Programación II, 2015-2016
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 la generación de 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 parte de P4_E2. Cada profesor indicará en clase si se ha de realizar alguna entrega y cómo: papel, e-mail, Moodle, etc.
Semana 2: terminar P4_E2 y código de P4_E3.
Semana 3: 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: P1_Prog2_G2161_P05.zip).
Las **fechas de subida a Moodle del fichero** son las siguientes:

- Los alumnos de ***Evaluación Continua, el día 30 de abril*** *(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 {
        EleTree *info;
        struct _NodeBT *left;
        struct _NodeBT *right;
} NodeBT;

struct _Tree {
        NodeBT *root;
};
```

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

2. Implementación del TAD EleTree.

En este apartado se pide implementar un EleTree que encapsule un puntero a entero. El apéndice 2 muestra la lista de funciones que forman el prototipo de este TAD.

3. Prueba del TAD Tree con un EleTree que almacena un puntero a 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:

```
> cat numeros.txt
5
7
6
3
1
2
4
```

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

Ejercicio 2 (P4_E2). Árboles Binarios de Búsqueda de puntos

1. Modificación del TAD Punto (Point).

El TAD Point sirve para representar un punto en el espacio. En este ejercicio modificaremos la definición utilizada hasta ahora en las prácticas por la siguiente, que permite un TAD más genérico (no limitado a puntos de dos dimensiones como hasta ahora):

```
/* En point.c */
struct _Point {
   char* name;
   float* coord;
   int dim;
};
```

Las principales diferencias son:

- Cambiamos el símbolo por un nombre (reservando memoria dinámica para este campo).
- Las coordenadas son de tipo float.
- El número de coordenadas no está limitado.
- Las coordenadas se guardarán en memoria dinámica, por lo que tendremos que reservar memoria para ellas sabiendo que su cantidad se almacena en el atributo dim.

Las funciones asociadas a este TAD están listadas en el apéndice 3.

2. Prueba del TAD Tree con un EleTree que almacena un puntero a punto.

Una vez se ha modificado el TAD Point, para tener árboles binarios de búsqueda de puntos es necesario codificar las funciones de EleTree para una nueva implementación de la estructura EleTree. En este apartado se pide implementar las funciones del apéndice 2 con esta nueva versión del TAD EleTree.

3. Comprobación de implementación usando un programa externo.

En este apartado se pide probar los prototipos de los TADs implementados hasta ahora utilizando un programa que os entregamos (**p4_e2.c**) que debería funcionar sin cambiar nada del mismo (y que se compila usando el fichero **makefile_ext**, el cual genera varios ejecutables a partir del mismo fichero). Este programa se puede utilizar con los ficheros locations.txt (y sus versiones locations10.txt, locations1K.txt, locations10K.txt y locations50K.txt), cargando los puntos contenidos en ellos e insertando dichos puntos en el árbol después de procesarlos de una manera concreta (básicamente se ordenan en memoria y se insertan para que el árbol se cree de manera lo más balanceada posible). En el futuro veréis estrategias más generales (donde lo que se balancea es el árbol según se va creando) pero que están fuera del alcance de esta práctica, por lo que nos limitaremos a hacer un preprocesamiento *adecuado* de los datos de entrada.

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

```
> ./p4_e2 locations10K.txt
10000 líneas leídas

Tiempo de creación del árbol: 20000 ticks (0.020000 segundos)
Numero de nodos: 10000
Profundidad: 32
Introduce un punto para buscar en el árbol (siguiendo el mismo formato que en
el fichero de entrada):
2 1 1 a
Elemento NO encontrado!

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

```
> ./p4_e2_bal locations10K.txt
10000 líneas leídas
Datos ordenados

Tiempo de creación del árbol: 10000 ticks (0.010000 segundos)
Numero de nodos: 10000
Profundidad: 14
Introduce un punto para buscar en el árbol (siguiendo el mismo formato que en
el fichero de entrada):
2 1 1 a
Elemento NO encontrado!

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

A continuación se muestra la salida con un fichero más pequeño:

```
> cat puntos.txt
3 1 1 1 uno
3 1 2 1 dos
3 1 3 1 tres
```

```
> ./p4_e2 puntos.txt
3 líneas leídas

Tiempo de creación del árbol: 0 ticks (0.000000 segundos)
Numero de nodos: 3
Profundidad: 3
Introduce un punto para buscar en el árbol (siguiendo el mismo formato que en
el fichero de entrada):
3 1 1 1 a
Elemento NO encontrado!

