Índices y Ordenamiento

Héctor Borges
Carmen Camacho
Marina Ramos
Gaudy Rodríguez

Agenda

- Indices estructurados con arboles:
  - Indices.
    - Alternativas para Entrada de Datos k* en un Índice
    - Alternativas para entrada de datos
  - Acceso Secuencial Indexado y Árbol B+
  - Indexed Sequential Access Method (ISAM)
    - Reglas generales para un índice ISAM
    - Creacion del Indice
  - Árbol B+: El Índice más usado
    - Formato de un Nodo
    - Búsqueda de un k* en un árbol
    - *Insertando entradas de datos en árboles B+*
    - Borrar una entrada de dato del árbol B+
    - Comparación por prefijo de llave
    - Bulk Loading

… Agenda

- Hash
  - Índices Estructurados con Hash
  - Hashing Estático
  - Deficiencias del Hashing Estático
  - Resumen
  - Hashing Dinámico
    - Hashing Extensible
      - La magia está en el directorio…
      - Ejemplos
      - Análisis de Casos
    - Hashing Lineal
      - Ejemplos
      - Para terminar…

… Agenda

- Ordenamiento Externo e Interno
  - Ordenamiento
    - Tipos de Ordenamiento
  - Ordenamiento Externo (MergeSort)
    - Código en C
    - MergeSort de Dos Vias
  - Ordenamiento Interno (QuickSort)
    - Código en C
    - Ejemplo
    - Ventajas y Desventajas
    - Condiciones Iniciales
  - HeapSort
    - Código en C
    - Ejemplo
    - Ventajas y Desventajas
    - Condiciones Iniciales
  - Bibliografía

Índices Estructurados con Árboles

Índices

- Un archivo nos permite recuperar registros:
  - Especificando *rid*, ó
  - Recorriendo todos los registros secuencialmente
- Algunas veces sera necesario recuperar los registros especificando el *valor de uno o mas campos*, Ej,
  - Buscar todos los estudiantes del curso "MAT"
  - Buscar estudiantes con promedio > 16
- Indices son archivos estructurados que nos permiten contestar eficientemente consultas basadas en valores.

Índices

- Cada indice tiene asociada una **llave**.
  - Los registros almacenados en un archivo de indices, llamados **entradas**, nos permiten encontrar registros dando un valor de búsqueda para la llave.
- Las técnicas de organizacion o estructuras de datos para los archivos de indices son llamados **metodos de acceso**, se conocen varios, incluyendo Arboles y estructuras hash.

Índices

- Un *índice* sobre un archivo acelera las selecciones que se hacen sobre *los campos llave de búsqueda* del índice.
  - Cualquier subconjunto de los campos de una relación puede ser la llave de búsqueda para un índice sobre una relación.
  - *La llave de búsqueda* no es la misma que la *llave* (mínimo conjunto de campos que identifican como único un registro en una relación).
- Un índice contiene una colección de *entradas de datos*, y permite una eficiente recuperación de todos los entradas de datos **k*** dando un valor **k** para la llave.

Alternativas para Entrada de Datos k* en un Índice

- Tres alternativas:
  - ✓ Registro de datos con un valor **k** para la llave.
  - ✓ <**k**, rid del registro de datos> con valor **k** para la llave
  - ✓ <**k**, lista de rids registros de datos> con valor **k** para la llave
- Elegir la alternativa para las entradas de datos corresponde a la técnica de indexamiento usada para localizar la entrada de datos dando un valor **k** a la llave.
  - Ejemplos de técnicas de indexamiento: Árboles B+, estructuras hash.
  - Comúnmente, los índices contienen información auxiliar que dirige la búsqueda de una entrada de datos deseada.

Alternativas para entrada de datos

- Alternativa 1:
  - Si se usa ésta, las estructura del índice **es una organización de archivo por registros de datos** (Como un archivo Heap o un archivo ordenado).
  - **En mayoría de las veces un índice** sobre una colección de registros puede usar la Alternativa 1. (Si no, los registros duplicados, conducen a almacenamiento duplicado y potencial inconsistencia)
  - Si **los registros de datos son muy grandes**, el número de páginas que contienen las entras de datos es mayor. Esto implica que el tamaño de la información auxiliar en un el índice es tan también grande.

Alternativas para entrada de datos

- Alternativas 2 y 3:
  - **Las entradas de datos son normalmente mas pequeñas que el registro de datos.** Entonces, es mejor que la Alternativa 1 con registros de datos grandes, especialmente si las llaves de búsqueda son pequeñas (La parte de la estructura del índice usada para dirigir la búsqueda es mucho mas pequeña que con la Alternativa 1)
  - Si se requiere mas de un índice sobre un archivo, **la mayoría de las veces un índice puede usar la Alternativa 1**; el resto debe usar la Alternativa 2 o 3.
  - **La alternativa 3 es más compacta que la 2**, pero provoca tamaño variable de las entradas de datos aunque las llaves de búsqueda sea de largo fijo.

Acceso Secuencial Indexado y Árbol B+

Índices para búsquedas y reportes

Hasta el momento hemos estudiado 2 tipos de estructuras de archivos:

- Acceso secuencial:
  - Archivos que pueden recorrerse secuencialmente (registros contiguos) regresando los registros en algún orden específico
- Indexamiento:
  - Archivos de datos que poseen archivos de índices por diferentes llaves que permiten acceder los datos de distintas maneras, el archivo de datos se encuentra sin ningún orden.

Alternativas para Entrada de Datos k* en un Índice

- Los árboles B y los árboles B+ son casos especiales de árboles de búsqueda. Un árbol de búsqueda es un tipo de árbol que sirve para guiar la búsqueda de un registro, dado el valor de uno de sus campos.
- En estructuras ISAM y árboles B+, las páginas que están en las hojas contienen las **entradas de datos**.
- Denotaremos una entrada de datos con el valor k para una llave de búsqueda **k***.
- Las páginas que no están en las hojas contienen **entradas de índice** de la forma <valor de llave, id de página> y son usadas para dirigir la búsqueda de la entrada de datos deseada (almacenada en una hoja).

Índices Tipo Árbol.

- Estático
  -> ISAM es una estructura estática
- Dinámico
  -> B-Tree, estructura dinámica.

