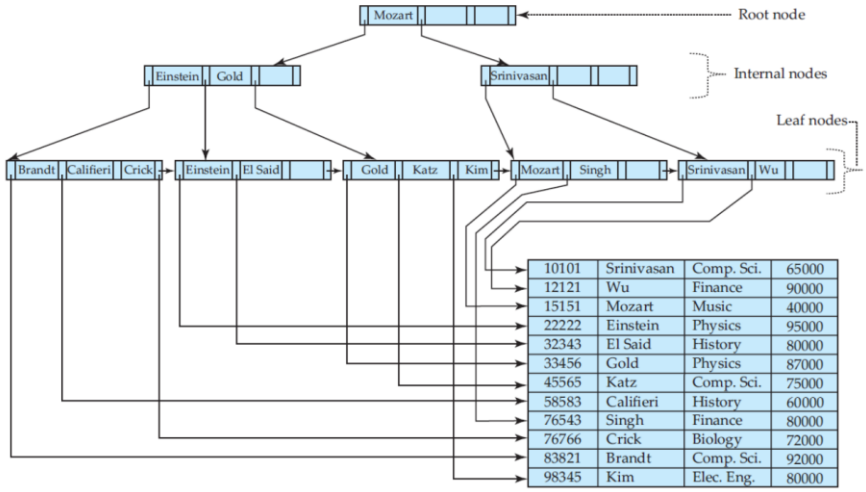
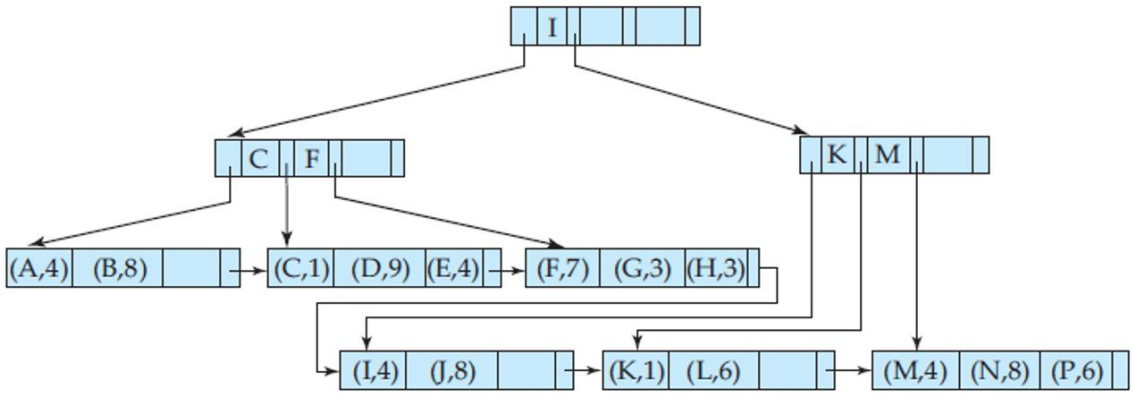
BDII 7 – Indexación y Asociación

Extensiones del árbol B+



Organización de archivos con árboles B+

- Tradicionalmente, los nodos **hoja** de los árboles B+ almacenan **punteros** a los registros en el disco.
  - Una mejora es la ORGANIZACIÓN DE ARCHIVOS CON ÁRBOLES B+, en la que en los nodos **hoja** de los árboles B+ se almacenan los **registros** en sí.
    - Así, se elimina el acceso intermedio al archivo de datos, con lo que:
      - ✓ Se reduce el número de **accesos a disco**.
      - ✓ Se simplifica el **diseño de los archivos**.
- Normalmente los registros son más grandes que los punteros, por lo que:
  cantidad máx. de **registros** que se puede almacenar en una **hoja** (n) < cantidad máx. de **punteros** que se puede almacenar en un **nodo interno**.
  - ↳ De todas formas, se sigue requiriendo que la cantidad de registros almacenados en una hoja sea $\geq n/2$ (que estén al menos **medio llenas**).



