Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА» (РУТ (МИИТ))

Институт транспортной техники и систем управления

Кафедра «Управление и защита информации»

ОТЧЕТ
К КУРСОВОЙ РАБОТЕ

по дисциплине

«Языки программирования»

Работу выполнил
студент группы ТКИ-442 _____ И.А. Кожак
подпись, дата

Работу проверил _____ _____
подпись, дата

Москва 2022

Содержание

Постановка задачи

Разработать программу на языке Си++ (ISO/IEC 14882:2014), демонстрирующую решение поставленной задачи.

Общая часть

Разработать шаблоны классов, объекты которых реализуют типы данных, указанные ниже. Для этих шаблонов классов разработать необходимые конструкторы, деструктор, конструктор копирования. Разработать операции: добавления/удаления элемента (уточнено в задаче); получения количества элементов; доступа к элементу (перегрузить оператор []). Разработать итератор для указанных шаблонов классов.

Задачи

а) Шаблон «AVL Дерево». Добавление/удаление элемента.

1 Алгоритм решения задачи

1.1 Задача 1

В рамках решения данной задачи был разработан класс «`Tree`». Данный класс содержит следующие private поля:

- root – тип Node*, содержит корневой элемент дерева
- функции rotateright и rotateleft с возвращаемым значением Node*
- функция bfactor с возвращаемым значением int
- функция height с возвращаемым значением int
- функция fixheight
- функция balance с возвращаемым значением Node*
- функция destroy с возвращаемым значением void, используемая деструктором для очистки памяти
- рекурсивная функция добавления элемента
- рекурсивная функция удаления элемента

Также класс содержит следующие public поля:

- вспомогательный класс Node
- функция-геттер корня дерева getRoot с возвращаемым значением Node*
- конструктор по умолчанию
- деструктор
- функция добавления элемента
- функция удаления элемента
- функция поиска минимума, возвращаемое значение Node*
- функция поиска максимума, возвращаемое значение Node*
- функция поиска произвольного элемента, возвращаемое значение Node*
- три функции обхода и вывода дерева: PreOrder, InOrder, PostOrder
- функция подробного вывода дерева AdvOutput

При разработке этого класса был создан вспомогательный класс Node. Данный класс содержит следующие private поля: left, right, parent – тип Node*, data – тип T, height – тип int. Также класс содержит следующие public поля:

- конструктор по элементу
- конструктор по умолчанию
- функция-сеттер, задающая значение поля left

– функция-сеттер, задающая значение поля right

– функция-сеттер, задающая значение поля data

– функция-сеттер, задающая значение поля parent

– функция-сеттер, задающая значение поля height

– функция-геттер, возвращающая значение поля left

– функция-геттер, возвращающая значение поля right

– функция-геттер, возвращающая значение поля data

– функция-геттер, возвращающая значение поля parent

– функция-геттер, возвращающая значение поля height

Также при разработке этого класса был создан класс Iterator, содержащий private поля parent типа Tree<T>, current типа Node* и следующие public поля:

– конструктор по родительскому дереву

– конструктор копирования

– перегрузка оператора *

– перегрузка оператора ++ (постфиксный и префиксный)

– перегрузка оператора - - (постфиксный и префиксный)

– перегрузки операторов == и !=

2 Выполнение задания

2.1 Задача 1

a) Node

2.1.1 конструктор по элементу

Инициализирует поле data переданным в конструктор элементом, поля left, right и parent значением nullptr, а поле height значением 1

2.1.2 конструктор по умолчанию

Инициализирует поля left, right и parent значением nullptr, поле height значением 1, а поле data результатом вызова конструктора по умолчанию от переданного параметра шаблона

2.1.3 функции-сеттеры, задающие значения полей left, right, data, parent, height

Инициализирует соответствующие поля переданным в конструктор элементом соответствующего типа

2.1.4 функции-геттеры, возвращающие значения полей left, right, data, parent, height

Возвращают значения соответствующих полей

б) Iterator

2.1.5 конструктор по родительскому дереву

Инициализирует поля parent и current переданными в конструктор элементами (значение current по умолчанию nullptr)

2.1.6 конструктор копирования

Инициализирует поля класса соответствующими полями переданного в конструктор класса

