Sé feliz, ¿Quién te lo impide?

CONTENIDO

1. VERDADERO O FALSO

1.1 V/F: 'Si una función fue declarada con *virtual void foo()=0;* entonces no es necesario proveer una definición de *foo* en el programa.'

VERDADERO. La puedo declarar y nunca instanciar un objeto de esa clase. No necesariamente tengo que implementarla.

1.2 V/F: 'El enlazado interno con *static* del lenguaje C es bastante parecido al uso de *private* en C++'

VERDADERO. Tanto static como private permiten que solo un modulo especifica tenga acceso a una variable o función.

1.3 V/F: 'En C++, un programa produce fallos en forma segura si la asignación de memoria dinámica falla'

FALSO. Se produce un fallo de forma segura si la asignación de memoria dinámica fue realizada con new o new[] ya que se lanza una excepción cuando no se pudo hacer la tarea. En caso de realizar dicha asignación con malloc, calloc o realloc, el programador deberá realizar una validación para asegurarse de que se pudo realizar la tarea.

1.4 V/F: 'El uso de excepciones puede ser dificultoso porque en el momento de desenrollarse el stack los destructores de los objetos locales no son llamados y eso puede causar memory leaks'

FALSO. Al ser lanzada una excepción la misma 'burbujea' desenrollando el stack hasta que algún metodo o función la atrape. Además, al desenrollar stack se llama a los destructores de las clases. Esto evita memory leaks. Sin embargo, al ser atrapada la excepción el programa continua desde ese punto y si se liberó memoria dinámica, podría tratar de acceder a esa memoria que ya no existe.

1.5 V/F: 'Si una clase requiere un destructor para liberar algún recurso, entonces es muy probable que también requiera un constructor copiador y una sobreescritura de operador ='

VERDADERO. 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 pasa con el operador de asignación.

1.6 V/F: 'La herencia refiere a como código idéntico puede producir resultados diferentes, dependiendo del tipo de objeto concreto que se está procesando'

FALSO. La herencia apunta a programar objetos distintos que cumplen con el criterio 'es un'. Permite usar la programación por diferencia y así escribir menos código. (ver 2.22).

1.7 V/F: 'Para recorrer grafos en forma eficiente conviene siempre representarlos con lista de adyacencia'

FALSO. Para recorrer un grafo *disperso*, se usan las *listas de adyacencia* porque cada nodo enumera sólo los vecinos existentes. La complejidad para recorrerlo es $O(V+E)$. Si el grafo es denso, conviene usar matrices de adyacencia ya que al tener un grafo denso va a haber muchas uniones entre nodos, por lo cual no va a haber campos vacíos que desperdicien memoria. Por otro lado, la matriz evita tener que apuntar a los punteros next de las listas de adyacencia.

1.8 V/F: 'La *búsqueda por interpolación* es siempre computacionalmente eficiente'

FALSO. Si las claves no están uniformemente distribuidas la complejidad es $O(n)$. En cambio si las claves están uniformemente distribuidas la complejidad es $O(log(log(n)))$.

1.9 V/F: 'Para una tabla hash de tamaño N, la función h(k) = (k + rand()) % N es una buena función hash'

FALSO. Está función no sirve, ya que para cada clave no se produce un valor determinístico debido a que la función rand no devuelve siempre el mismo número.

1.10 V/F: 'Los miembros privados de una clase se heredan pero no son accesibles por las clases derivadas'

VERDADERA. Es por eso que si se quiere contar con métodos y datos miembros que no puedan ser utilizados desde el 'exterior' de la clase pero que si sean accesibles para clases derivadas se debe usar protected:

2. JUSTIFICAR BREVEMENTE

2.1 En C++, ¿cuándo conviene usar punteros y cuándo referencias?

Conviene usar referencia cuando no buscamos cambiar la referencia al objeto y para no tener que usar $*$ o -> y así tener un código más claro.

2.2 ¿Qué es un constructor por defecto implícito y que hace?

El constructor por defecto *implícito* es el que provee el compilador cuando no se define un constructor por defecto. Inicializa todos los datos miembros con sus constructores por defecto.

(Obs: Un constructor implícito se define si y solo si el usuario no definió ninguno de forma explícita).

2.3 ¿Cuándo conviene usar programación orientada a objetos (POO) y cuándo programación imperativa?

