附录与参考资料

APPENDIX AND REFERENCES

生成每份报告基础格式的 python 代码:

```python
import os

def load_typ_file(rep_dir):
    typ_file_path = os.path.join(rep_dir, 'template.typ')
    with open(typ_file_path, 'r') as file:
        typ_content = file.read()
    return typ_content

def read_dev_files(dev_dir):
    file_data = {}
    for subdir, _, files in os.walk(dev_dir):
        for file_name in files:
            if file_name.endswith(('.cpp', '.h', '.hpp')):
                file_path = os.path.join(subdir, file_name)
                with open(file_path, 'r') as file:
                    content = file.read()
                file_data[file_path] = content
    return file_data
def read_dev_files(dev_dir):
    all_data = {}
    for subdir, _, files in os.walk(dev_dir):
        subdir_data = {}
        for file_name in files:
            if file_name.endswith(('.cpp', '.h', '.hpp')):
                file_path = os.path.join(subdir, file_name)
                with open(file_path, 'r') as file:
                    content = file.read()
                subdir_data[file_name] = content
        if subdir_data:
            all_data[subdir] = subdir_data
    return all_data
import re

def split_text(text):
    pattern = r'/\*.*\*/'
    matches = re.findall(pattern, text, re.DOTALL)

    if matches:
        last_match = matches[-1]
        remaining_text = text.replace(last_match, '', 1)
        last_match = last_match.replace("/*","")
        last_match = last_match.replace("*/","")
        return remaining_text, last_match
```

```
    else:
        return text,""
s = '''       rowspanx(8)[STL 风格的\ 泛型的\ 基础数据结构\ 容器实现],[1],[基于双向链表的`linkedList`],
      (),[2],[基于增长数组的`vector`],
      (),[3],[基于块状数组的`dataBlock`],
      (),[4],[实现基于循环增长数组的`deque`],
      (),[5],[基于`vector`实现`stack`],
      (),[10],[基于 R-BTree 实现`set`],
      (),[11],[基于 R-BTree 实现`map`],
      (),[15],[基于 Heap 实现`priority_queue`],
      rowspanx(4)[基础树/图结构],[6],[树上 dfs（基础信息）],
      (),[7],[图上 bfs（最短路）],
      (),[8],[二叉树三序遍历],
      (),[9],[R-BTree 的基本实现],
      rowspanx(4)[特殊结构\ 及其应用],[12],[字典树`Trie`],
      (),[13],[线段树`segTree`],
      (),[14],[堆`Heap`],
      (),[16],[霍夫曼树`Huffman-tree`],
      rowspanx(3)[在算法中应用],[17],[算数表达式求值（栈）],
      (),[18],[括号匹配（栈）],
      (),[19],[高精度计算], '''
t = s.split("\n");
m = {};
for x in t:
    text = "rowspanx(8)[STL 风格的\\ 泛型的\\ 基础数据结构\\ 容器实现],[1],[基于双向链表的`linkedList`]"

    # 定义正则表达式模式
    pattern = re.compile(r'\[([^\]]+)\]')
    matches = pattern.findall(x)
    m[int(matches[-2])] = matches[-1];
#     print(matches)
m
rep_dir = 'Rep'
dev_dir = 'Dev'

# Load Rep/0.typ file
template = load_typ_file(rep_dir)
# print(f"Loaded 0.typ content:\n{typ_content}")

# Read files in Dev subdirectories
file_data = read_dev_files(dev_dir)
```

```
# Print the loaded file names and content (for demonstration purposes)
for file_path, sub_paths in file_data.items():
    print(f"File: {file_path}")
    MAINCODE = ""
    COMMENTS = ""
    for sub_path,text in sub_paths.items():
        code,comment = split_text(text);
        MAINCODE += "== `" + sub_path + "`\n"
        MAINCODE += "#sourcecode[
```

