Programación I

Práctica N°2 – Estructuras de datos

Notas preliminares

- No se aceptarán soluciones que no compilen.
- Los ejercicios marcados con ★ son de entrega **obligatoria**.
- Las entregas son **en grupos de hasta dos personas**.
- La entrega de los ejercicios se hará en papel (imprimiendo los .java que se entregan) y por mail, en un archivo .zip con los archivos java correspondientes.

---

1. Listas

Dada la clase ListaInt:

```
class NodoInt
{
  int elemento;
  NodoInt siguiente;
}

class ListaInt
{
  NodoInt primero;
}
```

Ejercicio 1

a) Escribir el método de instancia **int** `largo`() que devuelve el largo de la lista.
b) Escribir el método de instancia **boolean** `estaVacia`() que informa si la lista está o no vacía. Esta función debe tener un orden de complejidad constante ($O(1)$).
c) Escribir el método de instancia **int** `suma`() que devuelve la suma de los elementos de la lista. La suma de la lista vacía vale 0.
d) Escribir el método de instancia **double** `promedio`() que devuelve el promedio de los elementos de la lista. El promedio de la lista vacía vale 0.
e) Escribir el método de instancia **int** `iesimo`(**int** `i`) que toma como parámetro una posición y devuelve el elemento que se encuentra en dicha posición de la lista. Por ejemplo, `iesimo`(1) de [2,5,6] devuelve 5.
REQUIERE: `0 <= i < largo()`.
f) Escribir el método de instancia **int** `maximo`() que devuelve el máximo valor de la lista.
REQUIERE: ¬ `vacia()`.
g) Escribir el método de instancia **boolean** `estaOrdenada`() que informa si la lista tiene a sus elementos ordenados de menor a mayor. La lista vacía siempre está ordenada.

h) Escribir el método de instancia **boolean** `esSinDuplicados`() que informa si la lista no tiene elementos repetidos.

Ejercicio 2 ★

Expandir la clase ListaInt con los siguientes métodos. Se pide resolver cada problema **sin acudir a la creación de listas auxiliares**, es decir, exclusivamente manipulando los nodos existentes (o creando a lo sumo uno nuevo cuando haga falta).

a) Escribir el método de instancia **void** `rotarDerecha`() que modifica la lista de modo que el último elemento (si es que tiene alguno) pasa a ser el primero. Por ejemplo: rotarDerecha sobre [1,2,3,4] modifica a la lista de modo que queda [**4**,1,2,3]. Notar que si la lista está vacía o tiene un sólo elemento, `rotarDerecha`() no la modifica.

b) Escribir el método de instancia **void** `agregarEnPosicion`(**int** `pos`, **int** `elem`) que toma una posición `pos` y un elemento `elem` e inserta un nuevo nodo con dicho elemento en la posición especificada. Agregar en la posición 0, es equivalente a agregar el elemento como primer elemento de la lista, y agregar en la posición dada por `largo()` es equivalente a agregarlo al final de la lista. Por ejemplo,

- `agregarEnPosicion(0,15)` sobre [5,10,20] resulta en [15,5,10,20].
- `agregarEnPosicion(2,15)` sobre [5,10,20] resulta en [5,10,15,20].
- `agregarEnPosicion(3,15)` sobre [5,10,20] resulta en [5,10,20,15].

REQUIERE: `0 <= i <= largo()`.

c) Escribir el método de instancia **void** `insertarOrdenado`(**int** `e`) que agrega un nodo a la lista conteniendo el elemento pasado como parámetro en la posición que corresponda de modo que la lista siga estando ordenada. REQUIERE: la lista está ordenada.

d) Escribir el método de clase
static void `intercambiarColas`(`ListaInt l1`, **int** `pos1`, `ListaInt l2`, **int** `pos2`) que corta la lista `l1` a partir de `pos1` y `l2` a partir de `pos2` e intercambia las colas de ambas listas a partir de dichas posiciones. Por ejemplo,

- `intercambiarColas` sobre `l1`=[2,4,6,8] y `l2`=[1,3,5,7] con `pos1`=2 y `pos2`=2 resulta en: `l1`=[2,4,5,7] y `l2`=[1,3,6,8]
- `intercambiarColas` sobre `l1`=[2,4,6,8] y `l2`=[1,3,5,7] con `pos1`=1 y `pos2`=3 resulta en: `l1`=[2,7] y `l2` =[1,3,5,4,6,8]
- `intercambiarColas` sobre `l1`=[2,4,6,8] y `l2`=[1,3,5,7] con `pos1`=0 y `pos2`=0 resulta en: `l1`=[1,3,5,7] y `l2`=[2,4,6,8]

Ejercicio 3 ★

Expandir la clase ListaInt con los siguientes métodos. Reutilizar los métodos que ya estén a disposición cuando sea posible. En los métodos que trabajen sobre más de una lista **se debe evitar generar aspectos de aliasing entre éstas**.