Reconstrucción del árbol para mejorar la eficiencia – también si no se usa organización de archivos con árboles B+

- Al construir el árbol B+ se puede hacer que los nodos hoja adyacentes en el árbol sean contiguos también en el disco, lo que permite que una **búsqueda secuencial** por el **índice** se realice como una **lectura secuencial** en el **disco**.
- Sin embargo, conforme se van **redistribuyendo** los nodos (por inserciones y borrados), aunque sigan organizados lógicamente en el árbol, no tienen por qué estarlo físicamente en el disco. Así, aunque dos hojas sean adyacentes en el árbol, pueden estar en lugares diferentes del disco.
  - Para que una búsqueda secuencial de hojas se siga correspondiendo con una búsqueda secuencial en el disco se puede volver a **reconstruir el índice** para volver a alinear la estructura lógica del árbol con el almacenamiento físico.

Objetos grandes

- Las organizaciones de archivos con árboles B+ también sirven para guardar **objetos grandes** (como los $clob$ y los $blob$) mayores que un bloque dividiéndolos en secuencias de **registros** más pequeños.
  - La **clave de búsqueda** será el número de registro, que puede ser **secuencial** o por el **desplazamiento** que tengan en el objeto.

Redistribución entre múltiples hermanos

- En los árboles B+ tradicionales, la **inserción** o **borrado** puede provocar **divisiones** y **combinaciones** de nodos adyacentes.
  Cuando se usan organizaciones de archivos con árboles B+ es muy importante la **utilización del espacio**, ya que hay menos entradas en las hojas.
  - Se puede mejorar la utilización del espacio implicando **varios nodos** hermanos en la **redistribución** por inserción o borrado.
    - ✓ Se maximiza la **ocupación media** de los nodos.
    - ✓ Se evitan **divisiones innecesarias**.

Índices secundarios y reubicación de registros

- ÍNDICE PRIMARIO (o ÍNDICE DE AGRUPACIÓN) ⟶ índice que permite leer todos los registros de un archivo en orden físico en el disco.
- ÍNDICE SECUNDARIO ⟶ índices que no permiten esto.

- Cuando un registro se **mueve entre hojas** (p. e., tras una división) hay que actualizar todos los **índices secundarios** que almacenan punteros a él.

- Cada hoja puede contener muchos registros y cada uno de ellos puede estar en diferentes ubicaciones de cada índice secundario, por lo que la reorganización de una hoja puede exigir **muchos accesos a disco** para actualizar todos los índices secundarios afectados.
  - Para evitar esto, en los índices secundarios, en lugar de punteros a los registros, se almacenan los **valores de la clave de búsqueda** del índice primario.
  - ✓ Reduce en gran medida el **coste de actualización** de los índices debido a la reorganización de los archivos.
  - × Incrementa el **coste de acceso** a los datos mediante un índice secundario, pues ahora localizar un registro exige dos pasos:
    1. Usar el índice secundario para buscar los valores de la clave de búsqueda del índice primario.
    2. Usar el índice primario para buscar los registros correspondientes.

ÍNDICES SOBRE CADENAS DE CARACTERES

- La creación de índices de árboles B+ sobre atributos de tipo cadena de caracteres plantea dos problemas:
  - Las cadenas pueden ser de **longitud variable** ⟶ el número de entradas de cada nodo no tiene por qué ser el mismo, luego el *fan-out* es variable.
  - Las cadenas pueden ser **largas** ⟶ grado de salida bajo, por lo que la altura del árbol es mayor (y se necesitan más accesos para recorrerlo).
- Esto se puede solucionar con la COMPRESIÓN DEL PREFIJO, en la que no se almacena la clave de búsqueda completa en los nodos internos, sólo se almacena el **prefijo suficiente** para distinguir las claves de los subárboles bajo ella.

