Tarea 6
Implementación de Estructuras de Datos
Curso 2021

Índice

1. Introducción

El objetivo de esta tarea es implementar nuevos TADs ya conocidos, y la aplicación de esos TADs y los desarrollados en las tareas anteriores.
En particular se implementará el TAD *Grafo* y algunos de sus algoritmos.

1.1. Descripción de grafo

De manera informal un grafo es un conjunto de elementos, llamados vértices, tal que entre cada par de vértices puede haber o no un vínculo, llamado arista. Si entre dos vértices hay una arista se dice que ambos son adyacentes. Por ejemplo los vértices pueden representar alumnos y hay una arista entre un par de alumnos si son compañeros en algún curso. En otro ejemplo los vértices representan ciudades y hay una arista entre dos ciudades si hay un tramo de carretera que las une directamente (o sea, sin necesidad de pasar por una ciudad intermedia). Las aristas pueden tener un valor (costo, peso) y si es así se dice que el grafo es ponderado. En el ejemplo de las ciudades el valor de una arista puede ser la longitud del tramo de carretera de las ciudades que une.
En esta tarea el grafo tiene N vértices, identificados del 1 al N y hasta M aristas, siendo N y M parámetros pasados al crear el grafo. Las aristas tienen un valor de tipo `double`. Dado un vértice v al conjunto de sus vértices adyacentes se les llama los vecinos de v.

Nota A cada par de vértices (v_1, v_j) se le puede hacer corresponder un número que lo identifica. Una forma, entre otras, es el número

$$(\text{mín}(v_i, v_j) - 1) \times N + (\text{máx}(v_i, v_j) - 1).$$

Se debe notar que al par (v_i, v_j) le corresponde el mismo número que al par (v_j, v_i). En cambio a cualquier otro par en el que al menos uno de los componentes es distinto a v_i y a v_j le corresponde un número diferente.

2. Materiales

Los materiales para realizar esta tarea se extraen de *MaterialesTarea6.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 **mapping.h**, **colaDePrioridad.h** y **grafo.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 `mapping.cpp`, `colaDePrioridad.cpp` y `grafo.cpp`.
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 qué 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 5: *utils*, *info*, *cadena*, *usoTads*, *iterador*, *binario*, *pila*, *colaBinarios*, *colCadenas*, *avl* y *conjunto*.

4.2. Módulos nuevos

4.2.1. Módulo *mapping*

En este módulo se debe implementar mappings de capacidad acotada de asociaciones de `nat` a `double`. Se incluye una prueba de tiempo para esta módulo.

4.2.2. Módulo *grafo*

En este módulo se debe imlementar los grafos que se describen en la introducción. Se incluye una prueba de tiempo para esta módulo.

4.2.3. Módulo *colaDePrioridad*

En este módulo se debe implementar colas de prioridad de elementos de tipo `nat` en un rango acotado. La prioridad de los elementos es de tipo `double`. Se incluye una prueba de tiempo para esta módulo.

5. Entrega

Se mantienen las consideraciones reglamentarias y de procedimiento de las tareas anteriores.
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`, `mapping.cpp`, `colaDePrioridad.cpp`, `grafo.cpp` y `usoTads.cpp`:

- **Entrega6.tar.gz**

Este archivo se obtiene mediante la ejecución del siguiente comando

```
tar zcvf Entrega6.tar.gz -C src cadena.cpp binario.cpp iterador.cpp pila.cpp colaBinarios.cpp
avl.cpp conjunto.cpp colCadenas.cpp colaDePrioridad.cpp mapping.cpp grafo.cpp  usoTads.cpp
```

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 Entrega6.tar.gz -C src cadena.cpp binario.cpp iterador.cpp pila.cpp colaBinarios.cpp
avl.cpp conjunto.cpp colCadenas.cpp colaDePrioridad.cpp mapping.cpp grafo.cpp  usoTads.cpp
cadena.cpp
binario.cpp
iterador.cpp
pila.cpp
colaBinarios.cpp
avl.cpp
conjunto.cpp
colCadenas.cpp
colaDePrioridad.cpp
mapping.cpp
grafo.cpp
usoTads.cpp
-- El directorio y archivo a entregar es:
???/tarea6/Entrega6.tar.gz
```

Además se genera el archivo `sha.txt` que contiene una clave que identifica al archivo `Entrega6.tar.gz`. Ese archivo también se puede generar mediante la utilidad `shasum`:

```
$ shasum -b Entrega6.tar.gz > sha.txt
```

5.1. Plazos de entrega

El plazo para la entrega es el **viernes 25 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 Funcionamiento y Reglamento del Laboratorio.