Ordenación externa

Prof. Cristian Cappo

abril/2018

Contenido

- Conceptos, motivación
- **Problema 1**: ordenar un archivo con muchos datos
- **Problema 2**: almacenar un gran cantidad de información y accederla eficientemente.
  - **B-Árboles**
    - Estructura
    - Operaciones

Discos

- La memoria principal es limitada (MB a GB)
- La memoria caché aún más (KB a MB)
- Los discos sin embargo van de GB a TB y hasta PB(*experimentalmente*)
  - Su costo es bajo

    pero su problema

    es la baja velocidad
  - Giran entre 5400

    y 15000 rpm

The IBM hard drive in 1956:

* Stored 5 megabytes (MB)
* Cost $11,000 per megabyte
* Was 60 inches long x 68 inches high x 29 inches deep
* Weighed about 1 ton

https://www.backblaze.com/blog/94-trillion-petabyte/

Comparación de precios en almacenamiento

Tabla de comparación de Precios – US$/MB

Medium	1996	1997	2000	2004	2006	2008	2011
RAM	$45.00	7.00	1.500	0.3500	0.1500	0.0339	0.0138
Disk	0.25	0.10	0.010	0.0010	0.0005	0.0001	0.0001
USB drive	–	–	–	0.1000	0.0900	0.0029	0.0018
Floppy	0.50	0.36	0.250	0.2500	–	–	–
Tape	0.03	0.01	0.001	0.0003	–	–	–
Solid State	–	–	–	–	–	–	0.0021

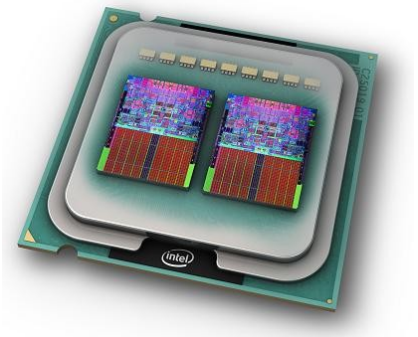
2013: RAM=0.009 **Disk=**0.00005 **USB=**0.00128 **Solid State=**0,001
2018: RAM=0.006 **Disk=**0.000032 **USB=**0.00075 **Solid State=**0,00031

Nota interesante: los ratios de tiempo, espacio y costo entre la memoria y el disco se han mantenido estables sorpresivamente por cerca de 20 años

Fuente: C. Shaffer. *Data Structures and Algorithm Analysis* (Java Edition). 2013

- El tiempo acceso a estos dispositivos varía:
  - Cache 1 GHz (1,000,000,000)
  - Memoria principal (*5-10 ns*) 100 MHz(100,000,000)
  - Discos (*9-10 ms*) 100 Hz (100)
- El ratio es de 1 millón, es decir, el disco es 1 millón de veces más lento que la memoria RAM.

Valores aproximados del año 2011

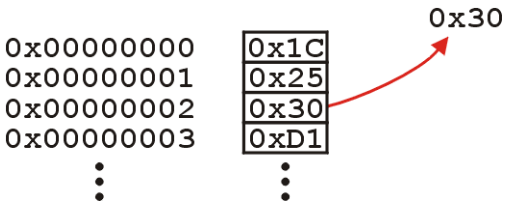


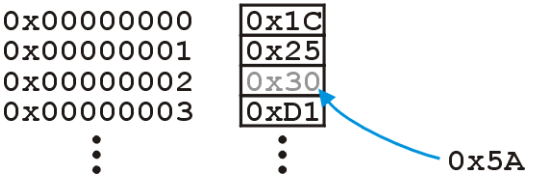
- La memoria (en general) es direccionable por byte
  - Cada byte tiene su propia dirección
  - Cada byte puede ser accedido o modificado
- Para acceder a un bit, debe accederse al byte que lo contiene

- Los discos son bloque-direccionables
  - El tamaño del bloque es variable
  - Asumimos que los bloques son de 4 KBytes (para nuestros ejemplos)
- Para acceder a un byte de un bloque debe leerse el bloque entero que lo contiene.

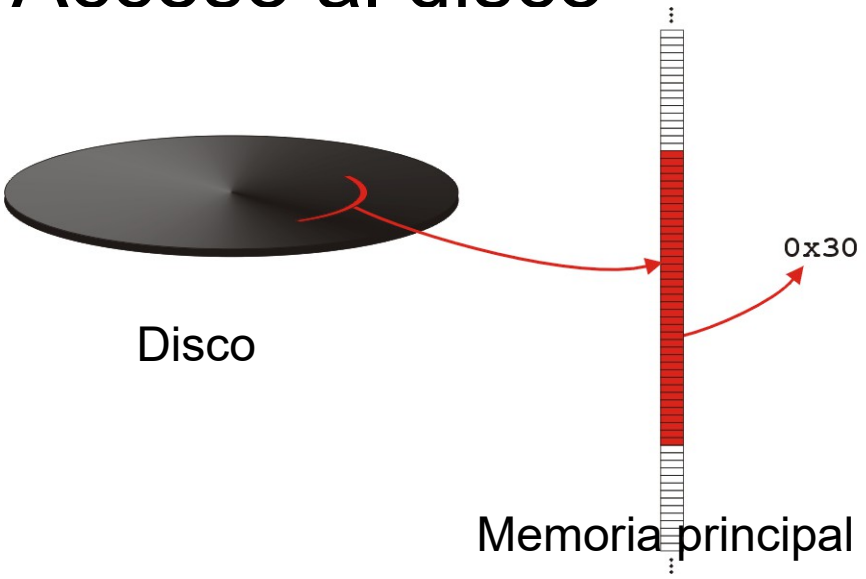
- El controlador de I/O lee y escribe en el disco por bloques.
- El bloque es copiado a la memoria principal
  - DMA (Direct Memory Access)
- Para acceder a un byte del disco, el controlador debe leer el bloque entero de 4 KB (4096 bytes) (*puede ser otros tamaños de bloque pero esto es lo usual*)

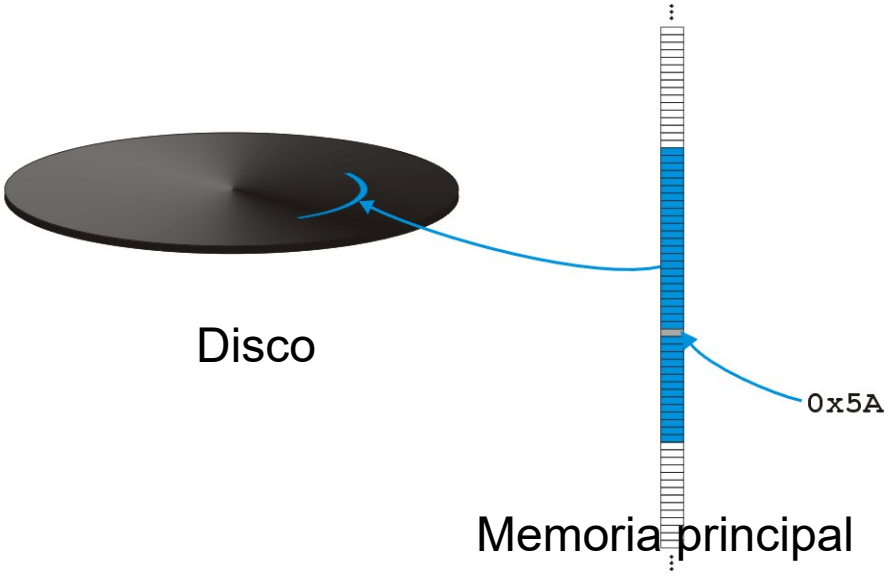
- Acceso a la memoria





- Acceso al disco





- Cuando se da formato a un disco duro, se especifica (o se puede especificar) el tamaño de bloque.
- Todos los archivos ocupan un número entero de bloques
  - El último bloque puede ser parcialmente ocupado

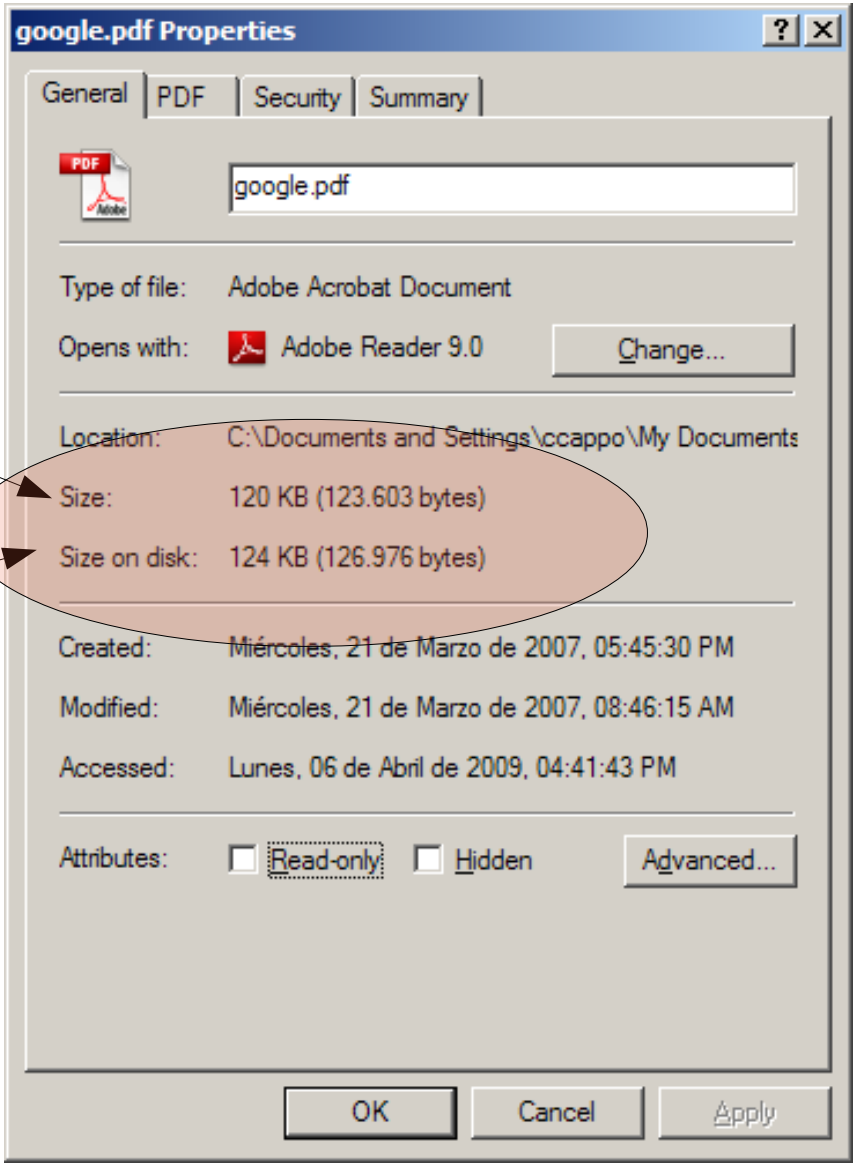
```
C:> format d: /A:4096
C:> format d: /A:8192
C:> format d: /A:16K
```

Ejemplo en Windows (NTFS)

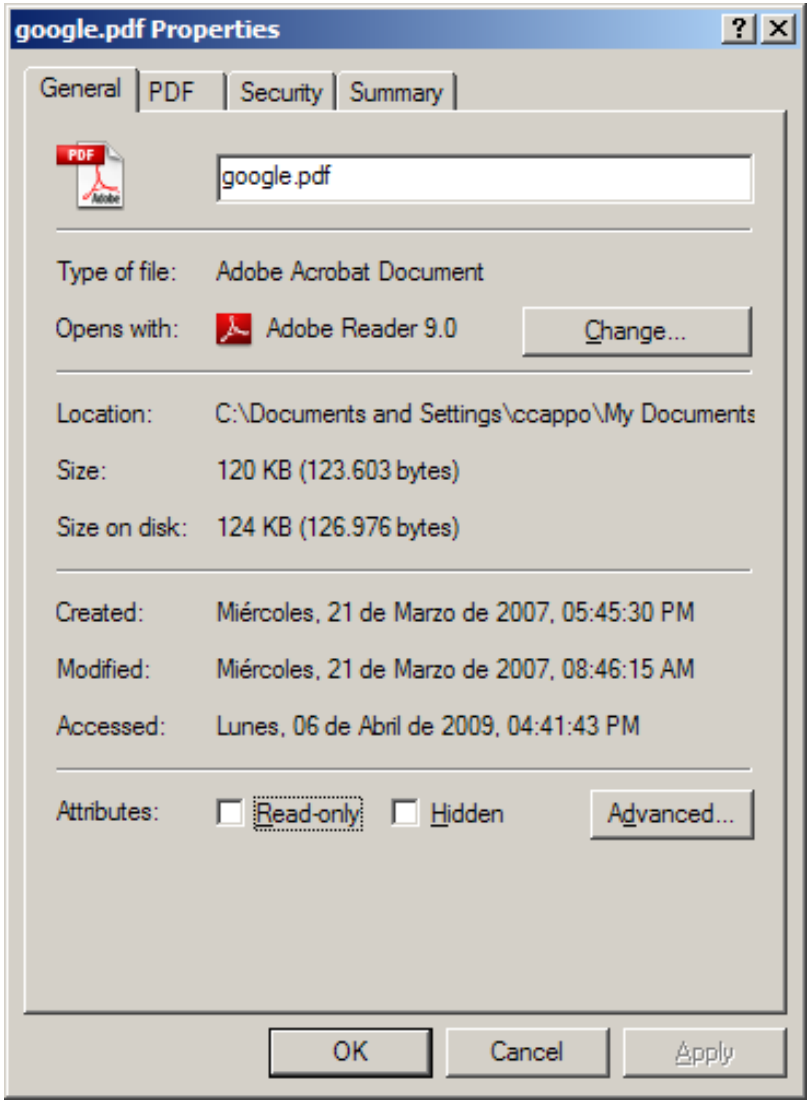
```
# mkfs -t ext4 -b 1024 /dev/sdb1
# mkfs -r ext3 -b 2048 /dev/sdb2
```

Ejemplo en Linux

- Por ejemplo, el archivo mostrado a la derecha tiene123.603 bytes
- Pero su tamaño en disco es 126.976 (31 * 4096)



- Consecuentemente
  - *3373 bytes no se utilizan*
- En tamaños de bloque muy grande existe un desperdicio de espacio y en tamaños muy pequeños se aumenta la E/S



- **En resumen**:
  - Los discos son mucho, muchísimo más lentos, que la RAM.
  - Los discos se dividen en bloques:
    - Por comodidad usamos 4 KB
  - Acceder a un bloque entero tiene el mismo costo de acceder a un solo byte dentro del bloque.

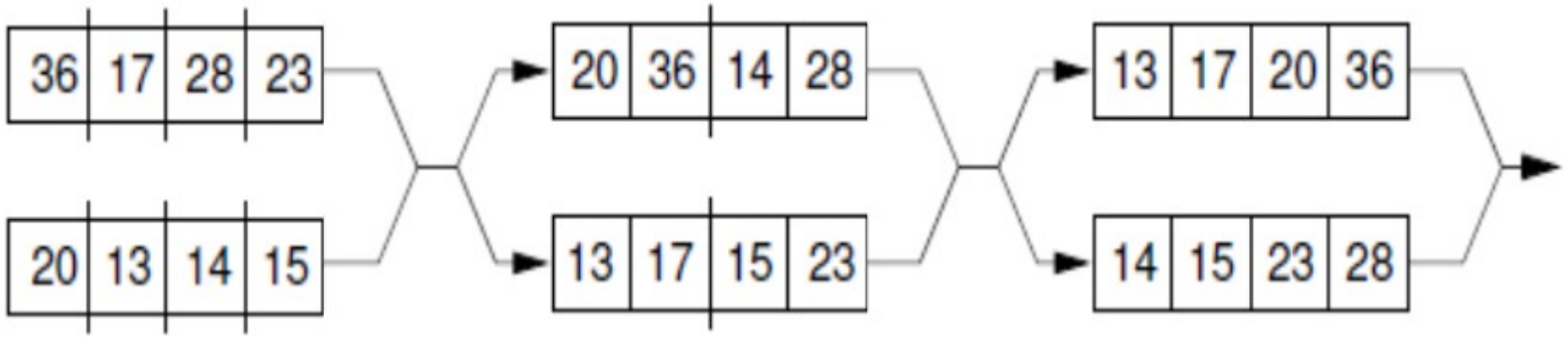
Problema 1: Ordenar un archivo con muchos datos

¿Cómo puede ordenar un archivo que se encuentra en disco duro y que contiene, por ejemplo 7000000 de datos de personas (cedula, apellido, nombre, etc). Queremos la lista ordenada por apellido y nombre.

¿Cómo lo plantearía en lenguaje C?

Ejemplo utilizando la idea de MergeSort (*MergeSort externo*)

- Queremos ordenar la secuencia 36,17,28,23,20,13,14,15 que se encuentra en memoria.



Optimizaciones posibles a Mergesort externo

- Utilizar un algoritmo de ordenación interno al principio con un número de datos que puede leerse de una vez del disco.
- Podemos aumentar la cantidad de ***carreras*** (en vez de dos carreras que vimos en el ejemplo). Así se convierte en un algoritmo multicarreras.
- Hay muchas variantes pero todas son basadas en los siguientes dos pasos:
  - Partir el archivo en carreras iniciales
  - Combinar las carreras para formar un solo archivo ordenado

Problema 2: Almacenar gran cantidad de datos y accederlos de forma eficiente

- Suponga que nosotros queramos guardar una gran cantidad de datos ordenados.
- Consideremos 10^9 ítems (1.000.000.000)
  - 4 byte de clave más el dato asociado
  - Asumimos que los punteros necesitan 4 bytes

Restricciones y asunciones

- Debemos guardar en disco duro
  - 32 bits solo puede acceder a 4 GB (teóricamente) (asumimos un máquina de 32 bits)
  - RAM es volátil
- Asumamos que el nodo raíz esta en memoria principal.
- Las direcciones de bloques son de 32 bits
- Tiempo de acceso a disco: 10 ms.

