enunciados-examenes.pdf

marmar_co

Estructuras de Datos

2º Grado en Ingeniería Informática

Facultad de Informática
Universidad Complutense de Madrid

Estructuras de datos

Grados de la Facultad de Informática (UCM)

1 de julio de 2022

Normas de realización del examen

1. El examen dura **3 horas**.
2. Debes desarrollar e implementar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección `http://exacrc`.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen.
4. Escribe tu **nombre y apellidos** en el espacio reservado para ello en los ficheros proporcionados.
5. Dispones de un fichero plantilla para cada ejercicio. Debes utilizarlo atendiendo a las instrucciones que se dan en cada fichero y escribiendo tu solución en los espacios reservados para ello, siempre entre las etiquetas `<answer>` y `</answer>`.
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.
7. Si necesitas consultar la documentación de C++, está disponible en `http://exacrc/cppreference`.

Ejercicio 1 (2 puntos)

Desparizar una lista consiste en eliminar sus elementos pares (dejando únicamente los impares), añadiendo esos pares en el mismo orden al final de otra lista.

En concreto, dada la clase `ListLinkedDouble<T>`, que implementa el TAD lista mediante listas doblemente enlazadas (circulares y con nodo fantasma) y un contador con el número de elementos, debes añadir un nuevo método `void desparizar(List & aux)`, que despariza la lista `this`, quedándose esta última con los elementos impares, y moviendo los nodos con valores pares en orden al final de la lista `aux` pasada como parámetro. Por ejemplo, dada la lista `l = [20, 4, 5, 7, 10, 3]`, y la lista `aux = []`, tras la llamada `l.desparizar(aux)` la lista `l` tiene los valores `[5, 7, 3]` y la lista `aux` los valores `[20, 4, 10]`.

La lista `aux` puede no estar vacía. Si en el ejemplo anterior `aux = [11, 67]`, después de ejecutarse `l.desparizar(aux)`, tendremos `aux = [11, 67, 20, 4, 10]`

Importante: Para la implementación del método no pueden crearse, directa o indirectamente, nuevos nodos mediante `new` ni borrar nodos mediante `delete`; han de reutilizarse los nodos de la lista de entrada. Tampoco se permite copiar valores de un nodo a otro. El coste de la operación ha de ser lineal con respecto a la longitud de la lista.

Entrada

La entrada comienza con un número que indica el número de casos de prueba que vienen a continuación. Cada caso de prueba consiste en dos líneas: la primera es la descripción de la lista a desparizar, mientras que la segunda es la lista `aux`. Cada lista se representa mediante una secuencia con sus elementos (números enteros distintos de cero), finalizando con `0`, que no forma parte de la lista.

Salida

Para cada caso de prueba se imprimirán dos líneas: una con el contenido de la lista `this` tras llamar al método `desparizar` y otra con el contenido de la lista `aux` tras esa misma llamada.

Entrada de ejemplo

```
4
1 2 3 4 5 6 7 8 9 10 0
0
2 1 0
22 0
4 6 8 0
1 0
0
2 0
```

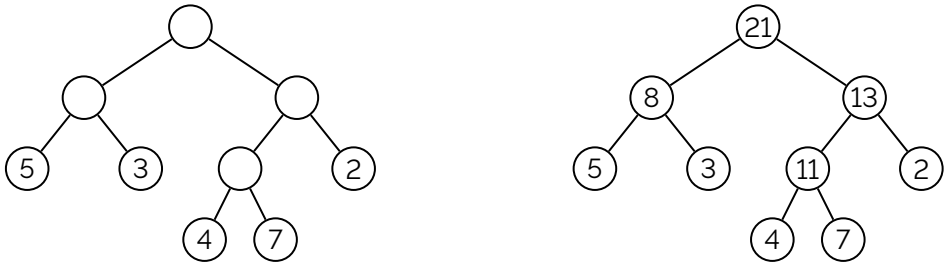
Salida de ejemplo

```
1 3 5 7 9
2 4 6 8 10
1
22 2

1 4 6 8

2
```

Ejercicio 2 (2 puntos)

Dado un árbol binario con números enteros en los nodos, de los cuales solamente nos interesan los valores en las hojas, queremos implementar una función `acumula` que recibe ese árbol y devuelve otro que tiene la misma estructura que el primero pero donde los valores de los nodos internos se han sustituido por la suma de las hojas que tienen como descendientes (es decir, un nodo interno *X* se sustituye por la suma de las hojas del subárbol que tiene a *X* como raíz).

Por ejemplo, si aplicáramos la operación al árbol de la izquierda (donde solamente aparecen los valores de las hojas) obtendríamos el árbol de la derecha.



Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en la descripción de un árbol binario de enteros, donde los nodos internos tendrán siempre el valor `0` y las hojas podrán tener cualquier valor entre 1 y 500. El árbol vacío se representa con un '`.`' y un árbol no vacío con raíz `R`, hijo izquierdo `i` e hijo derecho `dr` se representa como `(iz R dr)`.

Salida

Para cada caso de prueba, se escribirá una línea con la representación del árbol obtenido tras aplicar la operación `acumula` al árbol de la entrada.

Entrada de ejemplo

```
4
.
(. 0 (. 7 .))
(((. 3 .) 0 .) 0 (. 4 .))
(((. 5 .) 0 (. 3 .)) 0 (((. 4 .) 0 (. 7 .)) 0 (. 2 .)))
```

Salida de ejemplo

```
.
(. 7 (. 7 .))
(((. 3 .) 3 .) 7 (. 4 .))
(((. 5 .) 8 (. 3 .)) 21 (((. 4 .) 11 (. 7 .)) 13 (. 2 .)))
```

Ejercicio 3 (3 puntos)

Queremos implementar un TAD `GaleriaArte` que simule el comportamiento de una galería de arte simplificada. En una galería de arte se exponen las obras de diversos artistas. En cualquier momento la galería puede aceptar exponer nuevas obras, o puede vender alguna de las que tiene expuestas. Se hará uso de los tipos `artista` y `obra` que se representan con un `string`.

Las operaciones de la galería de arte son las siguientes:

- `nueva_obra(c,a,p)`: añade a la galería una nueva obra `c`, del artista `a`, con un precio `p`. Si la obra `c` ya estaba en la galería, se lanza una excepción del tipo `invalid_argument` con el mensaje `Obra ya en la galeria`.
- `venta_obra(c)`: simula la venta de la obra `c`, lo que incluye que se elimina de la galería de arte. Si la obra no está en la galería se lanzará una excepción del tipo `invalid_argument` con el mensaje `Obra no existente`.
- `mas_vendidos()`: devuelve un vector con todos los artistas cuyo total de ventas (suma de los precios de las obras que se han vendido de ese artista) es el máximo en la galería, ordenados por orden alfabético de menor a mayor.
- `obras_por_antiguedad(k)`: devuelve en un vector las *k* obras que más tiempo han permanecido en la galería sin venderse, ordenadas desde la que más tiempo lleva en la galería a la que menos. Si no hay *k* obras en la galería se devolverán todas ellas.

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.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea: el nombre de la operación seguido de sus argumentos. La palabra `FIN` en una línea indica el final de cada caso.

Los nombres de los artistas y obras de arte no tienen blancos.

