Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет «Компьютерные науки и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу

«Дискретный анализ»

Тема работы

“Исследование качества программ”

Студент : А.А. Дудовцев

Группа : М8О-208Б-22

Оценка : _____

Дата : _____

Подпись : _____

Москва, 2024

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.

```
qwz@qwz-VirtualBox:~/DA2-3$ 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
 50.34      0.75     0.75 13682660     0.05     0.05  Node::Search(char*)
 18.79      1.03     0.28   154891     1.81     6.12  Node::SplitChild(int, Node*)
  7.38      1.14     0.11   500003     0.22     0.38  main
  7.38      1.25     0.11                             Node::BorrowFromNext(int)
  6.71      1.35     0.10  4619113     0.02     0.02  Node::InsertNonFull(Data)
  3.36      1.40     0.05                             _init
  2.01      1.43     0.03  4325443     0.01     0.25  Node::Remove(char*)
  2.01      1.46     0.03   634939     0.05     0.05  Node::RemoveFromNonLeaf(int)
  1.34      1.48     0.02   154891     0.13     0.13  Node::Merge(int)
  0.67      1.49     0.01                             Node::RemoveFromLeaf(int)
  0.00      1.49     0.00  3274509     0.00     0.00  _fini
  0.00      1.49     0.00   516692     0.00     0.05  Node::FillNode(int)
  0.00      1.49     0.00   500003     0.00     0.02  BTree::Search(char*)
  0.00      1.49     0.00   500002     0.00     0.05  BTree::AddNode(Data)
  0.00      1.49     0.00   500000     0.00     0.11  BTree::DeleteNode(char*)
  0.00      1.49     0.00   236101     0.00     0.00  Node::BorrowFromPrev(int)
  0.00      1.49     0.00   154901     0.00     0.00  Node::Node(int, bool)
```

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 удалят из внутреннего узла, нужно проверять, что в его детях достаточное количество элементов.

Протестируем утилиту valgrind:

Утилита показала, что у меня есть утечки памяти. Используя флаг, --leak-check=full, я понял, что я не очищал массив указателей на детей в фунциях 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:

```
==358820== Memcheck, a memory error detector
==358820== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==358820== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==358820== Command: ./a.out
==358820==
==358820==
==358820== HEAP SUMMARY:
==358820==     in use at exit: 122,880 bytes in 6 blocks
==358820==   total heap usage: 7 allocs, 1 frees, 195,584 bytes allocated
==358820==
==358820== LEAK SUMMARY:
==358820==    definitely lost: 0 bytes in 0 blocks
==358820==    indirectly lost: 0 bytes in 0 blocks
==358820==      possibly lost: 0 bytes in 0 blocks
==358820==    still reachable: 122,880 bytes in 6 blocks
==358820==         suppressed: 0 bytes in 0 blocks
==358820== Reachable blocks (those to which a pointer was found) are not shown.
==358820== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==358820==
==358820== For lists of detected and suppressed errors, rerun with: -s
==358820== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

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 утилитом, код получается более быстрым и стабильным.