ESTRUCTURA DE DATOS NO LINEALES

huge RECOPILACIÓN DE PREGUNTAS DE TEORÍA

AUTOR/A:SHOWE0, SHOWE1 Y SHOWE2

Índice general

Parte I

Práctica 6

1. Preguntas de Árboles

1.1. Preguntas de Árboles

¿Por qué surgen los arboles?

Porque nos permiten crear una jerarquía y en algunos casos poder realizar búsquedas en Orden logaritmico `O(log n)`.

¿En qué situaciones es conveniente utilizar un vector de posiciones relativas?

Sabemos que un vector de posiciones relativas está relleno por los elementos nulos, por tanto, cuanto más nodos contenga el árbol, es decir, más bajo sea su nivel, menos memoria será desaprovechada y más eficiencia espacial habrá.

¿Cuántos tipos de recorrido de árboles en anchura existen?

Solamente hay un recorrido en anchura, que comienza en la raiz del árbol, luego a su hijo izquierdo y luego al derecho, y así sucesivamente con todos los nodos del árbol. No confundir con que hay 2 tipos de implementaciones, iterativa y recursiva, pero el recorrido es el mismo.

¿Cuántos tipos de recorridos de árboles en profundidad existen?

Hay 3 tipos de recorridos:

- **Preorden:** Se procesa primero la raíz, luego se vista el hijo izquierdo y los hermanos
  - **Procesar raiz**
  - **Recorrer hijo izquierdo**
  - **Recorrer Hijo derecho (Abin) / hermanos (Agen)**
- **Inorden:** Se visita primero el hijo izquierdo, luego se procesa la raíz y por último los hermanos.
  - **Recorrer hijo izquierdo**
  - **Procesar raiz**

    - **Recorrer Hijo derecho (Abin) / hermanos (Agen)**

- **Postorden:** Se visita primero el hijo izquierd, luego los hermanos (En el caso de que sea Agen) y por último la raiz.

    - **Recorrer hijo izquierdo**
    - **Recorrer Hijo derecho (Abin) / hermanos (Agen)**
    - **Procesar raiz**

¿Qué condiciones tiene que cumplir los elementos de un árbol para poder realizar las búsquedas con un coste menos que O(n)?

La condición a cumplir es que el árbol esté lo más equilibrado posible y sus elementos estén ordenados, haciendo que pasemos de un coste `O(n)` a un coste `O(log n)`.

¿Puede reconstruirse un árbol unívocamente dado su inorden?

No, nos hace falta el **inorden** para ver la relación entre los subarboles, es decir, ver cuales son hijos izquierdos y cuales derechos, y el **preorden** o **postorden** para conocer donde está la raiz.

1.2. Preguntas de Abin

¿Por qué las operaciones de Insertar y Eliminar son de O(1) en la representación vectorial o de celdas enlazadas de árboles binarios?

La inserción en un Abin es de `O(1)` porque al tener una variable llamada **numNodos** (numero totales de nodos) y que se corresponde con la posición de último nodo insertado, haciendo que no tengamos que recorrer el árbol, puediendo acceder directamente a la posición donde se insertará el nuevo nodo.

Idem con eliminación, salvo que intercambias el último nodo insertado con el nodo a eliminar y decrementas el **numNodos**.

Nota: El caso de buscar no aplica esta lógica, es de orden `O(n)`.

¿La eliminación de nodos de un árbol binario se puede conseguir con O(1), cuando se utiliza una representación vectorial con índice al padre, hijo izquierdo e hijo derecho?

Si, debido a que la eliminación en un árbol binario con representación vectorial es siempre de coste `O(1)` debido a que intercambiamos el nodo a eliminar con el último nodo insertado en el vector **técnica de compactación**.

Ventajas de la representación de árboles binarios con celdas enlazadas frente a matrices.

OJO Cuando dice matrices: Se refiere a la representación vectorial con índice al padre, hijo izquierdo e hijo derecho.

