Splay tree

- Self adjusting binary search tree
- any tree works great if it is balanced (search, insert, delete) like red black , avl , btree ..
- They used to maintain balance info
- In splay tree, no state info is maintained but performs the same way or better.
- it is lazy , means no bother to balance beforehand
- they balance when any operation performed

- **why we use it :**

1. **path shortening**
2. **rebalancing**

- **if the search is long (if path is long), shorten the path**
- case : shorten one path can make other path long (we need to take care of that)
- why search is fast in balanced bst : relative size of two subtree is " close" .
- search is still slow due to the fat node situated low in the tree, so it promotes the this fat node to up in the tree usins rotations.

An Initial Idea

- Begin with an arbitrary BST.
- After looking up an element, repeatedly rotate that element with its parent until it becomes the root.
  - Intuition:
  - Recently-accessed elements will be up near the root of the tree, lowering access time.
  - Unused elements stay low in the tree.

The Problem

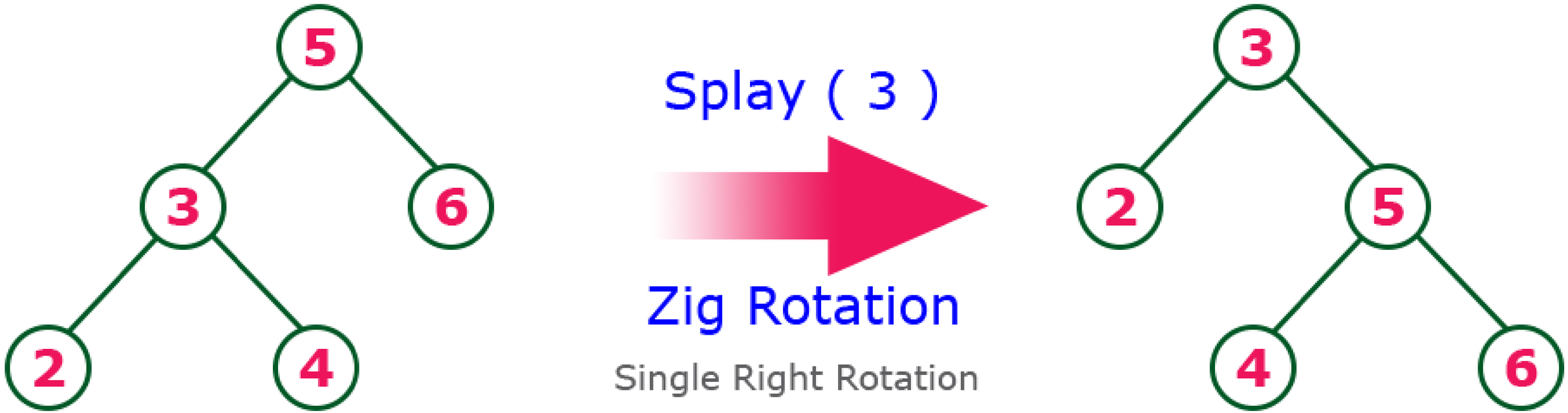
- The “rotate to root” method might result in n accesses taking time Θ(n2).
- Why?
- Rotating an element x to the root significantly “helps” x, but “hurts” the rest of the tree.
- Most of the nodes on the access path to x have depth that increases or is unchanged.

A More BalancedApproach

- Rotates an element to the root of the tree, but does so in a way that's more “fair”to other nodes in the tree.
- There are three cases for splaying.

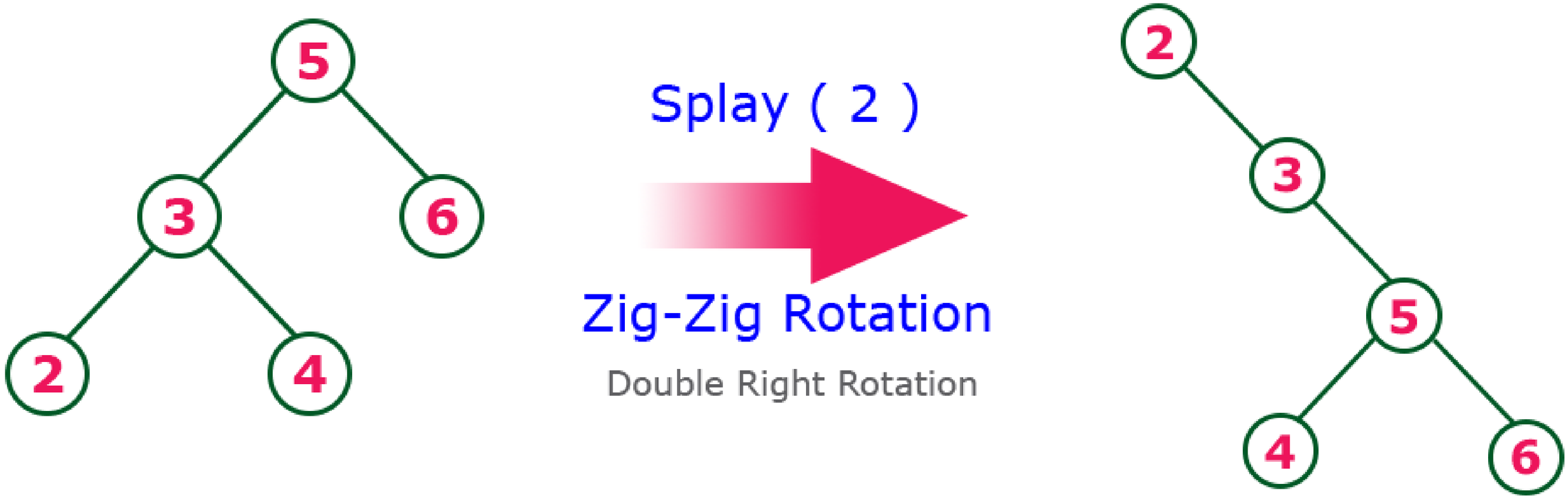
```
while n.parent != NULL //node is not root

if n.parent == T.root //node is child of root, one rotation
if n == n.parent.left //left child
RIGHT_ROTATE(T, n.parent)
else //right child
LEFT_ROTATE(T, n.parent)
else //two rotations
```

