Programación II, 2018-2019

Escuela Politécnica Superior, UAM

Práctica 4: Árboles Binarios de Búsqueda

OBJETIVOS

- Familiarización e implementación en C del tipo abstracto de datos que representa un Árbol Binario de Búsqueda.
- Aplicar a un caso real los árboles binarios de búsqueda. La aplicación que se plantea pretende enfrentar al estudiante a dos aspectos relevantes: la creación del TAD propiamente dicha y la manipulación de un volumen de información importante.

NORMAS

Igual que en prácticas anteriores, los ***programas*** que se entreguen deben:

- Estar escritos en ***ANSI C,*** siguiendo las **normas de programación** establecidas.
- Compilar sin errores ni *warnings*, estableciendo las banderas "-ansi" y "-pedantic" al compilar.
- **Ejecutarse sin problema en una consola de comandos**.
- Incorporar un adecuado **control de errores**; es justificable que un programa no admita valores inadecuados, pero no que se comporte de forma anómala con dichos valores.
- La ***memoria*** que se entregue debe elaborarse sobre el modelo propuesto y entregado con la práctica.

PLAN DE TRABAJO

Semana 1: código de P4_E1 y P4_E2. Cada profesor indicará en clase si se ha de realizar alguna entrega y cómo: papel, e-mail, Moodle, etc.

Semana 2: código de P4_E3 y P4_E4, que son todos los ejercicios.

La entrega final se realizará a través de Moodle, **siguiendo escrupulosamente las instrucciones** indicadas en el enunciado de la práctica 0 referentes a la organización y nomenclatura de ficheros y proyectos. Se recuerda que el fichero comprimido que se debe subir debe llamarse **Px_Prog2_Gy_Pz**, siendo '**x**' el número de la práctica, '**y**' el grupo de prácticas y '**z**' el número de pareja (ejemplo de entrega de la pareja 5 del grupo 2161: P4_Prog2_G2161_P05.zip).

Las **fechas de subida a Moodle del fichero** son las siguientes:

- Los alumnos de ***Evaluación Continua, el día 8 de mayo*** *(hasta las 23:30 h. de la noche*).
- Los alumnos de ***Evaluación Final,*** según lo especificado en la normativa.

PARTE 1. EJERCICIOS

Ejercicio 1 (P4_E1). Árboles Binarios de Búsqueda de enteros

1. Implementación del TAD Árbol Binario de Búsqueda (Tree).

Al igual que en implementaciones de TADs anteriores, partimos de una estructura genérica que nos permite aislar la definición del TAD de los tipos que permite almacenar. En este caso, la estructura es la siguiente:

```
/* En tree.h */
typedef struct _Tree Tree;

/* En tree.c */
typedef struct _NodeBT {
        void* info;
        struct _NodeBT* left;
        struct _NodeBT* right;
} NodeBT;

struct _Tree {
    NodeBT *root;

    destroy_element_function_type   destroy_element_function;
    copy_element_function_type      copy_element_function;
    print_element_function_type     print_element_function;
    cmp_element_function_type       cmp_element_function;
};
```

En el apéndice 1 se muestran los prototipos de las funciones de este ejercicio.

2. Prueba del TAD Tree con datos de tipo entero.

Con el objetivo de evaluar el funcionamiento de las funciones codificadas, se desarrollará un programa **p4_e1.c** que trabajará con árboles binarios de búsqueda de enteros. Este programa recibirá como argumento un fichero, donde cada línea contendrá un número, y los introducirá uno a uno en el árbol. Una vez leídos todos los números, se mostrará por pantalla el número de nodos así como su profundidad. Al final, permitirá buscar un número introducido por consola.

Salida esperada para el siguiente fichero:

```
> cat numeros.txt
5
7
6
3
1
2
4
> ./p4_e1 numeros.txt
Numero de nodos: 7
Profundidad: 3
> Introduzca un numero: 5
Numero introducido: 5
El dato 5 se encuentra dentro del Arbol
```

Ejercicio 2 (P4_E2). Funciones de recorridos de un árbol

1. Implementación de funciones para recorrer el TAD Tree.

Complementando las funciones del prototipo del TAD Tree, se pide implementar distintos recorridos de un árbol binario. Los distintos recorridos serán en *orden previo*, *orden medio* y *orden posterior* según lo explicado en teoría. En todos los casos las funciones de recorrido volcarán en un archivo el resultado de la salida de las mismas (ver en apéndice 1 la definición de las funciones).

