EXÁMENES FINALES 8/7/2021 y 21/7/2021. Ambos con modalidad oral.

- ¿Cómo se pueden recorrer los nodos de un grafo? BFS y DFS.
- ¿Cuál es el algoritmo de búsqueda más eficiente? Acá hay que tener en cuenta todo, que esté ordenado, que esté distribuido uniformemente y el hashing.
- ¿Se puede hacer búsquedas con listas enlazadas?
    - Respondo que no y justifico, y después me dice cómo haría para mejorar esto: listas con arreglos.
- Después hablamos de cuando se dice que un objeto "es un" o está "contenido en", desde el punto de vista de herencia.
- ¿Cómo implementaría un árbol para una organización jerárquica de una empresa?
    - Hablar de árboles generales y la mejor forma de implementarlo (hijo por izquierda, hermano por derecha, ya que facilita la programación y ahorra mucha memoria)
- ¿Para qué sirven las excepciones en C++?
    - Ojo con esto, Marc es muy específico con el vocabulario. Yo defendí que era para facilitar la programación y lo correcto era "reducir la complejidad".
- Diferencias entre encapsulamiento y herencia (derivó en una charla media larga sobre muchos temas de la POO).
    - Encapsulamiento se refiere a agrupar funciones y datos miembro en un mismo objeto con una interfaz (datos y funciones miembro públicas) de modo que abstrae al usuario de la clase de su funcionamiento interno. Herencia se refiere a agrupar el código que tienen en común varias clases en otra de mayor jerarquía de modo de realizar programación por diferencia, programar solo lo que cambia respecto a la clase padre.
- ¿Cómo implementamos un diccionario generalista? (Tipo no sabes nada de como es el diccionario)
- ¿Cómo recorremos un grafo?
    - La respuesta sería que se puede recorrer por BFS o DFS.
- ¿Cuál es el peligro de hacer sobrecarga de operadores?
    - Creo que podes tener problemas de código confuso y además tener que tener cuidado con las reglas de conmutabilidad y precedencia.
- ¿Cuándo tiene sentido hacer un destructor virtual?
- ¿Cuáles son los algoritmos de ordenamiento más eficientes, en el caso generalista?
- ¿Qué problema hay al hacer la inserción con el algoritmo de inserción en árboles binarios que vimos en clase?
    - Está el tema ese que podes tener un árbol que es una lista degenerada (creo que así se llamaba). Podría ocurrir que un árbol quede desbalanceado y eso es malo porque hace que sea menos eficiente la implementación del árbol mediante un arreglo y la búsqueda/acceso pasa de ser O(log2(n)) a O(n) en el caso que todos los nodos estén sobre un mismo subárbol.
- ¿Para qué sirven las excepciones?
    - Para reducir la complejidad del código. En C había que pensar todo el tiempo en los posibles errores que podrían surgir y cubrirlos inmediatamente. C++ provee excepciones (objetos con información del error) que pueden ser atrapadas en un bloque de control para esas excepciones.
- ¿Cómo implementaría un híbrido entre listas enlazadas y arreglos y otros para que lo usaría? (Ventajas desventajas ejemplos,etc)

