Código: 75.001

Asignatura: Estructura de la Información

Curso: 2009/2010 – primer semestre

PEC 2

Ejercicio 1 [2 puntos]

Apartado 1.1) Describid el funcionamiento de un algoritmo que permita fusionar dos árboles AVL de tamaño N y M respectivamente. ¿Qué coste tiene esta operación de fusión?

El algoritmo consiste en un bucle que, mediante un iterador preorden, por ejemplo, pueda recorrer el árbol de menor tamaño y va insertando los elementos en el otro árbol. Si el árbol más pequeño tiene N elementos, el bucle dará N vueltas. En cada vuelta ejecuta unas operaciones constantes (leer un elemento del primer AVL y asignarlo al segundo) y otras de coste logarítmico (buscar la posición que ha de ocupar el elemento y reestructurar el árbol resultante). En el peor de los casos, el coste asintótico será: O(N • log (N+M))

Apartado 1.2) Implementad en Java una extensión de la clase ArbolAVL de la biblioteca de clases para dotarla de un método que fusione el árbol actual con un árbol que recibís como parámetro. El método debe tener la siguiente firma:

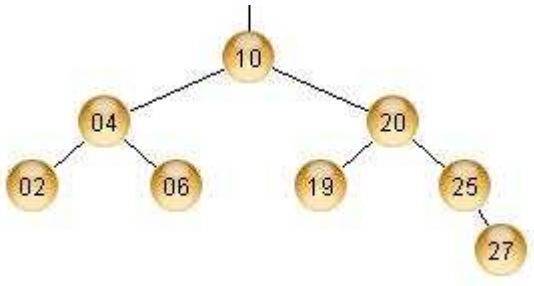
public void fusionar(ArbolAVL arbol)

```
public void fusionar(ArbolAVL<E> arbol)
{
	Recorrido<E> r = arbol.recorridoPreorden();
	while(r.haySiguiente())
	{
		this.insertar(r.siguiente().getElem());
	}
}
```

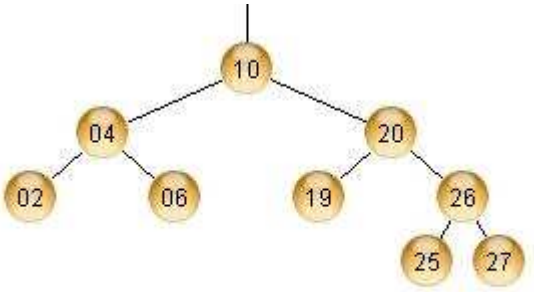
Apartado 1.3) Aplicad el algoritmo de fusión para fusionar los elementos del árbol X con el árbol Y y mostrad el árbol AVL resultado. Si la operación de fusión provoca desequilibrios comentad qué desequilibrio se produce y con qué rotación se soluciona.

Árbol X

Árbol Y



Aplicando el algoritmo de fusión solo se debe añadir el único nodo del árbol X al árbol Y. Se produce un desequilibrio DE y se obtiene el siguiente árbol AVL resultado:



Ejercicio 2 [2 puntos]

Apartado 2.1) Implementad en Java un TAD que os permita gestionar los empleados de una empresa multinacional con las siguientes premisas:

- No podéis usar delegación.
- Usad las clases de la JCF.
- El número de empleados es muy grande y conocido.
- Se requiere un acceso rápido a los datos del empleado a partir de su dni.
- Para cada empleado guardaremos los siguientes datos:
  - Dni
  - Nombre
  - Apellidos
  - Teléfono
- Debéis ofrecer operaciones para:
  - Dar de alta un empleado
  - Consultar un empleado por dni
  - Dar de baja un empleado
  - Consultar si un dni corresponde a un empleado
  - Listar todos los empleados
  - Listar todos los dni de los empleados
  - Consultar el número d'empleados o Consultar el número de empleados con un nombre determinado
  - Añadir un conjunto d'empleados

Con todos los datos que comenta el enunciado parece adecuado definir el TAD como una clase que extienda del TAD HasMap de la JCF puesto que:

- Elegimos un TAD Tabla porque todas las operaciones excepto la consulta del número de empleados con un nombre determinado las ofrece la interface Map de la JCF

- Elegimos un HashMap (implementación de tabla de dispersión) porque nos piden acceso rápido a los datos del empleado por clave y el número de empleados se muy grande pero conocido.

