1 Array Based on Tree

An Array Based on Tree (ABT) is a data structure that represents an array in the form of a self-balancing binary tree that maintains a balance between the size of its left and right subtrees. Unlike binary search tree that sort nodes based on their key values, ABT maintains the insertion order of elements. ABT not only has the ability of random access but also has the ability of dynamic editing. The time complexity of its random access, insertion, and deletion operations is O(log n).

1.1 Applicability

Array and list are two common data structures. Array allow random access to elements in O(1) time, while list allow insertion and deletion of elements in O(1) time. However, sometimes we need a data structure that combines the advantages of both array and list. As a result, the concept of an Array Based on Tree was invented. ABT balances the performance of random access, insertion, and deletion operations well, making it suitable for application scenarios that require both random access and dynamic editing.

1.2 Properties

An Array Based on Tree is a data structure based on a non-sorted 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 ABTs.

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

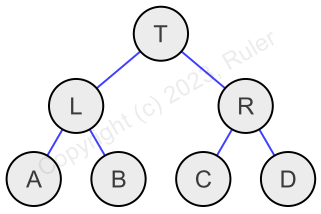

Figure1.2 Array Based on Tree

According to the properties of ABT, 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 ABT include a parent node, two child nodes, and the size of nodes in the subtree where the node is located.

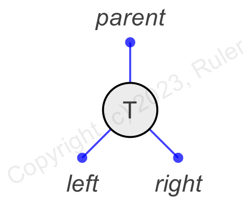

Figure1.3 Node

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

```
template <class T>
struct ab_tree_node
{
    using node_type             = ab_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

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.

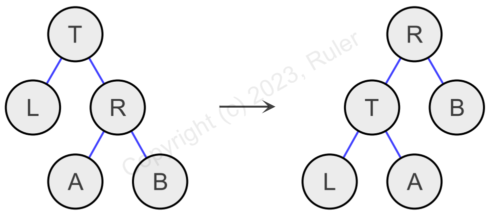

Figure 1.4.1 Left rotation

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

```cpp
node_pointer left_rotate(node_pointer t) const
{
  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

Right rotation is similar to left rotation but in the opposite direction. 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.