cppn" + code + "n

```
]\n"
if len(comment) > 0 :
    COMMENTS += "
```

n" + comment + "n

```
\n"
    subdir_name = os.path.basename(file_path);
    nt = template
    nt = nt.replace("MAINCODE",MAINCODE);
    nt = nt.replace("TESTCASES",COMMENTS);
    nt = nt.replace("maintitle",m[int(subdir_name)])
    nt = nt.replace("INDEXS",subdir_name);
    # print(m[int(subdir_name)])
    # if(int(subdir_name) <= 18):
    #     continue
    rep_file_path = os.path.join(rep_dir, f"{subdir_name}.typ")
    f = open(rep_file_path,'w')
    f.write(nt)
    # print(rep_file_path);
    # print(f"Content:\n{content}")
    # print('-' * 80)
```

渲染报告用 typst 模板:

```
#import "@preview/tablex:0.0.6": tablex, hlinex, vlinex, colspanx, rowspanx
#import "@preview/codelst:2.0.1": sourcecode
// Display inline code in a small box
// that retains the correct baseline.
```

```
#set text(font:("Times New Roman","Source Han Serif SC"))
#show raw: set text(
    font: ("consolas", "Source Han Serif SC")
  )
#set page(
  paper: "a4",
)
#set text(
    font:("Times New Roman","Source Han Serif SC"),
    style:"normal",
    weight: "regular",
    size: 13pt,
)

#let nxtIdx(name) = box[ #counter(name).step()#counter(name).display()]
#set math.equation(numbering: "(1)")

#show raw.where(block: true): block.with(
  fill: luma(240),
  inset: 10pt,
  radius: 4pt,
)

#set math.equation(numbering: "(1)")

#set page(
  paper:"a4",
  number-align: right,
  margin: (x:2.54cm,y:4cm),
  header: [
    #set text(
      size: 25pt,
      font: "KaiTi",
    )
    #align(
      bottom + center,
      [ #strong[暨南大学本科实验报告专用纸(附页)] ]
    )
    #line(start: (0pt,-5pt),end:(453pt,-5pt))
  ]
)

/*----*/
```

```
= maintitle
\
#text(
  font:"KaiTi",
  size: 15pt
)[
课程名称 #underline[#text("      数据结构      ")]成绩评定 #underline[#text("
")]\
实验项目名称 #underline[#text(" ") maintitle #text(" ")]指导老师
#underline[#text("   干晓聪   ")]\
实验项目编号 #underline[#text("  INDEXS  ")]实验项目类型 #underline[#text("
设计性  ")]实验地点 #underline[#text(" 数学系机房 ")]\
学生姓名 #underline[#text("   郭彦培   ")]学号 #underline[#text("   2022101149
")]\
学院 #underline[#text(" 信息科学技术学院 ")]系 #underline[#text(" 数学系 ")]专
业 #underline[#text(" 信息管理与信息系统 ")]\
实验时间 #underline[#text(" 2024 年 6 月 13 日上午
")]#text("~")#underline[#text("  2024 年 7 月 13 日中午  ")]\
]
#set heading(
  numbering: "1.1."
  )


= 实验目的



= 实验环境

计算机：PC X64

操作系统：Windows + Ubuntu20.0LTS

编程语言：C++：GCC std20

IDE：Visual Studio Code


= 程序原理
\

#pagebreak()
```

