CS 6400 A

Database Systems Concepts and Design

Lecture 12
10/12/26

Announcements

- Assignment 2 leaderboard is live!
- Project proposal grade will be released
- Reminder for midterm regrade request (until Wednesday midnight)
- Quiz 1 details:
  - Time: 30 min, during class time Monday Oct 19
  - Format: Open book and notes, closed Internet (same as midterm)
  - Contents covered: lec 9 (hardware, storage) – lec 12 (b+ tree)
  - Files->Past Exams->cs6400_fa25_final.pdf, Q1 Access Methods

Assignment 2: NL2SQL Leaderboard

This is the leaderboard which decides the extra credit.
Each row is one student's latest graded submission.
Overall is the average accuracy of the two databases.
To find your rank, look for the row with the two accuracies from your feedback comment.
We check for new grades every 5 minutes. Last checked Oct 12, 3:57 PM ET.

Rank	Toxicology	Thrombosis Prediction	Overall
1	67.9%	67.9%	**67.9%**
2	62.3%	47.2%	**54.7%**
Starter code	39.6%	11.3%	25.5%

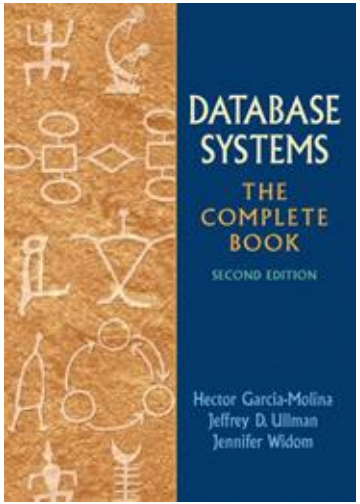
Agenda

1. B+-Tree Basics
2. B+-Tree Operations
3. B+ Tree Cost Model

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 14.2: B-Tree



Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang, CS145 (Intro to Big Data Systems) taught by Peter Bailis, and CS 6530 (Advanced Database Systems) taught by Prashant Pandey.

1. B+-Tree Basics

B Tree/B+ Tree Overview

They are search trees

- Common misunderstanding: B does not mean binary!
- More general index structure that is commonly used in commercial DBMS's

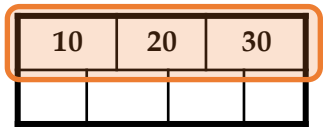
Idea in B-Trees:

- Balanced, height adjusted tree
- Stores data (keys and values) in all nodes (both internal and leaf)
- Leaf nodes are independent; no connections between them

Idea in B+-Trees:

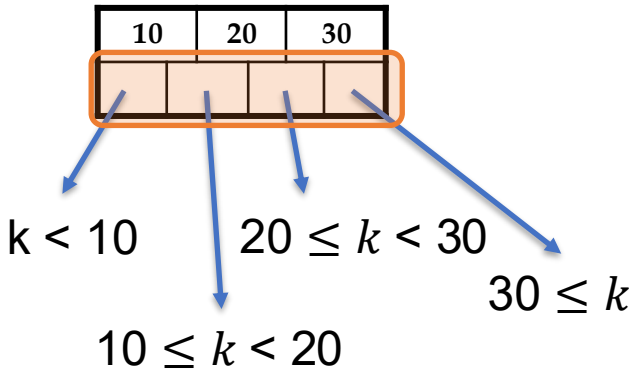
- Stores data only in leaf nodes; make leaves into a linked list (for range queries)
- Most popular variant (our focus this lecture)

B+ Tree Basics



n keys in a node

B+ Tree Basics

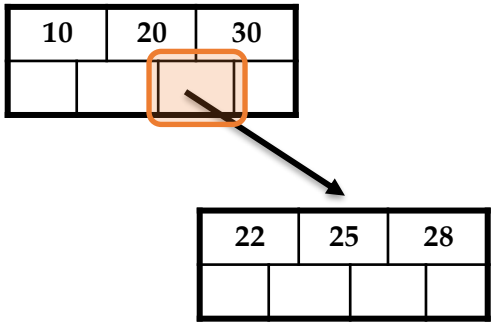


The n keys in a node define n+1 ranges

Maximum fanout (f): Maximum number of child pointers per internal node

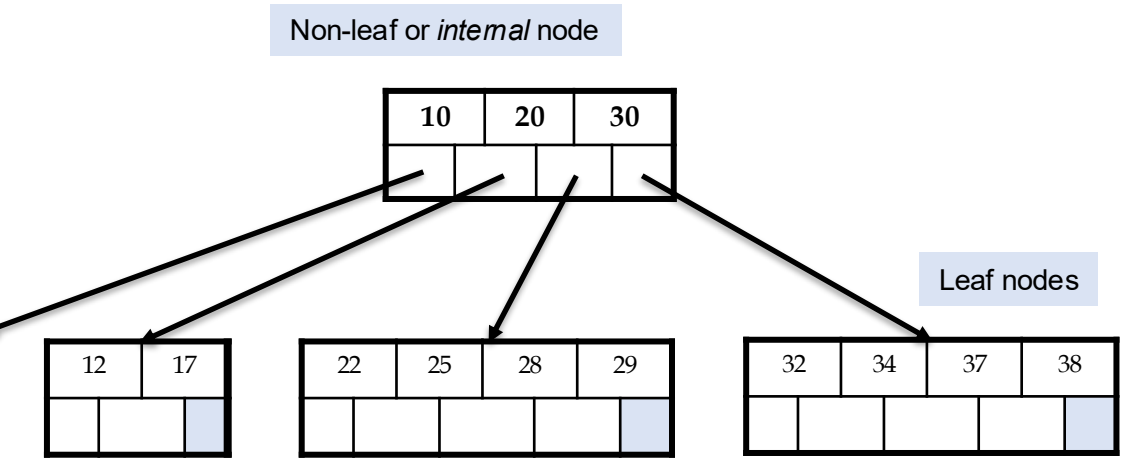
B+ Tree Basics

Non-leaf or *internal* node



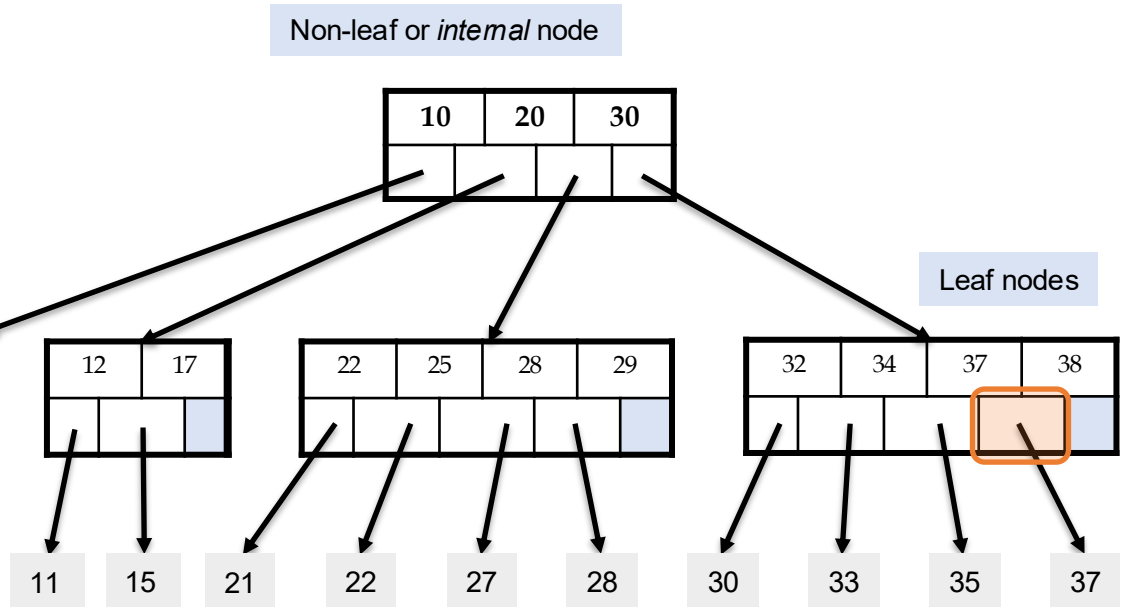
For each range, in a non-leaf node, there is a pointer to another node with keys in that range

B+ Tree Basics



Leaf nodes are different from internal nodes:

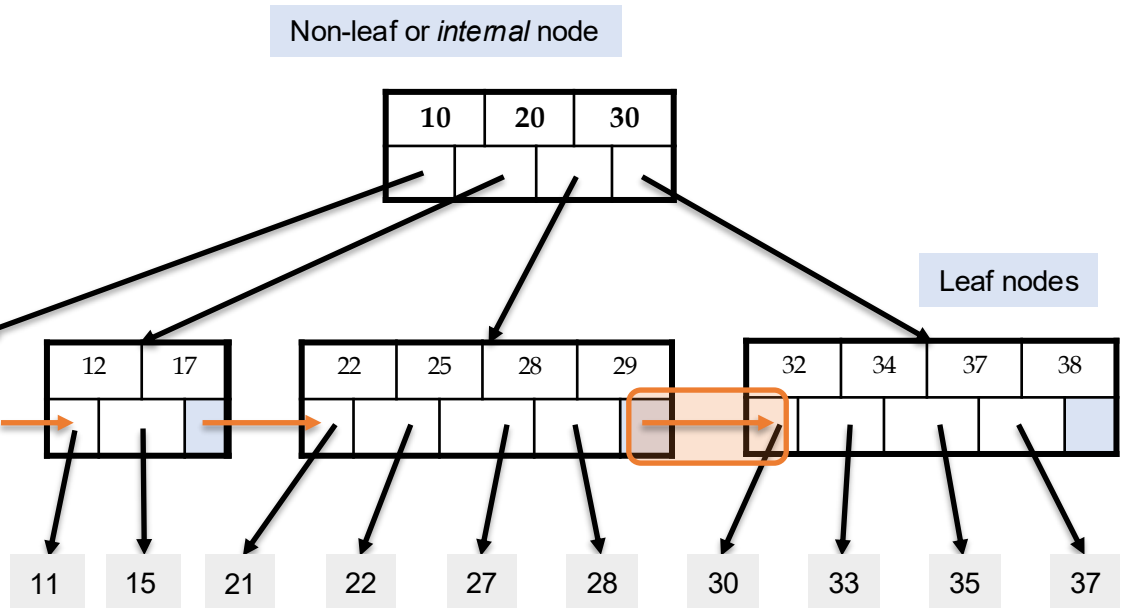
B+ Tree Basics



Leaf nodes are different from internal nodes:

Their key slots contain pointers to data records

B+ Tree Basics



Assuming unclustered index

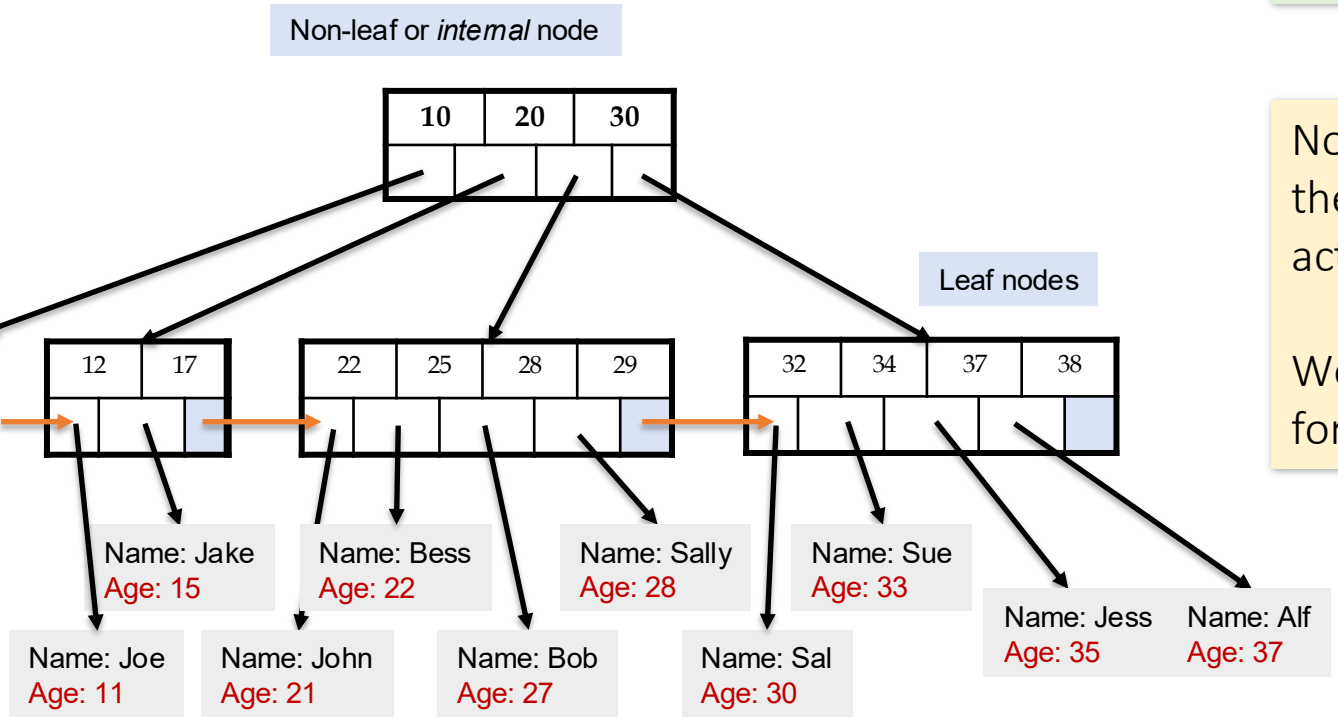
Leaf nodes are different from internal nodes:

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

Assuming unclustered index



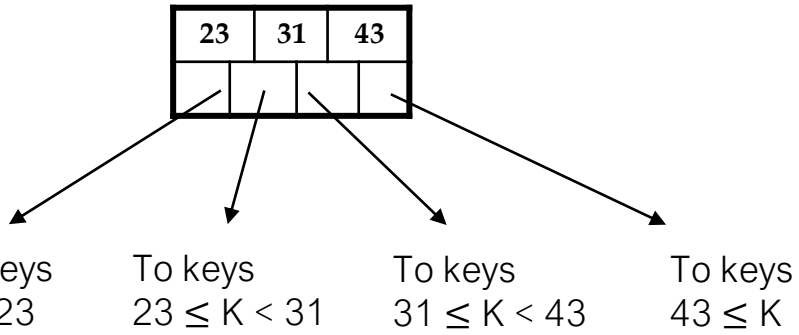
Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display…

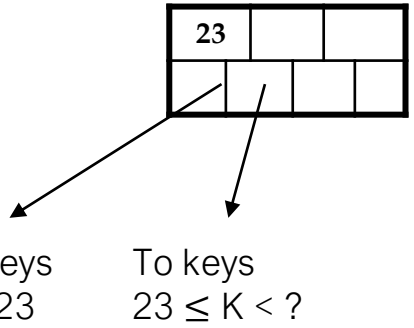
B+ Tree occupancy requirement: interior nodes

Every node (except root) must be at least ”half-full”

Full



Minimal

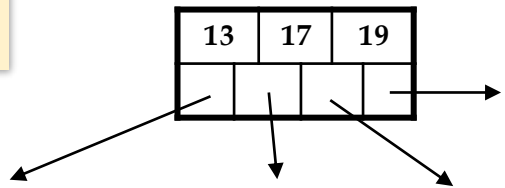


At least half of the **pointers** much be used

B+ Tree occupancy requirement: leaf nodes

Every node (except root) must be at least ”half-full”

Full



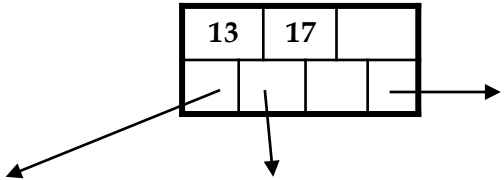
To next leaf in sequence

To record with key 13

To record with key 17

To record with key 19

Minimal



Counts even if null

To record with key 13

To record with key 17

At least half of the **keys** must be used

Occupancy requirement: why does it matter?

Ensure that the tree remains balances

- Nodes split when they get too full
- Nodes merge/redistribute when they get too empty