Salida

Para cada caso de prueba se escribirán los datos que se piden. Las operaciones que generan salida son:

- `mas_vendidos`, escribe la cabecera: `Los artistas que mas han vendido son:` seguido del nombre de los artistas uno en cada línea.
- `obras_por_antiguedad`, escribe la cabecera `Obras mas antiguas en la galeria:`. En las líneas siguientes se escribe el nombre de las *k* obras que más tiempo llevan en la galería una por línea. Si no hay *k* obras en la galería se escribirán todas ellas.

Si alguna operación produce una excepción se mostrará el mensaje `ERROR:` seguido del mensaje de la excepción como resultado de la operación, y nada más.

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

Entrada de ejemplo

```
nueva_obra Autorretrato Goya 150
nueva_obra Las_meninas Velazquez 90
nueva_obra La_metamorfosis Dali 45
nueva_obra Fiesta_en_Figueras Dali 45
venta_obra La_metamorfosis
nueva_obra La_venus_del_espejo Velazquez 120
venta_obra Fiesta_en_Figueras
mas_vendidos
obras_por_antiguedad 2
venta_obra Las_meninas
mas_vendidos
nueva_obra Biarritz Sorolla 80
nueva_obra El_pie_herido Sorolla 50
venta_obra Autorretrato
mas_vendidos
nueva_obra La_rendicion_de_Breda Velazquez 130
nueva_obra Cosiendo_la_vela Sorolla 70
nueva_obra Aquelarre Goya 100
venta_obra El_pie_herido
nueva_obra La_metamorfosis Dali 90
obras_por_antiguedad 10
venta_obra La_metamorfosis
mas_vendidos
FIN
nueva_obra Autorretrato Goya 150
venta_obra Las_meninas
mas_vendidos
FIN
```

Salida de ejemplo

```
Los artistas que mas han vendido son:
Dali
Obras mas antiguas en la galeria:
Autorretrato
Las_meninas
Los artistas que mas han vendido son:
Dali
Velazquez
Los artistas que mas han vendido son:
Goya
Obras mas antiguas en la galeria:
La_venus_del_espejo
Biarritz
La_rendicion_de_Breda
Cosiendo_la_vela
Aquelarre
La_metamorfosis
Los artistas que mas han vendido son:
Dali
---
ERROR: Obra no existente
Los artistas que mas han vendido son:
---
```

Estructuras de datos

Grados de la Facultad de Informática (UCM)

27 de mayo de 2022

Normas de realización del examen

1. El examen dura **3 horas**.
2. Debes desarrollar e implementar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección `http://exacrc`.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen.
4. Escribe tu **nombre y apellidos** en el espacio reservado para ello en los ficheros proporcionados.
5. Dispones de un fichero plantilla para cada ejercicio. Debes utilizarlo atendiendo a las instrucciones que se dan en cada fichero y escribiendo tu solución en los espacios reservados para ello, siempre entre las etiquetas `<answer>` y `</answer>`.
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.
7. Si necesitas consultar la documentación de C++, está disponible en `http://exacrc/cppreference`.

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.

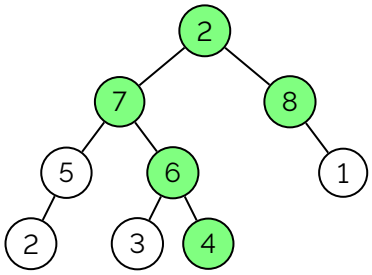
— Ryan Campbell

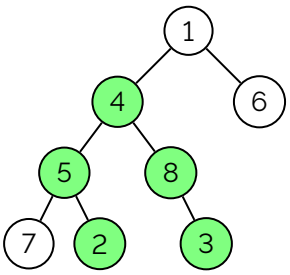
si lees esto me debes un besito

Ejercicio 1 (2 puntos)

Definimos un *camino* entre dos nodos de un árbol binario como una secuencia de nodos $n_1\ n_2\ \ldots\ n_k$ 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).

Dado un árbol binario de números enteros positivos nos queremos fijar en el camino que tiene la mayor cantidad de números pares. En particular, queremos saber esa cantidad máxima. Por ejemplo, en los siguientes árboles están marcados caminos con una cantidad máxima de números pares. En el árbol de la izquierda el camino tiene 4 pares y no hay ningún camino con más. En el árbol de la derecha el camino con más pares tiene 3. El camino señalado es uno de ellos, pero hay otros como el 2 5 4 8, o el 6 1 4 8 3 (todos con 3 pares).





Implementa una función que reciba un árbol de la clase `BinTree` y calcule la máxima cantidad de números pares que se pueden encontrar en un camino. Justifica el coste de esta función.

Entrada

La entrada comienza con el número de casos que vienen a continuación. Cada caso de prueba consiste en la descripción de un árbol binario.

Salida

Para cada caso, se escribirá la máxima cantidad de números pares que se puede encontrar en un camino.

Entrada de ejemplo

```
4
((((. 2 .) 5 .) 7 ((. 3 .) 6 (. 4 .))) 2 (. 8 (. 1 .)))
((((. 7 .) 5 (. 2 .)) 4 (. 8 (. 3 .))) 1 (. 6 .))
((. 3 .) 1 (. 5 (. 7 .)))
((. 2 .) 1 .)
```

Salida de ejemplo

```
4
3
0
1
```

Ejercicio 2 (2 puntos)

Queremos implementar una función que reciba una lista y la modifique para que tenga solamente la primera aparición de cada elemento diferente.

```
template<typename T>
void eliminar_repetidos(std::list<T> & l);
```

Por ejemplo, si la lista es `l = [1, 5, 1, 1, 2, 5, 2]` entonces `eliminar_repetidos(l)` haría que la lista se transformara en `l = [1, 5, 2]`. Es decir, solamente se conserva la primera aparición de cada valor, y estos mantienen su orden de aparición.

Implementa esta función, que debe manipular la lista de forma abstracta, es decir, sin acceso a su representación interna.

El coste de la operación debe ser lineal con respecto al número de elementos en la lista. Justifica el coste de tu implementación.

Entrada

Cada caso de prueba está formado por tres líneas. La primera contendrá el carácter `N` si los elementos de la lista son números, o el carácter `P` si los elementos son palabras. La segunda línea contendrá un entero $L > 0$ que representa la longitud de la lista. La tercera línea contendrá los L elementos de la lista. Si son números estarán en el rango $[0..10^9]$. Si son palabras, estarán formadas por no más de 30 caracteres de la 'a' a la 'z'.

Salida

Para cada caso de prueba se escribirá una línea con los elementos de la lista después de eliminar los elementos repetidos.

Entrada de ejemplo

```
N
7
1 5 1 1 2 5 2
P
6
perro gato perro gato perro gata
P
1
uno
```

Salida de ejemplo

```
1 5 2
perro gato gata
uno
```

Ejercicio 3 (3 puntos)

Queremos implementar un TAD `GaleriaArte` que simule el comportamiento de una galería de arte simplificada. En una galería de arte se exponen los obras de diversos artistas. En cualquier momento la galería puede aceptar exponer nuevas obras, o puede vender alguna de las que tiene expuestas. Se hará uso de los tipos `artista` y `obra` que se representan con un `string`.

