Database Technology
Indexing

Fang Wei-Kleiner

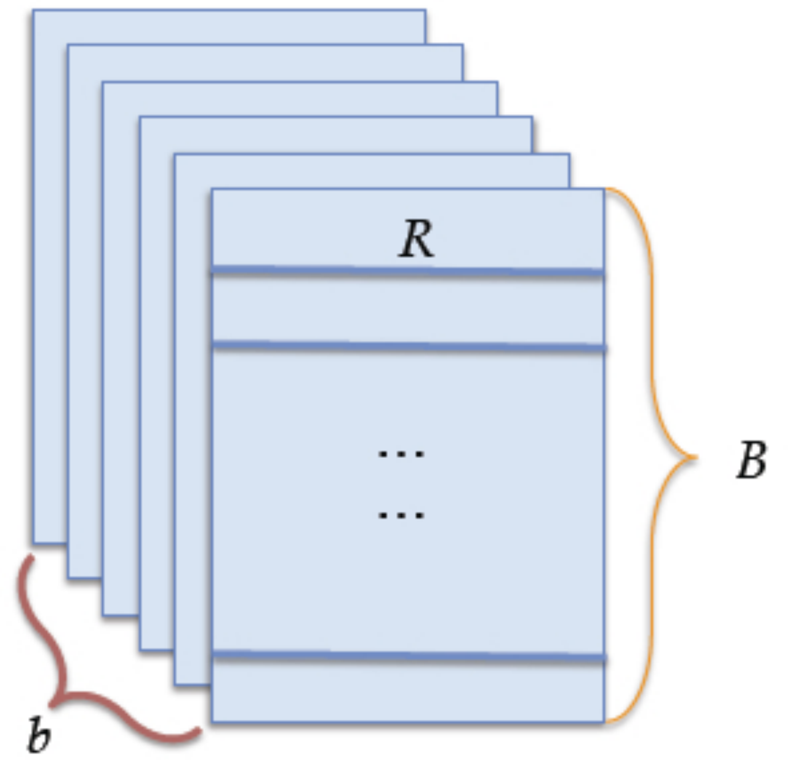
Files and records

- Let us assume
  - B is the size in bytes of the block.
  - R is the size in bytes of the record.
  - r is the number of records in the file.
- Blocking factor (number of records in each block):

$$bfr = \left\lfloor \frac{B}{R} \right\rfloor$$

- Blocks needed for the file:

$$b = \left\lceil \frac{r}{bfr} \right\rceil$$



Primary index

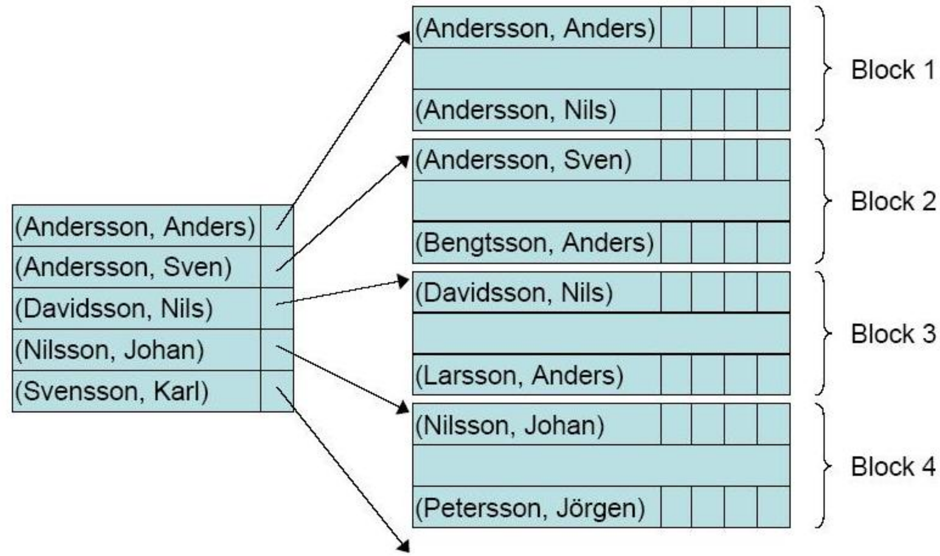
- Let us assume that the ordering field is a **key**.
- Primary index = **ordered** file whose records contain two fields:
  - One of the ordering key values.
  - A pointer to a disk block.

  binary search !
- There is one record for each data block, and the record contains the ordering key value of the first record in the data block plus a pointer to the block.

Primary index

Ki	Pi

K1 = (Andersson, Anders)	P1 = Address of block 1
K2 = (Andersson, Sven)	P2 = Address of block 2
K3 = (Davidsson, Nils)	P3 = Address of block 3
K4 = (Nilsson, Johan)	P4 = Address of block 4
K5 = (Svensson, Karl)	P5 = Address of block 5



- Why is it faster to access a random record via a binary search in index than in the file ?
- What is the cost of maintaining an index? If the order of the data records changes…

