Problema D

Ante el cambio de equipo continuo por parte de los jugadores, la Liga de Fútbol Profesional necesita un nuevo gestor de futbolistas, más eficiente, que le permita conocer el equipo actual que tiene fichado a cada jugador, o ciertos datos sobre el historial de fichajes.

La implementación del sistema se realizará mediante un TAD `gestor_futbolistas` con las siguientes operaciones:

- `fichar(jugador, equipo)`: si el `jugador` (un `string`) no está dado de alta en el sistema, entonces se registra por primera vez como perteneciente al `equipo` (otro `string`). Si el `equipo` no estaba dado de alta, se le da en ese momento. Si el `jugador` ya estaba dado de alta, la operación supone un cambio de equipo, pasando a estar fichado por el nuevo `equipo`. Si el `jugador` ya estaba fichado por este `equipo`, la operación no tiene ningún efecto.
- `equipo_actual(jugador)`: devuelve el equipo actual por el que está fichado este `jugador`. En caso de que el `jugador` no esté dado de alta, lanzará una excepción `domain_error` con mensaje `Jugador inexistente`.
- `fichados(equipo)`: devuelve cuántos jugadores tiene fichados actualmente el `equipo`. En caso de que el equipo no esté dado de alta, lanzará una excepción `domain_error` con mensaje `Equipo inexistente`.
- `ultimos_fichajes(equipo, n)`: devuelve una lista con los `n` últimos jugadores fichados por el `equipo` ($n > 0$) y que aún siguen estando fichados por ese `equipo`. La lista estará ordenada por el momento en el que fueron fichados, primero el último fichaje. En caso de que el `equipo` tenga menos de `n` jugadores, se devolverán todos, ordenados de la misma manera. En caso de que el equipo no esté dado de alta, lanzará una excepción `domain_error` con mensaje `Equipo inexistente`.
- `cuantos_equipos(jugador)`: devuelve el número de equipos distintos por los que ha estado fichado el `jugador`. Si el `jugador` no está dado de alta en el sistema, devolverá 0.

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. Los nombres de jugadores y equipos serán cadenas de caracteres sin espacios. La palabra `FIN` en una línea indica el final de cada caso.

Salida

La operación `fichar` no produce salida, salvo en caso de error. Con respecto a las restantes:

- Tras llamar a `equipo_actual` debe imprimirse una línea con el texto `El equipo de XXX es YYY`, siendo `XXX` el jugador consultado e `YYY` su equipo.
- Tras llamar a `fichados` debe imprimirse una línea con el texto `Jugadores fichados por XXX: N`, siendo `XXX` el nombre del equipo consultado y `N` su número de jugadores.
- Tras llamar a `ultimos_fichajes` debe imprimirse una línea con el texto `Ultimos fichajes de XXX:` seguida por los últimos jugadores fichados, separados por espacios, y siendo `XXX` el nombre del equipo consultado.

- Tras llamar a `cuantos_equipos` debe imprimirse una línea con el texto `Equipos que han fichado a XXX: N`, donde `XXX` es el jugador consultado y `N` el número de equipos por los que ha estado fichado.

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

```
fichar jugador1 equipo1
fichar jugador2 equipo1
ultimos_fichajes equipo1 3
fichar jugador2 equipo2
fichar jugador3 equipo1
ultimos_fichajes equipo1 2
fichar jugador1 equipo3
fichar jugador4 equipo1
cuantos_equipos jugador1
fichados equipo1
fichados equipo2
equipo_actual jugador4
fichar jugador5 equipo1
fichar jugador4 equipo1
ultimos_fichajes equipo1 8
FIN
fichados equipo2
cuantos_equipos jugador1
fichar jugador1 equipo1
fichar jugador1 equipo2
ultimos_fichajes equipo1 7
ultimos_fichajes equipo2 7
ultimos_fichajes equipo3 7
equipo_actual jugador10
FIN
```

