Ejercicio 1

2 puntos

Las convenciones de código son reglas y estándares que guían la escritura de programas para mejorar la legibilidad y consistencia del código. Estas convenciones incluyen aspectos como el formato del código, la nomenclatura de variables, funciones y clases, el uso de comentarios y la organización del código. Seguir estas convenciones no solo facilita la comprensión del código por parte de otros programadores, sino que también ayuda a prevenir errores y a mantener un código más ordenado a lo largo del tiempo.

Con respecto al formato, existen distintas convenciones en lo que respecta a la *indentación* del código, es decir, la práctica de insertar espacios al comienzo de cada línea para estructurar visualmente el programa. En el lenguaje C, la indentación está ligada a los bloques de código, que se delimitan entre llaves. Sin embargo, según el estilo utilizado, las llaves se colocan de una u otra manera. Por ejemplo:

Estilo Kernighan y Ritchie

```
while (x == y) {
  f(x);
}
```

Estilo Allman

```
while (x == y)
{
  f(x);
}
```

Estilo Whitesmiths

```
while (x == y)
  {
  f(x);
  }
```

El estilo *Whitesmiths* coloca las llaves de apertura y cierre en líneas separadas. La llave de apertura `'{'` se sitúa una o más posiciones a la derecha con respecto a la línea anterior. El resto de sentencias del bloque, incluyendo la llave de cierre `'}'`, tienen la misma indentación que la llave de apertura. Por ejemplo, el siguiente fragmento de código sigue este estilo de indentación, donde los puntos (`·`) denotan espacios en blanco.

```
void mi_funcion(int x)   // Sin indentacion: 0 espacios
··{                      // Indentacion: 2 espacios
··if (x > 0)             // Indentacion: 2 espacios
····{                    // Indentacion: 6 espacios
····f(x);                // Indentacion: 6 espacios
····}                    // Indentacion: 6 espacios
··}                      // Indentacion: 2 espacios
```

Ten en cuenta que la diferencia (en número de espacios) entre un nivel de indentación y el siguiente puede variar. Lo importante es que un nivel de indentación mayor tiene más espacios al principio de la línea que un nivel menor.

El objetivo de este ejercicio es determinar si un programa en C está correctamente indentado según el estilo Whitesmiths. Para ello basta con conocer el primer carácter no blanco de cada línea, y saber cuántos espacios tiene delante. Por tanto, supongamos una lista con tantos elementos como líneas tiene el programa. Cada elemento es un par *(n, c)*, donde *c* contiene el primer carácter no blanco de la línea y *n* es el número de espacios que hay antes de *c*. Por ejemplo, el programa anterior puede representarse mediante la lista de pares `[(0, 'v'), (2, '{'), (2, 'i'), (6, '{'), (6, 'f'), (6, '}'), (2, '}')]`. Por simplicidad, puedes suponer que el programa no tiene líneas en blanco, y que la primera línea del programa no contiene una llave de apertura.

Escribe una función con la siguiente cabecera:

```
int comprobar_whitesmiths(const list<pair<int, char>> &programa);
```

La función debe devolver -1 si la indentación del programa pasado como parámetro sigue el estilo Whitesmiths. En caso contrario, debe devolver un valor entero indicando el número de la primera línea que no está correctamente indentada. Puedes suponer que el programa de entrada es sintácticamente correcto (es decir, las llaves están correctamente equilibradas) y que cada una de las llaves del programa, tanto de apertura como de cierre, están en una línea propia.

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

Entrada

La entrada consta de varios casos de prueba. Cada uno de ellos consiste en un número *N* que indica el número de líneas del programa. Después vienen *N* líneas con el texto del programa. Suponemos que se utilizan espacios en blanco (y no tabuladores) para la indentación.

La entrada finaliza con un programa de 0 líneas, que no se procesa.

Salida

Para cada caso de prueba debe imprimirse la palabra `CORRECTO` si el programa de entrada sigue la convención de estilo Whitesmiths para la indentación. En caso contrario, debe imprimirse el número de la primera línea que no está correctamente indentada. Las líneas se numeran comenzando desde el 1.

Entrada de ejemplo

```
11
#include <stdio.h>
int main()
  {
  int x;
  scanf("%d", &x);
  if (x > 0)
       {
       printf("Valor positivo\n");
       }
  return 0;
  }
4
int main()
{         // <- Esta linea y las dos siguientes deben indentarse
int x;
}
1
  const int x = 3;  // <- El nivel de indentacion inicial debe ser 0
5
int main()
  {
    int x;  // <- La indentacion debe ser la misma que la de la llave '{'
    x = 3;
  }
0
```