2.1.7 перегрузка оператора *

Возвращает разыменованный указатель current

2.1.8 перегрузка оператора ++

1) Префиксный – Проверяет не содержится ли в поле current текущего класса или в поле root родительского класса значение nullptr. Если содержится – завершает вызов возвращая разыменованный указатель на текущий класс. В противном случае, если указатель на правое поддерево элемента current не равен nullptr, присваивает полю current значение результата поиска минимума в правом поддереве элемента current (фактически переходя в "самый левый"узел поддерева) и возвращает разыменованный указатель на текущий класс. Иначе, создает новую локальную переменную Current типа Node* и инициализирует ее элементом current текущего класса. После этого при первой возможности переходим "вправо вверх"(текущий элемент должен находится в левой ветви элемента, в который переходим). Пока это невозможно при переходе используем временный элемент Current.
2) Постфиксный – Создает копию класса, затем применяет на классе префиксную операцию ++ и возвращает сохраненную до этого копию класса.

2.1.9 перегрузка оператора - -

1) Префиксный – Проверяет не содержится ли в поле current текущего класса или в поле root родительского класса значение nullptr. Если содержится – завершает вызов возвращая разыменованный указатель на текущий класс. В противном случае, если указатель на левое поддерево элемента current не равен nullptr, присваивает полю current значение результата поиска максимума в левом поддереве элемента current (фактически переходя в "самый правый"узел поддерева) и возвращает разыменованный указатель на текущий класс. Иначе, создает новую локальную переменную Current типа Node* и инициализирует ее элементом current текущего класса. После этого при первой возможности переходим "влево вверх"(текущий элемент должен находится в правой ветви элемента, в который переходим). Пока это невозможно при переходе используем временный элемент Current.

2) Постфиксный – Создает копию класса, затем применяет на классе префиксную операцию - - и возвращает сохраненную до этого копию класса.

2.1.10 перегрузки операторов == и !=

Возвращают результаты соответствующих операций, используя как операнды соответствующие значения полей current переданных элементов

в) Tree

2.1.11 функция rotateright

Создает локальную перменную q типа Node* записывая в нее указатель на левое поддерево переданного элемента. Если указатель на родителя переданного элемента равен nullptr присваивает полю root значение q. Затем присваивает полю left переданного элемента значение поля right созданного элемента q. После присваивает полю right созданного элемента q значение переданного элемента. Затем присваивает полю parent созданного элемента значение поля parent переданного элемента. После присваивает полю parent переданного элемента значение q. Затем делает два вызова функции fixheight, передавая как параметр вначале q, затем p (переданный элемент), и завершает вызов возвращая q.

2.1.12 функция rotateleft

Создает локальную перменную p типа Node* записывая в нее указатель на правое поддерево переданного элемента. Если указатель на родителя переданного элемента равен nullptr присваивает полю root значение p. Затем присваивает полю right переданного элемента значение поля left созданного элемента p. После присваивает полю left созданного элемента p значение переданного элемента. Затем присваивает полю parent созданного элемента значение поля parent переданного элемента. После присваивает полю parent переданного элемента значение p. Затем делает два вызова функции fixheight, передавая как параметр вначале q (переданный элемент), затем p, и завершает вызов возвращая q.

2.1.13 функция bfactor

Возвращает разность результатов функций height от правого поддерева переданного элемента и от левого соответственно

2.1.14 функция height

Если значение переданного элемента не равно nullptr возвращает значение поля height переданного элемента, в противном случае возвращает 0

2.1.15 функция fixheight

Получает результаты вызова функции height от левого и правого поддерева переданного элемента и присваивает полю height переданного элемента наибольшее значение, из полученных результатов вызова функций, увеличенное на единицу

2.1.16 функция balance

Вызывает функцию fixheight от переданного узла. Затем получает результат вызова функции bfactor от переданного узла. В случае если результат вызова равен 2, получает результат вызова функции bfactor от правого поддерева переданного узла. Если он меньше нуля, присваивает правому поддереву переданного узла результат вызова функции rotateright от правого поддерева переданного узла. После возвращает результат вызова функции rotateleft от переданного узла. В случае если вызова функции bfactor от переданного узла равен -2, получает результат вызова функции bfactor от левого поддерева переданного узла. Если он больше нуля, присваивает левому поддереву переданного узла результат вызова функции rotateleft от левого поддерева переданного узла. После возвращает результат вызова функции rotateright от переданного узла. Если ни одно условие не выполнилось, возвращает переданный узел.

