ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«Российский университет транспорта»  
(РУТ (МИИТ))

Институт транспортной техники и систем управления

Кафедра «Управление и защите информации»

отчёт

о Лабораторной работе №2

По дисциплине «СУБД»

Выполнил: ст. гр. ТКИ-441   
Степанов Д. Е.  
Проверил: к. т. н., доцент   
Васильева М. А.

Москва 2023

Оглавление

[Постановка задачи 3](#_Toc154610484)

[Теоретические сведения 4](#_Toc154610485)

[Диаграмма классов 6](#_Toc154610486)

[Код программы 7](#_Toc154610487)

[1 node.hpp 7](#_Toc154610488)

[2 treap.hpp 10](#_Toc154610489)

[3 iterator.hpp 18](#_Toc154610490)

[4 main.cpp 20](#_Toc154610491)

[Результаты автоматичекого тестирования 22](#_Toc154610492)

[1 Результат тестирования класса treap::Node 22](#_Toc154610493)

[2 Результат тестирования класса treap::Iterator 22](#_Toc154610494)

[3 Результат тестирования класса rut::Treap 22](#_Toc154610495)

[Вывод 23](#_Toc154610496)

Постановка задачи

Разработать структуру данных на языке программирования С++ в ООП парадигме (Декартово дерево). Реализовать библиотеку классов, обеспечить выполнение операций CRUD (create, read, update, delete) для любого типа данных (template). Ни один метод не должен зависеть от консоли или файла, используйте представление в строку.

Написать итератор для работы с коллекцией (разработанной структурой данных).

Все методы покрыть тестами (модульное тестирование).

Задание выполнять в системе контроля версий. Проверка задания осуществляется посредством pull request в веб-хостинге GitHub (collaborator MarinaPaley). В репозитории необходимо иметь рабочий код, так, чтобы для проверки преподаватель смог склонировать репозиторий и запустить тесты, не прибегая к помощи специальных инструментов.

Теоретические сведения

Декартово дерево (также известное как «ПиВо» – от Пирамида и дереВо, «КуРево» – от Куча и деРево или «ДерАмида» от Дерево и пирАмида) – это структура данных, объединяющая в себе черты двоичного дерева поиска и двоичной кучи (heap). Каждый узел дерева представляет собой пару значений: ключ и приоритет. Ключи узлов удовлетворяют свойству двоичного дерева поиска, то есть для каждого узла все значения в его левом поддереве меньше его ключа, а в правом – больше. При этом приоритеты узлов подчиняются свойству кучи, где приоритет каждого узла не меньше приоритетов его потомков.

Операции на декартовом дереве включают в себя операции вставки, удаления, поиска элемента по ключу, а также разделения и слияния деревьев.

Разделение:

Разделение позволяет разбить дерево на два поддерева по заданному ключу K, причем левый результат разбиения содержит все узлы с   
ключами <= K, а правый результат – все остальные узлы.

Слияние:

Слияние позволяет соединить два поддерева таким образом, чтобы не нарушить ограничения, накладываемые на порядок узлов в результирующем декартовом дереве.

Вставка:

При вставке нового элемента с ключом K и приоритетом P, возможны два алгоритма:

1. Новый элемент добавляется как новый лист в соответствующее место дерева по ключу. Затем, если приоритет нового узла больше приоритета его родителя, выполняется операция поворота вверх, чтобы сохранить свойство кучи;
2. По ключу нового элемента К происходит разбиение дерева, с дальнейшем слиянием нового элемента с последующим слиянием нового узла с правой частью результата. А затем слиянием левой части результата с результатом, полученным в результате предыдущей операции.

Удаление:

При удалении узла с ключом K, производится поиск поддерева с корнем, у которого ключ соответствует ключу К. Далее вместо найденного поддерева вставляется результат слияния его левого и правого поддеревьев.

Поиск:

Поиск элемента по ключу осуществляется стандартным образом, сравнивая ключ поочередно с корнем и переходя в левое или правое поддерево в зависимости от результата сравнения.

Декартово дерево находит применение в задачах, где требуется эффективная реализация операций вставки, удаления и поиска элементов, таких как управление структурами данных в параллельных алгоритмах, а также в решении некоторых задач геометрии и оптимизации.

Диаграмма классов

На рисунке 1 представлена UML-диаграмма классов, спроектированных в результате выполнения лабораторной работы.

Изображение выглядит как текст, снимок экрана, Шрифт, Параллельный

Автоматически созданное описание

Рисунок 1 – UML-диаграмма классов

Код программы

1. node.hpp

#pragma once

#include <random>

#include <memory>

namespace treap {

/\*\*

\* @brief Treap Node template implementation.

