Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский университет
«Высшая школа экономики»

Кафедра «Компьютерная безопасность»

ОТЧЕТ
К ЛАБОРАТОРНОЙ РАБОТЕ №2

по дисциплине

«Языки программирования»

Работу выполнил
студент группы СКБ-201 _____ Г.П. Кашкин
подпись, дата

Работу проверил _____ С.А. Булгаков
подпись, дата

Москва 2021

Содержание

Постановка задачи

Разработать программу на языке Си++ (ISO/IEC 14882:2020), демонстрирующую решение поставленной задачи.

Общая часть

Разработать класс ADTи унаследовать от него классы, разработанные в рамках лабораторной ра-боты 1. Разработать набор классов, объекты которых реализуют типы данных, указанные ниже. Для этих классовразработать необходимые конструкторы, деструктор, конструктор копирования. Разрабо-тать операции: добавления/удаления элемента (уточнено в задаче); получения количества элемен-тов; доступа к элементу (перегрузить оператор []). При ошибках запускать исключение.В главной функции разместить тесты, разработанные с использованием библиотеки GoogleTest.

Задачи

а) Динамический массив указателей на объекты `ADT`. Размерностьмассива указателей увеличивается в момент его переполнения. Начальная раз-мерность задается как параметр конструктора, значение по умолчанию 0.Добавление/удале-ниеэлемента в произвольное место.

б) Стек, представленный динамическим массивом указа-телейна хранимые объекты `ADT`. Размерность стека увеличивается в моментего переполне-ния.Начальная размерность задается как параметр конструктора, значение по умолчанию 0.Добавление/удаление элемента в начало и в конец.

в) Односвязный список, содержащий указатели на объекты `ADT`. Добавление/удаление элемента в произвольное место.

г) Циклическая очередь, представленная динамическим массивомуказателей на хранимые объектыADT. Добавление/удаление элемента в произ-вольное место.

д) Двоичное дерево, содержащее указатели на объекты `ADT`. Добавление/удаление элемента в произвольное место.

1 Алгоритм решения задачи

1.1 Задание 1

Для решения задачи был реализован собственный класс с интерфейсом аналогичным интерфейсу `std::vector`
//TODO написать про выбор оптимальной константы для увеличения capacity

1.2 Задание 2

Для решения задачи была написана структура дека (быстрое добавлени и удаление объектов из обоих концов). Класс реализован на базе динамического массива, помимо него отслеживается вместимость, текущий размер и позиции начала и конца значащих элементов. Таким образом, когда начало или конец находится на границе выделенной памяти, новый элемент добавляется с противоположной стороны, как и все последующие. Для большего удобства выбран инвариант вместимости контейнера как степени двойки, потому что тогда `(pos + n) % capacity` будет корректным с учетом переполнения для всех n. Помимо этого, `capacity` хранится в виде величины на единицу меньше вместимости, таким образом вместо операции деления с остатком можно использовать операцию побитового И (`size_t new_id = (id - n) & capacity` будет корректным в том числе когда $id < n$), что немного быстрее и изящнее. Получается, устредненная ассимптотика аналогична ассимптотике классического вектора и равна $O(1)$, как и доступ по индексу.

1.3 Задание 3

Для решения данной задачи был написан связный список, состоящей из внешнего класса и сервисной структуры ноды. В ноде хранится только id следующего значения и полезная нагрузка, в структуре хранится `ObjPool`, выполняющий функции простейшего аллокатора, id первой ноды и размер списка. `ObjPool` в свою очередь действует как односвязный список на базе динамического массива. При выдаче ноды из пула, возвращается значение из начала списка, при возвращении оно ставится в начало, таким образом это происходит с константной асимптотикой, но не требует вызова медленных системных инструкций аллоцирования. Так как список односвязный по условию, реализовн только `forward_iterator`, а добавление, удаление или модификация ноды происходит с линейной сложностью, потому что до нее нужно добраться поштучно перебирая предков с начала.

1.4 Задание 4

Циклическая очередь реализована так же на базе дека. Так как добавление и удаление объекта в середину в любом случае имеет линейную сложность, оно реализовано в виде переаллоцирования в новый массив с выравниванием концов.

1.5 Задание 5

Для данного задания был взят уже написанный в предыдущей лабораторной класс декартового дерева. Для оптимизированной индексации была добавлена запланированная поддежка `random_access_iterator` базирующаяся на двухсвязности нод дерева, предподсчета размера поддерева для каждой ноды и базовом инварианте дерева поиска (все левые дети меньше, все правые больше). Таким образом, все требуемые операции совершаются за $O(\log n)$ - усредненную высоту дерева.

