Primer Parcial de Estructura de Datos y Algoritmos

Ejer 1	Ejer 2	Ejer 3	Nota
/4	/4	/2	/10

Duración: 2 horas 15 minutos

Condición Mínima de Aprobación. Deben cumplir estas 2 condiciones:

- Sumar **no menos de cuatro** puntos
- Sumar por lo menos 3 puntos entre el ejercicio 1 y 2.

Muy Importante

Al terminar el examen deberían subir los siguientes 2 grupos de archivos, según lo explicado en los ejercicios:

1) Clases Java: **Lista.java y IndexWithDuplicates.java + IndexWithDuplicatesTest.java** según lo pedido. Si hay código auxiliar, entregarlo también.

2) Para todos los ejercicios que no consistan en implementar código Java y se pida calcular complejidades, dibujar matrices, completar cuadros, hacer seguimientos, etc. pueden optar por alguna de estas estrategias:

   a. O completar un documento (txt, Word, pdf) y subirlo también

   b. O directamente resolverlo en hojas de papel y sacarle fotos (formato jpg, png o pdf) y subir todas las imágenes.

3) **No pedir calculadora prestada**! Si precisan hacer cuentas, escribir un código Java donde en el main hagan las cuentas que necesitan ! Java sabe sumar, sacar raíces, multiplicar, calcular logaritmos… Ese código no es preciso entregarlo, reemplaza la calculadora.

Ejercicio 1

Bajar de Campus Lista.java

Se tiene una lista con header simplemente enlazada ordenada ascendentemente. Se proveen tres constructores:

- El primero **public Lista ()** crea la lista con header pero sin items
- El segundo **public Lista (int maxNumero)** crea la lista con header donde los ítems contienen valores correlativos, empezando por el 1 hasta maxNumero inclusive.
- El tercero **public Lista (int lower, int numberItems, boolean sorpresa)** genera la lista con header con una cantidad de **numerItems** nodos. El primer nodo contiene el valor pedido l**ower**, y el pxmo numero en el nodo será el anterior + 2 o el anterior + 5, según el valor de la variable boolean **sorpresa**, la cual oscila entre true/false. En este caso, los números no serán consecutivos, pero sí ordenados ascendentemente.

Como se observa**, a futuro** se agregarán otros constructores siempre garantizando que los valores en Items son ascendentes, ya que la **lista con header simplemente encadenada está ordenada ascendentemente (eso está asegurado).**

El código ya implementado (**no cambiar lo recibido**) está en **Lista.java**

Lo que se quiere es implementar el método randomSplitListas que genera un arreglo de nuevas listas con los nodos de la lista original distribuidos de forma aleatoria entre las listas de dicho arreglo. Leer detenidamente lo pedido a continuación.

Se pide

1.1) implementar el método **Lista [] randomSplitListas(Integer nLists)** teniendo en cuenta:

- recibe el parámetro nLists el cual indica la cantidad de listas con header (del tipo Lista proporcionado en este código) que debe generar para el arreglo que deberá devolver.
- Debe dividir el contenido de la lista en nLists nuevas listas. Para ello por cada elemento de la lista original se debe llamar a getRandom(nLists) que devolverá el número de la lista donde irá ese elemento.
- Si alguna ranura no recibe elementos, se genera una lista con header pero sin items. (no va null en la ranura)
- **Super importante**: Este método no debe crear nuevos nodos Item. Los nodos (de tipo Item) de las listas en el arreglo deben reusar a los que estaban en la lista original. Más aún, cuando el método finalice, la lista original debe quedar sin ningún item porque sus elementos ya son parte de las nuevas listas. **Este método no creará directa ni indirectamente nuevos Items**. Usa lo que ya fue alocado para la lista original.

1.2) Calcular la complejidad temporal y espacial del método **randomSplitListas**. Justificar su cálculo.

A continuación, se muestran algunos casos de uso. El algoritmo debe funcionar correctamente para cualquier otro caso de uso, donde la lista original sea la descripta.

