CS 225 – Data Structures

ZJUI – Spring 2022

Lecture 8: Index Data Structures

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

8 Relations and Index Data Structures

With this section of the course we make a step towards the field of (relational) databases, where all the problems we investigated so far by data structures that effectively and efficiently enable access to and update of bulk data come together

In relational databases we deal with large relations, i.e. sets of tuples

We do not just require access to single records, but to sets of records to support operations on bulk data

Furthermore, as databases aim at persistent storage, we usually assume the data to reside on external storage (magnetic disk drives)

As databases accumulate data, the request to deal with bulk data is essential

However, we will not look into the low-level interface between data in buffers and in external storage, as for this database management systems do not rely on the operating system, but organise block access themselves

8.1 Relations

A ***relation schema*** is a finite set $RS = \{A_1, \ldots, A_n\}$ of **attributes**, each associated with a **domain** $dom(A_i)$ $(1 \leq i \leq n)$

Here understand a domain just as a set, though it naturally comes with operations applicable to values of that set, i.e. it is a data type

We use the notation $RS = \{A_1 : T_1, \ldots, A_n : T_n\}$, if we want to emphasise the domains $T_i = dom(A_i)$

A ***database schema*** is a finite set $S = \{RS_1, \ldots, RS_k\}$ of relation schemata

Tuples and Database Instances

If $RS = \{A_1 : T_1, \ldots, A_n : T_n\}$ is a relation schema, then an n-tuple $(A_1 : v_1, \ldots, A_n : v_n)$ with $v_i \in T_i$ is an ***RS-tuple***

A ***relation*** for the relation schema $RS = \{A_1 : T_1, \ldots, A_n : T_n\}$ is a finite set of RS-tuples

A ***database instance*** of a database schema $S = \{RS_1, \ldots, RS_k\}$ is a family $\{R_i \mid 1 \leq i \leq k\}$, where each R_i is an RS_i-relation

Instead of database instance we simply talk of a database or an instance

Note that not only relations may be large and containing many tuples, but also tuples may contain values for many attributes

Keys and Foreign Keys

Databases are constrained by many conditions: keys and foreign keys are relevant for the physical implementation

A ***key*** on a relation schema $RS = \{A_1 : T_1, \ldots, A_n : T_n\}$ is a subset $K = \{A_{i_1}, \ldots, A_{i_\ell}\}$ of the set of attributes of RS

An RS-relation R satisfies the ***key constraint*** defined by K iff for any two tuples $t_1, t_2 \in R$ with $t_1[K] = t_2[K]$ (i.e. they coincide on the attributes in K) are equal

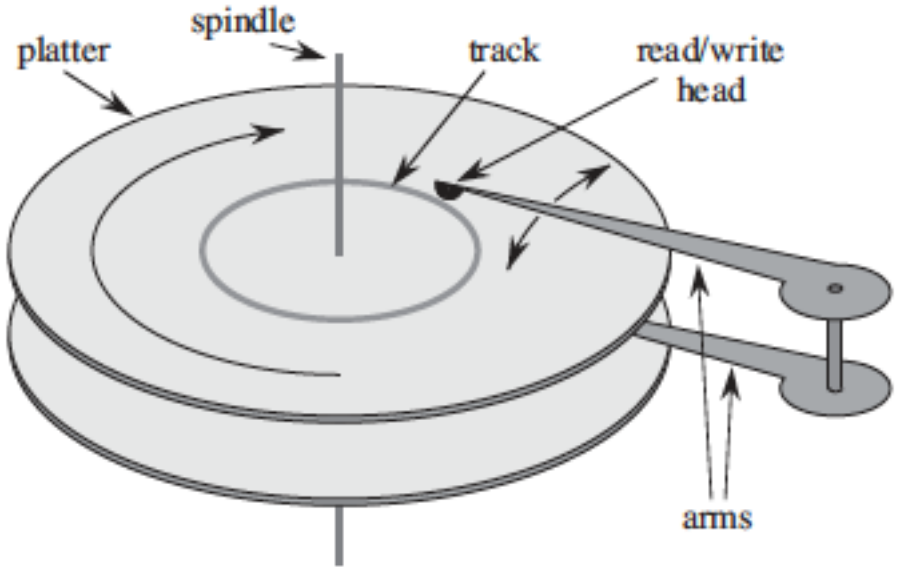
In other words, the key values determine the tuples

A subset $F = \{A_{i_1}, \ldots, A_{i_\ell}\}$ of the set of attributes of RS is a ***foreign key*** iff F is a key of another relation schema $RS' \in S$

If F is a foreign key on RS referring to a key of RS', then relations R and R' for RS and RS', respectively, satisfy the ***foreign key constraint*** defined by F iff for every tuple $t \in R$ there is a tuple $t' \in R'$ such that $t[F] = t'[F]$

Magnetic Disk Storage

Recall the organisation of a magnetic disk:



- One or more **platters** with magnetisable surface rotate at a constant speed around a spindle

Magnetic Disk Storage / cont.

- Concentric circles on the platter surfaces define **tracks**, in which data can be stored
- The collection of tracks with the same radius on a stack of platters is called a **cylinder**
- Data can be read or written by a **read/write head** that can be moved to different tracks by means of a mechanical **arm**
- Usually, data on a track is divided into several **blocks** using a fixed block size—blocks are the atomic unit for the transfer of data between disk storage and main memory
- Due to the use of mechanical components access to data on disks is comparably slow and even slower, when multiple tracks/cylinders need to be accessed

External Storage of Relations

The tuples of a relation will be stored in several blocks, each with a block address

Then we can assume that each block comprises a **block header** and an array of tuples—we ignore tuples that span over more than a single block

The length of the array depend on the blocksize and the space required for each tuple

For each relation we provide one or more **(access) keys**—not to be confused with the keys defined above for relations

- A **primary (access) keys** is a single attribute or a combination of attributes, for which values in the relation are unique (i.e. they actually define keys)

- A **secondary (access) keys** is a single attribute or a combination of attributes, for which values in the relation need not be unique

Buffer and Index Structure

An **index structure** is a data structure for the organisation of primary/secondary keys and the tuples they refer to

Different to all data structures we investigated so far we usually separate the key and the tuple, because the latter one is usually very large and subject to several access keys

We therefore also talk about the **main data** that are separated from the indices

To access data in external storage the corresponding block first has to be moved into a buffer in main memory, where we refer to it as a **page**

Consequently, the buffer manager has to maintain the translation of block addresses into main memory addresses—we will tacitly ignore this and make the simplifying assumption that the data always resides in the buffer

The buffer manager also has to maintain the replacement of pages when needed—the corresponding strategies are beyond the scope of this course

Operations of Interest

Same as for binary search trees (and its variants: AVL, splay, red-black trees) we have to consider operations for ***find***, ***insert*** and ***delete*** a tuple in a relation given a primary key

In case of a secondary key this extends to multiple tuples in case of ***find*** and ***delete***

In case of an ***insert*** key constraint must be preserved

We may also consider retrieval of a range of tuples given minimum and maximum values of a primary key

The operations may trigger the reorganisation of the index structure and the main data

For the latter ones we usually foresee also **overflow blocks** to avoid reorganisation to occur too often

8.2 B-Trees

We first consider the case of primary keys

B-trees are balanced, multiway search trees (similar to (a, b)-trees) with the difference that the corresponding tuples are not stored in the nodes of the tree, but in a block in external storage

Therefore, in the index structure we have to consider pairs (k, p), where k is a primary key, and p is a pointer to the associated tuple—nonetheless we refer to such a pair simply as a *key*

However, for the search in the index structure (for *find* and *delete*) only the key k is relevant

Likewise, for *insert* we need a primary key k and the associated tuple t (from which k can be derived), but for finding a location, where the new tuple is to be stored only the primary key k will be relevant

Definition

A ***B-tree*** of **order** m is a tree with the following properties:

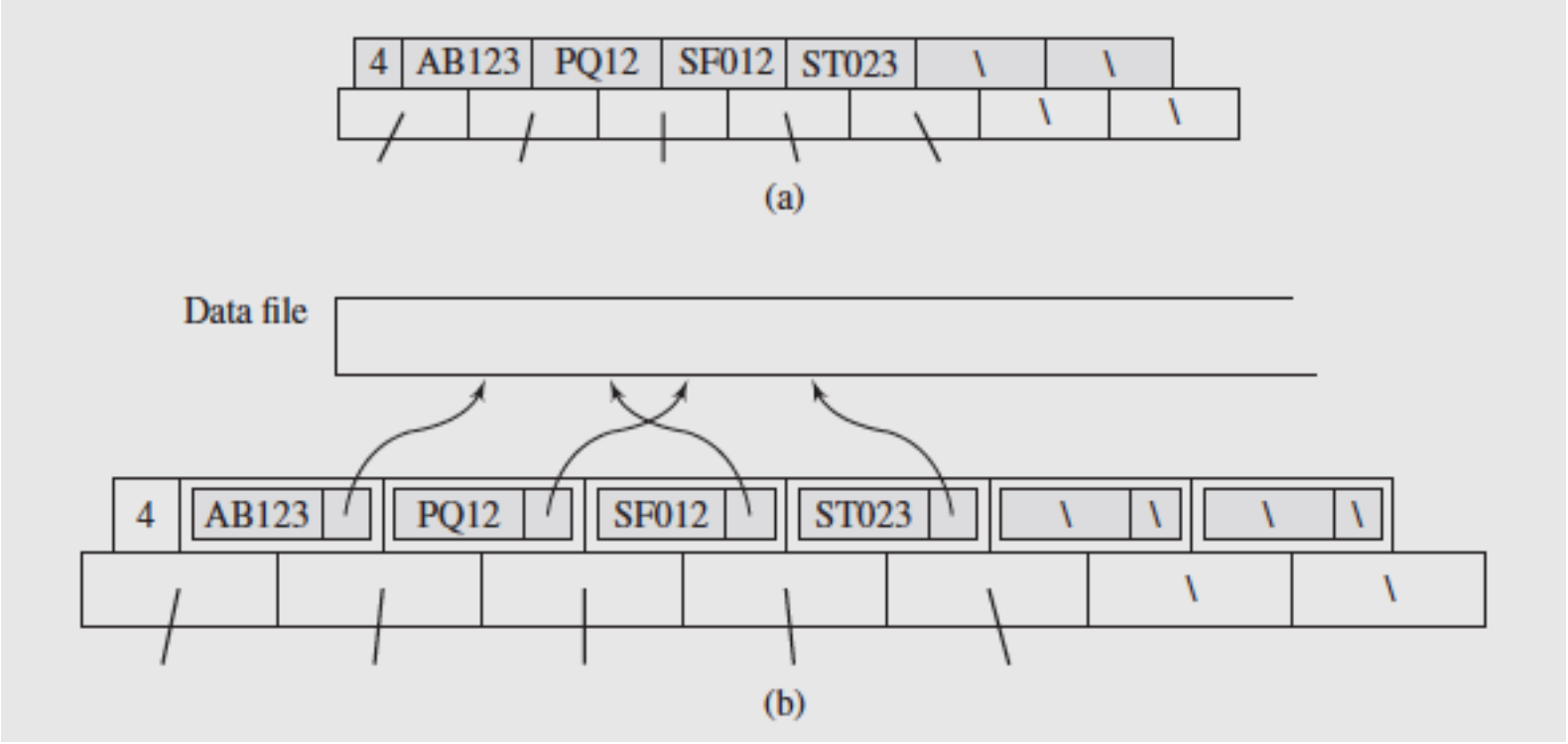
- Each node contains a sequence $(k_1, p_1), \ldots, (k_s, p_s)$ of keys with $s \geq 1$ for the root (unless the root is also a leaf) and $\lceil m/2 \rceil \leq s + 1 \leq m$ for all other nodes
- Each non-leaf node with s keys has exactly $s + 1$ children
- All simple paths from the root to a leaf have the same length
- All nodes in the i'th successor tree of a node with keys $(k_1, p_1), \ldots, (k_s, p_s)$ contain only keys (k, p) with $k_{i-1} < k < k_i$

Naturally, the last condition applies in this form only for $2 \leq i \leq s$—for $i = 1$ we only require $k < k_1$, and for $i = s + 1$ we require $k_s < k$