- Consideramos utilizar la estructura árbol para guardar esta información

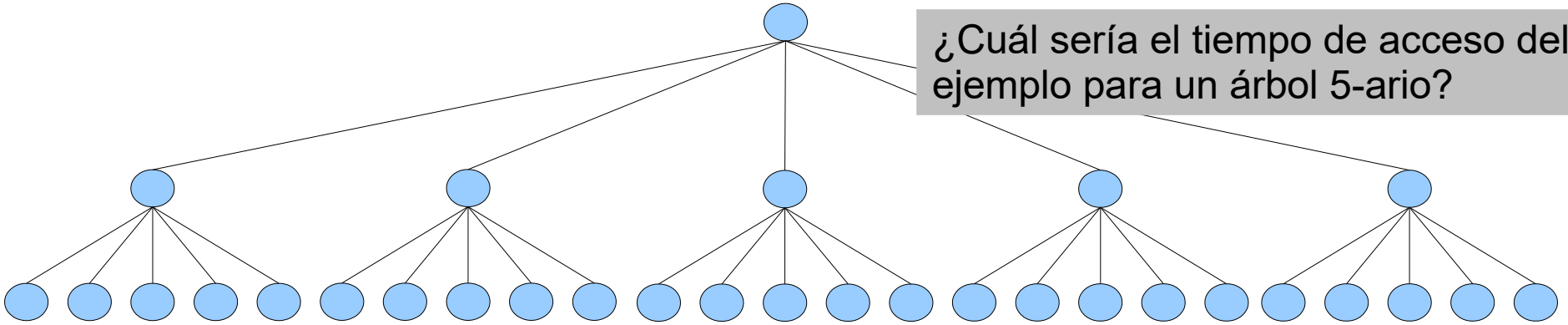
- Hemos visto que en general acceder a un árbol balanceado es O(*lg n*).
- Consideramos para nuestro análisis
  - Un árbol binario de búsqueda
  - Un árbol M-ario
  - B-Arbol

BST

- Un BST con 10^9 entradas tiene un ***mínimo*** de 29 de altura ($\lg_2 (10^9)$) = 29
- Solo los punteros requieren cerca de 8 GB
  - 32 bits solo accede a 4 GB.
- Es muy probable que dos nodos no se encuentren en el mismo bloque, entonces:
  - Acceder a las hojas requiere que 29 bloques sean copiados desde el disco duro
- Con 10 ms de tiempo de acceso, esto puede requerir cerca de 0,29 s (10*29/1000)

Árbol M-ario

- Consideremos un árbol M-ario
  - El BST hace que tengamos que leer demasiados bloques. Además podemos esperar lo peor, que no se encuentre balanceado.
  - Podemos tener más ramificaciones en el árbol, así tenemos menos altura en el árbol. Así, mientras un árbol binario perfecto de 31 nodos tiene 5 niveles, un árbol 5-ario de 31 nodos tiene sólo tres niveles.



¿Cuál sería el tiempo de acceso del ejemplo para un árbol 5-ario?

Árbol M-ario

- Entonces un árbol M-ario permite ramificaciones de *M* hijos.
- A medida que el grado de ramificación aumenta, la profundidad disminuye.
- La altura de árbol binario completo es aproximadamente $log_2 n$, mientras que la altura de un árbol M-ario perfecto es aproximadamente $log_M n$.

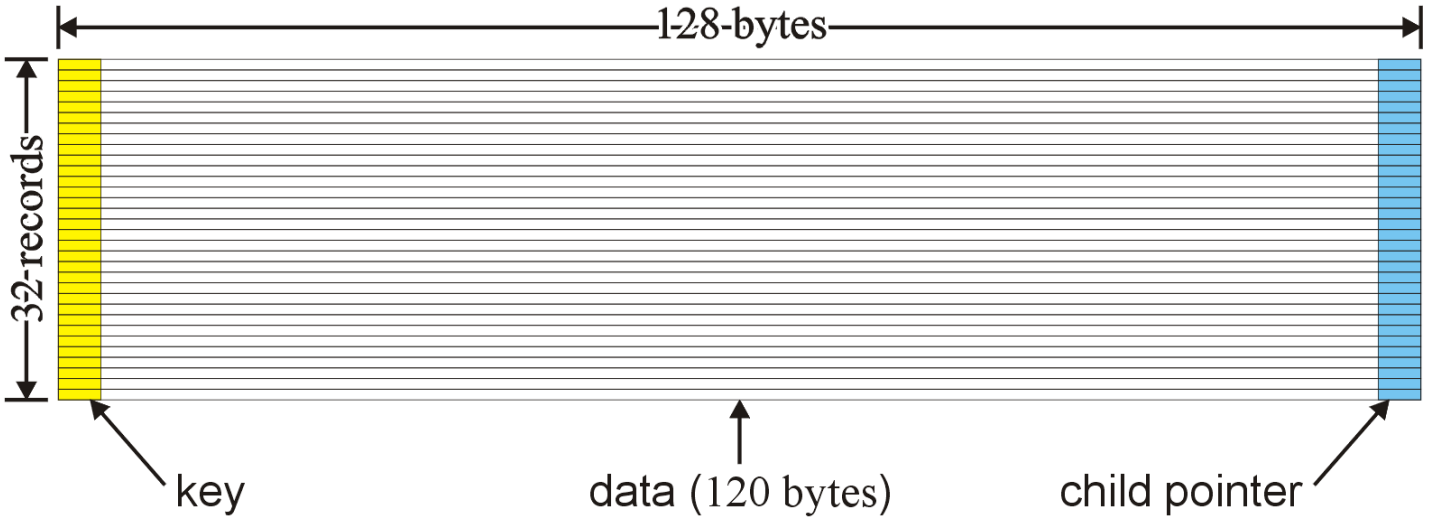
- Continuando con el ejemplo, tratemos que colocar la mayor cantidad de claves y punteros en un bloque.
  - Clave tiene 4 bytes
  - Punteros tienen 4 Bytes (32 bits)
- Asumimos que un bloque tiene 4 KB
  - 4 KB / (4 Bytes + 4 Bytes) = 512
- Entonces podemos tener un nodo de 512 ramificaciones en cada bloque.

- Un árbol M-ario óptimo tiene una altura de $\log_{512} 10^9 = 3$
- Ahora las búsquedas requieren aprox. 30 ms, cerca de 10 veces más rápido que el BST.

Árbol M-ario realista

- Problema:
  - Solo guardamos clave y punteros
  - No guardamos otra información asociada
- En general querríamos guardar más información.

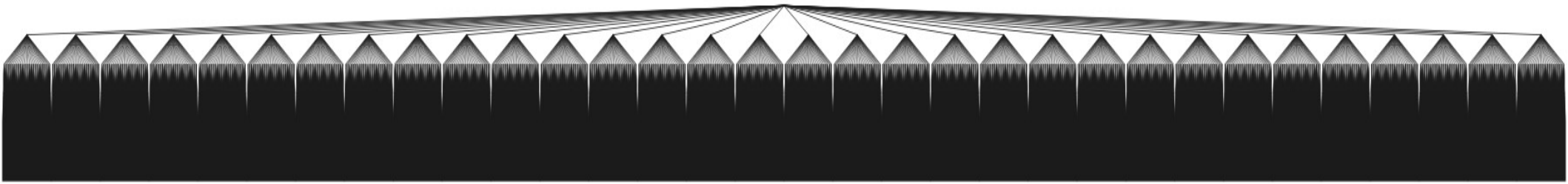
- Asumimos que queremos guardar
  - La clave de 4 bytes
  - Un puntero de 4 bytes (al bloque hijo), y 120 bytes de información
- Consecuentemente 4KB / 128 bytes = 32
- Cada bloque puede guardar 32 registros
  - Usamos un árbol M-ario de 32 ramificaciones.



- Así, la máxima altura debería ser:

$$\log_{32} 10^{9} = 5$$

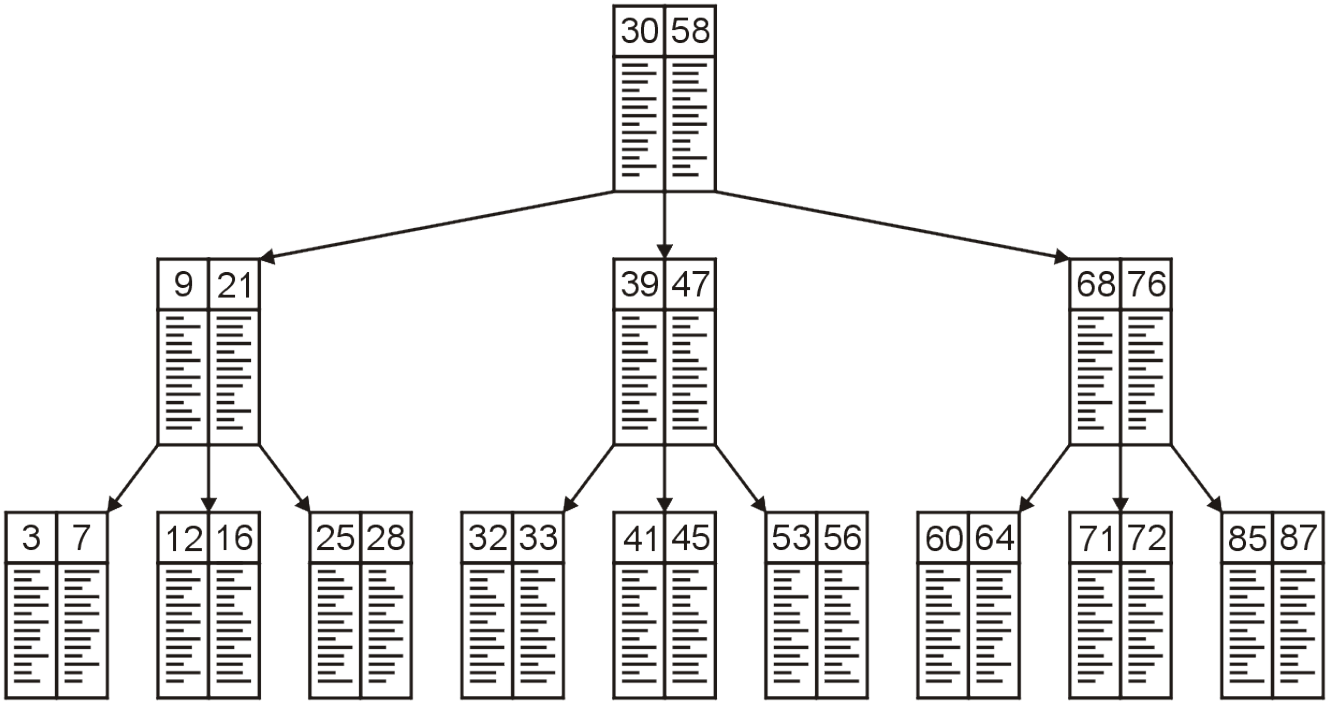
- Esto incrementa el tiempo de búsqueda en 67% sobre un árbol M-ario de 512 ramificaciones.



- Problema:
  - Nosotros no queremos incrementar el tiempo de búsqueda.
  - Cerca de 97 % de todos los registros están en las hojas. *Por cada raíz existen 32 hijos (31/32).*
- Solución:
  - Mover los datos (4 byte de clave y 120 bytes de información por registro) a las hojas.

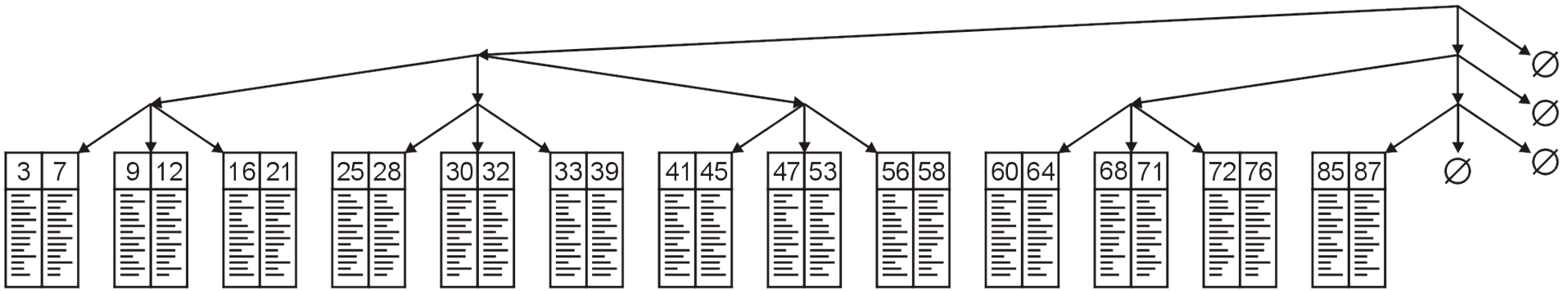
Árbol M-ario realista

- Considere este árbol M-ario de 3 ramificaciones
  - Clave entera y datos



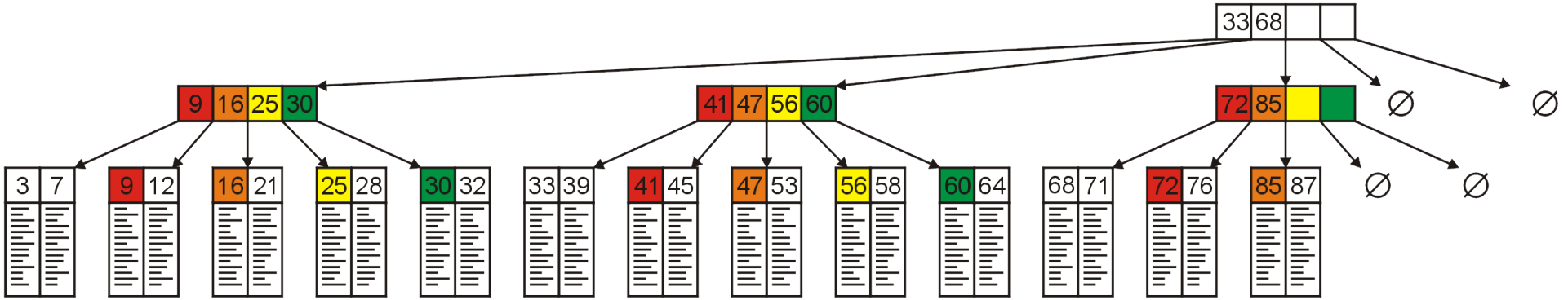
Árbol M-ario realista

- Movemos todos los datos a las hojas, pero necesitamos acceder a las hojas..

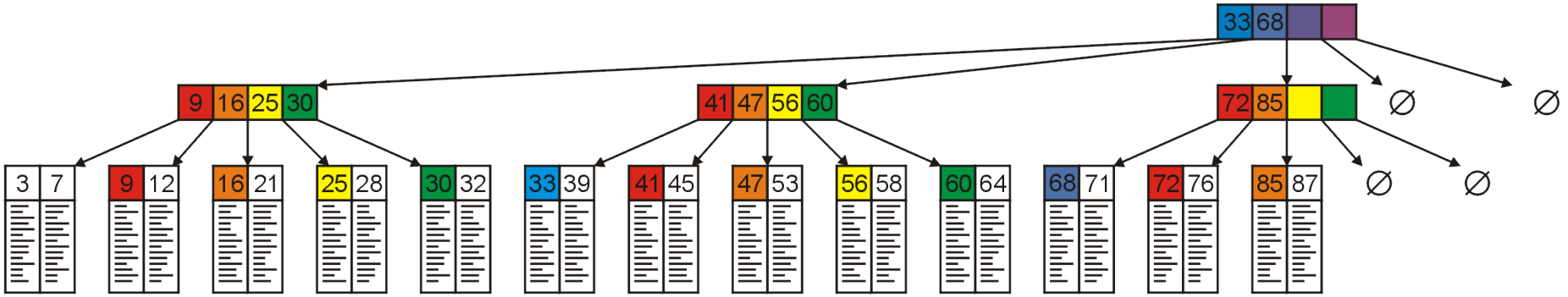


- Entonces, como dejamos todos los datos en las hojas, podríamos guardar más claves y punteros en los nodos internos.
- Tenemos pues, que los nodos internos pueden ser M-arios con más ramificaciones.

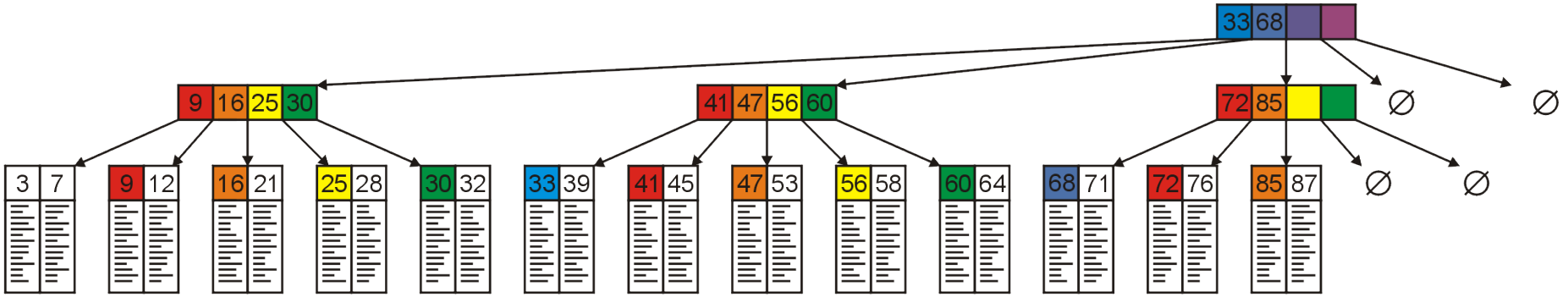
- Un árbol M-ario de ramificación 5
  - Guardamos la menor entrada de cada subárbol menos del primero.



- Y podemos hacer en todos los niveles



- Buscar ahora es sencillo,
  - Encontrar las entradas 28 y 63.



