

# Algoritmos y Estructuras de Datos II

TALLER - 23 de mayo de 2024

## Laboratorio 6: Árboles Binarios de Búsqueda

- Revisión 2024: Marco Rocchietti

### Objetivos

1. Implementar el TAD ABB con sus operaciones elementales
2. Familiarizarse con una interfaz de usuario básica TUI
3. Definir la invariante de representación de un ABB
4. Manejo de Strings en C
5. Implementar un TAD String
6. Implementar un TAD Diccionario
7. Utilizar `valgrind` para eliminar *memory leaks*
8. Utilizar `gdb` para erradicar bugs en los programas

### Ejercicio 1: TAD ABB

Se debe implementar el TAD `abb` (árbol binario de búsqueda) siguiendo la especificación que se encuentra en `abb.h`. Este tipo abstracto de datos está diseñado para guardar enteros en una estructura de árbol binario siguiendo la definición vista en el teórico. El TAD no permite tener elementos repetidos, por lo cual es parecido a un conjunto en ese aspecto.

En el archivo `abb.c` está dada la estructura de representación `struct s_abb` que tiene la siguiente definición:

```
struct _s_abb {
    abb_elem elem;          // Elemento del nodo
    struct s_abb *left;     // Rama izquierda
    struct s_abb *right;    // Rama derecha
};
```

Se encuentra definida de manera incompleta la función:

```
static bool invrep(abb tree)
```

que debe verificar la invariante de representación del TAD. La invariante debe asegurar que la estructura de nodos es consistente con la definición de Árbol Binario de Búsqueda. Para poder verificar la propiedad fundamental de los ABB de manera “sencilla”, será necesario programar esta función de manera **recursiva**. La interfaz del TAD cuenta con las siguientes funciones que se deben implementar:

Función	Descripción
<code>abb abb_empty(void)</code>	Crea un árbol binario de búsqueda vacío
<code>abb abb_add(abb tree, abb_elem e)</code>	Agrega un nuevo elemento al árbol
<code>bool abb_is_empty(abb tree)</code>	Indica si el árbol está vacío
<code>bool abb_exists(abb tree, abb_elem e)</code>	Indica si el elemento <code>e</code> está dentro de <code>tree</code>
<code>unsigned int abb_length(abb tree)</code>	Devuelve la cantidad de elementos del árbol
<code>abb abb_remove(abb tree, abb_elem e)</code>	Elimina el elemento <code>e</code> del árbol <code>tree</code>
<code>abb_elem abb_root(abb tree)</code>	Devuelve el elemento que está actualmente en la raíz del árbol
<code>abb_elem abb_max(abb tree)</code>	Devuelve el máximo elemento del árbol
<code>abb_elem abb_min(abb tree)</code>	Devuelve el mínimo elemento del árbol
<code>void abb_dump(abb tree, abb_ord ord)</code>	Muestra el contenido del árbol por la pantalla usando el orden indicado por <code>ord</code>
<code>abb abb_destroy(abb tree)</code>	Destruye la instancia <code>tree</code> liberando toda la memoria utilizada.

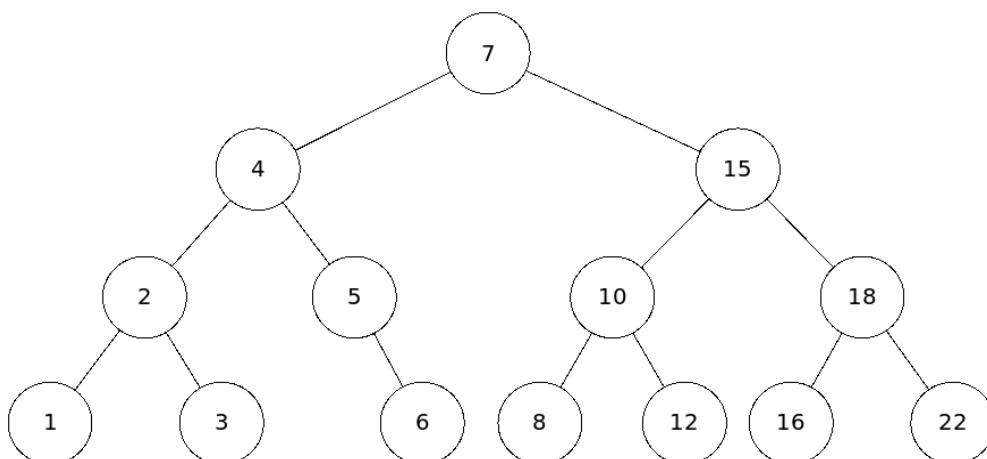
Notar que en `abb.h` se especifican las pre y postcondiciones de las funciones del TAD, y en `abb.c` se verifican estas condiciones con `assert()` para asegurar que la implementación que se realice cumpla con estas propiedades.

