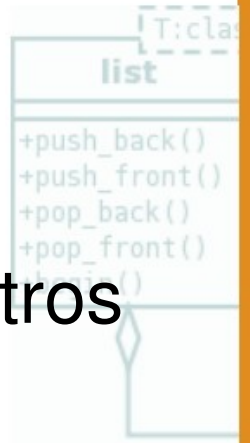
Tema 6: Árboles B y B+

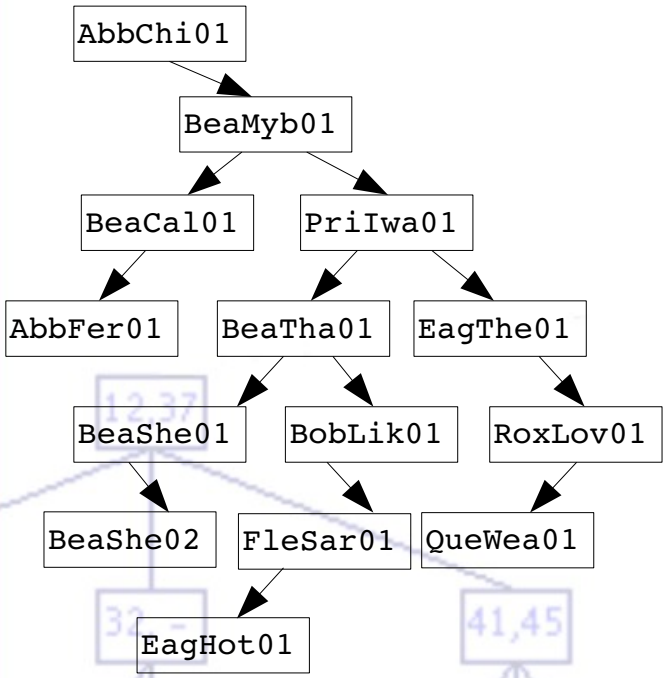
1. Gestión mediante índices de grandes volúmenes de información
2. Árboles B
3. Acceso secuencial indexado: Árboles B+

Gestión mediante índices de grandes volúmenes de información



- Los índices simples proporcionan un método sencillo y eficiente para manejar ficheros de registros
- Pero ¿Que ocurre si el volúmen del fichero es tal que los índices no pueden mantenerse en memoria?
- Mantener un índice simple en disco, aunque posible, es extremadamente ineficiente
  - Las inserciones deben ser ordenadas, y por tanto implican abrir huecos y desplazar registros
  - Los borrados implican eliminar huecos (en el fichero índice no puede utilizarse el borrado lógico)
  - Las búsquedas binarias requieren demasiados accesos. Buscar un registro entre 1000000 registros → ≃20 accesos

- Una posible alternativa consiste ordenar lógicamente los registros siguiendo un esquema de árbol binario de búsqueda

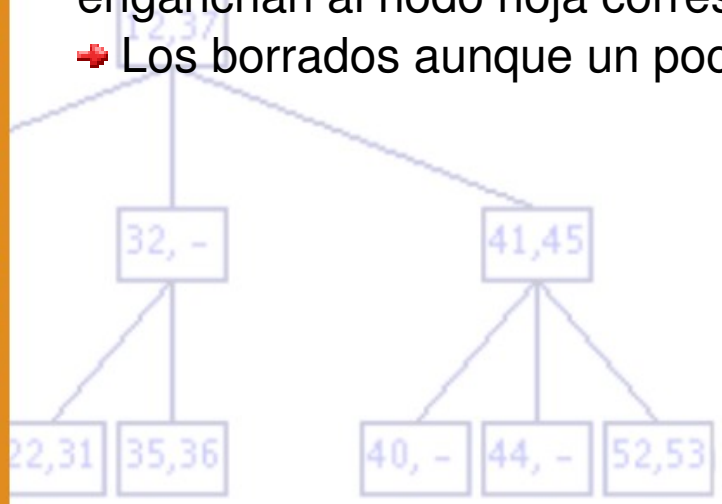


Representación lógica

	Izq	Der	Clave	Resto de campos...
0	-1	34	AbbChi01	Chiquitita|Abba|1979|3:34
34	68	102	BeaMyb01	My Bonnie|Beatles|1964|3:45
68	170	-1	BeaCal01	California Girls|Beach Boys|1965|1:27
102	204	136	PriIwa01	I Wanna Be Your Lover|Prince|1979|1:40
136	-1	306	EagThe01	The Long Run|Eagles|1979|1:15
170	-1	-1	AbbFer01	Fernando|Abba|1976|3:35
204	272	238	BeaTha01	Thank You Girl|Beatles|1964|2:07
238	272	340	BobLik01	Like A Rolling Stone|Bob Dylan|1965|1:28
272	-1	-1	BeaShe01	She Loves You|Beatles|1964|3:36
306	408	-1	RoxLov01	Love Is The Drug|Roxy Music|1976|3:03
340	374	-1	FleSar01	Sara|Fleetwood Mac|1979|2:13
374	-1	-1	EagHot01	Hotel California|Eagles|1977|3:28
408	-1	-1	QueWea01	We Are The Champions|Queen|1977|2:16
442	-1	-1	BeaShe02	She's A Woman|Beatles|1964|3:50

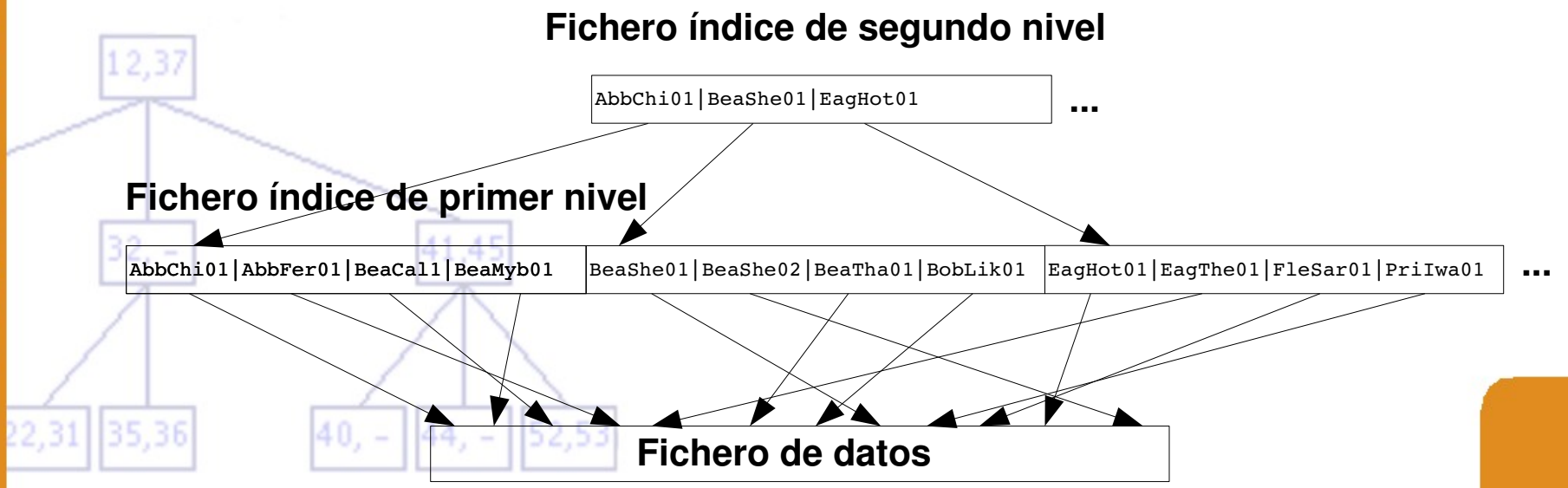
Representación física

- Una pequeña cabecera en el fichero indica la posición del nodo raiz del árbol
- Los nodos hoja son aquellos cuyos índices izquierda y derecha apuntan a -1
- Representación muy compacta: tanto los datos como los índices están almacenados en un único fichero
- Operaciones muy eficientes
  - Una búsqueda implica sólo O(log n) lecturas en el fichero
  - No es necesario ordenar físicamente los datos. Los datos se insertan al final y se enganchan al nodo hoja correspondiente en O(log n)
  - Los borrados aunque un poco más complejos, también requieren O(log n) lecturas



