Capítulo 2

Algoritmos y Estructuras de Datos para Memoria Secundaria

Modelo de Computación Externa

- En los algoritmos que discutimos en el capítulo anterior
  - Se supuso que los datos caben completos en la memoria principal
  - ¿Qué sucede si los datos de entrada en un problema no caben en la memoria principal?
    - La mayoría de los computadores disponen de dispositivos externos de almacenamiento
      - Discos
      - Dispositivos de almacenamiento masivo
      - Se les conoce como memoria secundaria

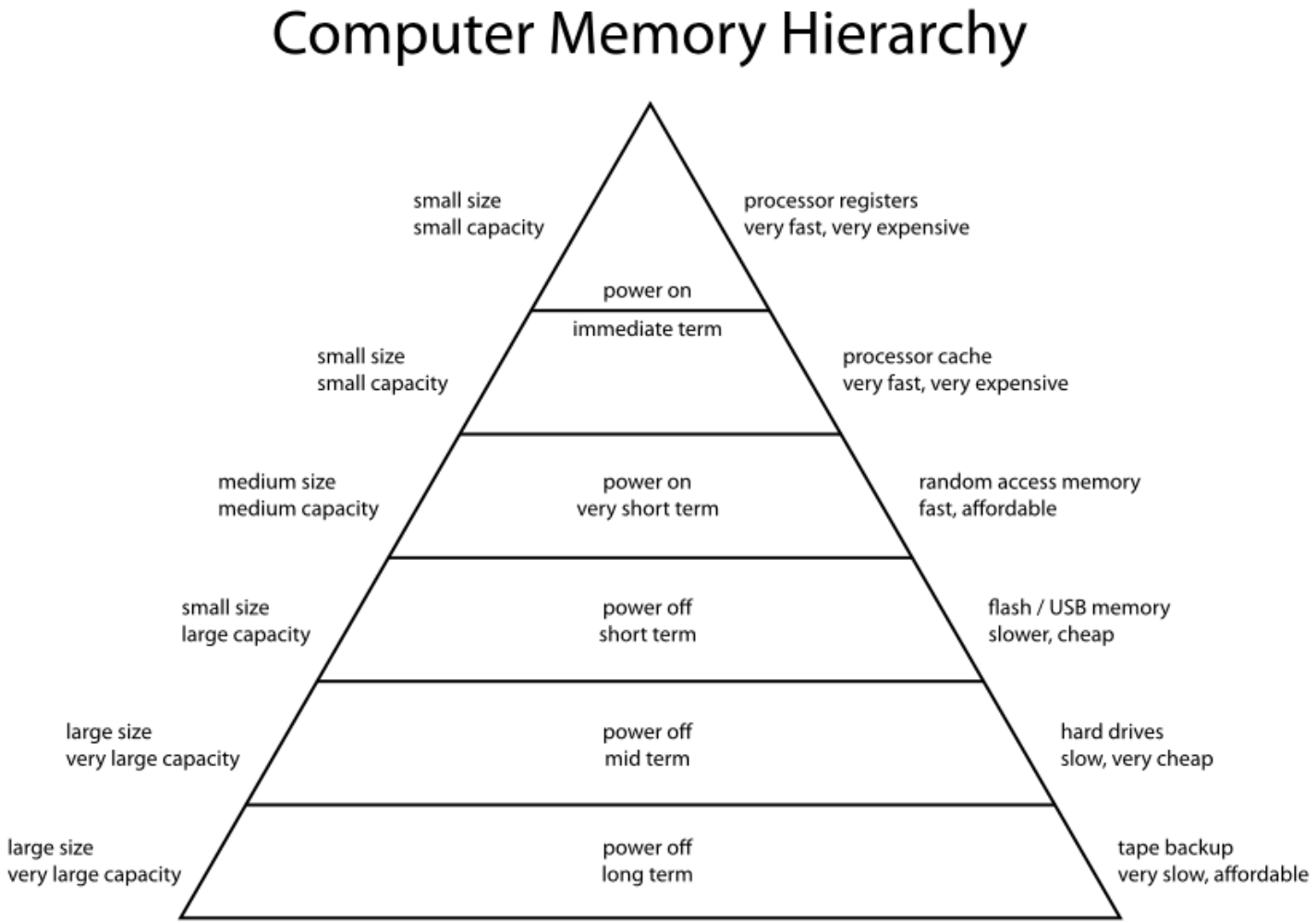
Modelo de Computación Externa

- Estos dispositivos de almacenamiento tienen características de acceso distintas a la memoria principal
  - Existen estructuras de datos y algoritmos para utilizar estos dispositivos más eficazmente
  - En este capítulo se estudiarán algoritmos en memoria secundaria para implementar:
    - Ordenación
    - Colas de prioridad
    - Diccionarios

Modelo de Computación Externa

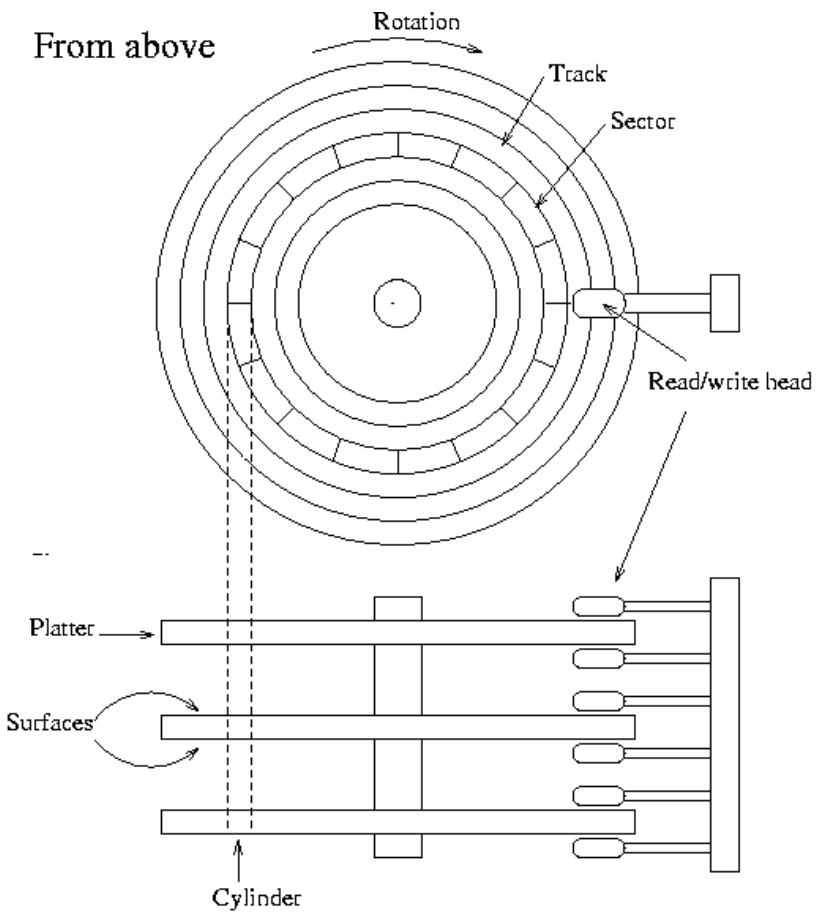
- La arquitectura de memoria del computador tiene muchos niveles
  - Registros
  - Cache L1/L2/L3
  - Memoria RAM
  - Disco duro magnético / Disco de estado sólido
  - Almacenamiento en la red
  - Cinta magnética
  - CD y DVDs también son “memoria” (terciaria)

Modelo de Computación Externa



Modelo de Computación Externa

- Memoria secundaria
  - Esquema disco duro
    - Sector = bloque



Modelo de Computación Externa

- Memoria secundaria
  - Dividida en bloques de igual tamaño
  - Tamaño del bloque varía dependiendo de varios factores (físicos, sistema operativo, sistema de archivos, etc.)
    - Tamaños usuales: 512 bytes, 4096 bytes
  - Archivo: “lista enlazada” de bloques
    - Puede estar implementado como un árbol
      - Hojas almacenan la información
      - Nodos interiores referencian otros nodos

