CS 4440 A

Emerging Database Technologies

Lecture 6
01/29/24

Announcements

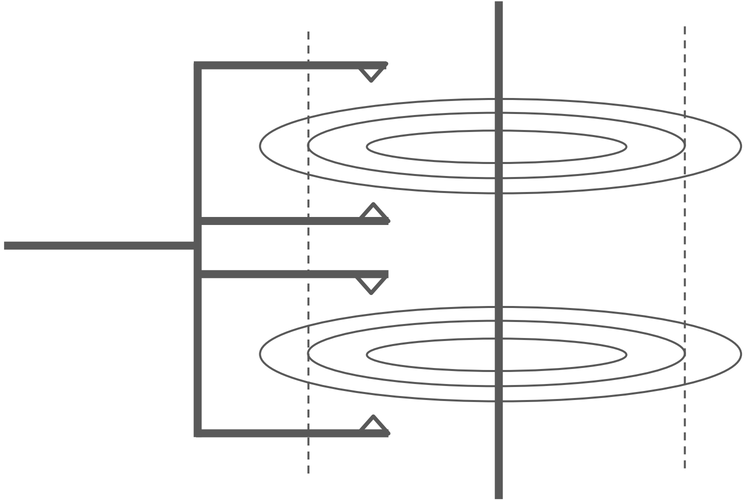
- Assignment 1 due today @ 11:59PM
  - Technology presentation group will be announced by next Monday

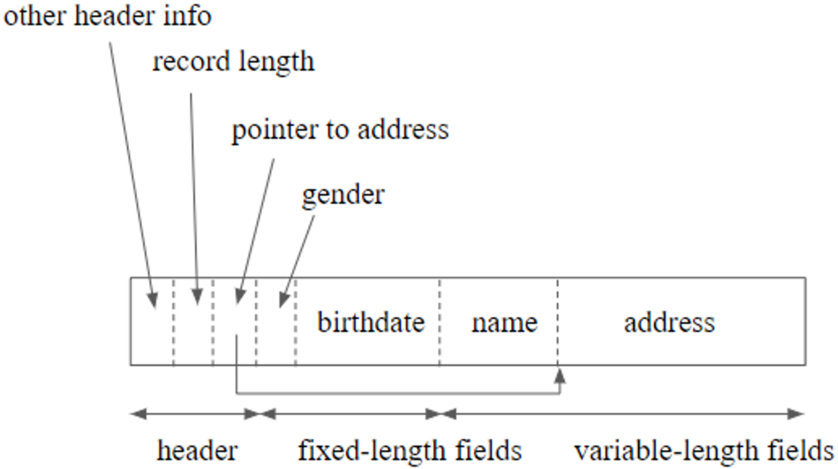
- Please sign up on Canvas if you have finalized your project group

Recap

- Hardware
  - Storage hierarchy
  - Secondary storage
  - Disk access time
  - Speeding up disk access

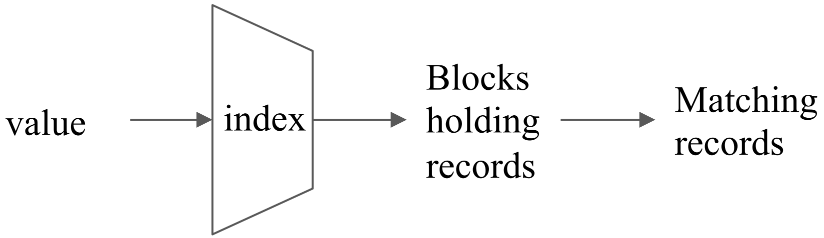
- File System Structure
  - Fixed-length records
  - Variable-length records





Index

- A data structure that takes field values and quickly finds records containing them
- Can find tuples of a relation without scanning the entire database



Using Indexes in SQL

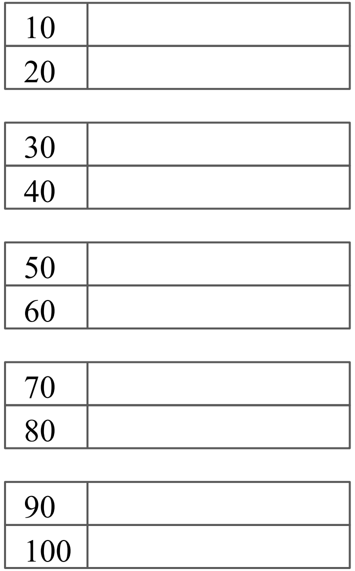
- An index is used to efficiently find tuples with certain values of attributes
- An index may speed up lookups and joins
- However, every built index makes insertions, deletions, and updates to relation more complex and time-consuming

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

```
DROP INDEX KeyIndex;
```

Sequential file

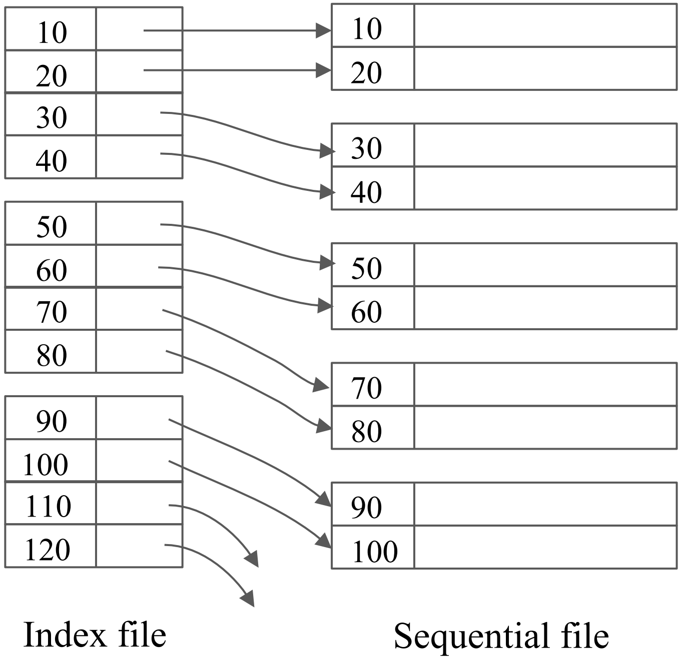
- A file containing tuples of a relation sorted by their primary key



Sequential file

Dense index

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

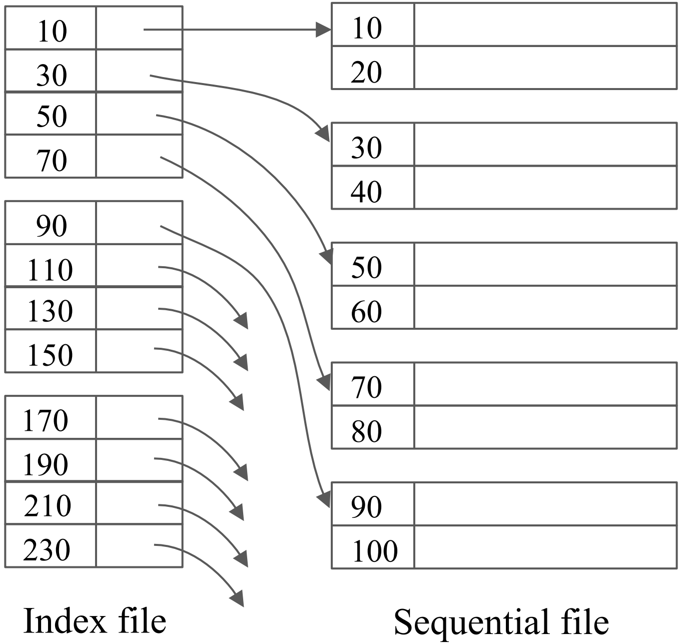


Dense index

- 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

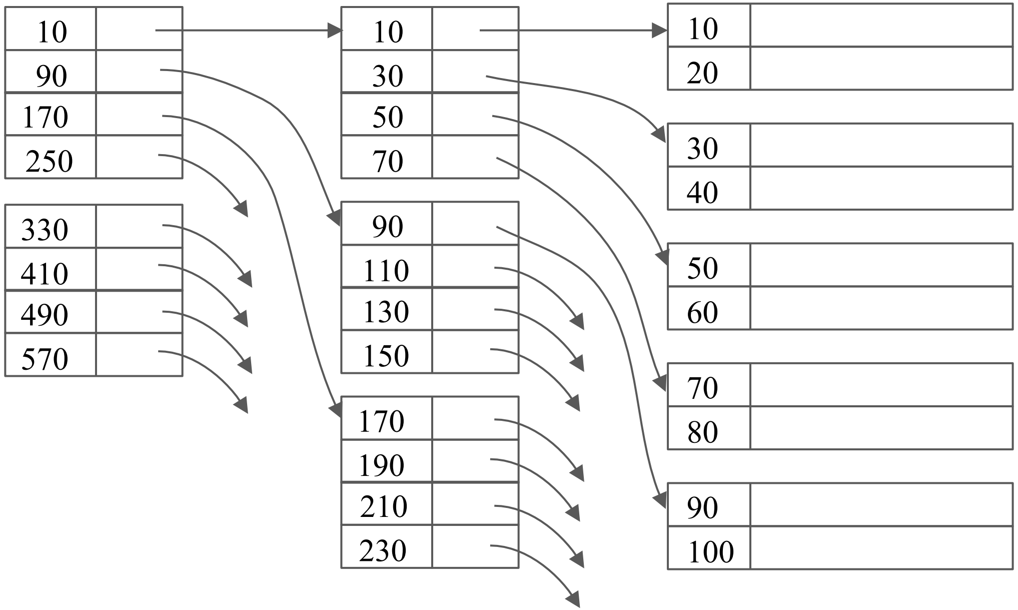


Exercise #1

- 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
- Later: B-tree structure does this better



