Estructuras de Datos

Grado en Ingeniería en Informática

Examen de Mayo de 2013

Nombre: Grupo:

Pregunta 1:

Contestar V/F las siguientes cuestiones, teniendo en cuenta que una respuesta incorrecta invalida una correcta.

[V] Una cola con prioridad es siempre una eedd lineal, de acceso secuencial y dinámica.

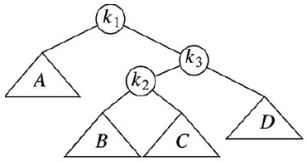
[V] Un patrón de clase instanciado para el tipo T=int puede que no compile al instanciarlo para T=MiClase

[F] Si la clase A tiene como atributo un vector dinámico de objetos tipo B entonces entre ambas clases debe existir una relación de composición.

[F] Un dato almacenado en un vector dinámico, que no sufre ningún tipo de modificación, puede que cambie su posición de memoria. Sin embargo no ocurre lo mismo si se insertara en una lista enlazada.

[F] Si P y Q son dos conjuntos de bits de tamaños 7 y 15 respectivamente, entonces P.interseccion(Q) puede tener tamaño 10.

[F] Dado el siguiente árbol B, una inserción de un dato en los subárboles B o C implica una rotación doble, primero a izquierdas y luego a derechas.



[V] En una tabla hash que contiene casillas vacías y disponibles, la búsqueda no para cuando se encuentra una casilla disponible.

[F] Averiguar los ejes de entrada a un nodo en un grafo dirigido implementado mediante listas invertidas es más eficiente que implementado sobre una matriz de adyacencia.

[F] Si un nodo de un árbol B de orden 21 se queda con 10 elementos, entonces el nodo siempre desaparece reubicando sus datos en los nodos hermanos.

[V] Para almacenar millones de estrellas se ha optado por utilizar una malla regular. Para implementar la función de búsqueda de una estrella conocida su ascensión y declinación sólo es necesario visitar una casilla de la malla.

Pregunta 2:

Sea una estructura de conjuntos disjuntos con los elementos 1 a 15. Indicar de que manera evolucionaría internamente la estructura de datos tras las siguientes operaciones (usar todas las optimizaciones y mejoras estudiadas):

unir(1, 2), unir(3, 8), unir(6, 7), unir(6, 9), unir(4, 5), unir(1, 3), unir(6, 4), unir(6, 1), unir(10, 6), buscar(8), buscar(5)

Pregunta 3:

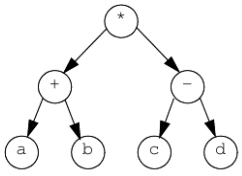
1. Definir la interfaz de la clase *ConjuntoDisjunto* que representa a una colección de conjuntos disjuntos con la estructura que hemos estudiado en la teoría. Implementar la función de búsqueda de un dato en el conjunto:

   ```
   int ConjuntoDisjunto::busca(int dato);
   ```

2. Un árbol puede representar una expresión matemática. Para ello se supone que todos los operadores son binarios y están en los nodos internos del árbol, mientras que los nodos hoja poseen los operandos. Implementar la operación *evalua()* de la clase *ArbolExpresion* con la siguiente definición (se da por hecho que el árbol está bien construido):

```
class NodoExpresion {
  char operador; // Valores *, -, +, / o " "
  float operando; //válido cuando operador = " "
  NodoExpresion *op1, *op2;
};

class ArbolExpresion {
  NodoExpresion *raiz;
public:
  ...
  float evalua();
  ...
};
```



Pregunta 4:

Implementar de manera simplificada el funcionamiento del clásico juego *Breakout* de Atari. Como es conocido, el juego trata de eliminar una serie de ladrillos o bloques situados en la parte superior de la pantalla haciendo rebotar una bola en

una pala que controla el jugador, situada en la parte inferior de la pantalla. El diagrama UML adjunto representa las clases involucradas:

- *Bola*: representa la bola del juego. Tiene una posición determinada *(x, y)* y un desplazamiento en una dirección (dx, dy). Tanto *dx* como *dy* pueden valer *-1* o *+1*. La operación *actualizar()* actualiza la posición de la bola en cada paso del juego, sumando los desplazamientos a las coordenadas de la bola. Las operaciones *reboteX()* y *reboteY()* invierten los desplazamientos correspondientes.
- *Pala*: representa la pala del juego. Está situada en la posición *(x, y)* y tiene dos operaciones para moverse un paso a la izquierda o derecha atendiendo a las ordenes del usuario. La pala tiene una longitud fija de 4 posiciones y nunca puede salir de la pantalla por la izquierda o derecha de la pantalla. La operación *choca()* devuelve true si colisiona con la bola (no implementar).
- Ladrillo: representa un ladrillo del muro. Tiene una posición determinada en la pantalla *(x, y)* y una longitud fija de dos posiciones. La operación *choca()* devuelve true si colisiona con la bola (no implementar).
- Breakout: es el controlador del juego. La operación *iniciar()* prepara el escenario de juego para la pantalla de 50x50 posiciones, de la siguiente manera (ver captura de pantalla):
  - La pala se situa en la parte baja, en el centro de la pantalla *(22, 2)*.
  - La bola se inicia con $y = 25$ y x, dx y dy generados aleatoriamente dentro de sus valores posibles[1].
  - Se crean 8 muros de ladrillos de lado a lado de la pantalla en las filas 49, 48, 47, 46, 43, 42, 41 y 40.
  - La puntuación se inicia a 0.
- La operación *juego()* realiza un paso del juego de la siguiente manera:
  - Se comprueba si la bola choca con la pala, en cuyo caso se produce un rebote en y de la misma.
  - Se comprueba si la bola choca con las paredes laterales $(x = 0, x = 49)$ en cuyo caso se produce un rebote en x de la misma. Si la bola choca con la pared superior $(y = 49)$ se produce un rebote en y.
  - Se comprueba si la bola choca con algún ladrillo, en cuyo caso el ladrillo se elimina, se produce un rebote en y de la bola y se suma un punto a la puntuación. Si no quedan ladrillos, la operación devuelve el código NIVEL_SUPERADO (valor *1*).
  - Si la bola sale de la pantalla por la parte inferior, el jugador ha perdido y se devuelve el código JUGADOR_PIERDE (valor *2*).
  - Actualizar la posición de la bola. La operación termina con código JUEGO_EN_CURSO (valor *0*).
- Las operaciones *teclaIzq()* y *teclaDer()* mueven la pala a la izquierda o derecha respectivamente.

Implementar las clases descritas, exceptuando las operaciones *choca()* de *Ladrillo* y *Pala*. Utilizar una estrategia o estructura de datos que minimice el número de ladrillos en la comprobación de colisiones de la bola con el muro.

[1] para generar un número aleatorio entre *0* y n - *1* puede hacerse mediante: *random() % n*