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: М. С. Гаврилов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №2

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 264 - 1. Разным словам может быть поставлен в соответствие один и тот же номер.

Программа должна обрабатывать строки входного файла до его окончания. Каждая строка может иметь следующий формат:

+ 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 параметром команды. В случае успеха, программа должна вывести «OK», в случае неудачи выполнения операции, программа должна вывести описание ошибки (см. ниже).

! 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.

Для всех операций, в случае возникновения системной ошибки (нехватка памяти, отсутствие прав записи и т.п.), программа должна вывести строку, начинающуюся с «ERROR:» и описывающую на английском языке возникшую ошибку.
Используемая структура данных: Красно-чёрное дерево.

1 Описание

Требуется реализовать программу-словарь, работающую на основе красно-черного дерева. 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(lon(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

```
1
2  const char BLACK = 1;
3  const char RED = 2;
4
5  class TRBNode { //
6  public:
7      char* Key;
8      unsigned long long Value;
9      char Colour = BLACK;
10
11     TRBNode* Parent = nullptr;
12     TRBNode* Left = nullptr;
13     TRBNode* Right = nullptr;
14 };
```

Методы, реализованные для класса узла:

TRBNode	
TRBNode(){}	стандартный конструктор. Выделяет память для строки
TRBNode(char* inpkey,unsigned long long inpval) {}	Конструктор, заполняющий узел входными данными
~ TRBNode(){}	Деструктор. Освобождает память от строки
TRBNode* Sibling(){}	Метод, отыскивающий второго ребенка родителя этого узла
bool Isleft(){}	Метод, проверяющий, не является ли этот узел левым ребенком своего родителя
bool Islist(){}	Метод, проверяющий, не является ли этот узел листом (конечным узлом черного цвета, не несущем в себе никаких данных)

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

Для вставки/удаления узлов, а также последующей перебалансировки нам потребуются следующие функции:

Фкнкции балансировки	
void Swap(TRBNode* lhs, TRBNode* rhs){}	Функция обмена данными (ключом и значением) между узлами
TRBNode* LeftTurn(TRBNode* centre) {}	Функция левого поворота вокруг указанного узла
TRBNode* RightTurn(TRBNode* centre){}	Функция правого поворота вокруг указанного узла
TRBNode* ARMreballance(TRBNode* start_element, TRBNode* root){}	Рекурсивная функция, осуществляющая перебалансировку дерева после удаления узла (корень передается на случай его смены в ходе балансировки)
TRBNode* AISreballance(TRBNode* start_element,TRBNode* root){}	Рекурсивная функция, осуществляющая перебалансировку дерева после вставки узла

Реализуем класс дерева.

```
1
2 class TRBTree {
3 private:
4     TRBNode* TreeRoot = nullptr;
5 }
```

В дереве хранится только корень, работа с деервом осуществляется через его методы:

TRBTree	
∼TRBTree (){}	Деструктор, вызывающий рекурсивную функцию уничтожения дерева
int Add(char* key,unsigned long long	value) {}
Метод, добавляющий элемент в дерево. В случае успеха возвращает 0, иначе – код ошибки	
int Remove(char* inpkey){}	Метод, удаляющий элемент из дерева. В случае успеха возвращает 0, иначе – код ошибки

TRBNode* Find(char* inpkey){}	Метод, осуществляющий отыскание узла вдереве. В случае успеха возвращает указатель на него, иначе – нулевой указатель
int SaveToDisk(const std::string& path){}	Метод, осуществляющий сохранения дерева в файл. Принимает строку, содержащую путь к файлу. В случае успеха возвращает 0, иначе – код ошибки
int LoadFromDisk(const std::string& path){}	Метод, осуществляющий считывание дерева из файла. Принимает строку, содержащую путь к файлу. В случае успеха возвращает 0, иначе – код ошибки
void Destroy(){}	Действует аналогично ∼TRBTree ()

