МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации

**

**ЛАБАРАТОРНАЯ РАБОТА №3**

**«**Объектно-ориентированное программирование»

**по дисциплине: «*Программирование*»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АБ-320», «АВТФ»  *Сычук Алексей Александрович*  «2» декабря 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  *Ассистент кафедры ЗИ*  *Исаев Глеб Андреевич*  «2» декабря 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2024

**Цели и задачи работы:** изучение основных принципов объектно ориентированного программирования и основ юнит-тестирования.

**Задание к работе:** Самостоятельно решить задачи в соответствии с индивидуальным вариантом.

**Методика выполнения работы:**

1. Разработать алгоритмы решения задачи по индивидуальному заданию.

2. Написать и отладить программы решения задачи (С++, Go или Rust).

3. Протестировать работу программ на различных исходных данных.

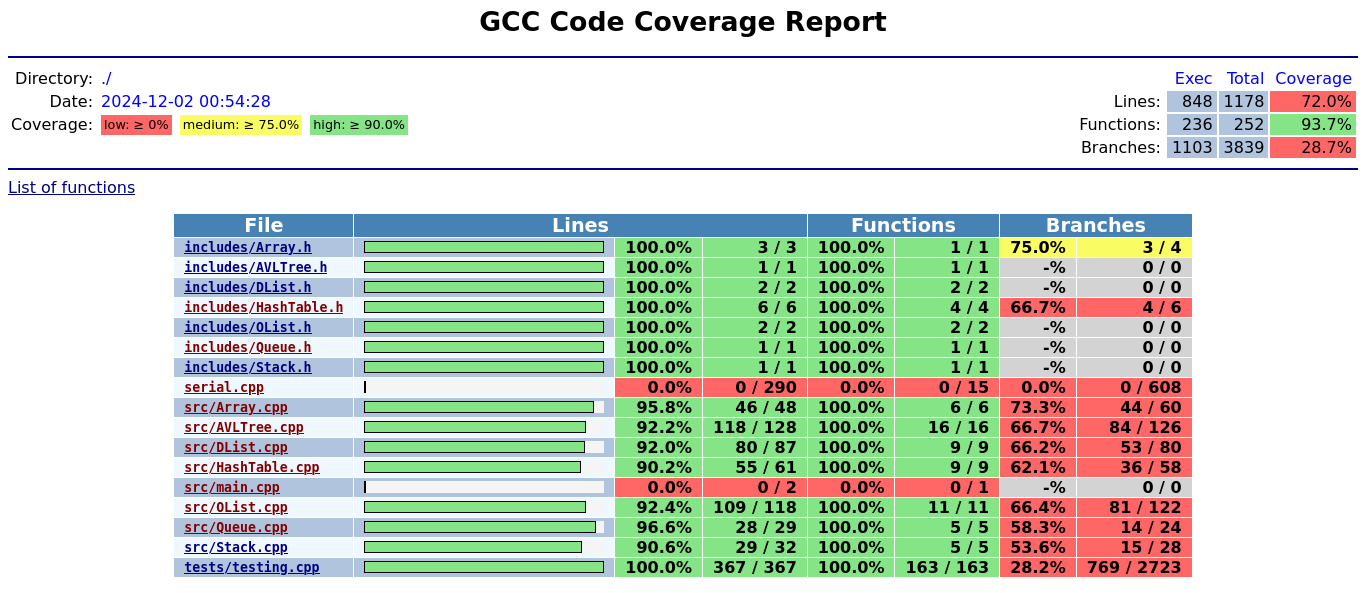
4. По запросу преподавателя быть готовым модифицировать/добавить алгоритмы/блоки кода в контексте ООП.

5. Ответить на теоретические вопросы к лабораторной работе на выбор преподавателя (не менее двух вопросов).

**Github:** <https://github.com/golimghostpy/Laba_3.git>

**Ход работы:**

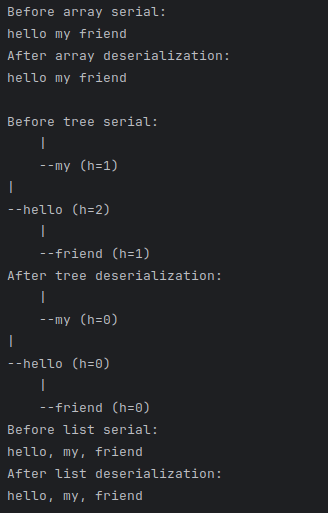
Результат покрытия тестами классов на С++:

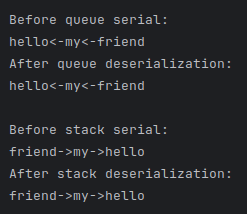


Результат покрытия тестами очереди на Go:

