Estructuras de datos

Grados de la Facultad de Informática (UCM)

26 de Mayo de 2023

Normas de realización del examen

1. El examen dura **3 horas**.
2. Debes desarrollar e implementar soluciones para cada uno de los ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección `http://exacrc`.
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, apellidos y usuario del juez** en el espacio reservado para ello en los ficheros proporcionados.
5. Dispones de un fichero plantilla para cada ejercicio. Debes utilizarlo atendiendo a las instrucciones que se dan en cada fichero y escribiendo tu solución en los espacios reservados para ello, siempre entre las etiquetas `//@ <answer>` y `//@ </answer>`.
6. 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.
7. Si necesitas consultar la documentación de C++, está disponible en `http://exacrc/cppreference`.
8. A lo largo del examen se te pedirá que te identifiques y rellenes tus datos en una hoja de firma.

Primero resuelve el problema. Entonces, escribe el código.

— John Johnson

Comentar el código es como limpiar el cuarto de baño;
nadie quiere hacerlo, pero el resultado es siempre
una experiencia más agradable para uno mismo y sus invitados.

— Ryan Campbell

Ejercicio 1. Método de la burbuja (2 puntos)

En este problema se ordenan de menor a mayor los valores de una lista enlazada simple con el método de la burbuja. En este método se va desplazando el elemento mayor de la lista desde el principio hacia el final. Los elementos ya ordenados quedan en la parte final de la lista, de forma que cada elemento que se desplaza lo hace una posición menos que el anterior.

Por ejemplo, dada la lista: `[5, 2, 3, 6, 1, 7, 4]`

1. Se compara el elemento 5 con el 2, como el 5 el mayor que el dos, se intercambia con el valor 2, es decir se desplaza hacia el final.
2. A continuación se comparan el valor 5 con el 3, como el valor 5 es mayor que el tres se intercambian.
3. Se compara el valor 5 con el 6 y como el 5 es menor que el 6 no se intercambian y se continua el proceso con el valor 6.
4. Se compara el valor 6 con el 1 y como el 6 es mayor se intercambian.
5. Se compara el valor 6 con el 7 y no se intercambian.
6. Se compara el valor 7 con el 4 y se intercambian.
7. Cuando termina de desplazarse el valor máximo, la lista es: `[2, 3, 5, 1, 6, 4, 7]`
8. En este momento el valor 7 ya está ordenado y se repite el proceso empezando por el principio de la lista, valor 2, desplazando el máximo hacia la derecha como se ha hecho anteriormente. En este caso el proceso termina cuando el máximo llega a la posición anterior al nodo con valor 7.

Se pide implementar una función que dado un puntero al nodo en el que comienza la parte ya ordenada de la lista, desplace el elemento mayor de la lista hasta la posición anterior a la apuntada por el puntero que se da como parámetro en la forma descrita en el método de la burbuja. La función debe devolver un puntero al nodo con el valor máximo que se ha colocado al comienzo de la parte ya ordenada.

La función implementada se utiliza para implementar el algoritmo de ordenación con el método de la burbuja que se proporciona en la plantilla.

La función se implementará en la parte privada de la clase **`ListLinkedSingle`** que se incluye en la `plantilla1.cpp` y que implementa las **listas enlazadas simples con nodo fantasma**.

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. El coste de la operación implementada ha de ser lineal con respecto al número de elementos de la lista. El coste del algoritmo de ordenación que utiliza el método implementado es cuadrático respecto al número de elementos de la lista.

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 muestra el número de elementos de la lista y la segunda indica los valores de la lista desde el primero hasta el último. La entrada finaliza con un cero que no debe procesarse.

El número de elementos de la lista es mayor que cero, y los valores pueden almacenarse en una variable de tipo `int`.

Salida

Para cada caso de prueba se escribe un una línea el contenido de la lista ordenado.

