DATA STRUCTURES PART III

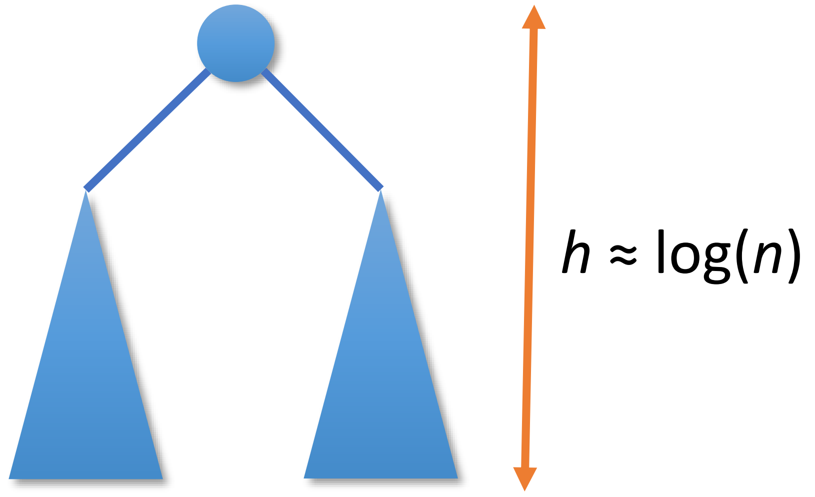
Sid Chi-Kin Chau

[Lecture 4]

Recap from Previous Lecture

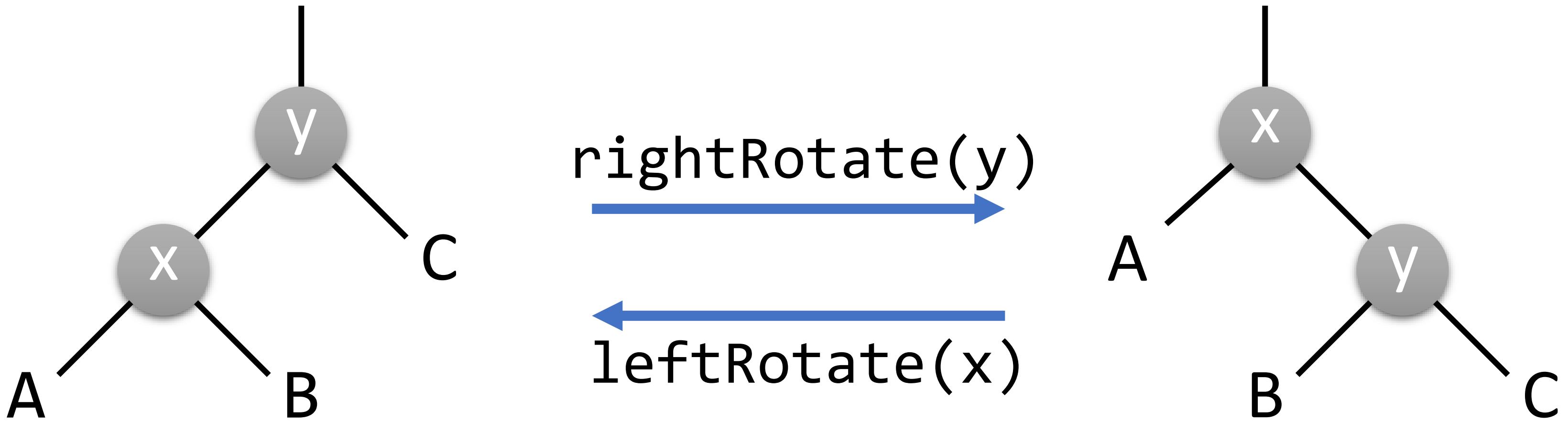
- Balanced search tree
  - Belong to binary search tree
  - But with a height of O(log(n)) guaranteed for n items
  - Height h = maximum number of **edges** from the root to a leaf

- Examples
  - Red-black tree
  - AVL tree
  - B-tree



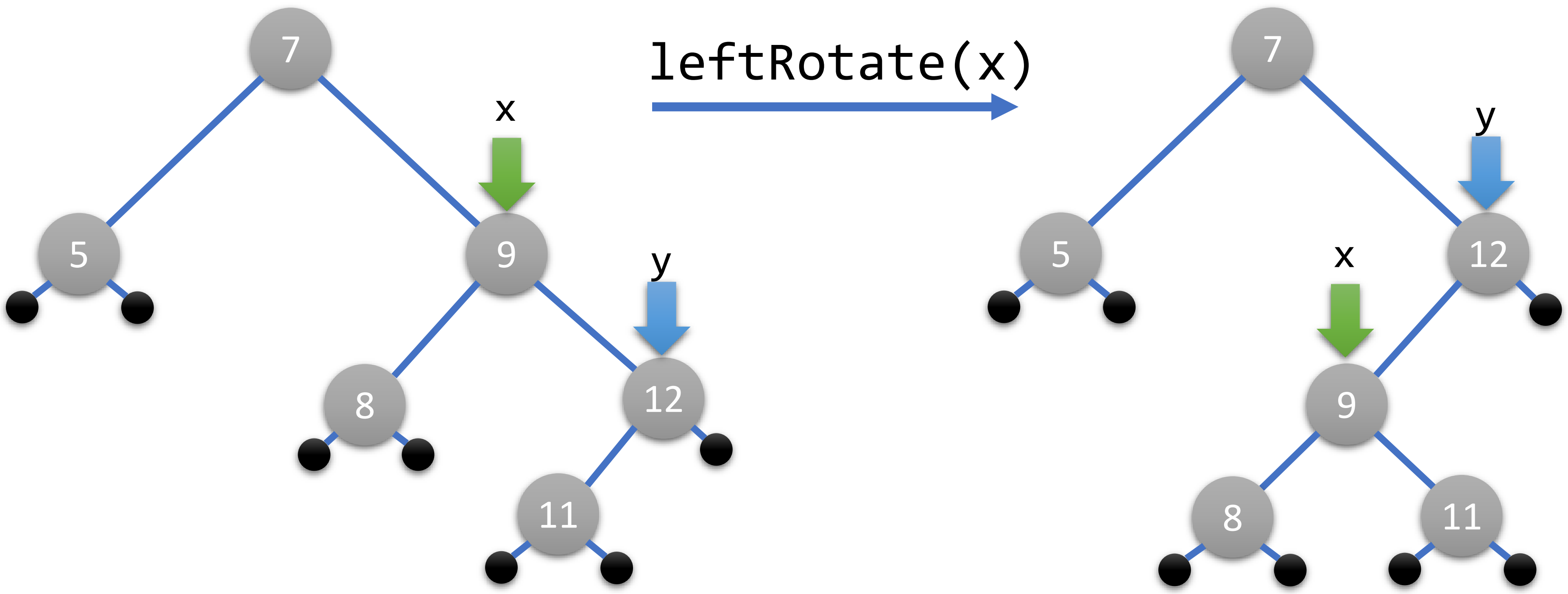
Tree Rotation

- Our basic operation for changing tree structure is called rotation
  - Rotation preserves inorder key ordering
  - How would tree rotation actually work?



