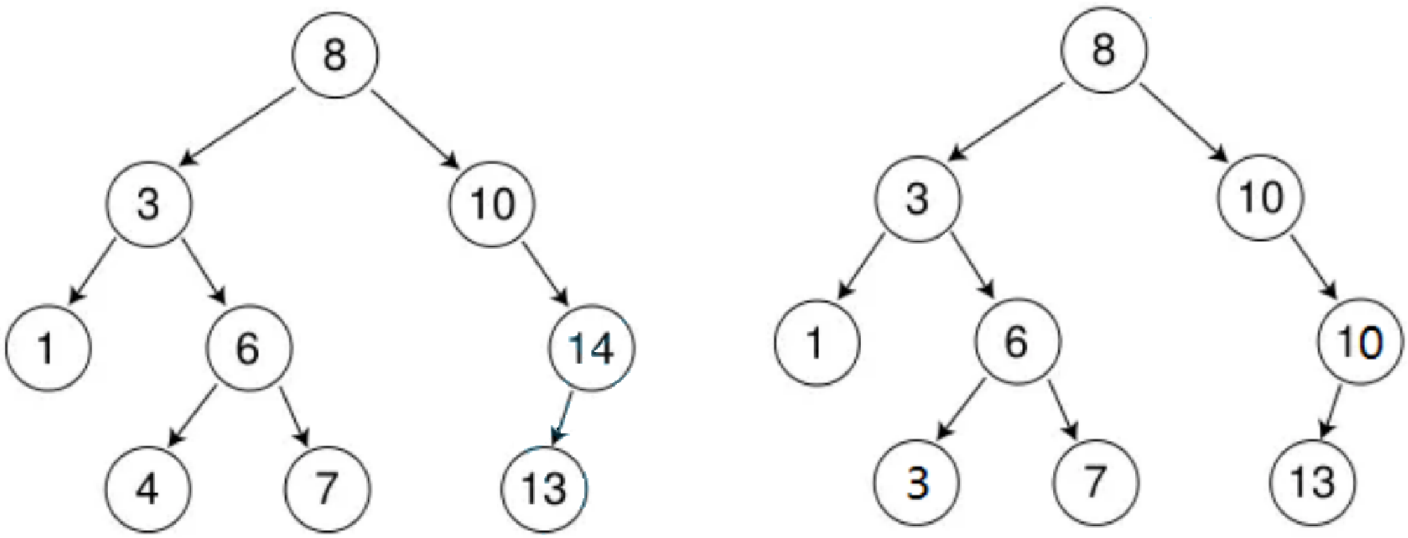
BSTree复习

1. 二叉搜索树的概念

二叉搜索树又称二叉排序树，它或是一颗空树，又或者是满足以下性质的二叉树：

- 左子树所有节点的值都小于根节点的值
- 右子树所有节点的值都大于根节点的值
- 左右子树也分别是二叉搜索树
- 二叉搜索树中可以支持插入相等的值，也可以不支持插入相等的值，具体看使用场景。map/set/multimap/multiset 系列的底层容器就是二叉搜索树，其中 map/set 不支持插入相等的值, multimap/multiset支持插入相等值。



