Лабораторная работа № 3 по курсу дискретного анализа: Исследование качества программ

Выполнила студентка группы 08-208 МАИ *Шевлякова София.*

Условие

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.

Метод решения

Изучение утилит valgrind и gprof для исследования качества программ и использование их для оптимизации программы.

- Valgrind — инструментальное ПО, предназначенное в основном для контроля использования памяти и обнаружения её утечек. С помощью этой утилиты можно обнаружить попытки использования (обращения) к неициализированной памяти, работа с памятью после её освобождения и некоторые другие.
- Утилита gprof позволяет измерить время работы всех функций, методов и операторов программы, количество их вызовов и долю от общего времени работы программы в процентах.

Valgrind

Valgrind — инструментальное программное обеспечение, предназначенное для отладки использования памяти, обнаружения утечек памяти, а также профилирования.
Первоначальная версия программы содержала ошибку:

```
sonikxx@LAPTOP-9UGJH447:~/DA/lab3$ valgrind --leak-check=full ./a.out < 1e2.txt |
    cat > output.txt
==5401== Memcheck, a memory error detector
==5401== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5401== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5401== Command: ./a.out
==5401==
==5401==
==5401== HEAP SUMMARY:
==5401==    in use at exit: 7,967 bytes in 31 blocks
==5401==  total heap usage: 65 allocs, 34 frees, 90,103 bytes allocated
==5401==
==5401== 7,967 bytes in 31 blocks are definitely lost in loss record 1 of 1
```

```
==5401==   at 0x483C583: operator new[](unsigned long) (in
   /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5401==   by 0x10EC9C: TTreap::insert(char const*, unsigned long) (in
   /home/sonikxx/DA/lab3/a.out)
==5401==   by 0x109B38: main (in /home/sonikxx/DA/lab3/a.out)
==5401==
==5401== LEAK SUMMARY:
==5401==   definitely lost: 7,967 bytes in 31 blocks
==5401==   indirectly lost: 0 bytes in 0 blocks
==5401==     possibly lost: 0 bytes in 0 blocks
==5401==   still reachable: 0 bytes in 0 blocks
==5401==        suppressed: 0 bytes in 0 blocks
==5401==
==5401== For lists of detected and suppressed errors, rerun with: -s
==5401== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Ошибку заметил *valgrind* и запуская его с флагом *–leak-check=full* можно понять, где была создана память, которая не освобождалась. В функции вставки создаётся новый объект с помощью *new*, но он не освобождается при удалении вершины.

Для этого добавим деконструктор структуре *node*, который освободит память, созданную при вставке.

Запустим программу со внесёнными изменениями, чтобы убедиться, что других ошибок нет.

```
sonikxx@LAPTOP-9UGJH447:~/DA/lab3$ valgrind ./a.out < 1e6.txt | cat > output.txt
==4838== Memcheck, a memory error detector
==4838== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4838== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4838== Command: ./a.out
==4838==
==4838==
==4838== HEAP SUMMARY:
==4838==    in use at exit: 0 bytes in 0 blocks
==4838==  total heap usage: 335,633 allocs, 335,633 frees, 49,921,951 bytes
   allocated
==4838==
==4838== All heap blocks were freed -- no leaks are possible
==4838==
==4838== For lists of detected and suppressed errors, rerun with: -s
==4838== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Вывод программы говорит, что не было ошибок во время исполнения, и утечек памяти не обнаружено.

Gprof

Используя утилиту gprof, можем отследить, где и сколько времени проводила программа, тем самым выявляя слабые участки. Возьмем достаточно большой тест (1000000 строк) и вызовем gprof:

```
sonikxx@LAPTOP-9UGJH447:~/DA/lab3$g++ -pg main.cpp
sonikxx@LAPTOP-9UGJH447:~/DA/lab3$ ./a.out < ../testing/test.txt | cat > output.txt
sonikxx@LAPTOP-9UGJH447:~/DA/lab3$ gprof a.out | cat > profile.info
```

Оформим вывод программы в виде таблицы:

% time	total seconds	self seconds	cals	self ms/cals	total ms/cals	name
46.31	0.06	0.06	1999998	30.10	30.10	TTreap::split
23.15	0.09	0.03	1999998	15.05	15.05	TTreap::merge
15.44	0.11	0.02	-	-	-	main
11.58	0.13	0.02	999999	15.05	75.25	TTreap::cut
3.86	0.13	0.01	999999	5.02	35.12	TTreap::join
0.00	0.13	0.00	999999	0.00	0.00	StringToLower
0.00	0.13	0.00	339491	0.00	110.37	TTreap::insert
0.00	0.13	0.00	330301	0.00	110.37	TTreap::remove
0.00	0.13	0.00	330207	0.00	110.37	TTreap::find
0.00	0.13	0.00	167752	0.00	0.00	TTreap::node::node
0.00	0.13	0.00	167752	0.00	0.00	TTreap::node::~node
0.00	0.13	0.00	1	0.00	0.00	TTreap::destroy
0.00	0.13	0.00	1	0.00	0.00	TTreap::TTreap
0.00	0.13	0.00	1	0.00	0.00	TTreap::~TTreap

Из таблицы мы видим, что чаще всего вызываются функции split и merge, что логично, ведь они лежат в основе функций cut и join, которые вызываются каждый раз при поиске, вставке и удалении элемента.

Для ускорения работы программы необходимо минимизировать количество вызовов функции split, так как она имеет самое больше время self ms/cals (время одного вызова).

Например, функцию поиска можно реализовывать без split, используя обычный поиск в бинарном дереве.

Выводы

При выполнении лабораторной работы я познакомилась с профилированием, крайне необходимым для качественной разработки, изучила возможные методы работы с ним, применив их на практике. Ранее я не использовала утилиты Valgrind для контроля утечек памяти, а также gprof, которая выводит число вызовов функций при работе программы, определяет время работы каждой функции как обособленно, так и в сравнении с общим временем работы программы, что позволяет найти наиболее часто используемую функцию и в первую очередь оптимизировать именно её.