20-0: **Indexing**

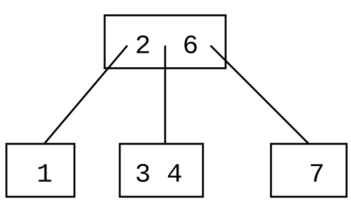
- Operations:
  - Add an element
  - Remove an element
  - Find an element, using a key
  - Find all elements in a range of key values

20-1: **Indexing**

- Sorted List
  - Find / Find in Range fast
  - Add / Remove slow
- Unsorted List / Hash Table
  - Add, Find, Remove fast (hash)
  - Find in Range slow
- Binary Search Tree
  - All operations are fast (O(lg n))
  - *if* the tree is balanced

20-2: **Indexing**

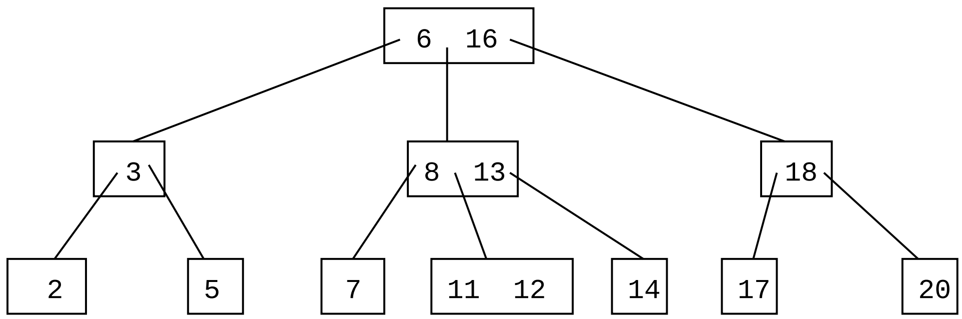
- Generalized Binary Search Trees
  - Each node can store several keys, instead of just one
  - Values in subtrees between values in surrounding keys
  - For non leaves, # of children = # of keys + 1



20-3: **2-3 Trees**

- Generalized Binary Search Tree
  - Each node has 1 or 2 keys
  - Each (non-leaf) node has 2-3 children
    - hence the name, 2-3 Trees
  - All leaves are at the same depth

20-4: **Example 2-3 Tree**



20-5: **Finding in 2-3 Trees**

- How can we find an element in a 2-3 tree?

20-6: **Finding in 2-3 Trees**

- How can we find an element in a 2-3 tree?
  - If the tree is empty, return false
  - If the element is stored at the root, return true
  - Otherwise, recursively find in the appropriate subtree

20-7: **Inserting into 2-3 Trees**

- Always insert at the leaves
- To insert an element:
  - Find the leaf where the element would live, if it was in the tree
  - Add the element to that leaf

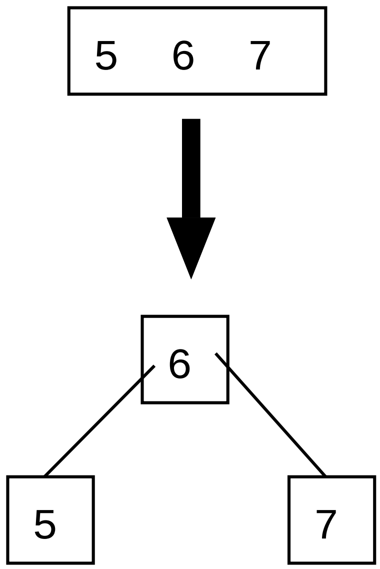
20-8: **Inserting into 2-3 Trees**

- Always insert at the leaves
- To insert an element:
  - Find the leaf where the element would live, if it was in the tree
  - Add the element to that leaf
    - What if the leaf already has 2 elements?

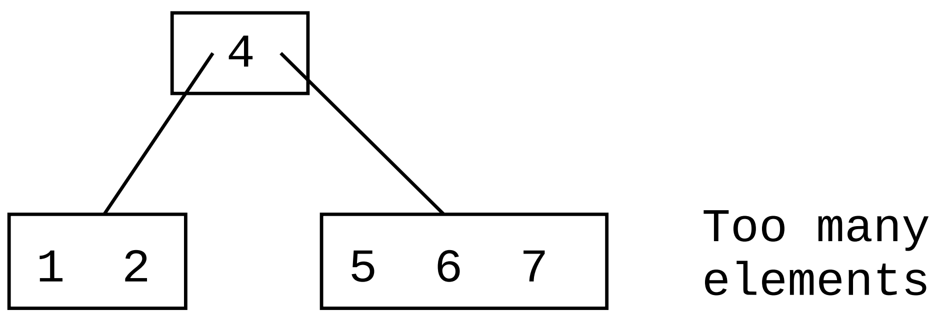
20-9: **Inserting into 2-3 Trees**

- Always insert at the leaves
- To insert an element:
  - Find the leaf where the element would live, if it was in the tree
  - Add the element to that leaf
    - What if the leaf already has 2 elements?
    - Split!

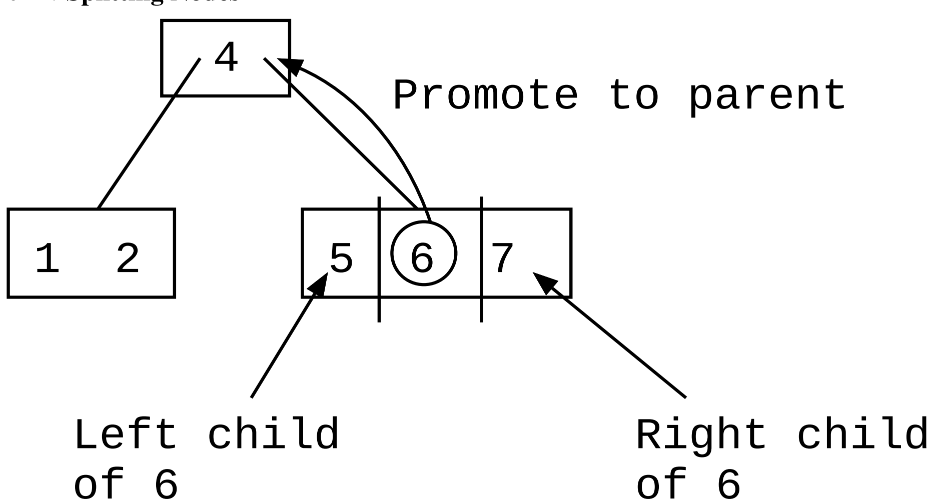
20-10: **Splitting Nodes**



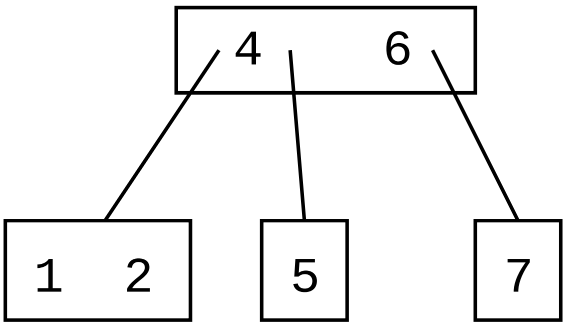
20-11: **Splitting Nodes**



20-12: **Splitting Nodes**



20-13: **Splitting Nodes**

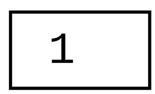


20-14: **Splitting Root**

- When we split the root:
  - Create a new root
  - Tree grows in height by 1

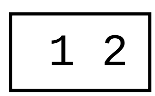
20-15: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