- **Ventajas de celdas enlazadas:**
  - **Eficiencia espacial:** No se desperdicia memoria, ya que solo se reserva memoria para los nodos que se necesitan. En la matricial se desperdicia memoria ya que se reserva memoria para todos los nodos posibles, sin saber si esta se va a rellenar entera.
  - **Eficiencia temporal:** Las operaciones de inserción y eliminación son de `O(1)`, al igual que la búsqueda. En la matricial la búsqueda es de `O(n)`.

¿Puede construirse de forma única un árbol binario dado, conociendo su preorden y el peso de cada nodo (número de nodos descendientes suyos)?

Con el preorden (o postorden) conocemos la raiz, pero no conocemos los hijos izquierdos y derechos (para eso no hace falta el **inorden**) ya que el peso de los nodos solamente nos dice cuantos hijos tiene pero no nos da información si son hijos izquierdos o derechos.

¿Tiene sentido el concepto de árbol terciario de búsqueda?

Si, tenemos el caso de los árboles B donde un árbol que tiene m = 3 y k = 2, donde m(número maximo de hijos por nodo) y k(número de claves como máximo cada nodo). Este tipo de árboles nos permiten almacenar varios elementos en un mismo nodo, lo que conlleva una reducción en el acceso de memoria secundaria y el tiempo de búsqueda.

Sean A y B dos árboles binarios diferentes, indique si puede ocurrir simultáneamente que: Pre(A) = Post(B) y Pre(B) = Post(A).

Si, puede ocurrir, ya que el recorrido en preorden y postorden de un árbol no nos dice nada sobre la estructura del árbol, es decir, no nos dice si un nodo es hijo izquierdo o derecho de otro nodo. Entonces: puede darse que encontremos los mismos elementos y al recorrerlos nos den igual Preorden y Postorden, pero que la ramificación de los nodos sea diferente.

También se puede dar si los árboles son vacíos.

Las operaciones del TAD Árbol Binario permiten insertar y eliminar hojas pero no nodos internos.

Con insertar se cumple, ya que solo podemos insertar hijos izquierdo/derecho si de antes no lo tiene.

Con eliminar tambien se cumple, ya que como precondición tenemos que el nodo a eliminar tiene que ser hoja, es decir, no tener hijos.

Para destruir un Árbol Binario completo implementado mediante una representación vectorial, no es necesario eliminar los nodos uno a uno en postorden, esto sólo se haría si se trata de un subárbol del mismo.

FALSO: Para destruir un árbol binario completo implementado mediante una representación vectorial, SI es necesario eliminar los nodos uno a uno en postorden, ya como precondición para eliminar un nodo, tenemos que eliminar a sus hijos primero (el nodo en el que estás siempre va a ser una hoja).

La eficiencia espacial de la representación de un Árbol Binario mediante un vector de posiciones relativas será mejor cuantos menos nodos falten en el nivel inferior.

Será mayor cuanto más lleno esté el arbol, es decir, cuanto más nodos tenga, ya que si el árbol está lleno, la eficiencia espacial será máxima, ya que no habrá nodos nulos.

Función contarNodos de un Árbol Binario siempre es de O(n).

FALSO: La función contarNodos de un Árbol Binario en la representación vectorial es de `O(1)`, ya que al tener una variable llamada **numNodos** (numero totales de nodos) y que se corresponde con la posición de último nodo insertado, haciendo que no tengamos que recorrer el árbol.

1.3. Preguntas de Agen

¿Por qué no se puede implementar un árbol general con un vector de posiciones relativas?

Porque no podemos conocer su grado (número máximo de hijos), en la representación con posiciones relativas las relaciones entre los nodos (Padre/Hijos) van en relación con el mismo. Luego si no conocemos su grado no podemos prever la posición que deberían ocupar los hijos.

¿Se podría usar las listas doblemente enlazadas en los árboles generales mediante listas de hijos?

Sí, pero no en nuestro caso, ya que en nuestro TAD no podemos acceder a el hermano izquierdo de un nodo, por tanto no tendría sentido usar listas doblemente enlazadas.

¿Puede determinarse un árbol general a partir del inorden y postorden?

No, necesitamos los 3 recorridos para conocer su estructura, porque llegará un momento en el que no sabemos si ese nodo es hijo izquierdo o hermano derecho.

