# 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) 。

## 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。

## 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。

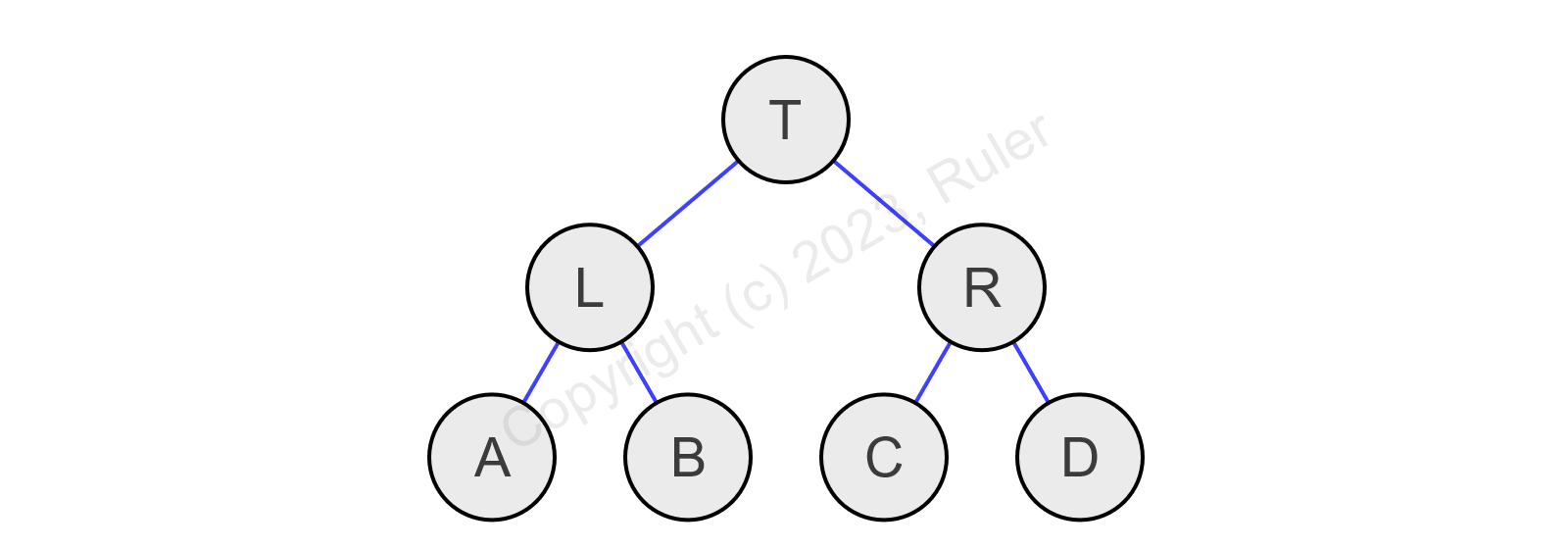


图1.2 基于树的数组

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))

