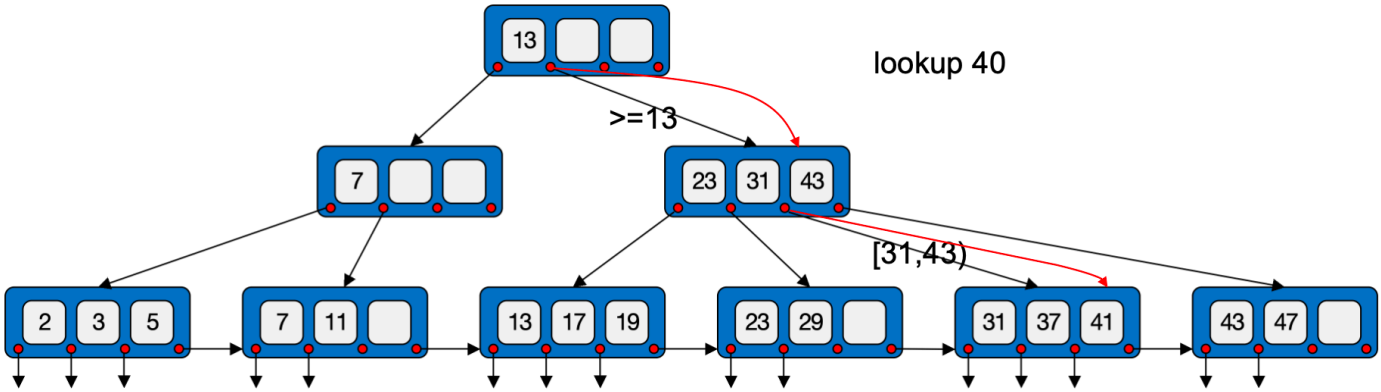
数据库系统原理与实现

姓名：洪宇
学号：2022080901022

实验二

设计

project 2 与 1 最重要的区别在于：在1中，数据块间只是一个简单的单链表，且 CRUD 只针对单个数据块；而在2中，块被组织为B+树，且分为索引块和数据块两类。一颗典型的B+树如下图：



B+树主要有以下几点性质：

1. 设内节点最大分支数为 M，则内节点键数范围为 $[\lfloor \frac{M}{2} \rfloor, M-1]$（注意：该范围并没有完全统一的说法）
2. 一个键的右指针所指向的子节点所持有的键都大于等于这个键
3. 实际记录只存在叶节点中，内节点存键和其指向的块的 id
4. 叶节点串成一条单链表

具体设计为：

- 在超块中增加一个 `root` 字段，记录B+树根节点的块 id。
- 对于索引块：
  - 索引块即内节点。
  - 增加一个宏 `BLOCK_TYPE_INDEX`，且索引块的 `type` 字段均置为该值。该字段用于判断某一个块是索引还是数据。
  - 索引块中的记录只有两个字段，其主键为某一记录 R 的主键，而另一个字段则为要查找到 R 所需要搜索的子节点的块 id。主键对应图中每个白色方框中的数字，块 id 则对应数字右侧的指针。

    - 节点最左侧的指针由该块的 `next` 字段给出。
- 对于数据块：
    - 数据块即叶节点。
    - 与 project 1 中数据块的不同之处在于：其 `next` 字段指向叶节点串成的单链表中的后继节点。

实现

首先说明下文中“键”的含义：内节点的记录可以看作是一个键值对，因此“键”即指代某条记录的主键值。

搜索

算法流程

设查找的键为 k，节点的键值数组为 key，算法流程为：

1. 从根节点出发，在 `slots` 数组中二分查找到第一个满足 $key[i] \leqslant k$ 的下标 i；
2. 接着搜索 $key[i]$ 对应的子节点；
3. 重复1-2，直到到达叶节点；
4. 在叶节点中查找 k，若无对应键则表示记录不存在，否则返回该记录。

实现

搜索函数的签名为：

```
int search(void *keybuf, unsigned int len, std::vector<struct iovec> &iov);
```

其中，`keybuf` 指向要查找的主键值，`len` 为该主键的字节数；当查找成功时，`iov` 用于存储查到的记录。这个签名看起来有些奇怪，因为 `keybuf` 和 `len` 似乎有些多余——之所以需要这两个形参，是因为想要复用 project 1 中可以搜索单个块的 `insertRecord()` 函数，而后者需要这两个参数。

初始化

project 2 函数的初始化与 project 1 大体一致，最大的区别在于需要先向 `kBuffer` 借超块（id 为0），再通过超块的 `root` 字段来获取根节点。其余诸如获取表的元数据、根据主键数据类型动态开辟内存等都在逻辑上与 project 1 类似。

搜索

project 2 中所有 CRUD 函数的搜索和回溯均用栈和 `while` 循环来实现。在 `insert` 中，该栈声明为 `std::stack<unsigned int>` 类型，存储的是路径上的块 id。

