

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

```
1  export module BST;
2  import std;
3  using namespace std;
4
5  export template <typename KeyType, typename ValueType>
6  class BST {
7  private:
8      struct Node {
9          KeyType key;
10         ValueType val;
11         Node* left{};
12         Node* right{};
13
14         explicit Node(KeyType k, ValueType v) : key(k), val(v) {};
15     };
16
17     Node* root{};
18
19
20     BST(const BST<KeyType, ValueType>& rhs) = delete;
21     BST<KeyType, ValueType>& operator=(const BST<KeyType, ValueType>& src) =
delete;
22 public:
23
24     BST() = default;
25
26     ~BST() {
27         while (root != nullptr) {
28             root = deleteMinNode(root);
29         }
30     }
31
32     void deleteMinNode() {
33         root = deleteMinNode(root);
34     }
35
36     Node* findMinNode() {
37         return findMinNode(root);
38     }
```

```

39
40
41     void put(KeyType&& key, ValueType&& val) {
42         insertNode(root, std::forward<KeyType>(key), std::forward<ValueType>
43         (val));
44     }
45     ValueType get(KeyType&& key) {
46         auto&& t = findNode(root, std::forward<KeyType&&>(key));
47         if (t == nullptr) throw std::out_of_range("GET: NOT FOUND IT.");
48         return t->val;
49     }
50     void show() {
51         displayTree(root);
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;

```

```

90         x = x→right;
91     }
92     else {
93         temp = findMinNode(x→right);    // 右子树中最小的左子树
94         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
95         x→left = temp→left;            // 左子树不变。
96     }
97     delete temp;
98     return x;
99 }
100
101
102
103 void insertNode(Node*& t, KeyType&& key, ValueType&& val) {
104     if (t == nullptr) {
105         t = new Node(key, val);
106     }
107     else {
108         if (key != t→key) {
109             if (key < t→key) {
110                 insertNode(t→left, std::forward<KeyType>(key),
std::forward<ValueType>(val));
111             }
112             else {
113                 insertNode(t→right, std::forward<KeyType>(key),
std::forward<ValueType>(val));
114             }
115         }
116     }
117 }
118
119 Node* findNode(Node* t, KeyType&& key) {
120     if (t == nullptr) return nullptr;
121     if (key == t→key) return t;
122     if (key < t→key) { return findNode(t→left, std::forward<KeyType&&>
(key)); }
123     else { return findNode(t→right, std::forward<KeyType&&>(key)); }
124 }
125
126 void displayTree(Node* t) {
127     if (t != nullptr) {
128         displayTree(t→left);
129         cout << t→key << " : " << t→val << endl;
130         displayTree(t→right);
131     }
132 }
133
134 };

```

GET(FIND)

迭代写法

```
1 Node* findNode(Node *t, KeyType &&key) {
2     while (t != nullptr) {
3         if (t->key == key) return t;
4         else if (t->key > key) {
5             t = t->left;
6         } else {
7             t = t->right;
8         }
9     }
10    return nullptr;
11 }
12
```

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

```
1 Node* findNode(Node *t, KeyType &&key) {
2     if (t == nullptr) return nullptr;
3     if (key == t->key) return t;
4     if (key < t->key) { return findNode(t->left, std::forward<KeyType&&>
5 (key)); }
6     else { return findNode(t->right, std::forward<KeyType&&>
7 (key)); }
8 }
```

PUT(ININSERT)

```
1 void insertNode(Node*& t, KeyType&& key, ValueType&& val) {
2     if (t == nullptr) {
3         t = new Node(key, val);
4     }
5     else {
6         if (key != t->key) {
7             if (key < t->key) {
8                 insertNode(t->left,
9                             std::forward<KeyType>(key),
10                            std::forward<ValueType>(val));
11             }
12             else {
13                 insertNode(t->right,
14                             std::forward<KeyType>(key),
15                            std::forward<ValueType>(val));
16             }
17         }
18     }
19 }
```

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

```
1 Node* minNode(Node* x) {
2     if (x->left == nullptr)
3         return x;
4     return minNode(x->left);
5 }
```

删除最小的结点

```
1 Node* deleteMinNode(Node* x) {
2     if (x->left == nullptr)
3         Node *r = x->right;
4         delete x;
5         return r;
6
7     x->left = deleteMinNode(x->left);
8     return x;
9 }
```

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）。*/

```
1 Node* deleteNode(Node* t, KeyType&& key) {
2
3     Node* x = findNode(t, std::forward<KeyType>(key));
4     Node* temp{};
5     if (x->right == nullptr) {
6         temp = x;
7         x = x->left;
8     }
9     else if (x->left == nullptr) {
10        temp = x;
```

```
11         x = x→right;
12     }
13     else {
14         temp = findMinNode(x→right);    // 右子树中最小的左子树
15         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
16         x→left = temp→left;            // 左子树不变。
17     }
18     delete temp;
19     return x;
20 }
```

遍历二叉树(下列是中序变量--以根记)

```
1 void displayTree(Node *t) {
2     if (t != nullptr) {
3         displayTree(t->left);
4         cout << t->key << endl;
5         displayTree(t->right);
6     }
7 }
8
9 void show() {
10     displayTree(root);
11 }
```