

Estructuras de Datos y Algoritmos

Grados de la Facultad de Informática (UCM)

Examen SEGUNDO CUATRIMESTRE, 2 de julio de 2019

Normas de realización del examen

1. Debes desarrollar e implementar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección <http://exacrc>.
2. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen.
3. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que subas al juez.
4. Del enlace **Material para descargar** dentro del juez puedes descargar un archivo comprimido que contiene material que puedes utilizar para la realización del examen (transparencias de clase, implementación de las estructuras de datos, una plantilla de código fuente y ficheros de texto con los casos de prueba de cada ejercicio del enunciado).
5. Los ficheros con las implementaciones de las estructuras de datos están instalados en el juez, por lo que no es necesario subirlos como parte de tu solución (y conviene no hacerlo).
6. Los ejercicios están identificados con el nombre del tema de la asignatura en el que habrían aparecido si hubieran sido propuestos como ejercicios durante el curso. Para obtener la máxima puntuación, las soluciones deberán seguir los criterios exigidos a los ejercicios de ese tema durante el curso.
7. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.
8. Al terminar el examen, dirígete al puesto del profesor y rellena con tus datos la hoja de firmas que él tendrá. Muéstrale tu documento de identificación.

Primero resuelve el problema. Entonces, escribe el código.

— John Johnson

*Comentar el código es como limpiar el cuarto de baño;
nadie quiere hacerlo, pero el resultado es siempre
una experiencia más agradable para uno mismo y sus invitados.*

— Ryan Campbell

Ejercicio 1. Tipos de datos lineales (2 puntos)

Entremezclar dos listas consiste en formar otra tomando alternativamente elementos de cada una de esas listas. Por ejemplo, si entremezclamos la lista 1 3 5 7 con la lista 2 4 6 8, obtenemos la lista 1 2 3 4 5 6 7 8. Cuando una de las dos listas se queda vacía, se cogen todos los elementos de la otra.

Queremos extender la clase `deque` (lo importante para el ejercicio es que internamente la cola está representada con una lista enlazada circular doble de nodos dinámicos con nodo fantasma) con una nueva operación `entremezclar` que mezcle una lista enlazada doble (la del objeto `this`) con otra (recibida como argumento), comenzando a coger elementos de la primera lista. La lista recibida como argumento pasará a ser vacía.

Para resolver este ejercicio no se puede crear ni destruir memoria dinámica (hacer `new` o `delete`) al entremezclar, ni tampoco modificar los valores almacenados en los nodos de las listas enlazadas.

Entrada

La entrada consta de una serie de casos de prueba. En la primera línea aparece el número de casos de prueba que vendrán a continuación.

Cada caso se muestra en cuatro líneas. La primera contiene el número N de elementos de la lista principal (un número entre 0 y 100.000). En la segunda línea se muestran esos N elementos, números entre 1 y 1.000.000, separados por espacios. La tercera línea contiene el número M de elementos de la segunda lista (un número entre 0 y 100.000) y la cuarta línea contiene esos M elementos, números entre 1 y 1.000.000.

Salida

Para cada caso de prueba se escribirá una línea que contendrá los elementos de la lista modificada tras entremezclarla con los elementos de la segunda lista.

Entrada de ejemplo

```
3
4
1 3 5 7
4
2 4 6 8
3
7 3 9
0

2
2 1
4
5 3 4 6
```

Salida de ejemplo

```
1 2 3 4 5 6 7 8
7 3 9
2 5 1 3 4 6
```

Ejercicio 2. Aplicaciones de tipos abstractos de datos (3 puntos)

Queremos gestionar la adquisición y venta de productos por parte de una tienda. De vez en cuando la tienda recibe nuevos productos (identificados por un código, que es un **string** sin espacios) y señalados con una fecha. Las unidades del producto adquirido se guardan en el almacén, salvo que haya clientes que hubieran intentado comprar ese determinado producto cuando no había existencias y se hubieran colocado en la lista de espera. En ese caso, los clientes son servidos en riguroso orden de llegada. También puede haber venta de productos en existencia. En ese caso, se venden siempre las unidades con una fecha menor.

Para ello se desea disponer de las siguientes operaciones:

- constructora: al comienzo el almacén está vacío y no hay clientes en espera.
- **adquirir(COD, F, CANT)**: gestiona la adquisición de **CANT** unidades del producto **COD** con fecha **F**. Si hubiera clientes esperando la llegada de este producto, devuelve los identificadores de los clientes que han podido ser servidos, en el orden en que hicieron la petición. El resto de unidades (si las hay) se guardan en el almacén.
- **vender(COD, CLI)**: gestiona la venta de una unidad del producto **COD** al cliente **CLI** (un **string** sin espacios). Si hay existencias, la operación devuelve **true** y la fecha del producto vendido (la menor entre las disponibles). Si no hay existencias, devuelve **false** y añade al cliente a la lista de espera de este producto (un cliente puede aparecer varias veces en la lista de espera).
- **cuantos(COD)**: devuelve cuántas unidades tiene la tienda del producto **COD** (independientemente de la fecha).
- **hay Esperando(COD)**: indica si hay clientes en la lista de espera del producto **COD**.

Selecciona un tipo de datos adecuado para representar la información. Puedes utilizar el tipo **fecha** que os proporcionamos. En la cabecera de cada función debe indicarse el coste de la misma. Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realiza en la función **resuelveCaso** que os proporcionamos.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso consiste en una serie de líneas donde se muestran las operaciones a realizar, sobre una tienda inicialmente vacía: el nombre de la operación seguido de sus argumentos. Cada caso termina con una línea con la palabra **FIN**.

Salida

Para cada caso de prueba, se escribirá una línea por operación de la siguiente manera:

- **adquirir**, muestra **PRODUCTO ADQUIRIDO** seguido de los códigos de los clientes que estuvieran en la lista de espera (si los había) y hayan podido llevarse una unidad;
- **vender**, si hay existencias en ese momento se muestra **VENDIDO** y la fecha del producto vendido; si no, se muestra **EN ESPERA**;
- **cuantos**, muestra el número devuelto por la operación;
- **hay Esperando**, si hay clientes esperando a que llegue ese producto escribe **SI**, y en caso contrario escribe **NO**.

Cada caso termina con una línea con tres guiones (---).

Entrada de ejemplo

```
vender lapiz Ana
adquirir lapiz 10/06/19 3
vender lapiz Pedro
adquirir boli 20/06/19 3
adquirir boli 15/06/19 2
vender boli Pedro
vender boli Luis
vender boli Marta
cuantos boli
hay Esperando boli
FIN
vender boli Ana
hay Esperando boli
hay Esperando lapiz
vender boli Pedro
vender boli Luis
adquirir boli 20/06/19 2
cuantos boli
hay Esperando boli
FIN
```

Salida de ejemplo

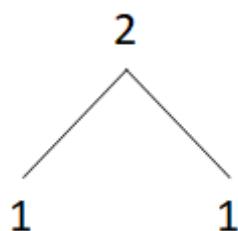
```
EN ESPERA
PRODUCTO ADQUIRIDO Ana
VENDIDO 10/06/19
PRODUCTO ADQUIRIDO
PRODUCTO ADQUIRIDO
VENDIDO 15/06/19
VENDIDO 15/06/19
VENDIDO 20/06/19
2
NO
---
EN ESPERA
SI
NO
EN ESPERA
EN ESPERA
PRODUCTO ADQUIRIDO Ana Pedro
0
SI
---
```

Estructuras de Datos y Algoritmos

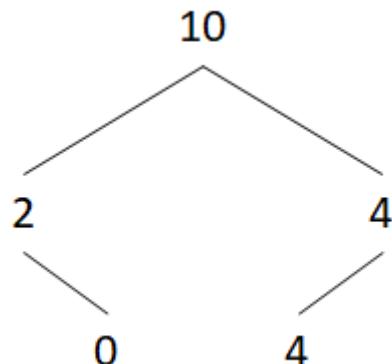
Grados en Ingeniería Informática

Examen Segundo Cuatrimestre, 2 de julio de 2019. Grupos B y D

1. (4 puntos) Un nodo interno de un árbol binario de enteros se dice que es *sumativo* si su valor es igual a la suma de los valores de sus descendientes. A modo de ejemplo, un árbol vacío o un árbol hoja tendrán 0 nodos internos sumativos. El árbol



tendrá 1 nodo interno sumativo, y el árbol



tendrá 2 nodos internos sumativos. **Implementa** el subprograma

```
int numNodosSumativos (Arbin<int> a)
```

que devuelva el número de nodos internos sumativos que tiene el árbol dado. **Determina de forma justificada la complejidad** de dicho subprograma.

2. (6 puntos) Nos han encargado implementar un módulo para la gestión de surtidores en una gasolinera. La gasolinera tiene surtidores, cada uno de los cuales puede servir distintos tipos de combustible. Cuando un vehículo llega a la gasolinera, se pone en la cola de espera de alguno de los surtidores y reposta combustible (los detalles de pagos no se cubrirán en este módulo). Así mismo, en cualquier momento un vehículo puede abandonar la cola de espera y seguir su camino. La implementación de este módulo se llevará a cabo como un TAD `GestorSurtidores` con las siguientes operaciones:

- `crea`: creación de un nuevo gestor de surtidores.
- `an_surtidor(id_surtidor)`: se añade un nuevo surtidor con identificador `id_surtidor`. Si el surtidor ya existe en el sistema, se señala un error (excepción `ESurtidorDuplicado`).
- `carga(id_surtidor, tipo_combustible, num_litros)`: carga `num_litros` de combustible `tipo_combustible` en el surtidor `id_surtidor`. Si el surtidor no existe en el sistema, se señala un error (excepción `ESurtidorNoExiste`).
- `pon_en_espera(id_vehiculo, id_surtidor)`: pone en espera el vehículo `id_vehiculo` en el surtidor `id_surtidor`. Si el vehículo ya está esperando en otro surtidor, o bien `id_surtidor` no existe, se señala un error (excepción `ELlegadaVehiculo`).
- `vende(id_surtidor, tipo_combustible, num_litros) → resul`: realiza una venta de `num_litros` de combustible `tipo_combustible` al primer vehículo que está esperando en el surtidor `id_surtidor`. Tras la venta, se actualiza la reserva de combustible del surtidor y el vehículo abandona la gasolinera.
Una vez realizada la venta, la operación devuelve un objeto que contiene los siguientes campos: (i) el identificador del vehículo al que se ha realizado la venta; y (ii) la cantidad de combustible de tipo `tipo_combustible` que queda aún en el surtidor.
Para que esta operación puede realizarse: (i) el surtidor debe existir; (ii) debe haber vehículos esperando para abastecerse en el mismo; (iii) el surtidor debe disponer de la cantidad suficiente de combustible solicitado; y (iv) `num_litros` debe ser mayor que 0. Si se viola alguna de estas condiciones, la operación señala un error (excepción `EErrorVenta`).

- `abandona(id_vehiculo)`: el vehículo `id_vehiculo` abandona la cola en la que está esperando y sale de la gasolinera. Si el vehículo no está esperando en ninguna cola, la operación no tiene efecto.

Dado que este módulo es un sistema crítico, debe **elegirse una representación** que permita implementar lo más eficientemente posible las operaciones pedidas, así como **llevar a cabo dicha implementación**. Para cada operación debe **indicarse justificadamente su complejidad**.

Estructuras de Datos y Algoritmos

Grados de la Facultad de Informática (UCM)

Examen SEGUNDO CUATRIMESTRE, 31 de mayo de 2019

Normas de realización del examen

1. Debes desarrollar e implementar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección <http://exacrc>.
2. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen.
3. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que subas al juez.
4. Del enlace **Material para descargar** dentro del juez puedes descargar un archivo comprimido que contiene material que puedes utilizar para la realización del examen (transparencias de clase, implementación de las estructuras de datos, una plantilla de código fuente y ficheros de texto con los casos de prueba de cada ejercicio del enunciado).
5. Los ficheros con las implementaciones de las estructuras de datos están instalados en el juez, por lo que no es necesario subirlos como parte de tu solución (y conviene no hacerlo).
6. Los ejercicios están identificados con el nombre del tema de la asignatura en el que habrían aparecido si hubieran sido propuestos como ejercicios durante el curso. Para obtener la máxima puntuación, las soluciones deberán seguir los criterios exigidos a los ejercicios de ese tema durante el curso.
7. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.
8. Al terminar el examen, dirígete al puesto del profesor y rellena con tus datos la hoja de firmas que él tendrá. Muéstrale tu documento de identificación.

Primero resuelve el problema. Entonces, escribe el código.

— John Johnson

*Comentar el código es como limpiar el cuarto de baño;
nadie quiere hacerlo, pero el resultado es siempre
una experiencia más agradable para uno mismo y sus invitados.*

— Ryan Campbell

Ejercicio 1. Tipos de datos lineales (3 puntos)

Estamos programando un *bot* de Telegram que enviará noticias a sus seguidores y ahora nos toca tratar el escabroso tema de los accidentes de aviación. Cuando se produce uno de estos accidentes, suelen hacerse comentarios del estilo “*Este es el accidente más grave desde febrero de 1995*”.

Nos han pasado un listado (ordenado cronológicamente) de accidentes ocurridos en el pasado y queremos estar preparados para que, cuando ocurra el siguiente accidente, podamos producir un comentario como el anterior.

De hecho, para probar esta funcionalidad, queremos saber cuál habría sido el comentario cuando se produjo cada uno de los accidentes conocidos.

Entrada

La entrada está formada por una serie de casos. Cada caso comienza con el número N de accidentes conocidos (un número entre 1 y 250.000). A continuación aparecen N líneas con la descripción de cada uno de ellos: una fecha (con formato DD/MM/AAAA) y el número de víctimas en ese accidente. Todas las fechas son distintas y el listado está ordenado cronológicamente de menor a mayor.

Salida

Para cada caso se escribirán N líneas. La i -ésima línea contendrá la fecha del último accidente anterior que tuvo (estrictamente) más víctimas que el accidente i -ésimo. Si no existe tal accidente (en particular, eso ocurre siempre para el primero), se escribirá NO HAY en su lugar.

Después de cada caso se escribirá una línea con tres guiones (---).

Entrada de ejemplo

```
6
19/12/1990 50
01/02/2000 80
10/05/2001 30
20/10/2005 10
08/07/2007 60
10/07/2007 40
```

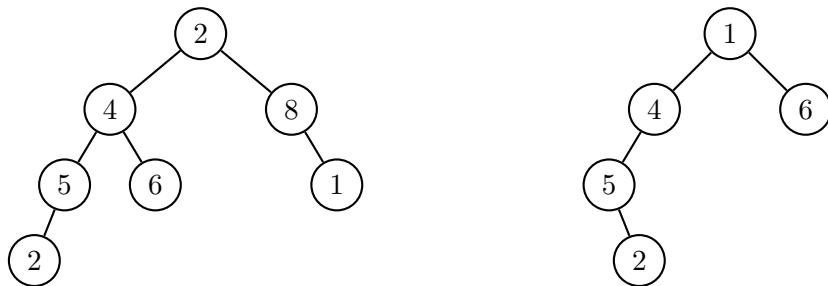
Salida de ejemplo

```
NO HAY
NO HAY
01/02/2000
10/05/2001
01/02/2000
08/07/2007
---
```

Ejercicio 2. Tipos de datos arborescentes (3 puntos)

Definimos un *camino* entre dos nodos de un árbol binario como una secuencia de nodos $n_1 n_2 \dots n_k$ sin repeticiones (por cada nodo del árbol se pasa como mucho una vez) tal que para todo par de nodos consecutivos $n_i n_{i+1}$ ($1 \leq i < k$) uno de ellos siempre es padre del otro (n_i es padre de n_{i+1} o n_{i+1} es padre de n_i). Definimos la *longitud* de un camino $n_1 n_2 \dots n_k$ como el número de nodos que lo forman, k .

Dado un árbol binario de números enteros positivos queremos calcular la longitud del camino más largo formado todo él por números pares. Por ejemplo, el siguiente árbol de la izquierda tiene el camino 6 4 2 8 de longitud 4. En cambio, en el árbol de la derecha todos los caminos formados únicamente por números pares tienen longitud 1.



Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en una serie de números enteros positivos con la descripción de un árbol binario: el árbol vacío se representa con el valor -1; un árbol no vacío se representa con el valor de un nodo (que denota la raíz), seguido primero de la descripción del hijo izquierdo y después de la descripción del hijo derecho.

Salida

Para cada caso, se escribirá la longitud del camino más largo entre dos nodos del árbol, formado todo él por números pares.

Entrada de ejemplo

```
3
2 4 5 2 -1 -1 -1 6 -1 -1 8 -1 1 -1 -1
1 4 5 -1 2 -1 -1 -1 6 -1 -1
3 4 2 -1 -1 6 -1 8 -1 -1 5 4 -1 2 -1 -1 -1
```

Salida de ejemplo

```
4
1
4
```

Ejercicio 3. Diccionarios (4 puntos)

Tenemos una lista de las películas emitidas durante el último año en una cadena de televisión. De cada película conocemos los actores que intervienen en ella y el tiempo que aparecen en pantalla durante la película. Queremos obtener la película y el actor *preferidos* por la cadena. La película *preferida* es aquella que más veces se ha emitido, mientras que el actor *preferido* es aquel que más minutos aparece en pantalla contando todas las emisiones. Si existen varios actores que han aparecido el mismo tiempo máximo mostraremos todos ellos por orden alfabético. Si existen varias películas que se han emitido el mayor número de veces, mostraremos la que se ha emitido más recientemente de todas ellas.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso se muestra en varias líneas. En la primera se indica el número *P* de películas distintas que se han emitido. A continuación se muestra, para cada película, su título y el número *A* de actores que aparecen en ella. En la línea siguiente se muestra el nombre de cada actor y su tiempo de actuación (un número positivo) en esta película, separados por blancos. A continuación se muestra el número *E* de películas emitidas por la cadena de televisión en todo el año, seguido de los títulos de las películas en el orden en que fueron emitidas. La entrada finaliza con una línea con cero películas.

Los títulos de las películas y los nombres de los actores son cadenas de caracteres sin espacios en blanco. Los títulos y los nombres de actores están escritos siempre de la misma manera.

Salida

La salida de cada caso se escribirá en dos líneas. En la primera se muestra el máximo número de veces que se ha emitido una película seguido del título de la película. Si existen varias películas que se han emitido el mismo número máximo de veces, se muestra el título de la última emitida. En la segunda línea se muestra el máximo tiempo que ha aparecido un actor en pantalla seguido de los nombres de los actores que han aparecido ese tiempo máximo en orden alfabético.

Entrada de ejemplo

```
2
pelicula1 3
actor1 10 actor2 30 actor3 5
pelicula2 2
actor1 10 actor4 30
3
pelicula2 pelicula2 pelicula1
4
pelicula1 3
actor3 5 actor1 10 actor2 20
pelicula3 1
actor5 40
pelicula4 1
actor1 40
pelicula2 2
actor1 10 actor4 15
5
pelicula2 pelicula1 pelicula2 pelicula3 pelicula1
0
```

Salida de ejemplo

```
2 pelicula2
60 actor4
2 pelicula1
40 actor1 actor2 actor5
```

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática

Examen Segundo Cuatrimestre, 30 de mayo de 2019. Grupos B y D

1. (2.5 puntos) “Estremecer” una lista consiste en colocar todos los elementos que aparecen en posiciones pares, por orden de aparición, seguidos de todos los que aparecen en posiciones impares, por orden inverso de aparición (primero el último, después el penúltimo, etc.). Por ejemplo, el resultado de “estremecer” a

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

es

0	2	4	6	7	5	3	1
---	---	---	---	---	---	---	---

(importante: consideramos que las posiciones comienzan en 0: es decir, la posición del primer elemento es la 0, la del segundo elemento es la 1, etc.)

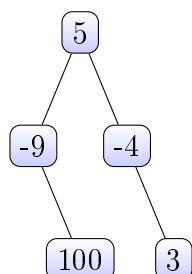
Añade una nueva operación mutadora

```
void estremece();
```

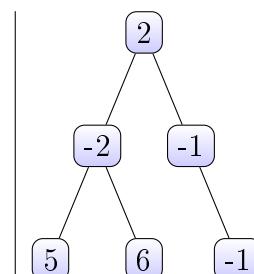
a la implementación del TAD Lista basada en nodos doblemente enlazados, que “estremezca” la lista, y determina justificadamente su complejidad. Dicha operación no puede invocar, ni directa ni indirectamente, operaciones de manejo de memoria dinámica (`new`, `delete`), ni tampoco puede realizar asignaciones a los contenidos de los nodos.

2. (2.5 puntos) Un “árbol de ganancias y pérdidas” es un árbol binario de enteros en el que los valores positivos de los nodos representan ganancias, mientras que los negativos representan pérdidas. Un nodo es “rentable” cuando todos sus ancestros son rentables, y, además, la suma de los valores de todos sus ancestros y el del suyo propio es positiva (dicha suma se denomina la “renta” del nodo). El “árbol de ganancias y pérdidas” es “rentable” cuando tiene alguna hoja “rentable” (la “renta” de dicha hoja se denomina una “renta” del árbol).

Ejemplos:



Árbol “rentable”; “renta” máxima: 4



Árbol no “rentable”

Debes programar el procedimiento:

```
void mejor_renta(Arbin<int> a, bool & es_rentable, int & renta_maxima)
```

que determine: (i) si el “árbol de ganancias y pérdidas” a es “rentable” o no; y, en caso positivo, (ii) determine la “renta” máxima del árbol. Si el árbol es “rentable”, tras finalizar la ejecución el parámetro `es_rentable` valdrá `true`, y `renta_maxima` contendrá la “renta” máxima del árbol (es decir, la mayor “renta” de las hojas que son rentables). Si el árbol no es “rentable”, tras finalizar la ejecución el parámetro `es_rentable` valdrá `false`. En este caso, el valor de `renta_maxima` es irrelevante. También debes determinar justificadamente la complejidad del procedimiento.

- 3. (5 puntos)** La tienda Outlet aGoGo nos ha encargado programar un TAD para gestionar su sistema de ofertas online. Este sistema permite ofertar cantidades limitadas de sus productos a precios especiales. Cada vez que un cliente desea beneficiarse de una oferta, el sistema lo pone en la lista de espera para dicha oferta, gestionando las ventas por estricto orden de llegada, hasta que se agoten las unidades disponibles (un cliente podrá esperar simultáneamente para comprar más de un producto). El TAD deberá incluir las siguientes operaciones:

- **crea**: Crea un sistema de venta vacío.
- **an_oferta(producto, num_unidades)**: Crea una nueva oferta para el producto `producto`, con `num_unidades` unidades disponibles. Esta es una operación parcial: no debe existir ya un producto con el mismo nombre en el sistema, y, además, el número de unidades debe ser positivo.
- **pon_en_espera(cliente, producto)**: Añade al cliente `cliente` a la lista de espera para la compra del producto en oferta `producto`. En caso de que el cliente ya esté esperando para comprar dicho producto, la operación no tendrá ningún efecto. Esta es una operación parcial: el producto al que se hace referencia debe estar ofertándose actualmente en el sistema.
- **cancela_espera(cliente, producto)**: Elimina al cliente `cliente` de la lista de espera del producto en oferta `producto`. Si el cliente no está esperando en la lista de espera del producto, la operación no tendrá ningún efecto. Esta es una operación parcial: el producto debe estar ofertándose en el sistema.
- **num_en_espera(producto)→num_clientes**: Devuelve el número de clientes que están esperando para comprar el producto `producto`. Esta es una operación parcial: el producto debe estar ofertándose en el sistema.
- **venta(producto, num_unidades)**: Registra una venta de `num_unidades` unidades del producto `producto` al primer cliente de la lista de espera. Dicho cliente se elimina de la lista de espera. En caso de que, tras dicha venta, no queden más unidades, la venta para el producto se cerrará, eliminado la oferta del sistema (y, por tanto, todos los clientes que están esperando se quedarán sin producto). Esta es una operación parcial: el producto debe estar ofertándose actualmente en el sistema, la lista de espera para dicho producto no debe estar vacía, y el número de unidades solicitado no debe sobrepasar el número de unidades disponibles.
- **primer_en_espera(producto)→cliente**: Devuelve el nombre del primer cliente que está esperando para comprar el producto `producto`. Esta es una operación parcial: el producto se debe estar ofertando actualmente, y su lista de espera no debe estar vacía.
- **lista_ventas()→Lista**: Devuelve una lista del número de unidades vendido para cada producto desde la puesta en marcha del sistema. Cada elemento de esta lista consiste en: (i) el nombre del producto; y (ii) todas las unidades vendidas de este producto (si un producto se ha ofertado varias veces, este valor será el total vendido para cada oferta). La lista estará ordenada alfabéticamente por los nombres de productos.

Aparte de realizar la implementación del TAD, debes indicar justificadamente cuál es la complejidad de cada operación. Al tratarse de un sistema crítico, la implementación deben ser lo más eficientes posible.

Estructura de Datos y Algoritmos

Grados de la Facultad de Informática (UCM)

Examen FINAL SEPTIEMBRE, 6 de septiembre de 2018

1. (2 puntos) Se dice que dos vectores de enteros del mismo tamaño y cada uno con todos sus elementos distintos entre sí son *desplazamiento circular* el uno del otro cuando la secuencia de elementos de ambos son iguales si no se tiene en cuenta la posición de comienzo de las mismas.

