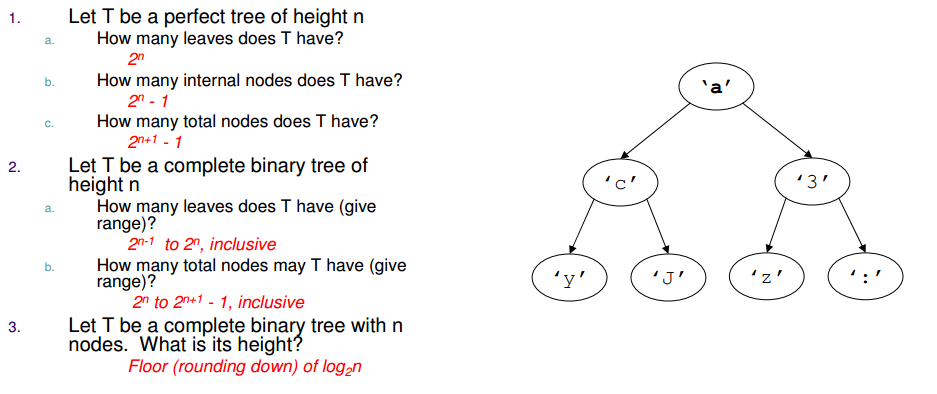
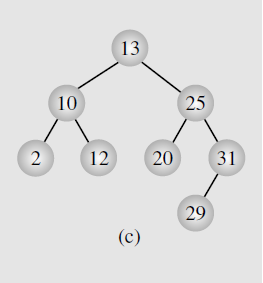
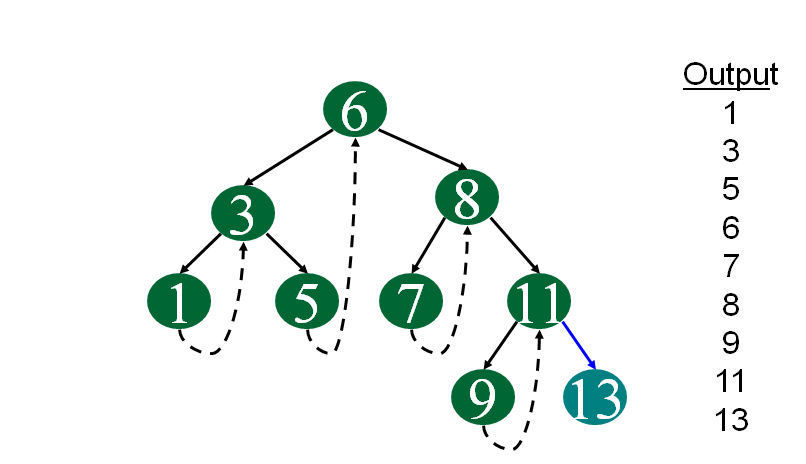
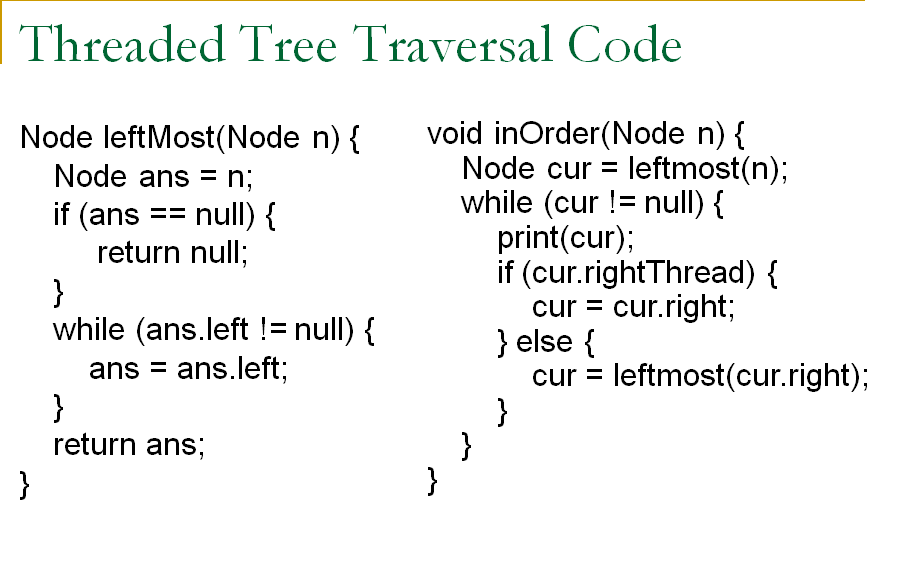
1. Path Length: Number of edges from root node to a given node.
2. Level of a node: The length of the path from root to the node + 1. This is also the number of nodes in the path.
   1. Root node is at level 0 or at level 1
3. Height of a tree: Maximum level of a tree. Empty tree has height 0.
4. Tree Nodes:
   1. Level 1 = Root node = 20 = 1, height = 0, number of nodes at level h = 1
   2. Level 2 = child nodes = 21 = 2, height = 1, number of nodes at level h = 2
   3. Level 3 = child nodes = 22 = 4 , height = 2, number of nodes at level h = 4
   4. Level (i+1) = child nodes = 2i = 2i nodes
   5. There are at most 2i nodes at level (i+1)
5. At level (i+1), there are (where i is the height)
   1. Leaves: 2i
   2. Non terminal nodes 2i-1
   3. Total = 2i+1-1
6. Number of nodes n in a complete binary tree is between 2h (minimum) and 2h+1-1 (maximum)
7. Complete Binary Tree
   1. Every level except the last level is full
   2. The leaves are as far to the left as possible.
8. Perfect Binary Tree: All leaves are present.
9. Tree Properties
10. Number of leaves (m) is greater than number of nodes (k) and m = k+1
11. Tree Traversal: There are n! Different traversals in a tree but without any order.
12. Sample Tree: 
13. Breath first traversal:
    1. Use a Queue
    2. 13, 10, 25, 2, 12, 20, 31, 29
14. Pre-Order (V, L, R)
    1. 13, 10, 2, 12, 25, 20, 31, 29
15. In-Order( L V R) : 2, 10, 12, 13, 20, 25, 29, 31
16. Post-Order (L R V) : 2, 12, 10, 20, 29, 31, 25, 13
17. Iterative pre-order, in a loop
    1. Print a node,
    2. Add its right child to a stack
    3. Add its left child to a stack.
18. Iterative in-order
    1. Collect all left most nodes, i.e. add a node to a stack, go to nodes left.
    2. If left node is empty, pop a node from stack, print it
    3. Go to its right
    4. Go back to a.
19. Iterative in-order (add node and its right child)
    1. Add node and its right child while going left.
    2. On an empty node, print nodes with no right child in a loop
    3. Print node with one right child
    4. Now switch to the right child.
20. Iterative post order
