Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности

Высшая школа программной инженерии

КУРСОВАЯ РАБОТА

Алгоритмы работы со словарями

по дисциплине «**Алгоритмы и структуры данных**»

Выполнил студент гр. 5130904/30003 Новиков Вадим Дмитриевич

Руководитель Шемякин Илья Александрович

«___» _____ 2024 г.

Санкт-Петербург

2024 г.

Содержание

1. Введение. Общая постановка задачи

Тема: Алгоритмы работы со словарями

1. Для разрабатываемого словаря реализовать основные операции:
   - `INSERT` (ключ, значение) — добавить запись с указанным ключом и значением;
   - `SEARCH` (ключ) — найти все записи с указанным ключом;
   - `REMOVE` (ключ, значение) — удалить запись с указанным ключом и значением.
2. Предусмотреть обработку и инициализацию исключительных ситуаций, связанных, например, с проверкой значения полей перед инициализацией и присваиванием.
3. Программа должна быть написана в соответствии со стилем программирования: C++ Programming Style Guidelines (http://geosoft.no/development/cppstyle.html).
4. Тесты должны учитывать как допустимые, так и не допустимые последовательности входных данных.

Вариант 1.1.2.
Англо-русский словарь. АВЛ-дерево

Разработать и реализовать алгоритм работы с англо-русским словарем, реализованным как АВЛ-дерево.

Узел АВЛ-дерева должен содержать:

- Ключ — английское слово
- Показатель (фактор) сбалансированности
- Информационная часть — ссылка на список, содержащий переводы английского слова, отсортированные по алфавиту (переводов слова может быть несколько).

2. Основная часть работы

2.1. Описание алгоритма решения и используемых структур данных.

АВЛ-дерево — один из первых видов сбалансированных двоичных деревьев поиска, изобретённый в 1962 году советскими учёными Адельсон-Вельским и Ландисом [1]. Аббревиатура АВЛ образована первыми буквами фамилий его создателей. Особенность АВЛ-дерева заключается в том, что для любого его узла высота правого поддерева отличается от высоты левого поддерева не более чем на единицу. Это обеспечивается следующим образом:

1. Особая структура узлов. Помимо ключа и указателей на узлы родителя, левого и правого ребёнка они содержат показатель (фактор) сбалансированности — разность высот правого и левого поддеревьев. Он может принимать 3 значения: -1, 0 или 1 в зависимости от того, какое из поддеревьев имеет большую высоту. Если фактор сбалансированности имеет отличное от представленных значение (-2 или 2), дерево является несбалансированным и ему требуется перебалансировка.
2. Адаптация алгоритмов вставки и удаления элементов. Ситуация несбалансированности чаще всего возникает при редактировании дерева, то есть при вставке или удалении элементов, поэтому соответствующие алгоритмы включают в себя механизм перебалансировки, основанный на правом и левом поворотах. Более подробно эти алгоритмы описаны в главе 2.2.

2.2. Анализ алгоритма

2.2.1. Алгоритмы балансировки узлов

Как уже было сказано ранее, при вставке и удалении элементов в АВЛ-дерево возможно возникновение ситуации, когда фактор сбалансированности у некоторых узлов оказывается равен -2 или 2, то есть происходит разбалансировка поддерева. Для исправления этой ситуации применяются алгоритмы поворота: малый левый, малый правый, большой левый и большой правый [2]. Обозначим фактор балансировки bf.

1. Малый левый поворот узла a вокруг его правого ребёнка b используется в случае, когда $bf(a)=2$ и $bf(b)\geq 0$. При этом узел b становится родителем узла a, а узел a становится левым ребёнком узла b.

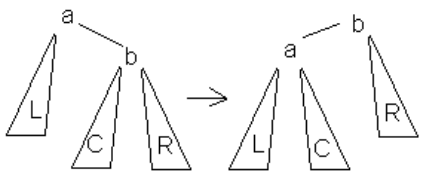

Рис. 1: Малый левый поворот

2. Малый правый поворот является симметричной копией левого и используется в случае, когда $bf(a)=-2$ и $bf(b)\leq 0$.

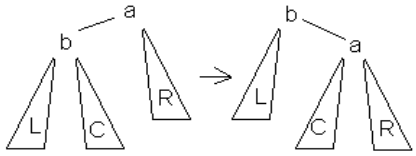

Рис. 2: Малый правый поворот

3. Большой левый поворот сводится к двум простым поворотам — сначала правый поворот узла b, затем левый поворот узла a. Применяется, когда $bf(a)=2$ и $bf(b)<0$.

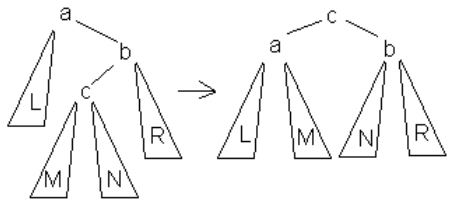

Рис. 3: Большой левый поворот

4. Большой правый поворот также представляет собой комбинацию двух простых поворотов — сначала правый поворот вокруг узла b, затем левый поворот вокруг узла. Применяется, когда $bf(a)=-2$ и $bf(b)>0$.

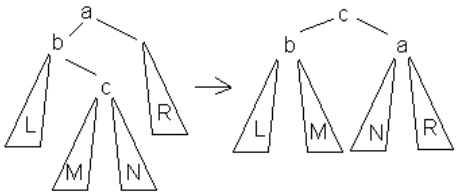

Рис. 4: Большый правый поворот

Анализ возможных ситуаций показывает, что выполнить перебалансировку узла a можно следующим образом:

Если $bf(a)=2$, то проверяем фактор сбалансированности правого ребёнка. Если он больше или равен 0, то выполняем левый поворот вокруг узла a, иначе выполняем большой левый поворот вокруг того же узла.

Если $bf(a)=-2$, то проверяем фактор сбалансированности левого ребёнка. Если он меньше или равен 0, то выполняем правый поворот вокруг узла a, иначе выполняем большой правый поворот вокруг того же узла.

Если $-1\le bf(a)\le 1$, ничего не делаем.

2.2.2. Алгоритм вставки

Алгоритм вставки в АВЛ-дерево работает по тому же принципу, что алгоритм вставки в несбалансированное бинарное дерево поиска. Единственное отличие заключается в том, что после вставки необходимо выполнить перебалансировку корневого узла. Время работы — $O(\log n)$.

Начинаем поиск места для вставки с корневого узла. Если вставляемый элемент меньше элемента в текущем узле, то переходим к его левому ребёнку, а если больше, переходим к правому. Повторяем до тех пор, пока не достигнем элемента, равного вставляемому или являющегося листом. В первом случае завершаем работу функции и сообщаем, что вставка не удалась, во втором

случае выделяем память под новый элемент, связываем его с найденным листом и выполняем перебалансировку поддерева. Для этого поднимаемся вверх к корню и пересчитываем факторы балансировки у узлов. Если мы поднялись в узел *i* из левого поддерева, то уменьшаем его фактор балансировки на 1, если из правого — увеличиваем на 1 [3]. Если баланс вершины стал равен 2 или -2, выполняем один из четырёх поворотов и, если после этого баланс вершины стал равен 0, то останавливаемся, иначе продолжаем подъём.

2.2.3. Алгоритм поиска

Алгоритм поиска в АВЛ-дереве полностью идентичен алгоритму поиска в несбалансированном бинарном дереве поиска. Время работы — $O(\log n)$.

Начинаем поиск нужного элемента с корневого узла. Если вставляемый элемент меньше элемента в текудощем узле, то переходим к его левому ребёнку, а если больше, переходим к правому. Повторяем до тех пор, пока не достигнем элемента, являющегося листом или равного искомому. В первом случае завершаем работу функции и сообщаем, что элемента нет в дереве, во втором случае предоставляем доступ к этому элементу.

2.2.4. Алгоритм удаления

Алгоритм удаления из АВЛ-дерева работает по тому же принципу, что алгоритм удаления из несбалансированного бинарного дерева поиска. Единственное отличие заключается в том, что после удаления необходимо выполнить перебалансировку. Время работы — $O(\log n)$.

Начинаем поиск удаляемого элемента с корневого узла. Если удаляемый элемент меньше элемента в текущем узле, то переходим к его левому ребёнку, а если больше, переходим к правому. Повторяем до тех пор, пока не достигнем элемента, являющегося «ребёнком» листа или равного удаляемому. В первом случае завершаем работу функции и сообщаем, что удаление не удалось. Во втором случае:

1. Если найденный элемент является листом, отвязываем от него родителя, если он есть.
2. Если найденный элемент имеет одного ребёнка, ставим этого ребёнка на его место (привязываем родителя элемента к ребёнку, а ребёнка к родителю).
3. Если найденный элемент имеет двух детей, заменяем его на наименьший элемент из его правого поддерева (или наибольший из левого поддерева).