Definimos primero la clase Employee

```
public class Employee {
	private String dni;
	private  String name;
	private  String surname;
	private  String phone;

	public Employee(String dni, String name, String surname, String phone)
	{
		this.dni = dni;
		this.name = name;
		this.surname = surname;
		this.phone = phone;
	}

	/* Getters and setters */
	public String getDni() {
		return dni;
	}
	public String getName() {
		return name;
	}
	public String getSurname() {
		return surname;
	}
	public String getPhone() {
		return phone;
	}
	public void setDni(String dni) {
		this.dni = dni;
	}
	public void setName(String name) {
		this.name = name;
	}
	public void setPhone(String phone) {
		this.phone = phone;
	}
	public void setSurname(String surname) {
		this.surname = surname;
	}
}
```

y el TAD:

```
import java.util.HashMap;
public class MyTAD<TKey extends String, TValue extends Employee> extends HashMap<TKey,
TValue> {
	public int count(String name)
	{
		int ocurrences = 0;
		for(Employee employee : this.values())
		{
			if(employee.getName().equals(name))
			{
				ocurrences++;
			}
		}
		return ocurrences;
	}
}
```

Apartado 2.2) Implementad en Java un TAD que os permita gestionar los nombres de los empleados de una empresa multinacional con las siguientes premisas:

- No podéis usar herencia.
- Utilizad las clases de la biblioteca de clases de la asignatura.
- El número de empleados es grande pero desconocido.
- Se requiere una consulta eficiente.
- Debéis ofrecer operaciones para:
  - Dar de alta un nombre, si el nombre ya existe no hace nada
  - Consultar si un nombre existe
  - Eliminar un nombre
  - Listar todos los nombres
  - Consultar el número de nombres que empiecen por un carácter
  - Añadir un conjunto de nombres

Con todos los datos que comenta el enunciado parece adecuado definir el TAD como una clase que utilice las operaciones del TAD ConjuntoAVLImpl de la biblioteca de clases de la asignatura puesto que:

- Elegimos un TAD Conjunto porque todas las operaciones pedidas excepto las dos últimas las ofrecen los conjuntos y las claves y valores que debemos almacenar son los mismos.
- Escogemos un implementación basada en AVL porque nos piden una consulta eficiente pero como no sabemos el número de elementos no es recomendable utilizar una implementación basada en una Tabla.

```
import uoc.ei.tads.*;

public class MyTADConjunto {

	private ConjuntoAVLImpl<String> cjt = new ConjuntoAVLImpl<String>();

	public boolean estaVacio() {
		return cjt.estaVacio();
	}

	public void insertar(String elem) {
		cjt.insertar(elem);
	}

	public boolean esta(String elem) {
		return cjt.esta(elem);
	}

	public String borrar(String elem) {
		return cjt.borrar(elem);
	}

	public Iterador<String> elementos() {
		return cjt.elementos();
	}

	public int countFirstLetter(char letter)
	{
		int ocurrences = 0;
		for(Iterador<String> it = cjt.elementos(); it.haySiguiente();)
		{
```

```
                        if(it.siguiente().charAt(0) == letter)
                        {
                                ocurrences++;
                        }
                }
                return ocurrences;
        }

        public void insertarRango(Iterador<String> it)
        {
                while(it.haySiguiente())
                {
                        this.insertar(it.siguiente());
                }
        }
}
```

Ejercicio 3 [2,5 puntos]

Apartado 3.1) Suponed que hemos diseñado una función de dispersión que tiene como claves los nombres y apellidos de los estudiantes matriculados en la UOC; suponed también que entre nombre y apellidos se pueden ocupar 60 caracteres. Se define la función de dispersión de la siguiente manera:

$$h(c_1...c_{60}) = (\sum k: 1 \leq k \leq 60 : ascii(c_k) * 2^k) \bmod 256$$

Comenta los posibles inconvenientes de la función de dispersión escogida y proponed una alternativa que los solucione.