The factor $\frac{s+1}{m}$ is commonly called the **fill factor** of a node

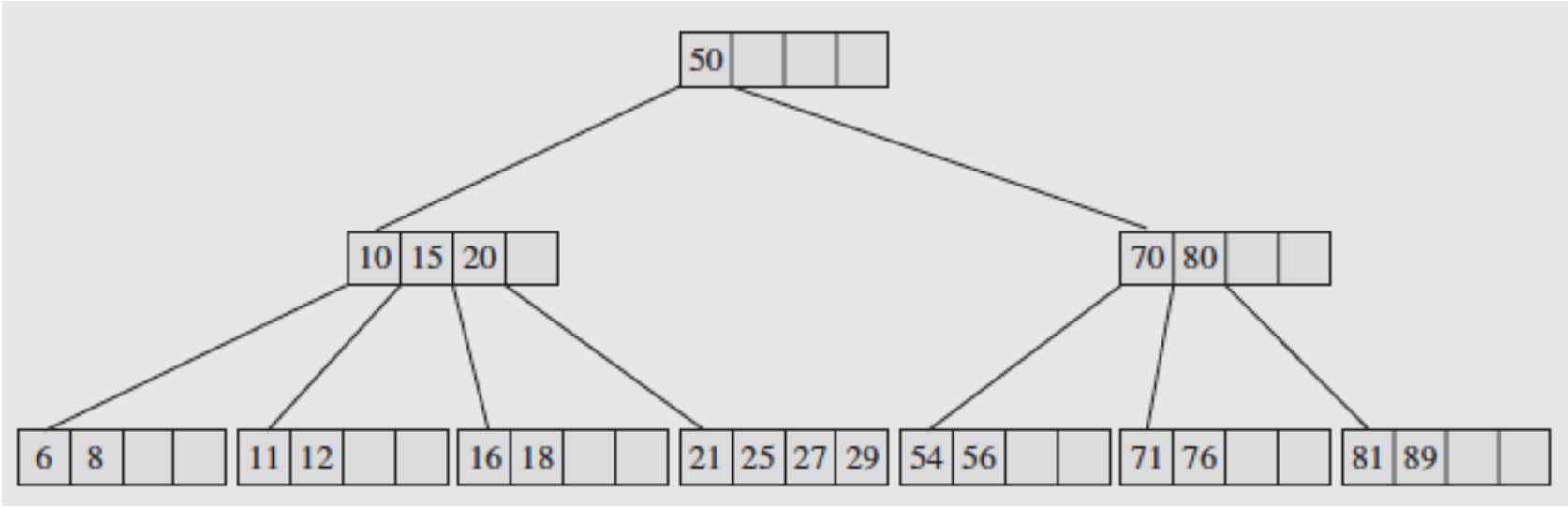
Example 1

The upper part (a) shows a node in a B-tree of order 7 with four keys omitting the pointers to external storage, the lower part (b) shows the same node with the pointers to the main data included



Example 2

In abbreviated form (omitting pointers to external storage and a counter for the number s of keys in a node) a B-tree of order 5 takes the following form:



Normally, for database index structures the order m of B-trees is rather high, as the number of tuples is very large, and a small height of the tree is advantageous for search

Search in a B-Tree

In order to find a tuple we assume to be given a primary key k

Then search in a B-tree is realised analogous to search in a BST or AVL tree:

- Search for a key (k, p) in a sequence of nodes starting from the root of the tree
- Search a node sequentially until either $k = k_i$ or $k_{i-1} < k < k_i$ holds for some i
  - In the former case use the pointer p_i to retrieve the sought tuple
  - In the latter case continue the search with the i'th child node—in case of a leaf return that no tuple with the given key exists

Clearly, the time complexity of search in a B-tree is in $O(h)$, where h is the height of the tree

The worst case (i.e. largest height) results, when the root contains just one key, and the number of children of each non-leaf node (except the root) is $s + 1 = \lceil m/2 \rceil$

Complexity of Search in a B-Tree

Then the are 2 nodes on level 2 of the tree, and the number of nodes on level $i+1$ is $(s+1)$-times the number of nodes on level i—hence (by inducton) the number of nodes on level i is $2(s+1)^{i-2}$ for $i \geq 2$

This gives $2s(s+1)^{i-2}$ keys on level $i \geq 2$, in total

$$1 + 2s \sum_{i=0}^{h-2} (s+1)^i = 1 + 2s \frac{(s+1)^{h-1} - 1}{s} = 2(s+1)^{h-1} - 1$$

If n is the number of nodes in a B-tree of height h, we must have

$$n \geq 2(s+1)^{h-1} - 1 \qquad \text{or} \qquad h \leq \log_{s+1} \frac{n+1}{2} + 1$$

Example. For $n = 2,000,000$ and $m = 200$ we have $s + 1 \geq 100$ and thus $h \leq \log_{100} 10^6 + 1 = 4$

That is, to find a tuple in a relation with 2,000,000 tuples it suffices to search through 4 nodes of a B-tree—these nodes may have to be loaded into main memory

Insertion into a B-Tree

Insertion of a new tuple t with primary key k is also rather straightforward

As for *find* we first search for a key (k, p) in the B-tree; if successful, the new tuple cannot be inserted

If not successful, the search continues to a leaf, which may contain the keys $(k_1, p_1), \ldots, (k_$

The case $s + 1 < m$ is simple: first determine i with $k_i < k < k_{i+1}$

Then insert t into the main data and return the address p, and update the leaf to contain the keys $(k_1, p_1), \ldots, (k_i, p_i), (k, p), (k_{i+1}, p_{i+1}), \ldots, (k_s, p_s)$

The case $s + 1 = m$ requires the leaf to be **split**

Insertion into a B-Tree / cont.

In case $s+1=m$ also determine i with $k_i < k < k_{i+1}$, insert t into the main data and return the address p

The keys (including (k,p)) are then $(k'_1,p'_1),\ldots,(k'_{s+1},p'_{s+1})$

With $i=\lceil\frac{m+1}{2}\rceil$ create two new leaves with the keys $(k'_1,p'_1),\ldots,(k'_{i-1},p'_{i-1})$ and $(k'_{i+1},p'_{i+1}),\ldots,(k'_{s+1},p'_{s+1})$

Then add the key (k'_i,p'_i) together with the pointers to the two leaves into the parent node

If non-leaf nodes need to be split, proceed in the same way

If finally the root needs to be split, create a new root containing only a single key (k'_i,p'_i)

Insertion into Main External Storage

We still need to consider the insertion into main external storage—we may assume that the tuples of a relation are ordered with respect to the primary key

As (in both cases above) we determined $k_i < k$, then the address p_i gives us the address of the tuple with key k_i, and the new tuple needs to be inserted right after it

However, the next tuple in the same block (or in the successor block) has key k_{i+1} (unless $i = s$), so we need to sort the data

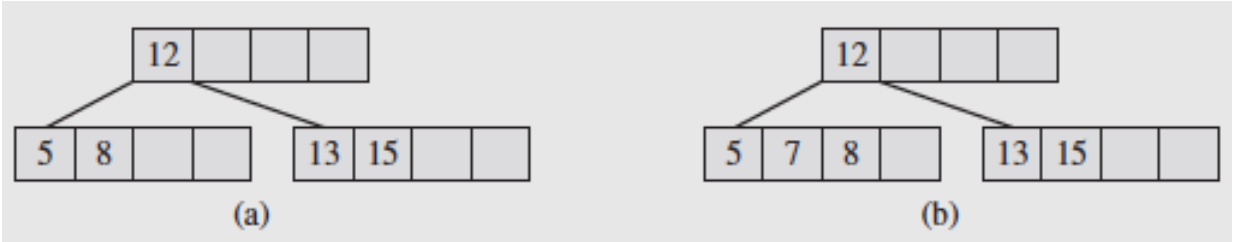
In order to avoid sorting to become necessary too often we usually use one or more **overflow blocks**, in which new tuples are stored first

Only if the available space is used up, the tuples are sorted—at the same time tuples that are marked as deleted are physically removed from storage

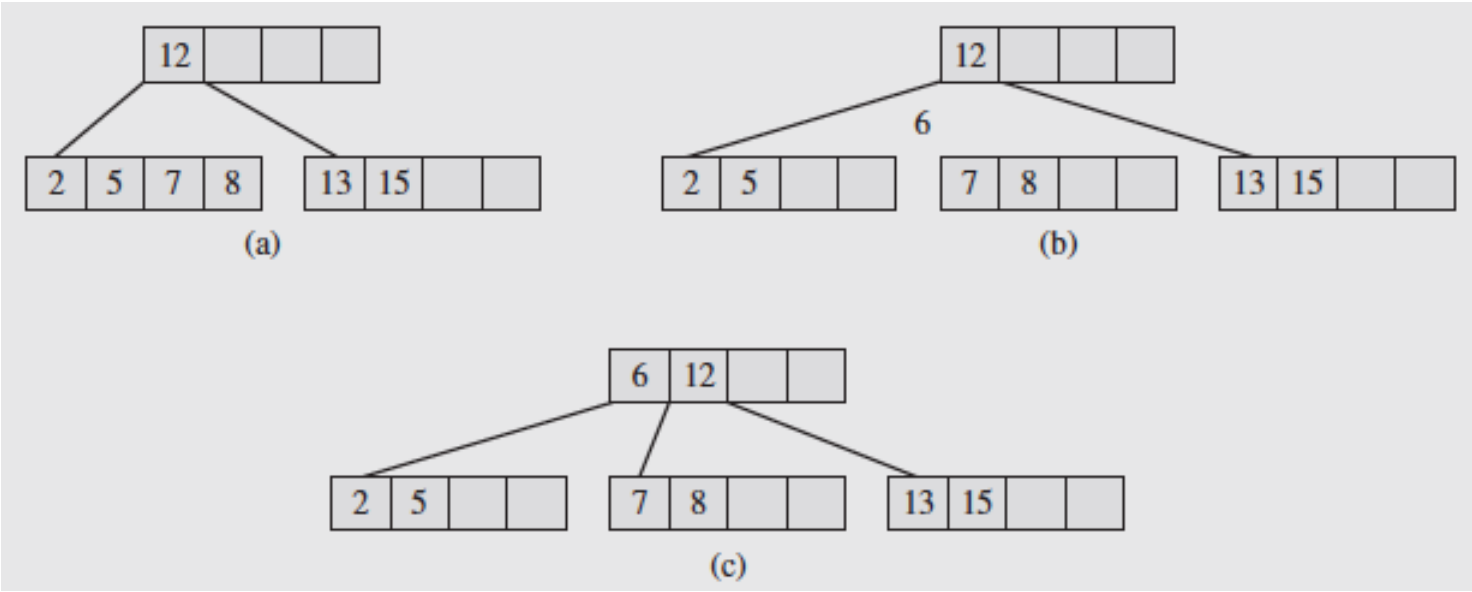
The number of blocks that should be available for the main data associated with a leaf in the B-tree depends on the size of tuples and the order of the B-tree

Example 1

Insertion of a key 7:

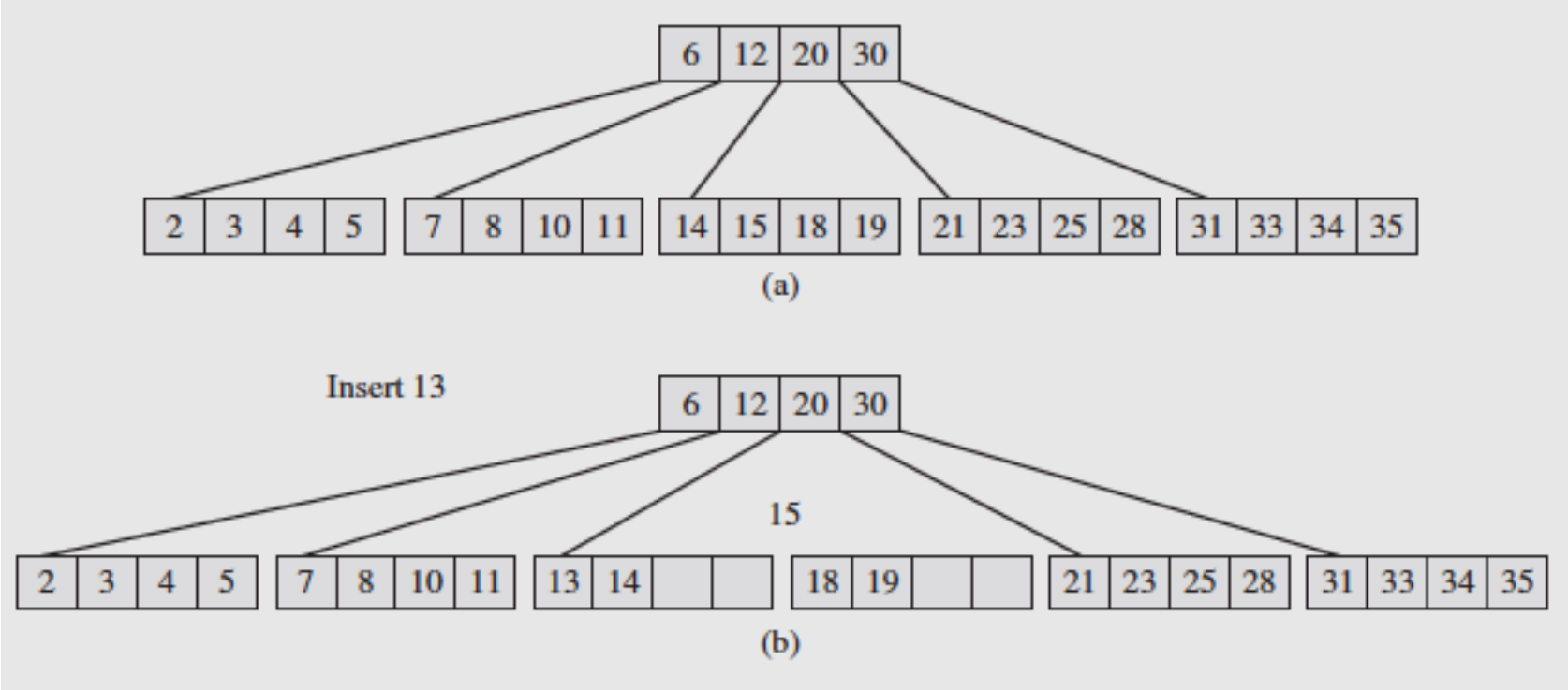


Insertion of a key 6 with splitting of a leaf:



Example 2

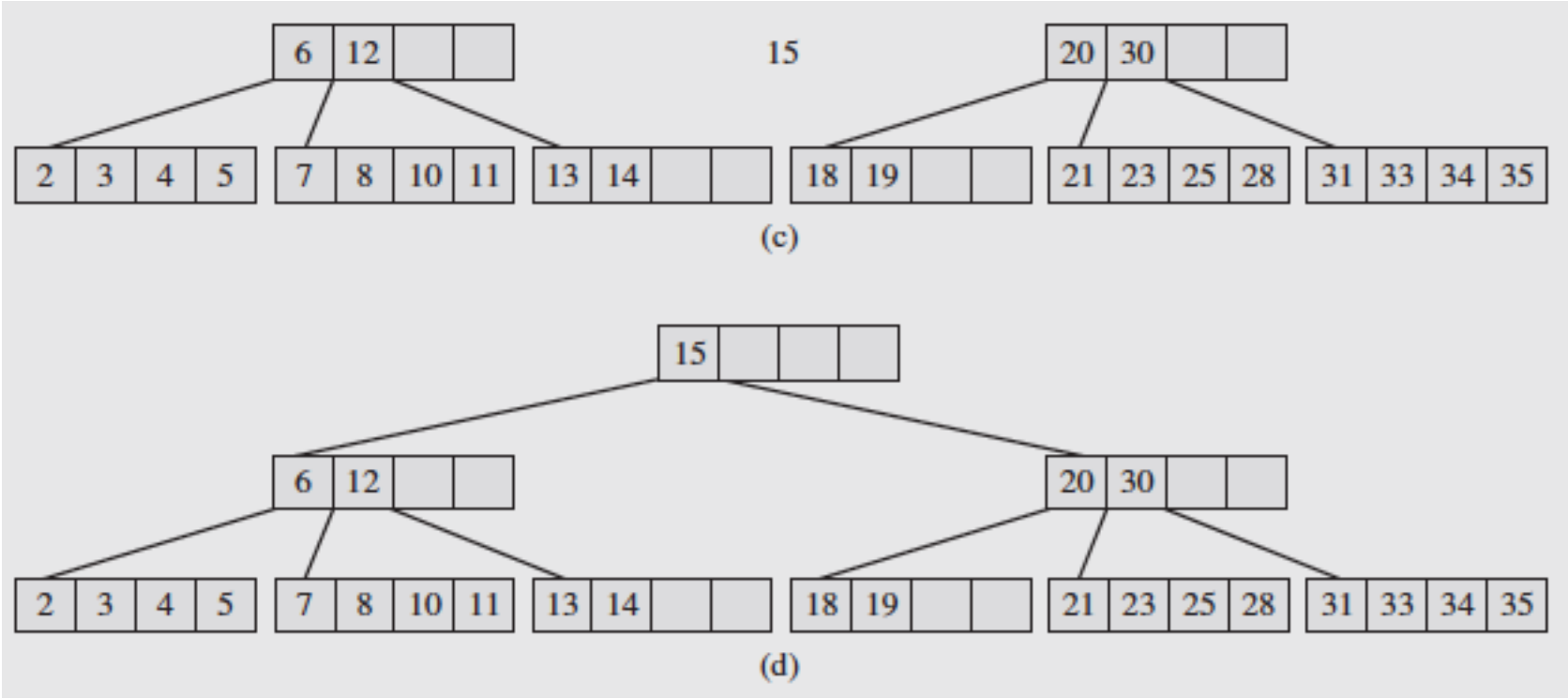
Insertion of a key 13 with splitting of a leaf:



This requires the insertion of the key 15 into the parent node

Example 2 (cont.)

Insertion of a key 15 into a non-leaf node with splitting of this node:

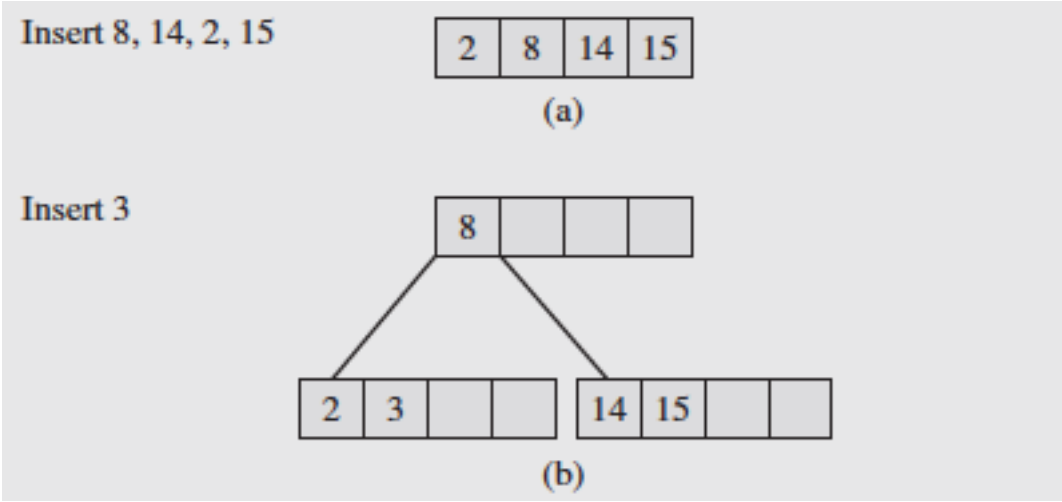


This requires the creation of a new root

Example 3

Creating a B-tree by insertion keys 8, 14, 2, 15, 3, 1, 16, 6, 5, 27, 37, 18, 25, 7, 13, 20, 22, 23, 24

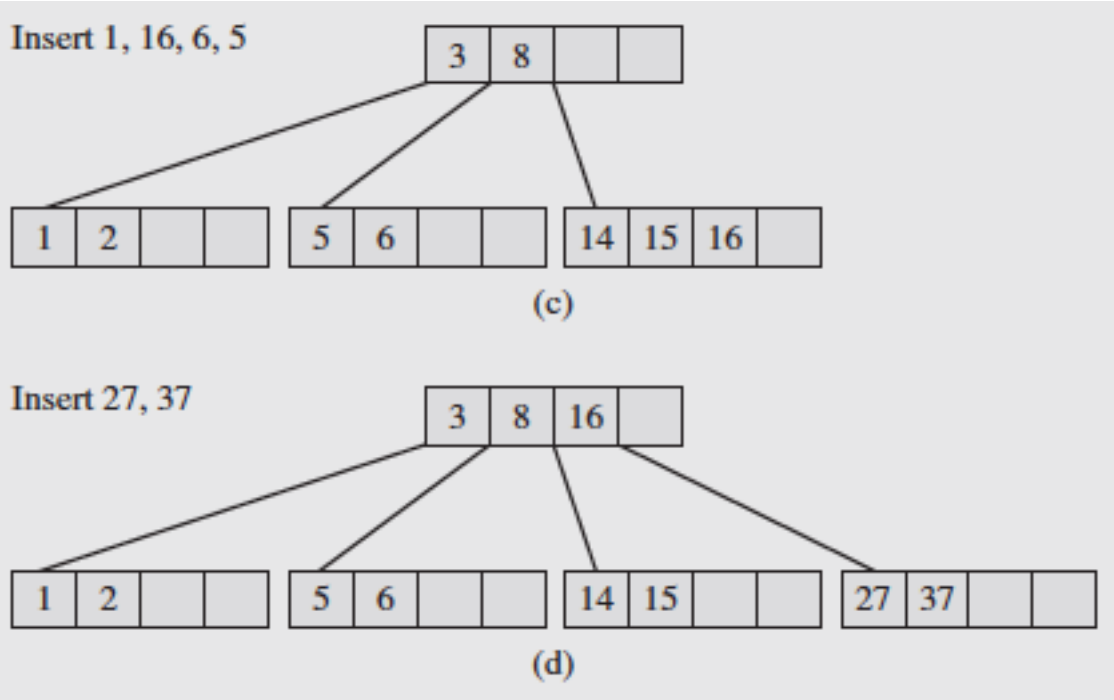
The first split (of the root/leaf) occurs with the insertion of 3:



We always neglect the insertion of a new tuple into the main data

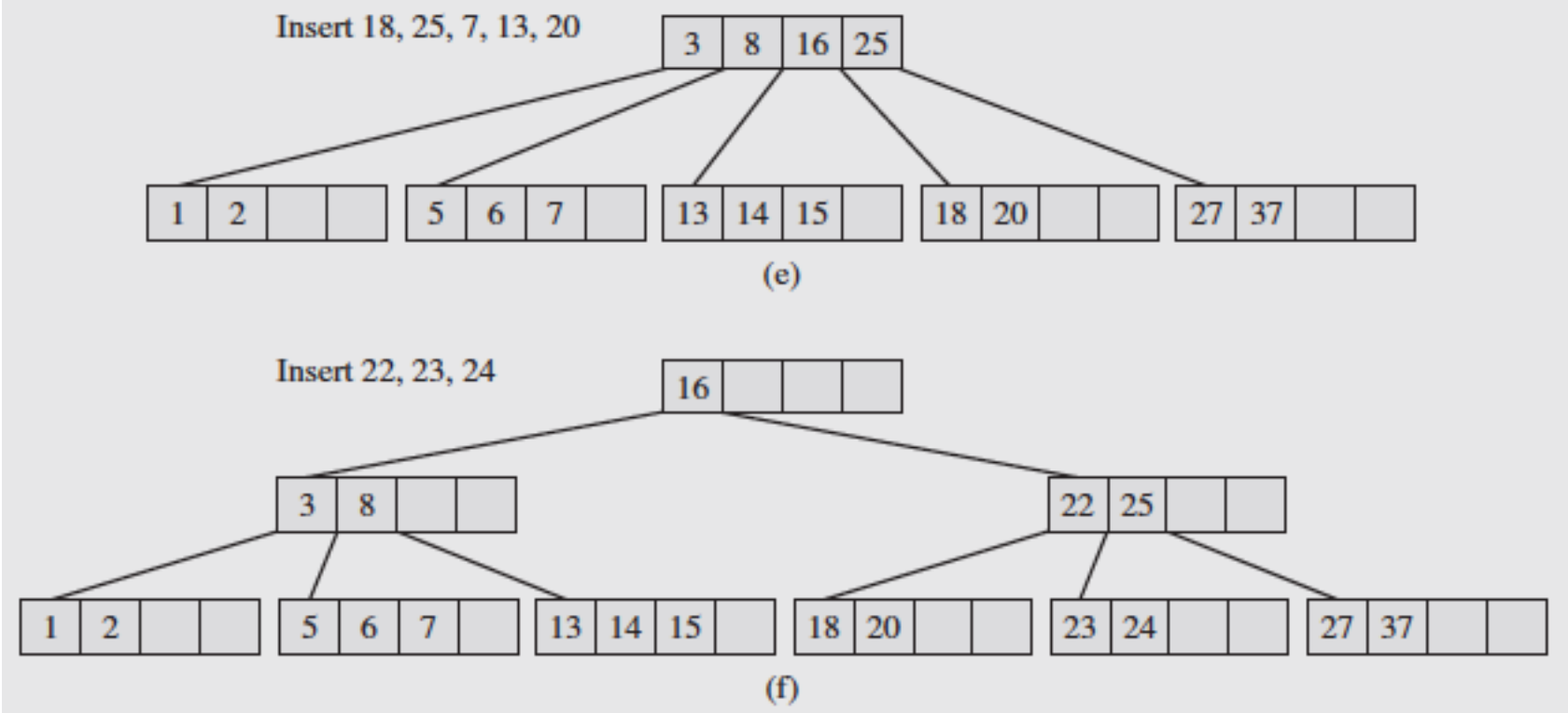
Example 3 (cont.)

The insertion of 5 leads to the next split of a leaf, and the insertion of 37 requires another leaf to be split:



Example 3 (cont.)

Finally, the insertion of 24 leads to the creation of a new root:



Complexity of Insertion into a B-tree

The (binary) search for a key in a node with s keys requires time in $O(\log(s+1))$

The insertion of a key into a node as well as the insertion of a tuple into main data requires constant time

This leads to time complexity for an insert in $O(h \cdot \log(s+1)) = O(\log n)$, as we $h \leq \log_{s+1} \frac{n+1}{2} + 1$

In addition, the height h of the tree determines the number of accesses to external storage

However, this calculation is still misleading, as splitting a node requires the creation of a new block in external storage, so it is necessary to consider, how often such a split may occur

Also the sorting of the data in blocks in external storage has to be taken into account

Complexity of Splitting Nodes

How often does a splitting of a node occur?

Assume that the B-tree has a height h and p nodes

As the height is h, there must have been $h-1$ splittings of the root, each producing 2 new nodes—this accounts for $2(h-1)+1$ nodes

If another node is split, it leads to exactly one additional node

As there are $p-2(h-1)-1$ nodes created this way, the total number of splits is $p-2(h-1)-1+(h-1)=p-h$.