2. 二叉搜索树的性能分析

最优情况下，二叉搜索树为完全二叉树(或者接近完全二叉树)，其高度为： log2 N

最差情况下，二叉搜索树退化为单支树(或者类似单支)，其高度为： N

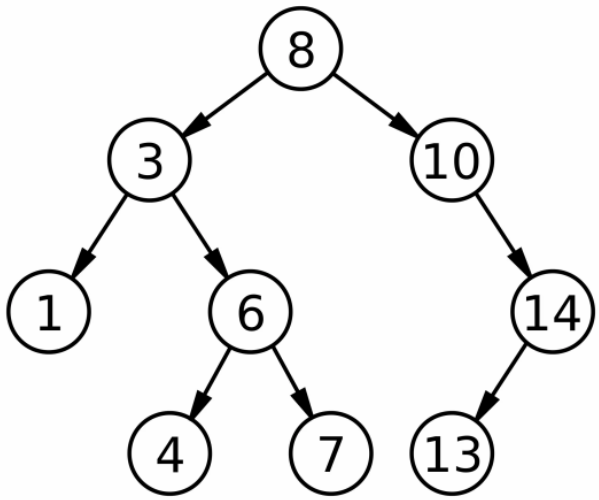
所以综合而言二叉搜索树增删查改时间复杂度为： $O(N)$

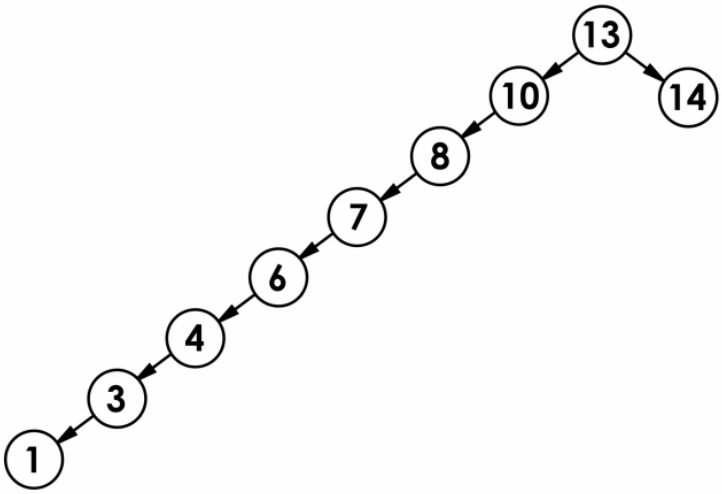
这样的效率是无法满足需求的，二叉树的变形AVLTree和RBTree才能适用于我们在内存中存储数据和搜索数据。

