Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №4

по курсу «Логика и основы алгоритмизации

в инженерных задачах»

на тему «Бинарное дерево поиска»

Выполнил: ст. гр. 24ВВВ1

Будников А.С.

Принял:

к.т.н, доцент Юрова О. В.

к.т.н. Деев М.В.

Пенза

2025

**Цель работы:**

Реализовать бинарное дерево поиска.

**Лабораторное задание:**

1. Реализовать алгоритм поиска вводимого с клавиатуры значения в уже созданном дереве.
2. Реализовать функцию подсчёта числа вхождений заданного элемента в дерево.
3. \*Изменить функцию добавления элементов для исключения добавления одинаковых символов.
4. \*Оценить сложность процедуры поиска по значению в бинарном дереве.

**Ход работы:**

1. Для поиска вводимого значения с клавиатуры было добавлено 2 функции:

a) bool search(T value) – функция, которая посредством вспомогательного рекурсивного метода последовательно проходит по ветвям созданного дерева в зависимости от текущего значения и проверяет вхождение элемента. В случае, если элемент найден, метод возвращает *true*, иначе *false*.

б) void get\_path(T value) const – функция, которая посредством вспомогательного рекурсивного метода последовательно проходит по ветвям созданного дерева в зависимости от текущего значения. В случае, если элемент найден, вспомогательный метод выводит пути ко всем его вхождениям в консоль, иначе функция выбрасывает ошибку.

Код всех методов представлен в листинге программы.

2. Подсчет числа вхождений осуществляет приватный рекурсивная функция void count\_entries\_helper(Node<T>\* r, int& counter, const T& value) const, которая последовательно проходит по ветвям созданного дерева и увеличивает счетчик, объявленный в публичном методе int count\_entries(const T& value) const, и через который осуществляется взаимодействие с программой.

3. Для исключения добавления одинаковых элементов в *BST*, рекурсивный метод Node<T>\* insert\_recursive\_repeat(Node<T>\* node, T value) был продублирован, а в условии прохождения по ветвям дерева сравнительные операторы «>=» и «<=» были заменены на «>» и «<» для того, чтобы обеспечить свойство уникальности элементов в бинарном дереве поиска, т.е. условие «<=» позволяет элементам, равным текущему узлу, попадать в левое поддерево, что приводит к появлению повторяющихся значений в дереве, а строгие неравенства игнорируют случай равенства.

4. Процедура *search\_recursive*, которая реализует классический поиск в бинарном дереве поиска. Её временная сложность принципиально зависит от сбалансированности дерева. В идеальном случае сбалансированного дерева высотой *h ≈ log₂n* сложность составляет *O(log n)*, поскольку на каждом уровне рекурсии отбрасывается примерно половина оставшегося поддерева. Однако в худшем сценарии вырожденного дерева, превратившегося в линейный список, сложность деградирует до *O(n)*, где n - количество узлов. Пространственная сложность определяется глубиной рекурсии и составляет *O(h)*, что в сбалансированном случае равно *O(log n)*, а в вырожденном - *O(n)*.

Процедура *find\_path* демонстрирует принципиально иную алгоритмическую природу, реализуя полный обход дерева без использования свойств упорядоченности. Её временная сложность всегда составляет *O(n)*, поскольку в худшем случае необходимо посетить все узлы независимо от расположения искомого элемента. Эта процедура не прекращает поиск после первого найденного совпадения, продолжая обход всего дерева для выявления всех возможных путей. Пространственная сложность также равна *O(h)* из-за рекурсивной природы и хранения текущего пути, но с дополнительными затратами на вывод найденных путей, которые в худшем случае могут достигать *O(n×h)*.

**Вывод:** в ходе выполнения лабораторной работы по реализации бинарного дерева поиска были успешно решены поставленные задачи и получены практические навыки работы с древовидными структурами данных. Была разработана программа, реализующая основные операции с бинарным деревом поиска, включая добавление элементов, поиск значений и подсчет вхождений. Основным достижением работы стало создание эффективного алгоритма поиска в бинарном дереве, который продемонстрировал логарифмическую сложность *O(log n)* в среднем случае для сбалансированного дерева. Однако было выявлено, что в худшем случае, при вырождении дерева в линейный список, сложность поиска возрастает до *O(n)*, что подчеркивает важность поддержания сбалансированности структуры.

