Proyecto 2 - Tipos Abstractos de Datos

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 código de la primer parte del proyecto 2, para proveer una implementación en C de los siguientes TADs:

- Pares de valores http://en.wikipedia.org/wiki/Tuples
  - Es una 2-upla que tiene como primer elemento un `index_t` y como segundo elemento un `data_t`.
- Listas enlazadas: http://en.wikipedia.org/wiki/Linked_list
  - Es una secuencia de los pares de valores descriptos en el punto anterior.

2. Instrucciones

En la figura 2 se muestra un diagrama análogo al presentado en la primer parte del proyecto 2, con la diferencia de que ahora los TADs a implementar son las listas enlazadas y los pares de clave-valor (ver los módulos resaltados en **rojo**).

Las tareas de esta segunda parte del proyecto 2 son las siguientes:

- Crear archivos `list.c` y `pair.c` en donde se provee las implementaciones de los tipos de datos lista enlazada y par de clave y valor, respectivamente, cumpliendo al pie de la letra las especificaciones dadas en los archivos de cabecera (ya provistas por la cátedra en el proyecto anterior) `list.h` y `pair.h`.

  Ojo que `list.h` tiene cambios respecto del año pasado.
- Las implementaciones de los tipos mencionados arriba debe ser oculta. Es decir, deben usar la técnica de punteros a estructuras cuando implementen los TADs requeridos, tal como se hizo en la primer parte con el TAD `dict`.
- El resto de los archivos (todos los `.h` y los `.c` dados por la cátedra o implementados en la parte 1) no deben sufrir cambios (excepto que necesiten arreglar algún bug). Es decir, todos los demás TADs deben quedar igual que en el proyecto anterior (inclusive la interfaz con el usuario).
- Testear el nuevo ejecutable, confirmando que siempre estén usando código objeto enteramente producido por ustedes. Para ello, antes de compilar, asegurarse de lo siguiente:
  - que dentro del archivo `Makefile`, la variable `LIBS` no tiene ningún `.o` especificado
  - que todos los archivos `.o` fueron borrados del directorio, corriendo el comando (dentro del directorio del proyecto):