Далее удаляем найденный элемент из памяти и выполняем перебалансировку. От удалённого узла поднимаемся вверх к корню и пересчитываем факторы балансировки. Если мы поднялись в узел *i* из левого поддерева, то увеличиваем его фактор балансировки на 1, если из правого — уменьшаем на 1. Если после этого баланс вершины стал равен 1 или -1, то подъём можно остановить, то как высота этого поддерева не изменилась. Если баланс стал равен 0, то высота поддерева уменьшилась, и подъём нужно продолжить Если баланс стал равен 2 или -2, выполняем одно из четырёх вращений и, если после этого баланс вершины стал равен нулю, продолжаем подъём, иначе останавливаемся.

2.3. Описание спецификации программы (детальные требования)

Программа работает со словарём, хранящимся в оперативной памяти. В словаре хранятся ключи (английские слова) и соответствующие им значения (русские слова).

Реализовано 3 команды: `INSERT`, `SEARCH` и `REMOVE`.

- `INSERT` (ключ, значение) — добавить запись с указанным ключом и значением.
- `SEARCH` (ключ) — найти все записи с указанным ключом.
- `REMOVE` (ключ, значение) — удалить запись с указанным ключом и значением.

Для хранения англо-русского словаря и команд для взаимодействия с ним используется АВЛ-дерево. Для реализации интерфейса команд используется функция `std::bind`, предназначенная для создания одних функциональных объектов на основе других.

2.4. Описание программы (структура программы, форматы входных и выходных данных)

Программа, реализующая англо-русский словарь на основе АВЛ-дерева, написана на языке C++ с использованием стандарта C++14.

На основе АВЛ-дерева реализованы множество (`AvlTreeSet`) и словарь (`AvlTreeMap`). Множество хранит упорядоченные уникальные ключи, словарь хранит упорядоченные уникальные ключи, каждому из которых соответствует единственное значение. Созданы отдельные классы для множества, словаря, их узлов и стратегий обхода (итераторов). Итераторы являются двунаправленными,

то есть имеется возможность обходить АВЛ-дерево как прямым, так и в обратным инфиксным обходом.

Сам англо-русский словарь реализован как словарь множеств, где ключом является строка, а значением — упорядоченное множество слов. Тип данных — `AvlTreeMap<std::string, AvlTreeSet<std::string>>`. Для данного типа реализован класс-обёртка `EngRusDictionary`, в котором определены методы для более удобного взаимодействия с такой композицией типов извне.

Описание команд:

1. `INSERT <ключ> <значение>`

   Описание: добавить запись с указанным ключом и значением.
   Если элемент с таким ключом и значением уже существует, то выводится надпись `<INVALID_COMMAND>`, иначе элемент добавляется в словарь.

   Пример использования:

   ```
   INSERT good хороший
   INSERT good хорошо
   INSERT good товар
   INSERT good товар
   ```

   Ожидаемый результат:

   ```
   <INVALID_COMMAND>
   ```

2. `SEARCH <ключ>`

   Описание: найти все записи с указанным ключом.
   Если элементов с таким ключом не существует, то выводится надпись `<INVALID_COMMAND>`, иначе выводятся все элементы с текущим ключом в алфавитном порядке.

   Пример использования:

   ```
   SEARCH good
   INSERT good хороший
   INSERT good хорошо
   INSERT good товар
   SEARCH good
   ```

   Ожидаемый результат:

   ```
   <INVALID_COMMAND>
   товар
   хороший
   хорошо
   ```

3. `REMOVE <ключ> <значение>`

   Описание: удалить запись с указанным ключом и значением.
   Если элемент с таким ключом и значением не существует, то выводится надпись `<INVALID_COMMAND>`, иначе элемент удаляется из словаря.

   Пример использования:

```
INSERT good хороший
INSERT good хорошо
INSERT good товар
REMOVE good товар
REMOVE good товар
SEARCH good
```

   Ожидаемый результат:

```
<INVALID_COMMAND>
хороший
хорошо
```

3. Заключение

В ходе работы были изучено устройство АВЛ-дерева и механизмы его балансировки, принципы работы со словарём с использованием АВЛ-деревьев, разработана программа, обеспечивающая удобное взаимодействие со словарем. Полученные результаты говорят об удобстве и эффективности взаимодействия со словарём.

Приложение 1. Текст программы

// main.cpp

```
#include <iostream>
#include <functional>
#include <limits>
#include "commands.hpp"

int main()
{
    using namespace coursework;

    EngRusDictionary dict;

    AvlTreeMap<std::string, std::function<void(std::istream&, std::ostream&)>>
commands;

    using namespace std::placeholders;
    commands.insert("INSERT", std::bind(cmd::insert, _1, _2, std::ref(dict)));
    commands.insert("SEARCH", std::bind(cmd::search, _1, _2, std::cref(dict)));
    commands.insert("REMOVE", std::bind(cmd::remove, _1, _2, std::ref(dict)));

    std::string cmd;

    while (std::cin >> cmd)
    {
        auto it = commands.search(cmd);

        if (it == commands.end())
        {
            std::cout << "<INVALID_COMMAND>\n";
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        }
        else
        {
            it->second(std::cin, std::cout);
        }
    }

    return 0;
}
```

// commands.hpp

```
#ifndef COMMANDS_HPP
#define COMMANDS_HPP

#include "EngRusDictionary.hpp"
#include <iosfwd>

namespace coursework
{
    namespace cmd
    {
        void insert(std::istream& in, std::ostream& out, EngRusDictionary&
dict);
        void search(std::istream& in, std::ostream& out, const EngRusDictionary&
dict);
```

```
        void remove(std::istream& in, std::ostream& out, EngRusDictionary&
dict);
    }
}

#endif // COMMANDS_HPP
```

// commands.cpp

```
#include "commands.hpp"
#include <ostream>

void coursework::cmd::insert(std::istream& in, std::ostream& out,
EngRusDictionary& dict)
{
    std::string key;
    std::string value;
    in >> key >> value;

    bool res = dict.insert(std::move(key), std::move(value));

    if (!res)
    {
        out << "<INVALID_COMMAND>\n";
    }
}

void coursework::cmd::search(std::istream& in, std::ostream& out, const
EngRusDictionary& dict)
{
    std::string key;
    in >> key;

    auto res = dict.search(key);

    if (res.begin() == res.end())
    {
        out << "<INVALID_COMMAND>\n";
        return;
    }

    for (const auto& i : res)
    {
        out << i << "\n";
    }
}

void coursework::cmd::remove(std::istream& in, std::ostream& out,
EngRusDictionary& dict)
{
    std::string key;
    std::string value;
    in >> key >> value;

    bool res = dict.remove(key, value);

    if (!res)
    {
        out << "<INVALID_COMMAND>\n";
    }
}
```

// EngRusDictionary.hpp

```
#ifndef ENG_RUS_DICTIONARY_HPP
#define ENG_RUS_DICTIONARY_HPP

#include <string>
#include "AvlTreeSet.hpp"
#include "AvlTreeMap.hpp"

namespace coursework
{
    class EngRusDictionary
    {
    public:

        using Subdict = AvlTreeSet<std::string>;
        using Dict = AvlTreeMap<std::string, Subdict>;

        EngRusDictionary() = default;

        EngRusDictionary(const EngRusDictionary&) = default;
        EngRusDictionary(EngRusDictionary&&) noexcept = default;

        EngRusDictionary& operator=(const EngRusDictionary&) = default;
        EngRusDictionary& operator=(EngRusDictionary&&) noexcept = default;

        ~EngRusDictionary() noexcept = default;

        Dict::Iterator begin();
        Dict::Iterator end();
        Dict::ConstIterator cbegin() const;
        Dict::ConstIterator cend() const;

        bool insert(std::string key, std::string value);
        const Subdict& search(const std::string& rhs) const;
        bool remove(const std::string& key, const std::string& value);

    private:

        Dict dict_;
    };
}

#endif // ENG_RUS_DICTIONARY_HPP
```

// EngRusDictionary.cpp

```
#include "EngRusDictionary.hpp"
#include "AvlTreeSetIterator.hpp"
#include <utility>

coursework::EngRusDictionary::Dict::Iterator
coursework::EngRusDictionary::begin()
{
    return dict_.begin();
}

coursework::EngRusDictionary::Dict::Iterator coursework::EngRusDictionary::end()
{
    return dict_.end();
}
```

```
coursework::EngRusDictionary::Dict::ConstIterator
coursework::EngRusDictionary::cbegin() const
{
    return dict_.cbegin();
}

coursework::EngRusDictionary::Dict::ConstIterator
coursework::EngRusDictionary::cend() const
{
    return dict_.cend();
}

bool coursework::EngRusDictionary::insert(std::string key, std::string value)
{
    auto it = dict_.search(key);

    if (it == dict_.end())
    {
        auto res = dict_.insert(std::move(key), Subdict())-
>second.insert(std::move(value));
        return res != dict_.begin()->second.end();
    }

    return it->second.insert(std::forward<std::string>(value)) != it-
>second.end();
}

const coursework::EngRusDictionary::Subdict&
coursework::EngRusDictionary::search(const std::string& rhs) const
{
    return dict_.search(rhs)->second;
}

bool coursework::EngRusDictionary::remove(const std::string& key, const
std::string& value)
{
    auto it = dict_.search(key);

    if (it == dict_.end())
    {
        return false;
    }

    return it->second.remove(value) != it->second.end();
}
```

