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: Р. Р. Гаптулхаков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

Дата: 03.12.20
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 парамером 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, отсутсвие прав записи и т.п.), программа должна вывести строку, начинающуюся с «ERROR:» и описывающую на английском языке возникшую ошибку.

Вариант структуры данных: АВЛ-дерево.

1 Описание

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1. [3]

АВЛ-дерево с n ключами имеет высоту h=O(logN). [4]

Балансировка. Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев */h(L)h(R)/=2*, изменяет связи предок-потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева */h(L)h(R)/1*, иначе ничего не меняет. Для балансировки будем хранить для каждой вершины разницу между высотой её левого и правого поддерева *diff[i]=h(L)h(R)*.

Для балансировки вершины используются один из 4 типов вращений: малое левое и правое, большое левое и правое. Малое левое используется тогда, когда баланс вершины равен -2, а баланс правого поддерева <= 0 Большое левое используется тогда, когда баланс вершины равен -2, баланс правого пооддерева равен 1, при этом все потомки этого поддерева должны быть сбалансированы. Симметрично описаны усовия для правых вращений. В литературе иногда не пишут «большое» вращение, так как оно состоит из 2-ух малых.

Вставка элемента. Пусть нам надо добавить ключ k. Будем спускаться по дереву, как при поиске ключа k. Если мы стоим в вершине n и нам надо идти в поддерево, которого нет, то делаем ключ k листом, а вершину его корнем. Дальше поднимаемся вверх по пути поиска и пересчитываем баланс у вершин. Если мы поднялись в вершину p из левого поддерева, то $balance[p]$ увеличивается на единицу, если из правого, то уменьшается на единицу. Если пришли в вершину и её баланс стал равным нулю, то это значит высота поддерева не изменилась и подъём останавливается. Если пришли в вершину и её баланс стал равным 1 или -1, то это значит высота поддерева изменилась и подъём продолжается. Если пришли в вершину и её баланс стал равным 2 или -2, то делаем одно из четырёх вращений и, если после вращения баланс стал равным нулю, то останавливаемся, иначе продолжаем подъём.

Так как в процессе добавления вершины мы рассматриваем не более, чем *O(h)* вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет *O(logn)* операций.

Удаление вершины. Для простоты опишем рекурсивный алгоритм удаления. Если вершина — лист, то удалим её, иначе найдём самую близкую по значению вершину n, переместим её на место удаляемой вершины и удалим вершину n. От удалённой вершины будем подниматься вверх к корню и пересчитывать баланс у вершин. Если мы поднялись в вершину с из левого поддерева, то $balance[p]$ уменьшается на единицу, если из правого, то увеличивается на единицу. Если пришли в вершину и её баланс стал равным 1 или -1, то это значит, что высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины стал равным нулю, то высота

поддерева уменьшилась и подъём нужно продолжить. Если баланс стал равным 2 или -2, следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.

В результате указанных действий на удаление вершины и балансировку суммарно тратится, как и ранее, *O(h)* операций. Таким образом, требуемое количество действий — *O(logn)*.

Структура узла содержит:

- Ключ;
- Значение;
- Высоту узла;
- Указатель на левого сына;
- Указатель на правого сына;

Программма считывает данные из стандартного потока ввода. С помощью блоков «if-else» программа определяет заданное действие.

2 Исходный код

Проект состоит из 4 файлов: main.cpp, bst.hpp, bst.cpp, makefile.

- **main.cpp**:основной файл, в котором происходит взаимодействие программы с пользователем, функции для работы с файлами;

- **bst.hpp**: Объявление структур, классов, их методов;

- **bst.cpp**: реализация методов класссов, структур;

Таблица методов и функций

main.cpp	
Функция	Описание
int main()	Главная функция, в которой происходит чтение данных и создание словаря.
int Sizekey()	Считает длину ключа в узле.
bool Save()	Сохраняет дерево в файл.
bst::TreeNode* Load()	Загружает дерево из файла.
void DeleteTree()	Удаляет дерево.
bst.hpp	
Тип данных	Описание
class Tree	Класс AVL-дерева.
struct TreeNode	Структура узла AVL-дерева.
TreeNode(char* key, uint_64 value, int height)	Конструктор копирования.
TreeNode()	Деструктор.
bst.cpp	
Функция	Описание
void Insert(char* key, uint64$_t value, TreeNode * node$)	Вставка узла в дерево.
TreeNode* Search(char* key, TreeNode* node)	Поиск узла в дереве.
TreeNode* SearchForDelete(char* key, TreeNode *node)	Поиск узла в дереве для удаления.
void Delete(TreeNode* node)	Удаление узла из дерева.
TreeNode* Balance(TreeNode* node)	Вычисление высот после вставки или удаления.
TreeNode* remove_r(TreeNode* node)	минимальный элемент правого поддерева.
void RotateRight(TreeNode* node)	Правый поворот

