|  |  |
| --- | --- |
| **Российский университет транспорта (МИИТ) Институт транспортной техники и систем управления Кафедра «Управление и защита информации»** | |
| **Отчет по практическому заданию по теме «Структуры данных» по дисциплине «Системы управления базами данных»** | |
|  | Выполнил:  Студенты группы ТКИ-442  Родионов Д. О.  Чайка Е. В |
|  | Проверил:  Доцент кафедры УиЗИ к.т.н., с.н.с  Васильева М. А. |
| Москва 2023 | |

Оглавление

[1. Условие задачи 3](#_Toc153497941)

[ Слияние двух деревьев 3](#_Toc153497942)

[ Разложение дерева на два поддерева 3](#_Toc153497943)

[ Вставку элемента 3](#_Toc153497944)

[ Поиск элемента 3](#_Toc153497945)

[ Удаление элемента 3](#_Toc153497946)

[2. Текст программы на языке С++ 3](#_Toc153497947)

[2.1. ../src/Node.hpp 3](#_Toc153497951)

[2.2. ../src/DecartTree.hpp 4](#_Toc153497952)

[2.3. ../src/main.cpp 8](#_Toc153497953)

[2.4. ../src/CmakeLists.txt 8](#_Toc153497954)

[2.5. ../test/NodeTest.cpp 8](#_Toc153497955)

[2.6. ../test/BSTTest.cpp 9](#_Toc153497956)

[2.7. ../test/CmakeLists.txt 11](#_Toc153497957)

[2.8. CmakeLists.txt 11](#_Toc153497958)

[3. Результат работы программы 12](#_Toc153497959)

[4. UML диаграмма классов 13](#_Toc153497960)

1. Условие задачи

Бинарное дерево поиска.

Определить класс «Node», включающий следующую информацию:

* Данные любого типа;
* Указатели на поддеревья и родителя;

Определить класс «DecartTree», включающий следующую информацию:

* Указатель на корень дерева;

В программе предусмотреть:

* + Слияние двух деревьев
  + Разложение дерева на два поддерева
  + Вставку элемента
  + Поиск элемента
  + Удаление элемента

1. Текст программы на языке С++
   1. ../src/Node.hpp

#pragma once  
  
#include <random>  
#include <memory>  
  
namespace mit {  
 template<typename T>  
 struct Node {  
 T value;  
 int priority;  
 std::shared\_ptr<Node<T>> left, right, parent;  
  
 explicit Node():  
 left(nullptr), right(nullptr), parent(nullptr), value(0), priority(randomInt()) {}  
  
 explicit Node(T value):  
 left(nullptr), right(nullptr), parent(nullptr), value(value), priority(randomInt()) {}  
  
 explicit Node(T value, int priority):  
 left(nullptr), right(nullptr), parent(nullptr), value(value), priority(priority) {};  
  
 friend std::ostream& operator<<(std::ostream& ostream, mit::Node<T>& node) {  
 ostream << "{" << "Priority: " << node.priority << ", " << "Value: " << node.value << "}";  
 return ostream;  
 }  
  
 protected:  
 /\*\*  
 \* @brief Pseudorandom number generator for priority and/or value of node  
 \* @return Pseudorandom number  
 \*/  
 static int randomInt() {  
 auto randomDevice = std::random\_device();  
 auto mt = std::mt19937(randomDevice());  
 auto distribution = std::uniform\_int\_distribution<int>();  
  
 return distribution(mt);  
 }  
 };  
}

* 1. ../src/DecartTree.hpp

#ifndef **HEAP\_DECARTTREE\_H**#define **HEAP\_DECARTTREE\_H**#include "Node.hpp"  
#include "CComparable.hpp"  
#include <utility>  
#include <memory>  
#include <sstream>  
  