Entrada de ejemplo

```
7
5 2 3 6 1 7 4
8
1 2 3 4 5 6 7 8
8
8 7 6 5 4 3 2 1
10
4 6 1 3 7 4 3 7 9 2
0
```

Salida de ejemplo

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

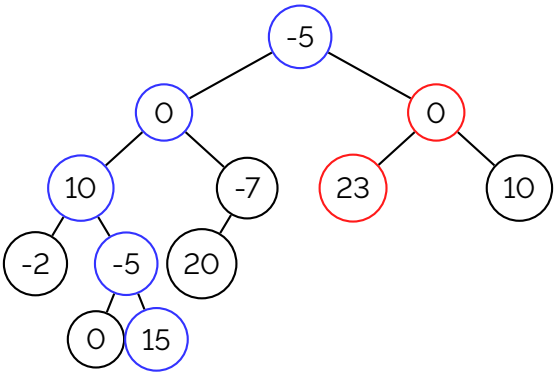
Ejercicio 2. Fantasía Final (2 puntos)

Para continuar su aventura, el héroe legendario debe adentrarse en las peligrosas cuevas del este. Afortunadamente, tiene un mapa muy completo en el que aparece toda la información relevante. Por ello, debe estudiar el mapa para prepararse antes de explorar las cuevas. Las cuevas pueden tener valiosos botines, pero también feroces enemigos. El objetivo de nuestro héroe es saber la ruta que debe tomar para maximizar el botín ganado, y saber cuántos puntos de vida como mínimo necesita inicialmente para obtenerlo (sabiendo que, como mínimo, debe empezar con 1 punto de vida). Si hay dos rutas que conducen a la misma cantidad de botín, el héroe prefiere aquella que necesite una menor cantidad de vida inicial para atravesarla.

El mapa está representado por un árbol binario de números enteros. La entrada a las cuevas está en la raíz, mientras que las salidas están en las hojas. Cada nodo del árbol es un valor v cuyo significado se ilustra a continuación:

- $v > 0$: un tesoro que el héroe cogerá agrandando su botín.
- $v < 0$: un enemigo que decrementará en $|v|$ los puntos de vida del héroe. Si el héroe se queda sin puntos de vida, la partida finaliza de inmediato. El héroe tiene que llegar a alguna de las hojas del árbol con al menos 1 punto de vida.
- $v = 0$: una poción que permitirá al héroe ganar 1 punto de vida.

Por ejemplo, dado el siguiente árbol:



El botín máximo obtenible es 25, el cual se consigue recorriendo la ruta de nodos azules. Los puntos de vida mínimos necesarios para hacer esa ruta serían 10. Hay 25 monedas (15 y 10), 10 puntos de daño (-5 y -5) y 1 poción (un nodo con 0). Tener menos de 10 puntos de vida no le permitiría llegar a la raíz con vida. Aunque la ruta representada en rojo permitiría al héroe empezar con 5 puntos de vida, esa ruta le daría menos monedas de oro, por ello no es la solución buscada.

Importante: Se debe implementar una función recursiva *externa* a la clase `Arbin` que explore el árbol de manera eficiente calculando los dos datos necesarios. La función no podrá tener parámetros de entrada/salida. Usa el fichero `plantilla2.cpp`.

Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en la descripción de un árbol binario de enteros con valores comprendidos entre -10^8 y 10^8. El árbol vacío se representa con un '`.`' y el árbol no vacío con raíz `R`, hijo izquierdo `i` e hijo derecho `dr` se representa como `(iz R dr)`.

Salida

Para cada árbol se escribirá una línea con los dos números separados por un espacio. Primero la ganancia máxima y después los puntos de vida mínimos para obtener tal ganancia.

