

BST Tree

verbovyar

November 29 2020 time:1:30

AVL Tree

1) Структура узла

```
1 struct Node {  
2     int key = 0;  
3     Node* parent = nullptr;  
4     Node* left = nullptr;  
5     Node* right = nullptr;  
6     unsigned int height = 0;  
7 };
```

- 1) Ключ или значение узла
- 2) Родитель, предшествующий узел
- 3) Левый узел
- 4) Правый узел
- 5) Высота соответствующего узла

2) Структура дерева

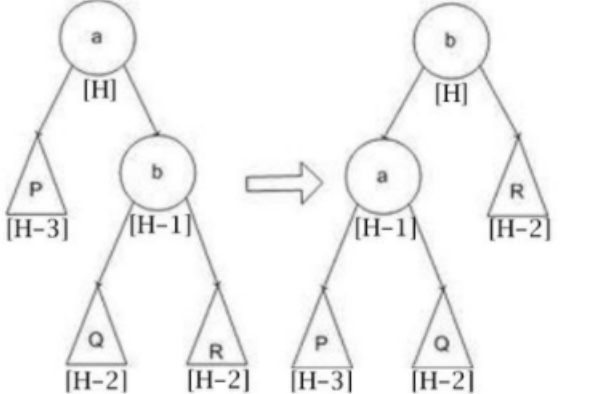
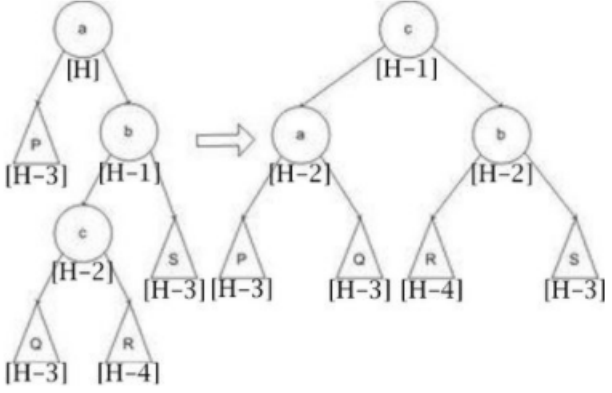
```
1 AVLTree{  
2     Node* root = nullptr;  
3     int size = 0;  
4 };
```

- 1) Корень дерева
- 2) Размер дерева, кол-во элементов

3) Повороты

Опишем операции балансировки, а именно малый левый поворот, большой левый поворот и случаи их возникновения. Балансировка нам нужна для операций добавления и удаления узла. Для исправления факторов баланса, достаточно знать факторы баланса двух(в случае большого поворота — трех) вершин перед поворотом, и исправить значения этих же вершин после поворота. Обозначим фактор баланса вершины **i** как **balance[i]**. Операции поворота делаются на том шаге, когда мы находимся в правом сыне вершины **a**, если мы производим операцию добавления, и в левом сыне, если мы производим операцию удаления. Вычисления производим заранее, чтобы не

допустить значения **2** или **-2** в вершине а. На каждой иллюстрации изображен один случай высот поддеревьев. Нетрудно убедиться, что в остальных случаях всё тоже будет корректно

Тип вращения	Иллюстрация
<p>Малое левое вращение</p>	
<p>Большое левое вращение</p>	

код поворотов

```
1 void SmallLeft(AVLTree* tree , Node* node)
2 {
3     if (node != nullptr)
4     {
5         Node* temp = node->right;
6         node->right = temp->left;
7         if (temp->left != nullptr)
8         {
9             temp->left->parent = node;
10        }
11
12        temp->parent = node->parent;
13        if (node->parent == nullptr)
14        {
15            tree->root = temp;
16        }
17        else if (node == node->parent->left)
18        {
19            node->parent->left = temp;
20        }
21        else
22        {
23            node->parent->right = temp;
24        }
25
26        temp->left = node;
27        node->parent = temp;
28
29        SetHeight(node);
30        SetHeight(node->parent);
31    }
32 }
```

```
1 void BigLeft(AVLTree* tree , Node* node)
2 {
3     if (node != nullptr)
4     {
5         SmallRight(tree , node->right);
6         SmallLeft(tree , node);
7     }
8 }
```

4) Баланс

```
1 int Balance(Node* node)
2 {
3     return GetHeight(node->right) - GetHeight(node->left);
4 }
```

```
1 inline
2 int GetHeight(Node* node) {
3     return node ? node->height : 0;
4 }
5 inline
6 void SetHeight(Node* node) {
7     if (node != nullptr)
8     {
9         node->height = fmax(GetHeight(node->left);
10            GetHeight(node->right)) + 1;
11     }
12 }
```