Acceso Secuencial Indexado y Árbol B+

Problemas:

- Desafortunadamente el archivo secuencial es difícil de mantener.
- Los índices nos permiten hacer búsquedas rápidamente pero ...nos permiten recorrer el archivo secuencialmente ??
  - En realidad el B-Tree sí nos permite recorrer el archivo en orden, pero no en una manera sencilla ya que debemos hacer demasiados "seeks" para poder hacerlo.

Acceso Secuencial Indexado y Árbol B+

En la práctica en muchas situaciones se requieren de ambas cosas (búsquedas y accesos secuenciales), por ejemplo:

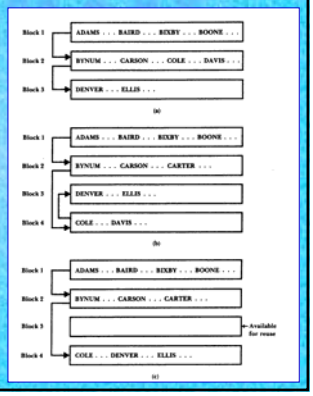
- En la universidad
  - Búsquedas: horarios
  - Secuenciales: Reportes de calificaciones que se imprimen
- En una empresa
  - Búsquedas: agenda telefónica, vacaciones
  - Secuenciales: cheques de nómina
- Las 2 soluciones existentes:
  - ISAM
  - B+ Tree

Indexed Sequential Access Method (ISAM)

Básicamente es la idea de tener un índice esparcido, de manera que el archivo de datos lo agrupamos por bloques y en cada bloque existe un "rango" ordenado de los Datos, como vemos en la figura:

ISAM

- Lo importante a resaltar aquí es que los bloques no necesariamente están formando una secuencia en el archivo, es decir, dentro del archivo de datos los bloques pueden estar desordenados, aquí el detalle es mantener un apuntador a cada bloque subsecuente como se muestra en la figura



Reglas generales para un índice ISAM

- Podemos iniciar el archivo con un bloque o varios (definiendo previamente los rangos y dejando bloques vacíos reutilizables).
- Cuando agregamos datos ubicamos el bloque correspondiente en base al índice en memoria y procedemos a agregar la información.
- Si al insertar notamos que el bloque ya esta lleno entonces tenemos que partirlo en 2, muy similar al btree solo que aqui no necesitamos un separador de ramas, simplemente creamos otro bloque, dividimos equitativamente los datos y ajustamos los apuntadores entre los bloques
- Para eliminar, ubicamos el bloque y eliminamos el registro
  Si al eliminar notamos que se pueden mezclar o unir dos bloques con pocos registros entonces lo podemos hacer y el bloque libre lo marcamos para reutilización

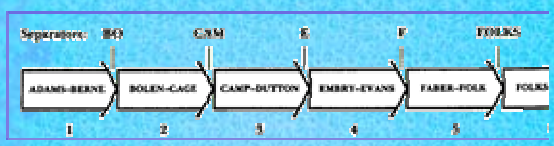
Creación del Índice

Puede ser muy simple como el que se muestra en la tabla.

Key	Block number
BERNE	1
CAGE	2
DUTTON	3
EVANS	4
FOLK	5
GADDIS	6

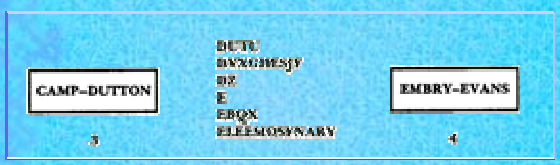
Donde sólo se guardan los máximos valores de cada bloque; quizás si las llaves fueran numéricas todo sería muy simple y podría quedar bien este esquema, pero en especial para cadenas (aunque también aplica para números) lo que en realidad se busca es tener separadores que sirvan para varios casos ya que si eliminamos el máximo valor de un bloque entonces tendremos que alterar el índice.

Creación del Índice



La figura muestra una primera solución de separadores basándose en los prefijos de las palabras. De esta manera nos evitamos el tener que estar actualizando constantemente el índice. Estos separadores de rangos nos son únicos, dependerá del programador el definirlos.

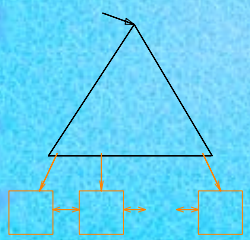
Creación del Índice



Muestra una lista de opciones para el caso de los bloques 3 y 4.

Árbol B+: El Índice más usado

- F = fan-out*, N = # páginas en las hojas.
- Cada nodo contiene **d** <= *m* <= 2**d** entradas.



Entradas de Índices

(Búsqueda directa)

Entrada de Datos

("Conjunto de secuencias")

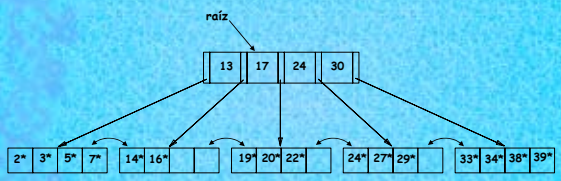
Formato de un Nodo

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

Ki son los valores de claves de búsqueda.

Pi son punteros a los hijos para nodos no hojas o punteros a registros o cajones de punteros a registros para nodos hojas.

Ejemplo de árbol B+



Búsqueda de un k* en un árbol

```
func find (search key value K) returns nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, K);                          // searches from root
endfunc

func tree_search (nodepointer, search key value K) returns nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if K < K_1 then return tree_search(P_0, K);
    else,
        if K ≥ K_m then return tree_search(P_m, K);        // m = # entries
        else,
            find i such that K_i ≤ K < K_{i+1};
            return tree_search(P_i, K)
endfunc
```