= 程序代码

MAINCODE

= 测试数据与运行结果

运行上述`_PRIV_TEST.cpp`测试代码中的正确性测试模块，得到以下内容：

TESTCASES

可以看出，代码运行结果与预期相符，可以认为代码正确性无误。

用作参考的 RB_tree 实现（CPP STL）

```
#ifndef RBTREE_MAP_HPP
#define RBTREE_MAP_HPP

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

template <typename Key, typename Value, typename Compare = std::less<Key> >
class RBTreeMap {
 private:
    using USize = size_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 &)> &;

        enum { RED, BLACK } color = RED;

        enum Direction { LEFT = -1, ROOT = 0, RIGHT = 1 };

        Key key;
        Value value{};

        Ptr parent = nullptr;
        Ptr left = nullptr;
        Ptr right = nullptr;

        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 bool isRoot() const noexcept { return this->parent ==
nullptr; }

        inline bool isRed() const noexcept { return this->color == RED; }

        inline bool isBlack() const noexcept { return this->color ==
BLACK; }
```

```cpp
    inline Direction direction() const noexcept {
        if (this->parent != nullptr) {
            if (this == this->parent->left.get()) {
                return Direction::LEFT;
            } else {
                return Direction::RIGHT;
            }
        } else {
            return Direction::ROOT;
        }
    }

    inline Ptr &sibling() const noexcept {
        assert(!this->isRoot());
        if (this->direction() == LEFT) {
            return this->parent->right;
        } else {
            return this->parent->left;
        }
    }

    inline bool hasSibling() const noexcept {
        return !this->isRoot() && this->sibling() != nullptr;
    }

    inline Ptr &uncle() const noexcept {
        assert(this->parent != nullptr);
        return parent->sibling();
    }

    inline bool hasUncle() const noexcept {
        return !this->isRoot() && this->parent->hasSibling();
    }

    inline Ptr &grandParent() const noexcept {
        assert(this->parent != nullptr);
        return this->parent->parent;
    }

    inline bool hasGrandParent() const noexcept {
        return !this->isRoot() && this->parent->parent != nullptr;
    }

    inline void release() noexcept {
```

```
            // avoid memory leak caused by circular reference
            this->parent = nullptr;
            if (this->left != nullptr) {
                this->left->release();
            }
            if (this->right != nullptr) {
                this->right->release();
            }
        }

        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 Direction = typename Node::Direction;
    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; }
    };

    RBTreeMap() noexcept = default;

    ~RBTreeMap() noexcept {
        // Unlinking circular references to avoid memory leak
        this->clear();
    }

    /**
     * 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 {

        if (this->root != nullptr) {
            this->root->release();
            this->root = nullptr;
        }
        this->count = 0;
    }

    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");
```

```
            }
        }
    }

    Value &getOrDefault(K key) {
        if (this->root == nullptr) {
            this->root = Node::from(key);
            this->root->color = Node::BLACK;
            this->count += 1;
            return this->root->value;
        } else {
            return this
                    ->getNodeOrProvide(this->root, key,
                                                                    [&key]() { return
Node::from(key); })
                    ->value;
        }
    }

    bool contains(K key) const {
        return this->getNode(this->root, key) != nullptr;
    }

    void insert(K key, V value) {
        if (this->root == nullptr) {
            this->root = Node::from(key, value);
            this->root->color = Node::BLACK;
            this->count += 1;
        } else {
            this->insert(this->root, key, value);
        }
    }

    bool insertIfAbsent(K key, V value) {
        USize sizeBeforeInsertion = this->size();
        if (this->root == nullptr) {
            this->root = Node::from(key, value);
            this->root->color = Node::BLACK;
            this->count += 1;
        } else {
            this->insert(this->root, key, value, false);
        }
        return this->size() > sizeBeforeInsertion;
    }
```

```
    Value &getOrInsert(K key, V value) {
        if (this->root == nullptr) {
            this->root = Node::from(key, value);
            this->root->color = Node::BLACK;
            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); }

    bool remove(K key) {
        if (this->root == nullptr) {
            return false;
        } else {
            return this->remove(this->root, key, [](ConstNodePtr) {});
        }
    }

    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");
            }
        }
    }

    Entry getCeilingEntry(K key) const {
```

```
        if (this->root == nullptr) {
            throw NoSuchMappingException("No ceiling entry in this map");
        }

        NodePtr node = this->root;

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

            if (compare(key, node->key)) {
                /* key < node->key */
                if (node->left != nullptr) {
                    node = node->left;
                } else {
                    return node->entry();
                }
            } else {
                /* key > node->key */
                if (node->right != nullptr) {
                    node = node->right;
                } else {
                    while (node->direction() == Direction::RIGHT) {
                        if (node != nullptr) {
                            node = node->parent;
                        } else {
                            throw NoSuchMappingException(
                                    "No ceiling entry exists in this map");
                        }
                    }
                    if (node->parent == nullptr) {
                        throw NoSuchMappingException("No ceiling entry
exists in this map");
                    }
                    return node->parent->entry();
                }
            }
        }

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

    Entry getFloorEntry(K key) const {
```

```
        if (this->root == nullptr) {
            throw NoSuchMappingException("No floor entry exists in this
map");
        }

        NodePtr node = this->root;

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

            if (compare(key, node->key)) {
                /* key < node->key */
                if (node->left != nullptr) {
                    node = node->left;
                } else {
                    while (node->direction() == Direction::LEFT) {
                        if (node != nullptr) {
                            node = node->parent;
                        } else {
                            throw NoSuchMappingException("No floor entry
exists in this map");
                        }
                    }
                    if (node->parent == nullptr) {
                        throw NoSuchMappingException("No floor entry exists
in this map");
                    }
                    return node->parent->entry();
                }
            } else {
                /* key > node->key */
                if (node->right != nullptr) {
                    node = node->right;
                } else {
                    return node->entry();
                }
            }
        }

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