¿Por qué interesa que los árboles B tengan poca altura?

Porque cuanto más bajo sea el árbol, menos accesos a memoria secundaria necesitaremos para acceder a un nodo, es decir, el tiempo de búsqueda será menor.

Dado que interesa reducir al máximo la altura de un árbol B, ¿por qué no aumentamos "indefinidamente" el número de hijos del árbol?

Porque al final estariamos accediendo a todos nodos para buscarlo, es decir, el tiempo de búsqueda sería de `O(n)`. Por tanto, el número de hijos debe ser equilibrado, es decir, no tener muchos hijos pero tampoco pocos.

¿Existe alguna operación claramente ineficiente en los árboles generales representados mediante listas de hijos?

Una operación sería la búsqueda de un nodo, debido a que si un nodo tiene n hermanos, para acceder a él tendremos que recorrer la lista de hermanos.

Las operaciones del TAD Árbol General no permiten insertar y eliminar nodo internos, solo nodos hoja.

FALSO: Podemos insertar un nodo como hijo Izquierdo o como hermano derecho independientemente de que tenga o no hijos o hermanos.

Para reconstruir un Árbol General solo necesitamos su recorrido en inorden y en postorden o preorden, pero no hace falta el grado del árbol ya que cada nodo puede tener un grado diferente.

No, necesitamos los 3 recorridos para conocer su estructura, porque llegará un momento en el que no sabemos si ese nodo es hijo izquierdo o hermano derecho.

La representación del TAD Árbol General mediante listas de hijos es más eficiente en espacio cuanto más bajo es el árbol para un número determinado de nodos.

Cuanto más bajo, cuanto menos profundo es, por tanto menos listas necesitas, ergo: es más eficiente en espacio.

Se define el desequilibrio de un Árbol General como la máxima diferencia entre las alturas de los subárboles más bajo y más alto de cada nivel. Esta definición y la diferencia de longitudes entre la rama más larga y más corta de dicho árbol son equivalentes.

FALSO: La diferencia de longitudes entre la rama más larga y más corta de un árbol coincide con el desequilibrio del subarbol que forma todo el arbol general, sin embargo, el desequilibrio de un árbol general es el maximo de los desequilibrios de todos sus subarboles y puede ser que en un subarbol de este haya más diferencia que en el caso general.

La función para calcular la profundidad de un Árbol General es similar a la del Árbol Binario y del mismo coste O(log n).

FALSO: El coste de calcular la profundidad en un Agen y en un Abin es de coste `O(n)`, porque se visitan los nodos al menos una vez.

1.4. Preguntas de ABB

¿Qué consiguen los árboles binarios de búsqueda (ABB)?

Debido a su estructura, y ordenación de los elementos, nos permite realizar búsquedas en `O(log n)` en el caso promedio (que esté equilibrado).

¿Es siempre la eliminación de elementos en un ABB de orden mayor que O(n)?

No, depende del grado de desequilibrio del ABB, si está equilibrado, la eliminación de elementos es de `O(log n)`, en el caso peor si es de `O(n)`. Como mucho va a ser de `O(n)`, pero nunca va a ser mayor.

¿Qué condiciones debe cumplir un ABB para que la búsqueda sea menor que O(n)?

Que esté equilibrado.

Al insertar en un Árbol B, si el nuevo elemento no cabe en el nodo que le correspondería, se divide el nodo en dos y se promociona un elemento al nodo padre, y en este caso, se permite que exista algún nodo con un solo elemento, teniendo en cuenta el mínimo permitido según el orden del árbol.

VERDADERO: En un árbol B un nodo tendrá K ((m-1)/2)minimo y (m-1)máximo claves. Por tanto, puede haber un nodo que contenga solamente un elemento después que haya sido promocionado.

Al insertar en un árbol B, si el nuevo elemento no cabe en el nodo que le correspondería, se divide el nodo en dos y se promociona la mediana al nodo padre. No se permite que existan nodos con menos elementos de la mitad (por defecto) de la capacidad de un nodo.

FALSO: Se promociona o el mayor de los menores o el menor de los mayores, pero no la mediana.