B-Arbol

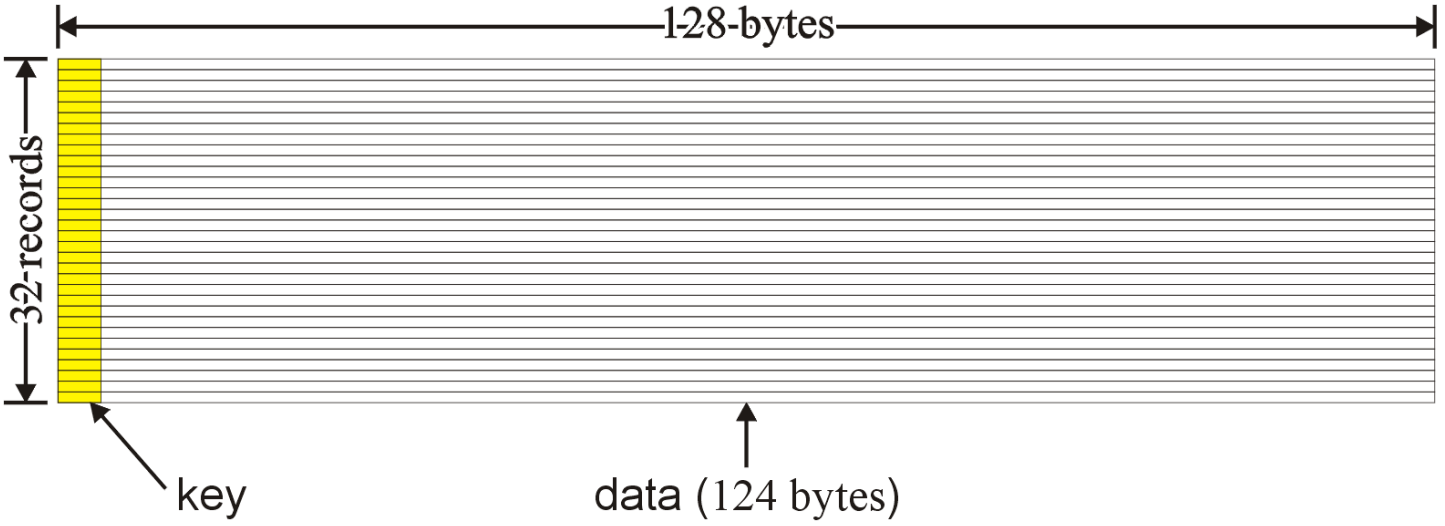
- Un árbol B (o B-Arbol) usa esta idea y
  - Propuesta originalmente por Rudolf Bayer y E. McCreight en el año 1972.
  - Agrega restricciones para asegurar el balance y la consistencia
- La meta es:
  - Minimizar el acceso a disco
  - Tener un tiempo de acceso uniforme
- Los datos son guardados en las hojas y los nodos internos forman un árbol M-ario.

B-Arbol

- El tamaño de bloque puede afectar
  - La cantidad de datos que pueden ser guardados en las hojas
  - La elección de *M* para los nodos internos
- Maximizar la cantidad de datos que pueden ser guardados en un bloque (de 4 KB para nuestros ejemplos)

B-Arbol

- Suponga que nuestra clave tiene 4 bytes y los datos tiene 124 bytes
- Un bloque de 4 KB guarda L=32 registros y requeriría:
  - $10^9/32$ = 31.250.000 bloques de hoja

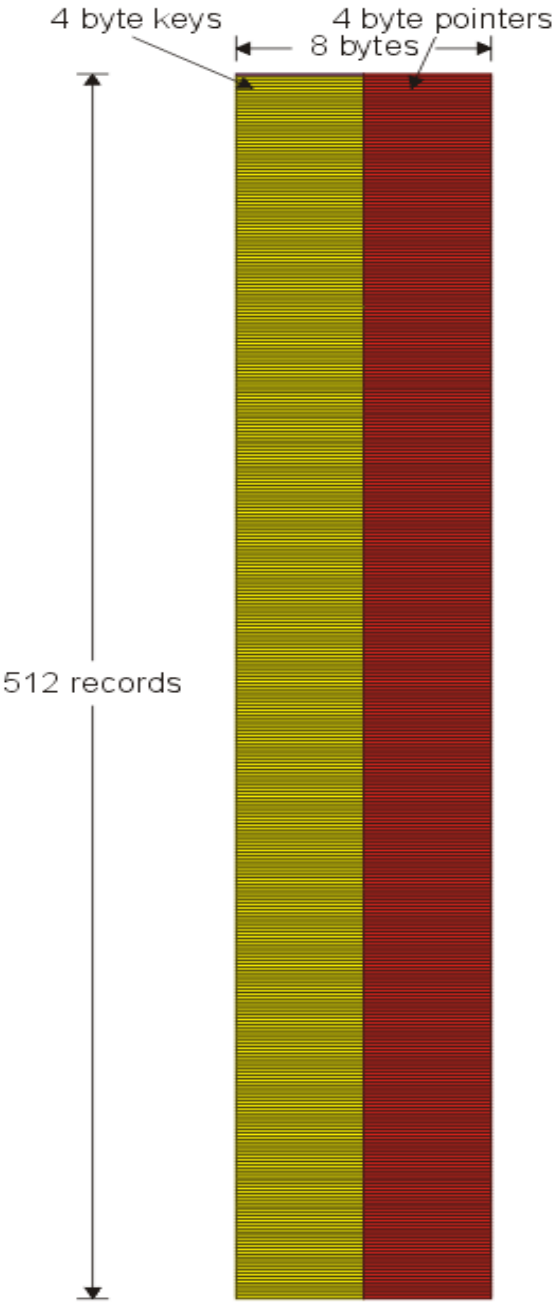


B-Arbol

- Vamos al nodo interno
- Ya que todos los datos están en la hoja, ahora solo necesitamos guardar:
  - Claves (que permitan la navegación)
  - Punteros

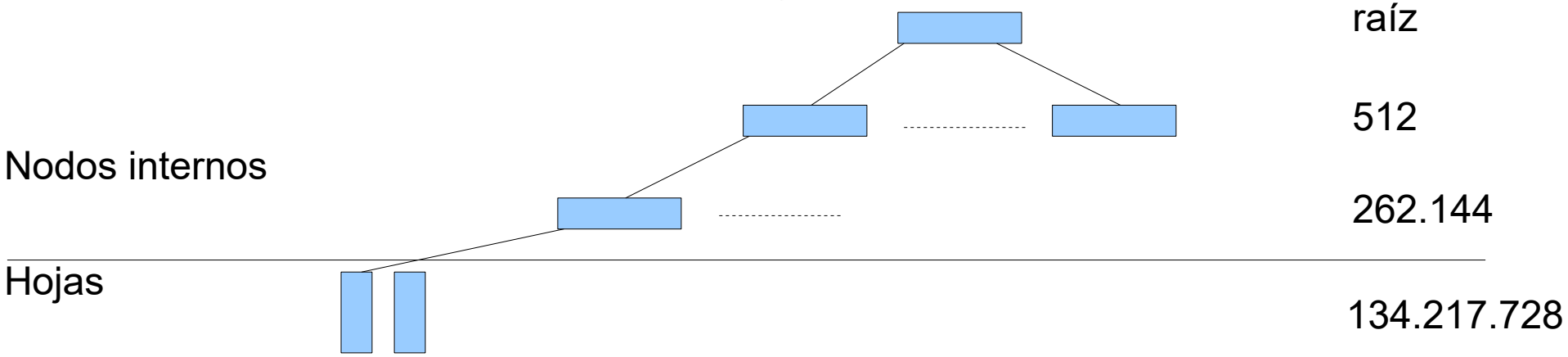
B-Arbol

- Esto, dentro de los 4KB, nosotros podemos guardar 512 par de clave-puntero.
- Esto permite tener un árbol M-ario de 512 ramificaciones

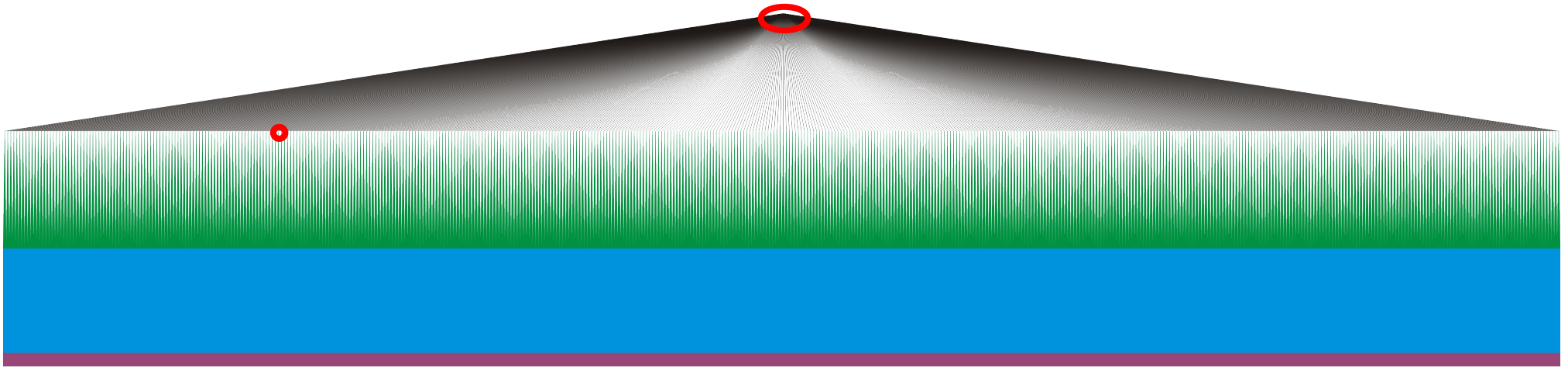


B-Arbol

- Mil millones de registros requieren:
  - $10^9/32 = 31.250.000$ bloques de hoja
  - Un árbol M-ario de 512 ramificaciones con altura $\log_{512}(10^9/32) = 2.76 \approx 3$
  - El nivel más bajo tiene 512^2 hojas por 512-ario nodos para un total de $512^3 = 134.217.728$ direcciones.
  - La altura total es de 3.



- Cada acceso requiere tres accesos a disco o 30 ms.
  - La raíz siempre esta en memoria
  - Nosotros buscamos el hijo requerido

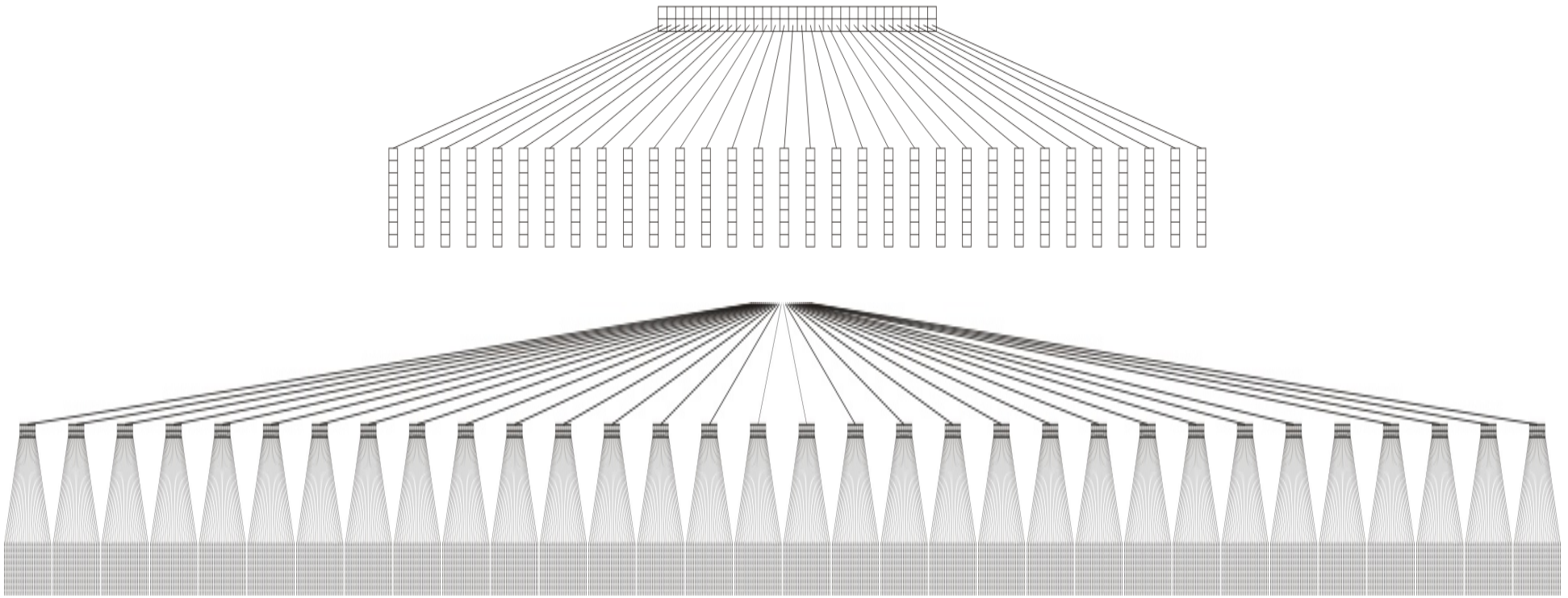


B-Arbol

- Un B-Arbol es un árbol donde:
  - Todos los nodos hojas son arreglos de ***L*** registros ordenados por una clave.
  - Todos los nodos internos forman un árbol M-ario que guarda M-1 claves y M sub-arboles.
- Además:
  - Requerimos agregar la condición de que todos los bloques de las hojas tengan la misma altura
    - Esto garantiza el mismo tiempo de acceso para todos los registros.

B-Arbol

- Un B-Arbol de altura 1 y 2
  - Con nodos internos de 32 ramificaciones
  - 8 registros por hoja M=32 L=8



B-Arbol

- En muchos casos los nodos pueden no llenarse
  - Hojas parcialmente llenas
  - Nodos internos con menos de M sub-árboles
- Entonces, debemos agregar reglas que aseguren un balance razonable

B-Arbol

- Definimos un B-Arbol de orden M como sigue:
  - Todos los datos se guardan en las hojas
  - Todas las hojas están
    - En el mismo nivel
    - Contienen entre $\lceil L/2 \rceil$ y L entradas
      - Es decir al menos medio llenos
    - L depende del tamaño del registro que se guardará y el tamaño del bloque.

B-Arbol

- El nodo raíz puede ser
  - Un nodo hoja o
  - Un nodo interno M-ario que tiene entre 2 y M hijos
- Todos los otros nodos internos
  - Son M-arios entre $\lceil M/2 \rceil$ a M hijos
  - Guardan M-1 claves para guiar la búsqueda donde k representa el menor valor en el sub-arbol k.

B-Arbol

- El mejor caso de nuestro ejemplo fue

$$\lfloor \log_{512} (10^9/32) \rfloor + 1 = 3$$

- Asumiendo el peor caso
  - Todos los nodos hojas son llenos a la mitad (16) y
  - Todos los nodos internos llenos a la mitad (256)
  - Entonces la altura del árbol sería

$$\lfloor \log_{256} (10^9/16) \rfloor + 1 = 4$$

B-Arbol

- Nro. de registros para el mejor y peor caso con M=512 y L=32

ALTURA
0
1
2
3
4
5
6
7
8

REGISTROS	MEMORIA
32	2^{12} B
1.64×10^{3}	2^{21} B
8.39×10^{6}	2^{30} B
4.30×10^{9}	2^{39} B
2.20×10^{12}	2^{48} B
1.13×10^{15}	2^{57} B
5.76×10^{17}	2^{66} B
1.64×10^{20}	2^{75} B
1.64×10^{23}	2^{84} B

REGISTROS	MEMORIA
2	2^{12} B
32	2^{20} B
8.19×10^{3}	2^{28} B
2.10×10^{6}	2^{36} B
5.37×10^{8}	2^{44} B
1.37×10^{11}	2^{52} B
3.52×10^{13}	2^{60} B
9.01×10^{15}	2^{68} B
2.31×10^{18}	2^{76} B

B-Arbol

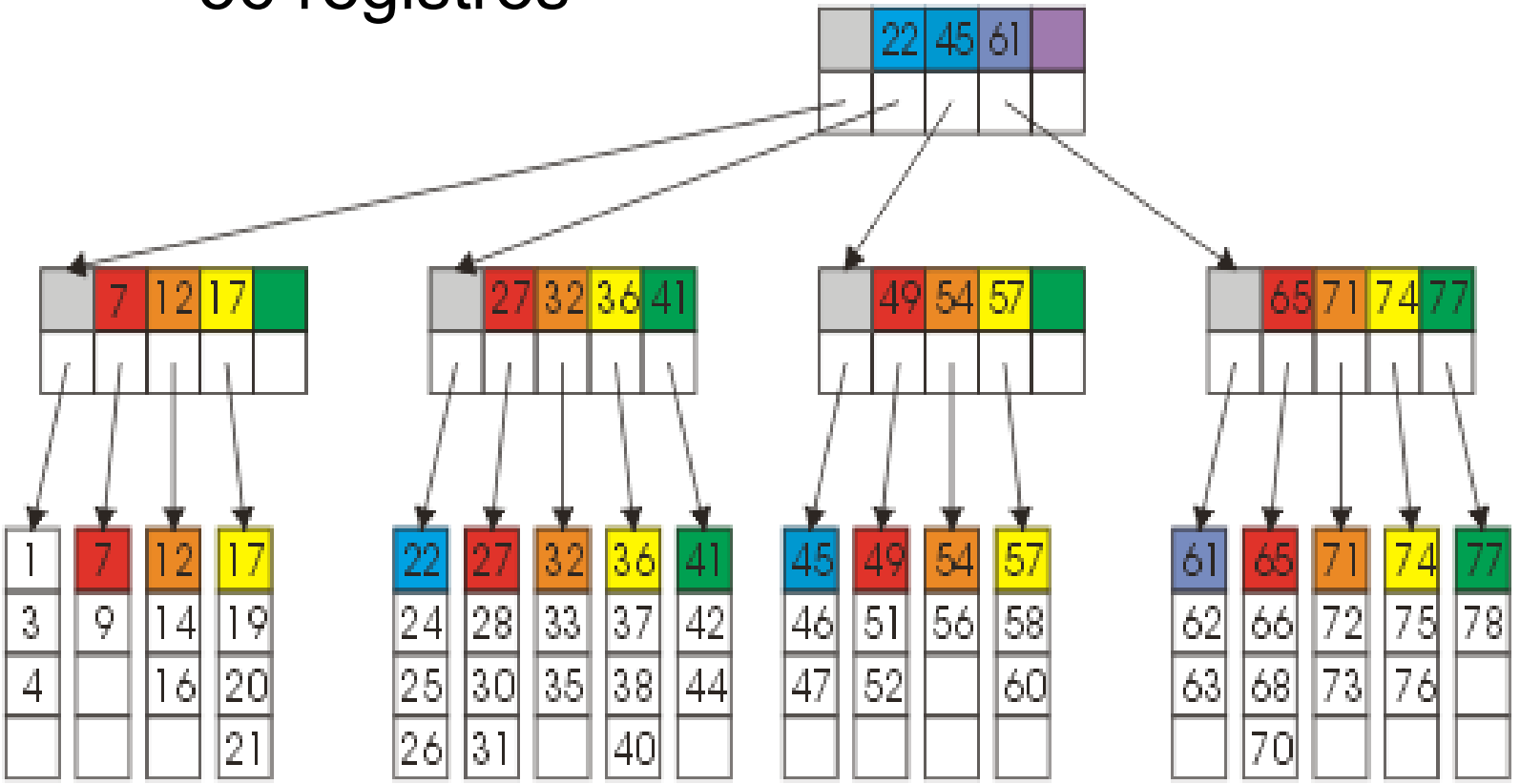
- Los nodos hojas utilizan arreglos para guardar datos.
- El costo de manipular estos arreglos es O(L) , sin embargo es ínfimo comparado con el acceso a disco.

