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。

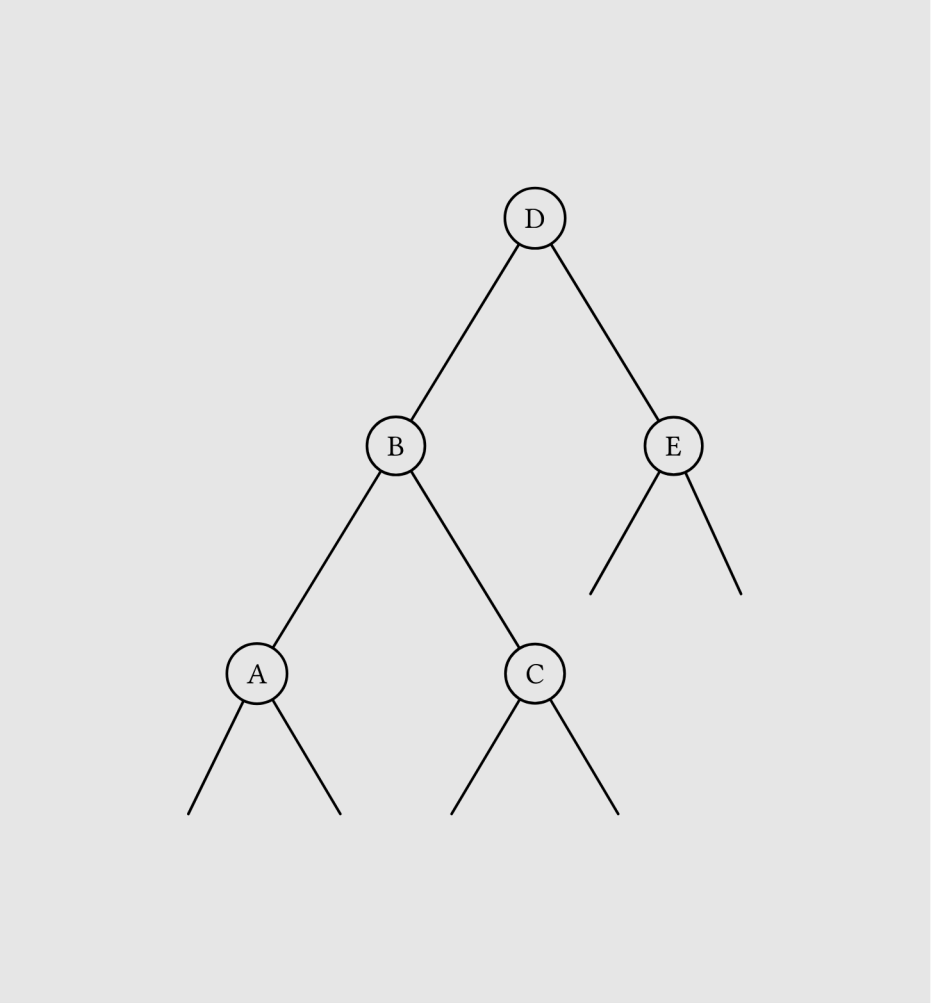
删除结点

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。

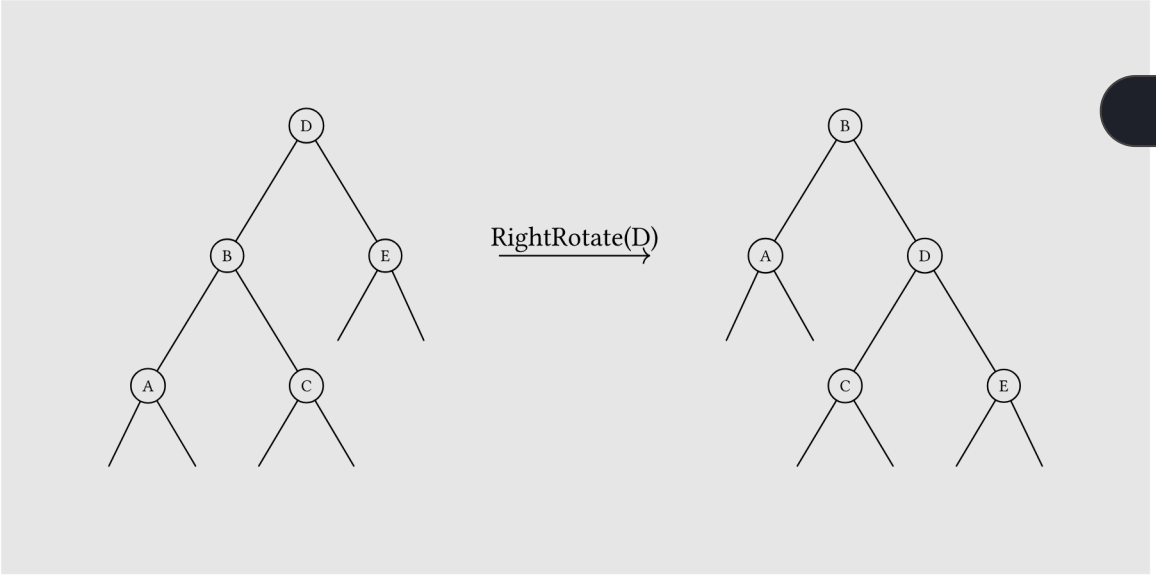


$h(A) \geq h(C)$

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

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

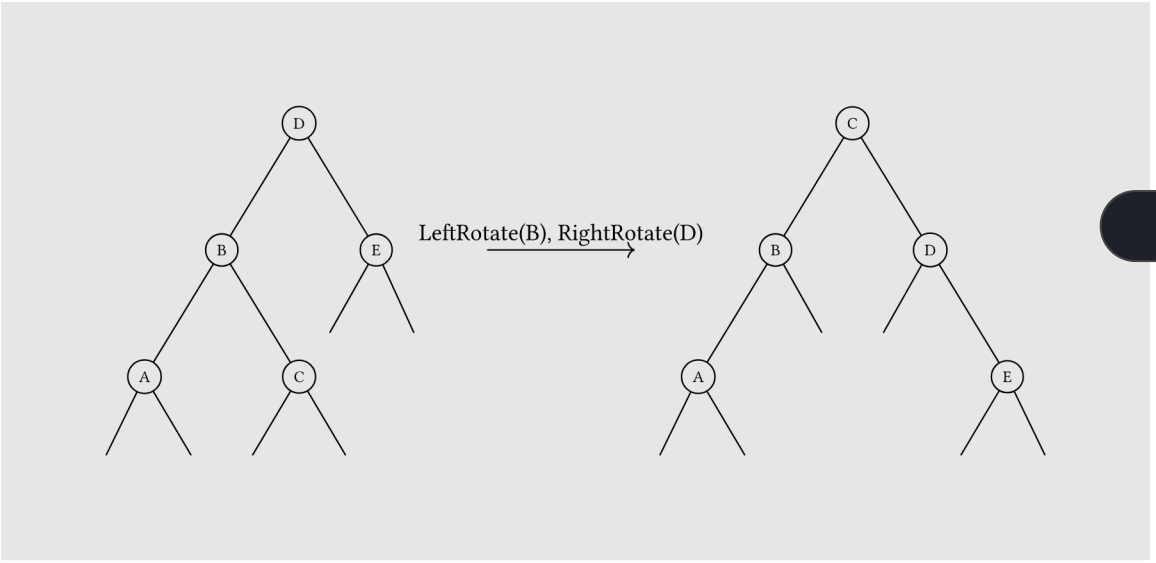
因此旋转后的节点 B 和 D 也满足性质 2。

$h(A) < h(C)$

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

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

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 ← ls_p, r ← rs_p
3       if h(l) − h(r) = 2
4           if h(ls_l) ≥ h(rs_l)
5               RightRotate(p)
6           else
7               LeftRotate(l)
8               RightRotate(p)
9       else if h(l) − h(r) = −2
10          if h(ls_r) ≤ h(rs_r)
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

```cpp
/**
 * @brief An AVLTree-based map implementation
 * @details The map is sorted according to the natural ordering of its
 *  keys or by a {@code Compare} function provided; This implementation
 *  provides guaranteed log(n) time cost for the contains, get, insert
 *  and remove operations.
 */

#ifndef AVLTREE_MAP_HPP
#define AVLTREE_MAP_HPP

#include <cassert>
#include <cstddef>
#include <cstdint>
#include <functional>
#include <memory>
#include <stack>
#include <utility>
#include <vector>

/**
 * An AVLTree-based map implementation
 * https://en.wikipedia.org/wiki/AVL_tree
 * @tparam Key the type of keys maintained by this map
 * @tparam Value the type of mapped values
 * @tparam Compare
 */
template <typename Key, typename Value, typename Compare =
std::less<Key> >
class AvlTreeMap {
 private:
  using USize = size_t;
  using Factor = int64_t;

  Compare compare = Compare();

 public:
  struct Entry {
    Key key;
    Value value;

    bool operator==(const Entry &rhs) const noexcept {
      return this->key == rhs.key && this->value == rhs.value;
    }

    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);
```

```
      }
    }

    inline Entry entry() const { return Entry{key, value}; }

    static Ptr from(const Key &k) { return
std::make_shared<Node>(Node(k)); }

    static Ptr from(const Key &k, const Value &v) {
      return std::make_shared<Node>(Node(k, v));
    }
  };

  using NodePtr = typename Node::Ptr;
  using ConstNodePtr = const NodePtr &;
  using NodeProvider = typename Node::Provider;
  using NodeConsumer = typename Node::Consumer;

  NodePtr root = nullptr;
  USize count = 0;

  using K = const Key &;
  using V = const Value &;

 public:
  using EntryList = std::vector<Entry>;
  using KeyValueConsumer = const std::function<void(K, V)> &;
  using MutKeyValueConsumer = const std::function<void(K, Value
&)> &;
  using KeyValueFilter = const std::function<bool(K, V)> &;

  class NoSuchMappingException : protected std::exception {
   private:
    const char *message;

   public:
    explicit NoSuchMappingException(const char *msg) :
message(msg) {}

    const char *what() const noexcept override { return message;
}
  };

  AvlTreeMap() noexcept = default;

  /**
