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

Выполнил студент группы М8О-209Б-23 *Кривошапкин Егор*.

## Условие

Кратко описывается задача:

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

## Метод решения

Изучение 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 поиска утечек и отладки использования памяти.

```
egorx2000@DESKTOP-4CIE031:/mnt/d/documents/c++/da_labs/3/src$ g++ main.cpp -o main
egorx2000@DESKTOP-4CIE031:/mnt/d/documents/c++/da_labs/3/src$ valgrind --leak-check=full ./main < ./input.txt
==618== Memcheck, a memory error detector
==618== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==618== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==618== Command: ./main
==618==
==618==
==618== HEAP SUMMARY:
==618==     in use at exit: 0 bytes in 0 blocks
==618==   total heap usage: 55,677 allocs, 55,677 frees, 11,385,039 bytes allocated
==618==
==618== All heap blocks were freed -- no leaks are possible
==618==
==618== For lists of detected and suppressed errors, rerun with: -s
==618== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
egorx2000@DESKTOP-4CIE031:/mnt/d/documents/c++/da_labs/3/src$
```

Как видим, утечек памяти с помощью Valgrind обнаружено не было, и моя программа корректно управляет памятью. При использовании моего теста было произведено 55677 аллокаций и освобождений памяти.

## gprof

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

Применю gprof на том же тесте.

