INDEXING AND HASHING

Indexes as auxiliary access structures

Indexes are auxiliary files that provide **secondary access paths** which enable an alternative access to the main records

- do not affect physical placement of records on disks (they exist in parallel to the primary file organization)
- efficient access is enabled according to values of particular fields called **indexing fields**
- provide **pointers** to records based on search conditions on indexing fields

Indexes require additional maintenance costs and take additional place

- insert and delete operations are slower due to the index synchronization
- trade-off on the number of indexes to create on the database

Different types of indexes

There are many different types of indexes. We are focusing here on the following types

- **single-level indexes** - based on ordered files
- **multi-level indexes**
- **tree data structures**
  - **B-trees**

Another characterization of indexes:

- **dense** indexes - contain one pointer for each value of the indexing field
- **sparse** indexes - do not contain all values of the indexing field
  - number of index entries is smaller than number of records

Single-level ordered indexes

Single-level ordered indexes follow the approach to making indexes for books

- index keeps pointers to all blocks that contain records that have particular value of the indexing field
- values of the indexing field are *ordered* in the index and it's possible to do *binary search* on those fields

Types of ordered indexes

- **primary index** as an index defined on the ordering key field
  - each record has different value on that field
- **clustered index** is an index defined on the field which is ordering field but not unique in one file
- **secondary index** is an index defined on any non-ordering field

Primary indexes (single-level indexes)

A **primary index** is an ordered file defined on the ordering field which is the primary key

– index records are of fixed length with two fields:

$$< K(i), P(i) >$$

1. $K(i)$ - contains a value of the primary key
2. $P(i)$ - is a pointer to a disk block of the data file where that record is placed

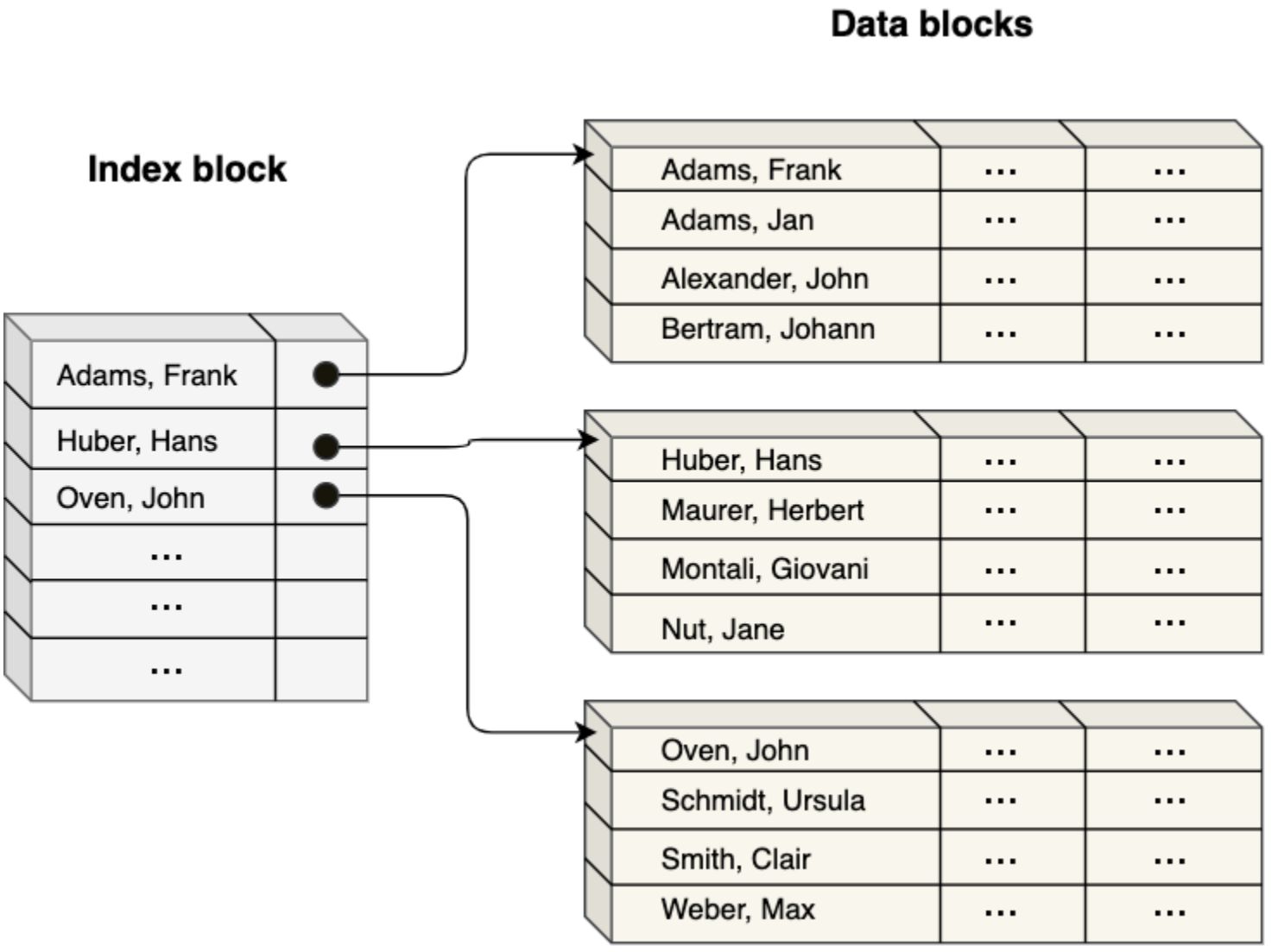
– record whose primary key is $K(i)$ lies in the block whose address is $P(i)$

Primary index contains an index entry for each block of the corresponding ordered data file

– **anchor record** is the first record in each block of the data file whose primary key value represents the block

– primary keys of anchor records are keys of the primary index

Primary index - example



Primary indexes

Since primary indexes include one entry (record) for each block of the data file they are sparse indexes

- number of index entries ((key,pointer) pairs) is equal to the number of blocks
- binary search is used to locate an index record with a required value

Number of block accesses to retrieve a searched record using binary search is

$$log_2 b_i + 1$$

- b_i represents total number of blocks in the index file
- one additional block access to the data file is necessary to read the block containing the actual record

This number of accesses is better than $log_2 b$ which is required for access ordered file with no primary index

- $b_i < b$ because index records take less space

Ordered file - example

Suppose that we have an ordered data file with records of a fixed size:

- number of records $r = 30000$
- record size $R = 100$ bytes
- block size $B = 1024$ bytes
- block organization is unspanned

Calculating the number of block accesses necessary to retrieve a record without using an index:

- $bfr = \lfloor \frac{1024}{100} \rfloor = 10$
- necessary number of blocks in the data file
  $b = \lceil \frac{r}{bfr} \rceil = \lceil \frac{30000}{10} \rceil = 3000$
- binary search on the ordering field requires
  $\lceil log_2 b \rceil = \lceil log_2 3000 \rceil = 12$ block accesses

Primary index - example

Suppose that our ordered file from the last slide uses a primary index having records that consist of:

- 1.an ordering key of 9 bytes
- 2.a pointer field with the size of 6 bytes
- thus the total size of an index record is 9 + 6 = 15 bytes
- every block of the data file has its corresponding index entry $\Longrightarrow r_i = 3000$ index blocks
- index block size: 1024 bytes

Calculating the cost to find a record using the primary index:

- $bfr_i = \left\lfloor \frac{1024}{15} \right\rfloor = 68 \Longrightarrow b_i = \left\lceil \frac{r_i}{bfr_i} \right\rceil = \left\lceil \frac{3000}{68} \right\rceil = 45$ - index blocks
- binary search on the index takes $\lceil log_2 b_i \rceil = \lceil log_2 45 \rceil = 6$ accesses
- an additional block access is required to access the data file $\Longrightarrow$ 6 + 1 = 7 block accesses which is 41.7% better than the previous case

Primary index - insertion and deletion

Inserting and deleting operations in ordered files with indexes are very (computationally) expensive

- firstly, many record blocks in the ordered data file have to be moved
- moving records in data file blocks causes changing anchor records
- changing anchor records requires changing entire part of the index
- improvements can be achieved using overflow files or linked lists of overflow records

Primary index - summary

Primary index files are searched using binary search algorithm just like ordered files

Search on the primary key using an index is more efficient than search through the original ordered file because

- records of the primary index are smaller and its blocking factor is mostly greater than the blocking factor of the data file
- primary index is sparse and contains only anchor fields - only one entry for each block of the data file

Insert and delete operations are now more expensive

- index files have to be changed in addition to data file

Search over non-indexing fields is still done using linear search

- the existence of the primary index does not affect it

Clustering index (single-level index)

Clustering indexes are indexes on ordering fields which do not have to be unique (more records can have the same value).

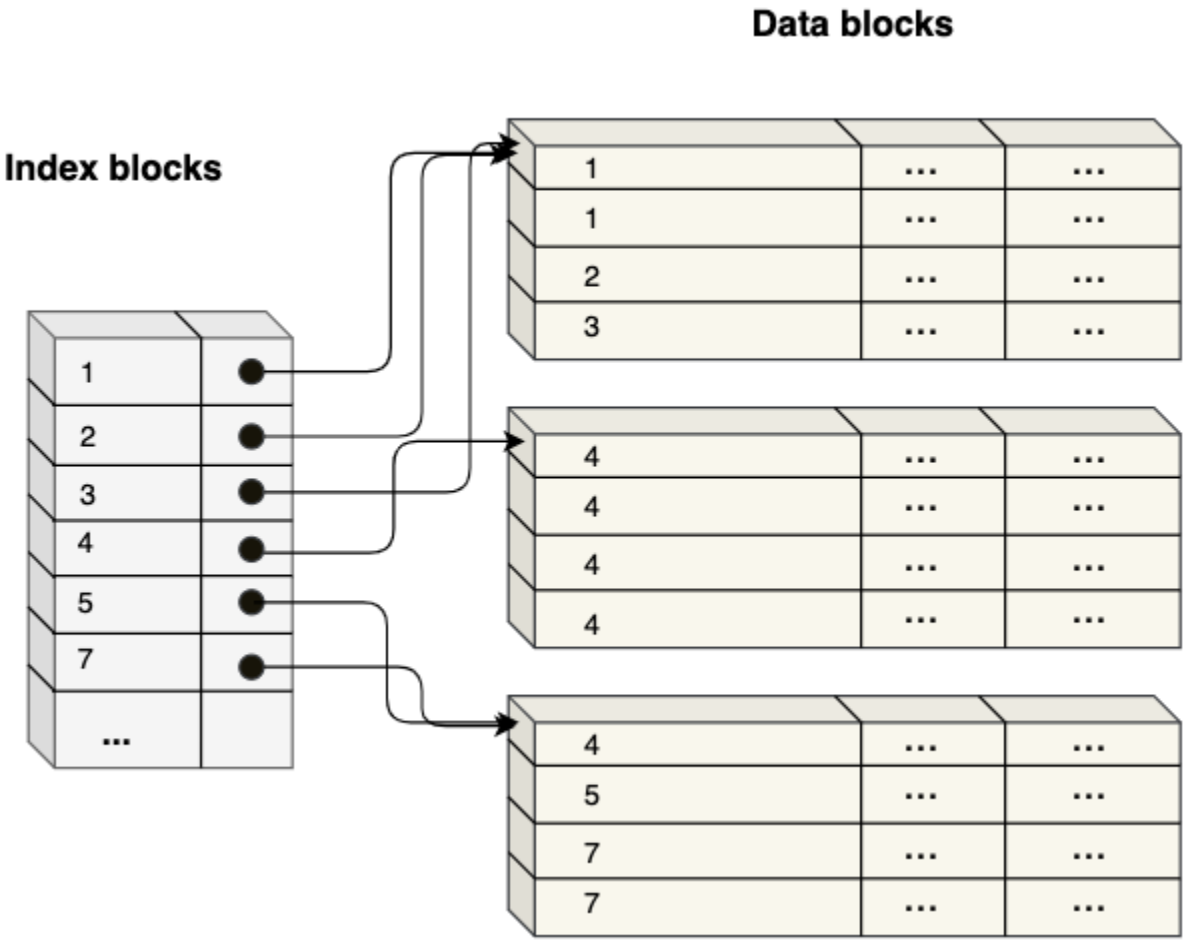
- these fields are called **clustering fields**
- clustering index contains an entry for each *distinct value* of the clustering field in the data file
- pointer of an index entry points to the first block in the data file that has a record with that value for its clustering field

