

PROGRAMACIÓN I

PARCIAL 2

**EVALUACIÓN PARCIAL – RESOLUCIÓN DE CASOS**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FACULTAD: | **Tecnología Informática** | | | | |
| CARRERA: | **Analista Programador** | | | | |
| ALUMNO/A: |  | | | | |
| SEDE: | **Buenos Aires** | | LOCALIZACIÓN: | **Online** | |
| ASIGNATURA: | **Programación I** | | | | |
| CURSO: | **1O** | | TURNO: |  | |
| PROFESOR: | **Hunt, Alejandro** | | FECHA: |  | |
| TIEMPO DE RESOLUCIÓN: | | **Según cronograma** | EXAMEN PARCIAL NRO: | | **2** |
| MODALIDAD DE RESOLUCIÓN: | | | Virtual Domiciliario | | |

**Criterios de calificación:** Para acreditar los saberes deberá obtener, al menos, el 60% de los aspectos conceptuales, además de, al menos, el 60% de los aspectos procedimentales.

**Criterios de resolución:** Los alumnos recibirán la consigna del examen en la fecha de evaluación prevista por el cronograma de la asignatura. Deberán resolver y entregar el este examen previo al vencimiento indicado en la actividad. En caso de que el mismo sea entregado luego del vencimiento, se procederá a una penalización en el puntaje final, la cual dependerá del tiempo de demora.

**Criterios de evaluación:** Se evaluará la claridad en el planteamiento de los aspectos conceptuales y procedimentales. En los aspectos procedimentales, no solo se evaluará que el desarrollo provea una solución correcta a la consigna planteada, sino también la claridad, eficiencia y aplicación de los conceptos vistos en la cursada.

**Resultados de aprendizaje:**

* Comprender el concepto de árbol binario y poder utilizarlo en el desarrollo de sistemas informáticos para resolver problemas.
* Comprender e implementar las operaciones que aplican a los árboles binarios.
* Comprender el concepto de archivo y su implicancia en el desarrollo de software.
* Manipular archivos por medio de un programa y poder realizar operaciones de altas, bajas y modificaciones.
* Utilizar corte de control en la manipulación de archivos.
* Conocer diferentes paradigmas de programación.

|  |  |  |
| --- | --- | --- |
| **Criterio** | **Calificación** | **Observaciones** |
| **Aspectos Conceptuales** | | |
| Pregunta 1 |  |  |
| Pregunta 2 |  |  |
| Pregunta 3 |  |  |
| Pregunta 4 |  |  |
| Pregunta 5 |  |  |
| **Aspectos procedimentales** | | |
| Desarrollo del punto 1 |  |  |
| Desarrollo del punto 2 |  |  |
|  | | |
| Calificación final |  |  |

# Forma de entrega del examen

Se deberá entregar un archivo comprimido (.zip, .rar, .7z) que contenga, un documento con la resolución de todos los aspectos conceptuales y una solución que contenga el proyecto con la resolución de los aspectos procedimentales. En este último punto, el proyecto deberá incluir el código fuente desarrollado, el cual deberá compilar con el fin de ser ejecutado para realizar pruebas.

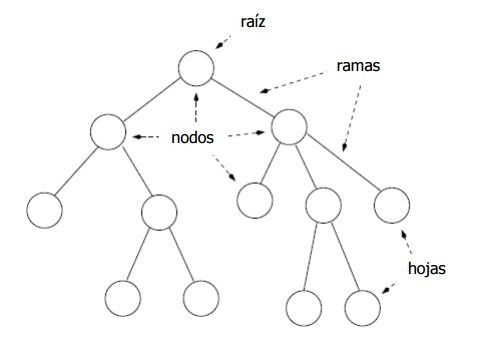
Los archivos entregados deberán tener el siguiente formado:

* Materia\_comision\_2021\_2P\_Apellido\_Nombre.ZIP. E**jemplo**:
* Programacion1\_1O\_2021\_2P\_Hunt\_Alejandro.ZIP

# Aspectos conceptuales

1. ¿Qué diferencias hay entre un grafo y un árbol?