Example

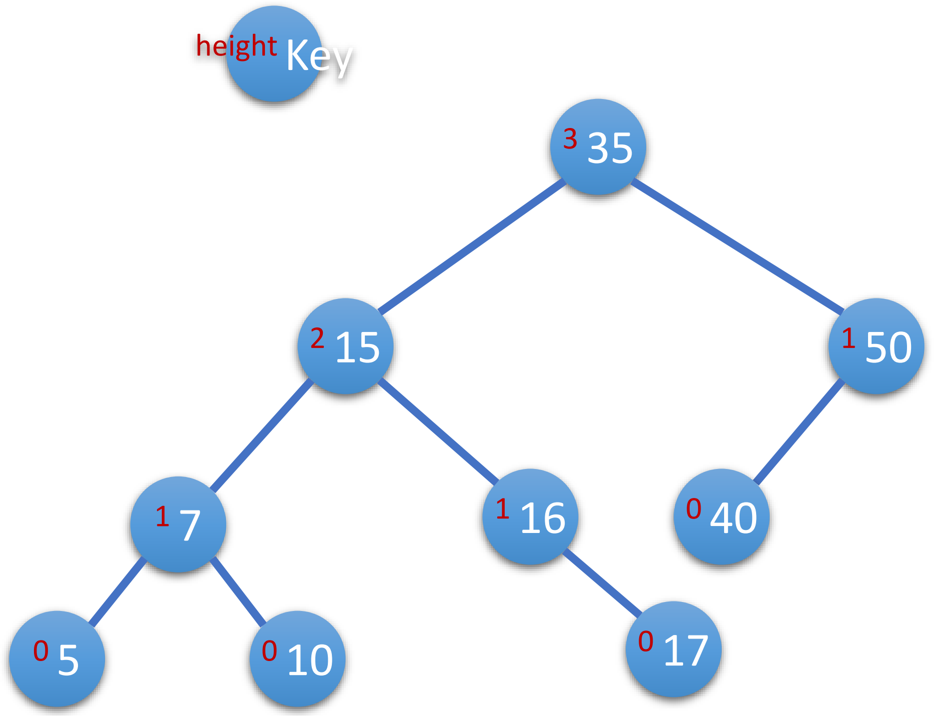
- Left Rotate at node 9

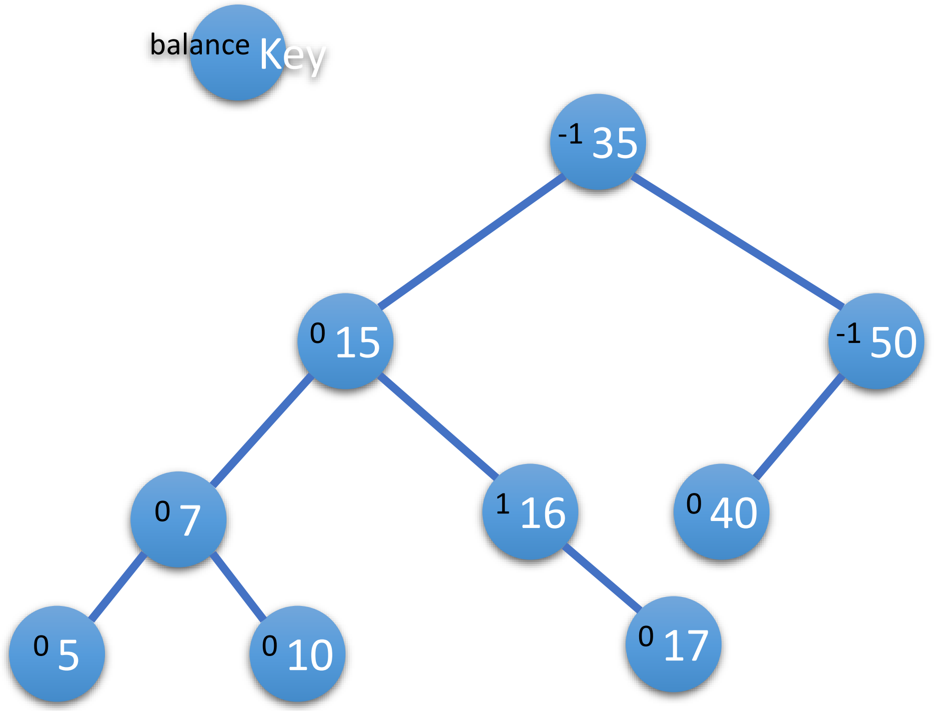


AVL Tree

- AVL tree is binary search tree that has a height difference between with left and right subtrees of a node is at most 1
  - In BST, left subtree keys are less than the root and right subtree keys are greater than the root
  - Each node has a height of the subtree rooted at that node
- Balance:
  - Define the balance of a node n
    - b(n) = Height with only Right Subtree - Height with only Left Subtree
    - Legal values for AVL tree: b(n) in {-1, 0, 1}
      - Otherwise, b(n) > 1 or b(n) < -1, then it is an unbalanced AVL tree

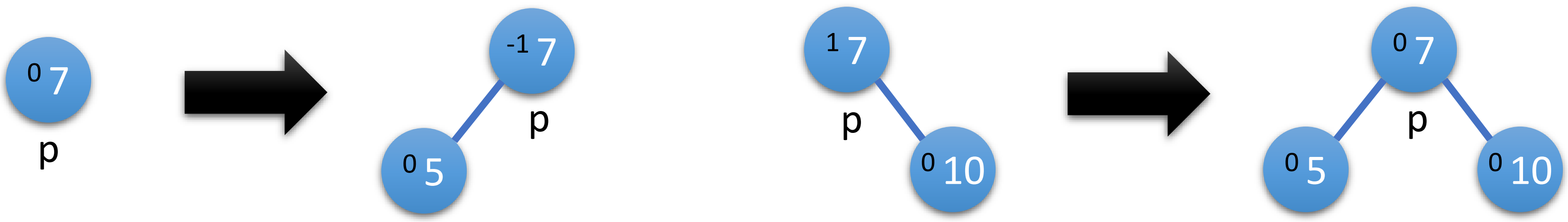
AVL Tree Example





AVL Tree Insertion

- Adding a new node cannot make an unbalanced parent
  - Assume the parent p's balance in b(p) in {-1, 0, 1} before insertion
  - The parent will have
    - b(p) = 0 if already has a child,
    - b(p) = 1 if a new left child, or b(p) = -1 if a new right child
  - Otherwise it would not be our parent or the parent would have been unbalanced already

