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。

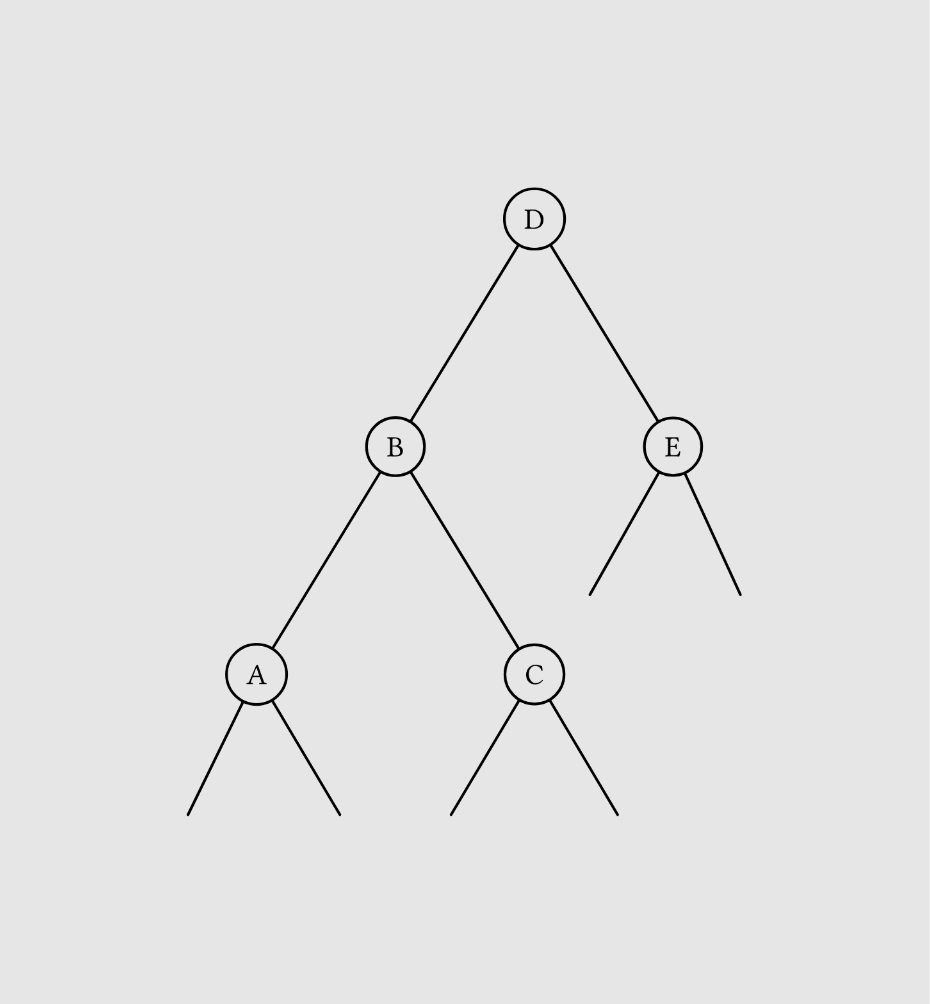
删除结点

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。

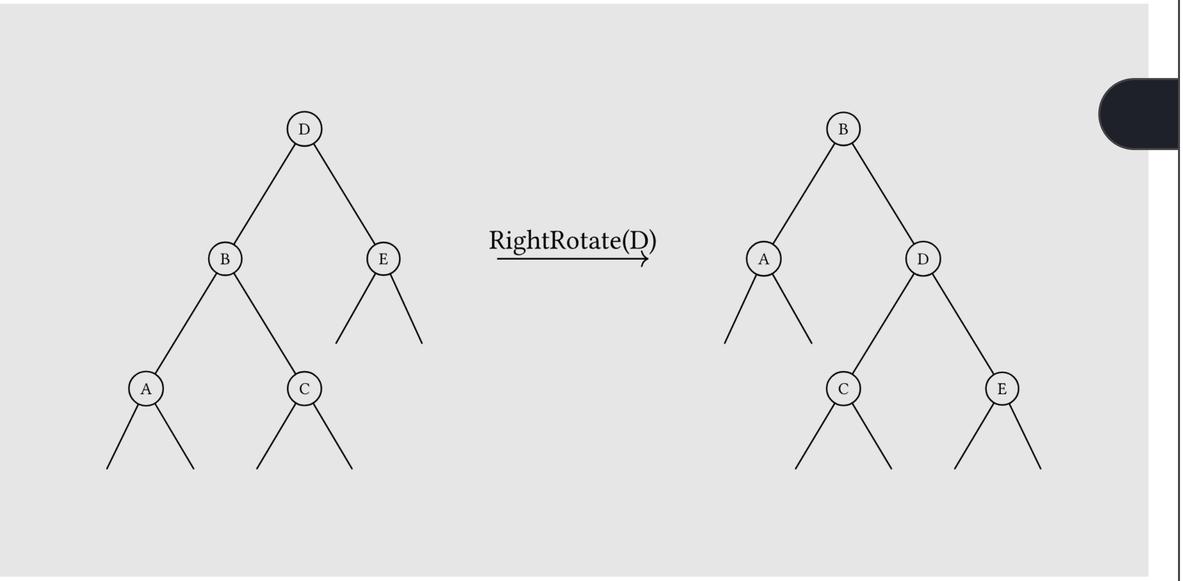


$h(A)\geq h(C)$

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

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

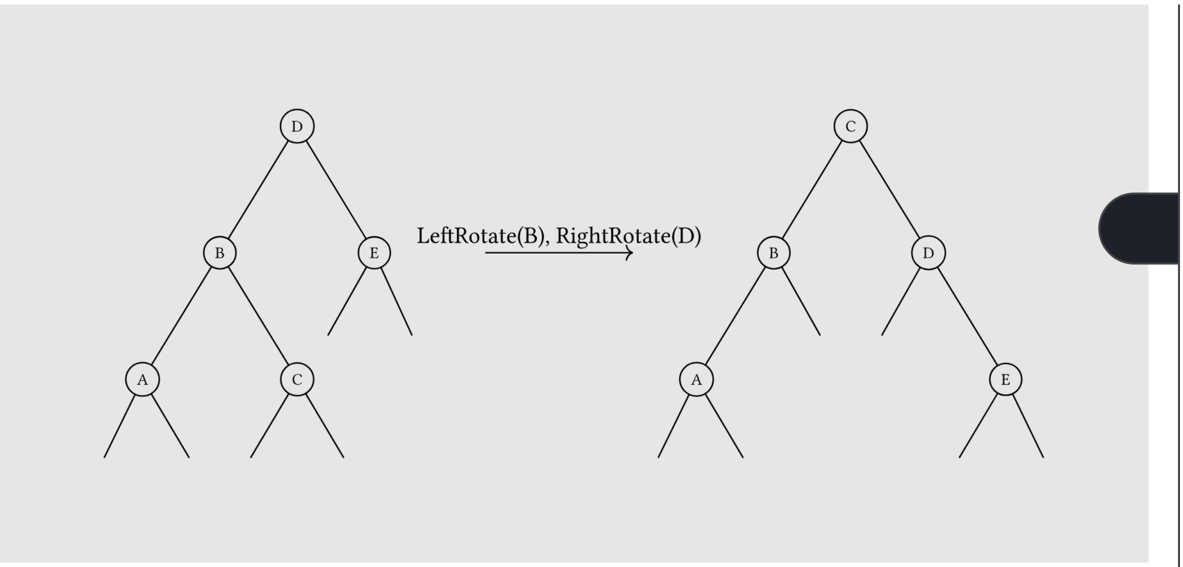
因此旋转后的节点 B 和 D 也满足性质 2。

$h(A) < h(C)$

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

```
function MaintainBalance($p$)
    $l \leftarrow ls_p, r \leftarrow rs_p$
    if $h(l) - h(r) = 2$
        if $h(ls_l) \geq h(rs_l)$
            RightRotate($p$)
        else
            LeftRotate($l$)
            RightRotate($p$)
    else if $h(l) - h(r) = -2$
        if $h(ls_r) \leq h(rs_r)$
            LeftRotate($p$)
        else
            RightRotate($r$)
            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 {
      return this->key != rhs.key || this->value != rhs.value;
    }
  };

 private:
  struct Node {
    using Ptr = std::shared_ptr<Node>;
    using Provider = const std::function<Ptr(void)> &;
    using Consumer = const std::function<void(const Ptr &)> &;

    Key key;
    Value value{};

    Ptr left = nullptr;
    Ptr right = nullptr;

    USize height = 1;

    explicit Node(Key k) : key(std::move(k)) {}

    explicit Node(Key k, Value v) : key(std::move(k)),
value(std::move(v)) {}

    ~Node() = default;

    inline bool isLeaf() const noexcept {
      return this->left == nullptr && this->right == nullptr;
    }

    inline void updateHeight() noexcept {
      if (this->isLeaf()) {
        this->height = 1;
      } else if (this->left == nullptr) {
        this->height = this->right->height + 1;
      } else if (this->right == nullptr) {
        this->height = this->left->height + 1;
      } else {
        this->height = std::max(left->height, right->height) +
1;
      }
    }

    inline Factor factor() const noexcept {
      if (this->isLeaf()) {
        return 0;
      } else if (this->left == nullptr) {
        return (Factor)this->right->height;
      } else if (this->right == nullptr) {
        return (Factor) - this->left->height;
      } else {
        return (Factor)(this->right->height - this->left-
>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;
  }

  /**
   * If the specified key is not already associated with a
value, associates
   * it with the given value and returns the value, else returns
the associated
   * value.
   * @param key
   * @param value
   * @return
   */
  Value &getOrInsert(K key, V value) {
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
      this->count += 1;
      return root->value;
    } else {
      NodePtr node = getNodeOrProvide(this->root, key,
                                      [&]() { return
Node::from(key, value); });
      return node->value;
    }
  }

  Value operator[](K key) const { return this->get(key); }

