Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

Специальность «Программное обеспечение информационных технологий»

Контрольная работа № 1

«БИНАРНые ДЕРЕВья»

По курсу: Алгоритмы и структуры данных

Выполнил: студент гр. 281073

Буйновский М.В.

Проверил: Осмоловский А.А.

Минск 2023

**Введение**

В данной контрольной работе рассматривается тема бинарных деревьев поиска и их применение в программировании. Бинарные деревья представляют собой структуры данных, где каждая вершина может иметь не более двух потомков - левого и правого ветвей. Особый подкласс бинарных деревьев - деревья поиска, которые характеризуются условием, что значение информационного поля вершины больше значения из левого поддерева и меньше значения из правого поддерева.

Контрольная работа состоит из двух частей. В первой части требуется ввести от 10 до 15 целых чисел и построить из них бинарное дерево поиска с использованием указателей. Затем необходимо выполнить обходы этого дерева прямым, симметричным и обратным способами. Также требуется реализовать процедуры поиска, вставки и удаления элементов в бинарном дереве поиска.

Во второй части работы рассматривается построение, поиск и обход дерева без использования рекурсивных алгоритмов. Для этого используется метод, называющийся «прошивка дерева». Использование прошитых деревьев имеет свои недостатки, такие как более сложное включение новых вершин и необходимость поддерживать два типа связей. Поэтому прошитые деревья рекомендуется использовать в задачах, где изменения в дереве происходят редко, а обходы выполняются часто.

Кроме того, в работе упоминается АВЛ-дерево, которое является развитием бинарного дерева поиска. Оно отличается от обычного дерева поиска тем, что поддерживает сбалансированность дерева при включении и удалении элементов. Для этого каждый узел дерева содержит вспомогательное поле, которое указывает на равновесие поддеревьев данного узла.

**Задание 1**

Ввести 10-15 целых чисел и построить из них с помощью указателей бинарное дерево поиска. Обойти его прямым, симметричным и обратным способами. Реализовать процедуры поиска, вставки и удаления элементов бинарного дерева поиска.

**Код программы:**

**colors.h**

#pragma once

#include <string>

const std::string COLOR\_RESET = "\033[0m";

const std::string COLOR\_RED = "\033[31m";

const std::string COLOR\_GREEN = "\033[32m";

const std::string COLOR\_YELLOW = "\033[33m";

const std::string COLOR\_CYAN = "\033[36m";

**bst.h**

#pragma once

#include <iostream>

#include <string>

#include "colors.h"

template <typename T> class BST {

public:

struct Node {

T value;

// Parent only used to beautifully display nodes, it is not used in anything else

Node \*left, \*right, \*parent;

Node(T value, Node \*parent) : value(value), left(nullptr), right(nullptr), parent(parent) {

}

};

private:

BST<T>::Node \*tree = nullptr;

// Colored "left" and "right" words

const std::string left\_string = COLOR\_YELLOW + "left" + COLOR\_RESET;

const std::string right\_string = COLOR\_CYAN + "right" + COLOR\_RESET;

const std::string moving\_left\_string = " - moving " + this->left\_string;

const std::string moving\_right\_string = " - moving " + this->right\_string;

// Colored "value" output

std::string valueString(T value) {

return "'" + COLOR\_YELLOW + std::to\_string(value) + COLOR\_RESET + "'";

}

// Shows colored node. Green - parent, Yellow - left of parent, Cyan - right of parent

void showNode(BST<T>::Node \*node) {

if (!node->parent) {

std::cout << COLOR\_GREEN;

} else if (node->parent->left == node) {

std::cout << COLOR\_YELLOW;

} else {

std::cout << COLOR\_CYAN;

}

std::cout << node->value << COLOR\_RESET << " ";

}

void showPreOrder(BST<T>::Node \*node) {

if (!node) {

return;

}

showNode(node);

showPreOrder(node->left);

showPreOrder(node->right);

}

void showInOrder(BST<T>::Node \*node) {

if (!node) {

return;

}

showInOrder(node->left);

showNode(node);

showInOrder(node->right);

}

void showPostOrder(BST<T>::Node \*node) {

if (!node) {

return;

}

showPostOrder(node->left);

showPostOrder(node->right);

showNode(node);

