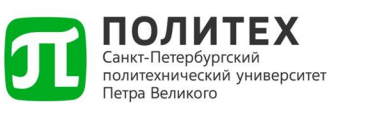
**Санкт-Петербургский политехнический университет Петра Великого  
  
Институт компьютерных наук и технологий  
Высшая школа программной инженерии**

****

**К У Р С О В А Я Р А Б О Т А**

**Алгоритмы работы со словарями**по дисциплине **«Алгоритмы и структуры данных»**

|  |  |
| --- | --- |
| Выполнил студент гр. | 3530904/10003  Майстренко Дмитрий |
| Руководитель | Шемякин Илья Александрович |
|  | **«\_\_» \_\_\_\_\_\_\_\_\_**2022 |

**Оглавление**

[**Введение. Общая постановка задачи:** 3](#_Toc105626674)

[**Основная часть работы.** 4](#_Toc105626675)

[**1.** **Описание алгоритма решения и используемых структур данных** 4](#_Toc105626676)

[**Красно-чёрное дерево** 4](#_Toc105626677)

[**Коллекция переводов** 6](#_Toc105626678)

[**2.** **Анализ алгоритма** 7](#_Toc105626679)

[**3.** **Описание спецификации программы (детальные требования)** 9](#_Toc105626680)

[*Требование к командам* 9](#_Toc105626681)

[*Приёмочные тесты* 9](#_Toc105626682)

[**4.** **Описание программы** 11](#_Toc105626683)

[**Заключение** 12](#_Toc105626684)

[**Список использованных источников** 13](#_Toc105626685)

[**Приложение 1 – Код программы** 14](#_Toc105626686)

[**./<ROOT>/Comparators.hpp** 14](#_Toc105626687)

[**./<ROOT>/Comparators.сpp** 15](#_Toc105626688)

[**./<ROOT>/Dictionary.hpp** 16](#_Toc105626689)

[**./<ROOT>/Dictionary.cpp** 17](#_Toc105626690)

[**./<ROOT>/errorRaisers.hpp** 19](#_Toc105626691)

[**./<ROOT>/errorRaisers.cpp** 20](#_Toc105626692)

[**./<ROOT>/Executors.hpp** 21](#_Toc105626693)

[**./<ROOT>/Executors.cpp** 22](#_Toc105626694)

[**./<ROOT>/RBT.hpp** 24](#_Toc105626695)

[**./<ROOT>/Set.hpp** 45](#_Toc105626696)

[**./<ROOT>/Transformators.hpp** 52](#_Toc105626697)

[**./<ROOT>/main.cpp** 53](#_Toc105626698)

[**Приложение 2 – Протоколы отладки** 55](#_Toc105626699)

# **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 запись с указанным ключом
   * DELETE (ключ) – удалить запись с указанным ключом
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>).

1. Тесты должны учитывать как допустимые, так и недопустимые последовательности входных данных.

**Вариант 1.1.4.**

**Англо-русский словарь. Красно-чёрное дерево.**Разработать и реализовать алгоритм работы с англо-русским словарём, реализованным как красно-чёрное дерево.

Узел дерева должен содержать:

* Ключ – английское слово
* Информационная часть – ссылка на список, содержащий переводы английского слова, отсортированные по алфавиту (переводов слова может быть несколько).

# **Основная часть работы.**

## **Описание алгоритма решения и используемых структур данных**

Англо-русский словарь – класс, основная задача которого добавлять, удалять и хранить данные. Одной единицей данных является английское слово и соответствующий ему список русских слов, являющихся смысловыми эквивалентами.

### **Красно-чёрное дерево** [1][2]

#### Общие сведения

Для хранения данных внутри класса была выбрана и самостоятельно реализована структура данных красно-чёрное дерево, которое является одним из видов самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и логарифмическое время поиска, вставки и удаления элементов.

Красное-чёрное дерево имеет следующие свойства:

* 1. Узел может быть либо красным, либо чёрным (так как данное свойство имеет только два состояния, оно организовано переменной логического типа) и имеет двух потомков (однако потомок может быть не только узлом, но и листом – то есть не содержать данных никаких данных, кроме цвета);
  2. Корень – как правило чёрный;
  3. Все листья – чёрные;
  4. Оба потомка каждого красного узла – чёрные;
  5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Данные ограничения гарантируют путь от корня до самого дальнего листа не более, чем вдвое длиннее, чем до самого длинного и дерево примерно – не строго – сбалансированно.

Балансировка происходит после каждых вставки и удаления, причём косвенно рекурсивным методом – практически каждый пункт подразумевает при несоблюдении условия в нём переход к следующему.

Все рекурсивные вызовы функций хвостовые и преобразуются в циклы, так что алгоритм требует памяти О(1).

В дереве были также организованы некоторые алгоритмы, которые не участвуют в командах словаря. Это было реализовано для общности структуры данных. Также, было принято решение использовать шаблон для универсальности и переносимости.

#### Вставка

Вставка в красно-чёрное дерево ничем не отличается от вставки в обычное бинарное дерево поиска за исключением того, что в красно-чёрном дереве новые данные помещаются в лист, после чего он становится узлом и приобретает свои листы.

Однако после вставки происходит самобалансировка дерева, опирающаяся на свойства красно-чёрного дерева.

Определим,

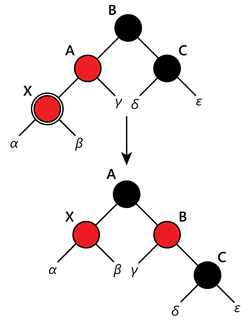
* Дедушка – родительский элемент родительского элемента данного узла;
* Дядя – дочерний элемент дедушки данного узла, не являющийся родительским;

Рис. 1

* Брат – второй потомок родителя данного узла;
* Поворот – изменение связей друг относительно друга четырёх элементов, а также смена цветов (Рис.1 – поворот вправо).

Балансировка дерева осуществляется, как уже было упомянуто, косвенной рекурсией. Выбор осуществлён в связи с тем фактом, что балансировка дочерних ряда узлов может нарушить балансировку родительских. В связи с чем требуется при балансировке идти снизу (от вставленного) вверх (к узлу), по пути рассматривая 5 различных ситуаций (текущий узел – узел, к которому применяется балансировка):

1. Текущий узел в корне дерева – перекрасить его в чёрный цвет;
2. Предок текущего узла чёрный – всё корректно (данный пункт нужен для исключения его из последующих ситуаций);
3. Родитель и дядя красные – перекрасить их в чёрный, а дедушку – в красный и вызов и перейти в ситуацию 1;
4. Родитель является красным, а дядя чёрным – выполнить поворот на родителе в сторону, зависящую от того, каким потомком (левым или правым) является текущий узел и его родитель (причём текущий и родитель должны быть разной стороны потомками);
5. Родитель является красным, а дядя чёрным – выполнить поворот на дедушке в сторону, зависящую от того, каким потомком (левым или правым) является текущий узел и его родитель (причём текущий и родитель должны быть одной стороны потомками);

Стоит отметить, что каждая ситуация (за исключением пятой), при невыполнении условия переходит в следующих пункт.