Also, the number of keys in a B-tree of order m with p nodes is at least $1+(p-1)(\lceil m/2 \rceil - 1)$.

Complexity of Splitting Nodes / cont.

Thus, the rate of splits with respect to the number of keys in a B-tree amounts to

$$\frac{p-h}{1+(p-1)(\lceil m/2\rceil-1)} = \frac{1}{\frac{1}{p-h}+\frac{p-1}{p-h}(\lceil m/2\rceil-1)}$$

We have $\lim_{p\to\infty}\frac{1}{p-h}=0$ and $\lim_{p\to\infty}\frac{p-1}{p-h}=1$

Hence, the average probability of a split in case of an insertion becomes

$$\frac{1}{\lceil m/2\rceil-1}$$

This shows that on average we have to expect $\frac{1}{\lceil m/2\rceil-1}$ creations of new blocks in external storage associated with a single insertion, which adds to the time complexity of the insertion

Deletion from a B-Tree / 1

In principle, deletion from a B-tree works analogous to insertion with nodes being merged, when the number of children falls below the minimum $\lceil m/2 \rceil$

Given a key k search the tree for a node, in which (k, p) appears as a key—so let's assume that $k = k_i$ and $p = p_i$

We have to distinguish between the deletion from a leaf and the deletion from a non-leaf node

Case 1. Deletion from a leaf. If after deletion there are still at least $\lceil m/2 \rceil - 1$ keys left in the node, then simply shift the keys $(k_{i+1}, p_{i+1}), \ldots, (k_s, p_s)$

If after deletion we have an underflow, i.e. there are less than $\lceil m/2 \rceil - 1$ keys left in the node N, then we have to consider several subcases

Deletion from a B-Tree / 2

- **Case 1.1.** If there is a left or right sibling N' with more than $\lceil m/2 \rceil - 1$ keys, then consider the key (k'_j, p'_j) in the parent such that the j'th and $(j+1)$st children are the two nodes N and N' under consideration

  Let $(k^*_1, p^*_1), \ldots, (k^*_{s'}, p^*_{s'})$ be the keys in N, N' and (k'_j, p'_j) in order, i.e. $k^*_i < k^*_{i+1}$ for all i

  Take $q = \lceil s'/2 \rceil$, build two nodes N_1, N_2 with the keys $(k^*_1, p^*_1), \ldots, (k^*_{q-1}, p^*_{q-1})$, and $(k^*_{q+1}, p^*_{q+1}), \ldots, (k^*_{s'}, p^*_{s'})$, respectively

  Then insert the key (k^*_q, p^*_q) together with pointers to N_1 and N_2 into the parent node

Deletion from a B-Tree / 3

- **Case 1.2.** If there is no such sibling, simply merge two siblings N and N' and the separating key (k'_j, p'_j) in the parent node

  Remove (k'_j, p'_j) in the parent node and its two adjacent pointers to children by a single pointer to the merged node

  Treat the parent node as if it were a leaf node, i.e. check if the deletion of (k'_j, p'_j) causes an underflow and if yes, proceed in exactly the same way

- **Case 1.3.** Consider the special case, where there is no such sibling and the separating key (k'_j, p'_j) is the only key in the root

  In this case the node N and its sibling N' were the only successors of the root, and after the merge the new node becomes the new root

Deletion from a B-Tree / 4

Case 2. Deletion from a non-leaf. In this case assume $k = k_i$, then search either for the largest key $k^* = k_{max}$ in the subtree rooted at the i'th child or for the smallest key $k^* = k_{min}$ in the subtree rooted at the $(i+1)$st child

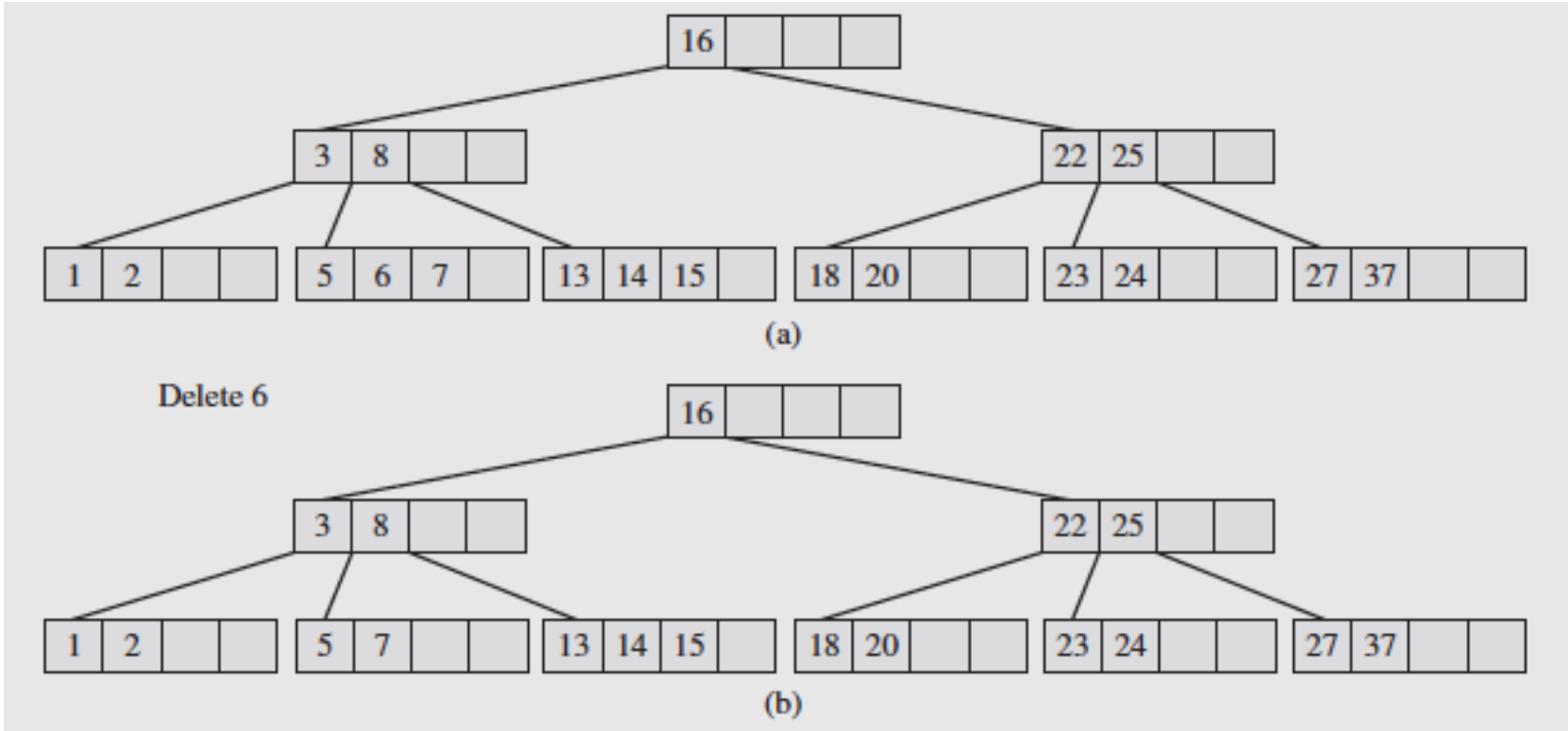
Clearly, a key (k^*, p^*) can only appear in a leaf

Then replace (k, p) by (k^*, p^*) in the non-leaf node and delete (k^*, p^*) from its leaf

Follow the same prcedure as for the deletion of any other key in a leaf

Example

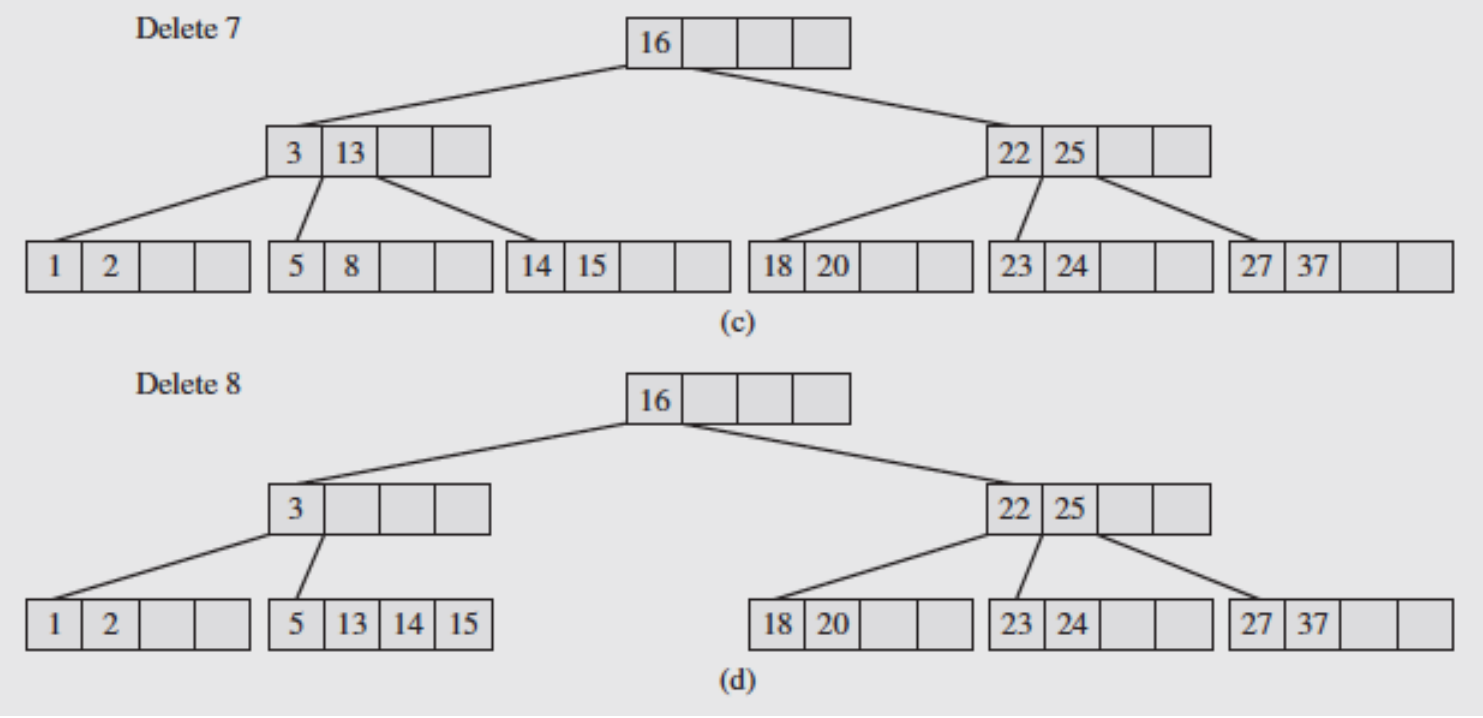
Deletion of a key 6 only affects the leaf, in which the key is found:



A follow-on deletion of the key 7 will require a redistribution

Example / cont.

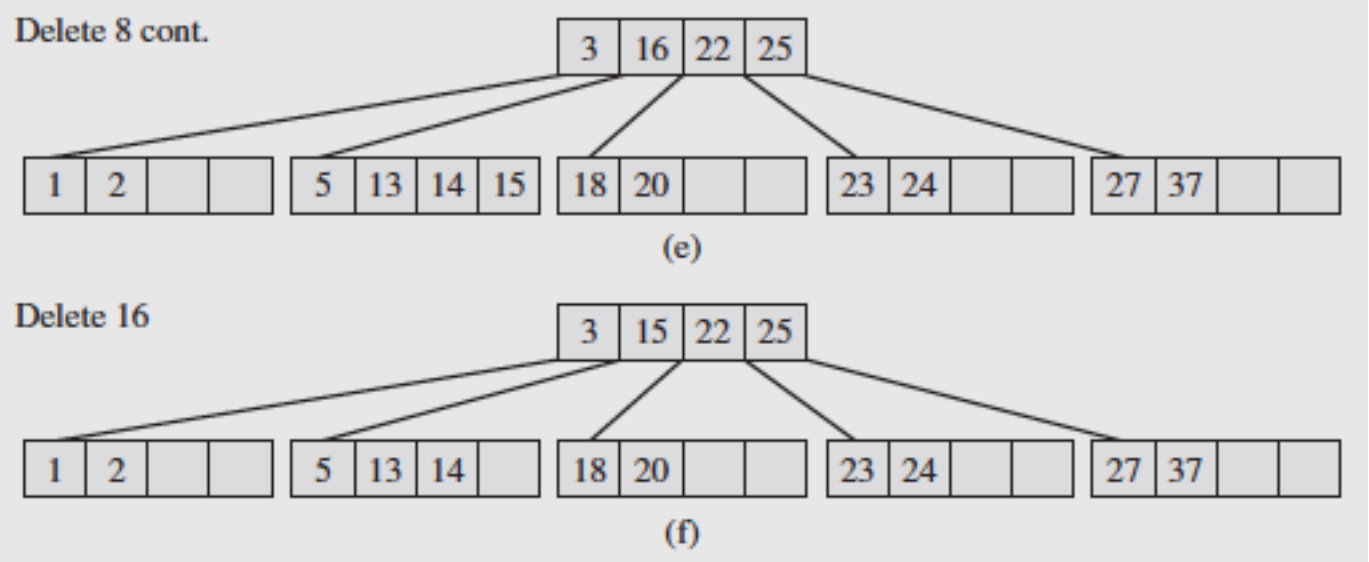
In the case of the deletion of key 7 causing an underflow in the leaf the leaf, its right sibling togather with the separator key 8 in the parent are merged, split equally with a new separator key moved to the parent



A follow-on deletion of key 8 causes again an underflow and requires a reorganisation causing just a merge of two leaves and a deletion in the parent node

Example / cont.

