Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Segundo Cuatrimestre, 9 de Septiembre de 2014.

1. (3 puntos) Dado un árbol binario de enteros, se entiende que es un árbol genealógico correcto si cumple las siguientes reglas, producto de interpretar el entero de cada nodo como el *año de nacimiento* del individuo, el hijo izquierdo como su primer hijo de un máximo de dos, y el hijo derecho como el segundo hijo:

- La edad del padre siempre supera en al menos 18 años las edades de cada uno de los hijos.
- La edad del segundo hijo (si existe) es al menos dos años menos que la del primer hijo (no hay hermanos gemelos/mellizos en estos árboles).
- Los árboles genealógicos de ambos hijos son también correctos.

Implementa una función que reciba un árbol binario que permita averiguar si un árbol binario es o no árbol genealógico correcto y, en caso de serlo, el número de generaciones distintas que hay en la familia.

2. (3 puntos) Extiende la implementación de los árboles de búsqueda, añadiendo la siguiente operación:

```
Iterador busca(const Clave &clave) const;
```

que busque en el árbol la clave dada y devuelva un *iterador* que permita recorrer todos los elementos contenidos en el árbol desde esa clave.

A modo de recordatorio, la definición (parcial) de la clase interna `Arbus::Iterador` vista en clase es:

```
class Iterador {
public:

   // ...

protected:

   // Para que pueda construir objetos del
   // tipo iterador
   friend class Arbus;

   // ...

   // Puntero al nodo actual del recorrido
   // NULL si hemos llegado al final.
   Nodo *_act;
```

```
    // Ascendientes del nodo actual
    // aún por visitar
    Pila<Nodo*> _ascendientes;
};
```

3. (4 puntos) Estás trabajando en un nuevo videojuego llamado *CiudadMatic*: un simulador de ciudades, llenas de edificios de distinto tipo. Será posible construir y reparar estos edificios gastando dinero, y al final de cada turno (después de haber construido y reparado tantos edificios como se quiera), recaudar los impuestos que generen. Necesitarás implementar las siguientes operaciones:

- *CiudadMatic*: inicializa una nueva ciudad vacía, con el dinero que se pase como argumento disponible para ser gastado.
- *nuevoTipo*: añade un nuevo tipo de edificio al sistema, con un identificador proporcionado por el usuario (por ejemplo, "bar"), un coste de construcción, una cantidad de impuestos generada por turno, y una calidad de base (máximo de turnos sin reparar). No devuelve nada.
- *insertaEdificio*: dado el nombre de un edificio (por ejemplo, "El Bar de Moe"), y el identificador de su tipo, y asumiendo que se disponga del dinero necesario para construirlo, añade el edificio a la ciudad y resta su coste de construcción del dinero disponible. No devuelve nada.
- *reparaEdificio*: repara el edificio cuyo identificador se pase como argumento a "como recién construido", a un coste del 10 % del coste de construcción (descartando los decimales), independientemente de lo estropeado que estuviese. No devuelve nada.
- *finTurno*: todos los edificios construidos generan los impuestos que les corresponden por su tipo. Después, todos se estropean por un punto de calidad, y aquellos que lleguen a 0 son derribados y eliminados de la ciudad. Devuelve el dinero total disponible para el nuevo turno (impuestos generados + dinero no gastado del turno anterior).
- *listaEdificios*: dado un identificador de tipo de edificio, devuelve una lista con los edificios de ese tipo que están actuamente construidos, por orden de antigüedad (primero el más viejo).

Desarrolla en C++ una implementación de la clase *CiudadMatic* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. En cada operación, indica también, de forma razonada, su complejidad.