#### Удаление

Удаление происходит следующим образом: удаляемый элемент меняется местами с самым меньшим элементом правого поддерева или с самым большим левого. Затем, перемещённый удаляемый элемент удаляется и дерево балансируется. Такое перемещение упрощает балансировку – исключается ситуация, когда у текущего могут быть два потомка – и не нарушает свойство бинарного дерева поиска.

Далее, для балансировки рассматриваются 6 следующий ситуаций:

1. Текущий является корнем – ничего делать не требуется (данный пункт создан во избежание действий над корнем);
2. Брат является красным – меняем местами цвета родителя и брата и делаем поворот влево на родителе;
3. Родитель, брат и все потомки брата являются чёрными – перекрашиваем брата в красный;
4. Родитель - красный, а брат и все его потомки являются чёрными – меняем цвета брата и родителя;
5. Брат является чёрным, левый его потомок - красный, а правый - чёрный – делаем поворот на брате, в зависимости от того, каким потомком является текущий элемент;
6. Брат - чёрный, правый его потомок является красным – делаем поворот на родителе, в зависимости от того, каким потомком является текущий.

### **Коллекция переводов**

В качестве коллекции переводов было выбрано упорядоченное множество с уникальными элементами. Вставка гарантированно происходит за время О(n) и даёт нам возможность выполнять необходимое слияние словарей с аналогичной скоростью.

## **Анализ алгоритма** [2]

В данной работе используются 2 структуры данных: красно-чёрное дерево и упорядоченное множество.

* Упорядоченное множество имеет скорость О(n) (вставка, доступ, удаление);
* Красно-чёрное дерево имеет скорость O(log(n)) (вставка, доступ, удаление);

На графике 1 представлено последовательное добавление элементов по возрастанию (от 0 до 999999) и график зависимости количества необходимых операций сравнения и присваивания от количества элементов в дереве. Из него явно видно, что скорость вставки растёт логарифмически, как и предсказывала теория.

График 1

На графике 2 представлено последовательное удаление элементов по возрастанию (от 0 до 999999) и график зависимости количества необходимых операций сравнения и присваивания от количества элементов в дереве. На нём мы не видим явного очертания логарифмической функции, в связи с тем, что удалялись элементы не по порядку количества необходимых операций. Однако при проведении линии, соединяющей «верхние» точки, мы снова получим логарифмический график, что также подтверждает теорию.

График 2

## **Описание спецификации программы (детальные требования)**

### *Требование к командам*

Программа должна быть выполнена в виде консольного приложения и предусматривать обработку следующих команд (в угловых скобках указаны аргументы данной команды):

* INSERT <new-en-word> <new-ru-word> – вставка английского слова <new-en-word> и соответствующего ему русского слова  
  <new-ru-word>. Если <new-en-word> уже имеется в словаре, то <new-ru-word> добавляется в список переводов <new-en-word>.
* PRINT\_ALL – вывод всех английских слов, имеющихся в словаре, и соответствующих им переводов на русский язык.
* DELETE\_ENGLISH <en-word> – удаление английского слова   
  <en-word> и всех соответствующих ему переводов на русский язык.
* DELETE\_TRANSLATE <en-word> <ru-word> – удаление перевода <ru-word> в английском слове <en-word>.
* TRANSLATE <en-word> – вывод всех переводов <en-word>.

*Исключительные ситуации:*

* Если в словаре нет английских слов, но производится попытка использования команды PRINT\_ALL, то вывести сообщение: <DICTIONARY IS EMPTY>
* Если в <en-word> или в <ru-word> передаётся несуществующее в словаре английское слово или несуществующий перевод соответственно, то вывести сообщение: <NO SUCH WORD>
* Если производится попытка использования несуществующей команды, то вывести сообщение: <NO SUCH COMMAND>

### *Приёмочные тесты*

|  |  |  |
| --- | --- | --- |
| № | **Случай** | Тесты |
| 1 | Вставка нового слова в словарь. Результатом должно стать добавление нового слова с переводом в словарь. | **Input:**  INSERT WORD СЛОВО  PRINT\_ALL  **Expect:**  WORD – [ СЛОВО ] |
| 2 | Вставка уже существующего слова в словарь, но с другим переводом. Результатом должно стать пополнение списка переводов указанного английского слова. | **Input:**  INSERT WORD СЛОВО  INSERT WORD СЛОВЕЧКО  PRINT\_ALL  **Expect:**  WORD – [ СЛОВЕЧКО СЛОВО ] |
| 3 | Удаление слова из словаря. Результатом должно стать удаление английского слова и соответствующих ему переводов из словаря. | **Input:**  INSERT WORD СЛОВО  INSERT WORD СЛОВЕЧКО  INSERT TREE ДЕРЕВО  DELETE\_ENGLISH WORD  PRINT\_ALL  **Expect:**  TREE - [ ДЕРЕВО ] |
| 4 | Вывод переводов конкретного английского слова. Результатом должен стать вывод всех переводов указанного английского в стандартный поток вывода. | **Input:**  INSERT WORD СЛОВО  INSERT WORD СЛОВЕЧКО  INSERT TREE ДЕРЕВО  TRANSLATE TREE  **Expect:**  [ ДЕРЕВО ] |
| 5 | Удаление перевода. Результатом должно стать удаление указанного перевода указанного английского слова. | **Input:**  INSERT WORD СЛОВО  INSERT WORD СЛОВЕЧКО  INSERT TREE ДЕРЕВО  DELETE\_TRANSLATE WORD СЛОВЕЧКО  PRINT\_ALL  **Expect:**  TREE - [ ДЕРЕВО ]  WORD - [ СЛОВО ] |
| 6 | Вывод словаря. Результатом должен стать вывод всех слов с их переводами в стандартный поток вывода. | **Input:**  INSERT WORD СЛОВО  INSERT WORD СЛОВЕЧКО  INSERT TREE ДЕРЕВО  PRINT\_ALL  **Expect:**  TREE - [ ДЕРЕВО ]  WORD - [ СЛОВЕЧКО СЛОВО ] |
| 7 | Применение команды PRINT\_ALL к пустому словарю. Результатом должен стать вывод сообщения <DICTIONARY IS EMPTY> в стандартный поток ошибок. | **Input:**  PRINT\_ALL  **Expect:**  <DICTIONARY IS EMPTY> |
| 8 | Передача в аргумент <en-word> или <ru-word> несуществующее в словаре английское слово или несуществующий перевод соответственно. Результатом должен стать вывод в стандартный поток ошибок сообщения: <NO SUCH WORD>. | **Input:**  INSERT TREE ДЕРЕВО  INSERT TREE ДРЕВО  TRANSLATE WORD  DELETE\_ENGLISH WORD  DELETE\_TRANSLATE TREE ЯБЛОКО  **Expect:**  <NO SUCH WORD>  <NO SUCH WORD>  <NO SUCH WORD> |
| 9 | Попытка использования несуществующей команды. Результатом должен стать вывод в стандартный поток ошибок сообщения: <NO SUCH COMMAND>. | **Input:**  MAKE\_SOMETHING\_UNFORESEEN  **Expect:**  <NO SUCH COMMAND> |

