Carnegie Mellon

15-826: Multimedia (Databases) and Data Mining

Lecture#2: Primary key indexing – B-trees

Christos Faloutsos - CMU

www.cs.cmu.edu/~christos

Reading Material

[Ramakrishnan & Gehrke, 3rd ed, ch. 10]

Problem

Given a large collection of (multimedia) records, find similar/interesting things, ie:

- Allow fast, approximate queries, and
- Find rules/patterns

Outline

Goal: 'Find similar / interesting things'

- Intro to DB
- Indexing - similarity search
- Data Mining

Indexing - Detailed outline

- primary key indexing
  - B-trees and variants
  - (static) hashing
  - extendible hashing
- secondary key indexing
- spatial access methods
- text
- ...

In even more detail:

- B – trees
- B+ - trees, B*-trees
- hashing

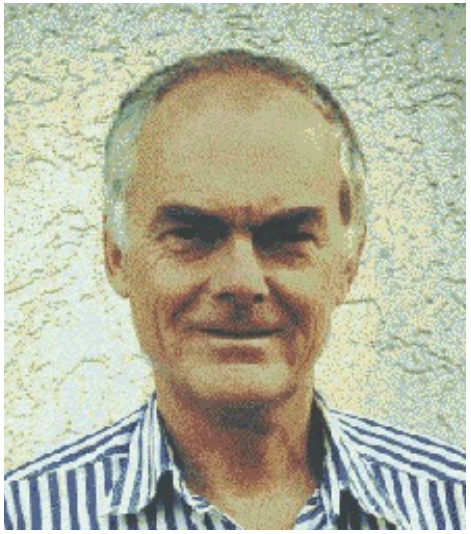
Primary key indexing

- find employee with ssn=123

B-trees

- the **most successful** family of index schemes (B-trees, B^{+}-trees, B^{*}-trees)
- Can be used for primary/secondary, clustering/non-clustering index.
- balanced "n-way" search trees

Citation

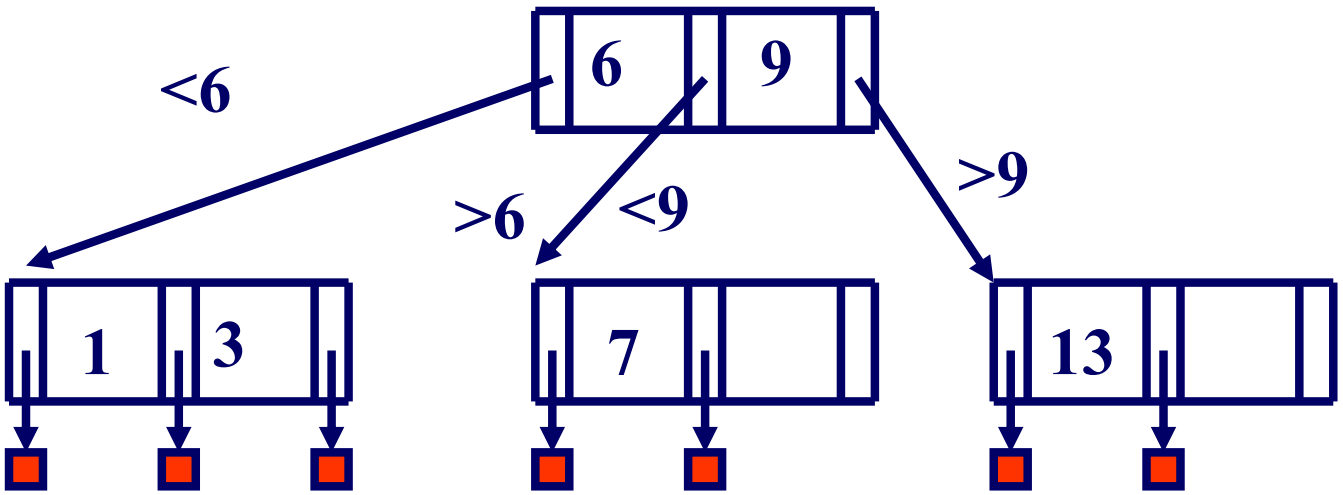


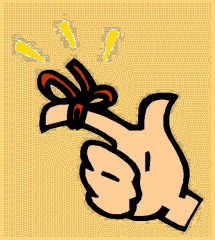
- Rudolf Bayer and Edward M. McCreight, *Organization and Maintenance of Large Ordered Indices*, Acta Informatica, 1:173-189, 1972.

- Received the *2001 SIGMOD innovations* award
- among the most cited db publications
  - www.informatik.uni-trier.de/~ley/db/about/top.html

B-trees

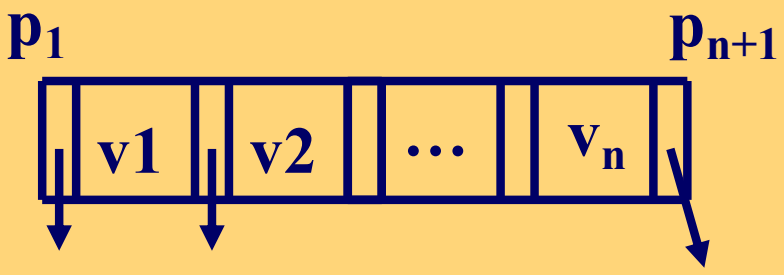
Eg., B-tree of order 3:

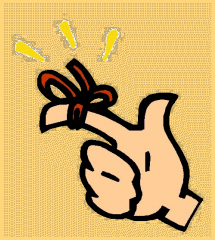




B - tree properties:

- each node, in a B-tree of order *d*:
  - Key order
  - at most $n=2d$ keys
  - at least *d* keys (except root – it may have just 1 key)
  - all leaves at the same level
  - if number of pointers is *k*, then node has exactly *k-1* keys
  - (leaves are empty)