Modelo de Computación Externa

- Memoria secundaria
  - Por ejemplo
    - Suponer que 4 bytes son suficientes para almacenar la dirección de un bloque
    - Los bloques son de 4096 bytes
    - Bloque raíz puede almacenar 1024 direcciones
      - Archivos de hasta 1024 bloques (4 megabytes) se pueden representar como un bloque raíz y bloques hojas que contienen los datos del archivo
      - Archivos de 2^20 bloques (2^32 bytes) se pueden representar con un árbol de altura 2

Modelo de Computación Externa

- Memoria secundaria
  - La operación básica en un archivo es leer un bloque a un buffer en memoria principal
    - Buffer es un área reservada en memoria principal cuyo tamaño es igual a la de un bloque de memoria secundaria
    - Típicamente, el sistema operativo facilita la lectura de los bloques en el orden en que aparecen en su “lista enlazada”

Modelo de Computación Externa

- Memoria secundaria
  - La operación de escritura en un archivo es similar
    - Los datos se escriben en el buffer en memoria principal en forma secuencial
    - Cuando se llena el buffer, se escribe el bloque correspondiente en algún bloque disponible en la memoria secundaria, y se agrega dicho bloque al final de la “lista enlazada” del archivo

Modelo de Computación Externa

- Medida de costo para operaciones en memoria secundaria
  - Supuesto: El tiempo de encontrar un bloque en memoria secundaria y copiarlo a la memoria principal es costoso comparado con procesar los datos en dicho bloque
    - Nota: Es importante verificar este supuesto cuando uno desea analizar algoritmos que manejan datos en memoria secundaria

Modelo de Computación Externa

- Medida de costo para operaciones en memoria secundaria
  - Ejemplo:
    - Bloque de 1.000 enteros, disco rota a 1.000 RPM
    - Tiempo requerido para copiar el bloque en memoria
      - Posicionar el cabezal del disco hasta la pista que contiene el bloque (seek time)
      - Tiempo de espera que el bloque se posicione debajo del cabezal (rotational latency time)
      - En total: promedio de 100 milisegundos (escritura =)
      - ¿Tiempo para leer un valor de la memoria principal?

Modelo de Computación Externa

- Medida de costo para operaciones en memoria secundaria
  - Ejemplo:
    - El computador en el mismo tiempo puede realizar del orden de millones de instrucciones
    - Esto es más que suficiente para procesar los 1.000 números enteros cuando estén en memoria principal
      - Encontrar mínimo/máximo
      - Incluso ordenarlos con un algoritmo eficiente

Modelo de Computación Externa

- Medida de costo para operaciones en memoria secundaria
  - Por lo tanto, para evaluar el tiempo de ejecución de un algoritmo que opera con datos almacenados en memoria secundaria, es muy importante considerar el número de veces que se lee un bloque y se copia a memoria principal
  - Llamaremos a esta operación un *acceso a disco*

Modelo de Computación Externa

- Medida de costo para operaciones en memoria secundaria
  - Otro supuesto:
    - Los bloques son de tamaño fijo
      - No se puede “simular” el hacer un algoritmo más rápido aumentando el tamaño del bloque, lo que implica disminuir el número de accesos a disco
    - Como consecuencia, la medida de costo para algoritmos en memoria secundaria será el número de accesos a disco

Ordenamiento Externo

- Ordenar datos almacenados en archivos se conoce como ordenamiento externo
- Se estudiará cómo un algoritmo basado en mezclar archivos (“merge sorting”) puede ordenar un archivo con n registros en O(log n) pasadas sobre el archivo

Ordenamiento Externo

- Merge sorting
  - Idea esencial: organizar el archivo en “runs” cada vez mayores
    - Un run es una secuencia de registros $r_1, \ldots, r_k$ ordenada
  - Se dice que un archivo de registros $r_1, \ldots, r_m$ está organizado en runs de largo k si se cumple que

$$\forall i \geq 0 \text{ tal que } ki \leq m,\ r_{k(i-1)+1}, r_{k(i-1)+2}, \ldots, r_{ki} \text{ es un run de largo } k$$

Ordenamiento Externo

- Merge sorting
  - Adicionalmente, si m no es divisible por k, y m = pk+q, con q<k, la secuencia de registros r_{m-q+1}, r_{m-q+2}, …, r_m, llamada tail (cola), es un run de largo q

$$\forall i \geq 0 \text{ tal que } ki \leq m,\ r_{k(i-1)+1}, r_{k(i-1)+2}, \ldots, r_{ki} \text{ es un run de largo } k$$

Ordenamiento Externo

- Merge sorting
  - Ejemplo de un una secuencia de enteros organizada en runs de largo 3 (y con tail)

7	15	29	8	11	13	16	22	31	5	12

Ordenamiento Externo

- Merge sorting
  - El paso básico de merge sort en archivos es comenzar con dos archivos f_1 y f_2, organizados en runs de largo k
  - Suponer que:
    - 1) El número de runs, incluyendo tails, en f_1 y f_2 difieren a lo más en uno
    - 2) A lo más uno entre f_1 y f_2 tiene tail
    - 3) El archivo que tiene tail tiene al menos tantos runs como el otro archivo

Ordenamiento Externo

- Merge sorting
  - Proceso:
    - Leer un run de cada archivo f_1 y f_2
    - Mezclar los runs
    - Anexar el run resultante de largo 2k en alguno de los archivos g_1 y g_2
    - Al alternar entre g_1 y g_2, los archivos resultantes no sólo estarán organizados en runs de largo 2k, sino que además cumplirán con los supuestos 1), 2) y 3)
      - 2) y 3) se cumplen si el tail se mezcla con (o es) el último run creado

Ordenamiento Externo

- Merge sorting
  - El algoritmo comienza dividiendo los n registros en dos archivos f_1 y f_2, lo más equitativamente posible
  - Todo archivo tiene la propiedad de estar organizado en runs de largo 1
  - Se comienza mezclando estos runs de largo 1 y distribuyéndolos en g_1 y g_2, organizados en runs de largo 2
    - A esto le denominaremos una pasada

Ordenamiento Externo

- Merge sorting
  - Los archivos quedan vacíos f_1 y f_2, y se mezclan g_1 y g_2 en f_1 y f_2
  - f_1 y f_2 quedan organizados con runs de largo 4
  - Se itera nuevamente para obtener g_1 y g_2 organizados en runs de largo 8
  - Etc.

Ordenamiento Externo

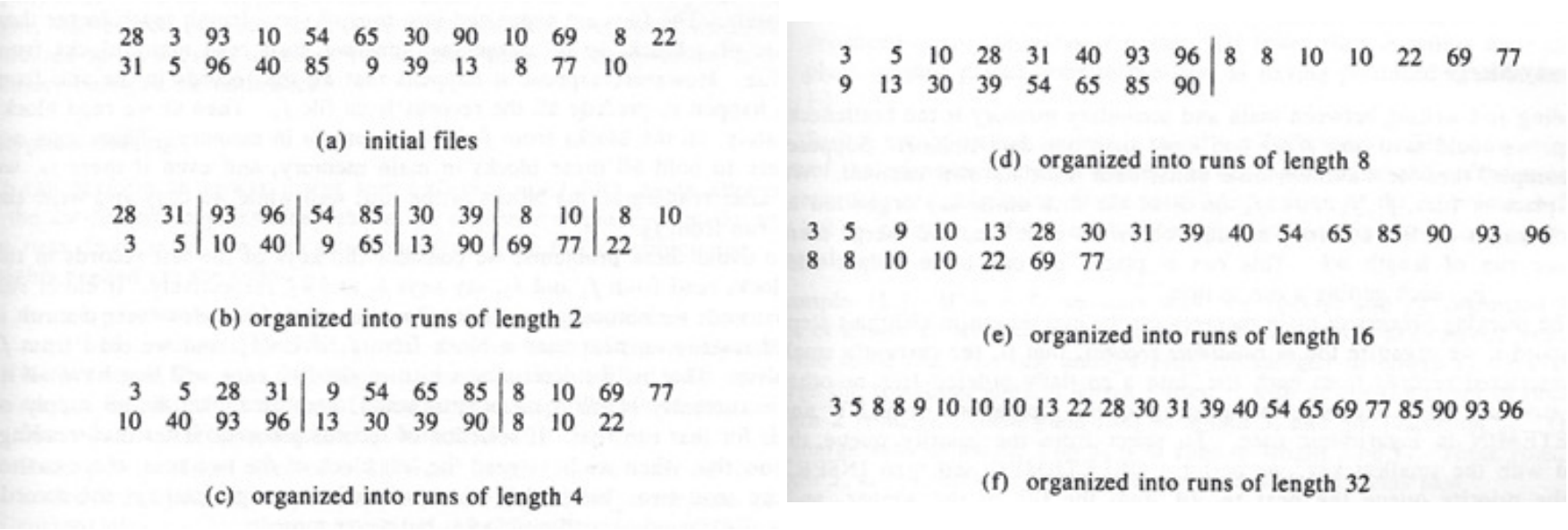
- Merge sorting
  - Luego de i pasadas, se tienen dos archivos organizados en runs de largo 2^i
  - Si 2^i >= n, entonces uno de los dos archivos está vacío y el otro contiene un único run de largo n
    - Es decir, el archivo está ordenado
  - Como 2^i >= n cuando i >= log n, se concluye que se requieren ceil(log n) pasadas

