Lecture 10
2-3 Trees B Trees

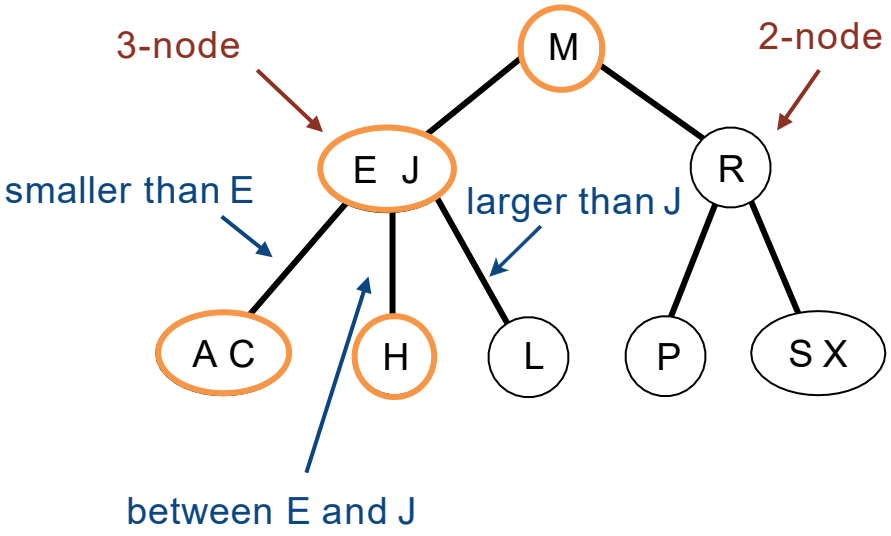
Department of Computer Science
Hofstra University

2−3 Trees

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 leaf has same length.

Symmetric order: In-order traversal yields keys in ascending order.



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)

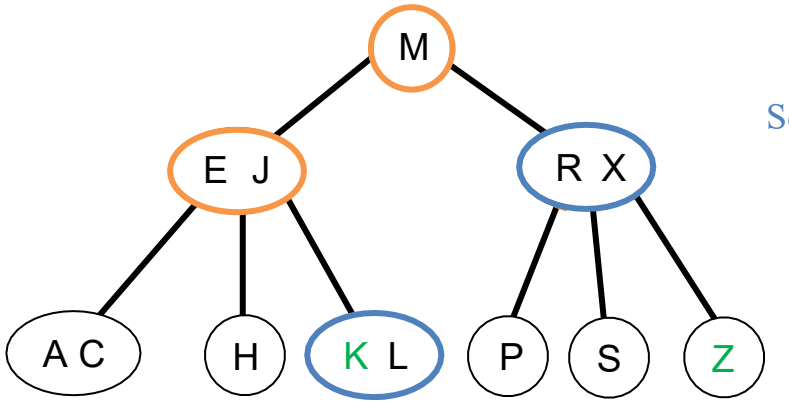
Link is null

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.

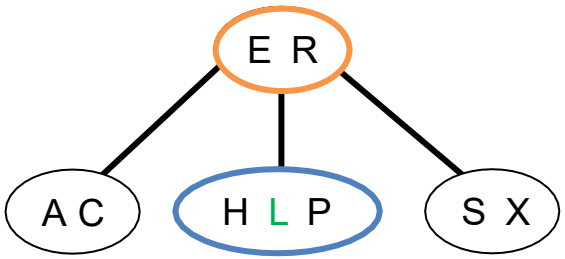
Insert in 2−3 Trees (Contd.)

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`

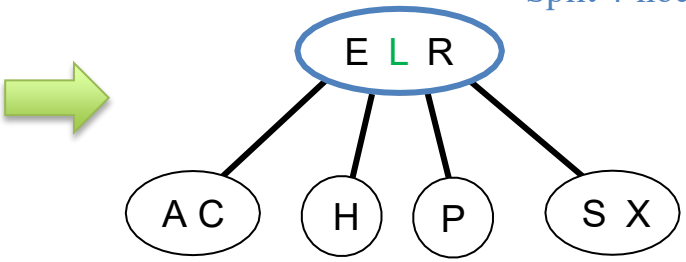
L is between E and R (go middle)



Search ends here

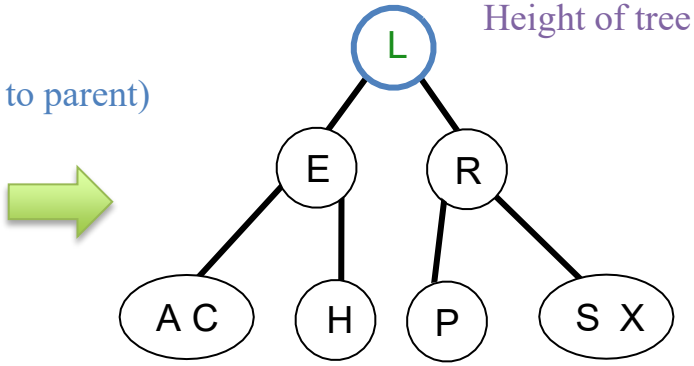
Replace 3-node with temporary 4-node containing L

Split 4-node (move L to parent)



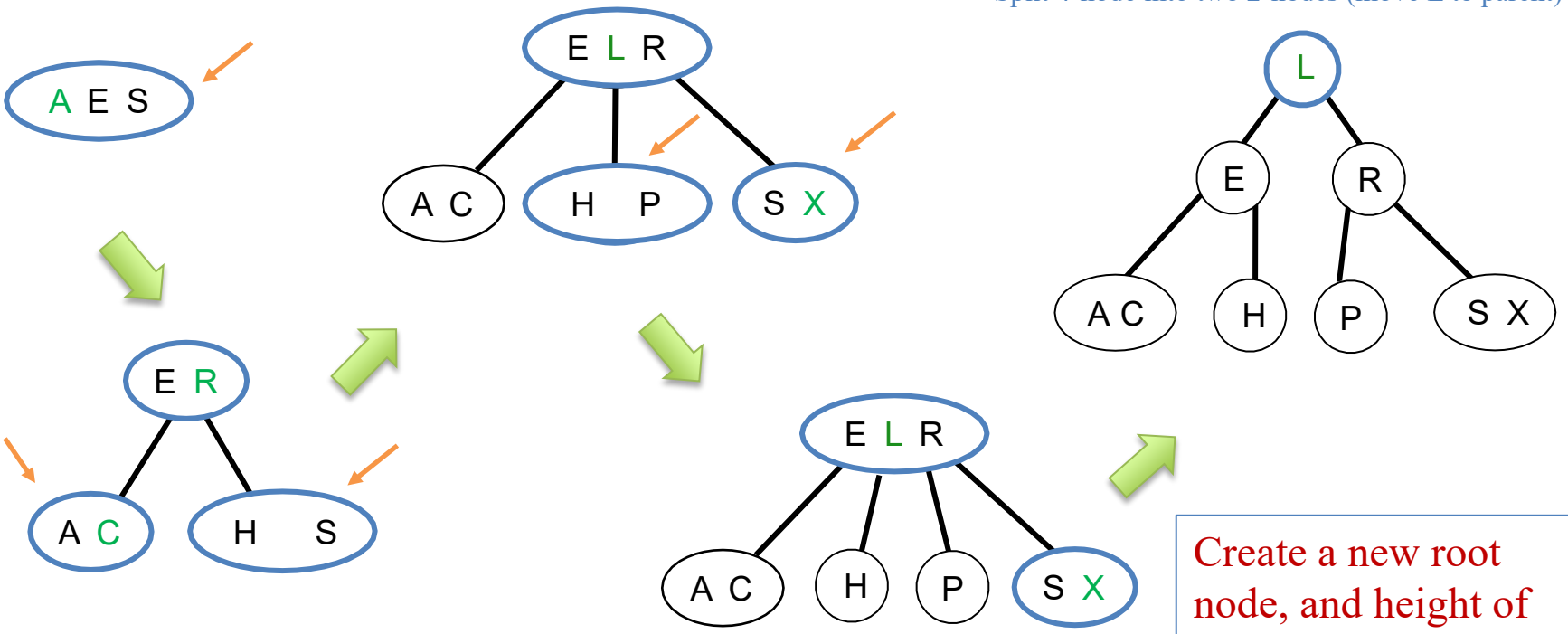
Split 4-node (move L to parent)