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

Secondary index

- Unlike a primary index, does not determine the placement of records

20	
40	

10	
20	

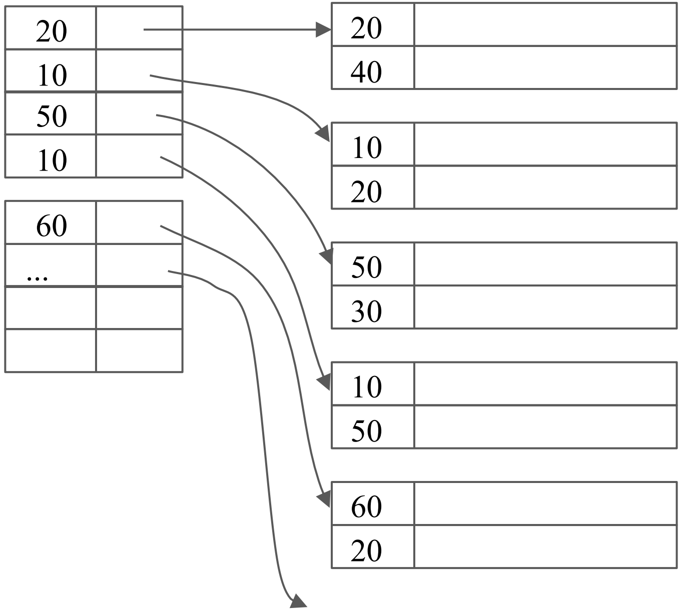
50	
30	

10	
50	

60	
20	

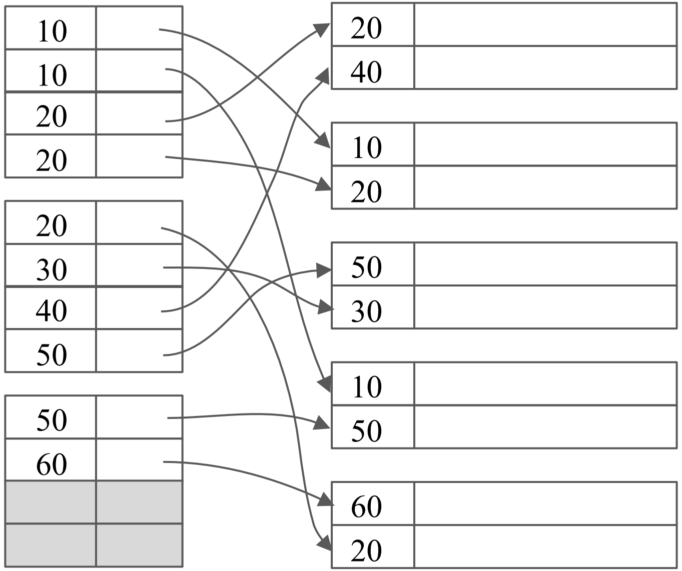
Secondary index

- Using a sparse index doesn't make sense



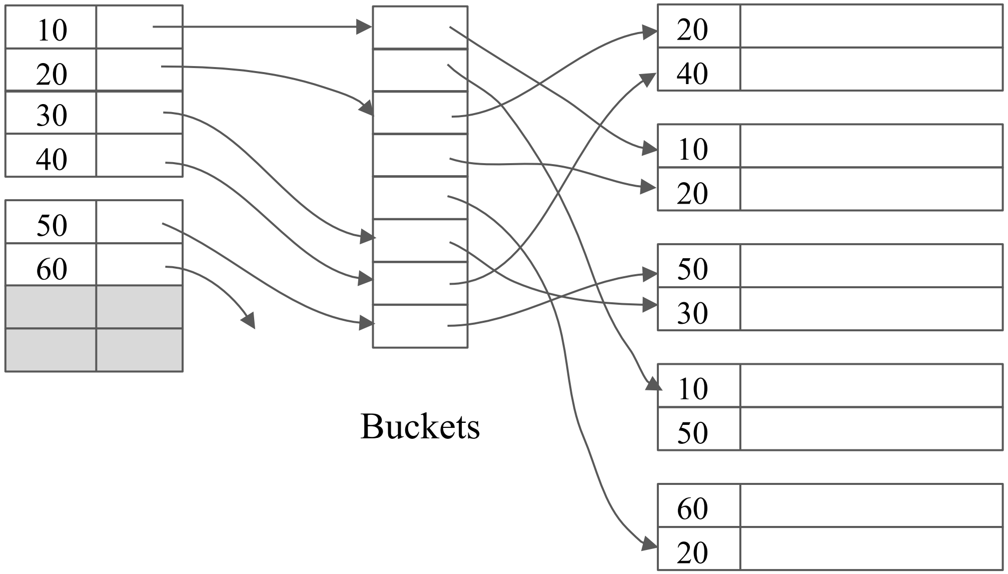
Secondary index

- As a result, secondary indexes are always dense



Secondary index

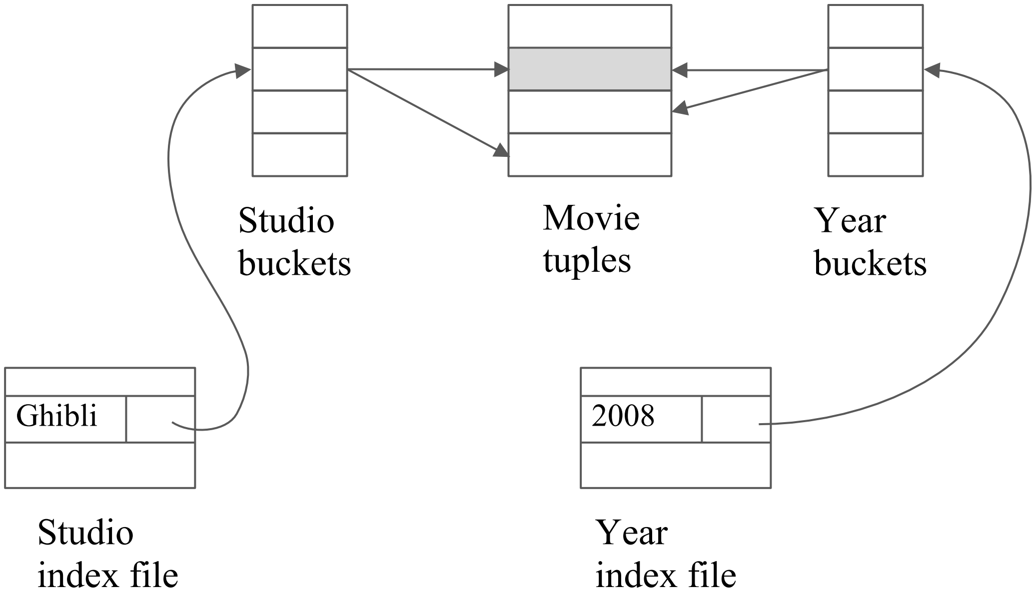
- To remove redundant keys in secondary index file, use level of indirection



When is indirection and secondary index useful?

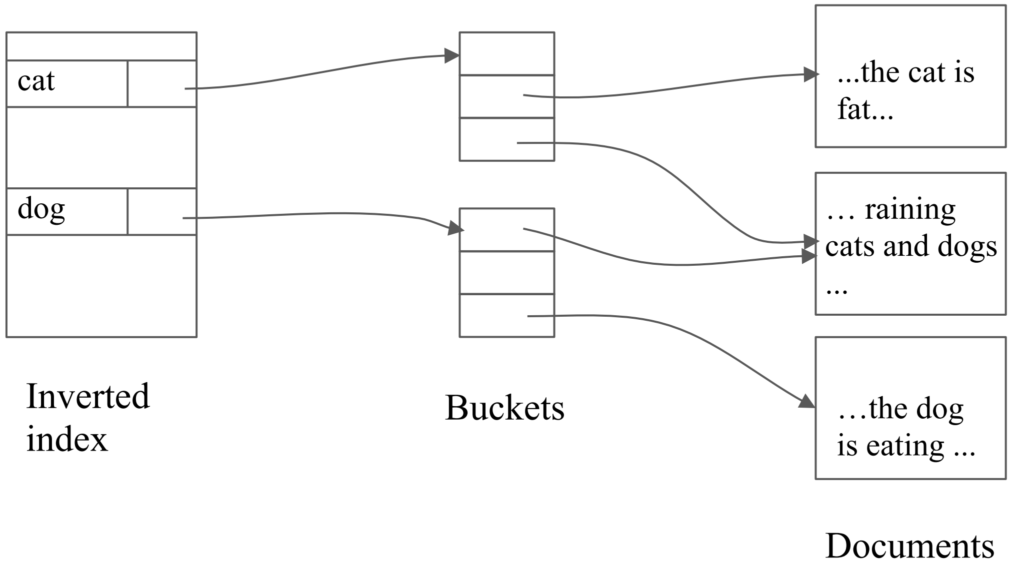
- When a key is larger than a pointer and each key appears twice on average
- Another advantage: use bucket pointers without looking at most of the records

```
SELECT title
FROM Movies
WHERE studioName = 'Ghibli'
AND year = 2008;
```