- Problema grave: desequilibrado del árbol tras varias inserciones y borrados
  - Solución posible: utilizar un árbol balanceado (p. e. AVL), aunque resolver las rotaciones en el fichero requiere nuevos accesos
- La búsqueda binaria no es suficientemente eficiente
- El uso del disco es muy deficiente
  - Cada acceso implica leer un bloque, y de este bloque únicamente se utiliza un registro, cuyos índices izquierda y derecha apuntan, con mucha probabilidad, a un registro de otro bloque

- Los índices multinivel hacen un uso del disco mucho más eficiente que los árboles binarios
  - Se define un registro de claves, de tamaño igual a un bloque de disco o un divisor de éste
  - El índice de primer nivel consiste en un fichero de registros de claves con apuntadores al fichero de datos
  - El índice de segundo nivel sirve para indexar el índices de primer nivel y tiene la misma estructura que éste. Cada entrada aquí incluye la clave de la primera entrada de cada registro del índice de primer nivel y un apuntador a éste
  - El índice de tercer nivel indexa al de segúndo nivel, etc.

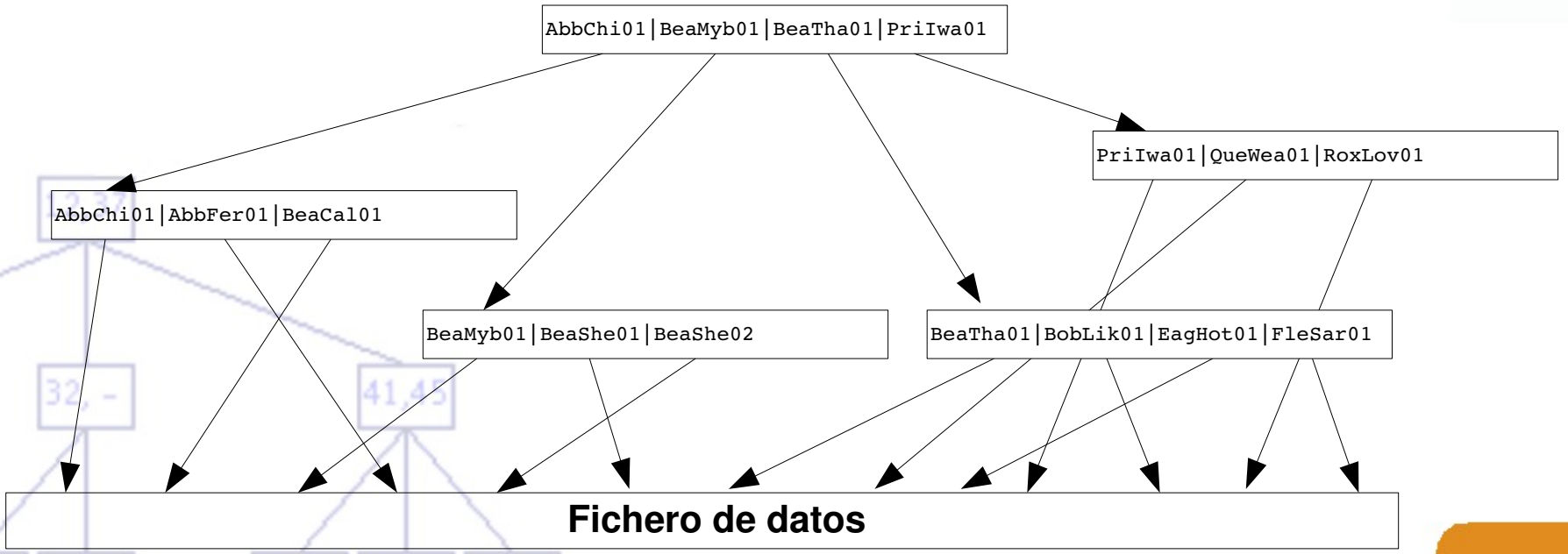


- La búsqueda en un índice multinivel es muy eficiente, ya que requiere un único acceso por cada nivel
  - Buscar la clave en el registro del índice de mayor nivel mediante búsqueda binaria
  - Obtenemos una entrada cuya clave puede ser igual o inmediantemente inferior a la buscada. Seguimos el apuntador al bloque del siguiente nivel y repetimos este proceso hasta llegar al fichero de datos
- Los índice multinivel son esencialmente una estructura de ficheros estática. Las inserciones y borrados son costosos
  - La inserción requiere recorrer los distintos niveles insertando una nueva clave en los registros de índices correspondientes. Esto implica abrir un hueco moviendo índices al siguiente registro en cascada
  - El borrado deja huecos que deben ser eliminado siguiendo un proceso similar

Árboles B

- Descubiertos en 1972 por Bayer y McCreight, son el estándar hoy en día para la indexación de ficheros de datos
- Los árboles B son índices multinivel que resuelven los problemas de la inserción y borrado de registros y se basan en dos reglas
  - Permitir que los registros índice no estén llenos
  - No mover índices de un registro a otro cuando está lleno. En su lugar, dividir en dos el registro. Cuando dos registros están muy vacíos, unirlos en uno solo
- Cada registro de claves es un nodo del árbol B
  - El número máximo de claves por nodo (*m*) constituye el orden del árbol B
  - El número mínimo de claves permitidas en un nodo es normalmente la mitad del orden del árbol, con la excepción de la raiz

- Dos propiedades de las hojas:
  - Todas las hojas tienen la misma profundidad (pertenecen al mismo nivel)
  - Las hojas poseen apuntadores a los registros en el fichero de datos, de forma que el nivel completo de las hojas forma un índice ordenado del fichero de datos
- El árbol B se guarda íntegramente en un único fichero índice, de manera similar al árbol binario estudiado anteriormente



Búsqueda en un árbol B

- La búsqueda en un árbol B es tan eficiente como en un índice multinivel, con un orden $O(log_m n)$, siendo m el orden del árbol y n el número de elementos
  - Para un fichero de 1000000 de datos indexado mediante un árbol B de orden 10 localizamos cualquier dato con 6 accesos
- El procedimiento de búsqueda es el siguiente

1 Comenzar por el nodo raiz. Se lee de disco a memoria si es necesario

2 Realizar una búsqueda binaria o secuencial para localizar la clave *k* igual o inmediatamente inferior al valor de la clave búscada. Obtener la dirección *d* asociada a dicha clave

3 Si el nodo anterior no es una hoja, leer el siguiente nodo, apuntado por la dirección *d* y pasarlo a memoria. Volver al paso 2

4 En caso contrario leer el registro del fichero de datos apuntado por la dirección *d*

Inserción en un árbol B

- La inserción de una clave se realiza mediante dos pasadas:
  - Una pasada hacia abajo en la que se realiza una búsqueda para encontrar la hoja que le corresponde a la clave
  - Una pasada hacia arriba en la que se dividen los nodos cuando es necesario
- Los pasos a seguir son los siguientes