    - Podríamos hacer que cada nodo en una lista enlazada guarde un puntero a un arreglo (en el caso que los elementos se repitan frecuentemente) o bien un arreglo con punteros a listas enlazadas (como en la implementación de una tabla hash).
- ¿Cuál es la dificultad de mergesort?
- ¿Qué formas hay de representar un grafo y para qué se usan? (En código) - Sub pregunta relacionada a la anterior: de acuerdo al tipo de representación, ¿cuál utilizaría si quiero emplear la menor cantidad de cálculo y por qué?
- Miembros estáticos en C++: ¿qué son?, ¿cuáles son sus características? ¿para qué sirven?
- ¿Se puede hacer búsqueda binaria sin árboles de búsqueda binaria? (esta era la tramposa)
- ¿Qué es el polimorfismo? ¿Qué tiene que ver esto con los métodos virtuales?
    - El comportamiento del objeto se determina en tiempo de ejecución. Está muy vinculado a la herencia porque es inherentemente polimórfica (un objeto de una clase hija es a la vez la padre). Para poder llamar a los métodos de un objeto apuntado por un puntero cuyo tipo de dato es la clase padre, debo definir los métodos en la padre como virtual.
- ¿Por qué no tendría sentido usar un nodo tailer con listas simplemente enlazadas?
- ¿Cómo se puede guardar un árbol general a disco?
    - ¡Con árboles secuenciales! Serializan la estructura de un arbol y permiten archivar en memoria de forma eficiente (no guardamos punteros).
- ¿Por qué preferirías usar vector (stl) a arreglos de C++?
- ¿Cómo se ejecuta la excepción? Las tres etapas involucradas.
- ¿Qué pasa con el objeto cuando burbujea la excepción?
    - Se llama al destructor de la misma. Se desenrolla el stack.
- ¿Diversas formas de recorrer un grafo? BFS y DFS.
- (Pregunta trampa) ¿Cómo visitas un grafo? (No es la misma que la anterior). Me dijo con un for, es fácil…
- ¿Cuál es el árbol genérico más eficiente en memoria? ¿Por qué lo es?
    - Es el de nodo dinámico con hijo por izquierda, hermano por derecha y es el mejor porque no tenés overhead o tanta memoria sin utilizar al implementarlo.
- Diferencias entre listas doble y simplemente enlazadas.
- Ejemplo PUNTUAL de funciones amigas. (Iba por el lado de operadores de inserción, le encanta ese ejemplo)
- ¿Cuándo conviene usar matriz de adyacencia y cuando listas de adyacencia?
- ¿Por qué no existe el ordenamiento in-orden en árboles generalizados?
    - En un recorrido in orden en un árbol binario, se visita el nodo luego de haber visitado el hijo por izquierda… En árboles generales no es natural la definición.
- ¿Cómo conviene representar un grafo en código, y cuál conviene en cuanto al cálculo?
- ¿Qué algoritmo de ordenamiento conviene usar si no conocemos nada?
- ¿Cuándo conviene usar una plantilla de clases?
- ¿Conviene implementar una cola, con una lista o un arreglo?
- ¿Cuándo conviene implementar pilas con arreglos y cuando con listas enlazadas?

- ¿En que se distinguen los árboles k-arios de los generales? En esta tengan cuidado de sí dicen puede tener hasta k-hijos o si dicen tiene tantos hijos.
- ¿Para qué sirve la sintaxis de inicialización de miembros?
    - Para llamar al constructor del dato miembro y es necesario para inicializar las referencias a una clase.
- ¿En qué formas se puede implementar árboles de búsqueda binaria?
    - Con un árbol binario de búsqueda, con nodos hijos por derecha e izquierda.
- ¿Qué hay que tener en cuenta cuando se sobrecarga el operador =?
    - Que si el objeto contiene un puntero al realizar la asignación con el operador por defecto se puede realizar una copia superficial. Mismo problema que con el constructor copiador.
- ¿Cuándo conviene usar nodos hojas especiales?
    - Para árboles binarios llenos con mucha altura ya que por el teorema del árbol binario lleno las hojas son el número de nodos internos más uno.
- ¿Qué algoritmo de ordenamiento tiene que ser estable?
- En las colas con arreglo, ¿por qué no es buena idea aprovechar al máximo la capacidad?
- Diferencia entre constructor implícito y constructor por defecto.
- ¿Por qué alguien quisiera implementar un constructor copiador?
- Diferencia entre new y new[].
- Problemas de hacer delete sin [] con algo que fue creado con new []
- Diferencia árboles binarios de búsqueda y árboles binarios.
    - En árboles binarios de búsqueda hay una relación entre los valores que tomen los nodos y sus hijos izquierdos y derecho. En un árbol binario, en general, no se impone ninguna restricción en ese sentido.
- Excepciones, explicar y porque es mejor usarlas antes que hacer el handleo de errores con devolución de flags y eso.
- Diferencia entre Quicksort y Mergesort.
- ¿Cuál es el concepto de hashing y cómo se resuelven las colisiones?
- ¿Cuándo usar matriz de adyacencia y cuándo lista de adyacencia?
- Diferencia entre colas con arreglo y colas con listas enlazadas. Si en un ejemplo en el que uno sabe cuántos elementos tendrá en la cola, ¿cuál es mejor?
- Comparación entre herencia y polimorfismo.
    - En esta es importante decir que toda herencia conlleva un polimorfismo.
- Comparación entre búsqueda binaria en un arreglo, y búsqueda con árbol de búsqueda binaria.
    - Las 2 se parecen mucho pero es mejor la búsqueda binaria en un arreglo, porque si tenés un árbol desbalanceado tardas mucho más que con un arreglo.
- ¿Para qué sirven los métodos estáticos?
    - Cuando queremos implementar algo que sea común a todos los objetos de la clase.
- Comparación entre búsqueda binaria con arreglo y BST. ¿Cuándo es más conveniente usar uno y no otro?
    - El problema con BST es que, además de tener que construir el árbol si recibe un conjunto de datos desordenados, podría suceder que no esté balanceado y en un caso límite la búsqueda llega a O(n). En la búsqueda binaria con un arreglo, la búsqueda es siempre O(log2(n)) sin importar cómo recibo los

