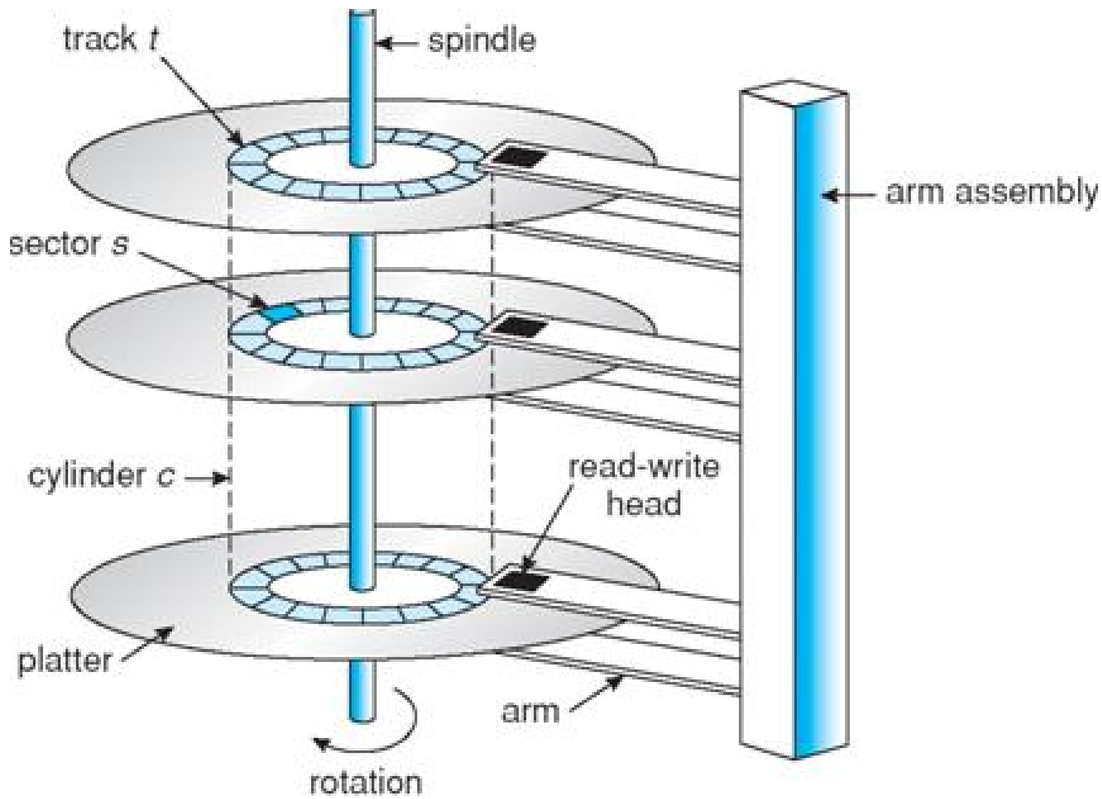
Indexing in databases

The problem

The problem

- Around 10.000.000 cars in the Netherlands
- Query: search a car based on license plate
- Assumptions:
  - A tuple (record) takes 400 bytes
  - A hard disk block contains 16 kbyte, so we have 40 records on a block
  - A disk IO takes ~5 msec
- Maximum search time (complete table scan)
  - 10.000.000 / 40 = 250.000 disk IO
  - 1250 sec ~= 21 minutes
- Required search time : < 1 sec

Hard disk


track t
spindle
arm assembly
sector s
cylinder c
read-write
head
platter
arm
rotation

Main memory vs harddisk

- **Main memory**
  - Typical size : 4 – 256 GB
  - Access time: ~100 nsec (10^{-7} sec)
  - Volatile

- **Harddisk**
  - Typical size : 1 – 14 TB
  - Access time: 5–10 msec   (without clustering)
  - Non-volatile

- **SSD**
  - Typical size : 128 GB – 4 TB (expensive)
  - Access time: ~ 0,1 msec (10^{-4} sec)
  - Non-volatile

- Unit of traffic: block (2 – 32 kbyte)

The solution: indexing

- Indexing enables a quick table search, based on the value of a specific attribute
- Indexing also supports query processing and optimization
- Indexing (automatically) supports primary key maintenance and uniqueness constraints (other candidate keys)
- Syntax for SQL DDL:

```
CREATE INDEX Person_dob_ndx
ON Person (date_of_birth);
CREATE UNIQUE INDEX Person_ppn_ndx
ON Person (passport_number);
```