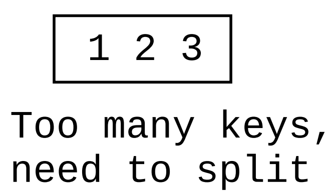
20-16: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



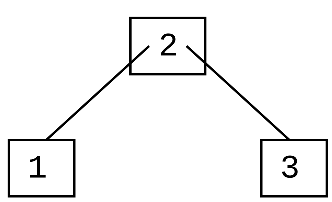
20-17: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



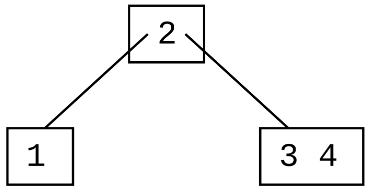
20-18: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



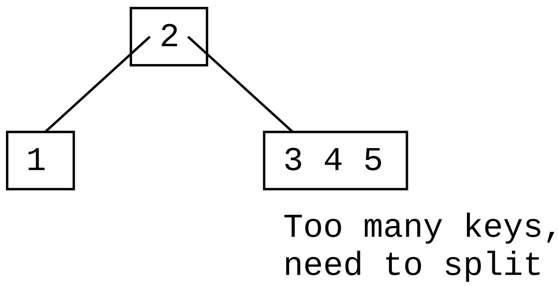
20-19: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



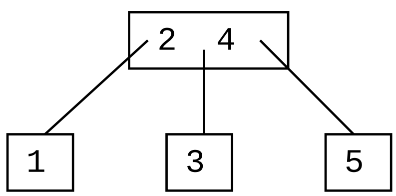
20-20: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



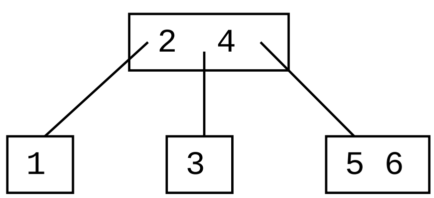
20-21: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



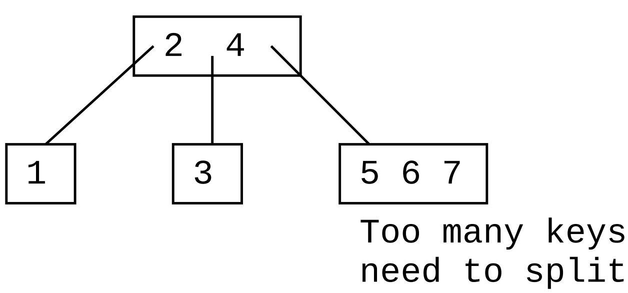
20-22: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



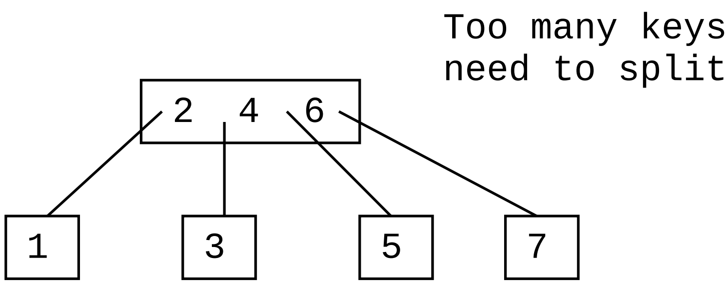
20-23: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



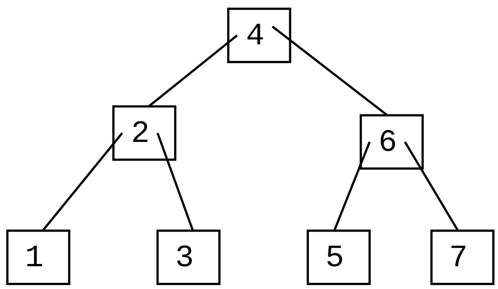
20-24: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



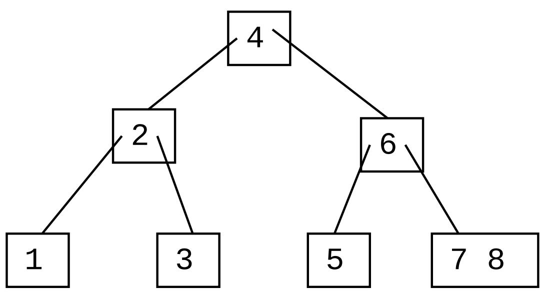
20-25: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



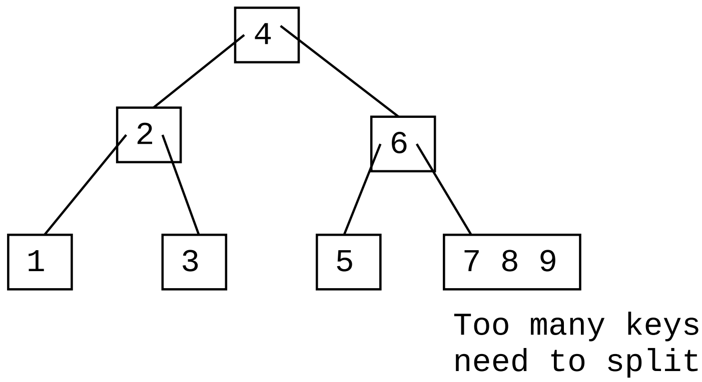
20-26: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



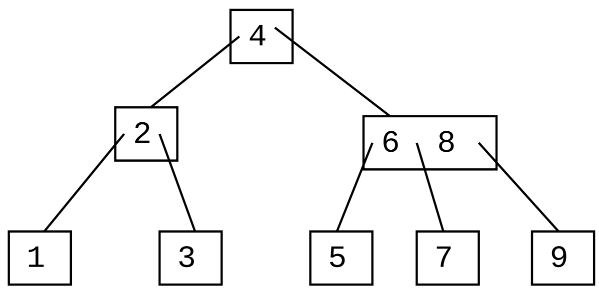
20-27: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



20-28: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree



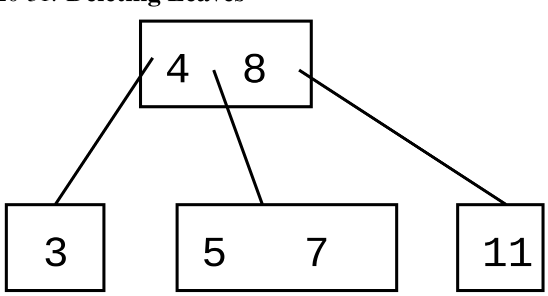
20-29: **Deleting from 2-3 Tree**

- As with BSTs, we will have 2 cases:
    - Deleting a key from a leaf
    - Deleting a key from an internal node