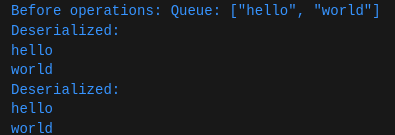


Сериализация и десериализация на С++:

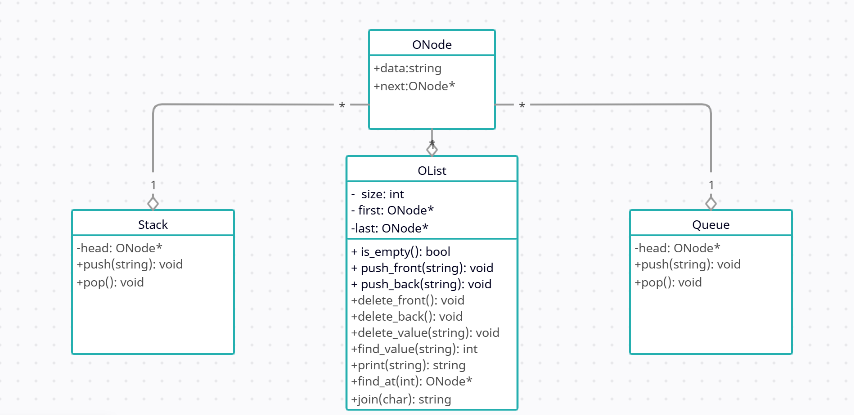


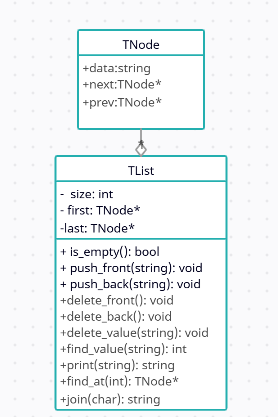


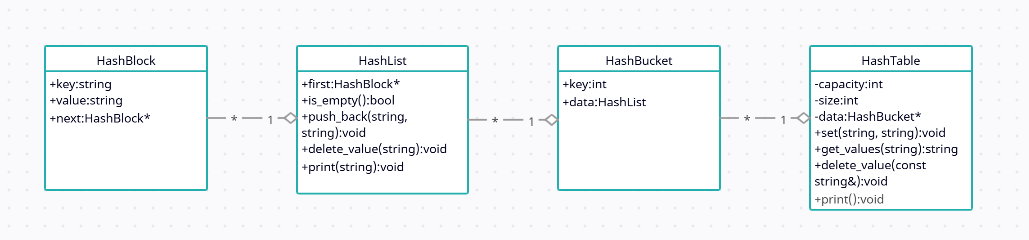
Сериализация и десериализация на Go:

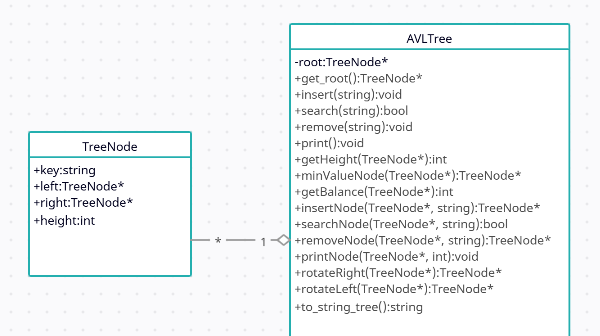


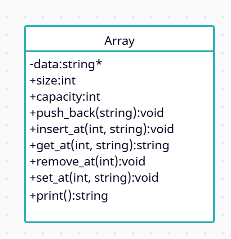
UML диаграммы:











Анализ российских стандартов оценки качества ПО.

В условиях стремительного развития информационных технологий и увеличения объема программного обеспечения, которое используется в различных сферах деятельности, становится особенно актуальным вопрос обеспечения качества программных продуктов. Одним из ключевых аспектов этого процесса является разработка и внедрение стандартов, которые определяют требования к системам и программному обеспечению. В Российской Федерации таким стандартом является национальный стандарт, охватывающий системную и программную инженерию, а также требования и оценку качества систем и программного обеспечения, известный как SQuaRE (Systems and Software Quality Requirements and Evaluation).

Данный стандарт устанавливает единые подходы к оценке качества программного обеспечения, включая его восстанавливаемость, что является важным аспектом для обеспечения надежности и устойчивости программных систем. В этом документе мы рассмотрим основные положения стандарта, его значение для отрасли, а также основные критерии и методы оценки восстанавливаемости программного обеспечения.

1. Значение стандарта SQuaRE

Стандарт SQuaRE был разработан для того, чтобы обеспечить единый подход к оценке качества программного обеспечения и систем. Он включает в себя набор требований, которые помогают организациям определить, как оценивать и управлять качеством своих продуктов. Основная цель стандарта — создание единой базы для оценки качества, что позволяет упростить взаимодействие между различными участниками процесса разработки и эксплуатации программного обеспечения.