## **Описание программы**

Программа выполнена в виде консольного приложения, взаимодействующей с пользователем непосредственно через командную строку.

Для взаимодействия с пользователем, используются следующие потоки:

* Стандартный поток ввода. Все команды и их аргументы принимаются с использованием стандартного потока ввода.
* Стандартный поток вывода. Корректно использованные команды, подразумевающие какой-либо вывод (PRINT\_ALL и TRANSLATE) производят вывод в стандартный поток вывода.
* Стандартный поток ошибок. Сообщения о некорректном использовании команды (или о попытке использования несуществующей команды) выводятся в стандартный поток ошибок.

# **Заключение**

В ходе данной выполнения данной курсовой работы была изучена структура и работа структуры данных Красно-чёрное дерево, её алгоритмы балансировки при вставке и удалении.

Были реализованы:

* 2 структуры данных:
  + Словарь на основе красно-чёрного дерево;
  + Множество на основе отсортированного списка с уникальными значениями;
* Класс, использующий приведённые выше структуры данных для добавления, изменения, удаления и вывода английских слов и их переводов;
* Интерфейс работы с классом, описанным выше, при помощи командной строки.

# **Список использованных источников**

|  |  |
| --- | --- |
| [1] | «Красно-чёрное дерево,» 2 Ноябрь 2019. [В Интернете]. Available: https://ru.wikipedia.org/wiki/Красно-чёрное\_дерево. |
| [2] | AnROm, «Красно-черные деревья: коротко и ясно,» 10 Июнь 2017. [В Интернете]. Available: https://habr.com/ru/post/330644/. |

# **Приложение 1 – Код программы**

./<ROOT>/Comparators.hpp

#ifndef COMPARATORS\_HPP

#define COMPARATORS\_HPP

#include <string>

struct StringComparator

{

bool operator()(const std::string&, const std::string&);

};

struct IntComparator

{

bool operator()(int, int);

};

#endif

./<ROOT>/Comparators.сpp

#include "Comparators.hpp"

bool StringComparator::operator()(const std::string& first, const std::string& second)

{

return first < second;

}

bool IntComparator::operator()(int first, int second)

{

return first < second;

}

./<ROOT>/Dictionary.hpp

#ifndef DICTIONARY\_HPP

#define DICTIONARY\_HPP

#include "RBT.hpp"

#include "Set.hpp"

#include "Comparators.hpp"

class Dictionary

{

public:

Dictionary() = default;

bool insertTranslate(const std::string& english, const std::string& russian);

bool deleteTranslate(const std::string& english, const std::string& russian);

bool deleteWord(const std::string& english);

bool translate(const std::string& english, std::ostream& out);

void print(std::ostream& out);

bool empty() const;

private:

using SetString = Set< std::string, StringComparator >;

RBT< std::string, SetString, StringComparator > translations;

};

#endif

./<ROOT>/Dictionary.cpp

#include "Dictionary.hpp"

#include "Transformators.hpp"

bool Dictionary::insertTranslate(const std::string& english, const std::string& russian)

{

SetString temp;

temp.insertItem(russian);

translations.insert\_or\_transform(english, temp, SetTransformator< std::string, StringComparator >{ temp });

return true;

}

bool Dictionary::deleteTranslate(const std::string& english, const std::string& russian)

{

try

{

SetString& trans = translations.at(english);

if (!trans.deleteItem(russian))

{

return false;

}

return true;

}

catch (const std::out\_of\_range&)

{

return false;

}

}

bool Dictionary::deleteWord(const std::string& english)

{

return translations.remove(english);

}

bool Dictionary::translate(const std::string& english, std::ostream& out)

{

try

{

SetString setStr;

setStr = translations.at(english);

out << setStr << '\n';

return true;

}

catch (const std::out\_of\_range&)

{

return false;

}

}

void Dictionary::print(std::ostream& out)

{

translations.printValues(out);

}

bool Dictionary::empty() const

{

return translations.empty();

}

./<ROOT>/errorRaisers.hpp

#ifndef ERRORS\_RAISERS\_HPP

#define ERRORS\_RAISERS\_HPP

void raiseEmptyDicitonary();

void raiseUnexistingWord();

void raiseUnexistingCommand();

#endif

./<ROOT>/errorRaisers.cpp

#include "errorsRaisers.hpp"

#include <stdexcept>

void raiseEmptyDicitonary()

{

throw std::invalid\_argument("<DICTIONARY IS EMPTY>\n");

}

void raiseUnexistingWord()

{

throw std::invalid\_argument("<NO SUCH WORD>\n");

}

void raiseUnexistingCommand()

{

throw std::invalid\_argument("<NO SUCH COMMAND>\n");

}

./<ROOT>/Executors.hpp

#ifndef EXECUTORS\_HPP

#define EXECUTORS\_HPP

#include <iosfwd>

#include <string>

#include "Dictionary.hpp"

void executeCommand(Dictionary&, const std::string&, std::ostream&, std::istream&);

#endif

./<ROOT>/Executors.cpp

#include "Executors.hpp"

#include <iostream>

#include "errorsRaisers.hpp"

void executeCommand(Dictionary& dict, const std::string& command, std::ostream& out, std::istream& in)

{

if (command == "INSERT")

{

std::string eng;

out << "English: ";

in >> eng;

std::string rus;

out << "Russian: ";

in >> rus;

dict.insertTranslate(eng, rus);