  Value &operator[](K key) { return this->getOrDefault(key); }

  /**
   * Removes the mapping for a key from this map if it is
present;
   * Returns true if the mapping is present else returns false
   * @param key the key of the mapping
   * @return bool
   */
  bool remove(K key) {
    if (this->root == nullptr) {
      return false;
    } else {
      return this->remove(this->root, key, [](ConstNodePtr) {});
    }
  }

  /**
   * Removes the mapping for a key from this map if it is
present and returns
   * the value which is mapped to the key; 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 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;
```

```
    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 {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No floor 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 floor 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 floor entry exists in this
map");
  }

  /**
   * Gets the entry for the least key greater than 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,
   * 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
   */
  void forEachMut(MutKeyValueConsumer action) {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value);
});
  }

  /**
   * Returns a list containing all of the entries in this map.
   * @return AvlTreeMap<Key, Value>::EntryList
   */
  EntryList toEntryList() const {
    EntryList entryList;
    this->inorderTraversal(
        [&](ConstNodePtr node) { entryList.push_back(node-
>entry()); });
    return entryList;
  }

 private:
  static NodePtr rotateLeft(ConstNodePtr node) {
    // clang-format off
    //     |                       |
    //     N                       S
    //    / \     l-rotate(N)     / \
    //   L   S    ==========>    N   R
    //      / \                 / \
    //     M   R               L   M
    NodePtr successor = node->right;
    // clang-format on
    node->right = successor->left;
    successor->left = node;

    node->updateHeight();
    successor->updateHeight();

    return successor;
  }

  static NodePtr rotateRight(ConstNodePtr node) {
    // clang-format off
    //       |                   |
    //       N                   S
    //      / \   r-rotate(N)   / \
    //     S   R  ==========>  L   N
    //    / \                     / \
    //   L   M                   M   R
    NodePtr successor = node->left;
    // clang-format on
    node->left = successor->right;
    successor->right = node;

    node->updateHeight();
    successor->updateHeight();

    return successor;
  }

  static void swapNode(NodePtr &lhs, NodePtr &rhs) {
    std::swap(lhs->key, rhs->key);
    std::swap(lhs->value, rhs->value);
    std::swap(lhs, rhs);
  }

  static void fixBalance(NodePtr &node) {
    if (node->factor() < -1) {
      if (node->left->factor() < 0) {
        // clang-format off
        //  Left-Left Case
