UNLP. Facultad de Informática.

Algoritmos y Estructuras de Datos 2007 **Trabajo Práctico Nro. 6**

Árboles Binarios de Búsqueda, árboles AVL y Heap.

Objetivos

- Representar e implementar las operaciones de las abstracciones árboles binarios de búsqueda, árboles AVL y HEAP.
- Describir soluciones utilizando las abstracciones.

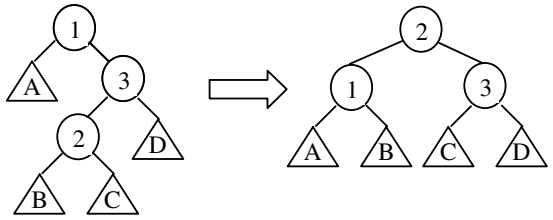
Ejercicio 1.

a) Muestre en papel (dibuje) las transformaciones que sufre un árbol binario de búsqueda (inicialmente vacío) al insertar cada uno de los siguientes elementos: 3, 1, 4, 6, 8, 2, 5, 7.
b) Muestre como queda el árbol al eliminar los elementos: 7, 1 y 6.
c) A partir de un árbol binario de búsqueda nuevamente vacío, muestre 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 sobre la altura del árbol a partir de a) y c)?

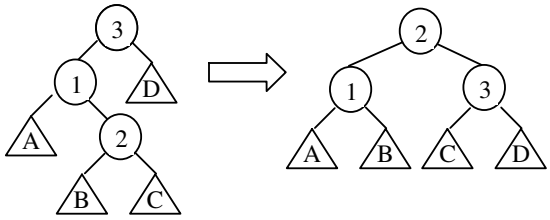
Ejercicio 2.

a) Muestre las transformaciones que sufre un árbol AVL (inicialmente vacío) al insertar cada uno de los siguientes elementos: 40, 20, 30, 38, 33, 36, 34, 37. Indique el tipo de rotación empleado en cada balanceo.
b) Dibuje como queda el árbol al eliminar los elementos: 20, 36, 37, 40. Considere que para eliminar un elemento con dos hijos lo reemplaza por el mayor de los más pequeños.

Rotación doble (entre un nodo y su hijo derecho, y entre este último y su hijo izquierdo)



Rotación doble (entre un nodo y su hijo izquierdo, y entre este último y su hijo derecho)



Ejercicio 3.

Muestre las transformaciones que sufre una min HEAP (inicialmente vacía) al realizar las siguientes operaciones:

a) Insertar 10, 12, 1, 14, 6
b) Eliminar dos elementos
c) Insertar 5, 8, 15, 3
d) Eliminar dos elementos

Ejercicio 4.

Considere la siguiente especificación de la clase **ArbolBinarioDeBusqueda** (con la representación hijo izquierdo e hijo derecho):

ArbolBinarioDeBusqueda
- NodoBinario raiz
+ArbolBinarioDeBusqueda() +incluye(Object dato): boolean +agregar(Object dato) +eliminar(Object dato)

NodoBinario
- Object dato - NodoBinario hijoIzquierdo - NodoBinario hijoDerecho
NodoBinarioDeBusqueda(Object dato) getDato():Object getHijoIzquierdo():NodoBinario getHijoDerecho():NodoBinario setDato(Object dato) setHijoIzquierdo(NodoBinario unHijo) setHijoDerecho(NodoBinario unHijo)

El constructor **ArbolBinariodeBusqueda()** inicializa un árbol binario de búsqueda vacío, es decir, la raíz en null.

El mensaje **incluye (Object dato)** retorna true si dato está contenido en el árbol, falso en caso contrario.

El mensaje **agregar (Object dato)** agrega dato al árbol.

El mensaje **eliminar (Object dato)** elimina dato del árbol.

Implemente la clase **ArbolBinarioDeBusqueda** (la clase **NodoBinario** ya la implementó en la práctica anterior).

- Tenga presente que si bien los métodos **incluye (Object dato)**, **agregar(Object dato)** y **eliminar(Object dato)** reciben un dato de tipo Object, el mismo también debe ser de tipo **Comparable**.

Ejercicio 5.
Considere la siguiente especificación de la clase **ArbolAVL** (con la representación hijo izquierdo e hijo derecho):

ArbolAVL
- NodoAVL raiz
+ArbolAVL() +incluye(Object dato): boolean +agregar(Object dato) +eliminar(Object dato) -rotacionSimpleIzquierda(NodoAVL nodo) -rotacionSimpleDerecha (NodoAVL nodo) -rotacionDobleIzqquierda(NodoAVL nodo) -rotacionDobleDerecha (NodoAVL nodo)

NodoAVL
- Object dato - int altura - NodoAVL hijoIzquierdo - NodoAVL hijoDerecho
NodoAVL(Object dato) getDato():Object getHijoIzquierdo():NodoAVL getHijoDerecho():NodoAVL getAltura(): int setDato(Object dato) setHijoIzquierdo(NodoAVL unHijo) setHijoDerecho(NodoAVL unHijo) setAltura(int h)

El constructor **ArbolAVL()** inicializa un árbol AVL vacío, es decir, la raíz en null.