}

else if (command == "PRINT\_ALL")

{

if (!dict.empty())

{

dict.print(out);

}

else

{

raiseEmptyDicitonary();

}

}

else if (command == "DELETE\_ENGLISH")

{

std::string eng;

out << "English: ";

in >> eng;

if (!dict.deleteWord(eng))

{

raiseUnexistingWord();

}

}

else if (command == "TRANSLATE")

{

std::string eng;

out << "English: ";

in >> eng;

if (!dict.translate(eng, out))

{

raiseUnexistingWord();

}

}

else if (command == "DELETE\_TRANSLATE")

{

std::string eng;

out << "English: ";

in >> eng;

std::string rus;

out << "Russian: ";

in >> rus;

if (!dict.deleteTranslate(eng, rus))

{

raiseUnexistingWord();

}

}

else

{

raiseUnexistingCommand();

}

}

./<ROOT>/RBT.hpp

#ifndef RBT\_HPP

#define RBT\_HPP

#include <ostream>

#include <stdexcept>

#include "Transformators.hpp"

#ifdef COLORFUL

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <cstring>

#include <windows.h>

#undef max

namespace

{

void setcol(unsigned int a)

{

HANDLE handle = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(handle, a);

}

}

#endif

#ifdef TEST\_MODE

long long iter = 0;

#endif

template < typename Key\_t, typename Value\_t, typename KeyComp >

class RBT

{

public:

RBT();

RBT(const RBT< Key\_t, Value\_t, KeyComp >& other);

RBT(RBT< Key\_t, Value\_t, KeyComp >&& other) noexcept;

RBT< Key\_t, Value\_t, KeyComp >& operator=(const RBT< Key\_t, Value\_t, KeyComp >& other);

RBT< Key\_t, Value\_t, KeyComp >& operator=(RBT< Key\_t, Value\_t, KeyComp >&& other) noexcept;

~RBT();

size\_t getHeight() const;

bool isThere(const Key\_t& key) const;

bool insert(const Key\_t& key, const Value\_t& value);

template < class Transformator >

void insert\_or\_transform(const Key\_t& key, const Value\_t& value, Transformator trans);

bool remove(const Key\_t& key);

void printValues(std::ostream& out) const;

void printTree(std::ostream& out) const;

Value\_t& at(const Key\_t& key);

bool empty() const;

private:

struct Node

{

Node();

bool isNIL();

~Node();

Key\_t\* key\_;

Value\_t\* value\_;

bool isBlack;

Node\* left\_;

Node\* right\_;

Node\* p\_;

};

Node\* root\_;

void copyTree(Node\*& in, Node\* from);

size\_t getHeight(const Node\* node) const;

Node\* searchNode(const Key\_t& key) const;

void deleteTreePart(Node\* root);

Node\* insertNode(const Key\_t& key, const Value\_t& value);

bool insertNode(const Key\_t& key, const Value\_t& value, Node\*& interacted);

void printPart(const std::string& prefix, Node\* node, bool isLeft, std::ostream& out) const;

void inorderWalk(Node\* node, std::ostream& out) const;

Node\* getGrandpa(Node\* grandson);

Node\* getUncle(Node\* nephew);

Node\* getSibling(Node\* n);

void makeLeftRotation(Node\* n);

void makeRightRotation(Node\* n);

void insertCase1(Node\* inserted);

void insertCase2(Node\* inserted);

void insertCase3(Node\* inserted);

void insertCase4(Node\* inserted);

void insertCase5(Node\* inserted);

void replaceNode(Node\* n, Node\* child);

void deleteOneChild(Node\* n);

void deleteCase1(Node\* n);

void deleteCase2(Node\* n);

void deleteCase3(Node\* n);

void deleteCase4(Node\* n);

void deleteCase5(Node\* n);

void deleteCase6(Node\* n);

};

template< typename K, typename V, typename C >

RBT< K, V, C >::Node::Node():

key\_(nullptr),

value\_(nullptr),

p\_(nullptr),

left\_(nullptr),

right\_(nullptr),

isBlack(true)

{}

template< typename Key\_t, typename Value\_t, typename KeyComp >

bool RBT< Key\_t, Value\_t, KeyComp >::Node::isNIL()

{

return static\_cast< bool >(!key\_) && static\_cast< bool >(!value\_);

}

template< typename K, typename V, typename C >

RBT< K, V, C >::Node::~Node()

{

delete value\_;

delete key\_;

}

template< typename K, typename V, typename C >

RBT< K, V, C >::RBT():

root\_(new Node())

{}

template< typename K, typename V, typename C >

RBT< K, V, C >::RBT(const RBT< K, V, C >& other):

root\_(new Node())

{

copyTree(root\_, other.root\_);

}

template< typename K, typename V, typename C >

RBT< K, V, C >::RBT(RBT< K, V, C >&& other) noexcept:

root\_(nullptr)

{

std::swap(root\_, other.root\_);

}

template< typename K, typename V, typename C >

RBT< K, V, C >& RBT< K, V, C >::operator=(const RBT< K, V, C >& other)

{

if (this != &other)

{

RBT< K, V, C > temp(other);

std::swap(root\_, temp.root\_);

}

return \*this;

}

template< typename K, typename V, typename C >

RBT< K, V, C >& RBT< K, V, C >::operator=(RBT< K, V, C >&& other) noexcept

{

if (this != &other)

{

std::swap(root\_, other.root\_);

}

return \*this;

}

template< typename K, typename V, typename C >

RBT< K, V, C >::~RBT()

{

if (root\_)

{

deleteTreePart(root\_);

}

}

template<typename Key\_t, typename Value\_t, typename KeyComp>

size\_t RBT< Key\_t, Value\_t, KeyComp >::getHeight() const

{

return getHeight(root\_);

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

bool RBT< Key\_t, Value\_t, KeyComp >::isThere(const Key\_t& key) const

{

return static\_cast< bool >(searchNode(key));

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

bool RBT< Key\_t, Value\_t, KeyComp >::remove(const Key\_t& key)

{

Node\* forDelete = searchNode(key);

if (forDelete && root\_->left\_->isNIL() && root\_->right\_->isNIL())

{

delete root\_->left\_;

delete root\_->right\_;

delete root\_->key\_;

delete root\_->value\_;

root\_->left\_ = nullptr;

root\_->right\_ = nullptr;

root\_->key\_ = nullptr;

root\_->value\_ = nullptr;

#ifdef TEST\_MODE

iter += 8;

