Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: И. С. Глушатов
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №3

Задача: 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 версии.
- Общих выводов о выполнении лабораторной работы, полученном опыте.

1 Описание

Инструменты, которыми я пользовался: valgrind, plot, perf
Схема директорий:

- main.cpp
- string.hpp
- avltree.hpp
- tests_creator.py
- makefile
- bash_script.sh
- gnuplot_script.txt
- perf.data
- plots (папка с графиками)
- tests (папка с тестами)
- scrins (папка с скриншотами)
- report (папка с отчетом)

Во-первых, для того, чтобы анализировать свою программу, время ее исполения и утечки памяти, нужно было написать генератор тестов. Для данной задачи я выбрал язык Python.

```
import random
import copy
from random import choice
from string import ascii_uppercase

randstring = lambda x=1: ''.join(choice(ascii_uppercase) for i in range(x))

if __name__ == "__main__":

    count_of_tests = 5
    max_count_of_commands = 50
    count_of_commands = 1

    actions = ["+", "!", "?", "!"] # + - ? !
    for test_number in range(1, count_of_tests+1):
        tree_elems = {}
        saved = 0
        saved_file = {}
        test_file = open("tests/" + str(test_number) + ".t", 'w')
        file_with_answers = open("tests/" + str(test_number) + ".txt", "w")

        for _ in range( count_of_commands ):
            action = random.choice( actions )
            if action == "+":
                key = randstring()
                value = random.randint( 1, 2**64-1 )
```

```
                test_file.write("+ " + key + " " + str(value) + "\n")

                key = key.lower()
                answer = "Exist"
                if key not in tree_elems:
                    answer = "OK"
                    tree_elems[key] = value
                file_with_answers.write(answer + "\n")

            elif action == "-":
                key = randstring()
                test_file.write("- " + key + "\n")

                key = key.lower()
                answer = "NoSuchWord"
                if key in tree_elems:
                    del tree_elems[key]
                    answer = "OK"
                file_with_answers.write(answer + "\n")

            elif action == "?":
                search_exist_element = random.choice( [ True, False ] )
                key = random.choice([key for key in tree_elems.keys()]) if
                    search_exist_element and len(tree_elems.keys()) > 0 else randstring
                    ()
                test_file.write(key + "\n")

                key = key.lower()
                if key in tree_elems:
                    answer = "OK: " + str(tree_elems[key])
                else:
                    answer = "NoSuchWord"
                file_with_answers.write(answer + "\n")

            elif action == "!":
                act_file = random.choice(["Load test", "Save test"])
                if act_file == "Save test":
                    test_file.write(action + " " + act_file + "\n")
                    saved_file = tree_elems.copy()
                    file_with_answers.write("OK" + "\n")
                    saved = 1

                elif saved == 1 and act_file == "Load test":
                    test_file.write(action + " " + act_file + "\n")
                    tree_elems = {}
                    tree_elems = saved_file.copy()
                    file_with_answers.write("OK" + "\n")

        count_of_commands+=(max_count_of_commands-count_of_commands)/count_of_tests
```

Генератор тестов хорошо работает и не учитывает только случаи, когда нельзя будет открыть файл по каким-то причинам. Поэтому он не сможет проверить ошибки с ERROR. Так как я хочу испытывать произвольное число тестов и причем их большое количество, я написал bash файл, где будет запускаться моя прорамма, перенаправляя вывод в файл, а потом с помощью утилиты diff сравнивать файлы от питона и от моей программы.

```
#!/bin/bash
max=6

for ((i = 1; i < ${max}; i++))
```

```
do
  echo "test: $i ";
  ./main.out < tests/${i}.t > tests/${i}ans.txt;
  echo "ans: $i ";
  diff tests/${i}ans.txt tests/${i}.txt
done
```

Переменная max равняется количеству тестов, которые нужно прогнать.