## 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。

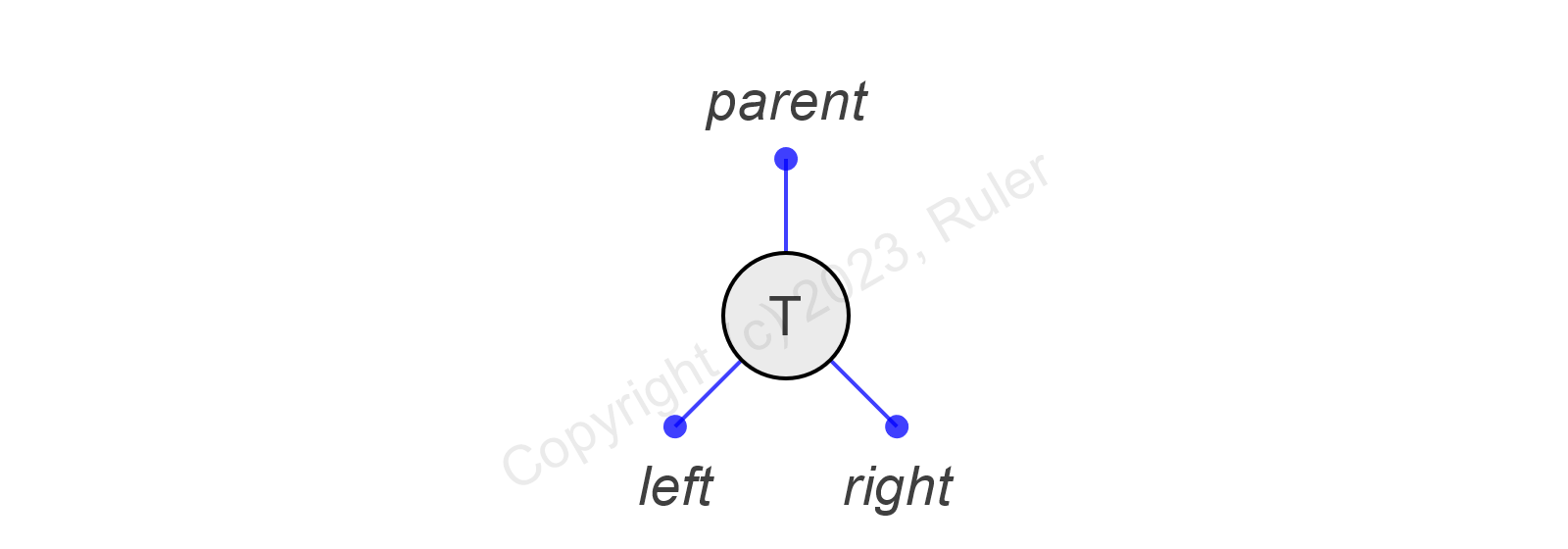


图1.3 节点

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. node\_pointer parent;
10. node\_pointer left;
11. node\_pointer right;
12. size\_t size;
13. T data;
14. };

## 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。

### 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。

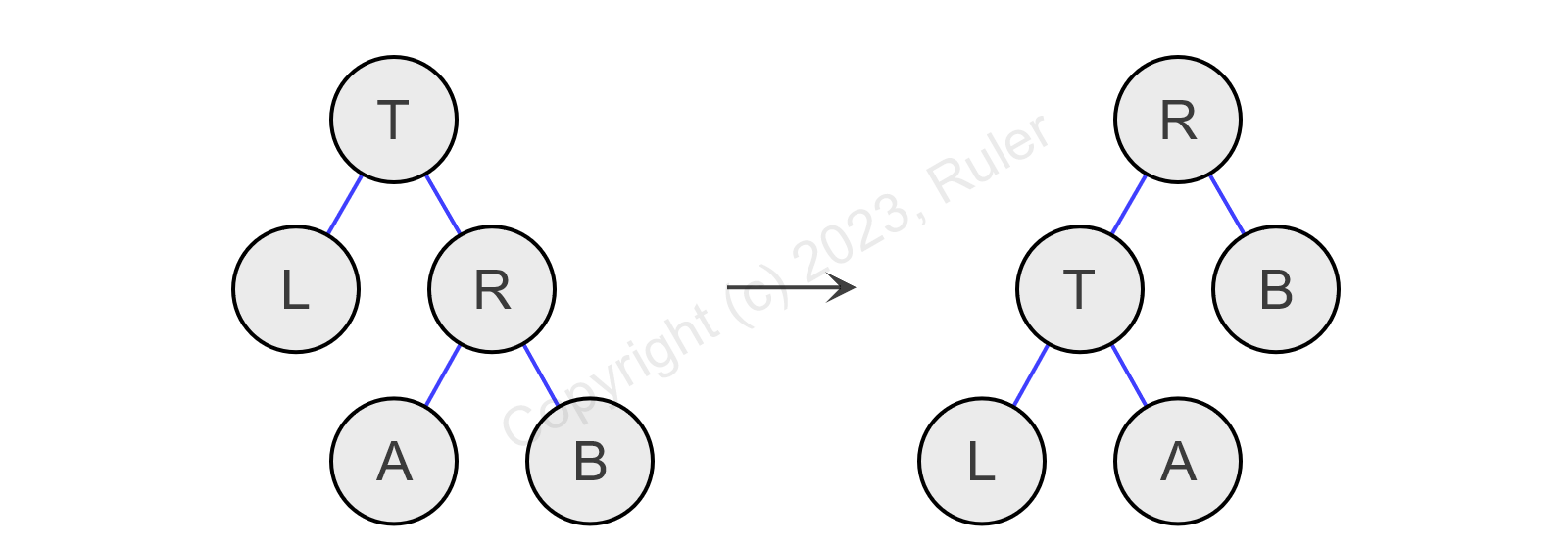


图 1.4.1 左旋转

左旋转的C++代码如下：

1. node\_pointer left\_rotate(node\_pointer t)
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. }