Clustering index is an ordered file where each entry consists of two fields

- first field has the same type as the clustering field
- second field is a disk block pointer

Clustering index is also sparse index

Clustering index - example



Clustering index with one pointer for each distinct value of the clustering field

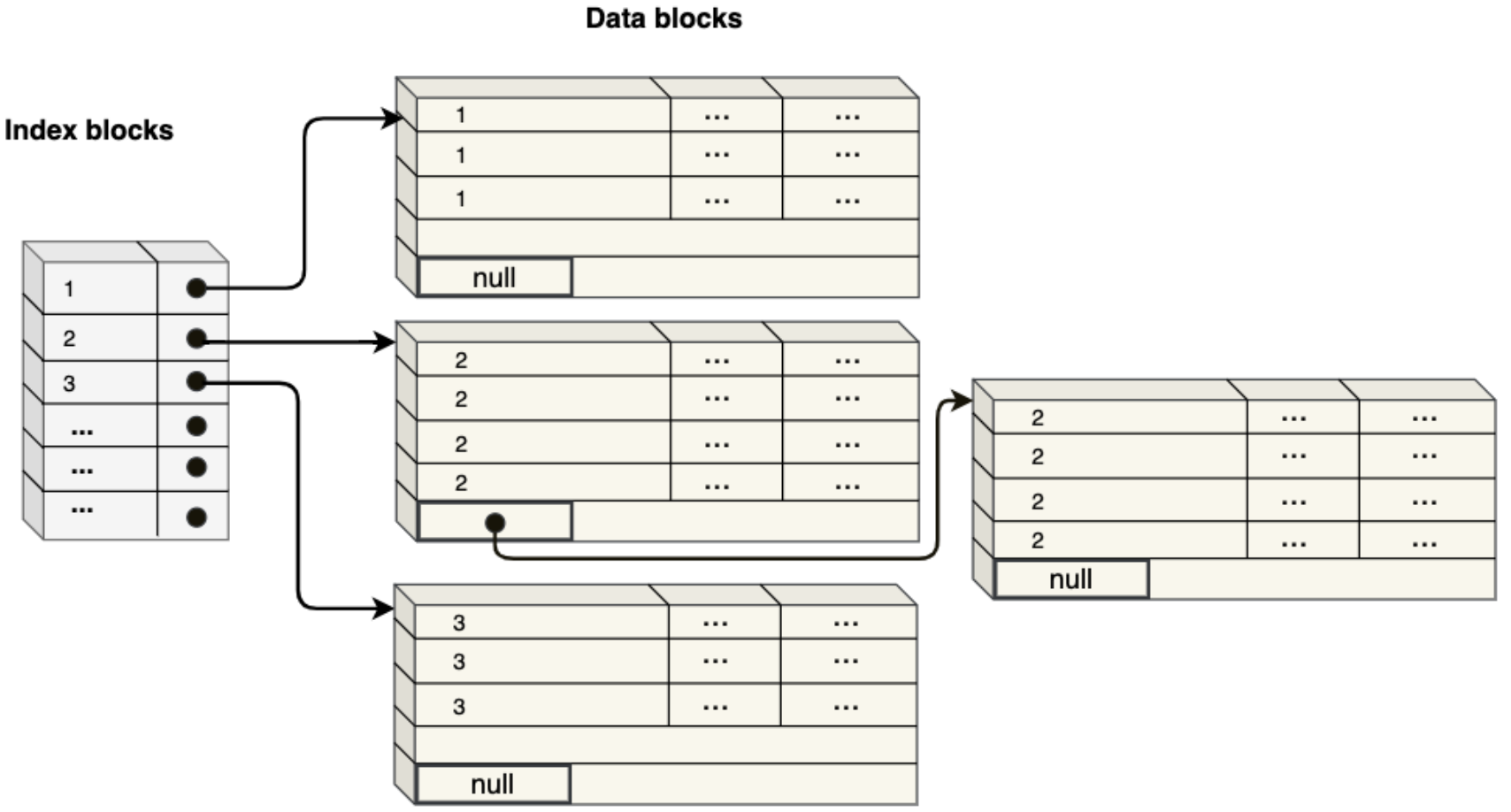
Clustering index - insert and delete operations

Clustering index has the same problems with complex insert and delete operations as primary index

Alternative implementation proposed to mitigate these problems

- one block or cluster of blocks reserved for each clustering value
- a sort of overflow-list can be used
- inserting of a new element in the block which is full leads to inserting elements in the overflow-block
  - overflow-block is linked with a pointer to the original block
- null pointer specifies that there are no more entries with that specific value

Clustering index with overflow blocks



Alternative implementations to improve insert and delete operations

– entire block or cluster of contiguous blocks can be reserved for one value

Secondary indexes

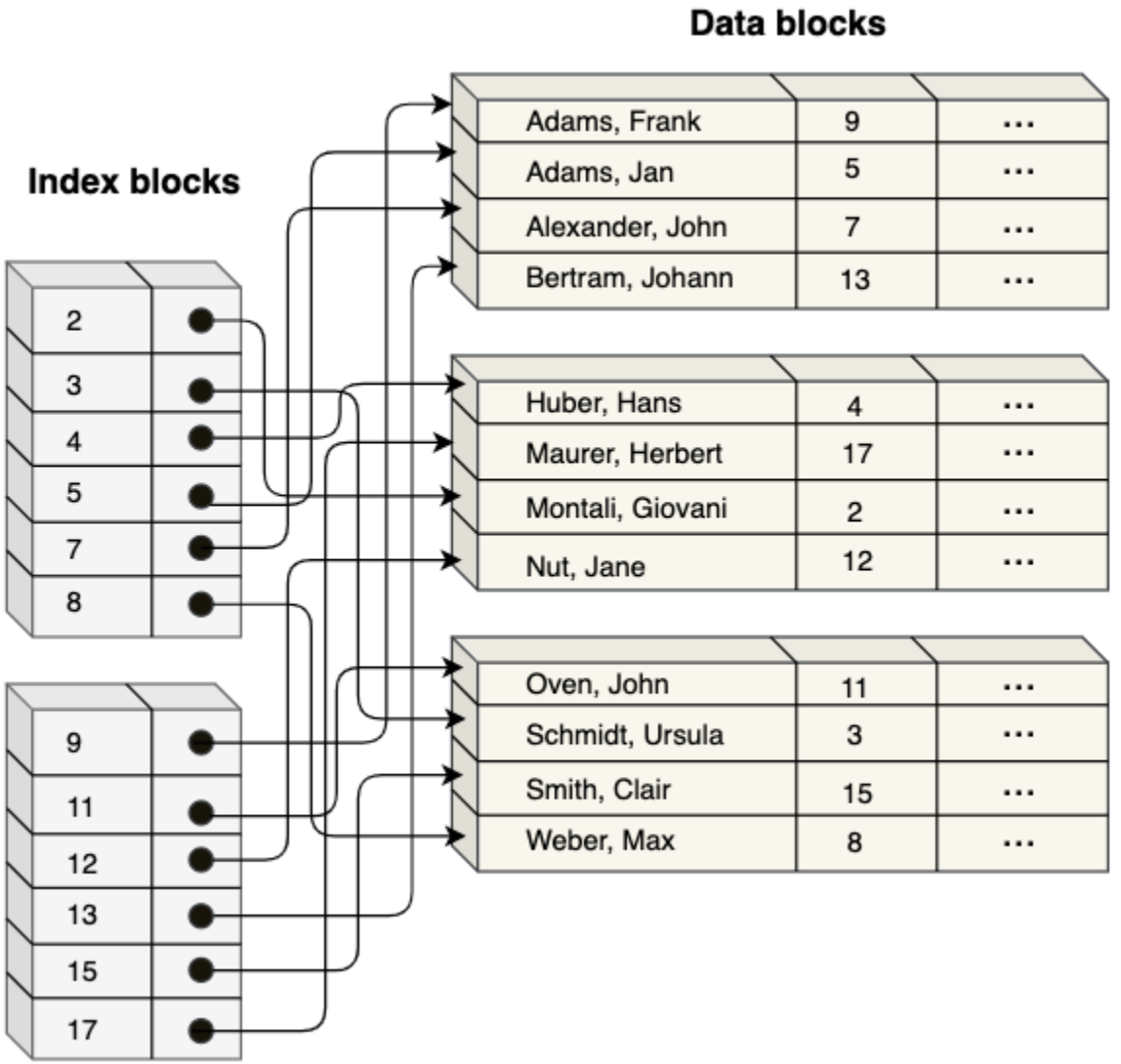
Secondary indexes provide secondary means of accessing data files

- exists together with some primary access paths
- secondary index is an ordered file defined on a field which is a **non-ordering field** (field can be unique, candidate key or it can have duplicated values)
  - many secondary indexes can be created for the same data file (in contrast to primary and clustering indexes)
- secondary index has records that are pairs of two fields $< K(i), P(i) >$
  - first field contains a value of a non-ordering field and is called **indexing field**
  - second field contains a block pointer or a record pointer