Supongamos un ABB con el elemento x en una hoja cuyo padre tiene el valor y. Entonces, y es el menor elemento mayor que x o bien, y es el mayor elemento menor que x.

VERDADERO: Si x es el hijo izquierdo de y, entonces y es el menor elemento mayor que x. Si x es el hijo derecho de y, entonces y es el mayor elemento menor que x.

Para conseguir que la anchura del árbol B sea menor, me interesa crear nodos con el mayor tamaño posible.

FALSO: Como el número de elementos o claves de un nodo está ligado al número de nodos del árbol, a mayor número de nodos, mayor número de claves y por ende, mayor anchura y altura.

La propiedad de búsqueda de un ABB permite encontrar un elemento en un tiempo de O(n) en el caso peor.

VERDADERO

La eliminación de elemento en un ABB puede llegar a tener un coste O(n).

VERDADERO: En el caso peor, si el árbol está desequilibrado, la eliminación de elementos puede ser de `O(n)`.

El recorrido en preorden de un ABB determina unívocamente el ABB del que procede.

Si está equilibrado, si, ya que el recorrido en preorden nos dice la raiz, luego el hijo izquierdo y por último el hijo derecho. Pero si no lo está, necesitamos el inorden para saber si un nodo es hijo izquierdo o derecho.

Los nuevos elementos de un árbol B se insertan en las hojas y, si es necesario, se reorganiza el árbol.

VERDADERO: En un árbol B, los nuevos elementos se insertan en las hojas, pero si no cabe en el nodo hoja, se promociona un elemento al padre y se divide el nodo.

En un árbol B de orden m, todos los nodos contienen un mínimo de [m-1/2] claves, y un máximo de m-1.

VERDADERO

1.5. Preguntas de APO

¿En qué condiciones puede un árbol ser un APO y ABB simultáneamente?

Cuando o bien estén vacios o solamente contengan un nodo, el raíz.

¿Influye el orden de inserción de los elementos para el desequilibrio de un APO?

No. En un APO, los elementos se insertan al final y luego se flotan hasta satisfacer la condición de ordenación y equilibrio, por tanto, no influye el orden de inserción.

¿Influye el número de elementos de un APO a su desequilibrio?

Si. En un APO, no es lo mismo insertar 3 nodos que al reordenarlos se cumple la condición de equilibrio, que insertar 4 nodos y que no se cumpla la condición de equilibrio (queda un nodo en nivel abajo)

¿Puede reconstruirse un APO de forma unívoca dado su recorrido en preorden?

VERDADERO: Mediante el preorden podemos obtener el número de nodos y con ello podemos calcular la altura del APO h $= \log_2 n$. Con la altura y el número de nodos podemos reconstruir el APO ya que sabemos cuando parar en el recorrido en preorden, es decir, decidir cual es el ultimo nivel.

¿Es posible obtener coste O(n) en la eliminación de un nodo cualquiera de un APO?

FALSO: La eliminación de un nodo cualquiera de un APO es de $\log_2 n$ en el caso promedio, ya que al eliminar un nodo, se intercambia con su descendiente menor a él.

¿Es posible obtener coste O(n) en la inserción/eliminación de la raíz de un APO?

FALSO: La eliminación de un nodo cualquiera de un APO es de $\log_2 n$ en el caso promedio, ya que al eliminar un nodo, se intercambia con su descendiente menor a él.

Del mismo modo, la inserción se hace en $\log_2 n$ en el caso promedio, ya que se inserta al final y luego se flota hasta satisfacer la condición de ordenación y equilibrio.

Los elementos de un APO se obtienen en orden mediante la extracción sucesiva de estos.

VERDADERO: En el TAD APO solo podemos acceder al elemento que se encuentra en la cima, el cual es el menor del APO. La extracción la realizamos leyendo la cima y suprimiendola. Al eliminar la cima, el APO se reorganiza para que el siguiente elemento más pequeño sea la cima. Por tanto, podemos obtener los elementos en orden de menor a mayor.

Todo APO min-max cumple estrictamente con las condiciones que hemos definido para un APO, pero no ocurre al contrario.

