NVMLab

代码文件

- $./code/mian-queue.cpp$：使用队列进行实现的代码
- $./code/mian-treap.cpp$：使用无旋treap进行实现的代码

队列版本

主体思路

- 在NVM上使用队列对数据进行维护。
- 每次读入一对key和$value$时，在队列末尾开一个节点，储存这对值。
- 程序中断后进行恢复时，读取NVM上队列中的每一对key和$value$，然后使用map对值进行存储，对数据进行恢复。
- 查询时，使用map的相关功能进行查询。

优化

- 使用$unordered_map$维护每个键值在队列中的位置，如果遇到一个已经出现过的键值（$unordered_map$中存在），那么可以找到这个键值在队列中的位置，直接修改队列中的$value$。保证队列中的元素的键值不互相同，可以减少队列中元素的数量，在恢复的过程中可以减少对map的修改，从而降低运行时间。
- 根据CPU的缓存策略，可以使用$memcpy$代替$pmemobj_memcpy_persist$，可以使代码效率大为提高，但是保证正确性的具体原因未知，故代码中**仍然使用**$pmemobj\_memcpy\_persist$。
- 观察测试代码可以发现，在实现过程中不需要支持回滚，即可以通过放弃回滚能力，来优化程序性能。

  阅读**main.cpp**可以发现，模拟器通过调用$exitPrintGrade$来打印成绩并结束程序。而中止程序只会发生在$nextQuery$过程中和$pmemSyncArrive$过程中。

  对于$nextQuery$，这个函数是在获取操作的过程中被调用，只需要保证在获取询问之前数据结构是正确的，在该过程中退出不会影响队列的结构，即不会影响程序的正确性。

对于$pmemSyncArrive$，观察发现，有且仅有在每个事务(transaction)结束执行和调用TX_ALLOC时，该函数会对参数now_op、$pmem_op_sum$和$pmem_rounds$进行修改，程序是否中止与这三个数的关系有关。在$mian.cpp$中，仅在新建节点的时候使用了事务，且在该事务中使用了TX_ALLOC。所以，只要新建节点后，再把节点加入队列，这样可以避免由于程序中止对队列造成的破坏。

核心代码

- 数据结构定义

```
struct my_root {
    TOID(struct tree) root;
}; // 创建根节点

struct tree {
    int root, cnt;
    TOID(struct node) nd[];
}; // 定义一个名字叫tree的队列

struct node {
    char key[KEY_LEN + 1], val[VAL_LEN + 1];
}; // 队列中每个节点的结构
```

- 新建节点

```
inline int INSERT(PMEMobjpool *pop, struct tree *p, const char
*key, const char *val) {
    int pos = -1;
    TX_BEGIN(pop) {
        pos = (p->cnt + 1); // 获得新节点的位置
        TOID(struct node) x = TX_ALLOC(struct node, sizeof(struct
node));
        // 申请对应大小的空间
        pmemobj_memcpy_persist(pop,
                               D_RW(x)->key,
                               key,
                               KEY_LEN);
        D_RW(x)->key[KEY_LEN] = '\0'; // 拷贝key
        pmemobj_memcpy_persist(pop,
                               D_RW(x)->val,
                               val,
                               VAL_LEN);
```

```
        D_RW(x)->val[VAL_LEN] = '\0'; // 拷贝value
        p->nd[pos] = x; // 记录新节点
    } TX_END
    p->cnt += 1; // 将新节点加入队列，避免回滚
    return pos;
}
```

- 恢复数据

```
if (!create) {
        struct tree *p = D_RW(rootp->root);
        int n = p->cnt;
        for (int i = 1; i <= n; ++i) {
            state[D_RO(p->nd[i])->key] = D_RO(p->nd[i])->val;
        } // 读取队列中的每一个元素，并且加入到map中
    }
```