1.1. Основные компоненты стандарта

Стандарт SQuaRE включает в себя несколько ключевых компонентов:

- Качество: Определение качества программного обеспечения и систем, включая его характеристики и атрибуты.

- Требования к качеству: Установление требований к качеству на разных уровнях разработки и эксплуатации.

- Методы оценки: Разработка методов и инструментов для оценки качества программного обеспечения, включая его восстанавливаемость.

Эти компоненты позволяют организациям более эффективно управлять качеством программного обеспечения, обеспечивая его соответствие установленным требованиям.

2. Оценка качества программного обеспечения

Оценка качества программного обеспечения — это процесс, который включает в себя сбор и анализ данных о программном продукте с целью определения его соответствия установленным требованиям. В рамках стандарта SQuaRE оценка качества осуществляется по нескольким критериям, включая функциональность, надежность, удобство использования, производительность и, что особенно важно, восстанавливаемость.

Фазы оценки:



**Базовая фаза** определяет операционные характеристики системы при отсутствии введенных возмущений. Данная фаза предназначена для определения базового уровня производительности, который должен использоваться далее для сравнения с результатами фазы тестирования и удовлетворять всем требованиям, определяемым рабочей нагрузкой.

**Фаза тестирования** определяет операционные характеристики системы под рабочей нагрузкой в присутствии возмущающих воздействий. Необходимо, чтобы на данной фазе были использованы те же установки и конфигурация, что и на базовой фазе.

Фаза тестирования разбивается на множество последовательных периодов инъекции возмущения. Эти инъекционные периоды должны следовать один за другим в указанной последовательности.

Цель **фазы проверки** - удостовериться, что реакция системы на возмущения не повлияла на целостность системы. Проверка во время этой фазы должна подтвердить, что система находится в согласованном состоянии.

2.1. Критерии оценки качества

Критерии оценки качества программного обеспечения, предусмотренные стандартом SQuaRE, включают:

- Функциональность: Способность системы выполнять заданные функции и задачи.

- Надежность: Способность системы выполнять свои функции в определенных условиях в течение заданного времени.

- Удобство использования: Легкость, с которой пользователи могут взаимодействовать с системой.

- Производительность: Эффективность работы системы в отношении использования ресурсов.

- Восстанавливаемость: Способность системы восстанавливаться после сбоя или потери данных.

Каждый из этих критериев имеет свои подкатегории и параметры, которые позволяют более точно оценить качество программного обеспечения.

3. Модуль оценки восстанавливаемости

Восстанавливаемость — это важный аспект качества программного обеспечения, который определяет, насколько эффективно система может восстановиться после сбоя или потери данных. Модуль оценки восстанавливаемости в стандарте SQuaRE включает в себя набор требований и методов, которые помогают организациям оценить этот параметр.

3.1. Определение восстанавливаемости

Восстанавливаемость определяется как способность системы восстанавливаться после сбоев и продолжать функционировать. Это включает в себя не только восстановление данных, но и возможность быстрого восстановления работы системы после возникновения проблем.

3.2. Критерии оценки восстанавливаемости

Для оценки восстанавливаемости программного обеспечения используются следующие критерии:

- Время восстановления: Время, необходимое для восстановления системы после сбоя.

- Процент успешных восстановлений: Доля случаев, когда система успешно восстанавливается после сбоев.

- Наличие резервных копий: Наличие и регулярность создания резервных копий данных и конфигураций системы.

Эти критерии позволяют организациям оценить, насколько эффективно их программное обеспечение может справляться с проблемами и сбоями, что является критически важным для обеспечения надежности и устойчивости систем.

4. Практическое применение стандарта SQuaRE

На практике стандарт SQuaRE может быть использован как для оценки качества новых программных продуктов, так и для улучшения существующих решений. Организации могут применять методы и критерии, предусмотренные стандартом, для проведения регулярных аудитов качества программного обеспечения, что позволяет выявлять и устранять проблемы на ранних стадиях разработки.

4.1. Внедрение стандарта в организации

Внедрение стандарта SQuaRE в организацию требует комплексного подхода, включая:

- Обучение персонала: Обучение сотрудников методам и критериям оценки качества программного обеспечения.

- Разработка процессов: Создание и внедрение процессов, основанных на стандарте, для оценки и управления качеством.

- Использование инструментов: Применение инструментов для автоматизации процессов оценки качества и восстанавливаемости.

Эти шаги помогут организациям более эффективно управлять качеством программного обеспечения и повышать его надежность.