Salida de ejemplo

```
Ultimos fichajes de equipo1: jugador2 jugador1
Ultimos fichajes de equipo1: jugador3 jugador1
Equipos que han fichado a jugador1: 2
Jugadores fichados por equipo1: 2
Jugadores fichados por equipo2: 1
El equipo de jugador4 es equipo1
Ultimos fichajes de equipo1: jugador5 jugador4 jugador3
---
ERROR: Equipo inexistente
Equipos que han fichado a jugador1: 0
Ultimos fichajes de equipo1:
Ultimos fichajes de equipo2: jugador1
ERROR: Equipo inexistente
ERROR: Jugador inexistente
---
```

Problema C

El ministerio de trabajo desea agilizar los trámites realizados por las oficinas de empleo. Nos piden diseñar un sistema, en el cual las personas se apuntan al paro indicando aquellos empleos para los que están cualificadas. Cuando se recibe en la oficina una oferta de empleo, se selecciona a la persona cualificada para ese empleo que lleva más tiempo apuntada en la oficina.

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

- `altaOficina(nombre, empleo)`: añade una persona (un `string`) a un empleo (`string`) para el que está cualificada. Si el empleo no está dado de alta en la aplicación, se le dará en este momento. Si la persona no está apuntada en la oficina se dará de alta con el empleo para el que está cualificada. Si la persona ya está apuntada en la oficina se añadirá a su solicitud el nuevo empleo, si no estaba apuntada para él. Si ya estuviese apuntada para el empleo que solicita no se la volverá a apuntar.

- `ofertaEmpleo(empleo)`: La operación devuelve la persona cualificada para realizar este empleo que más tiempo lleva apuntada en la oficina. La persona dejará de estar apuntada en la oficina de empleo. Si un empleo solo tenía a esta persona apuntada dejará de estar en la aplicación. Si el empleo no está en la aplicación lanza una excepción `domain_error` con mensaje `No existen personas apuntadas a este empleo`.

- `listadoEmpleos(persona)`: Obtiene un vector con los empleos para los que está cualificada una persona en orden alfabético. En caso de que la persona no exista, lanza una excepción `domain_error` con mensaje `Persona inexistente`.

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.

Los nombres de las personas y las descripciones de los empleos son cadenas de caracteres sin blancos.

Salida

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

- `ofertaEmpleo`, que debe escribir una línea con el empleo que se ofrece seguido del nombre de la persona seleccionada.

- `listadoEmpleos`, que debe escribir una línea con el nombre de la persona seguido de todos los empleos para los que está cualificada la persona, 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

```
altaOficina Carlota administracion
altaOficina Carlota secretariado
altaOficina Carlota recursosHumanos
altaOficina Juan administracion
altaOficina Juan riesgosLaborales
ofertaEmpleo administracion
altaOficina Beatriz administracion
altaOficina Beatriz jardineria
altaOficina Beatriz biblioteca
ofertaEmpleo administracion
listadoEmpleos Carlota
altaOficina Carlota informatica
altaOficina Beatriz informatica
listadoEmpleos Beatriz
ofertaEmpleo informatica
ofertaEmpleo informatica
ofertaEmpleo informatica
ofertaEmpleo biblioteca
FIN
ofertaEmpleo informatica
altaOficina Ivan informatica
altaOficina Ivan administracion
listadoEmpleos Ivan
ofertaEmpleo informatica
ofertaEmpleo administracion
listadoEmpleos Ivan
FIN
```

Salida de ejemplo

```
administracion: Carlota
administracion: Juan
ERROR: Persona inexistente
Beatriz: administracion biblioteca informatica jardineria
informatica: Carlota
informatica: Beatriz
ERROR: No existen personas apuntadas a este empleo
ERROR: No existen personas apuntadas a este empleo
---
ERROR: No existen personas apuntadas a este empleo
Ivan: administracion informatica
informatica: Ivan
ERROR: No existen personas apuntadas a este empleo
ERROR: Persona inexistente
---
```

Problema B

Queremos gestionar la sala de espera del servicio de urgencias de un hospital muy especial, donde los pacientes suelen mejorar solo por esperar.