Una buena idea, dada la naturaleza recursiva de los árboles, es pensar sus operaciones de forma recursiva. Se pueden dar definiciones iterativas o recursivas según se crea conveniente, pero seguramente **el código se va a simplificar bastante usando recursión**. Se incluye una implementación de `abb_dump()` como ejemplo que, aunque luego hay que completar, se puede ver cómo se recorre el árbol completo aprovechando la recursión. Las operaciones de `add`, `exists`, `max`, `min` y `destroy` salen realmente fácil aprovechando esta técnica de programación.

La operación más delicada que se debe implementar es

```
abb abb_remove(abb tree, abb_elem e)
```

ya que no es trivial cómo eliminar un nodo del árbol sin romper la definición de ABB.



Borrar cualquiera de las hojas (elementos 1, 3, 6, 8, 12, 16 y 22) es muy sencillo, pero para borrar el elemento 4 hay que pensar cómo reorganizar el árbol.

Para probar la implementación se incluye el archivo `main.c` que construye un árbol binario de búsqueda a partir de un archivo y lo muestra por pantalla, junto con la raíz del árbol, el mínimo y el máximo. Para compilar se puede usar el `Makefile` incluido haciendo:

```
$ make
```

y luego, para ejecutar el programa con el archivo `abb_example.in` de entrada:

```
$ ./readtree input/abb_example.in
```

El archivo `abb_example.in` tiene el árbol de ejemplo mostrado anteriormente.

**a)** Completar las implementaciones de las operaciones descritas anteriormente a excepción de `abb_dump()` la cual tiene una implementación que funciona aunque está incompleta.

**b)** Modificar el archivo `main.c` para que luego de cargar el árbol desde el archivo de entrada, se puedan probar las distintas funcionalidades del TAD. Se debe implementar con un ciclo entonces una interfaz que permita al usuario realizar una de las siguientes operaciones en cada iteración:

1. Mostrar el árbol por pantalla
2. Agregar un elemento
3. Eliminar un elemento
4. Chequear existencia de elemento
5. Mostrar la longitud del árbol
6. Mostrar raíz, máximo y mínimo del árbol
7. Salir del programa

Para elegir qué acción realizar se debe solicitar un número de entrada. Para las opciones 2, 3 y 4 se le deberá pedir al usuario que ingrese el elemento a agregar, eliminar o chequear respectivamente. Al salir debe liberarse toda la memoria utilizada.



*Asegurarse de que la implementación esté libre de memory leaks usando la herramienta `valgrind` con la opción `--leak-check=full`*

**c)** En este punto el programa muestra el contenido del árbol con los elementos en orden creciente. ¿Se entiende por qué? ¿Cómo es el árbol que se obtiene si los elementos se agregan en ese orden? Completar la función y dividir su definición según el valor del parámetro `ord`. Para el caso `ABB_IN_ORDER` usar la definición original, para el caso `ABB_PRE_ORDER` hacer que se muestren los elementos en un orden que permita reconstruir el árbol original (debe seguir siendo recursiva la definición). El caso `ABB_POST_ORDER` completarlo con un criterio que parezca conveniente. Finalmente cambiar `main.c` para que se use a `abb_dump()` con el modo `ABB_PRE_ORDER`.

**d\*) OPCIONAL:** Agregar una opción al programa para definir en qué modo se usa `abb_dump()`, cada modo se refiere a una de las constantes `ABB_PRE_ORDER`, `ABB_IN_ORDER` o `ABB_POST_ORDER`.

## Ejercicio 2: TAD String

Se va a implementar en el próximo ejercicio un TAD Diccionario, pero para ello será necesario tener un manejo de cadenas más práctico del que provee C. Para ello se definirá el TAD String.

a) Completar la implementación del TAD String, la interfaz del TAD tiene las siguientes funciones:

Función	Descripción
<code>string string_create(const char *word)</code>	Crea un nuevo string a partir de la cadena en <code>word</code>
<code>unsigned int string_length(string str)</code>	Devuelve la longitud del <i>string</i>
<code>bool string_less(string str1, string str2)</code>	Indica si <code>str1</code> es menor que <code>str2</code> usando el orden alfabético habitual.
<code>bool string_eq(string str1, string str2)</code>	Indica si la cadena <code>str1</code> tiene el mismo contenido que la cadena <code>str2</code>
<code>string string_clone(string str)</code>	Genera una copia del string <code>str</code>
<code>const char* string_ref(string str)</code>	Devuelve un puntero al contenido de la cadena <code>str</code>
<code>void string_dump(string str, FILE *file)</code>	Escribe el contenido de <code>str</code> en el archivo <code>file</code>
<code>string string_destroy(string str)</code>	Destruye <code>str</code> liberando la memoria utilizada.

b) Crear un archivo `main.c` que utilice las funciones `string_less()` y `string_eq()`.