    ```
    $ rm *.o
    ```

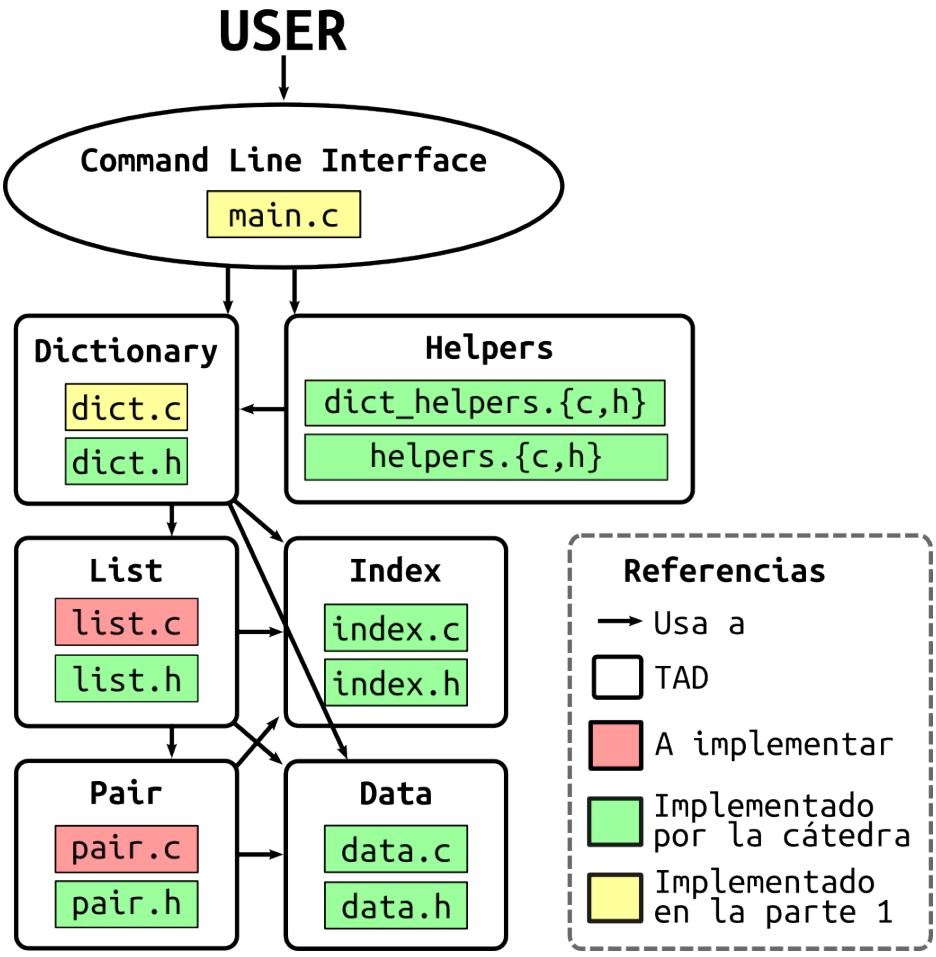

Figura 1: Diagrama de TADs

- El programa resultante debe estar libre de "memory leaks". Usar `valgrind` para confirmar tal situación, como en la primer parte, corriendo el comando:

  ```
  $ valgrind --leak-check=full --show-reachable=yes ./dictionary
  ```

2.1. Sobre el TAD `pair`

El TAD `pair` a implementar en C es equivalente a la siguiente definición en pseudo-código de una 2-upla que almacena como primer valor algo de tipo `index_t`, y como segundo valor algo de tipo `data_t`:

```
(index_t, data_t)
```

Es decir, que la estructura `struct _pair_t` deberá ser elegida de manera tal que se pueda almacenar dos miembros: un `index_t` y un `data_t`.

Para manipular los index y los data hay que usar todos los métodos documentados en los archivos `index.h` y `data.h`.

2.2. Sobre el TAD `list`

El TAD `list` a implementar en C es equivalente a la siguiente definición en pseudo-código (pensar en Haskell, lista de pares de clave-valor):

`[(index_t, data_t)]`

que dada la definición del tipo nuevo `pair_t` de la subsección anterior, se puede escribir de manera equivalente:

`[pair_t]`

Es decir, que la estructura `struct _list_t` deberá ser elegida de manera tal que se puedan almacenar nodos que contengan un `pair_t` adentro.

Para definir la estructura del TAD `list`, seguir el apunte del teórico sobre listas enlazadas y mapear idénticamente a C los dos tipos definidos como `node` y `list`. Notar que el tipo `node` es un detalle de implementación de la lista, y no debe ser expuesto bajo ningún concepto en el archivo de cabecera `list.h`.

Si uno piensa cada nodo como una figura geométrica, y cada puntero como una flecha, la lista de pares que tienen que implementar puede graficarse de la siguiente manera:

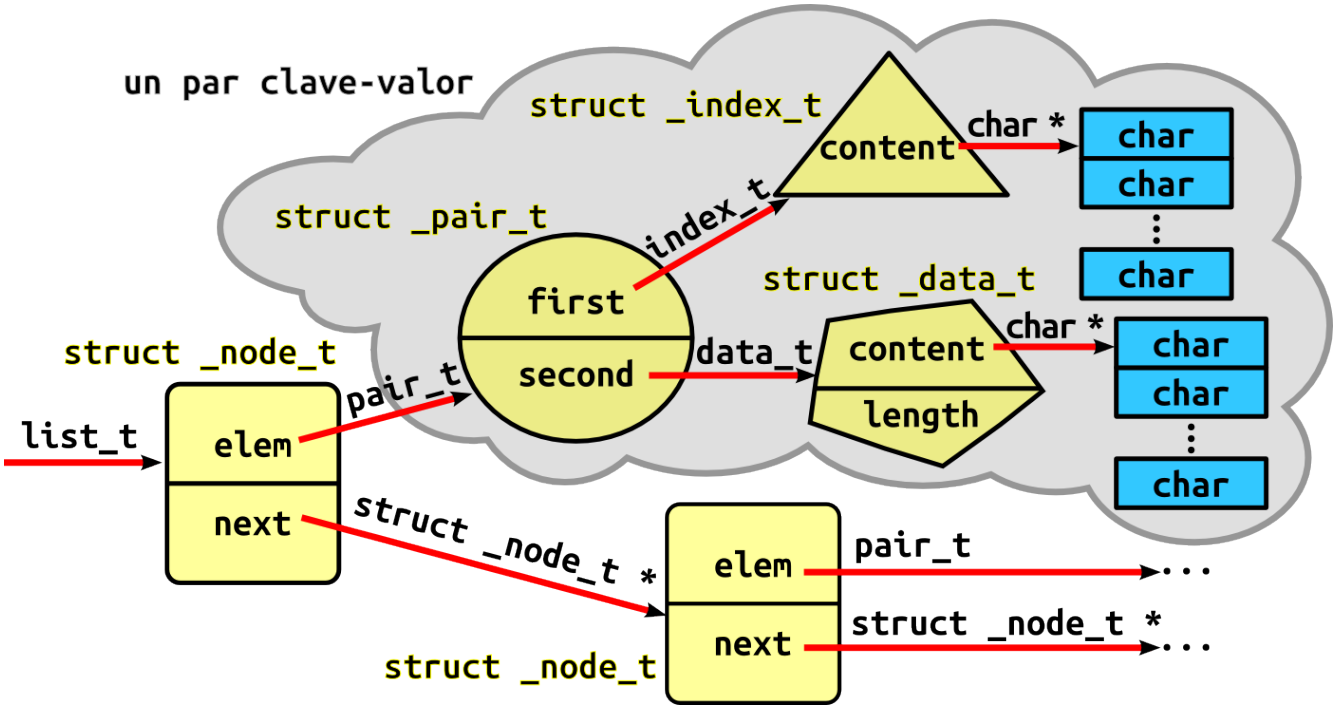

Figura 2: La lista enlazada de pares clave-valor

Todas las figuras en amarillo son estructuras de C, y todas las flechas rojas son punteros (direcciones de memoria).

Las cajas rectangulares con miembros **elem** y **next** son nodos de la lista, y la nube completa representa un par clave-valor completo. Entonces, el miembro `next` de un nodo apunta a un par clave-valor, y el miembro `next` apunta al nodo siguiente de la lista. Si un nodo es el último de la lista, `next` es un puntero nulo (es decir, vale `NULL`).

Notar que la representación de la lista en sí es **únicamente** un puntero al primer nodo, **y nada más**.

Dentro del par clave-valor, el círculo es la estructura del TAD `pair`, el pentágono es un valor (representa al TAD `data`) y el triángulo una clave (representa al TAD `index`).

Notar además que cada index y data tienen, a su vez, memoria asociada para almacenar sus contenidos (son las cajitas rectangulares celestes).

Puntos $\star$

Recomendación importante: 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 $\star$ 1 *Largo de diccionario de orden constante*

La lista presentada en este proyecto provee una función para consultar el largo de la misma, pero es de orden lineal. Es decir, para N elementos, calcular el largo requiere recorrer esos N elementos.

Modificar **únicamente** el TAD `dict` para que la consulta del largo del diccionario tenga orden constante en vez de lineal.

Ejercicio $\star$ 2 *Lista ordenada*

Se puede cambiar la implementación del TAD `list` de manera tal que en todo momento la lista permanezca ordenada?

Si sí, hacer los cambios necesarios para que esto ocurra EN ARCHIVOS SEPARADOS de la resolución base del proyecto. Notar que esto **no** significa ordenar la lista completa cada vez que se la modifica.

Este cambio produce una mejora en la complejidad de la búsqueda, inserción y/o borrado de palabras en el diccionario?

Ayuda Dependiendo de lo que piensen en este punto, pueden que necesiten agregar un método nuevo en el TAD `list`, y por ende modificar alguna llamada a este TAD desde `dict`.

En tal caso, elegir `list_add` como nombre del nuevo método, y borrar del código el método `list_append`. Habrá una tarea separada en Jaime para subir este estrella.

3. Entrega y evaluación

- Fecha de entrega del código:
  - Desde el Lunes 28 de Abril 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 29 de Abril a las 14. Llegar a horario pero no ocupar las máquinas del lab.

4. 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.
- Recordar que la traducción de pseudocódigo al lenguaje C **no** es directa. En particular, tener en cuenta que lo que se llama `tuple` en el teórico, en C es un `struct`.