### 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。

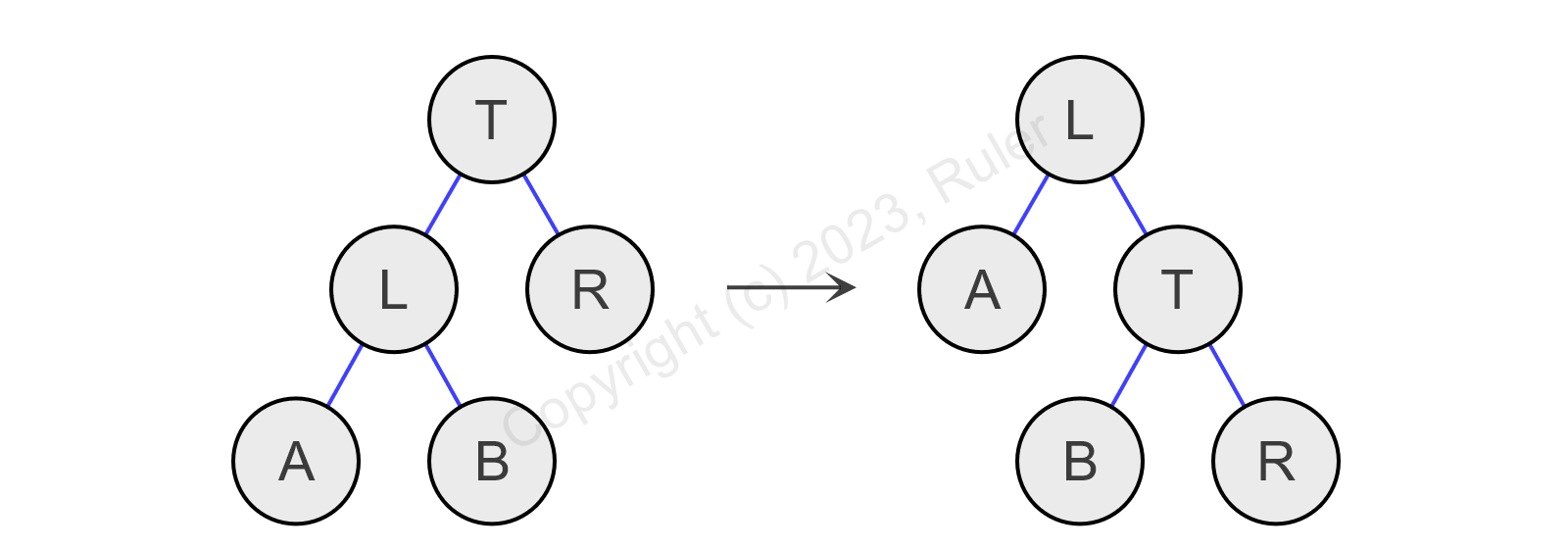


图 1.4.2 右旋转

右旋转的C++代码如下：

1. node\_pointer right\_rotate(node\_pointer t)
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. }