```
    Entry getHigherEntry(K key) {
        if (this->root == nullptr) {
            throw NoSuchMappingException("No higher entry exists in this
map");
        }

        NodePtr node = this->root;

        while (node != nullptr) {
            if (compare(key, node->key)) {
                /* key < node->key */
                if (node->left != nullptr) {
                    node = node->left;
                } else {
                    return node->entry();
                }
            } else {
                /* key >= node->key */
                if (node->right != nullptr) {
                    node = node->right;
                } else {
                    while (node->direction() == Direction::RIGHT) {
                        if (node != nullptr) {
                            node = node->parent;
                        } else {
                            throw NoSuchMappingException(
                                    "No higher entry exists in this map");
                        }
                    }
                    if (node->parent == nullptr) {
                        throw NoSuchMappingException("No higher entry
exists in this map");
                    }
                    return node->parent->entry();
                }
            }
        }

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

    Entry getLowerEntry(K key) const {
        if (this->root == nullptr) {
            throw NoSuchMappingException("No lower entry exists in this
```

```
map");
        }

        NodePtr node = this->root;

        while (node != nullptr) {
            if (compare(key, node->key) || key == node->key) {
                /* key <= node->key */
                if (node->left != nullptr) {
                    node = node->left;
                } else {
                    while (node->direction() == Direction::LEFT) {
                        if (node != nullptr) {
                            node = node->parent;
                        } else {
                            throw NoSuchMappingException("No lower entry
exists in this map");
                        }
                    }
                    if (node->parent == nullptr) {
                        throw NoSuchMappingException("No lower entry exists
in this map");
                    }
                    return node->parent->entry();
                }
            } else {
                /* key > node->key */
                if (node->right != nullptr) {
                    node = node->right;
                } else {
                    return node->entry();
                }
            }
        }

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

    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);
        }
    }

    void forEach(KeyValueConsumer action) const {
        this->inorderTraversal(
                [&](ConstNodePtr node) { action(node->key, node-
>value); });
    }

    void forEachMut(MutKeyValueConsumer action) {
        this->inorderTraversal(
                [&](ConstNodePtr node) { action(node->key, node-
>value); });
    }

    EntryList toEntryList() const {
        EntryList entryList;
        this->inorderTraversal(
                [&](ConstNodePtr node) { entryList.push_back(node-
>entry()); });
        return entryList;
    }

 private:
    static void maintainRelationship(ConstNodePtr node) {
        if (node->left != nullptr) {
            node->left->parent = node;
        }
        if (node->right != nullptr) {
            node->right->parent = node;
        }
    }