}

public:

void insert(T value) {

if (!this->tree) {

std::cout << COLOR\_GREEN << "Tree is empty, creating the root" << COLOR\_RESET << std::endl;

this->tree = new BST<T>::Node(value, nullptr);

return;

}

std::cout << "Searching for the suitable space:" << std::endl;

BST<T>::Node \*prev = nullptr;

BST<T>::Node \*current = tree;

while (current) {

prev = current;

if (current->value > value) {

std::cout << moving\_left\_string << std::endl;

current = current->left;

} else if (current->value < value) {

std::cout << moving\_right\_string << std::endl;

current = current->right;

} else {

std::cout << COLOR\_RED << "The same value is found. Returning" << COLOR\_RESET << std::endl;

return;

}

}

if (prev->value > value) {

std::cout << "Inserting new node " << this->left\_string << " with the " << valueString(value) << std::endl;

prev->left = new BST<T>::Node(value, prev);

} else if (prev->value < value) {

std::cout << "Inserting new node " << this->right\_string << " with the " << valueString(value) << std::endl;

prev->right = new BST<T>::Node(value, prev);

}

}

BST<T>::Node \*search(T value) {

if (!this->tree) {

std::cout << "The tree is empty, nothing to search" << std::endl;

return nullptr;

}

BST<T>::Node \*current = this->tree;

while (current) {

if (current->value > value) {

std::cout << moving\_left\_string << std::endl;

current = current->left;

} else if (current->value < value) {

std::cout << moving\_right\_string << std::endl;

current = current->right;

} else {

std::cout << "The value " << this->valueString(value) << COLOR\_GREEN << " was found" << COLOR\_RESET

<< std::endl;

return current;

}

}

std::cout << "The value " << this->valueString(value) << COLOR\_RED << " was not found" << COLOR\_RESET

<< std::endl;

return nullptr;

}

void show() {

if (!this->tree) {

std::cout << "The tree is empty, nothing to show" << std::endl;

return;

}

std::cout << "Pre-order traversal: ";

this->showPreOrder(this->tree);

std::cout << std::endl;

std::cout << "In-order traversal: ";

this->showInOrder(this->tree);

std::cout << std::endl;

std::cout << "Post-order traversal: ";

this->showPostOrder(this->tree);

std::cout << std::endl;

}

BST<T>::Node \*remove(T value) {

if (!this->tree) {

std::cout << "The tree is empty, nothing to remove" << std::endl;

return this->tree;

}

std::cout << "Searching for the value " << this->valueString(value) << std::endl;

BST<T>::Node \*current = this->tree;

BST<T>::Node \*previous = nullptr;

while (current) {

if (current->value > value) {

previous = current;

std::cout << moving\_left\_string << std::endl;

current = current->left;

} else if (current->value < value) {

previous = current;

std::cout << moving\_right\_string << std::endl;

current = current->right;

} else {

std::cout << "The value " << this->valueString(value) << COLOR\_GREEN << " was found" << COLOR\_RESET

<< std::endl;

break;

}

}

if (!current) {

std::cout << "The value " << this->valueString(value) << COLOR\_RED << " was not found" << COLOR\_RESET

<< std::endl;

return nullptr;

}

if (!current->left || !current->right) {

std::cout << COLOR\_YELLOW << "Value has only one child" << COLOR\_RESET << std::endl;

BST<T>::Node \*new\_current;

if (!current->left) {

new\_current = current->right;

} else {

new\_current = current->left;

}

if (!previous) {

std::cout << "The value is the root, replacing the root" << std::endl;

this->tree = new\_current;

return new\_current;

}

if (current == previous->left) {

std::cout << "Placing " << left\_string << " child on the place of the current" << std::endl;

previous->left = new\_current;

} else {

std::cout << "Placing " << right\_string << " child on the place of the current" << std::endl;

previous->right = new\_current;

}

free(current);

} else {

std::cout << COLOR\_YELLOW << "Value has 2 children" << COLOR\_RESET << std::endl;

previous = nullptr;

BST<T>::Node \*to\_remove = nullptr;

std::cout << "Finding leftmost value of a right child" << std::endl;

to\_remove = current->right;

while (to\_remove->left) {

std::cout << moving\_left\_string << std::endl;

previous = to\_remove;

to\_remove = to\_remove->left;

}

std::cout << "Removing the value" << std::endl;

if (previous) {

previous->left = to\_remove->right;

} else {

current->right = to\_remove->right;

