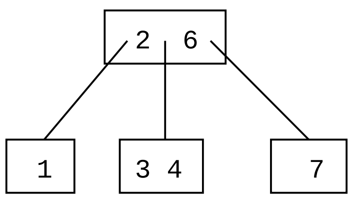
11-0: **Binary Search Trees**

- Binary Tree data structure
- All values in left subtree < value stored in root
- All values in the right subtree > value stored in root

11-1: **Generalizing BSTs**

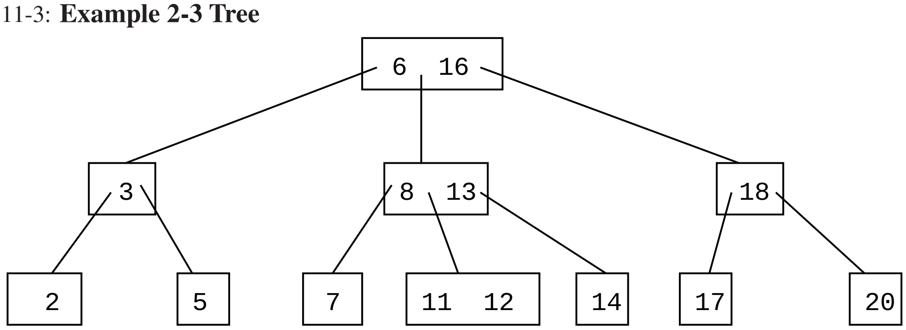
- Generalized Binary Search Trees
  - Each node can store several keys, instead of just one
  - Values in subtrees between values in surrounding keys
  - For non leaves, # of children = # of keys + 1



11-2: **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

11-3: **Example 2-3 Tree**



11-4: **Finding in 2-3 Trees**

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

11-5: **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

11-6: **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

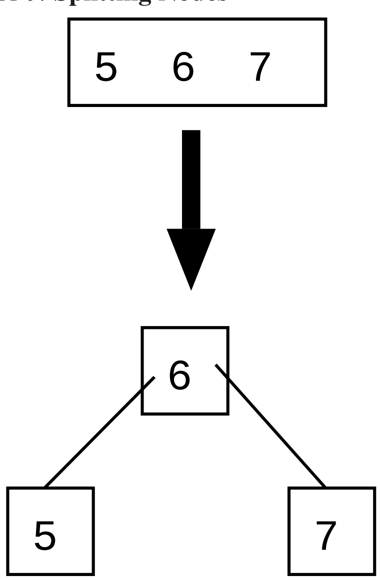
11-7: **Inserting into 2-3 Trees**

- 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
    - What if the leaf already has 2 elements?

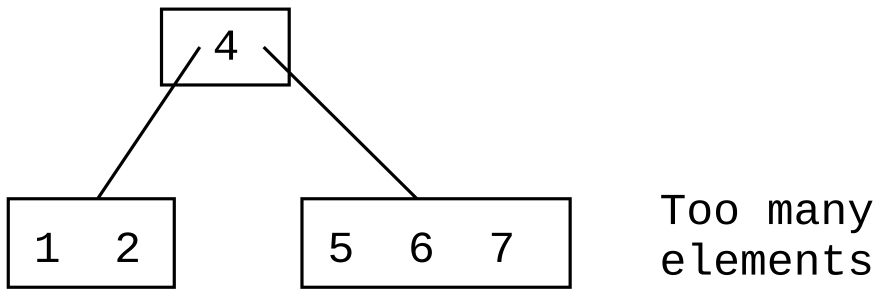
11-8: **Inserting into 2-3 Trees**

- 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?
    - Split!

