Lecture 10
2-3 Trees B Trees

Department of Computer Science
Hofstra University

2–3 Trees

The 2-3 tree is a way to generalize BSTs to provide the flexibility that we need to guarantee fast performance

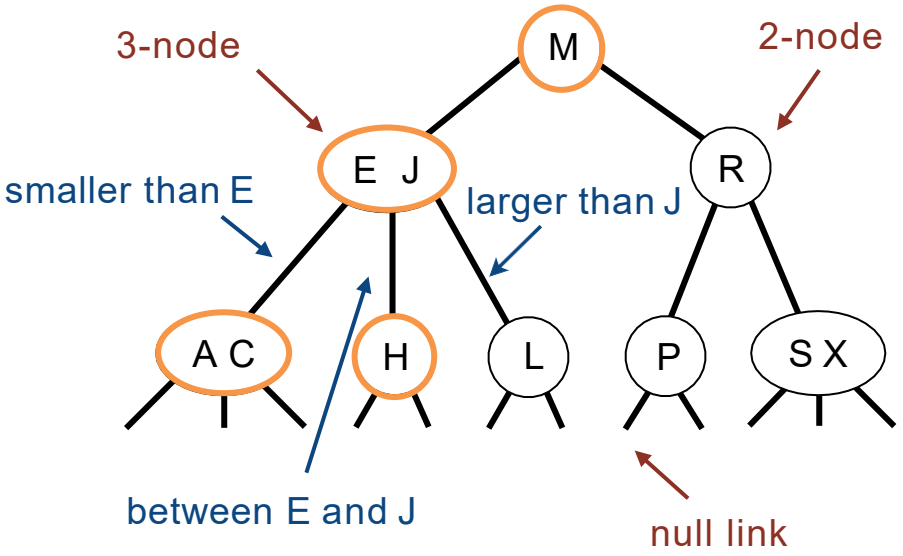
Allow 1 or 2 keys per node
- 2-node: one key, two children.
- 3-node: two keys, three children.

Perfect balance: Every path from root to null link has same length.

how to maintain?

Symmetric order: Inorder traversal yields keys in ascending order.

The search is a generalization of the search in BSTs



Search
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

```
Search for H
```
H is less than M (go left)
H is between E and J (go middle)

Search hit

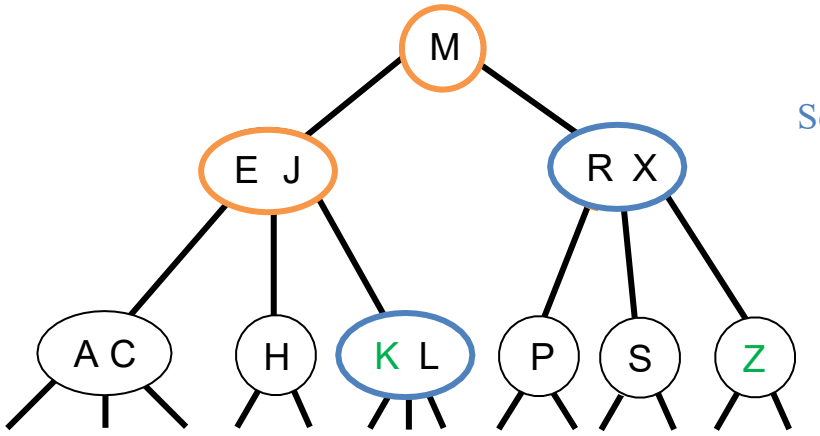
```
Search for B
```
B is less than M (go left)
B is less than E (go left)
B is between A and C (go middle)
Link is null

Search miss

Insert in 2–3 Trees

Insertion into a 2-node at bottom.

- Search for key, as usual
- Add new key to 2-node to create a 3-node.



`Insert K`

K is less than M (go left)

K is larger than J (go right)

K is less than L

Search ends here and replace 2-node with 3-node containing K

`Insert Z`

Z is larger than M (go right)

Z is larger than R (go right)

Z is larger than X

Search ends here

Replace 3-node with temporary 4-node containing Z

Split 4-node into two 2-nodes (pass middle key to parent)

Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

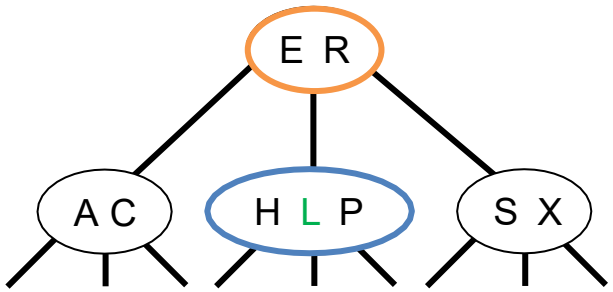
Insert in 2−3 Trees (Contd.)

Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

`Insert L`

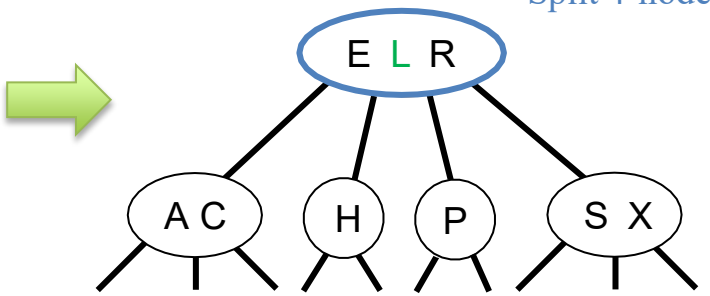
L is between E and R (go middle)



Search ends here

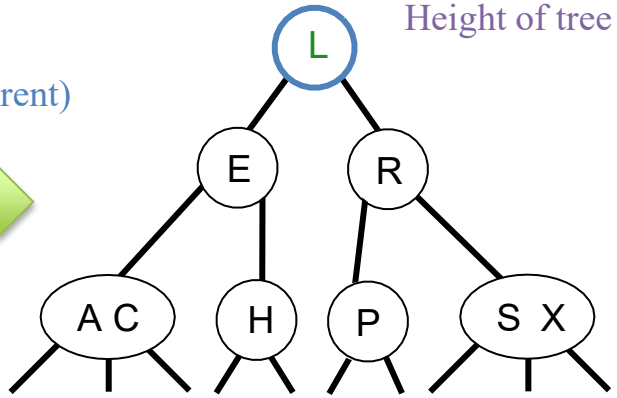
Replace 3-node with temporary 4-node containing L

Split 4-node (move L to parent)



Split 4-node (move L to parent)

Height of tree increases by 1



2–3 Trees Construction

`Insert S` Create 2-node in the empty tree

`Insert E` Convert 2-node into 3-node

`Insert A` Convert 3-node into 4-node

Split 4-node into two 2-nodes (move E to parent)

`Insert R` Convert 2-node into 3-node

`Insert C` Convert 2-node into 3-node

`Insert H` Convert 3-node into 4-node

Split 4-node into two 2-nodes (move R to parent)

