1 parent =  [3, 5, 10]
[3, 4] leaf height = 1 parent =  [3, 5, 10]
[5, 6] leaf height = 1 parent =  [3, 5, 10]
[10, 11, 12] leaf height = 1 parent =  [3, 5, 10]
['1a', '2b', '3c', '4d', '5e', '6f', '10j', '11k', '12 l']
```

插入 13，叶结点分裂，引起父节点分裂，父节点分裂产生索引为[10,12]的结点，10 会上升到父节点（根结点）：




```
insert  (13, ' m')
10
B+ Tree:
[10] inter height =  0 parent = None
[3, 5] inter height =  1 parent =  [10]
[12] inter height =  1 parent =  [10]
[1, 2] leaf height = 2 parent =  [3, 5]
[3, 4] leaf height = 2 parent =  [3, 5]
[5, 6] leaf height = 2 parent =  [3, 5]
[10, 11] leaf height = 2 parent =  [12]
[12, 13] leaf height = 2 parent =  [12]
['1a', '2b', '3c', '4d', '5e', '6f', '10j', '11k', '12 l', '13 m']
```

插入 7，8，9：




```
insert  (9, 'i')
13
B+ Tree:
[10] inter height =  0 parent = None
[3, 5, 7] inter height =  1 parent =  [10]
[12] inter height =  1 parent =  [10]
[1, 2] leaf height = 2 parent =  [3, 5, 7]
[3, 4] leaf height = 2 parent =  [3, 5, 7]
[5, 6] leaf height = 2 parent =  [3, 5, 7]
[7, 8, 9] leaf height = 2 parent =  [3, 5, 7]
[10, 11] leaf height = 2 parent =  [12]
[12, 13] leaf height = 2 parent =  [12]
['1a', '2b', '3c', '4d', '5e', '6f', '7g', '8h', '9i', '10j', '11k', '12 l', '13 m']
```

3）查询

引入两个二分查找（其实插入也用到了）：

```
# 二分查找，返回的位置及其之后的元素都是大于element的
def binary_search_right(sortedList, element, low=0, high=None):

# 二分查找，  返回的位置之前的都是小于element的
def binary_search_left(sortedList, element, low=0, high=None):
```

核心代码逻辑：

```
# 寻找键值所在叶结点
def search_key(n, k):
    if n.isLeaf():
        sortedList = [x.key for x in n.keyValueList]
        i = binary_search_left(sortedList, k)
        return i, n
    else:
        i = binary_search_right(n.indexValueList, k)
        return search_key(n.pointerList[i], k)
```

• 从根结点开始，首先从结点内部查找（由于结点内部是升序的，二分查找即可）。

• 比如查找 4，结点内部存放 1,5,8,3 那么查到 5 就可以停了，沿着对应的 5 左边的指针（区间 1<=x<5）继续向下查找。

• 直到最后进入叶节点，返回键值所在叶结点和叶结点中键值的位置。

查询函数为：`def search(self, low=None, high=None)`，可以查询在某一范围的键值。

• 如果 low 为 None，查找所有键值小于等于 high 的记录；

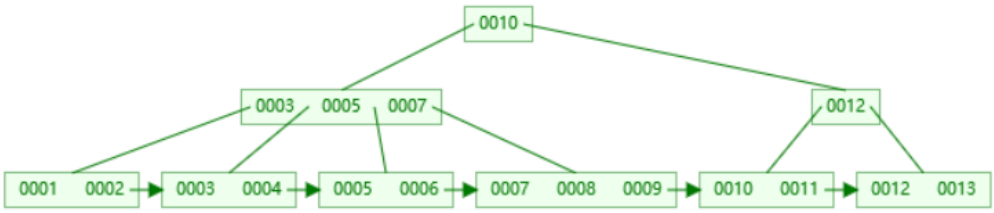
• 如果 high 为 None，查找所有键值大于等于 low 的记录；

• 如果 low 和 high 都不为 None，且 low<=high，查找所有键值大于等于 low 且小于等于 high 的记录。

代码逻辑大致相同，仅展示 low 为 None 的情况，从第一片叶结点出发，沿着 brother 指针往后寻找，直到键值大于 high：