    1. Add right, and the node itself to the stack while visiting left most nodes.
    2. If left most node is null, pop a node from stack.
    3. Print the node if its parent is not next to it.
    4. Have to do a swap between a node and its right child when popping from stack.
21. Level Order Traversal
    1. Use a queue.
22. Threaded Trees: The leaf nodes’s unused left/right pointers can be used to point to inorder successor or predessor.
23. Right Threaded Tree:
    1. <http://geeksquiz.com/threaded-binary-tree/>
    2. Disk: C:\Users\kg\Desktop\interviews\Trees\ThreadedTree.ppt
    3. Each leaves unused right child points to its successor.
    4. 
    5. 
    6. Inorder right threaded tree traversal above.
    7. Preorder is the same except the node is printed while doing “leftmost”.
    8. TODO: Postorder
    9. Technique:
       1. Find left most node first.
       2. From the node’s right child, go to the left most of the nodes right child’s right child, e.g. node = n, then n->right->right
24. Convert a binary tree to a right threaded tree:
    1. <http://www.geeksforgeeks.org/convert-binary-tree-threaded-binary-tree/>
25. Morris Tree Transformation, In order
    1. If node has no left child, print the node, go to left.
    2. Otherwise, make the rightmost child of the left child’s node as parent of this node and go to left.
    3. If the rightmost child of the left node is the node itself, then break the link print the node and go to the right.
26. Morris Tree Transformation: preorder
    1. Move the print to when the link is created rather than when link is broken.
27. Morris Tree Post Order
    1. Make the root a left child of a null root.
    2. Do whats done in above
    3. When © is found, print the range of nodes in reverse order.
28. Tree Insertion recursive
    1. Always update pointers for a node’s left and right after recursion
29. Tree insertion iterative:
    1. Keep a previous pointer as the loop pointer will become null
30. Tree insertion threaded
    1. Do not go to right if a node has a thread.
    2. Right node has no successor just create a child
    3. Right node is threaded, then make the child node threaded and the right node point to child node.
31. **Diameter of a tree**:

# Binary Search Trees

## Floor

Consider a search Tree as:

Figure

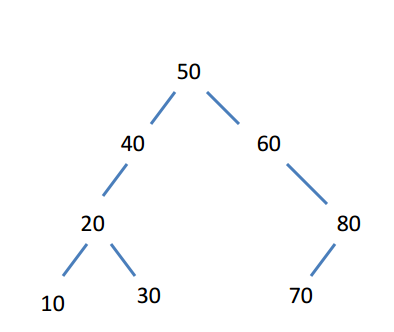
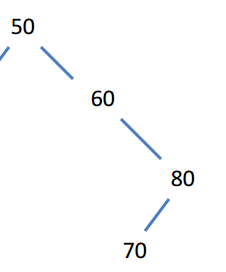
 

Figure 1

Figure 2

* Floor of a key k: If k < root, then floor of k is in the left sub tree (can be null). If k > root, then floor of k maybe in the right sub-tree or is the root itself
* Floor (45) in Figure 1: 45 < 50, so the floor must be in the left sub tree of 50. Here it is 40. The root node, i.e. 50 can not be the floor so it is in the left sub tree, can be null (Figure 2)
* Floor (45) in Fig2: 45 < 50. However, node (50) has no left child. This means floor is null. 50 can not be the floor of 50 as 50 > 45.
* Floor (62): 62 > 50, so the floor may be in the right sub tree. If it is not, then 50 is the floor of 62.

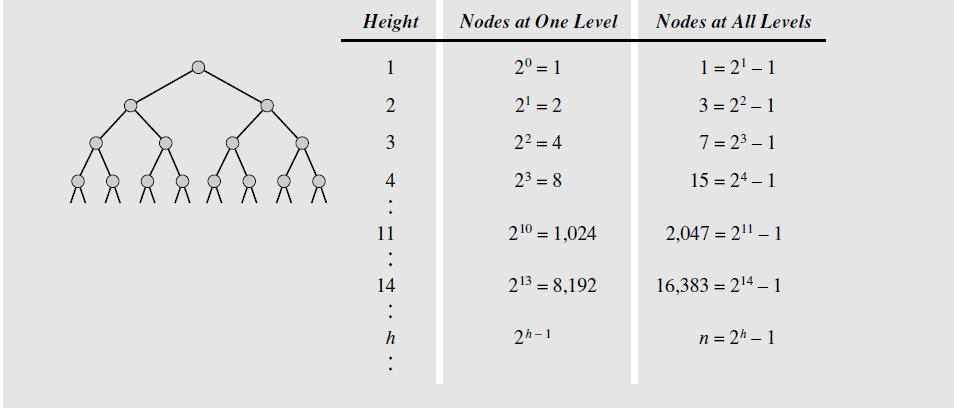
## Ceil

* Ceil of a key:
  + k == root: return root.
  + k < root, ceil maybe in the left sub-tree or is the root itself.
  + k > root, ceil is in the right sub-tree or is null the root cannot be the ceiling of k as k > root.
* Ceil(45): 45 < 50, 50 is the ceil of 45.
* Ceil(58): 60 as 58 > 50, so the ceil MUST be in the right sub-tree.

