Week 12: Height Balancing Trees

CS-250 Data Structure and Algorithms

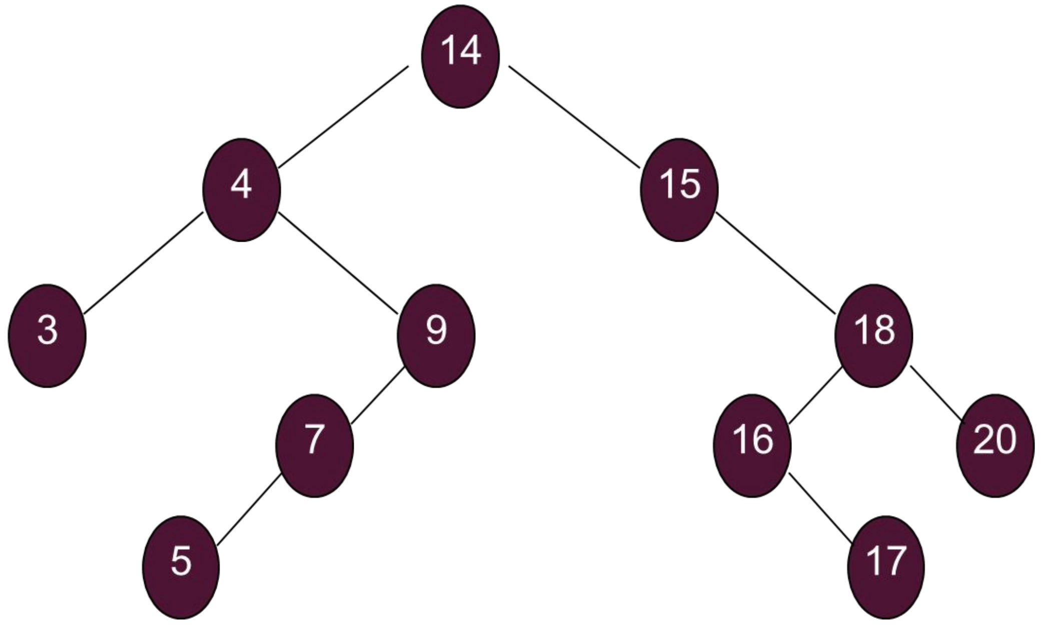
DR. Mehwish Fatima | Assist. Professor
Department of AI & DS | SEECS, NUST

OUTLINE

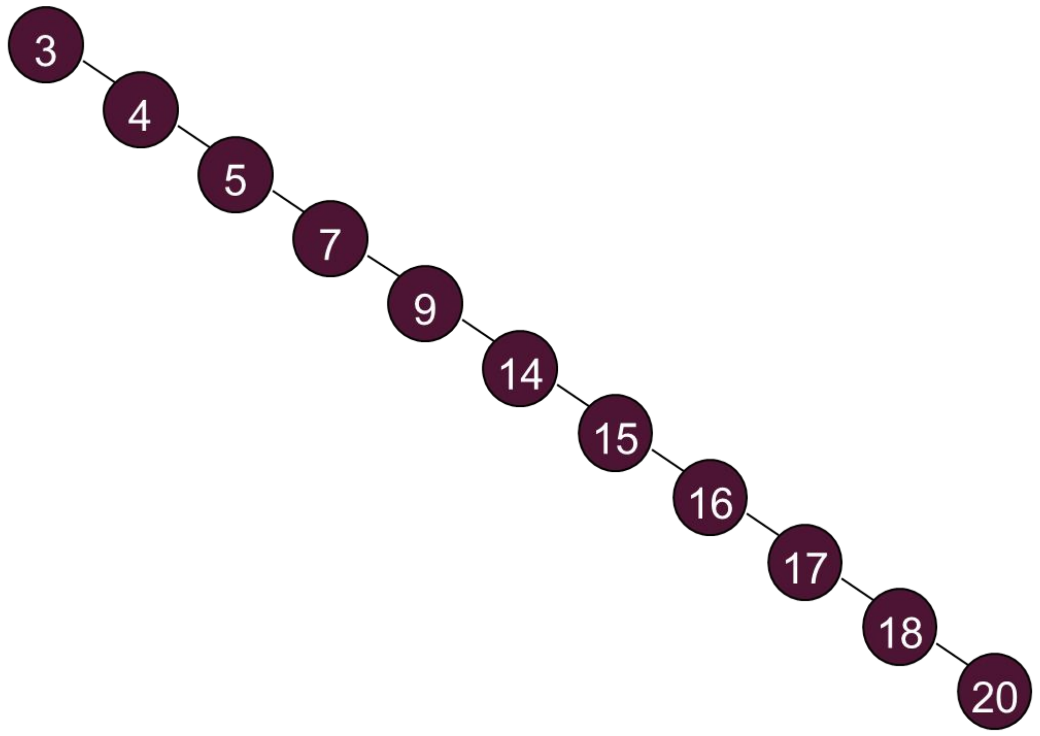
- Search Trees
  - Self Balancing Binary Search Trees
    - AVL
      - Insertion
      - Deletion

DEGENERATE BINARY SEARCH TREE

- BST for 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17

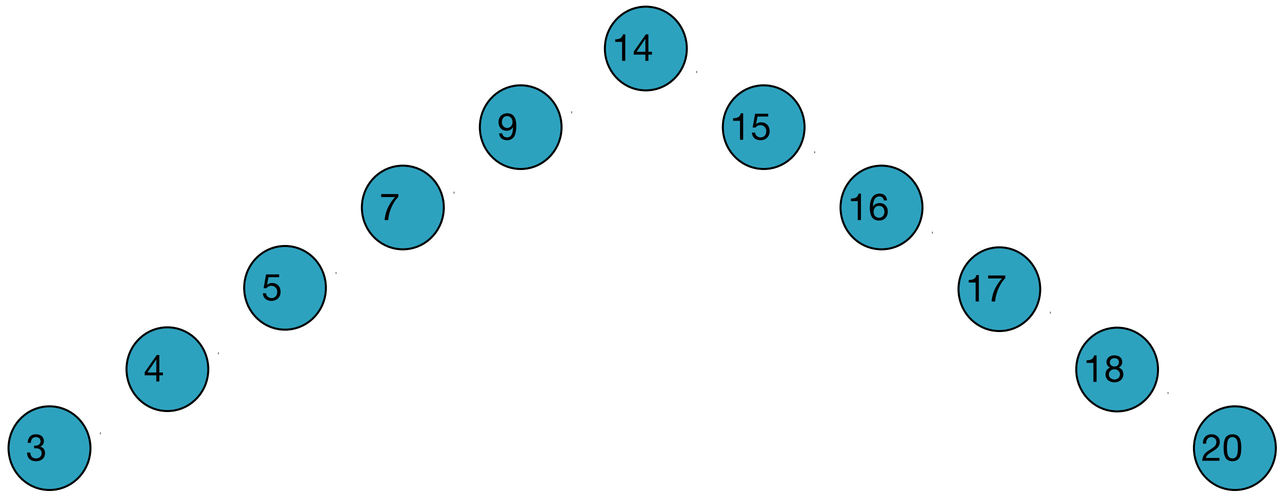


- BST for 3 4 5 7 9 14 15 16 17 18 20



BALANCED BST

- We should keep the tree ***balanced***.
- One idea would be to have the left and right sub-trees have the same height



Does not force the tree to be ***shallow***.