```
if low is None:
    # 寻找所有<=high的键值对，从第一片叶结点出发，
    while True:
        for keyValue in leaf.keyValueList:
            if keyValue.key <= high:
                result.append(keyValue)
            else:
                return result
            if leaf.brother is None:
                return result
            else:
                leaf = leaf.brother
```

结果展示：



bpTree.search(1, 3)：

```
search:
['1a', '2b', '3c']
```

bpTree.search(3, 3)：

```
search:
['3c']
```

bpTree.search(10, 15)：

```
search:
['10j', '11k', '12 l', '13 m']
```

bpTree.search(15, 15)：

```
search:
[]
```

4）删除

核心代码逻辑：

• 从根节点出发抵达索引值所在叶结点及叶结点中索引值位置，拟删除；

• 如果正常删除之后，该叶结点数不少于一半，则可能需要调整父节点对应位置的索引值；

• 如果删除后，该叶结点数少于一半，递归进入父节点调整：要么合并，要么从兄弟结点借，如果父节点也不合法，再次递归向上直到根结点或者不再引起合并。（调整的策略是能合并，合并后结点数目合法就合并，否则从左右结点借）。

```
def delete_node(node, canDelete=True):
    if node.isLeaf():
        if canDelete:
            # 找到叶结点，删除对应的键值对（如果存在）
            sortedList = [x.key for x in node.keyValueList]
            index = binary_search_left(sortedList, key)
            try:
                keyValue = node.keyValueList[index]
            except IndexError:
                return -1
            else:
                if keyValue.key != key:
                    return -1
                else:
                    # 正常删除之后，可能会引起结点合并等动作
                    node.keyValueList.remove(keyValue)
                    # 如果删除后结点数少于一半，需要在父节点进行调整，要么合并，要么从兄弟结点借
                    if node.isLessThanHalf():
                        delete_node(node.parent, False)
                    else:
                        # 如果删除后结点数目仍然合法，父节点的索引值可能需要改变
                        if node.parent is not None:
                            index = binary_search_right(node.parent.indexValueList, key)
                            parent_indexList = node.parent.indexValueList
                            if node.keyValueList[0].key > parent_indexList[index - 1]:
                                parent_indexList[index - 1] = node.keyValueList[0].key
                    return 0
```

```
else:
    index = binary_search_right(node.indexValueList, key)
    # 索引到的结点是最右的，可能需要从左兄弟结点借
    if index == len(node.indexValueList):
        leftChild = node.pointerList[index - 1]
        rightChild = node.pointerList[index]
        # 如果索引到要删除key所在结点子女少于一半，则要合并或者借元素，否则正常删除
        if rightChild.isLessThanHalf():
            if rightChild.isLeaf():
                # 如果可以合并，就合并，否则从左兄弟借
                if len(rightChild.keyValueList) + len(leftChild.keyValueList) \
                        <= self.__order - 1:
                    return delete_node(merge(node, index - 1), canDelete)
                else:
                    transfer_leftToRight(node, index - 1)
            else:
                # 如果可以合并，则合并，否则从左兄弟借
                if len(rightChild.pointerList) + len(leftChild.pointerList) \
                        <= self.__order:
                    return delete_node(merge(node, index - 1), canDelete)
                else:
                    transfer_leftToRight(node, index - 1)
        else:
            if canDelete:
                return delete_node(rightChild, canDelete)
    else:
        # 索引到的结点不是最右，统一从右兄弟结点借
        # 操作与上类似
```

合并结点：

