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

Выполнил студент группы М8О-208Б-21 *Армишев Кирилл.*

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 отладки использования памяти.

```
==4777==    by 0x109C0F: AVLTree::addNode(char*, unsigned long) (main.cpp:38)
==4777==    by 0x1096E7: main (main.cpp:283)
==4777==
==4777== LEAK SUMMARY:
==4777==    definitely lost: 1,321,204 bytes in 48,618 blocks
==4777==    indirectly lost: 0 bytes in 0 blocks
==4777==      possibly lost: 40 bytes in 1 blocks
==4777==    still reachable: 122,880 bytes in 6 blocks
==4777==         suppressed: 0 bytes in 0 blocks
==4777==
==4777== For lists of detected and suppressed errors, rerun with: -s
==4777== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Как видим, Valgrind обнаружил утечки памяти. По его выводу можно понять, что утечка памяти происходит в функции AVLTree::addNode(char*, unsigned long) на строке 38 файла main.cpp. В сообщении от Valgrind указана возможная потеря в 40 байт в блоке памяти, выделенном операцией malloc, что может быть связано с функцией AVLTree::addNode().

Я обнаружил утечку: в 38 строчке с помощью malloc я выделяю память под хранение значения ключа и в дальнейшем в функции _removeNode я делаю удаление delete p, но здесь я предпологал, что delete очистит и память, выделенную под ключ malloc-ом, но это не так, вот объяснение:

Операция delete освобождает как сам объект, так и память, выделенную под его члены-указатели (если таковые есть). Однако, в данном случае p->key - это указатель на строку, которая была выделена отдельно с помощью функции malloc(), и delete не освобождает память, выделенную под эту строку. Поэтому перед удалением узла p с помощью delete, мне нужно явно освободить память, выделенную под его ключ p->key, с помощью free().

Я отладил программу, добавив строчку free(p->key) в функцию _removeNode и _clearAVLTree и еще раз прогнал через Valgrind:

```
==4871==
==4871== HEAP SUMMARY:
==4871==     in use at exit: 122,880 bytes in 6 blocks
==4871==   total heap usage: 97,247 allocs, 97,241 frees, 3,470,252 bytes allocated
==4871==
==4871== LEAK SUMMARY:
==4871==    definitely lost: 0 bytes in 0 blocks
==4871==    indirectly lost: 0 bytes in 0 blocks
==4871==      possibly lost: 0 bytes in 0 blocks
==4871==    still reachable: 122,880 bytes in 6 blocks
==4871==         suppressed: 0 bytes in 0 blocks
==4871== Rerun with --leak-check=full to see details of leaked memory
==4871==
==4871== For lists of detected and suppressed errors, rerun with: -s
==4871== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Теперь нет утечек памяти.

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.

