二叉排序树

宋星霖

2024 年 2 月 9 日

1 结构讲解

二叉排序树，又名二叉搜索树，是一种特殊的二叉树。它的特性是左子树 < 根节点，右子树 > 根节点，并且它的中序遍历是有序的。它可以处理与排名相关的检索需求。

2 二叉排序树的插入

根据二叉排序树的性质，我们可以先将待插入节点与当前根节点比较，如果比当前根节点小，插入到左子树；如果比当前根节点大，插入到右子树；如果当前根节点为 nullptr，将待插入节点插入在此。

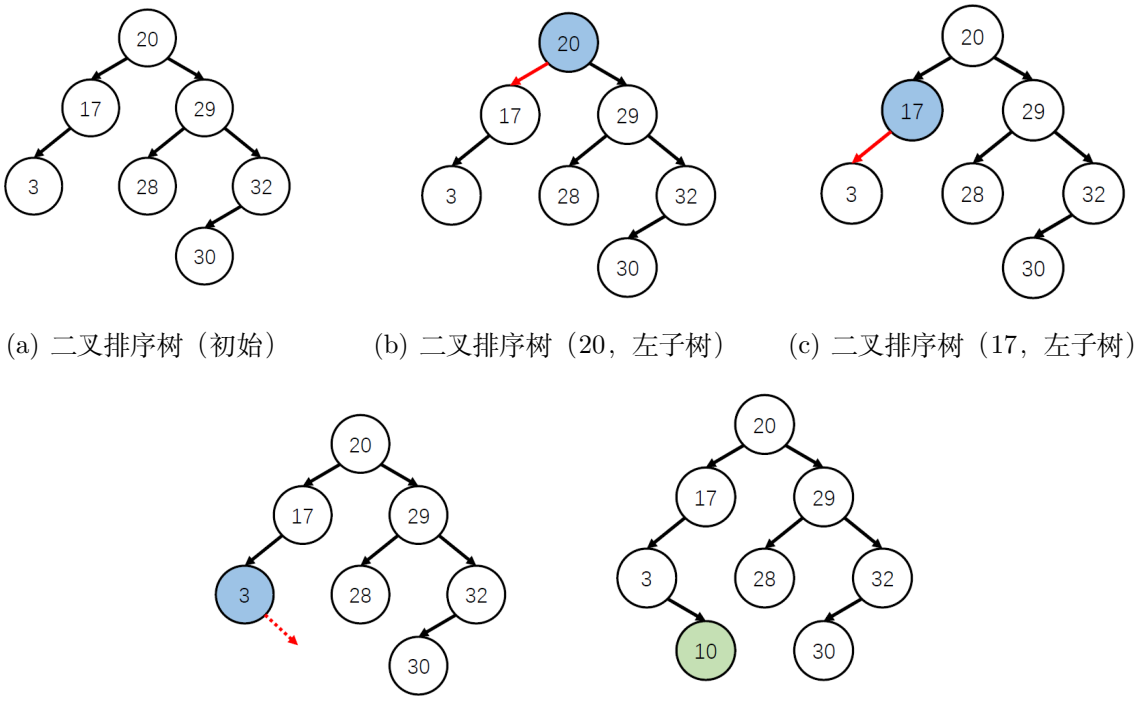
演示：

二叉排序树如图 1(a) 所示

现在我们要插入节点 10，我们进行如下操作：拿节点 10 与二叉排序树根节点 20 比较，因为 $10 < 20$，所以节点 10 应该插入到 20 的左子树中，如图 1(b) 所示。

接下来，我们拿节点 10 与节点 17 进行比较，因为 $10 < 17$，所以节点 10 应该插入到 17 的左子树中，如图 1(c) 所示。

之后，我们拿节点 10 与节点 3 比较，因为 $10 > 3$，所以节点 10 因该插入到节点 3 的右子树中，但是 3 的右子树是空树，所以节点 10 成为了 3 的右孩子，如图 1(d) 和 1(e) 所示。



(a) 二叉排序树（初始）　(b) 二叉排序树（20，左子树）　(c) 二叉排序树（17，左子树）

(d) 二叉排序树（3，右子树）　(e) 二叉排序树（完成）

图 1: 二叉排序树（插入）

3　二叉排序树的删除

跟二叉排序树的插入差不多，待删除结点如果比当前根节点小，到左子树删除待删除结点；如果比当前根节点大，到右子树删除待删除结点；如果等于当前根节点，删除当前根节点。；如果当前根节点为 nullptr，说明待删除结点不存在。

删除当前根节点有三种情况：

- 当前节点为叶子节点：直接删除。
- 当前节点只有一棵子树：删除当前节点，提升唯一子树，如图2所示
- 当前节点有两颗子树：将当前节点与它的 前驱[1]/后继[2] 调换，之后删除当前节点的 前驱/后继。[3]，如图组 3 所示。