Case 1: Zig



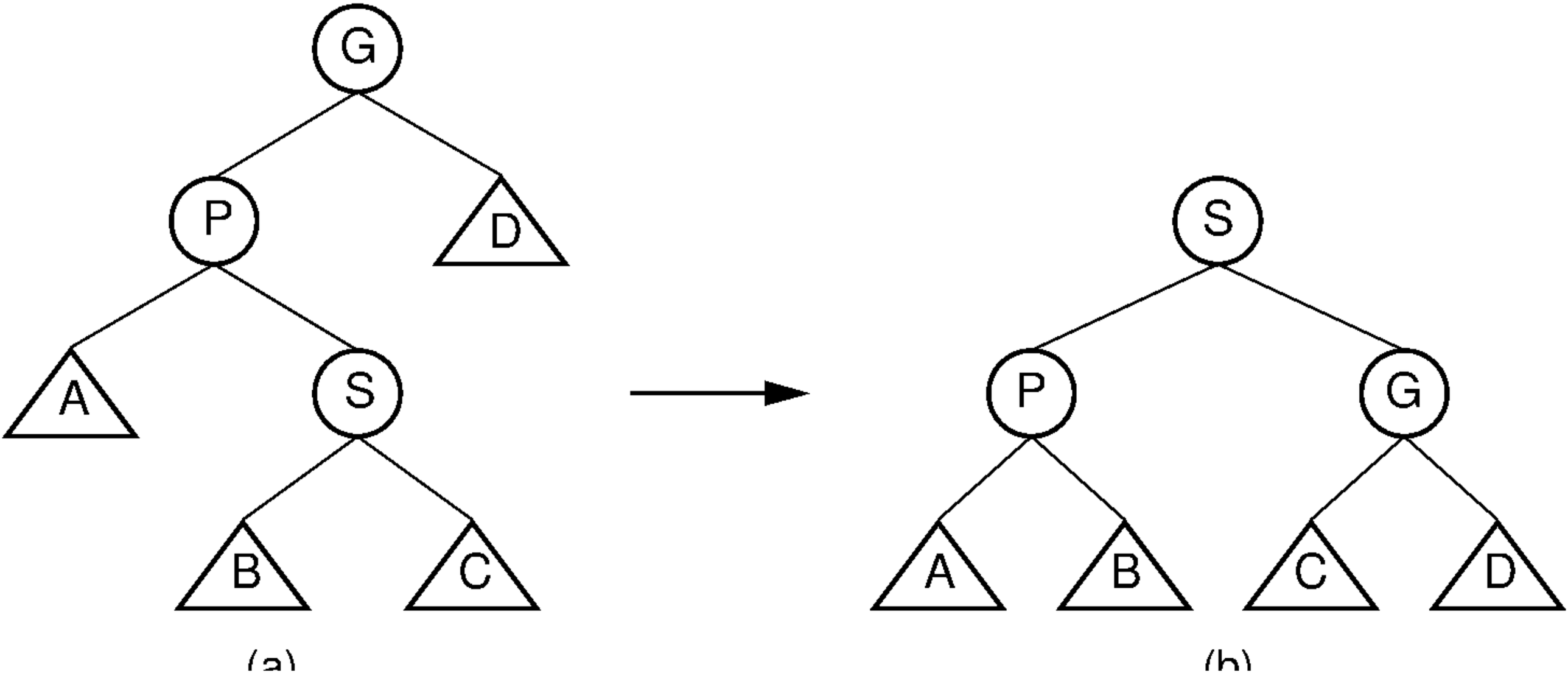
Zag Rotation

Zig-Zig Rotation



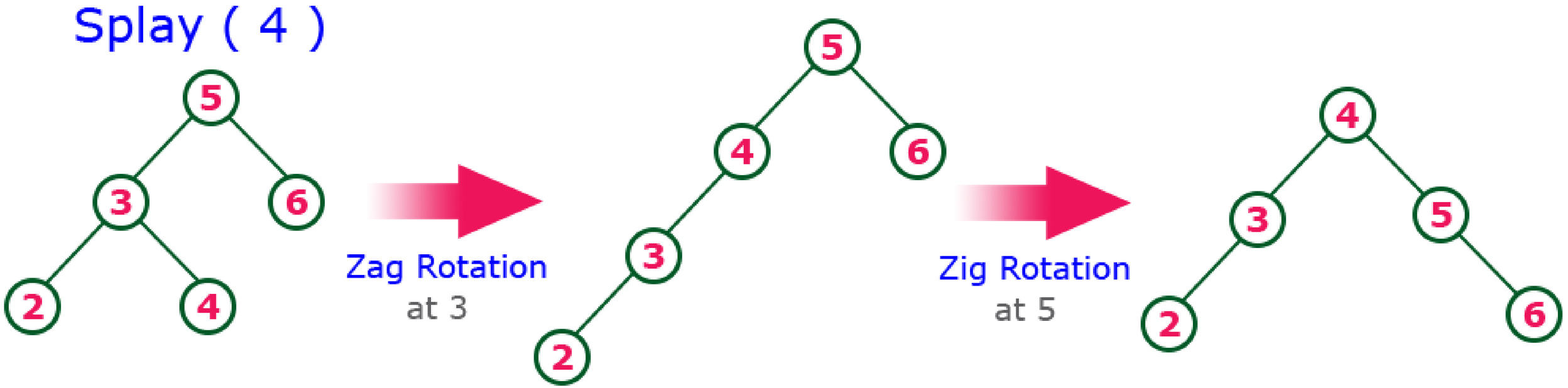
Zag-Zag Rotation

Zig-Zag Rotation



(a) (b)

Zig-Zag Rotation



Zag - zig rotation

Types of Splaying

1. **Bottom up splay**
2. **Top down splay**

B0ttom up Splaying

We will start by passing the tree (T) and the node which is going to be splayed (n).