在进入 `while` 循环前先将根节点压栈，随后在循环中：

1. 出栈，获取当前块 id；
2. 对该块调用 `searchRecord`（此处省略了向 `kBuffer` 借块，以后都不再赘述该过程），获得在 `slots` 中的下标 `ret`；
3. 若该块为叶节点，则：
   i. 若 `ret` 大于等于 `slot` 总数，表明记录不存在；
   ii. 否则，将 `ret` 对应的记录赋值给 `iov`（通过 `Record` 来获得块中记录的方式已在前文给出；在源码中，我将其封装为了 `getRecord` 函数）；
   iii. 注意：因为 `ret` 对应的是 lower bound，所以该记录**不一定是要查询的记录**。因此，需要通过 `memcmp` 来对比要查询的主键（`keybuf` 指向的值）和获得的记录（`iov[keyIdx]` 指向的值）是否相等；
   iv. 搜索结束，根据 `memcmp` 的结果返回成功或失败。
4. 若该块为内节点，则：
   i. 若 `ret` 大于等于 `slot` 总数，则获取该索引块中的最后一条记录，并将该记录中对应块 id 的值压栈；
   ii. 否则，将 `ret` 处的记录赋给一个临时 `iovec` 数组 `tmp`，并用 `memcmp` 比较 `tmp[keyIdx]` 和 `keybuf` 指向的值。此时要回收前文的伏笔了：
      - 若二者相等，那么直接搜索 `ret` 处记录对应的子节点，即将 `slots[ret]` 对应的块 id 压栈；
      - 若二者不等，**有两种情况**：
      a. `ret != 0`：假设 `slots[2]` 对应的键值为3，`slots[3]` 对应的键值为5，而查询的键值为4，则 `ret` 的值为3，因为 lower bound 返回的是大于等于该值的第一个元素所在的下标，所以接下来应该将 `slots[ret - 1]` 对应的块 id 压栈；
      b. `ret == 0`：假设 `slots[0]` 对应的键值为0，而查询的键值为-1，则 `ret` 的值为0（对应从左往右第二个指针），但实际上应该搜索最左的指针，即将 `next` 字段存储的子节点块 id 压栈。

一方面，`search` 的代码和上文实现思路的对应关系还是很直接的；另一方面，源码中充满了字节序转换、缓冲区借块和释放等细节，因此贴在报告中会显得较为冗余，在此不再给出。代码细节可以直接参看源码及注释。

插入

算法流程

1. 与搜索的流程类似，先定位到应插入到的叶节点；
2. 将要插入的记录插入到该叶节点中；
3. 判断叶节点是否溢出（超过B+树定义中键的范围），若是则将其分裂：
   i. 创建一个新的叶节点，将记录中主键较大的一半转移到新节点中；
   ii. 将新节点的最小键及新节点的块 id 构成的记录插入到父节点的正确位置。

4. 在向上回溯中，重复第三步。

实现

实现与算法流程最大的区别在于：不通过记录数来判断是否溢出，而是在 `insertRecord` 返回 `false`，即该块已无足够的空闲空间时进行分裂。首先，这样的实现更加简单；其次，因为可能有变长字段，而且每个记录的大小在实际场景中并不固定，所以无法用一个固定的阈值来判断块是否需要分裂。

`insert` 函数，包括后文中介绍的 `remove` 和 `update`，都只需接受一个 `iovec` 数组引用 `iov` 作为参数，并返回一个表示是否成功的状态值。

`insert` 中最难处理的是产生了分裂的情况，因为在父节点中插入新记录之后，父节点也有可能需要分裂，因此该过程在回溯的过程中是可能多次发生的。我的解决方案是：

- 栈仍然只存放块 id。
- 将整个过程在逻辑上分为三部分：
  i. 向下搜索到叶节点
  ii. 叶节点进行插入，若有溢出则分裂新块，并将新块的最小键插入到父节点中
  iii. 向上回溯
- 在向上回溯中，若某一个块需要插入记录，则在回溯到该节点前，它的子节点已经尝试过将记录插入该块了。这也就意味着，当在回溯的 `while` 循环中将某个块 id 出栈，并准备开始处理这个块 id 指向的块时，只有三种情况：
  - 该块不需要被插入记录
  - 该块被成功插入了记录
  - 该块需要插入记录，但失败了，于是记录尚未被插入且该块需要分裂
- 上面的三种情况中，前两种都不需要进行任何额外操作，可以继续向上回溯；但第三种则需要分裂出一个新块、移动记录，再将新块的最小键插入到父节点中。因此：
  i. 为了明确该块是否属于第三种情况，需要一个布尔变量 `needToSplit` 来标识。在上一轮循环中，若该块的子节点尝试插入该块且失败，则 `needToSplit = true`；
  ii. 因为子节点并没有插入成功，所以还需要一个 `iovec` 数组 `rec` 来保存子节点要插入的记录。

由上述分析，给出主循环 `while(!stk.empty)`（其中 `stk` 为栈）的实现如下：

1. 获取栈顶块 id 指向的块。
2. 调用 `searchRecord` 搜索，将结果（记录在 `slots` 数组中的下标）赋值给 `ret`。
3. 若该块为叶节点：
   i. 出栈。注意原栈顶元素是该叶节点的块 id。
   ii. 获取 `ret` 处的记录，并用 `memcmp` 比较该记录与待插入记录的主键是否相同。若相同，直接返回插入失败。
   iii. 否则，用 `insertRecord` 往叶节点中插入记录，并将返回值存在变量 `pret` 中。