#endif

return true;

}

Node\* tempDel = nullptr;

#ifdef TEST\_MODE

iter += 6;

#endif

if (forDelete)

{

#ifdef TEST\_MODE

iter += 1;

#endif

if (!forDelete->right\_->isNIL())

{

tempDel = forDelete;

tempDel = tempDel->right\_;

#ifdef TEST\_MODE

iter += 2;

#endif

while (!tempDel->left\_->isNIL())

{

tempDel = tempDel->left\_;

#ifdef TEST\_MODE

iter += 2;

#endif

}

}

else if (!forDelete->left\_->isNIL())

{

tempDel = forDelete;

tempDel = tempDel->left\_;

#ifdef TEST\_MODE

iter += 3;

#endif

while (!tempDel->right\_->isNIL())

{

tempDel = tempDel->right\_;

#ifdef TEST\_MODE

iter += 2;

#endif

}

}

else

{

tempDel = forDelete;

#ifdef TEST\_MODE

iter += 1;

#endif

}

std::swap(forDelete->key\_, tempDel->key\_);

std::swap(forDelete->value\_, tempDel->value\_);

#ifdef TEST\_MODE

iter += 2;

#endif

deleteOneChild(tempDel);

return true;

}

else

{

return false;

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp>::printValues(std::ostream& out) const

{

if (!root\_->isNIL())

{

inorderWalk(root\_, out);

}

}

template< typename Key\_t, typename V, typename C >

void RBT< Key\_t, V, C >::printTree(std::ostream& out) const

{

printPart("", root\_, true, out);

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

Value\_t& RBT< Key\_t, Value\_t, KeyComp >::at(const Key\_t& key)

{

Node\* totalNode = searchNode(key);

if (totalNode)

{

return \*totalNode->value\_;

}

else

{

throw std::out\_of\_range("No element with such key\n");

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

bool RBT< Key\_t, Value\_t, KeyComp >::empty() const

{

return root\_->isNIL();

}

template< typename K, typename V, typename C >

bool RBT< K, V, C >::insert(const K& key, const V& value)

{

Node\* forBalance = insertNode(key, value);

#ifdef TEST\_MODE

iter += 2;

#endif

if (root\_->isNIL())

{

root\_ = forBalance;

#ifdef TEST\_MODE

iter += 1;

#endif

}

#ifdef TEST\_MODE

iter += 1;

#endif

if (forBalance)

{

insertCase1(forBalance);

return true;

}

return false;

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

template< class Tranformator >

void RBT<Key\_t, Value\_t, KeyComp>::insert\_or\_transform(const Key\_t& key, const Value\_t& value, Tranformator trans)

{

Node\* forBalance = nullptr;

bool isAlreadyIn = !insertNode(key, value, forBalance);

if (root\_->isNIL())

{

root\_ = forBalance;

}

if (!isAlreadyIn)

{

insertCase1(forBalance);

return;

}

else

{

Value\_t tempValue = \*forBalance->value\_;

delete forBalance->value\_;

forBalance->value\_ = new Value\_t(trans(tempValue));

}

}

template< typename K, typename V, typename C >

typename RBT< K, V, C >::Node\* RBT< K, V, C >::insertNode(const K& key, const V& value)

{

#ifdef TEST\_MODE

iter += 1;

#endif

Node\* interacted = nullptr;

insertNode(key, value, interacted);

return interacted;

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT<Key\_t, Value\_t, KeyComp >::copyTree(Node\*& in, Node\* from)

{

in->isBlack = from->isBlack;

if (!from->isNIL())

{

in->key\_ = new Key\_t(\*from->key\_);

in->value\_ = new Key\_t(\*from->value\_);

}

Node\* left = nullptr;

if (from->left\_)

{

left = new Node();

copyTree(left, from->left\_);

}

Node\* right = nullptr;

if (from->right\_)

{

right = new Node();

copyTree(right, from->right\_);

}

if (left)

{

left->p\_ = in;

in->left\_ = left;

}

if (right)

{

in->right\_ = right;

right->p\_ = in;

}

}

template< typename K, typename V, typename C >

size\_t RBT< K, V, C >::getHeight(const Node\* node) const

{

size\_t leftHeight = 0;

size\_t rightHeight = 0;

if (node->left\_)

{

leftHeight = getHeight(node->left\_) + 1;

}

if (node->right\_)

{

rightHeight = getHeight(node->right\_) + 1;

}

return std::max(leftHeight, rightHeight);

}

template< typename K, typename V, typename C >

typename RBT< K, V, C >::Node\* RBT< K, V, C >::searchNode(const K& key) const

{

Node\* current = root\_;

while (!current->isNIL())

{

if (C{}(key, \*current->key\_))

{

current = current->left\_;

}

else if (C{}(\*current->key\_, key))

{

current = current->right\_;

}

else

{

return current;

}

}

return nullptr;

}

template< typename K, typename V, typename C >

void RBT< K, V, C >::deleteTreePart(Node\* root)

{

if (root->left\_)

{

deleteTreePart(root->left\_);

}

if (root->right\_)

{

deleteTreePart(root->right\_);

}

delete root;

}

template< typename K, typename V, typename C >

void RBT< K, V, C >::makeLeftRotation(Node\* n)

{

#ifdef TEST\_MODE

iter += 3;

#endif

Node\* pivot = n->right\_;

pivot->p\_ = n->p\_;

if (n->p\_)

{

#ifdef TEST\_MODE

iter += 2;

#endif

(n->p\_->left\_ == n ? n->p\_->left\_ : n->p\_->right\_) = pivot;

}

#ifdef TEST\_MODE

iter += 2;

#endif

n->right\_ = pivot->left\_;

if (pivot->left\_)

{

#ifdef TEST\_MODE

iter += 1;

#endif

pivot->left\_->p\_ = n;

}

n->p\_ = pivot;

pivot->left\_ = n;

#ifdef TEST\_MODE

iter += 3;

#endif

if (n == root\_)

{

#ifdef TEST\_MODE

iter += 1;

#endif

root\_ = pivot;

}

}

template< typename K, typename V, typename C >

void RBT< K, V, C >::makeRightRotation(Node\* n)

{

#ifdef TEST\_MODE

iter += 3;

#endif

Node\* pivot = n->left\_;

pivot->p\_ = n->p\_;

if (n->p\_)

{

#ifdef TEST\_MODE

iter += 2;

#endif

(n->p\_->left\_ == n ? n->p\_->left\_ : n->p\_->right\_) = pivot;

}

#ifdef TEST\_MODE

iter += 2;

