Universidad de Córdoba
Escuela Politécnica Superior de Córdoba

ESTRUCTURAS DE DATOS

GRADO EN INGENIERÍA INFORMÁTICA
Segundo curso. Segundo cuatrimestre.
Curso académico 2018 – 2019

TERCERA PRÁCTICA

ÁRBOLES BINARIOS ORDENADOS

- **OBJETIVO.** Implementar un árbol binario ordenado de elementos genéricos. Para ello, los alumnos deberán implementar las funciones vistas en las clases de teoría.
- **ENUNCIADO**. Dada la clase abstracta de interfaz ***ArbolBinarioOrdenado***, implementar la clase ***ArbolBinarioOrdenadoEnlazado*** que heredará de la primera, utilizando celdas enlazadas. Para la implementación de las celdas enlazadas se utilizará la clase ***NodoArbolBinario*** cuya implementación se realizará en la parte privada de la clase ***ArbolBinarioOrdenadoEnlazado***.

- **PRIMERA PARTE. Implementación de la clase nodo**
  - Se trabajará con 2 ficheros, aunque el alumno solo debe **modificar** 1:
    - ***arbolbinariooordenado.hpp*:** código con la interfaz necesaria para definir un árbol binario ordenado.
    - ***arbolbinariooordenadoenlazado.hpp***: código con la implementación del árbol binario ordenado enlazado y del nodo.
  - **Observación**:
    - Las funciones o fragmentos de código a completar vienen indicados en el código por medio del comentario "*// TODO*"
  - **Atributos**:
    - Cada nodo deberá tener
      - Un tipo parametrizado que hará referencia a la información que almacena (*_info*)
      - Un puntero a su hijo izquierdo (*_izquierdo*)
      - Un puntero a su hizo derecho (*_derecho*)
  - **Constructores**
    - ***NodoArbolBinario(info: G)***
      - Crea un nuevo nodo con la información de "*info"*
      - Postcondición
        - El nodo creado no tiene hijos
    - ***NodoArbolBinario (n: NodoArbolBinario)***
      - Crea un nuevo nodo a partir de otro nodo.
      - Postcondición
        - El nodo creado es igual al nodo "*n*".
  - **Métodos**
    - ***G getInfo()***
      - Devuelve la información contenida en el nodo
    - ***NodoArbolBinario * getIzquierdo()***
      - Devuelve el puntero al hijo izquierdo
    - ***NodoArbolBinario * getDerecho()***
      - Devuelve el puntero al hijo derecho

- ***Lógico esHoja()***
  - Comprueba si el nodo actual es hoja, es decir, no tiene hijo izquierdo ni derecho
- ***Void recorridoPreOrden(op: OperadorNodo)***
  - Aplica el operador "*op"* y delega sobre los hijos para realizar el recorrido en preorden
- ***Void reccorridoPostOrden(op: OperadorNodo)***
  - Aplica el operador "*op"* y delega sobre los hijos para realizar el recorrido en inorden
- ***Void recorridoInOrden(op: OperadorNodo)***
  - Aplica el operador "*op"* y delega sobre los hijos para realizar el recorrido en postorden
- ***Void setInfo(info: G)***
  - Establece el valor informativo del nodo actual
- ***Void setIzquierdo(izquierdo: NodoArbolBinario *)***
  - Establece el puntero al hijo izquierdo del nodo actual
- ***Void setDerecho(derecho: NodoArbolBinario *)***
  - Establece el puntero al hijo derecho del nodo actual
- ***NodoArbolBinario operador = (n: NodoArbolBinario)***
  - Operador de asignación. Operador que copia el nodo "*n"* en el nodo actual
  - **Precondición**
    - El nodo "*n"* debe ser diferente del nodo actual
  - **Postcondición**
    - El nodo actual debe ser igual al nodo "*n"*

- **SEGUNDA PARTE. Implementación del árbol binario ordenado**
  - Se trabajará con los mismos ficheros que la primera parte, aunque ahora se completará la implementación del árbol.
  - **Observación**:
    - Las funciones o fragmentos de código a completar vienen indicados en el código por medio del comentario "*// TODO*"
  - **Atributos**:
    - Se deben de establecer los punteros necesarios para la implementación del árbol binario ordenado enlazado
      - Raíz del árbol (*_raiz*)
      - Cursor al elemento actual del árbol (*_actual*)
      - Cursor que apunte al padre de *_actual (_padre)*
  - **Constructores**
    - ***ArbolBinarioOrdenadoEnlazado()***
      - Crea un nuevo árbol vacío
      - **Postcondición**
        - El árbol creado está vacío
    - ***ArbolBinarioOrdenadoEnlazado(a: ArbolBinarioOrdenadoEnlazado)***
      - Crea un nuevo árbol a partir de otro árbol.
      - **Postcondición**
        - El árbol creado es igual al árbol "*a*".
  - **Métodos**
    - ***Lógico estaVacio()***
      - Comprueba si el árbol está vacío
    - ***G raiz()***
      - Obtiene el dato almacenado en la raíz
      - **Precondición**