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)

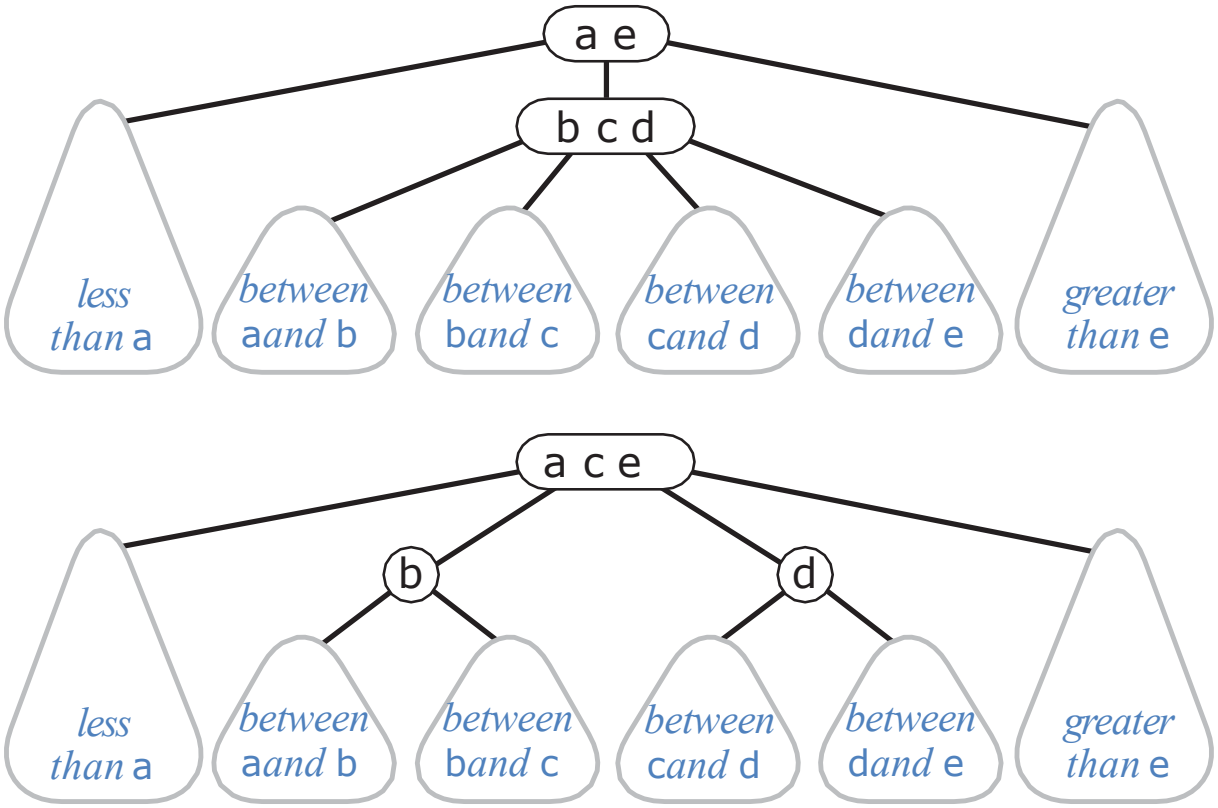


Create a new root node, and height of the tree grows by one

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.

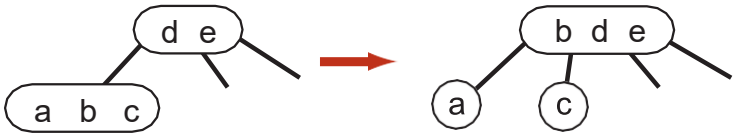
Proof. Each transformation maintains symmetric order and perfect balance.

Splitting a 4-node

root

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

left



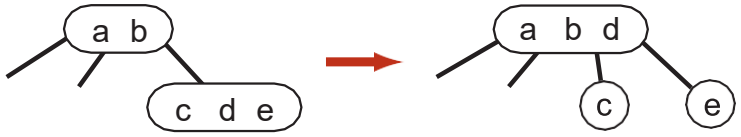
parent is a 2-node: split and parent turns into a 3-node

