Trees

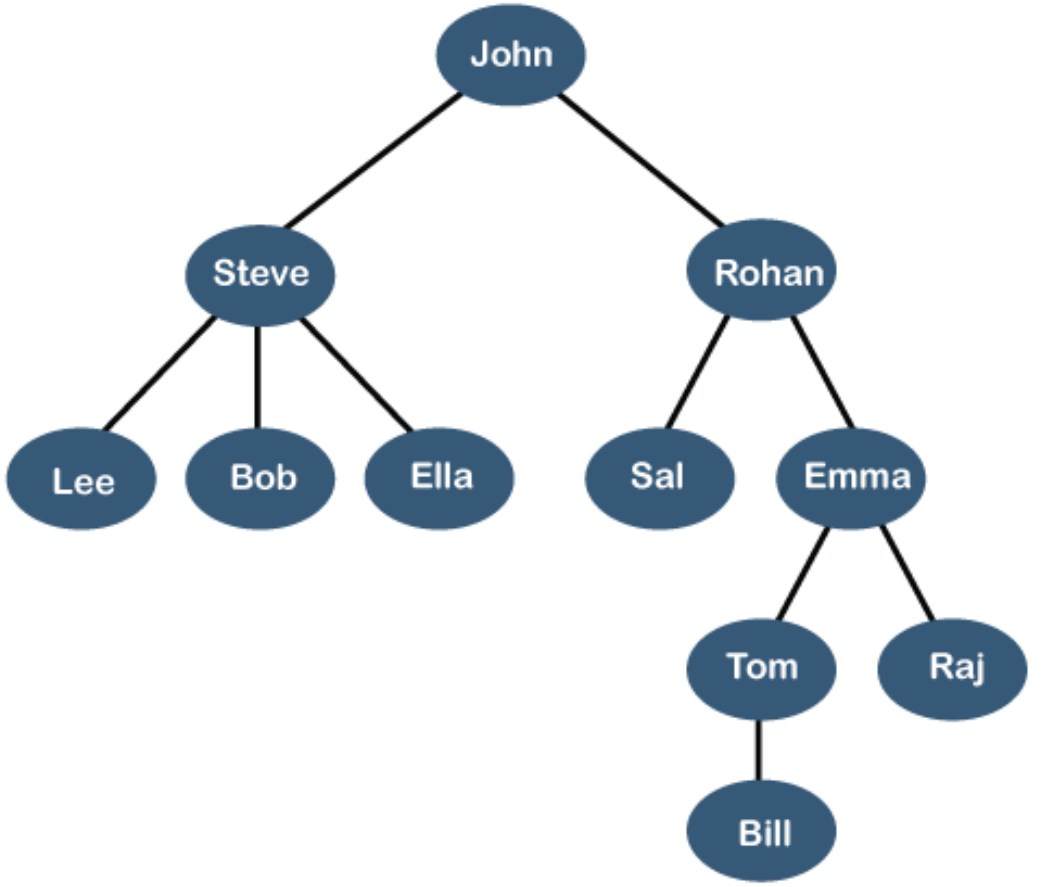
By Asst Prof Christopher Uz

Course: DSAA

Pillai College of Engineering

What is Trees?

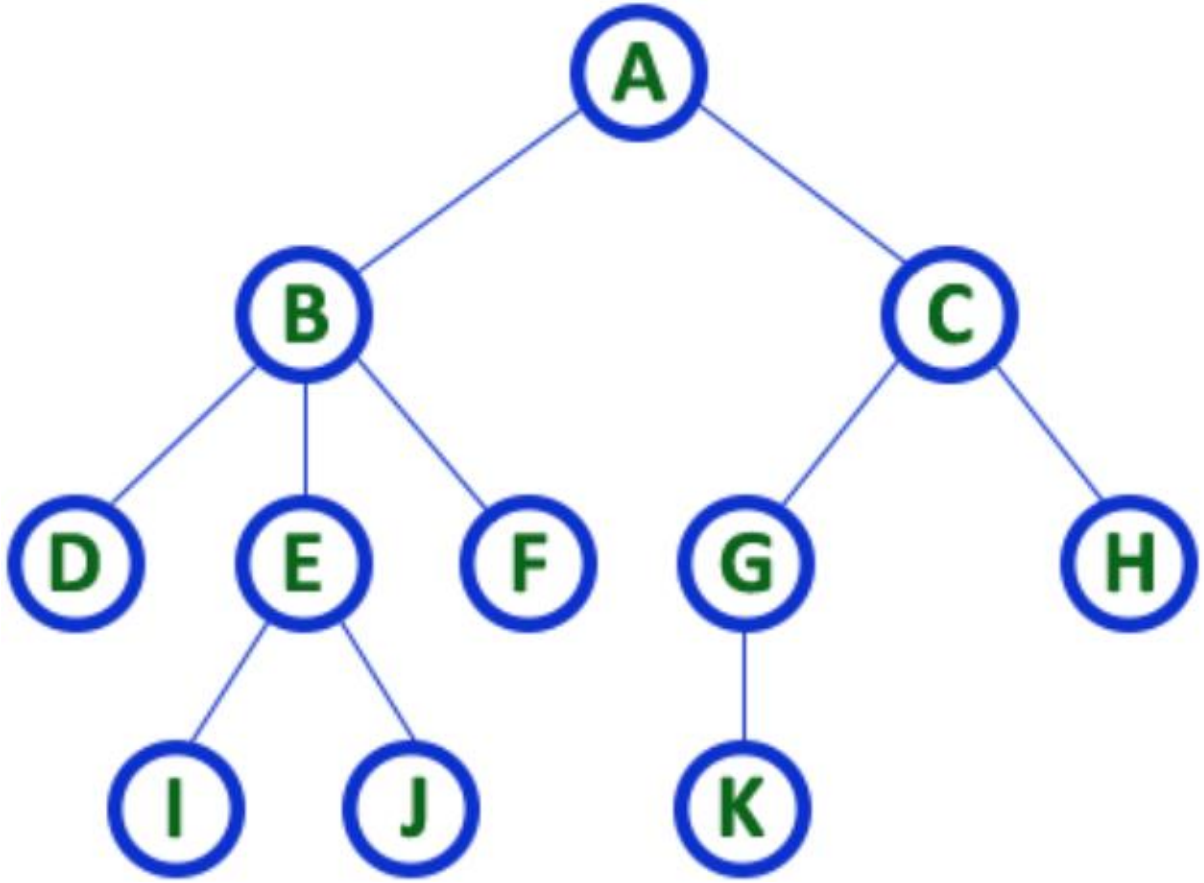
- Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition
- A tree is a connected graph without any circuits
- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges
- E.g., We want to show the employees and their positions in the hierarchical form then it can be represented as



Why Tree Data Structure?

- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially
- In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure

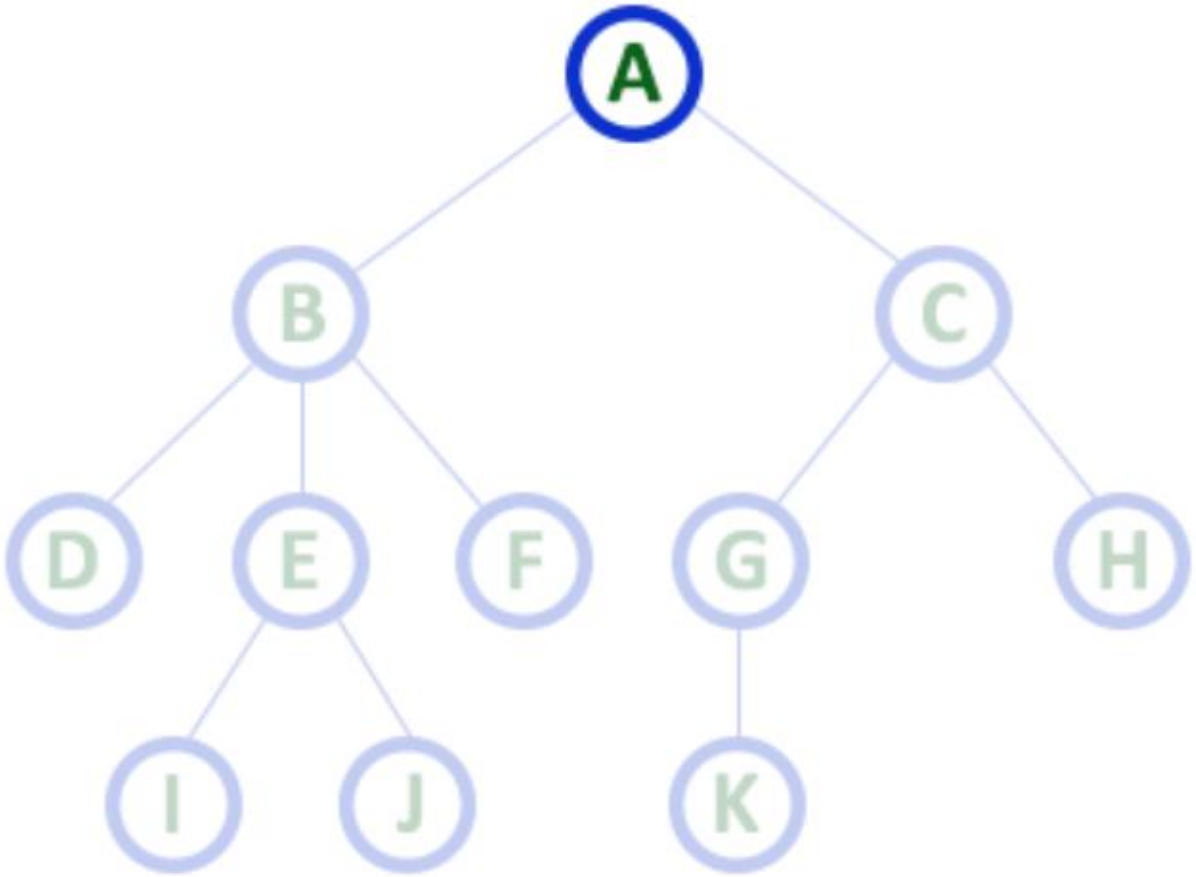
Trees Terminology: Node



TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

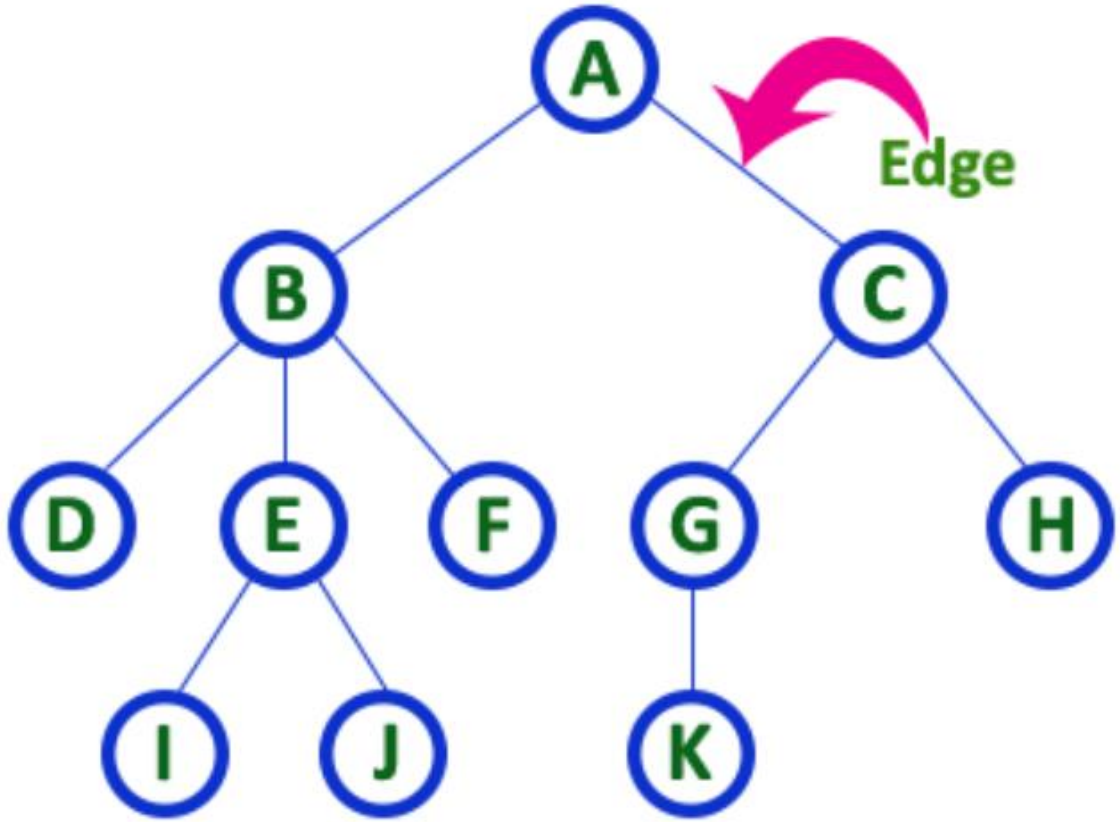
Trees Terminology: Root



Here ‘A’ is the ‘root’ node

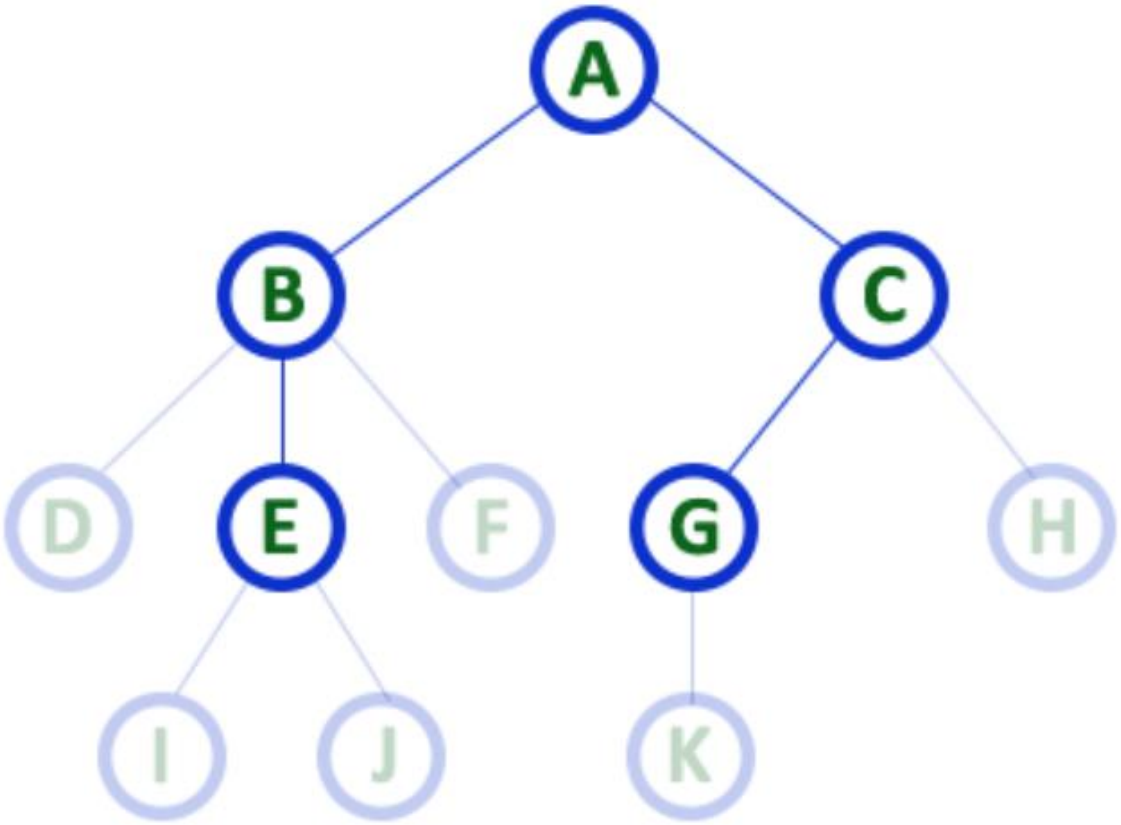
- In any tree the first node is called as ROOT node

Trees Terminology: Edge



- In any tree, 'Edge' is a connecting link between two nodes.

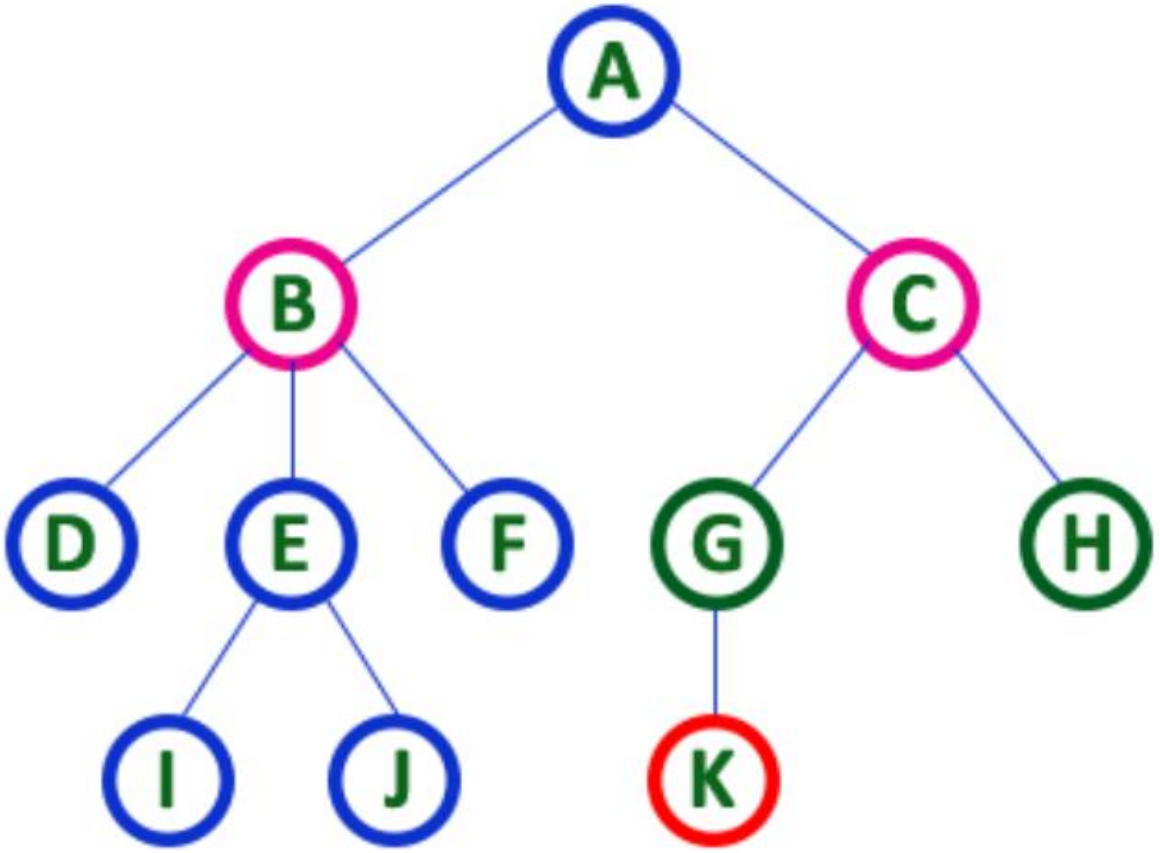
Trees Terminology: Parent



Here A, B, C, E & G are Parent nodes

- **In any tree the node which has child / children is called 'Parent'**
- **A node which is predecessor of any other node is called 'Parent'**