void RotateLeft(TreeNode* node)	Левый поворот.
void Rebalance(TreeNode* node)	Перебалансировка дерева.
void UpdateBalance(TreeNode* node)	Обновляет балансы для вставки.
void UpdateBalanceDelete(TreeNode* node)	Обновляет балансы для удалеиня.
void print_Tree(TreeNode* p,int level)	Вывод деерва.
bool Copykey(TreeNode* fst, TreeNode* scd)	Копирование ключа узла.

3 Тест производительности

Генератор тестов создёт файл входных данных для программы. В нём реализованы операции вставки, удаления, поиска. Бенчмарк считает время работы АВЛ-дерева и структуры *std::map*, реализованной на красно-чёрном дереве. Отличие красно-чёрного дерева от АВЛ состоит в том, что балансировка в красно-чёрном дерева выполняется за *O (1)*, тогда как в АВЛ за *O (log n)* Это даёт значительный прирост производительности. Для замерки времени использовалась бибилиотека **chrono**.
Тесты создавались с помощью программы на языке Python:

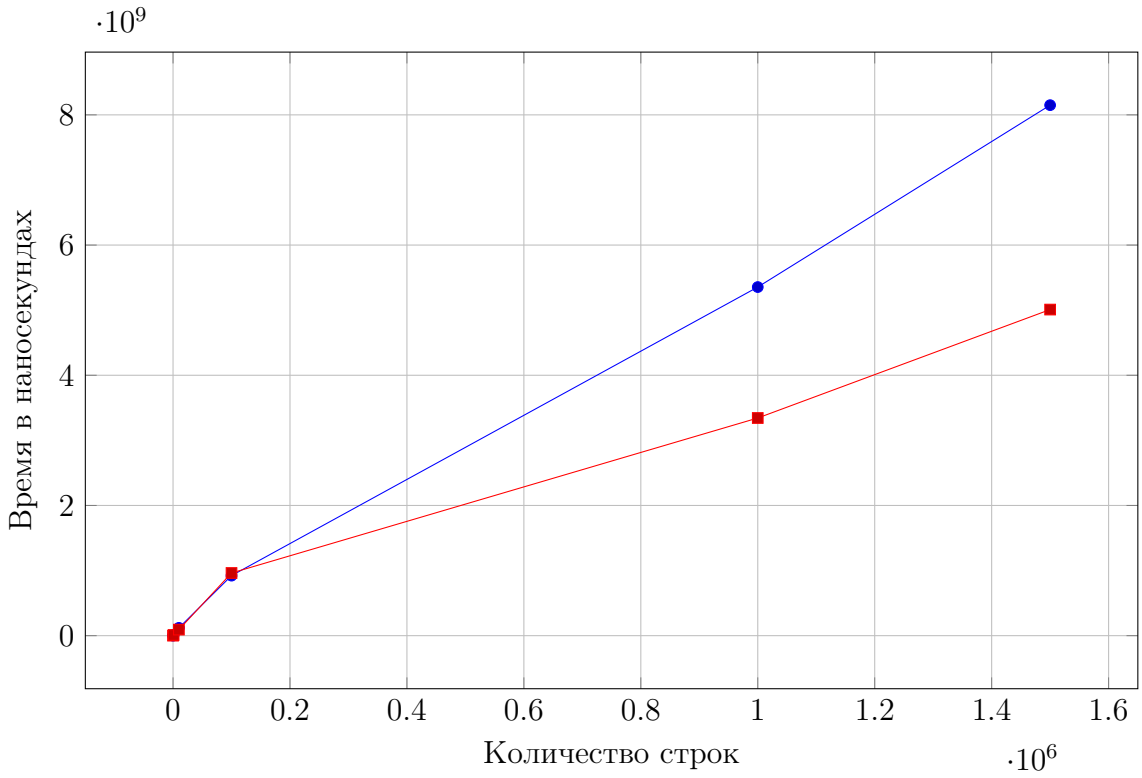
```
1  import sys
2  import random
3  import string
4
5  #! /usr/bin/env python3
6
7  # -*- coding: utf-8 -*-
8
9  import sys
10 import random
11 import string
12
13 def get_random_key():
14     return random.choice( string.ascii_letters )
15
16 if __name__ == "__main__":
17     # ,   1 .
18     if len(sys.argv) != 2:
19         print( "Usage: {0} <count of tests>".format( sys.argv[0] ) )
20         sys.exit(1)
21
22     count_of_tests = int( sys.argv[1] )
23
24     actions = [ "+", "?", "-", "!" ]
25
26     for enum in range( count_of_tests ):
27         keys = dict()
28         test_file_name = "tests/{:02d}".format( enum + 1 )
29         with open( "{0}.t".format( test_file_name ), 'w' ) as output_file, \
30              open( "{0}.a".format( test_file_name ), "w" ) as answer_file:
31
32             #      1  100 .
33             for _ in range( random.randint(8 * 10 ** 5, 10 ** 6) ):
34                 action = random.choice( actions )
35                 if action == "+":
36                     key = get_random_key()
37                     value = random.randint(1, 100)
38                     output_file.write("+ {0} {1}\n".format( key, value ))
```

```
39                      key = key.lower()
40                      #         ,
41                      # ,    ,   ---   .
42                      answer = "Exist"
43                      if key not in keys:
44                          answer = "OK"
45                          keys[key] = value
46                      answer_file.write( "{0}\n".format( answer ) )
47
48                  elif action == "?":
49                      search_exist_element = random.choice([True, False])
50                      key = random.choice([key for key in keys.keys() ]) if
                            search_exist_element and len(keys.keys()) > 0 else
                            get_random_key()
51                      output_file.write("{0}\n".format(key))
52                      key = key.lower()
53                      if key in keys:
54                          answer = "OK: {0}".format(keys[key])
55                      else:
56                          answer = "NoSuchWord"
57                      answer_file.write("{0}\n".format(answer))
58                  elif action == "-":
59                      search_exist_element = random.choice([True, False])
60                      key = random.choice([key for key in keys.keys() ]) if
                            search_exist_element and len(keys.keys()) > 0 else
                            get_random_key()
61                      output_file.write("- {0}\n".format(key))
62                      key = key.lower()
63                      if key in keys:
64                          answer = "OK: {0}".format(keys[key])
65                          del keys[key]
66                      else:
67                          answer = "NoSuchWord"
68                      answer_file.write("{0}\n".format(answer))
```

