Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: Е. С. Пищик
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

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.

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

Вариант дерева: AVL-дерево.

1 Дневник отладки

1. Напишем файл benchmark.cpp в котором замерим время работы std::map и AVL-дерева при помощи библиотеки <chrono> (пункт: скорость выполения).

2. При помощи valgrind оценим утечки и расход памяти для std::map и собственного AVL-дерева (пункт: потребление оперативной памяти).

3. В собственной реализации были найдены некоторые незначительные утечки памяти: still reachable: 122,880 bytes in 6 blocks, они возникают из-за использования оптимизаций, таких как std::ios::sync_with_stdio(false) и std::cin.tie(nullptr).

4. При помощи gprof выяснили, что наибольшее время занимают операции со сравнением и копированием ключа, т.е. строки. Сами операции AVL-дерева выполняются достаточно быстро.

2 Скорость выполнения

Тест представляет из себя следующее: создаем объекты std::map и наш avl. Вставляем в оба объекта по 1 млн. элементов с ключом=значению в диапазоне от 0 до 999999. Измеряем время работы для std::map и avl. Далее 1 млн. раз ищем элемент с ключом=значению=999999 и замеряем время для std::map и avl. Последний тест - 1 млн. раз удаляем значение(от 999999 до 0) из std::map и avl, замеряем время.

```
pe4eniks$ ./benchmark
Insert map time: 6.99499 seconds
Insert avl time: 12.1846 seconds
Find map time: 7.5032 seconds
Find avl time: 3.30145 seconds
Delete map time: 10.0912 seconds
Delete avl time: 9.73254 seconds
```

Как видно, что удаление в avl работает совсем чуть-чуть быстрее чем в std::map, вставка в avl работает значительно более медленно, чем в std::map, поиск в avl работает значительно быстрее чем в std::map.

3 gprof

Скомпилируем программу с флагом -pg. Запустим нашу программу с 3 млн. команд, где 1 млн. вставок, 1 млн. поисков и 1 млн. удалений. После этого в текущей директории создался файл gmon.out, чтобы посмотреть результат профилирования выполним команду gprof solution gmon.out.

```
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  ns/call  ns/call  name
33.57      0.52     0.52 60652960     8.58     8.58  operator<(TData const&,TData
const&)
13.23      0.73     0.21  1000000   205.10   623.59  TAvl::InsertPrint(TAvlNode*,
TData,unsigned long,bool const&)
12.91      0.93     0.20 37214943     5.38     5.38  TAvl::ReBalance(TAvlNode*)
10.97      1.10     0.17 45652960     3.73     3.73  TData::TData(TData const&)
6.45      1.20     0.10  1000000   100.05   407.94  TAvl::RemovePrint(TAvlNode*,
TData,bool const&)
5.16      1.28     0.08 49572619     1.61     1.61  operator>(TData const&,TData
const&)
5.16      1.36     0.08  2000000    40.02    74.35  operator>>(std::istream&,TData&)
2.58      1.40     0.04  5000000     8.00     8.00  TData::TData(char const*)
1.94      1.43     0.03  5241177     5.73     5.73  TData::operator=(TData
const&)
1.94      1.46     0.03  1000000    30.02   264.44  TAvl::Find(TAvlNode*,TData)
1.94      1.49     0.03                             TAvlFinal::DInsert()
1.29      1.51     0.02                             TAvlFinal::DRemove()
0.65      1.52     0.01  2000000     5.00     5.00  TData::TData()
0.65      1.53     0.01  1043058     9.59     9.59  TAvl::RotateLeft(TAvlNode*)
0.65      1.54     0.01                             TAvlFinal::DFind(TData
const&)
0.65      1.55     0.01                             operator<<(std::ostream&,TData
const&)
0.32      1.55     0.01   355937    14.05    23.65  TAvl::RotateLeftRight(TAvlNode*)
0.00      1.55     0.00   696397     0.00     0.00  TAvl::RotateRight(TAvlNode*)
0.00      1.55     0.00   241177     0.00     7.85  TAvl::RemoveMin(TAvlNode*,
TAvlNode*)
0.00      1.55     0.00    28583     0.00     0.00  TAvl::RotateRightLeft(TAvlNode*)
```

По данной таблице можно понять, что больше всего врмени тратится на operator<, который используется для сравнения строк, другие операторы сравнений также тратят достаточно много времени, как и конструкторы копирования для строк. Также достаточно много времени тратится на вставку, перебалансировку и удаление, но этого не избежать. Отсюда можно сделать вывод, что очень много времени тратится не на сортировку, а на сравнение/копирование строк, т.е. на вспомогательные задачи.

