Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: А. А. Почечура
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2022

Лабораторная работа №3

Задача: Необходимо провести исследование скорости выполнения и потребления оперативной памяти. В случае выявления ошибок или явных недочётов, требуется их исправить.

В данной работе для оценки потребления оперативной памяти будет использована утилита valgrind, а для оценки скорости работы программы - утилита gprof.

1 Описание

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. Общих выводов о выполнении лабораторной работы, полученном опыте.

2 Дневник выполнения работы

Основные этапы создания программы:

1. Реализация необходимых алгоритмов.

2. Выявление логических ошибок в коде программы.

3. Выявление утечек памяти.

4. Выявление неэффективно работающих участков кода.

3 Используемые средства

1 Valgrind

Valgrind – это инструмент для отслеживания утечек памяти (и не только), который удобно использовать при работе с терминалом. Для его использования необходимо скомпилировать программу и вызвать valgrind от исполняемого файла. Эта утилита не только сообщит о наличии утечек, но и покажет, при исполнении какой функции они произошли. Valgrind сам подсказывает, какие ключи использовать для более наглядного отчёта об ошибках.

```
root@DESKTOP-5HM2HTK:~# valgrind ./a.out <tests/1.t
==1305== Memcheck,a memory error detector
==1305== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
```

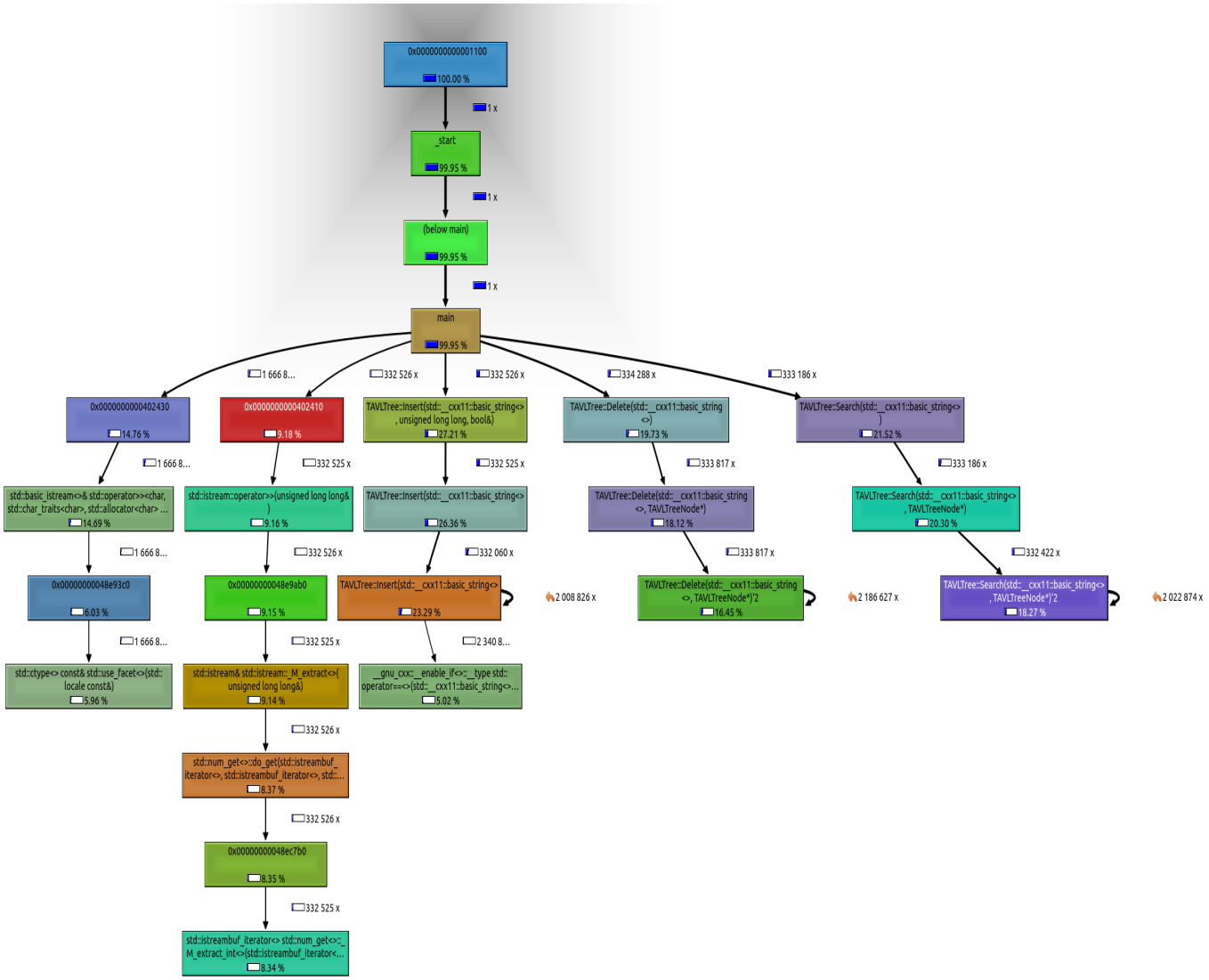
```
==1305== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1305== Command: ./a.out
==1305==
...
==1301==
==1301== HEAP SUMMARY:
==1301==     in use at exit: 122,880 bytes in 6 blocks
==1301==   total heap usage: 39,358 allocs,39,352 frees,3,028,856 bytes allocated
==1301==
==1301== LEAK SUMMARY:
==1301==    definitely lost: 0 bytes in 0 blocks
==1301==    indirectly lost: 0 bytes in 0 blocks
==1301==      possibly lost: 0 bytes in 0 blocks
==1301==    still reachable: 122,880 bytes in 6 blocks
==1301==         suppressed: 0 bytes in 0 blocks
==1301== Rerun with --leak-check=full to see details of leaked memory
==1301==
==1301== For lists of detected and suppressed errors,rerun with: -s
==1301== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@DESKTOP-5HM2HTK:~#
```

Как видно из сводки valgrind, моя программа очищает всю задействованную оперативную память даже на больших входных данных.

