МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: АВЛ-деревья

Студент гр. 9304 _____ Шуняев А.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Шуняев А.В.

Группа 9304

Тема работы: АВЛ-деревья

Исходные данные:

АВЛ-дерево, язык программирования C++, сборка программы под Windows.

Содержание пояснительной записки:

«Содержание», «Задание», «Основные теоретические положения», «Описание структур данных и используемых функций», «Описание интерфейса пользователя», «Тестирование», «Заключение»

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 23.11.2020

Дата сдачи реферата: 21.12.2020

Дата защиты реферата: 21.12.2020

Студент _____ Шуняев А.В.

Преподаватель _____ Филатов А.Ю.

АННОТАЦИЯ

В данной работе была реализована программа, выполняющая построение АВЛ-дерева, а также вставку новых элементов и удаление старых по ключу. Вставка и удаление сопровождаются пояснениями и демонстрацией дерева.

SUMMARY

In this work, a program was implemented that builds an AVL tree, as well as inserting new elements and deleting old ones by key. Insertion and deletion are accompanied by an explanation and demonstration of the tree.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы

1. Реализация программы для демонстрации алгоритма вставки и удаления элементов из АВЛ-дерева, чтобы на каждом шаге пользователю давалось пояснение, что происходит в данный в момент и какой алгоритм используется.
2. Визуализация структур данных.

Задание

Вариант 15. Демонстрация. АВЛ-дерево:

По заданной последовательности элементов Elem построить структуру данных определённого типа – АВЛ-дерево;

Предусмотреть возможность повторного выполнения с другим элементом.

Пояснение задания

На вход программы подается последовательность чисел, разделенная пробелами. Требуется: построить АВЛ-дерево, продемонстрировать вставку и удаление элементов.

1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

1.1. АВЛ-дерево

АВЛ-дерево — сбалансированное двоичное дерево поиска, в котором поддерживается следующие свойства:

- Для каждой его вершины высота её двух поддеревьев различается не более чем на 1.
- Для каждого узла верно, что ключ левого потомка меньше, чем ключ данного узла, а ключ правого потомка – больше.

1.2. Балансировка дерева

Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев $|h(L)-h(R)|=2$, изменяет связи предок-потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева $|h(L)-h(R)|\leqslant 1$, иначе ничего не меняет.

Выражение $|h(L) - h(R)| \leq 1$ называется баланс фактор. Для его вычисления в узлах хранится их высота.

Для балансировки вершины используются повороты влево и вправо, а также большие повороты влево и вправо. Повороты представлены на рисунке 1.

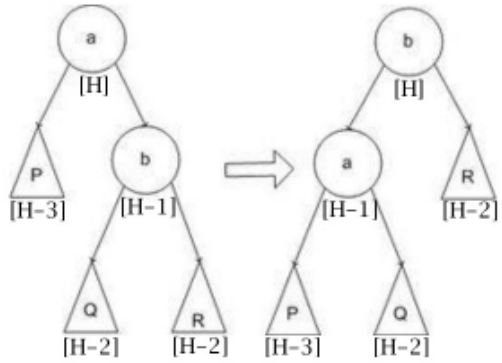
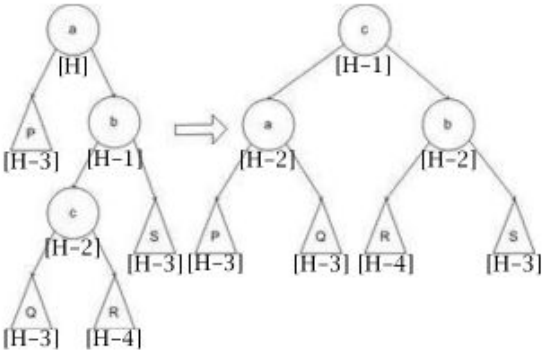
Тип вращения	Иллюстрация
Малое левое вращение	
Большое левое вращение	

Рисунок 1 – Повороты дерева

1.3. Вставка элементов

Пусть нам надо добавить ключ t. Будем спускаться по дереву, как при поиске ключа t. Если мы стоим в вершине a и нам надо идти в поддерево, которого нет, то делаем ключ t листом, а вершину a его корнем. Дальше поднимаемся вверх по пути поиска и пересчитываем баланс у вершин.

