10. The AVL Tree: Implementation & Testing

10.1 The Sorted Set & The AVL Tree

While Redis is often referred to as a key-value store, the "value" part of Redis is not restricted to plain strings, lists, hashmaps, and sorted sets are quite nice things to have. Redis is also referred to as the "data structure server" due to its rich set of data structures.

Redis is often used as an in-memory cache, and when storing data in memory, there is an advantage of freely using data structures. The sorted set data structure in Redis is quite a unique and useful thing. Not only it offers the ability to sort your data in order, but also has the unique feature of querying ordered data by rank. If you put 20M records into a sorted set, you can get the record that ranked at 10M, without going through the first 10M records, this is a feat that can not be emulated by current SQL databases.

As the name "sorted set" implies, it's a data structure for sorting. Trees, balanced binary trees, are popular data structures for storing sorted data. Among various data structures, the author found the AVL tree particularly simple and easy to code, which will be used in this book to implement sorted set. The real Redis project uses skiplist which is also considered easy to code.

The idea of the AVL tree is to restrict the height difference between the left subtree and the right subtree. The height difference between subtrees is restricted to be at most one, never reaching two. When inserting/removing nodes from an AVL tree, the height difference can temporarily reach two, which is then fixed by the node rotations. The rotation operation is the basis of balanced binary trees, which is also used by other balanced trees like the RB tree. After the rotation, a node with a subtree height difference of two is reduced back to be at most one.

10.2 The AVL Tree Definition

Let's start with the tree node:

```
struct AVLNode {
    uint32_t depth = 0;
    uint32_t cnt = 0;
    AVLNode *left = NULL;
    AVLNode *right = NULL;
    AVLNode *parent = NULL;
};

static void avl_init(AVLNode *node) {
    node->depth = 1;
    node->cnt = 1;
    node->left = node->right = node->parent = NULL;
}
```

This is a regular binary tree node with extra fields. The `depth` field is the height of the tree. The `cnt` field is the size of the tree, this field is not specific to the AVL tree, it is used to implement the rank-based query, which will be explained in the next chapter.

Listing some helper functions:

```
static uint32_t avl_depth(AVLNode *node) {
    return node ? node->depth : 0;
}

static uint32_t avl_cnt(AVLNode *node) {
    return node ? node->cnt : 0;
}

static uint32_t max(uint32_t lhs, uint32_t rhs) {
    return lhs < rhs ? rhs : lhs;
}

// maintaining the depth and cnt field
static void avl_update(AVLNode *node) {
    node->depth = 1 + max(avl_depth(node->left), avl_depth(node->right));
    node->cnt = 1 + avl_cnt(node->left) + avl_cnt(node->right);
}
```

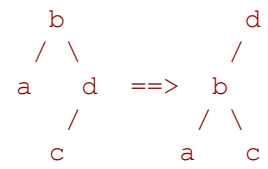
10.3 Balancing By Rotation

The node rotation code:

```
static AVLNode *rot_left(AVLNode *node) {
    AVLNode *new_node = node->right;
    if (new_node->left) {
        new_node->left->parent = node;
    }
    node->right = new_node->left;
    new_node->left = node;
    new_node->parent = node->parent;
    node->parent = new_node;
    avl_update(node);
    avl_update(new_node);
    return new_node;
}

static AVLNode *rot_right(AVLNode *node) {
    // a mirror of the rot_left()
    // code omited...
}
```

A visualization of the `rot_left` operation:



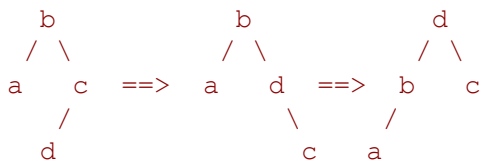
The `avl_fix_left` and `avl_fix_right` are functions for fixing excess height difference:

```
// the left subtree is too deep
static AVLNode *avl_fix_left(AVLNode *root) {
    if (avl_depth(root->left->left) < avl_depth(root->left->right)) {
        root->left = rot_left(root->left);
    }
    return rot_right(root);
}

// the right subtree is too deep
```

```
static AVLNode *avl_fix_right(AVLNode *root) {
    if (avl_depth(root->right->right) < avl_depth(root->right->left)) {
        root->right = rot_right(root->right);
    }
    return rot_left(root);
}
```

If the right subtree is too deep, a left rotation will fix it. Before the left rotation, we may need a right rotation on the right subtree to ensure the right subtree is leaning in the correct direction. Here is the visualization:



The `avl_fix` function fixes everything after an insertion/deletion operation. It goes from the initially affected node to the root node. Since the rotation may change the root of the tree, the root node is returned. This is the core of our AVL tree implementation.

```
// fix imbalanced nodes and maintain invariants until the root is reached
static AVLNode *avl_fix(AVLNode *node) {
    while (true) {
        avl_update(node);
        uint32_t l = avl_depth(node->left);
        uint32_t r = avl_depth(node->right);
        AVLNode **from = NULL;
        if (node->parent) {
            from = (node->parent->left == node)
                ? &node->parent->left : &node->parent->right;
        }
        if (l == r + 2) {
            node = avl_fix_left(node);
        } else if (l + 2 == r) {
            node = avl_fix_right(node);
        }
        if (!from) {
            return node;
        }
        *from = node;
        node = node->parent;
```

```
    }
}
```

10.4 Insertion and Deletion

Insertion for binary trees is easy, just walk down from the root until you find an empty subtree and place the new node here, then call up `avl_fix` for maintenance.

Deletion is more complicated. If the target node has no subtree, just remove it straight, if it has one subtree, replace the node with that subtree. The problem arises when the node has both subtrees, we can't remove it straight, instead, we remove its sibling in the right subtree, and swap it with the detached sibling. Here is the function for removing a node:

```
// detach a node and returns the new root of the tree
static AVLNode *avl_del(AVLNode *node) {
    if (node->right == NULL) {
        // no right subtree, replace the node with the left subtree
        // link the left subtree to the parent
        AVLNode *parent = node->parent;
        if (node->left) {
            node->left->parent = parent;
        }
        if (parent) {
            // attach the left subtree to the parent
            (parent->left == node ? parent->left : parent->right) = node->left;
            return avl_fix(parent);
        } else {
            // removing root?
            return node->left;
        }
    } else {
        // swap the node with its next sibling
        AVLNode *victim = node->right;
        while (victim->left) {
            victim = victim->left;
        }
        AVLNode *root = avl_del(victim);

        *victim = *node;
        if (victim->left) {
```

```
            victim->left->parent = victim;
        }
        if (victim->right) {
            victim->right->parent = victim;
        }
        AVLNode *parent = node->parent;
        if (parent) {
            (parent->left == node ? parent->left : parent->right) = victim;
            return root;
        } else {
            // removing root?
            return victim;
        }
    }
}
```

This is the generic function for removing nodes from a binary tree, with the AVL-tree-specific `avl_fix`.

Readers with experiences with the RB tree may notice how small and simple the AVL tree implementation is. The maintenance code for RB tree node deletion is significantly more complicated than the insertion; while the AVL tree uses the same function `avl_fix` for both insertion and deletion, this symmetry greatly reduces the efforts required to code an AVL tree.

The AVL tree is significantly more complicated than the hashtable we coded before. Thus, we need to invest more time on testing. The testing code also demonstrates the usage of those AVL tree functions.

10.5 Testing Setup

Here are our testing data types. If you are not familiar with intrusive data structures, read the hashtable chapter.

```
struct Data {
    AVLNode node;
    uint32_t val = 0;
};

struct Container {
```

```
    AVLNode *root = NULL;
};
```

The insertion code:

```
static void add(Container &c, uint32_t val) {
    Data *data = new Data();
    avl_init(&data->node);
    data->val = val;

    if (!c.root) {
        c.root = &data->node;
        return;
    }

    AVLNode *cur = c.root;
    while (true) {
        AVLNode **from =
            (val < container_of(cur, Data, node)->val)
            ? &cur->left : &cur->right;
        if (!*from) {
            *from = &data->node;
            data->node.parent = cur;
            c.root = avl_fix(&data->node);
            break;
        }
        cur = *from;
    }
}
```

This demonstrates the deletion of nodes:

```
static bool del(Container &c, uint32_t val) {
    AVLNode *cur = c.root;
    while (cur) {
        uint32_t node_val = container_of(cur, Data, node)->val;
        if (val == node_val) {
            break;
        }
        cur = val < node_val ? cur->left : cur->right;
    }
```

```
    if (!cur) {
        return false;
    }

    c.root = avl_del(cur);
    delete container_of(cur, Data, node);
    return true;
}
```

