Proyecto 3
Árboles Binarios de Búsqueda

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Natalia Bidart, Matías Bordese, Diego Dubois,
Leonardo Rodríguez, David Arch, Maximiliano Bustos.

Objetivo

El objetivo de este proyecto es extender el código del proyecto 2 de manera tal que el TAD `dict` esté implementado usando árboles binarios de búsqueda como su contenedor interno y oculto.

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

- Árbol Binario de Búsqueda (en inglés, *Binary Search Tree*[1]). De ahora en más, nos referiremos a éstos como **BST**.

Notar que este proyecto es análogo a aquel dado en Algoritmos y Estructuras de Datos I (el proyecto al que hacemos referencia es este, se recomienda su repaso antes de hacer el presente, especialmente las definiciones recursivas de los algoritmos de agregar, buscar y borrar en árboles).

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**). Notar que para este proyecto no se les entrega ningún archivo, con lo cual se tendrán que crear desde cero los módulos a desarrollar.

[1] Ver en wikipedia

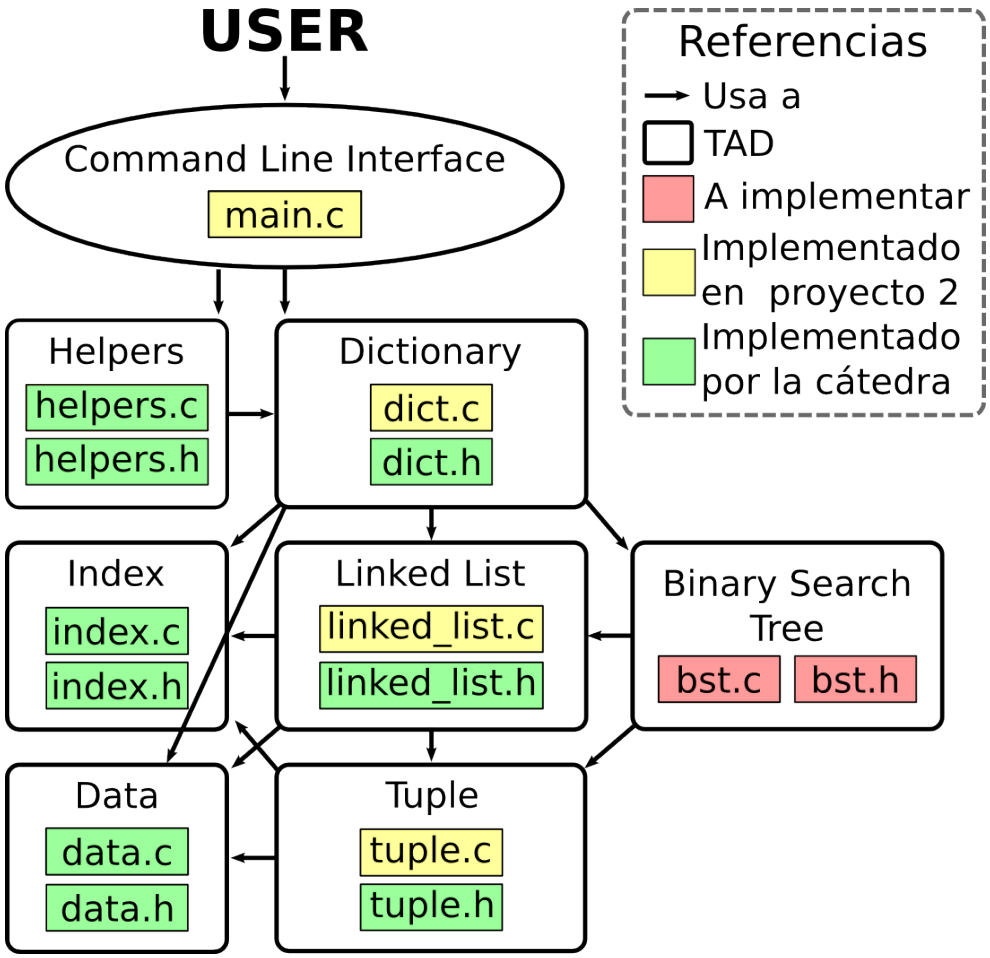


Figura 1: Diagrama de TADs

Las tareas concretas 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 "data.h"
#include "index.h"
#include "linked_list.h"
```

```
typedef struct _bst_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 call are freed.
 *
 * POST: the result will not be NULL, and bst_length(result) will be 0.
 */

bst_t bst_destroy(bst_t bst);
/*
 * Free the resources allocated for the given 'bst', and set it to NULL.
 *
 * PRE: 'bst' must not be NULL.
 */

bst_t bst_add(bst_t bst, index_t index, data_t data);
/*
 * Return a BST equals to 'bst' with the ('index', 'data') added to it.
 *
 * The given 'index' and 'data' are inserted in the BST, so do not destroy
 * them.
 *
 * PRE: all 'bst', 'index' and 'data' must be not NULL. Also, the pair
 * ('index', 'data') must not exist in the given BST.
 *
 * POST: the length of the result will be the same as the length of 'bst'
 * plus one. The elements of the result will be the same as the one in 'bst'
 * with the new pair ('index', 'data') added acordingly (see:
 * http://en.wikipedia.org/wiki/Binary_search_tree
 * for specifications about behavior).
 *
 */

bst_t bst_remove(bst_t bst, index_t index);
/*
 * Return a BST equals to 'bst' with the (index, <data>) tuple
 * occurrence removed.
 *
 * Please note that 'index' may not be in the BST (thus an unchanged
 * BST is returned).
```