// AvlTreeSet.hpp

```
#ifndef AVL_TREE_SET_HPP
#define AVL_TREE_SET_HPP

#include <initializer_list>
#include <utility>
#include "AvlTreeSetIterator.hpp"
#include "AvlTreeSetNode.hpp"
#include "AvlTreeTraverseStrategy.hpp"

namespace coursework
{
    template <typename T>
```

```
    class AvlTreeSet
    {
        using Node = detail::AvlTreeSetNode<T>;

    public:

        using Iterator = detail::AvlTreeSetIterator<T>;
        using ConstIterator = Iterator;
        using ReverseIterator = detail::AvlTreeSetIterator<T,
detail::ReversedInfixTraverse<detail::AvlTreeSetNode<T>>>;
        using ConstReverseIterator = ReverseIterator;

        AvlTreeSet();
        AvlTreeSet(const AvlTreeSet& rhs);
        AvlTreeSet(AvlTreeSet&& rhs) noexcept;
        AvlTreeSet(std::initializer_list<T> rhs);
        template <typename InputIterator>
        AvlTreeSet(InputIterator begin, InputIterator end);

        AvlTreeSet& operator=(const AvlTreeSet& rhs);
        AvlTreeSet& operator=(AvlTreeSet&& rhs) noexcept;

        virtual ~AvlTreeSet() noexcept;

        Iterator begin() const;
        Iterator end() const;
        ConstIterator cbegin() const;
        ConstIterator cend() const;

        ReverseIterator rbegin() const;
        ReverseIterator rend() const;
        ConstReverseIterator crbegin() const;
        ConstReverseIterator crend() const;

        Iterator insert(T rhs);
        Iterator search(const T& rhs) const;
        Iterator remove(const T& rhs);

        const Node* getRoot() const;

    private:

        void clear(Node* rhs);
        Node* root_;
    };
}

template <typename T>
coursework::AvlTreeSet<T>::AvlTreeSet():
    root_(nullptr)
{}

template <typename T>
coursework::AvlTreeSet<T>::AvlTreeSet(const AvlTreeSet& rhs)
{
    AvlTreeSet<T> temp;

    for (const auto& i : rhs)
    {
        temp.insert(i);
    }
```

```
    root_ = temp.root_;
    temp.root_ = nullptr;
}

template <typename T>
coursework::AvlTreeSet<T>::AvlTreeSet(std::initializer_list<T> rhs)
{
    AvlTreeSet<T> temp;

    for (const auto& i : rhs)
    {
        temp.insert(i);
    }

    root_ = temp.root_;
    temp.root_ = nullptr;
}

template <typename T>
coursework::AvlTreeSet<T>::AvlTreeSet(AvlTreeSet&& rhs) noexcept:
    root_(rhs.root_)
{
    rhs.root_ = nullptr;
}

template <typename T>
coursework::AvlTreeSet<T>& coursework::AvlTreeSet<T>::operator=(const
AvlTreeSet& rhs)
{
    if (this != &rhs)
    {
        AvlTreeSet<T> temp(rhs);
        std::swap(root_, temp.root_);
    }

    return *this;
}

template <typename T>
coursework::AvlTreeSet<T>& coursework::AvlTreeSet<T>::operator=(AvlTreeSet&&
rhs) noexcept
{
    if (this != &rhs)
    {
        clear(root_);
        root_ = rhs.root_;
        rhs.root_ = nullptr;
    }

    return *this;
}

template <typename T>
coursework::AvlTreeSet<T>::~AvlTreeSet() noexcept
{
    clear(root_);
}

template <typename T>
```

```
typename coursework::AvlTreeSet<T>::Iterator coursework::AvlTreeSet<T>::begin()
const
{
    if (root_ == nullptr)
    {
        return end();
    }

    Node* res = root_;

    while (res->left_ != nullptr)
    {
        res = res->left_;
    }

    return Iterator(root_, res);
}

template <typename T>
typename coursework::AvlTreeSet<T>::Iterator coursework::AvlTreeSet<T>::end()
const
{
    return Iterator(root_, nullptr);
}

template <typename T>
typename coursework::AvlTreeSet<T>::ConstIterator
coursework::AvlTreeSet<T>::cbegin() const
{
    return begin();
}

template <typename T>
typename coursework::AvlTreeSet<T>::ConstIterator
coursework::AvlTreeSet<T>::cend() const
{
    return end();
}

template <typename T>
typename coursework::AvlTreeSet<T>::ReverseIterator
coursework::AvlTreeSet<T>::rbegin() const
{
    if (root_ == nullptr)
    {
        return rend();
    }

    Node* res = root_;

    while (res->right_ != nullptr)
    {
        res = res->right_;
    }

    return ReverseIterator(root_, res);
}

template <typename T>
typename coursework::AvlTreeSet<T>::ReverseIterator
coursework::AvlTreeSet<T>::rend() const
```

```
{
    return ReverseIterator(root_, nullptr);
}

template <typename T>
typename coursework::AvlTreeSet<T>::ConstReverseIterator
coursework::AvlTreeSet<T>::crbegin() const
{
    return rbegin();
}

template <typename T>
typename coursework::AvlTreeSet<T>::ConstReverseIterator
coursework::AvlTreeSet<T>::crend() const
{
    return rend();
}

template <typename T>
typename coursework::AvlTreeSet<T>::Iterator coursework::AvlTreeSet<T>::insert(T
rhs)
{
    if (root_ == nullptr)
    {
        root_ = new Node(std::move(rhs));
        return begin();
    }

    Node* curr = root_;
    Node* prev = nullptr;

    while (curr != nullptr && rhs != curr->key_)
    {
        prev = curr;
        curr = rhs < curr->key_ ? curr->left_ : curr->right_;
    }

    if (curr != nullptr)
    {
        return end();
    }

    curr = new Node(std::move(rhs), prev);
    (curr->key_ < curr->parent_->key_ ? curr->parent_->left_ : curr->parent_-
>right_) = curr;

    bool isRebalanced = false;

    Node* prevBack = curr;
    Node* currBack = prev;

    while (currBack != nullptr && !isRebalanced)
    {
        if (prevBack == currBack->left_)
        {
            --currBack->factor_;
        }
        else if (prevBack == currBack->right_)
        {
            ++currBack->factor_;
        }
```

```
        currBack = currBack->balance(root_);

        isRebalanced = currBack->factor_ == 0;

        prevBack = currBack;
        currBack = currBack->parent_;
    }

    return Iterator(root_, curr);
}

template <typename T>
typename coursework::AvlTreeSet<T>::Iterator
coursework::AvlTreeSet<T>::search(const T& rhs) const
{
    Node* curr = root_;

    while (curr != nullptr && rhs != curr->key_)
    {
        curr = rhs < curr->key_ ? curr->left_ : curr->right_;
    }

    return Iterator(root_, curr);
}

template <typename T>
typename coursework::AvlTreeSet<T>::Iterator
coursework::AvlTreeSet<T>::remove(const T& rhs)
{
    Node* curr = root_;

    while (curr != nullptr && rhs != curr->key_)
    {
        curr = rhs < curr->key_ ? curr->left_ : curr->right_;
    }

    if (curr == nullptr)
    {
        return end();
    }

    Node* res = detail::stepForward(root_, curr);

    if (curr->left_ == nullptr && curr->right_ == nullptr)
    {
        if (curr != root_)
        {
            (curr == curr->parent_->left_ ? curr->parent_->left_ : curr-
>parent_->right_) = nullptr;
        }
        else
        {
            root_ = nullptr;
        }
    }
    else if (curr->left_ == nullptr || curr->right_ == nullptr)
    {
        Node* const currChild = curr->left_ != nullptr ? curr->left_ : curr-
>right_;
```

```
        if (curr == root_)
        {
            currChild->parent_ = nullptr;
            root_ = currChild;
        }
        else
        {
            currChild->parent_ = curr->parent_;
            (curr == curr->parent_->left_ ? curr->parent_->left_ : curr-
>parent_->right_) = currChild;
        }

        bool isRebalanced = false;

        Node* prevBack = currChild;
        Node* currBack = prevBack->parent_;

        while (currBack != nullptr && !isRebalanced)
        {
            if (prevBack == currBack->left_)
            {
                ++currBack->factor_;
            }
            else if (prevBack == currBack->right_)
            {
                --currBack->factor_;
            }

            currBack = currBack->balance(root_);

            isRebalanced = std::abs(currBack->factor_) == 1;

            prevBack = currBack;
            currBack = currBack->parent_;
        }
    }
    else
    {
        Node* prev = nullptr;
        Node* const temp = curr;

        curr = curr->right_;

        while (curr->left_ != nullptr)
        {
            prev = curr;
            curr = curr->left_;
        }

        Node* prevBack = temp;

        const bool hasAnyChildren = prev != nullptr;

        (hasAnyChildren ? prev->left_ : temp->right_) = curr->right_;

        if (curr->right_ != nullptr)
        {
            curr->right_->parent_ = hasAnyChildren ? prev : temp;
        }

        *const_cast<T*>(&temp->key_) = std::move(curr->key_);
```

```
        bool isRebalanced = false;

        Node* currBack = prevBack->parent_;

        while (currBack != nullptr && !isRebalanced)
        {
            if (prevBack == currBack->left_)
            {
                ++currBack->factor_;
            }
            else if (prevBack == currBack->right_)
            {
                --currBack->factor_;
            }

            currBack = currBack->balance(root_);

            isRebalanced = std::abs(currBack->factor_) == 1;

            prevBack = currBack;
            currBack = currBack->parent_;
        }

        res = temp;
    }

    delete curr;
    return Iterator(root_, res);
}

template <typename T>
const typename coursework::AvlTreeSet<T>::Node*
coursework::AvlTreeSet<T>::getRoot() const
{
    return root_;
}

template <typename T>
void coursework::AvlTreeSet<T>::clear(Node* rhs)
{
    if (rhs != nullptr)
    {
        clear(rhs->left_);
        clear(rhs->right_);
        delete rhs;
    }
}

#endif // AVL_TREE_SET_HPP
```