Дальше выводится граф вызовов (тут я вывел его в укороченном виде), в каждой строке графа указывается функция, в предыдущих строках выводятся функции вызывавшие данную функцию, а в последующих строках функции вызываемые данной функцией. Таким образом можно определить, где происходят вызовы наиболее долгих функций, ведь сама функция может выполнятся быстро, но функции вызываемые ей могут тратить много времени.

```
index % time    self  children    called     name
<spontaneous>
[1]     47.7    0.03    0.71                 TAvlFinal::DInsert() [1]
0.21    0.42 1000000/1000000     TAvl::InsertPrint(TAvlNode*,TData,unsigned
long,bool const&) [2]
0.04    0.03 1000000/2000000     operator>>(std::istream&,TData&) [10]
0.01    0.00 2000000/45652960     TData::TData(TData const&) [9]
0.01    0.00 1000000/2000000     TData::TData() [14]
-----------------------------------------------
21140998             TAvl::InsertPrint(TAvlNode*,TData,unsigned long,bool const&)
[2]
0.21    0.42 1000000/1000000     TAvlFinal::DInsert() [1]
[2]     40.2    0.21    0.42 1000000+21140998 TAvl::InsertPrint(TAvlNode*,TData,
unsigned long,bool const&) [2]
0.18    0.00 21140998/60652960     operator<(TData const&,TData const&) [3]
0.11    0.00 21140998/37214943     TAvl::ReBalance(TAvlNode*) [8]
0.09    0.00 23140998/45652960     TData::TData(TData const&) [9]
0.02    0.00 14922953/49572619     operator>(TData const&,TData const&) [11]
0.00    0.00  323460/355937      TAvl::RotateLeftRight(TAvlNode*) [17]
0.01    0.00  574773/1043058     TAvl::RotateLeft(TAvlNode*) [15]
0.00    0.00  344683/696397      TAvl::RotateRight(TAvlNode*) [26]
0.00    0.00    8271/28583       TAvl::RotateRightLeft(TAvlNode*) [27]
21140998             TAvl::InsertPrint(TAvlNode*,TData,unsigned long,bool const&)
[2]
-----------------------------------------------
0.14    0.00 16511962/60652960     TAvl::RemovePrint(TAvlNode*,TData,
bool const&)[5]
0.18    0.00 21140998/60652960     TAvl::InsertPrint(TAvlNode*,TData,unsigned
```

```
long,bool const&) [2]
0.20    0.00 23000000/60652960     TAvl::Find(TAvlNode*,TData) [7]
[3]     33.5    0.52    0.00 60652960         operator<(TData const&,TData
const&) [3]
-----------------------------------------------
<spontaneous>
[4]     33.2    0.02    0.49                 TAvlFinal::DRemove() [4]
0.10    0.31 1000000/1000000     TAvl::RemovePrint(TAvlNode*,TData,
bool const&) [5]
0.04    0.03 1000000/2000000     operator>>(std::istream&,TData&) [10]
0.01    0.00 2000000/45652960     TData::TData(TData const&) [9]
0.01    0.00 1000000/2000000     TData::TData() [14]
-----------------------------------------------
15511962             TAvl::RemovePrint(TAvlNode*,TData,bool const&) [5]
0.10    0.31 1000000/1000000     TAvlFinal::DRemove() [4]
[5]     26.3    0.10    0.31 1000000+15511962 TAvl::RemovePrint(TAvlNode*,TData,
bool const&) [5]
0.14    0.00 16511962/60652960     operator<(TData const&,TData const&) [3]
0.09    0.00 15994316/37214943     TAvl::ReBalance(TAvlNode*) [8]
0.06    0.00 15511962/45652960     TData::TData(TData const&) [9]
0.02    0.00 11649666/49572619     operator>(TData const&,TData const&) [11]
0.00    0.00  241177/241177      TAvl::RemoveMin(TAvlNode*,TAvlNode*) [18]
0.00    0.00  103576/1043058     TAvl::RotateLeft(TAvlNode*) [15]
0.00    0.00   32477/355937      TAvl::RotateLeftRight(TAvlNode*) [17]
0.00    0.00  323131/696397      TAvl::RotateRight(TAvlNode*) [26]
0.00    0.00   19795/28583       TAvl::RotateRightLeft(TAvlNode*) [27]
15511962             TAvl::RemovePrint(TAvlNode*,TData,bool const&) [5]
-----------------------------------------------
```

