AVL 树

宋星霖

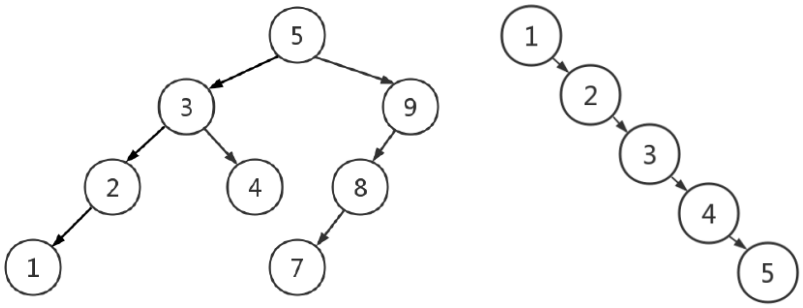
2024 年 2 月 10 日

1 平衡二叉排序树定义

在讲解平衡二叉排序树之前，请读者看一组示例：

> 1：[5, 9, 8, 3, 2, 4, 1, 7]
> 2：[1, 2, 3, 4, 5]

现在，请读者用这两组数据构造两颗棵二叉排序树。
若不出意外，构造出来的二叉排序树因该如图 1 所示：



(a) 二叉排序树 1　　(b) 二叉排序树 2

图 1: 二叉排序树

如图 1(b) 所示，该二叉排序树退化成了一个链表，查找时间复杂度退化为 $\Theta(n)$ ，然而，正常的二叉排序树查找时间复杂度为 $\Theta(\log n)$ 。为了避免二叉排序树退化成链表，前辈们发明了**平衡二叉排序树**。

平衡二叉排序树是特殊的一种二叉排序树，其可以将查找时间复杂度稳定在 $\Theta(\log n)$ 。笔者将讲述的 AVL 树正是平衡二叉排序树的一种。

2 AVL 树

2.1 简介

如上一节所讲，AVL 树是一种平衡二叉排序树，其比二叉排序树多了一个性质:
$|H(left) - H(right) \leqslant 1|$ ，其中，$H(n)$ 表示 n 节点的高度，left 表示左孩子，right 表示右孩

子。

AVL 树的名字起源于其发明者 *G.M.Adelson.Velsky* 和 *E.M.Landis* 的名字。AVL 树距今有 62 年的历史，其优点是因为限制了树高，所以 AVL 树不会退化成链表

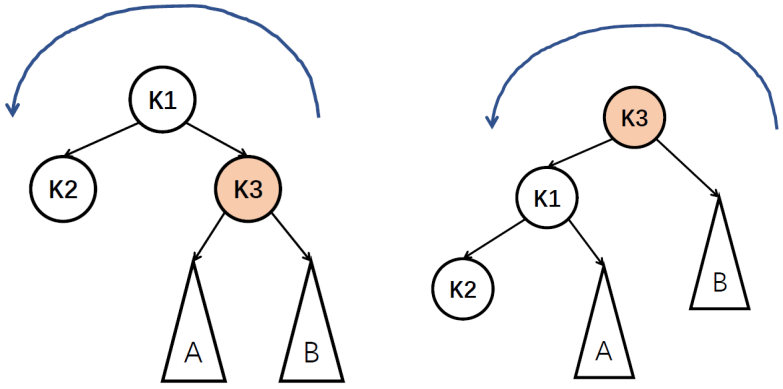
2.2 操作：左旋

为了限制树高，AVL 树引入了两个操作: 左旋和右旋。

左旋的目的是让根节点的右孩子成为根节点，根节点成为根节点的右孩子的左孩子。左旋的方法是:

1. 记录根节点的右孩子。
2. 将根节点的右指针指向根节点的右孩子的左孩子
3. 将根节点的右孩子的左指针指向根节点。

如图组 2 所示。



(a) 左旋 1　　(b) 左旋 2

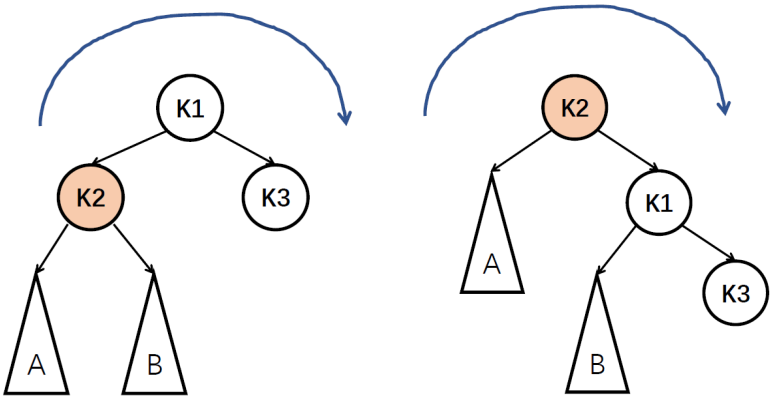
图 2: 左旋

2.3 操作：右旋

右旋的目的是让根节点的左孩子成为根节点，根节点成为根节点的左孩子的左右孩子。右旋是左旋的对称操作，所以其方法与左旋类似:

1. 记录根节点的左孩子。
2. 将根节点的左指针指向根节点的左孩子的右孩子
3. 将根节点的左孩子的右指针指向根节点。

如图组 3 所示。



(a) 右旋 1　　(b) 右旋 2

图 3: 右旋

2.4 定义：失衡类型

失衡类型一共有四个：**LL**、**LR**、**RL**、**RR**。

LL 指的是根节点的左子树和右子树相比左子树更高 (Left)，根节点的左子树的左子树和右子树相比左子树更高 (Left)。类似的，LR 指的是根节点的左子树和右子树相比左子树更高 (Left)，根节点的左子树的左子树和右子树相比右子树更高 (Right)。

RR 指的是根节点的左子树和右子树相比右子树更高 (Right)，根节点的右子树的左子树和右子树相比右子树更高 (Right)。

RL 指的是根节点的左子树和右子树相比右子树更高 (Right)，根节点的右子树的左子树和右子树相比左子树更高 (Left)。

这四种失衡类型中，LL 与 RR 为互逆的，LR 与 RL 是互逆的，也即调整 LL 与 RR 失衡的方法是相反的，调整 RL 与 LR 失衡的方法是相反的，比如调整 LL 失衡的方法是右旋，那么调整 RR 失衡的方法是左旋。

因此，在此笔者只讲述 LL 与 LR 型失衡。

2.4.1 LL 型失衡

正如上文所讲，LL 型失衡的调整方法为右旋，如图 4 所示。

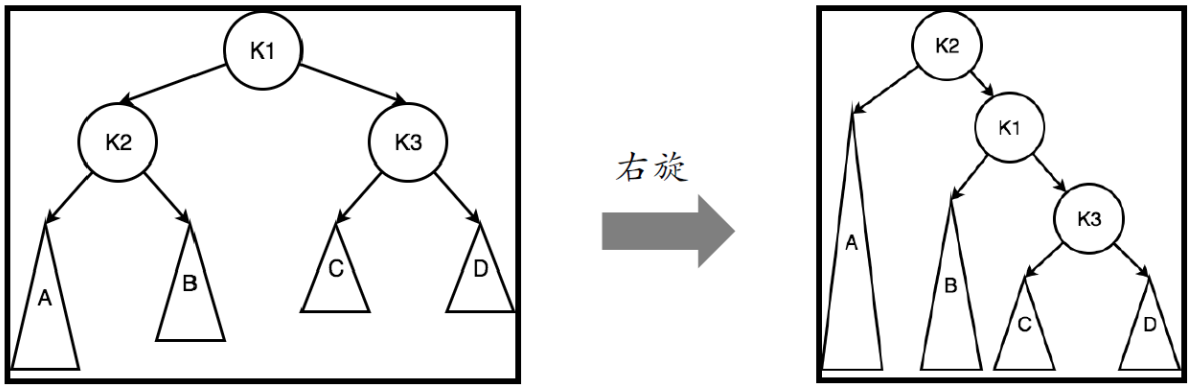


图 4: LL 型失衡

为了证明调整方法是正确的，笔者在此进行数学证明：

证明.

$$
\begin{aligned}
&\because K_2 = K_3 + 2 \\
&\quad K_2 = h_a + 1 \\
&\quad K_3 = \max(h_c, h_d) + 1 \\
&\therefore h_a = \max(h_c, h_d) + 2 \\
&\because h_b = h_a - 1 \\
&\therefore h_a = h_b + 1 \\
&\because K_2 = K_3 + 2 \\
&\quad K_2 = h_a + 1 \\
&\quad h_b = h_a - 1 \\
&\therefore h_b = K_3 \\
&\therefore h_a = \max(h_c, h_d) + 2 \\
&\quad h_a = h_b + 1 \\
&\quad h_b = K_3
\end{aligned}
$$

