Código: 75.001

Asignatura: Estructura de la Información

Curso: 2010/2011 – segundo semestre

PEC 1

Ejercicio 1 [2 puntos]

Una cooperativa de taxis decide hacer una aplicación para gestionar la asignación de los servicios a sus taxistas. Se desea hacer las asignaciones en el mismo orden que se han generado. Es decir, si en el momento que se solicitan servicios no hay ningún taxi disponible, en cuanto un taxista quede libre tomará el primer servicio pedido. El mismo criterio se desea aplicar a los taxistas. En caso de que haya taxistas libres, en el momento que se pida un servicio, se asignará al taxista que haga más tiempo que haya entregado.

El TAD *TaxiManagement* deberá tener la siguiente funcionalidad:

- Obtener el taxi libre con mayor tiempo de espera.
- Dar de alta un servicio pedido por un cliente.
- Asignar el servicio más antiguo en el taxi libre con más tiempo de espera.
- Dar por concluido un servicio.

Apartado 1.1) Dad la signatura del TAD *TaxiManagement*. Es decir, indicad el nombre que daríais a las operaciones encargadas de cada funcionalidad requerida, indicando también, cuales deberían ser los parámetros de entrada y cuál la salida en caso de que se necesitara. Suponed que existen los TADs *Taxi* para gestionar los datos de un taxi y *Service* para gestionar los datos de un servicio de taxi.

```
TAD TaxiManagement signatura{
        Taxi getTaxi();
        void addService( Service service);
        void assignServiceToTaxi();
        void endService(Service);
}
```

La operación *assignServiceToTaxi*() no recibe parámetro para asegurar la coherencia del enunciado. Si pasáramos el taxi o el servicio, podría no coincidir con el taxi ni el servicio que más tiempo de espera llevan. En esta signatura también suponemos que Service conoce el taxi que se le ha asignado para asegurar que el taxi que deberá dejarse libre corresponda al del servicio realizado.

Apartado 1.2) El TAD específico que se pide estará compuesto por TADs genéricos de la biblioteca de la asignatura. Indica cuáles y cuántos serán necesarios para poder llevar a cabo la gestión.

Necesitaremos dos Colas, una de taxis y otra de servicios que habrá que implementar como:

Cola <Taxi> taxiQueue;
Cola <Service> serviceQueue;

Para mantener los servicios en curso podríamos usar una lista:

Lista <Service> servicesInProgress;

Apartado 1.3) Indica qué operaciones de los TADs de biblioteca de la asignatura que has elegido deberás llamar para resolver las operaciones del TAD *TaxiManagement*. Es decir, suponiendo que se le pidiera la funcionalidad de añadir el número de teléfono desde donde se ha llamado y el TAD escogido fuera una lista, una posible respuesta sería: ***añadirAlfinal(teléfono)***. Si la funcionalidad fuera eliminar el número de teléfono, una posible respuesta sería: **obtener un recorrido con la operación *posiciones*(). Iterar por las diferentes posiciones hasta encontrar la posición donde se encuentra el teléfono buscado. Con la posición encontrada llamar a *borrar(posicion)***.

Taxi getTaxi ():

- consultar el primer elementos de la cola de taxis.

void addService (Servicio service):

- encola el servicio a la cola de servicios.

void assignServiceToTaxi ():

- Desencolar el primer taxis de la cola de taxis
- Desencolar el primer servicio de la cola de servicios
- Asociar el taxis y el servicio extraídos.
- Añadir el servicio asociado a la lista de servicios en curso.

void endService (Service):

- Eliminar el servicio de la lista de servicios en curso
- Obtener el taxi asociado al servicio,
- Encola el taxi en la cola de taxis
- Eliminar el servicio del sistema

Apartado 1.4) Haz la especificación contractual del TAD. Utiliza de modelo la especificación del apartado 1.2.3 del Módulo 1 de los materiales docentes. Se valorará especialmente la concisión (ausencia de elementos redundantes o innecesarios), precisión (definición correcta del resultado de las operaciones), completitud (consideración de todos los casos posibles en que se puede ejecutar cada operación) y falta de ambigüedades (conocimiento exacto de cómo se comporta cada operación en todos los casos posibles) de la solución. Es importante responder a este apartado usando una descripción condicional y no procedimental. La experiencia nos demuestra que no siempre resulta fácil distinguir entre ambas descripciones, es por ello que hacemos especial hincapié insistiendo que pongáis

mucha atención en vuestras definiciones. A título de ejemplo indicaremos que la descripción condicional (la correcta a usar en el contrato) de *llenar un vaso vacío con agua* seria:

> @pre el vaso se encuentra vacío.
> @post el vaso se encuentra lleno de agua

En cambio una descripción procedimental (incorrecta) tendría una forma similar a:

> @pre el vaso debería encontrarse vacío, si no se encontrara vacío debería vaciarse.
> @post Se acerca el vaso al grifo y se echa agua hasta que esté lleno

Debéis también tener en cuenta que un contrato debería disponer de invariante siempre que esta fuera necesaria para describir el TAD.