Insertando entradas de datos en árboles B+

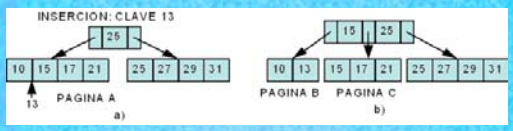
- Se busca la hoja correcta L.
- Se coloca la entrada de dato sobre L.
- Esto puede suceder recursivamente.
- Las divisiones "agrandan" al árbol; las divisiones de la raíz incrementan la altura del árbol.

```
proc insert (nodepointer, entry, newchildentry)
// Inserts entry into subtree with root '*nodepointer'; degree is d;
// 'newchildentry' is null initially, and null upon return unless child is split

if *nodepointer is a non-leaf node, say N,
    find i such that K_i ≤ entry's key value < K_{i+1};        // choose subtree
    insert(P_i, entry, newchildentry);                         // recursively, insert entry
    if newchildentry is null, return;                          // usual case; didn't split child
    else,                                                      // we split child, must insert *newchildentry in N
        if N has space,                                        // usual case
            put *newchildentry on it, set newchildentry to null, return;
        else,                                                  // note difference wrt splitting of leaf page!
            split N:                                           // 2d+1 key values and 2d+2 nodepointers
            first d key values and d+1 nodepointers stay,
            last d keys and d+1 pointers move to new node, N2;
            // *newchildentry set to guide searches between N and N2
            newchildentry = & ((smallest key value on N2, pointer to N2));
            if N is the root,                                  // root node was just split
                create new node with (pointer to N, *newchildentry);
                make the tree's root-node pointer point to the new node;
            return;

if *nodepointer is a leaf node, say L,
    if L has space,                                            // usual case
    put entry on it, set newchildentry to null, and return;
    else,                                                      // once in a while, the leaf is full
        split L: first d entries stay, rest move to brand new node L2;
        newchildentry = & ((smallest key value on L2, pointer to L2));
        set sibling pointers in L and L2;
        return;
endproc
```

Insertar dentro del árbol B+



Inserción de la clave 13 en un árbol B+.

a) Antes de insertar la clave.

b) Después de insertarla.

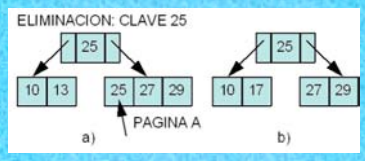
rrar una entrada de dato del árbol

- Iniciar en la raíz, buscar la hoja L a la que pertenece la entrada.
- Remover la entrada.
- Si se unen, se debe borrar la entrada (puntero a *L* o el hermano) de el padre de *L*.
- La unión se podría propagar a la raíz, y decremento la altura.

```
proc delete (parentpointer, nodepointer, entry, oldchildentry)
// Deletes entry from subtree with root '*nodepointer'; degree is d;
// 'oldchildentry' null initially, and null upon return unless child deleted
if *nodepointer is a non-leaf node, say N,
    find i such that K_i ≤ entry's key value < K_{i+1};      // choose subtree
    delete(nodepointer, P_i, entry, oldchildentry);          // recursive delete
    if oldchildentry is null, return;                        // usual case: child not deleted
    else,                                                    // we discarded child node (see discussion)
        remove *oldchildentry from N,                        // next, check minimum occupancy
        if N has entries to spare,                           // usual case
            set oldchildentry to null, return;               // delete doesn't go further
        else,                                                // note difference wrt merging of leaf pages!
            get a sibling S of N;                            // parentpointer arg used to find S
            if S has extra entries,
                redistribute evenly between N and S through parent;
                set oldchildentry to null, return;
            else, merge N and S                              // call node on rhs M
                oldchildentry = &(current entry in parent for M);
                pull splitting key from parent down into node on left;
                move all entries from M to node on left;
                discard empty node M, return;

if *nodepointer is a leaf node, say L,
    if L has entries to spare,                               // usual case
        remove entry, set oldchildentry to null, and return;
    else,                                                    // once in a while, the leaf becomes underfull
        get a sibling S of L;                                // parentpointer used to find S
        if S has extra entries,
            redistribute evenly between L and S;
            find entry in parent for node on right;          // call it M
            replace key value in parent entry by new low-key value in M;
            set oldchildentry to null, return;
        else, merge L and S                                  // call node on rhs M
            oldchildentry = &(current entry in parent for M);
            move all entries from M to node on left;
            discard empty node M, adjust sibling pointers, return;
endproc
```

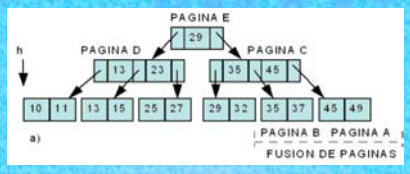
Eliminando Claves en un árbol B+

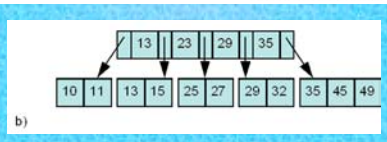


Eliminación de la clave 25

a) Antes de eliminar la clave.

b) Después de eliminarla.





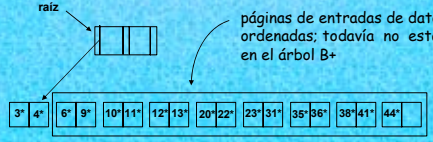
Eliminación de la clave 37
a) Antes de eliminarla.
b) Después de eliminarla.

Comparación por prefijo de llave

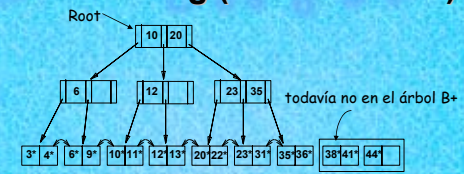
- Es importante para incrementar.
- Los valores de las llaves en las entradas de los índices solamente `dirigen el tráfico'; y a menudo puede comprimirlos.
- Insertar/eliminar: se debe modificar convenientemente.

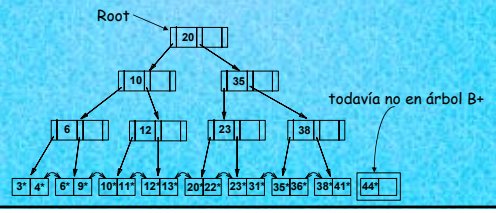
Bulk Loading a un árbol B+

- Si se tiene una colección grande de registros, y queremos crear un árbol B+ sobre algún campo, construirlo con repetidas inserciones seria muy lento, con *Bulk Loading* se puede realizar mas eficientemente.
- *Inicialización*: Ordenar todas las entradas de datos, insertar un puntero a la primera página (hoja) en una nueva página (raíz).



Bulk Loading (Continuación)





Bulk Loading (Continuación)

Opción 1: Múltiples inserts.

Lento.

No permite almacenamiento secuencial de hojas.

Opción 2: *Bulk Loading*

Tiene ventajas sobre el control de concurrencia.

Pocas operaciones I/O durante la construcción.

Las hojas se almacenaran secuencialmente (y encadenadas, por supuesto).

Se puede controlar el "factor de llenado" de las páginas.

Hash

Índices Estructurados con Hash

✓ Los índices tipo Hash son mejores para *búsquedas por igualdad* y no soportan búsquedas por rango.

✓ Existen técnicas hash estáticas y dinámicas como en los árboles ISAM y B+.

