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

Выполнил студент группы М8О-208Б-20 *Ядров Артем*.

Условие

Кратко описывается задача:

1. Для реализации словаря из предыдущей лабораторной работы необходимо провести исследование скорости выполнения и потребления оперативной памяти.

Метод решения

зучение утилит для исследования качества программ таких как gcov, gprof, valgrind, и их использование для оптимизации программы.

Valgrind

Valgrind — инструментальное программное обеспечение, предназначенное для отладки использования памяти, обнаружения утечек памяти, а также профилирования.

В ходе выполнения лабораторной работы утилита будет использована исключительно для отладки использования памяти.

```
==10756==
==10756== HEAP SUMMARY:
==10756==     in use at exit: 125,255 bytes in 56 blocks
==10756==   total heap usage: 145 allocs, 89 frees, 278,123 bytes allocated
==10756==
==10756== LEAK SUMMARY:
==10756==    definitely lost: 320 bytes in 5 blocks
==10756==    indirectly lost: 2,055 bytes in 45 blocks
==10756==      possibly lost: 0 bytes in 0 blocks
==10756==    still reachable: 122,880 bytes in 6 blocks
==10756==         suppressed: 0 bytes in 0 blocks
==10756== Rerun with --leak-check=full to see details of leaked memory
==10756==
==10756== For lists of detected and suppressed errors, rerun with: -s
==10756== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Как видим, Valgrind не обнаружил утечек памяти. Можно спать спокойно. Однако, существует существенный недостаток утилиты: она не отслеживает выход за границы выделенной памяти. Поэтому я стараюсь использовать средство fsanitize.

gprof

Gprof - это инструмент для профилирования программы. Мы можем отследить, где и сколько времени проводила программа, тем самым выявляя слабые участки.

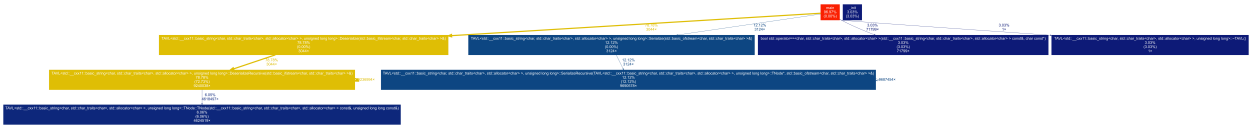
Возьмем достаточно большой тест и применим утилиту gprof.