Entrada de ejemplo

```
6
((. -2 .) 4 (. -1 .))
((. -5 (. 8 (. -1 .) )) 0 (. 8 (. 0 (. -1 .))))
((((. -2 .) 10 ((. 0 .) -5 (. 15 .))) 0 ((. 20 .) -7 .)) -5 ((. 23 .) 0 (. 10.)))
((. 20 .) 0 (. -2 .))
((. 4 .) 2 (. -7 (. 10 .)))
((((. 0 .) 0 .) -5 .) -3 (. -6 .))
```

Salida de ejemplo

```
4 2
8 1
25 10
20 1
12 8
0 9
```

Ejercicio 3. Batallas Fantasía Final (3 puntos)

Debemos simular un sistema de batallas por turnos entre héroes y villanos del juego *Fantasia Final*. Cuando un héroe o un villano aparecen en la batalla, tienen que esperar a que todos los personajes que estuvieran presentes anteriormente en la batalla ataquen. El personaje al que le toca el turno, ya sea héroe o villano, realiza un ataque y vuelve a esperar su turno para volver a atacar. Si el ataque derrota al adversario, éste desaparece definitivamente de la batalla. Los villanos tienen un nombre, una determinada cantidad de puntos de vida y un valor de daño al atacar. Los héroes, por otro lado, además de su nombre y sus puntos de vida, tienen una lista de ataques que van aprendiendo a lo largo de la batalla, cada uno con su valor de daño correspondiente. El nombre de cada personaje es único, es decir, tampoco existirán un villano y un héroe con el mismo nombre.

Se pide implementar un TAD `SistemaBatallas` que implemente las siguientes operaciones:

- `aparece_villano(v,s,a)`: Registra el nuevo villano `v` en la batalla con `s` puntos de vida y valor de ataque `a`. Si el villano no había aparecido en la batalla, se registra. Si el personaje ya había aparecido anteriormente en la batalla, se lanza una excepción del tipo `invalid_argument` con el mensaje `Personaje ya existente`.
- `aparece_heroe(h,s)`: Registra el nuevo héroe `h` en la batalla con `s` puntos de vida. Si el héroe no había aparecido en la batalla, se registra. Si el personaje ya había aparecido anteriormente en la batalla, se lanza una excepción del tipo `invalid_argument` con el mensaje `Personaje ya existente`.
- `aprende_ataque(h,a,v)`: Añade el ataque `a` con daño `v` a la lista de ataques del héroe `h`. Si el héroe no está en la batalla, se lanza una excepción del tipo `invalid_argument` con el mensaje `Heroe inexistente`. Si el héroe conoce ya ese ataque, se lanza una excepción del tipo `invalid_argument` con el mensaje `Ataque repetido`.
- `mostrar_ataques(h)`: Devuelve un vector de pares (nombre, daño) ordenados lexicográficamente con aquellos ataques aprendidos por el héroe `h`. Si el héroe no está registrado, se lanza una excepción del tipo `invalid_argument` con el mensaje `Heroe inexistente`.
- `mostrar_turnos()`: Devuelve un vector de pares con el nombre de todos los héroes y villanos acompañados de su cantidad correspondiente de puntos de vida, ordenados según su turno.
- `villano_ataca(v,h)`: El villano `v` ataca al héroe `h`, restándole a los puntos de vida del héroe el valor de daño de `v`. Tras el ataque, `v` vuelve a pedir turno cuando corresponda. Si el villano o el héroe no están registrados, se lanza una excepción con el mensaje `Villano inexistente` y `Heroe inexistente`, respectivamente. Si no es el turno del villano pasado como parámetro, se lanza una excepción con el mensaje `No es su turno`. Si el héroe atacado pierde todos sus puntos de vida, desaparece completamente de la batalla y la función devuelve el valor cierto.
- `heroe_ataca(h,a,v)`: El héroe `h` ataca al villano `v`, restándole a los puntos de vida del villano el valor de daño del ataque `a`. Tras el ataque, `h` vuelve a pedir turno cuando corresponda. Si el villano o el héroe no están registrados, se lanza una excepción con el mensaje `Villano inexistente` y `Heroe inexistente`, respectivamente. Si no es el turno del héroe pasado como parámetro, se lanza una excepción con el mensaje `No es su turno`. Si el héroe `h` no conoce el ataque `a`, se lanza una excepción con el mensaje `Ataque no aprendido`. Si el villano atacado pierde todos sus puntos de vida, desaparece completamente de la batalla y la función devuelve el valor cierto.