a) Escribir el método de instancia `ListaInt buscarTodos(int n)` que toma un entero y devuelve una nueva lista que contiene las **posiciones** en las que aparece dicho entero en la lista ordenadas de menor a mayor.
b) Escribir el método de instancia **void** `anexar(ListaInt otraLista)` que agrega al final de esta lista todos los elementos de la lista recibida como parámetro.
c) Escribir el método de clase **static** `ListaInt concatenar(ListaInt l1, ListaInt l2)` que toma como parámetros dos listas y crea una nueva que tiene los elementos de la primera lista, seguidos de los de la segunda.
d) Escribir el método de instancia `ListaInt reversa()` que devuelve una nueva lista que tiene los mismos elementos de la lista actual, pero en orden inverso.
e) Escribir un método de clase que tome como parámetros dos listas **que ya se encuentran ordenadas de menor a mayor** y devuelva una nueva lista con los elementos de ambas listas ordenados de menor a mayor. Se pide escribir un método cuya complejidad temporal de peor caso sea $O(n+m)$, donde n y m son los tamaños de las dos listas recibidas. El método debe tener la siguiente signatura: **static** `ListaInt combinarListasOrdenadas(ListaInt l1, ListaInt l2)`. Requiere: `l1.estaOrdenada() && l2.estaOrdenada()`.

Ejercicio 4

Expandir la clase ListaInt con los siguientes métodos. Reutilizar los métodos que ya estén a disposición cuando sea posible.

a) Escribir el método de instancia `ListaInt dameElementosEnPosiciones(ListaInt pos)` que toma una lista **ordenada y sin repetidos** de posiciones y devuelve una nueva lista con los elementos de la lista original que se encuentran en dichas posiciones. Se pide escribir un método cuya complejidad temporal de peor caso sea $O(n)$, donde n es el largo de la lista original.
Requiere: `pos.estaOrdenada() && pos.esSinRepetidos()`
b) Escribir el método de clase **static** `ListaInt interseccion(ListaInt l1, ListaInt l2)` que toma dos listas sin repetidos y devuelve una nueva lista conteniendo los elementos que están presentes en ambas listas. El método no debe modificar las listas recibidas y la lista resultado no debe tener repetidos tampoco.
c) Escribir el método de clase **static** `ListaInt resta(ListaInt l1, ListaInt l2)` que toma dos listas sin repetidos y devuelve una nueva lista conteniendo los elementos que están presentes en `l1` pero no en `l2`. El método no debe modificar las listas recibidas y la lista resultado no debe tener repetidos tampoco.
d) Escribir el método de clase **static** `ListaInt restaSimetrica(ListaInt l1, ListaInt l2)` que toma dos listas sin repetidos y devuelve una nueva lista conteniendo tanto los elementos que están presentes en `l1` pero no en `l2` como los elementos que están en `l2` pero no en `l1`. El método no debe modificar las listas recibidas y la lista resultado no debe tener repetidos tampoco.

2. TADs: Pilas

Ejercicio 5

Dada la clase PilaInt, con las operaciones:

- **void** `apilar`(**int** `n`)
- **void** `desapilar`() [REQUIERE: `!estaVacia()`]
- **boolean** `estaVacia`()
- **int** `tope`() [REQUIERE: `!estaVacia()`]

a) Escribir un método de clase que tome como parámetros dos pilas **que ya se encuentran ordenadas con el menor elemento en el tope** y devuelva una nueva pila con los elementos de ambas pilas ordenados de menor a mayor (con el menor elemento en el tope). El método debe tener la siguiente signatura: **static** `PilaInt combinarPilasOrdenadas(PilaInt p1, PilaInt p2)`.

Nota: a diferencia del ejercicio 3.e, las dos pilas pasadas como parámetro se vaciarán.
Pista: usar una pila auxiliar además de la que almacena el resultado.

Ejercicio 6

Escribir el método de clase **static boolean** `estanBalanceados(String signos)` que toma como parámetro una cadena de caracteres que se garantiza que sólo contiene paréntesis, corchetes o llaves que abren o que cierran (o sea, cualquiera de los caracteres "()[]{}". La función devuelve `true` si los paréntesis, corchetes o llaves que se abren, se van cerrando y en el orden correcto. Por ejemplo,

`"(()())"`	**sí** están balanceados.
`")("`	**no** están balanceados.
`"([]{[]})"`	**sí** están balanceados.
`"[(}"`	**no** están balanceados.
`"())"`	**no** están balanceados.
`"[(])"`	**no** están balanceados.

Dato: resolverlo utilizando una pila auxiliar.