left

middle

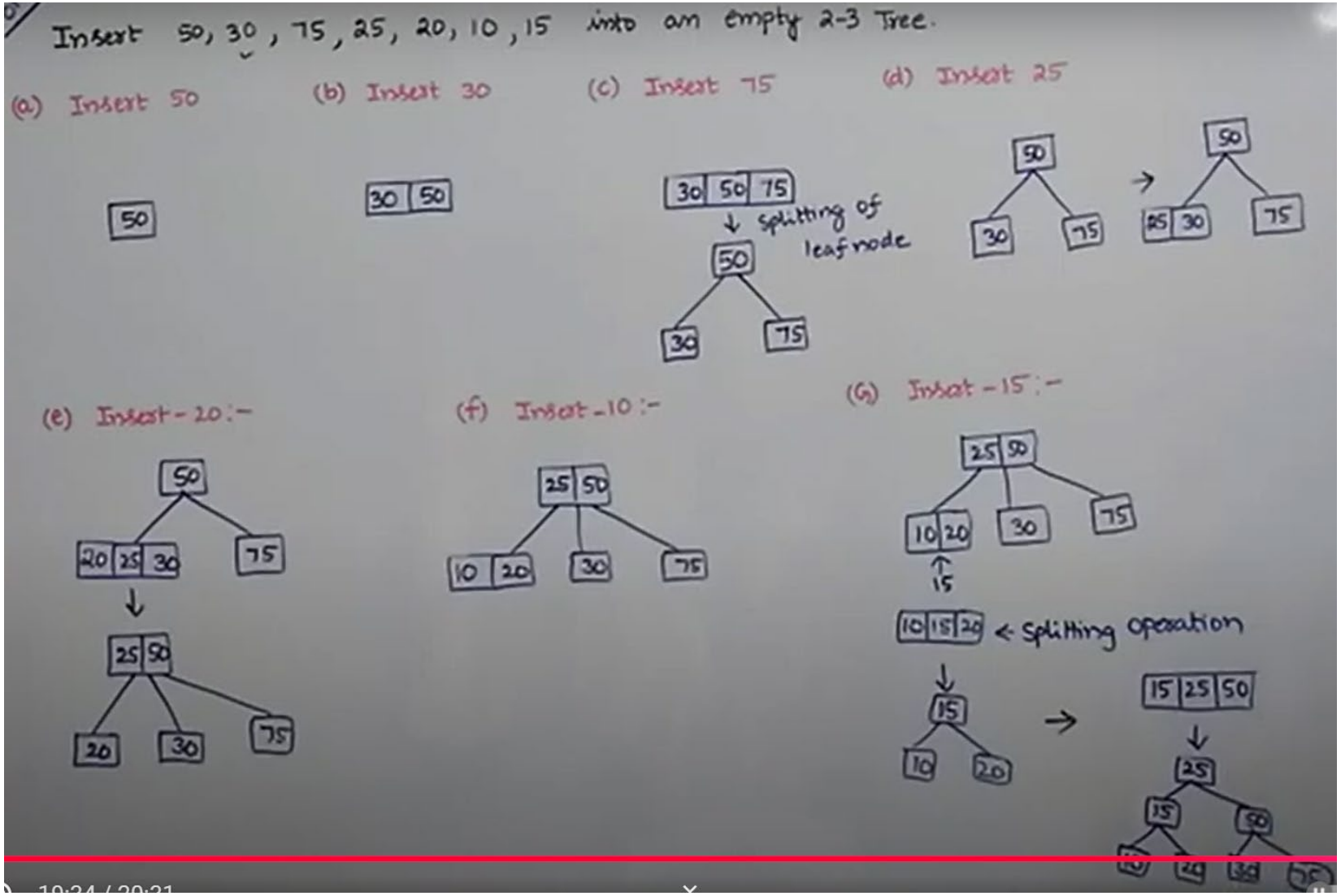
right

right



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

EXAMPLE PROBLEM ON 2-3 TREE INSERTION



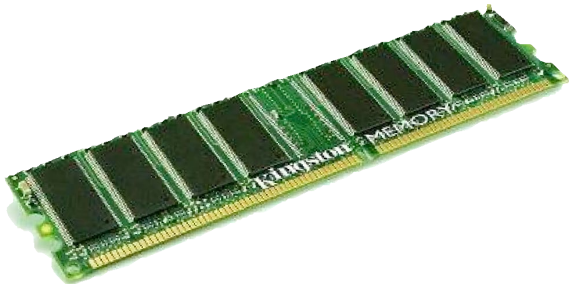
- EXAMPLE PROBLEM ON 2-3 TREE INSERTION, DIVVELA SRINIVASA RAO
  - https://www.youtube.com/watch?v=2S5Ld-FQ2dM

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

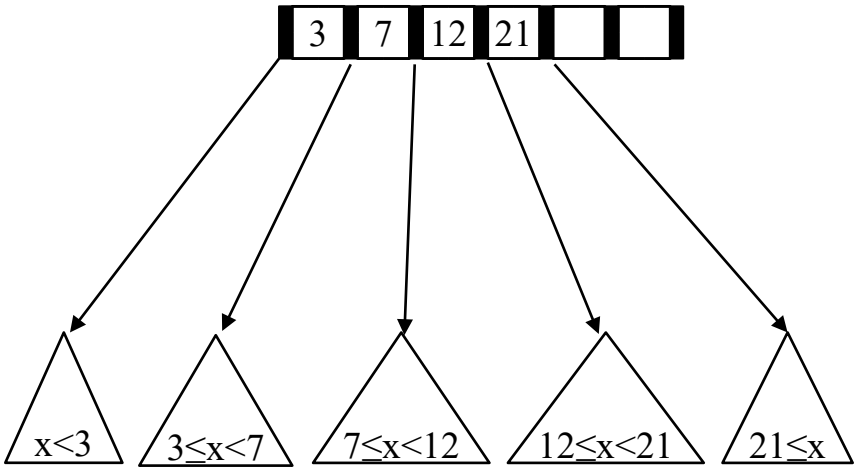
- Index structures for large datasets cannot be stored in main memory, hence must be stored on disk
- Time required to access disk is much larger than time to access main memory
- Goal: access data with minimum number of probes.

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

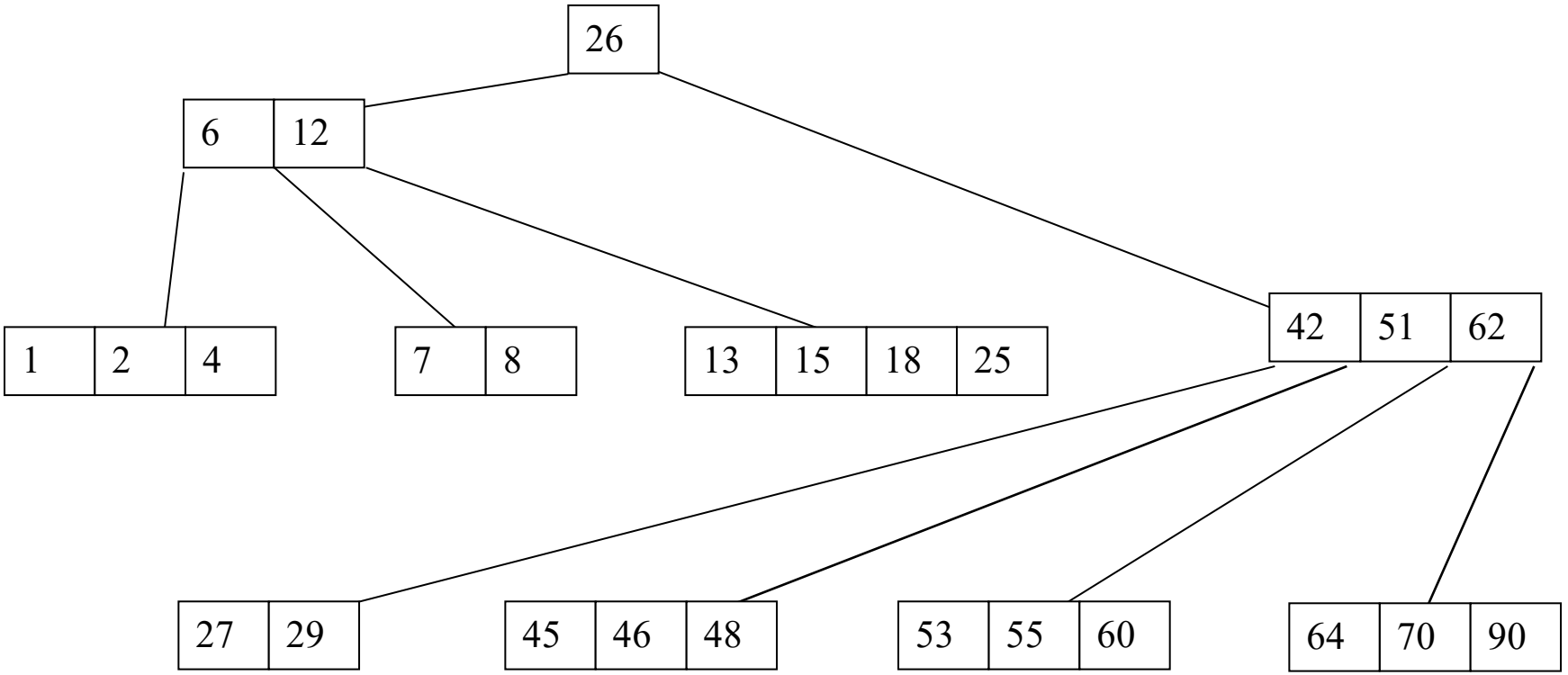
- A B-tree of order *M* is an *M*-way tree (also called "of order *M*"):
  1. It is a search tree: number of keys in each non-leaf node is one less than the number of its children, and these keys partition the keys in the children in sorted order
  2. Each node has at most *M* children (contains at most $M-1$ keys)
  3. Each non-leaf node (except the root) has at least $\lceil M/2 \rceil$ children (contains at least $\lceil M/2 \rceil - 1$ keys) (i.e., must be at least "half-full")
  4. The root has at least 2 children (contains at least 1 key)
  5. All leaf nodes are at the same level (always balanced)
- Special cases:
  - *M* = *3*: 2-3 tree (each non-leaf node has 2--3 children, contains 1--2 keys)
  - *M* = *4*: 2-4 tree (each non-leaf node has 2--4 children, contains 1--3 keys)
  - *M* = *5*: 3-5 tree (each non-leaf node has 3--5 children, contains 2--4 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