**El grafo es un concepto más amplio, siendo árbol un tipo de grafo. En ese sentido, la pregunta formalmente no se ajusta mucho a ambos conceptos, pues se podría responder que no hay ninguna diferencia entre un árbol y un grafo acíclico conexo no-dirigido:**



**GRAFO ACÍCLICO CONEXO NO DIRIGIDO**

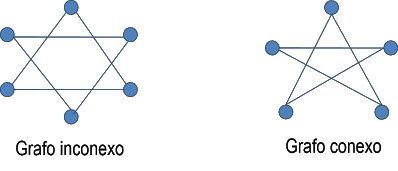
**Es decir, con el ejemplo dado, que es una imagen intercambiable (lo de “raíz”, “ramas”, “nodos” y “hojas” solo fue puesto para apuntalar el concepto, nada más).**

**Pero lo veamos con algunas precisiones (lo cual no es innecesario, sino que pienso que ayudará a consolidar el concepto teniendo en mente una estructura de árbol).**

**Veamos:**

|  |  |
| --- | --- |
| **Grafo cíclico:** cuando contiene algún ciclo simple. | **Grafo acíclico:** no contiene ningún ciclo simple. |
|  |  |

**Grafo conexo/inconexo:**decimos que es un grafo conexo, si es posible formar un camino desde cualquier vértice a cualquier otro en el grafo.



|  |  |
| --- | --- |
| **Grafo dirigido:** cada arista se asocia con dos vértices en forma ordenada. | **Grafo no-dirigido:** los lados no están orientados (no son flechas) |
|  |  |

1. ¿Qué es un árbol AVL? Explique su principal ventaja.

**Un árbol binario es una estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. Es decir, no pueden tener más de dos hijos (de ahí el nombre "binario")**

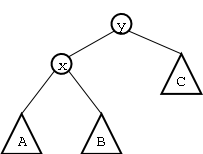
**Por definición, un árbol binario de búsqueda es un árbol binario en el cual cada nodo cumple con que todos los nodos de su subárbol izquierdo son menores que la raíz y todos los nodos del subárbol derecho son mayores que la raíz.**

**Un árbol binario de búsqueda AVL es, además (y finalmente), balanceado (las siglas AVL son por sus creadores: Adelson-Velskii y Landis).**

**Como principal ventaja se señala que simplifica las operaciones de búsqueda:**

* **Es balanceado: la altura de nodos de una rama no difiere más de una unidad respecto de la otra rama.**
* **Es equilibrable: el factor de equilibrio es la diferencia entre las alturas de una rama respecto de la otra, que puede ser -1(nodo izquierdo un nivel más alto), 0 (perfectamente equilibrado) y 1 (nodo derecho es un nivel más alto). Un árbol AVL se equilibra mediante la aplicación de rotaciones.**

**Rotaciones... ¿tocamos el tema...?**

****

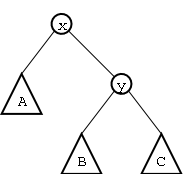
**Analicemos este árbol de búsqueda que debería cumplir con lo siguiente:**

**\*) x < y.**

**\*) todos los nodos del subárbol A deben ser menores que x e y.**

**\*) todos los nodos del subárbol B deben ser mayores que x pero menores que y.**

**\*) todos los nodos del subárbol C deben ser mayores que y (y por lo tanto, que x).**



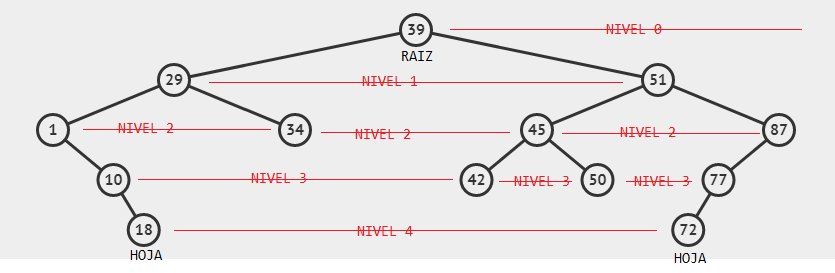
**En el resultado luego de una modificación “simple” (es decir, una de tantas técnicas) puede verse que, por las desigualdades descriptas en el párrafo anterior, el nuevo árbol sigue manteniendo el orden entre sus nodos, es decir, sigue siendo un árbol binario de búsqueda**

**Esta ilustración no es exhaustiva (para nada) y es al solo propósito de mostrar el concepto de «rotación».**

1. Dibuje un árbol binario de búsqueda de nivel 4 e indique
   1. El nodo raíz
   2. Los nodos hojas
   3. El nivel

**Un árbol de búsqueda binaria (ABB) es un árbol binario en el que cada vértice tiene solo hasta 2 hijos que satisfacen la propiedad ABB:**