// AvlTreeSetIterator.hpp

```
#ifndef AVL_TREE_SET_ITERATOR_HPP
#define AVL_TREE_SET_ITERATOR_HPP

#include <iterator>
#include "AvlTreeSetNode.hpp"
#include "AvlTreeTraverseStrategy.hpp"

namespace coursework
```

```
{
    template <typename T>
    class AvlTreeSet;

    namespace detail
    {
        template <typename T, typename S =
StraightInfixTraverse<AvlTreeSetNode<T>>>
        class AvlTreeSetIterator: std::iterator<std::bidirectional_iterator_tag,
const T>
        {
            friend class AvlTreeSet<T>;
            using Node = detail::AvlTreeSetNode<T>;

        public:

            AvlTreeSetIterator();
            AvlTreeSetIterator(const AvlTreeSetIterator&) = default;
            ~AvlTreeSetIterator() = default;

            AvlTreeSetIterator& operator=(const AvlTreeSetIterator&) = default;

            AvlTreeSetIterator& operator++();
            AvlTreeSetIterator operator++(int);
            AvlTreeSetIterator& operator--();
            AvlTreeSetIterator operator--(int);

            const T& operator*() const;
            const T* operator->() const;

            bool operator==(const AvlTreeSetIterator& rhs) const;
            bool operator!=(const AvlTreeSetIterator& rhs) const;

        private:

            Node* root_;
            Node* node_;
            explicit AvlTreeSetIterator(Node* root, Node* node);
        };
    }
}

template <typename T, typename S>
coursework::detail::AvlTreeSetIterator<T, S>::AvlTreeSetIterator():
    root_(nullptr),
    node_(nullptr)
{}

template <typename T, typename S>
coursework::detail::AvlTreeSetIterator<T, S>::AvlTreeSetIterator(Node* root,
Node* node):
    root_(root),
    node_(node)
{}

template <typename T, typename S>
coursework::detail::AvlTreeSetIterator<T, S>&
coursework::detail::AvlTreeSetIterator<T, S>::operator++()
{
    node_ = S::next(root_, node_);
    return *this;
```

```
}

template <typename T, typename S>
coursework::detail::AvlTreeSetIterator<T, S>
coursework::detail::AvlTreeSetIterator<T, S>::operator++(int)
{
    AvlTreeSetIterator<T, S> temp(*this);
    ++(*this);
    return temp;
}

template <typename T, typename S>
coursework::detail::AvlTreeSetIterator<T, S>&
coursework::detail::AvlTreeSetIterator<T, S>::operator--()
{
    node_ = S::prev(root_, node_);
    return *this;
}

template <typename T, typename S>
coursework::detail::AvlTreeSetIterator<T, S>
coursework::detail::AvlTreeSetIterator<T, S>::operator--(int)
{
    AvlTreeSetIterator<T, S> temp(*this);
    --(*this);
    return temp;
}

template <typename T, typename S>
const T& coursework::detail::AvlTreeSetIterator<T, S>::operator*() const
{
    return node_->key_;
}

template <typename T, typename S>
const T* coursework::detail::AvlTreeSetIterator<T, S>::operator->() const
{
    return &node_->key_;
}

template <typename T, typename S>
bool coursework::detail::AvlTreeSetIterator<T, S>::operator==(const
AvlTreeSetIterator& rhs) const
{
    return root_ == rhs.root_ && node_ == rhs.node_;
}

template <typename T, typename S>
bool coursework::detail::AvlTreeSetIterator<T, S>::operator!=(const
AvlTreeSetIterator& rhs) const
{
    return !(*this == rhs);
}

#endif // AVL_TREE_SET_ITERATOR_HPP
```

// AvlTreeSetNode.hpp

```
#ifndef AVL_TREE_SET_NODE_HPP
#define AVL_TREE_SET_NODE_HPP
```

```
#include <utility>

namespace coursework
{
    namespace detail
    {
        template <typename T>
        struct AvlTreeSetNode
        {
            const T key_;

            AvlTreeSetNode* parent_;
            AvlTreeSetNode* left_;
            AvlTreeSetNode* right_;

            int factor_;

            AvlTreeSetNode(T&& key,
                           AvlTreeSetNode* parent = nullptr,
                           AvlTreeSetNode* left = nullptr,
                           AvlTreeSetNode* right = nullptr);

            AvlTreeSetNode* rotateLeft(AvlTreeSetNode*& root) noexcept;
            AvlTreeSetNode* rotateRight(AvlTreeSetNode*& root) noexcept;
            AvlTreeSetNode* balance(AvlTreeSetNode*& root) noexcept;
        };
    }
}

template <typename T>
coursework::detail::AvlTreeSetNode<T>::AvlTreeSetNode(T&& key,
                                                      AvlTreeSetNode* parent,
                                                      AvlTreeSetNode* left,
                                                      AvlTreeSetNode* right):
    key_(std::move(key)),
    parent_(parent),
    left_(left),
    right_(right),
    factor_(0)
{}

template <typename T>
coursework::detail::AvlTreeSetNode<T>*
coursework::detail::AvlTreeSetNode<T>::rotateLeft(AvlTreeSetNode<T>*& root)
noexcept
{
    AvlTreeSetNode<T>* pivot = right_;

    if (pivot->factor_ == 1)
    {
        factor_ = 0;
        pivot->factor_ = 0;
    }
    else if (pivot->factor_ == 0)
    {
        factor_ = 1;
        pivot->factor_ = -1;
    }

    right_ = pivot->left_;
```

```
    if (pivot->left_ != nullptr)
    {
        pivot->left_->parent_ = this;
    }

    pivot->parent_ = parent_;

    if (parent_ != nullptr)
    {
        (this == parent_->left_ ? parent_->left_ : parent_->right_) = pivot;
    }
    else
    {
        root = pivot;
    }

    pivot->left_ = this;
    parent_ = pivot;

    return pivot;
}

template <typename T>
coursework::detail::AvlTreeSetNode<T>*
coursework::detail::AvlTreeSetNode<T>::rotateRight(AvlTreeSetNode<T>*& root)
noexcept
{
    AvlTreeSetNode<T>* pivot = left_;

    if (pivot->factor_ == -1)
    {
        factor_ = 0;
        pivot->factor_ = 0;
    }
    else if (pivot->factor_ == 0)
    {
        factor_ = -1;
        pivot->factor_ = 1;
    }

    left_ = pivot->right_;

    if (pivot->right_ != nullptr)
    {
        pivot->right_->parent_ = this;
    }

    pivot->parent_ = parent_;

    if (parent_ != nullptr)
    {
        (this == parent_->left_ ? parent_->left_ : parent_->right_) = pivot;
    }
    else
    {
        root = pivot;
    }

    pivot->right_ = this;
    parent_ = pivot;
```

```
    return pivot;
}

template <typename T>
coursework::detail::AvlTreeSetNode<T>*
coursework::detail::AvlTreeSetNode<T>::balance(AvlTreeSetNode<T>*& root)
noexcept
{
    if (factor_ == 2)
    {
        if (right_->factor_ < 0)
        {
            right_->rotateRight(root);
        }

        return rotateLeft(root);
    }

    if (factor_ == -2)
    {
        if (left_->factor_ > 0)
        {
            left_->rotateLeft(root);
        }

        return rotateRight(root);
    }

    return this;
}

#endif // AVL_TREE_SET_NODE_HPP
```

