Programación y Estructuras de Datos

CUADERNILLO 2

(Curso 2018-2019)

Árbol Binario de Búsqueda (ABB) de TPoro ("TABBPoro")

Qué se pide:

A partir de las estructuras de datos desarrolladas en el **Cuadernillo 1,** se pide construir una clase que representa un **Árbol Binario de Búsqueda (ABB)** de objetos de tipo **TPoro** ("**TABBPoro"**).

Para ello, la clase del árbol ABB a desarrollar empleará las clases **TPoro**, **TVectorPoro y TListaPoro** desarrolladas en el **Cuadernillo 1.**

Internamente, el ABB contendrá un puntero a un nodo.

Para representar cada nodo del ABB, se tiene que definir la clase **TNodoABB** con su forma canónica (constructor, constructor de copia, destructor y sobrecarga del operador asignación) como mínimo.

En la clase **TABBPoro** se tienen que implementar los recorridos en profundidad (INORDEN, PREORDEN y POSTORDEN) y en anchura (NIVELES). Para reducir el coste del cálculo de los recorridos en el ABB, hace falta añadir métodos auxiliares a la parte privada de la clase: **InordenAux, PreordenAux, PostordenAux** (más adelante se detalla su formato).

Prototipo de la Clase "**TNodoABB**"

PARTE PRIVADA

```
// El elemento del nodo
TPoro item;

// Subárbol izquierdo y derecho
TABBPoro iz, de;
```

FORMA CANÓNICA

```
// Constructor por defecto
TNodoABB ();

// Constructor de copia
TNodoABB (TNodoABB &);

// Destructor
~ TNodoABB ();

// Sobrecarga del operador asignación
TNodoABB & operator=( TNodoABB &);
```

Prototipo de la Clase "**TABBPoro**"

PARTE PRIVADA

```
// Puntero al nodo
TNodoABB *nodo;

// AUXILIAR : Devuelve el recorrido en inorden
void InordenAux(TVectorPoro &, int &);

// AUXILIAR : Devuelve el recorrido en preorden
void PreordenAux(TVectorPoro &, int &);

// AUXILIAR : Devuelve el recorrido en postorden
void PostordenAux(TVectorPoro &, int &);
```

FORMA CANÓNICA

```
// Constructor por defecto
TABBPoro();

// Constructor de copia
TABBPoro(TABBPoro &);

// Destructor
~TABBPoro();

// Sobrecarga del operador asignación
TABBPoro & operator=(TABBPoro &);
```

MÉTODOS

```
// Sobrecarga del operador igualdad
bool operator==(TABBPoro &);

// Devuelve TRUE si el árbol está vacío, FALSE en caso contrario
bool EsVacio();

// Inserta el elemento en el árbol
bool Insertar(TPoro &);

// Borra el elemento en el árbol
bool Borrar(TPoro &);

// Devuelve TRUE si el elemento está en el árbol, FALSE en caso contrario
bool Buscar(TPoro &);

// Devuelve el elemento en la raíz del árbol
TPoro Raiz();

// Devuelve la altura del árbol (la altura de un árbol vacío es 0)
int Altura();

// Devuelve el número de nodos del árbol (un árbol vacío posee 0 nodos)
int Nodos();

// Devuelve el número de nodos hoja en el árbol (la raíz puede ser nodo hoja)
int NodosHoja();


// Devuelve el recorrido en inorden
TVectorPoro Inorden();

// Devuelve el recorrido en preorden
TVectorPoro Preorden();

// Devuelve el recorrido en postorden
TVectorPoro Postorden();

// Devuelve el recorrido en niveles
TVectorPoro Niveles();


// Suma de dos ABB
TABBPoro operator+( TABBPoro &);

// Resta de dos ABB
TABBPoro operator-( TABBPoro &);


// Sobrecarga del operador salida
friend ostream & operator<<(ostream &, TABBPoro &);
```

Aclaraciones:

- Se permite amistad entre las clases **TABBPoro** y **TNodoABB**.

- La forma de emplear los métodos AUXILIARES para las ordenaciones, es (por ejemplo, para el caso del recorrido en Inorden):

```
// Devuelve el recorrido en inorden
TVectorPoro TABBPoro::Inorden()
{
  // Posición en el vector que almacena el recorrido
       int posicion = 1;

// Vector del tamaño adecuado para almacenar todos los nodos
       TVectorPoro v(Nodos());
       InordenAux(v, posicion);
       return v;
}
```

  De este modo, se reduce el coste de crear múltiples objetos de tipo **TVectorPoro**, ya que sólo se emplea uno durante todo el cálculo del recorrido.