iv. 判断是否插入失败，即是否需要分裂。若是，则进行与 project 1 的 `update` 函数中类似的操作：分裂新块、插入新记录。**注意要维护叶节点的单链表。**
v. 获取新块的**最小键** k，即其 `slots` 数组中下标为0的元素所对应记录的主键。
vi. 获取新块的块 id i，将 (k, i) 赋值给 `iovec` 数组 `rec` 中（其中 k 为主键）。
vii. 获得栈顶元素对应的块，该块即原块的父节点。
viii. 尝试将 `rec` 插入到父节点中。若插入失败，则将 `needToSplit` 置为 `true`。
ix. 叶节点处理完毕，开始回溯，进入一个嵌套的 `while` 循环。在该循环中：
    a. 出栈，并获得对应的块。比如第一次循环获得的便是叶节点的父节点。
    b. 如果 `needToSplit == true`，则首先将其置为 `false`，并进行分裂块、在本节点中插入 `rec`、在父节点中插入 (k, i) 记录的操作。此处 k 和 i 的含义与上文中的类似。实际上，相当于执行一遍与叶节点往自身插入记录的过程类似的逻辑。从这里可以看出，整个回溯过程中的分裂在实质上是递归的。
x. 注意：出了嵌套的 `while` 循环后，即栈为空时，`needToSplit` 仍可能为 `true`，这表明**根节点需要分裂，因而要产生新根**。该情况与上述回溯过程中的相关部分的不同点在于：
    a. 在分裂出一个新块之后，还需要调用 `table_->allocate()` 来产生另一个新块作为新根；
    b. 对于名为 `super` 的超块，还需要调用 `super.setRoot(rootId)` 来维护超块中的根 id。

4. 若该块为内节点，则需要分三种情况，其中一种情况是由于 `remove` 的设计而出现的，在这里先埋个伏笔。其余两种是：
    i. `ret` 大于等于 `slot` 总数：将最大的 `slot` 对应的块 id 压栈。
    ii. `ret` 在 `slots` 数组的下标范围内：此时的处理与 `search` 中在向下搜索时对于内节点情况的处理相同，即注意分 `memcmp` 返回0、`ret > 0`、`ret == 0` 三种情况讨论。

值得一提的是，project 2 的实现复用了 project 1 中对单个块的 CRUD 函数。如在 `insert` 中复用了 `insertRecord`，接下来在 `remove` 中也会复用 `removeRecord`，而 `searchRecord` 则在需要在单个块中搜索时都会用到。

说明

有必要再次说明一下为什么我基本采用了以文字来描述实现（而非直接贴代码），并且将整个实现过程描述得比较细致的原因。一方面，代码中有太多与核心无关的部分（字节序、缓冲区、获取块中记录的繁琐流程等）；另一方面，我认为仅描述大致实现思路是很容易的，但在真正的实现中，有许多细节和不同的情况需要考虑，因此如果不按照函数流程来解释清楚，则不容易把实现思路阐述得清晰到位，同时也难以表现出来工作量。

删除

终于来到三个 project 中最难啃的硬骨头了！或许读者认为 `insert` 的逻辑已经够繁杂的了，

但你将会看到——**`insert` 的麻烦程度还远不及 `remove` 的一半。**

算法流程

和上文的两个函数一样，仍然需要先查找到待删除的记录所在的叶节点，若待删除记录的主键不存在则返回，否则：

1. 删除该记录
2. 若该节点为根，则直接返回
3. 若叶节点键数不足（即小于 $\lfloor \frac{M}{2} \rfloor$，称为下溢），则：
    i. 尝试借键：找到兄弟节点中键最多的，记为 S（注意，由于该节点必不为根，所以一定有至少一个兄弟节点），向其借键。若借出键后 S 也下溢，则退还键；否则，更新父节点中分割该节点和 S 的键。
    ii. 若借键失败，则合并：将 S 与该节点合并为同一个节点，并更新父节点中的记录。
4. 在回溯中，对每个节点执行是否下溢的判断，若是，则进行借键或合并的操作。

实现

`remove` 的实现需要用到借键函数 `borrow` 及 合并函数 `merge`，先介绍这两个函数的实现会使 `remove` 中的一些设计显得更自然。比如，与 `search` 和 `insert` 不同，栈在 `remove` 中的元素类型是 `std::pair<unsigned int, int>`，分别存放块 id 及该块在其父节点的 `slots` 数组中的下标；而如此设计的原因就与借键与合并函数的设计有关。

值得一提的是，先尝试借键、失败后再合并的顺序是有原因的：借键只涉及单个记录的转移，而一旦合并就要移动某个块中所有的记录，显然合并的开销比借键的开销要大得多。

借键

为什么要跟踪对应的父节点中的下标?

记下溢的块为 B，则借键的首要问题便是——往哪个兄弟节点借键。显然，这需要知道 B 对应了父节点中的哪一个指针。如果 B 对应了最左指针，则它没有左兄弟，只能向右兄弟借键；如果 B 对应从左往右第二个指针，则它的左兄弟需要由父节点的 `next` 字段给出，而不是父节点的某条记录；如果 B 对应最右指针，则它只能向左兄弟借键。因此，`borrow` 函数需要接受一个**指示 B 在父节点中下标的形参 `idx`** ——也即栈元素类型之所以如此设计的原因。

可以考虑一下为什么在 `insert` 中同样需要往父节点中插入记录，但却不需要跟踪某个块在父节点中的下标：在 `insert` 中，插入父节点时调用的 `insertRecord` 会找到插入的正确位置，因此没有跟踪下标的必要。

