1.1 Comparación entre un ABB y un AVL

Un árbol binario de búsqueda (ABB) mantiene los elementos ordenados, pero no garantiza equilibrio, por lo que la complejidad de las operaciones depende de su altura. En el mejor caso, cuando el árbol está equilibrado, la búsqueda tiene complejidad O(log N), pero en el peor caso puede degenerar en una estructura lineal con complejidad O(N).

Por el contrario, un árbol AVL es un árbol binario de búsqueda autoequilibrado que garantiza una altura O(log N), manteniendo así la complejidad O(log N) en las operaciones de búsqueda, inserción y eliminación. Para ello, cada nodo almacena un factor de equilibrio y, tras inserciones o borrados, pueden ser necesarias rotaciones para restaurar el equilibrio.

Como ventaja, el AVL ofrece tiempos de acceso más predecibles y eficientes; como inconveniente, introduce mayor complejidad y coste en las operaciones de modificación debido al mantenimiento del equilibrio.

1.2 Comparación entre un todos los tipos de árboles (B+, B, B*, AVL y ABB)

Los árboles binarios de búsqueda y los árboles AVL son estructuras orientadas a memoria principal. Aunque el AVL garantiza una altura O(log N) mediante rotaciones, su grado es bajo (binario), lo que implica árboles relativamente altos cuando el número de elementos es grande.

Los árboles B y sus variantes (B* y B+) están diseñados para trabajar eficientemente sobre memoria secundaria. Agrupan múltiples claves en nodos llamados páginas o bloques, reduciendo significativamente la altura del árbol y, por tanto, el número de accesos a disco necesarios para las operaciones.

Un árbol B de orden m almacena hasta m−1 claves por nodo y mantiene todas las hojas al mismo nivel, garantizando una altura O(log N). Las claves internas actúan como separadores que mantienen el orden, preservando la propiedad de búsqueda.

El árbol B* mejora al árbol B aumentando el grado mínimo de ocupación de los nodos, logrando un mejor aprovechamiento del espacio y reduciendo la frecuencia de divisiones.

El árbol B+ separa índices y datos: las claves internas solo actúan como índices y todos los datos se almacenan en las hojas, las cuales suelen estar enlazadas. Esto mejora el rendimiento en búsquedas por rango y simplifica el acceso secuencial, a costa de duplicar claves y usar más espacio.

En comparación con los AVL, los árboles B ofrecen una altura mucho menor y un rendimiento superior en sistemas con grandes volúmenes de datos almacenados en disco, aunque con estructuras más complejas y mayor coste en operaciones de inserción y eliminación.

2. Montículos binarios

Un montículo binario es un árbol binario completo o semicompleto con una propiedad de orden parcial: en un montículo de máximos cada nodo es mayor o igual que sus hijos, y en uno de mínimos menor o igual. No existe un orden total entre los elementos.

Esta estructura se utiliza principalmente para implementar colas con prioridad, ya que permite acceder al elemento de mayor o menor prioridad en tiempo constante y realizar inserciones y eliminaciones en tiempo O(log N).

Los montículos se representan habitualmente mediante arrays de forma implícita, lo que elimina el uso de punteros. Esta representación es especialmente eficiente cuanto más lleno esté el árbol, ya que los nodos inexistentes ocupan espacio en el array. Por ello, es fundamental que el árbol se mantenga completo y se rellene por niveles de izquierda a derecha.

Como inconveniente, esta representación no es tan flexible para modificaciones arbitrarias como un árbol basado en punteros.

Un montículo binario puede utilizarse para implementar la cola de procesos listos en un sistema operativo, donde cada proceso tiene una prioridad. El montículo permite seleccionar eficientemente el proceso con mayor prioridad para su ejecución, manteniendo un coste logarítmico en inserciones y extracciones

3. Grafos (Estructura y propiedades)

Ejemplo Opción Dinámica Representación

En el ejemplo de exámen pone una foto de un grafo que representa las islas de la provincia de Tenerife para lo que se recomienda una estructura estática como una Matriz de Adyacencia y como en las prácticas también utilizamos esta representación indagué con el Chat GPT un ejemplo realista donde una

lista de adyacencia es más eficiente, por si lo pregunta en el exámen saber visualizarlo.

