11. The AVL Tree and the Sorted Set

11.1 The Sorted Set Data Type

Based on the AVL tree in the last chapter, the sorted set data structure can be easily added. The structure definition:

```
struct ZSet {
    AVLNode *tree = NULL;
    HMap hmap;
};

struct ZNode {
    AVLNode tree;
    HNode hmap;
    double score = 0;
    size_t len = 0;
    char name[0];
};

static ZNode *znode_new(const char *name, size_t len, double score) {
    ZNode *node = (ZNode *)malloc(sizeof(ZNode) + len);
    assert(node);   // not a good idea in real projects
    avl_init(&node->tree);
    node->hmap.next = NULL;
    node->hmap.hcode = str_hash((uint8_t *)name, len);
    node->score = score;
    node->len = len;
    memcpy(&node->name[0], name, len);
```

```
    return node;
}
```

The sorted set is a sorted list of pairs of `(score, name)` that supports query or update by the sorting key, or by the `name`. It's a combination of the AVL tree and hashtable, and the pair node belongs to both, which demonstrates the flexibility of intrusive data structures. The `name` string is embedded at the end of the pair node, in the hope of saving up some space overheads.

11.2 Insertion and Deletion

The function for tree insertion is roughly the same as the testing code seen from the previous chapter:

```
// insert into the AVL tree
static void tree_add(ZSet *zset, ZNode *node) {
    if (!zset->tree) {
        zset->tree = &node->tree;
        return;
    }

    AVLNode *cur = zset->tree;
    while (true) {
        AVLNode **from = zless(&node->tree, cur) ? &cur->left : &cur-
        >right;
        if (!*from) {
            *from = &node->tree;
            node->tree.parent = cur;
            zset->tree = avl_fix(&node->tree);
            break;
        }
        cur = *from;
    }
}
```

`zless` is the helper function for comparing two pairs:

```
// compare by the (score, name) tuple
static bool zless(
    AVLNode *lhs, double score, const char *name, size_t len)
{
```

```
    ZNode *zl = container_of(lhs, ZNode, tree);
    if (zl->score != score) {
        return zl->score < score;
    }
    int rv = memcmp(zl->name, name, min(zl->len, len));
    if (rv != 0) {
        return rv < 0;
    }
    return zl->len < len;
}

static bool zless(AVLNode *lhs, AVLNode *rhs) {
    ZNode *zr = container_of(rhs, ZNode, tree);
    return zless(lhs, zr->score, zr->name, zr->len);
}
```

The insertion/update subroutines:

```
// update the score of an existing node (AVL tree reinsertion)
static void zset_update(ZSet *zset, ZNode *node, double score) {
    if (node->score == score) {
        return;
    }
    zset->tree = avl_del(&node->tree);
    node->score = score;
    avl_init(&node->tree);
    tree_add(zset, node);
}

// add a new (score, name) tuple, or update the score of the existing tuple
bool zset_add(ZSet *zset, const char *name, size_t len, double score) {
    ZNode *node = zset_lookup(zset, name, len);
    if (node) {
        zset_update(zset, node, score);
        return false;
    } else {
        node = znode_new(name, len, score);
        hm_insert(&zset->hmap, &node->hmap);
        tree_add(zset, node);
        return true;
    }
}
```

```
// lookup by name
ZNode *zset_lookup(ZSet *zset, const char *name, size_t len) {
    // just a hashtable look up
    // code omitted...
}
```

11.3 Rank-Based Queries

Here is the primary use case of sorted sets: the range query.

```
// find the (score, name) tuple that is greater or equal to the argument,
// then offset relative to it.
ZNode *zset_query(
    ZSet *zset, double score, const char *name, size_t len, int64_t offset)
{
    AVLNode *found = NULL;
    AVLNode *cur = zset->tree;
    while (cur) {
        if (zless(cur, score, name, len)) {
            cur = cur->right;
        } else {
            found = cur;    // candidate
            cur = cur->left;
        }
    }

    if (found) {
        found = avl_offset(found, offset);
    }
    return found ? container_of(found, ZNode, tree) : NULL;
}
```

The range query is just a regular binary tree look-up, followed by an offset operation. The offset operation is what makes the sorted set special, it is not a regular binary tree walk.