Национальный стандарт Российской Федерации в области информационных технологий и системной инженерии, SQuaRE, является важным инструментом для оценки и управления качеством программного обеспечения. Включение модуля оценки восстанавливаемости в стандарт позволяет организациям более эффективно справляться с проблемами, связанными с надежностью и устойчивостью их программных систем.

Применение стандарта SQuaRE способствует повышению качества программного обеспечения, что, в свою очередь, улучшает удовлетворенность пользователей и снижает риски, связанные с эксплуатацией программных систем. В условиях современного мира, где информационные технологии играют ключевую роль в различных сферах жизни, стандарты качества становятся неотъемлемой частью успешного функционирования организаций.

Основные виды тестирования.

**Модульное тестирование**, или **юнит-тестирование** (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

**Преимущества:**

* Поощрение изменений

Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (регрессионное тестирование). Это поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений.

* Упрощение интеграции

Модульное тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируя отдельные части программы, а затем программу в целом.

* Документирование кода

Модульные тесты можно рассматривать как «живой документ» для тестируемого класса. Клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

* Отделение интерфейса от реализации

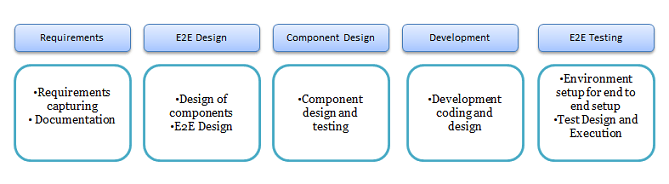
Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Например, класс пользуется базой данных; в ходе написания теста программист обнаруживает, что тесту приходится взаимодействовать с базой. Это ошибка, поскольку тест не должен выходить за границу класса. В результате разработчик абстрагируется от соединения с базой данных и реализует этот интерфейс, используя свой собственный mock-объект. Это приводит к менее связанному коду, минимизируя зависимости в системе.

**Интеграционное тестирование** – это тип тестирования, при котором программные модули объединяются логически и тестируются как группа. Как правило, программный продукт состоит из нескольких программных модулей, написанных разными программистами. Целью нашего тестирования является выявление багов при взаимодействии между этими программными модулями и в первую очередь направлен на проверку обмена данными между этими самими модулями. Именно поэтому оно также называется «I & T» (интеграция и тестирование), «тестирование строк» и иногда «тестирование потоков».

**Сквозное тестирование** (End-to-end, E2E, Chain testing) — это вид тестирования, используемый для проверки программного обеспечения от начала до конца, а также его интеграцию с внешними интерфейсами. Цель сквозного тестирования состоит в проверке всего программного обеспечения на предмет зависимостей, целостности данных и связи с другими системами, интерфейсами и базами данных для проверки успешного выполнения полного производственного сценария.

Наряду с программной системой тестирование также обеспечивает проверку пакетной обработки и обработки данных из других вышестоящих и нижестоящих систем. Отсюда и название «End-to-End». Сквозное тестирование обычно проводится после функционального и системного тестирования. Для его проведения используются реальные данные и тестовая среда для имитации рабочего режима.

Процесс сквозного тестирования:



**UI-тестирование** (тестирование пользовательского интерфейса) — это процесс проверки графического интерфейса приложения для обеспечения его корректной работы и удобства использования. Оно фокусируется на взаимодействии пользователя с интерфейсом, включая элементы управления, кнопки, формы и навигацию. Основная цель UI-тестирования — убедиться, что приложение соответствует требованиям пользователей и обеспечивает интуитивно понятный опыт.

UI-тесты могут быть как ручными, так и автоматизированными. Автоматизированное тестирование часто выполняется с использованием фреймворков, таких как Selenium, Cypress или TestComplete, что позволяет быстро проверять различные сценарии взаимодействия. Эти тесты помогают выявлять визуальные ошибки, проблемы с совместимостью на разных устройствах и браузерах, а также обеспечивают, что изменения в коде не нарушают функциональность интерфейса.

Список источников:

1. <https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5>
2. <https://logrocon.ru/news/intgration_testing>
3. <https://habr.com/ru/companies/otus/articles/681066/>
4. <https://docs.cntd.ru/document/1200121070?marker=7DI0KA&section=text>

**Вывод**

В ходе выполнения данной лабораторной работы были изучены основные виды тестирования, а также юнит-тестирование на языках С++ и Go. Также был получен опыт реализации сериализации и десериализации данных самописных классов.