[1]前驱指当前二叉排序树的中序遍历中当前节点的前一个结点。

[2]后继指当前二叉排序树的中序遍历中当前节点的后一个结点。

[3]当前节点的前驱和后继是当前节点左子树的最大值和当前节点右子树的最小值，所以这两个节点只有 1 咳或 0 颗子树，是前面两种情况。

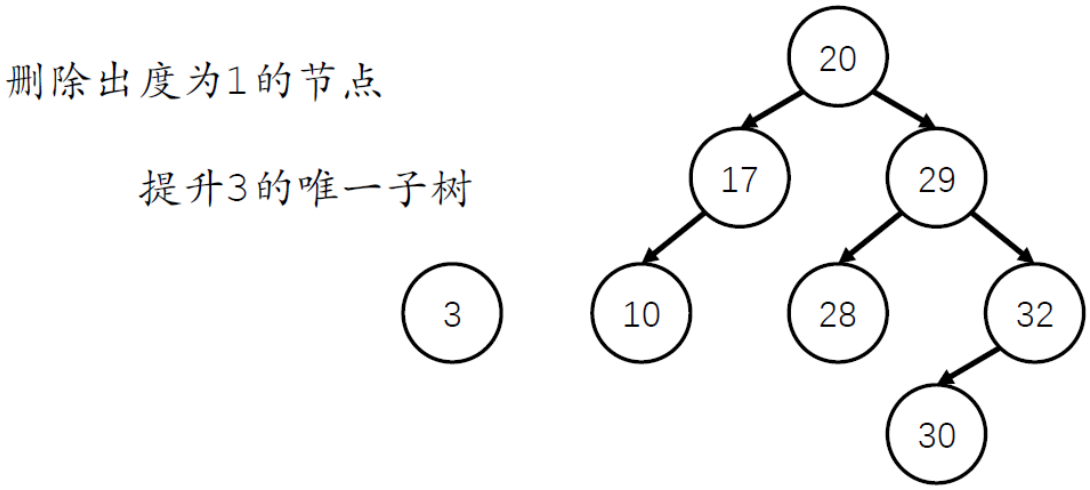
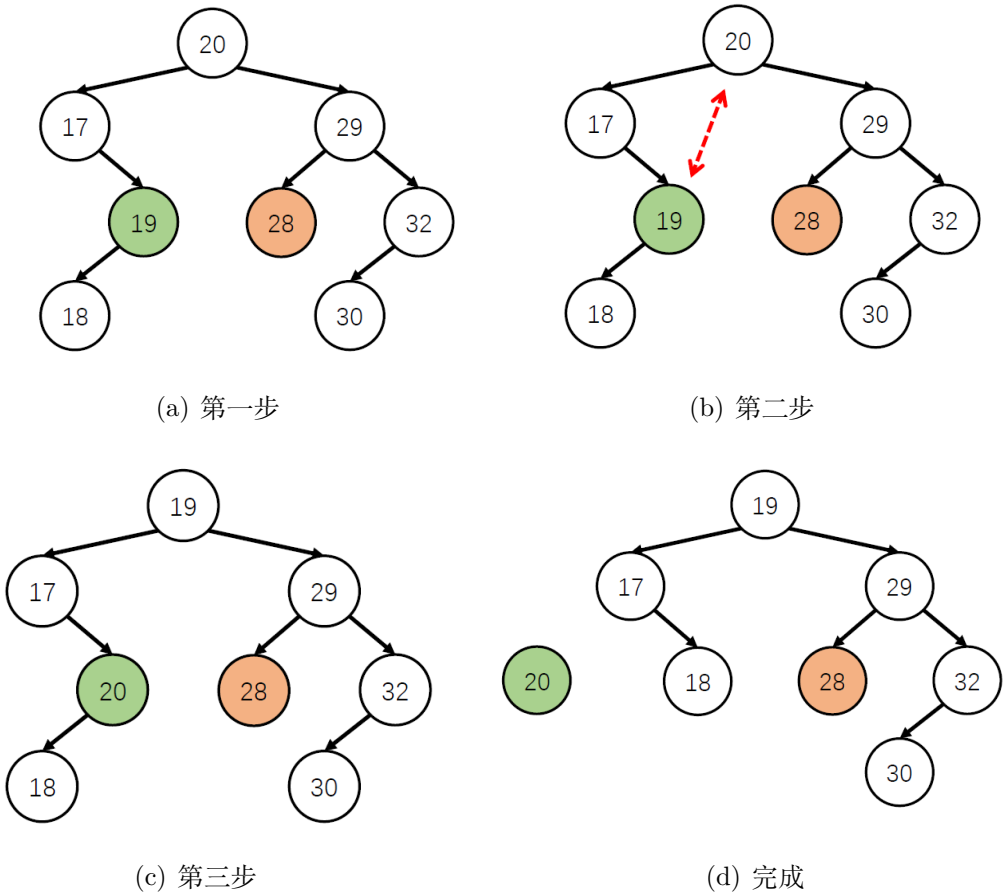