5) Добавление вершины

Пусть нам надо добавить ключ **t**. Будем спускаться по дереву, как при поиске ключа **t**. Если мы стоим в вершине **a** и нам надо идти в поддереву, которого нет, то делаем ключ **t** листом, а вершину **a** его корнем. Далее поднимаемся вверх по пути поиска и пересчитываем баланс у вершин. Если мы поднялись в вершину **i** из левого поддерева, то **diff[i]** увеличивается на единицу, если из правого, то уменьшается на единицу. Если пришли в вершину и её баланс стал равным нулю, то это значит высота поддерева не изменилась и подъём останавливается. Если пришли в вершину и её баланс стал равным **1** или **-1**, то это значит высота поддерева изменилась и подъём продолжается. Если пришли в вершину и её баланс стал равным **2** или **-2**, то делаем одно из четырёх вращений и, если после вращения баланс стал равным нулю, то останавливаемся, иначе продолжаем подъём.

Так как в процессе добавления вершины мы рассматриваем не более, чем **O(h)** вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет **O(logn)** операций.

```
1 void Insert(AVLTree* tree, int key) {
2     Node* new_parent = nullptr;
3     Node* new_leaf = tree->root;
4     Node* node = (Node*)calloc(1, sizeof(Node));
5     node->parent = new_parent;
6     node->key = key;
7     if (tree->size == 0)
8     {
9         tree->root = node;
10    }
11    else
12    {
13        while (new_leaf)
14        {
15            new_parent = new_leaf;
16            if (key < new_leaf->key)
17            {
18                new_leaf = new_leaf->left;
19            }
20            else
21            {
22                new_leaf = new_leaf->right;
23            }
24        }
25        node->parent = new_parent;
26        if (!new_parent)
27        {
28            tree->root = node;
29        }
30        else if (node->key < new_parent->key)
31        {
32            new_parent->left = node;
33        }
34        else
35        {
36            new_parent->right = node;
37        }
38        FixUp(tree, node);
39    }
40    tree->size++;
41 }
42 }
```

```

1 void FixUp(AVLTree* tree , Node* node)
2 {
3     while (node != nullptr)
4     {
5         if (Balance(node) == LEFT_ROTATE)
6         {
7             if (Balance(node->right) < 0)
8             {
9                 BigLeft(tree , node);
10            }
11            else
12            {
13                SmallLeft(tree , node);
14            }
15        }
16        if (Balance(node) == RIGHT_ROTATE)
17        {
18            if (Balance(node->left) > 0)
19            {
20                BigRight(tree , node);
21            }
22            else
23            {
24                SmallRight(tree , node);
25            }
26        }
27        SetHeight(node);
28        node = node->parent;
29    }
30 }
31
32

```

5) Добавление вершины

Для простоты опишем рекурсивный алгоритм удаления. Если вершина — лист, то удалим её, иначе найдём самую близкую по значению вершину **a**, переместим её на место удаляемой вершины и удалим вершину **a**. От удалённой вершины будем подниматься вверх к корню и пересчитывать баланс у вершин. Если мы поднялись в вершину **i** из левого поддерева, то **diff[i]** уменьшается на единицу, если из правого, то увеличивается на единицу. Если пришли в вершину и её баланс стал равным **1** или **-1**, то это значит, что высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины стал равным нулю, то высота поддерева уменьшилась и подъём нужно продолжить. Если баланс стал равным **2** или **-2**, следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.

В результате указанных действий на удаление вершины и балансировку суммарно тратится, как и ранее, **O(h)** операций. Таким образом, требуемое количество действий — **O(logn)**.

```

1 void Erase(AVLTree* tree , Node* node)
2 {
3     if (node == nullptr)
4     {
5         return;
6     }
7
8     Node* new_node = nullptr;
9
10    if (node->left == nullptr)
11    {
12        new_node = node->right;
13        GetChild(tree , node , node->right);
14        FixUp(tree , new_node);
15
16        return;
17    }
18
19    if (node->right == nullptr)
20    {
21        new_node = node->left;
22        GetChild(tree , node , node->left);
23        FixUp(tree , new_node);
24
25        return;
26    }
27
28    new_node = node->right;
29    while (new_node->left != nullptr)
30    {
31        new_node = new_node->left;
32    }
33
34    if (new_node->parent != node)
35    {
36        GetChild(tree , new_node , new_node->right);
37        new_node->right = node->right;
38        new_node->right->parent = new_node;
39    }
40    else {
41        new_node->right->parent = new_node;
42    }
43
44    GetChild(tree , node , new_node);
45    new_node->left = node->left;
46    new_node->left->parent = new_node;
47
48    —tree->size;
49
50    FixUp(tree , new_node);
51 }
52
53 void Erase(AVLTree* tree , int key)
54 {
55     Erase(tree , FindSubtree(tree->root , key));
56 }

