Ejercicio 1

2 puntos

Dada una lista doblemente enlazada circular y con nodo fantasma de caracteres alfabéticos, se pide eliminar de la lista aquellas letras consecutivas que están repetidas, de forma que en la lista resultante no queden dos letras iguales seguidas. Las letras que se eliminan de la lista original se deben guardar en otra lista, en el orden en que se han eliminado.

Para implementar el ejercicio debe añadirse el siguiente método a la clase `ListLinkedDouble`:

```
ListLinkedDouble quitar_duplicados();
```

El método `quitar_duplicados` debe recorrer y eliminar los caracteres iguales seguidos. El método devolverá la lista con las letras eliminadas de la lista original, guardadas en el orden en que fueron eliminadas. Por ejemplo, si `xs` denota la lista `[a, a, b, b, c]`, tras la llamada `xs.quitar_duplicados()`, la lista `xs` pasa a ser `[a, b, c]`, y la lista devuelta por la llamada es `[a, b]`.

Indica y justifica el coste de `quitar_duplicados` en función del tamaño de la lista de entrada.

Importante: en la implementación de `quitar_duplicados` no está permitido crear, directa o indirectamente, nuevos nodos mediante `new`, ni liberar nodos mediante `delete`. La única excepción es la creación del nodo fantasma (realizada por el constructor de `ListLinkedDouble`) de la lista devuelta por `quitar_duplicados`.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso se muestra en dos líneas. En la primera se muestra el número de elementos de la lista, $n \geq 0$. En la línea siguiente se muestran todos los valores de la lista, caracteres del alfabeto anglosajón. Los valores se añaden a la lista por la derecha. La entrada termina con el valor `-1`.

Salida

Para cada caso de prueba se escriben dos líneas. En la primera se muestra el contenido de la lista después de eliminar los caracteres consecutivos iguales. En la segunda línea se muestran los caracteres eliminados, en el orden en que fueron eliminados.

Entrada de ejemplo

```
5
a a b b c
4
u u u u
6
a b a b a b
6
a a g h b b
7
o p o r r r y
-1
```

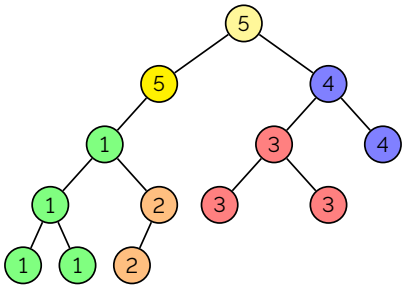
Salida de ejemplo

```
[a, b, c]
[a, b]
[u]
[u, u, u]
[a, b, a, b, a, b]
[]
[a, g, h, b]
[a, b]
[o, p, o, r, y]
[r, r]
```

Ejercicio 2

2 puntos

En este ejercicio consideramos árboles binarios de enteros, en el que cada entero identifica un tipo de sección determinado. Una sección es un conjunto de nodos con el mismo identificador conectados entre sí por caminos que no pasan por nodos con distintos identificadores. En el árbol de la siguiente figura, podemos distinguir 5 secciones diferentes, señaladas con distintos colores. Dos nodos tienen el mismo color si y solo si pertenecen a la misma sección:



Como puede verse, hay cinco secciones: la sección verde tiene cuatro nodos, la sección roja tiene tres nodos, y las tres restantes (azul, naranja y amarilla) tienen dos nodos cada una. Por tanto, la más grande es la verde. El objetivo de este ejercicio es, dado un árbol de enteros, determinar el tamaño (es decir, el número de nodos), de la sección más grande.

En este ejercicio se pide:

1. Definir una función `seccion_mayor` con la siguiente cabecera:

   ```
   int seccion_mayor(const BinTree<int> &tree)
   ```

   Esta función debe devolver el tamaño de la sección más grande del árbol de enteros pasado como parámetro. Por ejemplo, para el árbol mostrado en la figura anterior, la función debe devolver el valor 4.

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. Indica también la recurrencia utilizada en el caso de llamadas recursivas.

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 de enteros, Cada nodo tiene un entero, indicando la sección a la que pertenece.

Salida

Para cada árbol se escribirá una línea con el tamaño de la sección más grande.

Entrada de ejemplo

```
4
(((((. 1 .) 1 (. 1 .)) 1 ((. 2 .) 2 .)) 5 .) 5 (((. 3 .) 3 (. 3 .)) 4 (. 4 .)))
((. 1 .) 2 (. 3 .))
((. 2 (. 2 (. 2 .))) 2 (. 1 (. 1 (. 3 .))))
(((. 2 .) 1 (. 2 .)) 1 (. 2 .))
```

Salida de ejemplo

```
4
1
4
2
```

Ejercicio 3

3 puntos

Queremos gestionar un sistema de reservas en un camping. El camping se compone de una serie de parcelas colocadas en fila, una a continuación de la anterior. Las parcelas están numeradas de modo que la parcela número 0 se encuentra a la entrada del camping, a su derecha está la parcela 1, a continuación la parcela 2, y así sucesivamente. Por otro lado, si caminamos hacia la izquierda desde la parcela número 0 nos encontramos con las parcelas -1, -2, -3, etc.