B-Arbol - ejemplo

- Considere el siguiente B-Arbol
  - M=5 L=4
  - 56 registros

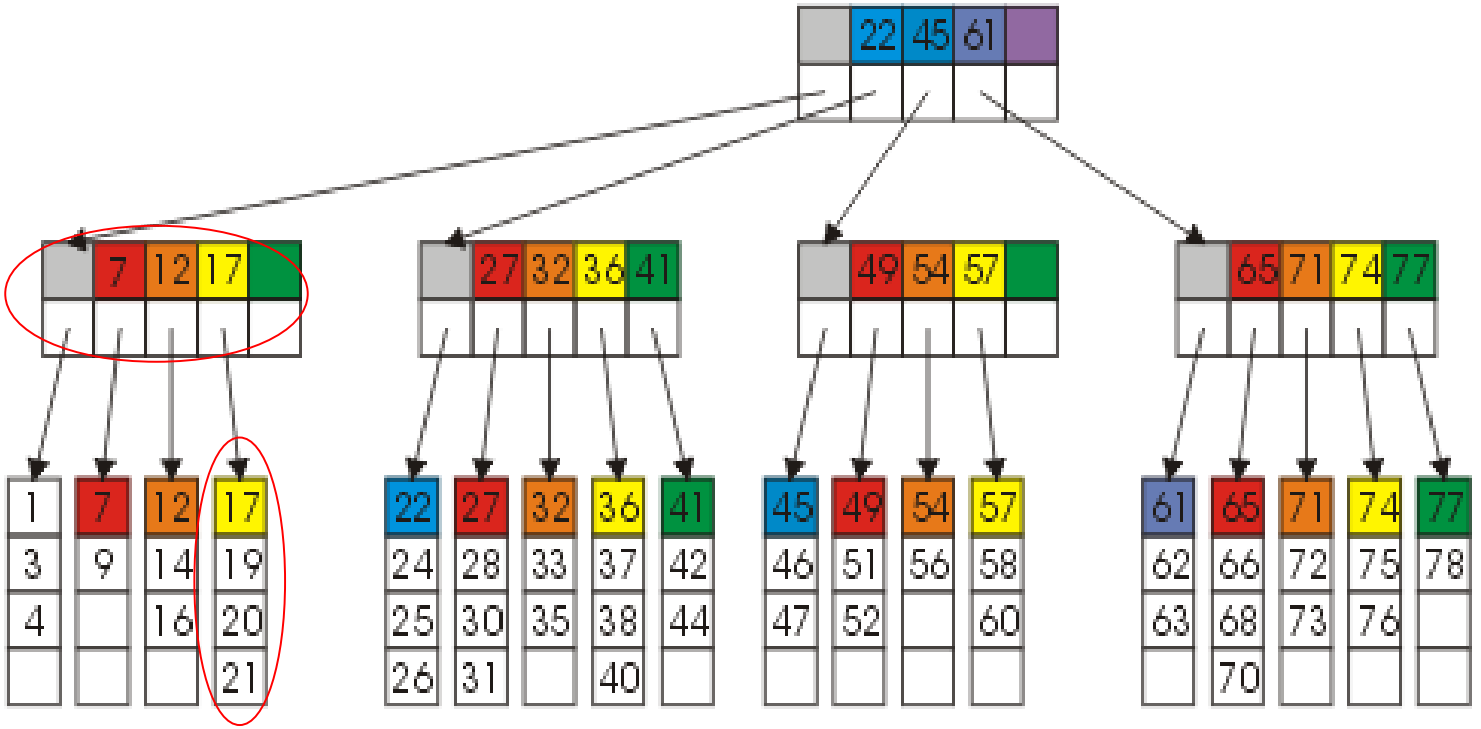
5-way tree nodes: 10 × 4 B = 40 B

leaf nodes: 4 × (4B + 6B) = 40 B



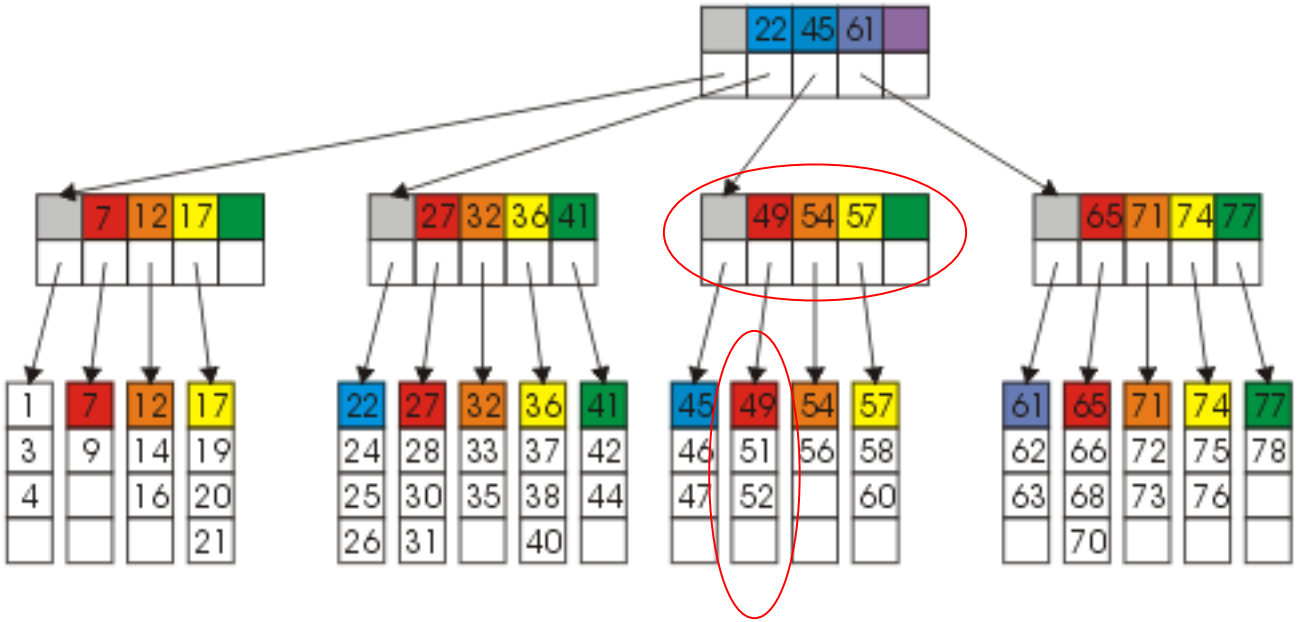
B-Arbol

- Buscamos el 19
  - 19 < 22 implica que el 19 esta en el primer sub-arbol
  - 19 > 17 implica que el 19 esta en el último sub-arbol
  - Un búsqueda binaria lo encuentra



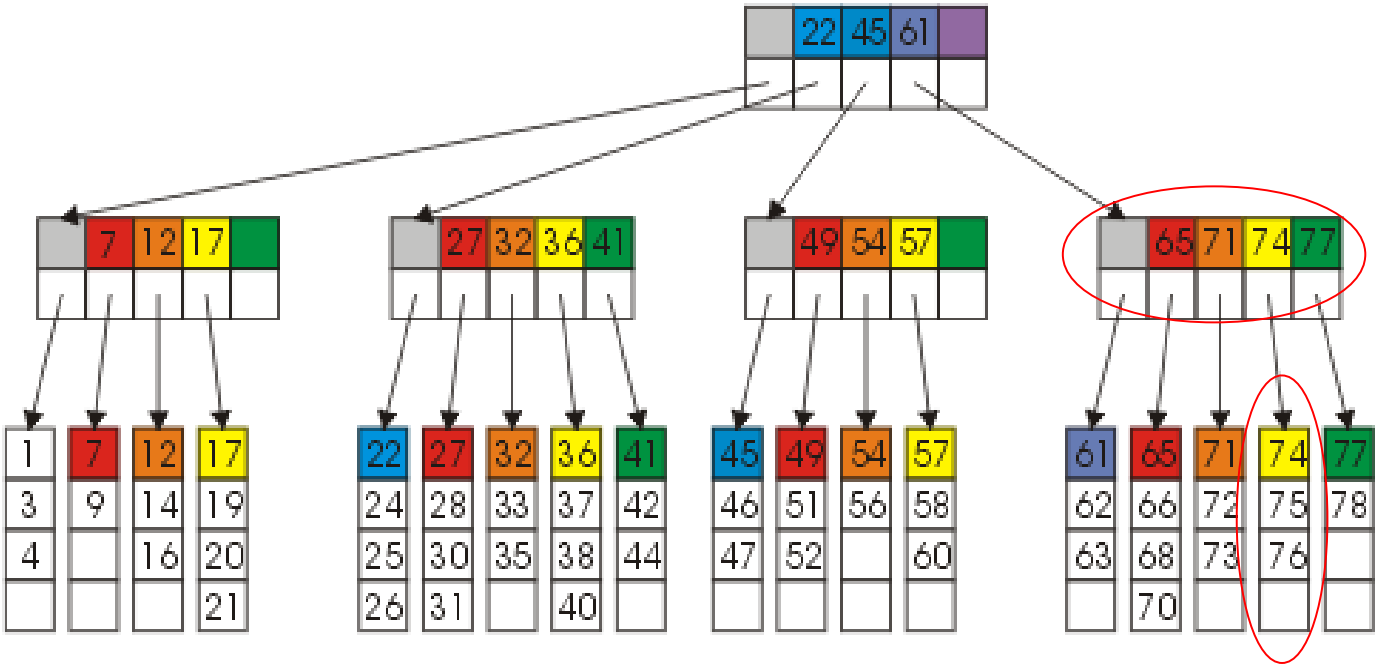
B-Arbol

- Buscamos el 50
  - 45 < 50 < 61 implica que el 50 esta en el 3er sub-arbol
  - 49 < 50 < 54 implica que el 50 esta en el 2do sub-arbol
  - Una búsqueda binaria no lo encuentra, por tanto no esta en el árbol



B-Arbol

- Buscamos el 80
  - 80 > 61 implica que esta en el último sub-arbol
  - 80 > 77 implica que esta en el último sub-arbol
  - Una búsqueda binaria no lo encuentra y por tanto no esta en el árbol



B-Arbol

- **Búsqueda** (resumen):
  - El nodo raíz siempre esta en memoria principal
  - Cada búsqueda requiere que dos bloques sean leídos a memoria
  - Si el dato asociado se encuentra, y se requiere modificarlo, solo un bloque debe ser guardado
  - Tiempo total: 20 ms (o 30 ms si se quiere grabar)

B-Arbol

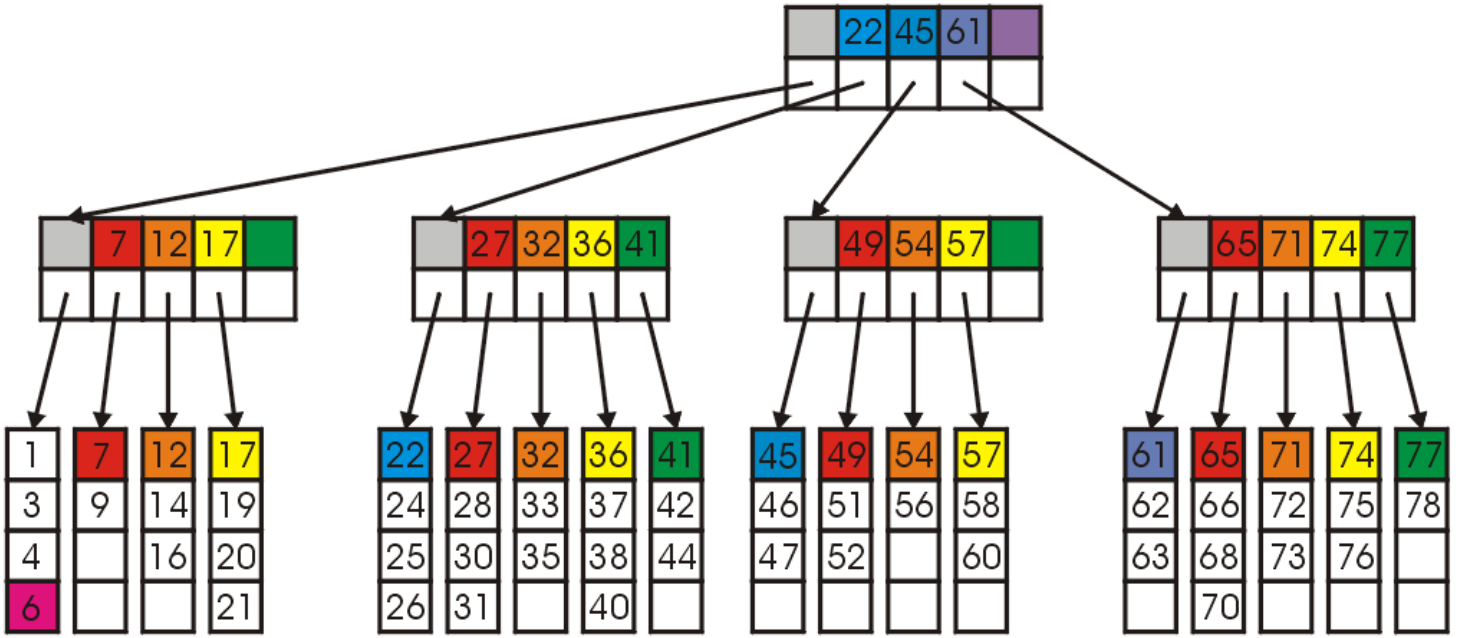
- **Inserción**:
  - Ocurre en los nodos hojas
  - Puede causar que el nodo hoja tenga más de L entradas, entonces implica:
    - Partir el nodo hoja en dos bloques de hojas
    - Actualizar el padre

B-Arbol

- Incrementar las entradas en el padre puede causar que tenga más de M entradas, implica:
  - Partir en dos
  - Actualizar su padre
  - Hacer recursivamente hasta la raíz, si es necesario.

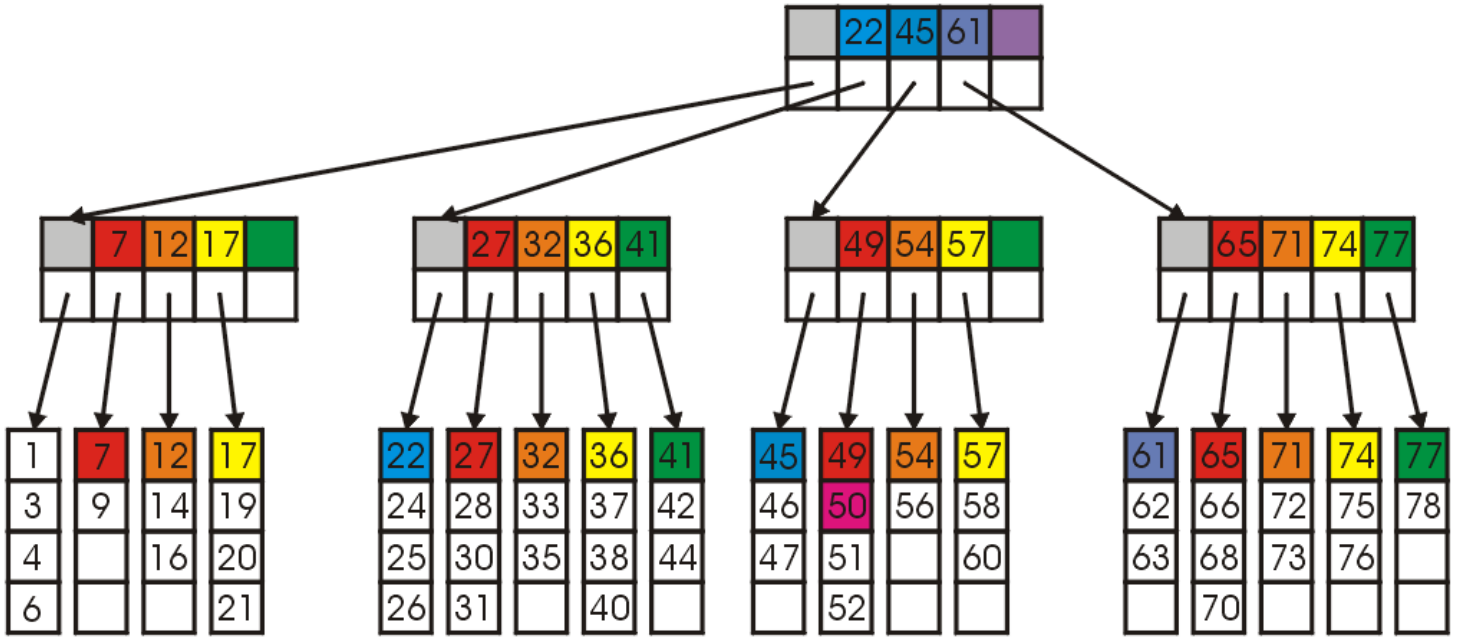
B-Arbol

- **Inserción** (ejemplo)
  - Insertar 6 requiere solo agregarlo en el primer bloque hoja