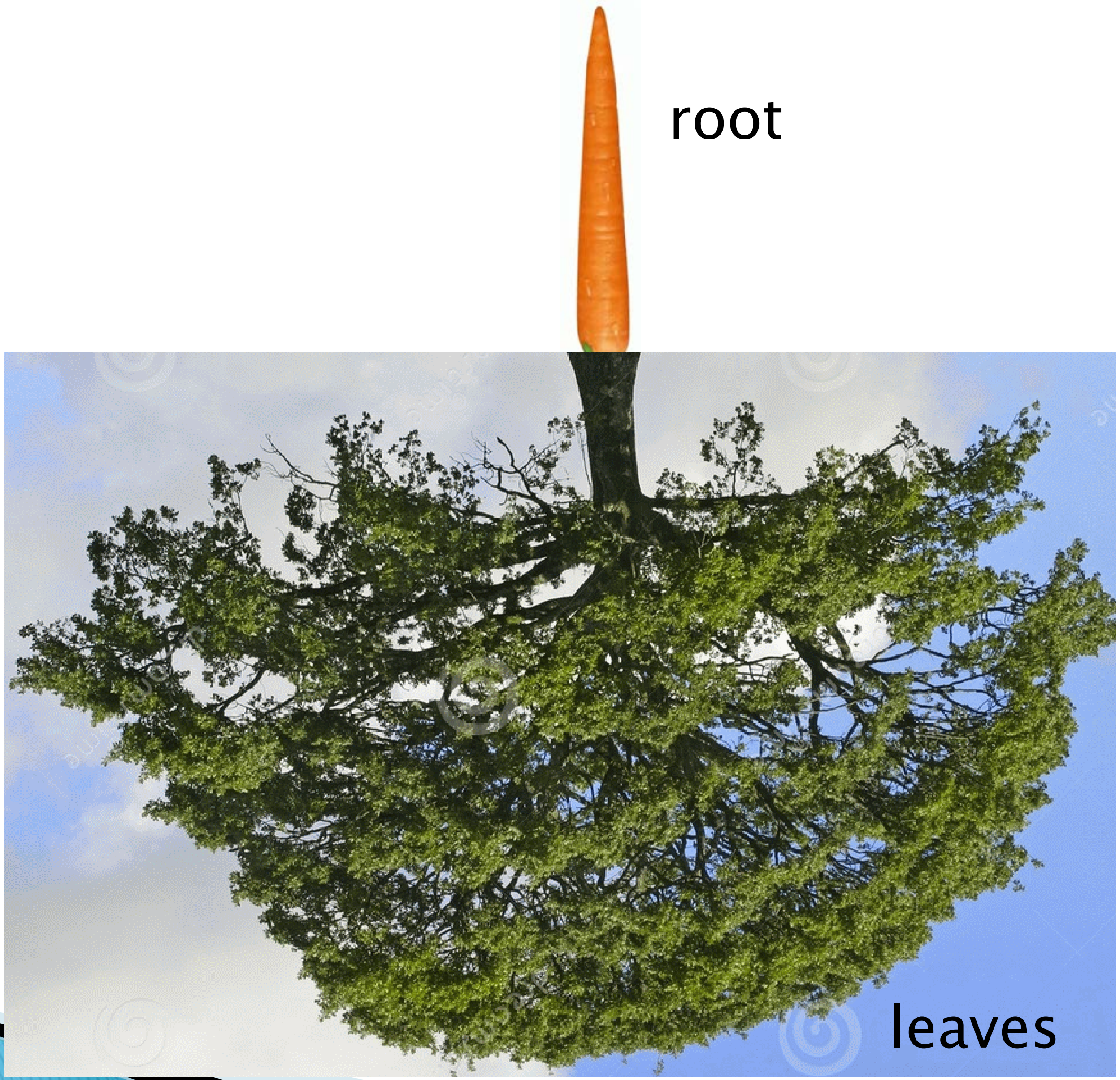
Indexing: how does it work?

