Министерство транспорта РФ

Федеральное государственное автономное

образовательное учреждение высшего образования

«Российский университет транспорта» (МИИТ)

Отчет по лабораторной работе №2

по дисциплине

«Системы управления базами данных»

Выполнил:

студент группы ТКИ-441,

Кузнецов К. К.

Проверил:

к.т.н., доц. Васильева М. А.

Москва – 2023 г.

Оглавление

[1 Задание 3](#_Toc154613261)

[2 Рекомендации 4](#_Toc154613262)

[3 Выбранная структура данных 5](#_Toc154613263)

[3.1 Принцип работы кучи: 5](#_Toc154613264)

[3.2 Описание решения 5](#_Toc154613265)

[3.3 UML-диаграмма классов 6](#_Toc154613266)

[4 Листинг программы 7](#_Toc154613267)

[4.1 heap.h 7](#_Toc154613268)

[4.2 heap.cpp 12](#_Toc154613269)

[4.3 heap\_test.cpp 15](#_Toc154613270)

[5 Результаты модульного тестирования 20](#_Toc154613271)

[6 Результаты Review 21](#_Toc154613272)

[7 Вывод 22](#_Toc154613273)

1. Задание

Разработать структуру данных на языке программирования С++ в ООП парадигме. Реализовать библиотеку классов, обеспечить выполнение операций CRUD (create, read, update, delete) для любого типа данных (template). Переопределить операторы сдвига влево и вправо для возможности чтения и записи из любого потока (консоль, строка, файл). Ни один метод не должен зависеть от консоли или файла, используйте представление в строку. Написать итератор для работы с коллекцией (разработанной структурой данных). По возможности использовать умные указатели (smart pointer), заместо «сырых» (raw). Использование стандартных структур данных возможно в качестве вспомогательных, например, при обходе дерева можно использовать std::vector. Для избежания недоразумений, лучше обсудить использование с преподавателем индивидуально.

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 (если разрабатываете в MS Visual Studio, то репозиторий должен содержать решение и минимум три проекта: запускаемый, библиотеку классов, тесты, если разрабатываете в другой IDE, то необходимо приложить cmake файлы для запускаемого и тестируемого проектов). Использовать для решения последнюю версию стандарта языка С++ (20).

Задание считается зачтенным, если получен approve. Варианты 1–7 для максимальной оценки «удовлетворительно» на экзамене.

1. Рекомендации

Начинать работу над структурой данных лучше с API для конкретного типа данных, например int. После подтверждения преподавателем минимального набора функционала, необходимо реализовать все объявленные методы и отладить на них тесты. Разработку шаблонного класса выбранной коллекции удобно вести, применяя TDD (test driven development) [3], так как необходимые классы для целого типа данных вами уже отлажены. Необходимо будет «скопировать» логику тестов для любого ссылочного типа данных, например строки, и уже имея готовую тестовую базу, разработать библиотеку на шаблонах.

Все члены классов должны иметь документацию в стиле doxygen.

Необходимо предоставить отчет о работе, содержащий UML диаграмму библиотеки классов.

1. Выбранная структура данных

Вариант №18 – **Куча (heap)**

* 1. Принцип работы кучи:

Куча — это двоичное дерево, удовлетворяющее основное свойство кучи. В максимальной куче (max heap) для каждого узла значение его ключа больше или равно значению ключей его потомков. В минимальной куче (min heap) для каждого узла значение его ключа меньше или равно значению ключей его потомков.

Двоичное дерево удобно представлять в виде массива, где элементы уровней дерева хранятся последовательно. Например, корень дерева находится в индексе 0, его потомки — в индексах 1 и 2, потомки узла в индексах 3 и 4, и так далее.

* 1. Описание решения

В представленной реализации получилась максимальная куча (max heap). В максимальной куче для каждого узла значение его ключа больше или равно значению ключей его потомков. Таким образом, в корне кучи всегда содержится максимальный элемент.

Это отражается в реализации методов heapifyUp() и heapifyDown(), где сравнения проводятся с использованием оператора >. Когда выполняется восстановление свойств кучи, если добавленный элемент (или корень при удалении) меньше его потомков, происходит обмен местами, чтобы восстановить порядок максимальной кучи.

Элементы реализованной Кучи хранятся в векторе data в порядке, обеспечивающем представление двоичного дерева в виде массива. Элементы добавляются в конец вектора, и при этом поддерживаются свойства кучи, а именно:

* Корень дерева находится в начале вектора (data[0]).
* Для любого узла с индексом i его левый потомок находится в индексе 2 \* i + 1, а правый потомок - в индексе 2 \* i + 2.

Такой порядок удобен для эффективного выполнения операций вставки и удаления в куче. Это обеспечивает компактное представление дерева в массиве и удобный доступ к элементам.