2 Выполнение задания

2.1 Задание 1

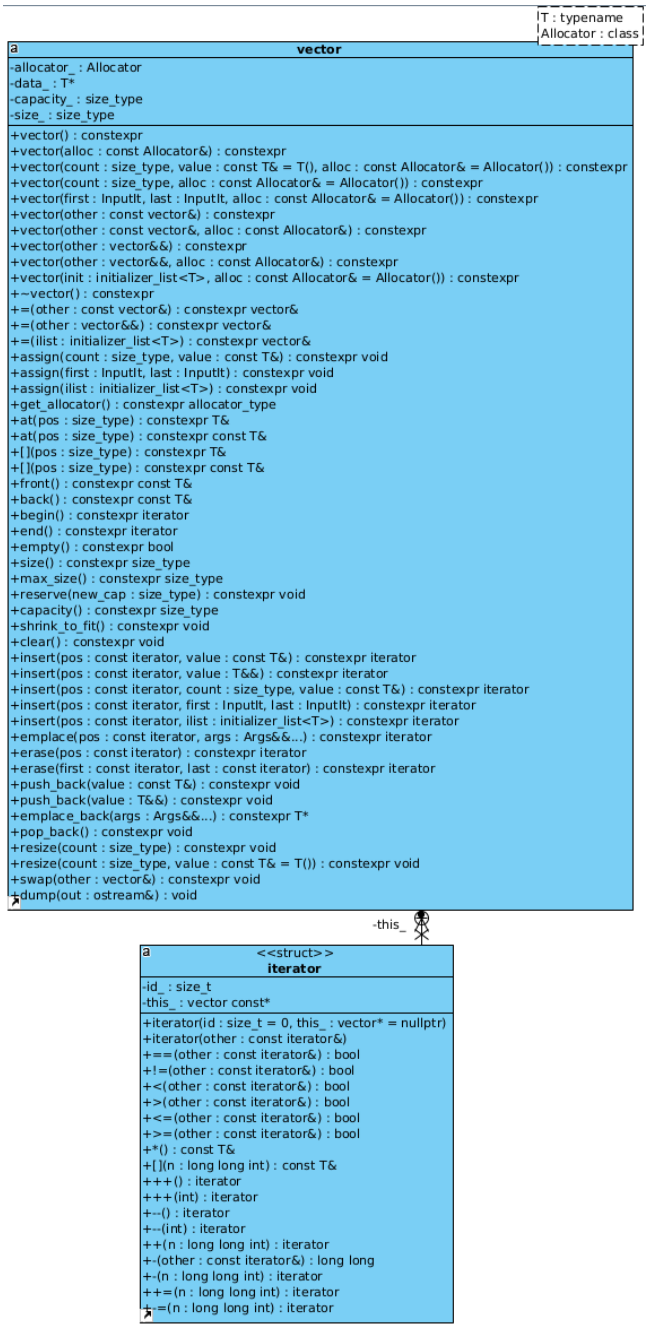

Рис. 1. UML 2.0 diagram for vector class

Для решения данной задачи разработан класс `vector`, хранящий в себе динамический массив объектов, текущий размер, вместимость и аллокатор для работы с системными вызовами. Аллокатор может предоставляться как параметр шаблонна, по умолчанию используется `std::allocator` и вся работа с динамической памятью базируется на его интерфейсе. Интерфес класса намеренно копирует `std::vector` в стандарте `c++20`.
//TODO распространиться на тему дедуцирования шаблона указателей в конструкторе и прочих, когда разберусь
Очевидно, что на такой простой линейной структуре легко реализовать `random_access` итераторы, что и было сделано. Для базовой безопасности при отладке расставлены ассерты, при проблемах с выделением памяти бросаются соответствующие ошибки.