        //       |
        //       C                 |
        //      /   r-rotate(C)    B
        //     B    ==========>   / \
        //    /                  A   C
        //   A
        // clang-format on
        node = rotateRight(node);
      } else {
        // clang-format off
        //  Left-Right Case
        //     |                   |
        //     C                   C                 |
        //    /   l-rotate(A)     /   r-rotate(C)    B
        //   A    ==========>    B    ==========>   / \
        //    \                 /                  A   C
        //     B               A
        // clang-format on
        node->left = rotateLeft(node->left);
        node = rotateRight(node);
      }
    } else if (node->factor() > 1) {
      if (node->right->factor() > 0) {
        // clang-format off
        //  Right-Right Case
        //   |
        //   C                     |
        //    \     l-rotate(C)    B
        //     B    ==========>   / \
        //      \                A   C
        //       A
        // clang-format on
        node = rotateLeft(node);
      } else {
        // clang-format off
        //  Right-Left Case
        //   |                 |
        //   A                 A                     |
        //    \   r-rotate(C)   \     l-rotate(A)    B
        //     C  ==========>    B    ==========>   / \
        //    /                   \                A   C
        //   B                     C
        // clang-format on
        node->right = rotateRight(node->right);
        node = rotateLeft(node);
      }
    }
  }

  NodePtr getNodeOrProvide(NodePtr &node, K key, NodeProvider
provide) {
    assert(node != nullptr);

    if (key == node->key) {
      return node;
    }

    assert(key != node->key);

    NodePtr result;

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left == nullptr) {
        result = node->left = provide();
        this->count += 1;
        node->updateHeight();
      } else {
        result = getNodeOrProvide(node->left, key, provide);
        node->updateHeight();
        fixBalance(node);
      }
    } else {
      /* key > node->key */
      if (node->right == nullptr) {
        result = node->right = provide();
        this->count += 1;
        node->updateHeight();
      } else {
        result = getNodeOrProvide(node->right, key, provide);
        node->updateHeight();
        fixBalance(node);
      }
    }

    return result;
  }

  NodePtr getNode(ConstNodePtr node, K key) const {
    assert(node != nullptr);

    if (key == node->key) {
      return node;
    }

    if (compare(key, node->key)) {
      /* key < node->key */
      return node->left == nullptr ? nullptr : getNode(node->left, key);
    } else {
      /* key > node->key */
      return node->right == nullptr ? nullptr : getNode(node->right, key);
    }
  }

  void insert(NodePtr &node, K key, V value, bool replace = true) {