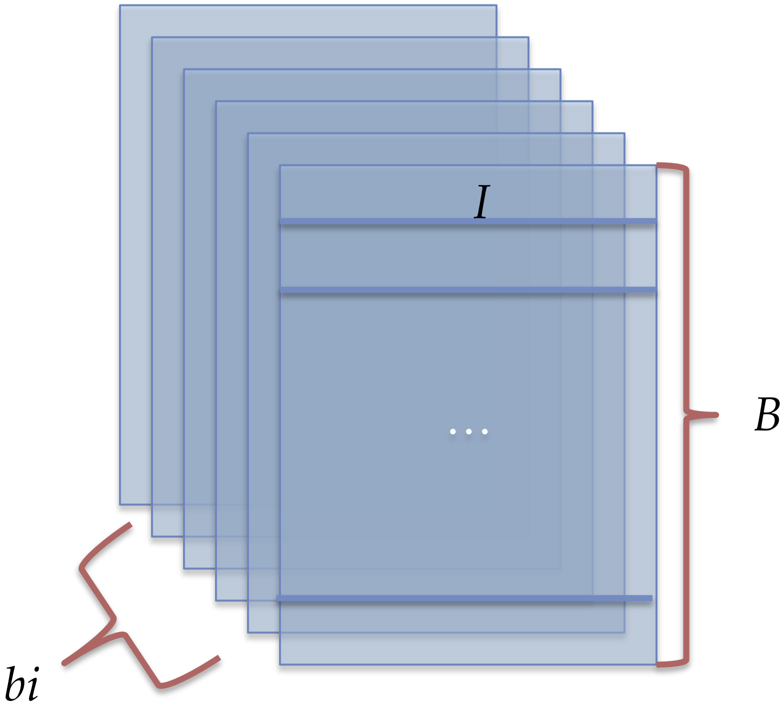
Primary Index

- B is the size in bytes of the block.
- I is the size in bytes of the index.
- x is the number of index entries (for primary index x=b).
- Blocking factor index:

$$bfr_i = \left\lfloor \frac{B}{I} \right\rfloor$$

- Blocks needed for the file:

$$b_i = \left\lceil \frac{b}{bfr_i} \right\rceil$$



Exercise

- Assume an ordered file whose ordering field is a key. The file has 1000000 records of size 1000 bytes each. The disk block is of size 4096 bytes (unspanned allocation). The index record is of size 32 bytes.
- How many disk block accesses are needed to retrieve a random record when searching for the key field
  - Using no index ?
  - Using a primary index ?

Primary index

- What is the cost for maintaining a primary index?
- Insert
- Delete
- Update

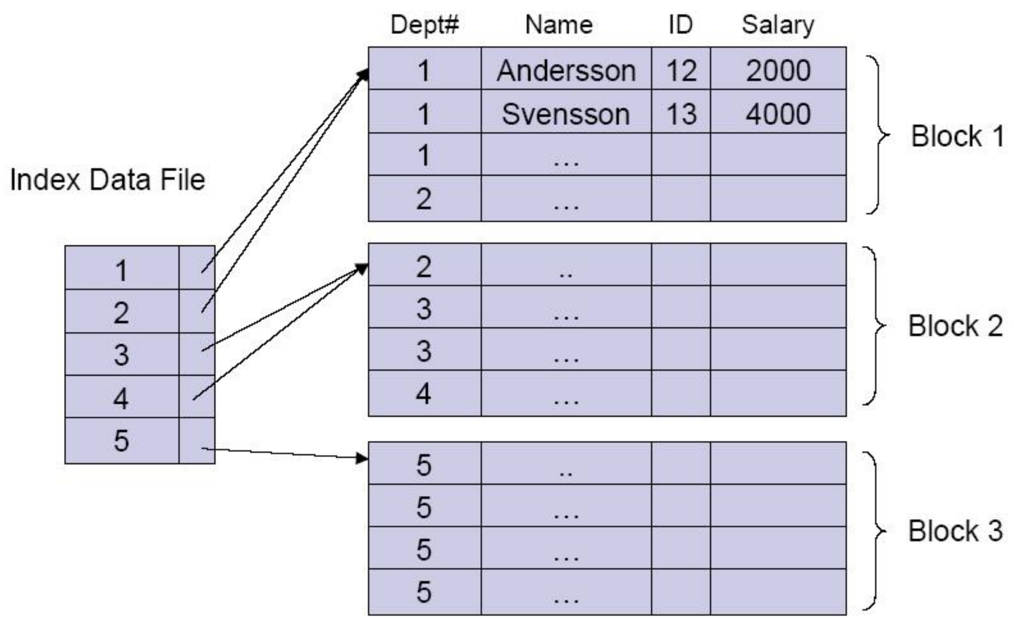
Clustering index

- Now, the ordering field is a non-key.
- Clustering index = **ordered** file whose records contain two fields:
  - One of the ordering field values.
  - A pointer to a disk block.

binary search !

- There is one record **for each distinct** value of the ordering field, and the record contains the ordering field value plus a pointer to the **first** data block where that value appears.

Clustering index



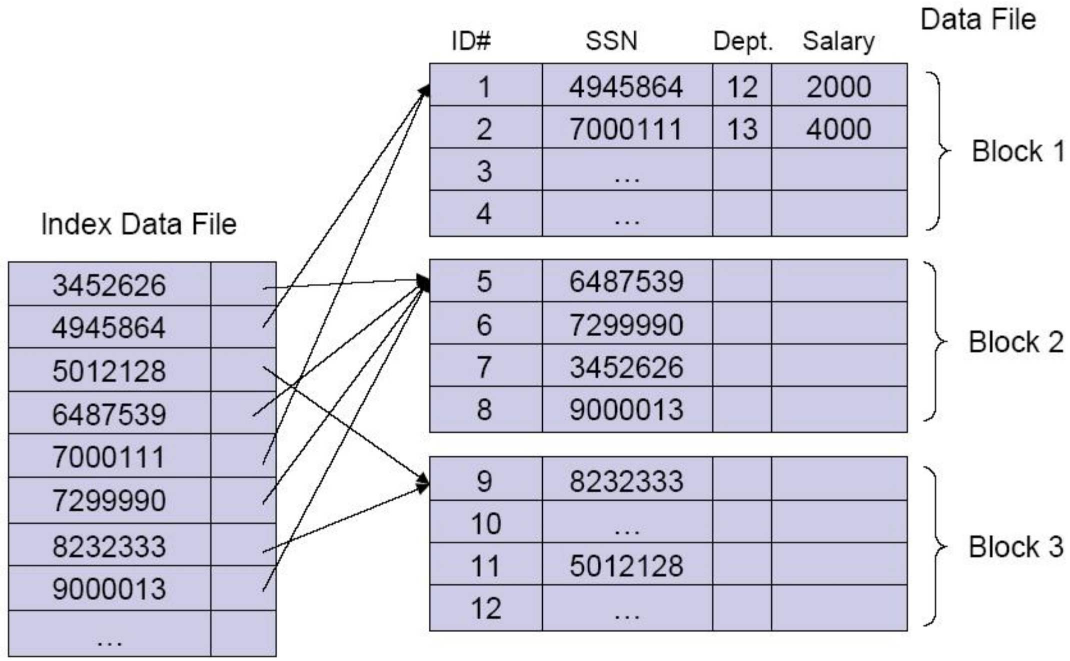
- Efficiency gain ? Maintenance cost ?