Let's review the `AVLNode`:

```
struct AVLNode {
    uint32_t depth = 0;
    uint32_t cnt = 0;
```

```
    AVLNode *left = NULL;
    AVLNode *right = NULL;
    AVLNode *parent = NULL;
};
```

It has an extra `cnt` field (the size of the tree), which is not explained in the previous chapter. It is used by the `avl_offset` function:

```
// offset into the succeeding or preceding node.
// note: the worst-case is O(log(n)) regardless of how long the offset is.
AVLNode *avl_offset(AVLNode *node, int64_t offset) {
    int64_t pos = 0;    // relative to the starting node
    while (offset != pos) {
        if (pos < offset && pos + avl_cnt(node->right) >= offset) {
            // the target is inside the right subtree
            node = node->right;
            pos += avl_cnt(node->left) + 1;
        } else if (pos > offset && pos - avl_cnt(node->left) <= offset) {
            // the target is inside the left subtree
            node = node->left;
            pos -= avl_cnt(node->right) + 1;
        } else {
            // go to the parent
            AVLNode *parent = node->parent;
            if (!parent) {
                return NULL;
            }
            if (parent->right == node) {
                pos -= avl_cnt(node->left) + 1;
            } else {
                pos += avl_cnt(node->right) + 1;
            }
            node = parent;
        }
    }
    return node;
}
```

With the size information embedded in the node, we can determine whether the offset target is inside a subtree or not. The offset operation runs in two phases: firstly, it walks up along the tree if the target is not in a subtree, then it walks down the tree, narrowing the

distance until the target is met. The worst-case is `O(log(n))` regardless of how long the offset is, which is better than offsetting by walking to the succeeding node one by one (best-case of `O(offset)`). The real Redis project uses a similar technique for skip lists.

It is a good idea to stop and test the new `avl_offset` function now.

```
static void test_case(uint32_t sz) {
    Container c;
    for (uint32_t i = 0; i < sz; ++i) {
        add(c, i);
    }

    AVLNode *min = c.root;
    while (min->left) {
        min = min->left;
    }
    for (uint32_t i = 0; i < sz; ++i) {
        AVLNode *node = avl_offset(min, (int64_t)i);
        assert(container_of(node, Data, node)->val == i);

        for (uint32_t j = 0; j < sz; ++j) {
            int64_t offset = (int64_t)j - (int64_t)i;
            AVLNode *n2 = avl_offset(node, offset);
            assert(container_of(n2, Data, node)->val == j);
        }
        assert(!avl_offset(node, -(int64_t)i - 1));
        assert(!avl_offset(node, sz - i));
    }

    dispose(c.root);
}
```

11.4 New Commands & Testing

For now, we have implemented major functionalities of the sorted set. Let's add the sorted set type to our server.

```
enum {
    T_STR = 0,
    T_ZSET = 1,
};
```

```
// the structure for the key
struct Entry {
    struct HNode node;
    std::string key;
    std::string val;
    uint32_t type = 0;
    ZSet *zset = NULL;
};
```

The rest of the code is considered trivial, which will be omitted in the code listing.

Adding a Python script for testing new commands:

```
CASES = r'''
$ ./client zscore asdf n1
(nil)
$ ./client zquery xxx 1 asdf 1 10
(arr) len=0
(arr) end
# more cases...
'''

import shlex
import subprocess

cmds = []
outputs = []
lines = CASES.splitlines()
for x in lines:
    x = x.strip()
    if not x:
        continue
    if x.startswith('$ '):
        cmds.append(x[2:])
        outputs.append('')
    else:
        outputs[-1] = outputs[-1] + x + '\n'

assert len(cmds) == len(outputs)
for cmd, expect in zip(cmds, outputs):
```

```
    out = subprocess.check_output(shlex.split(cmd)).decode('utf-8')
    assert out == expect, f'cmd:{cmd} out:{out}'
```

Exercises:

1. The `avl_offset` function gives us the ability to query sorted set by rank, now do the reverse, given a node in an AVL tree, find its rank, with a worst-case of `O(log(n))`. (This is the `zrank` command.)
2. Another sorted set application: count the number of elements within a range. (also with a worst-case of `O(log(n))`.)
3. The `11_server.cpp` file already contains some sorted set commands, try adding more.

Source code:

- 11_client.cpp
- 11_server.cpp
- avl.cpp
- avl.h
- common.h
- hashtable.cpp
- hashtable.h
- test_cmds.py
- test_offset.cpp
- zset.cpp
- zset.h

Check it out