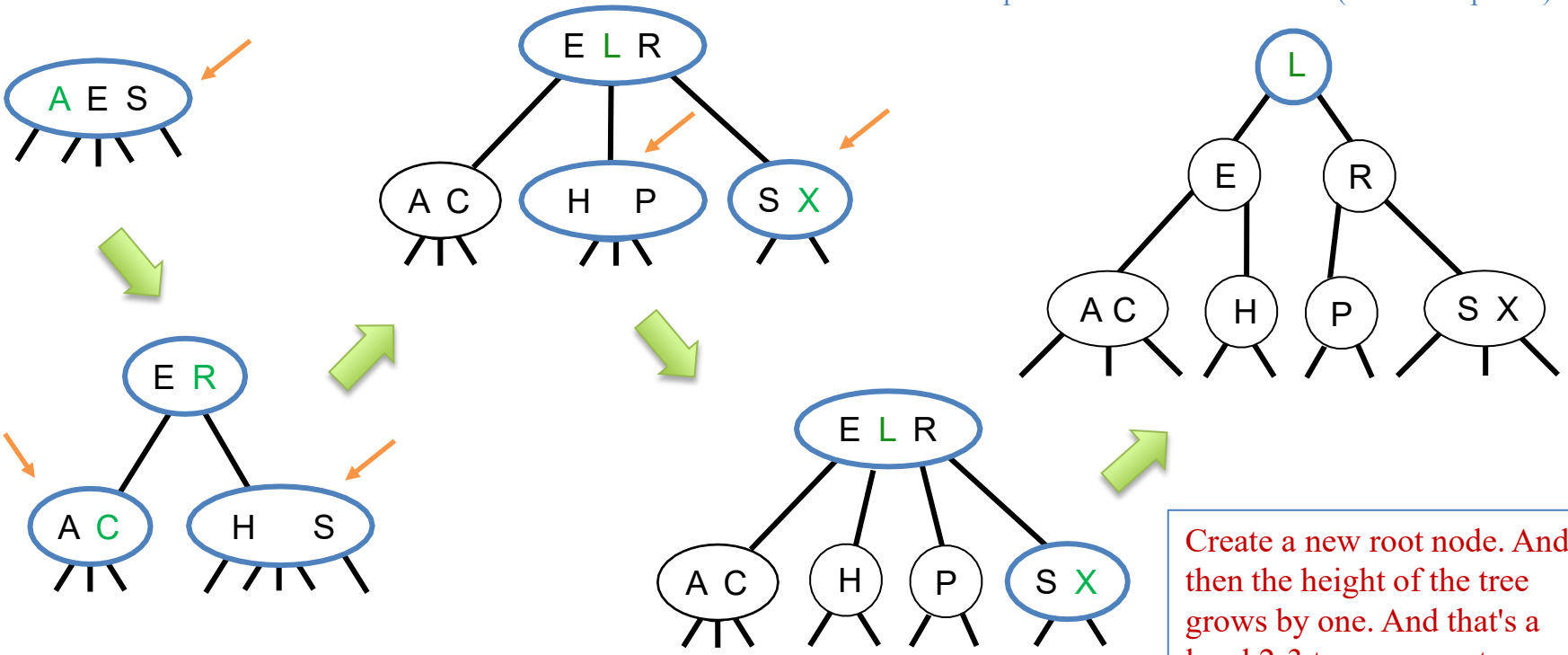
`Insert X` Convert 2-node into 3-node

`Insert P` Convert 2-node into 3-node

`Insert L` Convert 3-node into 4-node

Split 4-node into two 2-nodes (move L to parent)

Split 4-node into two 2-nodes (move L to parent)

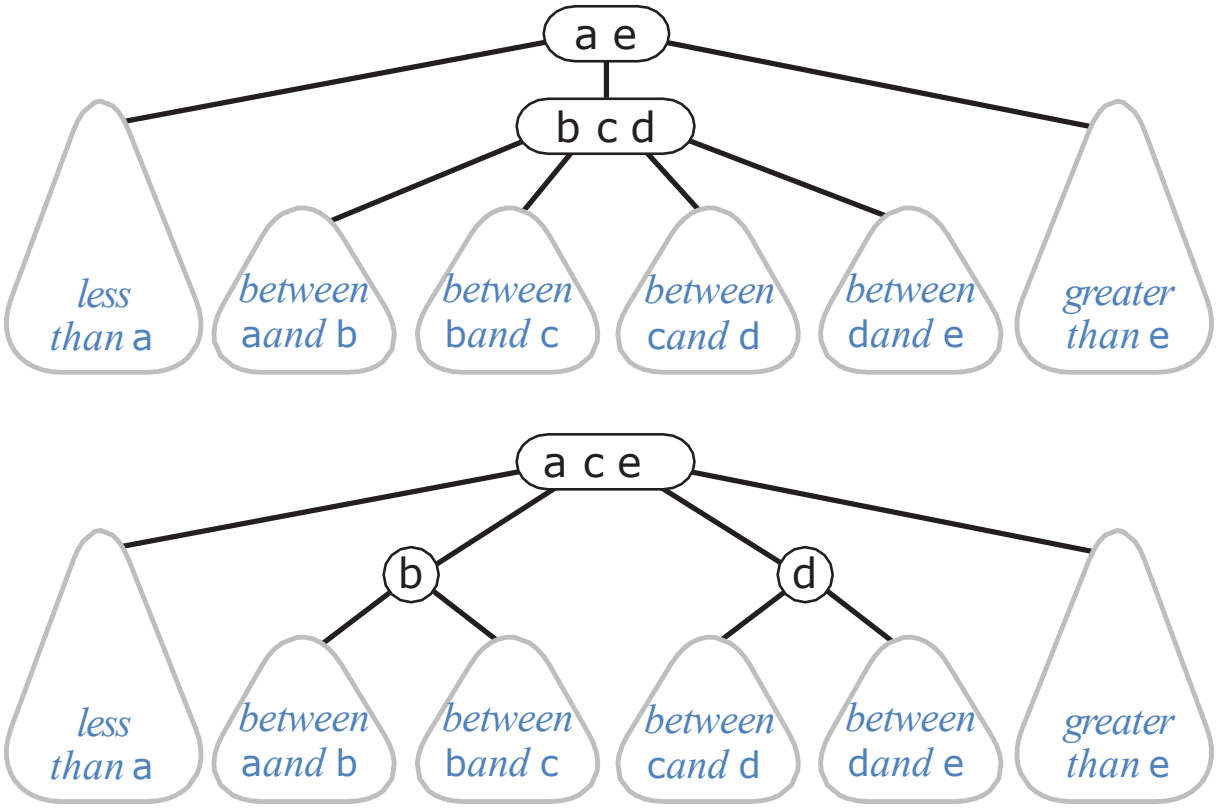


Create a new root node. And then the height of the tree grows by one. And that's a legal 2-3 tree, so we stop.

Local Transformations in a 2-3 Tree

Converting a 2-node to a 3-node

Converting a three to a four, and then splitting and passing a node up



These operations maintain tree balance.

local transformation: constant number of operations.

Only involves changing a constant number of links, and is independent of the tree size.

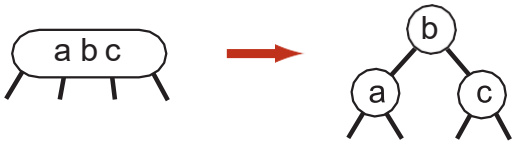
Global Properties in a 2-3 Tree

Invariants. Maintains symmetric order and perfect balance.