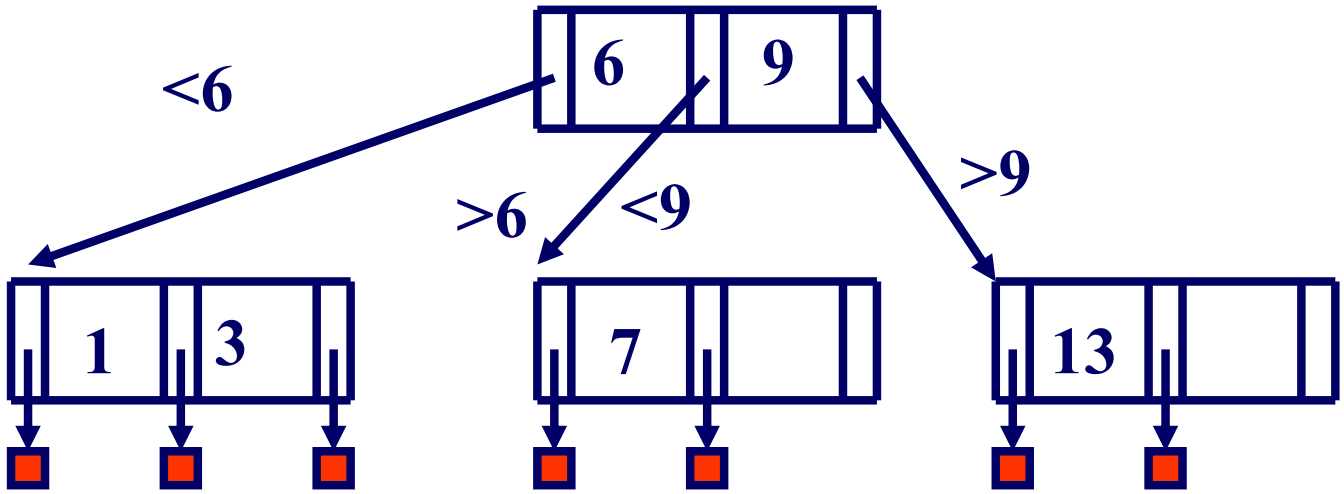


Properties

- "block aware" nodes: each node -> disk page
- O(log (N)) for everything! (ins/del/search)
- typically, if n = 50 - 100, then 2 - 3 levels
- utilization >= 50%, guaranteed; on average 69%

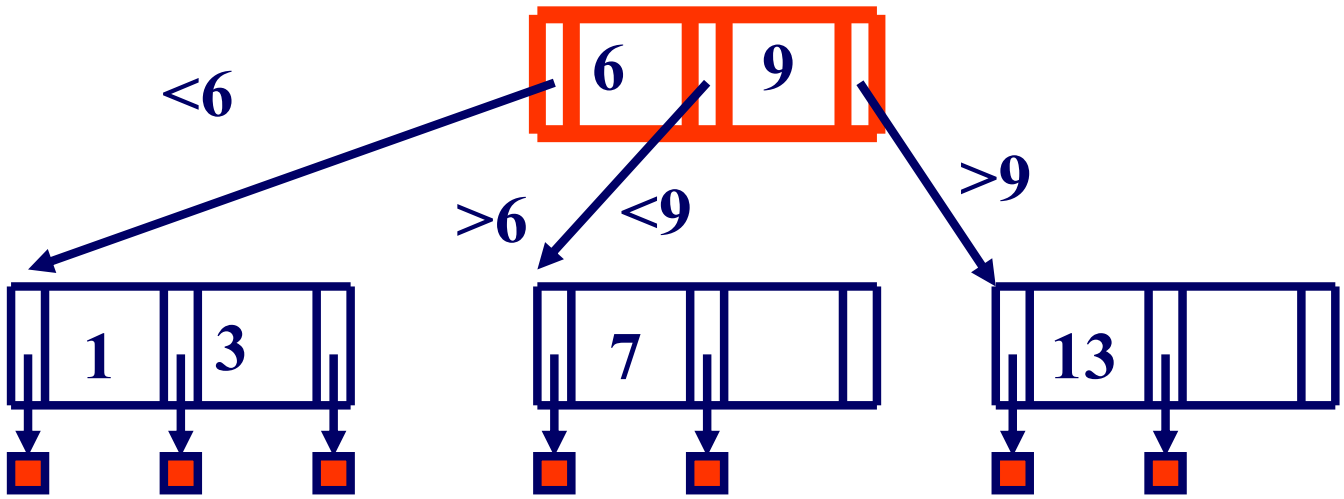
Queries

- Algo for exact match query? (eg., ssn=8?)



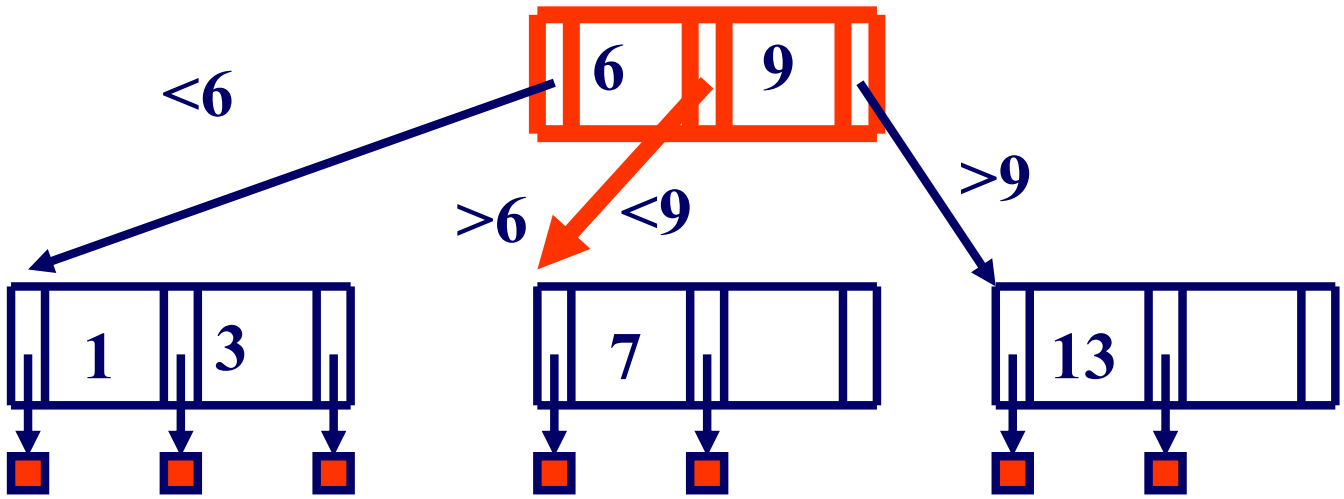
Queries

- Algo for exact match query? (eg., ssn=8?)