- Primera condición de un APO: Es completo salvo por el último nivel, que está lleno de izquierda a derecha. **VERDADERO**, tanto APO como APO min-max cumplen esta condición.
- Segunda condición de un APO: Cada nodo es menor que sus hijos. **FALSO**, en un APO min-max, se intercalan niveles min, cuya condición es esa, con niveles max, cuya condición es que cada nodo es mayor que sus hijos.

Por tanto, ambas afirmaciones son falsas.

La propiedad de orden parcial de un APO no implica que siempre va a estar equilibrado.

VERDADERO, el estar equilibrado depende del número de elementos insertados, no de la propiedad de orden parcial.

Si un Árbol es un APO, tiene un desequilibrio en valor absoluto menor o igual que 1, pero no todo desequilibrio menor o igual que 1 indica que sea un APO.

La primera afirmación es verdadera, ya que un APO siempre está completo salvo por el último nivel, que está lleno de izquierda a derecha, por tanto, el desequilibrio en valor absoluto siempre será menor o igual que 1.

La segunda afirmación tambien es verdadera, ya que para que sea un APO tambien tiene que cumplir la propiedad de orden parcial, es decir, que cada nodo es menor que sus hijos, por tanto no todos los arboles con desequilibrio menor o igual a 1 son APOs.

El recorrido en anchura de un APO no proporciona el acceso en orden a los elementos almacenados. O se pueden obtener los elementos en orden de un APO.

VERDADERO, ya que recorrer el arbol en anchura no implica que los elementos estén ordenados. Pueden existir niveles que sean menores que sus descendientes pero que al recorrerlos en anchura no estén ordenados. (diapositiva 5 APO)

1.6. Preguntas de AVL

¿Qué aportan los AVL frente a los ABB?

Los AVL nos asegura que la altura del árbol no sea mayor que $O(\log n)$, lo que nos asegura que las operaciones de inserción, eliminación y búsqueda sean de $O(\log n)$ en el peor caso.

Explica por qué se exige un equilibrio perfecto en los AVL.

Traducción de equilibrio perfecto: 1, 0 o -1. Se exige para garantizar que la búsqueda sean de $O(\log n)$ en el peor caso.

¿Influye el orden de inserción de los datos en la altura de un AVL?

No, ya que se auto equilibria para poder cumplir su propiedad de equilibrio perfecto.

El factor de equilibrio en los AVL.

- Si el desequilibrio es 0: El árbol está equilibrado.
- Si el desequilibrio es 1: El arbol tiene un nodo más en el subárbol derecho que en el izquierdo.
- Si el desequilibrio es -1: El arbol tiene un nodo más en el subárbol izquierdo que en el derecho.

El factor de equilibrio se calcula como la diferencia entre la altura del subárbol izquierdo y la altura del subárbol derecho.

El cálculo de la altura de un árbol es de O(n), excepto para los AVL, en cuyo caso el orden del cálculo de la altura es logarítmico.

VERDADERO. Lo pone en las diapositivas.

La propiedad de equilibrio de un AVL permite encontrar un elemento en un tiempo de O(log n) en el caso peor.

VERDADERO. Lo pone en las diapositivas.

La inserción en el mismo orden de un conjunto de elementos en un ABB y un AVL no tendría porque dar como resultado árboles de la misma altura.

VERDADERO. Un AVL tiene autoequilibrado, por tanto la altura del árbol no tiene porque ser la misma que la de un ABB aunque se inserten en el mismo orden.

Un AVL es un ABB y el recíproco no es cierto.

VERDADERO. Un AVL es un ABB, ya que cumple con la propiedad de orden parcial (todo nodo menor que la raiz es hijo izquierdo y todo nodo mayor que la raiz es hijo derecho), pero no todo ABB es un AVL, ya que no todos los ABB cumplen con la propiedad de equilibrio perfecto.

Es cierto que todos los AVL son ABB y es cierto que algunos ABB no sean AVL

La primera afirmacion es verdadera (ejercicio anterior), la segunda afirmación tambien es verdadera, ya que no todos los ABB cumplen con la propiedad de equilibrio perfecto.

La propiedad de equilibrio de un AVL no implica que su altura sea la mínima posible.