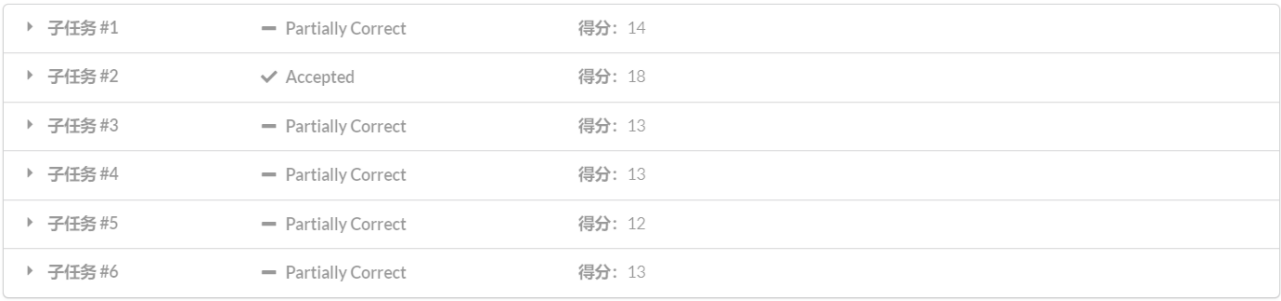
- 处理SET、GET和NEXT操作

```
while (1) {
        Query q = nextQuery();
        switch (q.type) {
            case Query::SET:
                if (mp.count(q.key))
                    CHANGE(pop, D_RW(rootp->root), mp[q.key],
q.value.c_str());
                    // 如果已经出现过，在修改队列中的元素
                else {
                    int pos = INSERT(pop,
                                     D_RW(rootp->root),
                                     q.key.c_str(),
                                     q.value.c_str());
                    mp[q.key] = pos;
                } // 如果没有出现过，才执行插入操作
                break;

            case Query::GET:
                if (state.count(q.key)) {
                    q.callback(state[q.key]);
                } // 使用map判断元素是否出现过
                else {
```

```
                    q.callback("-");
                }
                break;

            case Query::NEXT:
                if (auto it = state.upper_bound(q.key); it !=
state.end())
                    q.callback(it->first);
                else
                    q.callback("-");
                break;
                // 使用map的upper_bound函数求下一个元素
            default:
                throw
std::invalid_argument(std::to_string(q.type));
        }
    }
```

算法性能

- 在oj上评测的成绩（86分）：



平衡树版本

主体思路

- 在NVM上使用无旋$treep$维护数据。
- 读入一对key和$value$，新建一个节点储存这对KV，然后将这个节点插入到平衡树上。
- 恢复数据时，直接加载对应的pool即可。对于查询，可以直接在平衡树上进行查询，后继的查询也可以在平衡树上完成。

优化

- 相比于使用队列维护KV，无旋$treep$可以直接在NVM上储存平衡树的结构，恢复时无需重新建树，对查询性能有一定程度的提高。
- 使用$unordered_map$维护每个键值所在的节点的编号。如果一个键值已经出现过了，那么直接修改对应节点的$value$。这样可以减少插入次数，也可以减少平衡树上的节点数量，提高SET操作的性能。
- 在实现过程中不需要支持回滚，可以通过放弃回滚能力，来优化程序性能。具体分析在上文已经体现了。
- 在一个node中储存16个节点，即每次调用TX_ALLOC的时候，申请一段更大的空间，后续新建节点的时候，直接使用这次申请到的空间，而不再单独申请空间。多个节点一起申请一次空间可以减少对NVM的读写次数，从而提高程序的性能。
- 对于无旋$treap$节点的id值（用于保持平衡树节点的平衡），使用特定的公式代替随机数，可以减少开销，优化程序的常数。

核心代码

- 数据结构定义

```
struct my_root {
    TOID(struct tree) root;
};  // 创建根节点

struct tree {
    int root, cnt;
    TOID(struct node) nd[];
};  // 创建一棵平衡树

struct node {
    char key[KEY_LEN * node_size], val[VAL_LEN * node_size];
    int ls[node_size], rs[node_size], id[node_size];
}; // 每个node中储存16个节点
```

- 新建节点

```
inline int NewNode(PMEMobjpool *pop, const char *key, const char
*val) {
    int pos = -1, offset;
    TX_BEGIN(pop) {
        pos = (p->cnt += 1);
        TOID(struct node) x;
        offset = (pos - 1) & tago;
```

```
        // 计算当前节点在node中的偏移量
        pos = (pos - 1) >> tag;
        // 计算当前节点所在node的编号
        if (offset == 0) {
            x = TX_ALLOC(struct node, sizeof(struct node));
            p->nd[pos] = x;
        } // 16个节点一起申请一次空间，储存在一个node中
        D_RW(p->nd[pos])->id[offset] = st = (st * 482711);
        // 计算无旋treap的id值
        pmemobj_memcpy_persist(pop,
                               D_RW(p->nd[pos])->key + offset *
KEY_LEN,
                               key,
                               KEY_LEN);
        pmemobj_memcpy_persist(pop,
                               D_RW(p->nd[pos])->val + offset *
VAL_LEN,
                               val,
                               VAL_LEN);
        // 拷贝数据
    } TX_END
    return p->cnt;
}
```