> @pre true
> @post $return el taxi que fa més temps que es troba esperant un servei.
> **Taxi getTaxi():**
>
> @pre true
> @post service s'ha afegit al TAD com a servei en espera
> **void addService(Servei service):**
>
> @pre true
> @post El taxi i el servei amb més temps d'espera es troben associats i el servei en curs.
> **void assignServiceToTaxi():**
>
> @pre el servei passat per paràmetre es troba en curs
> @post El taxi associat a service es troba en espera i service no és un servei del sistema.
> **void endService(Service service):----------**
>
> @ pre true
> @ post $ return el taxi que hace más tiempo que se encuentra esperando un servicio.
> **Taxi getTaxi ():**
>
> @ pre true
> @ post service se ha añadido al TAD como servicio a la espera
> **void addService (Servicio service):**
>
> @ pre true
> @ post El taxi y el servicio con más tiempo de espera se encuentran asociados y el servicio en curso.
> **void assignServiceToTaxi ():**
>
> @ el servicio pasado por parámetro se encuentra en curso
> @ post El taxi asociado a service se encuentra en espera y service no es un servicio del sistema.
> **void endService (Service service):**

Ejercicio 2 [2 puntos]

Apartado 2.1) Escribe un algoritmo que permita contabilizar cuántas veces se repite un vector de caracteres en el contenido de un otro vector. Ejemplo: sea el vector v1 = {u, n, , v, e, c, t, o, r, , d, e, , c, a, r, a, c, t, e, r, e, s, , d, e, , c, u, a, l, q, u, i, e, r, , t, i, p, o} y el vector v2 = {d, e}. Si se pide cuantas veces se repite v2 en v1, la respuesta deberá ser 2 veces.

```
1.  public int countSubstrings(char[] v1, char[] v2){
2.      int pos1;
3.      int res=0;
4.      int pos2 =0;
5.      for(pos1=0; pos1<v1.length; pos1++){
6.          pos2=0;
7.          while(pos1+pos2<v1.length && pos2<v2.length && v1[pos1+pos2]==v2[pos2]){
8.              pos2++;
9.          }
10.         if(pos2==v2.length){
11.             res++;
12.         }
13.     }
14.     return res;
15.  }
```

Apartado 2.2) Haz un estudio del coste temporal del algoritmo.

	código	coste		
1.	**public** int countSubstrings(...			
2.	int pos1;			
3.	int res=0;	1		
4.	int pos2 =0;	1		
5.	**for**(pos1=0; pos1<v1.length; ...	1; 1; 1		
6.	pos2=0;	1		
7.	**while**(pos1+pos2<v1.length...	7		
8.	pos2++;	1		
9.	}		M (7+1) +7	
10.	**if**(pos2==v2.length){	1		
11.	res++;	1		
12.	}		2	
13.	}			1+N (1+1+1+ M(8)+7+2)+1
14.	**return** res;	1		
15.	}			1+1+1 + N (1+1+1+ M(8)+7+2)+1+1

T(N) = 5+N(12+8M)=4+12N+8NM

Por lo tanto diremos que el algoritmo tiene una complejidad asintótica de O (NM)

Ejercicio 3 [2 puntos]

Indica qué árbol tiene los siguientes recorridos en preorden {2,5,1,7,3,9,4,12} y en inorden {5,7,1,3,2,9,12,4}. Razona la respuesta explicando cómo llegas a la conclusión.

1) la raíz es 2 porque es el primer elemento del preodre

preordre: **2**,5,1,7,3,9,4,12

2) 5,7,1,3 se encuentran a la izquierda de la raíz y 9,12,4 a la derecha, de acuerdo con el inorden
inordre: 5,7,1,3,**2**,9,12,4

3) la raíz del subárbol izquierdo es el 5
preordre: 2,**5**,1,7,3,9,4,12

4) El 5 no tiene hijo izquierdo pero si hijo derecho
inordre: **5**,7,1,3,2,9,12,4

5) El árbol derecho del nodo de valor 5 tiene por raíz el 1 con el 7 como hijo izquierdo y el 3 como derecho.
Preordre: 2,5,**1**,7,3,9,4,12

inordre: 5,7,**1**,3,2,9,12,4

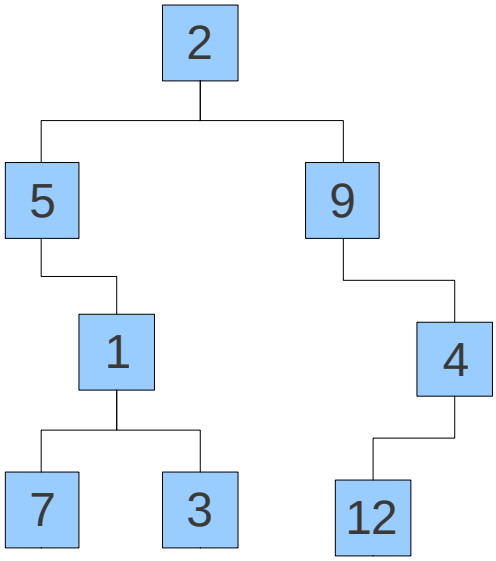
6) el 9 es el raíz subárbol derecho
preordre: 2,5,1,7,3,**9**,4,12

