Московский авиационный институт (национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа по курсу "Дискретный анализ" №2

Студент: Клименко В. М.

Преподаватель: Макаров Н. К.

Группа: М8О-203Б-22

Дата: _____

Оценка: _____

Подпись: _____

Оглавление

Цель работы

Научиться писать сбалансированные деревья.

Постановка задачи

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.

Общие сведения о программе

Программа состоит из трех функций:

`main` - точка входа программы, в ней происходит считывание команд и их экзекуция.

Для имплементации AVL дерева было создано три структуры данных:

- `key_value` – структура данных, содержащая в себе ключ – строку и значение – 64-х битное число
    - `key_value_new` – создание нового `key_value`
- `node` – структура данных – вершина AVL дерева, содержит в себе ссылку на левого и правого сына, значение и высоту
    - `node*` – функции для работы с вершинами
- `avl` – структура данных, представляющее AVL дерево, хранится только корень дерева – ссылка на node
    - `avl*` – функции для работы с AVL деревом

Также, для сохранения и считывания `avl` было создано две функции – для сохранения в файл и для загрузки из файла в формате:

```
k: <key> v: <value>\n
```

Реализация

`main.c`:

```c
#include <stdio.h>
#include <stdlib.h>

#include <inttypes.h>
#include <string.h>

#include <ctype.h>
#include <stdbool.h>

#include <limits.h>


#ifdef DMALLOC
#include "dmalloc.h"
#endif


#define max(a, b) ((a) > (b) ? (a) : (b))

#ifndef PATH_MAX
#define PATH_MAX 260
#endif

#define KEY_LEN 257
#define VALUE_LEN 20
#define COMMAND_BUFFER_LENGTH 1 + 1 + KEY_LEN + 1 + VALUE_LEN + 1 + 1
#define FILE_LINE_LEN 2 + 1 + KEY_LEN + 1 + 2 + 1 + VALUE_LEN + 1 + 1



void str_to_lower(char *string, size_t length) {
      for (size_t char_index = 0; char_index < length - 1 &&
string[char_index] != '\0'; ++char_index) {
      string[char_index] = tolower(string[char_index]);
      }

      string[length - 1] = '\0';
}



typedef struct {
      char key[KEY_LEN];
```

```
	uint64_t value;
} key_value;


key_value key_value_new(const char key[KEY_LEN], uint64_t value) {
	key_value new;

	strncpy(new.key, key, KEY_LEN);
	new.key[KEY_LEN - 1] = '\0';
	new.value = value;

	return new;
}



typedef struct node {
	struct node *left;
	struct node *right;
	key_value value;
	int64_t height;
} node;


node* node_new(key_value value) {
	node *new = calloc(1, sizeof(node));

	if (new == NULL) return NULL;

	new->value = value;

	return new;
}


void node_free(node *current) {
	if (current == NULL) return;
	node_free(current->left);
	node_free(current->right);
	free(current);
}
```

```
int64_t node_height(node *current) {
	if (current == NULL) return -1;
	return current->height;
}

int64_t node_balance(node *current) {
	if (current == NULL) return 0;
	return node_height(current->left) - node_height(current->right);
}

int64_t node_update_height(node *current) {
	if (current == NULL) return -1;

	int64_t left_height = node_height(current->left);
	int64_t right_height = node_height(current->right);
	current->height = max(left_height, right_height) + 1;

	return current->height;
}



typedef enum {
	avl_result_ok,
	avl_result_error,
	avl_result_exists,
} avl_result;

typedef struct {
	node *root;
} avl;


avl* avl_new() {
	return calloc(1, sizeof(avl));
}


/*

	A		B
   / \		       / \
  B   z     =>   x   A
```

```
 / \              / \
x   y            y   z

*/
node* avl_rotate_right(node *top) {
      node *left_son = top->left;
      node *moved_child = left_son->right;

      // rotate
      left_son->right = top;
      top->left = moved_child;

      // update heights
      node_update_height(top);
      node_update_height(left_son);

      return left_son;
}

/*

  A               B
 / \              / \
x   B       =>   A   z
   / \          / \
  y   z        x   y

*/
node* avl_rotate_left(node *top) {
      node *right_son = top->right;