La implementación del sistema se realizará mediante un TAD `urgencias` con las siguientes operaciones:

- `nuevo_paciente(paciente, gravedad)`: registra a un nuevo `paciente` (un `string`), que ya ha sido preevaluado por un médico que le ha asignado cierta gravedad (un `int`): leve (1), media (2) o grave (3). El `paciente` pasa a la sala de espera. Si el `paciente` ya estaba registrado, lanzará una excepción `domain_error` con mensaje `Paciente repetido`. Si la `gravedad` dada no es un número entre 1 y 3, se lanzará una excepción `domain_error` con el mensaje `Gravedad incorrecta`.
- `gravedad_actual(paciente)`: devuelve el entero que representa la gravedad actual de `paciente`. Si el paciente no está en la sala de espera, se lanzará una excepción `domain_error` con mensaje `Paciente inexistente`.
- `siguiente()`: devuelve el nombre del paciente al que le toca ser atendido, que abandona la sala de espera. Los pacientes se atienden teniendo en cuenta su gravedad: primero los graves, luego los de gravedad media y por último los leves. Dentro de la misma gravedad se tiene en cuenta el orden de llegada (el primero que llega es el más prioritario), o el que provoca la operación `mejora` de cambio de gravedad, explicada a continuación. Si no hay pacientes se lanzará una excepción `domain_error` con mensaje `No hay pacientes`.
- `mejora(paciente)`: registra el hecho de que un paciente mejora estando en la sala de espera. Si estaba grave pasa a gravedad media y si tenía gravedad media pasa a leve. Para el orden de atención se coloca como el más prioritario de los que tienen la nueva gravedad. Si el paciente estaba leve, entonces se recupera y abandona las urgencias. Si el paciente no existe, se lanzará una excepción `domain_error` con mensaje `Paciente inexistente`.
- `recuperados()`: devuelve una lista ordenada alfabéticamente (y sin repeticiones) con los pacientes que han pasado alguna vez por el servicio de urgencias y se han recuperado del todo mientras esperaban.

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. Los nombres de los pacientes serán cadenas de caracteres sin espacios. La palabra `FIN` en una línea indica el final de cada caso.

Salida

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

- Tras llamar a `gravedad_actual` debe imprimirse una línea con el texto `La gravedad de XXX es N`, siendo `XXX` el paciente consultado y `N` su gravedad (1, 2 o 3).
- Tras llamar a `recuperados` debe imprimirse una línea con el texto `Lista de recuperados:` seguida por los nombres de los pacientes recuperados, separados por espacios.

- Tras llamar a `siguiente` debe imprimirse una línea con el texto `Siguiente paciente: XXX`, donde `XXX` es el paciente al que le toca ser atendido.

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_paciente Luis 2
nuevo_paciente Ana 3
nuevo_paciente Eva 3
nuevo_paciente Ivan 2
mejora Ana
siguiente
siguiente
siguiente
gravedad_actual Ivan
mejora Ivan
recuperados
mejora Ivan
nuevo_paciente Luz 1
mejora Luz
recuperados
FIN
nuevo_paciente Pedro 1
nuevo_paciente Pedro 2
mejora Pedro
recuperados
nuevo_paciente Pedro 3
mejora Pedro
gravedad_actual Pedro
siguiente
gravedad_actual Pedro
nuevo_paciente Pedro 4
siguiente
FIN
```

Salida de ejemplo

```
Siguiente paciente: Eva
Siguiente paciente: Ana
Siguiente paciente: Luis
La gravedad de Ivan es 2
Lista de recuperados:
Lista de recuperados: Ivan Luz
---
ERROR: Paciente repetido
Lista de recuperados: Pedro
La gravedad de Pedro es 2
Siguiente paciente: Pedro
ERROR: Paciente inexistente
ERROR: Gravedad incorrecta
ERROR: No hay pacientes
---
```

Problema A

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
```

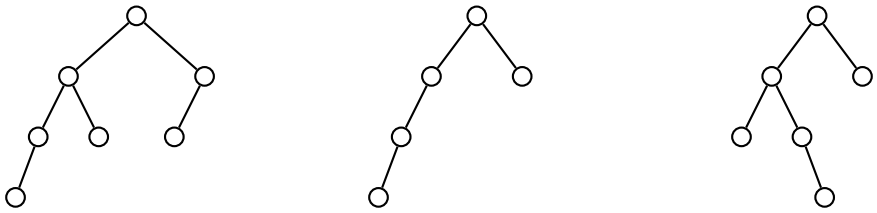
Salida de ejemplo

```
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
---
```

Problema E

Se dice que un árbol binario está *inclinado a la izquierda* si o bien es el árbol vacío o una hoja, o bien la altura de su hijo izquierdo es mayor que la altura de su hijo derecho y ambos hijos están también inclinados a la izquierda.