图 2: 删除只有一棵子树的节点



(a) 第一步　(b) 第二步

(c) 第三步　(d) 完成

图 3: 删除有两颗子树的节点

4 代码演示[4]

Listing 1: binary_search_tree.cpp

```cpp
#include <bits/stdc++.h>
#include <tools.hpp>
#define KEY(n) (n ? n->mKey : -1)
using namespace std;
class BinarySearchTree {
private:
    class TreeNode {
    public:
        int mKey{};
        TreeNode *mLeft{}, *mRight{};
        TreeNode ();
        TreeNode (int key);
        ~TreeNode ();
    };
    TreeNode* mRoot;
    TreeNode* insertT (TreeNode* root, int key);
    TreeNode* eraseT (TreeNode* root, int key);
    TreeNode* predecessor (TreeNode* root);
    void outputT (TreeNode* root);
    void inOrderT (TreeNode* root);

public:
    BinarySearchTree ();
    ~BinarySearchTree ();
    void insert (int key);
    void erase (int key);
    void output ();
    void inOrder ();
};
BinarySearchTree::TreeNode::TreeNode ()
    : mKey (0), mLeft (nullptr), mRight (nullptr) {}
BinarySearchTree::TreeNode::TreeNode (int key)
    : mKey (key), mLeft (nullptr), mRight (nullptr) {}
BinarySearchTree::TreeNode::~TreeNode () {
    delete mLeft;
    delete mRight;
}
BinarySearchTree::BinarySearchTree () {
    mRoot = nullptr;
}
BinarySearchTree::~BinarySearchTree () {
    delete mRoot;
}
void BinarySearchTree::insert (int key) {
```

[4]tools.h 头文件是 get_rand<>() 的头文件，作用是生成更好的随机数

```
45      mRoot = insertT (mRoot, key);
46  }
47  void BinarySearchTree::erase (int key) {
48      mRoot = eraseT (mRoot, key);
49  }
50  BinarySearchTree::TreeNode*
51  BinarySearchTree::insertT (TreeNode* root,          // NOLINT
52                             int key) {
53      if (root == nullptr)
54          return new TreeNode (key);
55      if (key == root->mKey)
56          return root;
57      if (key < root->mKey)
58          root->mLeft = insertT (root->mLeft, key);
59      else
60          root->mRight = insertT (root->mRight, key);
61      return root;
62  }
63  BinarySearchTree::TreeNode*
64  BinarySearchTree::eraseT (TreeNode* root,          // NOLINT
65                            int key) {
66      if (root == nullptr)
67          return root;
68      if (key < root->mKey)
69          root->mLeft = eraseT (root->mLeft, key);
70      else if (key > root->mKey)
71          root->mRight = eraseT (root->mRight, key);
72      else {
73          if (root->mLeft == nullptr && root->mRight == nullptr) {
74              delete root;
75              return nullptr;
76          } else if (root->mLeft == nullptr
77                  || root->mRight == nullptr) {
78              TreeNode* temp = root->mLeft ? root->mLeft : root->mRight;
79              free (root);
80              return temp;
81          } else {
82              TreeNode* temp = predecessor (root);
83              root->mKey = temp->mKey;
84              root->mLeft = eraseT (root->mLeft, temp->mKey);
85          }
86      }
87      return root;
88  }
89  BinarySearchTree::TreeNode*
90  BinarySearchTree::predecessor (TreeNode* root) {          // NOLINT
91      TreeNode* temp = root->mLeft;
92      while (temp->mRight)
93          temp = temp->mRight;
94      return temp;
```

```
95  }
96  void BinarySearchTree::outputT (TreeNode* root) {          // NOLINT
97      if (root == nullptr)
98          return;
99      printf ("(%d : %d : %d)\n", KEY (root), KEY (root->mLeft),
100             KEY (root->mRight));
101     outputT (root->mLeft);
102     outputT (root->mRight);
103 }
104 void BinarySearchTree::output () {
105     outputT (mRoot);
106 }
107 void BinarySearchTree::inOrderT (TreeNode* root) {          // NOLINT
108     if (root == nullptr)
109         return;
110     inOrderT (root->mLeft);
111     printf ("%d ", root->mKey);
112     inOrderT (root->mRight);
113 }
114 void BinarySearchTree::inOrder () {
115     inOrderT (mRoot);
116 }
117 int main () {
118 #define MAX_OP 10
119     BinarySearchTree* tree = new BinarySearchTree;
120     for (int i = 0; i < MAX_OP; i++) {
121         int key = get_rand<int> (0, 100);
122         printf ("insert key %d to BST\n", key);
123         tree->insert (key);
124     }
125     tree->output ();
126     printf ("in order : ");
127     tree->inOrder ();
128     puts ("");
129     int x;
130     while (scanf ("%d", &x)) {
131         if (x == -1)
132             break;
133         printf ("erase %d from BST\n", x);
134         tree->erase (x);
135         tree->inOrder ();
136         puts ("");
137     }
138     delete tree;
139     return 0;
140 }
```