说明：二分查找也可以实现O(log2 N)级别的查找效率，但是二分查找有两大缺陷：

- 需要存储在支持下标访问的结构中，并且有序

- 插入和删除数据效率很低，因为存储在下标随机访问的结构中，插入和删除数据一般需要挪动数据。

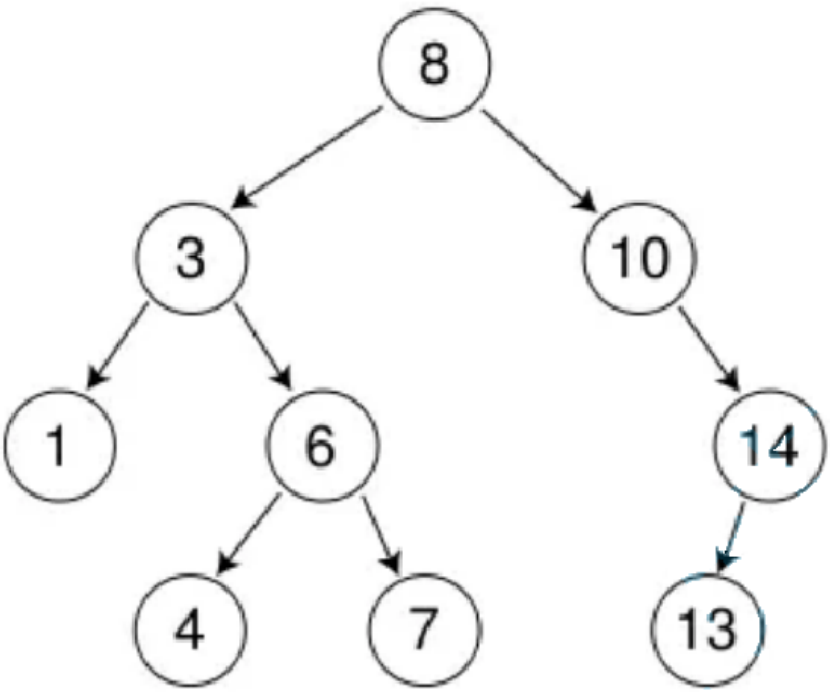


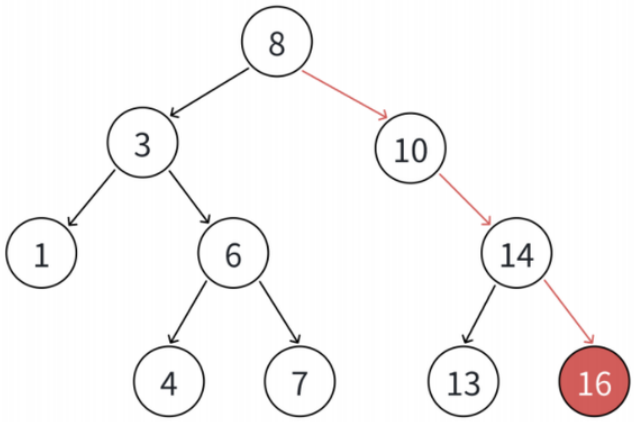


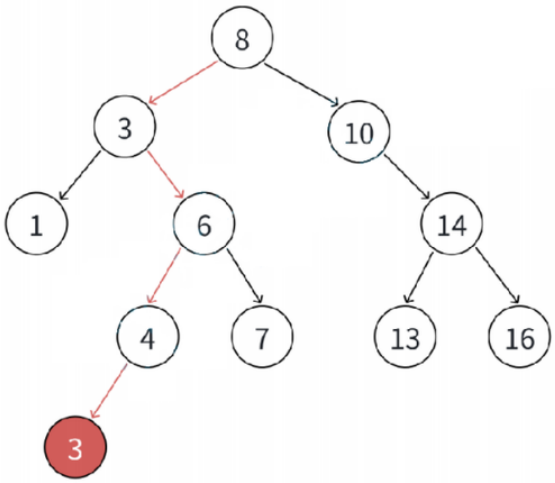
3. 二叉搜索树的插入

1. 树为空，则直接新增结点，赋值给root指针
2. 树不为空，按二叉搜索树的性质，插入值比当前结点大的往右走，小的往左走。
3. 如果支持插入相等的值，插入值跟当前结点相等的值可以往左走，也可以往右走，找到空位置，插入新结点。

（需要保持逻辑一致性，不要一会往右走，一会往左走）。

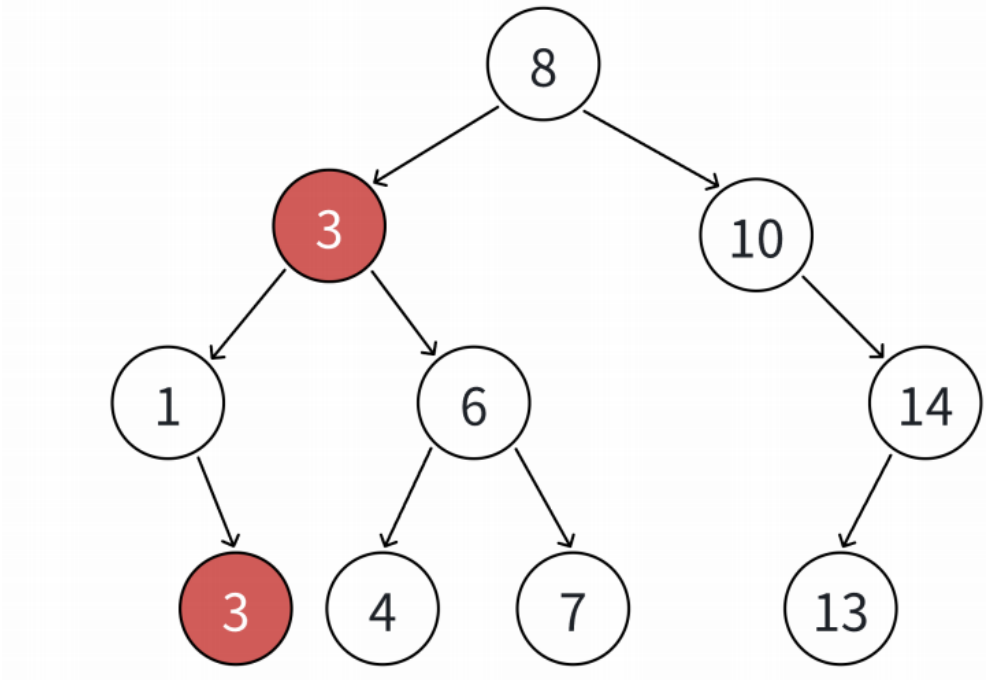






4. 二叉搜索树的查找

1. 从根开始比较，如果x比当前结点值大则往右查找，反之往左查找
2. 最多查找高度次，走到为空还没找到，则这哦值不存在
3. 如果不支持插入相等值，找到x即可返回
4. 如果支持插入相等值，一般来说要求查找中序的第一个x