CARGA MASIVA

- Al construir un árbol B+ de una **relación más grande que la MP** y un **índice sin agrupamiento** sobre ella también **más grande que la MP**: Según se recorre la relación, seguramente los nodos hoja a los que se accede no estén en la MP (ya que no están ordenados) por lo que cada vez que se añade una entrada a una hoja se necesita una lectura en disco.
- Esto se puede solucionar con la técnica de CONSTRUCCIÓN POR CARGA MASIVA (*bulk loading*) DESDE ABAJO, que consiste en:
  1. Crear un **archivo temporal** con todos los registros **ordenados** por clave.
  2. **Trocear** las entradas en bloques, que formarán el nivel de hoja del árbol.
  3. Para crear las entradas del siguiente nivel del árbol (hasta llegar a la raíz) se usa el valor mínimo de cada bloque y un puntero a él.
  - ✓ Así, todas las entradas que van a un nodo hoja concreto aparecen seguidas, por lo que sólo hará falta **una operación de lectura por nodo,** aunque se inserten muchas entradas en él.

ÁRBOLES B+ EN MEMORIAS FLASH

- Las memorias flash tienen un comportamiento ligeramente diferente al disco:
  - Las **escrituras** son más **lentas** ⟶ no permiten actualizaciones en el mismo lugar de los datos, por lo que cada una se convierte en una copia + escritura de un bloque, lo que requiere que la copia antigua se tenga que borrar.
  - El **borrado** se realiza por **bloques**.
  - La **lectura** es muy **rápida**.
  - Los **bloques** suelen ser más **pequeños**.
- Para usarlas, se realizan **adaptaciones** sobre los árboles B+ como:
  - **Reducir** el **tamaño de los nodos** para adaptarse al tamaño de los bloques.
  - **Minimizar** el número de **escrituras y borrados**, aprovechando el acceso rápido aleatorio para realizar **muchas lecturas**.

COMPARACIÓN ÁRBOL B Y B+

- En los ÁRBOLES B+ solo las **hojas** contienen los datos completos, los nodos internos se usan para navegación.
- En los ÁRBOLES B tanto los **nodos internos** como las **hojas** contienen claves de búsqueda y punteros a registros.
  - ✓ Los valores de la clave aparecen una sola vez en todo el árbol (si son únicos) por lo que se **evita** la **pérdida de espacio por redundancia** de claves de los B+
  - × Es necesario incluir un campo adicional para un **puntero** por cada clave de búsqueda de un **nodo interno**, por lo que se **reduce su número de entradas**.
    - × Como hay menos entradas en los nodos internos, el árbol tiene un **grado de salida menor** y por tanto puede tener **más profundidad**.
  - ✓ Una **búsqueda** podría terminar **antes** de llegar a una hoja.
  - × Puede ser que en cada nodo interno se tengan que comprobar **todas las claves** que contiene, y, como la **altura es mayor**, la probabilidad de encontrar la clave buscada pronto no es mucho más alta.
    - ↳ Entonces las búsquedas en un árbol B son más **rápidas** para **algunas** claves y más **lentas** para otras.
  - × El **borrado** es más **complicado** porque puede realizarse en un nodo interno.
- Por tanto, el **árbol B+ es el estándar** habitual en sgbd y muchas veces el término árbol B se usa para referirse a él.

ACCESO CON VARIAS CLAVES

- Ahora trataremos la optimización de consultas que involucran **más de un atributo de búsqueda**.

▷ $SELECT * FROM\ empleados\ WHERE\ departamento =' Ventas'\ AND\ salario > 20000$

VARIOS ÍNDICES DE CLAVE ÚNICA

- Cuando existen **índices separados para cada atributo de búsqueda**, se pueden usar de forma combinada para procesar este tipo de consultas.
  1. Obtener de cada índice la lista de identificadores de registro (RIDs) que cumplen con una de las condiciones.
  2. Hacer la intersección de las listas para hallar los registros que satisfacen todas las condiciones.

× Si alguna condición es **poco selectiva** (devuelve una lista muy grande) el método es lento y pesado.
× Si la **intersección** de las listas resulta en **pocos** registros, no es eficiente.
↳ En estos casos es mejor usar índices de mapas de bits (*bitmaps*).

✓ **Alta flexibilidad** → se pueden combinar índices dinámicamente sin haberlos diseñado previamente para consultas específicas.

ÍNDICES SOBRE VARIAS CLAVES

○ Una alternativa más eficiente son los índices con CLAVE DE BÚSQUEDA COMPUESTA, que está formada por la **concatenación** de los valores de los atributos involucrados en el predicado.