然而，如果想要不在 `remove` 中跟踪下标，则为了获取 B 在父节点中对应了哪个指针，需要：

1. 获取 B 的最小键 k
2. 搜索 k 在父节点所有记录中的位置

3. 注意：lower bound 带来的分类讨论问题仍然存在！

因此，尽管不跟踪下标也是可行的，但将带来许多不必要的开销。

确定左/右兄弟

说明：所有“兄弟”均指代的是 B 的兄弟节点。

思路很简单：当左/右兄弟存在时，通过父节点来获取它们的块 id，接着借到对应的块；接着，比较它们的 `freesize` 大小，选择 `freesize` 更小的兄弟借键（因为它更可能借出键后不会下溢）。

对于左兄弟：

1. 判断 `idx != -1`，即 B 不对应父节点中最左指针。
2. 若 `idx == 0`，则左兄弟由父节点的 `next` 获取。
3. 否则，获取父节点 `slots[idx - 1]` 对应记录所指向的块。

对于确定右兄弟，只需判断是否有 `idx < getSlots() - 1` 即可。接着便是比较二者的 `freesize` 的大小（如果有），以确定向哪个兄弟借键。

借键

要借键，首先便要知道记录的结构，而记录结构又由节点类型决定。若为内节点，那么记录结构固定，是一个（主键，块 id）的键值对；但若为叶节点，那么记录结构就无法提前得知了。由此引出需要传给 `borrow` 的第二个参数——`std::vector<struct iovec> &dataIov`，用来指示叶节点的记录结构。通过以下代码可以确定传给 `getRecord` 的实参究竟是哪一个：

```
std::vector<struct iovec> &iovRef =
    sibling.getType() == BLOCK_TYPE_DATA ? dataIov : iov;
```

借键的核心思路是：

1. 先根据向左还是右兄弟借键，确定借最右还是最左键。
2. 删除兄弟中该条记录。
3. 判断兄弟是否下溢：
    i. 若下溢，则重新插入该条记录，返回 `false`。
    ii. 否则：
        a. 在 B 中插入该记录。
        b. 删除父节点中原先分隔了 B 和兄弟的记录（称为*中位键*）。注意：如果向左兄弟借键，则该记录在父节点中的下标就是 `idx`；但若向右兄弟借键，则为 `idx + 1`。
        c. 在父节点中插入新的中位键。说起来容易，但在获取中位键时仍然要分左/右兄弟讨论：若为左，则中位键是 B 中 `slots[0]` 对应记录的主键（最小键）；若为

右，则是右兄弟的最小键。

在最后获取新的中位键时，有一个细节值得说明：

首先，**主键的数据类型是不能预先确定的**，在本报告的开头部分已经阐释了如何通过 `vector` 声明 `char` 数组来解决字节数不定的问题；另一方面，在获取中位键时，需要先获取子节点的最小记录（存储到 `iovRef`），再获取该记录的主键，接着将主键值赋给用于插入父节点的 `splitIov` 的第一个字段。这里涉及到的问题是：

1. 不能直接解引用 `iovRef` 的指针成员来赋值给 `splitIov` 所指向的内存，因为数据类型（核心为字节数）不定。
2. 当兄弟为叶节点时，主键所在下标并不一定为0。

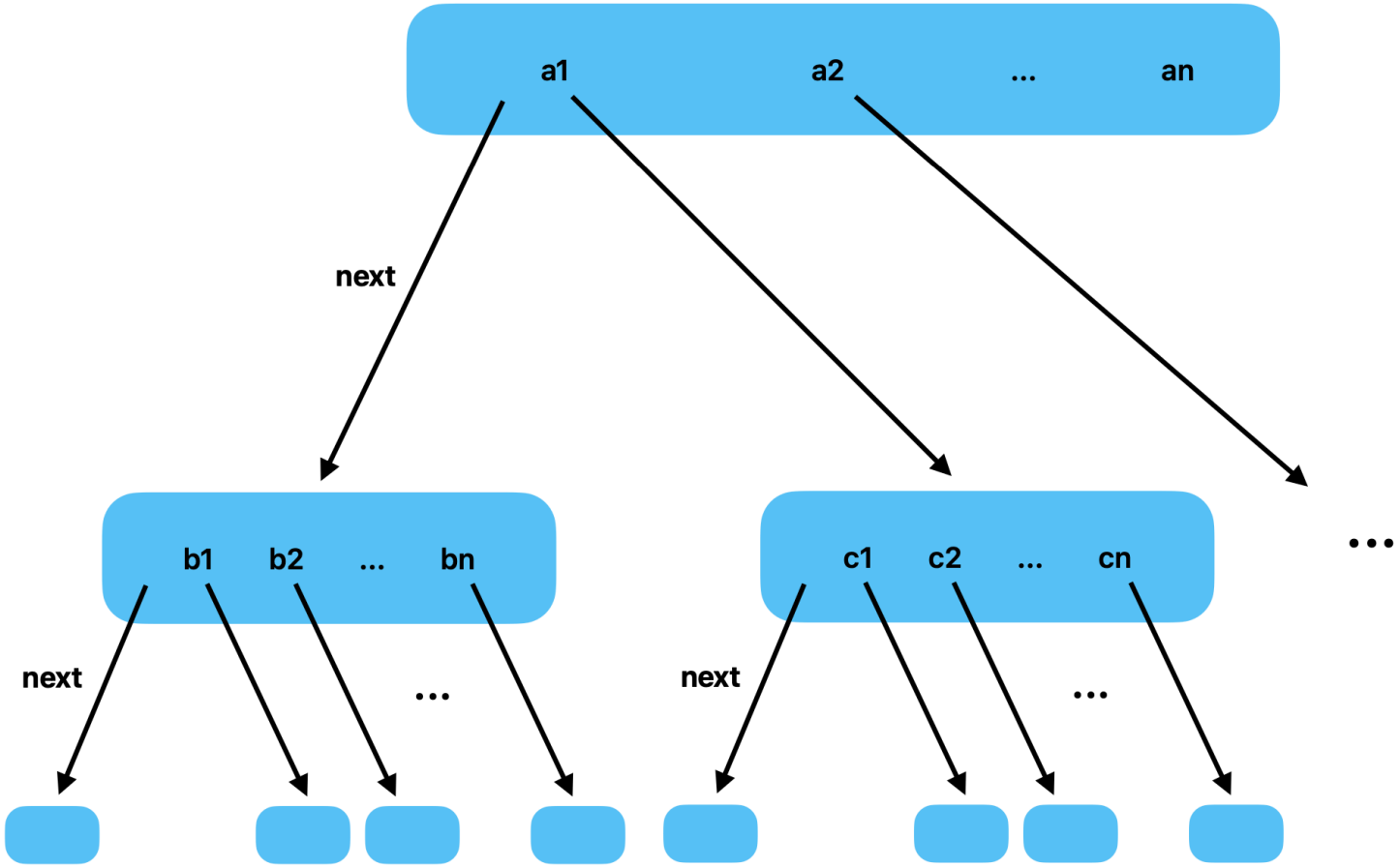
因此直接使用 `memcpy` 拷贝，代码为：

```
// splitIov 的主键字段指向 splitKey
memcpy(
    &splitKey[0],
    iovRef[data.getType() == BLOCK_TYPE_DATA ? keyIdx : 0].iov_base,
    keySize);
```

