1 基于树的数组

基于树的数组（An Array Based on Tree，简称 ABT），是一种基本数据结构，它是基于平衡二叉树的数组，并根据其左右子树的节点数来维持平衡。不同于二叉搜索树（Binary Search Tree），ABT 维持的是元素的插入顺序而非节点的键值顺序。ABT 不仅具备随机访问能力，而且具备动态编辑能力，其随机访问、插入和删除操作的时间复杂度均为 O(log n) 。

1.1 适用场景

数组（Array）和链表（List）是两种常见的数据结构。数组的随机访问时间复杂度为 O(1)，而链表的插入和删除时间复杂度为 O(1)。然而，有时我们需要一种数据结构，能够结合数组和链表的优点。因此，基于树的数组（Array Based on Tree）的概念应运而生。ABT 很好地平衡了随机访问、插入和删除操作的性能，适用于同时需要随机访问与动态编辑的应用场景。

1.2 特性

ABT 是一种基于非排序二叉树的数据结构，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 ABT。

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

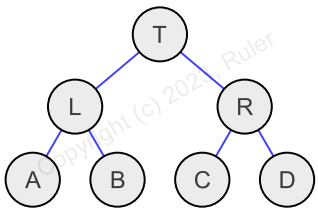

图 1.2 基于树的数组

根据 ABT 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

ABT 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。

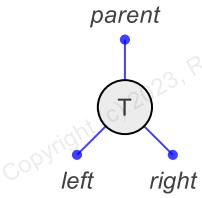

图 1.3 节点

节点定义的 C++代码如下：

```cpp
template <class T>
struct 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 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。

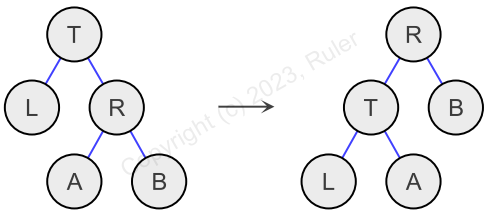

图 1.4.1 左旋转

左旋转的 C++代码如下：

```
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。

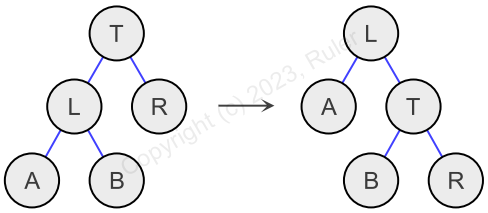

图 1.4.2 右旋转

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 ABT 进行插入或者删除操作后，可能违反 ABT 的特性，需要对以 T 为根的 ABT 进行重新平衡。再平衡的前提是 T 的子树均满足 ABT 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 ABT 的特性；而左子树 T 和右子树 R 可能违反 ABT 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。

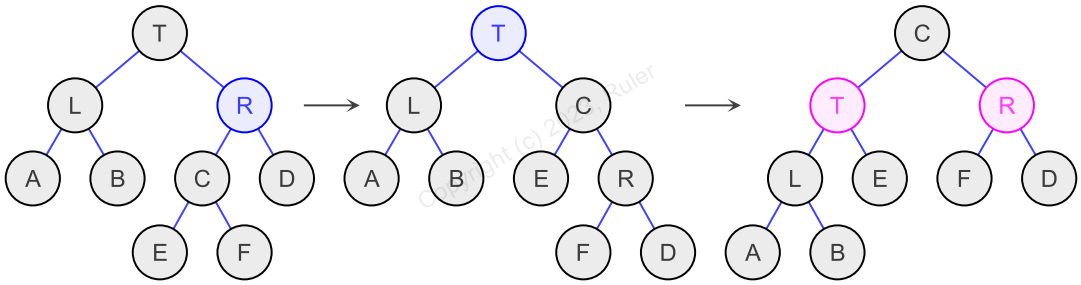

图 1.5-1 第（1）种情况

(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 ABT 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。

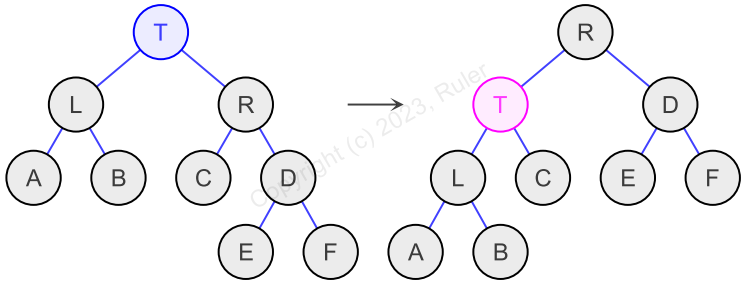

图 1.5-2 第（2）种情况

(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 ABT 的性质；而左子树 L 和右子树 T 可能违反 ABT 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。