Two fundamental techniques

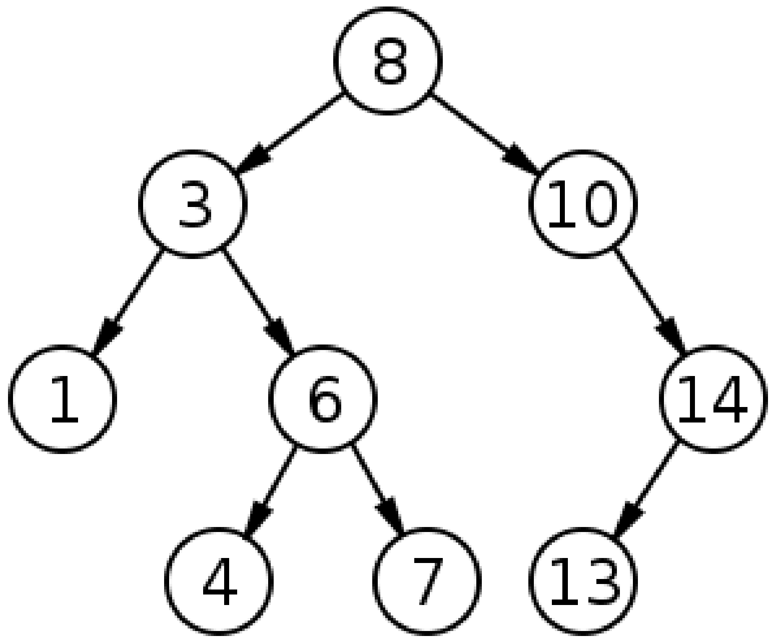
1. Search tree
2. Hash index

Both techniques applicable to main memory as well as external memory

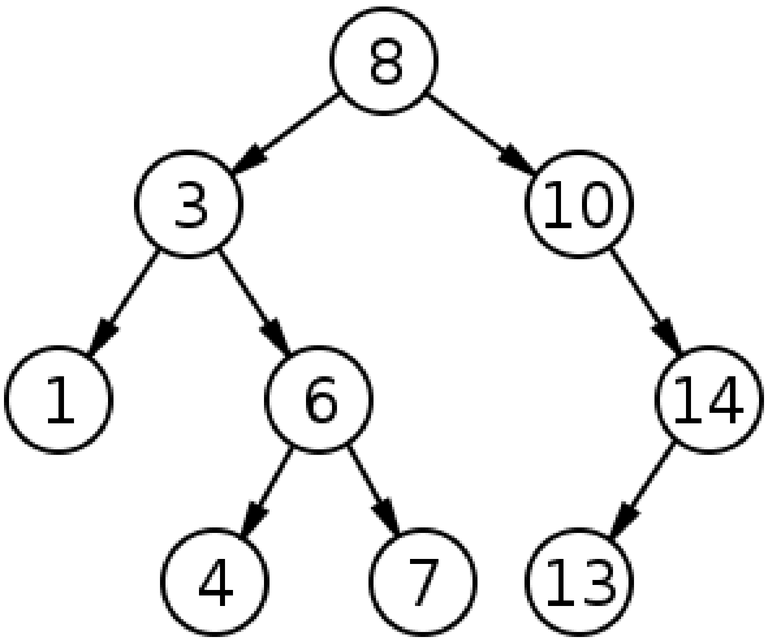
Tree according to computer scientists



Binary search tree

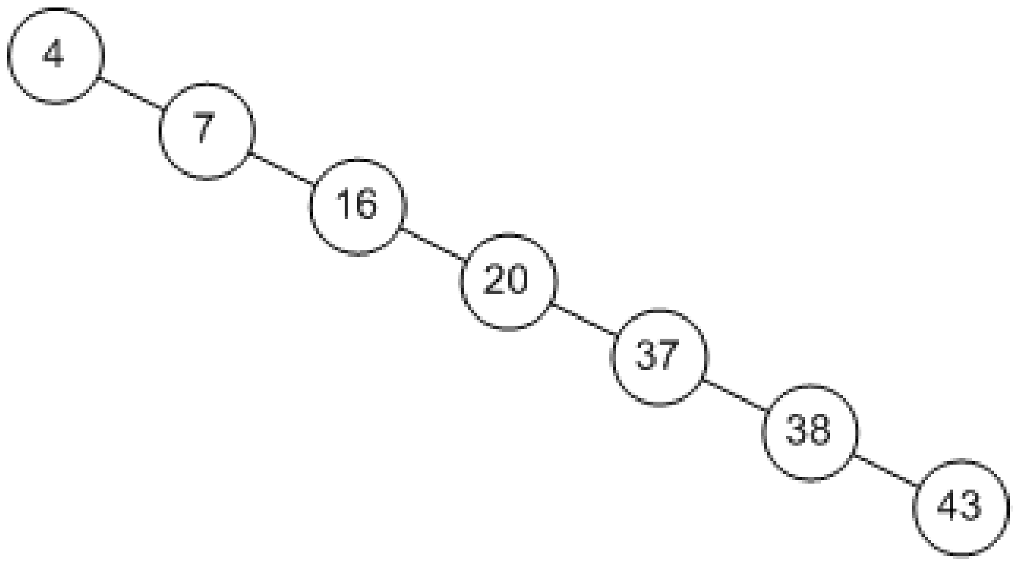


Intermezzo

