Binary Tree

Kriengsak Treeprapin

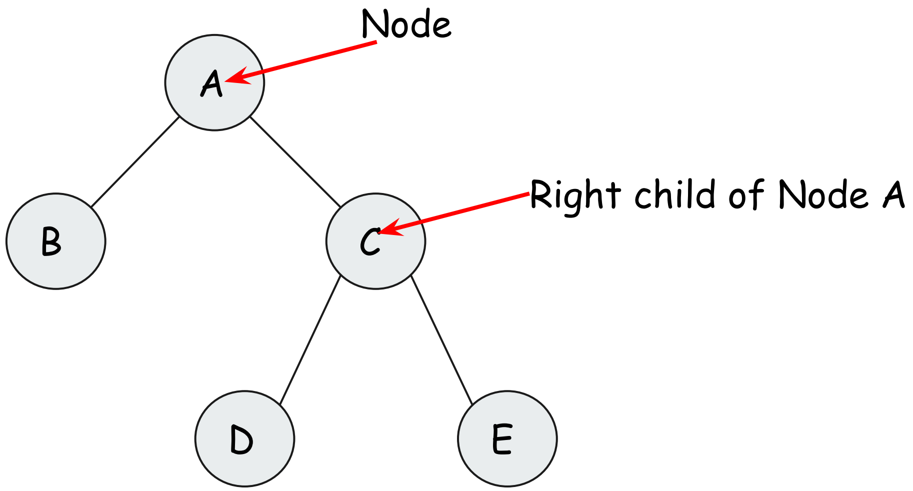
Agenda

- Introduction
  - Properties and types of binary tree
  - Handshaking lemma
- Tree Traversal
- Binary Search Tree
- Self-balancing Binary Search Tree
- Misc

Binary Tree

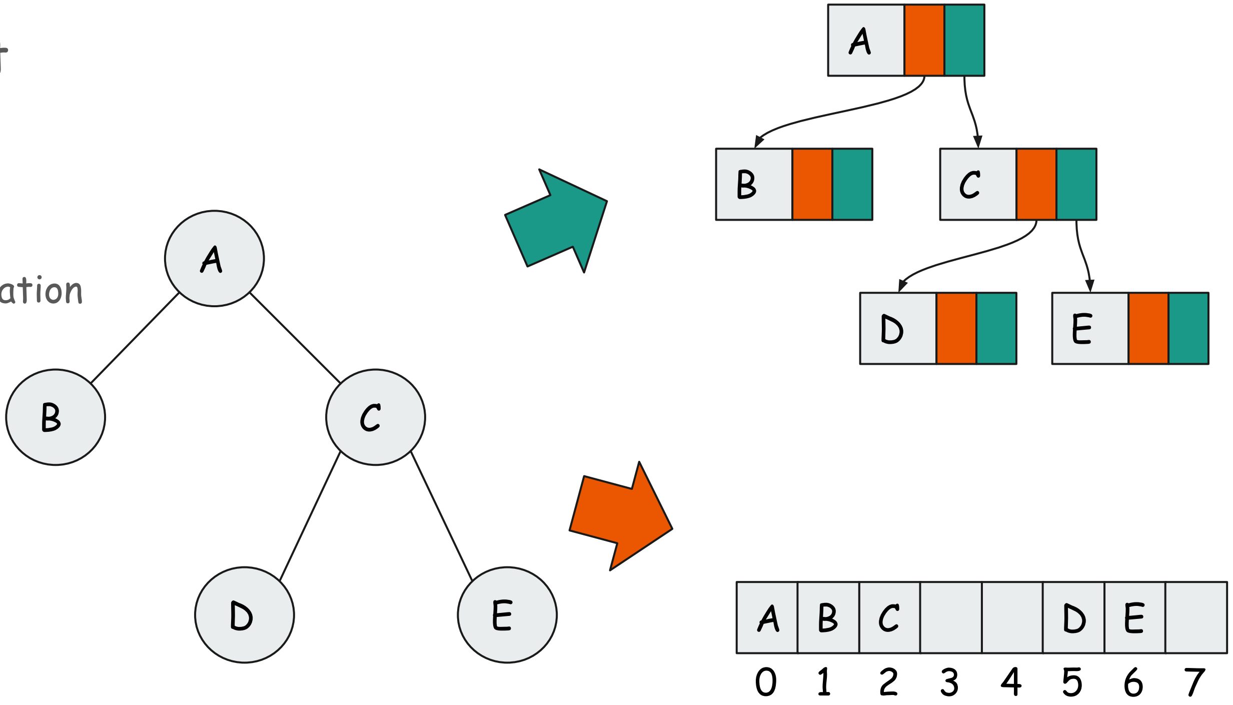
A tree in which no node can have more than two children (typically name them the left and right child).

Binary Tree



Implementation of Binary Tree

1. With Linked List
    a. save memory
2. With Array
    a. easy implementation



Binary Tree : Properties

1. The maximum number of nodes at level ‘k’ of a binary tree is 2^{k-1}.
2. Maximum number of nodes in a binary tree of height ‘h’ is $2^h - 1$.
3. In a Binary Tree with N nodes, minimum possible height or minimum number of levels is $\lceil \log_2(N+1) \rceil$.
4. A Binary Tree with L leaves has at least $\lceil \log_2 L \rceil + 1$ levels.
5. In Binary tree, number of leaf nodes is always one more than nodes with two children. ⇒ Handshaking lemma

Handshaking lemma

The statement that every finite undirected graph has an even number of vertices with odd degree.

in a party of people some of whom shake hands, an even number of people must have shaken an odd number of other people's hands.

A consequence of the degree sum formula :

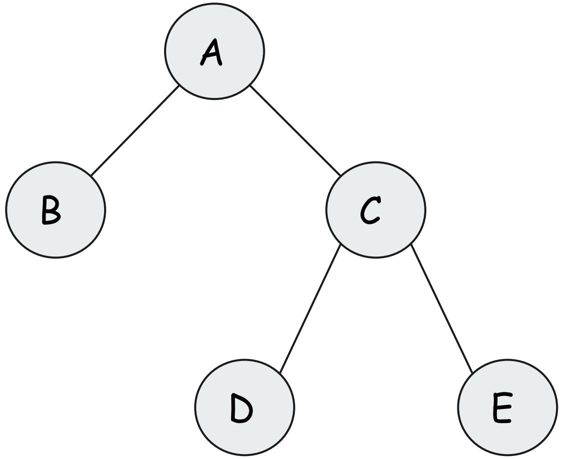
$$\Sigma_{v \in V} \deg(v) = 2|E|$$

when V is vertex set and E is edge set

Types of Binary tree

Full Binary Tree :

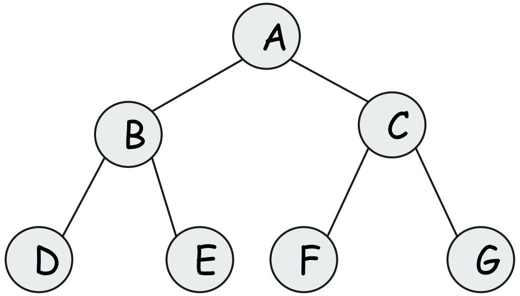
A Binary Tree is full if every node has 0 or 2 children.

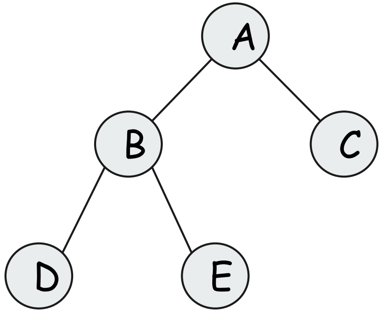


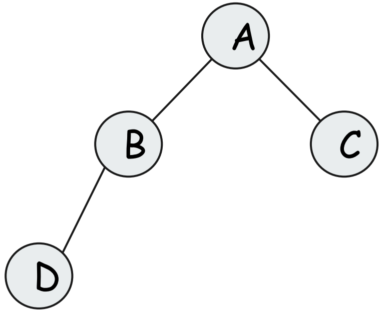
Types of Binary tree

Complete Binary Tree :

A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.



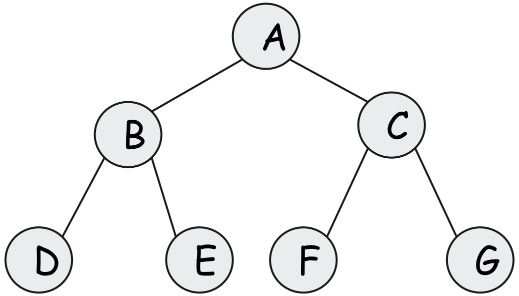




Types of Binary tree

Perfect Binary Tree :

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.



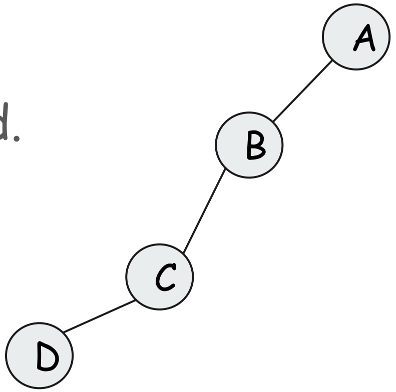
Types of Binary tree

Balanced Binary Tree :

A binary tree is balanced if height of the tree is O(Log n) where n is number of nodes.

Degenerate tree (Worst case binary tree):

A Tree where every internal node has one child.

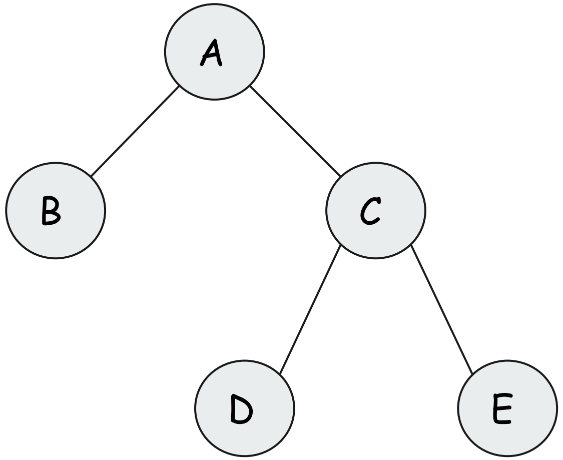


Tree Traversal

Tree Traversal

Tree can traversed in 4 ways :

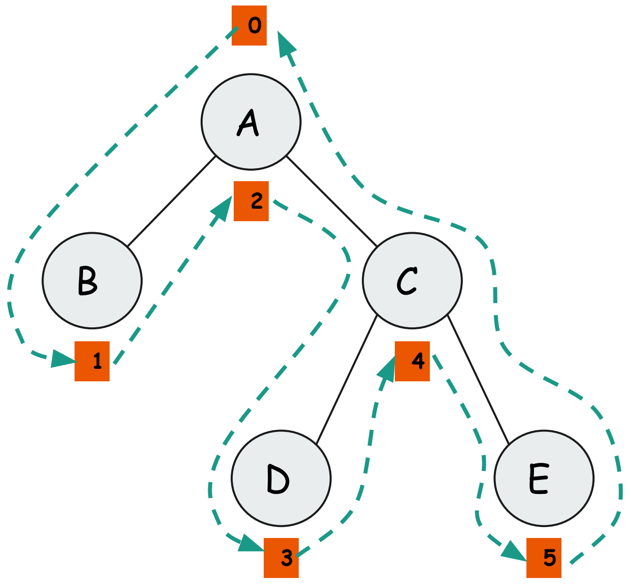
1.	Inorder traversal	⇒	DFS
2.	Preorder traversal	⇒	DFS
3.	Postorder traversal	⇒	DFS
4.	Level order traversal	⇒	BFS



Inorder Traversal

Algorithm :

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

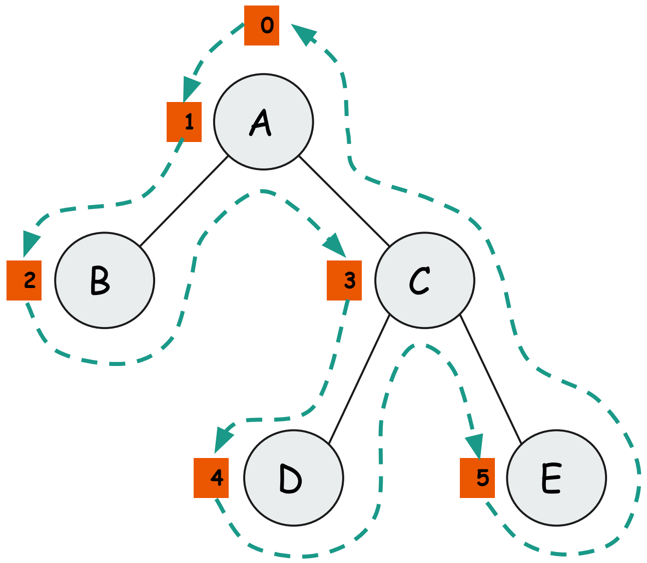


[B, A, D, C, E]

Preorder Traversal

Algorithm :

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

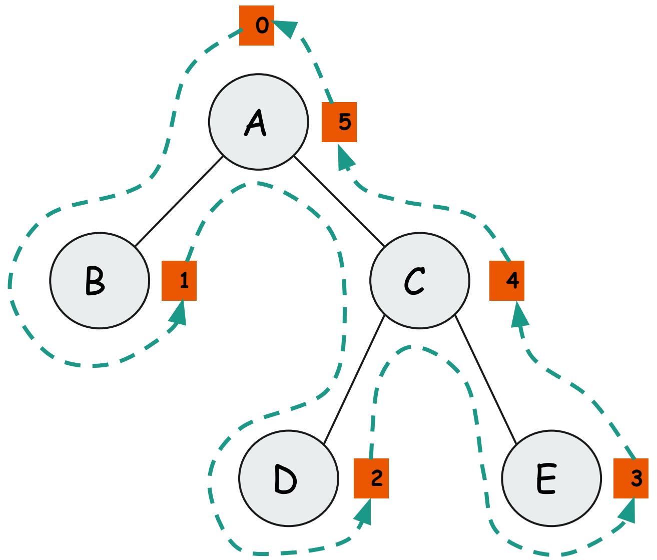


[A, B, C, D, E]

Postorder Traversal

Algorithm :

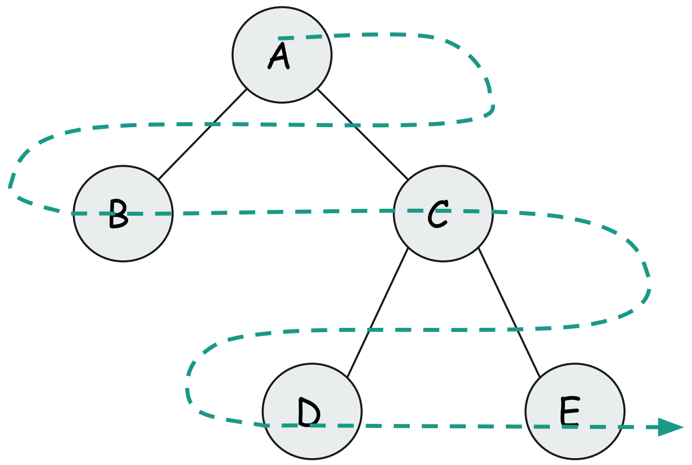
1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.