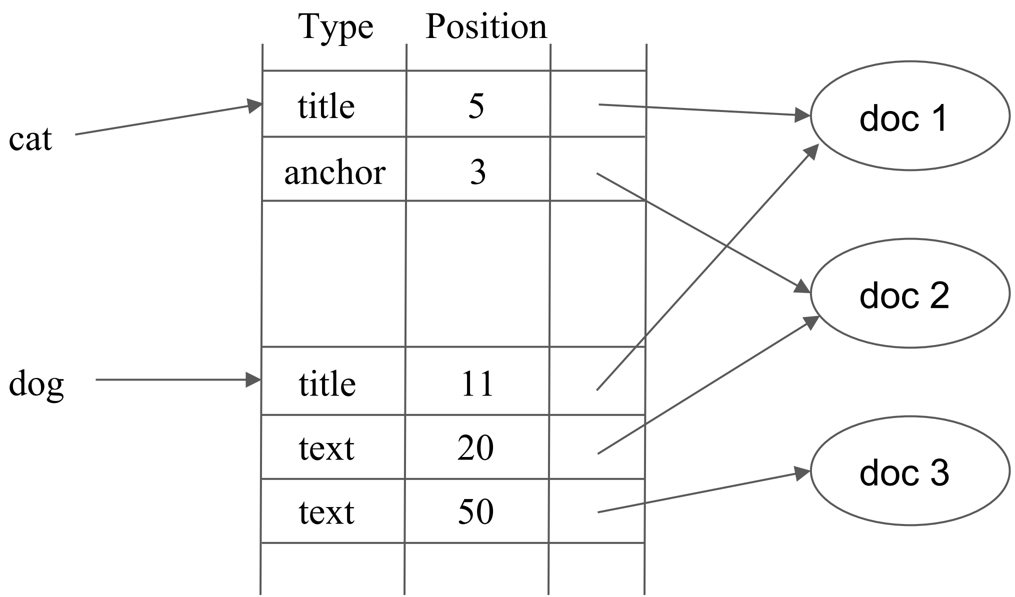
Inverted index

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



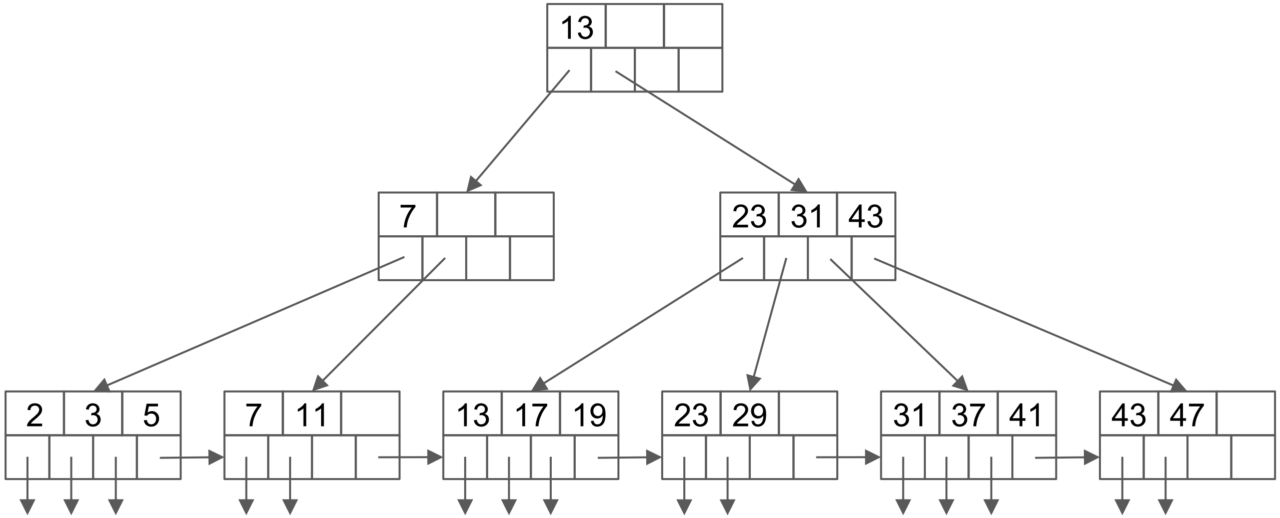
Store more information in inverted index

- 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



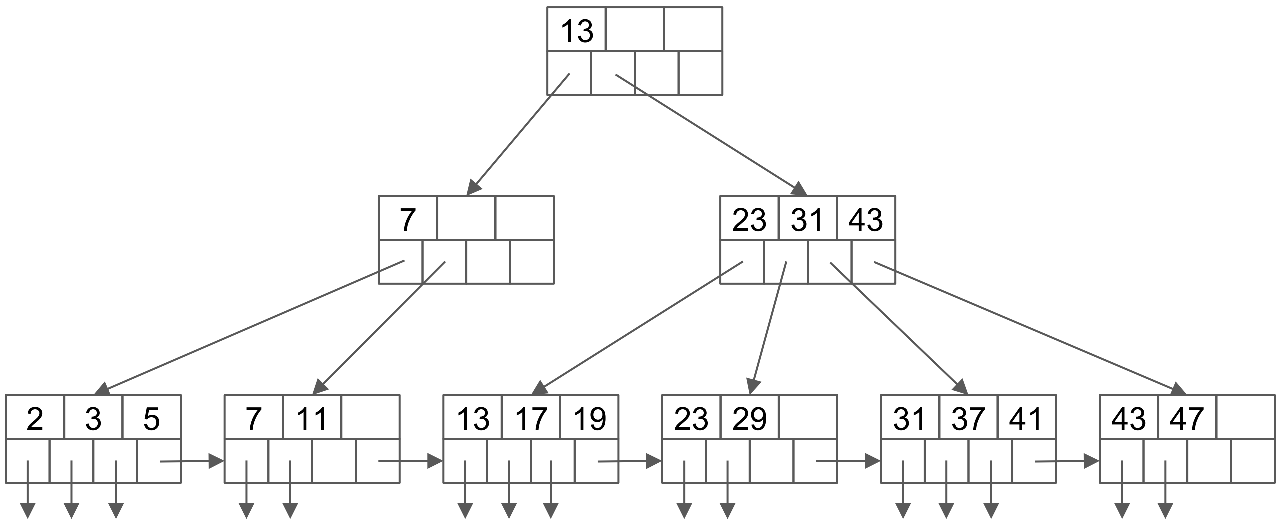
B-tree

- 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

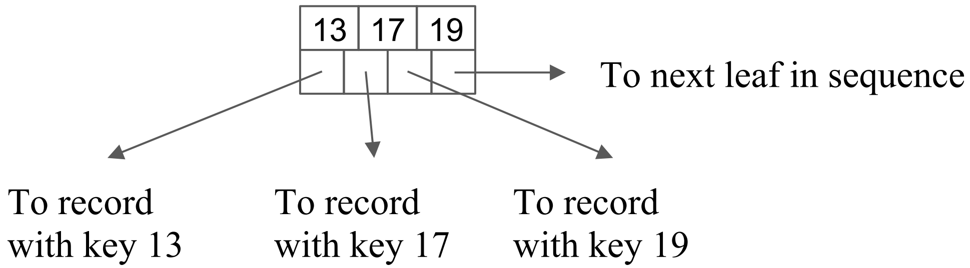
- Parameter: $n = 3$ (n search keys and $n + 1$ pointers per block)
- The keys in leaf nodes are record keys sorted from left to right
- Assume all keys are distinct for now



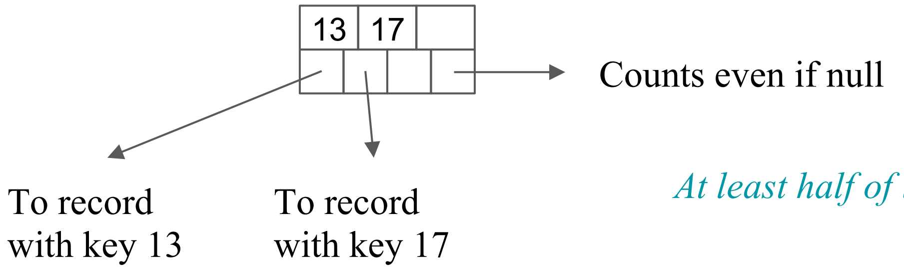
Typical leaf

- $n = 3$

Full



Minimal

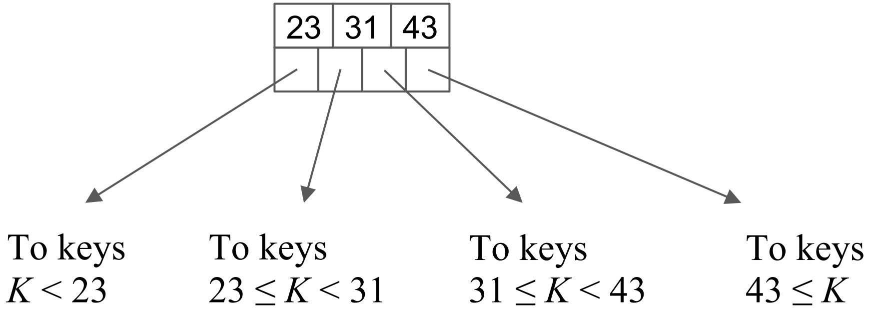


At least half of the keys must be used

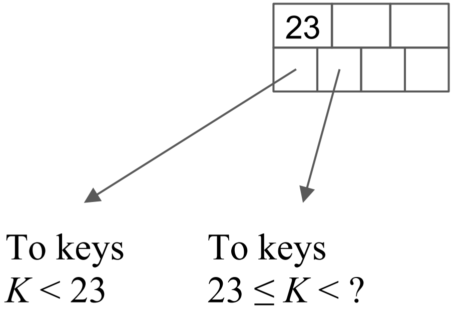
Typical interior node

- $n = 3$

Full



Minimal



At least half of the pointers must be used

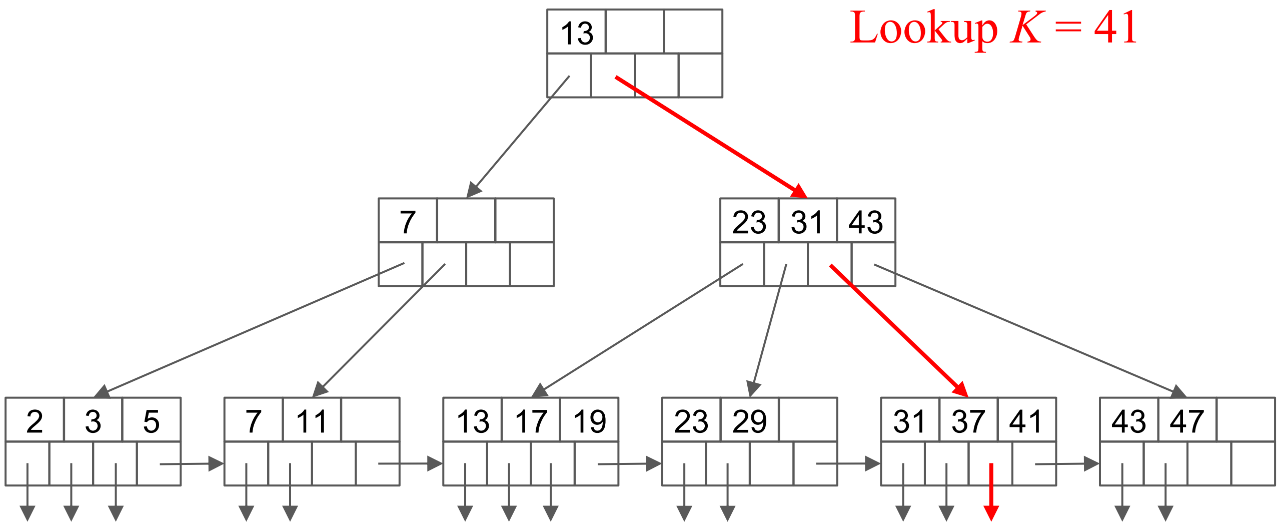
Nodes must be “full enough”