Trees Terminology: Child



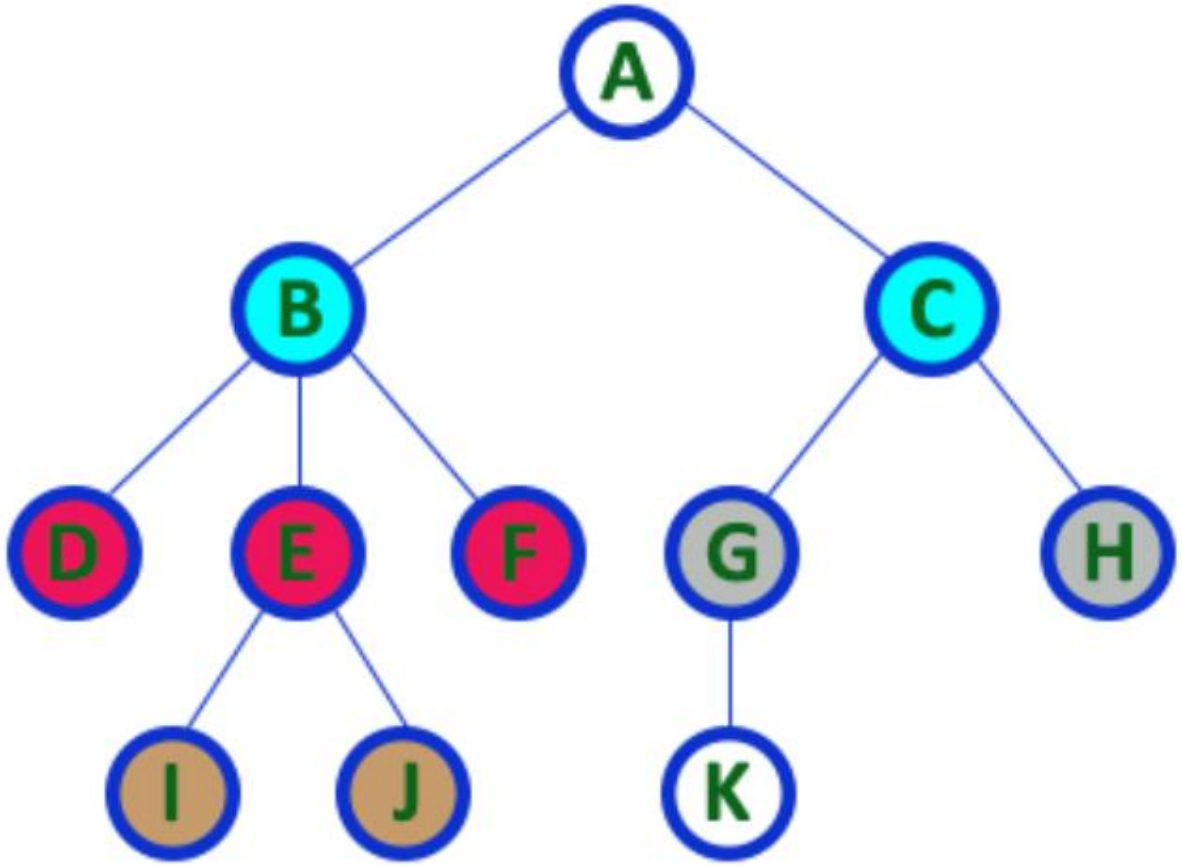
Here B & C are Children of A

Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

Trees Terminology: Siblings



Here B & C are Siblings

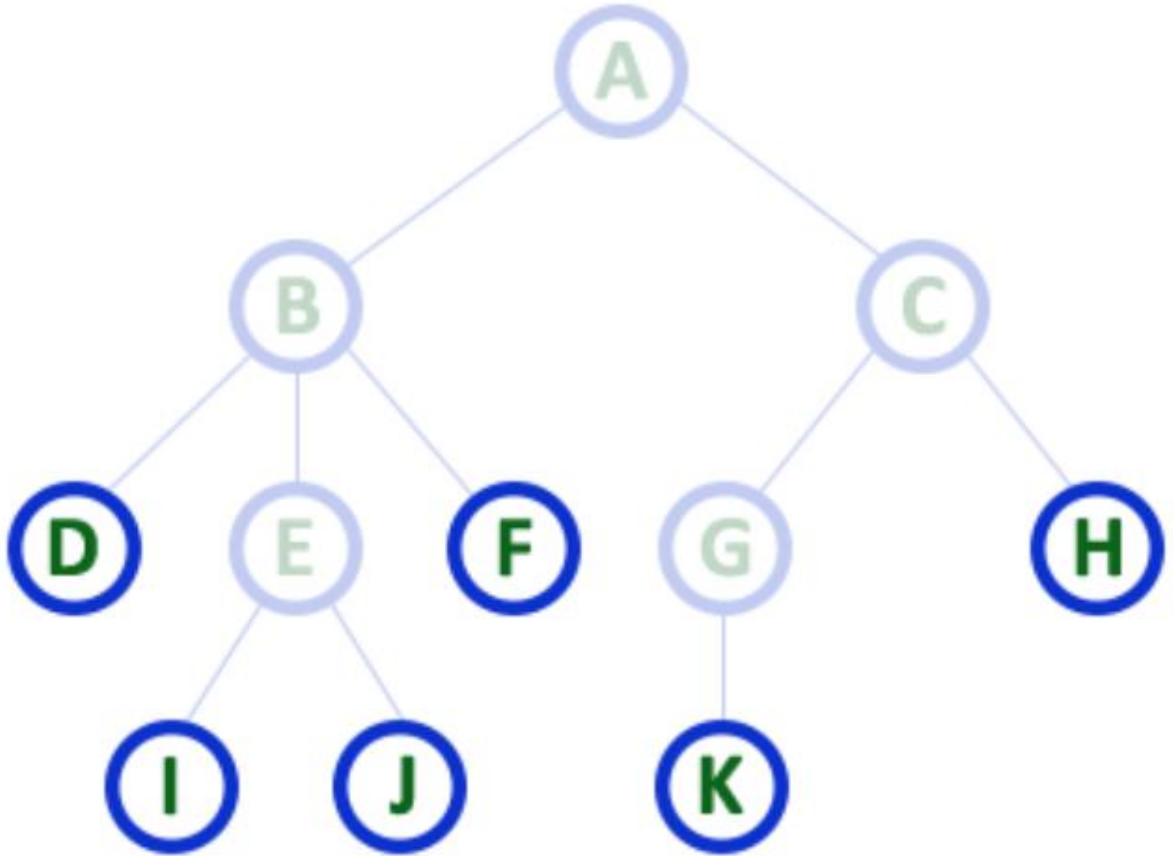
Here D E & F are Siblings

Here G & H are Siblings

Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

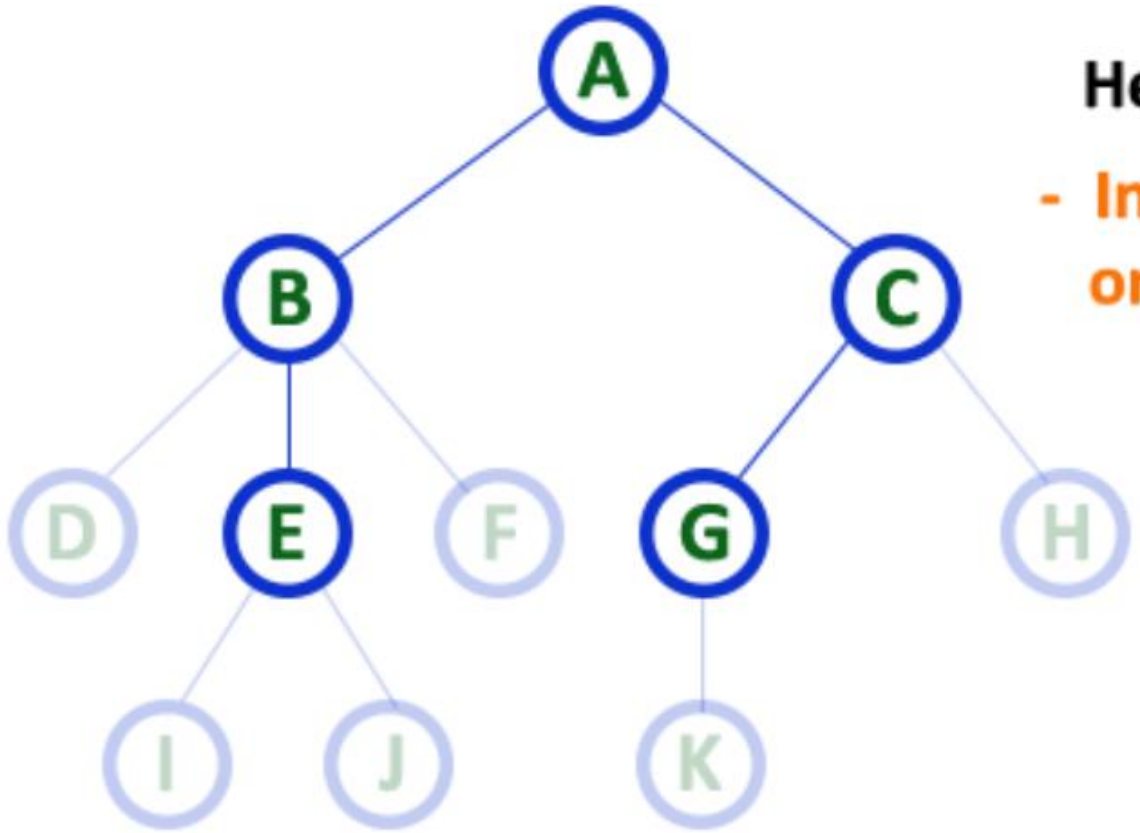
Trees Terminology: Leaf



Here D, I, J, F, K & H are Leaf nodes

- **In any tree the node which does not have children is called 'Leaf'**
- **A node without successors is called a 'leaf' node**

Trees Terminology: Internal Nodes (Non-Terminal Nodes)

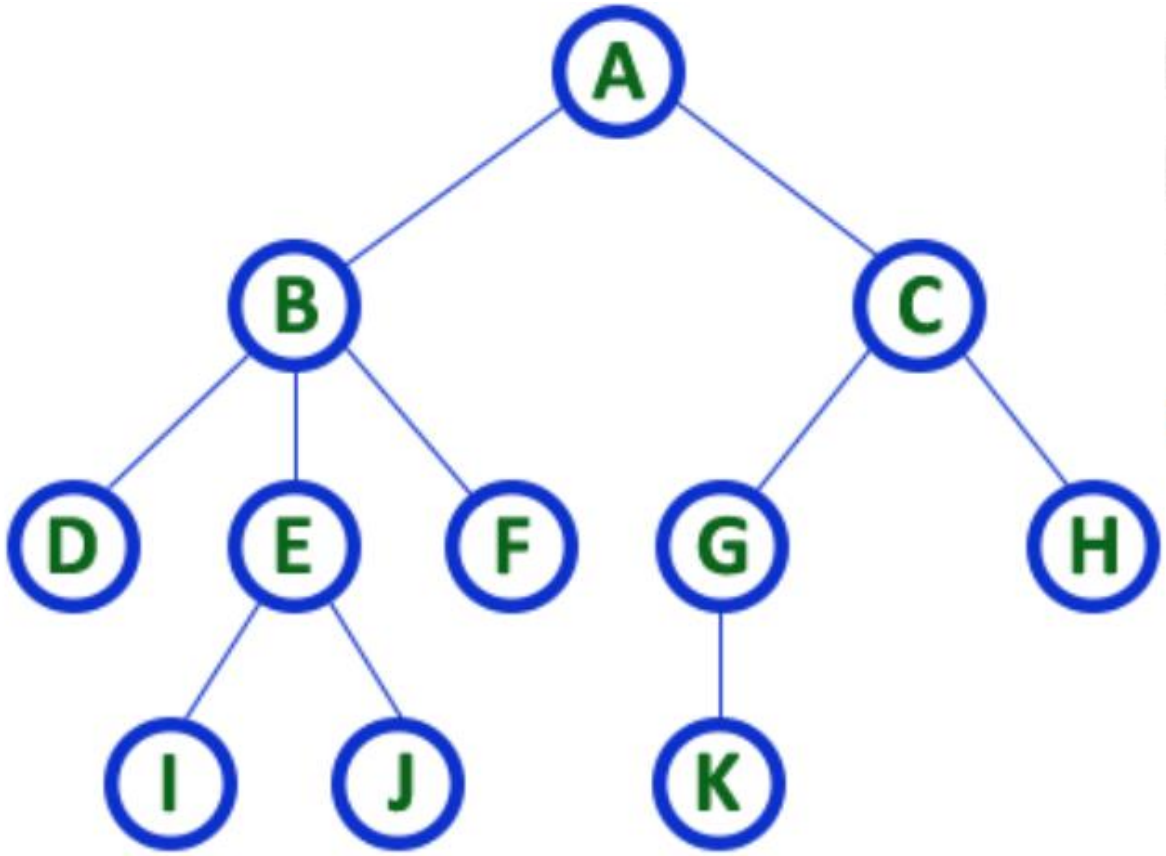


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node

- Every non-leaf node is called as 'Internal' node

Trees Terminology: Degree



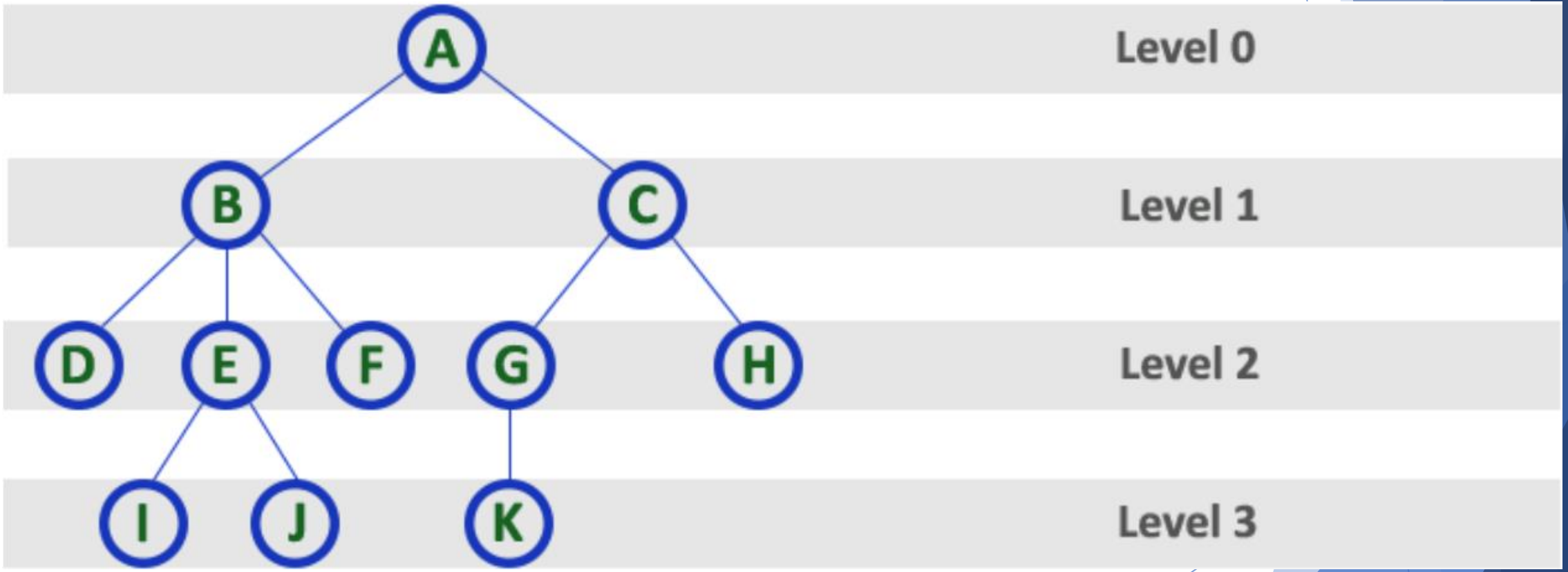
Here Degree of B is 3

Here Degree of A is 2

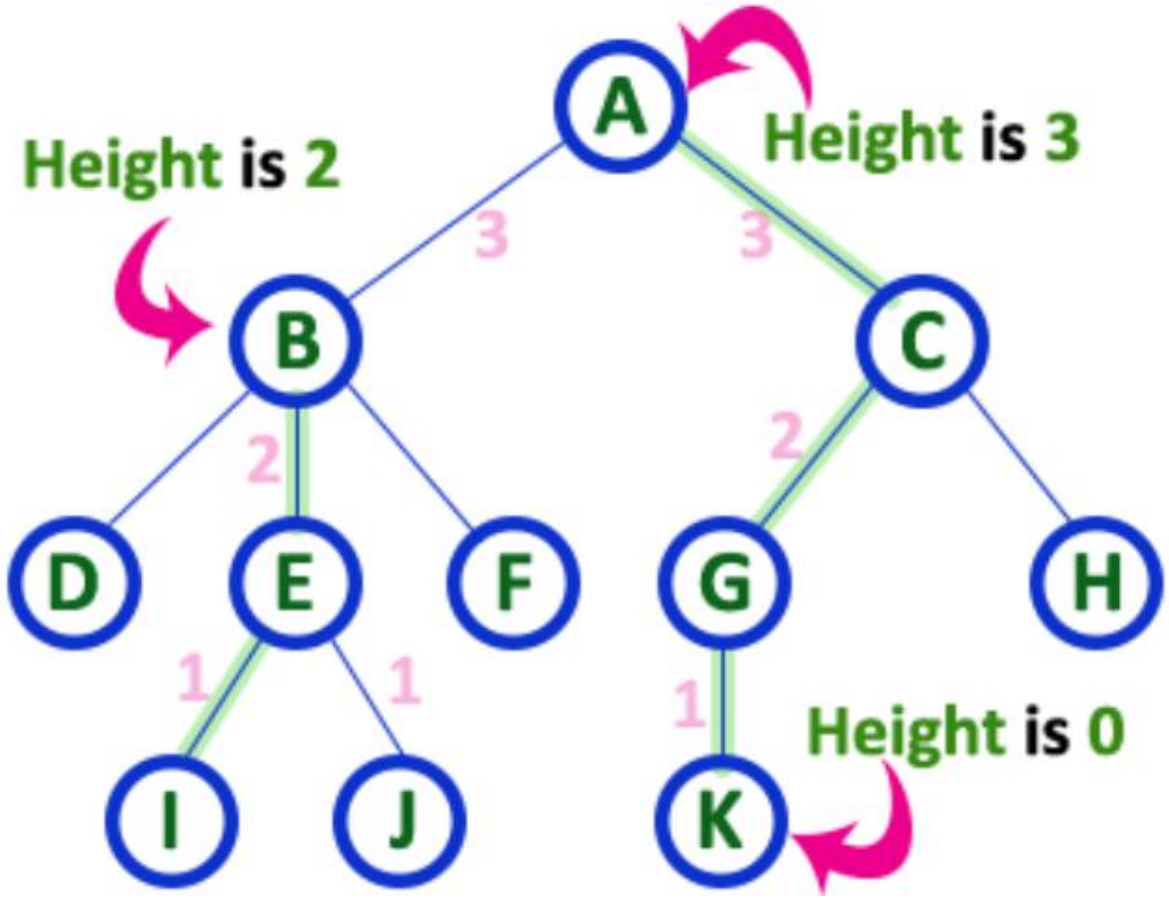
Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

Trees Terminology: Level



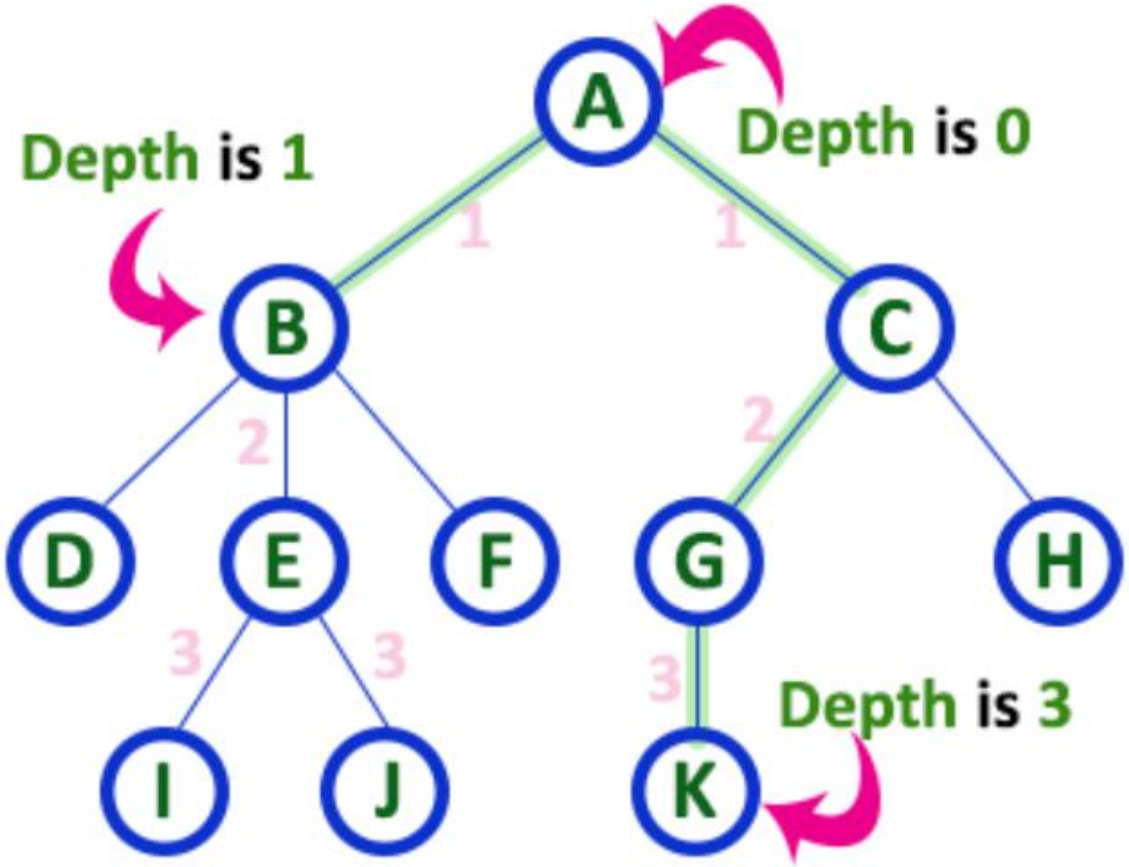
Trees Terminology: Height



Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

Trees Terminology: Depth

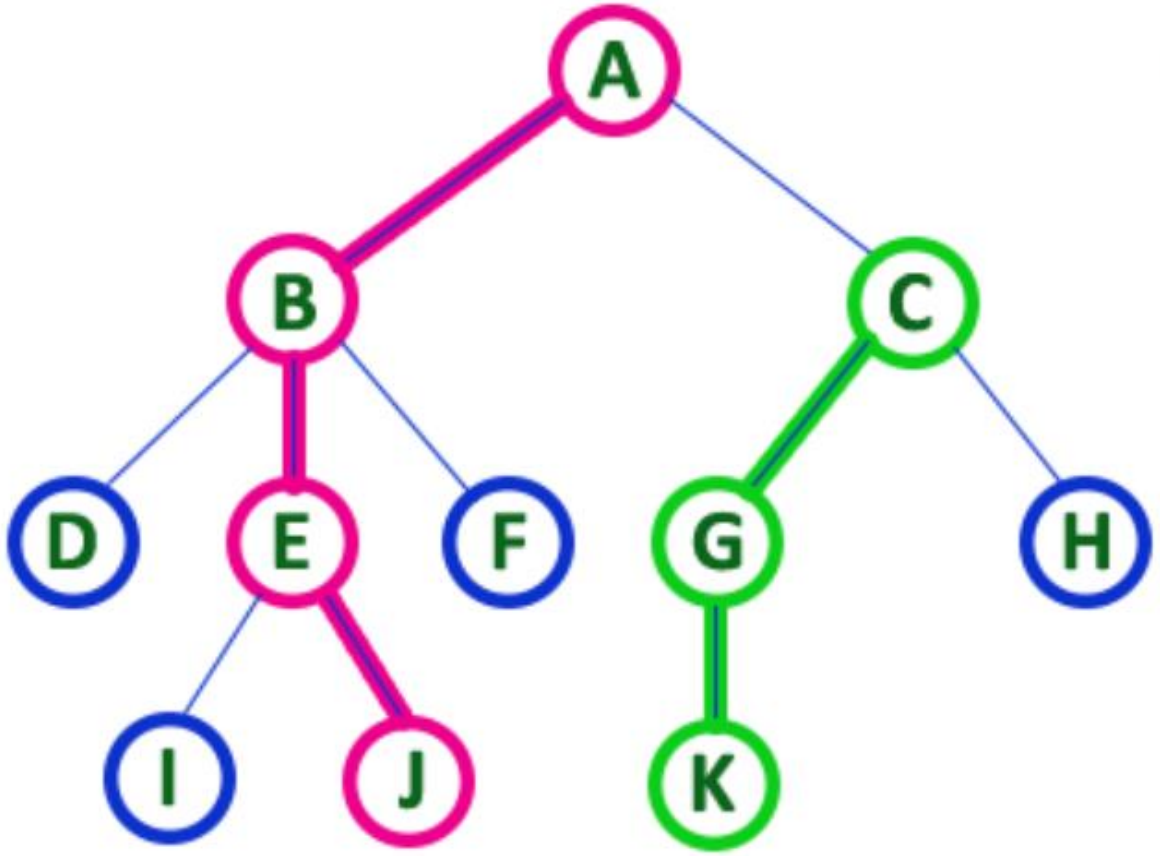


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.

- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

Trees Terminology: Path



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

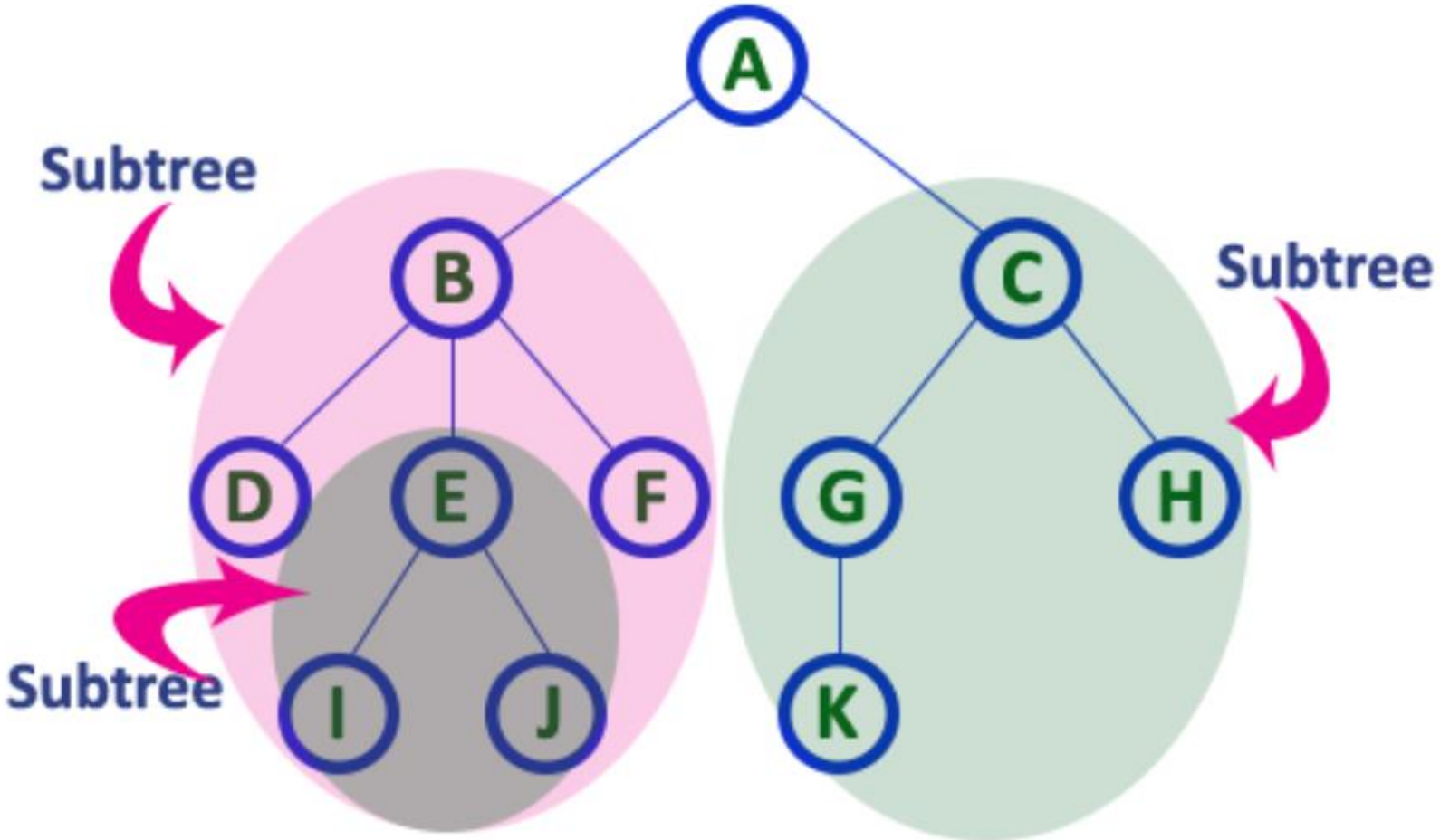
Here, 'Path' between A & J is

A - B - E - J

Here, 'Path' between C & K is

C - G - K

Trees Terminology: Sub Tree

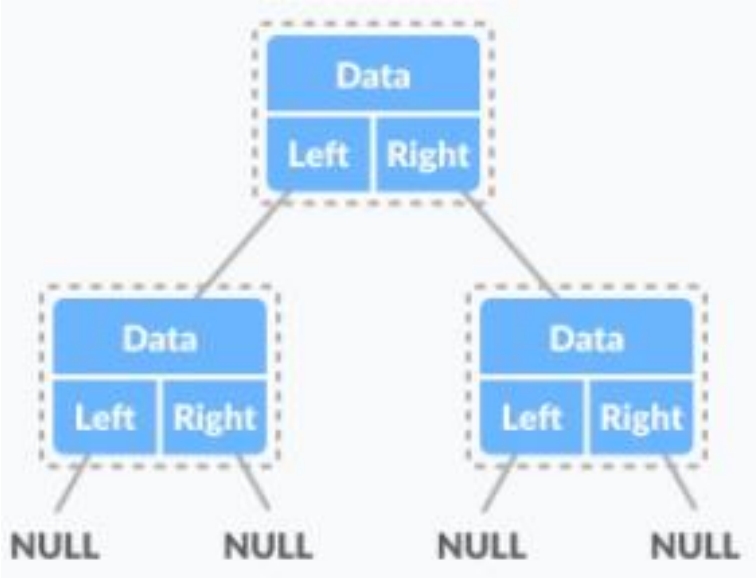


Tree Applications

- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not
- Heap is a kind of tree that is used for heap sort
- A modified version of a tree called Tries is used in modern routers to store routing information
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write

Binary Tree

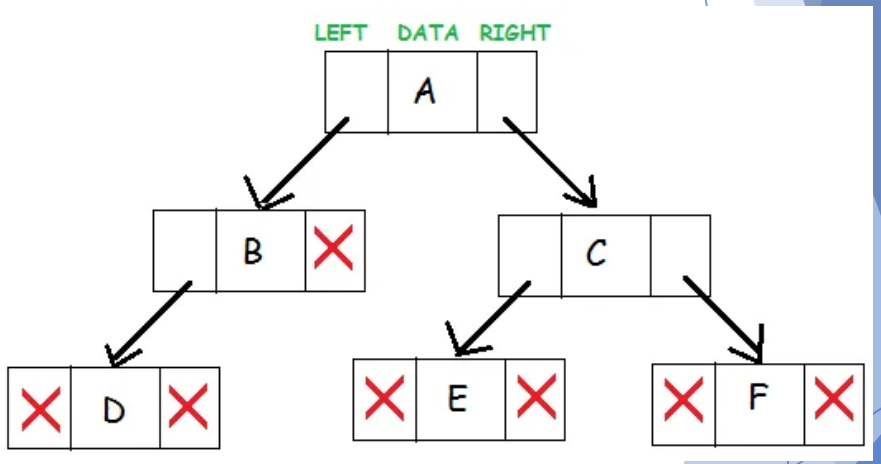
- A binary tree is a special type of tree data structure in which every node can have a maximum of 2 children
- Two children generally referred as left child and right child
- Each node contains three components:
  - Pointer to left subtree
  - Pointer to right subtree
  - Data element
- The topmost node in the tree is called the root
- An empty tree is represented by NULL pointer



Binary Tree Representation

- We use a double linked list to represent a binary tree

- A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type




```
struct node {
        int data;
        struct node *left;
        struct node *right;
};
```

Binary Tree Traversal

- The process of visiting the nodes is known as tree traversal

- There are mainly two types of tree traversals used to visit a node
  1. Depth First Traversal
     - Inorder Traversal
     - Preorder Traversal
     - Postorder Traversal
  2. Breadth First Traversal

Binary Tree Traversal…

- **Depth First Traversal:**
  Depth First Traversal, as the name suggests, is a traversal technique, in which we traverse the tree in a depth first manner. This means that we start from the root node and keep going down the "depth" of the tree until we reach a leaf node and then traverse back again to the node we started with.

- **Breadth First Traversal (Level Order Traversal):**
  Breadth First Traversal, also called level order traversal of the tree, is a traversal technique in which we traverse the tree in a "breadth first" manner. This means that nodes are traversed level-by-level in the tree from left to right.

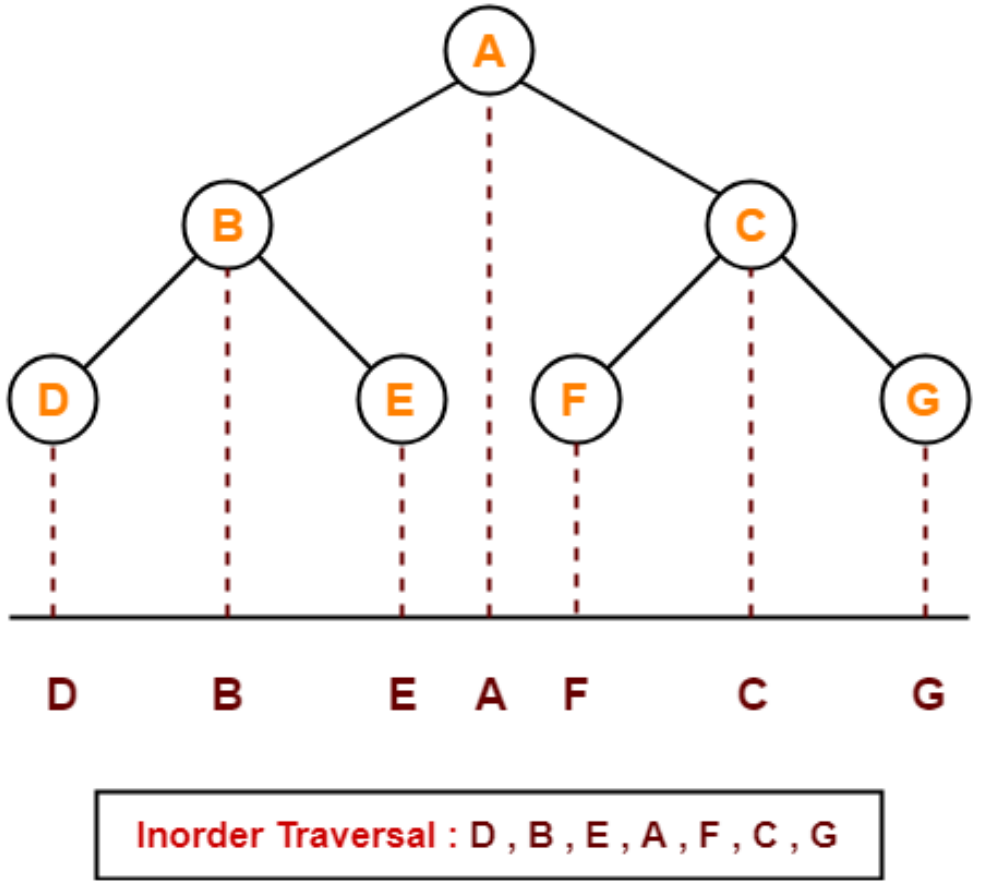
Inorder Traversal Algorithm

- Traverse the left sub tree i.e., call Inorder(left sub tree)
- Visit the root
- Traverse the right sub tree i.e., call Inorder(right sub tree)

inorder(root->left)

display(root->data)

inorder(root->right)



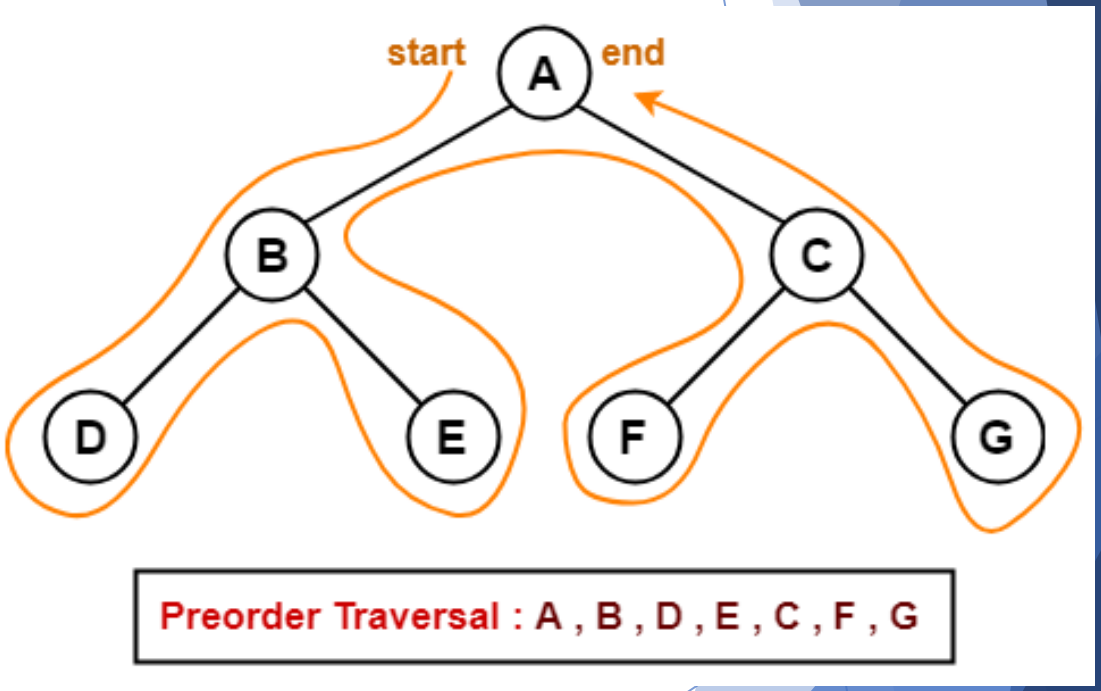
Preorder Traversal Algorithm

- Visit the root
- Traverse the left sub tree i.e., call Preorder (left sub tree)
- Traverse the right sub tree i.e., call Preorder (right sub tree)

display(root->data)

preorder(root->left)

preorder(root->right)



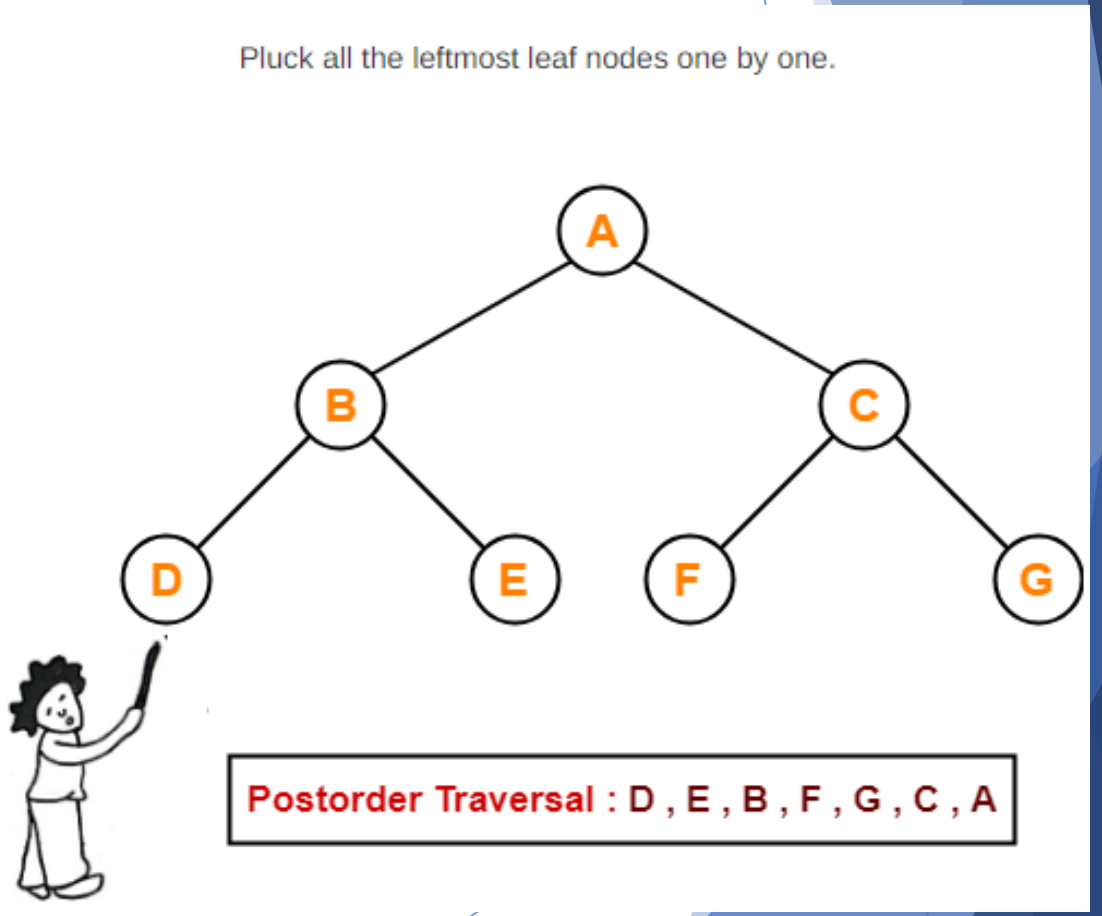
Postorder Traversal Algorithm

- Traverse the left sub tree i.e., call Postorder (left sub tree)
- Traverse the right sub tree i.e., call Postorder (right sub tree)
- Visit the root

postorder(root->left)

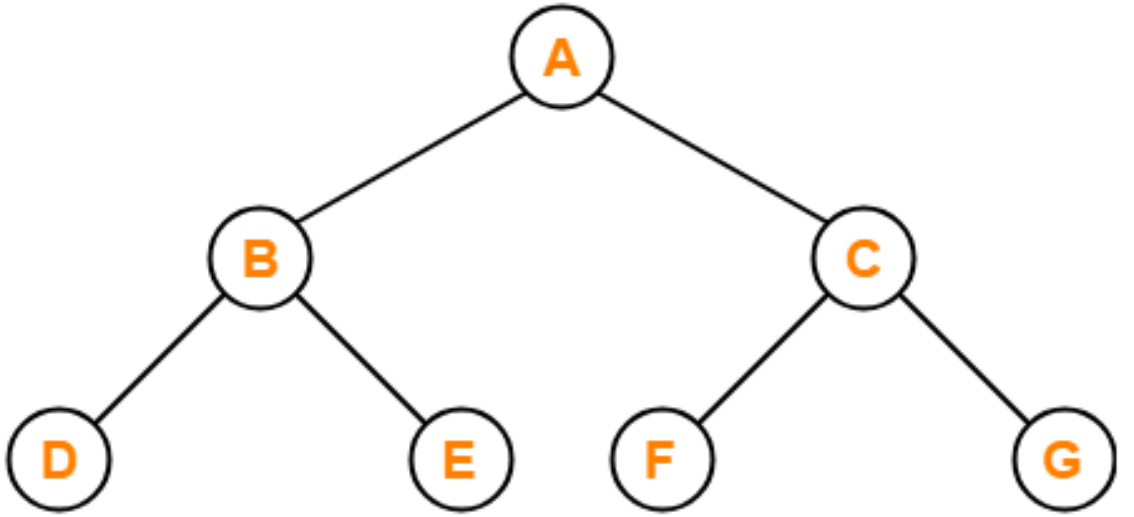
postorder(root->right)

display(root->data)



Breadth First Traversal (Level Order Traversal)