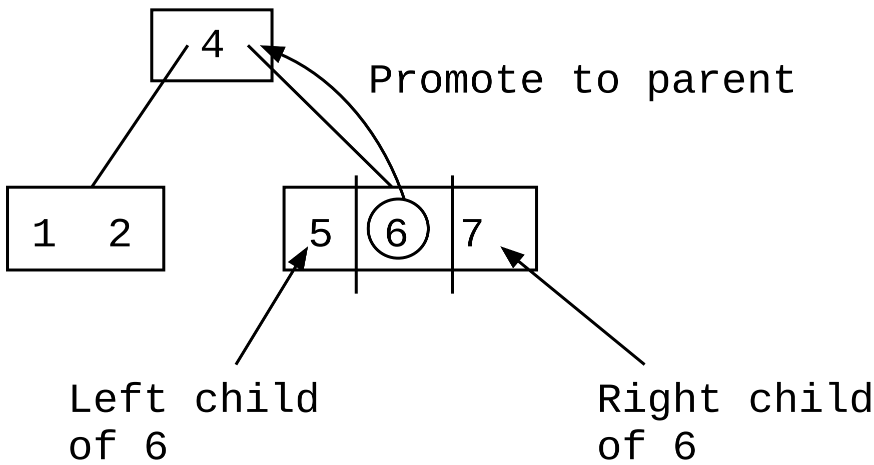
11-9: **Splitting Nodes**



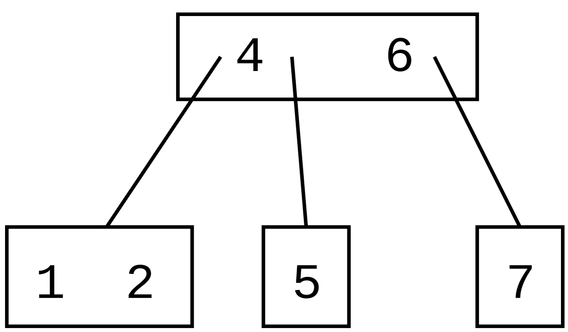
11-10: **Splitting Nodes**



11-11: **Splitting Nodes**



11-12: **Splitting Nodes**

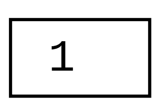


11-13: **Splitting Root**

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

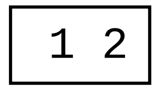
11-14: **2-3 Tree Example**

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



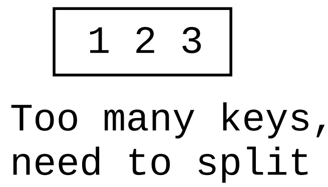
11-15: **2-3 Tree Example**

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



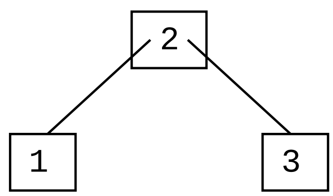
11-16: **2-3 Tree Example**

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



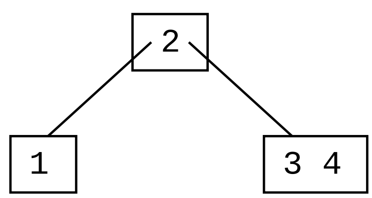
11-17: **2-3 Tree Example**

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



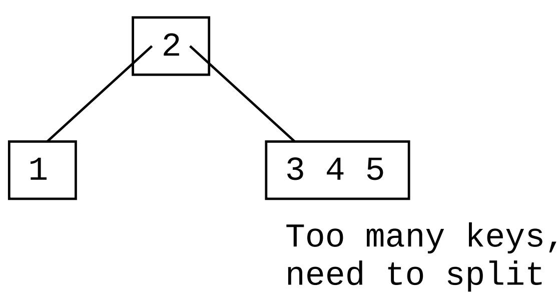
11-18: **2-3 Tree Example**

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



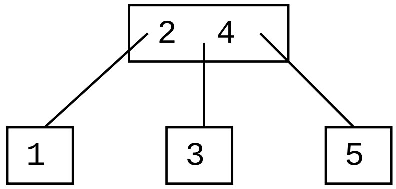
11-19: **2-3 Tree Example**

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



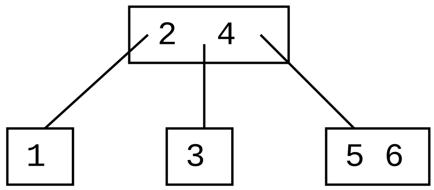
11-20: **2-3 Tree Example**

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



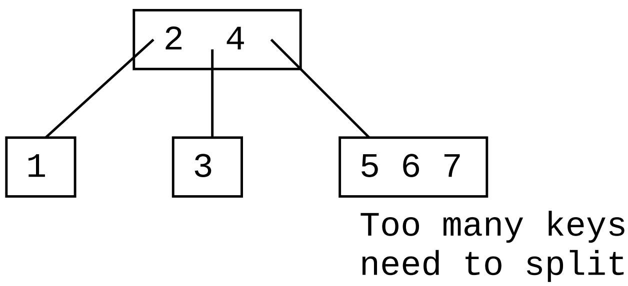
11-21: **2-3 Tree Example**

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



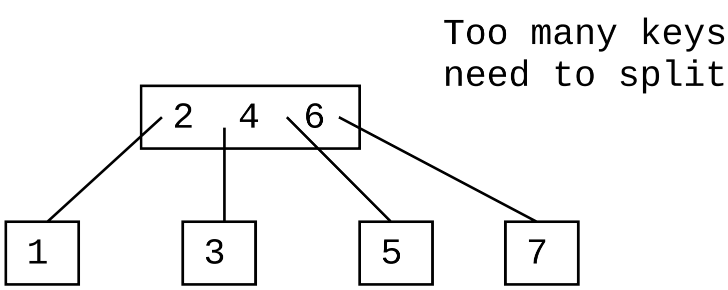
11-22: **2-3 Tree Example**

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



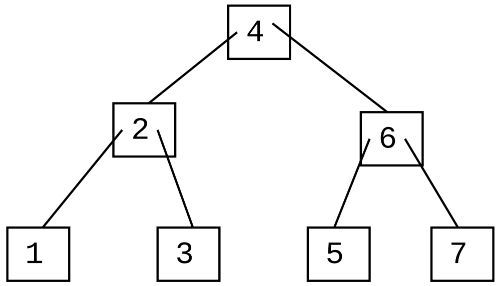
11-23: **2-3 Tree Example**

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



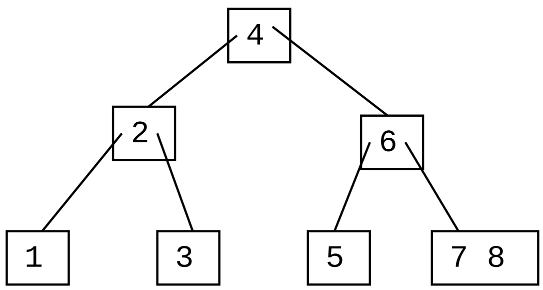
11-24: **2-3 Tree Example**

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



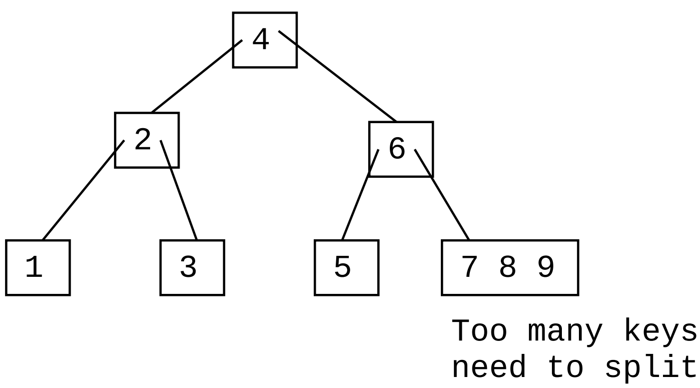
11-25: **2-3 Tree Example**

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



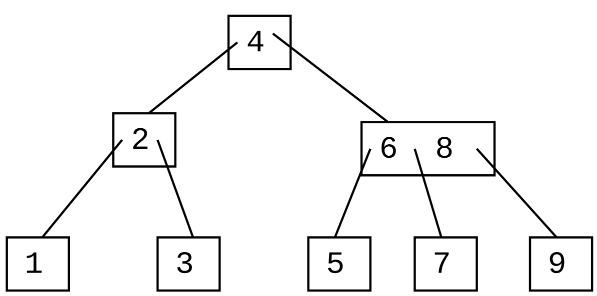
11-26: **2-3 Tree Example**

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



11-27: **2-3 Tree Example**

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



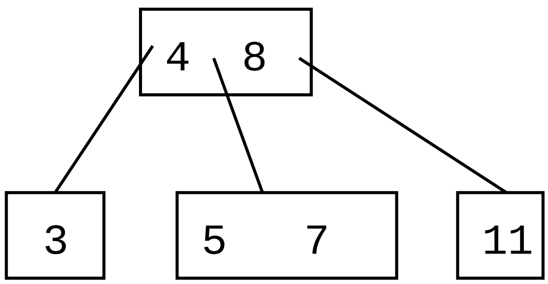
11-28: **Deleting from 2-3 Tree**

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

11-29: **Deleting Leaves**

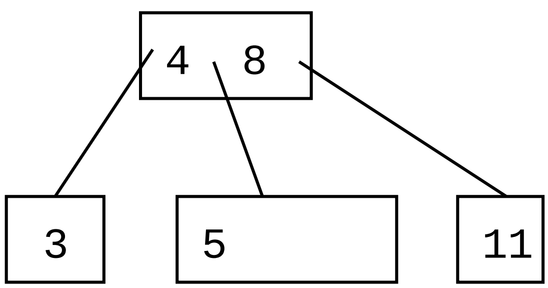
- If leaf contains 2 keys
  - Can safely remove a key

11-30: **Deleting Leaves**



- Deleting 7

11-31: **Deleting Leaves**

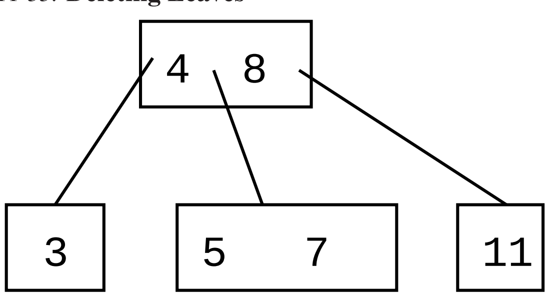


- Deleting 7

11-32: **Deleting Leaves**

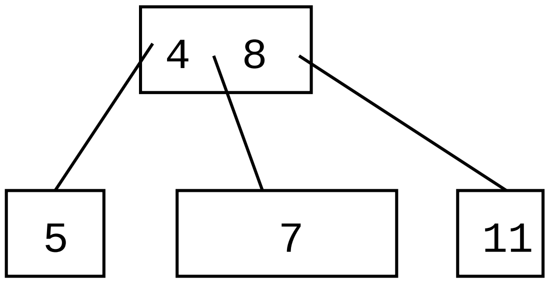
- If leaf contains 1 key
  - Cannot remove key without making leaf empty
  - Try to steal extra key from sibling

11-33: **Deleting Leaves**



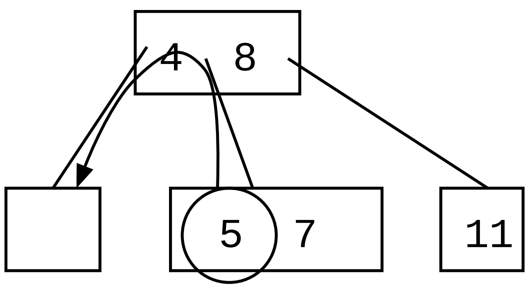
- Deleting 3 – we can steal the 5

11-34: **Deleting Leaves**



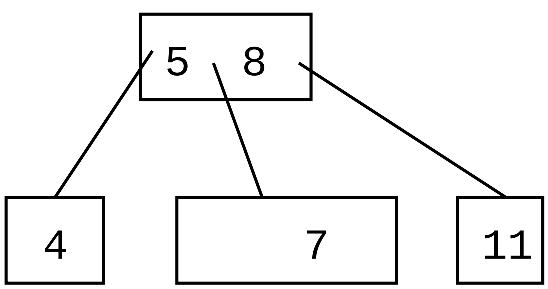
- Not a 2-3 tree. What can we do?