1 Realizar la búsqueda del nodo hoja que corresponde a la clave

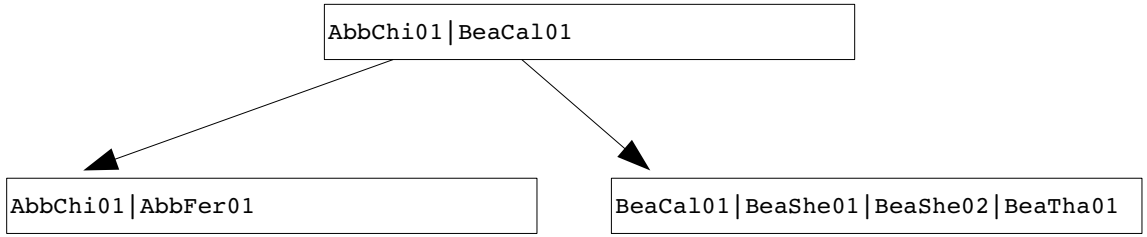
2 Insertar la clave en el nodo

3 Si la inserción no es posible por estar el nodo lleno, crear un nuevo nodo y pasar a éste la mitad de las claves. Realizar la inserción de la clave en el nodo que le corresponda

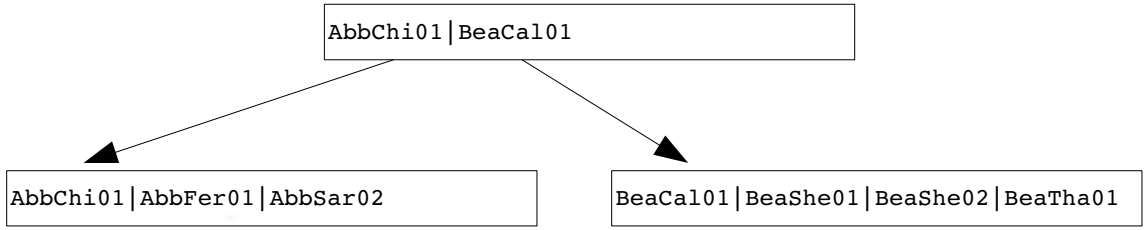
4 Si la clave insertada es la menor del nodo, actualizar la clave en el nodo padre (y realizar lo mismo, si es necesario, hasta la raiz)

5 Si se ha dividido el nodo original, insertar la primera clave del nuevo nodo en el nodo padre (para ello volver al paso 2)

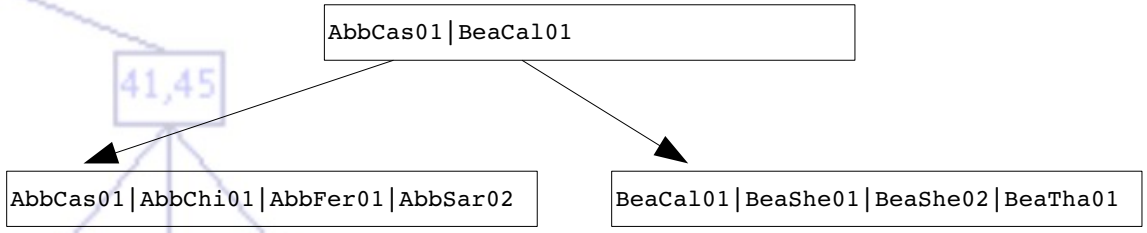
- Ejemplo:



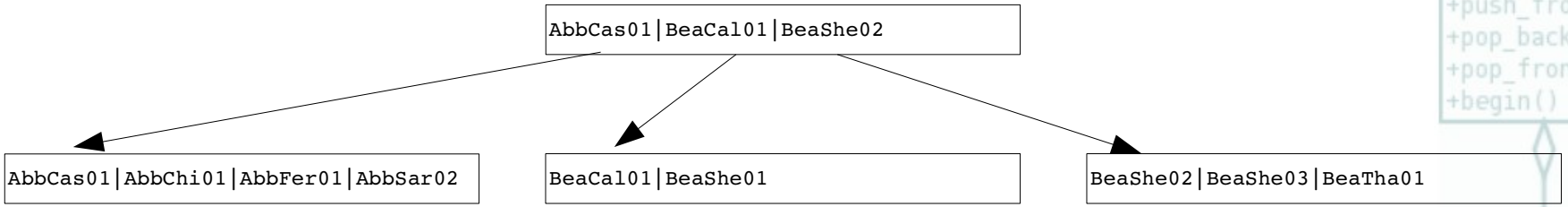
- Insertar *AbbSar02*. El nodo se inserta sin problemas



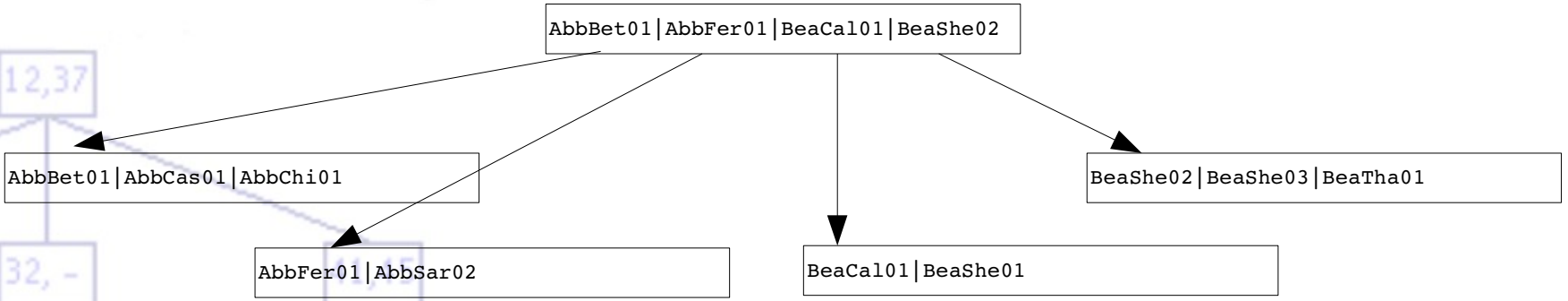
- Insertar *AbbCas01*. El nodo se inserta sin problemas, aunque al situarse en primera posición es necesario actualizar la clave del padre



- Insertar *BeaShe03*. No hay espacio en el nodo, por tanto es necesario dividir y redistribuir las claves. La clave se inserta en el segundo nodo



- Insertar *AbbBet01*. Una situación que combina las dos anteriores



Borrado en un árbol B

- El borrado también requiere dos pasadas
  - Una pasada hacia abajo en la que se realiza una búsqueda para encontrar la hoja que contiene la clave
  - Una pasada hacia arriba en la que se unen nodos cuando su ocupación baja del umbral permitido y se redistribuyen claves cuando lo anterior no es posible
- Los pasos a seguir son los siguientes

1 Realizar la búsqueda del nodo hoja que contiene la clave

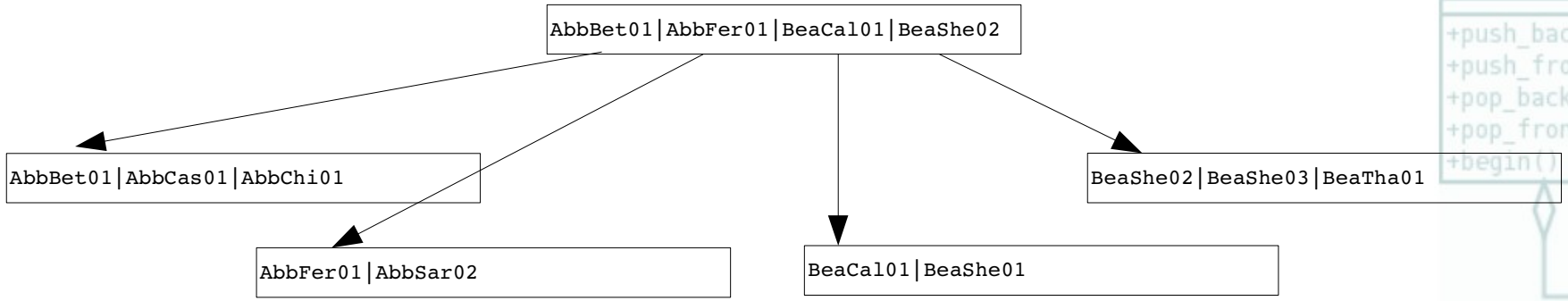
2 Eliminar la clave del nodo

3 Si la ocupación del nodo es superior a m/2 y la clave ocupaba la primera posición del nodo, actualizar los nodos de niveles superiores estableciendo una nueva primera clave

