PRÁCTICA 4: TAD Árbol Binario de Búsqueda (BST)

OBJETIVOS

- Aprender a utilizar el TAD Árbol Binario de Búsqueda para resolver problemas.
- Aprender a elegir la estructura de datos apropiada para implementar el TAD anterior.
- Codificar sus primitivas y utilizarlo en un programa principal.
- Aprender a manejar la recursión.

NORMAS

Los programas que se entreguen deben:

- Estar escritos en C, siguiendo las normas de programación establecidas.
- Compilar sin errores ni warnings incluyendo las banderas `-Wall` y `-pedantic`.
- 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.
- No producir fugas de memoria al ejecutarse.

PLAN DE TRABAJO

- **Semana 1:** Ejercicio 1.
- **Semana 2:** Ejercicio 2.
- **Semana 3:** Ejercicio 3.

Cada profesor indicará en clase cómo hacer el seguimiento, como por ejemplo entregas parciales semanales por Moodle o email, preguntas en clase, etc.

La entrega final se realizará a través de Moodle, siguiendo escrupulosamente las instrucciones indicadas en el enunciado referentes a la organización y nomenclatura de ficheros y proyectos. Se recuerda que el fichero comprimido que se debe entregar debe llamarse `Px_EDAT_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 2112: `P4_EDAT_G2112_P05.zip`).

El fichero comprimido debe contener la siguiente organización de ficheros:

```
--- P4_EDAT_Gy_Pz/
 |--- data
 |--- out
 |--- bstree.c
 |--- bstree.h
 |--- file_utils.c
 |--- file_utils.h
 |--- grades.h
 |--- Makefile
 |--- Makefile_ext
 |--- Memoria.txt
 |--- p4_e1.c
 |--- p4_e2.c
```

```
    |--- p4_e3.c
    |--- search_queue.c
    |--- search_queue.h
    |--- types.h
    |--- vertex.c
    |--- vertex.h
```

El archivo `Memoria.txt` debe contener vuestras respuestas a las reflexiones planteadas en este enunciado al final de cada ejercicio. También puede ser un archivo `.docx` o `.pdf`.

Las fechas de entrega son las siguientes:

- Los alumnos de Evaluación Continua, la semana del **5 de mayo** hasta la hora de inicio de la clase de prácticas. Fecha límite: 9 de mayo 23:59 horas.
- Los alumnos de Evaluación Final, según lo especificado en la normativa.

MAKEFILE

El archivo **Makefile** a entregar en esta práctica debe realizar lo siguiente:

1. Construir el programa `p4_e1`.
2. Construir el programa `p4_e2`. Podéis usar `file_utils.c` para las funciones `string_xxxx` necesarias.
3. Construir el programa `p4_e3`. Podéis usar `file_utils.c` para las funciones `float_xxxx` necesarias.
4. Incluir al final el makefile `makefile_ext` dado, el cual contiene reglas para ejecutar los programas `p4_e1` y `p4_e2` con diversos archivos y datos de entrada.

EJERCICIO 1

En este ejercicio se trabajará con el tipo abstracto de datos Árbol Binario de Búsqueda (BST, del inglés *Binary Search Tree*). Se proporcionan las declaraciones del TAD y sus primitivas en el fichero `bstree.h` y una implementación casi completa de `bstree.c` (donde solo faltan por completar las primitivas especificadas en los siguientes apartados). La estructura de datos que se usa para implementar el TAD BST es la siguiente:

```
struct _BSTree {
  BSTNode *root;
  P_ele_print print_ele;
  P_ele_cmp cmp_ele;
};
```

En concreto, contiene dos punteros a las funciones que servirán para imprimir y comparar elementos, que se pasan como argumentos al inicializar un árbol; además, contiene un puntero al nodo raíz. Los nodos del árbol son del tipo privado `_BSTNode`, que está formado por un puntero al elemento que alma cena y dos punteros a sus nodos hijos, izquierdo y derecho, que serán NULL si no existen. La estructura empleada queda, por tanto, como sigue:

```
typedef struct _BSTNode {
  void *info;
  struct BSTNode *left;
  struct BSTNode *right;
} BSTNode;
```

Al igual que en implementaciones anteriores, el campo `info` almacenará la referencia (no una copia) del elemento recibido.