Вспомогательные функции для int SaveToDisk(const std::string path) {}

SaveToDisk	
int SaveNode(std::ofstream& fout, TRBNode* ldb) {}	Функция, осуществляющая запись узла ldb в поток fout
int SaveTree(std::ofstream& fout, TRBNode* root){}	Рекурсивная функция, осуществляющая запись узлов дерева в необходимом для последующего прочтения порядке

Вспомогательные функции для int LoadFromDisk(const std::string path) {}

SaveToDisk	
TRBNode* LoadNode(std::ifstream& fin,TRBNode* pst) {}	Функция, осуществляющая чтение узла из поток a fin. Если узел прочитан, то создается труктура узла, указатель на которую и возвращается. Иначе выбрасывается исключение.
TRBNode* LoadTree(std::ifstream& fin, TRBNode* root){}	Рекурсивная функция, осуществляющая чтение узлов дерева из потока, при регистрации исключения от функции чтения узла сообщение об ошибке проходит до точки первоначального вызова

Также для класса дерева реализован ряд дополнительных функций, ненужных при непосредственной работе программы, однако крайне необходимых при отладке:

SaveToDisk	
void Print() {}	Метод, осуществляющий вызов рекурсивной функции вывода всего дерева на экран
int ChekBlackHeighth(){}	Метод, осуществляющий комплексную проверку целостности структуры дерева, в т.ч. постоянство черной высоты и отсутствие некорректных указателей на предков/потомков. Проверка осуществляется вызовом вспомогательной рекурсивной функции

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

```
1  int main() {
2      TRBTree maintree;
3
4      char *inpstr = (char*)calloc(260,sizeof(char));
5      char *word = (char*)calloc(260,sizeof(char));
6      unsigned long long inpval;
7
8      if(inpstr == NULL || inpstr == nullptr || word == NULL || word == nullptr) {
9          printf("ERROR: allocation error\n");
10         return -1;
11     }
12
13     while (scanf("%s", inpstr) != EOF) {
14         //maintree.Print();
15         // ++counter;
16         // if (counter > 9) {
17         // std::cout<<"break here\n";
18         // }
19         if(strlen(inpstr)>256) {
20             printf("ERROR: uncorrect input\n");
21             continue;
22         }
23         if (inpstr[0] == '+') {
24             if(scanf("%s %llu", word, &inpval) == EOF) {
25                 break;
26             }
27             if(strlen(word)>256) {
28                 printf("ERROR: uncorrect input\n");
29                 continue;
30             }
31             int mrk = maintree.Add(word, inpval);
```

```
32              if (mrk == 0) {
33                  printf("OK\n");
34              }
35              else {
36                  if (mrk == -7) {
37                      printf("Exist\n");
38                  }
39                  else
40                  if (mrk == -3) {
41                      printf("ERROR: empty input\n");
42                  }
43                  else
44                  if (mrk == -4) {
45                      printf("ERROR: untraced allocation error\n");
46                  }
47                  else
48                  if (mrk == -1) {
49                      printf("ERROR: out of memory\n");
50                  }
51                  else {
52                      printf("ERROR: unknown error\n");
53                  }
54              }
55          }
56          else
57          if (inpstr[0] == '-') {
58              if(scanf("%s", word) == EOF) {
59                  break;
60              }
61              if(strlen(word)>256) {
62                  printf("ERROR: uncorrect input\n");
63                  continue;
64              }
65              int mrk = maintree.Remove(word);
66              if (mrk == 0) {
67                  printf("OK\n");
68              }
69              else
70              if (mrk == -8) {
71                  printf("NoSuchWord\n");
72              }
73              else
74              if (mrk == -1) {
75                  printf("ERROR: out of memory\n");
76              }
77              else {
78                  printf("ERROR: unknown error\n");
79              }
80          }
```

```
81          else
82          if (inpstr[0] == '!') {
83              std::string path;
84              if(scanf("%s", word) == EOF) {
85                  break;
86              }
87              if (strcmp(word, "Save") == 0) {
88                  std::cin >> path;
89
90                  int mrk = maintree.SaveToDisk(path);
91                  if (mrk == 0) {
92                      printf("OK\n");
93                      continue;
94                  }
95                  else
96                  if (mrk == 1) {//
97                      printf("OK\n");
98                      continue;
99                  }
100                 else
101                 if (mrk == -1) {
102                     printf("ERROR: unable to open file\n");
103                     continue;
104                 }
105                 else
106                 if (mrk == -2) {
107                     printf("ERROR: unable to write file\n");
108                     continue;
109                 }
110                 else
111                 if (mrk == -3) {
112                     printf("ERROR: file acsess error\n");
113                     continue;
114                 }
115                 else
116                 {
117                     printf("ERROR: something gone wrong\n");
118                     continue;
119                 }
120             }
121             else
122             if (strcmp(word, "Load") == 0) {
123                 std::cin >> path;
124
125                 int mrk = maintree.LoadFromDisk(path);
126                 if (mrk == 0) {
127                     printf("OK\n");
128                     continue;
129                 }
```

```
130                 else
131                 if (mrk == -1) {
132                     printf("ERROR: file is damaged\n");
133                     continue;
134                 }
135                 else
136                 if (mrk == -2) {
137                     printf("ERROR: wrong format of file\n");
138                     continue;
139                 }
140                 else
141                 if (mrk == -3) {
142                     printf("ERROR: file acsess error\n");
143                     continue;
144                 }
145                 else
146                 {
147                     printf("ERROR: something gone wrong\n");
148                     continue;
149                 }
150             }
151 
152         }
153         else {
154             TRBNode* res = maintree.Find(inpstr);
155             if (res != nullptr) {
156                 printf("OK: %llu\n",res->Value);
157             }
158             else {
159                 printf("NoSuchWord\n");
160             }
161         }
162 
163     }
164     maintree.Destroy();
165     free(inpstr);
166     free(word);
167 
168     return 0;
169 }
```