## Order Statistics

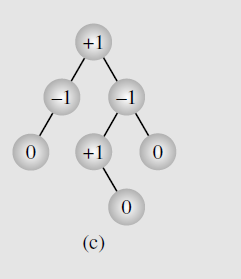
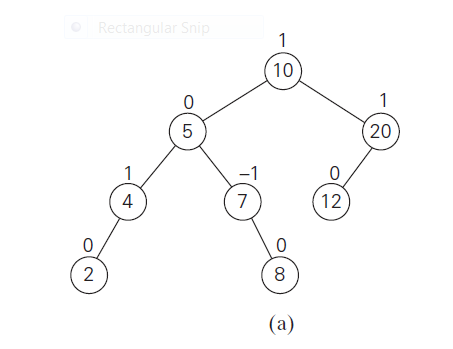
<http://www.zrzahid.com/kth-smallestminimum-element-in-a-bst-rank-of-a-bst-node/>

# Height Balanced Trees

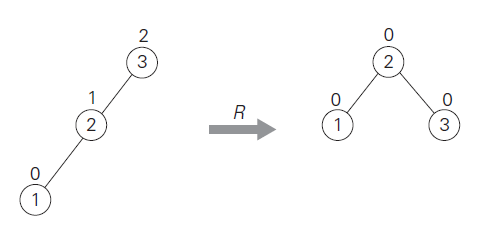


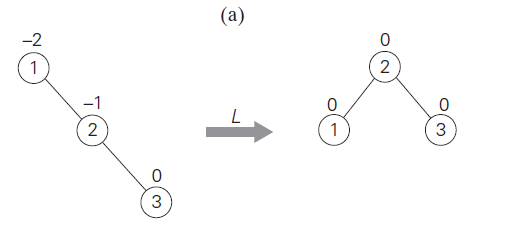
* Height = h, nodes at that level = 2(h-1)
* Level = h, total number of nodes = 2h-1

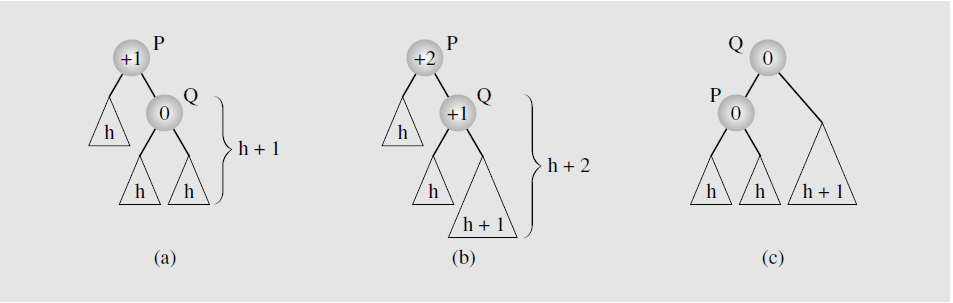
# AVL Trees

* <http://www.geeksforgeeks.org/avl-tree-set-1-insertion/>
* A binary tree where the height of the left subtree and the right subtree of every node differs by at most 1.
* Balance Factor: Difference between the height of the left and the right sub trees.
* 
* 

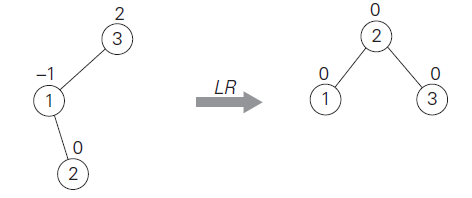
|  |  |  |  |
| --- | --- | --- | --- |
| Node | Left Height | Right Height | Balance Factor (Lh-Rh) |
| 2 | -1 | -1 | -1-(-1) = 0 |
| 4 | 0 | -1 | 0-(-1) = 1 |
| 5 | 1 | 1 | 1-1=0 |
| 10 | 2 | 1 | 2-1=1 |