As the deletion in the non-leaf node caused an underflow, the node is merged with its right sibling and the separator key of the parent, creating in this case a new root



Finally, the deletion of the key 16 in a non-leaf node requires the two children to be merged and redistributed with a new separator key moved to the parent

Complexity of Deletion from a B-Tree

As for insertion we see that the deletion of key from a B-tree requires time in $O(\log n)$, but it is more decisive to ask how often a merge of two nodes occurs

We can concentrate on the deletion from a leaf (case 1), as deletion from a non-leaf node (case 2) has been reduced to this

As in the case of insertion we have $p - h$ merges of nodes, when deleting nodes from a B-tree of height h with p nodes to end with a trivial B-tree

From this we infer that the average probability of a merge is $\frac{1}{\lceil m/2 \rceil - 1}$

That is, on average we have to expect $\frac{1}{\lceil m/2 \rceil - 1}$ deletions of blocks in external storage associated with a deletion from a B-tree

The Fill-Factor

According to our definition the fill-factor of a B-tree node is always between 1/2 and 1

If nodes are not full, it may happen that quite a lot of space (almost 50%) is wasted

However, using random sequences of insertions and deletions the average fill-factor is ≈ 0.69

B*-trees are a modification of B-trees that require a minimum fill-factor of 2/3

With B*-trees splitting is even less frequent, but then two nodes are split into three new nodes, when splitting becomes necessary

Likewise for deletions three nodes may be merged and split into two nodes

8.3 Implementation of B-Trees

An implementation of B-trees will be provided for the labs; here we sketch the fundamental considerations

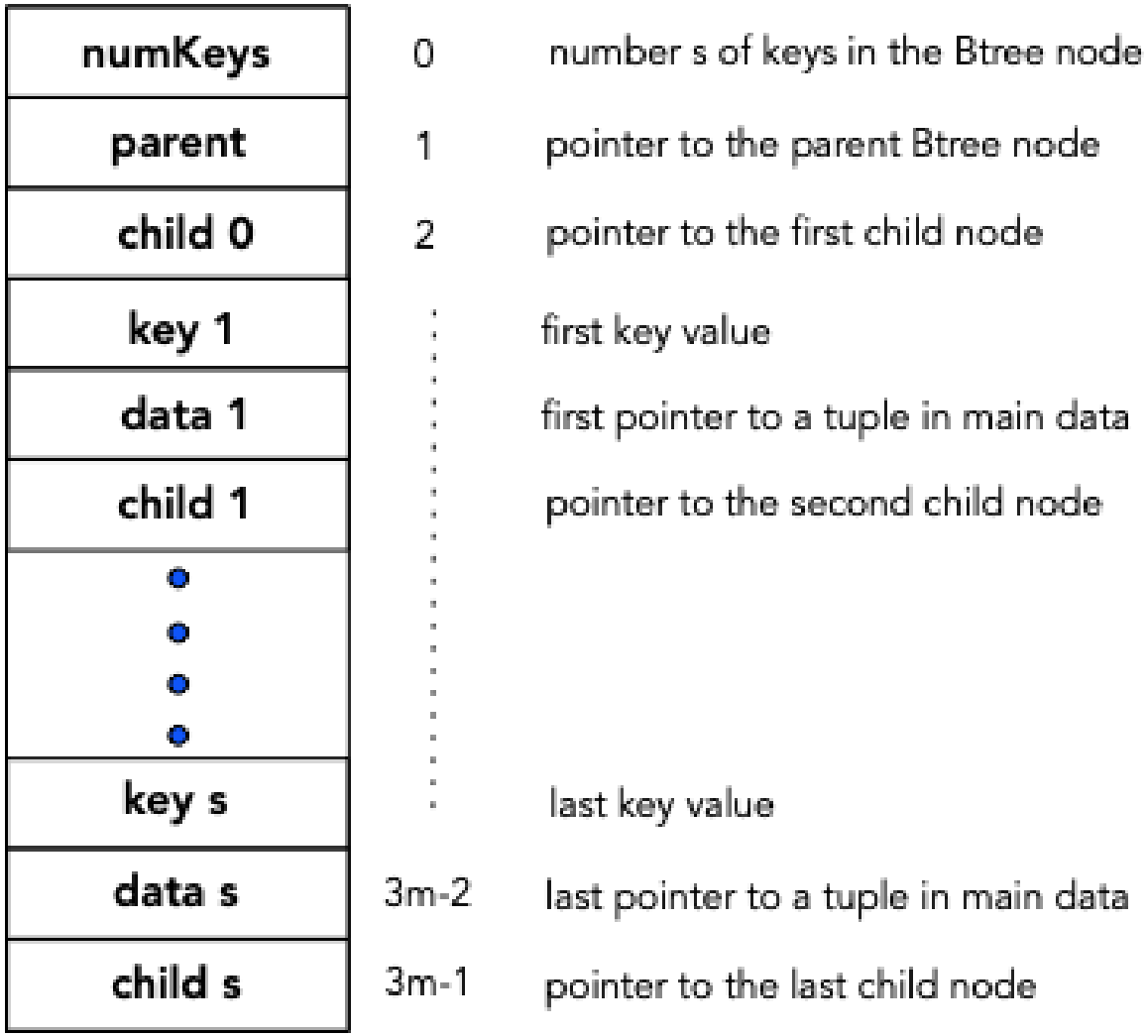
For the implementation of B-trees of order m we first have to decide on the data structure for the nodes

A straightforward choice is the use of an array, where we store

- the s key values $k_1, \ldots, k_s$
- the s pointers $p_1, \ldots, p_s$ to main data associated with these keys
- the pointers to the $s + 1$ children with $s + 1 \leq m$
- in addition, storing the number s of keys and a pointer to the parent node in the tree (None for the root) seems useful

This gives rise to an array of size $3m$

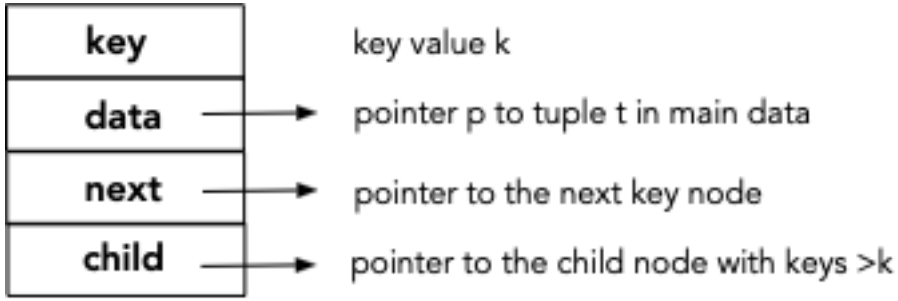
Array Representation of a B-Tree Node



An Alternative B-Tree Node Representation

The array representation has the advantage that it enables binary search, but when a key is deleted, then the following keys have to be shifted

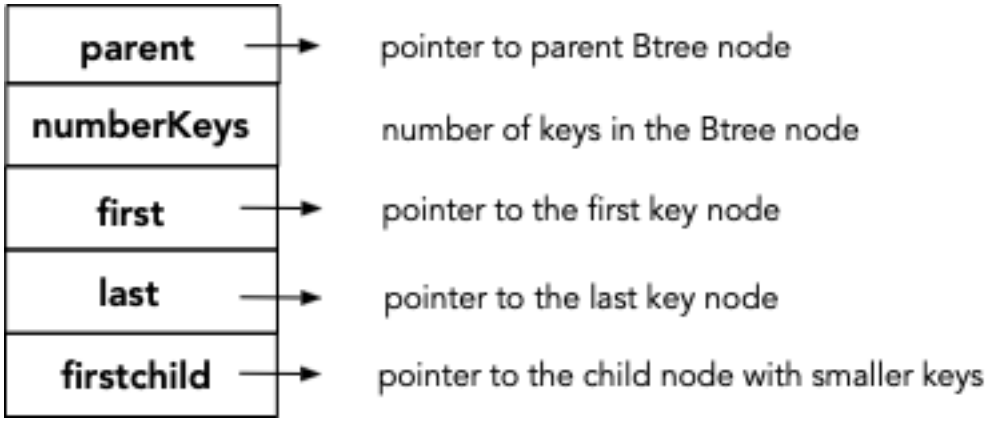
Alternatively, we can use a form of linked list to represent a B-tree node, in which case we need representation for the *key nodes* in such a list



In this representation we combine the key (k_i, p_i) together with the pointer to the child for keys k with $k_i < k < k_{i+1}$

An Alternative B-Tree Node Representation / cont.

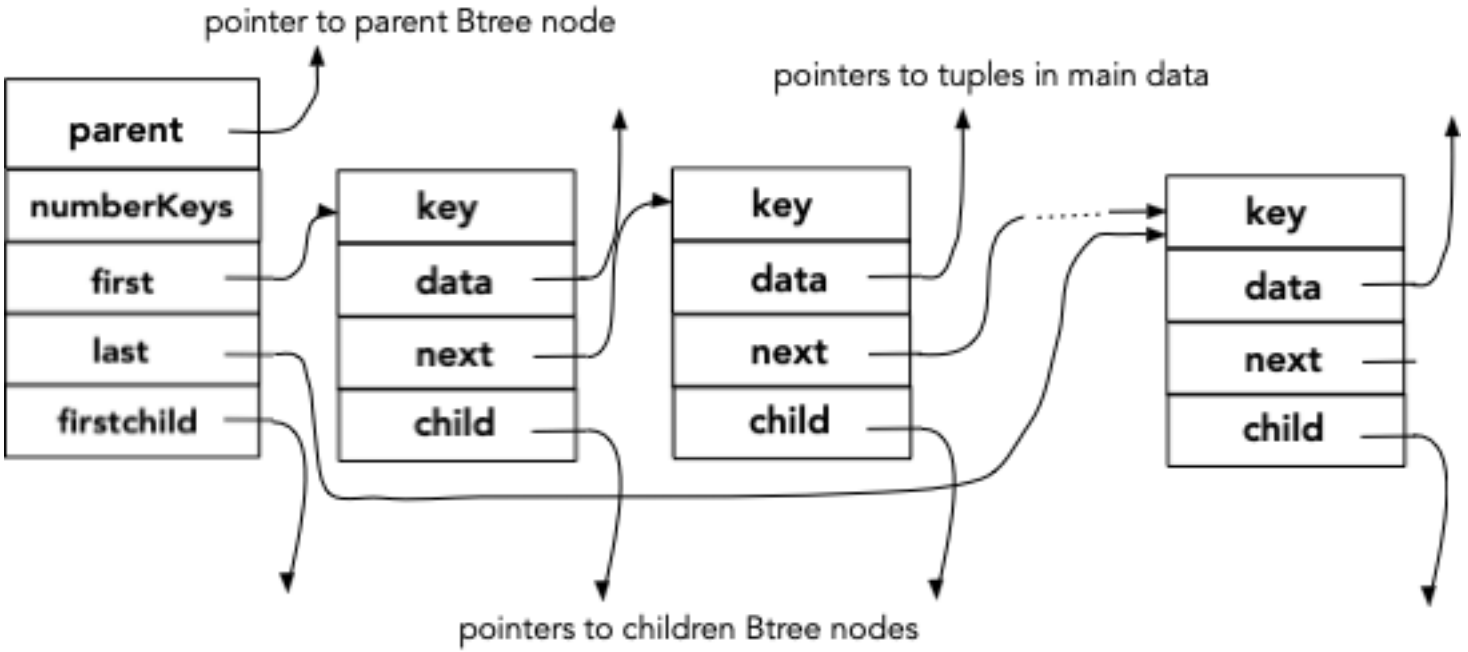
In addition, we then need to have a different first node of the linked list, where we store the pointer to the parent node and the number s of keys



The *first*, *last* and *next* pointers are used to link the different key nodes together

An Alternative B-Tree Node Representation / cont.