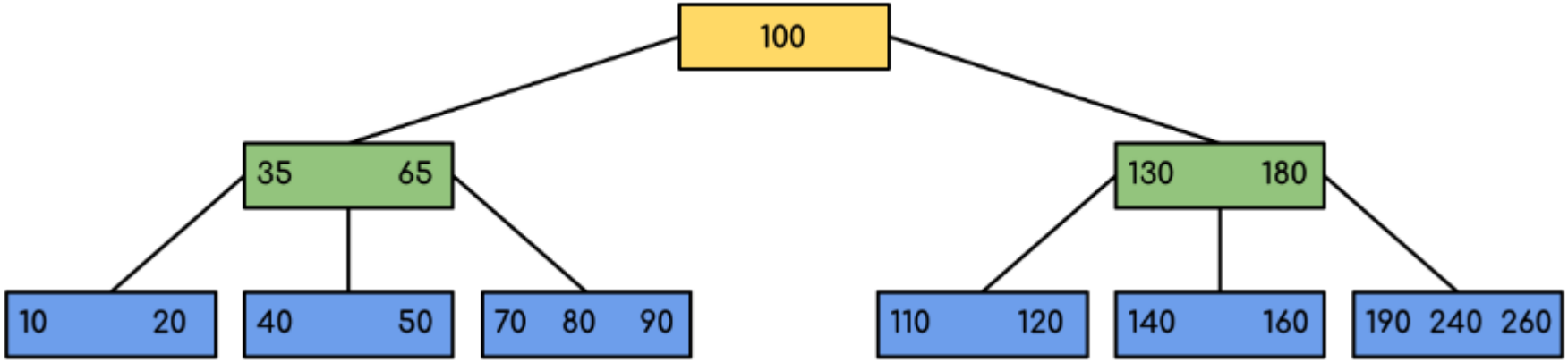
A B-tree of order 5 (M = 5)

