# 1 大小平衡树

大小平衡树（Size-Balanced Tree，简称 SBT）是一种自平衡二叉搜索树，它根据其左右子树大小来维持平衡。子树大小指节点所在子树的节点总数，含子根节点。SBT 支持标准的二叉搜索树操作，如插入、删除、搜索、选择和排名，并保证时间复杂度均为 O(logn)。

## 1.1 适用场景

大小平衡树仅依赖节点大小来维持平衡，因此无需引入额外信息即可提供选择和排名操作。尽管 SBT 可以提供标准二叉搜索树的所有功能，但就性能而言，它仍然无法取代红黑树（Red-Black Tree）。然而，SBT 特别适用于高性能的选择和排序操作，例如通过索引访问元素、中值排序等。

## 1.2 特性

SBT 是一种基于二叉树的数据结构，具有以下特性：

(1) 左子树节点数不小于其两个侄子所在子树的节点数。

(2) 右子树节点数不小于其两个侄子所在子树的节点数。

(3) 左右子树均为大小平衡树。

考虑以下示例，其中 T 是 SBT 的节点，L 和 R 是其子节点，A、B、C 和 D 是满足上述 SBT 特性的子树。

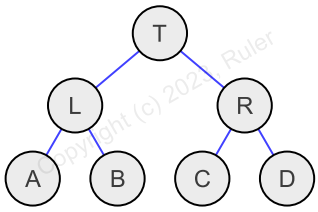


图 1.2 大小平衡树

根据 SBT 的特性，节点 T 必须满足：

(1) size(L) >= max(size(C), size(D))

(2) size(R) >= max(size(A), size(B))

## 1.3 节点

SBT 的节点包括一个父节点、两个子节点以及该节点所在子树中的节点数量。

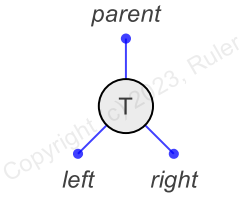

图 1.3 节点

节点定义的 C++代码如下：

```cpp
template <class T>
struct sb_tree_node
{
  using node_type            = sb_tree_node<T>;
  using node_pointer         = node_type*;
  using const_node_pointer   = const node_type*;
  using node_reference       = node_type&;
  using const_node_reference = const node_type&;

  node_pointer               parent;
  node_pointer               left;
  node_pointer               right;
  size_t                     size;
  T                          data;
};
```

## 1.4 旋转操作

如同其他自平衡二叉树，当插入或删除节点导致大小平衡树失去平衡时，需要通过旋转操作来恢复平衡。

常见的旋转操作包括左旋转和右旋转，这可以通过交换节点和子树的位置来实现。下面介绍左右旋转的操作过程。

### 1.4.1 左旋转

左旋转用于使节点 T 的右子节点 R 成为其父节点，并使其右子节点 R 的左子节点 A 成为其右子节点。此操作使原节点 T 成为其右子节点 R 的左子节点，从而维持二叉树的平衡。

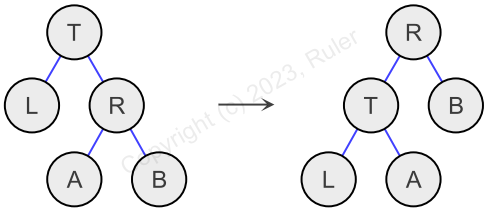


图 1.4.1 左旋转

左旋转的 C++代码如下：

```cpp
node_pointer left_rotate(node_pointer t)
{
  node_pointer r = t->right;
  t->right = r->left;
  if (r->left)
      r->left->parent = t;
  r->parent = t->parent;
```

```
    if (t == header->parent)
        header->parent = r;
    else if (t == t->parent->left)
        t->parent->left = r;
    else
        t->parent->right = r;
    r->left = t;
    r->size = t->size;
    t->parent = r;
    t->size = (t->left ? t->left->size : 0) + (t->right ? t->right->size : 0) + 1;
    return r;
}
```

