

AVL 树

AVL 树，是一种平衡的二叉搜索树。由于各种算法教材上对 AVL 的介绍十分冗长，造成了很多对 AVL 树复杂、不实用的印象。但实际上，AVL 树的原理简单，实现也并不复杂。

性质

1. 空二叉树是一个 AVL 树
2. 如果 T 是一棵 AVL 树，那么其左右子树也是 AVL 树，并且 $|h(ls) - h(rs)| \leq 1$ ，h 是其左右子树的高度
3. 树高为 $O(\log n)$

平衡因子：右子树高度 - 左子树高度

树高的证明

设 f_n 为高度为 n 的 AVL 树所包含的最少节点数，则有

$$f_n = \begin{cases} 1 & (n = 1) \\ 2 & (n = 2) \\ f_{n-1} + f_{n-2} + 1 & (n > 2) \end{cases}$$

根据常系数非齐次线性差分方程的解法， $\{f_n + 1\}$ 是一个斐波那契数列。这里 f_n 的通项为：

$$f_n = \frac{5 + 2\sqrt{5}}{5} \left(\frac{1 + \sqrt{5}}{2} \right)^n + \frac{5 - 2\sqrt{5}}{5} \left(\frac{1 - \sqrt{5}}{2} \right)^n - 1$$

斐波那契数列以指数的速度增长，对于树高 n 有：

$$n < \log_{\frac{1+\sqrt{5}}{2}}(f_n + 1) < \frac{3}{2} \log_2(f_n + 1)$$

因此 AVL 树的高度为 $O(\log f_n)$ ，这里的 f_n 为结点数。

过程

插入结点

与 BST（二叉搜索树）中类似，先进行一次失败的查找来确定插入的位置，插入节点后根据平衡因子来决定是否需要调整。

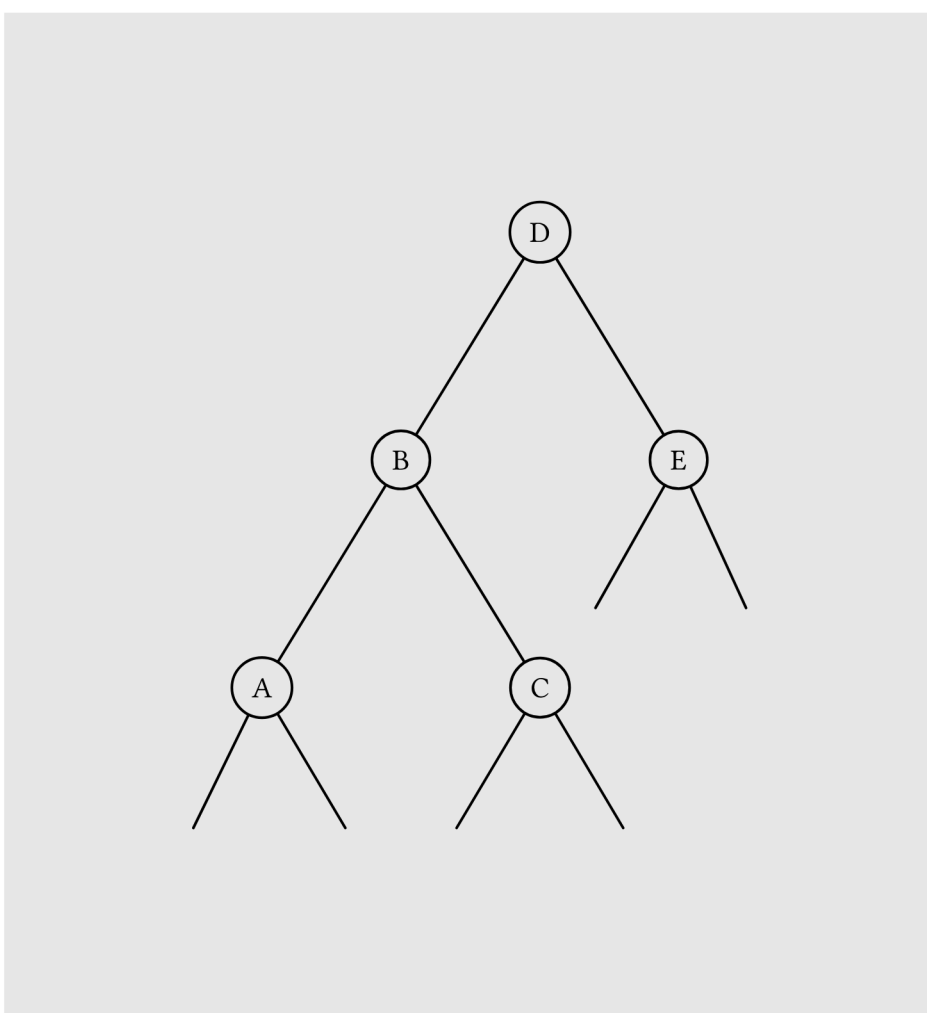