Proof. Each transformation maintains symmetric order and perfect balance.

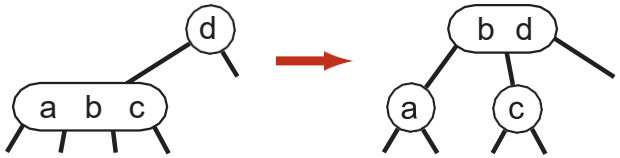
Splitting a 4-node

root

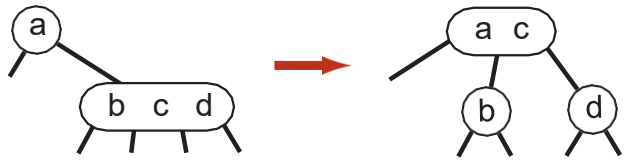


parent is a 2-node: split and parent turns into a 3-node

left

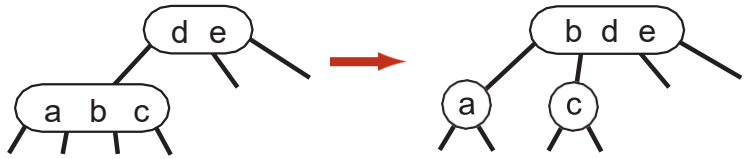


right

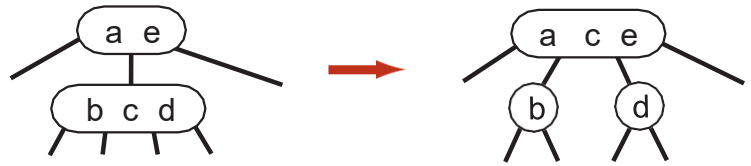


parent is a 3-node: split and parent turns into a 4-node; parent splits again (omitted in figures)

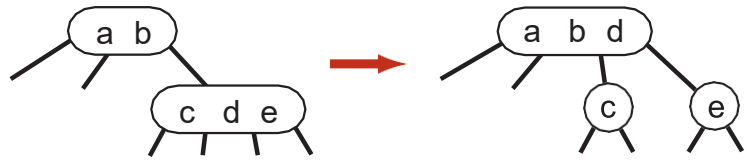
left



middle



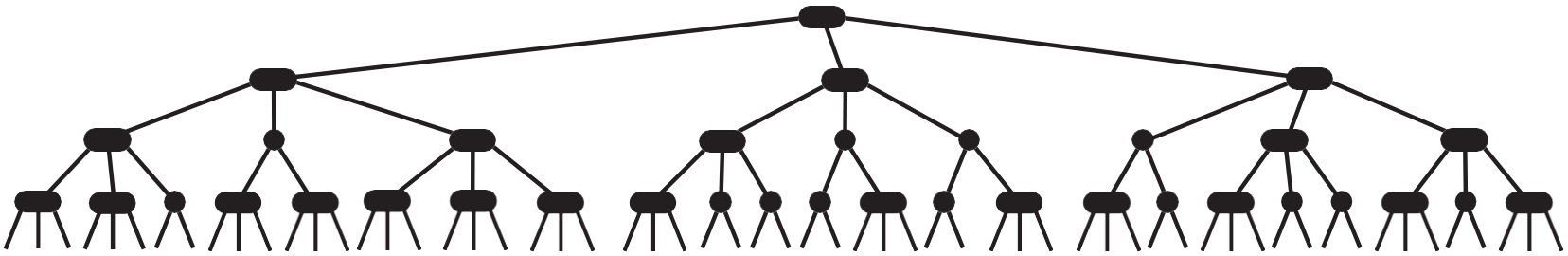
right



If it is perfect balance before, it is perfect balance afterwards

Performance of 2-3 Tree

Perfect balance: Every path from root to null link has same length.



So, that's going to give us, a very easy way to describe a performance.

Operations have costs that's proportional to the path link from root to the bottom.

Tree height

- Worst case: $\log_2 n$. [all 2-nodes]
- Best case: $\log_3 n \approx 0.631 \log 2 n$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

	Best case	Average case	Worst case
BST	O(1)	O(log n)	O(n)
2-3 Tree	O(1)	O(log n)	O(log n)

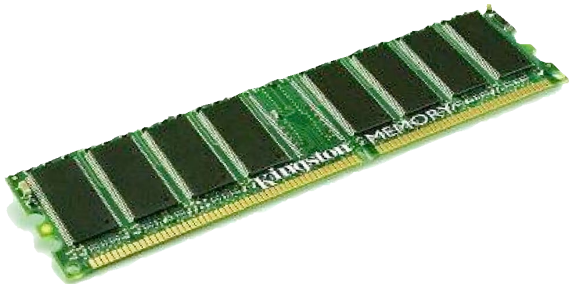
2-3 tree model guaranteed logarithmic performance for search and insert.

Motivation for B-tree: File system model

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).

slow



fast

Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

Goal. Access data using minimum number of probes.

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory, hence must be stored on disk
- Time required to access disk is much larger than time to access main memory
- Goal: access data with minimum number of probes.

Motivation (cont.)

- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses, e.g., $\log_2 20{,}000{,}000$ is about 24
- We know we cannot improve on the log *n* lower bound on search for a binary tree
- But we can use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