## 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。

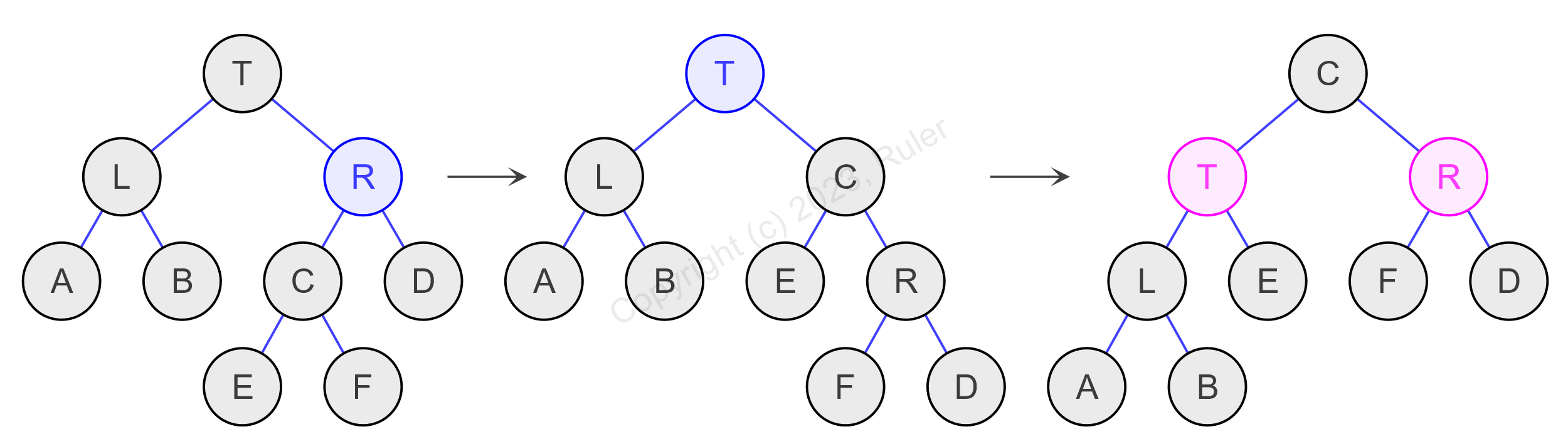


图1.5‑1 第（1）种情况

1. 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。

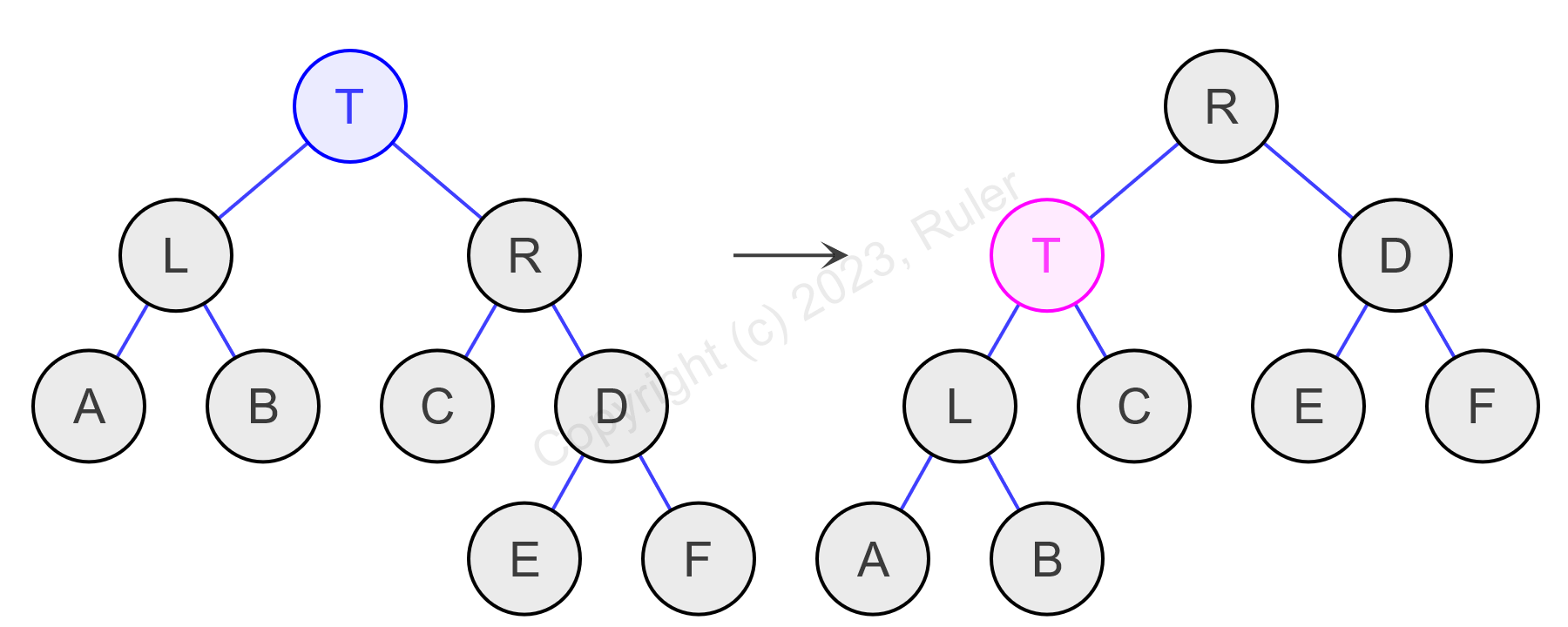


图 1.5‑2 第（2）种情况

1. 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。

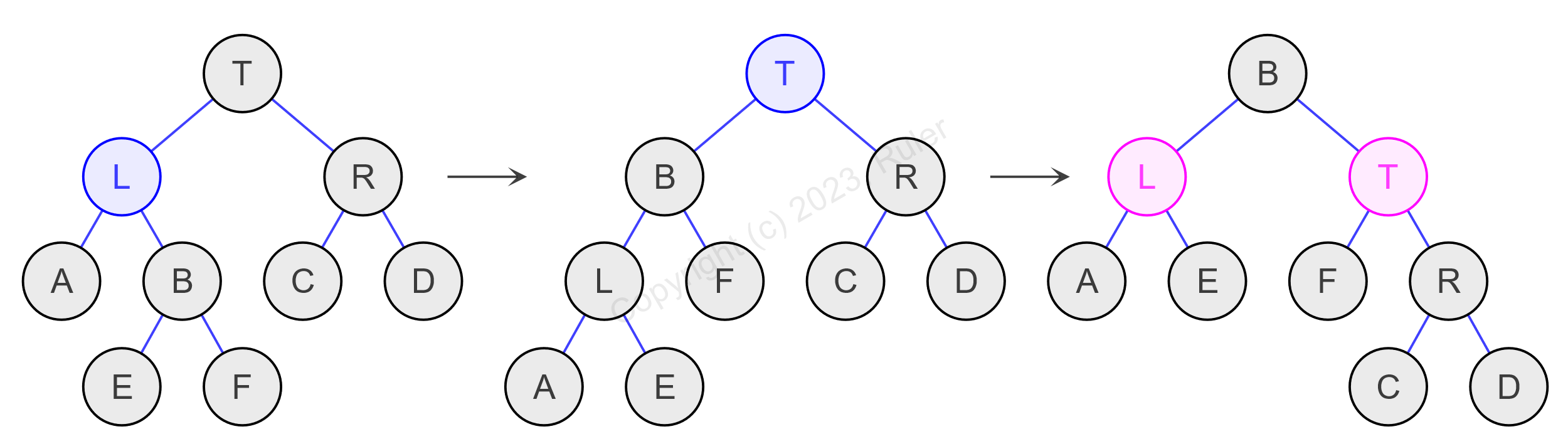


图 1.5‑3 第（3）种情况

1. 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。

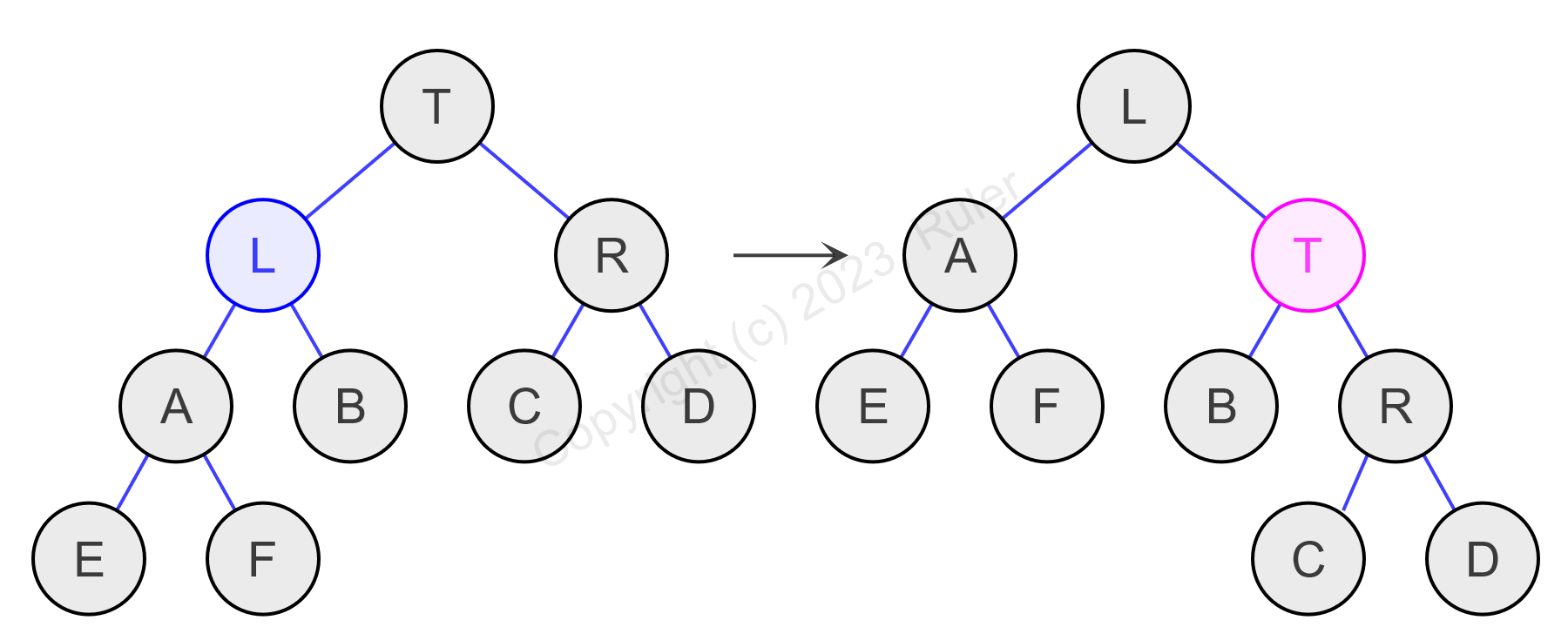


图 1.5‑4 第（4）种情况

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);
22. t = insert\_rebalance(t, true);
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 = insert\_rebalance(t->left, false);
37. t->right = insert\_rebalance(t->right, true);
38. t = insert\_rebalance(t, false);
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 = insert\_rebalance(t->right, true);
45. t = insert\_rebalance(t, false);
46. }
47. }
48. }
49. return t;
50. }

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. node\_pointer erase\_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 = 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. }

## 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。

1. 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. do
31. {
32. // rebalance after insertion
33. t = insert\_rebalance(t->parent, t == t->parent->right);
34. } while (t->parent != header);
35. }
36. }
37. else if (t->left)
38. {
39. t = t->left;
40. while (t->right)
41. t = t->right;
42. // inserts the node
43. 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. }