20-30: **Deleting Leaves**

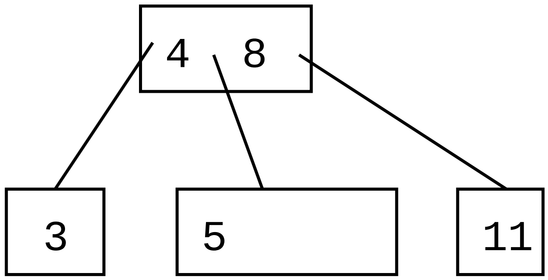
- If leaf contains 2 keys
    - Can safely remove a key

20-31: **Deleting Leaves**



- Deleting 7

20-32: **Deleting Leaves**

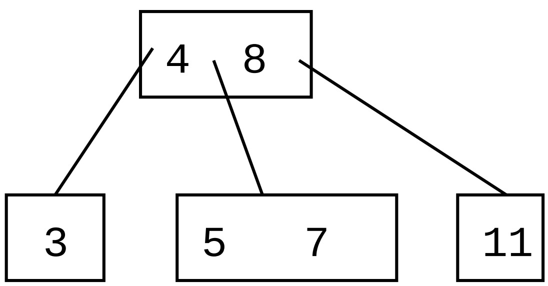


- Deleting 7

20-33: **Deleting Leaves**

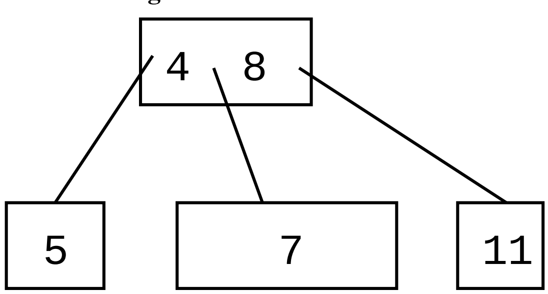
- If leaf contains 1 key
  - Cannot remove key without making leaf empty
  - Try to steal extra key from sibling

20-34: **Deleting Leaves**



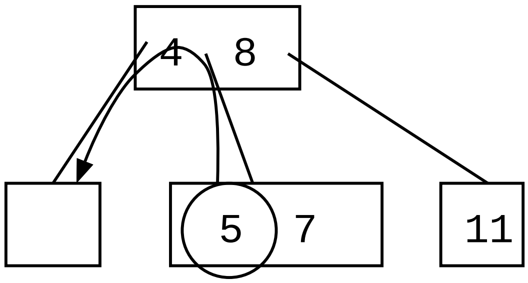
- Deleting 3 – we can steal the 5

20-35: **Deleting Leaves**



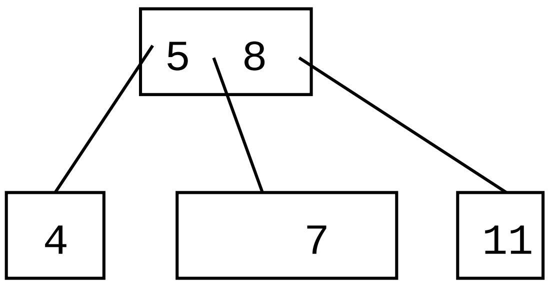
- Not a 2-3 tree. What can we do?

20-36: **Deleting Leaves**



- Steal key from sibling *through parent*

20-37: **Deleting Leaves**

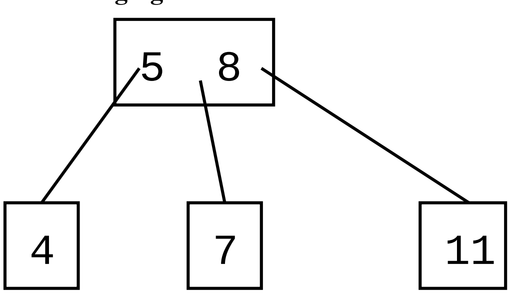


- Steal key from sibling *through parent*

20-38: **Deleting Leaves**

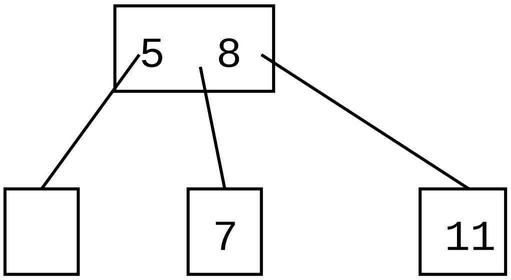
- If leaf contains 1 key, and no sibling contains extra keys
  - Cannot remove key without making leaf empty
  - Cannot steal a key from a sibling
  - Merge with sibling
    - split in reverse

20-39: **Merging Nodes**



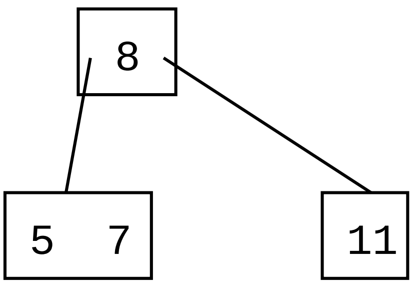
- Removing the 4

20-40: **Merging Nodes**



- Removing the 4
- Combine 5, 7 into one node

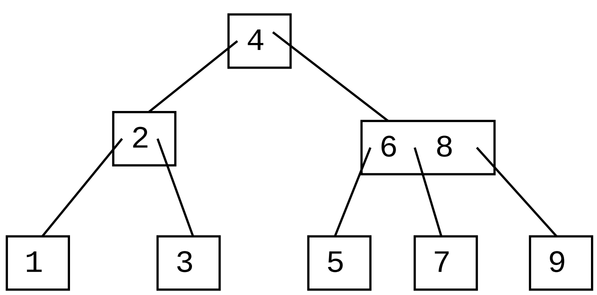
20-41: **Merging Nodes**



20-42: **Merging Nodes**

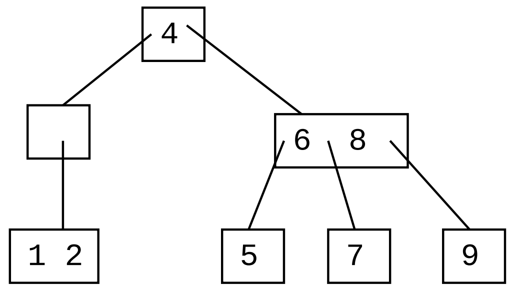
- Merge decreases the number of keys in the parent
    - May cause parent to have too few keys
- Parent can steal a key, or merge again