[B, D, E, C, A]

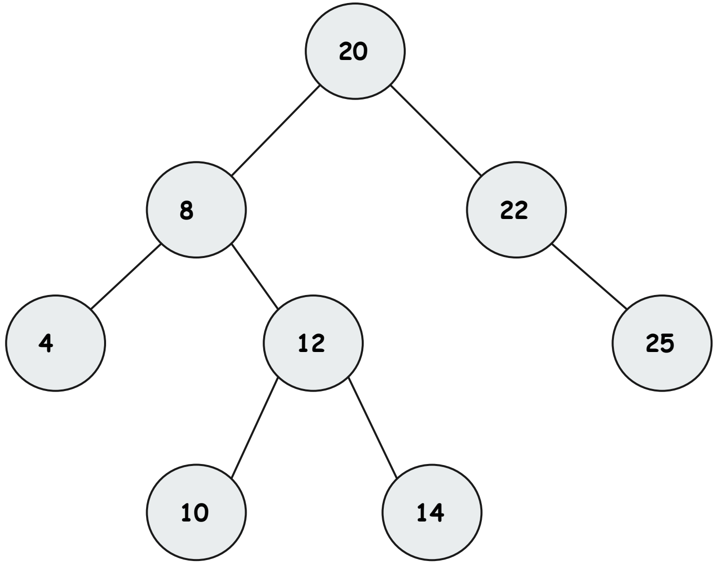
Level Order Traversal

[A, B, C, D ,E]



Challenge of Tree Traversal

print boundary nodes of the binary tree Anti-Clockwise starting from the root. For example, boundary traversal of the following tree is [20, 8, 4, 10, 14, 25, 22]



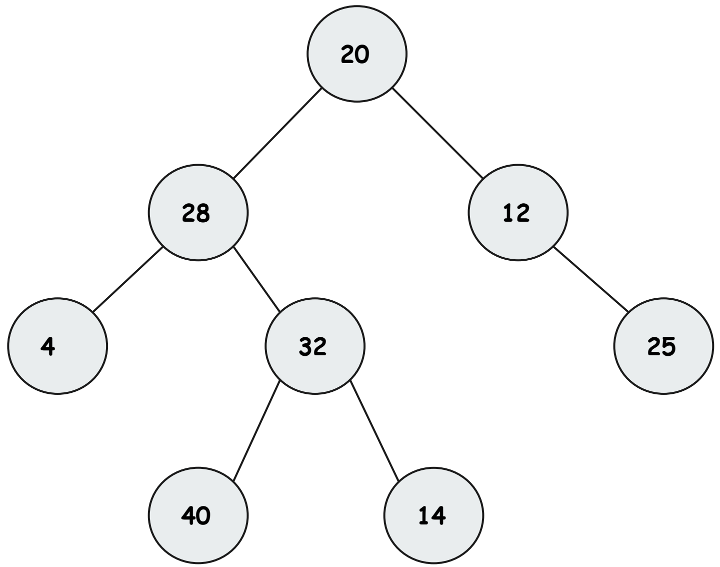
Binary Search Tree

Binary Search tree

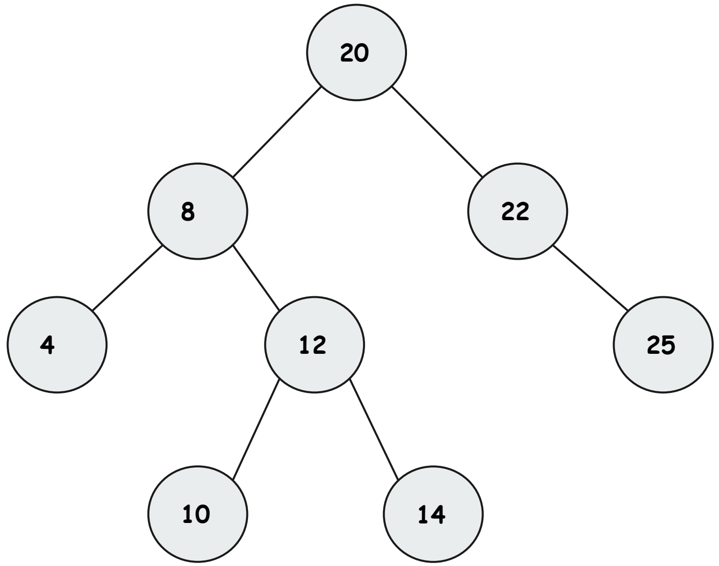
A node-based binary tree which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

Binary Search tree



Not Binary Search Tree



Binary Search Tree

Searching A Key

Algorithms :

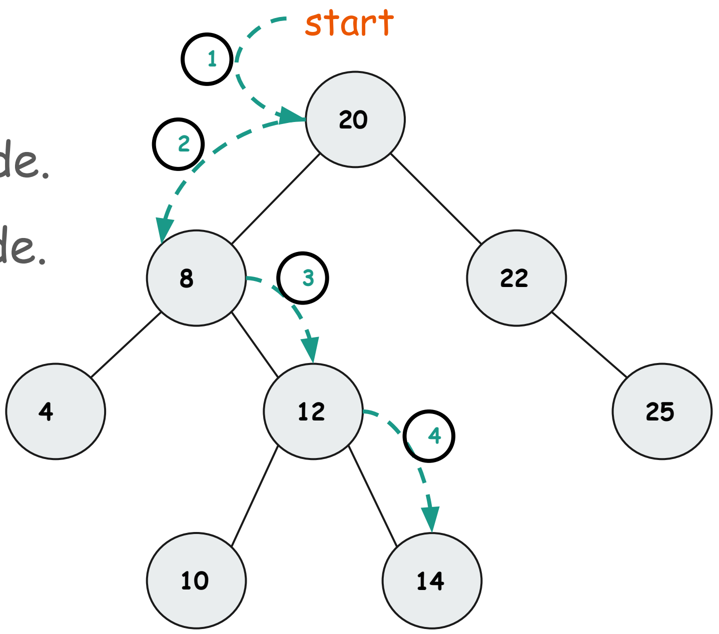
- Compare key with root node, if the key is present at root, we return root.
- If key is greater than root's key, we recur for right subtree of root node.
- Otherwise we recur for left subtree.

Try this : https://visualgo.net/en/bst

Searching A Key

EX) Searching for “14”

1) compare 14 with 20 ⇒ go to left node.
2) compare 14 with 8 ⇒ go to right node.
3) compare 14 with 12 ⇒ right node.
4) compare 14 with 14 ⇒ found !



Insertion of a Key

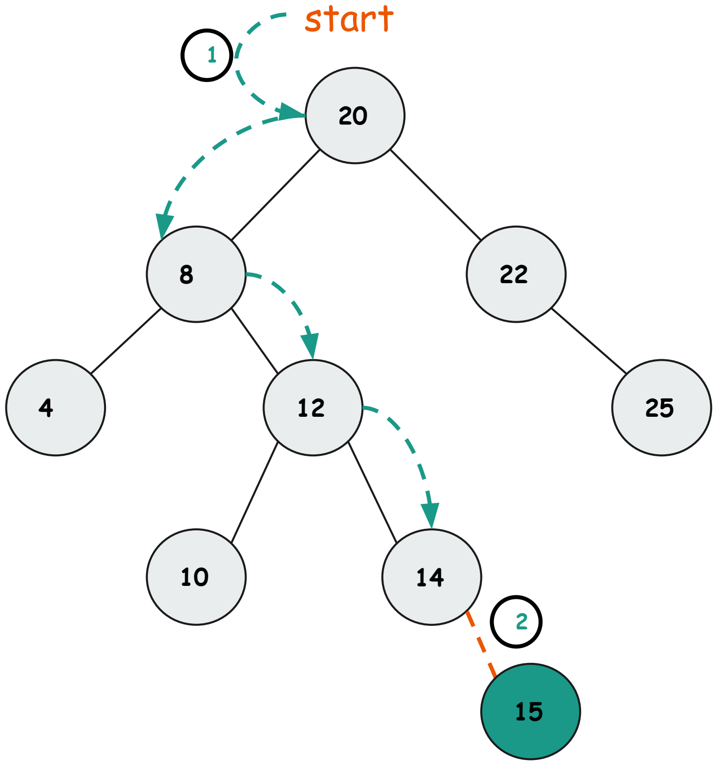
Algotithm :

- Searching a key from root till hit a leaf node.
- Add the new node as a child of the leaf node.

Insertion of a Key

EX) Insert key “15”

1) searching key 15 till reach node 14
2) add new node 15 as the right child of node 14



Delete a Key

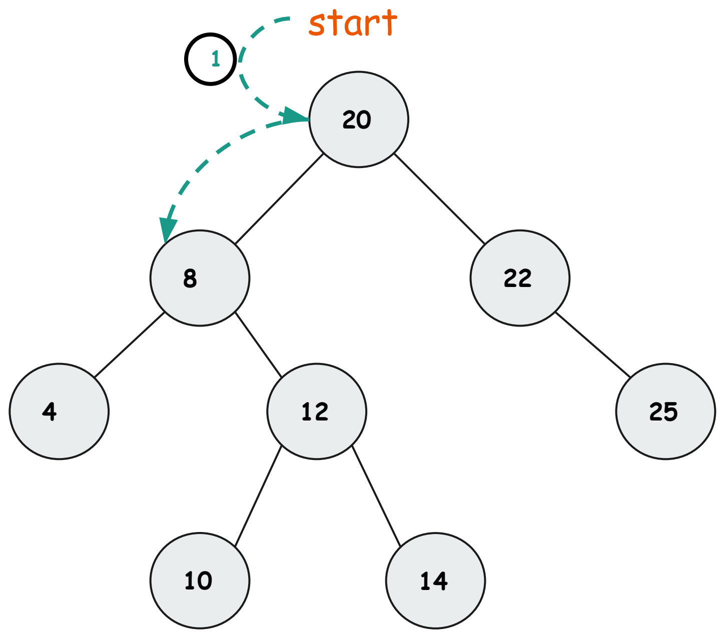
Algotithm :

- Searching a key from root and delete it.
- replace node by...
    - If it has no children, replace by NULL.
    - If it has 1 child, replace by that child.
    - If it has 2 children, replace by the node with the smallest value in its right subtree.