* **todos los vértices en el subárbol izquierdo de un vértice deben tener un valor más pequeño que el suyo y**
* **todos los vértices en el subárbol derecho de un vértice deben contener un valor mayor que el suyo**

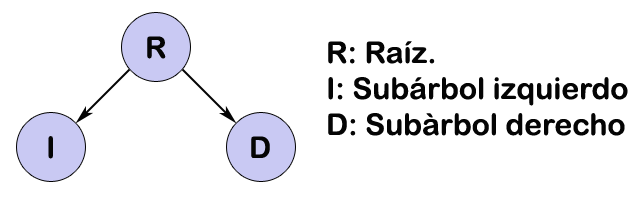


**El nivel se define como el número de ramas que hay que recorrer para llegar a la raíz de un nodo.**

1. ¿Qué es recorrer un árbol en preorden o inorden? Ejemplifique con el árbol del punto 3.

**Un árbol binario es un conjunto finito de nodos que consta de un nodo raíz que tiene dos subárboles binarios denominados subárbol izquierdo y subárbol derecho.**

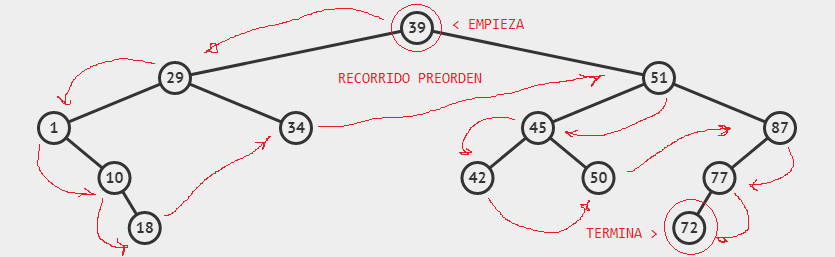
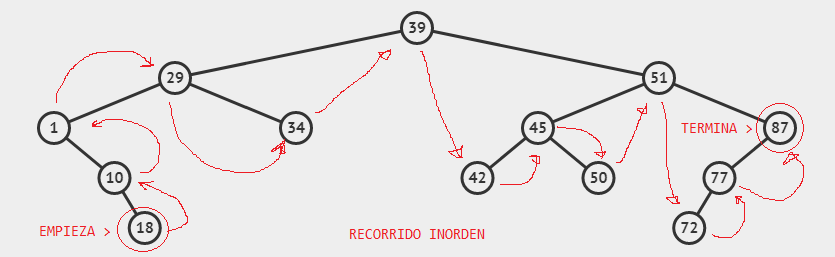
**El árbol binario es una estructura de datos muy útil cuando el tamaño de la estructura no se conoce, se necesita acceder a sus elementos ordenadamente.**



|  |  |
| --- | --- |
| **Preorden:** | **R, I, D** |
| **Inorden:** | **I, R, D** |
| **Postorden:** | **I, D, R** |

*(dado un orden, el elemento clave que lo diferencia es la ubicación del nodo raíz)*

**Básicamente se pueden utilizar tres formas para recorrer un árbol binario: preorden, inorden, postorden.**

* **En el orden preorden se recorre de la siguiente manera:**raíz, subárbol izquierdo, subárbol derecho.  
  
* **En el orden inorden se recorre de la siguiente manera:**subárbol izquierdo, raíz, subárbol derecho.  
  ****
* **En el orden postorden se recorre de la siguiente manera:**subárbol izquierdo, subárbol derecho, raíz.

1. Explique las características más relevantes del paradigma orientado a objetos.

**Los cuatro principios de la programación orientada a objetos son: encapsulación, abstracción, herencia y polimorfismo.**

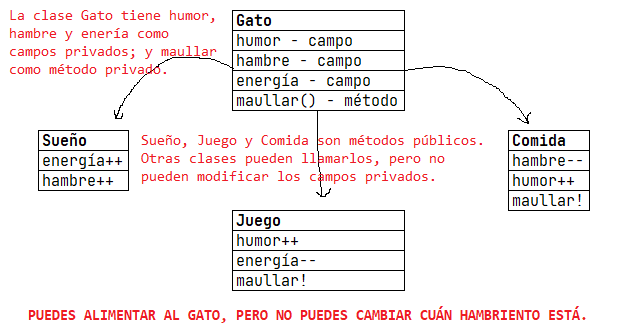
**ENCAPSULAMIENTO**

**Digamos que tenemos un programa. Tiene algunos objetos lógicamente diferentes que se comunican entre sí, de acuerdo con las reglas definidas en el programa.**