2.1.17 функция destroy

Рекурсивно проходит все дерево освобождая память от элементов, начиная с конца

2.1.18 функция-геттер корня дерева getRoot с возвращаемым значением Node*

Возвращает значение поля root

2.1.19 конструктор по умолчанию

Инициализирует поле root значением nullptr

2.1.20 деструктор

Вызывает функцию destroy передавая в нее значение поля root

2.1.21 рекурсивная функция добавления элемента

Рекурсивно проходит дерево, сравнивая данные, хранимые деревом, с данными, хранимые вставляемым элементом, для нахождения такого места добавления, чтобы после него структура не противоречила определению сбалансированного дерева. После выполнения вставки возвращает результат балансировки дерева (вызов функции balance от текущего элемента).

2.1.22 функция добавления элемента

Создает новый объект типа Node, инициализируя его переданными данными и передает указатель на него в рекурсивную функцию добавления элемента

2.1.23 рекурсивная функция удаления элемента

Если указатель на текущий элемент равен nullptr возвращает его. В противном случае с помощью рекурсивных вызово проходит дерево до тех пор, пока не будет найден удаляемый элемент, или пока дерево не закончится. В случае если удаляемый элемент был найден выполняет следующие действия: если значение хотя бы одного из полей найденного элемента right или left равно nullptr, получает значение не nullptr поля и копирует его данные в найденный элемент. Если значения обоих полей равны nullptr очищает память, выделенную под найденный элемент (удаляет элемент). Если значения обоих полей не равны nullptr получает наименьший элемент в правом поддереве (результат вызова Min от поля right найденного элемента), копирует из него данные в найденный элемент и присваивает полю right найденного элемента результат рекурсивного вызова функции удаления элемента от данных минимального элемента в правом поддереве и правого поддерева найденного элемента. Затем, проверяет итоговое значение найденного элемента. Если оно равно nullptr, возвращает его, завершая вызов. Иначе присваивает полю height найденного элемента максимальное значение из полей height его поддеревьев, увеличенное на единицу. После получает результат вызова функции bfactor от текущего элемента, и перебирая все возможные сочетания результата этого вызова и результата вызова этой же функции от поддеревьев найденного элемента выполняет необходимое для балансировки расбалансированного дерева действие: rotateright для значений > 1 и $>= 0$ для левого поддерева, Current->setLeft(rotateleft(Current->getLeft())); rotateright для > 1 и < 0 для левого поддерева, rotateleft для < -1 и $<= 0$ для правого поддерева, Current->setRight(rotateright(Current->getRight())); rotateleft для < -1 и > 0 для правого поддерева. В случае невыполнения ни одного из условий возвращает найденный элемент.

2.1.24 функция удаления элемента

Вызывает рекурсивную функция удаления элемента, передавая как текущий элемент значение поля root

2.1.25 функция поиска минимума, возвращаемое значение Node*

Рекурсивно проходит дерево до самого "левого"узла и возвращает указатель на него (вызовы совершаются пока значение поля left рассматриваемого элемента не станет равно nullptr)

2.1.26 функция поиска максимума, возвращаемое значение Node*

Рекурсивно проходит дерево до самого "правого"узла и возвращает указатель на него (вызовы совершаются пока значение поля right рассматриваемого элемента не станет равно nullptr)

2.1.27 функция поиска произвольного элемента, возвращаемое значение Node*

Рекурсивно проходит все элементы дерева, сравнивая данные, хранимые ими с данными, переданными в функцию

2.1.28 три функции обхода и вывода дерева: PreOrder, InOrder, PostOrder

Рекурсивно проходят дерево, вызывая функцию вывода данных, хранимых элементом в определенное время: для PreOrder сразу после вхождения в функцию (вызов), для InOrder после завершения рекурсивных вызовов от левого поддерва (при таком выводе получаем упорядоченную последовательность), для PostOrder после окончания цепочки рекурсивных вызовов.