*namespace* mit {  
  
 */\*\*  
 \* In computer science, the treap and the randomized binary search tree  
 \* are two closely related forms of binary search tree data structures  
 \* that maintain a dynamic set of ordered keys and allow binary searches  
 \* among the keys. After any sequence of insertions and deletions of keys,  
 \* the shape of the tree is a random variable with the same probability  
 \* distribution as a random binary tree; in particular, with high probability  
 \* its height is proportional to the logarithm of the number of keys, so that  
 \* each search, insertion, or deletion operation takes logarithmic time to perform.  
 \*/  
 template*<mit::CComparable T> *class* DecartTree {  
 *private*:  
 *typedef* std::shared\_ptr<mit::Node<T>> Node;  
  
 Node root;  
  
 */\*\*  
 \* Recursive method for checking if element exists in the tree  
 \* @param value element to search for  
 \* @param node tree node to start checking from  
 \* @return true/false  
 \*/  
 bool* contains\_(T value, Node node) {  
 *if* (node == *nullptr*) {  
 *return false*;  
 } *else if* (node->value == value) {  
 *return true*;  
 } *else if* (node->value >= value) {  
 *return* contains\_(value, node->left);  
 } *else* {  
 *return* contains\_(value, node->right);  
 }  
 }  
  
 */\*\*  
 \* Recursive method for merging two trees into one  
 \* @param leftNode root of the first tree  
 \* @param rightNode root of the second tree  
 \* @return root of the new tree  
 \*/* Node merge\_(Node leftNode, Node rightNode) {  
 *if* (!leftNode) {  
 *return* rightNode;  
 } *else if* (!rightNode) {  
 *return* leftNode;  
 } *else* {  
 *if* (leftNode->priority > rightNode->priority) {  
 leftNode->right = merge\_(leftNode->right, rightNode);  
 *return* leftNode;  
 } *else* {  
 rightNode->left = merge\_(leftNode, rightNode->left);  
 *return* rightNode;  
 }  
 }  
 }  
  
 */\*\*  
 \* Recursive method for splitting one tree by priority key into two  
 \* @param node node to split from  
 \* @param keyForSplit priority key  
 \* @return pair of roots  
 \*/* std::pair<Node, Node> split\_(Node node, T keyForSplit) {  
 *if* (!node) {  
 *return* {*nullptr*, *nullptr*};  
 } *else* {  
 *if* (node->value <= keyForSplit) {  
 *auto* [left, right] = split\_(node->right, keyForSplit);  
 node->right = left;  
 *return* {node, right};  
 } *else* {  
 *auto* [left, right] = split\_(node->left, keyForSplit);  
 node->left = right;  
 *return* {left, node};  
 }  
 }  
 }  
  
 */\*\*  
 \* Recursive method that prints nodes into ostream  
 \* @param node node  
 \* @param ostream stream  
 \*/  
 static void* inOrderPrint\_(Node node, std::ostream& ostream) {  
 *if* (node == *nullptr*) {  
 *return*;  
 }  
 *else* {  
 ostream << \*node << ", ";  
 }  
  
 inOrderPrint\_(node->left, ostream);  
 inOrderPrint\_(node->right, ostream);  
 }  
  
 *public*:  
 DecartTree() : root(*nullptr*) {}  
 ~DecartTree() = *default*;  
  
 */\*\*  
 \* Merges this tree with another tree  
 \* @param treeForMerge another tree  
 \*/* [[maybe\_unused]] *void* merge(mit::DecartTree<T> treeForMerge) {  
 *this*->root = merge\_(*this*->root, treeForMerge.root);  
 }  
  
 */\*\*  
 \* Splits tree on two subtrees  
 \* @param keyForSplit key for split  
 \* @return pair of subtrees  
 \*/* [[maybe\_unused]] std::pair<Node, Node> split(T keyForSplit) {  
 *auto* [leftTree, rightTree] = split\_(*this*->root, keyForSplit);  
 *this*->root = leftTree;  
 *return* {leftTree, rightTree};  
 }  
  
 */\*\*  
 \* Adds element to tree with random priority  
 \* @param value element  
 \*/* [[maybe\_unused]] *void* add(T value) {  
 *auto* [leftTree, rightTree] = split\_(*this*->root, value);  
 *auto* newNode = Node(*new* mit::Node(value));  
  