\*

\* @tparam C should implement Comparable Interface (unfortunately there are no such thing in C++)

\* @tparam T just data of any type for storing in Node

\*/

template <typename C, typename T> class Node;

/\*\*

\* @brief std::shared\_ptr<treap::Node<C, T>> type alias.

\*

\* @tparam C should implement Comparable Interface (unfortunately there are no such thing in C++)

\* @tparam T just data of any type for storing in Node

\*/

template <typename C, typename T> using NodePtr = std::shared\_ptr<Node<C,T>>;

template <typename C, typename T>

class Node : public std::enable\_shared\_from\_this<Node<C, T>> {

private:

C key;

C priority;

T data;

public:

/\*\*

\* @brief Construct a new Node object with default attributes:

\*

\* @attention This constructor randomly generates node key & priority. Data has default T{} value.

\*/

Node();

/\*\*

\* @brief Construct a new treap Node object from passed params

\*

\* @param key Node key

\* @param priority node priority

\* @param data stored in Node data

\*/

Node(C key, C priority, T data);

/\*\*

\* @brief Disabled copy constructor

\*/

Node(const Node& other) = delete;

/\*\*

\* @brief Default move constructor

\*/

Node(Node&& other) = default;

/\*\*

\* @brief Disabled assignment operator

\*/

Node& operator=(const Node& other) noexcept = delete;

/\*\*

\* @brief Default move assignment operator

\*/

Node& operator=(Node&& other) noexcept = default;

/\*\*

\* @brief Default Node destructor

\*/

~Node() = default;

C getKey() const noexcept { return this->key; };

C getPriority() const noexcept { return this->priority; };

T getData() const noexcept { return this->data; };

NodePtr<C, T> ptr() { return this->shared\_from\_this(); };

bool hasLessPriorityThan(const Node<C, T> &other) const;

bool hasGreaterPriorityThan(const Node<C, T> &other) const;

/\*\*

\* @brief Spaceship operator overload (with by-key comparation)

\*

\* @param lha The node for compare

\* @param rha The node for compare

\*

\* @return 1 in case of lha < rha, -1 in case of lha > rha, 0 in case of lha == rha

\*/

friend auto operator<=>(const Node& lha, const Node& rha) {

if (std::less<C>{}(lha.key, rha.key)) return -1;

if (std::greater<C>{}(lha.key, rha.key)) return 1;

return 0;

}

/\*\*

\* @brief Equity operator overload

\*

\* @param lha The node for equity check

\* @param rha The node for equity check

\* @return boolean value

\*/

friend bool operator==(const Node& lha, const Node& rha) {

return (lha <=> rha) == 0;

}

};

template <typename C, typename T>

Node<C, T>::Node() {

std::random\_device randomizer;

std::mt19937 gen{randomizer()};

using eligible\_dist = std::conditional\_t<

std::is\_integral\_v<C>,

std::uniform\_int\_distribution<C>,

std::uniform\_real\_distribution<C>

>;

eligible\_dist dis{std::numeric\_limits<C>::min(), std::numeric\_limits<C>::max()};

this->key = dis(gen);

this->priority = dis(gen);

}

template <typename C, typename T>

Node<C, T>::Node(C key, C priority, T data) : key{key}, priority{priority}, data{data} {};

template <typename C, typename T>

bool Node<C, T>::hasLessPriorityThan(const Node<C, T> &other) const {

return std::less<C>{}(this->getPriority(), other.getPriority());

};

template <typename C, typename T>

bool Node<C, T>::hasGreaterPriorityThan(const Node<C, T> &other) const {

return std::greater<C>{}(this->getPriority(), other.getPriority());

};

}

1. treap.hpp

#pragma once

#include <vector>

#include "treap/node.hpp"

#include "treap/iterator.hpp"

namespace rut {

/\*\*

\* @brief Treap template implementation.

\*

\* @tparam C should implement Comparable Interface (unfortunately there are no such thing in C++).

\* @tparam T just data of any type for storing in Treap nodes.

\*/

template<typename C, typename T> class Treap;

/\*\*

\* @brief std::shared\_ptr<rut::Treap<C, T>> type alias.

\*

\* @tparam C should implement Comparable Interface (unfortunately there are no such thing in C++).

\* @tparam T just data of any type for storing in Treap nodes.

\*/

template<typename C, typename T> using TreapPtr = std::shared\_ptr<rut::Treap<C, T>>;

/\*\*

\* @brief Sub-treaps of current treap root.

\*

\* @tparam C C should implement Comparable Interface (unfortunately there are no such thing in C++).