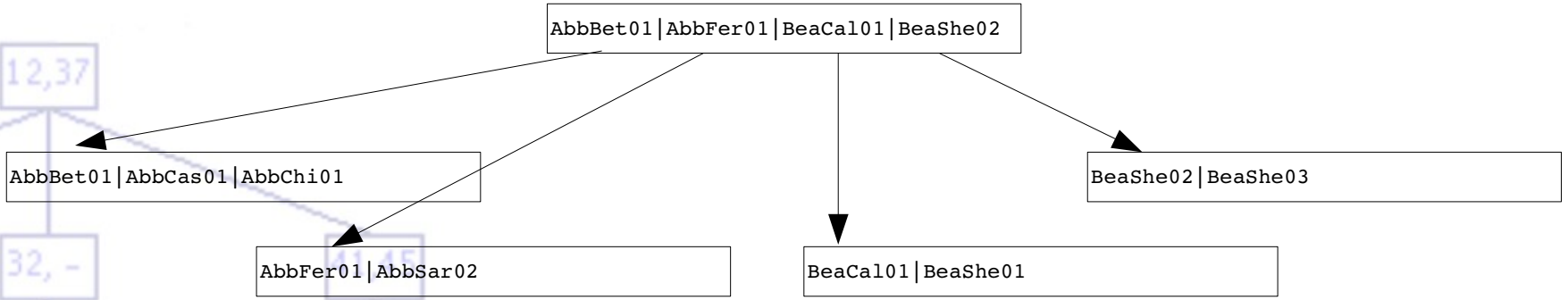
4 Si la ocupación del nodo es inferior a *m/2* y el hermano a la izquierda o derecha tiene espacio suficiente, pasar las claves al hermano y eliminar el nodo completo. Actualizar el padre eliminado la clave correspondiente (para ello volver al paso 2)

5 Si la ocupación del nodo es inferior a *m/2* y los hermanos a izquierda o derecha no tienen espacio suficiente, redistribuir las claves con uno de los dos hermanos. Actualizar las claves de los niveles superiores para reflejar los cambios

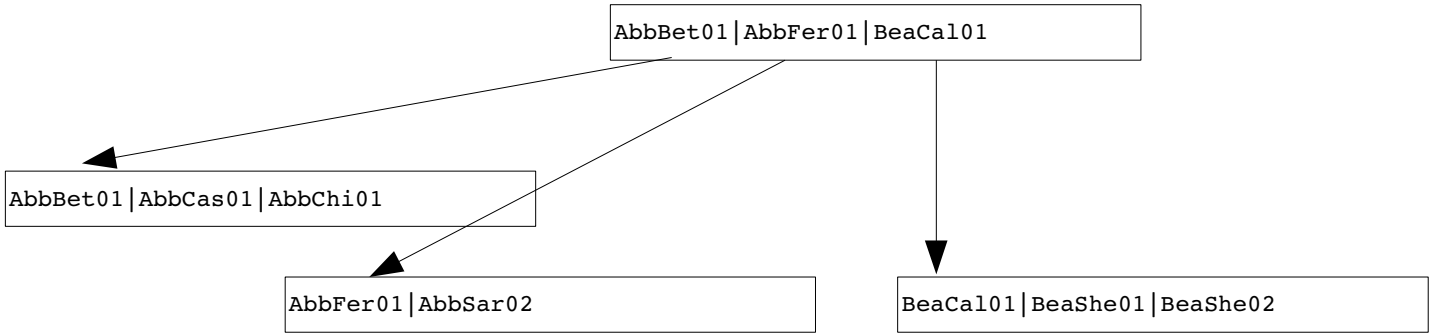
- Ejemplo:



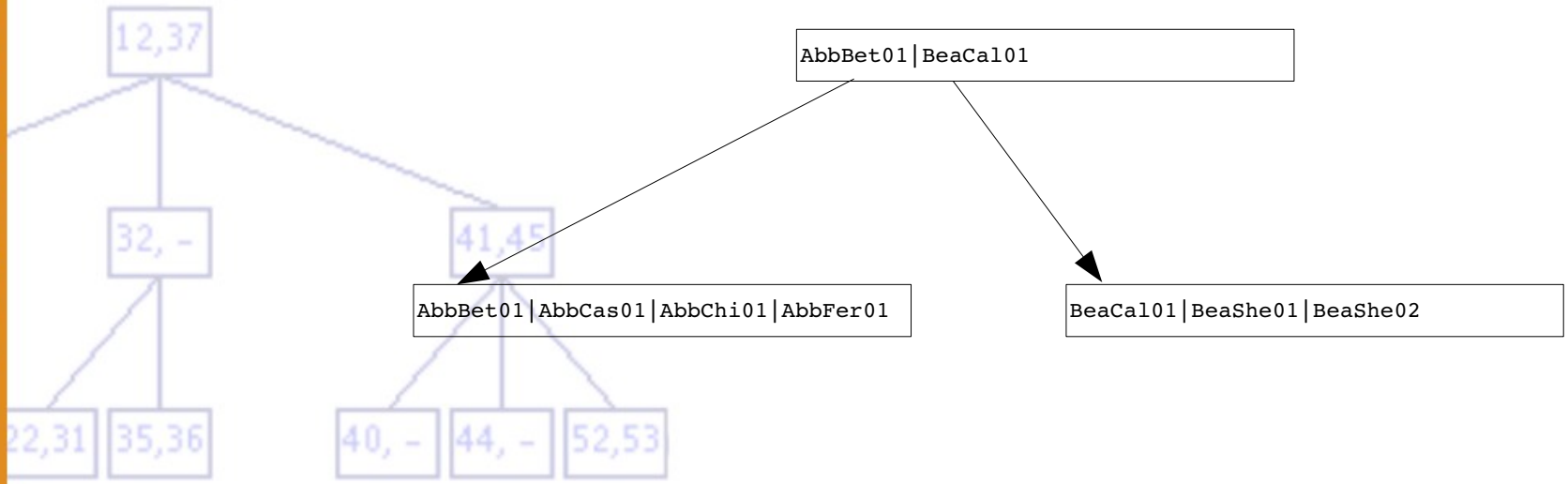
- Borrar *BeaTha01*. La clave se elimina trivialmente



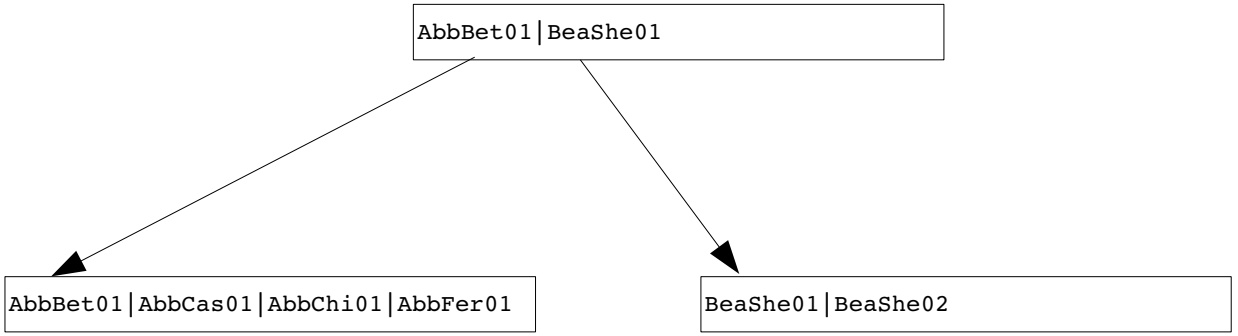
- Borrar *BeaShe03*. La ocupación del nodo cae por debajo del umbral. El hermano a la izquierda tiene espacio para albergar la clave restante (BeShe02)



