CS 6400 A

Database Systems Concepts and Design

Lecture 8
09/16/24

Logistics

Please sign up for project groups on canvas!

- 56 people still unassigned
- Project proposal due Oct 2 (no late day)

Midterm logistics

- Sep 25 (in class)
- Open book open notes (no laptop)
- Contents covered: up until Sep 9 lecture Design Theory II
- Review lecture next Monday

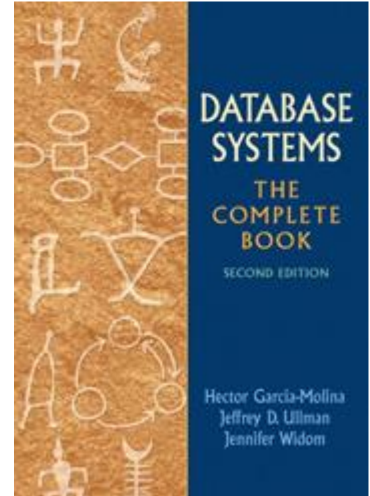
Agenda

1. Index Overivew
2. Index structure basics
3. B+-Trees

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 14.1: Index-Structure Basics
- Chapter 14.2: B-Tree



Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang and CS145 (Intro to Big Data Systems) taught by Peter Bailis.

1. Index Overview

Index Motivation

Person(name, age)

Suppose we want to search for people of a specific age

First idea: Sort the records by age… we know how to do this fast!

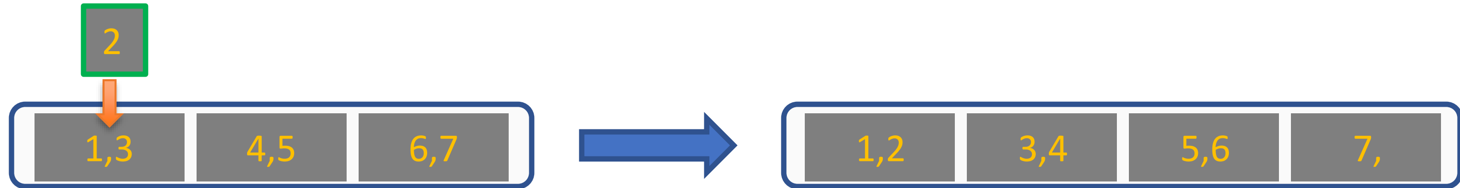
How many IO operations to search over *N sorted* records?

- Simple scan: *O(N)*
- Binary search: $O(\log_2 N)$

Could we get even cheaper search? E.g. go from $\log_2 N \rightarrow \log_{200} N$?

Index Motivation

What about if we want to **insert** a new person, but keep the list sorted?



We would have to potentially shift *N* records, requiring up to ~ 2*N/P IO operations (where P = # of records per page)!

- We could leave some "slack" in the pages…

Could we get faster insertions?

Index Motivation

What about if we want to be able to search quickly along multiple attributes (e.g. not just age)?

- We could keep multiple copies of the records, each sorted by one attribute set… this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

We'll create separate data structures called indexes to address all these points

Indexes: High-level

An *index* on a file speeds up selections on the *search key fields* for the index.

- Search key properties
  - Any subset of fields
  - is **not** the same as *key of a relation*

Example:

Product(name, maker, price)

On which attributes would you build indexes?

More precisely

An *index* is a **data structure** mapping search keys to sets of rows in a database table

- Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table

An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)

- We'll cover both, but mainly consider secondary indexes

Operations on an Index

Search: Quickly find all records which meet some *condition on the search key attributes*

- Point queries, range queries, …

Insert / Remove entries

- Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance

Using Indexes in SQL

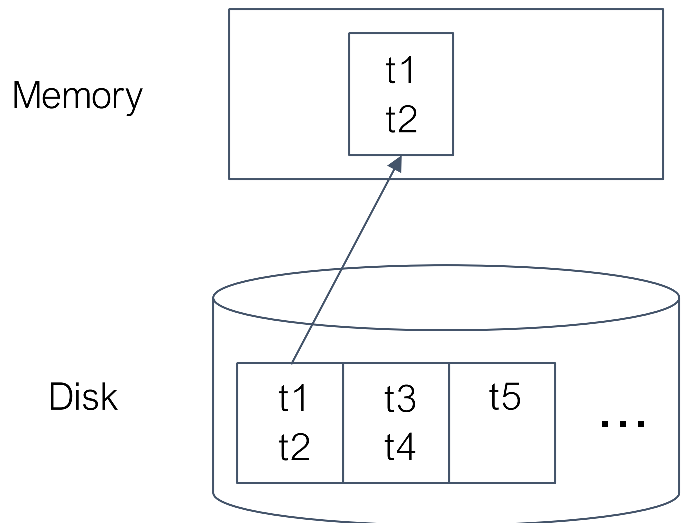
- An index is used to efficiently find tuples with certain values of attributes
- An index may speed up lookups and joins
- However, every built index makes insertions, deletions, and updates to relation more complex and time-consuming

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

```
DROP INDEX KeyIndex;
```

Simple cost model

- Multiple tuples are stored in blocks on disk
- Every block needed is always retrieved from disk
- Disk I/Os are expensive

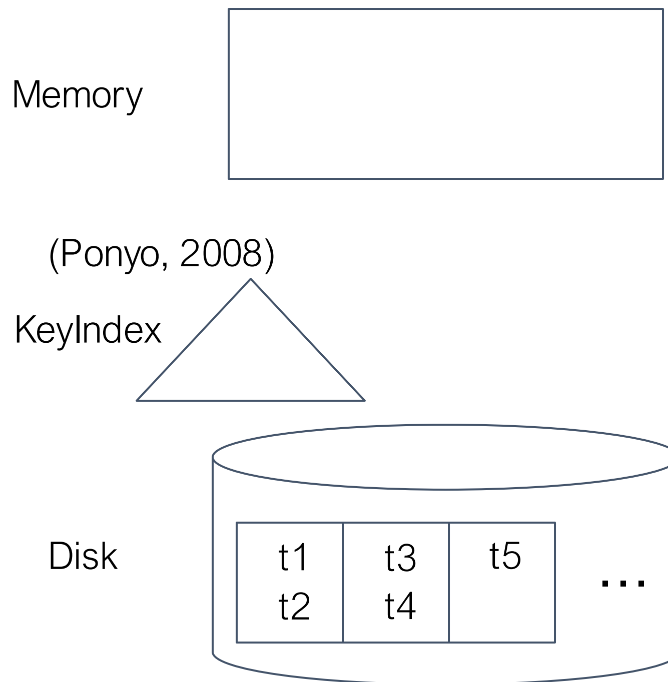


Index on a key

- An index on a key is often useful
- Retrieve at most one block to memory for tuple
  - Possibly other blocks for the index itself

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

```
SELECT *
FROM Movies
WHERE title = 'Ponyo' AND year = 2008;
```

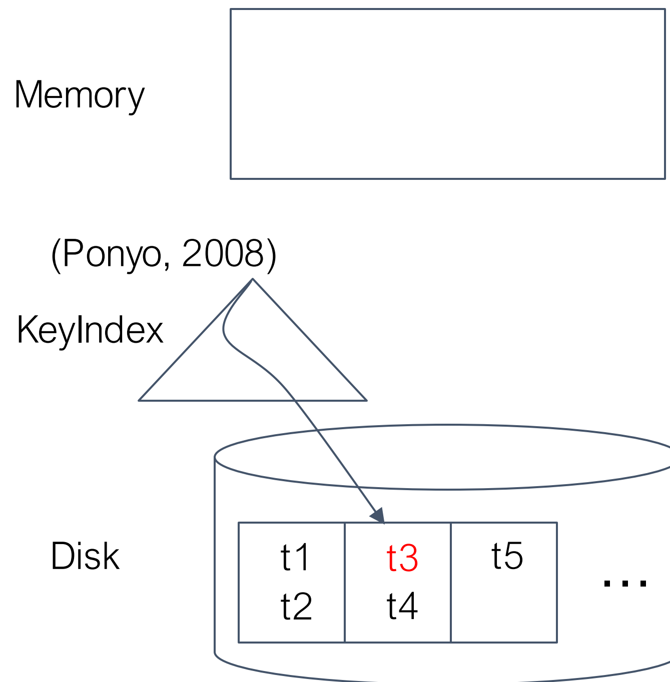


Index on a key

- An index on a key is often useful
- Retrieve at most one block to memory for tuple
    - Possibly other pages for the index itself

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

```
SELECT *
FROM Movies
WHERE title = 'Ponyo' AND year = 2008;
```

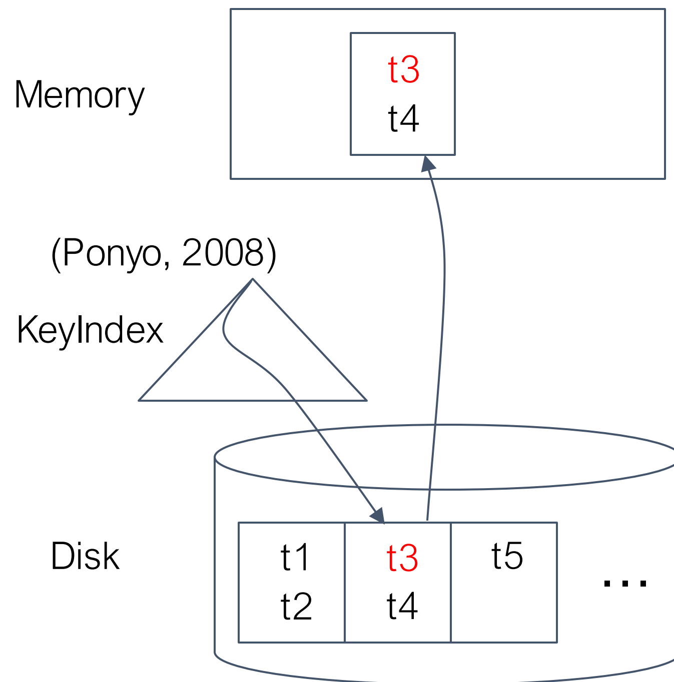


Index on a key

- An index on a key is often useful
- Retrieve at most one block to memory for tuple
    - Possibly other pages for the index itself

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

```
SELECT *
FROM Movies
WHERE title = ‘Ponyo’ AND year = 2008;
```