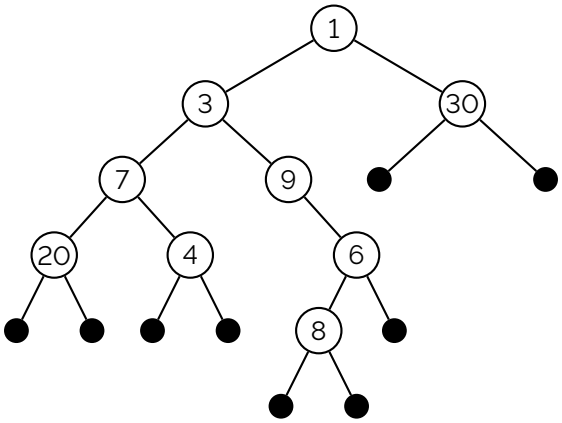
Salida de ejemplo

```
CORRECTO
2
1
3
```

Ejercicio 2

2 puntos

Las montañas *influencers* tienen una curiosa —aunque ordenada— red de caminos de senderismo. En su cima hay un hito del cual salen dos caminos que descienden por la ladera. Cada uno de esos caminos desemboca en un parador, o en otro hito. Si desemboca en un hito, el camino se bifurca en otros dos caminos descendentes que pueden desembocar en un parador o en otro hito, y así sucesivamente. Todas las ramificaciones desembocan eventualmente en un parador. Por ejemplo, la siguiente figura representa la red de caminos de una montaña *influencer*. Los círculos numerados representan lugares *influencers*, donde cada número indica cuánto de interesante es, y los círculos de color negro representan paradores:



Me gusta presumir ante mis amigos de lo interesantes que son mis rutas de senderismo. Para poder demostrarlo hago una foto de los lugares que voy encontrando por el camino y luego las publico en las redes sociales. Por eso, más que llegar muy alto, me interesa que mi ruta maximice el interés total de los lugares por los que paso. Eso sí: la ruta ha de empezar en un parador, terminar en otro parador distinto, y no debe atravesar dos veces el mismo lugar.

Por ejemplo, en el árbol de la figura superior, uno de mis caminos preferidos sería el que empieza en uno de los paradores situados por debajo del lugar valorado con 20. Desde ahí subiría hasta el lugar valorado con 1, y bajaría hasta uno de los paradores situados por debajo del lugar valorado con 30. En total, la valoración total de mi ruta sería de $20 + 7 + 3 + 1 + 30 = 61$. Si hubiese decidido ir desde el lugar 8 hasta el lugar 3, y de ahí a uno de los paradores bajo el lugar 4, entonces la valoración total sería de $8 + 6 + 9 + 3 + 7 + 4 = 37$.

1. Escribe una función con la siguiente cabecera:

   ```
   int max_interes_ruta(const BinTree<int> &montanya)
   ```

   La función debe devolver la valoración máxima que se podría obtener al atraversar cualquier ruta que comience y termine en un parador. La valoración de una ruta es la suma de los intereses de los hitos que se recorren en la misma. Puedes utilizar las funciones auxiliares recursivas que sean necesarias.

2. Indica y justifica el coste de `max_interes_ruta` y de las funciones auxiliares que definas.

Entrada

La entrada contiene varias líneas, cada una de ellas describiendo una montaña influencer. El carácter punto (`.`) representa un parador. Una cadena de la forma `(iz h dr)` determina un hito cuyo interés es `h`, del cual salen dos caminos descendentes, que llegan a los hitos o paradores descritos por `iz` y `dr`. Los valores de interés son números mayores o iguales que 0.

La entrada finaliza con una montaña con un único parador (es decir, la cadena `"."`), que no se procesa.

Salida

Para cada caso se escribirá un número entero que indique la valoración máxima que puede obtenerse al recorrer una ruta que empiece y termine en un parador. Entendemos que la valoración de un recorrido es la suma de los intereses de los hitos que se encuentran en el recorrido.

Entrada de ejemplo

```
((((. 20 .) 7 (. 4 .)) 3 (. 9 ((. 8 .) 6 .))) 1 (. 30 .))
((((. 9 .) 3 .) 2 (. 4 .)) 1 ((. 6 .) 5 ((. 8 .) 7 .)))
(((. 3 .) 2 .) 1 .)
(. 1 .)
.
```

Salida de ejemplo

```
61
35
6
1
```

Ejercicio 3

3 puntos

En la casa de subastas "*Barato, Barato*" se llevan a cabo subastas simultáneas de objetos de gran valor. Cada participante comienza con un saldo inicial y puede pujar por cada uno de los objetos en los que está interesado, entregando el importe pujado. Además, cada puja es secreta; ningún participante conoce la cantidad pujada por el resto. Esto provoca que varios participantes puedan pujar la misma cantidad para un objeto. Al cerrarse la subasta de un objeto determinado, este es ganado por aquel participante que haya pujado una cantidad mayor por él. En caso de haber varios participantes que hayan pujado dicha cantidad, el objeto es asignado al participante que primero pujó (en orden cronológico). El resto de los participantes que perdieron la puja recuperan el dinero pujado. Las pujas son simultáneas, por lo que los participantes pueden apostar por tantos objetos como deseen, pero un participante no puede pujar dos veces por el mismo objeto. En cualquier momento, el participante puede abandonar la casa de subastas, llevándose lo ganado hasta el momento y perdiendo el dinero apostado de las subastas pendientes de cerrarse.

