Data Structure

TREE

UNIT - 3

ANKIT VERMA

WWW.ANKITVERMA.IN
WWW.ANKITVERMA.CO.IN

DEPARTMENT OF INFORMATION TECHNOLOGY

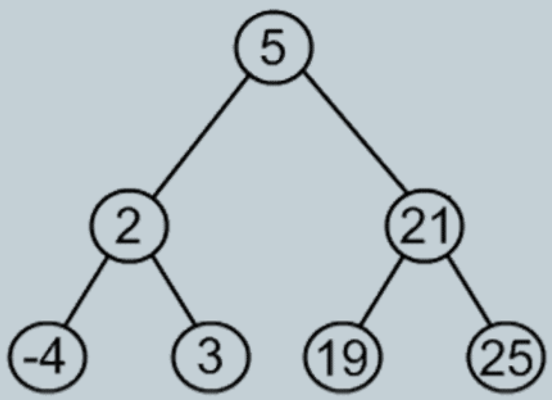
Which Book To Follow ?

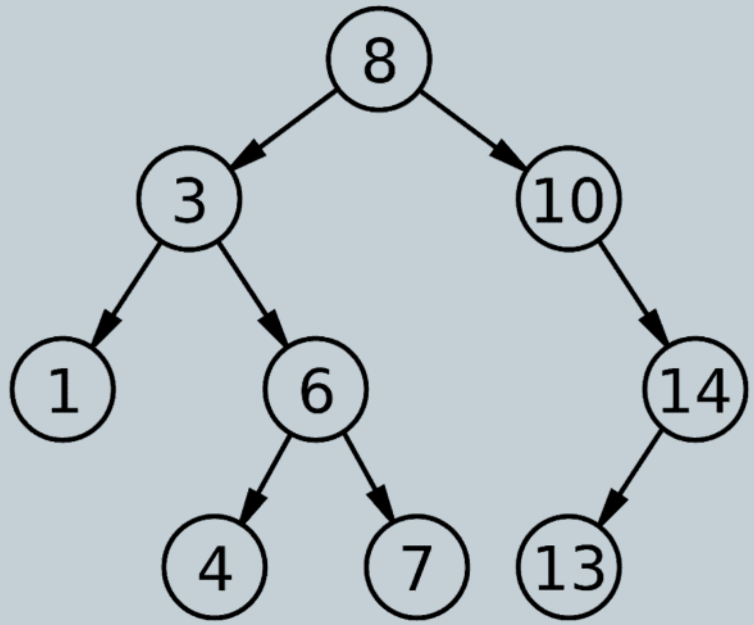
- Text Book
  - Data Structures, Schaum's Outlines, Seymour Lipschutz
- Reference Book
  - Data Structure Using C, Udit Aggarwal

Binary Search Tree

Binary Search Tree (BST)

- Binary Search Tree
  - Binary Tree T is called Binary Search Tree (or Binary Sorted Tree) if each Node N of T has following property:
    - Value at N is Greater than every value in the Left Subtree & is Less than every value in Right Subtree of N.
  - Assumption:
    - All the Node Values are Distinct.





Binary Search Tree

- Binary Search Tree Construction
  - Suppose six Numbers are Inserted in order into empty Binary Search Tree:
    - 40, 60, 50, 33, 55, 11

Searching in Binary Search Tree

- ALGORITHM: FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
  - The algorithm finds location LOC of ITEM in Binary Search Tree T and also location of parent PAR of ITEM.

1. If ROOT = NULL, then: Set LOC := NULL and PAR := Null, and Return.
2. If ITEM = INFO[ROOT], then Set LOC := ROOT and PAR := NULL, and Return.
3. If ITEM < INFO[ROOT], then:

    Set PTR := LEFT[ROOT] and SAVE := ROOT.

   Else:

    Set PTR := RIGHT[ROOT] and SAVE := ROOT.
4. Repeat Steps 5 & 6 while PTR != NULL:
5. If ITEM = INFO[PTR], then: Set LOC := PTR and PAR := SAVE, and Return.
6. If ITEM < INFO[PTR], then:
7. Set SAVE := PTR and PTR := LEFT[PTR].
8. Else:
9. Set SAVE := PTR and PTR := RIGHT[PTR].
10. Set LOC := NULL and PAR := SAVE
11. Exit.

Insertion in Binary Search Tree

- ALGORITHM: INSBST (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)
  - The algorithm finds location LOC of ITEM in Binary Search Tree T and adds ITEM as new node in T at location LOC.

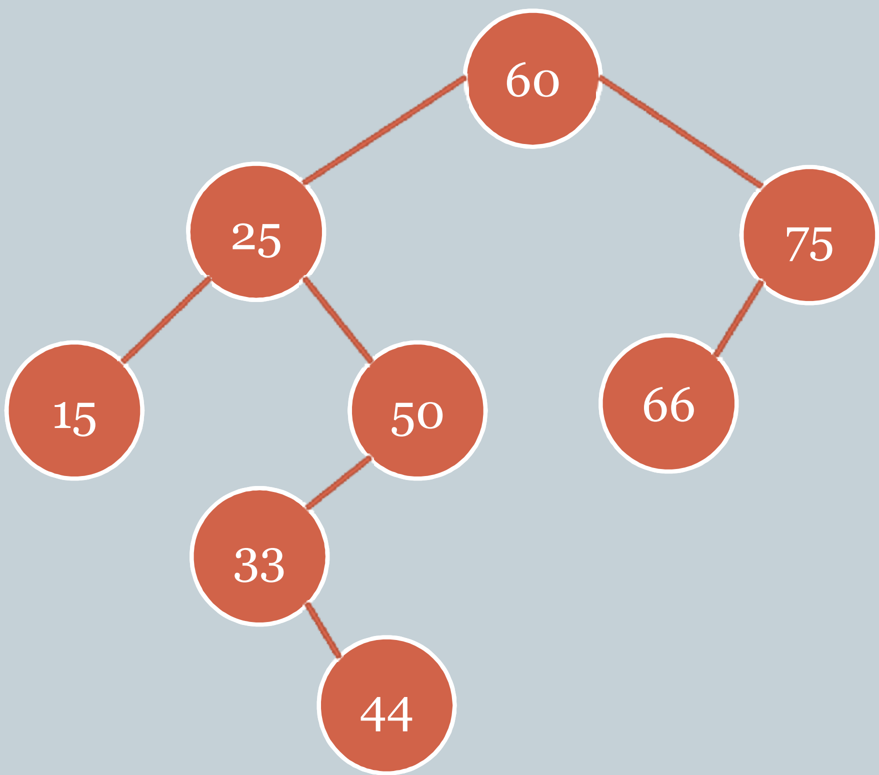
1. Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. If LOC = NULL, then Exit.
3. a) If AVAIL = NULL, then: Write OVERFLOW, and Exit.
   b) Set NEW := AVAIL, AVAIL := LEFT[AVAIL] and INFO [NEW] := ITEM.
   c) Set LOC := NEW, LEFT[NEW] := NULL and RIGHT[NEW] := NULL
4. If PAR = NULL, then:
   Set ROOT := NEW.
   Else If ITEM < INFO[PAR], then:
   Set LEFT[PAR] := NEW.
   Else
   Set RIGHT[PAR] := NEW.
5. Exit

Deleting in Binary Search Tree

- Deleting in Binary Search Tree (BST)
  - Three Cases of Deletion:
    - Case 1
      - Node N has No Children
    - Case 2
      - Node N has exactly One Children
    - Case3
      - Node N has Two Children

Deleting in Binary Search Tree

- Linked Representation of Binary Search Tree (BST)
  - Suppose Binary Search Tree T appears in Memory as follows:



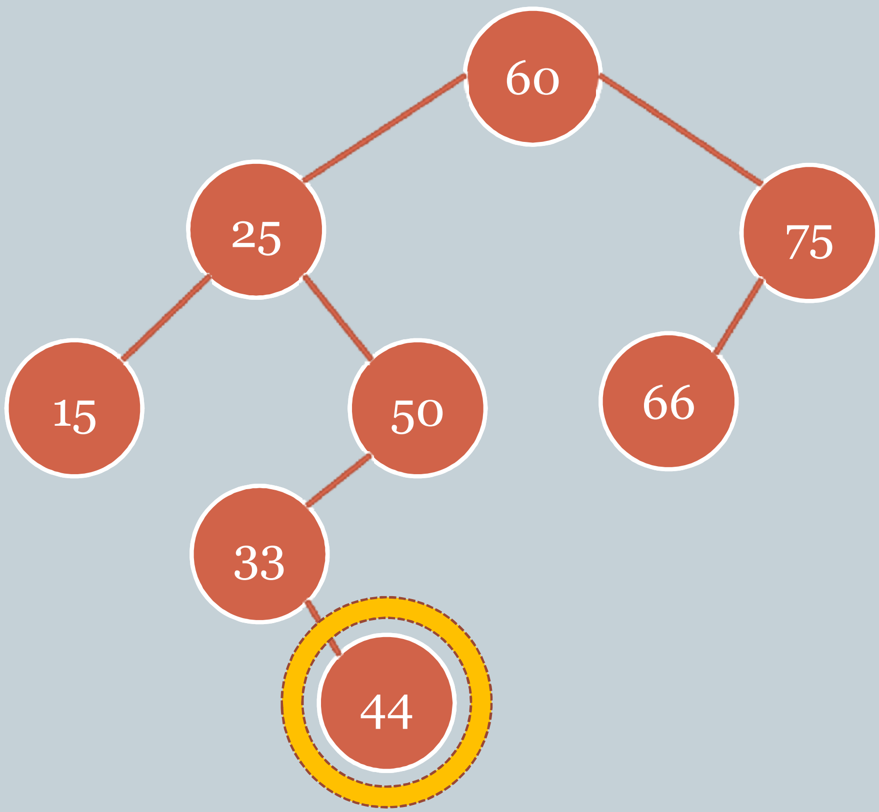
ROOT: 3

AVAIL: 5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

Deleting in Binary Search Tree

- Case 1 – Node N has No Children
  - Node 44 is Deleted from T then T appears in Memory as follows:



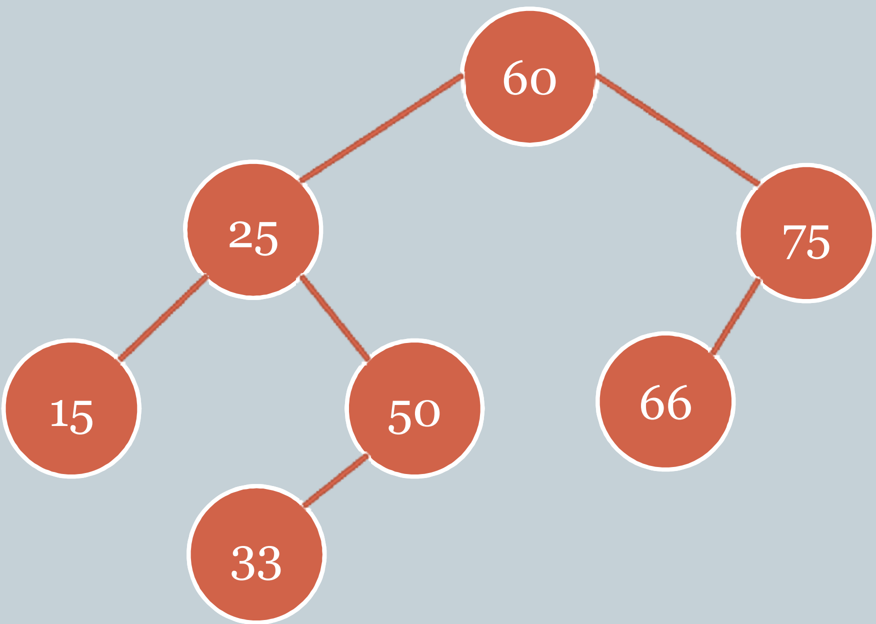
ROOT: 3

AVAIL: 5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

Deleting in Binary Search Tree

- Case 1 – Node N has No Children
  - Node 44 is Deleted from T then T appears in Memory as follows:



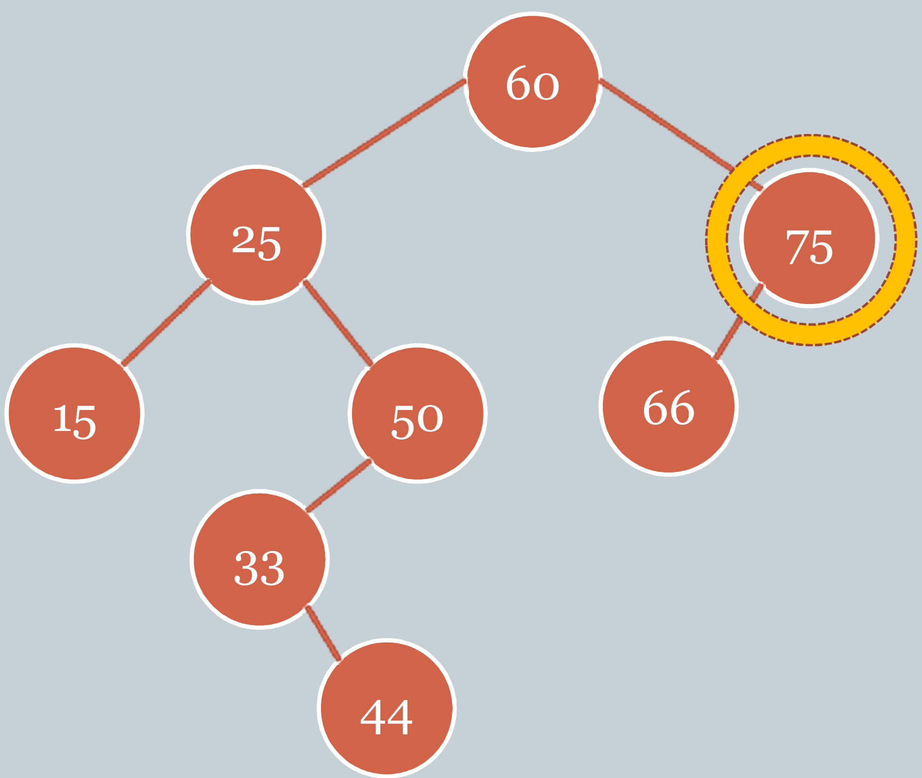
ROOT: 3

AVAIL: 9

	INFO	LEFT	RIGHT
1	33	0	0
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9		5	
10	50	1	0

Deleting in Binary Search Tree

- Case 2 – Node N Has exactly One Child
  - Node 75 is Deleted from T then T appears in Memory as follows:



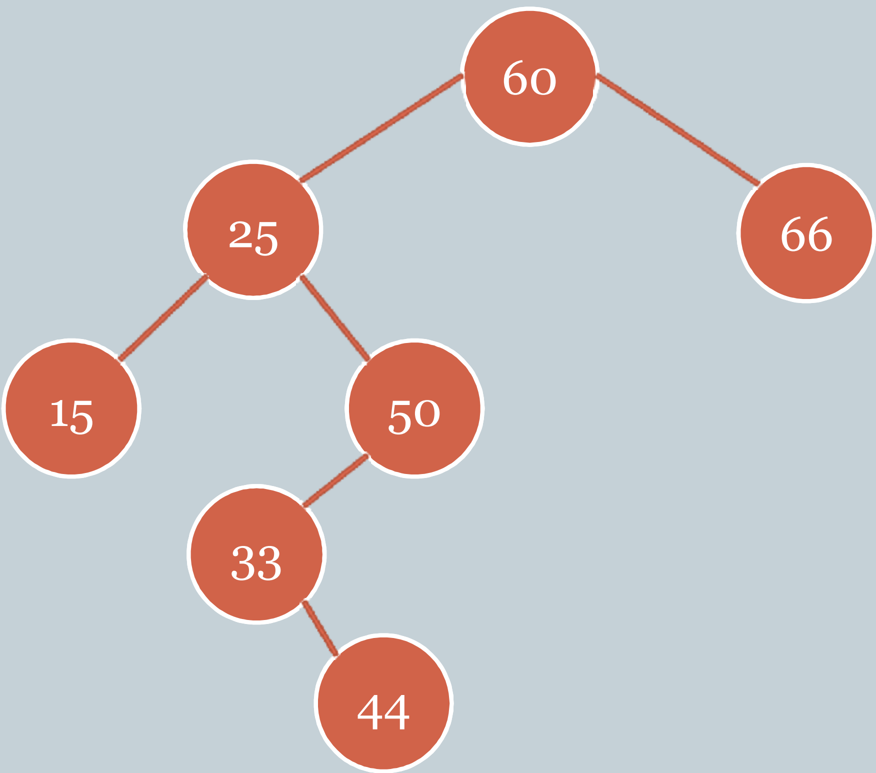
ROOT: 3

AVAIL: 5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

Deleting in Binary Search Tree

- Case 2 – Node N Has exactly One Child
  - Node 75 is Deleted from T then T appears in Memory as follows:



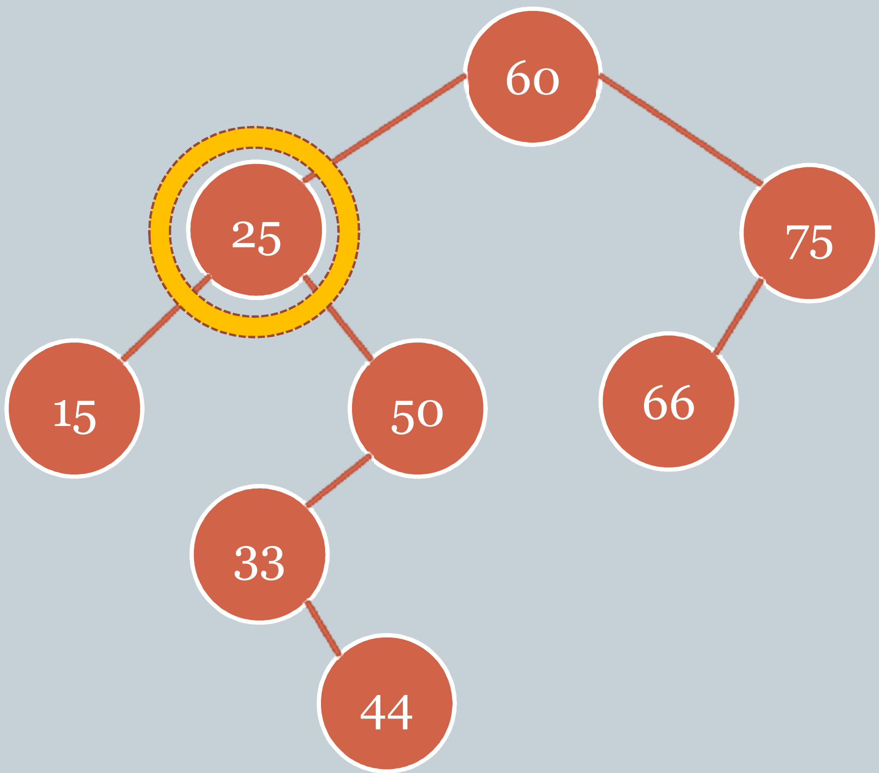
ROOT: 3

AVAIL: 7

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	4
4	66	0	0
5		6	
6		0	
7		5	
8	15	0	0
9	44	0	0
10	50	1	0

Deleting in Binary Search Tree

- Case 3 – Node N has Two Children
  - Node 25 is Deleted from T then T appears in Memory as follows:



ROOT

3

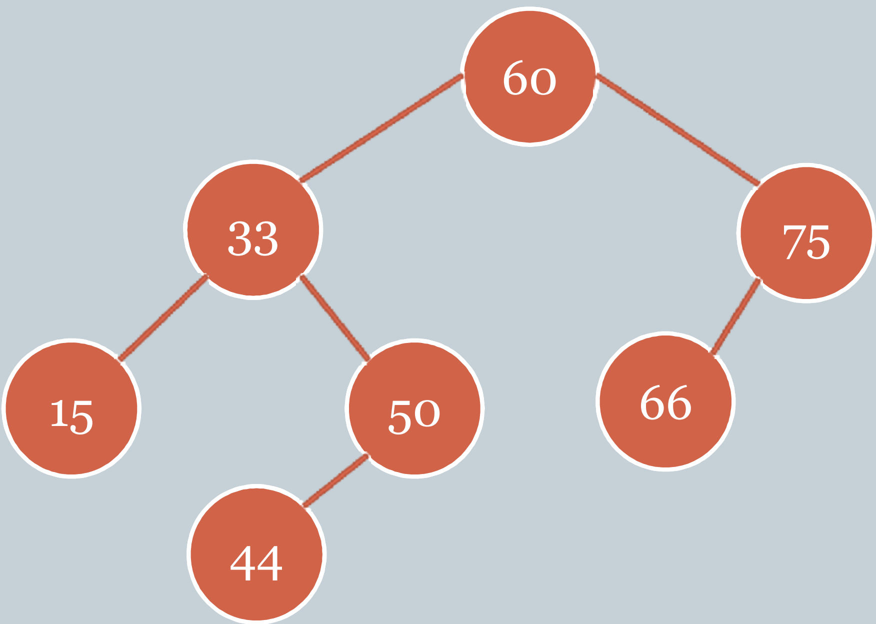
AVAIL

5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

Deleting in Binary Search Tree

- Case 3 – Node N has Two Children
  - Node 25 is Deleted from T then T appears in Memory as follows:



ROOT: 3

AVAIL: **2**

	INFO	LEFT	RIGHT
1	33	**8**	**10**
2		**5**	
3	60	**1**	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	**9**	0

Deleting in Binary Search Tree

- ALGORITHM: CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
  - The algorithm deletes node N at location LOC, where N does not have two children, it have one child or no child. Pointer PAR have location of Parent of N, if PAR = NULL indicates N is root node. Pointer CHILD gives location of only child of N, if CHILD = NULL indicates N has no children.

1. If LEFT[LOC] = NULL and RIGHT[LOC] = NULL, then:
   Set CHILD := NULL.
   Else If LEFT[LOC] != NULL, then:
   Set CHILD := LEFT[LOC].
   Else
   Set CHILD := RIGHT[LOC].
2. If PAR != NULL, then:
   If LOC = LEFT[PAR], then:
   Set LEFT[PAR] := CHILD
   Else:
   Set RIGHT[PAR] := CHILD.
   Else:
   Set ROOT := CHILD.
3. Exit

Deleting in Binary Search Tree

- ALGORITHM: CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
  - The algorithm deletes node N at location LOC, where N have two children. Pointer PAR have location of Parent of N, if PAR = NULL indicates N is root node. Pointer SUC gives location of inorder successor of N, and PERSUC gives location of parent of inorder successor.

1. a) Set PTR := RIGHT[LOC] and SAVE := LOC
   b) Repeat while LEFT[PTR] != NULL:
      Set SAVE := PTR and PTR := LEFT[PTR].
   c) Set SUC := PTR and PARSUC := SAVE.
2. Call CASEA(INFO , LEFT, RIGHT, ROOT, SUC, PARSUC).
3. a) If PAR != NULL, then:
      If LOC = LEFT[PAR], then:
         Set LEFT[PAR] := SUC.
      Else:
         Set RIGHT[PAR] := SUC.
   Else:
      Set ROOT := SUC.
   b) Set LEFT[SUC] := LEFT[LOC] and RIGHT[SUC] := RIGHT[LOC].
3. Exit

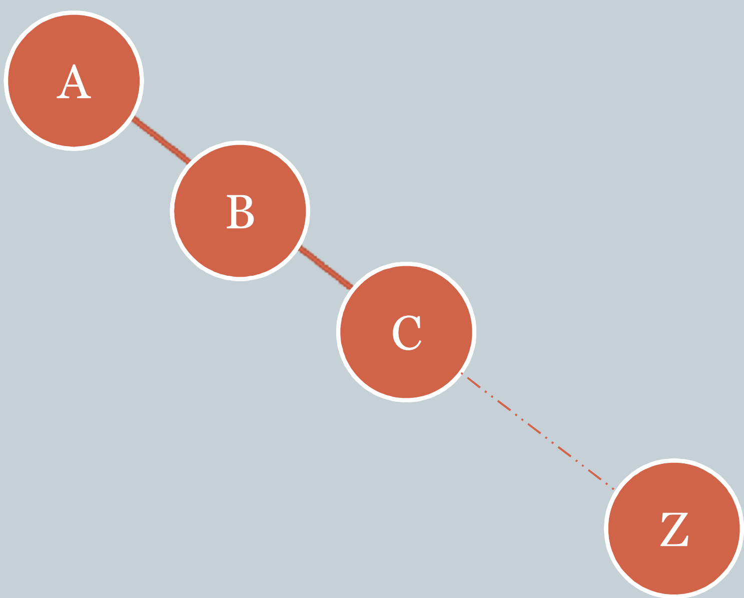
Deleting in Binary Search Tree

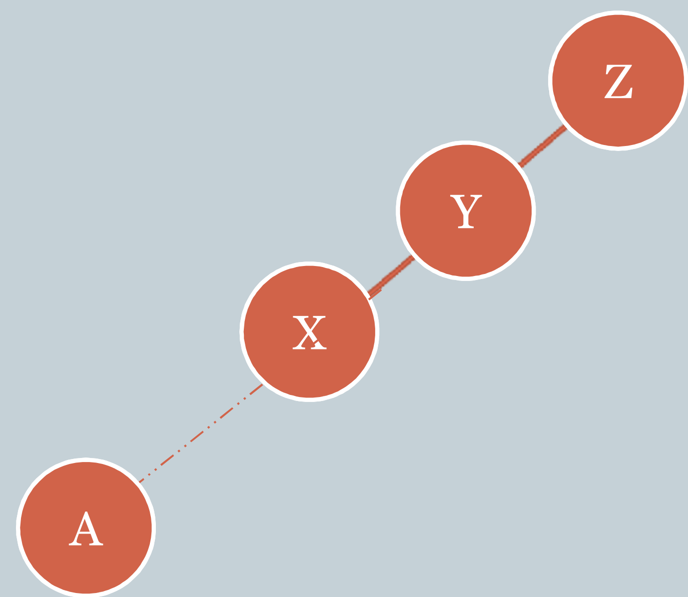
- ALGORITHM: DEL (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)
  - T is Binary Search Tree in Memory and an ITEM of information is given. This algorithm deletes ITEM from the tree T.
    1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
    2. If LOC = NULL, then: Write: ITEM not in tree, and Exit.
    3. If RIGHT[LOC] != NULL and LEFT[LOC] != NULL, then:
       Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
       Else:
       Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
    2. Set LEFT[LOC] := AVAIL and AVAIL := LOC.
    3. Exit

AVL Tree

Skewed Binary Search Tree

- Skewed Binary Search Tree
  - Right Skewed
    - A, B, C, D, ... , Z
  - Left Skewed
    - Z, Y, X, W, ... , A





Skewed Binary Search Tree

- Disadvantages of Skewed Binary Search Tree
  - Worst Case Time Complexity of Search is O(n)
    - Take More Time for Searching in Worst Case
- Solution to Overcome Disadvantage
  - Binary Search Tree should be of Balanced Height
    - Worst Case Time Complexity of Search will be O(log n)
      - Take Less Time for Searching in Worst Case

AVL Tree

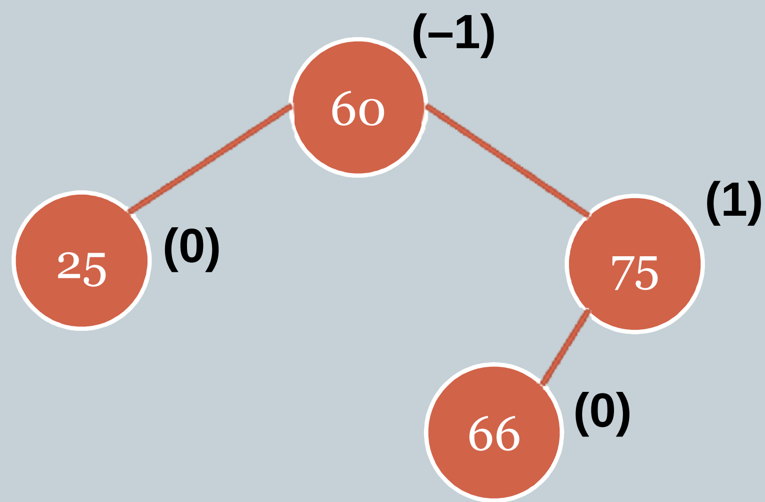
- Adelson – Velskii & Lendis (AVL) Tree
  - Most popular Balanced Tree
  - Introduced in 1962 by Adelson – Velskii & Lendis
  - Also known as AVL Tree or AVL Search Tree
  - Worst Case Time Complexity of Search is Less
  - Take Less Time for Searching in Worst Case

AVL Tree

- Definition
  - An empty Binary Tree is an AVL Tree. A non empty Binary T is an AVL Tree if given T^L & T^R to be the Left & Right Subtrees of T and $h(T^L)$ & $h(T^R)$ to be the Heights of Subtrees T^L & T^R respectively, T^L & T^R are AVL Trees and $|h(T^L) - h(T^R)| <= 1$
  - Balance Factor (BF)
    - Is known as Balance Factor (BF)
    - Balancing Factor of a Node in AVL Tree can be either 0, 1 or – 1

AVL Tree

- Representation of AVL Tree
  - AVL Tree like Binary Search Tree are represented using Linked Representation and every Node registers its Balance Factor



Insertion in AVL Tree

- Insertion in AVL Tree
  - Inserting Element in AVL Tree is similar to Binary Search Tree
  - After Insertion, Balance Factor of any Node may affected and Tree may become Unbalanced
  - Rotations are made on Unbalanced Tree to Restore the Balanced Tree
  - To perform Rotation it is necessary to identify Node A whose BF(A) is neither 0, 1 or –1, and

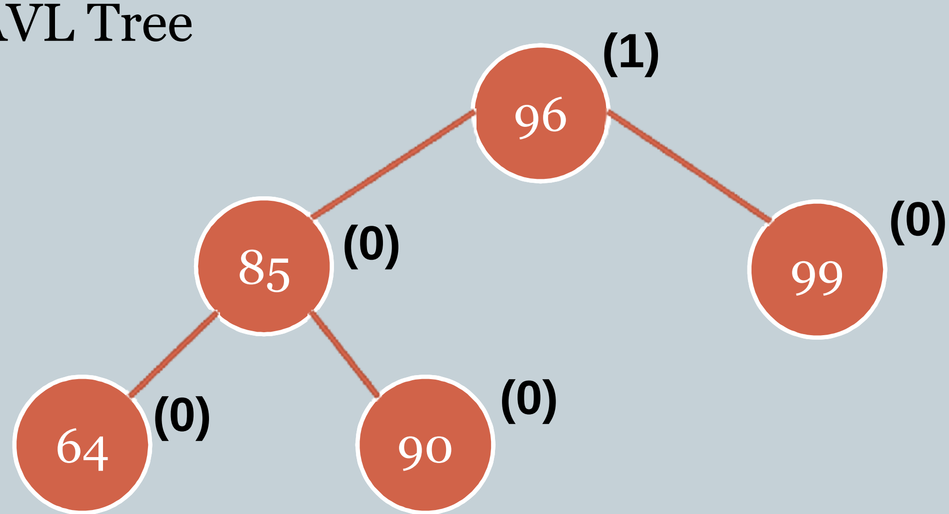
    which is nearest ancestor to Inserted Node on path from Inserted Node to Root
    - This implies that all Nodes on the Path from the Inserted Node to A will have their Balance Factors to be either 0, 1 or –1

Rotations while Insertion in AVL Tree

- Rebalancing Rotations while Insertion in AVL Tree are classified into Four Types:
  - LL Rotation
    - Inserted Node is in Left Subtree of Left Subtree of Node A
  - RR Rotation
    - Inserted Node is in Right Subtree of Right Subtree of Node A
  - LR Rotation
    - Inserted Node is in Right Subtree of Left Subtree of Node A
  - RL Rotation
    - Inserted Node is in Left Subtree of Right Subtree of Node A

