Estrategias de Programación y Estructuras de Datos

Tareas de Evaluación Junio 2020

Normativa general

En este documento se encuentra el enunciado de las cuatro tareas de evaluación para la convocatoria de junio de 2020. Junto a este documento se encuentra un directorio con la estructura de clases para realizar las tareas de programación.

El plazo de entrega de las mismas se extiende desde las 10:00 del lunes 25 de mayo hasta las 10:00 del jueves 4 de junio. Siguiendo las recomendaciones del Rectorado, transmitimos a los estudiantes que durante este periodo **realicen las entregas (en la medida de lo posible) entre las 20:00 y las 08:00** para no saturar los servidores de aLF durante la realización de pruebas síncronas en la plataforma.

La respuesta a las tareas se entregará a través de la misma tarea donde se ha publicado este enunciado, en la página de Evaluación Continua del curso virtual:

https://2020.cursosvirtuales.uned.es/dotlrn/grados/asignaturas/71901043-20/one-community?page_num=6

El formato de entrega será un fichero comprimido en ZIP que deberá contener:

- El directorio con la estructura de clases proporcionado por el Equipo Docente, en el que se incluirán las clases que habrán de ser programadas en las diferentes actividades.
- Cuatro documentos en formato PDF (uno por tarea) con la respuesta a las cuestiones teóricas planteadas en cada una de las tareas. El nombre de cada uno de estos documentos incluirá el número de tarea y los apellidos y nombre del estudiante, de la siguiente manera: "T1 Perez Gomez Ana.pdf", "T3 Martin Gonzalez Pedro.pdf", etc.
- En la primera página cada uno de estos documentos se deberá indicar claramente nombre, apellidos y DNI/pasaporte del estudiante.

Por restricciones de almacenamiento, **el fichero comprimido no podrá superar los 10MB de tamaño**.

Para la evaluación de las respuestas, además de la **corrección** y **justificación de los razonamientos**, se valorará la **claridad de las explicaciones** y la **presentación** de las mismas.

Tarea 1: secuencias doblemente enlazadas y listas con acceso por punto de interés

Calificación: 3 puntos.

1.1 Ejercicios de programación:

1. En primer lugar, se ha de modificar la clase `Sequence<E>` para obtener la clase `SequenceDL<E>` que implemente la interfaz `SequenceDLIF<E>` proporcionada por el Equipo Docente. Esta clase implementa secuencias doblemente enlazadas, en las que los elementos se organizan linealmente y cada uno conoce cuál es su siguiente elemento y su elemento anterior.

2. A continuación se ha de programar la clase `ListIP<E>` que implemente la interfaz `ListIPIF<E>` proporcionada por el Equipo Docente, la cual representa listas con acceso por punto de interés. Esta clase ha de heredar **obligatoriamente** de la clase `SequenceDL<E>` para así poder usar una secuencia doblemente enlazada como estructura de soporte.

Se proporciona una clase `Tarea1` con un método `main` que realiza a modo de prueba algunas operaciones sobre una lista con punto de interés que contiene valores enteros. Recomendamos que los estudiantes realicen más pruebas a fin de comprobar que su implementación cumple los requisitos indicados.

1.2 Cuestiones teórico-prácticas:

1. Calcule el coste asintótico temporal de las ocho operaciones prescritas en la interfaz `ListIPIF<E>`, tal y como se han implementado en la clase `ListIP<E>`, en función del tamaño de la secuencia.

2. Detalle **todos los puntos de la práctica** donde aparece utilizada una lista con acceso por posición (la perteneciente a la implementación de referencia de la asignatura) y para cada uno de ellos argumente razonadamente si sustituir en ese punto la lista con acceso por posición por una lista con acceso por punto de interés mejoraría el coste de las operaciones afectadas. Tenga en cuenta las restricciones con las que se ha realizado la práctica.