* 1. UML-диаграмма классов

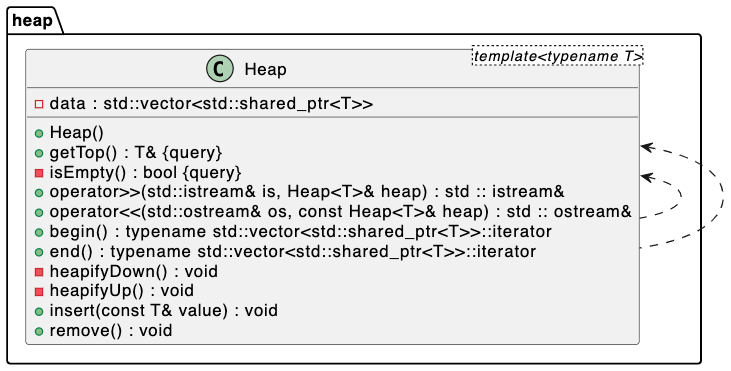


Рисунок 1 – UML-диаграмма класса Heap

1. Листинг программы
   1. heap.h

#include <iostream>

#include <vector>

#include <memory>

namespace heap

{

/\*\*

\* @brief Heap Class template implementation.

\*

\* The Heap class is a template class that represents a binary heap data structure.

\* It provides methods to insert, remove, and access elements in the heap.

\* The heap is implemented using a vector of shared pointers to elements.

\*

\* @tparam T Type of elements stored in the heap.

\*/

template <typename T>

class Heap

{

private:

std::vector<std::shared\_ptr<T>> data; /\*\*< Vector of shared pointers to store elements in the heap. \*/

/\*\*

\* @brief Helper method to maintain the heap property by moving the last element up.

\*

\* This method is called after inserting a new element to maintain the heap property.

\*/

void heapifyUp();

/\*\*

\* @brief Helper method to maintain the heap property by moving the root element down.

\*

\* This method is called after removing the root element to maintain the heap property.

\*/

void heapifyDown();

/\*\*

\* @brief Check if the heap is empty.

\*

\* @return true if the heap is empty, false otherwise.

\*/

bool isEmpty() const;

public:

/\*\*

\* @brief Default constructor for the Heap class.

\*

\* Initializes an empty heap.

\*/

Heap();

/\*\*

\* @brief Insert a new element into the heap.

\*

\* @param value The value of the element to be inserted.

\*/

void insert(const T &value);

/\*\*

\* @brief Remove the top element from the heap.

\*

\* Removes the element with the highest priority (root) from the heap.

\*/

void remove();

/\*\*

\* @brief Get the value of the top element in the heap.

\*

\* @return A constant reference to the value of the top element.

\*/

const T &getTop() const;

/\*\*

\* @brief Overloaded output stream operator to print the elements of the heap.

\*

\* @param os The output stream.

\* @param heap The Heap object to be printed.

\* @return A reference to the output stream.

\*/

friend std::ostream &operator<<(std::ostream &os, const Heap<T> &heap)

{

for (const auto &item : heap.data)

{

os << \*item << " ";

}

return os;

};

/\*\*

\* @brief Overloaded input stream operator to insert values into the heap.

\*

\* Reads values from the input stream and inserts them into the heap.

\* @param is The input stream.

\* @param heap The Heap object to insert values into.

\* @return A reference to the input stream.

\*/

friend std::istream &operator>>(std::istream &is, Heap<T> &heap)

{

T value;

while (is >> value)

{

heap.insert(value);

}

return is;

};

/\*\*

\* @brief Get an iterator pointing to the beginning of the heap.

\*

\* @return An iterator pointing to the beginning of the heap.

\*/

typename std::vector<std::shared\_ptr<T>>::iterator begin()

{

return data.begin();

};

/\*\*

\* @brief Get an iterator pointing to the end of the heap.

\*

\* @return An iterator pointing to the end of the heap.

\*/

typename std::vector<std::shared\_ptr<T>>::iterator end()

{

return data.end();

};

};

}

#include "heap.cpp"

* 1. heap.cpp

#pragma once

using namespace heap;

template <typename T>

Heap<T>::Heap() {}

template <typename T>

void Heap<T>::insert(const T &value)

{

data.push\_back(std::make\_shared<T>(value));

heapifyUp();

}

template <typename T>

void Heap<T>::remove()

{

if (isEmpty())

{

throw std::out\_of\_range("Heap is empty");

}

std::swap(data[0], data.back());

data.pop\_back();

heapifyDown();

}

template <typename T>

const T &Heap<T>::getTop() const

{

if (isEmpty())

{

throw std::out\_of\_range("Heap is empty");

}

return \*data[0];

}

template <typename T>

bool Heap<T>::isEmpty() const

{

return data.empty();

}

template <typename T>

void Heap<T>::heapifyDown()

{

int index = 0;

while (true)

{

int leftChildIndex = 2 \* index + 1;

int rightChildIndex = 2 \* index + 2;

int largestIndex = index;

if (leftChildIndex < data.size() && \*data[leftChildIndex] > \*data[largestIndex])

{

largestIndex = leftChildIndex;

}

if (rightChildIndex < data.size() && \*data[rightChildIndex] > \*data[largestIndex])

{

largestIndex = rightChildIndex;

}

if (largestIndex != index)

{

std::swap(data[index], data[largestIndex]);

index = largestIndex;

}

else

{

break;

}

}

}

template <typename T>

void Heap<T>::heapifyUp()

{

int index = data.size() - 1;

while (index > 0)

{

int parentIndex = (index - 1) / 2;

if (\*data[index] > \*data[parentIndex])

{

std::swap(data[index], data[parentIndex]);

index = parentIndex;

}

else

{

break;

}

}

}

* 1. heap\_test.cpp

#include "gtest/gtest.h"

#include "../src/heap.h"

/\*\*

\* @brief MyData Class implementation.