Secondary indexes

- The index is now on a **non**-ordering field.
- Let us assume that that is a **key**.
- Secondary index = **ordered** file whose records contain two fields:
  - One of the non-ordering field values.
  - A pointer to a disk record or block.
- There is one record per data record.

binary search !

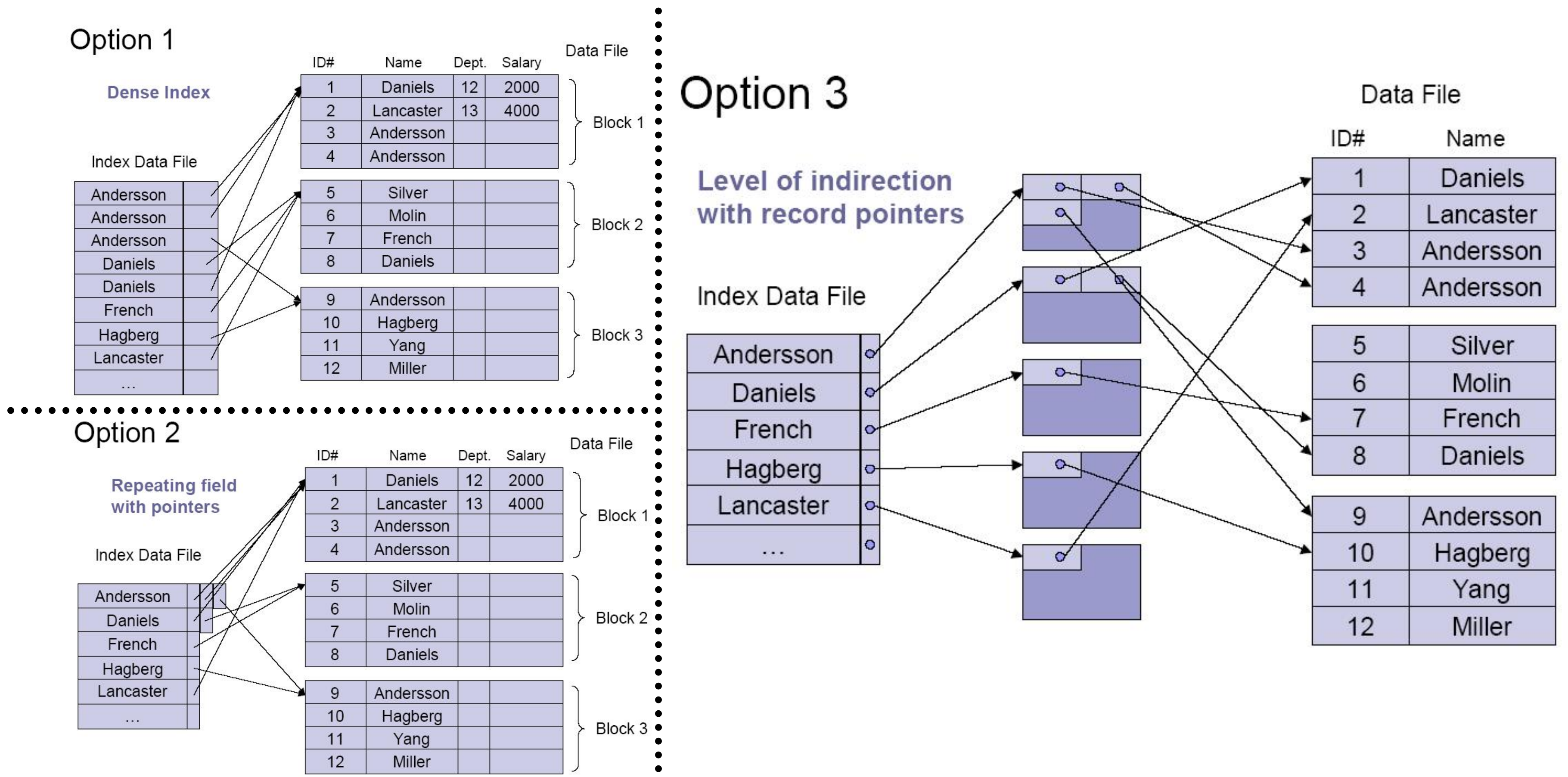
Secondary indexes



- Efficiency gain ? Maintenance cost ?