**La encapsulación se logra cuando cada objeto mantiene su estado privado, dentro de una clase. Otros objetos no tienen acceso directo a este estado. En cambio, solo pueden llamar a una lista de funciones públicas, llamadas métodos.**

**Entonces, el objeto administra su propio estado a través de métodos, y ninguna otra clase puede tocarlo a menos que se permita explícitamente. Si desea comunicarse con el objeto, debe utilizar los métodos proporcionados. Pero (de forma predeterminada), no puede cambiar el estado.**

**Supongamos que estamos creando un pequeño juego. Hay gente y hay un gato. Se comunican entre sí. Queremos aplicar la encapsulación, por lo que encapsulamos toda la lógica "gato" en una clase Gato. Puede verse así:**

****

**Aquí el "estado" del gato son las variables privadas estado de humor, hambre y energía. También tiene un método privado maullar(). A este lo pueden llamar cuando quiera, pero las otras clases no pueden decirle al gato cuándo maullar.**

**Lo que pueden hacer se define en los métodos públicos Sueño(), Juego() y Comida(). Cada uno de ellos modifica el estado interno de alguna manera y puede invocar a maullar(). Así es como se realiza la vinculación entre el estado privado y los métodos públicos.**

**Eso es encapsulación.**

**ABSTRACCIÓN**

**Se puede pensar en la abstracción como una extensión natural de la encapsulación.**

**En el diseño orientado a objetos, los programas suelen ser extremadamente grandes. Y los objetos separados se comunican mucho entre sí. Por lo tanto, mantener una base de código grande como esta durante años, con cambios en el camino, es difícil.**

**La abstracción es un concepto que tiene como objetivo aliviar este problema.**

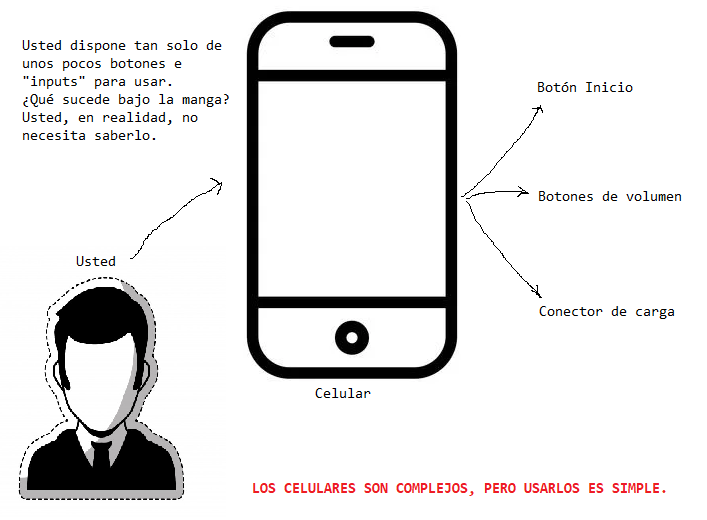
**Aplicar abstracción significa que cada objeto solo debe exponer un mecanismo de alto nivel para usarlo.**

**Este mecanismo debería ocultar los detalles de implementación interna. Solo debe revelar operaciones relevantes para los otros objetos.**

**Piense en una máquina de café. Hace muchas cosas y hace ruidos extraños debajo del capó. Pero todo lo que tienes que hacer es poner café y presionar un botón.**

**Preferiblemente, este mecanismo debe ser fácil de usar y rara vez debe cambiar con el tiempo. Piense en ello como un pequeño conjunto de métodos públicos que cualquier otra clase puede llamar sin "saber" cómo funcionan.**

**¿Otro ejemplo real de abstracción? Piense en cómo usa su teléfono:**

****

**Usted interactúa con tu teléfono usando solo unos pocos botones. ¿Qué está pasando bajo el capó? No es necesario que lo sepa: los detalles de implementación están ocultos. Solo necesita conocer un breve conjunto de acciones.**

**Los cambios de implementación, por ejemplo, una actualización de software, rara vez afectan la abstracción que utiliza.**

**HERENCIA**

**Bien, vimos cómo la encapsulación y la abstracción pueden ayudarnos a desarrollar y mantener una gran base de código.**

