1 Size-Balanced Tree

A Size-Balanced Tree (SBT) is a type of self-balancing binary search tree that maintains a balance between the size of its left and right subtrees. The size of a subtree is the total number of nodes in that subtree, including the root node. SBT supports the standard binary search tree operations such as insertion, deletion, searching, selection, and ranking in O(log n) time.

1.1 Applicability

A Size-Balanced Tree only depends on the node size to maintain balance, so it can provide selection and ranking operations without introducing additional information. Although SBT can provide all the functions of a standard binary search tree, it still cannot replace Red-Black Tree (RBT) from a performance perspective. However, SBT is particularly suitable for high-performance selection and sorting operations, such as accessing elements by index, median filtering, and so on.

1.2 Properties

A Size-Balanced Tree is a data structure based on binary tree and has the following properties:

(1) The size of the left child node is not less than the sizes of its two nephew nodes.

(2) The size of the right child node is not less than the sizes of its two nephew nodes.

(3) Both left and right subtrees are Size-Balanced Trees.

Consider the following example where T is the node of SBT, L and R are its child nodes, A, B, C, and D are subtrees that also satisfy the above properties of SBT.

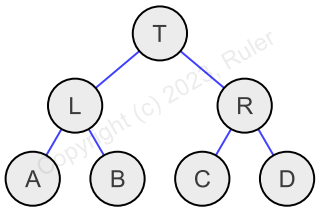

Figure1.2 Size-Balanced Tree

According to the properties of SBT, the node T must satisfy:

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

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

1.3 Node

The node of SBT include a parent node, two child nodes, and the size of nodes in the subtree where the node is located.

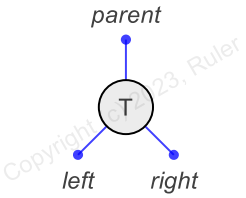

Figure1.3 Node

The C++code for node definition is as follows:

```
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 Rotations

Like other self-balancing binary search trees, rotation operations are necessary to restore balance when inserting or deleting nodes causes the Size-Balanced Tree to become unbalanced.

Common rotation operations include left rotation and right rotation, which can be achieved by exchanging the position of nodes and subtrees. The following describes the operation process of left and right rotation.

1.4.1 Left rotation

The left rotation is used to make the right child node R of node T become its parent node and make the left child node A of its right child node R become its right child node. This operation makes the original node T become the left child node of its right child node R, thereby maintaining the balance of the binary tree.

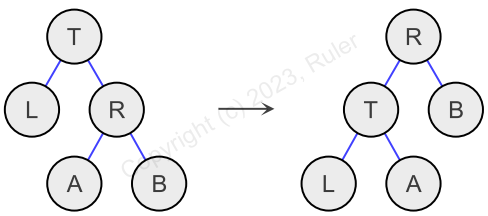

Figure 1.4.1 Left rotation

The C++ code for left rotation is as follows:

```
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 Right rotation

The right rotation is similar to the left rotation but in the opposite direction. The right rotation is used to make the left child node L of node T become its parent node and make the right child node B of its left child node L become its left child node. This operation makes the original node T become the right child node of its left child node L, thereby maintaining the balance of the binary tree.