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

```
> ./p4_e2_bal puntos.txt
3 líneas leídas
Datos ordenados

Tiempo de creación del árbol: 0 ticks (0.000000 segundos)
Numero de nodos: 3
Profundidad: 2
Introduce un punto para buscar en el árbol (siguiendo el mismo formato que en
el fichero de entrada):
3 1 1 1 uno
Elemento encontrado!

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

Ejercicio 3 (P4_E3). 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. (opcional) Extender con funciones que devuelven el recorrido del árbol en forma de lista.

De manera opcional, se pide codificar funciones similares a las anteriores que recorren un árbol pero almacenando el resultado en una lista (de manera que, con dicha lista, al imprimirla a un fichero se consiga el mismo resultado que invocando a los métodos que vuelcan en fichero). Los prototipos de estas funciones también se encuentran en el apéndice 1.

3. 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.

Salida esperada para el caso de un árbol de enteros:

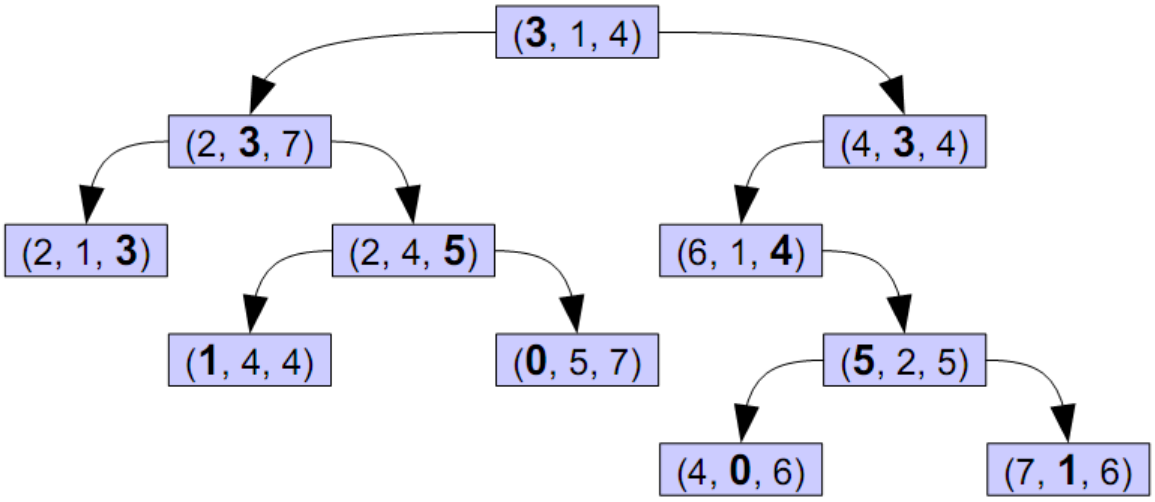
```
> ./p4_e3 numeros.txt
Numero de nodos: 7
Profundidad: 4
Orden previo: 5 3 1 2 4 7 6
Orden medio: 1 2 3 4 5 6 7
Orden posterior: 2 1 4 3 6 7 5
> Introduzca un numero: 9
Numero introducido: 9
El dato 9 NO se encuentra dentro del Arbol
```

Ejercicio 4 (P4_E4). Árboles Binarios de Búsqueda Dimensionales

1. Re-implementación del TAD Tree.

En este ejercicio se pide volver a implementar el TAD Tree pero codificando determinadas primitivas de manera que estén más orientadas a almacenar datos donde el concepto de "dimensión" esté bien definido (como los puntos).

Para implementar estos Árboles Binarios de Búsqueda Dimensionales (en un fichero de nombre **dtree.c**, usando el mismo tree.h que antes) tendremos que cambiar la forma en la que se insertan y buscan elementos en un árbol. En concreto, a cada nivel del árbol miraremos una coordenada distinta del elemento a insertar/buscar (de manera secuencial, volviendo a la primera coordenada cuando estemos en niveles mayores que el número de coordenadas), como en el ejemplo siguiente, donde se muestra en negrita la componente que se compara cuando se llega a ese nodo (tanto para insertar como para buscar):



Como se puede ver, es necesario que los EleTree tengan un concepto de dimensión, así como que soporten estas comparaciones por coordenadas. Por ello, en el apéndice 2 se incluyen dos funciones que permiten esa funcionalidad (las dos últimas).

Además, hay que notar que dado que el TAD Tree es una estructura opaca, nos veremos obligados a codificar de nuevo todas las primitivas y no sólo las que verdaderamente son distintas.

Por último, comentar que este tipo de estructura tiene los mismos costes de búsqueda e inserción que los árboles binarios de búsqueda normales, no obstante, facilitan un tipo de operación que no está bien definida en los árboles normales, es la denominada como *búsqueda de vecinos cercanos* y consiste en recuperar los N puntos que están más cercanos a uno dado. Esta operación queda fuera de la práctica, pero se deja como ejercicio pensar cómo se podría implementar (y cómo se podría llegar a hacer en árboles binarios de búsqueda estándares).

2. Modificación del TAD EleTree y Point.

Como se ha dicho antes, el TAD EleTree necesita integrar el concepto de dimensionalidad, para lo cual se han incluido dos funciones en el apéndice 2 (eletree_getDim y eletree_cmpd). En consecuencia, si se quieren tener árboles binarios de búsqueda dimensionales de puntos, habrá que completar su prototipo con dichas funcionalidades; en particular, se tendrá que implementar un método de comparación por coordenadas (point_cmpd, ya incluido en el apéndice 3).

3. Prueba de la nueva implementación del TAD Tree.

Para esta prueba basta con utilizar dos de los ejecutables que se generan usando el programa que se utilizó en P4_E2.

Estas son las salidas esperadas de los programas (comparadlas con las de P4_E2):