Delete a Key

EX) Delete key “8”

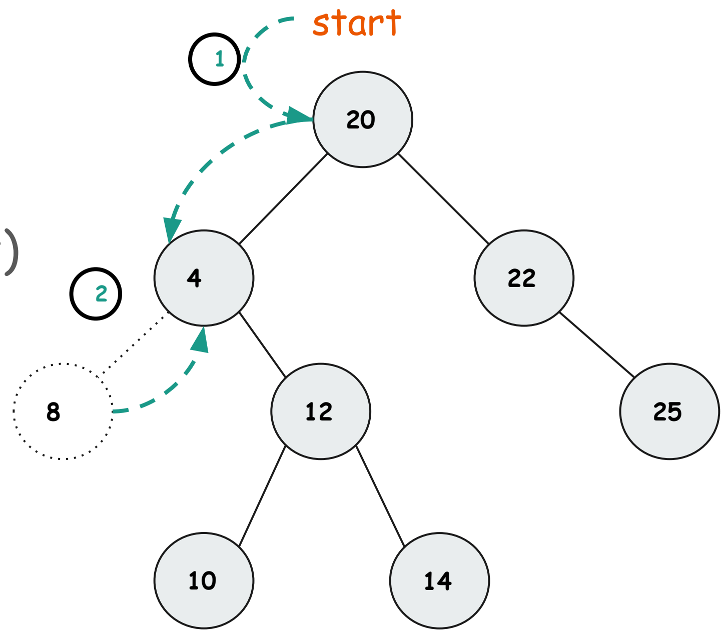
1. searching key 8 till reach node 8



Delete a Key

EX) Delete key "8"

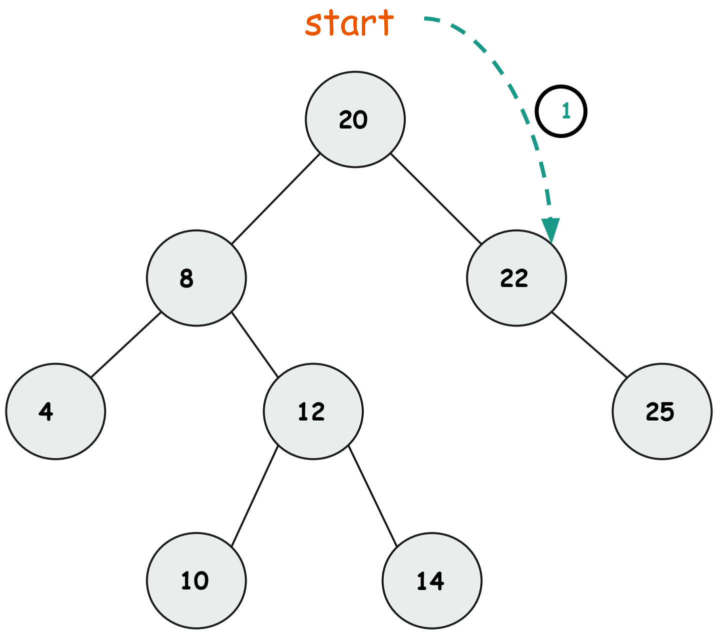
1. searching key 8 till reach node 8
2. replace with node 4 (the right-most)



Delete a Key

EX) Delete key “22”

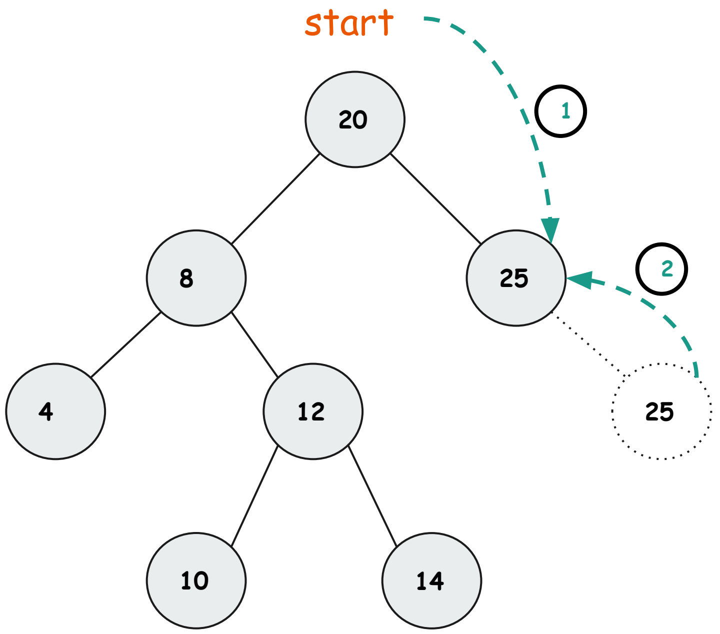
1. searching key 22 till reach node 22



Delete a Key

EX) Delete key "22"

1. searching key 22 till reach node 22
2. replace by node 25 (its child).



Runtime Analysis of Binary Search tree

- All BST operations are O(d), where d is tree depth.
- Best case running time is O(log N)
- Worst case running time is O(N)

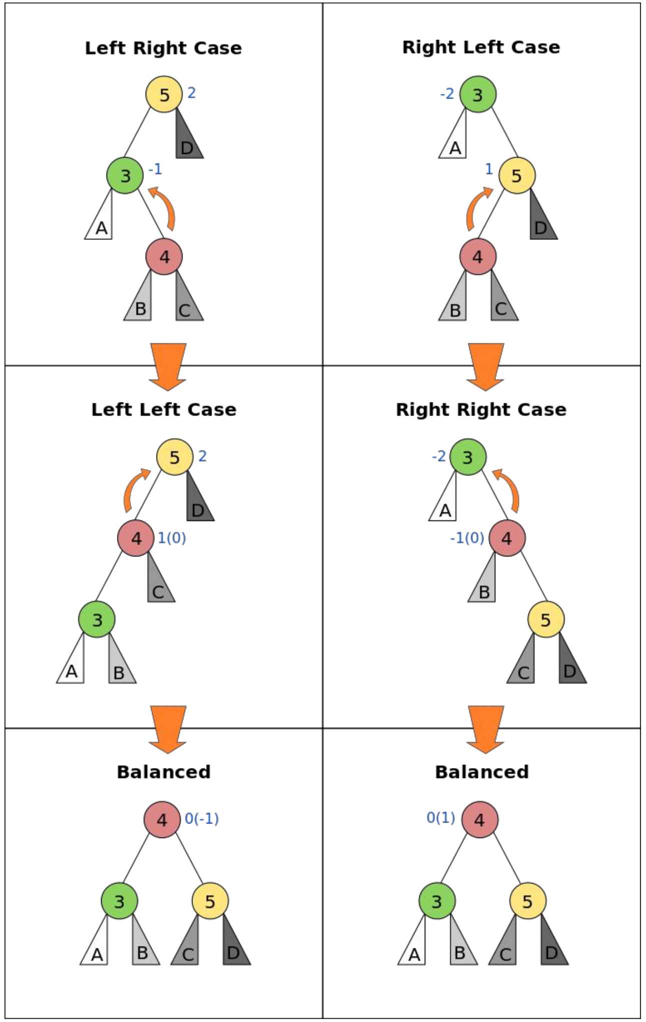
Self-balancing Binary Search Tree

Self-balancing Binary Search Tree

a self-balancing binary search tree is any node-based binary search tree that automatically keeps its height small in the face of arbitrary item insertions and deletions.

⇒ Kept the performance in $O(\log N)$

AVL Tree

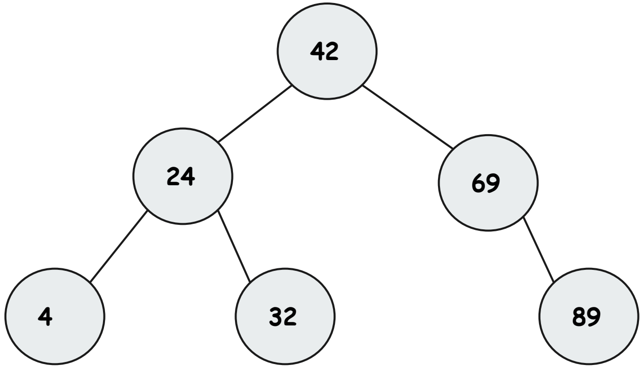

Left Right Case
Right Left Case
Left Left Case
Right Right Case
Balanced
Balanced
5
2
3
-1
4
A
B
C
D
-2
1
1(0)
-1(0)
0(-1)
0(1)

AVL Tree

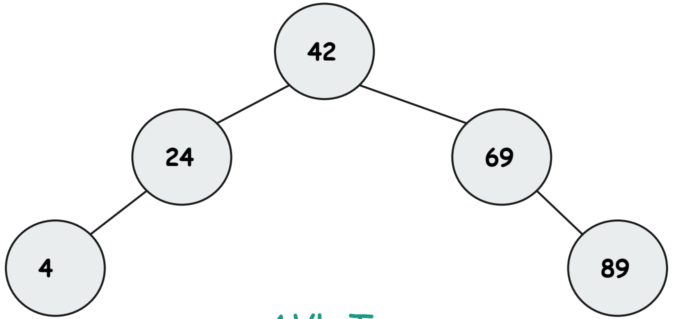
An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition.

Identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1.

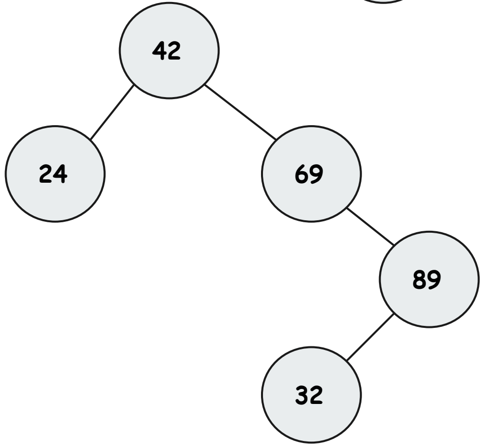
AVL Tree



AVL Tree

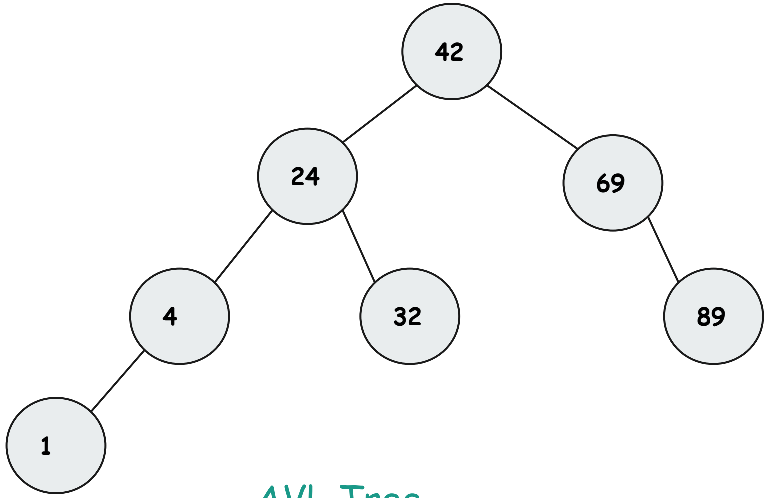


AVL Tree

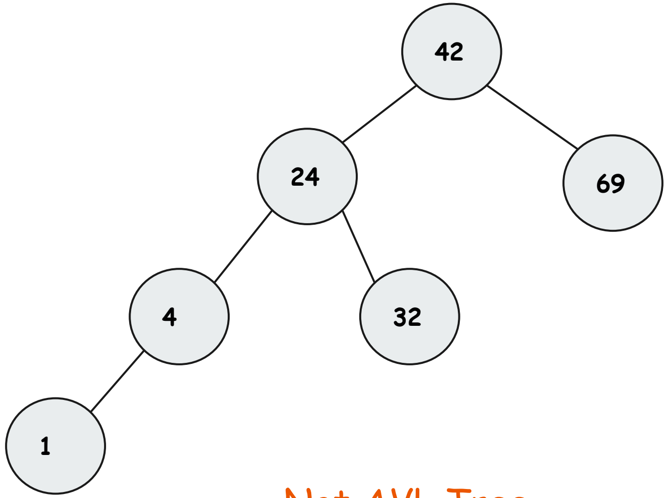


Not AVL Tree

AVL Tree



AVL Tree



Not AVL Tree

Balance Factor in AVL Tree

AVL tree checks the height of the left and the right subtrees and assures that the difference is not more than 1.

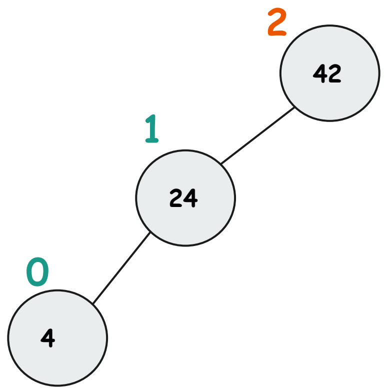
This difference is called the Balance Factor.

BalanceFactor = **height**(left-subtree) - **height**(right-subtree)

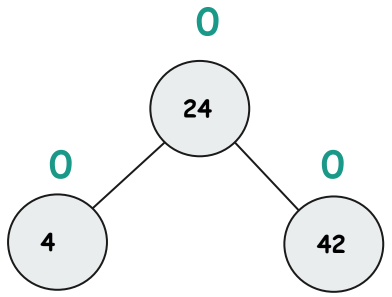
Balance Factor in AVL Tree

AVL tree checks the height of the left and the right subtrees and assures that the difference is not more than 1.

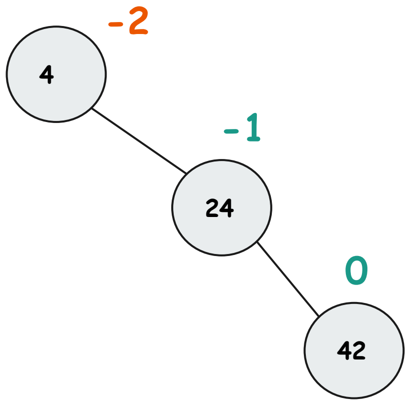
This difference is called the Balance Factor.