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

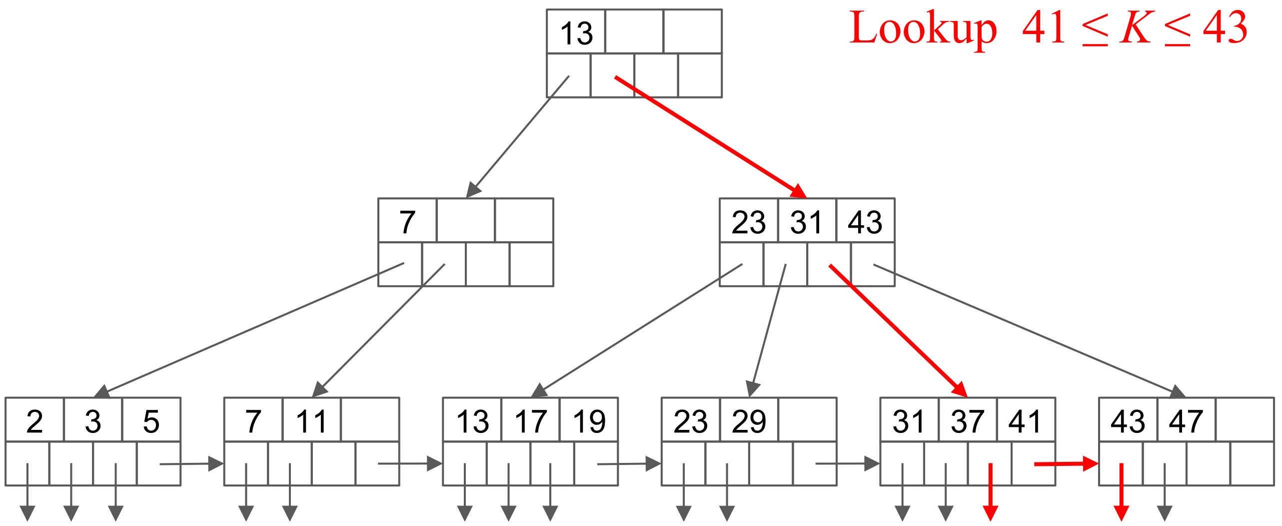
Lookup

- Search for key K recursively



Lookup

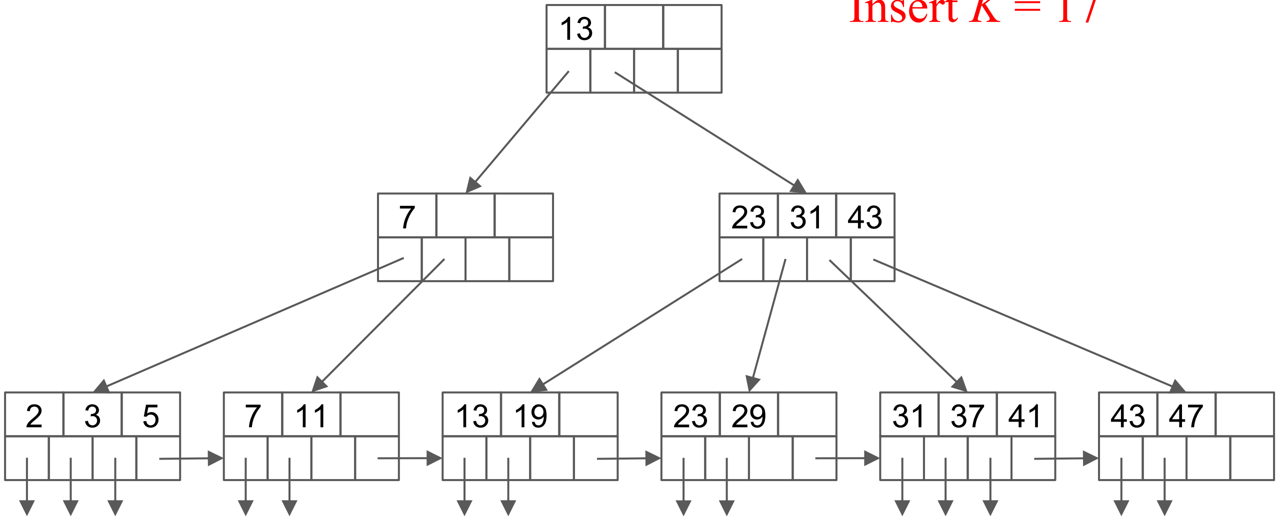
- For range query [a, b], search for key a then scan leaves to right until we pass b



Insertion

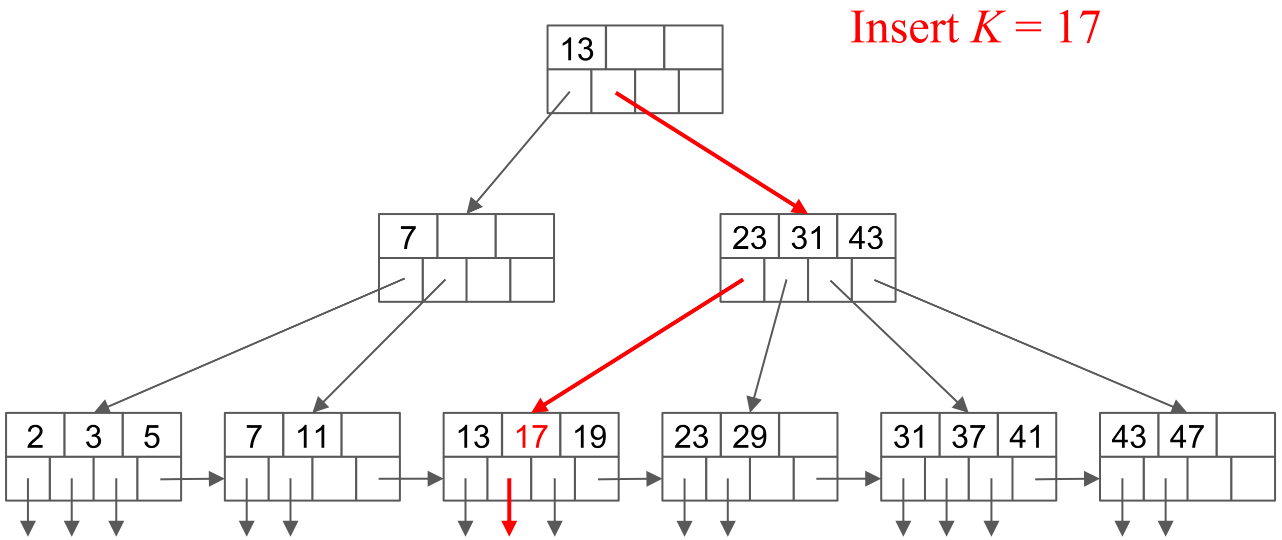
- Find place for new key in a leaf
- If there is space, put key in leaf