**Листинг программы:**

**Файл *main.cpp***

#include <iostream>  
#include <limits>  
#include <cstdlib>  
  
#ifndef \_WIN32  
#include <windows.h  
#endif  
  
#include "../lib/tui/binary\_tree\_tui.h"  
  
int main() {  
#ifndef \_WIN32  
 HANDLE h = GetStdHandle(STD\_OUTPUT\_HANDLE);  
 DWORD dwMode = 0;  
  
 if (!GetConsoleMode(h, &dwMode)) {  
 throw std::runtime\_error("GetConsoleMode failed");  
 return 1;  
 }  
  
 dwMode |= ENABLE\_VIRTUAL\_TERMINAL\_PROCESSING;  
  
 if (!SetConsoleMode(h, dwMode)) {  
 throw std::runtime\_error("SetConsoleMode failed");  
 return 1;  
 }  
#endif  
  
 std::cout << "Binary Tree Playground" << std::endl;  
 std::cout << "============================" << std::endl;  
  
 std::cout << "Choose data type:" << std::endl;  
 std::cout << "1 - int" << std::endl;  
 std::cout << "2 - double" << std::endl;  
 std::cout << "3 - string" << std::endl;  
 std::cout << "4 - char" << std::endl;  
 std::cout << "Enter choice (1-4): ";  
  
 char type\_choice;  
 std::cin >> type\_choice;  
 std::cin.ignore(std::numeric\_limits<std::streamsize>::*max*(), '\n');  
  
 switch (type\_choice) {  
 case '1': {  
 BinaryTreePlayground::BinaryTreePlaygroundManager<int> manager;  
 manager.run();  
 break;  
 }  
 case '2': {  
 BinaryTreePlayground::BinaryTreePlaygroundManager<double> manager;  
 manager.run();  
 break;  
 }  
 case '3': {  
 BinaryTreePlayground::BinaryTreePlaygroundManager<std::string> manager;  
 manager.run();  
 break;  
 }  
 case '4': {  
 BinaryTreePlayground::BinaryTreePlaygroundManager<char> manager;  
 manager.run();  
 break;  
 }  
 default: {  
 std::cout << "Invalid choice! Using int by default." << std::endl;  
 BinaryTreePlayground::BinaryTreePlaygroundManager<int> manager;  
 manager.run();  
 break;  
 }  
 }  
  
 return 0;  
}

**Файл *binary\_tree.h***

//  
// Created by oblepixa on 9/28/25.  
//  
  
#ifndef BINARY\_TREE\_H  
#define BINARY\_TREE\_H  
#include <iostream>  
#include <queue>  
  
// Template class for node of a Binary tree  
template<typename T>  
class Node {  
public:  
 // Data held by the node  
 T data;  
 // Pointer to the left child  
 Node\* left;  
 // Pointer to the right child  
 Node\* right;  
  
 // Constructor to initialize node with a value  
 explicit Node (T value) : data(value), left(nullptr), right(nullptr) {}  
 // Destructor  
 ~Node() = default;  
  
 // Disable copying  
 Node(const Node&) = delete;  
 Node& operator=(const Node&) = delete;  
 Node(Node&&) = delete;  
 Node& operator=(Node&&) = delete;  
};  
  
template<typename T>  
class BinaryTree {  
private:  
 // Pointer to the root of the tree  
 Node<T>\* root;  
  
 // Method to properly clear the tree if destructor was called  
 void clear\_recursive(Node<T>\* node) {  
 if (node) {  
 clear\_recursive(node->left);  
 clear\_recursive(node->right);  
 delete node;  
 }  
 }  
  
 // Recursive method to search for a value in the tree  
 bool search\_recursive(Node<T>\* current, T value) {  
 if (current == nullptr) return false;  
 if (current->data == value) return true;  
  
 if (value < current->data) return search\_recursive(current->left, value);  
 return search\_recursive(current->right, value);  
 }  
  
 // Method for Recursive inorder traversal of the tree  
 void inorder\_recursive(Node<T>\* node) {  
 if (node != nullptr) {  
 inorder\_recursive(node->left);  
 std::cout << node->data << " ";  
 inorder\_recursive(node->right);  
 }  
 }  
  
 // Method for Recursive preorder traversal of the tree  
 void preorderRecursive(Node<T>\* node) {  
 if (node != nullptr) {  
 std::cout << node->data << " ";  
 preorderRecursive(node->left);  
 preorderRecursive(node->right);  
 }  
 }  
  
 void count\_entries\_helper(Node<T>\* r, int& counter, const T& value) const {  
 if (r == nullptr) return;  
  
 if (value == r->data) ++counter;  
  
 count\_entries\_helper(r->left, counter, value);  
 count\_entries\_helper(r->right, counter, value);  
 }  
  
 void print\_tree\_helper(Node<T>\* r, const int level) const {  
 if (r == nullptr) return;  
  