Not balance



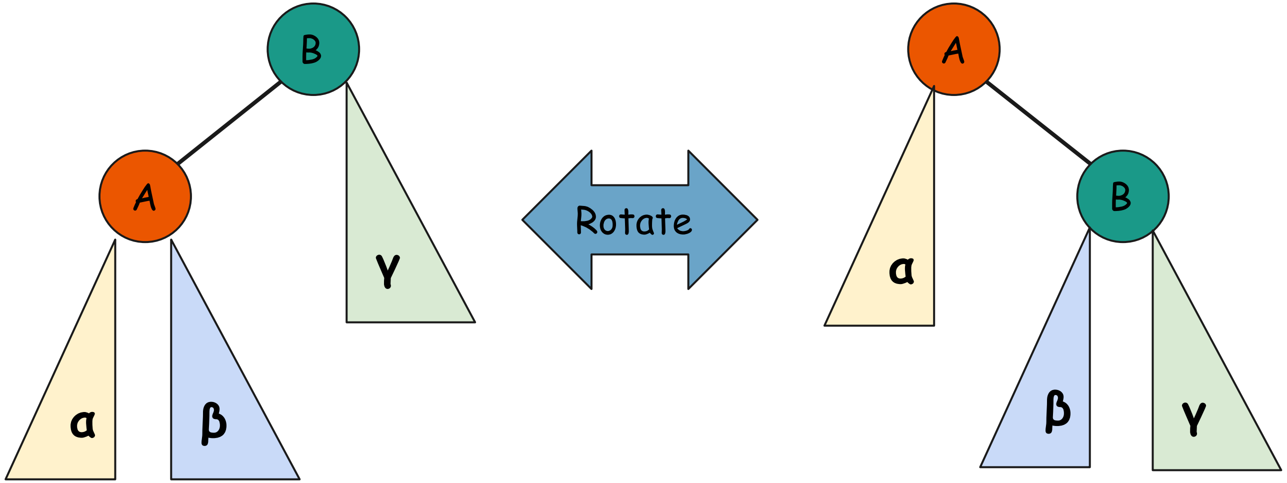
Balance



Not balance

Rotations in AVL Tree

When the balance factor larger than 1, AVL tree do rotations to balance itself.



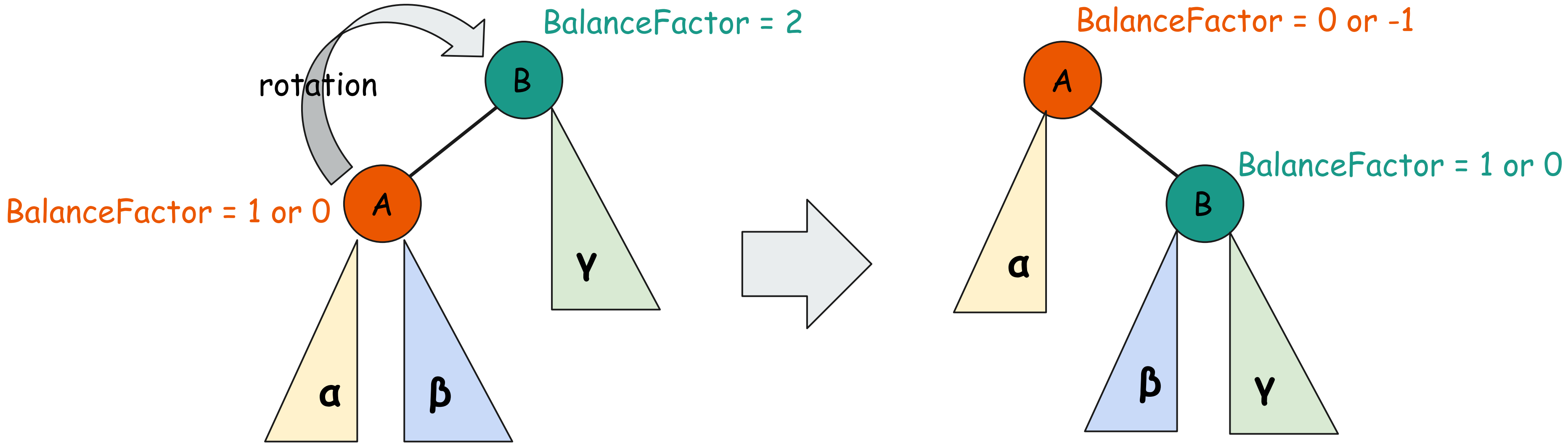
Rotations in AVL Tree

AVL tree perform the following four kinds of rotations for re-balance:

- Right rotation
- Left rotation
- Left-Right rotation
- Right-Left rotation

Single Right Rotation

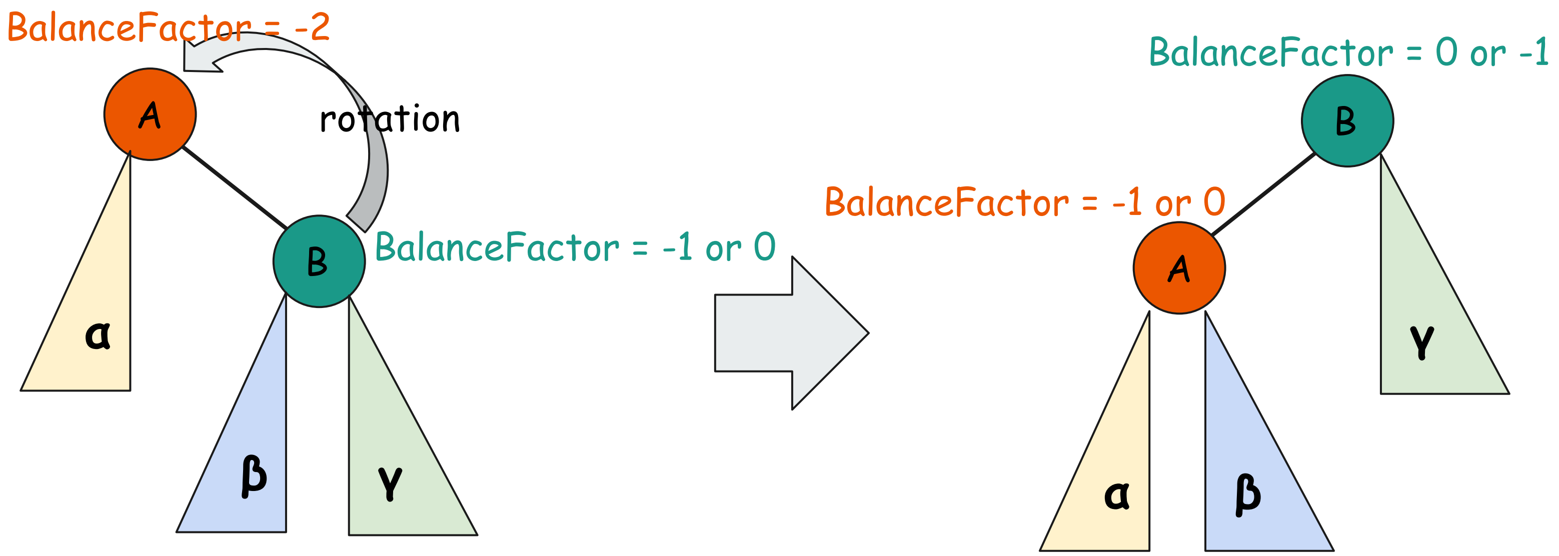
- If a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation .



Single Left Rotation

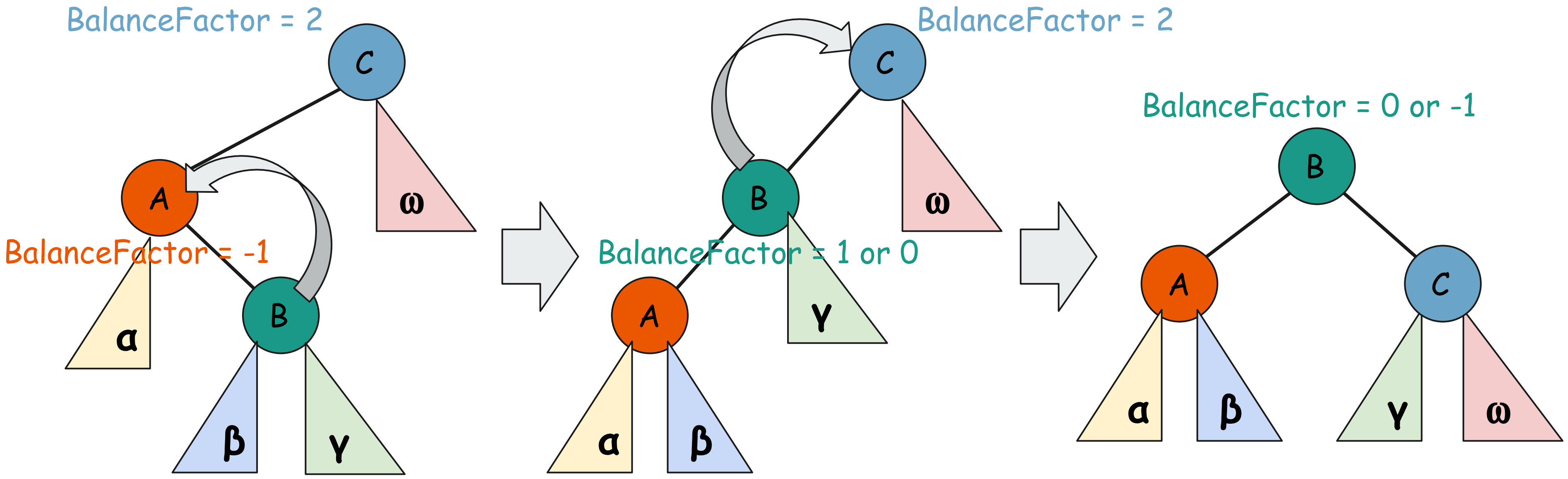
- If a node is inserted in the right subtree of the right subtree.

  The tree then needs a left rotation .



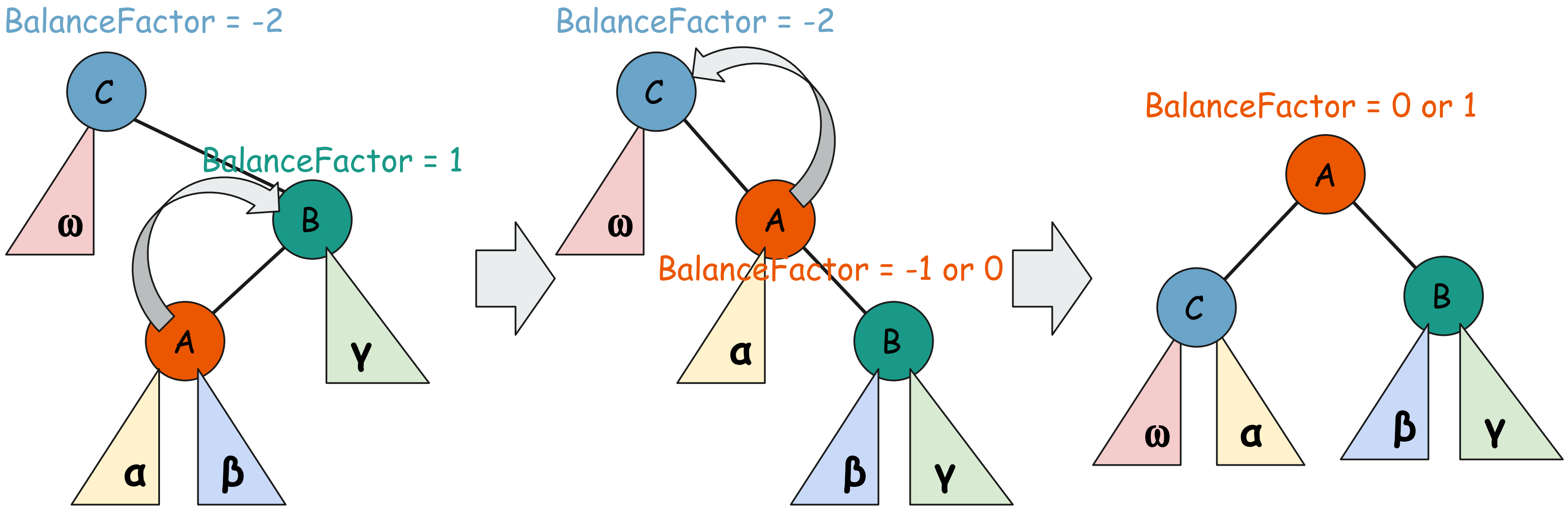
Left-Right Rotation

- If a node is inserted in the left subtree of the right subtree. The tree then needs a left-right rotation .



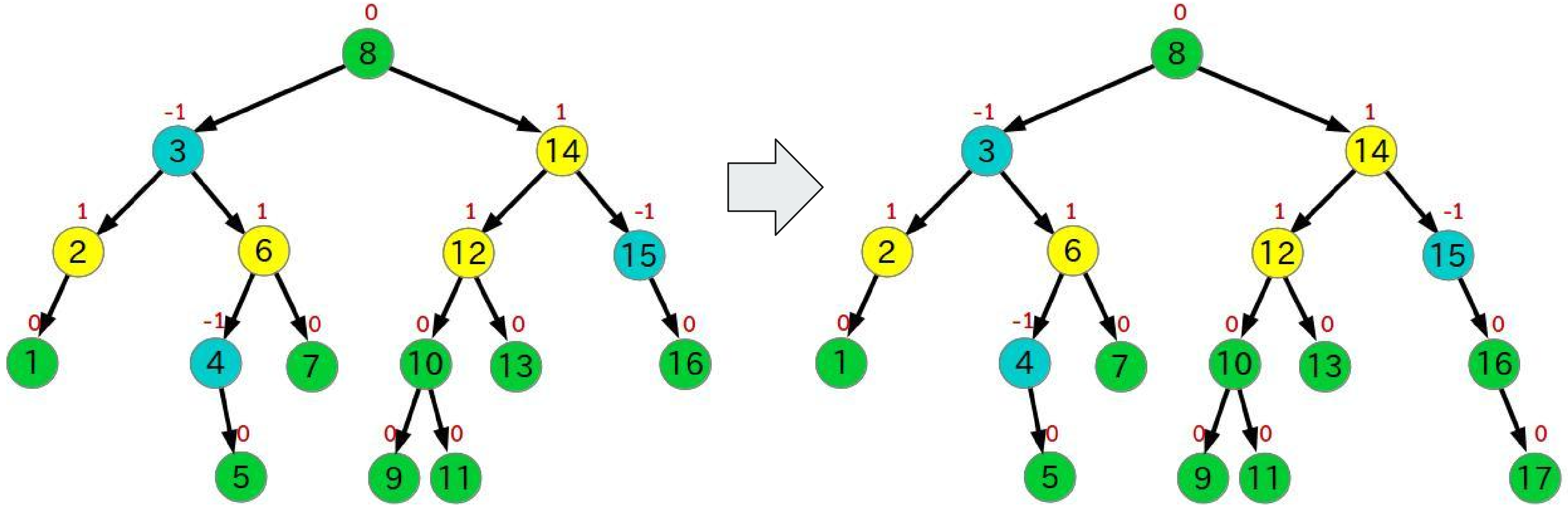
Right-Left Rotation

- If a node is inserted in the right subtree of the left subtree. The tree then needs a right-left rotation .