// AvlTreeMap.hpp

```
#ifndef AVL_TREE_MAP_HPP
#define AVL_TREE_MAP_HPP

#include <initializer_list>
#include <utility>
#include "AvlTreeMapIterator.hpp"
#include "AvlTreeMapConstIterator.hpp"
#include "AvlTreeMapNode.hpp"
#include "AvlTreeTraverseStrategy.hpp"

namespace coursework
{
    template <typename T, typename U>
    class AvlTreeMap
    {
        using Node = detail::AvlTreeMapNode<T, U>;

    public:

        using Iterator = detail::AvlTreeMapIterator<T, U>;
        using ConstIterator = detail::AvlTreeMapConstIterator<T, U,
detail::StraightInfixTraverse<Node>>;
        using ReverseIterator = detail::AvlTreeMapIterator<T, U,
detail::ReversedInfixTraverse<Node>>;
        using ConstReverseIterator = detail::AvlTreeMapConstIterator<T, U,
detail::ReversedInfixTraverse<Node>>;
```

```
        AvlTreeMap();
        AvlTreeMap(const AvlTreeMap& rhs);
        AvlTreeMap(AvlTreeMap&& rhs) noexcept;
        AvlTreeMap(std::initializer_list<std::pair<T, U>> rhs);
        template <typename InputIterator>
        AvlTreeMap(InputIterator begin, InputIterator end);

        AvlTreeMap& operator=(const AvlTreeMap& rhs);
        AvlTreeMap& operator=(AvlTreeMap&& rhs) noexcept;

        virtual ~AvlTreeMap() noexcept;

        Iterator begin();
        Iterator end();
        ConstIterator cbegin() const;
        ConstIterator cend() const;
        ConstIterator begin() const;
        ConstIterator end() const;

        ReverseIterator rbegin();
        ReverseIterator rend();
        ConstReverseIterator crbegin() const;
        ConstReverseIterator crend() const;
        ConstReverseIterator rbegin() const;
        ConstReverseIterator rend() const;

        Iterator insert(T key, U value);
        Iterator search(const T& key) const;
        Iterator remove(const T& key);

        const Node* getRoot() const;

    private:

        void clear(Node* rhs);
        Node* root_;
    };
}

template <typename T, typename U>
coursework::AvlTreeMap<T, U>::AvlTreeMap():
    root_(nullptr)
{}

template <typename T, typename U>
coursework::AvlTreeMap<T, U>::AvlTreeMap(const AvlTreeMap& rhs)
{
    AvlTreeMap<T, U> temp;

    for (const auto& i : rhs)
    {
        temp.insert(i.first, i.second);
    }

    root_ = temp.root_;
    temp.root_ = nullptr;
}

template <typename T, typename U>
```

```
coursework::AvlTreeMap<T, U>::AvlTreeMap(std::initializer_list<std::pair<T, U>>
rhs)
{
    AvlTreeMap<T, U> temp;

    for (const auto& i : rhs)
    {
        temp.insert(i.first, i.second);
    }

    root_ = temp.root_;
    temp.root_ = nullptr;
}

template <typename T, typename U>
coursework::AvlTreeMap<T, U>& coursework::AvlTreeMap<T, U>::operator=(const
AvlTreeMap& rhs)
{
    if (this != &rhs)
    {
        AvlTreeMap<T, U> temp(rhs);
        std::swap(root_, temp.root_);
    }

    return *this;
}

template <typename T, typename U>
coursework::AvlTreeMap<T, U>::AvlTreeMap(AvlTreeMap&& rhs) noexcept:
    root_(rhs.root_)
{
    rhs.root_ = nullptr;
}

template <typename T, typename U>
coursework::AvlTreeMap<T, U>& coursework::AvlTreeMap<T,
U>::operator=(AvlTreeMap&& rhs) noexcept
{
    if (this != &rhs)
    {
        clear(root_);
        root_ = rhs.root_;
        rhs.root_ = nullptr;
    }

    return *this;
}

template <typename T, typename U>
coursework::AvlTreeMap<T, U>::~AvlTreeMap() noexcept
{
    clear(root_);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::Iterator coursework::AvlTreeMap<T,
U>::begin()
{
    if (root_ == nullptr)
    {
        return end();
```

```
    }

    Node* res = root_;

    while (res->left_ != nullptr)
    {
        res = res->left_;
    }

    return Iterator(root_, res);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::Iterator coursework::AvlTreeMap<T,
U>::end()
{
    return Iterator(root_, nullptr);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ConstIterator coursework::AvlTreeMap<T,
U>::cbegin() const
{
    if (root_ == nullptr)
    {
        return cend();
    }

    Node* res = root_;

    while (res->left_ != nullptr)
    {
        res = res->left_;
    }

    return ConstIterator(root_, res);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ConstIterator coursework::AvlTreeMap<T,
U>::cend() const
{
    return ConstIterator(root_, nullptr);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ConstIterator coursework::AvlTreeMap<T,
U>::begin() const
{
    return cbegin();
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ConstIterator coursework::AvlTreeMap<T,
U>::end() const
{
    return cend();
}

template <typename T, typename U>
```

```
typename coursework::AvlTreeMap<T, U>::ReverseIterator coursework::AvlTreeMap<T,
U>::rbegin()
{
    if (root_ == nullptr)
    {
        return rend();
    }

    Node* res = root_;

    while (res->right_ != nullptr)
    {
        res = res->right_;
    }

    return ReverseIterator(root_, res);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ReverseIterator coursework::AvlTreeMap<T,
U>::rend()
{
    return ReverseIterator(root_, nullptr);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ConstReverseIterator
coursework::AvlTreeMap<T, U>::crbegin() const
{
    if (root_ == nullptr)
    {
        return crend();
    }

    Node* res = root_;

    while (res->right_ != nullptr)
    {
        res = res->right_;
    }

    return ConstReverseIterator(root_, res);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ConstReverseIterator
coursework::AvlTreeMap<T, U>::crend() const
{
    return ConstReverseIterator(root_, nullptr);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ConstReverseIterator
coursework::AvlTreeMap<T, U>::rbegin() const
{
    return rbegin();
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::ConstReverseIterator
coursework::AvlTreeMap<T, U>::rend() const
```

```
{
    return rend();
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::Iterator coursework::AvlTreeMap<T,
U>::insert(T key, U value)
{
    if (root_ == nullptr)
    {
        root_ = new Node(std::move(key), std::move(value));
        return begin();
    }

    Node* curr = root_;
    Node* prev = nullptr;

    while (curr != nullptr && key != curr->key_)
    {
        prev = curr;
        curr = key < curr->key_ ? curr->left_ : curr->right_;
    }

    if (curr != nullptr)
    {
        return end();
    }

    curr = new Node(std::move(key), std::move(value), prev);
    (curr->key_ < curr->parent_->key_ ? curr->parent_->left_ : curr->parent_-
>right_) = curr;

    bool isRebalanced = false;

    Node* prevBack = curr;
    Node* currBack = prev;

    while (currBack != nullptr && !isRebalanced)
    {
        if (prevBack == currBack->left_)
        {
            --currBack->factor_;
        }
        else if (prevBack == currBack->right_)
        {
            ++currBack->factor_;
        }

        currBack = currBack->balance(root_);

        isRebalanced = currBack->factor_ == 0;

        prevBack = currBack;
        currBack = currBack->parent_;
    }

    return Iterator(root_, curr);
}

template <typename T, typename U>
```

```
typename coursework::AvlTreeMap<T, U>::Iterator coursework::AvlTreeMap<T,
U>::search(const T& key) const
{
    Node* curr = root_;

    while (curr != nullptr && key != curr->key_)
    {
        curr = key < curr->key_ ? curr->left_ : curr->right_;
    }

    return Iterator(root_, curr);
}

template <typename T, typename U>
typename coursework::AvlTreeMap<T, U>::Iterator coursework::AvlTreeMap<T,
U>::remove(const T& key)
{
    Node* curr = root_;

    while (curr != nullptr && key != curr->key_)
    {
        curr = key < curr->key_ ? curr->left_ : curr->right_;
    }

    if (curr == nullptr)
    {
        return end();
    }

    Node* res = detail::stepForward(root_, curr);

    if (curr->left_ == nullptr && curr->right_ == nullptr)
    {
        if (curr != root_)
        {
            (curr == curr->parent_->left_ ? curr->parent_->left_ : curr-
>parent_->right_) = nullptr;
        }
        else
        {
            root_ = nullptr;
        }
    }
    else if (curr->left_ == nullptr || curr->right_ == nullptr)
    {
        Node* const currChild = curr->left_ != nullptr ? curr->left_ : curr-
>right_;

        if (curr == root_)
        {
            currChild->parent_ = nullptr;
            root_ = currChild;
        }
        else
        {
            currChild->parent_ = curr->parent_;
            (curr == curr->parent_->left_ ? curr->parent_->left_ : curr-
>parent_->right_) = currChild;
        }

        bool isRebalanced = false;
```

```
    Node* prevBack = currChild;
    Node* currBack = prevBack->parent_;

    while (currBack != nullptr && !isRebalanced)
    {
        if (prevBack == currBack->left_)
        {
            ++currBack->factor_;
        }
        else if (prevBack == currBack->right_)
        {
            --currBack->factor_;
        }

        currBack = currBack->balance(root_);

        isRebalanced = std::abs(currBack->factor_) == 1;

        prevBack = currBack;
        currBack = currBack->parent_;
    }
}
else
{
    Node* prev = nullptr;
    Node* const temp = curr;

    curr = curr->right_;

    while (curr->left_ != nullptr)
    {
        prev = curr;
        curr = curr->left_;
    }

    Node* prevBack = temp;

    const bool hasAnyChildren = prev != nullptr;

    (hasAnyChildren ? prev->left_ : temp->right_) = curr->right_;

    if (curr->right_ != nullptr)
    {
        curr->right_->parent_ = hasAnyChildren ? prev : temp;
    }

    *const_cast<T*>(&temp->key_) = std::move(curr->key_);

    bool isRebalanced = false;

    Node* currBack = prevBack->parent_;

    while (currBack != nullptr && !isRebalanced)
    {
        if (prevBack == currBack->left_)
        {
            ++currBack->factor_;
        }
        else if (prevBack == currBack->right_)
        {
```

```
                --currBack->factor_;
            }

            currBack = currBack->balance(root_);

            isRebalanced = std::abs(currBack->factor_) == 1;

            prevBack = currBack;
            currBack = currBack->parent_;
        }

        res = temp;
    }

    delete curr;
    return Iterator(root_, res);
}

template <typename T, typename U>
const typename coursework::AvlTreeMap<T, U>::Node* coursework::AvlTreeMap<T,
U>::getRoot() const
{
    return root_;
}

template <typename T, typename U>
void coursework::AvlTreeMap<T, U>::clear(Node* rhs)
{
    if (rhs != nullptr)
    {
        clear(rhs->left_);
        clear(rhs->right_);
        delete rhs;
    }
}

#endif // AVL_TREE_MAP_HPP
```