#endif

n->left\_ = pivot->right\_;

if (pivot->right\_)

{

#ifdef TEST\_MODE

iter += 1;

#endif

pivot->right\_->p\_ = n;

}

n->p\_ = pivot;

pivot->right\_ = n;

#ifdef TEST\_MODE

iter += 3;

#endif

if (n == root\_)

{

#ifdef TEST\_MODE

iter += 1;

#endif

root\_ = pivot;

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::insertCase1(Node\* inserted)

{

#ifdef TEST\_MODE

iter += 1;

#endif

if (!inserted->p\_)

{

#ifdef TEST\_MODE

iter += 1;

#endif

inserted->isBlack = true;

}

else

{

insertCase2(inserted);

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::insertCase2(Node\* inserted)

{

#ifdef TEST\_MODE

iter += 1;

#endif

if (inserted->p\_->isBlack)

{

return;

}

else

{

insertCase3(inserted);

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::insertCase3(Node\* inserted)

{

Node\* uncle = getUncle(inserted);

Node\* grandpa = nullptr;

#ifdef TEST\_MODE

iter += 4;

#endif

if (uncle && !uncle->isBlack)

{

inserted->p\_->isBlack = true;

uncle->isBlack = true;

grandpa = getGrandpa(inserted);

grandpa->isBlack = false;

#ifdef TEST\_MODE

iter += 4;

#endif

insertCase1(grandpa);

}

else

{

insertCase4(inserted);

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::insertCase4(Node\* inserted)

{

Node\* grandpa = getGrandpa(inserted);

#ifdef TEST\_MODE

iter += 3;

#endif

if (inserted == inserted->p\_->right\_ && inserted->p\_ == grandpa->left\_)

{

makeLeftRotation(inserted->p\_);

inserted = inserted->left\_;

}

else if (inserted == inserted->p\_->left\_ && inserted->p\_ == grandpa->right\_)

{

makeRightRotation(inserted->p\_);

inserted = inserted->right\_;

#ifdef TEST\_MODE

iter += 1;

#endif

}

insertCase5(inserted);

}

template< typename K, typename V, typename C >

void RBT< K, V, C >::insertCase5(Node\* inserted)

{

Node\* grandpa = getGrandpa(inserted);

#ifdef TEST\_MODE

iter += 1;

#endif

inserted->p\_->isBlack = true;

grandpa->isBlack = false;

#ifdef TEST\_MODE

iter += 4;

#endif

if (inserted == inserted->p\_->left\_ && inserted->p\_ == grandpa->left\_)

{

makeRightRotation(grandpa);

}

else

{

makeLeftRotation(grandpa);

}

}

template< typename K, typename V, typename C >

void RBT< K, V, C >::replaceNode(Node\* n, Node\* child)

{

child->p\_ = n->p\_;

(n == n->p\_->left\_ ? n->p\_->left\_ : n->p\_->right\_) = child;

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::deleteOneChild(Node\* n)

{

Node\* child = (n->right\_->isNIL() ? n->left\_ : n->right\_);

#ifdef TEST\_MODE

iter += 2;

#endif

replaceNode(n, child);

#ifdef TEST\_MODE

iter += 1;

#endif

if (n->isBlack)

{

#ifdef TEST\_MODE

iter += 1;

#endif

if (!child->isBlack)

{

child->isBlack = true;

#ifdef TEST\_MODE

iter += 1;

#endif

}

else

{

deleteCase1(child);

}

}

#ifdef TEST\_MODE

iter += 1;

#endif

delete n;

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::deleteCase1(Node\* n)

{

#ifdef TEST\_MODE

iter += 1;

#endif

if (n->p\_)

{

deleteCase2(n);

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::deleteCase2(Node\* n)

{

Node\* s = getSibling(n);

#ifdef TEST\_MODE

iter += 2;

#endif

if (!s->isBlack)

{

n->p\_->isBlack = false;

s->isBlack = true;

#ifdef TEST\_MODE

iter += 3;

#endif

if (n == n->p\_->left\_)

{

makeLeftRotation(n->p\_);

}

else

{

makeRightRotation(n->p\_);

}

}

deleteCase3(n);

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::deleteCase3(Node\* n)

{

Node\* s = getSibling(n);

#ifdef TEST\_MODE

iter += 5;

#endif

if (n->p\_->isBlack && s->isBlack && s->left\_->isBlack && s->right\_->isBlack)

{

s->isBlack = false;

#ifdef TEST\_MODE

iter += 1;

#endif

deleteCase1(n->p\_);

}

else

{

deleteCase4(n);

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::deleteCase4(Node\* n)

{

Node\* s = getSibling(n);

#ifdef TEST\_MODE

iter += 5;

#endif

if (!n->p\_->isBlack && s->isBlack && s->left\_->isBlack && s->right\_->isBlack)

{

s->isBlack = false;

n->p\_->isBlack = true;

#ifdef TEST\_MODE

iter += 2;

#endif

}

else

{

deleteCase5(n);

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::deleteCase5(Node\* n)

{

Node\* s = getSibling(n);

#ifdef TEST\_MODE

iter += 2;

#endif

if (s->isBlack)

{

#ifdef TEST\_MODE

iter += 3;

#endif

if (n == n->p\_->left\_ && s->right\_->isBlack && !s->left\_->isBlack)

{

s->isBlack = false;

s->left\_->isBlack = true;

#ifdef TEST\_MODE

iter += 2;

#endif

makeRightRotation(s);

}

else if (n == n->p\_->right\_ && s->left\_->isBlack && !s->right\_->isBlack)

{

s->isBlack = false;

s->right\_->isBlack = true;

#ifdef TEST\_MODE

iter += 5;

#endif

makeLeftRotation(s);

}

}

deleteCase6(n);

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::deleteCase6(Node\* n)

{

Node\* s = getSibling(n);

s->isBlack = n->p\_->isBlack;

n->p\_->isBlack = true;

#ifdef TEST\_MODE

iter += 4;

#endif

if (n == n->p\_->left\_)

{

s->right\_->isBlack = true;

#ifdef TEST\_MODE

iter += 1;

#endif

makeLeftRotation(n->p\_);

}

else

{

s->left\_->isBlack = true;

#ifdef TEST\_MODE

iter += 1;

#endif

makeRightRotation(n->p\_);

}

}

template< typename K, typename V, typename C >

bool RBT< K, V, C >::insertNode(const K& key, const V& value, Node\*& interacted)

{

if (root\_->isNIL())

