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:

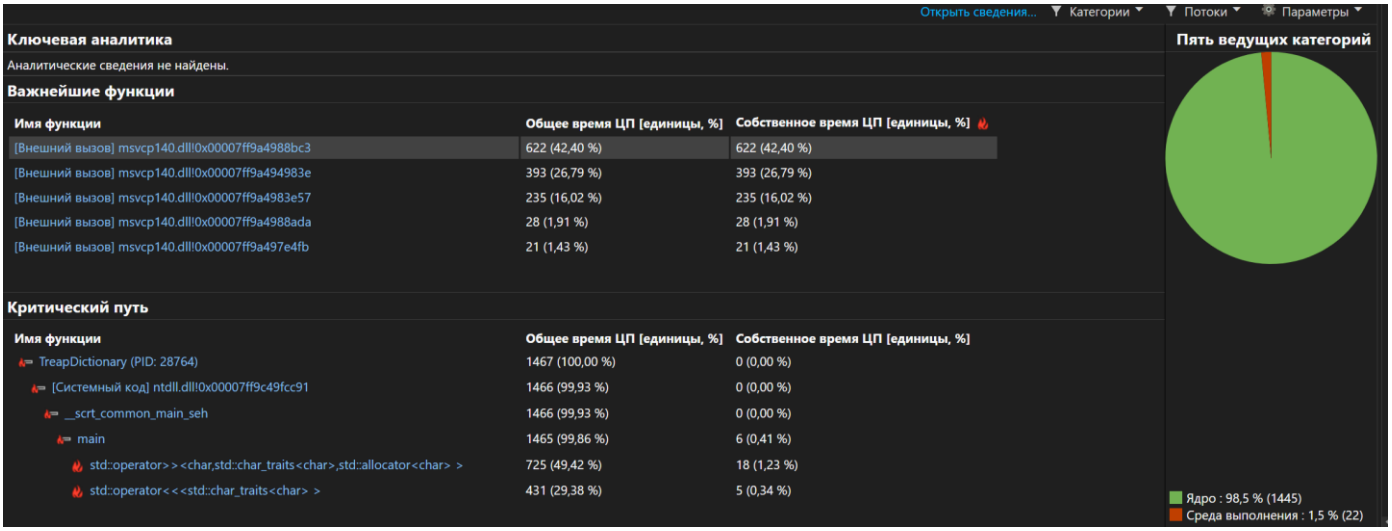
1. Для времени исполнения работы:

    a. **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:

    a. **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:

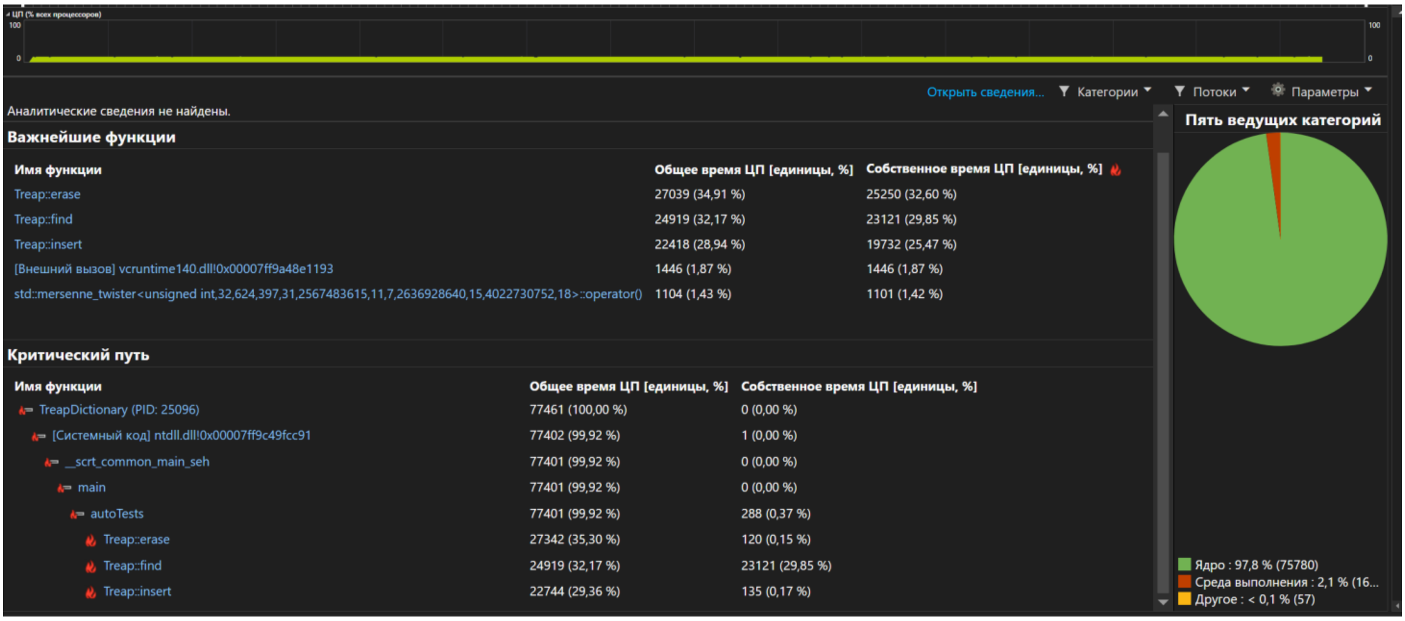


Для более детального анализа переходим по имени функции **TreapDictionary**, получаем следующий результат:

Текущее представление: Дерево вызовов | Развернуть критический путь | Показывать критический путь. | Сбросить корень

Имя функции	Общее время ЦП [единицы, %]	Собственное время ЦП [единиц...	Модуль	Категория
TreapDictionary (идентификатор процесса: ...	1467 (100,00 %)	0 (0,00 %)	Несколько модулей	
[Внешний вызов] ntdll.dll!0x00007ff9c49fc...	1466 (99,93 %)	0 (0,00 %)	ntdll	Ядро | Среда выполнения
__scrt_common_main_seh	1466 (99,93 %)	0 (0,00 %)	treapdictionary	Среда выполнения
main	1465 (99,86 %)	6 (0,41 %)	treapdictionary	Среда выполнения
std::operator>><char,std::char_tr...	725 (49,42 %)	18 (1,23 %)	treapdictionary	Среда выполнения
std::operator<<<std::char_traits<ch...	431 (29,38 %)	5 (0,34 %)	treapdictionary	
[Внешний вызов] msvcp140.dll!0x000...	235 (16,02 %)	235 (16,02 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	21 (1,43 %)	21 (1,43 %)	msvcp140	
Treap::insert	13 (0,89 %)	0 (0,00 %)	treapdictionary	Среда выполнения
Treap::insert	9 (0,61 %)	2 (0,14 %)	treapdictionary	
[Внешний вызов] ucrtbase.dll!0x000...	1 (0,07 %)	1 (0,07 %)	ucrtbase	Среда выполнения
[Внешний вызов] ucrtbase.dll!0x000...	1 (0,07 %)	1 (0,07 %)	ucrtbase	Среда выполнения
[Внешний вызов] ucrtbase.dll!0x000...	1 (0,07 %)	1 (0,07 %)	ucrtbase	Среда выполнения
operator new	1 (0,07 %)	0 (0,00 %)	treapdictionary	Среда выполнения
Treap::erase	10 (0,68 %)	0 (0,00 %)	treapdictionary	Среда выполнения
Treap::erase	6 (0,41 %)	1 (0,07 %)	treapdictionary	Среда выполнения
operator new	4 (0,27 %)	0 (0,00 %)	treapdictionary	Среда выполнения
Treap::find	7 (0,48 %)	1 (0,07 %)	treapdictionary	Среда выполнения
[Внешний вызов] ucrtbase.dll!0x000...	2 (0,14 %)	2 (0,14 %)	ucrtbase	Среда выполнения
operator new	2 (0,14 %)	0 (0,00 %)	treapdictionary	Среда выполнения
[Внешний вызов] ucrtbase.dll!0x000...	1 (0,07 %)	1 (0,07 %)	ucrtbase	Среда выполнения
[Внешний вызов] ucrtbase.dll!0x000...	1 (0,07 %)	1 (0,07 %)	ucrtbase	Среда выполнения
[Внешний вызов] msvcp140.dll!0x000...	3 (0,20 %)	3 (0,20 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	2 (0,14 %)	2 (0,14 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	2 (0,14 %)	2 (0,14 %)	msvcp140	
[Системный код] 0xfffff800c08de1a4	1 (0,07 %)	1 (0,07 %)	[Неизвестный код]	
[Внешний вызов] msvcp140.dll!0x000...	1 (0,07 %)	1 (0,07 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	1 (0,07 %)	1 (0,07 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	1 (0,07 %)	1 (0,07 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	1 (0,07 %)	1 (0,07 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	1 (0,07 %)	1 (0,07 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	1 (0,07 %)	1 (0,07 %)	msvcp140	
[Внешний вызов] msvcp140.dll!0x000...	1 (0,07 %)	1 (0,07 %)	msvcp140	
[Внешний вызов] vcruntime140.dll!0x0...	1 (0,07 %)	1 (0,07 %)	vcruntime140	
[Внешний вызов] vcruntime140.dll!0x0...	1 (0,07 %)	1 (0,07 %)	vcruntime140	
[Внешний вызов] ucrtbase.dll!0x00007ff...	1 (0,07 %)	1 (0,07 %)	ucrtbase	Среда выполнения
[Внешний вызов] ntdll.dll!0x00007ff9c4a25c1e	1 (0,07 %)	1 (0,07 %)	ntdll	Ядро