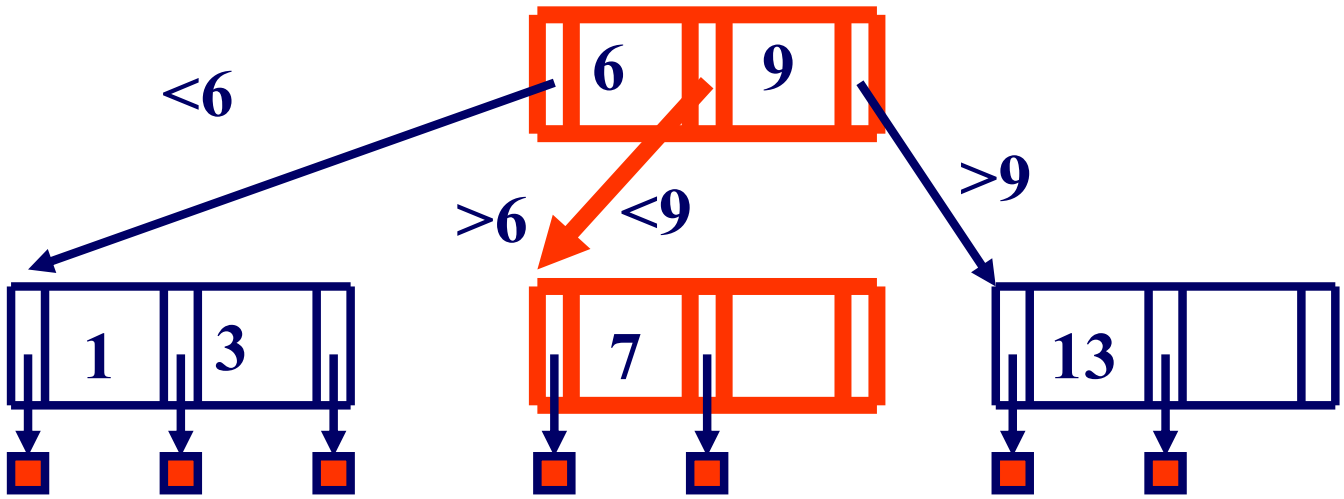
Queries

- Algo for exact match query? (eg., ssn=8?)



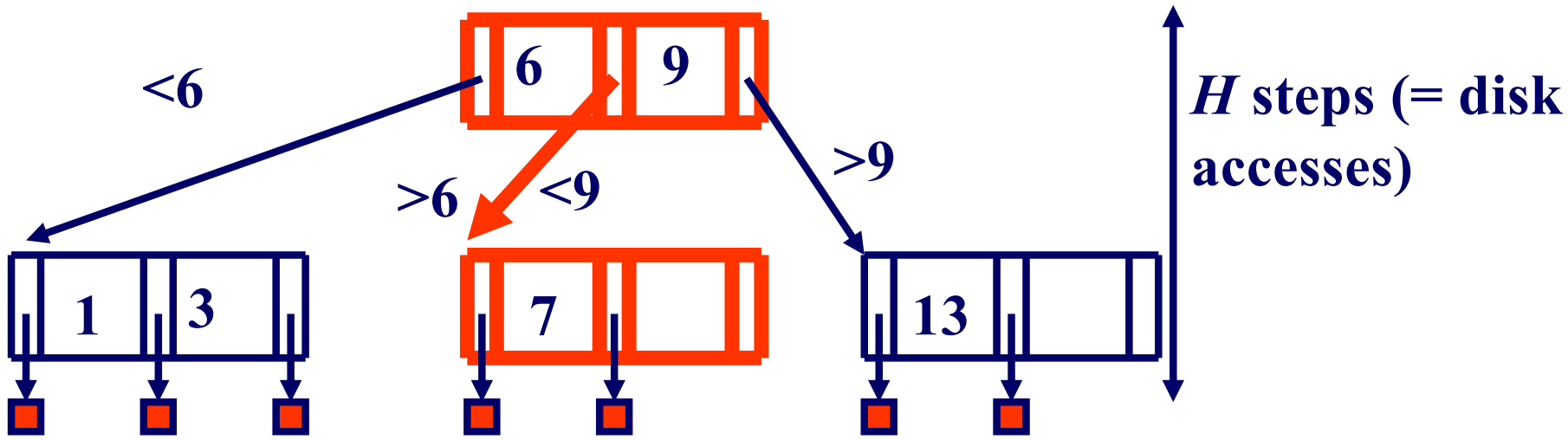
Queries

- Algo for exact match query? (eg., ssn=8?)



Queries

- Algo for exact match query? (eg., ssn=8?)

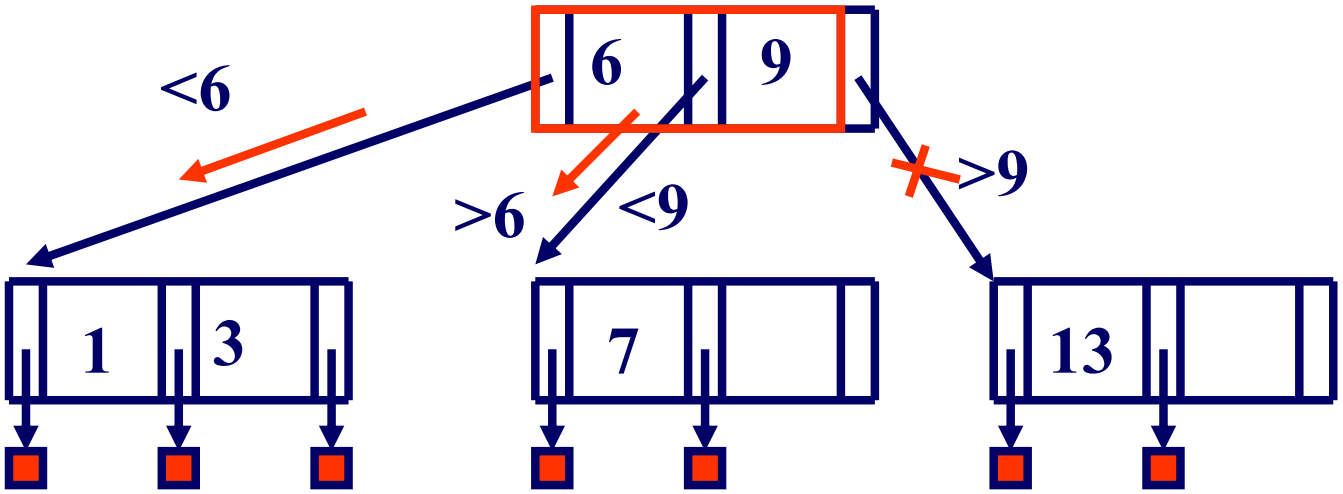


Queries

- what about range queries? (eg., $5<salary<8$)
- Proximity/ nearest neighbor searches? (eg., $salary \sim 8$)

