Resumen Final FOD

# Archivos:

Un **archivo** es una colección de registros que abarcan entidades con un aspecto en común, para un propósito en particular.

**Archivo físico:** es el archivo que reside en memoria secundaria y es administrado por el SO.

**Archivo lógico:** es el archivo utilizado en el algoritmo.

Formas de acceder a los datos de un archivo:

**Secuencial:** el acceso a cada elemento se realiza luego de haber accedido a su inmediato anterior. El recorrido se realiza del primero hasta el último de los elementos, siguiendo el orden físico de estos.

**Secuencial indizado:** el acceso a los elementos de un archivo se realiza teniendo presente algún tipo de organización previo, sin tener en cuenta el orden físico.

**Directo:** es posible recuperar un elemento de dato de un archivo con un solo acceso, conociendo sus características, más allá de que exista un orden físico o lógico predeterminado.

Operaciones básicas sobre archivos:

**assign:** vincula el archivo lógico utilizado por el algoritmo con el archivo físico, administrado por el SO.

**rewrite:** indica que el archivo va a ser creado, por lo que solo habilita la operación de escritura sobre él archivo, borrando también todo dato existente previamente.

**reset:** indica que el archivo ya existe, siendo válidas las operaciones de lectura/escritura de información sobre el archivo. Al ejecutarse, el SO coloca el puntero direccionando al primer registro disponible del archivo.

**close:** coloca la marca de eof al final del archivo. Al ejecutarse la instrucción, se transfiere definitivamente la información volcada sobre el archivo a disco, transfiriendo el buffer a disco.

**EOF:** retorna verdadero si el puntero del archivo referencia a la marca de eof, o falso en caso contrario.

**filesize:** indica la cantidad de elementos que conforman el archivo.

**filespos:** retorna un numero entero que indica la posición actual del puntero sobre el archivo (va desde 0 hasta la cantidad de elementos del archivo).

**seek:** se utiliza para modificar la posición del puntero dado un archivo y un número entero (0 a filesize del archivo).

## Buffers:

Las lecturas y escrituras desde/hacia un archivo se realizan sobre **buffers**, una memoria intermedia (ubicada en RAM) entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en memoria secundaria, o donde los datos residen una vez recuperados de dicha memoria secundaria.

La operación **read** lee desde un buffer y, en caso de no contar con información el SO realiza una operación **input**, trayendo más información al buffer.

La diferencia radica en que cada operación **input** transfiere desde el disco una serie de registros. De esta forma, cada determinada cantidad de instrucciones **read**, se realiza una operación input. Cada read se mide en nanosegundos mientras que los inputs en milisegundos.

De forma similar procede la operación write; se escribe en el buffer, y si no se cuenta con espacio suficiente, se descarga el buffer a disco por medio de una operación **output**, dejándolo el buffer nuevamente vacío.

#### Eliminación de un elemento:

**Baja física:** consiste en borrar efectivamente la información del archivo, recuperando el espacio físico. Se puede realizar de 2 formas:

* Generando un nuevo archivo que tenga solo los elementos válidos del original, excluyendo los borrados.
* Utilizando el mismo archivo de datos, realizando los reacomodamientos que se consideren necesarios (truncate).

**Baja lógica:** consiste en borrar la información del archivo, marcando el elemento como borrado, pero sigue ocupando el espacio dentro del archivo.

La ventaja del borrado lógico tiene que ver con la performance, basta con localizar el registro a eliminar y colocar sobre él una marca que indique que no está disponible. Con la información borrada que continua ocupando espacio se puede aprovechar de 2 formas:

* **Recuperación del espacio** que consistente en utilizar el proceso de baja física para compactar el archivo (usando cualquier de las técnicas de borrado físico).
* **Reasignación de espacio** la cual consiste en recuperar el espacio, utilizando los lugares indicados como borrados para realizar altas.

#### Eliminación de registros de longitud fija

Como todos los registros de un archivo de longitud fija son del mismo tamaño, no es difícil reutilizar los registros eliminados. La técnica consiste en agrupar todos los espacios disponibles en una lista, que se crea reuniendo todos los registros eliminados en forma de una lista enlazada.

