Pràctica 3

SISTEMES OPERATIUS II

DAVID DE LA PAZ / ORIOL RABASSEDA

ÍNDEX

Observacions del codi

En aquest apartat explicarem detalls sobre la implementació del codi no mencionats a la resta d'apartats respecte a algunes decisions de disseny:

- Les funcions *insert_node_tree()* i *insert_destination* són funcions noves d'aquesta pràctica que tenen exactament el mateix codi d'addició d'elements que la pràctica anterior però en una funció apart per tal de poder executar fragments de codi des del menú a l'hora de crear una base de dades des d'un fitxer donat.
- Les funcions de la pràctica anterior es troben al fitxer *data_base.c*, d'aquesta manera se separa el menú de la implementació de la base de dades d'aeroports. Això ha propiciat la creació d'un header.
- Makefiles: en C, es considera bona pràctica tenir un makefile per cada carpeta. A més, prevé d'errors a l'hora d'agafar fitxers de subcarpetes i crear els .o en la carpeta actual. Per tal de poder mantenir la modularitat i tenir cada cosa en carpetes separades s'han creat 4 makefiles: un pel menú, un per les funcions de la base de dades, un per l'arbre i un per la llista enllaçada. Aleshores, en executar la comanda *make* des de la localització del menú, s'especifica que per crear l'executable són necessaris tots els .o i el que es demana per crear-los és l'execució del make de la carpeta inferior. D'aquesta manera hi ha un sistema iteratiu de makes que es criden quan es necessiten. De manera igual, el clean usa els cleans de les carpetes inferiors. Aquest sistema implica que qualsevol modificació en arxius inferiors no serà reconeguda pel make superior en fer make i caldrà executar make clean abans del make.
- Tot i que hi ha bastants mètodes que utilitzen la paraula 'database', en realitat tots treballen amb un simple rb_tree.
- Per tal de fer més fàcil el recompte de nodes a l'arbre, hem afegit un comptador de nodes a l'estructura rb_tree (d'aquesta manera ens estalviem un postorder només per comptar).
- Qualsevol tipus d'error que es produeixi, el missatge s'envia a través del canal d'error. Ha sigut el consens el que ens ha determinat que és un missatge i que és un error.
- Tant les funcions *delay()* com *max_destinations()* de la base de dades imprimeixen els resultats des de data base i no ho retornen (decisió del programador).

Funcionament del programa

Aquesta tercera pràctica parteix de funcions de l'altra pràctica per tal d'operar amb arbres binaris d'aeroports ja vists a l'anterior pràctica. El main utilitza una plantilla de menú que ofereix a l'usuari un menú amb diverses opcions a realitzar. Aquest imprimeix per pantalla les opcions disponibles, llegeix la entrada d'usuari amb un *fgets* i mitjançant un switch s'escull la comanda a realitzar. Les funcions que pot dur a terme l'usuari són les següents: crear un nou arbre, guardar un arbre a memòria, llegir un arbre guardat a disc, consultar informació sobre l'arbre i sortir del programa. Anem a veure ara amb més detall en qué consisteix cadascuna d'aquestes opcions.

Creació de l'arbre: El programa demana a l'usuari que introdueixi la localització d'un fitxer d'aeroports i un de vols. A continuació cridem al mètode ***build_database(str1, str2, tree)*** que rep els 2 char pointers i crea un nou arbre d'aeroports utilitzant les funcions de la pràctica 2. El mètode en qüestió crea l'arbre donada la llista d'aeroports, introdueix els vols i retorna l'arbre construït. Si algun dels dos fitxers no existeix, aleshores no retornarà res. És dins d'aquest métode on comprovem si l'arbre és buit o no. Si tenim ja un arbre carregat, l'esborrem amb el métode ***delete_database(tree)*** (expliquem aquest procés en més detall a *sortida de programa*).

Emmagatzemar arbre: Aquesta opció permet a l'usuari guardar l'arbre actual com a un fitxer de text a on indiqui l'usuari. Primer de tot el programa llegeix amb un *fgets* la localització on l'usuari vol guardar l'arbre. Un cop obtinguda, s'obre un nou fitxer d'escriptura utilitzant la operació ***fopen(str1, "w")***. Ara, el primer pas consisteix a escriure el *magic number* i el nombre de nodes al fitxer fent servir la operació ***fwrite()***. Per guardar els nodes de l'arbre el recorrem fent postorder: per cada nou node, guardem l'aeroport d'origen i el número de vols amb ***fwrite()***, i tot seguit recorrem la seva linked-list escrivint per cada element el nom de l'aeroport de destí, el número de vols i el retard. D'aquesta manera continuem recorrent el arbre fins escriure totes les dades al fitxer.

