Tarea 5
Implementación de Tipos Abstractos de Datos - 2
Curso 2021

Índice

1. Introducción y objetivos

El objetivo de esta tarea es la implementación de **Tipos Abstractos de Datos (TADs)** y de nuevas estructuras cumpliendo requerimientos de tiempo de ejecución. Mantenemos los módulos de las tareas anteriores. Los detalles sobre los módulos de tareas anteriores se presentan en la Sección 4.1.

También trabajaremos sobre módulos nuevos, que implementan árboles balanceados, conjuntos y colecciones de cadenas. Los detalles de los módulos nuevos se presentan en la Sección 4.2.

Como corresponde al concepto de TAD no se puede usar la representación de un tipo fuera de su propio módulo.
Recordar que:

- **El correcto uso de la memoria será parte de la evaluación**.
- **En esta tarea se puede trabajar únicamente en forma individual**.

El resto del presente documento se organiza de la siguiente forma. En la Sección 2 se presenta una descripción de los materiales disponibles para realizar la presente tarea, y en la Sección 3 se detalla el trabajo a realizar. Luego, en la Sección 4 se explica mediante ejemplos el comportamiento esperado de algunas de las funciones a implementar.

2. Materiales

Los materiales para realizar esta tarea se extraen de *MaterialesTarea5.tar.gz*. Para conocer la estructura de los directorios ver la Sección **Materiales** de Estructura y funcionamiento del Laboratorio.

En esta tarea los archivos en el directorio `include` son los de la tarea anterior a los que se agregan **avl.h**, **conjunto.h** y **colCadenas.h**.

En el directorio `src` se incluyen ya implementados **utils.cpp** e **info.cpp**.

3. ¿Qué se pide?

Ver la Sección **Desarrollo** de Estructura y funcionamiento del Laboratorio.

En esta tarea se deben implementar los mismos archivos que en la tarea anterior y además los nuevos.
Se debe tener en cuenta que en `binario.h` y `usoTads.h` se agregaron nuevas funciones.
En algunas operaciones se piden requerimientos de tiempo de ejecución. Ese tiempo es el del peor caso, a menos que se especifique otra cosa de manera explícita.

Se sugiere implementar y probar los tipos y las funciones siguiendo el orden de los ejemplos que se encuentran en el directorio **test**.

En la primera línea de cada caso de prueba se indica que entidades se deben haber implementado. Después de implementar se compila y se prueba el caso.

Los siguientes comandos compilan, ejecutan el test (sin valgrind) y verifican que la salida generada por su programa es igual a la salida esperada:

```
$ make
$ ./principal < test/N.in > test/N.sal
$ diff test/N.out test/N.sal
```

Si la salida generada por su programa es igual a la salida esperada no se imprime nada.

Pero, para tener en cuenta el correcto uso de la memoria y ser notificado de posibles errores de memoria, se debe ejecutar además el siguiente comando:

```
$ valgrind --leak-check=full ./principal <test/NN.in> test/NN.sal
```

Se puede hacer una verificación rápida mediante reglas `t-CASO`, donde `CASO` se debe sustituir por el nombre de uno de los archivos `in` del directorio `test`:

```
$ make t-01
---- Bien ----
```

Si el resultado no es correcto se debería proceder al método descrito antes.

Al terminar, para confirmar, se compila y prueban todos los casos mediante la regla `testing` de `Makefile`:

```
$ make testing
```

Al ejecutar el comando `make testing` los casos se prueban usando `valgrind` y la utilidad `timeout`:

```
$ timeout 4 valgrind -q --leak-check=full ./principal < test/01.in > test/01.sal
```

`timeout` es una utilidad que establece un plazo para la ejecución de un proceso (4 en este ejemplo). Si la ejecución demorara más de ese plazo `timeout` terminaría ese proceso.

Los test que controlan eficiencia de tiempo se corren sin el uso de `valgrind`. En estos test mantener habilitado los `assert` puede enlentencer la ejecución por lo que se sugiere incluir la opción de compilación `-DNDEBUG`. En `Makefile` esto se consigue comentando la línea 50

```
#  CCFLAGS = -Wall -Werror -I$(HDIR) -g
```

y descomentando la línea 53

```
 CCFLAGS = -Wall -Werror -I$(HDIR) -g -DNDEBUG
```

Se debe recordar que la regla `testing` se dejará disponible sobre la fecha de entrega de la tarea.

Ver el material disponible sobre *Makefile*: Automatización de procedimientos.

Ver también el Método de trabajo sugerido.

4. Descripción de los módulos y algunas funciones

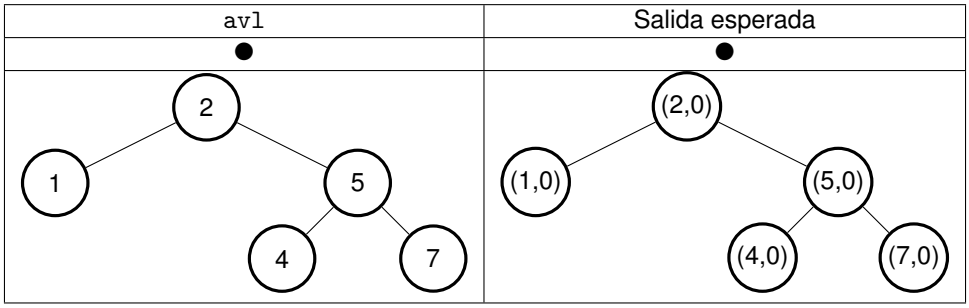
4.1. Módulos de tareas anteriores

Se mantienen los mismos módulos de la Tarea 4: *utils*, *info*, *cadena*, *usoTads*, *iterador*, *binario*, *pila* y *colaBinarios*.
En el módulo *binario* se agregó la operación *avlABinario*. Dado un `TAvl` devuelve un `TBinario`. Se incluye una prueba de tiempo para esta operación.

