CHAPTER

10. Balanced Binary Tree

10.1 Sorted set for sorting

A sorted set is a collection of sorted (score, name) pairs.

```
std::set<std::pair<double, std::string>> by_score;
```

The obvious use case is ranking, e.g., retrieving the top-k items from a high-score board. But it's generic enough for anything that needs sorting.

Tuple comparison

The (score, name) pairs are ordered by 2 fields, similar to a multi-column DB index.

$$(s_1, n_1) < (s_2, n_2) \implies s_1 < s_2 \;||\; (s_1 = s_2 \;\&\&\; n_1 < n_2)$$

A use case is to use the same score for all pairs, so they are ordered solely by name. The "name" is a byte string, but it's possible to encode anything as bytes while preserving the sort order, so the sorted set can sort anything.

Multi-indexed data

The score is a 64-bit float, it can be non-unique. The name is unique, you can delete a pair by name or update the score by name. So the collection is indexed in 2 ways:

```
struct SortedSet {
    std::set<std::pair<double, std::string>> by_score;
    std::unordered_map<std::string, double>  by_name;
};
```

Multiple indexes on the same data are common in databases.

```
CREATE TABLE sorted_set (
    `name` string,
```

```
    `score` double,
    PRIMARY KEY `name`,               -- primary index
    INDEX by_score (`score`, `name`)  -- secondary index
);
```

Sorting indexes

Indexing by score requires a sorting data structure. In databases, sorting indexes are either B+trees or LSM-trees, both of which are non-trivial. For in-memory data stores, there are more choices, including various **binary trees**.

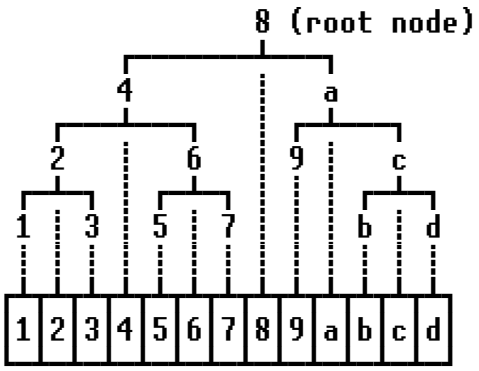
A lot of database use involves sorting data, that's why database indexes are mostly trees rather than hashtables, even though trees are slower. An in-memory data store can have both indexes, like the Redis sorted set.

Sorting data structures in STL are based on the **RB tree** (`std::map` and `std::set`). Real Redis uses **Skip list** for sorted set. We will code a much easier **AVL tree** instead.

10.2 Tree data structures

Sorting data by divide and conquer

Almost all sorting data structures are trees, because a tree node divides the data into non-overlapping subtrees.

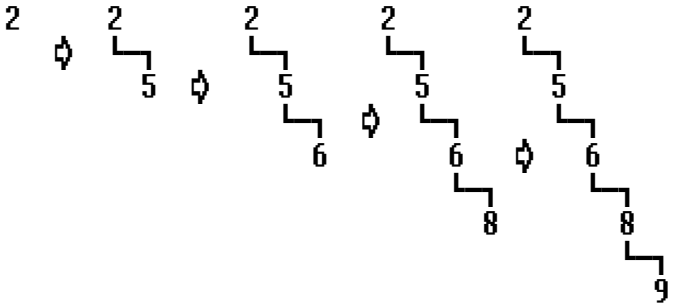


The root node represents the range $(-\infty, +\infty)$. The range is reduced by following a child node. To search a key in the tree, keep the key in the range while walking down.

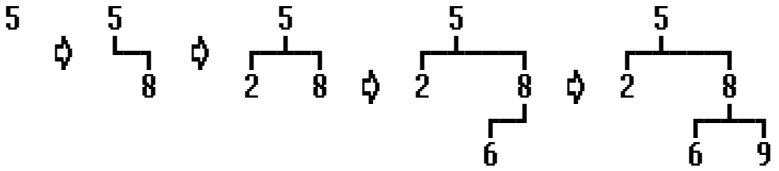
Unbalanced binary trees

To insert a key into the tree, first do a search until reaching a leaf node, then place the new node at one of its empty child pointers. The insert order affects the tree shape.