BALANCED BST

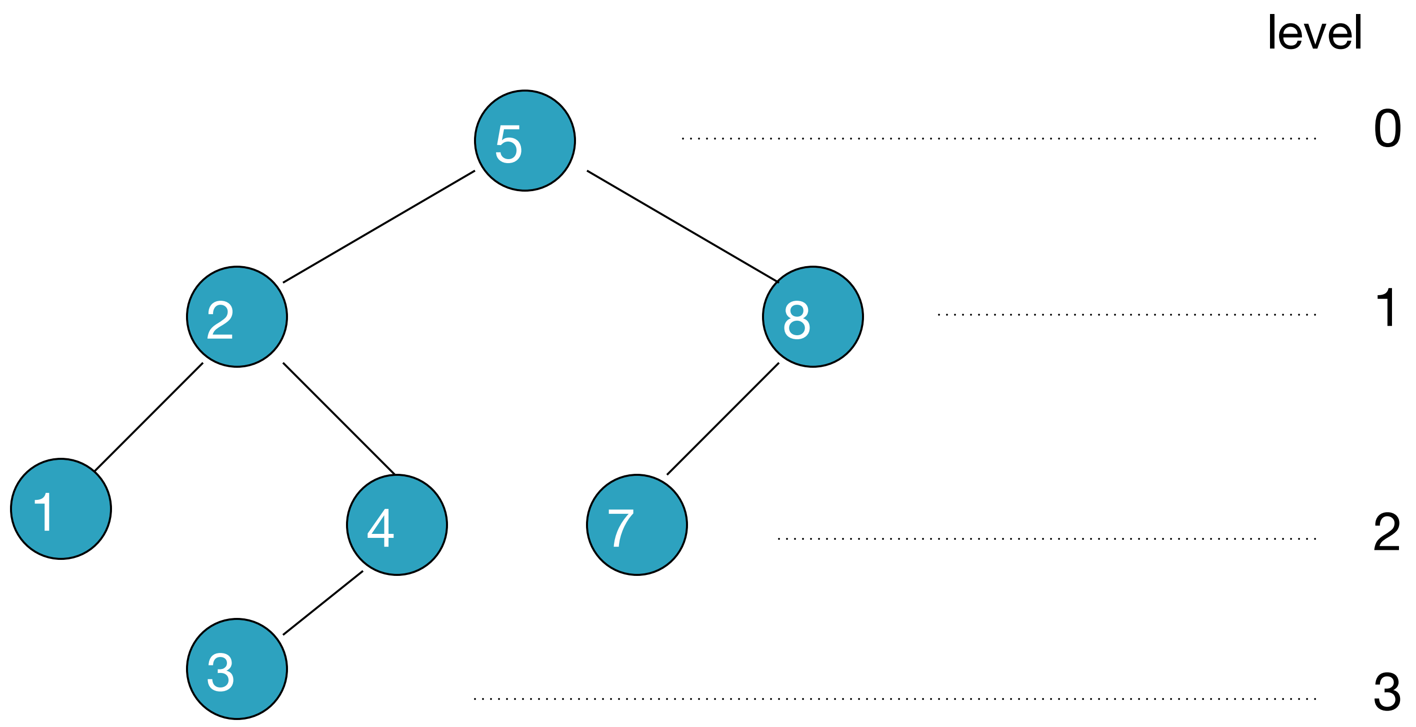
- We could insist that every node must have left and right subtrees of same height.
- But this requires that the tree be a complete binary tree.
- To do this, there must have (2^{d+1} – 1) data items, where d is the depth of the tree.
- This is too rigid a condition.

AVL TREE

- AVL (Adelson-Velskii and Landis) tree.
- An AVL tree is identical to a BST except
  - height of the left and right subtrees can differ by at most 1.
  - height of an empty tree is defined to be (–1).