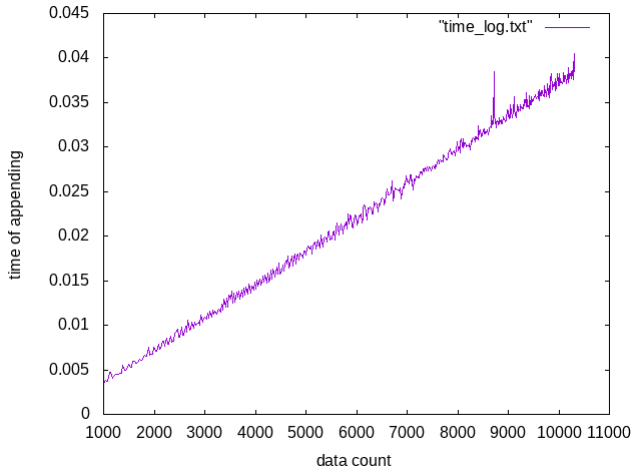
В самой программе я установил таймер из библиотеки <chrono>, а так же счетчик команд. Результат времени работы программы и счетчик команд выводятся через пробел в файл time_log.txt, откуда gnuplot считывает данные и строит график зависимости время исполнения программы от количества входных данных. Для gnuplot так же написан скрипт. Мы устанавливаем название осей, расширение изображения, и директорию, куда будет выводиться картинка.

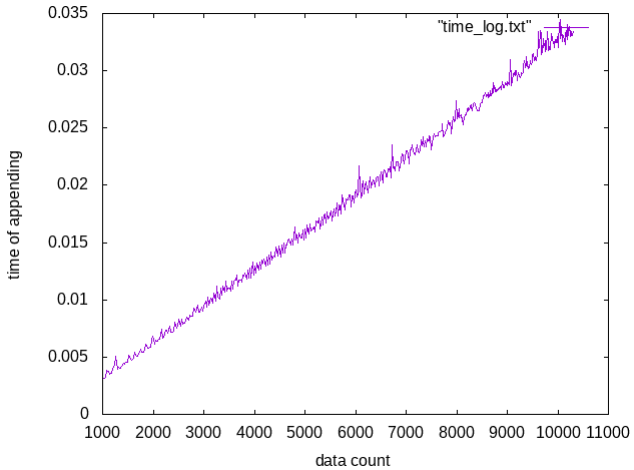
```
set xlabel "data count"
set ylabel "time of appending"
set terminal png
set output "plots/plot.png"
plot "time_log.txt" with lines
pause -1
```

Makefile собирает все скрипты вместе и запускает их.

```
#!/bin/bash
CC = g++
FLAGS = -pedantic -Wall -std=c++11 -Werror -Wno-sign-compare -O2 -lm

all: main.out

main.out: main.cpp avltree.hpp string.hpp
  ${CC} ${FLAGS} main.cpp -o main.out

run:
  rm -rf time_log.txt
  python tests_creator.py
  ./bash_script.sh
  gnuplot gnuplot_script.txt

testss:
  python tests_creator.py
```

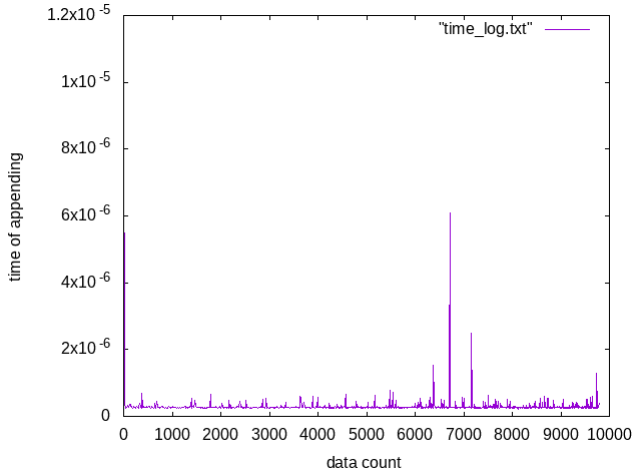
Первые результаты бенчмарка я опробовал на тестах с только вставкой и на тестах с вставкой, удалением и поиском.





Для меня было удивлением увидеть такие результаты, ведь я ожидал график с ас-

симптотикой $nlog_2n$, а увидел прямопропорциональную зависимость времени исполения прорамммы от количеств элементов. Поэтому я решил провести тест, где строился график зависимости времени вставки в дерево от количества уже вставленных элементов в него, т.е. время вставки последнего элемента в дерево.