Los inconvenientes que se detectan en la función son los siguientes:

- Siempre da par con lo cual estamos desaprovechando la mitad de la tabla.
- En cualquier ordenador producirá desbordamiento.
- El valor de r se 256, la UOC tendrá bastantes más alumnos que 256
- Por todo lo anterior, h depende del aspecto de la cadena de entrada.
- Los valores generados no serán equiprobables, puesto que la función ascii toma en cuenta los 127 caracteres y no todos aparecen en los nombre y apellidos

Hay muchas soluciones posibles, una función de dispersión que soluciona los problemas mencionados podría ser la fórmula que tenéis a los apuntes del módulo cogiendo b=26 y r el número primero más próximo al valor supuesto de alumnos de la UOC:

$$h(c_1...c_{60}) = (\sum k: 1 \leq k \leq 60 : conv(c_k) * b^{k-1}) \bmod r$$

Apartado 3.2) ¿Es cierto que en un TAD tabla con una función de dispersión mal diseñada se puede degradar el rendimiento de la operación de búsqueda por clave hasta llegar a ser lineal respecto del número de elementos de la tabla? Poned un ejemplo de función de dispersión con este comportamiento.

Sí que es cierto, la función de dispersión utilizada en un TAD tabla es clave por conseguir un buen rendimiento en las operaciones de búsqueda, una mala función de dispersión puede degradar completamente el rendimiento de las búsquedas. Si vamos al extremo, una función de dispersión que coloque las entradas siempre a una posición fija de la tabla produciría un rendimiento lineal a las operaciones de búsqueda porque deberá recorrer toda la lista de sinónimos del elemento y esta contendrá todos los elementos de la tabla.

Apartado 3.3) Suponed que tenéis un TAD tabla y que no conseguís una función de dispersión que proporcione una distribución equiprobable de los elementos, cómo podríais mejorar la eficiencia de la búsqueda sin variar la función de dispersión?

Una posible manera de mejorar la eficiencia de las operaciones de búsqueda seria utilizar un árbol AVL para las listas de sinónimos, con esta solución pasaríais de una eficiencia lineal a logarítmica respeto al número de sinónimos en las búsquedas que provoquen colisión. Tened en cuenta pero que con esta implementación también estaríais penalizando las inserciones.

Ejercicio 4 [3,5 puntos]

Debemos implementar un prototipo para realizar votaciones por Internet experimentando un sistema electoral con listas abiertas. on este sistema los candidatos pertenecen a algún partido pero los electores no votan a un partido, sino a un candidato concreto.

Una vez realizadas las votaciones se hace la división entera entre el número total de votantes y los E escaños, obteniendo, así, la *cuota* necesaria para obtener un escaño. Todos los candidatos que están por encima de la *cuota* obtienen escaño, y los que están por debajo quedan provisionalmente fuera.

Si el proceso se acaba aquí es muy probable que no se ocupen todos los E escaños, puesto que seguramente algunos de los candidatos pueden haber superado la *cuota* y un candidato solo puede ocupar un escaño, aunque le hayan votado para ocupar uno o dos. Con el fin de ocupar todos los E escaños, se debe redistribuir este excedente de *cuota*, pero ¿a quién? Nosotros nos interesamos por la siguiente solución:

En el momento de votar se pide a los electores que ordenen los candidatos de más a menos prioritario. De esta manera, el excedente de votos de un candidato X se puede redistribuir de acuerdo con la voluntad de los electores. Normalmente, pero, los electores sólo ordenan unos pocos candidatos (empíricamente se sabe que con 10 candidatos ya es suficiente por garantizar el proceso).