LL Rotation

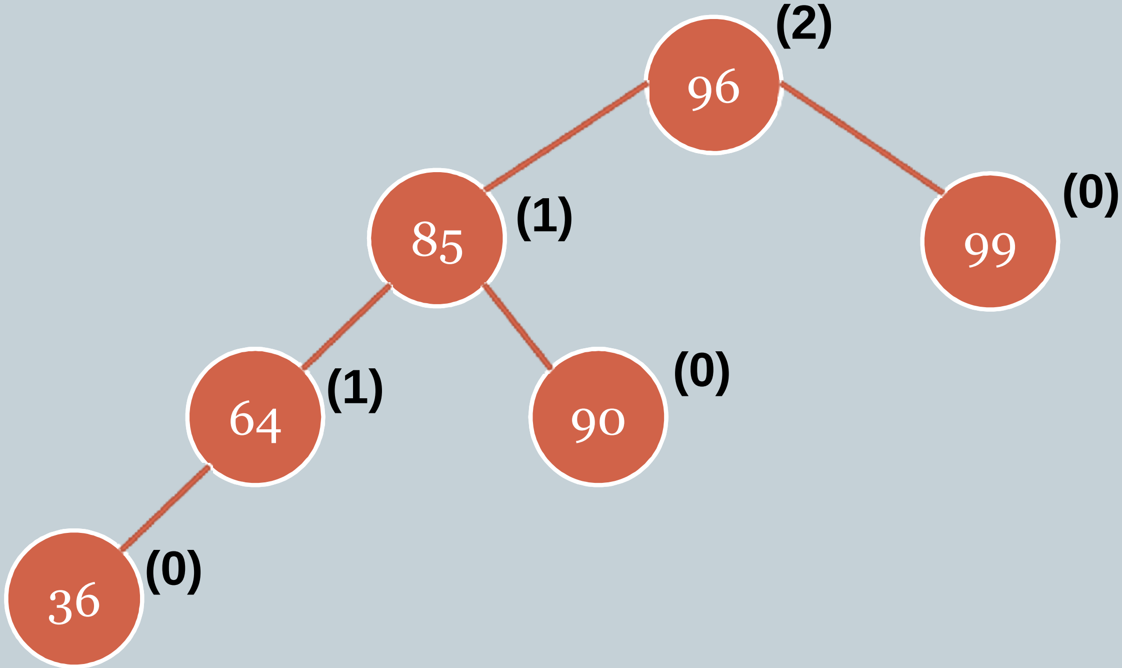
- LL Rotation
  - Inserted Node is in Left Subtree of Left Subtree of Node A
  - Example:
    - Initial AVL Tree



[Balanced AVL Tree]

LL Rotation

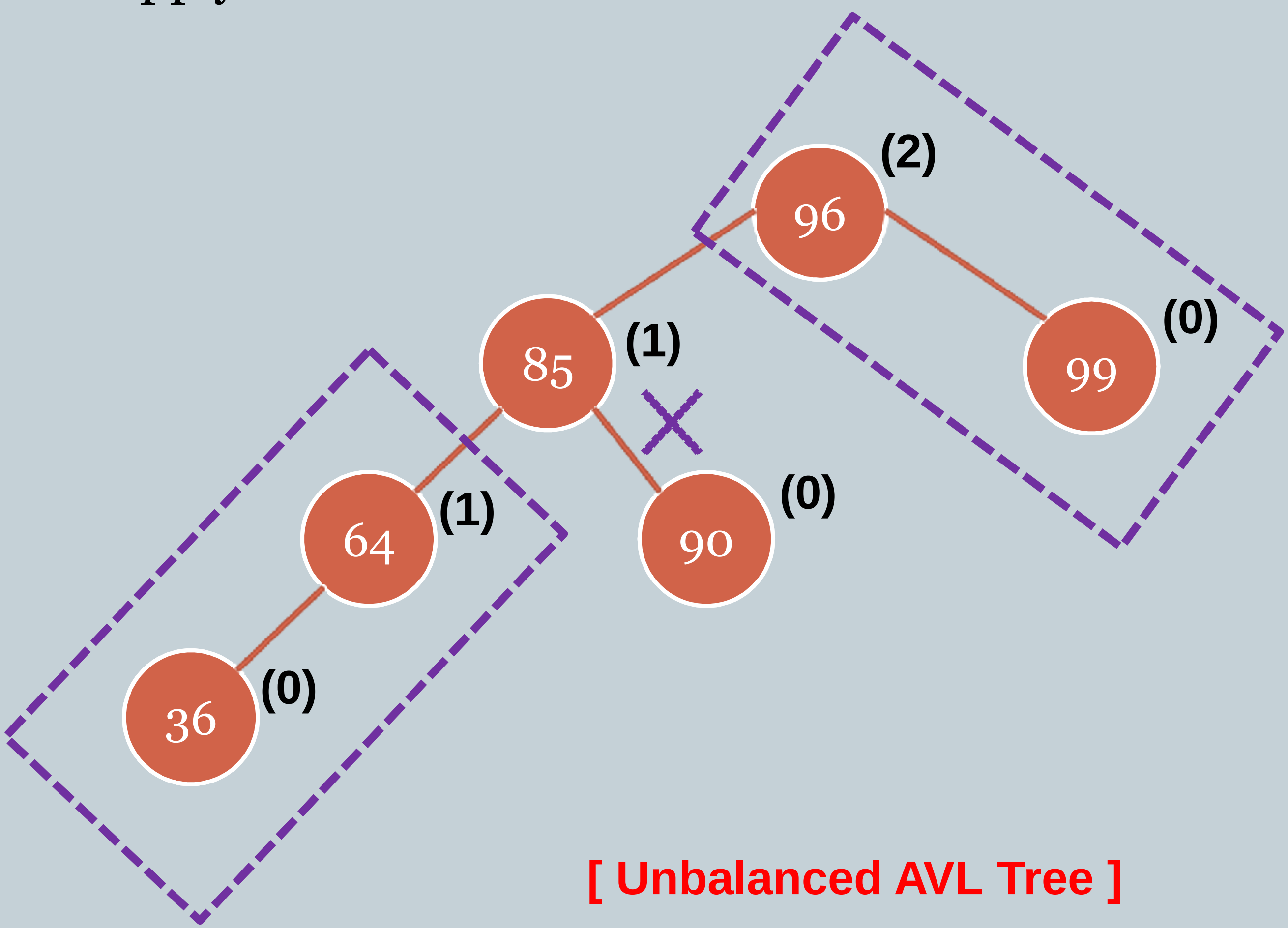
- Insert Node 36



[Unbalanced AVL Tree]

LL Rotation

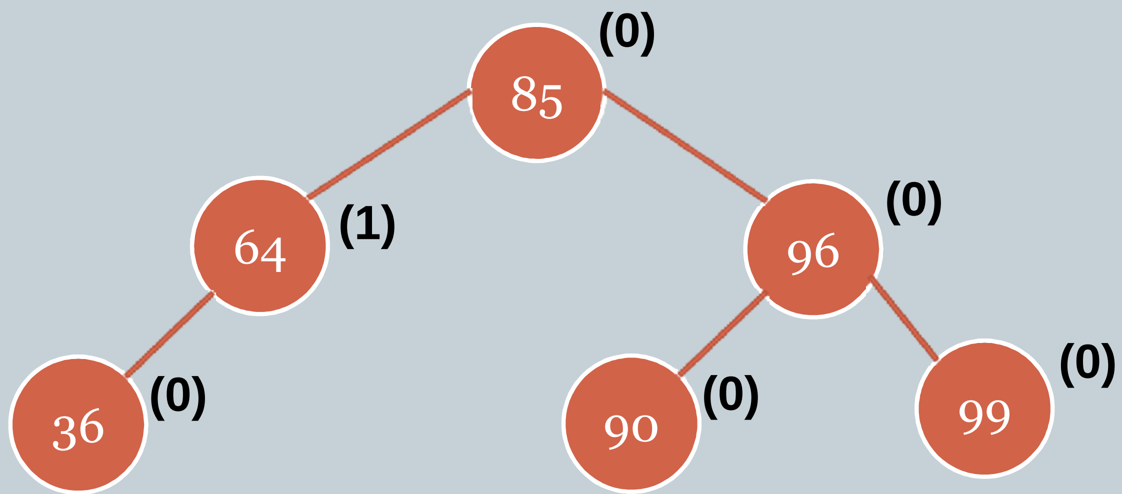
- Apply LL Rotation



[Unbalanced AVL Tree]

LL Rotation

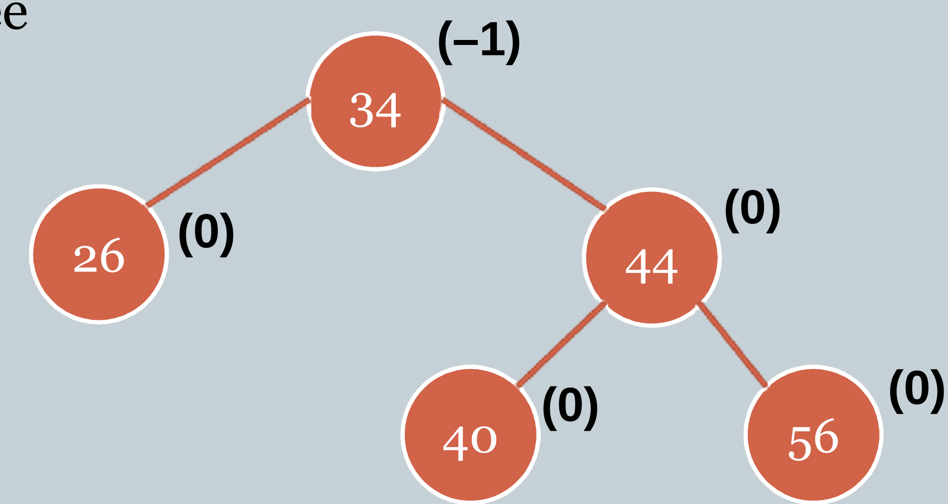
- After LL Rotation



[Balanced AVL Tree]

RR Rotation

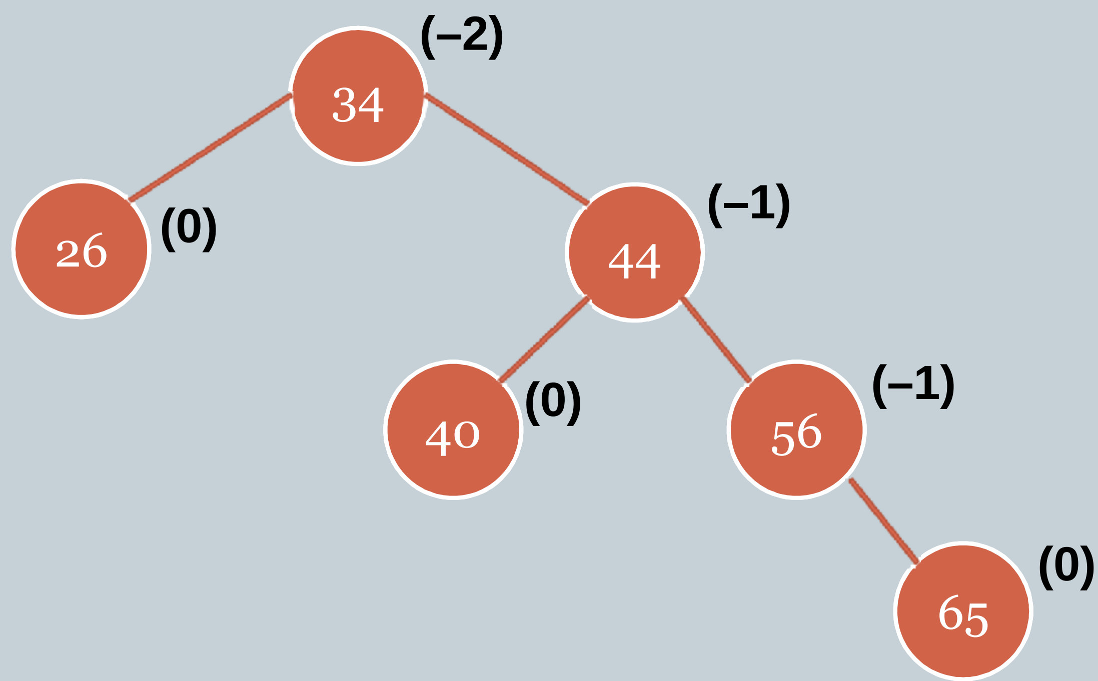
- RR Rotation
  - Inserted Node is in Right Subtree of Right Subtree of Node A
  - Example:
    - Initial AVL Tree



[Balanced AVL Tree]

RR Rotation

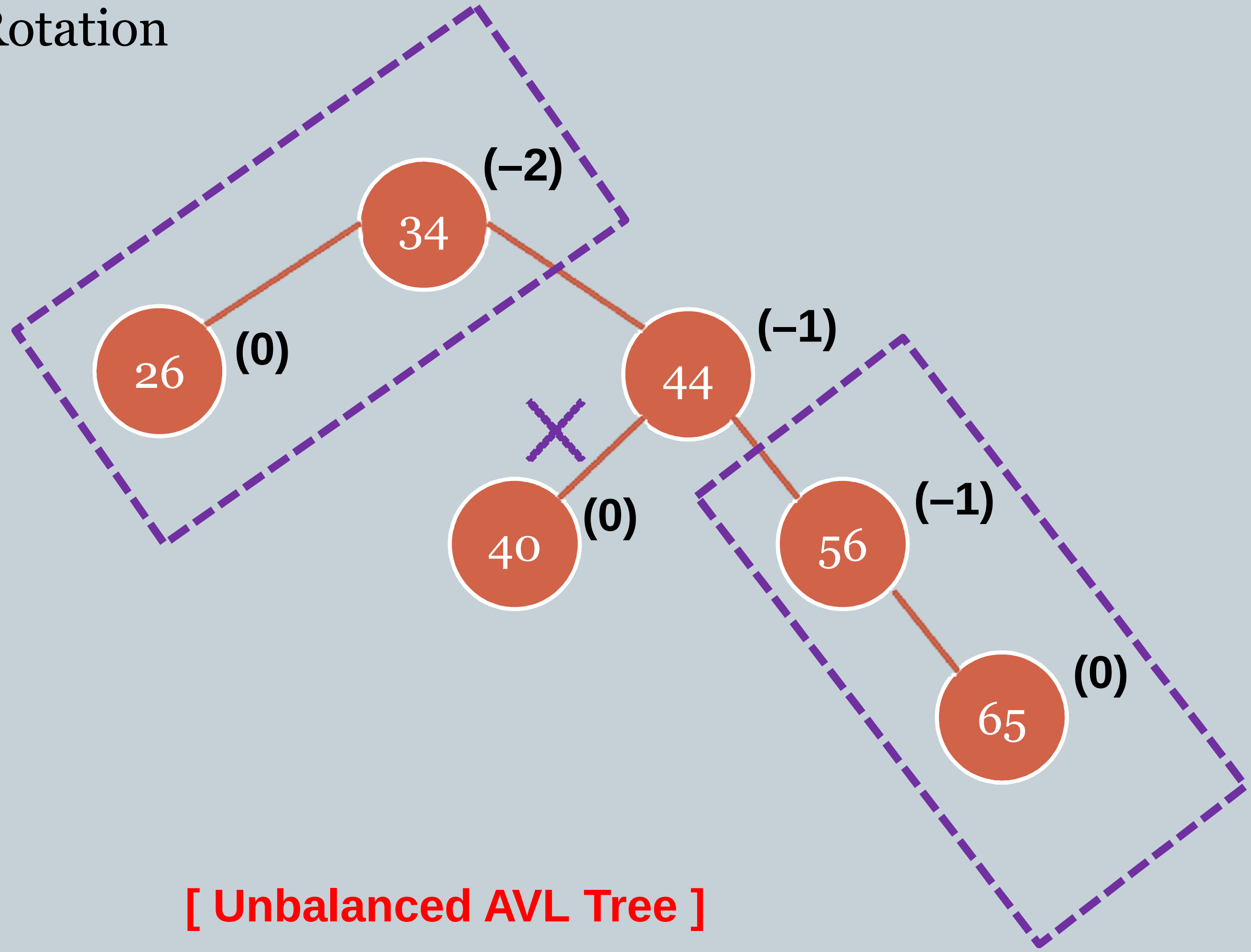
- Insert 65



[Unbalanced AVL Tree]