// AvlTreeMapIterator.hpp

```
#ifndef AVL_TREE_MAP_ITERATOR_HPP
#define AVL_TREE_MAP_ITERATOR_HPP

#include <iterator>
#include <utility>
#include "AvlTreeMapNode.hpp"
#include "AvlTreeTraverseStrategy.hpp"

namespace coursework
{
    template <typename T, typename U>
    class AvlTreeMap;

    namespace detail
    {
        template <typename T, typename U, typename S =
StraightInfixTraverse<AvlTreeMapNode<T, U>>>
        class AvlTreeMapIterator: std::iterator<std::bidirectional_iterator_tag,
const T>
        {
            friend class AvlTreeMap<T, U>;
```

```
            using Node = AvlTreeMapNode<T, U>;
            using Data = std::pair<const T&, U&>;

        public:

            AvlTreeMapIterator();
            AvlTreeMapIterator(const AvlTreeMapIterator&) = default;
            ~AvlTreeMapIterator() = default;

            AvlTreeMapIterator& operator=(const AvlTreeMapIterator&) = default;

            AvlTreeMapIterator& operator++();
            AvlTreeMapIterator operator++(int);
            AvlTreeMapIterator& operator--();
            AvlTreeMapIterator operator--(int);

            Data& operator*();
            const Data& operator*() const;

            Data* operator->();
            const Data* operator->() const;

            bool operator==(const AvlTreeMapIterator& rhs) const;
            bool operator!=(const AvlTreeMapIterator& rhs) const;

        private:

            Node* root_;
            Node* node_;
            explicit AvlTreeMapIterator(Node* root, Node* node);
        };
    }
}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapIterator<T, U, S>::AvlTreeMapIterator():
    root_(nullptr),
    node_(nullptr)
{}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapIterator<T, U, S>::AvlTreeMapIterator(Node* root,
Node* node):
    root_(root),
    node_(node)
{}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapIterator<T, U, S>&
coursework::detail::AvlTreeMapIterator<T, U, S>::operator++()
{
    node_ = S::next(root_, node_);
    return *this;
}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapIterator<T, U, S>
coursework::detail::AvlTreeMapIterator<T, U, S>::operator++(int)
{
    AvlTreeMapIterator<T, U, S> temp(*this);
    ++(*this);
```

```
    return temp;
}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapIterator<T, U, S>&
coursework::detail::AvlTreeMapIterator<T, U, S>::operator--()
{
    node_ = S::prev(root_, node_);
    return *this;
}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapIterator<T, U, S>
coursework::detail::AvlTreeMapIterator<T, U, S>::operator--(int)
{
    AvlTreeMapIterator<T, U, S> temp(*this);
    --(*this);
    return temp;
}

template <typename T, typename U, typename S>
typename coursework::detail::AvlTreeMapIterator<T, U, S>::Data&
coursework::detail::AvlTreeMapIterator<T, U, S>::operator*()
{
    return node_->data_;
}

template <typename T, typename U, typename S>
const typename coursework::detail::AvlTreeMapIterator<T, U, S>::Data&
coursework::detail::AvlTreeMapIterator<T, U, S>::operator*() const
{
    return **this;
}

template <typename T, typename U, typename S>
typename coursework::detail::AvlTreeMapIterator<T, U, S>::Data*
coursework::detail::AvlTreeMapIterator<T, U, S>::operator->()
{
    return &node_->data_;
}

template <typename T, typename U, typename S>
const typename coursework::detail::AvlTreeMapIterator<T, U, S>::Data*
coursework::detail::AvlTreeMapIterator<T, U, S>::operator->() const
{
    return &**this;
}

template <typename T, typename U, typename S>
bool coursework::detail::AvlTreeMapIterator<T, U, S>::operator==(const
AvlTreeMapIterator& rhs) const
{
    return root_ == rhs.root_ && node_ == rhs.node_;
}

template <typename T, typename U, typename S>
bool coursework::detail::AvlTreeMapIterator<T, U, S>::operator!=(const
AvlTreeMapIterator& rhs) const
{
    return !(*this == rhs);
}
```

```
#endif // AVL_TREE_MAP_ITERATOR_HPP

// AvlTreeMapConstIterator.hpp

#ifndef AVL_TREE_MAP_CONST_ITERATOR_HPP
#define AVL_TREE_MAP_CONST_ITERATOR_HPP

#include <iterator>
#include <utility>
#include "AvlTreeMapNode.hpp"
#include "AvlTreeTraverseStrategy.hpp"

namespace coursework
{
    template <typename T, typename U>
    class AvlTreeMap;

    namespace detail
    {
        template <typename T, typename U, typename S =
StraightInfixTraverse<AvlTreeMapNode<T, U>>>
        class AvlTreeMapConstIterator:
std::iterator<std::bidirectional_iterator_tag, const T>
        {
            friend class AvlTreeMap<T, U>;
            using Node = AvlTreeMapNode<T, U>;
            using Data = std::pair<const T&, const U&>;

        public:

            AvlTreeMapConstIterator();
            AvlTreeMapConstIterator(const AvlTreeMapConstIterator&) = default;
            ~AvlTreeMapConstIterator() = default;

            AvlTreeMapConstIterator& operator=(const AvlTreeMapConstIterator&) =
default;

            AvlTreeMapConstIterator& operator++();
            AvlTreeMapConstIterator operator++(int);
            AvlTreeMapConstIterator& operator--();
            AvlTreeMapConstIterator operator--(int);

            const Data& operator*() const;
            const Data* operator->() const;

            bool operator==(const AvlTreeMapConstIterator& rhs) const;
            bool operator!=(const AvlTreeMapConstIterator& rhs) const;

        private:

            Node* root_;
            Node* node_;
            explicit AvlTreeMapConstIterator(Node* root, Node* node);
        };
    }
}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapConstIterator<T, U, S>::AvlTreeMapConstIterator():
    root_(nullptr),
```

```
    node_(nullptr)
{}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapConstIterator<T, U,
S>::AvlTreeMapConstIterator(Node* root, Node* node):
    root_(root),
    node_(node)
{}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapConstIterator<T, U, S>&
coursework::detail::AvlTreeMapConstIterator<T, U, S>::operator++()
{
    node_ = S::next(root_, node_);
    return *this;
}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapConstIterator<T, U, S>
coursework::detail::AvlTreeMapConstIterator<T, U, S>::operator++(int)
{
    AvlTreeMapConstIterator<T, U, S> temp(*this);
    ++(*this);
    return temp;
}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapConstIterator<T, U, S>&
coursework::detail::AvlTreeMapConstIterator<T, U, S>::operator--()
{
    node_ = S::prev(root_, node_);
    return *this;
}

template <typename T, typename U, typename S>
coursework::detail::AvlTreeMapConstIterator<T, U, S>
coursework::detail::AvlTreeMapConstIterator<T, U, S>::operator--(int)
{
    AvlTreeMapConstIterator<T, U, S> temp(*this);
    --(*this);
    return temp;
}

template <typename T, typename U, typename S>
const typename coursework::detail::AvlTreeMapConstIterator<T, U, S>::Data&
coursework::detail::AvlTreeMapConstIterator<T, U, S>::operator*() const
{
    return node_->constData_;
}

template <typename T, typename U, typename S>
const typename coursework::detail::AvlTreeMapConstIterator<T, U, S>::Data*
coursework::detail::AvlTreeMapConstIterator<T, U, S>::operator->() const
{
    return &node_->constData_;
}

template <typename T, typename U, typename S>
bool coursework::detail::AvlTreeMapConstIterator<T, U, S>::operator==(const
AvlTreeMapConstIterator& rhs) const
```

```
{
    return root_ == rhs.root_ && node_ == rhs.node_;
}

template <typename T, typename U, typename S>
bool coursework::detail::AvlTreeMapConstIterator<T, U, S>::operator!=(const
AvlTreeMapConstIterator& rhs) const
{
    return !(*this == rhs);
}

#endif // AVL_TREE_MAP_CONST_ITERATOR_HPP
```