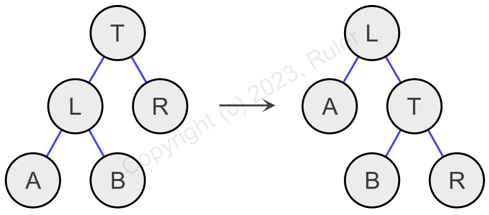

Figure 1.4.2 Right rotation

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

```cpp
node_pointer right_rotate(node_pointer t) const
{
    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 ABT, the properties of ABT may be violated, and it is necessary to rebalance the ABT rooted at node T. The prerequisite is that the child nodes of T are already ABTs themselves. There are four cases to consider when rebalancing:

(1) When 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 ABT, while the left subtree T and the right subtree R may violate the properties of ABT. 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, the ancestor nodes of the node C should be rebalanced one by one until the root node is reached.

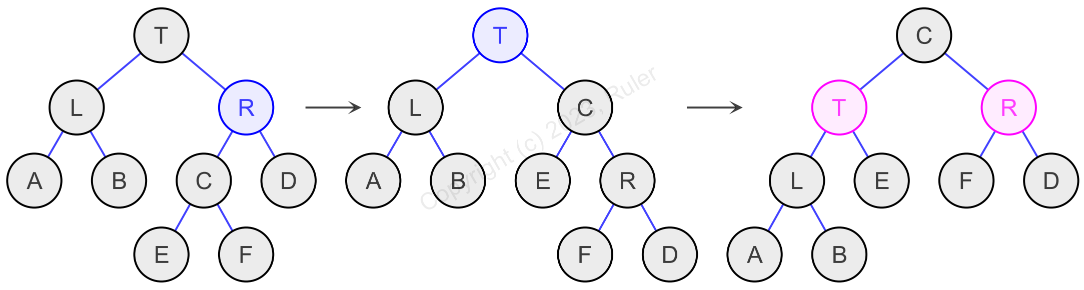

Figure 1.5-1 Case 1

(2) When 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 ABT. 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, the ancestor nodes of the node R should be rebalanced one by one until the root node is reached.

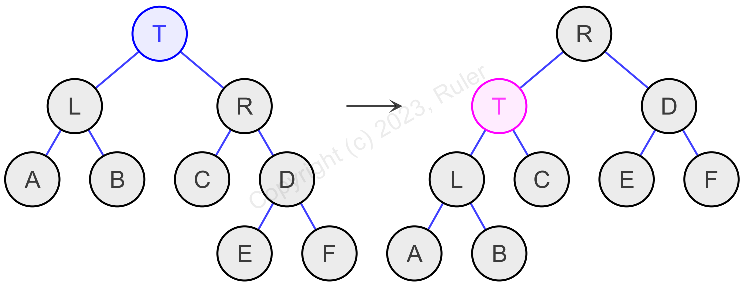

Figure 1.5-2 Case 2

(3) When 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 ABT, while the left subtree L and the right subtree T may violate the properties of ABT. 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, the ancestor nodes of the node B should be rebalanced one by one until the root node is reached.

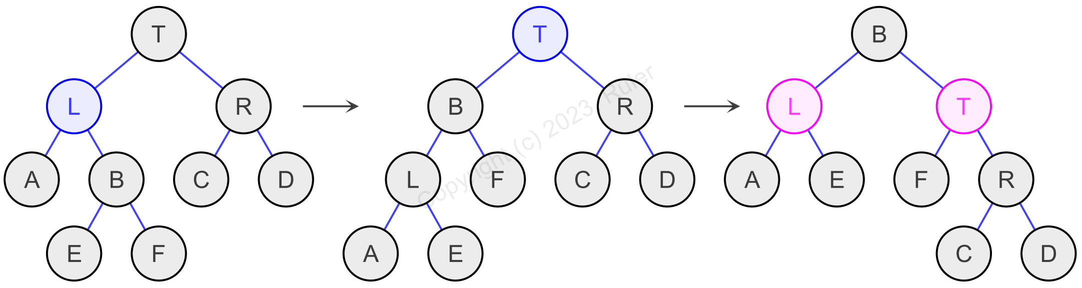

Figure 1.5-3 Case 3

(4) When 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 ABT. 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, the ancestor nodes of the node L should be rebalanced one by one until the root node is reached.

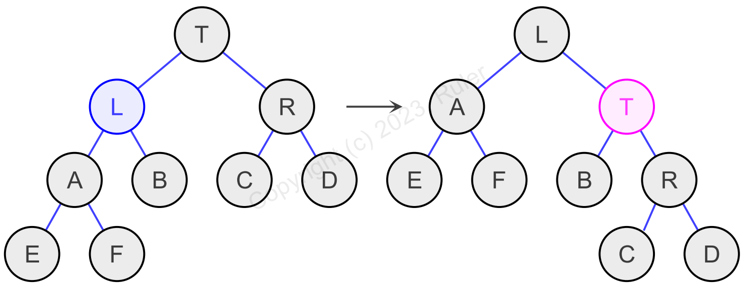

Figure 1.5-4 Case 4

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

```cpp
void 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);
              insert_rebalance(t->left, false);
              insert_rebalance(t->right, 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);
              insert_rebalance(t->left, 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);
              insert_rebalance(t->left, false);
              insert_rebalance(t->right, true);
          }
          // case 4. size(T.right) < size(T.left.left)
```

```cpp
            else if (t->left->left && right_size < t->left->left->size)
            {
                t = right_rotate(t);
                insert_rebalance(t->right, true);
            }
        }
    }
    if (t->parent != header)
        insert_rebalance(t->parent, t == t->parent->right);
}
```

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