✓ El hashing se puede usar no solo para la organización de archivos, sino también para la creación de la estructura de índices.

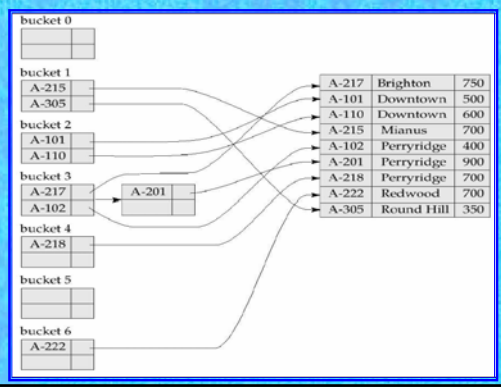
✓ Un índice hash organiza claves de búsqueda, con sus punteros a registros asociados, dentro de una estructura de archivo de hash.

✓ Estrictamente hablando, los índices de hash siempre son índices secundarios

✓ Si el propio archivo es organizado con hash, un índice de hash primario separado usando la misma clave de búsqueda es innecesario.

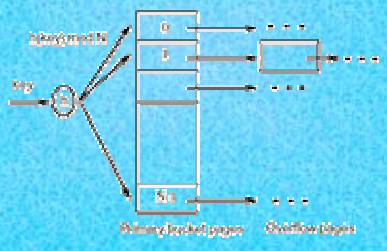
✓ Las inserciones y borrados requieren más de una operación.

Ejemplo de un Índice de Hash



Hashing Estático

h(*k*) mod M = cubo al que pertenece una entrada de dato con llave k. (M= # de cubos)



Hashing Estático

✓ Los cubos contienen *entradas de datos*.

✓ La función hash se aplica al valor de la llave de búsqueda del campo de un registro r.

✓ h(key)=(a*key+b) generalmente trabaja bien.

✓ Muchas páginas overflow encadenadas pueden degradar el rendimiento.

✓ El hashing extensible y lineal: son técnicas para corregir el problema.

✓ La función de hash es utilizada para localizar registros para su acceso, inserción y borrado.

✓ Registros con diferentes valores de claves de búsqueda pueden ser mapeados al mismo cajón, de esta forma una vez accedido el cajón se debe hacer una búsqueda secuencial para encontrar el registro.

Deficiencias del Hashing Estático

✓ En el hashing estático, la función *h* mapea los valores de clave de búsqueda a un conjunto fijo *B* de cajones.

✓ Las bases de datos crecen con el tiempo. Si el número inicial de cajones es demasiado chico, la performance se degradará porque abra demasiados cajones de desbordamiento.

✓ Si el tamaño del archivo va ser grande en un futuro, se pueden tener un gran numero de cajones, pero así se estará desperdiciando espacio al principio.

✓ Si la base de datos se achica, nuevamente se estará gastando espacio.

✓ Una opción es reorganizar periódicamente el archivo con una nueva función de has, pero esto es muy caro.

✓ Este problema puede ser evitado utilizando técnicas que permitan que el numero de cajones sean modificados dinámicamente.

Resumen

❀ Árboles B+ es una estructura dinámica.

❀ Inserts/deletes mantienen el árbol balanceado por altura; costo log F N.

❀ Un valor grande para fan out (**F**) significa una profundidad rara vez mayor a 3 o 4.

❀ Casi siempre es mejor que tener un archivo ordenado.

❀ Comúnmente se ocupa en un 67% en promedio.

❀ Generalmente es preferible a una estructura ISAM.

❀ Se deben hacer consideraciones sobre *bloqueos*.

Resumen (II)

❀ Se ajusta muy bien al crecimiento.

❀ Si las entradas de datos son registros de datos, las divisiones pueden cambiar los el valor de rid!.

❀ La compresión de las llaves incrementa fan-out, y reduce la altura.

❀ Bulk loading puede ser mucho mas rápido que repetidas inserciones para crear un árbol B+ sobre un conjunto de datos muy grande.

❀ Es el índice mas usado en los DBMS por su versatilidad. Además de ser uno de los componentes mas optimizados del DBMS.

Resumen (III)

Hash

- Muy buen comportamiento en la inserción y en la recuperación por condiciones de igualdad.

- No funciona bien para condiciones con relaciones de orden

Arboles B y B+:

- Buen comportamiento en recuperación tanto por condiciones de igualdad como de orden. Buen comportamiento en la inserción

- Ocupa más disco.

Hashing Dinámico

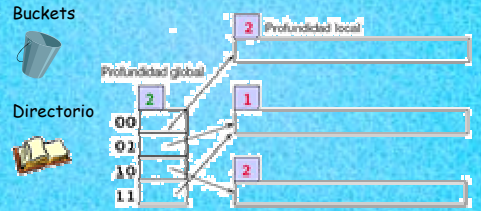
Resuelve las deficiencias del Hashing estático, como el deterioro del rendimiento por la ocurrencia de muchos overflows o la saturación de secciones de la tabla de hash.

Hablaremos de los dos enfoques más importantes:

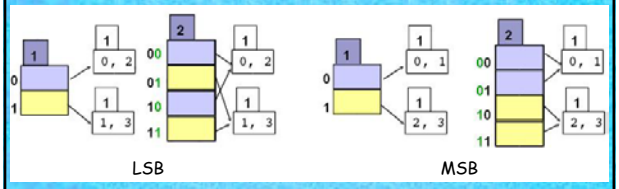
- Hashing Extensible
- Hashing Lineal

Hashing Extensible

- Técnica de indexación que evita los recorridos secuenciales y las áreas de saturación separadas.
- Elementos principales:
  - Buckets
  - Directorio

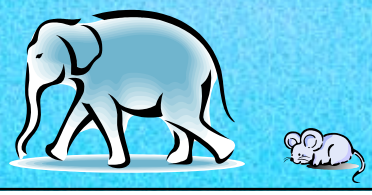


Implementaciones



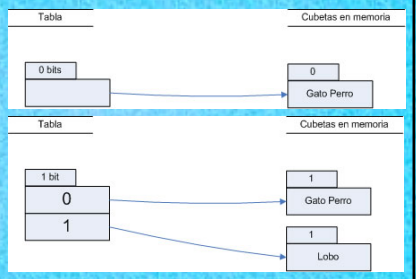
La magia está en el directorio…

- Modificar el archivo para duplicar los buckets sería muy costoso.
- El directorio es mucho más pequeño que el archivo.
- Es muy probable que el directorio quepa en memoria principal: Un archivo de 100 MB, a 100 bytes por registro, con páginas de 4K conteniendo 1 millón de registros (como entrada de datos) y 25 mil elementos de directorio