Por ejemplo, [7, 9, 1, 3, 5] es desplazamiento circular de [1, 3, 5, 7, 9] y de [7, 9, 1, 3, 5], pero no lo es de [3, 5, 1, 7, 9].

Se ha desarrollado el siguiente algoritmo para comprobar si dos vectores son desplazamiento circular el uno del otro:

```
method desplazamiento (v : array<int>, w : array<int>) returns (i : int, b : bool)
requires v != null && w != null && ...
ensures ...
{
    var N := v.Length;
    i := 0;
    while (i < N && v[0] != w[i])
        invariant ...
        decreases ...
        { i := i+1; }
        assert ...;
    var j := 0;
    while (j < N && v[j] == w[(i+j)%N])
        invariant ...
        decreases ...
        { j := j + 1; }
        b := j == N;
}
```

Se pide:

1. Especificar la función **desplazamiento**.
 2. Escribir una *postcondición* para el primer bucle.
 3. Escribir sendos *invariantes* y *funciones de cota* para los dos bucles de forma que permitan demostrar la corrección de la función.
 4. Indicar y justificar el *coste asintótico* en el caso peor de la función.
2. (3 puntos) Tenemos dos cadenas de caracteres de la misma longitud y no vacías. La primera se denomina *cadena principal* y la segunda es la *cadena auxiliar*, con la que podemos intercambiar los elementos de la cadena principal que se encuentren en la misma posición. El objetivo es minimizar el número de caracteres diferentes de la cadena principal haciendo intercambios con la cadena auxiliar.

Por ejemplo, dada la *cadena principal* **aba**, con dos caracteres diferentes: **a** y **b** y la *cadena auxiliar* **eao**, podemos intercambiar los elementos en la segunda posición (el carácter **b** de la *cadena principal* por el carácter **a** de la *cadena auxiliar*) y obtenemos una cadena con un único carácter diferente: **a**. El intercambio tanto de los primeros como de los últimos elementos incrementa el número de elementos diferentes por lo que no es conveniente realizarlo.

Se pide implementar un algoritmo de *vuelta atrás* que devuelva el mínimo número de caracteres diferentes que se puede conseguir en la cadena principal. Completa para ello el código del fichero **main2Final.cpp**.

Ten en cuenta que las cadenas están formadas por caracteres de la '**a**' a la '**z**' del código ASCII. La posición de una letra, **car**, en el alfabeto teniendo en cuenta la distribución de los caracteres del código ASCII se calcula como: **car - 'a'**.

- 3. (3 puntos)** Queremos extender la clase `deque` (lo importante para el ejercicio es que internamente la cola está representada con una lista enlazada circular doble de nodos dinámicos con nodo fantasma) con una nueva operación `engordar` que añada a una lista enlazada doble el contenido de otra lista enlazada doble dada de la siguiente forma: los nodos de la segunda lista se colocarán alternativamente al principio y al final de la primera lista. La lista recibida como argumento pasará a ser vacía.

Para resolver este ejercicio no se puede crear ni destruir memoria dinámica (hacer `new` o `delete`), ni tampoco modificar los valores almacenados en las listas enlazadas.

Completa el código del fichero `main3.cpp` con la implementación de esta nueva operación.

- 4. (2 puntos)** Dada una secuencia de números enteros positivos (no necesariamente ordenados y con posibles repeticiones) y un rango $[inf, sup]$, se quieren imprimir en orden creciente todos los números del rango que no estén en la secuencia. Para ello se pide implementar una función `enRangoYNoEnSecuencia` que dada la secuencia de entrada `sec` como `list<int>`, y los enteros `inf` y `sup`, devuelva la secuencia de salida, también como `list<int>`, con los números del rango que no estén en `sec`, ordenados crecientemente. Se valorará el coste del algoritmo implementado, el cual debes indicar y justificar. Aclaración: Puedes usar TADs auxiliares. Ejemplo: dada la secuencia 3 9 1 3 7 2 y el rango [3,8] se debe imprimir la secuencia 4 5 6 8.

Completa el código del fichero `main4.cpp` con la implementación de esta operación.

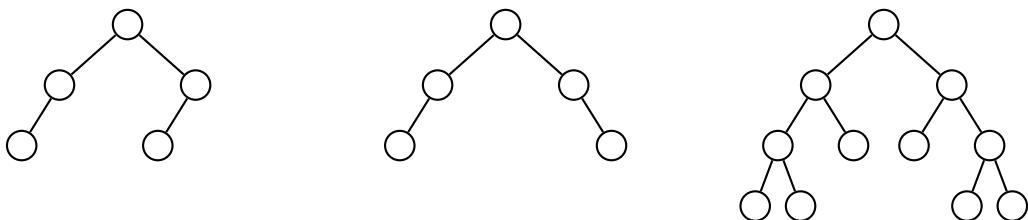
Normas de realización del examen

1. Debes programar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección <http://exacrc/domjudge/team>.
2. Escribe comentarios que expliquen tu solución, justifiquen por qué se ha hecho así y ayuden a entenderla. Calcula la complejidad de todas las funciones que implementes.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen. El nombre de usuario y contraseña que has estado utilizando durante la evaluación continua **no** son válidos.
4. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que entregues.
5. Descarga el fichero <http://exacrc/Docum353637.zip> que contiene material que debes utilizar para la realización del examen (implementación de las estructuras de datos, ficheros con código fuente y ficheros de texto con algunos casos de prueba de cada ejercicio del enunciado).
6. Si la necesitas, puedes encontrar ayuda sobre la librería estándar de C++ en <http://exacrc/cpp/reference/en/>.
7. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.

Estructuras de Datos y Algoritmos

Examen de SEPTIEMBRE, Segundo Cuatrimestre, 6 de septiembre de 2018

- 1. (2.5 puntos)** Un árbol binario no vacío es *simétrico* respecto al eje vertical que pasa por la raíz si al “doblarlo” por ese eje todo nodo de un lado coincide con un nodo del otro. Por ejemplo, de los siguientes árboles, el de la izquierda no es simétrico pero los otros dos sí lo son.



Completa el código del fichero `main1parcial.cpp` con la implementación de una función que, dado un árbol binario no vacío, averigüe si es simétrico o no.

- 2. (2.5 puntos)** Implementa una función que dada una cola de números enteros que está ordenada crecientemente según el valor absoluto de sus elementos, la modifique de forma que quede ordenada crecientemente según el valor de sus elementos. Puedes utilizar estructuras de datos auxiliares; justifica tu elección.

Por ejemplo, si la cola contiene los elementos $-1, 1, -3, 4, 5, -7, 9, 10, -15$, al final debe contener los elementos $-15, -7, -3, -1, 1, 4, 5, 9, 10$.

Completa el código del fichero `main2parcial.cpp` con la implementación de esta función.

- 3. (3 puntos)** Queremos extender la clase `deque` (lo importante para el ejercicio es que internamente la cola está representada con una lista enlazada circular doble de nodos dinámicos con nodo fantasma) con una nueva operación `engordar` que añada a una lista enlazada doble el contenido de otra lista enlazada doble dada de la siguiente forma: los nodos de la segunda lista se colocarán alternativamente al principio y al final de la primera lista. La lista recibida como argumento pasará a ser vacía.

Para resolver este ejercicio no se puede crear ni destruir memoria dinámica (hacer **new** o **delete**), ni tampoco modificar los valores almacenados en las listas enlazadas.

Completa el código del fichero `main3.cpp` con la implementación de esta nueva operación.

4. (2 puntos) Dada una secuencia de números enteros positivos (no necesariamente ordenados y con posibles repeticiones) y un rango $[inf, sup]$, se quieren imprimir en orden creciente todos los números del rango que no estén en la secuencia. Para ello se pide implementar una función `enRangoYNoEnSecuencia` que dada la secuencia de entrada `sec` como `list<int>`, y los enteros `inf` y `sup`, devuelva la secuencia de salida, también como `list<int>`, con los números del rango que no estén en `sec`, ordenados crecientemente. Se valorará el coste del algoritmo implementado, el cual debes indicar y justificar. Aclaración: Puedes usar TADs auxiliares. Ejemplo: dada la secuencia 3 9 1 3 7 2 y el rango [3,8] se debe imprimir la secuencia 4 5 6 8.

Completa el código del fichero `main4.cpp` con la implementación de esta operación.

Normas de realización del examen

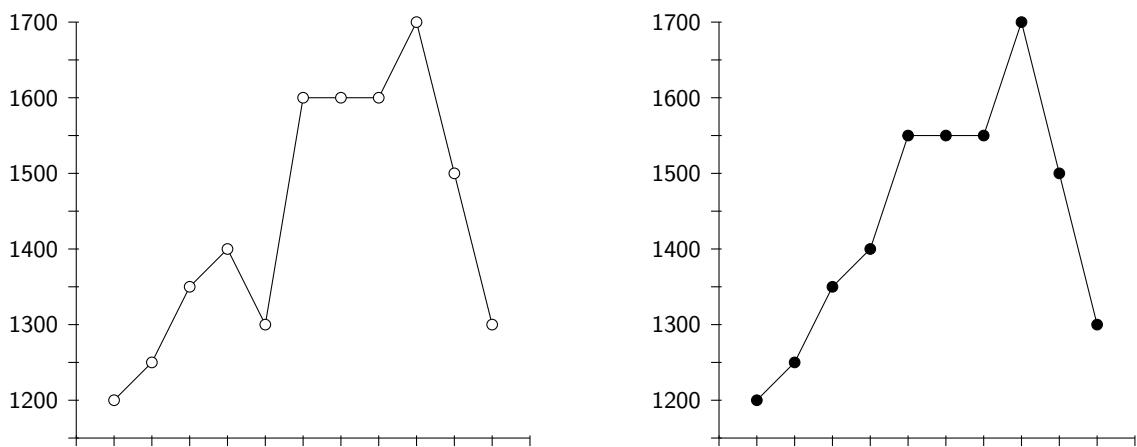
1. Debes programar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección <http://exacrc/domjudge/team>.
2. Escribe comentarios que expliquen tu solución, justifiquen por qué se ha hecho así y ayuden a entenderla. Calcula la complejidad de todas las funciones que implementes.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen. El nombre de usuario y contraseña que has estado utilizando durante la evaluación continua **no** son válidos.
4. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que entregues.
5. Descarga el fichero <http://exacrc/Docum353637.zip> que contiene material que debes utilizar para la realización del examen (implementación de las estructuras de datos, ficheros con código fuente y ficheros de texto con algunos casos de prueba de cada ejercicio del enunciado).
6. Si la necesitas, puedes encontrar ayuda sobre la librería estándar de C++ en <http://exacrc/cpp/reference/en/>.
7. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.

Estructuras de Datos y Algoritmos
Grados de la Facultad de Informática (UCM)

Examen FINAL JUNIO, 1 de junio de 2018

- 1. (3 puntos)** Con la llegada del buen tiempo, el grupo de senderistas “Caminando voy” quiere preparar una serie de excursiones por la Sierra de Madrid. Las excursiones deben ser aptas para todos los socios (hay un numeroso grupo de jubilados y otro de niños de corta edad). En concreto, se requiere que las cuestas arriba no se hagan demasiado penosas. Para ello, para cada excursión se ha confeccionado un perfil de desniveles, consistente en una secuencia de cotas de altura (valores enteros no negativos). Las cuestas arriba se corresponden con segmentos estrictamente crecientes, y el desnivel de una cuesta será la diferencia entre su cota más alta y su cota más baja. Una excursión se considerará *apta* si en todas las cuestas arriba el desnivel no supera un cierto valor estipulado $D \geq 0$.

Por ejemplo, supongamos $D = 300$ metros. Una excursión con el perfil $[1200, 1250, 1350, 1400, 1300, 1600, 1600, 1600, 1700, 1500, 1300]$ sería *apta* (primera gráfica). En cambio, una de perfil $[1200, 1250, 1350, 1400, 1550, 1550, 1550, 1700, 1500, 1300]$ sería *no apta* (segunda gráfica).

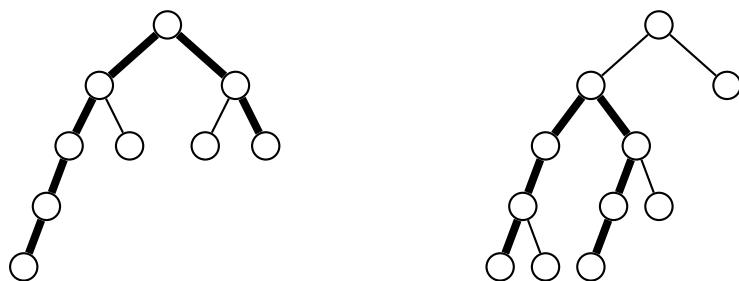


Especifica, diseña e implementa una función que reciba una secuencia no vacía de enteros no negativos (≥ 0) representando el perfil de una excursión, y un entero no negativo correspondiente al desnivel máximo permitido, y determine si la excursión es apta o no.

Escribe el *invariante* y la *función de cota* que permitan demostrar la corrección del algoritmo implementado. Por último, justifica el *coste* del algoritmo conseguido.

El fichero `main1.cpp` contiene código que puedes completar para resolver este ejercicio.

- 2. (3 puntos)** Definimos un *camino* en un árbol binario como una secuencia de nodos $n_1 n_2 \dots n_k$ sin repeticiones (por cada nodo del árbol se pasa como mucho una vez) tal que para todo par de nodos consecutivos $n_i n_{i+1}$ ($1 \leq i < k$) uno de ellos siempre es padre del otro (n_i es padre de n_{i+1} o n_{i+1} es padre de n_i). Definimos la *longitud* de un camino $n_1 n_2 \dots n_k$ como el número de nodos que lo forman, k . Y definimos el *diámetro* de un árbol como la longitud del camino más largo del árbol. Por ejemplo, los dos árboles siguientes tienen diámetro 7 y un camino de esa longitud aparece resaltado con trazo más grueso en los árboles.



Implementa una función externa a la clase `bintree` que calcule el diámetro de un árbol binario.

El fichero `main2.cpp` contiene ya el código necesario para este ejercicio, salvo la función `diametro`.

3. (4 puntos) Tu tarea consiste en producir la clasificación final de un concurso de programación conociendo los envíos de soluciones a los distintos problemas que han realizado los equipos participantes. Para la clasificación se tienen en cuenta las siguientes reglas:

- El criterio principal para ordenar a los equipos es el número de problemas resueltos. Cuantos más problemas se resuelvan, mejor será el puesto en la clasificación.
- A igualdad de problemas resueltos, los equipos que hayan tardado menos tiempo en resolverlos se clasifican primero.
- El tiempo utilizado por un equipo es igual a la suma del tiempo necesario para resolver cada problema resuelto correctamente.
- El tiempo necesario para resolver un problema es la suma del tiempo transcurrido desde el comienzo del concurso hasta el primer envío correcto a ese problema, más 20 minutos de penalización por cada envío (a ese problema) incorrecto anterior. Los envíos posteriores al primer envío correcto se ignoran.
- Los problemas que finalmente no se resuelven no penalizan.
- Si dos equipos resuelven el mismo número de problemas en el mismo tiempo, serán ordenados por su nombre de menor a mayor.

La entrada consta de una serie de casos de prueba. En la primera línea aparece el número de casos que vendrán a continuación. Cada caso consiste en una serie de descripciones de envíos, cada uno en una línea. Cada envío está formado por el nombre del equipo (una cadena de caracteres sin espacios), el nombre del problema (otra cadena de caracteres sin espacios), los minutos transcurridos desde el comienzo del concurso, y el veredicto del juez (AC, WA, TLE, etc). Un envío se considera correcto solamente si el veredicto es AC. Los envíos de cada caso están ordenados de menor a mayor número de minutos. El último envío está seguido de una línea con la palabra FIN.

Para cada caso de prueba se escribirá la clasificación final. Esta contendrá una línea por cada equipo que haya realizado algún envío, donde aparecerá el nombre del equipo, el número de problemas resueltos y el tiempo empleado en resolverlos. Los equipos aparecen ordenados según las reglas de clasificación, comenzando por el mejor clasificado. La clasificación de cada caso estará seguida por una línea con “----”.

El fichero `main3.cpp` contiene código que puedes completar para resolver este ejercicio.

Normas de realización del examen

1. Debes programar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección <http://exacrc/domjudge/team>.
2. Escribe comentarios que expliquen tu solución, justifiquen por qué se ha hecho así y ayuden a entenderla. Calcula la complejidad de todas las funciones que implementes.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen. El nombre de usuario y contraseña que has estado utilizando durante la evaluación continua **no** son válidos.
4. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que entregues.
5. Descarga el fichero <http://exacrc/Docum122448.zip> que contiene material que debes utilizar para la realización del examen (implementación de las estructuras de datos, ficheros con código fuente y ficheros de texto con algunos casos de prueba de cada ejercicio del enunciado).
6. Si la necesitas, puedes encontrar ayuda sobre la librería estándar de C++ en <http://exacrc/cpp/reference/en/>.
7. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.

Estructuras de Datos y Algoritmos

Grados de la Facultad de Informática (UCM)

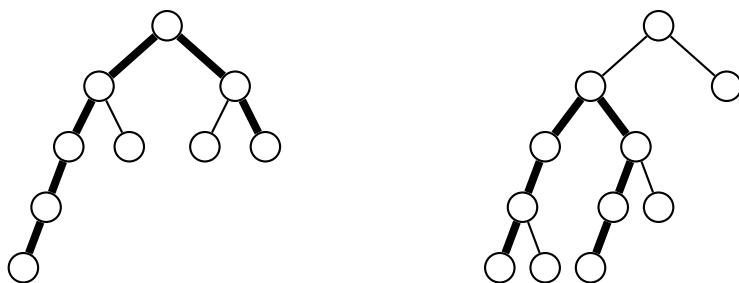
Examen de JUNIO, Segundo Cuatrimestre, 1 de junio de 2018

1. (3 puntos) Queremos extender la clase `queue` (lo importante para el ejercicio es que internamente la cola está representada con una lista enlazada simple de nodos dinámicos) con una nueva operación que inserte en una lista enlazada ordenada los elementos de otra lista enlazada ordenada recibida como argumento, de tal forma que la lista resultante quede también ordenada. La lista recibida como argumento pasará a ser vacía.

Para resolver este ejercicio no se puede crear ni destruir memoria dinámica (hacer `new` o `delete`), ni tampoco modificar los valores almacenados en las listas enlazadas.

Completa el código del fichero `main1.cpp` con la implementación de esta nueva operación.

2. (3 puntos) Definimos un *camino* en un árbol binario como una secuencia de nodos $n_1 n_2 \dots n_k$ sin repeticiones (por cada nodo del árbol se pasa como mucho una vez) tal que para todo par de nodos consecutivos $n_i n_{i+1}$ ($1 \leq i < k$) uno de ellos siempre es padre del otro (n_i es padre de n_{i+1} o n_{i+1} es padre de n_i). Definimos la *longitud* de un camino $n_1 n_2 \dots n_k$ como el número de nodos que lo forman, k . Y definimos el *diámetro* de un árbol como la longitud del camino más largo del árbol. Por ejemplo, los dos árboles siguientes tienen diámetro 7 y un camino de esa longitud aparece resaltado con trazo más grueso en los árboles.



Implementa una función externa a la clase `bintree` que calcule el diámetro de un árbol binario.

El fichero `main2.cpp` contiene ya el código necesario para este ejercicio, salvo la función `diametro`.

3. (4 puntos) Tu tarea consiste en producir la clasificación final de un concurso de programación conociendo los envíos de soluciones a los distintos problemas que han realizado los equipos participantes. Para la clasificación se tienen en cuenta las siguientes reglas:

- El criterio principal para ordenar a los equipos es el número de problemas resueltos. Cuantos más problemas se resuelvan, mejor será el puesto en la clasificación.
- A igualdad de problemas resueltos, los equipos que hayan tardado menos tiempo en resolverlos se clasifican primero.
- El tiempo utilizado por un equipo es igual a la suma del tiempo necesario para resolver cada problema resuelto correctamente.
- El tiempo necesario para resolver un problema es la suma del tiempo transcurrido desde el comienzo del concurso hasta el primer envío correcto a ese problema, más 20 minutos de penalización por cada envío (a ese problema) incorrecto anterior. Los envíos posteriores al primer envío correcto se ignoran.
- Los problemas que finalmente no se resuelven no penalizan.
- Si dos equipos resuelven el mismo número de problemas en el mismo tiempo, serán ordenados por su nombre de menor a mayor.

La entrada consta de una serie de casos de prueba. En la primera línea aparece el número de casos que vendrán a continuación. Cada caso consiste en una serie de descripciones de envíos, cada uno en una línea. Cada envío está formado por el nombre del equipo (una cadena de caracteres sin espacios), el nombre del problema (otra cadena de caracteres sin espacios), los minutos transcurridos desde el comienzo del concurso, y el veredicto del juez (**AC**, **WA**, **TLE**, etc.). Un envío se considera correcto solamente si el veredicto es **AC**. Los envíos de cada caso están ordenados de menor a mayor número de minutos. El último envío está seguido de una línea con la palabra **FIN**.

Para cada caso de prueba se escribirá la clasificación final. Esta contendrá una línea por cada equipo que haya realizado algún envío, donde aparecerá el nombre del equipo, el número de problemas resueltos y el tiempo empleado en resolverlos. Los equipos aparecen ordenados según las reglas de clasificación, comenzando por el mejor clasificado. La clasificación de cada caso estará seguida por una línea con “**----**”.

El fichero `main3.cpp` contiene código que puedes completar para resolver este ejercicio.

Normas de realización del examen

1. Debes programar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección <http://exacrc/domjudge/team>.
2. Escribe comentarios que expliquen tu solución, justifiquen por qué se ha hecho así y ayuden a entenderla. Calcula la complejidad de todas las funciones que implementes.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen. El nombre de usuario y contraseña que has estado utilizando durante la evaluación continua **no** son válidos.
4. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que entregues.
5. Descarga el fichero <http://exacrc/Docum122448.zip> que contiene material que debes utilizar para la realización del examen (implementación de las estructuras de datos, ficheros con código fuente y ficheros de texto con algunos casos de prueba de cada ejercicio del enunciado).
6. Si la necesitas, puedes encontrar ayuda sobre la librería estándar de C++ en <http://exacrc/cpp/reference/en/>.
7. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.

Estructura de Datos y Algoritmos

Grados de la Facultad de Informática de la UCM (Grupos C y F). Curso 2016-2017

Examen final de septiembre Tiempo: 3 horas

Ejercicio 1 [2.5 puntos]

Sea un vector $V[0..N], N \geq 1$ de enteros *no repetidos*. Se dice que forma un *valle* si existe una posición tal que todos los elementos anteriores a ella forman una secuencia estrictamente decreciente y a partir de la misma todos los elementos son estrictamente creciente:

Formalmente:

$$\{P[0..N], N \geq 0\} \quad (1)$$

$$\{b = \exists n : 0 \leq n < N : \text{valley}(0, n, N)\} \quad (2)$$

donde

$$\text{valley}(0, n, N) \equiv (\forall i : 0 \leq i < n : V[i] > V[i + 1]) \wedge (\forall i : n \leq i < N : V[i] < V[i + 1]) \quad (3)$$

- Programar un procedimiento imperativo que dado un vector de enteros de entrada determine si los valores de éste determinan una configuración de *valle*.
- Justificar su complejidad.

Ejercicio 2 [2.5 puntos]

El juego de cartas *WhiteJacket* es cada vez más popular. Su funcionamiento es como sigue: Juega solo un jugador y hay dos barajas de cartas en la mesa. En cada jugada hay tres opciones, coger una carta del montón izquierdo, coger una carta del montón derecho o plantarse. Se empieza con 0 puntos. Coger una nueva carta (de cualquiera de los montones) resta n puntos, siendo n el número de jugada (la primera jugada resta un punto, la segunda resta dos puntos, etc.), y también suma p puntos siendo p el valor numérico de la carta (siempre positivo).

Se pide implementar un algoritmo que encuentre la secuencia de jugadas con mejor puntuación y su puntuación asociada. En caso de empate se debe elegir la secuencia más corta, y en caso de empate en longitud la que se encuentre primero, entendiendo que el algoritmo explorará antes las jugadas que cogen del montón izquierdo. La secuencia debe devolverse como un `List<bool>`, siendo `false` coger del montón izquierdo y `true` coger del derecho.

La función principal proporcionada para hacer pruebas lee los dos montones en sendos arrays (secuencias de enteros no negativos acabando cada lectura con un -1 que no se incluye en el array), llama a la función pedida, y muestra por pantalla la mejor puntuación y en la siguiente línea la mejor secuencia (ver ejemplos). El proceso se repite hasta introducir un array vacío (es decir, un -1).

Entrada	Salida
2 4 -1	3
3 1 -1	0 0
1 5 -1	8
7 4 -1	1 1
1 -1	0
1 -1	
1 2 -1	1
2 2 -1	1

Ejercicio 3 [2.5 puntos]

Extiende el TAD Cola visto en clase (`Queue.h`) con una nueva operación interna y pública cuya cabecera en C++ es

```
void cuela(const T& a, const T& b);
```

que mueve al elemento `b` de su posición a la posición inmediatamente detrás del elemento `a`. En caso de haber múltiples apariciones de los elementos `a` y/o `b` se considerará: la primera aparición de `a`, y la primera aparición de `b` tras la primera aparición de `a`. Si alguno de los elementos no se encuentra en la cola, o bien, si `b` no aparece detrás de `a`, la operación no tendrá efecto. Indica y justifica la complejidad de la operación implementada. *Requisitos:* No se puede crear ni destruir memoria dinámica, ni tampoco modificar los valores almacenados en la cola.