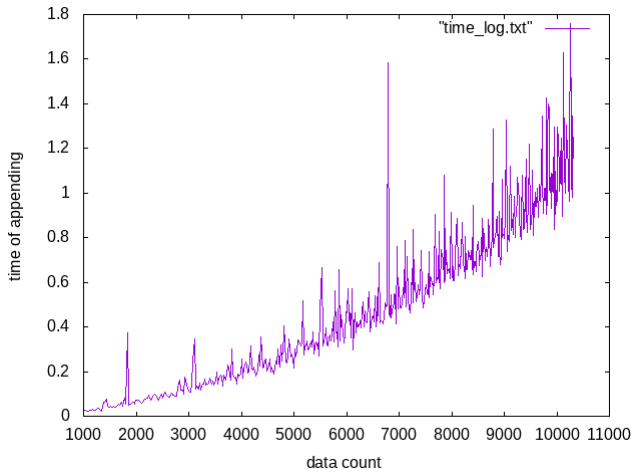
И по данному графику я понял, почему два до этого давали такой результат. На таком маленьком числе элементов (порядка 10000) время вставки было практически константным. Скорее всего, если бы я мог задействовать одно ядро компьютера полностью под свою программу и взять тесты побольше, то я бы увидел плавный рост, однако не обладая такими возможностями, я удовлетворился этими результатами.

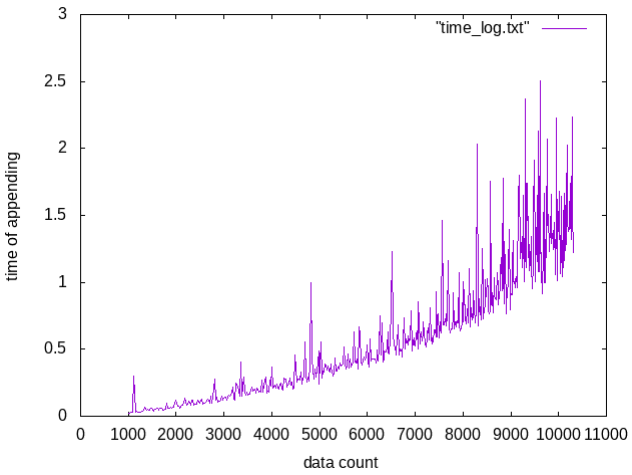
Несмотря на такие хорошие показатели, моя программа не сразу прошла на чекере и причина тому теперь как никогда ясна - команды Save и Load сильно тормозили процесс. Поэтому пришло время провести тест, добавив эти две команды. Сначала я использовал версию программы, где загрузка дерева проходила с помощью вставки в обычное бинарное дерево. Потом я сделал еще одну версию, где сохранение не сильно отличалось от старой, а загрузка производилось с помощью флагов, которые означали есть ли ребенок слева и справа.

```
1  void TAVLTree::Save(const TAVLTree *head, FILE *s) {
2    if (head) {
3      fprintf(s, "%s %llu ", head->key.str, head->value);
4      Save(head->left, s);
5      Save(head->right, s);
6    }
7  }
8
9  TAVLTree* TAVLTree::BinInsert(TAVLTree *head, const NMystd::TString& key, const
       unsigned long long value) {
10   if (head) {
11     if (key > head->key) {
12       head->right = BinInsert(head->right, key, value);
13     } else {
14       head->left = BinInsert(head->left, key, value);
15     }
16     return head;
17   }
18   return new TAVLTree(key, value);
19 }
20
21 TAVLTree* TAVLTree::Load(std::ifstream &in) {
22    TAVLTree *res = 0;
23    char str[256];
24    unsigned long long val;
25    while (fscanf(s, "%s %llu ", str, &val)>0) {
```

```
    res = res->BinInsert(res, NMystd::TString(str), val);
   }
   return res;
}
```

```
void TAVLTree::Save(const TAVLTree *head, FILE *s) {
  if (!head) {
    return;
  }

  fprintf(s, "%s %llu %u ", head->key.str, head->value, head->h);
  if (head->left) {
    fprintf(s, "1 ");
    Save(head->left, s);
  } else {
    fprintf(s, "0 ");
  }
  if (head->right) {
    fprintf(s, "1 ");
    Save(head->right, s);
  } else {
    fprintf(s, "0 ");
  }
}

TAVLTree* TAVLTree::Load(std::ifstream &in) {
  TAVLTree *res;
  int flag;
  char str[256];
  unsigned long long val;
  unsigned int h;


  if (in >> str >> val >> h) {
    res = new TAVLTree(NMystd::TString(str), val);
    res->h = h;
    in >> flag;
    if (flag==1) {
      res->left = Load(in);
    } else {
      res->left = NULL;
    }
    in >> flag;
    if (flag==1) {
      res->right = Load(in);
    } else {
      res->right = NULL;
    }
  } else {
    res = NULL;
  }
  return res;
}
```