Ordenamiento Externo

- Merge sorting
  - Cada pasada requiere realizar la lectura y la escritura de dos archivos de largo aproximado n/2
  - El número total de bloques que se accesan es de 2n/b, con b el número de registros en un bloque
  - Por lo tanto, el número de bloques leídos y escritos en el proceso completo de ordenamiento es de O((n log n)/b)
    - Cuesta lo mismo que hacer O(log n) pasadas sobre los datos almacenados

Ordenamiento Externo

- Merge sorting



Ordenamiento Externo

- Merge sorting
  - Notar que el proceso de mezcla no requiere almacenar un run completo en memoria principal
    - Lee y escribe un registro a la vez
  - Mejora
    - En vez de comenzar con runs de largo 1, comenzar con grupos de k registros que quepan en memoria principal
    - Ordenar estos grupos en memoria principal con un algoritmo eficiente
    - Organizar inicialmente los archivos con runs de largo k

Ordenamiento Externo

- Multiway Merge
  - El método de Merge sorting requiere de cuatro archivos para ordenar
  - ¿Qué pasa si uno puede disponer de más archivos?
    - Por ejemplo, se tienen varios discos, cada uno de ellos con un canal de comunicación independiente
    - Multiway Merge

Ordenamiento Externo

- Multiway Merge
  - Se tienen 2m discos, cada uno con su propio canal
  - Se guardan archivos f_1, f_2,…, f_m, en m de los discos, organizados en runs de largo k
  - Se leen m runs, uno de cada archivo, y se mezclan en un run de tamaño mk
    - Este run se guarda en uno de los m archivos de salida g_1, g_2,…, g_m, cada uno de ellos guardando un run en turnos
    - A esto se le denomina m-way merge

Ordenamiento Externo

- Multiway Merge
  - El proceso de mezcla en memoria principal se puede realizar en O(log m) pasos por registro
    - Organizando los registros candidatos en una cola de prioridad (min-heap)
  - Si se tienen n registros, y el largo de los runs se multiplica por m en cada pasada
    - Después de i pasadas el run es de largo m^i
    - Si $m^i >= n$, el archivo está ordenado
      - $i = \log_m n$ pasadas

Ordenamiento Externo

- Multiway Merge
  - Como $\log_m n = \log n / \log m$, este algoritmo ahorra un factor log m en lecturas de un registro
  - Incrementar m indefinidamente no sirve
    - Con m muy grande, el tiempo de mezcla en la memoria principal se torna costoso, aumenta con factor log m

Ordenamiento Externo

- Polyphase Sorting
  - Una m-way merge se puede implementar usando sólo m+1 archivos
    - Alternativa a la estrategia Multiway Merge que usa 2m archivos
  - Idea general de Polyphase Sorting
    - Mezclar runs de m archivos en un archivo de salida
      - No es necesario mezcla todos los runs
    - Cada pasada produce archivos de distinto tamaño
      - El largo en una pasada es la suma de los largos de los runs producidos en las m pasadas anteriores

Ordenamiento Externo

- Polyphase Sorting
  - Ejemplo usando tres archivos (m = 2)
    - Se comienza con dos archivos f_1 y f_2 organizados en runs de largo 1
    - Los registros de f_1 y f_2 se mezclan para obtener runs de largo dos en el tercer archivo
      - Se mezclan runs suficientes para vaciar f_1
    - Se mezclan los runs restantes de f_2 con un número igual de runs de largo 2 del archivo f_3
      - El resultado se guarda en f_1
    - Se sigue iterando de esta forma

Ordenamiento Externo

- Polyphase Sorting
  - Ejemplo usando tres archivos (m = 2)
    - El largo de de los runs resulta ser
      - 1, 1, 2, 3, 5, 8, 13, 21, …
    - Números de Fibonacci
  - Para mantener el proceso hasta que el archivo esté ordenado, el número inicial de registros en f_1 y f_2 deben ser dos números consecutivos de Fibonacci
    - Se ocupan “dummy runs” si faltan
    - Esto optimiza el número de pasadas

Ordenamiento Externo

- Polyphase Sorting
  - Ejemplo usando tres archivos (m = 2)

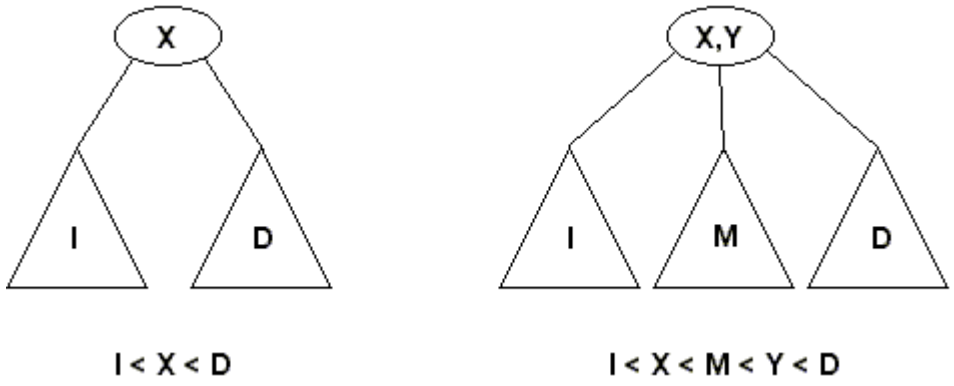
Después de pasada	f_1	f_2	f_3
Inicial	13(1)	21(1)	vacío
1	vacío	8(1)	13(2)
2	8(3)	vacío	5(2)
3	3(3)	5(5)	vacío
4	vacío	2(5)	3(8)
5	2(13)	vacío	1(8)
6	1(13)	1(21)	vacío
7	vacío	vacío	1(34)

Diccionarios en memoria externa

- TDA Diccionario (métodos principales)
  - Buscar(x)
  - Agregar(x,y)
  - Cambiar(x,y)
  - Borrar(x)
- Implementaciones en memoria
  - Arreglos, listas ordenadas, árboles de búsqueda binaria, árboles balanceados, árboles de búsqueda digital, Splay tree, Skip list, hashing…

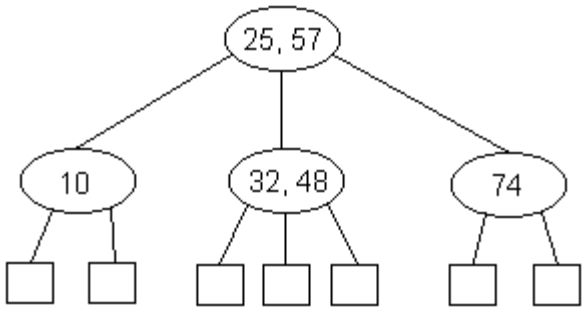
Diccionarios en memoria externa

- Árboles 2-3
  - Los árboles 2-3 son árboles cuyos nodos internos pueden contener hasta 2, y por lo tanto un nodo interno puede tener 2 o 3 hijos
    - Operaciones en tiempo O(log n)