La POO es recomendable cuando se tienen programas con un alto nivel de complejidad, mientras que para la imperativa cuando la complejidad no es tan elevada y el programa se puede desarrollar de forma secuencial.

2.4 ¿Cuándo conviene usar sobrecarga de funciones y cuándo plantillas de funciones?

Conviene usar plantillas de funciones cuando la función realiza la misma tarea sin importar el tipo de dato (Por ejemplo buscar el mínimo entre dos valores). Por otro lado se usa sobrecarga de datos, cuando la función a utilizar depende del tipo de dato (por ejemplo la impresión de un int comparada con la impresión de un string).

2.5 ¿Cuándo conviene implementar un árbol binario con un arreglo?

Conviene implementarlo cuando el árbol es completo o casi completo. Sino no lo fuera no sería eficiente en memoria debido a los campos vacíos.

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

Al elegir un pivote se busca que los subgrupos tengan una cantidad similar de elementos, y al elegir el elemento central estadísticamente se minimizan los 'worst case scenarios'. En general elegir el elemento central conviene para datos parcialmente ordenados, para otros casos elegirlo de manera aleatoria también es buena opción.

2.7 En una estructura de datos de tipo lista, ¿el comienzo de la lista coincide siempre con el primer elemento?

No siempre, si se usan *headers* en comienzo de la lista NO coincide con el primer elemento. Por ejemplo, listas simplemente enlazadas.

2.8 ¿Cuándo conviene usar listas ordenadas y cuando listas desordenadas?

Las listas ordenadas se usan cuando me interesa tener objetos ordenados por una clave. También me puede interesar insertar y/o eliminar objetos de una posición específica.

Las listas desordenadas se usan cuando no nos interesa el orden particular de los elementos.

2.9 ¿Cuándo conviene usar listas con arreglo y cuándo listas enlazadas?

Para insertar o eliminar elementos conviene lista enlazada con complejidad $O(1)$, en cambio la lista con arreglo tiene complejidad $O(n)$ ya que se deben mover todos los elementos a la derecha del insertado/eliminado. Para acceder a los elementos de forma arbitraria conviene lista con arreglo con complejidad $O(1)$, en cambio la lista enlazada tiene complejidad $O(n)$ (se deben recorrer los elementos previos al buscado).

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

El problema con el uso de los algoritmos estándar es que no garantizan el balanceo del árbol. Es decir que la búsqueda podría volverse muy ineficiente. La complejidad pasaría de $O(log(n)$ a $O(n)$

2.11 ¿Por qué solemos añadir marcas a las estructuras de datos de grafos?

Se agregan marcas ya que es importante saber si un nodo fue visitado, para que los recorridos no realicen ciclos y visiten todas las componentes conexas.

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

El BFS se utiliza para recorrer el árbol en anchura, el DFS se usa para realizar una búsqueda profunda. Se prefiere el BFS cuando le querés dar prioridad a algo cerca del nodo raíz, de lo contrario se usa DFS.

2.13 En búsqueda por hashing, ¿por qué la función hash debe mezclar los bits cuanto mejor posible?

Para evitar cualquier correlación entre valores que puedan producir colisiones.

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

Esta representación es eficiente en la inserción y eliminación de nodos, y evita implementar estructuras secundarias como listas que ocupan menos memoria.

2.15 En todas las estructuras de datos vistas en la materia, ¿por qué es importante la idea de posición?

La posición es importante para poder recorrer las estructuras de datos.

2.16 ¿Cuál es la diferencia entre la sobrecarga y la sobreescritura de métodos?

Cuando se hace sobrecarga, se definen nuevos métodos. En cambio, cuando se sobreescribe un método, se 'pisa' el método con una nueva definición del mismo.

2.17 Tanto las listas doblemente enlazadas como los árboles binarios poseen nodos con dos punteros a nodo. Siendo sus estructuras de datos iguales, ¿en qué se diferencian?

En el árbol binario, cada nodo tiene punteros a nodos hijos, en cambio en la lista doblemente enlazada tiene un puntero al nodo anterior y otro puntero al nodo siguiente.

En forma más clara: las listas doblemente enlazadas representan una estructura **lineal**, mientras que los árboles binarios representan una estructura **jerárquica**.

