Proyecto 3 - Árboles Binarios de Búsqueda

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Leonardo Rodríguez, Diego Dubois, Santiago Ávalos, Gonzalo Peralta, Jorge Rafael.

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 2 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**).

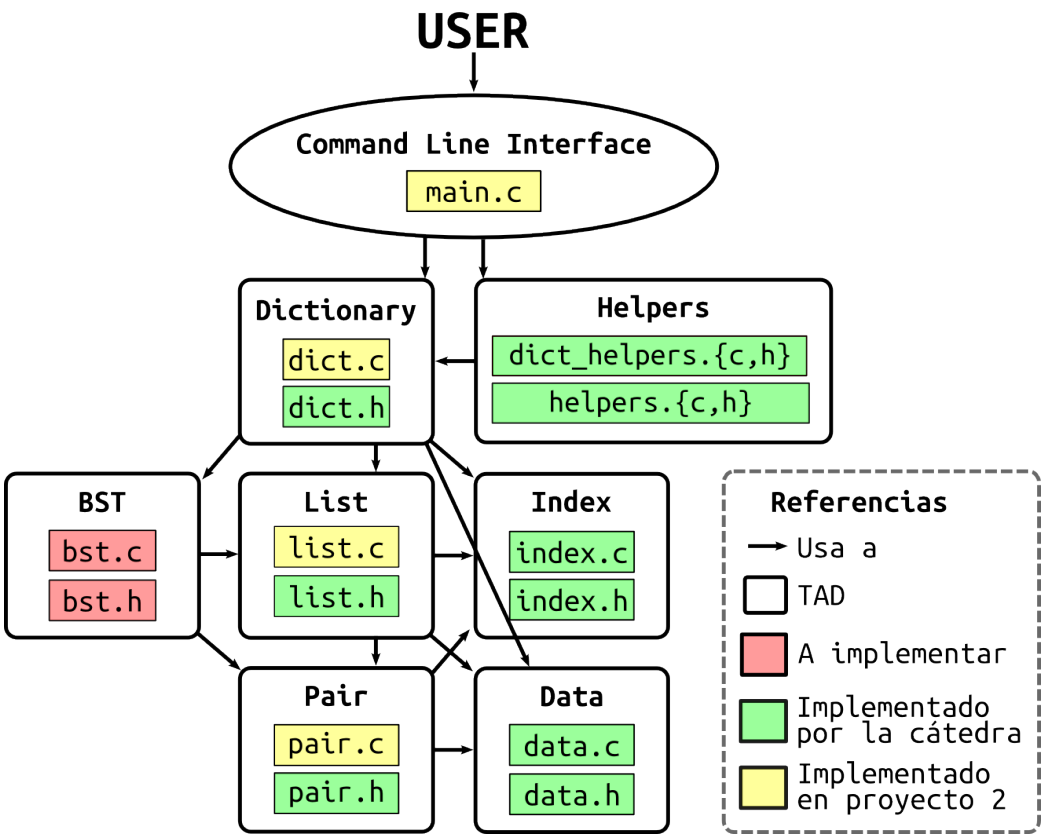

Figura 1: Diagrama de TADs

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:

```
#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);
/*
 * Returns 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.
 *
 */

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);
/*
 * Returns the number of elements in the given 'bst'.
 * This method has a linear order complexity.
 */

bool bst_is_equal(bst_t bst, bst_t other);
/*
 * Returns 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).
 */

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

```
bst_t bst_add(bst_t bst, index_t index, data_t data);
/*
 * Returns 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: 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 ones 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);
/*
 * Returns 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).
 *
 *
 * 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);
/*
 * Returns 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 an exact copy of 'bst'.
 * In particular, bst_is_equal(result, bst) holds.
 */

list_t bst_to_list(bst_t bst, list_t list);
/*
 * This function appends to the given 'list' a copy of all the
 * elements of the 'bst' in ascending order.
 *
 * The result's length is equal to bst_length(bst)
 * plus list_length(list).
 *
 * In particular, a call to bst_to_list(bst, list_empty()) will return
 * a sorted list with all and only the elements of the 'bst'.
```

```
 */

/* All the functions assume that the input pointers are *valid*: that
 * is, depending on the implementation, the pointers must point either
 * to 'NULL' or to a well-formed structure. It is guaranteed that the
 * pointers remain valid after the execution of the functions. */

#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).

Agregamos 2 nuevos requisitos para aprobar este proyecto:

- Proveer un `Makefile` tal que:
  - El comando `make` compila el projecto.
  - El comando `make clean`) borra los archivos objeto y el ejecutable.
  - El comando `make valgrind` ejecuta el diccionario usando valgrind con los flags usuales.
- El archivo `bst.c` debe pasar el test de estilo de código de Jaime.

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`.

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).

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 correspondiente 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`.