SPLAY(T, n)

We have to splay the node n to the root. So, we will use a loop and perform suitable rotations and stop it when the node n reaches to the root.

SPALY(T, n)

SPALY(T, n)

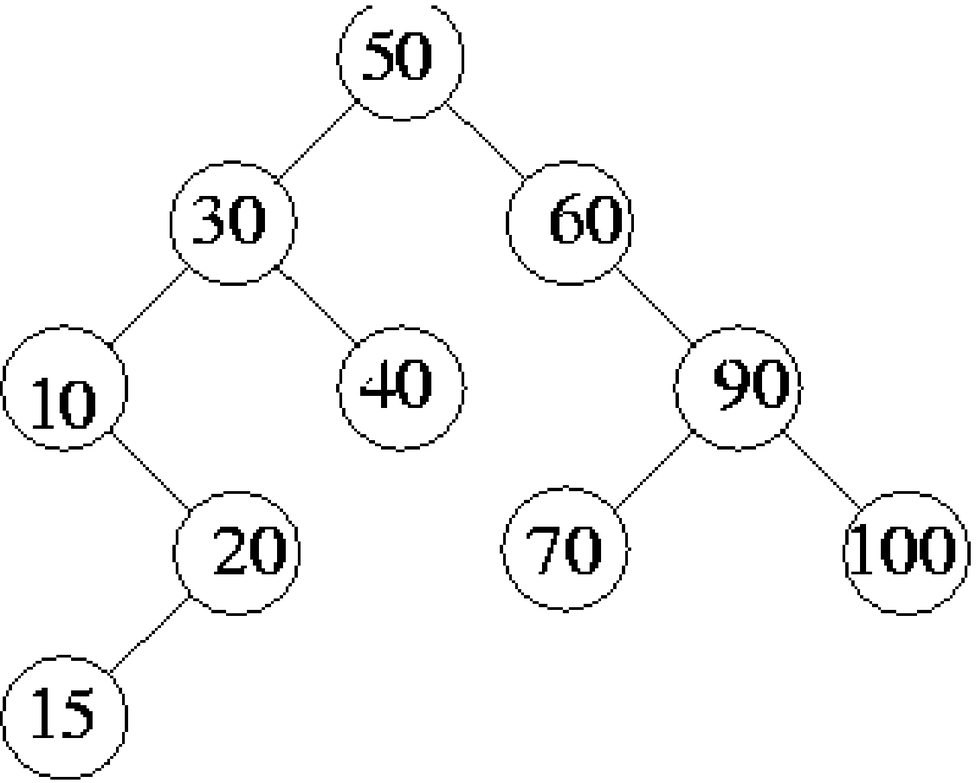
while n.parent != NULL //node is not root

if n.parent == T.root //node is child of root, one rotation
if n == n.parent.left //left child
RIGHT_ROTATE(T, n.parent)
else //right child
LEFT_ROTATE(T, n.parent)
else //two rotations

Search (i, t)/ Same as BST

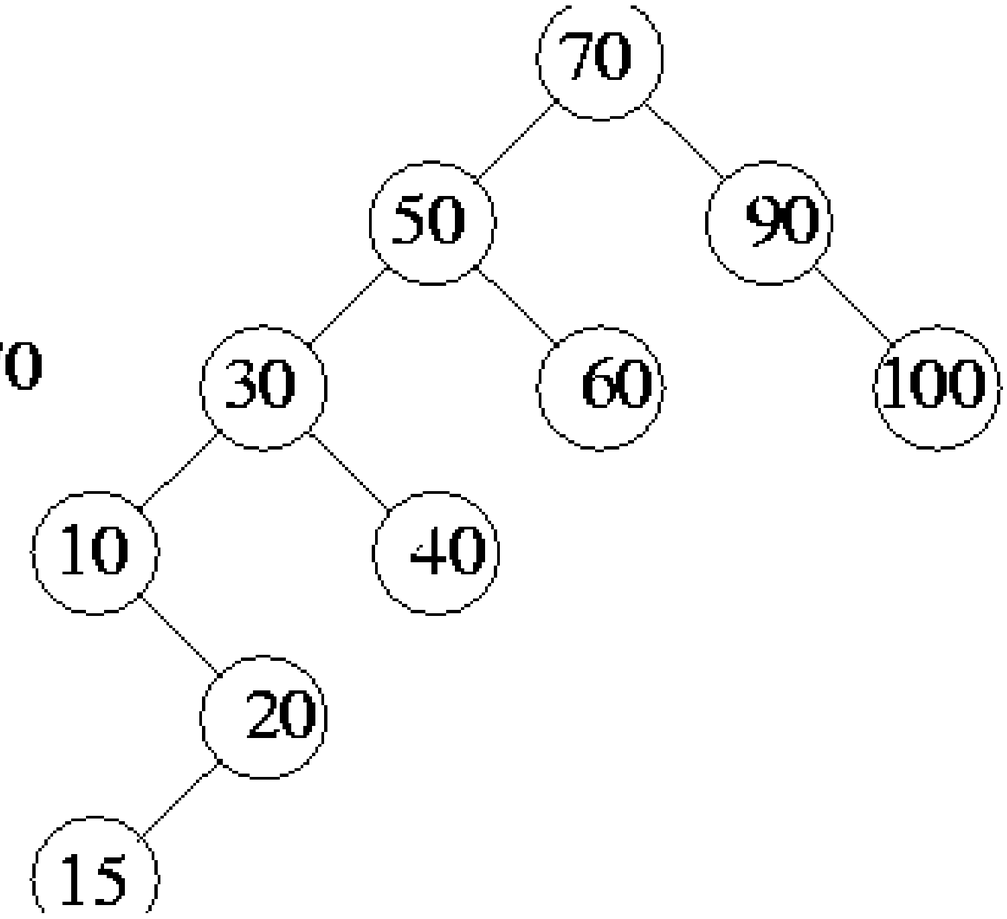
If item i is in tree t, return a pointer to the node containing i; otherwise return a pointer to the null node.