✓ **Se puede usar** y **es eficiente** para consultas del tipo:
- Igualdad sobre los dos atributos que forman la clave → $WHERE\ departamento = 'Ventas'\ AND\ salario = 20000$
- Igualdad sobre el primero y comparación sobre el segundo → $WHERE\ departamento = 'Ventas'\ AND\ salario > 20000$
- Igualdad sobre el primero y rango sobre el segundo → $WHERE\ departamento = 'Ventas'\ AND\ salario\ BETWEEN\ 1\ AND\ 30$
- Igualdad sólo sobre el primero → $WHERE\ departamento = 'Ventas'$
  - La condición es equivalente a una **consulta por intervalos** sobre el intervalo: $(\ ('Ventas', -\infty), ('Ventas', \infty)\ )$.

× **Se puede usar,** pero **no es eficiente** para consultas del tipo:
- Comparación sobre el primer atributo → $WHERE\ departamento >' Ventas'\ AND\ salario = 20000$
  ↳ Para cada valor de $departamento$ que vaya alfabéticamente antes que $'Ventas'$ hay que localizar los registros con un salario de 20000.
  Debido a la ordenación de los registros en el archivo, probablemente cada uno esté en un bloque de disco diferente.
  - La condición **no** es equivalente a una **consulta por intervalos**.

× **No se puede usar** en consultas de tipo:
- La condición no involucra el primer atributo.

↳ En estos casos es mejor usar índices de mapas de bits (*bitmaps*) o árboles R.

ÍNDICES DE COBERTURA

○ Un ÍNDICE DE COBERTURA es aquel que, además de la clave de búsqueda, contienen en sus entradas algunos de los **atributos requeridos por la consulta**.
- Esto significa que **no es necesario acceder al archivo de datos** para recuperar el resultado de la consulta.

▷ P. e.: un índice sobre $(departamento, salario, nombre)$ permitiría recuperar todos los datos desde el índice sin tener que acceder a los registros.

✓ **Reducción** significativa del número de **accesos a disco**, ya que los datos están contenidos en el índice.
✓ En entornos con **consultas repetitivas** sobre pocas columnas, el **rendimiento** puede mejorar dramáticamente.
- Se obtendría el mismo resultado usando un índice sobre una **clave de búsqueda compuesta**,
  pero estos índices reducen el tamaño de las claves de búsqueda, lo que permite un mayor *fan-out* en los nodos internos, **reduciendo la altura del árbol**.

ASOCIACIÓN ESTÁTICA

ESTRUCTURA

○ Un CAJÓN es una unidad de almacenamiento de tamaño fijo que puede guardar uno o más registros.
↳ Normalmente se corresponde con un **bloque** de disco, pero podría tener un tamaño mayor o menor.

Sea K el conjunto de todos los valores de clave de búsqueda y B el de todas las direcciones de cajón.

○ Una FUNCIÓN DE ASOCIACIÓN (función hash) h es una función $h: K \rightarrow B$ que calcula en función de un valor concreto de clave de búsqueda la dirección de uno de los cajones donde puede estar un registro con esa clave.

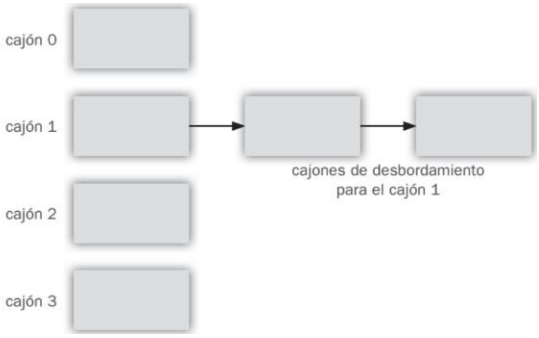
FUNCIONES DE ASOCIACIÓN

- Como durante la etapa de diseño **no se conocen los valores** de la clave de búsqueda que se almacenarán, lo deseable es elegir una función hash que tenga:
  - **Uniformidad** → asigna a cada cajón el mismo número de valores de la clave de búsqueda dentro del conjunto de todos los valores posibles, evitando concentraciones (*skew*).
  - **Aleatoriedad aparente** → el resultado de la función no está relacionado con ningún ordenamiento de los valores de la clave de búsqueda visible exteriormente.
  - **Determinismo** → siempre devuelve el mismo resultado para la misma clave.