2.2 Задания 2 и 4

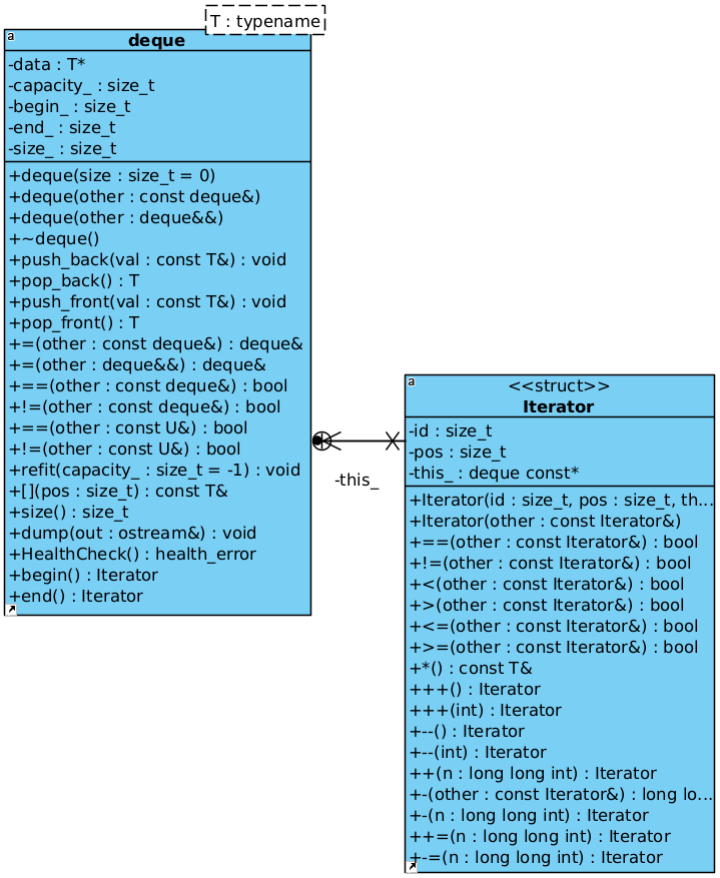

Рис. 2. UML 2.0 diagram for deque class

Для решения данной задачи разработан класс `deque`, основаный на массиве в динамической памяти. Для обработки выхода конца данный за край массива позиция постоянно делится с остатком на capacity (в данном случае $\texttt{capacity} = 2^n - 1$, так что используется эквивалентное и более оптимальное `size_t new_id = (id + n) & capacity`). Для упрощения итераторы хранят не только текущий id, но и позицию в деке. Помимо этого хранится указатель на структуру дека для упрощенного доступа к его полям. Так как хранится позиция, инвалидированный указатель (он же `end()`) это указатель с позицией равной длинне дека, что позволяет узнавать длинну структуры одним из каноничных способов, через `random_access` итераторы `end() - begin()`. Для комфортной отладки реализованы функции проверки состояния дека с макросом, возвращающим информацию о проблеме и делающем дамп структуры. Помимо проверки базовых инвариантов (`size <= capacity`) пустующие объекты заполняются специальным значением (`POISON`) и элементы массива проверяются на "отравленность"в соответствии со своим расположением относительно `begin` и `end`. Так как все эти функции значительно (в десятки раз) замедляют работу программы, их можно отключить определив макрос `NDEBUG`. Из-за особенностей алгоритма заполнения элементов `Poison` значением, невозможно использовать макросы отладки на типах шаблона меньше `int`, это учтено в тестах.

2.3 Задание 3

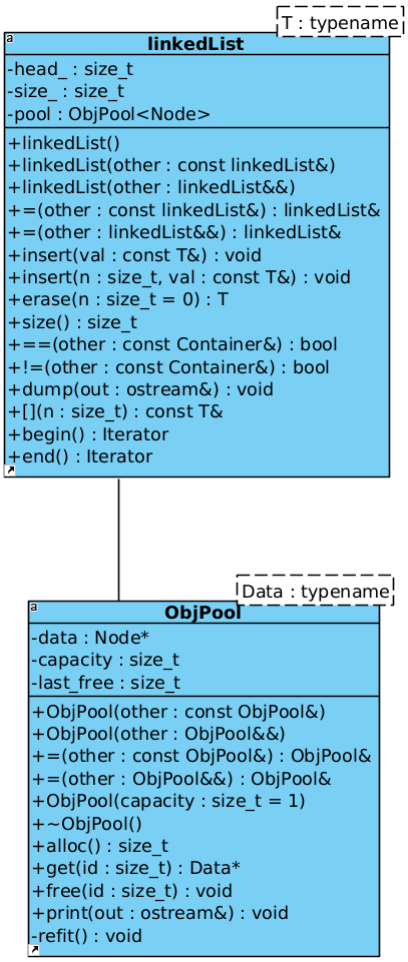

Рис. 3. UML 2.0 diagram for linkedlist class

Для решения данной задачи разработан класс `linkedlist`, базирующийся на `ObjPool`. Таким образом хранится не указатель на ноду, а id в `ObjPool`. В классе хранится id начала списка и его длинна как сервисное значение. Добавление и удаление из произвольного места работает через серию переходов к следующей ноде от начальной и вставляет элемент до ноды с заданой позицией. Сравнение списков осуществляется посимвольно через итераторы, которые из-за односвязности списка только `forward`. Конструкторы копирования и перемещения прото создают аналогичный пул и константы на вершины, после копирования новая структура будет полностью эквивалентна старой.