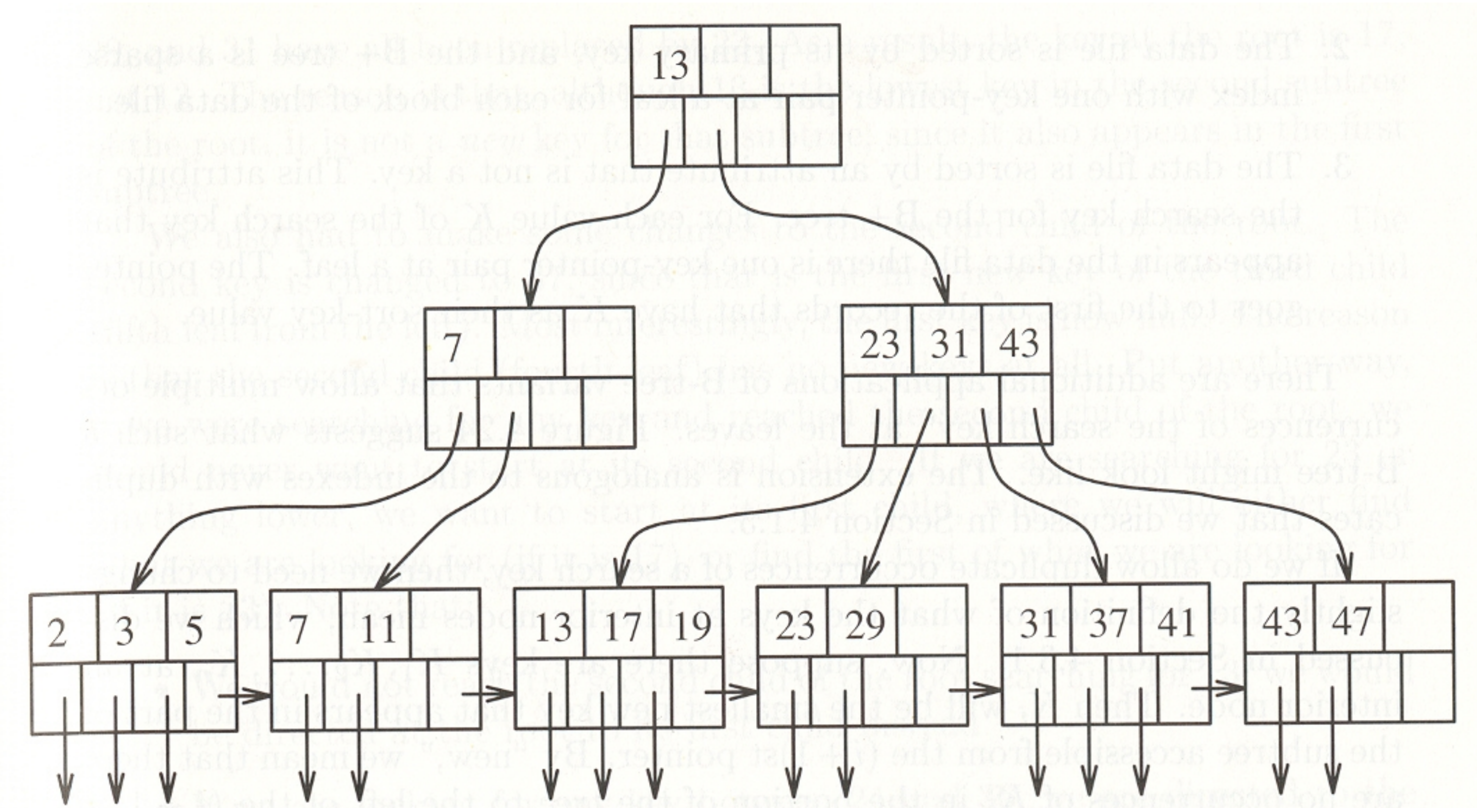


- Add: 2 18 9 5
- How many nodes could a tree with 3 levels contain?
- How many nodes could a tree with N levels contain?
- How many steps does it (roughly) take to fulfill a search action in a tree with M nodes?
- Add: 20 30 40 60 50

Binary search tree: balancing



B-tree (main memory & harddisk)