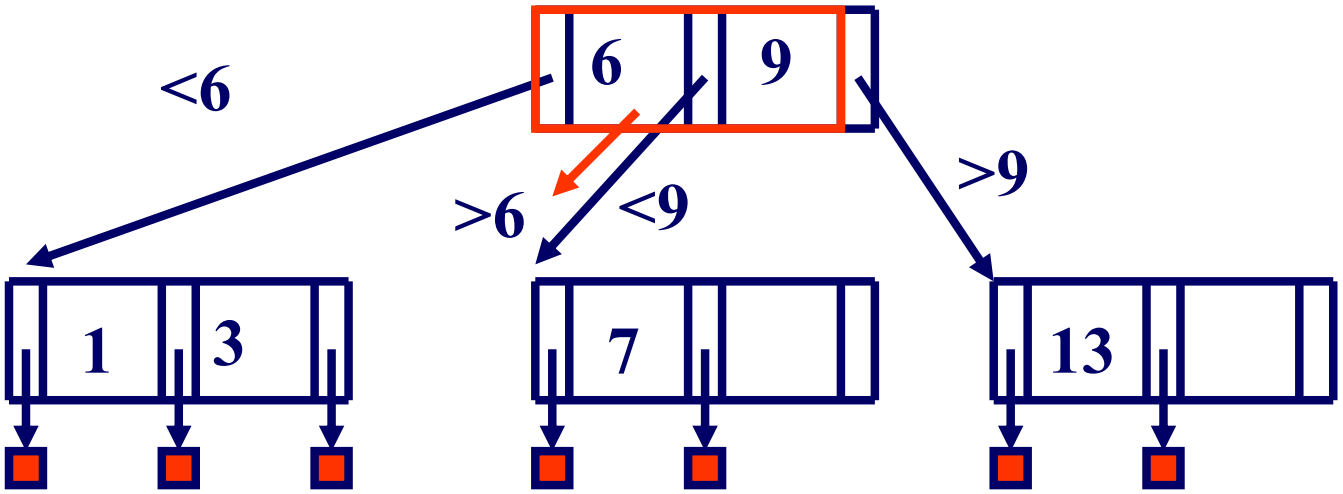
Queries

- what about range queries? (eg., ***5<salary<8***)
- Proximity/ nearest neighbor searches? (eg., *salary ~ 8*)



Queries

- what about range queries? (eg., *5<salary<8*)
- Proximity/ nearest neighbor searches? (eg., ***salary ~ 8***)

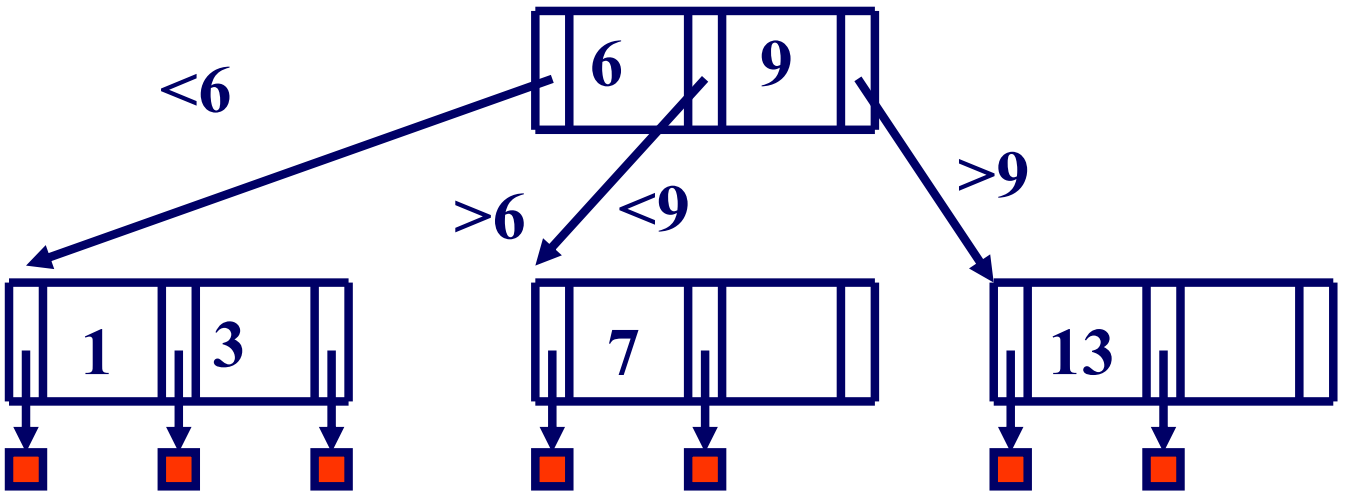


B-trees: Insertion

- Insert in leaf; on overflow, push middle up (recursively)
- split: preserves B - tree properties

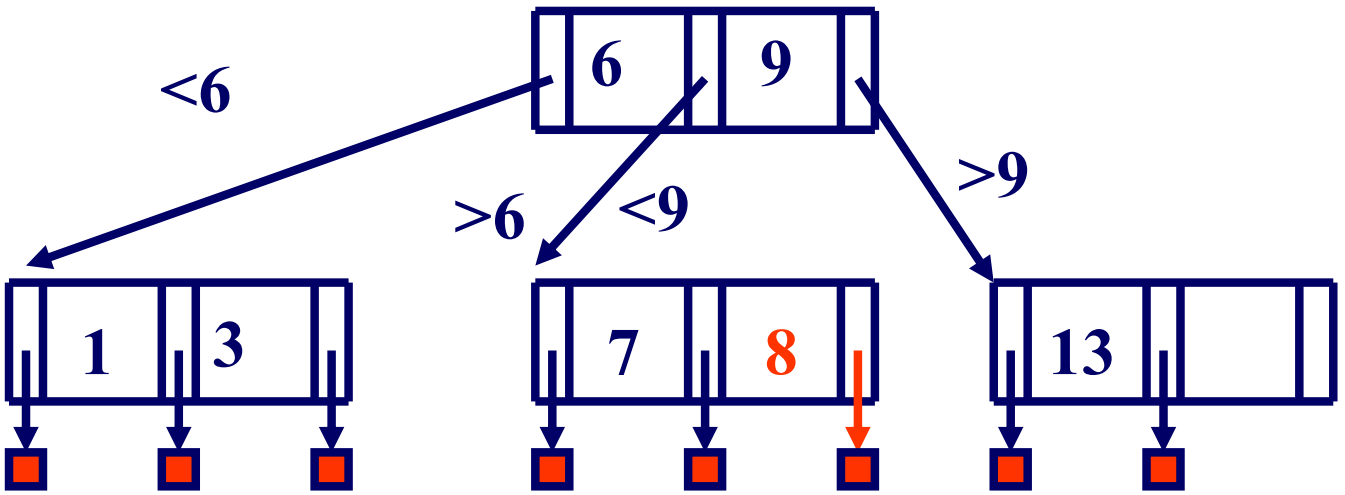
B-trees

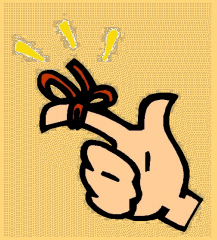
Easy case: Tree T0; insert '8'



