Московский Авиационный Институт (Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу

**«Дискретный анализ»**

Профилирование

Студент: Бойцов Иван Алексеевич

Группа: М8О–212Б–22

Вариант: 5. 0

Преподаватель: Н.Д.Глушин

Оценка:

Дата: Подпись:

Москва, 2024.

Условие

**Вариант: 5. 0**

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.

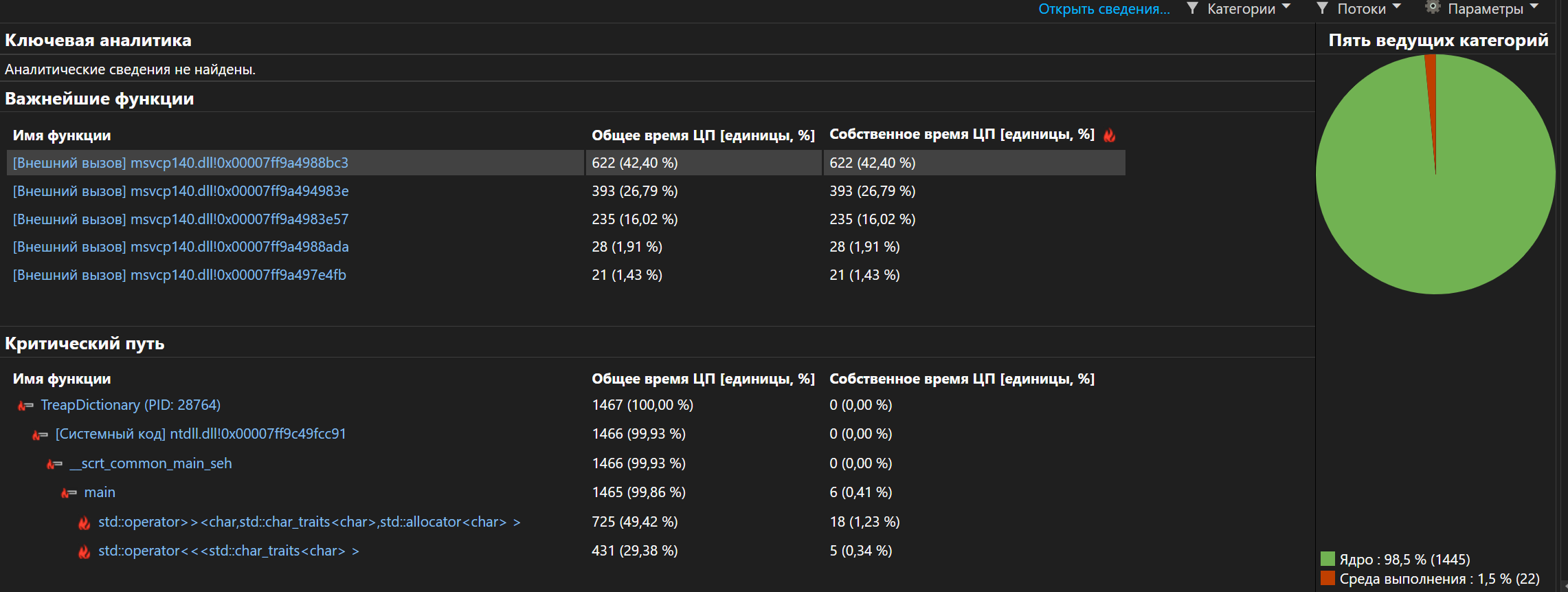
Минимальный набор используемых средств должен содержать утилиту ***gprof*** и библиотеку ***dmalloc***, однако их можно заменять на любые другие аналогичные или более развитые утилиты (например, Valgrind или Shark) или добавлять к ним новые (например, gcov).