20-43: **Merging Nodes**



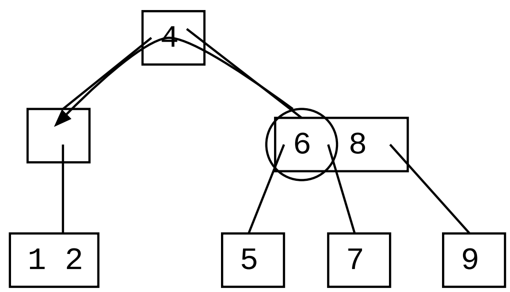
- Deleting the 3 – cause a merge

20-44: **Merging Nodes**



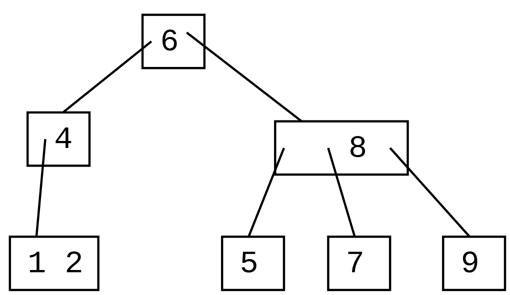
- Deleting the 3 – cause a merge
- Not enough keys in parent

20-45: **Merging Nodes**



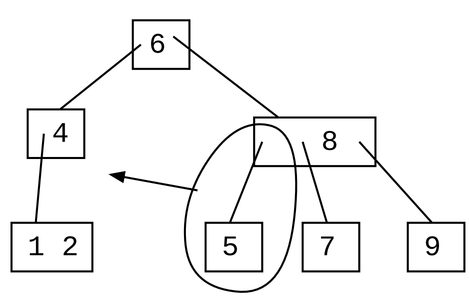
- Steal key from sibling

20-46: **Merging Nodes**



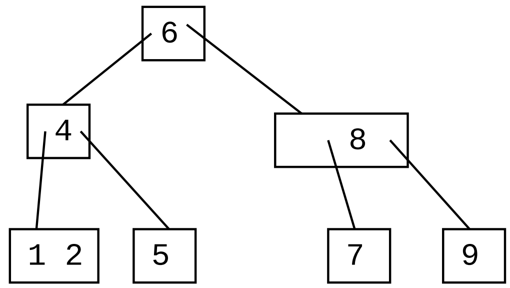
- Steal key from sibling

20-47: **Merging Nodes**



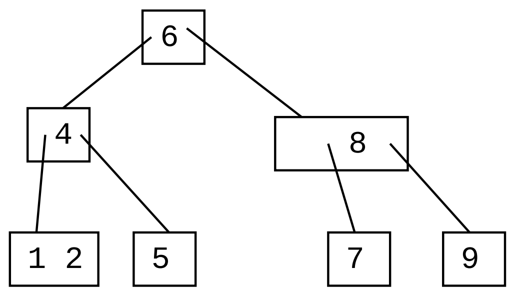
- When we steal a key from an internal node, steal nearest subtree as well

20-48: **Merging Nodes**



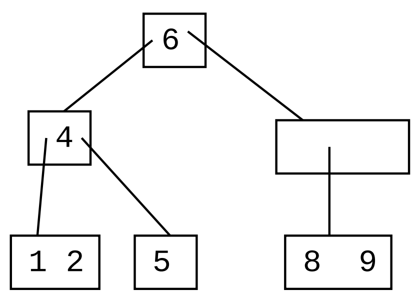
- When we steal a key from an internal node, steal nearest subtree as well

20-49: **Merging Nodes**



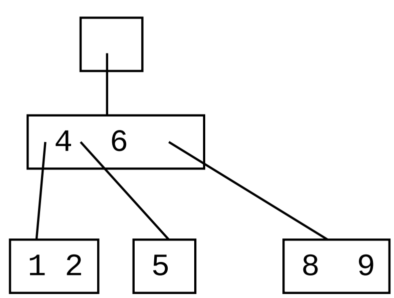
- Deleting the 7 – cause a merge

20-50: **Merging Nodes**



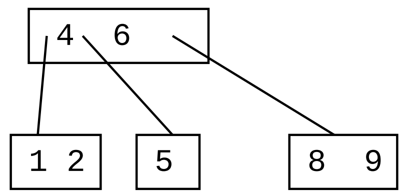
- Parent has too few keys – merge again

20-51: **Merging Nodes**



- Root has no keys – delete

20-52: **Merging Nodes**



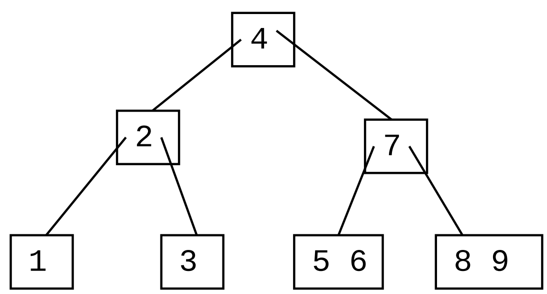
20-53: **Deleting Interior Keys**

- How can we delete keys from non-leaf nodes?
  - *HINT:* How did we delete non-leaf nodes in standard BSTs?

20-54: **Deleting Interior Keys**

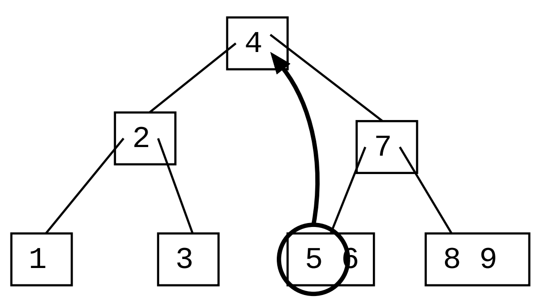
- How can we delete keys from non-leaf nodes?
  - Replace key with smallest element subtree to right of key
  - Recursivly delete smallest element from subtree to right of key
- (can also use largest element in subtree to left of key)