datos. Ahora bien, la inserción y eliminación de elementos en un arreglo puede ser costosa, hasta una complejidad de O(n). Por ello, si nos limitamos a la búsqueda, conviene el arreglo, si debemos actualizar constantemente conviene el BST.

- ¿Cuál es el problema de la herencia múltiple?
    - Problema del diamante: C++ no protege al heredar de dos clases que heredan de la misma… por lo que se estaría incluyendo dos veces la clase abuelo!
- ¿Qué detalles deben cuidarse al sobrecargar el operador de asignación?
- ¿Es posible hacer búsqueda binaria en listas?
- ¿Por qué razón alguien usaría excepciones y no usaría el mecanismo clásico de devolver el error por método/función?
- ¿Comparación listas simplemente enlazadas con doblemente enlazadas? ¿Se puede usar el tipo de dato de lista doblemente enlazada para un árbol binario? ¿Cuál es la principal diferencia?
- ¿Cómo recorrer un árbol?
- El problema de preorder 2 para la raíz, ¿existe también para árboles generales?
- Funciones amigas, ¿qué son y para qué sirven?
- Manera más eficiente de implementar un diccionario generalizado.
    - Hashing.
- Diferencias entre quicksort y mergesort. Decir cuál es mejor.
- ¿Por qué empíricamente quicksort es mejor que mergesort?
- Aplicación de métodos estáticos.
- ¿Para qué sirve la sobrecarga de operadores?
- ¿Se pueden hacer búsqueda de árboles binarios en listas?
- ¿Cómo se implementa la sintaxis de inicialización de datos miembros?
- Comparación Insertion sort y selection sort, ¿cuándo se usa cada uno?
- ¿Por qué es mejor usar excepciones y no el mecanismo clásico de devolver errores?
- PARA HACER UN RECORRIDO de grafos, ¿es mejor usar matriz o lista de adyacencia?
- Cuando se crea un objeto con new y hay un destructor virtual en la clase del objeto. Cuando se destruye, ¿a qué destructor se llama?
- Diccionarios: mejor manera de hacer un diccionario genérico.
- ¿Cómo implementar un diccionario con hashing?

COMPENDIO DE PREGUNTAS DE PARCIALES:

Primer parcial:

Elegir la opción correcta y **justificar** brevemente la opción elegida.

"Los miembros privados de una clase...

Respuesta seleccionada: sólo pueden accederse por la clase y las funciones y clases amigas."

En C++, ¿cuándo conviene usar punteros y cuándo referencias? **Justificar** brevemente.

Conviene usar referencias:

a) cuando no nos interesa cambiar la referencia al objeto.
b) para no tener que apuntar con el operador * o ->, lo cual resulta en código más facil de entender.

¿Qué es un constructor por defecto implícito y qué hace? **Justificar** brevemente.

Rta.: el constructor por defecto implícito es provisto por el lenguaje C++ cuando no se define el constructor por defecto. Su función es inicializar todos los datos miembros con sus constructores por defecto.

Analizar la veracidad de la siguiente afirmación. **Justificar** brevemente.

"Si una función fue declarada con **virtual void foo() = 0;** entonces no es necesario proveer una definición de **foo** en el programa".