Если после добавления дерево разбалансировалось, требуется выполнить балансировку.

1.4. Удаление элементов

Если вершина — лист, то удалим её, иначе найдём самую близкую по значению вершину a, переместим её на место удаляемой вершины и

удалим вершину a. Далее поднимаемся вверх по пути поиска и пересчитываем высоту.

Если после добавления дерево разбалансировалось, требуется выполнить балансировку.

2. ОПИСАНИЕ СТРУКТУР ДАННЫХ И ИСПОЛЬЗУЕМЫХ ФУНКЦИЙ

- Класс Node – Реализация элемента списка
- Поле key_ класса Node – хранит ключ элемента
- Поле height_ класса Node – хранит высоту дерева. Требуется для вычисления баланс фактора. Он нужен для балансировки дерева. Вычисляется путем вычитания высоты левого поддерева из правого.
- Поле counter_ класса Node – хранит количество элементов с заданным ключом
- Поля left_ и right_ класса Node – хранят ссылки на левое и правое поддеревья
- Класс AVLTree – реализация обертки над списком для более удобного использования
- Метод Insert – добавляет элемент в дерево
- Метод Remove – удаляет элемент из дерева
- Метод FindMin – находит наименьший элемент в поддереве (вспомогательная ф-ция для удаления)
- Метод RemoveMin – удаляется наименьший элемент из поддерева (вспомогательная функция для удаления)
- Метод Find – находит элемент по ключу
- Метод NodeBalansing – производит балансировку дерева
- Метод RotateLeft – поворачивает дерево влево (нужна в балансировке)
- Метод RotateRight – поворачивает дерево вправо (нужна в балансировке)
- Метод AdaptHeight – устанавливает правильное значение высоты в элементе
- Метод BalanceFactor – находит баланс фактор
- Метод FindAll – находит количество элементов с заданным ключем
- Метод PrintTree – отрисовывает дерево в консоль

- Метод Demonstration – запускает демонстрацию
- Enum class DemoState – хранит перечисление состояний демонстрации

3. Описание интерфейса пользователя

Сначала программа запрашивает у пользователя последовательность чисел, разделенных пробелами. После этого строиться дерево. Далее спрашивается, что нужно продемонстрировать (вставку или удаление). Затем, в зависимости от выбора, программа просит ввести ключ, который надо вставить или удалить.

4. ТЕСТИРОВАНИЕ

4.1. Вставка элемента. Рисунки 2 и 3.