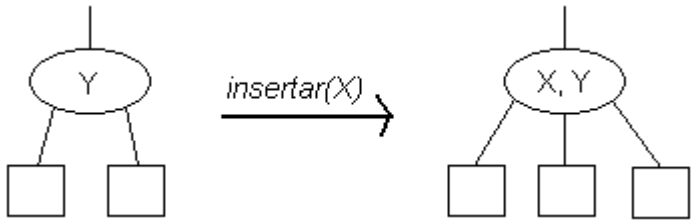
Diccionarios en memoria externa

- Árboles 2-3
  - Una propiedad de los árboles 2-3 es que todas las hojas están a la misma profundidad, es decir, los árboles 2-3 son árboles perfectamente balanceados (altura es log n)



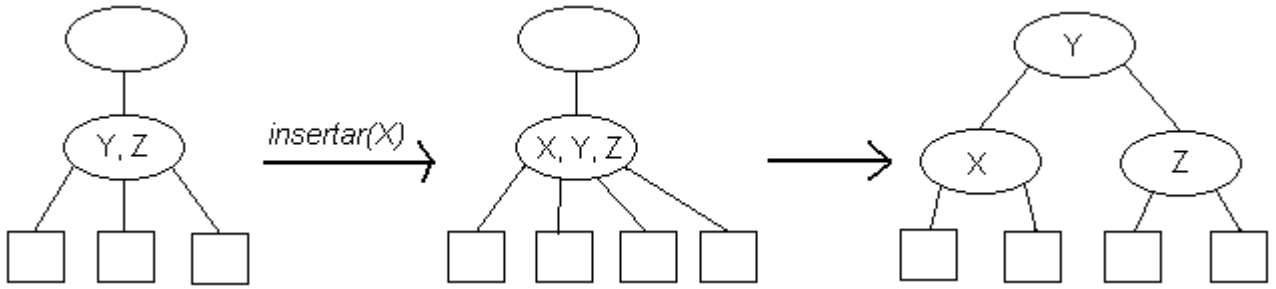
Diccionarios en memoria externa

- Árboles 2-3
  - Inserción: si el nodo donde se inserta X tenía una sola llave (dos hijos), ahora queda con dos llaves



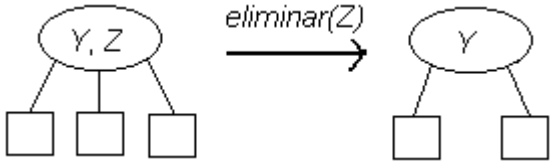
Diccionarios en memoria externa

- Árboles 2-3
  - Inserción: si el nodo donde se inserta X tenía dos llaves (tres hijos), queda transitoriamente con tres llaves (cuatro hijos) y se dice que está saturado (overflow)



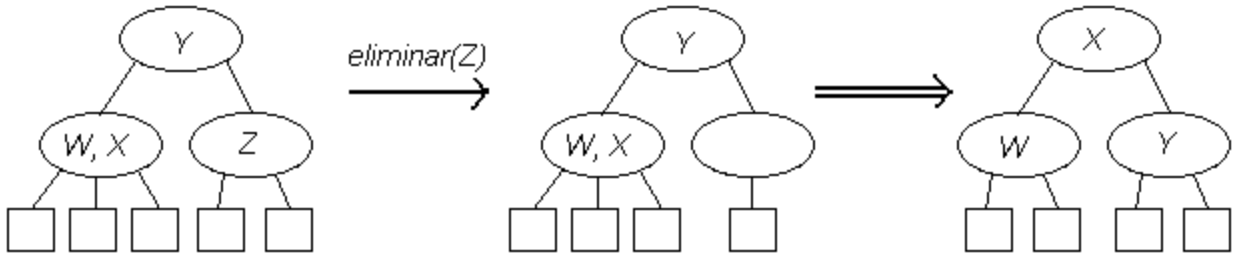
Diccionarios en memoria externa

- Árboles 2-3
  - Eliminación: sin perder generalidad se supondrá que el elemento a eliminar, Z, se encuentra en el nivel más bajo del árbol
    - Caso 1: el nodo donde se encuentra Z contiene dos elementos, en este caso se elimina Z y el nodo queda con un solo elemento



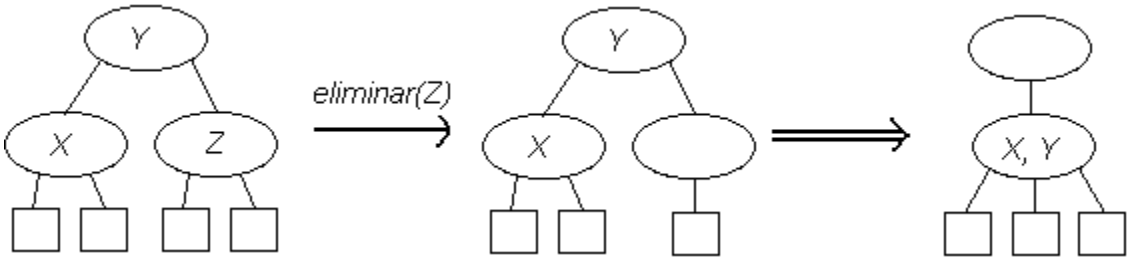
Diccionarios en memoria externa

- Árboles 2-3
  - Eliminación
    - Caso 2: el nodo donde se encuentra Z contiene un solo elemento, en este caso al eliminar el elemento Z el nodo queda sin elementos (underflow)
      - Si el nodo hermano posee dos elementos, se le quita uno y se inserta en el nodo con underflow



Diccionarios en memoria externa

- Árboles 2-3
  - Eliminación
    - Caso 2: el nodo donde se encuentra Z contiene un solo elemento, en este caso al eliminar el elemento Z el nodo queda sin elementos (underflow)
      - Si el nodo hermano contiene solo una llave, se le quita un elemento al padre y se inserta en el nodo con underflow



Diccionarios en memoria externa

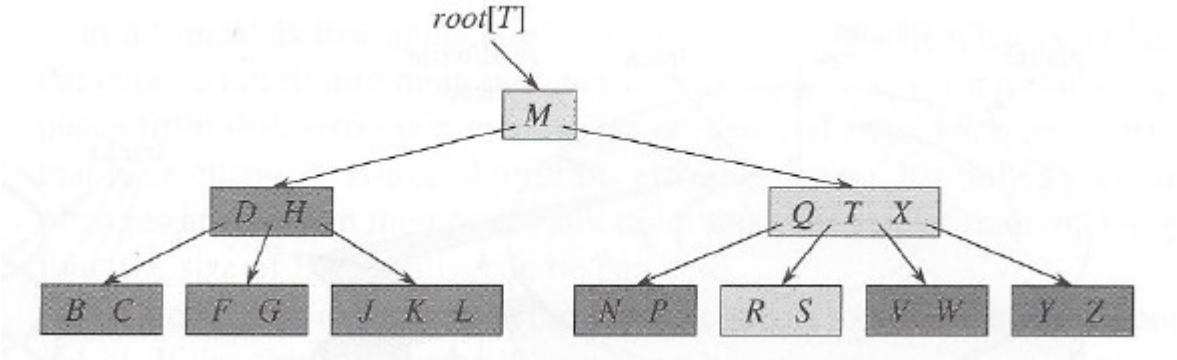
- B-trees
  - Árboles balanceados
  - Diseñados para ser almacenados en memoria secundaria
  - Implementan un diccionario

Diccionarios en memoria externa

- B-trees
  - Todos nodo x contiene
    - Número de llaves n[x] en nodo x
    - Las llaves, almacenadas en orden no decreciente
      - $key_1[X] <= key_2[x] <= \ldots <= key_{n[x]}[x]$
    - Valor booleano leaf[x], que indica si x es una hoja
  - Además contiene n[x]+1 referencias a sus hijos
    - $c_1[X] <= c_2[x] <= \ldots <= c_{n[x]+1}[x]$
  - Las llaves separan los rangos de las llaves almacenadas en cada subárbol

