Proyecto 3 - Árboles Binarios de Búsqueda

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Natalia Bidart, Matías Bordese, Diego Dubois, Leonardo Rodríguez.

1. Objetivo

El objetivo de este proyecto es extender el proyecto 2 de manera tal que el TAD `dict` use ahora, como contenedor de palabras y definiciones, un árbol binario de búsqueda (en vez de usar listas).

Para ello, habrá que definir y proveer una implementación en C del siguiente TAD:

- Árbol Binario de Búsqueda (en inglés, *Binary Search Tree*). De ahora en más, nos referiremos a éstos como **BST**: http://en.wikipedia.org/wiki/Binary_Search_Tree

2. Instrucciones

En la figura 1 se muestra un diagrama análogo al presentado en el proyecto anterior, con la diferencia de que ahora el TAD a implementar es el BST (ver los módulos resaltados en **rojo**).

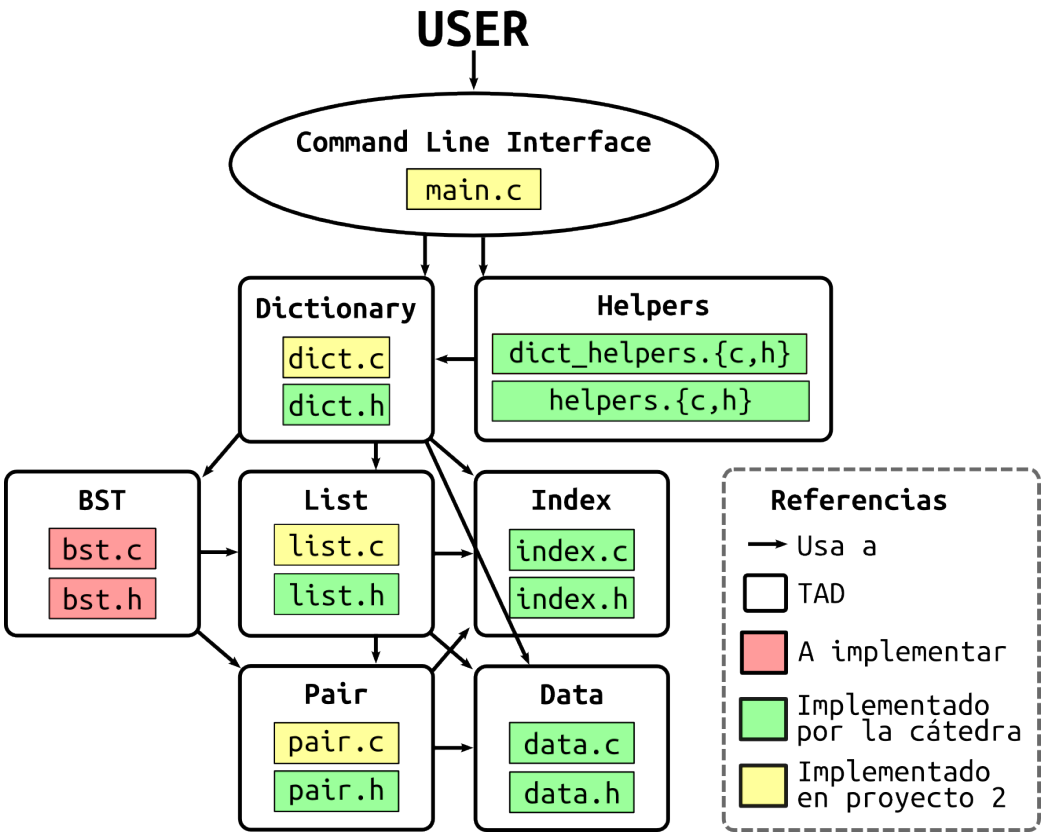

Figura 1: Diagrama de TADs

Notar que para este proyecto no se les entrega ningún archivo, con lo cual se tendrán que crear desde cero los módulos a desarrollar. Las tareas de este proyecto son las siguientes:

- Crear archivos `bst.h` y `bst.c` en donde se provea la interfaz e implementación, respectivamente, del tipo de datos "árbol binario de búsqueda", cumpliendo al pie de la letra las siguientes especificaciones:

```c
#ifndef _BST_H
#define _BST_H

#include <stdio.h>
#include <stdbool.h>
#include "data.h"
#include "index.h"
#include "list.h"


typedef struct _tree_node_t *bst_t;

bst_t bst_empty(void);
/*
 * Return a newly created, empty binary search tree (BST).
 *
 * The caller must call bst_destroy when done using the resulting BST,
 * so the resources allocated by this BST constructor are freed.
 *
 * POST: the result is a valid BST pointer, and bst_length(result) is 0.
 */

bst_t bst_destroy(bst_t bst);
/*
 * Free the resources allocated for the given 'bst', and set it to NULL.
 */

unsigned int bst_length(bst_t bst);
/*
 * Return the amount of elements in the given 'bst'.
 * This method has a linear order complexity. For reference, see:
 * http://en.wikipedia.org/wiki/Linear_time#Linear_time
 *
 * PRE: 'bst' is not NULL.
 */

bool bst_is_equal(bst_t bst, bst_t other);
/*
 * Return whether the given 'bst' is equal to 'other'.
 *
 * Equality is defined by comparing both BSTs, node by node, and ensuring that
 * each node is equal as a whole (ie, that both pairs are equal).
 *
 * PRE: 'bst' and 'other' are valid BST pointers.
 */

data_t bst_search(bst_t bst, index_t index);
/*
```

```
 * Return the data associated to the given 'index' in the given 'bst',
 * or NULL if the 'index' is not in 'bst'.
 *
 * The caller must NOT free the resources allocated for the result when done
 * using it.
 *
 * PRE: 'bst' and 'index' are valid pointers.
 */

bst_t bst_add(bst_t bst, index_t index, data_t data);
/*
 * Return the given 'bst' with the pair ('index', 'data') added to it.
 *
 * The given 'index' and 'data' are inserted in the BST,
 * so they can not be destroyed by the caller (they will be destroyed when
 * bst_destroy is called).
 *
 * PRE: all 'bst', 'index' and 'data' are valid pointers.
 * Also, there is no pair in the given BST such as its index is equal to
 * 'index' (this means, bst_search for 'index' must be NULL).
 *
 * POST: the length of the result is the same as the length of 'bst'
 * plus one. The elements of the result are the same as the one in 'bst'
 * with the new pair ('index', 'data') added accordingly (see:
 * http://en.wikipedia.org/wiki/Binary_search_tree
 * for specifications about behavior).
 */

bst_t bst_remove(bst_t bst, index_t index);
/*
 * Return the given 'bst' with the pair which index is equal to 'index'
 * removed.
 *
 * Please note that 'index' may not be in the BST (thus an unchanged
 * BST is returned).
 *
 * PRE: both 'bst' and 'index' are valid pointers.
 *
 * POST: the length of the result is the same as the length of 'bst'
 * minus one if 'index' existed in 'bst'. The elements of the result are
 * the same as the ones in 'bst' with the entry for 'index' removed.
 */

bst_t bst_copy(bst_t bst);
/*
 * Return a newly created copy of the given 'bst'.
 *
 * The caller must call bst_destroy when done using the resulting BST,
 * so the resources allocated by this BST contructor are freed.
 *
 * POST: the result is a valid BST pointer and it is an exact copy of 'bst'.
 * In particular, bst_is_equal(result, bst) holds.
 */
```

```c
list_t bst_to_list(bst_t bst, list_t list);
/*
 * Return a sequence that is a flatten representation of 'bst'.
 *
 * PRE: 'bst' is a valid BST pointer, 'list' is a valid list pointer
 *
 * POST: the result is a valid list, and the result's length is the same
 * as the given BST's length. Every pair in the BST exists in the returned
 * list, and the BST ordering is preserved.
 *
 * In other words, the resulting list has to be in ascending ordered, not
 * because a sorting algorithm was applied to the list, but because it was
 * built in a way that elements are ordered.
 *
 */

#endif
```

- La implementación del TAD mencionado arriba debe ser oculta. Es decir, deben usar la técnica de punteros a estructuras cuando implementen el TAD requerido (tal cual como han hecho hasta ahora con el `dict` y las `list`).
- Hacer las modificaciones mínimas a `dict.c` tales que la implementación del mismo use árboles en vez de listas enlazadas. Notar que la interfaz del diccionario (el `dict.h`) debe quedar idéntica al proyecto pasado, y asimismo, el resto de los archivos no deben sufrir cambios (excepto que necesiten arreglar algún bug). Es decir, todos los demás TADs deben quedar iguales a los del proyecto anterior (inclusive la interfaz con el usuario).
- Testear esta nueva implementación, confirmando que siempre estén usando código objeto enteramente producido por ustedes. Es decir, borrar todos los `.o` antes de compilar.
- Proveer un `Makefile` tal que se pueda compilar el proyecto al correr el comando `make`.

2.1. Detalles sobre el TAD `bst`

El TAD `bst` a implementar en C es análogo al árbol binario de búsqueda visto en el teórico.

