PRACTICO N° 11
Colecciones y Tablas

- *En todas las implementaciones puede agregar métodos para favorecer la modularización.*
- *En la página de la materia, en el apartado "Material Adicional" puede encontrar implementaciones parciales de algunas clases necesarias para la realización de este práctico.*

EJERCICIO 1. La clase **NominaEmpleados** modelada en el diagrama permite almacenar información referida a los empleados de una empresa. Los datos se almacenan en un arreglo parcialmente ocupado. El atributo **cant** mantiene la cantidad de elementos ocupados en el arreglo.
Cuando se produce un **alta** de un Empleado se agrega un elemento en la posición cant y cant se incrementa en 1. Se asume que la clase cliente controló que la estructura no está completamente ocupada.
Cuando se produce la **baja** de un Empleado, el último elemento reemplaza al elemento dado de baja, de modo que en el arreglo queden siempre ocupadas las cant primeras componentes. Esto es, los elementos que corresponden a subíndices 0 a cant-1 son referencias ligadas y las que siguen deben ser referencias nulas.
El comando ***ordenar*** reacomoda los elementos de la nómina de empleados de modo tal que el arreglo quede ordenado por legajo hasta que se realicen nuevas altas o bajas. El siguiente algoritmo muestra el método de ordenamiento:

```
para i entre 0 y n-1
  Buscar el empleado con menor legajo entre las posiciones i  y n-1
  Ubicarlo en la posición i
```

Empleado
<<atributos de instancia>> legajo: entero nombre:String cantHoras: entero valorHora: real
<<Constructor>> Empleado(leg:entero,nombre:String; canth: entero, valorh:real) **<<Consultas>>** obtenerLegajo():entero obtenerSueldo():real obtenerCantHoras():entero obtenerValorHoras():real obtenerNombre():String

NominaEmpleados
<<atributos de instancia>> nomina: Empleado[] cant: entero
<<Constructor>> NominaEmpleados(max:entero) **<<Comandos>>** alta (emp:Empleado) baja (leg:entero) ordenar() **<<Consultas>>** cantActualEmpleados():entero cantMaximaEmpleados():entero buscarEmpleado(leg:entero):Empleado existeEmpleado (emp:Empleado):boolean existeEmpleado (leg:entero):boolean existeEmpleado(nom:String):boolean contarSupHoras (h:entero):entero sumarSueldos (): real

contarSupHoras (h:entero) cuenta la cantidad de empleados que trabajan más de *h* horas

a. Implemente la clase **NominaEmpleados** propuestas en el diagrama.
b. Diseñe casos de prueba adecuados para verificar cada uno de los servicios.
c. Implemente una clase tester.

EJERCICIO 2. La clase TablaEspecies permite representar la información referida a las especies observadas por un instituto de investigación (la clase Especie es la implementada en el práctico 5). La tabla se representa en un arreglo parcialmente ocupado con n elementos. La clase cliente es responsable de decidir en qué posición se insertan los elementos y puede recorrer la estructura usando el servicio ***obtenerEspecie***. Si ya existe un elemento en la posición p, se sobrescribe el nuevo. Si se **elimina** un elemento, la posición queda nula. Si la estructura no está completa, las componentes vacías pueden estar intercaladas con las ocupadas. El comando ***comprimir*** arrastra los elementos de modo tal que todas las componentes vacías queden al final. El comando ***ordenar*** comprime la estructura y luego ordena las componentes ocupadas de acuerdo al ritmo de crecimiento, si dos especies tienen el mismo ritmo las ordena por nombre. El método de ordenamiento es el de la burbuja.

Dada una tabla T, el **método de ordenamiento Burbuja** (ascendente) consiste en comparar cada elemento T_i con T_{i+1} e intercambiándolos si $T_i > T_{i+1}$ con $0 <= i < n-1$. El proceso se repite hasta que todos los elementos están ordenados. Es decir, se compara el primero elemento con el segundo y si no están ordenados se intercambian; se compara el segundo con el tercero y si no están ordenados se intercambian y así siguiendo comparando todos los pares hasta comparar los dos últimos. Cuando todos los pares fueron comparados una vez, el último elemento es el mayor y el proceso debe repetirse sin considerar el último elemento que ya está ordenado.