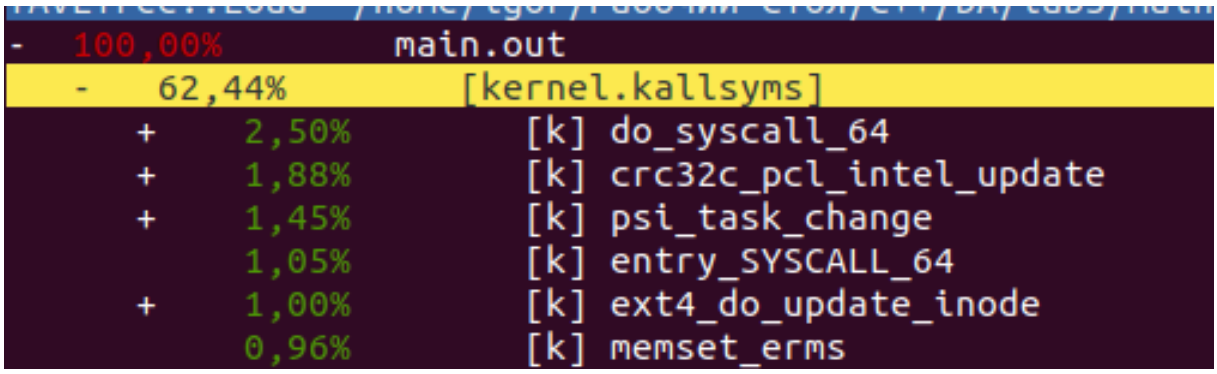




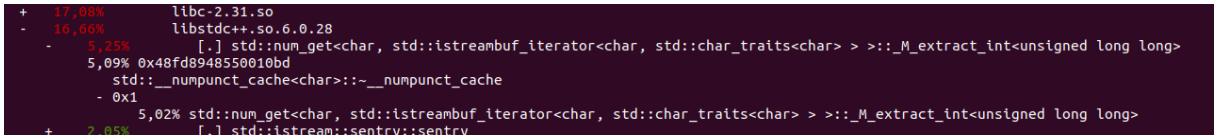
Несмотря на то, что изначальный вариант программы имеет лучшие результаты, видно, что ассимптотика у них одинаковая, а из-за шума посторонних процессов компьютера сложно наверняка определить, какой из вариантов лучше. В первом случае сложность загрузки $nlog_2n$, так как мы вытаскиваем из файла по одному элементу ($O(n)$), и вставляем в дерево за log_2n. Во втором случае мы встречаемся с элементом только один раз, вставляем и больше к нему не возвращаемся, из чего следует сложность ($O(n)$). Однако как мы убедились на прошлых графиках, вставка на таком количестве элементов производится почти за константное время, поэтому графики выходят с практически одинаковой ассимптотикой. Почему не линейный вид графика? Потому что считается время выполнения всей программы, а значит чем больше команд подается, тем больше сохранений и загрузок будут появляться в разных местах теста, когда в дереве то 10 элементов, то 10000. Точный расчет сложности от таких случайных тестов должен включать теорию вероятностей, поэтому на данный момент остается для меня нерешаемой задачей.

Утилита Perf предоставляет инструмент анализа производительности в Linux посредством счетчиков. К сожалению, документация этой утилиты крайне сложная и размытая, однако я попытался чуть-чуть разобраться. Задача, которую я хочу достичь с помощью Perf - узнать распределение нагрузки частей программы на процессор. Что работает медленнее (а значит дольше), а что работает хорошо. Первая сложность появилась на этапе запуска этой утилиты. Выдавалась ошибка об отказе доступа, которая решалась запуском с sudo. Разобравшись со множеством непонятного синтаксиса команд, я смог запустить в итоге трассировку именной моей пользовательской программы, а не всех процессов компьютера.