- Breadth First Traversal of a tree prints all the nodes of a tree level by level
- Breadth First Traversal is also called as Level Order Traversal



Recursive Inorder Traversal(Binary Tree)

- Traverse the left sub tree i.e., call Inorder(left sub tree)
- Display the root
- Traverse the right sub tree i.e., call Inorder(right sub tree)

```
// Inorder Traversal
void inorderTraversal(struct node* root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ->", root->item);
    inorderTraversal(root->right);
}
```

Recursive Preorder Traversal(Binary Tree)

- Display the root
- Traverse the left sub tree i.e., call Preorder (left sub tree)
- Traverse the right sub tree i.e., call Preorder (right sub tree)

```
// Preorder Traversal
void preorderTraversal(struct node* root) {
    if (root == NULL) return;
    printf("%d ->", root->item);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
```

Recursive Postorder Traversal(Binary Tree)

- Traverse the left sub tree i.e., call Postorder (left sub tree)
- Traverse the right sub tree i.e., call Postorder (right sub tree)
- Display the root

```
// Postorder traversal
void postorderTraversal(struct node* root) {
    if (root == NULL) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ->", root->item);
}
```

Create, Insert left/right (Binary Tree)

```
// Create a new Node
struct node* createNode(value) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->item = value;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
}
```

```
// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
    root->left = createNode(value);
    return root->left;
}
```

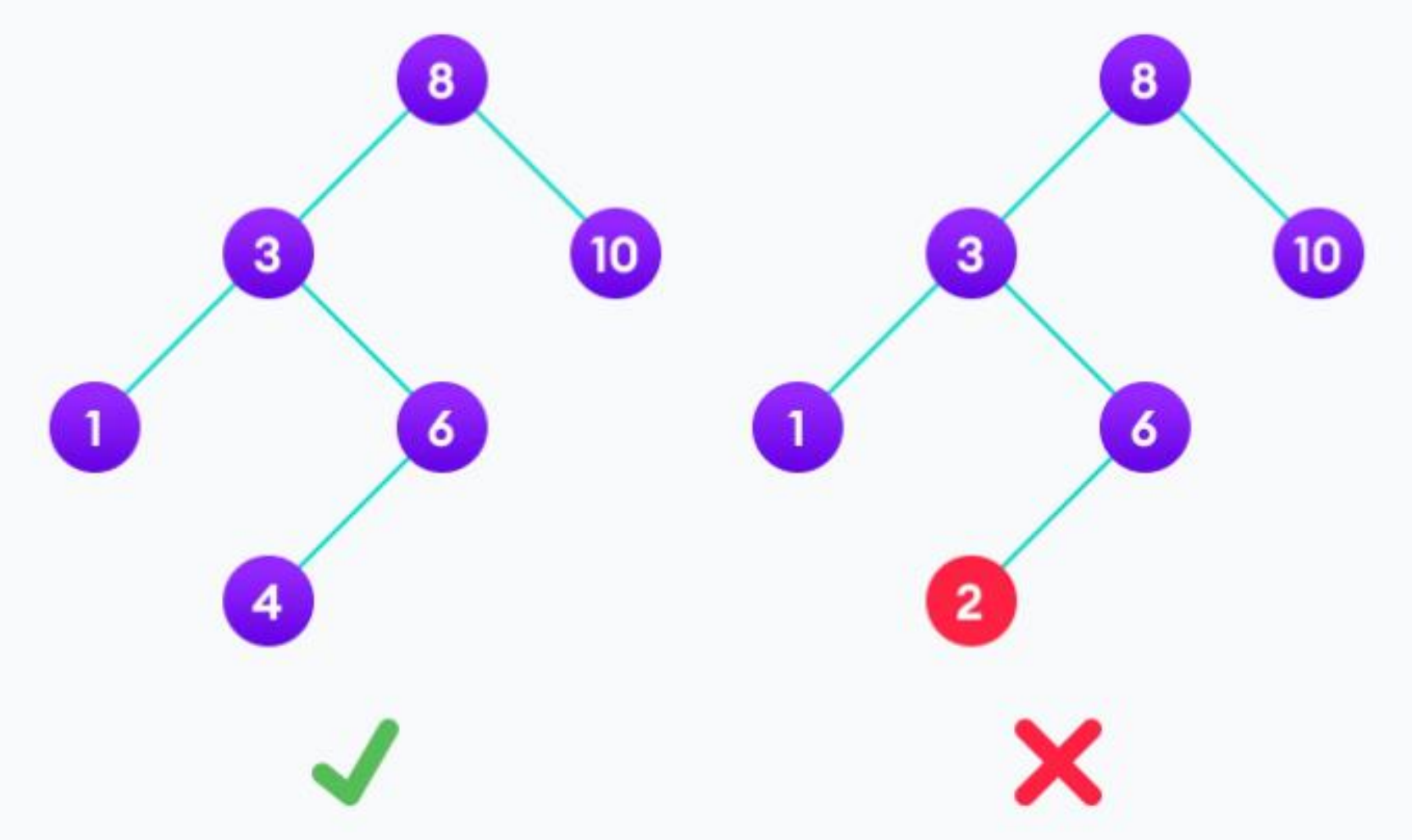
```
// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
    root->right = createNode(value);
    return root->right;
}
```

Binary Search Tree

Binary Search Tree(BST)

- Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers
- It is called a binary tree because each tree node has a maximum of two children
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time
- The properties that separate a binary search tree from a regular binary tree is
  - All nodes of left subtree are less than the root node
  - All nodes of right subtree are more than the root node
  - Both subtrees of each node are also BSTs i.e., they have the above two properties

Binary Search Tree(BST)...



Binary Search Tree Operations

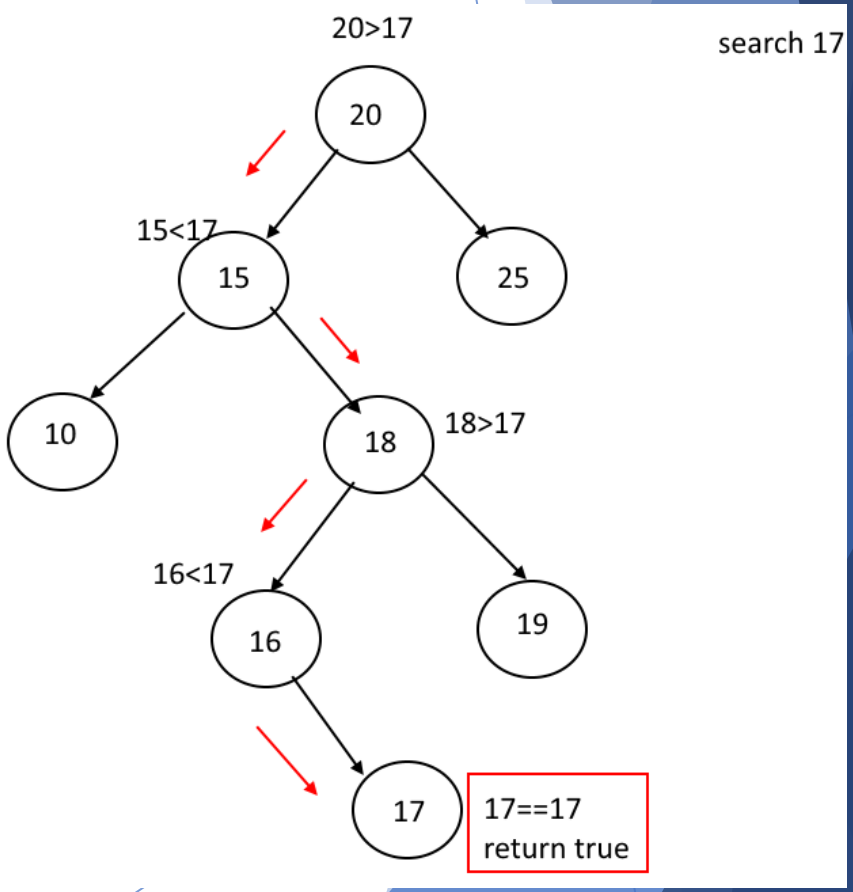
- Insertion – Inserts a node in a tree/create a tree.
- Search – Searches for a node in a tree
- Traversal – Traversing a tree
- Deletion – Deletes a node from the tree

Search Operation (BST)

- The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root
- If the value is below the root, we can say for sure that the value is not in the right subtree
- We need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree
- The function search recursively searches the subtrees

Search BST Algorithm

- If root == NULL then return NULL value
- If number == root->data then return root->data
- If number < root->data then return search(root->left)
- If number > root->data then return search(root->right)



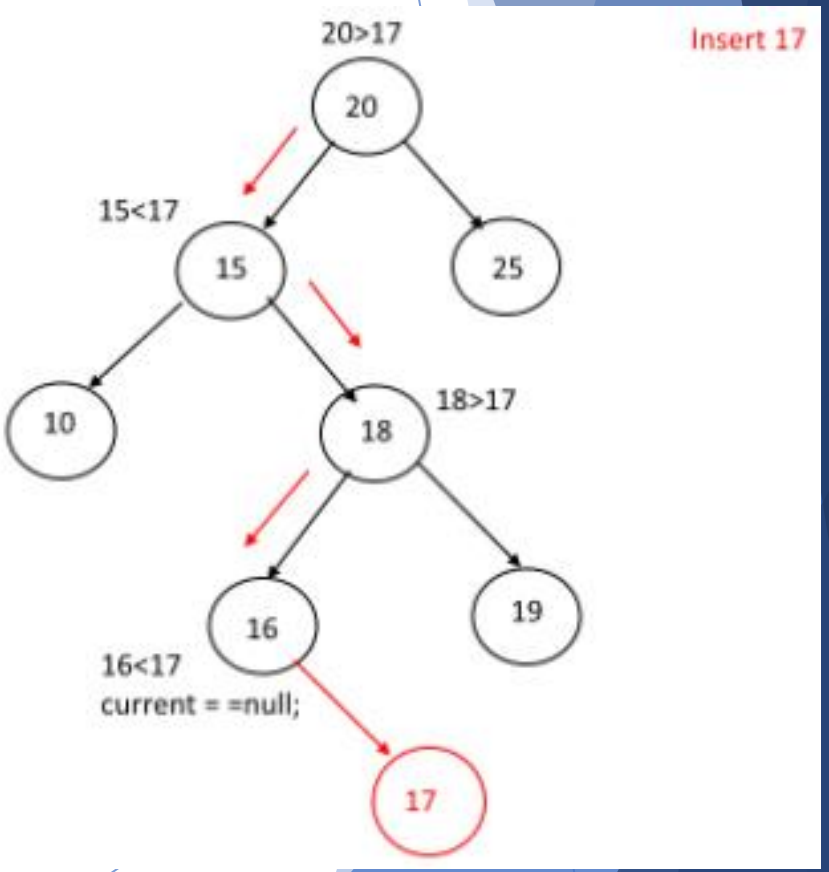
Search in BST

```
// C function to search a given key in a given BST
struct node* search(struct node* root, int key) {
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

Insertion Operation (BST)

- Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root

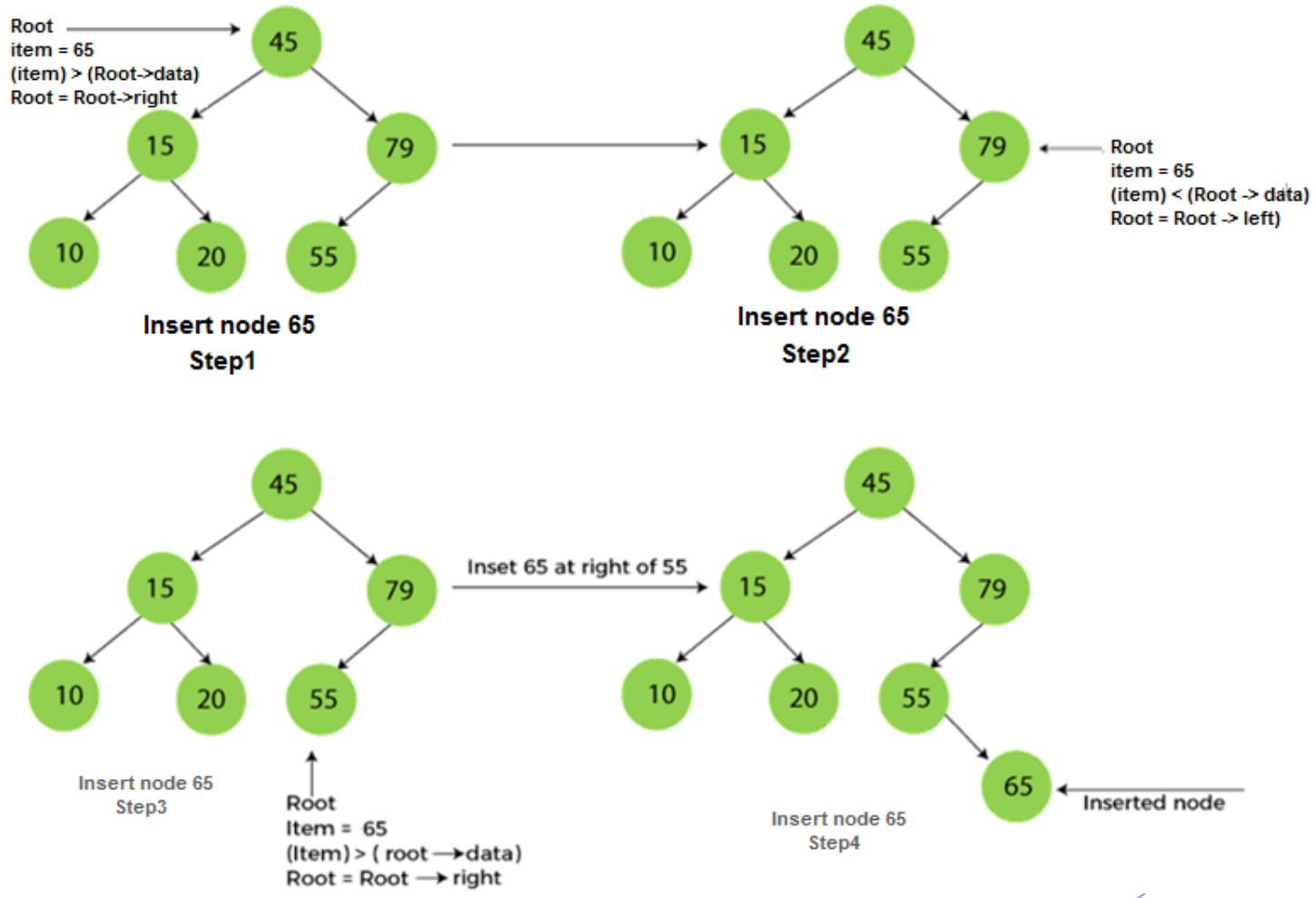
- We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there



Insertion BST Algorithm

- To insert a node our first task is to find the place to insert the node
- Take current = root
- Start from the current and compare root->data with n
- If current->data is greater than n that means we need to go to the left of the root
- If current->data is smaller than n that means we need to go to the right of the root
- If any point of time current is null that means we have reached to the leaf node, insert your node here with the help of parent node

Insertion in BST



Insertion in BST