RR Rotation

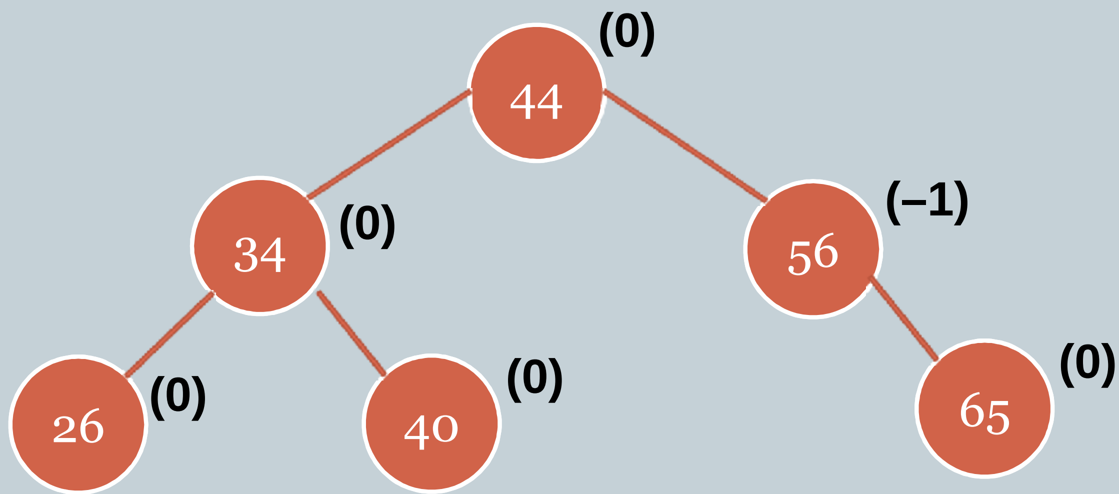
- Apply RR Rotation



[Unbalanced AVL Tree]

RR Rotation

- After RR Rotation



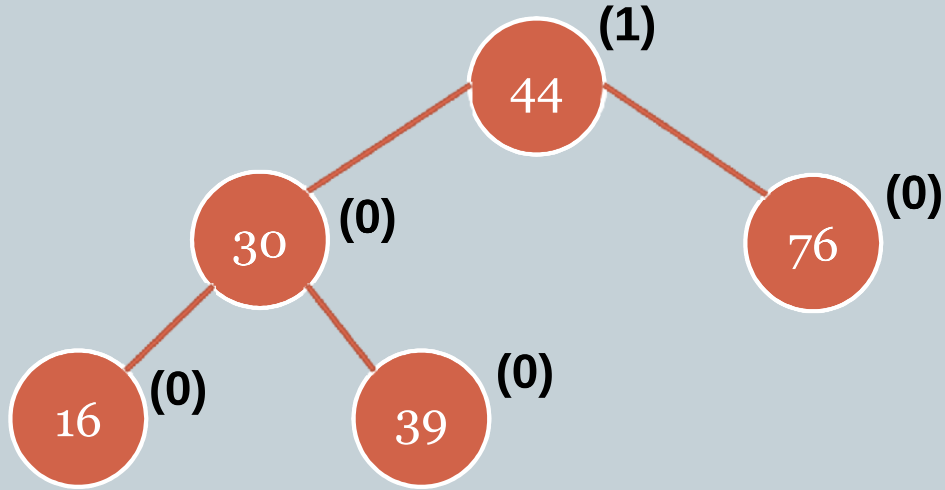
[Balanced AVL Tree]

LR & RL Rotation

- LR & RL Rotation
  - Both are Similar in nature but are Mirror Images of One Another
  - LL & RR are called **Single Rotations** but
    LR & RL are called **Double Rotations**.
  - LR Rotation
    - Inserted Node is in Right Subtree of Left Subtree of Node A
    - LR is accomplished by RR followed by LL Rotation
  - RL Rotation
    - Inserted Node is in Right Subtree of Left Subtree of Node A
    - RL is accomplished by LL followed by RR Rotation
  - Time Complexity of AVL Insertion is O(height) = O(log n)

LR Rotation

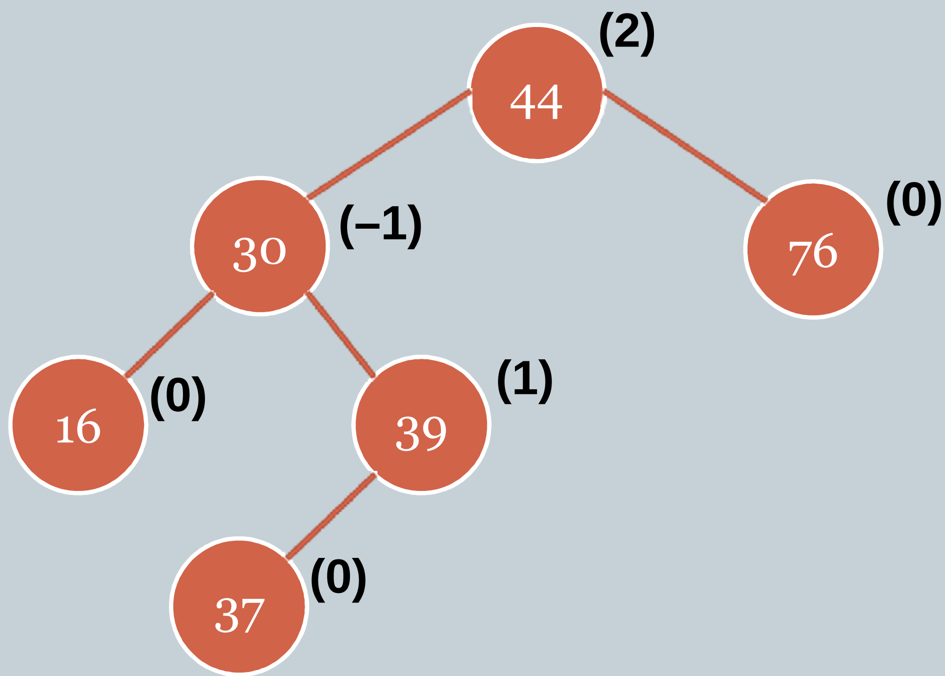
- Example:
  - Initial AVL Tree



[Balanced AVL Tree]

LR Rotation

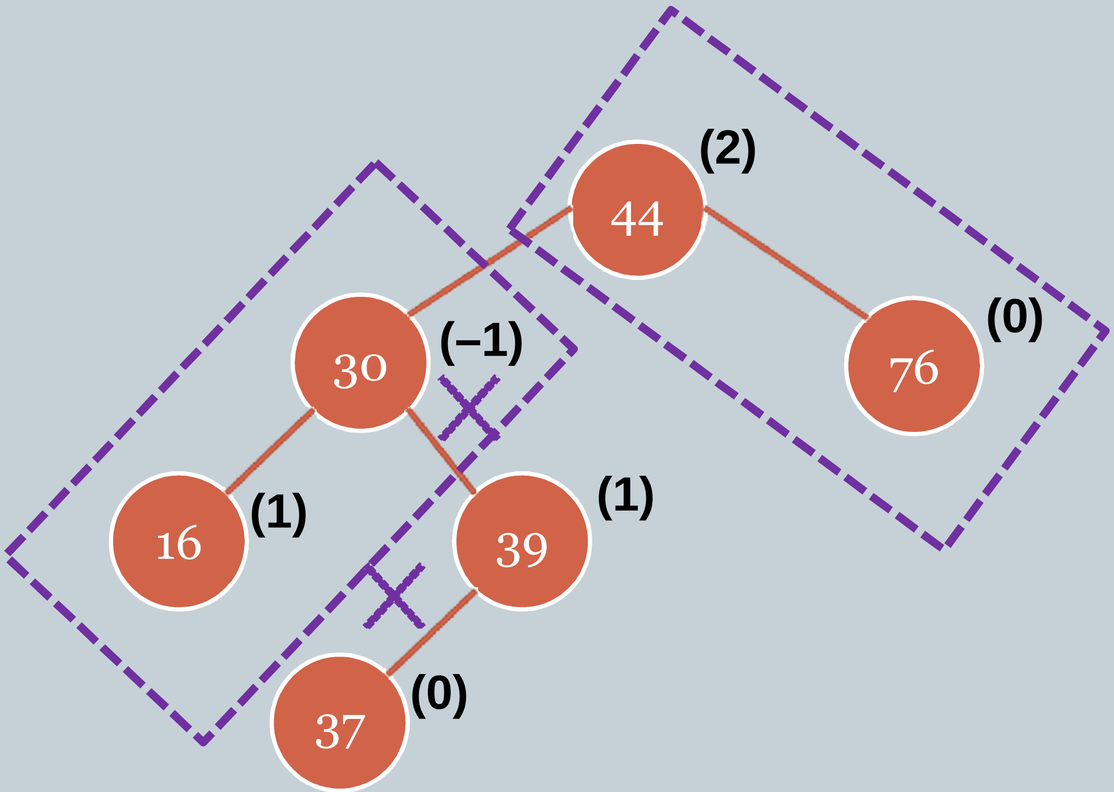
- Insert Node 37



[Unbalanced AVL Tree]

LR Rotation

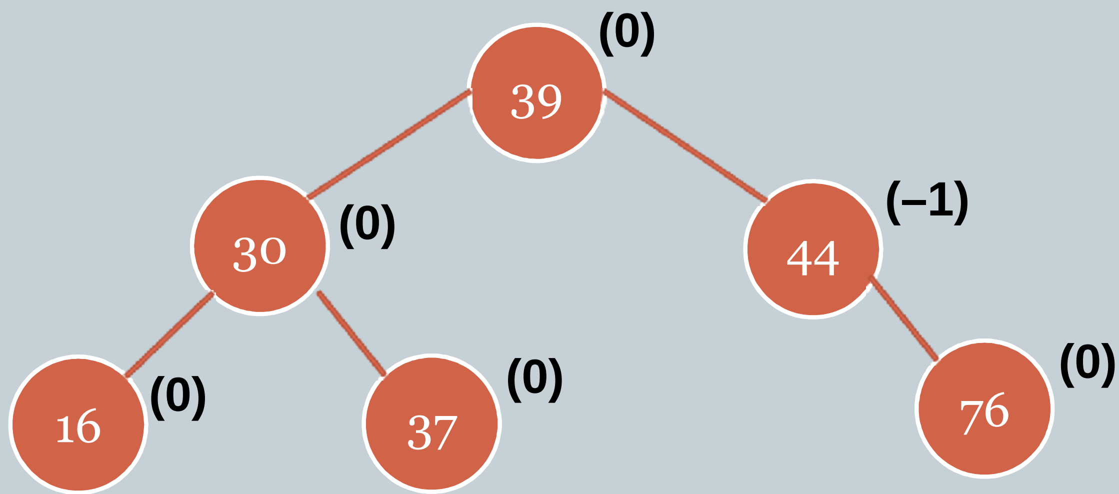
- Apply LR Rotation



[Unbalanced AVL Tree]

LR Rotation

- After LR Rotation



[Balanced AVL Tree]

AVL Tree Construction

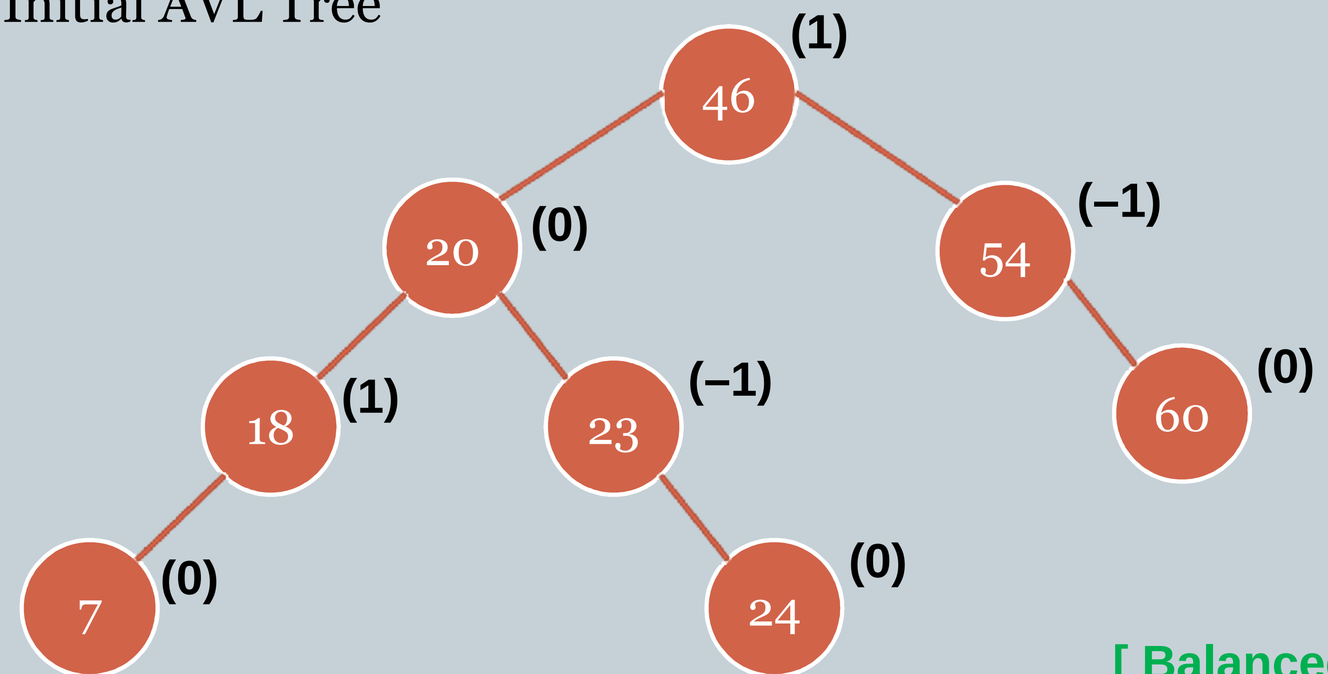
- Construct AVL Tree for following Elements:
  - 64, 1, 14, 26, 13, 110, 98, 85

Deletion in AVL Tree

- Deletion in AVL Tree
  - Imbalance due to Deletion, One or More Rotations need to be applied to Balance the Tree
  - After Deletion of Node X, let A be Closest Ancestor Node on path from X to Root, with BF +2 or –2
  - To Restore Balance the Rotation are classified as L or R depend on whether Deletion occurred at Left or Right Subtree of A
  - Depending on Value of BF(B) where B is the Root of Left or Right Subtree of A, the
    - R Rotation are further classified
      - R 0, R 1 & R –1
    - L Rotation are further classified
      - L 0, L 1 & L –1