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

    void rotateLeft(ConstNodePtr node) {
        assert(node != nullptr && node->right != nullptr);
```

```
        // clang-format on
        NodePtr parent = node->parent;
        Direction direction = node->direction();

        NodePtr successor = node->right;
        node->right = successor->left;
        successor->left = node;

        maintainRelationship(node);
        maintainRelationship(successor);

        switch (direction) {
            case Direction::ROOT:
                this->root = successor;
                break;
            case Direction::LEFT:
                parent->left = successor;
                break;
            case Direction::RIGHT:
                parent->right = successor;
                break;
        }

        successor->parent = parent;
    }

    void rotateRight(ConstNodePtr node) {
        assert(node != nullptr && node->left != nullptr);
        // clang-format on

        NodePtr parent = node->parent;
        Direction direction = node->direction();

        NodePtr successor = node->left;
        node->left = successor->right;
        successor->right = node;

        maintainRelationship(node);
        maintainRelationship(successor);

        switch (direction) {
            case Direction::ROOT:
                this->root = successor;
                break;
```

```
            case Direction::LEFT:
                parent->left = successor;
                break;
            case Direction::RIGHT:
                parent->right = successor;
                break;
        }

        successor->parent = parent;
    }

    inline void rotateSameDirection(ConstNodePtr node, Direction direction)
{
        assert(direction != Direction::ROOT);
        if (direction == Direction::LEFT) {
            rotateLeft(node);
        } else {
            rotateRight(node);
        }
    }

    inline void rotateOppositeDirection(ConstNodePtr node, Direction
direction) {
        assert(direction != Direction::ROOT);
        if (direction == Direction::LEFT) {
            rotateRight(node);
        } else {
            rotateLeft(node);
        }
    }

    void maintainAfterInsert(NodePtr node) {
        assert(node != nullptr);

        if (node->isRoot()) {
            // Case 1: Current node is root (RED)
            //    No need to fix.
            assert(node->isRed());
            return;
        }

        if (node->parent->isBlack()) {
            // Case 2: Parent is BLACK
            //    No need to fix.
```

```
            return;
        }

        if (node->parent->isRoot()) {
            // clang-format off
            // Case 3: Parent is root and is RED
            //     Paint parent to BLACK.
            //        <P>                 [P]
            //         |      ====>        |
            //        <N>                 <N>
            //     p.s.
            //        `<X>` is a RED node;
            //        `[X]` is a BLACK node (or NIL);
            //        `{X}` is either a RED node or a BLACK node;
            // clang-format on
            assert(node->parent->isRed());
            node->parent->color = Node::BLACK;
            return;
        }

        if (node->hasUncle() && node->uncle()->isRed()) {
            // clang-format off
            // Case 4: Both parent and uncle are RED
            //     Paint parent and uncle to BLACK;
            //     Paint grandparent to RED.
            //                [G]                         <G>
            //                / \                         / \
            //            <P> <U>     ====>     [P] [U]
            //            /                            /
            //        <N>                         <N>
            // clang-format on
            assert(node->parent->isRed());
            node->parent->color = Node::BLACK;
            node->uncle()->color = Node::BLACK;
            node->grandParent()->color = Node::RED;
            maintainAfterInsert(node->grandParent());
            return;
        }

        if (!node->hasUncle() || node->uncle()->isBlack()) {
            // Case 5 & 6: Parent is RED and Uncle is BLACK
            //     p.s. NIL nodes are also considered BLACK
            assert(!node->isRoot());
```

```
            if (node->direction() != node->parent->direction()) {
                // clang-format off
                // Case 5: Current node is the opposite direction as parent
                //     Step 1. If node is a LEFT child, perform l-rotate to
parent;
                //                     If node is a RIGHT child, perform r-
rotate to parent.
                //     Step 2. Goto Case 6.
                //            [G]                                 [G]
                //            / \        rotate(P)        / \
                //        <P> [U]    ========>    <N> [U]
                //            \                               /
                //            <N>                         <P>
                // clang-format on