2.4 Задание 5

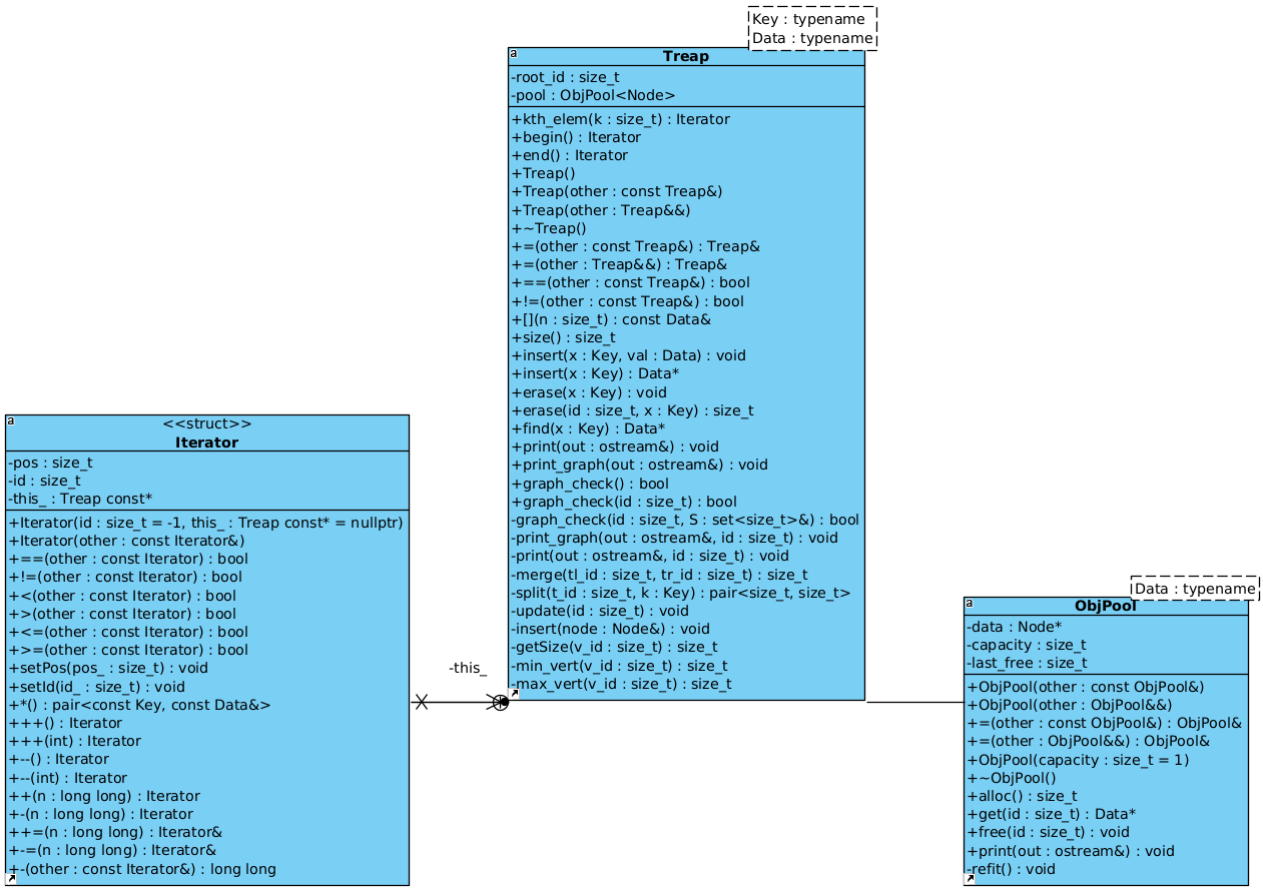

Рис. 4. UML 2.0 diagram for treap class

Для `treap` определена внутренняя структура ноды, хранящая необходимые сервисные значения, то есть индекс предков и детей, случайный приоритет, размер поддерева данной вершины, ключ и полезную нагрузку. Для оптимизации выделения и освобождения памяти используется класс `ObjPool` (object pool), представляющий из себя связанный список доступных для выделения нод `treap`. При аллоцировании занимается крайняя в списке нода, если в `ObjPool` заканчиваются свободные ноды, он переаллоцируется целиком на память в два раза больше. Так как при переаллоцировании ломаются все указатели на выделенные ноды, приходится использовать индексы. На декартовом дереве реализована поддержка `random_access_access_iterator`, функции вставки, поиска и удаления элементов по ключу. Помимо этого есть несколько сервисных функций облегчающих отладку и небольшой макрос того же свойства. Хранение размеров поддеревьев в нодах `treap` реализовано для расширения итераторов в `random_access`, последнее было дописано и работает за $O(\log n)$. Разыменование итератора возвращает пару из ключа и его полезной нагрузки. `Treap` как и `ObjPool` базируется на механике шаблонов, таким образом и декартово дерево и пул можно использовать в других проектах, что и было сделано в данной лабораторной.

3 Получение исполняемых модулей

Для всего проекта использовалась система сборки `cmake`. В конфигурации системы сборки прописаны пять режимов компиляции: basic, sanitizer, release, debug и coverage с разными наборами флагов компиляции (использование посредством флага `-DCMAKE_BUILD_TYPE=*`), в каждом из них прописано использование требуемого стандарта `c++20`, а для проверки на покрытие тестов специфицируется компилятор `clang++`, из-за отсутсвия универсального набора флагов у основных компиляторов. Помимо этого `cmake` автоматически скачивает с github и подключает библиотеку для unit тестов `GoogleTest`, которая используется для проверки корректности программ. Для каждого задания создан отдельный конфиг, который рекурсивно подключается в корневом `CMakeLists.txt`. Создан простой `python3` скрипт, парсящий результаты анализа покрытия и опускающий подробности касающиеся системных библиотек и `GoogleTest`. Так же в конфигурационный файл добавлена миловидная летающая капибара.

4 Тестирование