B-trees

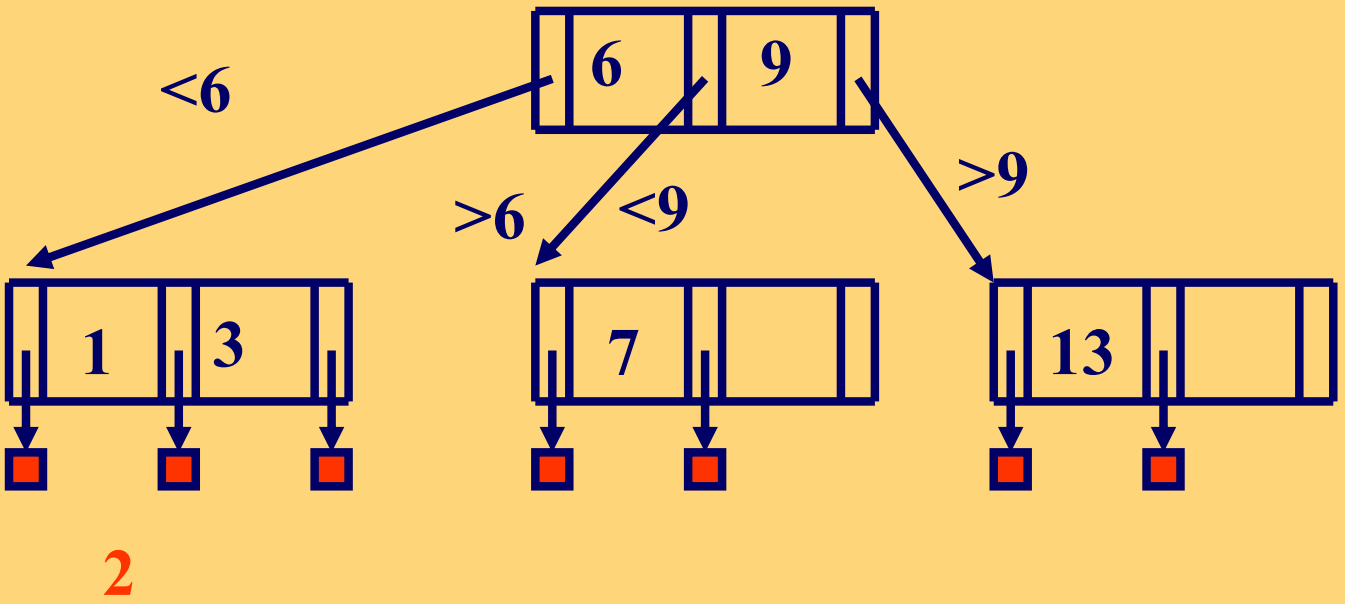
Tree T0; insert ‘8’

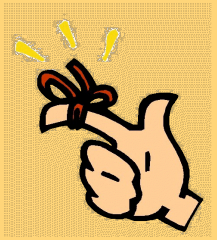




B-trees

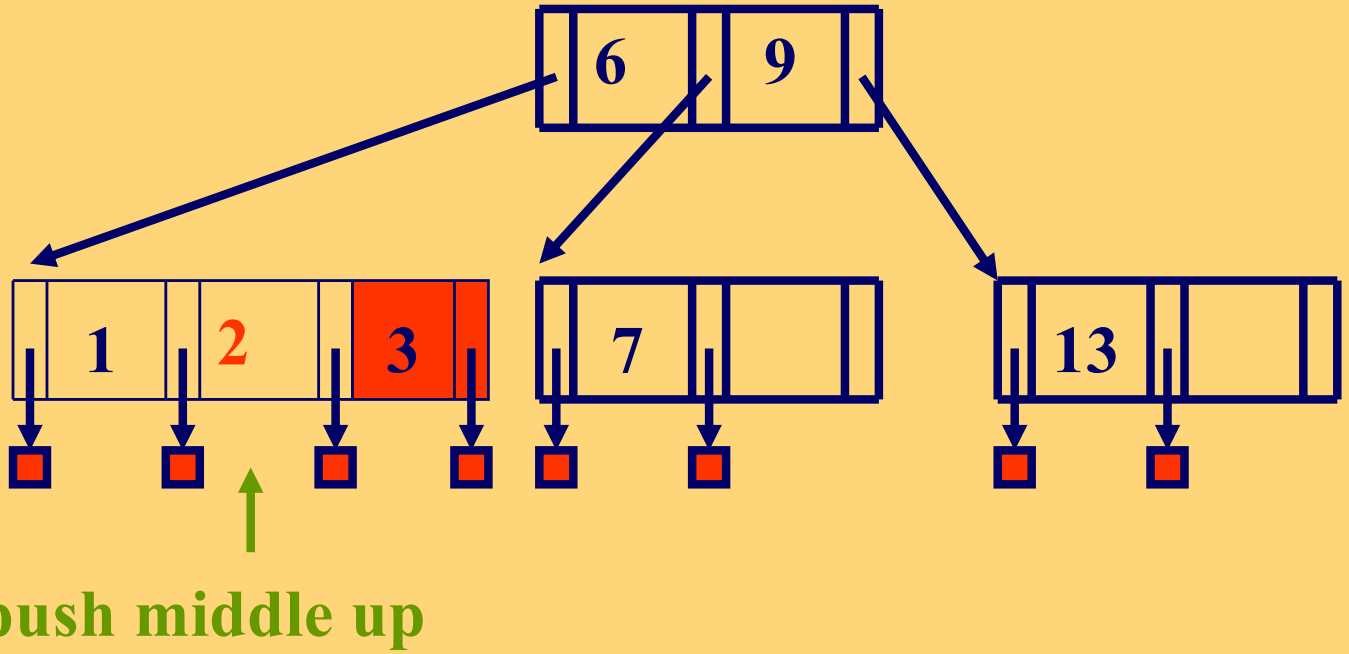
Hardest case: Tree T0; insert '2'