El procedimiento de redistribución de excedente es el siguiente: Para cada candidato C con excedente, se divide el excedente entre sus votantes NV, obteniendo el excedente proporcional (un valor decimal). Es decir, cada votante del candidato C "gasta" una parte para el candidato C y, el resto, la dedica al siguiente candidato que haya elegido. Cada vez que se redistribuye el excedente de un candidato, este pasa al grupo de candidatos seleccionados y ya no intervendrá más en el proceso de redistribución.

Este procedimiento se va repitiendo hasta que desaparece el excedente de todos los candidatos. Por ejemplo, si suponemos que tenemos 100 votos, 4 escaños, 5 candidatos (C1, C2, C3, C4 y C5) y que a las votaciones C1 obtiene 50 votos, C2 obtiene 35 y C3 obtiene 15 los otras dos no obtienen ningún voto. Con estos datos tenemos:

- Cuota: 25 (100 votos /4 escaños)

- C1 tiene un excedente de cuota de 25 (50-25) y un excedente proporcional de (25/50) = 0,5. El 50% del voto de los electores del candidato C1 se acumula a su segunda opción.

- C2 tiene un excedente de cuota de 10 (35-25) y un excedente proporcional de (10/35) = 0,28. El 28% del voto de los electores del candidato C2 se acumula a su segunda opción.

- C3, C4 y C5 no tienen excedente de cuota.

El procedimiento de redistribución se haría de la siguiente forma:

- Elegir uno de los candidatos con excedente (suponemos Cx).
- Recorrer todos los votos del candidato y, para cada uno de los votos:
    - Elegir el primer candidato del voto (recordad que en el momento de la votación se pide a los electores que ordenen de mayor a menor loas candidatos a los que dan su voto) que no haya entrado todavía en el procedimiento de redistribución (suponemos que es Cy)
    - Añadir el voto a Cy (actualizando la fracción del voto con el excedente proporcional de Cx)
    - Actualizar el número de votos de Cy (suma de las fracciones de todos los votos de Cy)
    - Comprobar si Cy supera la cuota
- Actualizar el número de votos de Cx (ahora serán igual a la cuota porqué hemos redistribuido su excedente)
- Marcar que el candidato Cx ya ha entrado en el procedimiento de redistribución.

Una vez redistribuidos todos los excedentes, empezaría una segunda etapa de eliminación de candidatos hasta quedar E (el número de escaños), pero por el momento no implementaremos esta segunda etapa.

Para la resolución del ejercicio hacemos las siguientes consideraciones:

- El número de partidos P es pequeño y conocido.

- El número de candidatos C es relativamente grande y conocido (del orden de miles).
- El número de votantes V es desconocido (depende del grado de participación) y puede ser muy grande (del orden de millones).
- El número de escaños E es pequeño y conocido.

Se quiere diseñar un TAD Votaciones con las operaciones siguientes:

1) crear(). Crear la estructura, inicialmente vacía.
2) anadirPartido(Partido). Añade un partido a la estructura.
3) anadirCandidato(Candidato, Partido). Añade un candidato asociado a un partido.
4) anadirVotante(Votante, cola<Candidato>). Registra un votante con su lista de preferencias y actualiza el total de votos del candidato de primera opción. Si el votante ya había votado, da un mensaje d'error. Se supone que la lista de candidatos es correcto.
5) finalizarVotaciones(): Cierra las votaciones y prepara la estructura para el proceso de redistribución.
6) quedanExcedentes(). Devuelve un booleano indicando si quedan candidatos con excedente >0 .
7) redistribuirExcedente(): Redistribuye el excedente del candidato con más excedente.
8) listadoCandidatosSeleccionados(). Devuelve un iterador para recorrer los candidatos seleccionados. No importa el orden. Inicialmente esta lista está vacía y se va ampliando a cada paso del proceso de redistribución.
9) partidoMasVotado(): Partido devuelve el partido más votado.