删除结点

删除和 BST 类似，将结点与后继交换后再删除。

删除会导致树高以及平衡因子变化，这时需要沿着被删除结点到根的路径来调整这种变化。

平衡的维护

插入或删除节点后，可能会造成 AVL 树的性质 2 被破坏。因此，需要沿着从被插入/删除的节点到根的路径对树进行维护。如果对于某一个节点，性质 2 不再满足，由于我们只插入/删除了一个节点，对树高的影响不超过 1，因此该节点的平衡因子的绝对值至多为 2。由于对称性，我们在此只讨论左子树的高度比右子树大 2 的情况，即下图中 $h(B) - h(E) = 2$ 。此时，还需要根据 $h(A)$ 和 $h(C)$ 的大小关系分两种情况讨论。需要注意的是，由于我们是自底向上维护平衡的，因此对节点 D 的所有后代来说，性质 2 仍然是被满足的。

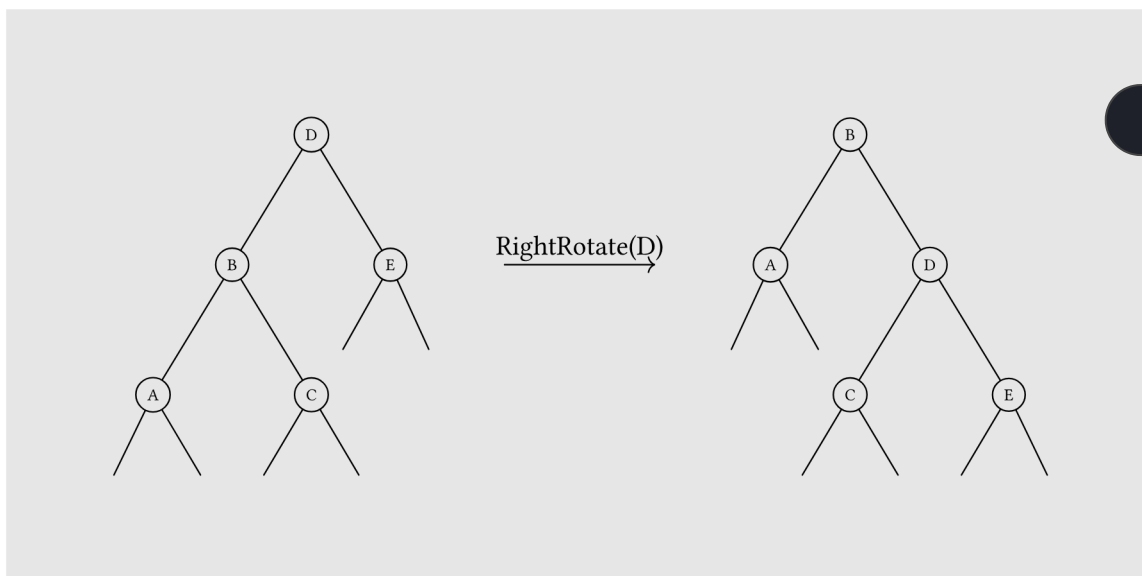


$$h(A) \geq h(C)$$

设 $h(E) = x$ ，则有

$$\begin{cases} h(B) = x + 2 \\ h(A) = x + 1 \\ x \leq h(C) \leq x + 1 \end{cases}$$

其中 $h(C) \geq x$ 是由于节点 B 满足性质 2，因此 $h(C)$ 和 $h(A)$ 的差不会超过 1。此时我们对节点 D 进行一次右旋操作（旋转操作与其它类型的平衡二叉搜索树相同），如下图所示。



显然节点 A、C、E 的高度不发生变化，并且有

$$\begin{cases} 0 \leq h(C) - h(E) \leq 1 \\ x + 1 \leq h'(D) = \max(h(C), h(E)) + 1 = h(C) + 1 \leq x + 2 \\ 0 \leq h'(D) - h(A) \leq 1 \end{cases}$$