La estructura `struct _tree_node_t` mostrada en el `bst.h` representa a un BST, que como se muestra en el apunte 06.arbolesbinarios.pdf, consta de:

- su elemento raíz, que almacena el valor (un `pair_t` en este caso)
- su sub-árbol izquierdo (que es también un `bst_t`)
- su sub-árbol derecho (que, oh sorpresa, es también un `bst_t`)

Es tarea de los alumnos pensar y definir correctamente esta estructura, oculta dentro del `bst.c`.

Luego, la implementación de todos los métodos públicos, que debe respetar los algoritmos descriptos en el apunte del teórico:

- `bst_destroy`
- `bst_is_equal`
- `bst_search`

- `bst_add`
- `bst_remove`
- `bst_copy`
- `bst_to_list`

va a estar hecha en función de llamadas **recursivas** a ellas mismas.

Para la implementación de `bst_to_list`, leer detenidamente http://en.wikipedia.org/wiki/Tree_sort, que explica en detalle las tres formas de aplanar un árbol. Usar las listas enlazadas del proyecto 2 como estructura para guardar el árbol aplanado (dejar esta implementación para el final).

Desde el `dict`, `bst_to_list` tiene que ser llamado con el BST interno como primer parámetro, y una lista vacía como segundo parámetro.

2.1.1. Un poco de terminología

Una **hoja** (*leaf*) es un árbol tal que:

- Su nodo es un `pair_t` no vacío.
- Su árbol izquierdo es vacío.
- Su árbol derecho es vacío.

Es decir, algo de la siguiente forma representa una hoja:

```
("perro", "animal que ladra")
     /    \
    /      \
  NULL     NULL
```

Por el contrario, lo siguiente es un árbol **inválido**:

```
      NULL
     /    \
    /      \
  NULL     NULL
```

Es importante ver que para hacer una implementación eficiente del método `bst_to_list`, **no** se debe usar el TAD `list` que mantiene los elementos ordenados (que sólo existe si hicieron el estrella 2 del proyecto 2). Es decir, usar la implementación de lista básica que provee `list_append` y agrega elementos al final, y no la que tiene `list_add`.

3. Puntos ⋆

Recomendación usual: no empezar con los puntos estrella hasta que no se tengan absolutamente resueltos los puntos obligatorios, incluyendo la corroboración de ausencia de *memory leaks* y de accesos inválidos a memoria.

Ejercicio ⋆ 1 *Diferentes formas de aplanar un BST (nivel de dificultad: medio)*

Pensar y documentar cómo las diferentes formas de implementar `bst_to_list` afecta el orden de los nodos en la lista resultante.

Pensar también cómo impacta en la carga de un `bst` desde archivo si las palabras y sus definiciones están almacenadas en el archivo en orden alfabético. Implementar un `bst_to_list` tal que al guardar el árbol a disco usando este método, y luego cargarlo con el método ya dado `dict_from_file`, el árbol resultado esté balanceado (tan balanceado como estaba antes de aplanarlo).

Notar que este punto estrella no apunta a que implementen árboles balanceados, sino que se refiere sólo al guardado y a la carga de un árbol desde archivo.

Tampoco se deberían hacer modificaciones a ningún TAD que no sea el `bst`.

Ejercicio ⋆ 2 *Versión imperativa de los algoritmos (nivel de dificultad: alto)*

Proveer en un archivo `bst.c` separado, la implentación imperativa (es decir, la versión iterativa de los algoritmos) de todos los métodos del `bst`.

Notar que van a necesitar definir e implementar un TAD auxiliar `stack` para poder completar las implementaciones iterativas.

4. Entrega y evaluación

- Fecha de entrega del código:
  - Desde el Lunes 12 de Mayo a las 00:00 hrs hasta las 23:59 hrs del mismo día.
  - Obligatorio hacerlo a través de Jaime, habrá una tarea específica para subir el código (separada de la tarea para probar los proyectos).
  - Se permitirá una sola entrega por grupo, así que asegurarse de subirla cuando esté lo suficientemente probada.
- El parcialito se rinde el Martes 13 de Mayo a las 14. Llegar a horario pero no ocupar las máquinas del lab. No hace falta traer ningún archivo ni pendrive.

5. Recordar

- Comentar e indentar el código apropiadamente, siguiendo el estilo de código ya provisto por la cátedra.
- Todo el código tiene que usar la librería estándar de C, y no se puede usar extensiones GNU de la misma.
- El programa resultante **no** debe tener *memory leaks* **ni** accesos (read o write) inválidos a la memoria.