Esta se forma acomodando campos de enlace en los espacios de registros eliminados de tal forma que cada registro de la lista contenga un campo que proporcione el número relativo del registro del que sea el siguiente borrado de la lista, hasta el último.

#### Eliminación de registros de longitud variable

Al igual que como se hace en los registros de longitud fija, un elemento puede ser eliminado de manera lógica y física. Sin embargo, cuando se desea recuperar el espacio borrado lógicamente con nuevos elementos, no solo debe disponerse de lugar, sino que el lugar debe tener espacio suficiente.

Para ello, el proceso de inserción debe localizar el lugar dentro del archivo más adecuado al nuevo elemento. Existen 3 formas genéricas para la selección de este espacio:

* **Primer Ajuste:** consiste en seleccionar el primer espacio disponible donde quepa el registro a insertar.
* **Mejor Ajuste:** Elige el espacio de menor tamaño donde quepa el registro.
* **Peor Ajuste:** selecciona el espacio de mayor tamaño, asignando para el registro solo los bytes necesarios, en vez del espacio entero.

**Fragmentación Interna**: ocurre cuando a un elemento se le asigna mayor espacio del necesario, desperdiciando el espacio restante. Los registros de longitud fija tienden a generar fragmentación interna. Además, las técnicas de primer y mejor ajuste suelen implementar una variante de recuperación de espacio que también generan fragmentación interna.

**Fragmentación Externa**: ocurre cuando el espacio que no se usa está entre los registros individuales. El espacio si está disponible, pero es demasiado pequeño para reutilizarse.

La técnica de peor ajuste solo asigna los bytes necesarios y el resto del espacio sigue figurando en la lista de disponibles. De esta forma, es posible que la técnica de peor ajuste genera fragmentación externa dentro del archivo.

# Índices:

Un **índice** es una estructura de datos adicional que permite agilizar el acceso a la información almacenada en un archivo. En dicha estructura se almacenan las claves de los registros del archivo, junto a la referencia de acceso a cada registro asociado a la clave.

Es necesario que las claves del índice permanezcan ordenadas. La característica fundamental de un índice es que posibilita imponer orden en un archivo sin que realmente este se reacomode.

Al ser de menor tamaño que el archivo de datos, es más probable que un archivo índice quepa en memoria principal. Además, se puede realizar búsqueda binaria por ser un archivo de longitud fija, permitiendo una notable mejora en performance de búsqueda.

Para acceder a un dato deseado a través de una clave secundaria, primero se accede al índice secundario correspondiente a la clave secundaria. Allí se obtiene la clave primaria, y se accede con dicha clave al índice primario para obtener finalmente la dirección efectiva del registro que contiene la información buscada.

Con la utilización de índices, el archivo de datos se trata como un archivo serie, dónde cada elemento es insertado al final del archivo y no hay orden físico de los datos. Cada archivo de índice tendrá entonces la siguiente estructura: el índice, los punteros a los hijos del nodo y la dirección (NRR) del registro completo en el archivo de datos si es un índice primario, o la clave primaria si es un índice secundario.

### Árbol binario

Es una estructura de datos no lineal, en la cual cada nodo puede tener a lo sumo dos hijos, teniendo los elementos menores a la izquierda, y los mayores a la derecha. Por esta razón, la búsqueda en un árbol binario es de O(Log(n)), ya que esta va desde la raíz y en cada revisión se descarta la mitad del archivo restante.

Una **ventaja** que tienen los árboles binarios es la **inserción de elementos**. Mientras que los archivos normales se desordenan frente a una inserción y reordenarlo resulta costoso, los árboles binarios pueden mantenerse ordenados de manera mucho menos costosa, encontrando al padre del elemento (lo que implicara Log(n) lecturas) y dos operaciones de escritura (el nuevo elemento y la actualización del padre).

De forma similar cuando se **borra un elemento** se necesitan Log(n) lecturas para encontrar al padre y dos escrituras para eliminar el nodo.

Pero los árboles binarios también presentan problemas, el desempeño en la búsqueda en un árbol binario es bueno cuándo está **balanceado**, el caso degenerado de un árbol binario transforma al mismo en una lista, quedando la performance de búsqueda de orden lineal.

Además, la elección de la raíz del árbol será determinante en si permanecerá balanceado o no. El problema es que al generar un archivo que implementa un índice de búsqueda, es imposible a priori determinar cuál es la mejor raíz, dado que dependerá de los elementos de datos que se inserten.