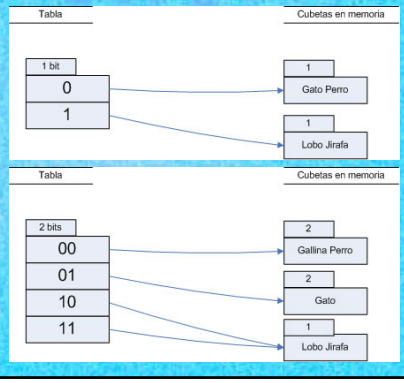
Ejemplo

Orden	Clave	Bits
1	Perro	0010
2	Gato	0101
3	Lobo	1001
4	Jirafa	1100
5	Gallina	0001
6	Oso	1010
7	Burro	1110
8	Vaca	0110
9	Caballo	0111
10	Ave	1111



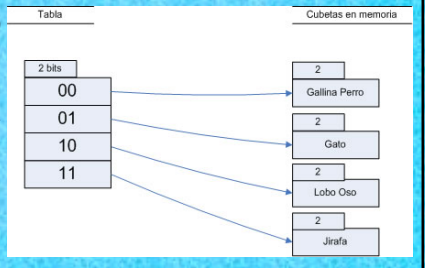
Ejemplo

Orden	Clave	Bits
1	Perro	0010
2	Gato	0101
3	Lobo	1001
4	Jirafa	1100
5	Gallina	0001
6	Oso	1010
7	Burro	1110
8	Vaca	0110
9	Caballo	0111
10	Ave	1111



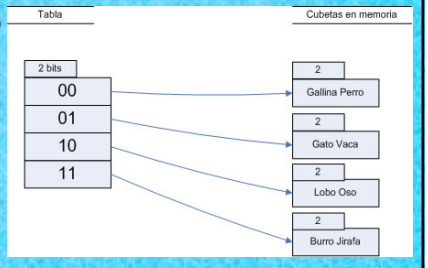
Ejemplo

Orden	Clave	Bits
1	Perro	0010
2	Gato	0101
3	Lobo	1001
4	Jirafa	1100
5	Gallina	0001
6	Oso	1010
7	Burro	1110
8	Vaca	0110
9	Caballo	0111
10	Ave	1111



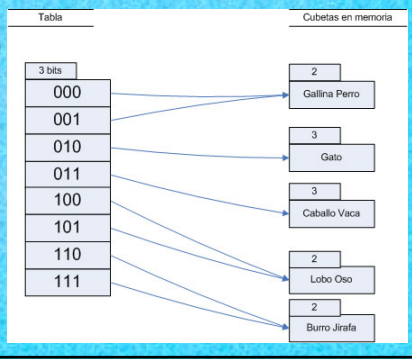
Ejemplo

Orden	Clave	Bits
1	Perro	0010
2	Gato	0101
3	Lobo	1001
4	Jirafa	1100
5	Gallina	0001
6	Oso	1010
7	Burro	1110
8	Vaca	0110
9	Caballo	0111
10	Ave	1111



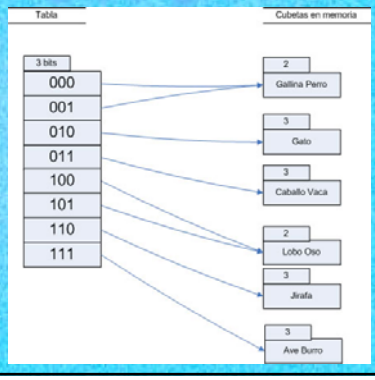
Ejemplo

Orden	Clave	Bits
1	Perro	0010
2	Gato	0101
3	Lobo	1001
4	Jirafa	1100
5	Gallina	0001
6	Oso	1010
7	Burro	1110
8	Vaca	0110
9	Caballo	0111
10	Ave	1111



Ejemplo

Orden	Clave	Bits
1	Perro	0010
2	Gato	0101
3	Lobo	1001
4	Jirafa	1100
5	Gallina	0001
6	Oso	1010
7	Burro	1110
8	Vaca	0110
9	Caballo	0111
10	Ave	1111



Análisis de Casos

- Inserción:
  - El bucket está lleno y tiene que ser desdoblado en dos sin afectar al directorio.
  - El directorio se duplica al desdoblarse un bucket (añadiendo un nuevo bit al espacio de direcciones).
  - Que el directorio y el bucket tengan espacio suficiente para aceptar el registro sin modificar su estructura.
- Eliminación:
  - Si los buckets no quedan vacíos no se hace nada.
  - Si un bucket queda vacío, deja de ser diseccionado. El directorio redirecciona su apuntador al bucket al que apuntaba antes del desdoblamiento (donde coinciden los bits de direccionamiento). Para terminar se le resta 1 a
  - Si la profundidad local de todos los buckets es menor que la profundidad global, es posible reducir el directorio a la mitad eliminando el MSB de direccionamiento y reagrupar los pares de buckets que compartan hashKey

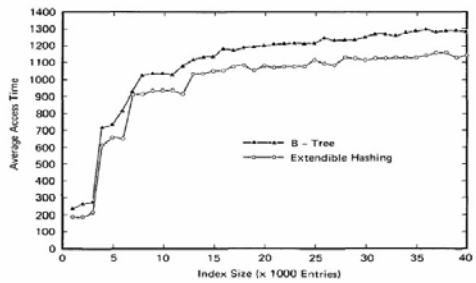


Fig. 12. Access time. Detailed simulation of access times averaged over 1000 accesses for index sizes in multiples of 1000 entries. The time to fetch a page is 1000. The maximum number of entries on a page is 400.

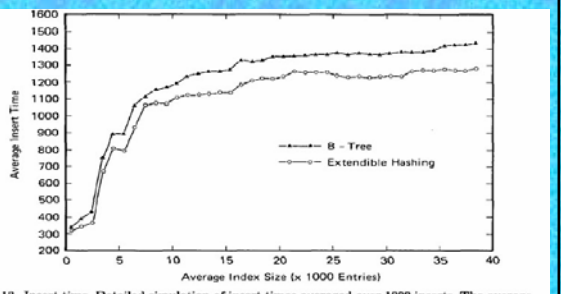


Fig. 13. Insert time. Detailed simulation of insert times averaged over 1000 inserts. The average insert time is plotted against the index size after 500 of the 1000 inserts. The time to fetch a page is 1000. The maximum number of entries on a page is 400.