 print\_tree\_helper(r->right, level + 1);  
 for (int i = 0; i < level; i++) {  
 std::cout << " ";  
 }  
  
 std::cout << r->data << std::endl;  
 print\_tree\_helper(r->left, level + 1);  
 }  
  
 bool find\_path(Node<T>\* r, T target, std::vector<T>& current\_path) const {  
 if (r == nullptr) return false;  
  
 bool found\_any = false;  
 current\_path.push\_back(r->data);  
  
 if (r->data == target) {  
 for (const auto& val : current\_path) std::cout << val << " ";  
 std::cout << std::endl;  
 found\_any = true;  
 }  
  
 const bool left = find\_path(r->left, target, current\_path);  
 const bool right = find\_path(r->right, target, current\_path);  
  
 current\_path.pop\_back();  
 return found\_any || left || right;  
 }  
  
 Node<T>\* insert\_recursive\_repeat(Node<T>\* node, T value) {  
 if (node == nullptr) return new Node<T>(value);  
 if (value <= node->data) node->left = insert\_recursive\_repeat(node->left, value);  
 else node->right = insert\_recursive\_repeat(node->right, value);  
 return node;  
 }  
  
 Node<T>\* insert\_recursive(Node<T>\* node, T value) {  
 if (node == nullptr) return new Node<T>(value);  
 if (value < node->data) node->left = insert\_recursive(node->left, value);  
 else if (value > node->data) node->right = insert\_recursive(node->right, value);  
 return node;  
 }  
  
 int height\_recursive(Node<T>\* node) const {  
 if (node == nullptr) return -1;  
 const int left\_height = height\_recursive(node->left);  
 const int right\_height = height\_recursive(node->right);  
  
 if (left\_height > right\_height) return left\_height + 1;  
 return right\_height + 1;  
 }  
  
public:  
 // Constructor to initialize the tree  
 BinaryTree() : root(nullptr) {}  
 // Destructor  
 ~BinaryTree() {  
 clear\_recursive(root);  
 root = nullptr;  
 }  
 // Disable copying  
 BinaryTree(const BinaryTree&) = delete;  
 BinaryTree& operator=(const BinaryTree&) = delete;  
 BinaryTree(BinaryTree&&) = default;  
 BinaryTree& operator=(BinaryTree&&) = default;  
  
 // Access methods  
 Node<T> \*get\_root() { return root; }  
 const Node<T> \*get\_root() const { return root; }  
  
 // Methods to insert node in the binary tree (excluding the same elements)  
 void insert\_node(T value, const bool repeat) {  
 if (!repeat) {  
 if (search(value)) return;  
 root = insert\_recursive(root, value);  
 return;  
 }  
 root = insert\_recursive\_repeat(root, value);  
 }  
  
 // Method to search for a value in the tree  
 bool search(T value) {  
 return search\_recursive(root, value);  
 }  
  
 // Method to perform inorder traversal of the tree  
 void inorder() {  
 inorder\_recursive(root);  
 std::cout << std::endl;  
 }  
  
 // Method to perform preorder traversal of the tree  
 void preorder() {  
 preorderRecursive(root);  
 std::cout << std::endl;  
 }  
  
 // Method to print the tree  
 void print\_tree() const {  
 print\_tree\_helper(root,0);  
 }  
  
 // Method of calculating the number of entries of a given element into a tree.  
 int count\_entries(const T& value) const {  
 int n = 0;  
 count\_entries\_helper(root, n, value);  
 return n;  
 }  
  
 void find\_levels() const{  
 std::cout << "Min level: 0" << std::endl;  
 std::cout << "Max level: " << height\_recursive(root) << std::endl;  
 }  
  
 // Method to search a path to a value in the tree  
 void get\_path(T value) const {  
 std::vector<T> current\_path;  
  
 if (const bool found = find\_path(root, value, current\_path); !found) throw std::runtime\_error("Not found");  
 }  
};  
  
#endif //BINARY\_TREE\_H

**Файл *binary\_tree\_tui.h***

#ifndef BINARY\_TREE\_PLAYGROUND\_H  
#define BINARY\_TREE\_PLAYGROUND\_H  
  
#include "../binarytree/binary\_tree.h"  
#include <functional>  
#include <iostream>  
#include <sstream>  
#include <memory>  
#include <unordered\_map>  
#include <utility>  
#include <vector>  
#include <algorithm>  
#include <iomanip>  
  