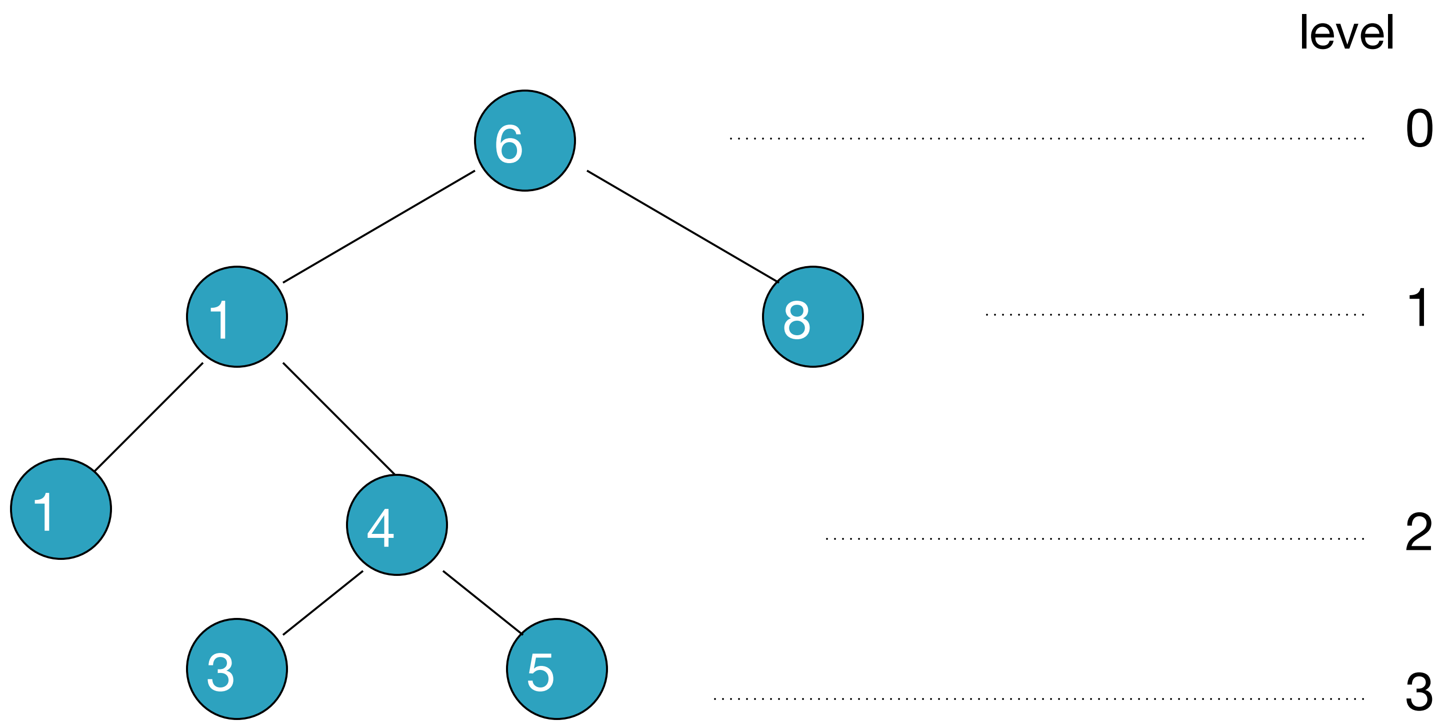
AVL TREE

- An AVL Tree



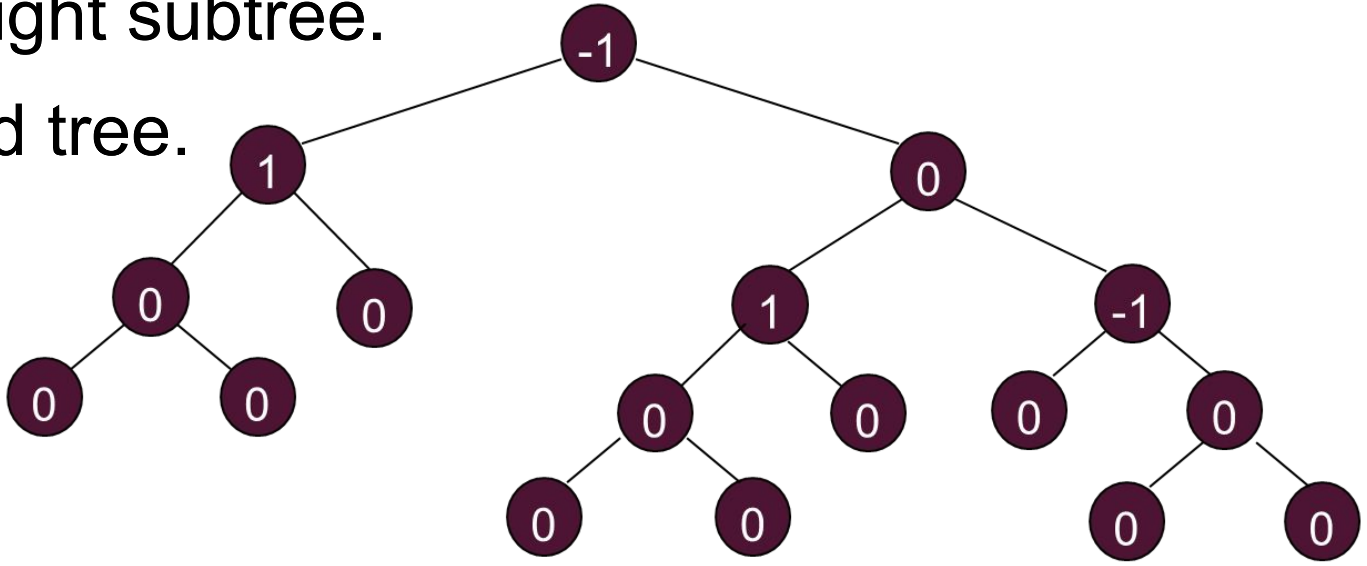
AVL TREE

- Not an AVL tree



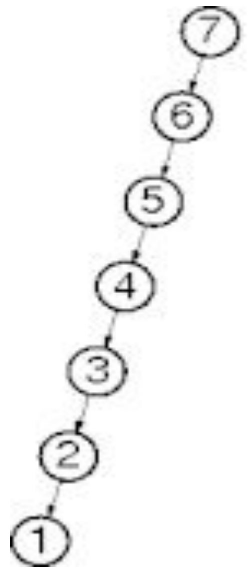
BALANCED BINARY TREE

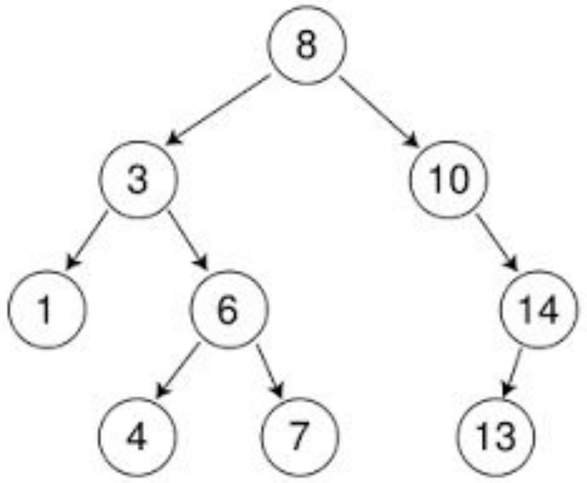
- The height of a binary tree is the maximum level of its leaves (also called the depth).
- The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.
- Here, for example, is a balanced tree.
- Each node has an indicated balance of 1, 0, or –1.

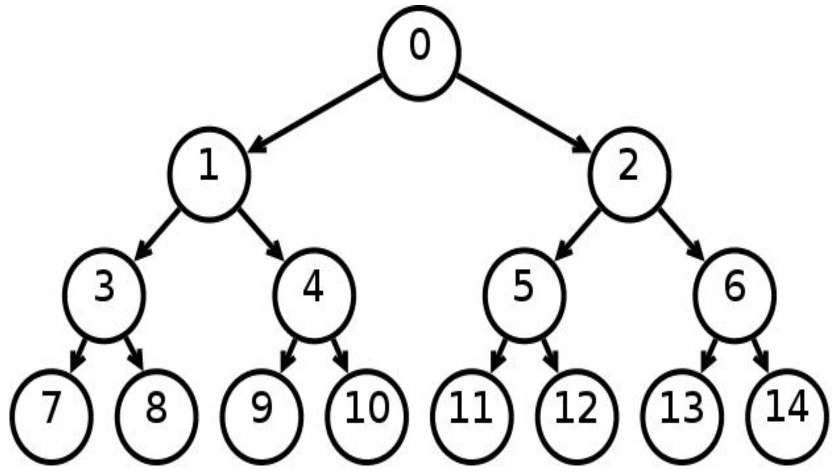


TREE BALANCING

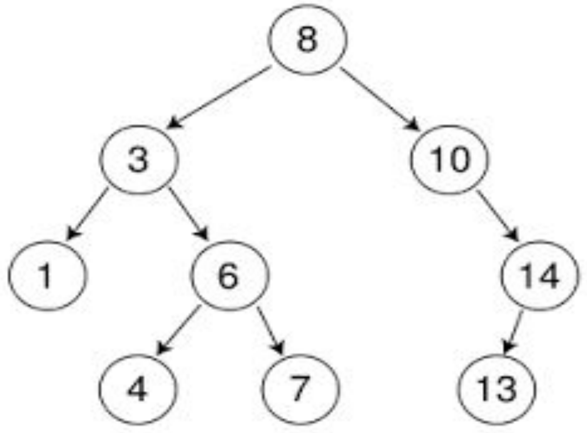
- Time complexity of tree algorithms depends upon their height.
  - Where height can vary from O(log n) to O(N)
    - Total nodes $N=2^{H+1}-1$
  - What is the average case height of a binary tree?
  - What is the worst case height of a binary tree?







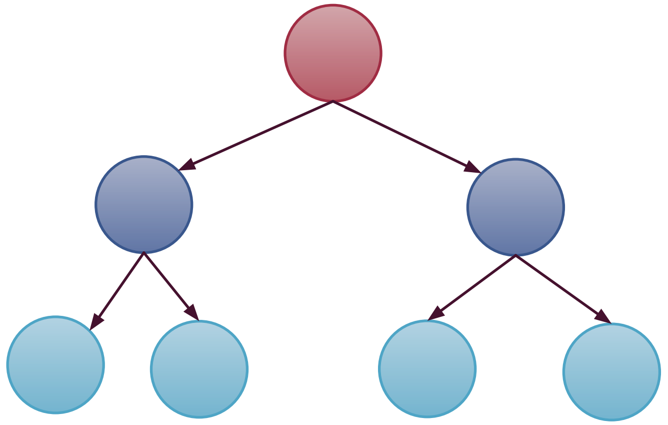
TREE BALANCING

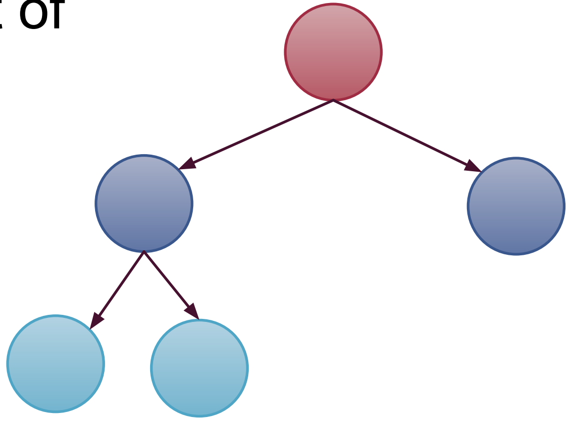


- If a tree is not well balanced and is skewed, it can lead to a worst case time complexity of O(N) because height H is equal to N-1.
  - In a BST, when we delete a node and bring a node only from one side, let say from left sub-tree (predecessor) or right subtree (Successor) then tree will become skewed ultimately.
    - If we delete root 3-4 times and always bring next node from right
    - What will be final tree?
  - If we are given an almost sorted list of numbers then resultant tree will also be a skewed tree again.
    - What will be BST of 1 2 3 4 5 6 7?

BALANCED TREE

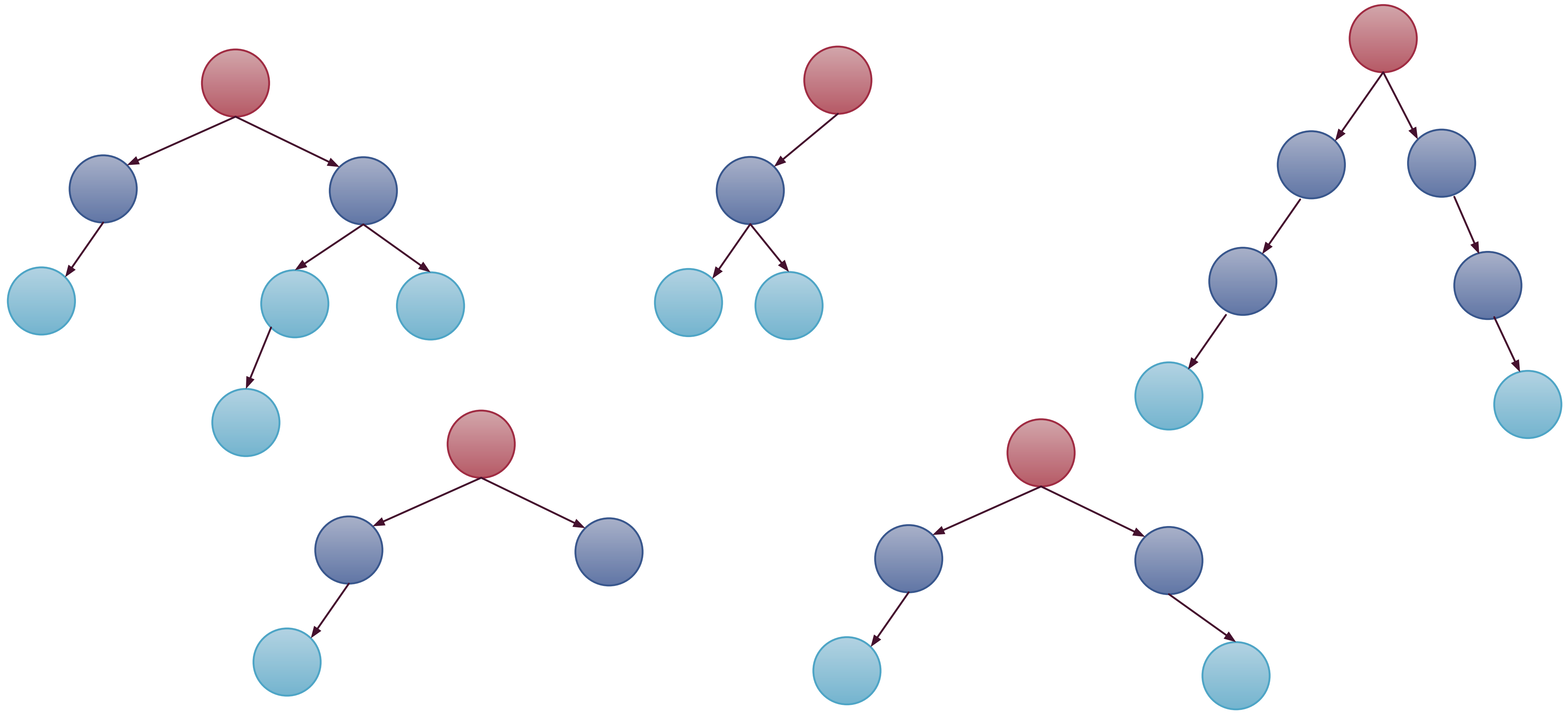
- To handle such situation, we need to ensure the tree to be height balanced.
  - A perfectly height balanced tree is where height of left subtree is equal to height of right subtree
    - But this case is not very common.
  - But, we can achieve an almost perfectly balanced tree
    - We will define a height balanced tree as: a tree in which height of left subtree and right subtree differs by almost 1
    - **Note**: This lecture assume Height= max Depth/level +1
    - It will make more sense here
    - So, single node Height=1, empty tree Height =0