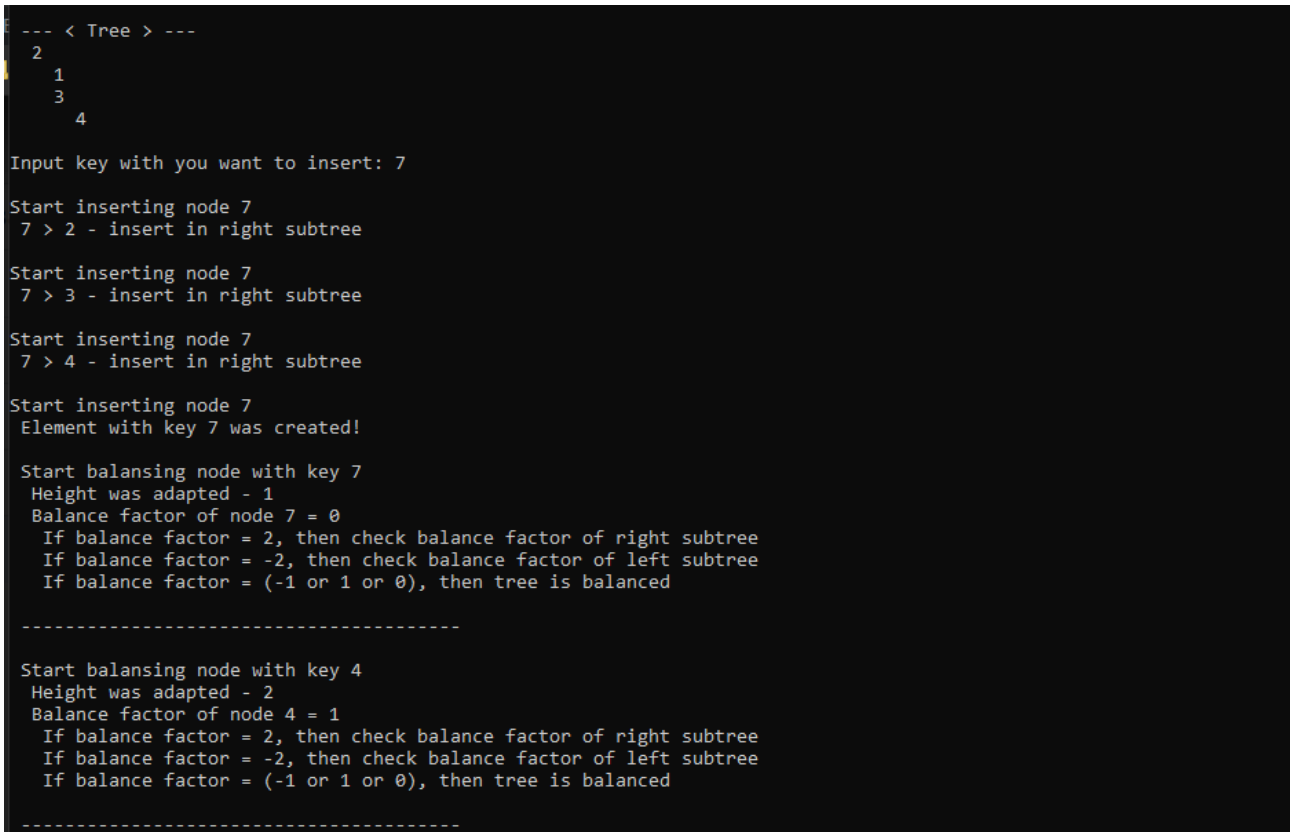


Рисунок 2 – Вставка элемента

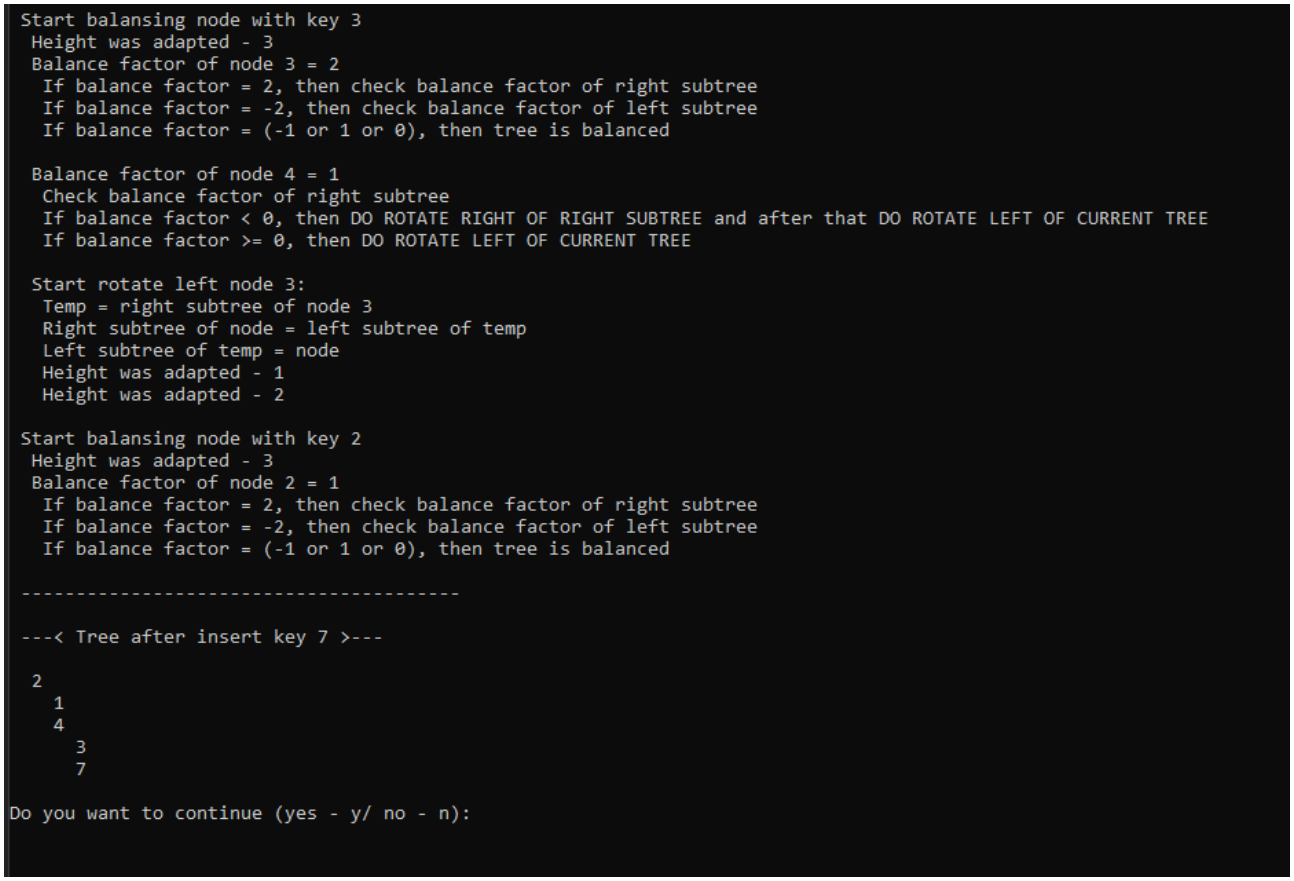


Рисунок 3 – Вставка элемента

4.2. Удаление элемента. Рисунок 4

```
--- < Tree > ---
 4
   2
     1
     3
   6
     5
     7

Input key with you want to remove: 6

 6 > 4 - search in right subtree

Element with key 6 was found!
 Counter Decreased!
 If counter = 0, then node will be delete
 If counter > 0, then it mean that there are more then one element with key 6
 Counter = 0

 Temp_left = left subtree of node
 Temp_right = right subtree of node
 Node is removed!

 Temp_right isn't empty, then find min key in right subtree and save it in variable "min":
  Finding min in right subtree...
   If node has left subtree, then find min in left subtree
   Else return node

    Node with key 7 does not have left subtree

  Right subtree of min = temp_right without min element
  Left subtree of min = temp_left

  Removing min in right subtree...
    Left subtree of node with key 7 is empty, so right subtree of this node puts on node's place

 Start balansing node with key 7
  Height was adapted - 2
  Balance factor of node 7 = -1
   If balance factor = 2, then check balance factor of right subtree
   If balance factor = -2, then check balance factor of left subtree
   If balance factor = (-1 or 1 or 0), then tree is balanced

 ---------------------------------------

 Start balansing node with key 4
  Height was adapted - 3
  Balance factor of node 4 = 0
   If balance factor = 2, then check balance factor of right subtree
   If balance factor = -2, then check balance factor of left subtree
   If balance factor = (-1 or 1 or 0), then tree is balanced

 ---------------------------------------

 ---< Tree after remove key 6 >---

  4
    2
      1
      3
    7
      5

Do you want to continue (yes - y/ no - n):
```

Рисунок 4 – Удаление элемента

ЗАКЛЮЧЕНИЕ

Были изучена структура АВЛ-дерево, функции для работы с ней: вставка и удаление элементов по ключу, балансировка. Решена задача реализации АВЛ-дерева на C++, а также визуализации структур данных.