Insert $K = 17$



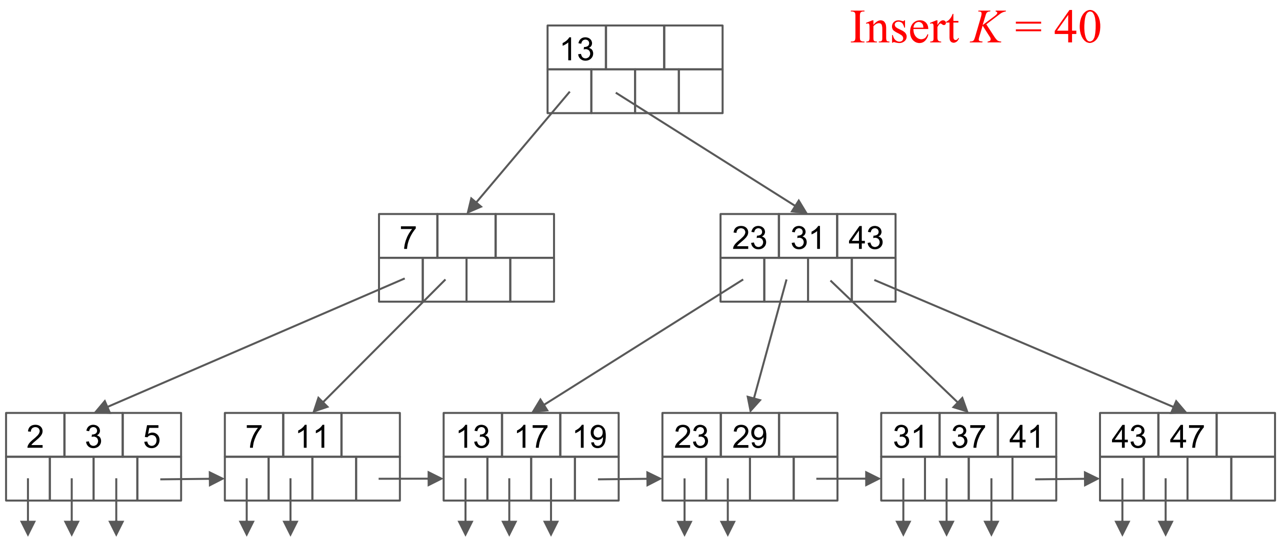
Insertion

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



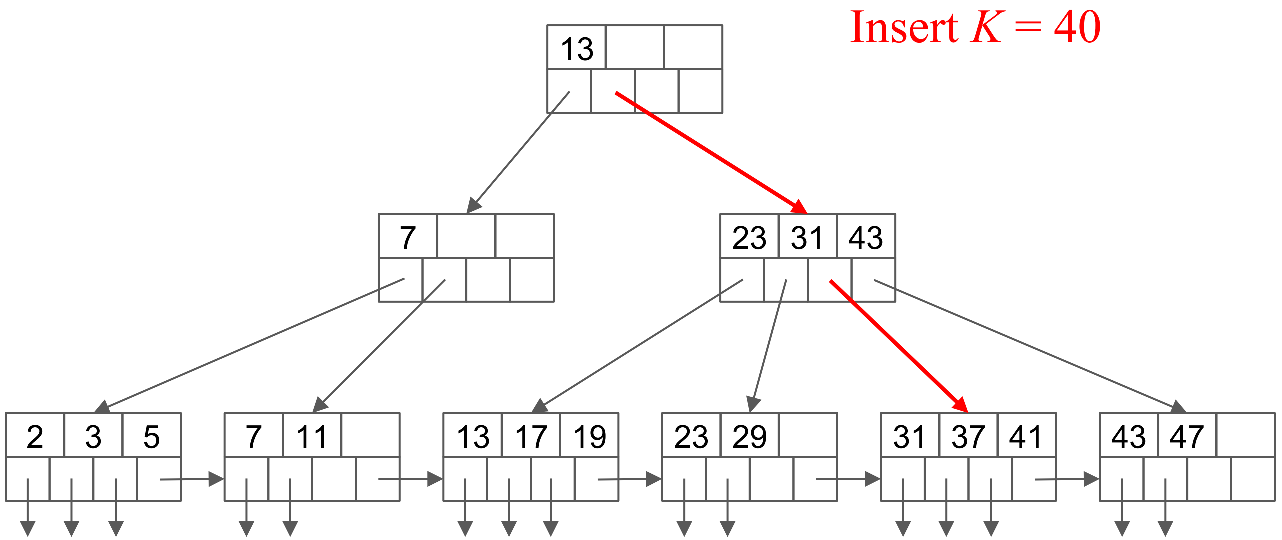
Insertion

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



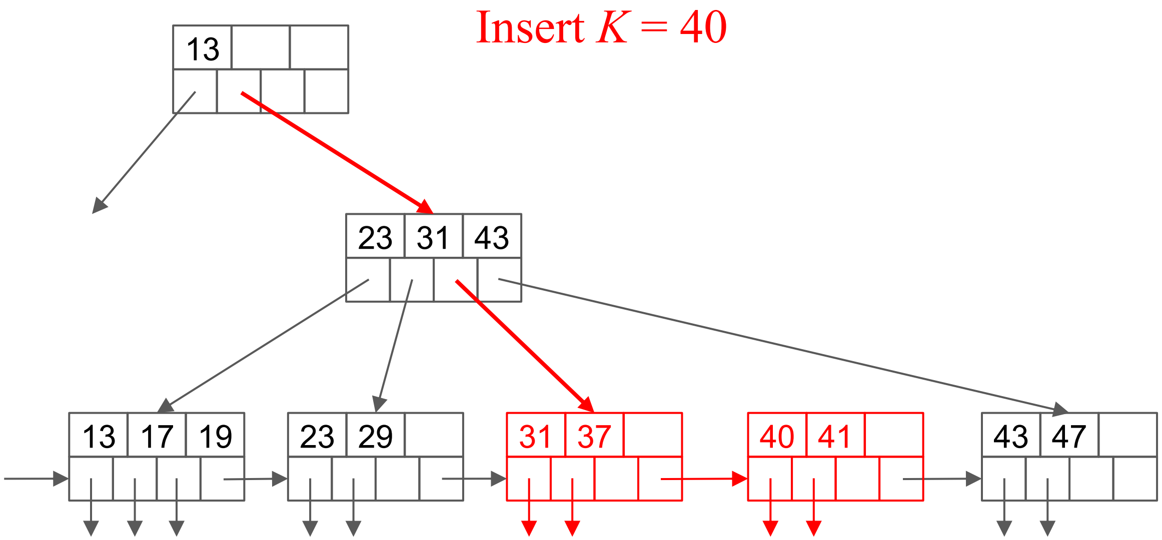
Insertion

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



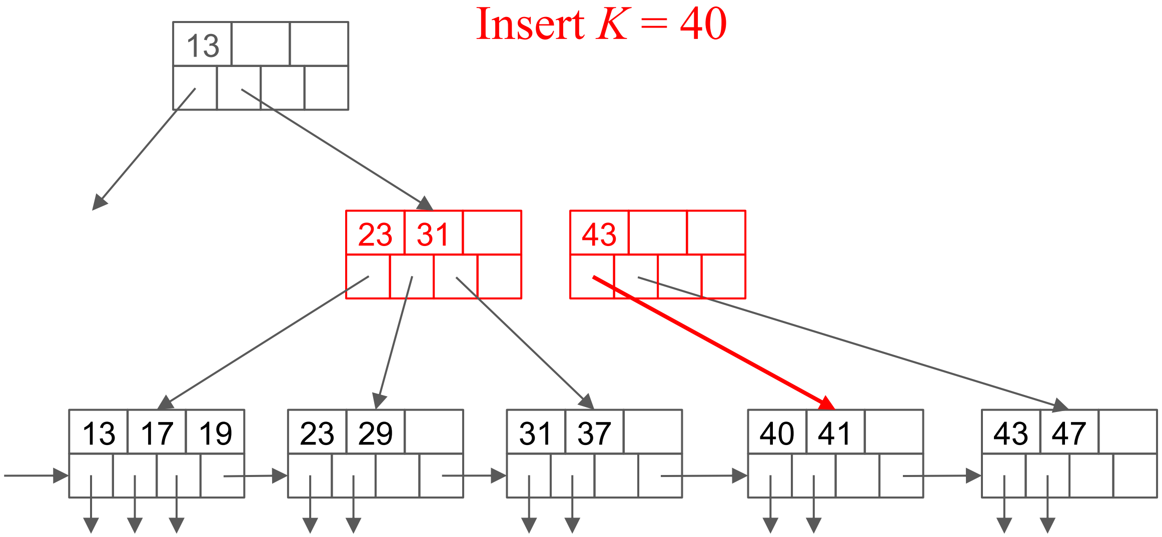
Insertion

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



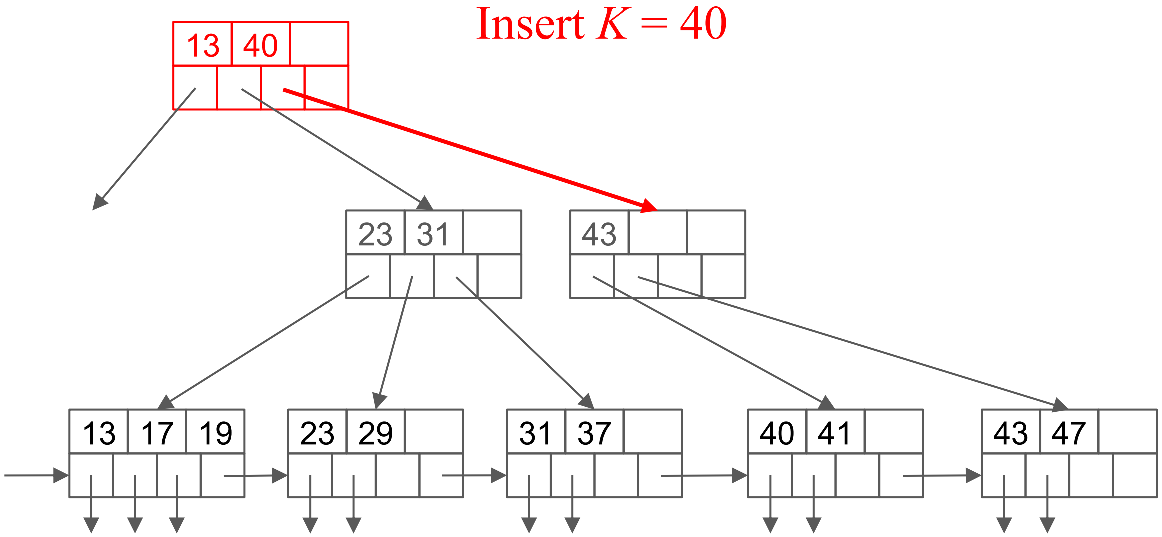
Insertion

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



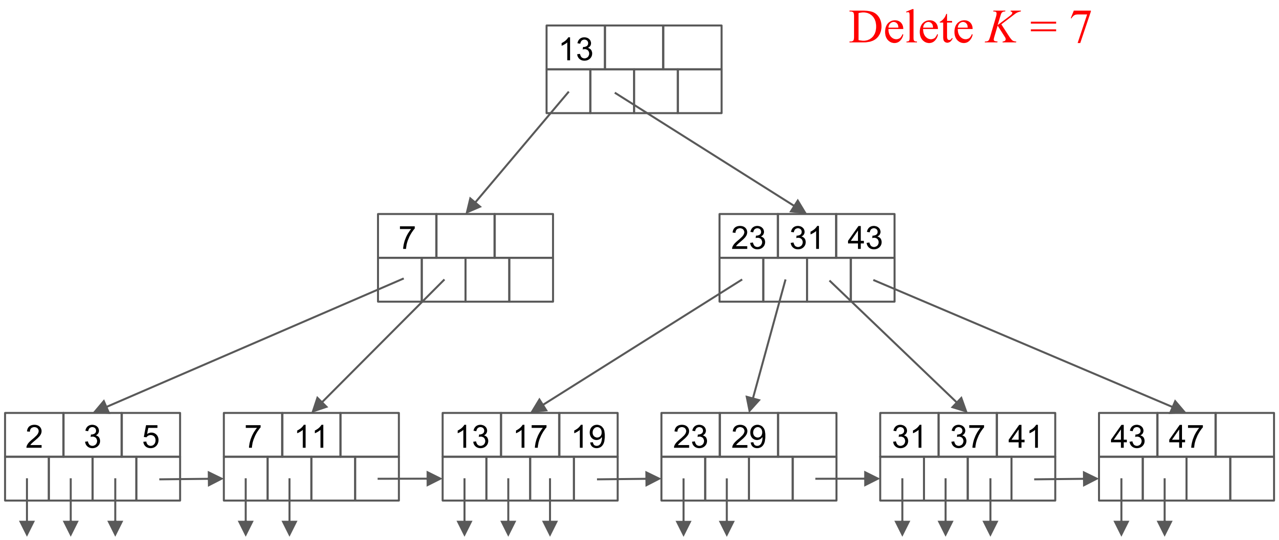
Insertion

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



Deletion

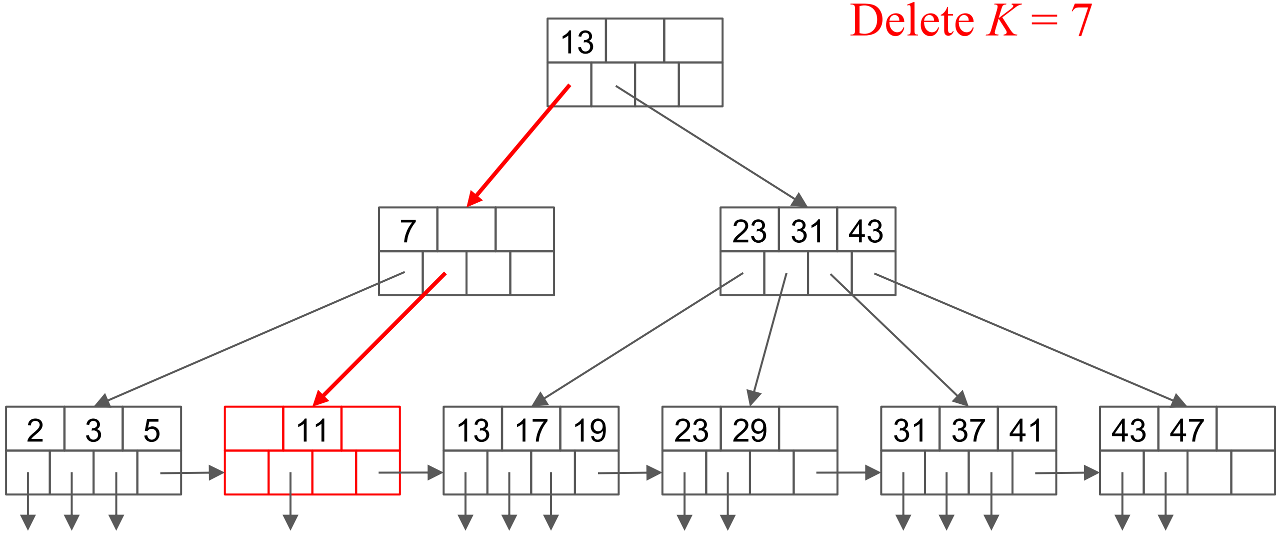
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