Requisitos de eficiencia:

- Las operaciones 1, 2, 3 i 5 no deben ser especialmente eficientes.
- Las operaciones 4, 6 i 7 deben ser el máximo de eficientes posible.

Os pedimos lo siguiente:

Apartado 4.1) Realizad un dibujo de la estructura de datos resultante, que deje claras las partes que la componen mediante las representaciones gráficas vistas en la asignatura. Podéis poner una breve descripción (dos o tres líneas) de cada componente de la estructura (por ejemplo, la estrategia de representación: secuencial, encadenada por vectores, encadenada indirecta, etc.). Debe quedar claro qué es la información contenida en cada uno de estos componentes.

Apartado 4.2) Estudiad la eficiencia de las operaciones anadirVotante() y redistribuirExcedente(). Concretamente, para cada operación, debéis describir brevemente su comportamiento indicando los pasos que la componen (con frases como por ejemplo:

“insertar en el árbol AVL / borrar de la tabla de dispersión / consulta del piló / ordenar el vector...”), comentando la eficiencia asintótica de cada paso y dando la eficiencia total de la operación.

Usaremos una lista acotada para guardar los P partidos que se presentan a las elecciones. Sabemos que P es pequeño y conocido y, por lo tanto, una lista acotada se ajusta a las necesidades, puesto que el tiempo de consulta de un partido será despreciable (debido a la medida pequeña de P).

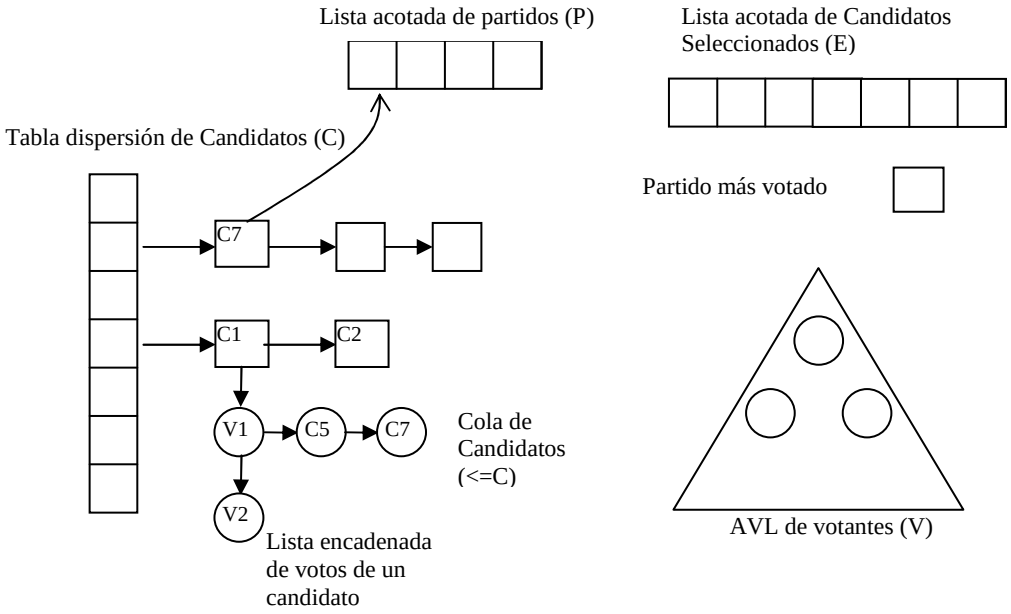
Podríamos utilizar una lista encadenada para guardar los candidatos de un partido, pero en el enunciado no hay ningún método que requiera esta información y, por lo tanto, descartamos esta estructura. En cambio, para cada candidato si que guardaremos una referencia al partido al que pertenece (para poder contabilizar los votos de cada partido).

