Tarea 1
Introducción a los Tipos Abstractos de Datos
Curso 2020

Índice

1. Introducción y objetivos

Esta tarea tiene los siguientes objetivos específicos:

- Familiarizarse con el ambiente de trabajo del laboratorio.
- Realizar un primer acercamiento al lenguaje C.
- Comenzar a familiarizarse con la división del código de programas compuestos por módulos distribuidos en múltiples archivos.
- Trabajar con los TADs *Lista*, *Pila*, *Cola* y *Mapping*. Concretamente, usar el TAD *Lista* (su tipo y operaciones) para implementar los otros TADs.

En esta tarea se puede trabajar únicamente en forma individual. La tarea es **eliminatoria**.

2. Materiales

Los archivos para la tarea se extraen de *MaterialesTarea1.tar.gz*. Para conocer la estructura de los directorios ver la Sección **Materiales** de Funcionamiento y Reglamento del Laboratorio.

En esta tarea los archivos en el directorio `include` son **utils.h**, **listaInfo.h**, **pilaInfo.h**, **colaInfo.h** y **mapping.h**.

En el directorio `src` se incluye ya implementado **listaInfo.cpp**. Además se incluye el archivo **ejemploPilaInfo.png** que contiene la imagen de una posible implementación del tipo y las operaciones que se piden del módulo **pilaInfo**. El contenido de este archivo **puede** copiarse en el archivo **pilaInfo.cpp**.

3. ¿Qué se pide?

Ver la Sección **Desarrollo** de Funcionamiento y Reglamento del Laboratorio.

En esta tarea solo se deben implementar los archivos **pilaInfo.cpp**, **colaInfo.cpp** y **mapping.cpp** ya que el archivo **listaInfo.cpp** ya está implementado y que **utils.h** consiste solo en definiciones de constantes y tipos.

La implementación de los tres archivos se debe hacer usando las entidades (tipos y operaciones) declaradas en **listaInfo.h**.

Para implementar `pilaInfo.cpp` se puede usar el código que está en `ejemploPilaInfo.png`.

La generación del ejecutable se hace compilando los archivos implementados **listaInfo.cpp**, **pilaInfo.cpp**, **colaInfo.cpp**, **mapping.cpp** y **principal.cpp** y enlazando los archivos objeto obtenidos:

```
$ g++  -Wall -Werror -Iinclude -g  -c principal.cpp -o obj/principal.o
$ g++  -Wall -Werror -Iinclude -g  -c src/listaInfo.cpp -o obj/listaInfo.o
$ g++  -Wall -Werror -Iinclude -g  -c src/pilaInfo.cpp -o obj/pilaInfo.o
$ g++  -Wall -Werror -Iinclude -g  -c src/colaInfo.cpp -o obj/colaInfo.o
$ g++  -Wall -Werror -Iinclude -g  -c src/mapping.cpp -o obj/mapping.o
$ g++  -Wall -Werror -Iinclude -g  obj/principal.o obj/listaInfo.o obj/pilaInfo.o
   obj/colaInfo.o obj/mapping.o -o principal
```

El archivo **Makefile** (ver la Sección Makefile de Material Complementario) provee la regla **principal**, que es la predeterminada, para la compilación.

Por lo tanto los anteriores comandos pueden ser sustituidos por

```
$ make principal
```

o simplemente

```
$ make
```

Se sugiere crear los tres archivos a implementar e incluir todas las funciones requeridas con un código mínimo para que se pueda compilar (por ejemplo, `return NULL`). Para cada uno de los módulos implementar el struct y la función `crear` correspondiente y luego implementar cada función y correr el caso de prueba correspondiente, en el siguiente orden:

- `pilaInfo`: `esVaciaPila`, `apilar`, `cima`, `desapilar`.
- `colaInfo`: `esVaciaCola`, `encolar`, `frente`, `desencolar`.
- `mapping`: `esClave`, `asociar`, `valor`, `desasociar`.

Cuando se comprueba que hay un error es muy probable que sea causado por la última función implementada. Si para corregir ese error se modifica algo del código anteriormente verificado se debe volver a correr todos los casos en el mismo orden. Con este método los errores se detectan de manera temprana.

En el archivo **Makefile** se provee la regla **testing** que automatiza la ejecución y comparación de todos los casos. Se sugiere que se utilice después de correr uno por uno los casos, con el propósito de ratiifcar que la implementación es correcta para todos los casos.

Se sugiere que antes de entregar se haga una corrida para verificación en las máquinas de facultad, ya sea de manera presencial o remota.