Rta.: La afirmación es cierta: puedo declarar una función con virtual void foo() = 0; y nunca instanciar un objeto de esa clase; el lenguaje no me exige implementarla.
Esta situación se puede dar, por ejemplo, en una arquitectura de plug-ins. Puede que un plug-in no necesite crear objetos de cierta clase base abstracta definida por el programa que llama a los plug-ins; entonces la clase quedará sin definir, pero el plug-in compilará.

Analizar la veracidad de la siguiente afirmación. **Justificar** brevemente.

"Uno de los usos más importantes de **void *** en lenguaje C es resolver la falta de plantillas".

La afirmación es falsa.
Rta.: las plantillas se usan cuando necesitamos realizar la *misma* operación sobre diferentes tipos de datos. void * es mucho más general porque permite realizar *diferentes* operaciones en función del tipo de datos.

Pregunta 1 5 de 10 puntos

Analizar la veracidad de la siguiente afirmación. **Justificar** brevemente.

"El enlazado interno con **static** del lenguaje C es bastante parecido al uso de **private:** en C++".

Respuesta seleccionada:	En el lenguaje C, 'static' refiere a la visibilidad de una variable o función en un mismo archivo .c, o bien a la constancia en la memoria de la variable, es decir, se utiliza para mantener la variable, junto con su información, en memoria desde que es creada hasta la finalización del programa. En C++, utilizar 'private:' dentro de una clase indica que ciertas variables o métodos son accesibles únicamente dentro del código de la misma clase. Por ende, no siempre es 'static' en C igual a 'private:' en C++. Este último no mantiene la información de una variable hasta que finaliza la ejecución del programa.
Respuesta correcta:	[None]
Comentarios para respuesta:	Definiste correctamente **static**, pero la pregunta refería al "enlazado interno", y por tanto a la ausencia de visibilidad de una variable o función fuera del objeto compilado (¿recuerdas la explicación sobre la tabla de símbolos que di en clase?) Por tanto, **static** impide que otros módulos puedan enlazar con una variable o función. En ese sentido **static** se comporta en modo similar a **private:**.

Pregunta 2

10 de 10 puntos

¿Cuándo conviene usar programación orientada a objetos y cuándo programación imperativa? **Justificar** brevemente.

Respuesta seleccionada:	La programación orientada a objetos es recomendable cuando se tienen programas con un elevado nivel de complejidad, mientras que la imperativa cuando la complejidad del problema no es tan elevada y el programa se puede llevar desarrollar de manera secuencial. Cabe destacar que una excepción a esto es el kernel Linux, que está programado en C a pesar de la gran complejidad del mismo.
Respuesta correcta:	[None]
Comentarios para respuesta:	Excelente. Contrarrespuesta a una buena observación: Linux está programado en C por eficiencia (para evitar apuntar a punteros **this**).

¿Qué es un constructor por defecto implícito y qué hace? **Justificar** brevemente.

Respuesta seleccionada:	Un constructor por defecto implícito es aquel constructor que el compilador de C++ asigna a todas las clases que no tengan uno declarado de manera explícita. El mismo no hace nada, es decir, está en blanco.
Respuesta correcta:	[None]
Comentarios para respuesta:	La primera parte de la respuesta, muy bien. Pero hace una cosa: inicializa todos los datos miembros con sus constructores por defecto (si en un constructor necesito llamar a los constructores de los datos miembros con parámetros, debo usar la sintaxis de inicialización de datos miembro).

Analizar la veracidad de la siguiente afirmación. **Justificar** brevemente.

"En C++, un programa produce fallos en forma segura si la asignación de memoria dinámica falla".

Respuesta seleccionada:	Si la asignación de memoria dinámica se realiza con 'malloc', 'calloc' o 'realloc', los fallos no se producen en "forma segura". Se deben prevenir utilizando las mismas herramientas y técnicas que se utilizan en el lenguaje C o lanzar nuestra propia excepción. Cuando se realiza la asignación con un 'new' o 'new[]', sí produce fallos de "forma segura", ya que lanza una exepción cuando el mismo no pudo llevarse a cabo la asignación deseada.

Analizar la veracidad de la siguiente frase. **Justificar** brevemente.

"El uso de excepciones puede ser dificultoso porque en el momento de desenrollarse el stack, los destructures de los objetos locales no son llamados y eso puede causar memory leaks".