      node *moved_child = right_son->left;

      // rotate
      right_son->left = top;
      top->right = moved_child;

      // update heights
      node_update_height(top);
      node_update_height(right_son);

      return right_son;
}
```

```
node* avl_rebalance(node *current) {
	int64_t balance = node_balance(current);

	if (balance > 1) {
	if (node_balance(current->left) < 0) current->left =
avl_rotate_left(current->left);
	current = avl_rotate_right(current);
	}

	if (balance < -1) {
	if (node_balance(current->right) > 0) current->right =
avl_rotate_right(current->right);
	current = avl_rotate_left(current);
	}

	return current;
}


node* _avl_insert(node *current, key_value value, avl_result *result) {
	// insert a leaf
	if (current == NULL) {
	node *new_node = node_new(value);

	if (new_node == NULL) *result = avl_result_error;
	else *result = avl_result_ok;

	return new_node;
	}

	// choose a branch
	int cmp = strcmp(value.key, current->value.key);

	if (cmp < 0) {
	current->left = _avl_insert(current->left, value, result);
	} else if (cmp > 0) {
	current->right = _avl_insert(current->right, value, result);
	} else {
	return current;
	}

	// update height
```

```
	node_update_height(current);

	return avl_rebalance(current);
}

avl_result avl_insert_lowercase(avl* tree, const key_value value) {
	avl_result result = avl_result_exists;
	tree->root = _avl_insert(tree->root, value, &result);
	return result;
}

avl_result avl_insert(avl *tree, const key_value value) {
	key_value key_value_lowercase = {
	.value = value.value
	};

	strncpy(key_value_lowercase.key, value.key, KEY_LEN);
	str_to_lower(key_value_lowercase.key, KEY_LEN);

	return avl_insert_lowercase(tree, key_value_lowercase);
}


node* _avl_delete(node *current, const char *key, avl_result *result) {
	if (current == NULL) return NULL;

	// choose a branch
	int cmp = strcmp(key, current->value.key);

	if (cmp < 0) {
	current->left = _avl_delete(current->left, key, result);
	} else if (cmp > 0) {
	current->right = _avl_delete(current->right, key, result);
	} else {
	*result = avl_result_ok;
	// only one child or no children
	if (current->left == NULL) {
		node *right_child = current->right;
		free(current);
		return right_child;
	} else if (current->right == NULL) {
		node *left_child = current->left;
		free(current);
```

```
            return left_child;
        }

        // two children: get minimal element in the right child
        node *minimal = current->right;
        while (minimal->left != NULL) minimal = minimal->left;

        // copy from it and delete
        current->value = minimal->value;

        avl_result _;
        current->right = _avl_delete(current->right, minimal->value.key, &_);
        }

        // update height
        node_update_height(current);

        return avl_rebalance(current);
}

avl_result avl_delete(avl *tree, const char *key) {
        if (tree->root == NULL) return avl_result_error;

        char lowercase_key[KEY_LEN];
        strncpy(lowercase_key, key, KEY_LEN);
        str_to_lower(lowercase_key, KEY_LEN);

        avl_result result = avl_result_error;
        tree->root = _avl_delete(tree->root, lowercase_key, &result);

        return result;
}


uint64_t _avl_find(node *current, const char *key, avl_result *result) {
        if (current == NULL) return 0;

        int cmp = strcmp(current->value.key, key);

        if (cmp < 0) {
        return _avl_find(current->right, key, result);
        } else if (cmp > 0) {
        return _avl_find(current->left, key, result);
```

```
	} else {
	*result = avl_result_ok;
	return current->value.value;
	}
}

uint64_t avl_find(avl tree, const char *key, avl_result *result) {
	char lowercase_key[KEY_LEN];
	strncpy(lowercase_key, key, KEY_LEN);
	str_to_lower(lowercase_key, KEY_LEN);

	*result = avl_result_error;
	uint64_t value = _avl_find(tree.root, lowercase_key, result);
	if (*result == avl_result_ok) return value;

