Московский авиационный институт
(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Дискретный анализ»

Студент: А. Довженко
Преподаватель: Д. Е. Ильвохин
Группа: 08-207
Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №1

Задача: Требуется разработать программу, осуществляющую ввод пар ключ-значение, их упорядочивание по возрастанию ключа указанным алгоритмом сортировки за линейное время и вывод отсортированной последовательности.

Вариант сортировки: Поразрядная сортировка.

Вариант ключа: MD5-суммы (32-разрядные шестнадцатиричные числа).

Вариант значения: Числа от 0 до $2^{64} - 1$.

1 Описание

Как известно, поразрядная сортировка относится к сортировкам за линейное время, которые достигают такой сложности благодаря отсутствию в них операций сравнения. У этой сортировки есть несколько вариантов реализации, например, зависящих от того, с каких разрядов начат проход – с младших или со старших. Я выбрала LSD (least significant digit), т.е. сортировка проходит от младших разрядов к старшим. Это приводит к тому, что короткие ключи идут раньше длинных, а ключи одного размера сортируются по алфавиту. Важно, чтобы сортировка разряда обладала устойчивостью.

Суть поразрядной сортировки заключается в том, что d-значные числа последовательно сортируются по разрядам: в моем случае от младшего к старшему. При использовании внутренней устойчивой и линейной сортировки происходит корректное упорядочивание, и поразрядная сортировка тоже становится линейной. То есть поразрядная сортировка линейна в тех случаях, когда в ней в качестве внутренней применяется какая-то линейная сортировка. Следуя традициям классики, в качестве внутренней я беру сортировку подсчетом, поэтому мне понадобилось привлечение дополнительной памяти. Ее идея проста: чтобы упорядочить последовательность от 0 до n, члены которой лежат в интервале 0...k, достаточно "подсчитать"их количество в массиве count[0...k], инкрементируя count[i] для обнаруженного элемента i. Дополнительный обход позволит узнать, сколько существует элементов count[i], не превосходящих i. Для установления и сохранения нового порядка потребуется массив out[0...n], заполняемый в обратном порядке: для i = n...0: count[i] = count[i] - 1, out[count[i]] = out[i]. Декремент count[i] отвечает за корректное расположение равных элементов: планомерное декрементирование count[i] "сдвигает"позицию следующего элемента с таким же ключом влево, что на пару с обратным порядком обхода обеспечивает устойчивость сортировки.

Задача сводится к тому, чтобы заполнить вектор записями вида Ключ-Значение и произвести упорядочение записей по ключу.

2 Исходный код

main.c	
Нет функций	-
vector.c	
TVector VectorCreate(void);	Создание вектора.
void VectorDestroy(TVector *vec);	Удаление вектора.
void VectorInsert(TVector vec, char *key, TVal val);	Вставка элемента в вектор.
void VectorResize(TVector vec);	Увеличение вектора.
void VectorPrint(TVector vec);	Печать вектора.
void ReadItem(TVector vec);	Считывание элемента со стандартного потока ввода.
void VectorRadixSort(TVector vec);	Поразрядная сортировка вектора.

```
1  struct item {
2      char key[33];
3      TVal value;
4  }; //*TVectorItem
5
6  struct vector {
7      uint32_t size;
8      uint32_t freespace;
9      TVectorItem array;
10 }; //*TVector;
```

3 Консоль

```
karma@karma:~/mai_study/DA/lab1$ make
gcc    -c -o vector.o vector.c
gcc    -c -o main.o main.c
gcc -std=c99 -w -pipe -O3 -Wextra -Werror -Wall -Wno-sign-compare -pedantic
-lm vector.o main.o -o run
karma@karma:~/mai_study/DA/lab1$ cat test
1d6bfbe2c3c42c56b311487a656d4584 10253866777938993906
2133e23d7cca77d3c6a8a34a5923735c 6467819907924557406
8d332bea3ee2556b5e1e4b65e87517b6 3956118187340415490
ce36a52c7bc8c17a996579fc3578e4de 4374577838840431210
8435f7eec5c2b78c779dce4a16e1f7b7 14383281951375636185
784f629ef791da5a9e555bad5394e785 5498355550932583612
41155225ad9eccdc2819fe22126944b4 2738006378864610962
a4c64615267fd861cfead96e36d4d2eb 3487057267340666166
c2928cdf5b8fcf791a8b9f7a48c5f4ce 128736790059447787
karma@karma:~/mai_study/DA/lab1$ valgrind --leak-check=full ./run <test
==11085== Memcheck,a memory error detector
==11085== Copyright (C) 2002-2015,and GNU GPL'd,by Julian Seward et al.
==11085== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==11085== Command: ./run
==11085==
1d6bfbe2c3c42c56b311487a656d4584 10253866777938993906
2133e23d7cca77d3c6a8a34a5923735c 6467819907924557406
41155225ad9eccdc2819fe22126944b4 2738006378864610962
784f629ef791da5a9e555bad5394e785 5498355550932583612
8435f7eec5c2b78c779dce4a16e1f7b7 14383281951375636185
8d332bea3ee2556b5e1e4b65e87517b6 3956118187340415490
a4c64615267fd861cfead96e36d4d2eb 3487057267340666166
c2928cdf5b8fcf791a8b9f7a48c5f4ce 128736790059447787
ce36a52c7bc8c17a996579fc3578e4de 4374577838840431210
==11085==
==11085== HEAP SUMMARY:
==11085==     in use at exit: 0 bytes in 0 blocks
==11085==   total heap usage: 9 allocs,9 frees,7,056 bytes allocated
==11085==
==11085== All heap blocks were freed --no leaks are possible
==11085==
==11085== For counts of detected and suppressed errors,rerun with: -v
==11085== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

4 Тест производительности

Тест производительности представляет из себя следующее: я случайным образом генерирую пять файлов (на 100 записей, на 1к записей, на 100к записей, на 1кк записей и на 15кк записей – input100, input1k, input100k, input1kk, input15kk соответственно). Также генерирую файл с 10000 уже отсортированными строками (inputsort) и файл с 10000 строк, отсортированных по убыванию (inputreverse). Каждая строка состоит из ключа и значения согласно варианту задания. В качестве эталонной сортировки беру qsort из <stdlib.h>. Ее сложность составляет O(n lg(n)).