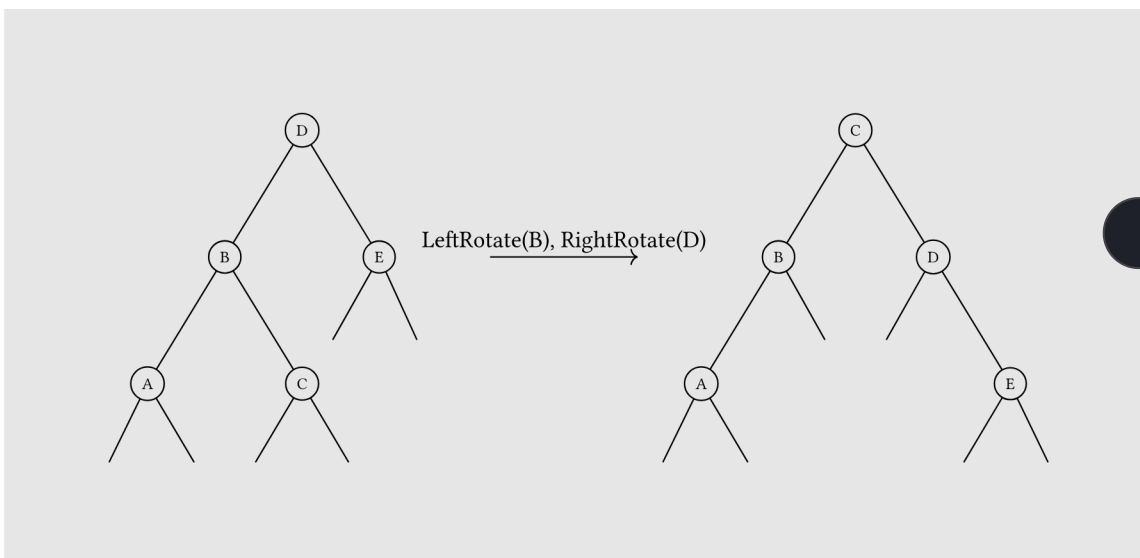
因此旋转后的节点 B 和 D 也满足性质 2。

$$h(A) < h(C)$$

设 $h(E) = x$ ，则与刚才同理，有

$$\begin{cases} h(B) = x + 2 \\ h(C) = x + 1 \\ h(A) = x \end{cases}$$

此时我们先对节点 B 进行一次左旋操作，再对节点 D 进行一次右旋操作，如下图所示。



显然节点 A、E 的高度不发生变化，并且 B 的新右儿子和 D 的新左儿子分别为 C 原来的左右儿子，则有

$$\begin{cases} x-1 \leq h'(rs_B), h'(ls_D) \leq x \\ 0 \leq h(A) - h'(rs_B) \leq 1 \\ 0 \leq h(E) - h'(ls_D) \leq 1 \\ h'(B) = \max(h(A), h'(rs_B)) + 1 = x + 1 \\ h'(D) = \max(h(E), h'(ls_D)) + 1 = x + 1 \\ h'(B) - h'(D) = 0 \end{cases}$$

因此旋转后的节点 B、C、D 也满足性质 2。

维护平衡操作：伪代码

```

1  function MaintainBalance(p)
2      l ← lsp, r ← rsp
3      if h(l) - h(r) = 2
4          if h(lsl) ≥ h(rsl)
5              RightRotate(p)
6          else
7              LeftRotate(l)
8              RightRotate(p)
9      else if h(l) - h(r) = -2
10         if h(lsr) ≤ h(rsr)
11             LeftRotate(p)
12         else
13             RightRotate(r)
14             LeftRotate(p)

```

与其他平衡二叉搜索树相同，AVL 树中节点的高度、子树大小等信息需要在旋转时进行维护。

其他操作

AVL 树的其他操作（Predecessor、Successor、Select、Rank 等）与普通的二叉搜索树相同。

参考代码

下面的代码是用 AVL 树实现的 `Map`，即有序不可重映射：

参考代码

```
1  /**
2   * @brief An AVLTree-based map implementation
3   * @details The map is sorted according to the natural ordering
4   of its
5   * keys or by a {@code Compare} function provided; This
6   implementation
7   * provides guaranteed log(n) time cost for the contains, get,
8   insert
9   * and remove operations.
10  */
11
12  #ifndef AVL_TREE_MAP_HPP
13  #define AVL_TREE_MAP_HPP
14
15  #include <cassert>
16  #include <stddef.h>
17  #include <stdint.h>
18  #include <functional>
19  #include <memory>
20  #include <stack>
21  #include <utility>
22  #include <vector>
23
24  /**
25   * An AVLTree-based map implementation
26   * https://en.wikipedia.org/wiki/AVL\_tree
27   * @tparam Key the type of keys maintained by this map
28   * @tparam Value the type of mapped values
29   * @tparam Compare
30   */
31  template <typename Key, typename Value, typename Compare =
32  std::less<Key> >
33  class AVLTreeMap {
34  private:
35      using USize = size_t;
36      using Factor = int64_t;
37
38      Compare compare = Compare();
39
40  public:
41      struct Entry {
42          Key key;
43          Value value;
44
45          bool operator==(const Entry &rhs) const noexcept {
46              return this->key == rhs.key && this->value == rhs.value;
47          }
48
49          bool operator!=(const Entry &rhs) const noexcept {
```

```

50     return this->key != rhs.key || this->value != rhs.value;
51 }
52 };
53
54 private:
55 struct Node {
56     using Ptr = std::shared_ptr<Node>;
57     using Provider = const std::function<Ptr(void)> &;
58     using Consumer = const std::function<void(const Ptr &)> &;
59
60     Key key;
61     Value value{};
62
63     Ptr left = nullptr;
64     Ptr right = nullptr;
65
66     USize height = 1;
67
68     explicit Node(Key k) : key(std::move(k)) {}
69
70     explicit Node(Key k, Value v) : key(std::move(k)),
71 value(std::move(v)) {}
72
73     ~Node() = default;
74
75     inline bool isLeaf() const noexcept {
76         return this->left == nullptr && this->right == nullptr;
77     }
78
79     inline void updateHeight() noexcept {
80         if (this->isLeaf()) {
81             this->height = 1;
82         } else if (this->left == nullptr) {
83             this->height = this->right->height + 1;
84         } else if (this->right == nullptr) {
85             this->height = this->left->height + 1;
86         } else {
87             this->height = std::max(left->height, right->height) +
88 1;
89         }
90     }
91
92     inline Factor factor() const noexcept {
93         if (this->isLeaf()) {
94             return 0;
95         } else if (this->left == nullptr) {
96             return (Factor)this->right->height;
97         } else if (this->right == nullptr) {
98             return (Factor) - this->left->height;
99         } else {
100             return (Factor)(this->right->height - this->left-
101 >height);