### 1.4.2 右旋转

右旋转与左旋转类似，但方向相反。右旋转用于使节点 T 的左子节点 L 成为其父节点，并使其左子节点 L 的右子节点 B 成为其左子节点。此操作使原节点 T 成为其左子节点 L 的右子节点，从而维持二叉树的平衡。

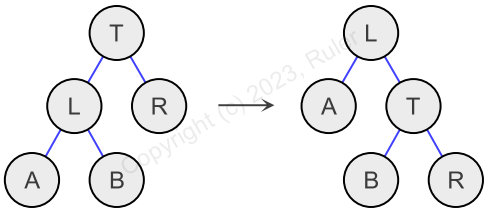


图 1.4.2 右旋转

右旋转的 C++代码如下：

```
node_pointer right_rotate(node_pointer t)
{
    node_pointer l = t->left;
    t->left = l->right;
    if (l->right)
        l->right->parent = t;
    l->parent = t->parent;
    if (t == header->parent)
        header->parent = l;
    else if (t == t->parent->right)
        t->parent->right = l;
    else
        t->parent->left = l;
    l->right = t;
    l->size = t->size;
    t->parent = l;
    t->size = (t->left ? t->left->size : 0) + (t->right ? t->right->size : 0) + 1;
    return l;
}
```

## 1.5 重新平衡

当对 SBT 进行插入或者删除操作后，可能违反 SBT 的特性，需要对以 T 为根的 SBT 进行重新平衡。再平衡的前提是 T 的子树均满足 SBT 的特性。重新平衡需要考虑以下 4 种情况：

(1) size(T.left)<size(T.right.left)

可能发生在节点 T 的右子树插入节点或者左子树删除节点后。先对 T 的右子节点 R 进行右旋转，再对 T 进行左旋转。此时，子树 A、B、D、E、F 和 L 仍然满足 SBT 的特性；而左子树 T 和右子树 R 可能违反 SBT 的特性。对于左子树 T，由于其右子树的节点减少，可能出现 size(T.right)<size(T.left.child)，需要重新平衡。对于右子树 R，由于其左子树的节点减少，可能出现 size(R.left)<size(R.right.child)，需要重新平衡。最后，对节点 C 及其祖先节点逐一进行重新平衡，直到根节点为止。

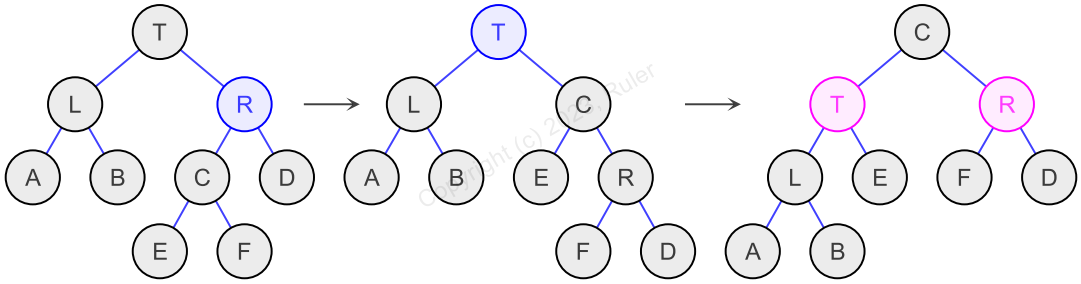


图 1.5-1 第（1）种情况

(2) size(T.left) < size(T.right.right)

可能发生在节点 T 的右子树插入节点或者左子树删除节点后。对节点 T 进行左旋转后，子树 A、B、C、D、E、F、L 仍然满足 SBT 的特性；而左子树 T 的右子树的节点减少，可能出现 size(T.right)<size(T.left.child)，需要重新平衡。最后，对节点 R 及其祖先节点逐一进行重新平衡，直到根节点为止。