2.18 Para implementar una búsqueda de palabras en ASCII, ¿qué es más eficiente: una búsqueda binaria o un trie?

Para implementar una búsqueda de palabras en ASCII, conviene usar tries, ya que es útil a la hora de realizar búsqueda rápidas en cadenas de texto. Por ejemplo si tengo 14 caracteres, la busqueda se realiza en 14 pasos. En un nodo hay una combinación de letras, si no esta la que busco, avanzo al otro nodo y así sigo hasta encontrar lo que busco.

O sea, si n es el largo de la palabra, un trie requiere, en promedio, $O(n)$ comparaciones.

Y si m, la cantidad de elementos, la búsqueda binaria requiere, en promedio $O(nlog(m))$.

2.19 ¿Cuál es el algoritmo de búsqueda más eficiente?

Hay que tener en cuenta todo, como podrían estar ordenados los datos, si están distribuidos uniformemente, complejidad de memoria y/o cálculo y el hashing.

2.20 Desde el punto de vista de herencia, ¿Cuando se dice que un objeto 'es un' o está 'contenido en'?

Conviene usar la herencia cuando un objeto de clase A *es* a su vez *un* objeto de clase B. Por ejemplo, un dálmata (clase A) es un perro (clase B).

Por otra parte, si se puede decir que un objeto de clase A *esta contenido* en un objeto de clase B no conviene usar herencia. Por ejemplo, un perro (clase A) está en un auto (clase B).

2.21 ¿Cómo implementaría un árbol para una organización jerárquica de una empresa?

Es como el ejemplo de la municipalidad que dio en clase. Un nodo tiene mayor jerarquía que sus hijos y menor jerarquía que su padre. A su vez, para una mayor eficiencia conviene implementarlo con la forma 'hijo por izquierda, hermano por derecha' (ver 2.14).

2.22 Diferencias entre encapsulamiento y herencia

El encapsulamiento se refiere a agrupar funciones y datos miembro en un mismo objeto con una interfaz de modo que abstrae al usuario del funcionamiento interno de la clase. En cambio, 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.

2.23 ¿Cuál es el peligro de hacer sobrecarga de operadores?

Se puede tener código confuso y problemas con la precedencia, la concatenación o la auto-asignación.

2.24 Diferencias entre listas doble y simplemente enlazadas ¿Por qué no tendría sentido usar un nodo trailer con listas simplemente enlazadas?

Las listas doblemente enlazadas pueden ser recorridas de atrás para adelante y viceversa, los nodos header y trailer simplifican mucho la lógica ya que marcan el principio y el final. En cambio las listas simplemente enlazadas solo pueden recorrerse hacia adelante, y se terminan cuando el puntero vale NULL. Justamente por esto no tiene sentido un nodo trailer en una lista simplemente enlazada.

2.25 ¿Por qué se prefiere usar vector en vez de arreglos de C++?

El vector no me pide una cantidad constante de elementos. Sin embargo, puedo establecer un tamaño fijo y usar al vector como un arreglo, sin embargo esto puede reducir la comprensión del código.

2.26 ¿Cómo se ejecuta la excepción? Las tres etapas involucradas.

En el bloque **try** se verifican los posibles errores, cuando ocurre un error se lanza la excepción con **throw** cortando la ejecución del try. Y el **catch** propio de la excepción, la 'atrapa' y hace lo que tenga que hacer.

2.27 ¿Qué algoritmo de ordenamiento conviene usar si no conocemos nada?

Usaría Quicksort, ya que los que 'son más eficientes' me piden condiciones iniciales para ser mejores. En cambio con Quicksort puedo elegir un pivote del medio y va a tener $O(nlog(n))$ de cálculo en el caso promedio y $O(log(n))$ de memoria en todos los casos. (ver 2.6).

2.28 ¿Cuál es el problema de la herencia múltiple?

Problema del diamante, pasa cuando una clase hereda de otras clases, que a su vez heredan de una clase base.

2.29 Para implementar una cola, ¿Conviene una lista o un arreglo?

Usaría una lista, ya que la cantidad de elementos no es fija y puedo ir agregándolos generalmente en back, además en una cola no me interesa acceder a un elemento en una posición arbitraria (Es la única ventaja de un arreglo frente a una lista).