- Los **TPoro** en el árbol están ordenados en función del volumen.

- Para simplificar los algoritmos, el árbol NO puede contener elementos con el mismo volumen. Por lo tanto, sólo se podrá insertar 1 **TPoro** vacío, al no poder repetirse el volumen en el árbol.

- El **Constructor de Copia** tiene que realizar una copia exacta duplicando todos los nodos del árbol.

- El **Destructor** tiene que liberar toda la memoria que ocupe el árbol.

- Si se asigna un árbol a un árbol no vacío, se destruye el árbol inicial. La **Asignación** tiene que realizar una copia exacta duplicando todos los nodos del árbol.

- En el **operador "=="**, dos árboles son iguales si poseen los mismos elementos independientemente de la estructura interna del árbol (NO se exige que la estructura de ambos sea la misma).

- **Insertar** devuelve TRUE si el elemento se puede insertar y FALSE en caso contrario (por ejemplo, porque el elemento a insertar ya existe en el árbol).

- **Borrar** devuelve TRUE si el elemento se puede borrar y FALSE en caso contrario (por ejemplo, porque no existe en el árbol). El criterio de borrado es sustituir por el mayor de la izquierda.

- **Raiz** devuelve el **TPoro** raíz del árbol. Si el árbol está vacío, devuelve un **TPoro** vacío.

- Los 4 recorridos devuelven un vector (**TVectorPoro**) en el que todas las posiciones están ocupadas por los elementos del árbol (no pueden quedar posiciones sin asignar). Si el árbol está vacío, se devuelve un vector vacío (vector de dimensión 0).

- El operador **SALIDA** muestra el recorrido por Niveles del ABB, con el formato pedido en el **Cuadernillo 1** para la clase **TVectorPoro**.

- Para implementar el recorrido por Niveles hace falta utilizar una estructura de tipo COLA de punteros a ABB. Para implementar el uso de esta estructura, el alumno puede emplear los elementos que considere oportunos. Algunas opciones son:

  - Estructuras "**queue**" pre-definidas en las STL (consultar http://www.cplusplus.com/reference/queue/queue/?kw=queue, y en la siguiente dirección se muestra un ejemplo de uso: http://www.cplusplus.com/reference/queue/queue/push/)
  - Una adaptación de la estructura **TListaPoro** para que se comporte como COLA de punteros a ABB.
  - Etc.

- En el **operador "+",** primero se tiene que sacar una copia del operando (árbol) de la izquierda y a continuación insertar los elementos del operando (árbol) de la derecha según su recorrido por **Inorden**.

- En el **operador "-",** se recorre el operando (árbol) de la izquierda por **Inorden** y si el elemento NO está en el operando (árbol) de la derecha, se inserta en el árbol resultante (inicialmente vacío) y el proceso se repite para todos los elementos del operando de la izquierda.

ANEXO 1. Notas de aplicación general sobre el contenido del Cuadernillo.

Cualquier modificación o comentario del enunciado se publicará oportunamente en el Campus Virtual.

En su momento, se publicarán como materiales del UACLOUD los distintos FICHEROS de MAIN (**tad.cpp**) que servirán al alumno para realizar unas pruebas básicas con los TAD propuestos.
No obstante, se recomienda al alumno que aporte sus propios ficheros **tad.cpp** y realice sus propias pruebas con ellos.
(El alumno tiene que crearse su propio conjunto de ficheros de prueba para verificar de forma exhaustiva el correcto funcionamiento de la práctica. Los ficheros que se publicarán en el CV son sólo una muestra y en ningún caso contemplan todos los posibles casos que se deben verificar)

Todas las operaciones especificadas en el Cuadernillo son obligatorias.

Si una clase hace uso de otra clase, en el código nunca se debe incluir el fichero **.cpp**, sólo el **.h**.

El paso de parámetros como constantes o por referencia se puede cambiar dependiendo de la representación de cada tipo y de los algoritmos desarrollados. Del mismo modo, el alumno debe decidir si usa el modificador CONST , o no.

En la parte PUBLIC no debe aparecer ninguna operación que haga referencia a la representación del tipo, sólo se pueden añadir operaciones de enriquecimiento de la clase.