Expected tree height: O(log N)
Important for minimizing disk I/Os

Stable and predictable performance

- Under modifications
- Also good for query planning

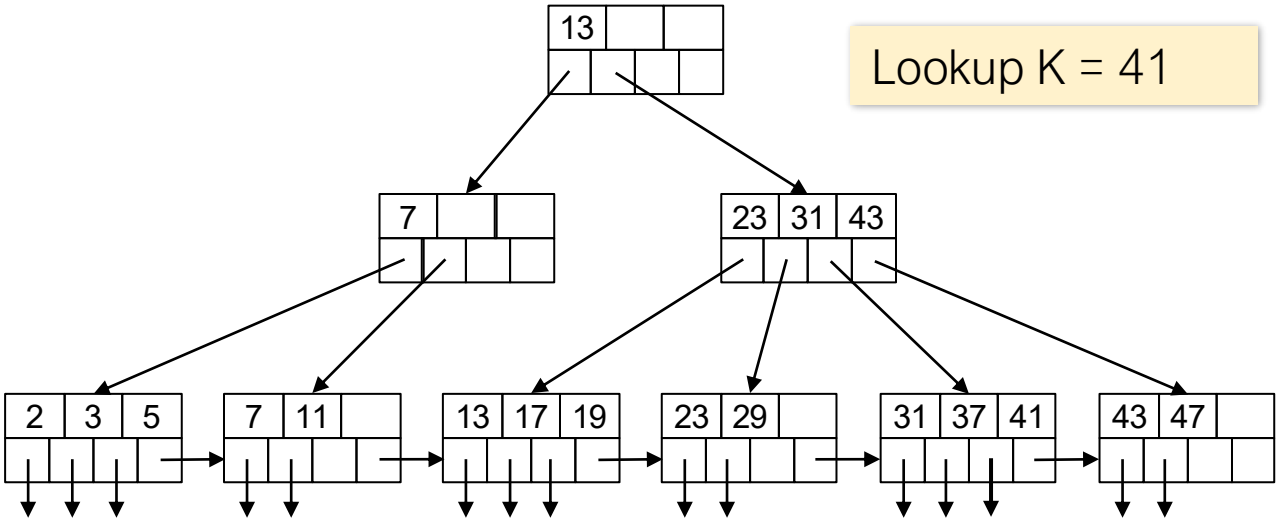
Efficient space utilization

- at least 50% full

2. B+-Tree Operations

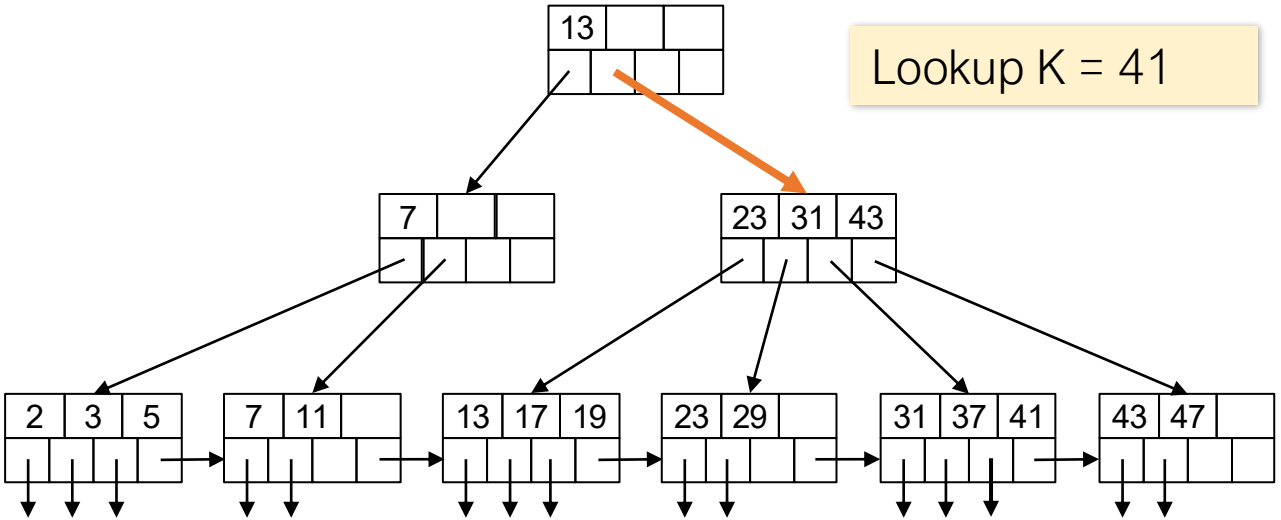
B+ Tree: Lookup

- Search for key K recursively



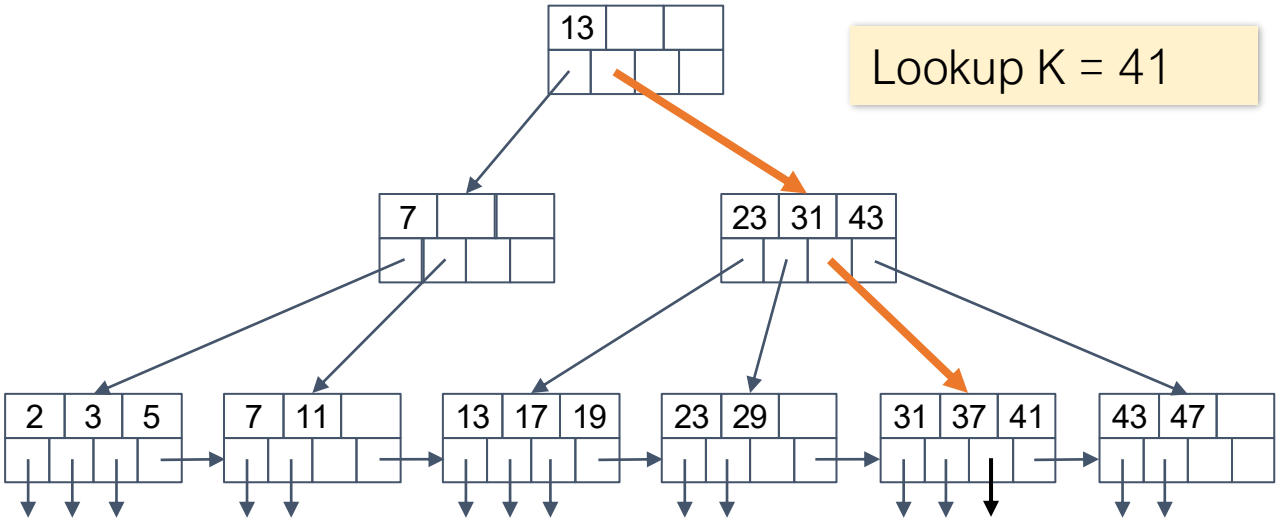
B+ Tree: Lookup

- Search for key K recursively



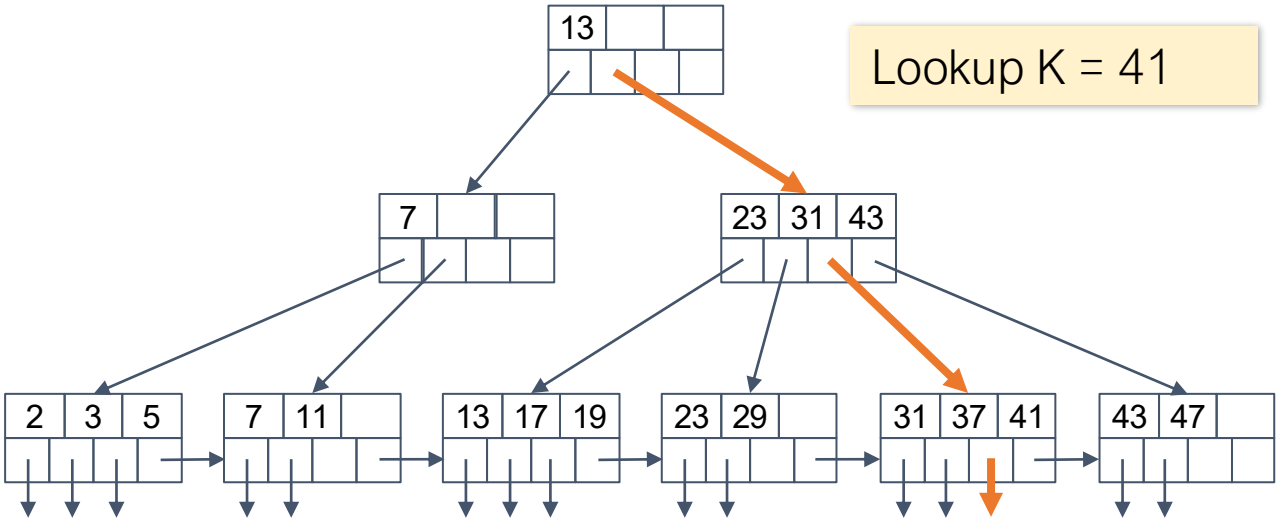
B+ Tree: Lookup

- Search for key K recursively



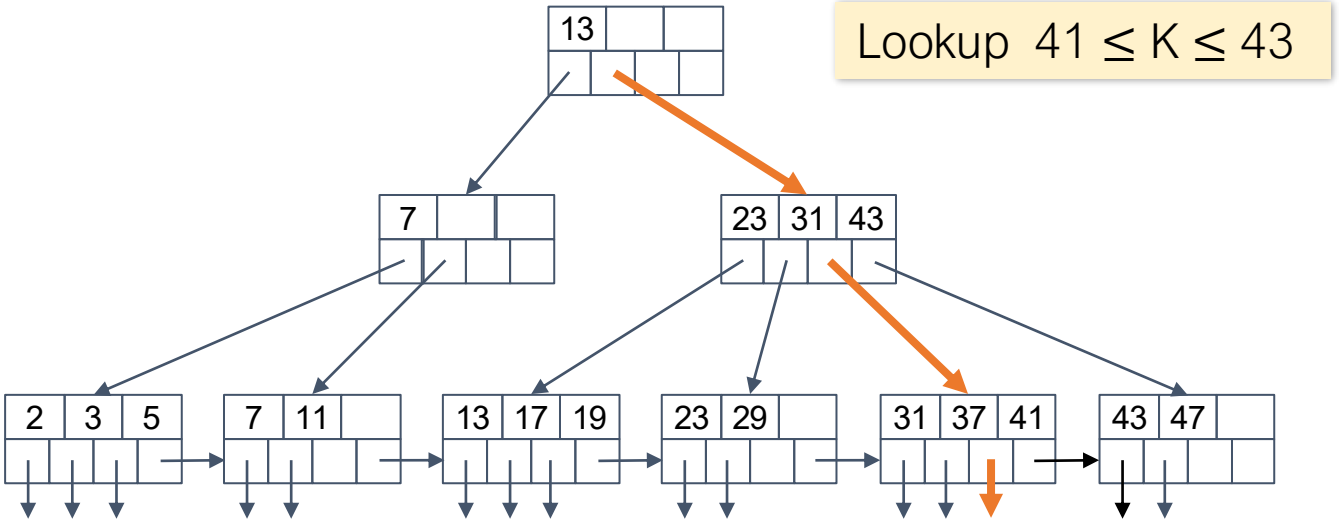
B+ Tree: Lookup

- Search for key K recursively



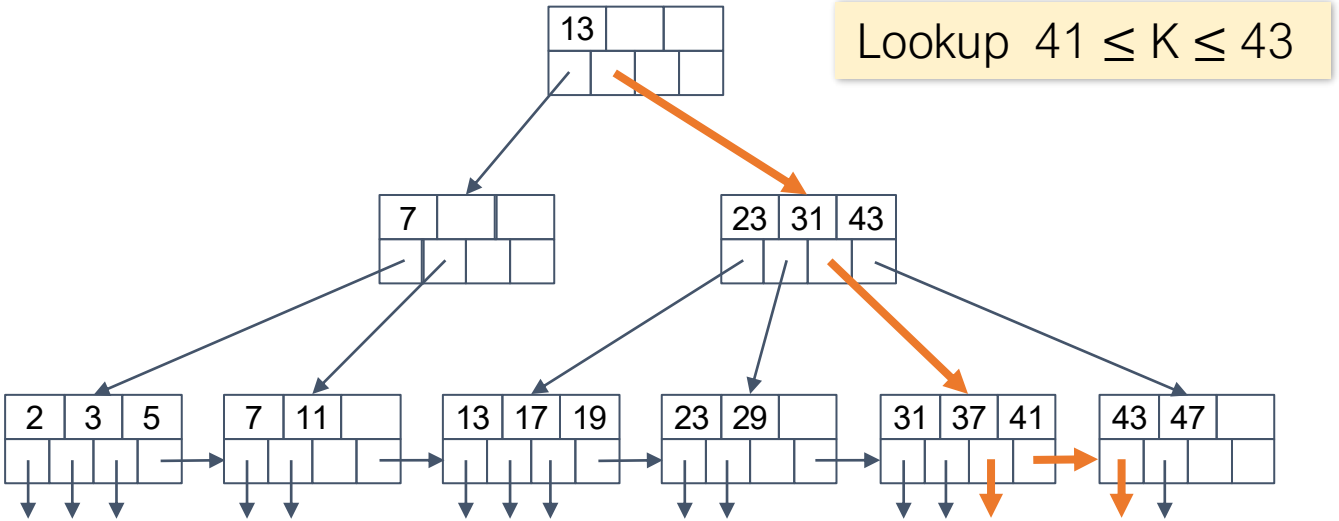
B+ Tree: Lookup