- Ejemplos de funciones hash sobre **cadenas de caracteres**:
  - **Módulo** ⟶ se suman las representaciones binarias de los caracteres de la clave y se calcula el módulo del resultado.
    $$sum(k)\ \%\ B$$
  - **Multiplicación por una constante** ⟶ partiendo de que la clave es una cadena de n caracteres de longitud y que c es una constante.
    $$s[0] * c^{n-1} + s[1] * c^{n-2} + \cdots + s[n-1]\ \%\ B.$$
  - **Hash criptográfico** (MD5, CRC) ⟶ cuando se requiere de **alta dispersión**.

GESTIÓN DE DESBORDAMIENTO DE CAJONES

- Si un cajón se queda sin espacio, se produce DESBORDAMIENTO DE CAJONES.
  Esto puede suceder por varias razones:
  - **Cajones insuficientes** ⟶ se debe elegir un número de cajones que sea $> núm\ registros/tamaño\ cajón\ (en\ registros)$
    - Para reducir la probabilidad de desbordamiento se puede escoger un número de cajones $(núm\ registros/tamaño\ cajón) * (1 + d)$ donde d es un **factor de corrección** (normalmente cercano a 0.2)
      - × Se **desperdicia** algo de espacio al final de los cajones.
  - **Atasco** ⟶ algunos cajones tienen asignados más registros que otros, por lo que uno puede desbordar, aunque haya otros con espacio libre.
    Esto puede pasar si:
    - Muchos registros tienen la **misma clave de búsqueda**.
    - La función hash genera una **distribución no uniforme** de las claves de búsqueda.



- El desbordamiento de cajones se trata mediante CADENAS DE CAJONES DE DESBORDAMIENTO:
  1. Cada cajón tiene un puntero a una lista enlazada de cajones de desbordamiento.
  2. Si el cajón se llena, los registros adicionales se almacenarán en uno de estos cajones.

- **Buscar** un registro de clave de búsqueda k:
  1. Calcular $h(k)$ y acceder al cajón correspondiente.
  2. Recorrer este cajón.
  3. Si no se encuentra el registro, continuar en el siguiente cajón de desbordamiento.
  4. Repetir paso 4 hasta que se acabe la cadena de cajones de desbordamiento o se encuentre el registro.

- **Insertar** un nuevo registro de clave de búsqueda k.
  1. Calcular $h(k)$ y acceder al cajón correspondiente.
  2. - Si hay espacio libre ⟶ insertar directamente.
     - Si está lleno ⟶ comprobar los cajones de desbordamiento.
       3. Si se encuentra alguno con espacio disponible ⟶ insertar en él.
       4. Si se encuentra alguno con espacio disponible ⟶ crear un nuevo cajón de desbordamiento e insertar en él.

- **Borrar** un registro de clave de búsqueda k:
  1. Seguir el procedimiento de búsqueda descrito antes.
  2. Eliminar el registro y dejar el hueco, se aprovechará para futuras inserciones.
     - No se eliminan los **cajones de desbordamiento vacíos** inmediatamente (a no ser que se implemente alguna estrategia de limpieza).

- Esta estructura se denomina ASOCIACIÓN CERRADA.
  ↳ Es preferible para los sgbd.
- En la ASOCIACIÓN ABIERTA no hay cadenas de desbordamiento, cuando un registro no cabe en su bloque se inserta en otro cajón (p. e. el siguiente).