La función principal proporcionada para hacer pruebas lee la cola de enteros (secuencia de enteros no negativos acabando la lectura con un `-1` que no se incluye en la cola), después los enteros `a` y `b`, llama a la función pedida, y muestra por pantalla la cola resultante (ver ejemplos). El proceso se repite hasta introducir una cola vacía (es decir, un `-1`).

Entrada	Salida
1 2 3 4 -1	
1 3	1 3 2 4
1 2 3 4 -1	
1 4	1 4 2 3
1 2 3 4 -1	
2 1	1 2 3 4
3 1 2 1 3 4 -1	
1 3	3 1 3 2 1 4

Ejercicio 4 [2.5 puntos]

Apartado a) Extiende el TAD TreeMap visto en clase con una nueva operación interna y pública de nombre `keyInBounds`, que recibe una clave `k` y devuelve un booleano que será `true` si y solo si en la tabla existe alguna clave menor o igual que `k` y también alguna mayor o igual. Justifica la complejidad de la operación implementada.

Apartado b) Implementa una operación con igual nombre y comportamiento a la del apartado a) pero como función externa al TAD TreeMap, que por tanto recibe como parámetros un `TreeMap<K, V>` y una clave, y devuelve un booleano. Justifica la complejidad de la operación implementada.

Entrada	Salida
5 2 8 -1	
3	si
5 2 8 -1	
9	no
8 2 5 -1	
7	si
8 2 5 -1	
9	no

La función principal proporcionada para hacer pruebas en ambos apartados hace lo siguiente: construye un `TreeMap<int, Nada>` a partir de una secuencia de enteros no negativos que se van insertando, acabando la lectura con un `-1` que no se inserta, después lee la clave `k` (un entero), llama al método o a la función pedida, y muestra por pantalla el resultado obtenido en el formato “si” o “no” (ver ejemplos). El proceso se repite hasta introducir una secuencia vacía (es decir, un `-1`).

Instrucciones

- Entrega un fichero .cpp para cada ejercicio, nombrado `ejerX.cpp` siendo X el número del ejercicio, excepto para el ejerc. 3 y el apartado a) del 4 en los que se entregará el `Queue.h` y `TreeMap.h` resp.
- Al principio de cada fichero debe aparecer, en un comentario, vuestro nombre y apellidos, dni y puesto de laboratorio. También debéis incluir unas líneas explicando qué habéis conseguido hacer y qué no.
- Todo lo que no sea código C++ (explicaciones, especificaciones, invariantes, etc.) debe ir en los propios ficheros en comentarios debidamente indicados.
- Los TADs vistos y las plantillas para poder probar vuestras soluciones se obtienen pulsando en el icono del Escritorio “Publicacion docente ...”, después en “Alumno recogida docente”, y en el programa que se abre, abriendo en la parte derecha la carpeta EDA-F, arrastrando los ficheros a hlocal (en la izqda).
- La entrega se realiza pulsando en el icono del escritorio “Exámenes en Labs ...”, y posteriormente, utilizando el programa que se abre, colocando los ficheros a entregar en la carpeta de vuestro puesto (en el lado derecho).

Estructuras de Datos y Algoritmos

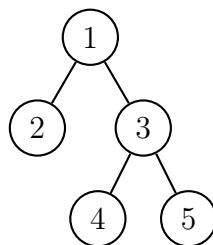
Grados en Ingeniería Informática

Examen Final, 12 de septiembre de 2017

Nombre: _____ Grupo: _____

Laboratorio: _____ Puesto: _____ Usuario de DOMjudge: _____

- (2 puntos)** Se desea contar el número de elementos de un vector que cumplen que la suma de los elementos a su izquierda es menor estricto que la suma de los elementos a su derecha. Se pide especificar una función que resuelva el problema; implementar como cuerpo de dicha función un algoritmo iterativo eficiente que resuelva el problema; escribir los invariantes y funciones de cota que permitan demostrar la corrección del algoritmo propuesto, y justificar adecuadamente el coste asintótico en el caso peor del algoritmo.
- (3 puntos)** Dado un conjunto de $n > 0$ números enteros (n par), se desea dividir el conjunto en dos subconjuntos con $n/2$ elementos cada uno de forma que la diferencia (en valor absoluto) entre las sumas de los elementos de los subconjuntos sea mínima. Implementar un algoritmo que devuelva una solución mínima e imprima por pantalla su valor.
- (2 puntos)** Implementa una función que, dado un árbol binario de enteros y un número entero no negativo k , determine el número de hojas cuya profundidad es mayor que k . Por ejemplo, para el siguiente árbol



la función devolvería 3 si $k = 0$ o $k = 1$, devolvería 2 si $k = 2$ y devolvería 0 si $k \geq 3$.

A parte de implementar este subprograma, debes indicar la complejidad del mismo.

- (3 puntos)** La DGT nos ha pedido ayuda para gestionar el *carnet por puntos*. Los conductores están identificados de manera única por su DNI y la cantidad de puntos de un conductor está entre 0 y 15 puntos inclusives. La implementación del sistema se deberá realizar como un TAD CarnetPorPuntos con las siguientes operaciones:

- **nuevo(dni):** Añade un nuevo conductor identificado por su `dni` (un `string`), con 15 puntos. En caso de que el `dni` esté duplicado, la operación lanza un error “Conductor duplicado”.
- **quitar(dni, puntos):** Le resta puntos a un conductor tras una infracción. Si a un conductor se le quitan más puntos de los que tiene, se quedará con 0 puntos. En caso de que el conductor no exista, lanza un error “Conductor inexistente”.
- **consultar(dni):** Devuelve los puntos actuales de un conductor. En caso de que el conductor no exista, lanza un error “Conductor inexistente”.

- **cuantos_con_puntos(puntos)**: Devuelve cuántos conductores tienen un determinado número de puntos. En caso de que el número de puntos no esté entre 0 y 15 lanza un error “Puntos no válidos”.

La implementación de las operaciones debe ser lo más eficiente posible. Por tanto, debes elegir una representación adecuada para el TAD, implementar las operaciones y justificar la complejidad resultante.

Estructura de Datos y Algoritmos

Grados de la Facultad de Informática de la UCM (Grupos C y F). Curso 2016-2017

Examen parcial 2º cuatr. de septiembre Tiempo: 3 horas

Ejercicio 1 [2.75 puntos]

Apartado a) [2.5 puntos] (Ejercicio 3 del final)

Apartado b) [0.25 puntos] Indica y justifica la complejidad en tiempo y en espacio que tendría la operación del apartado a) si se implementase como una función externa al TAD Queue.

Ejercicio 2 [2.75 puntos]

Apartado a) [1.5 puntos] (Ejercicio 3a del final)

Apartado b) [1 punto] (Ejercicio 3a del final)

Apartado c) [0.25 puntos] Indica y justifica las complejidades que tendrían la operación y la función de los apartados a) y b) para el caso de un **HashMap** en lugar de un **TreeMap**.

Ejercicio 3 [4.5 puntos]

Nos han encargado implementar un sistema para la gestión de la admisión en el Servicio de Urgencias de un Hospital. Cuando un paciente llega al servicio, se le toman los datos, se le asigna un código de identificación único (un número entero no negativo), y también se le asigna un nivel de urgencia, dependiendo de su estado (leve, normal, o grave). A partir de ahí, el paciente espera a ser atendido. El orden de atención da prioridad a los pacientes graves sobre los normales, y a los normales sobre los leves. Una vez atendido un paciente, sus datos se eliminan del sistema. Así mismo, en cualquier momento un paciente puede desistir de ser atendido. En este caso, en el control de salida se le solicita su número de identificación, y se elimina todo rastro de él del sistema. La implementación del sistema se deberá realizar como un TAD **GestionUrgencias** con las siguientes operaciones:

- **crea()**: Operación constructora que crea un sistema de gestión de urgencias vacío.
- **anadirPaciente(codigo, nombre, edad, sintomas, gravedad)**: Añade al sistema un nuevo paciente con código de identificación **codigo**, con nombre **nombre**, con edad **edad**, con una descripción de síntomas **sintomas**, y con código de gravedad **gravedad**. En caso de que el código ya se encontrase en el sistema la operación lanza una excepción con el mensaje “Paciente duplicado”.
- **infoPaciente(codigo, nombre, edad, sintomas)**: Devuelve en **nombre**, **edad** y **sintomas** la información correspondiente del paciente con código **codigo**. En caso de que el código no exista, se lanza una excepción con el mensaje “Paciente inexistente”.
- **siguiente(codigo, gravedad)**: Devuelve en **codigo** y **gravedad**, respectivamente, el código y la gravedad del siguiente paciente a ser atendido. Como se ha indicado antes, se atiende primero a los pacientes graves, después a los de nivel de gravedad normal, y por último a los leves. Dentro de cada nivel, los pacientes se atienden por orden de llegada. En caso de que no haya más pacientes se lanza una excepción con el mensaje “No hay pacientes”.
- **hayPacientes()**: Devuelve **true** si hay más pacientes en espera, y **false** en otro caso.
- **elimina(codigo)**: Elimina del sistema todo el rastro del paciente con código **codigo**. Si no existe tal paciente, la operación no tiene efecto.

Se pide implementar al completo el TAD **GestionUrgencias** incluyendo todas las operaciones indicadas así como justificar la complejidad de cada operación. Debes elegir una representación adecuada para el TAD de manera que las operaciones sean eficientes.

Instrucciones

- Entrega un fichero .cpp para cada ejercicio, nombrado ejerX.cpp siendo X el número del ejercicio, excepto para los apartados a) de los ejercicios 3 y 4 en los que se entregará el Queue.h y TreeMap.h resp.

- Al principio de cada fichero debe aparecer, en un comentario, vuestro nombre y apellidos, dni y puesto de laboratorio. También debéis incluir unas líneas explicando qué habéis conseguido hacer y qué no.
- Todo lo que no sea código C++ (explicaciones, especificaciones, invariantes, etc.) debe ir en los propios ficheros en comentarios debidamente indicados.
- Los TADs vistos y las plantillas para poder probar vuestras soluciones se obtienen pulsando en el icono del Escritorio “Publicacion docente ...”, después en “Alumno recogida docente”, y en el programa que se abre, abriendo en la parte derecha la carpeta EDA-F, arrastrando los ficheros a hlocal (en la izqda).
- La entrega se realiza pulsando en el icono del escritorio “Examenes en Labs ...”, y posteriormente, utilizando el programa que se abre, colocando los ficheros a entregar en la carpeta de vuestro puesto (en el lado derecho).

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática

Examen Segundo Cuatrimestre, 12 de septiembre de 2017

Nombre: _____ Grupo: _____

Laboratorio: _____ Puesto: _____ Usuario de DOMjudge: _____

1. (2.5 puntos) Extiende el TAD Cola implementado con lista enlazada simple visto en clase con una nueva operación cuya cabecera en C++ es

```
void llevarAlPrincipio(unsigned int pos);
```

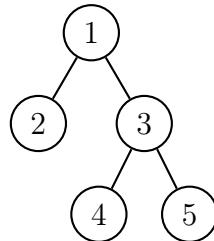
que lleva el elemento que está en la posición *pos* de la cola a la primera posición de la misma. Ten en cuenta que *pos* = 1 es el primer elemento de la cola; *pos* = 2 es el segundo; y así sucesivamente.

Si la posición de la cola no existe, deberá señalarse un error “Posicion inexistente”.

Aparte de implementar esta operación, debes indicar la complejidad de la misma.

Para resolver este ejercicio no se puede crear ni destruir memoria dinámica, ni tampoco modificar los valores almacenados en la cola.

2. (2.5 puntos) Implementa una función que, dado un árbol binario de enteros y un número entero no negativo *k*, determine el número de hojas cuya profundidad es mayor que *k*. Por ejemplo, para el siguiente árbol



la función devolvería 3 si *k* = 0 o *k* = 1, devolvería 2 si *k* = 2 y devolvería 0 si *k* ≥ 3.

Aparte de implementar este subprograma, debes indicar la complejidad del mismo.

3. (5 puntos) La DGT nos ha pedido ayuda para gestionar el *carnet por puntos*. Los conductores están identificados de manera única por su DNI y la cantidad de puntos de un conductor está entre 0 y 15 puntos inclusivos. La implementación del sistema se deberá realizar como un TAD CarnetPorPuntos con las siguientes operaciones:

- **nuevo(dni):** Añade un nuevo conductor identificado por su *dni* (un *string*), con 15 puntos. En caso de que el *dni* esté duplicado, la operación lanza un error “Conductor duplicado”.
- **quitar(dni, puntos):** Le resta puntos a un conductor tras una infracción. Si a un conductor se le quitan más puntos de los que tiene, se quedará con 0 puntos. En caso de que el conductor no exista, lanza un error “Conductor inexistente”.

- **recuperar(dni, puntos)**: Le añade puntos a un conductor enmendado. Si debido a una recuperación un conductor supera los 15 puntos, se quedará con 15 puntos. En caso de que el conductor no exista, lanza un error “Conductor inexistente”.
- **consultar(dni)**: Devuelve los puntos actuales de un conductor. En caso de que el conductor no exista, lanza un error “Conductor inexistente”.
- **cuantos_con_puntos(puntos)**: Devuelve cuántos conductores tienen un determinado número de puntos. En caso de que el número de puntos no esté entre 0 y 15 lanza un error “Puntos no válidos”.
- **lista_por_puntos(puntos)**: Produce una lista con los DNI de los conductores que poseen un número determinado de puntos. La lista estará ordenada por el momento en el que el conductor pasó a tener esos puntos, primero el que menos tiempo lleva con esos puntos. En caso de que el número de puntos no esté entre 0 y 15 lanza un error “Puntos no válidos”.

La implementación de las operaciones debe ser lo más eficiente posible. Por tanto, debes elegir una representación adecuada para el TAD, implementar las operaciones y justificar la complejidad resultante.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática

Examen Final, 27 de Junio de 2017.

Nombre: _____ Grupo: _____

Laboratorio: _____ Puesto: _____ Usuario de DOMjudge: _____

1. (3 puntos) Una **sucesión de Fibonacci genérica** es aquella en la que los dos primeros términos de la sucesión son dos números enteros cualesquiera. Por ejemplo, la siguiente sucesión cumple que es una sucesión de Fibonacci genérica:

3 7 10 17 27 44 71 115

Especifica, diseña e implementa un algoritmo iterativo que, dado un vector de números enteros, calcule la longitud del mayor segmento que cumpla que sus elementos forman una sucesión de Fibonacci genérica. Escribe el invariante y función de cota que permitan demostrar la corrección del algoritmo implementado. Por último, calcula y justifica el coste del algoritmo conseguido.

La función recibirá la longitud y el vector, y dará como salida, en líneas separadas, la longitud del segmento más largo.

Entrada		Salida
n	v	
1	3	1
2	3 7	2
2	7 3	2
3	3 7 11	2
3	3 7 10	3
4	1 1 3 7	2
4	1 2 3 7	3
4	3 7 10 17	4
4	-1 6 5 10	3
4	5 3 1 2	2
4	5 3 -1 2	3

2. (2 puntos) Se dice que un vector de números enteros es **tímido** si sus elementos están ordenados y ningún elemento aparece más veces de lo que indica su valor. Por ejemplo, serían tímidos

1 2 2 3 3 3 4 4 4 2 3 3 5 5 5 5 5

pero no lo serían

0 1 2 2 3 3 3 4 4 1 1 2 2 3 3 3 7

Se desea implementar un algoritmo que, dada una longitud n , un valor inicial ini y un valor final fin , con $ini < fin$, escriba todos los posibles vectores tímidos de longitud n con valores comprendidos entre ini y fin . Los vectores se deberán escribir en orden lexicográfico.

El programa leerá de la entrada los valores de n , ini y fin y escribirá, en líneas separadas, los distintos vectores tímidos obtenidos.

Entrada			Salida
n	ini	fin	
4	2	4	2 2 3 3
			2 2 3 4
			2 2 4 4
			2 3 3 3
			2 3 3 4
			2 3 4 4
			2 4 4 4
			3 3 3 4
			3 3 4 4
			3 4 4 4
			4 4 4 4

3. (2 puntos) Un nodo de un árbol binario de enteros se dice que es *singular* si la suma de los valores almacenados en sus nodos antepasados (incluido el que contiene la raíz) es igual a la suma de los valores almacenados en sus nodos descendientes. Implementa un subprograma que, dado un árbol binario de enteros, devuelva el número de nodos singulares que tiene.

A parte de implementar este subprograma, debes indicar la complejidad del mismo.

4. (3 puntos) Nos han encargado implementar un sistema para la gestión de la admisión en el Servicio de Urgencias de un Hospital. Cuando un paciente llega al servicio, se le toman los datos y se le asigna un código de identificación único (un número entero no negativo). A partir de ahí, el paciente espera a ser atendido. El orden de atención da prioridad a los pacientes por orden de llegada. Una vez atendido un paciente, sus datos se eliminan del sistema. Así mismo, en cualquier momento un paciente puede desistir de ser atendido. En este caso, en el control de salida se le solicita su número de identificación, y se elimina todo rastro de él del sistema.

La implementación del sistema se deberá realizar como un TAD `GestionAdmisiones` con las siguientes operaciones:

- `crea()`: Operación constructora que crea un sistema de gestión de admisiones vacío.
- `an_paciente(codigo, nombre, edad, sintomas)`: añade al sistema un nuevo paciente con código de identificación `codigo`, con nombre `nombre`, con edad `edad` y con una descripción de síntomas `sintomas`. En caso de que el código esté duplicado, la operación señala un error “*Paciente duplicado*”.
- `info_paciente(codigo, nombre, edad, sintomas)`: `nombre`, `edad` y `sintomas` devuelven la información correspondiente al paciente con código `codigo`. En caso de que el código no exista, levanta un error “*Paciente inexistente*”.
- `siguiente(codigo)`: almacena en `codigo` el código del siguiente paciente a ser atendido. En caso de que no haya más pacientes, esta operación levanta un error “*No hay pacientes*”.
- `hay_pacientes()`: devuelve `true` si hay más pacientes en espera, y `false` en otro caso.
- `elimina(codigo)`: elimina del sistema todo el rastro del paciente con código `codigo`. Si no existe tal paciente, la operación no tiene efecto.

Dado que este es un sistema crítico, la implementación de las operaciones debe ser lo más eficiente posible. Por tanto, debes elegir una representación adecuada para el TAD, implementar las operaciones y justificar la complejidad resultante.

Estructuras de datos y algoritmos

Grados en Informática (UCM)

FINAL DE JUNIO

Curso 2016/2017

Ejercicio 4 (2.25 puntos) Se tienen dos vectores de enteros ordenados a y b con n y m elementos respectivamente. Se desea calcular el *beneficio* del vector a con respecto a b . El beneficio del vector se calcula sumando los beneficios obtenidos por cada uno de sus elementos. El beneficio de un elemento $a[i]$ es el producto de la cantidad de elementos de b que son estrictamente menores que $a[i]$ y la cantidad de elementos de b que son estrictamente mayores que $a[i]$. Se pide:

1. (0,5 puntos) Especificar una función *beneficio*, que dados a , b , n y m como se indica en el párrafo anterior calcule el *beneficio* del vector a con respecto a b .
2. (1 punto) Implementar como cuerpo de dicha función un algoritmo iterativo eficiente que resuelva el problema.
3. (0,5 puntos) Escribir el invariante que permite demostrar la corrección del algoritmo propuesto y una función de cota.
4. (0,25 puntos) Indicar y justificar adecuadamente el coste asintótico en el caso peor del algoritmo.

Los apartados 1, 3 y 4 pueden hacerse en papel o en el fichero con el código. Para el apartado 2, se completará el código del fichero `main4.cpp`, en el que ya está implementada la entrada/salida.

Ejercicio 5 (1.5 puntos) Se tienen dos vectores de enteros ordenados y distintos entre sí a y b con n y $n - 1$ elementos respectivamente. Los elementos de b son los mismos que tiene a excepto uno que falta. Se pide implementar un algoritmo recursivo eficiente que encuentre ese valor que falta. Se debe indicar la recurrencia y el coste asintótico en el caso peor del algoritmo.

Para resolver este ejercicio, se completará el código del fichero `main5.cpp`, en el que ya está implementada la entrada/salida.

Ejercicio 6 (2 puntos) Queremos añadir a la clase `set` un nuevo método

```
std::pair<bool, T> lower_bound(T const& e) const;
```

que dado un elemento e devuelva si existen en el conjunto elementos mayores o iguales que e y en caso afirmativo cuál es el menor de ellos.

Copia el fichero `set_eda.h` en `set_modificado.h` y haz las modificaciones en este fichero. Escribe al principio un comentario indicando cuáles han sido las modificaciones, justificando tu solución. Para probar el nuevo método, el programa principal resolverá varios casos. En cada caso, se leerá una serie de valores que se añadirán a un conjunto. Después se leerá otra serie de valores (preguntas) y para cada uno de ellos se escribirá su *lower bound*, si existe. Este código se encuentra en el fichero `main6.cpp`, que **no** puede ser modificado.

Ejercicio 7 (4.25 puntos) Se desea diseñar un TAD para gestionar los alumnos de los distintos profesores de una autoescuela (tanto alumnos como profesores se identifican por su nombre, que es un `string`). Para ello se desea disponer de las siguientes operaciones:

- constructora: al comienzo ningún profesor tiene asignados alumnos.
- `alta(A, P)`: sirve tanto para dar de alta a un alumno como para cambiarle de profesor. Si el alumno no estaba matriculado en la autoescuela se le da una puntuación de cero. Si ha cambiado de profesor, se le da de alta con el nuevo, con la puntuación que tuviera, y se le da de baja con el anterior. La puntuación determinará quién se puede examinar.
- `es_alumno(A, P)`: comprueba si el alumno A está matriculado actualmente con el profesor P .

- `actualizar(A, N)`: aumenta en una cantidad `N` la puntuación del alumno `A`. Si el alumno no está dado de alta con ningún profesor, entonces se lanza una excepción `domain_error` con mensaje `El alumno A no esta matriculado.`
- `examen(P, N)`: obtiene una lista con los alumnos del profesor `P`, ordenados alfabéticamente, que se presentarán a examen por tener una puntuación mayor o igual a `N` puntos.

Implementar de forma eficiente el TAD `autoescuela`, justificando la representación elegida e indicando la complejidad de las operaciones implementadas.

El programa principal resolverá varios casos de prueba. Para cada caso leerá una serie de operaciones (con sus argumentos) y las irá aplicando a una autoescuela inicialmente vacía. Todo el código aparece en el fichero `main7.cpp`, que **no** puede ser modificado.

Normas de realización del examen

1. Debes programar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección <http://exacrc/domjudge/team>.
2. Escribe comentarios que expliquen tu solución, justifiquen por qué se ha hecho así y ayuden a entenderla. Calcula la complejidad de todas las funciones que implementes.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen. El nombre de usuario y contraseña que has estado utilizando durante la evaluación continua **no** son válidos.
4. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que subas al juez.
5. Puedes descargar el fichero <http://exacrc/documEDG.zip> que contiene material que puedes utilizar para la realización del examen (implementación de las estructuras de datos, ficheros con código fuente para completar y ficheros de texto con los casos de prueba de cada ejercicio del enunciado).
6. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática

Examen Segundo Cuatrimestre, 27 de Junio de 2017.

Nombre: _____ Grupo: _____

Laboratorio: _____ Puesto: _____ Usuario de DOMjudge: _____

1. (2.5 puntos) Extiende el TAD Cola implementado con lista enlazada simple visto en clase con una nueva operación cuya cabecera en C++ es

```
void desplaza (unsigned int pos, unsigned int dist);
```

que desplaza al elemento que está en la posición *pos* de la cola un número *dist* de posiciones hacia la izquierda. Ten en cuenta que *pos* = 1 es el primer elemento de la cola; *pos* = 2 es el segundo; y así sucesivamente.

Si el valor de *dist* es demasiado grande y no hay suficientes posiciones para hacer el desplazamiento, el elemento quedará situado en la primera posición de la cola.

Si la posición de la cola no existe, deberá señalarse un error “Posicion inexistente”.

Aparte de implementar esta operación, debes indicar la complejidad de la misma.

Para resolver este ejercicio no se puede crear ni destruir memoria dinámica, ni tampoco modificar los valores almacenados en la cola.

Entrada							Salida						
n	v						pos	dist					
7	1	2	3	4	5	6	7	4	1	1	2	4	3
7	1	2	3	4	5	6	7	4	2	1	4	2	3
7	1	2	3	4	5	6	7	4	3	4	1	2	3
7	1	2	3	4	5	6	7	4	4	4	1	2	3
7	1	2	3	4	5	6	7	4	0	1	2	3	4
7	1	2	3	4	5	6	7	0	3	1	2	3	4
7	1	2	3	4	5	6	7	8	5	1	2	3	4

2. (3 puntos) Un nodo de un árbol binario de enteros se dice que es *singular* si la suma de los valores almacenados en sus nodos antepasados (incluido el que contiene la raíz) es igual a la suma de los valores almacenados en sus nodos descendientes. Implementa un subprograma que, dado un árbol binario de enteros, devuelva el número de nodos singulares que tiene.

Aparte de implementar este subprograma, debes indicar la complejidad del mismo.