Indexes can be used in joins

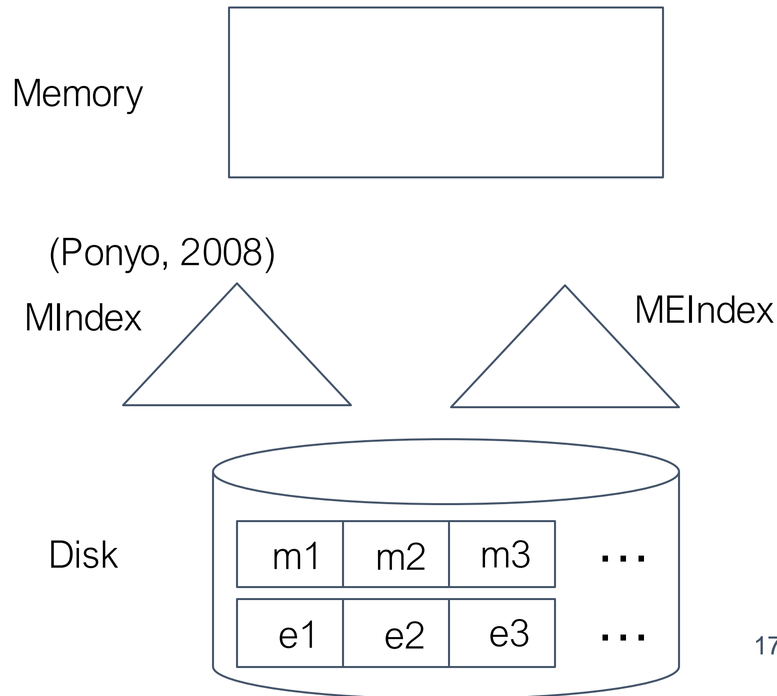
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = ‘Ponyo’ AND year = 2008
  AND producerC# = cert#;
```



Indexes can be used in joins

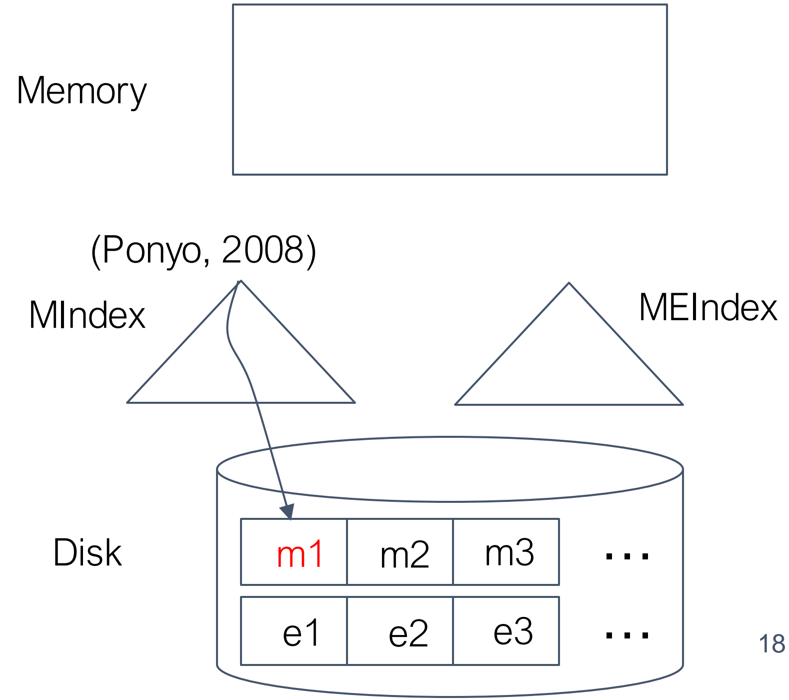
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = ‘Ponyo’ AND year = 2008
  AND producerC# = cert#;
```



Indexes can be used in joins

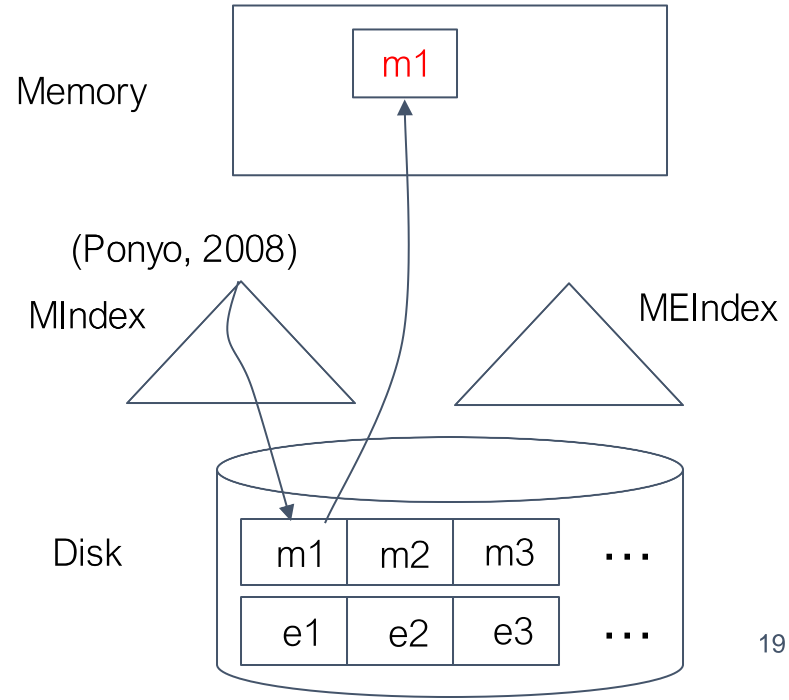
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = ‘Ponyo’ AND year = 2008
  AND producerC# = cert#;
```



Indexes can be used in joins

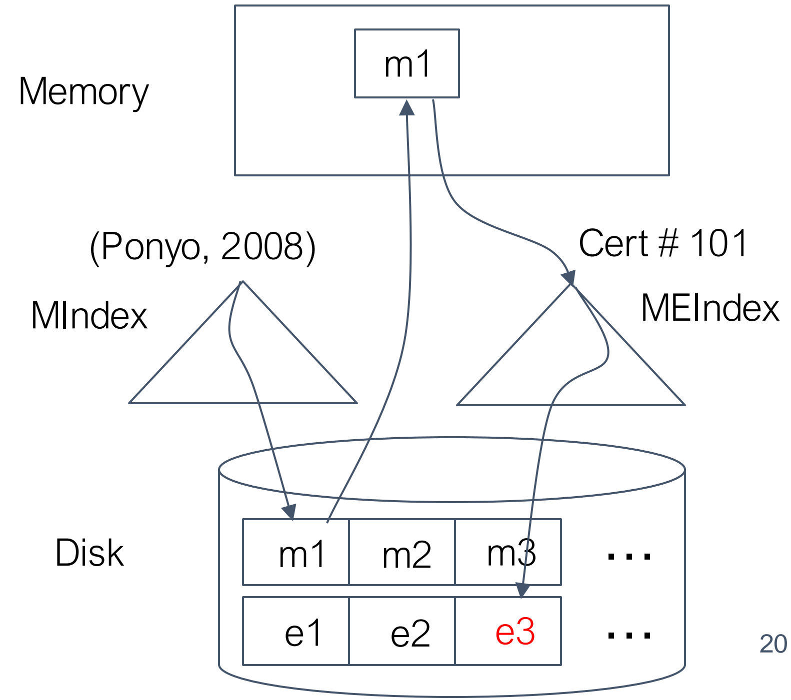
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Ponyo' AND year = 2008
  AND producerC# = cert#;
```