```

```

102     }
103 }
104
105     inline Entry entry() const { return Entry{key, value}; }
106
107     static Ptr from(const Key &k) { return
108 std::make_shared<Node>(Node(k)); }
109
110     static Ptr from(const Key &k, const Value &v) {
111         return std::make_shared<Node>(Node(k, v));
112     }
113 };
114
115 using NodePtr = typename Node::Ptr;
116 using ConstNodePtr = const NodePtr &;
117 using NodeProvider = typename Node::Provider;
118 using NodeConsumer = typename Node::Consumer;
119
120 NodePtr root = nullptr;
121 USize count = 0;
122
123 using K = const Key &;
124 using V = const Value &;
125
126 public:
127     using EntryList = std::vector<Entry>;
128     using KeyValueConsumer = const std::function<void(K, V)> &;
129     using MutKeyValueConsumer = const std::function<void(K, Value
130 &)> &;
131     using KeyValueFilter = const std::function<bool(K, V)> &;
132
133     class NoSuchMappingException : protected std::exception {
134     private:
135         const char *message;
136
137     public:
138         explicit NoSuchMappingException(const char *msg) :
139 message(msg) {}
140
141         const char *what() const noexcept override { return message;
142     }
143     };
144
145     AvlTreeMap() noexcept = default;
146
147     /**
148      * Returns the number of entries in this map.
149      * @return size_t
150      */
151     inline USize size() const noexcept { return this->count; }
152
153     /**

```



```

154     * Returns true if this collection contains no elements.
155     * @return bool
156     */
157     inline bool empty() const noexcept { return this->count == 0;
158 }
159
160     /**
161     * Removes all of the elements from this map.
162     */
163     void clear() noexcept {
164         this->root = nullptr;
165         this->count = 0;
166     }
167
168     /**
169     * Returns the value to which the specified key is mapped; If
170 this map
171     * contains no mapping for the key, a {@code
172 NoSuchElementException} will
173     * be thrown.
174     * @param key
175     * @return AvlTreeMap<Key, Value>::Value
176     * @throws NoSuchElementException
177     */
178     Value get(K key) const {
179         if (this->root == nullptr) {
180             throw NoSuchElementException("Invalid key");
181         } else {
182             NodePtr node = this->getNode(this->root, key);
183             if (node != nullptr) {
184                 return node->value;
185             } else {
186                 throw NoSuchElementException("Invalid key");
187             }
188         }
189     }
190
191     /**
192     * Returns the value to which the specified key is mapped; If
193 this map
194     * contains no mapping for the key, a new mapping with a
195 default value
196     * will be inserted.
197     * @param key
198     * @return AvlTreeMap<Key, Value>::Value &
199     */
200     Value &getOrDefault(K key) {
201         if (this->root == nullptr) {
202             this->root = Node::from(key);
203             this->count += 1;
204             return this->root->value;
205         } else {

```

```

206         return this
207         ->getNodeOrProvide(this->root, key,
208                             [&key]() { return Node::from(key);
209     })
210         ->value;
211     }
212 }
213
214 /**
215  * Returns true if this map contains a mapping for the
216  specified key.
217  * @param key
218  * @return bool
219  */
220 bool contains(K key) const {
221     return this->getNode(this->root, key) != nullptr;
222 }
223
224 /**
225  * Associates the specified value with the specified key in
226  this map.
227  * @param key
228  * @param value
229  */
230 void insert(K key, V value) {
231     if (this->root == nullptr) {
232         this->root = Node::from(key, value);
233         this->count += 1;
234     } else {
235         this->insert(this->root, key, value);
236     }
237 }
238
239 /**
240  * If the specified key is not already associated with a
241  value, associates
242  * it with the given value and returns true, else returns
243  false.
244  * @param key
245  * @param value
246  * @return bool
247  */
248 bool insertIfAbsent(K key, V value) {
249     USize sizeBeforeInsertion = this->size();
250     if (this->root == nullptr) {
251         this->root = Node::from(key, value);
252         this->count += 1;
253     } else {
254         this->insert(this->root, key, value, false);
255     }
256     return this->size() > sizeBeforeInsertion;
257 }