Ejercicio 7

Usualmente, estamos acostumbrados a la notación infija para escribir operaciones matemáticas, en donde los operadores se escriben entre sus dos operandos. Además de esta notación, existe la notación *polaca inversa*, en la cual los operadores se escriben después de haber escrito los operandos del mismo, por ejemplo, para escribir "`(3 + 4) * 2`" escribiríamos "`3 4 + 2 *`". Algo muy práctico de esta notación es que nos libera del uso de los paréntesis.

a) Probar reescribir expresiones en notación infija a notación polaca inversa para asegurarte de comprender la notación.
b) Escribir el método de clase **static int** `evaluar(String expresion)` que dada una expresión en notación polaca inversa, devuelve el resultado de evaluarla. La expresión está formada de enteros u operaciones, separados por espacios, donde las operaciones son alguna de "`*+/-`". En el caso de la división, se asume división entera. Utilizar la clase StringTokenizer para dividir el String y el método `Integer.parseInt(String)` para convertir un String en un entero.

3. TADs: Colas

Dada la clase ColaInt, con las operaciones:

- **void** `encolar`(**int** `n`)
- **void** `desencolar`() [REQUIERE: `!estaVacia()`]
- **boolean** `estaVacia`()
- **int** `frente`() [REQUIERE: `!estaVacia()`]

Ejercicio 8

Escribir un método de clase que tome como parámetros dos colas **que ya se encuentran ordenadas de menor a mayor** y devuelva una nueva cola con los elementos de ambas colas ordenados de menor a mayor. El método debe tener la siguiente signatura: **static** `ColaInt combinarColasOrdenadas(ColaInt c1, ColaInt c2)`.

Nota: a diferencia del ejercicio 3.e, las dos colas pasadas como parámetro van a quedar vacías.

Ejercicio 9 ★

Escribir un método de clase que tome como parámetros dos colas y encole en ellas los elementos de la cola actual de manera alternada, dejando vacía a la misma. El método debe tener la siguiente signatura: **void** `separarEn(ColaInt c1, ColaInt c2)`. Por ejemplo, al ejecutar `c.separarEn(c1,c2)` con `c1` y `c2` vacías:

- si `c` es [1,2,3,4,5,6] encola [1,3,5] en `c1` y [2,4,6] en `c2`.
- si `c` es [10,2,5] encola [10,5] en `c1` y [2] en `c2`.
- si `c` es [] encola [] en `c1` y [] en `c2`.

Nota: los elementos deben encolarse en `c1` y en `c2`, con lo cual no es necesario que éstas estén vacías.

4. TADs: Conjuntos y diccionarios

Ejercicio 10

Estamos planeando una fiesta, y necesitamos un programa para organizar la lista de invitados. El programa debe pedir al usuario los nombres de los invitados y guardarlos en un **Set** de strings. Si el usuario ingresa dos veces a la misma persona, se debe contar sólo una vez. Asumimos que no hay dos invitados con el mismo nombre.

a) Escribir un programa que pida el nombre de cada invitado y lo vaya agregando al conjunto, informando en todo momento la cantidad total de invitados. El programa debe terminar cuando el usuario ingresa "Listo" como nombre de invitado.
b) Agregar al programa un menú, con las siguientes opciones:
   a) Agregar un invitado
   b) Eliminar un invitado

c) Consultar la lista de invitados

d) Salir del programa

Ejercicio 11 ★

La *criba de Eratóstenes* es el siguiente algoritmo para encontrar todos los números primos entre 2 y n. Se comienza con el conjunto $A = \{2, \ldots, n\}$ compuesto por todos los números naturales entre 2 y n. Se eliminan de A todos los múltiplos de 2, luego se eliminan todos los múltiplos de 3, y se continúa así hasta eliminar de A todos los múltiplos de $[\sqrt{n}]$. Una vez completados estos pasos, sólo quedan en A los números primos contenidos en el conjunto inicial. El objetivo de este ejercicio es implementar un programa que le pida al usuario el número n y que encuentre todos los números primos entre 2 y n por medio de este algoritmo.

a) Implementar un programa que represente el conjunto A con un HashSet.
b) Implementar un programa que represente el conjunto A con un TreeSet.
c) Implementar un programa que represente el conjunto A con un arreglo de enteros.
d) Comparar los tiempos de ejecución de los tres programas para algún valor suficientemente grande de n.

Ejercicio 12 ★

Implementar una clase AgendaTelefonica que permita guardar los números de teléfono de un grupo de contactos. La clase debe tener los siguientes métodos:

- **void** `registrarTelefono`(`String nombre`, `String telefono`)
- `String consultarTelefono`(`String nombre`)
- **boolean** `contiene`(`String nombre`)

Utilizar una variable de instancia de tipo `HashMap<String, String>` para registrar los datos. El método `registrarTelefono` recibe dos strings con el nombre y el teléfono del contacto, y los guarda en el diccionario. El método `consultarTelefono` recibe un string con el nombre y retorna un string con el teléfono. Si la persona no está en la agenda, debe lanzar una excepción. El método `contiene` informa si la persona pasada como parámetro está o no en la agenda.