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

Выполнил студент группы М80-208Б-20 Морозов Артем Борисович.

Условие

Для реализации словаря из предыдущей лабораторной работы, необходимо провести исследование скорости выполнения и потребления оперативной памяти. В случае выявления ошибок или явных недочетов, требуется их исправить.

Дневник выполнения работы

Для выполнения лабораторной работы я воспользовался двумя утилитами: **gprof** и **valgrind**.

Для начала я запущу свою программу с утилитой **gprof**. Утилита gprof хороша тем, что позволяет увидеть время работы всех функций, реализованных в программе, количество их вызовов и вычисляет процентное соотношение работы конкретной функции по сравнению с работой всей программы.

Скомпилируем при помощи команды **g++ task2.cpp -pg -o main**, где -pg – специальный флаг, позволяющий вывести необходимые данные в файл gmon.out.

Предварительно сгенерируем файл **gprof_test.txt**, состоящий из 1000000 команд на поиск, вставку и удаление. После – запустим нашу программу с этим файлом: **./main < gprof_test.txt**

После компиляции создался файл gmon.out, который является выводом нашей утилиты. Просмотрим его при помощи следующей команды: **gprof main.**

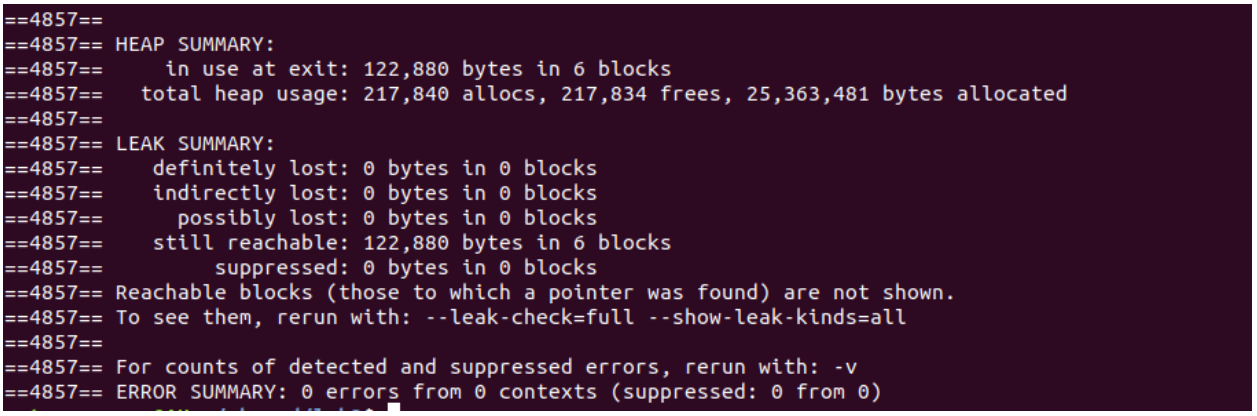
Результат вывода утилиты **gprof** следующий:

```
 time   seconds   seconds    calls  ms/call  ms/call  name
 22.22      3.57     3.57  3000000     0.00     0.00  StringBitFunctions::ToLower(std::__cxx11
 20.60      6.88     3.31  1000000     0.00     0.00  PatriciaTrie::Patricia::Search(std::__cx
 20.41     10.17     3.28  1000000     0.00     0.01  PatriciaTrie::Patricia::Insert(std::__cx
 16.80     12.87     2.70  1000000     0.00     0.00  PatriciaTrie::Patricia::Erase(std::__cxx
 13.38     15.02     2.15 167586480     0.00     0.00  StringBitFunctions::GetIndexBit(std::__
  4.23     15.70     0.68   992327     0.00     0.00  StringBitFunctions::FirstDifferentBit(st
r, std::char_traits<char>, std::allocator<char> >&)
  1.21     15.90     0.20 34535001     0.00     0.00  unsigned long const& std::max<unsigned l
  0.28     15.94     0.05  3999998     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<ch
::allocator<char> > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
  0.25     15.98     0.04  6000000     0.00     0.00  bool std::operator==<char, std::char_tra
> > const&, char const*)
  0.25     16.02     0.04                             main
  0.12     16.04     0.02        1    20.01    20.01  PatriciaTrie::Patricia::~Patricia()
  0.12     16.06     0.02                             frame_dummy
  0.09     16.08     0.02  2007840     0.00     0.00  std::char_traits<char>::compare(char con
  0.09     16.09     0.02   992328     0.00     0.00  PatriciaTrie::PNode::PNode()
```

Остальные функции показывали 0.00 в графе time.