Ejercicio 1a: Implementación de las primitivas de mínimo, máximo, contenido e inserción

Se pide implementar las siguientes primitivas públicas del TAD BST, junto con sus correspondientes funciones privadas (recursivas), y cuya definición completa se encuentra en bstree.h:

- `tree_find_min`: devuelve el elemento más pequeño almacenado en el árbol.
- `tree_find_max`: devuelve el elemento más grande almacenado en el árbol.
- `tree_contains`: comprueba si un elemento existe en el árbol.
- `tree_insert`: inserta un nuevo elemento en el árbol (si el elemento ya se encuentra en el árbol no se debe insertar, pero la función devolverá OK).

Ejercicio 1b: Implementación de la primitiva de eliminación

Se pide implementar la primitiva pública del TAD BST `tree_remove` para eliminar un elemento del árbol, cuya declaración se encuentra en `bstree.h`.

Para ello hay que implementar también la función privada recursiva `_bst_remove_rec`. Al igual que el resto de funciones privadas recursivas de BST, esta tendrá que recibir como argumento o bien el árbol (para poder acceder a las funciones que necesite), o bien directamente los punteros a función que requiera. En este caso, se aconseja diseñarlo de la segunda forma, de manera que `_bst_remove_rec` reciba como argumentos un puntero al nodo desde el que arrancará la búsqueda, un puntero al elemento que se está buscando (para ser eliminado) y un

puntero a la función de comparación. Lo primero que hará será comparar el elemento recibido con el campo `info` del nodo, usando la función de comparación. Si el elemento que se busca es menor que el que tiene el nodo continuará buscando en su subárbol izquierdo, y si es mayor en el subárbol derecho. En caso de que el elemento buscado esté en el nodo actual habrá que eliminarlo, y se dan tres posibles situaciones:

- Si el nodo no tiene ningún hijo simplemente se libera.
- Si tiene solo un hijo, ya sea izquierdo o derecho, se libera el nodo y se deja en su lugar a su hijo.
- En caso de tener dos hijos, hay que reemplazarlo por el siguiente elemento más grande, que será siempre el **elemento mínimo del subárbol derecho** (el elemento más pequeño del subárbol en el que todos los elementos son mayores que el elemento actual). Una vez reemplazado, se elimina el elemento del subárbol derecho, ya que ha cambiado de posición.

En cualquiera de los casos, la función devolverá un puntero al nodo que se quedará en esa posición del árbol, ya sea el nodo original, uno de sus hijos, o NULL si la posición queda vacía.

A continuación, se proporciona un pseudocódigo para la función privada recursiva `_bst_remove_rec`:

Algorithm 1: `_bst_remove_rec`: Función de eliminación recursiva para BST

```
Input  : Puntero al nodo actual del árbol: pn
         Puntero al elemento buscado: elem
         Puntero a la función de comparación: cmp_elem
output: Puntero al nodo que quedará en la posición actual
1  if pn es nulo then
2      return NULL
   /* Comparación de elem con la información del nodo actual.                    */
3  if elem es menor que pn.info then
       /* Buscar en el subárbol izquierdo.                                       */
4      pn.left = _bst_remove_rec(pn.left, elem, cmp_elem)
5  else if elem es mayor que pn.info then
       /* Buscar en el subárbol derecho.                                         */
6      pn.right = _bst_remove_rec(pn.right, elem, cmp_elem)
7  else if elem es igual que pn.info then
       /* Se ha encontrado el elemento que se va a eliminar.                     */
8      if pn no tiene hijos then
9          _bst_node_free(pn)
10         return NULL
11     else if pn solo tiene hijo derecho then
12         ret_node = pn.right
13         _bst_node_free(pn)
14         return ret_node
15     else if pn solo tiene hijo izquierdo then
16         ret_node = pn.left
17         _bst_node_free(pn)
18         return ret_node
19     else if pn tiene ambos hijos then
           /* Se libera pn y se reemplaza por el mínimo del subárbol derecho, que se elimina de su posición actual. */
20         aux_node = _bst_find_min_rec(pn.right)
21         pn.info = aux_node.info
22         pn.right = _bst_remove_rec(pn.right, aux_node.info, cmp_elem)
23         return pn
24 return pn
```

Ejercicio 1c: Comprobación de la implementación completa del TAD BST

Para probar la implementación de este TAD se proporciona el programa principal `p4_e1.c` que debe funcionar sin fugas de memoria, y producir la salida esperada que se incluye en el fichero `run_e1.out` (puede haber discrepancias en los mensajes de tiempos). Se proporcionan además más ejemplos de ficheros de datos con los que probar este programa.