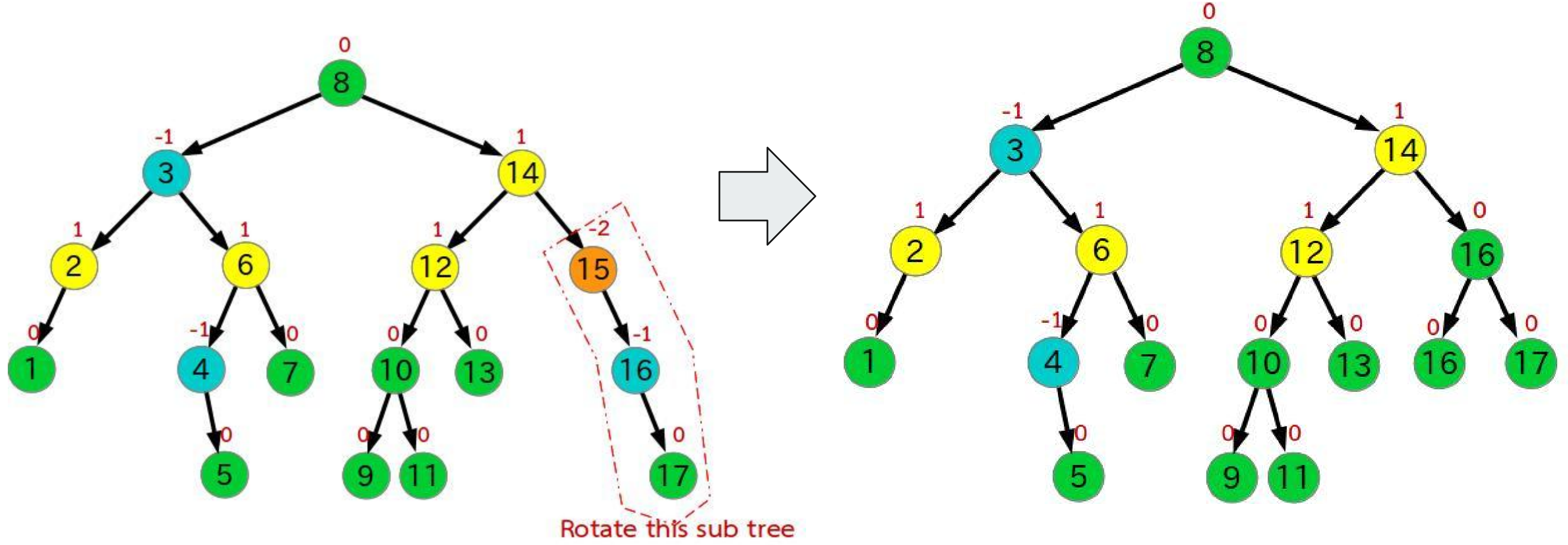
Example AVL Tree

- Insert "17"



Example AVL Tree

- Update Balance Factor and re-balance subtree.



Operations in AVL Tree

Insertion : augment the standard BST insert operation and re-balancing.

Deletion : augment the standard BST insert operation and re-balancing.

Performance of AVL Tree

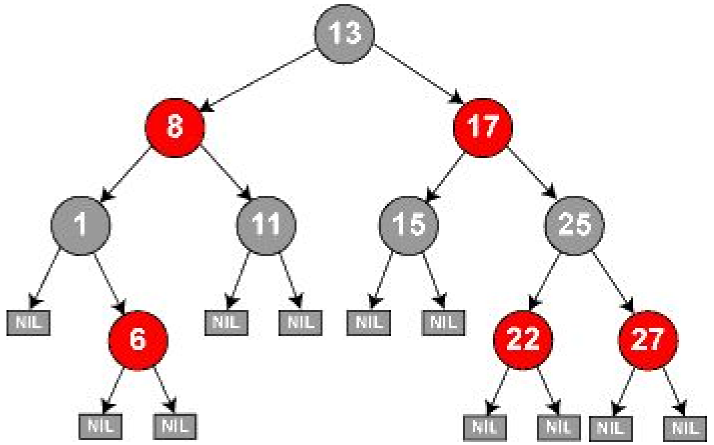
Space : O(N)

Searching : O(log N)

Insertion : O(log N)

Delete : O(log N)

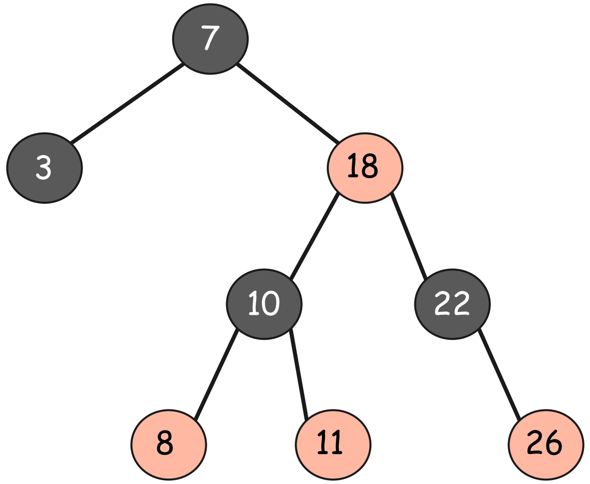
Red-black Tree



Red-Black Tree

A self-balancing Binary Search Tree where every node follows following rules.

1. Every node has a color either red or black.
2. Root of tree is always black.
3. There are no two adjacent red nodes.
4. Every path from root to a leaf node has same number of black nodes.



Red-Black Tree VS AVL Tree

The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion.

But, The performance (Big-O) are the same.

Properties of a Red-Black Tree

Black height : Is number of black nodes on a path from a node to a leaf. Leaf nodes are also counted black nodes.

Thus, a node of height h has black-height >= h/2.

Every Red-Black tree with N nodes has height <= 2 log(N + 1)

Balancing in Red-Black Tree

There are two tools to do balancing in Red-Black Tree :

- Recoloring
- Rotation (similar to AVL tree)

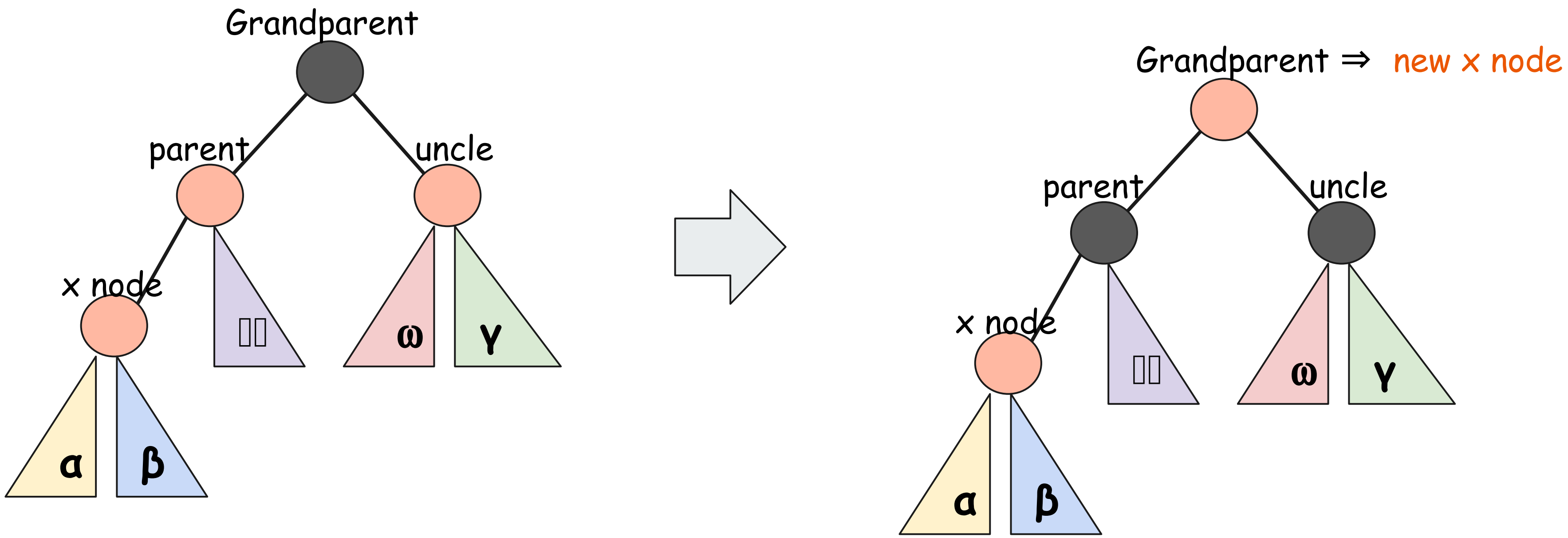
We try recoloring first, if recoloring doesn't work, then go for rotation. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Insertion in Red-Black Tree

Algorithm :

1. Perform BST insertion and make the color of newly inserted nodes (x) as RED.
2. If x is root, change color as BLACK.
3. If x in not root or x's parent is not BLACK do the following
    a. If x's uncle is RED
        i. Change color of parent and uncle as BLACK.
        ii. Color of grand parent as RED.
        iii. Change x = x's grandparent, repeat steps 2. and 3. for new x.

Recoloring

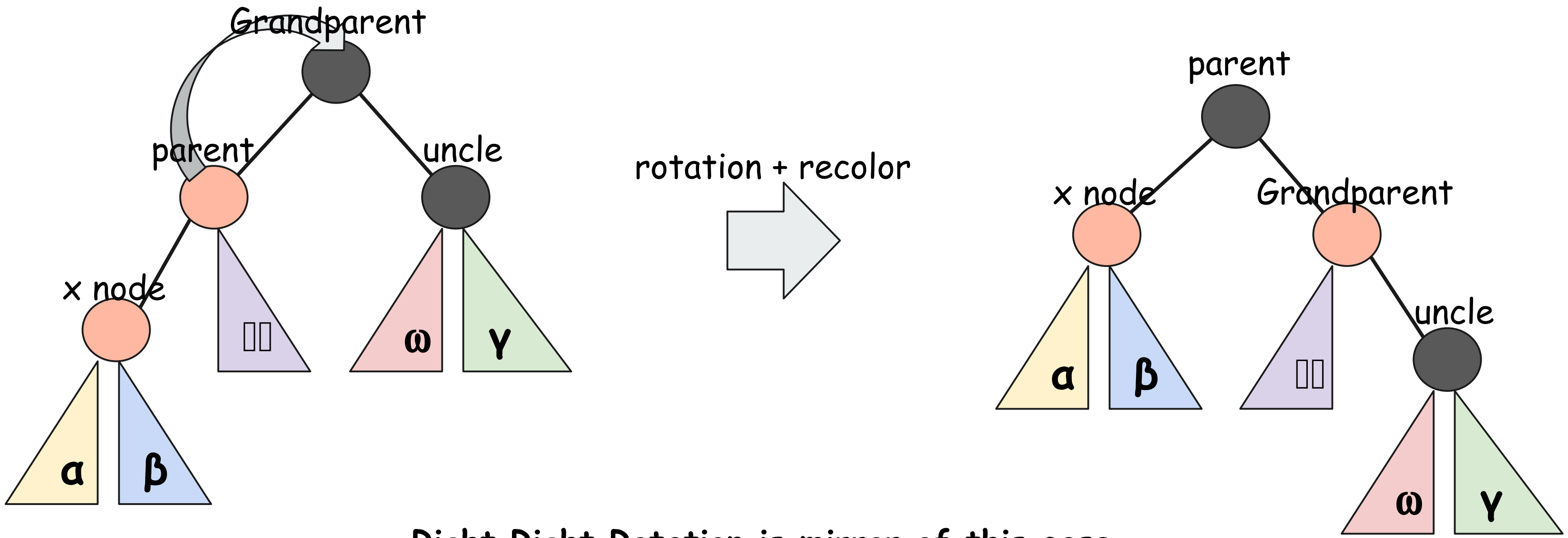

Grandparent
parent
uncle
x node
α
β
□□
ω
γ
Grandparent ⇒ new x node
parent
uncle
x node
α
β
□□
ω
γ

Insertion in Red-Black Tree

Algorithm :