   * Returns the number of entries in this map.
   * @return size_t
   */
  inline USize size() const noexcept { return this->count; }

  /**
```

```
    * Returns true if this collection contains no elements.
    * @return bool
    */
  inline bool empty() const noexcept { return this->count == 0;
}

  /**
   * Removes all of the elements from this map.
   */
  void clear() noexcept {
    this->root = nullptr;
    this->count = 0;
  }

  /**
   * Returns the value to which the specified key is mapped; If
this map
   * contains no mapping for the key, a {@code
NoSuchMappingException} will
   * be thrown.
   * @param key
   * @return AvlTreeMap<Key, Value>::Value
   * @throws NoSuchMappingException
   */
  Value get(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("Invalid key");
    } else {
      NodePtr node = this->getNode(this->root, key);
      if (node != nullptr) {
        return node->value;
      } else {
        throw NoSuchMappingException("Invalid key");
      }
    }
  }

  /**
   * Returns the value to which the specified key is mapped; If
this map
   * contains no mapping for the key, a new mapping with a
default value
   * will be inserted.
   * @param key
   * @return AvlTreeMap<Key, Value>::Value &
   */
  Value &getOrDefault(K key) {
    if (this->root == nullptr) {
      this->root = Node::from(key);
      this->count += 1;
      return this->root->value;
    } else {
```

```
      return this
          ->getNodeOrProvide(this->root, key,
                             [&key]() { return Node::from(key);
})
          ->value;
    }
  }

  /**
   * Returns true if this map contains a mapping for the
specified key.
   * @param key
   * @return bool
   */
  bool contains(K key) const {
    return this->getNode(this->root, key) != nullptr;
  }

  /**
   * Associates the specified value with the specified key in
this map.
   * @param key
   * @param value
   */
  void insert(K key, V value) {
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
      this->count += 1;
    } else {
      this->insert(this->root, key, value);
    }
  }

  /**
   * If the specified key is not already associated with a
value, associates
   * it with the given value and returns true, else returns
false.
   * @param key
   * @param value
   * @return bool
   */
  bool insertIfAbsent(K key, V value) {
    USize sizeBeforeInsertion = this->size();
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
      this->count += 1;
    } else {
      this->insert(this->root, key, value, false);
    }
    return this->size() > sizeBeforeInsertion;
  }
```

```
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 NoSuchMappingException} will be thrown.
307    * @param key
308    * @return AvlTreeMap<Key, Value>::Value
309    * @throws NoSuchMappingException
```

```
    */
   Value getAndRemove(K key) {
     Value result;
     NodeConsumer action = [&](ConstNodePtr node) { result =
node->value; };

     if (root == nullptr) {
       throw NoSuchMappingException("Invalid key");
     } else {
       if (remove(this->root, key, action)) {
         return result;
       } else {
         throw NoSuchMappingException("Invalid key");