### Árbol AVL

Es un árbol binario balanceado en altura dónde el **máximo desbalanceo posible es uno.** El algoritmo que mantiene el balanceo del árbol frente a inserciones o borrados es bastante sencillo, pero los costos de acceso de dichas operaciones aumentan considerablemente, por lo cual su implementación deja de ser viable.

### Paginación de árboles binarios

Cuando se accede al disco para transferir datos no se transfieren unos pocos bytes, sino que se transfiere una página completa. Cada página contiene un conjunto de nodos, los cuales están ubicados en direcciones físicas cercanas. Un árbol dividido en páginas permite realizar búsquedas más rápidas en memoria secundaria. Así, suponiendo que en un buffer caben 255 elementos, el tamaño de cada página sería entonces de 255 nodos, resultando la performance final de búsqueda del orden log256(N), es decir, logK+1(N), siendo N la cantidad de claves del archivo y K la cantidad de nodos por página.

Pero dividir un árbol en páginas implica un costo extra necesario para su reacomodamiento y para mantener su balanceo interno. Un algoritmo que soporta esta construcción será muy costoso de implementar y luego también en cuanto a performance.

### Árboles multicamino

Un árbol multicamino es una estructura de datos en la **cual cada nodo puede contener M-1 elementos y M hijos**. Ahora, se podría pensar en un árbol multicamino dónde su orden sea la cantidad de nodos que entran en una página, y además permitiendo obtener árboles mucho más bajos.

## Árboles B

Los árboles B son árboles multicamino con una construcción particular,que **permite mantenerlos balanceados a bajo costo.** Un árbol B posee las siguientes propiedades:

* Cada nodo del árbol puede contener, cómo máximo, M descendientes y M - 1 elementos.
* La raíz no posee descendientes directos o tiene al menos dos. En ningún caso la raíz puede tener un solo descendiente.
* Un nodo con X descendientes directos contiene X-1 elementos.
* Los nodos terminales tienen, como mínimo, M/2 - 1 elementos, y cómo máximo M – 1.
* Los nodos que no son hoja ni raíz tienen, cómo mínimo M/2 elementos.
* Todos los nodos hojas se encuentran al mismo nivel. Esta restricción permite concluir que la performance de un árbol B será respetada sin importar cómo se construya el árbol, es decir, sin importar el orden de llegada de los elementos.
* Los árboles B crecen en altura, es decir, **es la raíz la que se aleja de los nodos terminales**, de modo tal que siempre el árbol crezca de manera balanceada.
* La **eficiencia de búsqueda** queda acotada en una cantidad de accesos en el rango de [1, H], donde H es la altura del árbol.
* La inserción siempre se realiza en nodos terminales. Si el nodo a realizar la operación dispone de lugar, la inserción se realiza sin dificultades, pero si no lo hay, se produce overflow, en cuyo caso se divide el nodo, llegando a poder propagarse el overflow hasta la raíz si se da el peor de los casos.
* Para borrar un elemento en un árbol B, debe estar en un nodo terminal. Cuándo este es eliminado, se pueden dar dos situaciones: que no haya ningún problema, o que genere **underflow**. Esto sucede cuándo el nodo deja de cumplir la condición de contener al menos [M/2] – 1 elementos, y es necesario llevar a cabo algún tipo de acción para corregir la situación, tratando de afectar la menor cantidad de nodos posible para mantener los cambios acotados.
* Una alternativa posible para resolver este problema, es la de **redistribuir** los elementos entre el padre y los nodos adyacentes, de esta manera el cambio sólo afecta a éstos y no se propaga a través del árbol. Pero para esto se debe disponer de un nodo adyacente hermano con suficientes elementos para compartir.

Cuando no se pueda redistribuir, la única opción que queda es **concatenar** los nodos y eliminar al padre, la desventaja es que la concatenación puede propagar cambios a lo largo del árbol.

## Árboles B\*

Es una variante de los árboles B en la que se modifica el proceso de inserción. En lugar de realizarse una división siempre que se produzca overflow, se intentará redistribuir, por lo que se demorará la generación de nuevos nodos y por ende **el árbol crecerá en altura de forma más lenta,** lo que trae aparejado una mejor performance.