Дневник выполнения работы

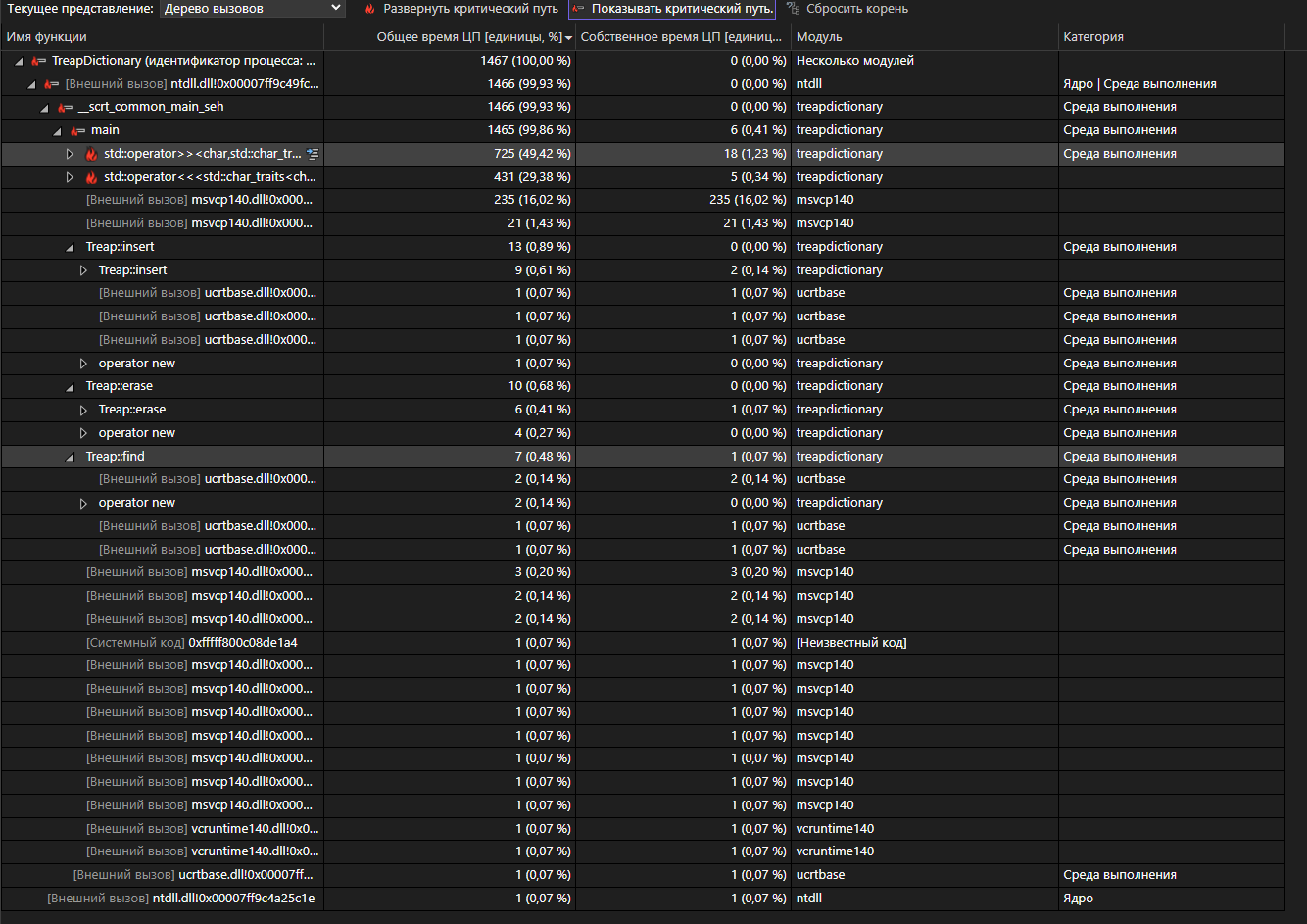
Для начала выберем инструменты для профилирования, разделив их на категории:

1. Для времени исполнения работы:  
   1. **Visual Studio Profiler** - это инструмент в Visual Studio, который помогает диагностировать использование памяти и процессора, а также другие проблемы на уровне приложения.
2. Для памяти:
   1. **Visual Leak Detector -** это сторонняя библиотека, которая является надстройкой над Debug CRT. Она позволяет искать утечки памяти в программе.