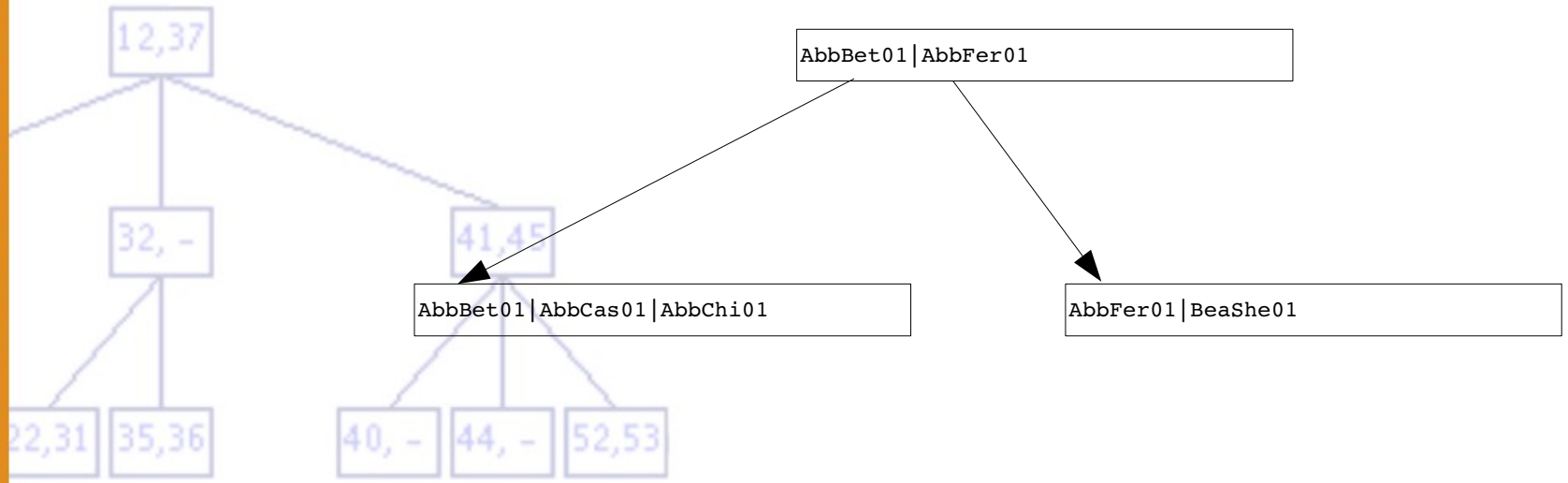
- Borrar *AbbSar02*. Situación similar a la anterior



- Borrar *BeaCal01*. Borrado trivial, aunque al tratarse de la primera clave, es necesario actualizar el padre

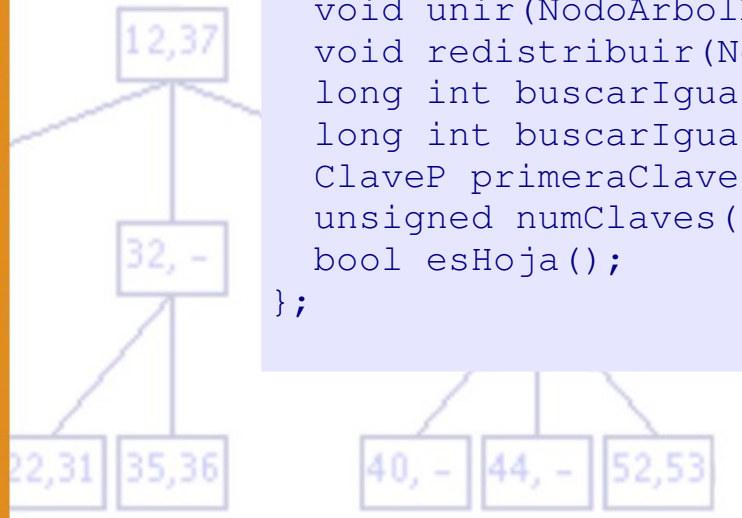


- Borrar *BeaShe02*. El hermano no tiene sitio para albergar la clave restante. Redistribuir en su lugar y actualizar la primera clave en el padre



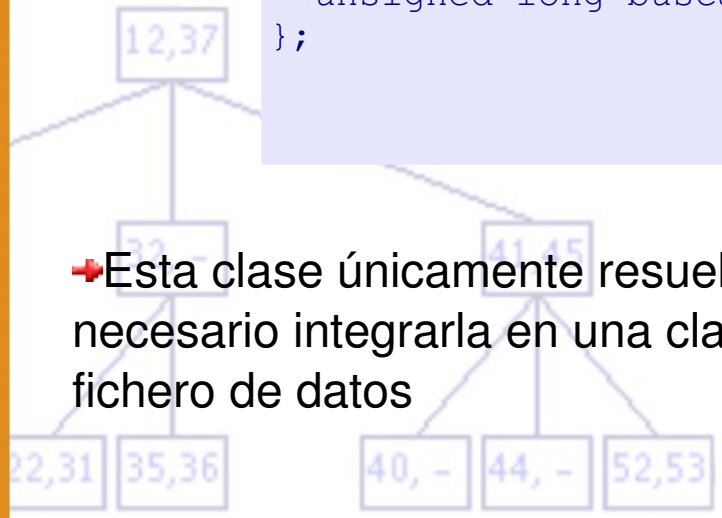
La clase para un nodo de un árbol B

```
template<class ClaveP, unsigned orden>
class NodoArbolB {
  ClaveP claves[orden];
  long int posiciones[orden];
  unsigned ocupacion;
  bool hoja;

public:
  NodoArbolB();

  void insertar(ClaveP clave, unsigned long posicion);
  void eliminar(ClaveP clave);
  void dividir(NodoArbolB &nodo);
  void unir(NodoArbolB &nodo);
  void redistribuir(NodoArbolB &nodo);
  long int buscarIgual(ClaveP clave);
  long int buscarIgualMenor(ClaveP clave);
  ClaveP primeraClave();
  unsigned numClaves();
  bool esHoja();
};
```

La clase para el árbol B

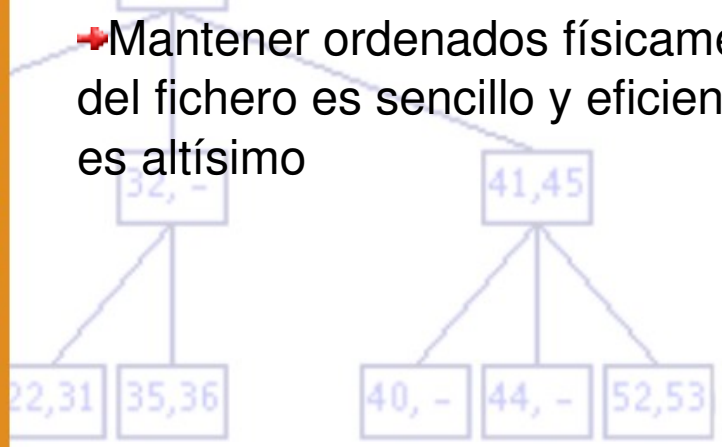
```
#include <string>
#include <fstream>
using namespace std;

template<class ClaveP, unsigned orden>
class ArbolB {
  Fichero<NodoArbolB<orden, ClaveP> > fDatos;

public:
  ArbolB(string fDatos, bool crear = false);

  void insertar(ClaveP claveP, unsigned long pos);
  void borrar(ClaveP claveP);
  unsigned long buscar(ClaveP claveP);
};
```