All leaves are at the same level 3

B-trees Example II

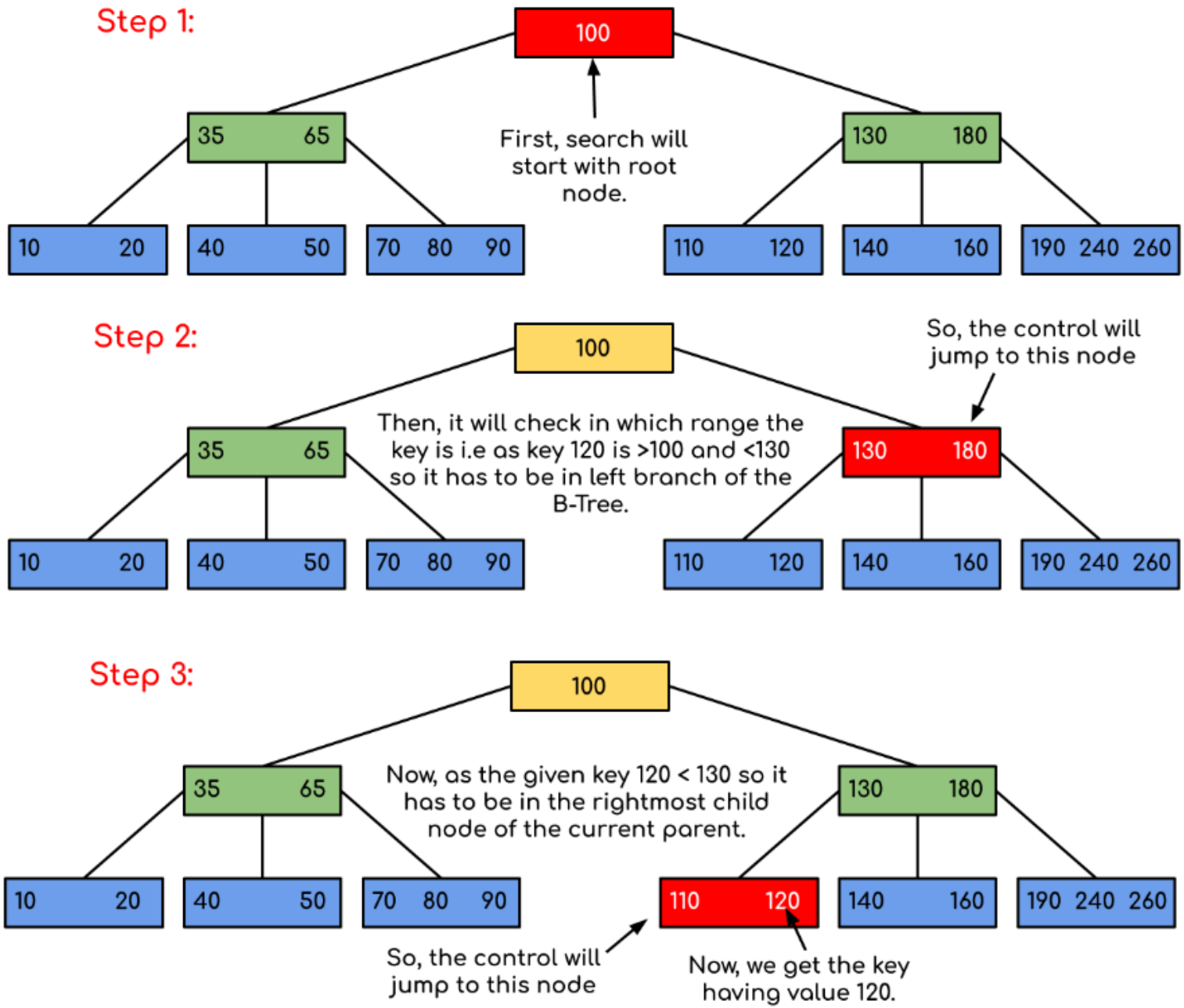
Generalization of 2-3 trees

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



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

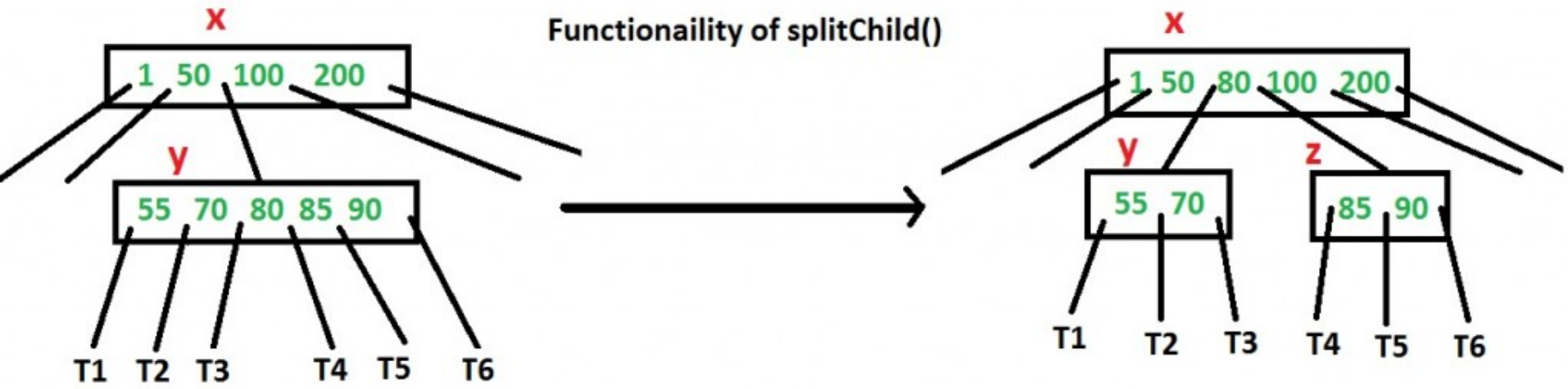
All leaves are at the same level 3



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



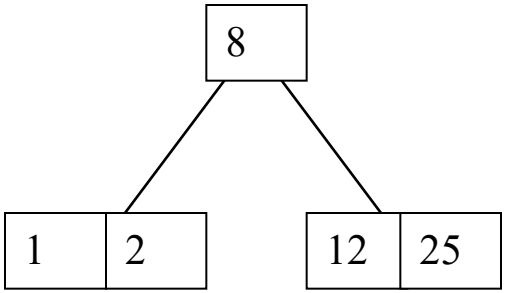
Constructing a B-tree

- 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 keys go into the root:

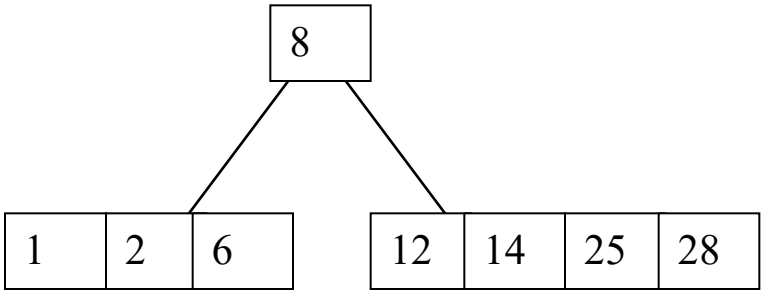
1	2	8	12

- To put the fifth key 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.)

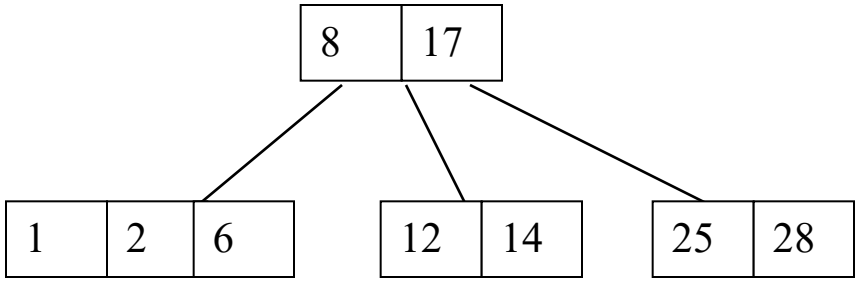


6, 14, 28 get added to the leaf nodes:

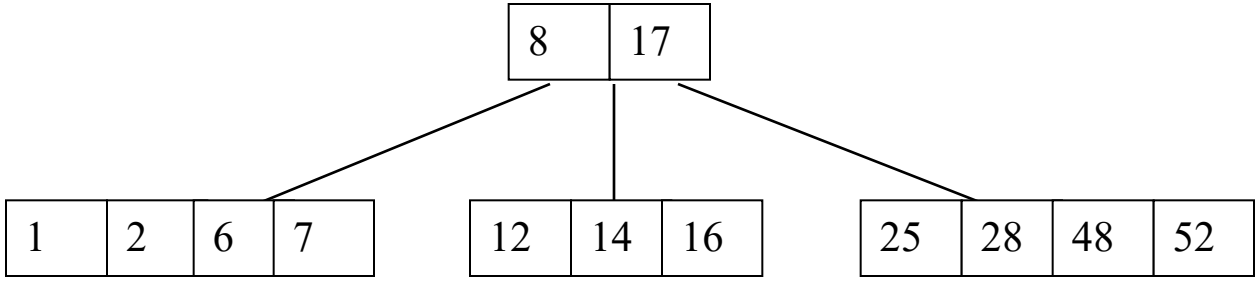


Constructing a B-tree (contd.)

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

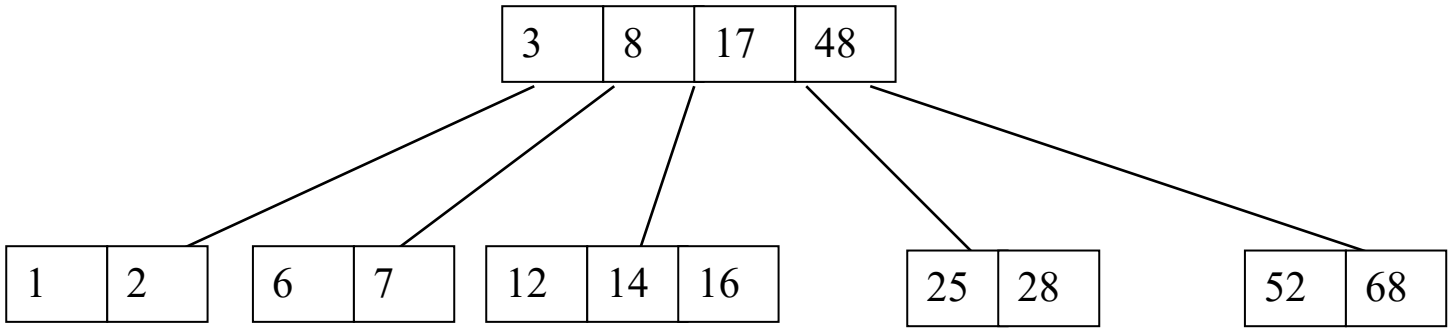


7, 52, 16, 48 get added to the leaf nodes

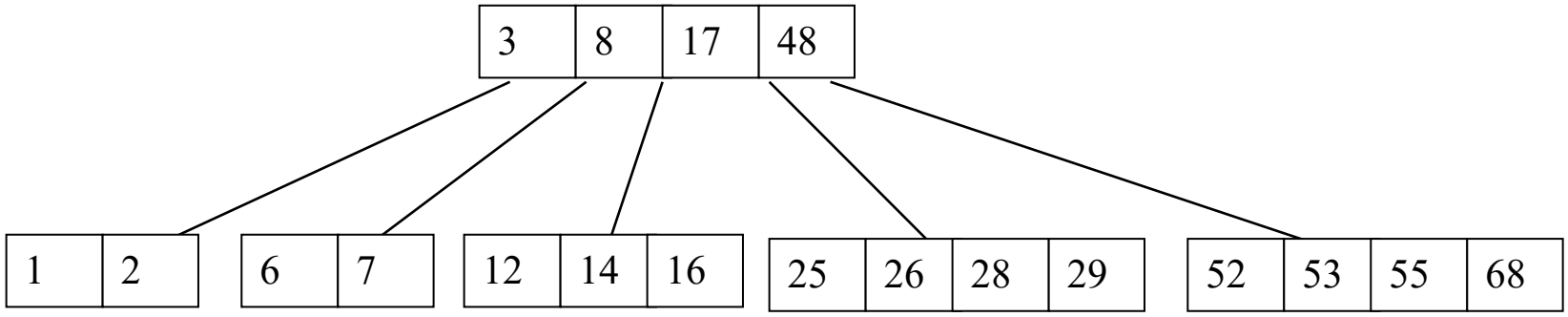


Constructing a B-tree (contd.)