Respuesta seleccionada:	Al ser lanzada una excepción, la misma "burbujea" hasta que algún método o función la "atrape". Una vez "atrapada", la ejecución del programa continúa desde la misma función que la haya "atrapado". De todos modos, previo a esto, a medida que el stack se desenrolla se van llamando a los destructores de cada clase según sea adecuado, por lo que no se causarían memory leaks por este movito. Lo que sí puede causar uno es el hecho de que el programa continúa ejecutándose desde la misma función que "atrapó" la excepción. Este sería el caso en que dicha función precise por algún motivo de esa memoria dinámica que fue previamente destruida (por ejemplo, trate de acceder a un objeto que ya no existe).

Analizar la veracidad de la siguiente afirmación. **Justificar** brevemente.

"Si una clase requiere un destructor para liberar algún recurso, entonces es muy probable que también requiera un constructor copiador y una sobreescritura de **operator=**".

Respuesta seleccionada:	Es cierto que en muchos casos sí sea necesario un constructor copiador y sobreescribir "operator=". La justificación es fácil de comprender mediante un ejemplo: Supongamos que al crear un objeto A, el constructor reserva memoria utilizando "malloc". Cuando se ejecute el destructor de la clase, suponiendo que el programador lo programó correctamente, se liberará la memoria asignada en el constructor mediante la correspondiente llamada a "free". ¿Qué sucede al inicializar otra instancia B que sea igual a A o queremos asignar A = B? En estos casos, será necesario copiar la información almacenada en A en B, y debemos hacerlo _a mano_, ya que el constructor copiador por defecto y el 'operator=' por defecto _no saben_ que la memoria es dinámica: sólo reconocen el puntero y terminan copiando este último. Esto implica que, al modificar esta memoria en B se modificaría A también.
Respuesta correcta:	[None]
Comentarios para respuesta:	El ejemplo es correcto, pero mejor es la regla general: si el destructor debe liberar algún recurso, esto se debe a que lo referencia a través de un puntero. Si no se define un constructor copiador, el constructor implícito copia el puntero pero no lo apuntado; lo mismo ocurre con el operador de asignación.

Question 1

0 / 10

¿Cuándo conviene usar sobrecarga de funciones y cuándo plantillas de funciones? **Justificar** brevemente.

Analizar la veracidad de la siguiente afirmación.
Justificar brevemente.

"La herencia refiere a como código idéntico puede producir resultados diferentes, dependiendo del tipo de objeto concreto que se está procesando".

Falso. La herencia refiere a cómo podemos programar objetos distintos que cumplen con el criterio "es un" (ejemplo: camiónm y auto son vehículos). Lo que nos permite hacer es utilizar la programación por diferencia y escribir menos código. Puede producir resultados disntintos pero no es la idea central e la herencia.

¿Cuál(es) de las siguientes afirmaciones acerca de los constructores por defecto es/son cierta(s)?
Justificar brevemente.

Correct
The correct answers are (2) No devuelve ningún tipo de datos., (3) El programador puede definirlo, pero C++ no lo exige., (6) Si el programador no lo define, lo provee el lenguaje C++ bajo determinadas condiciones, pero no siempre.

Analizar la veracidad de la siguiente afirmación.
Justificar brevemente.

"Sólo se puede usar un único bloque **try**/**catch** en un programa".

Falso. Cada try se puede usar con varios catch para dentro de un mismo código tratar varias excepciones de distintos tipos y tomar distintas acciones para cada una de ellas. Además, los bloques try-catch pueden anidarse y puede haber más de uno, localizados en distintas partes del código donde pueda haber problemas.

Analizar la veracidad de la siguiente afirmación. Justificar brevemente.

"Si una función recibe un parámetro de tipo **const char ***, no puede reasignar ni liberar la memoria asociada. El siguiente código, por ejemplo, no compila:

```
void foo(const char * inVal) {
    inVal = new const char[10]; //
no compila

    delete[] inVal;             //
no compila

}
```

"

Falso. Lo que es constante en esa declaración no es el puntero sino el contenido al que apunta. Mientras el puntero apunte a datos de tipo const char, puede ser tranquilamente modificado, y por supuesto desreferenciado.

Analizar la veracidad de la siguiente afirmación.
Justificar brevemente.