## 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。

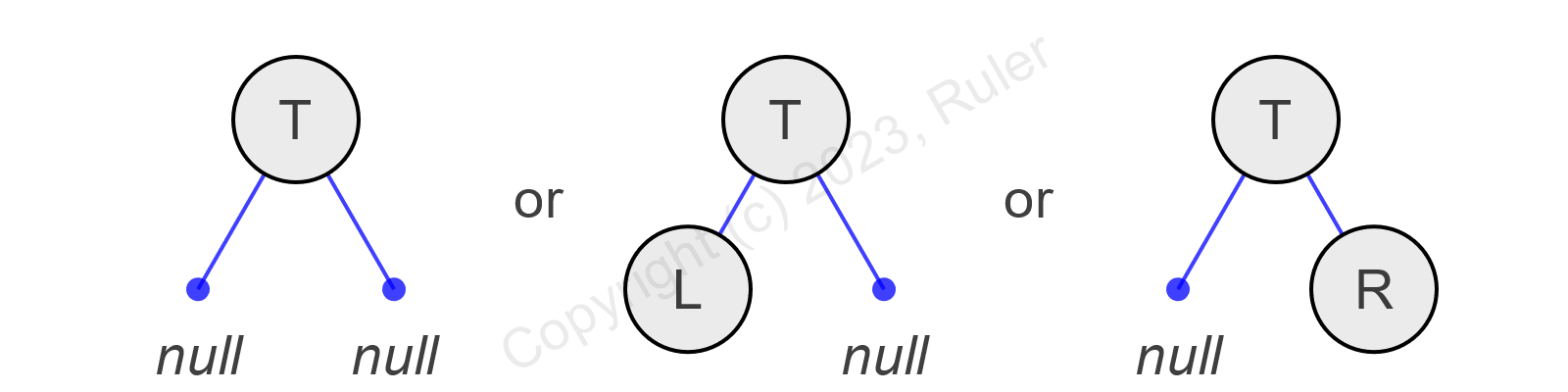


图 1.7‑1 第（1）种情况

1. 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。

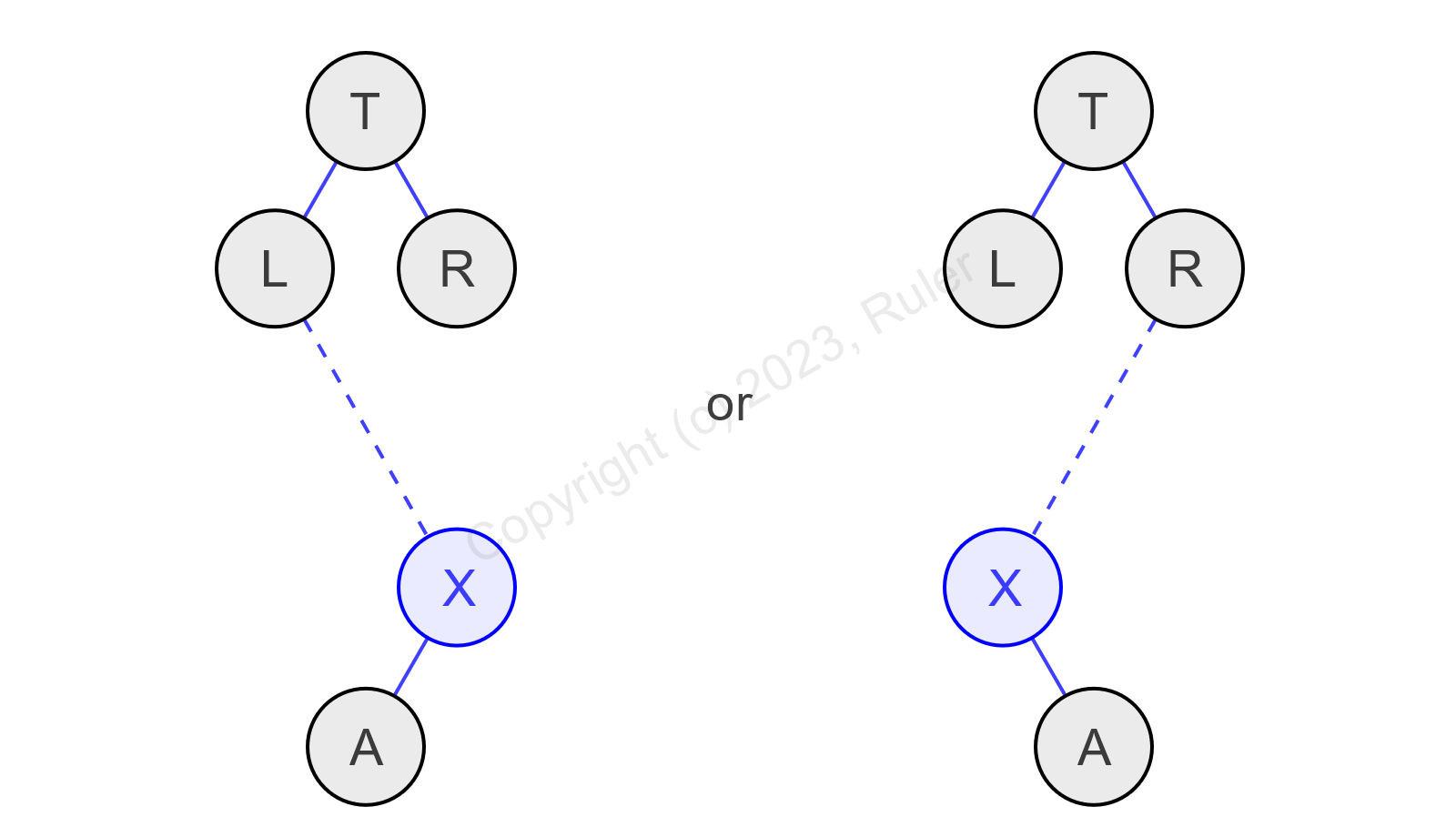


图 1.7‑2 第（2）种情况

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. }

## 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. }