3. (4.5 puntos) Nos han encargado implementar un sistema para la gestión de la admisión en el Servicio de Urgencias de un Hospital. Cuando un paciente llega al servicio, se le toman los datos, se le asigna un código de identificación único (un número entero no negativo), y también se le asigna un nivel de urgencia, dependiendo de su estado (leve, normal, o grave). A partir de ahí, el paciente espera a ser atendido. El orden de atención da prioridad a los pacientes graves sobre los normales, y a los normales sobre los leves. Una vez atendido un paciente, sus datos se eliminan del sistema. Así mismo, en cualquier momento un paciente puede desistir de ser atendido. En este caso, en el control de salida se le solicita su número de identificación, y se elimina todo rastro de él del sistema.

La implementación del sistema se deberá realizar como un TAD `GestionAdmisiones` con las siguientes operaciones:

- `crea()`: Operación constructora que crea un sistema de gestión de admisiones vacío.
- `an_paciente(codigo, nombre, edad, sintomas, gravedad)`: Añade al sistema un nuevo paciente con código de identificación `codigo`, con nombre `nombre`, con edad `edad`, con una descripción de síntomas `sintomas`, y con código de gravedad `gravedad`. En caso de que el código esté duplicado, la operación señala un error “Paciente duplicado”.
- `info_paciente(codigo, nombre, edad, sintomas)`: Almacena en `nombre`, `edad` y `sintomas` la información correspondiente del paciente con código `codigo`. En caso de que el código no exista, levanta un error “Paciente inexistente”.
- `siguiente(codigo, gravedad)`: Almacena en `codigo` y `gravedad`, respectivamente, el código y la gravedad del siguiente paciente a ser atendido. Como se ha indicado antes, se atiende primero a los pacientes graves, después a los de nivel de gravedad normal, y por último a los leves. Dentro de cada nivel, los pacientes se atienden por orden de llegada. En caso de que no haya más pacientes, esta operación levanta un error “No hay pacientes”.
- `hay_pacientes()`: Devuelve `true` si hay más pacientes en espera, y `false` en otro caso.
- `elimina(codigo)`: Elimina del sistema todo el rastro del paciente con código `codigo`. Si no existe tal paciente, la operación no tiene efecto.

Dado que este es un sistema crítico, la implementación de las operaciones debe ser lo más eficiente posible. Por tanto, debes elegir una representación adecuada para el TAD, implementar las operaciones y justificar la complejidad resultante.

Estructuras de datos y algoritmos

Grados en Informática (UCM)

SEGUNDO PARCIAL DE JUNIO

Curso 2016/2017

Ejercicio 1 (2.5 puntos) Queremos añadir a la clase `unordered_map` un nuevo método

```
iterator erase(iterator const& it);
```

que dado un iterador apuntando a un par $\langle \text{clave}, \text{valor} \rangle$, elimine dicho par del diccionario y devuelva un iterador al siguiente par. Si el iterador recibido no apunta a ningún par, la operación produce un error. Para llevar a cabo esta operación no puede utilizarse la función `hash`.

Copia el fichero `hashmap_eda.h` en `hashmap_modificado.h` y haz las modificaciones en este fichero. Escribe al principio un comentario indicando las modificaciones que has hecho, justificando tu solución. Para probar el nuevo método, el programa principal resolverá varios casos. En cada caso, se leerá una serie de claves que serán introducidas en un diccionario. Después este se recorrerá eliminando las claves pares y a continuación se recorrerá una vez más comprobando que las claves actuales son todas impares. Este código se encuentra en el fichero `main1.cpp`, que **no** puede ser modificado.

Ejercicio 2 (2 puntos) Queremos añadir a la clase `set` un nuevo método

```
std::pair<bool, T> lower_bound(T const& e) const;
```

que dado un elemento `e` devuelva si existen en el conjunto elementos mayores o iguales que `e` y en caso afirmativo cuál es el menor de ellos.

Copia el fichero `set_eda.h` en `set_modificado.h` y haz las modificaciones en este fichero. Escribe al principio un comentario indicando cuáles han sido las modificaciones, justificando tu solución. Para probar el nuevo método, el programa principal resolverá varios casos. En cada caso, se leerá una serie de valores que se añadirán a un conjunto. Después se leerá otra serie de valores (preguntas) y para cada uno de ellos se escribirá su *lower bound*, si existe. Este código se encuentra en el fichero `main2.cpp`, que **no** puede ser modificado.

Ejercicio 3 (5.5 puntos) Se desea diseñar un TAD para gestionar los alumnos de los distintos profesores de una autoescuela (tanto alumnos como profesores se identifican por su nombre, que es un `string`). Para ello se desea disponer de las siguientes operaciones:

- constructora: al comienzo ningún profesor tiene asignados alumnos.
- `alta(A, P)`: sirve tanto para dar de alta a un alumno como para cambiarle de profesor. Si el alumno no estaba matriculado en la autoescuela se le da una puntuación de cero. Si ha cambiado de profesor, se le da de alta con el nuevo, con la puntuación que tuviera, y se le da de baja con el anterior. La puntuación determinará quién se puede examinar.
- `es_alumno(A, P)`: comprueba si el alumno `A` está matriculado actualmente con el profesor `P`.
- `puntuacion(A)`: devuelve los puntos que tiene el alumno `A`. Si el alumno no está dado de alta con ningún profesor, entonces se lanza una excepción `domain_error` con mensaje `El alumno A no esta matriculado`.
- `actualizar(A, N)`: aumenta en una cantidad `N` la puntuación del alumno `A`. Si el alumno no está dado de alta con ningún profesor, entonces se lanza una excepción `domain_error` con mensaje `El alumno A no esta matriculado`.
- `examen(P, N)`: obtiene una lista con los alumnos del profesor `P`, ordenados alfabéticamente, que se presentarán a examen por tener una puntuación mayor o igual a `N` puntos.
- `aprobar(A)`: el alumno `A` aprueba el examen y es borrado de la autoescuela, junto con toda la información que de él existiera. Si el alumno no está dado de alta con ningún profesor, entonces se lanza una excepción `domain_error` con mensaje `El alumno A no esta matriculado`.

Implementar de forma eficiente el TAD `autoescuela`, justificando la representación elegida e indicando la complejidad de las operaciones implementadas.

El programa principal resolverá varios casos de prueba. Para cada caso leerá una serie de operaciones (con sus argumentos) y las irá aplicando a una autoescuela inicialmente vacía. Todo el código aparece en el fichero `main3.cpp`, que **no** puede ser modificado.

Normas de realización del examen

1. Debes programar soluciones para cada uno de los tres ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección <http://exacrc/domjudge/team>.
2. Escribe comentarios que expliquen tu solución, justifiquen por qué se ha hecho así y ayuden a entenderla. Calcula la complejidad de todas las funciones que implementes.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen. El nombre de usuario y contraseña que has estado utilizando durante la evaluación continua **no** son válidos.
4. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que subas al juez.
5. Puedes descargar el fichero <http://exacrc/documEDG.zip> que contiene material que puedes utilizar para la realización del examen (implementación de las estructuras de datos, ficheros con código fuente para completar y ficheros de texto con los casos de prueba de cada ejercicio del enunciado).
6. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.

Estructura de Datos y Algoritmos

Grado en Ingeniería del Software (Grupo F). Curso 2016-2017
Examen parcial de febrero Tiempo: 3 horas

Instrucciones

- Debéis entregar un fichero .cpp para cada ejercicio, nombrado ejerX.cpp siendo X el número del ejercicio.
- Al principio de cada fichero que entreguéis debe aparecer, en un comentario, vuestro nombre y apellidos, dni, puesto de laboratorio y usuario del juez si lo habéis usado. También debéis incluir unas líneas explicando qué habéis conseguido hacer y qué no.
- Todo lo que no sea código C++ (explicaciones, especificaciones, invariantes, etc.) debe ir en los propios ficheros .cpp en comentarios debidamente indicados.
- Si queréis utilizar el juez online, id a la mesa del profesor a recoger un usuario/password y entrad en <http://exacrc/domjudge/team>
- Las plantillas para poder probar vuestras funciones se obtienen pulsando en el icono del Escritorio “Publicacion docente ...”, después en “Alumno recogida docente”, y en el programa que se abre, abriendo en la parte derecha la carpeta EDA-F, arrastrando a hlocal (en la izquierda) el fichero PlantillasGrupoF.cpp.
- La entrega se realiza pulsando en el icono del escritorio “Examenes en Labs ...”, y posteriormente, utilizando el programa que se abre, colocando los ficheros a entregar en la carpeta de vuestro puesto (en el lado derecho).

Ejercicio 1 [4 puntos]

Dado un vector de números enteros positivos, ordenado en orden estrictamente creciente, se pide diseñar un algoritmo eficiente que elimine todos los números impares, dejando sólo los pares, ordenados de forma creciente. El procedimiento modificará el vector de entrada y su tamaño, y no deberá utilizar ningún array auxiliar. Los parámetros por lo tanto serán de entrada/salida.

Se pide:

- a) (1 punto) Especificar el problema.
- b) (1,5 puntos) Diseñar e implementar un algoritmo que resuelva el problema.
- c) (1 punto) Escribir un invariante y una función cota que permitan demostrar la corrección del algoritmo implementado.
- d) (0,5 puntos) Justificar el coste del algoritmo.

Entrada/Salida para el corrector automático

La entrada que espera el corrector automático consta de una serie de casos de prueba y acabará cuando se introduzca un -1 . Cada caso de prueba consta de dos líneas, en la primera se indica el número de elementos del vector. En la segunda se dan los elementos, ordenados en orden estrictamente creciente y separados por el carácter blanco. Se cumple que el número de elementos del vector es mayor o igual que cero y cada componente del vector es estrictamente mayor que cero.

Para cada caso de prueba se escribe una línea con los elementos que quedan en el vector después de eliminar los elementos impares dejando un espacio blanco entre cada par de elementos. Si en el vector no queda ningún elemento se deja una línea en blanco.

Entrada	Salida
5 3 5 6 8 10	6 8 10
4 2 4 5 7	2 4
5 1 3 5 7 9	
5 2 4 6 8 10	2 4 6 8 10
6 3 6 8 9 13 14	6 8 14

Ejercicio 2 [3 puntos]

Se tiene un vector con $k > 0$ valores enteros diferentes. Cada valor se encuentra repetido k_i veces, encontrándose consecutivos todos los valores iguales del vector. Se pide encontrar el número de valores distintos. El problema se debe resolver empleando la técnica de divide y vencerás. Si existen valores repetidos, el vector no debe recorrerse completo, evitando siempre que se pueda el acceso a los elementos repetidos.

Entrada/Salida para el corrector automático

La entrada que espera el corrector automático consta de una serie de casos de prueba y acabará cuando se introduzca un 0. Cada caso de prueba consta de dos líneas: en la primera se indica el número de elementos del vector y en la segunda los valores separados por un espacio.

Para cada caso de prueba se escribe el número de valores diferentes del vector en una línea.

Entrada	Salida
4 <intro> 3 3 1 1	2
3 <intro> 2 2 2	1
8 <intro> 6 6 3 3 8 8 1 1	4
10 <intro> 3 3 3 6 6 2 2 2 2 2	3
14 <intro> 7 7 7 7 4 4 4 4 9 9 2 2 2	4
20 <intro>	
5 5 5 5 5 5 5 4 4 4 4 4 1 1 1 7 9 2	6
1 <intro> 5	1
6 <intro> 6 5 5 5 5 2	3
6 <intro> 8 7 6 5 4 3	6

Ejercicio 3 [3 puntos]

Laura quiere construir una torre con piezas de colores. En su juego de construcciones hay piezas azules, rojas y verdes, de cada una de las cuales tiene un determinado número disponible, respectivamente a , r y v . Quiere construir una torre que contenga $n \geq 2$ piezas en total, cada una encima de la anterior. No le gusta el color verde, así que nunca coloca dos piezas verdes juntas, ni permite que el número de piezas verdes supere al de piezas azules en ningún momento mientras se va construyendo la torre. Además, como el color rojo es su favorito, las torres que construye siempre tienen en la parte inferior una pieza roja, y en la torre final el número de piezas rojas debe ser mayor que la suma de las piezas azules y verdes.

Implementar un algoritmo que muestre todas las formas posibles que tiene de construir una torre de la altura deseada cumpliendo con las restricciones mencionadas.

Entrada/Salida para el corrector automático

La entrada que espera el corrector automático consta de una serie de casos de prueba y acabará cuando se introduzca una línea con cuatro ceros. Cada caso de prueba se escribe en una línea y consta de 4 enteros separados por blancos. El primero es mayor que uno y representa la altura de la torre. Los tres siguientes son mayores o iguales que cero y representan los cubos de color azul, rojo y verde respectivamente.

Para cada caso de prueba se escriben todas las posibles torres, una en cada línea ordenadas por orden lexicográfico y separando cada par de colores por un espacio. Cada caso termina con una línea en blanco. Si no se puede construir la torre con los números de bloques dados se escribirá *SIN SOLUCION*.

Entrada	Salida
4 4 4 4	rojo azul rojo rojo rojo rojo azul rojo rojo rojo rojo azul rojo rojo rojo rojo
5 2 2 2	SIN SOLUCION
5 3 3 1	rojo azul azul rojo rojo rojo azul rojo azul rojo rojo azul rojo rojo azul rojo azul rojo rojo verde rojo azul rojo verde rojo rojo azul verde rojo rojo rojo rojo azul azul rojo rojo rojo azul rojo azul rojo rojo azul rojo verde rojo rojo azul verde rojo rojo rojo rojo azul azul rojo rojo rojo azul verde
2 1 2 1	rojo rojo

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática

Examen Primer Cuatrimestre, 10 de Febrero de 2017.

Nombre: _____ **Grupo:** _____

Laboratorio: _____ **Puesto:** _____

1. (4 puntos) Dado un vector de $n \geq 0$ enteros, se pide diseñar un algoritmo eficiente que (sin utilizar un array auxiliar) modifique el vector de forma que todos los elementos mayores o iguales que 0 queden colocados al principio del vector y los estrictamente menores que 0 a continuación, y que adicionalmente devuelva cual es la posición de separación entre positivos y negativos, siendo esta la posición del primer número negativo en el vector modificado o n en caso de que no haya ninguno. Se pide:

1. (1 punto) Especificar el problema.
2. (1,5 puntos) Diseñar e implementar un algoritmo que resuelva el problema.
3. (1 punto) Escribir un invariante y una función de cota que permitan demostrar la corrección del algoritmo implementado.
4. (0,5 puntos) Justificar el coste del algoritmo.

Entrada

La primera línea contiene un número que indica el número de casos de prueba que aparecen a continuación. Cada caso de prueba se compone de dos líneas. La primera de ellas tiene el número de elementos del vector. La segunda contiene los elementos del vector separados por blancos.

Salida

Para cada caso de prueba se escribirá en una línea el vector modificado y en otra línea la posición de separación. Se ha de tener en cuenta que la salida mostrada en el ejemplo es solamente una de las posibles.

Entrada de ejemplo

```
7
1
2
1
-3
6
5 4 1 9 0 2
5
-3 -1 -2 -7 -8
7
1 -3 2 -1 9 -6 -10
10
3 7 -100 1 0 1 4 6 8 100
10
0 -3 -4 -1 -9 -6 0 -5 -10 -20
```

Salida de ejemplo

```
2
1
-3
0
5 4 1 9 0 2
6
-3 -1 -2 -7 -8
0
1 9 2 -1 -3 -6 -10
3
3 7 100 1 0 1 4 6 8 -100
9
0 0 -4 -1 -9 -6 -3 -5 -10 -20
2
```

2. (3 puntos) Supongamos dado un vector ordenado de $n \geq 1$ elementos, en el que todos los elementos aparecen repetidos dos veces, excepto uno que solamente aparece una vez (por tanto n es impar). Se pide diseñar un algoritmo eficiente que devuelva la posición de dicho elemento. Justifica el coste del algoritmo, para lo cual has de plantear la recurrencia correspondiente.

Entrada

La primera línea contiene un número que indica el número de casos de prueba que aparecen a continuación. Cada caso de prueba se compone de dos líneas. La primera de ellas contiene el número de elementos del vector. La segunda contiene los elementos del vector separados por blancos.

Salida

Para cada caso de prueba se escribirá en una línea la posición del elemento que aparece una única vez.

Entrada de ejemplo

```
9
1
3
3
1 2 2
5
1 1 2 9 9
5
1 2 2 9 9
5
1 1 2 2 9
7
3 5 5 9 9 11 11
7
3 3 5 9 9 11 11
7
3 3 5 5 9 11 11
7
3 3 5 5 9 9 11
```

Salida de ejemplo

```
0
0
2
0
4
0
2
4
6
```

3. (3 puntos) Se desea llenar un vector de $2n$ posiciones (siendo $n \geq 1$) con dos apariciones de cada uno de los números 1 a n , de tal forma que el número de posiciones entre dos apariciones (excluyendo a estas) de cada número i es igual a i . Implementar un algoritmo que muestre todas las soluciones posibles y cuantas son.

Entrada

La primera línea contiene un número que indica el número de casos de prueba que aparecen a continuación. Cada caso de prueba se compone de una línea en la que aparece el entero n .

Salida

Para cada caso de prueba se mostrarán las posibles soluciones, cada una en una línea (no necesariamente en el orden que se muestra en el ejemplo) y al finalizar una linea indicando el número de soluciones encontradas, según el formato del ejemplo.

Entrada de ejemplo

```
7
1
2
3
4
5
6
7
```

Salida de ejemplo

```
Soluciones: 0
Soluciones: 0
2 3 1 2 1 3
3 1 2 1 3 2
Soluciones: 2
2 3 4 2 1 3 1 4
4 1 3 1 2 4 3 2
Soluciones: 2
Soluciones: 0
Soluciones: 0
1 4 1 5 6 7 4 2 3 5 2 6 3 7
1 4 1 6 7 3 4 5 2 3 6 2 7 5
1 5 1 4 6 7 3 5 4 2 3 6 2 7
1 5 1 6 3 7 4 5 3 2 6 4 2 7
1 5 1 6 7 2 4 5 2 3 6 4 7 3
1 5 1 7 3 4 6 5 3 2 4 7 2 6
1 6 1 3 5 7 4 3 6 2 5 4 2 7
1 6 1 7 2 4 5 2 6 3 4 7 5 3
1 7 1 2 5 6 2 3 4 7 5 3 6 4
1 7 1 2 6 4 2 5 3 7 4 6 3 5
2 3 6 2 7 3 4 5 1 6 1 4 7 5
2 3 7 2 6 3 5 1 4 1 7 6 5 4
...
Soluciones: 52
```

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática

Examen Primer Cuatrimestre, 9 de Febrero de 2017.

Nombre: _____

Laboratorio: _____ Puesto: _____ Usuario de DOMjudge: _____

Para la realización de los ejercicios que siguen utilizaremos las siguientes definiciones:

- Se dice que un vector es *creciente por los pelos* cuando, además de ser creciente, la diferencia entre un elemento y el elemento siguiente es como mucho uno.

Por ejemplo, los siguientes vectores de tamaño 4 cumplen la definición:

1	2	3	4
---	---	---	---

1	2	2	3
---	---	---	---

1	1	1	1
---	---	---	---

Fíjate que al no exigirse que el vector sea *estrictamente creciente*, el último vector con todos 1's cumple la definición.

Por su parte el vector

1	2	1	2
---	---	---	---

no es creciente por los pelos, pues no es creciente.

- Los vectores aburridos son aquellos en los que hay elementos que aparecen repetidos muchas veces. Se dice que un vector es *d-divertido* cuando ningún elemento se repite más de *d* veces.

Ejemplos de vectores *1-divertido* son

1	2	3	4
---	---	---	---

4	3	7	0
---	---	---	---

pues ningún elemento aparece más de una vez (fíjate que, en realidad, los vectores anteriores son también *10-divertidos*, pues ningún elemento aparece más de 10 veces).

Otros ejemplos de vectores y su grado de diversión más pequeño:

6	7	6	6
---	---	---	---

3-divertido

7	3	7	0
---	---	---	---

2-divertido

7	3	7	3
---	---	---	---

2-divertido

9	9	9	9
---	---	---	---

4-divertido

Las dos definiciones pueden combinarse. Por ejemplo, el vector

1	1	2	2	2	3	4	5
---	---	---	---	---	---	---	---

es *creciente por los pelos* y *3-divertido*

- 1. (4 puntos)** Especifica, diseña e implementa una función que reciba un vector de enteros de longitud $n \geq 0$ y un parámetro $d > 0$ y devuelva si el vector es *creciente por los pelos y d-divertido*. Escribe el invariante y función de cota que permitan demostrar la corrección del algoritmo implementado. Por último, justifica el coste del algoritmo conseguido.

La entrada comienza con una línea que contiene el número de casos de prueba. Cada caso de prueba contendrá los valores de d , n y los elementos del vector. El programa escribirá SI si el vector es *creciente por los pelos y d-divertido* y NO en caso contrario.

- 2. (3 puntos)** Diseña e implementa un algoritmo recursivo que reciba un vector de longitud $n \geq 0$ que *se sabe creciente* (no necesariamente estrictamente creciente) y devuelva si el vector es creciente por los pelos. Justifica el coste del algoritmo implementando.

Se valorarán soluciones que utilicen el esquema *divide y vencerás* y que no recorran el vector completo si no es necesario.

La entrada comienza con una línea que contiene el número de casos de prueba. Cada caso de prueba contendrá el valor de n y los elementos del vector. El programa escribirá SI si el vector es *creciente por los pelos y d-divertido* y NO en caso contrario.

- 3. (3 puntos)** Escribe una función para generar vectores crecientes por los pelos *d-divertidos*. En concreto, la función recibirá el tamaño del vector, el valor de d y el valor para el primer elemento del vector y generará la salida en orden lexicográfico. Se admite el uso de funciones auxiliares.

La entrada comienza con una línea que contiene el número de casos de prueba. La entrada contendrá los valores de n , d , y e . El programa escribirá en líneas separadas cada uno de los vectores del resultado, con los elementos separados por un espacio.

Entrada				Salida	
d	n	v			
1	4	1	2	3	4
1	4	1	2	1	2
1	4	1	2	3	3
1	4	4	3	2	1
2	4	1	2	3	4
2	4	1	2	3	3
2	4	1	1	2	2
2	4	1	1	3	3
2	4	1	1	1	2
2	4	1	1	1	3
5	4	1	1	1	1
5	4	1	1	3	4

Entrada				Salida	
n	v				
4	1	1	1	1	SI
4	1	1	1	2	SI
4	1	1	1	9	NO
4	1	1	2	2	SI
4	1	2	3	4	SI
4	1	2	3	3	SI
4	1	1	3	3	NO
4	1	2	4	5	NO

Entrada			Salida	
n	d	e		
3	3	1	1	1
			1	1
			2	1
			2	2
3	2	1	1	2
			2	1
			2	2
			3	1

```
// Escribe todos los vectores crecientes por los pelos de tamaño n
// que sean además d-divertidos y en los que el primer elemento sea e
void escribeCrecientesPorLosPelosDivertidos(int n, int d, int e);
```

Estructuras de Datos y Algoritmos

Doble Grado, Ingeniería Informática, de Computadores y del Software

Examen final, 9 de Septiembre de 2016.

1. **(2,5 puntos)** Se tiene almacenado en un vector de enteros el recorrido en preorden de un árbol completo de $n \geq 1$ niveles. Diseñar una función booleana recursiva que determine si el árbol que corresponde a dicho recorrido es un árbol binario de búsqueda.

Nota: en el recorrido en preorden de un árbol binario completo de n niveles, si la raíz está en la posición r , el recorrido del subárbol izquierdo abarca $2^{n-1} - 1$ elementos a partir de la posición $r + 1$ incluida, y el recorrido del subárbol derecho otros $2^{n-1} - 1$ elementos a partir de la posición $r + 2^{n-1}$ incluida.

2. **(2,5 puntos)** Un comerciante del desierto dispone de un camello para transportar sus mercancías. El camello tiene dos alforjas cuya capacidad en peso, p_1 y p_2 , son conocidas. El comerciante quiere cargar el camello de tal forma que se maximice el valor de los objetos transportados, teniendo en cuenta que los objetos no se pueden fraccionar. Sean v_i y w_i (ambos positivos) el valor y el peso del objeto i respectivamente ($0 \leq i < n$). Para poder viajar el camello debe ir equilibrado, por lo que se exigirá que la diferencia de pesos entre las alforjas no supere el 5% del peso total. Implementar un algoritmo que le indique al comerciante cómo cargar su camello.