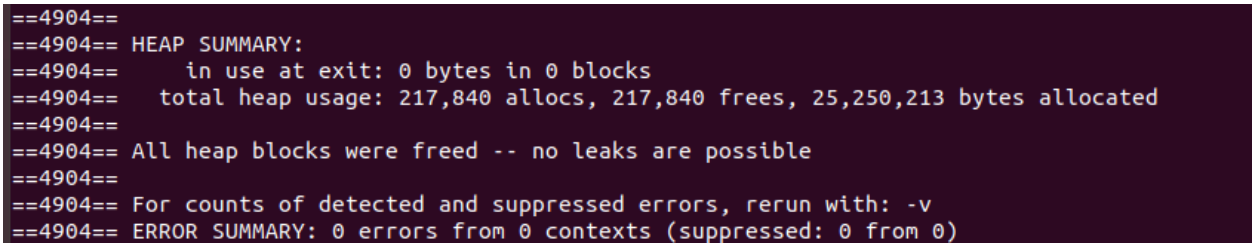
Как мы можем увидеть, на тесте в 1000000 строк для каждого метода, наибольшее время выполнения у всех трех основных операций с деревом: вставкой, поиском и удалением, а также у всех основных действий со строками: ToLower, GetIndexBit и FirstDifferentBit. Вероятно, это связано с тем, что на многие строки, подающиеся в дерево, могут быть и длиной в 256 символов, что довольно-таки много.

Давайте теперь запустим нашу программу через **valgrind** – одной из самых лучших утилит для поиска всевозможных утечек памяти в программе. Проверять будем со специальным флагом –leak-check=full:



Во многих источниках пишут, что, если в definitely lost 0 bytes и ERROR SUMMARY: 0 errors, то утечек памяти нет. Параметр still reachable, по мнению многих, означает, что указатель не утерян и может быть освобожден, но на данный момент не освобожден. Если верить подобным людям, то, по идее, уже на данном этапе утечек в программе нет, однако давайте попробуем убрать строку still reachable и сделать количество frees равным количеству allocs.

Немного поизучав различные источники, я пришел к выводу, что still reachable могут вызывать так называемые “строчки-ускорители”: **ios:: sync_with_stdio(false), cin.tie(0), cout.tie(0).** Закомментировав их, я полностью избавился от каких-либо дополнительных сообщений valgrind:



Вывод о найденных недочетах

После использования двух утилит недочетов не было обнаружено. Единственное, что могло бы условно считаться недочетом – это вышеупомянутые блоки still

reachable, однако это было исправлено при помощи комментирования “строк-ускорителей”. Ответ, почему эти строки влияют на неосвобождение некоторых блоков памяти, дает нам стандарт языка C++ - он просто говорит, что когда программа выходит из потока, сами объекты - и, следовательно, используемые ими буферы - не будут уничтожены.

Сравнение работы исправленной программы с предыдущей версией

Так как в исправленной программе были закомментированы, а не исправлены, всего лишь 3 строчки исходной программы, связанные с ее ускорением, то очевидно, что старая версия будет работать гораздо быстрее новой. Однако давайте в этом убедимся точно на каком-нибудь тесте – скажем, в 5000 строк на удаление, вставку и добавление.

Результат работы со “строками-ускорителями”:

```
Insertion time in PATRICIA: 19 ms
Searching time in PATRICIA: 18 ms
Erasing time in PATRICIA: 14 ms
```

Результат работы без “строк-ускорителей”:

```
Insertion time in PATRICIA: 140 ms
Searching time in PATRICIA: 151 ms
Erasing time in PATRICIA: 105 ms
```

Общие выводы о выполнении лабораторной работы

Лабораторная работа №3 по дискретному анализу помогла мне закрепить навыки работы с утилитой для нахождения утечек памяти valgrind, а также познакомила меня с очень интересной утилитой gprof, которая помогает пользователю увидеть время выполнения отдельных функций, что очень полезно для огромных программ. Помимо этого, я смог добиться полного очищения выделенной памяти.