}

current->value = to\_remove->value;

free(to\_remove);

}

return this->tree;

}

};

**main\_bst.cpp**

#include <cstdlib>

#include <iostream>

#include <stdlib.h>

#include "bst.h"

#include "colors.h"

int main(void) {

BST<int> tree;

while (true) {

std::cout << "Choose one of the options: " << std::endl

<< COLOR\_CYAN << " 1. " << COLOR\_RESET << "Insert new value." << std::endl

<< COLOR\_CYAN << " 2. " << COLOR\_RESET << "Remove value from the tree." << std::endl

<< COLOR\_CYAN << " 3. " << COLOR\_RESET << "Search a value in the tree." << std::endl

<< COLOR\_CYAN << " 4. " << COLOR\_RESET << "Show the binary tree using 3 methods." << std::endl

<< COLOR\_CYAN << " 5. " << COLOR\_RESET << "Exit the program." << std::endl;

int choice;

std::cout << "Choice: " << COLOR\_YELLOW;

std::cin >> choice;

std::cout << COLOR\_RESET << std::endl;

switch (choice) {

int value;

case 1:

int size;

std::cout << "Number of values to insert: " << COLOR\_YELLOW;

std::cin >> size;

std::cout << COLOR\_RESET << std::endl;

for (int i = 0; i < size; i++) {

std::cout << "Value to insert: " << COLOR\_YELLOW;

std::cin >> value;

std::cout << COLOR\_RESET;

tree.insert(value);

std::cout << std::endl;

}

break;

case 2:

std::cout << "Value to remove: " << COLOR\_YELLOW;

std::cin >> value;

std::cout << COLOR\_RESET;

tree.remove(value);

break;

case 3:

std::cout << "Value to search: " << COLOR\_YELLOW;

std::cin >> value;

std::cout << COLOR\_RESET;

tree.search(value);

break;

case 4:

tree.show();

break;

case 5:

std::cout << "Exiting the program." << std::endl;

return EXIT\_SUCCESS;

default:

std::cout << "Unknown option: " << COLOR\_RED << choice << COLOR\_RESET << "." << std::endl;

break;

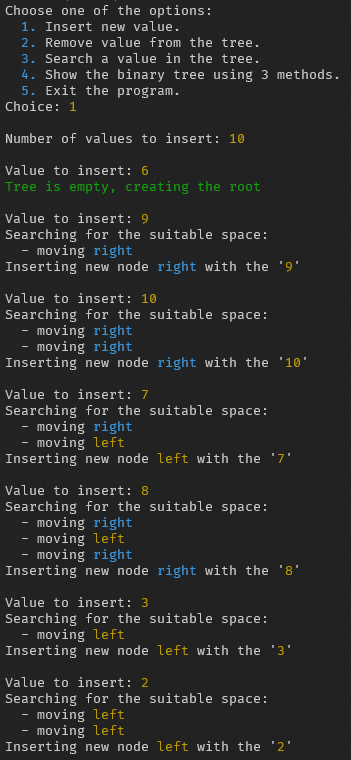
}

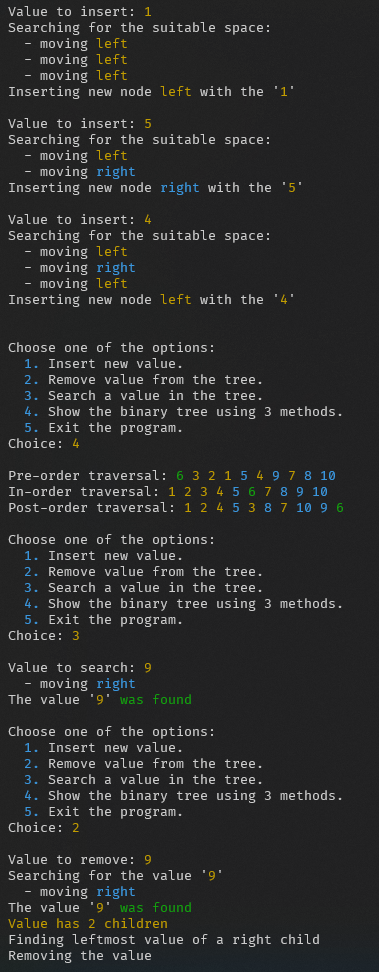
std::cout << std::endl;