En la parte PRIVATE de las clases se pueden añadir todos los atributos y métodos que sean necesarios para la implementación de los tipos.

Tratamiento de **excepciones**: todos los métodos darán un mensaje de error (en **cerr**) cuando el alumno determine que se produzcan **excepciones**; para ello, se pueden añadir en la parte privada de la clase aquellas operaciones y variables auxiliares que se necesiten para controlar las excepciones.

Se considera **excepción** aquello que no permite la normal ejecución de un programa (por ejemplo, problemas al reservar memoria, problemas al abrir un fichero, etc.). **NO** se considera **excepción** aquellos errores de tipo **lógico** debidos a las especificidades de cada clase .

De cualquier modo, todos los métodos deben devolver siempre una variable del tipo que se espera.
Los mensajes de error se mostrarán siempre por la salida de error estándar (cerr). El formato será:

ERROR: mensaje_de_error *(al final un salto de línea).*

ANEXO 2. Condiciones de ENTREGA.

2.1. Dónde, cómo, cuándo, valor.

La entrega de la práctica se realizará:

- Servidor: en el SERVIDOR DE PRÁCTICAS, cuya URL es : http://pracdlsi.dlsi.ua.es/

- Fecha: se habilitará la posibilidad de entrega en el Servidor de prácticas entre estas fechas:

  - **Lunes, 29/04/2019.**
  - **Viernes, 03/05/2019 (23:59)**

- A título INDIVIDUAL ; por tanto requerirá del alumno que conozca su USUARIO y CONTRASEÑA en el Servidor de Prácticas.

- Se podrá realizar cuantas entregas quiera el alumno: solo se corregirá la última práctica entregada. Tras cada entrega, el servidor enviará al alumno un INFORME DE COMPILACIÓN, para que el alumno compruebe que lo que ha entregado cumple las especificaciones pedidas y que se ha podido generar el ejecutable correctamente. Este informe también se podrá consultar desde la página web de entrega de prácticas del DLSI (http://pracdlsi.dlsi.ua.es e introducir el nombre de usuario y password). En caso de que la práctica esté correctamente entregada, compilada y ejecutada, en este informe debe salir lo siguiente:

```
==============================================
 DIFERENCIA CON FICHERO DE SALIDA DE REFERENCIA
==============================================

----------------------------------------
```

- Valor :
  - El Cuadernillo 1 vale un 7% de la nota final de PED.
  - **El Cuadernillo 2 vale un 7% de la nota final de PED**
  - El Cuadernillo 3 vale un 6% de la nota final de PED.
  - El EXAMEN PRÁCTICO vale un 30% de la nota final de PED.

2.2. Ficheros a entregar y comprobaciones.

La práctica debe ir organizada en 3 subdirectorios:

DIRECTORIO 'include' : contiene los ficheros (en MINÚSCULAS) :
- **"tabbporo.h" (** incluye las clases **: TABBoro , TNodoABB)**
- **"tporo.h"**
- **"tvectorporo.h"**
- **"tlistaporo.h" (**incluye las clases **: TListaPoro , TListaNodo y TListaPosicion)**

DIRECTORIO 'lib' : contiene los ficheros (NO deben entregarse los ficheros objeto "**.o**") **:**
- **"tabbporo.cpp" (** incluye las clases **: TABBoro , TNodoABB)**
- **"tporo.cpp"**
- **"tvectorporo. cpp "**
- **"tlistaporo.cpp"**

DIRECTORIO 'src' : contiene los ficheros :
- **"tad.cpp" (**fichero aportado por el alumno para comprobación de tipos de datos. No se tiene en cuenta para la corrección**)**

Además, en el directorio raíz , deberá aparecer el fichero **"nombres.txt"** : fichero de texto con los datos de los autores.

El formato de este fichero es:
1_DNI: *DNI1*
1_NOMBRE: *APELLIDO1.1 APELLIDO1.2, NOMBRE1*

2.3 Entrega final

Sólo se deben entregar los ficheros detallados anteriormente (ninguno más).

Cuando llegue el momento de la entrega, toda la estructura de directorios ya explicada (¡ATENCIÓN! excepto el **MAKEFILE**), debe estar comprimida en un fichero de forma que éste NO supere los 300 K (da igual el nombre del fichero .tgz que se entregue; al entrar en este .tgz deben aparecer SOLO los directorios y ficheros indicados anteriormente).

