Лабораторная работа №3 по курсу дискретного анализа: Исследование качества программ

Выполнил студент группы М8О-212Б-22 МАИ *Корнев 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:

- Дневника выполнения работы, в котором отражено что и когда делалось, какие средства использовались и какие результаты были достигнуты на каждом шаге выполнения лабораторной работы.
- Выводов о найденных недочётах.
- Сравнение работы исправленной программы с предыдущей версией.
- 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 выполнения лабораторной работы я буду использовать следующие утилиты:

- Анализ времени работы: gprof
- Анализ потребления памяти: valgrind

gprof

Утилита gprof позволяет измерить время работы всех функций в программе, а также количество их вызовов и долю от общего времени работы программы в процентах.

Для работы с утилитой gprof было необходимо скомпилировать программу с ключом `-pg` После запуска полученного исполняемого файла появился файл gmon.out, в котом содержалась информация предоставленная для анализа программы. Далее этот файл был обработан gprof для получения текстового файла с подробной информацией о времени работы и вызовах всех функций и операторов, которые использовались в программе. Вывод программы:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 28.57      0.02     0.02   526864     0.00     0.00  AVLTree::Search(std::__
 14.29      0.03     0.01        1    10.00    10.00  Node::InsertNode(AVLTre
 14.29      0.04     0.01                             Node::Print(int)
  0.00      0.04     0.00      216     0.00     0.00  _init
  0.00      0.04     0.00       44     0.00     0.00  AVLTree::InsertNode(std

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
```

```
ms/call    function per call, if this function is profiled,
     else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
     function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
     the function in the gprof listing. If the index is
     in parenthesis it shows where it would appear in
     the gprof listing if it were to be printed.
```



```
          Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 25.00% of 0.04 seconds


index % time    self  children    called     name
                0.00    0.00   33458/526864      Node::Print(int) [4]
                0.00    0.00   66542/526864      AVLTree::Load(std::basic_ifst
                0.02    0.00  426864/526864      AVLTree::Clear() [2]
[1]     50.0    0.02    0.00  526864            AVLTree::Search(std::__cxx11::ba
-----------------------------------------------

                                                  <spontaneous>
[2]     40.5    0.00    0.02                    AVLTree::Clear() [2]
                0.02    0.00  426864/526864      AVLTree::Search(std::__cxx11
-----------------------------------------------

                                                  <spontaneous>
[3]     31.3    0.00    0.01                    AVLTree::Load(std::basic_ifstrean
                0.01    0.00       1/1          Node::InsertNode(AVLTree*, s
                0.00    0.00   66542/526864      AVLTree::Search(std::__cxx11
-----------------------------------------------
```

```
                                                <spontaneous>
[4]      28.2    0.01    0.00                 Node::Print(int) [4]
                 0.00    0.00   33458/526864      AVLTree::Search(std::__cxx11
-----------------------------------------------------------------------------
                                    5434          Node::InsertNode(AVLTree*, s
                 0.01    0.00       1/1           AVLTree::Load(std::basic_ifst
[5]      25.0    0.01    0.00       1+5434     Node::InsertNode(AVLTree*, std::
                                    5434          Node::InsertNode(AVLTree*, s
-----------------------------------------------------------------------------
                                    1270          _init [11]
                 0.00    0.00     216/216         Node::Node(std::pair<std::__
[11]     0.0     0.00    0.00     216+1270    _init [11]
                 0.00    0.00      44/44          AVLTree::InsertNode(std::pai
                                    1270          _init [11]
-----------------------------------------------------------------------------
                 0.00    0.00      44/44          _init [11]
[12]     0.0     0.00    0.00      44          AVLTree::InsertNode(std::pair<st
-----------------------------------------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

index A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so it is easier to look up where the function is in the table.

% time This is the percentage of the 'total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '<spontaneous>' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

```
 called  This is the number of times the function called
this child '/' the total number of times the child
was called.  Recursive calls by the child are not
listed in the number after the '/'.

 name  This is the name of the child.  The child's index
number is printed after it.  If the child is a
member of a cycle, the cycle number is printed
between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole.  This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The '+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.


Copyright (C) 2012-2022 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.


Index by function name

   [5] Node::InsertNode(AVLTree*, std::pair<std::__cxx11::basic_string<char,
   [4] Node::Print(int)            [1] AVLTree::Search(std::__cxx11::basic_string
```

Операции с AVL-деревьями в графе вызовов указывают на конкретные методы, используемые для манипуляции AVL-деревьями, такие как InsertNode, RemoveNode, LeftRotate и RightRotate. Это дает лучшее понимание того, как реализована структура AVL-дерева в коде. Рекурсивные вызовы также указаны в графе вызовов, хотя их влияние на производите не всегда очевидно из-за отсутствия информации о времени. Из этих данных можно сделать вывод, что программа включает существенные манипуляции со строками и преобразова данных. Лямбда-функции и рекурсивные элементы указывают на определенный уровень сложности в структуре программы, с возможным использованием функциональных стилей программирования. Несмотря на подробные профилировочные данные, явные проблемы с производительностью не обнаружены. Это может означать, что программа либо эффективна, либо профилировка не смогла выявить наиболее трудоемкие операции.

Valgrind

Valgrind является утилитой для поиска ошибок в работе с памятью в программе, таких как утечки памяти и выход за границу массива. Результат работы `Valgrind`:

```
==47438== Memcheck, a memory error detector
==47438== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==47438== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==47438== Command: ./main
==47438== HEAP SUMMARY:
==47438== in use at exit: 122,880 bytes in 6 blocks
==47438== total heap usage: 15 allocs, 9 frees, 196,818 bytes allocated
==47438==
==47438== LEAK SUMMARY:
==47438== definitely lost: 0 bytes in 0 blocks
==47438== indirectly lost: 0 bytes in 0 blocks
==47438== possibly lost: 0 bytes in 0 blocks
==47438== still reachable: 122,880 bytes in 6 blocks
==47438== suppressed: 0 bytes in 0 blocks
==47438== Rerun with -leak-check=full to see details of leaked memory
==47438==
==47438== For lists of detected and suppressed errors, rerun with: -s
==47438== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Сообщение `still reachable: 122,880 bytes in 6 blocks` происходит из-за использования `ios::sync_with_stdio()`. Я его использовал для ускорения работы программы и это не является утечкой памяти. В результате использования утилиты Valgrind утечки памяти не выявлены — программа работает с памятью корректно.

Выводы

В ходе выполнения практической работы я овладел методами выявления узких мест и проблем в управлении памятью. Использование утилиты gprof для анализа времени выполнения программы помогает выявить функции, затрачивающие больше всего времени, что, в свою очередь, способствует оптимизации кода с целью улучшения производительности. Применение valgrind для анализа работы с памятью позволяет выявить утечки памяти, неправильное использование указателей и другие проблемы, которые могут привести к непредсказуемому поведению программы.