Una red social como Twitter/X donde hay millones de usuario pero cada uno solo sigue a unos pocos, se pueden aceptar solicitudes o incluso eliminar personas de la lista de seguidos. Una matriz de adyacencia sería insostenible porque es un grafo tremendamente disperso además en este caso $O(N+a) <<< O(N^2)$. Además, a cada usuario solo le interesa consultar sus seguidos, por lo que una lista de adyacencia es mucho más interesante que no una matriz de Adyacencia.

3.1 Estructura, tipo y justificación

En el ejemplo, una matriz de Adyacencia sería una mejor alternativa que una lista enlazada ya que el grafo no es especialmente disperso al tener una cantidad significativa de aristas comparadas con la que podría tener un grafo de 4 vértices por lo que el desperdicio de utilizar una matriz es poco relevante. Además, comprobar si hay una conexión (arista) entre dos islas (vértices) del mapa (grafo) es mucho más sencillo y no implica recorrer listas.

Pero el factor más fundamental es que la cantidad de islas de la provincia de Tenerife no va a cambiar por lo que no tiene sentido utilizar una implementación dinámica ya que no estaríamos utilizando sus beneficios, además que para un problema con tan pocos datos la Matriz es mucho más directa y las diferencias entre ambas implementaciones son poco sustanciales.

3.2 Calcular distancias mínimas entre cada par de vértices

Se utilizará el algoritmo de Floyd–Warshall, ya que permite calcular los caminos con peso mínimo entre todos los pares de islas en nuestro ejemplo y en cualquier grafo ponderado. Además, mediante la matriz de predecesores asociada es posible reconstruir el camino mínimo entre cualquier par de islas, incluyendo aquellos aue no corresponden a conexiones directas de la matriz de adyacencia definida en el apartado anterior.

3.3 Identificar puntos “vulnerables” de la red

Se buscarían los puntos de articulación, es decir, vértices que, al ser eliminados, provocan que el grafo no dirigido resultante quede dividido en más de una componente conexa. En el contexto del problema, esto implica que si una isla deja de funcionar, alguna de las restantes quedaría incomunicada del resto. Estos vértices pueden identificarse a partir de un recorrido recursivo en profundidad y del árbol de recubrimiento generado durante dicho recorrido

3.4 Árbol de expansión mínimo

Aplicaríamos o bien el algoritmo de Prim o Kruskal para generar el árbol de expansión mínimo que conecte todas las islas a través de las conexión con menor peso. Cualquier conexión que no forme parte del árbol, podrá ser eliminable sin afectar a la estructura básica del grafo. En un contexto tan pequeño como el que nos plantea el ejercicio no supondrá un gran cambio utilizar uno u otro.

4. Estructuras de datos

4.1 Estructura para guardar muchos datos e insertar constantemente de manera fácil

Para almacenar temporalmente la información que los robots web van enviando de forma continua, la estructura más adecuada es una **cola**. Esta permite gestionar eficientemente grandes volúmenes de datos siguiendo un criterio FIFO, garantizando que las páginas se procesen en el mismo orden en que se reciben. Además, se adapta bien a un flujo de entrada variable y continuo, ya que las operaciones de inserción y extracción son eficientes. De este modo, la cola actúa como un almacenamiento intermedio mientras la información se organiza e indexa posteriormente.

4.2 Estructura para ordenar alfabéticamente de manera eficiente

La estructura más adecuada para indexar la gran cantidad de páginas web de forma alfabética es un árbol B+, ya que está diseñado para gestionar grandes volúmenes de datos ordenados de manera eficiente. Aunque podría utilizarse un árbol AVL, cuando el volumen de datos es muy elevado resulta más eficiente trabajar con páginas grandes y un árbol más bajo, reduciendo así el número de accesos a memoria secundaria. Además, al estar los elementos dentro de cada página ordenados, es posible aplicar búsqueda binaria, manteniendo una complejidad similar a la del recorrido del árbol pero evitando accesos innecesarios a disco y reestructuraciones frecuentes.

4.3 Estructura para guardar el tránsito web de páginas que referencian a otras

La mejor opción y las más intuitiva en este apartado es un grafo dirigido, ya que representa perfectamente la idea de páginas que hacen referencia a otra, además se puede utilizar el peso de las aristas para modelar la cantidad de veces que una página web enlaza a otra. En este caso la mejor opción sería implementarlo mediante una lista enlazada ya que se trata de un grafo con muchísimos vértices pero enormemente disperso donde una matriz de adyacencia sería insostenible.