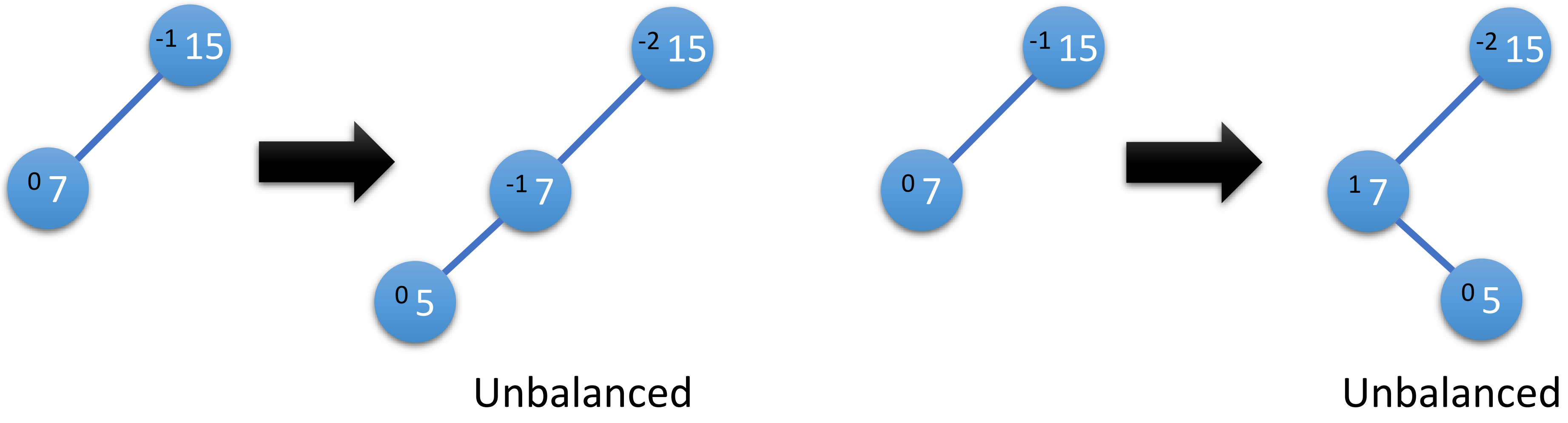


Still balanced

Still balanced

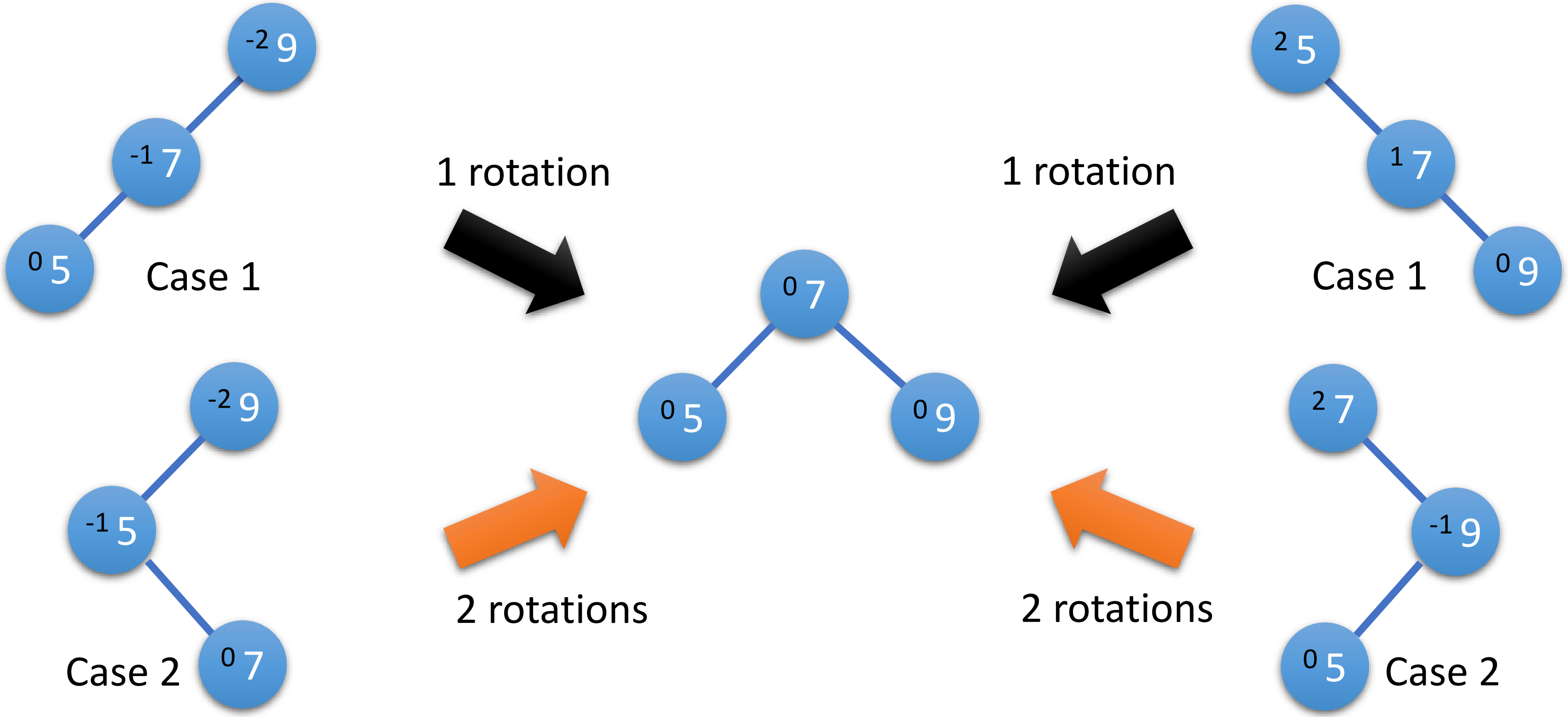
AVL Tree Insertion

- Even if the parent is not unbalanced, it is possible that the grandparent becomes unbalanced
- Hence, we need a way to re-balance the grandparent



AVL Tree Insertion

- Rotations are required to balance AVL tree
  - Case 1 (left-left or right-right): Requires 1 rotation
  - Case 2 (left-right or right-left): Requires 2 rotations (first convert to Case 1)



AVL Tree Insertion

- `Insert`[n]
  - If empty tree, then
    - Set n as root, b(n) = 0. Return

  - Else insert n (by walking the tree to a leaf p and inserting the new node as its child), set b(n) = 0, and look at its parent p
    - If b(p) was -1, then b(p) = 0. Return
    - If b(p) was +1, then b(p) = 0. Return
    - If b(p) was 0, then update b(p) and call `Insert-fix`[p, n]

AVL Tree Insertion

- Basic idea:
  - Work up ancestor chain updating balances of the ancestor chain or fix a node that is unbalanced