R0 Rotation

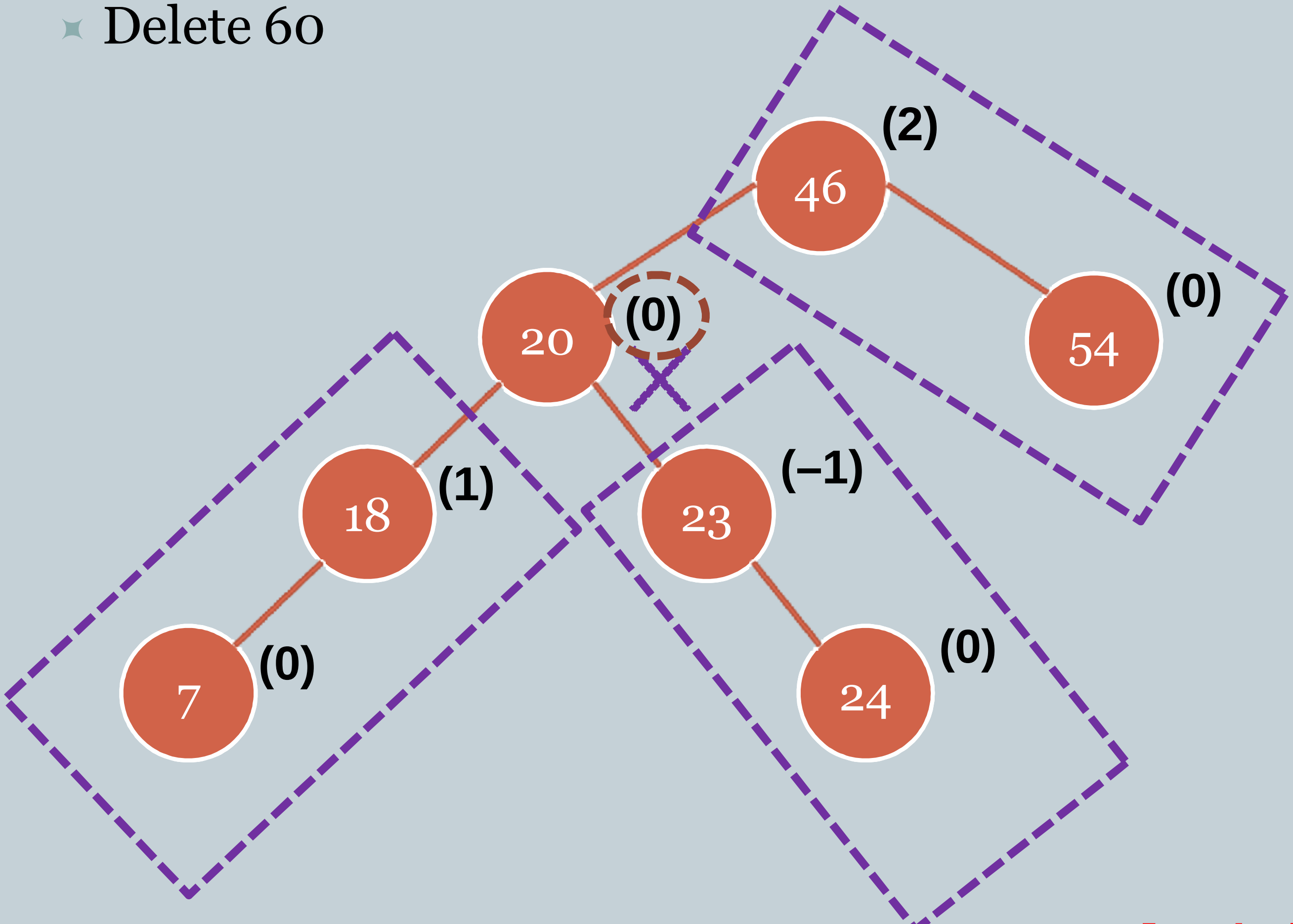
- R0 Rotation
  - If BF(B) = 0, the R0 Rotation is executed
  - Example:
    - Initial AVL Tree



[Balanced AVL Tree]

R0 Rotation

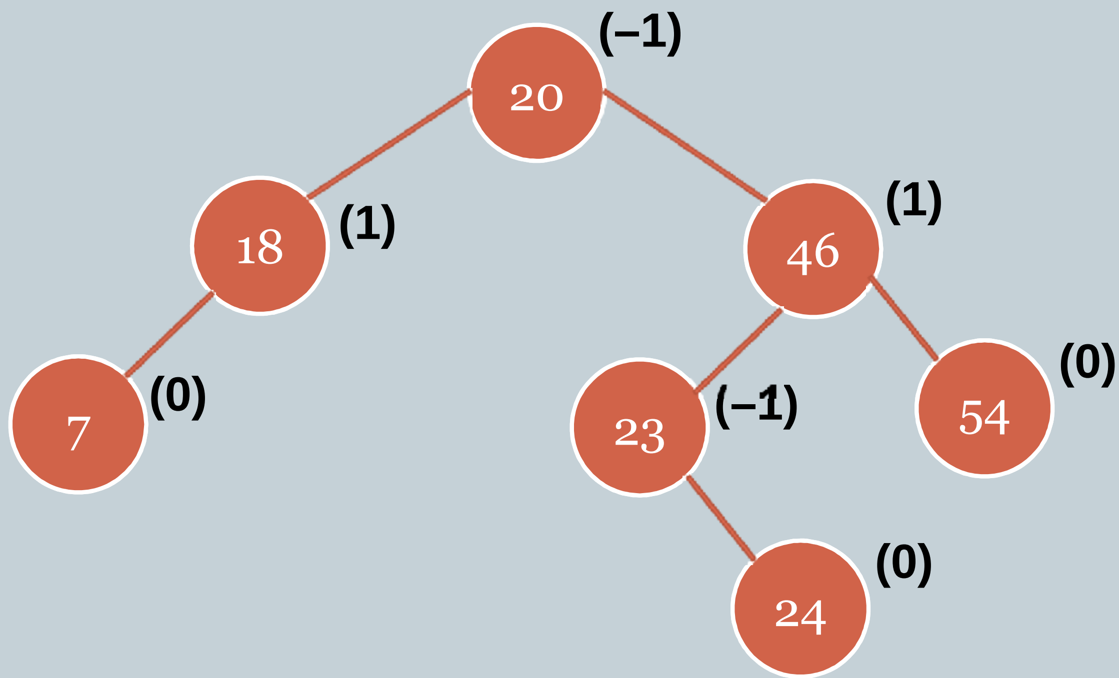
- Delete 60



[Unbalanced AVL Tree]

RO Rotation

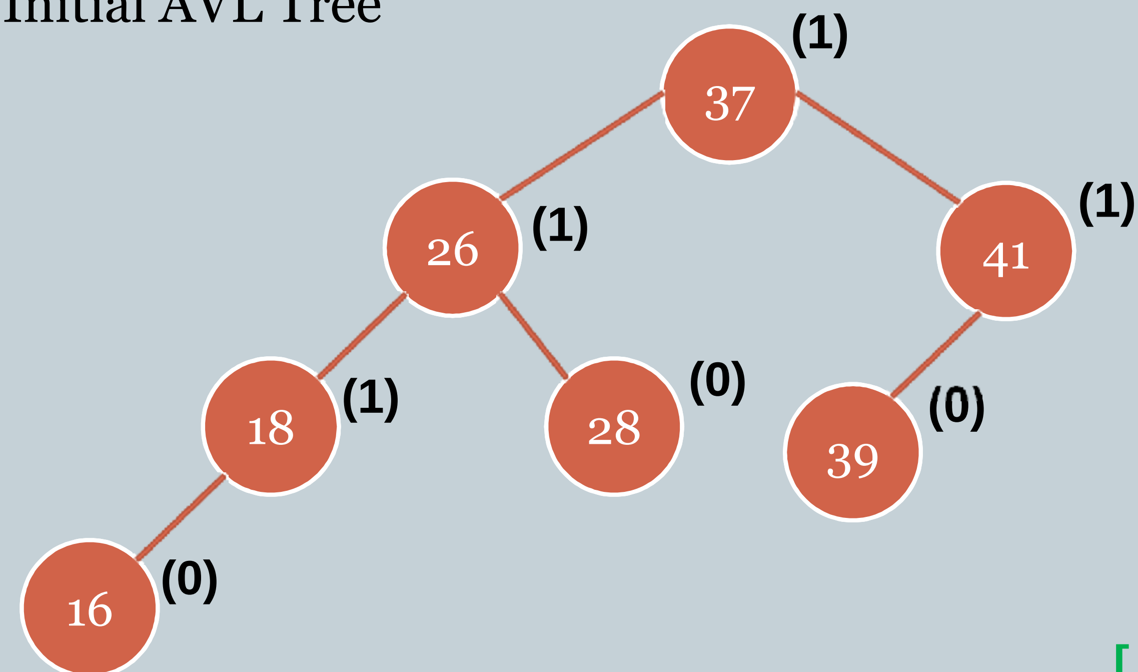
- After RO Rotation



[Balanced AVL Tree]

R1 Rotation

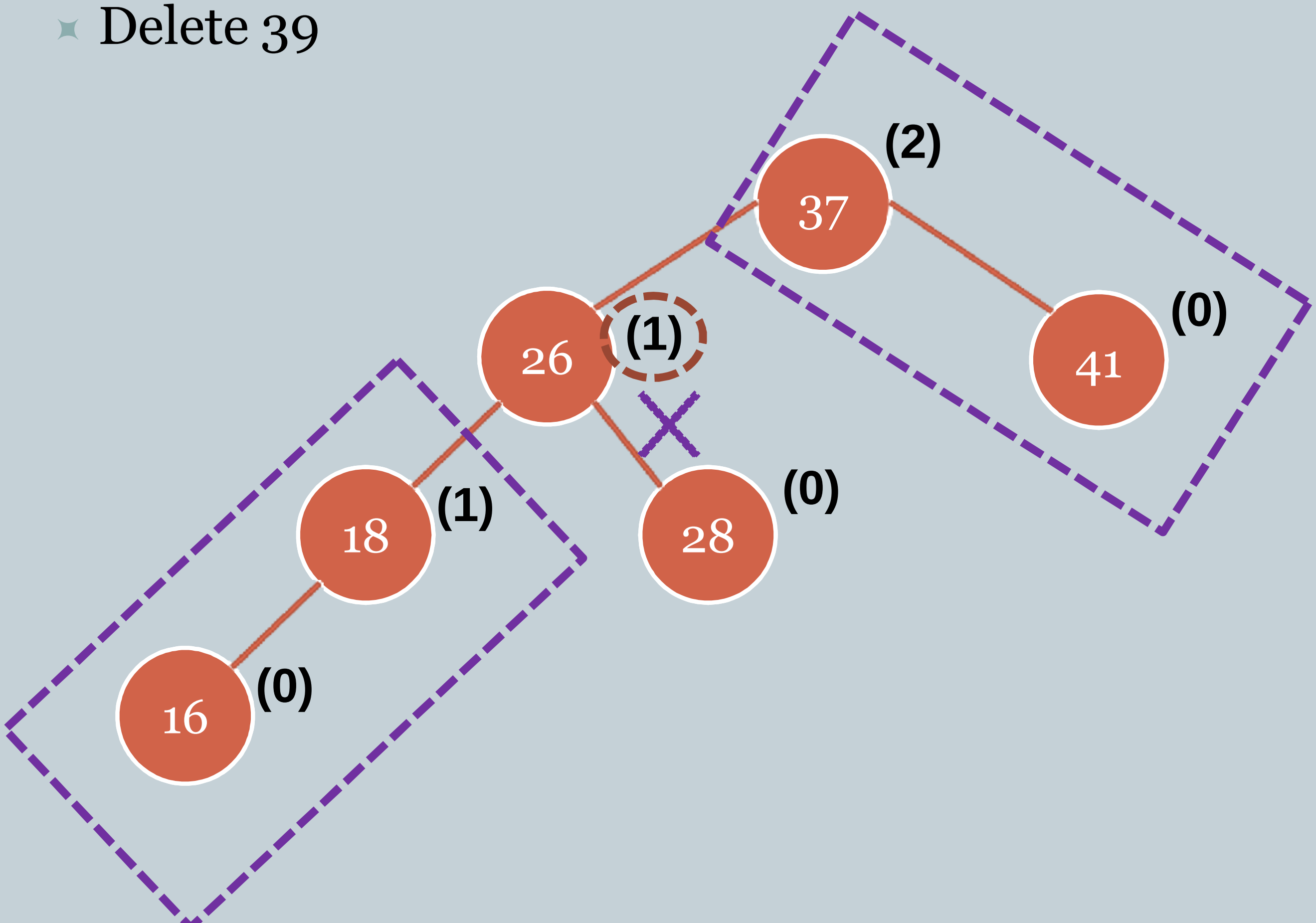
- R1 Rotation
  - If BF(B) = 1, the R1 Rotation is executed
  - Example:
    - Initial AVL Tree



[Balanced AVL Tree]

R1 Rotation

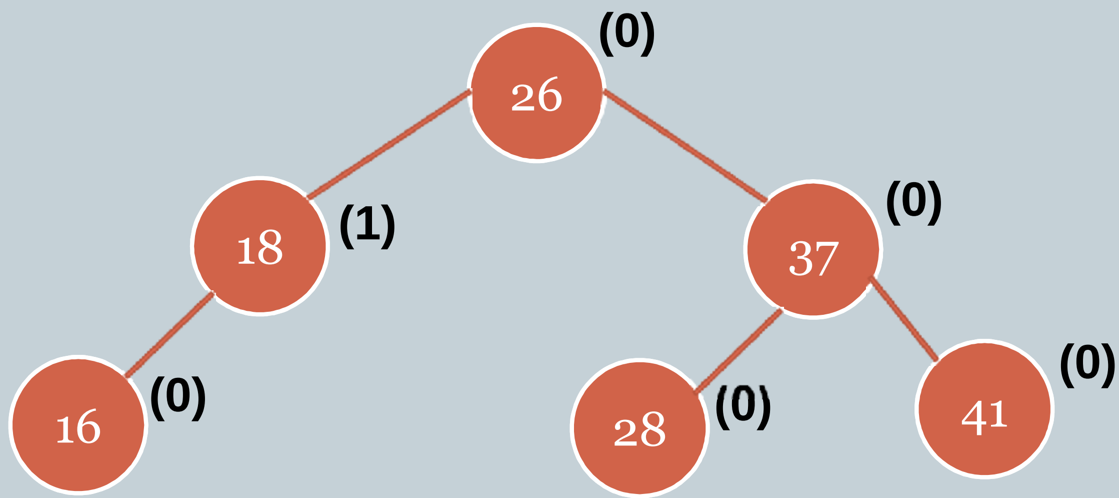
- Delete 39



[Unbalanced AVL Tree]

R1 Rotation

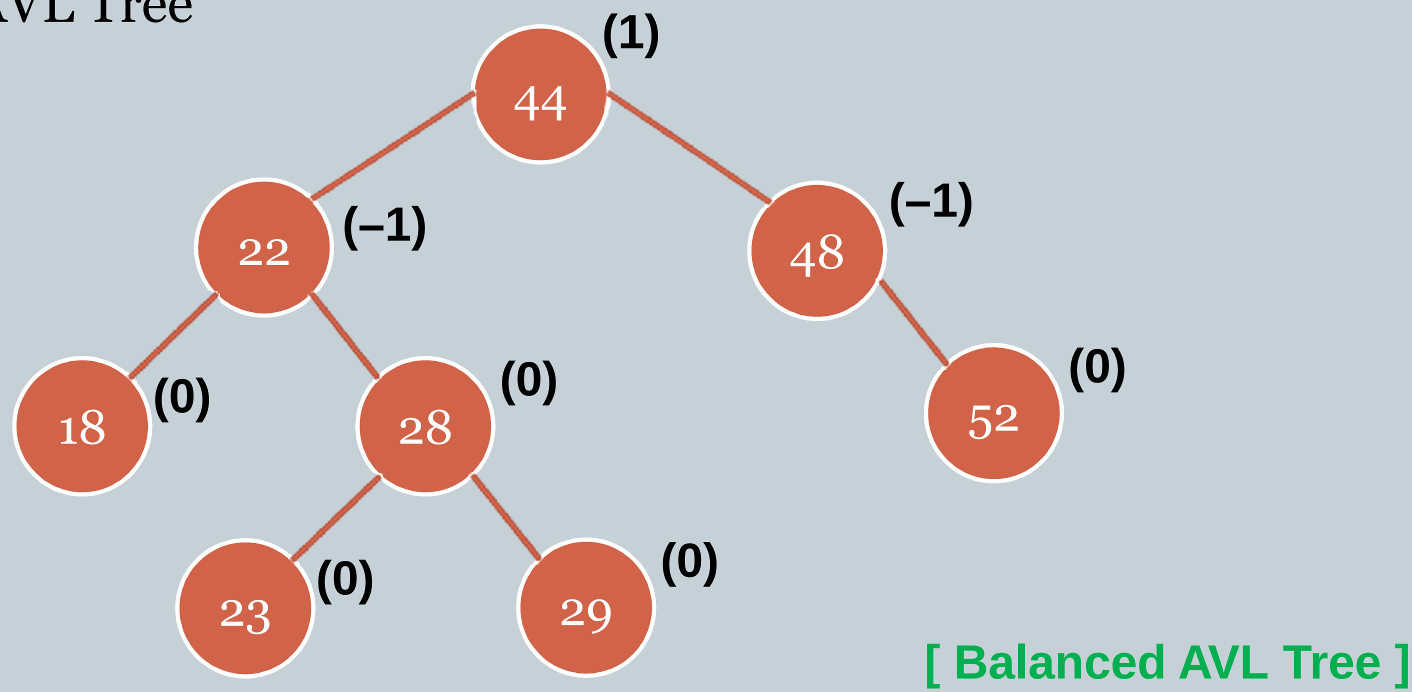
- After R1 Rotation



[Balanced AVL Tree]