 *this*->root = merge\_(leftTree, merge\_(newNode, rightTree));  
 }  
  
 */\*\*  
 \* Adds element to tree with set priority  
 \* @param value element  
 \* @param priority priority  
 \*/* [[maybe\_unused]] *void* add(T value, *int* priority) {  
 *auto* [leftTree, rightTree] = split\_(*this*->root, value);  
 Node newNode = Node(*new* mit::Node(value, priority));  
 *this*->root = merge\_(leftTree, merge\_(newNode, rightTree));  
 }  
  
 */\*\*  
 \* Adds all elements from the array to the tree with random priority  
 \* @param values elements  
 \*/* [[maybe\_unused]] *void* addAll(T values[]) {  
 *for* (*auto* data: values) {  
 *this*->add(data);  
 }  
 }  
  
 */\*\*  
 \* Checks if element is present in the tree  
 \* @param value element  
 \* @return true/false  
 \*/* [[maybe\_unused]] *bool* contains(T value) {  
 *return* contains\_(value, *this*->root);  
 }  
  
 */\*\*  
 \* Removes all elements from the tree  
 \*/* [[maybe\_unused]] *void* clear() {  
 root = *nullptr*;  
 }  
  
 */\*\*  
 \* Removes requested element from the tree  
 \* @param value element  
 \*/* [[maybe\_unused]] *void* remove(T value) {  
 *if* (*this*->contains(value)) {  
 *auto* [left, right] = *this*->split(value);  
 *this*->root = merge\_(right, merge\_(left->left, left->right));  
 }  
 }  
  
 */\*\*  
 \* Returns root of the tree  
 \* @return root  
 \*/* [[maybe\_unused]] Node getRoot() {  
 *return* root;  
 }  
  
 *friend* std::ostream& *operator*<<(std::ostream& ostream, mit::DecartTree<T>& tree) {  
 ostream << "Treap: [";  
 inOrderPrint\_(tree.root, ostream);  
 ostream << "]";  
 *return* ostream;  
 }  
 };  
  
} *// mit*#endif *//HEAP\_DECARTTREE\_H*

* 1. ../src/main.cpp

#include "DecartTree.hpp"  
  
void fillTree();  
  
int main() {  
 for (int i = 0; i < 100; i++) {  
 fillTree();  
 }  
  
 return 0;  
}  
  
void fillTree() {  
 auto tree = mit::DecartTree<int>();  
  
 for (int i = 0; i < 1000; i++) {  
 tree.add(rand());  
 }  
}

* 1. ../src/CmakeLists.txt

add\_executable(${EXECUTABLE\_NAME}  
 main.cc  
 DecartTree.hpp  
 Node.hpp  
)

* 1. ../test/NodeTest.cpp

#include <gtest/gtest.h>  
#include "../src/Node.hpp"  
  
**TEST**(NodeTest, CreateNodeSuccess) {  
 auto node = mit::Node<int>(10);  
  
 **ASSERT\_EQ**(10, node.value);  
 **ASSERT\_TRUE**(node.priority);  
 **ASSERT\_EQ**(nullptr, node.right);  
 **ASSERT\_EQ**(nullptr, node.left);  
 **ASSERT\_EQ**(nullptr, node.parent);  
}  
  
**TEST**(NodeTest, DifferentNodesHaveDifferentPriorities) {  
 auto firstNode = mit::Node<int>(10);  
 auto secondNode = mit::Node<int>(20);  
  
 **ASSERT\_NE**(firstNode.priority, secondNode.priority);  
}

* 1. ../test/BSTTest.cpp

#include <gtest/gtest.h>  
#include "../src/DecartTree.hpp"  
  
struct TestClass {  
 int state;  
  
 explicit TestClass(int state) : state(state) {}  
 friend auto operator<=>(const TestClass &lhs, const TestClass &rhs) = default;  
};  
  
**TEST**(DecartTreeTest, CreateTreeSuccess) {  
 auto tree = mit::DecartTree<int>();  
  
 auto root = tree.getRoot();  
 **ASSERT\_EQ**(root, nullptr);  
}  
  
  
**TEST**(DecartTreeTest, InsertDataWithPriorityIntoTreeSuccess) {  
 auto tree = mit::DecartTree<int>();  
  
 tree.add(10, 100);  
 tree.add(30, 120);  
 tree.add(20, 10);  
  
 auto root = tree.getRoot();  
  