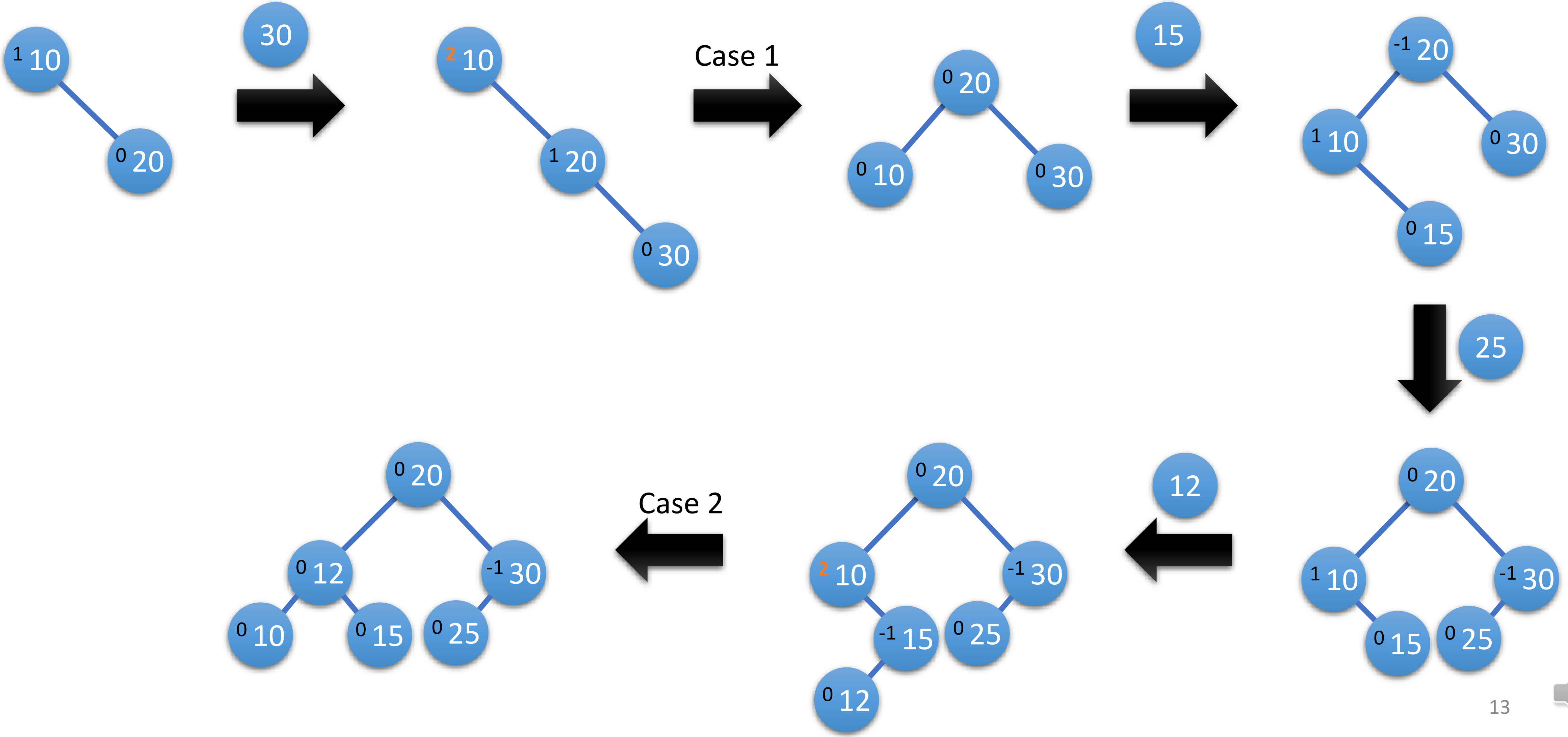
- `Insert-fix`[p, n]
  - Precondition: p and n are balanced: b(p), b(n) in {-1,0,1}
  - Postcondition: g, p, and n are balanced: b(g), b(p), b(n) in {-1,0,1}
    - If p is null or p.parent is null, return
    - Let g $\leftarrow$ p.parent

AVL Tree Insertion

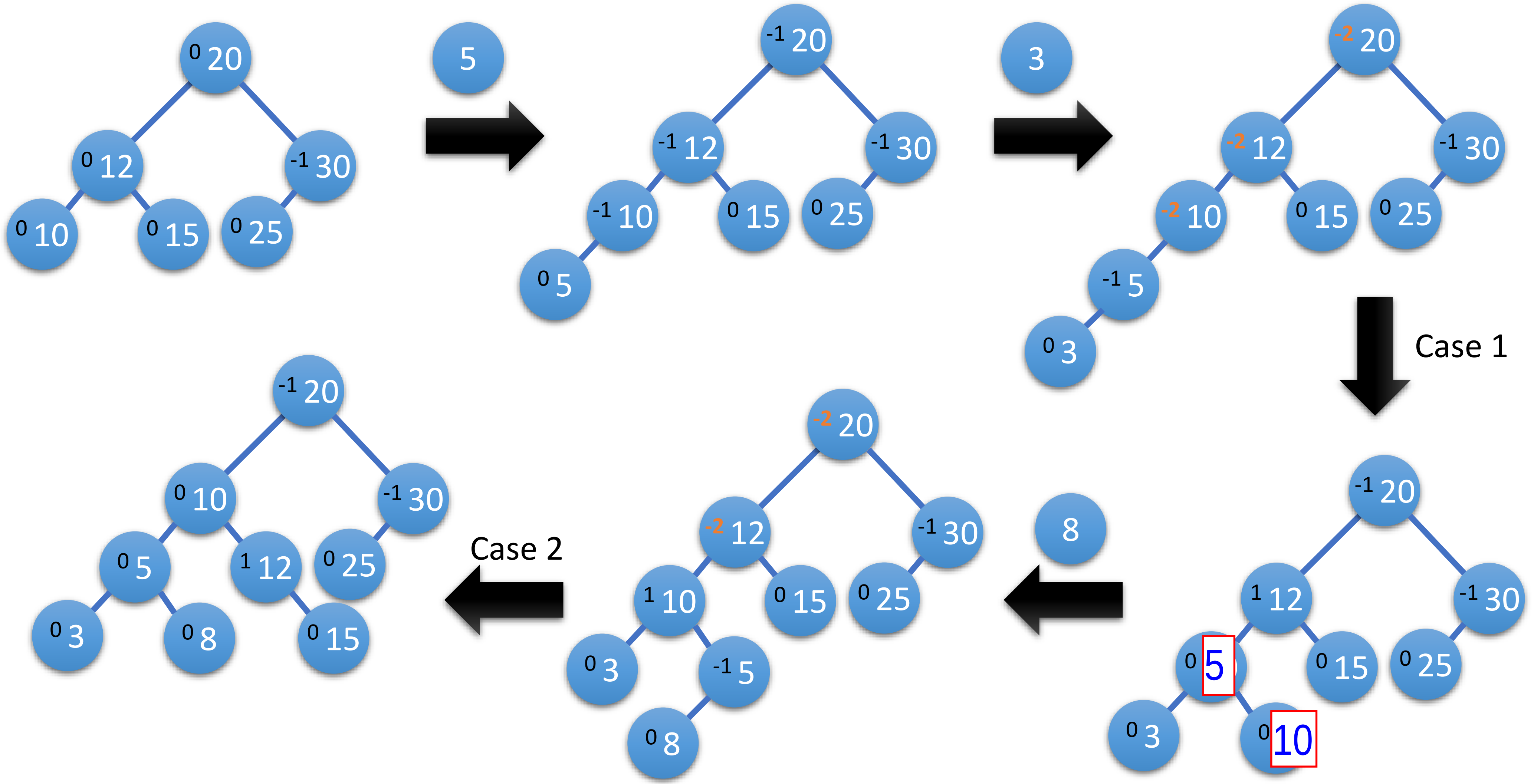
- `Insert-fix[p, n]`
  - Assume p is left child of g *// If p is right child, swap left/right, +/-*
  - Update b(g) *// Update g's balance to new accurate value for now*