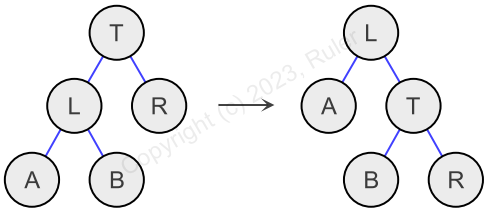

Figure 1.4.2 Right rotation

The C++ code for right rotation is as follows:

```
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 Rebalancing

After insertion or deletion in SBT, the properties of SBT may be violated, and it is necessary to rebalance the SBT rooted at node T. The prerequisite is that the child nodes of T are already SBTs themselves. There are four cases to consider when rebalancing:

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

It may occur when inserting a node into the right child of T or deleting a node from the left child of T. First, perform a right rotation on the right child node of T, and then perform a left rotation on T. Now, the subtrees A, B, D, E, F, and L still satisfy the properties of SBT, while the left subtree T and the right subtree R may violate the properties of SBT. For the left subtree T, due to the decrease in nodes of its right subtree, it may occur that size(T.right) < size(T.left.child), which requires rebalancing. For the right subtree R, due to the decrease in nodes of its left subtree, it may occur that size(R.left) < size(R.right.child), which requires rebalancing. Finally, rebalance node C and its ancestor nodes one by one until reaching the root node.

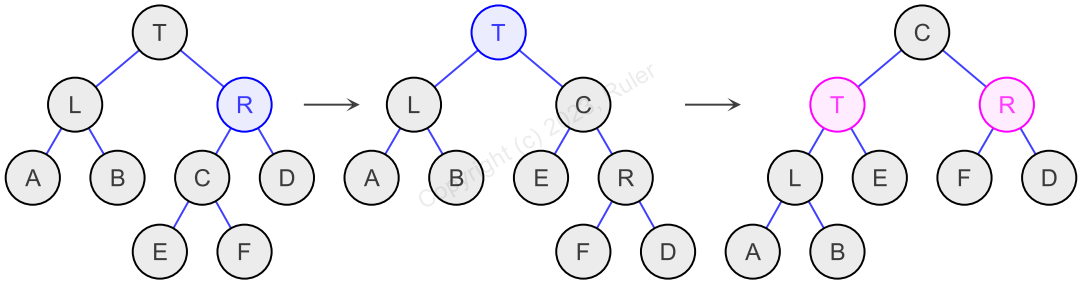

Figure 1.5-1 Case 1

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

It may occur when inserting a node into the right child of T or deleting a node from the left child of T. After performing a left rotation on T, the subtree A, B, C, D, E, F, and L still satisfy the properties of SBT. However, due to the decrease in nodes of the right subtree of T, it may occur that size(T.right) < size(T.left.child), which requires rebalancing. Finally, rebalance node R and its ancestor nodes one by one until reaching the root node.

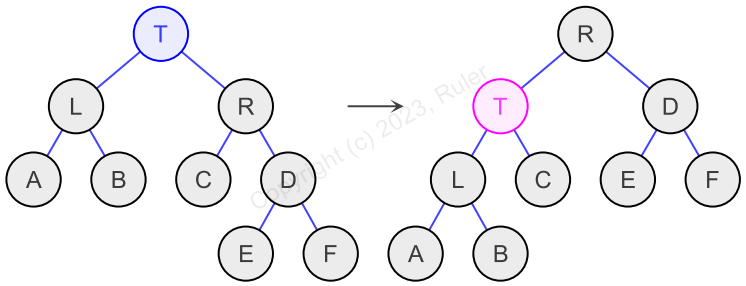

Figure 1.5-2 Case 2

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

It may occur when inserting a node into the left child of T or deleting a node from the right child of T. First, perform a left rotation on the left child node of T, and then perform a right rotation on T. Now, the subtrees A, C, D, E, F, and R still satisfy the properties of SBT, while the left subtree L and the right subtree T may violate the properties of SBT. For the left subtree L, due to the decrease in nodes of its right subtree, it may occur that size(L.right) < size(L.left.child), which requires rebalancing. For the right subtree T, due to the decrease in nodes of its left subtree, it may occur that size(T.left) < size(T.right.child), which requires rebalancing. Finally, rebalance node B and its ancestor nodes one by one until reaching the root node.

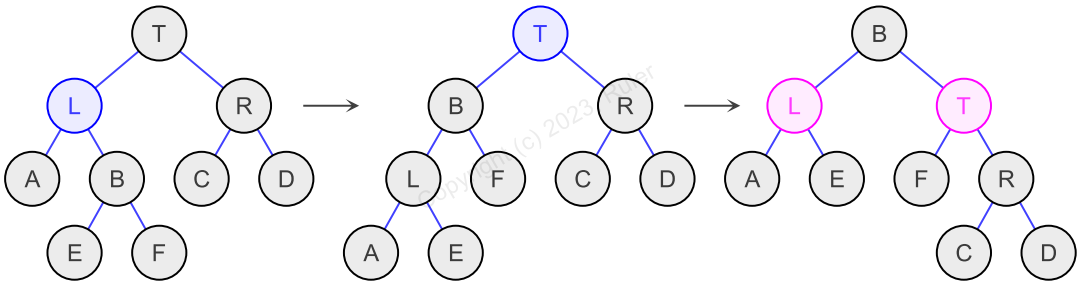

Figure 1.5-3 Case 3

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

It may occur when inserting a node into the left child of T or deleting a node from the right child of T. After performing a right rotation on T, the subtrees A, B, C, D, E, F, and R still satisfy the properties of SBT. However, due to the decrease in nodes of the left subtree of T, it may occur that size(T.left) < size(T.right.child), which requires rebalancing. Finally, rebalance node L and its ancestor nodes one by one until reaching the root node.

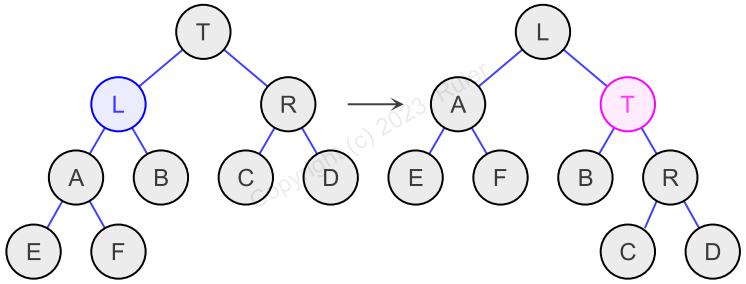

Figure 1.5-4 Case 4

The C++ code for rebalancing after inserting a node into the child node of T is as follows:

```
node_pointer insert_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 = insert_rebalance(t->left, false);
              t->right = insert_rebalance(t->right, true);
              t = insert_rebalance(t, true);
          }
          // 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 = 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;
}
```

The C++ code for rebalancing after deleting a node from the child node of T is as follows:

```
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);
```

```
35.                 t = right_rotate(t);
36.                 t->left = erase_rebalance(t->left, true);
37.                 t->right = erase_rebalance(t->right, false);
38.                 t = erase_rebalance(t, true);
39.             }
40.             // case 4. size(T.right) < size(T.left.left)
41.             else if (t->left->left && right_size < t->left->left->size)
42.             {
43.                 t = right_rotate(t);
44.                 t->right = erase_rebalance(t->right, false);
45.                 t = erase_rebalance(t, true);
46.             }
47.         }
48.     }
49.     return t;
50. }
```

1.6 Insertion

If the SBT is empty, directly add the node as the root node. Otherwise, find the insertion position according to the properties of the binary search tree and insert the new node. The node counts of its ancestor nodes should all be incremented by 1. When inserting a value in the left subtree of node T, the above-mentioned case (3) or case (4) may occur. When inserting a value in the right subtree of node T, the above-mentioned case (1) or case (2) may occur.

The C++ code for the insertion operation is as follows:

```
1.  template<class ...Args>
2.  node_pointer insert_node(Args&&... args)
3.  {
4.    // creates a new node
5.    node_pointer n = this->create_node(std::forward<Args>(args)...);
6.    n->left = nullptr;
7.    n->right = nullptr;
8.    n->size = 1;
9.    // if the tree is not empty
10.   if (header->parent)
11.   {
12.       // the initial value is root
13.       node_pointer t = header->parent;
14.       while (t)
15.       {
16.           // increases the size of nodes
17.           ++t->size;
18.           if (comp(n->data, t->data))
19.           {
20.               if (t->left)
21.                   t = t->left;
22.               else
```

```
23.                 {
24.                     // inserts the node
25.                     n->parent = t;
26.                     t->left = n;
27.                     if (t == header->left)
28.                         header->left = n;
29.                     do
30.                     {
31.                         // rebalance after insertion
32.                         t = insert_rebalance(t->parent, t == t->parent->right);
33.                     } while (t->parent != header);
34.                     t = nullptr;
35.                 }
36.             }
37.             else
38.             {
39.                 if (t->right)
40.                     t = t->right;
41.                 else
42.                 {
43.                     // inserts the node
44.                     n->parent = t;
45.                     t->right = n;
46.                     if (t == header->right)
47.                         header->right = n;
48.                     do
49.                     {
50.                         // rebalance after insertion
51.                         t = insert_rebalance(t->parent, t == t->parent->right);
52.                     } while (t->parent != header);
53.                     t = nullptr;
54.                 }
55.             }
56.         }
57.     }
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 Deletion

According to the properties of the binary search tree, find the deletion position. Assuming the

node to be deleted is T, it can be divided into the following two cases based on the number of child nodes of node T:

(1) Node T has at most one child node.

In this case, the node T can be deleted directly. The number of nodes of its ancestor nodes should be reduced by 1. If node T has a child node L or R, replace node T with its child node. Finally, rebalance the parent node of node T.

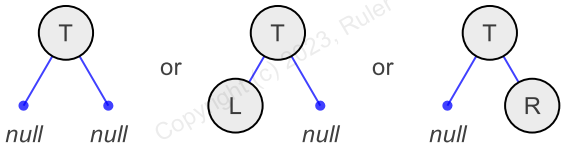

Figure 1.7-1 Case 1

(2) Node T has two child nodes.

In this case, the node T cannot be deleted directly, otherwise the entire tree will be destroyed. When the number of nodes in the left subtree of node T is less than the number of nodes in the right subtree, select the node with the minimum value in its right subtree as the actual deletion node X; otherwise, select the node with the maximum value in its left subtree as the actual deletion node X. Then swap the positions of node T and node X, and it is completely the same as the first case.