2.1.29 функция подробного вывода дерева

Соответствуют InOrder обходу, но помимо данных, хранимых узлом, выводит информация о поддеревьях, на которые он указывает

3 Получение исполняемых модулей

Для получения исполняемых модулей была использована система сборки cmake, также был написан файл CMakeLists.txt. В нем были выставлены следующие параметры сборки:

- std=C++14 – стандарт языка
- флаги компилятора:
  - Wall
  - pedantic-errors
  - fsanitize=undefined
- исполняемые файлы
  - linkedListTest
  - TreeTest

3.1 Файл `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.14)
project(AVL_Tree)

SET(CMAKE_CXX_STANDARD 14)

set(CMAKE_CXX_FLAGS
"${CMAKE_CXX_FLAGS} -Wall -pedantic-errors -fsanitize=undefined --
   coverage")
set(CMAKE_CXX_OUTPUT_EXTENSION_REPLACE ON)

set(OBJECT_DIR ${CMAKE_BINARY_DIR}/CMakeFiles/tree_gtest.dir/src)
message("-- Object files will be output to: ${OBJECT_DIR}")

include(FetchContent)
FetchContent_Declare(
  googletest
```

```
  URL https://github.com/google/googletest/archive/03597a01ee50ed33e9dfd640b249b4be3799d395.zip
)
# For Windows: Prevent overriding the parent project's compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)

#enable_testing()

add_executable(
  tree_gtest
  ../gtest.cpp
)
target_link_libraries(
  tree_gtest
  GTest::gtest_main
)

include(GoogleTest)

add_custom_target(init
    COMMAND ${CMAKE_MAKE_PROGRAM} clean
    COMMAND rm -f ${OBJECT_DIR}/*.gcno
    COMMAND rm -f ${OBJECT_DIR}/*.gcda
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
    )


add_custom_target(gcov
    COMMAND mkdir -p gcoverage
    #COMMAND ${CMAKE_MAKE_PROGRAM} test
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
    )
add_custom_command(TARGET gcov
    COMMAND echo "=================== GCOV ===================="
    COMMAND gcov -b ${CMAKE_SOURCE_DIR}/src/*.cpp -o ${OBJECT_DIR}
    COMMAND echo "-- Source diretorie: ${CMAKE_SOURCE_DIR}/src/"
    COMMAND echo "-- Coverage files have been output to ${CMAKE_BINARY_DIR}/gcoverage"
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/gcoverage
    )
add_dependencies(gcov tree_gtest)

gtest_discover_tests(tree_gtest)
```

Приложение А

А.1 Файл Node.h

```
#include <iostream>

namespace AVL_Tree {

    template<class T>
    class Node
        {
        protected:
            //закрытые переменные Node N; N.data = 10 вызовет ошибку
            T data;
            //не можем хранить Node, но имеем право хранить указатель
            Node* left;
            Node* right;
            Node* parent;
            //переменная, необходимая для поддержания баланса дерева
            int height;
        public:
            //доступные извне переменные и функции
            void setData(const T& d);

            T getData() const;

            int getHeight() const;

            Node* getLeft() const;

            Node* getRight() const;

            Node* getParent() const;

            void setLeft(Node* N);

            void setRight(Node* N);

            void setParent(Node* N);

            //Конструктор. Устанавливаем стартовые значения для указателей
            Node(T n);

            Node();

            void setHeight(int h);
        };
}

#include "Node.cpp"
```

A.2 Файл Node.cpp

```
namespace AVL_Tree {

        //доступные извне переменные и функции
    template<class T>
        void Node<T>::setData(const T& d) { data = d; }

    template<class T>
        T Node<T>::getData() const { return data; }

    template<class T>
        int Node<T>::getHeight() const { return height; }

    template<class T>
        Node<T>* Node<T>::getLeft() const { return left; }

    template<class T>
        Node<T>* Node<T>::getRight() const { return right; }

    template<class T>
        Node<T>* Node<T>::getParent() const { return parent; }

    template<class T>
        void Node<T>::setLeft(Node<T>* N) { left = N; }

    template<class T>
        void Node<T>::setRight(Node<T>* N) { right = N; }

    template<class T>
        void Node<T>::setParent(Node<T>* N) { parent = N; }

        //Конструктор. Устанавливаем стартовые значения для указателей
    template<class T>
        Node<T>::Node(T n)
        {
                data = n;
                left = right = parent = nullptr;
                height = 1;
        }

    template<class T>
        Node<T>::Node()
        {
                left = right = parent = nullptr;
                data = T();
                height = 1;
        }

    template<class T>
        void Node<T>::setHeight(int h)
```

```
        {
                height = h;
        }

    template<class T>
        std::ostream& operator<< (std::ostream& stream, const Node<T>& N) {
                stream << N.getData();
                return stream;
        };

