UNLP. Facultad de Informática.

**Algoritmos y Estructuras de Datos 2006** **Práctica 8**

## Árboles Binarios de Búsqueda, árboles AVL y Heap.

Objetivos

- Representar árboles binarios de búsqueda, implementar las operaciones de la abstracción y describir soluciones utilizándolos
- Representar árboles AVL, implementar las operaciones de la abstracción y describir soluciones utilizándolos

Ejercicio 1.

a) A partir de un árbol binario de búsqueda vacío, dibuje las transformaciones que sufre al insertar cada uno de los siguientes elementos: 3, 1, 4, 6, 8, 2, 5, 7.

b) Dibuje como queda el árbol al eliminar los elementos: 7, 1 y 6.

c) A partir de un árbol binario de búsqueda nuevamente vacío, dibuje las transformaciones que sufre al insertar cada uno de los siguientes elementos: 5,3,7,1,8,4,6.

d) Dibuje como queda el árbol al eliminar los elementos: 5, 3 y 7.

e) ¿Qué puede concluir a partir de los árboles de a) y c)?

Ejercicio 2.

a) Dado un árbol AVL vacío, dibuje las transformaciones que sufre al insertar en el orden dado cada uno de los siguientes elementos, indicando en cada caso el tipo de rotación empleado: 40, 20, 30, 38, 33, 36, 34, 37.

b) Dibuje como queda el árbol al eliminar los elementos: 20, 36, 37, 40.

Ejercicio 3.

Graficar las sucesivas transformaciones que sufre una **heap** con las siguientes operaciones:

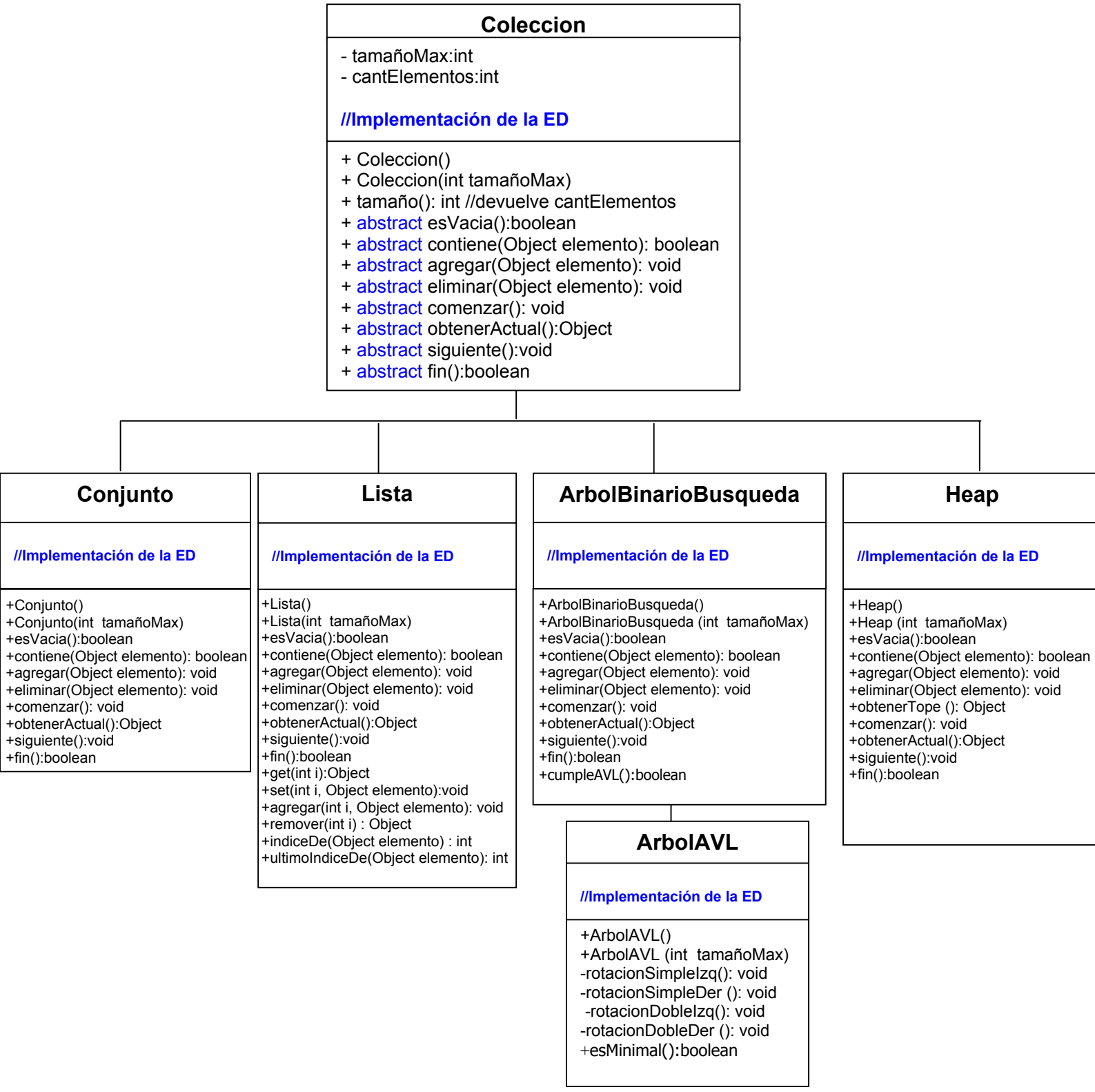
a) Insertar 10, 12, 1, 14, 6
b) Eliminar dos elementos
c) Insertar 5, 8, 15, 3, 9 7, 4
d) Eliminar dos elementos
e) Insertar 11, 13, 2