Types of Secondary indexes

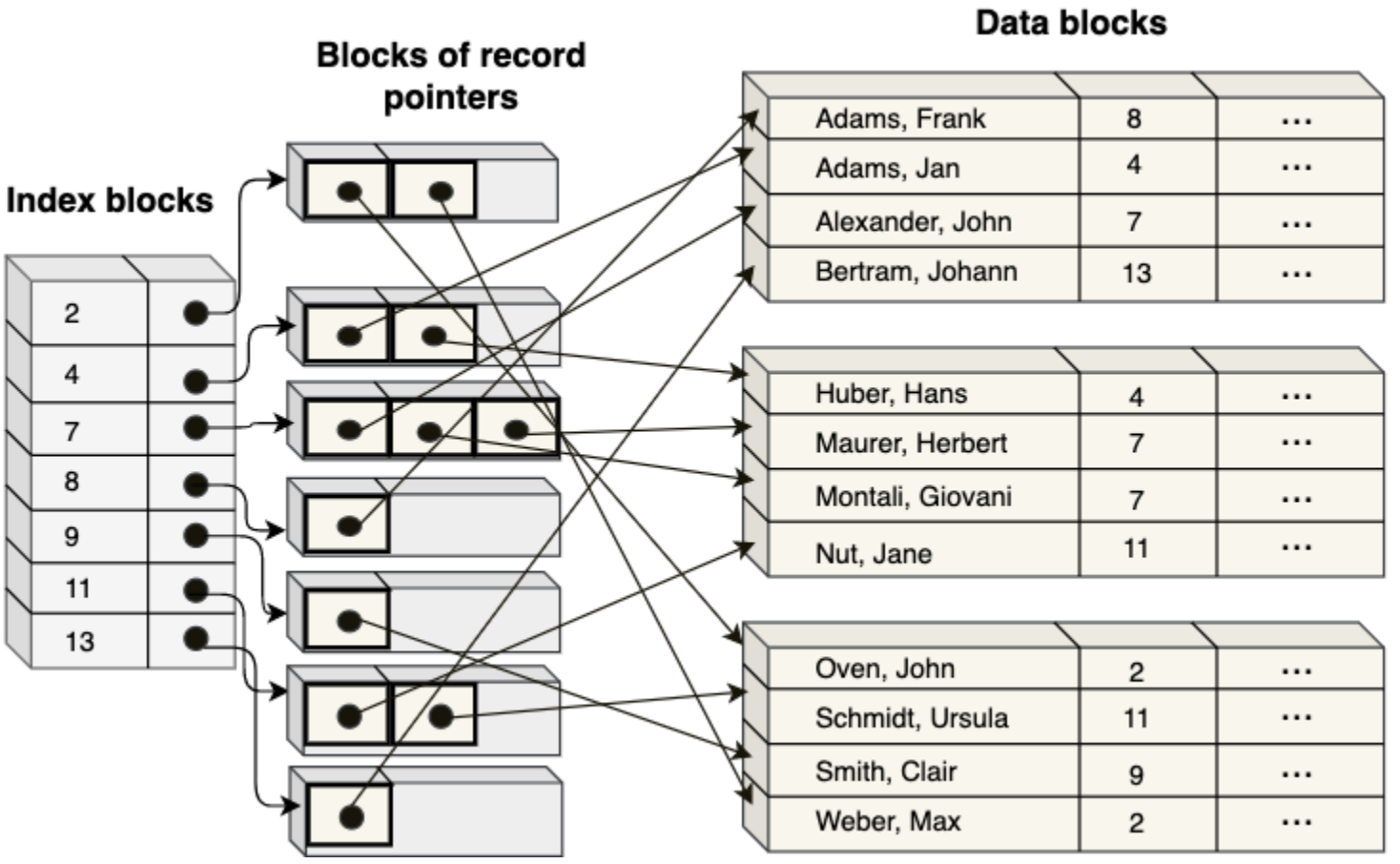
1. A secondary index which is created on a *key(unique) field* (secondary key) has one index entry for each record
   - represents *dense* index and has pointer to the block of the record or to the record itself
   - this corresponds to any UNIQUE key attribute
2. A secondary index which is created on a *non-key* field and can have for each index entry (indexing field) more corresponding records in the data file
   - pointer of the index entry has address of the location of the block with record pointers
   - represents *sparse* secondary index

Secondary index on a key field



Dense secondary index on a secondary key attribute

Secondary index on a non-key field



Sparse secondary index on a non-key attribute

Searching file - example

Suppose that we have a file with records of fixed size:

- number of records $r = 30000$
- record size $R = 100$ bytes
- block size $B = 1024$ bytes
- block organization is unspanned

Calculating the cost of the search operation on a non-ordering field (secondary key) with no index used:

- $bfr = \left\lfloor \frac{1024}{100} \right\rfloor = 10$ - blocking factor
- necessary number of blocks in the file $b = \left\lceil \frac{r}{bfr} \right\rceil = \left\lceil \frac{30000}{10} \right\rceil = 3000$
- only **linear search** is possible on the non-ordering field and requires on average ** $\frac{b}{2} = \frac{3000}{2} = 1500$ block accesses
- in the worse case when record does not exist searching requires $b = 3000$ accesses
- complexity of the operation is $O(b)$

Secondary index - search example

Suppose that the file from the last slide uses dense secondary index with the following properties

- secondary key of length 9 bytes as index entry key
- size of the pointer field 6 bytes
- size of index record is therefore 15 bytes
- every record of the data file has its corresponding index entry $\Longrightarrow r_i = 30000$ index blocks
- block size: 1024 bytes

Calculations for the number of block accesses to find a record using defined secondary index:

- $bfr_i = \left\lfloor \frac{1024}{15} \right\rfloor = 68 \Longrightarrow b_i = \left\lceil \frac{r_i}{bfr_i} \right\rceil = \left\lceil \frac{30000}{68} \right\rceil = 442$ - index blocks
- binary search on the index $\lceil log_2 b_i \rceil = \lceil log_2 442 \rceil = 9$ accesses
- one additional access is required to access data file $\Longrightarrow$ 9 + 1 = 10 accesses which is 99% better than the case with linear search (1500 accesses)

Multilevel-indexes

Multilevel indexes are created from the idea to provide better search than binary search

- each iteration of binary search algorithm reduces **record search space** (part of the index which should be searched) by factor 2

Multilevel indexes reduce the *record search space* by blocking factor bfr_i (blocking factor is larger than 2)

- value bfr_i is called **fan-out** of the multilevel index and it is denoted by **fo**
  $fo = bfr_i$
- searching a multilevel index requires approximately $log_{fo} b_i$ block accesses.
  - in most cases fo is much larger than 2 and this substantially improves the number of block accesses

Multilevel-indexes

Multilevel index consists of more levels (files)

- first level is an sorted (ordered) file with distinct values for each K(i)
- second and other upper levels are primary indexes over the previous levels
  - only anchor records from the previous level are included
- new levels are added until the number of records which remain can fit into a single block
- each level reduces the number of entries from the previous level by factor *fo*
- number of levels is calculated by $t = \lceil log_{fo}(r_1) \rceil$ (r_1 - number of entries on the first level)

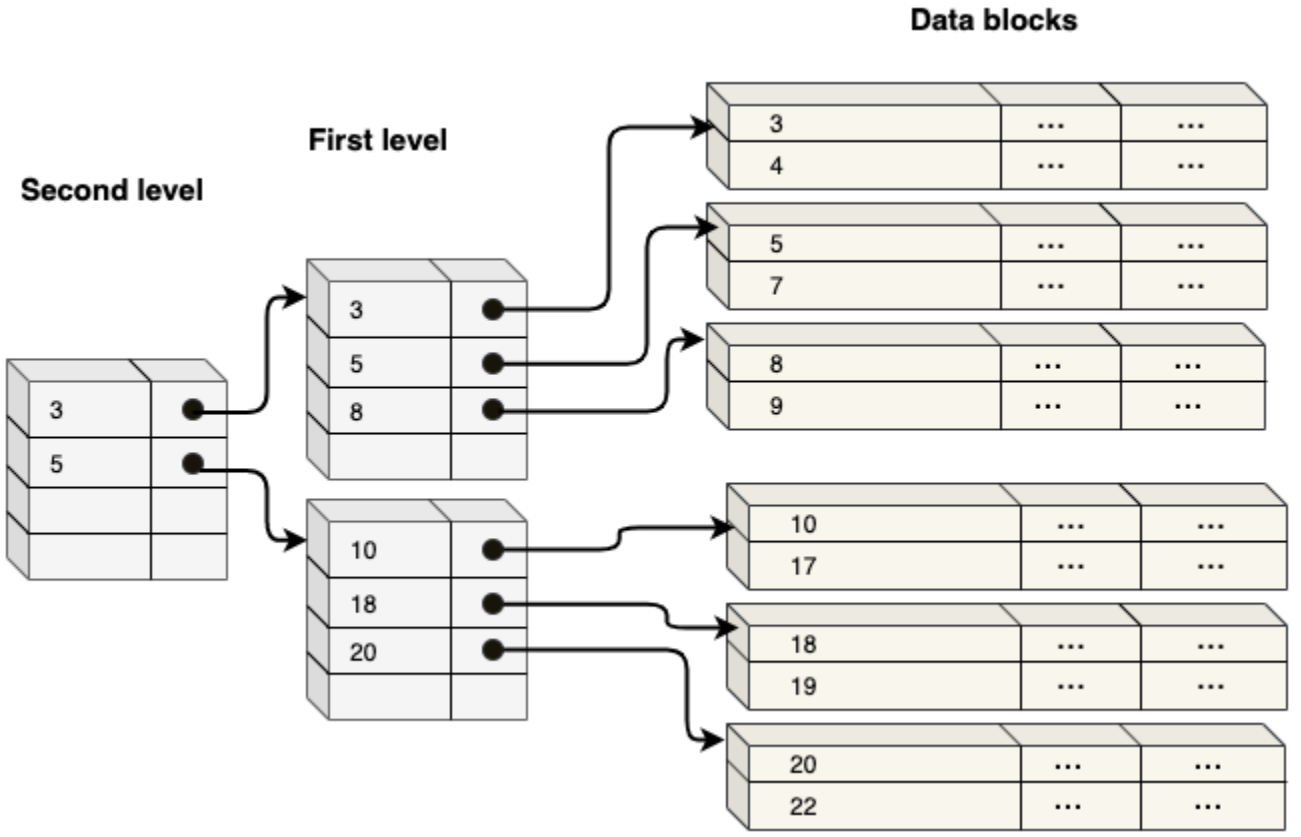
Searching operation on the field appearing in the index requires **only** one block access at each level

- number of accesses is therefore $\lceil log_{fo}(r_1) \rceil$

Multilevel scheme can be used for any type of indexes as long as the first level index uses fixed-length entries and contains all distinct values

- first level indexes can be primary, clustered or secondary index

Multilevel index - example



Multilevel index built over a primary key

Search in a multilevel index - example

We transform secondary index from our last example

- given data file has 30000 records and the index blocking factor $bfr_i = 68$
- our secondary index has $b_1 = 442$ blocks

Our goal is to convert this secondary index into a multilevel index

- secondary index becomes a first level index
- second level index has $b_2 = \lceil \frac{442}{68} \rceil = 7$ blocks
- third level index has $b_3 = \lceil \frac{7}{68} \rceil = 1$ block

Searching the multilevel index takes one block from each level and then to access our record we need

- $t = 3$ block access to index blocks
- plus one additional access to read block from the data file

Total number of block accesses is $t + 1 = 3 + 1 = 4$ which is a big improvement compared to 10 accesses using the secondary index

B-Trees

Challenges to improve insert and delete operations lead to an idea of **dynamic multilevel indexes** or B-trees

- because dynamic multilevel indexes have the same issues with insert and delete operations as single-level ordered indexes

B-trees and B^+-trees are cases of **search tree** data structure

- trees are formed of *nodes*
- nodes have *parent* node except only one node called **root**
- nodes which have children are called **inner nodes** (root node is also inner node)
- nodes without children are called **leaf nodes**
- **level** (**depth**) of a node is always one more than the level of it's parent
  - root node has level 0
- tree is called *balanced* if all leaves are at the same level

Search trees

Search tree of order p is a tree whose nodes can contain pointers and search values

– node has at most p-1 search values and p pointers to their children

– pointers can point to subtrees or can be null pointers

– order of pointers and search values is as follows

$$< P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q > \text{ where } q \leq p$$