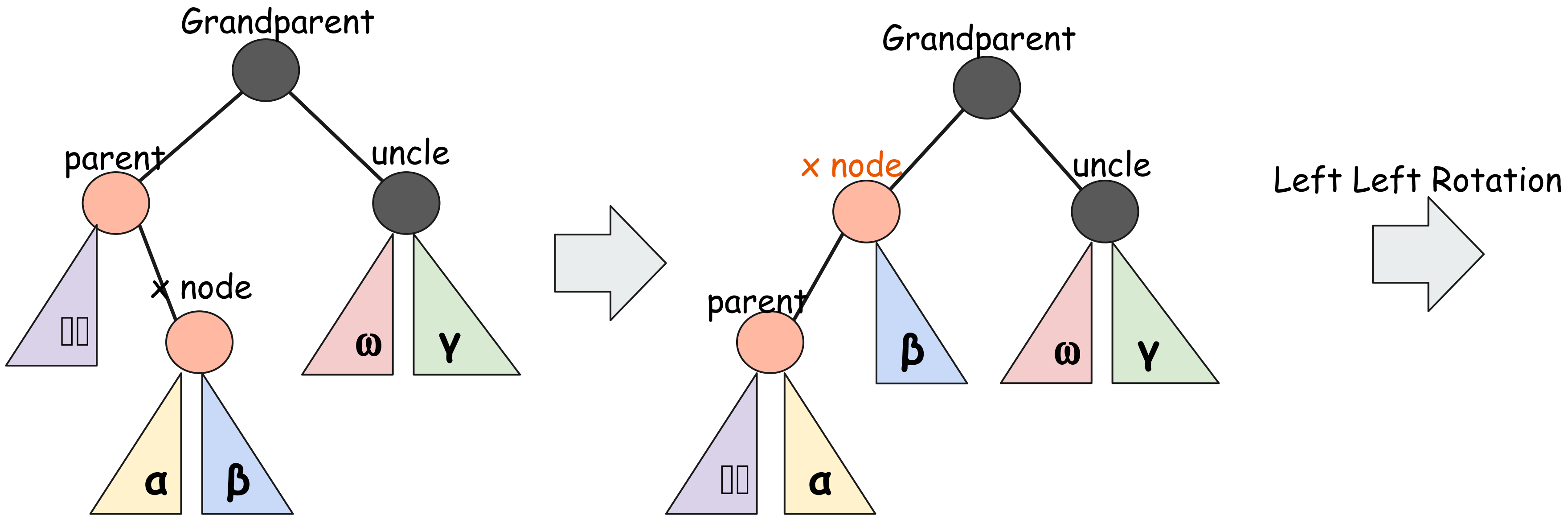
3. If x in not root or x's parent is not BLACK do the following
    b. If x's uncle is BLACK , there can be four rotate type for x, x's parent and x's grandparent
        i. Left-Left case
        ii. Right-Right
        iii. Left-Right
        iv. Right-Left

Left Left Rotation



Right Right Rotation is mirror of this case

Left Right Rotation



Right Left Rotation is mirror of this case

Deletion in Red-Black Tree

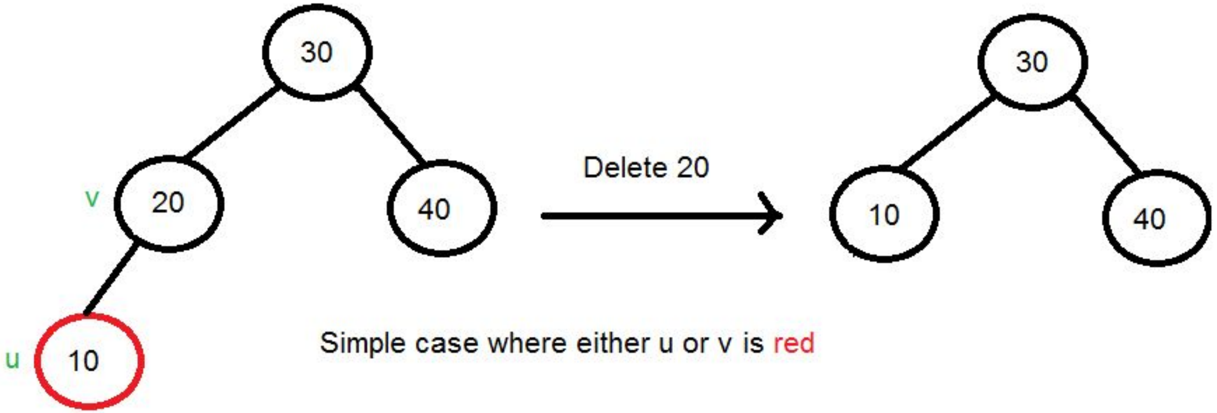
Algorithm :

1. Perform BST deletion and always end up deleting a node which is either leaf or has only one child
   a. For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child.
   b. Let ***v*** be the node to be deleted and ***u*** be the child that replaces ***v***.

Deletion in Red-Black Tree

Algorithm :

2. If either *u* or *v* is RED, we mark the replaced child as BLACK (No change in black height).

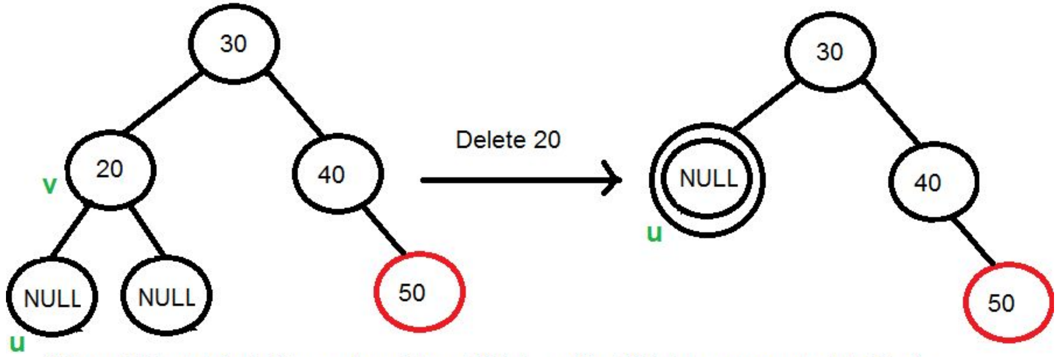


Simple case where either u or v is red

Deletion in Red-Black Tree

Algorithm :

3. If both ***u*** and ***v*** are BLACK.
   a. Color u as double black. Then convert this double black to single black.
   b. Do following while the current node ***u*** is double black or it is not root. Let sibling of node be ***s***.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

Deletion in Red-Black Tree

Algorithm :remove double black

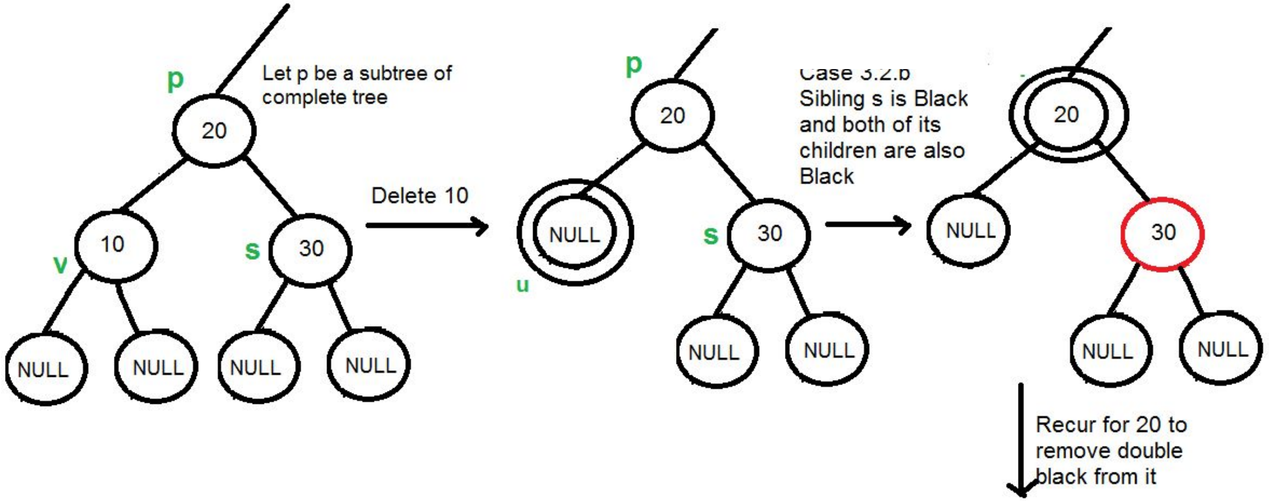
i. If sibling ***s*** is black and at least one of sibling's children is red, perform rotation(s). Let the red child of ***s*** be ***r***. This case can be divided in four subcases depending upon positions of ***s*** and ***r***.

1. Left Left Case (***s*** is left child of its parent and ***r*** is left child of ***s*** or both children of ***s*** are red).
2. Left Right Case (***s*** is left child of its parent and ***r*** is right child).
3. Right Right Case (***s*** is right child of its parent and ***r*** is right child of ***s*** or both children of ***s*** are red)
4. Right Left Case (s is right child of its parent and r is left child of s)

Deletion in Red-Black Tree

Algorithm : remove double black

ii. If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.



p
Let p be a subtree of complete tree
20
10
v
s
30
NULL
NULL
NULL
NULL
Delete 10
p
20
NULL
u
s
30
NULL
NULL
Case 3.2.b
Sibling s is Black and both of its children are also Black
20
NULL
30
NULL
NULL
Recur for 20 to remove double black from it

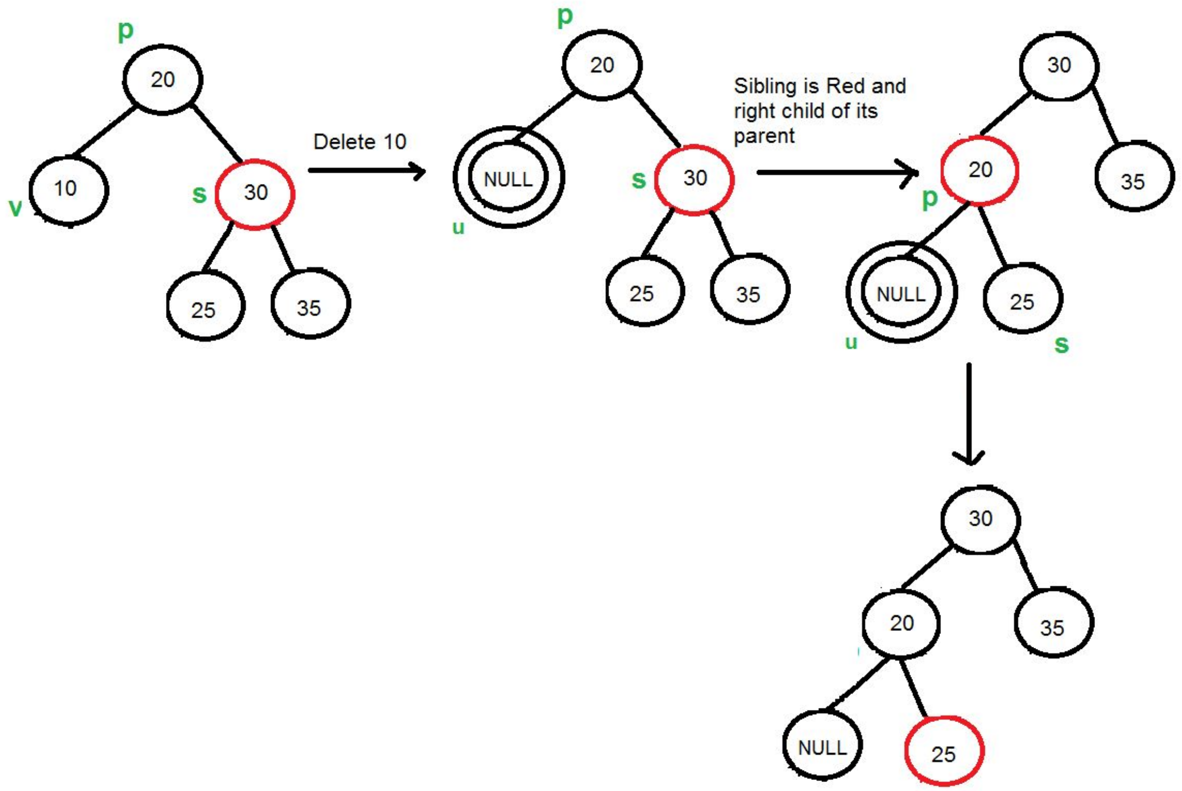
Deletion in Red-Black Tree

Algorithm : remove double black

iii. If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black. This mainly converts the tree to black sibling case (by rotation) and leads to case (i) or (ii). This case can be divided in two subcases.
   1. Left Case (s is left child of its parent). We right rotate the parent p.
   2. Right Case (s is right child of its parent). We left rotate the parent p.

Deletion in Red-Black Tree

Algorithm :



Performance of Red-Black Tree

Space : O(N)

Searching : O(log N)

Insertion : O(log N)

Delete : O(log N)

Red Black Tree for implementations

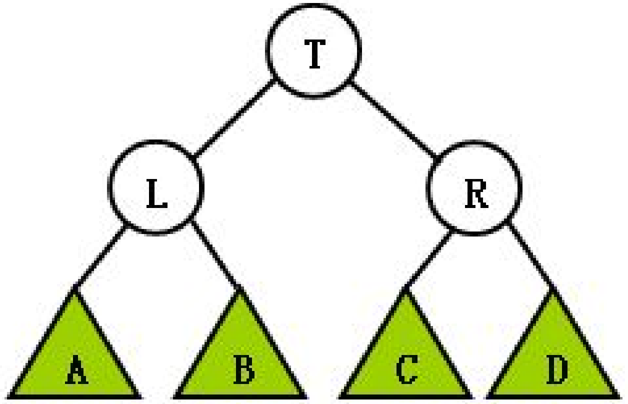
split and join for red black tree

https://goo.gl/R9v2xX

Left-Learning Red Black Tree

http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf

Size Balanced Tree



Size Balanced Tree

A size balanced tree (SBT) is a self-balancing binary search tree (BBST). The tree is rebalanced by examining the sizes of each node's subtrees.

