

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。

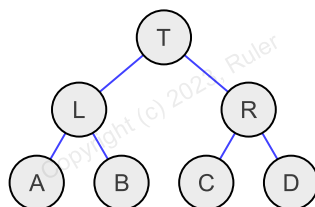


图 1.2 基于树的数组

根据 ABT 的特性, 节点 T 必须满足:

- (1) $\text{size}(L) \geq \max(\text{size}(C), \text{size}(D))$
- (2) $\text{size}(R) \geq \max(\text{size}(A), \text{size}(B))$

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。

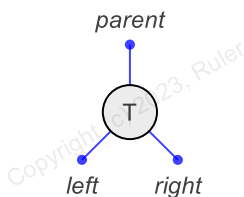


图 1.3 节点

节点定义的 C++ 代码如下:

```
1.  template <class T>
2.  struct ab_tree_node
3.  {
4.      using node_type          = ab_tree_node<T>;
5.      using node_pointer       = node_type*;
6.      using const_node_pointer = const node_type*;
7.      using node_reference     = node_type&;
8.      using const_node_reference = const node_type&;
9.
10.     node_pointer      parent;
11.     node_pointer      left;
12.     node_pointer      right;
13.     size_t            size;
14.     T                 data;
15. };
```

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。

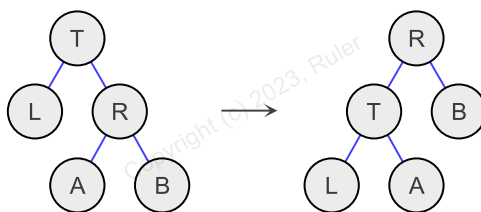


图 1.4.1 左旋转

左旋转的 C++ 代码如下:

```
1.  node_pointer left_rotate(node_pointer t) const
2.  {
```

```
3.  node_pointer r = t->right;
4.  t->right = r->left;
5.  if (r->left)
6.      r->left->parent = t;
7.  r->parent = t->parent;
8.  if (t == header->parent)
9.      header->parent = r;
10. else if (t == t->parent->left)
11.     t->parent->left = r;
12. else
13.     t->parent->right = r;
14. r->left = t;
15. r->size = t->size;
16. t->parent = r;
17. t->size = (t->left ? t->left->size : 0) + (t->right ? t->right->size : 0) + 1;
18. return r;
19. }
```

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。

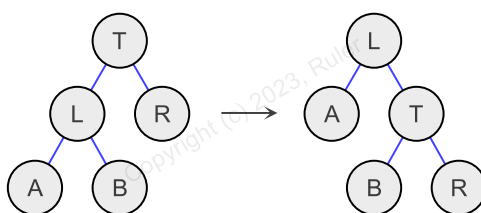


图 1.4.2 右旋转

右旋转的 C++代码如下：

```
1.  node_pointer right_rotate(node_pointer t) const
2.  {
3.      node_pointer l = t->left;
4.      t->left = l->right;
5.      if (l->right)
6.          l->right->parent = t;
7.      l->parent = t->parent;
8.      if (t == header->parent)
9.          header->parent = l;
10.     else if (t == t->parent->right)
11.         t->parent->right = l;
12.     else
13.         t->parent->left = l;
14.     l->right = t;
15.     l->size = t->size;
```

```

16.   t->parent = l;
17.   t->size = (t->left ? t->left->size : 0) + (t->right ? t->right->size : 0) + 1;
18.   return l;
19. }

```

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) $\text{size}(T.\text{left}) < \text{size}(T.\text{right}.\text{left})$ ，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 $\text{size}(T.\text{right}) < \text{size}(T.\text{left}.\text{child})$ ，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 $\text{size}(R.\text{left}) < \text{size}(R.\text{right}.\text{child})$ ，需要重新平衡。最后，对节点 C 的祖先节点逐一进行重新平衡，直到根节点为止。

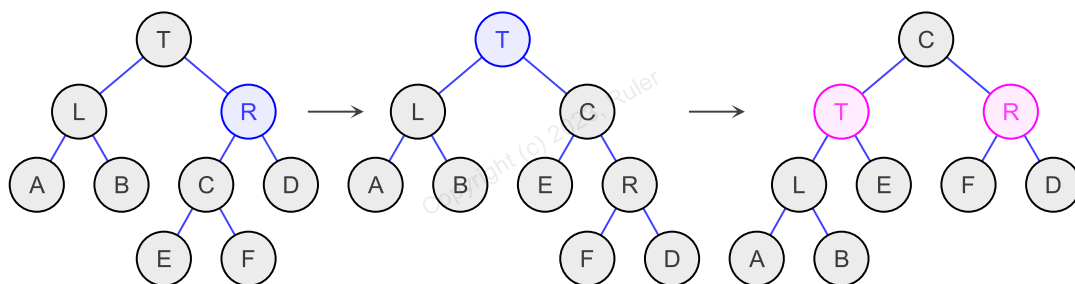


图 1.5-1 第 (1) 种情况

- (2) $\text{size}(T.\text{left}) < \text{size}(T.\text{right}.\text{right})$ ，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 $\text{size}(T.\text{right}) < \text{size}(T.\text{left}.\text{child})$ ，需要重新平衡。最后，对节点 R 的祖先节点逐一进行重新平衡，直到根节点为止。