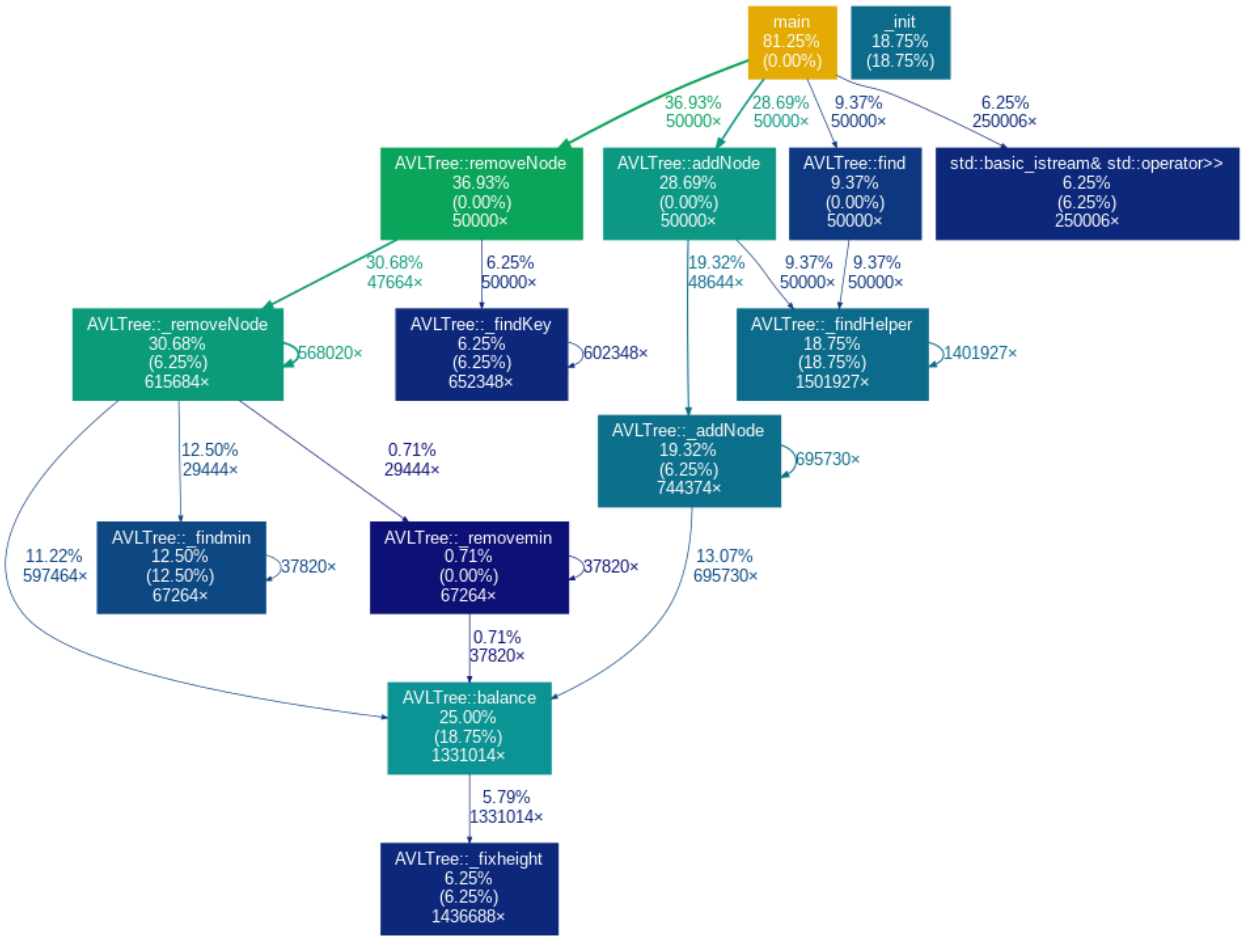
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 18.75      0.03     0.03  1331014     0.02     0.03  AVLTree::balance(Node*)
 18.75      0.06     0.03   100000     0.30     0.30  AVLTree::_findHelper(Node*, char*, unsigne
```

18.75	0.09	0.03				_init
12.50	0.11	0.02	29444	0.68	0.68	AVLTree::_findmin(Node*)
6.25	0.12	0.01	1436688	0.01	0.01	AVLTree::_fixheight(Node*)
6.25	0.13	0.01	250006	0.04	0.04	std::basic_istream<char, std::char_traits<
6.25	0.14	0.01	50000	0.20	0.20	AVLTree::_findKey(Node*, char*)
6.25	0.15	0.01	48644	0.21	0.64	AVLTree::_addNode(Node*, char*, unsigned l
6.25	0.16	0.01	47664	0.21	1.03	AVLTree::_removeNode(Node*, char*)
0.00	0.16	0.00	8232356	0.00	0.00	AVLTree::_height(Node*)
0.00	0.16	0.00	2679490	0.00	0.00	AVLTree::_editbalance(Node*)
0.00	0.16	0.00	50071	0.00	0.00	Node::Node(char*, unsigned long)
0.00	0.16	0.00	50000	0.00	1.18	AVLTree::removeNode(char*)
0.00	0.16	0.00	50000	0.00	0.30	AVLTree::find(char*, unsigned long*)
0.00	0.16	0.00	50000	0.00	0.92	AVLTree::addNode(char*, unsigned long)
0.00	0.16	0.00	29444	0.00	0.04	AVLTree::_removemin(Node*)
0.00	0.16	0.00	26998	0.00	0.01	AVLTree::_rotateleft(Node*)
0.00	0.16	0.00	25839	0.00	0.01	AVLTree::_rotateright(Node*)
0.00	0.16	0.00	2	0.00	0.00	AVLTree::_clearAVLTree(Node*)
0.00	0.16	0.00	2	0.00	0.00	AVLTree::clearAVL()
0.00	0.16	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(
0.00	0.16	0.00	1	0.00	0.00	AVLTree::Deserialize(std::basic_ifstream<c
0.00	0.16	0.00	1	0.00	0.00	AVLTree::_SerializeRecursive(Node*, std::b
0.00	0.16	0.00	1	0.00	0.00	AVLTree::_DeserializeRecursive(std::basic_
0.00	0.16	0.00	1	0.00	0.00	AVLTree::Serialize(std::basic_ofstream<cha
0.00	0.16	0.00	1	0.00	0.00	AVLTree::AVLTree()

Как мы видим, большую часть времени программа проводит в функция balance и _findHelper. Неудивительно, так как в моем тесте большинство операций - это вставка и удаление, а поиск и балансирова для этих функций являются весьма затратными операциями.

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.

```
      27            : class AVLTree {
      28            : public:
      29          1 :     AVLTree() {
      30          1 :         root = nullptr;
      31          1 :     }
      32            :
      33            :     // добавление узла
      34      50000 :     bool addNode(char* key, uint64_t value) {
      35      50000 :         if(_findHelper(root,key,&value)){
      36       1381 :             return false;
      37            :         }
      38      48619 :         char* pastekey=(char*)malloc((strlen(key)+1)*sizeof(char));
      39      48619 :         strcpy(pastekey,key);
      40      48619 :         root = _addNode(root, pastekey, value);
      41      48619 :         return true;
      42            :     }
      43            :
      44            :     // удаление узла
      45      50000 :     bool removeNode(char* key) {
      46      50000 :         if(_findKey(root,key)){
      47      48619 :             root = _removeNode(root, key);
      48      48619 :             return true;
      49            :         }
      50       1381 :         return false;
      51            :     }
      52            :
      53            :     // поиск узла
      54      50000 :     bool find(char* key, uint64_t* value) {
      55      50000 :         return _findHelper(root, key, value);
      56            :     }
      57            :
      58            :
      59        301 :     void clearAVL() {
      60        301 :         _clearAVLTree(root);
      61        301 :         root=nullptr;
      62        301 :     }
      63            :
      64        181 :     void Deserialize(std::ifstream &is) {
      65        181 :         root = _DeserializeRecursive(is);
      66        181 :     }
      67            :
      68        300 :     void Serialize(std::ofstream &ofs) {
      69        300 :         _SerializeRecursive(root, ofs);
      70        300 :     }
      71            :
```

Как видно, в моей программе покрыто 98.6% кода. Задействованы абсолютно все функции.

Выводы

Я познакомился с очень полезными инструментами:

1. Valgrind позволяет выявлять утечки памяти и профилировать код. Инструмент оказалася довольно не сложным и удобным.

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, где все очень наглядно видно.

Инструменты оказались очень полезными. Лабораторная работа помогла исправить утечки в памяти.