Estructuras de Datos y Algoritmos

Grados en Videojuegos

Examen Final, 9 de Febrero de 2017. Tipo A.

Normas de realización del examen

1. Debes programar soluciones para cada uno de los dos ejercicios, probarlas y subirlas al juez. El juez se utiliza exclusivamente para la entrega del ejercicio, y no realiza ninguna prueba adicional a los casos del ejemplo. Por lo tanto el veredicto del juez no es significativo y no debe tomarse como que el programa es correcto.

2. Para que los problemas estén presentados deben estar subidos al juez, aunque no estén correctos. La dirección es `http://exacrc/domjudge/team`. Debe escribirse la dirección completa.

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, el laboratorio y el puesto en el que estás y el usuario que vas a utilizar en el juez de examen en la primera línea de cada fichero que subas al juez.

5. Puedes descargar el fichero `http://exacrc/DocumentacionTipoA.zip` que contiene material que puedes utilizar para la realización del examen (implementación de las estructuras de datos que necesitas, plantilla de código fuente y ficheros de texto con los casos de prueba de cada ejercicio del enunciado).

6. Para obtener la máxima puntuación, las soluciones deberán seguir los criterios exigidos a los ejercicios durante el curso (en cuanto a eficiencia, simplicidad, buena programación etc.)

7. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta exclusivamente el último envio que hayas realizado del ejercicio.

8. Las notas de los ejercicios suman 5.2 puntos. La calificación obtenida en esta parte del examen será sumada a la obtenida en las preguntas cortas (1.8 puntos) y a la de evaluación continua (3 puntos) para obtener la calificación final de la asignatura.

Problema 1

(2 puntos) Diremos que un árbol binario es *zurdo* si el número de nodos de su lado izquierdo es mayor estrictamente que el número de nodos de su lado derecho y tanto su hijo izquierdo como su hijo derecho son también zurdos.

Implementa una función que dado un árbol binario decida si es zurdo.

Entrada

La entrada comienza con el número de casos de prueba. Para cada caso se muestra el recorrido en preorden del árbol incluyendo los punteros a nulo en una línea.

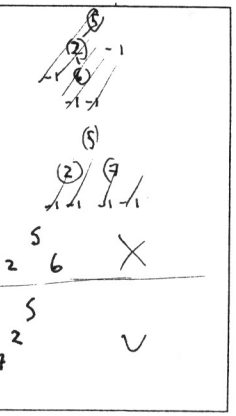
Los valores de los nodos serán números del 0 al 100. Los punteros a nulo de las hojas se representan con el valor -1.

Salida

Para cada caso de prueba se escribe en una línea *SI* si el árbol es zurdo y *NO* si el árbol no es zurdo.

Entrada de ejemplo

```
16
-1
5 -1 -1
5 2 -1 -1 6 -1 -1
5 2 7 -1 -1 -1 -1
5 -1 2 -1 7 -1 -1
5 8 2 -1 -1 -1 6 -1 -1
5 8 -1 4 -1 -1 9 -1 -1
5 8 -1 -1 4 1 -1 -1 -1
5 8 -1 -1 9 -1 2 -1 -1
4 6 5 2 1 -1 -1 -1 -1 -1 -1
3 6 5 1 -1 -1 -1 7 -1 -1 6 -1 -1
8 3 1 -1 -1 7 -1 -1 4 2 -1 -1 -1
4 1 -1 7 -1 -1 6 3 9 -1 -1 -1 -1
7 4 2 -1 -1 6 3 -1 -1 -1 8 2 -1 6 -1 -1 5 7 -1 -1 -1
7 5 9 -1 -1 6 3 -1 -1 -1 5 3 7 -1-1 -1 5 2 -1 -1 -1
6 8 9 5 2 -1 -1 -1 -1 4 6 -1 -1 -1 3 7 5 -1 -1 -1 4 -1 -1
```



Salida de ejemplo

```
SI
SI
NO
SI
NO
SI
NO
NO
NO
SI
SI
NO
NO
NO
NO
SI
```

Problema 2

(3.2 puntos) Los organizadores de un cierto juego on-line han decidido animar a sus jugadores mas torpes, dándoles varias partidas gratis para que puedan practicar. En concreto les darán `numPartidas` gratis a los jugadores que hayan perdido un mayor número de partidas consecutivas hasta este momento. Si varios jugadores han perdido el mismo número de partidas consecutivas, les darán las partidas a todos ellos. Una partida se considera ganada si su puntuación es mayor o igual que 5 y perdida si su puntuación es estrictamente menor que 5.

Implementa las siguientes funciones:

- Una función que dada una lista con los datos de las partidas de un jugador devuelva el mayor número de partidas consecutivas perdidas. La función debe tener coste lineal respecto al número de partidas jugadas.