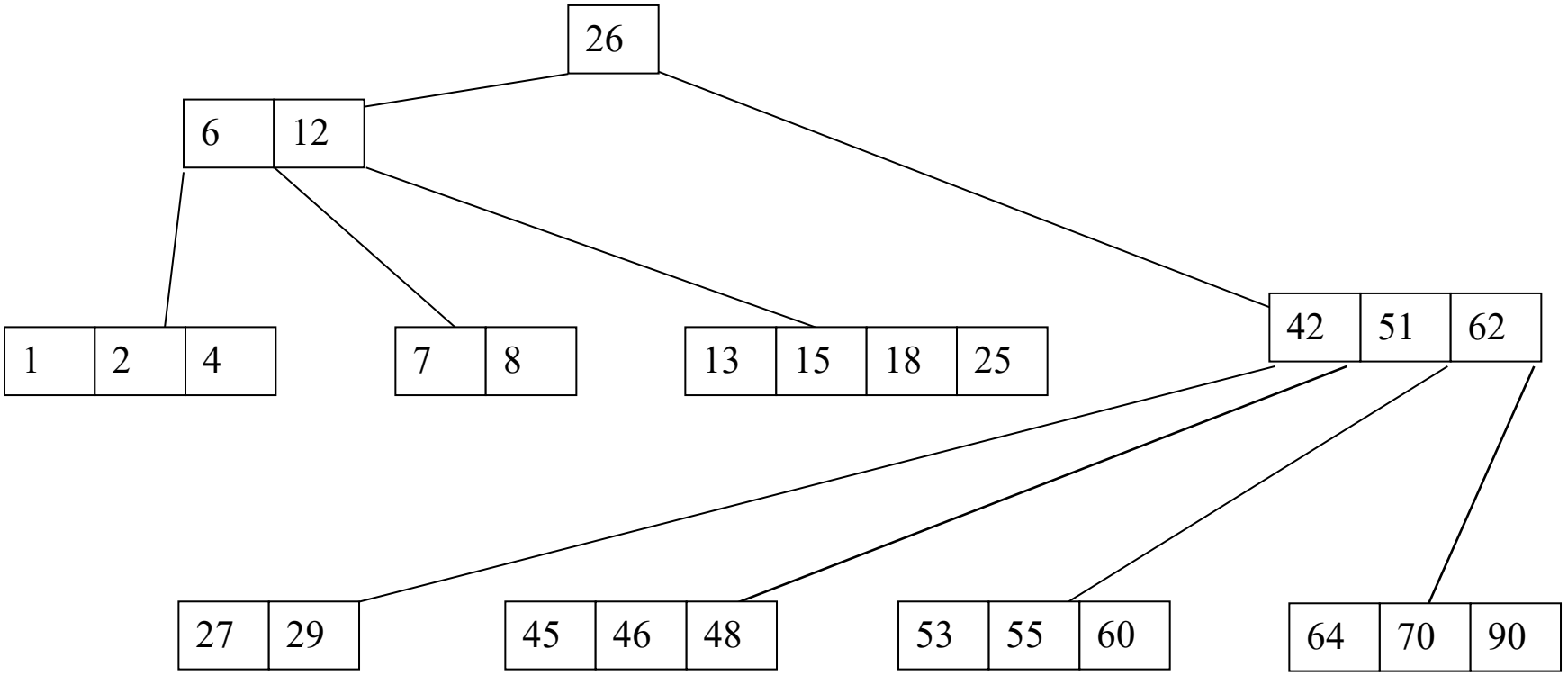
Definition of a B-tree

- A B-tree of order M is an M-way tree (i.e., a tree where each node may have up to M children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children, and these keys partition the keys in the children in sorted order (i.e., a search tree)
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil M / 2 \rceil$ children (i.e., must be at least "half-full")
  4. the root has at least 2 children (unless it is a leaf node)
  5. a leaf node contains no more than $M - 1$ keys

B-trees Example I

Generalization of 2-3 trees

- At least 2 children at root
- At least $\lceil M / 2 \rceil$ children in other nodes



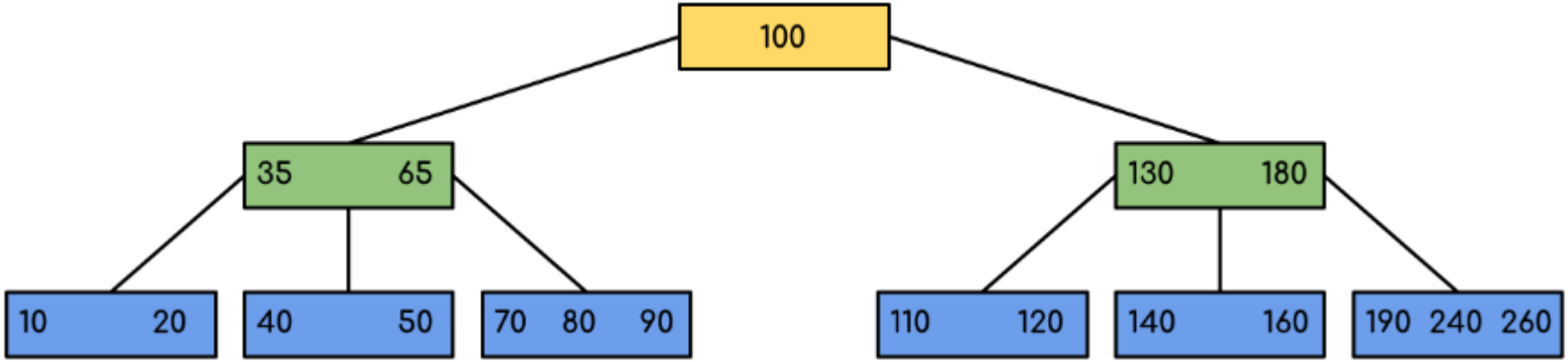
A B-tree of order 5 (M = 5)

All leaves are at the same level 3

B-trees Example II

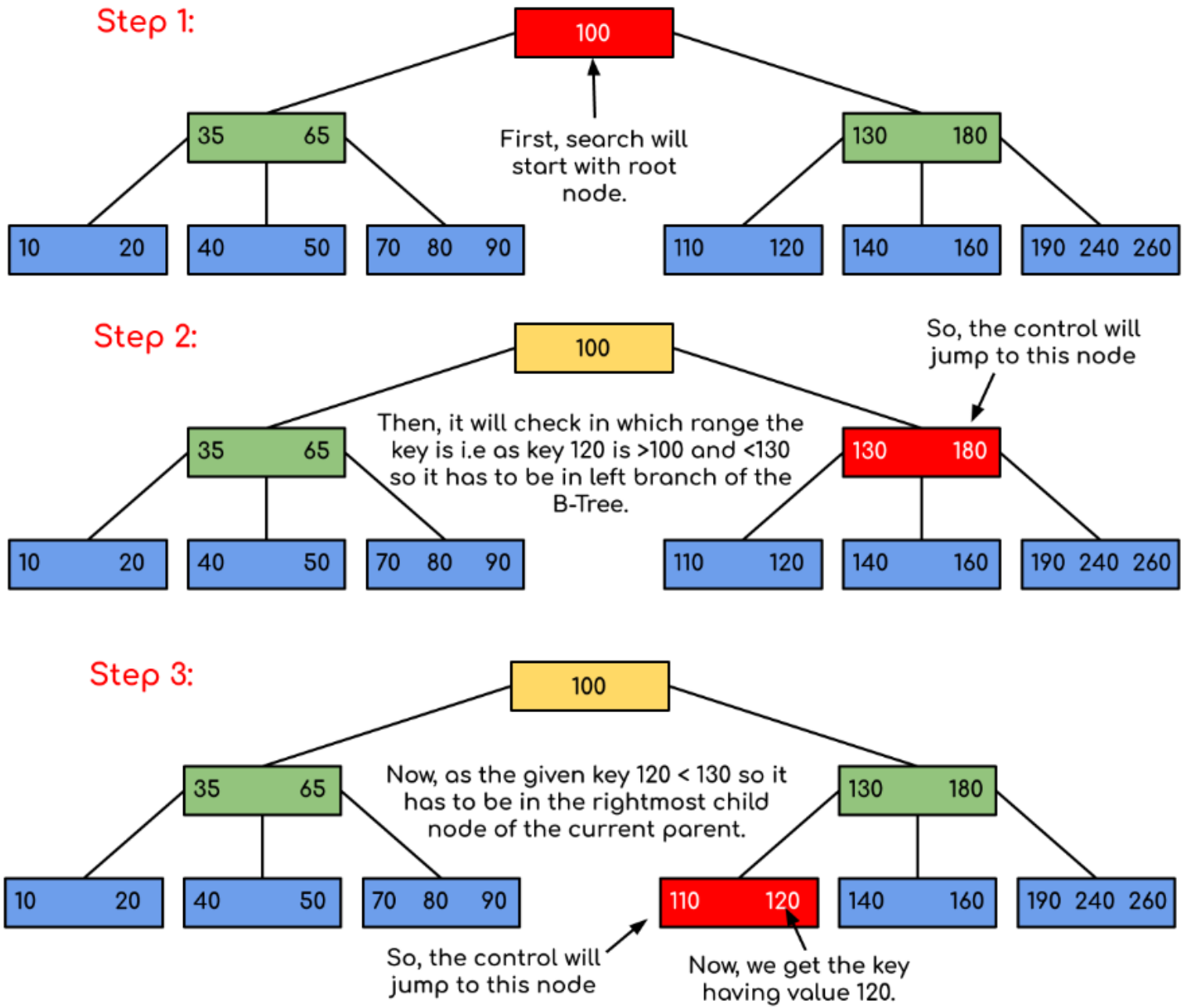
Generalization of 2-3 trees

- At least 2 children at root
- At least $\lceil M / 2 \rceil$ children in other nodes



