Отчет по проделанной работе лабораторных работ №4, 5
Лазарева Владимира Александровича

Цель работы

Цель задания - отработать навыки разработки кода на языке Си с повышенными требованиями к качеству кода. В рамках разработки используются такие техники, как статический и динамический анализ кода, модульное тестирование, формальная спецификация, дедуктивная верификация. Требуется реализовать небольшой программный модуль на языке Си согласно варианту задания и доказать отсутствие в нем ошибок времени исполнения и доказать выполнение функциональных требований. Задание состоит из нескольких этапов. На первом этапе пишется реализация модуля. На втором этапе применяется статический и динамический анализ кода, а также модульное тестирование. На третьем этапе формально записываются функциональные требования. На последних этапах используется дедуктивная верификация.

Этап 1. Реализация.

В ходе первого этапа были реализованы следующие методы

- int initializeMap(Map *map, int size) - Функция `initializeMap()` создаёт пустой ассоциативный массив с заданным числом максимально хранимых элементов `size`, выделяя под него динамическую память. На вход функции должен подаваться указатель на переменную-структуру, функция должна проинициализировать эту структуру. В случае неуспеха, функция должна вернуть ненулевое число, иначе функция должна вернуть 0.;
- void finalizeMap(Map *map) - Функция `finalizeMap()` освобождает динамическую память, используемую для ассоциативного массива `map`. На вход функции должен подаваться указатель на область памяти, проинициализированную функцией `initializeMap()`;
- int addElement(Map *map, Key *key, Value *value) - Функция `addElement()` добавляет в заданный ассоциативный массив `map` отображение заданного ключа `key` на заданное значение `value` и возвращает истину (единицу), если в нём было место для этого отображения. Если в ассоциативном массиве было недостаточно места, функция возвращает ложь (ноль). Функция не меняет переданные ключ и значение. Функция не добавляет и не удаляет другие отображения, кроме отображения по заданному ключу, если оно было. Функция имеет право изменять структуру ассоциативного массива, если это не отражается на содержащихся в нём парах. Ничего другого функция не делает.;
- int removeElement(Map *map, Key *key, Value *value) - Функция `removeElement()` удаляет сохранённое в ассоциативном массиве `map` значение по заданному ключу `key` (если оно там было). Функция не удаляет другие отображения, кроме отображения по заданному ключу, а также не добавляет новые отображения. Функция возвращает истину (единицу), если функция изменила ассоциативный массив, ложь (ноль) в противном случае. В случае, если значение было удалено и при этом переданный указатель `value` был отличен от нулевого, функция записывает значение, содержавшееся для заданного ключа, по данному указателю. Функция имеет право изменять структуру ассоциативного массива, если это не отражается на содержащихся в нём парах. Ничего другого функция не делает. ;
- int getElement(Map *map, Key *key, Value *value) - Функция `getElement()` возвращает по указателю `value` сохранённое в ассоциативном массиве `map` значение для

заданного ключа key и возвращает истину (единицу), если заданный ассоциативный массив имеет отображения для заданного ключа. В случае, если значение по заданному ключу не содержится в отображении, возвращается ложь (ноль) и ничего другого не происходит. Функция не меняет ассоциативный массив и переданный ключ. Ничего другого функция не делает.

И вспомогательные:

- long hash(Key *key) — расчет хеша;
- int getCalculatedIndex(Map *map, long hashValue, int index) — получение индекса элемента (из-за линейного пробирования).

Этап 2. Анализ кода и тестирование.

Особенности реализации:

Поиск/добавление/удаление ключа в ассоциативном массиве реализован как описано в Варианте B (хеш-таблица с
линейным пробированием).

Используется цикл по хеш-таблице, внутри которого вычисляется индекс ячейки. Это все последствия линейного пробирования.

1.1. Добавление

Если попытаться добавить элемент с тем же ключом, то ничего не выйдет. По условию уточняется, что таблица может
содержать только одно отображение. При попытке повторного добавления значения с таким же ключом будет
код возврата - 0.

При выявлении такого индекса ячейки, что existent != 1, в нее записывается key/value пара.

Если ячеек с existent == 0 не найдено, выбрасываем 0, в случае успеха - 1.

1.2. Удаление

В самом начале на основе хеша ключа рассчитывается первый потенциальный индекс ячейки. Если по этой ячейке existent == 1 и ключ совпадает,
то запись помечается как existent = 0, count уменьшается, ячейка становится доступна для записи. Если по заданному индексу нет такой ячейки,
то проходим по всей таблице в поисках ее. В случае, если не найдено выбрасываем 0, в случае успеха - 1.

1.3. Получение

Если existent ячейки == 1 и ключи идентичные, то в value передается значение ячейки, код возврата - 1.

В случае полного прохода по циклу (ячейка с данным идексом + ключом не найдена) код возврата — 0.

В ходе этого этапа был разработан модуль тестирования test.c, в котором было смоделировано поведения данной Мары

Тестирование проводилось при помощи следующих иструментов:

2.1. Статический анализ

* Clang ('scan-build clang --analyze map.c',
'scan-build clang -fsanitize=address map.c')
* Splint ('{PATH_TO_SPLINT_BINARY} -weak map.c')
* Cppcheck ('cppcheck map.c')

2.2. Динамический анализ
* Valgrind

2.3. Файл с тестами — tests.c

Этап 3. Формальная спецификация.

Перед выполнением данного этапа был получен пул методов, для которых было необходимо написать спецификации по требованиям, но не по условиям. В рамках моей задачи этими методами являются: initializeMap, removeElement, getElement. Для каждого из методов была составлена спецификация и трассировка, где каждому требованию из постановки соответствует как минимум один requiers/ensurense.

Этап 4. Доказательство safety.

На данном этапе пришлось устранять ошибки переполнения inta (обошел данный момент тем, что хэш будет типа long, а при расчете индекса приводиться обратно в int). Были проблемы с валидными указателями в массиве мапы — решением этого является вставка assert внутри методов (как проверка на валидность/корректность мапы, ограничение границ индексов, ограничение значения хеша).

Этап 5. Доказательство behavior.

На данном этапе были добавлены еще asserты и ensures для покрытия всех ветвей выполнения методов.

Выводы:

Написание спецификации и доказательство полной корректности — дело циклическое, постоянно что-то дописывается в предикаты, аксиомы, пред- и постусловия.

Солверы не всегда успевают доказать все за 5 сек и малым количеством памяти — лучше всего ее увеличить.

Assert внутри методов позволяют проверять на ходу значения тех или иных переменных, благодаря чему могут отброситься солверами некорректные пути.

Работать с памятью, метками, структурами и классами куда проще при помощи frama-c, нежели писать спецификации на WhyML.