麻烦的右兄弟

若向右兄弟借键，则开始时会更加棘手一点：

首先要明确：`next` **字段在叶节点和内节点中的含义是不同的**。在叶节点中，`next` 指向单链表中的后继节点，因此与借键无关；而在内节点中则是有关的。可以用一张图更好地说明问题（没有画出叶节点间的单链表）：



左子节点的 bn 指向的子树的键值范围为 $[bn, a1)$，而右子节点的 $next$ 指向的子树的键值范围为 $[a1, c1)$。因此，在借键时需要分两种情况：

- 右兄弟为叶节点：中位键即右兄弟 `slots[0]` 对应记录的主键。
- 右兄弟为内节点：
  i. 沿着 $c1$ 左侧的 `next` 指针到达右兄弟的子节点（记为 C）；
  ii. 获取 C 的最小键 k；
  iii. 将 $c1$ 指向的子节点作为右兄弟 `next` 指向的块；
  iv. 删除右兄弟 `slots[0]` 对应的记录；
  v. 若右兄弟下溢，则重新插入被删除的记录，并重新设置 `next` 为原来的值，返回借键失败；
  vi. 否则，将 $(k, block\_id(C))$ 构成的记录插入到 B（即图中左节点）中，并获取右兄弟现在的最小键，进而更新父节点中的中位键。这部分的流程在“核心思路”中已经涉及。

注意：右兄弟为内节点时，**上述步骤的操作顺序不能颠倒**！这是因为，判断是否下溢的根据是该块的 `freesize` 大小是否大于当块为空时的 `freesize` 大小的一半。因此，必须要先删除右兄弟中的键，才能判断其是否下溢，接着才能决定是否插入 B 并更新父节点。

合并

合并的第一步与借键相同，都是先确定左、右兄弟（如果有）的 `freesize`。不同的是，B 将选择 `freesize` 更大的兄弟进行合并。

合并方向是固定的

不论是向左还是右兄弟合并，总是将选中的两个节点中的右节点合并到左。记左节点为 L，右节点为 R。若节点为叶节点，则在维护单链表时，执行：

1. `L.next = R.next`
2. 合并 L、R

而如果将左合并到右，一方面需要判断 L 是否还有左兄弟；另一方面，若还有左兄弟，则还需将该左兄弟的 `next` 从原先指向 L 转为指向 R。然而链表是单向的，因此只能通过父节点来找到 L 的左兄弟，这将非常麻烦。

合并

为了代码更加模块化，我把给定需要合并的两个块 L、R 进行记录移动的逻辑封装到了函数 `mergeBlock` 中。其函数签名为：

```
void mergeBlock(
    unsigned int blockid,
    unsigned int parentId,
    int blockIdx,
    std::vector<struct iovec> &dataIov);
```

其中，调用该成员函数的对象即 L，`blockid` 即 R 的块 id，`blockIdx` 则为 R 在父节点 `slots` 数组中的下标。为什么要传入的是 R 对应的下标，而不是 L 的呢?

为什么是 R?

首先考虑应当如何移动记录。显然，可以使用一个循环，每轮循环中执行：

1. 删除 R 的一个记录
2. 将该记录插入到 L 中

显然可以对 R 调用 `removeRecord` 来删除记录。然而，可以对 L 调用 `insertRecord` 来插入记录吗?