Las operaciones de la galería de arte son las siguientes:

- `nueva_obra(c,a,p)`: añade a la galería una nueva obra `c`, del artista `a`, con un precio `p`. Si la obra `c` ya estaba en la galería, se lanza una excepción del tipo `invalid_argument` con el mensaje `Obra ya en la galeria`.
- `venta_obra(c)`: simula la venta de la obra `c`, lo que incluye que se elimina de la galería de arte. Si la obra no está en la galería se lanzará una excepción del tipo `invalid_argument` con el mensaje `Obra no existente`.
- `mas_vendidos()`: devuelve un vector con todos los artistas cuyo total de ventas (suma de los precios de las obras que se han vendido de ese artista) es el máximo en la galería, ordenados por orden alfabético de menor a mayor.
- `obras_por_antiguedad(k)`: devuelve en un vector las *k* obras que más tiempo han permanecido en la galería sin venderse, ordenadas desde la que más tiempo lleva en la galería a la que menos. Si no hay *k* obras en la galería se devolverán todas ellas.

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.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea: el nombre de la operación seguido de sus argumentos. La palabra `FIN` en una línea indica el final de cada caso.

Los nombres de los artistas y obras de arte no tienen blancos.

Salida

Para cada caso de prueba se escribirán los datos que se piden. Las operaciones que generan salida son:

- `mas_vendidos`, escribe la cabecera: `Los artistas que mas han vendido son:` seguido del nombre de los artistas uno en cada línea.
- `obras_por_antiguedad`, escribe la cabecera `Obras mas antiguas en la galeria:`. En las líneas siguientes se escribe el nombre de las *k* obras que más tiempo llevan en la galería una por línea. Si no hay *k* obras en la galería se escribirán todas ellas.

Si alguna operación produce una excepción se mostrará el mensaje `ERROR:` seguido del mensaje de la excepción como resultado de la operación, y nada más.

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

Entrada de ejemplo

```
nueva_obra Autorretrato Goya 150
nueva_obra Las_meninas Velazquez 90
nueva_obra La_metamorfosis Dali 45
nueva_obra Fiesta_en_Figueras Dali 45
venta_obra La_metamorfosis
nueva_obra La_venus_del_espejo Velazquez 120
venta_obra Fiesta_en_Figueras
mas_vendidos
obras_por_antiguedad 2
venta_obra Las_meninas
mas_vendidos
nueva_obra Biarritz Sorolla 80
nueva_obra El_pie_herido Sorolla 50
venta_obra Autorretrato
mas_vendidos
nueva_obra La_rendicion_de_Breda Velazquez 130
nueva_obra Cosiendo_la_vela Sorolla 70
nueva_obra Aquelarre Goya 100
venta_obra El_pie_herido
nueva_obra La_metamorfosis Dali 90
obras_por_antiguedad 10
venta_obra La_metamorfosis
mas_vendidos
FIN
nueva_obra Autorretrato Goya 150
venta_obra Las_meninas
mas_vendidos
FIN
```

Salida de ejemplo

```
Los artistas que mas han vendido son:
Dali
Obras mas antiguas en la galeria:
Autorretrato
Las_meninas
Los artistas que mas han vendido son:
Dali
Velazquez
Los artistas que mas han vendido son:
Goya
Obras mas antiguas en la galeria:
La_venus_del_espejo
Biarritz
La_rendicion_de_Breda
Cosiendo_la_vela
Aquelarre
La_metamorfosis
Los artistas que mas han vendido son:
Dali
---
ERROR: Obra no existente
Los artistas que mas han vendido son:
---
```

Estructuras de datos

Grados de la Facultad de Informática (UCM)

16 de julio de 2021 (convocatoria extraordinaria)

Normas de realización del examen

1. El examen dura **3 horas**.
2. Debes desarrollar e implementar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección `http://exacrc`.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen.
4. Escribe tu **nombre y apellidos** en el espacio reservado para ello en los ficheros proporcionados.
5. Dispones de un fichero plantilla para cada ejercicio. Debes utilizarlo atendiendo a las instrucciones que se dan en cada fichero y escribiendo tu solución en los espacios reservados para ello, siempre entre las etiquetas `<answer>` y `</answer>`.
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. Si necesitas consultar la documentación de C++, está disponible en `http://exacrc/cppreference`.

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

si lees esto me debes un besito

Ejercicio 1. Tipos de datos lineales (2 puntos)

Supongamos una lista ordenada de números enteros. Este ejercicio consiste en implementar un método que sume una cantidad dada a uno de los nodos de la lista y cambie su posición de modo que la lista resultante siga estando ordenada.

Partimos de la clase `ListLinkedDouble`, que implementa el TAD lista de números enteros mediante listas doblemente enlazadas circulares con nodo fantasma, y un contador con el número de elementos. Suponiendo que la lista está ordenada y tiene N elementos, el método `add_to()` recibe una posición `pos`, donde $0 \leq$ `pos` $< N$ (las posiciones comienzan a numerarse desde 0). También recibe un número entero positivo m. El método suma m al valor del nodo situado en la posición `pos`, y reubica dicho nodo para que la lista siga estando ordenada.

Por ejemplo, supongamos la lista `xs = [2, 4, 6, 10, 14, 20, 25, 26, 30]`. Tras realizar la llamada `xs.add_to(3, 18)`, sumaríamos `18` al elemento en la posición número `3` (que es el `10`). Con ello obtendríamos `10 + 18 = 28`. El resultado de la llamada ha de ser, por tanto, `xs = [2, 4, 6, 14, 20, 25, 26, 28, 30]`.

Importante: Para la implementación del método no pueden crearse, directa o indirectamente, nuevos nodos mediante `new` ni borrar nodos mediante `delete`; han de reutilizarse los nodos de la lista de entrada. Tampoco se permite copiar valores de un nodo a otro.

Entrada

La entrada comienza con un número que indica el número de casos de prueba que vienen a continuación. Cada caso de prueba consiste en dos líneas. La primera línea contiene tres números N, i, m, tales que $0 \leq i < N$ y $m \geq 0$, donde i es la posición del nodo que se quiere modificar, y m el valor que quiere sumarse a dicho nodo. La segunda línea contiene N números enteros que son los elementos de la lista ordenada sobre la que se aplicará dicho método.

Salida

Para cada caso de prueba se imprimirá una línea con el contenido de la lista tras llamar al método `add_to()`.

Entrada de ejemplo

```
3
9 3 18
2 4 6 10 14 20 25 26 30
9 3 50
2 4 6 10 14 20 25 26 30
9 0 11
2 4 6 10 10 20 25 26 30
```

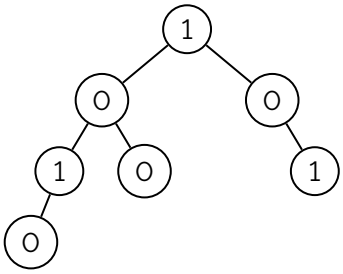
Salida de ejemplo

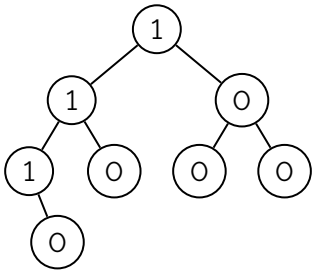
```
[2, 4, 6, 14, 20, 25, 26, 28, 30]
[2, 4, 6, 14, 20, 25, 26, 30, 60]
[4, 6, 10, 10, 13, 20, 25, 26, 30]
```

Ejercicio 2. Tipos de datos arborescentes (2 puntos)

Dado un árbol binario de ceros y unos, se dice que está *bien codificado* si es vacío o la diferencia entre la cantidad de ceros en el hijo izquierdo y en el hijo derecho no excede la unidad y, además, tanto el hijo izquierdo como el derecho están bien codificados. Dicho de otra forma, todos los nodos cumplen que la diferencia entre las cantidades de ceros en sus dos hijos es como mucho uno.