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



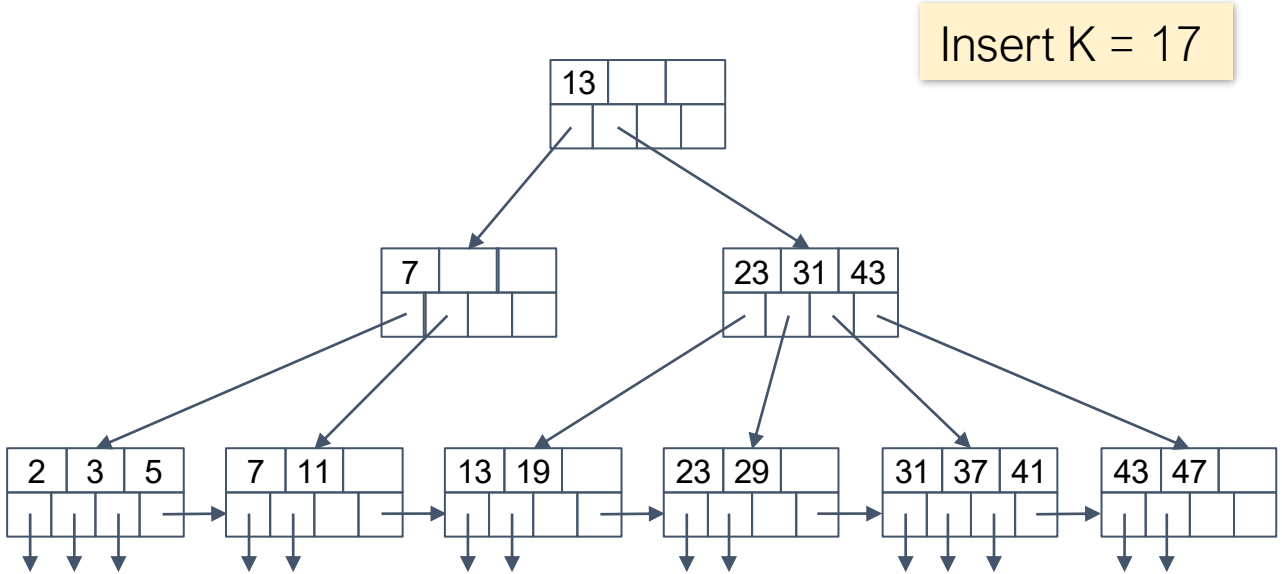
B+ Tree: Lookup

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



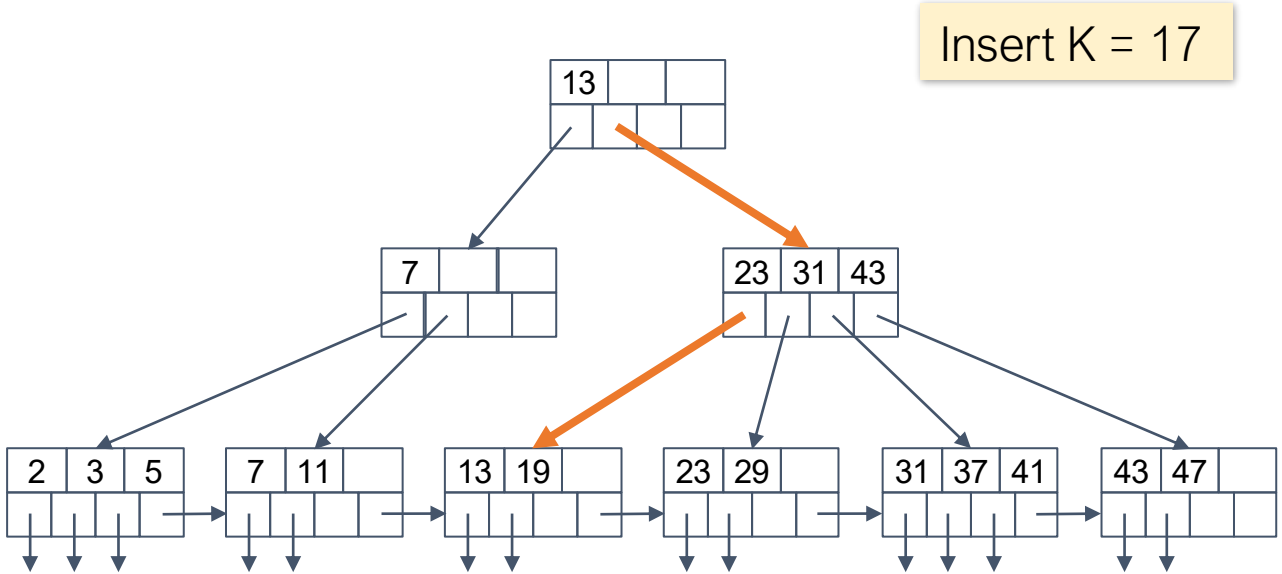
B+ Tree: Insertion

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



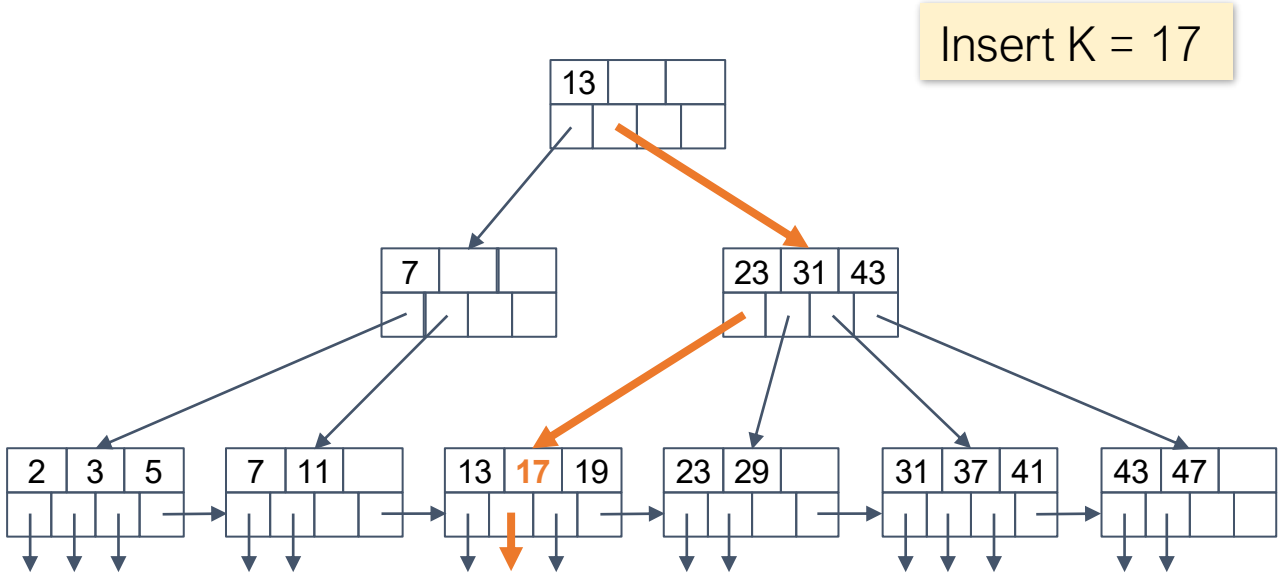
B+ Tree: Insertion

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



B+ Tree: Insertion

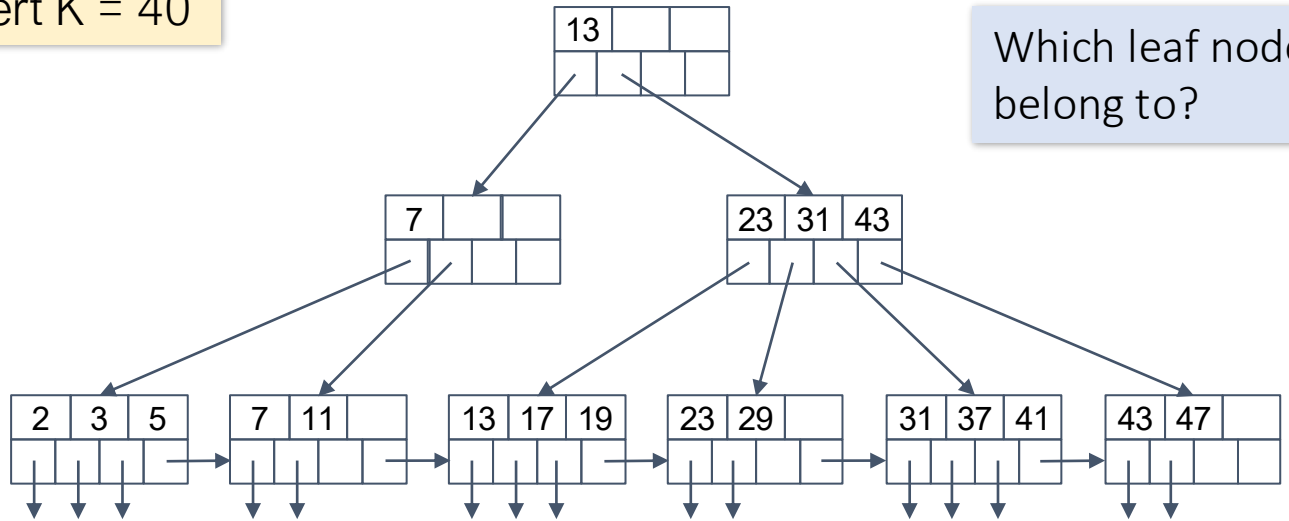
- Find place for new key in a leaf
- If there is space, put key in leaf



B+ Tree: Insertion

- A more complex insertion example:

Insert K = 40

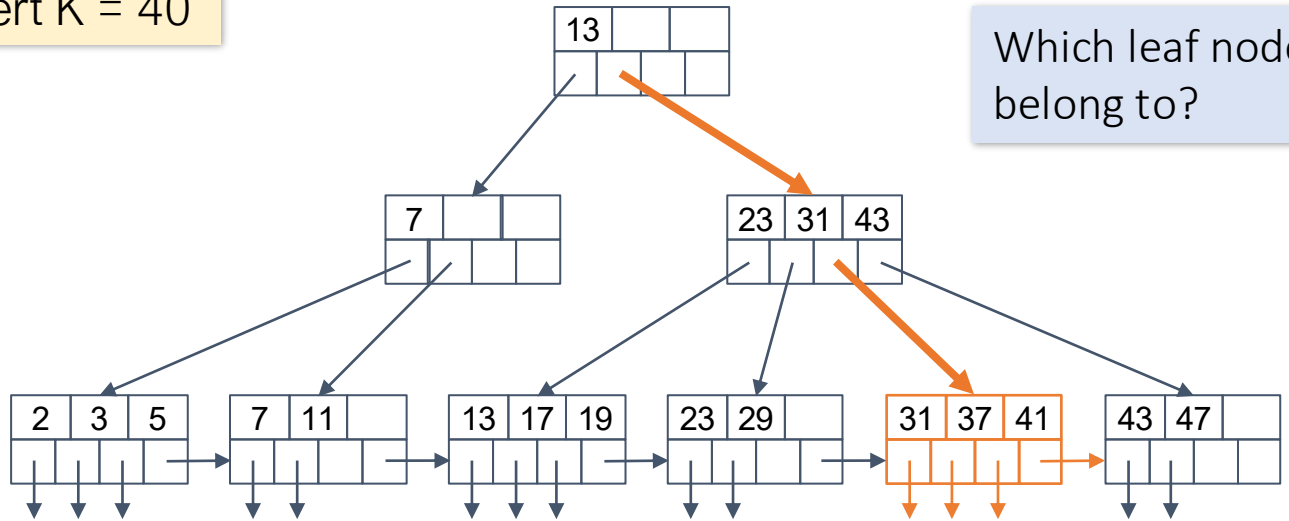


Which leaf node does K belong to?