如果记录字段长度可变，且 L、R 为叶节点，那么在把 R 的记录插入到 L 的过程中，**L 是可能溢出的**。因此，若二者为叶节点，则需要调用会处理分裂的 `insert` 来插入。然而，`insert` 是从根节点开始搜索的，这里有一个非常容易忽略的地方：**如果不先把父节点中指向 R 的记录删去，则 `insert` 会搜索到错误的位置**！这是因为，R 中的记录都是大于 L 中记录的，所以如果中位键没有被删去，那么通过 `insert` 来插入原 R 中的记录时，仍然会插入到 R 中，而不是我们想要的目的节点 L。这便是要传入 R 在父节点 `slots` 数组中的下标的原因。

读者可能会问：内节点的移动就没有这样的问题吗？这是因为，本实验假设了主键为定长类型，而内节点的记录是由主键和块 id 构成的键值对，因此长度是固定的。所以为了提升效率，

可以用不需要从根开始搜索的 `insertRecord` 来插入 L。内节点部分的代码如下：

```
while (data.getSlots()) {
    getRecord(data.buffer_, data.getSlotsPointer(), 0, tmpIov);
    insertRecord(tmpIov);
    data.removeRecord(tmpIov); // 为了可重用该 block
}

// 移动 data 的最左指针
DataBlock child;
child.setTable(table_);
child.attachBuffer(&bd2, data.getNext());

getRecordByIndex(
    child.buffer_, child.getSlotsPointer(), 0, tmpIov[0], keyIdx);
tmpVal = data.getNext();
intType->htobe(&tmpVal);
insertRecord(tmpIov);

kBuffer.releaseBuf(bd2);
```

由于前文为了篇幅考虑而没有给出很多代码，所以借由该段代码解释一下几个常见的操作。

1. 首先，`data.getSlots()` 条件用于判断 R 是否还有记录，若有，则先由 `getRecord` 获取 `slots[0]` 对应的记录（因为每次删除都会挤压 `slots`，所以只要有记录，`slots[0]` 就必然有记录），将该记录存到 `tmpIov` 中。
2. 插入和删除函数已在前文中介绍。
3. 此处和前文中“麻烦的右兄弟”部分的处理是一致的，即先获取 `data.getNext()` 所指向的块，再将该块的最小记录的主键通过 `getRecordByIndex` 赋值给 `tmpIov` 的第一个字段；接着，因为 `tmpIov[1].iov_base` 指向 `tmpVal`，所以可以把 R 的 `next` 所存储的块 id 赋给 `tmpVal` 来修改 `tmpIov`。注意：`next` **是以主机序存储的**，所以在把该记录写入 L 之前，需要调用 `htobe` 来转换为大端序。
4. 最后，每一个 `attachBuffer` 都必须要搭配一个 `releaseBuf` 来释放缓冲区。

实际上，这些代码大部分都是自解释的，和报告中的实现说明对照来看也很容易看懂。这也是前文没有贴出很多代码的原因之一。

伏笔回收

此处需要回收前文的伏笔：在 `insert` 的内节点情况中，曾提到有一种情况是因为合并逻辑的设计而产生的。该情况的代码为：

```
if (!data.getSlots()) {
    stk.push(data.getNext());
}
```

这是因为：在合并两个叶节点前，需要先删除 R 在父节点中对应的记录，**这意味着父节点可能暂时只有 `next` 指针而无记录**。为了搜索能正常进行，需要对该情况单独处理，即将 `next` 对应的块 id 压栈（即接下来搜索 L）。

回到 `merge` 中

`mergeBlock` 返回后， `merge` 便只剩下维护单链表的逻辑了，这里不再赘述。

删除

终于回到主函数 `remove` 了！ `remove` 的主循环和 `insert` 在结构上很相似，可以分成向下搜索到叶节点、叶节点进行删除及下溢处理、向上回溯三部分。有了前面对函数体流程的详细讲解，读者可以结合注释比较容易地理解源代码，因而在此我不打算再次从初始化开始一步步地分析，而是挑选核心要点来讲解。

基本框架

在 `insert` 的向上回溯中，基本逻辑是每到一个节点，执行：

1. 检查是否需要分裂
2. 若是，则执行分裂逻辑，并向父节点中插入中位键

这个循环需要有一个起点，那便是在回溯前，叶节点需要先往其父节点插入中位键，这样循环中的动作顺序才合理。

`remove` 的向上回溯过程也可以如此拆解，即每到一个节点，执行：

1. 检查是否下溢
2. 若是，则尝试借键或合并

显然，第二步应当会进一步导致父节点可能产生下溢，否则这个循环的递归结构是不成立的。而父节点会下溢的原因便是：在 `merge` 时，父节点会被删去一条记录（中位键），而在中位键被删后 `merge` 并不负责父节点是否下溢的检查。

自然， `remove` 也需要一个回溯的起点，即在回溯前，叶节点需要：

1. 调用 `removeRecord` 删除记录，若记录不存在则返回删除失败。
2. 通过与 `super.getRoot()` 比较，判断自己是否为根节点。若为根节点则无需处理下溢问题，直接返回成功。
3. 判断是否下溢。若是，则通过栈顶元素获取父节点的块，并先后尝试 `borrow` 和 `merge` 。

接着便可开始向上回溯，因为父节点此时可能被删除了记录，所以能接上“检查是否下溢”的逻辑。实际上，封装了 `borrow` 和 `merge` 函数之后，借键与合并的逻辑显得非常简洁：

```
// preRet == stk.top().second
if (!parent.borrow(preRet, data.getSelf(), iov)) { // 借键失败
    parent.merge(preRet, data.getSelf(), iov);
}
```