Было проведено тестирование полученной программы. Исходный код представлен в Приложении А.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

Файл main.cpp:

```
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#include "AVLTree.h"
#include "Node.h"


int main() {

	AVLTree tree;

	int input = 0;
	char c = 'y';
	char state_input;

	do {
		std::cout << "\nWhat do you want to demonstrate? (Insert - I / Remove - R): ";
		std::cin >> state_input;

		system("cls");
		std::cout << "\n --- < Tree > ---\n";
		tree.PrintTree(tree.Front());

		if (state_input == 'I') {
			tree.Demonstration(DemoState::InsertDemo);
		}
		else if (state_input == 'R') {
			tree.Demonstration(DemoState::RemoveDemo);
		}
		else {
			std::cout << "\nWrong input!";
```

```
            }

            std::cout << "\nDo you want to continue (yes - y/ no - n): ";
            std::cin >> c;
            std::cout << std::endl;

        } while (c == 'y');

        tree.~AVLTree();
        _CrtDumpMemoryLeaks();
        return 0;
}
```

Файл AVLTree.cpp:

```
#include "AVLTree.h"
#include "Node.h"

AVLTree::AVLTree()
{
        std::string str;

        std::cout << "Enter set of digits: ";
        std::getline(std::cin, str);
        std::istringstream iss(str);

        int digit;
        while (iss >> digit) {
                this->Insert(digit);
        }
}

AVLTree::~AVLTree()
{
}
```

```
// Demonstation
void AVLTree::PrintTree(std::shared_ptr<Node> node, int tab)
{
	int temp = tab;
	std::string str = "";
	while (temp != 0) {
		str += "  ";
		temp--;
	}

	if(node != nullptr) {
		std::cout << str << node->key_ << '\n';
		if (node->left_ != nullptr) {
			this->PrintTree(node->left_, tab + 1);
		}
		if (node->right_ != nullptr) {
			this->PrintTree(node->right_, tab + 1);
		}
	}
	else {
		std::cout << "Tree is empty!\n";
	}
}

void AVLTree::Demonstration(DemoState state)
{
	int input = 0;
	if (state == DemoState::InsertDemo) {
		std::cout << "\nInput key with you want to insert: ";
		std::cin >> input;

		this->Insert(input, DemoState::InsertDemo);

		std::cout << "\n ---< Tree after insert key " << input << " >--- \n\n";
```

```
            this->PrintTree(this->Front());
        }
        else if (state == DemoState::RemoveDemo) {
            std::cout << "\nInput key with you want to remove: ";
            std::cin >> input;

            this->Remove(input, DemoState::RemoveDemo);

            std::cout << "\n ---< Tree after remove key " << input << " >--- \n\n";
            this->PrintTree(this->Front());
        }
}

//////////////////////////////////////////////////////////////////////

std::shared_ptr<Node> AVLTree::Front()
{
    return this->head_;
}

std::shared_ptr<Node> AVLTree::Find(int key, DemoState state)
{
    return Node::Find(key, this->head_);
}

void AVLTree::Remove(int key, DemoState state)
{
    this->head_ = Node::Remove(key, this->head_, state);
}

void AVLTree::Insert(int key, DemoState state)
{
    this->head_ = Node::Insert(key, this->head_, state);
}
```

Файл AVLTree.h:

```
#pragma once
#include "Includes.h"

class Node;

class AVLTree
{
public:
	AVLTree();
	~AVLTree();

	void PrintTree(std::shared_ptr<Node> node, int tab = 1);
	void Demonstration(DemoState state);

	std::shared_ptr<Node> Front();
	std::shared_ptr<Node> Find(int key, DemoState state = DemoState::NoDemo);

	void Remove(int key, DemoState state = DemoState::NoDemo);
	void Insert(int key, DemoState state = DemoState::NoDemo);

private:
	std::shared_ptr<Node> head_ = nullptr;
};
```