```
 *
 * PRE: both 'bst' and 'index' must not be NULL.
 *
 * POST: the length of the result will be the same as the length of 'bst'
 * minus one if 'index' existed in 'bst'. The elements of the result will be
 * the same as the one 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 call are freed.
 *
 * POST: the result will not be NULL and it will be an exact copy of 'bst'.
 * In particular, bst_length(result) will be equal to the length of 'bst'.
 */

unsigned int bst_length(bst_t bst);
/*
 * Return the number of elements in the given 'bst'.
 * Constant order complexity.
 *
 * PRE: 'bst' must not be NULL.
 */

data_t bst_search(bst_t bst, index_t index);
/*
 * Return the data associated to the given 'index' in BST,
 * or NULL if the given 'index' is not in 'bst'.
 *
 * The caller must NOT free the resources allocated for the result when done
 * using it.
 *
 * PRE: both 'bst' and 'index' must not be NULL.
 */

linked_list_t bst_to_linked_list(bst_t bst);
/*
 * Return a sequence representation of 'bst'.
 *
 * PRE: 'bst' must not be NULL.
 *
 * POST: the result will not be NULL, and the result's length will be the same
```

```
 * as the given BST's length. Every value (index, data) in the BST will be in
 * the returned list, and the BST order should be preserved.
 *
 * In other words, the resulting list has to be ascendantly ordered.
 *
 */

void bst_dump(bst_t bst, FILE *fd);
/*
 * Dump the BST using its string representation to the given
 * file descriptor.
 *
 * PRE: 'bst' must not be NULL, and 'fd' must be a valid file descriptor.
 */

#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.
- 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.
- Proveer un `Makefile` tal que se pueda compilar el proyecto al correr el comando `make`. Se recomienda tomar el `Makefile` dado en el proyecto anterior y hacer las adaptaciones que hicieran falta.

Sobre el TAD `bst`

El TAD `bst` a implementar en C es análogo al árbol binario de búsqueda visto en el teórico. Como guía se sugieren la siguientes estructuras:

```
#include "bst.h"

/*
 * Tree implementation (private)
 */
```

```
#define EMPTY_TREE ((tree_t) NULL)

typedef struct _tree_t *tree_t;

struct _tree_t {
    tuple_t value;
    tree_t left;
    tree_t right;
};

/*
 * Binary Search Tree implementation
 */

struct _bst_t {
    tree_t root;
    unsigned int count;
};
```

Es decir, que la estructura `struct _bst_t` deberá tener un puntero a la raíz del árbol, y un contador de nodos.

La estructura `struct _tree_t` representa al árbol en sí, donde cada árbol consta de su nodo (que almacena el valor - un `tuple_t` en este caso), y de sus árboles izquierdo y derecho. Notar que esta estructura es **recursiva**.

Luego, la implementación de los siguientes métodos:

- `bst_destroy`
- `bst_add`
- `bst_remove`
- `bst_destroy`
- `bst_copy`
- `bst_search`
- `bst_to_linked_list`

va a ser en función de las implementaciones **recursivas** de los siguientes métodos que operan sobre `tree_t` (respectivamente):

- `tree_t tree_destroy(tree_t tree);`

- `tree_t tree_add(tree_t tree, tuple_t value);`
- `tree_t tree_remove(tree_t tree, index_t index);`
- `tree_t tree_copy(tree_t tree);`
- `data_t tree_search(tree_t tree, index_t index);`
- `linked_list_t tree_flatten(tree_t tree, linked_list_t list);` [2]

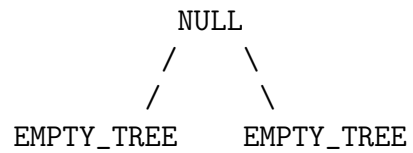
Es muy importante destacar que `bst_empty` no depende de ningún método que opere sobre `tree_t`, pero aún así es necesario proveer un constructor de `tree_t` que será usado por alguno/s de lo/s método/s listados arriba:

```
tree_t tree_leaf(tuple_t value);
```

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

- Su nodo es una `tuple_t` no vacía.
- Su árbol izquierdo es vacío.
- Su árbol derecho es vacío.

Por ende, lo siguiente es un árbol **inválido**:



Ejercicio ⋆ 1 *Proveer además la implentación imperativa de todos los métodos del* `bst` *y del* `tree`.

Ejercicio ⋆ 2 *Pensar y documentar cómo las diferentes formas de implementar* `tree_flatten` *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_linked_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.*

[2] Ver en wikipedia. Usar listas enlazadas para aplanar el árbol (dejar esta implementación para el final).

Recordar

- Entrega y evaluación: Martes 22 de Mayo.
- 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.