{

root\_->key\_ = new K(key);

root\_->value\_ = new V(value);

root\_->left\_ = new Node();

root\_->left\_->p\_ = root\_;

root\_->right\_ = new Node();

root\_->right\_->p\_ = root\_;

#ifdef TEST\_MODE

iter += 7;

#endif

interacted = root\_;

return true;

}

else

{

Node\* current = root\_;

#ifdef TEST\_MODE

iter += 1;

#endif

while (true)

{

if (C{}(key, \*current->key\_))

{

#ifdef TEST\_MODE

iter += 2;

#endif

if (current->left\_->isNIL())

{

current->left\_->key\_ = new K(key);

current->left\_->value\_ = new V(value);

current->left\_->p\_ = current;

current->left\_->isBlack = false;

current->left\_->left\_ = new Node();

current->left\_->left\_->p\_ = current;

current->left\_->right\_ = new Node();

current->left\_->right\_->p\_ = current;

#ifdef TEST\_MODE

iter += 9;

#endif

interacted = current->left\_;

return true;

}

current = current->left\_;

#ifdef TEST\_MODE

iter += 2;

#endif

}

else if (C{}(\*current->key\_, key))

{

#ifdef TEST\_MODE

iter += 2;

#endif

if (current->right\_->isNIL())

{

current->right\_->key\_ = new K(key);

current->right\_->value\_ = new V(value);

current->right\_->p\_ = current;

current->right\_->isBlack = false;

current->right\_->left\_ = new Node();

current->right\_->left\_->p\_ = current;

current->right\_->right\_ = new Node();

current->right\_->right\_->p\_ = current;

#ifdef TEST\_MODE

iter += 9;

#endif

interacted = current->right\_;

return true;

}

current = current->right\_;

#ifdef TEST\_MODE

iter += 2;

#endif

}

else

{

#ifdef TEST\_MODE

iter += 1;

#endif

interacted = current;

return false;

}

}

}

}

template< typename K, typename V, typename C >

void RBT< K, V, C >::printPart(const std::string& prefix, Node\* node, bool isLeft, std::ostream& out) const

{

if (!node->isNIL())

{

out << prefix;

out << (isLeft ? " |--" : " ---");

#ifdef COLORFUL

int colorNum = node->isBlack ? 7 : 12;

setcol(colorNum);

#endif

out << ' ' << \*node->key\_ << std::endl;

#ifdef COLORFUL

setcol(7);

#endif

printPart(prefix + (isLeft ? " | " : " "), node->right\_, true, out);

printPart(prefix + (isLeft ? " | " : " "), node->left\_, false, out);

}

}

template< typename Key\_t, typename Value\_t, typename KeyComp >

void RBT< Key\_t, Value\_t, KeyComp >::inorderWalk(Node\* node, std::ostream& out) const

{

if (!node->left\_->isNIL())

{

inorderWalk(node->left\_, out);

}

out << \*node->key\_ << " - " << \*node->value\_ << '\n';

if (!node->right\_->isNIL())

{

inorderWalk(node->right\_, out);

}

}

template< typename K, typename V, typename C >

typename RBT< K, V, C >::Node\* RBT< K, V, C >::getGrandpa(Node\* grandson)

{

#ifdef TEST\_MODE

iter += 3;

#endif

if (grandson && grandson->p\_ && grandson->p\_->p\_)

{

return grandson->p\_->p\_;

}

else

{

return nullptr;

}

}

template< typename K, typename V, typename C >

typename RBT< K, V, C >::Node\* RBT< K, V, C >::getUncle(Node\* nephew)

{

Node\* grandpa = getGrandpa(nephew);

#ifdef TEST\_MODE

iter += 2;

#endif

if (!grandpa)

{

return nullptr;

}

#ifdef TEST\_MODE

iter += 1;

#endif

return (nephew->p\_ == grandpa->left\_ ? grandpa->right\_ : grandpa->left\_);

}

template< typename K, typename V, typename C >

typename RBT< K, V, C >::Node\* RBT< K, V, C >::getSibling(Node\* n)

{

#ifdef TEST\_MODE

iter += 1;

#endif

return (n == n->p\_->left\_ ? n->p\_->right\_ : n->p\_->left\_);

}

#endif

./<ROOT>/Set.hpp

#ifndef SET\_H

#define SET\_H

#include <stdexcept>

template < typename T, typename Comp >

class Set

{

public:

Set();

Set(const Set< T, Comp >& other);

Set(Set< T, Comp >&& other) noexcept;

Set< T, Comp >& operator=(const Set< T, Comp >& other);

Set< T, Comp >& operator=(Set< T, Comp >&& other) noexcept;

bool insertItem(const T& item);

bool deleteItem(const T& item);

bool isThere(const T& item) const;

int getCount() const;

bool empty() const;

void unionSets(Set< T, Comp >& other);

template < typename T, typename Comp >

friend std::ostream& operator<<(std::ostream& stream, const Set< T, Comp >& dict);

~Set();

private:

void swap(Set< T, Comp >& other) noexcept;

struct Node

{

Node(const T& item, Node\* nextNode = nullptr);

T item\_;

Node\* next\_;

};

Node\* head\_;

int count\_;

};

template < typename T, typename Comp >

Set< T, Comp >::Set():

head\_(nullptr),

count\_(0)

{}

template < typename T, typename Comp >

bool Set< T, Comp >::insertItem(const T& item)

{

Node\* current = head\_;

if (count\_ == 0)

{

head\_ = new Node(item);

count\_++;

return true;

}

else

{

if (Comp{}(item, head\_->item\_))

{

head\_ = new Node(item, head\_);

count\_++;

return true;

}

while (current->next\_ && !Comp{}(item, current->next\_->item\_))

{

current = current->next\_;

}

if (!Comp{}(current->item\_, item) && !Comp{}(item, current->item\_))

{

return false;

}

else

{

current->next\_ = new Node(item, current->next\_);

count\_++;

return true;

}

}

}

template < typename T, typename Comp >

Set< T, Comp >::Set(const Set< T, Comp >& other):

count\_(other.count\_),

head\_(nullptr)

{

Set tempThis;

tempThis.count\_ = other.count\_;

tempThis.head\_ = new Node(other.head\_->item\_);

Node\* current = tempThis.head\_;

Node\* nextCurrent = other.head\_->next\_;

while (nextCurrent)

{

current->next\_ = new Node(nextCurrent->item\_);

current = current->next\_;

nextCurrent = nextCurrent->next\_;

}

swap(tempThis);

}

template < typename T, typename Comp >

Set< T, Comp >::Set(Set< T, Comp >&& other) noexcept:

head\_(other.head\_),

count\_(other.count\_)