读者可能注意到：在第二步中，我并没有处理根节点为空的情况。这是因为在第一步中，**若根节点为空则会因为记录不存在而直接返回。**

删除根

有一种特殊情况值得说明——根节点下溢。此处的根节点为内节点，因为叶节点的情况已经在上文的“回溯前”部分讨论过了。

根节点是B+树中比较特殊的存在，因为它没有兄弟节点可以借键或合并。这表明，根节点下溢时可分为两种情况：

1. 没有记录，即只剩一个 `next` 指针
2. 其他

第二种情况显然无需进行更多的处理。对于第一种情况，**只有可能根节点原来唯二的子节点合并了**，因此需要将根节点删去，并将超块的 `root` 字段设置为原根节点的 `next` 所存储的值。

向下搜索

前文已经介绍了三部分中第二、三部分的注意点，这里简要地说明一下第一部分的要点。向下搜索的过程大体上与 `search` 和 `insert` 的相关部分是一样的（都要注意 lower bound 的陷阱！），最大的区别在于，由于栈元素类型的变化，在调用 `searchRecord` 获得接下来搜索的节点（记为 N）在本节点的 `slots` 数组中的下标（`ret`）后，要将 N 的块 id 和 `ret` 一并压栈，以在回溯时的借键与合并中使用。

更新

`update` 仍然是先删除再插入。与 `updateRecord` 的情况不同，`insert` 和 `remove` 已经处理了上溢和下溢，因此 `update` 的实现非常简单：

```
if (remove(iov) == S_OK && insert(iov) == S_OK)
    return S_OK;
else
    return EFAULT;
```

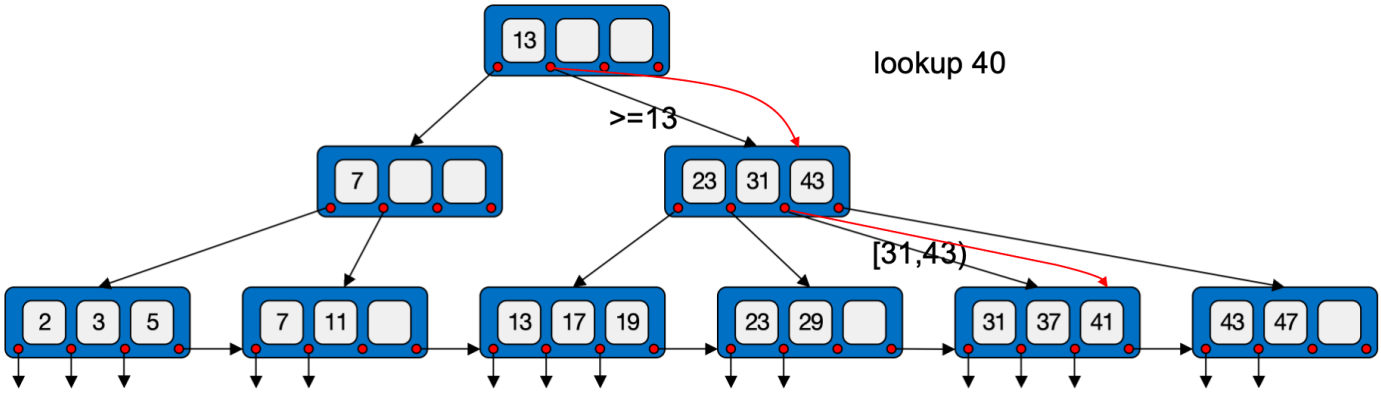
测试

project 2 的测试在 `TEST_CASE("IndexTest", "[p2]")` 中。与 project 1 类似，也是在 `.\make` 之后通过 `.\bin\utest [p2]` 来单独运行与 project 2 相关的测试用例。

搜索

为了方便编写测试用例，除了用于为 `iovec` 赋值的 `setIdxIov` 函数外，还编写了 `createBlock` 函数。这是因为对 `search` 的测试应尽量不依赖其他的函数，所以需要在测试 `search` 前“手动”创建一颗B+树，即直接通过 `table->allocate()` 来创建块，并手动设置块的类型和 `next` 字段。另外， `createBlock` 接受一个以 `iovec` 为元素类型的二维数组，并在一个循环中将二维数组中的所有记录插入创建的块中。

在测试用例中，首先创建一颗如下图所示的B+树：



每次创建块时，需要用 `REQUIRE` 检查 `createBlock` 的返回值，确保所有记录均正确插入。

测试分为6种情况：

1. 搜索最小键
2. 搜索最大键
3. 当搜索的键刚好与根节点中的某个键相等时
4. 当搜索的键刚好与内节点（不包括根）中的某个键相等时
5. 当搜索的键只出现在叶节点，而内节点中没有时
6. 当搜索的键不存在时

在用例中，与该6种情况对应的查询键分别为：2、47、13、43、37、12。查询的逻辑为：

1. 使用一个变量 `key` 存储要查询的键值，它对应了 `search` 的形参 `keybuf`；
2. 因为先前插入的记录均共有2个字段，所以声明一个有2个元素的 `iovec` 数组 `iov`，对应了形参 `iov`；
3. 调用 `search`，用 `REQUIRE` 检查返回值。
4. `search` 搜到的记录会存储在 `iov`，于是检查 `iov` 的各字段与之前插入的记录是否相符（硬编码）。