Caso de Uso A (main1):

Si se invoca al constructor new **Lista(10**), con lo cual la lista original tendrá **10 elementos secuenciales ascendentes** (es la única que hace new Item). 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10

Luego se solicita re distribuir aleatoriamente los distintos ítems en **4 listas**.

Dejamos la semilla del random en 1 (así está en el código): **private int randP = 1;**
Con esa semilla, la secuencia de 10 números random obtenida será: 2, 0, 1, 1, 0, 0, 1, 2, 3, 2

O sea, que el primer elemento de la lista original (con el 1) va a la lista 2, el segundo elemento de la lista original (con el 2) va a la lista 0, el tercer elemento de la lista original (con el 3) va a la lista 1, el cuarto elemento de la lista original (con el 4) va otra vez a la lista 1, y así siguiendo.

Al ejecutar

```
// caso A (main1)

public static void main(String[] args) {

Lista l = new Lista( 10 ); // l será: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10

// lista original al principio

System.out.print("First, the original list is ");

l.dump();


// distribuir entre 4

Lista[] caso = l.randomSplitListas( 4 );


for(int rec= 0; rec<caso.length; rec++) {

        System.out.print(String.format("list %d is ", rec));

        caso[rec].dump();

}

// lista original al final

System.out.print("Finally, the original list is ");

l.dump();

}
```

Lo obtenido será:

First, the **original list** is List with header: **first vble points to 1, last vble points to 10, items: 1->2->3->4->5->6->7->8->9->10**

list 0 is List with header: **first vble points to 2, last vble points to 6, items: 2->5->6**

list 1 is List with header: **first vble points to 3, last vble points to 7, items: 3->4->7**

list 2 is List with header: **first vble points to 1, last vble points to 10, items: 1->8->10**

list 3 is List with header: **first vble points to 9, last vble points to 9, items: 9**

Finally, **the original list** is List with header: **first vble points to null, last vble points to null, items:**

Caso de Uso B (main2):

Si se invoca al constructor new **Lista(5, 7, true**), con lo cual la lista original tendrá **7 elementos** (es la única que hace new Item):
5 -> 7 -> 12 -> 14->19->21->26

Luego se solicita re distribuir aleatoriamente los distintos ítems en **6 listas**.

Dejamos la semilla del random en 1 (así está en el código): **private int randP = 1;**
Con esa semilla, la secuencia de 7 números random obtenida será: 3, 4, 1, 3, 2, 4, 2

O sea, que el primer elemento de la lista original (con el 5) va a la lista 3, el segundo elemento de la lista original (con el 7) va a la lista 4, el tercer elemento de la lista original (con el 12) va a la lista 1, el cuarto elemento de la lista original (con el 14) va otra vez a la lista 3, y así siguiendo.

Al ejecutar

```
// caso B (main2)

public static void main(String[] args) {

Lista l = new Lista( 5, 7, true ); // l será: 5 -> 7 -> 12 -> 14->19->21->26

// lista original al principio

System.out.print("First, the original list is ");

l.dump();


Lista[] caso = l.randomSplitListas( 6 );

for(int rec= 0; rec<caso.length; rec++) {

    System.out.print(String.format("list %d is ", rec));

   caso[rec].dump();

}

// lista original al final

System.out.print("Finally, the original list is ");

l.dump();

}
```

Lo obtenido será:

First, the **original list** is List with header: **first vble points to 5, last vble points to 26, items: 5->7->12->14->19->21->26**

list 0 is List with header: **first vble points to null, last vble points to null, items:**

list 1 is List with header: **first vble points to 12, last vble points to 12, items: 12**

list 2 is List with header: **first vble points to 19, last vble points to 26, items: 19->26**

list 3 is List with header: **first vble points to 5, last vble points to 14, items: 5->14**

list 4 is List with header: **first vble points to 7, last vble points to 21, items: 7->21**

list 5 is List with header: **first vble points to null, last vble points to null, items**:

Finally, the **original list** is List with header: **first vble points to null, last vble points to null, items**:

Caso de Uso C (main3):

Si se invoca al constructor new **Lista(5, 7, false**), con lo cual la lista original tendrá **7 elementos** (es la única que hace new Item) pero con este contenido
5 -> 10 -> 12 -> 17->19->24->26

Luego se solicita re distribuir aleatoriamente los distintos ítems en **6 listas**.