BALANCED TREE

- Which trees not balanced?

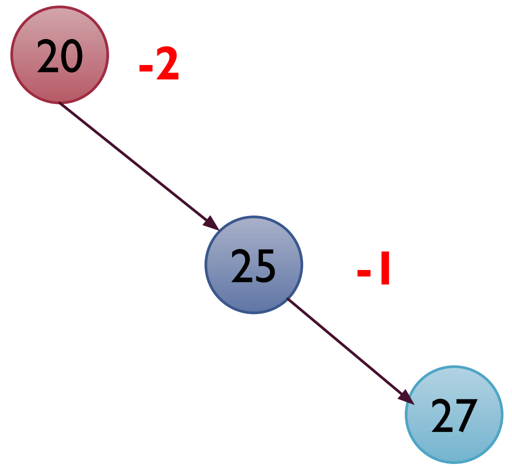


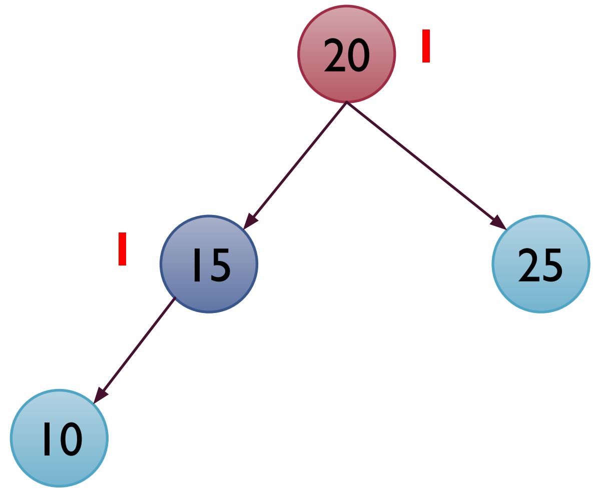
SELF BALANCING OR HEIGHT BALANCED BSTS

- Binary search trees that keep their height as small as possible.
  - This is achieved by doing rotations at time of insertion and deletion
    - AVL Trees
    - Red Black Trees
    - Splay Trees
    - Multiway search Trees like B- Trees, 2-3 Trees
    - And many others

AVL(ADELSON-VELSKY AND LANDIS) TREE

- A self balancing binary search tree in which height of left subtree and right subtree of each node differs by at most 1. So, balance factor of any node can be -1, 0 or 1
  - Balance Factor of node:
    - **height(left subtree of node) – height(right subtree of node**

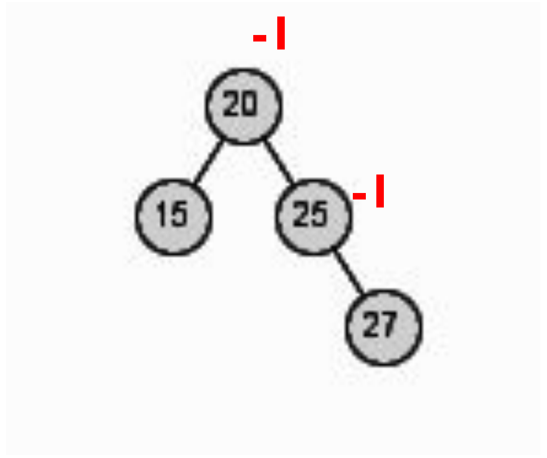


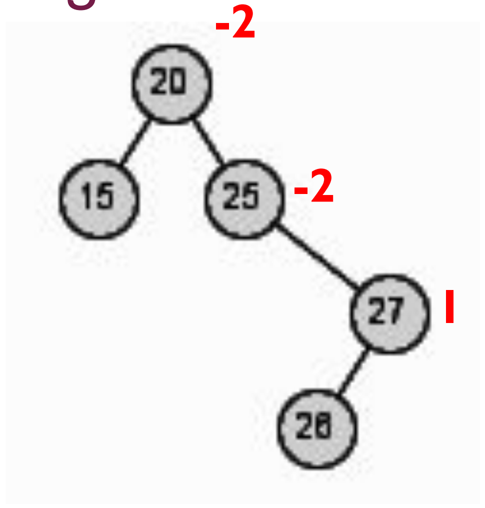


  - After insertion or deletion, if balance factor becomes more than 1 or less than -1, then a rotation is performed to get desired balance
  - So either we are inserting or deleting in a BST, we need to maintain height property to make it an AVL tree

AVL TREE

- Insertion
  - When we insert a node into a tree, height of few nodes can be affected.





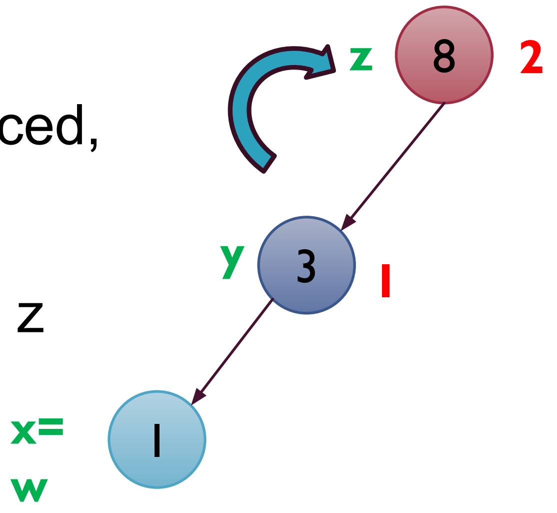
  - What is the relationship between the imbalanced node and newly inserted node?
    - If we are inserting a node in left side of tree then is it possible that height of nodes in right subtree can be affected? → NO
      - Means a newly inserted node can only affects its ancestor nodes at max

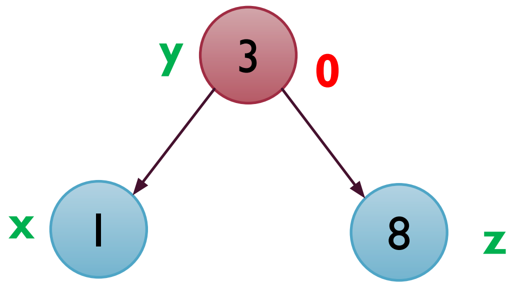
AVL-INSERTION

- Insertion
  1. Insert the node according to BST rule, let’s call it **w**
  2. Check balance factor of each ancestor of this node until an imbalanced ancestor is found
     - Let’s call it **z**
     - Let **y** be the root of taller sub-tree of z (**y** must be an ancestor of **w**)
     - Let **x** be the root of taller sub-tree of y (**x** must be ancestor of w or it can be **w** itself)
  3. Now perform rotation on this node combination.
     - Rotation will rearrange these nodes in a way, that both properties of AVL will be retained
       - Re-arranged tree will be a height balanced tree
       - It will be a BST
  - There are four rotation scenarios depending upon position of **y** and **x**
    - All four cases requires Single or Double rotations