}
```

Приложение Б

Б.1 Файл AVL$_T$ree.h

```
#include <iostream>
#include "Node.h"
#include "Iterator.h"

namespace AVL_Tree {

    template<class T>
    class Tree
    {
    private:
        //корень - его достаточно для хранения всего дерева
        Node<T>* root;

        Node<T>* rotateright(Node<T>* p); // правый поворот вокруг p

        Node<T>* rotateleft(Node<T>* q); // левый поворот вокруг q

        int bfactor(Node<T>* p) const;

        int height(Node<T>* p) const;

        void fixheight(Node<T>* p);

        Node<T>* balance(Node<T>* p); // балансировка узла p

        void destroy(Node<T>* current);

        //рекуррентная функция добавления узла. Устроена аналогично, но вызывает
↪   сама себя - добавление в левое или правое поддерево

        Node<T>* Add_R(Node<T>* N, Node<T>* Current);

        Node<T>* Remove_R(const T& data, Node<T>* Current);

    public:
        //доступ к корневому элементу
        Node<T>* getRoot() const;

        //конструктор дерева: в момент создания дерева ни одного узла нет, корень
↪   смотрит в никуда
        Tree<T>(const std::initializer_list<T> = {});

        Tree<T>(const Tree<T>& other);

        ~Tree<T>();
```

```
        //функция для добавления числа. Делаем новый узел с этими данными и
↪   вызываем нужную функцию добавления в дерево
        void Add(T n);

        void Remove(T n);

        Node<T>* Min(Node<T>* Current=nullptr) const;

        Node<T>* Max(Node<T>* Current = nullptr) const;

        //поиск узла в дереве. Второй параметр - в каком поддереве искать, первый
↪   - что искать
        Node<T>* Find(T data, Node<T>* Current) const;

        //три обхода дерева
        void PreOrder(Node<T>* N, std::ostream& stream) const;

        //InOrder-обход даст отсортированную последовательность
        void InOrder(Node<T>* N, std::ostream& stream) const;

        void PostOrder(Node<T>* N, std::ostream& stream) const;

        void AdvOutput(Node<T>* N, std::ostream& stream) const;

        Node<T>* getByIndex(const size_t index, Node<T>* Current, size_t&
↪   current_index) const;

        Node<T>* operator[](const size_t index) const;

        Tree<T>& operator=(const Tree<T>& other);

        iterator<T> begin() const;

        iterator<T> end() const;
    };
}

#include "AVL_Tree.cpp"
```

Б.2 Файл AVL$_T$*ree.cpp*

```
#include <algorithm>
namespace AVL_Tree {

        template<class T>
        Node<T>* Tree<T>::rotateright(Node<T>* p) // правый поворот вокруг p
        {
                if (!p) return p;

                Node<T>* q = p->getLeft();
                if (p->getParent() == nullptr) root = q;
                p->setLeft(q ? q->getRight() : q);
                if (q) {
                        q->setRight(p);
                        q->setParent(p ? p->getParent() : p);
                }
                p->setParent(q);

                fixheight(p);
                fixheight(q);
                return q;
        }

        template<class T>
        Node<T>* Tree<T>::rotateleft(Node<T>* q) // левый поворот вокруг q
        {
                if (!q) return q;