Файл Node.cpp:

```
#include "Node.h"

//Методы класса Node
Node::Node(int key, std::shared_ptr<Node> left, std::shared_ptr<Node> right)
{
	this->key_ = key;
	this->height_ = 1;
	this->counter_ = 1;
	this->left_ = left;
	this->right_ = right;
```

```
}

unsigned char Node::GetHeight()
{
	return this ? this->height_ : 0;
}

int Node::GetKey()
{
	return this->key_;
}

int Node::GetCounter()
{
	return this->counter_;
}

int Node::BalanceFactor(DemoState state)
{
	int temp = (this->right_->GetHeight()) - (this->left_->GetHeight());

	if (state != DemoState::NoDemo) {
		std::cout << "\n  Balance factor of node " << this->key_ << " = " << temp <<
'\n';
	}

	return temp;
}

void Node::AdaptHeight(DemoState state)
{
	unsigned char left_height = this->left_->GetHeight();
	unsigned char right_height = this->right_->GetHeight();

	this->height_ = (left_height > right_height ? left_height : right_height) + 1;
```

```
		if (state != DemoState::NoDemo) {
			std::cout << "  Height was adapted - " << (int) this->height_;
		}
	}

	//////////////////////////////////////////////////////

	//Удаление
	std::shared_ptr<Node> Node::Remove(int key, std::shared_ptr<Node> head, DemoState state)
	{
		if (state == DemoState::InsertDemo) {
			std::cout << "\nStart removing node " << key;
		}

		if (head == nullptr) {
			if (state == DemoState::RemoveDemo) {
				std::cout << "\nThere is no such key in the tree!\n";
			}

			return nullptr;
		}

		if (key < head->key_) {
			if (state == DemoState::RemoveDemo) {
				std::cout << "\n " << key << " < " << head->key_ << " - search in left subtree\n";
			}

			head->left_ = Node::Remove(key, head->left_, state);
		}
		else if (key > head->key_) {
			if (state == DemoState::RemoveDemo) {
```

```
                std::cout << "\n " << key << " > " << head->key_ << " - search in right
subtree\n";
            }

            head->right_ = Node::Remove(key, head->right_, state);
        }
        else {
            if (state == DemoState::RemoveDemo) {
                std::cout << "\nElement with key " << key << " was found!\n";
                std::cout << " Counter Decreased!\n";
                std::cout << " If counter = 0, then node will be delete\n";
                std::cout << " If counter > 0, then it mean that there are more then one
element with key " << key << "\n";
            }

            head->counter_--;

            if (state == DemoState::RemoveDemo) {
                std::cout << " Counter = " << head->counter_ << "\n";
            }

            if (head->counter_ == 0) {

                if (state == DemoState::RemoveDemo) {
                    std::cout << "\n Temp_left = left subtree of node";
                    std::cout << "\n Temp_right = right subtree of node";
                    std::cout << "\n Node is removed!";
                }

                std::shared_ptr<Node> temp_left = head->left_;
                std::shared_ptr<Node> temp_right = head->right_;

                head.reset();

                if (temp_right == nullptr) {
```

```
                        if (state == DemoState::RemoveDemo) {
                                std::cout << "\n\n Temp_right is empty, so temp_left puts
on the node's place\n";
                        }

                        return temp_left;
                }
                else {
                        if (state == DemoState::RemoveDemo) {
                                std::cout << "\n\n Temp_right isn't empty, then find min
key in right subtree and save it in variable \"min\": ";
                                std::cout << "\n  Finding min in right subtree...";
                                std::cout << "\n   If node has left subtree, then find min
in left subtree";
                                std::cout << "\n   Else return node\n";
                        }
                        std::shared_ptr<Node>   min   =   Node::FindMin(temp_right,
state);

                        if (state == DemoState::RemoveDemo) {
                                std::cout << "\n   Right subtree of min = temp_right
without min element";
                                std::cout << "\n  Left subtree of min = temp_left\n";
                        }
                        min->right_ = Node::RemoveMin(temp_right, state);
                        min->left_ = temp_left;

                        return Node::NodeBalancing(min, state);
                }
        }
    }

    return Node::NodeBalancing(head, state);
}
```

```
std::shared_ptr<Node> Node::FindMin(std::shared_ptr<Node> node, DemoState state)
{
	if (state == DemoState::RemoveDemo) {
		if (node->left_ != nullptr) {
			std::cout << "\n    Node with key " << node->key_ <<" has left subtree";
		}
		else {
			std::cout << "\n    Node with key " << node->key_ << " does not have left subtree\n";
		}
	}
	return node->left_ ? Node::FindMin(node->left_, state) : node;
}

std::shared_ptr<Node> Node::RemoveMin(std::shared_ptr<Node> node, DemoState state)
{
	if (state == DemoState::RemoveDemo) {
		std::cout << "\n  Removing min in right subtree...";
		if (node->left_ == nullptr) {
			std::cout << "\n    Left subtree of node with key " << node->key_ << " is empty, so right subtree of this node puts on node's place\n";
		}
		else {
			std::cout << "\n    Left subtree of node with key " << node->key_ << " isn't empty, so removing min in left subtree";
		}
	}

	if (node->left_ == nullptr) {
		return node->right_;
	}
	node->left_ = Node::RemoveMin(node->left_, state);
	return Node::NodeBalancing(node, state);
}
```

```
//Вставка
std::shared_ptr<Node> Node::Insert(int key, std::shared_ptr<Node> head, DemoState state)