Deletion

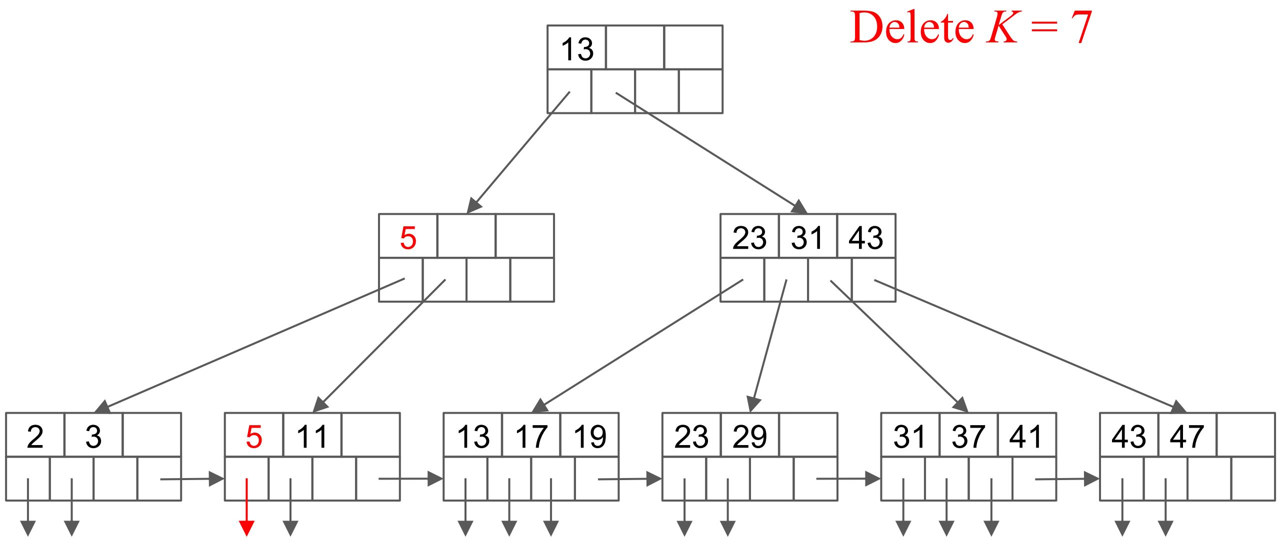
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling

Delete $K = 7$



Deletion

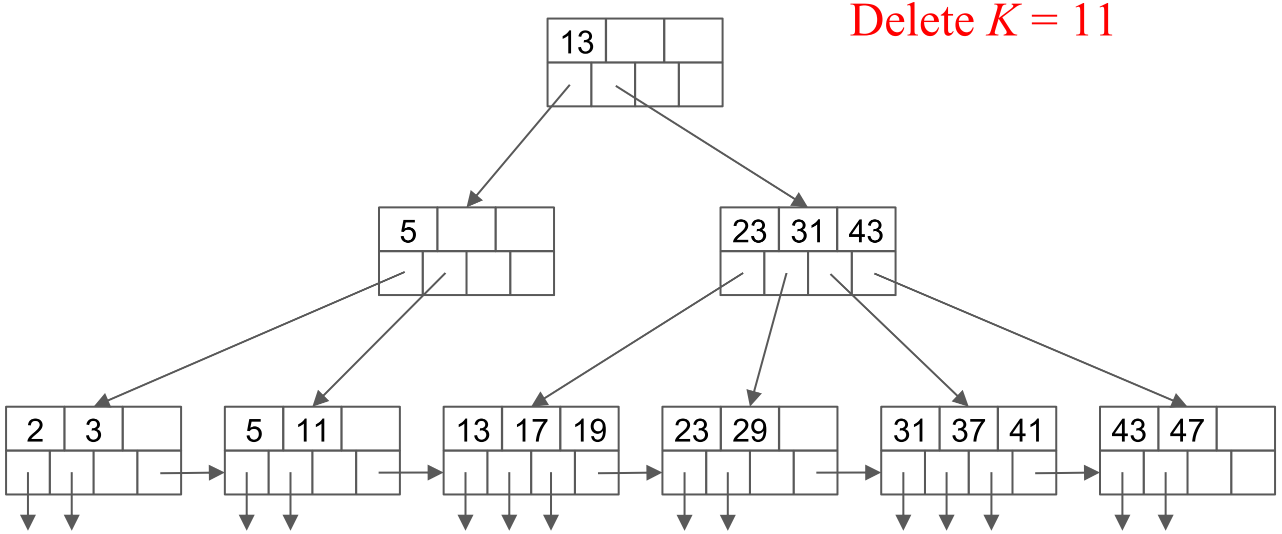
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



Deletion

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

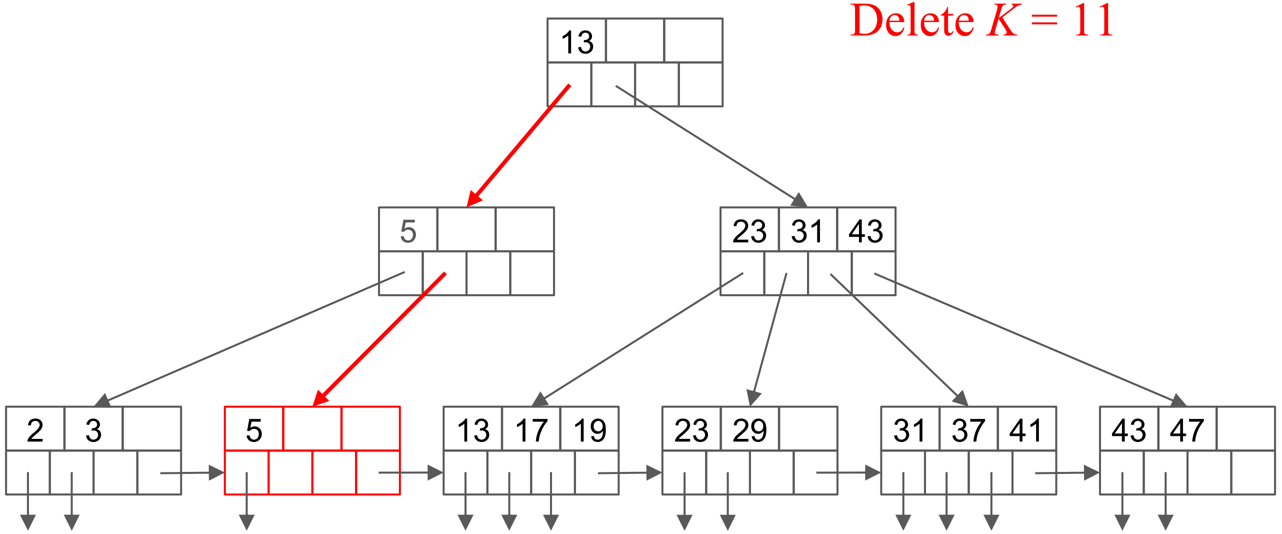
Delete $K = 11$



Deletion

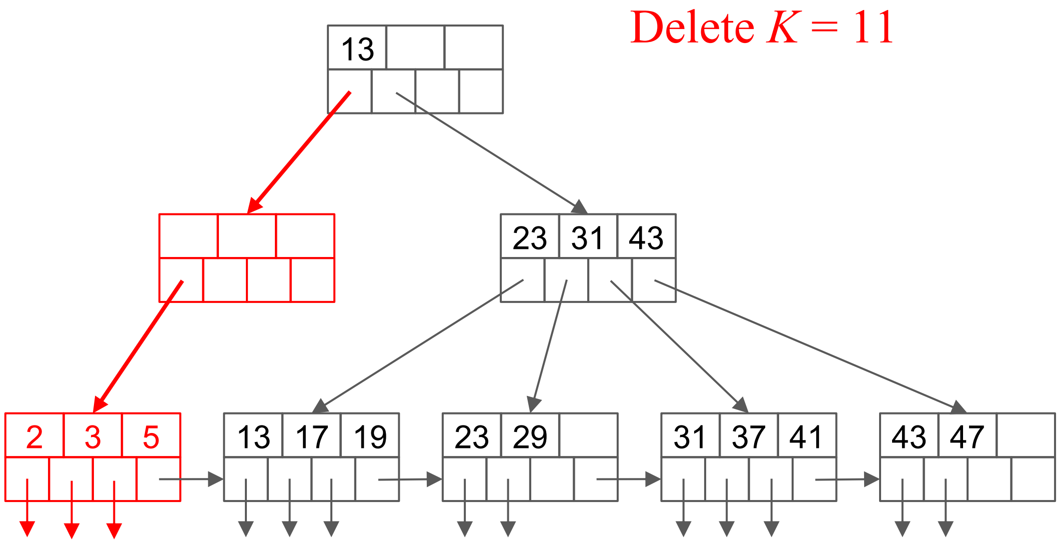
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling

Delete $K = 11$



Deletion

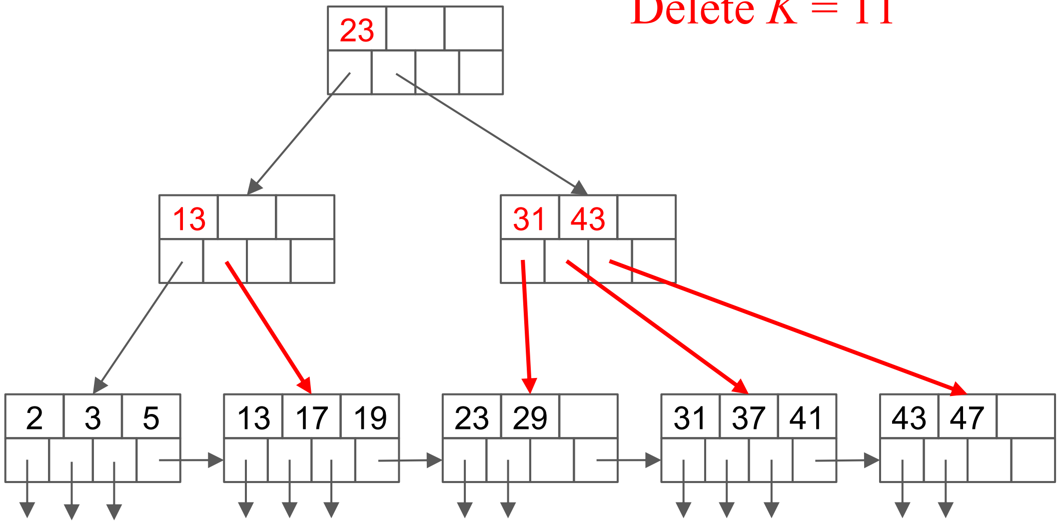
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



Deletion

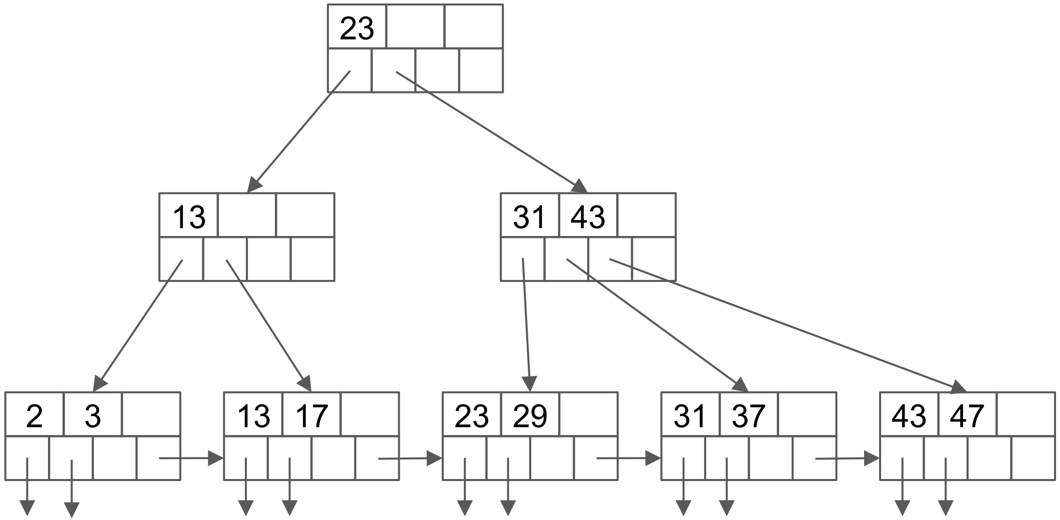
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling

Delete $K = 11$



Exercise #2

- Delete $K = 31$

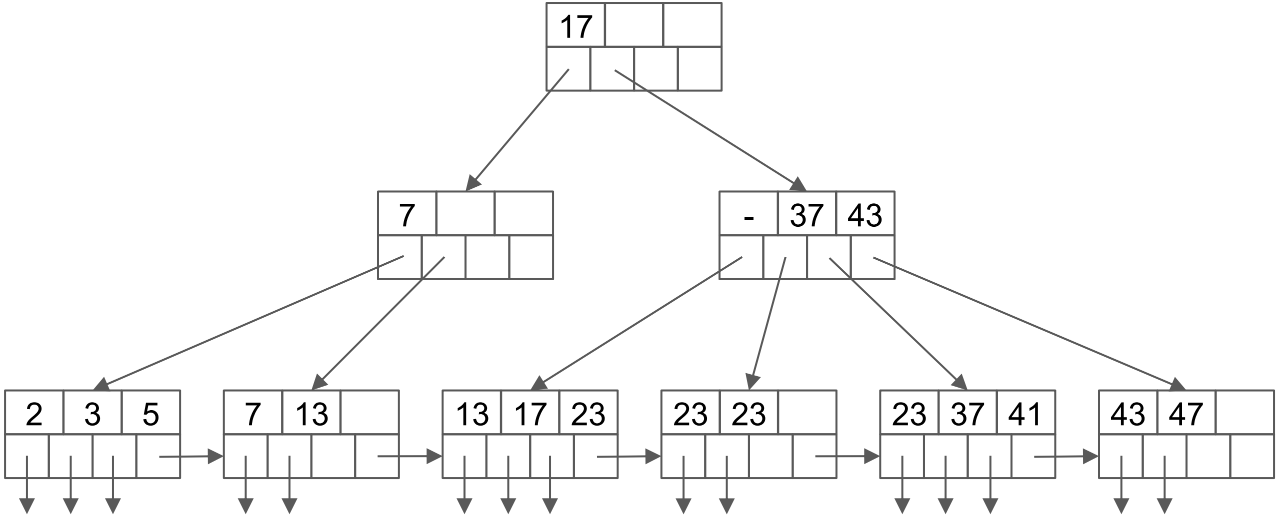


B-tree deletions in practice

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

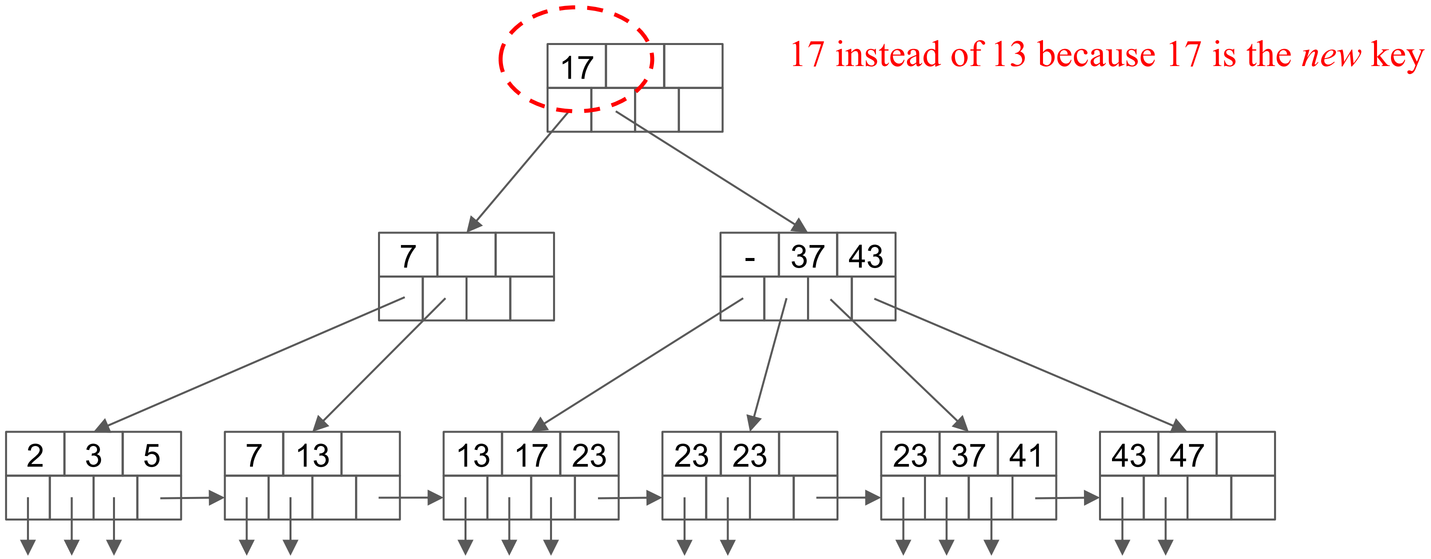
Allowing duplicate keys

- If an interior node has keys $K_1, K_2, \ldots, K_n$, then K_i is the smallest *new* key that appears in the part of subtree accessible from the $(i + 1)$st pointer



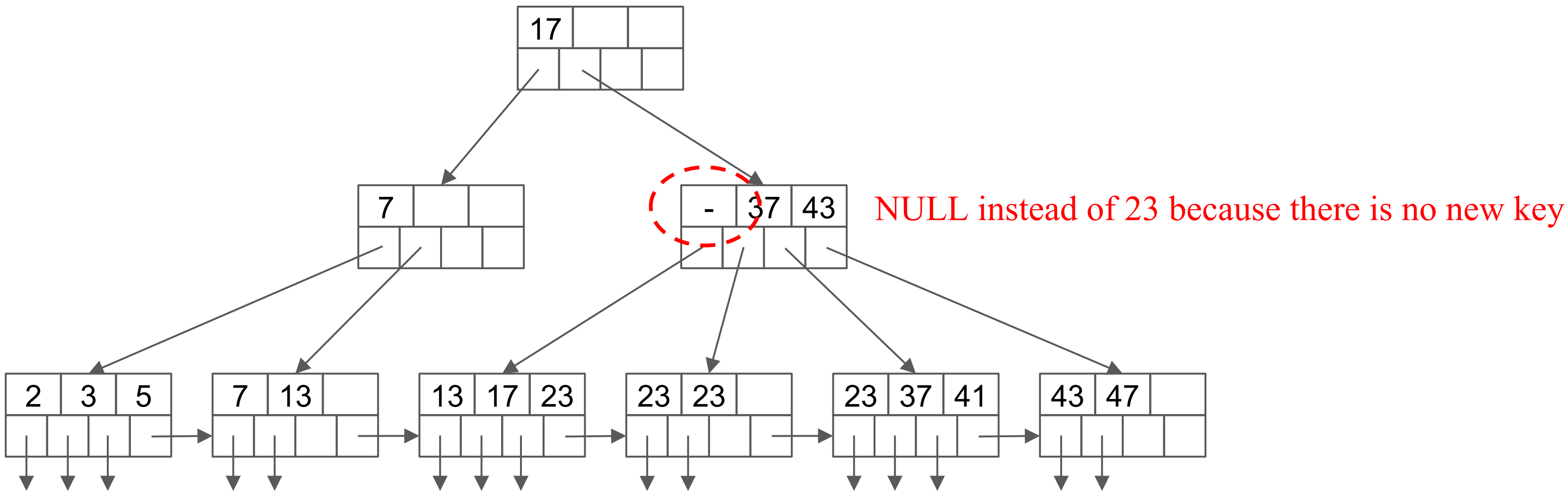
Allowing duplicate keys

- If an interior node has keys K_1, K_2, …, K_n, then K_i is the smallest *new* key that appears in the part of subtree accessible from the $(i + 1)$st pointer