3 Консоль

```
max@max-Swift:~/Рабочий стол/ДА/lab2$ g++ -g -Wall -o lab2ex Prog/DA_lab_2_nostl.cpp
max@max-Swift:~/Рабочий стол/ДА/lab2$ cat tests/test1
+ a 1
+ A 2
+ aa 18446744073709551615
aa
A
-A
a
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2ex <tests/test1
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

```
max@max-Swift:~/Рабочий стол/ДА/lab2$ cat tests/test2
! Save empty.b
! Load asdfghj
+ a 11
a
b
-a
! Save file
a
+ b 7
b
! Load file
a
b
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2ex <tests/test2
OK
ERROR: file acsess error
OK
OK: 11
NoSuchWord
OK
OK
NoSuchWord
OK
OK: 7
OK
NoSuchWord
NoSuchWord
max@max-Swift:~/Рабочий стол/ДА/lab2$
```

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 $1 * 10^6$ строк.

Так как отдельные операции доступа осуществляются настолько быстро, что отследить их невозможно, будем сравнивать время обработки целого файла, в котором находится $1 * 10^6$ случайно сгенерированных строк, половина которых содержат в себе команды добавления, а вторая половина поделена между командами удаления и отыскания.

Посчитаем среднее время работы программы, основанной на std::map

```
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1177ms
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1178ms
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1206ms
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1183ms
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1200ms
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2_bm <banchmark/test_of_absolutia
1>/dev/null
inp complete| 1181ms
max@max-Swift:~/Рабочий стол/ДА/lab2$
```

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

```
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 736ms
```

```
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 739ms
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 734ms
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 735ms
max@max-Swift:~/Рабочий стол/ДА/lab2$ ./lab2ex <banchmark/test_of_absolutia
1>/dev/null
inp complete| 735ms
max@max-Swift:~/Рабочий стол/ДА/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] *std::map — cppreference.com.*
URL: `https://en.cppreference.com/w/cpp/algorithm/stable_sort` (дата обращения: 22.11.2020)

[2] *Красно-чёрное дерево — Викиконспекты.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево` (дата обращения: 09.11.2020)

[3] *Красно-чёрное дерево — Википедия.*
URL: `https://ru.wikipedia.org/wiki/Красно-чёрное_дерево` (дата обращения: 09.11.2020).