Indexes can be used in joins

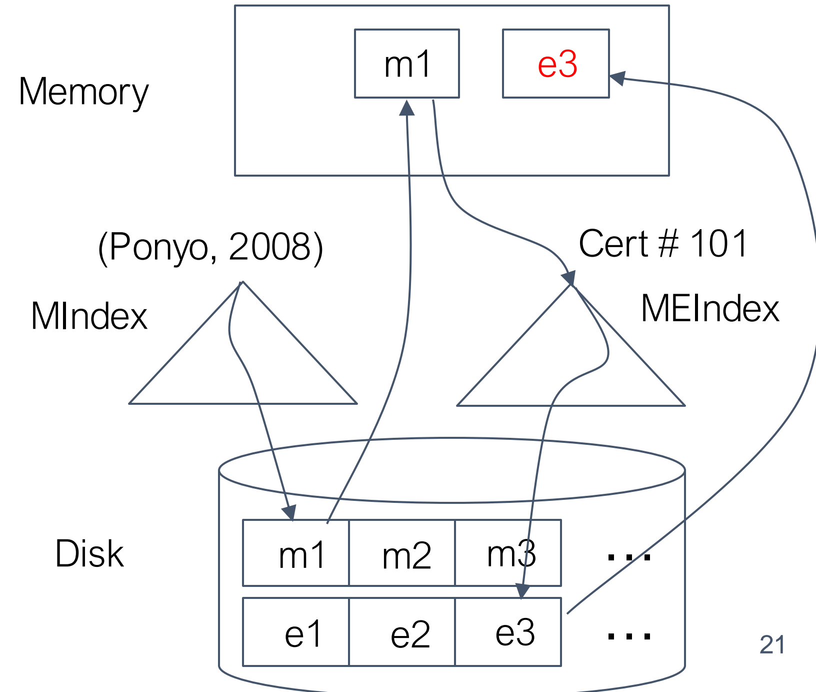
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = ‘Ponyo’ AND year = 2008
  AND producerC# = cert#;
```



2. Index Structure Basics

Sequential file

- A file containing tuples of a relation sorted by their primary key

10	
20	

30	
40	

50	
60	

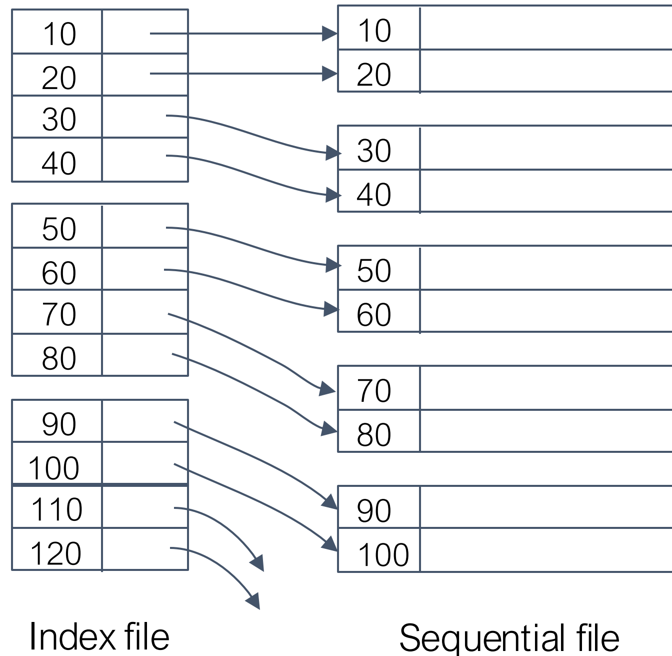
70	
80	

90	
100	

Sequential file

Dense index

- A sequence of blocks holding keys of records and pointers to the records



Dense index

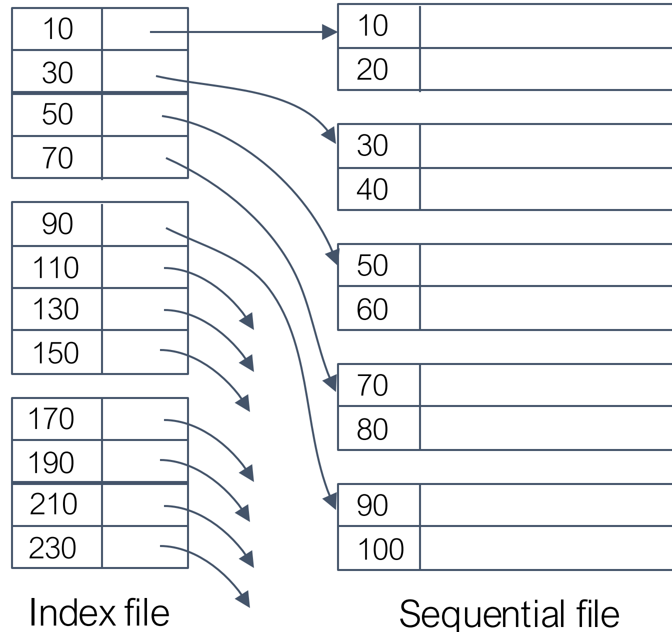
Given key K, search index blocks for K, then follow associated pointer

Why is this efficient?

- Number of index blocks usually smaller than number of data blocks
- Keys are sorted, so we can use binary search
- The index may be small enough to fit in memory

Sparse index

- Has one key-pointer pair per block of the data file
- Uses less space than dense index, but needs more time to find a record



Exercise #1

Suppose a block holds 3 records or 10 key-pointer pairs

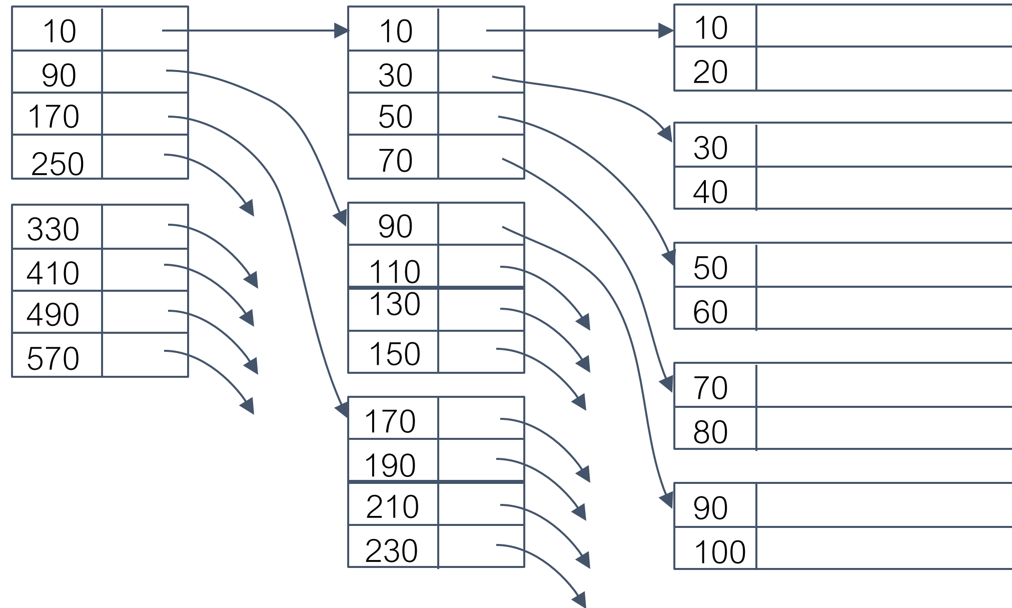
If there are n records in a data file, how many blocks are needed to hold