5. 二叉搜索树的删除

首先查找该元素是否存在，不存在则返回false

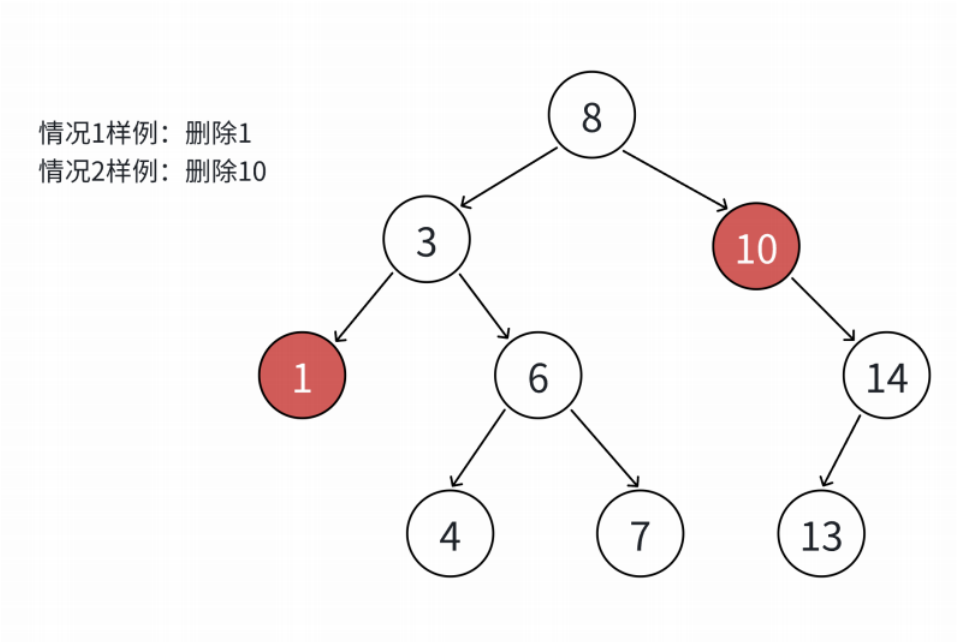
如果查找元素存在则分以下四种情况处理：

- 要删除结点N左右孩子均为空

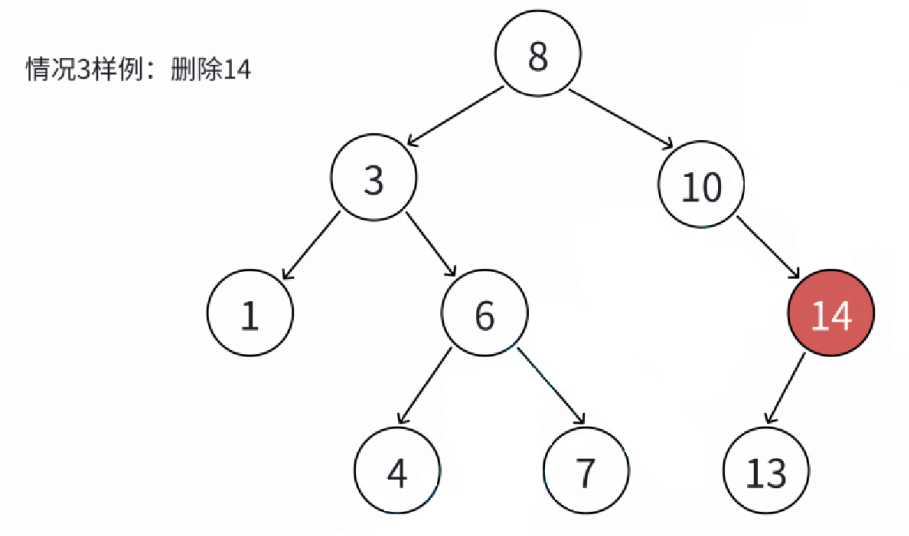
- 要删除结点N左孩子为空，右孩子不为空
- 要删除结点N右孩子为空，左孩子不为空
- 要删除结点N左右孩子均不为空