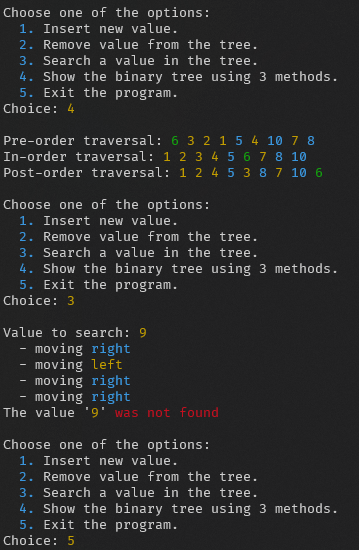
}

}

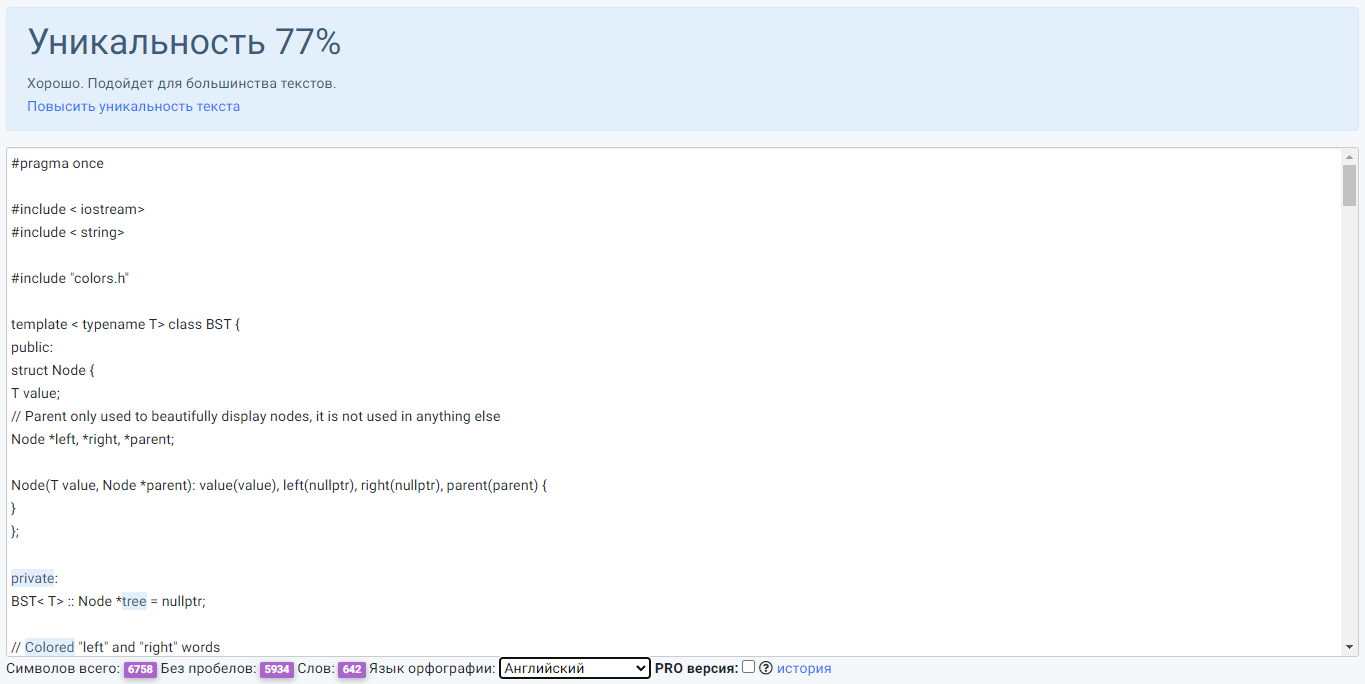
**Скриншоты работы программы:**







**Скриншот «Антиплагиат»:**



**Задание 2**

Ввести 10-15 целых чисел и построить из них бинарное дерево поиска. Выполнить симметричную прошивку бинарного дерева поиска. Обойти его согласно симметричному порядку следования элементов. Реализовать поиск и вставку элементов симметрично прошитого бинарного дерева.