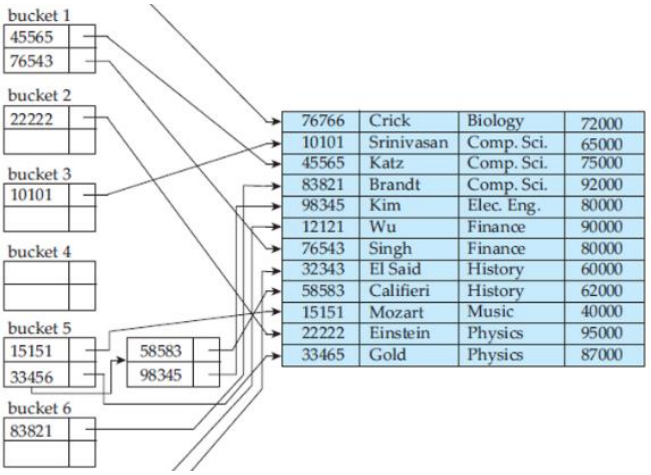
VENTAJAS Y LIMITACIONES

✓ Alta eficiencia en **búsquedas por igualdad** ⟶ en la mayoría de casos, se accede a un único bloque.
✓ **Fácil implementación** ⟶ estructura simple, ideal para sistemas con recursos limitados.
✓ **Consistencia** ⟶ el acceso directo evita la necesidad de recorrer estructuras más complejas como los árboles.

× No permite **búsquedas por rango** ⟶ como no hay orden entre los cajones, no se puede hacer $BETWEEN, >, <$, etc.
× **Escasa flexibilidad al crecer** el archivo ⟶ si el número de registros crece significativamente, los cajones se llenan y el número de colisiones aumenta.
× **Difícil reorganización** ⟶ una vez fijado el número de cajones, cambiarlo implica rehacer completamente la estructura.

ÍNDICES ASOCIATIVOS

- Un ÍNDICE HASH usa una función hash para convertir una clave de búsqueda en la dirección de cajón donde se encuentra el registro o su clave y un puntero a él.



HASHING DINÁMICO

- La necesidad de **fijar el conjunto B** presenta un problema con la técnica de asociación estática, pues la mayor parte de las bd **aumentan de tamaño** con el tiempo.
  Si se va a usar asociación estática, hay 3 opciones:
  - Elegir una **función hash** de acuerdo con el **tamaño actual** del archivo → se degradará el rendimiento a medida que la bd crezca en tamaño.
  - Elegir una **función hash** de acuerdo con el **tamaño previsto** del archivo en un momento del futuro → evita la degradación de rendimiento, pero puede que inicialmente se desperdicie mucho espacio.
  - **Reorganizar periódicamente** la estructura asociativa en respuesta al crecimiento del archivo → es una operación masiva que consume mucho tiempo y durante la cual no se puede acceder al archivo.

- La ASOCIACIÓN DINÁMICA permite **modificar dinámicamente la función hash** para adaptarse a los cambios de tamaño de la bd.
- La ASOCIACIÓN EXTENSIBLE mantiene la eficacia del **acceso directo** de la asociación tradicional, añadiendo una **estructura dinámica** que se ajusta automáticamente al crecimiento del archivo sin necesitar una reestructuración global del índice.

ESTRUCTURA

- El hashing extensible se basa en una **función hash** h que produce una cadena de b bits (por ejemplo, 32) y en una **tabla de direcciones** de tamaño 2^i.
- El valor de i $(1 \leq i \leq b)$ aumenta y disminuye con el tamaño de la bd, de manera que en un momento dado indica que se requieren $\boldsymbol{i}$ **bits de** $\boldsymbol{h(K)}$ para determinar el **cajón apropiado para** $\boldsymbol{K}$.
- Varias entradas consecutivas de la tabla pueden apuntar al mismo cajón, estas tendrán un **prefijo de asociación común**.
  Se asocia con cada cajón j un entero i_j que proporciona la longitud del prefijo de asociación común.
  - El número de entradas de la tabla de direcciones que apuntan al cajón j es 2^{i-i_j}

FUNCIONAMIENTO

- **Buscar** un registro de clave de búsqueda k:
  1. Tomar los i MSB de $h(k)$.
  2. Buscar la entrada de la tabla correspondiente a esa cadena de bits.
  3. Seguir el puntero al cajón de esa entrada.