                Node<T>* p = q->getRight();
                if (q->getParent() == nullptr) root = p;
                q->setRight(p ? p->getLeft() : p);
                if (p) {
                        p->setLeft(q);
                        p->setParent(q ? q->getParent() : q);
                }
                q->setParent(p);

                fixheight(q);
                fixheight(p);
                return p;
        }

        template<class T>
        int Tree<T>::bfactor(Node<T>* p) const
        {
                return p ? (height(p->getRight())-height(p->getLeft())) : 0;
        }

        template<class T>
        int Tree<T>::height(Node<T>* p) const
```

```
{
        return p?p->getHeight():0;
}

template<class T>
void Tree<T>::fixheight(Node<T>* p)
{
        if (!p) return;
        int hl = height(p->getLeft());
        int hr = height(p->getRight());
        p->setHeight((hl>hr?hl:hr)+1);
}

template<class T>
Node<T>* Tree<T>::balance(Node<T>* p) // балансировка узла p
{
        fixheight(p);
        if( bfactor(p)==2 )
        {
                if( bfactor(p->getRight()) < 0 )
                        p->setRight(rotateright(p->getRight()));
                return rotateleft(p);
        }
        if( bfactor(p)==-2 )
        {
                if( bfactor(p->getLeft()) > 0  )
                        p->setLeft(rotateleft(p->getLeft()));
                return rotateright(p);
        }
        return p; // балансировка не нужна
}

template<class T>
void Tree<T>::destroy(Node<T>* current) {
        if (current == nullptr)
                return;

        if (current->getLeft())
                destroy(current->getLeft());

        if (current->getRight())
                destroy(current->getRight());

        delete current;
}

    //рекуррентная функция добавления узла. Устроена аналогично, но вызывает
↪   сама себя - добавление в левое или правое поддерево
    template<class T>
    Node<T>* Tree<T>::Add_R(Node<T>* N, Node<T>* Current)
```

```
{
        if (N == nullptr) return nullptr;
        if (root == nullptr)
        {
                root = N;
                return N;
        }

        if (Current->getData() > N->getData())
        {
                //идем влево
                if (Current->getLeft() != nullptr)
                        Current->setLeft(Add_R(N, Current->getLeft()));
                else
                        Current->setLeft(N);
                        Current->getLeft()->setParent(Current);
        }
        if (Current->getData() < N->getData())
        {
                //идем вправо
                if (Current->getRight() != nullptr)
                        Current->setRight(Add_R(N, Current->getRight()));
                else
                        Current->setRight(N);
                        Current->getRight()->setParent(Current);
        }
        if (Current->getData() == N->getData())
        //нашли совпадение
                ;

        return balance(Current);
}

template<class T>
Node<T>* Tree<T>::Remove_R(const T& data, Node<T>* Current)
{
        if (Current == nullptr)
                return Current;

        //ищем элемент
        if ( data < Current->getData() )
                Current->setLeft(Remove_R(data, Current->getLeft()));

        else if( data > Current->getData() )
                Current->setRight(Remove_R(data, Current->getRight()));

        //нашли
        else if (data == Current->getData())
        {
                // у элемента одного поддерево или нет вообще
```

```
                        if( (Current->getLeft() == nullptr) ||
                                (Current->getRight() == nullptr) )
                        {
                                Node<T> *temp = Current->getLeft() ?
                                                        Current->getLeft() :
                                                        Current->getRight();

                                //нет поддеревьев
                                if (temp == nullptr)
                                {
                                        temp = Current;
                                        Current = nullptr;
                                }
                                else //одно поддерево
                                *Current = *temp; //копируем данные
                                                        // не пустого поддерева
                                free(temp);
                        }
                        else
                        {
                                // два поддерева
                                // получаем наименьшее в правом поддереве
                                Node<T>* temp = Min(Current->getRight());

                                // копируем из него данные в текущий элемент
                                Current->setData(temp->getData());

                                //рекурсивно вызываем функцию
                                Current->setRight((Remove_R(temp->getData(),
↪  Current->getRight())));
                        }
                }

                //если в дереве только один элемент
                if (Current == nullptr)
                return Current;

                //Обновляем высоту текущего элемента
                Current->setHeight(1 + std::max(height(Current->getLeft()),
↪  height(Current->getRight())));

                //получаем результат вызова bfactor от текущего элемента
                int balance = bfactor(Current);

                //проверяем не расбалансировался ли элемент

                // Первый случай
                if (balance > 1 && bfactor(Current->getLeft()) >= 0)
                        return rotateright(Current);
```

```
                // Второй случай
                if (balance > 1 && bfactor(Current->getLeft()) < 0)
                {
                        Current->setLeft(rotateleft(Current->getLeft()));
                        return rotateright(Current);
                }