2.30 En los árboles binarios de búsqueda, ¿qué problemas se dan cuando el árbol no está balanceado?

Cuando un árbol está balanceado, la búsqueda es eficiente, de complejidad $O(log(n))$. Si un árbol no ésta balanceado la búsqueda puede volverse ineficiente porque puede dejar de ser $O(log(n))$. En el peor de los casos, cuando la estructura del árbol se asemeja a una lista la complejidad es $O(n)$.

2.31 En C++, ¿por qué alguien se molestaría en declarar un destructor virtual?

Para asegurar una 'correcta destrucción' de un objeto de una clase hija o derivada cuando hablamos de herencia. Ya que se podría destruir un objeto de una clase derivada usando un puntero del tipo usado en la clase base, lo cual podría generar complicaciones.

2.32 En C++, ¿por qué no tiene sentido que los destructores tengan parámetros?

No tiene sentido ya que el destructor se encarga de destruir cualquier tipo de recurso utilizado.

Los objetos pueden ser destruidos porque habían sido creados en el stack y se salieron de alcance; habían sido creados en el stack y una excepción desenrolló el stack; fueron creados en memoria dinámica y son destruidos con delete o delete[].

2.33 En C++, ¿cuál es la causa más frecuente por la que alguien se molestaría en sobreescribir los operadores « y » de una clase?

La causa más frecuente es para poder usar flujos de entrada y/o de salida. Es decir, puede servir para serializar el objeto (obtener una representación de texto del objeto que, por ejemplo, puede ser guardada a disco para almacenar su estado, y luego leída para recuperarlo).

2.34 ¿Cuándo conviene implementar un grafo con una lista de adyacencia, con una matriz de adyacencia o con un conjunto de arcos?

Cuando el grafo es disperso ($|E|$ del orden de $|V|$), conviene usar una lista de adyacencia o un conjunto de de arcos, ya que requieren $O(|V|+|E|)$ de memoria, sin embargo hay que tener en cuenta

que si es necesario hacer recorridos, el conjunto de arcos es muy ineficiente porque no representa explícitamente los vecinos de un nodo.

Si el grafo es denso ($|E|$ del orden de $|V|^2$), conviene usar matriz de adyacencia ya que requieren $O(|V|2)$ de memoria.

2.35 ¿Cuándo conviene usar una look-up table y cuándo una tabla hash?

Conviene usar look-up table cuando las claves están densamente distribuidas, si no ocurre esto, la tabla será muy ineficiente en memoria. Es por esto que cuando las claves son dispersas conviene usar tabla hash, ya que las claves 'se concentran' en una tabla relativamente densa, se debe tener en cuenta que el uso de la tabla hash mejora la complejidad en memoria, reduce la complejidad en cálculo y puede haber colisiones.

2.36 En algoritmos y estructuras de datos disponemos de una multitud de algoritmos de ordenamiento. ¿Para qué solemos ordenar los datos?

Nos interesa ordenar los datos para de esta forma poder acceder o buscar datos de manera eficiente en calculo.

2.37 En C++, ¿por qué la biblioteca estándar STL está implementada con templates?

La STL estructuras de datos **genéricas**. Es por esto que se usan templates, ya que estos permiten reutilizar código pero con diferentes tipos de datos.

2.38 En algoritmos y estructuras de datos solemos usar la notación O grande para determinar el consumo de recursos en función de una variable, típicamente la cantidad de elementos n. ¿Es una medida exacta?

No es una medida exacta, la notación O grande da una cota superior al valor exacto. Solo se indica un 'orden' para poder comparar entre distintos algoritmos o estructuras de datos.

2.39 En C++, ¿cuál es el peligro de confundir delete con delete[]?

No se liberará la memoria correctamente (depende de la implementación de new[] y delete del sistema operativo).

2.40 Con datos aproximadamente pre-ordenados, ¿es mejor quicksort o insertion sort?

Cuando los datos están aproximadamente pre-ordenados, insertion sort es $O(n)$ en ordenamiento; quicksort es $O(nlog(n))$ en el caso promedio, y $O(n^2)$ en el peor caso.

2.41 ¿Por qué es mejor usar SQL en lugar de hacer las cosas nosotros mismos?