B-Arbol

- Insertar 50 requiere agregarlo en el arreglo

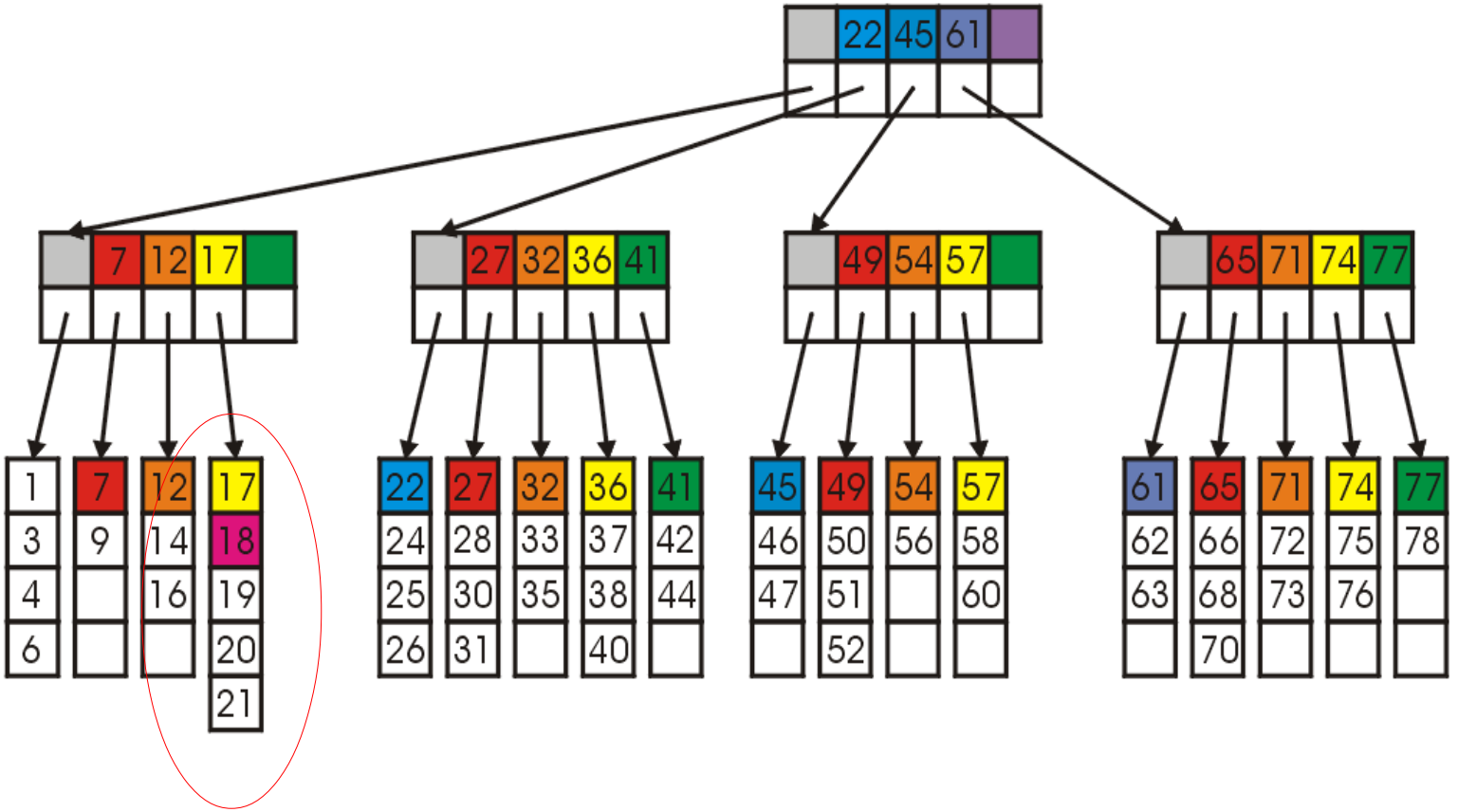


B-Arbol

- Inserción simple
  - Requiere dos accesos
  - Si el nodo hoja no esta lleno,
    - Requiere un tiempo O(L)
  - El nodo hoja debe guardarse
  - Tiempo total = 20 ms + 10 ms = 30 ms

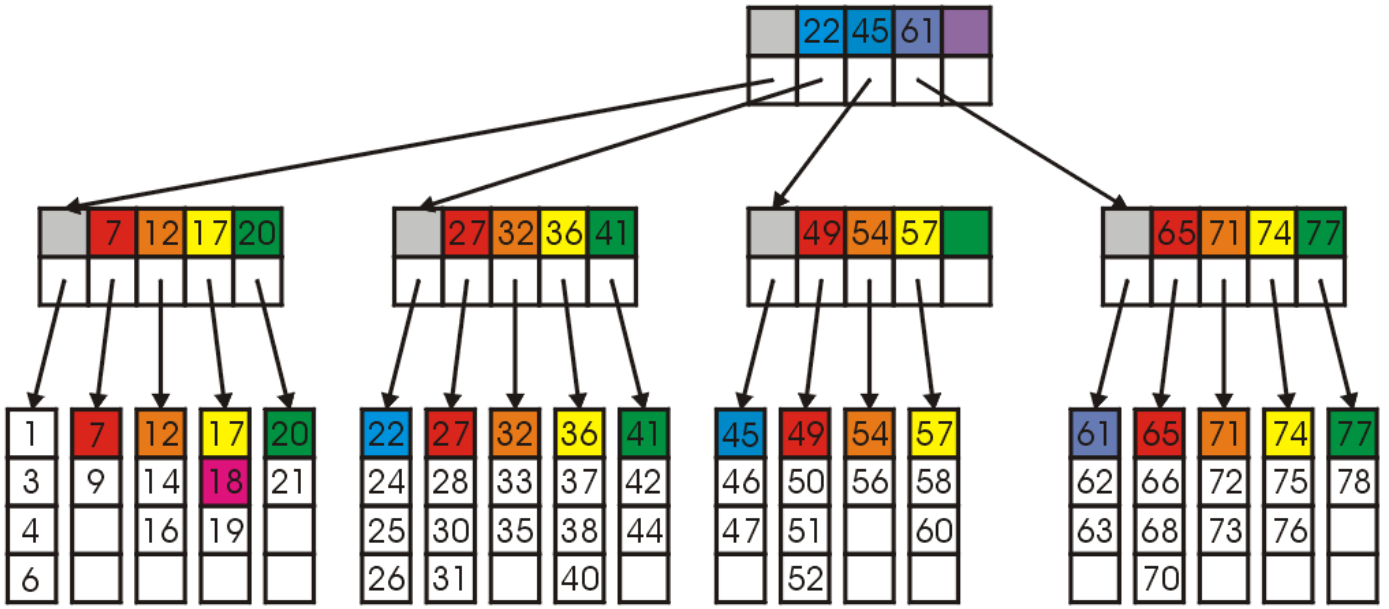
B-Arbol

- Insertar 18, el bloque hoja esta lleno, debemos partir.



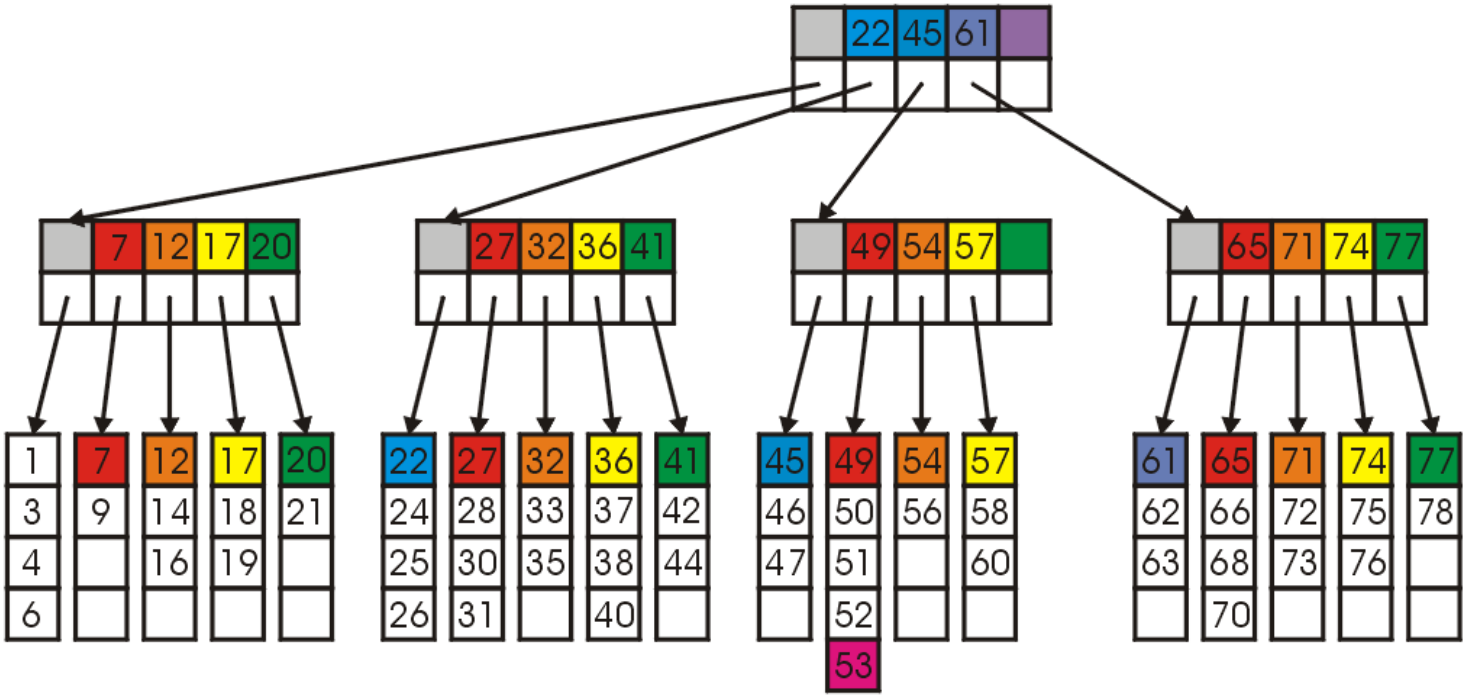
B-Arbol

- Actualizar el bloque hoja
- Actualizar el padre
- Grabar el nuevo bloque



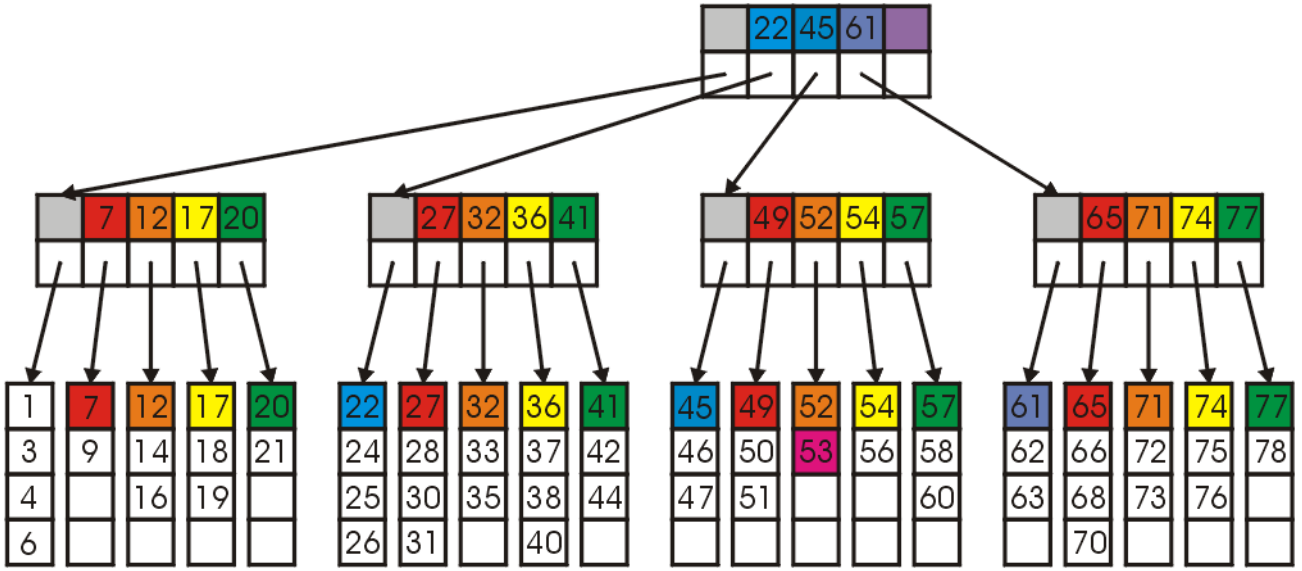
B-Arbol

- Insertar el 53
- Bloque hoja lleno, debemos partir



B-Arbol

- Actualizar el bloque hoja
- Actualizar el padre
- Grabar el nuevo bloque hoja

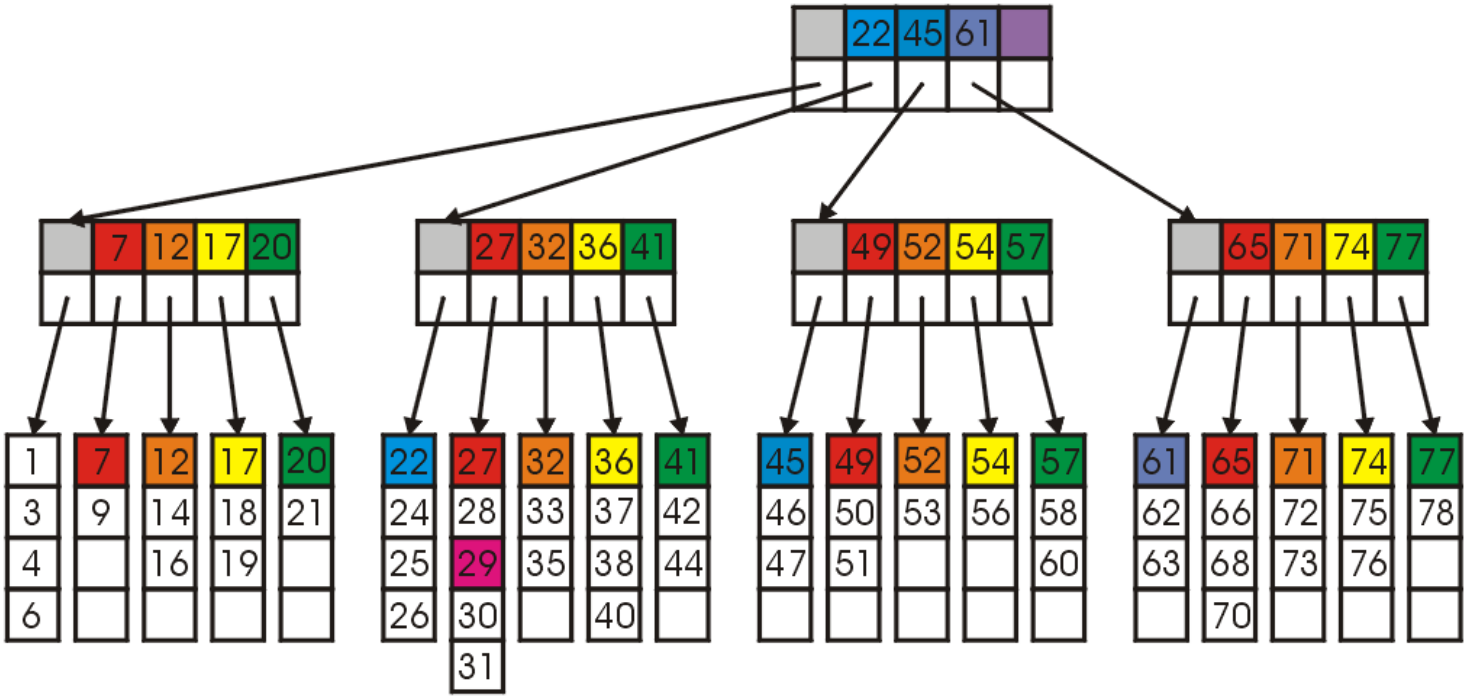


B-Arbol

- Un nivel de inserción, requiere:
  - 2 accesos
  - Una inserción O(L) en el bloque hoja
  - Una copia O(L)
- Cada nodo hoja tiene >= L/2 entradas
- Grabar el nodo original, el nuevo nodo y el padre modificado
- Tiempo total : 20 ms + 30 ms = 50 ms

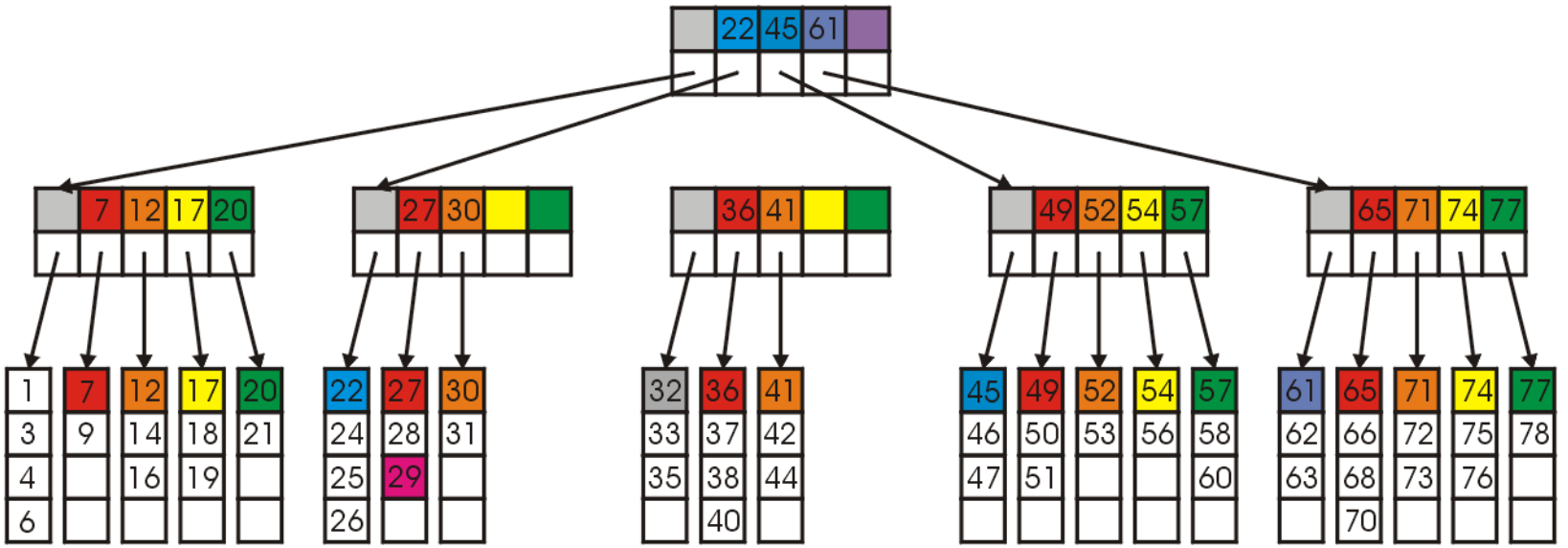
B-Arbol

- Dos niveles de inserción
  - Insertar el 29
  - Nodo hoja lleno, debemos partir