```

```

258
259     /**
260      * If the specified key is not already associated with a
261      value, associates
262      * it with the given value and returns the value, else returns
263      the associated
264      * value.
265      * @param key
266      * @param value
267      * @return
268      */
269     Value &getOrInsert(K key, V value) {
270         if (this->root == nullptr) {
271             this->root = Node::from(key, value);
272             this->count += 1;
273             return root->value;
274         } else {
275             NodePtr node = getNodeOrProvide(this->root, key,
276                                             [&]() { return
277 Node::from(key, value); });
278             return node->value;
279         }
280     }
281
282     Value operator[](K key) const { return this->get(key); }
283
284     Value &operator[](K key) { return this->getOrDefault(key); }
285
286     /**
287      * Removes the mapping for a key from this map if it is
288      present;
289      * Returns true if the mapping is present else returns false
290      * @param key the key of the mapping
291      * @return bool
292      */
293     bool remove(K key) {
294         if (this->root == nullptr) {
295             return false;
296         } else {
297             return this->remove(this->root, key, [](ConstNodePtr) {});
298         }
299     }
300
301     /**
302      * Removes the mapping for a key from this map if it is
303      present and returns
304      * the value which is mapped to the key; If this map contains
305      no mapping for
306      * the key, a {@code NoSuchElementException} will be thrown.
307      * @param key
308      * @return AVLTreeMap<Key, Value>::Value
309      * @throws NoSuchElementException

```

```

310     */
311     Value getAndRemove(K key) {
312         Value result;
313         NodeConsumer action = [&](ConstNodePtr node) { result =
314 node->value; };
315
316         if (root == nullptr) {
317             throw NoSuchMappingException("Invalid key");
318         } else {
319             if (remove(this->root, key, action)) {
320                 return result;
321             } else {
322                 throw NoSuchMappingException("Invalid key");
323             }
324         }
325     }
326
327     /**
328      * Gets the entry corresponding to the specified key; if no
329      such entry
330      * exists, returns the entry for the least key greater than
331      the specified
332      * key; if no such entry exists (i.e., the greatest key in the
333      Tree is less
334      * than the specified key), a {code NoSuchMappingException}
335      will be thrown.
336      * @param key
337      * @return AvlTreeMap<Key, Value>::Entry
338      * @throws NoSuchMappingException
339      */
340     Entry getCeilingEntry(K key) const {
341         if (this->root == nullptr) {
342             throw NoSuchMappingException("No ceiling entry in this
343 map");
344         }
345
346         NodePtr node = this->root;
347         std::stack<NodePtr> ancestors;
348
349         while (node != nullptr) {
350             if (key == node->key) {
351                 return node->entry();
352             }
353
354             if (compare(key, node->key)) {
355                 /* key < node->key */
356                 if (node->left != nullptr) {
357                     ancestors.push(node);
358                     node = node->left;
359                 } else {
360                     return node->entry();
361                 }

```

```

362     } else {
363         /* key > node->key */
364         if (node->right != nullptr) {
365             ancestors.push(node);
366             node = node->right;
367         } else {
368             if (ancestors.empty()) {
369                 throw NoSuchMappingException("No ceiling entry in
370 this map");
371             }
372
373             NodePtr parent = ancestors.top();
374             ancestors.pop();
375
376             while (node == parent->right) {
377                 node = parent;
378                 if (!ancestors.empty()) {
379                     parent = ancestors.top();
380                     ancestors.pop();
381                 } else {
382                     throw NoSuchMappingException("No ceiling entry in
383 this map");
384                 }
385             }
386
387             return parent->entry();
388         }
389     }
390 }
391
392 throw NoSuchMappingException("No ceiling entry in this
393 map");
394 }
395
396 /**
397  * Gets the entry corresponding to the specified key; if no
398  * such entry exists,
399  * returns the entry for the greatest key less than the
400  * specified key;
401  * if no such entry exists, a {@code NoSuchMappingException}
402  * will be thrown.
403  * @param key
404  * @return AVLTreeMap<Key, Value>::Entry
405  * @throws NoSuchMappingException
406  */
407 Entry getFloorEntry(K key) const {
408     if (this->root == nullptr) {
409         throw NoSuchMappingException("No floor entry exists in
410 this map");
411     }
412
413     NodePtr node = this->root;