Ejercicio 4.

Implementar la siguiente jerarquía de clases, de acuerdo a la siguiente especificación:

Las clases **Colección**, **Lista** y **Conjunto** mantienen la funcionalidad especificada en la practica 7.

Las clases **ArbolBinarioBusqueda** y **Heap** son subclases de Colección y **ArbolAVL** es subclase de **ArbolBinarioBusqueda**.




El método **cumpleAVL():boolean** determina si el ArbolBinario es un árbol AVL, es decir si para nodo se cumple que el alto de sus subárboles difieren en a lo sumo 1
Un árbol AVL es minimal de altura h (es decir, si se saca algún nodo deja de ser AVL o de tener altura *h*) si el número mínimo de nodos queda especificado por la siguiente recurrencia.

$$\min \begin{cases} 1 & \text{si } h = 0 \\ 2 & \text{si } h = 1 \\ 1 + \min(h-1) + \min(h-2) & \text{si } h > 1 \end{cases}$$

a) Modifique la clase **Colección** para que implemente la interface **java.util.Iterator**.
   i) ¿Qué métodos debe incorporar a la definición de la clase?
   ii) ¿Qué métodos serian redundantes y deberían eliminarse?
   iii) ¿De que clases deberían ser eliminados?

b) Modifique la lógica de los métodos agregar(), eliminar() y contiene() de **ArbolBinarioBusqueda**, **ArbolAVL** y de **Heap** de tal manera que sigan recibiendo como argumento elementos de tipo Object y a su vez obligar a que estos sean de tipo **Comparable**.
   i) ¿Qué métodos debe incorporar a la definición de la clase de los elementos?
   ii) ¿Qué nos garantiza que los elementos implementen esta interfaz?

c) Implemente una clase llamada **TestArbolBinarioBusqueda** en el paquete **estructurasdedatos.aplicaciones.tp8** que instancie un objeto de tipo **ArbolBinarioBusqueda** que agregue diez nombres de materias de la carrera (Strings), luego recórralo e imprima el contenido del mismo.

d) Ídem c), usando objetos del tipo Alumno de acuerdo a lo especificado en la práctica 2.
   i) ¿Necesitó hacerle alguna modificación a la clase Alumno? . En caso afirmativo, ¿cuál fue la misma?

Ejercicio 7
Implementar en Java una solución para los siguientes problemas.

a) Defina la clase **Secuencia** en el paquete **estructurasdetatos.aplicaciones.tp8**. Esta clase es una colección que almacena elementos no repetidos y ordenados. Debe implementar los métodos siguiente() y agregar() de forma tal que se resuelvan con O(log n) en el peor de los casos. El método buscar() puede tener cualquier complejidad.

