|  |  |
| --- | --- |
| **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  Волочинский И.О  Ефремов Д.Ю. |
|  | Проверил:  Доцент кафедры УиЗИ к.т.н., с.н.с  Васильева М. А. |
| Москва 2023 | |

Оглавление

[1. Условие задачи 3](#_Toc154511401)

[2. Текст программы на языке С++ 3](#_Toc154511402)

[2.1. Decart\_tree.cpp 3](#_Toc154511406)

[2.1. Decart\_tree.hpp. 4](#_Toc154511407)

[2.2. Node.cpp 5](#_Toc154511408)

[2.3. Node.hpp 6](#_Toc154511409)

[2.4. tests.cpp 7](#_Toc154511410)

[3.Результат работы программы 8](#_Toc154511411)

[3. UML диаграмма классов 10](#_Toc154511412)

1. Условие задачи

Реализовать декартово дерево на языке С++.

1. Текст программы на языке С++

4. 1. Decart\_tree.cpp

/\*\*

\* @file decart\_tree.hpp

\* @brief Заголовочный файл, содержащий класс DTree для декартового дерева

\*/

/\*\*

\* @class DTree

\* @brief Класс, представляющий декартово дерево

\* @details Декартово дерево - это бинарное дерево поиска, в котором каждому узлу соответствуют два значения: ключ и приоритет.

\* Узлы дерева упорядочены по ключу и выстроены по приоритету.

\*/

#include "decart\_tree.hpp"

#include <queue>

/\*\*

\* @brief Конструктор копирования

\* @param other Дерево, которое нужно скопировать

\*/

DTree::DTree(DTree &other){

Node \*tmp = new Node(other.get\_root().key);

root = tmp;

}

/\*\*

\* @brief Конструктор перемещения

\* @param other Дерево, которое нужно переместить

\*/

DTree::DTree(DTree &&other){

Node \*tmp = other.get\_root();

std::swap(root, tmp);

delete tmp;

}

/\*\*

\* @brief Конструктор по умолчанию

\*/

DTree::DTree(){

root = new Node(0);

}

/\*\*

\* @brief Конструктор с параметром

\* @param x Ключ корневого узла

\*/

DTree::DTree(int x){

root = new Node(x);

}

/\*\*

\* @brief Деструктор

\*/

~DTree::DTree(){

delete root;

}

/\*\*

\* @brief Вставить элемент в дерево

\* @param x Ключ элемента

\*/

void DTree::insert (int x) {

Pair q = Nodes::split(root, x);

Node \*t = new Node(x);

root = Nodes::merge(q.first, Nodes::merge(t, q.second));

}

/\*\*

\* @brief Удалить элемент из дерева

\* @param x Ключ элемента

\*/

void DTree::remove (int x) {

Pair q = Nodes::split(root, x);

Node \*deleted = Nodes::merge(q.first->l, q.first->r);

root = Nodes::merge(deleted, q.second);

}

/\*\*

\* @brief Проверить, есть ли элемент в дереве

\* @param x Ключ элемента

\* @return true, если элемент найден, иначе false

\*/

bool DTree::find(int x){

return Nodes::find(root, x);

}

/\*\*

\* @brief Получить вектор со всеми элементами дерева в порядке обхода в ширину

\* @return Вектор с ключами элементов

\*/

std::vector<int> DTree::get\_vector(){

std::vector<int> v;

std::queue<Node\*> q;

q.push(root);

while(!q.empty()){

Node\* n = q.front();

q.pop();

q.push(n->l);

q.push(n->r);

v.push\_back(n->key);

}

return v;

}

* 1. Decart\_tree.hpp.

#include "node.hpp"

#include <vector>

/\*\*

\* @brief Класс DTree представляет собой бинарное дерево поиска.

\* Каждый узел дерева содержит целочисленное значение, и дерево построено таким образом,

\* что для каждого узла все значения в левом поддереве меньше, чем значение узла,

\* а все значения в правом поддереве больше.

\*/

class DTree

{

Node\* root; /\*\*< Указатель на корневой узел дерева. \*/

public:

/\*\*

\* @brief Конструктор для создания объекта класса DTree.

\* @param x Значение корневого узла.

\*/

DTree(int x) noexcept;

/\*\*

\* @brief Конструктор по умолчанию для создания пустого дерева.

\*/

DTree() noexcept;

/\*\*

\* @brief Конструктор копирования.

\* @param other Дерево, которое необходимо скопировать.

\*/

DTree(DTree &other) noexcept;

/\*\*

\* @brief Конструктор перемещения.

\* @param other Дерево, которое необходимо переместить.

\*/

DTree(DTree &&other) noexcept;

/\*\*

\* @brief Деструктор, освобождает выделенную память.

\*/

~DTree();

/\*\*

\* @brief Вставка нового значения в дерево.

\* @param x Значение, которое необходимо вставить.

\*/

void insert(int x);

/\*\*

\* @brief Удаление значения из дерева.

\* @param x Значение, которое необходимо удалить.

\*/

void remove(int x);

/\*\*

\* @brief Поиск значения в дереве.

\* @param x Значение, которое необходимо найти.

\* @return true, если значение найдено; false в противном случае.

\*/

bool find(int x);

/\*\*

\* @brief Получение указателя на корневой узел дерева.

\* @return Указатель на корневой узел.