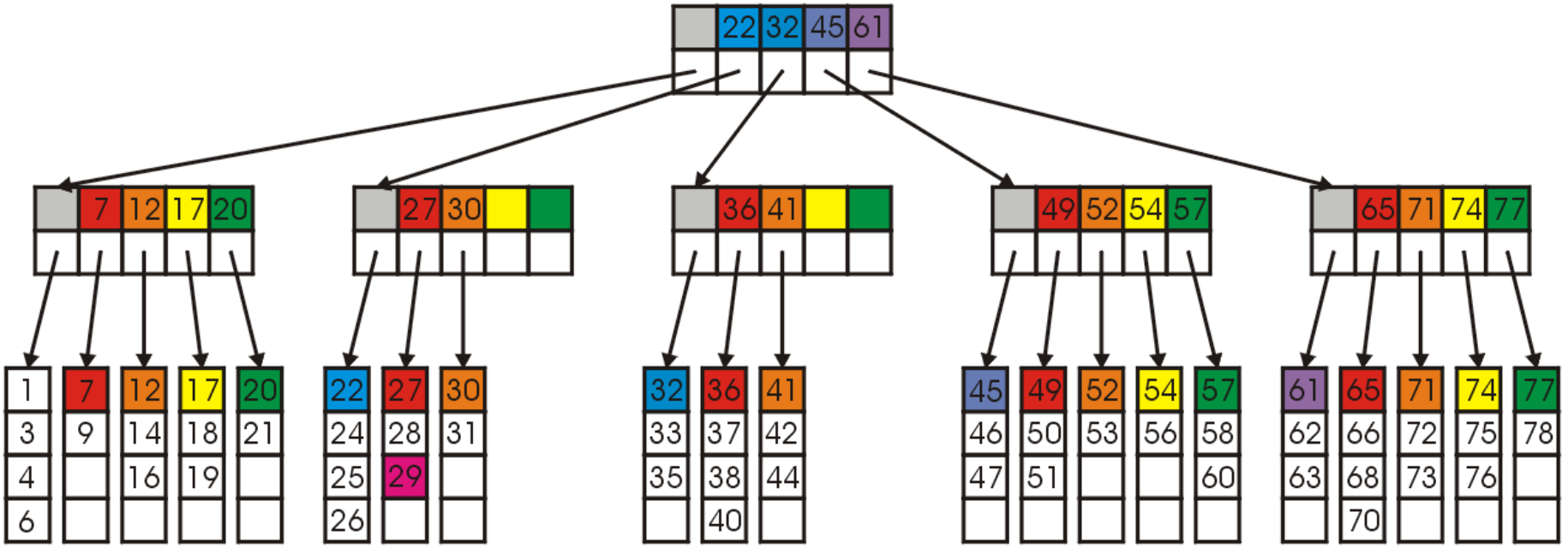
B-Arbol

- Nodo padre lleno, debemos partirlo



B-Arbol

- La raíz debe ser actualizada

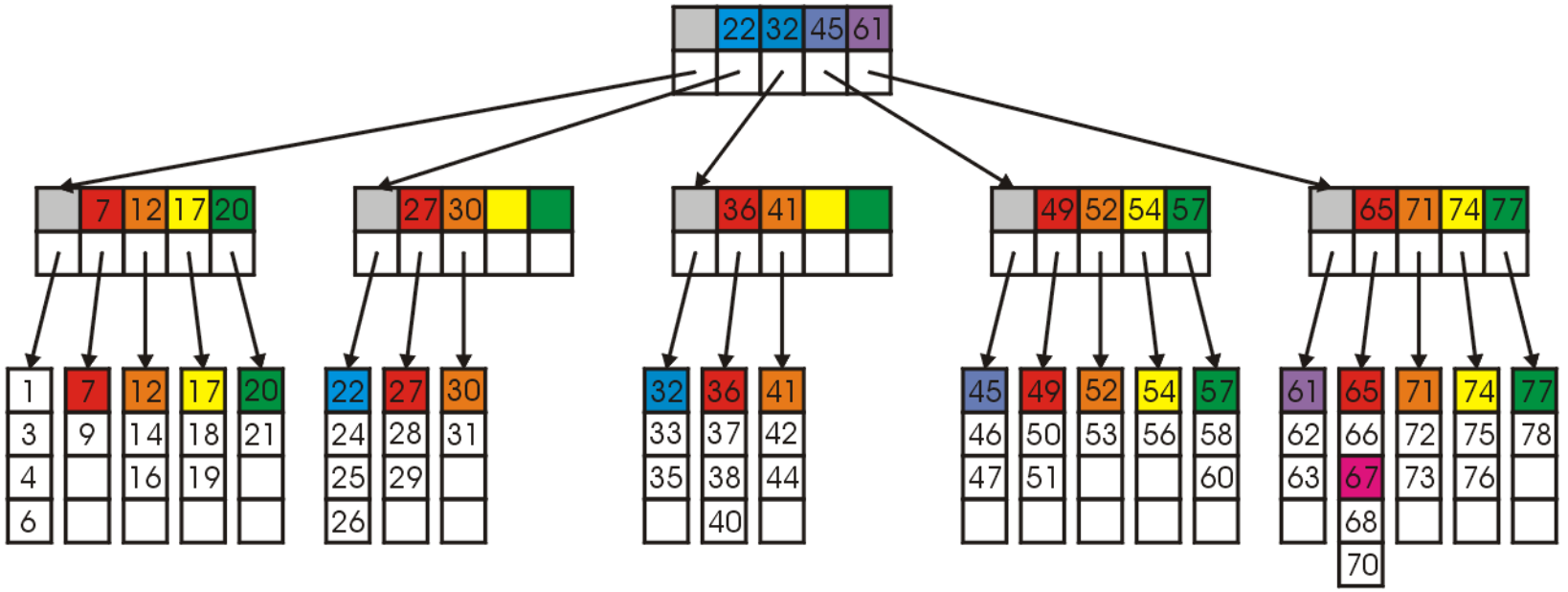


B-Arbol

- Se requiere
  - 2 accesos
  - O(L) inserción
  - O(L) copia
- Cada bloque interno tiene >= M/2 entradas
- Debemos guardar ambas hojas, ambos nodos internos y la raiz
- Total de tiempo = 20 ms + 50 ms = 70 ms

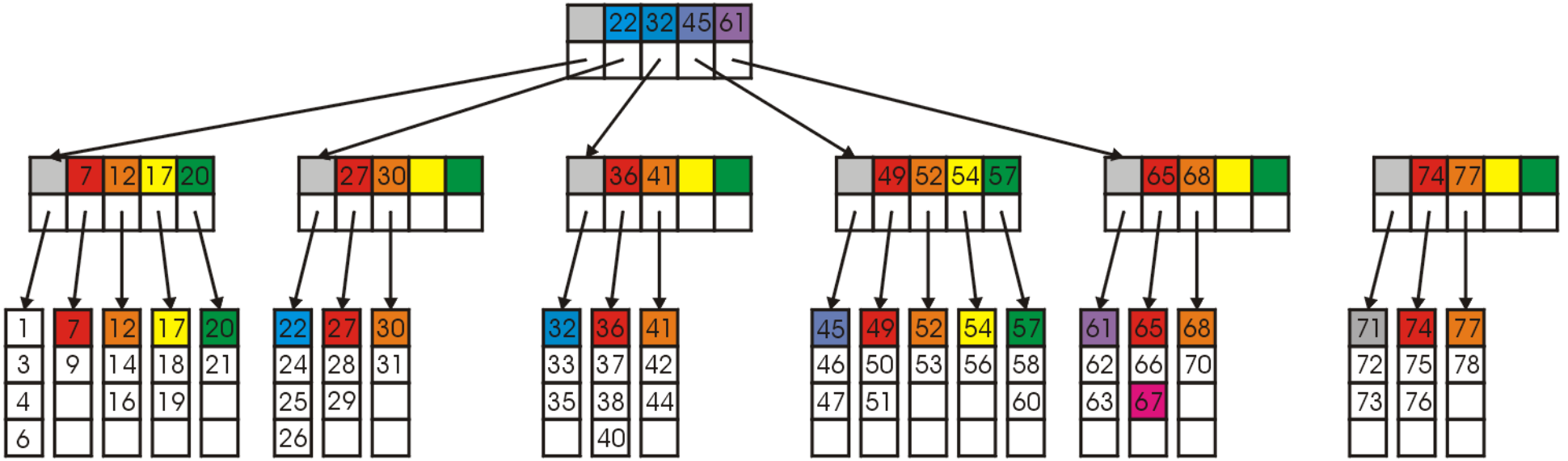
B-Arbol

- Insertar 67
- La hoja esta llena, partimos en dos



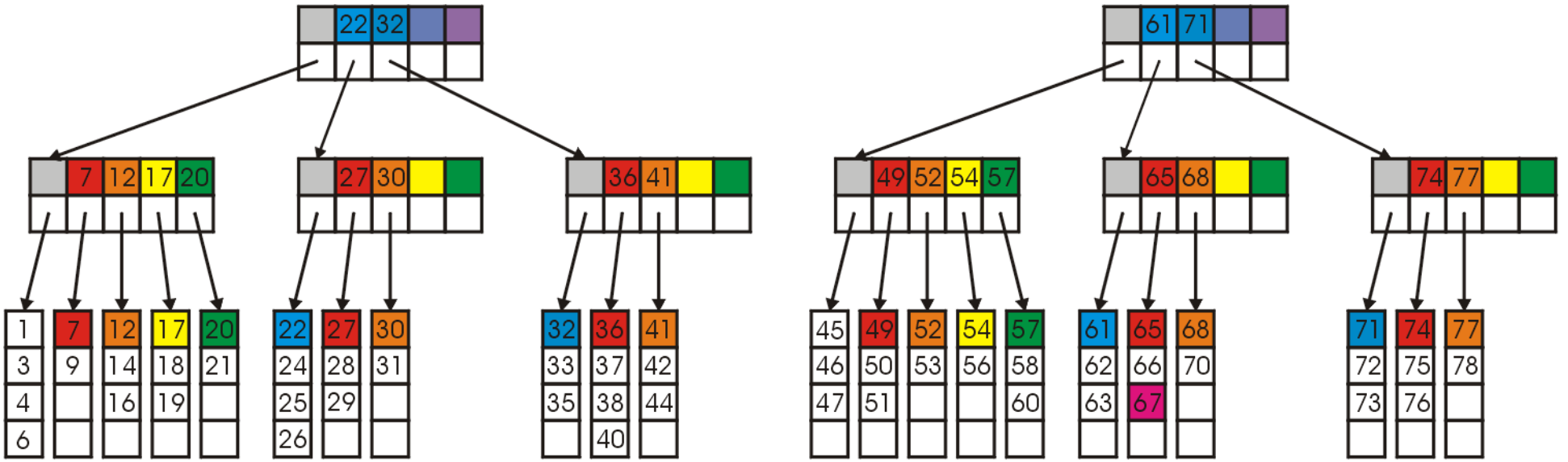
B-Arbol

- Padre esta lleno, lo partimos en dos



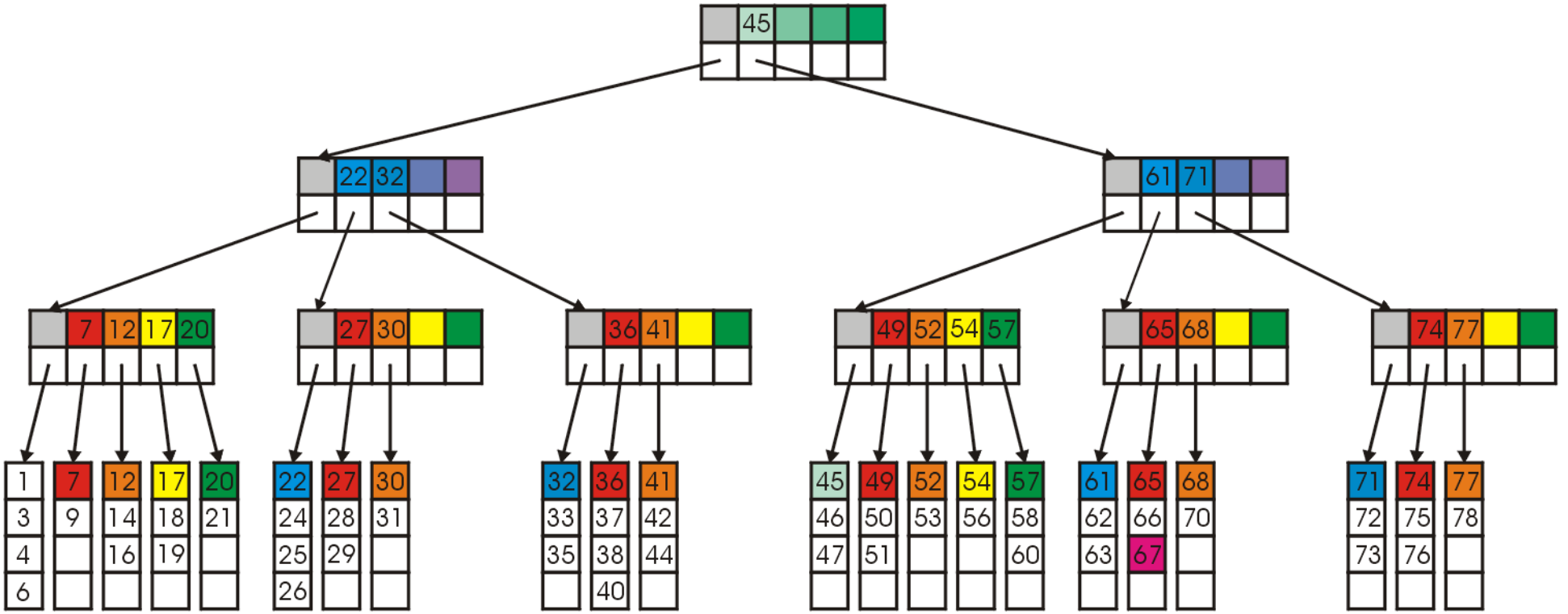
B-Arbol

- La raíz esta llena, partimos en dos



B-Arbol

- Creamos una nueva raíz (aumenta la altura del árbol)



B-Arbol

- Se requiere
  - 3 accesos
  - O(L) inserción
  - O(L) copia
- Cada nodo interno tiene >= M/2 entradas
- Nuevo bloque raíz tiene dos sub-arboles
- Debemos grabar 7 bloques (4 nuevos)
- Total tiempo = 30 ms + 70 ms = 100 ms

B-Arbol

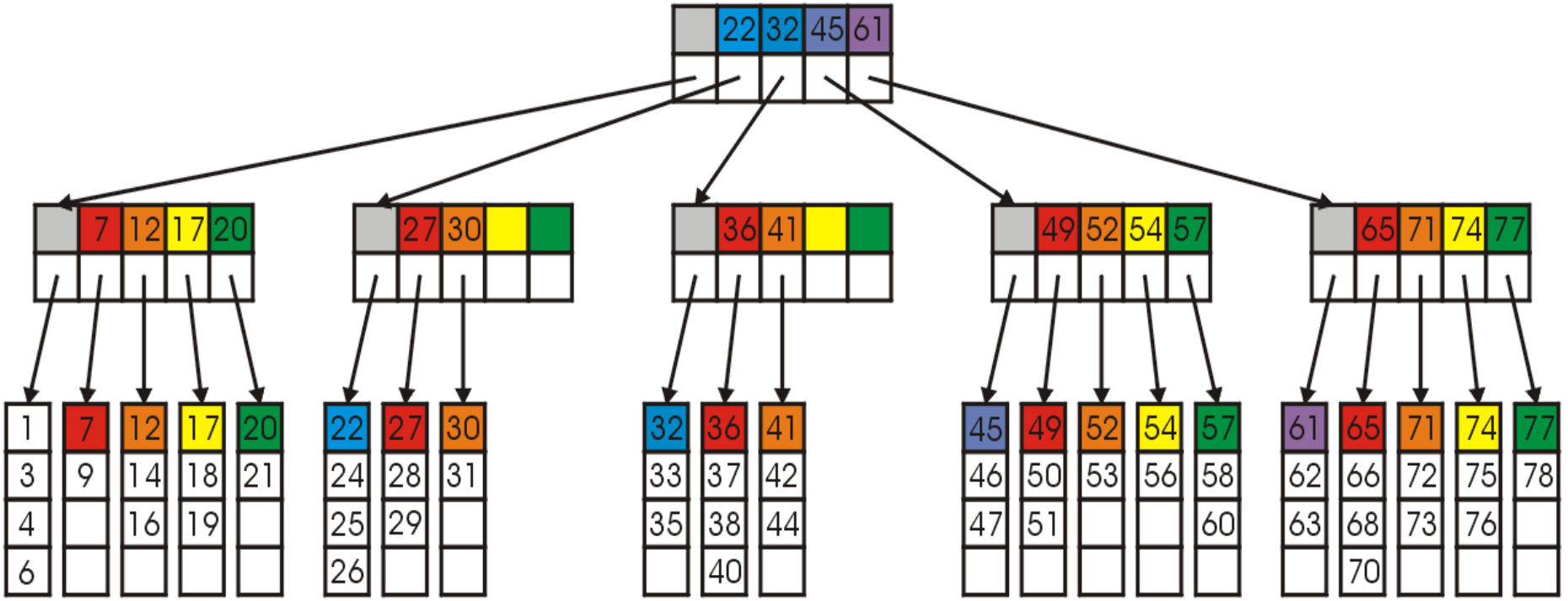
- **Eliminación**
  - Puede ocurrir que la condición en el nodo hoja sea violada < L/2 datos
  - Tomar la hoja de un vecino
  - Dos posible casos:
    - Juntos tienen <= L nodos : *mezclarlos.*
    - Juntos tienen > L nodos: *redistribuirlos.*