Putting these considerations together we obtain the following data structure for a B-tree node



The definition of a class is then rather straightforward

Main Data Organisation / Merging

In lieu of implementing the external memory management, for which a lower-level language would be required

We simply represent the main data by an array of blocks, each with a fixed blocksize, using a BLOCK class

The most tedious part concerns the merge of two nodes together with the separator from the (common) parent node

We have to discover two cases depending on which of the two nodes contains fewer keys—definitely, the number of keys in the two nodes is different

Deletion from a BTreeNode / Five Cases

For the *delete* operation on BTreeNode we have to distinguish four cases of underflow

- The left sibling has more than the minimum number of keys—in this case we use the auxiliary __merge function to merge the node with its left sibling, split it equally and create a new separator in the parent node
- Not the left but the right sibling has more than the minimum number of keys—in this case we use the auxiliary __merge function to merge the node with its right sibling, split it equally and create a new separator in the parent node
- The left sibling exists and has just the minimum number of keys—in this case we merge the node with its left sibling and the separator from the parent, and propagate the delete to the parent node
- Only the right sibling exists and has just the minimum number of keys—in this case we merge the node with its right sibling and the separator from the parent, and propagate the delete to the parent node

8.4 B$^+$-Trees

B-trees provide a powerful way to access data in external storage by means of a primary key

- We use a multi-way index tree, in which each key is coupled with a data pointer to external storage, where the tuple with this key is stored
- Using a minimum fill factor of 1/2 we can efficiently organise insertions and deletions of keys in a B-tree
- Using a large order for B-tree we can ensure that B-tree nodes occupy as much space as one block in external storage, which allows us to store also the index structure on external storage
- Then the low height of B-trees guarantees that the amount of paging, i.e. replacing blocks in the buffer in main memory, is kept reasonably low

Disadvantages

However, we also saw that we cannot keep tuples with keys in the same B-tree node together on the same block, because the insertion and deletion moves keys between nodes

For the same reason we cannot keep tuples stored in order (with respect to some total order $\leq$ on keys

This is particularly bad for range queries retrieving all tuples with keys between a lower and an upper bound

An alternative would be to store the data directly with their keys, i.e. in the B-tree, but this would destroy all advantages listed above, in particular, when tuples are big

The question is how B-trees could be enhanced to better support range queries without inefficiently hopping through main data storage and without giving up the major advantages of B-trees

The Idea of B$^+$-Trees

The question above leads to B^+-trees, which permit data pointers only in the leaves

If we have a sequence $c_0, k_1, c_1, k_2, \ldots, c_{s-1}, k_s, c_s$ of keys k_i and child pointers c_i in a non-leaf node, we request that following c_i we reach all records with keys k such such $k_i \leq k < k_{i+1}$ holds ($k < k_1$ for $i = 0$ and $k \geq k_s$ for $i = s$)

In this way we can store even more keys in a non-leaf node than in the case of B-trees

If we have a sequence $p_0, k_1, p_1, k_2, \ldots, p_{s-1}, k_s, p_s$ of keys k_i and data pointers p_i in a leaf, we can arrange that p_i points to a block (or a sequence of blocks) in external storage, in which we find all tuples with keys k such that $k_i \leq k < k_{i+1}$ holds ($k < k_1$ for $i = 0$ and $k \geq k_s$ for $i = s$)

In this way we do not need a key in the tree for every tuple in external storage

The Idea of B$^+$-Trees / cont.

Then we can order the tuples in each block referenced from a B$^+$-tree node according to the primary key

Having tuples in a block ordered allows us to exploit binary search, when searching for a particular tuple

In order to avoid too much overhead for sorting the blocks referenced from B$^+$-tree leaves, each block is coupled with an overflow block

Tuples are first inserted into the overflow block, so the binary search has to be coupled with a scan of the overflow block, if necessary, and sorting is only performed, when the overflow block is full

In addition, leaves are coupled by a linked list structure, which supports range queries

Clearly, in B$^+$-trees a search must always be conducted down to a leaf, and instead of direct retrieval of a tuple a search in one or two blocks becomes necessary

B^+-Tree Definition

A ***B^+-tree*** of **order** m is a tree with the following properties:

- Each non-leaf node contains a sequence $c_0, k_1, c_1, k_2, \ldots, c_{s-1}, k_s, c_s$ of s keys $k_1 < \cdots < k_s$ and $s+1$ children pointers c_i with $s \geq 1$ for the root and $\lceil m/2 \rceil \leq s+1 \leq m$ for all other nodes

- All nodes in the i'th successor tree of a non-leaf node contain only keys k with $k_{i-1} \leq k < k_i$

- Each leaf contains a sequence $p_0, k_1, p_1, k_2, \ldots, p_{s-1}, k_s, p_s$ of s keys $k_1 < \cdots < k_s$ and $s+1$ data pointers p_i to blocks in external storage with $\lceil m/2 \rceil \leq s+1 \leq m$ for all other nodes

- In addition, each leaf contains pointers *prev* and *next* to its left and right siblings, provided these exists

- All simple paths from the root to a leaf have the same length

Main Data Organisation

The data pointers in a B^+-tree leaf point to a block in main data storage rather than to a single tuple (as was the case for B-trees)

More generally, in particular for large-sized tuples, we may use sequences of blocks (so called *trains*), but we will ignore this subtlety here

This allows us to keep data in a block ordered with respect to the primary key used for the B^+-tree

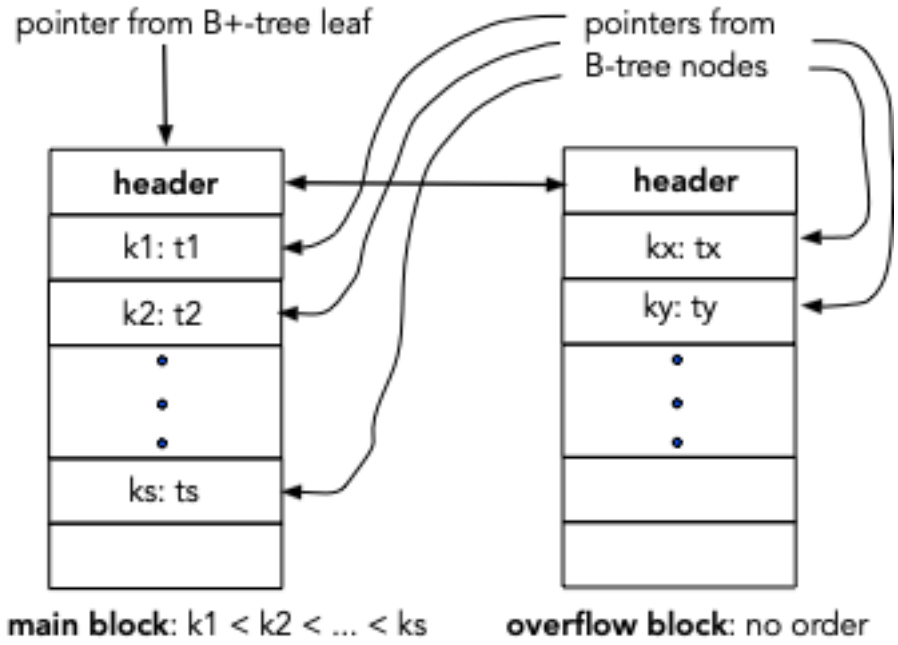
With sorted data we can exploit binary search to find a tuple for a given key, and we can easily split and merge blocks when it becomes necessary

However, order may be destroyed by every insertion and sorting is an expensive operation, hence sorting with every insertion may be too costly

To resolve this conflict between retrieval, reorganisation and insertion blocks are coupled with additional *overflow blocks*, in which data are not ordered

Then the retrieval of a tuple has to include linear search in the overflow block, and sorting is executed only occasionally

Main Data Organisation / cont.



Information to be stored in the header of a block: size (maximum numbers of tuples), number of tuples stored, number of tombstones (markers for deleted tuples), pointer to the associated overflow or main block, respectively

Tuples in such a pair of main and overflow block may also be referenced from any other index data structure, e.g. from nodes of a B-tree for a different primary key

Retrieval from Main Data

With this organisation of blocks including overflow blocks we can find a tuple, when we are given its key

Search in the B^+-tree leads to a leaf, from where we obtain the pointer to the block in main data storage

Then `find(key)` first exploits binary search in the main block: if found, the corresponding tuple can be returned

If not found, continue with a linear search in the referenced overflow block: when found the tuple is returned

If the data is further referenced from another index structure, e.g. from some node in a B-tree for a different primary key, then the data pointer in that B-tree points directly to the entry in the block (or overflow block)

For this we can exploit another operation `direct_find(pointer)`, which accesses directly the entry where the sought tuple is stored and returns this tuple

Deletion from Main Data

To delete a tuple we proceed analogously—given a key first apply combined binary and linear search in main and overflow block to locate the entry, in which the tuple is stored

If the tuple is found, mark it with a tombstone—use a PLACEHOLDER object for this—so that sorting can be avoided; information in the block headers must be updated accordingly

Nonetheless, the tuples marked with a tombstone impact on the performance of binary and linear search

When a deleted tuple is referenced from a different index data structure, e.g. from some node in some B-tree, the other key and data pointer must be deleted from that other B-tree

Same as for retrieval a delete may also be issued using a different index structure—we might receive a pointer to the tuple to be deleted from a node in a B-tree organised around a different primary key

With such a pointer as argument we can have another delete operation, which just marks the tuple at the given location with a tombstone

Insertion into Main Data

When inserting a tuple into a relation we always have all keys at hand, so we can assume that we use an associated B^+-tree to find the pointer (from a leaf) to the correct block in main data storage

We can simply insert the tuple into the overflow block at the first free place and leave the insertion into the main block for reorganisation

As long as the number of tuples in the main block is reasonably small, so that sorting is still inexpensive, we might as well insert a tuple directly into the main block and sort the entries

In both cases there is no need to update anything in the B^+-tree, unless a reorganisation of the block and its associated overflow block are executed—the data pointer in the leaf still points to the correct block

In case the data is subject to another B-tree for a different primay key, we need to determine this key of the inserted tuple and insert it together with a pointer to its location in the block or overflow block in a leaf of the B-tree

In case of direct insertion into the main block all data pointers to tuples with changed address have to be updated in the other B-tree—this shows hat this option should be handled with care

Example

Suppose we have a relation MOVIE with attributes *number*, *title*, *year*, *country*, *run_time*, *genre* with the following tuples:

```
(111,'Shakespeare in Love',1998,'UK',122,'romance')
(113,'The Cider House Rules',1999,'USA',125,'drama')
(114,'Gandhi',1982,'India',188,'drama')
(117,'American Beauty',1999,'USA',121,'drama')
(118,'Boys Dont Cry',1999,'USA',118,'drama')
(119,'Saving Private Ryan',1998,'USA',170,'action')
(123,'The Birds',1963,'USA',119,'horror')
(127,'The Matrix',1999,'USA',136,'action')
(128,'Toy Story',1995,'USA',81,'animation')
(131,'You have Got Mail',1998,'USA',119,'comedy')
(132,'Proof of Life',2000,'USA',135,'drama')
(133,'Hanging Up',2000,'USA',94,'comedy')
(136,'The Price of Milk',2000,'New Zealand',87,'romance')
(140,'The Footstep Man',1992,'New Zealand',89,'drama')
(141,'The Piano',1993,'New Zealand',121,'romance')
```

Example / cont.

Let us use *number* as primary key supported by a B^+-key—the block and its associated overflow block is to store tuples with values of *number* in the range $[100, 150)$

Assume that tuples with numbers 140, 127, 132, 118 and 119 are stored in the overflow block in this order, while all other tuples are stored in the main block in the given order

Furthermore, assume that we also use *title, year* as another primary key supported by a B-tree

Consider the insertion of a new tuple

```
(125,'Mad Max',1979,'Australia',88,'action')
```

This tuple would be stored in the overflow block in 6th position and the (other) key `('Mad Max',1979)` would be inserted together with the address of the overflow block and the index 5 (counting from 0) into the B-tree

Data Reorganisation in a Block

Reorganisation applies when either the number of tuples in the overflow block gets too high, there are two many gaps (i.e. tuples marked as being deleted) in the main block, or the total number of tuples gets too low

For all these case we use fixed threshold values or fill factors (usually between 1/2 and 2/3)

Reorganisation sorts all the tuples in the main and the overflow block, places **all** tuples into the main block leaving the overflow block empty, and changes all affected data pointers in another B-tree (if applicable)

If necessary the block is split into two, when there are too many tuples—then a new separating key must be built and inserted into the B^+-tree leaf together with a point to the new block

When there are too few tuples the block is merged with one of its siblings (when there are too few tuples) and either redistributed by splitting or eliminated

In case of an elimination a separating key and corresponding pointer to the eliminated block must be removed from the leaf of the B^+-tree

Example

Continue our previous example assuming that tuples with numbers 113, 131 and 136 have been deleted, so we have the followimng tuples in the main block

```
(111,'Shakespeare in Love',1998,'UK',122,'romance')
# (113,'The Cider House Rules',1999,'USA',125,'drama')
(114,'Gandhi',1982,'India',188,'drama')
(117,'American Beauty',1999,'USA',121,'drama')
(123,'The Birds',1963,'USA',119,'horror')
(128,'Toy Story',1995,'USA',81,'animation')
# (131,'You have Got Mail',1998,'USA',119,'comedy')
(133,'Hanging Up',2000,'USA',94,'comedy')
# (136,'The Price of Milk',2000,'New Zealand',87,'romance')
(141,'The Piano',1993,'New Zealand',121,'romance')
```

Example / cont.