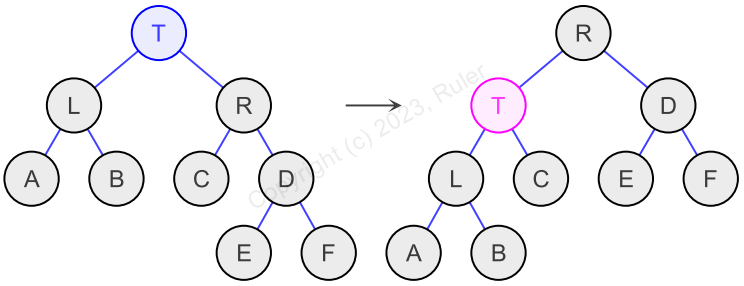


图 1.5-2 第（2）种情况

(3) size(T.right) < size(T.left.right)

可能发生在节点 T 的左子树插入节点或者右子树删除节点后。先对 T 的左子节点进行左旋转，再对 T 进行右旋转。此时，子树 A、C、D、E、F、R 仍然满足 SBT 的性质；而左子树 L 和右子树 T 可能违反 SBT 的特性。对于左子树 L，由于其右子树的节点减少，可能出现 size(L.right)<size(L.left.child)，需要重新平衡。对于右子树 T，由于其左子树的节点减少，可能出现 size(T.left)<size(T.right.child)，需要重新平衡。最后，对节点 B 及其祖先节点逐一进行重新平衡，直到根节点为止。

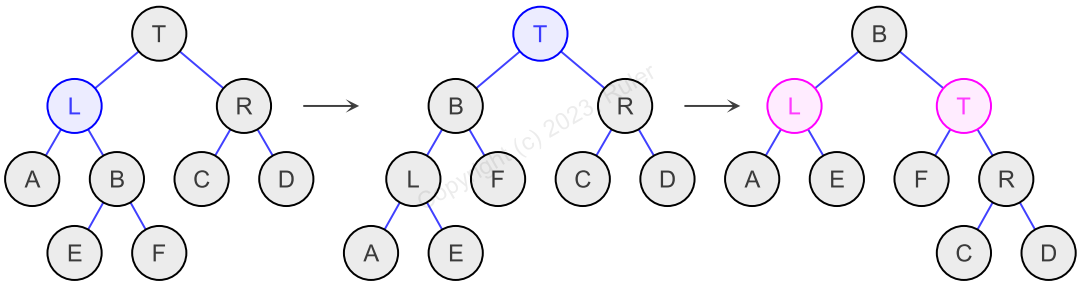


图 1.5-3 第（3）种情况

(4) size(T.right)<size(T.left.left)

可能发生在节点 T 的左子树插入节点或者右子树删除节点后。对节点 T 进行右旋转后，子树 A、B、C、D、E、F、R 仍然满足 SBT 的性质；而右子树 T 的左子树的节点减少，可能出现 size(T.left)<size(T.right.child)，需要重新平衡。最后，对节点 L 及其祖先节点逐一进行重新平衡，直到根节点为止。