\*

\* The MyData class represents a simple data container with a single integer value.

\* It includes methods for retrieving the stored value, as well as operators for

\* comparison, output, and input for convenient usage in data structures like heaps.

\*/

class MyData

{

private:

int value; /\*\*< The integer value stored in MyData. \*/

public:

/\*\*

\* @brief Constructor for MyData.

\*

\* @param val The initial value to be stored in MyData.

\*/

MyData(int val) : value(val) {};

/\*\*

\* @brief Get the stored value in MyData.

\*

\* @return The integer value stored in MyData.

\*/

int getValue() const

{

return value;

};

/\*\*

\* @brief Comparison operator for MyData.

\*

\* Enables comparison between two MyData objects based on their stored values.

\*

\* @param other Another MyData object to compare with.

\* @return true if the current object's value is greater than the other's, false otherwise.

\*/

bool operator>(const MyData &other) const

{

return value > other.getValue();

};

/\*\*

\* @brief Output stream operator for MyData.

\*

\* Enables convenient printing of MyData objects to an output stream.

\*

\* @param os The output stream.

\* @param data The MyData object to be printed.

\* @return A reference to the output stream.

\*/

friend std::ostream &operator<<(std::ostream &os, const MyData &data)

{

os << data.value;

return os;

};

/\*\*

\* @brief Input stream operator for MyData.

\*

\* Enables convenient reading of MyData objects from an input stream.

\*

\* @param is The input stream.

\* @param data The MyData object to store the read value.

\* @return A reference to the input stream.

\*/

friend std::istream &operator>>(std::istream &is, MyData &data)

{

is >> data.value;

return is;

};

};

TEST(HeapTest, InsertAndExtract)

{

Heap<MyData> heap;

heap.insert(MyData(5));

heap.insert(MyData(10));

heap.insert(MyData(3));

EXPECT\_EQ(heap.getTop().getValue(), 10);

heap.remove();

EXPECT\_EQ(heap.getTop().getValue(), 5);

heap.remove();

EXPECT\_EQ(heap.getTop().getValue(), 3);

heap.remove();

EXPECT\_THROW(heap.getTop(), std::out\_of\_range);

EXPECT\_THROW(heap.remove(), std::out\_of\_range);

}

TEST(HeapTest, Iterators)

{

Heap<MyData> heap;

heap.insert(MyData(5));

heap.insert(MyData(10));

heap.insert(MyData(3));

auto it = heap.begin();

EXPECT\_EQ((\*it)->getValue(), 10);

++it;

EXPECT\_EQ((\*it)->getValue(), 5);

++it;

EXPECT\_EQ((\*it)->getValue(), 3);

}

TEST(HeapTest, InsertUsingInputOperator)

{

Heap<int> heap;

std::istringstream input("5 10 3");

input >> heap;

EXPECT\_EQ(heap.getTop(), 10);

}

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

{

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

return RUN\_ALL\_TESTS();

}

1. Результаты модульного тестирования

[==========] Running 3 tests from 1 test suite.

[----------] Global test environment set-up.

[----------] 3 tests from HeapTest

[ RUN ] HeapTest.InsertAndExtract

[ OK ] HeapTest.InsertAndExtract (2 ms)

[ RUN ] HeapTest.Iterators

[ OK ] HeapTest.Iterators (0 ms)

[ RUN ] HeapTest.InsertUsingInputOperator

[ OK ] HeapTest.InsertUsingInputOperator (0 ms)

[----------] 3 tests from HeapTest (2 ms total)

[----------] Global test environment tear-down

[==========] 3 tests from 1 test suite ran. (2 ms total)

[ PASSED ] 3 tests.

1. Результаты Review

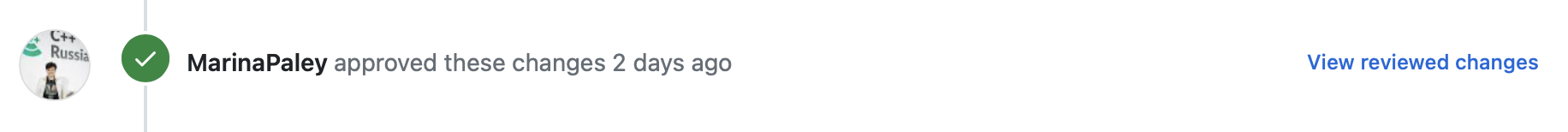


Рисунок 2 – Результаты review – получен approve

1. Вывод

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