       }
     }
   }

   /**
    * Gets the entry corresponding to the specified key; if no
such entry
    * exists, returns the entry for the least key greater than
the specified
    * key; if no such entry exists (i.e., the greatest key in the
Tree is less
    * than the specified key), a {@code NoSuchMappingException}
will be thrown.
    * @param key
    * @return AvlTreeMap<Key, Value>::Entry
    * @throws NoSuchMappingException
    */
   Entry getCeilingEntry(K key) const {
     if (this->root == nullptr) {
       throw NoSuchMappingException("No ceiling entry in this
map");
     }

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

     while (node != nullptr) {
       if (key == node->key) {
         return node->entry();
       }

       if (compare(key, node->key)) {
         /* key < node->key */
         if (node->left != nullptr) {
           ancestors.push(node);
           node = node->left;
         } else {
           return node->entry();
         }
```

```
      } else {
        /* key > node->key */
        if (node->right != nullptr) {
          ancestors.push(node);
          node = node->right;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No ceiling entry in
this map");
          }

          NodePtr parent = ancestors.top();
          ancestors.pop();

          while (node == parent->right) {
            node = parent;
            if (!ancestors.empty()) {
              parent = ancestors.top();
              ancestors.pop();
            } else {
              throw NoSuchMappingException("No ceiling entry in
this map");
            }
          }

          return parent->entry();
        }
      }
    }

    throw NoSuchMappingException("No ceiling entry in this
map");
  }

  /**
   * Gets the entry corresponding to the specified key; if no
such entry exists,
   * returns the entry for the greatest key less than the
specified key;
   * if no such entry exists, a {@code NoSuchMappingException}
will be thrown.
   * @param key
   * @return AvlTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getFloorEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No floor entry exists in
this map");
    }

    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
```

```
least
   * key greater than the specified key; if no such entry
exists,
   * a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return AvlTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getHigherEntry(K key) {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No higher entry exists in
this map");
    }

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

    while (node != nullptr) {
      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
          return node->entry();
        }
      } else {
        /* key >= node->key */
        if (node->right != nullptr) {
          ancestors.push(node);
          node = node->right;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No higher entry exists
in this map");
          }

          NodePtr parent = ancestors.top();
          ancestors.pop();

          while (node == parent->right) {