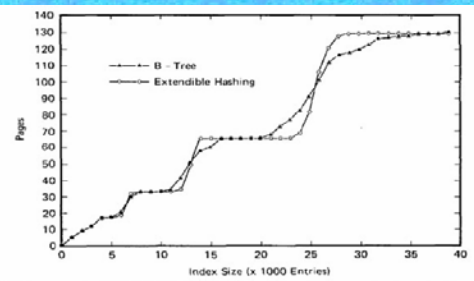


Fig. 14. Space required. Detailed simulation of the number of pages required as a function of the index size. The maximum number of entries on a page is 400.

Hashing Lineal

- Esquema de índices asociativos dinámicos inventado por Witold Litwin y publicado en una convención mundial de bases de datos en 1980. Este representa una alternativa mejor al hash extensible, de hecho no existe un algoritmo que supere su desempeño.
- En general el hashing lineal viene definido por los siguientes elementos:
  - **M** es el número de buckets primarios en el archivo.
  - **Split pointer** (n) controla cuál bucket es el siguiente en desdoblarse.
  - **Política de desdoblamiento** que especifica la condición que dispara el desdoblamiento del bucket señalado por el split pointer (generalmente un porcentaje de carga de los buckets).
  - Cada vez que el archivo que almacena la tabla de hash duplica su tamaño se dice que ocurre una **expansión completa**, en cuyo caso el split pointer se retorna a su condición inicial (0).
  - También se debe definir una **política de resolución de colisiones** (CHP), que puede ser de encadenamiento (trabajando con overflows), o no permitir el encadenamiento y aumentar el directorio cada vez que ocurre una colisión

Ejemplo

Insertar 10 registros con las siguientes claves hash:

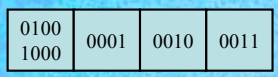
0011, 0010, 0100, 0001, 1000, 1110, 0101, 1010, 0111, 1100

- M=4
- Política de desdoblamiento: dividir cuando la carga de los buckets sea > 0.7
- CHP: encadenamiento

0 1 2 3

Ejemplo

Los primeros 5 son intuitivos:

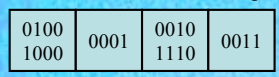


0 1 2 3

El factor de carga permanece por debajo de 0.7. el desdoblamiento no se ha disparado. n = 0

Ejemplo

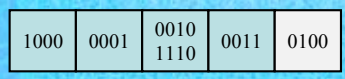
- Cuando se inserta el 6to registro tenemos:



0 1 2 3

n=0 antes de desdoblar

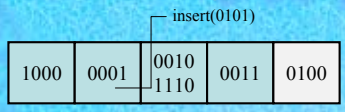
- Pero el factor de carga es 6/8= 0.75 > 0.70, por lo que se debe desboblar el bucket utilizando (h_1 = Key mod 2M):



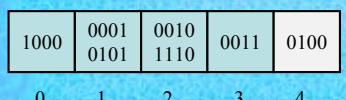
0 1 2 3 4

n=1 luego de desdoblar. El factor de carga es: 6/10=0.6 por lo que no se desdobla más

Ejemplo



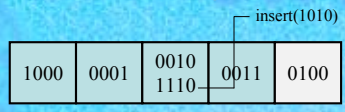
0 1 2 3 4



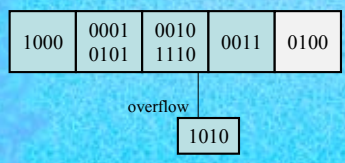
0 1 2 3 4

n=1
Factor de carga: 7/10=0.7
No se desdobla

Ejemplo

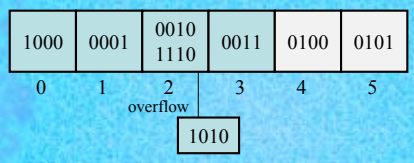


0 1 2 3 4



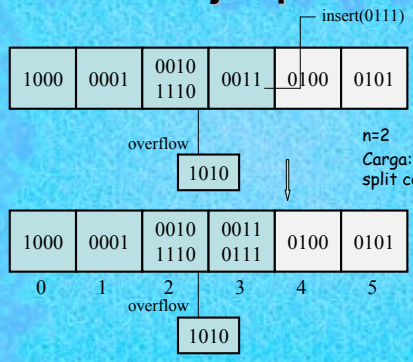
n=1
factor: 8/10=0.8
split con h_1.

Ejemplo



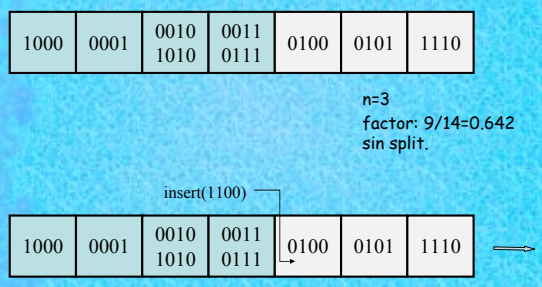
n=2
factor: 8/12=0.66
sin split

Ejemplo



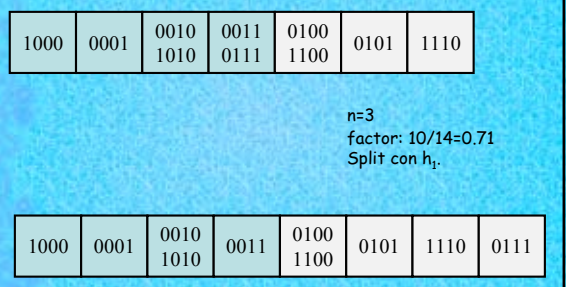
n=2
Carga: 9/12=0.75
split con h_1.

Ejemplo



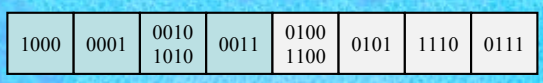
n=3
factor: 9/14=0.642
sin split.

Ejemplo



n=3
factor: 10/14=0.71
Split con h_1.

Ejemplo



n=4
factor: 10/16=0.625
sin split.

- En este punto todos los buckets M=4 se han desdoblado.
- El split pointer, n, debe ser igualado a 0.
- De ahora en adelante se utiliza el siguiente par de funciones de hashing h_1 y h_2 :
  $h_1(k) = k \bmod 2M$ y $h_2(k) = k \bmod 4M$.