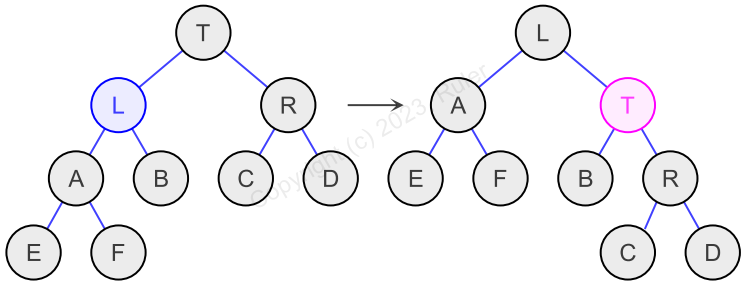


图 1.5-4 第（4）种情况

向节点 T 的子节点插入节点后进行重新平衡的 C++代码如下：

```
1.  node_pointer insert_rebalance(node_pointer t, bool flag)
2.  {
3.    if (flag)
4.    {
5.        if (t->right)
6.        {
7.            size_type left_size = t->left ? t->left->size : 0;
8.            // case 1: size(T.left) < size(T.right.left)
9.            if (t->right->left && left_size < t->right->left->size)
10.           {
11.               t->right = right_rotate(t->right);
12.               t = left_rotate(t);
13.               t->left = insert_rebalance(t->left, false);
14.               t->right = insert_rebalance(t->right, true);
15.               t = insert_rebalance(t, true);
16.           }
17.           // case 2. size(T.left) < size(T.right.right)
18.           else if (t->right->right && left_size < t->right->right->size)
19.           {
20.               t = left_rotate(t);
21.               t->left = insert_rebalance(t->left, false);
```

```
                t = insert_rebalance(t, true);
            }
        }
    }
    else
    {
        if (t->left)
        {
            size_type right_size = t->right ? t->right->size : 0;
            // case 3. size(T.right) < size(T.left.right)
            if (t->left->right && right_size < t->left->right->size)
            {
                t->left = left_rotate(t->left);
                t = right_rotate(t);
                t->left = insert_rebalance(t->left, false);
                t->right = insert_rebalance(t->right, true);
                t = insert_rebalance(t, false);
            }
            // case 4. size(T.right) < size(T.left.left)
            else if (t->left->left && right_size < t->left->left->size)
            {
                t = right_rotate(t);
                t->right = insert_rebalance(t->right, true);
                t = insert_rebalance(t, false);
            }
        }
    }
    return t;
}
```

从节点 T 的子节点删除节点后进行重新平衡的 C++代码如下：

```
node_pointer erase_rebalance(node_pointer t, bool flag)
{
    if (!flag)
    {
        if (t->right)
        {
            size_type left_size = t->left ? t->left->size : 0;
            // case 1: size(T.left) < size(T.right.left)
            if (t->right->left && left_size < t->right->left->size)
            {
                t->right = right_rotate(t->right);
                t = left_rotate(t);
                t->left = erase_rebalance(t->left, true);
                t->right = erase_rebalance(t->right, false);
                t = erase_rebalance(t, false);
            }
            // case 2. size(T.left) < size(T.right.right)
            else if (t->right->right && left_size < t->right->right->size)
```

```
            {
                t = left_rotate(t);
                t->left = erase_rebalance(t->left, true);
                t = erase_rebalance(t, false);
            }
        }
    }
    else
    {
        if (t->left)
        {
            size_type right_size = t->right ? t->right->size : 0;
            // case 3. size(T.right) < size(T.left.right)
            if (t->left->right && right_size < t->left->right->size)
            {
                t->left = left_rotate(t->left);
                t = right_rotate(t);
                t->left = erase_rebalance(t->left, true);
                t->right = erase_rebalance(t->right, false);
                t = erase_rebalance(t, true);
            }
            // case 4. size(T.right) < size(T.left.left)
            else if (t->left->left && right_size < t->left->left->size)
            {
                t = right_rotate(t);
                t->right = erase_rebalance(t->right, false);
                t = erase_rebalance(t, true);
            }
        }
    }
    return t;
}
```

## 1.6 插入操作

如果 SBT 为空，则直接插入新增节点，并作为 root 节点。否则，按照二叉搜索树的性质查找插入位置，插入新节点，其祖先节点的节点数都要加 1。当在节点 T 的左子树插入值时，可能出现上述第(3)种或者第(4)种情况；当在节点 T 的右子树插入值时，可能出现上述第(1)种或第(2)种情况。

插入操作的 C++代码如下：

```
template<class ...Args>
node_pointer insert_node(Args&&... args)
{
  // creates a new node
  node_pointer n = this->create_node(std::forward<Args>(args)...);
  n->left = nullptr;
  n->right = nullptr;
  n->size = 1;
```

```
    // if the tree is not empty
    if (header->parent)
    {
        // the initial value is root
        node_pointer t = header->parent;
        while (t)
        {
            // increases the size of nodes
            ++t->size;
            if (comp(n->data, t->data))
            {
                if (t->left)
                    t = t->left;
                else
                {
                    // inserts the node
                    n->parent = t;
                    t->left = n;
                    if (t == header->left)
                        header->left = n;
                    do
                    {
                        // rebalance after insertion
                        t = insert_rebalance(t->parent, t == t->parent->right);
                    } while (t->parent != header);
                    t = nullptr;
                }
            }
            else
            {
                if (t->right)
                    t = t->right;
                else
                {
                    // inserts the node
                    n->parent = t;
                    t->right = n;
                    if (t == header->right)
                        header->right = n;
                    do
                    {
                        // rebalance after insertion
                        t = insert_rebalance(t->parent, t == t->parent->right);
                    } while (t->parent != header);
                    t = nullptr;
                }
            }
        }
    }
```

```
58.    else
59.    {
60.        // inserts the node
61.        n->parent = header;
62.        header->parent = n;
63.        header->left = n;
64.        header->right = n;
65.    }
66.    return n;
67. }
```