 **ASSERT\_EQ**(root->value, 30);  
 **ASSERT\_EQ**(root->priority, 120);  
 **ASSERT\_EQ**(root->left->value, 10);  
 **ASSERT\_EQ**(root->left->priority, 100);  
 **ASSERT\_EQ**(root->left->right->value, 20);  
 **ASSERT\_EQ**(root->left->right->priority, 10);  
}  
  
**TEST**(DecartTreeTest, ContainsInTreeSuccess) {  
 auto tree = mit::DecartTree<int>();  
  
 tree.add(10, 100);  
 tree.add(30, 120);  
 tree.add(20, 10);  
  
 **ASSERT\_EQ**(tree.contains(10), true);  
 **ASSERT\_EQ**(tree.contains(300), false);  
}  
  
**TEST**(DecartTreeTest, InsertTestClassDataWithPriorityIntoTreeSuccess) {  
 auto tree = mit::DecartTree<TestClass>();  
  
 tree.add(TestClass(10), 100);  
 tree.add(TestClass(30), 120);  
 tree.add(TestClass(20), 10);  
  
 auto root = tree.getRoot();  
  
 **ASSERT\_EQ**(root->value.state, 30);  
 **ASSERT\_EQ**(root->priority, 120);  
 **ASSERT\_EQ**(root->left->value.state, 10);  
 **ASSERT\_EQ**(root->left->priority, 100);  
 **ASSERT\_EQ**(root->left->right->value.state, 20);  
 **ASSERT\_EQ**(root->left->right->priority, 10);  
}  
  
**TEST**(DecartTreeTest, RemoveDataSuccess) {  
 auto tree = mit::DecartTree<int>();  
  
 tree.add(10);  
 tree.add(100);  
 tree.add(20);  
 tree.add(200);  
 tree.add(25);  
 tree.add(84);  
  
 tree.remove(10);  
  
 **ASSERT\_EQ**(tree.contains(10), false);  
 **ASSERT\_EQ**(tree.contains(100), true);  
 **ASSERT\_EQ**(tree.contains(20), true);  
 **ASSERT\_EQ**(tree.contains(25), true);  
 **ASSERT\_EQ**(tree.contains(84), true);  
}  
  
**TEST**(DecartTreeTest, DeleteTreeSuccess) {  
 auto tree = mit::DecartTree<int>();  
  
 tree.add(10);  
 tree.add(100);  
 tree.add(20);  
  
 tree.clear();  
 **ASSERT\_EQ**(tree.getRoot(), nullptr);  
}  
  
int main(int argc, char \*\*argv) {  
 ::testing::InitGoogleTest(&argc, argv);  
  
 return RUN\_ALL\_TESTS();  
}

* 1. ../test/CmakeLists.txt

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(treapTest  
 DecartTreeTest.cc  
 NodeTest.cc)  
  
target\_link\_libraries(  
 treapTest  
 GTest::gtest\_main  
)  
  
include(GoogleTest)  
  
gtest\_discover\_tests(treapTest)

* 1. CmakeLists.txt

set(PROJECT\_NAME "Treap")  
set(EXECUTABLE\_NAME "treap")  
  
cmake\_minimum\_required(VERSION 3.26)  
project(${PROJECT\_NAME})  
  
set(CMAKE\_CXX\_STANDARD 23)  
  
include\_directories(.)  
add\_subdirectory(src)  
add\_subdirectory(test)

1. Результат работы программы

|  |
| --- |
|  |
| 1. Результат работы тестов для узла дерева |
|  |
| 1. Результат работы тестов для Декартова дерева |
|  |
| 1. Сводка по работе всех тестов |

|  |
| --- |
|  |
| 1. Анализ памяти с помощью Valgrind |

1. UML диаграмма классов

|  |
| --- |
|  |
| 1. UML диаграмма классов |