\* @tparam T T just data of any type for storing in Treap nodes.

\*

\* @param first - left sub-treap of current treap.

\* @param second - right sub-treap of current treap.

\*/

template <typename C, typename T> using SubTreaps = std::pair<TreapPtr<C, T>, TreapPtr<C, T>>;

template<typename C, typename T>

class Treap : public std::enable\_shared\_from\_this<Treap<C, T>> {

private:

treap::NodePtr<C, T> root;

SubTreaps<C, T> subTreaps;

void setRoot(treap::NodePtr<C, T> root) { this->root = root; };

void setLeftSubTreap(TreapPtr<C, T> treap) { this->subTreaps.first = treap; };

void setRightSubTreap(TreapPtr<C, T> treap) { this->subTreaps.second = treap; };

/\*\*

\* @brief Same as public Split method, but more eligible fore some cases.

\*

\* @param treap

\* @param separator

\* @return SubTreaps<C, T>

\*/

friend SubTreaps<C, T> split(TreapPtr<C, T> treap, C separator) {

if (treap == nullptr || treap->getRoot() == nullptr) return {nullptr, nullptr};

if (treap->getRoot()->getKey() <= separator) {

auto result = split(treap->rightSubTreap(), separator);

treap->setRightSubTreap(result.first);

return {treap, result.second};

}

else {

auto result = split(treap->leftSubTreap(), separator);

treap->setLeftSubTreap(result.second);

return {result.first, treap};

}

};

/\*\*

\* @brief Same as public Merge method, but more eligible for some cases.

\*

\* @param l

\* @param r

\* @return TreapPtr<C, T>

\*/

friend TreapPtr<C, T> merge(TreapPtr<C, T> l, TreapPtr<C, T> r) {

if (l == nullptr || l->getRoot() == nullptr) return r;

if (r == nullptr || r->getRoot() == nullptr) return l;

if (l->getRoot()->hasGreaterPriorityThan(\*r->getRoot())) {

auto newRight = merge(l->rightSubTreap(), r);

l->setRightSubTreap(newRight);

return l;

}

else {

auto newLeft = merge(l, r->leftSubTreap());

r->setLeftSubTreap(newLeft);

return r;

}

};

public:

/\*\*

\* @brief Construct a new empty Treap object with root node

\*

\* @attention Root node key & priority are randomly generated.

\*/

Treap();

/\*\*

\* @brief Construct a new Treap object as copy of the given Treap.

\*

\* @param other

\*/

Treap(const Treap& other);

treap::NodePtr<C, T> getRoot() const noexcept { return root; };

TreapPtr<C, T> leftSubTreap() const noexcept { return subTreaps.first; };

TreapPtr<C, T> rightSubTreap() const noexcept { return subTreaps.second; };

/\*\*

\* @brief Returns shared pointer of this object.

\* @attention bad\_weak\_pointer error will be raised if there are no existing shared pointers.

\*

\* @return TreapPtr<C, T>

\*/

TreapPtr<C, T> ptr() { return this->shared\_from\_this(); };

/\*\*

\* @brief Separate treap in 2 sub-treaps.

\*

\* @return Pair of sub-treaps which meets condition: 1st sub-treap nodes keys <= passed node key <= 2nd sub-treap nodes keys.

\*/

SubTreaps<C, T> Split(C key) const;

/\*\*

\* @brief Merge sub treap to current instance.

\*

\* @param other Subtreap to merge.

\* @return Result of treap merging.

\*/

TreapPtr<C, T> Merge(TreapPtr<C, T> other);

/\*\*

\* @brief Creates new node in treap.

\*

\* @param key Key of stored node.

\* @param priority Priority of stored node.

\* @param data Data of stored node.

\*/

void Insert(const C key, const C priority, const T data);

/\*\*

\* @brief Find node in treap by given key.

\*

\* @param key

\* @return treap::NodePtr<C, T> - found node (or nullptr)

\*/

treap::NodePtr<C, T> Find(const C key) const;

/\*\*

\* @brief Remove node from treap by given key.

\*

\* @param key

\* @return true if Node was removed,

\* @return false otherwise.

\*/

bool Remove(const C key);

/\*\*

\* @brief Returns vector of nodes in ordered traversal (by keys).

\*

\* @return std::vector<treap::NodePtr<C, T>> - Vector of tree nodes.