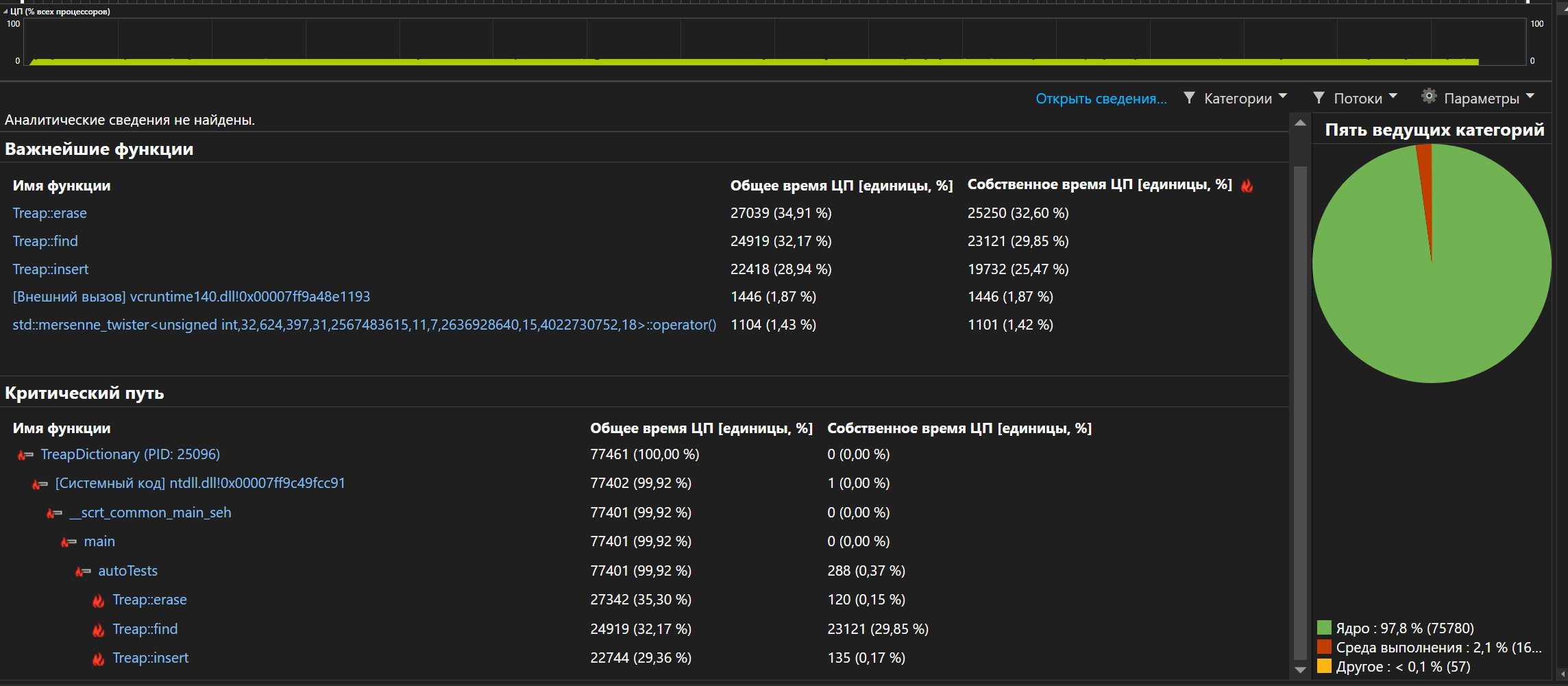
Начнём с **visual studio profiler**, а именно с проверки производительности. Для этого мы переводим проект в Release версию, нажимаем сочетание **Alt+F2**, для перехода в **Perfomance Profiler** в **Visual Studio** и выбираем пункт **CPU** **usage**, для того чтобы отследить сколько времени требуется каждому методу декартового дерева. Нажимаем на старт, тестируем программу минуту на различных тестах, получаем вот такой результат:



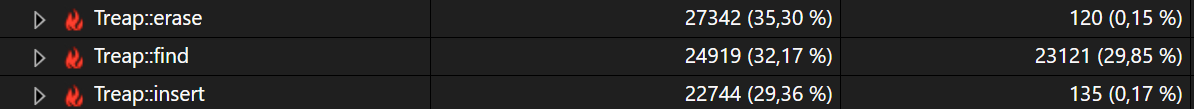
Для более детального анализа переходим по имени функции **TreapDictionary**, получаем следующий результат:



На данном скрине можно увидеть, что значительная часть времени выполнения программы приходится на внешний вызов библиотек. Это происходит потому, что программа сильно зависит от операций ввода-вывода. Однако мы можем также заметить функции **Treap::insert**, **Treap::find**, **Treap::erase**, которые соответствую нашей задаче. На данный момент на них приходится небольшое количество времени, поэтому изменим подход написав авто тесты. Сделаем их 10 000 000 для того, чтобы программа поработала подольше и повторим прошлые действия.

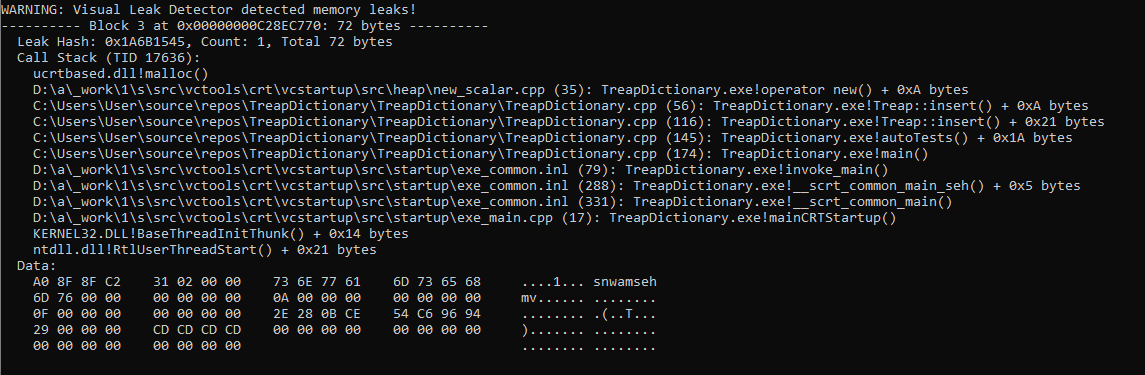


Вот теперь мы наглядно видим функции нашего дерева и так как их количество одинаковое, то мы можем оценить их производительность. Вот и они:



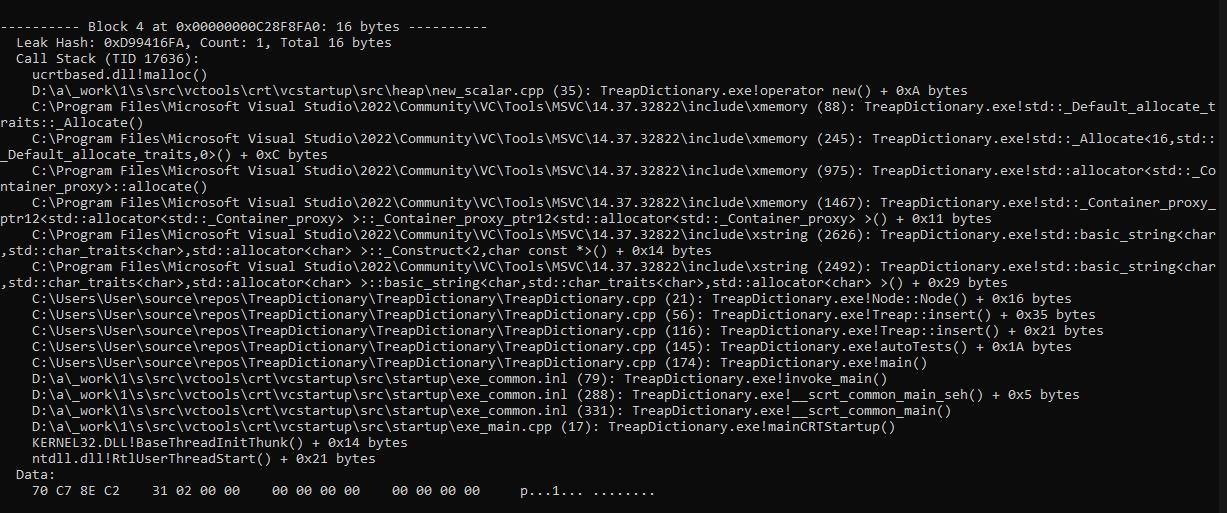
Из данного скрина мы видим, что на функцию удаления уходит большая часть времени, что указывает на то, что функцию можно следует оптимизировать. Также заметим, что функция find имеет большое **собственное время ЦП** – 29%, что может означать, что алгоритм необходимо пересмотреть и оптимизировать.