**Pero, ¿sabe cuál es otro problema común en el diseño de programación orientada a objetos?**

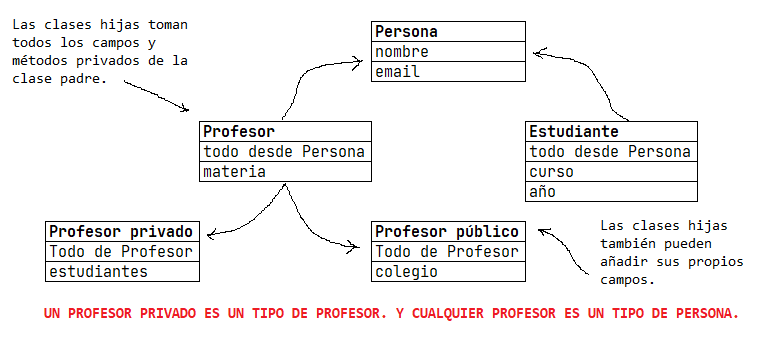
**Los objetos suelen ser muy similares. Comparten una lógica común. Pero no son del todo iguales. Ufff…**

**Entonces, ¿cómo reutilizamos la lógica común y extraemos la lógica única en una clase separada? Una forma de lograrlo es la herencia.**

**Significa que crea una clase (secundaria) derivando de otra clase (principal). De esta forma formamos una jerarquía.**

**La clase secundaria reutiliza todos los campos y métodos de la clase principal (parte común) y puede implementar los suyos propios (parte única).**

**Por ejemplo:**

****

**Si nuestro programa necesita gestionar profesores públicos y privados, pero también otro tipo de personas (como estudiantes), podemos implementar esta jerarquía de clases.**

**De esta manera, cada clase agrega solo lo que es necesario para ella mientras reutiliza la lógica común de las clases principales.**

**POLIMORFISMO**

**¡Nos quedamos con la palabra más compleja! Polimorfismo significa "muchas formas" en griego.**

**Así que ya conocemos el poder de la herencia y la usamos felizmente. Pero surge este problema:**

**Digamos que tenemos una clase principal y algunas clases secundarias que heredan de ella. A veces queremos usar una colección, por ejemplo, una lista, que contiene una combinación de todas estas clases. O tenemos un método implementado para la clase principal, pero también nos gustaría usarlo para los niños.**

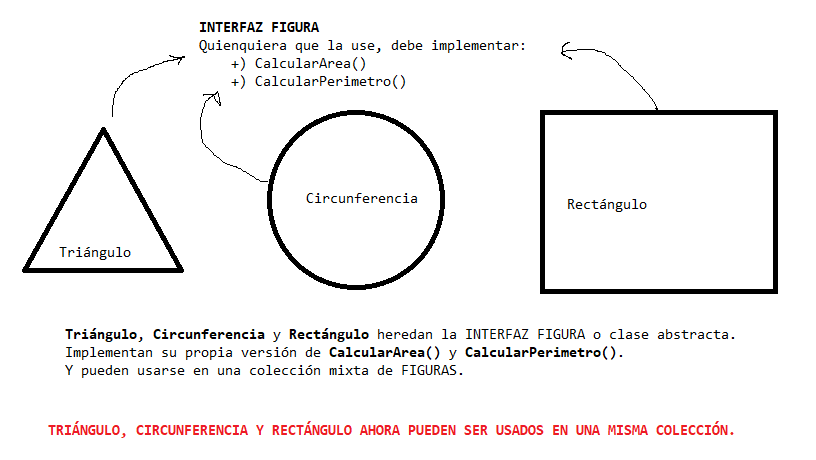
**Esto se puede resolver utilizando polimorfismo.**

**En pocas palabras, el polimorfismo proporciona una forma de usar una clase exactamente como su padre, por lo que no hay confusión con la mezcla de tipos. Pero cada clase secundaria mantiene sus propios métodos tal como están.**

**Esto suele suceder al definir una interfaz (principal) que se reutilizará. Describe varios métodos comunes. Luego, cada clase secundaria implementa su propia versión de estos métodos.**

**Cada vez que una colección (como una lista) o un método espera una instancia del padre (donde se describen los métodos comunes), el lenguaje se encarga de evaluar la implementación correcta del método común, independientemente del elemento secundario que se pase.**