TablaEspecies

<<atributos de instancia>>
T: Especie[]
n: entero

<<Constructor>>
TablaEspecies(n:entero)
<<Comandos>>
insertar(e: Especie, p: entero)
eliminarPosicion(p: entero)
eliminar(e: Especie)
eliminar(c:entero)
comprimir()
ordenar()
intercalar(t:TablaEspecies):TablaEspecies
<<Consultas>>
cantEspecies(): entero
obtenerEspecie(p: entero): Especie
existeEspecie(e: Especie): boolean
contarMasHembras(): entero
ritmoSuperior (s: real): TablaEspecies
crecienteSuperior (s:real) : boolean
hayMEspeciesSuperior(s:real,m:entero): boolean

Se asume que la posición p se controla siempre en la clase Cliente

Especie

<<atributos de instancia>>
nombre: String
hembras: entero
machos: entero
ritmo: real

<<constructor>>
Especie(nom: String)
<<comandos>>
asignarHembras(h: entero)
asignarMachos(m: entero)
asignarRitmo(r: real)
<<consultas>>
poblacionActual(): entero
poblacionEstimada(anios: entero) : entero
riesgo(): String
masHembras(): boolean
mayorRitmo(est: Especie): Especie
obtenerRitmo():real

- ***eliminar(c:entero)*** Elimina todas las especies que tienen menos de c individuos
- ***contarMasHembras(): entero*** Cuenta en cuántas especies hay más hembras que machos.
- ***ritmoSuperior (s: real): TablaEspecies*** Genera una tabla con las especies con ritmo superior a r.
- ***crecienteSuperior (s: real):boolean*** retorna verdadero si siempre se cumple que entre dos especies consecutivas la diferencia en valor absoluto entre los ritmos de crecimiento es mayor a s.
- ***hayMEspeciesSuperior(s:real,m:entero):boolean*** decide si hay al menos m especies con un ritmo superior a r.
- ***intercalar(t:TablaEspecies): TablaEspecies*** Genera una tabla de especies resultado de intercalar ordenadamente la tabla que recibe el mensaje y la tabla de especies pasada por parámetro. Se asume que el cliente se ha encargado de que tanto la tabla de especies que recibe el mensaje como la pasada por parámetro han sido ordenadas y comprimidas antes de invocar este mensaje.

a. Implemente la clase **TablaEspecies**, analizando particularmente la eficiencia de las soluciones.
b. Escriba un algoritmo para el método *ordenar* (implementando el método de ordenamiento Burbuja).
c. Optimice el método *ordenar* considerando que si en un recorrido completo no hay intercambios, los elementos están ordenados.
d. Optimice el método *ordenar* considerando que si en un recorrido completo el último intercambio se produjo en la posición p, todos los elementos que siguen a p están ordenados.
e. Implemente una clase que verifique los servicios de TablaEspecies usando casos de prueba adecuados.
f. Agregue un método en la clase tester que permita mostrar los nombres de las Especies cuya población estimada para dentro de 10 años sea menor que la actual.

Algoritmo Intercalar

DE L1, L2 dos colecciones ordenadas
DS L3
    i1=0 i2=0
    Mientras i1 < fin L1 y i2 < fin L2
      si $L1_{i1}$ es menor que $L2_{i2}$
        agregar $L1_{i1}$ al final de L3
        i1++
      sino
        agregar $L2_{i2}$ al final de L3
        i2++
    si i1 < fin L1 agregar el resto de las componentes de L1 en L3
sino agregar el resto de las componentes de L2 en L3.