Secondary indexes

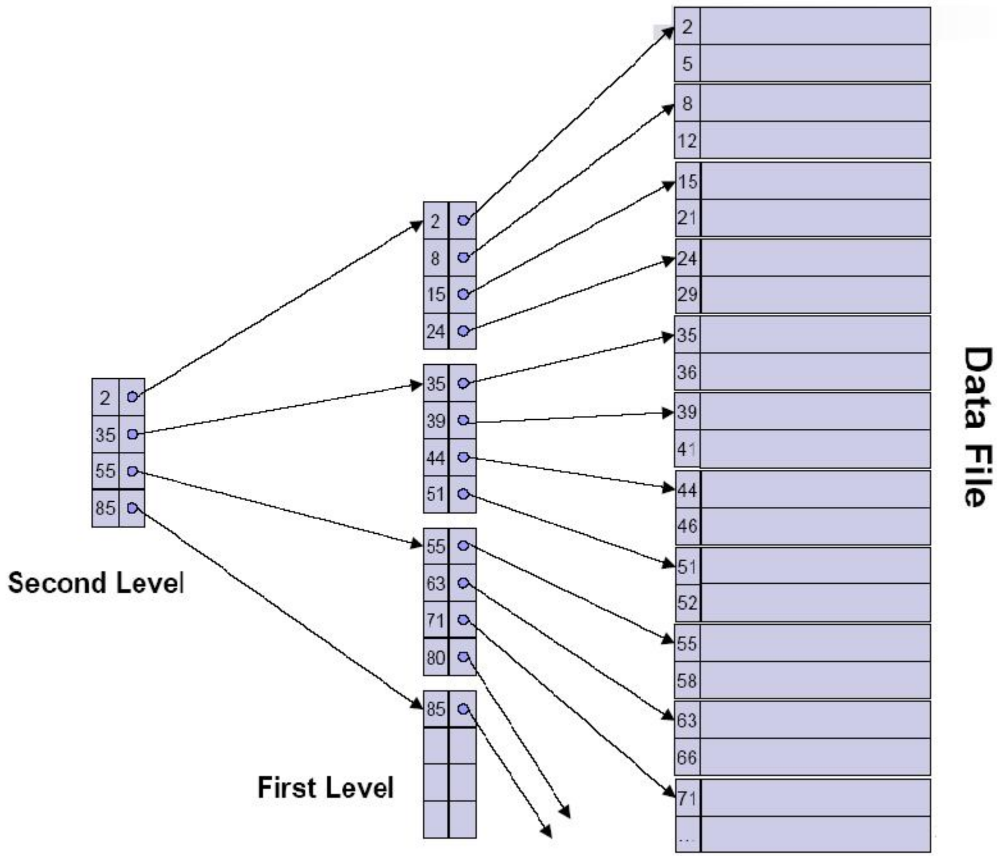
- Now, the index is on a non-ordering and **non**-key field.



Multilevel indexes

- Index on index (first level, second level, etc.).
- Works for primary, clustering and secondary indexes as long as the first level index has a **distinct** index value for every entry.
- How many levels ? Until the last level fits in a **single** disk block.
- How many disk block accesses to retrieve a random record?

Multilevel indexes



- Efficiency gain ? Maintenance cost ?

Exercise

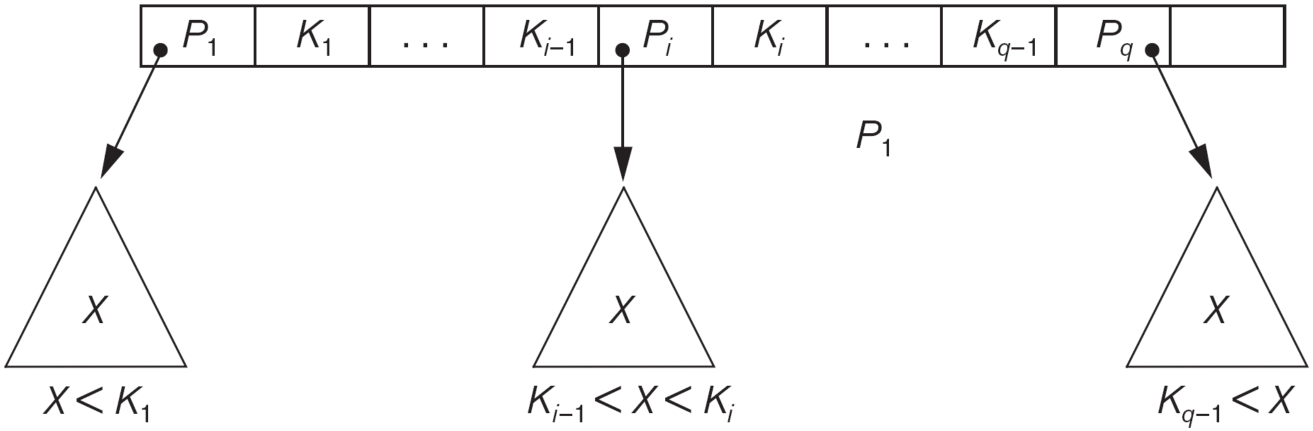
- Assume an ordered file whose ordering field is a key. The file has 1000000 records of size 1000 bytes each. The disk block is of size 4096 bytes (unspanned allocation). The index record is of size 32 bytes.
- How many disk block accesses are needed to retrieve a random record when searching for the non-ordering key field
  - Using no index ?
  - Using a secondary index ?
  - Using a multilevel index ?

Dynamic multilevel indexes

- Record insertion, deletion and update may be expensive operations. Recall that all the index levels are **ordered** files.
- Solutions:
  - Overflow area + periodic reorganization.
  - Dynamic multilevel indexes, based on B-trees and B+-trees.
- → Search tree
- → B-tree
- → B+-tree