Tarea 2: bicolas (o colas dobles)

Calificación: 3 puntos

Una **bicola o cola doble** (en inglés conocida como ***deque***) es una estructura de datos lineal que permite insertar y eliminar elementos tanto por el inicio como por el final.

2.1 Ejercicios de programación:

1. Programe la clase `Deque<E>` que implemente la interfaz `DequeIF<E>` proporcionada por el Equipo Docente. Esta clase ha de heredar **obligatoriamente** de la clase `SequenceDL<E>` (ver ejercicio 1).

2. Programe las clases `StackDeque<E>` y `QueueDeque<E>` de manera que implementen las interfaces `StackIF<E>` y `QueueIF<E>` respectivamente utilizando una bicola (`DequeIF<E>`) como estructura de soporte.

Se proporciona una clase `Tarea2` con un método `main` que realiza a modo de prueba algunas operaciones sobre las estructuras descritas anteriormente, compuestas por valores enteros. Además, la clase incluye un método que genera de forma pseudoaleatoria una bicola de longitud máxima n compuesta de valores enteros. Recomendamos que los estudiantes realicen más pruebas a fin de comprobar que su implementación cumple los requisitos indicados.

2.2 Cuestiones teórico-prácticas:

1. Calcule el coste temporal en el caso peor de las operaciones prescritas en la interfaz `DequeIF<E>`, tal y como se han implementado en la clase `Deque<E>`. ¿Qué coste tendrían las operaciones de las bicolas (`Deque<E>`) si heredasen de la clase `Sequence<E>` en lugar de `SequenceDL<E>`? **NOTA:** no se requiere implementar las bicolas de esta manera.

2. Para cada uno de los métodos **F**, **G** y **H** sobre bicolas que se pueden encontrar en las siguientes páginas, se pide:

    a) Explique qué es lo que calcula.
    b) Escriba una recurrencia que exprese su función de coste en el caso peor.
    c) Obtenga el coste en el caso peor a partir de la recurrencia anterior empleando las ecuaciones de resolución de recurrencias proporcionadas por el Equipo Docente.

Método F

```
public static <E extends Comparable<E>> boolean F(DequeIF<E> d){
	if(d.size()>1){
		if(d.getFront().compareTo(d.getBack())!=0){
			return false;
		}
		d.removeFront();
		d.removeBack();
		return F(d);
	}
	return true;
}
```

Método G

```
public static <E> void G(DequeIF<E> d){
	Gaux(d,d.size());
}
public static <E> void Gaux (DequeIF<E> d,int s){
	if(s>1){
		E e = d.getFront();
		d.removeFront();
		Gaux(d,s-1);
		d.insertBack(e);
	}
}
```

Método H

```
public static <E extends Comparable<E>> DequeIF<E> H(DequeIF<E> d){
	int s = d.size();
	if(s<=1){
		return d;
	}
	int m = s/2;
	IteratorIF<E> it = d.iterator();
	int cont = 1;
	Deque<E> daux1 = new Deque<E>();
	Deque<E> daux2 = new Deque<E>();
	while(cont<=m){
		daux1.insertBack(it.getNext());
		cont++;
	}
	while(cont<=s){
		daux2.insertBack(it.getNext());
		cont++;
	}
	daux1 = (Deque<E>) H(daux1);
	daux2 = (Deque<E>) H(daux2);
	if(daux1.getBack().compareTo(daux2.getFront())<=0){
		IteratorIF<E> itAux = daux2.iterator();
		while(itAux.hasNext()){
			daux1.insertBack(itAux.getNext());
		}
		return daux1;
	}
	DequeIF<E> R = Haux(daux1,daux2);
	return R;
}
```

```java
public static <E extends Comparable<E>> DequeIF<E> Haux(DequeIF<E> d1,
DequeIF<E> d2){
	DequeIF<E> R = new Deque<E> ();
	while(d1.size()>0 && d2.size()>0){
		if(d1.getFront().compareTo(d2.getFront())<=0){
			R.insertBack(d1.getFront());
			d1.removeFront();
		}
		else{
			R.insertBack(d2.getFront());
			d2.removeFront();
		}
	}
	if(d1.size()>0){
		IteratorIF<E> itAux = d1.iterator();
		while(itAux.hasNext()){
			R.insertBack(itAux.getNext());
		}
	}
	if(d2.size()>0){
		IteratorIF<E> itAux = d2.iterator();
		while(itAux.hasNext()){
			R.insertBack(itAux.getNext());
		}
	}
	return R;
}
```

Tarea 3: enriquecimiento de las operaciones de las pilas

Calificación: 2 puntos.