7) el 9 no tiene hijo izquierdo pero si derecho
inordre: 5,7,1,3,2,**9**,12,4

8) El 12 es hijo izquierdo del 4
preordre: 2,5,1,7,3,9,**4**,12

inordre: 5,7,1,3,2,9,12,**4**

El árbol quedará como:



Ejercicio 4 [2 puntos]

Apartado 4.1) Explicad con vuestras palabras las diferencias entre los TADs *Cola* y *Lista*. ¿En qué casos usaríais una cola y en qué otros una lista?

La Cola es una TAD mucho más dinámico que la lista. El primero es adecuado para algoritmos que necesiten tratar los datos del TAD sólo una vez ya que en desencolar extraemos también el dato del TAD. La lista en cambio mantiene un cierto orden y permite recorrer sus elementos múltiples veces sin cambiar la composición. Por otra parte, el orden de los elementos de la cola será siempre el mismo que el orden de llegada, en cambio la lista permite insertar elementos en cualquier posición. En cierta forma podemos decir que la cola es una lista específica para mantener sus elementos en el orden con el que lleguen y el tratamiento de estos implica su supresión del contenedor.

Apartado 4.2) Indica si la afirmación: "El coste asintótico describe a la perfección el rendimiento de una operación" es cierta o falsa, razonando tu respuesta.

El coste asintótico es una aproximación del comportamiento del coste de un algoritmo a medida que éste tenga que tratar un número creciente de datos. Por lo tanto la respuesta sería falsa.

Apartado 4.3) ¿Se puede buscar un elemento de una lista ordenada con repeticiones con un coste logarítmico? ¿Y si la lista no permite repeticiones?

El coste logarítmico al buscar dentro de una lista lineal, se podría podría conseguir haciendo búsqueda dicotómica. Desgraciadamente, este algoritmo se posiciona sobre cualquiera de los

datos repetidas de valor coincidente en la búsqueda. Esto implica que no es posible distinguir entre valores iguales en tiempo logarítmico. Por lo tanto diremos que no es posible buscar logarítmicamente en listas ordenadas que admitan valores repetidos.
Si que sería posible en listas ordenadas sin repeticiones, siempre y cuando la implementación de la lista se basara en un vector. Las listas encadenadas no soportan la búsqueda dicotómica porque no tenemos forma de seleccionar el valor que se encuentra en la posición intermedia entre dos posiciones dadas. Las implementaciones vectoriales, en cambio, al trabajar con posiciones numéricas si que permiten calcular posiciones intermedias.

Apartado 4.4) Explica brevemente cómo se relacionan los conceptos de "recursividad" y "recorridos de un árbol".

Decimos que los hijos de la raíz de un árbol tienen también todas las características de árbol y por lo tanto soportarán las mismas operaciones que el árbol principal. Esto tiene unas implicaciones muy interesantes ya que los hijos se considerarán también raíces del subárbol correspondiente y los hijos de los hijos (en caso de que existan) seguirán siendo árbol sobre los que aplicar las mismas operaciones que en cualquier árbol.

El recorrido de un árbol es una operación que se puede resolver recursivamente, aplicando la característica explicada en el párrafo anterior teniendo en cuenta que cualquier recorrido se obtiene combinando en diferente orden el recorrido parcial del hijo izquierdo, el recorrido parcial del hijo derecho y el nodo raíz.

Ejercicio 5 [2 puntos]

Apartado 5.1) Se pide que implementéis el TAD *PilaUnica*, que extiende del TAD *Pila* de la biblioteca pero que asegura cumplir siempre la invariante:

@invariante Todos sus elementos son diferentes.

Formalmente: $all (e1: pila,! $ exists (e2: pila, e1 = e2));

NOTA: la implementación debéis hacerla usando herencia. Es decir, solamente hace falta implementar las operaciones nuevas o las ya existentes que habría que cambiar.

```
public class PilaUnica<E> extends PilaEncadenadaImpl<E> {

   public void empilar(E elem) {
      boolean found = false;
      Iterador<E> it = this.elementos();
      while(it.haySiguiente() && !found){
         found = it.siguiente().equals(elem);
```

```
        }
        if(!it.haySiguiente () && !found){
            super.empilar(elem);
        }
    }
}
```

Apartado 5.2) ¿Qué coste tendría la operación de empilar un elemento?

Tendría una coste lineal porque hay recorrer todo el contenedor para asegurar que el elemento que se empila sea único.

Apartado 5.3) ¿Qué estructura usarías para mejorar su rendimiento?

Se podría usar un vector ordenado que permitiera buscar en tiempo logarítmico si el elemento a empilar se encuentra ya dentro del contenedor o no.