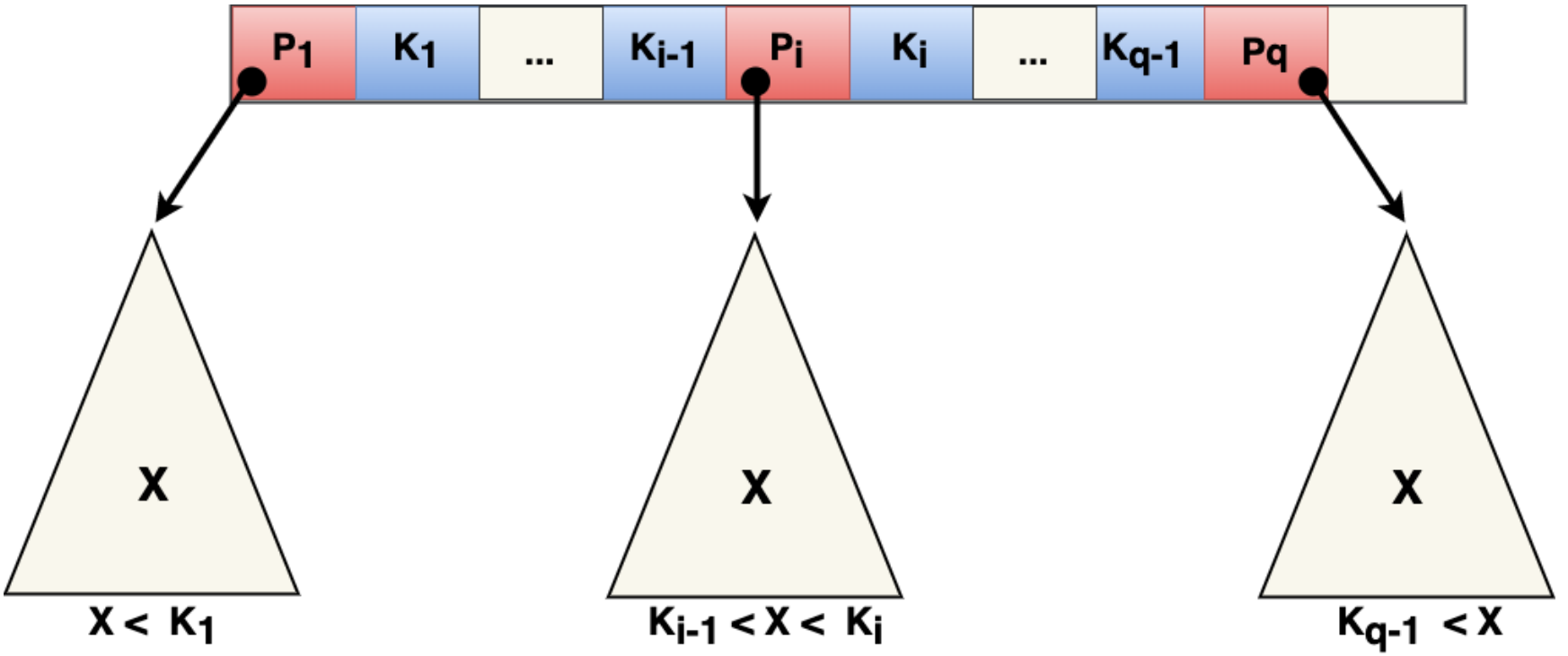
– all search values K_i should be unique

– for search values in a node holds: $K_1 < K_2 < \cdots < K_{q-1}$

– for all search values X in the subtree pointed at by P_i where $1 < i < q$ holds $K_{i-1} < X < K_i$

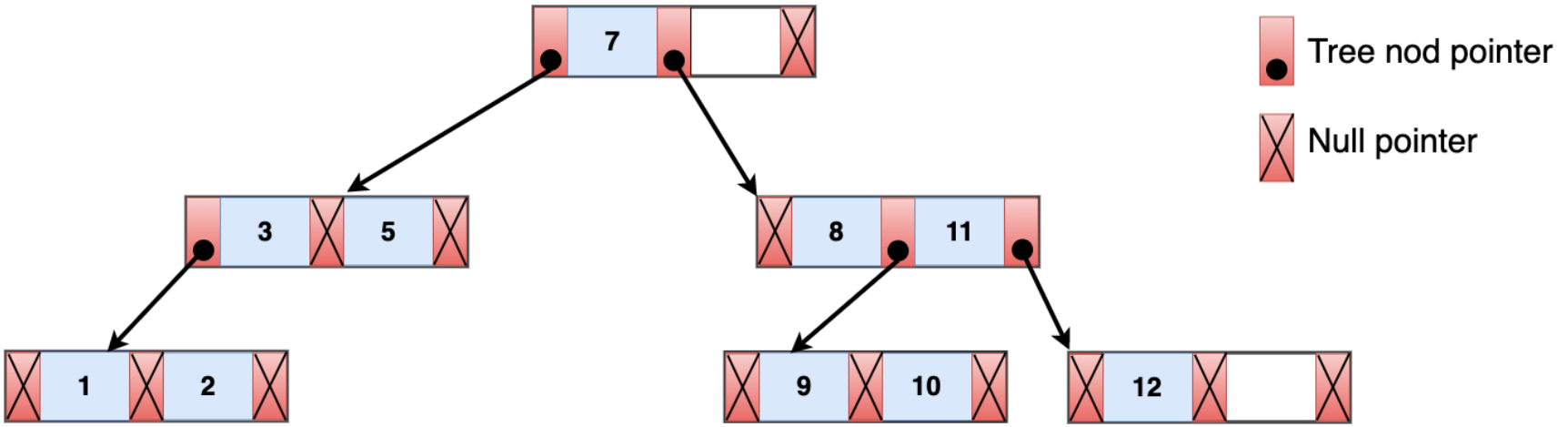
Search trees can be used as a mechanism to access records stored in data files

Search tree - structure



Order of search values and pointers in a search tree

Search tree - example



Main challenge for search trees is how to maintain trees **balanced**

– after delete and insert operations tree can skew in one side and leaf nodes can end up on different levels

Goals for the balanced tree are:

– nodes are evenly distributed so that the depth of the tree is minimal

– search time (time to find arbitrary key) should be uniform

Another challenge is how to reduce the need for restructuring when records are inserted and deleted

B-trees

B-trees are balanced search tree structures with additional constraints to address records on data files

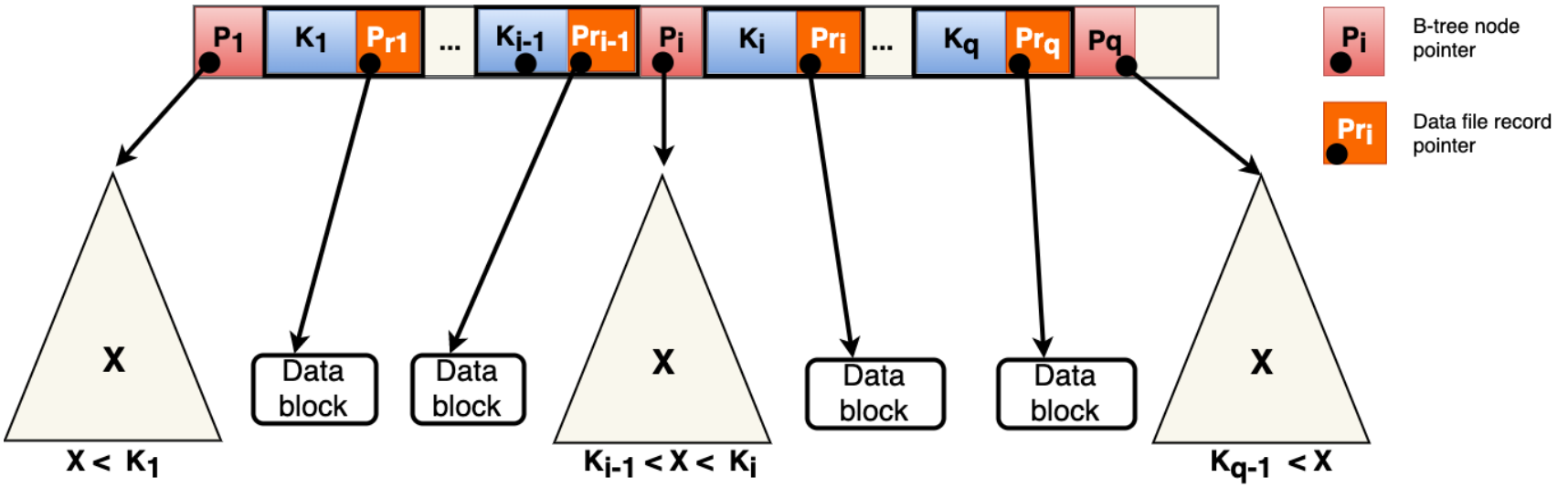
B-tree of order p is a tree whose inner nodes has the following structure:

$$< P_1, < K_1, Pr_1 >, P_2, < K_2, Pr_2 >, \ldots, < K_{q-1}, Pr_{q-1} >, P_q >$$

where $q \leq p$

- P_i is a tree pointer to another node of the B-tree
- K_i is a search value (it holds $K_1 < K_2 < \cdots < K_{q-1}$)
- Pr_i is a data pointer to the record containing the search key K_i(or to the block containing that record)
- each node has at most p tree pointers
- root has at least two tree pointers (if it's not the only node in the tree)
- each node except the root node has at least $\lceil \frac{p}{2} \rceil$ tree pointers

B-trees

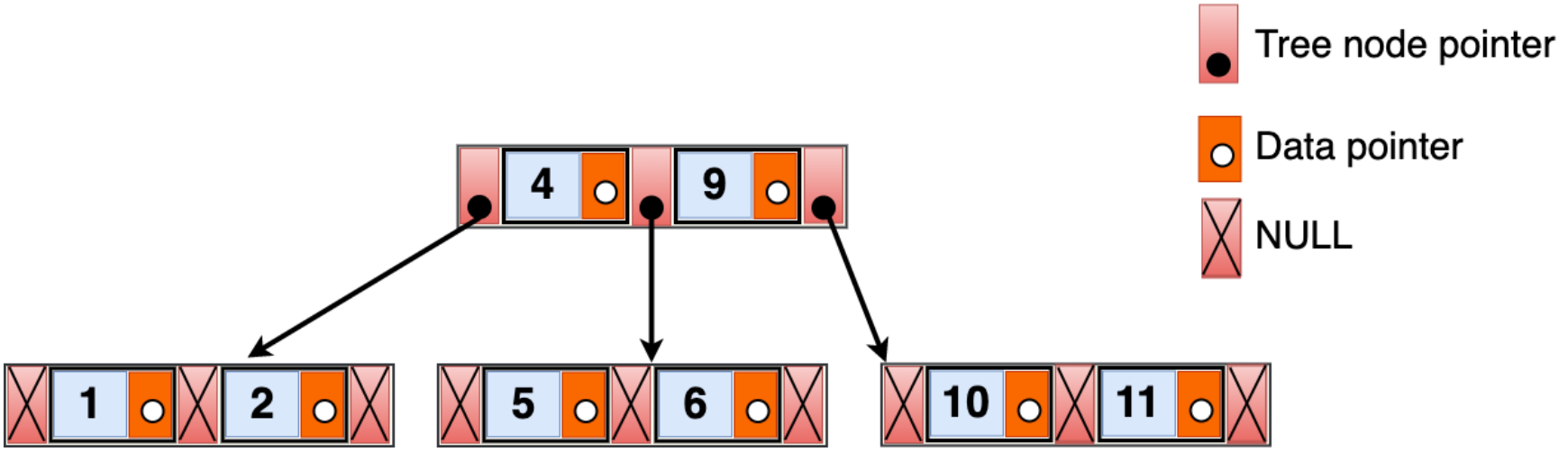


Important property of B-trees is that the nodes are at least half full

- tree is balanced - all leaf nodes are at the same level (depth)
  - tree node pointers of leaf nodes have NULL values
- B-tree can be defined both on a key field and on a nonkey field (can be more fields)
  - for the nonkey fields it is necessary to maintain clusters of blocks with pointers to the file records