Una pila es una estructura de datos que permite el acceso a sus elementos siguiendo el criterio LIFO (del inglés Last In, First Out, "último en entrar, primero en salir"). Sin embargo, en ciertas ocasiones, puede ser interesante acceder al elemento que se encuentra en el fondo de la pila, que fue el primero en entrar. Por ello, en este ejercicio se pide:

1. Describa el código (no es necesario implementarlo) y las precondiciones de los métodos:

   ```
   void pushBottom(E elem)
   void popBottom()
   ```

   que permitan, respectivamente, insertar un elemento en el fondo de la pila y sacar el elemento que se encuentra en el fondo **de una pila no vacía**. Para la realización de esta cuestión, **sólo se pueden utilizar las operaciones de pilas disponibles en el TAD `StackIF<E>`** proporcionado por el Equipo Docente (es decir, las operaciones han de ser válidas **para cualquier implementación de dicho TAD**) y, además, **no se permite el uso de ninguna estructura de datos auxiliar**. Calcule el coste asintótico temporal en el caso peor de ambas operaciones con las restricciones impuestas.

2. Dada esta nueva funcionalidad de una pila, se pide discutir y justificar qué estructura de soporte de las vistas en la asignatura (incluyendo estos ejercicios) sería la más adecuada para el TAD `inverseStackIF<E>`, que además de las operaciones de `StackIF<E>` incluya los métodos descritos en el apartado anterior. Calcule el coste asintótico temporal en el caso peor de ambas operaciones en base a la estructura de soporte elegida.

Tarea 4: árboles binarios de búsqueda con repeticiones

Calificación: 2 puntos.

En la asignatura hemos trabajado con árboles binarios de búsqueda que organizaban conjuntos de elementos sin elementos repetidos. A menudo, sin embargo, nos encontramos problemas en los que los datos pueden estar repetidos. Conteste a estas dos cuestiones:

1. Proponga una nueva versión de `BSTreeIF<E>` que considere elementos repetidos. Indique qué métodos añade y qué métodos modifica. Indique (sin programar) de qué forma modificaría la estructura de datos para acomodar elementos repetidos de manera que sea eficiente añadirlos, eliminarlos y saber cuántas veces está repetido un elemento.

2. Un AVL es un caso particular de árbol binario de búsqueda, por lo que podemos usar las mismas modificaciones del apartado anterior para crear una versión de AVL que pueda considerar elementos repetidos.

   Considere la secuencia de las primeras diez letras de su nombre y apellidos sin distinguir entre mayúsculas y minúsculas ni considerar diacríticos (si no llegaran a diez, repita letras volviendo a empezar por el principio de su nombre hasta llegar a esa cifra). Por ejemplo, si el nombre completo fuese “Bing Luo”, se deberá trabajar con la secuencia “BINGLUOBIN”.

   Introduzca secuencialmente las letras **en ese orden** en un árbol AVL que admita elementos repetidos mediante las operaciones de inserción y equilibrado (rotaciones simples y dobles) estudiadas en la asignatura. Especifique en cada paso por qué es necesario (o no) reequilibrar el árbol. En caso de que sea necesario reequilibrarlo, indique qué tipo de rotación hay que aplicar, por qué, y cuál es el resultado. ¿Cuántas operaciones han sido necesarias para obtener el árbol AVL final? ¿Cuál hubiera sido el orden de entrada de las diez letras que hubiera minimizado el número de operaciones?