Search Tree

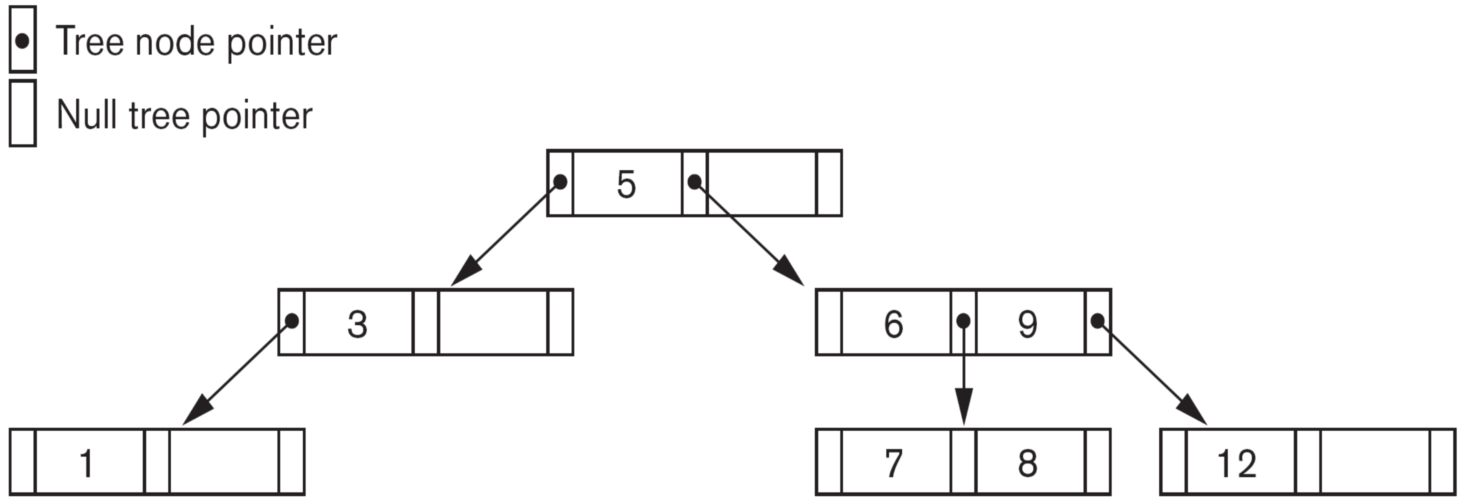
Figure 18.8
A node in a search tree with pointers to subtrees below it.



- A search tree of **order** ***p*** is a tree s.t.
  - Each node contains at most p-1 search values, and at most p pointers $<P_1, K_1, \ldots P_i, K_i \ldots K_{q-1}, P_q>$ where $q \leq p$
  - P_i: pointer to a child node
  - K_i: a search value (key)

→ within each node: $K_1 < K_2 < K_i < \ldots < K_{q-1}$

Figure 18.9
A search tree of order $p = 3$.



- Searching a value X over the search tree
  - Follow the appropriate pointer P_i at each level of the tree
    - → only one node access at each tree level
    - → time cost for retrieval equals to the depth *h* of the tree
    - Expected that *h* << *tree size (set of the key values)*
  - Is that always guaranteed?

Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- B stands for Balanced → all the leaf nodes are at the same level (both B-Tree and B+-Tree are balanced)
  - Depth of the tree is minimized
- These data structures are variations of search trees that allow efficient insertion and deletion of search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
  - Recall the multilevel index
  - Ensure big fan-out (number of pointers in each node)
- Each node is kept between half-full and completely full
  - Why?

Dynamic Multilevel Indexes Using B-Trees and B+-Trees (cont.)

- Insertion
  - An insertion into a node that is not full is quite efficient
    - If a node is full the insertion causes a split into two nodes
  - Splitting may propagate to other tree levels

- Deletion
  - A deletion is quite efficient if a node does not become less than half full
  - If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels of the tree
- In a B+-tree, all pointers to data records exists only at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

