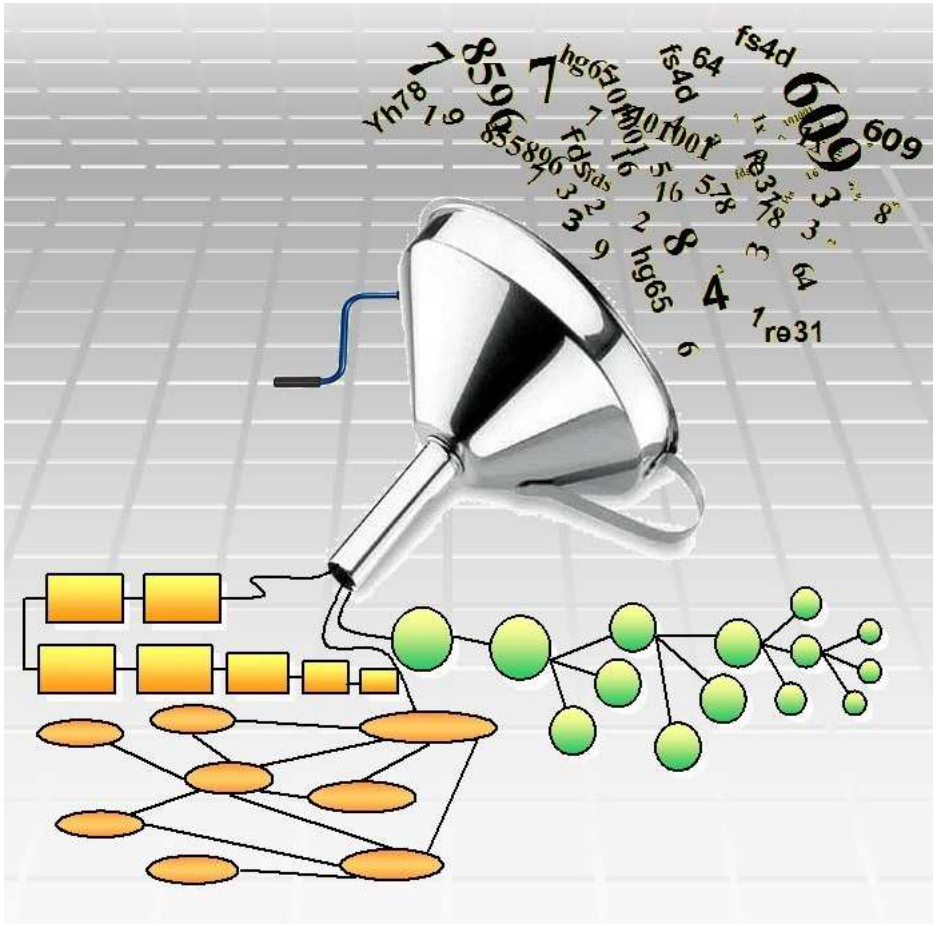
Memoria Practica 02



Autor:	David Subires Parra
Asignatura:	Estructuras de Datos y Algoritmos 1
Grado:	Ingeniería informática
Universidad:	Universidad de Almería
Curso Académico:	2013/2014

Estructura de datos: árbol AVL (Adelson-Velski y Landis)

Practica02.Ejercicio02

La clase **AVLTree** contiene a su vez dos clases, **AVLNode**<T> y **TreeIterator**:

AVLNode<T> → Clase que implementa el nodo que, combinado con otros nodos, forman el AVLTree.
Contiene un atributo "nodeValue" de tipo genérico que será dónde se almacene el valor del nodo, también contiene dos atributos "left y right" del tipo AVLNode<T>, que harán referencia al hijo izquierdo e hijo derecho de dicho nodo y además el atributo height(int) que será la altura del nodo(número de "niveles" que tiene por debajo).
Tiene un único constructor, AVLNode(T item), que establece como nodeValue el valor item pasado por parámetro, left y right igual a null y height igual a 0.