  - Case a: if b(g) = 0, return
  - Case b: if b(g) = -1, `Insert-fix`[g, p] *// Fix recursively*
  - Case c: if b(g) = -2
    - If Case 1, then `rotateRight`(g);
    - If Case 2, then `rotateLeft`(p); `rotateRight`(g);

Example


1 10
0 20
30
2 10
1 20
0 30
Case 1
0 20
0 10
0 30
15
-1 20
1 10
0 30
0 15
25
0 20
1 10
-1 30
0 15
0 25
12
0 20
2 10
-1 30
-1 15
0 25
0 12
Case 2
0 20
0 12
-1 30
0 10
0 15
0 25

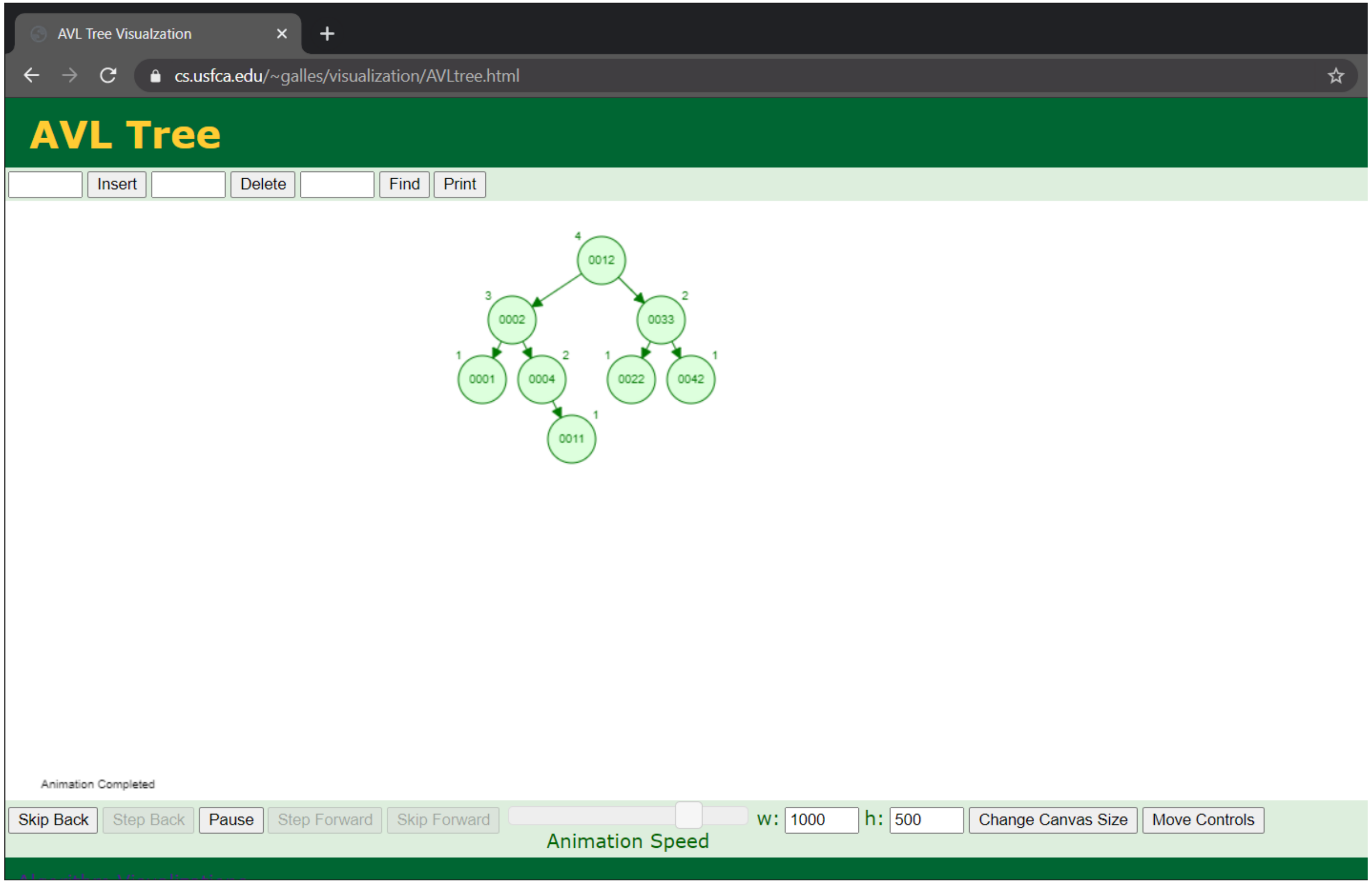
Example



AVL Tree Deletion

- Deletion operations may also require rebalancing via rotations
- The key idea is to update the balance of the nodes on the ancestor pathway
- If an ancestor gets out of balance then perform rotations to rebalance
  - Unlike insertion, rotations during deletion does not mean you are done, but need to continue recursively to the root
- More cases in AVL tree deletion

