二叉搜索树

二叉搜索树具有以下特性：

1. 每个节点都包含（也可能包含其他的数据）一个特定的被称为键的值，它定义了节点的顺序。
2. 键值是唯一的，也就是说任何键值在树中只能出现一次。
3. 树中的每个节点，其键值必须大于以它左孩子节点为根节点的子树的所有节点的键值，一定小于以它右孩子节点为根节点的子树的所有节点的键值。

如果BST满足特性，则 `get()` 操作的复杂度为 `O(logN)` 的

```
export module BST;
import std;
using namespace std;

export template <typename KeyType, typename ValueType>
class BST {
private:
    struct Node {
        KeyType key;
        ValueType val;
        Node* left{};
        Node* right{};

        explicit Node(KeyType k, ValueType v) : key(k), val(v) {};
    };

    Node* root{};


    BST(const BST<KeyType, ValueType>& rhs) = delete;
    BST<KeyType, ValueType>& operator=(const BST<KeyType, ValueType>& src) =
delete;
public:

    BST() = default;

    ~BST() {
        while (root != nullptr) {
            root = deleteMinNode(root);
        }
    }

    void deleteMinNode() {
        root = deleteMinNode(root);
    }

    Node* findMinNode() {
        return findMinNode(root);
    }
```

```
39
40
41      void put(KeyType&& key, ValueType&& val) {
42          insertNode(root, std::forward<KeyType>(key), std::forward<ValueType>
    (val));
43      }
44      ValueType get(KeyType&& key) {
45          auto&& t = findNode(root, std::forward<KeyType&&>(key));
46          if (t == nullptr) throw std::out_of_range("GET: NOT FOUND IT.");
47          return t->val;
48      }
49      void show() {
50          displayTree(root);
51      }
52
53      void deleteNode(KeyType&& key) {
54          root = deleteNode(root, std::forward<KeyType>(key));
55      }
56
57  private:
58
59      Node* findMinNode(Node* t) {
60          if (t->left == nullptr) {
61              return t;
62          }
63          else {
64              findMinNode(t->left);
65          }
66      }
67
68      Node* deleteMinNode(Node* t) {
69          if (t->left == nullptr) {
70              Node* temp = t;
71              t = t->right;
72              delete temp;
73          }
74          else {
75              t->left = deleteMinNode(t->left);
76          }
77          return t;
78      }
79
80      Node* deleteNode(Node* t, KeyType&& key) {
81
82          Node* x = findNode(t, std::forward<KeyType>(key));
83          Node* temp{};
84          if (x->right == nullptr) {
85              temp = x;
86              x = x->left;
87          }
88          else if (x->left == nullptr) {
89              temp = x;
```

```
            x = x->right;
        }
        else {
            temp = findMinNode(x->right);     // 右子树中最小的左子树
            x->right = deleteMinNode(temp->right);  // 在temp的右子树中删除x结点本身，并且让x的右子树指向temp的右子树
            x->left = temp->left;                   // 左子树不变。
        }
        delete temp;
        return x;
    }



    void insertNode(Node*& t, KeyType&& key, ValueType&& val) {
        if (t == nullptr) {
            t = new Node(key, val);
        }
        else {
            if (key != t->key) {
                if (key < t->key) {
                    insertNode(t->left, std::forward<KeyType>(key), std::forward<ValueType>(val));
                }
                else {
                    insertNode(t->right, std::forward<KeyType>(key), std::forward<ValueType>(val));
                }
            }
        }
    }

    Node* findNode(Node* t, KeyType&& key) {
        if (t == nullptr)   return nullptr;
        if (key == t->key)  return t;
        if (key < t->key) { return findNode(t->left, std::forward<KeyType&&>(key)); }
        else { return findNode(t->right, std::forward<KeyType&&>(key)); }
    }

    void displayTree(Node* t) {
        if (t != nullptr) {
            displayTree(t->left);
            cout << t->key << " : " << t->val << endl;
            displayTree(t->right);
        }
    }

};
```

GET(FIND)

迭代写法

```
    Node* findNode(Node *t, KeyType &&key) {
        while (t != nullptr) {
            if (t->key == key) return t;
            else if (t->key > key) {
                t = t->left;
            } else {
                t = t->right;
            }
        }
        return nullptr;
    }

```

由于get的次数远大于put，所以说get实现一个迭代版本很好。

递归写法

```
    Node* findNode(Node *t, KeyType &&key) {
        if (t == nullptr)   return nullptr;
        if (key == t->key)  return t;
        if (key < t->key) { return findNode(t->left, std::forward<KeyType&&>(key)); }
        else {              return findNode(t->right, std::forward<KeyType&&>(key));}
    }
```

PUT(INSERT)

```
    void insertNode(Node*& t, KeyType&& key, ValueType&& val) {
        if (t == nullptr) {
            t = new Node(key, val);
        }
        else {
            if (key != t->key) {
                if (key < t->key) {
                    insertNode(t->left,
                               std::forward<KeyType>(key),
                               std::forward<ValueType>(val));
                }
                else {
                    insertNode(t->right,
                               std::forward<KeyType>(key),
                               std::forward<ValueType>(val));
                }
            }
        }
    }
```

DELETE(REMOVE)

查找最小的结点

如果根结点的左子树为空，则BST中最小的键就是根结点；如果左子树非空，则BST中最小的结点是最左的左子树。

- 向左优先
- 没有左子树的最左结点

```
Node* minNode(Node* x) {
    if (x→left == nullptr)
        return x;
    return minNode(x→left);
}
```

删除最小的结点

```
Node* deleteMinNode(Node* x) {
    if (x→left == nullptr)
        Node *r = x→right;
        delete x;
        return r;

    x→left = deleteMinNode(x→left);
    return x;
}
```

删除任意的结点

1. 将指向即将被删除的结点的链接保存为 `temp` ；
2. 将 x 指向它的后继结点 `min(temp.right)` ；
3. 将 x 的右链接（原本指向一棵所有结点都大于 x.key 的二叉查找树）指向 `deleteMin(t.right)` ，也就是在删除后所有结点仍然都大于 x.key 的子二叉查找树；
4. 将 x 的左链接（本为空）设为 t.left（其下所有的键都小于被删除的结点和它的后继结点）。*/

```
    Node* deleteNode(Node* t, KeyType&& key) {

        Node* x = findNode(t, std::forward<KeyType>(key));
        Node* temp{};
        if (x→right == nullptr) {
            temp = x;
            x = x→left;
        }
        else if (x→left == nullptr) {
            temp = x;
```

```
            x = x→right;
        }
        else {
            temp = findMinNode(x→right);      // 右子树中最小的左子树
            x→right = deleteMinNode(temp→right);  // 在temp的右子树中删除x结点本身，并且让x的右子树指向temp的右子树
            x→left = temp→left;                   // 左子树不变。
        }
        delete temp;
        return x;
    }
```

遍历二叉树（下列是中序变量--以根记）

```
    void displayTree(Node *t) {
        if (t != nullptr) {
            displayTree(t->left);
            cout << t->key << endl;
            displayTree(t->right);
        }
    }

    void show() {
        displayTree(root);
    }
```