B-tree Structures

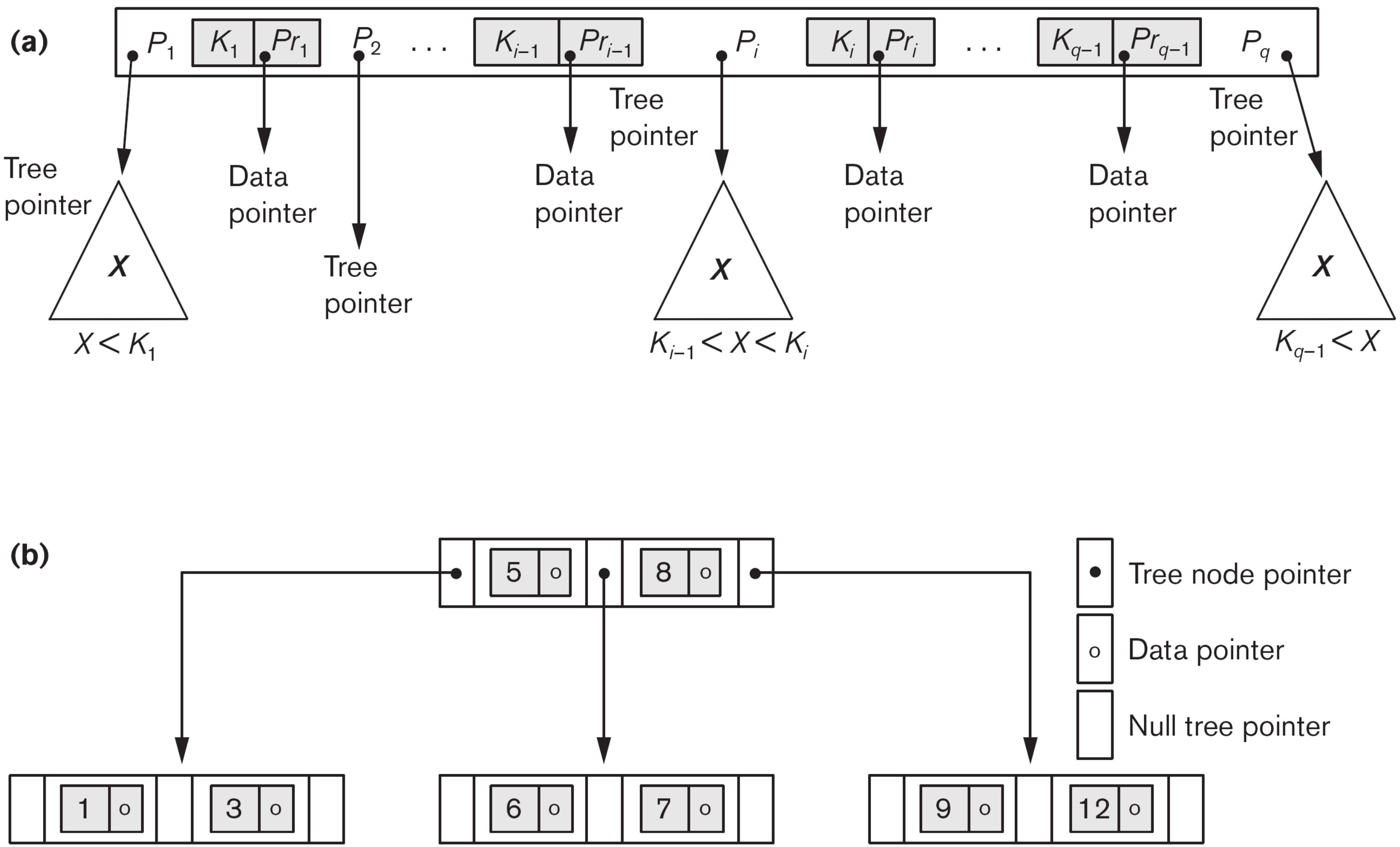
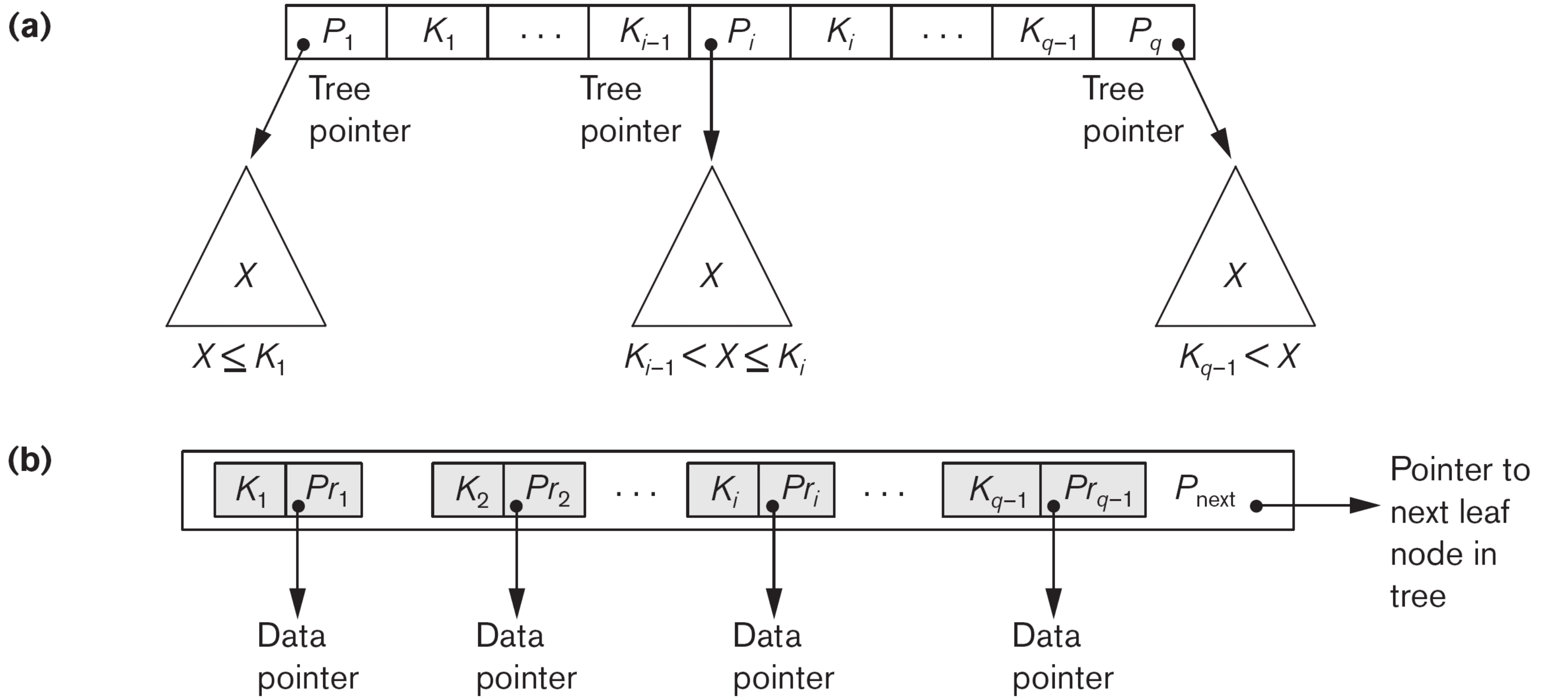


Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

The Nodes of a B+-tree

Figure 18.11
The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values.
(b) Leaf node of a B^+-tree with $q - 1$ search values and $q - 1$ data pointers.



P_{next} (pointer at leaf node): ordered access to the data records on the indexing fields

B+-trees: Retrieval

- Very fast retrieval of a random record

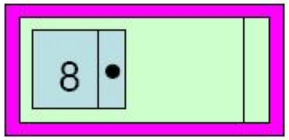
$$\left\lceil \log_{\left\lceil \frac{p}{2} \right\rceil} N \right\rceil + 1$$

  - p is the order of the internal nodes of the B+-tree.
  - N is the number of leaves in the B+-tree.
- How would the retrieval proceed ?
- Insertion and deletion can be expensive.