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:

Treap::erase	27342 (35,30 %)	120 (0,15 %)
Treap::find	24919 (32,17 %)	23121 (29,85 %)
Treap::insert	22744 (29,36 %)	135 (0,17 %)

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] 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:

```
WARNING: Visual Leak Detector detected memory leaks!
---------- Block 3 at 0x00000000C28EC770: 72 bytes ----------
  Leak Hash: 0x1A6B1545, Count: 1, Total 72 bytes
  Call Stack (TID 17636):
    ucrtbased.dll!malloc()
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\heap\new_scalar.cpp (35): TreapDictionary.exe!operator new() + 0xA bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (56): TreapDictionary.exe!Treap::insert() + 0xA bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (116): TreapDictionary.exe!Treap::insert() + 0x21 bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (145): TreapDictionary.exe!autoTests() + 0x1A bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (174): TreapDictionary.exe!main()
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl (79): TreapDictionary.exe!invoke_main()
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl (288): TreapDictionary.exe!__scrt_common_main_seh() + 0x5 bytes
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl (331): TreapDictionary.exe!__scrt_common_main()
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_main.cpp (17): TreapDictionary.exe!mainCRTStartup()
    KERNEL32.DLL!BaseThreadInitThunk() + 0x14 bytes
    ntdll.dll!RtlUserThreadStart() + 0x21 bytes
  Data:
    A0 8F 8F C2    31 02 00 00    73 6E 77 61    6D 73 65 68     ....1... snwamseh
    6D 76 00 00    00 00 00 00    0A 00 00 00    00 00 00 00     mv...... ........
    0F 00 00 00    00 00 00 00    2E 28 0B CE    54 C6 96 94     ........ .(..T...
    29 00 00 00    CD CD CD CD    00 00 00 00    00 00 00 00     )....... ........
    00 00 00 00    00 00 00 00                                   ........ ........
```

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:

```
---------- Block 4 at 0x00000000C28F8FA0: 16 bytes ----------
  Leak Hash: 0xD99416FA, Count: 1, Total 16 bytes
  Call Stack (TID 17636):
    ucrtbased.dll!malloc()
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\heap\new_scalar.cpp (35): TreapDictionary.exe!operator new() + 0xA bytes
    C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.37.32822\include\xmemory (88): TreapDictionary.exe!std::_Default_allocate_t
raits::_Allocate()
    C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.37.32822\include\xmemory (245): TreapDictionary.exe!std::_Allocate<16,std::
_Default_allocate_traits,0>() + 0xC bytes
    C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.37.32822\include\xmemory (975): TreapDictionary.exe!std::allocator<std::_Co
ntainer_proxy>::allocate()
    C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.37.32822\include\xmemory (1467): TreapDictionary.exe!std::_Container_proxy_
ptr12<std::allocator<std::_Container_proxy> >::_Container_proxy_ptr12<std::allocator<std::_Container_proxy> >() + 0x11 bytes
    C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.37.32822\include\xstring (2626): TreapDictionary.exe!std::basic_string<char
,std::char_traits<char>,std::allocator<char> >::_Construct<2,char const *>() + 0x14 bytes
    C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.37.32822\include\xstring (2492): TreapDictionary.exe!std::basic_string<char
,std::char_traits<char>,std::allocator<char> >::basic_string<char,std::char_traits<char>,std::allocator<char> >() + 0x29 bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (21): TreapDictionary.exe!Node::Node() + 0x16 bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (56): TreapDictionary.exe!Treap::insert() + 0x35 bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (116): TreapDictionary.exe!Treap::insert() + 0x21 bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (145): TreapDictionary.exe!autoTests() + 0x1A bytes
    C:\Users\User\source\repos\TreapDictionary\TreapDictionary\TreapDictionary.cpp (174): TreapDictionary.exe!main()
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl (79): TreapDictionary.exe!invoke_main()
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl (288): TreapDictionary.exe!__scrt_common_main_seh() + 0x5 bytes
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl (331): TreapDictionary.exe!__scrt_common_main()
    D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_main.cpp (17): TreapDictionary.exe!mainCRTStartup()
    KERNEL32.DLL!BaseThreadInitThunk() + 0x14 bytes
    ntdll.dll!RtlUserThreadStart() + 0x21 bytes
  Data:
    70 C7 8E C2    31 02 00 00    00 00 00 00    00 00 00 00     p...1... ........
```

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.