```
// Insert a node
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}
```

```
// Create a node
struct node *newNode(int item) {
  struct node *temp = (struct node
*)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}
```

Traversal in BST

- Same as Binary Tree Traversal refer slide 21

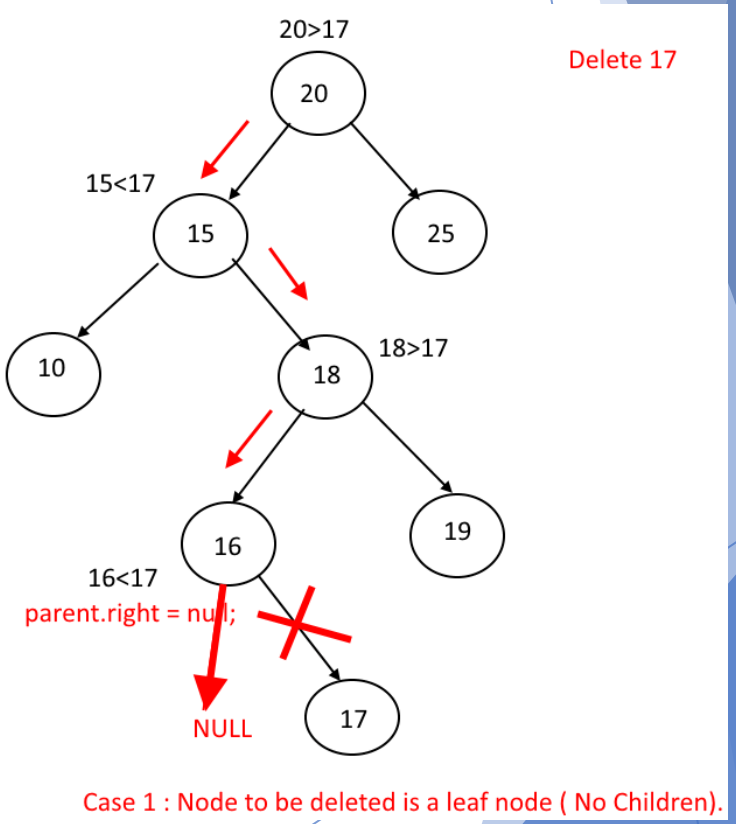
Deletion in BST

- Deletion in Binary Search Tree is much complicated compared to searching and inserting node

- When it comes to deleting a node from the binary search tree, following three cases are possible
  - Node to be deleted is a leaf node (No Children)
  - Node to be deleted has only one child
  - Node to be deleted has two childrens

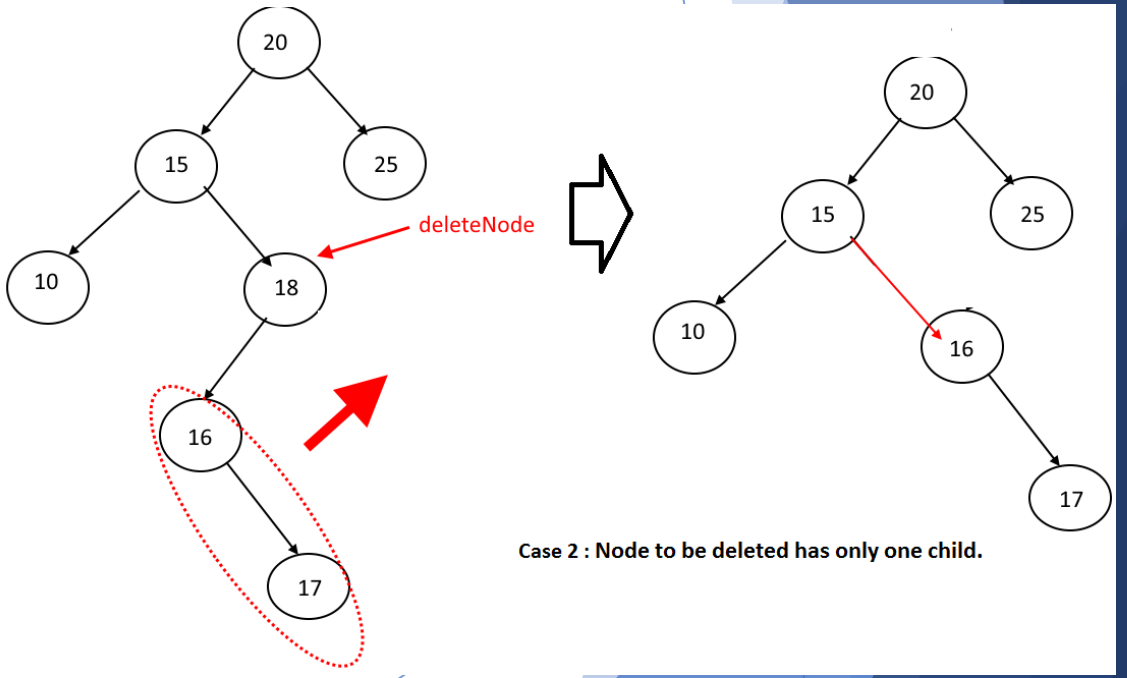
Node to be deleted is a leaf node

- Its a simple case, if a node to be deleted has no children, then just traverse to that node
- And keep track of parent node and the side in which the node exist(left or right) and set ***parent.left = null or parent.right = null***



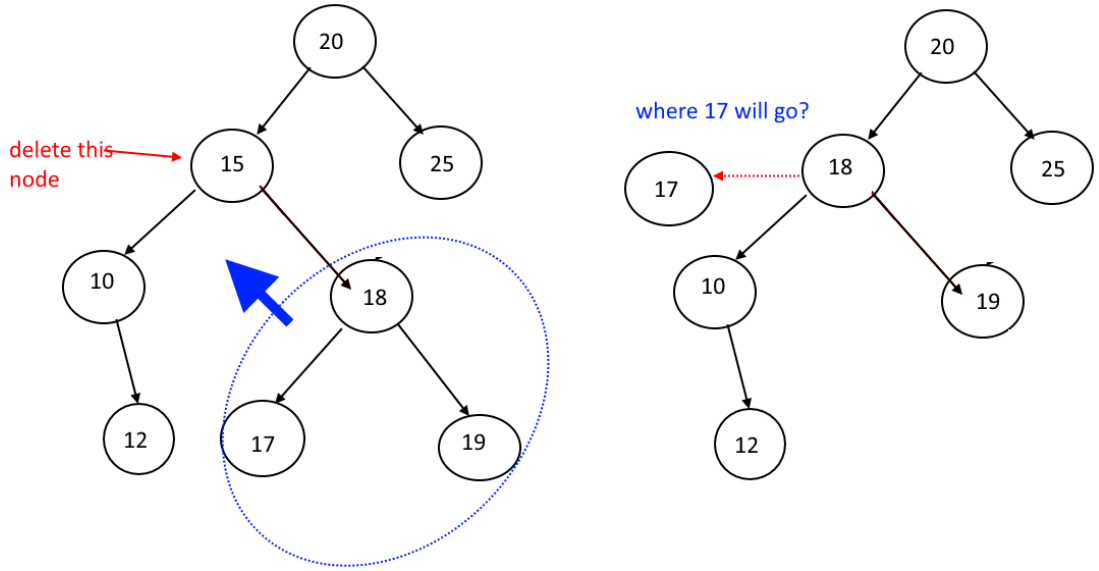
Node to be deleted has only one child

- if a node to be deleted(deleteNode) has only one child then just traverse to that node, keep track of parent node and the side in which the node exist(left or right)
- check which side child is null (since it has only one child)
- say node to be deleted has child on its left side. Then take the entire sub tree from the left side and add it to the parent and the side on which deleteNode exist



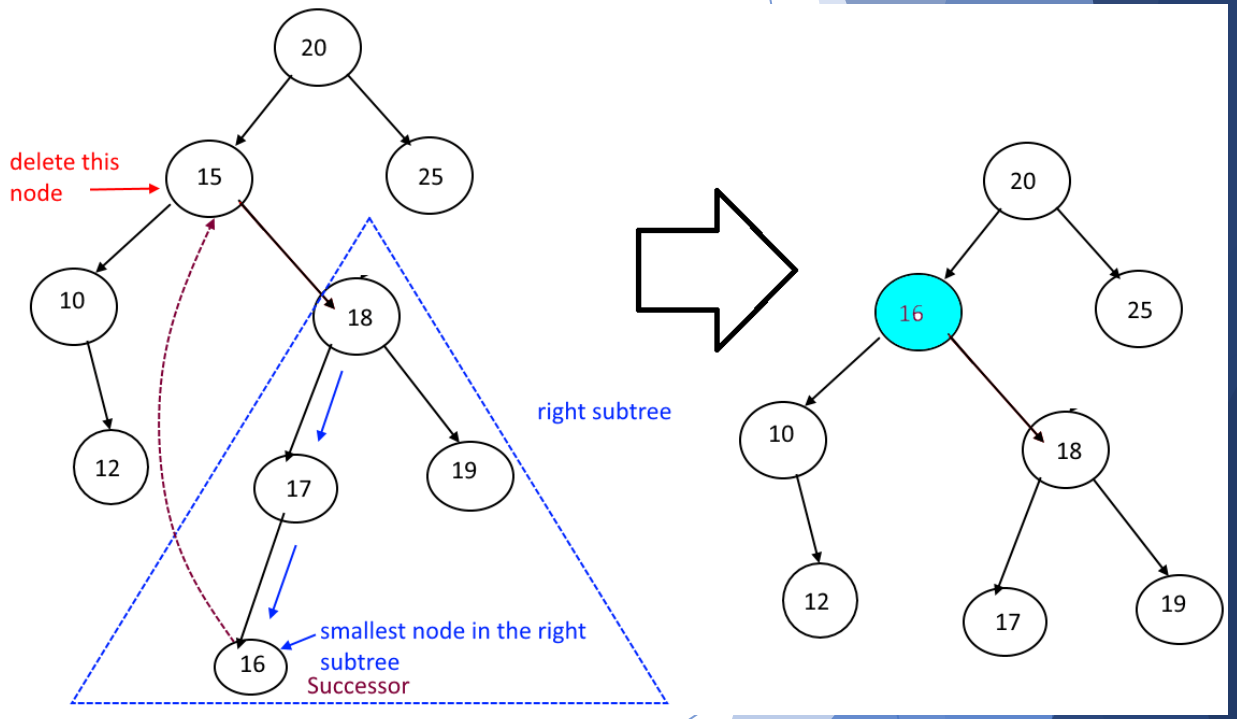
Node to be deleted has two children

- You just cannot replace the deleteNode with any of its child, Why? Refer the diagram on the right



Node to be deleted has two children…

- **Find The Successor node**
- Successor is the node which will replace the deleted node. Now the question is to how to find it and where to find it.
- *Successor is the smallest node in the right sub tree of the node to be deleted*



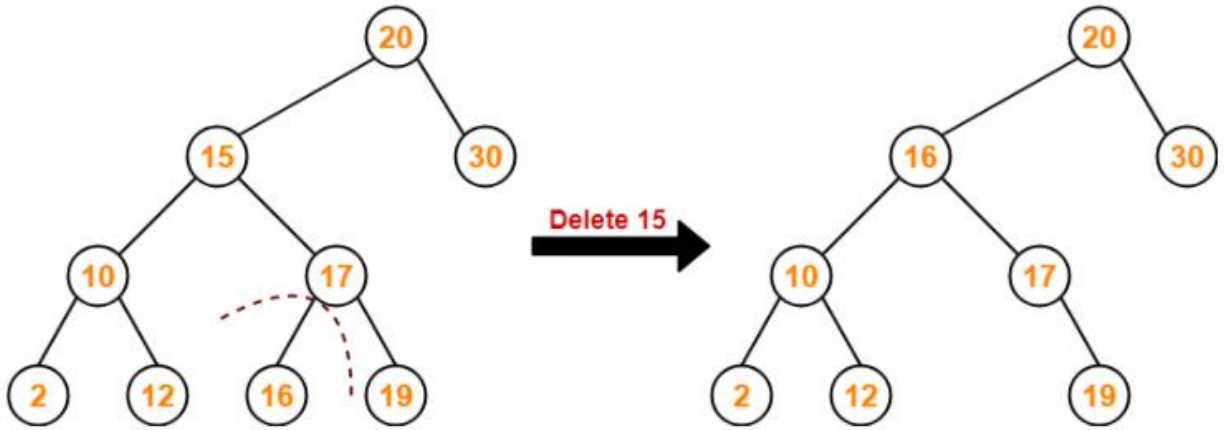
Node to be deleted has two children…

- Approach Method 1
- Visit to the right subtree of the deleting node
- Pluck the **least value** element called as inorder successor
- Replace the deleting element with its inorder successor

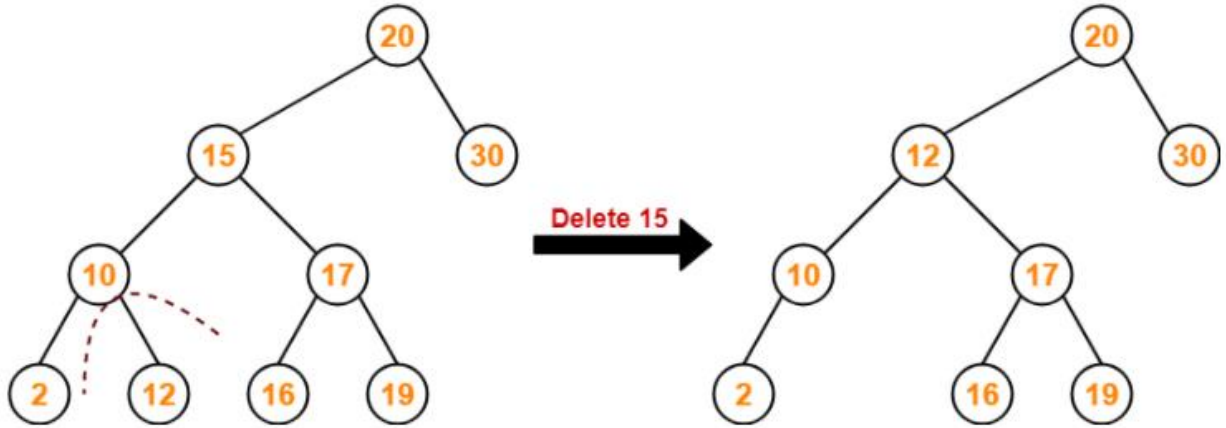
- Approach Method 2
- Visit to the left subtree of the deleting node
- Pluck the **greatest value** element called as inorder predecessor
- Replace the deleting element with its inorder predecessor

Node to be deleted has two children…

Approach Method 1



Approach Method 2



Delete node in BST

```
// Deleting a node
struct node *deleteNode(struct node *root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);

  else {
    // If the node is with only one child or no child
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // If the node has two children
    struct node *temp = minValueNode(root->right);

    // Place the inorder successor in position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}
```

Delete node in BST…

```
// Find the inorder successor
struct node *minValueNode(struct node *node) {
  struct node *current = node;

  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;

  return current;
}
```

Deleting Entire BST

- To delete/remove an entire binary search tree from the memory, we first delete the elements/nodes in the left sub-tree and then delete the nodes in the right sub-tree

```
IF TREE != NULL
    deleteTree (TREE->LEFT)
    deleteTree (TREE->RIGHT)
    Free (TREE)
[END OF IF]
```

Deleting Entire BST…

```
//  This function traverses tree in post order to delete each and every node of the tree
void deleteTree(struct node* node) {
    if (node == NULL) return;

    // first delete nodes in both subtrees
    deleteTree(node->left);
    deleteTree(node->right);