Insert order: 2, 5, 6, 8, 9:



Same data in a different order: 5, 8, 2, 6, 9:



In the first example, nodes are inserted in sorted order, resulting in a deep tree that resembles a linked list. This is the worst case with a search complexity of $O(N)$. While on average, the tree depth is $O(\log N)$, assuming random insert order.

Height-balanced trees

You do not control what's inserted into the tree, but you can **reshape** the tree after an insert to avoid making a very deep tree. The goal is to limit the **maximum tree depth** to $O(\log N)$ in the worst case. This is called a height-balanced tree, and is achieved by preserving some **invariants**:

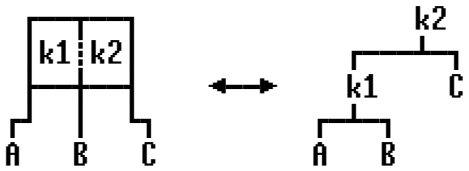
- B-tree: N-ary tree with the same height for all leaf nodes.
- RB tree: The maximum leaf height is within 2x of the minimum leaf height.
- AVL tree: The heights of the 2 subtrees can only differ by 1 at most.

Some trees don't guarantee the worst case, but are often good enough:

- Treap: Randomly rotate nodes to reshape the tree. Better than unbalanced trees, but doesn't reach the worst case goal.
- Skip list: Equivalent to n-ary trees, uses randomness like treap. Real Redis uses this. I haven't found a way to describe it in a few sentences.

B-tree -> 2-3-tree -> RB tree

B-tree achieves uniform heights because a node can store a variable number of keys, which is not possible for binary trees. However, it's possible to represent a 2-key B-tree node as 2 binary nodes.



An n-ary tree where a node can store either 1 or 2 keys is called a 2-3-tree. Replacing the 2-key nodes with 2 binary nodes results in a RB tree.

Comparisons of trees

All sorting data structures are equal in their capacities: query, modify, iterate. We'll choose the easiest option that achieves the worst case goal: AVL tree.

Tree	Worst case	Branch	Random	Difficulty
AVL tree	$O(\log N)$	2	No	Medium
RB tree	$O(\log N)$	2	No	High
B-tree	$O(\log N)$	n	No	High
Skip list	$O(N)$	n	Yes	Medium
Treap	$O(N)$	2	Yes	Low

10.3 Basic binary tree operations

A balanced binary tree is just an unbalanced binary tree with extra code to restore balance after insert and delete. Search and iterate are the same for all binary trees.

```
struct Node { Node *left; Node *right; };
```

Search and insert

Starting from the root, the target is either in the left or right subtree, if not hit.

```
// Pseudo code
Node **tree_search(Node **from, int32_t (*cmp)(Node *, void *), void *key) {
    Node *node = *from
    if (!node) {
        return from;    // NULL node
    }
    int32_t r = cmp(node, key);
    if (r < 0) {        // node < key
        return tree_search(&node->right, cmp, key);
```

```
    } else if (r > 0) { // node > key
        return tree_search(&node->left, cmp, key);
    } else {            // node == key
        return from;
    }
}
```

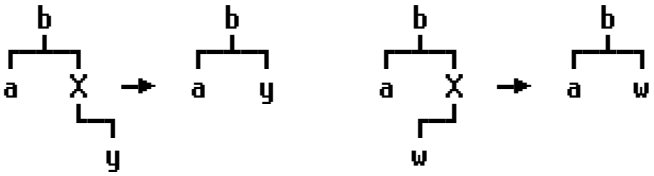
This uses the *incoming pointer* trick from the hashtable chapter. It returns the address of the pointer to the target node, so if the search ends at a NULL pointer, the address can be used for an insert.

```
struct Data { Node node; T key; };  // intrusive

Data *new_data = make_some_data();
Node **child = tree_search(&root, cmp, &new_data->key);
if (!*child) {   // not found
    *child = &new_data->node;   // insert
} // else: found a dup
```

Detach a node

The easy case: A node with 0 or 1 children. Just replace the subtree with the child.



The hard case: A node with both children.

1. Detach the successor node.
2. Swap with the successor node.

This works because:

- The successor is the leftmost node in the right subtree, it can be detached by the easy case because its left child is empty.
- Replacing a node with its successor doesn't change the data order.