\*/

Node\* get\_root() { return root; }

/\*\*

\* @brief Получение вектора значений в порядке обхода дерева.

\* @return Вектор значений.

\*/

std::vector<int> ger\_vector();

// Операторы присваивания запрещены

DTree& operator+=(const DTree&) = delete;

DTree& operator==(const DTree&) = delete;

DTree& operator-=(const DTree&) = delete;

DTree& operator\*=(const DTree&) = delete;

DTree& operator/=(const DTree&) = delete;

}

* 1. Node.cpp

#include "node.hpp"

Node::~Node(){

delete l;

delete r;

}

/\*\*

\*@brief Функция merge объединяет два поддерева l и r в одно дерево.

\*@param l Указатель на корень первого поддерева.

\*@param r Указатель на корень второго поддерева.

\*@return Указатель на корень объединенного дерева.

\*/

Node\* Nodes::merge (Node \*l, Node \*r) {

if (!l) return r;

if (!r) return l;

if (l->prior > r->prior) {

l->r = merge(l->r, r);

return l;

}

else {

r->l = merge(l, r->l);

return r;

}

}

/\*\*

\*@brief Функция split разделяет дерево с корнем p на два поддерева по ключу x.

\*@param p Указатель на корень дерева, которое нужно разделить.

\*@param x Ключ, по которому происходит разделение.

\*@return Пара указателей на корни получившихся поддеревьев. Первый элемент - левое поддерево, второй - правое поддерево.

\*/

Pair Nodes::split (Node \*p, int x) {

if (!p) return {0, 0};

if (p->key <= x) {

Pair q = split(p->r, x);

p->r = q.first;

return {p, q.second};

}

else {

Pair q = split(p->l, x);

p->l = q.second;

return {q.first, p};

}

}

/\*\*

\*@brief Функция find проверяет, содержится ли элемент с ключом x в дереве с корнем n.

\*@param n Указатель на корень дерева.

\*@param x Ключ, который нужно найти.

\*@return true, если элемент найден, false - в противном случае.

\*/

bool Nodes::find(Node \*n, int x) {

if (!n)

return false;

if (n->key == x)

return true;

else if (n->key > x)

return find(n->l, x);

else

return find(n->r, x);

}

* 1. Node.hpp

#include <iostream>

#ifndef DECART\_NODE\_H

#define DECART\_NODE\_H

struct Node;

typedef std::pair<Node\*, Node\*> Pair;

/\*\*

\* @brief Структура Node представляет узел для декартова дерева.

\* Каждый узел содержит целочисленное значение (key), приоритет (prior),

\* указатели на левого (l) и правого (r) потомка.

\*/

struct Node {

int key; /\*\*< Значение узла. \*/

int prior; /\*\*< Приоритет узла. \*/

Node \*l = 0, \*r = 0; /\*\*< Указатели на левого и правого потомка. \*/

/\*\*

\* @brief Конструктор для создания объекта класса Node.

\* @param \_key Значение узла и его приоритет.

\*/

Node (int \_key): key(\_key), prior(\_key){}

/\*\*

\* @brief Деструктор, освобождает выделенную память.

\*/

~Node();

};

/\*\*

\* @brief Пространство имен Nodes содержит функции для работы с узлами декартова дерева.

\*/

namespace Nodes {

/\*\*

\* @brief Поиск значения в декартовом дереве.

\* @param n Указатель на корень дерева.

\* @param x Значение, которое необходимо найти.

\* @return true, если значение найдено; false в противном случае.

\*/

bool find(Node \*n, int x);

/\*\*

\* @brief Слияние двух декартовых деревьев.

\* @param l Указатель на корень первого дерева.

\* @param r Указатель на корень второго дерева.

\* @return Указатель на корень нового дерева, полученного слиянием.

\*/

Node\* merge (Node \*l, Node \*r);

/\*\*

\* @brief Разделение декартова дерева по значению.

\* @param p Указатель на корень дерева.

\* @param x Значение, по которому производится разделение.

\* @return Пара указателей на корень двух получившихся деревьев.

\*/

Pair split (Node \*p, int x);

}

#endif

* 1. tests.cpp

#include "decart\_tree.hpp"

#include <gtest/gtest.h>

#include <iostream>

TEST (module\_name, test\_name) {

DTree t = DTree(1);

t.insert(121);

// Google Test will also provide macros for assertions.

ASSERT\_EQ(t.find(121), true);

}

TEST (module\_name, test\_name) {

DTree t = DTree(1);

t.insert(121);

t.remove(121);

// Google Test will also provide macros for assertions.

ASSERT\_EQ(t.find(121), false);

}

TEST (module\_name, test\_name) {

Node t = Node(1);

// Google Test will also provide macros for assertions.

ASSERT\_EQ(Nodes::find(1), true);

}

TEST (module\_name, test\_name) {

Node t = Node(1);

// Google Test will also provide macros for assertions.

ASSERT\_EQ(t.key, 1);

}

TEST (module\_name, test\_name) {

Node t = Node(1);

// Google Test will also provide macros for assertions.

ASSERT\_EQ(t.prior, 1);

}

int main(int argc, char\*\* argv){

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

3.Результат работы программы

|  |
| --- |
| Рисунок 1. Результат работы программы. |

1. UML диаграмма классов

|  |
| --- |
|  |
| Рисунок 2. Uml диаграмма классов |