    assert(node != nullptr);

    if (key == node->key) {
      if (replace) {
        node->value = value;
      }
      return;
    }

    assert(key != node->key);

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left == nullptr) {
        node->left = Node::from(key, value);
        this->count += 1;
        node->updateHeight();
      } else {
        insert(node->left, key, value, replace);
        node->updateHeight();
        fixBalance(node);
      }
    } else {
      /* key > node->key */
      if (node->right == nullptr) {
        node->right = Node::from(key, value);
        this->count += 1;
        node->updateHeight();
      } else {
        insert(node->right, key, value, replace);
        node->updateHeight();
        fixBalance(node);
      }
    }
  }

  bool remove(NodePtr &node, K key, NodeConsumer action) {
    assert(node != nullptr);

    if (key != node->key) {
      if (compare(key, node->key)) {
        /* key < node->key */
        NodePtr &left = node->left;
        if (left != nullptr && remove(left, key, action)) {
          node->updateHeight();
          fixBalance(node);
          return true;
        } else {
          return false;
        }
      } else {
        /* key > node->key */
        NodePtr &right = node->right;
        if (right != nullptr && remove(right, key, action)) {
          node->updateHeight();
          fixBalance(node);
          return true;
        } else {
          return false;
        }
      }
    }

    assert(key == node->key);
    action(node);

    if (node->isLeaf()) {
      // Case 1: no child
      node = nullptr;
    } else if (node->right == nullptr) {
      // clang-format off
      // Case 2: left child only
      //     P
      //     |  remove(N)  P
      //     N  ========>  |
      //    /              L
      //   L
      // clang-format on
      node = node->left;
      node->updateHeight();
    } else if (node->left == nullptr) {
      // clang-format off
      // Case 3: right child only
      //   P
      //   |    remove(N)  P
      //   N    ========>  |
      //    \              R
      //     R
      // clang-format on
      node = node->right;
      node->updateHeight();
    } else if (node->right->left == nullptr) {
      // clang-format off
      // Case 4: both left and right child, right child has no
left child
      //    |                 |
      //    N    remove(N)    R
      //   / \   ========>   /
      //  L   R             L
      // clang-format on
      NodePtr right = node->right;
      swapNode(node, right);
      right->right = node->right;
      node = right;
      node->updateHeight();
      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

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, RIvance, 5ab-juruo, ChungZH, diauweb, GoodCoder666, Great-designer, iamtwz, jimgreen2013, Tiphereth-A, Xeonacid