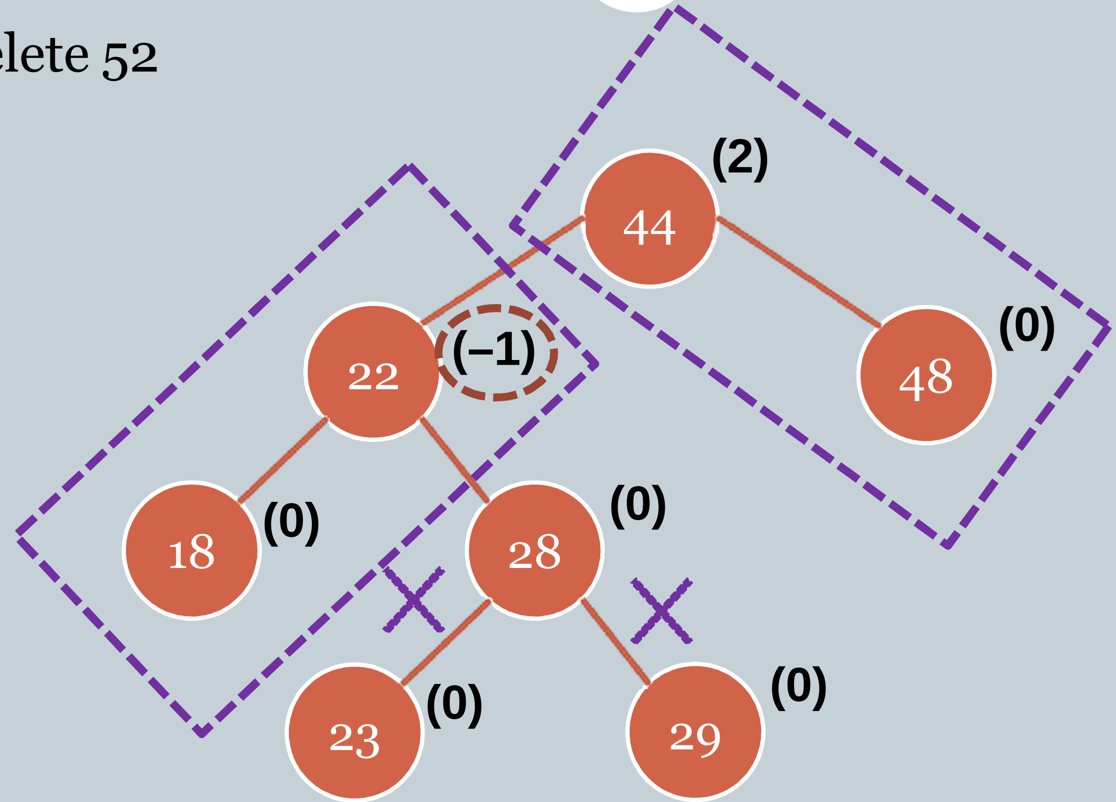
R-1 Rotation

- R-1 Rotation
  - If BF(B) = -1, the R-1 Rotation is executed
  - Example:
    - Initial AVL Tree



R-1 Rotation

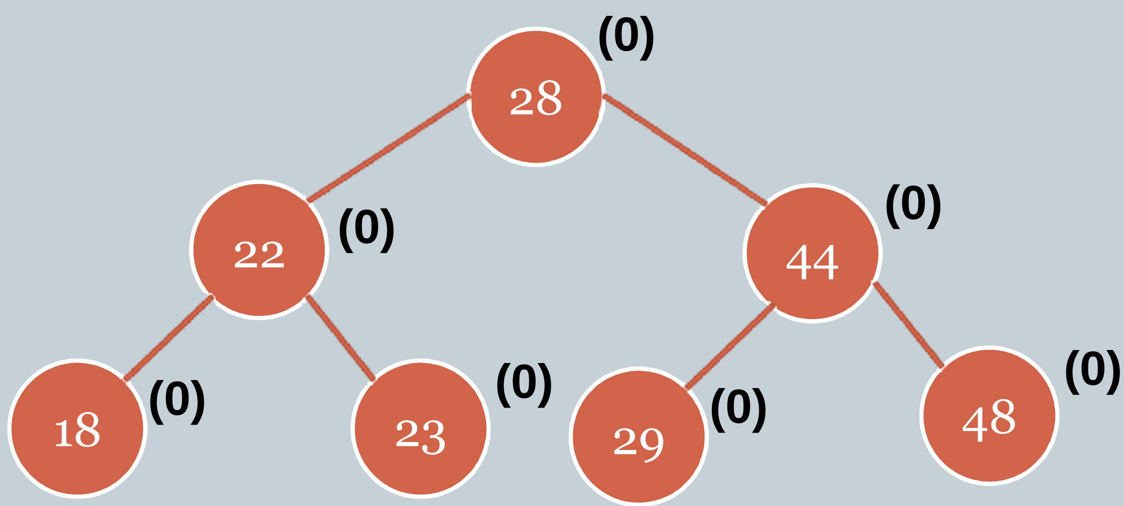
- Delete 52



[Unbalanced AVL Tree]

R-1 Rotation

- After R-1 Rotation



[Balanced AVL Tree]

m-Way Trees

m-Way Tree

- Data Structure Stored in Internal Memory
  - All Data Structure described till, stored in Internal Memory & Support Internal Information Retrieval
- Data Structure Stored in External Memory
  - Favor Retrieval & Manipulation of Data Stored in External Memory (i.e. Disks)
  - E.g. m-Way Tree, B Tree, B^+ Tree

m-Way Tree

- m-Way Tree
  - Generalized version of Binary Search Tree (BST)
  - Goal is to Minimize Accesses while retrieving Key from a File
  - Height h calls fro O(h) Number of Accesses for an Insert / Delete / Retrieval Operation
  - But Height h is close to $\log_m(n + 1)$, because Number of Elements in m-Way Tree of Height h Ranges from a Minimum of h to a Maximum of $m^h – 1$
  - So m-Way Tree of n Elements Ranges from a Minimum Height of $\log_m(n + 1)$ to Maximum Height of n
  - Hence there is need to maintain Balanced m-Way Tree. B-Tree are Balanced m-Way Tree

m-Way Tree

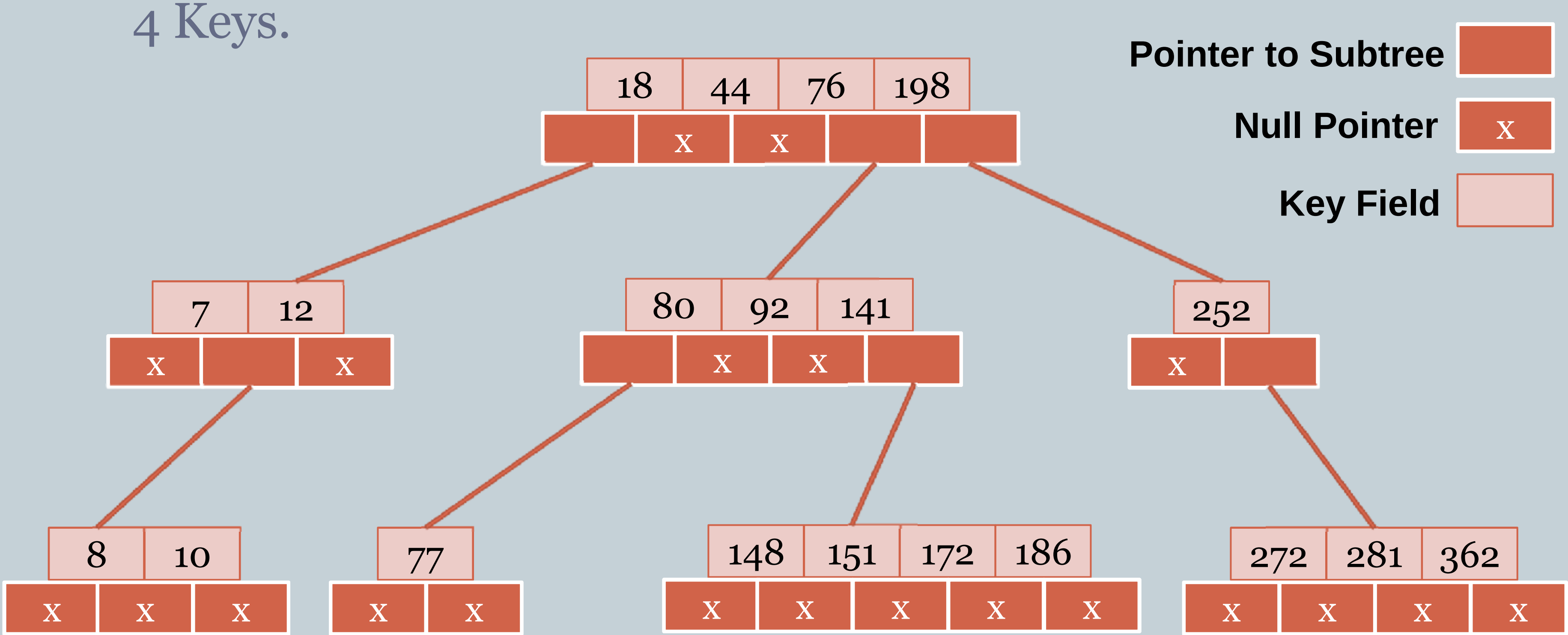
- Definition
  - An m-Way Tree T may be an Empty Tree
  - If T is Not Empty, it Satisfies following Properties:
    1. For some integer m, known as **Order of Tree**, each Node is of Degree which can reach a Maximum of m
       - Each Node has at most m Child Nodes
       - A Node is represented as $A_0, (K_1, A_1), (K_2, A_2) \dots (K_{m-1}, A_{m-1})$ where, K_i, $1 <= i <= m - 1$ are the Keys and A_i, $0 <= i <= m - 1$ are Pointers to Subtree of T
    2. If a Node has k Child Nodes where k <= m, then Node can have only (k – 1) Keys K1, K2, . . . , K_{k-1}, contained in Node such that $K_i < K_{i+1}$ and Each of the Keys Partitions all the Keys in Subtree into k Subsets

m-Way Tree

3. For a Node A_0, (K_1, A_1), (K_2, A_2) . . . (K_{m-1}, A_{m-1}), all Key Values in Subtree pointed to by A_i are less than the Key K_{i+1}, $0 <= i <= m - 2$, and all Key Values in Subtree pointed to by A_{m-1} are Greater than K_{m-1}
4. Each of Subtree in A_i, $0 <= i <= m - 1$ are also m-Way Trees

m-Way Tree

- Example: 5-Way Tree
  - Each Node has at most 5 Child Nodes and therefore has at most 4 Keys.

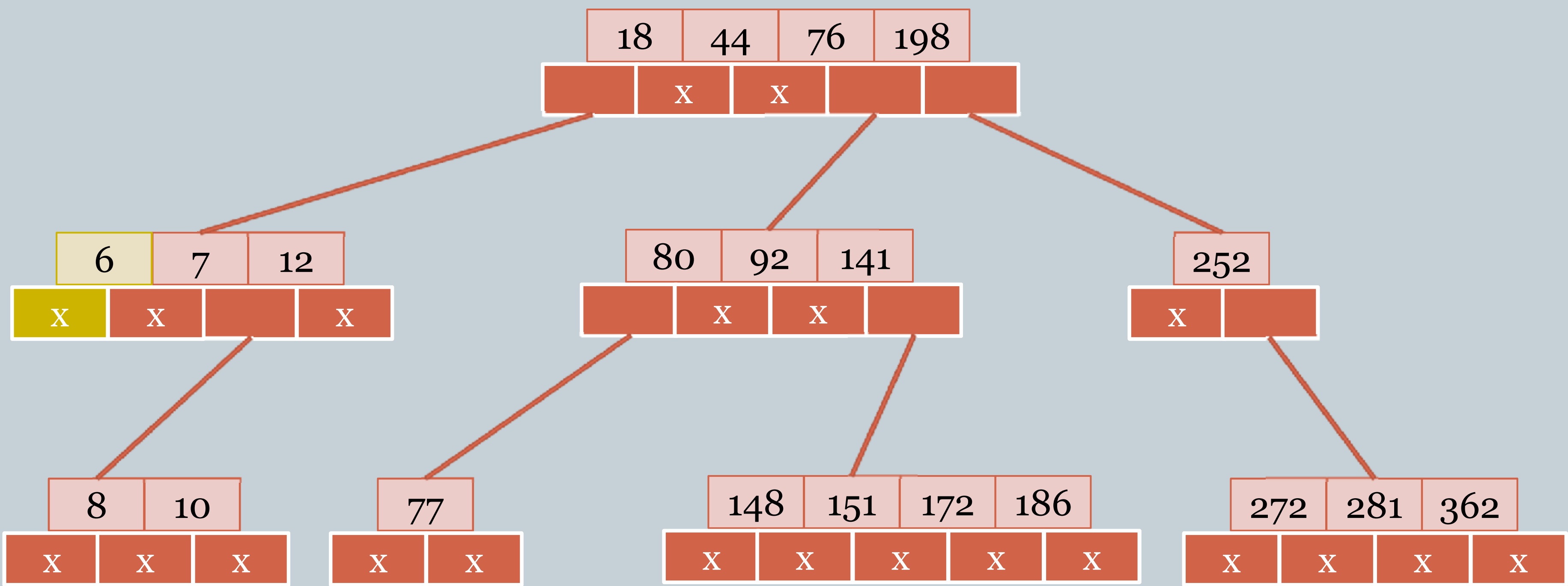


m-Way Tree Operations

- m-Way Tree Operations
  - Searching in m-Way Tree
    - Searching in m-Way Tree is Similar to Binary Search Tree
  - Insertion in m-Way Tree
    - Insertion in m-Way Tree is Similar to Binary Search Tree
      - Search the Location for New Element
      - Insert the Element in Searched Location
  - Deletion in m-Way Tree