对应的解决方案：

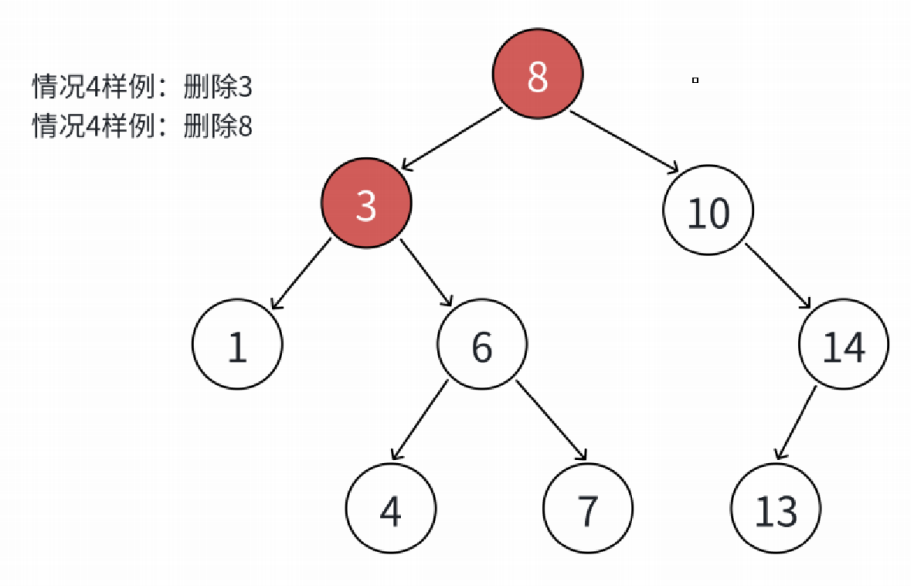
- 把N结点的父亲对应的孩子指针置为空，然后直接删除N结点
- 把N结点的父亲对应的孩子指针指向N的右孩子，直接删除N结点



- 把N结点的父亲对应的孩子指针指向N的左孩子，直接删除N结点



- 无法直接删除N结点，因为N结点的两个孩子结点无处安放，只能用替换法删除。找到N左子树的最大结点R(最右结点)或者N右子树的最小节点L(最左结点)替代N，因为这两个结点中的任意一个，放到N的位置都满足二叉搜索树的性质。让L或者R与N交换值，然后删除L或者R结点，R对应的情况也有可能是上述三种，再分情况删除。



6. 模拟二叉搜索树的实现

代码块

```cpp
#pragma once
#include<iostream>
#include<algorithm>

template<class K>
struct BSTNode
{
    K _key;
    BSTNode<K>* _left;
    BSTNode<K>* _right;

    BSTNode(K key)
        :_key(key)
        , _left(nullptr)
        , _right(nullptr)
    {}
};

// Binary Search Tree
template<class K>
class BSTree
{
        typedef BSTNode<K> Node;
public:
    bool Insert(const K& key)
    {
```

```
27          // 空树
28          if (_root == nullptr)
29          {
30                  _root = new Node(key);
31                  return true;
32          }
33
34          Node* cur = _root;
35          Node* parent = _root;
36          while (cur != nullptr)
37          {
38              parent = cur;
39              if (key > cur->_key) cur = cur->_right;
40              else if (key < cur->_key) cur = cur->_left;
41              else // 该版本不支持插入相等值
42              {
43                  std::cout << "该版本不支持插入相等值" << std::endl;
44                  return false;
45              }
46          }
47
48          cur = new Node(key);
49          if (key > parent->_key) parent->_right = cur;
50          else parent->_left = cur;
51
52          return true;
53      }
54
55      bool Find(const K& key)
56      {
57          Node* cur = _root;
58          while (cur != nullptr)
59          {
60              if (key > cur->_key) cur = cur->_right;
61              else if (key < cur->_key) cur = cur->_left;
62              else return true;
63          }
64
65          return false;
66      }
67
```