En el módulo *usoTads* se agregó la operación *interseccionDeConjuntos*.

4.1.1. avlABinario

Devuelve un `TBinario` con la misma forma y componentes naturales que los elementos del `TAvl` parámetro. El valor de los componentes reales es 0.



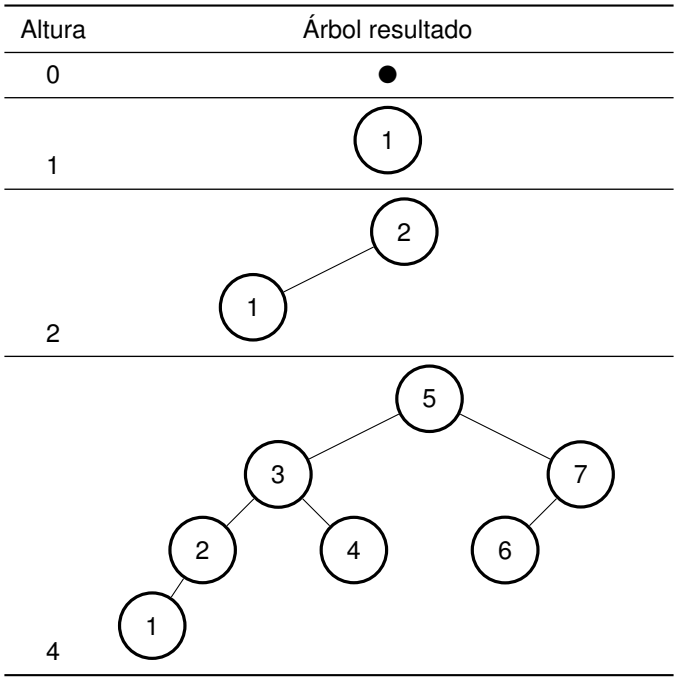
Se incluye una prueba de tiempo para esta operación.

4.2. Módulos nuevos

4.2.1. Módulo *avl*

En este módulo se debe implementar árboles blanceados (de tipo `TAvl`) de elementos de tipo `nat`.
Se muestran ejemplos de lo que se espera de la ejecución de algunas funciones.

minAvl Devuelve el `avl` que tiene la mínima cantidad de nodos para la altura pasada por parámetro. Los valores deben ser consecutivos y empezar en 1. En ningún nodo puede ocurrir que el subárbol derecho tenga más nodos que el subárbol izquierdo.



4.2.2. Módulo *conjunto*

En este módulo se debe implementar conjuntos de elementos de tipo `nat`. Se incluye una prueba de tiempo para esta módulo.

Para la implementación de las operaciones de este módulo es aceptable usar las operaciones que estaban en el módulo `usoTads`.

4.2.3. Módulo *colCadenas*

En este módulo se debe implementar colecciones de elementos de tipo `TCadenna`. La cantidad de cadenas está acotada por un parámetro pasado en la operación que crea la colección.

5. Entrega

Se mantienen las consideraciones reglamentarias y de procedimiento de la tarea anterior.
Se debe entregar el siguiente archivo, que contiene los módulos implementados `cadena.cpp`, `binario.cpp`, `iterador.cpp`, `pila.cpp`, `colaBinarios.cpp`, `avl.cpp`, `conjunto.cpp`, `colCadenas.cpp` y `usoTads.cpp`:

- **Entrega5.tar.gz**

Este archivo se obtiene mediante la ejecución del siguiente comando

```
$ tar zcvf Entrega5.tar.gz -C src cadena.cpp binario.cpp iterador.cpp pila.cpp colaBinarios.cpp avl.
```

Con esto se empaquetan los módulos implementados y se los comprime.
Lo mismo se obtiene al ejecutar la regla `entrega` del archivo *Makefile*:

```
$ make entrega
tar zcvf Entrega5.tar.gz -C src cadena.cpp binario.cpp iterador.cpp pila.cpp colaBinarios.cpp avl.cp
cadena.cpp
binario.cpp
iterador.cpp
pila.cpp
colaBinarios.cpp
avl.cpp
conjunto.cpp
colCadenas.cpp
usoTads.cpp
????/tarea5/Entrega5.tar.gz
```

Además se genera el archivo `sha.txt` que contiene una clave que identifica al archivo `Entrega5.tar.gz`.
Ese archivo también se puede generar mediante la utilidad `shasum`:

```
$ shasum -b Entrega5.tar.gz > sha.txt
```

5.1. Plazos de entrega

El plazo para la entrega es el **martes 8 de junio a las 18 horas**.

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 Reglamento del Laboratorio.

6. Tipos auxiliares

En cualquier módulo se pueden definir tipos auxiliares cuyo ámbito va desde su definición hasta el fin del archivo. En particular el tipo puede ser un `struct`.

```
  typedef struct {int entero; double real;} TIntDouble;
```

A diferencia de `Pascal`, en `C` un elemento de tipo `struct` se puede pasar como parámetro por valor y ser el resultado de una función:

```
TIntDouble incrementa_duplica(TIntDouble p) {
  p.entero = p.entero + 1;
  p.real = p.real * 2;
  return p;
}
```

El resultado del siguiente fragmento de código

```
 TIntDouble e_r;
 e_r.entero = 0;
 e_r.real = 1;

 TIntDouble res = incrementa_duplica(e_r);
 printf("e_r: (%d, %.2f); res: (%d, %.2f)\n", e_r.entero, e_r.real, res.↩
     entero, res.real);
```

es

```
e_r: (0, 1.00); res: (1, 2.00)
```