11-35: **Deleting Leaves**



- Steal key from sibling *through parent*

11-36: **Deleting Leaves**

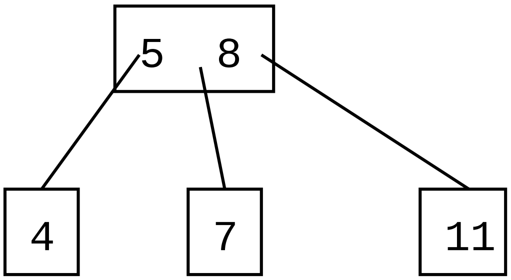


- Steal key from sibling *through parent*

11-37: **Deleting Leaves**

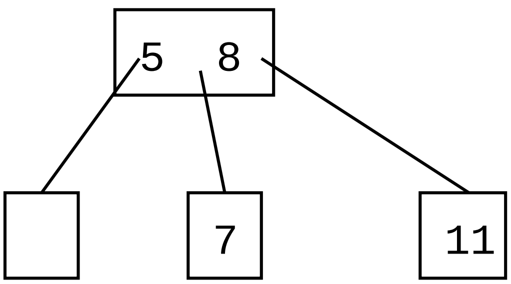
- If leaf contains 1 key, and no sibling contains extra keys
    - Cannot remove key without making leaf empty
    - Cannot steal a key from a sibling
    - Merge with sibling
        - split in reverse

11-38: **Merging Nodes**



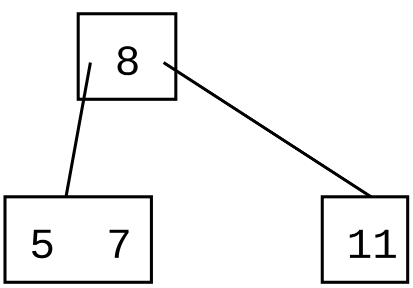
- Removing the 4

11-39: **Merging Nodes**



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

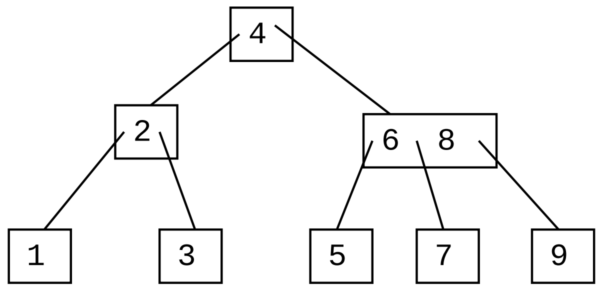
11-40: **Merging Nodes**



11-41: **Merging Nodes**

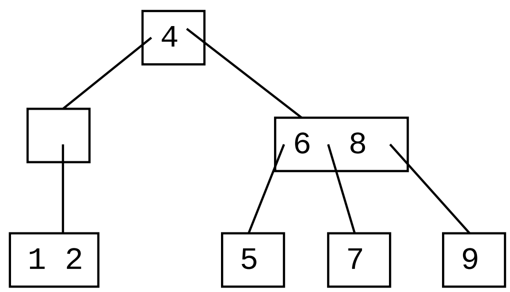
- Merge decreases the number of keys in the parent
    - May cause parent to have too few keys
- Parent can steal a key, or merge again

11-42: **Merging Nodes**



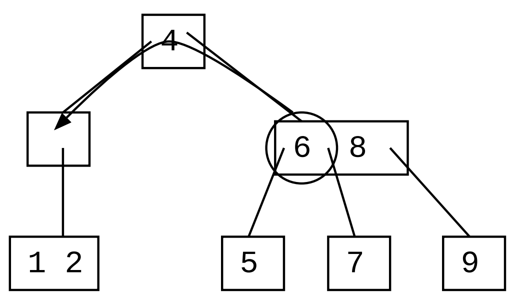
- Deleting the 3 – cause a merge

11-43: **Merging Nodes**



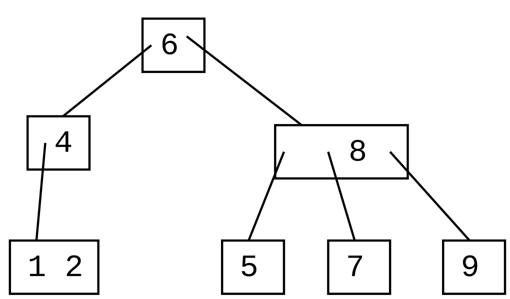
- Deleting the 3 – cause a merge
- Not enough keys in parent

11-44: **Merging Nodes**



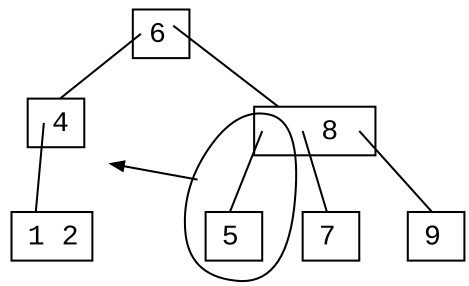
- Steal key from sibling

11-45: **Merging Nodes**



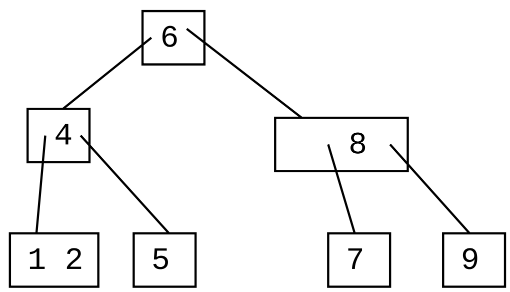
- Steal key from sibling

11-46: **Merging Nodes**



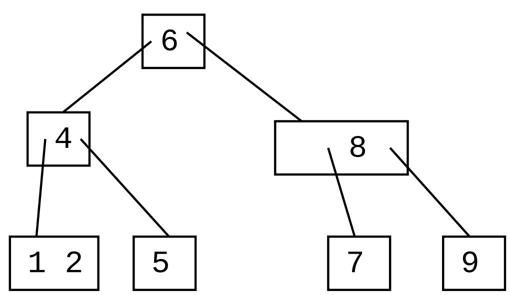
- When we steal a key from an internal node, steal nearest subtree as well

11-47: **Merging Nodes**



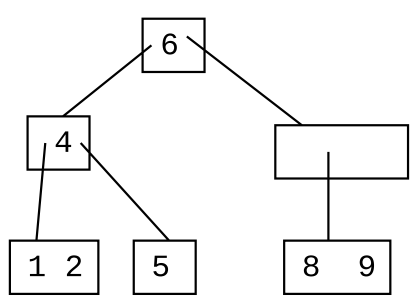
- When we steal a key from an internal node, steal nearest subtree as well