Diccionarios en memoria externa

- B-trees

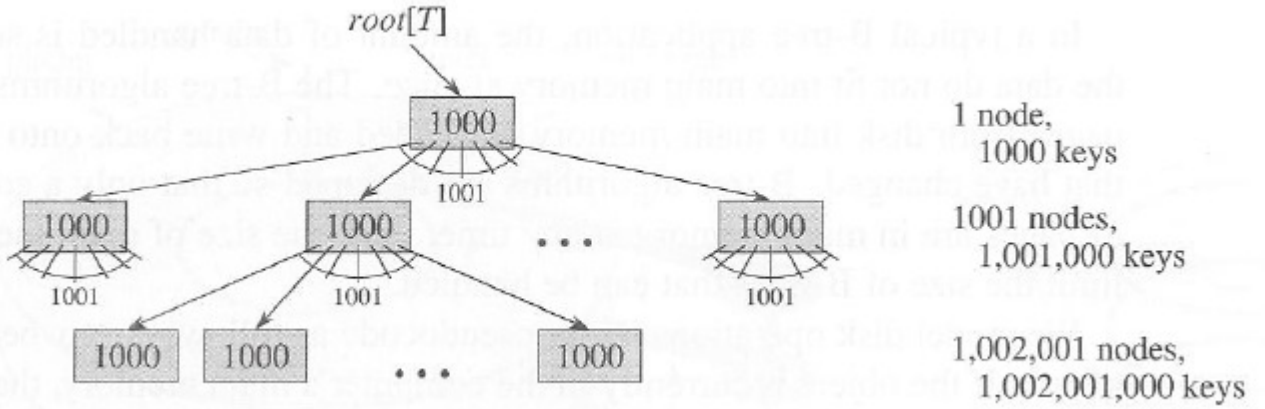


Diccionarios en memoria externa

- B-trees
  - Todas las hojas tienen la misma profundidad
  - Hay una cantidad mínima y máxima de llaves que puede contener un nodo
    - t >= 2, el grado mínimo de un B-tree
    - Cada nodo (excepto la raíz) debe tener al menos t-1 llaves, y cada nodo interno debe tener al menos t hijos
    - Cada nodo puede contener a lo más 2t-1 llaves (nodo lleno), y cada nodo interno puede tener hasta 2t hijos
    - Caso t = 2: árboles 2-3, en la práctica t es mucho más grande

Diccionarios en memoria externa

- B-trees



Diccionarios en memoria externa

- B-trees
  - Altura del árbol
    - Accesos a disco para la mayoría de las operaciones en un B-tree es proporcional a su altura
    - Teorema: la altura de un B-tree de grado t>=2 con n llaves es

$$h \leq \log_t \frac{n+1}{2}$$

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Convenciones:
    - La raíz del B-tree se almacena en memoria principal, por lo que no se requiere leerla desde el disco
      - Se requiere escribir en disco la raíz en caso que cambie
    - Cualquier nodo que se pase como parámetro tuvo que ya haber sido leído del disco

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Búsqueda
    - Muy parecido a buscar en un árbol de búsqueda binaria
      - Con “multiway branching”
      - Llamado inicial: B-Tree-Search(root[T], k)
      - Si k está en el B-tree, se retorna el par ordenado (y,i), nodo y y el indice i tal que $key_i[y] = k$
      - Si no está, se retorna NULL

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Búsqueda

```
B-Tree-Search(x,k)
1 i <- 1
2 while i <= n[x] and k > key_i[x]
3   i <- i+1
4 if i <= n[x] and k == key_i[x]
5   return(x,i)
6 if leaf[x]
7   return NULL
8 else
9   Disk-Read(c_i[x])
10  return B-Tree-Search(c_i[x],k)
```

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Búsqueda
    - Los nodos visitados durante la recursión forman un camino que parte en la raíz del B-tree
    - Por lo tanto, accesos a disco = $O(h) = O(\log_t n)$
    - Tiempo de CPU
      - Loop en líneas 2-3: $O(t)$
      - Tiemp CPU = $O(th) = O(t \log_t n)$

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Crear B-tree
    - Allocate-Node(): reserva un bloque de disco para ser usado como nodo nuevo en tiempo O(1)
    - O(1) accesos a disco y O(1) tiempo de CPU

```
B-Tree-Create(T)
1 x <- Allocate-Node()
2 leaf[x] <- True
3 n[x] <- 0
4 Disk-Write(x)
5 root[T] <- x
```

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Insertar
    - Más complicado que insertar en un árbol de búsqueda binaria
    - Se busca la hoja que debiera contener al nuevo valor, y se inserta ahí
      - El problema es que la hoja puede estar llena (con 2t-1 llaves)
      - Es necesario realizar un split de dicho nodo
        - Split: divide el nodo en dos nodos, cada uno t-1 llaves. La mediana de las llaves sube al nodo padre
          - El padre puede rebalsar también

Diccionarios en memoria externa

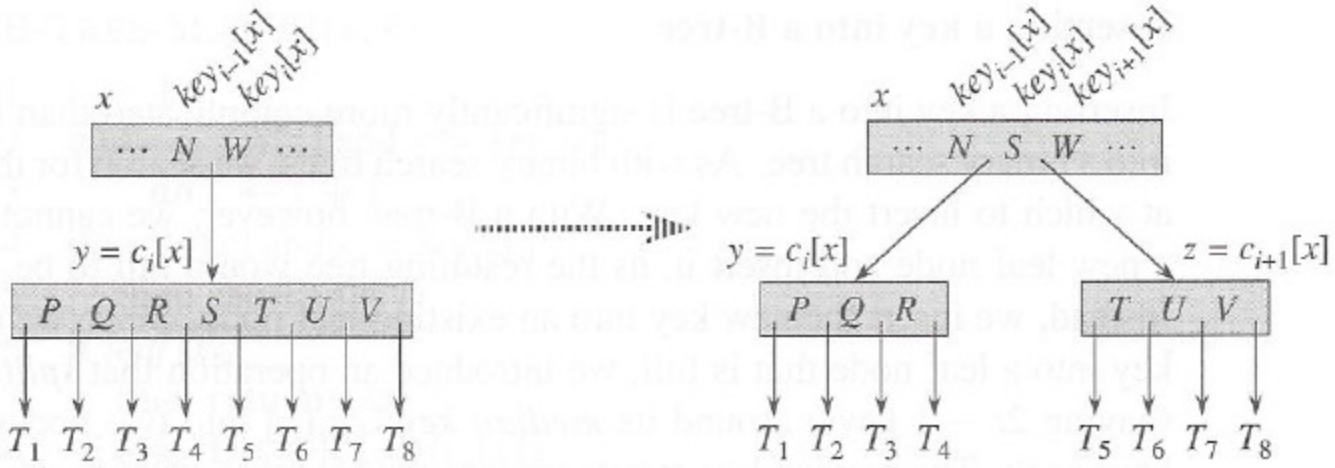
- Operaciones básicas en B-trees
  - Insertar
    - Es posible insertar una llave en un B-tree en un solo recorrido del árbol desde la raíz a una hoja
      - En ese caso, uno no quiere esperar para encontrarse con que debe realizar el split de un nodo
      - A medida que se recorre el árbol buscando la hoja donde insertar la nueva llave, se realizan splits de los nodos llenos encontrados en el camino (incluyendo la raíz)
      - Con esto, se asegura que si es necesario realizar el split de un nodo y, su padre NO está lleno

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Split de un nodo
    - B-Tree-Split-Child toma como parámetros
      - Un nodo interno NO lleno x (en memoria principal)
      - Un índice i
      - Un nodo y (en memoria principal también)
      - Se cumple que $y = c_i[x]$
    - Divide el nodo hijo en dos y ajusta x para que tenga un hijo adicional
      - Si la raíz se divide, se crea una nueva raíz y el árbol crece en un nivel

