Árboles

Estructura de datos organizada de manera **jerárquica**. Consiguen búsquedas en O(log n).

Para que las búsquedas se realicen en un orden menor que n, el árbol debe estar **equilibrado** y sus elementos deben mantener una **relación de orden** entre sí.

Tipos de recorridos:

- En **anchura**: por niveles (iterativa y recursiva).
- En **profundidad**: **preorden**, **inorden** y **postorden**. Con el recorrido en inorden y el preorden o postorden, podemos conocer el árbol.

En un árbol **no** se pueden producir **ciclos**, por lo tanto, **no es necesario marcar los nodos** como visitados cuando se recorren.

Árbol **completo**: rellenos hasta su último nivel. La mejor forma de representarlos es mediante un **vector de posiciones relativas**.

Árboles binarios

En la representación **vectorial** con índices al **padre**, **hijo izquierdo** e **hijo derecho**, las operaciones de **inserción** y **eliminación** se realizan en 0(1). Como todas las celdas libres están al final, se **inserta** en la primera libre (n.º máximo de nodos conocido) y cuando se **elimina**, se mueve el último nodo a la posición del hueco libre.

La representación mediante **celdas enlazadas** se utiliza cuando no conocemos a priori el **n.º máximo de nodos** o **no se accede** a la mayoría de ellos. En caso contrario, se utilizaría la representación mediante una matriz.

Árboles generales

Esta estructura **no se puede** representar mediante un **vector de posiciones relativas**, ya que no se puede relacionar los nodos a partir del grado (cada nodo tiene un grado distinto).

En la representación mediante listas de hijos se podrían utilizar **listas doblemente enlazadas** pero no es recomendable, ya que no se accede nunca al hermano izquierdo de un nodo.

Operaciones claramente **ineficientes** según representación:

- Mediante **listas de hijos**: todas las operaciones que tengan que ver con **hermano derecho**, ya que la búsqueda secuencial resulta O(n).

- Mediante **celdas enlazadas** con **punteros** al **hijo izquierdo** y el **hermano derecho**: la operación padre(n) es O(n).
- Mediante **celdas enlazadas** con **punteros** al **padre**, **hijo izquierdo** y **hermano derecho**: ninguna, todas son O(1).

ABB

Para que las búsquedas se realicen en un orden menor que n, el árbol debe estar **equilibrado** y sus elementos deben mantener una **relación de orden** entre sí.

La **eliminación** solo es 0(n) en el peor caso, cuando el **árbol** esté **degenerado** en una **lista**.

Árboles B

Son la generalización de los ABB. Por ejemplo, un **árbol terciario de búsqueda** sería un árbol B de orden m=3 y k=2, siendo m el n.º máximo de hijos de cada nodo y k el n.º máximo de elementos o claves de cada nodo.

Interesa que los árboles B tengan **poca altura**, ya que se necesitarán menos pasos para **encontrar la clave**, por tanto, menos **accesos a memoria secundaria**.

Tampoco debemos **aumentar** indefinidamente el **n.º de hijos**, ya que se puede exceder el **tamaño** de un **bloque** y serán necesarios más de un **acceso a memoria secundaria**.

AVL

Son **ABB equilibrados**, es decir, que la altura de los dos subárboles nunca difiere en más de una unidad. Se evita así que se degenere en un lista, por tanto, asegura que las **búsquedas**, **inserciones** y **eliminaciones** se efectúen en $0(\log_2 n)$.

El **factor de equilibrio** es la altura del **subárbol derecho menos** la del **subárbol izquierdo** y siempre será -1, 0 o 1.

La **altura** de un AVL depende del **n.º de nodos** no del orden en el que se inserten. Además, **no se exige** un **orden** exacto en la **inserción** de elementos, ya que el AVL resultante seguirá cumpliendo las propiedades. La operación insertar reorganiza el árbol, es decir, se **autoequilibra**.

APO

Son **árboles completos**, por lo tanto, siempre están **equilibrados**.

Se **inserta** y se **elimina** siempre la **raíz** del mismo por lo que el acceso se realiza en **orden constante** y flotar el elemento o hundirlo se hacen en $0(\log_2 n)$.

El **orden de inserción** de los elementos influye en la **posición** de éstos, ya que en función de cuándo se inserten se hundirán unos ancestros u otros. Por otro lado, no influye en el **desequilibrio**, ya que son árboles completos que se rellenan de izquierda a derecha, por lo tanto, tendrá un máximo desequilibrio de 1.