B+ Tree: Insertion

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

Insert K = 40

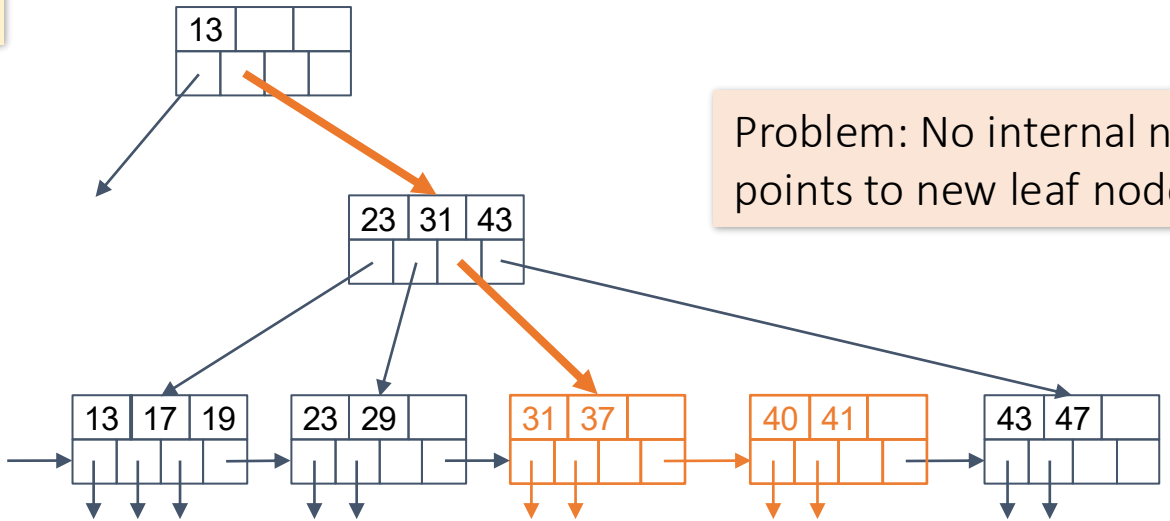


Which leaf node does K belong to?

B+ Tree: Insertion

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

Insert K = 40

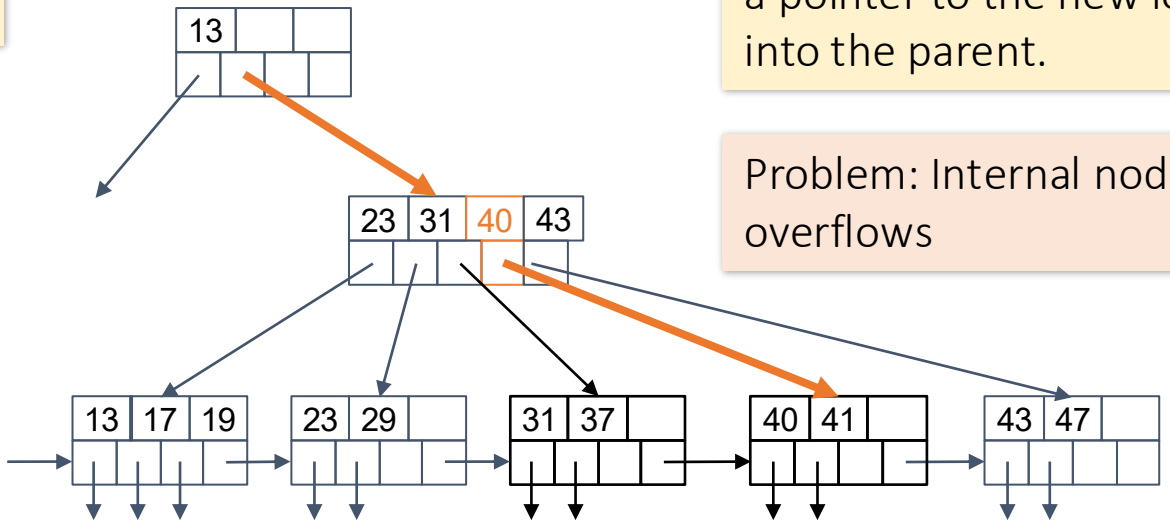


Problem: No internal node points to new leaf node

B+ Tree: Insertion

- If insertion overflows a node, split it and propagate a separator key and child pointer to the parent. Repeat if necessary.

Insert K = 40



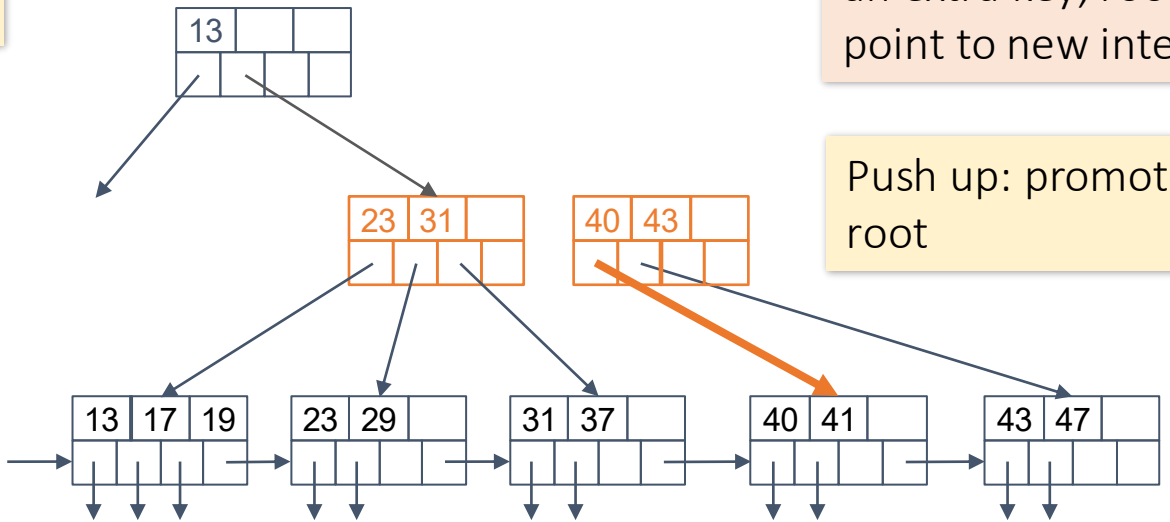
Copy up: Insert key 40 and a pointer to the new leaf into the parent.

Problem: Internal node overflows

B+ Tree: Insertion

- If insertion overflows a node, split it and propagate a separator key and child pointer to the parent. Repeat if necessary.

Insert K = 40



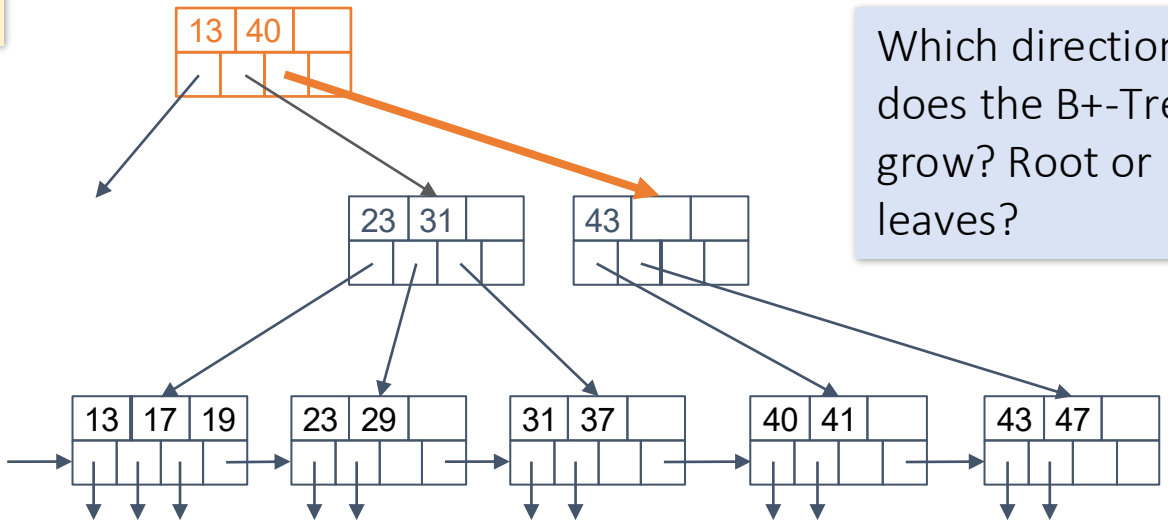
Problem: Internal node has an extra key; root does not point to new internal node

Push up: promote 40 to root

B+ Tree: Insertion

- If insertion overflows a node, split it and propagate a separator key and child pointer to the parent. Repeat if necessary.