20-55: **Deleting Interior Keys**



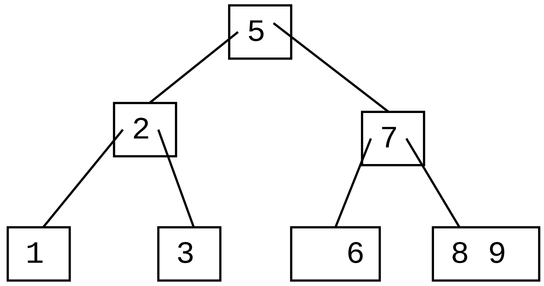
- Deleting the 4

20-56: **Deleting Interior Keys**

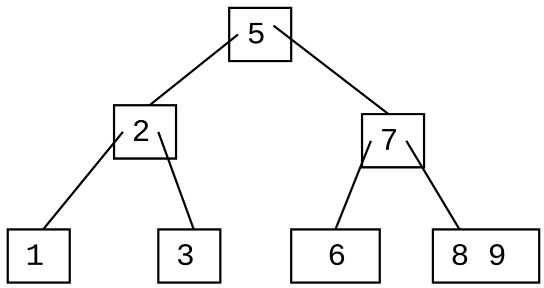


- Deleting the 4
- Replace 4 with smallest element in tree to right of 4

20-57: **Deleting Interior Keys**

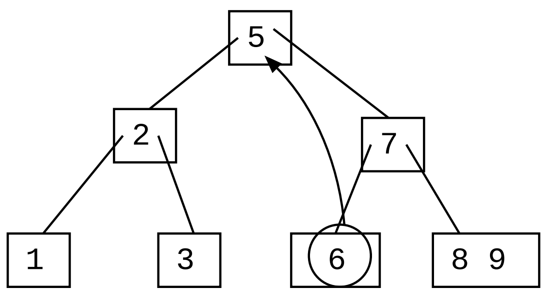


20-58: **Deleting Interior Keys**



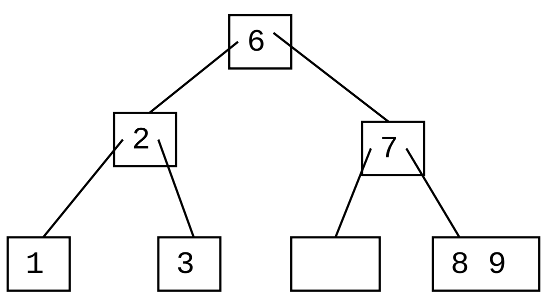
- Deleting the 5

20-59: **Deleting Interior Keys**



- Deleting the 5
- Replace the 5 with the smallest element in tree to right of 5

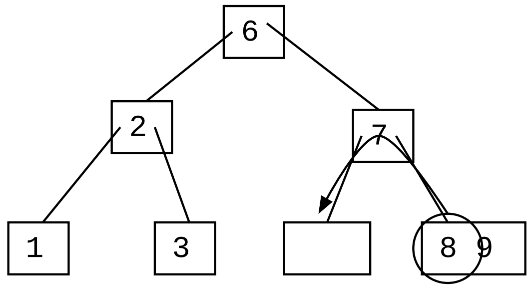
20-60: **Deleting Interior Keys**



- Deleting the 5
- Replace the 5 with the smallest element in tree to right of 5

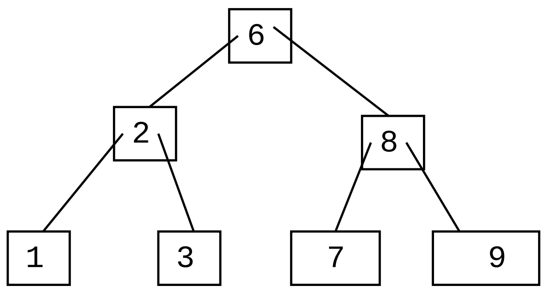
- Node with two few keys

20-61: **Deleting Interior Keys**

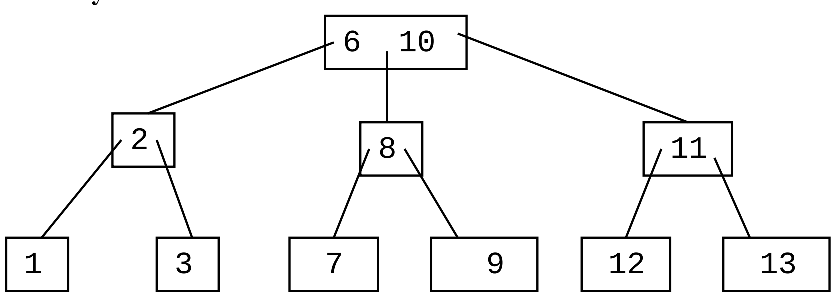


- Node with two few keys
- Steal a key from a sibling

20-62: **Deleting Interior Keys**

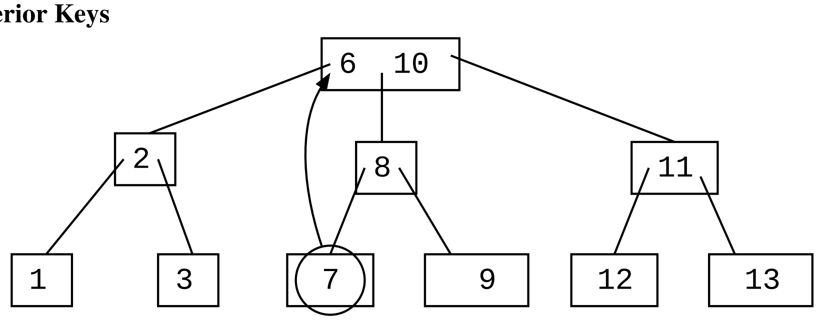


20-63: **Deleting Interior Keys**



- Removing the 6

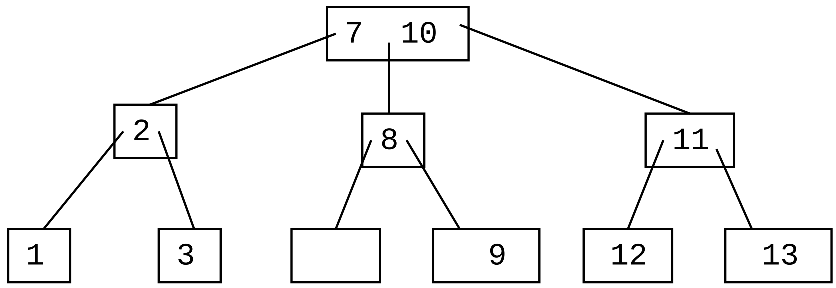
20-64: **Deleting Interior Keys**



- Removing the 6

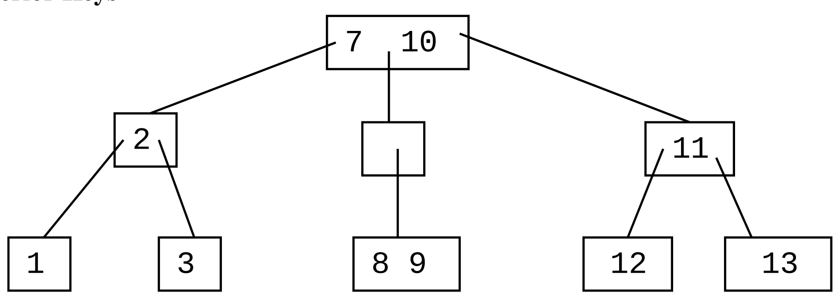
- Replace the 6 with the smallest element in the tree to the right of the 6

20-65: **Deleting Interior Keys**



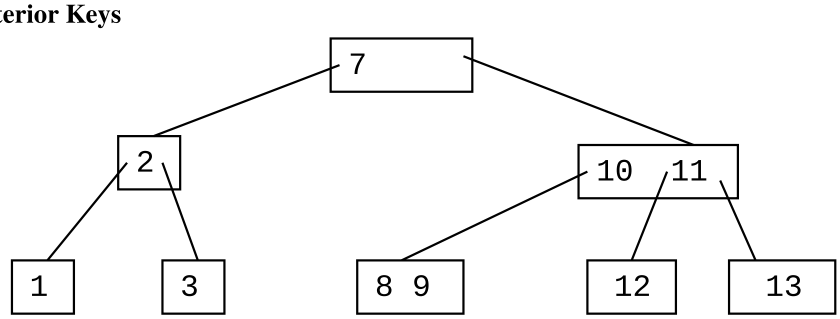
- Node with too few keys
  - Can’t steal key from sibling
  - Merge with sibling

20-66: **Deleting Interior Keys**



- Node with too few keys
  - Can’t steal key from sibling
  - Merge with sibling
  - (arbitrarily pick right sibling to merge with)