Delega el problema del acceso a datos; es fácil de adaptar a las necesidades de un proyecto; es estándar (lo cuál significa que es fácil cambiar una base de datos SQL por otra); es mantenido por una comunidad de programadores (lo cual significa que se actualiza continuamente: se añaden nuevas características, se soportan nuevos entornos de programación, se resuelven errores en el código, y se dispone de una comunidad que puede dar ayuda); y está muy probado (fue sometido a muchos tests de software).

2.42 ¿Para qué sirven los índices de SQL?

Permiten mejorar la velocidad de las operaciones que involucran ciertos campos de una base de datos.

2.43 En C++, ¿cómo eliminarías en forma eficiente los duplicados de una lista de tipo array? Responde en palabras, no en código.

Para buscar dos repetidos, necesitas dos for, que son $O(n^2)$. El problema se puede resolver sin embargo en forma mucho más eficiente. Si interesa preservar el orden de los elementos, podemos ordenar los elementos con algún algoritmo de ordenamiento (por ejemplo quicksort), y luego eliminar repetidos contiguos. El ordenamiento más rápido lleva $O(nlog(n))$. Eliminar los elementos contiguos lleva $O(n)$. Esta propuesta es, por tanto, $O(nlog(n))$. Si no interesa preservar el orden de los elementos,

podemos usar unordered_set. Insertar los elementos lleva $O(n)$. Antes de insertar un elemento, es posible comprobar si el elemento ya está, cosa que, para todos los elementos, lleva $O(n)$. De este modo no insertamos duplicados. Finalmente recorremos el conjunto y construimos una nueva lista, lo cual lleva $O(n)$. Esta propuesta es, por tanto $O(n)$. La segunda propuesta es aproximadamente $log_2(n)$ más rápido que la primera.

2.44 En C++, ¿cómo pueden pasarse parámetros al constructor de la clase base desde el constructor de la clase derivada?

Se pueden utilizar listas de inicialización. En la clase derivada se 'completa' una lista con los datos que pide el constructor de la clase base.

2.45 En C++, supongamos que tenemos un arreglo de número enteros muy grande. ¿Cuál es la forma más eficiente de determinar los 50 números más altos? Responde en palabras, no en código.

El problema puede resolverse en forma más eficiente con una lista (vector o list). Iteramos sobre todos los elementos del arreglo, y vamos insertando los elementos en el lugar que corresponde. Si la lista supera 50 entradas, la truncamos a 50 entradas. De este modo resultarán los 50 elementos más altos en orden en esta lista. El algoritmo tiene complejidad $O(n)$ ya que 50 (el tamaño máximo de la lista) es mucho menor a n, y el costo computacional de insertar en la lista es despreciable.

2.46 Siendo que no es posible crear objetos a partir de clases abstractas de C++, ¿por qué alguien se molestaría en definir una clase abstracta?

Al definir una clase abstracta, se puede obligar al programador a definir los métodos virtuales puros si es que desea crear objetos.

2.47 ¿Por qué solemos añadir marcas a los grafos?

Solemos añadir marcas para saber si un nodo fue visitado, y para que los recorridos no realicen ciclos y visiten todas las componentes conexas.

2.48 En C++, cuando usamos excepciones, ¿es obligatorio definir un gestor de excepciones?

Si bien no es obligatorio, ya que C++ no exije que se definan gestores de excepciones, si no se definen no tendría sentido usar excepciones, porque no serían atrapadas y detendrían la ejecución del programa

3. MULTIPLE CHOICE

3.1 *this* sirve para...

- referir, desde un método, al objeto para el que el método ha sido llamado.
- desambiguar entre parámetros y datos miembro de igual nombre.
- poder pasar a otros objetos, desde un método, una referencia al objeto propio.

3.2 Los destructores...

tienen por fin liberar los recursos reservados por un objeto.

3.3 ¿Por qué es importante escribir código independiente de la plataforma?

- No siempre es importante. Existen casos en los que no tiene sentido hacerlo.
- Porque permite abarcar a un público más grande con relativamente poco esfuerzo.

3.4 La sobrecarga de métodos...

suele aplicarse cuando los métodos realizan tareas similares.

3.5 En C++, ¿por qué es preferible usar el mecanismo de salida a consola de C++, a usar printf?

Porque asegura la seguridad de los tipos de datos.