Dejamos la semilla del random en 1 (así está en el código): **private int randP = 1;**
Con esa semilla, la secuencia de 7 números random es la misma que la anterior: 3, 4, 1, 3, 2, 4, 2

O sea, que el primer elemento de la lista original (con el 5) va a la lista 3, el segundo elemento de la lista original (con el 10) va a la lista 4, el tercer elemento de la lista original (con el 12) va a la lista 1, el cuarto elemento de la lista original (con el 17) va otra vez a la lista 3, y así siguiendo.

Al ejecutar

```
// caso de uso C (main 3)

public static void main(String[] args) {

Lista l = new Lista( 5, 7, false ); // l será: 5 -> 10 ->12-> 17 -> 19->24->26

// lista original al principio

System.out.print("First, the original list is ");

l.dump();



Lista[] caso = l.randomSplitListas( 6 );

for(int rec= 0; rec<caso.length; rec++) {

        System.out.print(String.format("list %d is ", rec));

        caso[rec].dump();

}
```

```
// lista original al final

System.out.print("Finally, the original list is ");

l.dump();

}
```

Lo obtenido será:

First, the **original list** is List with header: **first vble points to 5, last vble points to 26, items: 5->10->12->17->19->24->26**

list 0 is List with header: **first vble points to null, last vble points to null, items:**

list 1 is List with header: **first vble points to 12, last vble points to 12, items: 12**

list 2 is List with header: **first vble points to 19, last vble points to 26, items: 19->26**

list 3 is List with header: **first vble points to 5, last vble points to 17, items: 5->17**

list 4 is List with header: **first vble points to 10, last vble points to 24, items: 10->24**

list 5 is List with header: **first vble points to null, last vble points to null, items:**

Finally, the **original list** is List with header: **first vble points to null, last vble points to null, items:**

Caso de Uso D (main4):

Si se invoca al constructor new **Lista()**, con lo cual la lista original tendrá **0 elementos** Luego se solicita re distribuir aleatoriamente los distintos ítems en **4 listas**.

Al ejecutar

// caso de uso D (main4)

```
public static void main(String[] args) {

Lista l = new Lista(); // l tiene 0 items

// lista original al principio

System.out.print("First, the original list is ");

l.dump();

Lista[] caso = l.randomSplitListas( 4 );


for(int rec= 0; rec<caso.length; rec++) {

        System.out.print(String.format("list %d is ", rec));

        caso[rec].dump();

}

// lista original al final

System.out.print("Finally, the original list is ");

l.dump();

}
```

Lo obtenido será:

First, the **original list** is List with header: **first vble points to null, last vble points to null, items:**

list 0 is List with header: **first vble points to null, last vble points to null, items:**

list 1 is List with header: **first vble points to null, last vble points to null, items:**

list 2 is List with header: **first vble points to null, last vble points to null, items:**

list 3 is List with header: **first vble points to null, last vble points to null, items:**

Finally, the **original list** is List with header: first vble points to null, last vble points to null, items:

Ejercicio 2

Bajar de Campus IndexWithDuplicates.java e IndexWIthDuplicatesTest.java

Implementa una lista no paramétrica, que acepta duplicados y está descompactada (como la vista en clase).

Se quiere implementar un método adicional de instancia llamado **merge*(IndexWithDuplicates other).*** Este método debe fusionar dos índices ordenados en uno solo, manteniendo la integridad de la ordenación y respetando la múltiple aparición de elementos repetidos, si corresponde.

Se pide:

Implementar la función **public void merge(IndexWithDuplicates other)** la cual debe cumplir con los siguientes requisitos:

- fusiona dos índices ordenados que aceptan duplicados (el receptor del método y el parámetro other) en uno solo, preservando la ordenación ascendente. Como el método es de tipo void, esos valores quedarán en el receptor del método.