```

Вспомогательная функция для получения сына

```

1 void GetChild(AVLTree* tree, Node* node, Node* child)
2 {
3     if (node->parent == nullptr)
4     {
5         tree->root = child;
6     }
7     else if (node->parent->left == node)
8     {
9         node->parent->left = child;
10    }
11    else
12    {
13        node->parent->right = child;
14    }
15
16    if (child != nullptr)
17    {
18        child->parent = node->parent;
19    }
20 }

```

Таким образом, получаем двоичное сбалансированное дерево поиска.

Немного истории:

АВЛ-дерево (англ. AVL-Tree) — сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

АВЛ-деревья названы по первым буквам фамилий их изобретателей, **Г. М. Адельсона-Вельского** и **Е. М. Ландиса**, которые впервые предложили использовать АВЛ-деревья в 1962 году.

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются.

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от $\log(n + 1)$ до $1.44 \log(n + 2) - 0.328$. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем гарантированную логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве. Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается не равной нулю.

Полный код

```

1 #define _CRT_SECURE_NO_WARNINGS
2
3 #include <assert.h>
4 #include <math.h>
5 #include <stdio.h>
6 #include <string.h>
7 #include <stdlib.h>
8 #include <stdint.h>
9
10 const int RIGHT_ROTATE = -2;
11 const int LEFT_ROTATE = 2;
12
13 struct Node {
14     int key = 0;
15     Node* parent = nullptr;
16     Node* left = nullptr;
17     Node* right = nullptr;
18     unsigned int height = 0;
19 };
20
21 struct AVLTree
22 {
23     Node* root = nullptr;
24     size_t size = 0;
25 };
26
27 inline
28 int GetHeight(Node* node);
29 inline
30 void SetHeight(Node* node);
31
32 AVLTree* Construct(AVLTree* temp);
33 void DeleteAVLTree(AVLTree* tree);
34
35 void SmallLeft(AVLTree* tree, Node* node);
36 void SmallRight(AVLTree* tree, Node* node);
37 void BigLeft(AVLTree* tree, Node* node);
38 void BigRight(AVLTree* tree, Node* node);
39 bool Find(AVLTree* tree, int key);
40 Node* FindSubtree(Node* subtree_root, int key);
41
42 int Balance(Node* node);
43 void FixUp(AVLTree* tree, Node* node);
44
45 void Insert(AVLTree* tree, int key);
46
47 inline
48 int GetHeight(Node* node) {
49     return node ? node->height : 0;
50 }
51
52 inline
53 void SetHeight(Node* node) {
54     if (node != nullptr)
55     {
56         node->height = fmax(GetHeight(node->left),
57                             GetHeight(node->right)) + 1;
58     }
59 }
60
61 AVLTree* Construct(AVLTree* temp)
62 {
63     AVLTree* tree = (AVLTree*) calloc(1, sizeof(AVLTree));
64     tree->root = (Node*) calloc(1, sizeof(Node));
65     tree->size = 0;
66

```



```

67         return tree;
68     }
69
70     void DeleteAVLTree(AVLTree* tree)
71     {
72         tree->size = 0;
73         free(tree->root);
74     }
75
76     void SmallLeft(AVLTree* tree, Node* node)
77     {
78         if (node != nullptr)
79         {
80             Node* temp = node->right;
81             node->right = temp->left;
82             if (temp->left != nullptr)
83             {
84                 temp->left->parent = node;
85             }
86
87             temp->parent = node->parent;
88             if (node->parent == nullptr)
89             {
90                 tree->root = temp;
91             }
92             else if (node == node->parent->left)
93             {
94                 node->parent->left = temp;
95             }
96             else
97             {
98                 node->parent->right = temp;
99             }
100
101             temp->left = node;
102             node->parent = temp;
103
104             SetHeight(node);
105             SetHeight(node->parent);
106         }
107     }
108
109     void SmallRight(AVLTree* tree, Node* node)
110     {
111         if (node != nullptr)
112         {
113             Node* temp = node->left;
114             node->left = temp->right;
115             if (temp->right != nullptr)
116             {
117                 temp->right->parent = node;
118             }
119
120             temp->parent = node->parent;
121             if (node->parent == nullptr)
122             {
123                 tree->root = temp;
124             }
125             else if (node == node->parent->right)
126             {
127                 node->parent->right = temp;
128             }
129             else
130             {
131                 node->parent->left = temp;
132             }
133
134             temp->right = node;