Very convenient to implement the select-by-rank and get-rank operations.

Operations in Size Balanced Tree

Insert(v) : Inserts a node whose key is v into the SBT.

Delete(v) : Deletes a node whose key is v from the SBT.

Find(v) : Finds the node whose key is v and return it.

Operations in Size Balanced Tree

Rank(*v*) : Return the rank of v in the tree (order of the number).

Select(*k*) : Return the node which is ranked at the *k*th position.

Pred(*v*) : Return the node with maximum key which is less than v.

Succ(*k*) : Return the node with minimum key which is larger than v.

performance of all opeations are **O(log N)**

Property of Size Balanced Tree

For every node pointed by t in a SBT, guarantee that:

size(t->right) ≥ size(t->left->left), size(t->left->right)

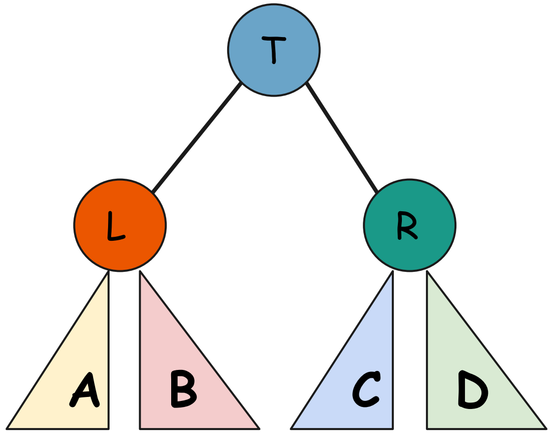
and

size(t->left) ≥ size(t->right->right), size(t->right->left)

Property of Size Balanced Tree

size(t->right) ≥ size(t->left->left), size(t->left->right)

size(t->left) ≥ size(t->right->right), size(t->right->left)

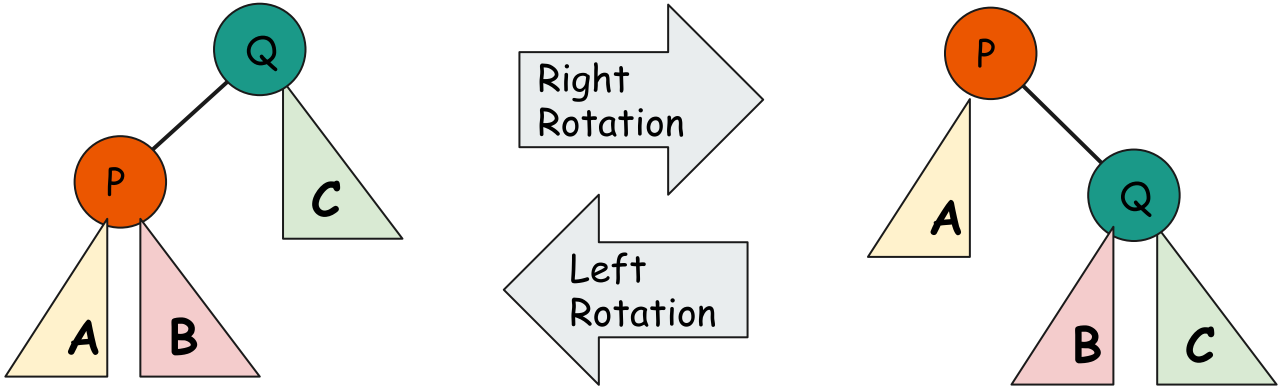


size(R) ≥ size(A), size(B)
size(L) ≥ size(C),size(D)

Rotation in Size Balanced Tree

Two type of rotations in SBT:

1. Right Rotation.
2. Left Rotation.



Rotation in Size Balanced Tree

Psuedocode: Left rotation

```
def left-rotate(t):
    k ← t.right
    t.right ← k.left
    k.left ← t
    k.size ← t.size
    t.size ← t.left.size + t.right.size + 1
    t ← k
```

Psuedocode: Right rotation

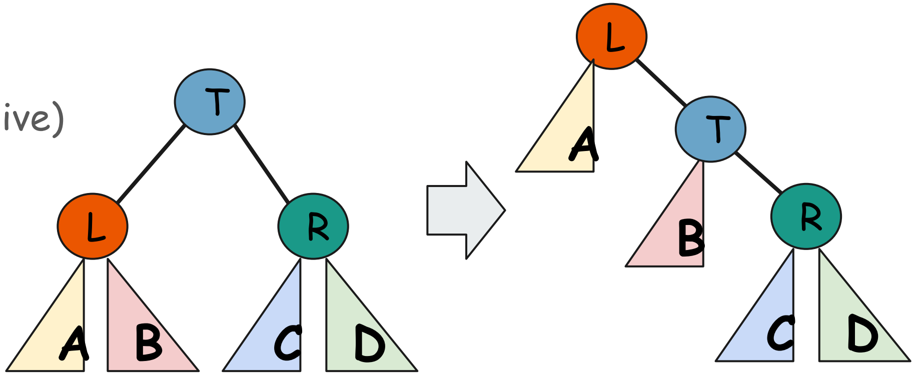
```
def right-rotate(t):
    k ← t.left
    t.left ← k.right
    k.right ← t
    k.size ← t.size
    t.size ← t.left.size + t.right.size + 1
    t ← k
```

Maintain in Size Balanced Tree

After performing a simple insertion, we need to adjust the SBT.

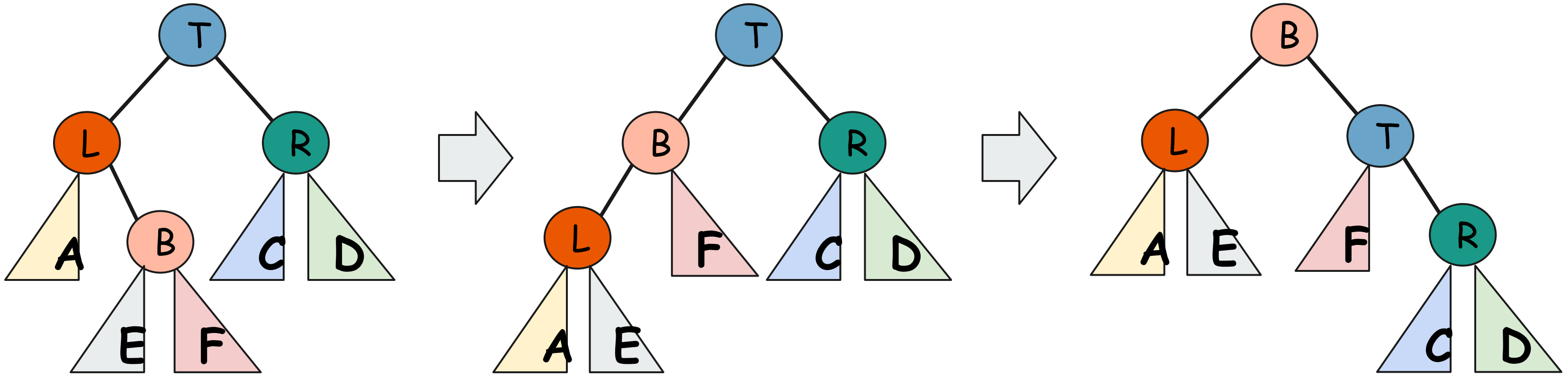
provide in four cases :

1. Case size(t->left->left) > size(t->right)
    a. perform right rotate at 'T'
    b. perform maintain again 'T' (recursive)
    c. perform maintain at 'L'



Maintain in Size Balanced Tree

2. Case size(t->left->right) > size(t->right)
   a. perform left rotate at 'L'
   b. perform right rotate at 'T'
   c. perform maintain 'L' and 'T'
   d. perform maintain at B'



Maintain in Size Balanced Tree

3. Case size(t->right>right) > size(t->left)
    a. symmetrical with case 1.

4. Case size(t->right->left) > size(t->left)
    a. symmetrical with case 2.

Maintain in Size Balanced Tree

Pseudocode of Maintain

amortized running time is O(1)

```
def maintain(t, flag):
    if flag:
        if t.left.size < t.right.left.size:        //case 1
            right-rotate(t.right)
            left-rotate(t)
        else if t.left.size < t.right.right.size:  //case 2
            left-rotate(t)
        else:
            done
    else:
        if t.right.size < t.left.right.size:       //case 3
            left-rotate(t.left)
            right-rotate(t)
        else if t.right.size < t.left.left.size:   //case 4
            right-rotate(t)
        else:
            done

    maintain(t.left, FALSE)   //maintain the left subtree
    maintain(t.right, TRUE)   //maintain the right subtree
    maintain(t, TRUE)         //maintain the whole tree
    maintain(t, FALSE)        //maintain the whole tree
```

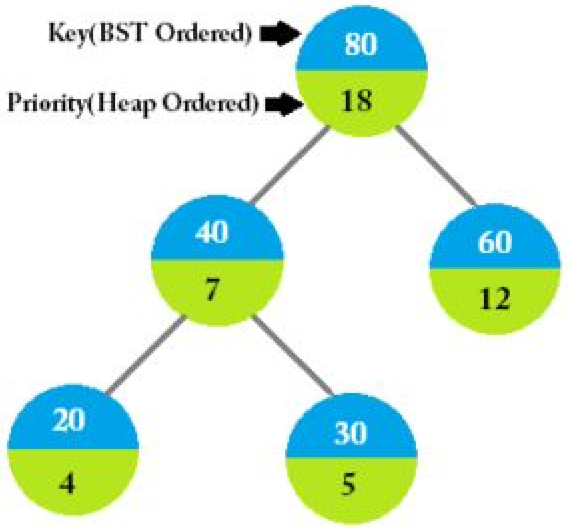
Select in Size Balanced Tree

Pseudocode (select *i* th position data)

```
def select(t, i):
    r ← t.left.size
    if i = r:
        return t
    else if i < r:
        return select(t.left, i)
    else
        return select(t.right, i - (r + 1))
```

for further reading : https://goo.gl/ia65bL

Treap



Treap

Treap is a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n).

The idea is to use Randomization and Binary Heap property to maintain balance with high probability.

The expected time complexity of search, insert and delete is O(Log n).

Treap

Every node of Treap maintains two values.

1. Key Follows standard BST ordering (left is smaller and right is greater)
2. Priority Randomly assigned value that follows Max-Heap property.

Treap

priorities allow to uniquely specify the tree that will be constructed, which can be proven using corresponding theorem.

Obviously, if you choose the priorities randomly, you will get non-degenerate trees on average, which will ensure O(logN) complexity for the main operations.

Operations of Treap

Search(x) : Perform standard BST Search to find x.

Insert(x) : Insert node x to treap

1. Create new node with key equals to x and value equals to a random value.
2. Perform standard BST insert.
3. Use **rotations** to make sure that inserted node's priority follows max heap property.

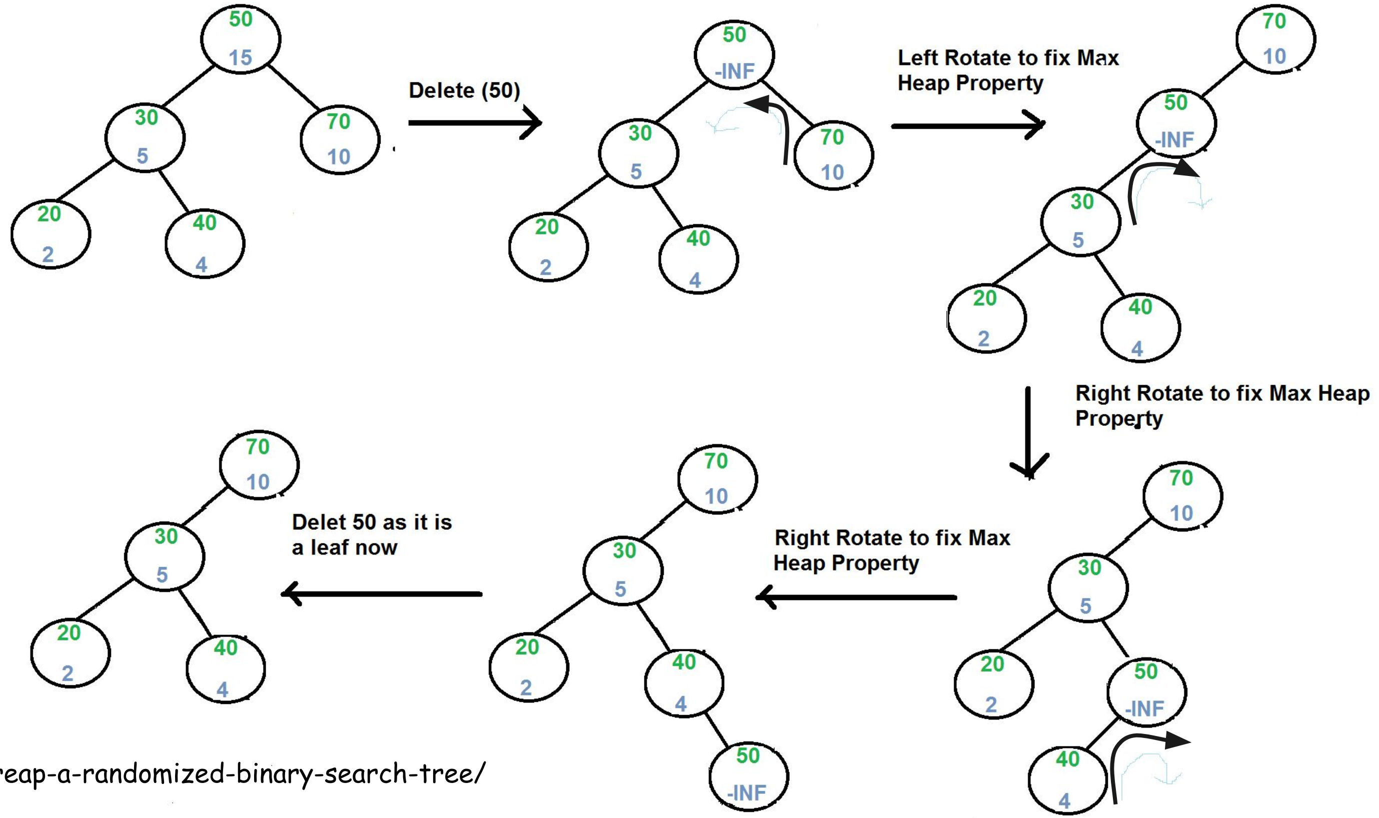
Operations of Treap

Delete(x) : Delete node x from treap.