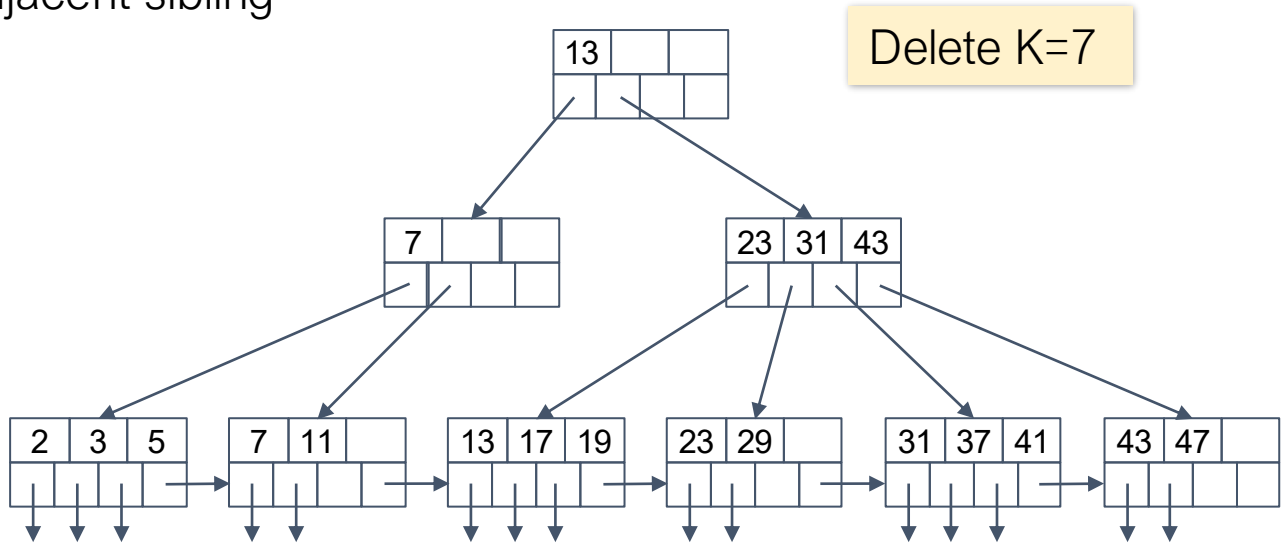
Insert K = 40



Which direction does the B+-Tree grow? Root or leaves?