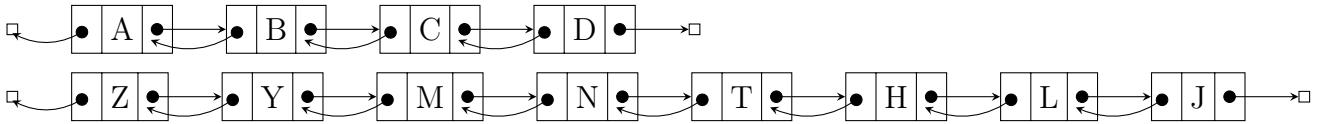
3. **(2 puntos)** Sea la clase `ListaEnlazadaDoble` que implementa listas doblemente enlazadas con punteros al primer y último nodo, no circulares y sin nodo cabecera:

```
template <class T>
class ListaEnlazadaDoble{
private:
    class Nodo {
public:
    Nodo() : _ant(NULL), _sig(NULL) {}
    Nodo(const T &elem) : _elem(elem), _sig(NULL), _ant(NULL) {}
    Nodo(Nodo* ant, const T &elem, Nodo *sig) : _elem(elem),
        _sig(sig), _ant(ant) {}
    T _elem;
    Nodo *_sig;    Nodo *_ant;
    };
    Nodo *_prim;   Nodo *_ult;
    ...
}
```

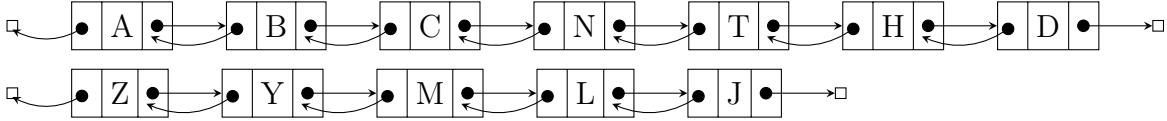
Se pide implementar el método `splice`, que recibe como parámetro una lista 1 y tres valores de tipo T, v1, v2 y v3. El método debe modificar la lista original añadiendo después del valor v1 los nodos de la lista 1 que se encuentran entre los valores v2 y v3 (ambos inclusive), y debe eliminarlos de la lista 1. Si el valor v1 no existe en la lista original, o si alguno de los valores v2, v3 no existe en la lista 1, o si v3 no se encuentra en una posición posterior a v2, las listas no se modifican. Se supone que todos los valores de las listas son diferentes.

La implementación debe ser lo más eficiente posible. Para ello, se debe: evitar liberar/reservar memoria y hacer copias de valores cuando no sea necesario, y recorrer las listas el menor número posible de veces. No se puede utilizar ninguna operación de la clase `ListaEnlazadaDoble` en la implementación.

Por ejemplo, dadas las siguientes listas de caracteres 11 y 12:



el resultado de la operación `11.splice(12, 'C', 'N', 'H')` será:



4. **(2 puntos)** A nuestro comerciante del ejercicio 2. le van tan bien los negocios desde que optimiza la carga de su camello, que ha decidido instalar en su almacén un sistema de gestión de los pedidos por internet. El sistema cuenta con una tabla en la que se almacenan todos los objetos junto con información sobre el precio, peso y número de unidades disponibles de cada uno de ellos. Ya se tienen implementadas operaciones para dar de alta nuevos objetos, y para modificar las características de un objeto existente.

```

typedef string Objeto;
typedef struct {
    float precio;
    float peso;
    int numUnidades;
} InfoObjeto;
class Pedidos {
    public: ... operaciones para dar de alta objetos y modificarlos ...
    private:
        HashMap<Objeto, InfoObjeto> objetos;
}

```

El sistema debe almacenar información sobre el pedido (lista de objetos) que debe entregarse cada día y la dirección de envío. Solo da tiempo a entregar un pedido al día. La clase `Pedidos` debe contar con las siguientes operaciones:

- **nuevoPedido:** Recibe una lista con los objetos que un cliente solicita, la fecha en la que desea recibir la mercancía y la dirección de envío. La operación debe construir una lista con los objetos existentes en el almacén que maximicen las ganancias del comerciante teniendo en cuenta que por ahora no cuenta nada mas que con su camello (función del ejercicio 2.), y encontrar la fecha en la que serán servidos. Ésta debe ser la fecha más próxima posterior a la solicitada que no tenga todavía asignada ningún pedido.
- **preparaPedidos:** Dadas dos fechas, devuelve una lista, ordenada por fecha, con información sobre los envíos que se realizarán entre las dos fechas dadas, ambas incluidas. Cada elemento de la lista contendrá la fecha de entrega, la lista de objetos a enviar y la dirección de envío.

Se puede hacer uso de un tipo `Fecha` que dispone de una operación para obtener la siguiente fecha a una dada. Se pide:

- **(0.5 puntos)** Diseñar una representación que permita implementar las operaciones pedidas de forma que el acceso a la información almacenada sea eficiente. Indicar qué operaciones adicionales ha de tener disponibles el tipo `Fecha` para poder usarlo en la representación propuesta.
- **(1.5 puntos)** Implementar la función `nuevoPedido` e indicar su coste.

5. (1 punto) Responde a las siguientes cuestiones (0.2 puntos cada una):

5.1) Dadas dos listas `List<int> l, t;` se quieren copiar los k primeros elementos de la lista `l` insertándolos en la lista `t`, de manera que queden en el mismo orden en que se encuentran en la lista `l`. Dado un iterador `ConstIterator it = l.cbegin()`, la operación correcta es:

- a) `for (;it != l.cend(); ++it) t.push_back(*it);`
- b) `for (;it != l.cend(); ++it) t.push_front(*it);`
- c) No se puede realizar utilizando este iterador
- d) Ninguna de las anteriores

5.2) Dada una operación de concatenación de listas con coste lineal en el número de nodos de la lista argumento, la siguiente implementación del recorrido en preorden de un árbol

```
List<T> Arbin<T>::preordenAux(Nodo *p) {  
    if (p == nullptr) return List<T>(); // Lista vacía  
    List<T> ret; ret.push_front(p->_elem);  
    ret.concat(preordenAux(p->_iz));  
    ret.concat(preordenAux(p->_dr));  
    return ret;  
}
```

tiene un coste en el caso peor (en función del número n de nodos del árbol) que está en:

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n \log n)$
- e) $O(n^2)$

5.3) En la pregunta anterior, si el árbol está balanceado, el coste del recorrido en preorden con la implementación dada será

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n \log n)$
- e) $O(n^2)$

5.4) La operación de añadir un nodo en un árbol de búsqueda tiene coste en el caso peor

1. logarítmico en el número de nodos del árbol
2. lineal en el número de nodos del árbol
3. logarítmico en la altura del árbol
4. lineal en la altura del árbol

Son correctas

- a) 1 y 3
- b) 2 y 3
- c) 1 y 4
- d) 2 y 4

5.5) Dada la secuencia de números **6, 13, 5, 7, 1, 20, 15**. Dibujar el árbol binario de búsqueda que se obtiene al insertar los elementos de la secuencia en ese orden. Partiendo de ese mismo árbol, dibujar también el árbol resultante tras borrar los elementos 6 y 7 en ese orden.

Estructuras de Datos y Algoritmos

Doble Grado, Ingeniería Informática, de Computadores y del Software

Examen parcial, 9 de Septiembre de 2016.

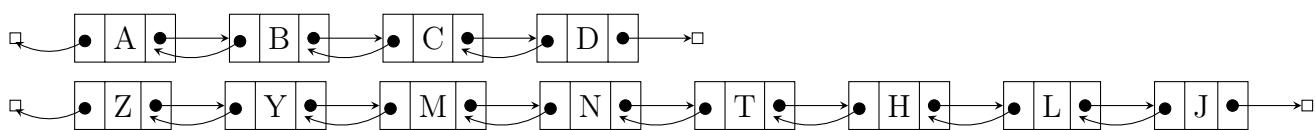
1. **(2.5 puntos)** Sea la clase `ListaEnlazadaDoble` que implementa listas doblemente enlazadas con punteros al primer y último nodo, no circulares y sin nodo cabecera:

```
template <class T>
class ListaEnlazadaDoble{
private:
    class Nodo {
public:
    Nodo() : _ant(NULL), _sig(NULL) {}
    Nodo(const T &elem) : _elem(elem), _sig(NULL), _ant(NULL) {}
    Nodo(Nodo* ant, const T &elem, Nodo *sig) : _elem(elem),
        _sig(sig), _ant(ant) {}
    T _elem;
    Nodo *_sig;    Nodo *_ant;
    };
    Nodo *_prim;    Nodo *_ult;
    ...
}
```

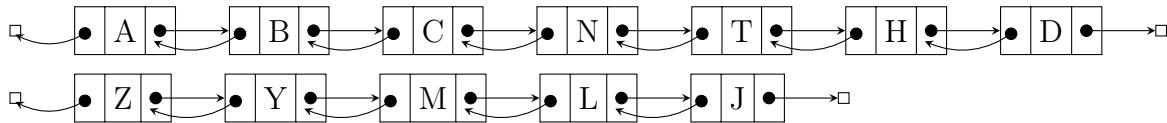
Se pide implementar el método `splice`, que recibe como parámetro una lista 1 y tres valores de tipo T, v1, v2 y v3. El método debe modificar la lista original añadiendo después del valor v1 los nodos de la lista 1 que se encuentran entre los valores v2 y v3 (ambos inclusive), y debe eliminarlos de la lista 1. Si el valor v1 no existe en la lista original, o si alguno de los valores v2, v3 no existe en la lista 1, o si v3 no se encuentra en una posición posterior a v2, las listas no se modifican. Se supone que todos los valores de las listas son diferentes.

La implementación debe ser lo mas eficiente posible. Para ello, se debe: evitar libera/reservar memoria y hacer copias de valores cuando no sea necesario, y recorrer las listas el menor número posible de veces. No se puede utilizar ninguna operación de la clase `ListaEnlazadaDoble` en la implementación.

Por ejemplo, dadas las siguientes listas de caracteres 11 y 12:



el resultado de la operación `l1.splice(l2, 'C', 'N', 'H')` será:



2. **(2 puntos)** Se quiere implementar un TAD para gestionar árboles generales, esto es, un árbol cuyos nodos tienen un número indeterminado de hijos. Se pide:

1. **(0.5 puntos)** Implementar una representación del TAD que almacene los hijos de un nodo en una lista.
2. **(1.5 puntos)** Implementar una operación `busca`, perteneciente a la clase, que reciba un elemento y devuelva cierto si el elemento está en el árbol y falso en caso contrario. Puede hacerse uso de funciones auxiliares privadas si se considera adecuado.

3. **(4.5 puntos)** Un comerciante ha decidido instalar en su almacén un sistema para recibir pedidos por internet y gestionarlos. El sistema cuenta con una tabla en la que se almacenan todos los objetos disponibles en el almacén junto con información sobre el precio, peso y número de unidades disponibles de cada uno de ellos. Ya se tienen implementadas operaciones para dar de alta nuevos objetos, y para modificar las características de un objeto existente.

```
typedef string Objeto;
typedef struct {
    float precio;
    float peso;
    int numUnidades;
} InfoObjeto;
class Pedidos {
public:    ... operaciones para dar de alta objetos y modificarlos ...
private:   HashMap<Objeto,InfoObjeto> objetos;
}
```

El sistema debe almacenar información sobre el pedido (lista de objetos) que debe entregarse cada día y la dirección de envío. Solo da tiempo a entregar un pedido al día. La clase **Pedidos** debe contar con las siguientes operaciones:

- **nuevoPedido:** Recibe una lista con los objetos que un cliente solicita, la fecha en la que desea recibir la mercancía y la dirección de envío. La operación debe almacenar el pedido asociándole la fecha en la que será entregado. Esta debe ser la fecha más próxima posterior a la solicitada que no tenga todavía asignada ningún pedido.
- **preparaPedidos:** Dadas dos fechas, devuelve una lista, ordenada por fecha, con información sobre los envíos que se realizarán entre las dos fechas dadas, ambas incluidas. Cada elemento de la lista contendrá la fecha de entrega, la lista de objetos a enviar y la dirección de envío.

Se puede hacer uso de un tipo **Fecha** que dispone de una operación para obtener la siguiente fecha a una dada. Se pide:

- **(0.5 puntos)** Diseñar una representación que permita implementar las operaciones pedidas de forma que el acceso a la información almacenada sea eficiente. Indicar qué operaciones adicionales ha de tener disponibles el tipo **Fecha** para poder usarlo en la representación propuesta.
- **(1.5 puntos)** Implementar la función **nuevoPedido** e indicar su coste.
- **(1 punto)** Nuestro comerciante ha descubierto que debido al sistema que tiene de asignación de fechas a los pedidos, está perdiendo buenos pedidos mientras sirve otros que no le reportan casi beneficio. Nos pide que modifiquemos la función **nuevoPedido** de forma que, al recibir un pedido, se le asigne la primera fecha posterior a aquella en que se desea recibir la mercancía, tal que no haya ningún pedido entre esta fecha y la asignada que tenga un coste menor que el del pedido que se está tratando. Si para ello es necesario modificar la fecha de algún pedido que ya estaba apalabrado no le importa, con tal de ir sirviendo aquellos que le proporcionan más beneficio.
- **(1.5 puntos)** Implementar la función **preparaPedidos**.

4. (1 punto) Responde a las siguientes cuestiones (0.2 puntos cada una):

4.1) Dadas dos listas `List<int> l, t;` se quieren copiar los k primeros elementos de la lista `l` insertándolos en la lista `t`, de manera que queden en el mismo orden en que se encuentran en la lista `l`. Dado un iterador `ConstIterator it = l.cbegin()`, la operación correcta es:

- a) `for (;it != l.cend(); ++it) t.push_back(*it);`
- b) `for (;it != l.cend(); ++it) t.push_front(*it);`
- c) No se puede realizar utilizando este iterador
- d) Ninguna de las anteriores

4.2) Dada una operación de concatenación de listas con coste lineal en el número de nodos de la lista argumento, la siguiente implementación del recorrido en preorden de un árbol

```
List<T> Arbin<T>::preordenAux(Nodo *p) {  
    if (p == nullptr) return List<T>(); // Lista vacía  
    List<T> ret; ret.push_front(p->_elem);  
    ret.concat(preordenAux(p->_iz));  
    ret.concat(preordenAux(p->_dr));  
    return ret;  
}
```

tiene un coste en el caso peor (en función del número n de nodos del árbol) que está en:

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n \log n)$
- e) $O(n^2)$

4.3) En la pregunta anterior, si el árbol está balanceado, el coste del recorrido en preorden con la implementación dada será

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n \log n)$
- e) $O(n^2)$

4.4) La operación de añadir un nodo en un árbol de búsqueda tiene coste en el caso peor

1. logarítmico en el número de nodos del árbol
2. lineal en el número de nodos del árbol
3. logarítmico en la altura del árbol
4. lineal en la altura del árbol

Son correctas

- a) 1 y 3
- b) 2 y 3
- c) 1 y 4
- d) 2 y 4

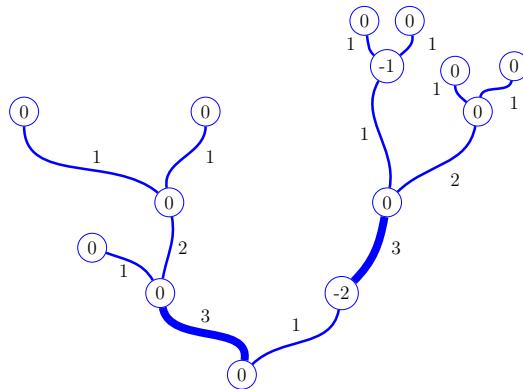
4.5) Dada la secuencia de números **6, 13, 5, 7, 1, 20, 15**. Dibujar el árbol binario de búsqueda que se obtiene al insertar los elementos de la secuencia en ese orden. Partiendo de ese mismo árbol, dibujar también el árbol resultante tras borrar los elementos 6 y 7 en ese orden.

Estructuras de Datos y Algoritmos

Doble Grado, Ingeniería Informática, de Computadores y del Software

Examen Final, 23 de Junio de 2016.

1. **(3 puntos)** Dados dos arrays v y w ordenados de manera estrictamente creciente, y dados dos naturales n y m tales que $n/2 \leq m \leq n$, se desea determinar si todos los elementos de w situados en las posiciones 0 a $m - 1$ están contenidos en las posiciones 0 a $n - 1$ de v . Se pide:
 1. Especificar el algoritmo.
 2. Implementar un algoritmo iterativo que resuelva el problema y escribir el invariante que cumple el bucle.
 3. Indicar y justificar adecuadamente el coste.
 2. **(2 puntos)** En un vector se tienen guardados $n \geq 2$ términos de una progresión aritmética de diferencia d dada desde el primero en adelante, pero uno de ellos falta. Se pide implementar un algoritmo eficiente que encuentre dicho término perdido. Por ejemplo, si el vector contiene los elementos 2, 4, 8, 10, 12, 14 y $d = 2$, el término que falta es el 6. Se considera que el primer y último elementos de la progresión nunca son los que se han perdido. Se recuerda que el término i -ésimo a_i de una progresión aritmética de diferencia d es $a_i = a_1 + (i - 1) * d$, siendo a_1 el primer término de la progresión. Se debe indicar y justificar el coste del algoritmo.
 3. **(2 puntos)** La compañía que gestiona la cuenca hidrográfica del río *Aguas Limpias* nos ha pedido que calculemos los tramos navegables de dicho río. Un tramo del río es navegable si contiene un caudal mayor o igual a $3 \text{ m}^3/\text{s}$. Como todos los ríos, el *Aguas Limpias* está formado por una serie de afluentes. Los afluentes que nacen de los manantiales llevan un caudal de un $1 \text{ m}^3/\text{s}$. Se considera que siempre confluyen exactamente dos afluentes. Cada vez que dos afluentes confluyen se calcula el caudal del tramo resultante como la suma de los caudales de sus afluentes. Adicionalmente, a lo largo del río se han construido varios embalses que hacen decrecer en una determinada cantidad el caudal del tramo saliente. En un embalse pueden confluir dos afluentes o solamente un tramo del río. La figura que se muestra a continuación es un ejemplo de un río que contiene 2 tramos navegables.



La estructura de afluentes se representa como un árbol binario cuyos nodos internos son o bien puntos de encuentro de dos afluentes, en cuyo caso el nodo contiene un 0, o bien un embalse, en cuyo caso el nodo contiene un número negativo que representa cuánto caudal puede absorber el embalse. Nótese que el caudal de un tramo no puede ser negativo pero sí 0. Implementa una función que dado un árbol que representa la estructura del río, calcule el número de tramos navegables.

4. **(2 puntos)** Los ingenieros de la empresa Apel están actualmente diseñando el software para su nuevo reproductor de música *iPud*, el cual debe permitir añadir y eliminar canciones del iPud, añadir una canción existente a la lista de reproducción, avanzar a la siguiente canción, obtener el tiempo total de la lista de reproducción, y obtener una lista con las canciones escuchadas recientemente. En particular, el TAD *IPud* debe contar con las siguientes operaciones:

- **create:** Crea un iPud vacío.
- **addSong(IP, S, A, D):** Añade la canción **S (string)** del artista **A (string)** con duración **D (int)** al iPud **IP**. Si ya existe una canción con el mismo nombre la operación dará error.
- **addToPlaylist(IP, S):** Añade la canción **S** al final de la lista de reproducción. Si la canción ya se encontraba en la lista entonces no se añade (es decir, la lista no tiene canciones repetidas). Si la canción no está en el iPud se devuelve error.
- **deleteSong(IP, S):** Elimina todo rastro de la canción **S** del iPud **IP**. Si la canción no existe la operación no tiene efecto.
- **play(IP):** La primera canción de la lista de reproducción abandona la lista de reproducción y se registra como reproducida. Si la lista es vacía la acción no tiene efecto.
- **current(IP):** Devuelve la primera canción de la lista de reproducción. Si es vacía se devuelve error.
- **totalTime(IP):** Obtiene la suma de las duraciones de las canciones que integran la lista de reproducción actual. Si es vacía se devuelve 0.
- **recent(IP, N):** Obtiene la lista con las **N** últimas canciones que se han reproducido (mediante la operación **play**), de la más reciente a la más antigua. Si el número de canciones reproducidas es menor que **N** se devolverán todas. La lista no tiene repeticiones, de manera que si una canción se ha reproducido más de una vez solo figurará la reproducción más reciente.

Se puede asumir que las canciones quedan únicamente determinadas por su nombre (**string**). Se ha elegido la siguiente representación para el TAD *IPud* usando los TADs vistos en clase:

```
typedef struct{
    string artist;
    int duration;
    bool inPlaylist; //indica si esta en la lista de reproduccion
    bool played; //indica si esta en la lista de reproducidas
} SongInfo;
class IPud{
public:
    ... las operaciones pedidas...
private:
    HashMap<string, SongInfo> songs;
    List<string> playlist; //la lista de reproduccion, sin repeticiones
    List<string> played; //las ya reproducidas, sin repeticiones
    int duration;
}
```

Se pide:

1. **(1 punto)** Indicar el coste de cada una de las operaciones con esta representación e implementar la operación **play**.
2. **(1 punto)** Modificar la representación anterior para que todas las operaciones tengan coste constante, excepto **recent** que no debe superar el coste lineal en **N**.

5. (1 punto) Responde a las siguientes cuestiones (0.2 puntos cada una):

- 5.1) Dada una lista enlazada simple con un puntero al primer nodo y un puntero al último, como la utilizada en la implementación de las colas dada en clase, indicar y justificar cuál de las siguientes operaciones no puede realizarse en tiempo constante.
- a) Añadir un elemento al comienzo de la lista
 - b) Borrar el elemento del comienzo de la lista
 - c) Añadir un elemento al final de la lista
 - d) Borrar el elemento del final de la lista
- 5.2) Sea un algoritmo de ordenación consistente en, dado un vector, insertar todos los elementos en un árbol binario de búsqueda, y posteriormente obtener en el vector original el recorrido en inorden del árbol. Se pide:
- a) Indicar el coste en tiempo y en espacio en el caso peor de dicho algoritmo.
 - b) Lo mismo pero suponiendo que se usa un árbol con lógica de re-equilibrado.
- 5.3) ¿Qué recorrido de un árbol binario de búsqueda de enteros se debe guardar para que pueda construirse posteriormente un árbol idéntico al inicial? Se debe tener en cuenta que sólo se guardarán los enteros contenidos en los nodos sin ninguna información adicional.
- a) El preorden
 - b) El inorden
 - c) El recorrido por niveles
 - d) El preorden o el recorrido por niveles
 - e) No es posible reconstruirlo igual que estaba con un solo recorrido
- 5.4) El coste de la operación insertar en una tabla hash es
1. en el caso peor lineal en el número de elementos de la tabla
 2. en el caso peor logarítmico en el número de elementos de la tabla
 3. en el caso promedio constante
 4. en el caso promedio logarítmico en el número de nodos de la tabla

Son correctas:

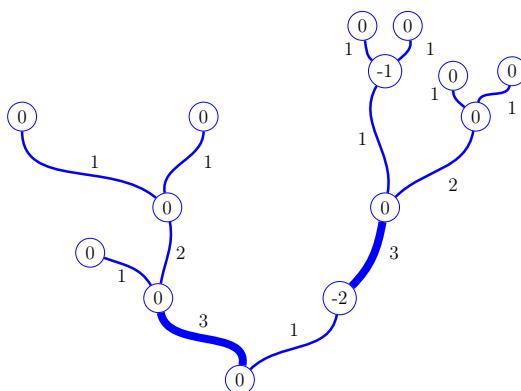
- a) 1 y 3
 - b) 1 y 4
 - c) 2 y 3
 - d) 2 y 4
- 5.5) Supongamos un `HashMap` con claves `int` y valores `char`, con tamaño inicial 5, cuya función hash se define como $hash(k) = 2 * k$, y que incluye equipamiento para duplicar su tamaño cuando se supera el 80 % de tasa de ocupación (como la implementación vista en clase). Inicialmente está vacío y se insertan los pares `<7,a>`, `<10,b>`, `<2,c>`, `<5,d>`, `<4,e>` en ese orden.
- a) Dibuja el estado final de la tabla. ¿Cuál es la tasa de ocupación?
 - b) ¿Qué problema tiene esta función hash sobre esta tabla?

Estructuras de Datos y Algoritmos

Doble Grado, Ingeniería Informática, de Computadores y del Software

Examen Parcial, 23 de Junio de 2016.

1. **(2 puntos)** La compañía que gestiona la cuenca hidrográfica del río *Aguas Limpias* nos ha pedido que calculemos los tramos navegables de dicho río. Un tramo del río es navegable si contiene un caudal mayor o igual a $3 \text{ m}^3/\text{s}$. Como todos los ríos, el *Aguas Limpias* está formado por una serie de afluentes. Los afluentes que nacen de los manantiales llevan un caudal de un $1 \text{ m}^3/\text{s}$. Se considera que siempre confluyen exactamente dos afluentes. Cada vez que dos afluentes confluyen se calcula el caudal del tramo resultante como la suma de los caudales de sus afluentes. Adicionalmente, a lo largo del río se han construido varios embalses que hacen decrecer en una determinada cantidad el caudal del tramo saliente. En un embalse pueden confluir dos afluentes o solamente un tramo del río. La figura que se muestra a continuación es un ejemplo de un río que contiene 2 tramos navegables.



La estructura de afluentes se representa como un árbol binario cuyos nodos internos son o bien puntos de encuentro de dos afluentes, en cuyo caso el nodo contiene un 0, o bien un embalse, en cuyo caso el nodo contiene un número negativo que representa cuánto caudal puede absorber el embalse. Nótese que el caudal de un tramo no puede ser negativo pero sí 0. Implementa una función que dado un árbol que representa la estructura del río, calcule el número de tramos navegables.

