Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Дискретный анализ»

Студент: Д. С. Пивницкий
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-19

Дата: 01.10.2020
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №1

Задача: Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до $2^{64}-1$. Разным словам может быть поставлен в соответствие один и тот же номер.

Программа должна обрабатывать строки входного файла до его окончания.

+ **word 34** добавить слово «word» с номером 34 в словарь. Программа должна вывести строку «OK», если операция прошла успешно, «Exist», если слово уже находится в словаре.

- word удалить слово «word» из словаря. Программа должна вывести «OK», если слово существовало и было удалено, «NoSuchWord», если слово в словаре не было найдено.

word найти в словаре слово «word». Программа должна вывести «OK: 34», если слово было найдено; число, которое следует за «OK:» — номер, присвоенный слову при добавлении. В случае, если слово в словаре не было обнаружено, нужно вывести строку «NoSuchWord».

! Save /path/to/file сохранить словарь в бинарном компактном представлении на диск в файл, указанный парамером команды. В случае успеха, программа должна вывести «OK», в случае неудачи выполнения операции, программа должна вывести описание ошибки.

! Load /path/to/file загрузить словарь из файла. Предполагается, что файл был ранее подготовлен при помощи команды Save. В случае успеха, программа должна вывести строку «OK», а загруженный словарь должен заменить текущий (с которым происходит работа); в случае неуспеха, должна быть выведена диагностика, а рабочий словарь должен остаться без изменений. Кроме системных ошибок, программа должна корректно обрабатывать случаи несовпадения формата указанного файла и представления данных словаря во внешнем файле.

1 Описание

Процесс разработки можно разделить на два этапа: создание и отладка структуры красно-черное дерево и реализация парсера команд (словаря), работающего с этим деревом.

Дерево должно хранить в своих вершинах слова (в формате строки) и соответствующие им числа. Так как слова в словаре должны быть отсортированы в алфавитном порядке, ключом будет считаться именно слово, а соответствующее ему число – значением.

Красно-черное дерево – самобалансирующееся дерево, узлы которого имеют атрибут цвета. При этом, дерево должно удовлетворять условиям: 1. Узел может быть либо красным, либо чёрным и имеет двух потомков; 2. Корень – чёрный. 3. Все листья — чёрные. 4. Оба потомка каждого красного узла — чёрные. 5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов. Для соблюдения этих условий после каждой операции вставки/удаления узла должна проводиться перебалансировка. Время, затрачиваемое на доступ к элементам, составляет $O(log(n))$, где n – число элементов в нем.

Перебаланисировка производится с помощью операций поворота и перекрашивания узлов, которые применяются в зависимости от текущей расцветки и положений узлов дерева. В ходе перебалансировки количество черных узлов на пути из элемента, для которого операция перебалансировки была вызвана до всех его листовых потомков уравниваются, однако так как при этом их суммарное число могло измениться, в некоторых случаях операция рекурсивно вызывается для его потомков, пока не отбалансируется все дерево начиная с корня.

В ходе разработки структуры «красно-черное дерево» также необходимо реализовать операции сохранения его в файл и загрузки из файла. Элементы дерева записываются в файл по очереди их прохождения в процессе обхода в глубину, при этом, вначале записывается текущий элемент, затем – оба его потомка, начиная с левого. Таким образом, во время чтения дерева из заранее сохраненного файла при обнаружении очередного элемента мы можем быть уверены, что следующий за ним элемент является его левым потомком. Если был найден лист (элемент в котором лежит строка длины ноль), то мы понимаем, что текущая ветвь закончилась, и далее записан правый потомок родителя считанного ранее узла.

Для того чтобы не тратить время на попытки расшифровки файлов, которые были созданы не программой словаря, в начало файла добавляется маркер, состоящий из трех символов. При чтении дерева файла структура строится отдельно от текущего дерева, и только если чтение прошло без ошибок, текущее дерево заменяется считанным

В самом словаре производится непрерывная обработка входных строк, в соответствии с результатом этой обработки для каждой введенной команды вызывается

метод заранее созданного дерева. Методы в большинстве своем возвращают код завершения, в соответствии с которым осуществляется вывод на экран информации о результате выполнения команды.