\*/

std::vector<treap::NodePtr<C, T>> inOrderedVector() const noexcept;

/\*\*

\* @brief Returns iterator for first node in InOrdered tree

\*

\* @return treap::Iterator<C, T>

\*/

treap::Iterator<C, T> begin() const;

/\*\*

\* @brief Returns iterator for last node in InOrdered tree

\*

\* @return treap::Iterator<C, T>

\*/

treap::Iterator<C, T> end() const;

friend bool operator==(const rut::Treap<C, T>& lha, const rut::Treap<C, T>& rha) {

auto lhaVector = lha.inOrderedVector();

auto rhaVector = rha.inOrderedVector();

if (lhaVector.size() != rhaVector.size()) return false;

bool result = true;

for (size\_t i = 0; i < lhaVector.size(); i++) {

result &= (lhaVector[i] == rhaVector[i]);

}

return result;

};

friend bool operator!=(rut::Treap<C, T>& lha, rut::Treap<C, T>& rha) { return !(lha == rha); };

};

template<typename C, typename T>

Treap<C, T>::Treap() {

setRoot(nullptr);

setLeftSubTreap(nullptr);

setRightSubTreap(nullptr);

};

template<typename C, typename T>

Treap<C, T>::Treap(const Treap<C, T>& other) {

setRoot(other.getRoot());

setLeftSubTreap(other.leftSubTreap());

setRightSubTreap(other.rightSubTreap());

}

template <typename C, typename T>

SubTreaps<C, T> Treap<C, T>::Split(C separator) const {

// IDK how to disallow instance mutation in other way, sorry :<

TreapPtr<C, T> thisCopy = std::make\_shared<Treap<C, T>>(\*this);

return split(thisCopy, separator);

}

template <typename C, typename T>

TreapPtr<C, T> Treap<C, T>::Merge(TreapPtr<C, T> other) {

return merge(ptr(), other);

};

template <typename C, typename T>

void Treap<C, T>::Insert(const C key, const C priority, const T data) {

auto node = std::make\_shared<treap::Node<C, T>>(key, priority, data);

if (root == nullptr) return setRoot(node);

auto result = std::make\_shared<Treap<C, T>>();

auto thisCopy = std::make\_shared<Treap<C, T>>(\*this);

auto [left, right] = split(thisCopy, key);

result->setRoot(node);

auto newRight = merge(result, right);

result = merge(left, newRight);

setRoot(result->getRoot());

setLeftSubTreap(result->leftSubTreap());

setRightSubTreap(result->rightSubTreap());

}

template <typename C, typename T>

treap::NodePtr<C, T> Treap<C, T>::Find(const C key) const {

if (getRoot() == nullptr) return nullptr;

if (getRoot()->getKey() == key) return getRoot();

if (std::greater{}(getRoot()->getKey(), key)) {

if (!leftSubTreap()) return nullptr;

return leftSubTreap()->Find(key);

} else {

if (!rightSubTreap()) return nullptr;

return rightSubTreap()->Find(key);

}

}

template <typename C, typename T>

bool Treap<C, T>::Remove(const C key) {

if (Find(key) == nullptr) return false;

if (getRoot() == nullptr) {

return false;

}

if (getRoot()->getKey() == key) {

auto replace = merge(leftSubTreap(), rightSubTreap());

if (replace == nullptr) {

return false;

} else {

setRoot(replace->getRoot());

setLeftSubTreap(replace->leftSubTreap());

setRightSubTreap(replace->rightSubTreap());

return true;

}

}

if (std::greater{}(getRoot()->getKey(), key)) {

if (!leftSubTreap()) return false;

if (!leftSubTreap()->Remove(key)) setLeftSubTreap(nullptr);;

return true;

} else {

if (!rightSubTreap()) return false;

if (!rightSubTreap()->Remove(key)) setRightSubTreap(nullptr);

return true;

}

}

template <typename C, typename T>

std::vector<treap::NodePtr<C, T>> rut::Treap<C, T>::inOrderedVector() const noexcept {

std::vector<treap::NodePtr<C, T>> result = std::vector<treap::NodePtr<C, T>>();

if (begin() == end()) return result;

for (treap::Node<C, T>& currentNode : \*this) {

result.push\_back(currentNode.ptr());

}

return result;

}

template <typename C, typename T>

treap::Iterator<C, T> rut::Treap<C, T>::begin() const {

auto thisRef = \*this; // for some reason treap::Iterator<C, T>(\*this) raises error

return treap::Iterator<C, T>(thisRef);

}

template <typename C, typename T>

treap::Iterator<C, T> rut::Treap<C, T>::end() const {

return treap::Iterator<C, T>(nullptr);

}

}

1. iterator.hpp

#pragma once

#include <cstddef>

#include <stack>

#include <optional>

#include <memory>

namespace rut { template<typename C, typename T> class Treap; };

namespace treap {

/\*\*

\* @brief Treap template bidirectional iterator implementation.