**Eche un vistazo a un boceto de la implementación de figuras geométricas. Reutilizan una interfaz común para calcular el área de superficie y el perímetro:**

****

**Tener estas tres figuras heredando la interfaz de la figura principal le permite crear una lista de triángulos, círculos y rectángulos mixtos. Y tratarlos como el mismo tipo de objeto.**

**Luego, si esta lista intenta calcular la superficie de un elemento, se encuentra y ejecuta el método correcto. Si el elemento es un triángulo, se llama a CalculateArea() del triángulo. Si es un círculo, entonces se llama a CalculateArea() del círculo. Y así.**

**Si tiene una función que opera con una figura usando su parámetro, no tiene que definirla tres veces: una para un triángulo, un círculo y un rectángulo.**

**Puede definirlo una vez y aceptar una figura como argumento. Ya sea que pase un triángulo, círculo o rectángulo, siempre que implementen CalcularParametro(), su tipo no importa.**

**¿QUÉ QUEDO FUERA DEL TINTERO?**

**El POO se basa en la idea de encapsular estado y operaciones en objetos.**

**El apunte menciona otras cosas:**

* **Un objeto es cualquier cosa real o abstracta que posee una estructura que lo define y acciones que lo controlan.**
* **Un objeto presenta una identidad, un estado y un comportamiento en un momento dado. La identidad de un objeto le permite ser distinguido de otros, y esto se da gracias a la dirección de memoria.**
* **Los objetos son distintos si ocupan distintas direcciones de memoria.**
* **El estado de un objeto es el conjunto de valores concretos que lo caracterizan en un momento dado (como peso, color, precio, etc.)**
* **Una clase es una construcción estática que describe comportamientos comunes y atributos, incluyendo en ella, datos y métodos o funciones.**
* **Los métodos por defecto son: el constructor (que inicializa valores) y el destructor (que libera recursos al finalizar la vida útil de una instancia de una clase creada en memoria).**
* **A esta instancia de una clase se la llama objeto, quien posee identidad, comportamiento y estado fijo o variable especificas a ese objeto.**
* **Instanciación es la acción por la cual se crean instancias de una clase. Los objetos creados corresponden al tipo de la clase que lo origina.**

**¡Vaya...! Una “explicación” un tanto esotérica. El libro no se enfoca mucho en POO tanto como en C#. Entonces, ¿alguna nota final sobre el tema? Sí, y debería ser con ejemplos en código, pero como creo que ya sería exceder a lo pedido por la consigna, lo dejamos aquí por el momento (aunque yo creo que con un de buenos ejemplos se entendería mejor, pero bueno).**

# Aspectos procedimentales

Genere un programa de Winforms en C# (en Visual Studio) en base a los siguientes requisitos:

Se poseen dos archivos que son exportados de un sistema de ingreso de personal en una compañía. Un archivo posee los datos de los empleados (empleados.txt), y el otro los ingresos que registraron dichos empleados en la empresa (ingresos.txt). Ambos archivos tienen sus datos separados por coma, y luego de la consigna se encuentra la especificación de estos (además se adjuntan ejemplos con el enunciado). Se pide generar un programa que permita obtener la siguiente información de estos archivos:

1. Promedio de hora de ingreso de cada empleado, sin considerar los ingresos después del mediodía (12hs).
2. Listado de personas que ingresan después del mediodía. (12hs).

Consideraciones a tener en cuenta:

* Utilice los controles que considere necesarios para presentar la información, siempre teniendo en cuenta que la misma debe ser clara y comprensible.
* No se requiere que la ubicación de los archivos sea dinámica, por lo que puede estar hardcodeada (fija en el código).
* Considere que los archivos adjuntos son solo un ejemplo de formato, y se pueden realizar pruebas con archivos mas grandes o chicos, siempre y cuando los mismos respeten el formato.
* No es necesario para este examen, realizar validaciones de formato y se puede asumir que todos los archivos que se utilizan poseen el formato especificado.
* Los ingresos no poseen información sobre el día en que son realizados y no es relevante para la consigna.

El archivo de empleados (empleados.txt) posee los siguientes datos separados por coma, en el orden que se describen y no van a existir legajos repetidos:

* Legajo
* Nombre
* Apellido

El archivo de ingresos (ingresos.txt) posee los siguientes datos separados por coma, en el orden que se describe:

* Hora de Ingreso
* Minuto de Ingreso
* Segundo de Ingreso
* Legajo