```
def merge(node, index):
    leftChild = node.pointerList[index]
    rightChild = node.pointerList[index + 1]
    if leftChild.isLeaf():
        # 如果合并的是叶结点，将右儿子值复制到左儿子
        leftChild.keyValueList = leftChild.keyValueList \
                                 + rightChild.keyValueList
        leftChild.brother = rightChild.brother
    else:
        # 如果合并的是内结点，将右儿子索引值和node结点index索引值复制到左儿子
        leftChild.indexValueList = \
            leftChild.indexValueList + [node.indexValueList[index]] \
            + rightChild.indexValueList
        # 将右儿子结点复制到左儿子
        leftChild.pointerList = leftChild.pointerList + rightChild.pointerList
        for leftChildChild in leftChild.pointerList:
            leftChildChild.parent = leftChild
    # 在node结点删除右儿子
    node.pointerList.remove(rightChild)
    # 在node结点删除索引值（已经移入左儿子作为合并后的结点 或者 合并叶结点之后要删除该索引值）
    node.indexValueList.remove(node.indexValueList[index])
    if not node.indexValueList and node.parent is None:
        # 如果node结点索引清空了，删掉该结点，重置根结点
        node.pointerList[0].parent = None
        self.__root = node.pointerList[0]
        del node
        return self.__root
    else:
        return node.parent
```

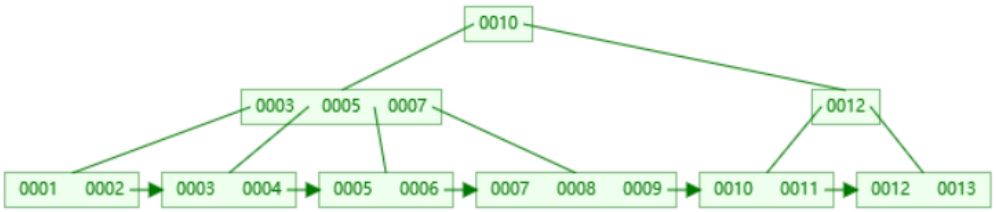
从左兄弟借元素：

```
def transfer_leftToRight(node, index):
    leftChild = node.pointerList[index]
    rightChild = node.pointerList[index + 1]
    if not leftChild.isLeaf():
        # 将index的最后一个结点追加到index+1的第一个结点
        rightChild.pointerList. \
            insert(0, leftChild.pointerList[-1])
        leftChild.pointerList[-1].parent = \
            rightChild
        # 追加index+1的索引值
        rightChild.indexValueList \
            .insert(0, node.indexValueList[index])
        # 更新node的index+1的索引值
        node.indexValueList[index] = \
            leftChild.indexValueList[-1]
        # 删除index的最后一个结点和索引值
        leftChild.pointerList.pop()
        leftChild.indexValueList.pop()
    else:
        # 将index的最后一个结点追加到index+1的第一个结点
        rightChild.keyValueList. \
            insert(0, leftChild.keyValueList[-1])
        # 删除index最后一个结点
        leftChild.keyValueList.pop()
        # 更新node的index索引值
        node.indexValueList[index] = rightChild. \
            keyValueList[0].key
```

从右兄弟借结点：同上。

结果演示：

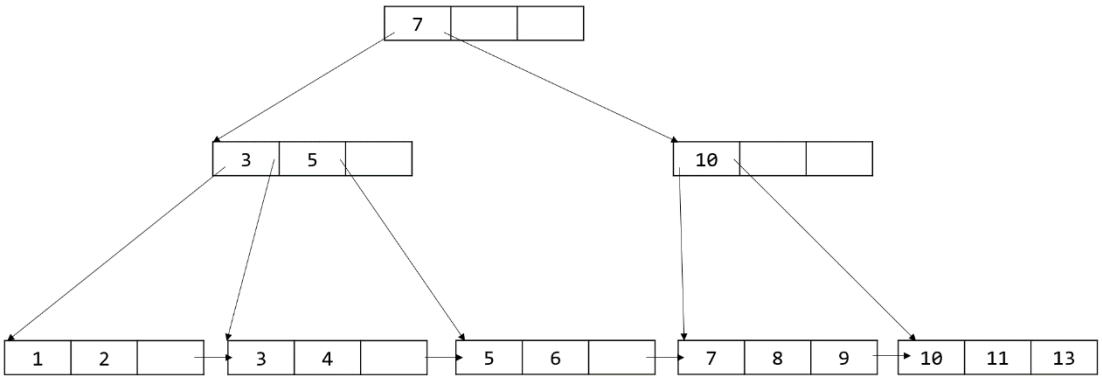
初始：



删除 12，叶结点变为[13]，不合法，进入父节点[12]进行调整，合并叶结点后为[10,11,13]，父节点只剩下一个指针（索引值也被清空了），不合法；

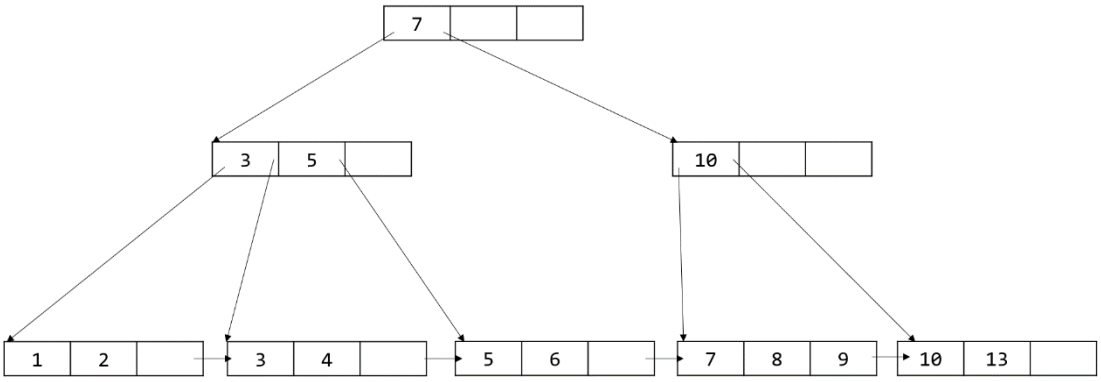
再次进入父节点[10]进行调整，左儿子为[3,5,7]，有 4 个子女，右儿子为[]，它只有一个子女，这样左儿子和右儿子不能合并（合并后会有 5 个子女）；

可以从左儿子借一个叶结点[7,8,9]，左儿子的最末尾一个索引值 7 要升到父结点，父结点的 10 下放到右儿子索引中：