FALSO. La propiedad de equilibrio de un AVL implica que la altura del árbol no sea mayor que $O(\log n)$, es decir, sea la minima posible para el numero de nodos que tiene.

En un AVL cada nodo tiene un factor de equilibrio de 1,0 o -1 y ello significa que todas las hojas se van a encontrar en el último nivel o en el penúltimo.

FALSO: No tiene nada que ver, puede haber hojas en cualquier nivel siempre que se cumpla el factor de equilibrio. (diapositiva 9 AVL)

La propiedad de búsqueda de un ABB permite encontrar un elemento en un tiempo de O(n) en el caso peor.

VERDADERO. La búsqueda en un ABB es de `O(n)` en el caso peor, ya que no está equilibrado.

La condición de equilibrio no perfecto de un AVL asegura que la inserción de un elemento se pueda hacer a un coste de O(log n) en el peor caso.

FALSO. La condición de equilibrio **perfecto** de un AVL asegura que la inserción de un elemento se pueda hacer a un coste de `O(log n)` en el peor caso.

2. Preguntas de Grafos

2.1. Preguntas de Particion

Dado el TAD Partición, decir qué combinación entre estructuras de datos y estrategia es necesaria para que tanto la búsqueda como la unión sean de O(1).

No existe ninguna combinación de estructuras de datos y estrategias que permitan que tanto la búsqueda como la unión sean de O(1).

- Si escogemos un vector de pertenencia:
  - Búsqueda: O(1)
  - Unión: O(n)
- Si escogemos un bosque de arboles y estrategia de unión por altura:
  - Búsqueda: $O(\log n)$
  - Unión: O(1)

¿Qué beneficio aporta la compresión de caminos en la implementación del TAD Partición?

Poder acercarnos a órdenes constantes en la buscar.

En el TAD Partición, ¿es posible emplear la unión en altura y la compresión de caminos a la vez?

Sí, es posible. En la unión por altura el árbol con menos altura siempre será hijo del árbol con más altura

¿Qué se consigue con la técnica de unión por tamaño? ¿y con la técnica de unión por altura?

Ambas técnicas aseguran un coste, en el peor caso, de log n para la operación encontrar.

- Por tamaño: El árbol con menos nodos se convierte en subárbol del que tiene mayor número de nodos.

- Por altura: El árbol menos alto se convierte en subárbol del otro.

En ambas técnicas se controla la altura del árbol resultante, consiguiendo que la operación unir() sea del orden de O(1) y que la operación encontrar sea del orden de O(log n).

¿Existe una estructura de datos que permita ejecutar simultáneamente las operaciones de unión y encontrar en tiempo constante?

No, pero utilizando bosques de arboles nos aseguramos que la union es de O(1). Si aplicamos la tecnica de comprersión de caminos, la busqueda se aproxima a O(1).

2.2. Preguntas de Grafos

¿Por qué surgen los grafos?

Surgen de la necesidad de tener una estructura de datos cuyos nodos no estén organizados de manera secuencial (como en las listas) ni tampoco estén relacionados de manera jerárquica (como en los árboles). En este sentido, lo único que diremos es que es una estructura que conecta los nodos de una red mediante aristas.

Explica las razones por las que no es necesario marcar los nodos visitados al realizar el recorrido de un grafo.

Esta afirmacion es incorrecta, ya que al no seguir ningún orden de jerarquia, podemos llegar a recorrer un mismo nodo varias veces (ciclo). Por lo tanto, es necesario marcar los nodos visitados para evitar recorrerlos más de una vez.

Comente la siguiente afirmación: "Prim y Kruskal resuelven el mismo problema y dan la misma solución".

Ambos resuelven el mismo problema (generan un arbol de expansión de coste minimo). Sin embargo, no siempre tienen que dar la misma solución. Prim parte de un nodo y va añadiendo aristas de menor coste, mientras que Kruskal parte de un grafo vacio y va añadiendo aristas de menor coste que no formen ciclos. Como ambos tienen un enfoque diferente y pueden existir diferentes arboles de expansión de coste minimo debido a que se pueden repetir aristar son mismo coste, el resultado no tiene por qué ser el mismo.