4. Descripción de los módulos

En esta sección se describen los módulos que componen la tarea. Estos son *utils*, *listaInfo*, *pilaInfo*, *colaInfo*, *mapping* y *principal*.

Los módulos *listaInfo*, *pilaInfo*, *colaInfo* y *mapping* son TADs y el módulo *utils* contiene definiciones de constantes y tipos usados en los otros módulos.

El módulo *principal* es un intérprete de comandos utilizado para probar las funciones implementadas en los otros módulos. Consiste en un único archivo, `principal.cpp`. Para poder compilar este archivo de manera separada, es decir, sin tener aún las implementaciones de los otros módulos, cada uno de estos últimos se dividen en dos: especificación (`.h`) e implementación (`.cpp`).

En los archivos de especificación se declaran constantes, tipos y funciones (operaciones). Entonces si en `principal.cpp` se incluyen estos archivos mediante

```
#include "include/utils.h"
#include "include/pilaInfo.h"
#include "include/colaInfo.h"
#include "include/mapping.h"
```

se pueden usar las entidades ahí declaradas y el compilador puede comprobar que el código fuente de `principal.cpp` es sintácticamente correcto y entonces generar el código objeto correspondiente.

De la misma forma si en `pilaInfo.cpp`, `colaInfo.cpp` y `mapping.cpp` se incluye el archivo de especificación `listaInfo.h` se pueden usar las operaciones de *listaInfo* para implementar las operaciones del módulo y compilar de manera separada.

En los archivos de especificación de los TADs se declaran tipos que serán implementados en los archivos de implementación correspondientes. Por ejemplo en `listaInfo.h` se define el tipo `TLista`:

```
typedef struct repLista * TLista;
```

Con esto se sabe que los elementos de tipo `TLista` son punteros y por lo tanto una variable de este tipo ocupa una cantidad conocida de bytes (4 bytes u 8 bytes depenediendo de la arquitectura). De esta manera si en `principal.cpp` o en `pilaInfo.cpp` se declara una variable de este tipo el compilador conoce cuánto

espacio va a ocupar, lo que permite generar el código objeto.

El tipo `struct repLista` (la representación de `TLista`) es el tipo de los elementos a los que apuntan los elementes de tipo `TLista`. Este tipo queda declarado pero su definición debe implementarse en el archivo `listaInfo.cpp`. El ámbito de esa definición será solo el archivo `listaInfo.cpp`. Un intento de declarar una variable de tipo `struct repLista` en `principal.cpp` o en `pilaInfo.cpp` generará un error de compilación.

Los elementos de tipo `TLista` son punteros. Al definirlos se debe obtener memoria de manera dinámica. En `C++` esto se hace con el operador `new`:

```
TLista lista = new repLista;
```

Con `Tista` se puede implementar otros TADs. Por ejemplo la representación de *TPila* se puede implementar de esta manera:

```
struct repPila {
  TLista lst;
};
```

o sea, con un registro con un único campo de tipo `TLista`. Si `p` es un elemento de tipo `TPila`, esto es, un puntero a un elemento de tipo `repPila`, mediante el operador de indirección `*` se accede a lo apuntado por `p`. Por lo tanto con `(*p).lst` se accede a la lista con la que se implementa la pila. Una manera alternativa de acceder a la lista es `p->lst`.

En el resto de esta sección se detallan los módulos *utils*, *listaInfo*, *pilaInfo*, *colaInfo* y *mapping*. q

4.1. Módulo *utils*

Este módulo consiste solo en el archivo de definiciones `utils.h`.
Se define:

- La constante `MAX` que es la máxima cantidad de elementos que puede contener una colección de tipo *listaInfo*, *pilaInfo*, *colaInfo* o *mapping*.
- El tipo `nat` que modela los naurales (enteros no negativos).
- El tipo `info_t` que modela un tipo genérico, que en esta caso está compuesto por un componente natural (`nat`) y otro real (`double`).

4.2. Módulo *lista*

El TAD *Lista* modela las secuencias de elementos que pueden ser accedidos mediante la posición que ocupan. En esta tarea los elementos son de tipo `info_t`.
Un posible conjunto de operaciones del TAD *Lista* es:

Crear que devuelve una lista sin elementos.

Longitud que devuelve la cantidad de elementos de la lista.