2. **(2,5 puntos)** Se tiene una lista doblemente enlazada con punteros al primer y último nodo, no circular, sin nodo cabecera, cuyos nodos tienen un campo `_info1` de tipo `int` y un campo `_info2` de tipo `T2`. La lista está parametrizada respecto al tipo `T2`:

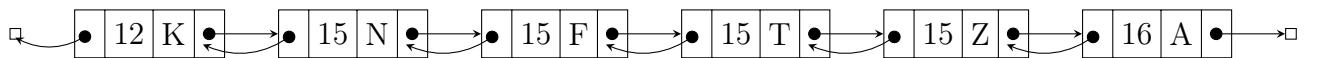
```
template <class T2>
class ListaEnlazadaDoble{
private:
    class Nodo {
public:
    Nodo() : _ant(NULL), _sig(NULL) {}
    Nodo(int n, const T2 &elem) : _info1(n), _info2(elem), _sig(NULL), _ant(NULL) {}
    Nodo(Nodo* ant, int n, const T2 &elem, Nodo *sig) : _info1(n), _info2(elem),
        _sig(sig), _ant(ant) {}
    int _info1;      T2 _info2;
    Nodo *_sig;     Nodo *_ant;
    };
    Nodo * _prim;    Nodo* _ult;
    ...
}
```

Los nodos de la lista se encuentran ordenados en orden creciente por el campo `_info1`, pudiendo existir muchos nodos con el mismo valor en dicho campo pero no nodos con el mismo valor en el campo `_info2`.

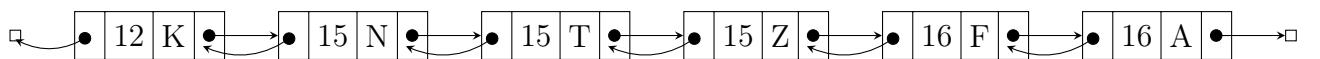
Se pide implementar una función `sumarUno` que reciba un puntero al comienzo de la lista enlazada y un valor de tipo `T2`, busque este valor en la lista y sume uno a su campo `_info1`. Si es necesario debe modificarse la lista para que siga estando ordenada por el campo `_info1`, situando el nodo a continuación de todos aquellos que tuviesen su mismo valor en el campo `_info1` antes de incrementar su valor. Si el valor dado no existe, la lista no se modifica.

La implementación debe ser lo mas eficiente posible. Para ello, se debe evitar liberar y reservar memoria, y hacer copias de los campos `_info1` e `_info2` cuando no sea necesario.

Por ejemplo, si la lista es:



el resultado de la operación `sumarUno(_prim, F)` será:



3. (4,5 puntos) Los ingenieros de la empresa Apel están actualmente diseñando el software para su nuevo reproductor de música *iPud*, el cual debe permitir añadir y eliminar canciones, añadir una canción existente a la lista de reproducción, avanzar a la siguiente canción, obtener el tiempo total de la lista de reproducción, y obtener una lista con las canciones escuchadas recientemente. En particular, el TAD *IPud* debe contar con las siguientes operaciones:

- **create**: Crea un *iPud* vacío.
- **addSong(IP, S, A, D)**: Añade la canción **S** (`string`) del artista **A** (`string`) con duración **D** (`int`) al *iPud* **IP**. Si ya existe una canción con el mismo nombre la operación dará error.
- **addToPlaylist(IP, S)**: Añade la canción **S** al final de la lista de reproducción. Si la canción ya se encontraba en la lista entonces no se añade (es decir, la lista no tiene canciones repetidas). Si la canción no está en el *iPud* se devuelve error.
- **deleteSong(IP, S)**: Elimina todo rastro de la canción **S** del *iPud* **IP**. Si la canción no existe la operación no tiene efecto.
- **play(IP)**: La primera canción de la lista de reproducción abandona la lista de reproducción y se registra como reproducida. Si la lista es vacía la acción no tiene efecto.
- **current(IP)**: Devuelve la primera canción de la lista de reproducción. Si la lista de reproducción es vacía se devuelve error.
- **totalTime(IP)**: Obtiene la suma de las duraciones de las canciones que integran la lista de reproducción actual. Si es vacía se devuelve 0.
- **recent(IP, N)**: Obtiene la lista con las **N** últimas canciones que se han reproducido (mediante la operación `play`), de la más reciente a la más antigua. Si el número de canciones reproducidas es menor que **N** se devolverán todas. La lista no tiene repeticiones, de manera que si una canción se ha reproducido más de una vez solo figurará la reproducción más reciente.

Se puede asumir que las canciones quedan únicamente determinadas por su nombre (`string`).

Se ha elegido la siguiente representación para el TAD IPud usando los TADs vistos en clase:

```
typedef struct{
    string artist;
    int duration;
    bool inPlaylist; //indica si esta en la lista
    bool played; //indica si esta en la lista de reproducidas
} SongInfo;
class iPud{
public:
    ... las operaciones pedidas...
private:
    HashMap<string, SongInfo> songs;
    List<string> playlist; //la lista de reproducción, sin repeticiones
    List<string> played; //las ya reproducidas, sin repeticiones
    int duration;
}
```

Se pide:

1. **(1 punto)** Indicar el coste de cada una de las operaciones con esta representación e implementar la operación `play`.
2. **(1 punto)** Modificar la representación anterior para que todas las operaciones tengan coste constante, excepto `recent` que no debe superar el coste lineal en N .
3. **(2,5 puntos)** Supongamos que se desea extender la funcionalidad del IPud para permitir manejar múltiples listas de reproducción, identificadas mediante un nombre único. En particular se dispondría de las operaciones:
 - `saveCurrentList`, que guarda la lista de reproducción actual con el nombre proporcionado;
 - `generateArtistList`, que genera una nueva lista de reproducción con las canciones de un artista (si es que hay alguna), y la almacena con el propio nombre del artista;
 - `setPlaylist` que pone como lista actual la lista proporcionada.
 - `allList` que devuelve una lista con los nombres de las listas de reproducción almacenadas.

Rediseña la representación del TAD de manera que se minimice el coste de las nuevas operaciones. Indica y justifica el coste esperado de las nuevas operaciones y de las antiguas que se hayan visto modificadas.

Implementa con dicha representación la operación `generateArtistList`.

4. (1 punto) Responde a las siguientes cuestiones (0.2 puntos cada una):

- 4.1) Dada una lista enlazada simple con un puntero al primer nodo y un puntero al último, como la utilizada en la implementación de las colas dada en clase, indicar y justificar cuál de las siguientes operaciones no puede realizarse en tiempo constante.
- Añadir un elemento al comienzo de la lista
 - Borrar el elemento del comienzo de la lista
 - Añadir un elemento al final de la lista
 - Borrar el elemento del final de la lista
- 4.2) Sea un algoritmo de ordenación consistente en, dado un vector, insertar todos los elementos en un árbol binario de búsqueda, y posteriormente obtener en el vector original el recorrido en inorden del árbol. Se pide:
- Indicar el coste en tiempo y en espacio en el caso peor de dicho algoritmo.
 - Lo mismo pero suponiendo que se usa un árbol con lógica de re-equilibrado.
- 4.3) ¿Qué recorrido de un árbol binario de búsqueda de enteros se debe guardar para que pueda construirse posteriormente un árbol idéntico al inicial? Se debe tener en cuenta que sólo se guardarán los enteros contenidos en los nodos sin ninguna información adicional.
- El preorden
 - El inorden
 - El recorrido por niveles
 - El preorden o el recorrido por niveles
 - No es posible reconstruirlo igual que estaba con un solo recorrido
- 4.4) El coste de la operación insertar en una tabla hash es
- en el caso peor lineal en el número de elementos de la tabla
 - en el caso peor logarítmico en el número de elementos de la tabla
 - en el caso promedio constante
 - en el caso promedio logarítmico en el número de nodos de la tabla
- Son correctas:
- 1 y 3
 - 1 y 4
 - 2 y 3
 - 2 y 4
- 4.5) Supongamos un `HashMap` con claves `int` y valores `char`, con tamaño inicial 5, cuya función hash se define como $hash(k) = 2 * k$, y que incluye equipamiento para duplicar su tamaño cuando se supera el 80% de tasa de ocupación (como la implementación vista en clase). Inicialmente está vacío y se insertan los pares `<7,a>`, `<10,b>`, `<2,c>`, `<5,d>`, `<4,e>` en ese orden.
- Dibuja el estado final de la tabla. ¿Cuál es la tasa de ocupación?
 - ¿Qué problema tiene esta función hash sobre esta tabla?

Estructuras de Datos y Algoritmos

Doble Grado, Ingeniería Informática, de Computadores y del Software

Examen Primer Cuatrimestre, 11 de Febrero de 2016.

1. (3 puntos) Diseñar y verificar, o bien derivar, un algoritmo iterativo eficiente que cumpla la siguiente especificación:

$$\{0 \leq n \leq longitud(v) \wedge \forall k : 0 \leq k < n : v[k] \geq 0\}$$

```
fun parImpar(int v[], int n) return int p
    {p = # i, j : 0 \leq i < j < n : v[i] \% 2 = 0 \wedge v[j] \% 2 = 1}
```

Indicar y justificar la complejidad del algoritmo.

2. (3,5 puntos) El ISBN de un libro consta actualmente de 13 dígitos $d_1 d_2 \dots d_{13}$. Al dígito d_{13} se le denomina dígito de control y se calcula a partir de los anteriores de la siguiente forma: primero se calcula la suma de multiplicar cada dígito de una posición par por 3 y cada dígito de una posición impar por 1. Los dígitos se numeran desde el 1 empezando por el más significativo:

$$d_1 * 1 + d_2 * 3 + d_3 * 1 + d_4 * 3 + d_5 * 1 + d_6 * 3 + d_7 * 1 + d_8 * 3 + d_9 * 1 + d_{10} * 3 + d_{11} * 1 + d_{12} * 3$$

El dígito d_{13} será aquel que sumado con el resultado de dicha suma de un múltiplo de 10. Por ejemplo, el dígito de control del número 336772900480 es 9 porque la suma anterior da como resultado 81 y $81 + 9 = 90$, que es múltiplo de 10.

Diseñar un algoritmo recursivo eficiente que ayude a calcular el dígito de control de un ISBN con cualquier número de dígitos. Igual que en el ejemplo, los dígitos anteriores vienen dados como un número entero. Por ejemplo, si el número dado es 2561, la suma será $2*1+5*3+6*1+1*3 = 26$ y por tanto el resultado sería 4. Indicar y justificar la complejidad del algoritmo.

3. (3,5 puntos) Consideremos una matriz, M , de números reales de dimensiones $N \times N$. Supongamos que esta matriz nos sirve para cartografiar cierto terreno que se ha dividido en forma de cuadrícula, de forma que $M[i][j]$ es la altura media de la casilla (i, j) .

Se desea construir una carretera entre dos puntos dados. Para que una carretera pueda unir dos casillas adyacentes (no en diagonal) es necesario que la diferencia entre sus alturas medias no supere cierto valor, h_{max} , es decir:

$$| M[a][b] - M[c][d] | \leq h_{max}$$

donde las casillas (a, b) y (c, d) son adyacentes.

Dada una matriz M , una altura máxima h_{max} y dos casillas del terreno distintas entre sí, diseña un algoritmo de vuelta atrás que calcule la carretera de menor longitud (si es que existe) que resuelva el problema. Se considera que la longitud de la carretera es el número de casillas por las que pasa, incluyendo tanto el origen como el destino.

Por ejemplo, si la matriz M de alturas es la siguiente y $h_{max} = 2$:

$$\begin{pmatrix} 0 & 3 & 4 \\ 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}$$

la carretera de menor longitud para ir desde la casilla $(0, 1)$ hasta la $(0, 0)$ es de longitud 4 y pasa por las casillas $(0, 1)$, $(1, 1)$, $(1, 0)$, $(0, 0)$. Hay otras carreteras válidas pero no óptimas, como por ejemplo la que recorre las casillas $(0, 1)$, $(1, 1)$, $(2, 1)$, $(2, 0)$, $(1, 0)$, $(0, 0)$, que es de longitud 6. Una carretera no válida sería la que va directamente de $(0, 1)$ a $(0, 0)$ al contener una diferencia de alturas entre casillas adyacentes superior al máximo permitido.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Final, Septiembre 2015

- 1.[3 ptos]** Los **números de Jacobsthal** son una sucesión de números enteros definida de la siguiente manera:

$$\begin{aligned} J(0) &= 0 \\ J(1) &= 1 \\ J(n) &= J(n-1) + 2 * J(n-2) \text{ si } n > 1 \end{aligned}$$

Dada la siguiente especificación de un algoritmo que calcula el n -ésimo número de Jacobsthal:

```
{n ≥ 0}  
fun Jacobsthal(int n) return int x  
{x = J(n)}
```

donde la función $J : \mathbb{N} \rightarrow \mathbb{N}$ es la definida anteriormente, diseñar y verificar (o derivar) un algoritmo iterativo que dado un natural $n \geq 0$ calcule $J(n)$ en tiempo en $O(n)$. Justificar adecuadamente el coste del algoritmo.

- 2. [3 ptos]** En una matriz de N filas y M columnas hay un rectángulo desde una posición desconocida (*fila, columna*) ($0 \leq \text{fila} < N, 0 \leq \text{columna} < M$) hasta la posición $(N-1, M-1)$ que contiene el número 1 en todas sus posiciones. El resto de posiciones de la matriz contienen el número 0. Se pide implementar un algoritmo eficiente que encuentre la posición (*fila, columna*). Justificar adecuadamente el coste del algoritmo.

Por ejemplo, si $N = 5, M = 4$, y la matriz de entrada es:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

el algoritmo debe devolver $(3, 1)$.

- 3. [2,5 ptos]** Implementa la siguiente función:

```
template <typename T>  
bool coinciden(const Arbin<T> &a, const List<T> &pre);
```

que diga si la lista `pre` coincide con el recorrido en preorden del árbol `a`. No puedes hacer uso de estructuras de datos auxiliares (arrays, pilas, colas, listas, árboles etc.) ni pedir/liberar memoria adicional (hacer `new` o `delete`).

4. [1,5 ptos] Implementa la siguiente función:

```
template <typename T>
void invierteBase(Stack<T> &p, int n);
```

que reciba la pila p y un valor entero n ($0 \leq n \leq p.size()$) y modifique p , de forma que los n valores de la cima queden en el orden que están y los $p.size() - n$ valores del fondo de la pila queden en orden inverso al de entrada.

En la implementación de la función deben utilizarse únicamente TADs vistos en clase, se valorará que se seleccionen los TADs más apropiados para la resolución del problema. No se pueden utilizar los arrays de C++. Se valorará la eficiencia de la solución propuesta.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Segundo Parcial, Septiembre 2015

1. [2 ptos] Extiende la implementación de las listas añadiendo una operación `invierte()` que invierta la lista de forma que el primer elemento pase a ser el último, el segundo el penúltimo y así sucesivamente. No debes utilizar memoria auxiliar (está permitido usar variables locales de tipo puntero siempre y cuando no se hagan `new's`, pero por ejemplo no pueden usarse variables de tipo T).

No se admitirán soluciones recursivas.

A continuación se muestra a modo de recordatorio las partes relevantes del TAD List.

```
template <typename T>
class List {
    private:
        class Nodo {
            public:
                Nodo() : _sig(NULL), _ant(NULL) {}
                Nodo(const T &elem) : _elem(elem), _sig(NULL), _ant(NULL) {}
                Nodo(Nodo *ant, const T &elem, Nodo *sig) :
                    _elem(elem), _sig(sig), _ant(ant) {}

                T _elem;
                Nodo *_sig;
                Nodo *_ant;
            };
            Nodo * _prim;    Nodo* _ult;    ...
        public:
            void invierte(); ...
    }
}
```

2. [2,5 ptos] Implementa la siguiente función:

```
template <typename T>
bool coinciden(const Arbin<T> &a, const List<T> &pre);
```

que diga si la lista `pre` coincide con el recorrido en preorden del árbol `a`. No puedes hacer uso de estructuras de datos auxiliares (arrays, pilas, colas, listas, árboles etc.) ni pedir/liberar memoria adicional (hacer `new` o `delete`).

3. [1,5 ptos] Implementa la siguiente función:

```
template <typename T>
void invierteBase(Stack<T> &p, int n);
```

que reciba la pila p y un valor entero n ($0 \leq n \leq p.size()$) y modifique p , de forma que los n valores de la cima queden en el orden que están y los $p.size() - n$ valores del fondo de la pila queden en orden inverso al de entrada.

En la implementación de la función deben utilizarse únicamente TADs vistos en clase, se valorará que se seleccionen los TADs más apropiados para la resolución del problema. No se pueden utilizar los arrays de C++. Se valorará la eficiencia de la solución propuesta.

4. [4 ptos] El periódico local quiere poner orden en su base de datos de noticias. Buscan poder relacionar rápidamente las noticias que se quieren publicar con noticias ya publicadas anteriormente para dar una información más completa a los lectores. Para ello quieren implementar un TAD que les permita manejar las noticias publicadas, bien obteniendo las últimas noticias que se han publicado sobre un tema o bien, refinando un poco más, obtener las últimas noticias sobre un tema que además contengan una cierta palabra. Como sólo son interesantes algunas palabras sobre cada tema, los periodistas pueden añadir al sistema las palabras de cada tema que consideren importantes. Las palabras interesantes pueden añadirse en cualquier momento y borrarse cuando ya no sean interesantes.

Las operaciones mínimas que se requieren son:

- *nuevaNoticia*: Añade una noticia sobre un cierto tema. Si el tema no existe en el sistema, se añade. Se garantiza que las noticias se añaden en el orden en que se producen, es decir las llamadas a la operación *nuevaNoticia* se realizarán por orden de publicación de las noticias.
- *ultimasNoticias*: Obtiene las n últimas noticias dadas de alta en el sistema sobre un cierto tema. La primera noticia de la lista debe ser la última que se ha producido. Si hay menos de n noticias sobre este tema en el sistema, devuelve todas ellas.
- *noticiasTermino*: Obtiene las últimas n noticias de un tema en que aparece una cierta palabra. La primera noticia de la lista debe ser la última que se ha producido. Si hay menos de n noticias, devuelve todas ellas.
- *nuevoTerminoBusqueda*: Crea un nuevo término de búsqueda para un cierto tema. Si el término ya existe la operación no tiene efecto.
- *eliminaTerminoBusqueda*: Elimina un término de búsqueda, pero no las noticias en las que aparece el término. Si el término no estaba dado de alta la operación no tiene efecto.
- *listarTerminos*: Obtiene una lista ordenada alfabéticamente con todos los términos definidos para un cierto tema.

Se supone que existe una clase *Noticia* que nos da acceso al contenido de una cierta noticia. Esta clase cuenta con un operación `bool esta (const string & p) const;` que dada una palabra nos dice si pertenece a la noticia.

Desarrolla en C++ una implementación de la clase *NoticiasPublicadas* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. Indica qué le exiges a los tipos que almacenan noticias, términos y temas. Determina de forma razonada cuál es la complejidad de cada operación.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Final, Junio 2015

- 1.[1 pto]** Dado un vector de n elementos ($n \geq 0$), se desea obtener la suma de todos los productos de valores situados en parejas de posiciones distintas con una complejidad $\mathcal{O}(n)$. La especificación del algoritmo es la siguiente:

```
{0 ≤ n < longitud(v)}  
fun sumaProductos (int v[], int n) return int p  
{p = ∑ i, j : 0 ≤ i < j < n : v[i] * v[j]}
```

Se han escrito los dos siguientes algoritmos para resolver el problema:

```
(a) k = n; p = 0; s = 0;  
    while (k>0)  
        { p = p + v[k-1] * s;  
          s = s + v[k-1];  
          k = k - 1;  
        };  
    return p;  
  
(b) k = 0; p = 0; s = 0;  
    while (k<n)  
        { p = p + v[k] * s;  
          s = s + v[k];  
          k = k + 1;  
        };  
    return p;
```

Escribir los invariantes de los bucles que permiten demostrar la corrección de cada uno de ellos.

- 2. [3 ptos]** De n actividades conocemos sus instantes de comienzo c_i y finalización f_i ; es decir, el intervalo de realización de la actividad i ($0 \leq i < n$) es $[c_i, f_i]$. Diseñar un algoritmo de vuelta atrás que imprima todos los subconjuntos con r o más actividades que no solapen entre sí (es decir, cuya intersección dos a dos es vacía).

Se puede suponer que los intervalos de las actividades vienen dados en un vector a de parejas de enteros (los instantes de comienzo y finalización) ordenado crecientemente por instante de comienzo.

- 3. [3 ptos]** Podemos utilizar los árboles binarios para representar los caminos en la falda de una montaña. La raíz del árbol representa la cima de la que salen una o dos rutas. Las distintas rutas según se va ensanchando la falda de la montaña se dividen en dos formando caminos que nunca se volverán a conectar. Un escalador está en la cima de la montaña (raíz del árbol) y se da cuenta de que en distintas intersecciones (marcadas en el árbol con 'X') hay amigos que necesitan su ayuda para subir. Tiene que bajar a cada una de las 'X' y ayudarles a subir de uno en uno. Implementa una función con la cabecera

```
int tiempoAyuda(const Arbin<char> &a);
```

que, dado uno de estos árboles binarios, devuelva el tiempo que tardará el escalador en ayudar a todos esos amigos si cada tramo de intersección a intersección lleva una hora en ser recorrido (en cada uno de los dos sentidos).

- 4. [3 ptos]** Cada uno de los profesores de distintas asignaturas de un mismo grupo de alumnos tiene una lista de faltas, es decir, para cada alumno van anotando cuantas veces ha faltado en esa asignatura (0 si no ha faltado nunca). Para gestionar esta información diseñan un tipo abstracto de datos *Faltas* con tres operaciones generadoras:

- *anadirAlumno*, que añade un alumno en todas las asignaturas con 0 faltas.
- *anadirFalta*, que incrementa en 1 el número de faltas de un alumno en una asignatura.
- *anadirAsignatura*, que construye una lista con los mismos alumnos de las demás asignaturas, cada uno de ellos con 0 faltas.

A la hora de implementar este TAD han decidido que la lista de faltas de una asignatura viene representada por un diccionario con clave *IdAlumno* y valor asociado el número de faltas del alumno en esa asignatura; y que todas las listas de faltas se hallan almacenadas en un diccionario con clave *IdAsignatura* (identificador de la asignatura) y valor asociado la lista de faltas de esa asignatura. El invariante de la representación incluye el hecho de que las listas de todas las asignaturas contienen exactamente los mismos alumnos:

```
class Faltas
{public :
    void anadirAlumno(const IdAlumno& a);
        //incorpora al alumno en todas las asignaturas que haya con 0 faltas
    void anadirFalta(const IdAlumno& a,const IdAsignatura& s);
        //incrementa en 1 las faltas del alumno en la asignatura
    void anadirAsignatura(const IdAsignatura& s);
        // incorpora todos los alumnos que ya esten presentes
        // en las otras asignaturas con 0 faltas

    ...otros metodos...
private:
    Diccionario<IdAsignatura,Diccionario<IdAlumno,int> > listas_faltas;}
```

En la reunión de fin de curso, ponen en común su información y desean añadir a este TAD las siguientes operaciones:

- *noFaltas*, que por orden alfabético (el dato sobre *IdAlumno*) devuelve una lista con todos los alumnos que no han faltado a ninguna clase en ninguna de las asignaturas.
- *totalFaltas*, que dado un alumno, devuelve el número de faltas que acumula entre todas las asignaturas.

Se pide:

- 1.[0,5 ptos]** Elegir justificadamente la implementación de cada uno de los dos diccionarios e indicar, en base a esta decisión, qué le exige a los tipos *IdAsignatura* e *IdAlumno*, y cuál es el coste que tendrían las operaciones generadoras arriba mencionadas.
- 2.[0,75 ptos]** Implementar la operación *totalFaltas* e indicar su coste.
- 3.[1,25 ptos]** Implementar la operación *noFaltas* e indicar su coste.
- 4.[0,5 ptos]** Explicar cómo mejorar la representación del tipo *Faltas* para que las dos operaciones anteriores, *noFaltas* y *totalFaltas*, sean más eficientes sin que se incremente el coste de las generadoras.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Segundo Parcial, Junio 2015

- 1. [3 puntos]** Extiende el TAD List visto en clase, con una nueva operación que modifique la lista intercalando sus nodos de la siguiente forma: supongamos que los nodos de la lista de izquierda a derecha son $n_1, n_2, n_3, n_4, \dots, n_{k-3}, n_{k-2}, n_{k-1}, n_k$, al finalizar la ejecución del método los nodos estarán colocados como $n_1, n_k, n_2, n_{k-1}, n_3, n_{k-2}, n_4, n_{k-3}, n_5 \dots$

Indica la complejidad de tu implementación. Esta debe ser lo más eficiente posible. Para ello, se debe evitar liberar y reservar memoria, y hacer copias de los campos. Si haces uso de métodos auxiliares, impleméntalos también.

A continuación se muestra a modo de recordatorio las partes relevantes del TAD List.

```
class List {
    private:
        class Nodo {
            public:
                Nodo() : _sig(NULL), _ant(NULL) {}
                Nodo(const T &elem) : _elem(elem), _sig(NULL), _ant(NULL) {}
                Nodo(Nodo *ant, const T &elem, Nodo *sig) :
                    _elem(elem), _sig(sig), _ant(ant) {}

                T _elem;
                Nodo *_sig;
                Nodo *_ant;
            };
            Nodo * _prim;    Nodo* _ult;    ...
        public:
            void doblarLista();    ...
    }
}
```

- 2. [3 puntos]** Podemos utilizar los árboles binarios para representar los caminos en la falda de una montaña. La raíz del árbol representa la cima de la que salen una o dos rutas. Las distintas rutas según se va ensanchando la falda de la montaña se dividen en dos formando caminos que nunca se volverán a conectar. Un escalador está en la cima de la montaña (raíz del árbol) y se da cuenta de que en distintas intersecciones (marcadas en el árbol con 'X') hay amigos que necesitan su ayuda para subir. Tiene que bajar a cada una de las 'X' y ayudarles a subir de uno en uno. Implementa una función con la cabecera

```
int tiempoAyuda(const Arbin<char> &a);
```

que, dado uno de estos árboles binarios, devuelva el tiempo que tardará el escalador en ayudar a todos esos amigos si cada tramo de intersección a intersección lleva una hora en ser recorrido (en cada uno de los dos sentidos).

