Estructuras de datos

Grados de la Facultad de Informática (UCM). Grupos A, C, E, F, DG

Examen Convocatoria Extraordinaria, 16 de julio de 2021.

Normas de realización del examen

1. El examen dura 3 horas.
2. Debes programar soluciones para cada uno de los tres ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección `http://exacrc`. Para la entrega el juez sólo tiene los datos de prueba del enunciado del problema.
3. 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.
4. 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.
5. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que subas al juez.
6. Dispones de un fichero plantilla para cada ejercicio.
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`.

Ejercicio 1. Sumar a un nodo de una lista ordenada

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

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. 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.

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

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
```

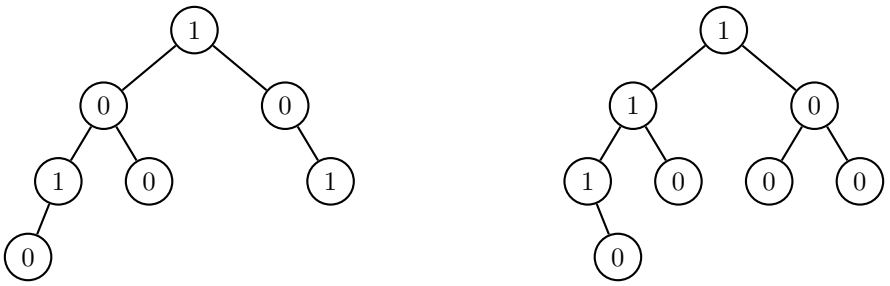
Salida de ejemplo

```
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. Árboles bien codificados

(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.

Requisitos de implementación.

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 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 bien codificado y un `NO` si no lo está.

Entrada de ejemplo

```
4
1 0 1 0 -1 -1 -1 0 -1 -1 0 -1 1 -1 -1
1 1 1 -1 0 -1 -1 0 -1 -1 0 0 -1 -1 0 -1 -1
-1
0 -1 1 -1 0 -1 1 -1 0 -1 -1
```

Salida de ejemplo

```
NO
SI
SI
NO
```

Ejercicio 3. El Grande

(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
---
```