namespace Colors {  
 // ANSI color codes for terminal output  
 const std::string RESET = "\033[0m";  
 const std::string RED = "\033[31m";  
 const std::string GREEN = "\033[32m";  
 const std::string YELLOW = "\033[33m";  
 const std::string BLUE = "\033[34m";  
 const std::string MAGENTA = "\033[35m";  
 const std::string CYAN = "\033[36m";  
 const std::string BOLD = "\033[1m";  
 const std::string UNDERLINE = "\033[4m";  
}  
  
namespace BinaryTreePlayground {  
  
 // Convert any value type to string for display purposes  
 template<typename T>  
 std::string value\_to\_string(const T& value) {  
 if constexpr (std::is\_same\_v<T, std::string>) {  
 return value;  
 } else if constexpr (std::is\_same\_v<T, char>) {  
 return std::string(1, value);  
 } else {  
 return std::to\_string(value);  
 }  
 }  
  
 // Wrapper class that adds history tracking and utility methods to BinaryTree  
 template<typename T>  
 class TreeWrapper {  
 private:  
 std::unique\_ptr<BinaryTree<T>> tree\_; // The actual binary tree  
 std::string name\_; // Name identifier for this tree  
 std::vector<std::string> history\_; // Operation history (last 20 operations)  
  
 public:  
 explicit TreeWrapper(std::string name) : name\_(std::move(name)) {  
 tree\_ = std::make\_unique<BinaryTree<T>>();  
 }  
  
 // Getters  
 [[nodiscard]] const std::string &get\_name() const { return name\_; }  
 BinaryTree<T>\* get\_tree() { return tree\_.get(); }  
 const BinaryTree<T>\* get\_tree() const { return tree\_.get(); }  
 [[nodiscard]] const std::vector<std::string>& get\_history() const { return history\_; }  
  
 // Add operation to history with size limit  
 void add\_to\_history(const std::string& operation) {  
 history\_.push\_back(operation);  
 if (history\_.size() > 20) {  
 history\_.erase(history\_.begin());  
 }  
 }  
  
 // Insert value into the tree and record operation  
 void insert(const T &value, bool& repeat) {  
 tree\_->insert\_node(value, repeat);  
 add\_to\_history("insert " + value\_to\_string(value));  
 }  
  
 // Search for value in tree and record operation with result  
 bool search(const T &value) {  
 const bool result = tree\_->search(value);  
 add\_to\_history("search " + value\_to\_string(value) + " -> " + (result ? "found" : "not found"));  
 return result;  
 }  
  
 // Perform inorder traversal and capture output  
 std::string inorder() {  
 const std::stringstream buffer;  
 std::streambuf\* old = std::cout.rdbuf(buffer.rdbuf());  
 tree\_->inorder();  
 std::cout.rdbuf(old);  
 add\_to\_history("inorder");  
 return buffer.str();  
 }  
  
 // Perform preorder traversal and capture output  
 std::string preorder() {  
 const std::stringstream buffer;  
 std::streambuf\* old = std::cout.rdbuf(buffer.rdbuf());  
 tree\_->preorder();  
 std::cout.rdbuf(old);  
 add\_to\_history("preorder");  
 return buffer.str();  
 }  
  
 // Count occurrences of value in tree  
 int count\_entries(const T &value) {  
 const int count = tree\_->count\_entries(value);  
 add\_to\_history("count " + value\_to\_string(value) + " -> " + std::to\_string(count));  
 return count;  
 }  
  
 // Get path to value in tree  
 std::string get\_path(const T &value) {  
 const std::stringstream buffer;  
 std::streambuf\* old = std::cout.rdbuf(buffer.rdbuf());  
 tree\_->get\_path(value);  
 std::cout.rdbuf(old);  
 add\_to\_history("path " + value\_to\_string(value));  
 return buffer.str();  
 }  
  
 // Print tree structure  
 std::string print\_tree() {  
 const std::stringstream buffer;  
 std::streambuf\* old = std::cout.rdbuf(buffer.rdbuf());  
 tree\_->print\_tree();  
 std::cout.rdbuf(old);  
 add\_to\_history("print");  
 return buffer.str();  
 }  
  
 std::string find\_level() {  
 const std::stringstream buffer;  
 std::streambuf\* old = std::cout.rdbuf(buffer.rdbuf());  
 tree\_->find\_levels();  
 std::cout.rdbuf(old);  
 add\_to\_history("find level");  
 return buffer.str();  
 }  
  
 // Check if tree is empty  
 [[nodiscard]] bool empty() const {  
 return tree\_->get\_root() == nullptr;  
 }  
  
 // Get number of nodes in tree  
 [[nodiscard]] int size() const {  
 return count\_nodes(tree\_->get\_root());  
 }  
  