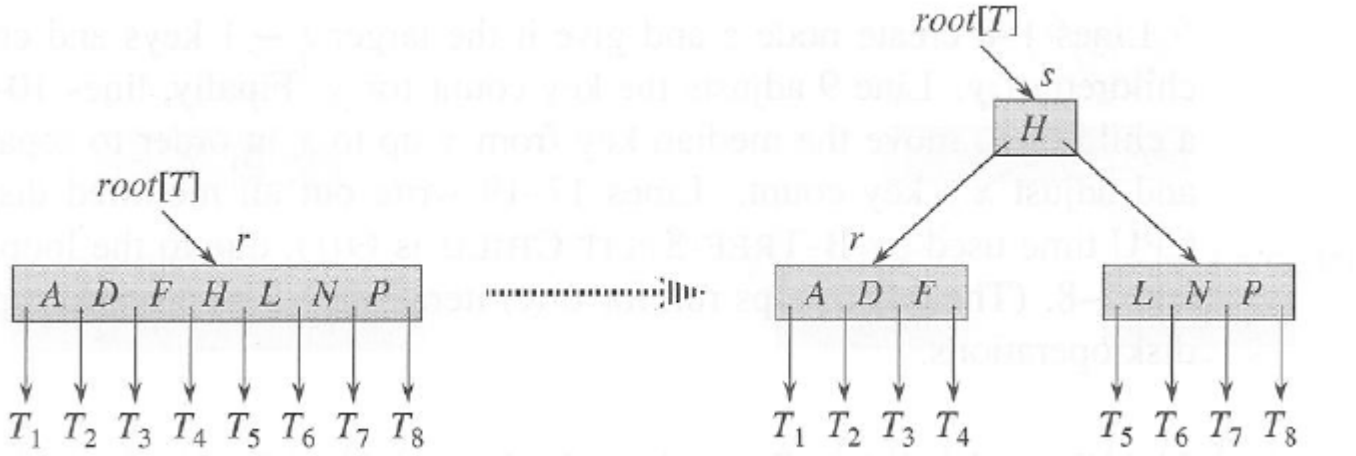
Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Split de un nodo



Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Split del nodo raíz



Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Implementación de split

```
B-Tree-Split-Child(x,i,y)
1  z <- Allocate-Node()
2  leaf[z] <- leaf[y]
3  n[z] <- t-1
4  for j <- 1 to t-1
5    key_j[z] <- key_{j+t}[y]
6  if not leaf[y]
7    for j <- 1 to t
8      c_j[z] <- c_{j+t}[y]
9  n[y] <- t-1
10 for j <- n[x]+1 downto i+1
11   c_{j+1}[x] <- c_j[x]
12 c_{i+1}[x] <- z
13 for j <- n[x] downto i
14   key_{j+1}[x] <- key_j[x]
15 key_i[x] <- key_t[y]
16 n[x] <- n[x]+1
17 Disk-Write(y)
18 Disk-Write(z)
19 Disk-Write(x)
```

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Tiempo de CPU usado por B-Tree-Split-Child es $\Theta(t)$, por ciclos en líneas 4-5 y 7-8
  - Se realizan O(1) accesos a disco

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Algoritmo de inserción
    - Se inserta la llave k en el B-tree de altura h en un solo recorrido del árbol hasta llegar a una hoja
      - Esto requiere O(h) accesos a disco
      - Tiempo de CPU: $O(th) = O(t \log_t n)$
    - Al utilizar el método B-Tree-Split-Child se garantiza que la recursión nunca desciende a un nodo lleno

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Algoritmo de inserción

```
B-Tree-Insert(T,k)
1  r <- root[T]
2  if n[r] = 2t-1
3     s <- Allocate-Node()
4     root[T] <- s
5     leaf[s] <- False
6     n[s] <- 0
7     c1[s] <- r
8     B-Tree-Split-Child(s,1,r)
9     B-Tree-Insert-Nonfull(s,k)
10 else B-Tree-InsertNonFull(r,k)
```

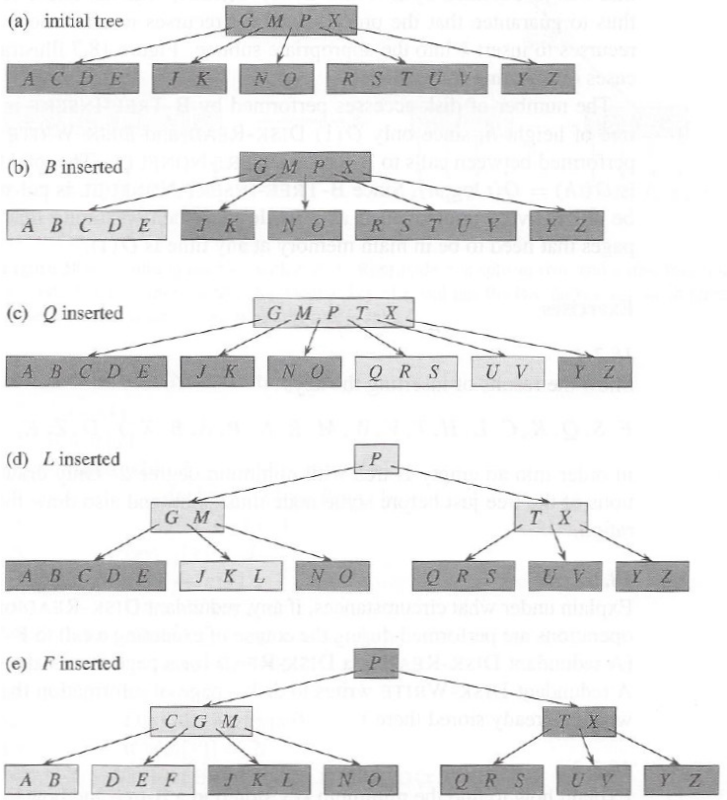
Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Algoritmo de inserción

```
B-Tree-Insert-Nonfull(x,k)
1  i <- n[x]
2  if leaf[x]
3    while i >= 1 and k < key_i[x]
4      key_{i+1}[x] <- key_i[x]
5      i <- i-1
6    key_{i+1}[x] <- k
7    n[x] <- n[x]+1
8    Disk-Write(x)
9  else
10   while i >= 1 and k < key_i[x]
11     i <- i-1
12   i <- i+1
13   Disk-Read(c_i[x])
14   if n[c_i[x]] = 2t-1
15     B-Tree-Split-Child(x,i,c_i[x])
16     if k > key_i[x]
17       i <- i+1
18   B-Tree-Insert-Nonfull(c_i[x],k)
```

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Ejemplo (t=3):



Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Borrar
    - Análogo a insertar, pero un poco más complicado
      - Una llave se puede borrar de cualquier nodo, no solamente una hoja
      - Borrar en un nodo interno implica re-acomodar los hijos del nodo
      - Es necesario prevenir que algún nodo quede con menos de t-1 llaves (exceptuando la raíz)

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Borrar
    - Suponga que el método B-Tree-Delete debe borrar la llave k del subárbol con raíz x
      - El procedimiento que veremos garantiza que el número de llaves de x es al menos t (uno más que el mínimo)
      - Esto puede requerir mover una llave a un nodo hijo ANTES de descender recursivamente a dicho nodo
      - Esto permite borrar una llave en un solo recorrido del árbol (con una excepción, que estudiaremos)
      - Si el nodo x se transforma en un nodo interno sin llaves, x se borra y su único hijo pasa a ser la nueva raíz

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Borrar (algoritmo)
    - 1. Si la llave k está en el nodo x y x es una hoja, borrar k
    - 2. Si k está en el nodo x y x es un nodo interno
      - a. Si el hijo y que precede a k en el nodo x tiene al menos t llaves, encontrar el predecesor k' de k en la rama de y
        - Recursivamente, borrar k', y reemplazar k por k'
      - b. Si el hijo z que sigue a k en el nodo x tiene al menos t llaves, encontrar el sucesor k' de k en la rama de z
        - Recursivamente, borrar k', y reemplazar k por k'

    (Nota: Reemplazar implica volver al nodo x, ésta es la excepción mencionada en slide anterior)