- **TreeIterator** → Clase que implementa un iterador adaptado al AVLTree.
  Sus atributos son ALStack<AVLNode<T>> (stack) curr y lastReturned del tipo AVLNode<T> y expectedModCount del tipo int.
  La pila se usa para almacenar temporalmente los nodos que recorremos, para poder acceder después a ellos de forma ordenada, ya que los nodos esta estructura no dispone del atributo parent.
  curr apunta hacia el nodo actual, y lastReturned hacia el último devuelto.
  expectedModCount sirve para comprobar la integridad de los datos entre el iterador y el AVLTree.
  El procedimiento que sigue para recorrer los nodos, con los métodos típicos de Iterator(hasNext(),current(), next()) es el siguiente:
  Al instanciar el TreeIterator, current apunta hacia el primero nodo hoja izquierdo, después, si éste tiene hijo derecho, current apuntara a la primera hoja izquierda de ese hijo derecho, y cuando current no tenga hijo derecho, current apuntara hacia el último elemento apliado en el Stack ALSTack.

Los atributos de la clase AVLTree son AVLNode<T> **root**, int **treeSize** y int **modCount**:

- **AVLNode**<T> root → Elemento que contiene la estructura de datos.
- **int** treeSize → Tamaño del AVLTree, o número de nodos que contiene root.
- **int** modCount → Se incrementa/decrementa cuando se añaden/eliminan elementos de root. Sirve para comprobar la integridad de los datos entre el AVLNode y TreeIterator.

AVLTree dispone sólo de un constructor, al que se invoca de la siguiente manera:
AVLTree() → La inicialización consiste en asignar null a root, y 0 a modCount y treeSize.

Los métodos más relevantes de AVLTree son:

- **add**(T item) → Añade el elemento item al AVLTree.
  Devuelve ture si la inserción se realiza correctamente, false en caso de que ya exista el elemento (no añade duplicados), y por lo tanto no lo inserta.
  Consiste en llamar al método privado addNode(AVLNode<T>,T), y en caso de que este no lance la expeción IlegalStateException, que significaría que el elemento item ya está en la estructura, aumentaría en uno treeSize y modCount.

Tiempo peor caso → log n.
Más eficiente que BSTree, con su tiempo peor caso → n

- **remove**(Object item) → Elimina el objeto item, pasado por parámetro, del AVLTree.
  Devuelve true si elimina el elemento, false en c aso contrario (el elemento no está en la estructura).
  Para realizar la operación llama al método privado remove(AVLNode<T>,T), el primer parámetro será la estructura de donde se quiere eliminar el elemento, y el segundo parámetro el elemento a borrar, item.

- **clear**() → Vacía lógicamente el AVLtree.
  Para ello, asgina root, y todos sus hijos derechos e izquierdos, el valor null, y treeSize = 0;

- **contains**(Object item) → Devuelve true, si item está en la estructura, y false en caso contrario.
  Para comprobarlo, llama al método privado findNode(Object), si éste devuelve null, el objeto no está en la estructura de datos, y contains devuelve false, en cualquier otro caso, devuelve true.

- **isEmpty**() → Devuelve true si el atributo treeSize es igual a 0, false en caso contrairo.

- **Iterator**() → Devuelve un iterador del tipo TreeIterator(Explicado anteriormente).

- **toArray**() → Devuelve un objeto del tipo Object[] con el contenido del AVLTree.
  Para ello, recorre el AVLTree mediante TreeIterator, y en el orden de éste , desde el valor más pequeño hasta el valor más grande, almacena el valor de cada AVLNode en el array Object[] para posteriormente devolverlo.