INSERTION

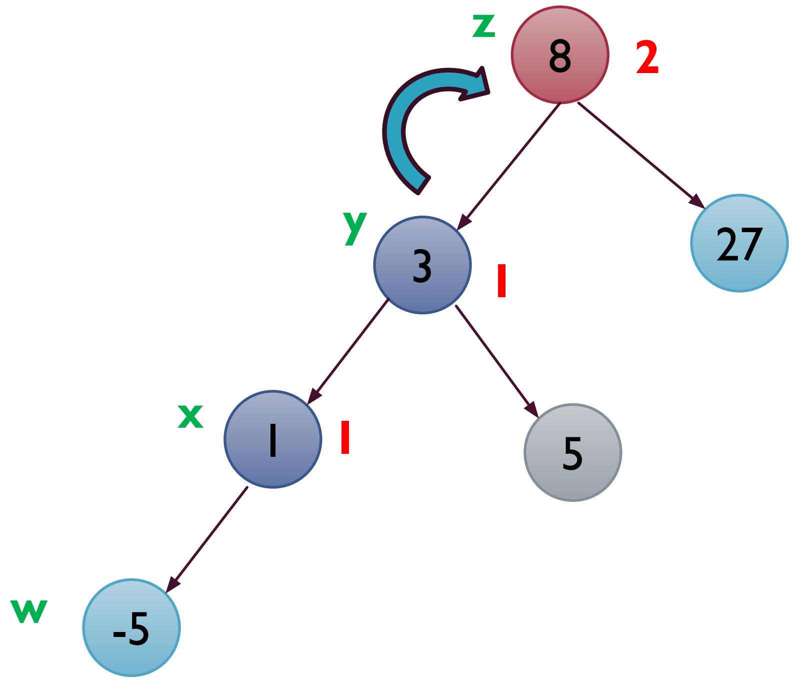
- Left-Left case
  - After inserting 1
    - We will go up to see if some ancestor has become imbalanced,
    - 3 is balanced, so go 1 step above to 8
    - Tree has become imbalanced at 8 that would be labeled as z
    - 3 would be y and 1(w) would be x
  - Rotation
    - Make z right child of y

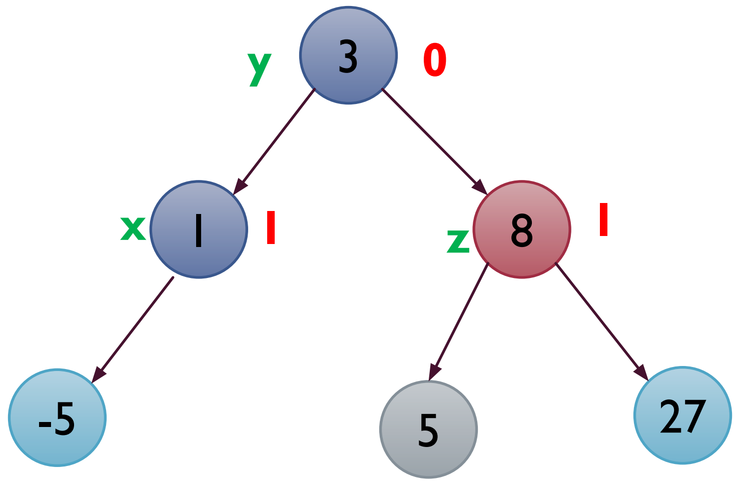




INSERTION

- Left-Left case
  - After inserting -5
  - Rotation?
    - Make z right child y
    - If y has already a right child,
      - Make this right child the left child of z

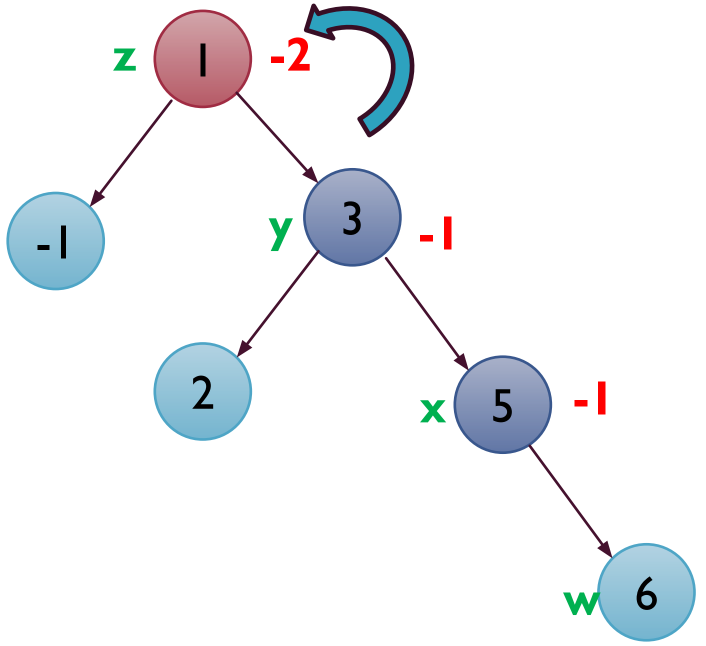


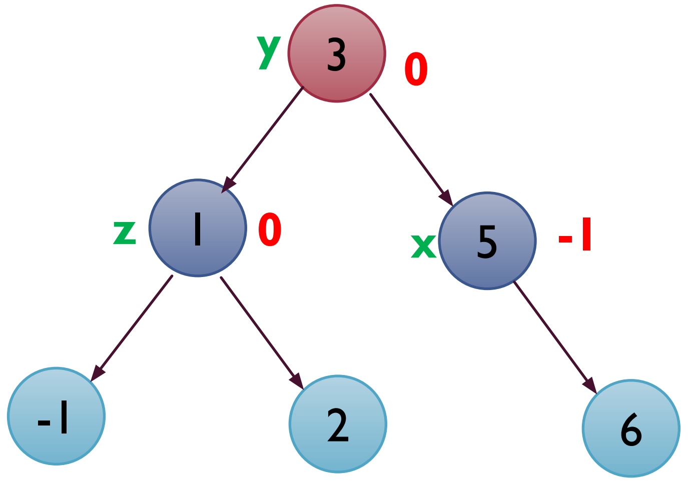


- Point to Remember: after rotation, height of tree will be same as it was before getting imbalanced
- Z(imbalanced parent) would always be a grandparent, it can never be an immediate parent

INSERTION

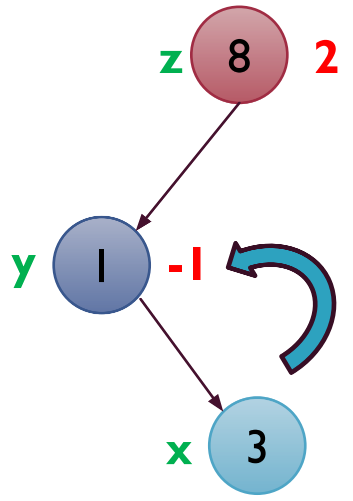
- Right-Right case (Mirror of Left-Left case)
  - After inserting 6
  - Rotation?
    - Make z left child of y
    - If y has already a left child,
      - Make it right child of z



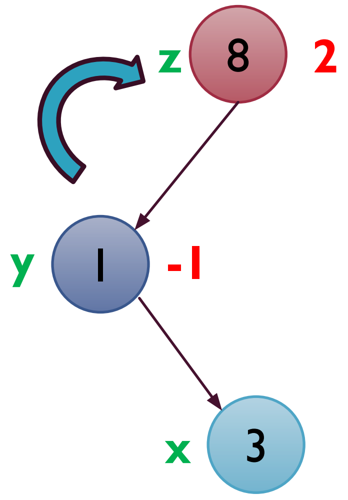


INSERTION

- Left-Right case
  - After inserting 3

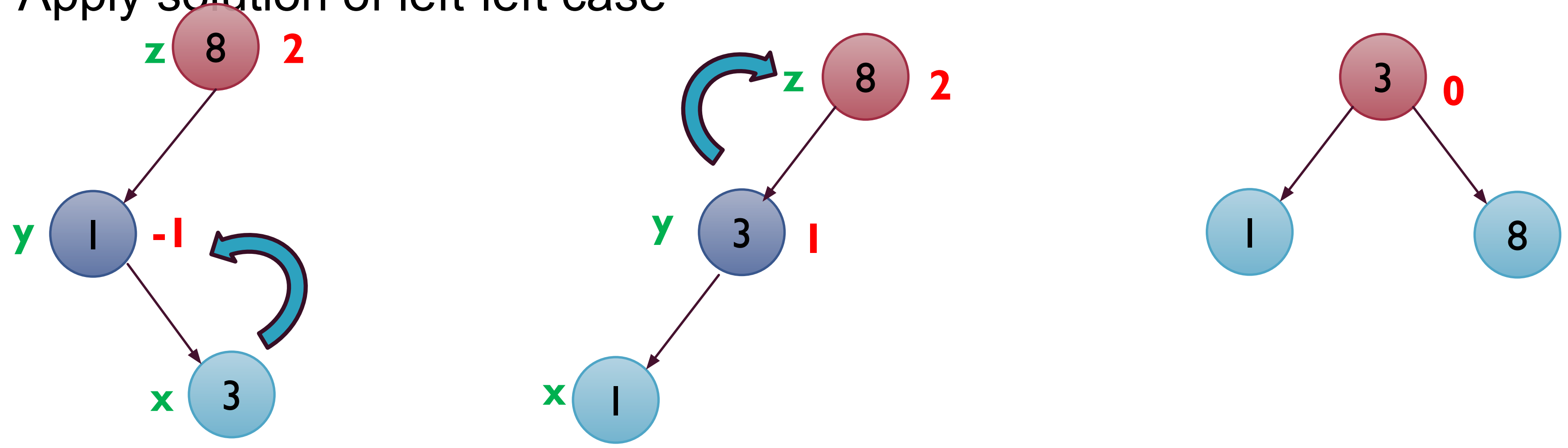


- A single rotation is not enough
- If we simply do rotation on z
- Tree will not be balanced