Ejemplo : **`tar –czvf PRACTICA.tgz *`**

El nombre del fichero **.tgz** entregado es indiferente. Debe contener SOLO los ficheros antes indicados.

2.4. Otros avisos referentes a la entrega.

No se devuelven las prácticas entregadas. Cada alumno es responsable de conservar sus prácticas.

La detección de prácticas similares ("copiados") supone el automático suspenso de TODOS los autores de las prácticas similares. No está permitido usar código propio de la práctica extraído de cualquier medio (Internet, compañeros, etc.). La detección de una práctica copiada supone el suspenso automático en la prueba y el envío de un informe al depto. y a la EPS que tomarán medidas oportunas (Normativa EPS y Vicerrectorado de la UA).

ANEXO 3. Condiciones de corrección.

ANTES de la evaluación:

La práctica se programará en el Sistema Operativo Linux, y en el lenguaje C++. **Se corregirá con la versión de compilador de C++ instalada en los laboratorios de la Escuela Politécnica Superior.**

La evaluación:

La práctica se corregirá casi en su totalidad de un modo automático, por lo que los nombres de las clases, métodos, ficheros a entregar, ejecutables y formatos de salida descritos en el enunciado de la práctica SE HAN DE RESPETAR EN SU TOTALIDAD.

A la hora de la corrección del Examen de Prácticas (y por tanto, de la práctica del Cuadernillo), se evaluará:
- El correcto funcionamiento de los TADs propuestos para el Cuadernillo
- El correcto funcionamiento de el/los nuevo/s método/s propuestos para programar durante el tiempo del Examen.

Uno de los objetivos de la práctica es que el alumno sea capaz de comprender un conjunto de instrucciones y sea capaz de llevarlas a cabo. Por tanto, es esencial ajustarse completamente a las especificaciones de la práctica.

Cuando se corrige la práctica, el corrector automático proporcionará ficheros de corrección llamados "**tad.cpp**". Este fichero utilizará la sintaxis definida para cada clase y los nombres de los ficheros asignados a cada una de ellas: únicamente contendrá una serie de instrucciones **#include** con los nombres de los ficheros "**.h**".

NOTA IMPORTANTE: Las prácticas **no se pueden modificar una vez corregidas** y evaluadas (no hay revisión del código). Por lo tanto, es esencial ajustarse a las condiciones de entrega establecidas en este enunciado.

En especial, llevar cuidado con los **nombres de los ficheros** y el **formato especificado para la salida.**

No está permitido usar código propio de la práctica, extraído de cualquier medio **(Internet, compañeros, etc.)**

ANEXO 4. Utilidades

Almacenamiento de todos los ficheros en un único fichero

Usar el comando **tar** para almacenar todos los ficheros en un único fichero:

- `$ tar cvzf practica.tgz *`

Para recuperarlo del disco en la siguiente sesión:

- `$ tar xvzf practica.tgz`

Utilización del depurador gdb

El propósito de un depurador como **gdb** es permitir que el programador pueda "ver"qué está ocurriendo dentro de un programa mientras se está ejecutando.

Los comandos básicos de gdb son:

1. **r (run)**: inicia la ejecución de un programa. Permite pasarle parámetros al programa. Ejemplo: r fichero.txt.
2. **l (list)**: lista el contenido del fichero con los números de línea.
3. **b (breakpoint)**: fija un punto de parada. Ejemplo: b 10 (breakpoint en la línea 10), b main (breakpoint en la función main).
4. **c (continue)**: continúa la ejecución de un programa.
5. **n (next)**: ejecuta la siguiente orden; si es una función la salta (no muestra las líneas de la función) y continúa con la siguiente orden.
6. **s (step)**: ejecuta la siguiente orden; si es una función entra en ella y la podemos ejecutar línea a línea.
7. **p (print)**: muestra el contenido de una variable. Ejemplo: p auxiliar, p this (muestra la dirección del objeto), p *this (muestra el objeto completo).
8. **h (help)**: ayuda.

Utilización de la herramienta VALGRIND

Se aconseja el uso de la herramienta VALGRIND para comprobar el manejo correcto de la memoria dinámica. Se puede encontrar una descripción más detallada de dicha herramienta en el libro **"C++ paso a paso"** de Sergio Luján.

Modo de uso:

valgrind - -tool=memcheck - -leak-check=full nombre_del_ejecutable