2 Callgrind

Эта утилита входит в состав инструмента valgrind. Она эмулирует каждую исполняемую инструкцию программы и на основании внутренних метрик о «стоимости» работы каждой инструкции выдает нужное нам заключение.

Чтобы использовать утилиту callgrind, нужно собрать программу с ключами -g и -no-pie. В папке запуска сгенерируется отчёт с именем «callgrind.out.<номер_процесса>».
Чтобы можно было быстро работать с отчётом, была задействована программа KCachegrind.
При выборе нужной функции, на экране отобразится цепочка функций, вызывающих выбраную и вызываемых выбранной.
Граф для моей реализации дерева:

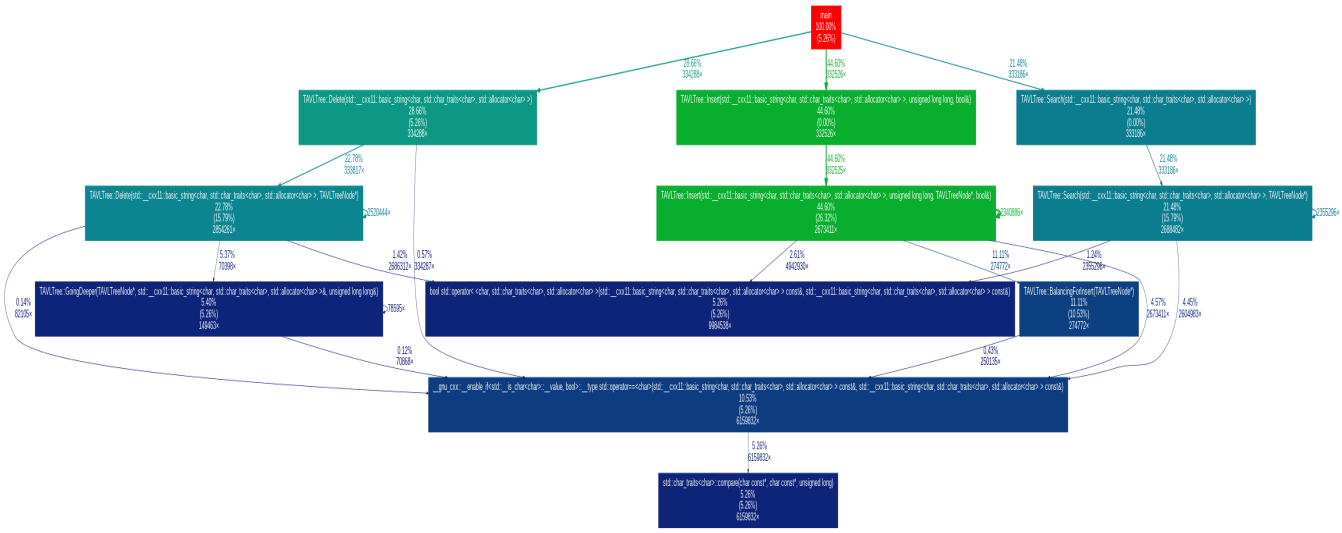


По нему мы можем заметить, что больше всего времени тратиться для выполнения функции добавления в дерево. Это связано с тем, что при добавлении элемента в $AVL-$дерево происходит ребалансировка, которая требует дополнительного времени.

3 Gprof

Профилировщик от google работает по другому принципу. Вместо того, чтобы анализировать каждую инструкцию исполняемой программы, он приостанавливает выполнение программы через равные промежутки времени и пытается определить, в какой функции в данный момент находится за счёт раскрутки стека. В результате,

это почти не влияет на производительность запущенного приложения. Но у такого подхода есть и свои слабые стороны. Для удобства анализа полученного отчёта можно его преобразовать в графическую форму с помощью утилиты gprof2dot, которая может преобразовывать его в формат .png или .svg.



Полученное дерево в результате работы этой программы показывает примерно те же результаты. Функция добавления по-прежнему занимает больше времени, чем другие функции. В данном отчёте gprof включил работу аллокатора и некоторых других функций в работы основных трёх функций: удаление элемента, добавление элемента и поиск.

4 Выводы

Выполнив третью лабораторную работу по курсу «Дискретный анализ», я научился пользоваться утилитами gprof и callgrind, а также улучшил свои практические знания об утилите valgrind. Это полезные знания, которые помогут мне оптимизировать программы. Добавление работает в моей реализации $AVL-$дерева дольше всего. Это может быть связано с тем, что для удаления я более детально разобрал различные случаи, из-за чего программа делает меньше лишних шагов.

Список литературы

[1] *Valgrind.*
URL: `https://ru.wikipedia.org/wiki/Valgrind` (дата обращения: 19.05.2022).

[2] *Gprof.*
URL: `https://en.wikipedia.org/wiki/Gprof` (дата обращения: 19.05.2022).