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 elección de los tipos y 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 los héroes y villanos son cadenas de caracteres sin blancos. Las cantidades de puntos de vida y los valores de ataque son números enteros positivos menores que 10^9.

Salida

Para cada caso de prueba se escribirán los datos que se piden. Las operaciones `aparece_villano`, `aparece_heroe`, y `aprende_ataque` no imprimen nada, salvo en caso de error. La comprobación de las excepciones indicadas hay que llevarlas a cabo en el orden indicado.

Las operaciones que generan datos de salida son:

- `mostrar_ataques`: si no ha producido ninguna excepción, imprime una primera línea con el mensaje `Ataques de` y el nombre del héroe, seguido de una línea con cada nombre del ataque y su daño ordenada alfabéticamente.
- `mostrar_turnos`: imprime una primera línea con `Turno:`, seguida de cada nombre y la cantidad de puntos de vida de héroes y villanos, en orden del turno que tienen.
- `villano_ataca`: si no ha producido ninguna excepción, imprime una línea con el nombre del villano, seguido de `ataca a`, seguido del nombre del héroe. En caso de que el héroe sea derrotado se imprime una segunda línea con el nombre del héroe seguido de la palabra `derrotado`.
- `heroe_ataca`: si no ha producido ninguna excepción, imprime una línea con el nombre del héroe, seguido de `ataca a`, seguido del nombre del villano. En caso de que el villano sea derrotado se imprime una segunda línea con el nombre del villano seguido de la palabra `derrotado`.

Si alguna operación produce una excepción se mostrará el mensaje `ERROR:` seguido del mensaje de la excepción como resultado de la operación, y nada más.

Cada caso termina con una línea con tres guiones (–).

Entrada de ejemplo

```
aparece_villano seymour 10 6
aparece_villano edea 12 10
aparece_heroe tidus 30
aprende_ataque tidus ataque_superpoderoso 10
aprende_ataque tidus ataque_impresionante 20
mostrar_ataques tidus
mostrar_turnos
villano_ataca seymour tidus
villano_ataca edea tidus
mostrar_turnos
heroe_ataca tidus ataque_impresionante seymour
villano_ataca edea tidus
mostrar_turnos
heroe_ataca tidus ataque_superpoderoso edea
mostrar_turnos
FIN
aparece_villano v1 5 4
aparece_heroe h1 3
aparece_villano v1 7 20
aparece_heroe h1 10
aprende_ataque h2 a1 10
aprende_ataque h1 a1 10
mostrar_ataques h2
villano_ataca v1 v2
heroe_ataca h1 a1 h2
villano_ataca v1 h1
heroe_ataca h1 a1 v1
aparece_heroe h1 10
heroe_ataca h1 a1 v1
villano_ataca v1 h1
heroe_ataca h1 a1 v1
aprende_ataque h1 a1 30
heroe_ataca h1 a1 v1
aparece_villano v1 5 4
FIN
```

Salida de ejemplo

```
Ataques de tidus
ataque_impresionante 20
ataque_superpoderoso 10
Turno:
seymour 10
edea 12
tidus 30
seymour ataca a tidus
edea ataca a tidus
Turno:
tidus 14
seymour 10
edea 12
tidus ataca a seymour
seymour derrotado
edea ataca a tidus
Turno:
tidus 4
edea 12
tidus ataca a edea
Turno:
edea 2
tidus 4
---
ERROR: Personaje ya existente
ERROR: Personaje ya existente
ERROR: Heroe inexistente
ERROR: Heroe inexistente
ERROR: Heroe inexistente
ERROR: Villano inexistente
v1 ataca a h1
h1 derrotado
ERROR: Heroe inexistente
ERROR: No es su turno
v1 ataca a h1
ERROR: Ataque no aprendido
h1 ataca a v1
v1 derrotado
---
```