A B-tree of order 4 (M = 4)

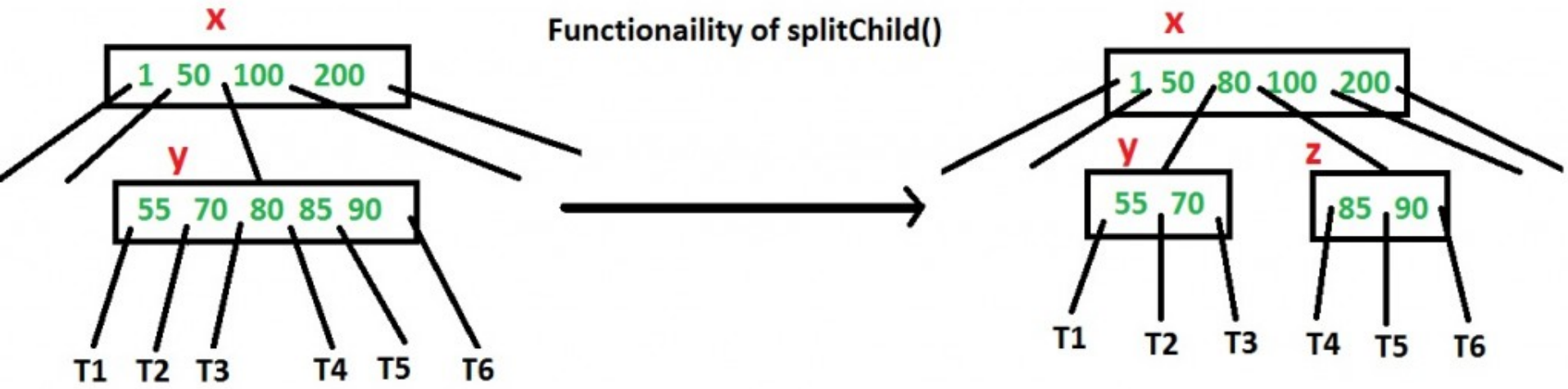
All leaves are at the same level 3



Searching for 120 in the given B-Tree

Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher



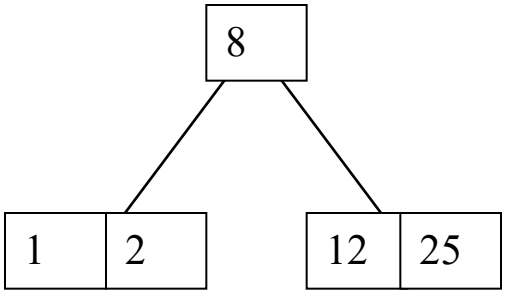
Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- The first four items go into the root:

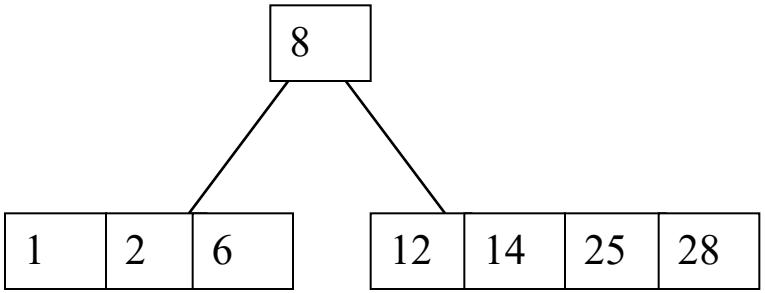
1	2	8	12

- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, promote the middle key to make a new root

Constructing a B-tree (contd.)

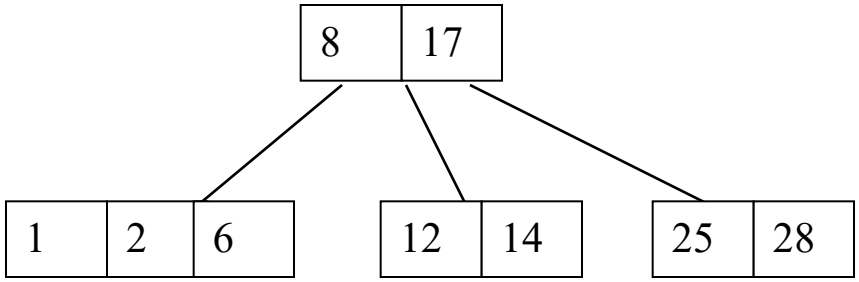


6, 14, 28 get added to the leaf nodes:

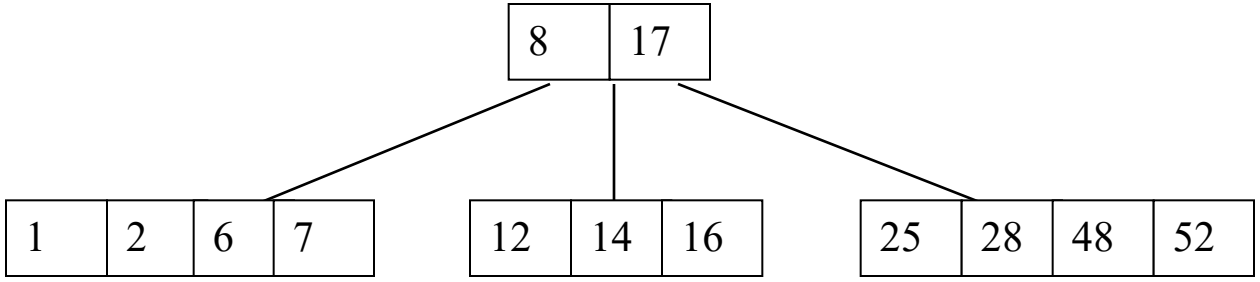


Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

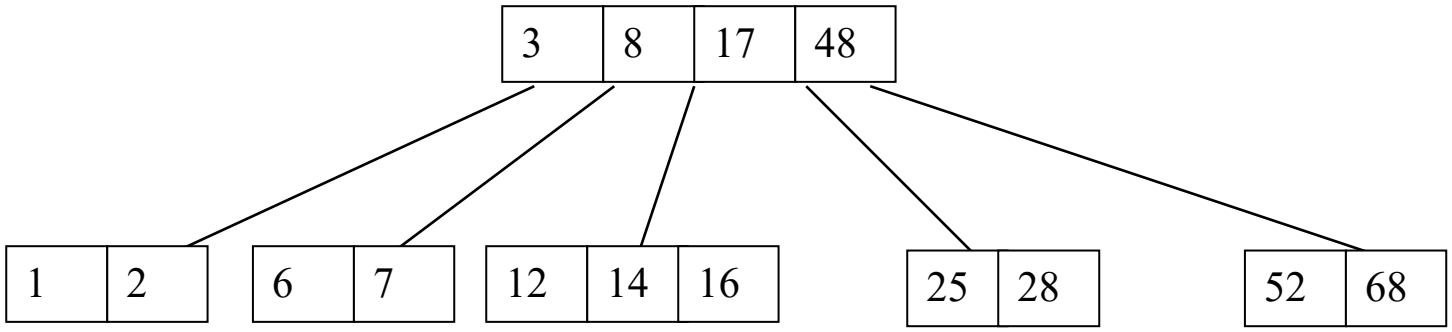


7, 52, 16, 48 get added to the leaf nodes

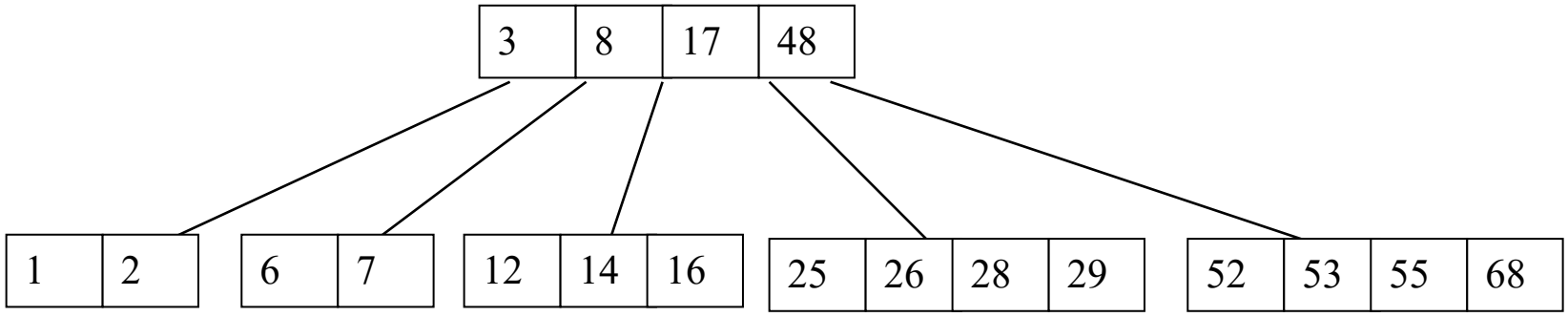


Constructing a B-tree (contd.)

Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root



26, 29, 53, 55 then go into the leaves

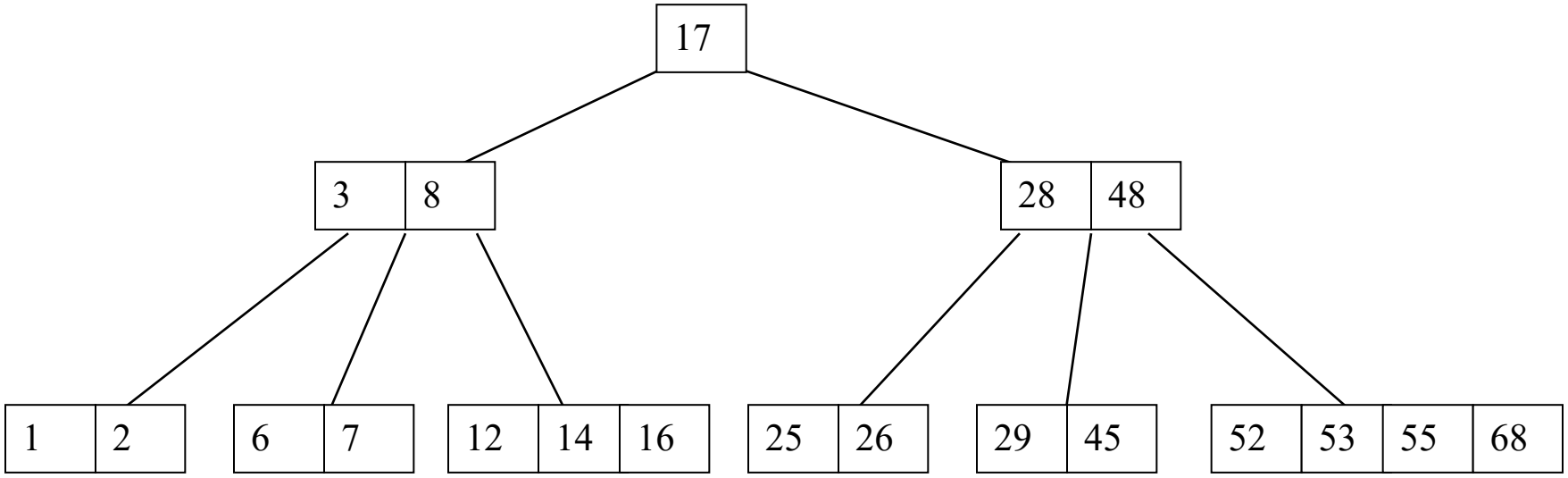


Constructing a B-tree (contd.)

Adding 45 causes a split of

25	26	28	29

and promoting 28 to the root then causes the root to split



Deletion from a B-tree

- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Explanations: why a key's predecessor or successor is guaranteed to be in a leaf

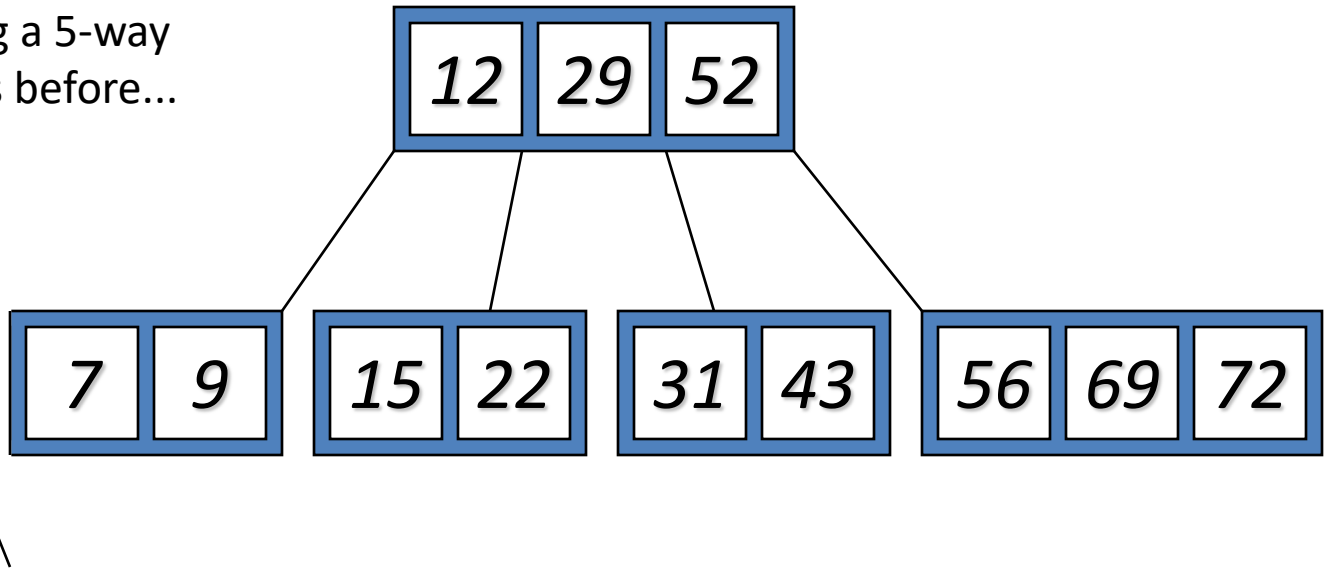
- Definition of Predecessor and Successor:
  - The predecessor of a key in an internal node is the largest key in its left subtree.
  - The successor is the smallest key in its right subtree
- Traversal to Leaves:
  - To find the predecessor, you follow the rightmost path from the left child of the key's node. Similarly, to find the successor, you follow the leftmost path from the right child. These paths always terminate at a leaf because B-trees are structured such that all keys in any subtree are stored in its leaves or internal nodes, and traversal through children eventually reaches leaves
- Key Replacement During Deletion:
  - When deleting a key from an internal node, it is replaced by either its predecessor or successor. This ensures that the B-tree remains balanced and satisfies its properties. Since predecessors and successors are located in leaf nodes, their removal does not disrupt the tree's structure beyond localized adjustments
- Ref. Slide 28, "Deletion from a BST" in Lecture 8-Binary Search Tree.