Para terminar…

- Éste enfoque maneja el problema de muchas páginas de overflow encadenadas sin usar un directorio.
- Maneja bien valores duplicados.
- En éste modelo el espacio de direcciones crece y decrece dinámicamente según sea necesario.
- Puede soportar cualquier número de inserciones y eliminaciones sin sufrir deterioro en su desempeño de acceso o espacio en memoria.
- En general, un registro puede ser encontrado con un solo acceso con una política de carga >0.9. para una política >0.8 la media es de 1.7 accesos. Debe haber un equilibrio entre la cantidad de accesos y la política de carga porque una política muy estricta con una función hash regular puede producir un exceso de overflows y aumentar la cantidad de accesos para encontrar un registro.

Ordenamiento Externo e Interno

Ordenamiento

- El ordenamiento es la operación de arreglar los registros de una tabla en algún orden secuencial, de acuerdo a un criterio especifico.
- Ventajas:
  - Facilitar las búsquedas de los elementos del conjunto ordenado.
  - Útil para eliminar copias duplicadas en una colección de registros.

Tipos de Ordenamiento

- Interno:
  - Los valores a ordenar están en memoria principal.
  - El tiempo de acceso a los elementos es constante.
  - Métodos de ordenamiento: Quicksort, Insertsort, Heapsort, etc.
- Externo:
  - Los valores a ordenas están en memoria secundaria.
  - Se transfieren bloques de información a memoria principal, se ordena y se regresa a memoria secundaria.
  - El tiempo de acceso depende de la ultima posición accedida.
  - Métodos de ordenamiento: Mergesort y Intercalación.

Ordenamiento Externo

Mergesort (Ordenamiento por Mezcla)

- Algoritmo que ordena una secuencia, disponiendo los términos en sus dos mitades según una condición determinada y aprovechando el paradigma algorítmico Divide y Vencerás.
- Complejidad O(nLog n).
- Es la base de los métodos de ordenamiento externo. Se utiliza un enfoque no recursivo, para optimizarlo, de la siguiente forma:
  - Se generan archivos de entrada, los mas grande posible y se va leyendo a trozos en memoria principal, ordenando a través del Quicksort (ordenamiento interno).
  - Se hace una mezcla múltiple, en vez de mezclar dos archivos. Como en cada iteración hay k candidatos a ser el siguiente elemento en salir, y siempre hay que extraer al mínimo de ellos y sustituirlo en la lista de candidatos por su sucesor, la estructura de datos apropiada para ello es un heap.

Código en C

Dadas dos secuencias ordenadas a[M] y b[N], entonces:

```
int i=1;
int j=1;
a[M+1] = INT_MAX; //Le asigna el máximo de los enteros
b[N+1] = INT_MAX;
For (int k=1;k<=M+N;k++) //Recorrido en forma lineal
{
   if (a[i]<b[j]) //Si el contenido de a < que el de b
   {
      C[k]= a[i]; // Guarda el menor en la Secuencia C
      i++;//aumenta i
   }
   else
   {
      C[k] = b[j];
      j++;
   }
}
```

Complejidad O(N+M), ya que N+M es la cantidad de iteraciones que realiza.

Mergesort de dos vías

- Se llama así ya que mezcla dos secuencia ordenadas, en una con la misma característica.
- Utiliza 3 Buffers:
  -2 para la entrada de datos, de cada una de las secuencias.
  -El ultimo, coloca la secuencia resultante en disco.
- Este algoritmo se generaliza para mezclar N archivos de entrada, Utilizando B Buffers y haciendo $\lceil N/B \rceil$ corridas del algoritmo.

Parte entera

Ordenamiento Interno

Quicksort

- Método mas rápido conocido.
- Naturalmente recursivo.
- Basado en la Técnica Divide y Vencerás.

Su algoritmo fundamental es:

- Se elige el pivote. Puede ser cualquiera, y lo llamaremos elemento de división ó pivote.
- Buscas la posición que le corresponde en la lista ordenada.
  - *Si el pivote termina en el centro de la lista, este sería el mejor caso, ya que se dividirían dos mitades iguales.*
  - *Si termina en el extremo de la lista, sería el peor caso. Ocurre en listas ordenadas o casi ordenadas.*

Quicksort

- Para encontrar la posición del pivote, se utilizan dos índices, i y j, donde i recorre desde la izquierda (primer elemento) y j desde la derecha (ultimo elemento). Simultáneamente.
- Si lista[i] > pivote y lista[j] < pivote, intercambiamos.
- Repetimos hasta que se crucen. En ese momento, insertamos el pivote en lista[i] ya que todos los elementos menores se encuentran a su izquierda y los mayores a su derecha.
- Realizamos la recursividad hasta que el largo sea igual a 1.

Código en C

- Parámetros: - lista, secuencia a ordenar.
  - inf, índice inferior.
  - sup, índice superior.

```
// Inicialización de variables

elem_div = lista[sup]; // Elegimos el elemento divisor
i = inf - 1; //contador de la izquierda
j = sup; //contador de la derecha.
band = 1; //contador aux

//Verificamos que no se crucen los límites

if (inf >= sup)
   return;
```

Código en C

```
//  Clasificamos la sublista

while (band)
   while (lista[++i] > elem_div);
   while (lista[--j] < elem_div);
      if (i < j) //si no se Cruzan
         temp = lista[i];
         lista[i] = lista[j];
         lista[j] = temp;
      else   //se cruzan
         band = 0;

//Clasificamos el elemento de división en su pos_final
temp = lista[i];
lista[i] = lista[sup];
lista[sup] = temp;
quick (lista, inf, i - 1);//Recursividad con la inferior
quick (lista, i + 1, sup);//Recursividad con la superior
```

Ejemplo

- Tenemos la secuencia, donde el pivote será 4.
- Comparamos 1 y 5. No se cruzan los índices, y 1 es menor que 4 y 5 es mayor. Intercambiamos.
- Avanzamos por la izq. y der. 3 y 2 son menores que 4, se mantienen.
- Avanzamos por la izquierda. 7 es mayor y 2 es menor que 4, intercambiamos.
- En este momento se cruzan. Y hacemos el intercambio, donde se coloca el pivote en su posición final.
- Se llama a la función recursiva, con cada sublista y al final se retornan sin hacer cambios

5	3	7	6	2	1	4
1	3	7	6	2	5	4
1	3	7	6	2	5	4
1	3	7	6	2	5	4
1	3	2	4	7	5	6

{1 3 2 → 1 2 3} {7 5 6 → 5 6 7}

1	2	3	4	5	6	7

Ventajas y Desventajas

- Ventajas:
  - Es muy rápido.
  - No requiere memoria adicional.
- Desventajas:
  - La implementación es complicada.
  - Por la recursividad utiliza muchos recursos.
  - Hay mucha diferencia entre el peor y mejor caso.