{
	if (state == DemoState::InsertDemo) {
		std::cout << "\nStart inserting node " << key;
	}

	if (head == nullptr) {
		if (state == DemoState::InsertDemo) {
			std::cout << "\n Element with key "<< key << " was created!\n";
		}
		return Node::NodeBalancing(std::make_shared<Node>(key), state);
	}
	else if (key < head->key_) {

		if (state == DemoState::InsertDemo) {
			std::cout << "\n " << key << " < " << head->key_ << " - insert in left
subtree\n";
		}

		head->left_ = Node::Insert(key, head->left_, state);
	}
	else if (key > head->key_) {
		if (state == DemoState::InsertDemo) {
			std::cout << "\n " << key << " > " << head->key_ << " - insert in right
subtree\n";
		}

		head->right_ = Node::Insert(key, head->right_, state);
	}
	else {
		if (state == DemoState::InsertDemo) {
			std::cout << "\n Element with key " << key << " has already exist! So
counter was increased\n";
		}
```

```
            head->counter_++;
        }

        return Node::NodeBalancing(head, state);
}

//Поиск
std::shared_ptr<Node> Node::Find(int key, std::shared_ptr<Node> head, DemoState state)
{
        if (head == nullptr) {
            return nullptr;
        }
        else if (key < head->key_) {
            return Node::Find(key, head->left_, state);
        }
        else if (key > head->key_){
            return Node::Find(key, head->right_, state);
        }
        else {
            return head;
        }
}


//////////////////////////////////////////////////////////

//Вспомогательные методы

std::shared_ptr<Node> Node::RotateRight(std::shared_ptr<Node> node, DemoState state)
{
        if (state != DemoState::NoDemo) {
            std::cout << "\n  Start rotate right node " << node->key_<< ":";
            std::cout << "\n   Temp = left subtree of node " << node->key_;
            std::cout << "\n   Left subtree of node = right subtree of temp";
            std::cout << "\n   Right subtree of temp = node";
```

```
        }

        std::shared_ptr<Node> temp = node->left_;
        node->left_ = temp->right_;
        temp->right_ = node;

        if (state != DemoState::NoDemo) {
                std::cout << "\n ";
        }
        node->AdaptHeight(state);

        if (state != DemoState::NoDemo) {
                std::cout << "\n ";
        }
        temp->AdaptHeight(state);

        if (state != DemoState::NoDemo) {
                std::cout << "\n";
        }

        return temp;
}

std::shared_ptr<Node> Node::RotateLeft(std::shared_ptr<Node> node, DemoState state)
{
        if (state != DemoState::NoDemo) {
                std::cout << "\n  Start rotate left node " << node->key_ << ":";
                std::cout << "\n   Temp = right subtree of node " << node->key_;
                std::cout << "\n   Right subtree of node = left subtree of temp";
                std::cout << "\n   Left subtree of temp = node";
        }

        std::shared_ptr<Node> temp = node->right_;
        node->right_ = temp->left_;
        temp->left_ = node;
```

```
            if (state != DemoState::NoDemo) {
                std::cout << "\n ";
            }
            node->AdaptHeight(state);

            if (state != DemoState::NoDemo) {
                std::cout << "\n ";
            }
            temp->AdaptHeight(state);

            if (state != DemoState::NoDemo) {
                std::cout << "\n";
            }

            return temp;
        }

        std::shared_ptr<Node> Node::NodeBalancing(std::shared_ptr<Node> node, DemoState state)
        {
            if (state != DemoState::NoDemo) {

                std::cout << "\n Start balansing node with key " << node->key_ << "\n";
            }

            node->AdaptHeight(state);

            int b_factor = node->BalanceFactor(state);

            if (state != DemoState::NoDemo) {
                std::cout << "  If balance factor = 2, then check balance factor of right subtree";
                std::cout << "\n   If balance factor = -2, then check balance factor of left subtree";
                std::cout << "\n  If balance factor = (-1 or 1 or 0), then tree is balanced\n";
```

```
        }

        if (b_factor == 2)
        {
            int b_factor_right_tree = node->right_->BalanceFactor(state);

            if (state != DemoState::NoDemo) {
                std::cout << "   Check balance factor of right subtree";
                std::cout << "\n   If balance factor < 0, then DO ROTATE RIGHT OF RIGHT SUBTREE and after that DO ROTATE LEFT OF CURRENT TREE ";
                std::cout << "\n   If balance factor >= 0, then DO ROTATE LEFT OF CURRENT TREE \n";
            }

            if (b_factor_right_tree < 0) {
                node->right_ = Node::RotateRight(node->right_, state);
            }
            return Node::RotateLeft(node, state);
        }
        else if (b_factor == -2)
        {
            int b_factor_left_tree = node->left_->BalanceFactor(state);

            if (state != DemoState::NoDemo) {
                std::cout << "   Check balance factor of right subtree";
                std::cout << "\n   If balance factor > 0, then DO ROTATE LEFT OF LEFT SUBTREE and, after that, DO ROTATE RIGHT OF CURRENT TREE ";
                std::cout << "\n   If balance factor <= 0, then DO ROTATE RIGHT OF CURRENT TREE \n";
            }

            if (b_factor_left_tree > 0) {
                node->left_ = Node::RotateLeft(node->left_, state);
            }
            return Node::RotateRight(node, state);
```

```
		}

		if (state != DemoState::NoDemo) {
			std::cout << "\n ---------------------------------------- \n";
		}

		return node;
	}
```