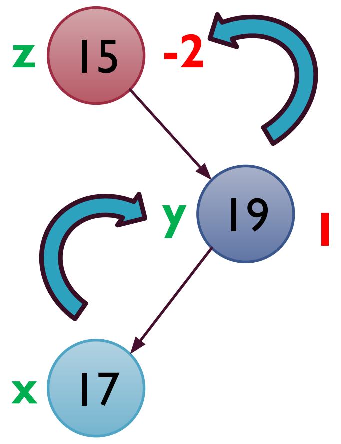
INSERTION

- Left-Right case
  - Rotations
    - Make it a left-left case by swapping values of y and x, and then x be the left child of y
    - Apply solution of left-left case

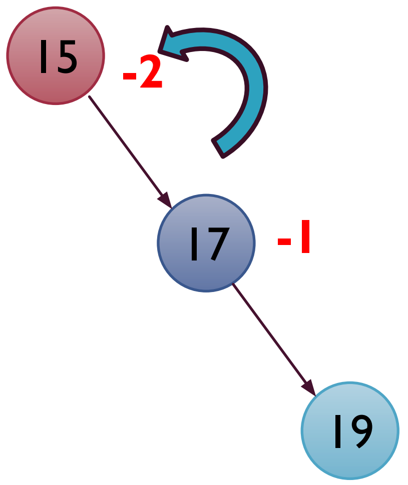


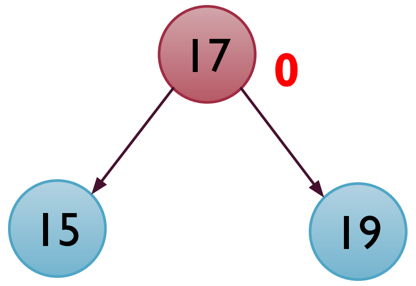
INSERTION

- Right-Left Case



- Make it right-right case
- Then apply solution of right-right case





INSERTION

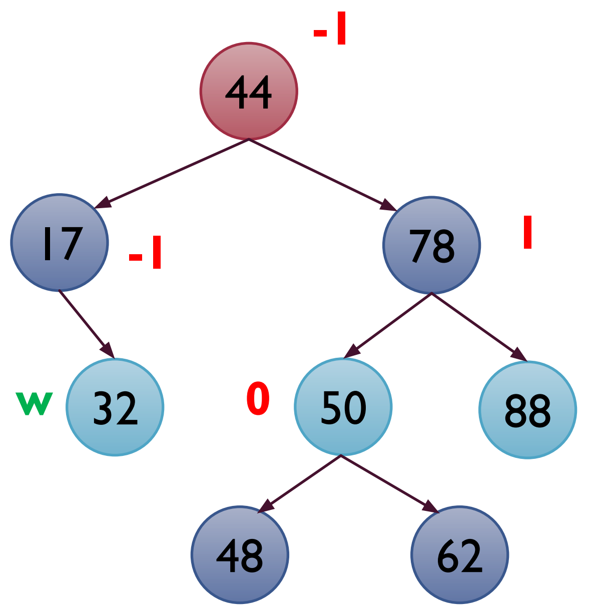
- So what is the time complexity of AVL Insertion?
- It performs two functions actually
  - BST Insertion is →O(logn)
  - Rebalancing
    - It involves two steps
      - Finding imbalanced ancestor ⟶ may take O(logn)
      - Rotation of nodes ⟶ constant time since maximum two rotations are required

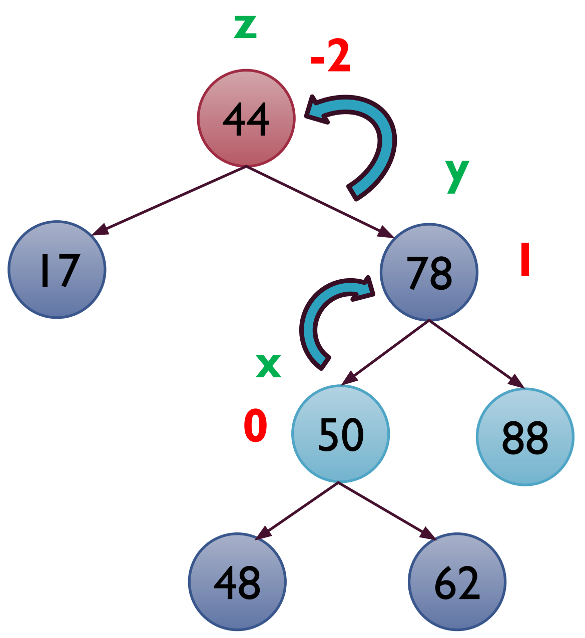
DELETION

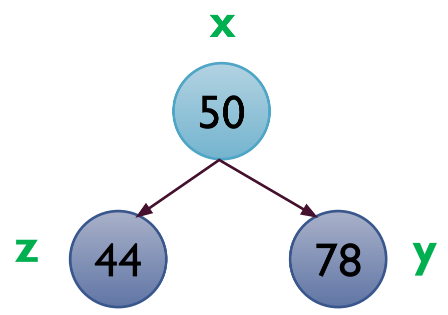
- Deletion is more complex than insertion, as it may require more than 2 rotations.
- In worst case it may lead to rotations at each level which would be equivalent to O(logn).
- Find the x, y and z nodes
  - Start from parent of deleted node w, go up in the path to root un-till the first node is found that is imbalanced, let's call it z.
    - Let y be the root of taller sub-tree of z (y must **not** be an ancestor of w)
    - And x be the root of taller sub-tree of y, if both sub-trees are of same height, then x will be root of sub-tree that is on same side as y (if y is left, x would also be left and vice versa).
  - Now perform rotation according to possible 4 cases.
  - After rebalancing z, move upward repeating same process until root.