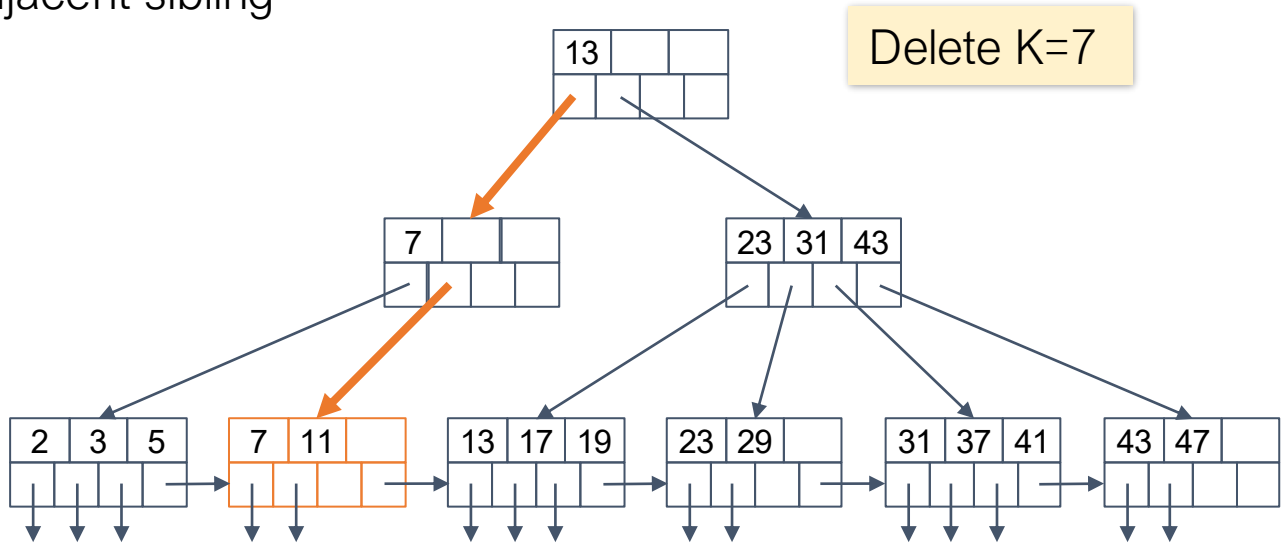
B+ Tree: Deletion

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



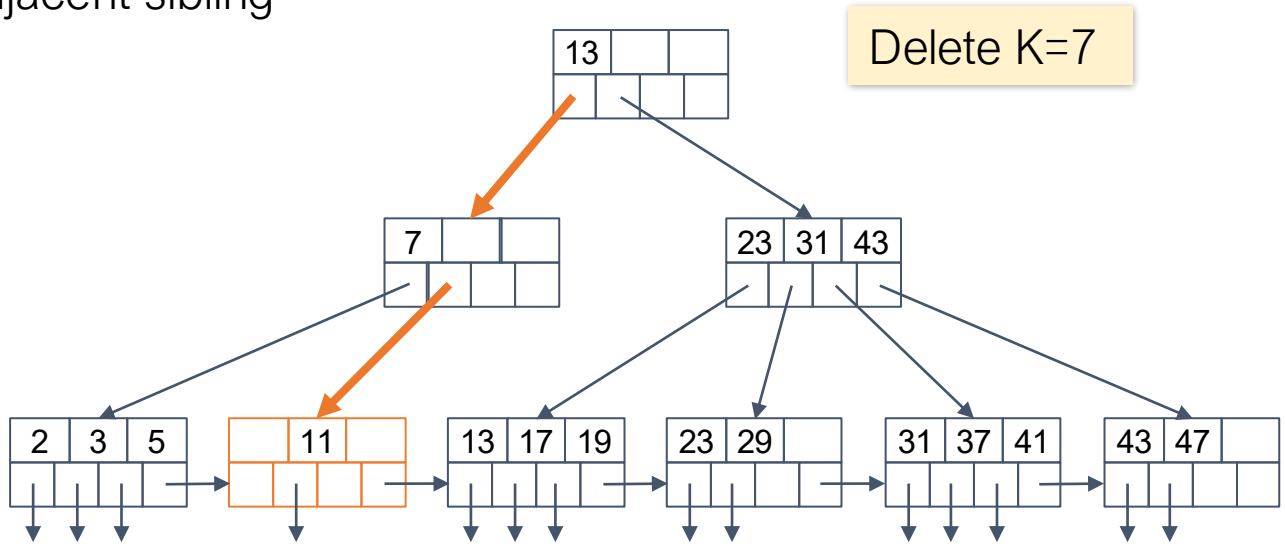
B+ Tree: Deletion

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



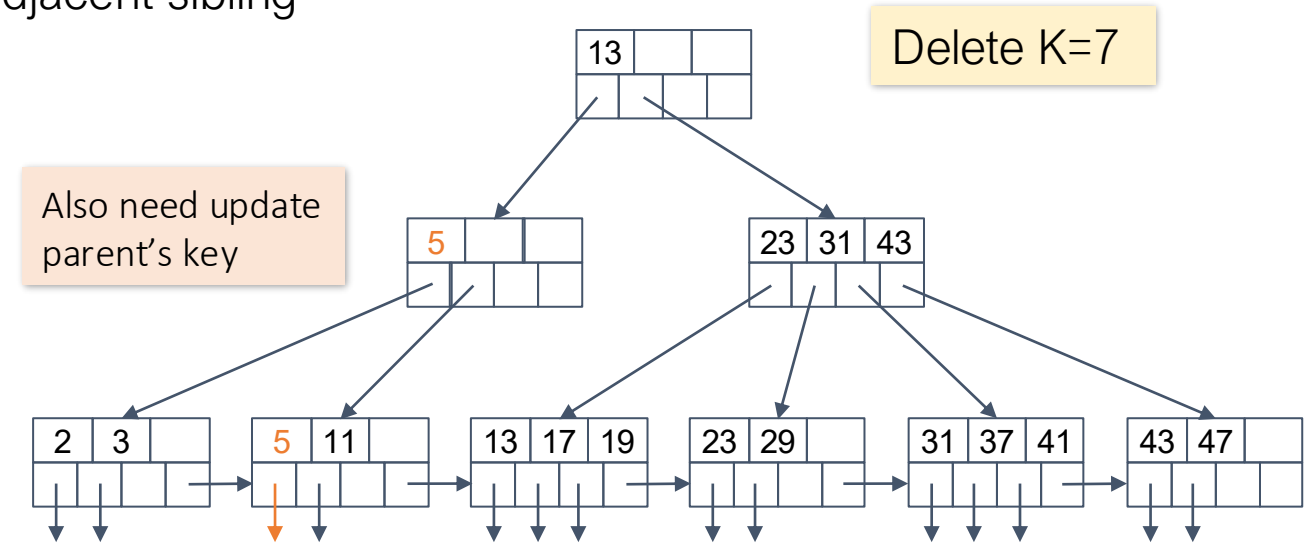
B+ Tree: Deletion

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



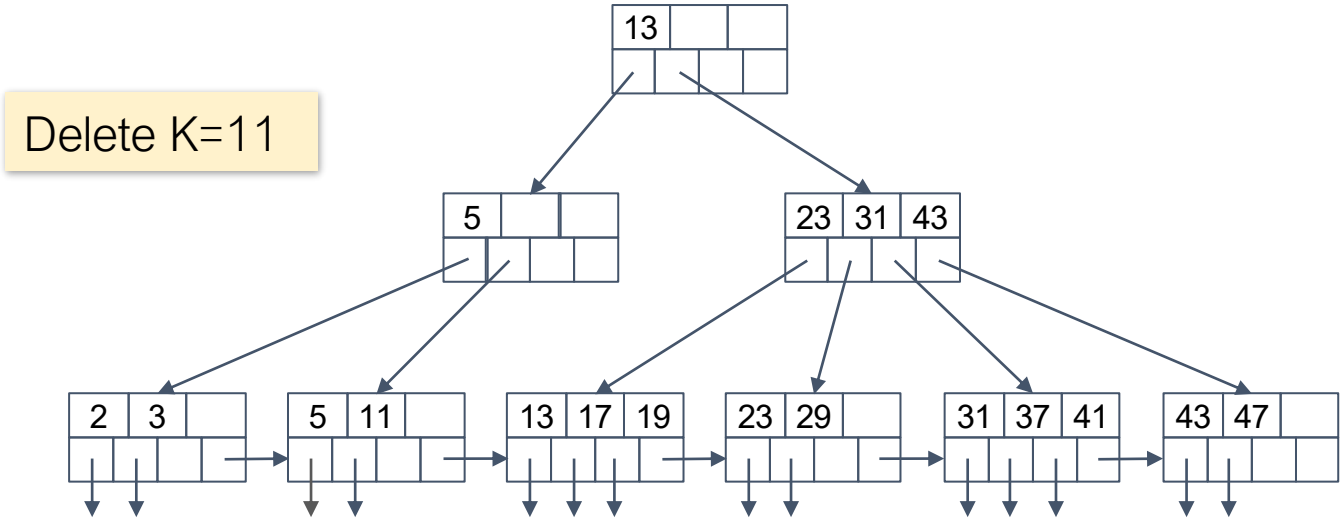
B+ Tree: Deletion

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



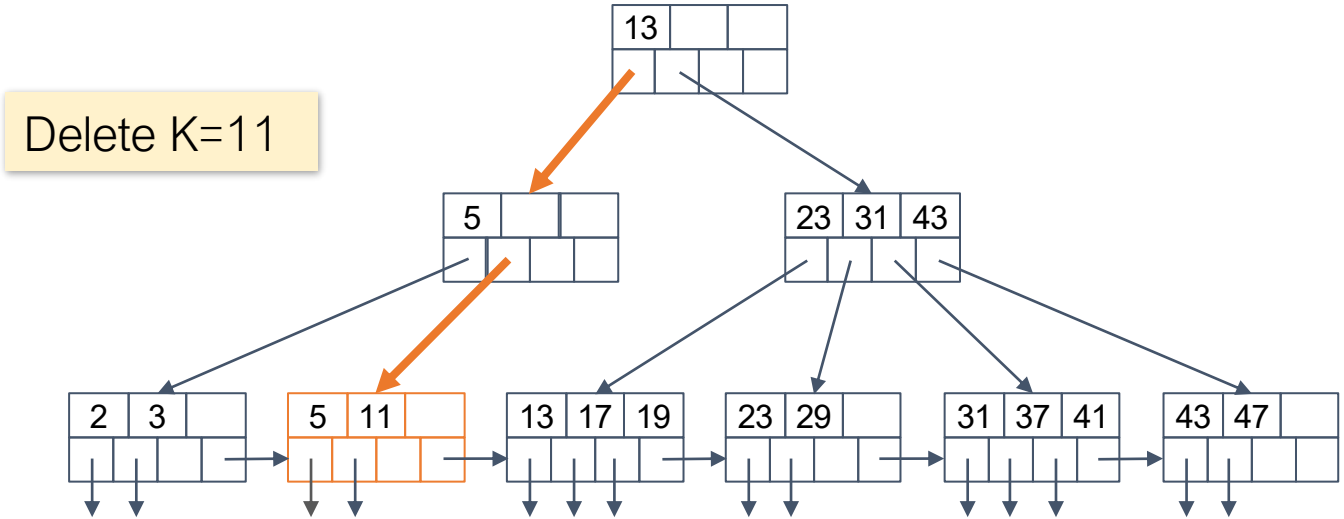
B+ Tree: Deletion

- A more complex deletion example:



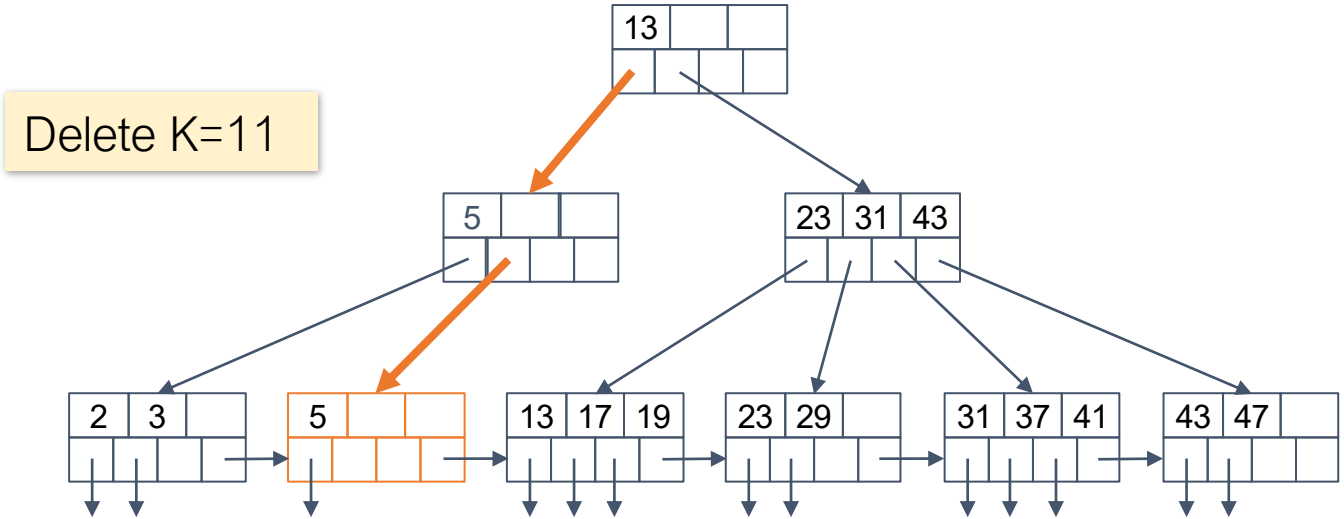
B+ Tree: Deletion

- A more complex deletion example:



B+ Tree: Deletion

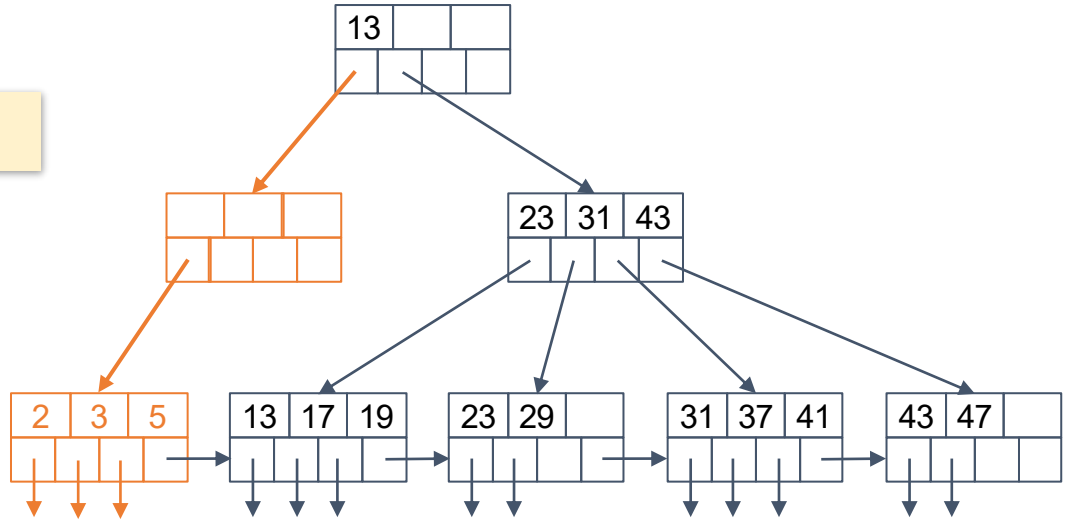
- A more complex deletion example:



B+ Tree: Deletion

- A more complex deletion example:

Delete K=11



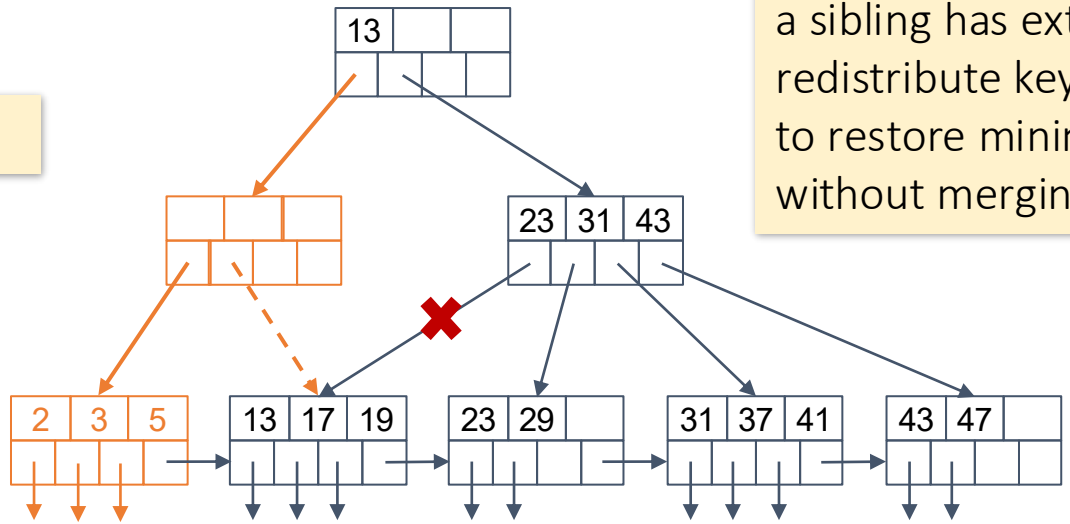
B+ Tree: Deletion

- A more complex deletion example:

What are remaining problems that we need to fix in the tree?

Rotation (redistribution): When a sibling has extra entries, redistribute keys and pointers to restore minimum occupancy without merging nodes.