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

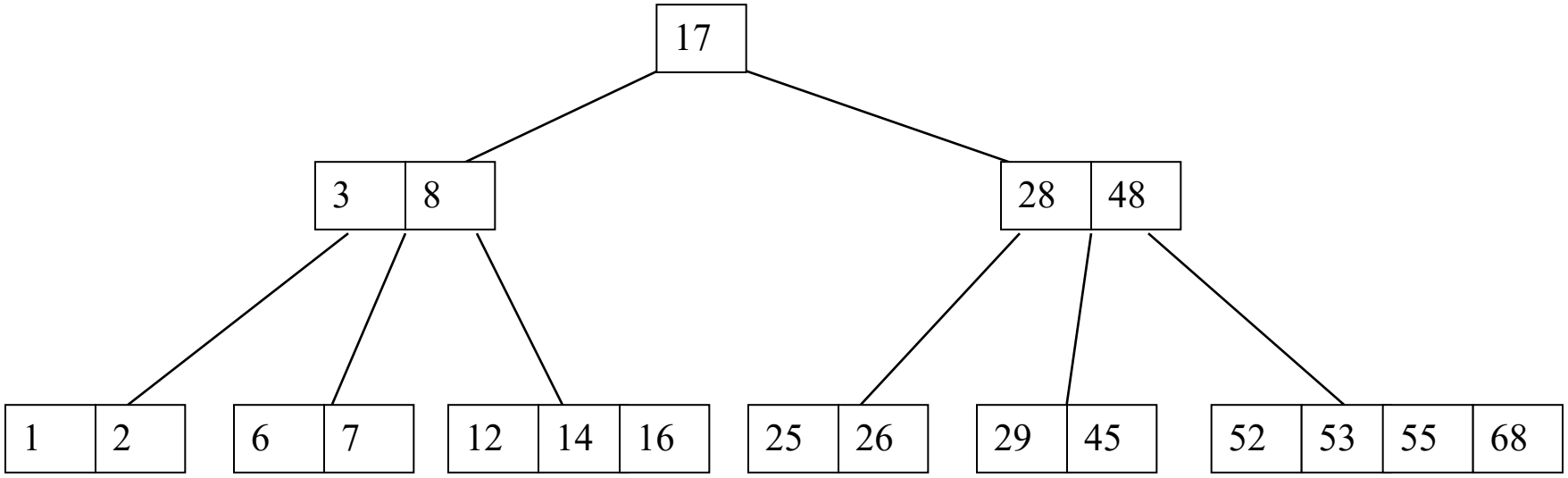


Constructing a B-tree (contd.)

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

- (B-tree deletion will NOT be covered in exam)
- 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

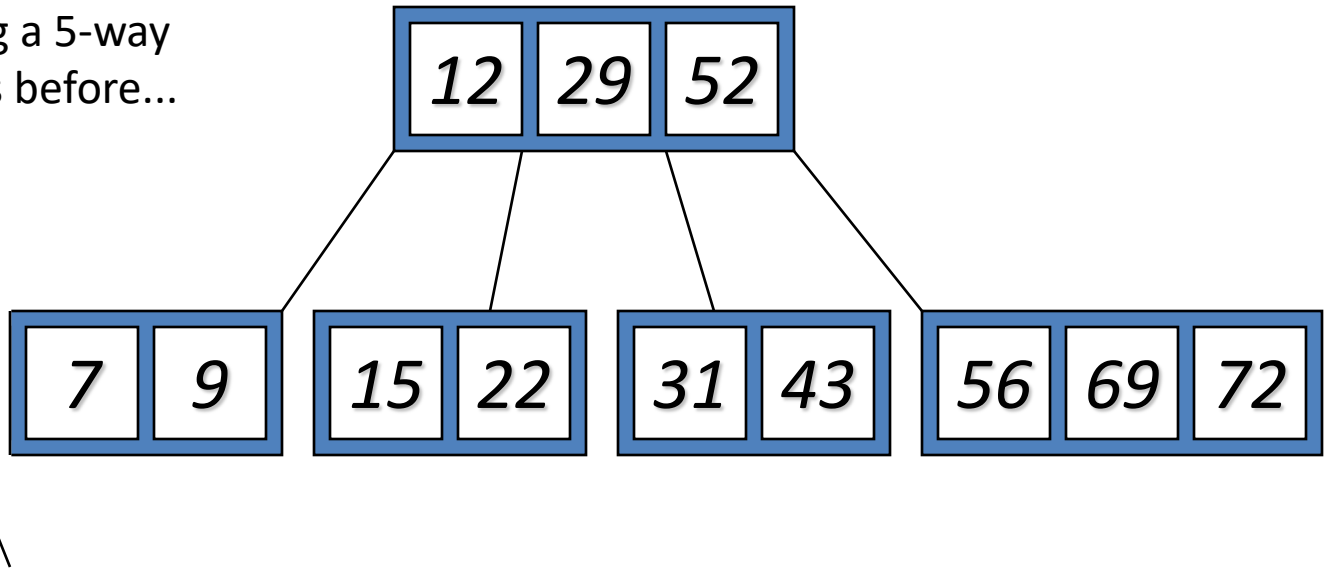
- Definition of Predecessor and Successor:
  - The predecessor of a key in an internal node is the largest key in its left subtree.
  - The successor is the smallest key in its right subtree
- Traversal to Leaves:
  - To find the predecessor, you follow the rightmost path from the left child of the key's 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