B-Arbol

- **Eliminación**:
  - Juntando dos nodos puede causar en el padre una violación a la condición:
    - Si contiene < M/2 nodos
  - Tomar un nodo interno del vecino
  - Dos posibles casos:
    - Juntos tienen <= M nodos : se mezclan
    - Juntos tienen > M nodos: se redistribuyen
  - Repetir lo mismo hacia la raíz, si es necesario

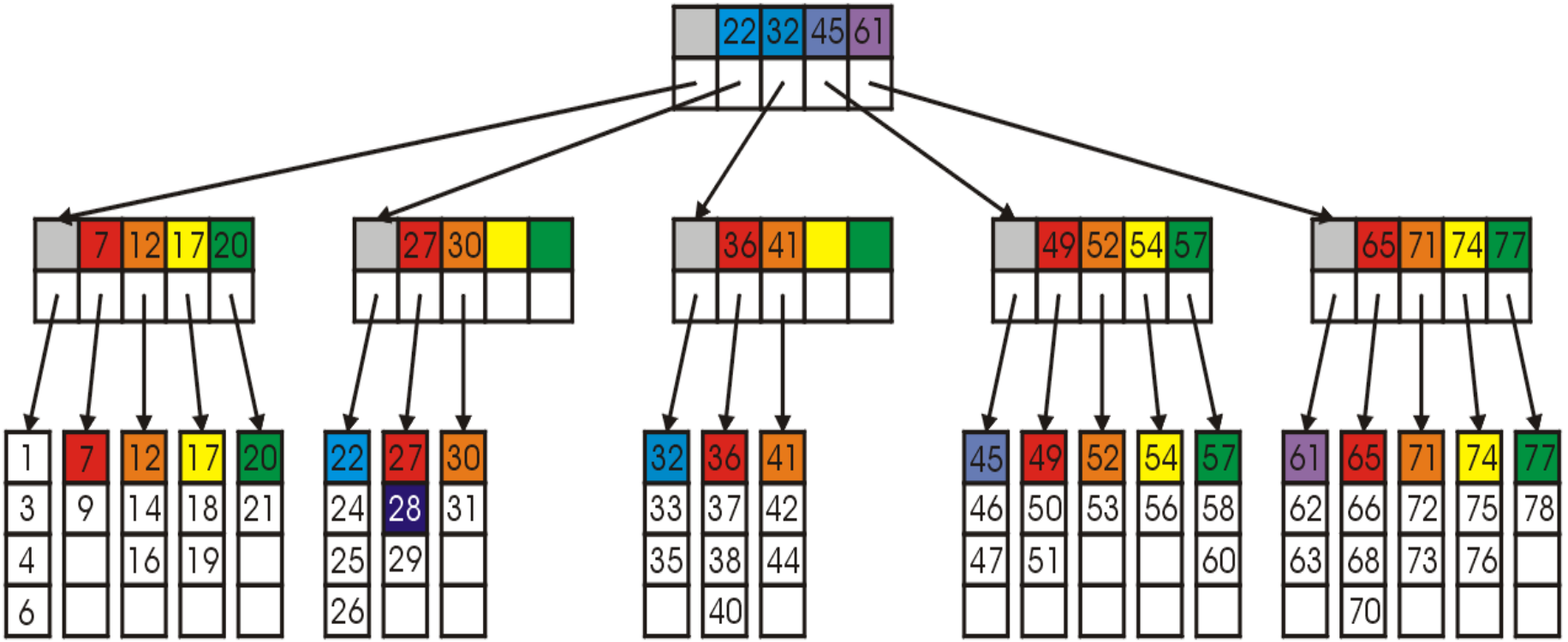
B-Arbol

- Eliminación (ejemplo)



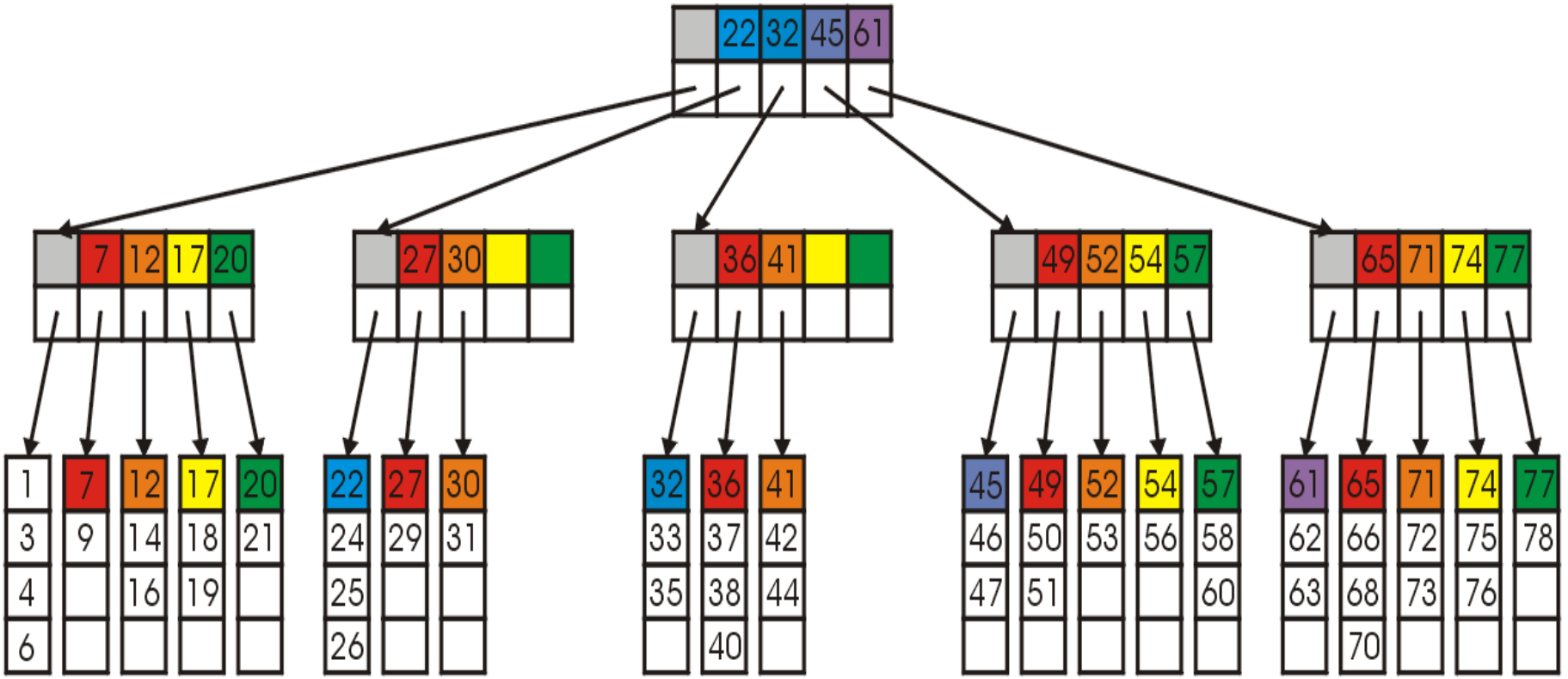
B-Arbol

- Eliminamos el 28
- El resto del bloque hoja tiene >= L/2 entradas



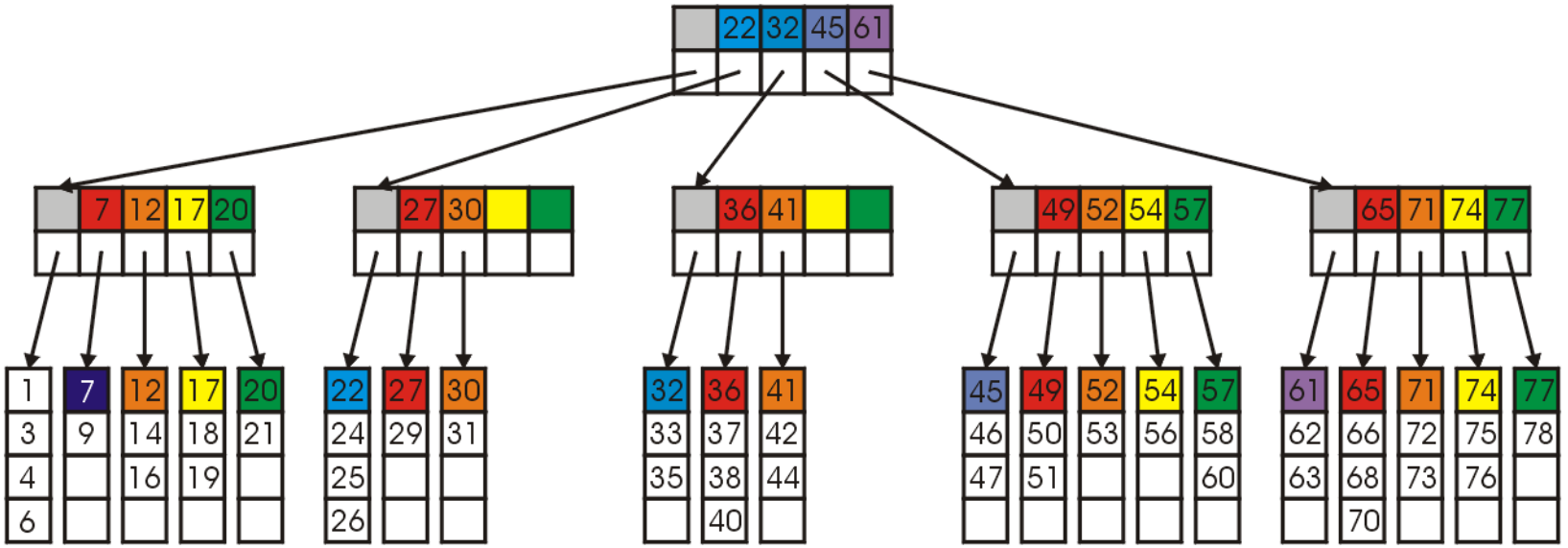
B-Arbol

- Por tanto, removemos el 28 del bloque hoja



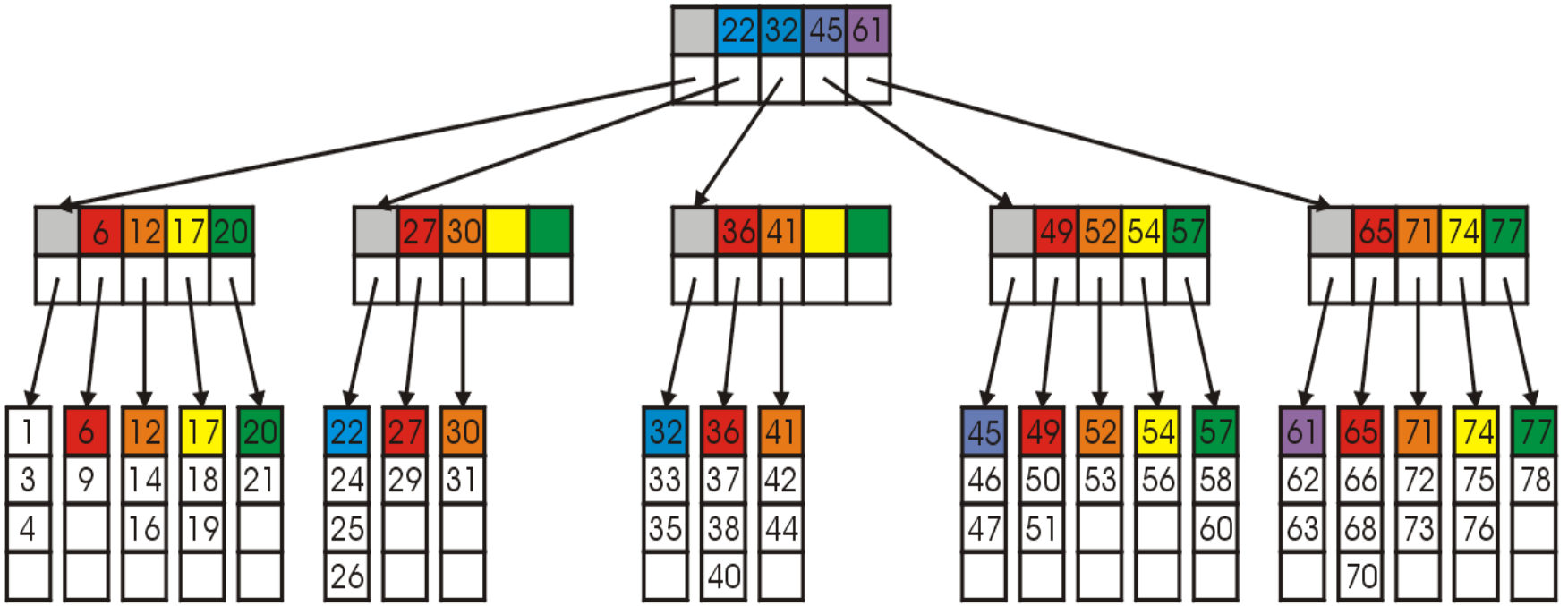
B-Arbol

- Removemos el 7
- El resto de su bloque tiene < L/2 entradas
- El bloque izq. tiene 4 entradas (redistribuimos)
- El bloque der tiene 3 entradas (mezclamos)



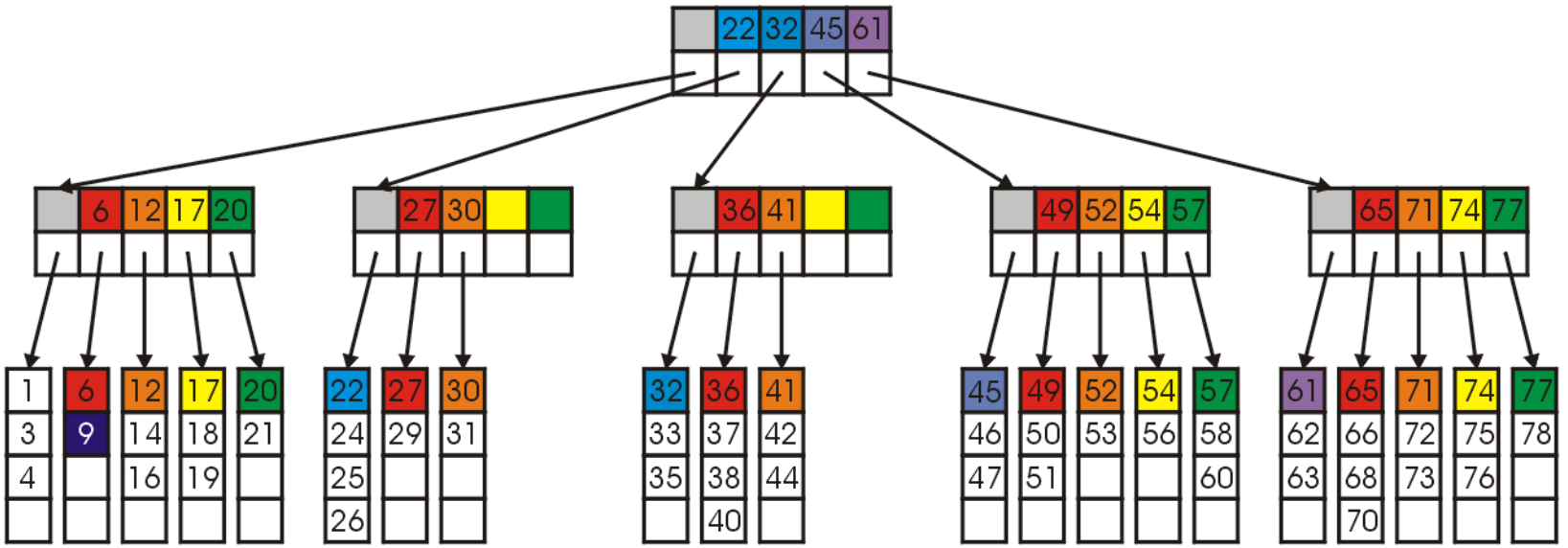
B-Arbol

- Redistribuimos (optamos por)
- Necesitamos actualizar el padre



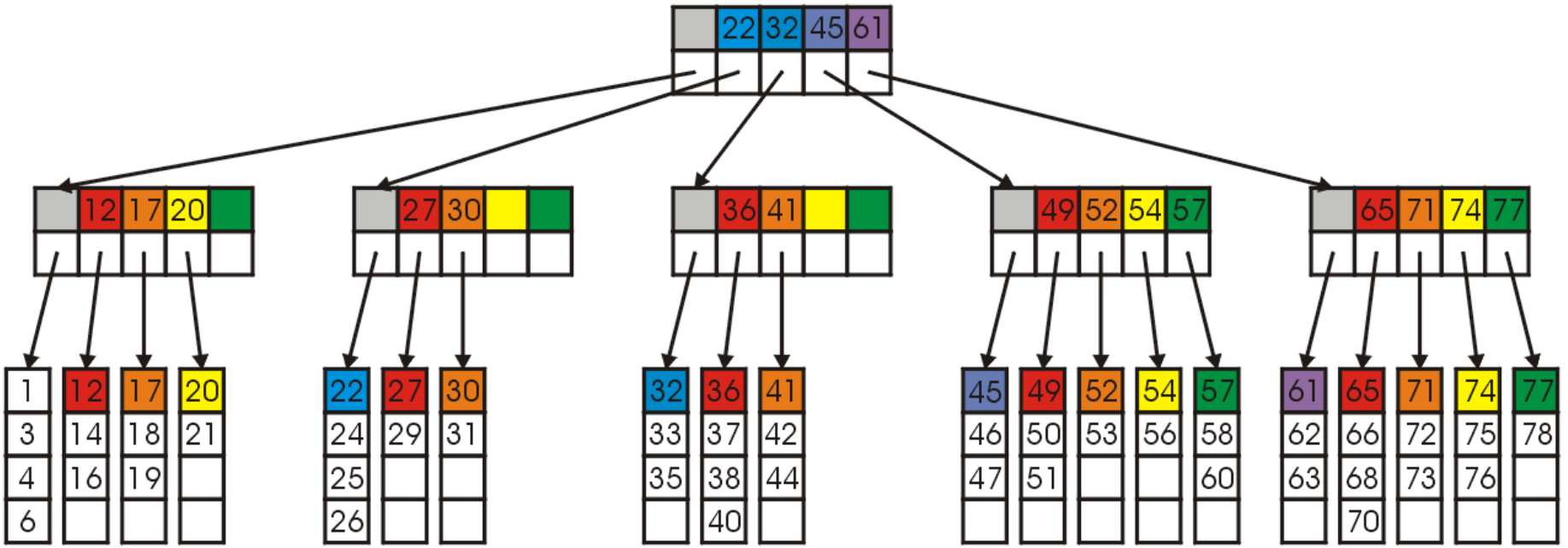
B-Arbol

- Removemos el 9
- El izq. y el der. tienen 3 entradas
- 3+1 <= L y por tanto nosotros mezclamos