Insertar que agrega un elemento a la lista en una posición determinada. Los elementos que estaban en posiciones mayores quedan desplazados. O sea, si la lista consistía en la secuencia $(e_1, e_2, \cdots, e_i, \cdots, e_n)$ y se agrega el elemento e en la posición i el resultado consiste en la secuencia $(e_1, e_2, \cdots, e, e_i, \cdots, e_n)$. El elemento e_i queda desplazado a la posición $i+1$, $\cdots$, el elemento e_n queda desplazado a la posición $n+1$.

La posición en la que se inserta el elemento debe estar entre 1 y $n+1$. Si es $n+1$ el nuevo elemento queda como último y ninguno es desplazado.

Remover que remueve de la lista el elemento que está en una posición determinada. Los elementos que estaban en posiciones mayores quedan desplazados. O sea, si la pila consistía en la secuencia $(e_1, e_2, \cdots, e_i, e_{i+1}, \cdots, e_n)$, con $n > 0$, el resultado de remover el elemento que está en la posición i consiste en la secuencia $(e_1, e_2, \cdots, e_{i+1}, \cdots, e_n)$. El elemento e_{i+1} queda desplazado a la posición i, $\cdots$, el elemento e_n queda desplazado a la posición $n-1$.

La posición de la que se remueve el elemento debe estar entre 1 y n.

Buscar que devuelve la posición en la que hay un elemento que cumple una determiada propiedad. Por ejemplo la posición en la que haya un elemento cuyo componente natural sea igual a un determinado parámetro.

Si ningún elemento cumple la propiedad se puede devolver una posición que no esté en el rango de las posiciones de la lista.

Info que devuelve el elemento que está en una posición determinada. O sea, si la lista consiste en la secuencia $(e_1, e_2, \cdots, e_i, \cdots, e_n)$, con $n > 0$, la operación `Info` con parámetreo i devuelve el elemento e_i.

La posición de la que se obtien elemento debe estar entre 1 y n.

En esta tarea se establece la restricción de que la lista debe ser acotada, o sea que la cantidad máxima de elementos que puede contener está determinada por la constante `MAX`.
Como consecuencia, la operacion `Insertar` no tiene efecto si la lista está llena.

4.3. Módulo *pila*

El TAD *Pila* modela las secuencias de elementos en las que el último elemento ingresado es el único al que se tiene acceso. En esta tarea los elementos son de tipo `info_t`.
Las operaciones estándar del TAD *Pila* son:

Crear que devuelve una pila sin elementos.

EsVacia que indica si la pila tiene elementos o no.

Apilar que agrega un elemento a la pila. O sea, si la pila consistía en la secuencia $(e_1, e_2, \cdots, e_n)$ y se agrega el elemento e el resultado consiste en la secuencia $(e_1, e_2, \cdots, e_n, e)$.

Cima que devuelve el último elemento agregado en la pila, que no puede ser vacía. O sea, si la pila consiste en la secuencia $(e_1, e_2, \cdots, e_n)$, con $n > 0$, devuelve el elemento e_n.

Desapilar que remueve de la pila, que no puede ser vacía, el último elemento agregado en la pila. O sea, si la pila consistía en la secuencia $(e_1, e_2, \cdots, e_{n-1}, e_n)$, con $n > 0$, el resultado consiste en la secuencia $(e_1, e_2, \cdots, e_{n-1})$.

En esta tarea se establece la restricción de que la pila debe ser acotada, o sea que la cantidad máxima de elementos que puede contener está determinada por la constante `MAX`.
Como consecuencia, la operacion `Apilar` no tiene efecto si la pila está llena.

4.4. Módulo *cola*

El TAD *Cola* modela las secuencias de elementos en las que el primer elemento ingresado de los que aún están en ellas es el único al que se tiene acceso. En esta tarea los elementos son de tipo `info_t`.
Las operaciones estándar del TAD *Cola* son:

Crear que devuelve una cola sin elementos.

EsVacia que indica si la cola tiene elementos o no.

Encolar que agrega un elemento a la cola. O sea, si la cola consistía en la secuencia $(e_1, e_2, \cdots, e_n)$ y se agrega el elemento e el resultado consiste en la secuencia $(e_1, e_2, \cdots, e_n, e)$.

Frente que devuelve el primer elemento agregado en la cola, que no puede ser vacía. O sea, si la cola consiste en la secuencia $(e_1, e_2, \cdots, e_n)$, con $n > 0$, devuelve el elemento e_1.

Desencolar que remueve de la cola, que no puede ser vacía, el primer elemento agregado en la cola. O sea, si la cola consistía en la secuencia $(e_1, e_2, \cdots, e_{n-1}, e_n)$, con $n > 0$, el resultado consiste en la secuencia $(e_2, \cdots, e_n)$.