**siguiente(Object x): Object**
Devuelve el elemento inmediatamente superior a x (x puede no existir).

**buscar(int k):Object**
Recupera el k_ésimo elemento mayor de la estructura ($k \leq n$).

**agregar(Object x): void**
agrega el elemento x en la estructura.

Nota: tenga presente que los elementos de esta colección deben ser de tipo **Comparable**.

b) Defina una clase llamada **Diccionario** en el paquete **estructurasdedatos.aplicaciones.tp8**. El Diccionario contiene una colección de pares de elementos. Cada para está formado por: una clave (única) y un valor asociado. El Diccionario no está ordenado. Implemente los métodos dados a continuación. Tenga en cuenta que los métodos agregar(), reemplazar() y recuperar() deben poseer tiempo de ejecución **O (log n)** en el peor de los casos:

**agregar(Object clave, Object valor): void**
Agrega un nuevo elemento en el Diccionario, dado por el par **clave** y **valor**.

**reemplazar(Object clave, Object valor): void**
Reemplaza el valor del elemento con la clave dada por **clave.**

**agregarATodos(Object valor): void**
Reemplaza el valor de todos los elementos del Diccionario por el dado por el argumento **valor.**

**recuperar(Object clave): Object**
Recupera el valor asociado a la clave dada por el argumento **clave**.

Nota: tenga presente que las elementos de esta colección deben ser de tipo **Comparable**.

c) Se desea definir un esquema de organización de memoria dinámica, que consiste en organizar bloques de memoria en forma eficiente. Cada bloque de memoria se mide en kbytes (en la memoria pueden existir muchos bloques de igual tamaño), se identifica por su dirección física de comienzo (representada por un número entero) y tiene un estado libre u ocupado.

Escriba una clase llamada **Memoria** en el paquete **estructurasdedatos.aplicaciones.tp8**, de manera tal que la estructura de datos definida, permita que los objetos soporten las siguientes operaciones con complejidad **O (log n)** en el peor de los casos.

**int pedirBloque(int kb): int**
Devuelve la dirección de comienzo de un bloque de memoria libre de tamaño igual a Kb. Si no hubiera devuelve –1. A su vez, cambia el estado del bloque a ocupado.

**liberarBloque(int dirComienzo): void**
Cambia el estado del bloque a libre.

Ejercicio 10.
a) Agregue a la clase Lista del trabajo práctico 3 el método **transformarEnHeap():Heap**, que retorna una **Heap**, que se construye agregando cada uno de los elementos de la lista a una heap usando el método agregar().
b) Agregue a la clase **ArbolBinario** del trabajo práctico 7 el método **transformarEnArreglo():Object[]** que copia un ArbolBinario completo a un Arreglo de Objetos ubicando los elemento por niveles. La raíz en el primer lugar, el hijo izquierdo en el siguiente, y el derecho a continuación, y así siguiendo. Luego implemente en la clase **Heap** el constructor **Heap(Object[] unArreglo)** que inicializa una heap a partir de reordenar los elementos del arreglo de objetos que recibe como parámetro.
Pruebe que los métodos anteriores funcionan correctamente definiendo una clase **TestHeap** en el paquete **estructurasdedatos.aplicaciones.tp8** que crea un árbol binario, lo convierte en una arreglo y crea una Heap a partir de este.
c) Calcular el tiempo de ejecución de los algoritmos definidos en a) y b) teniendo en cuenta que los datos ingresan ordenados:
i) en forma creciente.
ii) en forma decreciente.

Ejercicio 11.
Dada una secuencia de valores que representan los milímetros de lluvia caída, se necesita buscar el valor k-ésimo menor. La secuencia de 100.000 elementos está almacenada en un archivo. Para encontrar el k-ésimo elemento sólo se pueden almacenar en memoria k elementos simultáneamente. Resolver el ejercicio en pseudocodigo en papel (no pasar por máquina) de forma tal que la cantidad de comparaciones entre elementos sea mínima.
Defina una clase **PseudoOrdenador** en el paquete **estructuradedatos.aplicaciones.tp8**. Implemente un constructor PseudoOrdenador (Archivo unArchivo) y un método llamado buscarKesimo (int k): int.
Considere cuenta con una clase **Archivo** posee los métodos **leer():String**, **escribir(String s):void**, que permiten leer secuencialmente archivos de texto.