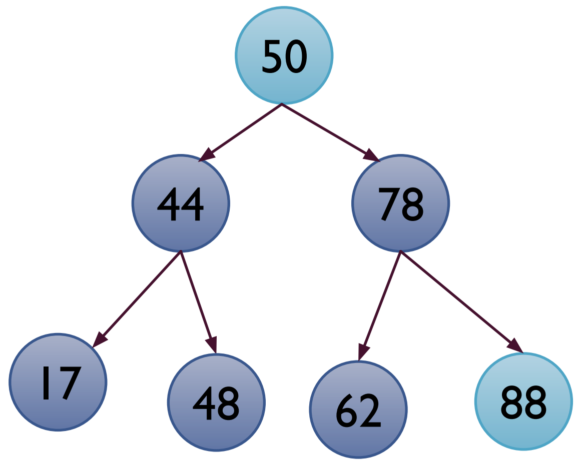
DELETION

- Deletion 32







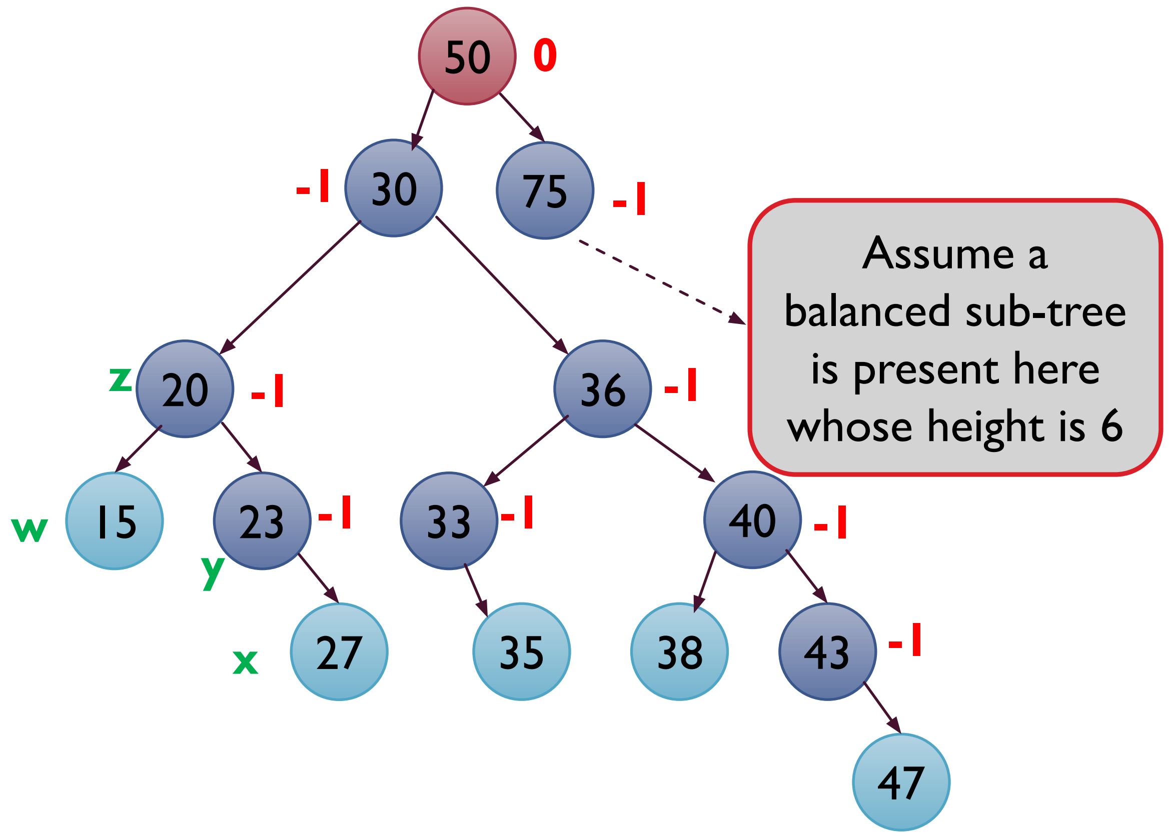


EXAMPLE

- If we delete 15
- Balance factor of 20 will be affected
- After rebalancing 20
  - Balance of 30 will be affected

  - And may be after rebalancing of 30
  - 50 can also become imbalanced

  - In worst case, we may need to balance all ancestors of deleted node
    - It will lead to O(logn) rotations



50
0
30
-1
75
-1
Assume a balanced sub-tree is present here whose height is 6
z
20
-1
36
-1
w
15
23
-1
y
33
-1
40
-1
x
27
35
38
43
-1
47

- http://en.wikipedia.org/wiki/AVL_tree
- http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-insert.html
- http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-delete.html
- http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-height.html

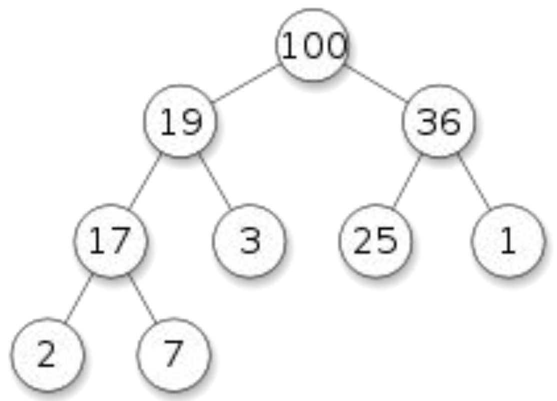
Binary Heap

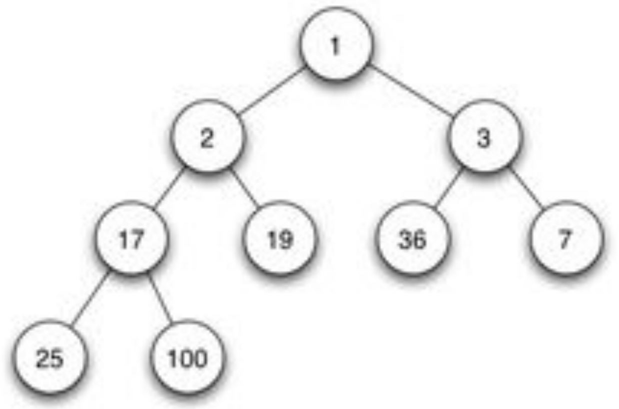
BINARY HEAP

- A binary heap is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:
  - Shape property
    - A binary heap is a complete binary tree; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
  - Heap property
    - All nodes are either greater than or equal to or less than or equal to each of its children, according to a comparison predicate defined for the heap.

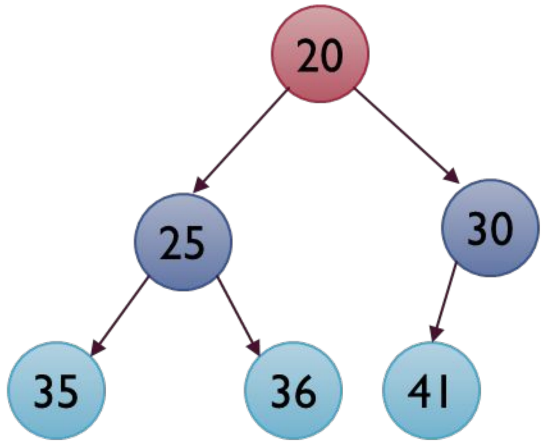
PRIORITY QUEUE

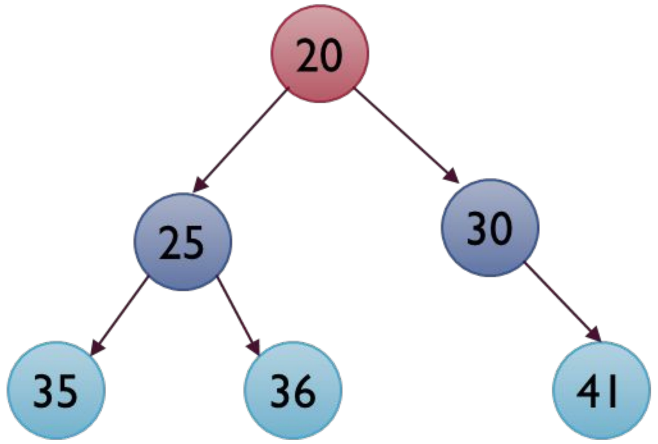
- Heaps with a mathematical "greater than or equal to" (≥) comparison predicate are called max-heaps.
- Heaps with a mathematical "less than or equal to" (≤) comparison predicate are called min-heaps. Min-heaps are often used to implement priority queues.

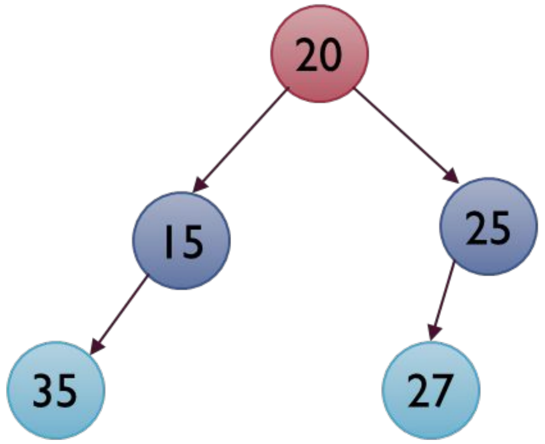




BINARY HEAP







Which tree is binary heap?

BINARY HEAP OPERATIONS

- Both the insert and remove operations modify the heap to adapt to the shape property first, by adding or removing from the end of the heap.
- Then the heap property is restored by traversing up or down the heap.
- Both operations take O(log *n*) time.

