Algoritmos y Estructuras de Datos II

TALLER - 18 de mayo de 2021

Laboratorio 6: Árboles Binarios de Búsqueda

- Revisión 2021: Marco Rocchietti
- Revisión 2020: Gonzalo Peralta
- Revisión 2019: Marco Rocchietti

Objetivos

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

Ejercicio 1: TAD ABB

a) Deben 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
};
```

Van a encontrar 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 deben implementar:

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

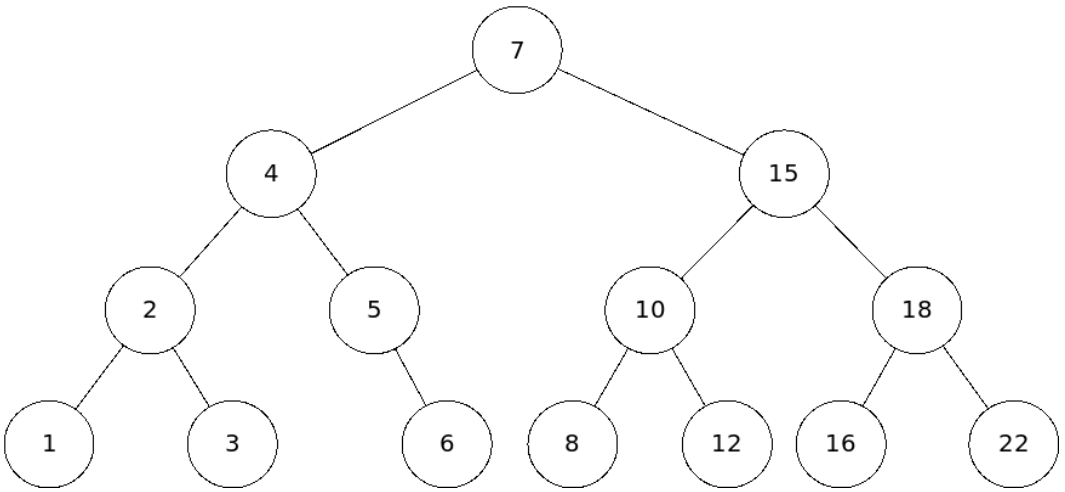
Van a notar que en **`abb.h`** se especifican las pre y postcondiciones de las funciones del TAD, y en **`abb.c`** se verifican estas condiciones con `assert()` para asegurar que la implementación que realicen cumpla con estas propiedades.

Una buena idea, dada la naturaleza recursiva de los árboles, es pensar sus operación de forma recursiva. Pueden dar definiciones iterativas o recursivas según lo crean conveniente, pero seguramente **el código se va a simplificar bastante usando recursión**. Se incluye una implementación de `abb_dump()` como ejemplo, donde 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 deberán 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 se reorganizar el árbol.

Para probar su implementación incluimos 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 pueden usar el **`Makefile`** incluído, por lo que solo deben ejecutar:

```
$ 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. Pueden modificar a **`main.c`** para probar la función **`abb_remove()`**. Una buena opción es permitir al usuario elegir un elemento para borrar, de manera tal que sea cómodo probar el correcto funcionamiento de esa función para distintos casos.

b) La función `abb_dump()` muestra por pantalla los elementos de manera ordenada. ¿Por qué se comporta de esta manera? ¿Cómo es el árbol que se obtiene agregando los elementos en el orden que muestra el contenido `abb_dump()`? Modificar la función `abb_dump()` para que muestre el contenido en el orden que permita reconstruir el árbol original (debe seguir siendo recursiva la definición).

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

Ejercicio 2: TAD String

Las cadenas en C se implementan como arreglos de caracteres. Los caracteres son valores del tipo `char` (que representa exactamente un caracter de *1 byte*), entonces para guardar un *string* en C podemos usar el siguiente arreglo:

```
char cadena[5];
```

En este ejemplo, `cadena` tiene capacidad para guardar un *string* de hasta 4 (cuatro) caracteres de longitud. Esto es así porque toda cadena en C debe terminar con el caracter `'\0'`, por lo cual ya tenemos un lugar ocupado. Si queremos armar el *string* con la palabra “hola” podemos hacer:

```
char cadena[5]={'h', 'o', 'l', 'a', '\0'};
printf("cadena: %s\n", cadena);
```

Otra forma más cómoda es hacer algo como:

```
char cadena[5]="hola";
printf("cadena: %s\n", cadena);
```

en este caso el caracter `'\0'` se agrega implícitamente en el arreglo `cadena`. Para no tener que contar la cantidad de caracteres que necesitamos, y cómo los *arrays* y los punteros son casi lo mismo en C, se puede definir una cadena directamente haciendo:

```
char *cadena="hola mundo!";
printf("cadena: %s\n", cadena);
```

el contenido del *array* apuntado por `cadena` es el siguiente:

'h'	'o'	'l'	'a'	' '	'm'	'u'	'n'	'd'	'o'	'!'	'\0'

Todas las cadenas que vimos hasta acá son estáticas, es decir que no pedimos memoria dinámica para construirlas. De hecho si hacemos `free(cadena)` se va a generar un error de memoria. Hay muchas funciones de la librería estándar de C para manejo de cadenas. Pueden investigarlas consultando las páginas de manual de linux:

```
$ man string
```

Muchas veces cuando necesitamos almacenar *strings* se requiere generar copias de su contenido en memoria dinámica y es importante saber cuándo hay que liberar esta memoria.

a) Deben completar la implementación del TAD String. La interfaz del TAD tiene las siguientes funciones:

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

```
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 vamos a necesitar 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.

Las operaciones del TAD Diccionario se listan a continuación:

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

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

SE RECOMIENDA dedicar un tiempo para estudiar todos los archivos involucrados y así entender el desarrollo en general. Recordar que en los archivos de *headers* (los .h) se encuentran las descripciones y guías para la correcta implementación.

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

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
```