Графики

Синий: **std::map** (красно-чёрное дерево).

Красный: AVL-дерево.


$\cdot 10^9$
8
6
4
2
0
Время в наносекундах
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
Количество строк
$\cdot 10^6$

4 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я научился применять реализовывать АВЛ-дерево с последующим применением его «в бою», то есть применение его на практике. Это помогло мне понять плюсы и минусы данной структуры и случаи, когда его стоит использовать. Стоит выделить функции сериализации нашего дерева, благодоря данной функции, мы может хранить дерево на жёстком жиске в виде бинароного или текстового файла. Это помогает нам хранить большой объём данных. Тестирование программы является не менее сложной задачей. Правильное тестирование помогает на раннем этапе исправлять замеченные ошибки. Так как АВЛ-дерево является сбалансированным, оно исключает случай выраждения дерева в список, сложность поиска для которого $O\ (n)$.

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] Дональд Э. Кнут. *Искусство программирования. Том 3. Сортировка и поиск, 2-е издание.* — Правообладатель «Диалектика-Вильямс», 2018. Перевод с английского: И. В. Красиков, В. Т. Тертышный. — 834 с. (ISBN 978-5-8459-0082-1, 0-201-89685-0)

[3] *AVL-Tree*
URL: `https://en.wikipedia.org/wiki/AVL_tree` (дата обращения: 25.11.2020).

[4] *AVL-Tree / Wikipedia*
URL: `https://clck.ru/SHh5H` (дата обращения: 25.11.2020).

[5] *AVL Tree Implication*
URL: `http://aliev.me/runestone/Trees/AVLTreeImplementation.html` (дата обращения: 15.11.2020).

[6] *Chrono in C++*
URL: `https://www.geeksforgeeks.org/chrono-in-c/` (дата обращения: 03.12.2020).