                // Третий случай
                if (balance < -1 && bfactor(Current->getRight()) <= 0)
                        return rotateleft(Current);

                // Четвертый случай
                if (balance < -1 && bfactor(Current->getRight()) > 0)
                {
                        Current->setRight(rotateright(Current->getRight()));
                        return rotateleft(Current);
                }

                return Current;
        }

        //доступ к корневому элементу
        template<class T>
        Node<T>* Tree<T>::getRoot() const { return root; }

        //конструктор дерева: в момент создания дерева ни одного узла нет, корень
↪   смотрит в никуда
        template<class T>
        Tree<T>::Tree(const std::initializer_list<T> elements) : root(nullptr) {
                for (const auto& el : elements) {
                        Add(el);
                }
        }

        template<class T>
        Tree<T>::Tree(const Tree<T>& other) : root(nullptr) {
                for (const auto& el : other) {
                        Add(el.getData());
                }
        }

        template<class T>
        Tree<T>::~Tree() { destroy(root); }

        //функция для добавления числа. Делаем новый узел с этими данными и
↪   вызываем нужную функцию добавления в дерево
        template<class T>
        void Tree<T>::Add(T n)
        {
                Node<T>* N = new Node<T>;
```

```
        N->setData(n);
        Add_R(N, root);
}

template<class T>
void Tree<T>::Remove(T n)
{
        Remove_R(n, root);
}

template<class T>
Node<T>* Tree<T>::Min(Node<T>* Current) const
{
        //минимум - это самый "левый" узел. Идём по дереву всегда влево
        if (root == nullptr) return nullptr;

        if(Current==nullptr)

        Current = root;

        while (Current->getLeft() != nullptr)
                Current = Current->getLeft();

        return Current;
}

template<class T>
Node<T>* Tree<T>::Max(Node<T>* Current) const
{
        //минимум - это самый "правый" узел. Идём по дереву всегда вправо
        if (root == nullptr) return nullptr;

        if (Current == nullptr)
                Current = root;

        while (Current->getRight() != nullptr)
                Current = Current->getRight();

        return Current;
}

//поиск узла в дереве. Второй параметр - в каком поддереве искать, первый
↪  - что искать
template<class T>
Node<T>* Tree<T>::Find(T data, Node<T>* Current) const
{
        //база рекурсии
        if (Current == nullptr) return nullptr;
        if (Current->getData() == data) return Current;
        //рекурсивный вызов
```

```
            if (Current->getData() > data) return Find(data,
↪ Current->getLeft());
            if (Current->getData() < data) return Find(data,
↪ Current->getRight());
            return nullptr;
    }

    //три обхода дерева
    template<class T>
    void Tree<T>::PreOrder(Node<T>* N, std::ostream& stream) const
    {
            if (N != nullptr)
                    stream << N->getData();

            if (N != nullptr && N->getLeft() != nullptr)
                    PreOrder(N->getLeft(), stream);

            if (N != nullptr && N->getRight() != nullptr)
                    PreOrder(N->getRight(), stream);
    }

    //InOrder-обход даст отсортированную последовательность
    template<class T>
    void Tree<T>::InOrder(Node<T>* N, std::ostream& stream) const
    {
            if (N != nullptr && N->getLeft() != nullptr)
                    InOrder(N->getLeft(), stream);

            if (N != nullptr)
                    stream << N->getData();

            if (N != nullptr && N->getRight() != nullptr)
                    InOrder(N->getRight(), stream);
    }

    template<class T>
    void Tree<T>::PostOrder(Node<T>* N, std::ostream& stream) const
    {
            if (N != nullptr && N->getLeft() != nullptr)
                    PostOrder(N->getLeft(), stream);

            if (N != nullptr && N->getRight() != nullptr)
                    PostOrder(N->getRight(), stream);

            if (N != nullptr)
                    stream << N->getData();
    }

    template<class T>
    void Tree<T>::AdvOutput(Node<T>* N, std::ostream& stream) const {
```