2. Prueba del módulo de recorridos.

Para probar el módulo de recorridos, basta con extender las pruebas realizadas en el ejercicio P4_E1 para que, además del número de nodos y la profundidad, se muestren por pantalla los tres recorridos del árbol.

Escribe un nuevo programa de nombre **p4_e2.c** que justo haga lo mismo que el p4_e1 pero que antes de pedir el número que debe ser buscado, muestre el árbol en los tres órdenes (pre, post e in).

Salida esperada para el caso de un árbol de enteros:

```
> ./p4_e2 numeros.txt
Numero de nodos: 7
Profundidad: 3
Orden previo: 5 3 1 2 4 7 6
Orden medio: 1 2 3 4 5 6 7
Orden posterior: 2 1 4 3 6 7 5
> Introduzca un numero: 9
Numero introducido: 9
El dato 9 NO se encuentra dentro del Arbol
```

Ejercicio 3 (P4_E3). Árboles Binarios de Búsqueda de nodos

1. Modificación del TAD Nodo (Node).

El TAD Node sirve para representar un nodo en un grafo. En este ejercicio modificaremos la definición utilizada hasta ahora en las prácticas por la siguiente, que permite un TAD con uno de los campos más genérico (no limitado a un tamaño máximo para el nombre, gracias al uso de memoria dinámica para el mismo):

```
/* En node_p4.c */

struct _Node {

   char* name;

   int id;

};
```

2. Prueba del TAD Tree con datos de tipo nodo.

Realiza un programa de prueba de este TAD Tree llamado ***p4_test-node-tree.c*** lea un archivo de nodos como el siguiente.

```
> cat nodos.txt
5 cinco
7 siete
6 seis
3 tres
1 uno
2 dos
4 cuatro
```

3. Comprobación de implementación usando un programa externo.

En este apartado se pide probar los prototipos de los TADs implementados hasta ahora de manera exhaustiva, utilizando un programa que os entregamos (***p4_e3.c***) que debería funcionar sin cambiar nada del mismo (y que se compila usando el fichero ***makefile_ext***. Este programa se puede utilizar con los ficheros dict.dat (y sus versiones dict10.dat, dict1K.dat, dict10K.dat y dict1M.dat), cargando los nodos contenidos en ellos e insertando dichos nodos en el árbol después de procesarlos de una manera concreta según cómo se le llame (básicamente, si se llama "***./p4_e3 <nobre_fichero> B***" se ordenan en memoria los nodos y se insertan para que el árbol se cree de la manera lo más balanceada posible, en cambio si se llama "***./p4_e3 <nombre_fichero> N***", se insertan en el árbol en el orden que se leen del fichero).

La salida esperada para uno de los ficheros que se entregan, ***dict10K.dat***, se muestra a continuación (los tiempos de creación y búsqueda pueden variar, ya que depende de la máquina que se utilice):

```
> ./p4_e3 dict10K.dat N
10000 líneas leídas

Tiempo de creación del árbol: 10000 ticks (0.010000 segundos)
Numero de nodos: 10000
Profundidad: 30
Introduce un nodo para buscar en el árbol (siguiendo el mismo formato que en el fichero de entrada):
3 a
Elemento NO encontrado!

Tiempo de búsqueda en el árbol: 0 ticks (0.000000 segundos)
```

```
> ./p4_e3 dict10K.dat B
10000 líneas leídas
Datos ordenados

Tiempo de creación del árbol: 10000 ticks (0.010000 segundos)
Numero de nodos: 10000
Profundidad: 13
Introduce un nodo para buscar en el árbol (siguiendo el mismo formato que en el fichero de entrada):
3 a
Elemento NO encontrado!

Tiempo de búsqueda en el árbol: 0 ticks (0.000000 segundos)
```

Nota: para indicar que se ha terminado de introducir el nodo, hay que pulsar *<intro>* en el teclado al terminar de escribir los datos del nodo.