- **Insertar** un registro de clave de búsqueda k:
  1. Seguir el procedimiento de búsqueda, que finaliza en el cajón j.
  2. - Si hay sitio en el cajón j → insertar en él.
     - Si el cajón j está lleno:

  – Si $i = i_j$ → sólo una entrada de la tabla apunta al cajón j, así que para dividir el cajón j hay que **aumentar el tamaño de la tabla**
  1. Incrementar en 1 el valor de i, duplicando el tamaño de la tabla.
  2. Sustituir cada entrada por dos entradas, ambas con el mismo puntero que la original.
  3. Asignar un nuevo cajón, el cajón z.
  4. Hacer que la segunda entrada que apunta a j apunte a z.
  5. Definir i_j e i_z como i.
  6. Volver a calcular la función hash para todos los registros del cajón j y asignarlos o bien al j o al z.
  7. Volver a intentar la inserción, que seguramente tenga éxito.

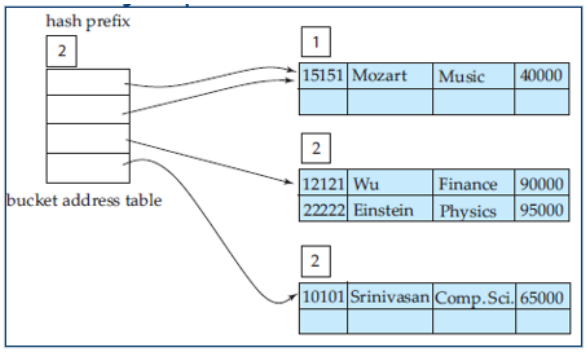
  - Si todos los registros del cajón j y el registro nuevo tienen el mismo prefijo de asociación → hay que **volver a dividir** el cajón.
    ↳ Si la función hash se eligió cuidadosamente, es **poco probable** que suceda esto.
  - Si todos los registros del cajón j tienen el mismo valor para la clave de búsqueda → la división no serviría para nada, así que se **usan cajones de desbordamiento** como en la asociación estática.

  – Si $i > i_j$ → más de una entrada de la tabla apunta al cajón j, por lo que se puede dividir el cajón **sin aumentar el tamaño de la tabla**.
  1. Asignar un nuevo cajón, el cajón z.
  2. Ajustar las entradas de la tabla que antes apuntaban al cajón j: la primera mitad se deja como estaba y las demás apuntan al cajón z.
  3. Definir i_j e i_z como $i_j + 1$.
  4. Volver a calcular la función hash para todos los registros del cajón j y asignarlos o bien al j o al z.
  5. Volver a intentar la inserción, que seguramente tenga éxito.

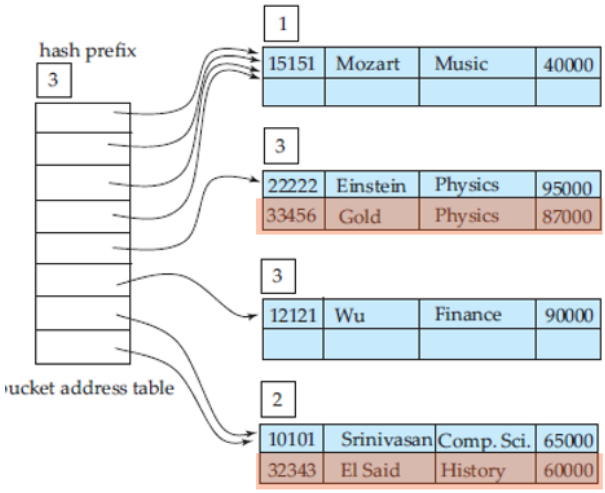
  - En el improbable caso de que vuelva a fallar, se aplica el caso $i = i_j$ o $i > i_j$ que corresponda.