Trees are symmetric, so this works if you use predecessor instead of successor.

```
// Pseudo code. Return the new tree without the removed node.
Node *node_detach(Node *node) {
    // easy case: one of the subtree is empty, return the other one
    if (!node->right) {
        return node->left;
    }
    // find the successor
    Node **to_victim = &node->right;    // incoming pointer
    while ((*to_victim)->left) {
        to_victim = &(*to_victim)->left;
    }
    Node *victim = *to_victim;
    // detach the successor by the easy case
    *to_victim = victim->right;
    // swap with the successor
    *victim = *node;
    return victim;
}
```

Search and delete

The pseudo code doesn't use the parent pointer, so `node_detach()` can only be used recursively, where the caller links the parent node to the new subtree.

```
Node *tree_delete(Node *node, int32_t (*cmp)(Node *, void *), void *key) {
    if (!node) {
        return NULL;    // not found
    }
    int32_t r = cmp(node, key);
    if (r < 0) {        // node < key
        node->right = tree_delete(node->right, cmp, key);
        return node;
    } else if (r > 0) { // node > key
        node->left = tree_delete(node->left, cmp, key);
        return node;
    } else {            // node == key
        return node_detach(node);
    }
}
```

```
Node *root = make_a_tree();
root = tree_delete(root, cmp, key);
```

Iterate in sorted order

The successor node is the leftmost node in the right subtree, except if the right subtree is empty, you must backtrack. That's why a tree node usually includes a parent pointer. Without parent pointers, the iterator must store the entire path of the nodes.

Parent pointers

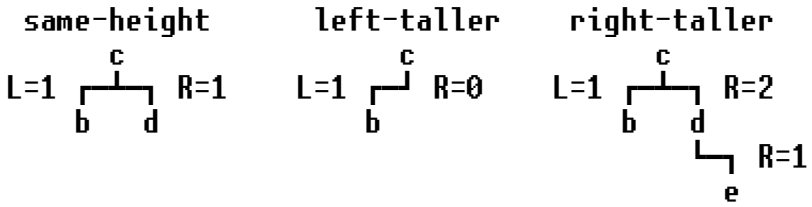
With parent pointers, a node can be deleted with just a pointer to the node. This is handy because you may get the reference to a node via another index (e.g., the multi-indexed sorted set).

The code above assumes no parent pointers, so we won't use them.

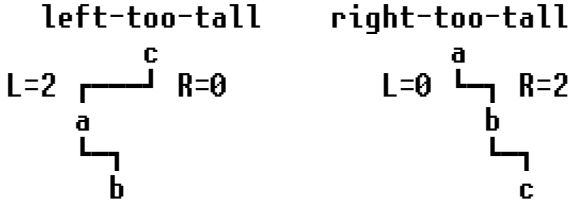
10.4 How do AVL trees work?

Invariant: Height difference by 1 at most

The height of a subtree is the maximum distance from the root to the leaves (0 for empty trees). For any node in an AVL tree, the height of its 2 subtrees can differ, but only by 1. These are valid AVL trees:



These are not valid AVL trees because the height difference is 2:

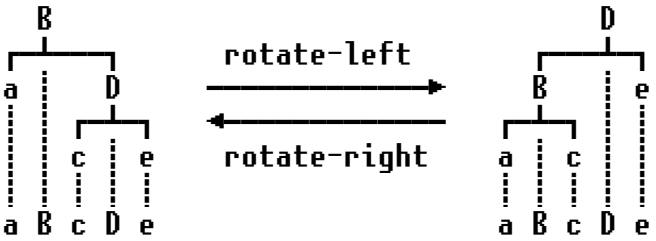


Break the invariant

Start from a valid AVL tree, then do a binary tree insert or delete, the height of some subtrees may increase or decrease by 1, possibly leading to a height difference of 2. The next step is to reshape the bad subtree.

Binary tree rotation keeps order

Rotations change the shape of a subtree while keeping its order.