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

Siguiendo con las figuras geométricas del proyecto anterior, la "memoria" de un BST se podría graficar de la siguiente forma:

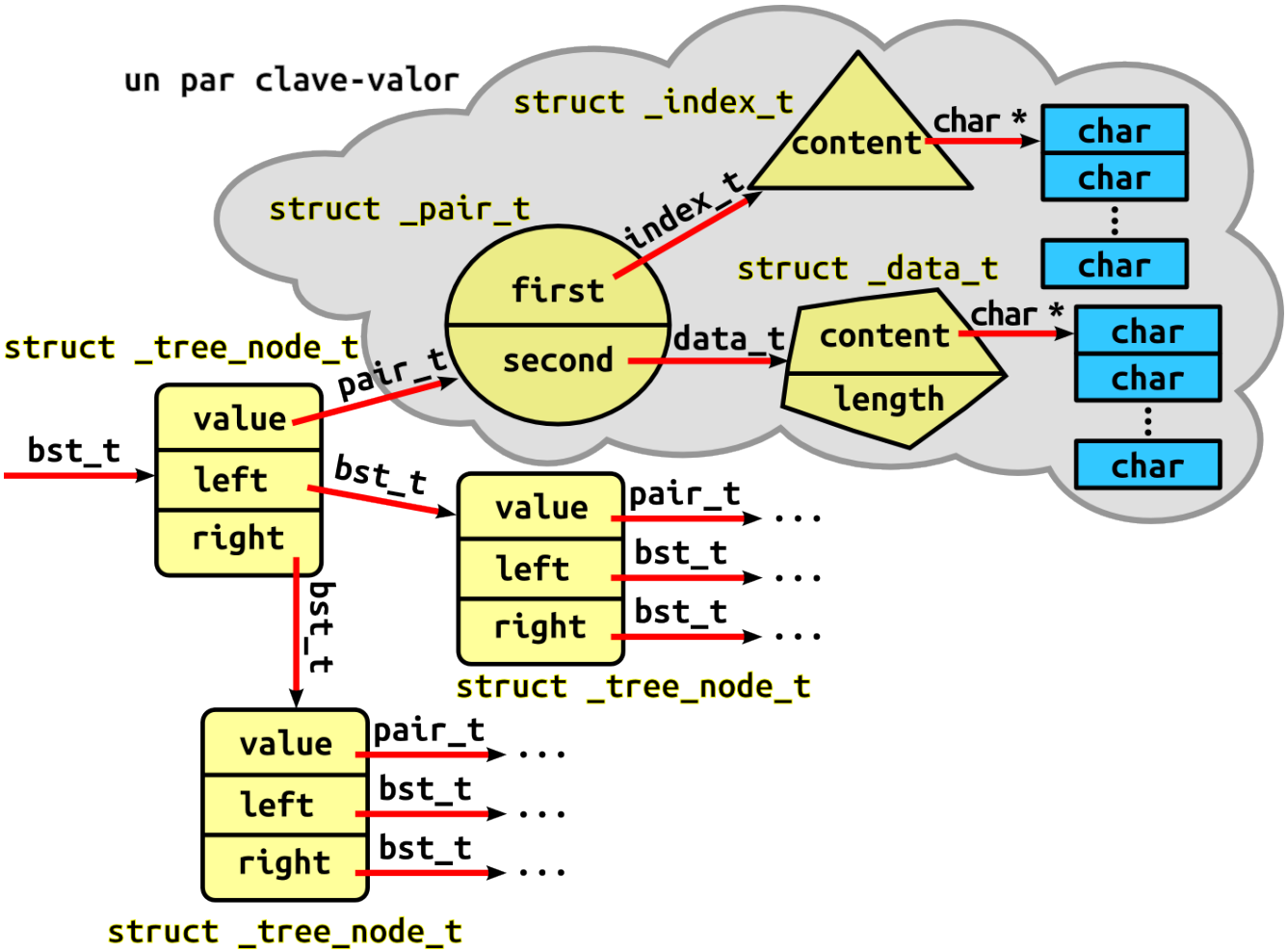

Figura 2: El árbol binario de búsqueda de pares clave-valor

Como antes, las cajas rectangulares con miembros **value**, **left** y **right** son nodos del árbol, y la nube completa representa un par clave-valor en su totalidad.

Un poco de terminología: Una **hoja** 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
```

3. Puntos ⋆

Hacer los puntos estrellas en carpetas separadas.

Ejercicio ⋆ 1 *Preservación de la estructura de árbol.*

Considerar el siguiente programa:

```c
#include "dict.h"
#include "dict_helpers.h"
#include <assert.h>

#define PATH_TO_DICT "input/small.dic"
#define PATH_TO_COPY "input/copy.dic"

int main(void) {
    dict_t dict = NULL;
    dict_t copy = NULL;
    dict = dict_from_file(PATH_TO_DICT);
    dict_to_file(dict, PATH_TO_COPY);
    copy = dict_from_file(PATH_TO_COPY);
    assert(dict_is_equal(dict, copy));
    return (0);
}
```

¿Por qué razón falla la aserción? En un archivo `bst.c` separado modificar únicamente la implementación de `bst_to_list` para corregir ese problema (la especificación de la nueva `bst_to_list` puede cambiar).

¿Qué sucede cuando se llama a `dict_from_file` con un archivo que tenga las palabras ordenadas? ¿Cómo queda la estructura del árbol? ¿Afecta eso en la eficiencia de los algoritmos?

Adjuntar un archivo .txt con las respuestas.

Ejercicio ⋆ 2 *Versión no recursiva de los algoritmos (difícil)*

Proveer en un archivo `bst.c` separado, la implementación imperativa 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:
  - Hasta el Martes 19 de Mayo a las 23:59 hs.
  - Obligatorio hacerlo a través de Jaime, habrá una tarea específica para subir el código (separada de la tarea para probar los proyectos).
- El parcialito se rinde el Martes 19 de Mayo a las 14hs.