```
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
75.00      0.24     0.24     3044     0.08     0.09  DeserializeRecursive(std::basic_ifstream<ch
12.50      0.28     0.04     3124     0.01     0.01  SerializeRecursive(TAVL<std::__cxx11::basic
6.25       0.30     0.02  4624519     0.00     0.00  TNode(std::__cxx11::basic_string<char, std::
3.12       0.31     0.01                             _init
1.56       0.32     0.01    71799     0.00     0.00  bool std::operator==<char, std::char_traits<
1.56       0.32     0.01        1     5.00     5.00  ~TAVL()
0.00       0.32     0.00   410758     0.00     0.00  GetHeight(TAVL<std::__cxx11::basic_string<ch
0.00       0.32     0.00   360080     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__c
0.00       0.32     0.00   225446     0.00     0.00  bool std::operator><char, std::char_traits<c
0.00       0.32     0.00   180040     0.00     0.00  bool __gnu_cxx::operator!=<char*, std::__cxx
0.00       0.32     0.00   161208     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__c
0.00       0.32     0.00   161208     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__c
0.00       0.32     0.00   159871     0.00     0.00  bool std::operator< <char, std::char_traits<
0.00       0.32     0.00   137829     0.00     0.00  Max(short, short)
0.00       0.32     0.00   137829     0.00     0.00  FixHeight(TAVL<std::__cxx11::basic_string<ch
0.00       0.32     0.00    67550     0.00     0.00  BFactor(TAVL<std::__cxx11::basic_string<char
0.00       0.32     0.00    64738     0.00     0.00  Balance(TAVL<std::__cxx11::basic_string<char
0.00       0.32     0.00    18832     0.00     0.00  FindTree(TAVL<std::__cxx11::basic_string<cha
0.00       0.32     0.00     6360     0.00     0.00  Find(std::__cxx11::basic_string<char, std::c
0.00       0.32     0.00     6301     0.00     0.00  Insert(std::__cxx11::basic_string<char, std:
0.00       0.32     0.00     6171     0.00     0.00  Remove(std::__cxx11::basic_string<char, std:
0.00       0.32     0.00     6022     0.00     0.00  InsertNode(TAVL<std::__cxx11::basic_string<c
0.00       0.32     0.00     3261     0.00     0.00  TNode::~TNode()
0.00       0.32     0.00     3124     0.00     0.01  Serialize(std::basic_ofstream<char, std::cha
0.00       0.32     0.00     3044     0.00     0.09  Deserialize(std::basic_ifstream<char, std::c
0.00       0.32     0.00     2137     0.00     0.00  LeftRotate(TAVL<std::__cxx11::basic_string<c
0.00       0.32     0.00     2064     0.00     0.00  RightRotate(TAVL<std::__cxx11::basic_string<
0.00       0.32     0.00      263     0.00     0.00  RemoveNode(TAVL<std::__cxx11::basic_string<c
0.00       0.32     0.00       82     0.00     0.00  Min(TAVL<std::__cxx11::basic_string<char, st
0.00       0.32     0.00       82     0.00     0.00  RemoveMin(TAVL<std::__cxx11::basic_string<ch
```

Как мы видим, большую часть времени (75%) программа проводит в рекурсивной функции DeserializeRecursive. Неудивительно, так как чтение и восстановление дерева - достаточно трудоемкая операция.

Также можно построить графы вызовов при помощи gprof2dot.



gcov

Gcov — свободно распространяемая утилита для исследования покрытия кода. Gcov генерирует точное количество исполнений для каждого оператора в программе и позволяет добавить аннотации к исходному коду. С помощью утилит lcov и genhtml можно получить html страницу с отчетом покрытия кода.

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	Lines:	217	238	91.2 %
Test:	report.info	Functions:	31	36	86.1 %
Date:	2022-05-15 15:20:54				

Directory	Line Coverage		Functions	
/home/yadroff/CLionProjects/DA/2lab/src	92.2 %	202 / 219	92.3 %	24 / 26
bits	88.2 %	15 / 17	87.5 %	7 / 8
ext	0.0 %	0 / 2	0.0 %	0 / 2

Generated by: LCOV version 1.14

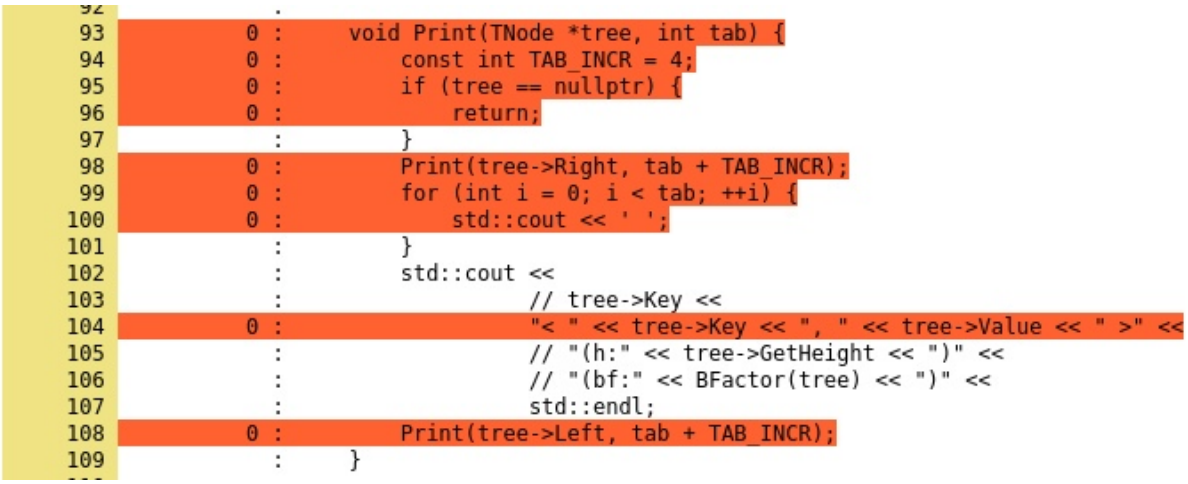
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - /home/yadroff/CLionProjects/DA/2lab/src	Lines:	202	219	92.2 %
Test:	report.info	Functions:	24	26	92.3 %
Date:	2022-05-15 15:20:54				

Filename	Line Coverage		Functions	
avl.hpp	90.8 %	157 / 173	92.0 %	23 / 25
main.cpp	97.8 %	45 / 46	100.0 %	1 / 1

Generated by: LCOV version 1.14

```
68214 :     TNode *InsertNode(TNode *tree, TNode *ins) {
68214 :         if (tree == nullptr) {
 6022 :             return ins;
      :         }
62192 :         if (ins->Key < tree->Key) {
29990 :             TNode *temp = InsertNode(tree->Left, ins);
29990 :             if (temp == nullptr)
    0 :                 return nullptr;
29990 :             tree->Left = temp;
32202 :         } else if (ins->Key > tree->Key) {
32202 :             TNode *temp = InsertNode(tree->Right, ins);
32202 :             if (temp == nullptr)
    0 :                 return nullptr;
32202 :             tree->Right = temp;
      :         } else {
    0 :             return nullptr;
      :         }
62192 :         FixHeight(tree);
62192 :         return Balance(tree);
      :     }
```



Как мы видим, в реализации avl-дерева покрыто 90.8% кода, в то время как в main.cpp покрыто 97.8% кода. Это объясняется тем, что в avl.hpp присутствует реализация печати дерева, а в main.cpp присутствует вызов этой печати. В тестах печать не использовалась, так как была заложена мной как служебная функция для проверки программы на правильность.

Выводы

Я познакомился с очень полезными инструментами:

1. Valgrind позволяет выявлять утечки памяти и профилировать код. Но для поиска утечек лучше использовать fsanitize, а для профилирования - утилиты ниже. Однако, инструмент довольно неплохо справляется со своей задачей.

2. gprof позволяет оценить производительность программы, выявляя слабые места в плане производительности. Также есть функция постройки графа вызовов.

3. gcov позволяет исследовать покрытие кода. Можно сгенерировать страницу формата html, в которой будет показано наше покрытие.

Инструменты являются полезными. В дальнейшем будущем я буду стараться использовать их как можно чаще.