По данному графу можно увидеть, что например функция DInsert сама выполняется достаточно быстро, но она вызывает функцию InsertPrint, которая выполняется достаточно продолжительное время.

4 Потребление оперативной памяти

Тест представляет из себя следующее: создаем объекты std::map и наш avl. Вставляем в оба объекта по 1 млн. элементов с ключом=значению в диапазоне от 0 до 999999, 1 млн. раз ищем элемент с ключом=значению=999999, 1 млн. раз удаляем значение(от 999999 до 0) из std::map и avl, запускаем с использованием valgrind сначала для avl потом для std::map и смотрим на использование и утечки памяти. Файл memory_test.txt представляет из себя 3 млн. команд, 1 млн. вида «+ i i», 1 млн. вида «i» и 1 млн. вида «- i». Исполняемый файл «map» получается при компиляции файла map.cpp в котором выполняется по 1 млн. операций вставки, поиска и удаления в std::map.

```
pe4eniks@pe4eniks-HP-Laptop-14-dk0xxx:~/solution/solution$ valgrind ./solution
<memory_test.txt
==2127== Memcheck,a memory error detector
==2127== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==2127== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2127== Command: ./solution
==2127==
==2127==
==2127== HEAP SUMMARY:
==2127==     in use at exit: 122,880 bytes in 6 blocks
==2127==   total heap usage: 56,894,144 allocs,56,894,138 frees,422,797,359
bytes allocated
==2127==
==2127== LEAK SUMMARY:
==2127==    definitely lost: 0 bytes in 0 blocks
==2127==    indirectly lost: 0 bytes in 0 blocks
==2127==      possibly lost: 0 bytes in 0 blocks
==2127==    still reachable: 122,880 bytes in 6 blocks
==2127==         suppressed: 0 bytes in 0 blocks
==2127== Rerun with --leak-check=full to see details of leaked memory
==2127==
==2127== For counts of detected and suppressed errors,rerun with: -v
==2127== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)


pe4eniks@pe4eniks-HP-Laptop-14-dk0xxx:~/solution/solution$ valgrind ./map
==2219== Memcheck,a memory error detector
==2219== Copyright (C) 2002-2017,and GNU GPL'd,by Julian Seward et al.
==2219== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
```

```
==2219== Command: ./map
==2219==
==2219==
==2219== HEAP SUMMARY:
==2219==     in use at exit: 0 bytes in 0 blocks
==2219==   total heap usage: 1,000,001 allocs,1,000,001 frees,72,072,704 bytes
allocated
==2219==
==2219== All heap blocks were freed --no leaks are possible
==2219==
==2219== For counts of detected and suppressed errors,rerun with: -v
==2219== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Как видно из тестов, в собственной реализации AVL-дерева есть незначительные утечки памяти, значительно больше количество аллокаций, очищений памяти, а объем аллоцированной памяти примерно в 6 раз больше чем в std::map. Как можно видеть число аллокаций и очисток в std::map совпадает, т.к. нет ни одной утечки памяти, в отличие от собственного AVL-дерева, где число аллокаций больше чем число очищений, разница равняется 6 блокам, которые отображаются в still reachable.

5 Выводы

Выполнив третью лабораторную работу по курсу «Дискретный анализ», я познакомился с очень полезными инструментами.

1. Valgrind позволяет оценивать работу с памятью в программе (искать утечки памяти, смотреть использование памяти).

2. Gprof позовляет оценить производительность работы программы.

При помощи данных инструментов удобно исправлять и улучшать свой код.