插入

再次强调：Catch2 的机制下，前一个 `SECTION` 中对内存作出的修改在之后的 `SECTION` 内仍然保留！

为了确认这一点，先检查上一 `SECTION` 中构建的B+树仍然存在：

```
long long preKeys[] = {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47};

for (int i = 0; i < 15; ++i) {
    bigint->htobe(&preKeys[i]);
    REQUIRE(data.search(&preKeys[i], sizeof(long long), iov) == S_OK);
}
```

接着通过大规模插入来测试代码。我先在现有的各个键间分散地插入了一些键，它们的分布如下：

```
// keys 存记录主键，vals 存记录第二个字段的值
std::vector<long long> keys = { 1, 8, 12, 15, 22, 30, 33, 44, 46, 48 };
std::vector<unsigned int> vals = { 10, 80, 120, 150, 220, 300, 330, 440, 460,
480 };
```

接着再递增地加入大量记录：

```
for (int i = 50; i < 2051; i += 2) {
    keys.push_back(i);
    vals.push_back(i * 10);
}
```

在上图的B+树中，各个块的 id 以 BFS 顺序分别为：1、2、3、4、5、6、7、8、9。这是在 `createBlock` 中进行了验证的：即传入期待的块 id（注意，`table->allocate()` 产生的块 id 是递增的），随后在 `createBlock` 中执行对实际产生的 id 与期待的 id 是否一致的检查：

```
// blockid 是 createBlock 接受的实参
unsigned int selfid = table->allocate();
REQUIRE(selfid == blockid);
```

知道图中各个块的 id 使得其他检查可以更加方便的进行。比如，为了验证在大规模插入中有块的分裂，令变量 `last` 存键最大的叶节点，于是在大规模插入前，通过 `REQUIRE(!last.getNext());` 来验证 `last` 此时没有后继节点；大规模插入后，则通过 `REQUIRE(last.getNext() == 10);` 验证其有了后继节点（`last` 的块 id 为9，所以下一个分配的必为10），即发生了分裂。

但仅验证插入成功、分裂成功仍不足，所以遍历之前插入的所有键值，在下面的循环中通过 `search` 验证记录都被插入到了正确的位置：

```
for (int i = 0; i < keys.size(); ++i) {
    REQUIRE(data.search(&keys[i], sizeof(long long), iov) == S_OK);
    REQUIRE(*(unsigned int *) iov[1].iov_base == vals[i]);
}
```

在上面的代码中，`iov[1].iov_base` 指向的是 `search` 搜到的记录的第二个字段，`vals[i]` 则是应当查到的记录的第二个字段（注意，`iov[1].iov_base` 并不指向 `vals[i]`），通过该条判断，可以验证取到的记录是正确的。

删除

在测试删除的用例的最开始，也先检查上一 `SECTION` 留下来的B+树（注意，是大规模插入后的），在此不再赘述。

接着，进行大规模删除，只留下在 `SECTION("search")` 中插入的记录（即上文的B+树图中叶节点的键）：

```
for (int i = 0; i < preKeys.size(); ++i) {
    tmpKey = preKeys[i];
    tmpVal = (unsigned int) preKeys[i] * 10;
    REQUIRE(data.remove(iov) == S_OK);
}
```

这是一个将深度为3的B+树删到只剩一个根节点（1号块）的过程。为了检查 `remove` 中对删除下溢的正确处理，先在大规模删除前检查1号块确实有子节点：

```
// 检查1号块 next 非空，即有子节点且为2号块
REQUIRE(data.getNext() == 2);
```

在删除后，验证其已无后继节点：

```
// 检查1号块的 next 已空，即发生了合并
REQUIRE(!data.getNext());
```

更新

刚进入 `SECTION("update")` 时，B+树只剩下一个节点，这显然不足以满足测试的需求，所以在一开始先进行大规模插入。在插入的同时，记录下插进去的键值，并且包含在前一 `SECTION` 中剩下来的键，把它们都存在 `preKeys` 数组中。

接着遍历 `preKeys`，对每个键值调用 `REQUIRE(data.search(&preKeys[i], (unsigned int) sizeof(long long), iov) == S_OK);` 来验证记录存在。

随后，将记录第2个字段的值均更新为原来的2倍，并通过 `REQUIRE(data.update(iov) == S_OK);` 在更新记录的同时检查其返回值。

和之前的测试用例一样，仅检查返回值并不能完全说明实现是正确的，还需要再次获取块中的记录并与更新的值进行对比：

```
for (int i = 0; i < preKeys.size(); ++i) {
    REQUIRE(
        data.search(&preKeys[i], (unsigned int) sizeof(long long), iov) ==
S_OK);
    intType->betoh(&val);
    REQUIRE(val == preVals[i] * 2);
}
```

在上面的代码中，会将搜到的记录存在 `iov` 指向的内存中，而 `iov[1].iov_base` 是指向 `val` 的。因此，通过验证 `REQUIRE(val == preVals[i] * 2);`，便可确定记录中的值已成功更新。

测试结果

```
PS C:\Users\yu\Projects\db\build> .\bin\utest [p2]
Filters: [p2]
===============================================================================
All tests passed (8970 assertions in 4 test cases)
```