En esta tarea se establece la restricción de que la cola debe ser acotada, o sea que la cantidad máxima de elementos que puede contener está determinada por la constante `MAX`.
Como consecuencia, la operacion `Encolar` no tiene efecto si la cola está llena.

4.5. Módulo *mapping*

El TAD *Mapping* modela un conjunto de pares ordenados $(clave, valor)$ tal que el primer componente no se repite. O sea un mapping es un conjunto $\{(k_1, v_1), \ldots, (k_n, v_n)\}$, con $k_i \neq k_j$ si $i \neq j$. De esta manera cada clave del mapping tiene asociado un valor. Se debe notar que en general un mismo valor puede estar asociado a más de una clave. El conjunto al que pertenecen las claves y el conjunto al que pertenecen los valores asociados pueden ser el mismo o no. Por ejemplo las claves pueden ser ciudades y los valores asociados la cantidad de habitantes. En esta tarea las claves son naturales (tipo `nat`) y los valores son reales (tipo `double`).
Las operaciones estándar del TAD *Mapping* son:

Crear que devuelve en conjunto vacío, sin pares.

Asociar que asocia una clave k, que no pertenecía al conjunto de claves del mapping, con un valor v. La clave y el valor dados constituyen un nuevo par del conjunto de asociaciones del mapping. O sea, si el mapping consistía en el conjunto $\{(k_1, v_1), \cdots, (k_n, v_n)\}$ y se asocia el par (k, v), con k no perteneciente al conjunto $\{k_1, \ldots, k_n\}$, el resultado consiste en el conjunto $\{(k_1, v_1), \ldots, (k_n, v_n), (k, v)\}$.

EsClave que indica si una clave k pertenece al conjunto de claves del mapping. O sea, si el mapping consiste en el conjunto $\{(k_1, v_1), \cdots, (k_n, v_n)\}$ la clave k es clave del mapping si y solo si pertenece al conjunto $\{k_1, \ldots, k_n\}$.

Valor que devuelve el valor asociado a una clave k, que es clave del mapping. O sea, si el mapping consiste en el conjunto $\{(k_1, v_1), \cdots, (k_i, v_i), \ldots, (k_n, v_n)\}$ y $k = k_i$ el valor asociado a k es v_i.

Desasociar que remueve del mapping la asociación correspondiente a una clave k que es clave del mapping. O sea, si el mapping consistía en el conjunto $\{(k_1, v_1), \ldots, (k_{i-1}, v_{i-1}), (k_i, v_i), (k_{i+1}, v_{i+1}), \ldots, (k_n, v_n)\}$ y $k = k_i$ el resultado consiste en el conjunto $\{(k_1, v_1), \ldots, (k_{i-1}, v_{i-1}), (k_{i+1}, v_{i+1}), \ldots, (k_n, v_n)\}$.

En esta tarea se establece la restricción de que el mapping debe ser acotado, o sea que la cantidad máxima de elementos que puede contener está determinada por la constante `MAX`.
Como consecuencia, la operacion `Asociar` no tiene efecto si el mapping está lleno.

5. Entrega

Ver la Sección **Entrega** de Funcionamiento y Reglamento del Laboratorio.
Se debe entregar el siguiente archivo, que contiene los módulos implementados **pilaInfo.cpp**, **colaInfo.cpp** y **mapping.cpp**:

- **Entrega1.tar.gz**

Este archivo se obtiene al ejecutar la regla `entrega` del archivo *Makefile*:

```
$ make entrega
tar zcf Entrega1.tar.gz  -C src pilaInfo.cpp colaInfo.cpp mapping.cpp
-- El directorio y archivo a entregar es:
  /tarea1/Entrega1.tar.gz
```

Con esto se empaquetan los módulos implementados y se los comprime.
Nota: En la estructura del archivo de entrega los módulos implementados deben quedar en la raíz, NO en el directorio `cpp` (y por ese motivo se usa la opción `-C cpp` en el comando `tar`).

5.1. Plazos de entrega

El plazo para la entrega es el **miércoles 18 de marzo a las 14:00**.

5.2. Identificación de los archivos de las entregas

Cada uno de los archivos a entregar debe contener, en la primera línea del archivo, un comentario con el número de cédula del estudiante, sin el guión y sin dígito de verificación.

Ejemplo:

```
/* 1234567 */
```

5.3. Individualidad

Ver la Sección **Individualidad** de Funcionamiento y Reglamento del Laboratorio.