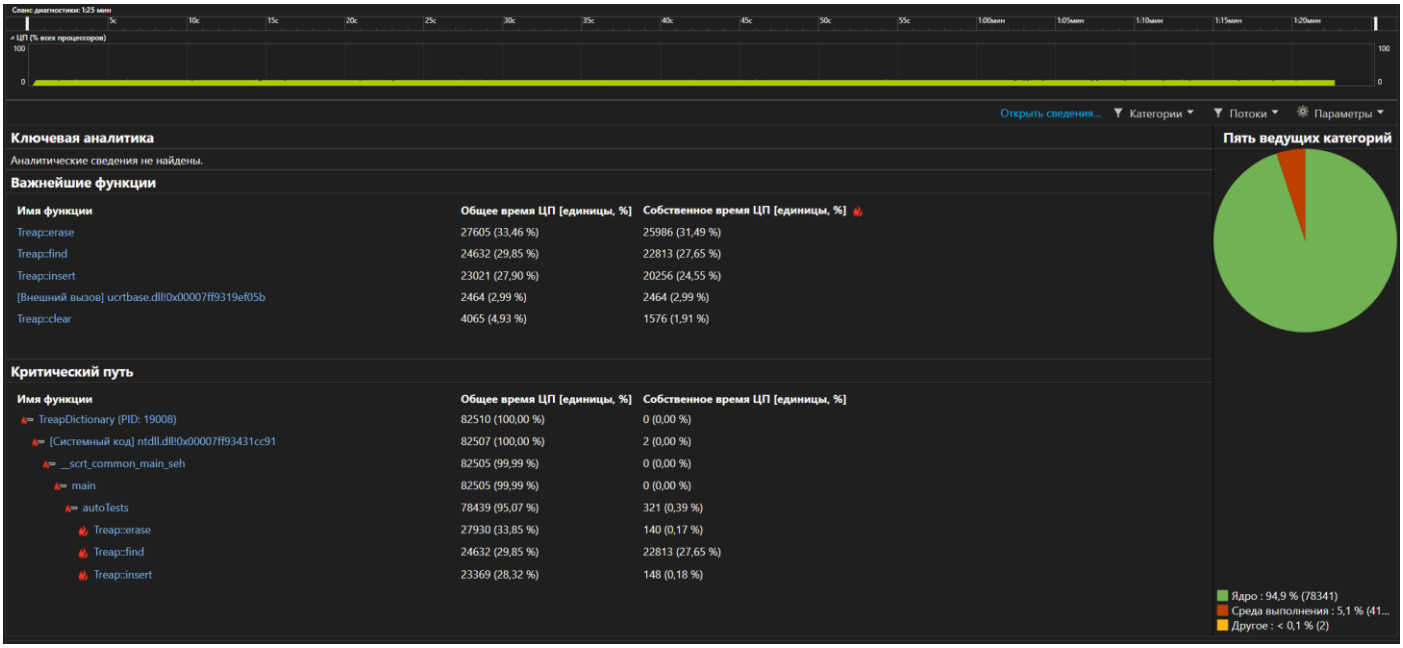
```
bool erase(Node*& current, const string& key) {
    if (!current) return false;  // Узел не найден
    if (key < current->key) {
        return erase(current->left, key);  // Идем в левое поддерево
    }
    if (key > current->key) {
        return erase(current->right, key); // Идем в правое поддерево
    }

    // Если ключи равны, удаляем текущий узел
    Node* old = current;
    current = merge(current->left, current->right);
    delete old;
    return true;
}
```

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

```
Node* find(Node* current, const string& key) const {
    while (current && current->key != key) {
        current = (key < current->key) ? current->left : current->right;
    }
    return current;
}
```

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%.

Treap::erase	27930 (33,85 %)	140 (0,17 %)	treapdictionary
Treap::find	24632 (29,85 %)	22813 (27,65 %)	treapdictionary
Treap::insert	23369 (28,32 %)	148 (0,18 %)	treapdictionary

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

```
    void clear(Node* node) {
        if (node) {
            clear(node->left);
            clear(node->right);
            delete node;
        }
    }

public:
    Treap() : root(nullptr) {}

    ~Treap() {
        clear(root);
    }
```

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

```
Visual Leak Detector read settings from: C:\Program Files (x86)\Visual Leak Detector\vld.ini
Visual Leak Detector Version 2.5.1 installed.
Insertion Time: 0 ms
Search Time: 0 ms
Deletion Time: 0 ms
No memory leaks detected.
Visual Leak Detector is now exiting.
```

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.