```
karma@karma:~/mai_study/DA/lab1$ ./run <input100
Type sort: radix
Time of working 0.000099 sec.
Type sort: qsort
Time of working 0.000017 sec.
karma@karma:~/mai_study/DA/lab1$ ./run <input1k
Type sort: radix
Time of working 0.001242 sec.
Type sort: qsort
Time of working 0.000198 sec.
karma@karma:~/mai_study/DA/lab1$ ./run <input100k
Type sort: radix
Time of working 0.140591 sec.
Type sort: qsort
Time of working 0.035068 sec.
karma@karma:~/mai_study/DA/lab1$ ./run <input1kk
Type sort: radix
Time of working 1.437047 sec.
Type sort: qsort
Time of working 0.582659 sec.
karma@karma:~/mai_study/DA/lab1$ ./run <input15kk
Type sort: radix
Time of working 21.222748 sec.
Type sort: qsort
Time of working 12.892512 sec.
karma@karma:~/mai_study/DA/lab1$ ./run <inputsort
Type sort: radix
Time of working 0.016643 sec.
Type sort: qsort
Time of working 0.000598 sec.
karma@karma:~/mai_study/DA/lab1$ ./run <inputreverse
```

```
Type sort: radix
Time of working 0.011865 sec.
Type sort: qsort
Time of working 0.001225 sec.
```

Как видно из тестов, моя поразрядная сортировка проигрывает быстрой сортировке из <stdlib.h> как на больших данных, так и на небольших. Так же можно заметить, что отсортированная последовательность данных является одним из "плохих"случаев для поразрядной сортировки, что связанно с большим количеством лишних операций внутри сортировки. Результат неудивителен, ведь люди, писавшие стандартную быструю сортирвку, гораздо опытнее и профессиональнее меня.

5 Отладка

1 посылка на чекер: Runtime error at test 10.t, got signal 11.
При перевыделении памяти была допущена опечатка и при вычислении нового участка необходимой памяти вместо vec->size (размер вектора) блок памяти умножался на vec->freespace (количество свободных ячеек памяти), который был равен 0.
2 и 3 посылки на чекер: Time limit exceeded at test 12.t.
Перевыделение памяти происходило сначала с шагом 100 (т.е. новый участок памяти, занимаемой вектором, был на 100 элементов больше старого), потом с шагом 2000. Оказалось, что такие итерации плохо влияют на временную сложность. Теперь память выделяется экспоненциально.

6 Выводы

Все в этом мире несовершенно. И мой код не исключение. Среди недочетов можно отметить небольшую скорость выполнения сортировки, недостаточное покрытие кода проверками, нерациональный расход памяти при хранении ключа и значения. К плюсам можно отнести надежность выполнения, сортировка по сути ограничена только размером оперативной памяти, код ортогонален.

Сложность моего алгоритма составляет O(k(s+n+s+n)), где k – постоянное количество разрядов ключа (k = 32), s – система счисления (s = 16), n – количество входных данных. Как мы видим, в некоторых случаях сортировка подсчетом может оказаться медленнее сортировок сравнением, когда O(nk) > O(n lg(n)). Поразрядная сортировка имеет свои минусы и плюсы. К последним можно отнести сортировку данных любой длины, что не может позволить себе та же сортировка подсчетом за линейное время. Минусы – это дополнительная память, много лишних операций, когда строки отличаются только в конце.

Значителен и иной факт: поразрядная сортировка, опирающаяся на сортировку подсчетом, обладает внушительной константой, запрятанной в O-символике, поскольку при сортировке подсчетом принято использовать дополнительный массив, сравнимый с исходным по занимаемой памяти. Можно использовать более экономные таблицы адресов или индексов, но даже 2 огромных массива целых чисел или указателей будут поддерживать константу на высоком уровне.

Хочется отметить, что сложность программирования подобной задачи – низкая, и сам процесс написания кода весьма механистичен. Это и логично: был использован один из самых ранних методов сортировки, оптимизированный для работы на довольно примитивных устройствах, использующих механистические принципы.