2 Исходный код

Вначале создаем класс узла дерева. В нем хранится ключ (указатель на строку), значение (переменная типа unsigned long long) и цвет (переменная типа char). Также в узле хранятся указатели на родителя и обоих детей этого узла.

Для компактности отчета здесь и далее я буду приводить методы класса только в таблице

```
const char BLACK = 1;
const char RED = 2;

class TRBNode {
public:
char* Key;
unsigned long long Value;
char Colour = BLACK;

TRBNode* Parent = nullptr;
TRBNode* Left = nullptr;
TRBNode* Right = nullptr;
```

Также реализуем вспомогательную функцию strequal, работающую аналогично strcmp, однако не реагирующую на различный регистр букв сравниваемых строк.

Реализуем класс дерева.

```
class TRBTree {
private:
  TRBNode* TreeRoot = nullptr;
}
```

В функции main будем производить обработку команд, предварительно создав дерево, в которое будем записывать слова.

```
int main() {
TRBTree maintree;

char *inpstr = (char*)calloc(260,sizeof(char));
char *word = (char*)calloc(260,sizeof(char));
unsigned long long inpval;

if(inpstr == NULL || inpstr == nullptr || word == NULL || word == nullptr) {
printf("ERROR: allocation error\n");
return -1;
}

while (scanf("%s", inpstr) != EOF) {
if(strlen(inpstr)>256) {
printf("ERROR: uncorrect input\n");
continue;
```

```
}
if (inpstr[0] == '+') {
if(scanf("%s %llu", word, &inpval) == EOF) {
break;
}
if(strlen(word)>256) {
printf("ERROR: uncorrect input\n");
continue;
}
int mrk = maintree.Add(word, inpval);
if (mrk == 0) {
printf("OK\n");
}
else {
if (mrk == -7) {
printf("Exist\n");
}
else
if (mrk == -3) {
printf("ERROR: empty input\n");
}
else
if (mrk == -4) {
printf("ERROR: untraced allocation error\n");
}
else
if (mrk == -1) {
printf("ERROR: out of memory\n");
}
else {
printf("ERROR: unknown error\n");
}
}
}
else
if (inpstr[0] == '-') {
if(scanf("%s", word) == EOF) {
break;
}
if(strlen(word)>256) {
printf("ERROR: uncorrect input\n");
continue;
}
int mrk = maintree.Remove(word);
if (mrk == 0) {
printf("OK\n");
}
else
if (mrk == -8) {
```

```
66 printf("NoSuchWord\n");
67 }
68 else
69 if (mrk == -1) {
70 printf("ERROR: out of memory\n");
71 }
72 else {
73 printf("ERROR: unknown error\n");
74 }
75 }
76 else
77 if (inpstr[0] == '!') {
78 std::string path;
79 if(scanf("%s", word) == EOF) {
80 break;
81 }
82 if (strcmp(word, "Save") == 0) {
83 std::cin >> path;
84
85 int mrk = maintree.SaveToDisk(path);
86 if (mrk == 0) {
87 printf("OK\n");
88 continue;
89 }
90 else
91 if (mrk == 1) {
92 printf("OK\n");
93 continue;
94 }
95 else
96 if (mrk == -1) {
97 printf("ERROR: unable to open file\n");
98 continue;
99 }
100 else
101 if (mrk == -2) {
102 printf("ERROR: unable to write file\n");
103 continue;
104 }
105 else
106 if (mrk == -3) {
107 printf("ERROR: file acsess error\n");
108 continue;
109 }
110 else
111 {
112 printf("ERROR: something gone wrong\n");
113 continue;
114 }
```

```
}
else
if (strcmp(word, "Load") == 0) {
std::cin >> path;

int mrk = maintree.LoadFromDisk(path);
if (mrk == 0) {
printf("OK\n");
continue;
}
else
if (mrk == -1) {
printf("ERROR: file is damaged\n");
continue;
}
else
if (mrk == -2) {
printf("ERROR: wrong format of file\n");
continue;
}
else
if (mrk == -3) {
printf("ERROR: file acsess error\n");
continue;
}
else
{
printf("ERROR: something gone wrong\n");
continue;
}
}

}
else {
TRBNode* res = maintree.Find(inpstr);
if (res != nullptr) {
printf("OK: %llu\n",res->Value);
}
else {
printf("NoSuchWord\n");
}
}
}

maintree.Destroy();
free(inpstr);
free(word);

return 0;
```

```
164 || }
```

