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. Compared with traditional array, ABT not only supports random access but also have editability. ABT supports the standard array and list operations such as random access, insertion, and deletion in O(log n) time.

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. It provides good performance for random access, insertion, and deletion operations.

Example

```
#include "ab_tree.h"

int main(void)
{
    // Creates an array containing integers.
    ab_tree<int> arr({ 7, 1, 8, 0 });
    // Prints all elements of an array.
    std::cout << "array = { ";
    for (auto itr = arr.cbegin(); itr != arr.cend(); ++itr)
        std::cout << *itr << ", ";
    std::cout << "}; \n";

    arr[1] = -1;
    // Selects the third element.
    auto itr = arr.select(2);
    // Inserts tow elements.
    arr.insert(itr, { 10, 16 });

    // Prints all elements of an array.
    std::cout << "array = { ";
    for (const auto& ele : arr)
        std::cout << ele << ", ";
    std::cout << "}; \n";

    return 0;
}
```

Output:

```
array = { 7, 1, 8, 0, };
array = { 7, -1, 10, 16, 8, 0, };
```

Interface

This is a header-only library designed for modern C++. Its interface is similar to that of the list and vector containers in the C++ standard library. Therefore, it can be used easily with almost no learning cost.

ab_tree

Defined in header <ab_tree.h>.

```
template <class T, class Allocator = std::allocator<T>>
class ab_tree;
```

Member types

member type	definition	notes
value_type	The first template parameter (T)	
allocator_type	The second template parameter (Allocator)	
reference	value_type&	
const_reference	const value_type&	
pointer	value_type*	
const_pointer	const value_type*	
size_type	an unsigned integral type that can represent any non-negative value of difference_type	usually the same as size_t
difference_type	a signed integral type	usually the same as ptrdiff_t
iterator	a bidirectional iterator to value_type	convertible to: const_iterator
const_iterator	a bidirectional iterator to const value_type	
reverse_iterator	a bidirectional reverse iterator to value_type	
const_reverse_iterator	a bidirectional reverse iterator to const value_type	
primitive_iterator	a bidirectional primitive iterator to value_type	convertible to: const_primitive_iterator

member type	definition	notes
const_primitive_iterator	a bidirectional primitive iterator to const value_type	
reverse_primitive_iterator	a bidirectional reverse primitive iterator to value_type	
const_reverse_primitive_iterator	a bidirectional reverse primitive iterator to const value_type	

Member functions

function	description
(constructor)	constructs the ab-tree *(public member function)*
(destructor)	destructs the ab-tree *(public member function)*
operator=	assigns values to the container *(public member function)*
assign	assigns values to the container *(public member function)*

Element access

function	description
at	access specified element with bounds checking *(public member function)*
operator[]	access specified element *(public member function)*
front	access the first element *(public member function)*
back	access the last element *(public member function)*

Iterators

iterator	description
begin cbegin	returns an iterator to the beginning *(public member function)*

iterator	description
end cend	returns an iterator to the end *(public member function)*
rbegin crbegin	returns a reverse iterator to the beginning *(public member function)*
rend crend	returns a reverse iterator to the end *(public member function)*
pbegin cpbegin	returns a primitive iterator to the beginning *(public member function)*
pend cpend	returns a primitive iterator to the end *(public member function)*
rpbegin crpbegin	returns a reverse primitive iterator to the beginning *(public member function)*
rpend crpend	returns a reverse primitive iterator to the end *(public member function)*

Capacity

function	description
empty	checks whether container is empty *(public member function)*
size	returns the number of elements *(public member function)*
max_size	returns the maximum possible number of elements *(public member function)*

Modifiers

unction	description
clear	clears the contents *(public member function)*
emplace_front	constructs an element in-place at the beginning *(public member function)*
emplace_back	constructs an element in-place at the end *(public member function)*
emplace	constructs element in-place *(public member function)*
push_front	inserts an element to the beginning *(public member function)*

unction	description
push_back	adds an element to the end *(public member function)*
pop_front	removes the first element *(public member function)*
pop_back	removes the last element *(public member function)*
insert	inserts elements *(public member function)*
erase	erases elements *(public member function)*
swap	swaps the contents *(public member function)*

Operations

function	description
select	selects the element at the specified location *(public member function)*