                // Step 1: Rotation
                NodePtr parent = node->parent;
                if (node->direction() == Direction::LEFT) {
                    rotateRight(node->parent);
                } else /* node->direction() == Direction::RIGHT */ {
                    rotateLeft(node->parent);
                }
                node = parent;
                // Step 2: vvv
            }

            // clang-format off
            // Case 6: Current node is the same direction as parent
            //     Step 1. If node is a LEFT child, perform r-rotate to
grandparent;
            //                     If node is a RIGHT child, perform l-
rotate to grandparent.
            //     Step 2. Paint parent (before rotate) to BLACK;
            //                     Paint grandparent (before rotate) to
RED.
            //                [G]                                 <P>
[P]
            //                / \        rotate(G)        / \
repaint       / \
            //            <P> [U]    ========>    <N> [G]     ======>    <N>
<G>
            //            /
\                                 \
            //        <N>
```

```
[U]                             [U]
            // clang-format on

            assert(node->grandParent() != nullptr);

            // Step 1
            if (node->parent->direction() == Direction::LEFT) {
                rotateRight(node->grandParent());
            } else {
                rotateLeft(node->grandParent());
            }

            // Step 2
            node->parent->color = Node::BLACK;
            node->sibling()->color = Node::RED;

            return;
        }
    }

    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();
                node->left->parent = node;
                maintainAfterInsert(node->left);
                this->count += 1;
            } else {
                result = getNodeOrProvide(node->left, key, provide);
            }
        } else {
            /* key > node->key */
            if (node->right == nullptr) {
```

```
                result = node->right = provide();
                node->right->parent = node;
                maintainAfterInsert(node->right);
                this->count += 1;
            } else {
                result = getNodeOrProvide(node->right, key, provide);
            }
        }

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

    void maintainAfterRemove(ConstNodePtr node) {
        if (node->isRoot()) {
            return;
        }

        assert(node->isBlack() && node->hasSibling());

        Direction direction = node->direction();

        NodePtr sibling = node->sibling();
        if (sibling->isRed()) {
            ConstNodePtr parent = node->parent;
            assert(parent != nullptr && parent->isBlack());
            assert(sibling->left != nullptr && sibling->left->isBlack());
            assert(sibling->right != nullptr && sibling->right->isBlack());
            rotateSameDirection(node->parent, direction);
            sibling->color = Node::BLACK;
            parent->color = Node::RED;
            sibling = node->sibling();
        }

        NodePtr closeNephew =
                direction == Direction::LEFT ? sibling->left : sibling-
```

```
>right;
        NodePtr distantNephew =
                direction == Direction::LEFT ? sibling->right : sibling-
>left;

        bool closeNephewIsBlack = closeNephew == nullptr || closeNephew-
>isBlack();
        bool distantNephewIsBlack =
                distantNephew == nullptr || distantNephew->isBlack();

        assert(sibling->isBlack());

        if (closeNephewIsBlack && distantNephewIsBlack) {
            if (node->parent->isRed()) {
                // clang-format off
                // Case 2: Sibling and nephews are BLACK, parent is RED
                //      Swap the color of P and S
                //              <P>                         [P]
                //              / \                         / \
                //          [N] [S]     ====>    [N] <S>
                //                  / \                         / \
                //              [C] [D]                 [C] [D]
                // clang-format on
                sibling->color = Node::RED;
                node->parent->color = Node::BLACK;
                return;
            } else {
                // clang-format off
                // Case 3: Sibling, parent and nephews are all black
                //      Step 1. Paint S to RED
                //      Step 2. Recursively maintain P
                //              [P]                         [P]
                //              / \                         / \
                //          [N] [S]     ====>    [N] <S>
                //                  / \                         / \
                //              [C] [D]                 [C] [D]
                // clang-format on
                sibling->color = Node::RED;
                maintainAfterRemove(node->parent);
                return;
            }
        } else {
            if (closeNephew != nullptr && closeNephew->isRed()) {
                // Step 1
```