 // Clear all nodes from tree  
 void clear() {  
 tree\_ = std::make\_unique<BinaryTree<T>>();  
 add\_to\_history("clear");  
 }  
  
 // Display tree statistics  
 void print\_stats() const {  
 auto root = tree\_->get\_root();  
 if (!root) {  
 std::cout << Colors::YELLOW << "Tree is empty" << Colors::RESET << std::endl;  
 return;  
 }  
  
 std::cout << Colors::CYAN << "=== Tree Statistics ===" << Colors::RESET << std::endl;  
 std::cout << "Root value: " << Colors::BOLD << root->data << Colors::RESET << std::endl;  
 std::cout << "Total nodes: " << Colors::BOLD << count\_nodes(root) << Colors::RESET << std::endl;  
 std::cout << Colors::BOLD;  
 tree\_->find\_levels();  
 std::cout << Colors::RESET;  
 std::cout << "Min value: " << Colors::BOLD << find\_min(root) << Colors::RESET << std::endl;  
 std::cout << "Max value: " << Colors::BOLD << find\_max(root) << Colors::RESET << std::endl;  
 }  
  
 private:  
 // Recursively count nodes in subtree  
 int count\_nodes(const Node<T>\* node) const {  
 if (!node) return 0;  
 return 1 + count\_nodes(node->left) + count\_nodes(node->right);  
 }  
  
 // Find minimum value in subtree  
 T find\_min(const Node<T>\* node) const {  
 while (node && node->left) node = node->left;  
 return node ? node->data : T{};  
 }  
  
 // Find maximum value in subtree  
 T find\_max(const Node<T>\* node) const {  
 while (node && node->right) node = node->right;  
 return node ? node->data : T{};  
 }  
 };  
  
 // Main manager class for handling multiple trees and user interactions  
 template<typename T>  
 class BinaryTreePlaygroundManager {  
 private:  
 std::unordered\_map<std::string, std::unique\_ptr<TreeWrapper<T>>> trees\_; // All created trees  
 std::unordered\_map<std::string, std::function<void(std::istringstream &)>> commands\_; // Command handlers  
 std::string current\_tree\_; // Currently selected tree name  
 int tree\_counter\_ = 0; // Counter for auto-generating tree names  
 std::vector<std::string> command\_history\_; // Command history (last 20 commands)  
 bool show\_colors\_ = true; // Flag for colored output  
  
 // Initialize all supported commands with their handlers  
 void initialize\_commands() {  
 commands\_ = {  
 // Create a new tree with optional name  
 {  
 "create", [this](std::istringstream &iss) {  
 std::string name;  
 if (!(iss >> name)) name = generate\_tree\_name();  
 handle\_create(name);  
 }  
 },  
 // Switch to using specified tree  
 {  
 "use", [this](std::istringstream &iss) {  
 std::string name;  
 if (!(iss >> name)) throw std::runtime\_error("Invalid tree name");  
 handle\_use(name);  
 }  
 },  
 // Remove specified tree  
 {  
 "remove", [this](std::istringstream &iss) {  
 std::string name;  
 if (!(iss >> name)) throw std::runtime\_error("Invalid tree name");  
 handle\_remove(name);  
 }  
 },  
 // Insert value into current tree  
 {  
 "insert", [this](std::istringstream &iss) {  
 T value;  
 bool repeat;  
 if (!(iss >> value) || !(iss >> repeat)) throw std::runtime\_error("Invalid value");  
 handle\_insert(value, repeat);  
 }  
 },  
 // Insert value into current tree (alias)  
 {  
 "+", [this](std::istringstream &iss) {  
 T value;  
 bool repeat;  
 if (!(iss >> value) || !(iss >> repeat)) throw std::runtime\_error("Invalid value");  
 handle\_insert(value, repeat);  
 }  
 },  
 // Search for value in current tree  
 {  
 "search", [this](std::istringstream &iss) {  
 T value;  
 if (!(iss >> value)) throw std::runtime\_error("Invalid value");  
 handle\_search(value);  
 }  
 },  
 // Perform inorder traversal  
 {"inorder", [this](std::istringstream&) { handle\_inorder(); }},  
 // Perform preorder traversal  
 {"preorder", [this](std::istringstream &) { handle\_preorder(); }},  
 // Count occurrences of value  
 {  
 "count", [this](std::istringstream &iss) {  
 T value;  
 if (!(iss >> value)) throw std::runtime\_error("Invalid value");  
 handle\_count(value);  
 }  
 },  
 // Show path to value  
 {  
 "path", [this](std::istringstream &iss) {  
 T value;  
 if (!(iss >> value)) throw std::runtime\_error("Invalid value");  
 handle\_path(value);  
 }  
 },  
 // Print tree structure  
 {"print", [this](std::istringstream &) { handle\_print(); }},  
 // Print levels of subtree  
 {"levels", [this](std::istringstream &) { handle\_get\_level(); }},  
 // Clear current tree  
 {"clear", [this](std::istringstream &) { handle\_clear(); }},  
 // List all available trees  
 {"list", [this](std::istringstream &) { handle\_list(); }},  
 // Check if current tree is empty  
 {"empty", [this](std::istringstream &) { handle\_empty(); }},  
 // Get size of current tree  
 {"size", [this](std::istringstream &) { handle\_size(); }},  
 // Show statistics of current tree  
 {"stats", [this](std::istringstream &) { handle\_stats(); }},  
 // Show command history  
 {"history", [this](std::istringstream &) { handle\_history(); }},  
 // Show operation history for current tree  
 {"treehistory", [this](std::istringstream &) { handle\_tree\_history(); }},  
 // Toggle colored output  
 {"colors", [this](std::istringstream &) { handle\_colors(); }},  
 // Show help information  
 {"help", [this](std::istringstream &) { handle\_help(); }},  
 // Show help information (alias)  
 {"?", [this](std::istringstream &) { handle\_help(); }}  
 };  
 }  
  