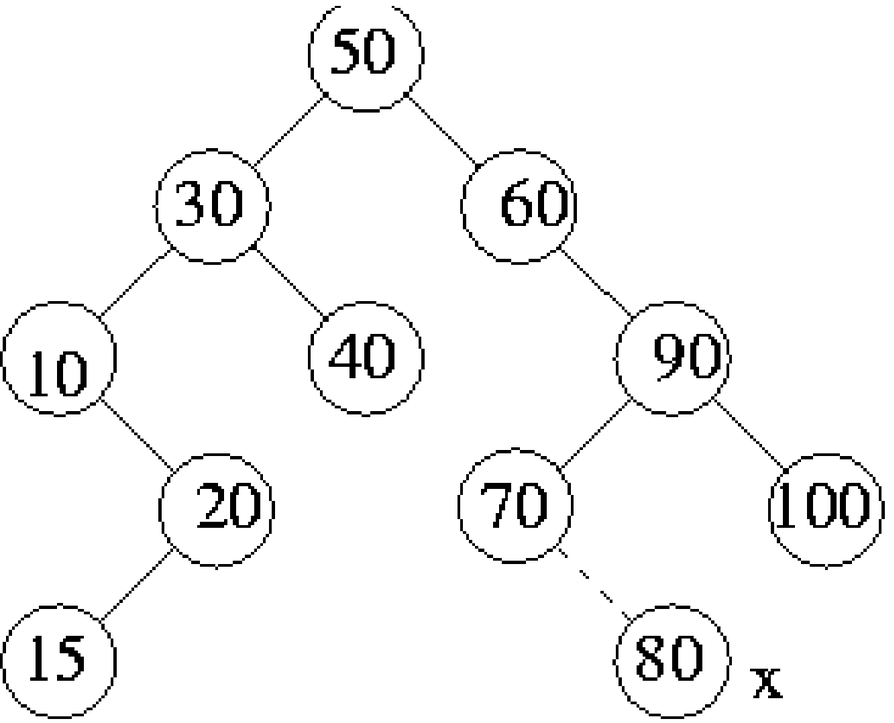
- Search down the root of t, looking for i
- If the search is successful and we reach a node x containing i, we complete the search by splaying at x and returning a pointer to x
- If the search is unsuccessful, i.e., we reach the null node, we splay at the last non-null node reached during the search and return a pointer to null.
- If the tree is empty, we omit any splaying operation.



Search (80,t)