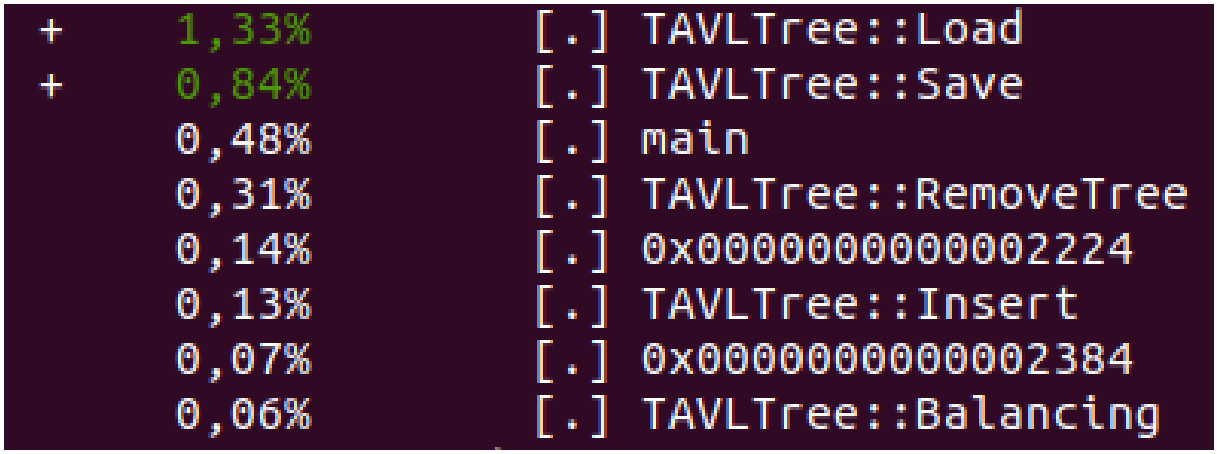
```
Samples: 1K of event 'cycles', Event count (approx.): 643017472

-  100,00%        main.out
   +   62,44%           [kernel.kallsyms]
   +   17,08%           libc-2.31.so
   +   16,66%           libstdc++.so.6.0.28
   +    3,37%           main.out
   +    0,22%           ld-2.31.so
   +    0,22%           [vdso]
+     0,00%        perf
```



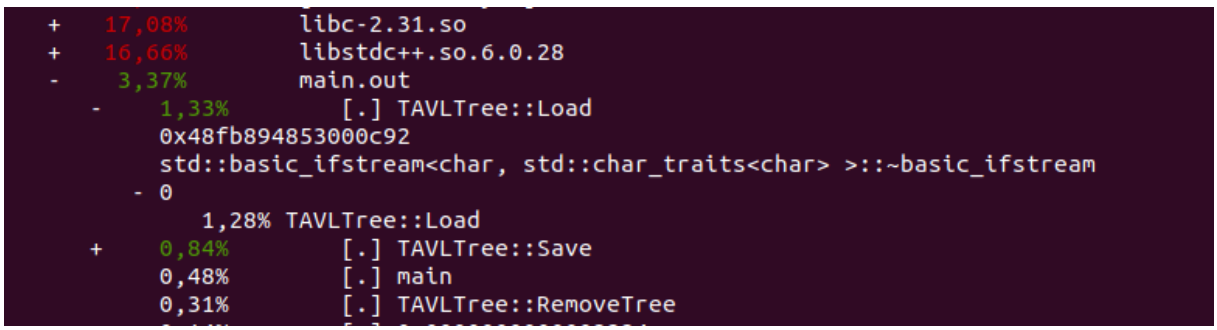
Из первых двух скриншотов мы видим, что большая часть нагрузок пала на системные вызовы ядра, что не удивительно. В основном все они внутри распределены равномерно и каждому уходит не более 2%, кроме do_syscall_64.