**Код программы:**

**colors.h**

#pragma once

#include <string>

const std::string COLOR\_RESET = "\033[0m";

const std::string COLOR\_RED = "\033[31m";

const std::string COLOR\_GREEN = "\033[32m";

const std::string COLOR\_YELLOW = "\033[33m";

const std::string COLOR\_CYAN = "\033[36m";

**threaded\_bst.h**

#pragma once

#include <iomanip>

#include <iostream>

#include <string>

#include "colors.h"

template <typename T> class ThreadedBST {

public:

struct Node {

T value;

// Parent only used to beautifully display nodes, it is not used in anything else

Node \*left, \*right, \*parent;

bool is\_left\_threaded, is\_right\_threaded;

Node(T value, Node \*parent, Node \*left, Node \*right)

: value(value), left(left), right(right), parent(parent), is\_left\_threaded(true), is\_right\_threaded(true) {

}

};

private:

ThreadedBST<T>::Node \*tree = nullptr;

// Colored "left" and "right" words

const std::string left\_string = COLOR\_YELLOW + "left" + COLOR\_RESET;

const std::string right\_string = COLOR\_CYAN + "right" + COLOR\_RESET;

const std::string moving\_left\_string = " - moving " + this->left\_string;

const std::string moving\_right\_string = " - moving " + this->right\_string;

const std::string is\_thread = COLOR\_RED + " [thread]" + COLOR\_RESET;

// Colored "value" output

std::string valueString(T value) {

return "'" + COLOR\_YELLOW + std::to\_string(value) + COLOR\_RESET + "'";

}

// Shows colored node. Green - parent, Yellow - left of parent, Cyan - right of parent

void showNode(ThreadedBST<T>::Node \*node) {

const int width = 2;

std::cout << " - ";

if (!node->parent) {

std::cout << COLOR\_GREEN;

} else if (node->parent->left == node) {

std::cout << COLOR\_YELLOW;

} else {

std::cout << COLOR\_CYAN;

}

std::cout << std::setw(width) << node->value << COLOR\_RESET << " [" << COLOR\_YELLOW << "l: ";

if (node->is\_left\_threaded && node->left) {

std::cout << std::setw(width) << node->left->value;

} else {

std::cout << std::string(width, ' ');

}

std::cout << COLOR\_RESET << " | " << COLOR\_CYAN << "r: ";

if (node->is\_right\_threaded && node->right) {

std::cout<< std::setw(width) << node->right->value;

} else {

std::cout << std::string(width, ' ');

}

std::cout << COLOR\_RESET << "]" << std::endl;

}

public:

void insert(T value) {

if (!this->tree) {

std::cout << COLOR\_GREEN << "Tree is empty, creating the root" << COLOR\_RESET << std::endl;

this->tree = new ThreadedBST<T>::Node(value, nullptr, nullptr, nullptr);

return;

}

std::cout << "Searching for the suitable space:" << std::endl;

ThreadedBST<T>::Node \*parent = nullptr;

ThreadedBST<T>::Node \*current = tree;

while (current) {

parent = current;

if (current->value > value) {

std::cout << moving\_left\_string;

if (current->is\_left\_threaded) {

std::cout << is\_thread << std::endl;

break;

}

std::cout << std::endl;

current = current->left;

} else if (current->value < value) {

std::cout << moving\_right\_string;

if (current->is\_right\_threaded) {

std::cout << is\_thread << std::endl;

break;

}

std::cout << std::endl;

current = current->right;

} else {

std::cout << COLOR\_RED << "The same value is found. Returning" << COLOR\_RESET << std::endl;

return;

}

}

if (parent->value > value) {

std::cout << "Inserting new node " << this->left\_string << " with the " << valueString(value) << std::endl;

parent->is\_left\_threaded = false;

parent->left = new ThreadedBST<T>::Node(value, parent, parent->left, parent);

} else if (parent->value < value) {

std::cout << "Inserting new node " << this->right\_string << " with the " << valueString(value) << std::endl;

parent->is\_right\_threaded = false;

parent->right = new ThreadedBST<T>::Node(value, parent, parent, parent->right);

}

}

ThreadedBST<T>::Node \*search(T value) {

if (!this->tree) {

std::cout << "The tree is empty, nothing to search" << std::endl;

return nullptr;

}

ThreadedBST<T>::Node \*current = this->tree;

while (current) {

if (current->value > value) {

std::cout << moving\_left\_string;

if (current->is\_left\_threaded) {

std::cout << is\_thread << std::endl;

break;