Diccionarios en memoria externa

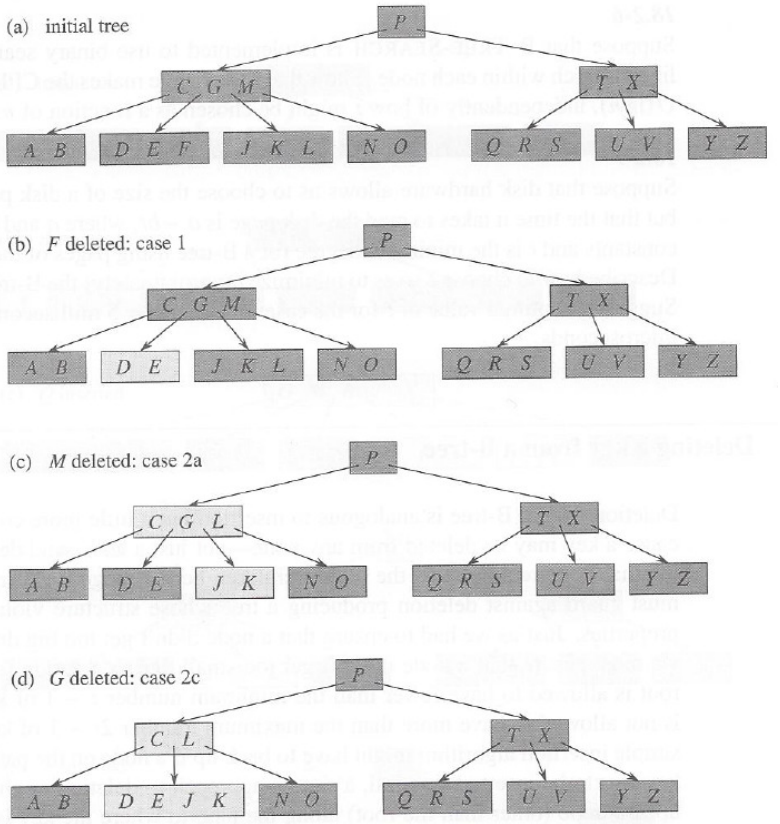
- Operaciones básicas en B-trees
  - Borrar (algoritmo)
    - 2. Si k está en el nodo x y x es un nodo interno (cont.)
      - c. Sino, significa que tanto y como z tienen t-1 llaves
        - Juntar k con todo z en el nodo y, por lo que x pierde tanto k como el puntero a z
        - Ahora, y contiene 2t-1 llaves
        - Luego, liberar z y recursivamente borrar k de y

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Borrar (algoritmo)
    - 3. Si k no está en el nodo interno x
      - Determinar el nodo $c_i[x]$ del subárbol apropiado que debe contener a k (si es que está en el B-tree)
      - Si $c_i[x]$ tiene sólo t-1 llaves, antes del llamado recursivo
        - a. Si $c_i[x]$ tiene un hermano adyacente con al menos t llaves, mover una llave a $c_i[x]$ y el puntero al hijo correspondiente a dicha llave
        - b. Si ambos hermanos tienen t-1 llaves, mezclar $c_i[x]$ con uno de ellos, lo que implica mover una llave desde x que será la mediana del nuevo nodo mezclado

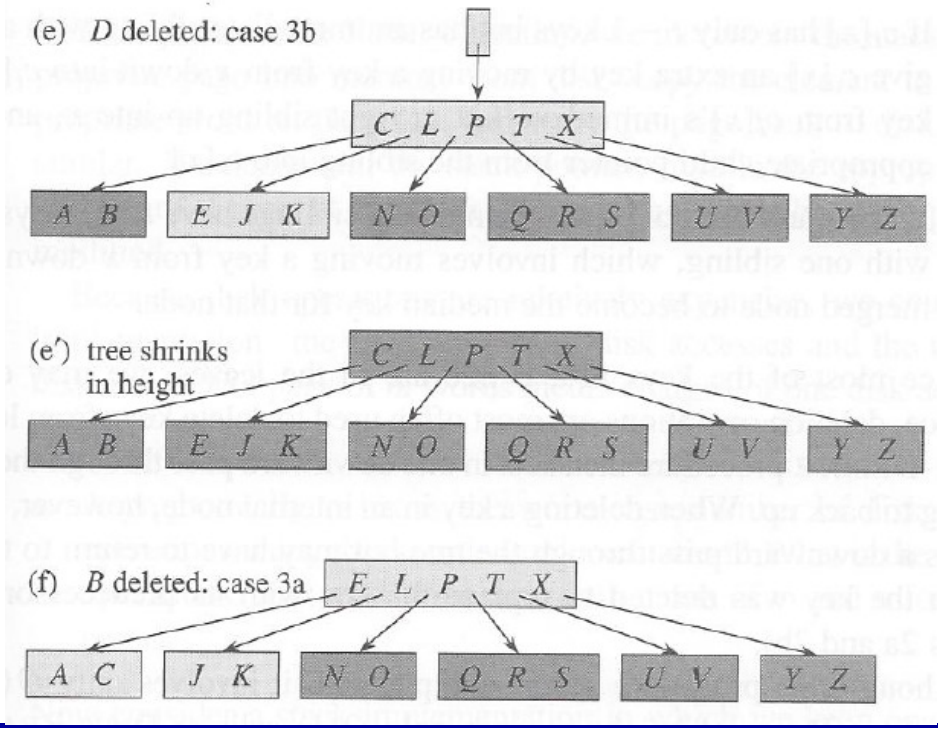
Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Ejemplos de borrado:



Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Ejemplos de borrado:



Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Borrar
    - Dado que la mayoría de las llaves están en las hojas del B-tree, en la práctica es allí donde se realizan la mayoría de los borrados
    - Algoritmo borra en un solo recorrido del árbol, pero puede que tenga que volver a un nodo para reemplazar una llave con su predecesor o sucesor
    - Costo:
      - O(h) operaciones a disco
      - O(th) = O(t $\log_t$ n) tiempo de CPU

Diccionarios en memoria externa

- Operaciones básicas en B-trees
  - Ejercicio: escribir seudocódigo del método borrar

Colas de prioridad en memoria externa

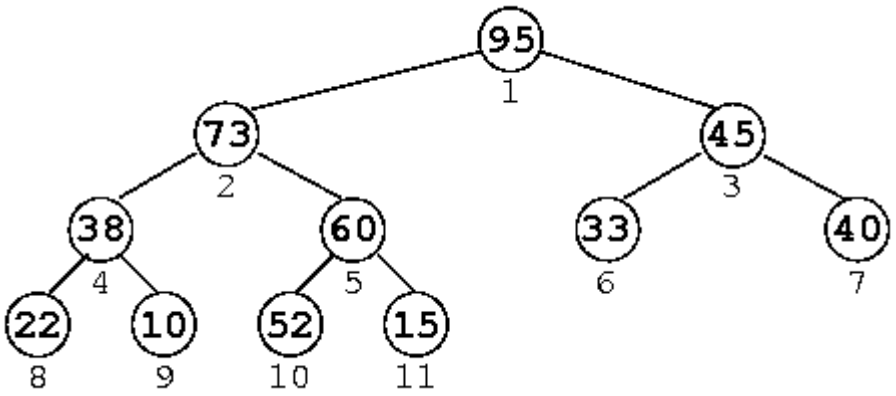
- Cola de prioridad
  - TDA que almacena un conjunto de datos que poseen una llave perteneciente a algún conjunto ordenado
  - Permite insertar nuevos elementos y extraer el máximo (o el mínimo, en caso de que la estructura se organice con un criterio de orden inverso)

Colas de prioridad en memoria externa

- Cola de prioridad
  - Es frecuente interpretar los valores de las llaves como prioridades, con lo cual la estructura permite insertar elementos de prioridad cualquiera, y extraer el de mejor prioridad.
- Operaciones asociadas
  - Insertar un nuevo valor a la cola de prioridad
  - Extraer el máximo (o el mínimo)

Colas de prioridad en memoria externa

- Cola de prioridad
  - Se implementa eficientemente utilizando una estructura llamada heap
    - Un heap es un árbol binario, pero se implementa como un arreglo de valores



95	73	45	38	60	33	40	22	10	52	15
1	2	3	4	5	6	7	8	9	10	11

Colas de prioridad en memoria externa

- Cola de prioridad
  - La característica que permite que un heap se pueda almacenar sin punteros es que, si se utiliza la numeración por niveles indicada, entonces la relación entre padres e hijos es
    - Hijos del nodo j = {2*j, 2*j+1}
    - Padre del nodo k = floor(k/2)
  - Altura del árbol: h = log n

Colas de prioridad en memoria externa

- Cola de prioridad
  - Un heap puede utilizarse para implementar una cola de prioridad
    - Almacenando los datos de modo que las llaves estén siempre ordenadas de arriba a abajo (a diferencia de un árbol de búsqueda binaria, que ordena sus llaves de izquierda a derecha)
    - El padre debe tener siempre mayor prioridad que sus hijos