Here is the function for verifying the correctness of the tree structure:

```
static void avl_verify(AVLNode *parent, AVLNode *node) {
    if (!node) {
        return;
    }

    assert(node->parent == parent);
    avl_verify(node, node->left);
    avl_verify(node, node->right);

    assert(node->cnt == 1 + avl_cnt(node->left) + avl_cnt(node->right));

    uint32_t l = avl_depth(node->left);
    uint32_t r = avl_depth(node->right);
    assert(l == r || l + 1 == r || l == r + 1);
    assert(node->depth == 1 + max(l, r));

    uint32_t val = container_of(node, Data, node)->val;
    if (node->left) {
        assert(node->left->parent == node);
        assert(container_of(node->left, Data, node)->val <= val);
    }
    if (node->right) {
        assert(node->right->parent == node);
        assert(container_of(node->right, Data, node)->val >= val);
    }
}
```

Code for comparing the contents of AVL tree with the expected data:

```
static void extract(AVLNode *node, std::multiset<uint32_t> &extracted) {
    if (!node) {
        return;
    }
    extract(node->left, extracted);
    extracted.insert(container_of(node, Data, node)->val);
    extract(node->right, extracted);
}

static void container_verify(
    Container &c, const std::multiset<uint32_t> &ref)
{
    avl_verify(NULL, c.root);
    assert(avl_cnt(c.root) == ref.size());
    std::multiset<uint32_t> extracted;
    extract(c.root, extracted);
    assert(extracted == ref);
}
```

Don't forget to clean up after tests:

```
static void dispose(Container &c) {
    while (c.root) {
        AVLNode *node = c.root;
        c.root = avl_del(c.root);
        delete container_of(node, Data, node);
    }
}
```

10.6 Test Cases

Our test cases start with simple things:

```
    Container c;

    // some quick tests
    container_verify(c, {});
    add(c, 123);
    container_verify(c, {123});
    assert(!del(c, 124));
```

```
    assert(del(c, 123));
    container_verify(c, {});

    // sequential insertion
    std::multiset<uint32_t> ref;
    for (uint32_t i = 0; i < 1000; i += 3) {
        add(c, i);
        ref.insert(i);
        container_verify(c, ref);
    }
```

Then we throw in random operations:

```
    // random insertion
    for (uint32_t i = 0; i < 100; i++) {
        uint32_t val = (uint32_t)rand() % 1000;
        add(c, val);
        ref.insert(val);
        container_verify(c, ref);
    }

    // random deletion
    for (uint32_t i = 0; i < 200; i++) {
        uint32_t val = (uint32_t)rand() % 1000;
        auto it = ref.find(val);
        if (it == ref.end()) {
            assert(!del(c, val));
        } else {
            assert(del(c, val));
            ref.erase(it);
        }
        container_verify(c, ref);
    }
```

Some more targeted tests. Given a tree of a certain size, perform insertion/deletion at every possible position.

```
static void test_insert(uint32_t sz) {
    for (uint32_t val = 0; val < sz; ++val) {
        Container c;
        std::multiset<uint32_t> ref;
```

```
        for (uint32_t i = 0; i < sz; ++i) {
            if (i == val) {
                continue;
            }
            add(c, i);
            ref.insert(i);
        }
        container_verify(c, ref);

        add(c, val);
        ref.insert(val);
        container_verify(c, ref);
        dispose(c);
    }
}

static void test_remove(uint32_t sz) {
    for (uint32_t val = 0; val < sz; ++val) {
        Container c;
        std::multiset<uint32_t> ref;
        for (uint32_t i = 0; i < sz; ++i) {
            add(c, i);
            ref.insert(i);
        }
        container_verify(c, ref);

        assert(del(c, val));
        ref.erase(val);
        container_verify(c, ref);
        dispose(c);
    }
}


    // insertion/deletion at various positions
    for (uint32_t i = 0; i < 200; ++i) {
        test_insert(i);
        test_remove(i);
    }
```

With the help of those test cases, the author did found and fixed a couple of mistakes while writing this chapter.

Exercises:

1. While there is not much code for our AVL tree, this AVL tree implementation is probably not a very efficient one. Our code contains some reductant pointer updates, which might be a source of optimization. Also, we don't need to store the height value for balancing, it is possible to store the height difference instead. Research and explore efficient AVL tree implementations.
2. Can you create more test cases? The test cases presented in this chapter are unlikely to be sufficient.

Source code:

> - avl.cpp
> - test_avl.cpp