// AvlTreeMapNode.hpp

```
#ifndef AVL_TREE_MAP_NODE_HPP
#define AVL_TREE_MAP_NODE_HPP

#include <utility>

namespace coursework
{
    namespace detail
    {
        template <typename T, typename U>
        struct AvlTreeMapNode
        {
            const T key_;
            U value_;

            std::pair<const T&, U&> data_ = {key_, value_};
            std::pair<const T&, const U&> constData_ = {key_, value_};

            AvlTreeMapNode* parent_;
            AvlTreeMapNode* left_;
            AvlTreeMapNode* right_;

            int factor_;

            AvlTreeMapNode(T&& key,
                           U&& value,
                           AvlTreeMapNode* parent = nullptr,
                           AvlTreeMapNode* left = nullptr,
                           AvlTreeMapNode* right = nullptr);

            AvlTreeMapNode* rotateLeft(AvlTreeMapNode*& root) noexcept;
            AvlTreeMapNode* rotateRight(AvlTreeMapNode*& root) noexcept;
            AvlTreeMapNode* balance(AvlTreeMapNode*& root) noexcept;
        };
    }
}

template <typename T, typename U>
coursework::detail::AvlTreeMapNode<T, U>::AvlTreeMapNode(T&& key,
                                                         U&& value,
                                                         AvlTreeMapNode* parent,
                                                         AvlTreeMapNode* left,
                                                         AvlTreeMapNode* right):
    key_(std::move(key)),
    value_(std::move(value)),
    parent_(parent),
```

```
    left_(left),
    right_(right),
    factor_(0)
{}

template <typename T, typename U>
coursework::detail::AvlTreeMapNode<T, U>* coursework::detail::AvlTreeMapNode<T,
U>::rotateLeft(AvlTreeMapNode*& root) noexcept
{
    AvlTreeMapNode<T, U>* pivot = right_;

    if (pivot->factor_ == 1)
    {
        factor_ = 0;
        pivot->factor_ = 0;
    }
    else if (pivot->factor_ == 0)
    {
        factor_ = 1;
        pivot->factor_ = -1;
    }

    right_ = pivot->left_;

    if (pivot->left_ != nullptr)
    {
        pivot->left_->parent_ = this;
    }

    pivot->parent_ = parent_;

    if (parent_ != nullptr)
    {
        (this == parent_->left_ ? parent_->left_ : parent_->right_) = pivot;
    }
    else
    {
        root = pivot;
    }

    pivot->left_ = this;
    parent_ = pivot;

    return pivot;
}

template <typename T, typename U>
coursework::detail::AvlTreeMapNode<T, U>* coursework::detail::AvlTreeMapNode<T,
U>::rotateRight(AvlTreeMapNode*& root) noexcept
{
    AvlTreeMapNode<T, U>* pivot = left_;

    if (pivot->factor_ == -1)
    {
        factor_ = 0;
        pivot->factor_ = 0;
    }
    else if (pivot->factor_ == 0)
    {
        factor_ = -1;
        pivot->factor_ = 1;
```

```
    }

    left_ = pivot->right_;

    if (pivot->right_ != nullptr)
    {
        pivot->right_->parent_ = this;
    }

    pivot->parent_ = parent_;

    if (parent_ != nullptr)
    {
        (this == parent_->left_ ? parent_->left_ : parent_->right_) = pivot;
    }
    else
    {
        root = pivot;
    }

    pivot->right_ = this;
    parent_ = pivot;

    return pivot;
}

template <typename T, typename U>
coursework::detail::AvlTreeMapNode<T, U>* coursework::detail::AvlTreeMapNode<T,
U>::balance(AvlTreeMapNode*& root) noexcept
{
    if (factor_ == 2)
    {
        if (right_->factor_ < 0)
        {
            right_->rotateRight(root);
        }

        return rotateLeft(root);
    }

    if (factor_ == -2)
    {
        if (left_->factor_ > 0)
        {
            left_->rotateLeft(root);
        }

        return rotateRight(root);
    }

    return this;
}

#endif // AVL_TREE_MAP_NODE_HPP
```

// AvlTreeTraverseStrategy.hpp

```
#ifndef AVL_TREE_TRAVERSE_STRATEGY_HPP
#define AVL_TREE_TRAVERSE_STRATEGY_HPP

namespace coursework
```

```
{
    namespace detail
    {
        template <typename T>
        T* stepForward(T* root, T* node);

        template <typename T>
        T* stepBackward(T* root, T* node);

        template <typename T>
        struct StraightInfixTraverse
        {
            static T* next(T* root, T* node);
            static T* prev(T* root, T* node);
        };

        template <typename T>
        struct ReversedInfixTraverse
        {
            static T* next(T* root, T* node);
            static T* prev(T* root, T* node);
        };
    }
}

template <typename T>
T* coursework::detail::stepForward(T* root, T* node)
{
    T* curr = node;

    while (curr->right_ == nullptr || curr->key_ < node->key_)
    {
        if (curr == root && (root->right_ == nullptr || curr->key_ < node-
>key_))
        {
            return nullptr;
        }

        curr = curr->parent_;

        if (curr->key_ > node->key_)
        {
            return curr;
        }
    }

    curr = curr->right_;

    while (curr->left_ != nullptr)
    {
        curr = curr->left_;
    }

    return curr;
}

template <typename T>
T* coursework::detail::stepBackward(T* root, T* node)
{
    T* curr = node;
```

```
    while (curr->left_ == nullptr || curr->key_ > node->key_)
    {
        if (curr == root && (root->left_ == nullptr || curr->key_ > node->key_))
        {
            return nullptr;
        }

        curr = curr->parent_;

        if (curr->key_ < node->key_)
        {
            return curr;
        }
    }

    curr = curr->left_;

    while (curr->right_ != nullptr)
    {
        curr = curr->right_;
    }

    return curr;
}

template <typename T>
T* coursework::detail::StraightInfixTraverse<T>::next(T* root, T* node)
{
    return stepForward(root, node);
}

template <typename T>
T* coursework::detail::StraightInfixTraverse<T>::prev(T* root, T* node)
{
    return stepBackward(root, node);
}

template <typename T>
T* coursework::detail::ReversedInfixTraverse<T>::next(T* root, T* node)
{
    return stepBackward(root, node);
}

template <typename T>
T* coursework::detail::ReversedInfixTraverse<T>::prev(T* root, T* node)
{
    return stepForward(root, node);
}

#endif // AVL_TREE_TRAVERSE_STRATEGY_HPP
```