- The data file and a dense index
- The data file and a sparse index

Multiple levels of index

If the index file is still large, add another level of indexing

- Basic idea of the B+-tree index

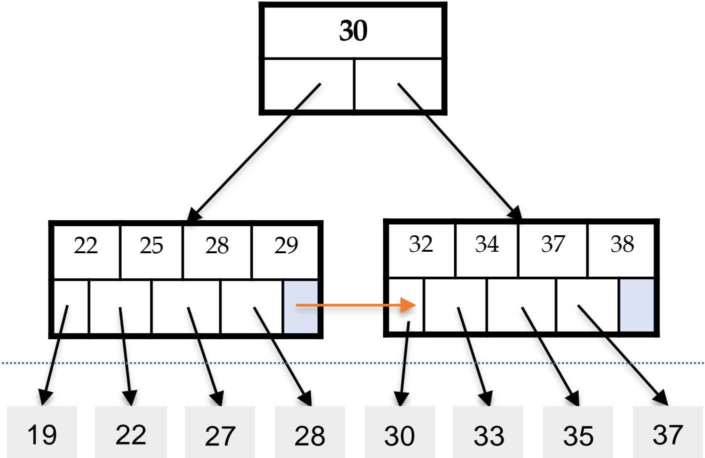


Q: Should the blocks of additional levels be dense or sparse?

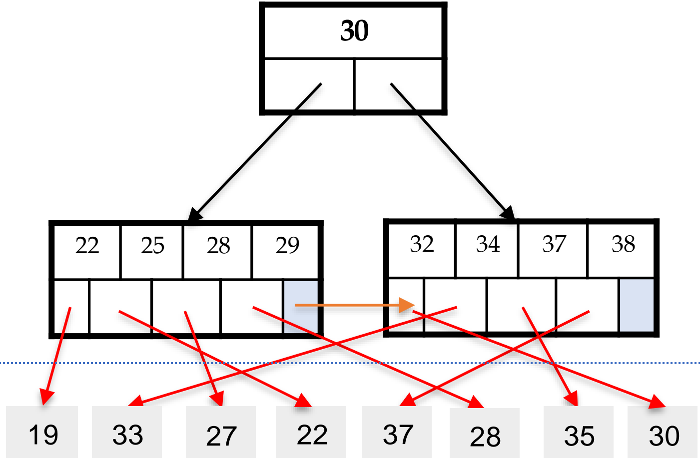
Clustered Indexes

An index is <u>clustered</u> if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Index Entries



Data Records

Clustered: often on primary key

Unclustered

Non-clustered/secondary index

Unlike a clustered index, does not determine the placement of records

20	
40	

10	
20	

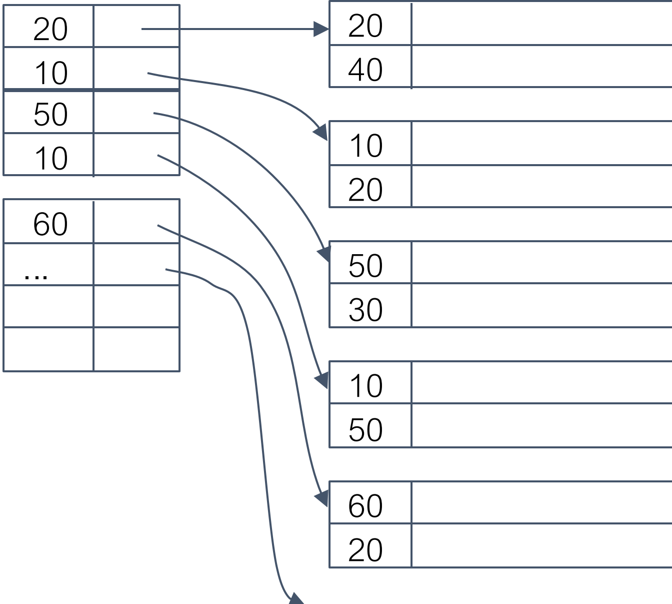
50	
30	

10	
50	

60	
20	

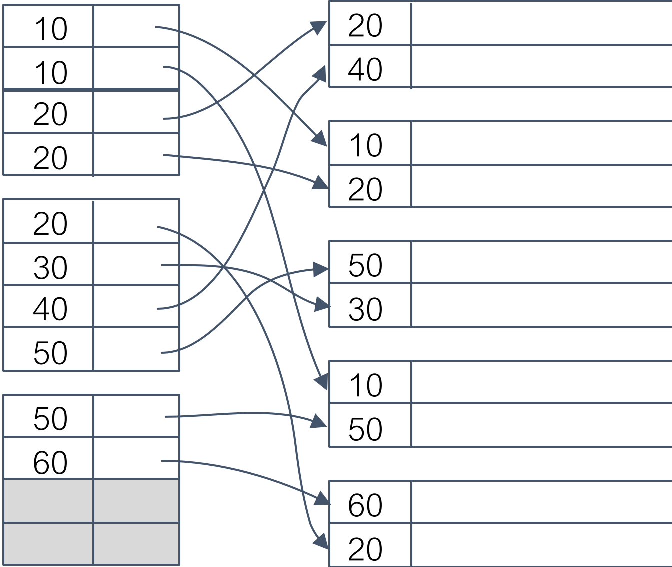
Non-clustered/secondary index

Using a sparse index doesn't make sense



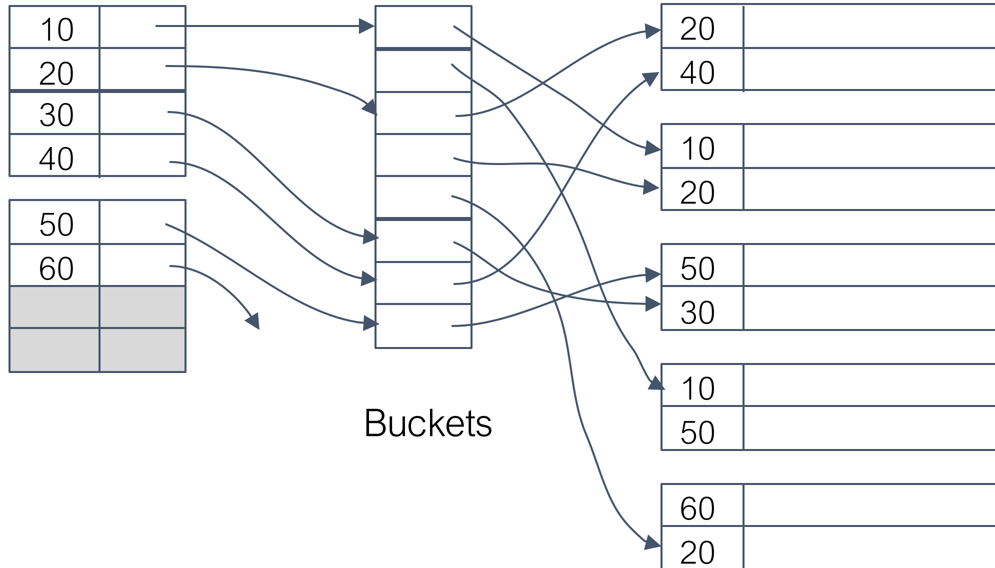
Non-clustered/secondary index

As a result, secondary indexes are always dense



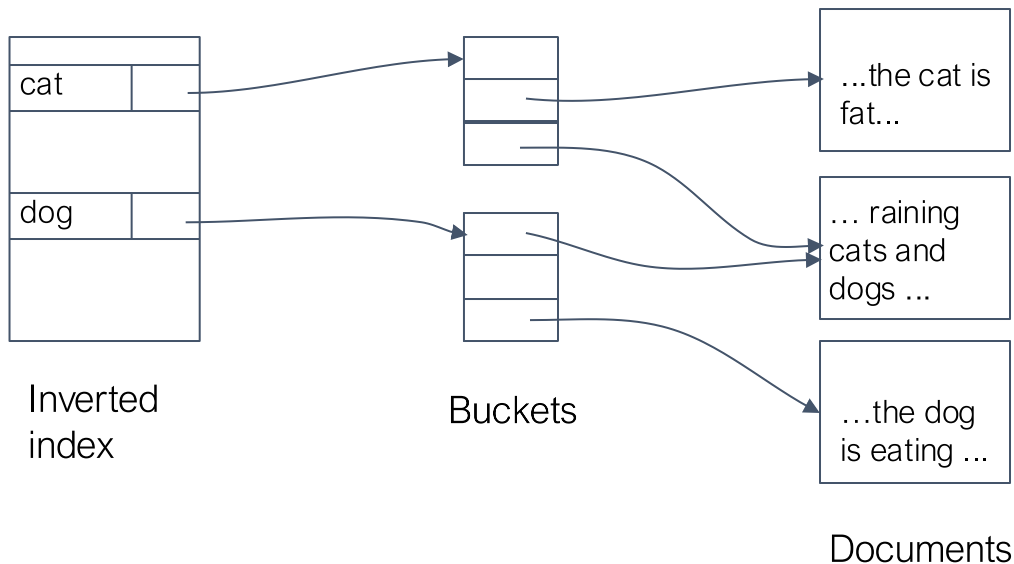
Secondary index

To remove redundant keys in secondary index file, use level of indirection