Deletion from a B-tree (2)

- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we look at the siblings immediately adjacent to the leaf in question:
  - 3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - 4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

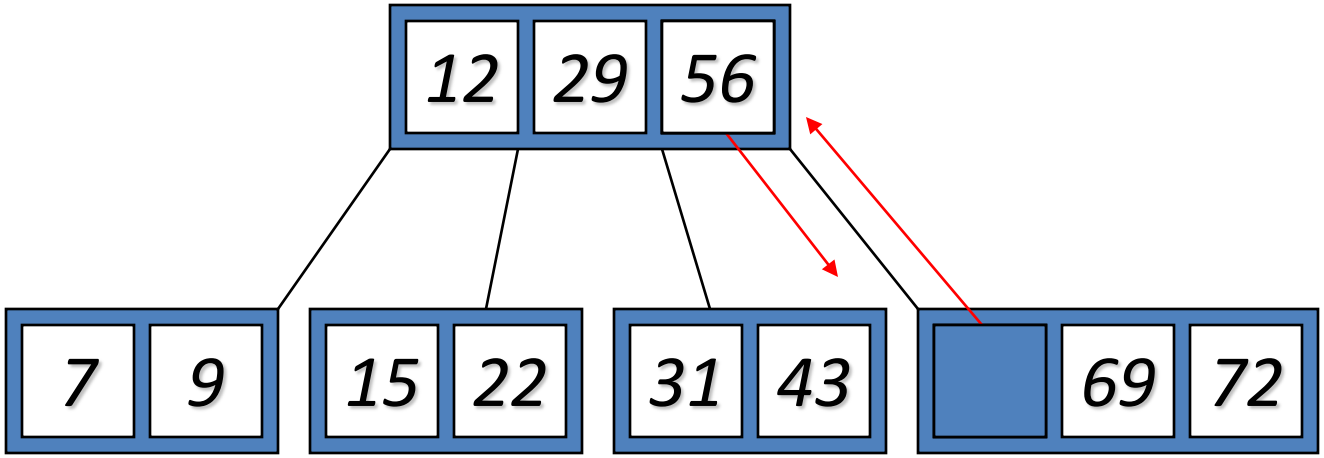
Type #1: Simple leaf deletion

Assuming a 5-way B-Tree, as before...