```

```

414     std::stack<NodePtr> ancestors;
415
416     while (node != nullptr) {
417         if (key == node->key) {
418             return node->entry();
419         }
420
421         if (compare(key, node->key)) {
422             /* key < node->key */
423             if (node->left != nullptr) {
424                 ancestors.push(node);
425                 node = node->left;
426             } else {
427                 if (ancestors.empty()) {
428                     throw NoSuchMappingException("No floor entry exists
429 in this map");
430                 }
431
432                 NodePtr parent = ancestors.top();
433                 ancestors.pop();
434
435                 while (node == parent->left) {
436                     node = parent;
437                     if (!ancestors.empty()) {
438                         parent = ancestors.top();
439                         ancestors.pop();
440                     } else {
441                         throw NoSuchMappingException("No floor entry
442 exists in this map");
443                     }
444                 }
445
446                 return parent->entry();
447             }
448         } else {
449             /* key > node->key */
450             if (node->right != nullptr) {
451                 ancestors.push(node);
452                 node = node->right;
453             } else {
454                 return node->entry();
455             }
456         }
457     }
458
459     throw NoSuchMappingException("No floor entry exists in this
460 map");
461 }
462
463 /**
464  * Gets the entry for the least key greater than the specified
465  * key; if no such entry exists, returns the entry for the

```

```

466 least
467     * key greater than the specified key; if no such entry
468 exists,
469     * a {@code NoSuchElementException} will be thrown.
470     * @param key
471     * @return AVLTreeMap<Key, Value>::Entry
472     * @throws NoSuchElementException
473     */
474     Entry getHigherEntry(K key) {
475         if (this->root == nullptr) {
476             throw NoSuchElementException("No higher entry exists in
477 this map");
478         }
479
480         NodePtr node = this->root;
481         std::stack<NodePtr> ancestors;
482
483         while (node != nullptr) {
484             if (compare(key, node->key)) {
485                 /* key < node->key */
486                 if (node->left != nullptr) {
487                     ancestors.push(node);
488                     node = node->left;
489                 } else {
490                     return node->entry();
491                 }
492             } else {
493                 /* key >= node->key */
494                 if (node->right != nullptr) {
495                     ancestors.push(node);
496                     node = node->right;
497                 } else {
498                     if (ancestors.empty()) {
499                         throw NoSuchElementException("No higher entry exists
500 in this map");
501                     }
502
503                     NodePtr parent = ancestors.top();
504                     ancestors.pop();
505
506                     while (node == parent->right) {
507                         node = parent;
508                         if (!ancestors.empty()) {
509                             parent = ancestors.top();
510                             ancestors.pop();
511                         } else {
512                             throw NoSuchElementException(
513                                 "No higher entry exists in this map");
514                         }
515                     }
516
517                     return parent->entry();

```

```

518     }
519 }
520 }
521
522     throw NoSuchMappingException("No higher entry exists in this
523 map");
524 }
525
526 /**
527  * Returns the entry for the greatest key less than the
528  specified key; if
529  * no such entry exists (i.e., the least key in the Tree is
530  greater than
531  * the specified key), a NoSuchMappingException will
532  be thrown.
533  * @param key
534  * @return AvlTreeMap<Key, Value>::Entry
535  * @throws NoSuchMappingException
536  */
537 Entry getLowerEntry(K key) const {
538     if (this->root == nullptr) {
539         throw NoSuchMappingException("No lower entry exists in
540 this map");
541     }
542
543     NodePtr node = this->root;
544     std::stack<NodePtr> ancestors;
545
546     while (node != nullptr) {
547         if (compare(key, node->key) || key == node->key) {
548             /* key <= node->key */
549             if (node->left != nullptr) {
550                 ancestors.push(node);
551                 node = node->left;
552             } else {
553                 if (ancestors.empty()) {
554                     throw NoSuchMappingException("No lower entry exists
555 in this map");
556                 }
557
558                 NodePtr parent = ancestors.top();
559                 ancestors.pop();
560
561                 while (node == parent->left) {
562                     node = parent;
563                     if (!ancestors.empty()) {
564                         parent = ancestors.top();
565                         ancestors.pop();
566                     } else {
567                         throw NoSuchMappingException("No lower entry
568 exists in this map");
569                     }

```



```

570         }
571
572         return parent->entry();
573     }
574     } else {
575         /* key > node->key */
576         if (node->right != nullptr) {
577             ancestors.push(node);
578             node = node->right;
579         } else {
580             return node->entry();
581         }
582     }
583 }
584
585     throw NoSuchElementException("No lower entry exists in this
586 map");
587 }
588
589 /**
590  * Remove all entries that satisfy the filter condition.
591  * @param filter
592  */
593 void removeAll(KeyValueFilter filter) {
594     std::vector<Key> keys;
595     this->inorderTraversal([&](ConstNodePtr node) {
596         if (filter(node->key, node->value)) {
597             keys.push_back(node->key);
598         }
599     });
600     for (const Key &key : keys) {
601         this->remove(key);
602     }
603 }
604
605 /**
606  * Performs the given action for each key and value entry in
607 this map.
608  * The value is immutable for the action.
609  * @param action
610  */
611 void forEach(KeyValueConsumer action) const {
612     this->inorderTraversal(
613         [&](ConstNodePtr node) { action(node->key, node->value);
614     });
615 }
616
617 /**
618  * Performs the given action for each key and value entry in
619 this map.
620  * The value is mutable for the action.
621  * @param action