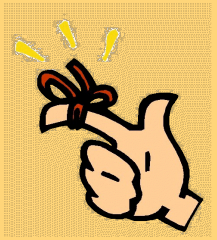




B-trees

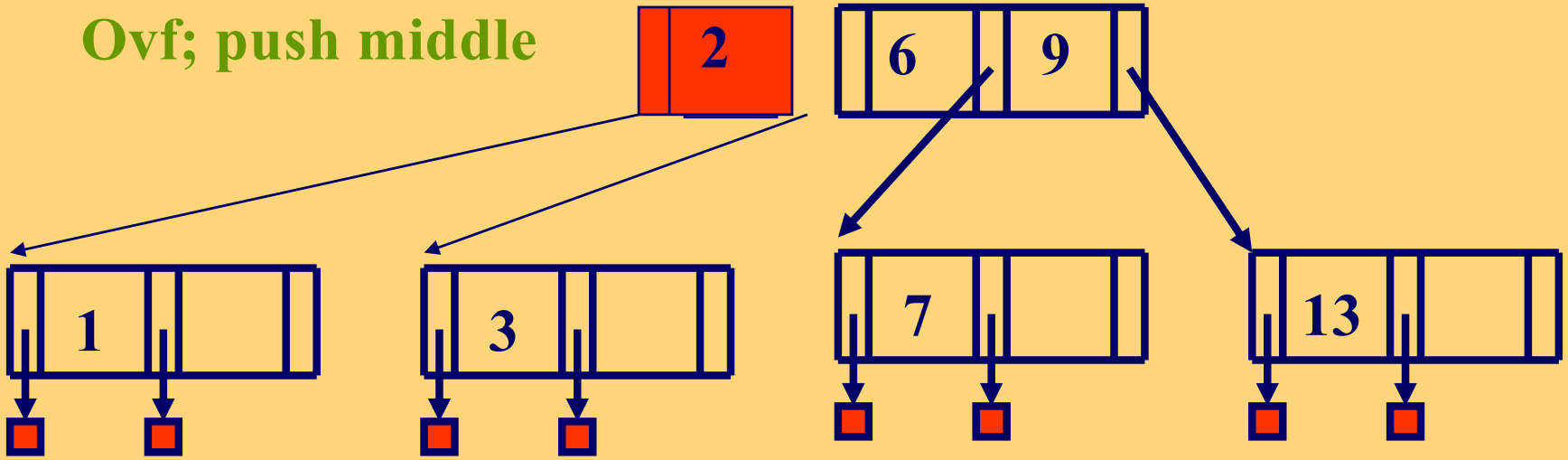
Hardest case: Tree T0; insert '2'

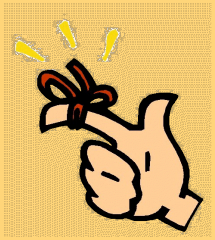




B-trees

Hardest case: Tree T0; insert '2'

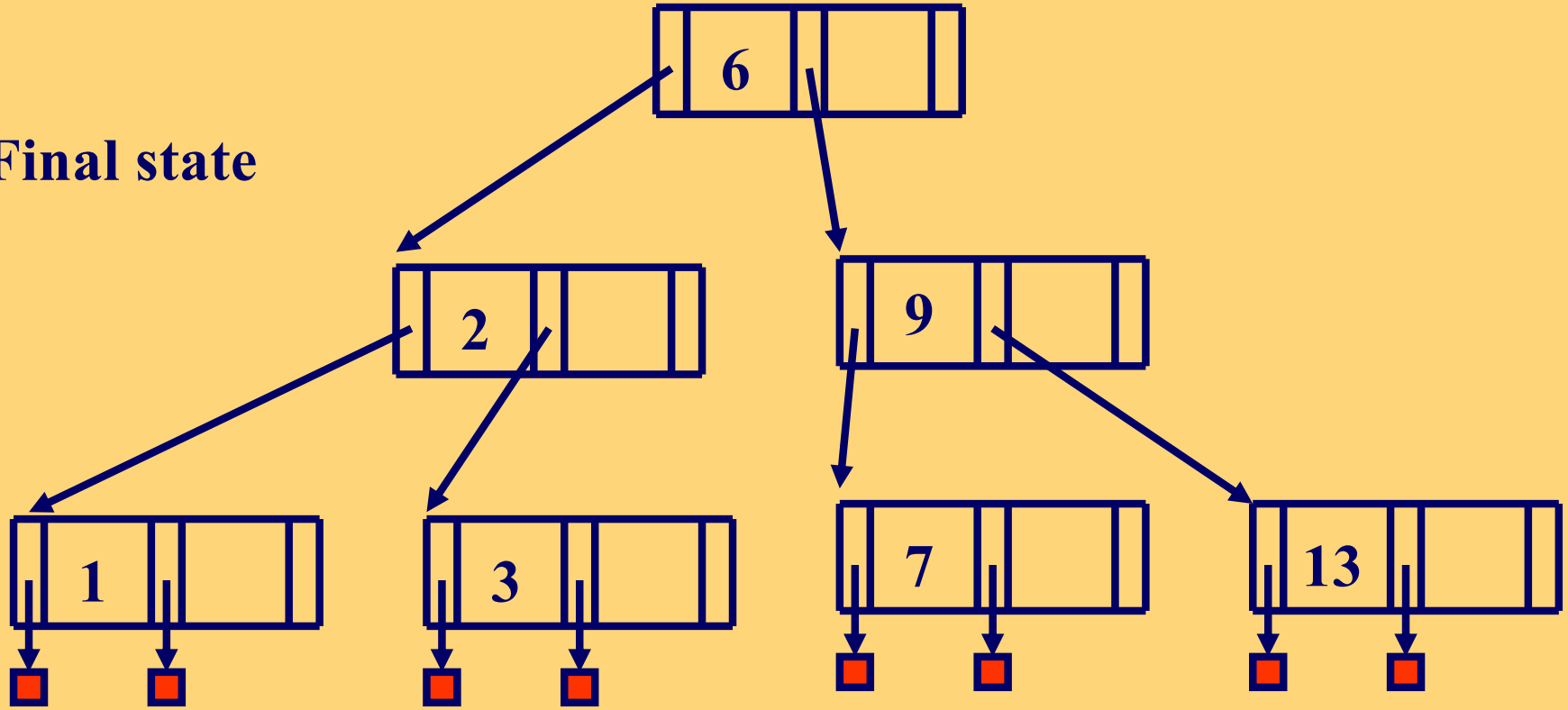




B-trees

Hardest case: Tree T0; insert '2'