        - El árbol no puede estar vacío
    - ***Lógico existeActual()***
        - Comprueba si "*_actual"* está apuntando a algún nodo
        - **Precondición**
            - El árbol no puede estar vacío
    - ***G actual()***
        - Devuelve el dato almacenado por el nodo "*_actual"*
        - **Precondición**
            - *"_actual"* debe de apuntar a algún nodo
    - ***Lógico buscar(it: G)***
        - Busca un elemento en el árbol y actualiza el cursor de "*_actual"* y "*_padre"* si lo encuentra
        - **Postcondición**
            - *"_actual"* debe apuntar al nodo encontrado, si lo encuentra
    - ***Lógico insertar (it: G)***
        - Inserta un elemento en el árbol, el cual deberá mantener el orden
        - **Postcondición**
            - El elemento debe estar en el árbol
            - El árbol debe estar ordenado
    - ***Lógico borrar ()***
        - Elimina el nodo apuntado por "*_actual"*
        - **Precondición**
            - *"_actual"* debe apuntar a algún nodo
        - **Postcondición**
            - El elemento borrado no debe existir
    - ***Void borrarArbol()***
        - Elimina el árbol por completo
        - **Precondición**
            - El árbol no puede estar vacío
        - **Postcondición**
            - El árbol debe estar vacío
    - ***void recorridoPreOrden(op: OperadorNodo)***
        - Realiza el recorrido preorden del árbol aplicando el operador "*op*".
    - ***void recorridoInOrden(op: OperadorNodo)***
        - Realiza el recorrido inorden del árbol aplicando el operador "*op*".
    - ***void recorridoPostOrden(op: OperadorNodo)***
        - Realiza el recorrido postorden del árbol aplicando el operador "*op*".

- **TERCERA PARTE. Abstracción de datos y validación**
    - Se trabajará con 5 ficheros, aunque el alumno solo debe **modificar** 1:
        - ***generarpersona.hpp*:** cabecera del fichero *generarpersona.cpp*.
        - ***generarpersona.cpp***: código con la función *generarDatosPersonales()*, la cual se encarga de generar una persona de manera aleatoria.
        - ***persona.hpp***: código responsable de la definición de la clase Persona.
        - ***persona.cpp***: implementa los métodos de la clase Persona.
        - ***main.cpp***: el alumno deberá codificar un fichero principal en el que, mediante la implementación de un menú deberá comprobar que la implementación realizada es correcta. Para ello, se deberán probar al menos las siguientes funciones:
            - Insertar un número de personas, dado por el usuario, en el árbol. Para generar los datos de una persona se usará la función generarDatosPersonales().
            - Comprobar si una persona está en el árbol dados sus datos.

        - Mostrar todos los elementos del árbol usando los recorridos Preorden, PostOrden y InOrden.
        - Borrar una persona dados sus datos.
        - Borrar el árbol completo.
    - **Observación**:
        - Las funciones o fragmentos de código a completar vienen indicados en el código por medio del comentario "*// TODO*"
    - **Verificación:**
        - El mismo fichero Makefile de la primera parte compilará *main.cpp* para generar *main.exe*, el cual servirá para probar la implementación realizada por el alumno.

- **ENTREGA Y EVALUACIÓN**
    - Duración de la práctica nº 3: tres sesiones de dos horas cada una.
    - **Plazo máximo de entrega**
        - 23:55 horas del domingo 28 de abril de 2019
    - Se proporciona un fichero comprimido denominado "practica-3.zip" que contiene los siguientes ficheros
        - **Practica-3.pdf**
            - Enunciado de la práctica 3 (este documento)
        - **Makefile**
            - make:
                - Compila el código y crea dos programas ejecutables denominados *main.exe* y *test.exe* que permiten probar la implementación del árbol binario ordenado.
            - make clean:
                - Borra ficheros superfluos
        - Todos los ficheros de código (*.hpp* y *.cpp*) descritos anteriormente.

    - **Al terminar la práctica**,
        - se deberá subir un fichero **comprimido** denominado "practica-3-**usuario**.zip",
        - donde **usuario** es el **login** de cada estudiante.
        - y que contenga todos los ficheros de la práctica.

    - **Observaciones**
        - Se debe usar el espacio de nombres de la asignatura: **ed**
        - Se debe comentar el código entre líneas.

- **Evaluación**
    - La calificación de la práctica se basará
        - en la calidad y completitud del trabajo realizado.
        - y en la **defensa presencial de cada estudiante**.
    - **Se valorará**
        - La correcta implementación del árbol binario ordenado
        - El correcto funcionamiento del programa principal propuesto como ejemplo.
        - El correcto funcionamiento del programa de test propuesto como ejemplo.
        - La claridad del código, así como de sus comentarios.
        - Un profundo conocimiento de la práctica codificada.