□

将证明出来的结果带入图 4 右边的 AVL 树，可得 LL 型失衡的调整方法为右旋是正确的。

2.4.2 LR 型失衡

对于 LR 型失衡，可以先以根节点的左孩子作为根节点进行左旋，再对根节点进行右旋，如图 5 所示。

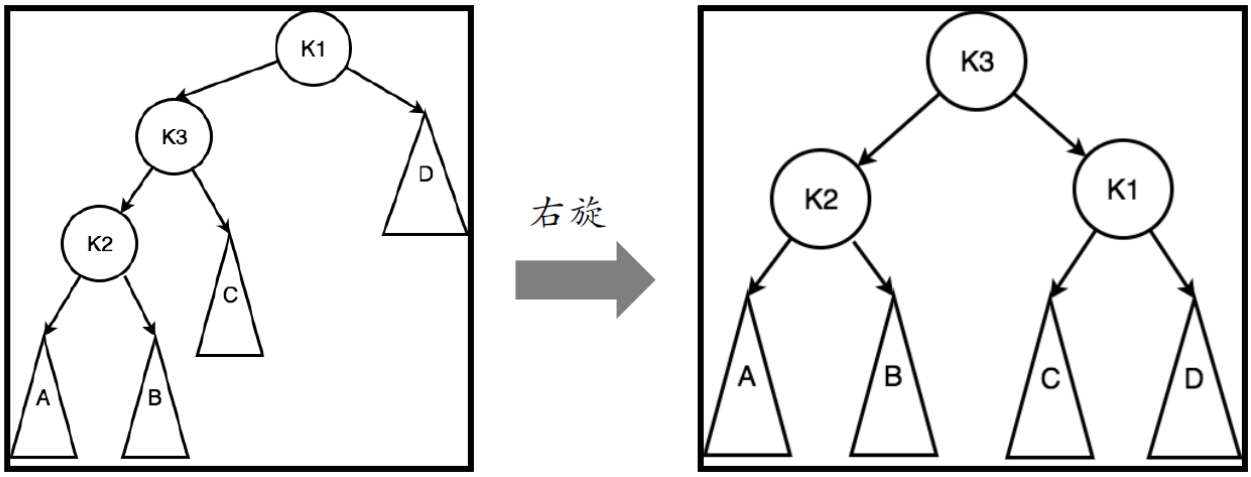


图 5: LR 型失衡

为了证明调整方法是正确的，笔者在此进行数学证明：

证明.

$$
\begin{aligned}
\because K_3 &= \max(h_b, h_c) + 1 \\
h_a &= K_3 - 1 \\
\therefore h_a &= \max(h_b, h_c) \\
\because K_2 &= h_d + 2 \\
K_2 &= K_3 + 1 \\
h_a &= K_3 - 1 \\
\therefore h_d &= K_2 - 2 = K_3 - 1 = h_a \\
\therefore h_a &= \max(h_b, h_c) = h_d
\end{aligned}
$$