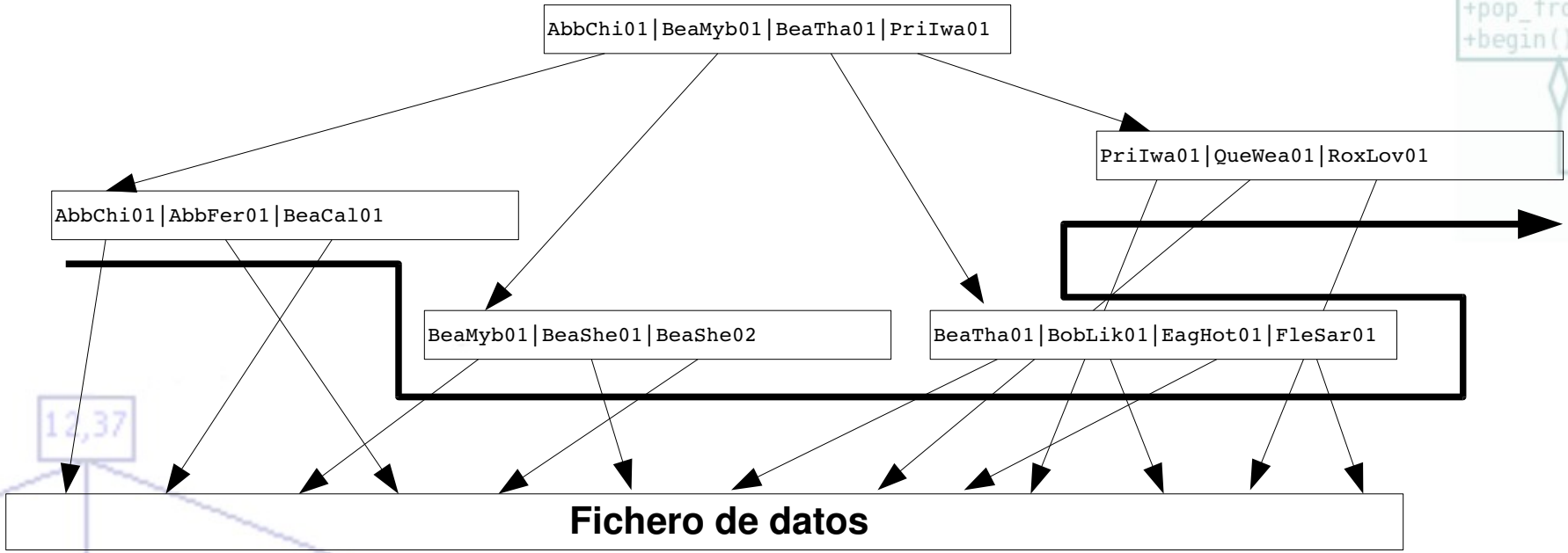
- Esta clase únicamente resuelve un índice en forma de árbol B. Posteriormente es necesario integrarla en una clase similar a *FicheroIndices* (tema anterior) con un fichero de datos

Acceso secuencial indexado: Árboles B+

- Los árboles B proporcionan una solución definitiva al problema de la indexación de un fichero para la búsqueda de un dato por clave
- Pero ¿Que ocurre si nuestra aplicación requiere un acceso secuencial a los registros siguiendo la ordenación dada por la clave?
  - Ejemplo: obtención de listados ordenados
- Hasta ahora tenemos dos posibilidades:
  - Mantener ordenados físicamente los registros del fichero de datos. El recorrido del fichero es sencillo y eficiente, aunque el coste de mantener el fichero ordenado es altísimo

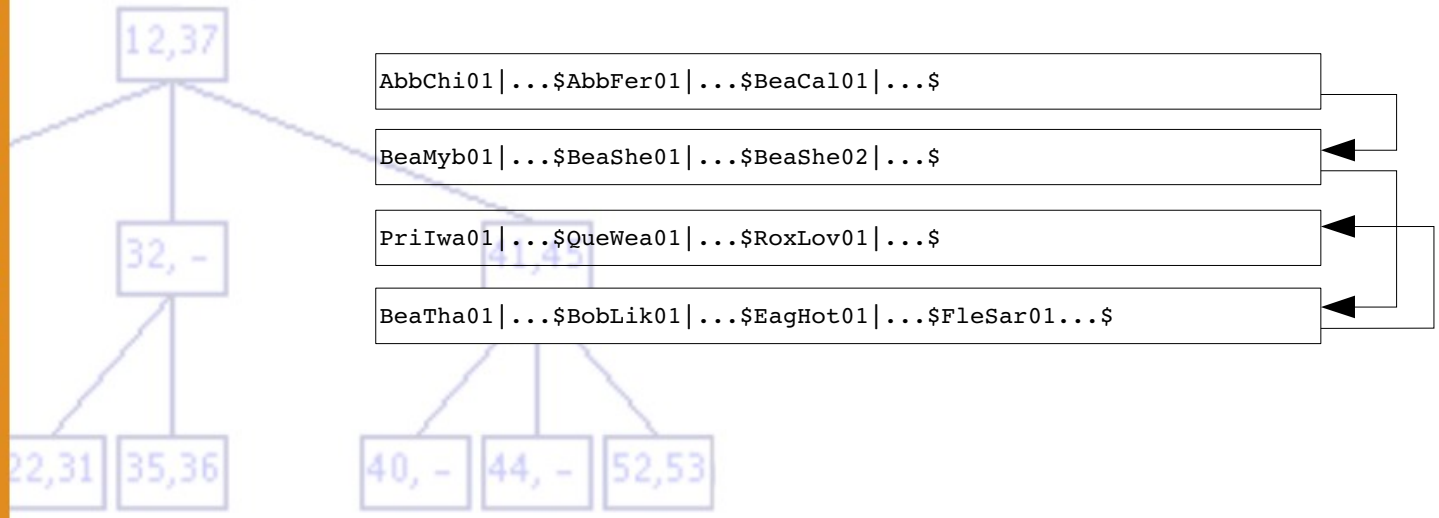


- Recorrer las hojas del árbol B de izquierda a derecha. Esto proporciona un listado de los registros ordenados por clave. Sin embargo implica un recorrido en inorden, leyendo todos los nodos del árbol B más accesos adicionales a los registros del fichero de datos



Ficheros de secuencias de bloques

- La solución al problema es mantener ordenados físicamente los registros del fichero, pero sólo a nivel de bloque
- Cada bloque va a contener varios registros, y el fichero de datos va a estar constituido por una secuencia ordenada lógicamente de bloques

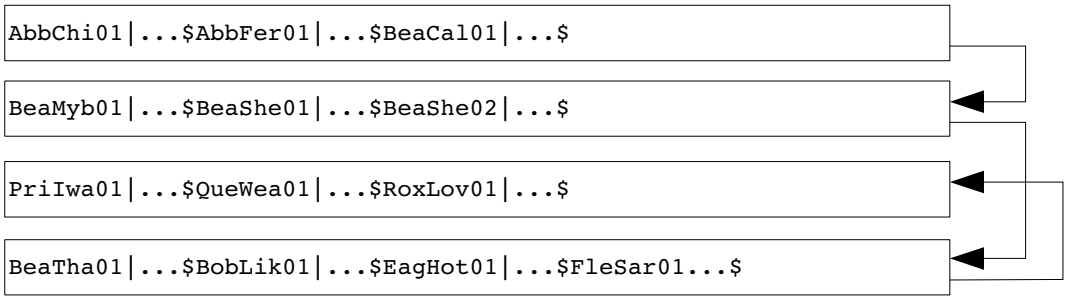


- Los bloques tienen las siguientes características:
  - Son de tamaño fijo. El mejor rendimiento se consigue si su tamaño es igual al de un bloque del sistema de ficheros
  - Un bloque puede estar parcialmente vacío, aunque su ocupación no puede ser menor del 50%
  - Los registros dentro de cada bloque están ordenados por la clave
  - Cada bloque contiene un apuntador al siguiente bloque en la secuencia ordenada. La ordenación lógica no tiene que coincidir con la física
- Mantener ordenado un bloque sí es eficiente puesto que se transfiere íntegramente a memoria para realizar una operación de inserción o borrado
- Si el tamaño del bloque es igual al bloque del sistema de ficheros, la transferencia requiere un único acceso