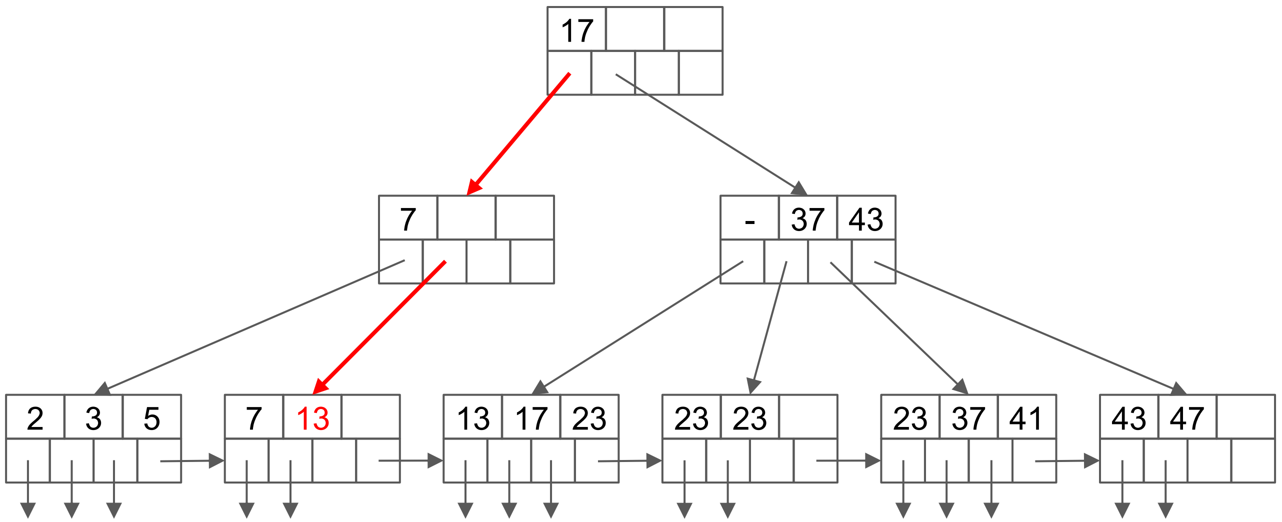
Allowing duplicate keys

- If an interior node has keys $K_1, K_2, \ldots, K_n$, then K_i is the smallest *new* key that appears in the part of subtree accessible from the $(i + 1)$st pointer



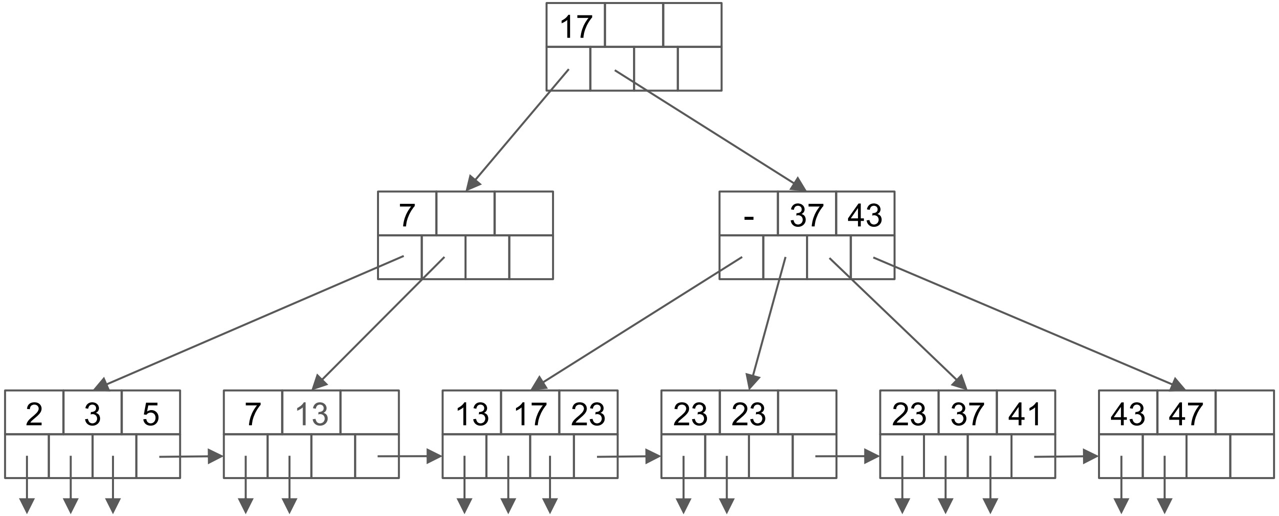
Allowing duplicate keys

- Searching for $K = 13$ can be done correctly



Allowing duplicate keys

- Q: How can we search for $K = 24$?

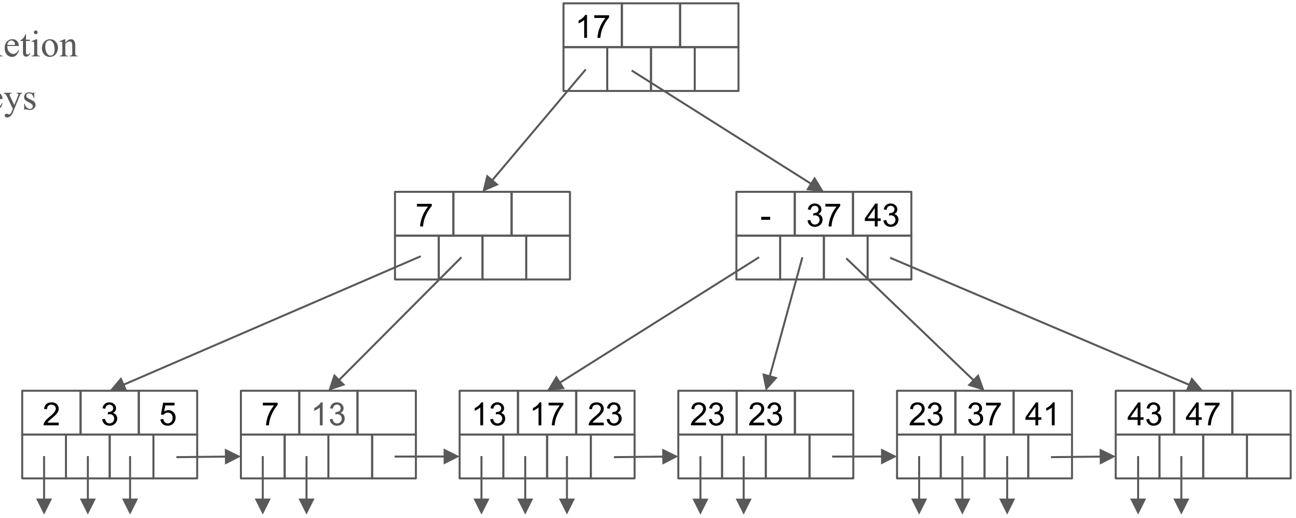


Efficiency

- B-tree reorganizations is negligible in practice if n is reasonably large
  - If a typical block has 100 pointers, a 3-level B-tree has 10,000 leaves and 1 million pointers to records
- The number of disk I/Os needed ≈
  - The number of tree levels (3 is a reasonable number) +
  - One (lookup) or two (insert/delete) for the record manipulation
- We can also keep the root block (and maybe the second-level nodes) permanently buffered in memory to save I/Os

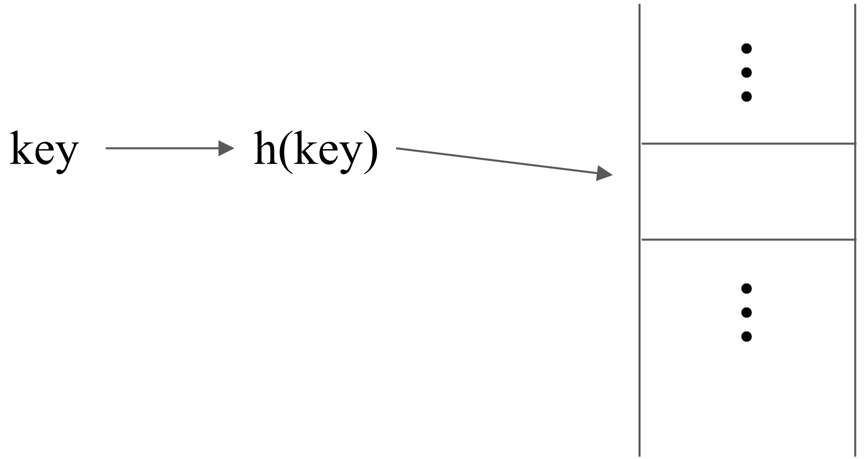
Recap

- B+ tree
  - Lookup, insertion, deletion
  - Handling duplicate keys
  - I/O efficiency



Hash table

- A hash function *h* takes a key and returns a block number from 0 to *B* - 1
- Blocks contain records and are stored in secondary storage
- Complexity:
  - O(1) operation complexity
  - O(n) storage complexity



Hash table: Design Decisions

- Hash Function
  - How to map a large key space into a smaller domain of array offsets
  - Trade-off between fast execution vs. collision rate

- Hashing Scheme
  - How to handle key collisions after hashing
  - Trade-off between allocating a large hash table vs. extra steps to location/insert keys

Hash function

- For any input key, return an integer representation of that key.
  - Output is deterministic
- Example:
  - Given a key that is a string, return the sum of the characters x_i modulo B (*i*.e., $\Sigma x_i \% B$)
    - This function is not idea since there might be many collisions
- We do NOT want to use a cryptographic hash function (e.g., SHA-256) for DBMS hash tables
- In general, we only care about the hash function's speed and collision rate.
- Current SOTA: xxHash