3 Консоль

```
(py37) ~ /DA/lab2$ g++ -g -Wall -o lab2ex Prog/DA_lab_2_nostl.cpp
(py37) ~ /DA/lab2$ cat tests/test1
+ a 1
+ A 2
+ aa 18446744073709551615
aa
A
-A
a
(py37) ~ /DA/lab2$ ./lab2ex <tests/test1
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

```
(py37) ~ /DA/lab2$ cat tests/test2
! Save empty.b
! Load asdfghj
+ a 11
a
b
-a
! Save file
a
+ b 7
b
! Load file
a
b
(py37) ~ /DA/lab2$ ./lab2ex <tests/test2
OK
ERROR: file acsess error
OK
OK: 11
NoSuchWord
OK
OK
NoSuchWord
OK
OK: 7
OK
NoSuchWord
NoSuchWord
```

4 Тест производительности

Проводим тест производительности. Для сравнения я сделал программу, использующую std::map, операции доступа к элементам которого выполняются за $O((log(n))$, где n – количество элементов в контейнере. На вход поступает файл с 10^6 строк.

Так как отдельные операции доступа осуществляются настолько быстро, что отследить их невозможно, будем сравнивать время обработки целого файла, в котором находится 10^6 случайно сгенерированных строк, половина которых содержат в себе команды добавления, а вторая половина поделена между командами удаления и отыскания.

```
(py37) ~ /DA/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1177ms
(py37) ~ /DA/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1178ms
(py37) ~ /DA/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1206ms
(py37) ~ /DA/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1183ms
(py37) ~ /DA/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1200ms
(py37) ~ /DA/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1181ms
```

Видим, что в среднем обработка файла занимает 1200ms. Теперь посчитаем среднее время работы программы с моей реализацией красно-черного ддерева.

```
(py37) ~ /DA/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 736ms
(py37) ~ /DA/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 739ms
(py37) ~ /DA/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
```

```
inp complete| 734ms
(py37) ~ /DA/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 735ms
(py37) ~ /DA/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 735ms
(py37) ~ /DA/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 739ms
```

Наблюдаем среднее время работы 735ms. Рискну предположить, что меньшая эффективность std::map как-то связана с ощутимым упором на универсальность при создании этой структуры.

5 Выводы

Выполнив эту лабораторную работу я поближе познакомился с различными сбалансированными деревьями. Работа с красно-черным деревом позволила мне хоть немного погрузиться в волшебный процесс балансировки, а также испытать неподдельное восхищение при виде самостоятельно отсортировавшегося по алфавитному порядку словаря. Однозначно полезный и весьма интересный опыт продолжает поступать ко мне от изощренных тестов чекера. Поиск ошибок без возможности наблюдать хотя бы вывод программы хоть и приносит уйму негативных эмоций, однако позволяет лучше понять, где и какого рода ошибки чаще всего допускаются, а каких ошибок искать не надо, дабы не уйти по ложному пути в бесконечный зацикленный поиск.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Поразрядная сортировка — Вики университета ITMO.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Цифровая_сортировка` (дата обращения: 01.10.2020).