Inverted index

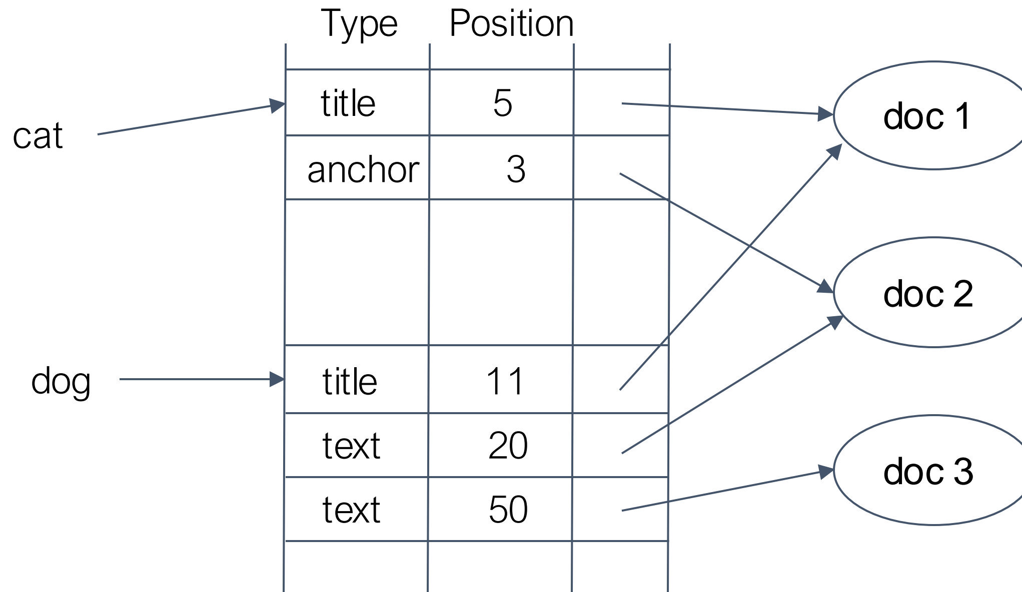
- Previous idea is used in text information retrieval
- Search for documents containing “cat” or “dog” (or both)



Store more information in inverted index

Can answer more complex queries like:

- Find documents where “dog” and “cat” are within 10 words
- Find documents about dogs that refer to other documents about cats

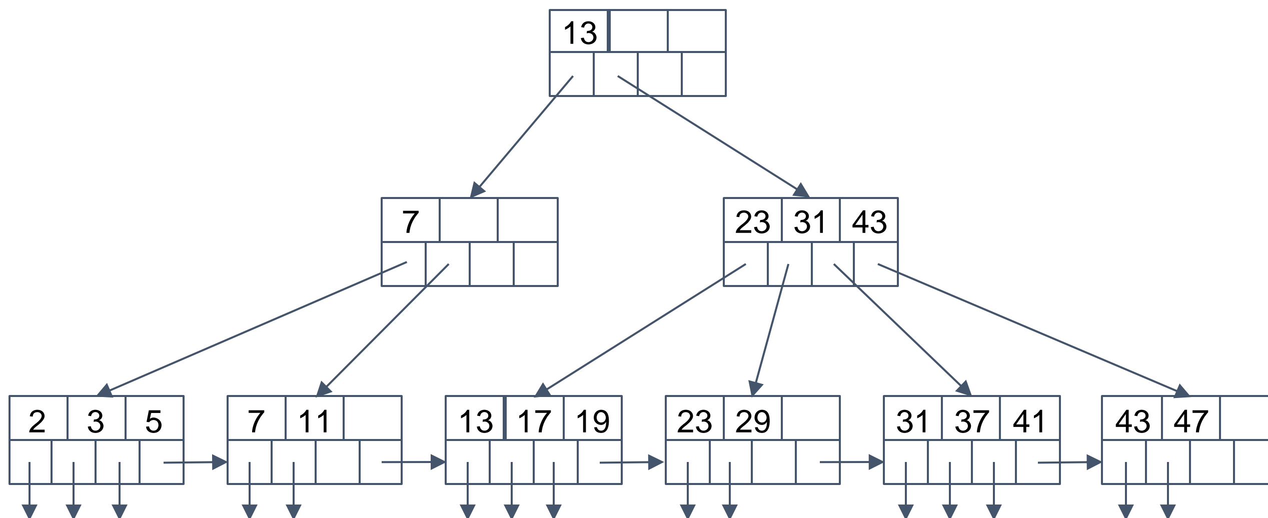


3. B+-Tree

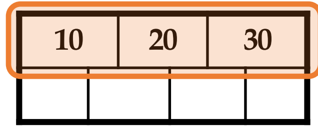
B-tree

More general index structure that is commonly used in commercial DBMS's

- Automatically maintains arbitrary number of levels
- Manages the space on blocks so that each block is at least half full
- We will study the most popular variant called the B+ tree



B+ Tree Basics

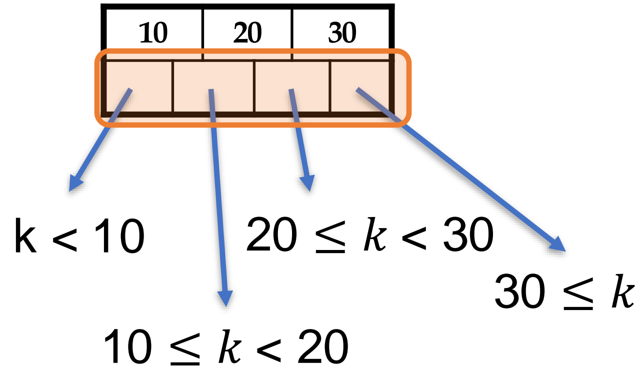


Parameter **d** = the degree

Each non-leaf ("interior") node has ≥ d and ≤ 2d keys*

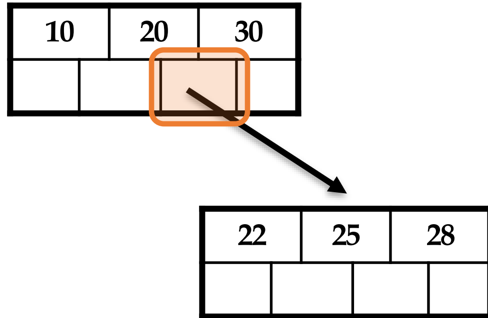
*except for root node, which can have between 1 and 2d keys

B+ Tree Basics



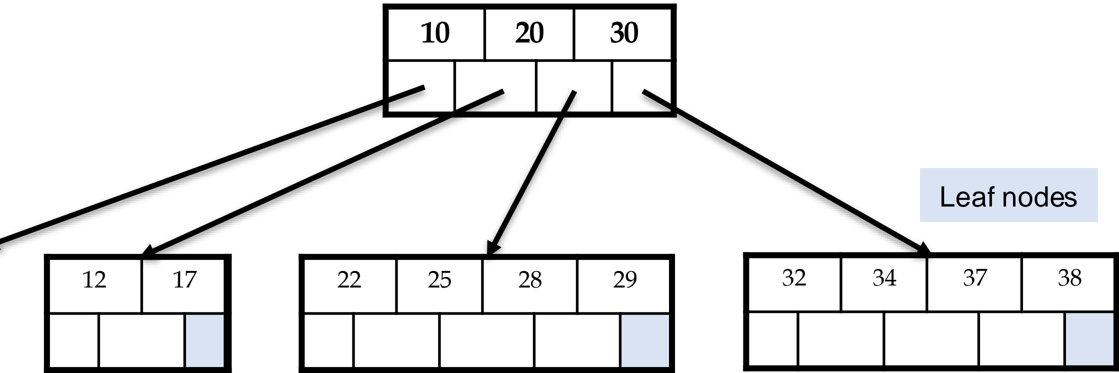
The n keys in a node define n+1 ranges