Cuando hay overflow, redistribuye sus elementos utilizando un nodo adyacente hermano. Si esto no es posible, dado que tanto el nodo que genera el overflow como su hermano están completos, se puede dividir dos nodos completos en tres nodos dos tercios completos.

Un árbol B\* posee las siguientes propiedades:

* Cada nodo del árbol puede contener, como máximo, M descendientes y M - 1 elementos.
* La raíz no posee descendientes o tiene al menos dos.
* Un nodo con X descendientes contiene x - 1 elementos.
* Los nodos hojas tienen como mínimo (2M-1) /3 - 1 elementos.
* Los nodos que no son terminales ni raíz tienen como mínimo (2M-1) /3 elementos.
* Todos los nodos hoja se encuentran al mismo nivel.

La única excepción en el tratamiento de árboles B\* se produce cuándo se completa el primer nivel del árbol, ya que la raíz no tiene hermanos no es posible redistribuir por lo cual, al dividir, los dos nodos hijos generados estarán llenos a la mitad en lugar de en 2/3.

El proceso de inserción puede ser regulado de acuerdo a 3 políticas básicas:

1. Política de un lado.
2. Política de un lado \*u\* otro lado.
3. Política de un lado \*y\* otro lado.

Cada política determina, en caso de overflow, el nodo adyacente hermano a tener en cuenta.

* Cuando se utiliza la **política de un lado** se debe elegir qué lado se utilizará (izquierda o derecha), cuando un nodo no posee un hermano a la izquierda, y se usa la política de un lado a la izquierda, se podrá utilizar el nodo de la derecha, pero sólo en este caso y viceversa.
* Cuando se utiliza la política de **un lado u otro**, se puede utilizar cualquiera de los dos hermanos adyacentes.
* Cuando se utiliza la política de **un lado y otro**,se puede utilizar cualquiera de los dos hermanos adyacentes, la diferencia con la política anterior radica en que, a la hora de la división, si la redistribución no fue posible, se utilizarán los tres nodos para generar cuatro nodos completos en 3/4.

**Si bien los árboles B\* generan árboles de menor altura se necesita de un mayor número de operaciones de entrada-salida sobre disco para poder ser implementada.**

## Árboles B+

El árbol B+ incorpora las características del árbol B, pero tienen los nodos terminales entrelazados para permitir un acceso secuencial. Además, los datos están representados en los nodos terminales, mientras que los nodos no terminales funcionan como separadores.

Al comenzar la creación de un árbol B+, el único nodo disponible, la raíz, actúa tanto como punto de partida para búsquedas como para acceso secuencial. Cuando se produce una división, el elemento promocionado es una copia (la menor de los elementos mayores) y no el elemento en sí; en caso de dividir un nodo no terminal, si se promociona el elemento en sí y no una copia.

A la hora de borrar un elemento en un árbol B+ se debe borrar siempre el elemento de la hoja, y si el dato fuera un separador el mismo se mantendría ya que aún sería un separador válido.

### Árboles B+ de prefijos simples

Un árbol B+ de prefijos simples es un árbol B+ dónde los separadores están representados por la mínima expresión posible de la clave que permita decidir si la búsqueda se realiza hacía la izquierda o la derecha. Los prefijos intentan aprovechar mejor el espacio físico.

**Conclusiones sobre árboles balanceados:**

Los árboles balanceados son estructuras muy poderosas y flexibles para la administración de índices asociados a archivos de datos, con un nivel de performance muy interesante. Sin embargo, no deben considerarse como la única solución posible a todos los problemas. La mejor solución para un archivo va a depender del archivo en sí y del propósito de uso de dicho archivo. Por ejemplo, si el archivo es pequeño no es necesario, ni aconsejable, utilizar un árbol, ya que el mismo agrega un nivel de indirección por cada índice generado. Las características compartidas por la familia de árboles B son:

* Manejo de bloques o nodos lo que facilita el manejo de E/S y disminuye la cantidad de accesos a disco.
* Están balanceados.
* Crecen de abajo hacia arriba.
* Son bajos y anchos.
* Es posible implementar archivos de longitud variable para mejorar performance y uso de espacio.