Файл Node.h:

```
	#pragma once
	#include "Includes.h"


	class Node
	{
		friend class AVLTree;

	public:
		Node(int key, std::shared_ptr<Node> left = nullptr, std::shared_ptr<Node> right = nullptr);

		unsigned char GetHeight();
		int GetKey();
		int GetCounter();
		int BalanceFactor(DemoState state = DemoState::NoDemo);
		void AdaptHeight(DemoState state = DemoState::NoDemo);

		static std::shared_ptr<Node> NodeBalancing(std::shared_ptr<Node> node, DemoState state = DemoState::NoDemo);
		static std::shared_ptr<Node> Remove(int key, std::shared_ptr<Node> head, DemoState state = DemoState::NoDemo);
		static std::shared_ptr<Node> Insert(int key, std::shared_ptr<Node> head, DemoState state = DemoState::NoDemo);
		static std::shared_ptr<Node> Find(int key, std::shared_ptr<Node> head, DemoState state = DemoState::NoDemo);
```

```
private:
        unsigned char height_;
        int key_;
        int counter_;
        std::shared_ptr<Node> left_;
        std::shared_ptr<Node> right_;

        static std::shared_ptr<Node> RotateRight(std::shared_ptr<Node> node, DemoState state = DemoState::NoDemo);
        static std::shared_ptr<Node> RotateLeft(std::shared_ptr<Node> node, DemoState state = DemoState::NoDemo);
        static std::shared_ptr<Node> FindMin(std::shared_ptr<Node> node, DemoState state = DemoState::NoDemo);
        static std::shared_ptr<Node> RemoveMin(std::shared_ptr<Node> node, DemoState state = DemoState::NoDemo);
};
```

Файл Includes.h:

```
#pragma once
#include <memory>
#include <string>
#include <sstream>
#include <iostream>

enum class DemoState
{
        InsertDemo,
        RemoveDemo,
        NoDemo
};
```