    // then delete the leaf node
    printf("\n Deleting node: %d", node->data);
    free(node);
}
```

Binary Search Tree Complexities

- **Time Complexity**

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

- Here, n is the number of nodes in the tree

- **Space Complexity**
- The space complexity for all the operations is O(n)

AVL Tree

What is AVL Tree?

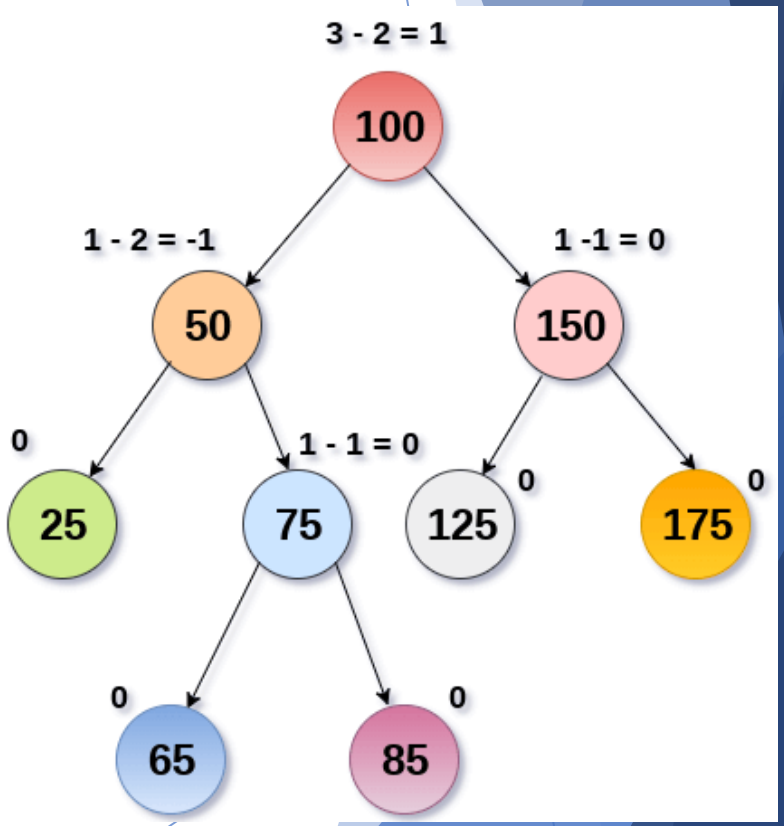
- AVL Tree can be defined as height/self balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree
- A node is called left heavy(1) if the longest path in its left subtree is one longer than the longest path of its right subtree
- A node is called right heavy(-1) if the longest path in the right subtree is one longer than the path in its left subtree
- A node is called balanced(0) if the longest path in both the right and left subtree are equal
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced
- AVL Tree helps avoid left and right skewed binary tree cases

Balance Factor

- Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node

- Balance Factor = (Height of Left Subtree - Height of Right Subtree)

- The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1



AVL Tree Complexities

- **Time Complexity**

Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

- Here, n is the number of nodes in the tree

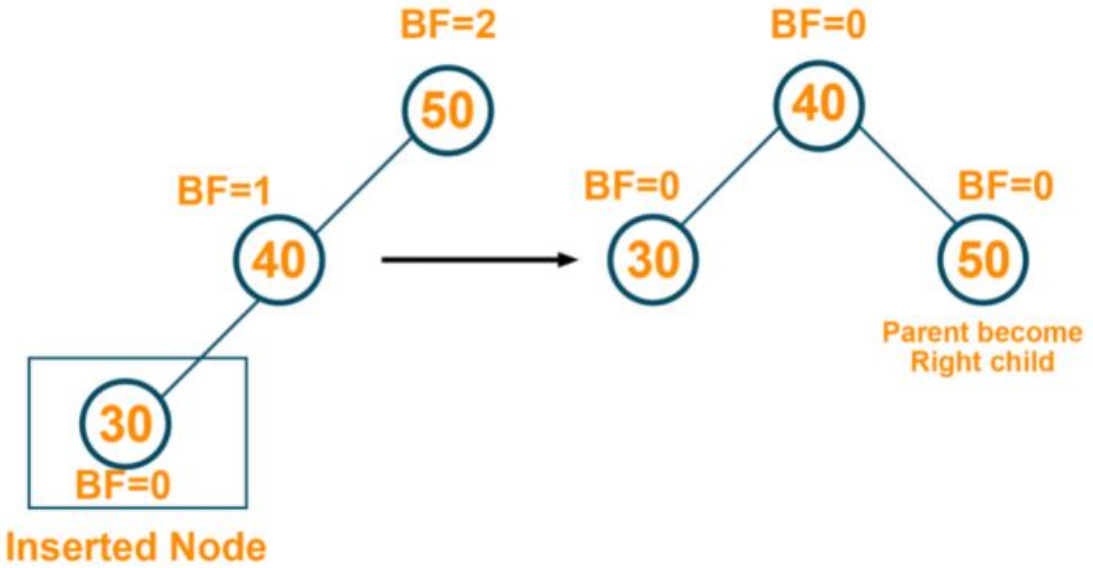
- **Space Complexity**
- The space complexity for all the operations is O(n)

AVL Rotation

- When certain operations like insertion and deletion are performed on the AVL tree, the balance factor of the tree may get affected
- If after the insertion or deletion of the element, the balance factor of any node is affected then this problem is overcome by using rotation
- Rotation is the method of moving the nodes of trees either to left or to right to make the tree heighted balance tree
- There are four categories of rotations
  - LL rotation: Inserted node is in the left subtree of left subtree of A
  - RR rotation: Inserted node is in the right subtree of right subtree of A
  - LR rotation: Inserted node is in the right subtree of left subtree of A
  - RL rotation: Inserted node is in the left subtree of right subtree of A

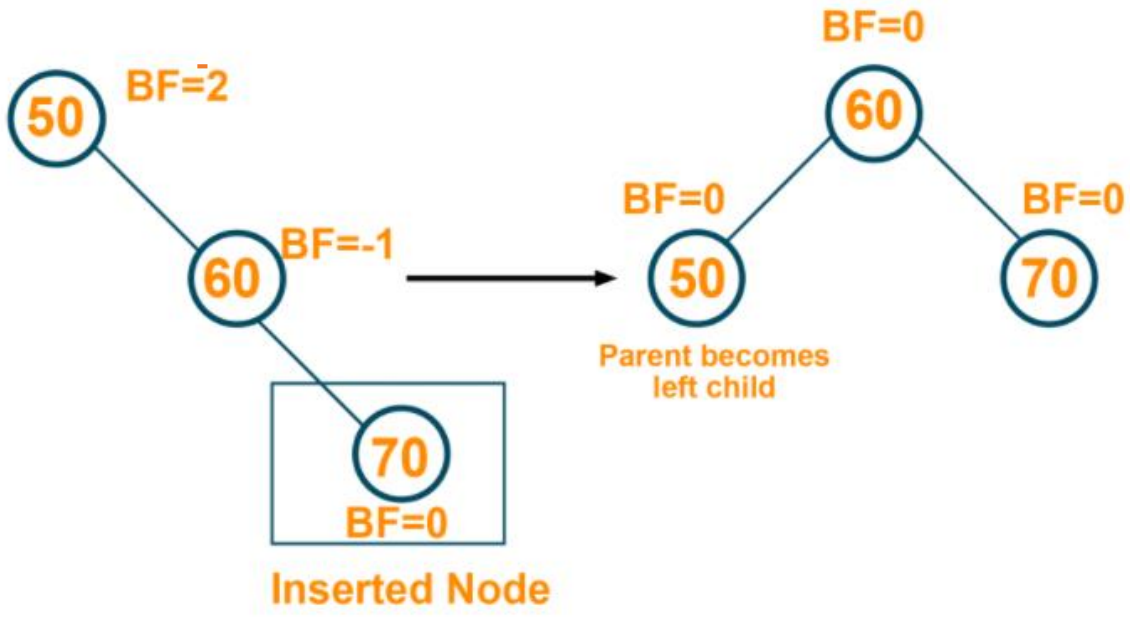
LL(Left Left) Rotation

- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation
- LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2
- Therefore, a parent becomes the right child in LL rotation



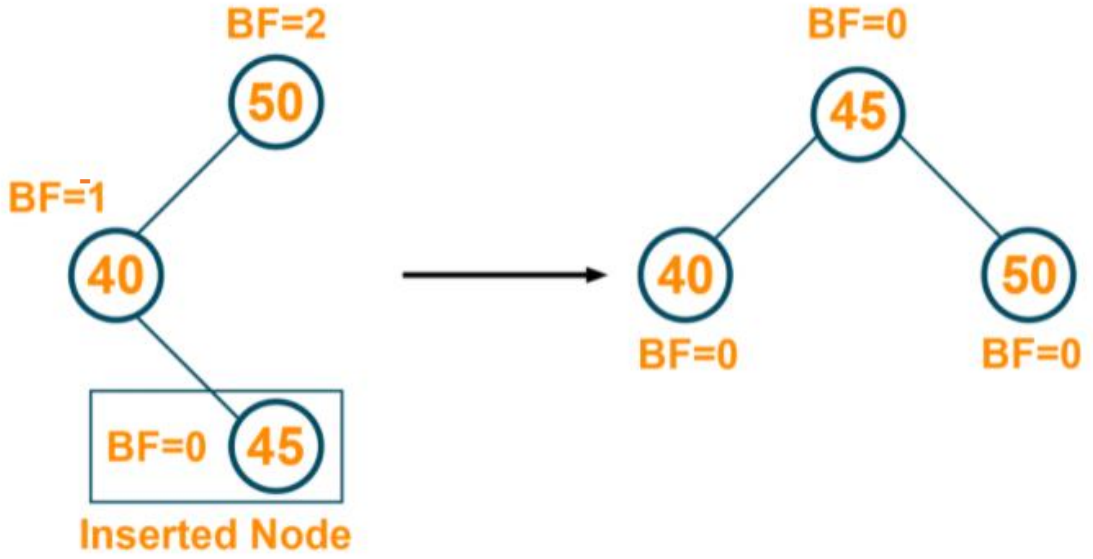
RR(Right Right) Rotation

- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation
- RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2
- Therefore, the parent becomes a left child in RR rotation

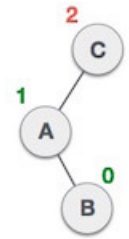


LR(Left Right) Rotation

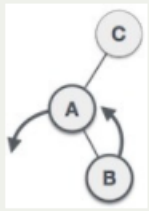
- Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation
- First RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1



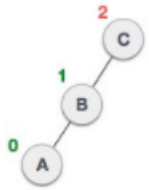
LR(Left Right) Rotation



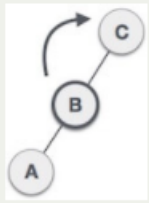
A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C



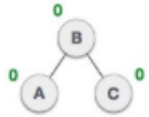
As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.



After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**



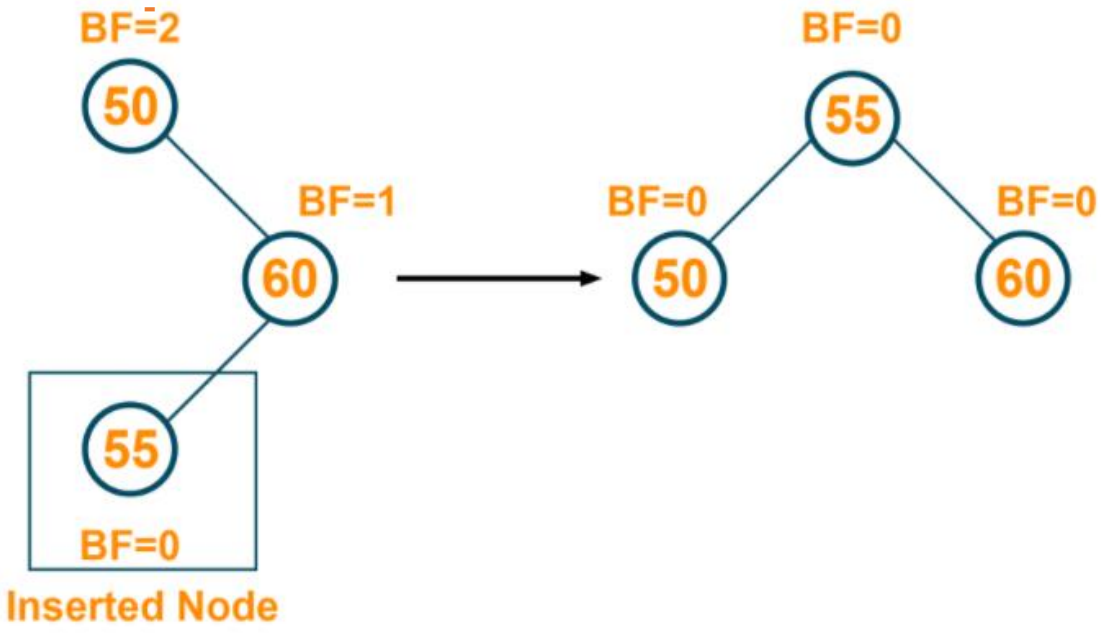
Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B



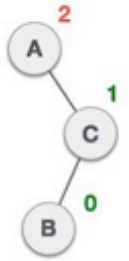
Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

RL(Right Left) Rotation

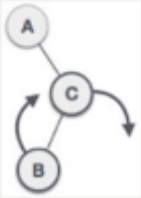
- Double rotations are bit tougher than single rotation which has already explained above. RL rotation = LL rotation + RR rotation
- First LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1



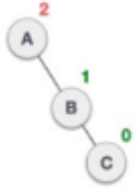
RL(Right Left) Rotation



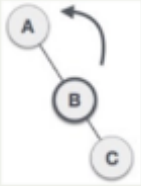
A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A



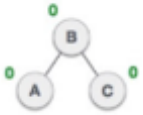
As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.



After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.



Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.



Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

Insertion Operation In AVL Tree

- In the AVL tree, the new node is always added as a leaf node
- After the insertion of the new node, it is necessary to modify the balance factor of each node in the AVL tree using the rotation operations
- The algorithm steps of insertion operation in an AVL tree are
  1) Find the appropriate empty subtree where the new value should be added by comparing the values in the tree
  2) Create a new node at the empty subtree
  3) The new node is a leaf and thus will have a balance factor of zero
  4) Return to the parent node and adjust the balance factor of each node through the rotation process and continue it until we are back at the root. Remember that the modification of the balance factor must happen in a **bottom-up** fashion

Construct an AVL tree by inserting elements in given order

- **63, 9, 19, 27, 18, 108, 99, 81**

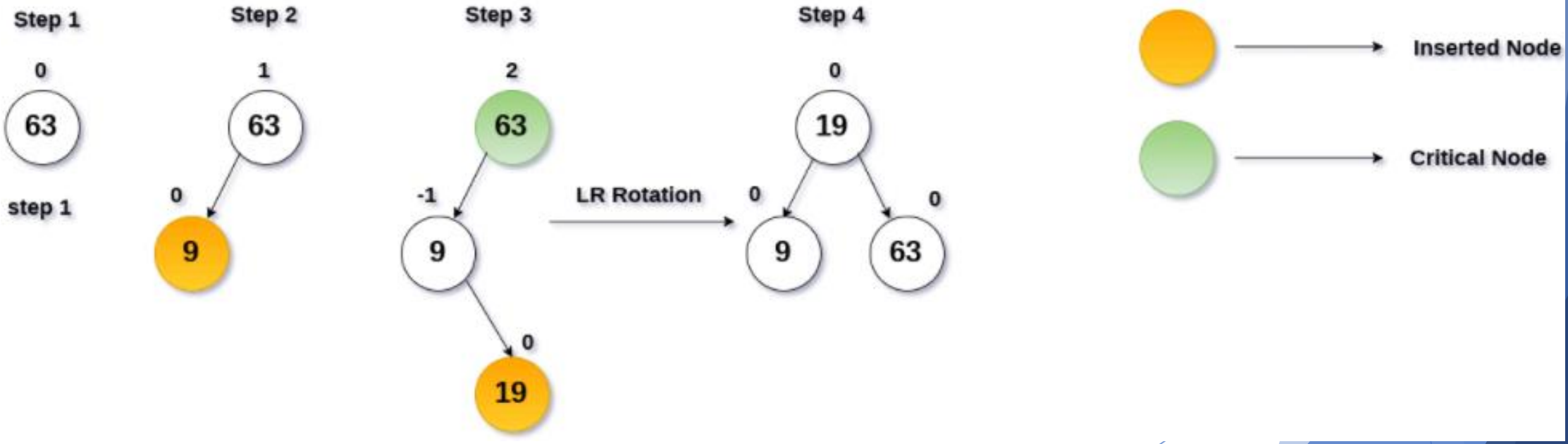
- At each step, we must calculate the balance factor for every node, if it is found to be more than 1 or less than -1, then we need a rotation to rebalance the tree

- The type of rotation will be estimated by the location of the inserted element with respect to the critical node

- All the elements are inserted in order to maintain the order of binary search tree

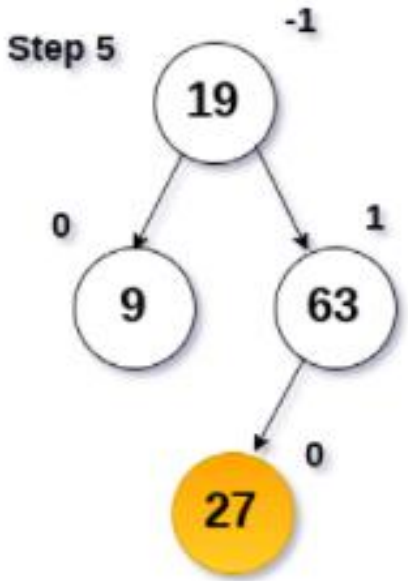
Construct an AVL tree…

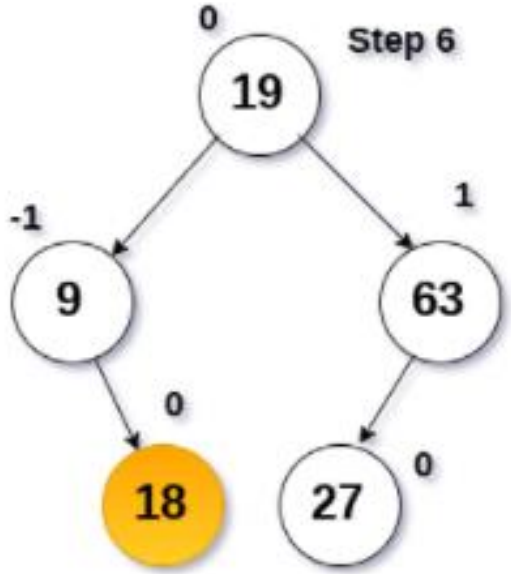
- 63, 9, 19, 27, 18, 108, 99, 81

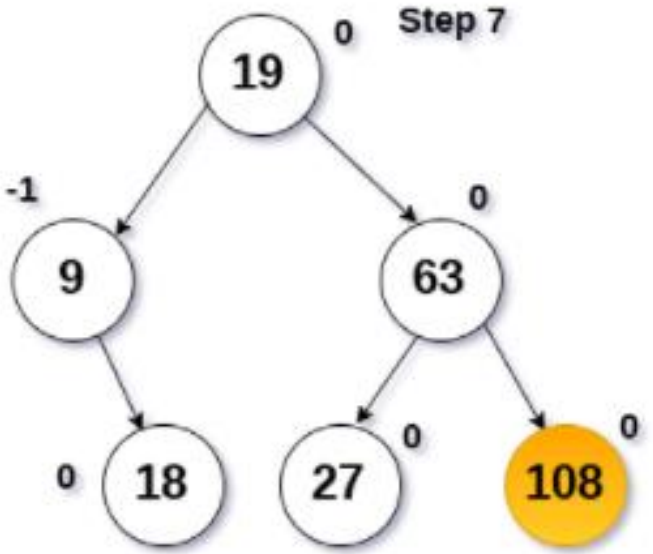


Construct an AVL tree…

- 63, 9, 19, 27, 18, 108, 99, 81

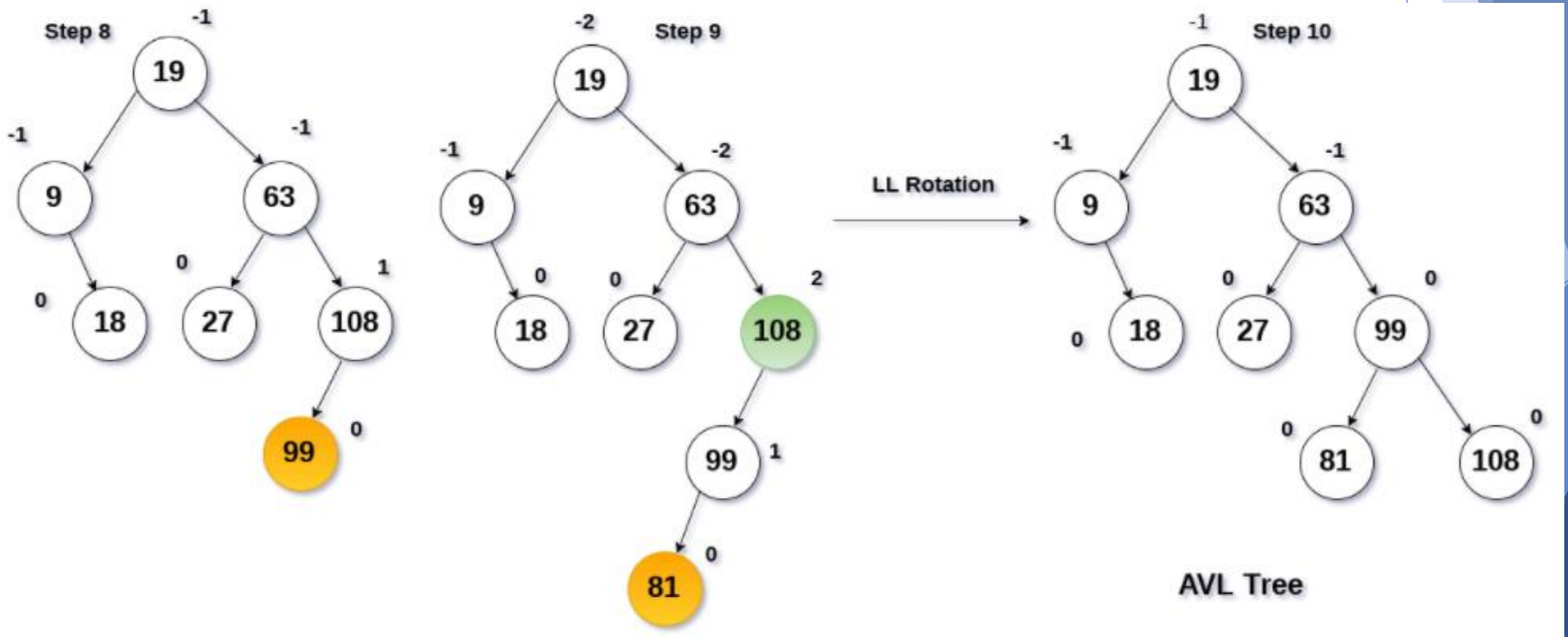






Construct an AVL tree…

- 63, 9, 19, 27, 18, 108, 99, 81

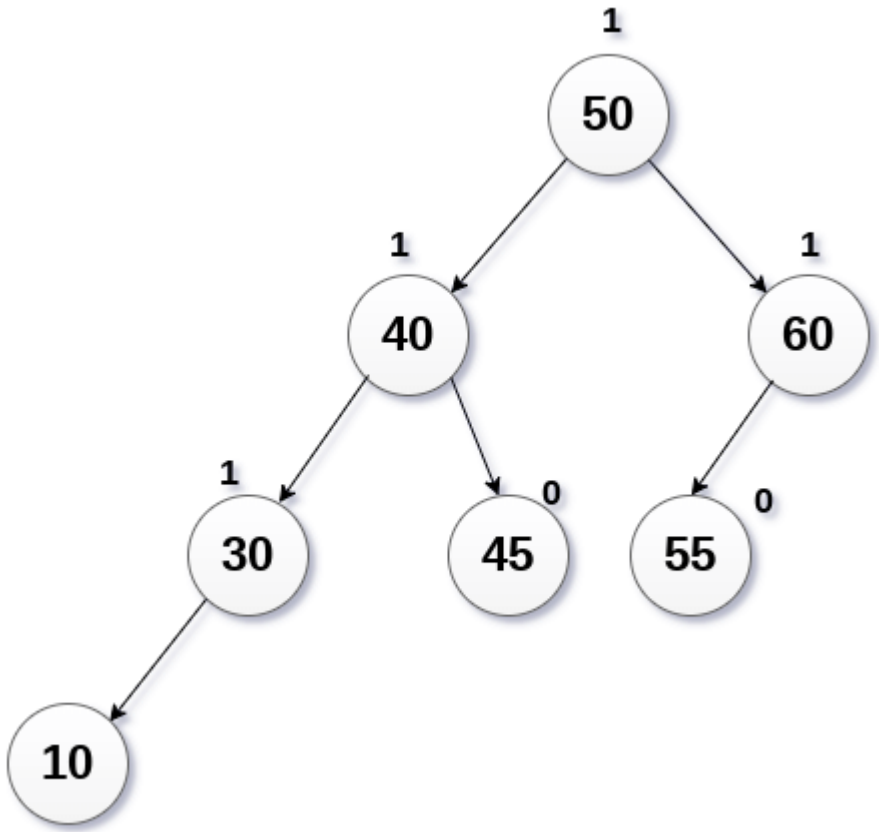


Deletion Operation in AVL

- The deletion operation in the AVL tree is the same as the deletion operation in BST