11-48: **Merging Nodes**



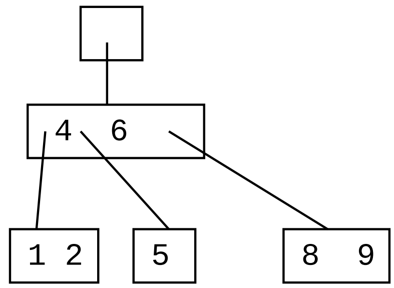
- Deleting the 7 – cause a merge

11-49: **Merging Nodes**



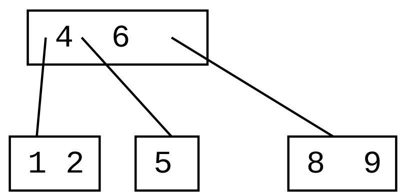
- Parent has too few keys – merge again

11-50: **Merging Nodes**



- Root has no keys – delete

11-51: **Merging Nodes**



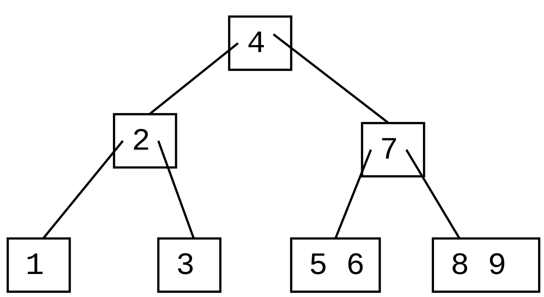
11-52: **Deleting Interior Keys**

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

11-53: **Deleting Interior Keys**

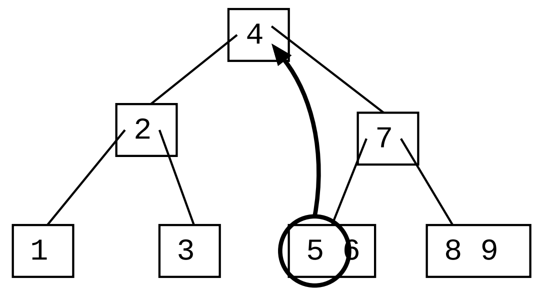
- How can we delete keys from non-leaf nodes?
  - Replace key with smallest element subtree to right of key
  - Recursivly delete smallest element from subtree to right of key
- (can also use largest element in subtree to left of key)

11-54: **Deleting Interior Keys**



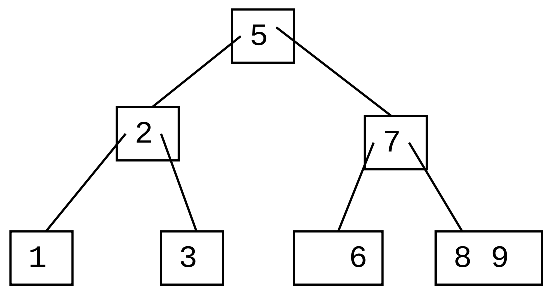
- Deleting the 4

11-55: **Deleting Interior Keys**

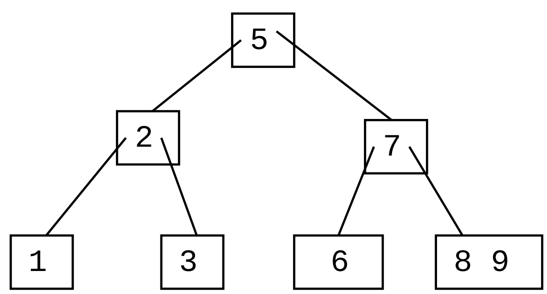


- Deleting the 4
- Replace 4 with smallest element in tree to right of 4

11-56: **Deleting Interior Keys**

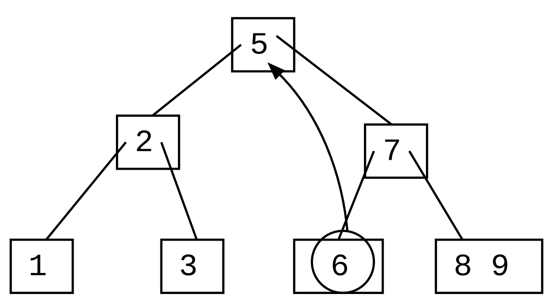


11-57: **Deleting Interior Keys**



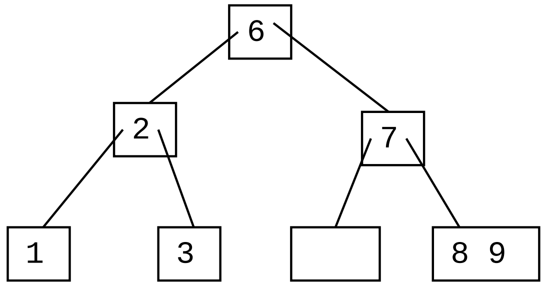
- Deleting the 5

11-58: **Deleting Interior Keys**



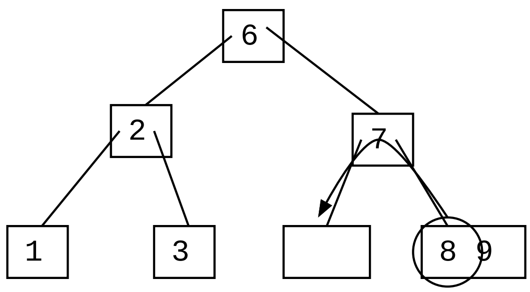
- Deleting the 5
- Replace the 5 with the smallest element in tree to right of 5

11-59: **Deleting Interior Keys**



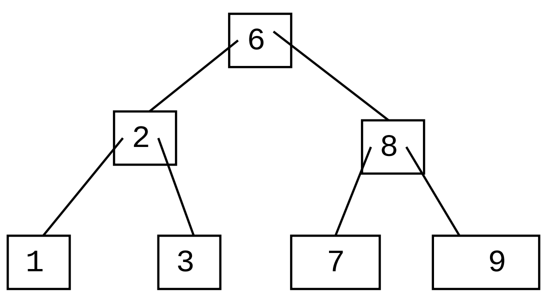
- Deleting the 5
- Replace the 5 with the smallest element in tree to right of 5
- Node with two few keys