20-67: **Deleting Interior Keys**



20-68: **Generalizing 2-3 Trees**

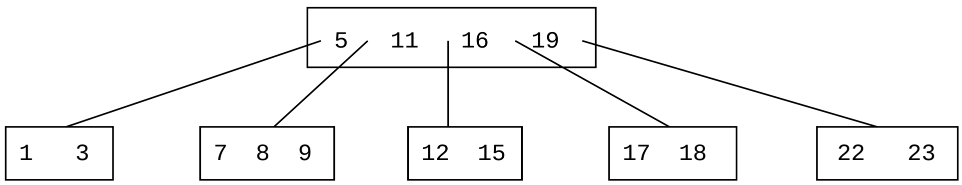
- In 2-3 Trees:
  - Each node has 1 or 2 keys
  - Each interior node has 2 or 3 children

- We can generalize 2-3 trees to allow more keys / node

20-69: **B-Trees**

- A B-Tree of maximum degree k:
  - All interior nodes have $\lceil k/2 \rceil \ldots k$ children
  - All nodes have $\lceil k/2 \rceil - 1 \ldots k - 1$ keys
- 2-3 Tree is a B-Tree of maximum degree 3

20-70: **B-Trees**

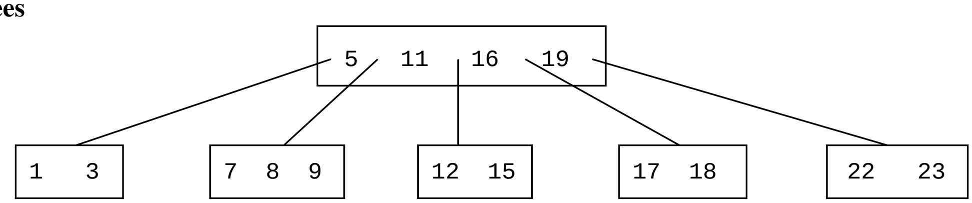


- B-Tree with maximum degree 5
  - Interior nodes have 3 – 5 children
  - All nodes have 2-4 keys

20-71: **B-Trees**

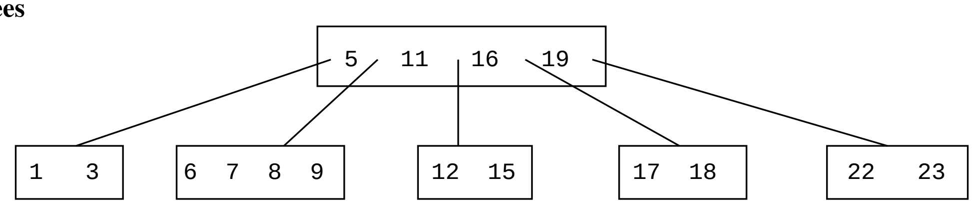
- Inserting into a B-Tree
  - Find the leaf where the element would go
  - If the leaf is not full, insert the element into the leaf
  - Otherwise, split the leaf (which may cause further splits up the tree), and insert the element

20-72: **B-Trees**

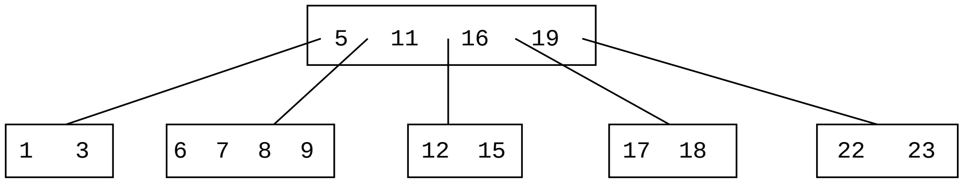


- Inserting a 6 ..

20-73: **B-Trees**

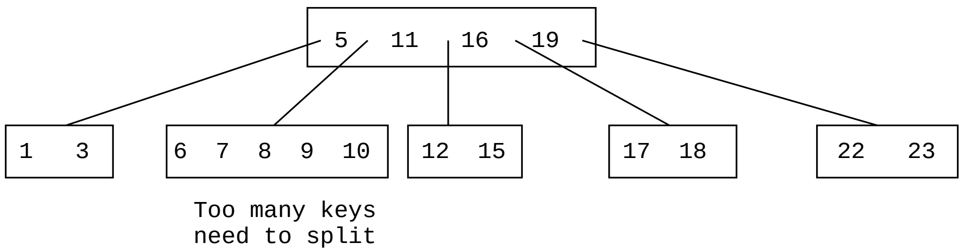


20-74: **B-Trees**



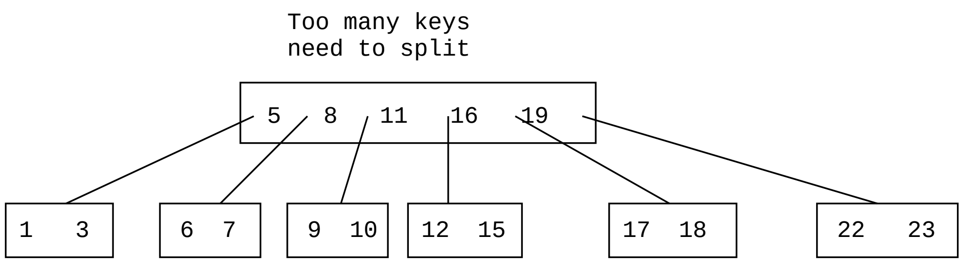
- Inserting a 10 ..

20-75: **B-Trees**



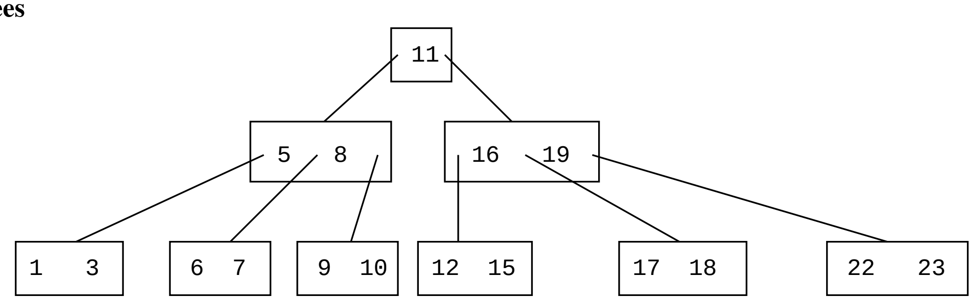
- Promote 8 to parent (between 5 and 11)
- Make nodes out of (6, 7) and (9, 10)

20-76: **B-Trees**



- Promote 11 to parent (new root)
- Make nodes out of (5, 8) and (6, 19)

20-77: **B-Trees**



- Note that the root only has 1 key, 2 children
- All nodes in B-Trees with maximum degree 5 should have at least 2 keys
- The root is an exception – it may have as few as one key and two children for any maximum degree

20-78: **B-Trees**

- B-Tree of maximum degree k
  - Generalized BST
  - All leaves are at the same depth
  - All nodes (other than the root) have $\lceil k/2 \rceil - 1 \ldots k-1$ keys
  - All interior nodes (other than the root) have $\lceil k/2 \rceil \ldots k$ children

20-79: **B-Trees**

- B-Tree of maximum degree k
  - Generalized BST
  - All leaves are at the same depth
  - All nodes (other than the root) have $\lceil k/2 \rceil - 1 \ldots k-1$ keys
  - All interior nodes (other than the root) have $\lceil k/2 \rceil \ldots k$ children
- Why do we need to make exceptions for the root?