Es necesario estudiar el código proporcionado para entender qué operaciones se realizan y qué primitivas del TAD se están utilizando, prestando especial atención a la opción que ordena o no los datos previamente a través de la función `qsort`.

Reflexiones sobre el TAD BST

Al ejecutar el programa `p4_e1.c` se puede comprobar que los tiempos de creación y búsqueda varían mucho según si el programa se ejecuta en modo normal o en modo ordenado.

- ¿Por qué es así?
- ¿Hay alguna propiedad del árbol que permita explicar este comportamiento?

EJERCICIO 2

En este ejercicio se implementará un TAD Cola de Búsqueda (SQ, del inglés SearchQueue) que almacenará los elementos ordenados según una función de comparación que recibirá como puntero al inicializarse.
Para implementarlo se usará como base el TAD BST del ejercicio anterior, que permite mantener los elementos ordenados. A la hora de extraer elementos siempre se hará según **el elemento mínimo de dicho árbol**.

Ejercicio 2a: Implementación de la cola de búsqueda

Se pide implementar en `search_queue.c` las primitivas de `search_queue.h`, usando para ello el TAD BST, teniendo en cuenta que la estructura de datos para el nuevo TAD SQ será:

```
struct _SearchQueue {
  BSTree *data;
};
```

Ejercicio 2b: Ordenación de ficheros de texto

Para comprobar el correcto funcionamiento del TAD SQ, se pide implementar un nuevo programa principal, `p4_e2.c`, que reciba dos argumentos, correspondientes al nombre de un fichero de entrada, y al nombre de un fichero de salida.

A continuación, el programa leerá cada línea del fichero de entrada, insertando la cadena leída en una cola SQ. Una vez leído el fichero de entrada, se vaciará la cola elemento a elemento imprimiendo cada línea en el fichero de salida. De esta forma, cuando termine la ejecución el fichero de salida deberá contener las líneas del fichero de entrada, pero ordenadas alfabéticamente.

Podéis usar `file_utils.c` para las funciones `string_xxxx` necesarias.

El programa debe funcionar sin fugas de memoria. Como ejemplo de ejecución, se puede ver el fichero de salida esperado `data_string_10.out` cuando el fichero de entrada es `data_string_10.txt`.

Reflexiones sobre el TAD SQ

- ¿Qué diferencias y similitudes hay entre el TAD SQ y el TAD Cola de la práctica anterior?
- ¿Qué coste (aproximado) tiene la operación de extraer un elemento en el TAD SQ? ¿Sería posible hacer que esta operación fuera $\mathcal{O}(1)$?

EJERCICIO 3

En este ejercicio realizaremos un programa de prueba de vuestro TAD `SearchQueue`. Este programa debe hacer lo siguiente:

1. Leer las calificaciones en un fichero de entrada que recibe como argumento.
2. Almacenarlas en una `SearchQueue` implementada sobre un árbol binario de búsqueda.
3. Procesar la información para producir estas salidas en este orden:
    - **Todas las notas ordenadas de menor a mayor** (usando `search_queue_print`).
    - **Media** de las notas (con 2 cifras decimales).
    - **Mediana** de las notas (con 2 cifras decimales).
    - **Tres notas más bajas** (usando `search_queue_pop`, una a una).
    - **Tres notas más altas** (usando `search_queue_getBack` y eliminando la nota del árbol para evitar repetición).

Podéis usar `file_utils.c` para las funciones `float_xxxx` necesarias.

Un ejemplo de salida para el fichero `grades.txt` proporcionado, el cual contiene 20 notas, es el siguiente. Generad esta salida en la terminal, usando `stdout`.

```
Ordered grades: 2.90 3.30 3.90 4.00 4.40 5.50 5.80 6.20 6.50 6.70 6.80 7.00 7.10 7.30 7.80 8.00 8.30 8.50 9.20 9.90

Mean: 6.47

Median: 6.60

Lowest grades: 2.90 3.30 3.90

Highest grades: 9.90 9.20 8.50
```