\*

\* @tparam C should implement Comparable Interface (unfortunately there are no such thing in C++).

\* @tparam T just data of any type for storing in Treap nodes.

\*/

template <typename C, typename T> class Iterator {

public:

using iterator\_category = std::bidirectional\_iterator\_tag;

using difference\_type = std::ptrdiff\_t;

using value\_type = treap::Node<C, T>;

using pointer = std::shared\_ptr<rut::Treap<C, T>>;

using reference = rut::Treap<C, T>&;

Iterator(reference r) {

auto left = std::make\_shared<rut::Treap<C, T>>(r);

// If we got blank root in beginning then whole treap is empty.

if (r.getRoot() == nullptr) left = nullptr;

while(left != nullptr) {

iter.push(left);

left = left->leftSubTreap();

}

if (iter.size() == 0) {

current = nullptr;

} else {

current = iter.top();

iter.pop();

}

}

Iterator(const pointer& p) {

auto left = p;

while(left != nullptr) {

iter.push(left);

left = left->leftSubTreap();

}

if (iter.size() == 0) {

current = nullptr;

} else {

current = iter.top();

iter.pop();

}

}

NodePtr<C, T> operator->() const { return (current != nullptr) ? current->getRoot() : nullptr ; };

Node<C, T>& operator\*() const { if (current != nullptr) return \*current->getRoot(); };

Iterator<C, T>& operator++() {

if (current == nullptr) return \*this;

if (current->rightSubTreap() != nullptr) {

iter.push(current->rightSubTreap());

if (current->rightSubTreap()->leftSubTreap() != nullptr) {

iter.push(current->rightSubTreap()->leftSubTreap());

}

}

if (iter.size() == 0) {

current = nullptr;

return \*this;

}

current = iter.top();

iter.pop();

return \*this;

}

Iterator<C, T> operator++(int) {

auto unchanged = \*this;

++(\*this);

return unchanged;

}

bool operator==(Iterator<C, T> const &other) const { return current == other.current; }

bool operator!=(Iterator<C, T> const &other) const { return current != other.current; }

private:

pointer current;

std::stack<pointer> iter;

};

}

1. main.cpp

#include <stdint.h>

#include <string>

#include <iostream>

#include "src/treap.hpp"

int main() {

auto node1 = std::make\_shared<treap::Node<uint32\_t, std::string>>(1, 1, std::string{"Hello"});

auto node2 = std::make\_shared<treap::Node<uint32\_t, std::string>>(2, 3, std::string{"world"});

auto node3 = std::make\_shared<treap::Node<uint32\_t, std::string>>(3, 2, std::string{"!"});

auto empty1 = rut::Treap<uint32\_t, std::string>();

auto empty2 = rut::Treap<uint32\_t, std::string>(empty1);

std::cout << (empty1 == empty2) << std::endl;

auto treapPtr = std::make\_shared<rut::Treap<uint32\_t, std::string>>();

treapPtr->Insert(3, 2, std::string{"world"});

treapPtr->Insert(2, 1, std::string{"Hello"});

treapPtr->Insert(3, 2, std::string{"!"});

for (auto& current : treapPtr->inOrderedVector()) {

std::cout << current->getData() << " ";

}

std::cout << std::endl;

}

Результаты автоматичекого тестирования

1. Результат тестирования класса treap::Node

[----------] 7 tests from Node  
[ RUN ] Node.DefaultConstructorWithIntegralType  
[ OK ] Node.DefaultConstructorWithIntegralType (0 ms)  
[ RUN ] Node.DefaultConstructorWithNotIntegralType  
[ OK ] Node.DefaultConstructorWithNotIntegralType (0 ms)  
[ RUN ] Node.ParametrizedConstructor  
[ OK ] Node.ParametrizedConstructor (0 ms)  
[ RUN ] Node.SpaceshipOperator  
[ OK ] Node.SpaceshipOperator (0 ms)  
[ RUN ] Node.EquityOperator  
[ OK ] Node.EquityOperator (0 ms)  
[ RUN ] Node.hasLessPriorityThan  
[ OK ] Node.hasLessPriorityThan (0 ms)  
[ RUN ] Node.hasGreaterPriorityThan  
[ OK ] Node.hasGreaterPriorityThan (0 ms)  
[----------] 7 tests from Node (0 ms total)

1. Результат тестирования класса treap::Iterator
2. Результат тестирования класса rut::Treap

Вывод

В результате выполнения лабораторной работы была изучена такая структура данных как Декартово дерево, изучены некоторые аспекты стандарта C++20, а также приобретены навыки автоматического тестирования с использованием библиотеки gtest.