11-60: **Deleting Interior Keys**

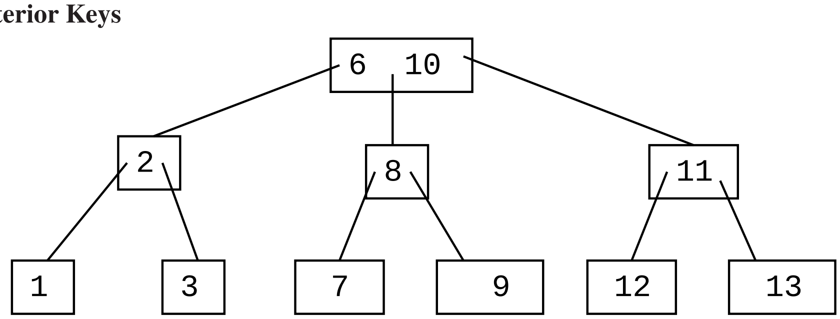


- Node with two few keys
- Steal a key from a sibling

11-61: **Deleting Interior Keys**

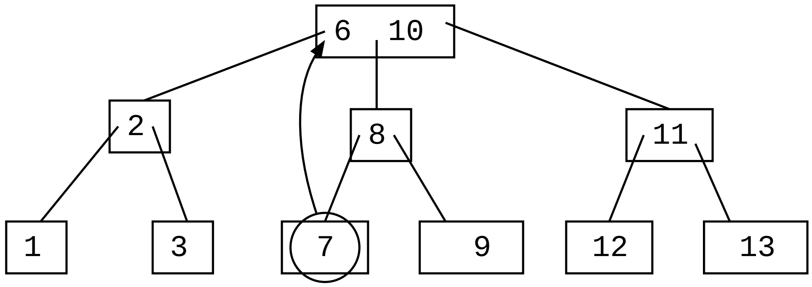


11-62: **Deleting Interior Keys**



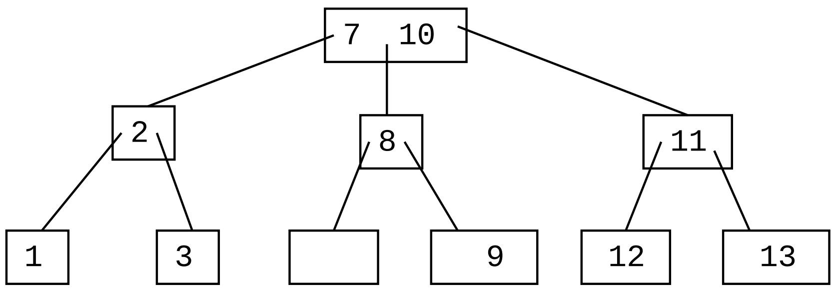
- Removing the 6

11-63: **Deleting Interior Keys**



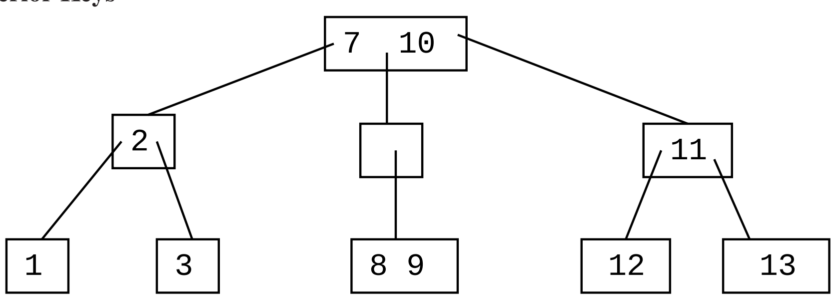
- Removing the 6
- Replace the 6 with the smallest element in the tree to the right of the 6

11-64: **Deleting Interior Keys**



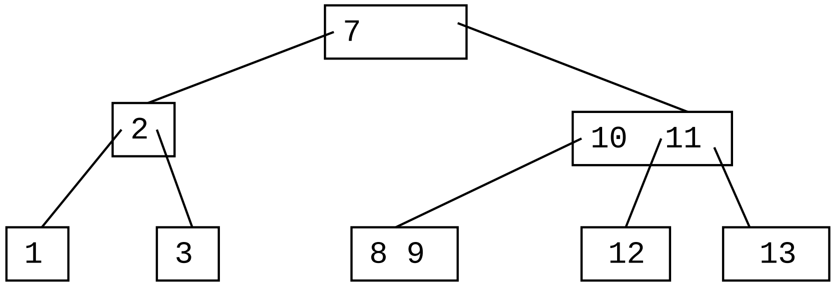
- Node with too few keys
  - Can't steal key from sibling
  - Merge with sibling

11-65: **Deleting Interior Keys**



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

11-66: **Deleting Interior Keys**



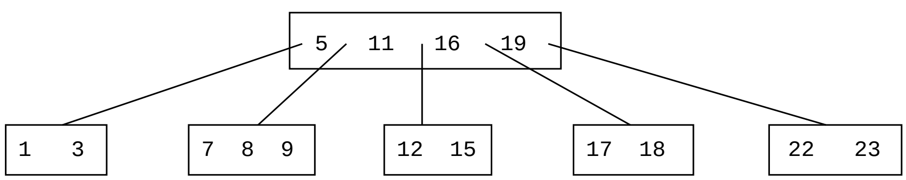
11-67: **Generalizing 2-3 Trees**

- 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

11-68: **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

11-69: **B-Trees**

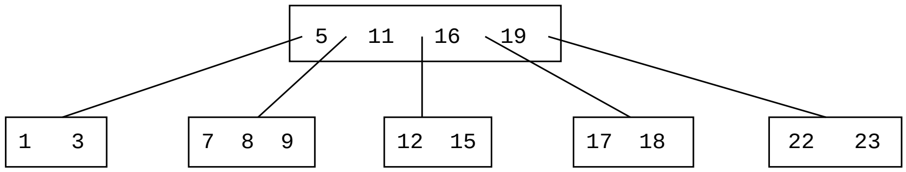


- B-Tree with maximum degree 5
  - Interior nodes have 3 – 5 children
  - All nodes have 2-4 keys

11-70: **B-Trees**

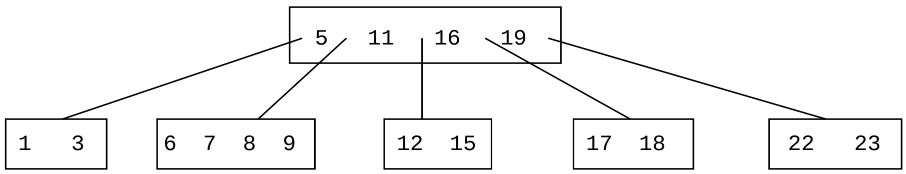
- Inserting into a B-Tree
  - Find the leaf where the element would go
  - If the leaf is not full, insert the element into the leaf
  - Otherwise, split the leaf (which may cause further splits up the tree), and insert the element

11-71: **B-Trees**

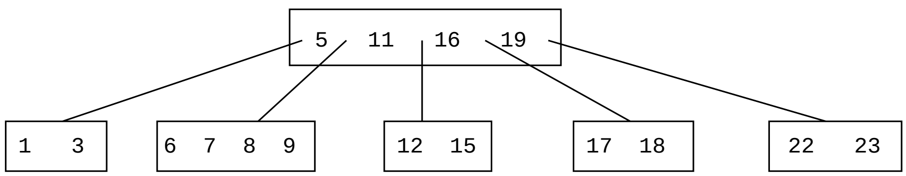


- Inserting a 6 ..

11-72: **B-Trees**

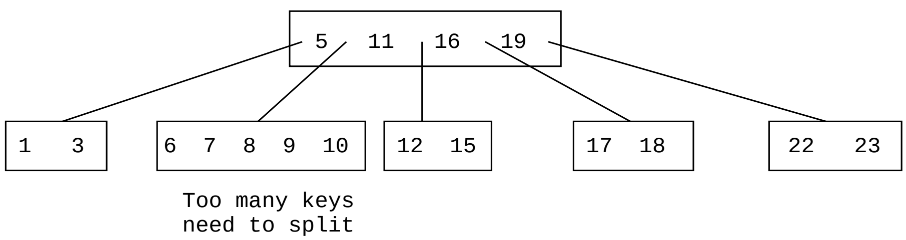


11-73: **B-Trees**



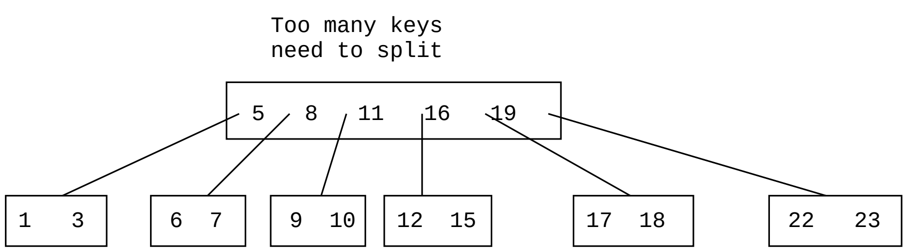
- Inserting a 10 ..