Quicksort – Condiciones Iniciales

- El archivo a clasificar ya está en memoria, dispuesto en slots contiguos.
- Los registros del archivo los denominaremos R1, R2, R3, ..., Ri, ..., Rn donde $1 \leq i \leq n$.
- Existen en memoria n slots, donde en cada uno de ellos se dispone un registro del archivo a clasificar.
- Todos los registros tienen una clave Ki, donde i es $1 \leq i \leq n$.
- Los registros están desordenados respecto de la clave mencionada.
- La clasificación será hecha en los mismos slots donde están los registros, conmutando registros hasta que queden ordenados.
- El proceso no requiere espacio adicional.
- Adaptamos el algoritmo y lo aplicamos.

Heapsort

- Ordenamiento por Montículos.
- Montículo: un montículo de tamaño n es como un árbol binario completo de n nodos, con las siguientes características:

1. Los valores están acomodados en los nodos, de tal manera que, para cada nodo i, $(k(i) \leq k(j)$ ó $(k(i) \geq k(j))$, dependiendo de la relación de orden escogida. Es decir, al recorrer el árbol, las claves están en orden descendente o ascendente.
2. El árbol se llena de izquierda a derecha, si algún nodo no esta en el mismo nivel que el resto, entonces estará lo más a la izquierda posible del árbol. Luego que los valores están ordenados en el montículo, se extraen de la cima del montículo de uno en uno hasta obtener el arreglo ordenado.

Código en C

```
void heapSort(int valores[], int tam_arreglo)
{
   int i, temp;

   for (i = (tam_arreglo / 2)-1; i ≥0; i--)
      reheap(valores, i, tam_arreglo);

   for (i = tam_arreglo-1; i ≥ 1; i--)
   {
      temp = valores[0];
      valores[0] = valores[i];
      valores[i] = temp;
      reheap(valores, 0, i-1);
   }
}
```

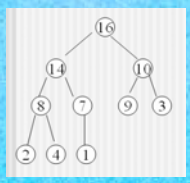
Código en C

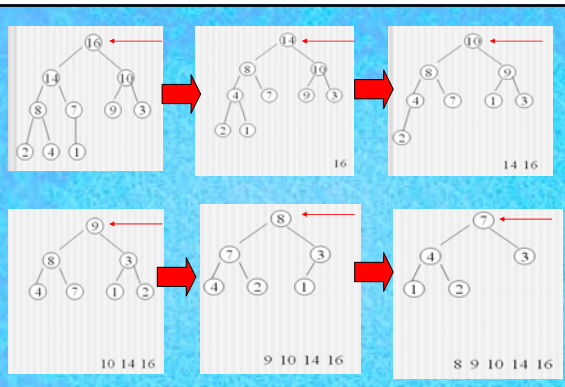
```
void reheap(int valores[], int raiz, int tam_arreglo)
{
   int band, max, temp;
   band = 0;
   while ((raiz*2 <= tam_arreglo) && (!band))
   {
      if (raiz*2 == tam_arreglo)
         max = raiz * 2;
      else if (valores [raiz * 2] < valores[raiz * 2 + 1])
         max = raiz* 2;
      else
         max = raiz * 2 + 1;
      if (valores[raiz] > valores[max])
      {
         temp = valores[raiz];
         valores[raiz] = valores[max];
         valores[max] = temp;
         raiz = max;
      }
      else
         band = 1;
   }
}
```

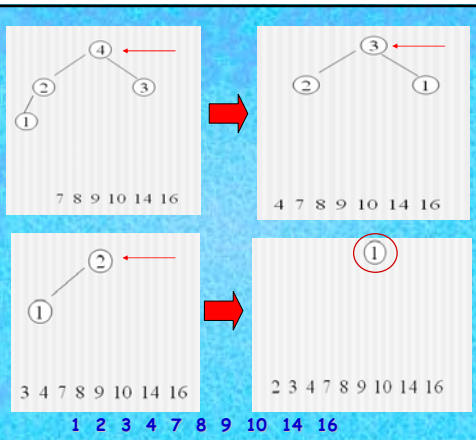
Ejemplo

Tenemos la siguiente secuencia y su árbol correspondiente y queremos ordenarlo desde 1 hasta 16.

A partir de esta situación inicial se realiza el bucle for de heapsort y se comienza con i = 9 y termina cuando i= 0.







Ventajas y Desventajas

- Ventajas:
  1. Su desempeño es en promedio como el del quicksort, ya que se comporta mejor que este en los peores casos. Aunque tiene mejor desempeño que cualquier otro método de ordenamiento, es mas difícil de programar.
  2. No necesita espacio adicional.
- Desventajas:
  1. Es lento, en la practica.
  2. No es estable.

Heapsort – Condiciones Iniciales

- El archivo a clasificar ya está en memoria, dispuesto en slots contiguos.
- Los registros del archivo los denominaremos R1, R2, R3, ..., Ri, ..., Rn donde 1≤ i ≤ n.
- Existen en memoria n slots, donde en cada uno de ellos se dispone un registro del archivo a clasificar.
- Todos los registros tienen una clave Ki, donde i es 1 ≤ i ≤ n.
- Los registros están desordenados respecto de la clave mencionada.
- La clasificación será hecha valiéndose de almacenamiento adicional, de cantidad fija.
- Aplicamos y adaptamos el algoritmo.

Bibliografía

- Database Management Systems, Ramakrishnan.
- Fundamentos de Bases de Datos, Silberschatz.
- http://es.wikipedia.org/wiki/Merge_sort
- http://es.wikipedia.org/wiki/Ordenamiento_externo
- http://informatica.futuroprofesional.cl/modules/tutorials/tutoriales/orden/mergesort.html
- http://rudomin.cem.itesm.mx/~erik/AnalisisAlgoritmos/OrdenamientosIRG.ppt
- http://www.conclase.net/c/orden/quicksort.html
- http://www.dcc.uchile.cl/~cc30a/apuntes/Ordenacion/


Hughes Aircraft