```
0
B+ Tree:
[7] inter height =  0 parent = None
[3, 5] inter height =  1 parent =  [7]
[10] inter height =  1 parent =  [7]
[1, 2] leaf height = 2 parent =  [3, 5]
[3, 4] leaf height = 2 parent =  [3, 5]
[5, 6] leaf height = 2 parent =  [3, 5]
[7, 8, 9] leaf height = 2 parent =  [10]
[10, 11, 13] leaf height = 2 parent =  [10]
```

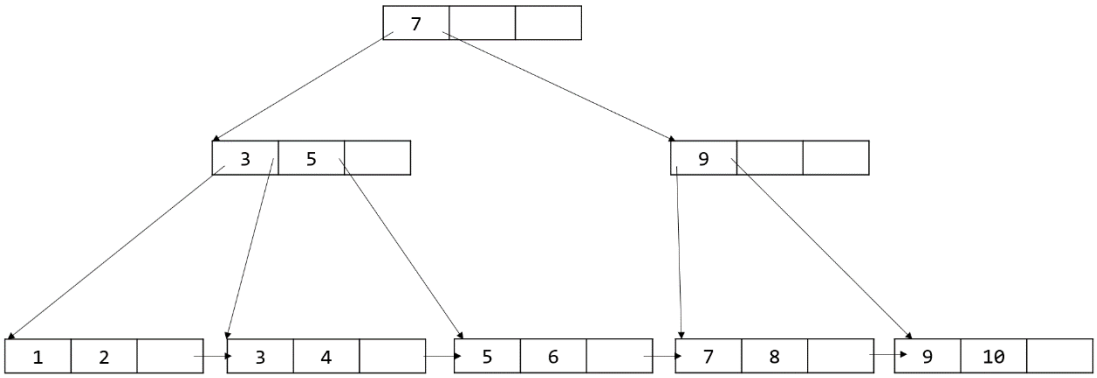
删除 11：




```
0
B+ Tree:
[7] inter height =  0 parent = None
[3, 5] inter height =  1 parent =  [7]
[10] inter height =  1 parent =  [7]
[1, 2] leaf height = 2 parent =  [3, 5]
[3, 4] leaf height = 2 parent =  [3, 5]
[5, 6] leaf height = 2 parent =  [3, 5]
[7, 8, 9] leaf height = 2 parent =  [10]
[10, 13] leaf height = 2 parent =  [10]
```

删除 13，叶结点为[10]，不合法，进入父结点调整，父结点观察到左儿子为

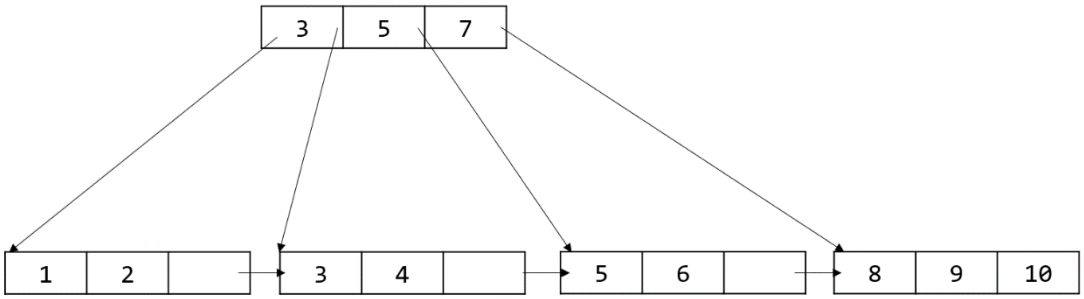
[7,8,9]，右儿子为[10]，不能合并（合并会造成叶结点数目超过合法值），所以可以从左儿子借一个 9 到右儿子，调整该父节点索引值为 9 即可。