## 1.7 删除操作

按照二叉搜索树的性质查找删除位置，假设待删除节点为 T，根据节点 T 的子节点数量，可分为以下两种情况：

(1) 节点 T 最多有一个子节点

在这种情况下，可以直接删除节点 T。其祖先节点的节点数减少 1。如果节点 T 具有子节点 L 或 R，则用其子节点替换节点 T。最后，重新平衡节点 T 的父节点。

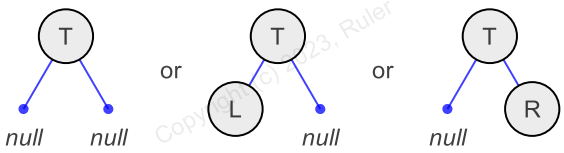


图 1.7-1 第（1）种情况

(2) 节点 T 有两个子节点

在这种情况下，不能直接删除节点 T，否则整个树将被破坏。当节点 T 的左子树 L 的节点数小于右子树 R 的节点数时，选择其右子树 R 中值最小的节点作为实际删除节点 X；否则，选择其左子树 L 中值最大的节点作为实际删除节点 X。然后交换节点 T 和节点 X 的位置，此时，与第（1）种情况完全相同。

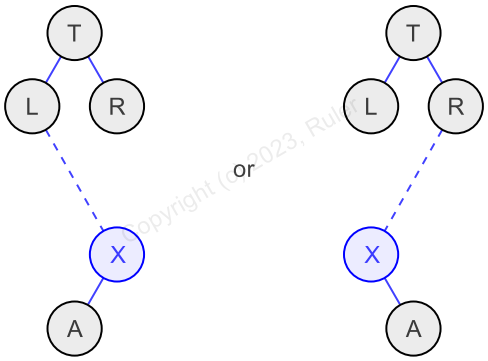


图 1.7-2 第（2）种情况

删除操作的 C++代码如下：

```
1.  void erase_node(node_pointer t)
```

```
2.  {
3.    bool flag;
4.    node_pointer x;
5.    node_pointer parent;
6.    // case 1. has one child node at most
7.    if (!t->left || !t->right)
8.    {
9.        x = t->left ? t->left : t->right;
10.       // the rebalance flag
11.       flag = (t == t->parent->right);
12.       // removes t node
13.       if (x)
14.           x->parent = t->parent;
15.       if (t == header->parent)
16.           header->parent = x;
17.       else if (t == t->parent->left)
18.           t->parent->left = x;
19.       else
20.           t->parent->right = x;
21.       if (t == header->left)
22.           header->left = x ? leftmost(x) : t->parent;
23.       if (t == header->right)
24.           header->right = x ? rightmost(x) : t->parent;
25.       // reduces the number of nodes
26.       for (node_pointer p = t->parent; p != header; p = p->parent)
27.           --p->size;
28.       if (t != header)
29.       {
30.           // rebalance after deletion
31.           node_pointer p = erase_rebalance(t->parent, flag);
32.           while (p != header)
33.               p = erase_rebalance(p->parent, p == p->parent->right);
34.       }
35.   }
36.   // case 2. has two child nodes
37.   else
38.   {
39.       if (t->left->size < t->right->size)
40.       {
41.           x = leftmost(t->right);
42.           // the rebalance flag
43.           flag = (x == x->parent->right);
44.           // reduces the number of nodes
45.           for (node_pointer p = x->parent; p != header; p = p->parent)
46.               --p->size;
47.           // replaces t node with x node and removes t node
48.           t->left->parent = x;
49.           x->left = t->left;
50.           if (x != t->right)
```

```
51.             {
52.                 x->parent->left = x->right;
53.                 if (x->right)
54.                     x->right->parent = x->parent;
55.                 t->right->parent = x;
56.                 x->right = t->right;
57.                 parent = x->parent;
58.             }
59.             else
60.                 parent = x;
61.             if (t == header->parent)
62.                 header->parent = x;
63.             else if (t == t->parent->left)
64.                 t->parent->left = x;
65.             else
66.                 t->parent->right = x;
67.             x->parent = t->parent;
68.             x->size = t->size;
69.         }
70.         else
71.         {
72.             x = rightmost(t->left);
73.             // the rebalance flag
74.             flag = (x == x->parent->right);
75.             // reduces the number of nodes
76.             for (node_pointer p = x->parent; p != header; p = p->parent)
77.                 --p->size;
78.             // replaces t node with x node and removes t node
79.             t->right->parent = x;
80.             x->right = t->right;
81.             if (x != t->left)
82.             {
83.                 x->parent->right = x->left;
84.                 if (x->left)
85.                     x->left->parent = x->parent;
86.                 t->left->parent = x;
87.                 x->left = t->left;
88.                 parent = x->parent;
89.             }
90.             else
91.                 parent = x;
92.             if (t == header->parent)
93.                 header->parent = x;
94.             else if (t == t->parent->left)
95.                 t->parent->left = x;
96.             else
97.                 t->parent->right = x;
98.             x->parent = t->parent;
99.             x->size = t->size;
```

```
100.        }
101.        // rebalance after deletion
102.        node_pointer p = erase_rebalance(parent, flag);
103.        while (p != header)
104.            p = erase_rebalance(p->parent, p == p->parent->right);
105.    }
106.    // destroy node
107.    this->destroy_node(t);
108. }
```