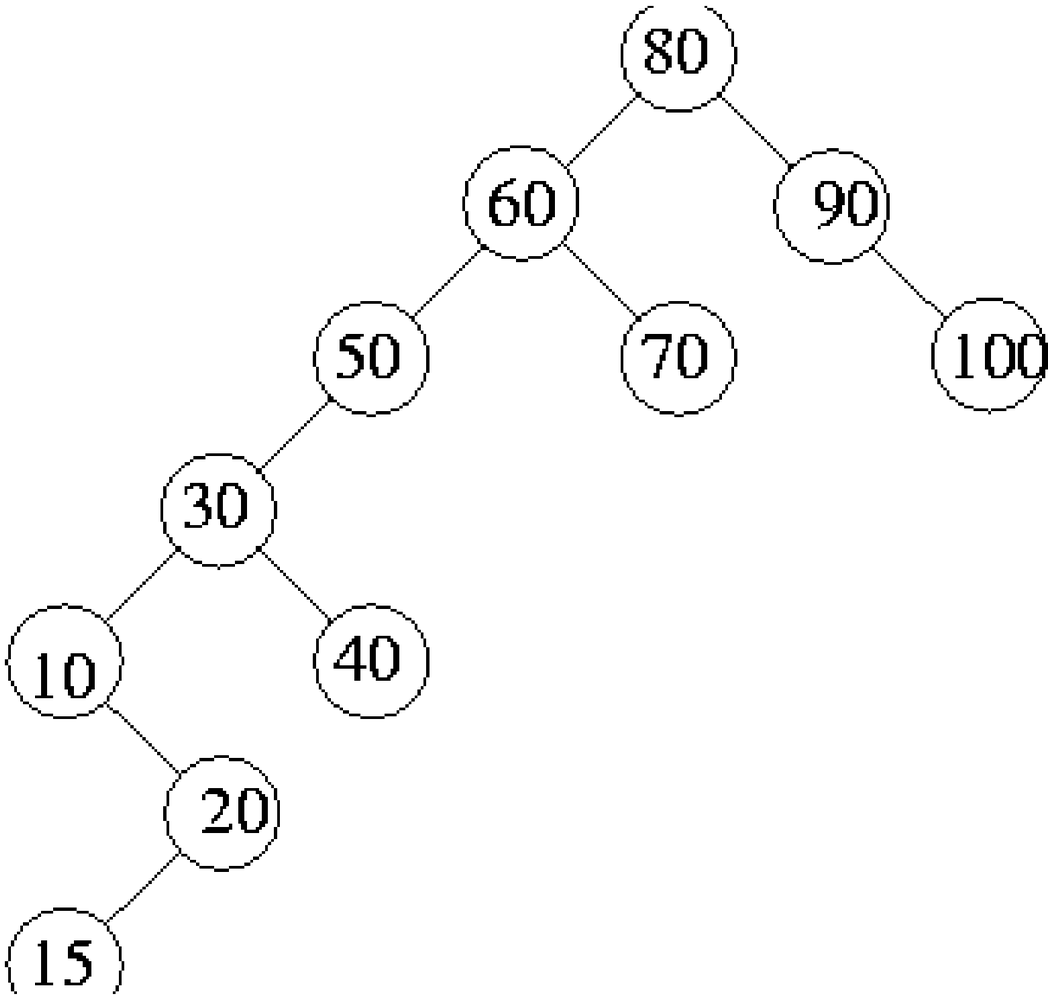
leads to splay at 70

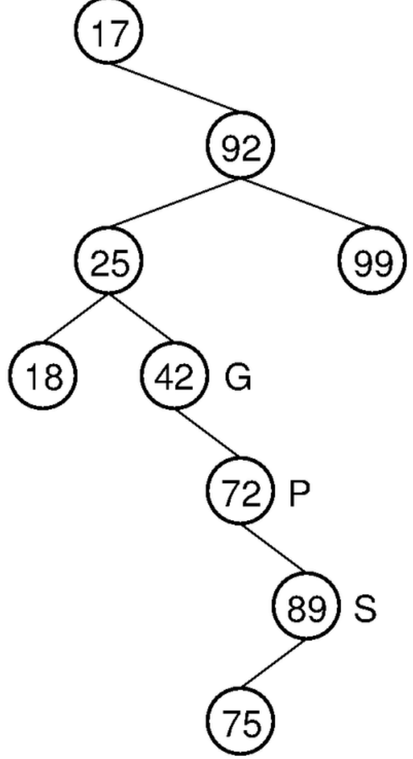




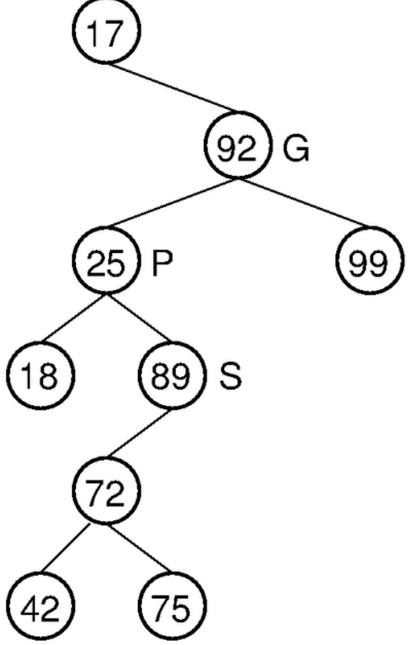
Insert (80, t)

Splay at x

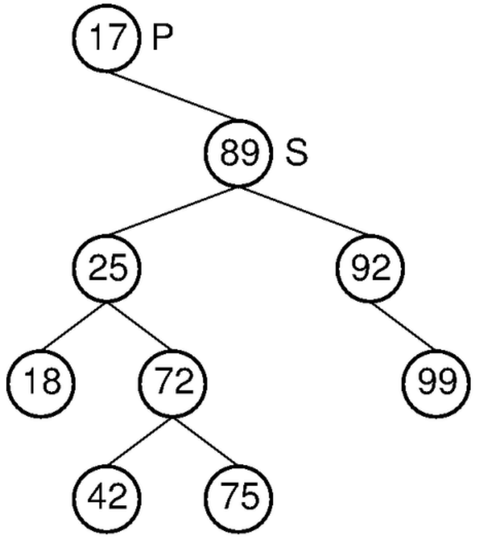


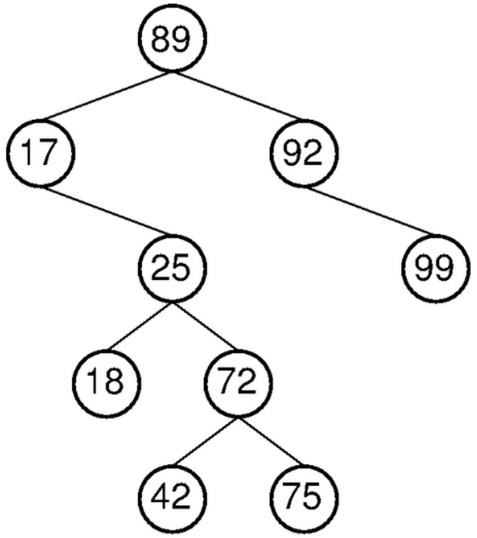

17
92
25
99
18
42
G
72
P
89
S
75

(a)


17
92
G
25
P
99
18
89
S
72
42
75

(b)