B-trees - example

B-tree of order $p = 3$



B-trees - insert operation

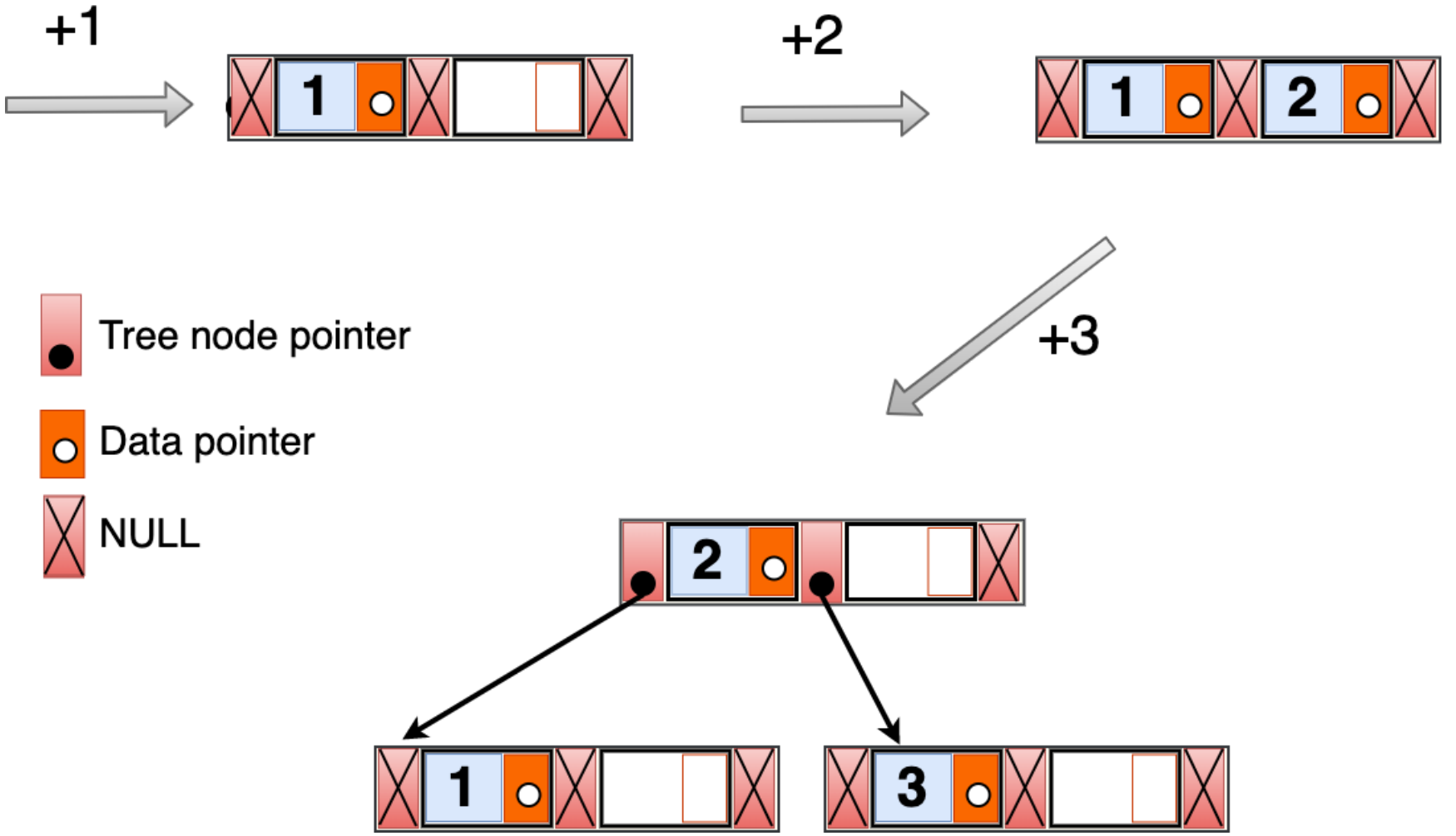
We can assume that a B-tree starts with a single node which is both root and leaf node

- search values (and data pointers) are inserted in the sorted order until the root node is full
- when the root is full it is split in two nodes at the *level 1* and the root at *level 0*
  - middle value is kept in the root node and the rest of elements are split evenly to the left and to the right node at the level 1
- inserting a new value into a non-root node which is full will induce
  - splitting of that node into two nodes at the same level and
  - moving the middle entry and two pointers to the new split nodes into the parent node
  - splitting can propagate all the way to the root node (where splitting the root creates a new level)

A new entry is always inserted directly into a leaf node.

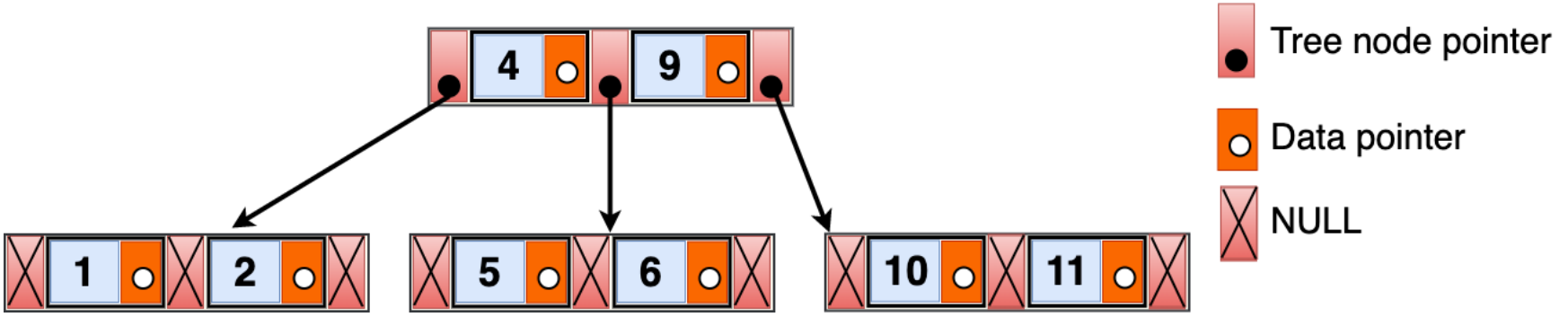
B-trees - inserting a new element

Inserting entries with search values 1, 2, 3 into a b-tree of order 3

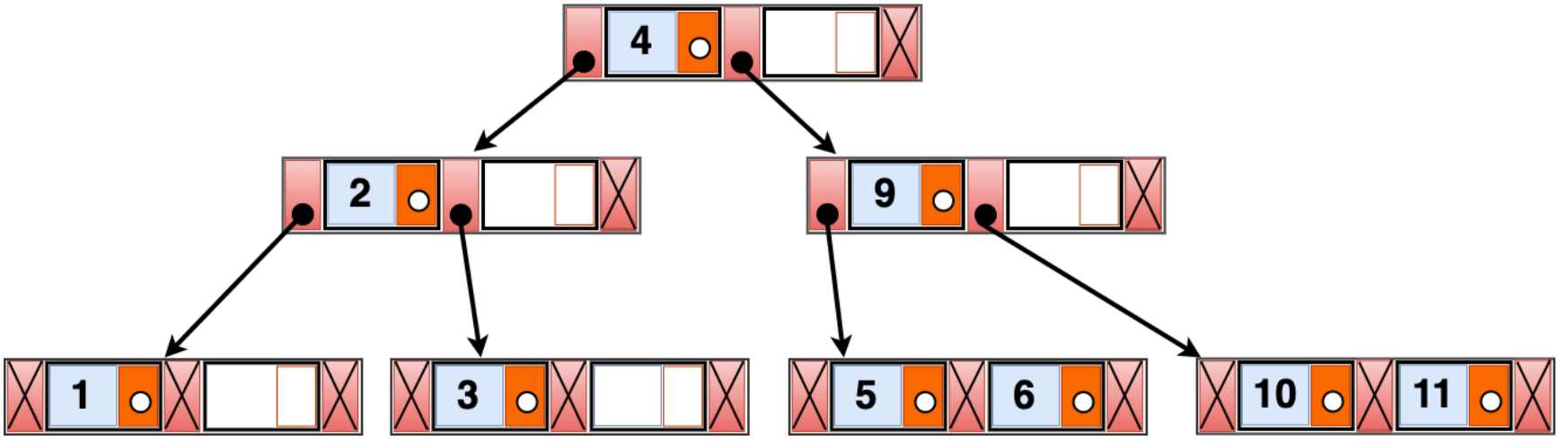


B-trees - inserting a new element

Inserting an entry into the B-tree of order $p = 3$ with propagation to the tree node



After the search value 3 is inserted, we get the following tree:



B-trees - delete operation

Delete operation preserves B-tree properties and consists of the following steps

Simple delete of entry from a leaf node

- delete from a leaf node that doesn't cause the node to be less than half full is simple removing of the entry
- delete operation which causes the root node to be empty leads to deleting the root node (tree is then empty)

Delete operation from an inner node

- deleted value is replaces with the minimal value from the right subtree or with the maximal value from the left subtree
- delete operation from an inner node in this way boils down to the delete operation from a leaf node

B-trees - delete operation -cont.