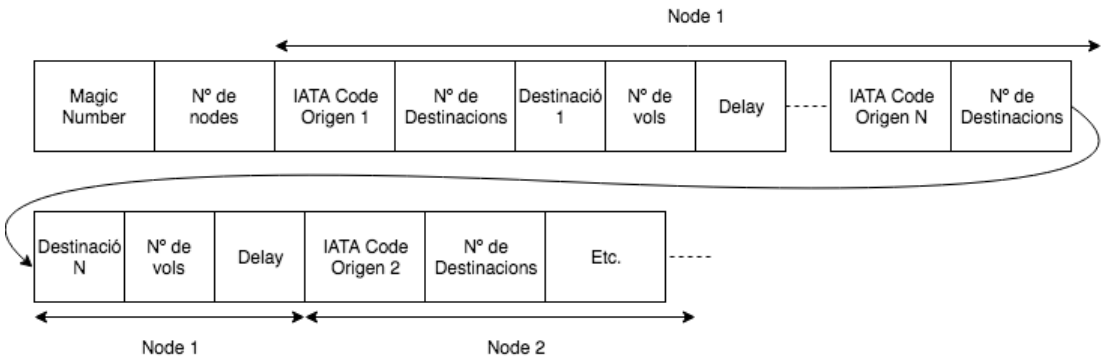

Fig. 1: Format d'emmagatzematge de les dades de l'arbre.

Llegir arbre: Ara l'usuari introdueix la localització d'un fitxer de text que contingui la informació d'un arbre d'aeroports i el programa l'utilitzarà per crear novament un arbre. Primer el programa rep la localització del fitxer amb *fgets* (Si ja tenim un arbre guardat, l'esborrem) i l'obrim amb ***fopen(str1, "r")***. Un cop obert el fitxer, primer de tot hem de comprovar que el *magic number* coincideix. De no ser així, el programa imprimeix un error i acabem. Si el nombre és vàlid, creem un arbre buit amb *create_database()* i llegim el nombre de nodes amb ***fread()***. Un cop sabem el nombre de nodes, podem recórrer amb un *for()* la llista de nodes. Per cada node, llegim el nom de l'aeroport d'origen i l'introduïm a l'arbre buit utilitzant funcions de la pràctica anterior amb *insert_node_tree()*. A continuació llegim el nombre de vols i tornem a iterar, llegint els 3 atributs de cada element de la linked-list i afegint-los a l'arbre amb el mètode *insert_destination()*, el qual crea l'element i l'afegeix a l'arbre. Aquest procés continua fins que hem afegit tots els elements a l'arbre.

Buscar informació: Aquesta comanda permet a l'usuari fer consultes d'informació sobre l'arbre, de manera idèntica a la tasca demanada a la pràctica 2. El programa llegeix l'input de l'usuari amb *fgets()*, el qual té 2 opcions: Si l'input és un salt de línia, la funció cridarà al mètode ***max_destinations()***, que imprimirà per

pantalla el node de l'arbre amb més destinacions. D'altra banda, si l'usuari introdueix el nom d'un aeroport, es cridarà a la funció ***delay()***, que calcularà el retard mig de tots els vols de l'aeroport especificat. Les funcions *max_destinations()* i *delay()* funcionen de manera idèntica a la pràctica anterior, per la qual cosa no tornem a comentar el seu funcionament.

Sortir del programa: Finalment, aquesta opció permet a l'usuari finalitzar l'execució del programa. Si tenim algun arbre inicialitzat, aleshores cridem a ***delete_database(tree)*** per tal d'eliminar l'arbre alliberant la memòria necessària. El mètode en qüestió simplement crida al mètode *delete_tree()* de la pràctica anterior, fa un *free* de l'arbre i el posa a NULL. Un cop alliberat, la opció 5 finalitza el bucle *do{}while()* del main, per la qual cosa el programa acaba.

Valgrind

Utilitzem Valgrind per comprovar que la memòria s'allibera correctament. Veiem que no dóna errors:

```
Menu

 1 - Creacio de l'arbre
 2 - Emmagatzemar arbre a disc
 3 - Llegir arbre de disc
 4 - Consultar informacio de l'arbre
 5 - Sortir

   Escull opcio: 5


==2807==
==2807== HEAP SUMMARY:
==2807==     in use at exit: 0 bytes in 0 blocks
==2807==   total heap usage: 36,547 allocs, 36,547 frees, 217,763 bytes allocated
==2807==
==2807== All heap blocks were freed -- no leaks are possible
==2807==
==2807== For counts of detected and suppressed errors, rerun with: -v
==2807== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

(Fixar-se en els blocs assignats i alliberats per comprovar la correcta execució del programa)