Nodes B and D swap their parent-child relationship. The links to their 3 (possibly empty) subtrees a, c, e are repositioned. The order of the data is unchanged, only 2 links are changed:

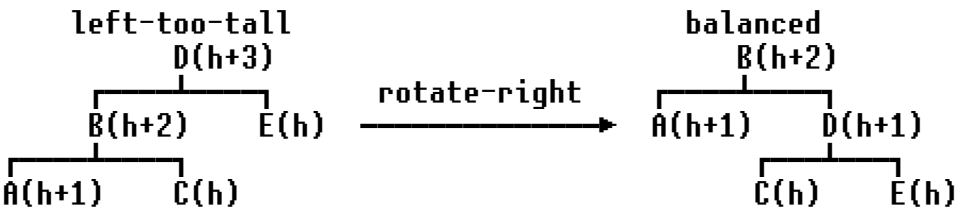
- The link between B and D is swapped, one of them being the parent.
- The inner subtree c is attached to either B or D.
- The links to a, e remain unchanged.

```
Node *rot_left(Node *node) {
    Node *new_node = node->right;
    Node *inner = new_node->left;
    node->right = inner;    // node -> inner
    new_node->left = node;  // new_node -> node
    return new_node;
}
```

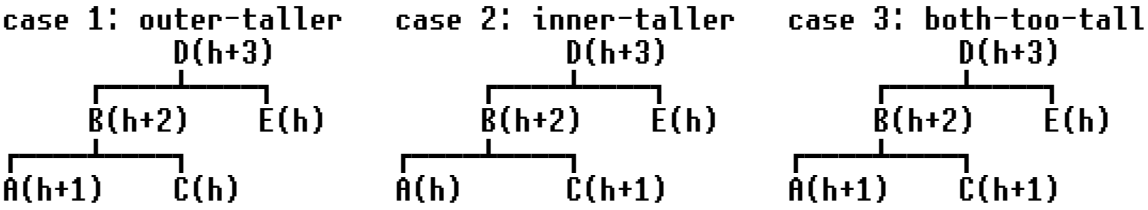
Binary tree rotation changes heights

Binary tree rotation is used in AVL tree, RB tree, Treap. It can change the tree height while keeping the data order. Let's fix the scenario where the left subtree B is taller than the right subtree E by 2.

Transformation 1:



Assuming B's subtrees A, C are valid AVL trees, we have 3 cases:

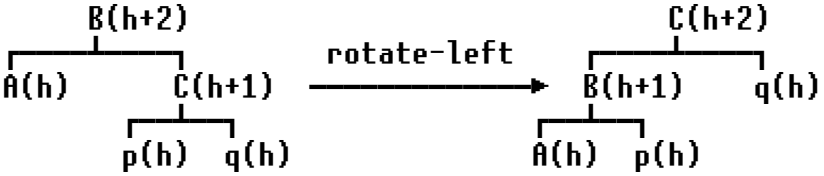


Transformation 1 fixed case 1 where the outer subtree A is taller than the inner subtree C.

Case 3, where both A and C are $h + 1$, is impossible, because a single insert or delete affects only one of the subtrees; if B is too tall, it's caused by either A or C, not both.

Case 2 can be transformed into case 1 by rotating B.

Transformation 2:



If the right subtree is taller, a left rotation makes the left subtree the taller one.

C's subtree p, q doesn't have to be h; one of them can be $h - 1$. However, the effect is still the same, and the following **transformation 1** still works. This visualization is left to the reader as an exercise.

Conclusion: There are 2 cases where the subtree heights differ by 2, which are fixed by 1 or 2 rotations.

10.5 AVL tree in action

Based on the unbalanced binary tree, we'll add the AVL code to fix imbalances.

Step 1: Auxiliary data in the tree node

An instrusive tree node doesn't contain data (see the hashtable chapter), except for the auxiliary data used by the AVL tree, which is the subtree height. The height is stored in the node, no need to traverse the tree to find out.

```
struct AVLNode {
    AVLNode *parent = NULL; // added
    AVLNode *left = NULL;
```

```
    AVLNode *right = NULL;
    uint32_t height = 0;    // auxiliary data for AVL tree
};

inline void avl_init(AVLNode *node) {
    node->left = node->right = node->parent = NULL;
    node->height = 1;
}


static uint32_t avl_height(AVLNode *node) {
    return node ? node->height : 0;
}
static void avl_update(AVLNode *node) {
    node->height = 1 + max(avl_height(node->left), avl_height(node->right));
}
```

Tree updates are bottom-up, so a height change propagates from children to parents.

Side note: Store the height difference in 2 bits

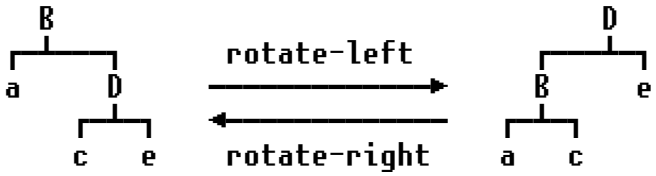
The exact height value is unnecessary because only the height difference is used. There are only 3 valid height differences so it can be stored in 2 bits.