Deleting operation from a leaf node which causes a node to be less than half full gives rise to a combination of the node entries with it's neighboring nodes in the following way:

1. *rotation* - borrowing a value from a neighbor(sibling) to the left or to the right that can give one value without violating the property to be half full
2. *merging* - node will be merged with it's neighboring node if there is no neighboring node to borrow an entry

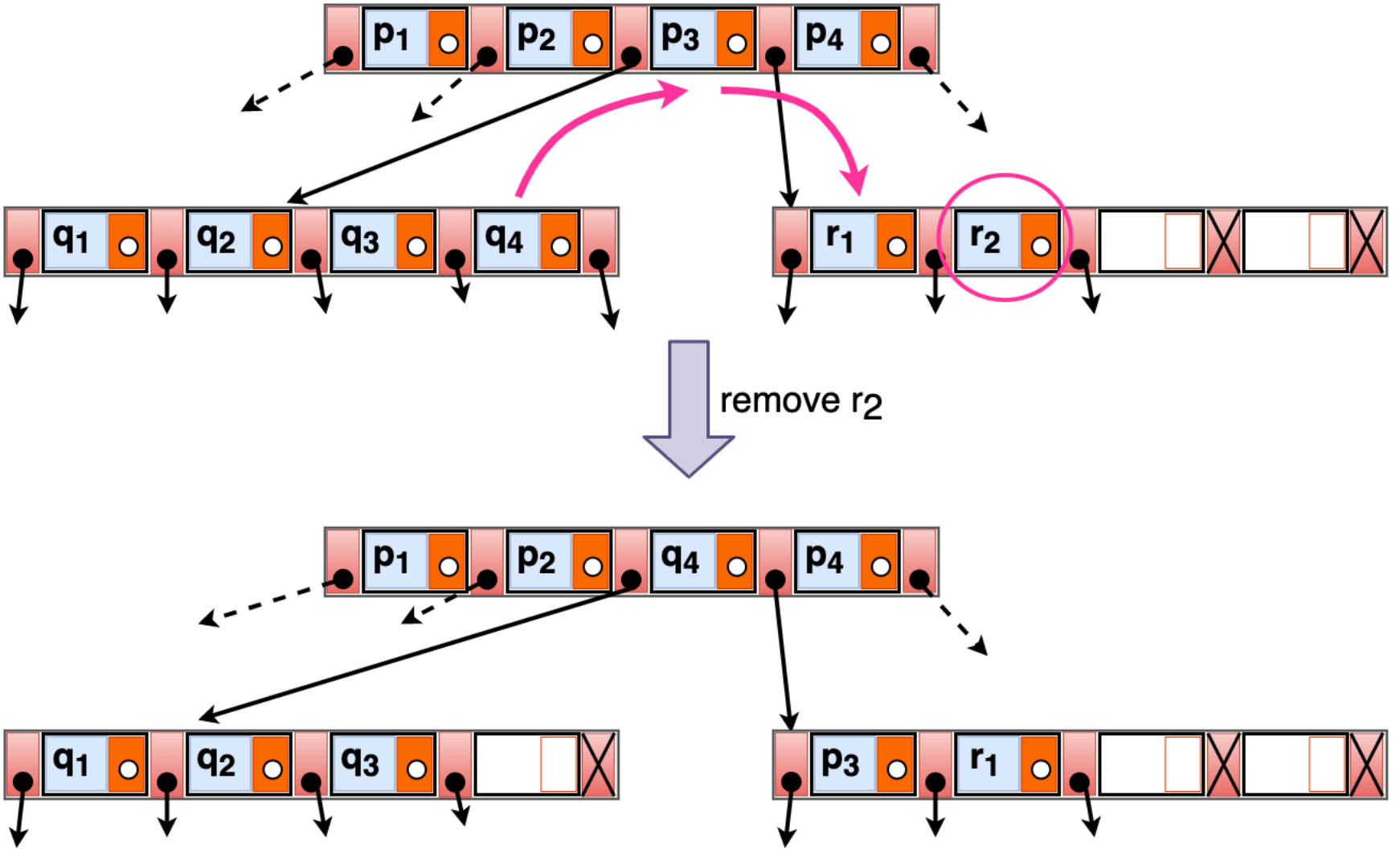
Algorithm first attempts to carry out rotation and if not possible than merging.

Delete operation can propagate to the upper levels

B-Tree simulation

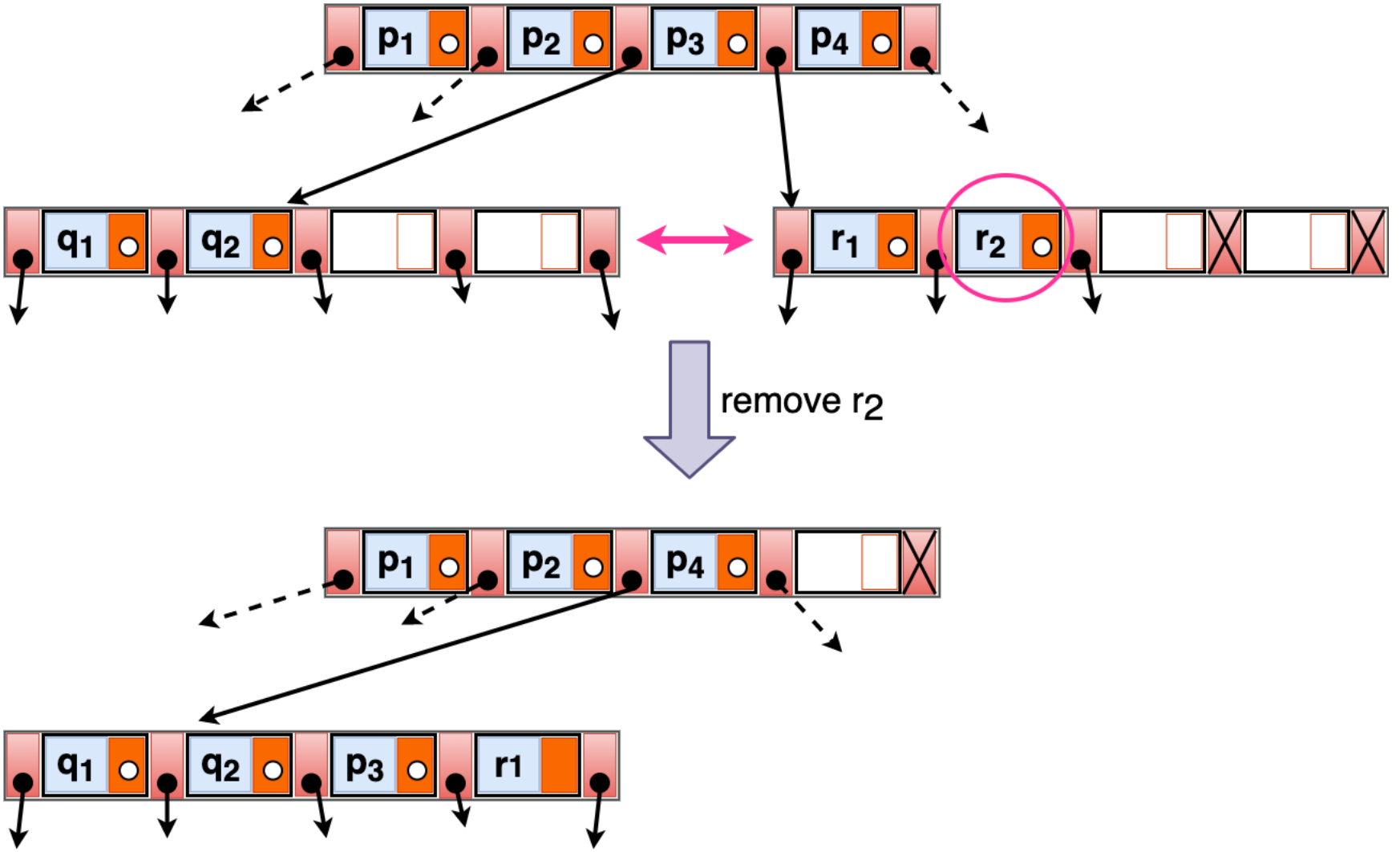
B-trees - delete operation - rotation

Delete entry with the value r_2 (B-tree of order 5)



B-trees - delete operation - merge

Delete entry with the value r_2 (B-tree of order 5)



B-trees - properties

Indexes with some types of B-trees are most frequently used RDBMS

Important advantage of B-trees is that search, insert and delete operation are done with the complexity of $O(log_n)$

Analyses have shown that after numerous random insertions and deletions b-trees stabilize with nodes which are approximately **69% full**

- thanks to this behavior splitting and combining occur only rarely
- insertion and deletion are efficient and only rarely they take more time

B-trees are part of the bigger family of B-trees whose important types are

- B^+ trees, B*-trees B-link trees

B^+-trees

B^+-trees are that type of B-trees which is the most often used for indexes

B^+-trees have data pointers **only** in the leaf nodes.

- leaf nodes are similar to the first level of a multilevel index
- leaf nodes contain entries for **every** value of the search field
- some search values are *repeated* in the internal nodes of the tree

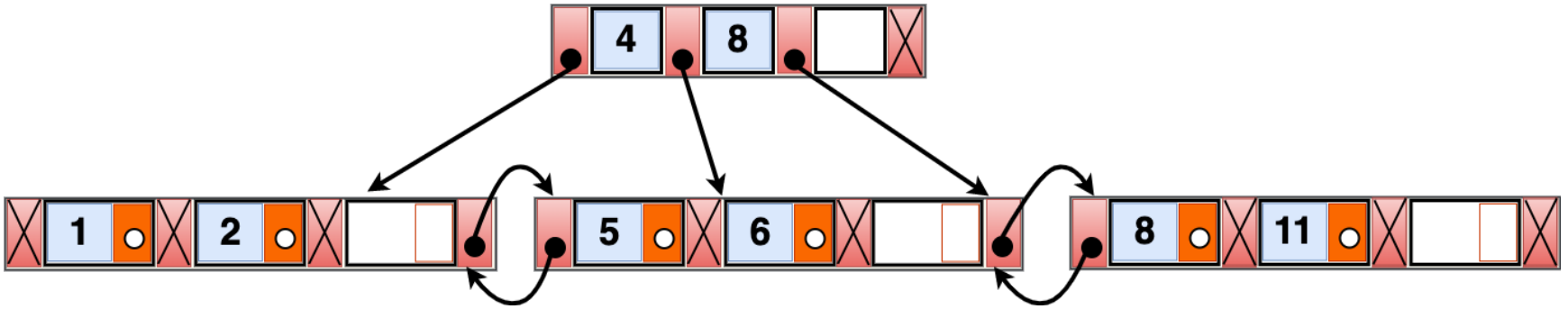
Leaf nodes are usually linked to provide ordered access on the search field of records

B^+ trees can better support concurrency of operations than classical B-trees

PostgreSQL implementation of B-tree is based on B-link trees that supports high concurrency of operations

B^+-tree example

Example of a B^+-tree of order 4



Internal nodes take less space than leaf nodes

- leaf nodes can have different blocking factor from the internal nodes
- $p_{leaf} \neq p_{int}$

B± tree simulation

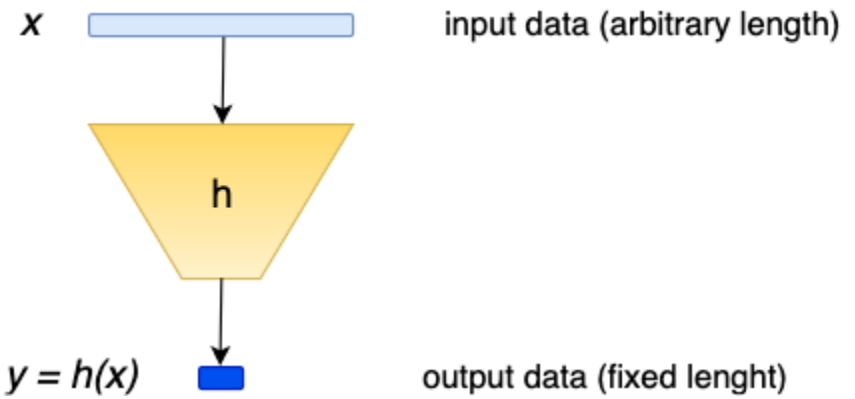
Hash function

A **hash function** is a function which maps data of an arbitrary size to data of a fixed-size length.

$$h : X \rightarrow Y$$

X - set of values of arbitrary size (key space)

Y - set of values of fixed size



Hash functions are sometimes called *compression functions* because cardinality of the output data set is smaller than the cardinality of the input data

– used to assign physical storage locations to key values in DBMS

Hash files

A **hash file** is a type of primary file organization based on hashing techniques

- provides very fast access under specific search conditions
- search condition is an equality condition defined on a **hash field**
- hash field is mostly a key field of the file and in that case it is called **hash key**