- Una función que obtenga de la tabla los datos de los jugadores; decida quien o quienes son los jugadores a los que se les darán partidas; incremente su *bonus* (número de partidas que ya tienen pagadas y pueden jugar) en tantas partidas como se indique en la entrada; y liste los nombres de los jugadores a los que se les ha incrementado su *bonus* y su nuevo *bonus*.

Para resolver el problema se utilizan los siguientes tipos:

- `tLista = std::list<int>` , guarda las puntuaciones de las partidas jugadas

- ```
  struct datosJugador {
      int bonus;
      tLista dlista;
  };
  ```

  Guarda los datos de un jugador. En el campo `bonus` almacena el número de partidas que el jugador puede jugar gratis, bien porque ya ha pagado o porque ha obtenido como *premios* anteriormente. El campo `dLista` guarda las puntuaciones de las partidas ya jugadas.

- `tPareja = std::pair<std::string,datosJugador >`. Una pareja, cuya primera componente es el identificador del jugador y la segunda sus datos.

- `tTabla = std::map<std::string,datosJugador>`. Una tabla cuya clave es el identificador del jugador y cuyo valor son sus datos.

Entrada

Los datos de entrada comienzan con el número de casos. Cada caso se describe en varias líneas. En la primera se muestra el número de jugadores seguido de la cantidad de partidas dadas como premio. A continuación se muestran los datos de cada jugador en dos líneas, En la primera el identificador del jugador (cadena de caracteres sin blancos) seguido de su *bonus* y del número de partidas que lleva jugadas. En la línea siguiente la lista de puntuaciones de todas las partidas jugadas.

La función `resolverCaso` que se proporciona con la documentación del examen tiene implementada la lectura de los datos.

Salida

Para cada caso de prueba se escribe escriben tantas líneas como ganadores haya del premio. En cada línea se escribe el identificador del ganador, seguido de su nuevo *bonus*. Los identificadores se deben escribir en orden lexicográfico. Los casos acaban con una línea en blanco

Entrada de ejemplo

```
2
3 1
Jugador1 4 6
8 2 6 3 4 9
Jugador2 0 4
2 9 7 3
Jugador3 2 6
3 4 2 8 2 0
4 2
Jugador1 0 4
7 5 8 7
Jugador2 9 8
3 4 4 7 2 4 1 8
Jugador3 0 6
6 5 3 4 0 6
Jugador4 5 3
3 3 3
```

Salida de ejemplo

```
Jugador3 3

Jugador2 11
Jugador3 2
Jugador4 7
```

Autor: Isabel Pita.
```

TADs de Estructuras de Datos y Algoritmos

(curso 2016-17)

stack<T> (#include<stack>)

stack()
void push(**const** T &elem)
void pop()
const T &top() **const**
bool empty() **const**
unsigned int size() **const**

queue<T>

queue()
void push_back(**const** T &elem)
void pop_front()
const T &front() **const**
bool empty() **const**
unsigned int size() **const**

deque<T>

Deque()
void push_back(**const** T &e)
void push_front(**const** T &e)
const T &back() **const**
const T &front() **const**
void pop_back()
void pop_front()
bool empty() **const**
unsigned int size() **const**

Arbin<T>

Arbin()
Arbin(**const** Arbin &otro)
Arbin(**const** Arbin &iz, **const** T &e, **const** Arbin &dr)
bool esVacio() **const**
const T &raiz() **const**
Arbin hijoIz() **const**
Arbin hijoDr() **const**

list<T>

list()
list(**const** list<T> &otra)
void push_back(**const** T &e)
void push_front(**const** T &e)
const T &back() **const**
const T &front() **const**
void pop_back()
void pop_front()
bool empty() **const**
unsigned int size() **const**
const T &at(**unsigned int** pos) **const**
const_iterator cbegin() **const**
const_iterator cend() **const**
iterator begin()
iterator end() **const**
iterator erase(**const** iterator &it)
void insert(**const** T &elem, **const** iterator &it)

list<T>::const_iterator

it++
*it

list<T>::iterator

it++
*it

map<C,V>, unordered_map<C,V>

map()ó unordered_map()
map(**const** map &otro)
ó unordered_map(**const** unordered_map &otro)
bool contains(**const** C &clave) **const**
const V &at(**const** C &clave) **const**
bool empty() **const**
V &operator[](**const** C &clave)
pair<iterator,bool> insert(**const** value_type &par)
size_type erase(**const** C &clave)
iterator erase(**const_iterator** position)
const_iterator find(**const** C &clave) **const**
const_iterator cbegin() **const**
const_iterator cend() **const**
iterator find(**const** C &clave)
iterator begin()
iterator end() **const**

map<C,V>::const_iterator

it++
***it**

map<C,V>::iterator

it++
***it**

typedef pair<const Key, T> value_type;

pair<T1,T2> **#include <utility>**

first (atributo público)

second (atributo público)