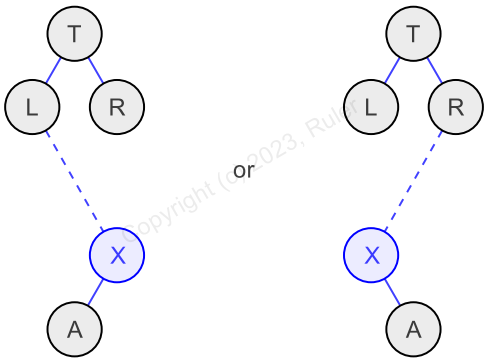

Figure 1.7-2 Case 2

The C++ code for the deletion operation is as follows:

```
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;
```

```
        // the rebalance flag
        flag = (t == t->parent->right);
        // removes t node
        if (x)
            x->parent = t->parent;
        if (t == header->parent)
            header->parent = x;
        else if (t == t->parent->left)
            t->parent->left = x;
        else
            t->parent->right = x;
        if (t == header->left)
            header->left = x ? leftmost(x) : t->parent;
        if (t == header->right)
            header->right = x ? rightmost(x) : t->parent;
        // reduces the number of nodes
        for (node_pointer p = t->parent; p != header; p = p->parent)
            --p->size;
        // rebalance after deletion
        if (t != header)
        {
            node_pointer p = erase_rebalance(t->parent, flag);
            while (p != header)
                p = erase_rebalance(p->parent, p == p->parent->right);
        }
    }
    // case 2. has two child nodes
    else
    {
        if (t->left->size < t->right->size)
        {
            x = leftmost(t->right);
            // the rebalance flag
            flag = (x == x->parent->right);
            // reduces the number of nodes
            for (node_pointer p = x->parent; p != header; p = p->parent)
                --p->size;
            // replaces t node with x node and removes t node
            t->left->parent = x;
            x->left = t->left;
            if (x != t->right)
            {
                x->parent->left = x->right;
                if (x->right)
                    x->right->parent = x->parent;
                t->right->parent = x;
                x->right = t->right;
                parent = x->parent;
            }
```

```
            else
                parent = x;
            if (t == header->parent)
                header->parent = x;
            else if (t == t->parent->left)
                t->parent->left = x;
            else
                t->parent->right = x;
            x->parent = t->parent;
            x->size = t->size;
        }
        else
        {
            x = rightmost(t->left);
            // the rebalance flag
            flag = (x == x->parent->right);
            // reduces the number of nodes
            for (node_pointer p = x->parent; p != header; p = p->parent)
                --p->size;
            // replaces t node with x node and removes t node
            t->right->parent = x;
            x->right = t->right;
            if (x != t->left)
            {
                x->parent->right = x->left;
                if (x->left)
                    x->left->parent = x->parent;
                t->left->parent = x;
                x->left = t->left;
                parent = x->parent;
            }
            else
                parent = x;
            if (t == header->parent)
                header->parent = x;
            else if (t == t->parent->left)
                t->parent->left = x;
            else
                t->parent->right = x;
            x->parent = t->parent;
            x->size = t->size;
        }
        // rebalance after deletion
        node_pointer p = erase_rebalance(parent, flag);
        while (p != header)
            p = erase_rebalance(p->parent, p == p->parent->right);
    }
    // destroy node
    destroy_node(t);
```

```
108. }
```

1.8 Searching

The search operation is exactly the same as that of the standard binary search tree. It starts by searching from the root node; if the value being searched for, "key," is less than the node's key value, the search continues in the left subtree. If "key" is greater than the node's key value, the search continues in the right subtree. If "key" is equal to the node's key value, the search is complete.

The C++ code for the searching operation is as follows:

```
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 Selection

The select operation provides random access function in SBT. Each node keeps track of the number of nodes in its subtree, and this information can be used to find elements of a specific rank. To search for the node with rank k, you check the number of nodes in the left subtree. If the size of the left subtree is greater than k, then continue searching in the left subtree. If the size of the left subtree is less than k, then search for the node with rank k - size - 1 in the right subtree. If the size is equal to k, the search is complete.

The C++ code for the selection operation is as follows:

```
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 Ranking

The rank operation is the inverse of the select operation, which returns the rank of a given key. If the key of the current node is not less than the given key, continue searching in the left subtree. Otherwise, return the rank of the current node plus the rank of the key in the right subtree.

The C++ code for the ranking operation is as follows:

```
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;
}
```