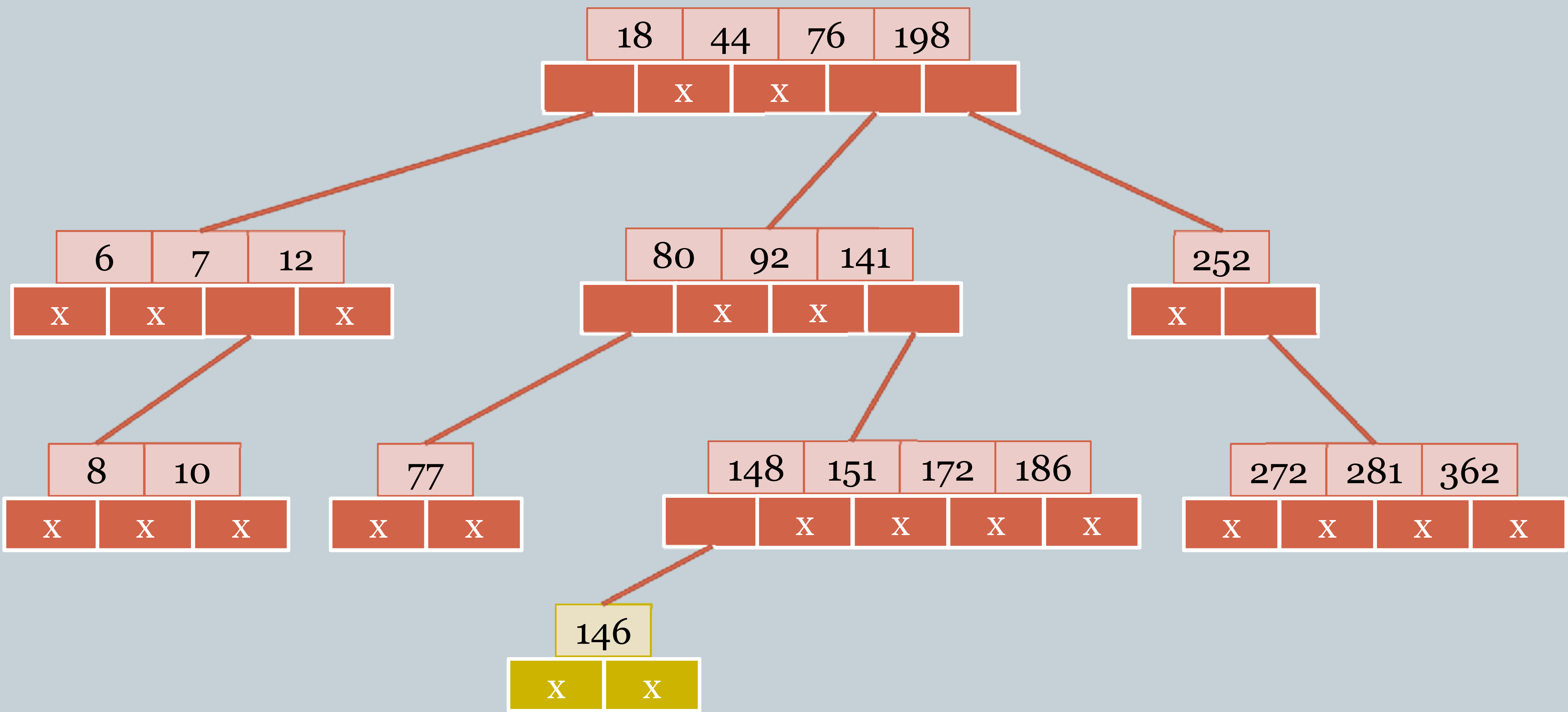
Insertion in m-Way Tree

- Insertion in m-Way Tree
  - Example:
    - Insert 6 in 5-Way Tree



Insertion in m-Way Tree

- Insert 146 in 5-Way Tree



m-Way Tree Construction

- m-Way Tree Construction
  - 3-Way Tree Constructed out of Empty Search Tree with following Keys in the order:
    - D, K, P, V, A, G

B-Tree

B-Tree

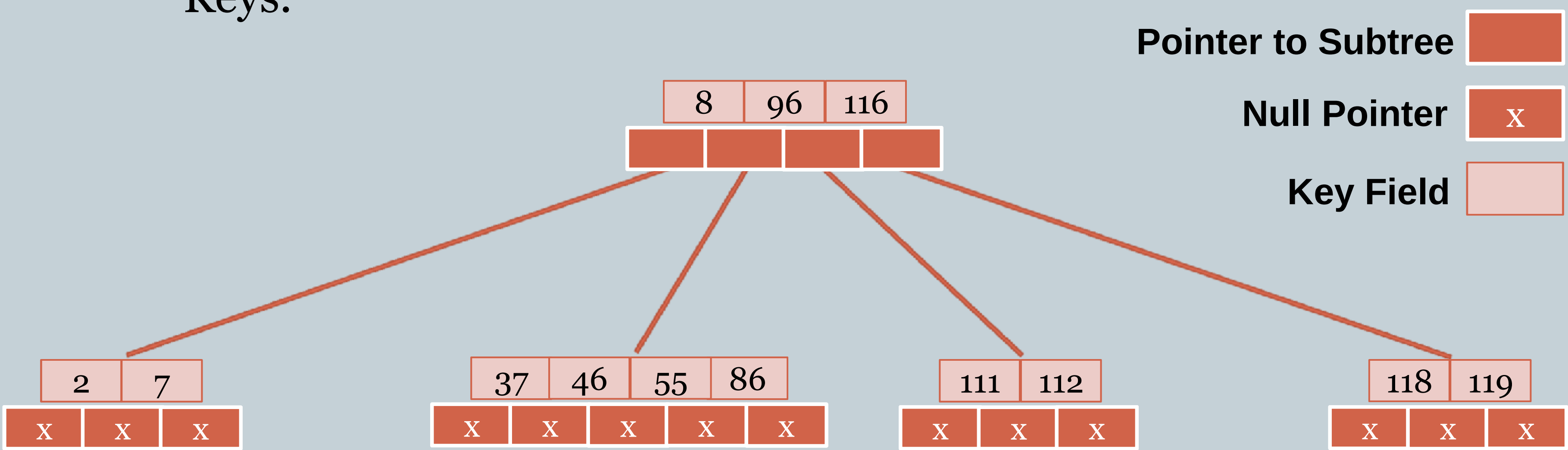
- B-Tree
  - M-Way Tree have Advantage of Minimizing File Accesses due to their Restricted Height
  - Height of Tree should be kept as Low as Possible and maintain Balanced m-Way Tree
  - Balanced m-Way Tree is called B-Tree

B-Tree

- Definition
  - A B-Tree of Order m, is Non Empty, is an m-Way Tree in which:
    - Root has at Least Two Child Nodes and at Most m Child Nodes
    - Internal Nodes except the Root have at Least $\left\lceil \frac{m}{2} \right\rceil$ Child Nodes and at Most m Child Nodes
    - Number of Keys in each Internal Node is One Less than the Number of Child Nodes and these Keys Partition the Keys in Subtrees of Node in a manner to that of m-Way TreeType equation here.
    - All Leaf Nodes are on Same Level

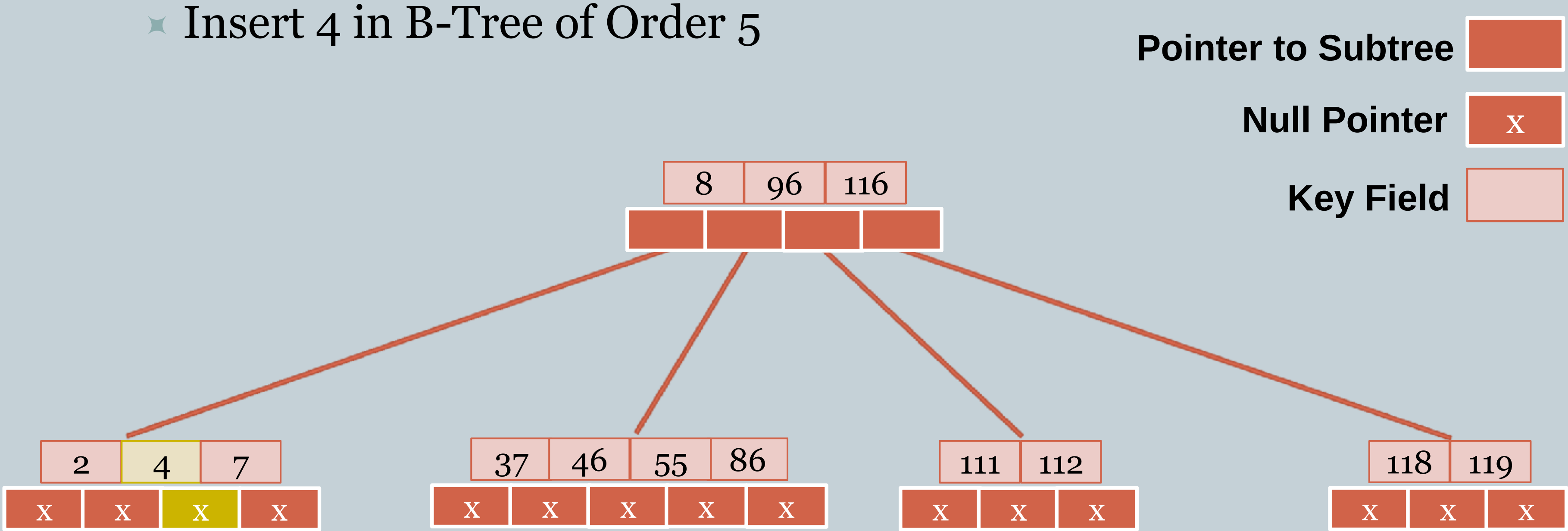
B-Tree of Order 5

- B-Tree of Order 5
  - Example
    - Each Node has at most 5 Child Nodes and therefore has at most 4 Keys.