- Garantizar la integridad de la duplicidad de elementos en el índice fusionado, de manera que, si un elemento está presente en ambos índices, este deberá aparecer tantas veces en el índice resultante como ocurrencias tenga en la fusión de los índices originales. El parámetro (other) queda intacto, es read only. El receptor es el que cuando termina el merge cambia porque contiene los valores fusionados.

- El merge, debe a lo sumo una sola vez, si es necesario, buscar más espacio para almacenar todos los elementos (los que tenía y los nuevos).

- Si el IndexWithDuplicates receptor tiene N componentes y el parámetro Other tiene M componentes, sin tener en cuenta la única alocación inicial para albergar a las N+M componentes, el método merge debe implementararse con complejidad temporal O(N + M).

- Se requiere realizar pruebas de unidad exhaustivas para verificar la correctitud y eficiencia del nuevo método merge. Completar el archivo IndexWithDuplicatesTest.java. El mismo debe incorporar los siguientes casos de uso que deben resultar exitosos en la ejecución. (se pueden agregar otros casos)

Caso de Uso 1 - Índices Ordenados sin Elementos Duplicados:

```
IndexWithDuplicates index1 = new IndexWithDuplicates();
index1.initialize(new int[] {1, 3, 5, 7});

IndexWithDuplicates index2 = new IndexWithDuplicates();
index2.initialize(new int[] {2, 4, 6, 8});

index1.merge(index2);

// Resultado esperado: [1, 2, 3, 4, 5, 6, 7, 8]
```

Caso de Uso 2 - Índices Ordenados con Elementos Duplicados:

```
IndexWithDuplicates index1 = new IndexWithDuplicates();
index1.initialize(new int[] {1, 1, 3, 5, 7});

IndexWithDuplicates index2 = new IndexWithDuplicates();
index2.initialize(new int[] {2, 4, 4, 6, 8});

index1.merge(index2);

// Resultado esperado: [1, 1, 2, 3, 4, 4, 5, 6, 7, 8]
```

Caso de Uso 3 - Índices Ordenados con Diferentes Cantidades de Elementos:

```
IndexWithDuplicates index1 = new IndexWithDuplicates();
index1.initialize(new int[] {1, 3, 5});

IndexWithDuplicates index2 = new IndexWithDuplicates();
index2.initialize(new int[] {2, 4, 6, 8, 10});

index1.merge(index2);

// Resultado esperado: [1, 2, 3, 4, 5, 6, 8, 10]
```

Ejercicio 3

Se tiene implementado el **IndexBuilder y TheSearcher** como fueron vistos en clase. Sin embargo, **en vez de usar StandardAnalyzer**, tanto en la indexación como en la búsqueda se utilizó **SimpleAnalyzer.**

Se tiene la siguiente colección de documentos, donde cada uno tiene **único campo llamado content de tipo TextField** con la siguiente información.

El archivo **a.txt** es docid **0**:
Fly Me to the Moon

El archivo **b.txt** es docid **1**:
Fly Fly Away

El archivo **c.txt** es docid **2**:
Bohemian Rhapsody

El archivo **d.txt** es docid **3**:
To the Moon and Back

El archivo **e.txt** es docid **4**:
Fly Like an Eagle

Dado el siguiente query:

content:Fly OR content:Moon

Se pide

- **Devolver el resultado que obtendría el usuario, mostrando la siguiente información**: en primera posición cuál es el **score** y a **qué documento corresponde** (docid y nombre del archivo), luego, en la segunda posición cuál es el **score** y **a qué documento corresponde** (docid y nombre del archivo), y así hasta que termine la visualización del resultado

- Justificar **cada score** obtenido en la ejecución, **indicando las fórmulas** y sobre qué valores se aplican y por qué esos valores (de dónde salen). Si no se muestran las fórmulas aplicadas junto con los valores que corresponden en cada caso o no se explica de dónde salen dichos valores, el ejercicio no se considerará como resuelto. **Detallar cada cálculo en forma explícita**.