Тестирование производится при помощи библиотеки `GoogleTest`, везде, где позволяет логика используется многократный проход теста с генератором случайных значений и запуск шаблонной функции тестирования от нескольких различных базовых типов. Кроме этого, структуры на промежуточных стадиях сравниваются с эталонными из стандартной библиотеки. Анализ покрытия показывает не менее 97% задействованных строк кода на всех тестах, оставшиеся являются редкими случаями поддержания инвариантов на уровне ниже интерфейса.

4.1 Задание 1

//TODO сделать тесты...

4.2 Задания 2 и 4

4.2.1 Добавление и удаление

Вначале создается дек большой случайной длинны, путем последовательного добавления случайных элементов в начало и конец. После этого происходит удаление части элементов с обоих концов в случайной последовательности. В проверяется работа оператора доступа по индексу (`operator[]`).

4.2.2 Копирование и перемещение

Многократное копирование и перемещение случайно заполненных деков и проверка операций присваивания.

4.2.3 Итераторы

Итераторы проверяются на эквивалентность `operator[]` и корректность с точки зрения задуманых инвариантов, в том числе и с помощью собственных макросов.

4.2.4 Ручное изменение вместимости

Ручное изменение вместимости проверяется на корректность и отсутствие утечек. на случано заполненных деках большой случайной длинны.

4.3 Задание 3

4.3.1 Вставление с краев

Проверяется вставление элементов на крайних значениях концов списка.

4.3.2 Вставление и удаление с краев

Проверяется корректность вставления и удаления элементов на крайних значениях концов списка в случайном порядке.

4.3.3 Операторы

Немногочисленные операторы тестируются на на повторяемость результатов операций, корректность копирования и стрессоустойчивость. Как и на прочих этапах производится сравнение с эталонным вектором.

4.3.4 Копирование и перемещение

Многократное копирование и перемещение случайно сгенерированных списков и проверка операций присваивания.

4.4 Задание 5

4.4.1 ObjPool

`ObjPool` тестировался вручную при помощи красивых функций вывода состояния и проверки встроенной инвариантов на крайних значениях в предыдущей лабораторной.

4.4.2 Итераторы

Итераторы проверяются на работоспособность и эквивалентность индексации вектора стандартной библиотеки.

4.4.3 Вставка и удаление

Вставка и удаление проверяются на эквивалентность аналогичным операциям вектора `stl`.

4.4.4 Копирование и перемещение

Многократное копирование и перемещение случайно сгенерированных деревьев и проверка операций присваивания.

Приложение А

А.1 Файл `time.hpp`

//TODO добавить исходники...