Final state



B-trees: Insertion

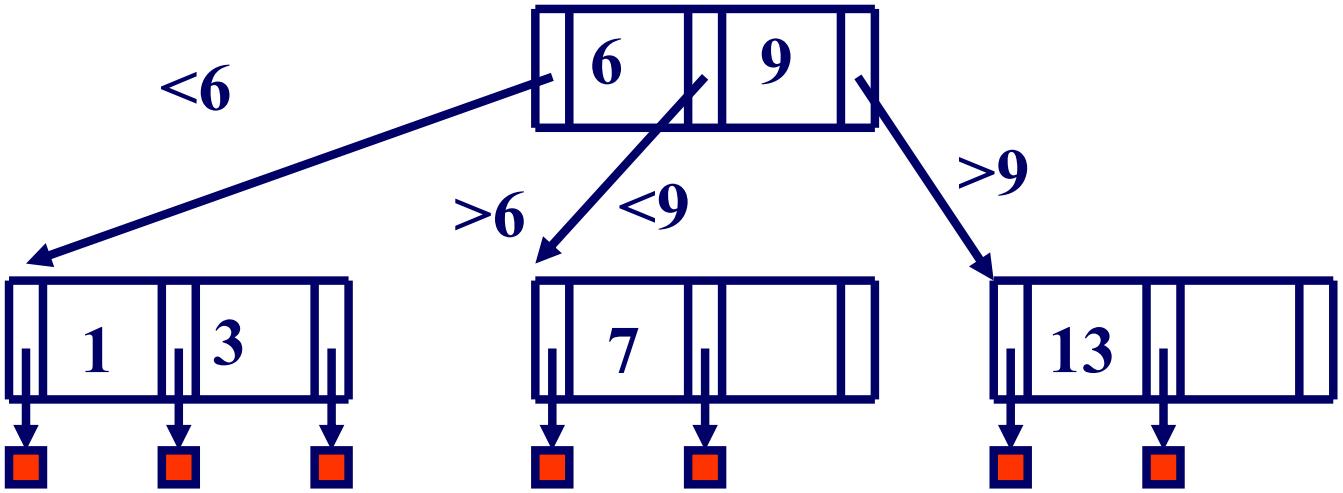
- Insert in leaf; on overflow, push middle up (recursively – 'propagate split')
- split: preserves all B - tree properties (!!)
- notice how it grows: height increases when root overflows & splits
- Automatic, incremental re-organization

Overview

- B – trees
- **B+ - trees, B*-trees**
- hashing

B+ trees - Motivation

if we want to store the whole record with the key –> problems (what?)

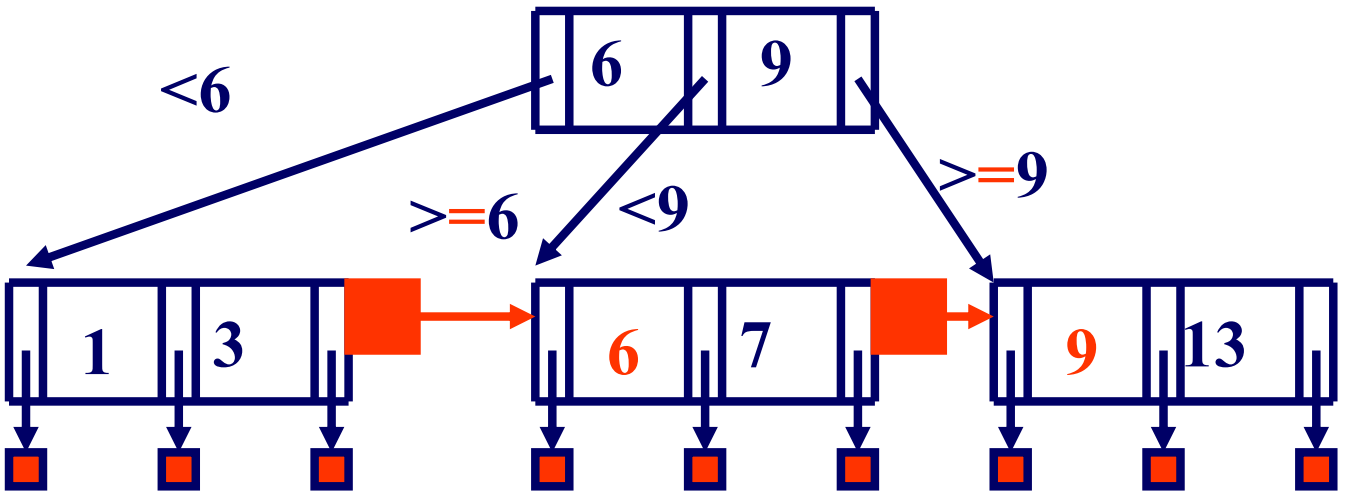


Solution: B^+ - trees

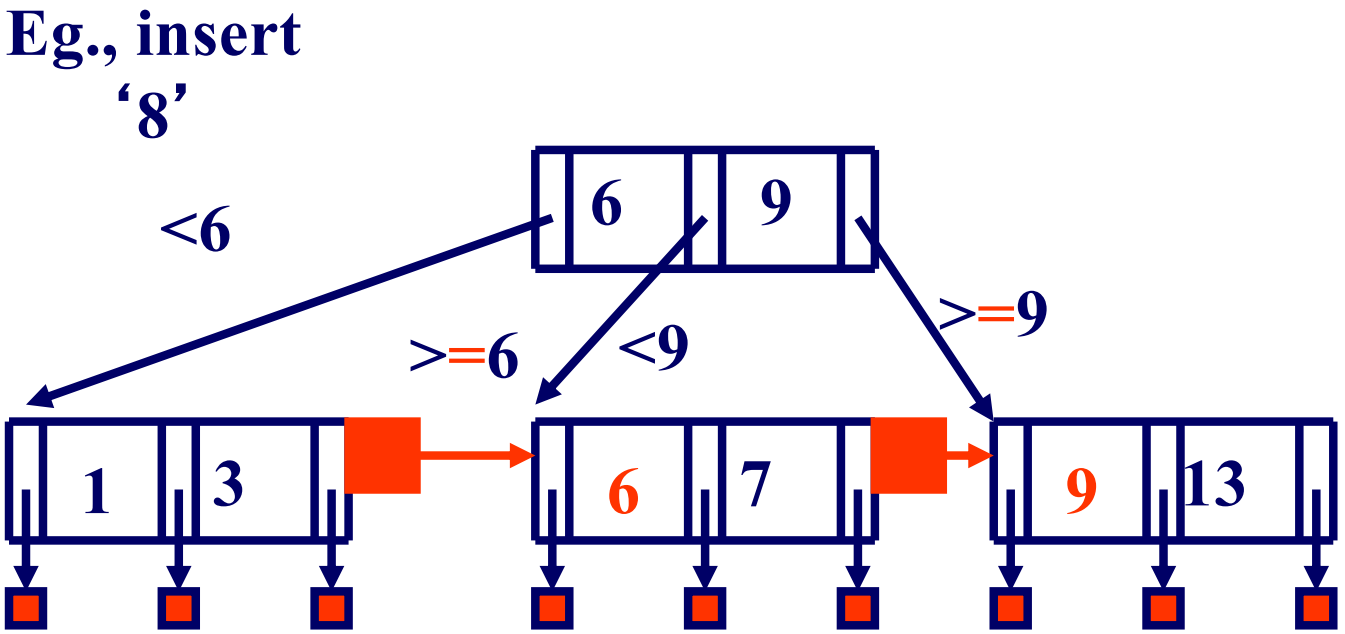
- They string all leaf nodes together
- AND
- replicate keys from non-leaf nodes, to make sure every key appears at the leaf level