* Start balancing at the node which has a balance factor of -2 or 2 and go up the tree
* Minimum number of nodes in an AVL Tree:
  + AVL(h) = AVL(h-1) + AVL(h-2) +1
* http://www.eternallyconfuzzled.com/tuts/datastructures/jsw\_tut\_avl.aspx
* Insertion: <http://www.geeksforgeeks.org/avl-tree-set-1-insertion/>
* Deletion: <http://www.geeksforgeeks.org/avl-tree-set-2-deletion/>
* Tree Rotation Example:
  + <http://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf>
* Left Left Case (Single Right Rotation)
  + Node is inserted in the left sub tree of the left child of root.
  + Perform a right rotation on the root.
  + The root has a balance factor of +1 before the insertion.
  + Rotate the edge connecting the root and its left child rightwards
* 
* Right Right Case (Single Right Rotation)
  + Node is inserted in the right sub tree of the right child of the root
  + Perform left rotation on the root
  + The root of the sub tree has a balance factor of -1 before insertion.

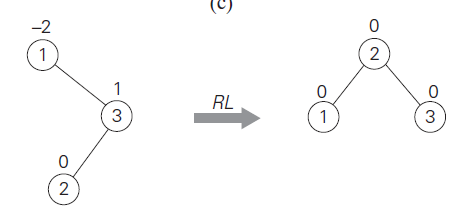


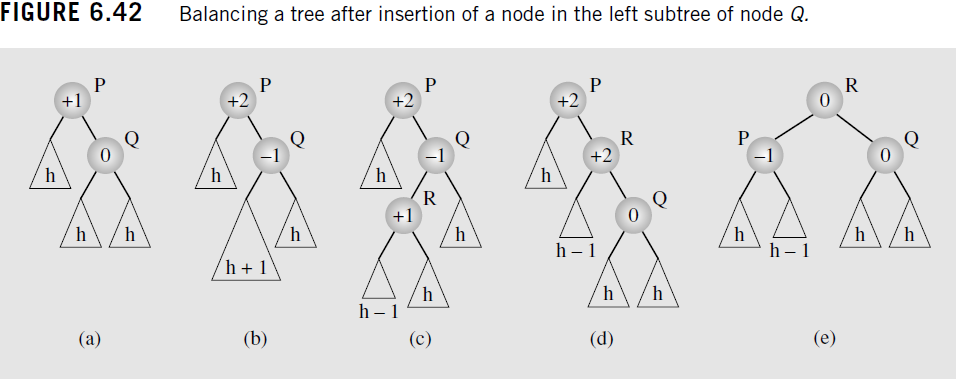


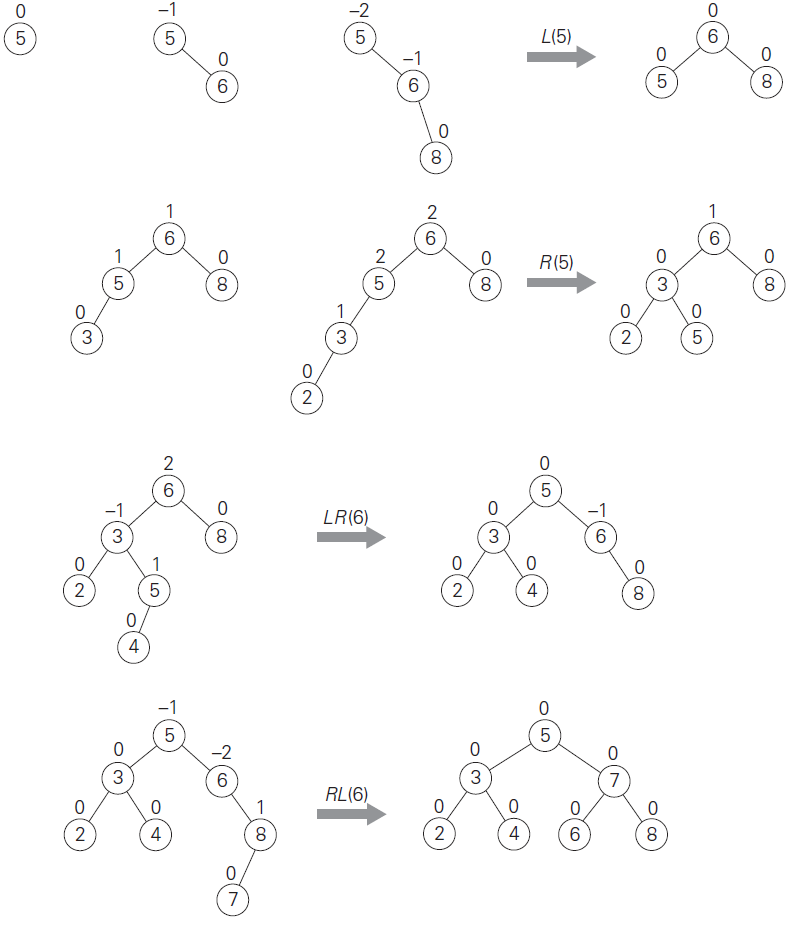
* Left Right Case (double left right rotation)
  + Node is inserted in the right sub-tree of the left child of the root
  + The root of the sub-tree has a balance factor of 1 before insertion.
    - Perform left rotation on the left child of the root.
    - Perform right rotation on the root.



* Right Left Case:
  + Node is inserted in the left sub-tree of the right child
  + Root of the sub tree has a balance factor of -1 before insertion.
    - Perform right rotation on the right child of the root
    - Perform left rotation on the root.





* Left or Right rotations take constant time as only a few pointers are updated
* Updating the height or getting the balance factor takes constant time
  + Start with the first node where the imbalance occurs, and keep moving up towards the root. Not all the nodes in the path have to be updated.
* Time complexity is same as a binary tree O(h) where h is the height
* Height of a balanced binary tree is O(log n) where n is number of nodes.
* Time complexity of insert in AVL tree is O(log n)
* AVL Tree is more balanced than a Red-Black tree
* There might be more rotations during insertions/deletions in AVL tree compared to a Red Black tree, this is why in STL and Java, Red Black tree is used.
* If insertions/deletions are less frequent, then use AVL Tree
* Building an AVL Tree (5, 6, 8, 3, 2, 4, 7) successive insertion.
* 

LR(6)

## XOR Trees

* + <parent XOR Left child>, data , <parent XOR Right child>
  + This allows a parent pointer to be stored as well
  + <parent XOR Left Child> XOR parent = Left child
  + a^b = c
  + a^c = b
  + b^c = a
  + similar to doubly linked list where there is one pointer that is (Left Node XOR Right Node).
* Expression Trees
  + TODO (Infix, Prefix, PostFix)