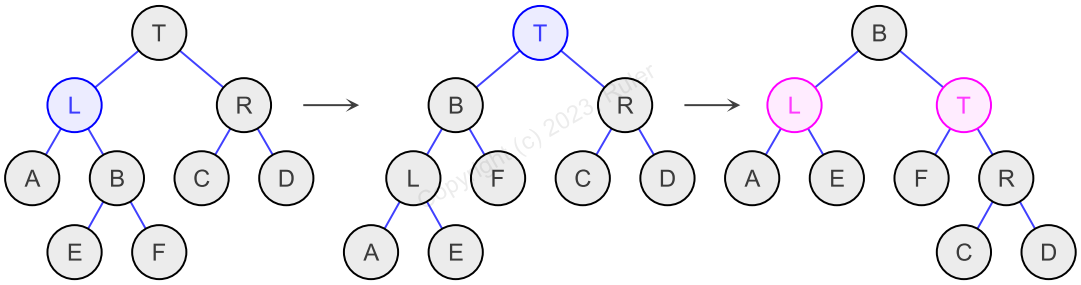

图 1.5-3 第（3）种情况

(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。

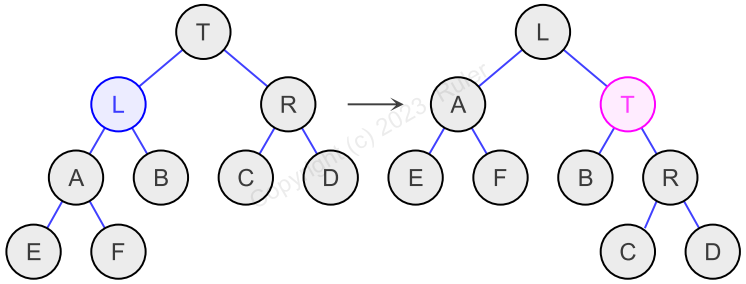

图 1.5-4 第（4）种情况

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);
```

```
            }
            // 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;
}
```

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);
```

```
13.                 t->left = erase_rebalance(t->left, true);
14.                 t->right = erase_rebalance(t->right, false);
15.                 t = erase_rebalance(t, false);
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 = erase_rebalance(t->left, true);
22.                 t = erase_rebalance(t, false);
23.             }
24.         }
25.     }
26.     else
27.     {
28.         if (t->left)
29.         {
30.             size_type right_size = t->right ? t->right->size : 0;
31.             // case 3. size(T.right) < size(T.left.right)
32.             if (t->left->right && right_size < t->left->right->size)
33.             {
34.                 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 插入操作

如果 ABT 为空，则直接添加该节点作为根节点。否则，根据插入节点 T 的子节点的情况，可以分为以下两种情况：

(1) 节点 T 没有左子节点

在这种情况下直接插入到节点 T 的左子树中。其祖先节点的节点数应全部加 1。可能出现上述第（3）种情况或者第（4）种情况，因此，节点 T 需要重新平衡。

(2) 节点 T 已有左子节点

在这种情况下选择其左子树中的最后一个节点作为实际插入节点 X。插入到节点 X 的右子树中，其祖先节点的节点数应全部加 1。可能会出现上述第（1）种情况或者第（2）种情况，因此，节点 X 需要重新平衡。

插入操作的 C++代码如下：

```
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;
          do
          {
              // rebalance after insertion
              t = insert_rebalance(t->parent, t == t->parent->right);
          } while (t->parent != header);
      }
  }
  else if (t->left)
  {
      t = t->left;
      while (t->right)
          t = t->right;
      // inserts the node
      n->parent = t;
```

```
44.         t->right = n;
45.         // increases the size of nodes
46.         for (node_pointer p = t; p != header; p = p->parent)
47.             ++p->size;
48.         do
49.         {
50.             // rebalance after insertion
51.             t = insert_rebalance(t->parent, t == t->parent->right);
52.         } while (t->parent != header);
53.     }
54.     else
55.     {
56.         // inserts the node
57.         n->parent = t;
58.         t->left = n;
59.         if (t == header->left)
60.             header->left = n;
61.         // increases the size of nodes
62.         for (node_pointer p = t; p != header; p = p->parent)
63.             ++p->size;
64.         do
65.         {
66.             // rebalance after insertion
67.             t = insert_rebalance(t->parent, t == t->parent->right);
68.         } while (t->parent != header);
69.     }
70.     return n;
71. }
```

1.7 删除操作

假设要删除的节点是 T，根据节点 T 的子节点数量，可以分为以下两种情况：

(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。

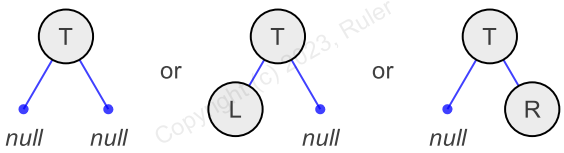

图 1.7-1 第（1）种情况

(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。

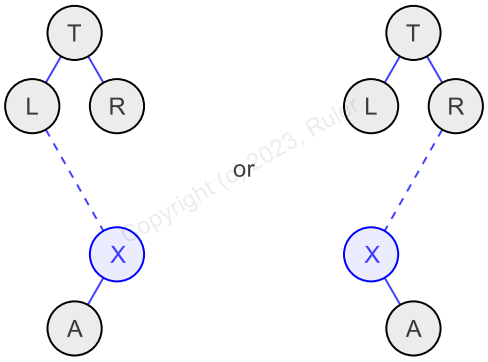

图 1.7-2 第（2）种情况

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);
```

```
        }
    }
    // 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)
            {
```

```
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 选择操作

选择操作为 ABT 提供随机访问功能。

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;
7.        if (left_size < k)
8.        {
9.            t = t->right;
10.           k -= (left_size + 1);
11.       }
12.       else if (k < left_size)
13.           t = t->left;
14.       else
15.           return t;
16.   }
17.   return header;
```

```
18. }
```