- In the AVL tree, the node is always deleted as a leaf node and after the deletion of the node, the balance factor of each node is modified accordingly. Rotation operations are used to modify the balance factor of each node.
- The algorithm steps of deletion operation in an AVL tree are:
  1) Locate the node to be deleted
  2) If the node does not have any child, then remove the node
  3) If the node has one child node, replace the content of the deletion node with the child node and remove the node
  4) If the node has two children nodes, find the inorder successor node ‘k' which has no child node and replace the contents of the deletion node with the ‘k’ followed by removing the node
  5) Update the balance factor of the AVL tree

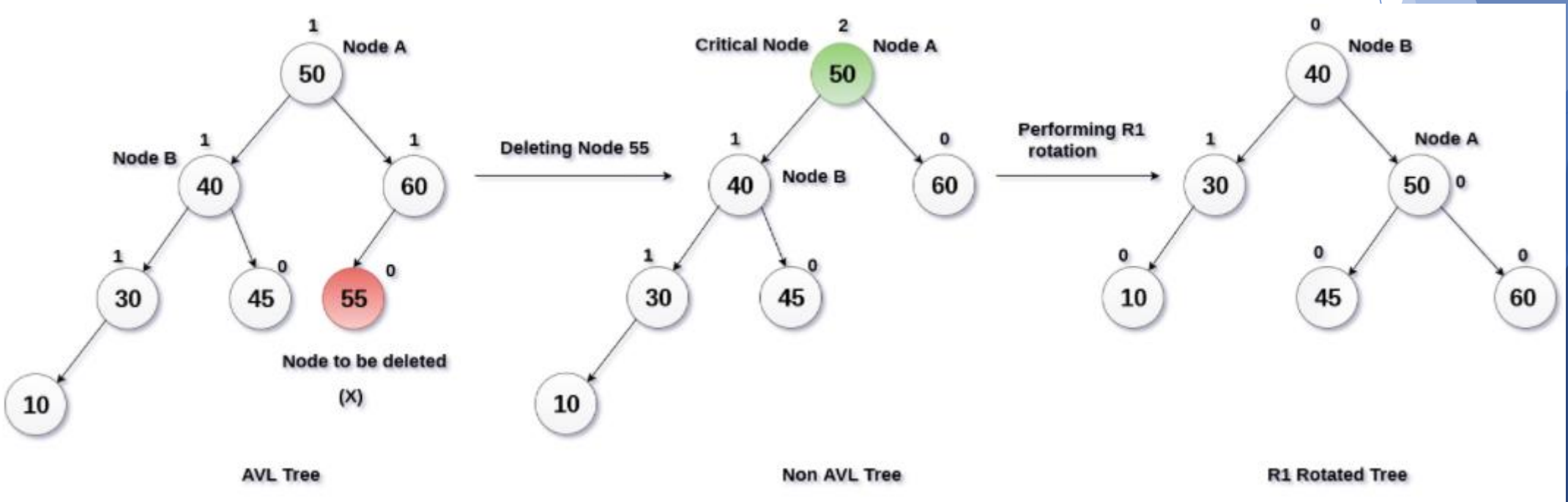
Deleting node from existing AVL tree

- Delete the node 55 from the AVL tree shown in the following image



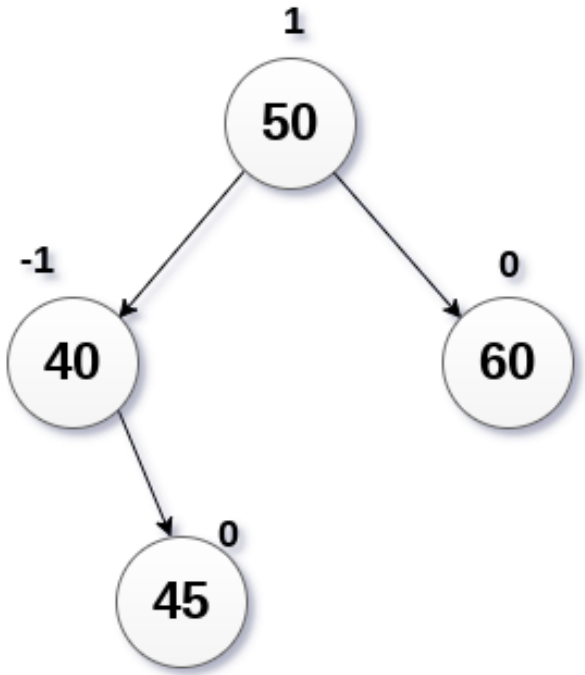
Deleting node from existing AVL tree…

- Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e., node A which becomes the critical node.
- This is the condition of LL rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A



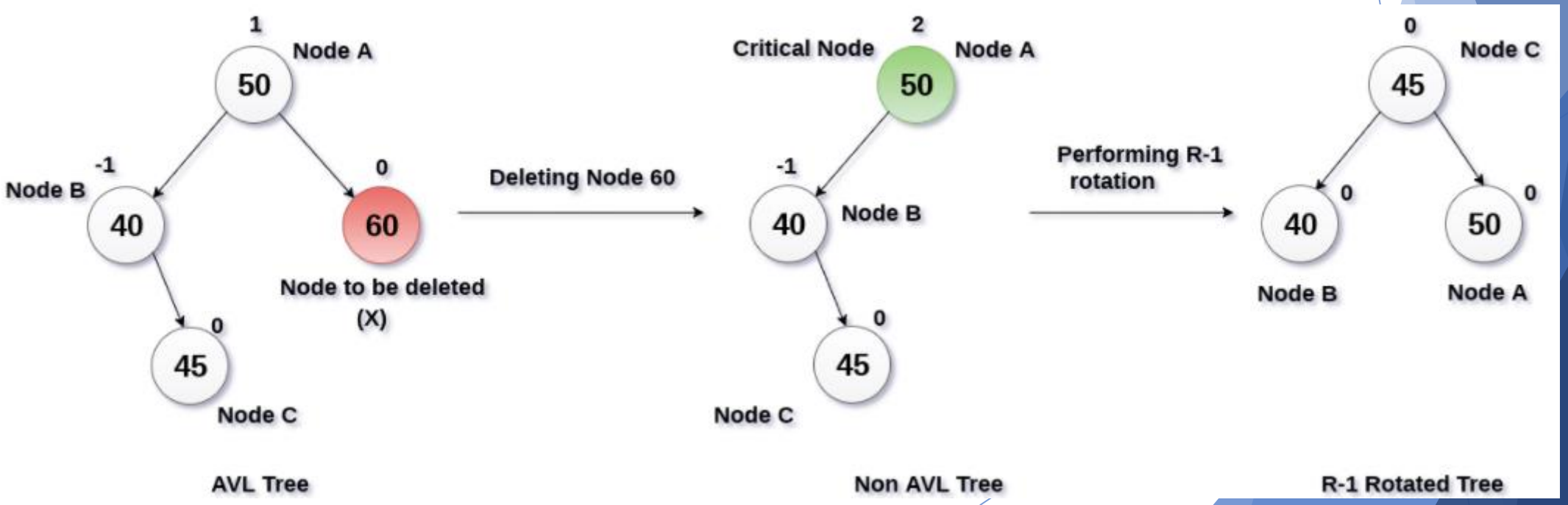
Deleting node from existing AVL tree

- Delete the node 60 from the AVL tree shown in the following image



Deleting node from existing AVL tree

- Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be LR rotated
- The node C i.e., 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child



Search in AVL Tree

- An AVL tree is just a special kind of BST, and you can search it the same way
- We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree
- Otherwise, if the key equals that of the root, the search is successful and we return the node
- If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree
- This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found before a null subtree is reached, then the key is not present in the tree

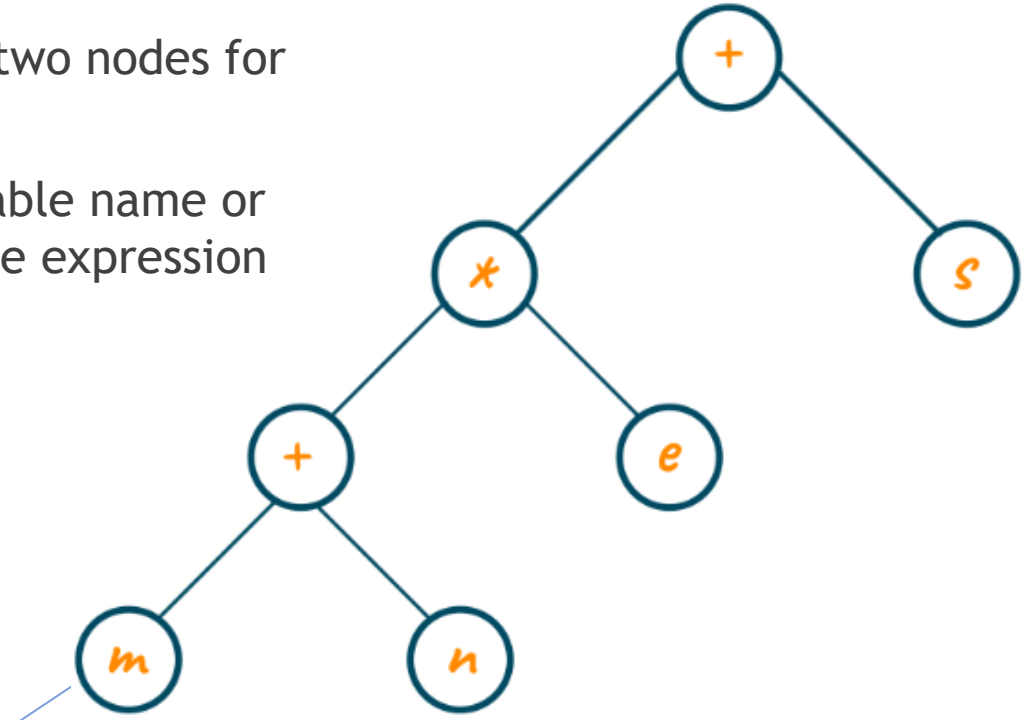
Traversal in AVL Tree

- As AVL tree is a special kind of BST, the traversing is same as BST
- Same as Binary Tree Traversal refer slide 21

Expression Tree

What is the Expression Tree?

- An expression tree is one form of binary tree that is used to represent the expressions
- A binary expression tree can represent two types of expressions i.e., Algebraic expressions and Boolean expressions
- Just like a binary tree, an expression tree has zero, one, or two nodes for each parent node
- The leaves of the expression tree are **operands** such as variable name or constants and the other nodes represents the **operator** of the expression
- Expression tree for the equation: (m+n)*e+s

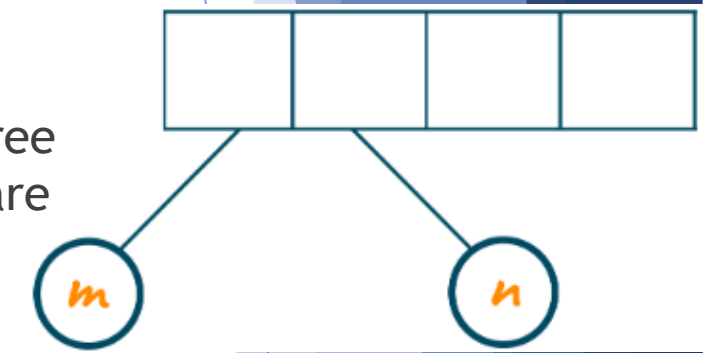


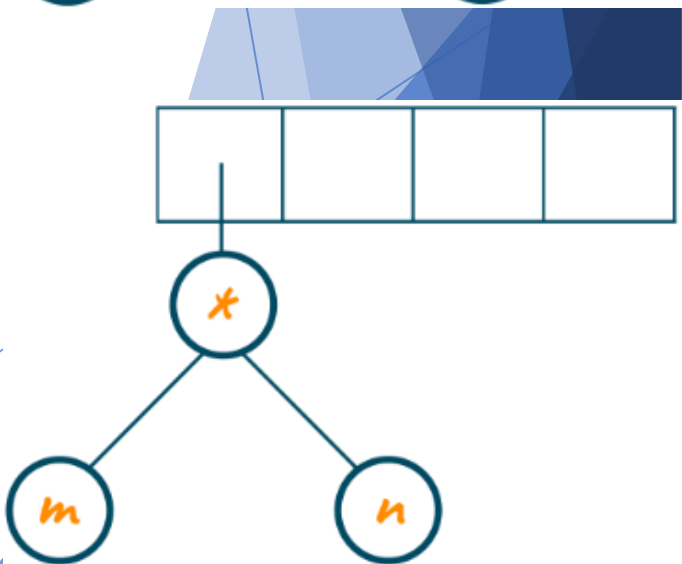
Construction of Expression Tree (Using Postfix Expression)

- Let us consider a **postfix expression** is given as an input for constructing an expression tree. Following are the step to *construct an expression tree*:
  - Scan the expression from left to right.
  - If an operand is found. Then create a node for this operand and **push** it into the stack
  - If an operator is found. **Pop** two nodes from the stack and make a node keeping the operator as the root node and the two items as the left and the right child. **Push** the newly generated node into the stack
  - Keep repeating the above two steps until we reach the end of our postfix expression
  - The node that is left behind in the stack represents the head of the expression tree

Construction of Expression Tree(Example)

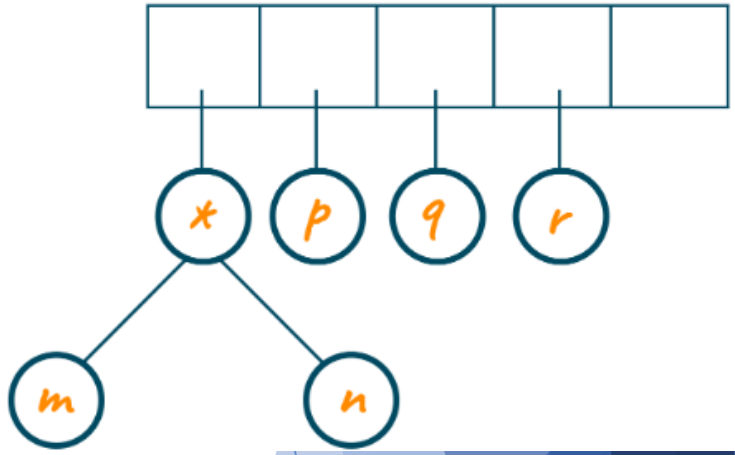
- Consider postfix notation is: m n * p q r + * +
- Here the first two symbols are operands, i.e., m and n. So, the one-node tree is created as shown in the side image, and the pointers of these operands are **pushed** into the stack
- The next in the equation is the "*" operator. Therefore, we will **pop** the operands pointers from the stack and form a new tree where the operator serves as root node and operands serves as left and right child. Later, the pointer to the tree is **pushed** into the stack as shown in the side example





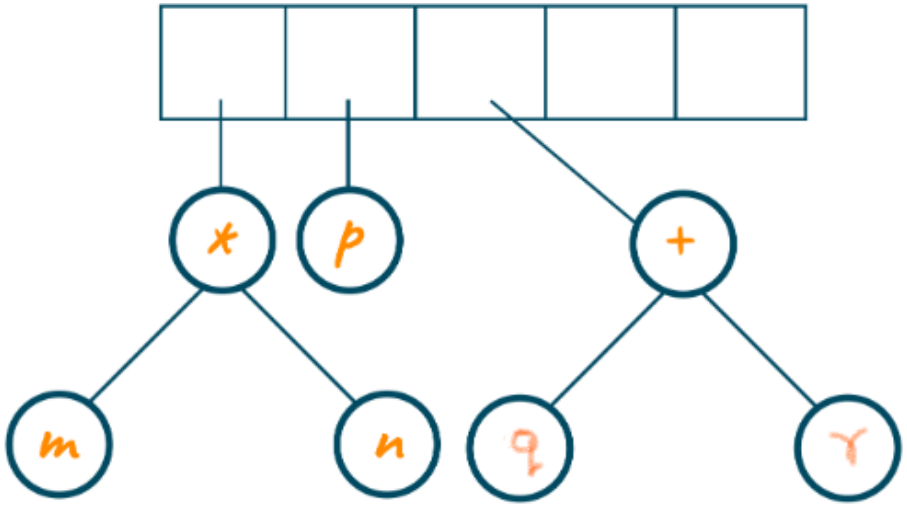
Construction of Expression Tree(Example)...

m n * p q r + * +



- Now, the postfix expression traverse to “p”, “q”, and “r”. As they are operands, the one-node tree is formed and the pointer to each node is **pushed** into the stack

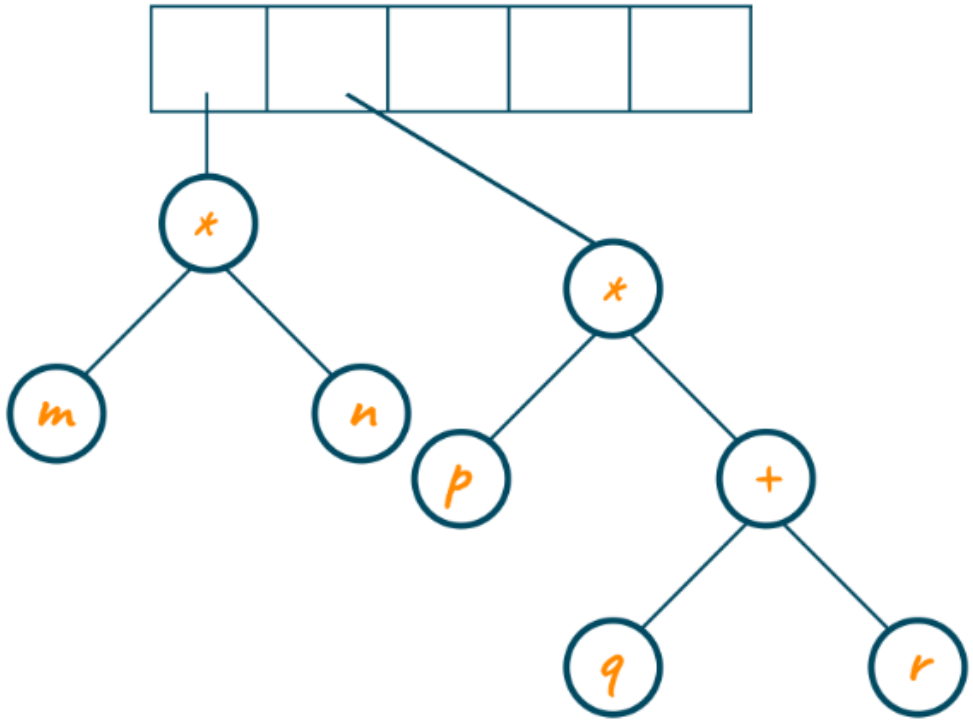
- Later, the operator “+” is encountered and it serves as the root node to the last two one-node operands in the stack. The pointer to this new tree is stored in the stack as shown in the below image



Construction of Expression Tree(Example)...

m n * p q r + * +

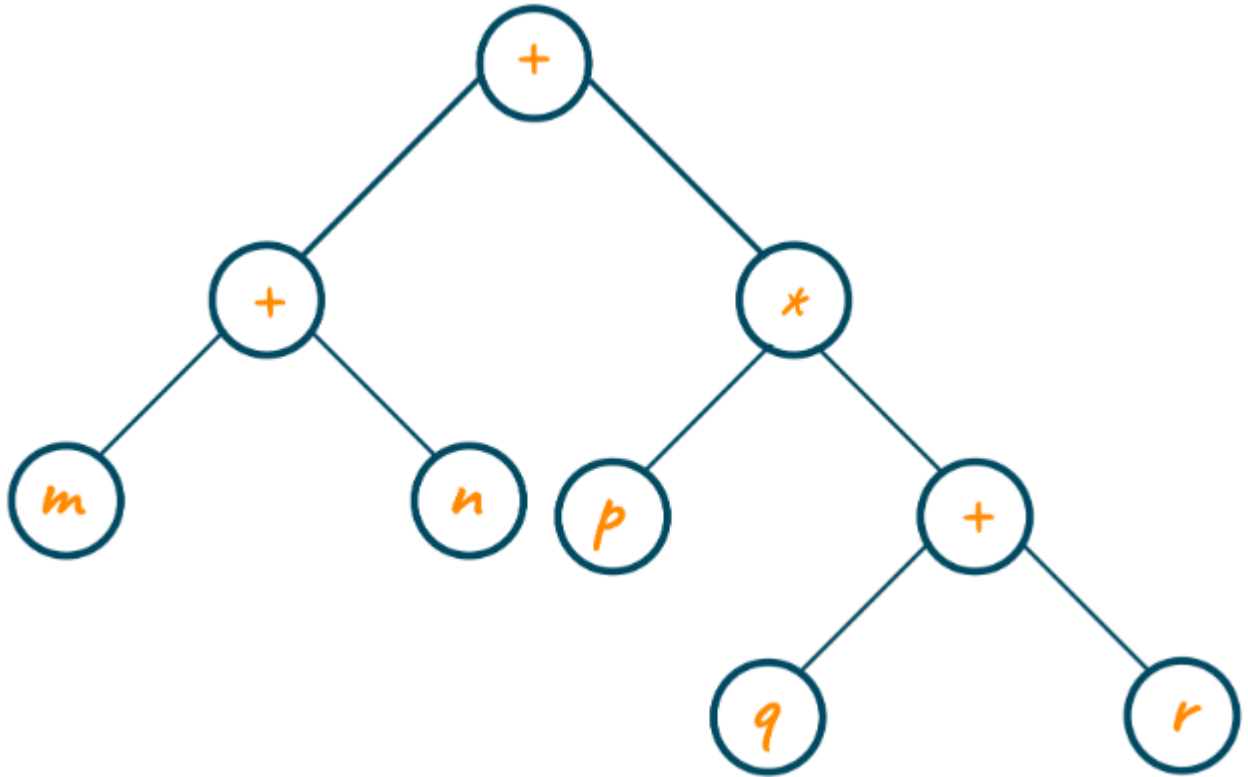
- Now, again "*" operator is read . The last two tree pointers are **popped** from the stack and a new tree is built with root node as "*" operator as shown in the below image



Construction of Expression Tree(Example)...

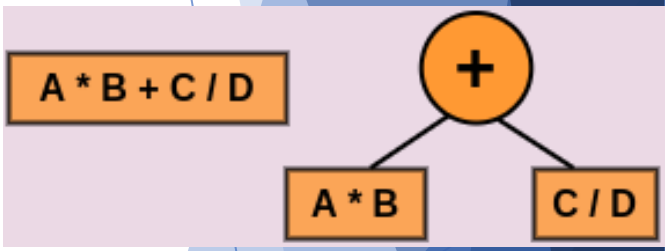
m n * p q r + * +

- At last, the two individual trees are combined with the "+" operator, and the final expression tree is formed. The pointer to the new tree is stored in the stack as shown below

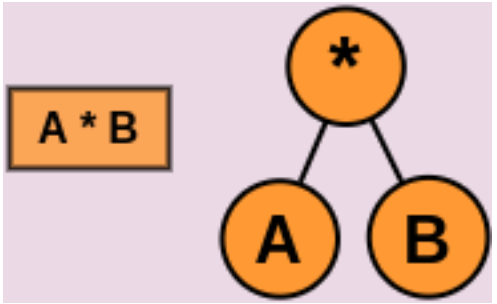


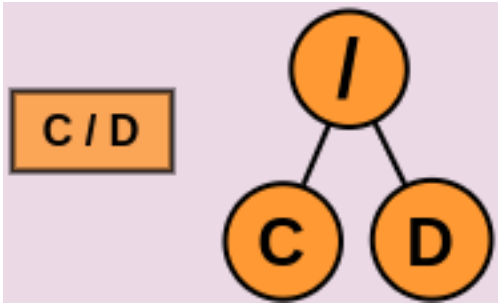
Construction of Expression Tree(Example) (Using Infix Expression)

- Scan the expression and according to the associativity and precedence find the operator which will be evaluated at last
- Expression: A * B + C / D
- In our example, the + operator will be evaluated in the last so keep it as the root node and divide the remaining expression into left and right subtrees



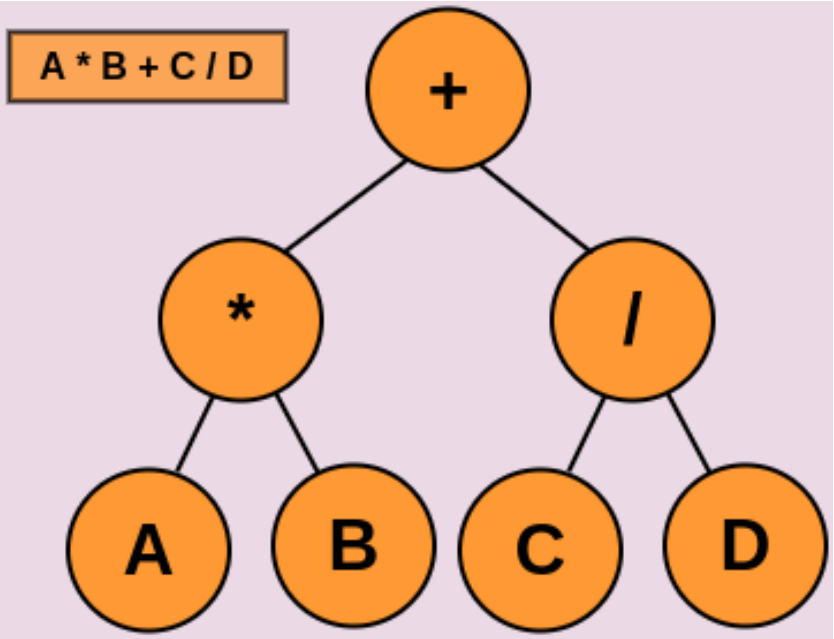
- Now solve the left and right subtree similarly





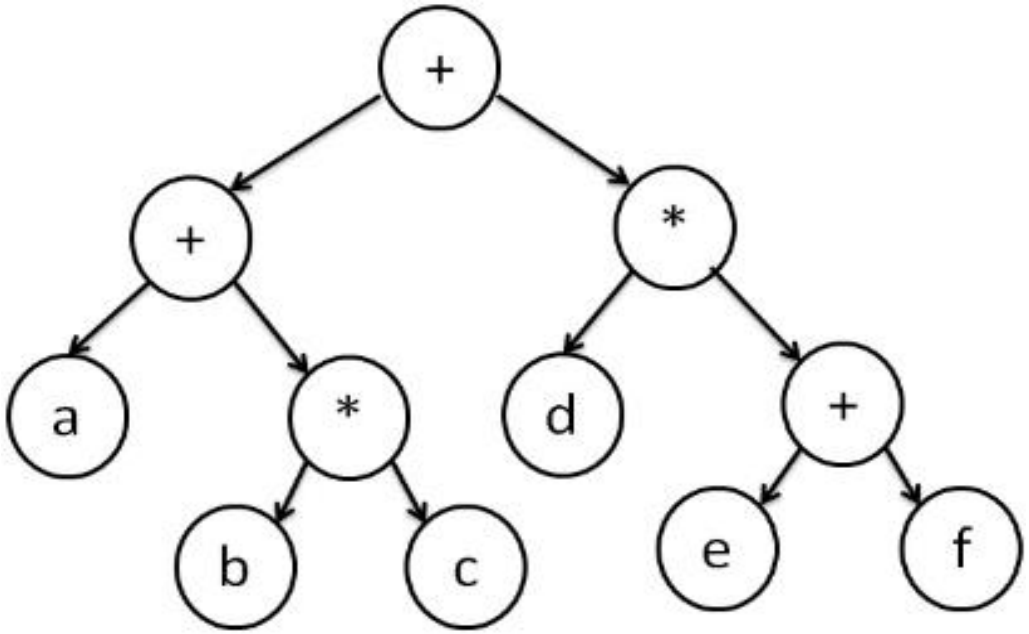
Construction of Expression Tree(Example) (Using Infix Expression)