```
> ./p4_e4 locations10K.txt
10000 líneas leídas

Tiempo de creación del árbol: 40000 ticks (0.040000 segundos)
Numero de nodos: 10000
Profundidad: 30
Introduce un punto para buscar en el árbol (siguiendo el mismo formato que en
el fichero de entrada):
2 1 1 a
Elemento NO encontrado!

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

```
> ./p4_e4_bal locations10K.txt
10000 líneas leídas
Datos ordenados

Tiempo de creación del árbol: 20000 ticks (0.020000 segundos)
Numero de nodos: 10000
Profundidad: 18
Introduce un punto para buscar en el árbol (siguiendo el mismo formato que en
el fichero de entrada):
2 1 1 a
Elemento NO encontrado!

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

Como sugerencia, podéis modificar los programas para mostrar sus recorridos, confirmando que efectivamente los árboles se están construyendo de forma distinta; usando el mismo fichero que en P4_E2, este sería su orden medio con la definición estándar de Tree (P4_E2):

```
> ./p4_e2 puntos.txt
3 líneas leídas

Tiempo de creación del árbol: 0 ticks (0.000000 segundos)
Numero de nodos: 3
Profundidad: 3
[(1.00, 1.00, 1.00): uno][(1.00, 2.00, 1.00): dos][(1.00, 3.00, 1.00): tres]
Introduce un punto para buscar en el árbol (siguiendo el mismo formato que en
el fichero de entrada):
3 1 1 1 uno
Elemento encontrado!

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

Y este el orden medio con la nueva definición:

```
> ./p4_e4 puntos.txt
3 líneas leídas

Tiempo de creación del árbol: 0 ticks (0.000000 segundos)
Numero de nodos: 3
Profundidad: 3
[(1.00, 2.00, 1.00): dos][(1.00, 3.00, 1.00): tres][(1.00, 1.00, 1.00): uno]
Introduce un punto para buscar en el árbol (siguiendo el mismo formato que en
el fichero de entrada):
3 1 1 1 uno
Elemento encontrado!

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

Ejercicio 5 (P4_E5). (Opcional) 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 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_e5 cadenas.txt
Numero de nodos: 8
Profundidad: 4
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!
```

Si, además, se enlaza con el ejercicio anterior (dtree.o) una salida posible sería la siguiente:

```
> ./p4_e5d cadenas.txt
Numero de nodos: 8
Profundidad: 4
abatir abridor ahora antes arbol madrid barcelona bueno
Introduce una cadena para buscar en el árbol (siguiendo el mismo formato que
en el fichero de entrada):
despues
Elemento NO encontrado!
```

PARTE 2.A. PREGUNTAS SOBRE LA PRÁCTICA

1. El árbol que se crea a partir de los ficheros de puntos (locations*.txt), ¿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_e2, p4_e2_bal, p4_e4 y p4_e4_bal (número de nodos, profundidad, recorridos, etc.).

PARTE 2.B. MEMORIA SOBRE LA PRÁCTICA

1. Decisiones de diseño

Explicad las decisiones de diseño y alternativas que se han considerado durante la práctica para cada uno de los ejercicios propuestos.

2. Informe de uso de memoria

Elaborad un informe sobre la salida del análisis de memoria realizado por la herramienta memcheck de valgrind para cada uno de los ejercicios propuestos. Debe tenerse en cuenta que **en la ejecución del código entregado no deben producirse ningún aviso ni alerta sobre uso inapropiado de memoria**.

3. Conclusiones finales

Se reflejará al final de la memoria unas breves conclusiones sobre la práctica, indicando los beneficios que os ha aportado la práctica, qué aspectos vistos en las clases de teoría han sido reforzados, qué apartados de la práctica han sido más fácilmente resolubles y cuáles han sido los más complicados o no se han podido resolver, qué aspectos nuevos de programación se han aprendido, dificultades encontradas, etc.