}

std::cout << std::endl;

current = current->left;

} else if (current->value < value) {

std::cout << moving\_right\_string;

if (current->is\_right\_threaded) {

std::cout << is\_thread << std::endl;

break;

}

std::cout << std::endl;

current = current->right;

} else {

std::cout << "The value " << this->valueString(value) << COLOR\_GREEN << " was found" << COLOR\_RESET

<< std::endl;

return current;

}

}

std::cout << "The value " << this->valueString(value) << COLOR\_RED << " was not found" << COLOR\_RESET

<< std::endl;

return nullptr;

}

ThreadedBST<T>::Node \*inorderSuccessor(ThreadedBST<T>::Node \*node) {

if (node->is\_right\_threaded) {

return node->right;

}

node = node->right;

while (node->is\_left\_threaded == false) {

node = node->left;

}

return node;

}

void show() {

if (!this->tree) {

std::cout << "The tree is empty, nothing to show" << std::endl;

return;

}

ThreadedBST<T>::Node \*current = this->tree;

while (!current->is\_left\_threaded) {

current = current->left;

}

while (current) {

showNode(current);

if (current->is\_right\_threaded) {

current = current->right;

continue;

}

current = current->right;

while (current->is\_left\_threaded == false) {

current = current->left;

}

}

}

};

**main\_threaded\_bst.cpp**

#include <cstdlib>

#include <iostream>

#include <stdlib.h>

#include "colors.h"

#include "threaded\_bst.h"

int main(void) {

ThreadedBST<int> tree;

while (true) {

std::cout << "Choose one of the options: " << std::endl

<< COLOR\_CYAN << " 1. " << COLOR\_RESET << "Insert new value." << std::endl

<< COLOR\_CYAN << " 2. " << COLOR\_RESET << "Search a value in the tree." << std::endl

<< COLOR\_CYAN << " 3. " << COLOR\_RESET << "Show the binary tree using 3 methods." << std::endl

<< COLOR\_CYAN << " 4. " << COLOR\_RESET << "Exit the program." << std::endl;

int choice;

std::cout << "Choice: " << COLOR\_YELLOW;

std::cin >> choice;

std::cout << COLOR\_RESET << std::endl;

switch (choice) {

int value;

case 1:

int size;

std::cout << "Number of values: " << COLOR\_YELLOW;

std::cin >> size;

std::cout << COLOR\_RESET << std::endl;

for (int i = 0; i < size; i++) {

std::cout << "Insert value: " << COLOR\_YELLOW;

std::cin >> value;

std::cout << COLOR\_RESET;

tree.insert(value);

std::cout << std::endl;

}

break;

case 2:

std::cout << "Insert value to search: " << COLOR\_YELLOW;

std::cin >> value;

std::cout << COLOR\_RESET << std::endl;

tree.search(value);

break;

case 3:

tree.show();

break;

case 4:

std::cout << "Exiting the program." << std::endl;

return EXIT\_SUCCESS;

default:

std::cout << "Unknown option: " << COLOR\_RED << choice << COLOR\_RESET << "." << std::endl;

break;