```

```

622     */
623     void forEachMut(MutKeyValueConsumer action) {
624         this->inorderTraversal(
625             [&](ConstNodePtr node) { action(node->key, node->value);
626         });
627     }
628
629     /**
630     * Returns a list containing all of the entries in this map.
631     * @return AvlTreeMap<Key, Value>::EntryList
632     */
633     EntryList toEntryList() const {
634         EntryList entryList;
635         this->inorderTraversal(
636             [&](ConstNodePtr node) { entryList.push_back(node-
637 >entry()); });
638         return entryList;
639     }
640
641     private:
642     static NodePtr rotateLeft(ConstNodePtr node) {
643         // clang-format off
644         //      |                      |
645         //      N                      S
646         //    / \    l-rotate(N)    / \
647         //   L  S    =====>   N  R
648         //    / \                / \
649         //   M  R                L  M
650         NodePtr successor = node->right;
651         // clang-format on
652         node->right = successor->left;
653         successor->left = node;
654
655         node->updateHeight();
656         successor->updateHeight();
657
658         return successor;
659     }
660
661     static NodePtr rotateRight(ConstNodePtr node) {
662         // clang-format off
663         //      |                      |
664         //      N                      S
665         //    / \    r-rotate(N)    / \
666         //   S  R    =====>   L  N
667         //    / \                / \
668         //   L  M                M  R
669         NodePtr successor = node->left;
670         // clang-format on
671         node->left = successor->right;
672         successor->right = node;
673

```

```

674     node->updateHeight();
675     successor->updateHeight();
676
677     return successor;
678 }
679
680 static void swapNode(NodePtr &lhs, NodePtr &rhs) {
681     std::swap(lhs->key, rhs->key);
682     std::swap(lhs->value, rhs->value);
683     std::swap(lhs, rhs);
684 }
685
686 static void fixBalance(NodePtr &node) {
687     if (node->factor() < -1) {
688         if (node->left->factor() < 0) {
689             // clang-format off
690             // Left-Left Case
691             //      |
692             //      C          |
693             //     /   r-rotate(C)   B
694             //    B   =====>   / \
695             //   /                A  C
696             //  A
697             // clang-format on
698             node = rotateRight(node);
699         } else {
700             // clang-format off
701             // Left-Right Case
702             //      |                |
703             //      C                C          |
704             //     /   l-rotate(A)   /   r-rotate(C)   B
705             //    A   =====>   B   =====>   / \
706             //   \                /                A  C
707             //    B                A
708             // clang-format on
709             node->left = rotateLeft(node->left);
710             node = rotateRight(node);
711         }
712     } else if (node->factor() > 1) {
713         if (node->right->factor() > 0) {
714             // clang-format off
715             // Right-Right Case
716             //      |
717             //      C          |
718             //     \   l-rotate(C)   B
719             //    B   =====>   / \
720             //   \                A  C
721             //    A
722             // clang-format on
723             node = rotateLeft(node);
724         } else {
725             // clang-format off

```

```

726         // Right-Left Case
727         //      |           |
728         //      A           A           |
729         //      \   r-rotate(C) \   l-rotate(A)   B
730         //      C =====>    B =====>    / \
731         //      /           \           A   C
732         //      B           C
733         // clang-format on
734         node->right = rotateRight(node->right);
735         node = rotateLeft(node);
736     }
737 }
738 }
739
740 NodePtr getNodeOrProvide(NodePtr &node, K key, NodeProvider
741 provide) {
742     assert(node != nullptr);
743
744     if (key == node->key) {
745         return node;
746     }
747
748     assert(key != node->key);
749
750     NodePtr result;
751
752     if (compare(key, node->key)) {
753         /* key < node->key */
754         if (node->left == nullptr) {
755             result = node->left = provide();
756             this->count += 1;
757             node->updateHeight();
758         } else {
759             result = getNodeOrProvide(node->left, key, provide);
760             node->updateHeight();
761             fixBalance(node);
762         }
763     } else {
764         /* key > node->key */
765         if (node->right == nullptr) {
766             result = node->right = provide();
767             this->count += 1;
768             node->updateHeight();
769         } else {
770             result = getNodeOrProvide(node->right, key, provide);
771             node->updateHeight();
772             fixBalance(node);
773         }
774     }
775
776     return result;
777 }