17
P
89
S
25
92
18
72
99
42
75


89
17
92
25
99
18
72
42
75

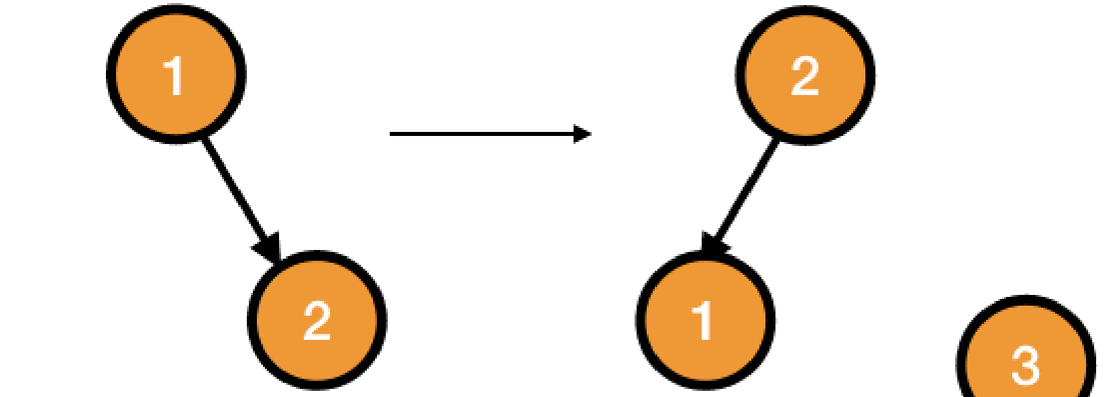
Insertion Operation in Splay Tree

- **Step 1 - Check whether tree is Empty.**
- **Step 2 - If tree is Empty then insert the new Node as Root node and exit from the operation.**
- **Step 3 - If tree is not Empty then insert the new Node as leaf node using Binary Search tree insertion logic.**
- **Step 4 - After insertion, Splay the new Node**

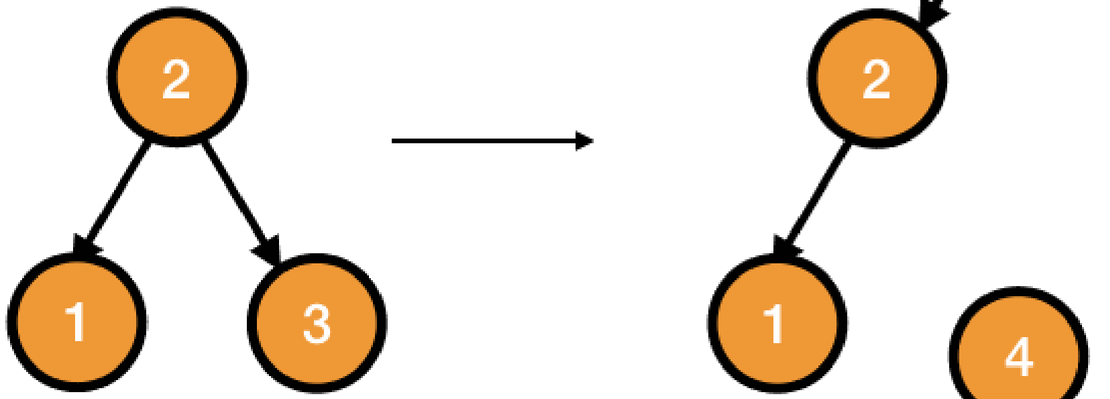
Bottom up insertion

Insert(T, n)

- temp = T.root
- while temp != Null.
  - y = temp
  - if n → data < temp → data.
    - temp = temp → left.
  - else temp = temp → right.
- n.Parent = y.
- if y == Null T.root = n.
- else if n → data < y → data
  - y → left = n
- else. y → right = n.



Insert 2



Insert 3

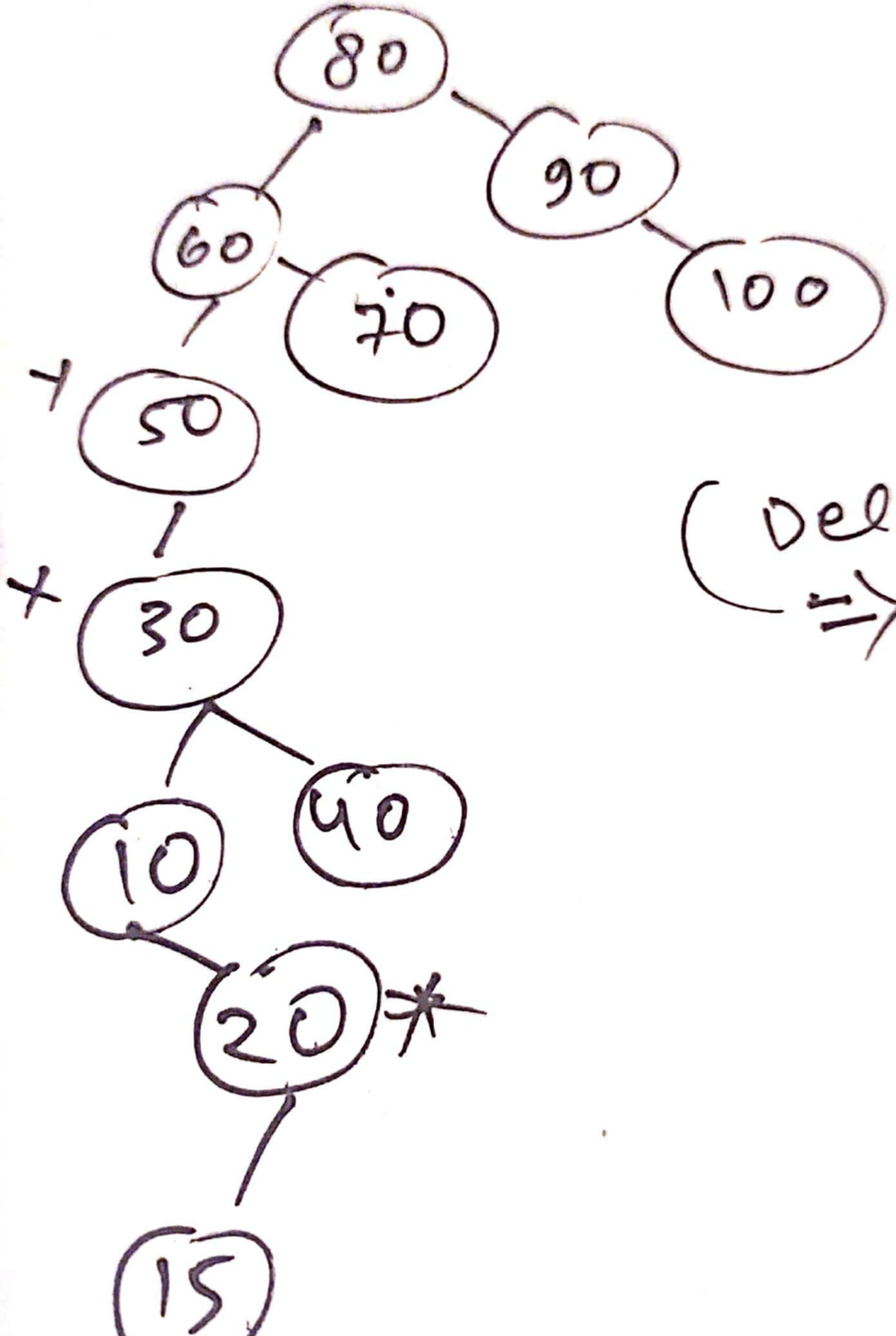
INSERT(T, n)

```
temp = T.root
y = NULL
while temp != NULL
    y = temp
    if n.data < temp.data
        temp = temp.left
    else
        temp = temp.right
n.parent = y
if y==NULL
    T.root = n
else if n.data < y.data
    y.left = n
else
    y.right = n
```