代码块

```
1    bool Erase(const K& key)
2   {
```

```
3       if (!Find(key)) return false;
4
5       Node* cur = _root;
6       Node* parent = _root;
7
8       while (cur != nullptr)
9       {
10          if (key > cur->_key)
11          {
12              parent = cur;
13              cur = cur->_right;
14          }
15
16          else if (key < cur->_key)
17          {
18              parent = cur;
19              cur = cur->_left;
20          }
21
22          else break;
23      }
24
25      // 对cur的孩子分四种情况讨论
26
27      // 4. 左右孩子均不为空，这里首先采用找左子树的最右结点来解决
28      if (cur->_left != nullptr && cur->_right != nullptr)
29      {
30          if (_root->_left != nullptr)
31          {
32              Node* R = _root->_left;
33              while (R->_right != nullptr)
34              {
35                      parent = R;
36                      R = R->_right;
37              }
38
39              std::swap(R->_key, cur->_key);
40              cur = R;
41          }
42
43          else if (_root->_left == nullptr && _root->_right != nullptr)
44          {
45              Node* L = _root->_left;
46              while (L->_left != nullptr)
47              {
48                      parent = L;
49                      L = L->_left;
```

```
            }

            std::swap(L->_key, cur->_key);
            cur = L;
        }
    }

    // 1. cur无孩子
    if (cur->_left == nullptr && cur->_right == nullptr)
    {
        if (parent->_left == cur) parent->_left = nullptr;
        else parent->_right = nullptr;

        delete cur;
        cur = nullptr;
    }

    // 2. cur有右孩子
    else if (cur->_right != nullptr && cur->_left == nullptr)
    {
        if (parent->_left == cur) parent->_left = cur->_right;
        else parent->_right = cur->_right;

        delete cur;
        cur = nullptr;
    }

    // 3. cur有左孩子
    else if (cur->_left != nullptr && cur->_right == nullptr)
    {
        if (parent->_left == cur) parent->_left = cur->_left;
        else parent->_right = cur->_left;

        delete cur;
        cur = nullptr;
    }

    return true;
```

代码块

```
void InOrder()
    {
        if (_root == nullptr) return;
        _InOrder(_root);
        std::cout << std::endl;
    }
```

```
7
8   private:
9       void _InOrder(Node* root)
10      {
11          if (root == nullptr) return;
12          std::cout << root->_key << " ";
13          _InOrder(root->_left);
14          _InOrder(root->_right);
15      }
16
17  private:
18      Node * _root = nullptr;
19  };
```

7. 二叉搜索树key和key/value使用场景

7.1 key搜索场景

key作为关键码存储在结构中，搜索时通过搜索关键码key得到需要搜索的值（也就是key），搜索场景只需要判断key是否存在。key的搜索场景实现的二叉搜索树支持增删查，但是不支持改，修改key值会破坏搜索树的结构。

7.2 key/value搜索场景

每一个关键码key都有一个对应的值value，value可以是任何类型对象。树的结构中（结点）除了需要存储key值，还要存储对应的value值，增/删/查都是以key值在树的规则上进行比较，可以快速查找到key对应的value。key/value的搜索场景实现的二叉搜索树支持修改，但是不能修改key值，因为这样会破坏树的性质。