3.6 Los templates de clase...

Permiten reutilizar el mismo código con diferentes tipos de datos.

3.7 ¿A qué tipo de eficiencia debemos prestarle más atención?

Debemos atender el cuello de botella de nuestro problema particular.

3.8 ¿Por qué debemos factorizar el código?

- Para evitar redundancia que, en caso de error, debamos revisar repetidamente.
- Para evitar errores, ya que simplificará la revisión de código.
- Para escribir código más claro, fácil de entender.
- Para evitar, a largo plazo, trabajo innecesario.

3.9 La sobrecarga de operadores de C++ permite...

usar los operadores de C++ con clases definidas por el usuario.

3.10 Cuando un programa posee una interfaz de usuario, conviene separarlo en modelo, vista y controlador porque esto...

- Fomentar la reutilización de código
- ayuda a reducir las dependencias entre las partes del código.

3.11 ¿Por qué debemos añadir comentarios en forma eficiente?

- Para documentar eficientemente el funcionamiento del software.
- Para facilitar la lectura del código.

3.12 ¿A qué tipo de eficiencia debemos prestarle más atención?

Debemos atender el cuello de botella de nuestro problema particular.

3.13 En C++, ¿cuál es la diferencia entre *delete* y *delete[]*?

delete se usa para liberar un solo objeto, mientras que *delete[]* se usa para liberar arreglos de objetos.

3.14 En una clase de C++, siempre conviene...

colocar la interfaz en la sección public.

3.15 ¿Cuál es una de las razones por las que debemos cuidar la indentación en C++?

Para distinguir los comentarios del código.

3.16 En las clases de C++, el objetivo de los métodos virtuales puros es:

- Impedir que el programador pueda crear objetos de esa clase.
- Declarar una interfaz.
- Declarar una clase abstracta.
- Obligar al programador a implementar el método virtual puro en las clases derivadas si desea crear sendos objetos.

3.17 ¿A qué nos referimos cuando hablamos de herencia múltiple en C++?

A derivar una clase derivada de múltiples clases base.

3.18 ¿Qué debemos colocar en un bloque try?

Sentencias que pueden causar excepciones, y sentencias que pueden ser salteadas en caso de una excepción.

3.19 La herencia es un mecanismo que permite definir clases...

que heredan todos los métodos y datos miembro de otra clase.

3.20 Respecto a excepciones ¿Cuál(es) de las siguientes afirmaciones es verdadera?

- En C++ es posible impedir que una función o método lance una excepción.
- new nunca devuelve NULL.
- Es preferible lanzar excepciones por copia y atraparlas por referencia.

3.21 Cuando pasamos un objeto por copia a una función...

la copia se inicializa con el constructor copiador.

3.22 ¿Cuándo debemos sobreescribir el operador de asignación?

- Cuando la clase gestiona memoria dinámica en datos miembros.
- Cuando la clase crea objetos con new y los gestiona como datos miembro.

3.23 En relación a los bloques catch, ¿cuál(es) de las siguientes afirmaciones es verdadera?

- Deben ubicarse inmediatamente después del bloque try.
- Admiten lanzar nuevas excepciones con throw.

3.24 Una clase abstracta de C++ puede contener:

métodos no-virtuales, métodos virtuales y métodos virtuales puros.

3.25 ¿Qué debemos colocar en un bloque try?

Sentencias que pueden causar excepciones, y sentencias que pueden ser salteadas en caso de una excepción.

3.26 En C++, para enlazar un método en forma dinámica es necesario:

declarar el método virtual.

3.27 ¿Qué ventajas tienen las excepciones?

- Pueden capturarse en cualquier parte del call stack.
- Un programador puede manejarlas dentro de sus métodos y funciones, o puede delegarlas a terceros.
- Eliminan el código de manejo de errores de la principal línea de ejecución del programa.

3.28 Si heredamos una clase con la declaración: *class A : public B, public C* ¿En qué orden se ejecutan los constructores?

B(); C(); A();

3.29 En un bloque try-catch anidado en otro bloque try-catch, si se ejecuta el bloque catch interior y no se generan nuevas excepciones, entonces:

La ejecución continúa en el código después del bloque try-catch interior.

3.30 El recorrido BFS en un árbol binario de búsqueda...

permite recorrer el árbol nivel a nivel.