Por ejemplo, de los siguientes árboles, el de la izquierda no está bien codificado porque la raíz no cumple la condición al tener tres ceros en su hijo izquierdo y solamente uno en su hijo derecho (aunque el resto de nodos sí cumplen la condición). El de la derecha sí está bien codificado.





Dado un árbol binario queremos averiguar si está bien codificado o no. Se debe implementar una función *externa* a la clase `BinTree` que explore el árbol de manera eficiente averiguando si está bien codificado o no. La función no podrá tener parámetros de entrada/salida.

Entrada

La entrada comienza con un número que indica el número de casos de prueba que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario mediante la notación vista en clase. El árbol vacío se representa mediante `.` y el árbol no vacío mediante `(iz x dr)`, siendo `x` la raíz, e `iz` y `dr` las representaciones de ambos hijos.

Salida

Para cada árbol, se escribirá una línea con un `SI` si el árbol está bien codificado y un `NO` si no lo está.

Entrada de ejemplo

```
4
((((. 0 .) 1 .) 0 (. 0 .)) 1 (. 0 (. 1 .)))
(((. 1 (. 0 .)) 1 (. 0 .)) 1 ((. 0 .) 0 (. 0 .)))
.
(. 0 (. 1 (. 0 (. 1 (. 0 .)))))
```

Salida de ejemplo

```
NO
SI
SI
NO
```

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

El Grande es un juego de mesa ambientado en la España medieval. En este juego, los jugadores tienen que colocar estratégicamente a sus caballeros por las distintas regiones del país. Decimos que un jugador *predomina* en una región cuando tiene **estrictamente** más caballeros en dicha región que cualquiera de los restantes jugadores. Por otro lado, decimos que una región está *en disputa* si está ocupada por al menos un caballero, pero ningún jugador predomina en la misma. En particular, las regiones vacías (esto es, sin caballeros) *no* están en disputa.

En este ejercicio trataremos de implementar un TAD que almacene la información correspondiente a una partida de este juego. Las operaciones a implementar son:

- `anyadir_jugador(jugador)`. Añade un jugador a la partida. Si el `jugador` (de tipo `string`) ya estaba inscrito en ella, se lanza una excepción `domain_error` con el mensaje `Jugador existente`.
- `colocar_caballero(jugador, region)`. Indica que el `jugador` (de tipo `string`) coloca un caballero en la `region` (de tipo `string`) indicada. Si el `jugador` no se encuentra inscrito en la partida, se lanza la excepción `domain_error` con el mensaje `Jugador no existente`. Si la `region` no existe, se dará de alta.
- `puntuacion(jugador)`. Devuelve el número de regiones en las que predomina el `jugador` pasado como parámetro. Si el jugador no se encuentra inscrito en la partida, se lanza la excepción `domain_error` con el mensaje `Jugador no existente`.
- `regiones_en_disputa()`. Devuelve un `vector<string>` con la lista de regiones que están en disputa. La lista ha de estar ordenada ascendentemente, en orden alfabético según el nombre de la región.
- `expulsar_caballeros(region)`. Elimina a todos los caballeros de la `region` pasada como parámetro. Si la región no existe o no tiene ningún caballero, se lanza la excepción `domain_error` con el mensaje `Region vacia`.

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.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea: el nombre de la operación seguido de sus argumentos. La palabra `FIN` en una línea indica el final de cada caso.

Los nombres de jugadores y regiones son cadenas de caracteres sin espacios en blanco.

Salida

Las operaciones que generan salida son:

- `puntuacion J`, que debe escribir una línea con el mensaje `Puntuacion de J: X`, donde `J` es el jugador del cual quiere obtenerse la puntuación, y `X` es la puntuación del mismo.

- `regiones_en_disputa`, que debe escribir una línea con el mensaje `Regiones en disputa:`, seguida de los nombres de las regiones que están en disputa, uno por línea.

Si una operación produce un error, entonces se escribirá una línea con el mensaje `ERROR:`, seguido del mensaje de la excepción que lanza la operación, y no se escribirá nada más para esa operación.

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

Entrada de ejemplo

```
anyadir_jugador jug1
anyadir_jugador jug2
colocar_caballero jug1 Granada
colocar_caballero jug2 Granada
colocar_caballero jug1 Granada
puntuacion jug1
puntuacion jug2
colocar_caballero jug2 Granada
puntuacion jug1
puntuacion jug2
colocar_caballero jug2 Aragon
colocar_caballero jug1 Aragon
regiones_en_disputa
FIN
colocar_caballero jug1 Sevilla
anyadir_jugador jug1
colocar_caballero jug1 Valencia
colocar_caballero jug1 Galicia
puntuacion jug1
expulsar_caballeros Valencia
puntuacion jug1
expulsar_caballeros Valencia
FIN
```

Salida de ejemplo

```
Puntuacion de jug1: 1
Puntuacion de jug2: 0
Puntuacion de jug1: 0
Puntuacion de jug2: 0
Regiones en disputa:
Aragon
Granada
---
ERROR: Jugador no existente
Puntuacion de jug1: 2
Puntuacion de jug1: 1
ERROR: Region vacia
---
```

Estructuras de datos

Grados de la Facultad de Informática (UCM)

17 de junio de 2021

Normas de realización del examen

1. El examen dura **3 horas**.
2. Debes desarrollar e implementar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección `http://exacrc`.
3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen.
4. Escribe tu **nombre y apellidos** en el espacio reservado para ello en los ficheros proporcionados.
5. Dispones de un fichero plantilla para cada ejercicio. Debes utilizarlo atendiendo a las instrucciones que se dan en cada fichero y escribiendo tu solución en los espacios reservados para ello, siempre entre las etiquetas `<answer>` y `</answer>`.
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. Si necesitas consultar la documentación de C++, está disponible en `http://exacrc/cppreference`.

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

si lees esto me debes un besito

Ejercicio 1. Tipos de datos lineales (2 puntos)

Desentramar una lista consiste en separar, por un lado, los elementos que están en las posiciones pares y, por otro lado, los que están en posiciones impares. Por ejemplo, tras desentramar la lista `[10, 9, 3, 8, 2, 7, 5]` se obtienen como resultado las listas `[10, 3, 2, 5]` y `[9, 8, 7]`.

Partimos de la clase `ListLinkedDouble<T>`, que implementa el TAD lista mediante listas doblemente enlazadas circulares con nodo fantasma, y un contador con el número de elementos. Queremos añadir un nuevo método `void unzip(ListLinkedDouble & dest)`, que desentrama la lista `this`, quedándose esta última con los elementos situados en posiciones pares (suponemos que las posiciones se empiezan a numerar desde el 0), y moviendo los de las posiciones impares al final de la lista `dest` pasada como parámetro. Por ejemplo, dada la lista `xs = [10, 1, 20, 2, 30, 3]` y la lista `zs = []`, tras la llamada `xs.unzip(zs)` la lista `xs` tiene los valores `[10, 20, 30]` y la lista `zs` tiene los valores `[1, 2, 3]`.