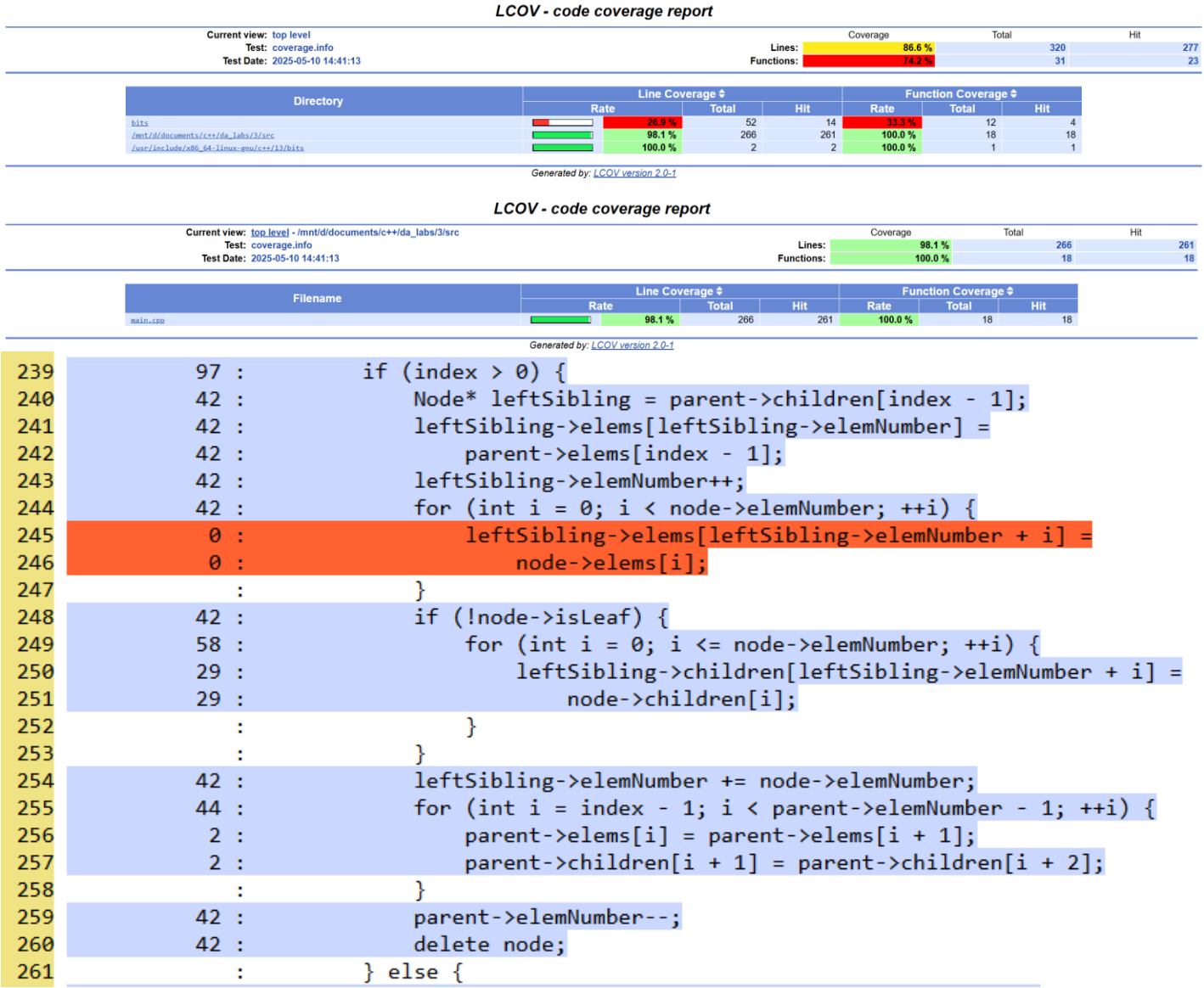
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 54.55      0.06     0.06  1847322     0.03     0.03  BTree::strLess(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  9.09      0.07     0.01   166599     0.06     0.50  BTree::insertElem(Node*, NodeElem const&)
  9.09      0.08     0.01   119209     0.08     0.13  BTree::splitNode(Node*, int)
  9.09      0.09     0.01      236    42.37    42.37  BTree::saveNode(std::ostream&, Node*)
  9.09      0.10     0.01                             _init
  4.55      0.10     0.01   649501     0.01     0.01  NodeElem::~NodeElem()
  4.55      0.11     0.01   363648     0.01     0.01  NodeElem::NodeElem()
  0.00      0.11     0.00  1847322     0.00     0.00  unsigned long const& std::min<unsigned long>(unsigned long const&, unsigned long const&)
  0.00      0.11     0.00   591390     0.00     0.00  NodeElem::operator=(NodeElem const&)
  0.00      0.11     0.00   166599     0.00     0.51  BTree::insert(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, unsigned long long)
  0.00      0.11     0.00   121216     0.00     0.04  Node::Node()
  0.00      0.11     0.00   121216     0.00     0.02  Node::~Node()
  0.00      0.11     0.00   121216     0.00     0.04  BTree::createNode()
  0.00      0.11     0.00   119254     0.00     0.00  NodeElem::NodeElem(NodeElem const&)
  0.00      0.11     0.00    89303     0.00     0.00  BTree::strEqual(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.11     0.00    28507     0.00     0.00  std::__is_constant_evaluated()
  0.00      0.11     0.00    17324     0.00     0.00  std::char_traits<char>::length(char const*)
  0.00      0.11     0.00    17324     0.00     0.00  bool std::operator==<char, std::char_traits<char>, std::allocator<char> >(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, char const*)
  0.00      0.11     0.00    11183     0.00     0.00  std::char_traits<char>::compare(char const*, char const*, unsigned long)
  0.00      0.11     0.00     7983     0.00     0.26  BTree::find(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.11     0.00     1534     0.00     0.43  BTree::removeElem(Node*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, Node*, int)
  0.00      0.11     0.00     1534     0.00     0.43  BTree::remove(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.11     0.00      248     0.00    11.28  BTree::deleteNode(Node*)
  0.00      0.11     0.00      248     0.00     0.00  BTree::BTree(int)
  0.00      0.11     0.00      248     0.00    11.28  BTree::~BTree()
  0.00      0.11     0.00      247     0.00   331.96  BTree::load(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.11     0.00      236     0.00    42.37  BTree::save(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.11     0.00      163     0.00     0.02  BTree::balanceNodes(Node*, Node*, int)
  0.00      0.11     0.00       45     0.00     0.00  BTree::getMaxLeft(Node*, int)
```

Как видим, большая часть времени работы программы (54,55%) тратится на сравнение строк, поскольку в функции используется посимвольное сравнение в цикле без оптимизаций, к тому же для каждого символа вызывается нетривиальная операция понижения регистра.

## 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. Проверим покрытия кода при выполнении моей программой всё того же теста, получив ту самую html-страницу:



**LCOV - code coverage report**

| | | | Coverage | Total | Hit |
|---|---|---|---|---|---|
| Current view: | top level | Lines: | 86.6 % | 320 | 277 |
| Test: | coverage.info | Functions: | 74.2 % | 31 | 23 |
| Test Date: | 2025-05-10 14:41:13 | | | | |

| Directory | Line Coverage Rate | Line Coverage Total | Line Coverage Hit | Function Coverage Rate | Function Coverage Total | Function Coverage Hit |
|---|---|---|---|---|---|---|
| bits | 26.9 % | 52 | 14 | 33.3 % | 12 | 4 |
| /mnt/d/documents/c++/da_labs/3/src | 98.1 % | 266 | 261 | 100.0 % | 18 | 18 |
| /usr/include/x86_64-linux-gnu/c++/13/bits | 100.0 % | 2 | 2 | 100.0 % | 1 | 1 |

Generated by: LCOV version 2.0-1

**LCOV - code coverage report**

| | | | Coverage | Total | Hit |
|---|---|---|---|---|---|
| Current view: | top level - /mnt/d/documents/c++/da_labs/3/src | Lines: | 98.1 % | 266 | 261 |
| Test: | coverage.info | Functions: | 100.0 % | 18 | 18 |
| Test Date: | 2025-05-10 14:41:13 | | | | |

| Filename | Line Coverage Rate | Line Coverage Total | Line Coverage Hit | Function Coverage Rate | Function Coverage Total | Function Coverage Hit |
|---|---|---|---|---|---|---|
| main.cpp | 98.1 % | 266 | 261 | 100.0 % | 18 | 18 |

Generated by: LCOV version 2.0-1