**Исходный код(представлен не весь, поскольку прилагался к 1 лр):**

**Testing.cpp**

#include "../includes/libs.h"

#include "gtest/gtest.h"

#include "../includes/Array.h"

#include "../includes/OList.h"

#include "../includes/DList.h"

#include "../includes/Queue.h"

#include "../includes/Stack.h"

#include "../includes/AVLTree.h"

#include "../includes/HashTable.h"

int main(int, char \*\*);

// Array testing --------------------------------------------------

TEST(Array, Create)

{

auto testArr = new Array;

ASSERT\_TRUE(testArr != nullptr);

}

TEST(Array, Push\_back\_and\_get)

{

Array testArr;

ASSERT\_TRUE(testArr.size == 0 && testArr.capacity == 1);

testArr.push\_back("hello");

testArr.push\_back("world");

ASSERT\_TRUE(testArr.size == 2 && testArr.capacity == 2);

ASSERT\_TRUE(testArr.get\_at(0) == "hello");

ASSERT\_THROW(testArr.get\_at(5), out\_of\_range);

}

TEST(Array, Insert\_at)

{

Array testArr;

ASSERT\_TRUE(testArr.size == 0 && testArr.capacity == 1);

testArr.insert\_at(0, "hello");

testArr.insert\_at(0, "world");

ASSERT\_TRUE(testArr.size == 2 && testArr.capacity == 2);

ASSERT\_TRUE(testArr.get\_at(0) == "world");

ASSERT\_THROW(testArr.insert\_at(100, "bye");, out\_of\_range);

}

TEST(Array, Remove\_at)

{

Array testArr;

testArr.push\_back("hello");

testArr.push\_back("how");

testArr.push\_back("you");

testArr.remove\_at(2);

ASSERT\_THROW(testArr.get\_at(2), out\_of\_range);

ASSERT\_THROW(testArr.remove\_at(3);, out\_of\_range);

}

TEST(Array, Set\_at)

{

Array testArr;

testArr.push\_back("hello");

testArr.push\_back("how");

testArr.push\_back("you");

testArr.set\_at(1, "changed");

ASSERT\_TRUE(testArr.get\_at(1) == "changed");

ASSERT\_THROW(testArr.set\_at(3, "bye");, out\_of\_range);

}

TEST(Array, Print)

{

Array testArr;

testArr.push\_back("hello");

testArr.push\_back("world");

ASSERT\_TRUE(testArr.print() == "hello world \n");

}

// One way list testing --------------------------------------------------

TEST(OList, Create)

{

auto testList = new OList;

ASSERT\_TRUE(testList != nullptr);

}

TEST(OList, Is\_empty)

{

OList testList;

ASSERT\_TRUE(testList.is\_empty());

testList.push\_back("hello");

ASSERT\_FALSE(testList.is\_empty());

}

TEST(OList, Push\_front)

{

OList testList;

testList.push\_front("hello");

testList.push\_front("bye");

ASSERT\_TRUE(testList.find\_at(0)->data == "bye");

}

TEST(OList, Push\_back)

{

OList testList;

testList.push\_back("hello");

testList.push\_back("bye");

ASSERT\_TRUE(testList.find\_at(1)->data == "bye");

}

TEST(OList, Delete\_front)

{

OList testList;

ASSERT\_THROW(testList.delete\_front(), out\_of\_range);

testList.push\_front("hello");

testList.push\_front("bye");

testList.delete\_front();

ASSERT\_TRUE(testList.find\_at(0)->data == "hello");

}

TEST(OList, Delete\_back)

{

OList testList;

ASSERT\_THROW(testList.delete\_back(), out\_of\_range);

testList.push\_back("hello");

testList.push\_back("bye");

testList.delete\_back();

ASSERT\_TRUE(testList.find\_at(0)->data == "hello");

}

TEST(OList, Delete\_value)

{

OList testList;

ASSERT\_THROW(testList.delete\_value("hi"), out\_of\_range);

testList.push\_back("hello");

testList.push\_back("bye");

ASSERT\_THROW(testList.delete\_value("hi"), out\_of\_range);

testList.delete\_value("hello");

ASSERT\_TRUE(testList.find\_at(0)->data == "bye");

}

TEST(OList, Find\_value)

{

OList testList;

ASSERT\_TRUE(testList.find\_value("hello") == -1);

testList.push\_back("hello");

testList.push\_back("bye");

ASSERT\_TRUE(testList.find\_value("hi") == -1);

ASSERT\_TRUE(testList.find\_value("bye") == 1);

}

TEST(OList, Find\_at)

{

OList testList;

ASSERT\_THROW(testList.find\_at(0), out\_of\_range);

testList.push\_back("hello");

ASSERT\_THROW(testList.find\_at(-5), out\_of\_range);

testList.push\_back("and");

testList.push\_back("bye");

ASSERT\_TRUE(testList.find\_at(1)->data == "and");

}

TEST(OList, Join)

{

OList testList;

testList.push\_back("hello");

testList.push\_back("and");

testList.push\_back("bye");

ASSERT\_TRUE(testList.join(' ') == "hello and bye ");

}

TEST(OList, Print)

{

OList testList;