```

```

778
779     NodePtr getNode(ConstNodePtr node, K key) const {
780         assert(node != nullptr);
781
782         if (key == node->key) {
783             return node;
784         }
785
786         if (compare(key, node->key)) {
787             /* key < node->key */
788             return node->left == nullptr ? nullptr : getNode(node-
789 >left, key);
790         } else {
791             /* key > node->key */
792             return node->right == nullptr ? nullptr : getNode(node-
793 >right, key);
794         }
795     }
796
797     void insert(NodePtr &node, K key, V value, bool replace =
798 true) {
799         assert(node != nullptr);
800
801         if (key == node->key) {
802             if (replace) {
803                 node->value = value;
804             }
805             return;
806         }
807
808         assert(key != node->key);
809
810         if (compare(key, node->key)) {
811             /* key < node->key */
812             if (node->left == nullptr) {
813                 node->left = Node::from(key, value);
814                 this->count += 1;
815                 node->updateHeight();
816             } else {
817                 insert(node->left, key, value, replace);
818                 node->updateHeight();
819                 fixBalance(node);
820             }
821         } else {
822             /* key > node->key */
823             if (node->right == nullptr) {
824                 node->right = Node::from(key, value);
825                 this->count += 1;
826                 node->updateHeight();
827             } else {
828                 insert(node->right, key, value, replace);
829                 node->updateHeight();

```

```

830         fixBalance(node);
831     }
832 }
833 }
834
835 bool remove(NodePtr &node, K key, NodeConsumer action) {
836     assert(node != nullptr);
837
838     if (key != node->key) {
839         if (compare(key, node->key)) {
840             /* key < node->key */
841             NodePtr &left = node->left;
842             if (left != nullptr && remove(left, key, action)) {
843                 node->updateHeight();
844                 fixBalance(node);
845                 return true;
846             } else {
847                 return false;
848             }
849         } else {
850             /* key > node->key */
851             NodePtr &right = node->right;
852             if (right != nullptr && remove(right, key, action)) {
853                 node->updateHeight();
854                 fixBalance(node);
855                 return true;
856             } else {
857                 return false;
858             }
859         }
860     }
861
862     assert(key == node->key);
863     action(node);
864
865     if (node->isLeaf()) {
866         // Case 1: no child
867         node = nullptr;
868     } else if (node->right == nullptr) {
869         // clang-format off
870         // Case 2: left child only
871         //      P
872         //      | remove(N) P
873         //      N =====> |
874         //      /              L
875         //      L
876         // clang-format on
877         node = node->left;
878         node->updateHeight();
879     } else if (node->left == nullptr) {
880         // clang-format off
881         // Case 3: right child only

```

```

882 // P
883 // | remove(N) P
884 // N =====> |
885 // \ R
886 // R
887 // clang-format on
888 node = node->right;
889 node->updateHeight();
890 } else if (node->right->left == nullptr) {
891 // clang-format off
892 // Case 4: both left and right child, right child has no
893 left child
894 // | |
895 // N remove(N) R
896 // / \ =====> /
897 // L R L
898 // clang-format on
899 NodePtr right = node->right;
900 swapNode(node, right);
901 right->right = node->right;
902 node = right;
903 node->updateHeight();
904 fixBalance(node);
} else {
// clang-format off
// Case 5: both left and right child, right child is not a
leaf
// Step 1. find the node N with the smallest key
// and its parent P on the right subtree
// Step 2. swap S and N
// Step 3. remove node N like Case 1 or Case 3
// Step 4. update height for P
// | | |
// N S S
// / \ / \ / \
// L .. swap(N, S) L .. remove(N) L ..
// | =====> | =====> L ..
// P P P
// / \ / \ / \
// S .. N .. R ..
// \ \ \
// R R R
// clang-format on

// Step 1
NodePtr successor = node->right;
NodePtr parent = node;
while (successor->left != nullptr) {
    parent = successor;
    successor = parent->left;
}
// Step 2

```

```

        swapNode(node, successor);
        // Step 3
        parent->left = node->right;
        // Restore node
        node = successor;
        // Step 4
        parent->updateHeight();
    }

    this->count -= 1;
    return true;
}

void inorderTraversal(NodeConsumer action) const {
    if (this->root == nullptr) {
        return;
    }

    std::stack<NodePtr> stack;
    NodePtr node = this->root;

    while (node != nullptr || !stack.empty()) {
        while (node != nullptr) {
            stack.push(node);
            node = node->left;
        }
        if (!stack.empty()) {
            node = stack.top();
            stack.pop();
            action(node);
            node = node->right;
        }
    }
};

#endif // AVL_TREE_MAP_HPP

```

其他资料

在 [AVL Tree Visualization](#) 可以观察 AVL 树维护平衡的过程。

[维基百科 -- AVL 树](#)

🔧 本页面最近更新：2024/8/22 00:45:02，[更新历史](#)

✎ 发现错误？想一起完善？ [在 GitHub 上编辑此页！](#)

👤 本页面贡献者： [Ir1d](#), [Wajov](#), [Enter-tainer](#), [mgt](#), [Rlvance](#), [5ab-juruo](#), [ChungZH](#), [diauweb](#),

[GoodCoder666](#), [Great-designer](#), [iamtwz](#), [jimgreen2013](#), [Tiphereth-A](#), [Xeonacid](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用