"Las listas de inicialización son meramente una sintaxis alternativa para configurar los datos miembros en el constructor. Son equivalentes en funcionalidad".

Falso, las listas de inicialización son armadas y ejecutadas en funciones fuera de la clase y se utilizan para crear objetos. Debido a esto, se necesita del constructor para poder utilizar una lista de inicialización y la lista de inicialización NO es un reemplazo del constructor: llama al constructor, no inicializa "per se" los datos miembro. Como información adicional, las listas de inicialización son muy útiles al utilizar arreglos de objetos ya que podemos, a partir de ellas, definir varios objetos de la misma clase en una colección ordenada, pero podemos utilizar para cada uno de ellos el constructor que querramos de los que tenemos definidos, para inicializar cada objeto de la forma más conveniente usando el constructor que más se adecue a nuestras necesidades.

Segundo parcial:

Pregunta 1 10 de 10 puntos

¿**Cuándo** conviene implementar un árbol binario con un arreglo? **Justificar** brevemente.

Respuesta seleccionada: Conviene implementar un arbol binario con arreglo cuando el mismo es completo o casi completo, debido a que de no serlo, se desperdiciaria mucho espacio en memoria con campos vacios.
Respuesta correcta: [None]
Comentarios para respuesta: Muy bien.

En una estructura de datos de tipo lista, ¿el **comienzo de la lista** coincide siempre con el **primer elemento**? **Justificar** brevemente.

Respuesta seleccionada: No necesariamente, en una lista se definen dos caracteristicas, head y tail. Head sería el comienzo de la lista. Dependiendo de que implementacion se este utilizando head puede ser varias cosas. Por ejemplo, cuando se utiliza un nodo header el comienzo de la lista no es el primer elemento sino solamente un nodo que sirve para facilitar las operaciones de listas y apunta al primer elemento.
Respuesta correcta: [None]
Comentarios para respuesta: Fantástico.

En el algoritmo de ordenamiento **quicksort**, ¿por qué solemos elegir como pivote el **elemento central**? **Justificar** brevemente.

Respuesta seleccionada: En el algoritmo quicksort se elige de pivote el elemento central ya que estadisticamente es una de las opciones que minimiza "worst case scenarios", es decir el caso cuando el algoritmo tarda O(n^2) en complejidad. Otra buena opción es elegir un elemento al azar. En general elegir el elemento central es bueno cuando se tienen datos parcialmente ordenados, para el resto de los casos elegir un elemento al azar es buena opción tambien.
Respuesta correcta: [None]
Comentarios para respuesta: Muy bien!

Analizar la veracidad de la siguiente afirmación:

"Para recorrer grafos en forma **eficiente** conviene siempre representarlos con listas de adyacencia".

Respuesta seleccionada: En el caso que un grafo sea disperso, conviene representarlo mediante una lista de adyacencia, ya que se tiene un arreglo de logitud V con punteros a listas enlazadas con los vertices. Para recorrerlo requiero O(V + E).

Para el caso que se tenga un grafo denso conviene representarlo mediante una matriz de adyacencia, ya que existen muchas uniones entre los nodos lo que ocasiona que la matriz este llena o casi llena, lo que no ocurriría con un grafo disperso en el que se desperdiciaria memoria con campos vacios en la matriz. Recorrer una matriz de adyacencia lleva O(V^2)

Por lo tanto la manera eficiente de representar un grafo depende de que tipo de grafo se este analizando.
Respuesta correcta: [None]
Comentarios para respuesta: Y... la matriz de adyacencia evita tener que apuntar a los punteros **next** de las listas de adyacencia! (es una pequeña ganancia).

Analizar la veracidad de la siguiente afirmación:

"Toda **lista** permite implementar eficientemente **colas**".

Respuesta seleccionada: Si se evalua la complejidad algoritmica tanto la implemenmtacion con arreglos como listas dan resultados similiares. La implementación con arreglos puede ser mas complicada o confusa ya que un arreglo tiene cantidad de datos fijos entonces tenemos que encontrar una solucion a varios problemas: que hacer cuando rear y front llegan al final de arreglo, como determinar que un arreglo circular esta lleno o vacio, etc. En ese sentido seria mas dificil aplicar una cola con arreglos, pero en complejidad computacional es igual a la implementacion con listas enlazadas.