Los mensaje **incluye(Object dato)**, **agregar(Object dato)** y eliminar**(Object dato)** se comportan igual que en el ArbolBinarioDeBusqueda, sólo que el **agregar(Object dato)** y **eliminar(Object dato)** se encargan de mantener el árbol balanceado.

Los cuatro mensajes de rotación se encargan de realizar la rotación correspondiente para el nodo en cuestión. Tenga presente que no son métodos públicos.

Implemente las clases **ArbolAVL y NodoAVL**

Tenga presente que si bien los métodos **incluye(Object dato**), **agregar(Object dato)** y **eliminar(Object dato)** reciben un dato de tipo Object, el mismo también debe ser de tipo **Comparable**.

Ejercicio 6.
a) Agregue a la clase ArbolBinarioDeBusqueda el método de instancia **cumpleAVL():boolean,** el cual determina si el ArbolBinarioDeBusqueda es un árbol AVL, es decir si para cada nodo se cumple que el alto de sus subárboles difieren en a lo sumo 1.

b) Agregue a la clase ArbolAVL el método **esMinimal():boolean**. 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:

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

Si necesita, podría definir un método privado, en la clase ArbolAVL, **cantidadDeNodos():int** y otro **min(int h):int**, para resolver **esMinimal():boolean**.

Ejercicio 7.
Considere la siguiente especificación de la clase **Heap**:

Heap
Object[] datos int cantElementos
+ Heap() + Heap(Lista l) + Heap(Object[] datos) + esVacia():boolean + agregar(Object dato) + tope(): Object

El constructor **Heap()** inicializa una Heap vacío.
El constructor **Heap(Lista l)** crea una Heap a partir de los elementos de una lista.
El constructor **Heap(Object[] datos)** crea e inicializa una Heap a partir de reordenar los elementos del arreglo de objetos que recibe como parámetro.
El mensaje **esVacia()** devuelve true si no hay elementos para extraer.
El mensaje **agregar(Object dato)** agrega dato a la heap.
El mensaje **tope()** elimina el tope y lo retorna.

Tenga presente que si bien el método **agregar**(Object dato) recibe un dato de tipo Object, el mismo también debe ser de tipo **Comparable**.

a) A partir de la especificación anterior implemente la Clase Heap

b) Calcular el tiempo de ejecución de los constructores: Heap(Lista l) usado para inicializar una Heap a partir de una lista y Heap(Object[] datos) usado para inicializar una Heap a partir de un arreglo. Calcule el tiempo teniendo en cuenta que los datos están ordenados:

(i) en forma creciente

(ii) en forma decreciente.

Ejercicio 8.
Implementar en Java una solución para los siguientes problemas.

a) Defina una clase llamada **Diccionario**. El mismo contiene una colección de pares de elementos. Cada elemento está formado por una clave única y un valor asociado.

Diccionario
- ¿? elementos
+Diccionario() +agregar(Object clave, Object valor) +reemplazar(Object clave, Object valor) +agregarATodos(Object valor) +recuperar(Object clave): Object

El mensaje **agregar(Object clave, Object valor)** agrega un nuevo elemento en el Diccionario, dado por el par **clave** y **valor**.

El mensaje **reemplazar(Object clave, Object valor)** reemplaza el valor del elemento con la clave dada por **clave**.

El mensaje **agregarATodos(Object valor)** reemplaza el valor de todos los elementos del Diccionario por el argumento **valor**.

El mensaje **recuperar(Object clave): Object** recupera el valor asociado a la **clave**.

Para la implementación de los métodos tenga en cuenta que **agregar()**, **reemplazar()** y **recuperar()** deben poseer tiempo de ejecución **O (log n)** en el **peor** de los casos. Determine que estructura de datos debe utilizar para poder cumplir con el orden solicitado, que cambios necesita realizar sobre ella y que estructuras adicionales necesita.

b) 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 kilobytes (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.

Memoria
¿?
+Memoria() +pedirBloque(int kilobytes):int +liberarBloque (int dirComienzo)

Bloque
- boolean libre - int tamaño - int direccionDeComienzo
+Bloque (int tam, int dirDeComienzo) +setOcupado() +setLibre() +getLibre()

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

El mensaje **liberarBloque(int direccionDeComienzo)** cambia el estado del bloque a libre.
Debe elegir la o las estructuras de datos adecuadas para implementar la clase **Memoria**, de manera tal que las operaciones anteriores se resuelvan con complejidad **O (log n)** en el peor de los casos.

Solución (no la tendrán los alumnos): La memoria debería tener dos árbolesAVL, uno ordenado por tamaño del bloque y el otro por direcciónDeComienzo. El tamaño de bloque se puede repetir, por lo cual, el árbol

con ese orden debería tener en cada nodo una lista de bloques para el mismo tamaño. En cambio, la dirección de comienzo es única. El mismo bloque debería estar en ambos árbolesAVL.
Para pedirBloque() se recorre el AVL por tamaño para buscar un bloque libre de la lista. Si lo encuentra, le cambia el estado. Para liberar un bloque, se recorre el AVL ordenado por direccionDeComienzo. Cuando lo encuentra se cambia el estado. Tener presente que el mismo bloque debería estar en dos arbolesAVL distintos, con dos criterios de ordenación distintos.