

# Лабораторная работа № 3 по курсу дискретного анализа: исследование качества программ

Выполнил студент группы 08-207 МАИ *Сысоев Максим.*

## Условие

Для реализации словаря из предыдущей лабораторной работы, необходимо провести исследование скорости выполнения и потребления оперативной памяти. В случае выявления ошибок или явных недочётов, требуется их исправить. Результатом лабораторной работы является отчёт, состоящий из:

1. Дневника выполнения работы, в котором отражено что и когда делалось, какие средства использовались и какие результаты были достигнуты на каждом шаге выполнения лабораторной работы.
2. Выводов о найденных недочётах.
3. Сравнение работы исправленной программы с предыдущей версией.
4. Общих выводов о выполнении лабораторной работы, полученном опыте.

Минимальный набор используемых средств должен содержать утилиту gprof и библиотеку dmalloc, однако их можно заменять на любые другие аналогичные или более развитые утилиты (например, Valgrind или Shark) или добавлять к ним новые (например, gcov)

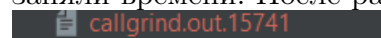
## Метод решения

В рамках исследования качества программы я буду использовать следующие утилиты:

- Анализ времени работы: callgrind и gprof2dot(для визуализации)
- Анализ покрытия тестами: gcov built-in CLion
- Анализ потребления памяти: valgrind

## Анализ времени работы: callgrind и gprof2dot(для визуализации)

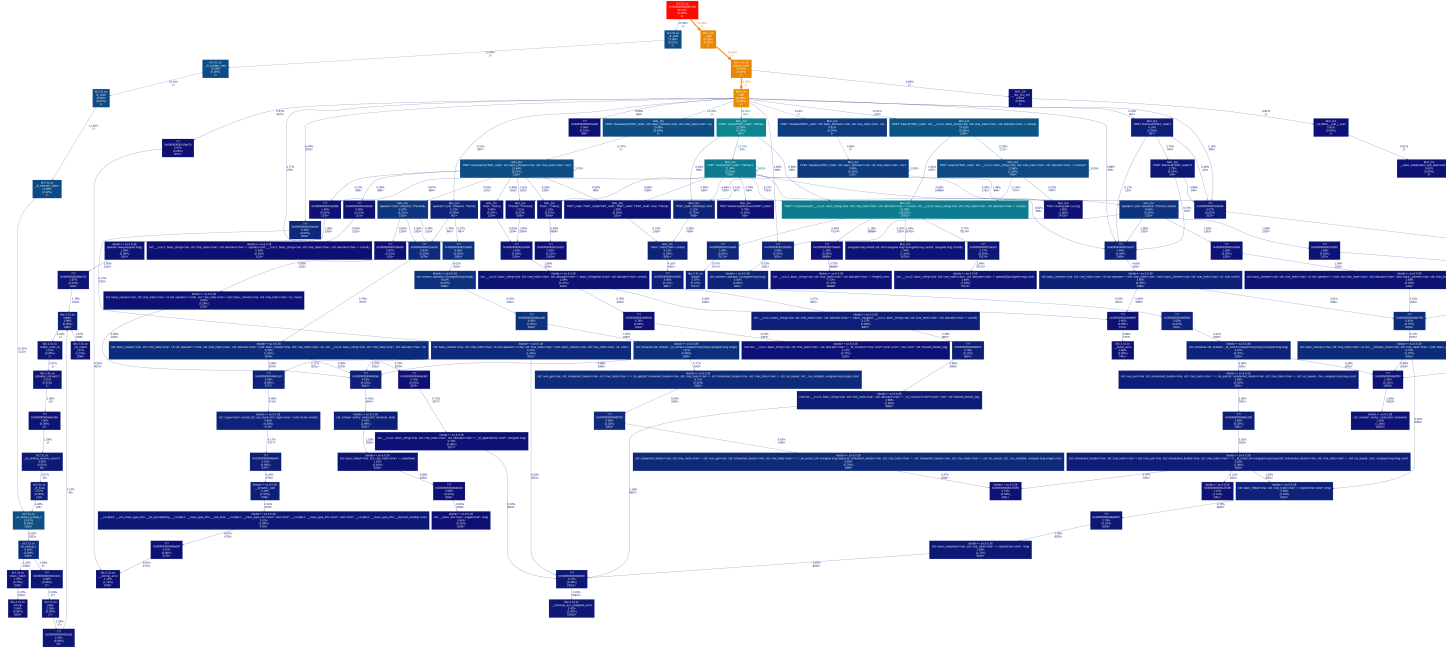
Хотелось попробовать неклассические утилиты, а современные и используемые. У valgrind есть специальный инструмент для анализа времени работы программы, которая покажет сколько раз вызывались функции, сколько в процентном соотношении они заняли времени. После работы callgrind вернёт файл в особом формате:



```
callgrind.out.15741
```