20-80: **B-Trees**

- Why do we need to make exceptions for the root?
  - Consider a B-Tree of maximum degree 5 with only one element

20-81: **B-Trees**

- Why do we need to make exceptions for the root?
  - Consider a B-Tree of maximum degree 5 with only one element
  - Consider a B-Tree of maximum degree 5 with 5 elements

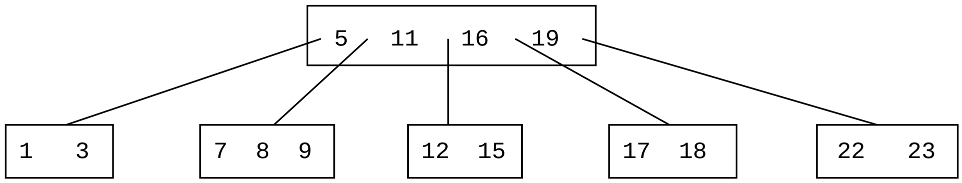
20-82: **B-Trees**

- Why do we need to make exceptions for the root?
  - Consider a B-Tree of maximum degree 5 with only one element
  - Consider a B-Tree of maximum degree 5 with 5 elements
  - Even when a B-Tree *could* be created for a specific number of elements, creating an exception for the root allows our split/merge algorithm to work correctly.

20-83: **B-Trees**

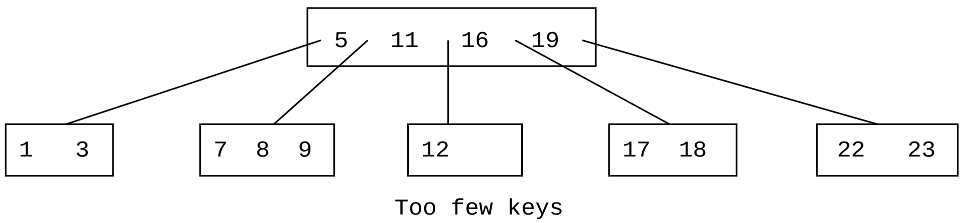
- Deleting from a B-Tree (Key is in a leaf)
  - Remove key from leaf
  - Steal / Split as necessary
  - May need to split up tree as far as root

20-84: **B-Trees**

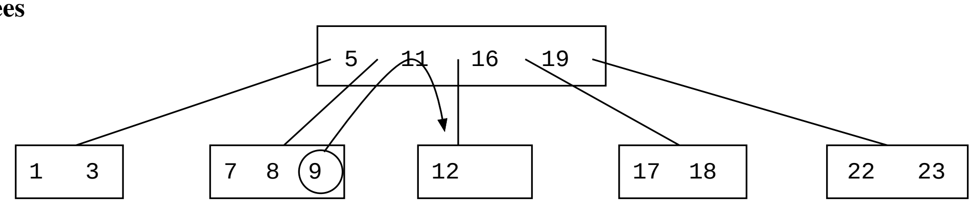


- Deleting the 15

20-85: **B-Trees**

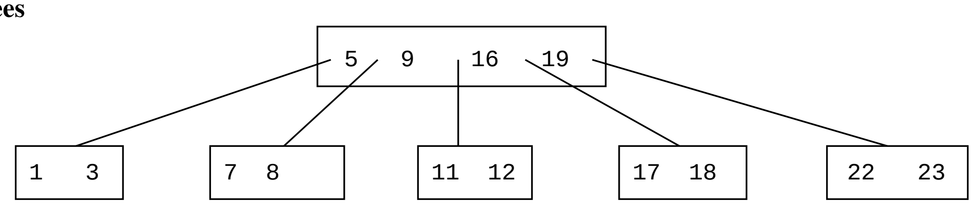


20-86: **B-Trees**

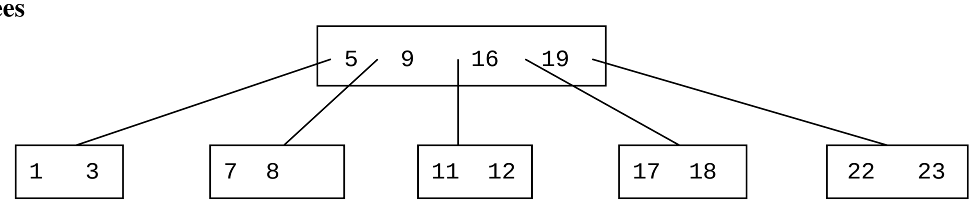


- Steal a key from sibling

20-87: **B-Trees**

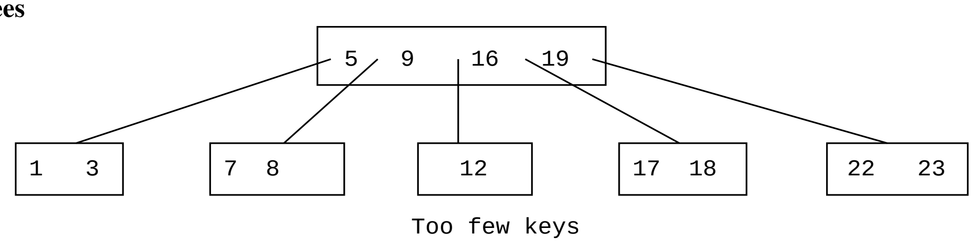


20-88: **B-Trees**

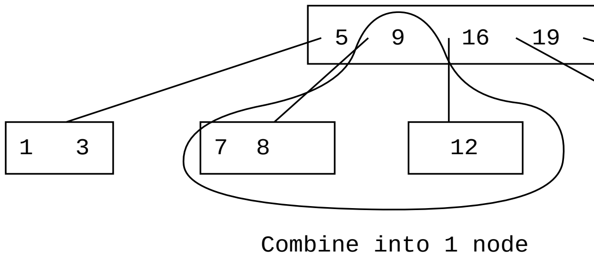


- Delete the 11

20-89: **B-Trees**

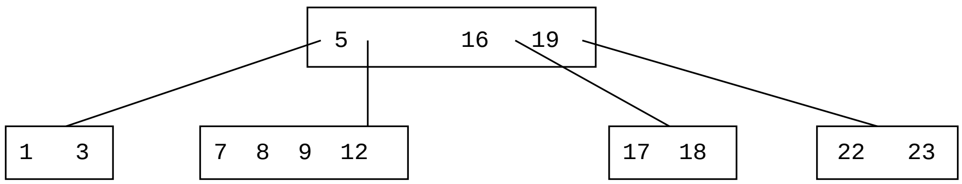


20-90: **B-Trees**



- Merge with a sibling (pick the left sibling arbitrarily)