SPLAY(T, n)

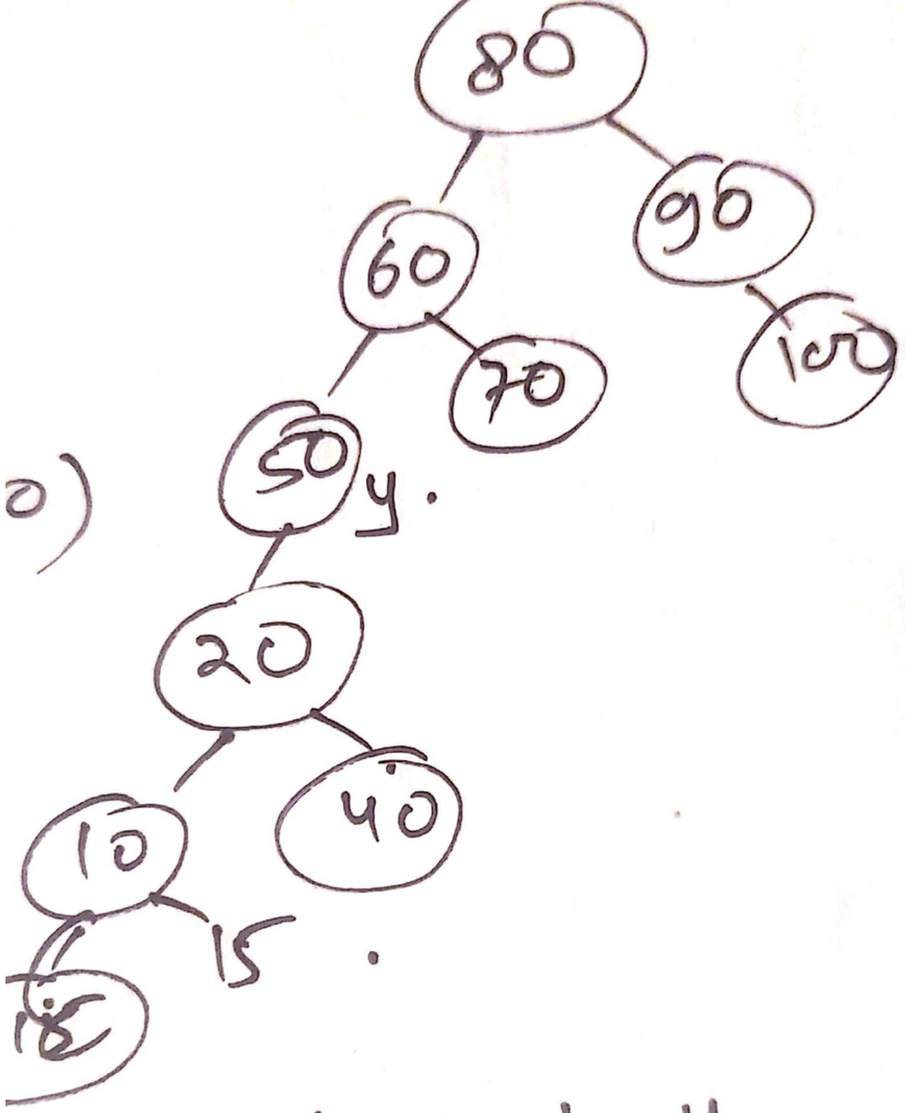
Delete (i, t)

- Search for i. If the search is unsuccessful, splay at the last non-null node encountered during search.
- If the search is successful, let x be the node containing i. Assume x is not the root and let y be the parent of x. Replace x by an appropriate descendent of y in the usual fashion and then splay at y.