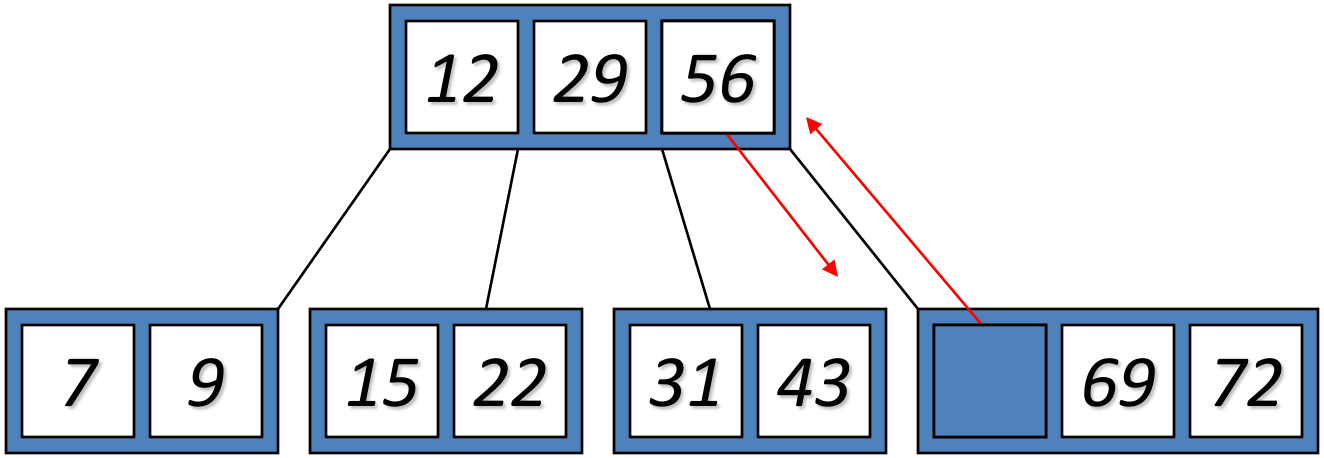
Type #1: Simple leaf deletion

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

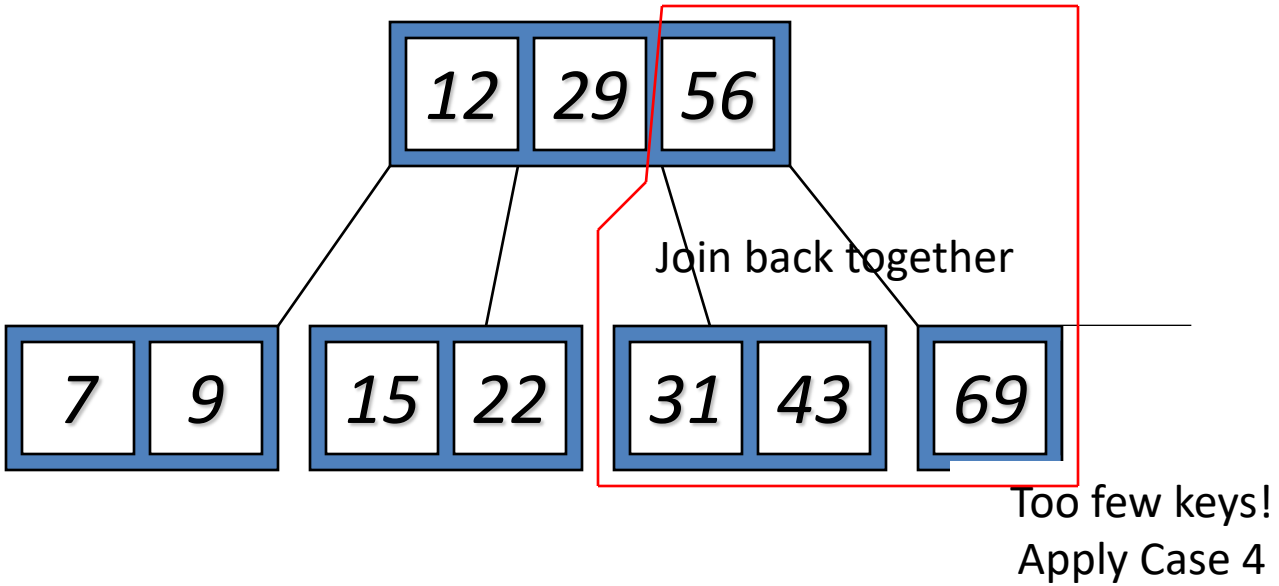


Delete 2: Since there are enough keys in the node, just delete it

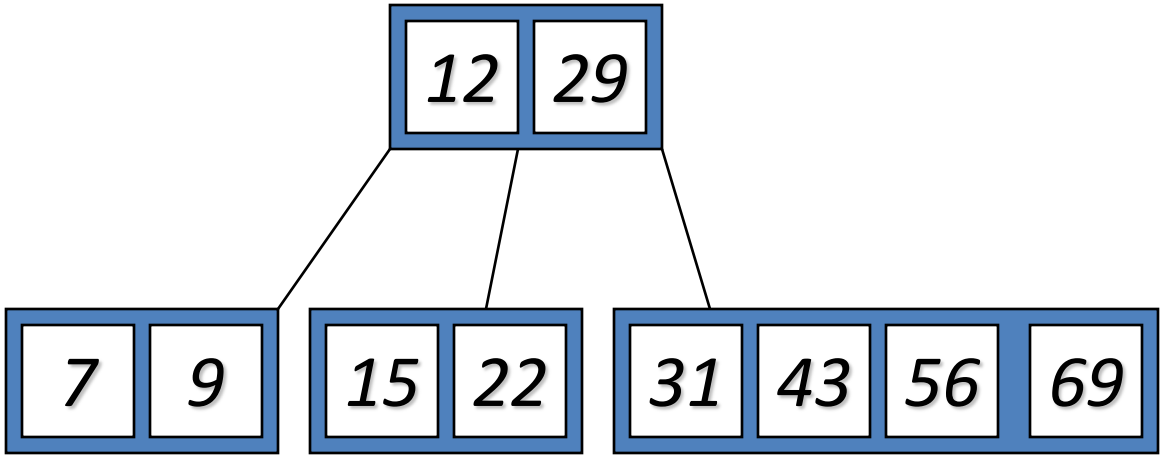
Type #2: Simple non-leaf deletion



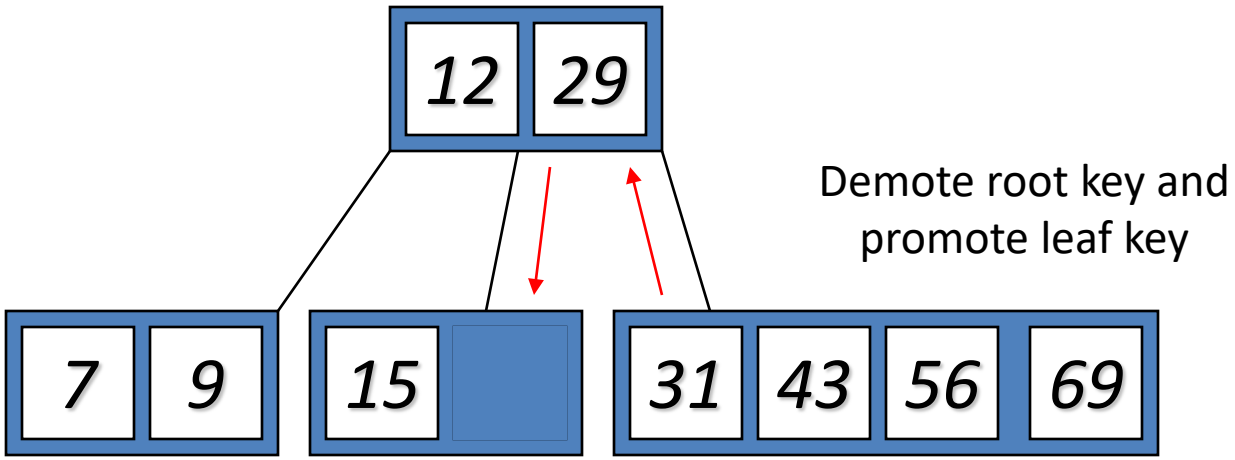
Type #4: Too few keys in node and its siblings



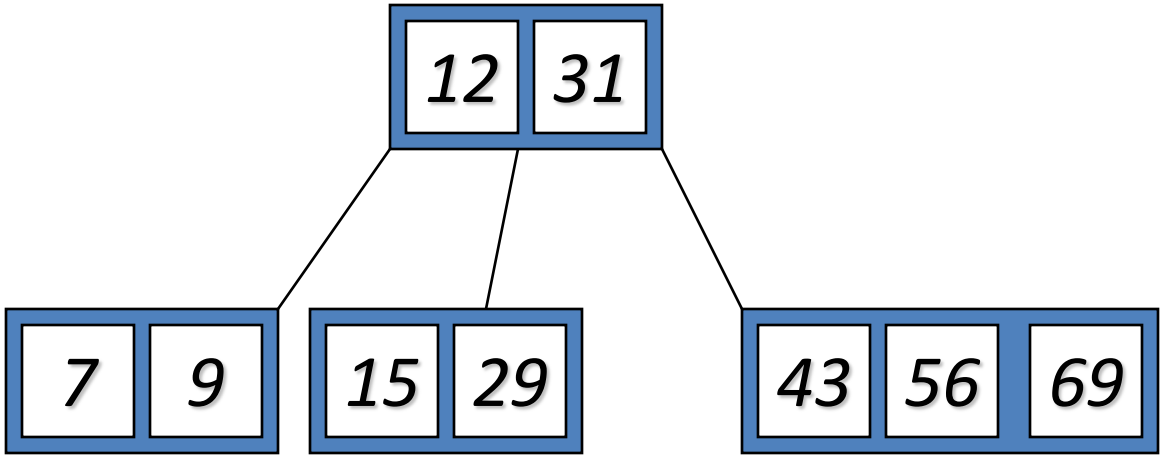
Type #4: Too few keys in node and its siblings