□

将推导出来的结果带入图 5 右边的 AVL 树，可得调整方法是正确的。

2.5 操作：插入和删除

AVL 树的插入和删除操作与普通二叉排序树的插入和删除操作差不多，只不过 AVL 树在每一轮递归的末尾要调整高度，解决失衡。

3 代码演示

Listing 1: AVL.cpp

```
#include <bits/stdc++.h>
#include <tools.hpp>
#define NIL (&(AVLTree::mNil))
#define H(n) (n->mH)
#define L(n) (n->mLeft)
#define R(n) (n->mRight)
#define K(n) (n->mKey)
using namespace std;
const char* gTypeStr[5] = { "", "maintain type : LL",
                            "maintain type : LR",
                            "maintain type : RR",
                            "maintain type : RL" };
class AVLTree {
private:
    class TreeNode {
    public:
        int mKey, mH;
        TreeNode *mLeft, *mRight;
        TreeNode () : mKey (-1), mH (0), mLeft (NIL), mRight (NIL) {}
        TreeNode (int key)
            : mKey (key), mH (1), mLeft (NIL), mRight (NIL) {}
        ~TreeNode () {
            if (this == NIL)
                return;
            delete mLeft;
            delete mRight;
        }
        void* operator new (size_t size) {
            return malloc (size);
        }
        void operator delete (void* ptr) {
            if (ptr == NIL)
                return;
            free (ptr);
        }
    };
    TreeNode* mRoot;
    static TreeNode mNil;
    static inline void updateHeight (TreeNode* root) {
        H (root) = (H (L (root)) > H (R (root)) ? H (L (root)) :
                                                  H (R (root)))
            + 1;
    }
    static TreeNode* leftRotate (TreeNode* root) {
        printf ("left rotate : %d\n", K (root));
        TreeNode* t = R (root);
```

```
47          R (root) = L (t);
48          L (t) = root;
49          updateHeight (root);
50          updateHeight (t);
51          return t;
52      }
53      static TreeNode* rightRotate (TreeNode* root) {
54          printf ("right rotate : %d\n", K (root));
55          TreeNode* t = L (root);
56          L (root) = R (t);
57          R (t) = root;
58          updateHeight (root);
59          updateHeight (t);
60          return t;
61      }
62      static TreeNode* maintain (TreeNode* root) {
63          if (abs (H (L (root)) - H (R (root))) <= 1)
64              return root;
65          int type = 0;
66          if (H (L (root)) > H (R (root))) {
67              if (H (R (L (root))) > H (L (L (root)))) {
68                  L (root) = leftRotate (L (root));
69                  type++;
70              }
71              root = rightRotate (root);
72              type++;
73          } else {
74              type = 2;
75              if (H (L (R (root))) > H (R (R (root)))) {
76                  R (root) = rightRotate (R (root));
77                  type++;
78              }
79              root = leftRotate (root);
80              type++;
81          }
82          printf ("%s\n", gTypeStr[type]);
83          return root;
84      }
85      TreeNode* insertT (TreeNode* root, int key) {          // NOLINT
86          if (root == NIL)
87              return new TreeNode (key);
88          if (root->mKey == key)
89              return root;
90          if (key < root->mKey)
91              root->mLeft = insertT (root->mLeft, key);
92          else
93              root->mRight = insertT (root->mRight, key);
94          updateHeight (root);
95          return maintain (root);
96      }
```

```
97      void outputT (TreeNode* root) {          // NOLINT
98          if (root == NIL)
99              return;
100         printf ("(%d[%d] | %d, %d)\n", K (root), H (root),
101                 K (L (root)), K (R (root)));
102         outputT (L (root));
103         outputT (R (root));
104     }
105     TreeNode* eraseT (TreeNode* root, int key) {          // NOLINT
106         if (root == NIL)
107             return root;
108         if (key < K (root))
109             L (root) = eraseT (L (root), key);
110         else if (key > K (root))
111             R (root) = eraseT (R (root), key);
112         else {
113             if (L (root) == NIL || R (root) == NIL) {
114                 TreeNode* tmp = L (root) != NIL ? L (root) : R (root);
115                 free (root);
116                 return tmp;
117             } else {
118                 TreeNode* temp = predecessor (root);
119                 root->mKey = temp->mKey;
120                 L (root) = eraseT (L (root), K (temp));
121             }
122         }
123         updateHeight (root);
124         return maintain (root);
125     }
126     static TreeNode* predecessor (TreeNode* root) {
127         TreeNode* temp = L (root);
128         while (R (temp) != NIL)
129             temp = R (temp);
130         return temp;
131     }
132     TreeNode* findT (TreeNode* root, int key) {          // NOLINT
133         if (root == NIL)
134             return NIL;
135         if (K (root) == key)
136             return root;
137         else if (key < K (root))
138             return findT (L (root), key);
139         else
140             return findT (R (root), key);
141     }
142
143 public:
144     AVLTree () : mRoot (NIL) {}
145     ~AVLTree () {
146         delete mRoot;
```

```
147     }
148     void insert (int key) {
149         mRoot = insertT (mRoot, key);
150     }
151     void erase (int key) {
152         mRoot = eraseT (mRoot, key);
153     }
154     void output () {
155         outputT (mRoot);
156     }
157     int find (int key) {
158         if (findT (mRoot, key) == NIL)
159             return 0;
160         else
161             return 1;
162     }
163     __attribute__ ((constructor)) friend void init_nil ();
164 };
165 AVLTree::TreeNode AVLTree::mNil;          // NOLINT
166 __attribute__ ((constructor)) void init_nil () {
167     NIL->mKey = -1;
168     NIL->mH = 0;
169     NIL->mLeft = NIL->mRight = NIL;
170 }
171 int main () {
172     AVLTree* t = new AVLTree;
173     int x;
174     // insert
175     while (scanf ("%d", &x)) {
176         if (x == -1)
177             break;
178         printf ("insert %d to AVLTree\n", x);
179         t->insert (x);
180         t->output ();
181     }          // erase
182     while (scanf ("%d", &x)) {
183         if (x == -1)
184             break;
185         printf ("erase %d to AVLTree\n", x);
186         t->erase (x);
187         t->output ();
188     }
189     // find
190     while (scanf ("%d", &x)) {
191         if (x == -1)
192             break;
193         printf ("find %d in AVLTree : %d\n", x, t->find (x));
194     }
195     delete t;
196     return 0;
```

```
197 }
```

A 附录：编码

在源代码中，我们定义了一个代表空的 TreeNode 对象：NIL。为了不占用内存空间，笔者将 NIL 类型声明为 static，同时写了一个函数 *init_nil()*（虽然在 linux 下没有用）为了释放时不释放 NIL，我们重载 new 和 delete 运算符，如果发现要销毁的指针为 NIL，则跳过这轮释放。因为如果多次销毁 NIL 会出现错误，而且 NIL 存储在静态内存区，由程序释放，因此设计成这样。此外，*init_nil()* 函数被声明称 AVLTree 类的友元，这样 *init_nil()* 才能访问到 NIL。