Si, en el ejemplo anterior, la lista `zs` no estuviese vacía antes de la llamada a `unzip`, se añadirían los elementos de las posiciones impares de `xs` al *final* de la lista `zs`. Por ejemplo, si inicialmente tuviésemos `zs = [5, 0]`, tras hacer `xs.unzip(zs)` tendríamos `zs = [5, 0, 1, 2, 3]`.

Importante: Para la implementación del método no pueden crearse, directa o indirectamente, nuevos nodos mediante `new` ni borrar nodos mediante `delete`; han de reutilizarse los nodos de la lista de entrada. Tampoco se permite copiar valores de un nodo a otro. El coste de la operación ha de ser lineal con respecto al número de elementos de la lista `this`.

Entrada

La entrada comienza con un número que indica el número de casos de prueba que vienen a continuación. Cada caso de prueba consiste en dos líneas: la primera de ellas denota la lista `this`, y la segunda denota la lista `dest`. Cada una de las listas se representa mediante una secuencia con sus elementos (números enteros distintos de cero). Cada secuencia finaliza con `0`, que no forma parte de la lista.

Salida

Para cada caso de prueba se imprimirán dos líneas: una con el contenido de la lista `this` tras llamar al método `unzip` y otra con el contenido de la lista `dest` tras esa misma llamada. Para imprimir las listas puedes utilizar el método `display()`, o la sobrecarga del operador << que se proporciona para listas.

Entrada de ejemplo

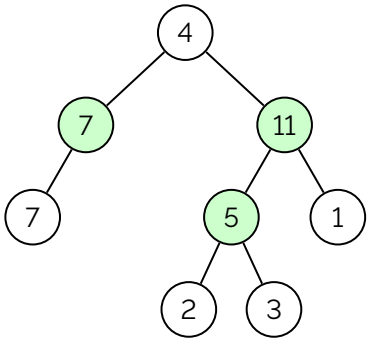
```
2
10 1 20 2 30 3 0
0
7 6 5 4 3 2 1 0
10 20 0
```

Salida de ejemplo

```
[10, 20, 30]
[1, 2, 3]
[7, 5, 3, 1]
[10, 20, 6, 4, 2]
```

Ejercicio 2. Tipos de datos arborescentes (2 puntos)

En un árbol binario de números enteros, decimos que un nodo es *cuadrado* si su valor es la suma de los valores de sus descendientes. Por ejemplo, dado el árbol mostrado en la figura, el nodo interno 7 es cuadrado, ya que la suma de sus descendientes (en este caso, solo hay uno) es 7. El nodo interno 5 también es cuadrado, ya que $5 = 2 + 3$. Por último, el nodo 11 también es cuadrado, ya que $11 = 5 + 2 + 3 + 1$. Los restantes nodos no son cuadrados.



En este ejercicio se pide:

1. Implementar una función `nodo_cuadrado_mayor_prof` con la siguiente cabecera:

   ```
   int nodo_cuadrado_mayor_prof(const BinTree<int> & tree);
   ```

   Esta función debe devolver el valor del nodo cuadrado dentro de `tree` que esté a mayor nivel de profundidad. Para el árbol mostrado anteriormente, el valor devuelto debe ser `5`. En caso de que varios nodos cuadrados se encuentren en un mismo nivel de profundidad, tendrá prioridad el que se encuentre más a la izquierda. Si el árbol no contiene ningún nodo cuadrado, la función debe devolver `-1`.

2. Indicar el coste, en el caso peor, de la función anterior. El coste debe estar expresado en función del número de nodos del árbol de entrada.

Entrada

La entrada comienza con un número que indica el número de casos de prueba que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario mediante la notación vista en clase. El árbol vacío se representa mediante `.` y el árbol no vacío mediante `(iz x dr)`, siendo `x` la raíz, e `iz` y `dr` las representaciones de ambos hijos.

Salida

Para cada árbol se escribirá el valor devuelto por la función `nodo_cuadrado_mayor_prof`.

Entrada de ejemplo

```
3
(((. 7 .) 7 .) 4 (((. 2 .) 5 (. 3 .)) 11 (. 1 .)))
((. 0 .) 4 ((. 2 .) 3 (. 1 .)))
((. 1 .) 3 .)
```

si lees esto me debes un besito

Salida de ejemplo

```
5
0
-1
```

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

La dirección del Hospital Central de Fanfanisflán quiere informatizar su consultorio médico por medio de un sistema que permita realizar al menos las siguientes operaciones:

- `altaMedico(medico)`: da de alta en el sistema a un nuevo médico (identificado por un `string`). Si el médico ya estaba dado de alta lanza una excepción `domain_error` con el mensaje `Medico existente`.
- `pedirConsulta(paciente, medico)`: registra que un paciente (identificado por un `string`) se pone a la espera para ser atendido por un médico. Si el médico no está dado de alta se lanza una excepción `domain_error` con el mensaje `Medico inexistente`. Un paciente puede estar esperando simultáneamente a varios médicos, pero es un error que un paciente pida consulta a un médico al que ya está esperando, lo que se indicará lanzando una excepción `domain_error` con el mensaje `Paciente ya esperando`.
- `siguientePaciente(medico)`: devuelve el paciente a quien le toca el turno, por ser el que más tiempo lleva esperando, para ser atendido por un médico, el cual debe estar dado de alta (si no se lanzará una excepción `domain_error` con el mensaje `Medico inexistente`) y debe tener algún paciente que le haya pedido consulta (si no se lanzará una excepción `domain_error` con el mensaje `Medico sin pacientes`).
- `atenderConsulta(medico)`: elimina de la lista de espera al paciente a quien le toca el turno, por ser el que más tiempo lleva esperando, para ser atendido por un médico, el cual debe estar dado de alta (si no se lanzará una excepción `domain_error` con el mensaje `Medico inexistente`) y debe tener algún paciente que le haya pedido consulta (si no se lanzará una excepción `domain_error` con el mensaje `Medico sin pacientes`).
- `abandonarConsulta(paciente)`: registra el hecho de que un paciente se ha cansado de esperar y abandona la consulta, dejando de estar en las listas de espera de los médicos para los que hubiera pedido consulta y aún no le hubieran atendido; si el paciente no estaba esperando a ningún médico se lanzará una excepción `domain_error` con el mensaje `Paciente inexistente`). La operación devuelve una lista ordenada alfabéticamente con los nombres de los médicos que dejan de tener que atender a este paciente.

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.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea: el nombre de la operación seguido de sus argumentos. La palabra `FIN` en una línea indica el final de cada caso.

Los nombres de los médicos y los pacientes son cadenas de caracteres sin espacios en blanco.

Salida

Para cada caso de prueba se escribirán los datos que se piden. Las operaciones que generan salida son:

- `siguientePaciente`, que debe escribir una línea con el nombre del siguiente paciente que será atendido por un médico.
- `abandonarConsulta`, que debe escribir una línea con el mensaje `Medicos abandonados:` seguido de los nombres de los médicos que han perdido a un paciente, en orden alfabético y separados por blancos.