B+ Tree Basics



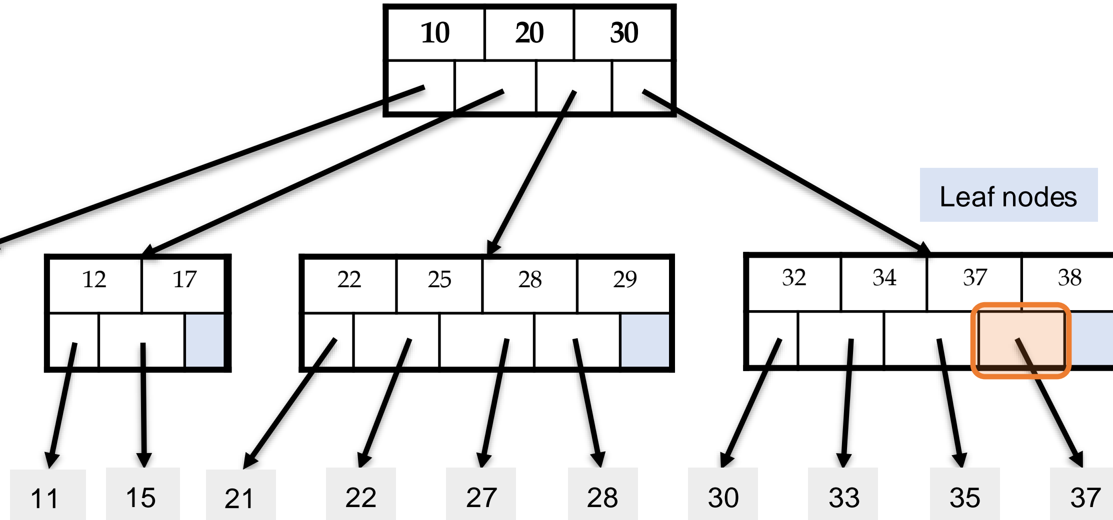
For each range, in a non-leaf node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



Leaf nodes also have between d and 2d keys, and are different in that:

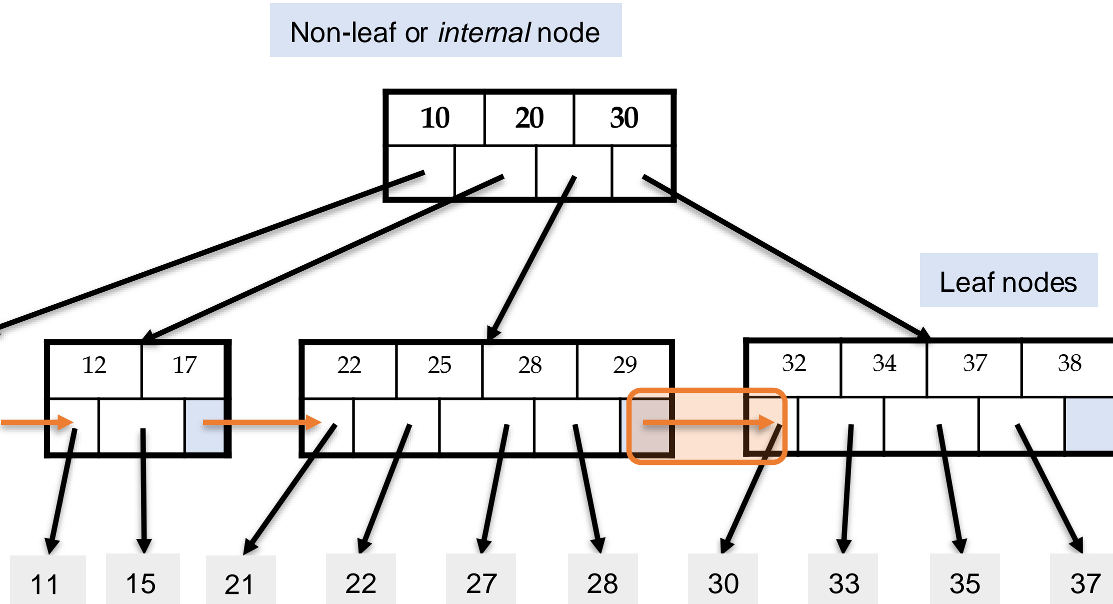
B+ Tree Basics



Leaf nodes also have between d and 2d keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics

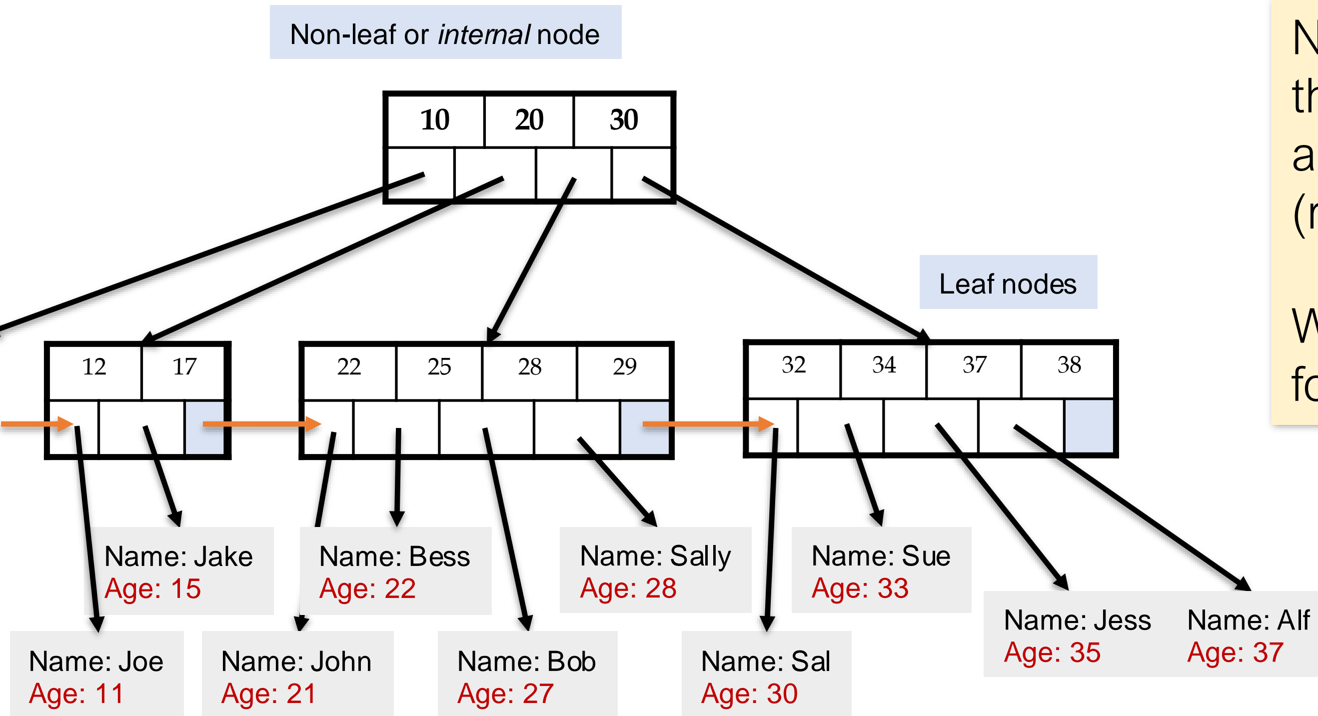


Leaf nodes also have between d and 2d keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, for **faster sequential traversal**

B+ Tree Basics



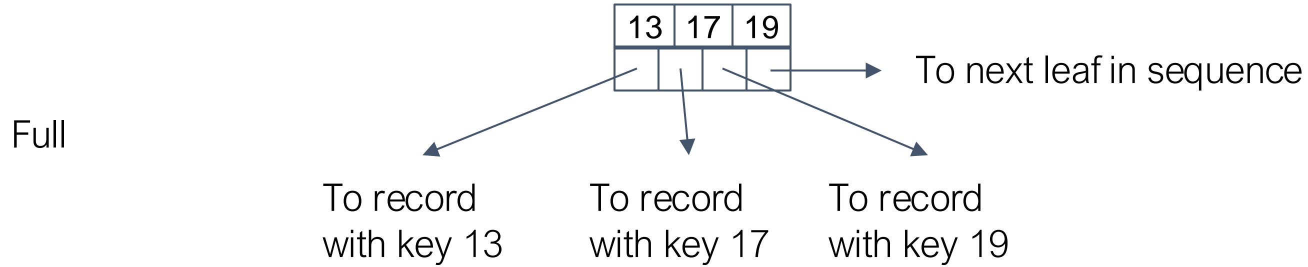
Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display…

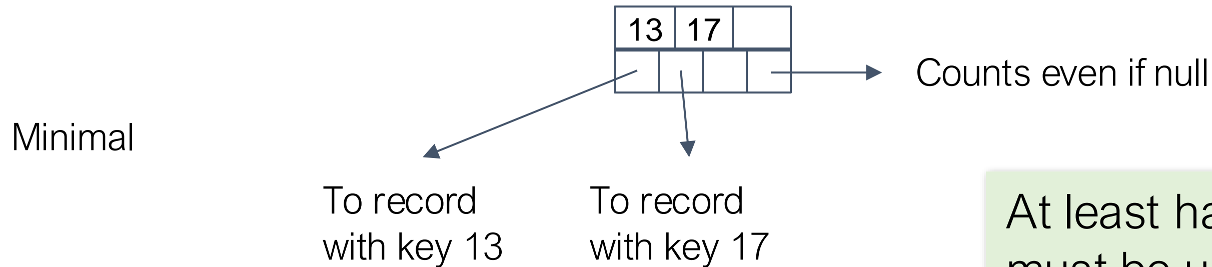
B+ Tree requirement: leaf nodes

n = 3

Full



Minimal

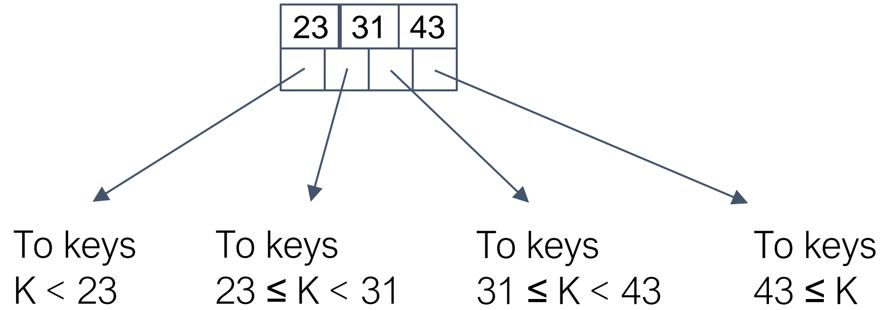


At least half of the keys must be used

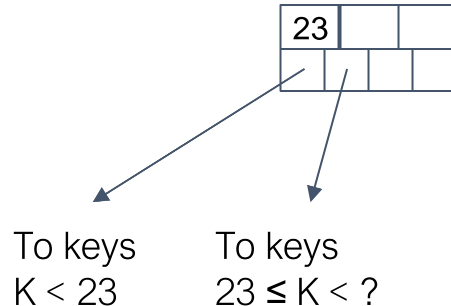
B+ Tree requirement: interior nodes

n = 3

Full



Minimal



At least half of the pointers much be used

Nodes must be “full enough”

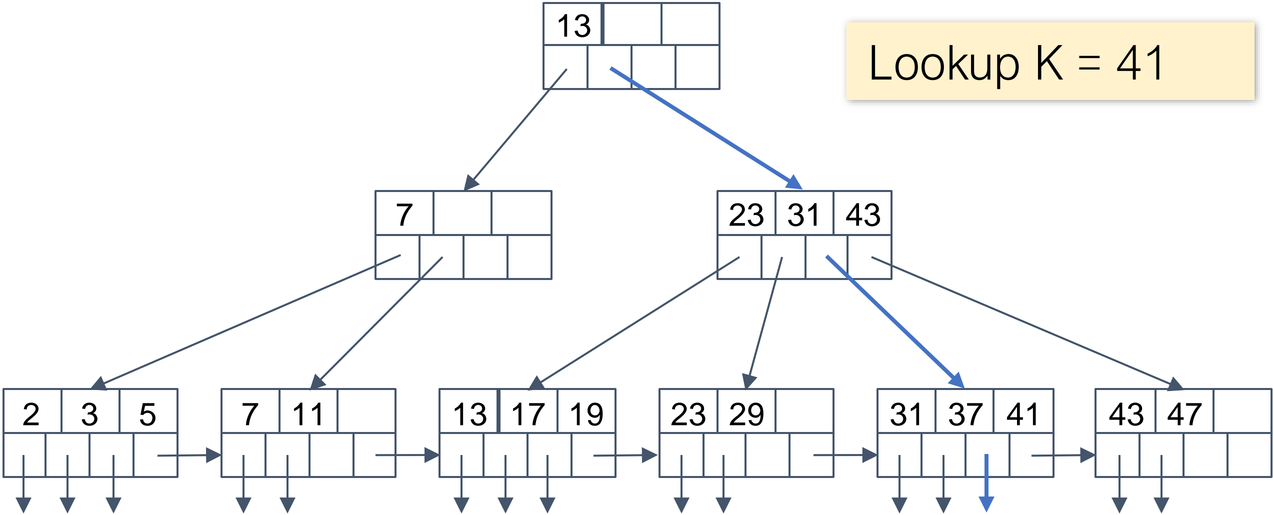