# Hashing:

Un archivo con acceso directo, es un archivo en el cual cualquier registro puede ser accedido sin acceder antes a otros registros; a diferencia de los archivos series, en el cual un registro solo está disponible cuándo un registro predecesor ha sido procesado.

Hashing es un método que mejora la eficiencia obtenida con árboles balanceados, **asegurando en promedio un acceso para recuperar la información**.

Es una técnica de almacenamiento y recuperación que usa una función para mapear registros en direcciones de almacenamiento en memoria secundaria. Ventajas y beneficios:

* No requiere almacenamiento adicional. El mismo archivo es el que resulta disperso, no es necesario tener una estructura auxiliar que actúe como soporte para acceder rápidamente a la información.
* Facilita la inserción y eliminación rápida de registros en el archivo, es decir, que éstos procesos se realizan de una manera eficiente en términos de accesos a disco, en general, con un solo acceso a disco.
* Localiza registros dentro de un archivo con un sólo acceso a disco en promedio. Hay condiciones en las cuales ésta característica no es respetada.

Desventajas y limitaciones:

* No es posible aplicar la técnica en archivos con registros de longitud variables.
* No es posible obtener un orden lógico de los datos.
* No es posible tratar con clave duplicas. Por lo que no es aplicable la función de hash sobre una clave secundaria. Cada registro debe, a priori, residir en direcciones diferentes del archivo. Si se aceptara trabajar con claves secundarias ésta propiedad no se cumpliría.

### Tipos de dispersión

El método de hashing presenta dos alternativas para su implementación:

**Hashing con espacio de direccionamiento estático:** Es la política donde el espacio disponible para dispersar los registros de un archivo de datos está fijado previamente. Así, la función de hash aplicada a una clave da como resultado una dirección física posible dentro del espacio disponible para el archivo. Si se necesitará asignar más espacio al archivo, sería necesario redispersarlo.

**Hashing con espacio de direccionamiento dinámico**: Es la política donde el espacio disponible para dispersar los registros de un archivo de datos aumenta o disminuye en función de las necesidades de espacio que en cada momento tiene el archivo. Así la función de hash da como resultado un valor intermedio, que será utilizado para obtener una dirección física posible para el archivo. Estas direcciones físicas no están establecidas a priori y son generadas dinámicamente.

### Función de hash

Es una función que recibe como entrada una clave y produce una dirección de memoria, dentro de un determinado rango, dónde almacena el registro asociado a dicha clave en el archivo de datos. La función podría retornar cualquier valor , por eso, previo a retornar el resultado, es necesario mapear dicho valor a un rango determinado.

Cuándo una función retorna valores iguales para claves distintas, dichas claves son llamadas **sinónimos**, la existencia de sinónimos provoca colisiones, y dado que no es posible almacenar dos registros en el mismo espacio físico, es necesario minimizar el número de colisiones a una cantidad aceptable y de éste manera tratar dichas colisiones como una condición excepcional. Para llevar a cabo esto existen las siguientes alternativas:

* Distribuir los registros de la forma más aleatoria posible.
* Utilizar más espacio de disco, aunque esto traerá aparejado desperdicio de espacio.
* Ubicar o almacenar más de un registro por cada dirección física en el archivo, es decir que en cada dirección no se contenga un solo registro, sino un nodo donde es posible almacenar más de un registro. El tamaño del nodo es determinado por la posibilidad de transferencia de información en cada operación de entrada/salida desde RAM hacia disco y viceversa.

### Densidad de empaquetamiento:

Es la relación entre el espacio disponible para el archivo de datos y la cantidad de registros que integran dicho archivo (nº nodos \* nº registros por nodo). Cuanto mayor sea la DE, mayor será la posibilidad de colisiones, dado que en ese caso se dispone de menos espacio para dispersar registros. Por otra parte, cuando la DE se mantiene baja, se desperdicia espacio en el disco, lo que genera fragmentación. La DE no es constante.

La acción de aumentar el espacio de direcciones implica cambiar la función de hash para adaptar tus resultados al nuevo espacio disponible, por lo cual hay que reubicar a todos los registros ya almacenados.

A medida que la capacidad de cada nodo aumenta, la probabilidad de saturación disminuye. Para el caso de disponer de una DE del 75% con nodos que pueden almacenar hasta 100 registros, la probabilidad de colisiones menor a 0,09%.