На конце pid. Для анализа нужно использовать какой-нибудь сторонний инструмент.

Самым известным является `kcachegrind`. Однако, у меня возникли проблемы с данной утилитой, поэтому пришлось искать другое решение. `gprof2dot`. Это скрипт, который построит граф вызовов функций (причём он будет полным, в отличие от `kcachegrind`).



Картинку в полном размере можно посмотреть [тут](#).

Читать график просто: На вершине указывается следующая информация:

function name

total time % ( self time % )

total calls

Где self time - время, затраченную только на эту функцию.

total time - время, затраченную на эту функцию и все дочерние элементы.

total calls - количество вызовов данной функции (включая рекурсивные).

На ребре пишут следующую информацию:

total time %

calls

где calls: это количество вызовов родительской функции, вызываемой дочерними элементами

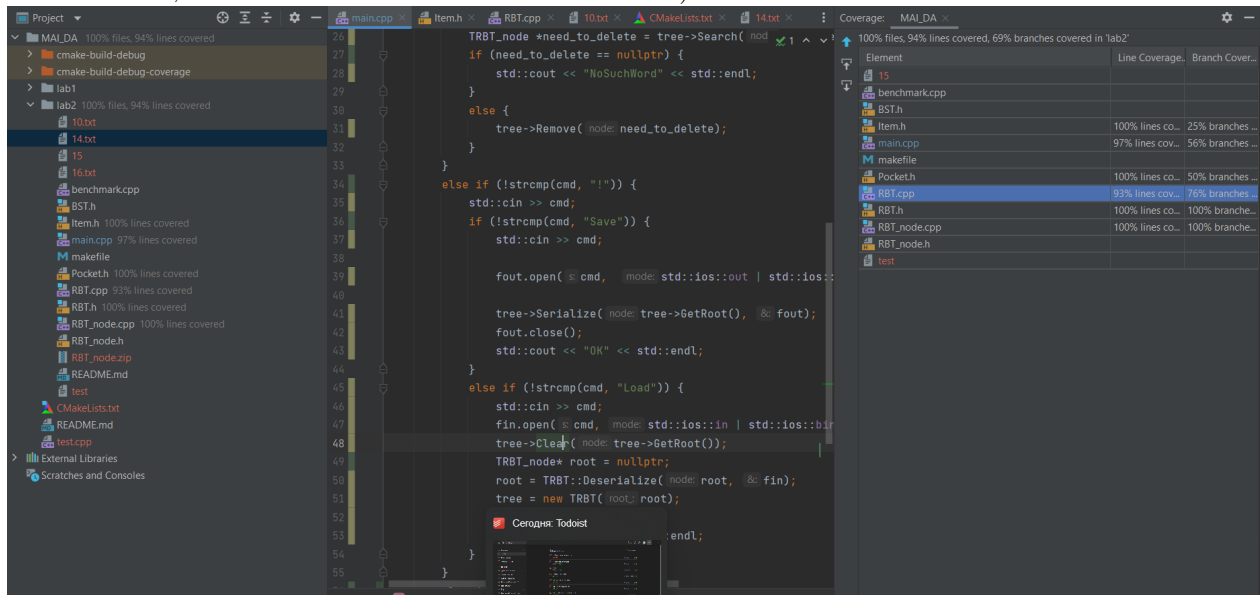
total time: является ли процент времени выполнения, переданного от дочерних элементов этому родительскому элементу (если таковой имеется)