Carnegie Mellon

B+ trees



B+ trees - insertion



Overview

- B – trees
- B+ - trees, **B*-trees**
- hashing

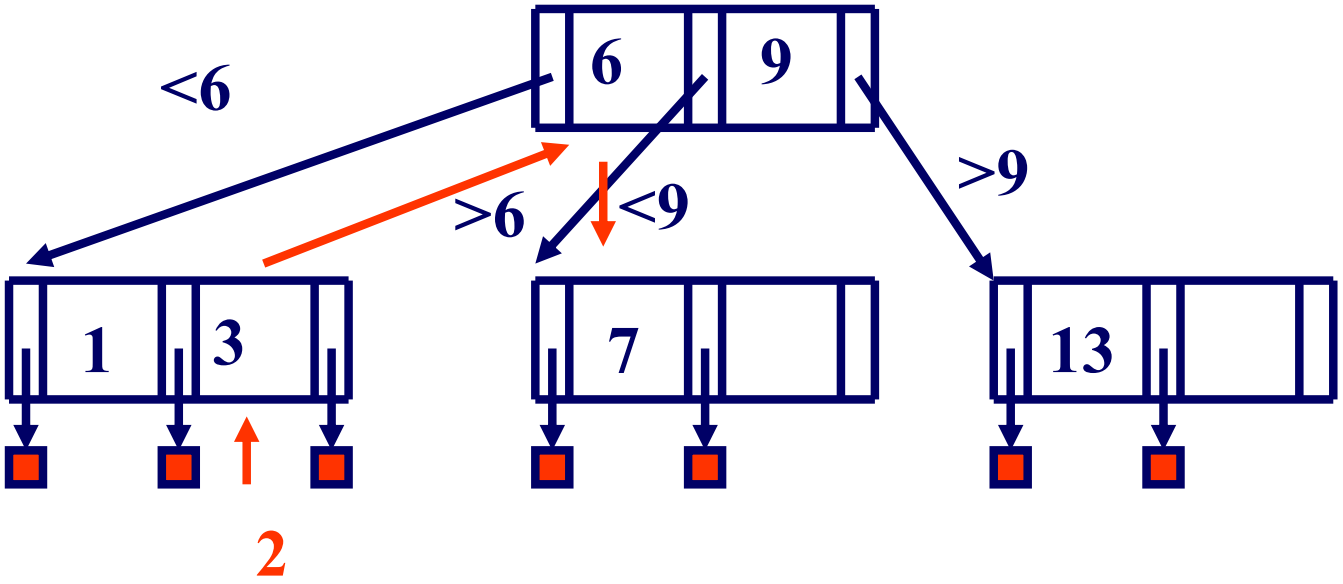
B*-trees

- splits drop util. to 50%, and maybe increase height
- How to avoid them?

B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling!

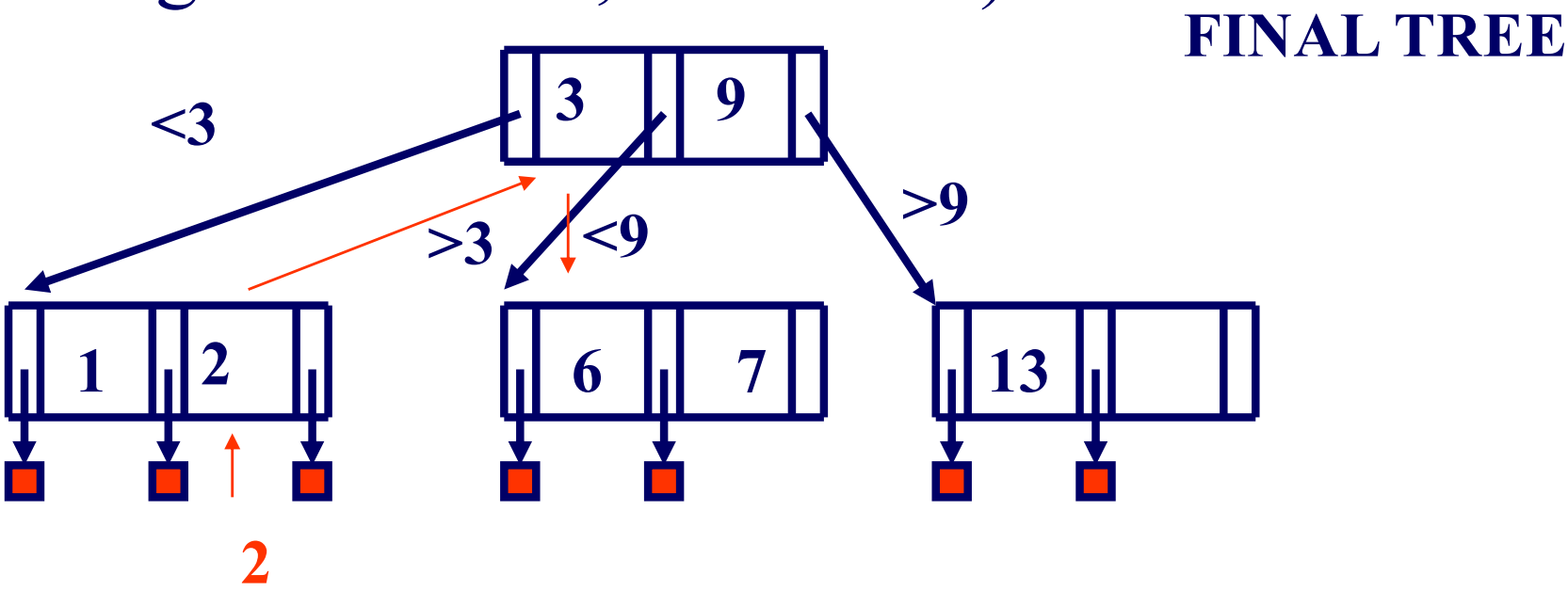
(through PARENT, of course!)



B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling!

(through PARENT, of course!)



B*-trees: deferred split!

- Notice: shorter, more packed, faster tree
- It's a rare case, where space utilization and speed improve together
- BUT: What if the sibling has no room for our 'lending'?

B*-trees: deferred split!

- BUT: What if the sibling has no room for our 'lending'?
- A: 2-to-3 split: get the keys from the sibling, pool them with ours (and a key from the parent), and split in 3.
- Details: too messy (and even worse for deletion)

Conclusions

- Main ideas: recursive; block-aware; on overflow -> split; **defer** splits
- All B-tree variants have excellent, **O(logN) worst-case performance for ins/del/search**
- B+ tree is the prevailing indexing method
- More details: [Knuth vol 3.] or [Ramakrishnan & Gehrke, 3rd ed, ch. 10]