Demo

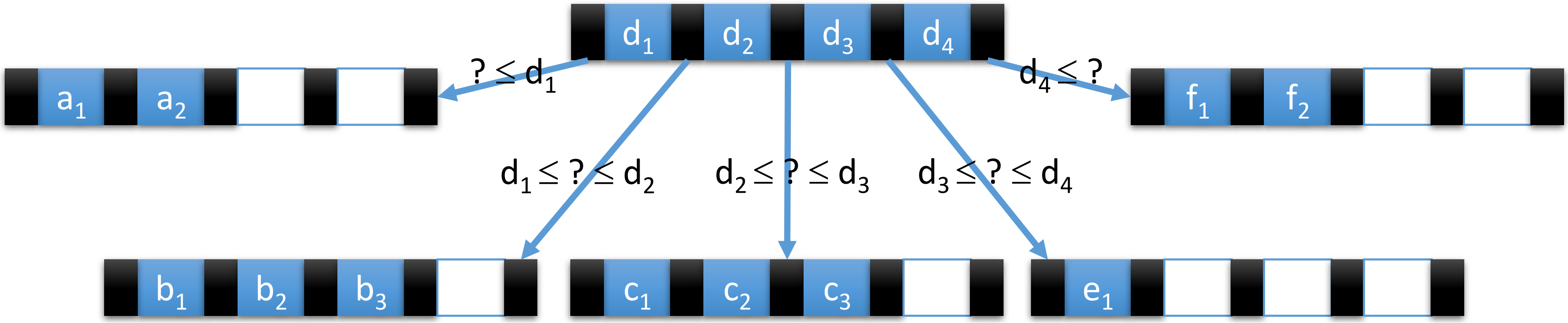


https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

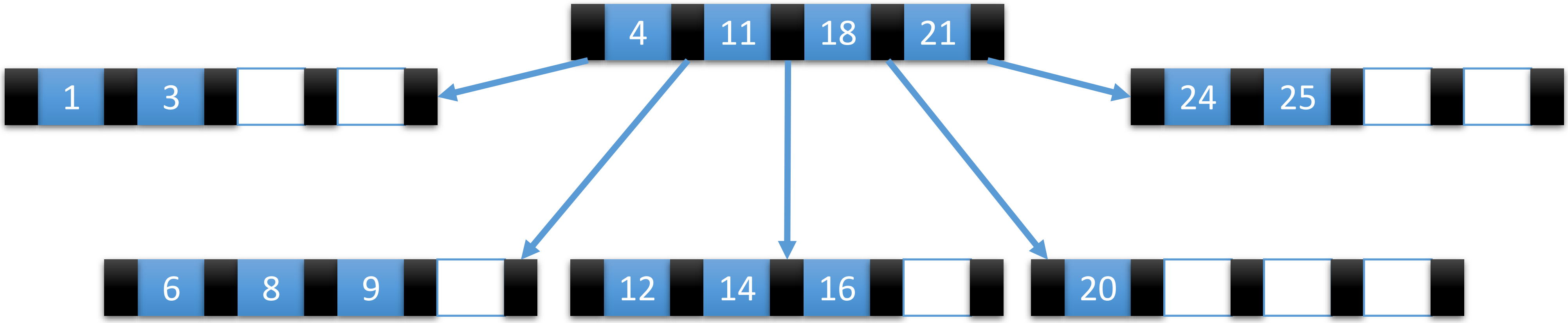
B-Tree

- Generalization of binary search trees
  - Not binary trees (i.e., more than two children per node)
  - The “B” stands for Bayer (the inventor)
- Designed for searching data stored on block-oriented devices
  - Map tree nodes into blocks
- Balanced tree
  - By keeping all leaves at the same level
- Each node can hold multiple items and have multiple children
- B-tree grows and shrinks from the root
  - Binary search tree only grows downward and also shrink from downward

B-Tree Structure



Example



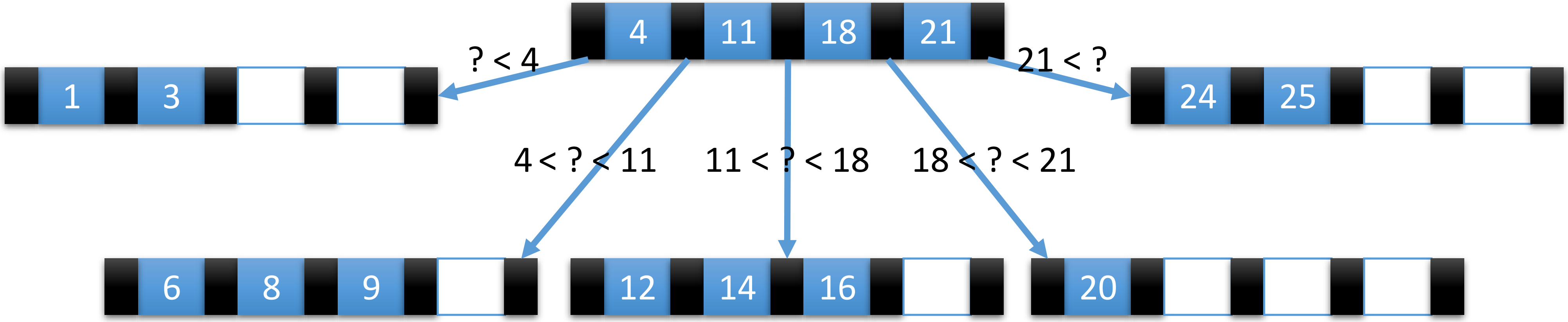
B-Tree Properties

- T is a B-tree of degree m (≥ 3) if it satisfies the following properties:
  1. All leaves of T appear on the same level of T
  2. Every node of T has at most m children
  3. Every node of T, except for the root and the leaves, has at least $m/2$ children
  4. The root of T is either a leaf or has at least two children
  5. An internal node with k children stores k-1 keys, and a leaf stores between $m/2$-1 and m-1 keys. The keys $v.\text{key}_i$, where $1 \leq i \leq k-1$, of a node v are maintained in sorted order, $v.\text{key}_1 \leq \ldots \leq v.\text{key}_{k-1}$
  6. If v is an internal node with k children, the k-1 keys of v separate the range of keys stored in the subtrees rooted at the children of v. If x_i is any key stored in the subtree rooted at the i-th child, the following holds:

$$x_1 \leq v.\text{key}_1 \leq x_2 \leq \ldots \leq v.\text{key}_{k-2} \leq x_{k-1} \leq v.\text{key}_{k-1} \leq x_k$$

Searching B-Tree

- Determine the range at each node
- Follow the pointer of the corresponding range as in binary search tree

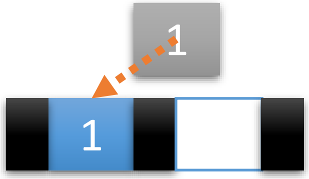


Organization of B-Tree

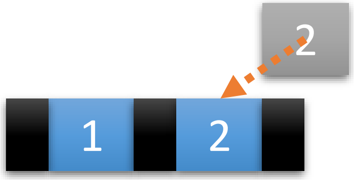
- Each non-leave node can have a variable number of children
- Must all be in a specific key range
- Number of child nodes typically vary between $m/2$ and m (with $m–1$ keys)
    - Split nodes that would otherwise have contained $m + 1$ children (after insertion)
    - Merge nodes that contain less than $m/2$ children (after deletion)

B-Tree Insertion

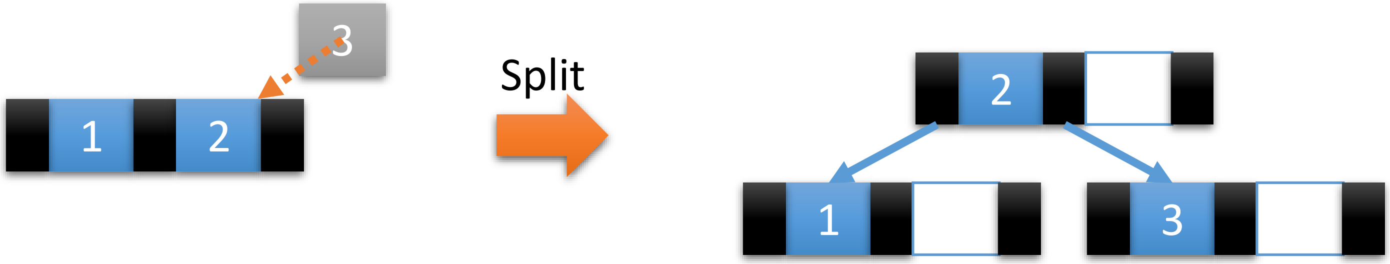
- Consider B-tree where each node can have up to 2 keys (3 children)
- Add 1:



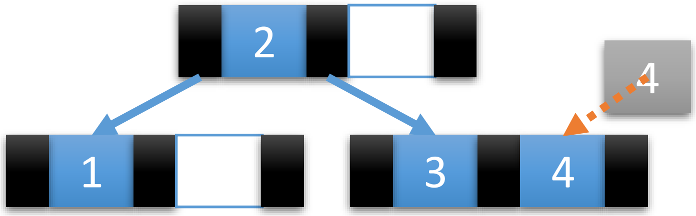
- Add 2:



- Add 3:

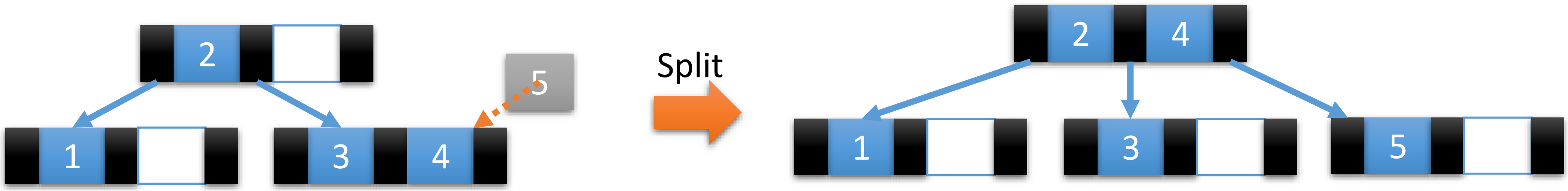


- Add 4:

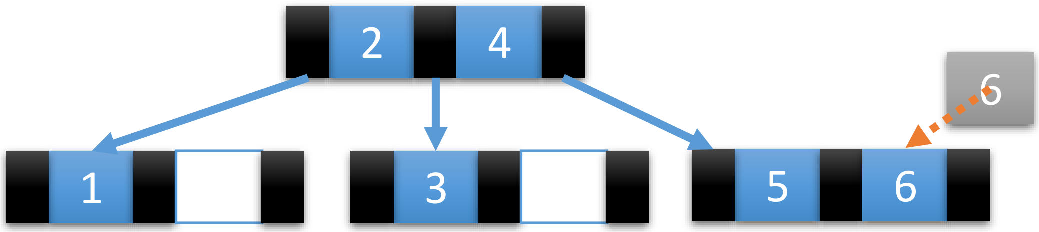


B-Tree Insertion

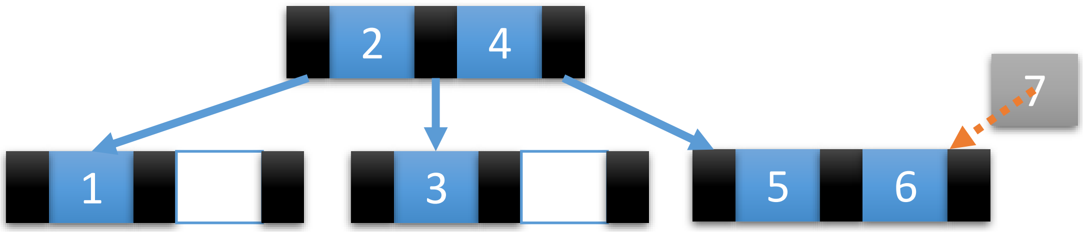
- Add 5:



- Add 6:

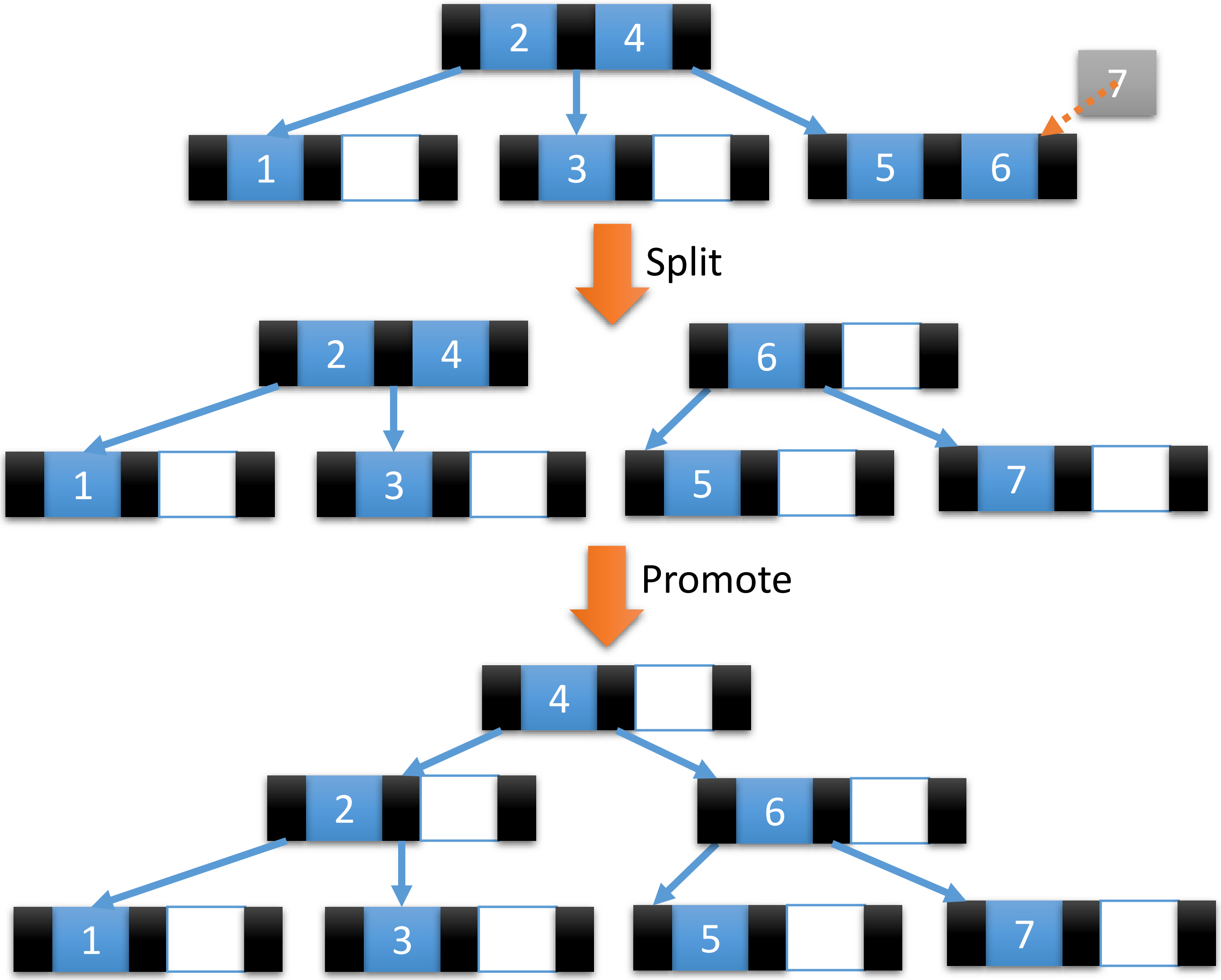


- Add 7:



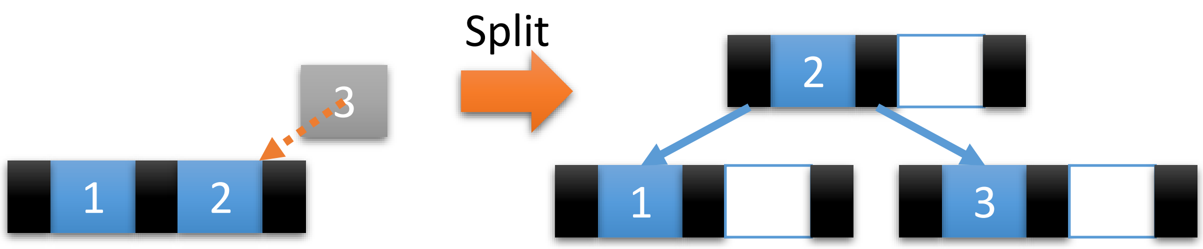
B-Tree Insertion

- Add 7:

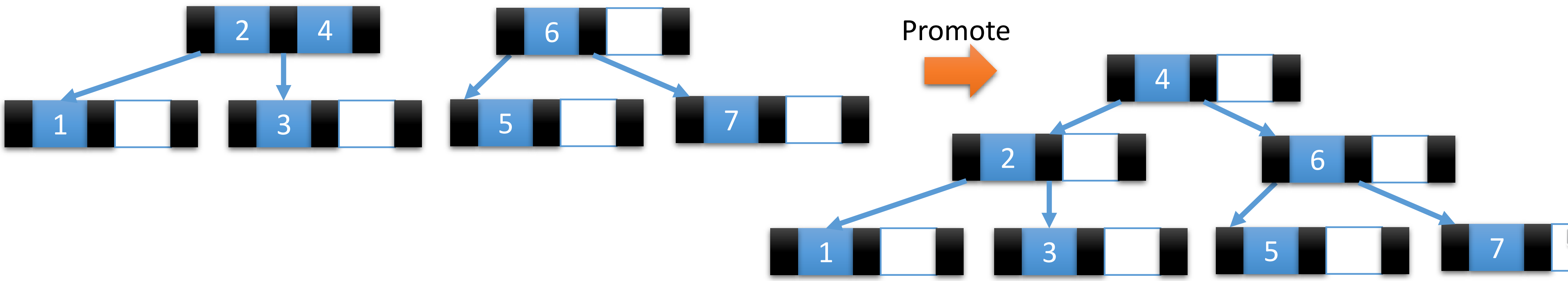


B-Tree Insertion Basic Operations

- Split:
  - When trying to add to a full node
  - Split node at the central value



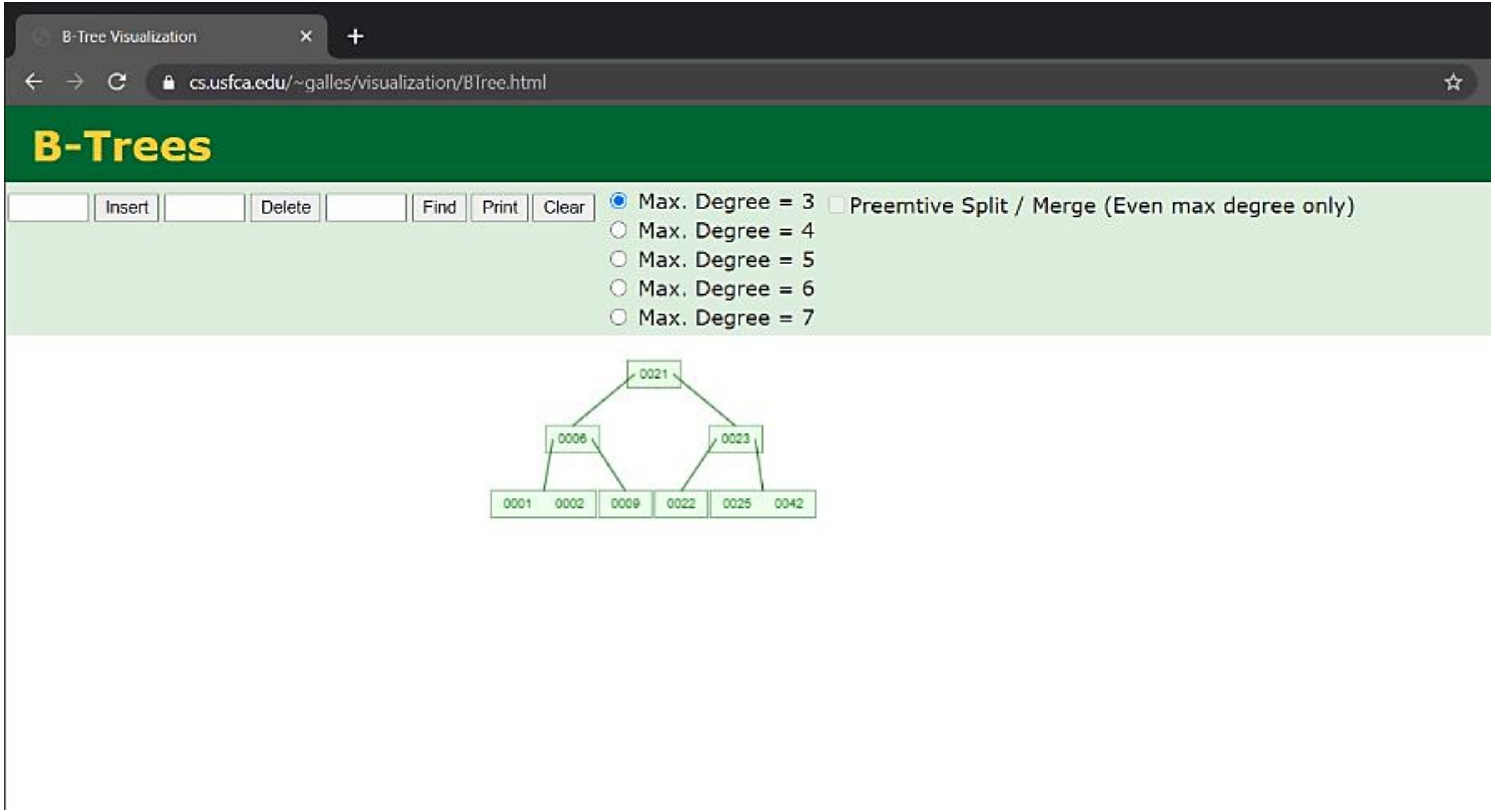
- Promote:
  - Must insert root of split node higher up
  - May require a new split



B-Tree Insertion (Basic Idea)

- Find target leaf L
- If L has less than m - 2 entries :
  - Add the entry
- Else
  - Repeat
    - Allocate new leaf L'
    - Pick the $m/2$ highest keys of L and move them to L'
    - Insert highest key of L and corresponding address leaf into the parent
    - If the parent is full
      - Split it and add the middle key to its parent node
  - Until a parent is found that is not full

Demo



https://www.cs.usfca.edu/~galles/visualization/BTree.html

Summary

- AVL tree
  - Keeping every node's balance in {-1, 0, 1}
  - Rotations are required during insertion and deletion
  - AVL tree is almost balanced tree

- B-tree
  - B-tree properties
  - Insertion

Reference

- Visualizations
  - https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
  - https://www.cs.usfca.edu/~galles/visualization/BTree.html