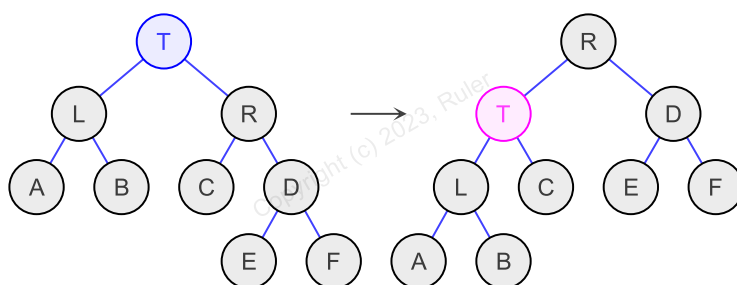


图 1.5-2 第 (2) 种情况

- (3) $\text{size}(T.\text{right}) < \text{size}(T.\text{left}.\text{right})$ ，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 $\text{size}(L.\text{right}) < \text{size}(L.\text{left}.\text{child})$ ，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

$\text{size}(T.\text{left}) < \text{size}(T.\text{right}.child)$, 需要重新平衡。最后, 对节点 B 的祖先节点逐一进行重新平衡, 直到根节点为止。

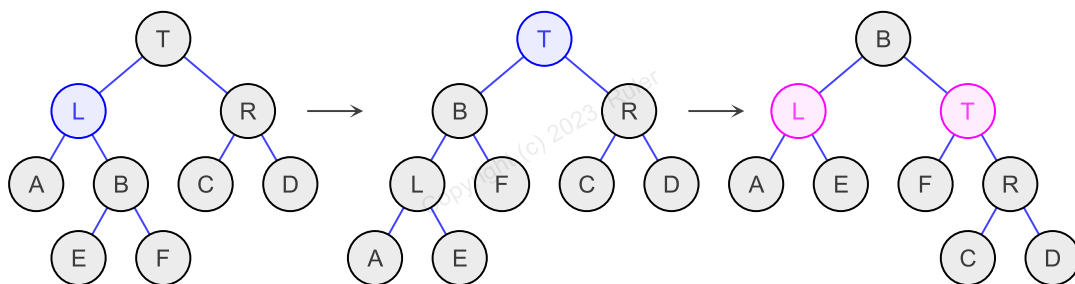


图 1.5-3 第 (3) 种情况

- (4) $\text{size}(T.\text{right}) < \text{size}(T.\text{left}.left)$, 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 $\text{size}(T.\text{left}) < \text{size}(T.\text{right}.child)$, 需要重新平衡。最后, 对节点 L 的祖先节点逐一进行重新平衡, 直到根节点为止。

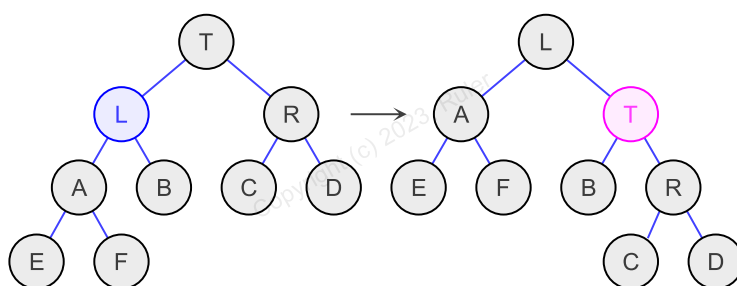


图 1.5-4 第 (4) 种情况

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. void 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.                insert_rebalance(t->left, false);
14.                insert_rebalance(t->right, true);
15.            }
16.            // case 2. size(T.left) < size(T.right.right)
17.            else if (t->right->right && left_size < t->right->right->size)
18.            {
```

```
19.         t = left_rotate(t);
20.         insert_rebalance(t->left, false);
21.     }
22. }
23. }
24. else
25. {
26.     if (t->left)
27.     {
28.         size_type right_size = t->right ? t->right->size : 0;
29.         // case 3. size(T.right) < size(T.left.right)
30.         if (t->left->right && right_size < t->left->right->size)
31.         {
32.             t->left = left_rotate(t->left);
33.             t = right_rotate(t);
34.             insert_rebalance(t->left, false);
35.             insert_rebalance(t->right, true);
36.         }
37.         // case 4. size(T.right) < size(T.left.left)
38.         else if (t->left->left && right_size < t->left->left->size)
39.         {
40.             t = right_rotate(t);
41.             insert_rebalance(t->right, true);
42.         }
43.     }
44. }
45. if (t->parent != header)
46.     insert_rebalance(t->parent, t == t->parent->right);
47. }
```

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:

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