El **n.º de elementos** sí que influye en el desequilibrio, ya que si: $numNodos = \sum_{i=0}^{h} 2^i$

el árbol tendrá el último nivel completo, por lo que el desequilibrio es 0, y si es menor, el desequilibrio es |1|.

Grafos

Representan **relaciones binarias** entre elementos de un conjunto. Estructura de datos que **conecta** los **nodos** de una red **mediante aristas**.

Es **necesario marcar los nodos visitados**, ya que, al no existir secuencialidad o jerarquía, nada nos impide entrar en un ciclo.

Algoritmo de Dijkstra

Calcula los **caminos** de **coste mínimo** entre **origen** y **todos los vértices** del grafo G.

No funciona correctamente con **valores negativos**, ya que al se un **algoritmo voraz**, se va quedando con la mejor solución hasta el momento, y en el caso de que los permitiéramos, pudiera ser que, habiendo llegado a una solución concreta, nos encontrásemos con un camino más corto, causado por éstos costes.

Algoritmo de Floyd

Calcula los **caminos** de **coste mínimo** entre cada **par de vértices** del grafo G.

La **diagonal principal** de la matriz de costes se coloca **a cero**, ya que todo vértice v tiene un camino hacia sí mismo de coste igual a 0.

En este algoritmo tampoco se permiten **costes negativos**, ya que también es un **algoritmo voraz**.

Algoritmo de Kruskal

Conecta todos los nodos de una red con un coste mínimo y da como solución un **árbol generador de coste mínimo**. Para que el algoritmo obtenga resultado, el grafo debe ser **no dirigido**, **conexo** y **ponderado**.

Es necesario **ordenar** las **aristas** ya que el algoritmo considera éstas en **orden creciente** de **costes**. El algoritmo comienza tomando la **arista** de **menor coste** y va formando el árbol generador de coste mínimo.

Asegura que **no** se producen **ciclos** porque va marcando las **aristas** de **menor coste** una a una, comprobando que no se producen ciclos con las ya marcadas, hasta tener **n-1** aristas (siendo n el n.º de nodos).

TAD Partición

Ayuda en la implementación del algoritmo de Kruskal. Una **partición no es** nunca un **árbol** aunque se represente mediante **bosque de árboles**. Aunque es cierto que los **vértices** de la **partición** serán los mismos que los que formen el árbol.

Procedimientos para la implementación del TAD Partición: Representación mediante **bosque de árboles** (con **control de altura**).

- **Unión por tamaño**: El **árbol** con **menos nodos** se convierte en **subárbol** del que tiene **mayor número** de **nodos**.
- **Unión por altura**: El **árbol menos alto** se convierte en **subárbol** del **otro**.

Otro procedimiento es la **compresión de caminos**, que en cada **búsqueda**, al **subir de nivel** hace que el nodo por el que pasa sea **hijo** de la **raíz** y así nos aproximamos a la solución ideal de dos niveles.

Se emplea la **unión por altura** y la **compresión de caminos** a la vez. Esta suma de procedimientos aseguran en el **peor caso** el **O(log n)**, y después de muchas ejecuciones podrían llegar a acercarse a O(1).

A tener en cuenta también que, en la representación mediante **bosque de árboles** con unión por altura, las raíces de los árboles se representan con números negativos para diferenciar los representantes canónicos. Además, en valor absoluto representa la altura del árbol más 1.

Algoritmo de Prim vs. Kruskal

Ambos algoritmos resuelven el **mismo problema**, conectan todos los nodos de una red con un coste mínimo y dan como resultado un árbol generador de coste mínimo. Pero para resolverlo, **Kruskal** ordena las **aristas** de **menor a mayor coste** y se ayuda del **TAD Partición**, y **Prim** va buscando la **aristas** de **coste mínimo** a lo largo del algoritmo. **Kruskal** resulta **más eficiente** en una constante multiplicativa.

Los algoritmos de **Prim** y **Kruskal** dan por hecho que el **grafo** es **no dirigido**, en caso contrario, operarían normalmente y obtendrían la solución sin indicar el sentido en el que habría que recorrerlo.

Ambos algoritmos resuelven el **mismo problema** pero **no tiene por qué** dar la **misma solución**, ya que puede haber **aristas** con el **mismo peso** y éstos resuelven el problema de distinta forma.