

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет «Компьютерные науки и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу

«Дискретный анализ»

Тема работы

“Исследование качества программ”

Студент : Ю.И. Катаев
Группа : М8О-210Б-21
Преподаватель : Н.К. Макаров
Оценка : _____
Дата : _____
Подпись : _____

Москва, 2022

1. Постановка задачи

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.

2. Описание

Для проведения профайлинга будем использовать утилиты `gprof` и `valgrind`. Поговорим подробнее о них и об их использовании.

Gprof - инструмент анализа производительности приложений. Он использует гибрид инструментария и выборки и был создан как расширенная версия более старого инструмента "prof". В отличие от `prof`, `gprof` способен собирать и распечатывать ограниченные графики вызовов. Чтобы начать пользоваться `gprof`, нужно сначала скомпилировать программу с ключом `-pg`:

```
g++ b-tree.cpp -pg
```

Затем запускаем программу так, как нам это нужно. В своём случае я сгенерировал файл `input.txt` в котором содержится 20000 команд добавления ключей, их проверки и удаления: `./a.out < input.txt`.

После выполнения будет сгенерирован файл `gmon.out`. В нём содержится вся информация, которую `gprof` смог собрать о программе во время её исполнения. Чтобы её просмотреть выполним команду:

```
gprof ./a.out
```

Valgrind - инструментальное программное обеспечение, предназначенное для отладки использования памяти, обнаружения утечек памяти, а также профилирования. Это один из самых популярных инструментов для профилирования программ. Чтобы приступить к его использованию, нам достаточно выполнить команду:

```
valgrind --leak-check=full ./a.out < in.txt
```

3. Ход работы

Запустим утилиту gprof.

```
→ src git:(main) × gprof ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self           self         total
time  seconds    seconds   calls   us/call   us/call   name
33.37    0.04     0.04    59999    0.67     0.67  Node::Search(char*)
25.03    0.07     0.03    60000    0.50     0.50  TolowerString(char*)
16.69    0.09     0.02     5636    3.55     3.55  Node::RemoveFromNonLeaf(int)
 8.34    0.10     0.01    25546    0.39     0.39  Node::Remove(char*)
 8.34    0.11     0.01    19909    0.50     1.00  Node::InsertNonFull(Data)
 8.34    0.12     0.01     6225    1.61     1.61  Node::Node(int, bool)
 0.00    0.12     0.00    31125    0.00     0.00  Data::Data()
 0.00    0.12     0.00    20000    0.00     2.17  BTree::DeleteNode(char*)
 0.00    0.12     0.00    20000    0.00     0.67  BTree::Search(char*)
 0.00    0.12     0.00    20000    0.00     1.67  BTree::AddNode(Data)
 0.00    0.12     0.00    19910    0.00     0.00  Node::RemoveFromLeaf(int)
 0.00    0.12     0.00    19470    0.00     0.00  Node::FillNode(int)
 0.00    0.12     0.00     8880    0.00     0.00  Node::BorrowFromPrev(int)
 0.00    0.12     0.00     6217    0.00     1.61  Node::SplitChild(int, Node*)
 0.00    0.12     0.00     6217    0.00     0.00  Node::Merge(int)
 0.00    0.12     0.00     5974    0.00     0.00  Node::BorrowFromNext(int)
 0.00    0.12     0.00        1    0.00     0.00  _GLOBAL__sub_I__ZN4NodeC2Eib
 0.00    0.12     0.00        1    0.00     0.00  __static_initialization_and_dest
ruction_0(int, int)
 0.00    0.12     0.00        1    0.00     0.00  BTree::BTree(int)
 0.00    0.12     0.00        1    0.00     0.00  BTree::~~BTree()
```

Проанализировав таблицу, можно увидеть, что чаще всего программе приходится вызывать функцию Search. Это можно объяснить тем, что я вызываю функцию Search при вставке и удалении элемента, чтобы удостовериться в том, что его нет в дереве. Следующая по частоте использования функция - TolowerString. Это очевидно, ведь при считывании любого ключа, его приходится приводить к нижнему регистру. Тройку замыкает функция RemoveFromNonLeaf. Эта функция действительно очень объемная, т.к. Прежде чем удалить из внутреннего узла, нужно проверить, что в его детях достаточное количество элементов

Протестируем утилиту valgrind:

```
==456== Memcheck, a memory error detector
==456== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==456== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info==456== Command: ./a.out
==456==
==456==
==456== HEAP SUMMARY:
==456==   in use at exit: 421,680 bytes in 6,231 blocks
==456== total heap usage: 18,682 allocs, 12,451 frees, 9,159,584 bytes allocated
==456==
==456== LEAK SUMMARY:
==456==   definitely lost: 298,800 bytes in 6,225 blocks
==456==   indirectly lost: 0 bytes in 0 blocks
==456==   possibly lost: 0 bytes in 0 blocks
==456==   still reachable: 122,880 bytes in 6 blocks
==456==   suppressed: 0 bytes in 0 blocks
==456== Rerun with --leak-check=full to see details of leaked memory
==456==
==456== For lists of detected and suppressed errors, rerun with: -s
==456== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Утилита показала, что у меня есть утечки памяти. Используя флаг, **--leak-check=full**, я понял, что я не очищал массив указателей на детей в функциях Merge (когда склеил две ноды в одну, а память под уже склеенной ноды не удалил) и DeleteNode (в случае, когда корень оказался пуст)

Вот результат после исправлений:

```
==599== Memcheck, a memory error detector
==599== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==599== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==599== Command: ./a.out
==599==
==599==
==599== HEAP SUMMARY:
==599==   in use at exit: 0 bytes in 0 blocks
==599== total heap usage: 18,678 allocs, 18,678 frees, 9,044,896 bytes allocated
==599==
==599== All heap blocks were freed -- no leaks are possible
==599==
==599== For lists of detected and suppressed errors, rerun with: -s
==599== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

0 errors from 0 contexts означает, что ни одной утечки не найдено и программа работает без утечек, чего мы и добивались.

4. Выводы

Проделав данную лабораторную работу, я познакомился с очень полезными утилитами valgrind и gprof, которые помогают отлаживать свою программу путем анализа времени и памяти программы. Анализ времени позволяет увидеть, сколько времени занимает каждая из функций, что поможет для оптимизации кода. Анализ потребления памяти позволяет выявить

неочевидные утечки памяти, которые благодаря утилите можно будет исправить. Благодаря этим утилитам, код получается более быстрым и стабильным.