3.31 Para resolver las colisiones de las tablas hash...

- podemos aplicar algúna política de resolución de colisiones de hashing cerrado.
- podemos usar un árbol binario de búsqueda auto-balanceado.

3.32 ¿Cuál es el mejor tipo de datos para almacenar elementos en orden por una clave?

Una cola de prioridad con lista con arreglo.

3.33 Siendo n el tamaño de la tabla hash, y k una clave, ¿cuál(es) de la(s) siguiente(s) son buenas funciones hash?

- h(k) = (k * k) % n
- h(k) = k % n

3.34 ¿Cuál es el mejor tipo de datos para implementar streaming de vídeo desde Internet?

Una cola con lista doblemente enlazada.

3.35 En un grafo, ¿cuál es el mejor algoritmo para encontrar un nodo por su clave?

Iterar sobre la lista de nodos.

3.36 ¿Cuál es el mejor tipo de datos para almacenar una lista de reproducción de música?

Una lista doblemente enlazada.

3.37 Los árboles binarios...

permiten representar una serie de decisiones binarias.

3.38 ¿Por qué debemos añadir marcas a los grafos?

Para que los recorridos no realicen ciclos y visiten todas las componentes conexas.

3.39 Si un grafo es denso y debemos recorrerlo frecuentemente...

en general, conviene implementarlo con una matriz de adyacencia.

3.40 ¿Cuál es el mejor tipo de datos para realizar un recorrido DFS de un árbol?

Una pila con deque.

3.41 Los índices de SQL:

Permiten acelerar las búsquedas.

3.42 Los archivos .csv:

- Permiten representar campos con diferentes tipos de dato.
- Permiten almacenar tablas de SQL.
- Permiten almacenar datos genéricos dispuestos en filas y columnas.

3.43 ¿Por qué nos interesaría ordenar una lista con arreglo?

- Para poder encontrar elementos en un rango de valores en forma eficiente.
- Para poder hacer búsqueda binaria.

3.44 SQL es:

- Un lenguaje para realizar peticiones a bases de datos.
- Una forma de definir las relaciones entre tablas.
- Un formato de archivo para archivar bases de datos.

3.45 En general selection sort peor es que quicksort. En cierta situación, sin embargo, podríamos preferir selection sort. ¿Por qué?

- Porque puede requerir menos memoria.
- Porque puede requerir menos swaps.

3.46 Radix sort...

- requiere claves que puedan mapearse a números enteros.
- es una forma iterativa de pigeonhole sort.
- requiere mucha memoria, comparable a la cantidad de datos que hay.
- puede ser más rápido que quicksort.

3.47 En general insertion sort es peor que quicksort. En cierta situación, sin embargo, podríamos preferir insertion sort. ¿Por qué?

- Porque puede requerir menos memoria.
- Porque puede requerir menos comparaciones.

3.48 Para un gran conjunto de datos, cuyas claves son números enteros, sin repetir, y distribuidos en forma densa (o sea, en un rango de valores comparable a la cantidad de datos), ¿cuál(es) algoritmos son eficientes para ordenar los datos?

- Pigeonhole sort
- Radix sort

3.49 Las bases de datos relacionales:

- Permiten realizar búsquedas que involucran múltiples tablas.
- Permiten definir campos que refieren a otras tablas.

3.50 ¿Cuando falla la búsqueda por interpolación lineal que vimos en clase? (o sea, la complejidad computacional deja de ser O(log(log(n))))

Cuando las claves no están uniformemente distribuidas

3.51 La complejidad computacional de peor caso de quicksort es peor que la de merge sort. ¿Por qué solemos preferir en la práctica quicksort?

Porque requiere menos memoria.

3.52 ¿Cuándo falla la implementación de quicksort que vimos en clase? (o sea, la complejidad computacional deja de ser O(n log(n)))

- Cuando los datos vienen pre-ordenados de menor a mayor.
- Cuando los datos vienen pre-ordenados de mayor a menor.
- Cuando el pivote no particiona los datos en sub-grupos de tamaño similar.

Si hay algún error groso o hay preguntas interesantes para agregar, manda mail.

Juan Martin - juanmrodriguez@itba.edu.ar

Cristian - cmeichtry@itba.edu.ar

Y como diría Marc (y la portada), espero que seas feliz mientras lees esto.