Se pide implementar un TAD `CasaDeSubastas` con las siguientes operaciones:

- `void nuevo_participante(string part, int saldo_inicial)`

  Añade un nuevo participante a la casa de subastas, con el saldo inicial pasado por parámetro. Lanza la excepción `std::domain_error("Participante ya existente")` si `part` ya está presente en la casa. Si el `saldo_inicial` no tiene un valor mayor que 0, lanza la excepción `std::domain_error("Saldo inicial no valido")`.

- `void nueva_subasta(string obj, int puja_min)`

  Añade un nuevo objeto listo para ser subastado, con una puja mínima indicada por parámetro. Lanza la excepción `std::domain_error("Objeto no valido")` si el objeto `obj` ya está siendo subastado por la casa de apuestas o ya ha sido vendido. Si `puja_min` no tiene un valor mayor que 0, lanza la excepción `std::domain_error("Puja inicial no valida")`.

- `void nueva_puja(string part, string obj, int cantidad)`

  Añade la puja del participante `part` para conseguir el objeto `obj` con un valor indicado por `cantidad`. Lanza las siguientes excepciones:

  - `std::domain_error("Participante no existente")` si `part` no existe.
  - `std::domain_error("Objeto no valido")` si `obj` no existe o ya ha sido vendido.
  - `std::domain_error("Participante repetido")`, si `part` ya ha realizado una puja para `obj`.
  - `std::domain_error("Cantidad no valida")` si la `cantidad` supera el saldo actual del participante o no supera la cantidad mínima requerida por el objeto `obj`.

- `list<string> subastas_ganadas(string part) const`

  Devuelve en una lista los objetos ganados por el participante en orden alfabético. Si `part` no existe, se lanza la excepción `std::domain_error("Participante no existente")`.

- `void abandonar_casa(string part)`

  El participante abandona la casa, eliminando toda la información relativa a él, incluidas las pujas de objetos cuya subasta está pendiente de cerrar. Si `part` no existe, se lanza la excepción `std::domain_error("Participante no existente")`.

- `string cerrar_subasta(string obj)`

  Se otorga `obj` a quien pujó la cantidad más alta, y si hay varios, al primero que lo hizo y no haya abandonado la casa, devolviendo su nombre. Lanza la excepción `std::domain_error("Objeto no valido")` si `obj` no existe o ya ha sido vendido. Lanza la excepción `std::domain_error("Objeto no vendido")`, si nadie ha pujado por él.

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 participantes y de objetos subastados son cadenas de caracteres sin espacios en blanco.

Salida

Tras ejecutar las operaciones `nuevo_participante`, `nuevo_objeto`, `nueva_puja` y `abandonar_casa` debe imprimirse una línea con la palabra `OK`, salvo en caso de error.

Tras ejecutar la operación `cerrar_subasta` se debe imprimir, salvo en caso de error, una línea con el texto `X ha sido ganado por: Y`, donde `X` es el objeto cuya subasta se ha cerrado, e `Y` es el participante ganador de la subasta.

Tras ejecutar la operación `objetos_ganados` se debe imprimir, salvo en caso de error, una línea con el texto `X ha ganado:` (donde `X` es el participante), seguido por los objetos que el participante ha ganado, cada uno precedido por un espacio.

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

```
nuevo_participante juan 100
nuevo_participante ana 160
nueva_subasta funko_luffy 80
nueva_subasta funko_goku 120
nueva_subasta funko_sailor 10
nueva_puja juan funko_luffy 90
nueva_puja juan funko_sailor 10
cerrar_subasta funko_sailor
nueva_puja ana funko_luffy 85
nueva_puja ana funko_goku 130
cerrar_subasta funko_luffy
nueva_puja ana funko_goku 130
cerrar_subasta funko_goku
subastas_ganadas juan
FIN
nuevo_participante juan 100
nuevo_participante ana 160
nueva_puja juan funko_pop 20
nueva_subasta funko_pop -5
nueva_subasta funko_luffy 80
nueva_puja juan funko_luffy 90
nueva_puja ana funko_luffy 90
nueva_puja juan funko_luffy 10
abandonar_casa juan
cerrar_subasta funko_luffy
FIN
nuevo_participante juan 100
nuevo_participante ana 160
nueva_subasta funko_luffy 80
nueva_puja juan funko_luffy 90
nueva_puja ana funko_luffy 85
abandonar_casa juan
cerrar_subasta funko_luffy
FIN
```

Salida de ejemplo

```
OK
OK
OK
OK
OK
OK
OK
funko_sailor ha sido ganado por: juan
OK
ERROR: Cantidad no valida
funko_luffy ha sido ganado por: juan
OK
funko_goku ha sido ganado por: ana
juan ha ganado: funko_luffy funko_sailor
---
OK
OK
ERROR: Objeto no valido
ERROR: Puja inicial no valida
OK
OK
OK
ERROR: Participante repetido
OK
funko_luffy ha sido ganado por: ana
---
OK
OK
OK
OK
OK
OK
funko_luffy ha sido ganado por: ana
---
```