Update algorithm guarantees 50% filling of nodes

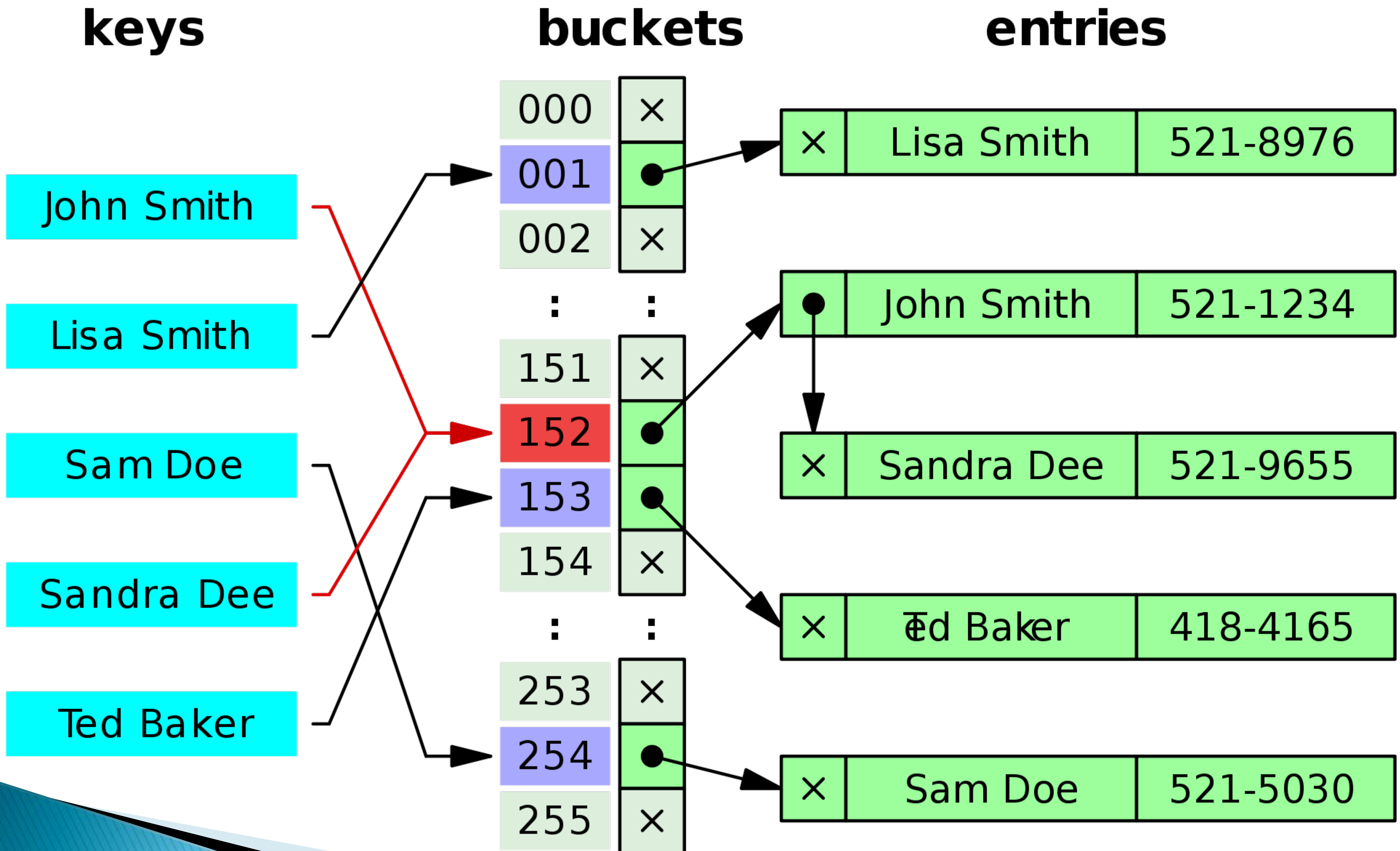
B–tree: characteristics

- Field: 4 byte integer
- Pointer: 8 bytes
- Block size: 16 kbyte
- Content: 683 – 1365 entries per block
- 2 levels: minimum nr of entries = 466489
- 3 levels : minimum nr of entries = 318 million
- 4 levels : minimum nr of entries = 217 billion

Hash table

- Memory reservation of N buckets: addresses 0..N−1
- Hashfunction *f*
  - Domain: all possible attribute values
  - Codomain: 0..N−1
- Hopefully, *f* distributes the values neatly

Hash table


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

Indexing: final words

- Hash indexing has a theoretical advantage: one disk access versus $^{k}\log(M)$ for B-tree
- Hash indexing has a fundamental disadvantage: range queries are not supported
- The k of $^{k}\log(M)$ is very large, so $^{k}\log(M)$ hardly exceeds 3 ...
- ... while the root of the B-tree (and possible the second level nodes) are often kept in main memory
- Overall, the B-tree is the winner