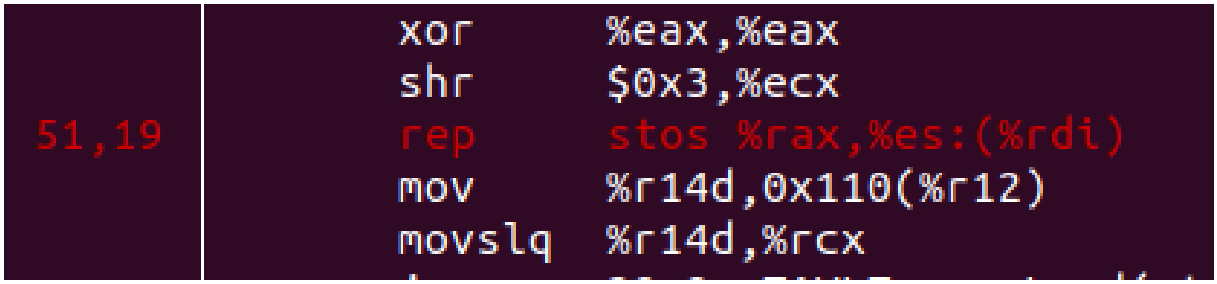


Из библиотек std больше всего уходит нагрузка на вызов функции num_get, связанная с вводом и выводом, записью и чтением чисел, так как изначально они представляются в виде символов. То есть это своего рода парсер, а так как ввода и вывода, записи и чтения при командах загрузки и сохранения дерева много, то такой расход ресурсов вполне объясняется.



Как и предполагалось, функция Load использует больше всего ресурсов, нежели функции вставки, удаления, перебалансировки и даже сохранения в файл.





И если мы пойдем глубже, то увидим, что у нас слишком часто используется ассемблерная команда REP повторения строковой операции, а именно REP STOS - заполнение блока по адресу содержимым. Странным оказалось то, что я не увидел нагрузку на сравнение строк, хотя казалось бы, что это занимает достаточное количества времени, и по-хорошему было бы их хэшировать, чтобы добиться оптимизации кода.

Привести интересные проверки с valgrind'ом я не смогу, так как все утечки памяти были устранены раннее. Класс NMystd::string, написанный для второй лабораторной работы по условию задачи имеет ограничение на 256 символов, поэтому я не делал динамическое выделение памяти для этого типа и никаких утечек там быть не могло, а AVL-дерево достаточно простое и утечки памяти, к примеру, при удалении узла было легко отследить и устранить.

2 Выводы

В ходе второй лабораторной работы я изучил инструменты gnuplot и perf. Gnuplot оказался не таким уж и сложным, а скорее даже приятным для построения графиков и анализа сложностей алгоритмов. Надеюсь, я и дальше не поленюсь и буду пользоваться данным инструментом. Perf оказался сложной утилитой. Для того, чтобы сделать тривиальные вещи пришлось потратить какое-то время, однако эта сложность определяет мощь всего инструмента. Утилита показывает все узкие горлышки программы и анализирует вплоть до кода на ассемблере. Благо из-за простоты поиска в интернете, можно изучить, за что именно отвечает кусок кода и примерно понять, как оптимизировать код, если на то будут причины. Valgrind старый инструмент, которым я пользуюсь, а в данной лабораторной работе даже не оказалось ошибок утечек памяти, однако при написании второй лабораторной работы эта утилита сильно экономит время и помогает предотвратить ошибки, которые могут выявиться только спустя какое-то время. В моем опыте были ситуации, что программы, написанные на ОС Linux Ubuntu не происходило никаких ошибок, однако при портировании кода на Windows 10 они случались. Оказалось, что Ubuntu снисходительнее относится к использованию неинициализированной памяти и поэтому на ней сложнее было находить утечки, однако Valgrind позволяет мониторить данные ситуации и вовремя замечать ошибки. Так же в данной лабораторной работе пришлось вспомнить bash, который тоже бывает полезен при разработке сложных программ и тестов для них. Результаты эффективности программы также оказались неожиданностью. Я совсем не ожидал увидеть такую скорость вставки в AVL-дерево, однако это можно было предположить, так как на чекере моя программа на самом длительном 13 тесте прошла за 40 секунд, в то время как максимально на выполнение дается 130 секунд.

Список литературы

[1] *Поисковик - Google.*
URL: `https://www.google.com/`

[2] *Сайт с подробной документацией библиотек C++*
URL: `https://en.cppreference.com/`

[3] *Сайт с постами про программирование (Perf)*
URL: `https://habr.com/ru/company/first/blog/442738/`

[4] *Сайт с постами про программирование (Gnuplot)*
URL: `https://habr.com/ru/company/ruvds/blog/517450/`

[5] *Еще немного про Perf*
URL: `https://selectel.ru/blog/perf-i-flamegraphs/`

[6] *Намного больше про Perf*
URL: `https://www.protocols.ru/WP/perf/`

[7] *Сайт с постами про программирование (Bash)*
URL: `https://habr.com/ru/company/ruvds/blog/325928/`