ASSERT\_TRUE(testList.print(" ") == "\n");

testList.push\_back("hello");

testList.push\_back("world");

ASSERT\_TRUE(testList.print(" ") == "hello world\n");

}

TEST(OList, Split)

{

string testString = "hello world";

OList testList = split(testString, " ");

ASSERT\_TRUE(testList.find\_at(0)->data == "hello" && testList.find\_at(1)->data == "world");

}

// Two way list testing --------------------------------------------------

TEST(TList, Create)

{

auto testList = new TList;

ASSERT\_TRUE(testList != nullptr);

}

TEST(TList, Is\_empty)

{

TList testList;

ASSERT\_TRUE(testList.is\_empty());

testList.push\_back("hello");

ASSERT\_FALSE(testList.is\_empty());

}

TEST(TList, Push\_front)

{

TList testList;

testList.push\_front("hello");

testList.push\_front("bye");

ASSERT\_TRUE(testList.find\_at(0)->data == "bye");

}

TEST(TList, Push\_back)

{

TList testList;

testList.push\_back("hello");

testList.push\_back("bye");

ASSERT\_TRUE(testList.find\_at(1)->data == "bye");

}

TEST(TList, Delete\_front)

{

TList testList;

ASSERT\_TRUE(testList.is\_empty());

ASSERT\_THROW(testList.delete\_front(), out\_of\_range);

testList.push\_front("hello");

testList.push\_front("bye");

testList.delete\_front();

ASSERT\_FALSE(testList.is\_empty());

ASSERT\_TRUE(testList.find\_at(0)->data == "hello");

}

TEST(TList, Delete\_back)

{

TList testList;

ASSERT\_THROW(testList.delete\_back(), out\_of\_range);

testList.push\_back("hello");

testList.push\_back("bye");

testList.delete\_back();

ASSERT\_TRUE(testList.find\_at(0)->data == "hello");

}

TEST(TList, Delete\_value)

{

TList testList;

ASSERT\_THROW(testList.delete\_value("hi"), out\_of\_range);

testList.push\_back("hello");

testList.push\_back("bye");

ASSERT\_THROW(testList.delete\_value("hi"), out\_of\_range);

testList.delete\_value("hello");

ASSERT\_TRUE(testList.find\_at(0)->data == "bye");

}

TEST(TList, Find\_value)

{

TList testList;

ASSERT\_TRUE(testList.find\_value("hello") == -1);

testList.push\_back("hello");

testList.push\_back("bye");

ASSERT\_TRUE(testList.find\_value("hi") == -1);

ASSERT\_TRUE(testList.find\_value("bye") == 1);

}

TEST(TList, Find\_at)

{

TList testList;

ASSERT\_THROW(testList.find\_at(0), out\_of\_range);

testList.push\_back("hello");

ASSERT\_THROW(testList.find\_at(-5), out\_of\_range);

testList.push\_back("and");

testList.push\_back("bye");

ASSERT\_TRUE(testList.find\_at(1)->data == "and");

}

TEST(TList, Print)

{

TList testList;

testList.push\_back("hello");

testList.push\_back("world");

ASSERT\_TRUE(testList.print(" ") == "hello world\n");

}

// Stack testing --------------------------------------------------

TEST(Stack, Create)

{

auto testStack = new Stack;

ASSERT\_TRUE(testStack != nullptr);

}

TEST(Stack, Is\_empty)

{

Stack testStack;

ASSERT\_TRUE(testStack.is\_empty());

testStack.push("hello");

ASSERT\_FALSE(testStack.is\_empty());

}

TEST(Stack, Push)

{

Stack testStack;

testStack.push("hello");

testStack.push("bye");

ASSERT\_TRUE(testStack.get\_head() == "bye");

}

TEST(Stack, Pop)

{

Stack testStack;

ASSERT\_THROW(testStack.pop(), out\_of\_range);

testStack.push("hello");

testStack.push("bye");

testStack.pop();

ASSERT\_TRUE(testStack.get\_head() == "hello");

}

TEST(Stack, Print)

{

Stack testStack;

testStack.push("hello");

testStack.push("world");

ASSERT\_TRUE(testStack.print() == "world->hello\n");

}

// Queue testing --------------------------------------------------

TEST(Queue, Create)

{

auto testQueue = new Queue;

ASSERT\_TRUE(testQueue != nullptr);

}

TEST(Queue, Is\_empty)

{

Queue testQueue;

ASSERT\_TRUE(testQueue.is\_empty());

testQueue.push("hello");

ASSERT\_FALSE(testQueue.is\_empty());

}

TEST(Queue, Push)

{

Queue testQueue;

testQueue.push("hello");

testQueue.push("bye");

ASSERT\_TRUE(testQueue.get\_head() == "hello");

}

TEST(Queue, Pop)