{

other.head\_ = nullptr;

other.count\_ = 0;

}

template < typename T, typename Comp >

bool Set< T, Comp >::deleteItem(const T& item)

{

Node\* current = head\_;

if (!Comp{}(head\_->item\_, item) && !Comp{}(item, head\_->item\_))

{

Node\* forDeleting = head\_;

head\_ = head\_->next\_;

delete forDeleting;

--count\_;

return true;

}

while (current->next\_ && !Comp{}(item, current->next\_->item\_))

{

if (!Comp{}(current->next\_->item\_, item) && !Comp{}(item, current->next\_->item\_))

{

Node\* forDeleting = current->next\_;

current->next\_ = current->next\_->next\_;

delete forDeleting;

--count\_;

return true;

}

current = current->next\_;

}

return false;

}

template < typename T, typename Comp >

bool Set< T, Comp >::isThere(const T& item) const

{

Node\* current = head\_;

while (current && !Comp{}(item, current->item\_))

{

current = current->next\_;

}

return current && !Comp{}(current->item\_, item) && !Comp{}(item, current->item\_);

}

template < typename T, typename Comp >

int Set< T, Comp >::getCount() const

{

return count\_;

}

template < typename T, typename Comp >

bool Set< T, Comp >::empty() const

{

return !head\_;

}

template < typename T, typename Comp >

Set< T, Comp >& Set< T, Comp >::operator=(const Set< T, Comp >& other)

{

if (this != &other)

{

Set temp(other);

swap(temp);

}

return \*this;

}

template < typename T, typename Comp >

Set< T, Comp >& Set< T, Comp >::operator=(Set< T, Comp >&& other) noexcept

{

if (this != &other)

{

swap(other);

}

return \*this;

}

template < typename T, typename Comp >

Set< T, Comp >::~Set()

{

Node\* current = head\_;

while (current)

{

Node\* forDelete = current;

current = current->next\_;

delete forDelete;

}

count\_ = 0;

}

template < typename T, typename Comp >

void Set< T, Comp >::unionSets(Set< T, Comp >& other)

{

if (this == &other)

{

throw std::invalid\_argument("You mustn't add a Set to itself\n");

}

Node\* curThis = head\_;

Node\* curOther = other.head\_;

if (!head\_)

{

swap(other);

return;

}

while (curOther)

{

if (Comp{}(curOther->item\_, head\_->item\_))

{

Node\* forAdding = curOther;

curOther = curOther->next\_;

forAdding->next\_ = head\_;

head\_ = forAdding;

curThis = head\_;

count\_++;

}

else if (curThis == head\_ && !Comp{}(curThis->item\_, curOther->item\_) && !Comp{}(curOther->item\_, curThis->item\_))

{

curOther = curOther->next\_;

}

else

{

while (curThis->next\_ && Comp{}(curThis->next\_->item\_, curOther->item\_))

{

curThis = curThis->next\_;

}

if (!curThis->next\_)

{

curThis->next\_ = curOther;

while (curOther)

{

curOther = curOther->next\_;

count\_++;

}

}

else if (Comp{}(curThis->next\_->item\_, curOther->item\_) || Comp{}(curOther->item\_, curThis->next\_->item\_))

{

Node\* forAdding = curOther;

curOther = curOther->next\_;

forAdding->next\_ = curThis->next\_;

curThis->next\_ = forAdding;

count\_++;

}

else

{

Node\* forDeleting = curOther;

curOther = curOther->next\_;

curThis = curThis->next\_;

delete forDeleting;

}

}

}

other.head\_ = nullptr;

other.count\_ = 0;

}

template < typename T, typename Comp >

std::ostream& operator<<(std::ostream& stream, const Set< T, Comp >& dict)

{

stream << "[ ";

typename Set< T, Comp >::Node\* current = dict.head\_;

while (current)

{

stream << current->item\_ << ' ';

current = current->next\_;

}

stream << ']';

return stream;

}

template < typename T, typename Comp >

void Set< T, Comp >::swap(Set< T, Comp >& other) noexcept

{

std::swap(head\_, other.head\_);

std::swap(count\_, other.count\_);

}

template < typename T, typename Comp >

Set< T, Comp >::Node::Node(const T& item, Node\* nextNode):

item\_(item),

next\_(nextNode)

{}

#endif

./<ROOT>/Transformators.hpp

#ifndef TRANSFORMATORS\_HPP

#define TRANSFORMATORS\_HPP

#include "Set.hpp"

template < typename T, typename Comp >

struct SetTransformator

{

Set< T, Comp > forUnion;

Set< T, Comp > operator()(Set< T, Comp > first)

{

first.unionSets(forUnion);

return first;

};

};

#endif

./<ROOT>/main.cpp

// #define TREE\_OUT

#ifdef TREE\_OUT

#define COLORFUL

#endif

#include <iostream>

#include "Dictionary.hpp"

#include "Executors.hpp"

int main()

{

#ifdef TEST\_MODE

RBT< int, int, IntComparator > numbers;

for (int i = 0; i < 1000000; i++)

{

iter = 0;

numbers.insert(i, i \* i);

if (i % 10000 == 0)

{

std::cout << iter << '\n';

}

}

std::cout << numbers.at(999999);

#elif defined TREE\_OUT

RBT< int, double, IntComparator > numbers;

for (int i = 0; i < 100; i++)

{

numbers.insert(i, 1.0 \* i \* i);

}

for (int i = 0; i < 100; i++)

{

numbers.insert\_or\_transform(i, 1.0 \* i \* i, [&](double curr)

{

return std::powl(curr, 1.5);

});

}

numbers.printTree(std::cout);

std::cout << '\n' << numbers.at(10) << '\n';

#else

Dictionary dict;

while (!std::cin.eof())

{

std::string command = "";

std::cout << "Command: ";

std::cin >> command;

try

{

executeCommand(dict, command, std::cout, std::cin);

}

catch (const std::invalid\_argument& e)

{

std::cerr << e.what();

}

}

#endif

return 0;

}

# **Приложение 2 – Протоколы отладки**

Общие свойства конфигурации:

* Разрядность платформы: х64
* Тип конфигурации: Приложение (.exe)
* Версия пакета SDK для Windows: 10.0
* Набор инструментов платформы: Visual Studio 2022 (v143)
* Стандарт языка C++: (стандарт ISO C++ 14)

Свойства конфигурации отладки:

* Ускоритель по умолчанию для AMP: Ускоритель ПО WARP