```
0
B+ Tree:
[7] inter height =  0 parent = None
[3, 5] inter height =  1 parent =  [7]
[9] inter height =  1 parent =  [7]
[1, 2] leaf height = 2 parent =  [3, 5]
[3, 4] leaf height = 2 parent =  [3, 5]
[5, 6] leaf height = 2 parent =  [3, 5]
[7, 8] leaf height = 2 parent =  [9]
[9, 10] leaf height = 2 parent =  [9]
```

删除 7，叶结点变为[8]，不合法，进入父结点[9]，进行调整，父节点观察到[8]和[9,10]可以合并，于是合并为[8,9,10]，该父节点索引 9 被删除，并且指针只剩下一个（指向合并结点）：

于是又会进入父结点[7]进行调整，它观察到左儿子[3,5]和右儿子[]总子女数为 4，可以合并，这次合并会使得 7 下移到合并结点。合并会使得该父节点[7]索引值清空，只剩下一个子女（即合并结点），而且它是根，不合法，销毁该结点，并将合并结点设置为新的根：



值得注意的是，B+树的非叶结点中关键码值可能在树的叶结点中并不存在，如上图中，7 已经从叶结点删除，但是它仍然存在于非叶结点。

```
0
B+ Tree:
[3, 5, 7] inter height =  0 parent = None
[1, 2] leaf height = 1 parent =  [3, 5, 7]
[3, 4] leaf height = 1 parent =  [3, 5, 7]
[5, 6] leaf height = 1 parent =  [3, 5, 7]
[8, 9, 10] leaf height = 1 parent =  [3, 5, 7]
```

5）复杂度分析

假设 B+树结点指针数最多为 n，索引文件中记录条数为 N，B+树结点至少是半满的，所以可以求得 B+树的高度为$\log_{\lceil \frac{n}{2} \rceil} N$.

一次插入、一次查询、一次删除操作的复杂度都正比于树高，为$O(\log_{\lceil \frac{n}{2} \rceil} N)$.

N 次插入复杂度为$O(N\log_{\lceil \frac{n}{2} \rceil} N)$.

如果以随机顺序插入，结点在平均情况下会比$\frac{2}{3}$更满，树高为$\log_{\lceil \frac{2}{3}n \rceil} N$.

6）100 万条记录实测

实测当 n=10 时，树高为 7（树根为第 0 层，最底层叶结点为第 6 层），与理论值（半满为 8.58 或$\frac{2}{3}$满为 7.2）相近。这样一次插入、查询或删除的效率会比较高，从树根向下查询 6 次即可抵达树叶执行操作，并且由于每个结点容量很大，一次更新操作的影响有限，一般不会引起大量的合并、从左右借结点的动作，从而确保 B+树在面临大规模数据时的高效性。

以下为最底层最右侧树叶：