  - ***hash function*** *or* ***randomizing function h*** *is applied to the hash field value which yields the address of the block on disk*

- retrieval of most records requires only a single-block access
- once the block is in the main memory buffer, the linear search for the record is carried out
- *complexity* of the lookup operation is $O(1)$ in the best case and $O(n)$ in the worst case
  - hash tables can be on average more efficient than search trees and other lookup structures and they are used for associative arrays, caches, sets, etc.
  - dictionary in Python and associative array in Java are examples of hashing structures

Internal hashing

Internal hashing is a hashing on files located in the main memory

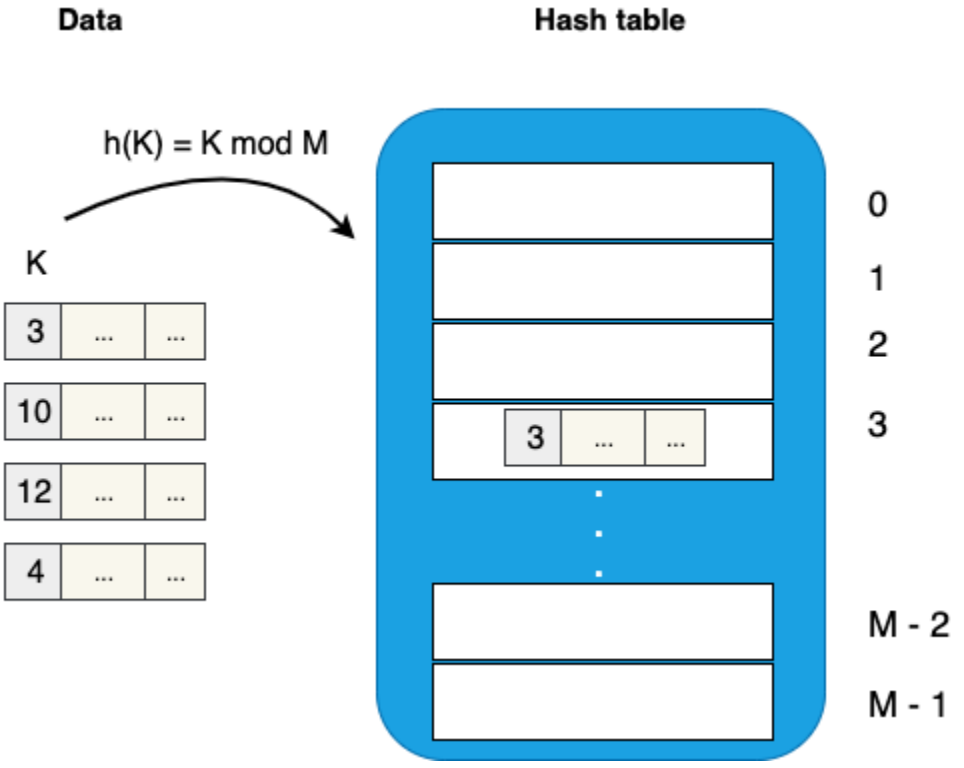
- usually implemented as a **hash table** using an array of records
- assume that the array has M **slots** (M indexes)
- a common cash function can be:

$$h(K) = K \ mod \ M$$

K - integer hash field value
$h(K)$ - remainder of K after division by M

- noninteger hash fields are transformed into integers to apply mod function
  - character string, for instance, can be transformed using ASCII codes and further to integers
  - Example: $A = 65, B = 66, C = 67$ then a hash function can be defined using addition of ASCII codes: $h(`ABC') = 198$

Internal hashing



Problem with collisions

– most hashing functions do not guarantee that distinct values will hash to distinct addresses

Collision

Collision occurs when the hash function generates the same index for more than one key

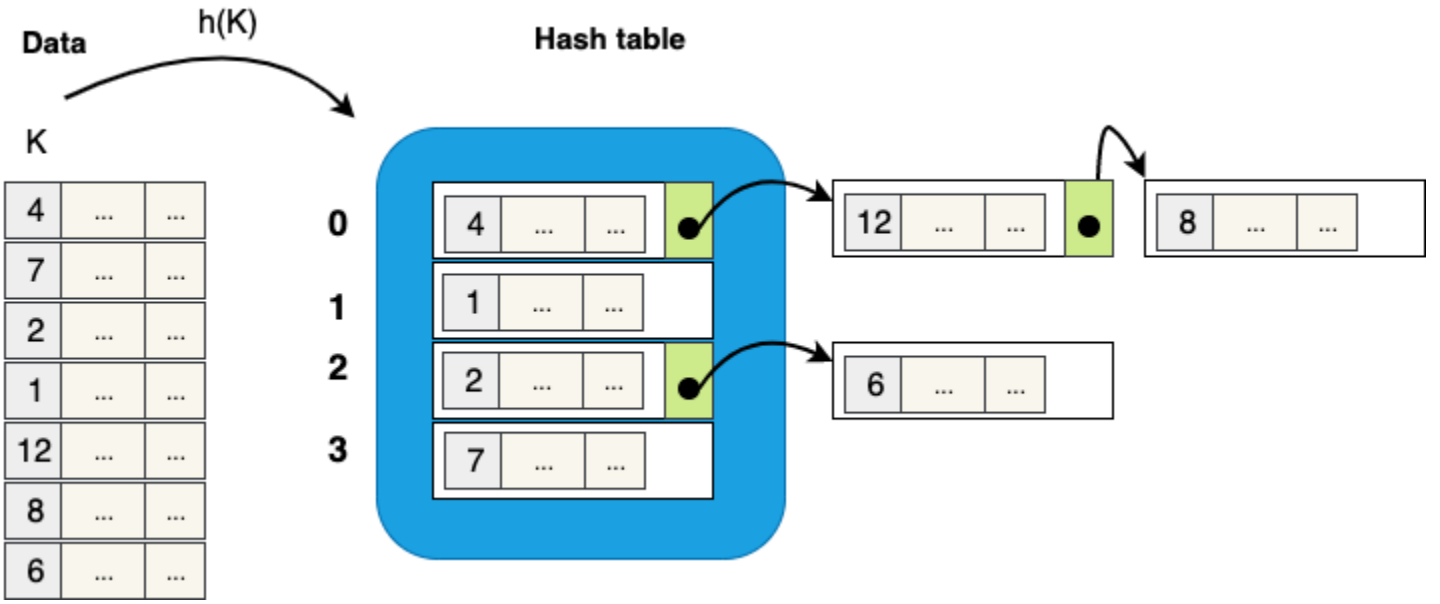
Collision resolution is the process of finding another position for the record when it's hash address is occupied

Some of collision resolution methods are:

- **Chaining** - some overflow locations are kept and pointers are maintained from occupied locations to overflow locations
  - overflow locations are stored as linked lists
- **Open addressing** - from the occupied position, the algorithm checks subsequent positions to find an empty position

Chaining

Chaining is a technique where liked lists of colliding elements are used to overcome collisions



$h(4) = 4\ mod\ 4 = 0$; $h(7) = 7\ mod\ 4 = 3$; $h(2) = 2\ mod\ 4 = 2$;

$h(1) = 1\ mod\ 4 = 1$; $h(12) = 12\ mod\ 4 = 0(collision)$; $h(8) = 8\ mod\ 4 = 0$ (collision);
$h(6) = 6\ mod\ 4 = 2(collision)$

As the number of overflow location grows, grows the complexity of lookup (search) operation

Open addressing

Open addressing uses basic hash table (bucket array) to find empty place to store data when location is already occupied.

Open addressing techniques include the following methods to find empty location

- **linear probing** - interval probes from the occupied location are fixed (1)
- **quadratic probing** - interval probes are calculated using quadratic polynomials
- **double hashing** - second hash function is applied

It has been shown that open addressing can show better performance than chaining if good hashing strategy is applied to deal with collisions

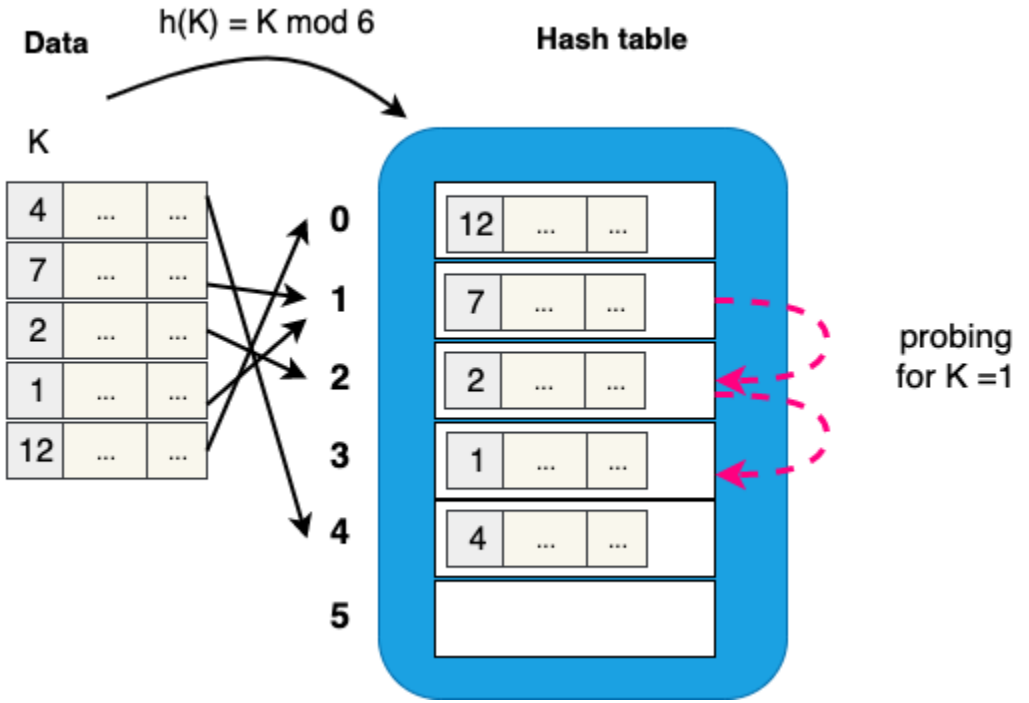
Open addressing with some improvements has many applications

Open addressing - Linear probing

Linear probing is a simple way open addressing where the algorithm looks linearly for an unoccupied location

– the interval between probes is fixed (usually 1)

– not effective if clusters are formed



$h(4) = 4\ mod\ 6 = 4$; $h(7) = 7\ mod\ 6 = 1$; $h(2) = 2\ mod\ 6 = 2$;

$h(1) = 1\ mod\ 6 = 1$ (collision) - checks the next location with index 2 (collision) -> checks the next and is inserted in the place with index 3; $h(12) = 12\ mod\ 6 = 0$

Load factor

Load factor is a parameter which can be monitored to preserve favorable complexity of lookup operations:

$$load\_factor = \frac{n}{k}$$

- n - number of entries in the hash table
- k - number of available locations

load factor around 0.7 guarantees good trade-off between time and space usage and good search efficiency

- default load factor for a HashMap in Java 10 is 0.75

Resizing is necessary when the load factor exceeds some threshold

- requires the change of the hashing function
- automatically triggered on some load factor value
- number of locations grows or shrinks commonly by doubling (*2) or halving(/2)

Hash functions

Hash functions can be applied to fields which are not integer fields

- strings can be, for instance, transformed to integer values by adding ASCII Code values or by multiplying them