Colas de prioridad en memoria externa

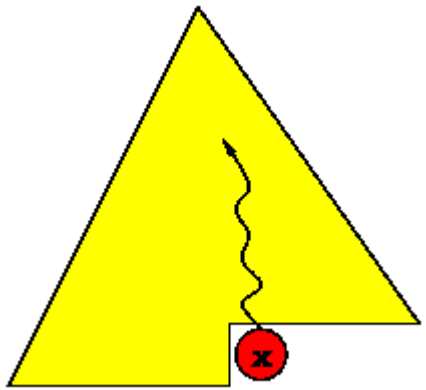
- Cola de prioridad
  - La inserción se realiza agregando el nuevo elemento en la primera posición libre del heap
    - Esto es, el próximo nodo que debería aparecer en el recorrido por niveles o, equivalentemente, un casillero que se agrega al final del arreglo
  - Después de agregar este elemento, la forma del heap se preserva, pero la restricción de orden no tiene por qué cumplirse

Colas de prioridad en memoria externa

- Cola de prioridad
  - Para resolver este problema
    - Si el nuevo elemento es mayor que su padre, se intercambia con él, y ese proceso se repite mientras sea necesario
    - Una forma de describir esto es diciendo que el nuevo elemento "trepa" en el árbol hasta alcanzar el nivel correcto según su prioridad

Colas de prioridad en memoria externa

- Cola de prioridad
  - Gráficamente:



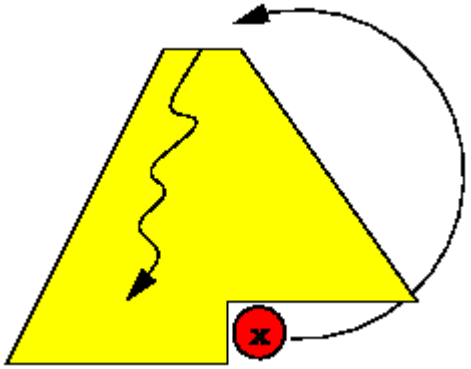
  - Costo peor caso: número constante de operaciones por nivel, O(h) = O(log n)

Colas de prioridad en memoria externa

- Cola de prioridad
  - El máximo está en la raíz del árbol (casillero 1 del arreglo)
    - Al sacarlo de ahí, podemos imaginar que ese lugar queda vacante
    - Para llenarlo, tomamos al último elemento del heap y lo trasladamos al lugar vacante
    - En caso de que no esté bien ahí de acuerdo a su prioridad, lo hacemos descender intercambiándolo siempre con el mayor de sus hijos

Colas de prioridad en memoria externa

- Cola de prioridad
  - Gráficamente:



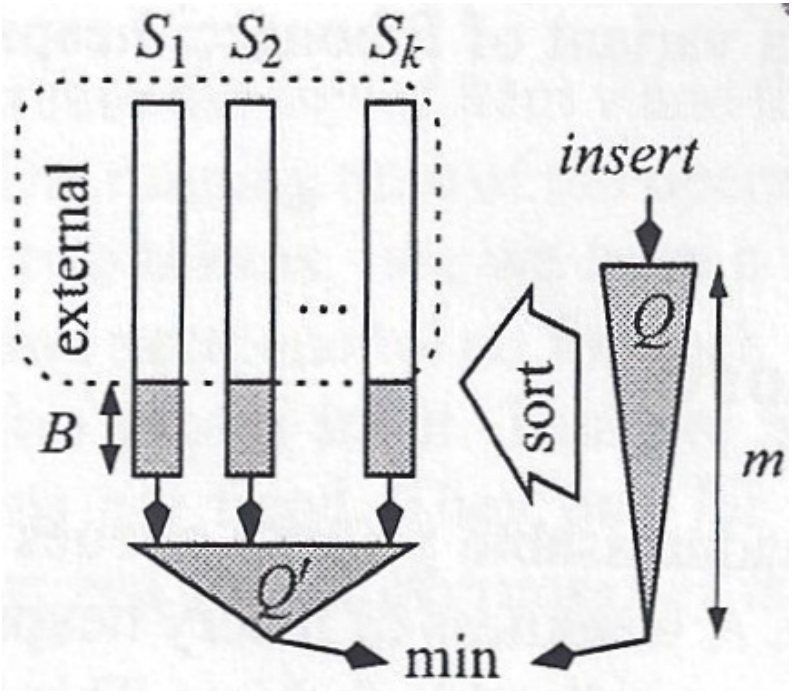
  - Costo peor caso: número constante de operaciones por nivel, O(h) = O(log n)

Colas de prioridad en memoria externa

- Cola de prioridad en memoria secundaria
  - Las colas de prioridad implementadas con un heap tienen una debilidad
    - En la operación de extracción, el descenso de un nodo hasta su posición correspondiente es impredecible
    - Si el heap estuviera almacenado en disco, esto implica accesos aleatorios al disco, lo que es muy lento
    - Estudiaremos una estructura de datos para colas de prioridad con accesos a disco más regulares

Colas de prioridad en memoria externa

- Componentes
  - Colas de prioridad en memoria primaria
  - Ordenamiento
  - Multiway merging
- Esquema básico:



Colas de prioridad en memoria externa

- Estructura de datos
  - Consiste en
    - Dos colas de prioridad Q (insertion queue) y Q' (deletion queue)
    - k secuencias ordenadas $S_1,\ldots, S_k$
  - Cada elemento en la cola de prioridad se almacena en Q, en Q' o en alguna de las secuencias ordenadas
    - El tamaño de Q está limitado por un parámetro m
    - Q' almacena el menor elemento de cada secuencia junto con un índice de la secuencia respectiva

Colas de prioridad en memoria externa

- Inserción
  - Los elementos nuevos se insertan en Q
  - Si Q está llena, ésta se vacía primero
    - En este caso, sus elementos forman una nueva secuencia ordenada

```
Priority-Queue-Insert(e)
1 if |Q| = m
2   k++
3   S_k = sort(Q)
4   Q = ϕ
5   Q'.insert(S_k.popFront)
6 Q.insert(e)
```

Colas de prioridad en memoria externa

- Inserción
  - El mínimo se guarda en Q o Q'
  - Si el mínimo está en Q' y viene de la secuencia s_i, el siguiente elemento mayor de s_i se inserta en Q'

```
Priority-Queue-DeleteMin()
1 if minQ <= minQ'
2   e = Q.deleteMin
3 else
4   (e,i) = Q'.deleteMin
5   if S_i != {} // si tiene elementos
6     Q'insert(S_i.popFront)
7 return e
```

Colas de prioridad en memoria externa

- Uso de memoria
  - Q y Q' se almacenan en memoria primaria
  - El parámetro m se fija tal que
    - Sea una fracción constante de la memoria principal
    - Sea múltiplo del tamaño de bloque B

Colas de prioridad en memoria externa

- Uso de memoria
  - Las secuencias ordenadas s_i se guardan mayoritariamente en memoria secundaria
    - Inicialmente, sólo los B elementos menores de s_i se almacenan en un buffer b_i en memoria principal
    - Cuando se elimina el último de los elementos de b_i, se cargan los siguientes B elementos de S_i
    - Note que efectivamente se mezclan las secuencias S_i

Colas de prioridad en memoria externa

- Complejidad
  - Cada elemento insertado se escribe en disco a lo más una vez, y se lee de vuelta a memoria principal a lo más una vez
  - Dado que todos los accesos a disco son en unidades de al menos un bloque lleno, el número de accesos a disco es a lo más n/B por n operaciones en la cola de prioridad

Colas de prioridad en memoria externa

- Requerimientos en memoria principal
  - Es a lo más m+kB+2k
  - Si M es el tamaño total de la memoria principal, la estructura cabe en memoria si
    - m = M/2
    - k <= floor((M/2-2k)/B) ≈ M/(2B)
  - Si se realizan muchas inserciones, la memoria principal se podría acabar
    - Esto ocurre después de m(1+floor((M/2-2k)/B))≈M^2/(4B) inserciones