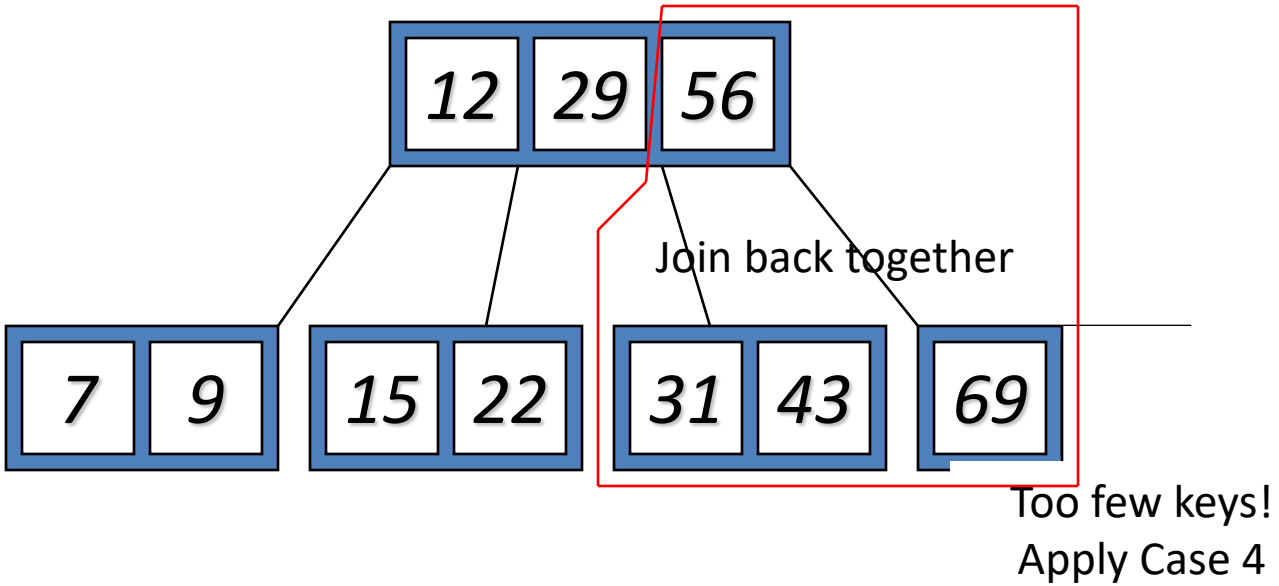


Delete 2: Since there are enough keys in the node, just delete it

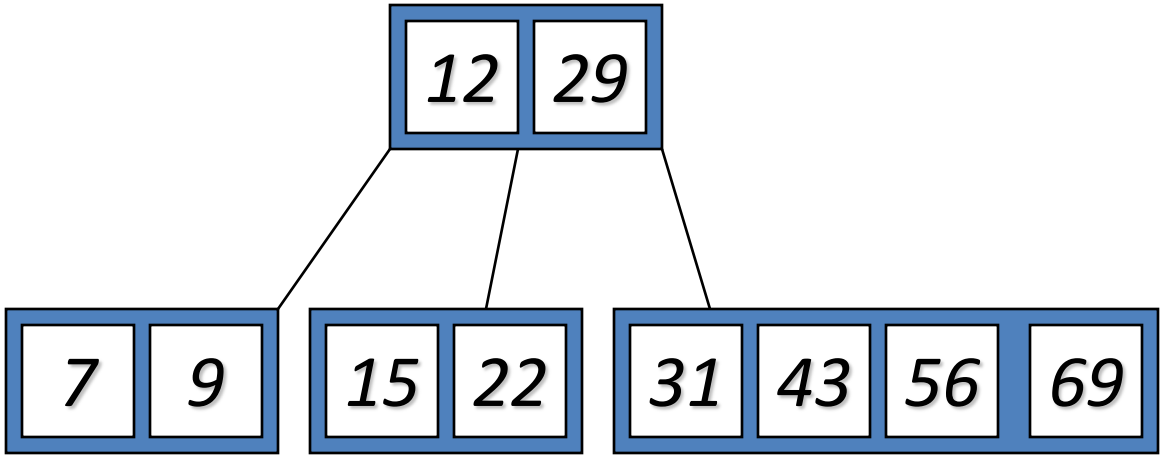
Type #2: Simple non-leaf deletion



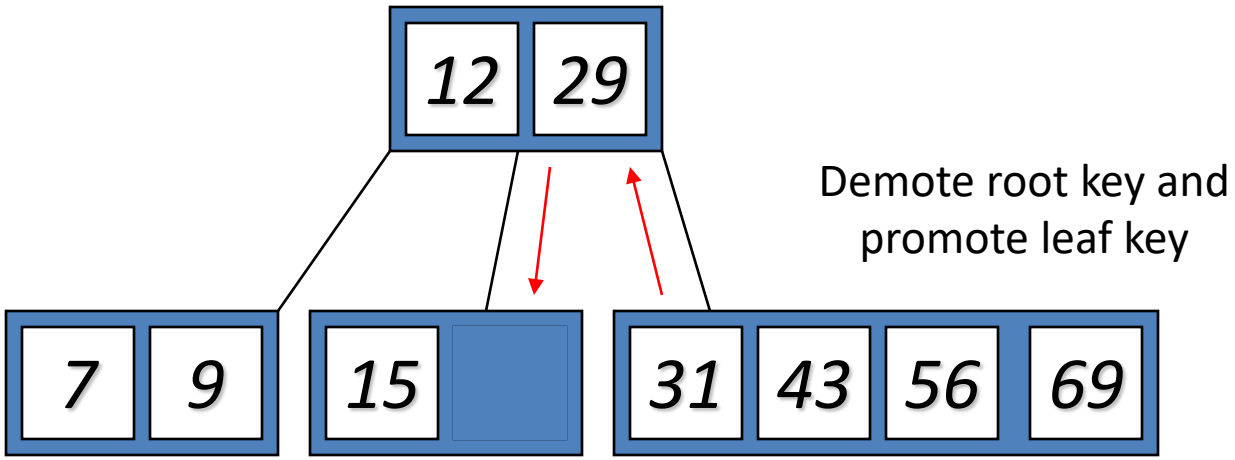
Type #4: Too few keys in node and its siblings



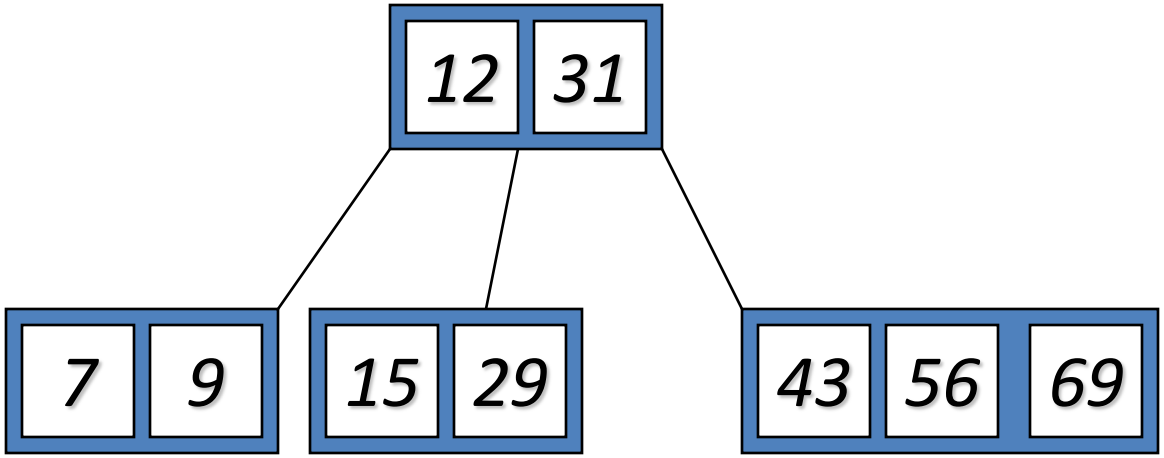
Type #4: Too few keys in node and its siblings



Type #3: Enough siblings



Type #3: Enough siblings



Analysis of B-Trees

- The maximum number of items in a B-tree of order m and height h:

  root $m - 1$

  level 1 $m(m - 1)$

  level 2 $m^2(m - 1)$

  . . .

  level h $m^h(m - 1)$

- So, the total number of items is

  $$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$

  $$[(m^{h+1} - 1)/ (m - 1)] (m - 1) = \boldsymbol{m^{h+1} - 1}$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Reasons for using B-Trees

- The cost of each disk read is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred and they fit within a page
  - e.g., a B-tree of order 101 and height 3 can hold $101^4 - 1$ items (about 100 million) and any item can be accessed with 3 disk reads (assuming each node fits within one memory page)
- If we take $m = 3$, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree

Comparing Trees

- Binary trees
  - Can become *unbalanced* and *lose* their good time complexity (big O)
  - AVL trees are strict binary trees that *overcome the balance problem*
  - Heaps remain balanced but only *prioritise* (not order) the keys

- Multi-way trees
  - B-Trees can be *m*-way, with at most *m* children
  - 2-3 tree is 3-way B-Tree. It *approximates* a permanently balanced binary tree, exchanging the AVL tree's balancing operations for insertion and (more complex) deletion operations

Balanced Trees in the Wild

- Red–black trees are widely used as system symbol tables.
  - Java: java.util.TreeMap, java.util.TreeSet.
  - C++ STL: map, multimap, multiset.
  - Linux kernel: completely fair scheduler, linux/rbtree.h.
  - Emacs: conservative stack scanning.
- B-tree variants. B+ tree, B*tree, B# tree, …
- B-trees (and variants) are widely used for file systems and databases.
  - Windows: NTFS.
  - Mac: HFS, HFS+.
  - Linux: ReiserFS, XFS, Ext3FS, JFS, BTRFS.
  - Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.

References

- The Most Elegant Search Structure | (a,b)-trees, Tom S
  - https://www.youtube.com/watch?v=lifFgyB77zc
- Lecture - 13 Trees
  - https://www.youtube.com/watch?v=JZhdUb5F7oY
- Lecture - 15 Insertion in Red Black Trees (talks about 2-3 trees)
  - https://www.youtube.com/watch?v=6QOKk_pcv3U
- Lecture - 16 Disk Based Data Structures (talks about B trees)
  - https://www.youtube.com/watch?v=VbVroFR4mq4
- B Tree tutorials
  - https://spetriuk.github.io/algorithms/B-Tree.%201.%20Introduction/
  - https://spetriuk.github.io/algorithms/B-Tree.%202.%20Insert%20Operation/
  - https://spetriuk.github.io/algorithms/B-Tree.%203.%20Delete%20Operation/