            node = parent;
            if (!ancestors.empty()) {
              parent = ancestors.top();
              ancestors.pop();
            } else {
              throw NoSuchMappingException(
                  "No higher entry exists in this map");
            }
          }

          return parent->entry();
```

```
        }
      }
    }

    throw NoSuchMappingException("No higher entry exists in this
map");
  }

  /**
   * Returns the entry for the greatest key less than the
specified key; if
   * no such entry exists (i.e., the least key in the Tree is
greater than
   * the specified key), a {@code NoSuchMappingException} will
be thrown.
   * @param key
   * @return AvlTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getLowerEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No lower entry exists in
this map");
    }

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

    while (node != nullptr) {
      if (compare(key, node->key) || key == node->key) {
        /* key <= node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No lower entry exists
in this map");
          }

          NodePtr parent = ancestors.top();
          ancestors.pop();

          while (node == parent->left) {
            node = parent;
            if (!ancestors.empty()) {
              parent = ancestors.top();
              ancestors.pop();
            } else {
              throw NoSuchMappingException("No lower entry
exists in this map");
            }
```

```
            }

            return parent->entry();
          }
        } else {
          /* key > node->key */
          if (node->right != nullptr) {
            ancestors.push(node);
            node = node->right;
          } else {
            return node->entry();
          }
        }
      }

      throw NoSuchMappingException("No lower entry exists in this
map");
    }

    /**
     * Remove all entries that satisfy the filter condition.
     * @param filter
     */
    void removeAll(KeyValueFilter filter) {
      std::vector<Key> keys;
      this->inorderTraversal([&](ConstNodePtr node) {
        if (filter(node->key, node->value)) {
          keys.push_back(node->key);
        }
      });
      for (const Key &key : keys) {
        this->remove(key);
      }
    }

    /**
     * Performs the given action for each key and value entry in
this map.
     * The value is immutable for the action.
     * @param action
     */
    void forEach(KeyValueConsumer action) const {
      this->inorderTraversal(
          [&](ConstNodePtr node) { action(node->key, node->value);
});
    }

    /**
     * Performs the given action for each key and value entry in
this map.
     * The value is mutable for the action.
     * @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  ..
          //       P                   P               |
          //      / \                 / \               P
          //     S  ..               N  ..             / \
          //      \                   \               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  // AVLTREE_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