```

```

135         node->parent = temp;
136
137         SetHeight(node);
138         SetHeight(node->parent);
139     }
140 }
141
142 void BigLeft(AVLTree* tree , Node* node)
143 {
144     if (node != nullptr)
145     {
146         SmallRight(tree , node->right);
147         SmallLeft(tree , node);
148     }
149 }
150
151 void BigRight(AVLTree* tree , Node* node)
152 {
153     if (node != nullptr)
154     {
155         SmallLeft(tree , node->left);
156         SmallRight(tree , node);
157     }
158 }
159
160 int Balance(Node* node)
161 {
162     return GetHeight(node->right) - GetHeight(node->left);
163 }
164
165 void FixUp(AVLTree* tree , Node* node)
166 {
167     while (node != nullptr)
168     {
169         if (Balance(node) == LEFT_ROTATE)
170         {
171             if (Balance(node->right) < 0)
172             {
173                 BigLeft(tree , node);
174             }
175             else
176             {
177                 SmallLeft(tree , node);
178             }
179         }
180
181         if (Balance(node) == RIGHT_ROTATE)
182         {
183             if (Balance(node->left) > 0)
184             {
185                 BigRight(tree , node);
186             }
187             else
188             {
189                 SmallRight(tree , node);
190             }
191         }
192
193         SetHeight(node);
194         node = node->parent;
195     }
196 }
197
198 void Insert(AVLTree* tree , int key) {
199     Node* new_parent = nullptr;
200     Node* new_leaf = tree->root;
201     Node* node = (Node*) calloc(1, sizeof(Node));
202     node->parent = new_parent;

```

```

203     node->key = key;
204
205     if (tree->size == 0)
206     {
207         tree->root = node;
208     }
209     else
210     {
211         while (new_leaf)
212         {
213             new_parent = new_leaf;
214
215             if (key < new_leaf->key)
216             {
217                 new_leaf = new_leaf->left;
218             }
219             else
220             {
221                 new_leaf = new_leaf->right;
222             }
223         }
224
225         node->parent = new_parent;
226         if (!new_parent)
227         {
228             tree->root = node;
229         }
230         else if (node->key < new_parent->key)
231         {
232             new_parent->left = node;
233         }
234         else
235         {
236             new_parent->right = node;
237         }
238
239         FixUp(tree, node);
240     }
241
242     tree->size++;
243 }
244
245 Node* FindSubtree(Node* subtree_root, int key)
246 {
247     if (subtree_root == nullptr)
248     {
249         return nullptr;
250     }
251
252     if (key < subtree_root->key)
253     {
254         return FindSubtree(subtree_root->left, key);
255     }
256
257     if (subtree_root->key < key)
258     {
259         return FindSubtree(subtree_root->right, key);
260     }
261
262     return subtree_root;
263 }
264
265 bool Find(AVLTree* tree, int key)
266 {
267     if (FindSubtree(tree->root, key) != nullptr)
268     {
269         return true;
270     }

```

```

271         return false;
272     }
273 }
274
275 void GetChild(AVLTree* tree, Node* node, Node* child)
276 {
277     if (node->parent == nullptr)
278     {
279         tree->root = child;
280     }
281     else if (node->parent->left == node)
282     {
283         node->parent->left = child;
284     }
285     else
286     {
287         node->parent->right = child;
288     }
289     if (child != nullptr)
290     {
291         child->parent = node->parent;
292     }
293 }
294
295 void Erase(AVLTree* tree, Node* node)
296 {
297     if (node == nullptr)
298     {
299         return;
300     }
301
302     Node* new_node = nullptr;
303
304     if (node->left == nullptr)
305     {
306         new_node = node->right;
307         GetChild(tree, node, node->right);
308         FixUp(tree, new_node);
309
310         return;
311     }
312
313     if (node->right == nullptr)
314     {
315         new_node = node->left;
316         GetChild(tree, node, node->left);
317         FixUp(tree, new_node);
318
319         return;
320     }
321
322     new_node = node->right;
323     while (new_node->left != nullptr)
324     {
325         new_node = new_node->left;
326     }
327
328     if (new_node->parent != node)
329     {
330         GetChild(tree, new_node, new_node->right);
331         new_node->right = node->right;
332         new_node->right->parent = new_node;
333     }
334     else {
335         new_node->right->parent = new_node;
336     }
337 }
338

```

```

339         GetChild(tree, node, new_node);
340         new_node->left = node->left;
341         new_node->left->parent = new_node;
342
343         --tree->size;
344
345         FixUp(tree, new_node);
346     }
347
348     void Erase(AVLTree* tree, int key)
349     {
350         Erase(tree, FindSubtree(tree->root, key));
351     }

```