```

19.         t = right_rotate(t);
20.         erase_rebalance(t->right, true);
21.     }
22. }
23. }
24. else
25. {
26.     if (t->right)
27.     {
28.         size_type left_size = t->left ? t->left->size : 0;
29.         // case 1: size(T.left) < size(T.right.left)
30.         if (t->right->left && left_size < t->right->left->size)
31.         {
32.             t->right = right_rotate(t->right);
33.             t = left_rotate(t);
34.             erase_rebalance(t->left, false);
35.             erase_rebalance(t->right, true);
36.         }
37.         // case 2: size(T.left) < size(T.right.right)
38.         else if (t->right->right && left_size < t->right->right->size)
39.         {
40.             t = left_rotate(t);
41.             erase_rebalance(t->left, false);
42.         }
43.     }
44. }
45. if (t->parent != header)
46.     erase_rebalance(t->parent, t == t->parent->right);
47. }

```

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：

```

1. template<class ...Args>
2. node_pointer insert_node(node_pointer t, 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 (t == header)
10.    {
11.        // if the tree is empty
12.        if (!header->parent)
13.        {
14.            // inserts the node
15.            n->parent = t;
16.            header->parent = n;
17.            header->left = n;
18.            header->right = n;
19.        }
20.        else
21.        {
22.            t = header->right;
23.            // inserts the node
24.            n->parent = t;
25.            t->right = n;
26.            header->right = n;
27.            // increases the size of nodes
28.            for (node_pointer p = t; p != header; p = p->parent)
29.                ++p->size;
30.            // rebalance after insertion
31.            insert_rebalance(t, true);
32.        }
33.    }
34.    else if (t->left)
35.    {
36.        t = t->left;
37.        while (t->right)
38.            t = t->right;
39.        // inserts the node
40.        n->parent = t;
41.        t->right = n;
42.        // increases the size of nodes
43.        for (node_pointer p = t; p != header; p = p->parent)
44.            ++p->size;
45.        // rebalance after insertion
46.        insert_rebalance(t, true);
47.    }
48.    else
49.    {
50.        // inserts the node
51.        n->parent = t;
52.        t->left = n;
53.        if (t == header->left)
54.            header->left = n;
```



```

55.     // increases the size of nodes
56.     for (node_pointer p = t; p != header; p = p->parent)
57.         ++p->size;
58.     // rebalance after insertion
59.     insert_rebalance(t, false);
60. }
61. return n;
62. }

```

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。

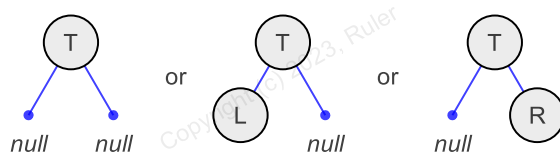


图 1.7-1 第 (1) 种情况

- (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。

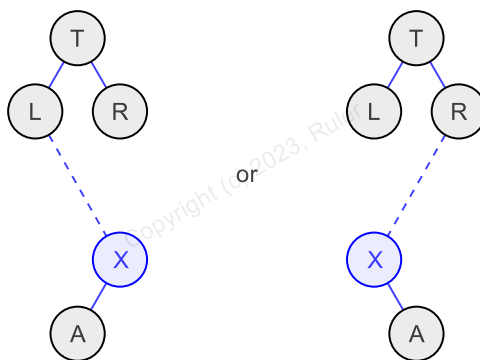


图 1.7-2 第 (2) 种情况

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.     // case 1. has one child node at most
6.     if (!t->left || !t->right)
7.     {

```

```
8.      x = t->left ? t->left : t->right;
9.      // the rebalance flag
10.     flag = (t == t->parent->right);
11.     // removes t node
12.     if (x)
13.         x->parent = t->parent;
14.     if (t == header->parent)
15.         header->parent = x;
16.     else if (t == t->parent->left)
17.         t->parent->left = x;
18.     else
19.         t->parent->right = x;
20.     if (t == header->left)
21.         header->left = x ? leftmost(x) : t->parent;
22.     if (t == header->right)
23.         header->right = x ? rightmost(x) : t->parent;
24.     // reduces the number of nodes
25.     for (node_pointer p = t->parent; p != header; p = p->parent)
26.         --p->size;
27.     // rebalance after deletion
28.     erase_rebalance(t->parent, flag);
29. }
30. // case 2. has two child nodes
31. else
32. {
33.     if (t->left->size < t->right->size)
34.     {
35.         x = leftmost(t->right);
36.         // the rebalance flag
37.         flag = (x == x->parent->right);
38.         // reduces the number of nodes
39.         for (node_pointer p = x->parent; p != header; p = p->parent)
40.             --p->size;
41.         // replaces t node with x node and removes t node
42.         t->left->parent = x;
43.         x->left = t->left;
44.         if (x != t->right)
45.         {
46.             x->parent->left = x->right;
47.             if (x->right)
48.                 x->right->parent = x->parent;
49.             t->right->parent = x;
50.             x->right = t->right;
51.         }
52.         if (t == header->parent)
53.             header->parent = x;
54.         else if (t == t->parent->left)
55.             t->parent->left = x;
56.         else
```

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

1.8 选择操作

选择操作在 ABT 中提供随机访问功能。

插入操作的 C++代码如下：

```
1.  node_pointer select_node(size_type k) const
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. }
```