Cada cliente puede reservar una parcela para un día determinado. Para simplificar, representamos los días como números naturales, comenzando por el cero, día de la inauguración del camping. Cuando un cliente hace una reserva, si la parcela indicada está disponible en el día dado, la reserva se hace efectiva y el cliente se convierte en el *titular* de esa parcela para ese día. Por el contrario, si la parcela ya estaba ocupada para ese día, se le añade a una lista de espera. Cada par *(parcela, día)* tiene su lista de espera propia.

Los dueños del camping son muy maniáticos con las reservas y no dejan a una persona reservar dos parcelas, aunque sea en días distintos. Además, tampoco permiten que una persona reserve si ya está en alguna lista de espera.

Se pide implementar las siguientes operaciones:

- `void nueva_reserva(const string &persona, int parcela, int dia)`

  Formaliza la reserva de la `parcela` indicada en el `dia` pasado como parámetro. La reserva se hará a nombre de la `persona` pasada como primer parámetro. Si la parcela ya estaba reservada ese día, se añade a la persona al final de la lista de espera correspondiente a esa parcela y día. Si la `persona` ya era titular de una reserva, o bien está en alguna lista de espera, la operación lanza la excepción `domain_error("Persona ya ha reservado")`.

- `void cancelar_reserva(const string &persona)`

  Cancela la reserva de la `persona` indicada, suponiendo que es titular de alguna parcela del camping en un día determinado. En ese caso, si hay otras personas en la lista de espera correspondiente a ese par *(parcela, día)*, la primera de ellas pasa a ser el nuevo titular de esa reserva. Si no, la parcela vuelve a quedar libre para ese día. La operación lanza la excepción `domain_error("No es titular de ninguna reserva")` si la `persona` no es titular de ninguna parcela.

- `string quien_reserva(int parcela, int dia) const`

  Devuelve el nombre del titular de la `parcela` dada en el `dia` pasado como parámetro. Si la `parcela` no está reservada por nadie para ese `dia`, la operación lanza una excepción `domain_error("Parcela no reservada")`.

- `bool puede_extender_reserva(const string &persona, int n) const`

  Suponiendo que la `persona` pasada como parámetro es titular de una parcela en un determinado día, devuelve `true` si la parcela que ha reservado también estaría disponible en los `n` días siguientes al día de la reserva, o `false` en caso contrario. Si la persona no es titular de ninguna parcela, se lanza la excepción `domain_error("No es titular de ninguna reserva")`. Esta operación no modifica ni extiende ninguna reserva; solo comprueba si sería posible extender la reserva de una persona.

- `int menor_distancia_vecinos(const string &persona) const`

  Suponiendo que la `persona` pasada como parámetro es titular de una parcela en un determinado día, indica la distancia que hay desde la parcela reservada por esa persona hasta la parcela más

cercana que esté reservada durante ese día. Definimos la *distancia* entre dos parcelas como el número de parcelas intermedias que hay entre ellas. En particular, si dos parcelas son contiguas, la distancia entre ellas es 0. Si no hay ninguna otra parcela reservada durante ese día (aparte de la reservada por la `persona` pasada como parámetro) se devuelve `-1`. Si la `persona` no es titular de ninguna `parcela`, se lanza la excepción `domain_error("No es titular de ninguna reserva")`.

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 personas son cadenas de caracteres alfanuméricos sin espacios.

Salida

Tras llamar a las operaciones `nueva_reserva` y `cancelar_reserva` se debe imprimir una línea con el texto `OK`, salvo en caso de error. Con respecto a las restantes:

- `quien_reserva`: se debe escribir el texto `(X, Y) reservada por P`, donde `X` es el número de parcela, `Y` es el número de día y `P` es el nombre devuelto por la operación.
- `puede_extender_reserva`: se debe escribir el texto `P puede extender la reserva N dias` o bien `P no puede extender la reserva N dias` según el resultado de la operación, donde `P` es el nombre de la persona y `N` es el número de días pasado como parámetro.
- `menor_distancia_vecinos`: se debe escribir el número devuelto por el método, o `INFINITO` si ese número es `-1`.

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.

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

Entrada de ejemplo

```
nueva_reserva lucia 2 3
nueva_reserva raul 2 3
quien_reserva 2 3
cancelar_reserva lucia
quien_reserva 2 3
nueva_reserva gumersindo 2 6
puede_extender_reserva raul 2
puede_extender_reserva raul 3
menor_distancia_vecinos raul
nueva_reserva gloria 10 3
menor_distancia_vecinos raul
nueva_reserva florencia -1 3
menor_distancia_vecinos raul
FIN
nueva_reserva lucia 1 6
nueva_reserva raul 1 6
nueva_reserva lucia 2 4
nueva_reserva raul 3 5
quien_reserva 2 4
cancelar_reserva raul
puede_extender_reserva raul 2
menor_distancia_vecinos raul
FIN
```

Salida de ejemplo

```
OK
OK
(2, 3) reservada por lucia
OK
(2, 3) reservada por raul
OK
raul puede extender la reserva 2 dias
raul no puede extender la reserva 3 dias
INFINITO
OK
7
OK
2
---
OK
OK
ERROR: Persona ya ha reservado
ERROR: Persona ya ha reservado
ERROR: Parcela no reservada
ERROR: No es titular de ninguna reserva
ERROR: No es titular de ninguna reserva
ERROR: No es titular de ninguna reserva
---
```