EJERCICIO 3. La clase ColeccionPoligonos modelada en el diagrama permite almacenar la representación de una **colección** de polígonos. Cada vez que se inserta un polígono en la colección se incrementa el valor de n. Si se almacenan n polígonos, se ocupan las primeras n componentes del arreglo. Cuando se elimina un polígono de la colección, los elementos se "arrastran" para seguir estando comprimidos. Cuando un servicio elimina varios, se arrastra una sola vez. Desde la clase cliente de la Colección no se acceden a los polígonos por posición.

Cada polígono de n lados se representa como una colección de n puntos. El servicio **obtenerPunto** retorna el punto que ocupa la posición p en la secuencia, comenzando desde la posición 1. Cada lado del polígono queda determinado por los elementos que ocupan las posiciones i e i+1 en la secuencia, excepto uno de los lados que se forma con los puntos que ocupan la posición 1 y n. Cada vez que se inserta un punto en un polígono, se incrementa el valor de n. Si se almacenan n puntos, se ocupan las primeras n componentes del arreglo. Cuando se elimina un punto del polígono, los elementos se "arrastran" para seguir estando comprimidos y en la **misma secuencia** que fueron ingresados. La clase cliente es responsable de controlar que no se inserte dos veces el mismo punto.

Punto
<<atributos de instancia>> x: real y: real
<<Constructor>> Punto(X,Y:real) **<<Comandos>>**

Poligono
<<atributos de instancia>> p: Punto [] n: entero
<<Constructor>> Polígono(max:entero) **<<Comandos>>**

ColecciónPoligonos
<<atributos de instancia>> cp: Poligono[] n: entero
<<Constructor>> ColeccionPoligonos(max:entero) **<<Comandos>>**

establecerX (v:real) establecerY (v:real) **<<Consultas>>** obtenerX ():real obtenerY ():real distancia(p:Punto):real

insertar (p:Punto) eliminar(p:Punto) **<<Consultas>>** cantPuntos():entero estaLlena():boolean obtenerPunto(p:entero):Punto existePunto (p:Punto):boolean perimetro():real

insertar (p:Poligono) eliminar(p:polígono) eliminarPoligonosNLados(n:entero) **<<Consultas>>** cantPoligonos():entero estaLlena():boolean existePoligono (pol:Poligono):boolean contarPoligonosPunto(p: Punto): entero mayorPerimetro():real

- ***eliminarPoligonosNLados(n:entero):*** Elimina de la colección de polígonos todos los polígonos que tengan N lados, dejando el arreglo comprimido.

a. Implemente las clases **Poligono y ColecciónPoligonos** propuestas en el diagrama.
b. Diseñe casos de prueba adecuados para verificar los servicios
c. Implemente una clase tester.

EJERCICIO 4. En un videojuego cada uno de los personajes tiene una misión asignada. Cada misión está formada por una **secuencia de acciones**. Toda acción tiene un **nombre** y una **duración**.
Implemente el siguiente modelo parcial para la clase **Mision** del videojuego:

Mision
<<Atributos de instancia>> m : Accion[] cant: entero
Mision(max:entero)
<<Consultas>> **subsecuenciaMision (s:String,t:entero): Mision**

Accion
nombre:String duración:real
Accion(n : String, d : real)
<<Consultas>> obtenerNombre():String obtenerDuracion(): real igualDuracion(a:Accion):boolean mayorDuracion(a:Accion):boolean igual(a:Accion)

La clase **Accion** brinda los métodos triviales.
El método subsecuenciaMision retorna, si existe, una Misión con una subsecuencia de acciones dentro de la misión que recibe el mensaje, que comienza con una acción de nombre s y continúa con la **mayor cantidad de acciones posibles**, que tengan una duración total de **t** unidades de tiempo. Si hay más de una subsecuencia con estas características, retorna la primera. Si no hay ninguna, retorna nulo.