11-74: **B-Trees**



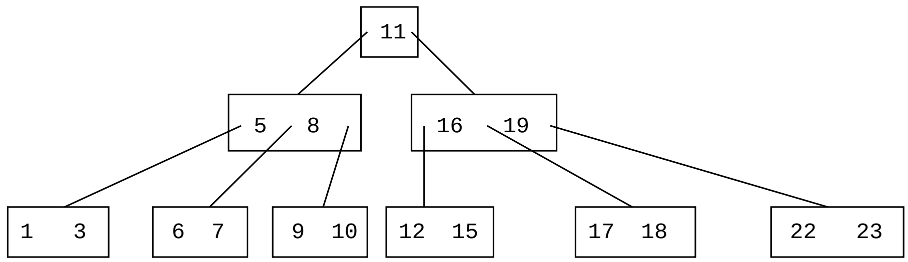
- Promote 8 to parent (between 5 and 11)
- Make nodes out of (6, 7) and (9, 10)

11-75: **B-Trees**



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

11-76: **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

11-77: **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

11-78: **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
- Why do we need to make exceptions for the root?

11-79: **B-Trees**

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

11-80: **B-Trees**

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

11-81: **B-Trees**

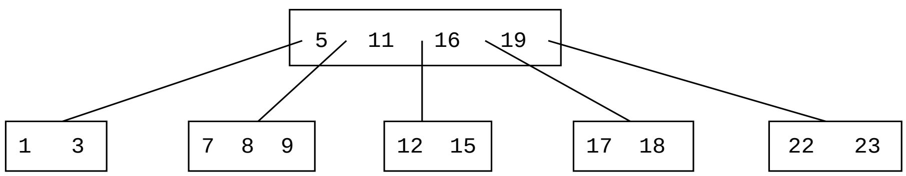
- Why do we need to make exceptions for the root?

- Consider a B-Tree of maximum degree 5 with only one element
- Consider a B-Tree of maximum degree 5 with 5 elements
- 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.

11-82: **B-Trees**

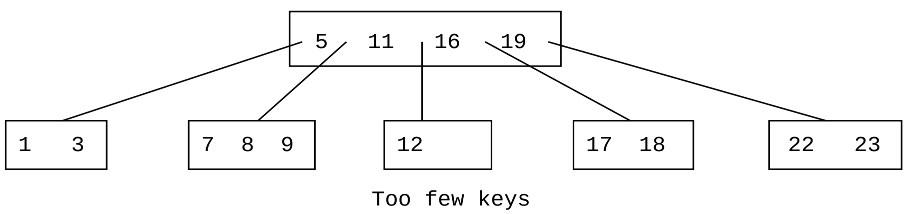
- Deleting from a B-Tree (Key is in a leaf)
  - Remove key from leaf
  - Steal / Split as necessary
  - May need to split up tree as far as root

11-83: **B-Trees**

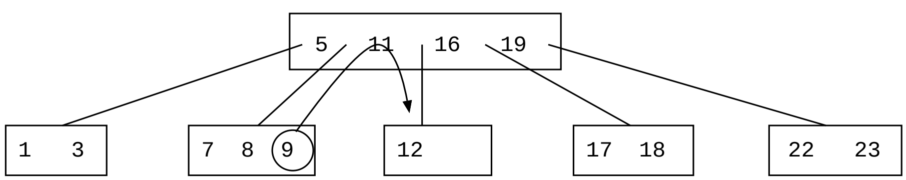


- Deleting the 15

11-84: **B-Trees**

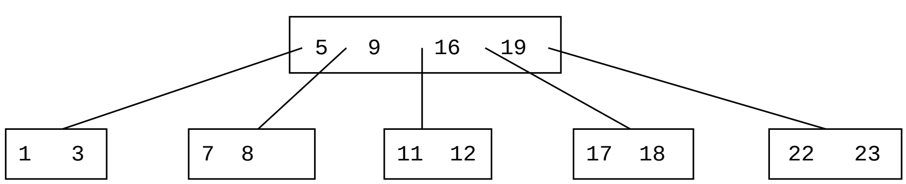


11-85: **B-Trees**

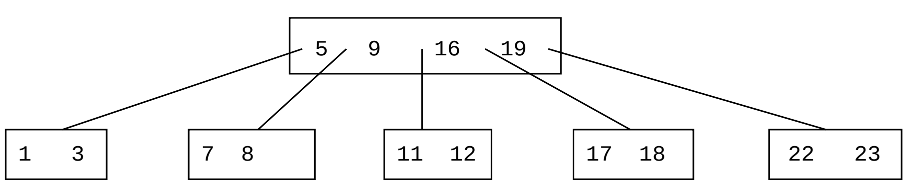


- Steal a key from sibling

11-86: **B-Trees**

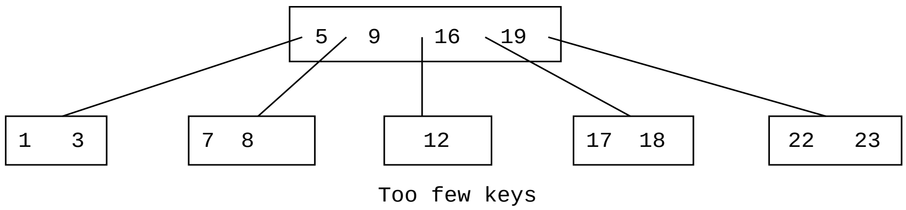


11-87: **B-Trees**

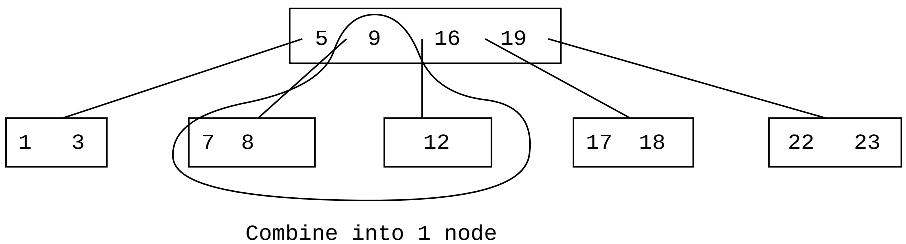


- Delete the 11

11-88: **B-Trees**

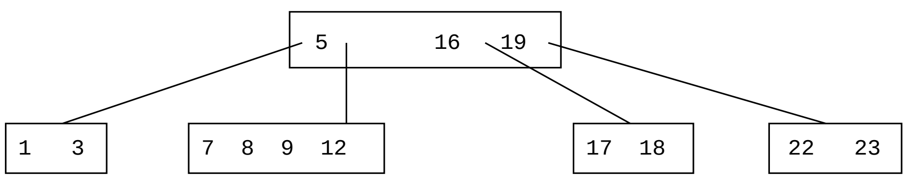


11-89: **B-Trees**



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

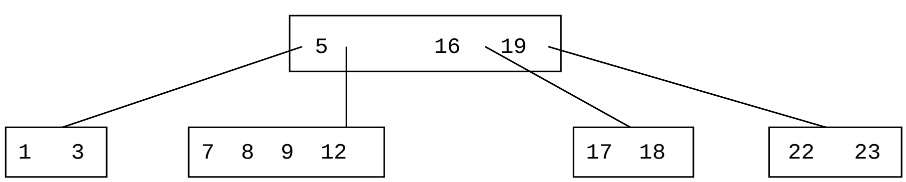
11-90: **B-Trees**



11-91: **B-Trees**

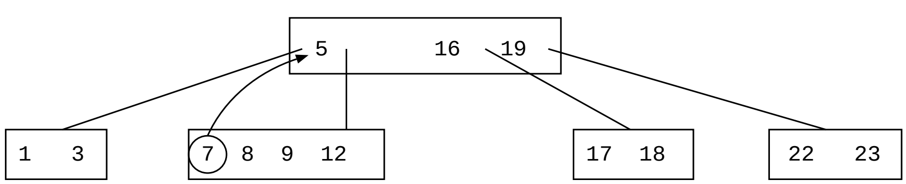
- Deleting from a B-Tree (Key in internal node)
  - Replace key with largest key in right subtree
  - Remove largest key from right subtree
  - (May force steal / merge)