Теперь проверим нашу программу на утечки памяти, используя стороннюю библиотеку **Visual Leak Detector**. Предварительно установим её с [сайта](https://marketplace.visualstudio.com/items?itemName=ArkadyShapkin.VisualLeakDetectorforVisualC) и подключим библиотеку к проекту. Однако теперь мы не будем использовать так много тестов, а обойдёмся одним, после чего получим в консоль логи об утечке памяти:



В первом логе видим, что в функции insert есть утечка в 72 байта на 56 строке, скорее всего она связана с вызовом **new** для выделения памяти, которая не была освобождена – создание нового узла.

Посмотрим второй лог:



Вторая утечка в 16 байт находится в строке 21 в конструкторе Node, в котором также используется **new** для создания узла дерева, память для которого не была освобождена.

Выводы о найденных недочётах

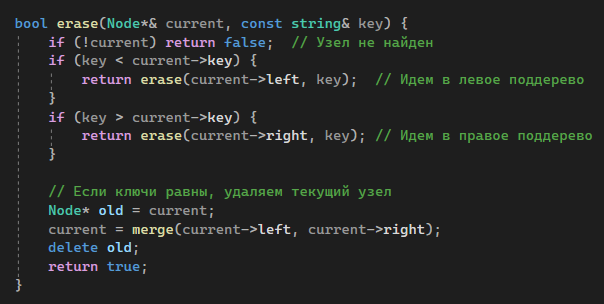
Подведем выводы о найденных проблемах:

* + Необходимо оптимизировать функцию erase и find;
  + Устранить утечки памяти на 21 и 56 строках.

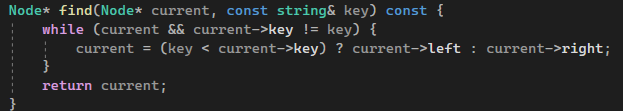
Сравнение работы исправленной программы

Для устранения потерь производительности оптимизируем функции erase и find:

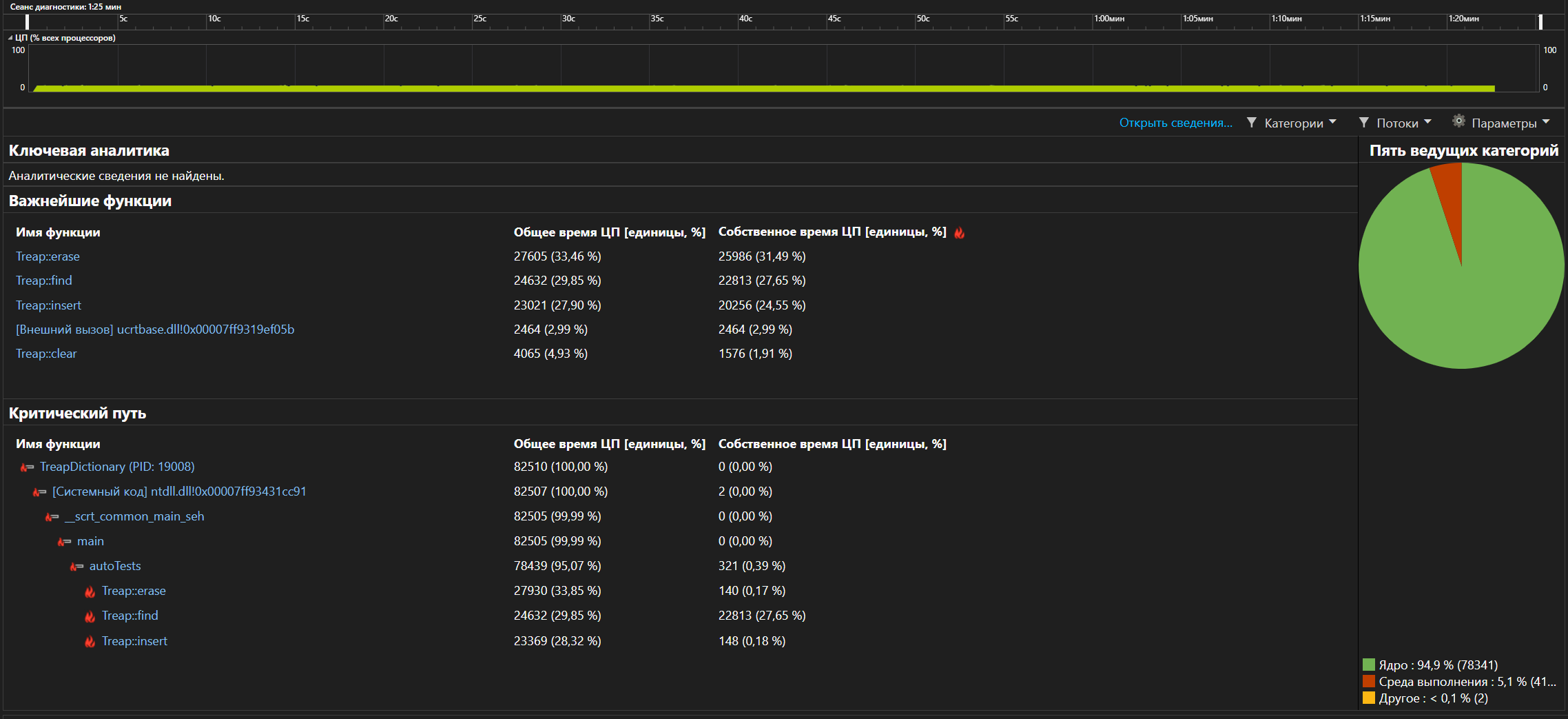
1. В функции **erase** можно убрать лишние рекурсивные вызовы и условия. Вместо использования тернарного оператора для выбора левого или правого поддерева, мы можем сразу определить нужное поддерево и вызвать **erase** только на нем. Кроме того, можно упростить логику **merge**, возвращая указатель напрямую, что уменьшит глубину кода.



1. В функции **find** можно добавить прямую проверку на неравенство, чтобы избежать лишних сравнений. Это уменьшит количество ветвлений.



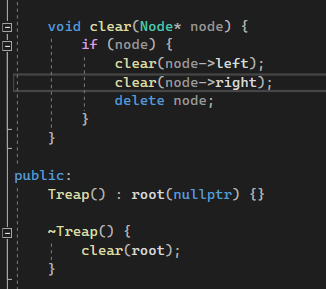
После этого проведем ещё одну диагностику:



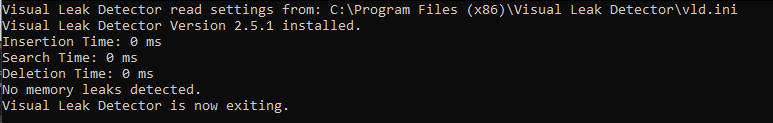
На скриншоте мы видим, что **общее время ЦП** и **собственное время ЦП** изменилось в оптимизированных функциях на 2% (erase) и на 4% (find), помимо этого **собственное время ЦП find** также упало на 2%.



Для того, чтобы устранить утечки памяти необходимо добавить функцию **clear**, которая рекурсивно удаляет все узлы дерева, освобождая выделенную память.



Проверим ещё раз программу на утечки и увидим, что все проблемы устранены:



Итак, мы немного оптимизировали функции удаление и поиска, а также устранили утечки памяти в нашей работе.

Вывод

Данная лабораторная работа была очень интересным опытом для меня, потому что я никогда не занимался профилированием. Она помогла мне лучше понять некоторые аспекты, написанного мной кода, а также наглядно показать, что изменения, которые я вношу в код и правда его оптимизируют и устраняют неполадки.