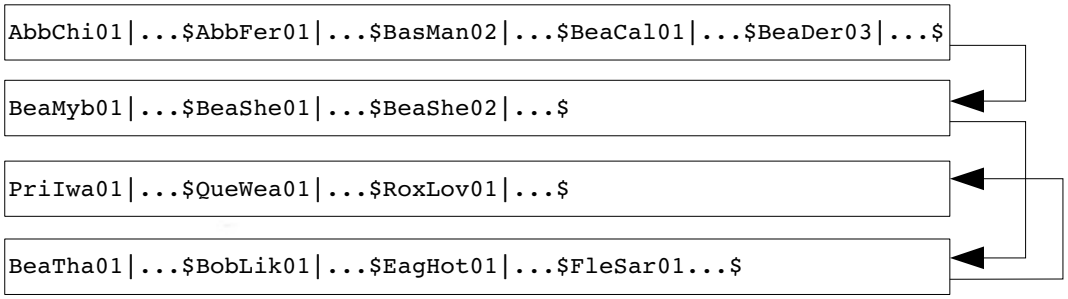
- Realizar un recorrido ordenado de los registros es sencillo y eficiente
  - Basta con ir leyendo de forma ordenada cada bloque y procesar los registros en su interior, siguiendo también su orden interno
- La inserción de un nuevo registro en el fichero implica los siguientes pasos

1 Localizar el bloque al que pertenece (esto lo haremos utilizando un árbol B, como veremos más adelante)
2 Pasar el bloque a memoria
3 Si el bloque tiene espacio para el nuevo registro, realizar su inserción en la posición correspondiente. Salvar el bloque en el fichero y terminar
4 Si el bloque no tiene espacio, entonces es necesario crear un nuevo bloque, al final del fichero o en un hueco existente de un bloque borrado anterior. Insertar el nuevo bloque en la lista, a continuación del bloque original
5 De forma similar a los árboles B, repartir los registros entre los dos bloques existentes e insertar el nuevo registro en el que corresponda. Salvar ambos bloques en el fichero

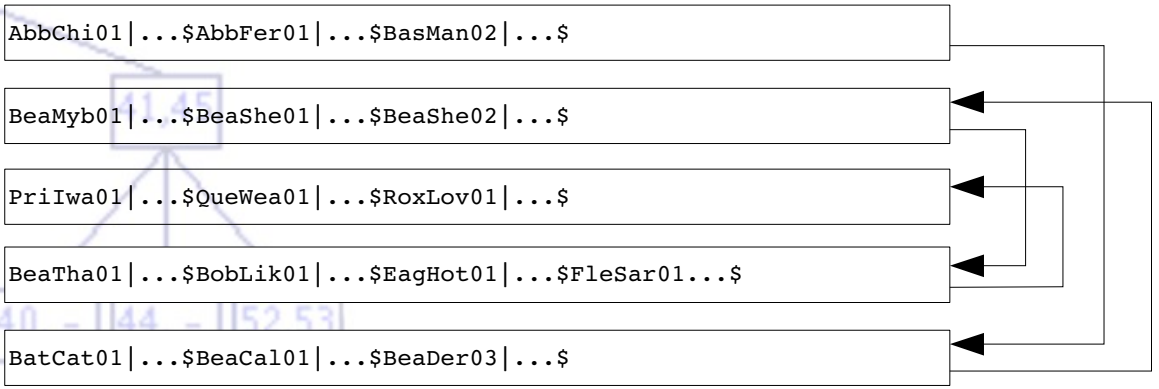
- Ejemplo:



- Insertar *BeaDer03* y *BasMan02*:



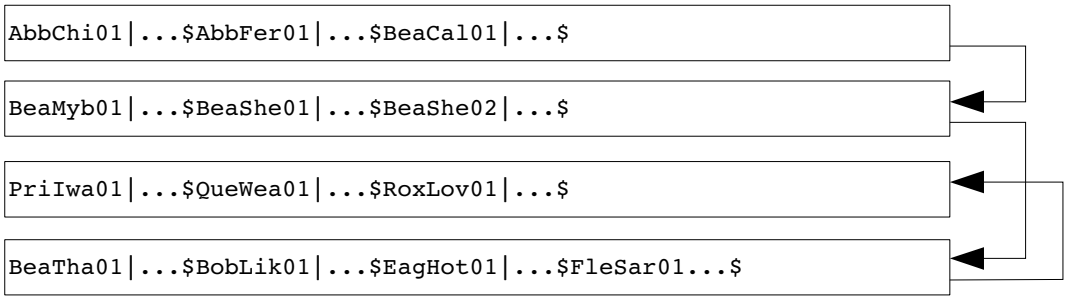
- Insertar *BatCat01*:



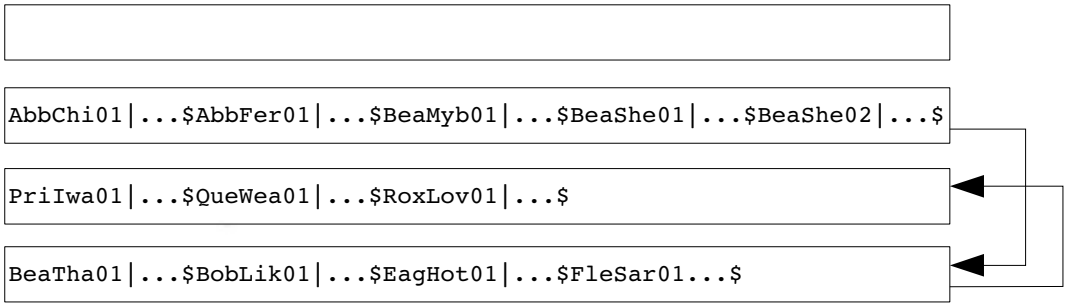
- El borrado es similar al borrado en un árbol B

1 Localizar el bloque donde está situado el registro
2 Pasar el bloque a memoria
3 Eliminar el registro del bloque. Si la ocupación del bloque sigue siendo superior al 50% terminar
4 En caso contrario comprobar la ocupación de los bloques anterior y posterior
5 Si uno de los dos tiene espacio para albergar los registros del bloque actual, pasar éstos al bloque en cuestión, desconectar el bloque actual y marcar como borrado para ser aprovechado en inserciones posteriores
6 En caso contrario, escoger uno de los dos bloques y redistribuir los registros entre ambos bloques de forma equitativa

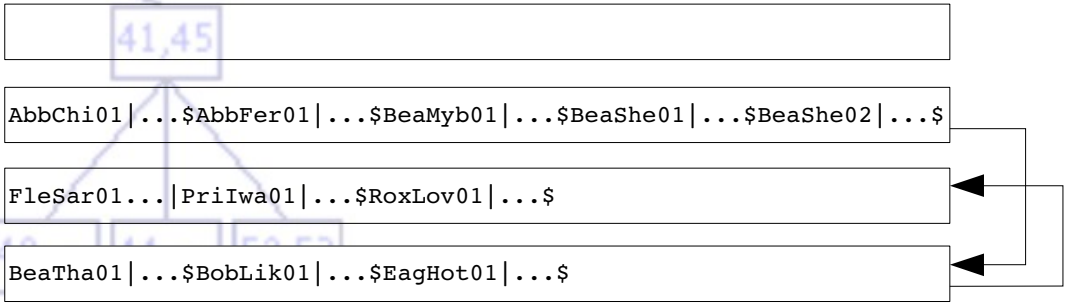
- Ejemplo:



- Eliminar *BeaCal01*. Los registros restantes se pasan al segundo bloque



- Eliminar *QueWea01*. De nuevo la ocupación del bloque cae por debajo del 50%. Los registros de los dos primeros bloques se redistribuyen