```
[999912, 999912, 999913, 999918, 999919, 999919, 999921] leaf height = 6 parent =  [999921, 999926, 999934, 999943, 999947]
[999921, 999921, 999922, 999923, 999923, 999924, 999924] leaf height = 6 parent =  [999921, 999926, 999934, 999943, 999947]
[999926, 999926, 999926, 999928, 999930, 999930, 999933] leaf height = 6 parent =  [999921, 999926, 999934, 999943, 999947]
[999934, 999935, 999936, 999938, 999939, 999940, 999941, 999942, 999943] leaf height = 6 parent =  [999921, 999926, 999934, 999943, 999947]
[999943, 999943, 999944, 999944, 999945, 999945, 999947] leaf height = 6 parent =  [999921, 999926, 999934, 999943, 999947]
[999947, 999949, 999949, 999949, 999951, 999952, 999953, 999955] leaf height = 6 parent =  [999921, 999926, 999934, 999943, 999947]
[999957, 999958, 999958, 999959, 999963, 999965] leaf height = 6 parent =  [999966, 999976, 999979, 999991]
[999966, 999966, 999968, 999970, 999971, 999972, 999973] leaf height = 6 parent =  [999966, 999976, 999979, 999991]
[999976, 999976, 999976, 999976, 999977, 999978, 999978] leaf height = 6 parent =  [999966, 999976, 999979, 999991]
[999979, 999979, 999979, 999980, 999981, 999982, 999984, 999986, 999986] leaf height = 6 parent =  [999966, 999976, 999979, 999991]
[999991, 999991, 999993, 999994, 999996, 999997, 999999, 1000000, 1000000] leaf height = 6 parent =  [999966, 999976, 999979, 999991]
Yes
```

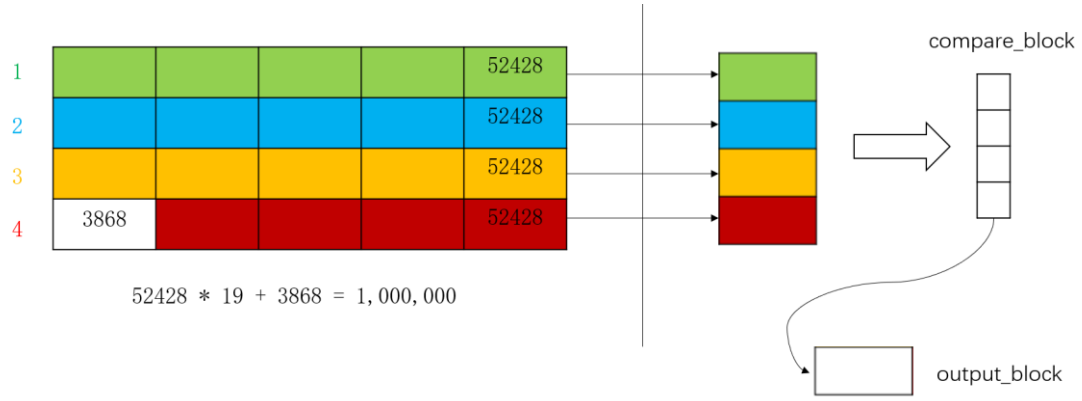
7）本次实验算法的局限性

未考虑重复键值的情况，这样删除时只会删除一个，叶结点仍有重复值。

3．多路归并排序算法

由于内存只有 1MB，属性 A 为 4 字节整数，所以一次最多只能读入 1024*256 个记录，100 万条记录需要读 4 次。这样可以把 100 万条记录划分为 4 个子集合，考虑到之后可以进行 4 路归并，每个子集合加载一块到内存，还得有一块作为输出块，内存至少容纳 5 块。故将每个子集合划分为 5 块，每块有 52428 个数。

综上，100 万条记录划分为 4 个子集合。每个子集合有 262140 个元素。每个子集合划分成 5 块，每块 52428 个元素，最后一个子集合有一块只有 3868 个元素。每个子集合加载一块到内存，加上输出块，以及比较块（仅 4 个元素）恰好装满内存。



$52428 * 19 + 3868 = 1,000,000$

$52428 * 5 = 262,140 < 1024 * 256 = 262144$

多路归并排序可分成两趟：第一趟划分子集并子集排序，第二趟各子集间归并排序。

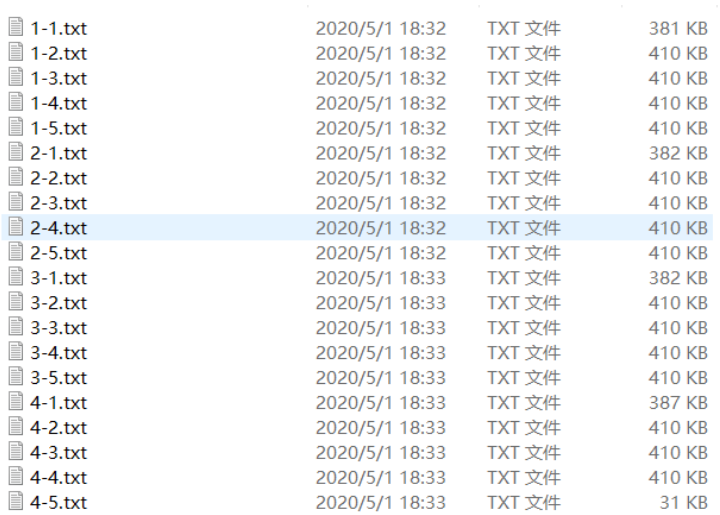
读取原文件，每次读满一个子集合（或者未读满但读到文件末尾），对子集合进行处理。