El número de candidatos es elevado y conocido. Esto nos permite utilizar una tabla de dispersión, puesto que no necesitamos ningún recorrido ordenado de candidatos. Además, el bajo coste de las consultas nos permitirán optimizar el método añadir votante.

Para añadir una votación se deberá verificar que el votante no lo haya hecho con anterioridad. A tal efecto utilizaremos un AVL para localizar los votantes rápidamente. No podemos utilizar una tabla de dispersión puesto que el número de votantes es desconocido. Una vez verificado ya podremos añadir la votación al candidato. Para cada candidato X utilizaremos una lista encadenada con todas las votaciones. Utilizamos una lista encadenada puesto que, ni que decir tiene, desconocemos el número de votaciones que tendrá cada candidato, y no necesitamos ninguna operación especial. Esta lista la utilizaremos para redistribuir los excedentes de los candidatos. Sin esta lista tendríamos que hacer un recorrido de todas las votaciones por localizar las del candidato con excedente, cosa que dispararía el método 7.

Cada votación consistirá de una cola de candidatos, de acuerdo con la elección del votante. Cada vez que asignamos una parte del voto a algún candidato, este desaparece de la cola. Esto nos permite liberar espacio que no necesitamos para nada más. Además, debemos saber qué fracción del voto hemos asignado al candidato actual. Inicialmente asignaremos todo el voto al primer candidato, pero en cada redistribución esta fracción se irá haciendo pequeña.

Durante el proceso de redistribución de excedentes debemos falta localizar rápidamente los candidatos con excedente. Proponemos la utilización de una estructura acotada al número de escaños E para guardar los Candidatos que han superado la cuota. Esta estructura se inicializa durante el método finalizarVotaciones haciendo un recorrido por todos los candidatos y añadiendo los que hayan superado la cuota. En cada paso del proceso de redistribución se escoge uno de los candidatos y se marca por evitar que se le aplique una segunda redistribución. Cada vez que un candidato supera la cuota a resultas de una redistribución, este candidato se añade a la estructura. Como se trata de una estructura de medida pequeña E, no perjudicaremos la eficiencia puesto que las operaciones tendrán un coste aproximado O(1). El siguiente dibujo muestra la estructura que hemos descrito:



Estudio de eficiencia de las operaciones pedidas:

4) anadirVotante(Votante, Cola<Candidato>)

- Buscar Votante en AVL de votantes --> O(log V)
- Buscar el primer candidato en la tabla de dispersión de candidatos --> O(1)
- Incrementar (+1) el contador de votos al primer candidato -->O(1)
- Crear un nuevo Voto con la cola de candidatos (eliminando el primero) e inicializando la fracción a 1 . --> O(1)
- Añadir un voto a la lista encadenada de votos del candidato --> O(1)
- Incrementar el contador de votos del partir del primer candidato, y actualizar el partido más votado (si procede) -->O(1)
- **Total: O(logV) -> O(logV)**

7) redistribuirExcedente().

- Acceder a un candidato X con excedente -> O(E)≈O(1)
- Calcular el excedente proporcional -> O(1)
- Hacer un recorrido para cada votación del candidato -> O(VC)
    - Localizar el siguiente candidato Y comprobando que no haya pasado ya por el proceso de redistribución--> O(E) ≈O(1)
    - Sacar el voto de la lista de X y añadirlo a la lista de votos de Y--> O(1)
    - Eliminar el candidato Y de la cola de preferencias del voto -->O(1)
    - Actualizar la fracción del voto utilizando el excedente proporcional de X--->O(1)
    - Incrementar los votos de Y de acuerdo con la nueva fracción --> O(1)
    - Si el candidato ha superado la cuota gracias a esta fracción, lo añadimos a la lista de candidatos seleccionados --> O(1)

- Actualizar los votos del candidato X (pasan a ser el valor de la cuota) --> O(1)
- Marcar el candidato X por no entrar más en el proceso de redistribución --> O(1)
- **Total: O(VC+E) ≈O(VC)**