A continuación se muestra la salida para el fichero más grande ***dict.dat***:

```
> ./p4_e3 dict.dat N
4001906 líneas leídas

Tiempo de creación del árbol: 10980000 ticks (10.980000 segundos)
Numero de nodos: 4001906
Profundidad: 54
Introduce un nodo para buscar en el árbol (siguiendo el mismo formato que en el fichero de entrada):
3 a
Elemento NO encontrado!

Tiempo de búsqueda en el árbol: 0 ticks (0.000000 segundos)
```

```
> ./p4_e3 dict.dat B
4001906 líneas leídas
Datos ordenados

Tiempo de creación del árbol: 3710000 ticks (3.710000 segundos)
Numero de nodos: 4001906
Profundidad: 21
Introduce un nodo para buscar en el árbol (siguiendo el mismo formato que en el fichero de entrada):
3 a
Elemento NO encontrado!

Tiempo de búsqueda en el árbol: 0 ticks (0.000000 segundos)
```

Ejercicio 4 (P4_E4). Árboles Binarios de Búsqueda de cadenas de caracteres

1. Implementación de un árbol que permita almacenar cadenas de caracteres.

En este ejercicio se pide implementar un árbol donde los elementos que se almacenen sean cadenas de caracteres. Para ello, hay que cumplir que:

- No se modifique la implementación de los árboles binarios.
- Las cadenas de caracteres no tengan un tamaño predefinido.

Explica en la memoria las decisiones que se han tenido que tomar para resolver este ejercicio.

2. Prueba de su correcto funcionamiento.

De manera similar al primer ejercicio, se pide codificar un programa **p4_e4.c** donde se prueben todas las funciones del prototipo del TAD Tree, leyendo los datos de un fichero. Una posible salida de este ejercicio sería (mostrando el orden medio del árbol resultante):

```
> cat cadenas.txt
arbol
abridor
antes
abatir
ahora
barcelona
madrid
bueno
```

```
> ./p4_e4 cadenas.txt
Numero de nodos: 8
Profundidad: 3
abatir abridor ahora antes arbol barcelona bueno madrid
Introduce una cadena para buscar en el árbol (siguiendo el mismo formato que en el fichero de entrada):
madrid
Elemento encontrado!
```

PARTE 2. PREGUNTAS SOBRE LA PRÁCTICA

1. El árbol que se crea a partir de los ficheros de nodos (*dict*.dat*), ¿es completo o casi completo? Justifica tu respuesta.

2. a) ¿Qué relación hay entre la "forma" de un árbol y sus recorridos?

b) ¿Se puede saber si un árbol binario de búsqueda está bien construido según sus recorridos?

3. Compara y describe las diferencias entre los árboles generados por los ejecutables *p4_e3* con el último argumento *B* o *N* (número de nodos, profundidad, recorridos, etc.).

Apéndice 1: tree.h

```
typedef struct _Tree Tree;

typedef void (*destroy_elementtree_function_type)(void*);

typedef void (*(*copy_elementtree_function_type)(const void*));

typedef int (*print_elementtree_function_type)(FILE *, const void*);

typedef int (*cmp_elementtree_function_type)(const void*, const void*);

/* Inicializa un Arbol vacio. */

Tree*   tree_ini(

   destroy_element_function_type f1,

   copy_element_function_type f2,

   print_element_function_type f3,

   cmp_element_function_type f4);

/* Indica si el Arbol esta o no vacio. */

Bool    tree_isEmpty( const Tree *pa);

/* Libera la memoria utilizada por un Arbol. */

void    tree_free(Tree *pa);

/* Inserta un elemento en un Arbol binario de busqueda copiándolo en memoria nueva. */

Status  tree_insert(Tree *pa, const void *po);

/* Escribe en el fichero f el recorrido de un Arbol en orden previo sin modificarlo. */

Status    tree_preOrder(FILE *f, const Tree *pa);

/* Escribe en el fichero f el recorrido de un Arbol en orden medio sin modificarlo. */

Status    tree_inOrder(FILE *f, const Tree *pa);

/* Escribe en el fichero f el recorrido de un Arbol en post orden sin modificarlo. */

Status    tree_postOrder(FILE *f, const Tree *pa);

/* Calcula la profundidad de un Arbol. Un Arbol vacio tiene profundidad -1. */

int    tree_depth(const Tree *pa);

/* Calcula el numero de nodos de un Arbol. Un Arbol vacio tiene 0 nodos. */

int    tree_numNodes(const Tree *pa);

/* Devuelve TRUE si se puede encontrar el elemento pe en el árbol pa */

Bool    tree_find(Tree* pa, const void* pe);
```