Por ejemplo si el nombre de la acción s es Caminar y t es 40 y la misión está formada por los siguientes objetos de la clase Acción:

Saltar	20
Trepar	20
Correr	12
Caminar	10
Saltar	25
Trepar	30
Caminar	20
Trepar	20
Comer	10

Beber	12
Caminar	8
Caminar	**10**
Saltar	**14**
Luchar	**16**
Saltar	11

secuenciaMision debe retornar la Misión formada por estos objetos:

Caminar	10
Saltar	14
Luchar	16

*Porque la secuencia comienza con Caminar, tiene una duración total de 40 unidades de tiempo y es la **subsecuencia más larga** que comienza con Caminar y tiene duración 40.*

EJERCICIO 5. En un hospital se mantiene información referida a las cirugías de acuerdo al siguiente diagrama de clases:

Cirugía
<<atributos de instancia>> codigo: entero pac : Paciente fecha : Fecha cirujano : String minutosDuracion : entero quirófano : entero
<<consultas>> igual(c:Cirugia):boolean menor(c:Cirugia):boolean

ColeccionOrdCirugias
<<atributos de instancia>> cirugías : Cirugía[] cantCirugía : entero
<<Comandos>> insertar(c: Cirugía): boolean agregar(c:Cirugia) eliminarOS(os: String) **<<Consultas>>** existeCirugia(c:Cirugía):boolean promedioDuracion(): real promedioDuracion(desde,hasta: Fecha): real promedioDuracion(edad: entero): real cirugiasCirujano(c: String): ColeccionOrdCirugias cirugíasOS(os: String): ColeccionOrdCirugias cirugiasPac(Pac: Paciente): entero intercalar(ColeccionOrdCirugias C): ColeccionOrdCirugias

*En la clase **Paciente**,* la consulta ***igual*** retorna verdadero si los pacientes tienen el mismo valor en el atributo nombre. Los métodos igual y menor en la clase ***Cirugía*** comparan por código.
*En la clase **ColecciónOrdCirugias:***
insertar(c: Cirugía) inserta, si hay lugar, un elemento ordenado por código, incrementando la cantidad.
agregar(c:Cirugia) inserta un elemento al final e incrementa cantidad.
eliminarOS(os: String) elimina todos los elementos que correspondan a la obra social dada. Actualiza la cantidad y comprime.
promedioDuracion() calcula el promedio de duración de todas las cirugías almacenadas
promedioDuracion(desde,hasta: Fecha) calcula el promedio de todas las cirugías correspondientes al período dado
promedioDuracion(edad: entero) calcula el promedio de todas las cirugías efectuadas a pacientes mayores a una edad dada
cirugiasCirujano(c: String) genera una estructura con las cirugías realizadas por un cirujano dado
cirugíasOS(os: String) genera una estructura con las cirugías asociadas a una Obra Social dada.
cirugiasPac(Pac: Paciente) cuenta la cantidad de cirugías efectuadas a un paciente dado
El método ***existeCirugía*** se implementa usando el siguiente algoritmo de *búsqueda binaria*:

Algoritmo BusquedaBinaria

DE ini, fin, buscado, L (colección ordenada)
DS existe

```
si ini > fin
  existe ← falso
sino
 mitad←(ini+fin)/2
 si buscado es igual a L_mitad
  existe ← verdadero
 sino
  si buscado < L_mitad
    existe ←BusquedaBinaria (ini,mitad-1,buscado)
  else
    existe ← BusquedaBinaria( mitad+1,fin,buscado)
```

Implemente la clase **ColeccionOrdCirugias** del diagrama de clases incluyendo los servicios triviales.

a. Implemente la clase **ColeccionOrdCirugias** propuesta en el diagrama.
b. Diseñe casos de prueba adecuados para verificar los servicios.