- **toString**() → Devuelve un String que contiene el valor de todos los nodos del AVLTree. Para ello recorre el AVLTree mediante treeIterator y va almacenando cada valor (una vez más en el orden de treeIterator) en el String para posteriormente devolverlo.
  La salida tendría el siguiente formato: [1, 2, 3]

- **find**(T) → Busca en el AVLTree el nodo con valor(nodeValue) igual a “item” pasado como parámetro. Si lo encuentra, devuelve el valor de dicho nodo, sino, devuelve null.
  Para realizar la operación llama al método privado findNode(Object).

Métodos **privados** más relevantes de **AVLTree**:

- **findNode**(Object) → Devuelve AVLNode con nodeValue igual a item, o null en caso de no encontrarlo.
  Para realizar la operación, compara el valor item, con el valor del nodo raiz(root), y en caso de que root no sea el nodo a buscar(lo más usual), vuelve a

comparar item con el hijo izquierdo/derecho dependiendo de si item es menor/mayor que el valor del nodo comparado. Realiza este proceso hasta llegar a un nodo vacío o encuentre el nodo buscado, y entonces devolvera null o el nodeValue correspondiente.

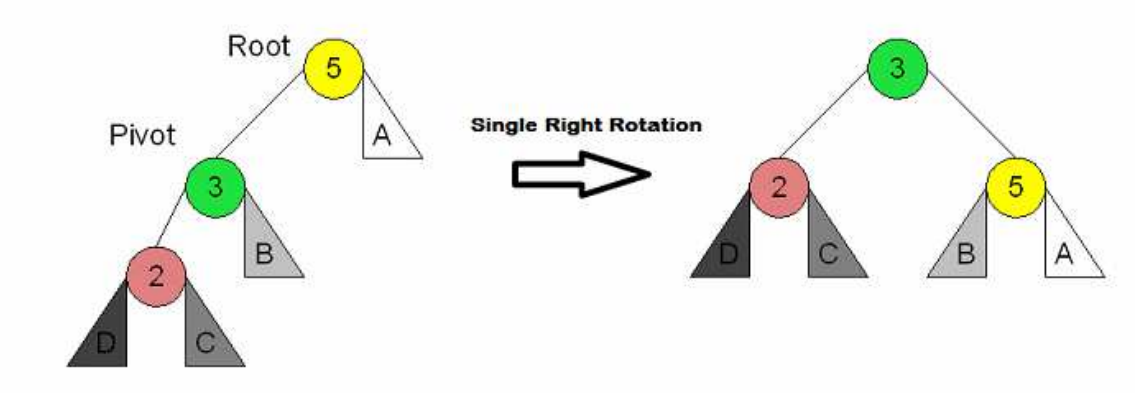
- **addNode**(AVLNode<T> t,T item) → Devuelve el AVL Node pasado por parametro, con el objeto T, pasado por parámetro tambien, añadido. En caso de ya existir en la estructura devuelve IlegalStateException.
  Para realizar la operación, este método se llama a si mismo aplicando recursividad. Primero compara el valor item, con el valor del nodo t. Si item es menor, se llama a si mismo pero con el hijo izquierdo de t [addNode(t.left, item)], si item es mayor que el valor del nodo t, se llama a si mismo pero con el hijo derecho de t. Estas llamadas recursivas acabarán cuando lleguen a un nodo t que sea igual a null, y entonces será ahí donde se añada el nuevo item, instanciando AVLNode con el valor item.
  Cuando el nodo t es igual a null, entonces t pasará a ser un nuevo AVLNode asignando item al nodeValue de éste. Después se le asigna a t su altura, height, que será el máximo de las alturas de sus hijos(si no tiene hijos, -1) más uno.
  Si la inserción se realiza en el sub-arbol izquierdo o derecho, se comprueba si se siguen cumpliendo las propiedades/restricciones del AVLTree, si no es así, (la altura de dos hijos, izquierdo y derecho, difiere en más de una unidad), se lanzará la operación necesaria para reestablecer las propiedades del AVLTree.[singleRotateRight, singleRotateLeft, doubleRotateRight, doubleRotateLeft]