Node type	Min. # pointers	Max. # pointers	Min. # keys	Max. # keys
Interior	$\lceil (n + 1) / 2 \rceil$	n + 1	$\lceil (n + 1) / 2 \rceil - 1$	n
Leaf	$\lfloor (n + 1) / 2 \rfloor$ **	n + 1	$\lfloor (n + 1) / 2 \rfloor$	n
Root	2 *	n + 1	1	n

* Exception: If there is only one record in the B-tree, there is one pointer in the root

** Not including the next leaf pointer

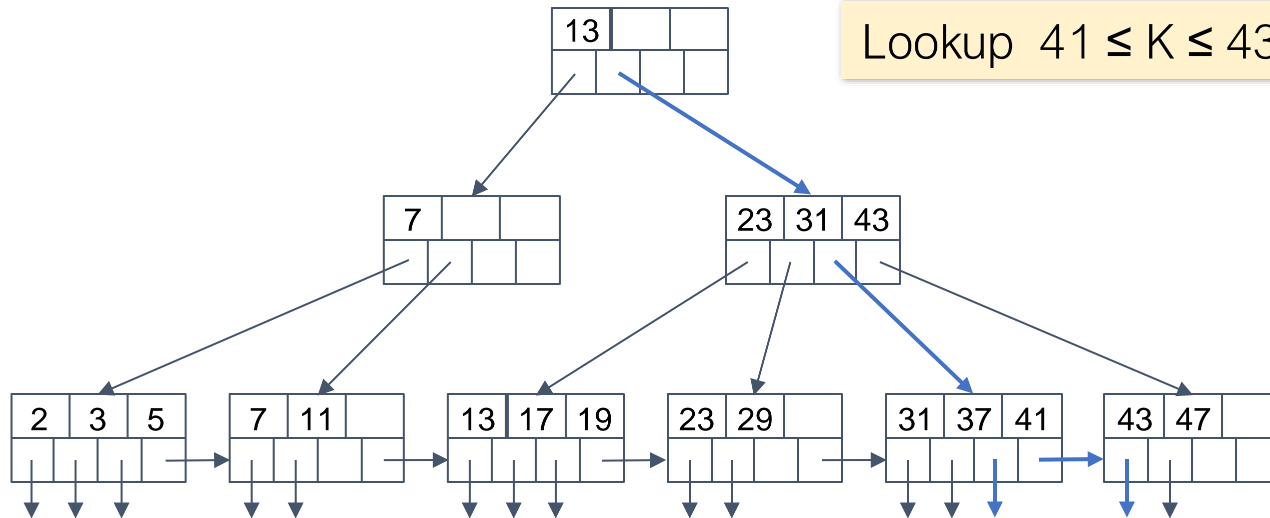
Lookup

- Search for key K recursively



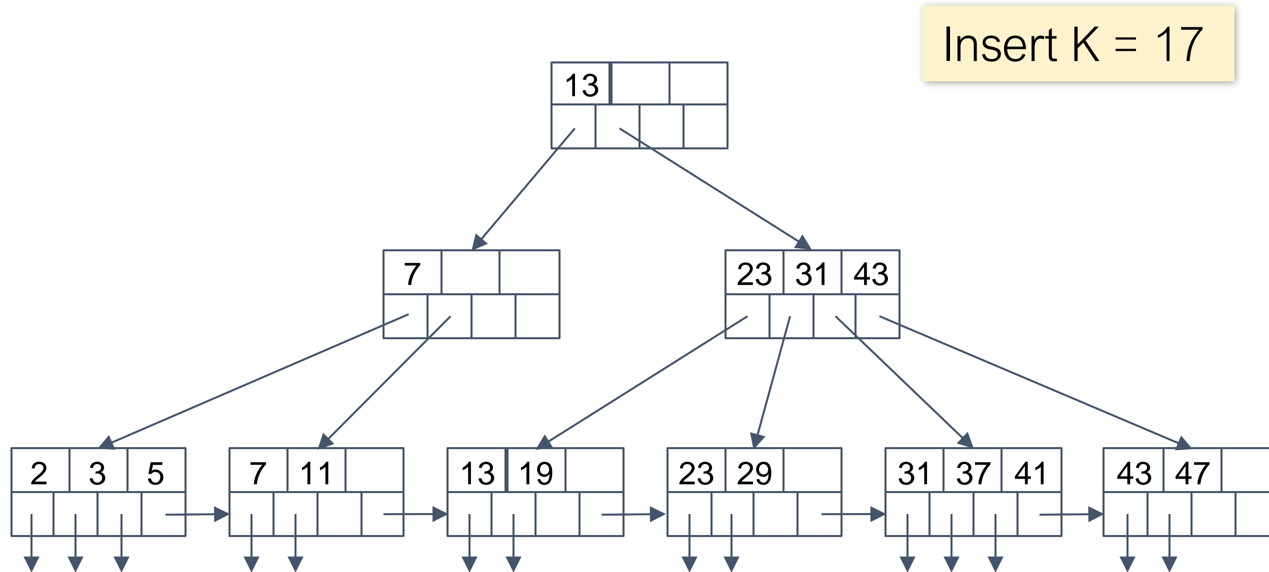
Lookup

- For range query [a, b], search for key a then scan leaves to right until we pass b



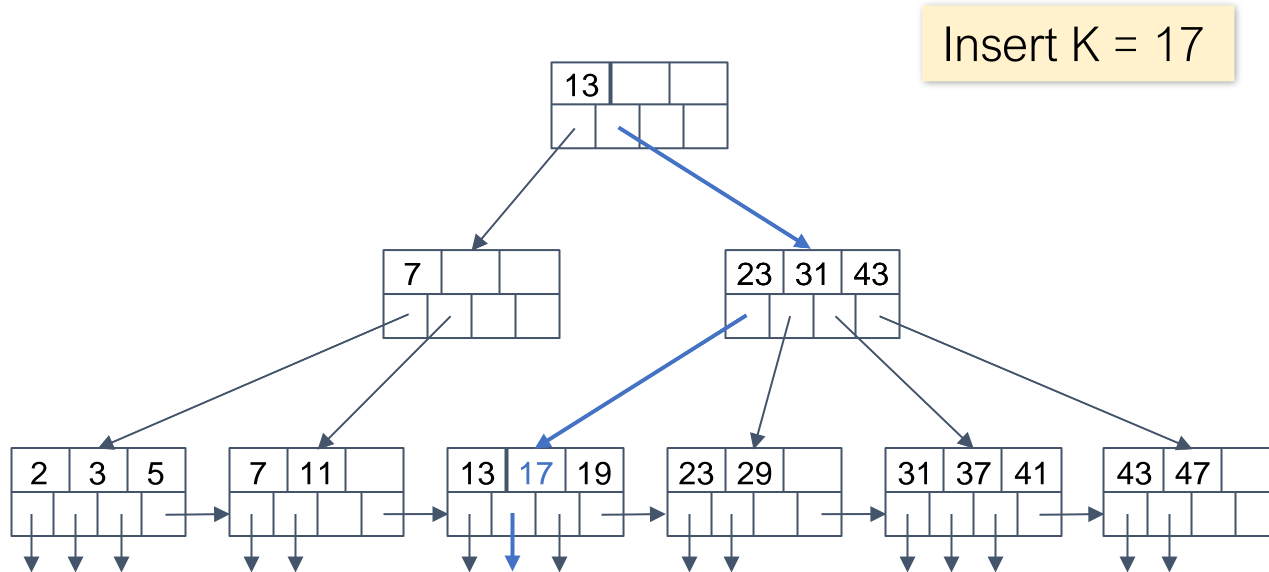
Insertion

- Find place for new key in a leaf
- If there is space, put key in leaf



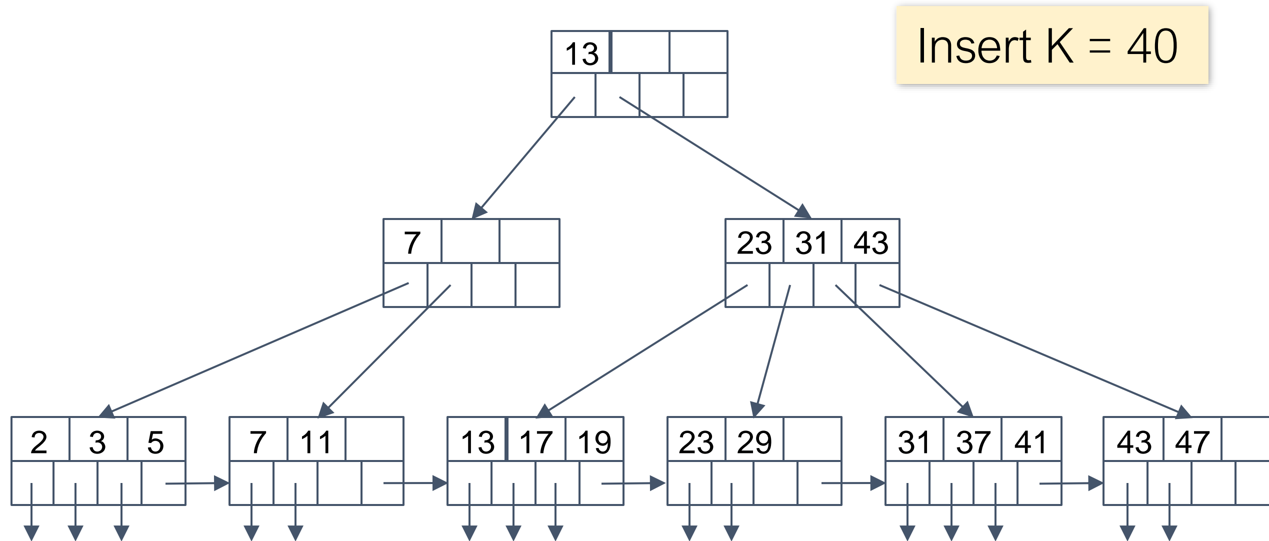
Insertion

- Find place for new key in a leaf
- If there is space, put key in leaf



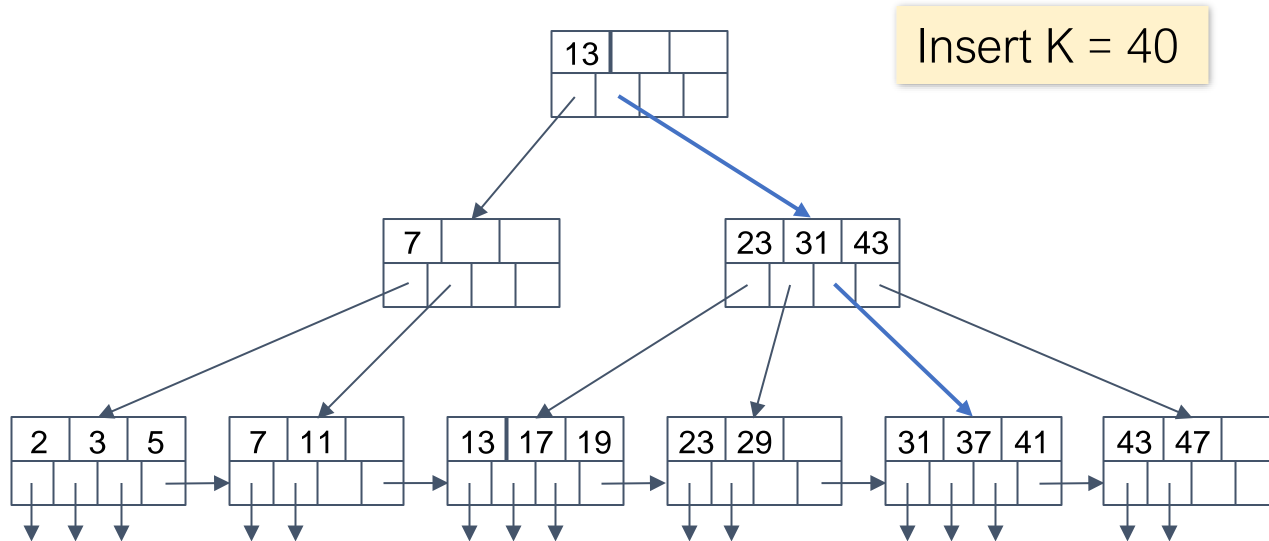
Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



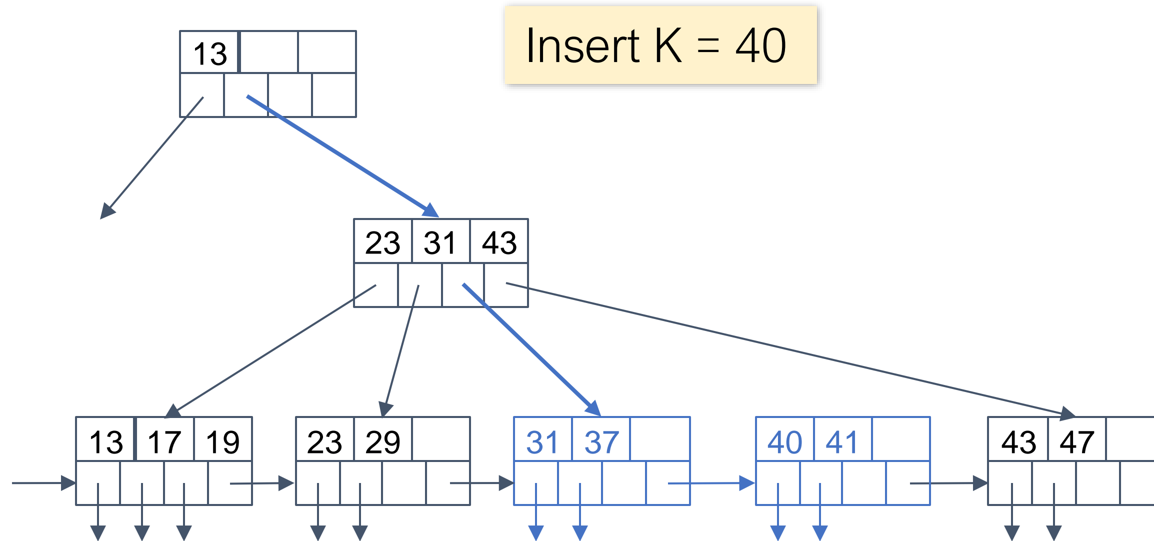
Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



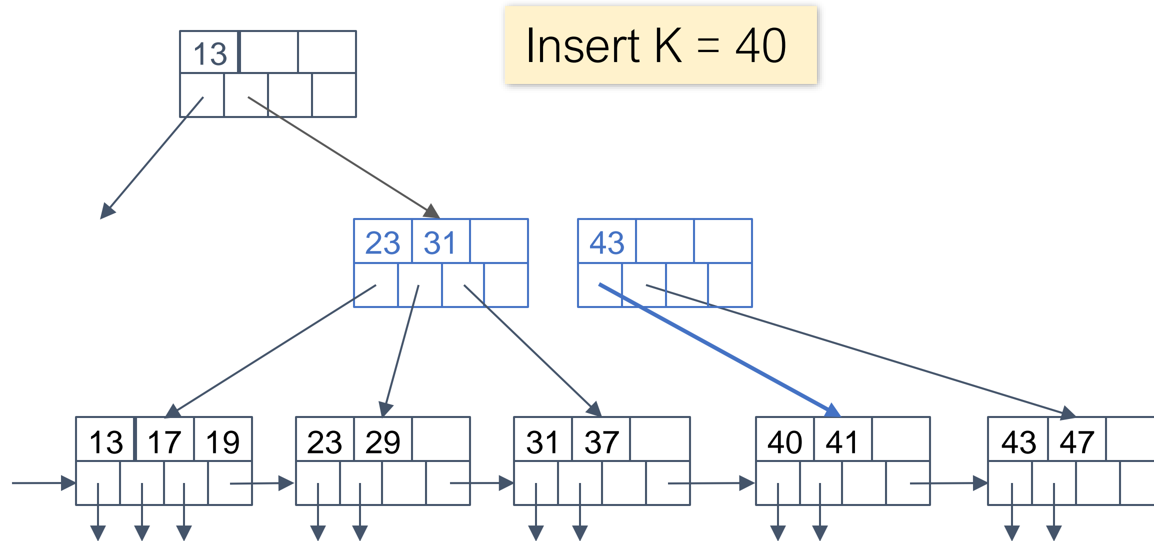
Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