Eso no quiere decir que no exiista algun tipo de lista que haga la complejidad disminuir. afirmar toda es muy egneral.
Respuesta correcta: [None]
Comentarios para respuesta: Muy bien.

¿Cuándo conviene usar **listas ordenadas** y cuándo **listas desordenadas**? **Justificar** brevemente.

Respuesta seleccionada: Las listas ordenadas son preferibles cuando, valga la redundancia, es importante el orden de los elementos como sería el caso si luego se van a realizar búsquedas entre los elementos de estas listas.
Por otra parte, las listas desordenadas son convenientes cuando no hay correlación entre la posición de un elemento y su valor, como podría ser en una cola o una pila, donde lo importante son un extremo o ambos extremos de la lista, según el caso, más no si sus elementos están ordenados de una manera particular. En la fila del banco importa quién llegó primero, no si es "más alto" o " más bajo" que el siguiente cliente esperando.

Respuesta correcta: [None]

Comentarios para respuesta: En el primer párrafo defines "ordenado" como "importante en el orden de sus elementos", por lo que "desordenado" debería ser "sin importar el orden". Sin embargo, en una pila o una cola el orden interesa, y mucho (de lo contrario, estas estructuras de datos no funcionarían porque los elementos dentro de la estructura se mezclarían).

Por tanto, usamos listas desordenadas cuando **no** nos interesa el orden particular de los elementos (como en una lista de compras de un supermercado, lo cual también es debatible por el orden de las góndolas...).

Analizar la veracidad de la siguiente afirmación:

"La **búsqueda por interpolación** es siempre computacionalmente eficiente".

Respuesta seleccionada: La afirmación es falsa. Si siempre fuera computacionalmente eficiente, sería el único método de búsqueda que se utilizaría.
Es cierto que la búsqueda por interpolación es computacionalmente eficiente, y el peor caso requiere O(n) comparaciones y el caso promedio O(log(log(n)).
Personalmente, utilizaría este algoritmo de búsqueda si tuviera un conjunto de números ordenados, mas no lo usaría para recorrer los archivos en un disco duro, en particular si se tiene en cuenta que la mayoría de los sistemas de archivos actuales son árboles, usaría otro método de búsqueda.

Respuesta correcta: [None]

Comentarios para respuesta: Te faltó decir algo. El algoritmo es O(log(log(n))) cuando las claves están **uniformemente distribuidas**!

Analizar la veracidad de la siguente afirmación:

"Implementar un **diccionario** con hashing es computacionalmente eficiente".

Respuesta seleccionada:	Es computacionalmente eficiente si se tiene un enorme conjunto de datos que se desea almacenar en el diccionario y la función hash tiene bajas probabilidades de tener colisiones (si se utiliza, por ejemplo, SHA-2). En otro caso, esto no es necesariamente cierto, pues de ocurrir una colisión (que asumimos es probable) debe implementarse, por ejemplo, un método de hashing abierto, que utiliza listas y, por ende, agrega complejidad al sistema y la búsqueda del valor en el diccionario se vuelve menos eficiente.
Respuesta correcta:	[None]
Comentarios para respuesta:	La probabilidad de colisiones depende de la relación entre la cantidad de elementos N y la cantidad de slots M. De hecho, será, en promedio, N/M.

Analizar la veracidad de la siguiente afirmación:

"Para una tabla hash de tamaño `N`, la función `h(x) = (x + rand()) % N` es una buena función hash".

Nota: `rand()` es parte de `stdlib.h` y devuelve un valor aleatorio entre `0` y `RAND_MAX` (un valor mayor a 32767).

Respuesta seleccionada:	La función provista posee un grave problema y es que no se puede recuperar un valor a pesar de estar utilizando la misma clave. Esto se debe a que rand() no devuelve siempre el mismo número, y la suma para una determinada clave x sería diferente casi siempre (la excepción es que rand(), casualmente, devuelva lo mismo en más de una llamada dado el mismo x).
Respuesta correcta:	[None]
Comentarios para respuesta:	[No se ha dado ninguna]

¿Cuándo conviene usar **listas con arreglo** y cuándo **listas enlazadas**? **Justificar** brevemente.