7.3 key/value版本BSTree模拟实现

代码块

```
1   #pragma once
2   #include<iostream>
3
4   template <class K, class V>
5   struct BSTNode
6   {
7       K _key;
8       V _value;
```

```
    BSTNode<K, V>* _left;
    BSTNode<K, V>* _right;
    BSTNode(const K & key, const V & value)
        :_key(key)
        , _value(value)
        , _left(nullptr)
        , _right(nullptr)
    {}
};

template<class K, class V>
class BSTree
{
    typedef BSTNode<K, V> Node;
public:
    BSTree() = default;

    BSTree(const BSTree<K, V>& t) { _root = Copy(t); }

    BSTree<K, V>& operator=(BSTree<K, V> t)
    {
        std::swap(_root, t._root);
        return *this;
    }

    ~BSTree() { Destroy(_root); }

    bool Insert(const K& key, const V& value)
    {
        // 空树
        if (_root == nullptr)
        {
            _root = new Node(key, value);
            return true;
        }

        Node* cur = _root;
        Node* parent = _root;
        while (cur != nullptr)
        {
            parent = cur;
            if (key > cur->_key) cur = cur->_right;
            else if (key < cur->_key) cur = cur->_left;
            else
            {
                std::cout << "该版本不支持插入相等值" << std::endl;
                return false;
```

```
            }
        }

        cur = new Node(key, value);
        if (key > parent->_key) parent->_right = cur;
        else parent->_left = cur;

        return true;
    }

    Node* Find(const K& key)
    {
        Node* cur = _root;
        while (cur != nullptr)
        {
            if (key > cur->_key) cur = cur->_right;
            else if (key < cur->_key) cur = cur->_left;
            else return true;
        }

        return false;
    }


```

代码块

```
bool Erase(const K& key)
    {
        if (!Find(key)) return false;
        Node* cur = _root;
        Node* parent = _root;

        while (cur != nullptr)
        {
            if (key > cur->_key)
            {
                parent = cur;
                cur = cur->_right;
            }
            else if (key < cur->_key)
            {
                parent = cur;
                cur = cur->_left;
            }
            else break;
```

```
20          }
21
22          // 对cur的孩子分四种情况讨论
23          // 4. 左右孩子均不为空，这里首先采用找左子树的最右结点来解决
24          if (cur->_left != nullptr && cur->_right != nullptr)
25          {
26              if (_root->_left != nullptr)
27              {
28                  Node* R = _root->_left;
29                  while (R->_right != nullptr)
30                  {
31                      parent = R;
32                      R = R->_right;
33                  }
34
35                  std::swap(R->_key, cur->_key);
36                  std::swap(R->_value, cur->_value);
37
38                  cur = R;
39              }
40
41              else if (_root->_left == nullptr && _root->_right != nullptr)
42              {
43                  Node* L = _root->_left;
44                  while (L->_left != nullptr)
45                  {
46                      parent = L;
47                      L = L->_left;
48                  }
49
50                  std::swap(L->_key, cur->_key);
51                  std::swap(L->_value, cur->_value);
52                  cur = L;
53              }
54          }
55          // 1. cur无孩子
56          if (cur->_left == nullptr && cur->_right == nullptr)
57          {
58              if (parent->_left == cur) parent->_left = nullptr;
59              else parent->_right = nullptr;
60
61              delete cur;
62              cur = nullptr;
63          }
64
65          // 2. cur有右孩子
66          else if (cur->_right != nullptr && cur->_left == nullptr)
```

```
67          {
68              if (parent->_left == cur) parent->_left = cur->_right;
69              else parent->_right = cur->_right;
70
71              delete cur;
72              cur = nullptr;
73          }
74
75          // 3. cur有左孩子
76          else if (cur->_left != nullptr && cur->_right == nullptr)
77          {
78              if (parent->_left == cur) parent->_left = cur->_left;
79              else parent->_right = cur->_left;
80
81              delete cur;
82              cur = nullptr;
83          }
84          return true;
85      }
86
87      void InOrder() { _InOrder(_root); }
```

代码块

```
1   private:
2       void _InOrder(Node* root)
3       {
4           if (root == nullptr) return;
5
6           _InOrder(root->_left);
7           cout << root->_key << ":" << root->_value << endl;
8           _InOrder(root->_right);
9
10          std::cout << std::endl;
11      }
12      void Destroy(Node* root)
13      {
14          if (root == nullptr) return;
15          Destroy(root->_left);
16          Destroy(root->_right);
17
18          delete root;
19          root = nullptr;
20      }
21
22      Node* Copy(Node* root)
23      {
```

```
24          if (root == nullptr) return;
25
26          Node* newnode = new Node(root->_key, root->_value);
27          newnode->_left = Copy(root->_left);
28          newnode->_right = Copy(root->_right);
29
30          return newnode;
31      }
32
33  private:
34      Node* _root = nullptr;
35  };
```