These 2 bits can be stored in a pointer because `malloc()` are aligned to at least 8 bytes on major platforms, which means the 3 least significant bits are unused.

```
static uint8_t avl_get_height_diff(AVLNode *node) {
    uintptr_t p = (uintptr_t)node->parent;
    return p & 0b11;                    // data from LSB
}
static AVLNode *avl_get_parent(AVLNode *node) {
    uintptr_t p = (uintptr_t)node->parent;
    return (AVLNode *)(p & (~0b11));    // clear the LSB
}
```

Some RB trees uses this space optimization. We won't bother with this because the next chapter will add more auxiliary data, making this pointless.

Step 2: Rotations



Rotation is simple without parent pointers:

```
Node *rot_left(Node *node) {
    Node *new_node = node->right;
    Node *inner = new_node->left;
    node->right = inner;    // node -> inner
    new_node->left = node;  // new_node -> node
    return new_node;
}
```

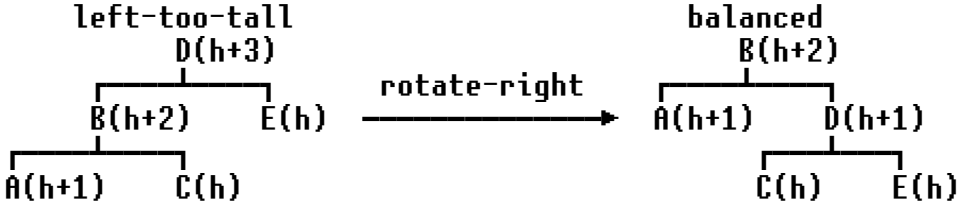
Add a few more lines for parent pointers and auxiliary data:

```
static AVLNode *rot_left(AVLNode *node) {
    AVLNode *parent = node->parent;
    AVLNode *new_node = node->right;
    AVLNode *inner = new_node->left;
    // node <-> inner
    node->right = inner;
    if (inner) {
        inner->parent = node;
    }
    // parent <- new_node
    new_node->parent = parent;  // NOTE: may be NULL
    // new_node <-> node
    new_node->left = node;
    node->parent = new_node;
    // auxiliary data
    avl_update(node);
    avl_update(new_node);
    return new_node;
}
```

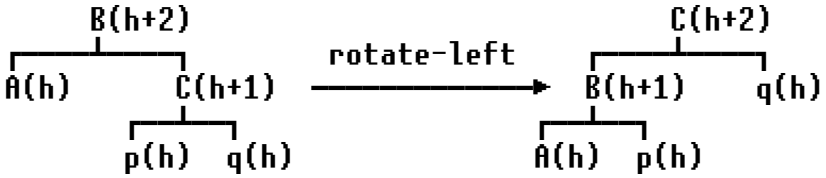
The rotated node links to the parent, but **the parent-to-child link is not updated here**. This is because the rotated node may be a root node without a parent, and only the caller knows how to link a root node, so this link is left to the caller.

Step 3: Fix the case of a height difference of 2

Transformation 1: A right rotation restores balance if the left subtree is taller by 2 and the left-left subtree is taller than the left-right subtree.



Transformation 2: A left rotation makes the left subtree taller if the right is taller.




```
static AVLNode *avl_fix_left(AVLNode *node) {   // the left is too tall
    if (avl_height(node->left->left) < avl_height(node->left->right)) {
        node->left = rot_left(node->left);      // Transformation 2
    }
    return rot_right(node);                     // Transformation 1
}
```

The result of `rot_left()` is assigned to the parent node to complete the bidirectional link. The result of `rote_right()` is left for the caller of `avl_fix_left()`.