- 无旋$treap$的插入操作

  无旋treap的插入分为两个步骤：分裂与合并。假设插入的KV对为key和$value$。

  分裂过程，将平衡树分裂为两个二叉搜索树，第一棵搜索树的键值全都小于key，第二棵树的权值全部大于key。由于插入之前已经判断了当前键值是否已经存在，所以分裂过程中不会遇到相等的情况。分裂结束之后，将第一棵平横树与新插入的点（看成只有一个节点的平横树）合并，然后再将得到的平横树与第二课平横树合并，即可得到新的平横树。

  插入的期望时间复杂度为$O(\log n)$。

```
void split(int ro, int &r1, int &r2, const char *key) {
    if (!ro) {
        r1 = r2 = 0;
        return;
    }
    int pos = (ro - 1 >> tag);
    int offset = (ro - 1 & tago);
    // 计算所在的node和偏移量
```

```
    if (cmp(D_RO(p->nd[pos])->key + offset * KEY_LEN, key) == -1) {
        r1 = ro;
        split(D_RO(p->nd[pos])->rs[offset], (D_RW(p->nd[pos])-
>rs[offset]), r2, key);
    } // 将当前节点分到第一棵平横树中
    else {
        r2 = ro;
        split(D_RO(p->nd[pos])->ls[offset], r1, (D_RW(p->nd[pos])-
>ls[offset]), key);
    } // 将当前节点分到第二棵平横树中
}

void merge(int &ro, int x, int y) {
    if (!x || !y) {
        ro = x | y;
        return;
    }
    int px = (x - 1 >> tag), py = (y - 1 >> tag);
    int ox = (x - 1 & tago), oy = (y - 1 & tago);
    // 计算所在的node与偏移量
    // 根据无旋treap满足小根堆的性质进行合并
    if (D_RO(p->nd[px])->id[ox] < D_RO(p->nd[py])->id[oy]) {
        ro = x;
        int pos = (ro - 1 >> tag);
        int offset = (ro - 1 & tago);
        merge(D_RW(p->nd[pos])->rs[offset], D_RO(p->nd[pos])-
>rs[offset], y);
    } // 第一棵平横树的id值更小，先合并
    else {
        ro = y;
        int pos = (ro - 1 >> tag);
        int offset = (ro - 1 & tago);
        merge(D_RW(p->nd[pos])->ls[offset], x, D_RO(p->nd[pos])-
>ls[offset]);
    } // 第二棵平横树的id值更小，先合并
}

inline void INSERT(PMEMobjpool *pop, int r3, const char *key, const
char *val) {
    int r1 = 0, r2 = 0;
    split(p->root, r1, r2, key); // 对原树进行分裂
    merge(r1, r1, r3);
    merge(p->root, r1, r2);
```

```
    // 合并
}
```

- 查询节点与后继

```
int FIND(int ro, const char *key) {
    if (ro == 0) return -1; // 找不到节点
    int pos = (ro - 1 >> tag);
    int offset = (ro - 1 & tago);
    // 计算所在node和偏移量
    int flag = cmp(D_RO(p->nd[pos])->key + offset * KEY_LEN, key);
    if (flag == 0) return ro;
    if (flag == -1) {
        return FIND(D_RO(p->nd[pos])->rs[offset], key);
    } // key比当前大，递归右儿子
    else {
        return FIND(D_RO(p->nd[pos])->ls[offset], key);
    } // key比当前小，递归左儿子
}

void FIND_NEXT(int ro, const char *key) {
    if (ro == 0) return;
    int pos = (ro - 1 >> tag);
    int offset = (ro - 1 & tago);
    // 计算所在node和偏移量
    int flag = cmp(D_RO(p->nd[pos])->key + offset * KEY_LEN, key);
    if (flag <= 0) {
        FIND_NEXT(D_RO(p->nd[pos])->rs[offset], key);
    } // key比当前大，递归右儿子
    else {
        INDEX = ro; // 记录合法答案
        FIND_NEXT(D_RO(p->nd[pos])->ls[offset], key);
    } // key比当前小，递归右儿子，寻找更加确切的后继
}
```

算法性能

- 在oj上评测的成绩（100分）：

子任务 #1	Accepted	得分：18
子任务 #2	Accepted	得分：18
子任务 #3	Accepted	得分：16
子任务 #4	Accepted	得分：16
子任务 #5	Accepted	得分：16
子任务 #6	Accepted	得分：16