Delete K=11

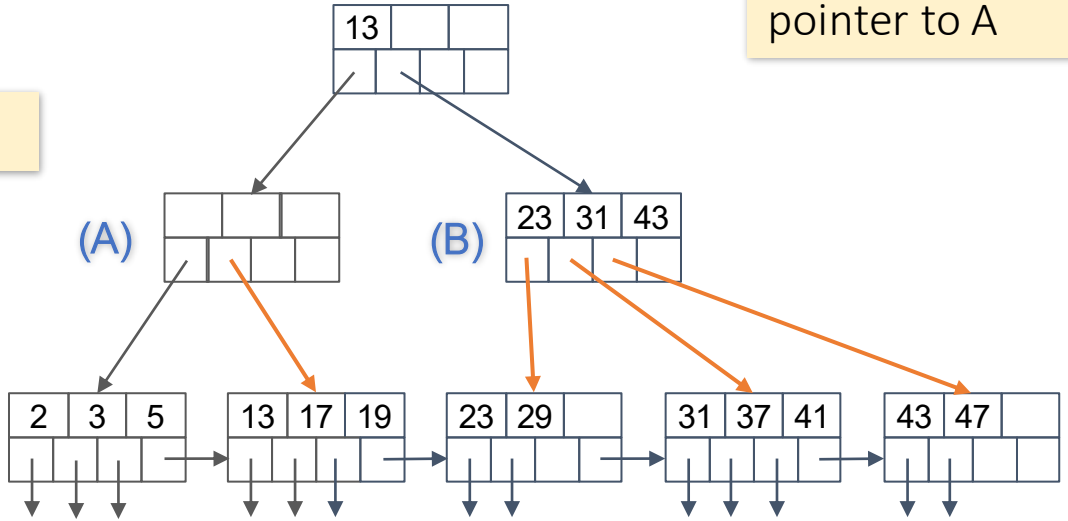


B+ Tree: Deletion

How to update keys:

Move B's leftmost child pointer to A

Delete K=11



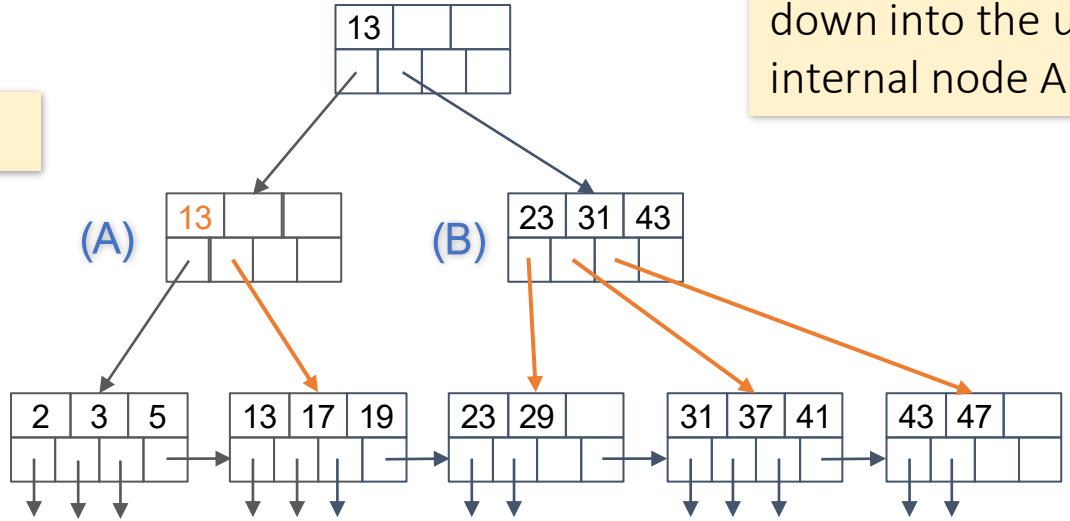
B+ Tree: Deletion

How to update keys:

- Parent key moves down to underflow node (A)

Delete K=11

Move the parent key (13) down into the underfull internal node A



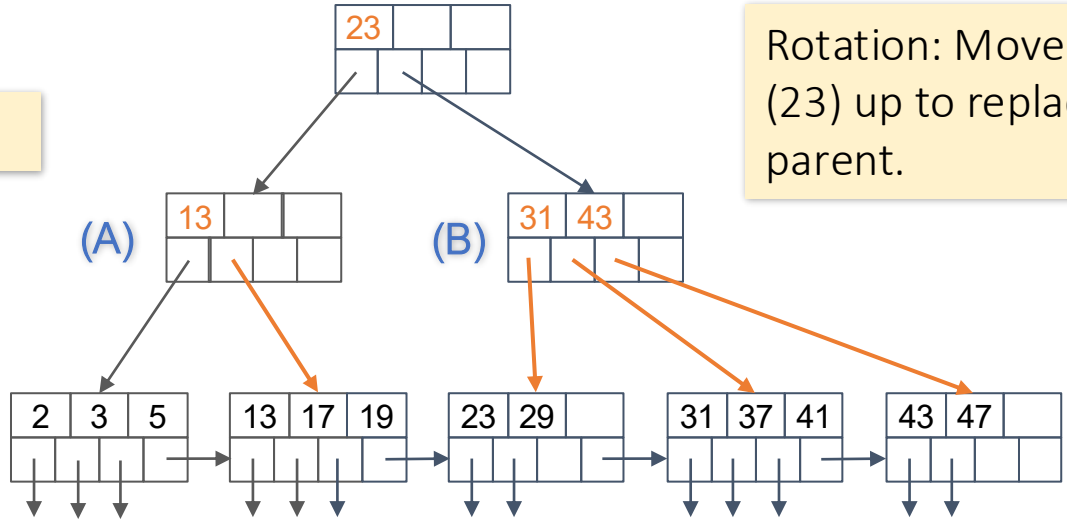
B+ Tree: Deletion

In practice, coalescing is sometimes not implemented because 1) it is hard to implement and 2) the tree will probably grow again.

How to update keys:

- Parent key moves down to underflow node (A)
- Smallest key from right sibling (B) moves up to parent

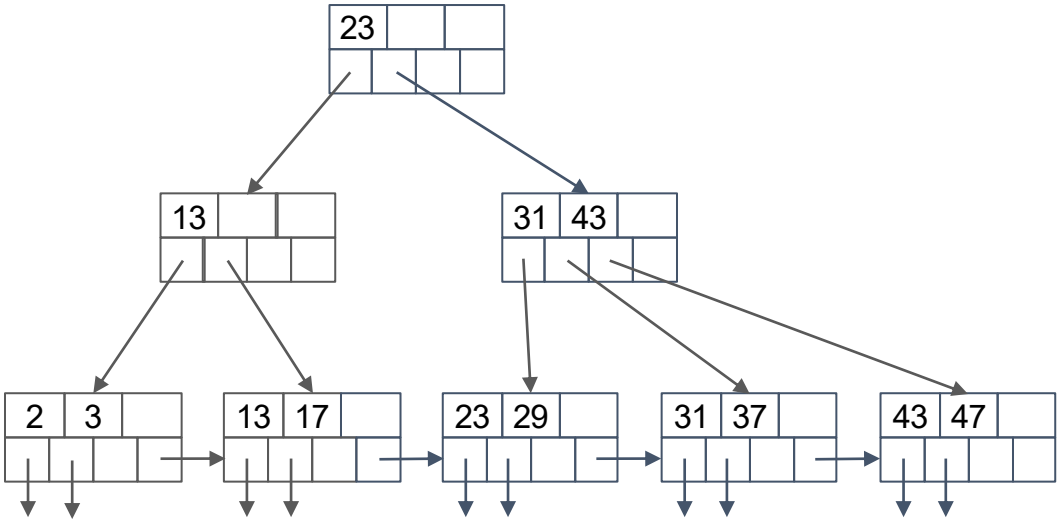
Delete K=11



Rotation: Move B's smallest key (23) up to replace 13 in the parent.

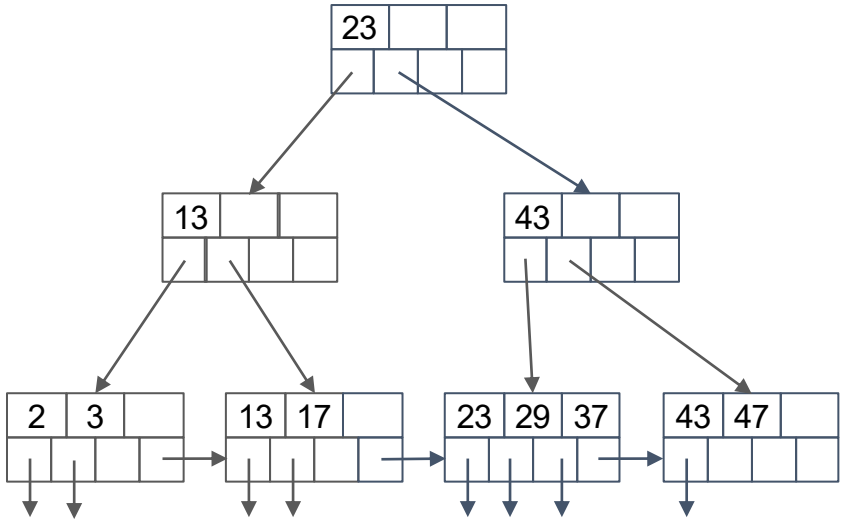
In-class Exercise

- Delete K = 31



In-class Exercise

- Delete K = 31



3. B+-Tree cost model

B+ Tree: High Fanout = Smaller & Lower IO

So why does B+ tree work?

As compared to binary search trees, B+ Trees have **high *fanout***

This means that the **depth of the tree is small** → getting to any element requires very few IO operations!

- Also can often store most or all of the B+ Tree in main memory!

The fanout is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic - we'll often assume it's constant just to come up with approximate eqns!

B+ Trees in Practice

Typical fill-factor: 67%.

- average fanout = 133

Top levels of tree sit *in the buffer pool*:

- Level 1 = 1 page = 8 KB
- Level 2 = 133 pages = 1 MB
- Level 3 = 17,689 pages = 133 MB

Fill-factor is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for **one IO**!

Simple Cost Model for Search

Suppose:

- *f* = average fanout ***(we'll assume it's constant for our cost model…)***
- *N* = the total number of *pages* we need to index
- *F* = fill-factor (usually ~= 2/3)

Our B+ Tree needs to have room to index *N / F* pages!

- We have the fill factor in order to leave some open slots for faster insertions

What height (*h*) does our B+ Tree need to be?

- h=1 → Just the root node- room to index f pages
- h=2 → f leaf nodes- room to index f^2 pages
- h=3 → f^2 leaf nodes- room to index f^3 pages
- …
- h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ Tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$!

Simple Cost Model for Search

Note that if we have ***B*** available buffer pages, by the same logic:

- We can store $\boldsymbol{L_B}$ levels of the B+ Tree in memory
- where $\boldsymbol{L_B}$ *is the number of levels such that the sum of all the levels' nodes fit in the buffer:*
  - $B \geq 1 + f + \cdots + f^{L_B - 1} = \sum_{l=0}^{L_B - 1} f^l$

In summary: to do exact search:

- We read in one page per level of the tree
- However, levels that we can fit in buffer are free!
- Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Simple Cost Model for Search

To do range search, we just follow the horizontal pointers

The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each *page* of the results- we phrase this as “Cost(OUT)”

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

In-class Exercise

Given a B+ tree indexing over $N = 100{,}000$ data pages with fill factor $F = 1$ and constant fanout $\mathrm{f} = 10$. Assume that each node of the B+ tree occupies one page, that there are $B = 11$ buffer pages available to store B+ Tree nodes.

What's IO cost of performing an exact search query on this index?

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$