Sistemes Operatius 2 - Laboratori 2

Autor: Taras Yarema

Data: 23 d'octubre de 2019

1. `fgets` vs `fscanf`

La funció `fgets` es llegeix fins a que troba un canvi de línia `"\n"`, i la guarda també. En canvi, la funció `fscanf` utilitza l'especificador `%s` que llegeix fins a qualsevol espai en blanc, canvi de línia, tabulador, etc. i no el guarda.

Per tant, en general, `fscanf` només és útil quan es llegeix un input del que se sap l'estructura prèviament. Donat que no s'especifica la mida del buffer a `fscanf`, es considera aquesta com una funció menys segura que `fgets`.

2. Refactor `arbre-binari`

2.1 `red-black-tree.h`

2.1.1 `struct node_data_`

Per a poder guardar paraules s'ha canviat la definició:

```
typedef struct node_data_
{
   char *key;
   int num_times;
} node_data;
```

2.1.2 Definició `find_node`

Donat el canvi anterior, s'ha canviat la definició a

```
node_data *find_node(rb_tree *tree, char *key);
```

2.2 `red-black-tree.c`

2.2.1 `free_node_data`

Donat el canvi en `node_data`, s'ha d'alliberar la memòria reservada per a `key`.

```
free(data->key);
```

2.2.2 `int compare_key1_less_than_key2(char *key1, char *key2)`

Es realitza una comparació alfabètica, sense tenir en compte la capitalització. Es pren la llargada de la clau més curta i es compara respecte aquesta.

Retorna `0` si `key1 > key2`, i `1` si `key1 <= key2`.

2.2.3 `int compare_key1_equal_to_key2(char *key1, char *key2)`

Es realitza una comparació alfabètica, sense tenir en compte la capitalització. En aquest cas la llargada conicideix, per tant, es compara respecte la llargada de `key1`.

Retorna `0` si `key1 != key2`, i `1` si son iguals.

3. Refactor `extraccio-paraules`

S'ha canviat la manera de parsejar les paraules de manera que si es troba un apòstrof, en comptes de parar, es comproven els caràcters anterior i següent. Si aquests són alfanumèrics (`isalpha`), la paraula en qüestió (amb l'apòstrof) és vàlida.

4. Features

Es considera (a priori) que l'estructura del directori és la següent:

```
/
├─ main.c
├─ utils.c
├─ process.c
├─ red-black-tree.c
├─ red-black-tree.h
├─ config.h
├─ Makefile
├─ words
└─ base_dades/
      ├─ llista.cfg
      ├─ llista_2.cfg
      ├─ ...
      └─ ...
```

4.1 `main.c`

Un cop fet `make`, l'ús de `./main` és:

```
./main fitxer_llistat diccionari
```

- `fitxer_llistat`: Per defecte és `base_dades/llista_2.cfg`. Es considera que la path dels arxius en el llistat donat estan en el mateix deirectori del fitxer. És a dir, per a `base_dades/llista_2.cfg` al

path és `base_dades/`.
- `diccionari`: Per defecte és `words`. Es pot passar qualsevol fitxer amb un llistat de paraules que es vulgui utilitzar com a diccionari.

L'ordre és important, no es poden passar els arguments al revés.

4.2 `config.h`

És un fitxer que conté definicions globals que s'utilitzen en altres fitxers.

- `MAX_CHARS`: Enter que defineix el màxim de caràcters a llegir.
- `DEBUG`: Printejar informació durant el desenvolupament.
- `DEBUG_TIME`: Printejar temps d'execució de funcions.
- `DEBUG_TREE`: Printejar informació especifica en el arxiu `red-black-tree.c`.
- `PRINT_CSV`: Si és `1`, es printeja en format CSV a `print_tree_positive` i `print_tree_positive_n`. En cas contrari, es realitza *pretty printing*.

4.3 `utils.c`

Aquest fitxer conté tres funcions que ajuden a mantenir el fitxer principal `main.c` més net.

Totes les funcions d'aquest fitxer tenen el mateix tipus de retorn. Retornen `0` si s'han executat correctament. Retornen `1` si hi ha hagut algun error. Concretament es retorna `1` si no s'ha pogut obrir el fitxer `f_name`.

4.3.1 `int init_tree_from_file(rb_tree *tree, const char *f_name)`

Inicialitza l'arbre `tree` amb les paraules del fitxer `f_name`.

4.3.2 `int print_tree_positive(rb_tree *tree, const char *f_name)`

Printeja a `stdout` tots els nodes `node` de `tree` tals que `node->num_times > 0`. Per a fer-ho es van llegint les paraules del diccionari `f_name`.

Depenent de `PRINT_CSV`,printeja en format CSV o en un format més bonic (estil taula SQL).

4.3.3 `int print_tree_positive_n(rb_tree *tree, const char *f_name, int n)`

Printeja a `stdout` tots els `n` nodes de `tree` amb més ocurrències. S'ha implementat utilitzant una mena de *rànking* que es va actualitzant amb cada clau llegida del fitxer `f_name`.

Depenent de `PRINT_CSV`,printeja en format CSV o en un format més bonic (estil taula SQL).

4.4 `process.c`

4.4.1 `int process_file(rb_tree *tree, const char *f_name)`

Donat un fitxer, llegir cada línia d'aquest i processar-la amb la funció `process_line`.

Retorna `0` si s'ha executat correctament. Retorna `1` si hi hagut algun error (en el moment d'obrir el fitxer `f_name`).

4.4.2 `int process_list_files(rb_tree *tree, const char *fl_name, const char *path)`

Donat un fitxer que conté una llista de fitxers, anar llegint cada nóm de fitxer i executan `process_file`.

Retorna `0` si s'ha executat correctament. Retorna `1` si hi hagut algun error (en el moment d'obrir el fitxer `f_name`).

5. Top 10

Com he comentat abans, s'ha implementat la funció nativa `print_tree_positive_n` a `utils.c`. Aquesta retorna les `n` parelles `key`, `num_times` amb major nombre d'aparicions (amb el format definit a `PRINT_CSV`).

Tot i així, si executem el codi utilitzant `print_tree_positive` s'imprimeixen a `stdout` totes les parelles amb `num_times > 0`. Si guardem dita informació a `data_file`, per exemple:

```
./main base_dades/llista.cfg data/words > data_file
```

podem extreure les `n` parelles amb més ocurrències amb l'ajut del següent *oneliner*, per a `n = 10`:

```
sort -t "," -k2 -rn data_file | head -10
```

1. Primer es crida `sort` amb les opciones següents:

- `-t ","`: Separa `data_file` per la coma (el cotingut esta en format CSV).
- `-k2`: Utilitza la segona columna per a ordenar.
- `-rn`: Ordena al revès i numèricament.

2. Finalment, es pipeja el output a la comanda `head -n` que retorna per pantalla les `n` primeres files.