¿Por qué el algoritmo de Kruskal asegura que no se producen ciclos?

Porque al utilizar una partición, se asegura que no se añadan vertices que ya hemos unido previamente, evitando así la formación de ciclos.

¿Es necesario ordenar las aristas en el algoritmo de Kruskal?

Si, es necesario ordenar las aristas para poder seleccionar la de menor coste en cada iteración.

¿Cuál es la condición que debe cumplir un grafo no dirigido para que Kruskal obtenga un resultado?

Que el grafo sea conexo y ponderado.

¿Por qué Kruskal no devuelve un grafo?

OJO: En nuestra representación Kruskal si devuelve un grafo, pero por dentro es un arbol de expansion de coste minimo. Teoricamente, Kruskal devuelve un Grafo ponderado conexo y aciclico, el cual es un arbol de expansion de coste minimo.

¿Qué pasaría si Prim y Kruskal operaran sobre un grafo dirigido?

Que no funcionarian, los algoritmos estan pensados para grafos no dirigidos.

Dado el algoritmo de Kruskal implementado mediante el TAD Partición, ¿son los mismos árboles los de la partición y los del algoritmo?

No son los mismos (o es muy improbable) ya que cuando acaba Kruskall, la estructura del árbol se almacena en el objeto partición, depende del orden de inserción de las aristas(por peso) y la raíz de ese árbol no tiene porque coincidir con la del árbol devuelto por el algoritmo.

¿En la representación mediante bosques de árboles con unión por altura, por qué las raíces de los árboles se representan con números negativos?

Porque en los datos almacenados en el vector almacenamos la altura del árbol en el que se encuentra el nodo raíz. Como esta altura debe ser 0, se representa con un numero negativo.

Diferencias y similitudes entre Prim y Kruskal.

Ambos resuelven el mismo problema (generar un arbol de expansion de coste minimo), pero con enfoques diferentes. Uno parte de un nodo y va añadiendo aristas de menor coste (Prim), mientras que el otro parte de un grafo vacio y va añadiendo aristas de menor coste que no formen ciclos (Kruskal). Por lo tanto, el resultado no tiene por qué ser el mismo. Además, Prim es Orden O(n$\hat{2}$) y Kruskal es O(a log n).

¿Qué ventajas e inconvenientes plantea el uso de matrices de adyacencia y costes para la representación de grafos?

- Ventajas: Son muy eficientes para comprobar si existe una arista entre un vértice y otro.
- Inconvenientes: Desaprovechan gran cantidad de memoria si el grafo no es completo. Además, si la matriz no es dinámica, no se pueden añadir o eliminar vértices.

¿Qué ventajas e inconvenientes plantea el uso de listas de adyacencia para la representación de grafos?

- Ventajas: Aprovecha el espacio de memoria, pues solo se representan los arcos existentes en el grafo. Además, permite añadir y suprimir vértices.
- Inconvenientes: Son poco eficientes para determinar si existe una arista entre dos vértices del grafo, pues esta operación implica recorrer la lista.

¿Existe alguna estructura para la representación de grafos que permita añadir y suprimir vértices?

Si, las listas de adyacencia. Tambien matrices y vectores SOLO si son dinámicas.

¿Por qué no se permiten los costes negativos en Floyd?

Floyd calcula el camino de coste minimo de ir de cualquier nodo a otro en el grafo. Si se permiten costes negativos, se pueden dar ciclos negativos, lo que haría que el algoritmo no terminara nunca.

El algoritmo de Dijkstra, ¿funciona correctamente con valores negativos?

No, Dijkstra no funciona con valores negativos, ya que el algoritmo se basa en ir añadiendo nodos con el menor coste posible, y si hay un ciclo negativo, el algoritmo no terminaría nunca.

¿Por qué se coloca infinito en la diagonal principal de la matriz de costes en el algoritmo de Floyd?

Porque la diagonal principal de la matriz de costes representa el coste de ir de un nodo a si mismo, y este coste siempre es 0. Por lo tanto, se coloca infinito para que no se tenga en cuenta en el algoritmo.