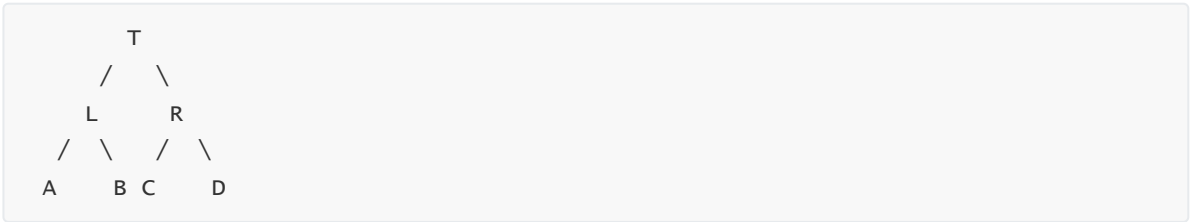
Implementation

Properties

An Array Based on Tree is a data structure based on a non-sorted binary tree and has the following properties:

- The size of the left child node is not less than the sizes of its two nephew nodes.
- The size of the right child node is not less than the sizes of its two nephew nodes.
- 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.



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

- size(L) >= max(size(C), size(D))
- size(R) >= max(size(A), size(B))

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.
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*;
    node_pointer       parent;
    node_pointer       left;
    node_pointer       right;
    size_t             size;
    T                  data;
};
```

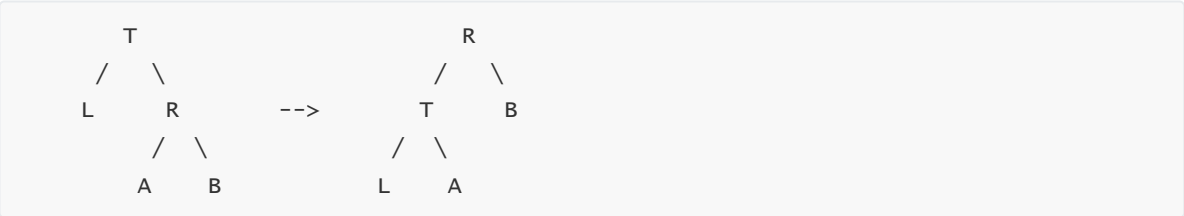
Rotations

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.

Left rotation

The left rotation operation is used to make the right subtree of a node its parent node, and to make the left subtree of its right subtree its right subtree. This operation makes the original node the left child node of its right child node, thereby maintaining the balance of the binary balancing tree.



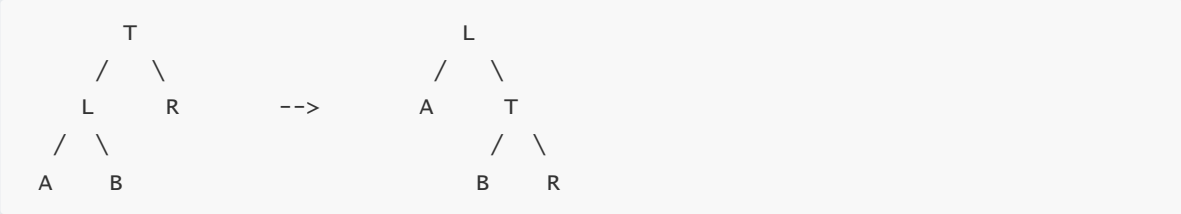
The C++ code for left rotation is as follows:

```
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;
}
```

Right rotation

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



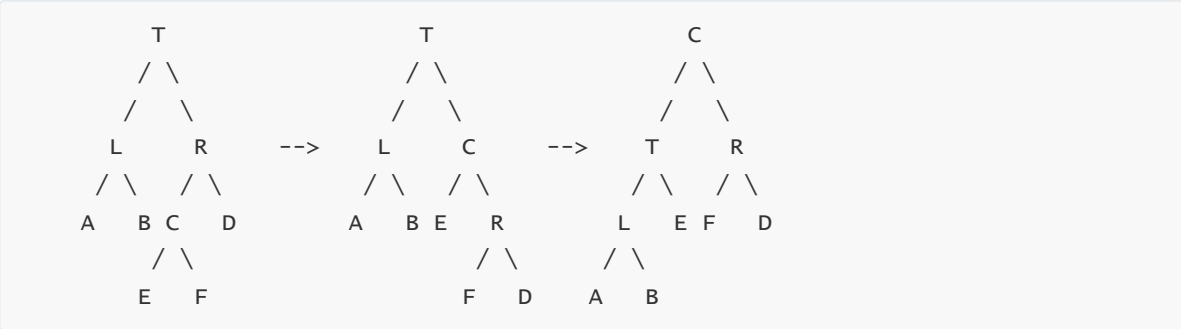
The C++ code for right rotation is as follows:

```
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;
}
```

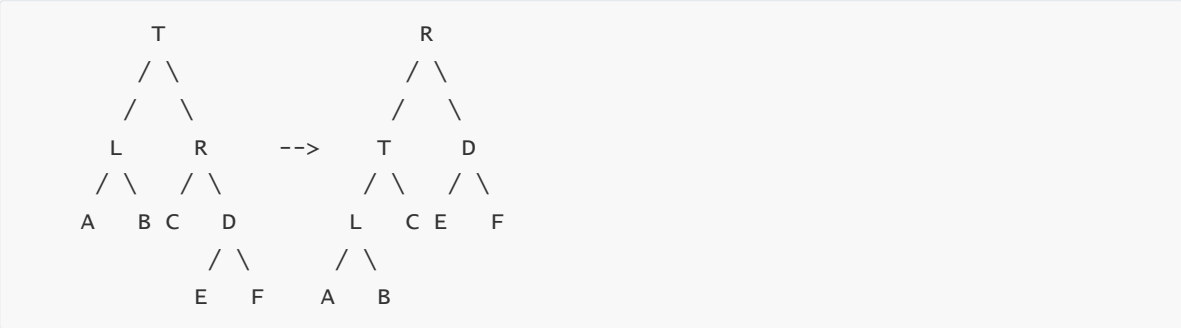
Rebalancing

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:

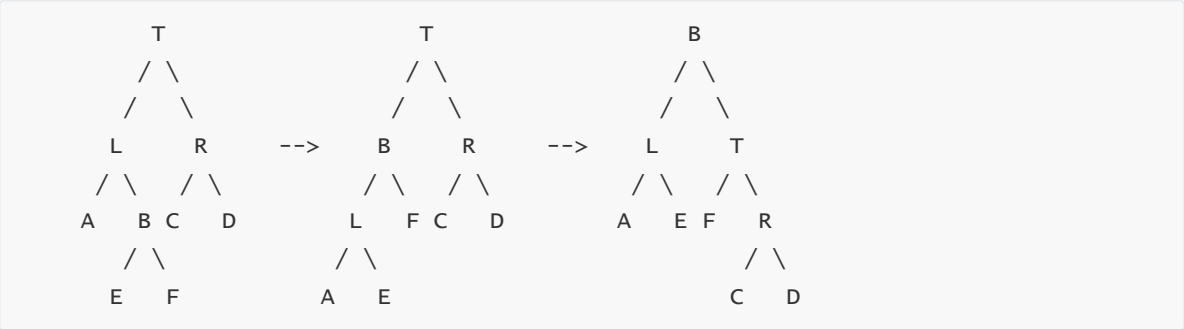
1. When size(T.left) < size(T.right.left), it may occur when inserting a node into the right child of T or deleting a node from the left child of T. First, perform a right rotation on the right child node of T, and then perform a left rotation on T. Now, the subtrees A, B, D, E, F, and L still satisfy 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.



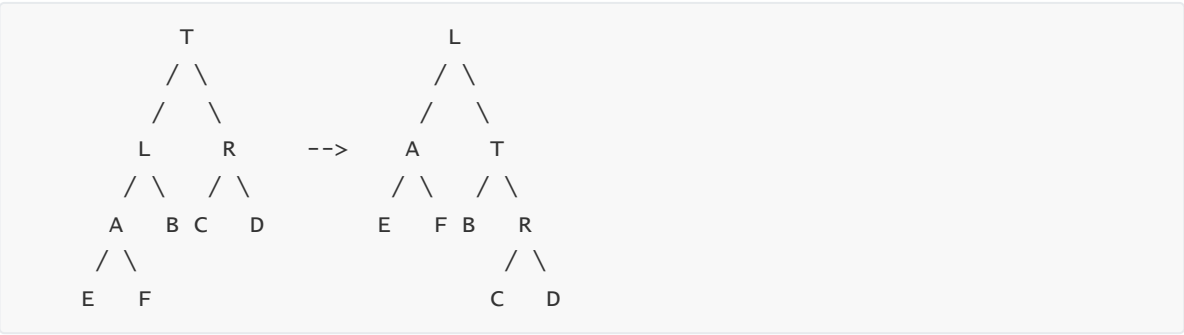
2. When size(T.left) < size(T.right.right), it may occur when inserting a node into the right child of T or deleting a node from the left child of T. After performing a left rotation on T, the subtree A, B, C, D, E, F, and L still satisfy the properties of ABT. However, due to the decrease in nodes of 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.



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.



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.



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)
            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:

```
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);
}
```

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;
}
```

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.
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.

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

```
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);
}
```

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;
}
```

Download

https://github.com/RulerCN/Array-Based-on-Tree

If you have any questions, please email: 26105499@qq.com