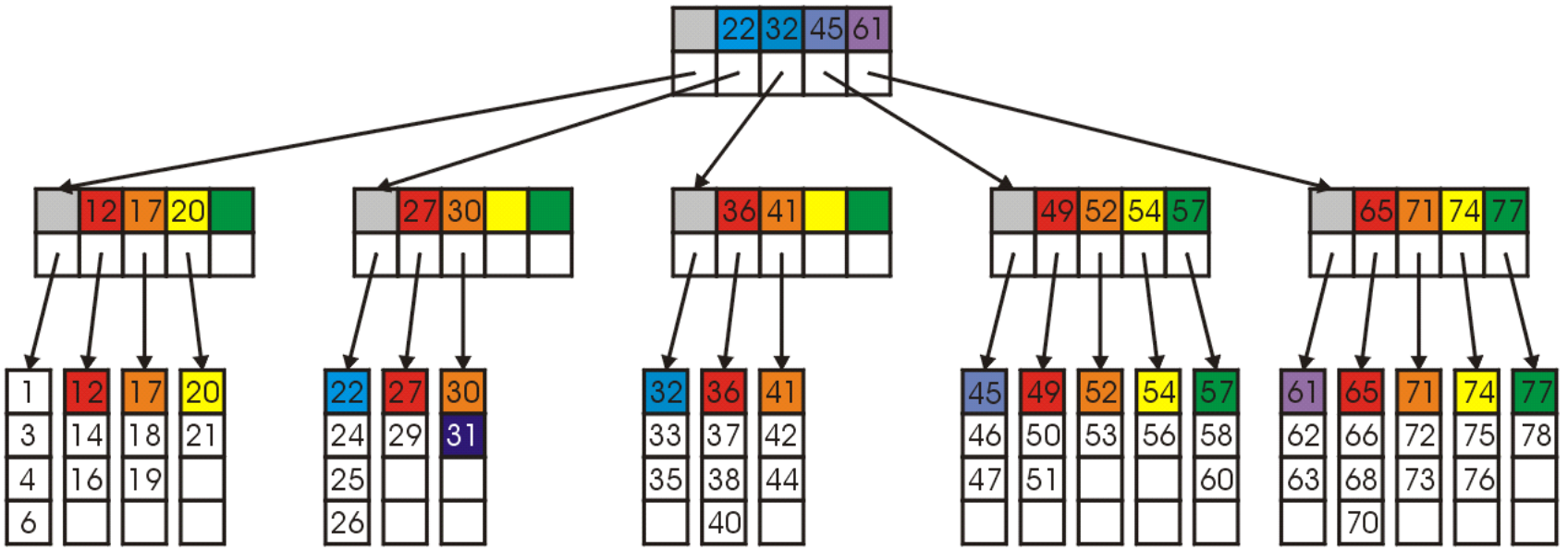
B-Arbol

- Necesitamos mezclar con el nodo izq.
- Y actualizar el padre



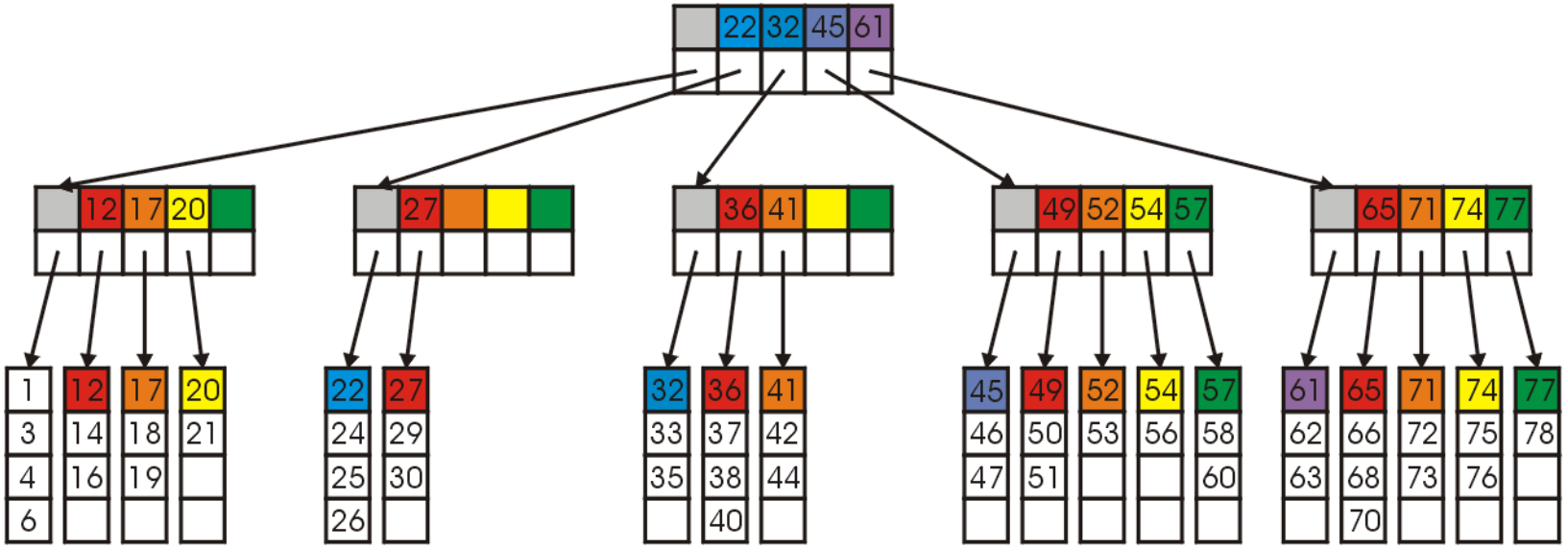
B-Arbol

- Remover el 31
  - El resto tiene menos de L/2 entradas
  - Solo puedo acceder al nodo izq.
  - Debemos mezclar: 2+1 < L



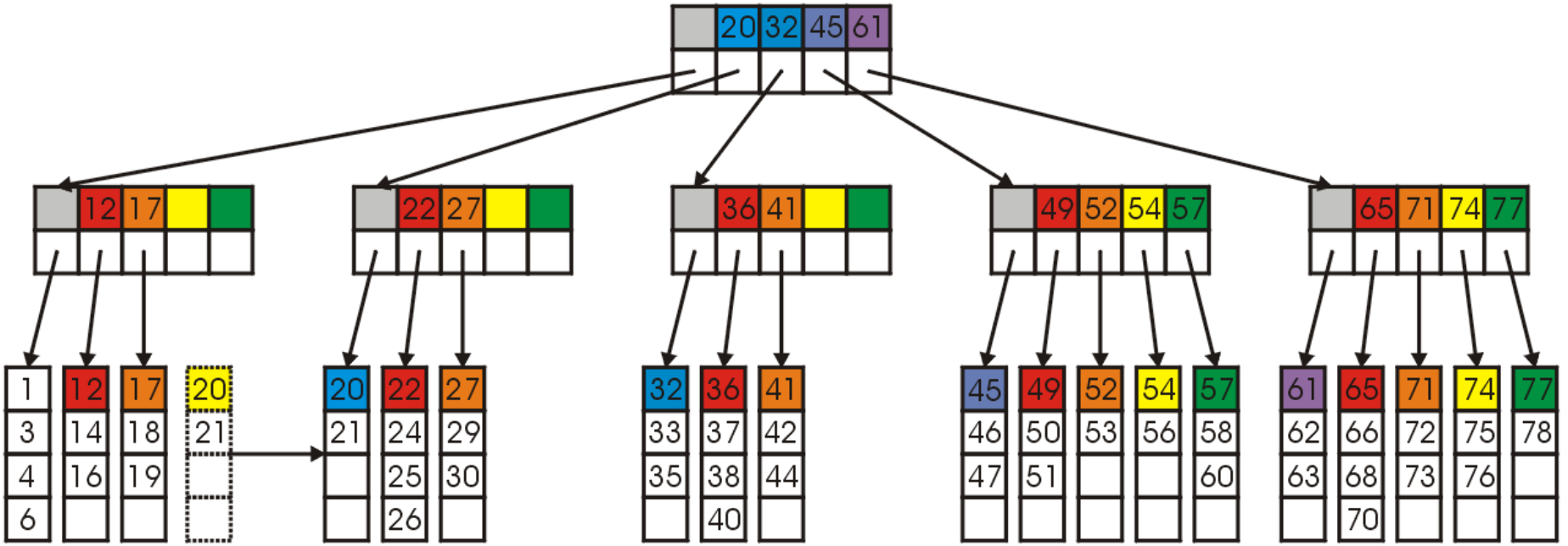
B-Arbol

- Debemos actualizar el padre
- Ahora el padre tiene < M/2 hijos
  - Dos opciones:
    - Redistribuir con el bloque izquierdo
    - Mezclar con el derecho



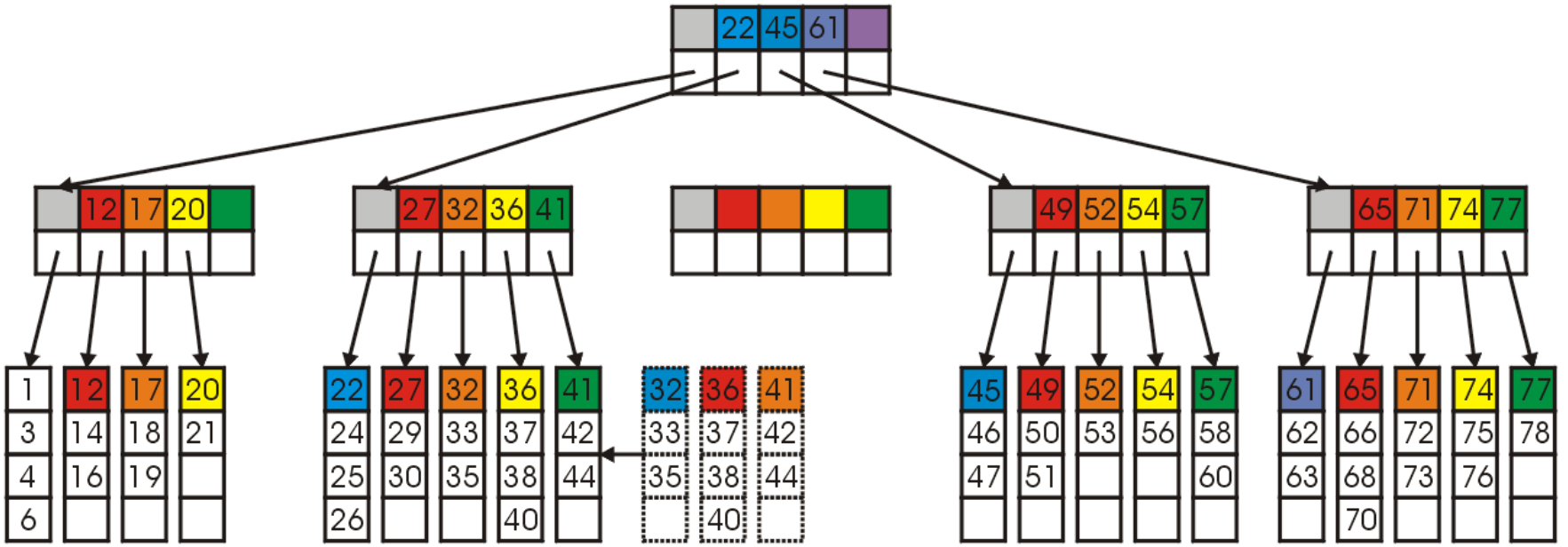
B-Arbol

- Si redistribuimos con el bloque izquierdo, debemos:
  - Actualizar ambos padres
  - Actualizar la raíz



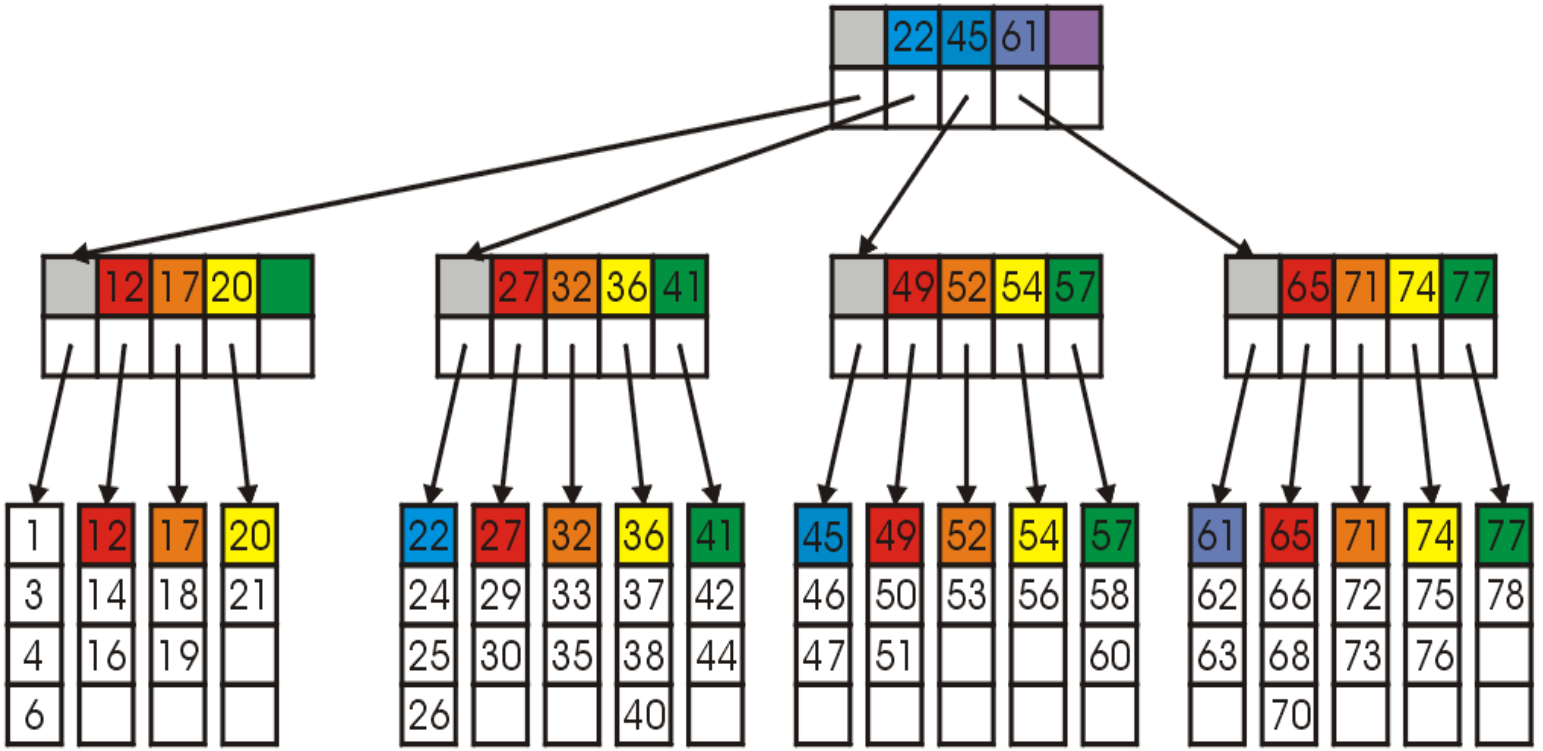
B-Arbol

- Si mezclamos con el bloque derecho, debemos:
  - Actualizar el padre
  - Borrar el otro bloque
  - Actualizar la raíz



B-Arbol

- Entonces, el número de hijos de la raíz disminuye.



B-Arbol

- El B-Arbol original guarda información en todos los nodos.
- Los B-Arboles vistos aquí se llaman ocasionalmente B+Arbol (B+Tree)
  - El requerimiento de que se llene la mitad puede tornarse más estricto y requerir que se llene 2/3, de modo a aprovechar mejor el espacio:
    - Se llaman B*Arbol o B*Tree
    - Requieren redistribuciones y mezclas más complejas

Ejercicios sobre ordenación externa (en PDF)

Referencias

- Estructura de datos en Java, Mark A. Weiss. 2013. (cap. 19).
- Data Structures and Algorithm Analysis, Clifford Shaffer. 2013 (cap. 8).
- T. Cormen, C. Leiserson, R. Rivest y C. Stein. Introduction to algorithms. 3rd. edition. MIT Press. 2009. (cap.18).
- Para una discusión sobre los tipos de B-Arbol puede leer
  - D. Comer. "*The Ubiquitous B-tree*", Computing Surveys 11 (1979), 121-137.

Tarea #7

- **Ejercicio 1**:
  - Implementación de mergeSort externo en Java
  - Evaluación de rendimiento
- **Ejercicio 2 :** dado el código BArbol
  - Convertir a Generic
  - Cambiar orden de ramificación
  - Pruebas de rendimiento, comparación con BST y AVL considerando penalización al acceso de un nodo.
  - Implementación de *between* y *eliminar*