	return 0;
}


void avl_free(avl *tree) {
	node_free(tree->root);
	free(tree);
}


void _avl_inorder_print(node *current) {
	if (current == NULL) return;

	printf("[");
	_avl_inorder_print(current->left);
	printf("] %s:%"PRIu64" {", current->value.key, node_height(current));
	_avl_inorder_print(current->right);
	printf("}");
}

void avl_inorder_print(avl tree) {
	_avl_inorder_print(tree.root);
	printf("\n");
}


void _avl_save_to_path(node *current, FILE *file) {
	if (current == NULL) return;
```

```
	_avl_save_to_path(current->left, file);
	fprintf(file, "k: %s v: %"PRIu64"\n", current->value.key,
current->value.value);
	_avl_save_to_path(current->right, file);
}

int avl_save_to_path(avl tree, const char *path) {
	FILE *file = fopen(path, "w");
	if (file == NULL) {
	return avl_result_error;
	}

	_avl_save_to_path(tree.root, file);

	fclose(file);

	return avl_result_ok;
}

avl* avl_load_from_path(const char *path) {
	FILE *file = fopen(path, "r");
	if (file == NULL) {
	return NULL;
	}

	char key[KEY_LEN];
	char line[FILE_LINE_LEN];
	uint64_t value;

	avl *tree = avl_new();
	while (fgets(line, FILE_LINE_LEN, file) != NULL) {
	if (line[0] != 'k') {
		avl_free(tree);
		fclose(file);
		return NULL;
	}
	sscanf(line, "k: %s v: %"PRIu64, key, &value);
	avl_insert_lowercase(tree, key_value_new(key, value));
	}

	fclose(file);
```

```
	return tree;
}



int main() {
	avl *tree = avl_new();

	char command_buffer[COMMAND_BUFFER_LENGTH];

	while (fgets(command_buffer, COMMAND_BUFFER_LENGTH, stdin) != NULL) {
	if (command_buffer[0] != '+' && command_buffer[0] != '-' &&
command_buffer[0] != '!') {
		memmove(command_buffer + 2, command_buffer,
COMMAND_BUFFER_LENGTH - 2);
		command_buffer[0] = '?';
		command_buffer[1] = ' ';
	}

	if (command_buffer[0] == '!') {
		char file_command[5];
		char path[PATH_MAX];

		sscanf(command_buffer, "! %s %s", file_command, path);

		switch (file_command[0]) {
			case 'S':;
			if (avl_save_to_path(*tree, path) == avl_result_ok)
printf("OK\n");
			else printf("ERROR: Could not write to a file %s\n",
path);
			break;
			case 'L':;
			avl *temp_tree = avl_load_from_path(path);
			if (temp_tree) {
				printf("OK\n");
				avl_free(tree);
				tree = temp_tree;
			} else {
				printf("ERROR: Could not read from file %s\n",
path);
			}
			break;
```

```
            }
        } else {
            char command = 0;
            char key[KEY_LEN];
            uint64_t value = 0;

            sscanf(command_buffer, "%c %s %"PRIu64, &command, key, &value);
            avl_result result = avl_result_error;

            switch (command) {
                case '+':;
                result = avl_insert(tree, key_value_new(key, value));
                switch (result) {
                    case avl_result_error:
                        printf("ERROR: Buy more RAM, lol\n");
                        break;
                    case avl_result_exists:
                        printf("Exist\n");
                        break;
                    case avl_result_ok:
                        printf("OK\n");
                        break;
                }
                break;
                case '-':;
                if (avl_delete(tree, key) == avl_result_ok)
printf("OK\n");
                else printf("NoSuchWord\n");
                break;
                default:;
                uint64_t value = avl_find(*tree, key, &result);
                if (result == avl_result_ok) printf("OK: %"PRIu64"\n",
value);
                else printf("NoSuchWord\n");
            }
        }
        }

        avl_free(tree);

        return 0;
}
```

Пример работы

Input:

+ a 1

+ A 2

+ aa 18446744073709551615

aaa

A

- A

a

Output:

OK

Exist

OK

OK: 18446744073709551615

OK: 1

OK

NoSuchWord

Вывод

В ходе лабораторной работы я научился писать такую структуру данных как сбалансированное дерево для сохранения, поиска и удаления элементов за $O(log_2 n)$.