Object-oriented languages use hashing frequently to identify objects

Hashing functions can be used for the encryption of entire files

- example SHA254, MD5
- these hashing functions are not fast enough to be used for databases

Advanced hashing functions:

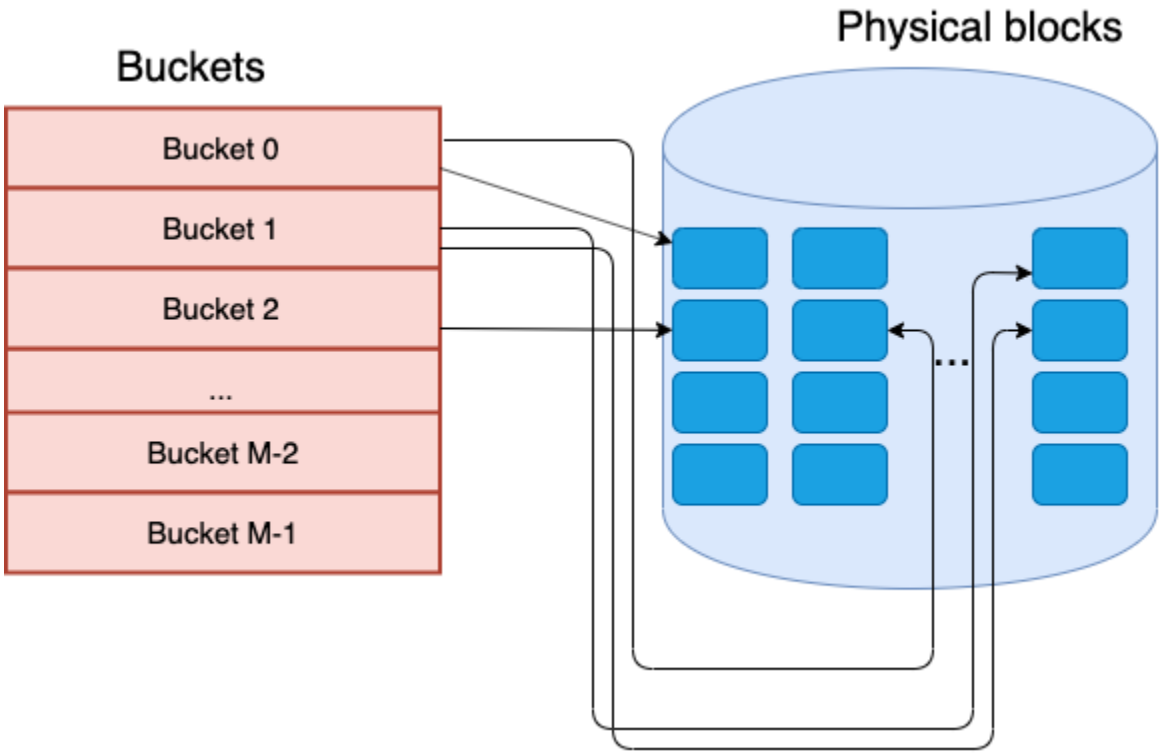
- MurmurHash
- XXHash
- FarmHash
- CRC-64

External hashing

Hashing used for disk files is called **external hashing**

- target address space on disks is divided in **buckets**
- *bucket* can be comprised of one or more blocks (frequently of a cluster of consecutive blocks)
- hashing function maps a key value into a relative block address
- collision problem is reduced by hashing many records to the same bucket
- variation of chaining can be used (similar to the case with internal hashing) to store overflow records when buckets get full
- drawback of hashing is that hashing functions are **not** order preserving - records are not placed in the order of any field

Buckets and blocks (pages)



Buckets contain addresses of physical blocks on disk

Static hashing

Static hashing is a hashing scheme where the number of allocated buckets is fixed

- fixed number of buckets is a serious drawback for dynamic files and can lead to
  - *long lists of overflow records* - if the number of actual records is greater than allocated capacity in buckets
  - *considerable unused space* - if the number of actual records is substantially lower than allocated capacity in buckets
- problems lead to expensive *reorganization* when a new number of blocks *M* is defined and hashing function has been changed
  - since we focus on external files the cost of the reorganization is high

Static hashing

Search operation over the hashing field can be the fastest possible

- complexity of search for an arbitrary record can be done in *O(1)* time in the best case
- if many records are inserted with collisions the search performance worsen and can be close to the complexity of the search operation on linked lists *O(n)*

Searching for a record over some field other than the hash field requires linear search (complexity as in an unordered file *O(n)*)

Delete operation can be implemented by removing the record from the bucket

- removing a record from overflow locations can imply update of linked list of unused locations

Dynamic hashing techniques

Dynamic hashing techniques try to tackle the problem with fixed hash address space in static hashing

- dynamic hash table can grow and shrink as the situation requires

Some of the solutions are:

- **extendible hashing** - stores **address structure** in addition to the data file
- **linear hashing** - no additional address structure
- **dynamic hashing** - address structure based on binary tree data structure

Extendible hashing

Extendible hashing maintains an additional address structure with values of hashed search fields (similar to indexing with the difference that indexes store search keys)

- address structure is an array of 2^d bucket addresses
- d - *global depth* of the directory
- d' - local depth - number of bits according to which bucket content is assigned
  - several directory pointers can point to the same bucket (it is considered only d' bits out of d bits)

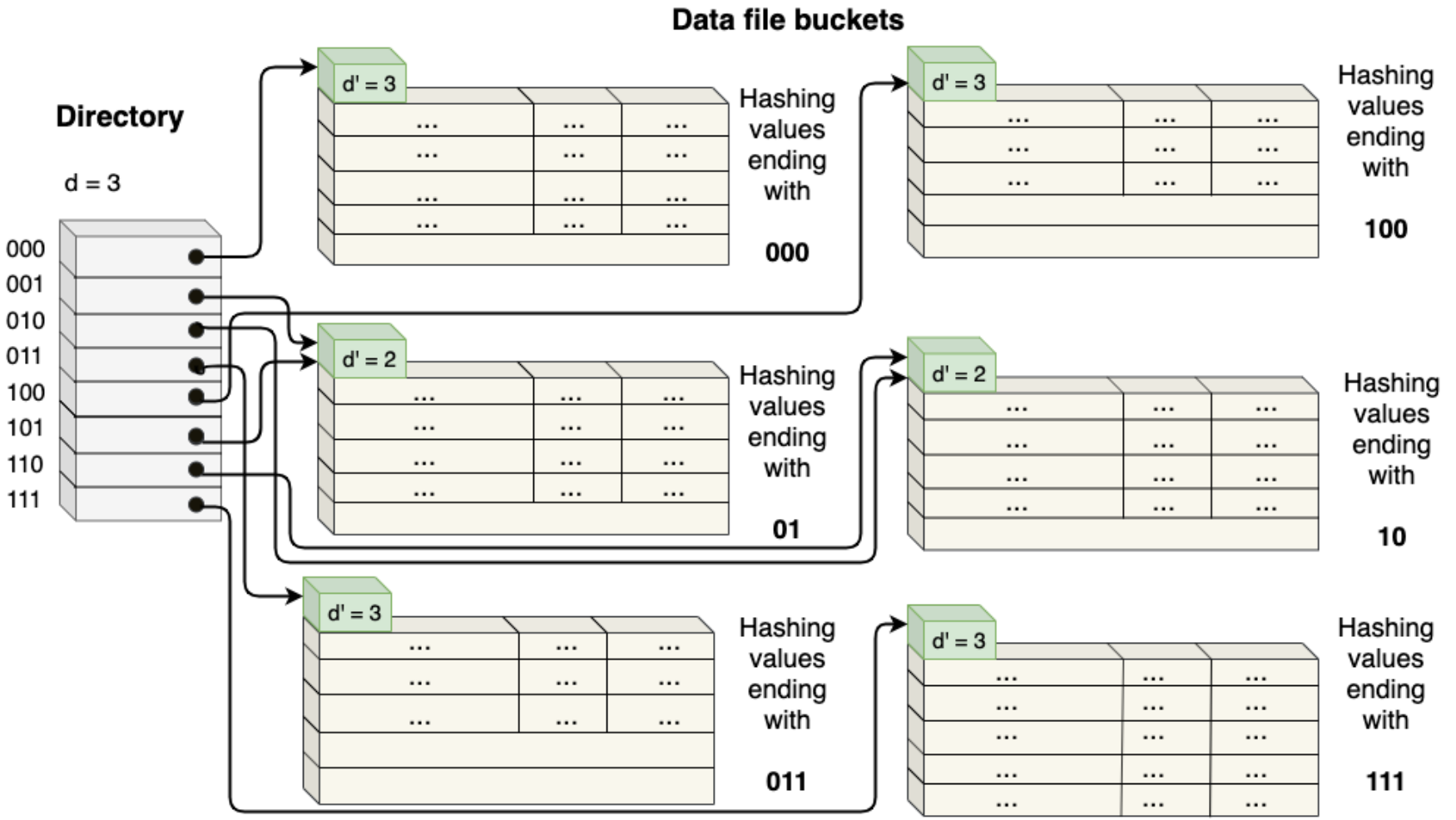
Value d is increased or decreased by one at a time which doubles or halves the number of entries in the directory array

- doubling occurs when the buffer whose $d' = d$ overflows
- halving occurs when $d' < d$ for all buckets after some deletion

Most record retrievals require two block accesses

- one to the directory
- one to the bucket

Extendible hashing



Bit sequences are calculated previously using some hash function.

– Example: $h(\text{`}John') = 001$

Extendible hashing

Doubling, in our example, occurs if the bucket for records whose hash values start with 111 got new element and then

- directory size is doubled having the new depth is $d = 4$
- each directory location is divided into two location that contain the same pointer
- bucket corresponding to 111 values would be separated in two buckets corresponding to 0111 and 1111 hash values
- all other pointers in the directory are doubled and they point to the same buckets

Advantages of extendible hashing

- performances of the file do not degrade as the file grows
- additional buckets are allocated dynamically and no unused space is reserved for future growth
- additional space reserved for the directory is just negligible
- scheme is considered desirable for dynamic files

Difference between Hash files and B-trees

Hash file

```
CREATE INDEX idx_employee_hash ON Employee USING HASH (ssn);
```

– search condition **must be an equality condition** on a hash field, otherwise the system must do the linear search

– hash index is used for the following query:

```
SELECT * FROM Employee where ssn = '987654321';
```

– hash index cannot be used for the following query:

```
SELECT * FROM Employee where ssn > '900000000';
```

– for most records only a single-block access is necessary to retrieve the record (O(1))

Difference between Hash files and B-trees

B-trees

```
CREATE INDEX idx_student ON student(year);
```

– search condition does not have to be equality

– structure enables queries with interval conditions

```
SELECT * FROM student where year>2 and year<5;
```

– search using only part of the key (such as only one attribute from a multiple-attribute key)

– logarithmic complexity ($O(log(n))$)

Review questions

- Search using primary index is more efficient than search through the ordered file itself. Explain why.
- Explain what is the advantage of multilevel indexes in comparison to single-level indexes.
- What disadvantages have ordered files as indexes in comparison to B-trees?
- Explain the complexity of lookup operation in hash tables?
- Why the complexity of lookup operation in hash tables with chaining can be high? Explain your answer.
- What is the difference between internal and external hashing?
- What is the difference between static and dynamic hashing.