Step 4: Fix imbalances after an insert or delete

The tree height propagates from the updated node to the root, and any imbalances are fixed during propagation. The fix may change the root node, so **the root node is returned** because the data structure doesn't store the root pointer.

```
// Called on an updated node:
// - Propagate auxiliary data.
// - Fix imbalances.
// - Return the new root node.
AVLNode *avl_fix(AVLNode *node) {
    while (true) {
        AVLNode **from = &node; // save the fixed subtree here
        AVLNode *parent = node->parent;
```

```
        if (parent) {
            // attach the fixed subtree to the parent
            from = parent->left == node ? &parent->left : &parent->right;
        }   // else: save to the local variable `node`
        // auxiliary data
        avl_update(node);
        // fix the height difference of 2
        uint32_t l = avl_height(node->left);
        uint32_t r = avl_height(node->right);
        if (l == r + 2) {
            *from = avl_fix_left(node);
        } else if (l + 2 == r) {
            *from = avl_fix_right(node);
        }
        // root node, stop
        if (!parent) {
            return *from;
        }
        // continue to the parent node because its height may be changed
        node = parent;
    }
}
```

The result of `avl_fix_*()` is assigned to the parent node to complete the link. If there's no parent, just return the new root node.

Step 5: Detach a node: The easy case

For the easy case of detaching a node with only 1 child, just replace the subtree with that child. But with parent pointers, there are many more lines:

```
// detach a node where 1 of its children is empty
static AVLNode *avl_del_easy(AVLNode *node) {
    assert(!node->left || !node->right);    // at most 1 child
    AVLNode *child = node->left ? node->left : node->right; // can be NULL
    AVLNode *parent = node->parent;
    // update the child's parent pointer
    if (child) {
        child->parent = parent; // can be NULL
    }
```

```
    // attach the child to the grandparent
    if (!parent) {
        return child;   // removing the root node
    }
    AVLNode **from = parent->left == node ? &parent->left : &parent->right;
    *from = child;
    // rebalance the updated tree
    return avl_fix(parent);
}
```

The parent of the detached node is fixed with `avl_fix()` and the root is returned.

Step 6: Detach a node: The hard case

The hard case with 2 children: just add parent pointers to `node_detach()`.

```
// detach a node and returns the new root of the tree
AVLNode *avl_del(AVLNode *node) {
    // the easy case of 0 or 1 child
    if (!node->left || !node->right) {
        return avl_del_easy(node);
    }
    // find the successor
    AVLNode *victim = node->right;
    while (victim->left) {
        victim = victim->left;
    }
    // detach the successor
    AVLNode *root = avl_del_easy(victim);
    // swap with the successor
    *victim = *node;    // left, right, parent
    if (victim->left) {
        victim->left->parent = victim;
    }
    if (victim->right) {
        victim->right->parent = victim;
    }
    // attach the successor to the parent, or update the root pointer
    AVLNode **from = &root;
    AVLNode *parent = node->parent;
```

```
    if (parent) {
        from = parent->left == node ? &parent->left : &parent->right;
    }
    *from = victim;
    return root;
}
```

Conclusion: The AVL tree API

A generic search function for an intrusive tree may be unnecessary because it's so trivial. A custom search function without compare callbacks is more flexible and performant. Our AVL tree API has only 2 functions:

- `avl_fix()`: Called on the parent node after an insert to restore balance.
- `avl_del()`: Detach a node. This implies `avl_fix()`.

Some pseudo code to show how to use them:

```
void search_and_insert(
    AVLNode **root, AVLNode *new_node, bool (*less)(AVLNode *, AVLNode *))
{
    // find the insert position
    AVLNode *parent = NULL; // place the new node as its child
    AVLNode **from = root;  // place the new node here
    for (AVLNode *node = *root; node; ) {
        from = less(new_node, node) ? &node->left : &node->right;
        parent = node;
        node = *from;
    }
    // link the new node
    *from = new_node;
    new_node->parent = parent;
    // fix the updated node
    *root = avl_fix(new_node);
}
```

```
AVLNode* search_and_delete(
    AVLNode **root, int32_t (*cmp)(AVLNode *, void *), void *key)
{
```

```
    for (AVLNode *node = *root; node; ) {
        int32_t r = cmp(node, key);
        if (r < 0) {            // node < key
            node = node->right;
        } else if (r > 0) {     // node > key
            node = node->left;
        } else {                // found
            *root = avl_del(node);
            return node;
        }
    }
    return NULL;                // not found
}
```

Tips on testing your code

- Generate test cases with code.
- Verify data structure properties and invariants.
- Compare to a reference data structure such as `std::set`.

Source code:

> - avl.cpp
> - avl.h
> - test_avl.cpp