B+-trees: Insertion

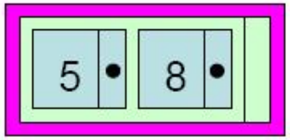
Insert: 8

B+-trees: Insertion



Insert: 5

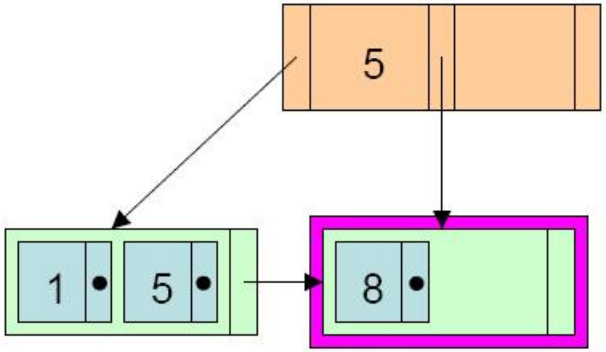
B+-trees: Insertion



Overflow – create a new level

Insert: 1

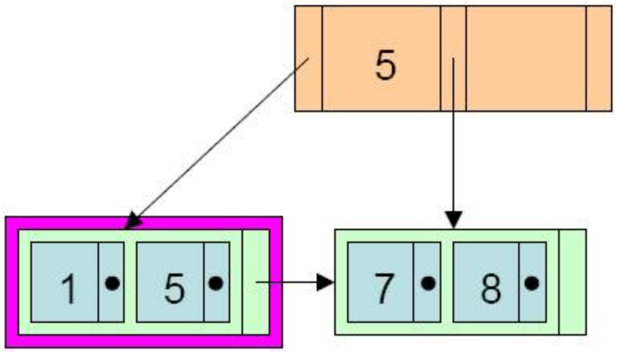
B+-trees: Insertion



Insert: 7

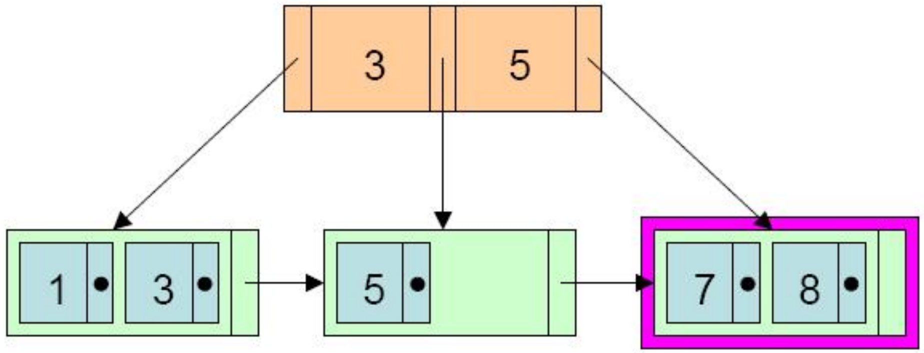
B+-trees: Insertion

Overflow - Split



Insert: 3

B+-trees: Insertion

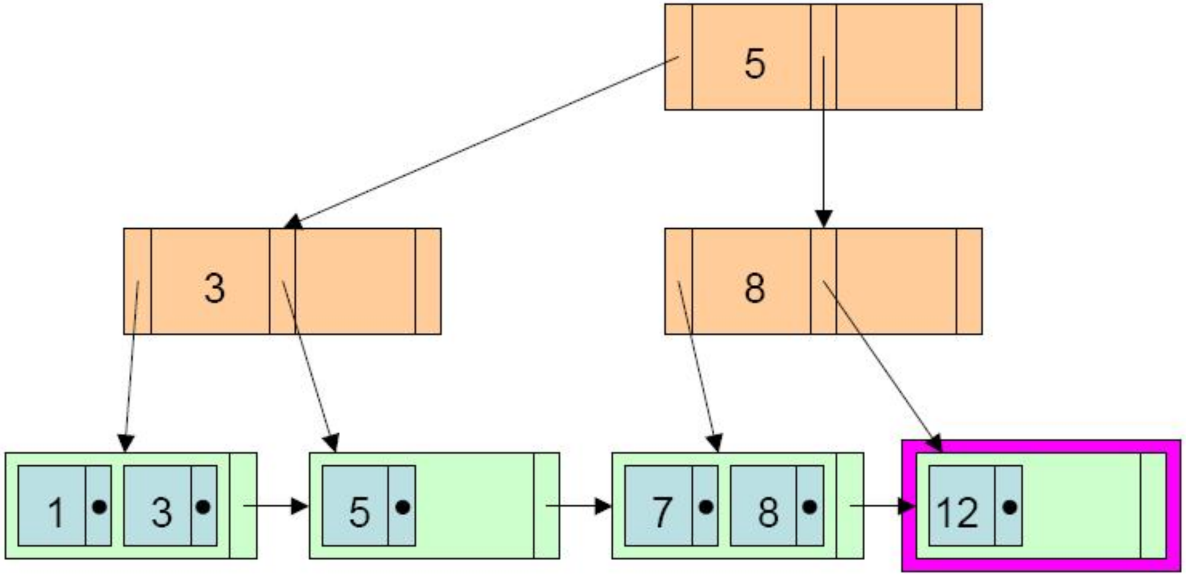


Overflow - Split

Propagates a new level

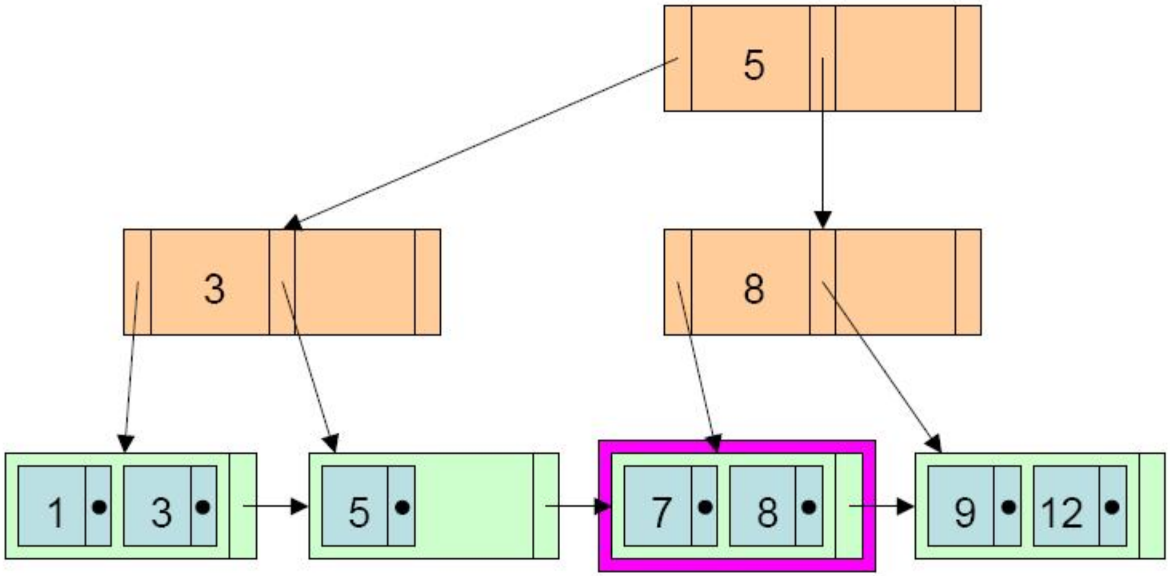
Insert: 12

B+-trees: Insertion



Insert: 9

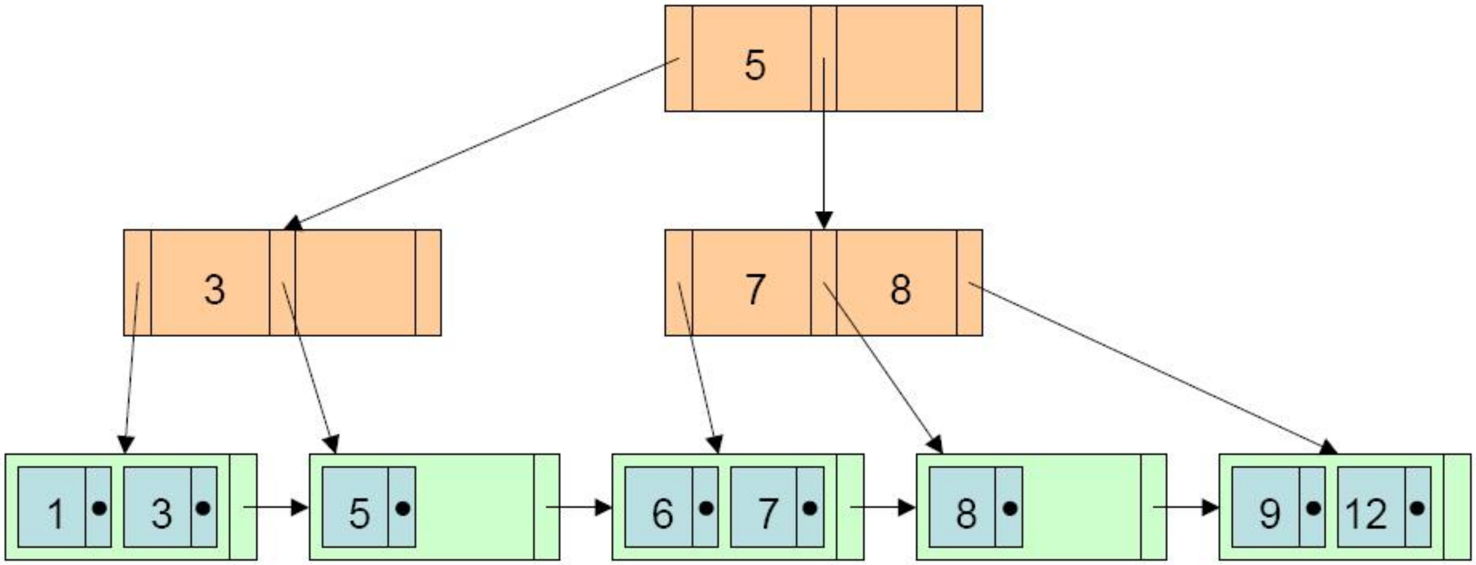
B+-trees: Insertion



Overflow – Split, propagates

Insert: 6

B+-trees: Insertion



Resulting B+-tree

B-trees: Order

One **node** must fit in one block:

$$p \cdot P_{block} + (p-1) \cdot (P_{record} + K) \leq B \Rightarrow p \leq \frac{B + P_{record} + K}{P_{block} + P_{record} + K}$$

p	order, number of block pointer entries in a node
P_{block}	size of a block pointer
P_{record}	size of a record pointer
K	size of a search key field

B+-trees: Order

One **internal node** must fit in one block:

$$p \cdot P_{block} + (p-1) \cdot K \le B \qquad \Rightarrow p \le \frac{B+K}{P_{block}+K}$$

One **leaf node** must fit in one block:

$$p_{leaf} \cdot (P_{record} + K) + P_{block} \le B \Rightarrow p_{leaf} \le \frac{B-P_{block}}{P_{record}+K}$$

p	order, number of pointer entries in an internal node
p_{leaf}	number of record pointer entries in a leaf node
P_{block}	size of a block pointer
K	size of a search key field
P_{record}	size of a record pointer

Exercise

- B=4096 bytes, P=16 bytes, K=64 bytes, node fill percentage=70 %.
- For both B-trees and B+-trees:
  - o Compute the order p.
  - o Compute the number of nodes, pointers and key values in the root, level 1, level 2 and leaves.
  - o If the results are different for B-trees and B+-trees, explain why this is so.