## Ejercicio 3: TAD Diccionario

En este ejercicio se debe implementar el TAD Diccionario. Como su nombre sugiere, el TAD almacenará palabras y definiciones (cada palabra tiene exactamente una definición). Las funcionalidades incluyen la búsqueda de palabras, agregar una nueva palabra junto con su definición, reemplazar la definición de una palabra ya existente, etc. La implementación estará basada en la especificación del teórico:

```
type Node of (K, V) = tuple
    left: pointer to (Node of (K,V))
    key: K
    value: V
    right: pointer to (Node of (K,V))
end tuple


type Dict of (K, V) = pointer to (Node of (K,V))
```


es decir, se implementa como un árbol binario de búsqueda cuyos nodos contienen una clave y un valor. Las claves serán las palabras y los valores son las definiciones de las mismas. El tipo para las claves será `key_t` y para los valores `value_t`, ambos definidos en el archivo `key_value.h`. De esta manera, variando la definición de las claves y valores podemos hacer diccionarios que contengan distintos tipos. Para este ejercicio en particular se necesitará guardar *strings* en el diccionario, por lo tanto `key_t` y `value_t` se definen ambos como sinónimos del TAD *String* implementado en el ejercicio 2 de donde debe copiarse el archivo `string.c`.

Las operaciones del TAD Diccionario se listan a continuación:

Función	Descripción
<code>dict_t dict_empty(void)</code>	Crea un diccionario vacío
<code>dict_t dict_add(dict_t dict, key_t word, value_t def)</code>	Agrega una nueva palabra <code>word</code> junto con su definición <code>def</code> . En caso que <code>word</code> ya esté en el diccionario, se actualiza su definición con <code>def</code> .
<code>value_t dict_search(dict_t dict, key_t word)</code>	Devuelve la definición de <code>word</code> contenida en el diccionario. Si no se encuentra devuelve <code>NULL</code>
<code>bool dict_exists(dict_t dict, key_t word)</code>	Indica si la palabra <code>word</code> está en el diccionario <code>dict</code>
<code>unsigned int dict_length(dict_t dict)</code>	Devuelve la cantidad de palabras que tiene actualmente el diccionario <code>dict</code>
<code>dict_t dict_remove(dict_t dict, key_t word)</code>	Elimina la palabra <code>word</code> del diccionario. Si la palabra no se encuentra devuelve el diccionario sin cambios.
<code>dict_t dict_remove_all(dict_t dict)</code>	Elimina todas las palabras del diccionario <code>dict</code>
<code>void dict_dump(dict_t dict, FILE *file)</code>	Escribe el contenido del diccionario <code>dict</code> en el archivo <code>file</code>
<code>dict_t dict_destroy(dict_t dict)</code>	Destruye la instancia <code>dict</code>

Para implementar la mayoría de las operaciones pueden adaptar el código hecho en el *ejercicio 1*.

	<i><b>Se recomienda</b> dedicar un tiempo para estudiar todos los archivos involucrados y así entender el desarrollo en general. Recordar que en los archivos de headers (los <code>.h</code>) se encuentran las descripciones y guías para la correcta implementación.</i>
---	---

	<i>Asegurarse de que la implementación esté libre de memory leaks usando la herramienta <code>valgrind</code> con la opción <code>--leak-check=full</code></i>
---	--

**a)** Implementar el TAD Diccionario. Completar la definición de la *invariante de representación* y chequear las pre y post condiciones de `dict.h` al estilo de lo que se vio en `abb.c` en el *ejercicio 1*. Asegurarse de **mantener el encapsulamiento** del TAD y la abstracción de los tipos `key_t` y `value_t`.

**b)** Completar la interfaz de usuario con las llamadas a las funciones que correspondan según la operación elegida por el usuario.

**c)** Copiar el `Makefile` del *ejercicio 1* y modificarlo para poder compilar este ejercicio y generar el ejecutable `dictionary`, es decir poder probar el ejercicio haciendo:

```
$ make
```

y luego

```
$ ./dictionary
```