- 3. [4 puntos]** Cada uno de los profesores de distintas asignaturas de un mismo grupo de alumnos tiene una lista de faltas, es decir, para cada alumno van anotando cuantas veces ha faltado en esa asignatura (0 si no ha faltado nunca). Para gestionar esta información diseñan un tipo abstracto de datos *Faltas* con tres operaciones generadoras:

- *anadirAlumno*, que añade un alumno en todas las asignaturas con 0 faltas.
- *anadirFalta*, que incrementa en 1 el número de faltas de un alumno en una asignatura.
- *anadirAsignatura*, que construye una lista con los mismos alumnos de las demás asignaturas, cada uno de ellos con 0 faltas.

A la hora de implementar este TAD han decidido que la lista de faltas de una asignatura viene representada por un diccionario con clave *IdAlumno* y valor asociado el número de faltas del alumno en esa asignatura; y que todas las listas de faltas se hallan almacenadas en un diccionario con clave *IdAsignatura* (identificador de la asignatura) y valor asociado la lista de faltas de esa asignatura. El invariante de la representación incluye el hecho de que las listas de todas las asignaturas contienen exactamente los mismos alumnos:

```

class Faltas
{public :
    void anadirAlumno(const IdAlumno& a);
        //incorpora al alumno en todas las asignaturas que haya con 0 faltas
    void anadirFalta(const IdAlumno& a,const IdAsignatura& s);
        //incrementa en 1 las faltas del alumno en la asignatura.
    void anadirAsignatura(const IdAsignatura& s);
        // incorpora todos los alumnos que ya esten presentes
        // en las otras asignaturas con 0 faltas

    ...otros metodos...
private:
    Diccionario<IdAsignatura,Diccionario<IdAlumno,int> > listas_faltas;}
```

En la reunión de fin de curso, ponen en común su información y desean añadir a este TAD las siguientes operaciones:

- *noFaltas*, que por orden alfabético (el dato sobre *IdAlumno*) devuelve una lista con todos los alumnos que no han faltado a ninguna clase en ninguna de las asignaturas.
- *totalFaltas*, que dado un alumno, devuelve el número de faltas que acumula entre todas las asignaturas.
- *maxFaltas*, que devuelve la asignatura donde mayor número de faltas hay entre todos los alumnos; si hay varias con el máximo número de faltas devuelve una cualquiera.

Se pide:

- 1.[0,5 ptos]** Elegir justificadamente la implementación de cada uno de los dos diccionarios e indicar, en base a esta decisión, qué le exige a los tipos *IdAsignatura* e *IdAlumno*, y cuál es el coste que tendrían las operaciones generadoras arriba mencionadas.
- 2.[0,75 puntos]** Implementar la operación *totalFaltas* e indicar su coste.
- 3.[1,25 puntos]** Implementar la operación *noFaltas* e indicar su coste.
- 4.[0,5 puntos]** Explicar cómo mejorar la representación del tipo *Faltas* para que las dos operaciones anteriores, *noFaltas* y *totalFaltas*, sean más eficientes sin que se incremente el coste de las generadoras.
- 5.[1 punto]** Extender la representación del tipo *Faltas* para que la operación *maxFaltas* sea de coste $O(m)$ siendo m el número de asignaturas. Con todas las extensiones propuestas en este apartado y el anterior implementar la generadora *anadirFalta*.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Primer Cuatrimestre, 12 de Febrero de 2015.

1. (3,5 puntos) Dado un vector de n elementos ($n \geq 0$), se desea obtener la suma de todos los productos de valores situados en parejas de posiciones distintas (con una complejidad $\mathcal{O}(n)$).

Ejemplo: para el array con contenido: [1, 3, 5, 7] se debe devolver $1*3+1*5+1*7+3*5+3*7+5*7$.

Para el array con contenido [6, 2, 5, 9, 1, 2] se debe devolver:

$$6*2 + 6*5 + 6*9 + 6*1 + 6*2 + 2*5 + 2*9 + 2*1 + 2*2 + 5*9 + 5*1 + 5*2 + 9*1 + 9*2 + 1*2$$

Se pide:

1. Especificación del algoritmo.
2. Diseño y verificación (o derivación) e implementación del algoritmo iterativo.
3. Justificación de la complejidad.

2. (3 puntos) La *imagen especular* de un número natural es el número que resulta al invertir el orden de sus dígitos. Por ejemplo, la *imagen especular* de 13492 es 29431 y la *imagen especular* del 1000 es el 1. Implementa dos algoritmos recursivos, uno final y otro no final, que calculen la *imagen especular* de un número natural representado como `unsigned int`. Indicar la llamada inicial a cada algoritmo con los valores iniciales de cada parámetro. Justifica el coste de cada algoritmo.

NOTA: Ten en cuenta que no se pide la especificación, ni la derivación / verificación formal de los algoritmos.

3. (3,5 puntos) Se dispone de una lista de n productos alimenticios entre los que se quiere escoger algunos para seguir una dieta adecuada. Para cada producto i ($0 \leq i < n$) se conoce su precio $p_i \geq 0$, su contenido en proteínas $q_i \geq 0$ y su cantidad de calorías $c_i \geq 0$. Se desea seleccionar algunos de estos productos (a lo sumo uno de cada) de forma que el precio total no supere un presupuesto M , el contenido proteico total sea al menos de Q y el valor calórico sea lo menor posible. Diseñar un algoritmo de vuelta atrás para encontrar la selección óptima, es decir, la que minimiza la cantidad de calorías. Se valorarán las podas realizadas.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Final, 9 de Septiembre de 2014.

- 1. (3 puntos)** Consideremos un vector $V[N]$ con $0 < N$ de números enteros, cuyos valores se han obtenido aplicando una rotación sobre un vector ordenado en orden estrictamente decreciente. Implementa un algoritmo que calcule el mínimo del vector con una complejidad $\mathcal{O}(\log n)$. El número de elementos sobre los que se aplica la rotación para obtener el vector de entrada cumple $0 \leq \text{elementos rotados} < N$ y no se conoce.

Ejemplo: un posible vector de entrada sería el vector 70 55 13 4 100 80 obtenido desplazando los dos primeros elementos del vector 100 80 70 55 13 4 al final del mismo.

- 2. (3 puntos)** Dada la siguiente serie

$$x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \dots$$

donde x es un número real tal que $|x| < 1$, y dado un entero $n \geq 0$, especifica y verifica o especifica y deriva, un algoritmo iterativo de coste lineal para calcular la suma de los n primeros términos de la serie. No se puede considerar que la operación de cálculo de una potencia se realiza en tiempo constante. Justificar el coste de la función implementada.

- 3. (4 puntos)** Estás trabajando en un nuevo videojuego llamado *CiudadMatic*: un simulador de ciudades, llenas de edificios de distinto tipo. Será posible construir y reparar estos edificios gastando dinero, y al final de cada turno (después de haber construido y reparado tantos edificios como se quiera), recaudar los impuestos que generen. Necesitarás implementar las siguientes operaciones:

- *CiudadMatic*: inicializa una nueva ciudad vacía, con el dinero que se pase como argumento disponible para ser gastado.
- *nuevoTipo*: añade un nuevo tipo de edificio al sistema, con un identificador proporcionado por el usuario (por ejemplo, “bar”), un coste de construcción, una cantidad de impuestos generada por turno, y una calidad de base (máximo de turnos sin reparar). No devuelve nada.
- *insertaEdificio*: dado el nombre de un edificio (por ejemplo, “El Bar de Moe”), y el identificador de su tipo, y asumiendo que se disponga del dinero necesario para construirlo, añade el edificio a la ciudad y resta su coste de construcción del dinero disponible. No devuelve nada.
- *reparaEdificio*: repara el edificio cuyo identificador se pase como argumento a “como recién construido”, a un coste del 10 % del coste de construcción (descartando los decimales), independientemente de lo estropeado que estuviese. No devuelve nada.
- *finTurno*: todos los edificios construidos generan los impuestos que les corresponden por su tipo. Después, todos se estropean por un punto de calidad, y aquellos que lleguen a 0 son derribados y eliminados de la ciudad. Devuelve el dinero total disponible para el nuevo turno (impuestos generados + dinero no gastado del turno anterior).

- *listaEdificios*: dado un identificador de tipo de edificio, devuelve una lista con los edificios de ese tipo que están actualmente construidos, por orden de antigüedad (primero el más viejo).

Desarrolla en C++ una implementación de la clase *CiudadMatic* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. En cada operación, indica también, de forma razonada, su complejidad.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Segundo Cuatrimestre, 9 de Septiembre de 2014.

- 1. (3 puntos)** Dado un árbol binario de enteros, se entiende que es un árbol genealógico correcto si cumple las siguientes reglas, producto de interpretar el entero de cada nodo como el *año de nacimiento* del individuo, el hijo izquierdo como su primer hijo de un máximo de dos, y el hijo derecho como el segundo hijo:

- La edad del padre siempre supera en al menos 18 años las edades de cada uno de los hijos.
- La edad del segundo hijo (si existe) es al menos dos años menos que la del primer hijo (no hay hermanos gemelos/mellizos en estos árboles).
- Los árboles genealógicos de ambos hijos son también correctos.

Implementa una función que reciba un árbol binario que permita averiguar si un árbol binario es o no árbol genealógico correcto y, en caso de serlo, el número de generaciones distintas que hay en la familia.

- 2. (3 puntos)** Extiende la implementación de los árboles de búsqueda, añadiendo la siguiente operación:

```
Iterador busca(const Clave &clave) const;
```

que busque en el árbol la clave dada y devuelva un *iterador* que permita recorrer todos los elementos contenidos en el árbol desde esa clave.

A modo de recordatorio, la definición (parcial) de la clase interna `Arbus::Iterador` vista en clase es:

```
class Iterador {
public:
    // ...
protected:
    // Para que pueda construir objetos del
    // tipo iterador
    friend class Arbus;
    // ...
    // Puntero al nodo actual del recorrido
    // NULL si hemos llegado al final.
    Nodo *_act;
```

```

// Ascendientes del nodo actual
// aún por visitar
Pila<Nodo*> _ascendientes;
};

```

- 3. (4 puntos)** Estás trabajando en un nuevo videojuego llamado *CiudadMatic*: un simulador de ciudades, llenas de edificios de distinto tipo. Será posible construir y reparar estos edificios gastando dinero, y al final de cada turno (después de haber construido y reparado tantos edificios como se quiera), recaudar los impuestos que generen. Necesitarás implementar las siguientes operaciones:

- *CiudadMatic*: inicializa una nueva ciudad vacía, con el dinero que se pase como argumento disponible para ser gastado.
- *nuevoTipo*: añade un nuevo tipo de edificio al sistema, con un identificador proporcionado por el usuario (por ejemplo, “bar”), un coste de construcción, una cantidad de impuestos generada por turno, y una calidad de base (máximo de turnos sin reparar). No devuelve nada.
- *insertaEdificio*: dado el nombre de un edificio (por ejemplo, “El Bar de Moe”), y el identificador de su tipo, y asumiendo que se disponga del dinero necesario para construirlo, añade el edificio a la ciudad y resta su coste de construcción del dinero disponible. No devuelve nada.
- *reparaEdificio*: repara el edificio cuyo identificador se pase como argumento a “como recién construido”, a un coste del 10 % del coste de construcción (descartando los decimales), independientemente de lo estropeado que estuviese. No devuelve nada.
- *finTurno*: todos los edificios construidos generan los impuestos que les corresponden por su tipo. Después, todos se estropean por un punto de calidad, y aquellos que lleguen a 0 son derribados y eliminados de la ciudad. Devuelve el dinero total disponible para el nuevo turno (impuestos generados + dinero no gastado del turno anterior).
- *listaEdificios*: dado un identificador de tipo de edificio, devuelve una lista con los edificios de ese tipo que están actualmente construidos, por orden de antigüedad (primero el más viejo).

Desarrolla en C++ una implementación de la clase *CiudadMatic* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. En cada operación, indica también, de forma razonada, su complejidad.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Final, 3 de Junio de 2014.

- 1. (3 puntos)** Alonso Rodríguez tiene que hacer la compra de la semana. Ha hecho una lista de n productos que quiere comprar. En su barrio hay m supermercados en cada uno de los cuales se dispone de todos esos productos. Pero como es un comprador compulsivo no quiere comprar más de tres productos en cada uno de los supermercados ya que así pasa más tiempo comprando (se puede suponer que $n \leq 3m$). Dispone de una lista de precios de los productos en cada uno de los supermercados. Se pide diseñar un algoritmo de vuelta atrás que permita a Alonso decidir cómo hacer la compra de forma que compre todo lo que necesita y que el coste total sea mínimo.
- 2. (3 puntos)** Dado un árbol binario, en cuya raíz se encuentra situado un tesoro y cuyos nodos internos pueden contener un dragón o no contener nada, se pide diseñar un algoritmo que nos indique la hoja del árbol cuyo camino hasta la raíz tenga el menor número de dragones. En caso de que existan varios caminos con el mismo número de dragones, el algoritmo devolverá el que se encuentre más a la izquierda de todos ellos.

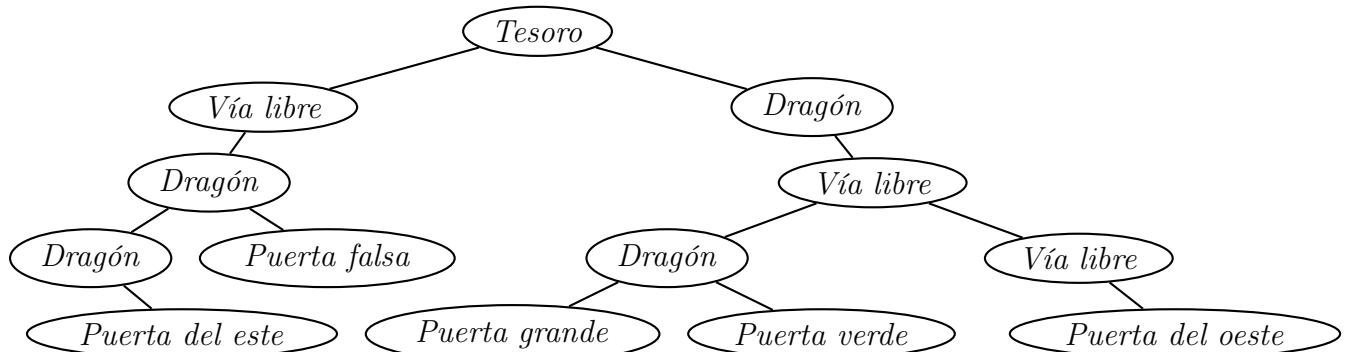
Para ello implementar una función que reciba un árbol binario cuyos nodos almacenan cadenas de caracteres:

- La raíz tiene la cadena *Tesoro*
- Los nodos internos tienen la cadena *Dragón* para indicar que en el nodo hay un dragón o la cadena *Vía libre* para indicar que no hay dragón.
- En cada hoja se almacena un identificador que no puede estar repetido.

y devuelva el identificador de la hoja del camino seleccionado. El árbol tiene como mínimo un nodo raíz y un nodo hoja diferente de la raíz. La operación no se implementa como parte de ningún TAD.

El coste de la operación implementada debe ser $\mathcal{O}(n)$.

Por ejemplo, dado el siguiente árbol el algoritmo devolverá la cadena de caracteres *Puerta falsa*.



3. (4 puntos) Eres un prestigioso diseñador de videojuegos, y estás trabajando ahora en uno llamado *EspacioMatic*, en el que habrá naves espaciales con distintos módulos (motores, cabinas, escudos, láseres, etcétera). Necesitarás implementar, como poco, las siguientes operaciones:

- *EspacioMatic*: inicializa el sistema de juego.
- *nuevaNave*: añade una nueva nave espacial (vacía) al sistema, con el identificador numérico proporcionado. Si se intenta usar un identificador ya existente produce error. No devuelve nada.
- *equipaNave*: dado el identificador de una nave, un nombre de módulo (una cadena como “motor”, “cabina”, “láser”, etcétera) y un nivel de funcionalidad (un entero ≥ 1), añade el módulo correspondiente a esa nave con el nivel indicado. Si esa nave ya tenía ese módulo, se suma el nuevo nivel al anterior (esto permite reparar módulos de naves). No devuelve nada.
- *estropeaNave*: dado el identificador de una nave, y un nombre de módulo, resta 1 al nivel de ese módulo en esa nave (asumiendo que tuviese un nivel > 0). Devuelve `true` si el módulo existía y tenía un nivel positivo antes de hacer la resta, ó `false` si ha sido imposible restar nivel ya que el módulo no existía o estaba ya a 0.
- *navesDefectuosas*: devuelve una lista de identificadores de naves que tienen uno o más módulos completamente estropeados (con un nivel de 0).
- *modulosNave*: dado el identificador de una nave, devuelve una lista con los nombres de los módulos que tiene equipados (tengan el nivel que tengan), ordenada alfabéticamente.

Desarrolla en C++ una implementación de la clase *EspacioMatic* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. En cada operación, indica también, de forma razonada, su complejidad.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Segundo Cuatrimestre, 3 de Junio de 2014.

1. (3 puntos) Extiende el TAD Lista visto en clase con una nueva operación cuya cabecera en C++ es:

```
void inserta(const T &elem, int pos);
```

que inserta el elemento `elem` en la posición `pos`, de forma que cuando `pos` es 0 se añade *al principio* de la lista (operación `Cons`), mientras que cuando es igual al número de elementos de la lista, se insertará al final (como `ponDr`). Indica la complejidad de la operación.

Si llamas a otros métodos, *debes implementarlos también*.

A continuación aparece, a modo de recordatorio, las partes relevantes del TAD Lista:

```
template <class T>
class Lista {
    ...
private:
    class Nodo {
        Nodo() : _sig(NULL), _ant(NULL) {}
        Nodo(const T &elem) : _elem(elem), _sig(NULL), _ant(NULL) {}
        Nodo(Nodo *ant, const T &elem, Nodo *sig) :
            _elem(elem), _sig(sig), _ant(ant) {}

        T _elem;
        Nodo *_sig;
        Nodo *_ant;
    };

    Nodo *_prim, *_ult;
    unsigned int _numElems;
};
```

2. (3 puntos) Dado un árbol binario, en cuya raíz se encuentra situado un tesoro y cuyos nodos internos pueden contener un dragón o no contener nada, se pide diseñar un algoritmo que nos indique la hoja del árbol cuyo camino hasta la raíz tenga el menor número de dragones. En caso de que existan varios caminos con el mismo número de dragones, el algoritmo devolverá el que se encuentre más a la izquierda de todos ellos.

Para ello implementar una función que reciba un árbol binario cuyos nodos almacenan cadenas de caracteres:

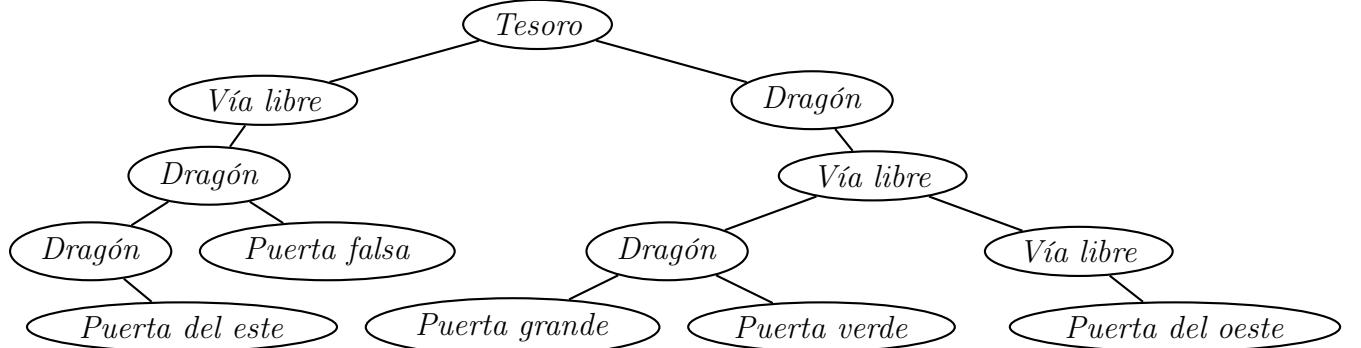
- La raíz tiene la cadena *Tesoro*

- Los nodos internos tienen la cadena *Dragón* para indicar que en el nodo hay un dragón o la cadena *Vía libre* para indicar que no hay dragón.
- En cada hoja se almacena un identificador que no puede estar repetido.

y devuelva el identificador de la hoja del camino seleccionado. El árbol tiene como mínimo un nodo raíz y un nodo hoja diferente de la raíz. La operación no se implementa como parte de ningún TAD.

El coste de la operación implementada debe ser $\mathcal{O}(n)$.

Por ejemplo, dado el siguiente árbol el algoritmo devolverá la cadena de caracteres *Puerta falsa*.



- 3. (4 puntos)** Eres un prestigioso diseñador de videojuegos, y estás trabajando ahora en uno llamado *EspacioMatic*, en el que habrá naves espaciales con distintos módulos (motores, cabinas, escudos, láseres, etcétera). Necesitarás implementar, como poco, las siguientes operaciones:

- *EspacioMatic*: inicializa el sistema de juego.
- *nuevaNave*: añade una nueva nave espacial (vacía) al sistema, con el identificador numérico proporcionado. Si se intenta usar un identificador ya existente produce error. No devuelve nada.
- *equipaNave*: dado el identificador de una nave, un nombre de módulo (una cadena como “motor”, “cabina”, “láser”, etcétera) y un nivel de funcionalidad (un entero ≥ 1), añade el módulo correspondiente a esa nave con el nivel indicado. Si esa nave ya tenía ese módulo, se suma el nuevo nivel al anterior (esto permite reparar módulos de naves). No devuelve nada.
- *estropeaNave*: dado el identificador de una nave, y un nombre de módulo, resta 1 al nivel de ese módulo en esa nave (asumiendo que tuviese un nivel > 0). Devuelve **true** si el módulo existía y tenía un nivel positivo antes de hacer la resta, ó **false** si ha sido imposible restar nivel ya que el módulo no existía o estaba ya a 0.
- *navesDefectuosas*: devuelve una lista de identificadores de naves que tienen uno o más módulos completamente estropeados (con un nivel de 0).
- *modulosNave*: dado el identificador de una nave, devuelve una lista con los nombres de los módulos que tiene equipados (tengan el nivel que tengan), ordenada alfabéticamente.

Desarrolla en C++ una implementación de la clase *EspacioMatic* basada en otros TADs conocidos, optimizando la complejidad temporal de las operaciones. En cada operación, indica también, de forma razonada, su complejidad.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software (grupo A)

Examen Parcial, 10 de Febrero de 2014.

- 1. (3 puntos)** Especifica, diseña y verifica o especifica y deriva, un algoritmo iterativo de coste lineal que dado un vector v de enteros, un valor entero positivo m tal que $0 \leq m \leq \text{longitud}(v)$ y un valor s igual a la suma de las m primeras posiciones del vector, devuelva el valor máximo que se puede obtener sumando m elementos consecutivos del vector.

Por ejemplo, sea $m = 3$, $s = 10$ y el vector

5	-4	9	7	-5	6	-1	10	0	3
---	----	---	---	----	---	----	----	---	---

la suma máxima de 3 elementos consecutivos es $15 = 6 -1 + 10$.

Justificar el coste de la función implementada.

- 2. (3 puntos)** Un vector de enteros mayores que 0 de longitud 2^n (donde n es un número natural) es *caucásico* si el valor absoluto de la diferencia entre el número de elementos pares de sus mitades es, a lo sumo, 2 y cada mitad también es *caucásica*.

Algunos ejemplos:

- $\{2, 4, 6, 8 \| 1, 3, 5, 7\}$ No es *caucásico*, porque su primera mitad tiene 4 elementos pares y la segunda 0.
- $\{2, 4, 6, 8 \| 2, 8, 5, 10\}$ Es *caucásico*.
- $\{2, 4, 8, 12, 3, 7, 9, 21 \| 10, 20, 30, 1, 3, 5, 7, 40\}$ No es *caucásico* ya que la primera mitad no lo es.

Diseña un algoritmo *recursivo* que determine si un vector de longitud 2^n es *caucásico*. El algoritmo debe de ser eficiente. Determina **justificadamente** la complejidad.

- 3. (4 puntos)** Deseamos organizar un festival de rock al aire libre para lo cual vamos a contratar exactamente a N artistas de entre M disponibles ($N < M$). No todos los artistas aceptan tocar juntos en el festival. Los “vetos” entre artistas son conocidos de antemano. Para cada artista $i \in \{1..M\}$ conocemos el beneficio o pérdida generado por dicho artista $B[i]$, es decir si $B[i] > 0$, dicho artista genera beneficio mientras que si $B[i] < 0$ genera pérdida. Diseñar un algoritmo de vuelta atrás que resuelva el problema de planificar el festival que maximice la suma de los beneficios/pérdidas de los artistas participantes.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Final, 11 de septiembre de 2013.