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

Si una operación produce un error, entonces se escribirá una línea con `ERROR:`, seguido del error que devuelve la operación, y no se escribirá nada más para esa operación.

Entrada de ejemplo

```
altaMedico DrSanchez
altaMedico DraFuentes
pedirConsulta Pedro DraFuentes
pedirConsulta Luis DrSanchez
pedirConsulta Pedro DrSanchez
siguientePaciente DrSanchez
abandonarConsulta Pedro
siguientePaciente DraFuentes
FIN
atenderConsulta DrDolittle
altaMedico DrDolittle
altaMedico DrDolittle
pedirConsulta E.Murphy DrDolittle
pedirConsulta E.Murphy DrDolittle
siguientePaciente DrDolittle
FIN
```

Salida de ejemplo

```
Luis
Medicos abandonados: DrSanchez DraFuentes
ERROR: Medico sin pacientes
---
ERROR: Medico inexistente
ERROR: Medico existente
ERROR: Paciente ya esperando
E.Murphy
---
```

Estructuras de Datos

Grados de la Facultad de Informática (UCM)

Convocatoria extraordinaria, 17 de septiembre de 2020

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. 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).

5. 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.

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.

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)

La intersección de dos listas ordenadas y sin repeticiones es la lista ordenada que contiene los elementos que tienen ambas listas. Por ejemplo, la intersección de la lista `1, 3, 4, 5, 8, 9` con la lista `2, 4, 8, 10` es la lista `4, 8`.

Queremos extender *mediante herencia* la clase `deque<int>`, que implementa las colas dobles de enteros mediante listas de nodos dinámicos, doblemente enlazados, circular y con nodo fantasma, con un nuevo método `interseccion` que recibe como argumento una lista ordenada y modifica la lista de `this` para que termine teniendo la intersección de ambas listas.

En la implementación del nuevo método no pueden hacerse nuevos `new`s, ni *copiar* los enteros de un nodo a otro. También hará falta un método `print` para mostrar por pantalla los elementos de la lista, de inicio a fin.

No modifiques ni subas al juez el fichero `deque_eda.h` cuya clase `deque` debes extender.

Entrada

La entrada consta de una serie de casos de prueba. La primera línea contiene el número de casos de prueba que vendrán a continuación. Cada caso ocupa dos líneas. Cada una de estas líneas representa una de las listas, y contiene sus elementos ordenados de menor a mayor, una serie de números entre 1 y 1.000.000, seguidos de un 0, que marca el final de la descripción de la lista, sin pertenecer a ella. En cada lista, todos sus elementos son distintos.

Salida

Para cada caso de prueba se escribirá en una línea la lista modificada tras hacer la intersección de ambas listas.

Entrada de ejemplo

```
3
1 3 4 5 8 9 0
2 4 8 10 0
1 4 6 0
2 9 0
1 2 3 0
1 2 3 0
```

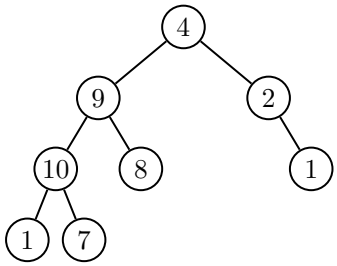
Salida de ejemplo

```
4 8

1 2 3
```

Ejercicio 2. Tipos de datos arborescentes (2 puntos)

Definimos un *camino* entre dos nodos de un árbol binario como una secuencia de nodos $n_1 n_2 \ldots n_k$ 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). La *suma* de un camino es la suma de los valores contenidos en los nodos que forman parte de él.

Por ejemplo, supongamos el siguiente árbol binario:



El camino formado por los nodos [1, 10, 9, 4, 2, 1] tiene como suma 27. Por otro lado, el camino formado por los nodos [7, 10, 9, 8] tiene como suma 34. De hecho, este último camino es el que tiene la *suma máxima*, en el sentido en el que no puede obtenerse otra suma mayor con otro camino distinto.

Se implementará una función externa a la clase `bintree` que explore el árbol calculando la suma máxima que puede obtenerse a partir de algún camino del árbol.

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 suma máxima de entre los caminos del árbol.

Entrada de ejemplo

```
3
4 9 10 1 -1 -1 7 -1 -1 8 -1 -1 2 -1 1 -1 -1
2 7 10 1 -1 -1 4 -1 -1 -1 5 -1 -1
2 1 -1 -1 -1
```

Salida de ejemplo

```
34
28
3
```

Ejercicio 3. Diccionarios (3 puntos)

Los aficionados al ciclismo tienen tres grandes acontecimientos en la temporada. En mayo se corre el Giro de Italia, en julio el Tour de Francia y en septiembre La Vuelta a España. Estas carreras se desarrollan por etapas. Al terminar cada etapa se muestra la clasificación de la etapa, que es el tiempo que ha tardado cada ciclista en hacer el recorrido de esa etapa. Por otro lado, tenemos la clasificación general de la carrera, que es el tiempo que ha tardado cada ciclista en recorrer todas las etapas ya transcurridas. Además de la clasificación de los ciclistas, es importante la clasificación de los equipos a los que estos pertenecen. Cada ciclista que participa en la carrera pertenece a un equipo. La clasificación general de los equipos se obtiene sumando el tiempo que han tardado los tres ciclistas del equipo que llegaron primero en cada etapa, siempre que hayan entrado en el tiempo máximo permitido para la misma. Si un equipo queda con menos de tres corredores deja de participar en esta clasificación.

Dependiendo de los kilómetros que se recorren en la etapa y del desnivel que tenga, la organización fija un tiempo máximo para recorrerla. Todos aquellos ciclistas que no llegan a la meta en este tiempo son descalificados y no vuelven a tomar la salida en las siguientes etapas. Por el contrario, suponemos que todos los ciclistas que no son descalificados en una etapa toman la salida en la etapa siguiente. Es decir, ningún ciclista abandona la competición por voluntad propia.

Dada la clasificación de una serie de etapas, la organización quiere que obtengamos tres resultados:

1. El tiempo total de cada equipo que sigue participando en la clasificación por equipos. El listado está ordenado según el orden alfabético de los equipos.

2. El nombre y el tiempo del primer ciclista en la clasificación general de ciclistas (el que menos tiempo acumulado lleva y en caso de empates, el que tiene un nombre menor alfabéticamente).

3. Para cada etapa, un listado de los ciclistas que han sido descalificados en la misma, ordenados de forma alfabética.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso comienza con una línea en que se indica el número de equipos que participan en la carrera. A continuación, para cada equipo se da su nombre seguido del número de ciclistas que ha inscrito en la carrera y del nombre de cada uno de estos ciclistas. Cuando termina la información de los equipos se muestra la clasificación de las etapas. Esta comienza con el número de etapas que vienen a continuación. Para cada etapa se muestra en una línea el número de ciclistas que toman la salida y el tiempo máximo permitido en esa etapa y en las líneas siguientes los nombres de los ciclistas y el tiempo que han tardado, ordenados según la clasificación de la etapa.

Los nombres de los equipos y de los ciclistas son cadenas de caracteres sin blancos. En una etapa nunca participa un ciclista que haya sido descalificado anteriormente.