```cpp
void erase_rebalance(node_pointer t, bool flag)
{
    if (flag)
    {
        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);
                erase_rebalance(t->left, false);
                erase_rebalance(t->right, 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);
                erase_rebalance(t->right, true);
            }
        }
    }
    else
    {
        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);
                erase_rebalance(t->left, false);
                erase_rebalance(t->right, 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);
              erase_rebalance(t->left, false);
          }
      }
  }
  if (t->parent != header)
      erase_rebalance(t->parent, t == t->parent->right);
}
```

1.6 Insertion

If the ABT is empty, directly add the node as the root node. Otherwise, it can be divided into the following two cases based on the situation of inserting child nodes of node T:

(1) Node T does not have a left child node, and directly insert into the left subtree of node T in this case. The node counts of its ancestor nodes should all be incremented by 1. The above-mentioned case 3 or case 4 may occur, so it is necessary to rebalance node T.

(2) Node T has a left child node, and selects the last node in its left subtree as the actual insertion node X in this case. Insert into the right subtree of node X, and the node counts of its ancestor nodes should all be incremented by 1. The above-mentioned case 1 or case 2 may occur, so it is necessary to rebalance node X.

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

```
template<class ...Args>
node_pointer insert_node(node_pointer t, 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 (t == header)
  {
      // if the tree is empty
      if (!header->parent)
      {
          // inserts the node
          n->parent = t;
          header->parent = n;
          header->left = n;
          header->right = n;
      }
      else
      {
          t = header->right;
```

```
            // inserts the node
            n->parent = t;
            t->right = n;
            header->right = n;
            // increases the size of nodes
            for (node_pointer p = t; p != header; p = p->parent)
                ++p->size;
            // rebalance after insertion
            insert_rebalance(t, true);
        }
    }
    else if (t->left)
    {
        t = t->left;
        while (t->right)
            t = t->right;
        // inserts the node
        n->parent = t;
        t->right = n;
        // increases the size of nodes
        for (node_pointer p = t; p != header; p = p->parent)
            ++p->size;
        // rebalance after insertion
        insert_rebalance(t, true);
    }
    else
    {
        // inserts the node
        n->parent = t;
        t->left = n;
        if (t == header->left)
            header->left = n;
        // increases the size of nodes
        for (node_pointer p = t; p != header; p = p->parent)
            ++p->size;
        // rebalance after insertion
        insert_rebalance(t, false);
    }
    return n;
}
```

1.7 deletion

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, and the node T can be deleted directly in this case. 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.

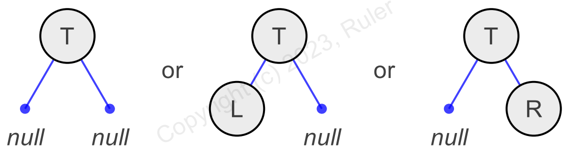

Figure 1.7-1 Case 1

(2) Node T has two child nodes, and the node T cannot be deleted directly in this case, 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.

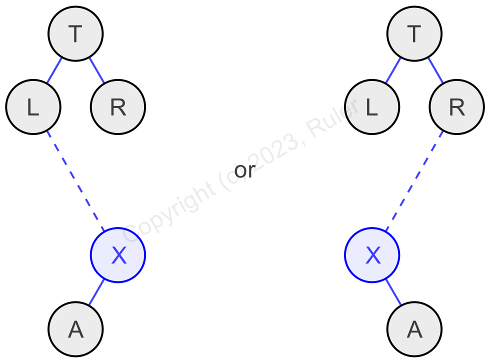

Figure 1.7-2 Case 2

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

```cpp
void erase_node(node_pointer t)
{
  bool flag;
  node_pointer x;
  // case 1. has one child node at most
  if (!t->left || !t->right)
  {
      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
        erase_rebalance(t->parent, flag);
    }
    // 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;
            }
            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;
          }
          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
      erase_rebalance(x->parent, flag);
  }
  // destroy node
  this->destroy_node(t);
}
```

1.8 selection

Selection operation provides random access function in ABT.

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

```
node_pointer select_node(size_type k) const
{
  node_pointer t = header->parent;
  while (t)
  {
      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;
}
```