Por ejemplo, de los siguientes árboles, los dos primeros sí están inclinados a la izquierda, pero el tercero no, ya que su hijo izquierdo no lo está.



Dado un árbol binario queremos averiguar si está inclinado a la izquierda o no.

Requisitos de implementación.

Se debe implementar una función *externa* a la clase `bintree` que explore el árbol de manera eficiente averiguando si está inclinado a la izquierda 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 una cadena de caracteres con la descripción de un árbol binario: el árbol vacío se representa con un '`.`'; un árbol no vacío se representa con un '`*`' (que denota la raíz), seguido primero de la descripción del hijo izquierdo y después de la descripción del hijo derecho.

Salida

Para cada árbol, se escribirá una línea con un `SI` si el árbol está inclinado a la izquierda y un `NO` si no lo está.

Entrada de ejemplo

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

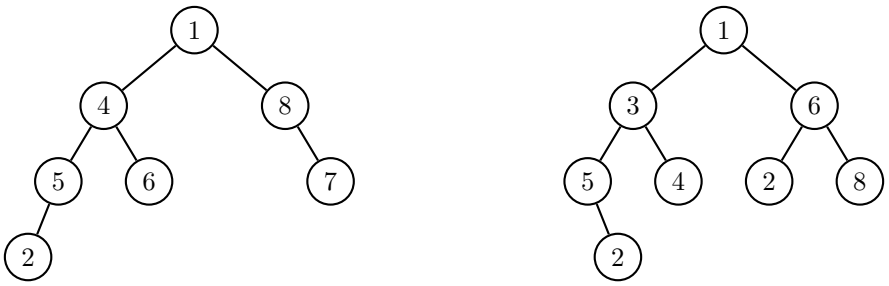
Salida de ejemplo

```
SI
SI
NO
NO
```

Problema D

Dado un árbol binario de enteros, se dice que está *pareado* si la diferencia entre la cantidad de números pares en el hijo izquierdo y en el hijo derecho no excede la unidad y, además, tanto el hijo izquierdo como el derecho están pareados. Dicho de otra forma, todos los nodos cumplen que la diferencia entre las cantidades de números pares en sus dos hijos es como mucho uno.

Por ejemplo, de los siguienes árboles, el de la izquierda no está pareado porque la raíz no cumple la condición, al tener tres pares 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 pareado.



Dado un árbol binario queremos averiguar si está pareado o no.

Requisitos de implementación.

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

Entrada

La entrada comienza con el número de casos que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario: si el árbol es vacío se representa con un –1; si no, primero aparece su raíz, y a continuación la descripción del hijo izquierdo y después la del hijo derecho, dadas de la misma manera.

Salida

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

Entrada de ejemplo

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

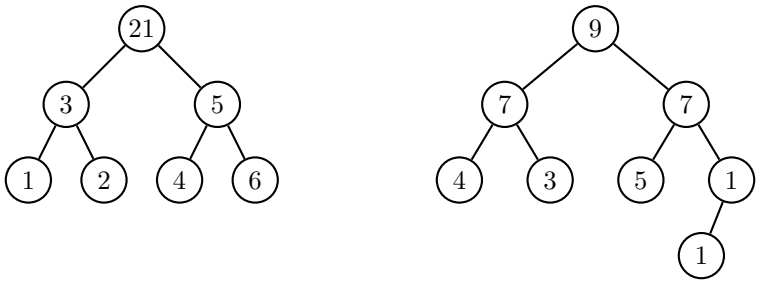
Salida de ejemplo

```
NO
SI
SI
NO
```

Problema C

En un árbol binario con números enteros en sus nodos, llamamos *nodo acumulador* a aquel cuyo valor sea igual a la suma de los valores de todos sus descendientes.