11-92: **B-Trees**



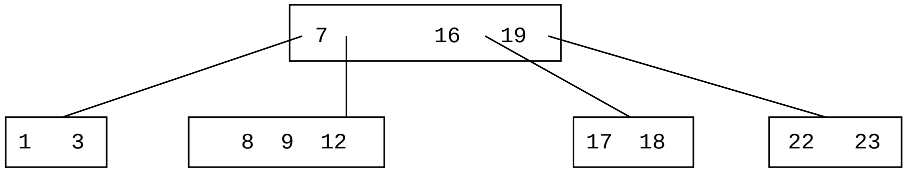
- Remove the 5

11-93: **B-Trees**

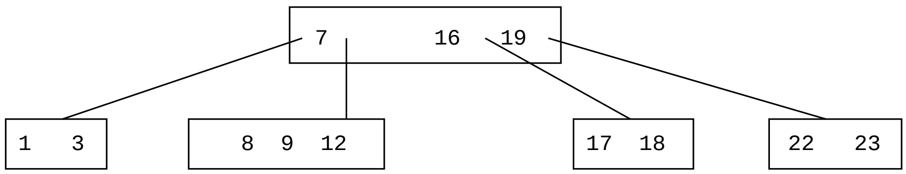


- Remove the 5

11-94: **B-Trees**

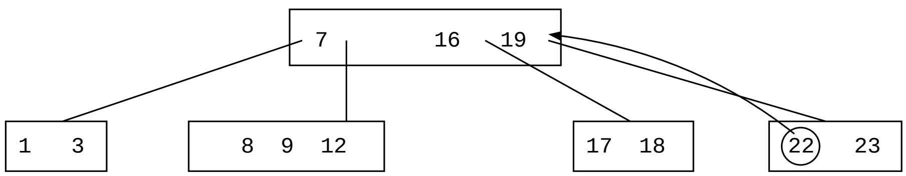


11-95: **B-Trees**



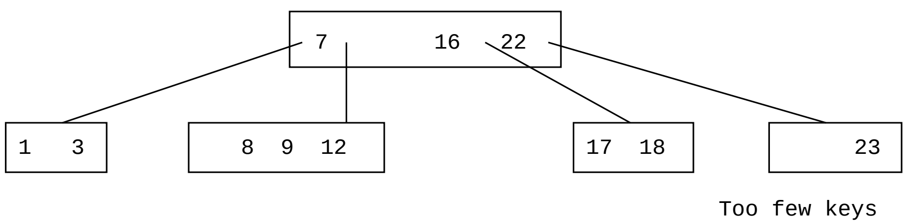
- Remove the 19

11-96: **B-Trees**

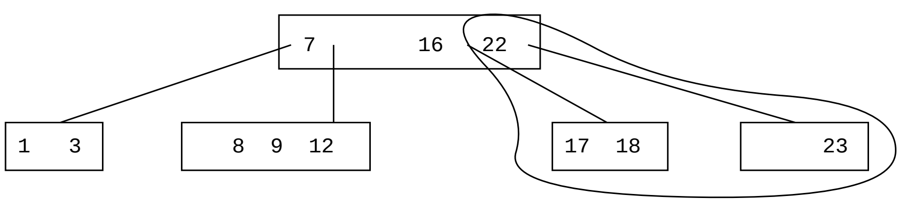


- Remove the 19

11-97: **B-Trees**

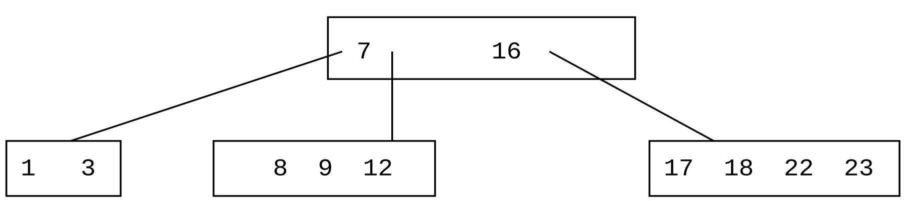


11-98: **B-Trees**



- Merge with left sibling

11-99: **B-Trees**



11-100: **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

11-101: **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

11-102: **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 be very small

11-103: **B-Trees**

- If the maximum degree of a B-Tree is odd (2-3 tree, 3-4-5 tree), then we can only split a node when it gets "over-full"
  - Examples for 2-3 trees on board
- If the maximum degree of a B-Tree is even (2-3-4 tree, 3-4-5-6, etc.):
  - We can split a node before it is "over-full"
  - We can merge nodes before they are "under-full"

11-104: **B-Trees**

- Preemptive Splitting
  - If the maximum degree is even, we can implement an insert with a single pass down the tree (instead of a pass down, and then a pass up to clean up)
  - When inserting into any subtree tree, if the root of that tree is full, split the root before inserting
    - Every time we want to do a split, we know our parent is not full.

(examples, use visualization) 11-105: **B-Trees**

- Preemptive Combining – Deleting from Leaves

- If the maximum degree is even, we can implement a delete with a single pass down the tree (instead of a pass down, and then a pass up to clean up)
- When deleting from any node (other than the root), combine / steal as necessary so that the node has more then the minimum # of keys
- When you get to a leaf, you are guaranteed that there will be an extra key in the leaf

(examples, deleting from leaves)

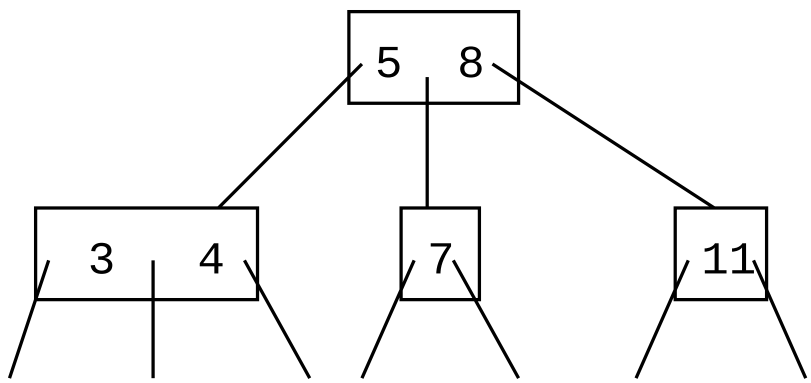
11-106: **B-Trees**

- Preemptive Combining
  - Deleting k from a non-leaf:
    - If the subtree left of k has $>$ minimum number of elements, replace k with largest element in the left subtree, splitting as you go down
    - If the subtree right of k has $>$ minimum number of elements, replace k with smallest element in the right subtree, splitting as you go down