Salida

Para cada caso de prueba se escriben los tres resultados descritos anteriormente, separados por líneas en blanco:

1. En primer lugar, la información relativa a equipos. Para cada uno de ellos se escribe una línea con su nombre seguido del tiempo total invertido por sus ciclistas. Este listado está ordenado alfabéticamente por nombre de equipo. Si no hubiera equipos con al menos tres ciclistas, se escribirá `CLASIFICACION VACIA`.

2. A continuación se muestra la información del primer ciclista clasificado: su nombre y el tiempo total invertido. En caso de empates en el tiempo mínimo, se muestra el que tenga un nombre alfabéticamente menor. Si todos los ciclistas han sido descalificados, se escribirá `NO HAY GANADOR`.

3. Por último se muestra la información de los ciclistas que han sido descalificados en cada etapa: en cada línea se mostrará el número de etapa seguido de los nombres de los ciclistas por orden alfabético y separados por blancos. Si en una etapa no hay ciclistas descalificados, aparecerá solamente el número de etapa en esa línea.

Al terminar la salida de un caso se escribirá una línea con tres guiones, `---`.

Entrada de ejemplo

```
3
Movistar 5  Valverde Mas Soler Cataldo Rojas
JumboVisma 5  Roglic Bennett Dumoulin Gesink Martin
Ineos 5  Bernal Amador Carapaz Castroviejo Rowe
2
15 50
Valverde 10  Roglic 20    Carapaz 20  Soler 20   Mas 30
Rojas 30     Bernal 30    Bennett 30  Rowe 30    Castroviejo 50
Amador 50    Dumoulin 60  Gesink 60   Martin 65  Cataldo 70
11 40
Castroviejo 5  Carapaz 15  Valverde 15  Roglic 15  Mas 20
Rojas 20       Bernal 20   Bennett 30   Rowe 30    Amador 50
Soler 55
1
Movistar 3 Valverde Mas Soler
2
3 14
Valverde 10   Mas 13   Soler 15
2 15
Mas 10   Valverde 11
```

Salida de ejemplo

```
Ineos 120
Movistar 115

Valverde 25

1 Cataldo Dumoulin Gesink Martin
2 Amador Soler
---
CLASIFICACION VACIA

Valverde 21

1 Soler
2
---
```

Estructuras de Datos y Algoritmos

Grados de la Facultad de Informática (UCM)

Junio 2020 — Tercera parte

En mi barrio hay una academia que ofrece cursos de chino mandarín estándar. La academia dispone de varios grupos, numerados del 1 al 6, que se corresponden con los distintos niveles de los exámenes estandarizados HSK, siendo el nivel 1 el más básico y el nivel 6 el más avanzado. Un estudiante puede comenzar matriculándose en un grupo de la academia, e ir avanzando progresivamente por los grupos superiores restantes. Cuando el estudiante supera el nivel 6, se le considera *graduado*.

La directoria de la academia quiere desarrollar un sistema para gestionar los estudiantes matriculados en sus cursos y aquellos que han finalizado sus estudios. Para ello necesita implementar un TAD `academia_chino` con las siguientes operaciones:

- `nuevo_estudiante(dni, grupo)`: Añade a la academia un estudiante con el `dni` dado (un `string`) y lo matricula en el `grupo` pasado como parámetro (un `int`). Si el estudiante ya estaba previamente en la academia (bien sea como estudiante actualmente matriculado o como estudiante graduado) se lanza una excepción `domain_error` con el mensaje `Estudiante existente`. Si el `grupo` dado no es un número de grupo válido, se lanza una excepción `domain_error` con el mensaje `Grupo incorrecto`.

- `promocionar(dni)`: Elimina al estudiante con el `dni` dado del grupo en el que esté matriculado y lo matricula en el grupo de nivel inmediatamente superior, salvo si el estudiante ya estuviese matriculado en el grupo 6, en cuyo caso el estudiante pasa a considerarse graduado. Si no existe ningún estudiante con el `dni` dado en la academia, se lanzará una excepción `domain_error` con el mensaje `Estudiante no existente`. Si el estudiante ya estaba graduado cuando se hace la llamada a `promocionar()`, se lanza una excepción `domain_error` con el mensaje `Estudiante ya graduado`.

- `grupo_estudiante(dni)`: Devuelve un `int` indicando el grupo en el que está matriculado el estudiante con el `dni` dado. Si el estudiante no existe en la academia se lanzará una excepción `domain_error` con el mensaje `Estudiante no existente`. Si el estudiante ya estaba graduado se lanza una excepción con el mensaje `Estudiante ya graduado`.

- `graduados()`: Devuelve una lista ordenada alfabéticamente de los DNIs de los estudiantes graduados en la academia.

- `novato(grupo)`: Devuelve el DNI de la persona que más recientemente se ha matriculado en el `grupo` dado (por ser nueva o por promoción) y que aún sigue matriculada en el mismo. Si el `grupo` es incorrecto, se lanzará una excepción `domain_error` con el mensaje `Grupo incorrecto`. Si el `grupo` está vacío, se lanzará una excepción `domain_error` con el mensaje `Grupo vacio` (sin tilde).

Requisitos de implementación:

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.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea: el nombre de la operación seguido de sus argumentos. La palabra `FIN` en una línea indica el final de cada caso.

Salida

Las operaciones `nuevo_estudiante` y `promocionar` no producen salida, salvo en caso de error. Con respecto a las restantes:

- Tras llamar a `grupo_estudiante` debe imprimirse una línea con el texto `XXX esta en el grupo N`, siendo `XXX` el DNI del estudiante consultado y `N` el grupo en el que está matriculado.
- Tras llamar a `graduados` debe imprimirse una línea con el texto `Lista de graduados:` seguida por los DNIs de los estudiantes graduados, separados por espacios.
- Tras llamar a `novato` debe imprimirse una línea con el texto `Novato de N: XXX`, donde `N` es el número de grupo consultado y `XXX` es el DNI resultante.

Cada caso termina con una línea con tres guiones (`---`). Si una operación produce un error, entonces se escribirá una línea con el mensaje `ERROR:`, seguido del error que devuelve la operación, y no se escribirá nada más para esa operación.

Entrada de ejemplo

```
nuevo_estudiante 123A 1
grupo_estudiante 123A
promocionar 123A
grupo_estudiante 123A
nuevo_estudiante 456B 2
novato 2
FIN
nuevo_estudiante 789C 6
nuevo_estudiante 123C 5
promocionar 123C
promocionar 789C
promocionar 123C
graduados
grupo_estudiante 123C
FIN
nuevo_estudiante 456D 6
nuevo_estudiante 456D 1
promocionar 456D
promocionar 456D
nuevo_estudiante 456D 1
promocionar 789E
FIN
```

Salida de ejemplo

```
123A esta en el grupo 1
123A esta en el grupo 2
Novato de 2: 456B
---
Lista de graduados: 123C 789C
ERROR: Estudiante ya graduado
---
ERROR: Estudiante existente
ERROR: Estudiante ya graduado
ERROR: Estudiante existente
ERROR: Estudiante no existente
---
```

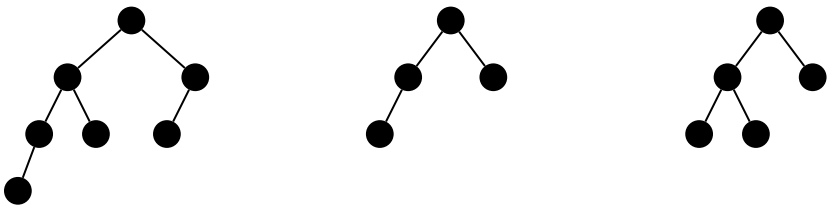
Estructuras de Datos y Algoritmos

Grados de la Facultad de Informática (UCM)

Junio 2020 — Segunda parte

Se dice que un árbol binario es *zurdo* si o bien es el árbol vacío o una hoja, o bien sus hijos izquierdo y derecho son ambos zurdos y más de la mitad de sus descendientes están en el hijo izquierdo.