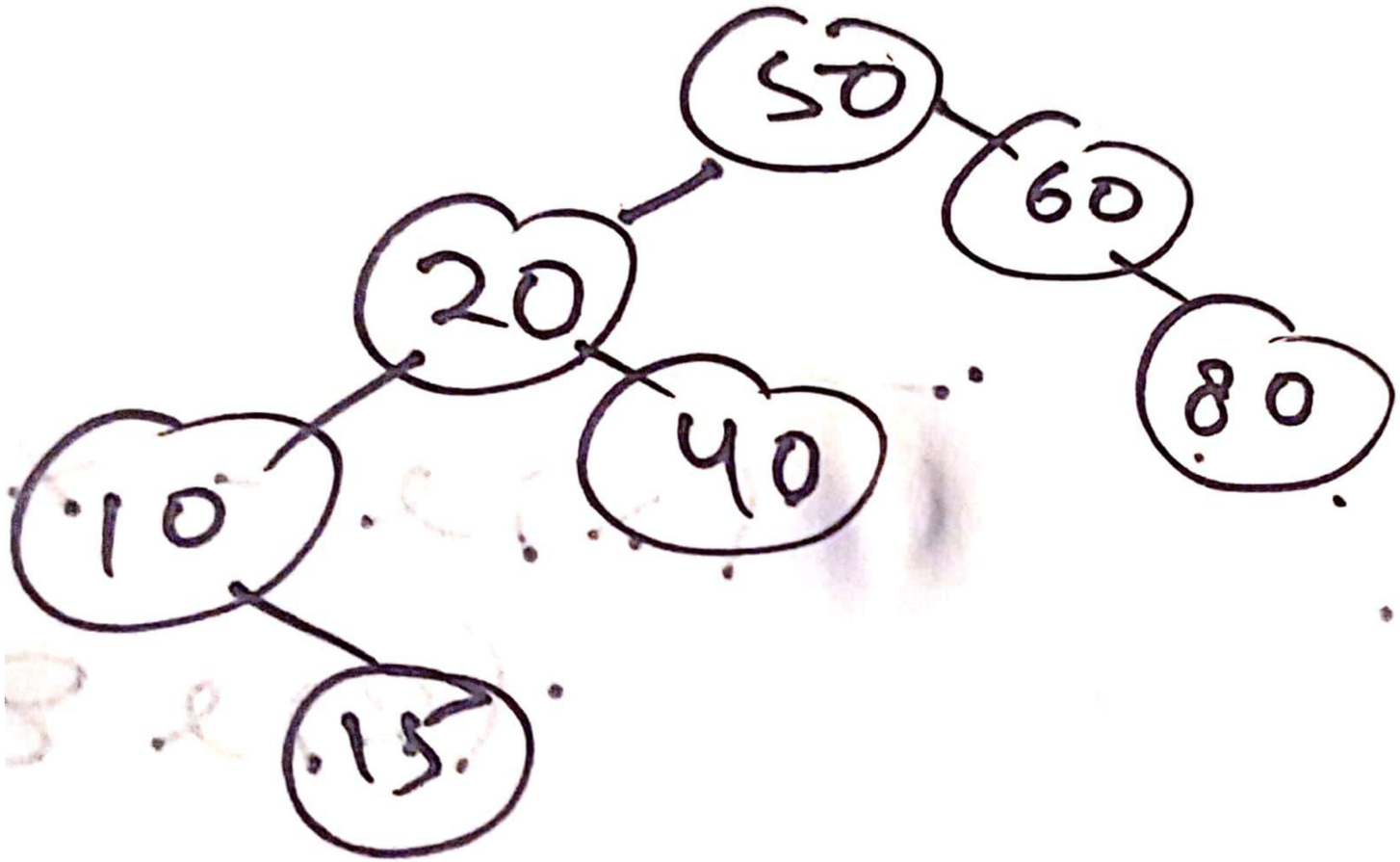
(Delete 30)
⇒

c)



splay at y.

Splay at y.



Top down Splaying

Amortized cost

Recently searched node is being dragged up in the tree towards the root so that next time it takes small time to find it.

- order of bst must be same in the rotations.
- **Access theorem :** Each node gets a weight wi , size s(x) = total weight of its descendants.
- Rank r(x) = log s(x)
- potential function = summation of all ranks
- each fat child/ descendant contributes big to the overall delay in search .

amortized cost to splay x given root t :

3 (r(t)- r(x)) +1 = O(log(s(t)/ s(x)).

- Analyze zig zig : node which is changing the rank will contribute to the potential.
- change in pf = r'x +r'y+r'Z - r(x)-r(y)-r(z).
- in zig zig (x is changing rank increasing , z is decreasing rank) y is same.
-

(11) $x - y - z \Rightarrow x - y - z$

Cost $= 2 + r'x + r'y + r'z - rx - ry - rz$

$r'(x) = r(z)$

$= 2 + r'y + r'z - r(x) - ry)$

$r'y + r'z - r(x) - ry)$

$r'(x) \geq r'(y)$ $\quad r(y) \geq r(x)$.

$\leq 2 + r'(x) + r'(z) - 2r(x)$.

Since. $r'(2) + r(x) = \log S_1'(2) + \log S(x)$

$= 2 \log \left(\frac{S_1'2 + S_1'(x)}{2} \right)$

$\leq$ since. $S_1'2 + S_1x \leq S_1'(x)$

$= 2 \log (S_1'x - \log 2)$.

$= 2(r'x - 1)$

So. $\boxed{r'2 \leq 2r'x - 2 - r(x)}$

put r'2 value →

$$= 2 + r'x + 2(r'x - 2 - r(x) - r(x) - r(x)$$

$$= 2 \quad 3r'x - 3r(x).$$

$$= \frac{3(r'x - r(x))}{}$$

Max. Potential.

Rank.

$\log N$

$\log N-1$

1.

Potential funct^n $\rightarrow$

$$\sum_{K=1}^{n} \text{rank}.$$

$$\sum_{1}^{n} \log n. \Rightarrow O(n \log n)$$

Balance Theorem :

A sequence of m splays in a tree of n nodes takes time
O(m log(n) + n log(n))

Here n log n is drop in potential as
cost / time complexity = actual cost + potential change
= m logn + max potential - min potential
= mlogn + nlogn -0

Advantages of Splay Tree

- Splay tree is the fastest type of binary search tree, which is used in a variety of practical applications such as GCC compilers.

- Improve searching by moving frequently accessed nodes closer to the root node. One of the practical uses is cache implementation
- IP routers use it
- data compression