{

Queue testQueue;

ASSERT\_THROW(testQueue.pop(), out\_of\_range);

testQueue.push("hello");

testQueue.push("bye");

testQueue.pop();

ASSERT\_TRUE(testQueue.get\_head() == "bye");

}

TEST(Queue, Print)

{

Queue testQueue;

testQueue.push("hello");

testQueue.push("world");

ASSERT\_TRUE(testQueue.print() == "hello<-world\n");

}

// AVLTree testing --------------------------------------------------

TEST(Tree, Create)

{

auto testTree = new AVLTree;

ASSERT\_TRUE(testTree != nullptr);

}

TEST(Tree, Get\_root)

{

AVLTree testTree;

ASSERT\_TRUE(testTree.get\_root() == nullptr);

testTree.insert("hello");

ASSERT\_TRUE(testTree.get\_root()->key == "hello");

}

TEST(Tree, Min\_value\_node)

{

AVLTree testTree;

testTree.insert("30");

testTree.insert("20");

testTree.insert("40");

ASSERT\_TRUE(testTree.minValueNode(testTree.get\_root())->key == "20");

}

TEST(Tree, Insert) {

AVLTree testTree;

testTree.insert("30");

testTree.insert("20");

testTree.insert("40");

ASSERT\_TRUE(testTree.search("30"));

ASSERT\_TRUE(testTree.search("20"));

ASSERT\_TRUE(testTree.search("40"));

ASSERT\_FALSE(testTree.search("10"));

}

TEST(Tree, Remove) {

AVLTree testTree;

testTree.insert("30");

testTree.insert("20");

testTree.insert("40");

testTree.insert("10");

ASSERT\_TRUE(testTree.search("10"));

testTree.remove("10");

ASSERT\_FALSE(testTree.search("10"));

}

TEST(Tree, Insert\_and\_balance) {

AVLTree testTree;

testTree.insert("30");

testTree.insert("20");

testTree.insert("10");

testTree.insert("10");

ASSERT\_TRUE(testTree.get\_root()->key == "20");

AVLTree testTree1;

testTree1.insert("10");

testTree1.insert("20");

testTree1.insert("30");

ASSERT\_TRUE(testTree1.get\_root()->key == "20");

AVLTree testTree2;

testTree2.insert("30");

testTree2.insert("10");

testTree2.insert("20");

ASSERT\_TRUE(testTree2.get\_root()->key == "20");

AVLTree testTree3;

testTree3.insert("10");

testTree3.insert("30");

testTree3.insert("20");

ASSERT\_TRUE(testTree3.get\_root()->key == "20");

}

TEST(Tree, Remove\_and\_balance) {

AVLTree testTree;

testTree.insert("2");

testTree.insert("1");

testTree.insert("3");

testTree.insert("4");

ASSERT\_TRUE(testTree.get\_root()->key == "2");

testTree.remove("1");

ASSERT\_TRUE(testTree.get\_root()->key == "3");

AVLTree testTree1;

testTree1.insert("3");

testTree1.insert("2");

testTree1.insert("4");

testTree1.insert("1");

ASSERT\_TRUE(testTree1.get\_root()->key == "3");

testTree1.remove("4");

ASSERT\_TRUE(testTree1.get\_root()->key == "2");

AVLTree testTree2;

testTree2.insert("3");

testTree2.insert("2");

testTree2.insert("4");

testTree2.insert("1");

ASSERT\_TRUE(testTree2.get\_root()->key == "3");

testTree2.remove("2");

ASSERT\_TRUE(testTree2.get\_root()->key == "3");

}

TEST(Tree, Print\_empty\_tree) {

AVLTree testTree;

testing::internal::CaptureStdout();

testTree.print();

string output = testing::internal::GetCapturedStdout();

ASSERT\_EQ(output, "");

}

TEST(Tree, Print\_nonempty\_tree) {

AVLTree testTree;

testTree.insert("30");

testTree.insert("20");

testTree.insert("40");

testing::internal::CaptureStdout();

testTree.print();

string output = testing::internal::GetCapturedStdout();

ASSERT\_TRUE(output.find("30") != string::npos);

ASSERT\_TRUE(output.find("20") != string::npos);

ASSERT\_TRUE(output.find("40") != string::npos);

}

TEST(Tree, To\_string\_tree) {

AVLTree testTree;

ASSERT\_EQ(testTree.to\_string\_tree(), "");

testTree.insert("30");

testTree.insert("20");

testTree.insert("40");

ASSERT\_EQ(testTree.to\_string\_tree(), "30 20 40 ");

}

// HashTable testing --------------------------------------------------

TEST(HashTable, Insert\_and\_get) {

HashTable testTable;

testTable.set("key1", "value1");

ASSERT\_EQ(testTable.get\_values("key1"), "value1");

testTable.set("key2", "value2");