Assume that some more tuples have been inserted into the overflow block, so that it contains the following tuples:

```
(140,'The Footstep Man',1992,'New Zealand',89,'drama')
(127,'The Matrix',1999,'USA',136,'action')
(132,'Proof of Life',2000,'USA',135,'drama')
(118,'Boys Dont Cry',1999,'USA',118,'drama')
(119,'Saving Private Ryan',1998,'USA',170,'action')
(125,'Mad Max',1979,'Australia',88,'action')
(139,'Strictly Ballroom',1992,'Australia',94,'comedy')
(109,'My Mother Frank',2000,'Australia',95,'comedy')
(145,'I Know What You Did Last Summer',1997,'USA',100,'horror
(101,'Cruel Intentions',1999,'USA',95,'romance')
(103,'Wild Things',1998,'USA',108,'crime')
```

Example / cont.

Suppose that in this situation we have to reorganise the data, and asume that we want to have not more than 12 tuples in one ordered main block

So we get two blocks (both with empty associated overflow block); the first block contains the following tuples:

```
(101,'Cruel Intentions',1999,'USA',95,'romance');
(103,'Wild Things',1998,'USA',108,'crime')
(109,'My Mother Frank',2000,'Australia',95,'comedy')
(111,'Shakespeare in Love',1998,'UK',122,'romance')
(114,'Gandhi',1982,'India',188,'drama')
(117,'American Beauty',1999,'USA',121,'drama')
(118,'Boys Dont Cry',1999,'USA',118,'drama')
(119,'Saving Private Ryan',1998,'USA',170,'action')
(123,'The Birds',1963,'USA',119,'horror')
```

A new separating key is 125, which is inserted together with a pointer to the new block into the leaf of the B^+-tree

Example / cont.

The second block contains the following remaining tuples:

```
(125,'Mad Max',1979,'Australia',88,'action')
(127,'The Matrix',1999,'USA',136,'action')
(128,'Toy Story',1995,'USA',81,'animation')
(132,'Proof of Life',2000,'USA',135,'drama')
(133,'Hanging Up',2000,'USA',94,'comedy')
(139,'Strictly Ballroom',1992,'Australia',94,'comedy')
(140,'The Footstep Man',1992,'New Zealand',89,'drama')
(141,'The Piano',1993,'New Zealand',121,'romance')
(145,'I Know What You Did Last Summer',1997,'USA',100,'horror
```

For almost all tuples the location, where they are stored, has been changed, which needs to be updated in the B-tree centred around `(title,year)`-keys

E.g. for the key `('Cruel Intentions',1999)` the tuple is no longer stored in the overflow block, and for the key `('The Piano',1993)` the tuple appears in the newly created block

Insertion Using a B$^+$-Tree

Let us now look into the insertion of a new tuple when using a B$^+$-tree instead of a B-tree

So assume we are given a tuple t to be inserted such that the key for the B$^+$-tree is k

Then we search the B$^+$-tree for k, which differs from the corresponding search in a B-tree by two small aspects:

- If k appears in a non-leaf node, the search is not finished; instead follow the pointer to the child node with key values $\geq k$
- Even if k appears in a leaf, it does not imply that a tuple with the key k exists in main data storage
- If k does not appear in a leaf, it does not imply that a tuple with the key k does not exist in main data storage

Insertion Using a B$^+$-Tree / cont.

When a leaf has been reached, follow the pointer to a block in main data storage for tuples with keys in the range $[k_1, k_2)$ $(k_1 \leq k \leq k_2)$ and search for k in the block and associated overflow block

Without splitting of a blocksin main data storage () the insertion does return at most an acknowledgement

After splitting a new separator key together with a pointer to the new block is returned—these are inserted into the leaf

Then proceed in the same way as for B-trees, i.e. if a node in the tree is split due to overflow, determine a separator key and insert it together with a pointer to the new node into the parent node

The only difference is that a non-leaf node does not contain data pointers

If a new root is created, the information about the B$^+$-tree is updated in the same way as for B-trees

Example

In our example the last inserted tuple was

```
(103,'Wild Things',1998,'USA',108,'crime')
```

So the search led to a leaf with the entries $\ldots, 100, p, 150, \ldots$ with a data pointer p to the block, into which the tuple needed to be inserted

The insertion triggered sorting and splitting of the block returning the new separator key 125 and a pointer p' to the new block

So we insert these into the leaf, which becomes $\ldots, 100, p, 125, p', 150, \ldots$

If any splitting becomes necessary, we proceed as with B-trees

Note that could as well have had the case that the leaf contains only $\ldots, 100, p$, while the right sibling leaf start with $p'', k', \ldots$ with $k' \geq 150$—this does not change anything

Handling of Other Index Structures

If there is another B-tree for another primary key on the same relation, then the insertion first produces a new tuple t in main data storage, and this tuple determines its other key k'

Then we have to insert k' together with a pointer p' to the location of the new tuple into the B-tree in a leaf

For this we can use the insert operation on B-trees, which requires a slight modification, as no tuple needs to be given as input and no insertion in main data storage is required

If the data is reorganised by sorting or splitting of a block, we obtain new data pointers for all keys in the B-tree associated with tuples in this block, then all these data pointers need to be updated in the B-tree

Note that though this does not require any splitting or merging of B-tree nodes, it is nonetheless an expensive operation

In practise it is therefore preferred not to use a second primary key with a B-tree index structure

Deletion Using a B$^+$-Tree

Now consider the deletion of a tuple using a primary key k, when a B$^+$-tree index structure is used

Again we search the B$^+$-tree for the key k, until we find (in a leaf) a data pointer to a block with tuples having key values in $[k_1, k_2)$ such that $k_1 \leq k < k_2$ holds

Then use the delete operation on the block to mark the tuple with key k as deleted

If the deleted tuple t is referenced from another B-tree, determine the corresponding (other) key k' and its address p' and delete (k', p') from the B-tree

We still have to consider the case, where the deletion triggers a merge of blocks

Deletion Using a B^+-Tree / cont.

After merging two possible cases arise:

- The block was merged with the left or right sibling and split again, so a new separator key together with the key that needs to be replaced is returned

  In this case we only have to update the leaf replacing one key by another

- The block was merged with the left or right sibling, so a key and data pointer that need to be deleted from the B^+-tree are returned

  In this case we delete the returned key and data pointer from the leaf

  If necessary, deletions are propagated upwards the B^+-tree, which is done is the same way as for B-trees with the same subtle difference that no data pointers need to be considered

The Python implementation for B^+-trees will be handled in the labs

Example

Consider again our MOVIE example assuming that after several deletions we have the following data in the main block (and and empty overflow block):

```
# (101,'Cruel Intentions',1999,'USA',95,'romance')
(103,'Wild Things',1998,'USA',108,'crime')
# (109,'My Mother Frank',2000,'Australia',95,'comedy')
# (111,'Shakespeare in Love',1998,'UK',122,'romance')
# (114,'Gandhi',1982,'India',188,'drama')
(117,'American Beauty',1999,'USA',121,'drama')
# (118,'Boys Dont Cry',1999,'USA',118,'drama')
# (119,'Saving Private Ryan',1998,'USA',170,'action')
(123,'The Birds',1963,'USA',119,'horror')
```

We merge this block with its right sibling

Example / cont.

Assume that the right sibling contains the following tuples (and its associated overflow block is empty):

```
# (125,'Mad Max',1979,'Australia',88,'action')
# (127,'The Matrix',1999,'USA',136,'action')
# (128,'Toy Story',1995,'USA',81,'animation')
(132,'Proof of Life',2000,'USA',135,'drama')
# (133,'Hanging Up',2000,'USA',94,'comedy')
(139,'Strictly Ballroom',1992,'Australia',94,'comedy')
# (140,'The Footstep Man',1992,'New Zealand',89,'drama')
# (141,'The Piano',1993,'New Zealand',121,'romance')
# (145,'I Know What You Did Last Summer',1997,'USA',100,'horro
```

Example / cont.

After merging we just have one block with the following tuples (and an empty associated block):

```
(103,'Wild Things',1998,'USA',108,'crime')
(117,'American Beauty',1999,'USA',121,'drama')
(123,'The Birds',1963,'USA',119,'horror')
(132,'Proof of Life',2000,'USA',135,'drama')
(139,'Strictly Ballroom',1992,'Australia',94,'comedy')
```

Then the separating key 125 and the pointer to the right sibling are deleted from the B^+-tree leaf

Furthermore, the location of tuples with (other) keys `('Wild Things',1998)`, `('American Beauty',1999)`, `('The Birds',1963)`, `('Proof of Life',2000)` and `('Strictly Ballroom',1992)` has changed, so the data pointers in the corresponding B-tree must be updated

8.5 Index Structures for Secondary Keys

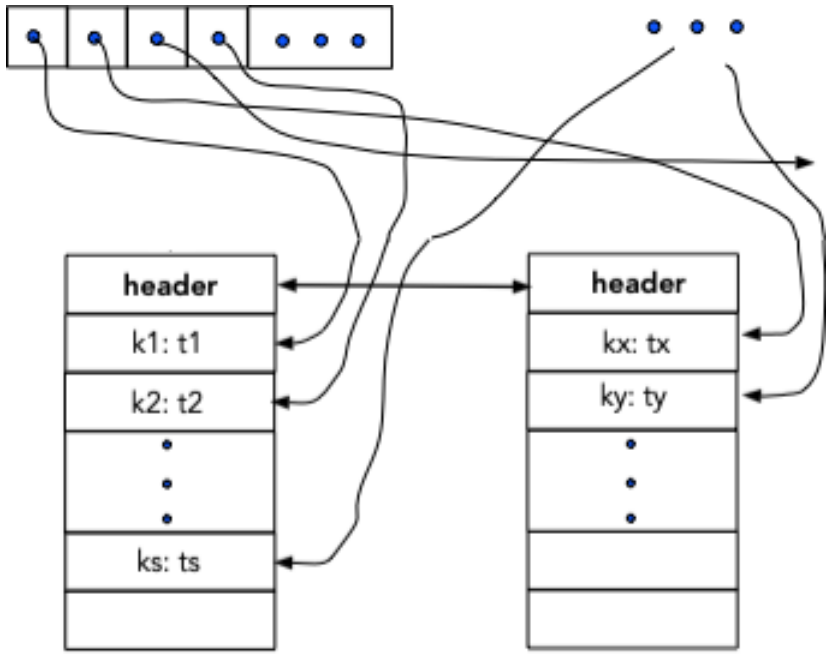
Let us now look into the issue of retrieving, inserting and deleting tuples using a secondary key, i.e. the number of tuples t with the same key k is arbitrary

In principle, we can still use a B-tree to search for a given key k, but there cannot be just a data pointer to all tuples with this key

- Storage of all tuples with the same secondary key k together in one block (or neighbouring blocks) is impossible for the same reason we used for single tuples accessed from a B-tree
- Using a linked list structure is likewise not recommended, as it would require to through the blocks (with a lot of paging) to retrieve all tuples with key k
- Therefore, we need a separate data structure, from which we can access the different tuples with key k

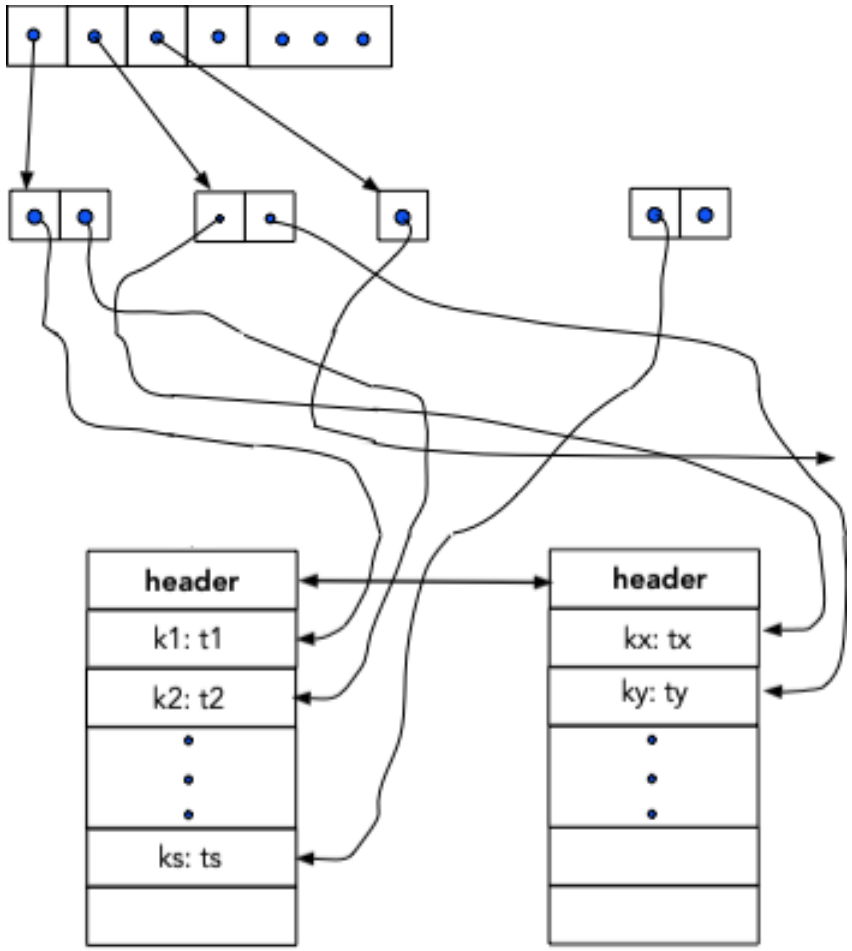
Spiders

Spiders (also known as **inversion technique**) provide one array per key k with data pointers as entries