## 1.8 搜索操作

搜索操作与标准二叉搜索树完全相同。从根节点开始查找，如果待查找值 key 小于节点键值，再在左子树中继续查找；如果 key 大于节点键值，再在右子树中继续查找；如果 key 等于节点键值，查找完成。

搜索操作的 C++代码如下：

```
1.  node_pointer find_node(const value_type& key)
2.  {
3.    node_pointer pre = header;
4.    node_pointer cur = header->parent;
5.    while (cur)
6.    {
7.        if (!comp(cur->data, key))
8.        {
9.            pre = cur;
10.           cur = cur->left;
11.       }
12.       else
13.           cur = cur->right;
14.   }
15.   if (comp(key, pre->data))
16.       pre = header;
17.   return pre;
18. }
```

## 1.9 选择操作

选择操作为 SBT 提供随机访问功能。每个节点都记录了其所在子树的节点数，利用该信息可以查找指定排名的元素。查询排名为 k 的节点，如果左子树的节点数 size 大于 k，那么继续在左子树中查找；如果左子树的节点数 size 小于 k，那么在右子树中查找排名为 k-size-1 的节点；如果 size 等于 k，查找完成。

插入操作的 C++代码如下：

```
1.  node_pointer select_node(size_type k)
2.  {
3.    node_pointer t = header->parent;
4.    while (t)
5.    {
6.        size_type left_size = t->left ? t->left->size : 0;
```

```
        if (left_size < k)
        {
            t = t->right;
            k -= (left_size + 1);
        }
        else if (k < left_size)
            t = t->left;
        else
            return t;
    }
    return header;
}
```

### 1.10 排名操作

排名操作是选择操作的逆操作，返回给定键 key 的排名。如果当前节点的键不小于 key，那么继续在左子树中查找；否则，返回当前节点的排名加上 key 在右子树中的排名。

插入操作的 C++代码如下：

```
size_type rank_node(const value_type& key)
{
    size_type rank = 0;
    node_pointer pre = header;
    node_pointer cur = header->parent;
    while (cur)
    {
        if (!comp(cur->data, key))
        {
            pre = cur;
            cur = cur->left;
        }
        else
        {
            rank += cur->left ? cur->left->size + 1 : 1;
            cur = cur->right;
        }
    }
    if (pre == header || comp(key, pre->data))
        rank = static_cast<size_type>(-1);
    return rank;
}
```