```
                if (N != nullptr && N->getLeft() != nullptr)
                        InOrder(N->getLeft(), stream);

                if (N != nullptr)
                        stream << "\nCurrent: " << N << ", left: " << N->getLeft()
↪  << ", right: " << N->getRight() << '\n' << N->getData() << '\n';

                if (N != nullptr && N->getRight() != nullptr)
                        InOrder(N->getRight(), stream);
        }

        template<class T>
        Node<T>* Tree<T>::getByIndex(const size_t index, Node<T>* Current, size_t&
↪  current_index) const
        {
                if (Current->getLeft() != nullptr) {
                        Node<T>* result = getByIndex(index, Current->getLeft(),
↪  current_index);
                        if (result != nullptr) return result;
                }

                if (current_index == index) return Current;
                ++current_index;

                if (Current->getRight() != nullptr) {
                        Node<T>* result = getByIndex(index, Current->getRight(),
↪  current_index);
                        if (result != nullptr) return result;
                }
                return nullptr;
        }

        template<class T>
        Node<T>* Tree<T>::operator[](const size_t index) const {
                size_t start_index=0;
                Node<T>* found = getByIndex(index, root, start_index);
                return found;
        }

        template<class T>
        Tree<T>& Tree<T>::operator=(const Tree<T>& other) {
                if (this == &other)
                return *this;

                Tree<T> temp(other);
                root = temp.root;
                temp.root = nullptr;

            return *this;
        }
```

```
        template<class T>
        iterator<T> Tree<T>::begin() const {
                return iterator<T>(*this, Min());
        }

        template<class T>
        iterator<T> Tree<T>::end() const {
                return iterator<T>(*this, nullptr);
        }
}
```

Приложение B

B.1 Файл Iterator.h

```
namespace AVL_Tree {

    template<class T> class Tree;
    template<class T> class Node;

    template<class T>
    class iterator {
        private:
            const Tree<T>& parent;
            Node<T>* current;

        public:
            iterator(const Tree<T>& parent, Node<T>* current = nullptr);
            iterator(const iterator& it);

            iterator& operator++();

            iterator& operator++(int);

            iterator& operator--();

            iterator& operator--(int);

            bool operator!=(iterator const& other) const;

            bool operator==(iterator const& other) const;

            Node<T>& operator*();
        };

}

#include "Iterator.cpp"
```

B.2 Файл Iterator.cpp

```
namespace AVL_Tree {

    template<typename T>
        iterator<T>::iterator(const Tree<T>& parent, Node<T>* current) :
↪   parent(parent), current(current) {}

    template<typename T>
        iterator<T>::iterator(const iterator& it) : parent(it.parent),
↪   current(it.current) {}

    template<typename T>
        iterator<T>& iterator<T>::operator++()
        {
                if (current == nullptr || parent.getRoot() == nullptr) return
↪   *this;

                if (current->getRight() != nullptr) {
                current = parent.Min(current->getRight());
                return *this;
                }

                Node<T>* Current = current;
                while(true) {
                if (Current->getParent() == nullptr) {
                        current = nullptr;
                        return *this;
                }

                if (Current->getParent()->getLeft() == Current){
                        current = Current->getParent();
                        return *this;
                }

                Current = Current->getParent();
                }

                current = nullptr;
                return *this;
        }

    template<typename T>
        iterator<T>& iterator<T>::operator++(int)
        {
                iterator notModified = *this;
                ++(*this);
                return notModified;
        }

    template<typename T>
```

```
    iterator<T>& iterator<T>::operator--()
    {
            if (current == nullptr || parent.getRoot() == nullptr) return
↪ *this;

            if (current->getLeft() != nullptr) {
            current = parent.Max(current->getLeft());
            return *this;
            }

            Node<T>* Current = current;
            while(true) {
            if (Current->getParent() == nullptr) {
                    current = nullptr;
                    return *this;
            }

            if (Current->getParent()->getRight() == Current){
                    current = Current->getParent();
                    return *this;
            }

            Current = Current->getParent();
            }

            current = nullptr;
            return *this;
    }

template<typename T>
    iterator<T>& iterator<T>::operator--(int)
    {
            iterator notModified = *this;
            --(*this);
            return notModified;
    }

template<typename T>
    bool iterator<T>::operator!=(iterator const& other) const { return
↪ (current != other.current); }

template<typename T>
    bool iterator<T>::operator==(iterator const& other) const { return
↪ (current == other.current); }

template<typename T>
    Node<T>& iterator<T>::operator*()
    {
            return *current;
    }
```

```
}
```