```
# 划分子集合
def split(filename='../data.csv', run=False):
    if run:
        child_sets = []
        num = 1
        data = pd.read_csv(filename, sep=',')
        for number, keyValue in data.iterrows():
            key = int(keyValue["key"])
            child_sets.append(key)
            if len(child_sets) == child_sets_size \
                    or number == record_size - 1:
                handle_child_sets(child_sets, num)
                num += 1
                child_sets.clear()
```

对于每个子集合先用 python 内置 sorted()函数内排序（升序），然后分块写回磁盘：

```
# 每个子集合先内排序，再划分为多块，写回磁盘
def handle_child_sets(child_sets, sets_num):
    block = []
    block_num = 1
    size = 0
    # 先排序
    for key in sorted(child_sets):
        size += 1
        block.append(key)
        if len(block) == block_size or \
                size == len(child_sets):
            # 将每块写回磁盘
            file_name = 'temp/' + str(sets_num) + '-' + str(block_num) + '.txt'
            with open(file_name, 'w', encoding='utf-8') as f:
                for k in block:
                    f.write(str(k) + '\n')
            block.clear()
            block_num += 1
```

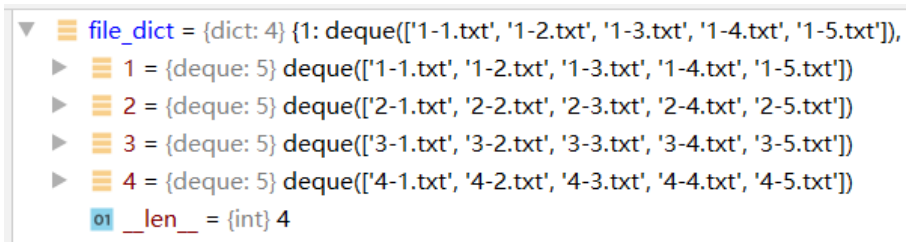
命名方式：第 i 个子集合的第 j 块命名为“i-j.txt”：



1-1.txt	2020/5/1 18:32	TXT 文件	381 KB
1-2.txt	2020/5/1 18:32	TXT 文件	410 KB
1-3.txt	2020/5/1 18:32	TXT 文件	410 KB
1-4.txt	2020/5/1 18:32	TXT 文件	410 KB
1-5.txt	2020/5/1 18:32	TXT 文件	410 KB
2-1.txt	2020/5/1 18:32	TXT 文件	382 KB
2-2.txt	2020/5/1 18:32	TXT 文件	410 KB
2-3.txt	2020/5/1 18:32	TXT 文件	410 KB
2-4.txt	2020/5/1 18:32	TXT 文件	410 KB
2-5.txt	2020/5/1 18:32	TXT 文件	410 KB
3-1.txt	2020/5/1 18:33	TXT 文件	382 KB
3-2.txt	2020/5/1 18:33	TXT 文件	410 KB
3-3.txt	2020/5/1 18:33	TXT 文件	410 KB
3-4.txt	2020/5/1 18:33	TXT 文件	410 KB
3-5.txt	2020/5/1 18:33	TXT 文件	410 KB
4-1.txt	2020/5/1 18:33	TXT 文件	387 KB
4-2.txt	2020/5/1 18:33	TXT 文件	410 KB
4-3.txt	2020/5/1 18:33	TXT 文件	410 KB
4-4.txt	2020/5/1 18:33	TXT 文件	410 KB
4-5.txt	2020/5/1 18:33	TXT 文件	31 KB

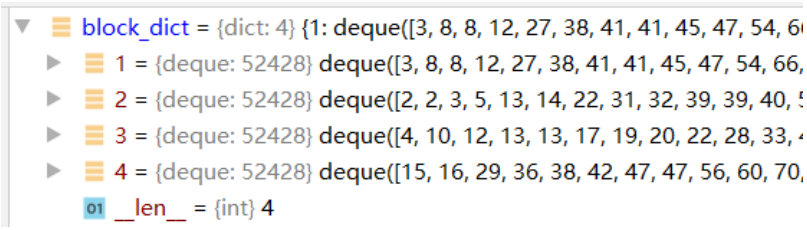
然后进入 4 路归并排序阶段。

对每个子集合建立队列的数据结构，消耗完一块则加载下一块，直到队列为空。故使用字典的数据结构，先对每个子集合建立文件名队列：



对某个子集合，每次从队列左部取出一个文件名，根据此文件名读取一块。

这一块仍然使用队列存储，对每个子集合读取的一块放到 block_dict 中。每次从队列左部取出元素，直到队列空，从 file_dict 中对应队列取出文件名，加载一块到 block_dict。



具体实现来说，首先初始化，将每个子集合的第一块加载到字典 block_dict 中：