Por ejemplo, de los siguientes árboles, el de la derecha no es zurdo (porque su hijo izquierdo no lo es) pero los otros dos sí.



Dado un árbol binario queremos averiguar si es zurdo o no.

Requisitos de implementación.

Se debe implementar una función *externa* a la clase `BinTree` que explore el árbol de manera eficiente averiguando si es zurdo o no. La función no podrá tener parámetros de entrada/salida.

Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en la descripción de un árbol binario de caracteres, donde todos los nodos contienen el carácter `*`.

Salida

Para cada árbol, se escribirá una línea con un `SI` si el árbol es zurdo y un `NO` si no lo es.

Entrada de ejemplo

```
4
****...*..**...
***...*..
***..*..*..
*.*..
```

Salida de ejemplo

```
SI
SI
NO
NO
```

Estructuras de Datos y Algoritmos

Grados de la Facultad de Informática (UCM)

Junio 2020 — Primera parte

Queremos extender la clase `ListLinkedDouble`, que implementa el TAD Lista mediante una lista doblemente enlazada circular con nodo fantasma, con un nuevo método

```
void partition(int pivot);
```

Este método recibe un entero `pivot` y reorganiza los nodos dinámicos de la lista enlazada de tal forma que al principio aparezcan aquellos que contengan enteros menores o iguales que el pivote y al final aparezcan los que contengan enteros mayores que el pivote.

Por ejemplo, si la lista es `l = [5, 10, 4, 7, 9, 3]` y se invoca el método `l.partition(8)`, la lista se transforma en `l = [5, 4, 7, 3, 10, 9]`. Es decir, al principio están todos los elementos menores o iguales a 8 seguidos de todos los elementos de la lista mayores que 8.

Observa que las posiciones relativas entre los elementos de las dos partes es la misma antes y después de la reorganización. Por ejemplo, como el `5` aparece antes que el `4` en la lista original y ambos son menores que el pivote, en la lista resultado aparecen al principio también así, el `5` antes que el `4`.

En la implementación del método no pueden hacerse nuevos `new`s, ni *copiar* los enteros de un nodo a otro.

Entrada

La entrada consta de una serie de casos de prueba. La primera línea contiene el número de casos de prueba que vendrán a continuación. Cada caso de prueba consiste en dos líneas. La primera contiene una lista, representada mediante una secuencia de números comprendidos entre 1 y 100.000, finalizando con un 0 que no se considera parte de la lista. La segunda línea contiene un número entero, que es el valor del parámetro `pivot` que debe utilizarse para esa lista.

Salida

Para cada caso de prueba se escribirán dos líneas con el resultado de la partición. La primera de ellas contendrá los elementos de la lista resultado de izquierda a derecha, es decir, desde el primero hasta el último (método `display()`). La segunda línea debe contener los elementos de la lista resultado en orden inverso, es decir, desde el último hasta el primero (método `display_reverse()`).

Entrada de ejemplo

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

Salida de ejemplo

```
[5, 4, 7, 3, 10, 9]
[9, 10, 3, 7, 4, 5]
[1, 2, 3, 4, 5, 6, 7]
[7, 6, 5, 4, 3, 2, 1]
[4, 3, 2, 1, 7, 6, 5]
[5, 6, 7, 1, 2, 3, 4]
```

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 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 formado `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
```

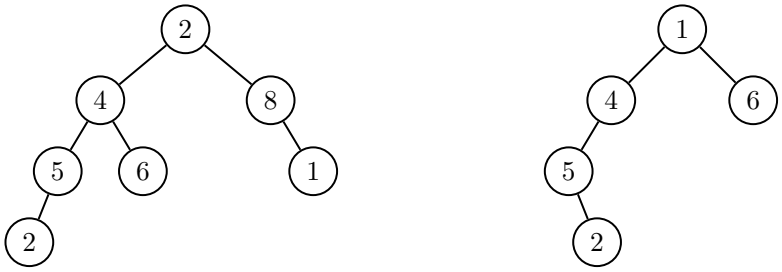
Salida de ejemplo

```
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 $n_1 n_2 \ldots n_k$ 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 $n_1 n_2 \ldots n_k$ 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

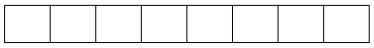
La salida de cada caso se escribirá en dos líneas. En la primera se muestra el máximo número de veces que se ha emitido una película seguido del título de la película. Si existen varias películas que se han emitido el mismo número máximo de veces, se muestra el título de la última emitida. En la segunda 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.

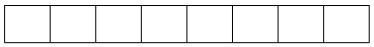
Entrada de ejemplo

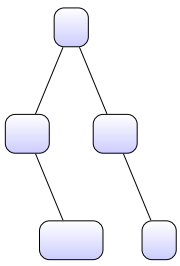
```
2
pelicula1 3
actor1 10 actor2 30 actor3 5
pelicula2 2
actor1 10 actor4 30
3
pelicula2 pelicula2 pelicula1
4
pelicula1 3
actor3 5 actor1 10 actor2 20
pelicula3 1
actor5 40
pelicula4 1
actor1 40
pelicula2 2
actor1 10 actor4 15
5
pelicula2 pelicula1 pelicula2 pelicula3 pelicula1
0
```

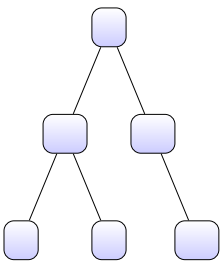
Salida de ejemplo

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









si lees esto me debes un besito

-
-
-
-
- →
-
- →
- →

si lees esto me debes un besito

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

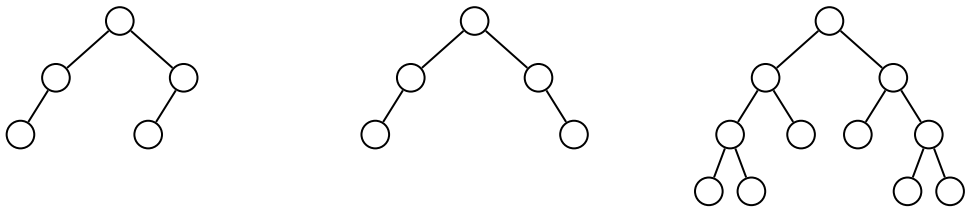
1. Debes programar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección `http://exacrc/domjudge/team`.
2. Escribe comentarios que expliquen tu solución, justifiquen por qué se ha hecho así y ayuden a entenderla. Calcula la complejidad de 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

Grados de la Facultad de Informática (UCM)

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.