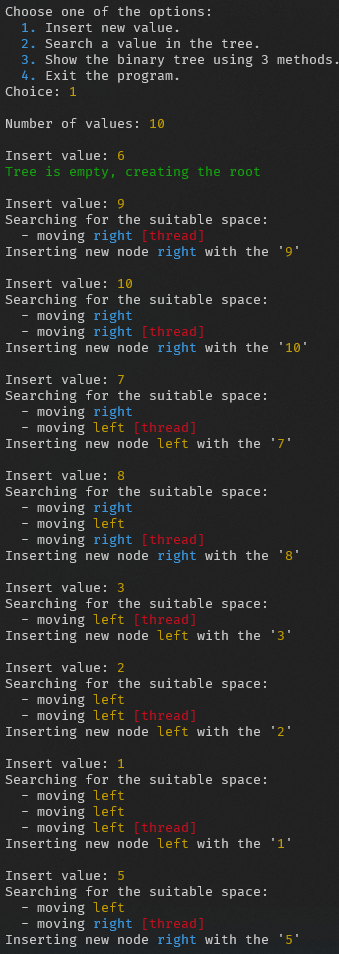
}

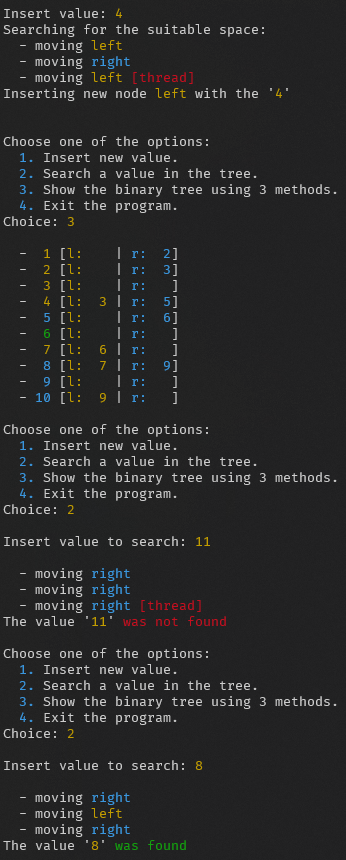
std::cout << std::endl;

}

}

**Скриншоты работы программы:**





**Скриншоты «Антиплагиат»**



**Контрольные вопросы**

1. Абстрактный тип данных "дерево" представляет собой структуру данных, состоящую из узлов, связанных между собой, где каждый узел может иметь ноль или более дочерних узлов. Дерево представляет иерархическую структуру, где один узел может выступать в качестве родительского для других узлов, а они, в свою очередь, могут быть дочерними узлами.
2. Бинарное дерево - это дерево, в котором каждый узел имеет не более двух дочерних узлов - левое и правое ветви. Бинарное дерево поиска - это особый тип бинарного дерева, где для каждого узла выполняется условие: значение информационного поля, связанного с вершиной, больше любого значения из левого поддерева и меньше, чем содержимое любого узла его правого поддерева.
3. Три основных способа обхода бинарных деревьев:
   1. Прямой (pre-order) обход: порядок следования вершин - корень, левое поддерево, правое поддерево.
   2. Симметричный (in-order) обход: порядок следования вершин - левое поддерево, корень, правое поддерево.
   3. Обратный (post-order) обход: порядок следования вершин - левое поддерево, правое поддерево, корень.
4. Достоинства прошивки бинарных деревьев:

* Быстрый обход без использования стека.
* Возможность определения предшественника и преемника для каждой вершины.
* Отсутствие необходимости в дополнительной памяти для стека при обходе.

Недостатки прошивки бинарных деревьев:

* Большее время вставки новой вершины из-за необходимости поддерживать два типа связей.
* Ограничение использования прошитых деревьев в задачах с частыми изменениями в структуре дерева.

1. АВЛ-дерево отличается от идеально сбалансированного бинарного дерева следующими особенностями:

* АВЛ-дерево может быть неидеально сбалансированным, но всегда сбалансировано по высоте поддеревьев. Разница в высоте поддеревьев каждого узла в АВЛ-дереве не превосходит 1, в то время как в идеально сбалансированном бинарном дереве разница в высоте поддеревьев равна 0.
* Для поддержания сбалансированности АВЛ-дерева при вставке и удалении элементов необходимо выполнять операции балансировки, такие как повороты и перебалансировки поддеревьев.
* Поиск элементов в АВЛ-дереве может быть выполнен за время, пропорциональное логарифму от количества узлов в дереве, что делает его эффективным для поиска.

**Заключение**

Во время выполнения данной контрольной работы были изучены бинарные деревья и их особенности. Освоены различные способы построения и обхода бинарных деревьев, включая прямой, симметричный и обратный порядок обхода. Также изучены процедуры поиска, вставки и удаления элементов в бинарных деревьях поиска. Особое внимание было уделено прошивке бинарных деревьев и ее преимуществам в оптимизации обхода дерева.

Выполнение заданий в рамках контрольной работы позволило закрепить теоретические знания о бинарных деревьях и их применении. Были получены знания реализовывать различные операции с данными в бинарных деревьях, а также применять прошивку для оптимизации обхода.

Также стоит отметить, что в работе было бегло рассмотрено АВЛ-дерево как развитие бинарного дерева поиска. Это дерево отличается от обычного дерева поиска тем, что поддерживает сбалансированность дерева при включении и удалении элементов. Это позволяет осуществлять поиск элементов в АВЛ-дереве за время, пропорциональное логарифму от количества узлов в дереве, что делает его эффективным для поиска.