Por ejemplo, de los siguientes árboles el de la izquierda tiene dos nodos acumuladores (el 3 y el 21) y el de la derecha tiene tres nodos acumuladores (los dos 7 y uno de los 1, el nodo interno).



Requisitos de implementación.

Se debe implementar una función *externa* a la clase `bintree` que explore el árbol de manera eficiente calculando el número de nodos acumuladores que contiene. No se pueden utilizar parámetros de entrada/salida.

Entrada

La entrada comienza con el número de casos que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario: si el árbol es vacío se representa con un –1; si no, primero aparece su raíz, y a continuación la descripción del hijo izquierdo y después la del hijo derecho, dadas de la misma manera.

Salida

Para cada árbol se escribirá el número de nodos acumuladores que contiene en una línea.

Entrada de ejemplo

```
4
21 3 1 -1 -1 2 -1 -1 5 4 -1 -1 6 -1 -1
9 7 4 -1 -1 3 -1 -1 7 5 -1 -1 1 1 -1 -1 -1
-1
0 -1 0 -1 0 -1 0 -1 -1
```

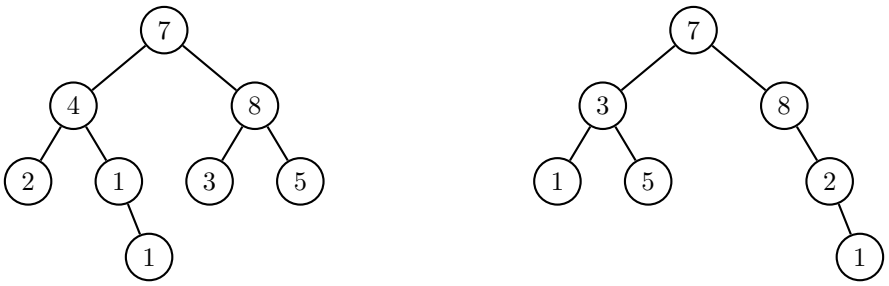
Salida de ejemplo

```
2
3
0
4
```

Problema B

Dado un árbol binario de enteros, se dice que es *diestro* si o bien es el árbol vacío o una hoja, o bien la suma de los valores en todos los nodos del hijo derecho es mayor que la suma de los valores de todos los nodos de su hijo izquierdo y, además, tanto el hijo izquierdo como el derecho son diestros.

Por ejemplo, de los siguienes árboles, el de la izquierda no es diestro porque el subárbol con raíz 4 no lo es (los descendientes en sus dos hijos suman lo mismo). El de la derecha sí es diestro (todos los nodos cumplen que la suma de los descendientes en su hijo derecho es mayor que la suma de los descendientes en su hijo izquierdo).



Dado un árbol binario queremos averiguar si es diestro 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 diestro o no. La función no podrá tener parámetros de entrada/salida.

Entrada

La entrada comienza con el número de casos que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario: si el árbol es vacío se representa con un –1; si no, primero aparece su raíz, y a continuación la descripción del hijo izquierdo y después la del hijo derecho, dadas de la misma manera.

Salida

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

Entrada de ejemplo

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

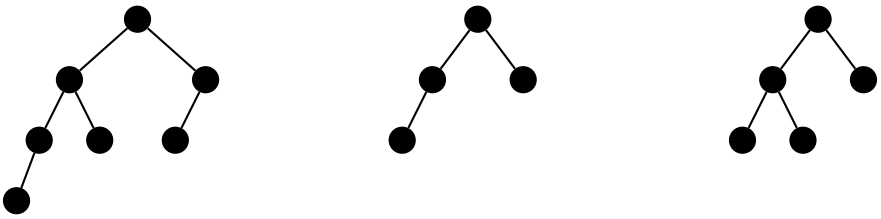
Salida de ejemplo

```
NO
SI
SI
NO
```

Problema A

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 una cadena de caracteres con la descripción de un árbol binario: el árbol vacío se representa con un '`.`'; un árbol no vacío se representa con un '`*`' (que denota la raíz), seguido primero de la descripción del hijo izquierdo y después de la descripción del hijo derecho.

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
```