Respuesta seleccionada:	Si necesito insertar o eliminar elementos de la lista me convendría usar listas enlazadas ya que son acciones de una complejidad O(1). En el caso de los arreglos, para insertar o eliminar debo mover todos los elementos restantes del arreglo para reacomodarlos tras la accion. No obstante, si debo acceder a posiciones arbitrarias de la lista, una lista con arreglo resulta mas optima porque el acceso es inmediato mientras que en una enlazada quizás debo recorrer muchos elementos hasta encontrarla (complejidad O(1)vsO(n) en el peor caso, respectivamente).
Respuesta correcta:	[None]
Comentarios para respuesta:	Afirmas que "En el caso de los arreglos, para insertar o eliminar debo mover todos los elementos restantes del arreglo para reacomodarlos tras la accion". Esto no es cierto, ya que no es necesario mover **todos** los elementos sino sólo los que están a la derecha.

En árboles binarios, ¿qué **problema** puede ocurrir con los algoritmos de inserción y eliminación básicos vistos en clase? **Justificar** brevemente.

Respuesta seleccionada:	El problema surge al querer eliminar un nodo interno. En ese caso se requiere analizar muchas condiciones primero. Por ejemplo, si se quiere eliminar un nodo con una sola hoja, se cambia el puntero del padre del eliminado al de la hoja. Pero si se trata de un nodo con dos nodos hijos, se deben analizar condiciones para que no se desbalancee el arbol en el caso de BSTs. Por caso, eliminar el nodo raiz en un BST requiere buscar el nodo que le sigue en valor para que no quede desbalanceado
Respuesta correcta:	[None]
Comentarios para respuesta:	Afirmas que **surge** un problema, pero solo al final de la respuesta mencionas el desbalanceo. Considero que reconoces aquí el problema (lo cual es correcto). Es falso que "El problema surge al querer eliminar un nodo interno"; el problema del desbalanceo que mencionas también puede producirse con **nodos hoja**. El problema también surge al **insertar** nodos (si inserto elementos ordenados con el algoritmo de inserción básico, obtendré un árbol degenerado en lista).

La búsqueda más eficiente se efectúa en un árbol binario balanceado. Desafortunadamente, la función Inserta no asegura que el árbol permanezca balanceado, el grado de balance depende del orden del orden en que son insertados los nodos en el árbol.

La altura de un árbol binario es el nivel máximo de sus hojas (profundidad). La altura del árbol nulo se define como –1. Un árbol binario balanceado es un árbol binario en el cual las alturas de los dos subárboles de todo nodo difiere a lo sumo en 1. El balance de un nodo en un árbol binario se define como la altura de su subárbol izquierdo menos la altura de su subárbol derecho. Cada nodo en un árbol binario balanceado tiene balance igual a 1, -1 o 0, dependiendo de si la altura de su subárbol izquierdo es mayor que, menor que o igual a la altura de su subárbol derecho.

¿Por qué solemos añadir **marcas** a las estructuras de datos de grafos? **Justificar** brevemente.

Respuesta seleccionada: Agregar marcas a los vertices de un grafo sirve para ayudar/implementar el recorrido de un grafo. Por ejemplo, al comenzar a recorrer un grafo ponemos las marcas de todos los vertices como NO_VISITADOS y a medida que se los recorre se va dejando una marca de VISITADO para diferenciarlos y no volver a visitar un vertice en mas de una oportunidad.

En los árboles generales podemos implementar los nodos con **hijo por izquierda, hermano por derecha**. ¿Qué ventajas tiene este enfoque? **Justificar** brevemente.

Rta.: la ventaja del enfoque de "hijo por izquierda, hermano por derecha" radica en que evita usar estructuras secundarias como listas, que ocupan memoria y requieren ser recorridas.

¿Cuándo conviene recorrer un **árbol** con BFS (breadth first search) en lugar de hacerlo con DFS (depth first search)? **Justificar** brevemente.

Rta.: en general preferimos BFS cuando queremos recorrer el árbol por niveles; por ejemplo, en juegos de ajedrez. Dimos unos ejemplos de esto en clase!

En todos las estructuras de datos vistas en la materia, ¿por qué es importante la idea de **posición**? **Justificar** brevemente.

Rta: la posición es importante para poder **recorrer** las estructuras de datos (tu respuesta menciona ordenar, no recorrer).