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: В. С. Епанешников
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-19

Дата:
Оценка:
Подпись:

Москва, 2021

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: AVL-дерево

1 Описание

Требуется написать реализацию AVL-дерева. AVL-дерево - дерево, каждый узел которого соответствует условию: модуль разности высоты левого поддерева и правого поддерева $<= 1$. Чтобы поддерживать данное условие, нужно уметь считать баланс каждого узла, баланс равен разности высоты левого и правого поддерева. При вставке и удалении требуется учитывать условие AVL-дерева и при нарушении делать перебалансировку дерева при помощи поворотов.

2 Исходный код

avl.cpp	
TAvlNode::TAvlNode()	Конструктор по умол.
TAvlNode::TAvlNode(char* k, uint64_t val, int h)	Конструктор с параметрами (ключ, значение, высота)
TAvlTree::TAvlTree()	Конструктор по умолчанию
TAvlTree::∼TAvlTree()	Деструктор (удаление дерева)
int TAvlTree::GetHeight(const TAvlNode* node) const	Получение высоты из узла
int TAvlTree::GetBalance(const TAvlNode* node) const	Вычисление баланса узла
void TAvlTree::RecountHeight(TAvlNode* node)	Пересчитывание высоты после поворотов
TAvlNode* TAvlTree::RightRotate(TAvlNode* node) TAvlNode* TAvlTree::LeftRotate(TAvlNode* node) TAvlNode* TAvlTree::DoubleRightRotate(TAvlNode* node) TAvlNode* TAvlTree::DoubleLeftRotate(TAvlNode* node)	Малые и большие повороты относительно узла
TAvlNode* TAvlTree::Balance(TAvlNode* node)	Балансировка дерева
TAvlNode* TAvlTree::SubInsert(TAvlNode* node, char* key, uint64_t value)	Вставка узла
void TAvlTree::Insert(char* key, uint64_t value)	Обёртка для вставки
TAvlNode* TAvlTree::Find(const char* key) const	Поиск элемента
AvlNode* TAvlTree::RemoveMin(TAvlNode* node, TAvlNode* curr)	Удаление минимума в поддереве
TAvlNode* TAvlTree::SubRemove(TAvlNode* node, const char* key)	Удаление узла
void TAvlTree::Remove(const char* key)	Обёртка для удаления
void TAvlTree::SubDeleteTree(TAvlNode* node)	Рекурсивное удаление дерева
void TAvlTree::DeleteTree()	Обёртка для удалениея дерева
void TAvlTree::SubSave(std::ostream& os, const TAvlNode* node)	Сохранение дерева в бинарный файл
TAvlNode* TAvlTree::SubLoad(std::istream& is)	Загрузка дерева из файла

```cpp
#pragma once
#include <iostream>
#include <stdint.h>
#include <cstring>

#define KEY_SIZE 257

struct TAvlNode {
  char key[KEY_SIZE] = {'\0'};
  uint64_t value;
  int height;
  TAvlNode* left;
  TAvlNode* right;

  TAvlNode();
  TAvlNode(char k[KEY_SIZE], uint64_t val, int h = 1);
  ~TAvlNode();
};

class TAvlTree {
private:
  TAvlNode* root;

  int GetHeight(const TAvlNode* node) const;
  int GetBalance(const TAvlNode* node) const;
  TAvlNode* RightRotate(TAvlNode* node);
  TAvlNode* LeftRotate(TAvlNode* node);
  TAvlNode* DoubleRightRotate(TAvlNode* node);
  TAvlNode* DoubleLeftRotate(TAvlNode* node);
  TAvlNode* Balance(TAvlNode* node);
  void RecountHeight(TAvlNode* node);
  TAvlNode* SubInsert(TAvlNode* node, char key[KEY_SIZE], uint64_t value);
  TAvlNode* RemoveMin(TAvlNode* node, TAvlNode* curr);
  TAvlNode* SubRemove(TAvlNode* node, const char key[KEY_SIZE]);
  void SubDeleteTree(TAvlNode* node);
  void SubSave(std::ostream& os, const TAvlNode* node);
  TAvlNode* SubLoad(std::istream& is);

public:
  TAvlTree();
  ~TAvlTree();

  TAvlNode* Find(const char key[KEY_SIZE]) const;
  void Insert(char key[KEY_SIZE], u_int64_t value);
  void Remove(const char key[KEY_SIZE]);
  void DeleteTree();
  void Save(const char path[KEY_SIZE]);
  void Load(const char path[KEY_SIZE]);
};
```

3 Консоль

```
+ word 34
OK
word
OK: 34
-word
OK
word
NoSuchWord
+ till 35
OK
+ till 67
Exist
! Save tree.bin
OK
till
OK: 35
-till
OK
till
NoSuchWord
! Load tree.bin
OK
till
OK: 35
-till
OK
till
NoSuchWord
```

4 Тест производительности

Тест производительности представляет из себя следующее: создаём std::map и TAvl. Вставляем в оба объекта по 1 млн. элементов с ключом=значению в диапазоне от 0 до 999999. Далее 1 млн. раз ищем элемент с ключом "500000". Последний тест - 1 млн. раз удаляем элемент со значением(от 999999 до 0) из std::map и и avl. Замеряем время каждого процесса.

```
MacBook:solution vladislove$ ./a.out
Insert map time: 0.107548 seconds
Insert avl time: 1.08258 seconds
Find map time: 0.085879 seconds
Find avl time: 0.108625 seconds
Remove map time: 0.083967 seconds
Remove avl time: 0.825851 seconds
```

Как видно, удаление и вставка в std::map работает значительно быстрее, чем в avl дереве. Это, скорее всего, связано с реализацией (при балансировке приходится каждый раз подниматься до корня дерева). Поиск в std::map работает совсем чуть-чуть быстрее.

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я познакомился с различными структурами данных. Научился работать с AVL-деревом. Такие структуры данных хорошо подходят для хранения и обработки большого объёма данных, так как поиск, вставка и удаление делаются за O(log(n)). Также важно знать, как устроены эти структуры, чтобы понимать, как работают некоторые стандартные контейнеры, например, std::map использует внутри RB-дерево.

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] *AVL-дерево Wiki.*

[3] *AVL-дерево Habr.*