La representación de un bloque en memoria

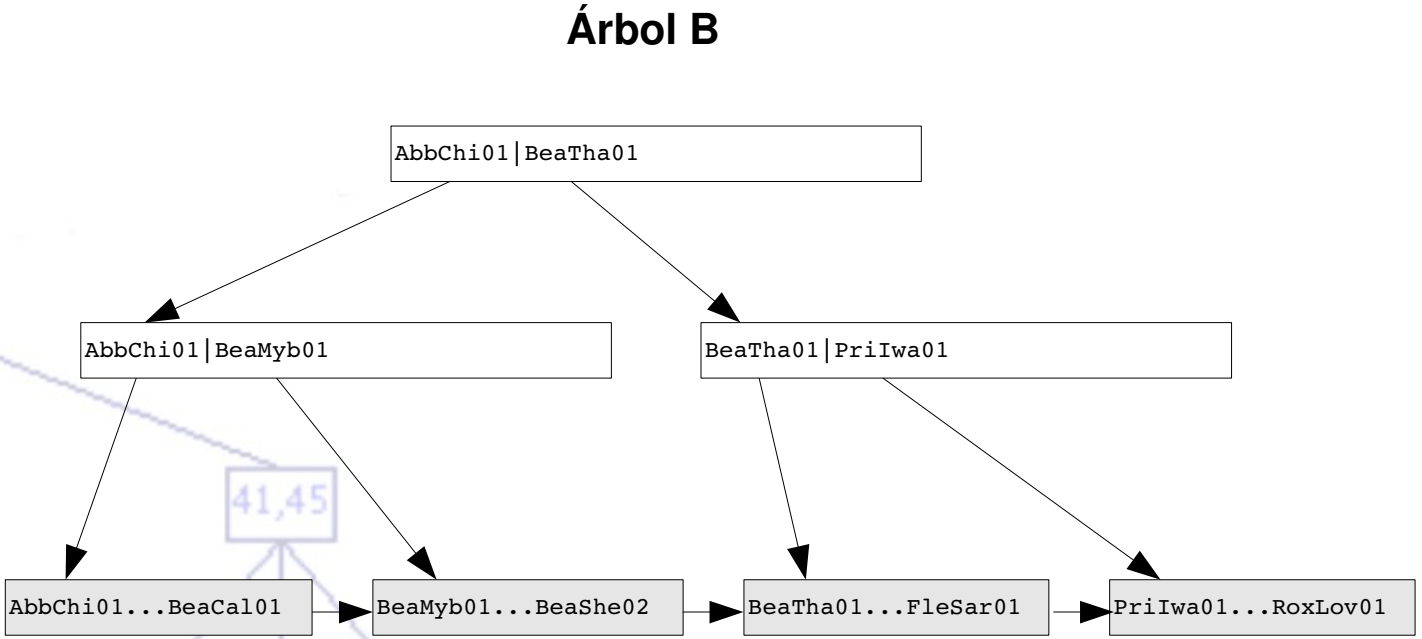
```
template<ClaveP clave, class Registro, unsigned tamBloq>
class BloqueReg {
  ClaveP claves[tamBloq];
  Registro registros[tamBloq];
  unsigned ocupacion;
  unsigned bloqueSig;

public:
  BloqueReg();

  void insertar(ClaveP clave, Registro reg);
  void eliminar(ClaveP clave);
  Registro leer(ClaveP clave);
  unsigned leerBloqueSig();

  void dividir(BloqueReg &bloque);
  void unir(BloqueReg &bloque);
  void redistribuir(BloqueReg &bloque);
  ClaveP primeraClave();
  unsigned numReg();

  // Iteración por el bloque
  void iniciarIteracion(ClaveP clave);
  bool obtenerSiguiente(ClaveP &clave, Registro &reg);
};
```

Fichero estructurado en bloques

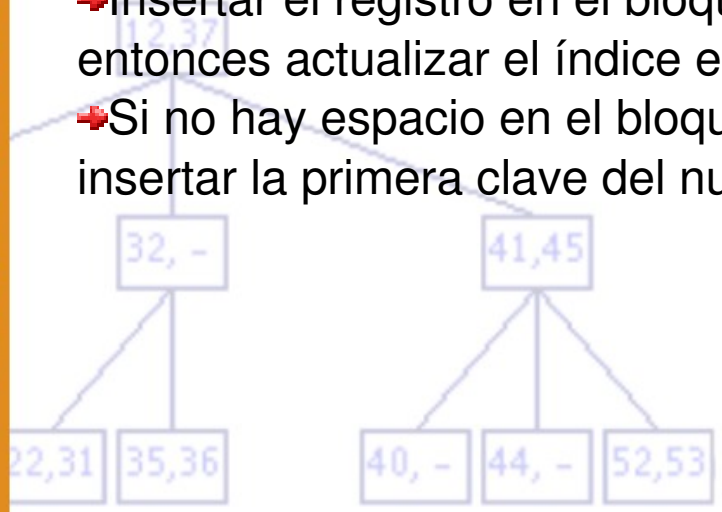
```
#include <string>
using namespace std;

template<class ClaveP, class Registro, unsigned tamBloq>
class FicheroBloques {
  Fichero<BloqueReg<ClaveP, Registro, tamBloq> > f

public:
  FicheroBloque(string fDatos);
  ~Fichero(); // Cierra el fichero

  void insertar(unsigned bloq, ClaveP clave, const Registro &reg);
  bool bloqueDividido(unsigned &bNuevo);

  void eliminar(unsigned bloq, ClaveP claveP);
  bool bloqueUnido(unsigned &bDest);
  bool bloqueRedistribuido(unsigned &bloq1, unsigned &bloq2);

  void leer(unsigned bloq, ClaveP claveP, Registro &reg);
  void actualizar(unsigned bloq, ClaveP claveP, const Registro &reg);
  ClaveP primeraClave(unsigned bloq);

  // Iteración por el fichero
  void iniciarIteracion(unsigned bloq, ClaveP clave);
  bool obtenerSiguiente(ClaveP &clave, Registro &reg);
};
```

Árboles B+

- Denominamos árbol B+ a la estructura que combina un fichero de datos estructurado en una secuencia de bloques de registros con un árbol B para indexar dichos bloques



- Un árbol B+ permite tanto accesos por clave (mediante el árbol B) como recorridos secuenciales a través de la secuencia de bloques
- Las hojas del árbol B contienen las claves iniciales de cada uno de los bloques, así como las direcciones de los mismos
- La inserción contempla los siguientes aspectos
  - Localizar mediante al árbol B el bloque que le corresponde
  - Insertar el registro en el bloque. Si el registro se situa en primera posición, entonces actualizar el índice en el árbol B
  - Si no hay espacio en el bloque, entonces se crea un nuevo bloque. Esto implica insertar la primera clave del nuevo bloque en el árbol B



- El borrado sigue los siguientes pasos:
  - Localizar el bloque al que pertenece el registro mediante el árbol B
  - Eliminar el registro. Si ocupaba la primera posición, actualizar los índices correspondientes en el árbol B
  - Si se produce la eliminación de un bloque, eliminar la entrada correspondiente en el árbol B
  - Si se produce una redistribución de registros, actualizar los índices correspondientes en el árbol B

La clase árbol B+

```
#include <string>
using namespace std;

template <class ClaveP, class Registro, unsigned tamBloq, unsigned orden>
class ArbolBMas {
  FicheroBloques<ClaveP, Registro, tamBloq> fDatos;
  ArbolB<ClaveP, orden> indice;

public:
  ArbolBMas(string fDatos, bool crear = false);
  ~ArbolBMas();

  void insertar(ClaveP claveP, Registro reg);
  void borrar(ClaveP claveP);
  Registro buscar(ClaveP claveP);

  // Iteración por el fichero
  void iniciarIteracion(ClaveP claveP);
  bool obtenerSiguiente(ClaveP &claveP, Registro &reg);
};
```