 // Get pointer to currently selected tree  
 TreeWrapper<T>\* get\_current\_tree() const {  
 auto it = trees\_.find(current\_tree\_);  
 if (it == trees\_.end()) throw std::runtime\_error("No tree selected! Use 'use <name>' first.");  
 return it->second.get();  
 }  
  
 // Generate unique tree name when not provided by user  
 std::string generate\_tree\_name() {  
 return "tree\_" + std::to\_string(++tree\_counter\_);  
 }  
  
 // Print colored message to console  
 void print\_colored(const std::string& message, const std::string& color = Colors::RESET) {  
 if (show\_colors\_) {  
 std::cout << color << message << Colors::RESET;  
 } else {  
 std::cout << message;  
 }  
 }  
  
 // Print colored message with newline  
 void println\_colored(const std::string& message, const std::string& color = Colors::RESET) {  
 print\_colored(message + "\n", color);  
 }  
  
 // Handle tree creation  
 void handle\_create(const std::string &name) {  
 std::string actual\_name = name.empty() ? generate\_tree\_name() : name;  
  
 if (trees\_.count(actual\_name)) {  
 println\_colored("Error: Tree '" + actual\_name + "' already exists!", Colors::RED);  
 return;  
 }  
  
 trees\_[actual\_name] = std::make\_unique<TreeWrapper<T>>(actual\_name);  
 current\_tree\_ = actual\_name;  
 println\_colored("✓ Created tree: '" + actual\_name + "'", Colors::GREEN);  
 println\_colored("Now using: " + actual\_name, Colors::CYAN);  
 }  
  
 // Handle switching to different tree  
 void handle\_use(const std::string &name) {  
 if (trees\_.count(name)) {  
 current\_tree\_ = name;  
 println\_colored("✓ Now using: " + name, Colors::GREEN);  
 } else {  
 println\_colored("Error: Tree '" + name + "' not found!", Colors::RED);  
 }  
 }  
  
 // Handle value insertion  
 void handle\_insert(const T &value, bool& repeat) {  
 auto tree = get\_current\_tree();  
 tree->insert(value, repeat);  
 println\_colored("✓ Inserted: " + value\_to\_string(value), Colors::GREEN);  
 }  
  
 // Handle value search  
 void handle\_search(const T &value) {  
 auto tree = get\_current\_tree();  
 bool found = tree->search(value);  
 std::string result = "Value '" + value\_to\_string(value) + "' was " +  
 (found ? "FOUND" : "NOT FOUND") + " in the tree";  
 println\_colored(result, found ? Colors::GREEN : Colors::YELLOW);  
 }  
  
 // Handle inorder traversal  
 void handle\_inorder() {  
 auto tree = get\_current\_tree();  
 println\_colored("Inorder traversal:", Colors::CYAN);  
 std::string result = tree->inorder();  
 if (result.empty()) {  
 println\_colored("(empty)", Colors::YELLOW);  
 } else {  
 std::cout << result;  
 }  
 }  
  
 // Handle preorder traversal  
 void handle\_preorder() {  
 auto tree = get\_current\_tree();  
 println\_colored("Preorder traversal:", Colors::CYAN);  
 std::string result = tree->preorder();  
 if (result.empty()) {  
 println\_colored("(empty)", Colors::YELLOW);  
 } else {  
 std::cout << result;  
 }  
 }  
  