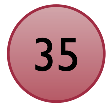
INSERTION

- To add an element to a heap we must perform an up-heap operation (also known as bubble-up, heapify-up, or cascade-up), by following this algorithm:

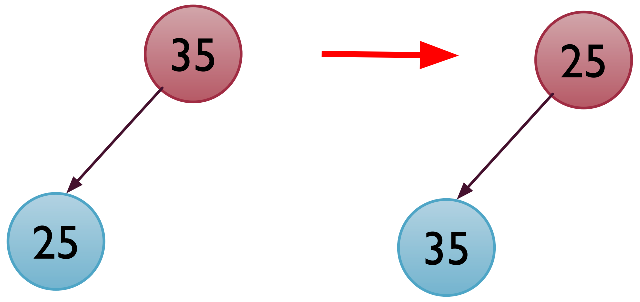
1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

HEAPIFY-UP

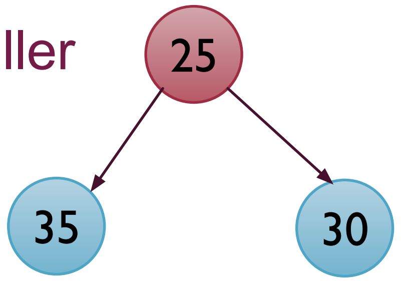
- Insert 35



- Insert 25
  - Swap with parent if parent is larger

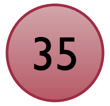


- Insert 30
  - Parent is already smaller

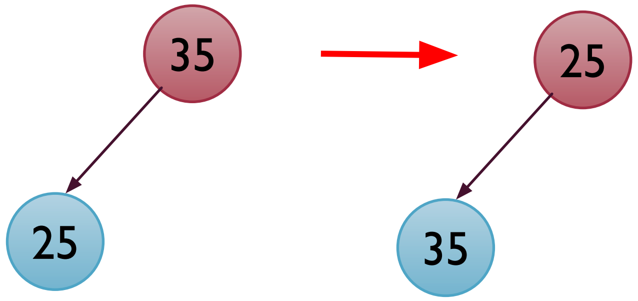


HEAPIFY-UP

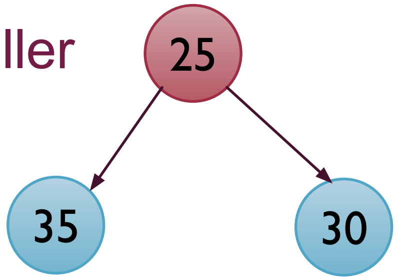
- Insert 35



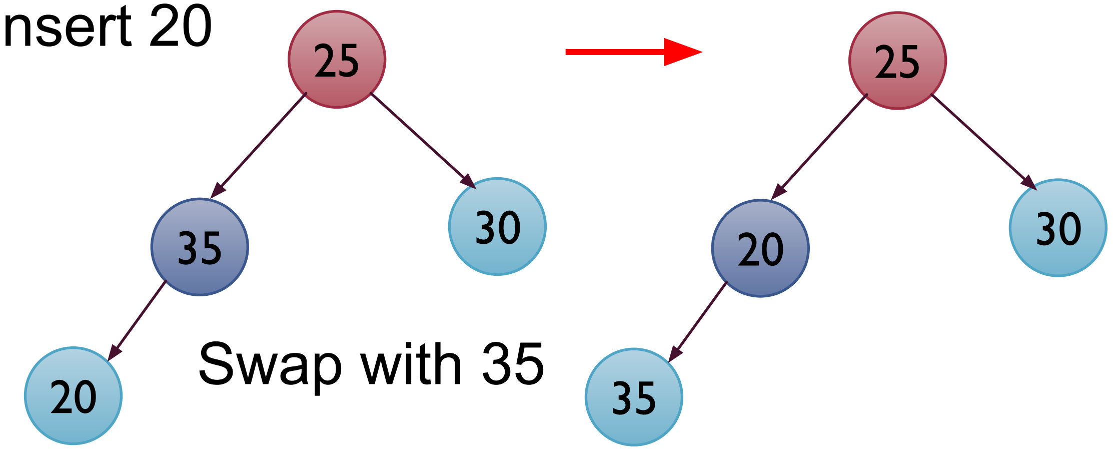
- Insert 25
  - Swap with parent if parent is larger



- Insert 30
  - Parent is already smaller

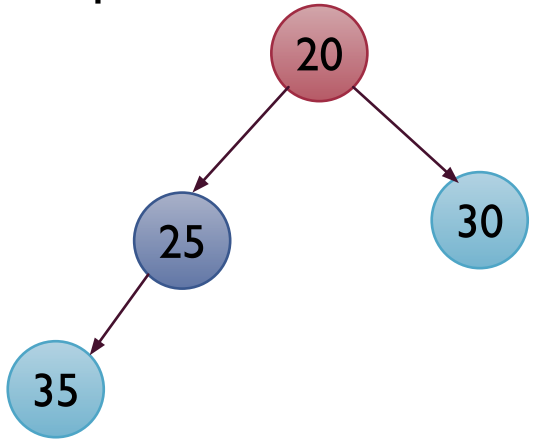


- Insert 20



- Swap with 35

- Swap with 25

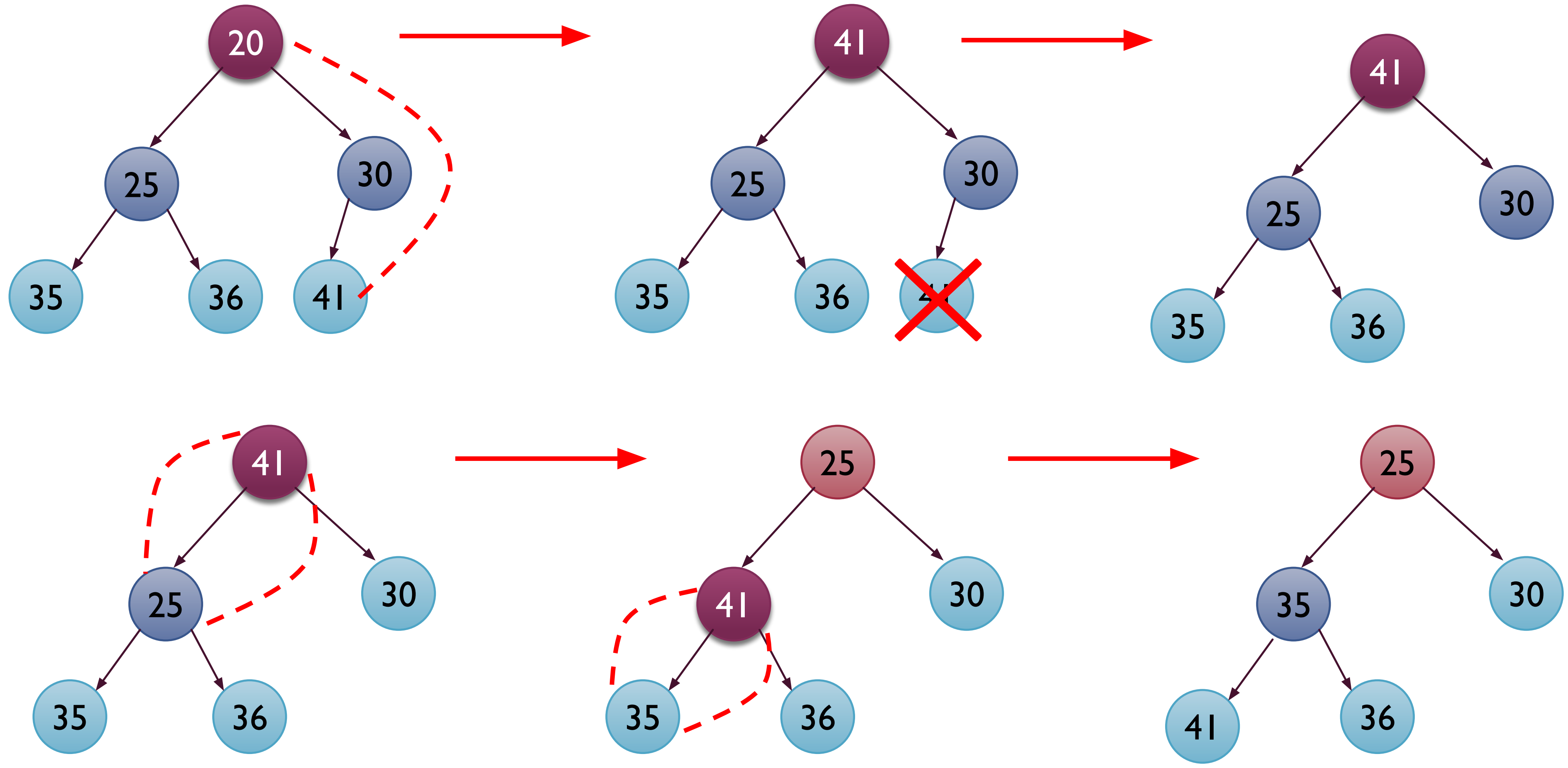


BINARY HEAP

- The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *heapify-down*, and *cascade-down*).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

HEAPIFY DOWN


20
25
30
35
36
41
41
25
30
35
36
41
41
25
30
35
36
41
25
30
35
36
25
41
30
35
36
25
35
30
41
36