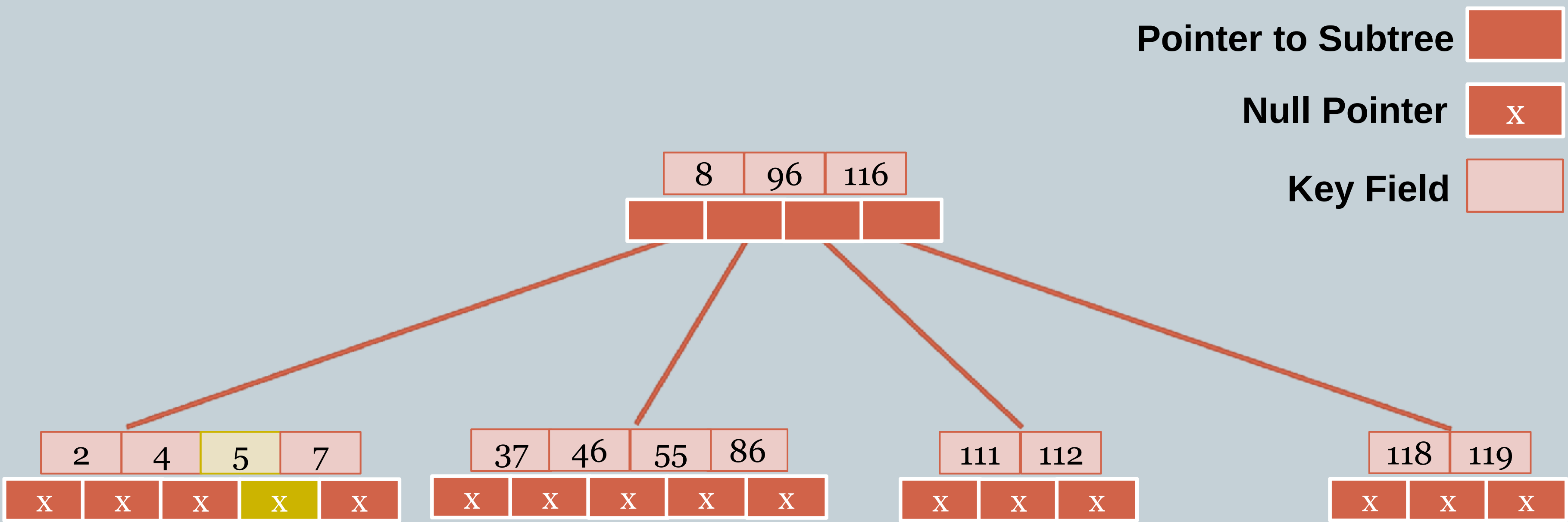
Insertion in B-Tree

- Insertion in B-Tree
  - Example:
    - Insert 4 in B-Tree of Order 5



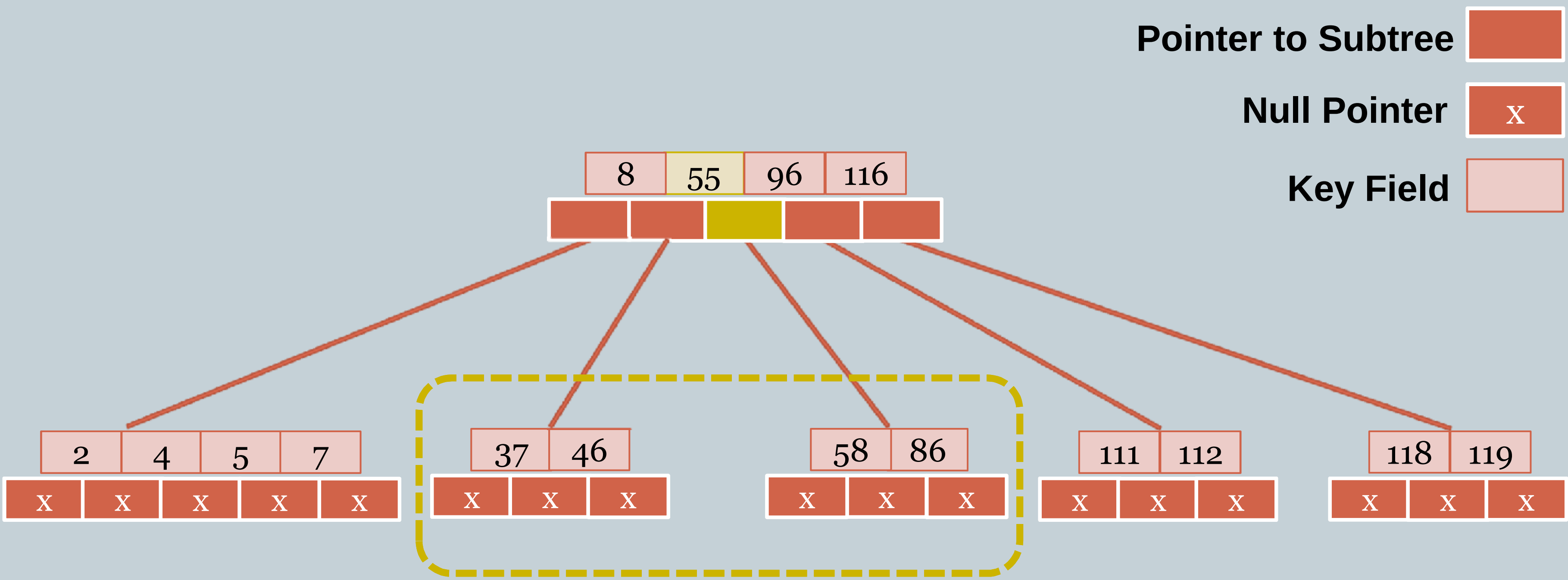
Insertion in B-Tree

- Insert 5 in B-Tree of Order 5



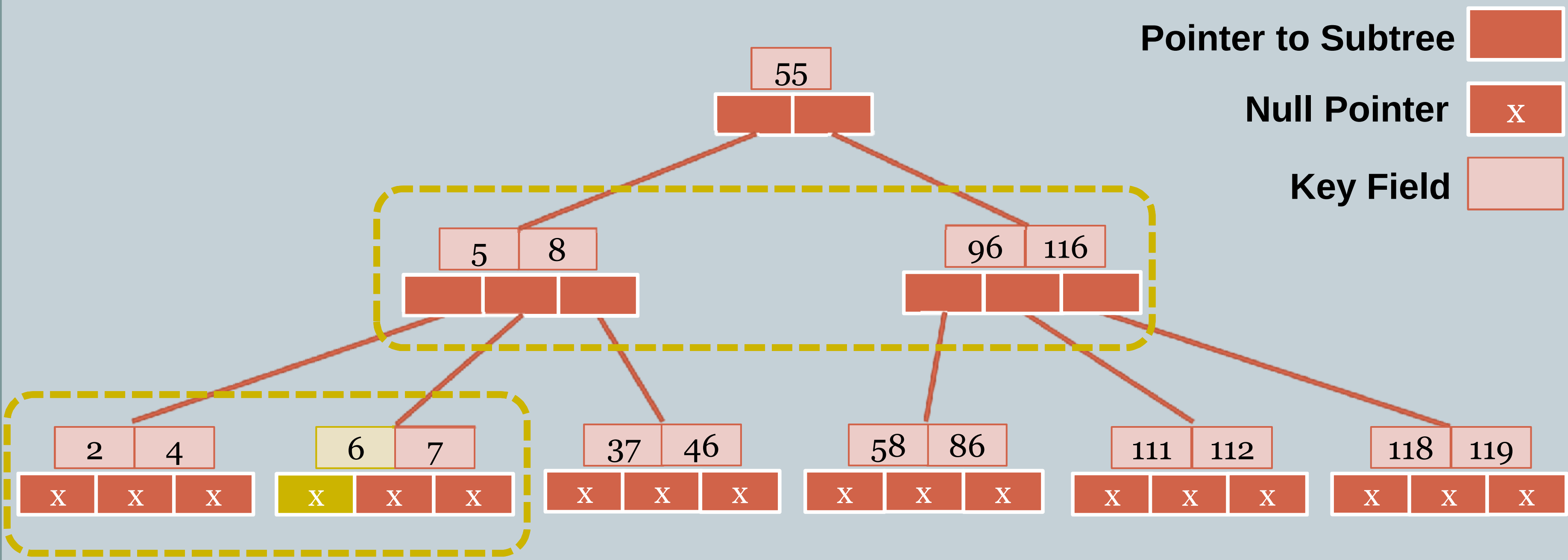
Insertion in B-Tree

- Insert 58 in B-Tree of Order 5



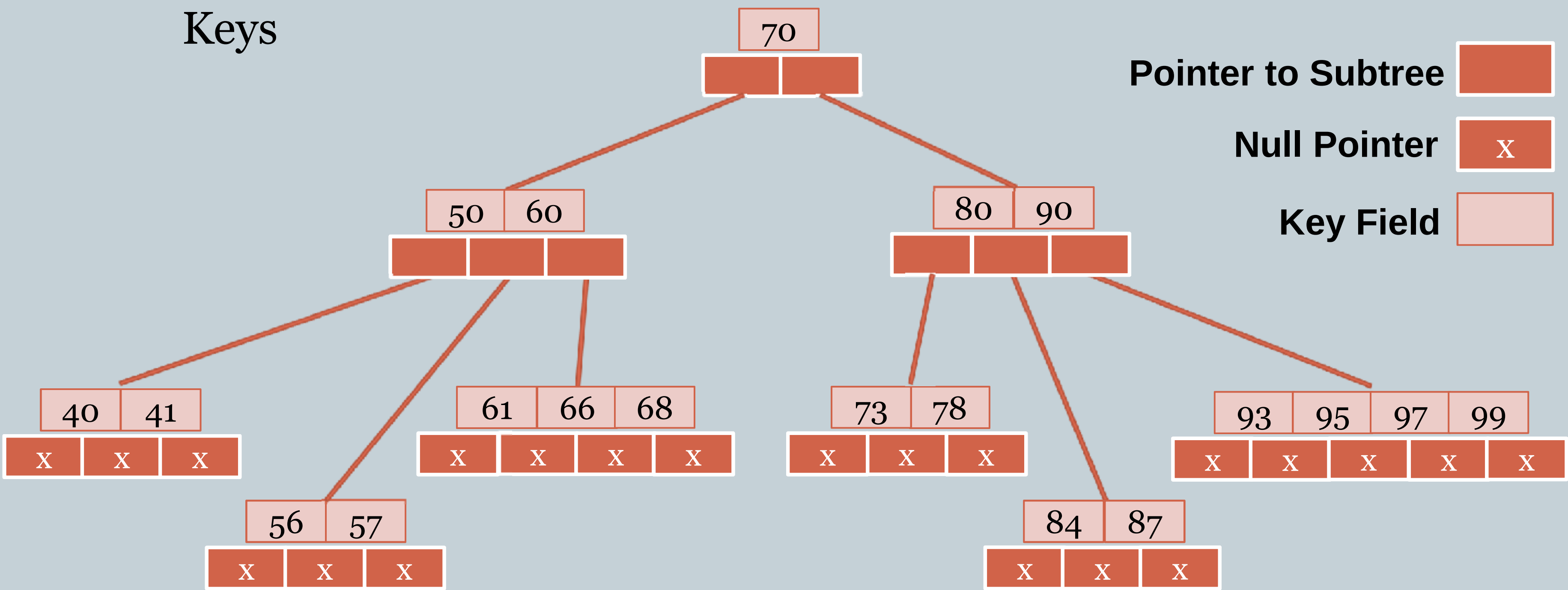
Insertion in B-Tree

- Insert 6 in B-Tree of Order 5



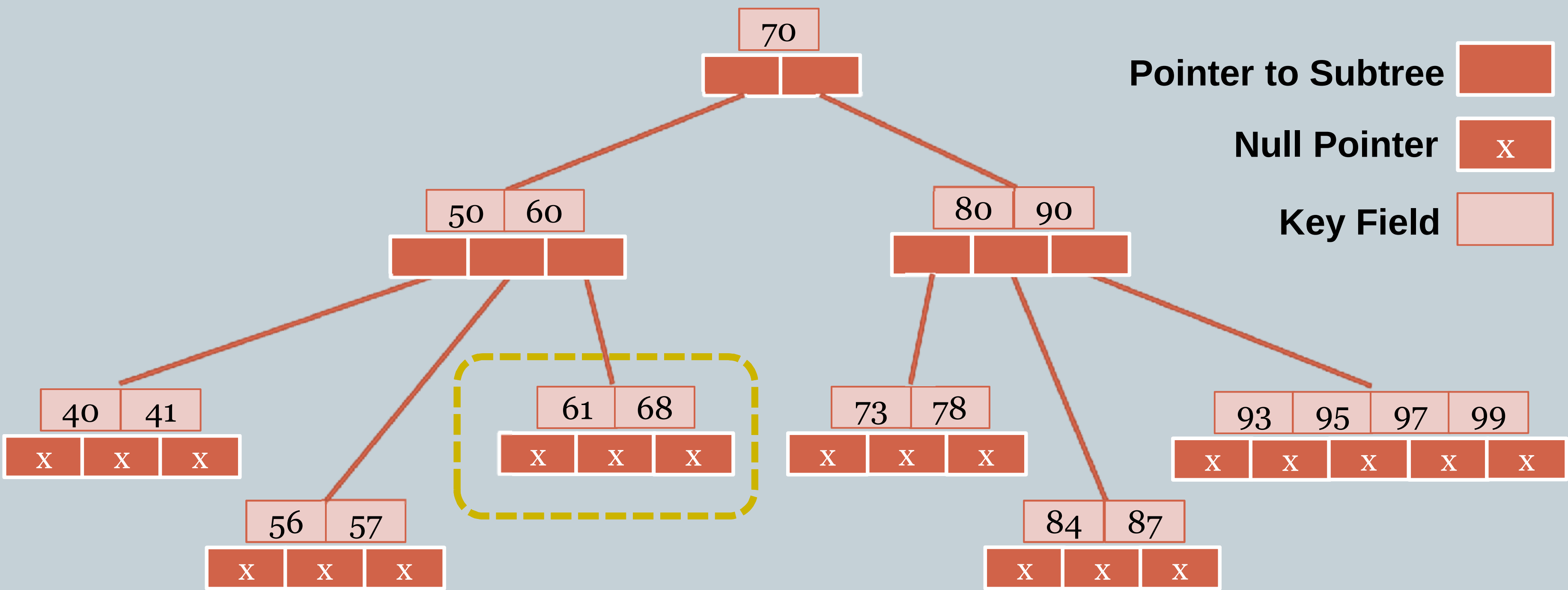
B-Tree of Order 5

- B-Tree of Order 5
  - Example:
    - Each Node has at most 5 Child Nodes and therefore has at most 4 Keys



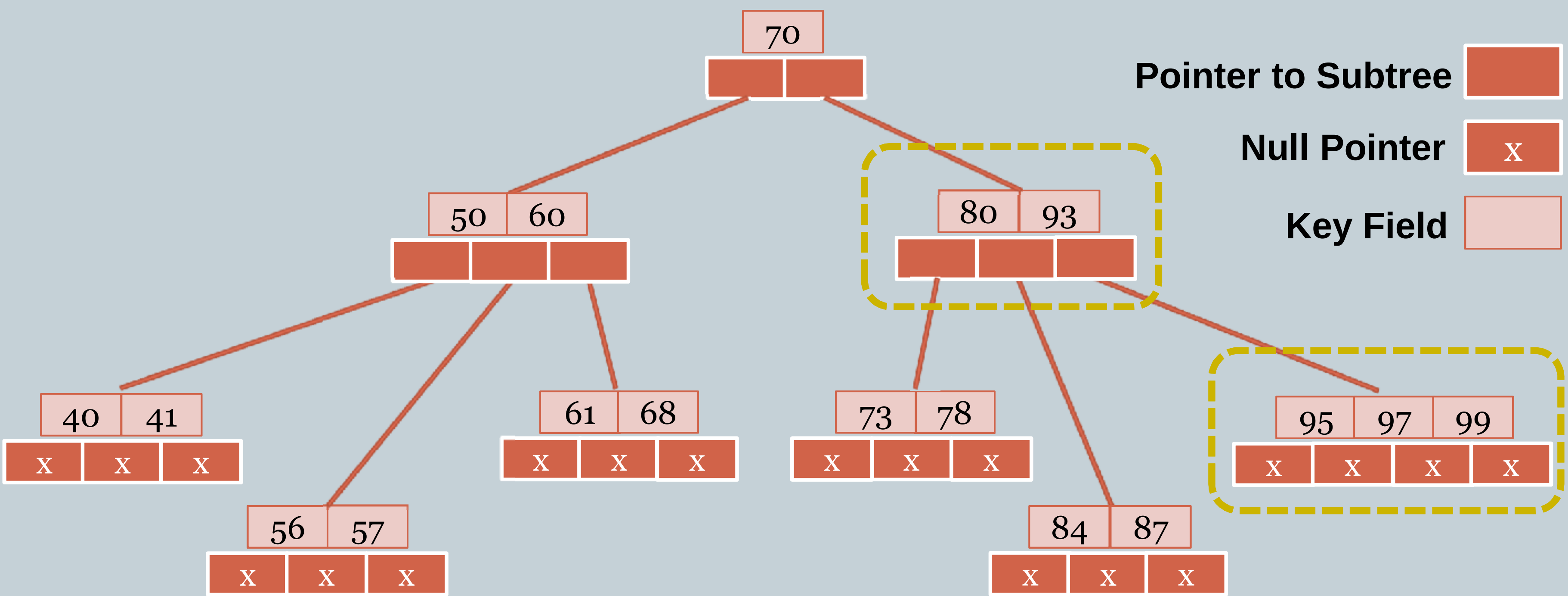
Deletion in B-Tree

- Deletion in B-Tree
  - Example:
    - Delete 66 in B-Tree of Order 5



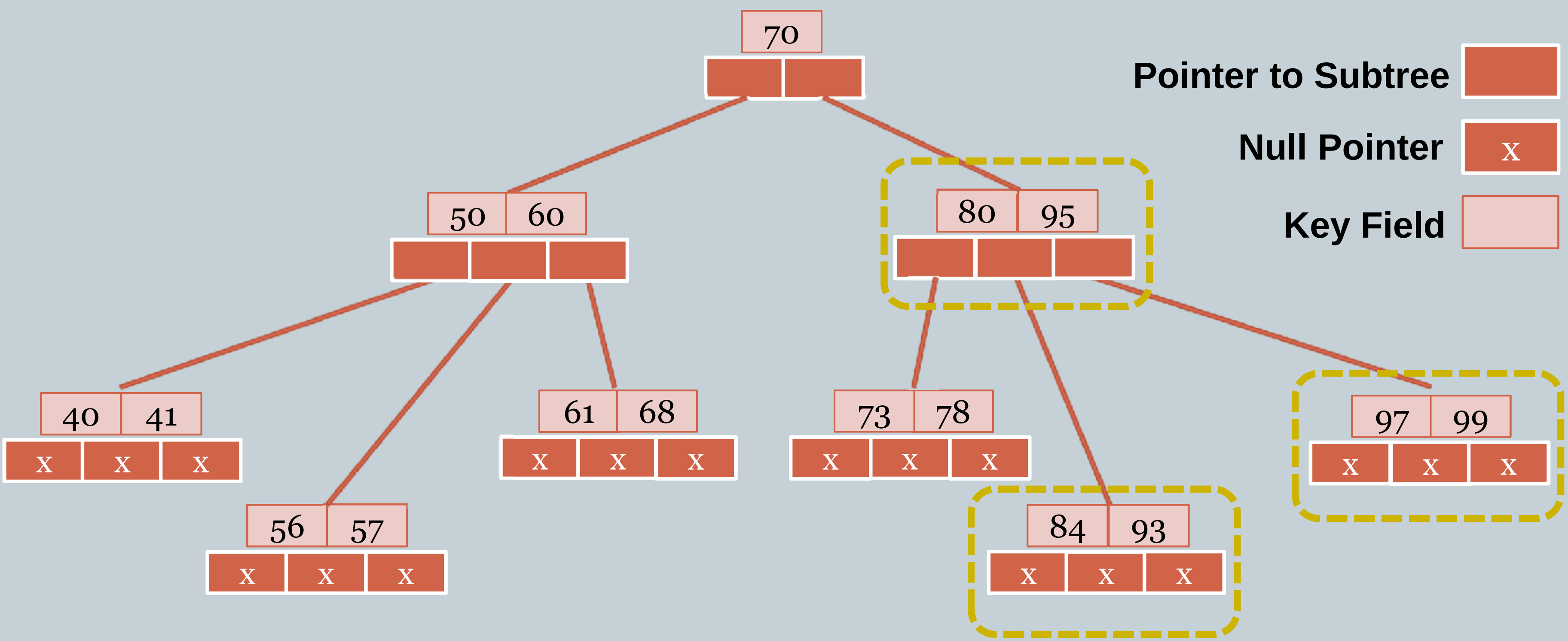
Deletion in B-Tree

- Delete 90 in B-Tree of Order 5



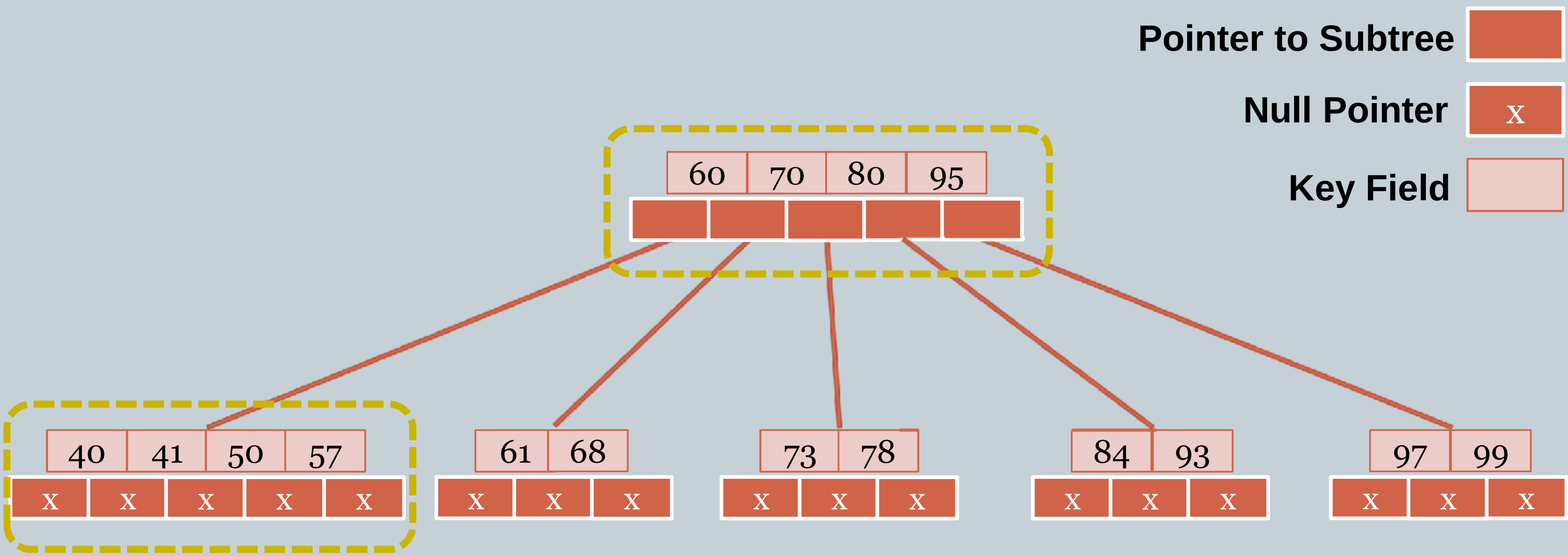
Deletion in B-Tree

- Delete 87 in B-Tree of Order 5



Deletion in B-Tree

- Delete 56 in B-Tree of Order 5



THANKYOU

DOUBTS

PRESENTATION BY:
ANKIT VERMA
(IT DEPARTMENT)