1. If node to be deleted is a leaf, delete it.
2. Else replace node's priority with minus infinite (-INF), and do appropriate rotations to bring the node down to a leaf.

Operations of Treap

Delete(50) :



50 15
30 5
70 10
20 2
40 4
Delete (50)
50 -INF
Left Rotate to fix Max Heap Property
Right Rotate to fix Max Heap Property
Right Rotate to fix Max Heap Property
Delet 50 as it is a leaf now

http://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/

Split and Merge in Treap

The rotation of tree feels messy!!

Treap supports two very powerful operations : split and merge , both in O(H) where H is height of treap.

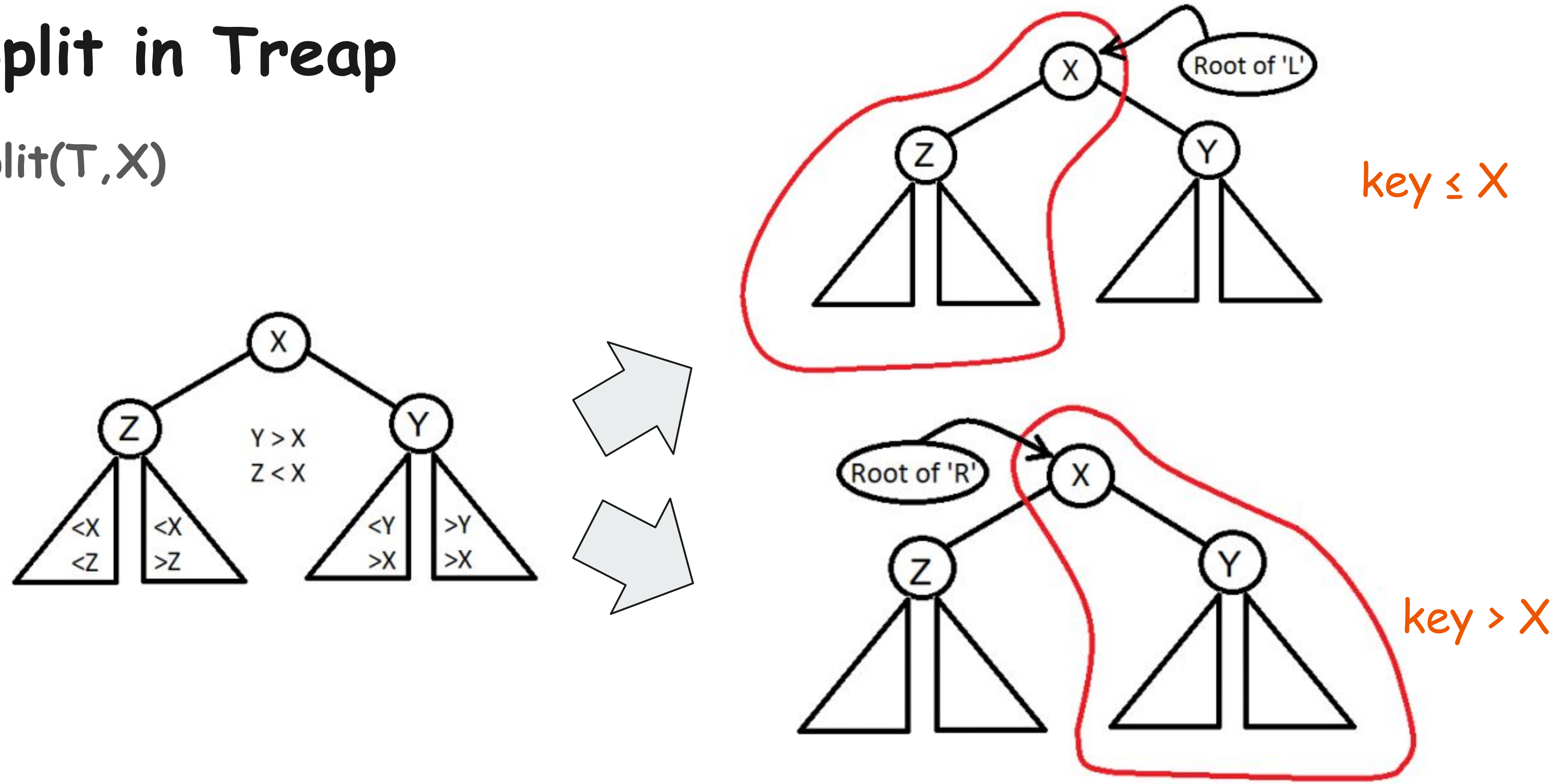
We can perform insert, delete by split and merge with out tree rotations !!.

Split in Treap

split(T,X) : It splits a given treap T into two different treaps L and R such that L contains all the nodes with key ≤ X and R contains all the nodes with key > X . The original treap T is destroyed after the split operation.

Split in Treap

split(T,X)



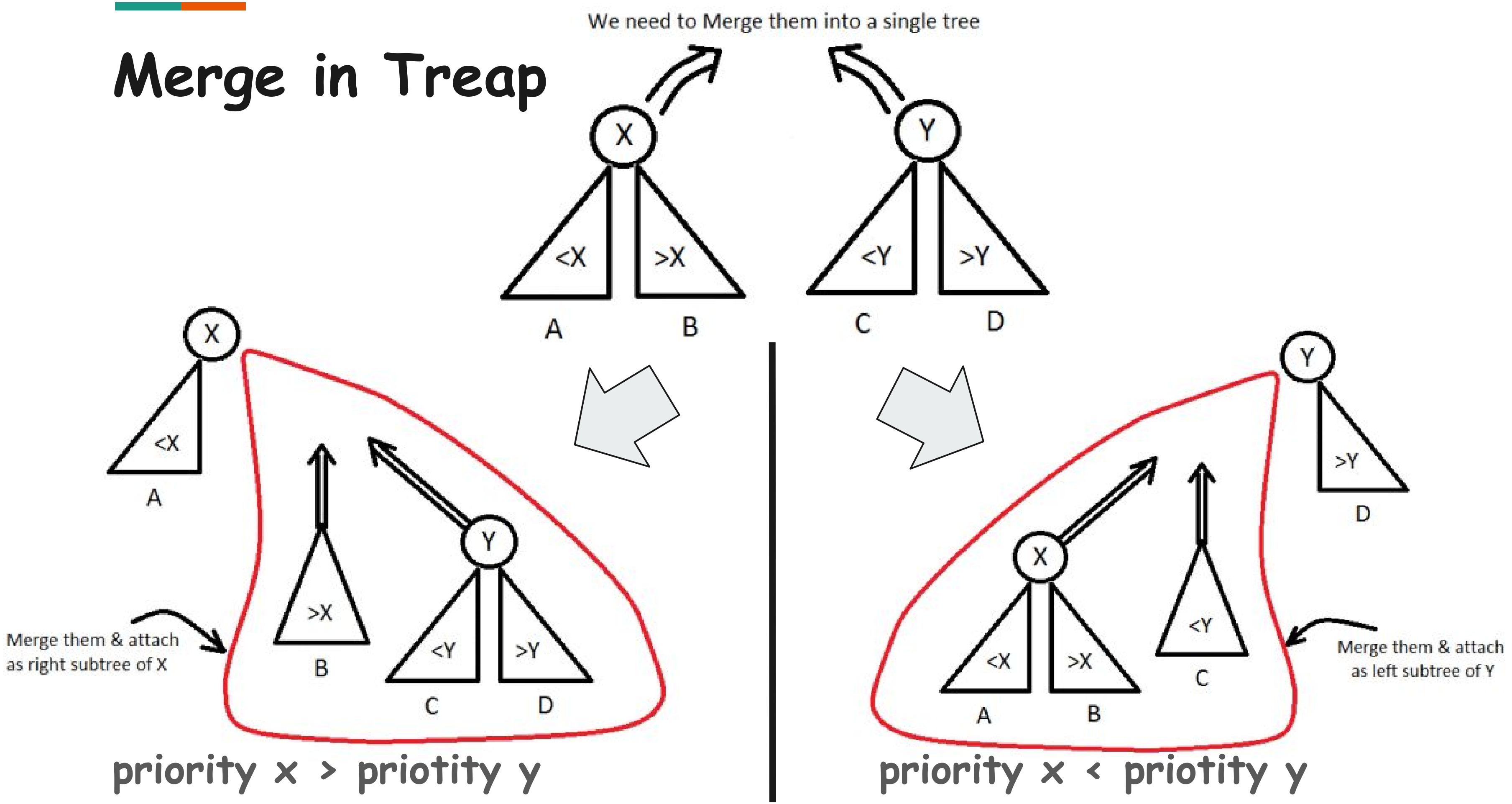
https://tanujkhattar.wordpress.com/2016/01/10/treaps-one-tree-to-rule-em-all-part-1/

Merge in Treap

merge(L,R) : The merge operation merges two given treaps L and R into a single treap T. L and R are destroyed after the operation.

A very important assumption of the merge operation is that the largest value of L is less than the smallest value of R. Hence two treaps obtained after a split operation can always be merged to give back the original treap.

Merge in Treap


We need to Merge them into a single tree
X
Y
<X
>X
<Y
>Y
A
B
C
D
Merge them & attach as right subtree of X
Merge them & attach as left subtree of Y
priority x > priotity y
priority x < priotity y

Operations with merge and split

Insert(X) :

1. Create new node X with random priority Y (call node (X, Y)).
2. find the position to insert X with simple BST searching.
3. If the correct position is found or encounter the first node E such its priority < Y, **split(E, X)** and attach L and R as left and right subtrees of node (X, Y)

Operations with merge and split

Delete(X) :

1. Go down the tree like a BST unless node to be deleted is found.
2. **merge** function for it's left and right subtrees and attach the resulting tree to the parent of the node to be deleted.

Implicit Treap

Implicit treap can be viewed as a dynamic array which get all the operations supported by a segment tree along with **the power to split an array into two parts and merge two different arrays into a single one in O(log N)**.

Implicit Treap

The key idea in implicit treap is, we use the array indices of elements as keys , instead of the values .

However

Since we are using the array index as the key of the BST this time, with each update (insertion / deletion) we will have to change O(n) values.

Implicit Treap

To avoid this, we will **not explicitly** store the index i at each node in the implicit treap and calculate this value on the fly.

⇒ Still using pointer for implementation!

The key value for any node x would be 1 + no of nodes in the BST that have key values less than x .

Implicit Treap

More specifically,

the implicit key for some node T is the number of vertices from node T to Leaf in the left subtree of this node plus similar values from node P to Leaf +1 for each ancestor P of the node T, if T is in the right subtree of P.

Split and Merge in Implicit Treap

split : Splits the array A[1..n] into two parts : L[1..pos] , R[pos+1..n] , about “pos”.

merge : merges L[1..n1] , R[1..n2] to form A[1..n1,n1+1,...n2] . Note that the condition for merge in treap (i.e. greatest element in l <= smallest element in r) is satisified here.

Insert and Delete in Implicit Treap

insert(x,i) : insert x at position "i"

1. split the treap about pos = i-1 such that we get two treaps , L[1..i-1] and R[i .. n].
2. Merge L and x , and then merge the resulting treap with R.

Insert and Delete in Implicit Treap

delete(i) : delete A[i] from the array

1. Split the treap about pos = i -1 such that we get two treaps L[1...i-1] and R[i...n].
2. Split R about pos = i such we get L' [i] and R'[i+1...n].
3. Merge L[1..i-1] and R'[i+1..n].

Misc.

Binary tree Problem

http://www.spoj.com/problems/HORRIBLE/

http://www.spoj.com/problems/GSS6/

http://www.spoj.com/problems/QMAX3VN/en/

https://www.codechef.com/problems/CARDSHUF/

Treap and Implicit Treap code sample

https://e-maxx-eng.appspot.com/data_structures/treap.html

https://www.cs.cmu.edu/~scandal/papers/treaps-spaa98.pdf