EJERCICIO 6. Dado el siguiente diagrama de clase:

Conjunto
<<atributos de instancia>> CE: Racional[] cantElementos: entero
<<constructor>> Conjunto(max: entero) **<<Comandos>>** insertar(e: Racional) **<<Consultas>>** pertenece (e: Racional): boolean vacio():boolean incluido(c: Conjunto) : boolean equals(c: Conjunto) : boolean disjunto(c :Conjunto) : boolean union(c: Conjunto) : Conjunto interseccion(c: Conjunto): Conjunto diferencia(c: Conjunto): Conjunto cardinalidad(): entero toString(): String

Racional
<<atributos de instancia>> num: entero den : entero
<<constructor>> Racional(n,d : entero) **<<Comandos>>** ... **<<Consultas>>** equals(r:Racional) : boolean

a. Implemente la clase **Conjunto** completo. Considere que inicialmente el conjunto está vacío y que la operación insertar no tiene ningún efecto si el elemento a insertar e ya pertenece al conjunto.
b. Verifique la clase **Conjunto** con el siguiente tester:

```
class testerCE {
  public static void main (String args[]){
      Conjunto c1,c2,c3,c4;
      Racional r1,r2,r3;

      r1 = new Racional (1,2); r2 = new Racional (1,3); r3 = new Racional(1,2);
      c1 = new Conjunto (4); c2 = new Conjunto (4); c3 = new Conjunto (4);

      c1.insertar(r1); c1.insertar(r2); c1.insertar(r3); c1.insertar(new Racional(1,2));
      c2.insertar(r2); c2.insertar(r1); c2.insertar(r3); c2.insertar(new Racional(1,2));
      c3.insertar(r1); c3.insertar(r2); c3.insertar(new Racional(2,5)); c3.insertar(new Racional(3,2));
      c4 = c3.diferencia(c1);
      c5 = c4.interseccion(c1);

      System.out.println(c1.toString());
      System.out.println(c2.toString());
      System.out.println(c3.toString());
      System.out.println(c4.toString());
      System.out.println(c5.toString());
      System.out.println ("r1 pertenece c1 ?"+c1.pertenece(r1));
      System.out.println ("r3 pertenece c3 ?"+c3.pertenece(r3));
      System.out.println ("c1 = c2 ?"+c1.equals(c2));
      System.out.println ("c1 = c3 ?"+c1.equals(c3));
      System.out.println ("c1 y c4 son disjuntos? "+ c4.disjunto(c1) + "c5 es vacia? "+ c5.vacio());
      }
}
```

c. Extienda la clase tester con casos de prueba que permitan verificar los demás servicios.

EJERCICIO 7. Dado el siguiente diagrama de clase:

ConjuntoOrdenado
<<atributos de instancia>> CE: Criatura[] cantElementos: entero
<<constructor>> Conjunto(max: entero) **<<Comandos>>** insertar(e: Criatura) **<<Consultas>>** pertenece (e: Criatura) : boolean vacio():boolean incluido(c: Conjunto) : boolean equals(c: Conjunto) : boolean disjunto(c :Conjunto) : boolean union(c: Conjunto) : Conjunto interseccion(c: Conjunto) : Conjunto diferencia(c: Conjunto) : Conjunto cardinalidad() : entero toString(): String

Criatura
<<atributos de instancia>> nombre:String energía: entero
<<constructor>> Criatura (nom:Nombre) **<<Comandos>>** jugar() descansar() **<<Consultas>>** obtenerNombre(): String obtenerEnergia(): entero igual(c:Criatura) : boolean mayor(c:Criatura): boolean

Cuando se crea una **Criatura** arranca con 100 de energía. Cuando una criatura juega consume 10 unidades de energía, siempre que tenga energía para hacerlo. Su energía nunca es negativa. Cuando descansa vuelve su energía a 100.
Implemente la clase **ConjuntoOrdenado** completa. Considere que inicialmente el conjunto está vacío.
El servicio **insertar** agrega un nuevo elemento ordenado x nombre en forma ascendente y no tiene ningún efecto si el elemento a insertar e ya pertenece al conjunto.
Los servicios igual y mayor comparan las criaturas por nombre.

Implemente la clase ConjuntoOrdenado diseñando soluciones eficientes.