 // Handle value counting  
 void handle\_count(const T &value) {  
 auto tree = get\_current\_tree();  
 const int count = tree->count\_entries(value);  
 const std::string message = "Value '" + value\_to\_string(value) + "' appears " +  
 std::to\_string(count) + " time(s) in the tree";  
 println\_colored(message, Colors::CYAN);  
 }  
  
 // Handle path display  
 void handle\_path(const T &value) {  
 auto tree = get\_current\_tree();  
 println\_colored("Path to '" + value\_to\_string(value) + "': ", Colors::CYAN);  
 const std::string result = tree->get\_path(value);  
 std::cout << result;  
 }  
  
 // Handle tree printing  
 void handle\_print() {  
 auto tree = get\_current\_tree();  
 println\_colored("Tree structure:", Colors::CYAN);  
 if (const std::string result = tree->print\_tree(); result.empty()) {  
 println\_colored("(empty)", Colors::YELLOW);  
 } else {  
 std::cout << result;  
 }  
 }  
  
 // Handle level display  
 void handle\_get\_level() {  
 auto tree = get\_current\_tree();  
 tree->find\_level();  
 if (const std::string result = tree->find\_level(); result.empty()) {  
 println\_colored("(empty)", Colors::YELLOW);  
 } else {  
 std::cout << result;  
 }  
 }  
  
 // Handle tree clearing  
 void handle\_clear() {  
 auto tree = get\_current\_tree();  
 tree->clear();  
 println\_colored("✓ Tree cleared", Colors::GREEN);  
 }  
  
 // Handle tree size query  
 void handle\_size() {  
 auto tree = get\_current\_tree();  
 std::cout << "Size: " << tree->size() << std::endl;  
 }  
  
 // Handle tree listing  
 void handle\_list() {  
 if (trees\_.empty()) {  
 println\_colored("No trees created!", Colors::YELLOW);  
 return;  
 }  
  
 println\_colored("Available trees:", Colors::CYAN);  
 for (const auto &[name, tree]: trees\_) {  
 std::string marker = (name == current\_tree\_) ? " → " : " ";  
 std::string status = tree->empty() ? "empty" : "non-empty";  
 std::string color = (name == current\_tree\_) ? Colors::GREEN : Colors::RESET;  
  
 print\_colored(marker + name, color);  
 std::cout << " (" << status << ")" << std::endl;  
 }  
 }  
  
 // Handle empty check  
 void handle\_empty() {  
 auto tree = get\_current\_tree();  
 const std::string result = tree->empty() ? "empty" : "not empty";  
 const std::string color = tree->empty() ? Colors::YELLOW : Colors::GREEN;  
 println\_colored(result, color);  
 }  
  
 // Handle statistics display  
 void handle\_stats() {  
 auto tree = get\_current\_tree();  
 tree->print\_stats();  
 }  
  
 // Handle command history display  
 void handle\_history() {  
 if (command\_history\_.empty()) {  
 println\_colored("No command history!", Colors::YELLOW);  
 return;  
 }  
  
 println\_colored("Command history (last 20):", Colors::CYAN);  
 for (size\_t i = 0; i < command\_history\_.size(); ++i) {  
 std::cout << " " << std::setw(2) << (i + 1) << ". " << command\_history\_[i] << std::endl;  
 }  
 }  
  
 // Handle tree operation history display  
 void handle\_tree\_history() {  
 auto tree = get\_current\_tree();  
 const auto& history = tree->get\_history();  
  
 if (history.empty()) {  
 println\_colored("No operations performed on this tree!", Colors::YELLOW);  
 return;  
 }  
  
 println\_colored("Operation history for '" + tree->get\_name() + "':", Colors::CYAN);  
 for (size\_t i = 0; i < history.size(); ++i) {  
 std::cout << " " << std::setw(2) << (i + 1) << ". " << history[i] << std::endl;  
 }  
 }  
  
 // Handle color output toggle  
 void handle\_colors() {  
 show\_colors\_ = !show\_colors\_;  
 const std::string status = show\_colors\_ ? "enabled" : "disabled";  
 println\_colored("Colors " + status, Colors::GREEN);  
 }  
  
 // Handle tree removal  
 void handle\_remove(const std::string &name) {  
 if (trees\_.count(name)) {  
 if (current\_tree\_ == name) {  
 current\_tree\_.clear();  
 }  
 trees\_.erase(name);  
 println\_colored("✓ Removed: " + name, Colors::GREEN);  
 } else {  
 println\_colored("Error: Tree '" + name + "' not found!", Colors::RED);  
 }  
 }  
  