It might even be advantageous to organise spiders in two levels pointing first to a block and then (optionally) to the tuples in that block, which leads to **block-centred spiders**

Block-Centred Spiders



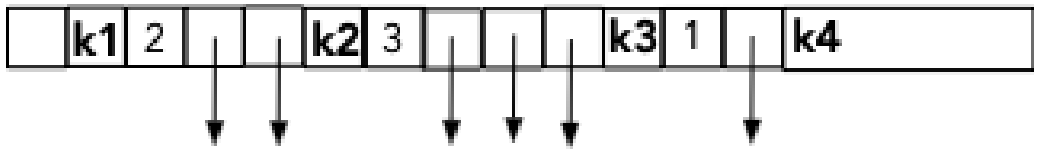
Spiders with B$^+$-trees

We might as well integrate the spiders of several keys $k_1 < \cdots < k_r$ in a single data structures

For each key k_i we take a list with

- first the key k_i itself
- second the number n_i of data pointers to tuples with key k_i
- then n_i data pointers

Then concatenate these lists for $k_1 < \cdots < k_r$ and use them as leaves in a B$^+$-tree



8.6 Multi-Dimensional Access Structures

Section 6 of this course contained an in-depth treatment of access structures to (relational) databases

All structures we discussed are one-dimensional in the sense, i.e. we assumed some uniform set of keys

However, we commonly have to deal with the situation that the key is defined by multiple attributes

In many cases, in particular in our sample databases, we could simply use an order on tuples to deal with these cases

However, if we have range conditions on multiple attributes, there is no way that the access will be sufficient

A range condition like $a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2$ can be dealt with using a modified range query with a condition $(a_1, b_1) \leq (A, B) \leq (a_2, b_2)$ followed by a selection operation

Spatial Data

However, this only makes sense for attributes that are unrelated

If we have to deal with spatial data, we may be interested in points or higher-dimensional objects in the data space, where the spatial relationships need to be preserved

We are therefore interested in dedicated multi-dimensional access structures, where all attributes in a key contribute equally to a search

This requires to reconsider the organisation of the data space such that spatial relationships are taken into account

In particular, we will look into the gridfile method for dimension refinement

Quad-Trees

An easy way to organise a two-dimensional data space is provided by **quad-trees**, which exploit quadrants

Here the keys are pairs (k_1, k_2), where $k_i \in T_i$ and T_i is a totally-ordered set $(i = 1, 2)$

Each key (k_1, k_2) provides a natural tiling of the data space into four regions:

NW. $\{(a, b) \mid a \leq k_1 \wedge b > k_2\}$ (north-west)

NE. $\{(a, b) \mid a > k_1 \wedge b > k_2\}$ (north-east)

SW. $\{(a, b) \mid a \leq k_1 \wedge b \leq k_2\}$ (south-west)

SE. $\{(a, b) \mid a > k_1 \wedge b \leq k_2\}$ (south-east)

Quad-Trees / cont.

This tiling allows us to create a tree with constant degree 4

Nodes in the tree contain a key (k_1, k_2), pointers to the four quadrants NW, NE, SW and SE defined by the key, and additional attributes associated with this key

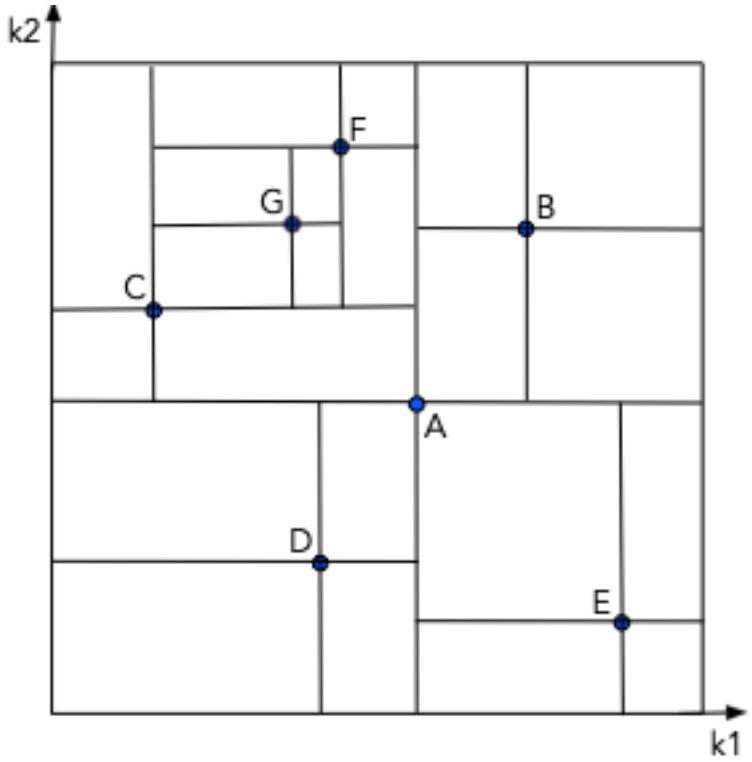
Any quadrant indicated by a key (k_1, k_2) can only contain objects in the quadrant of its parent

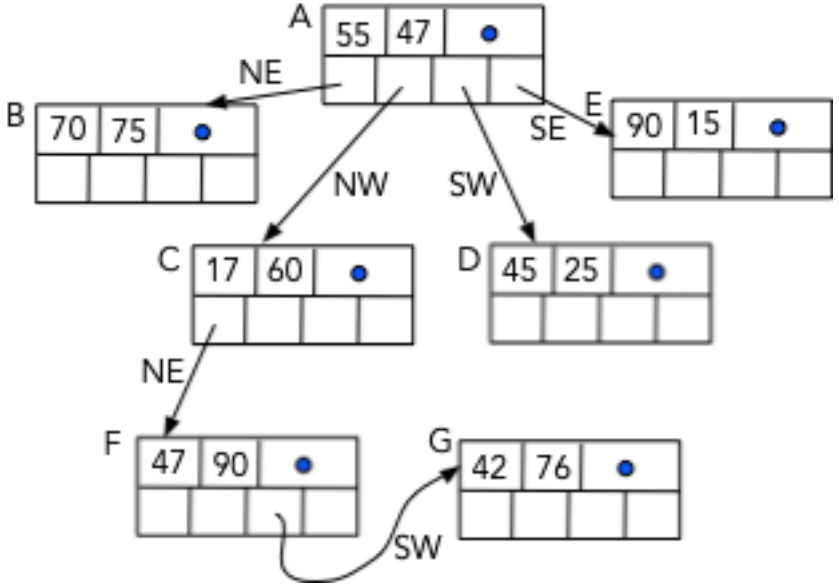
In this way a quad-tree provides a hierarchical fine tiling of the whole data space, and queries need to compare values of bot attributes of the key

Quad-trees can be easily generalised to oct-trees and hex-trees for three and four dimensions

Example

The left shows points in the data space and the right shows the corresponding quad-tree





k-d B-Trees

We have already seen the data structure of k-dimensional trees, which are organised analogous to binary search trees, but alternate on each level the attribute for the selection

k-d B-trees combine the ideas from k-d trees with those of B^+-trees

k-d B-trees inherit from B^+-trees the fill factors, the balancing property as well as the fact that only the leaves link to the main data

k-d B-trees inherit from k-d trees the alternating attributes used for the keys

In this way a hierarchical tiling (as for quad-, oct- and hex-trees) is achieved and combined with an efficient index structure inherited from B^+-trees

8.7 Journal Data Structure

The index structures we considered so far (B/B+-trees, Quad-Trees, k-d B-trees) are used to optimise the access to bulk data, in particular when stored on external storage

Assuming magnetic disks the bottleneck is the slow access due to the presence of mechanical parts, so the objective is to minimise the swapping of pages

This changes, when **flash memory** (solid state) is used instead of magnetic disks

In this case we trade the faster access to externally stored data for the problem to cope with **out-of-place** updates

In flash memory a block first has to be erased, before data can again be written onto it—we commonly talk about **eraseblocks** (**EB**s) as unit of storage on flash memory

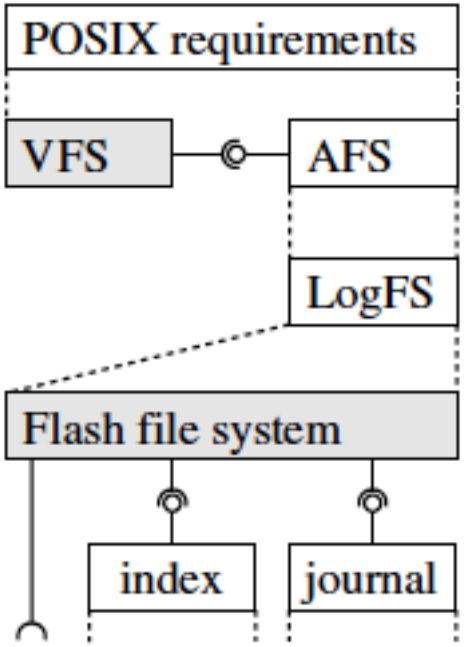
In the long run, erase operations actually destroy the EBs of a solid state drive

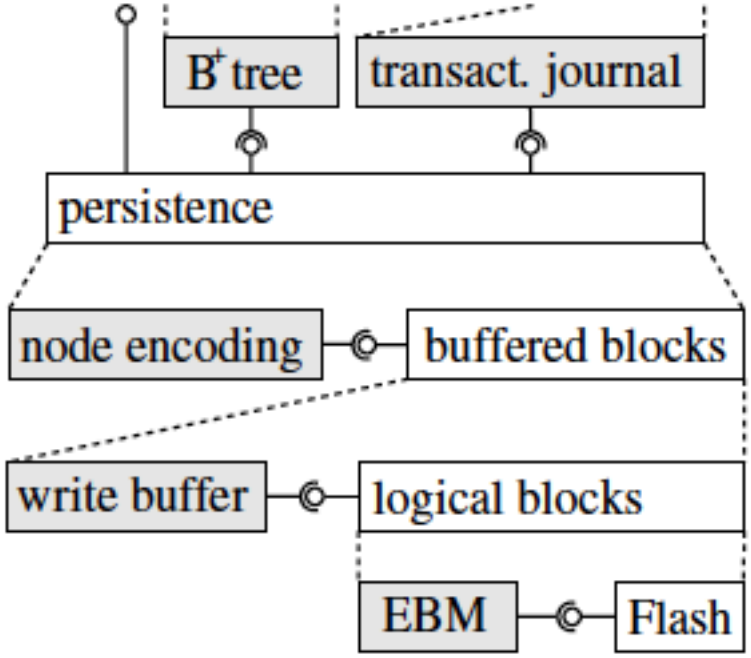
General Requirements

The standard for flash file management is POSIX, which combines a **virtual file system switch** (VFS) and an **abstract file system specification** (AFS)

VFS realises concepts common to all FS implementations such as path traversal—such a VFS exists in Linux and other operating systems

Upper and lower levels of the file system look as follows (to be explained):





Storage Areas

The storage is organised in **logical eraseblocks** (LEBs) that map to the physical EBs

There are six fixed storage areas:

- The **superblock** (covering one LEB) contains general parameters of the file system; it is rarely or never updated

- The **master node** stores the position of all flash structures that are not at fixed positions

  For recovery reasons the master node is doubled in two fixed LEBs

  If the master node is updated it is written sequentially to a new position in the corresponding LEBs

  An update of the master node becomes necessary, when a new index is committed

Index and Journal

To locate files (or EBs) on flash memory, an **index** structure called the **wandering tree** is used

The wandering tree is in fact a B+-tree, so the leaves contain pointers to the EBs, in which the real data is stored

In case of an in-place update to the index B+-tree, all EBs storing affected nodes from a leaf up to the root would need to be erased and rewritten

As this is highly inefficient (and would fast destroy the flash storage), updated nodes of the B+-tree are stored in a separate list called the **journal**

Only when the journal is reasonably filled the changes are committed, i.e. new versions of the index and the master node are written

Storage Areas / 2

- The **log** area is used to store the journal

  As the B+-tree in the index is never really correct, nodes of the index must be buffered

  When the flash memory is mounted, the correct index is restored by a procedure called **replay**

  The replay process is complicated; among others it uses a red-black tree to first bring the stored nodes into an appropriate order before updating the index

- The **LEB properties tree** (LPT) stores values that need to be known for all LEBs: free space, dirty space, and whether the EB is an index block or not

  LEB properties are also stored in a wandering tree, which requires the organisation of the LPT area to be like the file system organisation

Storage Areas / 3

- The **orphan area** keeps track of orphans, i.e. files that were deleted while open

  In normal processing the data in these files becomes unaccessible, when the file is closed

  However, in case of an unclean unmount, orphans have to be accounted for and deleted by the file system management

- The **main area** contain the EBs, in which the real data in the file system is stored

There is some ongoing work showing the correctness of flash file system implementations, in particular with respect to recovery, mounting, and unmounting