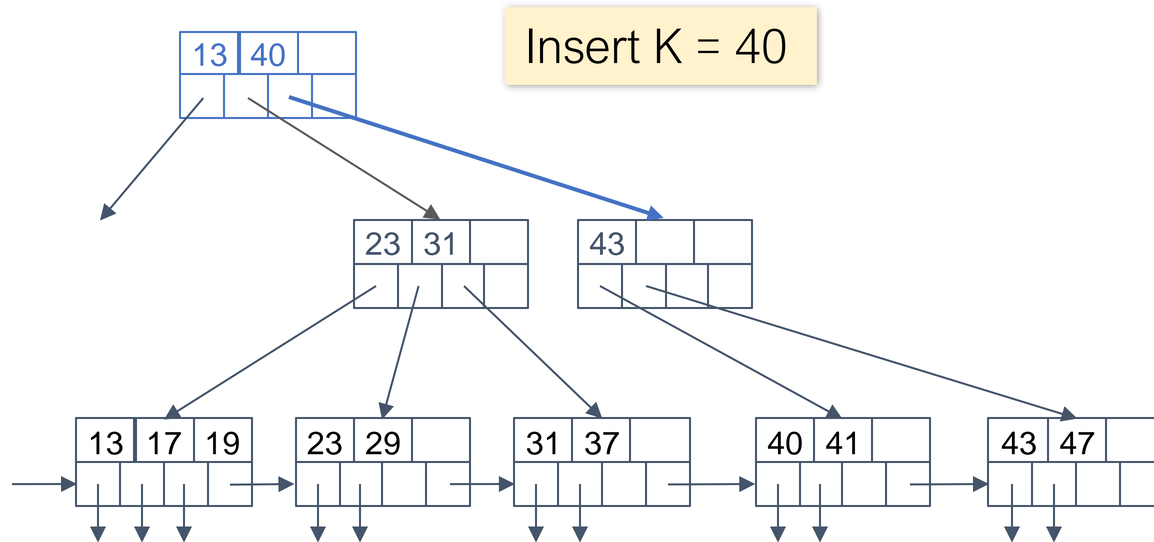
Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



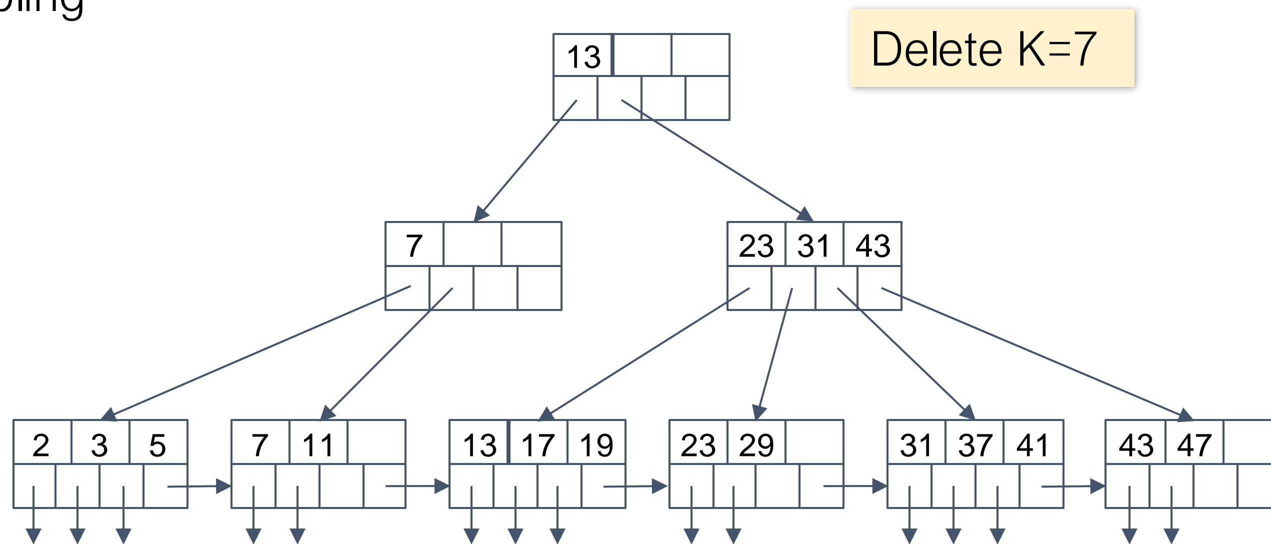
Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



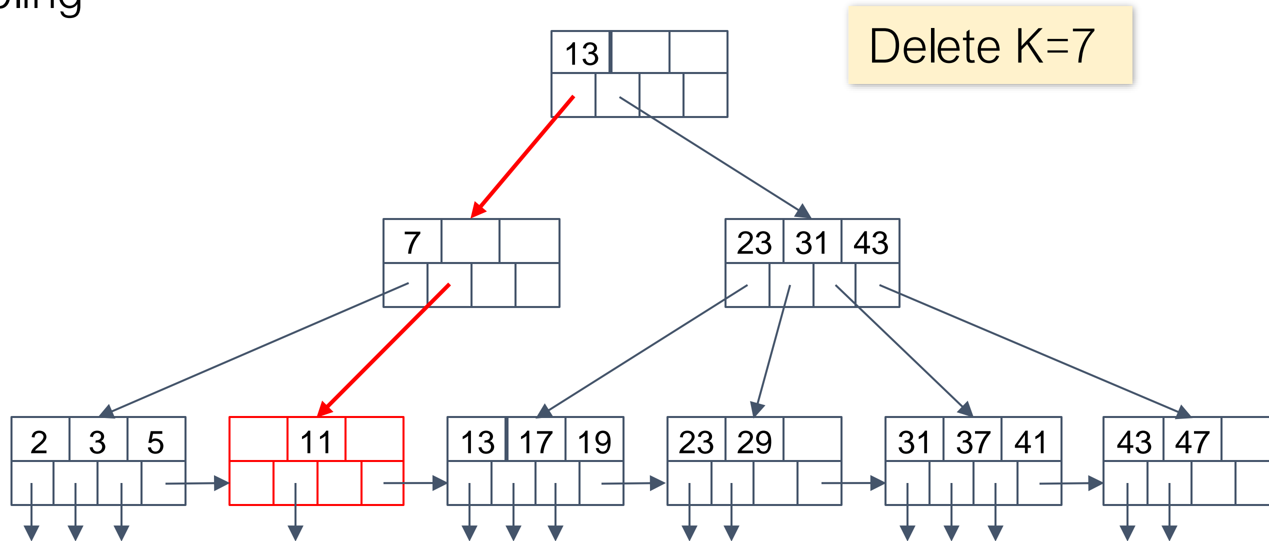
Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



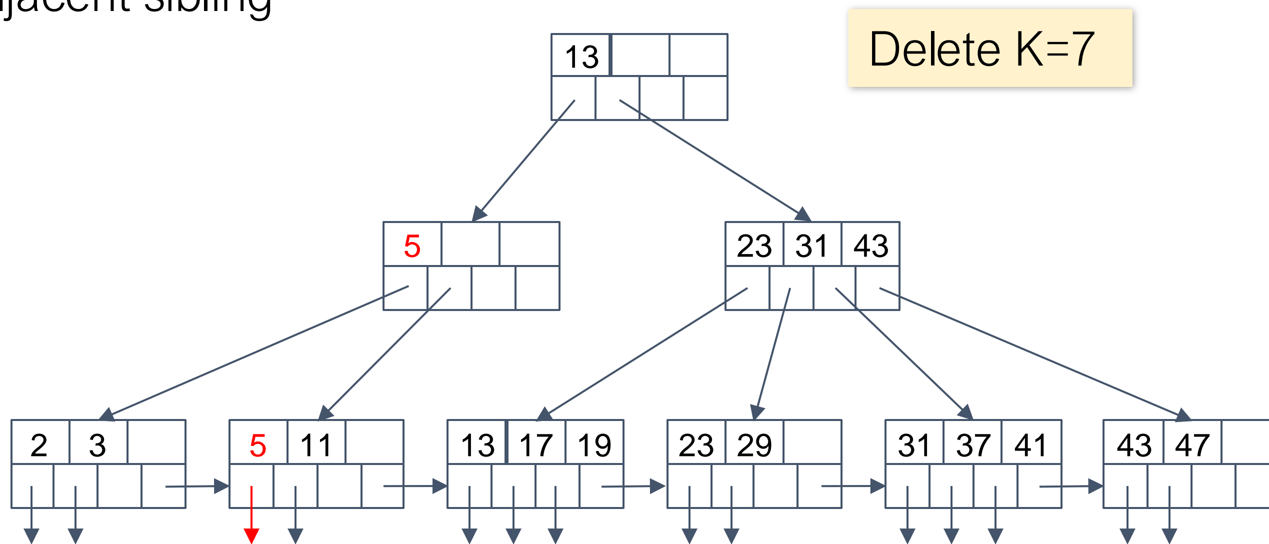
Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



Deletion

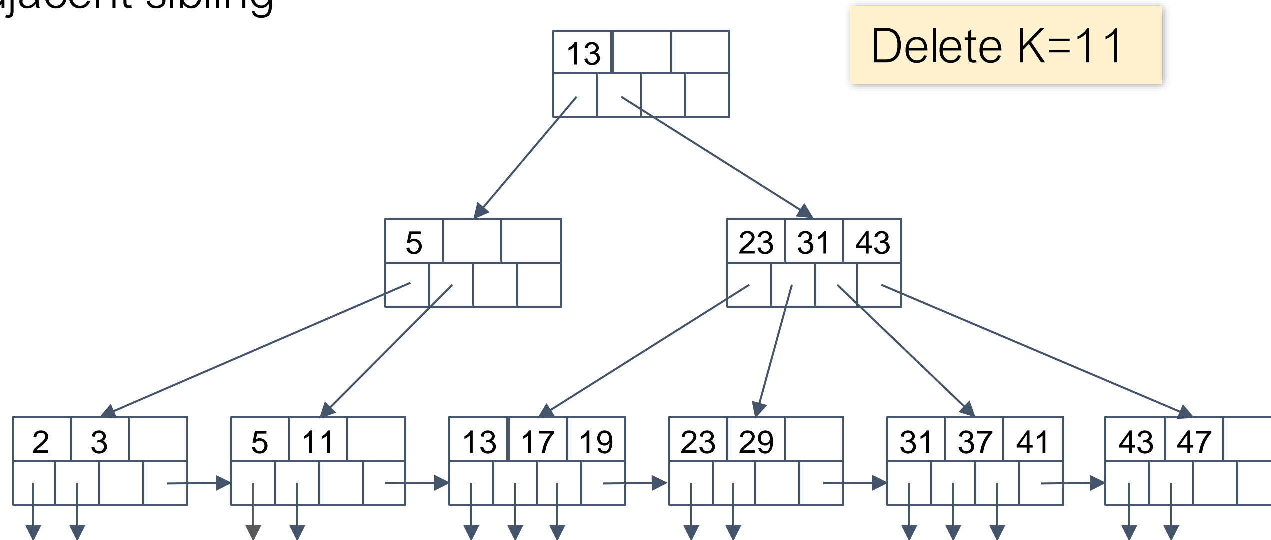
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



Deletion

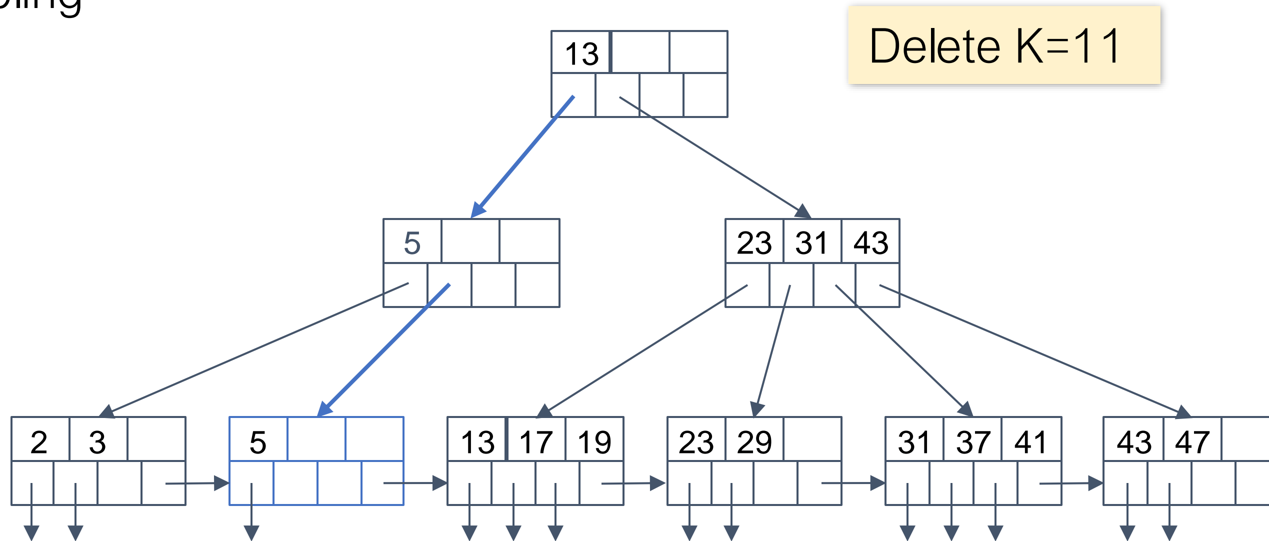
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling

Delete K=11



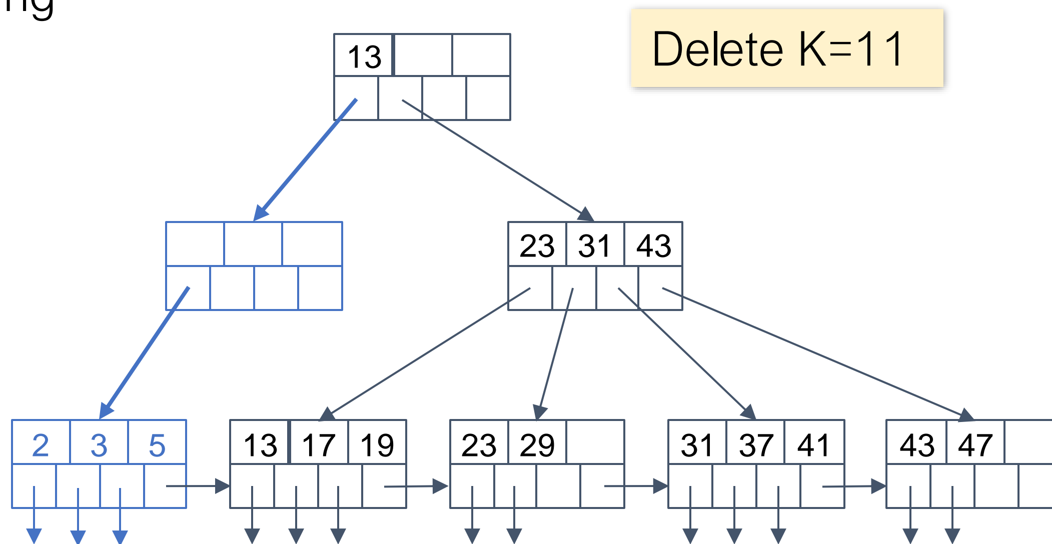
Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



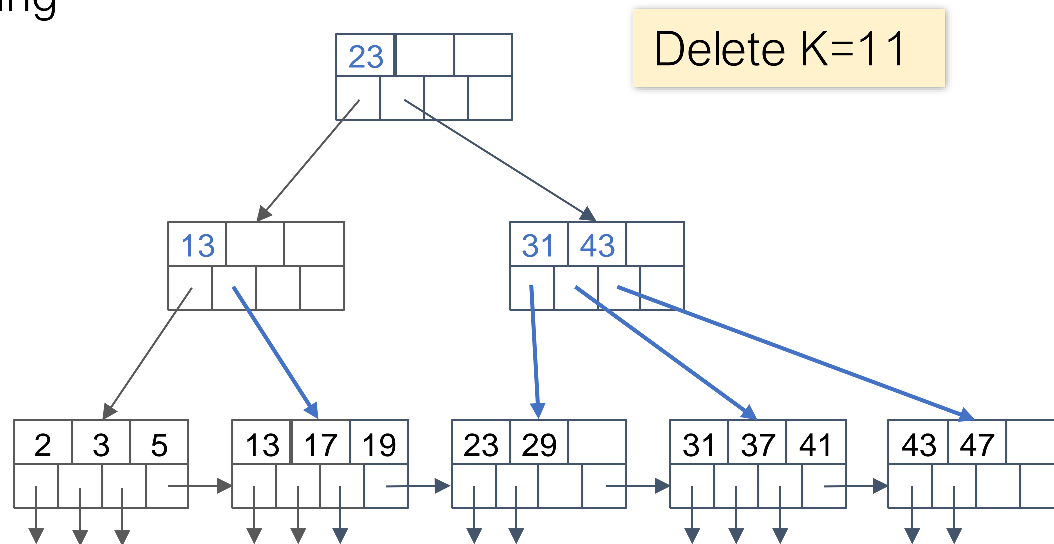
Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



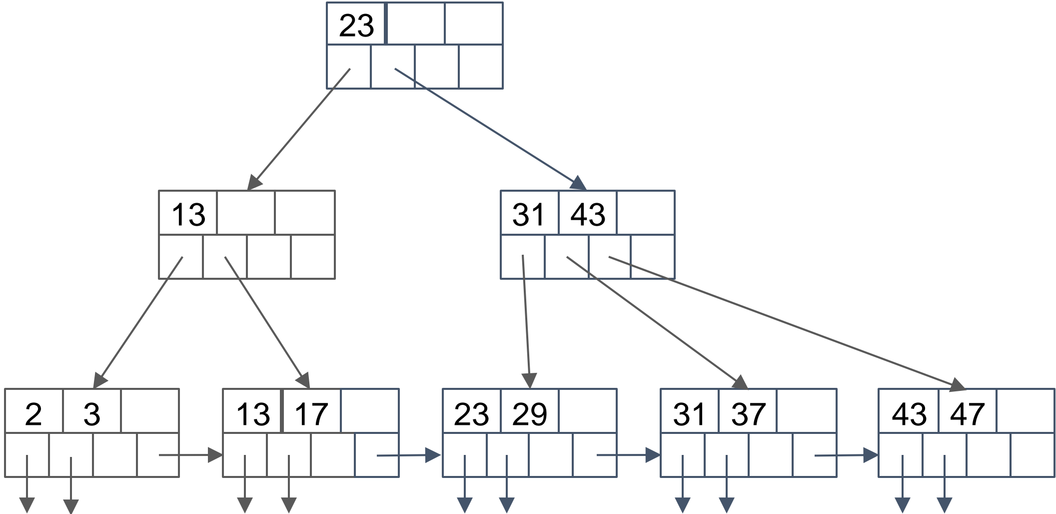
Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



Exercise #2

- Delete K = 31



B-tree deletions in practice

- Coalescing is sometimes not implemented because
  - It is hard to implement and
  - The B-tree will probably grow again