Type #3: Enough siblings



Type #3: Enough siblings



Analysis of B-Trees I

- For an M-way B-tree:
  - Each node has at most M children ⇒ the maximum branching factor is M
  - Each non-leaf node (except the root) has at least $\lceil M/2 \rceil$ children ⇒ the minimum branching factor is $\lceil M/2 \rceil$
- Maximum number of keys in an M-way B-tree with height h: This occurs when the tree is as full as possible (each non-leaf node has M children, with $M - 1$ keys per node)

  # keys in root (level 0): $M - 1$, with 1 node, $M - 1$ keys per node

  # keys at level 1: $M(M - 1)$, with M nodes, $M - 1$ keys per node

  # keys at level 2: $M^2(M - 1)$, with M^2 nodes, $M - 1$ keys per node

  . . .

  # keys at level h: $M^h(M - 1)$, with M^hnodes, $M - 1$ keys per node
- So, the maximum total number of keys is $M^{h+1} - 1$
  - $(1 + M + M^2 + \cdots + M^h)(M - 1) = [(M^{h+1} - 1)/(M - 1)](M - 1) = M^{h+1} - 1$

Analysis of B-Trees II

- Minimum number of keys in an *M*-way B-tree with height *h*: This occurs when the tree is as sparse as possible (root node has 2 children, with 1 key; each non-leaf node (except the root) has $\lceil M/2 \rceil$ children, with $\lceil M/2 \rceil - 1$ keys per node). Let $t = \lceil M/2 \rceil$ for brevity

  # keys in root (level 0): 1, with 1 node, 1 key per node

  # keys at level 1: $2(t-1)$, with 2 nodes, $t-1$ keys per node

  # keys at level 2: $2t(t-1)$, with $2t$ nodes, $t-1$ keys per node

  . . .

  # keys at level h: $2t^{h-1}(t-1)$, with $2t^{h-1}$nodes, $t-1$ keys per node

- So, the minimum total number of keys is $2\lceil M/2 \rceil^h - 1$
  - $1 + 2(t-1) + 2t(t-1) + \cdots + 2t^{h-1}(t-1) = 1 + 2[(t^h - 1)/(t-1)](t-1) = 2t^h - 1$
- Examples:
  - For $M = 3, h = 2$: Max: 3^3 − 1 = 26 keys; Min: $2 \cdot 2^2 - 1 = 7$ keys
  - For $M = 4, h = 1$: Max: 4^2 − 1 = 15 keys; Min: $2 \cdot 2^1 - 1 = 3$ keys

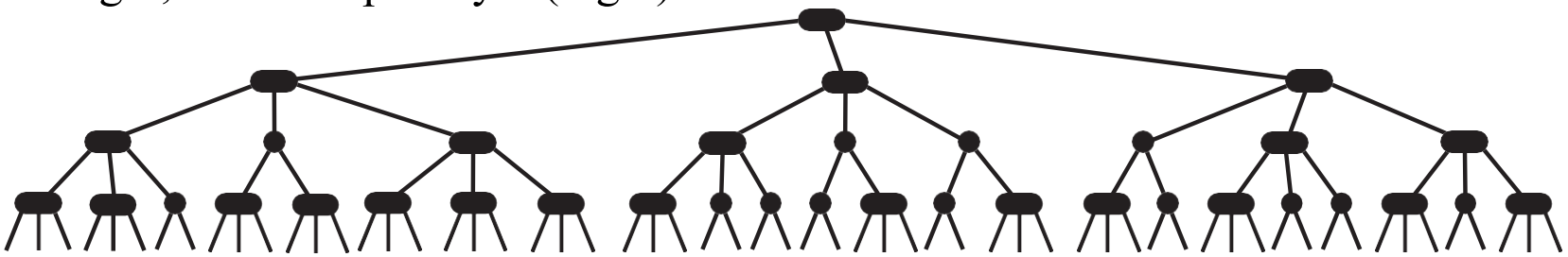
Recall: Height of a Binary Tree

- A binary tree with height h has maximum total number of nodes: $n = 2^{h+1} - 1$
- For a binary tree with n nodes, the height h is bounded by: $\lceil \log_2(n+1) \rceil - 1 \leq h \leq n - 1$
  - The lower bound represents a perfectly balanced tree, and the upper bound represents a degenerate tree (essentially a linked list).
  - The minimum height of a binary tree with n nodes is $\lceil \log_2(n+1) \rceil - 1$, which occurs in the most balanced configuration
  - The maximum height of a binary tree with n nodes is $n - 1$, which occurs in the case of a skewed tree (a linear chain or linked list).

Height of an *M*-way B Tree

- Perfect balance: Every path from root to leaf has same length.
- Minimum tree height h_{min}
  - This occurs when the tree is as full as possible
  - Maximum total number of keys $n = M^{h+1} - 1$
  - Minimum tree height $h_{min} = \lceil \log_M(n+1) \rceil - 1$
- Maximum tree height h_{max}
  - This occurs when the tree is as sparse as possible
  - Maximum total number of keys $n = 2\lceil M/2 \rceil^h - 1$
  - Minimum tree height $h_{min} = \lceil \log_{\lceil \frac{M}{2} \rceil}(\frac{n+1}{2}) \rceil$
  - It is not possible to have a linear chain since each non-leaf node must be at least "half-full" (with $\lceil M/2 \rceil - 1$ keys)
- In big-O notation, tree height is O(log n)
- All operations (search, insert and delete) have cost proportional to the tree height, with complexity O(log n)

	Best case	Average case	Worst case
BST	O(1)	O(log n)	O(n)
BTree	O(1)	O(log n)	O(log n)



Reasons for using B-Trees

- B-Trees are always balanced, i.e., all leaf nodes are at the same level
- The cost of each disk read is high but does not depend much on the amount of data transferred, if consecutive keys are transferred and they fit within a memory page
  - e.g., $M=101$ and $h = 3$ → $101^4 - 1$ (~100 million)
  - A B-tree of order $M=101$ and height 3 can hold ~100 million keys, and any key can be accessed with 3 disk reads (assuming each node fits within one memory page)

Comparing Trees

- Binary trees
  - Can become *unbalanced* and *lose* their good time complexity (big O)
  - AVL trees are self-balancing BSTs
  - Heaps are balanced but only *prioritise* (not order) the keys

- Multi-way trees
  - B-Trees can be *M*-way
  - 2-3 tree is 3-way B-Tree. It approximates a balanced BST, replacing the AVL tree's balancing operations with 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

- Understanding B-Trees: The Data Structure Behind Modern Databases, Spanning Tree
  - https://www.youtube.com/watch?v=K1a2Bk8NrYQ
  - The example is a 3-5 tree
- The Most Elegant Search Structure | (a,b)-trees, Tom S
  - https://www.youtube.com/watch?v=lifFgyB77zc
  - (a,b)-tree: a=$\lceil M/2 \rceil$, b=M>=2a-1=2$\lceil M/2 \rceil$-1
  - The example is a 2-4 tree
- B Tree tutorials
  - https://spetriuk.github.io/algorithms/B-Tree.%201.%20Introduction/
  - https://spetriuk.github.io/algorithms/B-Tree.%202.%20Insert%20Operation/