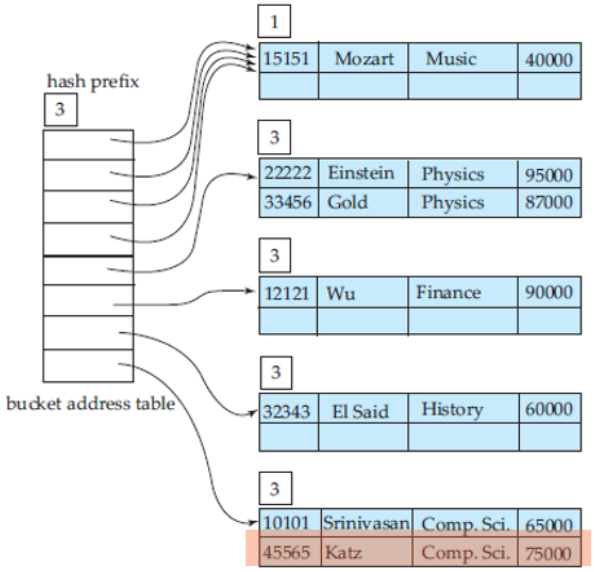
- **Borrar** un registro de clave de búsqueda k:
  1. Seguir el procedimiento de búsqueda, que finaliza en el cajón j.
  2. Borrar tanto el registro del archivo como la clave de búsqueda del cajón.

  - Si el cajón queda **vacío**, se elimina.
    - Se pueden **fusionar** varios **cajones** y **reducir** el tamaño de la **tabla** a la mitad.
      × Reducir el tamaño de la tabla es una operación muy **costosa** si la tabla es grande, por lo que solo merece la pena si el número de cajones se reduce considerablemente.



Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001





VENTAJAS Y LIMITACIONES

- ✓ **Escalabilidad incremental** ⟶ la estructura se adapta dinámicamente al crecimiento del archivo.
- ✓ **Espacio controlado** ⟶ la tabla de direcciones provoca una sobrecarga pequeña porque solo contienen un puntero por cada resultado hash para el i actual.
- ✓ **Reorganización localizada** ⟶ sólo afecta al cajón implicado, no toda la estructura.
- ✓ **Acceso directo eficiente** ⟶ la búsqueda sigue siendo $O(1)$ la mayoría de los casos.
- × No permite **búsquedas por rango** ⟶ como no hay orden entre los cajones, no se pueden hacer $BETWEEN, >, <$, etc.
- × **Tabla de direcciones adicional** ⟶ introduce una capa de indirección para acceder a la tabla, lo que puede requerir un acceso adicional a memoria o disco.
- × **Sobrecarga de metadatos** ⟶ cada cajón necesita almacenar su profundidad local.
- × **Posible fragmentación** ⟶ algunos cajones pueden quedar muy poco utilizados.

COMPARACIÓN ENTRE INDEXACIÓN ORDENADA Y HASHING

Muchos sgbd proporcionan **árboles B+** y muchos otros incorporan también **estructuras asociativas**.

- Para decidir cuál se debe usar, es muy importante analizar los **tipos de consultas** que se van a realizar:
  - Si la mayoría de las consultas son de la forma $clave = valor$ ⟶ es preferible un esquema **asociativo**.
    - Tiempo **medio** para búsquedas:
      - Al usar un índice ordenado ⟶ $O(\log(número\ de\ valores\ de\ clave\ en\ la\ relación))$.
      - Al usar una estructura asociativa ⟶ $O(1)$.
    - × Tiempo en el **peor caso** para búsquedas:
      - Al usar un índice ordenado ⟶ $O(\log(número\ de\ valores\ de\ clave\ en\ la\ relación))$.
      - Al usar una estructura asociativa ⟶ $O(número\ de\ valores\ de\ clave\ en\ la\ relación)$.
        ↳ Sin embargo, el peor de los casos es **poco probable**.
  - Si la mayoría de consultas son de la forma $clave \leq valor_2\ and\ clave \geq valor_1$ ⟶ es preferible un **índice ordenado**.
    - Al usar un **índice ordenado**:
      1. Se realiza una búsqueda del $valor_1$.
      2. Se sigue el orden de la cadena de punteros del índice para leer el siguiente cajón.
      3. Se continúa de esta manera hasta llegar a $valor_2$.
    - Al usar una **estructura asociativa** ⟶ se puede llevar a cabo una búsqueda de $valor_1$ y localizar el cajón correspondiente, pero no resulta fácil determinar en qué cajón hay que mirar a continuación, pues la función hash asigna valores a los cajones aleatoriamente.
      ↳ Entonces, habría que leer **todos los cajones** para encontrar las claves de búsqueda necesarias.

✓ Las **estructuras asociativas** son muy útiles para los **archivos temporales** creados durante el procesamiento de las **consultas**.