5. Tablas Hash y otras estructuras

La principal ventaja de una tabla hash es que, con una buena función hash, un factor de carga adecuado y una estrategia correcta de resolución de colisiones, las operaciones de búsqueda, inserción y eliminación tienen un coste O(1) en promedio. En el peor caso, cuando hay muchas colisiones, el coste puede degradarse a O(N).

La ordenación no es eficiente en una tabla hash, ya que sus elementos se insertan de manera aleatoria en las posiciones que indica la función hash, requeriría de una estructura adicional y una complejidad mayor pero en la práctica no se utilizan tablas hash para guardar datos ordenados.

El vector por su parte si puede ser ordenado gracias a algoritmos muy optimizados como QuickSort en un tiempo O(n*log(n)) a veces reserva más memoria de la necesaria al tener un tamaño fijo a diferencia de la lista enlazada, si bien esta última requiere de punteros adicionales para cada elemento que aumentan el coste en memoria.

La lista ordenada no permite el acceso aleatorio como Hash y Vector por lo que tiene que ser buscada de manera iterativa y la ordenación es más costosa que el Vector. En este último, en caso de estar ordenado se puede aplicar búsqueda binaria (O(log(n)) o en otro caso búsqueda secuencial (O(n)), sin embargo la lista para insertar en posiciones intermedias es más cómoda que el vector al evitar realizar desplazamientos.

Si simplemente se desea insertar valores al final llevando una cuenta el vector tiene acceso directo a través del índice mientras que en la lista dependemos de un puntero adicional que direccione el final de la lista o bien recorrerla iterativamente.

Finalmente, el uso de Encadenamiento incurre en un mayor gasto de memoria ya que cada dato pasa a ser un nodo de una lista enlazada. Sin embargo, la inserción suele ser más directa ya que no hay que buscar posiciones libres. La búsqueda suele ser parecida en ambos algoritmos pues en uno debemos recorrer la tabla hash mientras que en el otro las listas que conforman cada posición.

6. Algoritmo Voraz (Kruskal)

Kruskal es un algoritmo que genera un árbol de expansión mínimo a través de un grafo G al cuál se le han eliminado todas las aristas. Para ello utiliza un conjunto de aristas C, inicialmente todas ordenadas por peso, que representan las aristas todavía elegibles del grafo original. Dichas aristas son elegidas mediante la función seleccionar(C) y son agregadas al conjunto de aristas seleccionadas S (eliminadas por tanto de C), inicialmente vacío, a través de la función insertar(S, x), si cumplen

la condición de conectar dos componentes conexas distintas del grafo, es decir para esa arista x se valida factible(S, x), sin generar ciclos. Finalmente una vez finalizado el proceso se comprueba que el conjunto de aristas seleccionadas S conecte todos los vértices del grafo utilizando para ello la función solucion(S) que será equivalente a que el grafo G(V,S) esté formado por una única componente conexa.

7. Ramificación y Poda minimización

Para explorar el árbol de soluciones de la manera más veloz Ramificación y Poda lleva un umbral que deben cumplir las soluciones parciales para seguir siendo exploradas, para ello se fija una variable de Poda (C) que inicialmente es igual a la cota superior del nodo raíz y luego a medida que se van introduciendo nodos en la LNV y se descubren soluciones se va actualizando la cota de poda al menor valor garantizado por cualquier nodo intermedio o solución.

Cuando un nodo tiene un CI (cota inferior) superior a la variable de poda, entonces deja de explorarse y se poda, esto puede pasar antes de ser insertado en la lista de nodos vivos al ser extraído para comprobar si se debe ramificar. Los nodos se insertan en la LNV por orden creciente de costes estimados (LC, ya que es un problema de minimización) y puede seguirse una estrategia FIFO o LIFO en caso de empate, esta última suele ser más acertada ya que permite explorar primero el árbol de soluciones en profundidad ajustando mejor la variable de poda y realizando una poda más exhaustiva en futuras ramas.

El algoritmo termina cuando la LNV está vacía, bien porque hemos llegado a la solución de nuestra última rama, o hemos podado todos los nodos que teníamos en la LNV.