### Tratamiento de overflow en Hashing estático

Un desborde ocurre cuándo se da una colisión y el nodo no dispone de capacidad para almacenarlo. Cuando esto ocurre deben realizarse dos acciones: encontrar lugar al registro y asegurarse de que el mismo pueda ser encontrado posteriormente en esa nueva dirección.

**Saturación progresiva:**

Consiste en almacenar el registro en la dirección siguiente más próxima al nodo donde se produce la saturación.

Durante el proceso de búsqueda, al buscar un elemento y no encontrarlo en el lugar dónde debería estar, y el nodo esta completo o marcado como borrado, se debe continuar la búsqueda en los nodos subsiguientes hasta encontrar el elemento, o hasta encontrar un nodo que no esté completo que no haya sido borrado. Es un método simple, aunque con una eficiencia limitada.

**Saturación progresiva encadenada:**

Es una variante de la saturación progresiva muy similar. Un elemento que se intenta ubicar en una dirección completa es direccionado a la inmediata siguiente con espacio disponible.

La diferencia radica en que, una vez localizada la nueva dirección, esta se encadena o enlaza con la dirección base inicial, generando una cadena de búsqueda de elementos, lo que, en líneas generales, presente mejores de performance. Sin embargo, requiere que cada nodo manipule información extra (la dirección del siguiente nodo).

El problema general que presenta la saturación progresiva es que, a medida que se producen saturaciones, los registros tienden a esparcirse en nodos cercanos. Esto podría provocar un exceso de saturación sectoriza.

**Doble dispersión:**

El método consiste en disponer de dos funciones de hash. La primera obtiene a partir de la llave la dirección de base, en la cual el registro será ubicado.

La segunda función de hash se utiliza cuando se produce un overflow. Esta función en vez de retornar una dirección, devuelve un desplazamiento, el cual se suma a la dirección base obtenida con la primera función, generando así la nueva dirección dónde se intentará ubicar el elemento. Si se generase nuevamente overflow se deberá sumar de manera reiterada el desplazamiento obtenido tantas veces como sea necesario hasta encontrar una dirección con espacio para albergar al registro.

La doble dispersión tiende a dispersar los registros en saturación a lo largo del archivo lo que puede significar grandes movimientos de la cabeza lectora del disco y aumentar la latencia entre pistas y por consiguiente el tiempo de respuesta.

**Área de desbordes por separado:**

Implementa dos tipos de nodo, aquellos direccionables por la función de hash y aquellos de reserva, que sólo podrán ser utilizados en caso de saturación pero que no son alcanzables por la función de hash.

### Tratamiento de overflow para Hashing dinámico

**Hash virtual.**

**Hash dinámico.**

**Hash extensible:**

Consiste en comenzar a trabajar con un único nodo para almacenar registros e ir aumentando la cantidad de direcciones disponibles a medida que los nodos se completan. Éste método, al igual que todos los que trabajan con espacio dinámico, no utiliza el concepto de DE.

El principal problema que se tiene con los métodos dinámicos en general y con éste en particular, es que las direcciones de nodos no están prefijadas a priori, y por la tanto la función de hash no puede retornar una dirección fija. Entonces es necesario cambiar la política de trabajo de la función de dispersión.

Para el método extensible, la función de hash retorna un string de bits. La cantidad de bits que retorna determina la cantidad máxima de direcciones a las que puede acceder el método. El método necesita también de una estructura auxiliar, ésta estructura es una tabla que se administra en memoria principal y que contiene la dirección física de cada nodo. En resumen:

* Se utilizan solo los bits necesarios de acuerdo con cada instancia del archivo.
* Los bits tomados forman la dirección del nodo que se utilizará.
* Si se intenta insertar en un nodo lleno, deben reubicarse todos los registros allí contenidos entre el nodo viejo y el nuevo, para ellos se toma un bit más.
* La tabla tendrá tantas entradas como 2^i, siendo i el número de bits actuales para el sistema.
* El proceso de búsqueda asegura encontrar cada registro en un solo acceso.

Si se desea asegurar siempre un acceso a disco para recuperar la información, la variante de hash extensible lo logra. El costo que debe asumirse es el mayor procesamiento cuando una inserción genera overflow.