Apéndice 1: tree.h

```
typedef struct _Tree Tree;

/* Inicializa el árbol reservando memoria */
Tree*   tree_ini();
/* Libera el árbol y todos sus elementos */
void    tree_free(Tree *pa);

/* Comprueba si un árbol está vacío */
Bool    tree_isEmpty( const Tree *pa);

/* Inserta en el árbol pa el EleTree po, devolviendo ERROR si el elemento ya existía en el árbol */
Status  tree_insert(Tree *pa, const EleTree *po);

/* Devuelve TRUE si se puede encontrar pe en el árbol pa */
Bool    tree_findEleTree(Tree* pa, const EleTree* pe);

/* Devuelve la profundidad del árbol pa, considerando que un árbol vacío tiene profundidad -1 */
int     tree_depth(const Tree *pa);
/* Devuelve el número de nodos del árbol pa, considerando que un árbol vacío tiene 0 nodos */
int     tree_numNodes(const Tree *pa);

/******* Funciones de recorrido del árbol (P4_E3) ********/
/* Muestra el recorrido de un árbol en orden previo */
Status  tree_preOrder(FILE *f, const Tree *pa);
/* Muestra el recorrido de un árbol en orden medio */
Status  tree_inOrder(FILE *f, const Tree *pa);
/* Muestra el recorrido de un árbol en orden posterior */
Status  tree_postOrder(FILE *f, const Tree *pa);
/****** Para el ejercicio opcional de P4_E3 *******/
/* Inserta en la lista l el recorrido de un árbol en orden previo */
Status  tree_preOrderToList(List *l, const Tree *pa);
/* Inserta en la lista l el recorrido de un árbol en orden medio */
Status  tree_inOrderToList(List *l, const Tree *pa);
/* Inserta en la lista l el recorrido de un árbol en orden posterior */
Status  tree_postOrderToList(List *l, const Tree *pa);
```

Apéndice 2: eletree.h

```
typedef struct _EleTree EleTree;

/* Inicializa un EleTree reservando memoria */
EleTree* eletree_ini();
/* Libera un EleTree */
void   eletree_free(EleTree* pe);

/* Modifica los datos de un EleTree */
EleTree* eletree_setInfo(EleTree* pe, void* info);
/* Devuelve los datos de un EleTree */
void*  eletree_getInfo(const EleTree* pe);

/* Compara dos EleTree's devolviendo un número negativo, cero o positivo según si pe1 es menor, igual o mayor que pe2 */
int eletree_cmp(const EleTree* pe1, const EleTree* pe2);

/* Copia reservando memoria un EleTree */
EleTree* eletree_copy(const EleTree* pOrigin);

/* Imprime un EleTree devolviendo el número de caracteres escritos */
int eletree_print(FILE* pf, const EleTree* pe);

/***** Estas dos funciones se necesitan a partir del ejercicio P4_E4 *****/
/* Devuelve la dimensión de un EleTree */
int eletree_getDim(const EleTree* pe);
/* Compara la componente dim-ésima entre dos EleTree's */
int eletree_cmpd(const EleTree* pe1, const EleTree* pe2, const int dim);
```

Apéndice 3: point.h

```
typedef struct _Point Point;

/* Reserva memoria para un punto que tenga dim dimensiones */
Point * point_ini(int dim);
/* Libera el punto */
void point_free(Point *);

/* Obtiene la coordenada dim-ésima de p y la almacena en v, devolviendo OK si todo va bien */
Status point_getCoordinate(const Point * p, const int dim, float * v);
/* Devuelve el nombre almacenado en p */
const char* point_getName(const Point * p);
/* Devuelve el número de dimensiones de p */
int point_getDimensions(const Point * p);

/* Asigna el valor v a la coordenada dim-ésima al punto p */
Point * point_setCoordinate(Point * p, const int dim, const float v);
/* Asigna el nombre name al punto p */
Point * point_setName(Point *p, const char* name);

/* Copia un punto reservando memoria para dicho punto y todos sus atributos, copiando sus valores */
Point * point_copy(const Point * src);

/* Imprime un punto p en el fichero pf, devolviendo el número de caracteres impresos */
int point_print(FILE *pf, const Point *p);

/* Compara p1 con p2 devolviendo un número negativo, cero o positivo según si p1 es menor, igual o mayor que p2 */
int point_cmp(const Point * p1, const Point * p2);
/***** Necesario a partir de P4_E4 *****/
/* Compara la coordenada dim-ésima de p1 y la de p2 devolviendo un número negativo, cero o positivo según si p1 es menor, igual o mayor que p2 */
int point_cmpd(const Point * p1, const Point * p2, const int dim);
```