```
239          97 :          if (index > 0) {
240          42 :              Node* leftSibling = parent->children[index - 1];
241          42 :              leftSibling->elems[leftSibling->elemNumber] =
242          42 :                  parent->elems[index - 1];
243          42 :              leftSibling->elemNumber++;
244          42 :              for (int i = 0; i < node->elemNumber; ++i) {
245           0 :                  leftSibling->elems[leftSibling->elemNumber + i] =
246           0 :                      node->elems[i];
247             :              }
248          42 :              if (!node->isLeaf) {
249          58 :                  for (int i = 0; i <= node->elemNumber; ++i) {
250          29 :                      leftSibling->children[leftSibling->elemNumber + i] =
251          29 :                          node->children[i];
252             :                  }
253             :              }
254          42 :              leftSibling->elemNumber += node->elemNumber;
255          44 :              for (int i = index - 1; i < parent->elemNumber - 1; ++i) {
256           2 :                  parent->elems[i] = parent->elems[i + 1];
257           2 :                  parent->children[i + 1] = parent->children[i + 2];
258             :              }
259          42 :              parent->elemNumber--;
260          42 :              delete node;
261             :          } else {
```

Как видим, в непосредственно самой реализации B-Дерева покрыто 98.1% кода. Это объясняется тем, что в тесте исследуются не все случаи работы с деревом. Например, при удалении ни разу не произошло слияния узла с левым соседом.

## Выводы

Я познакомился с инструментами, позволяющими находить недостатки даже в, казалось бы, полностью рабочих программах:

1. valgrind позволяет выявлять утечки памяти и профилировать код.
2. gprof позволяет оценить производительность программы, посмотреть, какие структуры и функции замедляют программу, выявить слабые места в производительности.
3. gcov позволяет исследовать покрытие кода, определив его участки, которые можно удалить или сократить. Можно сгенерировать страницу формата html, в которой будет показано наше покрытие.

Инструменты являются крайне полезными. В дальнейшем будущем я буду стараться использовать их как можно чаще, что, несомненно, повысит качество написанных мною программ.