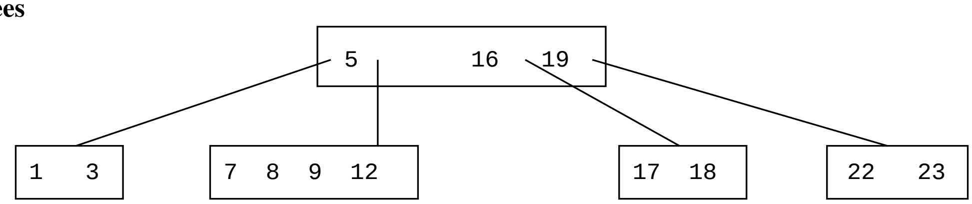
20-91: **B-Trees**



20-92: **B-Trees**

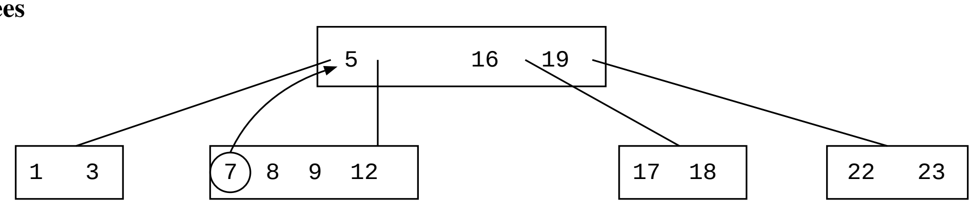
- Deleting from a B-Tree (Key in internal node)
  - Replace key with largest key in right subtree
  - Remove largest key from right subtree
  - (May force steal / merge)

20-93: **B-Trees**



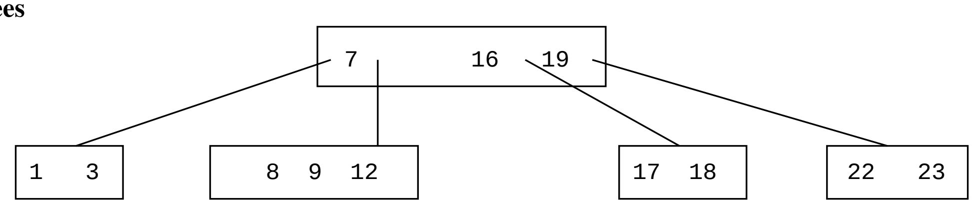
- Remove the 5

20-94: **B-Trees**

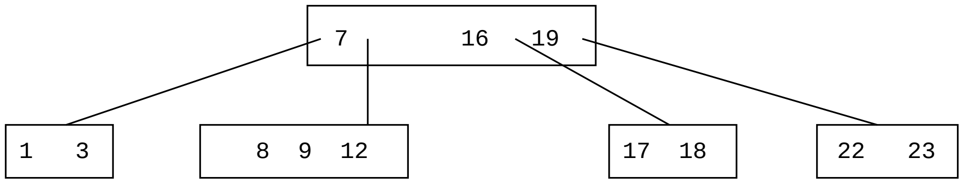


- Remove the 5

20-95: **B-Trees**

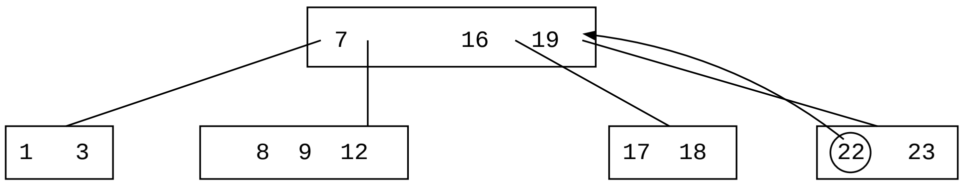


20-96: **B-Trees**



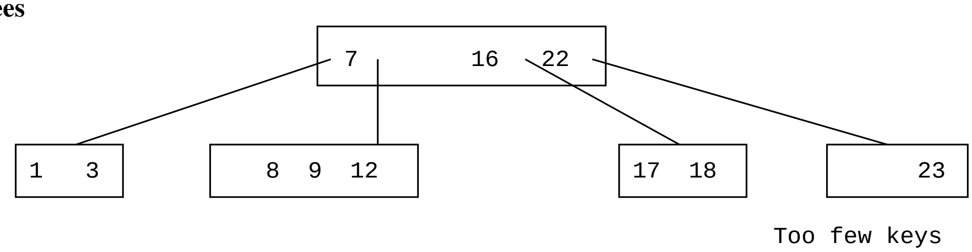
- Remove the 19

20-97: **B-Trees**

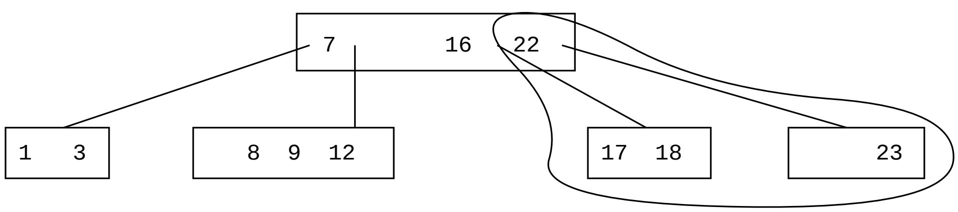


- Remove the 19

20-98: **B-Trees**

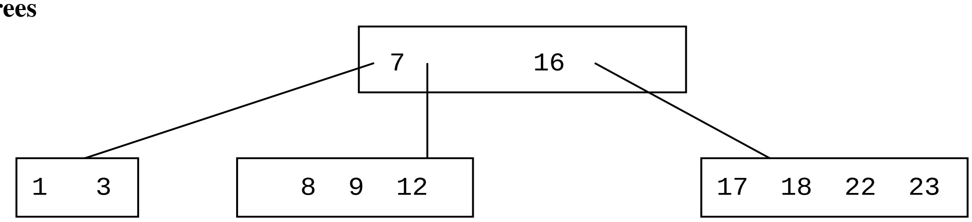


20-99: **B-Trees**



- Merge with left sibling

20-100: **B-Trees**



20-101: **B-Trees**

- Almost all databases that are large enough to require storage on disk use B-Trees
- Disk accesses are *very* slow
    - Accessing a byte from disk is 10,000 – 100,000 times as slow as accessing from main memory

- Recently, this gap has been getting even bigger

- Compared to disk accesses, all other operations are essentially free
- Most efficient algorithm minimizes disk accesses as much as possible

20-102: **B-Trees**

- Disk accesses are slow – want to minimize them
- Single disk read will read an entire sector of the disk
- Pick a maximum degree k such that a node of the B-Tree takes up exactly one disk block
  - Typically on the order of 100 children / node

20-103: **B-Trees**

- With a maximum degree around 100, B-Trees are very shallow
- Very few disk reads are required to access any piece of data
- Can improve matters even more by keeping the first few levels of the tree in main memory
  - For large databases, we can't store the entire tree in main memory – but we can limit the number of disk accesses for each operation to only 1 or 2