 // Handle help display  
 void handle\_help() {  
 println\_colored("\n" + Colors::BOLD + "=== Binary Tree Playground Commands ===" + Colors::RESET, Colors::CYAN);  
 std::cout << Colors::BOLD << "Tree Management:" << Colors::RESET << std::endl;  
 std::cout << " create [name] - Create new tree (auto-name if omitted)" << std::endl;  
 std::cout << " use <name> - Switch to tree" << std::endl;  
 std::cout << " remove <name> - Remove tree" << std::endl;  
 std::cout << " list - List all trees" << std::endl;  
  
 std::cout << Colors::BOLD << "\nTree Operations:" << Colors::RESET << std::endl;  
 std::cout << " insert <value> - Insert value into current tree" << std::endl;  
 std::cout << " search <value> - Search for value" << std::endl;  
 std::cout << " count <value> - Count occurrences of value" << std::endl;  
 std::cout << " path <value> - Show path to value" << std::endl;  
 std::cout << " clear - Clear current tree" << std::endl;  
  
 std::cout << Colors::BOLD << "\nTree Analysis:" << Colors::RESET << std::endl;  
 std::cout << " levels - Print min and max levels of subtree" << std::endl;  
 std::cout << " inorder - Inorder traversal" << std::endl;  
 std::cout << " preorder - Preorder traversal" << std::endl;  
 std::cout << " print - Print tree structure" << std::endl;  
 std::cout << " size - Get tree size" << std::endl;  
 std::cout << " stats - Show tree statistics" << std::endl;  
 std::cout << " empty - Check if current tree is empty" << std::endl;  
  
 std::cout << Colors::BOLD << "\nHistory & Settings:" << Colors::RESET << std::endl;  
 std::cout << " history - Show command history" << std::endl;  
 std::cout << " treehistory - Show tree operation history" << std::endl;  
 std::cout << " colors - Toggle color output" << std::endl;  
 std::cout << " help, ? - Show this help" << std::endl;  
 std::cout << " exit, quit - Exit playground" << std::endl;  
  
 std::cout << Colors::BOLD << "\nExamples:" << Colors::RESET << std::endl;  
 std::cout << " create mytree " << Colors::YELLOW << "# Create tree named 'mytree'" << Colors::RESET << std::endl;  
 std::cout << " insert 50 1 " << Colors::YELLOW << "# Insert value 50 with repeat" << Colors::RESET << std::endl;  
 std::cout << " stats " << Colors::YELLOW << "# Show tree statistics" << Colors::RESET << std::endl;  
 std::cout << Colors::BOLD << "========================================" << Colors::RESET << std::endl;  
 }  
  
 // Add command to history with size limit  
 void add\_to\_history(const std::string& command) {  
 command\_history\_.push\_back(command);  
 if (command\_history\_.size() > 20) {  
 command\_history\_.erase(command\_history\_.begin());  
 }  
 }  
  
 public:  
 BinaryTreePlaygroundManager() {  
 initialize\_commands();  
 }  
  
 // Main interactive loop  
 void run() {  
 println\_colored("\n" + Colors::BOLD + "Binary Tree Playground" + Colors::RESET, Colors::GREEN);  
 println\_colored("Type 'help' for commands, 'exit' to quit", Colors::CYAN);  
  
 std::string command;  
 while (true) {  
 // Display prompt with current tree context  
 if (current\_tree\_.empty()) {  
 print\_colored("bt-playground> ", Colors::YELLOW);  
 } else {  
 print\_colored("bt[" + current\_tree\_ + "]> ", Colors::GREEN);  
 }  
  
 if (!std::getline(std::cin, command)) break;  
 if (command.empty()) continue;  
  
 add\_to\_history(command);  
  
 std::istringstream iss(command);  
 std::string action;  
 iss >> action;  
  
 // Exit condition  
 if (action == "exit" || action == "quit") {  
 println\_colored("👋 Exiting Binary Tree Playground...", Colors::GREEN);  
 break;  
 }  
  
 try {  
 // Execute command if found  
 if (auto it = commands\_.find(action); it != commands\_.end()) {  
 it->second(iss);  
 } else {  
 println\_colored("Unknown command: '" + action + "'. Type 'help' for available commands.", Colors::RED);  
 }  
 } catch (const std::exception &e) {  
 println\_colored("Error: " + std::string(e.what()), Colors::RED);  
 } catch (...) {  
 println\_colored("Unknown error occurred", Colors::RED);  
 }  
 }  
 }  
 };  
}  
  
#endif // BINARY\_TREE\_PLAYGROUND\_H