1. (3 puntos)

Especifica y deriva, o especifica, diseña y verifica, una función *iterativa* que dados dos arrays A y B de longitudes n y m respectivamente ($n \leq m$) de enteros ordenados de manera estrictamente creciente, determine si todos los elementos de A aparecen en B . La función debe tener coste **lineal** y lo debes justificar.

2. (4 puntos)

Te han contratado para implementar un sistema de gestión de barcos de pesca. Cada barco tiene una bodega de carga donde los pescadores que van en el barco van depositando las capturas que realizan, anotando siempre la especie del pez y su peso. Cuando el barco llega a puerto, cada pescador se lleva a casa lo que ha pescado de cada especie.

Las operaciones públicas del TAD *BarcoMatic* son:

- *nuevo*: Crea una nueva instancia de la estructura *BarcoMatic*, recibiendo como argumento un el peso máximo (en kilos) admitido en la bodega.
- *altaPescador*: Da de alta a un pescador, identificado por su nombre. No devuelve nada.
- *nuevaCaptura*: Registra que un pescador (que debe estar registrado) ha pescado un ejemplar de tantos kilos de una especie concreta. Las especies y los pescadores se indican mediante sus nombres, y el peso en kilos se especifica mediante un número. Si el peso de la captura, añadido a la bodega, haría que la bodega excediese su capacidad, esta operación debe fallar.
- *capturasPescador*: Recibe el nombre de un pescador, y devuelve una lista de parejas especie-kilos. Si, para una especie dada, el pescador no ha pescado nada, no la debes incluir en la lista devuelta. Puedes asumir la existencia de un TAD *Pareja* $\langle A, B \rangle$ similar al visto en clase.
- *kilosEspecie*: Recibe el nombre de una especie, y devuelve el número total de kilos de esa especie pescados, sumando las capturas de todos los pescadores.
- *kilosPescador*: Recibe el nombre de un pescador, y devuelve el número total de kilos que ha pescado, sumando todas las especies.
- *bodegaRestante*: Devuelve el número de kilos restantes en la bodega.

Se pide: prototipos de las operaciones públicas, representación eficiente del TAD (basada en tipos vistos durante el curso), coste de cada operación (*especificando qué representa la N en cada caso concreto*) e implementación en C++ de todas las operaciones.

3. (3 puntos)

Podemos representar la discretización de una función $f(x)$ mediante una tabla (*Tabla* $\langle int, float \rangle$) que asocia a ciertos valores enteros de abscisa (x_1, x_2, \dots) sus correspondientes valores reales de ordenada ($f(x_1), f(x_2), \dots$). Suponiendo que:

- La tabla sólo contiene valores de abscisa positivos y consecutivos comenzando en $x = 1$.
- La función $f(x)$ discretizada es estrictamente creciente ($x_1 < x_2 \rightarrow f(x_1) < f(x_2)$).

Se pide implementar una función con coste $\mathcal{O}(\log n)$ que, dada una tabla como la anterior y un valor de ordenada y , devuelva la abscisa entera x asociada a y si el punto (x,y) existe en la tabla, y -1 si no existe.

```
int buscaX(const Tabla<int, float> &f, float y)
```

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Segundo Examen Parcial, 11 de septiembre de 2013.

1. (4 puntos)

Te han contratado para implementar un sistema de gestión de barcos de pesca. Cada barco tiene una bodega de carga donde los pescadores que van en el barco van depositando las capturas que realizan, anotando siempre la especie del pez y su peso. Cuando el barco llega a puerto, cada pescador se lleva a casa lo que ha pescado de cada especie.

Las operaciones públicas del TAD *BarcoMatic* son:

- *nuevo*: Crea una nueva instancia de la estructura *BarcoMatic*, recibiendo como argumento un el peso máximo (en kilos) admitido en la bodega.
- *altaPescador*: Da de alta a un pescador, identificado por su nombre. No devuelve nada.
- *nuevaCaptura*: Registra que un pescador (que debe estar registrado) ha pescado un ejemplar de tantos kilos de una especie concreta. Las especies y los pescadores se indican mediante sus nombres, y el peso en kilos se especifica mediante un número. Si el peso de la captura, añadido a la bodega, haría que la bodega excediese su capacidad, esta operación debe fallar.
- *capturasPescador*: Recibe el nombre de un pescador, y devuelve una lista de parejas especie-kilos. Si, para una especie dada, el pescador no ha pescado nada, no la debes incluir en la lista devuelta. Puedes asumir la existencia de un TAD *Pareja* $\langle A, B \rangle$ similar al visto en clase.
- *kilosEspecie*: Recibe el nombre de una especie, y devuelve el número total de kilos de esa especie pescados, sumando las capturas de todos los pescadores.
- *kilosPescador*: Recibe el nombre de un pescador, y devuelve el número total de kilos que ha pescado, sumando todas las especies.
- *bodegaRestante*: Devuelve el número de kilos restantes en la bodega.

Se pide: prototipos de las operaciones públicas, representación eficiente del TAD (basada en tipos vistos durante el curso), coste de cada operación (*especificando qué representa la N en cada caso concreto*) e implementación en C++ de todas las operaciones.

2. (3 puntos)

Dado un árbol binario, se llama *altura mínima* del árbol a la menor de sus distancias desde la raíz a un subárbol vacío. Se puede especificar formalmente mediante las ecuaciones:

$$\begin{aligned} hmin(avacio) &= 0 \\ hmin(cons(i, x, d)) &= 1 + \min(hmin(i), hmin(d)) \end{aligned}$$

Decimos que un árbol binario es *zurdo* si o, bien es vacío, o si no lo es, ambos hijos son zurdos y la altura mínima del hijo izquierdo nunca es menor que la del hijo derecho.

Se pide:

1. Dibujar algunos ejemplos no triviales de árboles zurdos y no zurdos
2. Implementar en C++ una función, de coste lineal con el número de nodos, que dado un árbol binario, determine si es o no zurdo. Justificar dicho coste.

3. (3 puntos)

Podemos representar la discretización de una función $f(x)$ mediante una tabla ($\text{Tabla} < \text{int}, \text{float} >$) que asocia a ciertos valores enteros de abscisa (x_1, x_2, \dots) sus correspondientes valores reales de ordenada ($f(x_1), f(x_2), \dots$). Suponiendo que:

- La tabla sólo contiene valores de abscisa positivos y consecutivos comenzando en $x = 1$.
- La función $f(x)$ discretizada es estrictamente creciente ($x_1 < x_2 \rightarrow f(x_1) < f(x_2)$).

Se pide implementar una función con coste $\mathcal{O}(\log n)$ que, dada una tabla como la anterior y un valor de ordenada y , devuelva la abscisa entera x asociada a y si el punto (x, y) existe en la tabla, y -1 si no existe.

```
int buscaX(const Tabla<int , float> &f , float y)
```

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Final, 12 de junio de 2013.

1. (3 puntos)

Se tiene un vector de enteros $a[0..n-1]$, que puede ser vacío, cuyo propósito es representar los coeficientes de un polinomio en una variable: $a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$. Se pide especificar, y derivar, o diseñar y verificar una función iterativa que, dado un vector de coeficientes y un valor de x , devuelva la evaluación del polinomio para ese valor. El coste ha de ser **lineal** y se ha de justificar su cálculo. La evaluación ha de seguir **obligatoriamente** la secuencia de cómputos impuesta por la llamada Regla de Horner:

$$p(x) = a_0 + a_1(\dots a_{n-3} + (a_{n-2} + a_{n-1}x)x\dots)x$$

Si el polinomio es vacío, se entenderá que su evaluación para cualquier x da cero.

2. (4 puntos)

Desarrolla MultaMatic, el nuevo sistema de gestión de multas de tráfico por exceso de velocidad. La red de carreteras contiene tramos vigilados en los que se coloca una cámara al principio del tramo y otra al final. Cada vez que un coche pasa frente a una cámara, se toma una foto de su matrícula y se apunta el momento en que pasó; si el tiempo transcurrido entre la foto del comienzo y la del final es demasiado breve, se le pone una multa. Para simplificar, asumiremos que los tramos no comparten cámaras ni se solapan entre sí. Las operaciones públicas del TAD son:

- *insertaTramo*: añade un nuevo tramo al sistema. Recibe un identificador de tramo, los identificadores de sus cámaras inicial y final, y el número mínimo de segundos que deben transcurrir entre las fotos de comienzo y final para *no* recibir multa. Si el tramo ya existía debe generar un error.
- *fotoEntrada*: se invoca cada vez que un coche entra en un tramo vigilado. Recibe el identificador de la cámara, la matrícula del coche, y el instante actual (en segundos desde el 1 de enero de 1970).
- *fotoSalida*: se invoca cada vez que un coche sale de un tramo vigilado. Recibe el identificador de la cámara, la matrícula del coche, y el instante actual. Si el coche ha ido demasiado rápido en el tramo, se le multará.
- *multasPorMatricula*: devuelve el número de multas asociadas a una matrícula.
- *multasPorTramos*: devuelve una lista con las matrículas de los coches multados en un determinado tramo. Si un coche ha sido multado varias veces, su matrícula aparecerá varias veces en la lista. Si el tramo no existe debe generar un error.

Se pide: prototipos de las operaciones públicas, representación eficiente del TAD (basada en tipos vistos durante el curso), coste de cada operación e implementación en C++ de todas las operaciones.

3. (3 puntos)

Dada una lista de números a_1, a_2, \dots, a_n se dice que dos números producen una inversión si $a_i > a_j$ con $i < j$. Se pide un algoritmo que calcule el número de inversiones que existen en una lista de números. Se debe utilizar el método **Divide y Vencerás**. Se valorará la eficiencia del algoritmo implementado. **Nota:** Se puede conseguir un coste del orden de $O(n \log n)$.

Calcular el coste del algoritmo implementado planteando la recurrencia y resolviéndola.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Segundo Examen Parcial, 12 de junio de 2013.

1. (4 puntos)

Desarrolla MultaMatic, el nuevo sistema de gestión de multas de tráfico por exceso de velocidad. La red de carreteras contiene tramos vigilados en los que se coloca una cámara al principio del tramo y otra al final. Cada vez que un coche pasa frente a una cámara, se toma una foto de su matrícula y se apunta el momento en que pasó; si el tiempo transcurrido entre la foto del comienzo y la del final es demasiado breve, se le pone una multa. Para simplificar, asumiremos que los tramos no comparten cámaras ni se solapan entre sí. Las operaciones públicas del TAD son:

- *insertaTramo*: añade un nuevo tramo al sistema. Recibe un identificador de tramo, los identificadores de sus cámaras inicial y final, y el número mínimo de segundos que deben transcurrir entre las fotos de comienzo y final para *no* recibir multa. Si el tramo ya existía debe generar un error.
- *fotoEntrada*: se invoca cada vez que un coche entra en un tramo vigilado. Recibe el identificador de la cámara, la matrícula del coche, y el instante actual (en segundos desde el 1 de enero de 1970).
- *fotoSalida*: se invoca cada vez que un coche sale de un tramo vigilado. Recibe el identificador de la cámara, la matrícula del coche, y el instante actual. Si el coche ha ido demasiado rápido en el tramo, se le multará.
- *multasPorMatricula*: devuelve el número de multas asociadas a una matrícula.
- *multasPorTramos*: devuelve una lista con las matrículas de los coches multados en un determinado tramo. Si un coche ha sido multado varias veces, su matrícula aparecerá varias veces en la lista. Si el tramo no existe debe generar un error.

Se pide: prototipos de las operaciones públicas, representación eficiente del TAD (basada en tipos vistos durante el curso), coste de cada operación e implementación en C++ de todas las operaciones.

2. (3 puntos)

Los árboles de búsqueda se comportan mejor si están equilibrados. Se entiende por *árbol equilibrado* aquél cuya talla de su hijo izquierdo y de su hijo derecho difieren a lo sumo en una unidad, y además ambos subárboles están equilibrados. Extiende la clase de los árboles binarios para que incorpore un nuevo método que decida en tiempo **lineal** si el árbol binario está equilibrado.

3. (3 puntos)

Dada una lista de números a_1, a_2, \dots, a_n se dice que dos números producen una inversión si $a_i > a_j$ con $i < j$. Se pide un algoritmo que calcule el número de inversiones que existen en una lista de números. Se debe utilizar el método de divide y vencerás. Se valorará la eficiencia del algoritmo implementado. **Nota:** Se puede conseguir un coste del orden de $O(n \log n)$.

Calcular el coste del algoritmo implementado planteando la recurrencia y resolviéndola.

Estructuras de Datos y Algoritmos

Grados en Ingeniería Informática, de Computadores y del Software

Examen Parcial, 11 de Febrero de 2013.

- 1. Diseño iterativo (4 puntos)** Especificar, diseñar, verificar y calcular el coste de un algoritmo que, dado un vector v de enteros que puede ser vacío ($n \geq 0$) y que tan solo puede contener los valores 0 y 1, devuelva un booleano que indique si el vector tiene la forma:

$$1 | 1 | \cdots | 1 | 1 | 0 | 0 | \cdots | 0 | 0$$

Las dos zonas de ceros y unos pueden tener cualquier longitud, incluida la nula.

Sugerencia (no penaliza si no se sigue): la verificación puede resultar más fácil si se elige un invariante de la forma:

$$1 | \cdots | 1 | ? ? ? ? ? | 0 | \cdots | 0$$

- 2. Diseño recursivo (4 puntos)** Se define el *histograma* de un vector v de enteros como otro vector w que contiene en la posición i la suma $v[0] + \dots + v[i]$. Así, por ejemplo, el vector $\{1, 2, 4, 3, -2\}$ tendría como histograma $\{1, 3, 7, 10, 8\}$.

Se pide especificar, diseñar, demostrar la corrección y calcular el coste de un algoritmo recursivo, lo más eficiente posible, que dado un vector de enteros que puede ser vacío devuelva su histograma.

- 3. Diseño TADs (2 puntos)** Diseñar un TAD *Polinomio* que permita manejar polinomios con coeficientes enteros en una indeterminada. Ejemplos de polinomios serían $3x^4 + 1$, $-2x^{99} + 5x^2$, ó $x^{507} + x^{200} + x$. El tipo *Polinomio* deberá permitir, además de la construcción de polinomios, su suma, resta y multiplicación (que devolverán nuevos polinomios); su evaluación para una x concreta; y la posibilidad de verificar si un polinomio está vacío o es igual a otro dado.

Se pide:

- Elegir un *tipo representante* para el TAD eficiente en tiempo y espacio, suponiendo que los polinomios están formados por k o menos monomios (elementos de la forma $c \cdot x^n$), donde tanto c como n pueden llegar a ser muy grandes (aunque quepan en un `int` estándar). La constante k debe formar parte de la implementación de tu TAD.
- Dar el *invariante de la representación* y la *relación de equivalencia* de la representación elegida.
- Implementar el archivo `polinomio.h` con la definición de la clase. Se debe incluir las operaciones públicas necesarias para manejar el tipo, ya sean constructoras, observadoras o modificadoras; y la parte privada con la representación. Para cada operación debe darse su precondición y postcondición.

Nota: No hace falta que implementes ninguna de estas operaciones; basta con que las declares y describas correctamente.

Estructuras de Datos y Algoritmos
Segundo curso de los Grados en Ingeniería Informática
Examen Final, Septiembre de 2012.

1. **(3,5 puntos)** Especificar y derivar formalmente, o diseñar y verificar, un algoritmo iterativo de coste lineal, que reciba un vector no-vacio V de enteros y un entero N con la longitud de V , y calcule en una sola pasada el número de veces que aparece repetido su elemento menor.
2. **(3 puntos)** Suponemos declarado un tipo **personaje** con las operaciones:

```
esDragon    : personaje -> bool  
esPrincesa : personaje -> bool
```

Tenemos una variable del TAD **Arbin<personaje>**, cada uno de cuyos nodos puede ser un dragón, una princesa, o ninguno de los dos. Un nodo se dice *accesible*, si en el camino que lo une con la raíz no hay ningún nodo-dragón. Se pide:

1. **(1 punto)** Implementar, como **usuaria del TAD**, una operación que devuelva el número de nodos-princesa accesibles
2. **(2 puntos)** Implementar, como **usuaria del TAD**, una operación que devuelva la princesa accesible más cercana a la raíz. Se puede suponer que existe al menos una princesa accesible. El algoritmo **no debe inspeccionar** nodos que se encuentren a mayor profundidad que la princesa más cercana.
3. **(3,5 puntos)** Los ferrys son barcos que sirven para trasladar coches de una orilla a otra de un río. Los coches esperan en fila al ferry y cuando éste llega un operario les va dejando entrar.

En nuestro caso el ferry tiene espacio para dos filas de coches (la fila de babor y la fila de estribor) cada una de N metros. El operario conoce de antemano la longitud de cada uno de los coches que están esperando a entrar en él y debe, según van llegando, mandarles a la fila de babor o a la fila de estribor, de forma que el ferry quede lo más cargado posible. Ten en cuenta que los coches deben entrar en el ferry **en el orden en que aparecen** en la fila de espera, por lo que en el momento en que un coche ya no entra en el ferry porque no hay hueco suficiente para él, no puede entrar ningún otro coche que esté detrás aunque sea más pequeño y sí hubiera hueco para él.

Implementa una función que reciba la capacidad del ferry en metros y la colección con las longitudes de los coches que están esperando (puedes utilizar el TAD que mejor te venga) y devuelva la colocación óptima de los coches, entendiendo ésta como la secuencia de filas en las que se van metiendo los coches. Supón la existencia de los símbolos **BABOR** y **ESTRIBOR**.

Ejemplo: Si el ferry tiene 5 metros de longitud y en la fila tenemos coches con longitudes (del primero al último) 2.5, 3, 1, 1, 1.5, 0.7 y 0.8, una solución óptima es [BABOR, ESTRIBOR, ESTRIBOR, ESTRIBOR, BABOR, BABOR].

Estructuras de Datos y Algoritmos

Grado en Ingeniería Informática

Examen Final, Junio de 2012.

1. **(4 puntos)** Especificar y derivar formalmente, o diseñar y verificar, un algoritmo iterativo eficiente que dado un vector no vacío de n enteros obtenga la longitud del segmento más largo de elementos consecutivos ordenados crecientemente por \leq . Calcular también su coste.
2. **(2 puntos)** Diseñar recursivamente, y sin usar estructuras de datos auxiliares ni el iterador de la clase, un método de la clase `Arbus` que dada una clave k que se sabe está en el árbol, devuelva la siguiente clave en orden creciente. Calcular el coste de dicho método.
3. **(4 puntos)**

Implementar una función que encuentre la forma *más rápida* de viajar desde una casilla de salida hasta una casilla de llegada de una rejilla. Cada casilla de la rejilla está etiquetada con una letra, de forma que en el camino desde la salida hacia la llegada se debe ir formando (de forma cíclica) una palabra dada. Desde una celda se puede ir a cualquiera de las cuatro celdas adyacentes.

Como ejemplo, a continuación aparece la forma más corta de salir de una rejilla de 5×8 en la que el punto de salida está situado en la posición $(0, 4)$ y hay que llegar a la posición $(7, 0)$ y la palabra que hay que ir formando por el camino es EDA.

0	M	D	A	A	E	E	D	A
1	A	E	E	D	D	A	N	D
2	D	B	D	X	E	D	A	E
3	E	A	E	D	A	R	T	D
4	E	D	M	P	L	E	D	A

0 1 2 3 4 5 6 7

Para la implementación puedes suponer la existencia de dos variables globales `N` y `M` que determinan el tamaño de la rejilla, así como la información de la propia rejilla,

```
char lab[N][M];
```

También puedes suponer la existencia de una variable global que contiene la palabra a utilizar,

```
Lista<char> palabra;
```

Las función recibirá el punto origen y el punto destino y deberá determinar la forma más rápida de viajar de uno a otro (si es que esto es posible), dando las direcciones que hay que ir cogiendo (en el caso del ejemplo será E, N, E, E, E, etc.). Si necesitas parámetros adicionales, añádelos indicando sus valores iniciales.

Estructuras de Datos y Algoritmos

Grado en Ingeniería Informática

Segundo Examen Parcial, Junio de 2012.

1. **(2 puntos)** Diseñar recursivamente, y sin usar estructuras de datos auxiliares ni el iterador de la clase, un método de la clase `Arbus` que dada una clave k que se sabe está en el árbol, devuelva la siguiente clave en orden creciente. Calcular el coste de dicho método.
2. **(4 puntos)** Se desea un TAD `Consultorio` que simule el comportamiento de un consultorio médico simplificado. Dicha especificación hará uso de los TADs `Medico` y `Paciente`, que se suponen ya conocidos. Las operaciones del TAD `Consultorio` son las siguientes:

- `ConsultorioVacio`: crea un nuevo consultorio vacío.
- `nuevoMedico`: da de alta un nuevo médico en el consultorio.
- `pideConsulta`: un paciente se pone a la espera de ser atendido por un médico.
- `pideConEnchufe`: igual que la anterior, pero el paciente será atendido por delante del resto de pacientes.
- `tienePacientes`: informa de si un médico tiene pacientes esperando.
- `siguientePaciente`: consulta el paciente al que le toca el turno para ser atendido por un médico dado.
- `atiendeConsulta`: elimina el siguiente paciente de un médico.
- `numCitas`: indica el número de citas que un mismo paciente tiene en todo el consultorio.

Se pide:

1. La cabecera de cada operación, indicando además si es generadora, observadora o modificadora, y si es total o parcial.
 2. La definición de la representación elegida para el TAD.
 3. El coste esperado para cada operación con esa representación.
 4. La implementación con todo detalle de las operaciones `pideConsulta`, `atiendeConsulta` y `numCitas`.
3. **(4 puntos)** Implementar una función que encuentre la forma más rápida de viajar desde una casilla de salida hasta una casilla de llegada de una rejilla. Cada casilla de la rejilla está etiquetada con una letra, de forma que en el camino desde la salida hacia la llegada se debe ir formando (de forma cíclica) una palabra dada. Desde una celda se puede ir a cualquiera de las cuatro celdas adyacentes.

Como ejemplo, a continuación aparece la forma más corta de salir de una rejilla de 5×8 en la que el punto de salida está situado en la posición $(0, 4)$ y hay que llegar a la posición $(7, 0)$ y la palabra que hay que ir formando por el camino es EDA.

0	M	D	A	A	E	E	D	A
1	A	E	E	D	D	A	N	D
2	D	B	D	X	E	D	A	E
3	E	A	E	D	A	R	T	D
4	E	D	M	P	L	E	D	A

0 1 2 3 4 5 6 7

Para la implementación puedes suponer la existencia de dos variables globales `N` y `M` que determinan el tamaño de la rejilla, así como la información de la propia rejilla,

```
char lab[N][M];
```

También puedes suponer la existencia de una variable global que contiene la palabra a utilizar,

```
Lista<char> palabra;
```

La función recibirá el punto origen y el punto destino y deberá determinar la forma más rápida de viajar de uno a otro (si es que esto es posible), dando las direcciones que hay que ir cogiendo (en el caso del ejemplo será E, N, E, E, E, etc.). Si necesitas parámetros adicionales, añádelos indicando sus valores iniciales.

Estructuras de Datos y Algoritmos

Grado en Ingeniería Informática

Examen Parcial, Febrero de 2012.

1. **(0,5 puntos)** El algoritmo A tarda $207 + 4n^2$ segundos en resolver un problema de tamaño n , mientras que el algoritmo B lo resuelve en $3n^4$ segundos. Razonar para qué valores de n es mejor cada uno de ellos.

2. **(0,5 puntos)** Compara las clases de complejidad O y Θ de las siguientes parejas de funciones:

1. $n \log n$ y $n\sqrt{n}$.
2. $(n+1)^2$ y $(n-1)^2$.
3. $(n+1)!$ y $n!$.
4. n^a y a^n , con $a \in \mathbb{R}^+, a > 1$.

3. **(0,5 puntos)** En el siguiente algoritmo, calcula el número exacto de veces que se ejecuta la acción A . Si $A \in O(n)$, indicar también cuál es la complejidad asintótica del algoritmo.

```
for (int i=0; i<n; i++)
    for (int j=0; j<=i; j++)
        {A}
```

4. **(0,5 puntos)** Los dos algoritmos siguientes reciben un vector de enteros con $n \geq 2$ elementos. Escribir formalmente sus postcondiciones:

1. Devuelve un booleano b que indica si hay exactamente una posición cuyo contenido vale el triple de su índice.
2. Devuelve un entero s que es el mayor valor que es posible conseguir sumando dos elementos del vector que contengan valores distintos.

5. **(4 puntos)** Un vector de n enteros con $n \geq 1$ diremos que es *montaña* si sus valores crecen estrictamente hasta un cierto índice llamado *cumbre* y a partir de él decrecen estrictamente. Se admiten vectores montaña cuya cumbre sea el primer índice o el último. Especificar, diseñar, verificar y razonar el coste de un algoritmo iterativo que, dado un vector montaña, devuelva su cumbre.

6. **(4 puntos)** Especificar, diseñar, verificar y razonar el coste de un algoritmo recursivo que, dado un vector de $n \geq 0$ enteros estrictamente creciente, determine en tiempo logarítmico si alguno de sus elementos coincide con el valor de su índice.