ASSERT\_EQ(testTable.get\_values("key2"), "value2");

}

TEST(HashTable, Get\_non\_exist) {

HashTable testTable;

ASSERT\_EQ(testTable.get\_values("nonexistent\_key"), "No such a key");

}

TEST(HashTable, Insert\_duplicate) {

HashTable testTable;

testTable.set("key1", "value1");

testTable.set("key1", "value2");

ASSERT\_EQ(testTable.get\_values("key1"), "value1");

}

TEST(HashTable, Delete) {

HashTable testTable;

ASSERT\_THROW(testTable.delete\_value("key1"), out\_of\_range);

testTable.set("key1", "value1");

ASSERT\_THROW(testTable.delete\_value("key2"), out\_of\_range);

testTable.delete\_value("key1");

ASSERT\_EQ(testTable.get\_values("key1"), "No such a key");

}

TEST(HashTable, Print) {

HashTable testTable;

testTable.set("key1", "value1");

testTable.set("key2", "value2");

testing::internal::CaptureStdout();

testTable.print();

string output = testing::internal::GetCapturedStdout();

ASSERT\_NE(output.find("0: "), string::npos);

}

TEST(HashTable, Insert\_alot) {

HashTable testTable;

for (int i = 0; i < 100; ++i) {

testTable.set("key" + to\_string(i), "value" + to\_string(i));

}

ASSERT\_EQ(testTable.get\_values("key99"), "value99");

}

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

{

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

return RUN\_ALL\_TESTS();

}

**Main.go**

package main

import (

"bufio"

"fmt"

"os"

)

func (q \*Queue) serialize(filename string) error {

file, err := os.Create(filename)

if err != nil {

return err

}

defer file.Close()

curr := q.Head

for curr != nil {

file.Write([]byte(curr.Data + "\n"))

curr = curr.Next

}

return nil

}

func deserialize(filename string) error {

file, err := os.Open(filename)

if err != nil {

return err

}

defer file.Close()

fmt.Println("Deserialized:")

scanner := bufio.NewScanner(file)

for scanner.Scan() {

fmt.Println(scanner.Text())

}

return nil

}

func main() {

q := construct\_queue()

q.push("hello")

q.push("world")

fmt.Println("Before operations:", q.print())

// Сериализация в бинарный файл

if err := q.serialize("queue.bin"); err != nil {

fmt.Println("Error serializing to binary:", err)

}

// Сериализация в текстовый файл

if err := q.serialize("queue.txt"); err != nil {

fmt.Println("Error serializing to text:", err)

}

deserialize("queue.bin")

deserialize("queue.txt")

}

**Queue.go**

package main

import (

"fmt"

"strings"

)

type Node struct {

Data string

Next \*Node

}

type Queue struct {

Head \*Node

}

func construct\_queue() \*Queue {

return &Queue{}

}

func (q \*Queue) push(element string) {

newNode := &Node{Data: element}

if q.Head == nil {

q.Head = newNode

return

}

current := q.Head

for current.Next != nil {

current = current.Next

}

current.Next = newNode

}

func (q \*Queue) pop() (string, error) {

if q.Head == nil {

return "", fmt.Errorf("Queue is empty")

}

popped := q.Head

q.Head = q.Head.Next

return popped.Data, nil

}

func (q \*Queue) print() string {

var sb strings.Builder

sb.WriteString("Queue: [")

curr := q.Head

for curr != nil {

if curr.Next != nil {

sb.WriteString(fmt.Sprintf("%q", curr.Data))

sb.WriteString(", ")

} else {

sb.WriteString(fmt.Sprintf("%q", curr.Data))

}

curr = curr.Next

}

sb.WriteString("]")

return sb.String()

}

**Queue\_test.go**

package main

import (

"testing"

)

func TestQueue(t \*testing.T) {

testQueue := construct\_queue()

testQueue.push("hello")

testQueue.push("world")

if got := testQueue.print(); got != `Queue: ["hello", "world"]` {

t.Errorf("print() = %v, want %v", got, `Queue: ["hello", "world"]`)

}

elem, err := testQueue.pop()

if err != nil {

t.Fatalf("pop() error = %v", err)

}

if elem != "hello" {

t.Errorf("pop() = %v, want %v", elem, "hello")

}

if got := testQueue.print(); got != `Queue: ["world"]` {

t.Errorf("print() = %v, want %v", got, `Queue: ["world"]`)

}

elem, err = testQueue.pop()

if err != nil {

t.Fatalf("pop() error = %v", err)

}

if elem != "world" {

t.Errorf("pop() = %v, want %v", elem, "world")

}

if got, err := testQueue.pop(); err == nil {

t.Errorf("pop() = %v, want error", got)

}

}