- **remove**(AVLNode<T> t,T item) → Devuelve t pero sin el nodo que contiene item. En caso de que root sea igual a null, o que item no se encuentre en la estructura, devuelve la excepción IllegalArgumentException.
  Para realizar la operación, este método se llama a si mismo aplicando recursividad. Primero compara el valor item, con el valor del nodo t. Si item es menor, se llama a si mismo pero con el hijo izquierdo de t [addNode(t.left, item)], si item es mayor que el valor del nodo t, se llama a si mismo pero con el hijo derecho de t. Estas llamadas recursivas acabarán cuando lleguen a un nodo t que sea igual a null (por lo tanto item no está en la estructura) o cuando se encuentre el item a eliminar, donde llamaría al método privado removeNode(), el cual se encargaría de eliminar dicho item del AVLNode, y devolver el nodo t pero sin el item eliminado.

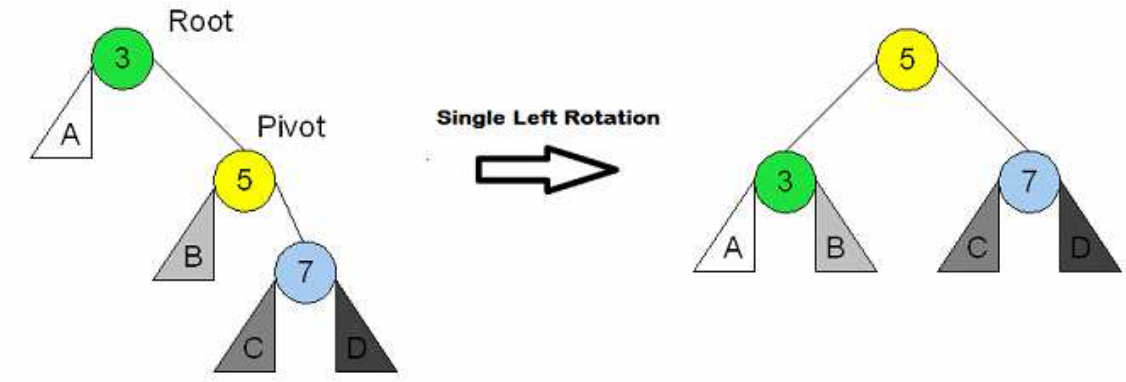
- **removeNode**(AVL<T>) → Elimina el nodo pasado por parámetro del AVLTree.
  Para ello, se comprueba si los dos hijos del nodo a elimiar son distintos a null(esto es, el nodo tiene dos hijos), en este caso el nodo eliminado es reemplazado por el nodo hoja más a la izquierda del sub-arbol derecho del nodo reemplazado.
  En cualquier otro caso, el nodo será reemplazado por su único hijo o por null si no tiene ninguno.
  En el caso de que el nodo tenga dos hijos, despues eliminar el nodo, y reemplazarlo, se comprueba si se siguen cumpliendo las propiedades/restricciones del AVLTree, si no es así, (la altura de dos hijos, izquierdo y derecho, difiere en más de una unidad), se lanzará la operación necesaria para reestablecer las propiedades del AVLTree.[singleRotateRight,

singleRotateLeft, doubleRotateRight, doubleRotateLeft]

- **singleRotateRight**(AVLNode<T> p) → Devuelve p, con la rotación simple derecha realizada.
  Consiste en desplazar el nodo p su hijo derecho, quedando en la posición de p el antes p.left. Para ello p quedará en p.right y p.left quedara en p. Con esto se consigue restar uno a la altura del hijo izquierdo de p(sumando 1 al hijo derecho). Quedando el AVLTree equilibrado.