(examples)

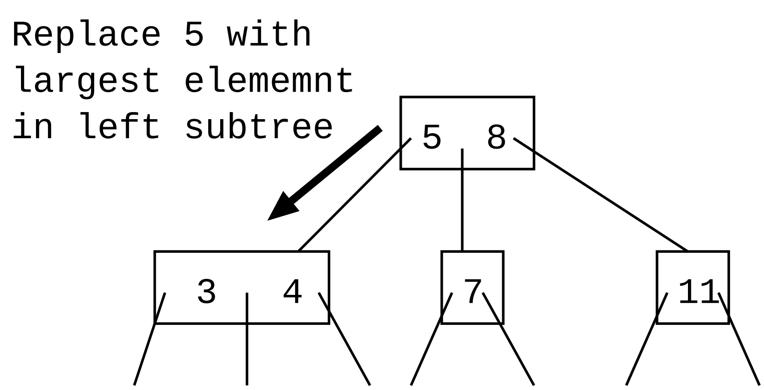
11-107: **B-Trees**

Deleting 5:



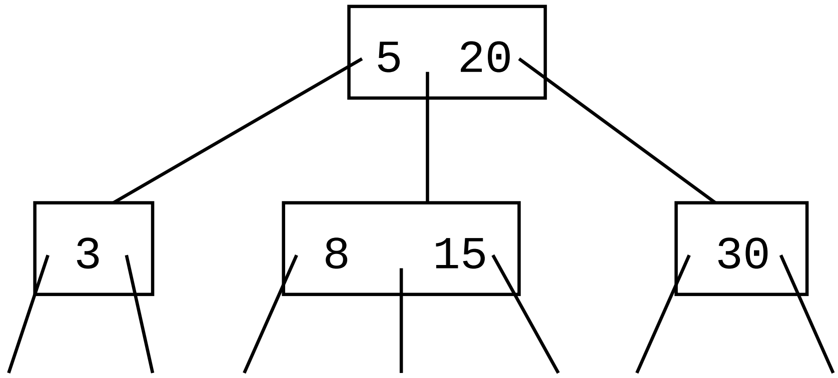
11-108: **B-Trees**

Deleting 5:



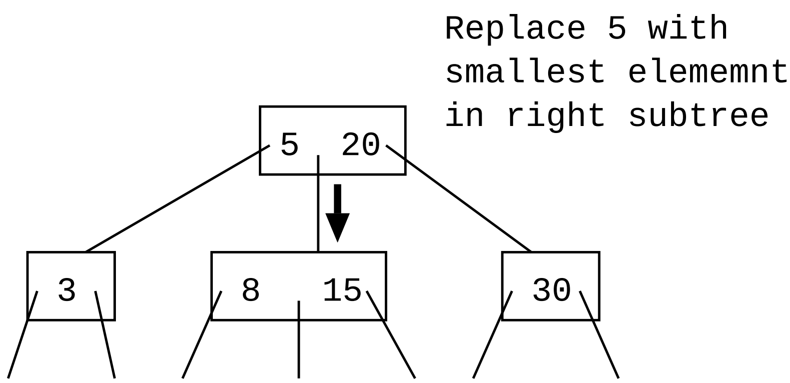
11-109: **B-Trees**

Deleting 5:



11-110: **B-Trees**

Deleting 5:

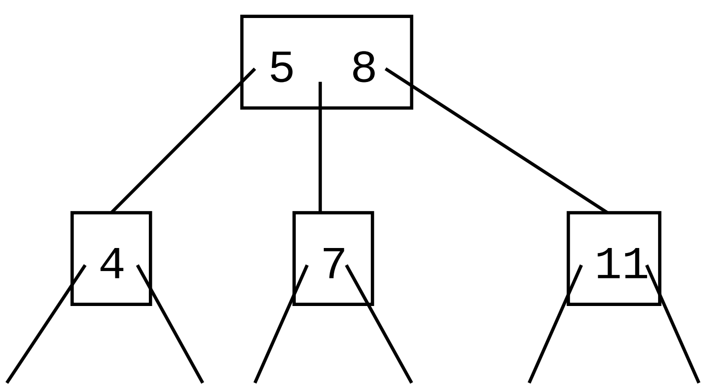


11-111: **B-Trees**

- Preemptive Combining
  - Deleting k from a non-leaf:
    - If the subtrees to the left & right of k subtrees both have the minimum # of elements, combine around k
    - Recursively remove k from this new node

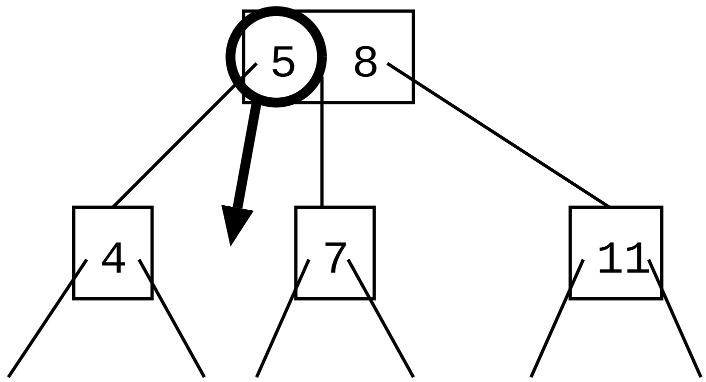
11-112: **B-Trees**

Deleting 5:



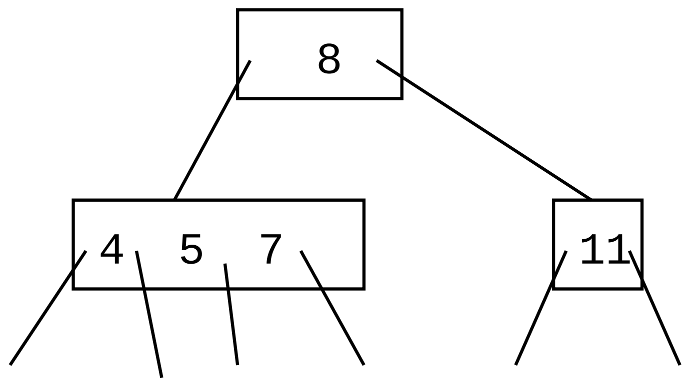
11-113: **B-Trees**

Merge around 5:



11-114: **B-Trees**

Delete 5 from new node:



11-115: **B-Trees**

- Preemptive Combining
  - Deleting k from a non-leaf:
    - If the subtrees to the left & right of k subtrees both have the minimum # of elements, combine around k
    - Recursively remove k from this new node
- Why do we need this case? Why can't we just replace key with largest value in left subtree, or smallest value in right subtree?

11-116: **B-Trees**

- Preemptive Combining
  - Deleting k from a non-leaf:
    - If the subtrees to the left & right of k subtrees both have the minimum # of elements, combine around k
    - Recursively remove k from this new node
- Why do we need this case? Why can't we just replace key with largest value in left subtree?
  - Immediately cause a merge, anyway
  - Harder to determine which location to copy largest element into

11-117: **B-Trees**

- Preemptive split/merge vs. “standard” split/merge
  - Advantages of the “standard” method?
  - Advantages of the “preemptive” method?
- Textbook uses “preemptive” method
  - Defines “minimum degree k” (with maximum degree = $2k$) instead of “maximum degree k” (with minimum degree = $\lceil \frac{k}{2} \rceil$)