```
file_dict = get_temp_file()
block_dict = dict()
for num in file_dict.keys():
    file_queue = file_dict.get(num)
    block_dict.setdefault(num)
    block_dict[num] = read_block('temp/' + file_queue.popleft())
```

然后从每块取出第一个元素（最小的）放到 compare_block 中相应位置。

```
# 输出块
output_block = []
# 比较块
compare_block = []
# 从加载到内存的块中分别取第一个元素（最小的）到比较块中
for num in block_dict.keys():
    key_queue = block_dict.get(num)
    compare_block.append(key_queue.popleft())
```

找到比较块中最小的元素，写到输出块：

```
key_min = sorted(compare_block)[0]
if key_min == sys.maxsize:
    write_block(output_block)
    return
output_block.append(key_min)
```

之后找到 compare_block 中最小元素的位置，进而找到其对应的子集合，从而找到该子集合的块 key_queue，从中取出最左元素放到比较块中。

```
location = compare_block.index(key_min) + 1
key_queue = block_dict.get(location)
if key_queue is None:
    # 如果是None，意味着此子集合已经遍历完了，就将比较块中对应位置设为最大整数
    compare_block[location - 1] = sys.maxsize
else:
    compare_block[location - 1] = key_queue.popleft()
```

如果某子集合的块元素被消耗完了，就加载下一块，如果没有下一块，那就设为 None。

```
if not key_queue:
    file_queue = file_dict.get(location)
    if len(file_queue) != 0:
        block_dict[location] = read_block('temp/' + file_queue.popleft())
    else:
        block_dict[location] = None
```

为验证代码正确性，使用 python 内置 sorted 函数直接对 1000000 条数据排序，作为标准结果进行对比，写入 standard.txt 中。

```
def standard_sort(run=False, filename='../data.csv'):
    if run:
        child_sets = []
        data = pd.read_csv(filename, sep=',')
        for number, keyValue in data.iterrows():
            key = int(keyValue["key"])
            child_sets.append(key)
        write_block(sorted(child_sets), 'standard.txt')
```

经比较发现，自己写的外部归并排序 result.txt 和标准文件 standard.txt 完全相同，可以说明正确性。

```
def compare():
    my = read_block('result.txt')
    standard = read_block('standard.txt')
    # 如果两个队列数目不一样，肯定不正确
    if my.__len__() != standard.__len__():
        print('Wrong!!!')
        return
    while True:
        x = my.popleft()
        y = standard.popleft()
        # 如果同一位置数不相等，肯定不正确
        if x != y:
            print('Wrong!!!')
            break
        # 上面已经确保两队列长度相等，故只需判断当一个队列空就说明正确
        if not my:
            print('Right!')
            break
```

算法效率分析：

外排序时间主要花费在读写磁盘块的次数。本次实验中对 100 万条数据有 4 次 I/O：

• 子集合排序阶段读取这 100 万条记录，分成了 20 个小文件写回磁盘。

• 每个子集合内排序使用的是 python 的 sorted 函数，底层实现是内部归并排序，效率很高。

• 归并阶段分别读取 20 个小文件，将最终结果写回到 result.txt 中。