# Binary Search Trees

* In-order predecessor of a node: The highest value in its left tree. This is also the right most node in the left tree.
* In-order successor of a node: The smallest value in the right child of the node.
* Binary Tree Insertion:
  + If inserting in the left subtree, update the left subtree pointer after inserting into the left subtree, e.g. (root->left = insert(root->left, data))
* Always update a node->left when an item is deleted from node->left and update node->right when an item is deleted from node->right.
* Tree Deletion (No child or one child) Simple Case
* Tree Deletion (Node has left and right child)
  + Replace node’s data with largest value from node->left (or)
  + Replace node’s data with smallest value from node->right
  + Then delete the new node->data (i.e. the smallest value in node->left) from node->left (and vice versa)
* Tree Deletion (Node has two children)
  + Replace node with left subtree’s largest node (NL)
  + Delete NL from node->left.
  + Vice versa for right subtree except use the smallest value from right subtree.
* Shortest path between two nodes = LCA between two nodes.
* Check if a tree is a BST
  + Check if node->data > node->left->data && node->data < node->right->data
  + Check if node->data > max(node->left) && node->data < min (node->right) && node->left is a bst and node->right is a bst
  + O(n^2)
* Check if a tree is a BST (Improved)
  + Check(Node, min, max) (NOTE: Starter(check(root, MIN, MAX))
  + If (node->data > min && node->data > max && check(node->left, min, node->data) && check (node->right, node->data, max)
  + O(n)
* Check if a tree is BST (Improved)
  + O(n)
  + Inorder traversal with previous node’s value as input.
  + Use INT\_MIN while traversing the left sub-tree.
  + Use node->data when traversing right sub-tree
* Convert BST to Circular DLL (TODO)
* Given a DLL, convert it to a BST (TODO)
* BST: Floor and Ceiling
* Union two BSTS: parent pointers are available so similar to merging two lists. Traverse each of the tree based on which value is greater or less.
* Union/Intersection: No parent pointer
  + Convert each tree into an array or a linked list O(m+n)
  + Find union/intersection O(m+n)
  + Recreate a binary search tree from this O(m+n)
  + Use a hash table
* Print all elements between k1 and k2
  + Use level order
  + Recursive
* Binary Tree solutions:
  + Always update the left or right subtree if something gets inserted in the left or right subtree
  + Can I use a preorder-postorder-inorder traversal?
  + Can I use a stack or a queue?
  + Can I use a level order (queue) traversal?
  + Insert sentinel value after a root and reinsert it if it is found when the queue is not empty?
  + Use a hash table?

## AVL Tree Applications

* <http://www.geeksforgeeks.org/median-of-stream-of-integers-running-integers/>
* <http://www.geeksforgeeks.org/maximum-of-all-subarrays-of-size-k/>
* <http://www.geeksforgeeks.org/count-smaller-elements-on-right-side/>
* AVL Tree is:
  + A binary search tree
  + The max difference between height of left sub-tree and right sub-tree is 1.
* Minimum Number of nodes in a tree of height h is:
  + T(h) = T(h-1) + T(h-2) +1
  + 1.44logn or logn
* Maximum number of nodes in a tree of height h is:
  + T(h) = T(h-1) + T(h-1)+1
  + O(2^h) = O(logn)
* At insertion or deletion, how many nodes in an AVL tree need to be updated?
* AVL Tree Insertions:
  + Insert in left sub-tree of the left child of X
  + Insert in right sub-tree of the left child of X
  + Insert in the left sub-tree of the right child of X
  + Insert in the right sub-tree of the right child of X

# Heaps

1. Min Priority Queue:
   1. Smallest element on the top A[1].
   2. A[i] <= A[2\*i] and A[i] <= A[2\*i+1]
2. Max Priority Queue
   1. Largest element on the top
   2. A[i] >= A[2\*i] and A[i] >= [2\*i+1]
3. Find: O(logn)
4. Item (i)
   1. Parent = i/2
   2. Left = 2\*i
   3. Right = 2\*i+1
5. Heapify (Max Priority Queue)
   1. Swim (node) : if node > parent
   2. Sink (node) : if node < left or right child.
6. Insert (Max Priority Queue)
   1. Add key to the end
   2. Call sink(N)
7. Remove (Max Priority Queue)
   1. Remove top, replace by item at A[N]
   2. Call Sink(1)
8. Multiway heap, e.g. 3 way: k🡺 3k-1, 3k, 3k+1.
9. Merge sorted arrays using a Heap: Add item from each array into a max or min heap. Then remove the first one, and read another item from source array of the item.
10. HeapSort (nLogn)
    1. Create a Max Heap.
    2. Move top element to end, swap with last and call sink.
11. Building a heap of array of size N
    1. Call sink from [N/2,1]
12. Dynamic Median finding:
    1. Keep median as a value v
    2. Have a min heap with values > v
    3. Have a max heap with values < v
    4. Add a new key to the appropriate heap (i.e. if k < v, add to max heap), if key > v, add to min heap. Replace v with root of the appropriate heap