```
                rotateOppositeDirection(sibling, direction);
                // Step 2
                closeNephew->color = Node::BLACK;
                sibling->color = Node::RED;
                // Update sibling and nephews after rotation
                sibling = node->sibling();
                closeNephew =
                        direction == Direction::LEFT ? sibling->left :
sibling->right;
                distantNephew =
                        direction == Direction::LEFT ? sibling->right :
sibling->left;
                // Step 3: vvv
            }

            assert(closeNephew == nullptr || closeNephew->isBlack());
            assert(distantNephew->isRed());
            // Step 1
            rotateSameDirection(node->parent, direction);
            // Step 2
            sibling->color = node->parent->color;
            node->parent->color = Node::BLACK;
            if (distantNephew != nullptr) {
                distantNephew->color = Node::BLACK;
            }
            return;
        }
    }

    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)) {
                    maintainRelationship(node);
                    return true;
                } else {
                    return false;
                }
            } else {
                /* key > node->key */
```

```
                NodePtr &right = node->right;
                if (right != nullptr && remove(right, key, action)) {
                    maintainRelationship(node);
                    return true;
                } else {
                    return false;
                }
            }
        }

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

        if (this->size() == 1) {
            // Current node is the only node of the tree
            this->clear();
            return true;
        }

        if (node->left != nullptr && node->right != nullptr) {
            // clang-format off
            // Case 1: If the node is strictly internal
            //     Step 1. Find the successor S with the smallest key
            //                     and its parent P on the right subtree.
            //     Step 2. Swap the data (key and value) of S and N,
            //                     S is the node that will be deleted in
place of N.
            //     Step 3. N = S, goto Case 2, 3
            //         |                                        |
            //         N                                        S
            //        / \                                    / \
            //     L    ..     swap(N, S)     L    ..
            //             |     =========>             |
            //             P                                        P
            //            / \                                    / \
            //         S    ..                              N    ..
            // 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);
            maintainRelationship(parent);
            // Step 3: vvv
        }

        if (node->isLeaf()) {
            // Current node must not be the root
            assert(node->parent != nullptr);

            // Case 2: Current node is a leaf
            //     Step 1. Unlink and remove it.
            //     Step 2. If N is BLACK, maintain N;
            //                     If N is RED, do nothing.

            // The maintain operation won't change the node itself,
            //    so we can perform maintain operation before unlink the
node.
            if (node->isBlack()) {
                maintainAfterRemove(node);
            }
            if (node->direction() == Direction::LEFT) {
                node->parent->left = nullptr;
            } else /* node->direction() == Direction::RIGHT */ {
                node->parent->right = nullptr;
            }
        } else /* !node->isLeaf() */ {
            assert(node->left == nullptr || node->right == nullptr);
            // Case 3: Current node has a single left or right child
            //     Step 1. Replace N with its child
            //     Step 2. If N is BLACK, maintain N
            NodePtr parent = node->parent;
            NodePtr replacement = (node->left != nullptr ? node->left :
node->right);
            switch (node->direction()) {
                case Direction::ROOT:
                    this->root = replacement;
                    break;
                case Direction::LEFT:
                    parent->left = replacement;
                    break;
                case Direction::RIGHT:
                    parent->right = replacement;
```

```
                    break;
                }

                if (!node->isRoot()) {
                    replacement->parent = parent;
                }

                if (node->isBlack()) {
                    if (replacement->isRed()) {
                        replacement->color = Node::BLACK;
                    } else {
                        maintainAfterRemove(replacement);
                    }
                }
            }

            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    // RBTREE_MAP_HPP
```