Как можно заметить, довольно большую часть времени забрал один вызов функции десериализации дерева. Оно и понятно: считывание информации, множественные вызовы `new`, рекурсия. Можно было заменить рекурсию на стек для ускорения, однако основную часть времени всё равно забирают `new`, а потому эта функция и должна быть

"медленной". К сожалению, нынешних знаний не хватает для устранения проблемы.

## Анализ покрытия тестами: gcov

Одной из основной программой для проверки кода на покрытие в IDE CLion является утилита gcov. При помощи среды разработки можно в режиме написания кода просмотреть, какие строки сколько раз исполнялись (красные — 0, желтые — частично вызывается, зеленые — исполняется постоянно):



Эта утилита помогает понять, какие функции вызываются, какие нет. Можно заметить, что покрытие одним из тестов (1000 запросов) равно 93% для основной функционального файла RBT. Если программа отработала верно на этом тесте, значит, она также правильно отработает для любых входных данных, использующих эти функции.

## Анализ потребления памяти: valgrind

Когда говорят "проверить на утечки" де-факто имеют ввиду запустить valgrind. Использовать буду инструмент memcheck. Данная утилита покажет, сколько мы аллоцировали памяти, сколько освободили. В случае утечки укажет, в какой строчке была выделена память, впоследствии не освободившаяся. Это очень удобно! Запустив valgrind на своей программе, я обнаружил следующее:

```

==16775==
==16775== HEAP SUMMARY:
==16775==      in use at exit: 16 bytes in 2 blocks
==16775==    total heap usage: 1,363 allocs, 1,361 frees, 189,584 bytes allocated
==16775==
==16775== 8 bytes in 1 blocks are definitely lost in loss record 1 of 2
==16775==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==16775==    by 0x10A5F1: main (main.cpp:15)
==16775==
==16775== 8 bytes in 1 blocks are definitely lost in loss record 2 of 2
==16775==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==16775==    by 0x10A932: main (main.cpp:51)
==16775==
==16775== LEAK SUMMARY:
==16775==    definitely lost: 16 bytes in 2 blocks
==16775==    indirectly lost: 0 bytes in 0 blocks
==16775==    possibly lost: 0 bytes in 0 blocks
==16775==    still reachable: 0 bytes in 0 blocks
==16775==    suppressed: 0 bytes in 0 blocks
==16775==
==16775== For lists of detected and suppressed errors, rerun with: -s
==16775== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

```

Утечка памяти в 16 байт. Данная проблема возникла из-за особенности архитектуры моей программы: отдельно структура дерева, хранящая указатель на ноду root, и сама нода. При выполнении запроса `TRBT::Clear()` -> будет удалено всё дерево, но указатель tree удалён не будет. Он указывает на удалённую область памяти, но сам по-прежнему валиден. Благодаря valgrind, можно заметить, в каких местах была выделена память, что позволит быстро решить эту проблему.

## Выводы

Чем больше программа, тем сложнее использовать "бумажные" методы анализа эффективности программы. Без инструментов уже не обойтись. В рамках данной работы были изучены утилиты для анализа времени, покрытия и памяти - основные составляющие программы. Анализ времени позволил увидеть, сколько времени занимает каждая из функций, что позволит направить мысли об оптимизации в нужное русло. Анализ покрытия позволил понять состоятельность теста. Лучший тест тот, который покрывает весь тестируемый код, учитывающий при этом краевые случаи. Анализ потребления памяти позволил выявить неочевидную утечку памяти, которая была успешно исправлена. Работа очень понравилась: после неё таким образом буду тестировать весь большой код.