- **singleRotateLeft**(AVLNode<T> p) → Devuelve p, con la rotación simple izquierda realizada.
  Consiste en desplazar el nodo p su hijo izquierdo, quedando en la posición de p el antes p.right. Para ello p quedará en p.left y p.right quedara en p. Con esto se consigue restar uno a la altura del hijo derecho de p(sumando 1 a la altura hijo izquierdo). Quedando el AVLTree equilibrado.



- **doubleRotateRight**(AVLNode<T> p) → Devuelve p, con la rotacion doble derecha realizada.
  Consiste en realizar una rotacion simple izquierda, singleRotateLeft(), sobre el hijo izquierdo de p, y posteriormente una simplre rotacion derecha, singleRotateRight, sobre p.
  Soluciona los desequilibrios del AVLTree.

- **doubleRotateLeft**(AVLNode<T> p) → Devuelve p, con la rotacion doble izquierda realizada.
  Consiste en realizar una rotacion simple derecha, singleRotateRight(), sobre el hijo derehco de p, y posteriormente una simplre rotacion izquierda, singleRotateLeft, sobre p.
  Soluciona los desequilibrios del AVLTree.

ArrayList vs AVLNode

Tiempos en los peores casos de ambas estructuras.
ArrayList:
Tiempo inserción: O(1)
Tiempo eliminación: O(n)
Tiempo búsqueda: O(n)

AVLTree:
Tiempo inserción: O(log n)
Tiempo eliminación: O(log n)
Tiempo búsqueda: O(log n)

ArrayList
- Ventajas:
  - Permite acceso aleatorio a todos sus elementos en O(1).
  - Permite añadir, borrar y obtener un elemento en una posición concreta en O(1)
  - Más eficiente a la hora de iterar sobre sus elementos.
- Inconvenientes:
  - Peores tiempos de eliminación y búsqueda
  - No mantiene orden entre los elementos
  - Es una estructura finita, se va redimensionando en función de su uso.

AVLTree:
- Ventajas:
  - Mantiene todos los datos ordenados
  - Mejores tiempos de eliminación y búsqueda
  - Es una estructura infinita, no necesita establecer/redimensionar su tamaño.
- Inconvenientes:
  - No permite acceso aleatorio a cualquier elemento

Por último, ArrayList por defecto no tiene método buscar, y AVLTree si. Esto facilita la implementación de la estructura de datos en la aplicación, y reduce el número de líneas. Aunque la función find se podría implementar también en ArrayList, no sería tan eficiente como AVLTree(O(log n) vs O(n)).

Implementar la clase CiudadDirecciones

Para incluir la clase CiudadDirecciones en nuestra estructura, habría que modificar el atributo ciudades, del tipo AVLTree<String> a AVLTree<CiudadDirecciones>, y modificar el método addCiudad(String) para que tenga en cuenta que si la ciudad exite, inserte la calle en dicha ciudad.

Consulta devolverEmpresasProyectosEuropeos:

```
public static ArrayList<String>
 devolverEmpresasProyectosEuropeos(AVLTree<EmpresaProyectos> listaEmpresas) {
    EmpresaProyectos tempEP;
    CiudadDirecciones tempCD;
    String empresas = "",direcciones = "";
    Iterator<EmpresaProyectos> iterador = listaEmpresas.iterator();
    while(iterador.hasNext()){
        tempEP = iterador.next();
        Iterator<ProyectoCiudades> iterador2 = tempEP.getProyectosciudades().iterator();
        while(iterador2.hasNext()){
            Iterator<CiudadDirecciones> iterador3 = iterador2.next().getCiudades().iterator();
                while(iterador3.hasNext()){
                    tempCD = iterador2.next();
                    if(tempCD.getContinente().equals("Europa"){
                        empresas += tempEP.getEmpresa()+"\n";
                        Iterator<String> iterador4 = tempCD.getDirecciones(){
                            direcciones += iterador4.next()+" ";
                        }
                        direcciones += "\n";
                    }
                }
        }
    }
    return empresas+direcciones;
}
```