Приложение 2. Протоколы отладки

// tests.cpp

```
#include "EngRusDictionary.hpp"
#include <iostream>

#include "AvlTreeSetTests.hpp"
#include "AvlTreeMapTests.hpp"
#include "EngRusDictionaryTests.hpp"

int main()
{
    using namespace coursework;

    AvlTreeSet<int> set;

    testSetOutput(set);
    testSetInsert(set, 15);
    testSetInsert(set, 15);
    testSetInsert(set, 23);
    testSetInsert(set, 23);
    testSetInsert(set, 13);
    testSetInsert(set, 13);
    testSetInsert(set, 14);
    testSetInsert(set, 14);
    testSetInsert(set, 25);
    testSetInsert(set, 25);
    testSetInsert(set, 18);
    testSetInsert(set, 18);
    testSetInsert(set, 16);
    testSetInsert(set, 16);
    testSetInsert(set, 17);
    testSetInsert(set, 17);

    AvlTreeSet<int> setCopy = set;

    testSetInsert(setCopy, 50000);
    testSetInsert(setCopy, 1000000);
    testSetInsert(set, 30);

    AvlTreeSet<int> setCopy2;
    setCopy2 = setCopy;

    testSetInsert(setCopy2, 123456789);
    testSetInsert(setCopy, 987654321);

    AvlTreeSet<int> set2 = std::move(set);

    testSetRemove(set2, 15);
    testSetRemove(set2, 15);
    testSetRemove(set2, 13);
    testSetRemove(set2, 13);
    testSetRemove(set2, 25);
    testSetRemove(set2, 25);
    testSetRemove(set2, 18);
    testSetRemove(set2, 18);
    testSetRemove(set2, 16);
    testSetRemove(set2, 16);
    testSetRemove(set2, 17);
```

```
testSetRemove(set2, 17);
testSetRemove(set2, 23);
testSetRemove(set2, 23);
testSetRemove(set2, 14);
testSetRemove(set2, 14);
testSetRemove(set2, 30);
testSetRemove(set2, 30);

AvlTreeSet<int> set3;
set3 = std::move(set2);

testSetInsert(set3, 15);
testSetInsert(set3, 23);
testSetInsert(set3, 13);
testSetInsert(set3, 14);
testSetInsert(set3, 25);
testSetInsert(set3, 18);
testSetInsert(set3, 16);
testSetInsert(set3, 17);

testSetRemove(set3, 17);
testSetInsert(set3, 17);

testSetRemove(set3, 17);
testSetInsert(set3, 17);

testSetRemove(set3, 17);
testSetInsert(set3, 17);

testSetRemove(set3, 17);
testSetInsert(set3, 17);

testSetRemove(set3, 17);
testSetInsert(set3, 17);

testSetRemove(set3, 15);
testSetRemove(set3, 16);
testSetRemove(set3, 17);
testSetRemove(set3, 18);
testSetRemove(set3, 23);
testSetRemove(set3, 25);
testSetRemove(set3, 13);
testSetRemove(set3, 14);
testSetRemove(set3, 25);

testSetInsert(set3, 17);
testSetInsert(set3, 16);
testSetInsert(set3, 15);
testSetInsert(set3, 14);
testSetInsert(set3, 13);
testSetInsert(set3, 12);
testSetInsert(set3, 11);
testSetInsert(set3, 10);

AvlTreeSet<int> set4;

testSetInsert(set4, 15);
testSetInsert(set4, 12);
testSetInsert(set4, 20);
testSetInsert(set4, 21);
testSetInsert(set4, 18);
```

```
    testSetInsert(set4, 22);
    testSetInsert(set4, 19);

    AvlTreeSet<int> set5;

    testSetInsert(set5, 17);
    testSetInsert(set5, 18);
    testSetInsert(set5, 19);
    testSetInsert(set5, 20);
    testSetInsert(set5, 21);
    testSetInsert(set5, 22);
    testSetInsert(set5, 23);
    testSetInsert(set5, 24);

    AvlTreeSet<int> setInit = {7, 2, 9, 10, 28, 65, 37};
    testSetOutput(setInit);

    AvlTreeMap<std::string, int> map;

    testMapOutput(map);

    std::string existing1 = "existing1";

    testMapInsert(map, existing1, 10);

    std::cout << existing1 << "\n";

    std::string existing2 = "existing2";

    testMapInsert(map, std::move(existing2), 14);

    std::cout << existing2 << "\n";

    AvlTreeMap<std::string, int> map2 = std::move(map);

    testMapInsert(map2, std::string("object"), 200);
    testMapInsert(map2, std::string("object"), 200);
    testMapInsert(map2, std::string("move"), 500);
    testMapInsert(map2, std::string("move"), 500);
    testMapInsert(map2, std::string("русский текст"), -12);
    testMapInsert(map2, std::string("русский текст"), -12);
    testMapInsert(map2, std::string("avl tree"), -50);
    testMapInsert(map2, std::string("avl tree"), -50);
    testMapInsert(map2, std::string("a"), 1);
    testMapInsert(map2, std::string("a"), 2);
    testMapInsert(map2, std::string("b"), 2);
    testMapInsert(map2, std::string("b"), 1);

    AvlTreeMap<std::string, int> map3;
    map3 = std::move(map2);

    testMapRemove(map3, std::string("b"));
    testMapRemove(map3, std::string("move"));
    testMapRemove(map3, std::string("русский текст"));
    testMapRemove(map3, std::string("avl tree"));

    AvlTreeMap<std::string, int> mapCopy = map3;
    testMapInsert(mapCopy, std::string("xLALALALALALALA"), 10000000);

    AvlTreeMap<std::string, int> mapCopy2;
    mapCopy2 = map3;
```

```
    testMapInsert(mapCopy2, std::string("xRAPAPAPAPAPAPAP"), -9999999);

    AvlTreeMap<int, int> mapInit = {{2, 3}, {4, 2}, {13, 0}};
    testMapOutput(mapInit);

    EngRusDictionary dict;

    testDictInsert(dict, "good", "хороший");
    testDictInsert(dict, "good", "хороший");
    testDictInsert(dict, "good", "товар");
    testDictInsert(dict, "good", "товар");
    testDictInsert(dict, "bad", "плохой");
    testDictInsert(dict, "bad", "плохой");

    EngRusDictionary dict2 = std::move(dict);

    testDictInsert(dict2, "direct", "направление");
    testDictInsert(dict2, "direct", "направление");

    testDictRemove(dict2, "good", "хороший");
    testDictRemove(dict2, "good", "хороший");

    EngRusDictionary dict3;
    dict3 = std::move(dict2);

    testDictRemove(dict3, "direct", "направления");
    testDictRemove(dict3, "directs", "направление");
    testDictRemove(dict3, "direct", "направление");

    EngRusDictionary dictCopy = dict3;
    testDictInsert(dictCopy, "cloud", "облако");

    EngRusDictionary dictCopy2 = dict3;
    testDictInsert(dictCopy2, "loud", "громкий");

    return 0;
}
```

// AvlTreeSetTests.hpp

```
#ifndef AVL_TREE_SET_TESTS_HPP
#define AVL_TREE_SET_TESTS_HPP

#include <iostream>
#include <cmath>
#include "AvlTreeSet.hpp"

namespace coursework
{
    template <typename T>
    void testSetOutput(const AvlTreeSet<T>& tree)
    {
        for (const auto& i : tree)
        {
            std::cout << i << " ";
        }

        std::cout << "\n";
    }

    template <typename T>
```

```
    std::size_t getHeight(const detail::AvlTreeSetNode<T>* node)
    {
        if (node == nullptr)
        {
            return 0;
        }

        return 1 + std::max(getHeight(node->left_), getHeight(node->right_));
    }

    template <typename T>
    bool checkBalance(const detail::AvlTreeSetNode<T>* node)
    {
        if (node == nullptr)
        {
            return true;
        }

        int lh = getHeight(node->left_);
        int rh = getHeight(node->right_);

        return std::abs(rh - lh) <= 1 && checkBalance(node->left_) &&
checkBalance(node->right_);
    }

    template <typename T>
    void testSetInsert(AvlTreeSet<T>& tree, T value)
    {
        tree.insert(std::move(value));
        testSetOutput(tree);
        std::cout << checkBalance(tree.getRoot()) << "\n";
    }

    template <typename T>
    void testSetRemove(AvlTreeSet<T>& tree, T value)
    {
        tree.remove(std::move(value));
        testSetOutput(tree);
        std::cout << checkBalance(tree.getRoot()) << "\n";
    }
}

#endif // AVL_TREE_SET_TESTS_HPP
```

// AvlTreeMapTests.hpp

```
#ifndef AVL_TREE_MAP_TESTS_HPP
#define AVL_TREE_MAP_TESTS_HPP

#include <iostream>
#include <cmath>
#include "AvlTreeMap.hpp"

namespace coursework
{
    template <typename T, typename U>
    void testMapOutput(const AvlTreeMap<T, U>& tree)
    {
        for (const auto& i : tree)
        {
            std::cout << i.first << " " << i.second << " ";
```

```
        }

        std::cout << "\n";
    }

    template <typename T, typename U>
    std::size_t getHeight(const detail::AvlTreeMapNode<T, U>* node)
    {
        if (node == nullptr)
        {
            return 0;
        }

        return 1 + std::max(getHeight(node->left_), getHeight(node->right_));
    }

    template <typename T, typename U>
    bool checkBalance(const detail::AvlTreeMapNode<T, U>* node)
    {
        if (node == nullptr)
        {
            return true;
        }

        int lh = getHeight(node->left_);
        int rh = getHeight(node->right_);

        return std::abs(rh - lh) <= 1 && checkBalance(node->left_) &&
checkBalance(node->right_);
    }

    template <typename T, typename U>
    void testMapInsert(AvlTreeMap<T, U>& tree, T key, U value)
    {
        tree.insert(std::move(key), std::move(value));
        testMapOutput(tree);
        std::cout << checkBalance(tree.getRoot()) << "\n";
    }

    template <typename T, typename U>
    void testMapRemove(AvlTreeMap<T, U>& tree, T key)
    {
        tree.remove(std::move(key));
        testMapOutput(tree);
        std::cout << checkBalance(tree.getRoot()) << "\n";
    }
}

#endif // AVL_TREE_MAP_TESTS_HPP
```

// EngRusDictionaryTests.hpp

```
#ifndef ENG_RUS_DICTIONARY_TESTS_HPP
#define ENG_RUS_DICTIONARY_TESTS_HPP

#include "EngRusDictionary.hpp"

namespace coursework
{
    void testDictInsert(EngRusDictionary& dict, std::string key, std::string
value);
```

```
    void testDictRemove(EngRusDictionary& dict, const std::string& key, const
std::string& value);
}

#endif // ENG_RUS_DICTIONARY_TESTS_HPP
```

// EngRusDictionaryTests.cpp

```
#include "EngRusDictionaryTests.hpp"
#include <iostream>

void coursework::testDictInsert(EngRusDictionary& dict, std::string key,
std::string value)
{
    dict.insert(std::move(key), std::move(value));

    for (const auto& i : dict)
    {
        for (const auto& j : i.second)
        {
            std::cout << i.first << " - " << j << "\n";
        }
    }

    std::cout << "\n";
}

void coursework::testDictRemove(EngRusDictionary& dict, const std::string& key,
const std::string& value)
{
    dict.remove(key, value);

    for (const auto& i : dict)
    {
        for (const auto& j : i.second)
        {
            std::cout << i.first << " - " << j << "\n";
        }
    }

    std::cout << "\n";
}
```