- After solving the right and left subtree our final expression tree would become like the image given below

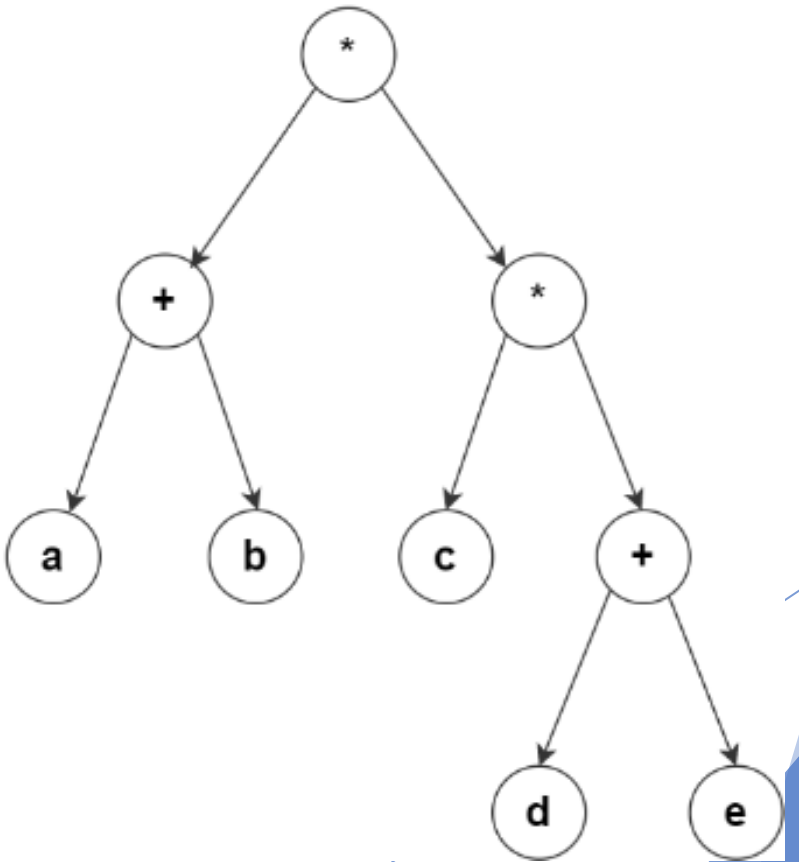


Construction of Expression Tree(Example) (Using Infix Expression)

- a + (b * c) + d * (e + f)



- (a + b) * (c * (d + e))

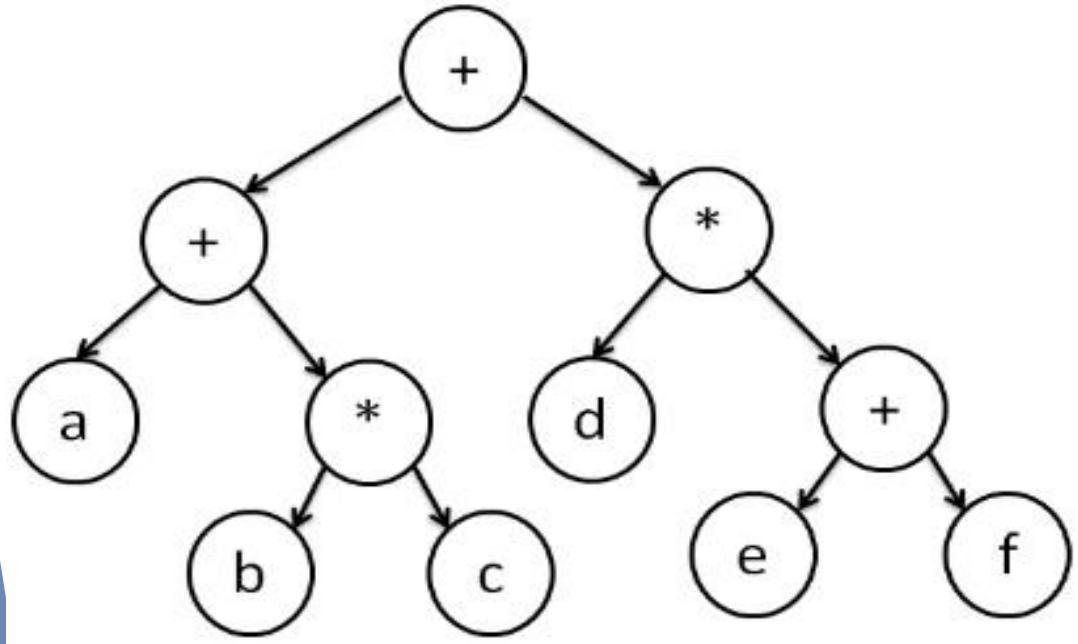


Traversal in Expression Tree

- As Expression Tree is a special kind of Binary tree therefore its traversing techniques are similar

- Same as Binary Tree Traversal refer slide 21

Expression Tree Traversal(Example)



- Infix expression:

  (a + (b * c)) + (d * (e + f))

- Prefix Expression:

  + + a * b c * d + e f

- Postfix Expression:

  a b c * + d e f + * +

Applications of Trees

Applications of Trees

- Store hierarchical data, like folder structure, organization structure, XML/HTML data
- Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
- B-Tree and B+ Tree : They are used to implement indexing in databases.
- Syntax Tree: Scanning, parsing , generation of code and evaluation of arithmetic expressions in Compiler design
- Huffman Tree: Huffman Coding is a technique of compressing data to reduce its size without losing any of the details
- Heap is a tree data structure which is implemented using arrays and used to implement priority queues

Why use Huffman Coding(Tree)?

- Consider the below string sent over network

- Each character occupies 8 bits in ASCII
- There are a total of 15 characters in the above string
- Thus, a total of 8 * 15 = 120 bits are required to send this string

- We use the Huffman Coding technique to compress the string to a smaller size

Huffman Coding

- Huffman coding is a lossless data compression algorithm
- The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters
- The most frequent character gets the smallest code and the least frequent character gets the largest code
- Huffman coding first creates a tree using the frequencies of the character and then generates code for each character
- Once the data is encoded, it has to be decoded. Decoding is done using the same tree
- Huffman Coding prevents any ambiguity in the decoding process using the concept of **prefix code** i.e., a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property

Huffman Algorithm

- Step 1: Create a leaf node for each character. Add the character and its weight or frequency of occurrence to the priority queue
- Step 2: Repeat Steps 3 to 5 while the total number of nodes in the queue is greater than 1
- Step 3: Remove two nodes that have the lowest weight (or highest priority)
- Step 4: Create a new internal node by merging these two nodes as children and with weight equal to the sum of the two nodes' weights
- Step 5: Add the newly created node to the queue

Huffman Coding(Tree) Sample

- Consider the below string sent over network

- 1. Calculate the frequency of each character in the string

- 2. Sort the characters in increasing order of the frequency. These are stored in a priority queue *Q*

- 3. Make each unique character as a leaf node

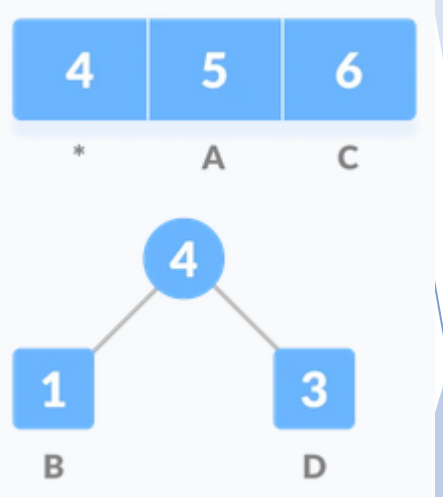
Huffman Coding(Tree) Sample…

- 4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies

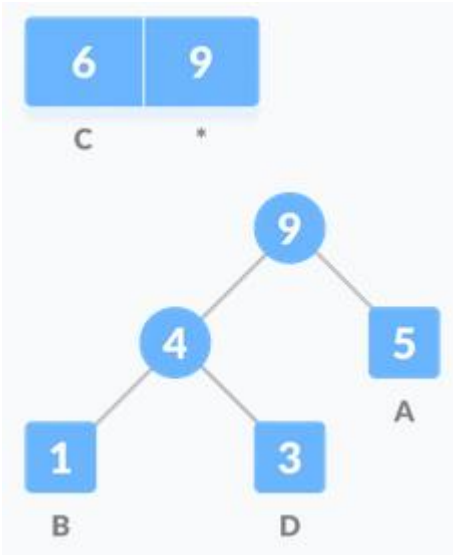
- 5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies

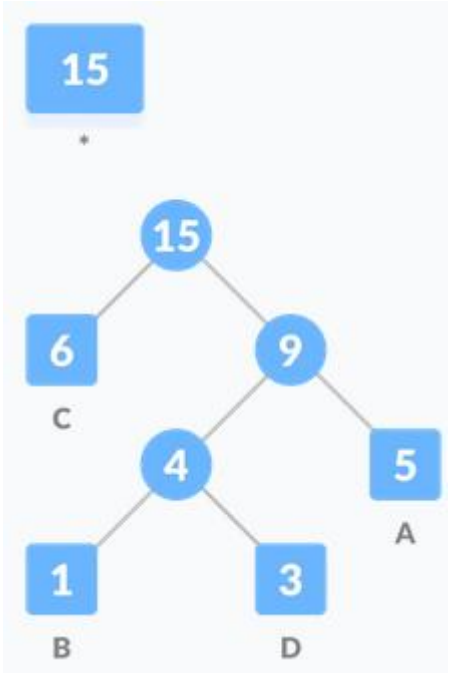
- 6. Insert node z into the tree

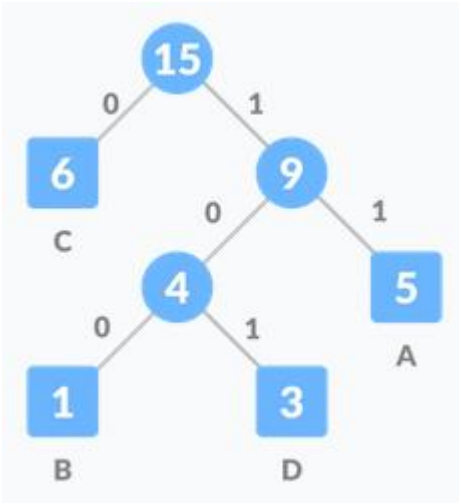
- 7. Repeat steps 3 to 5 for all the characters



Huffman Coding(Tree) Sample…







- 8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge

Huffman Coding(Tree) Sample…

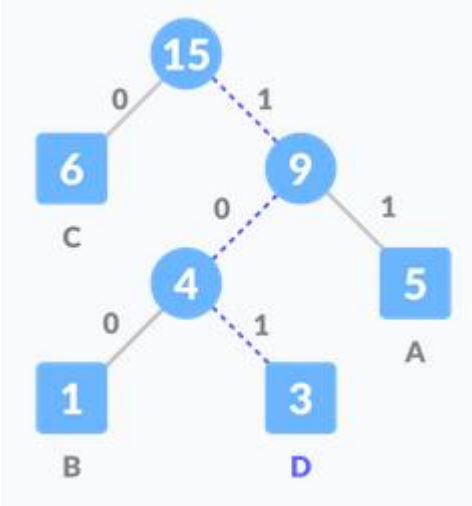
- For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below

Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

- Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28 = 75

Decoding the Huffman code

- For decoding the code, we can take the code and traverse through the tree to find the character
- Let 101 is to be decoded, we can traverse from the root as in the figure below



Multi-way Trees:
M-way Search Trees, B Tree, B+ Tree, Trie

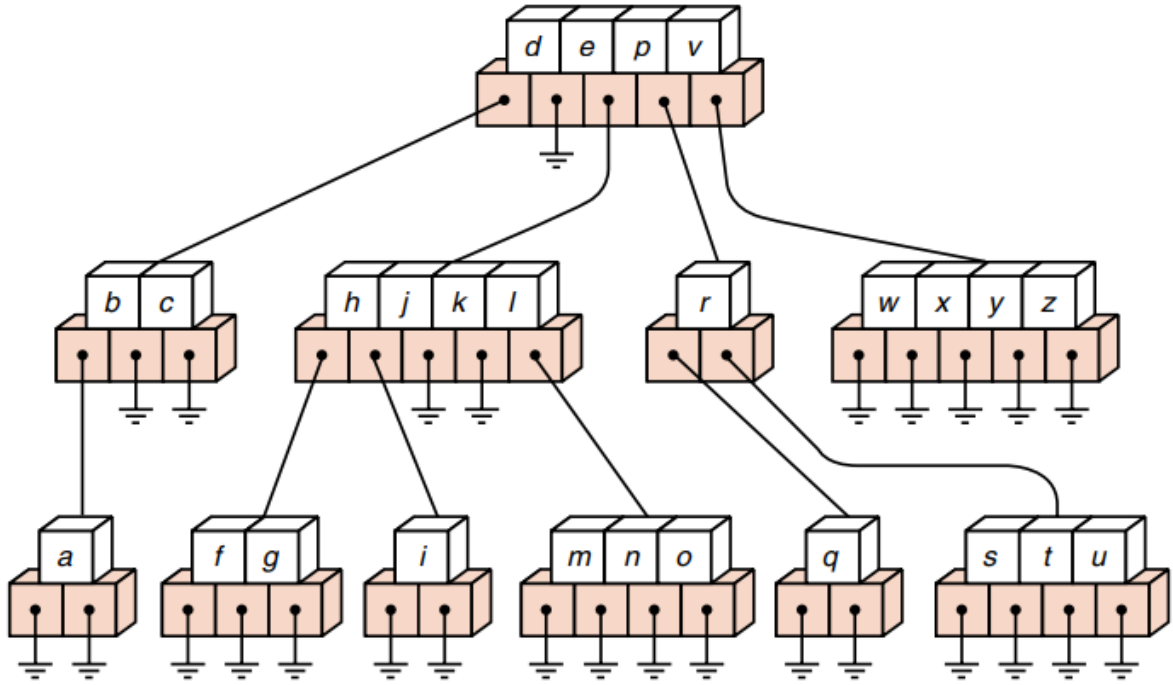
Multi-way Trees

- A main problem with binary tress is that the search path may become deep
- A multiway tree is a tree that can have more than two children
- A multiway tree of **order m** (or an m-way tree) is one in which a tree can have **m** children
- The nodes in an m-way tree will be made up of key fields, in this case m-1 key fields, and m pointers to children
- In short no of child/pointers of a node = m and keys/values of node = m-1
- Main uses of m-way trees are external file systems (ISAM, VSAM) and spell checkers

M-way Search Trees

What is M-way Search Trees?

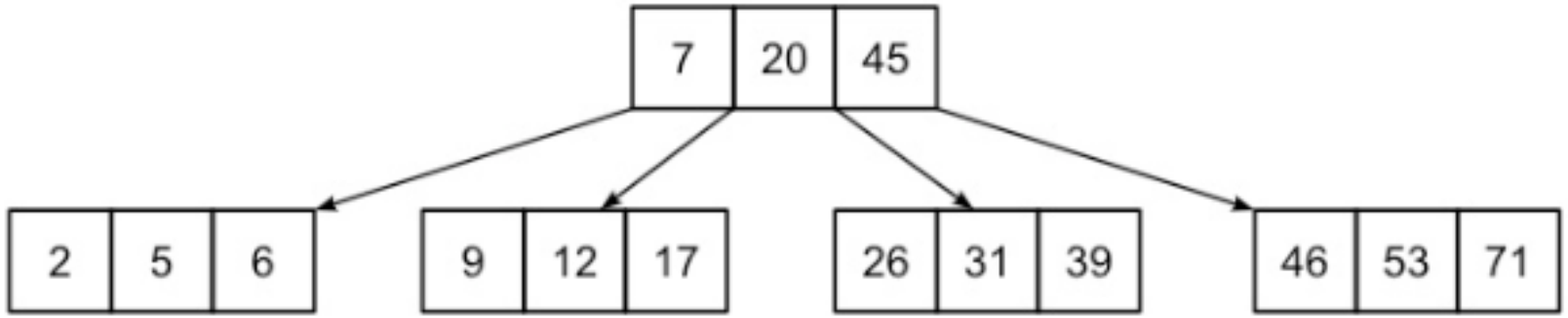
- The m-way search trees are multi-way trees which are generalized versions of binary trees where each node contains multiple elements
- In an m-way tree of order m, each node contains a maximum of m – 1 elements/keys and m children
- The keys in each node are in ascending order
- The keys in the first i children are smaller than the i^{th} key
- The keys in the last m-i children are larger than the i^{th} key



Traversal in M-way Search Trees

- The Inorder traversal of M-way (search) trees can be generalized in the following steps
  - Go to the left most node first and print its first element
  - Then check if there is any subtree between this element and the next element in that node. If it exists then apply the same approach on that sub-tree, else print element
  - Similarly move through all the elements in that node
  - Then apply same algorithm in the parent node and move up towards the root node recursively

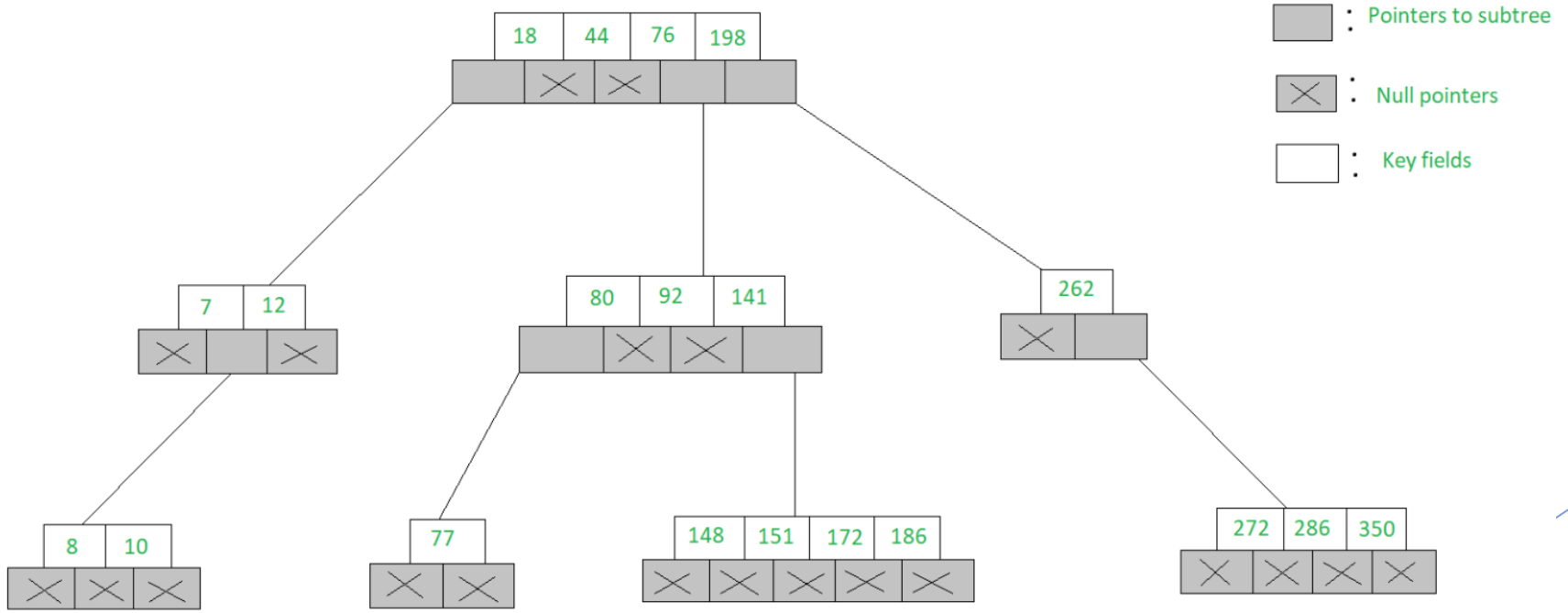
M-way Search Trees Traversal(Example)



- Output of Inorder Traversal of M-way Search Trees would be
- 2, 5, 6, 7, 9, 12,17, 20, 26, 31, 39, 45, 46, 53, 71

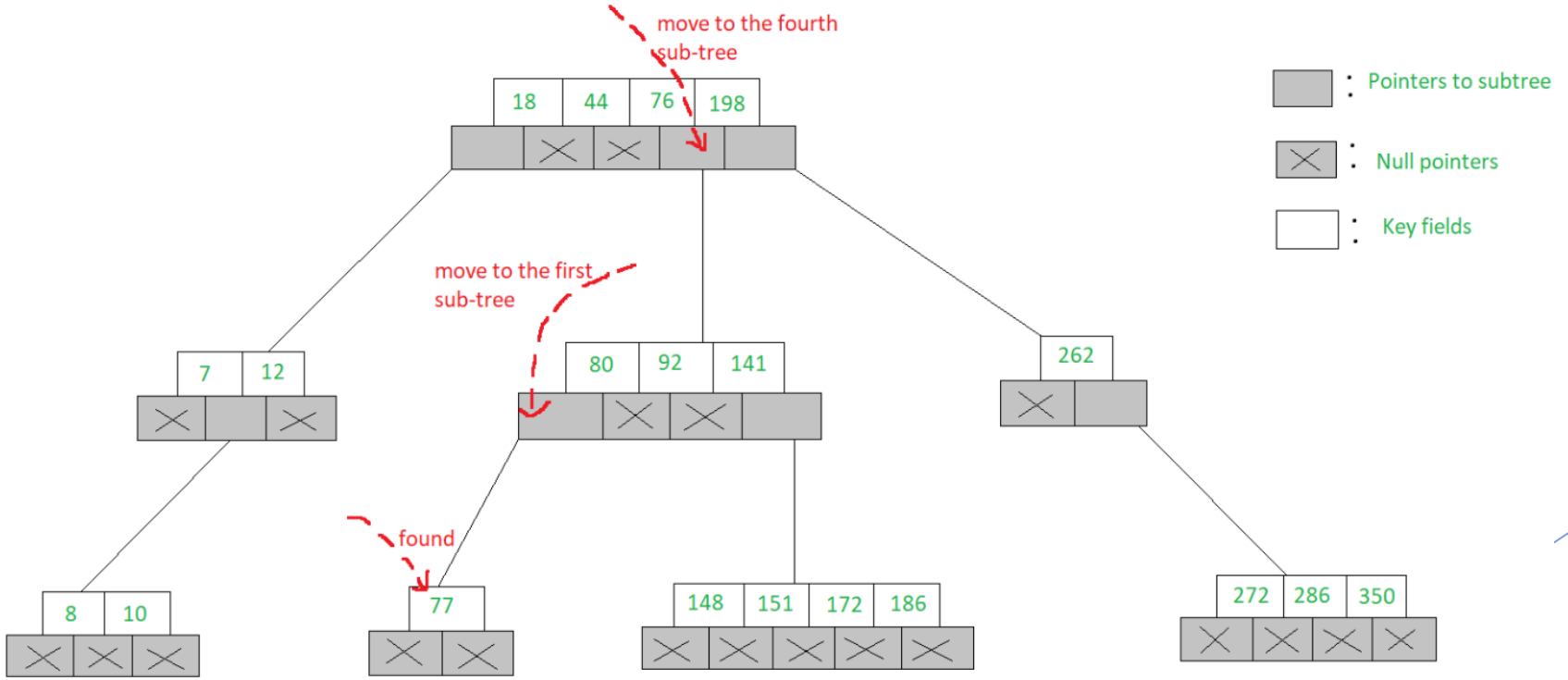
Searching in M-way Search Trees

- An example of a 5-way search tree is shown in the figure below. Let's search 77



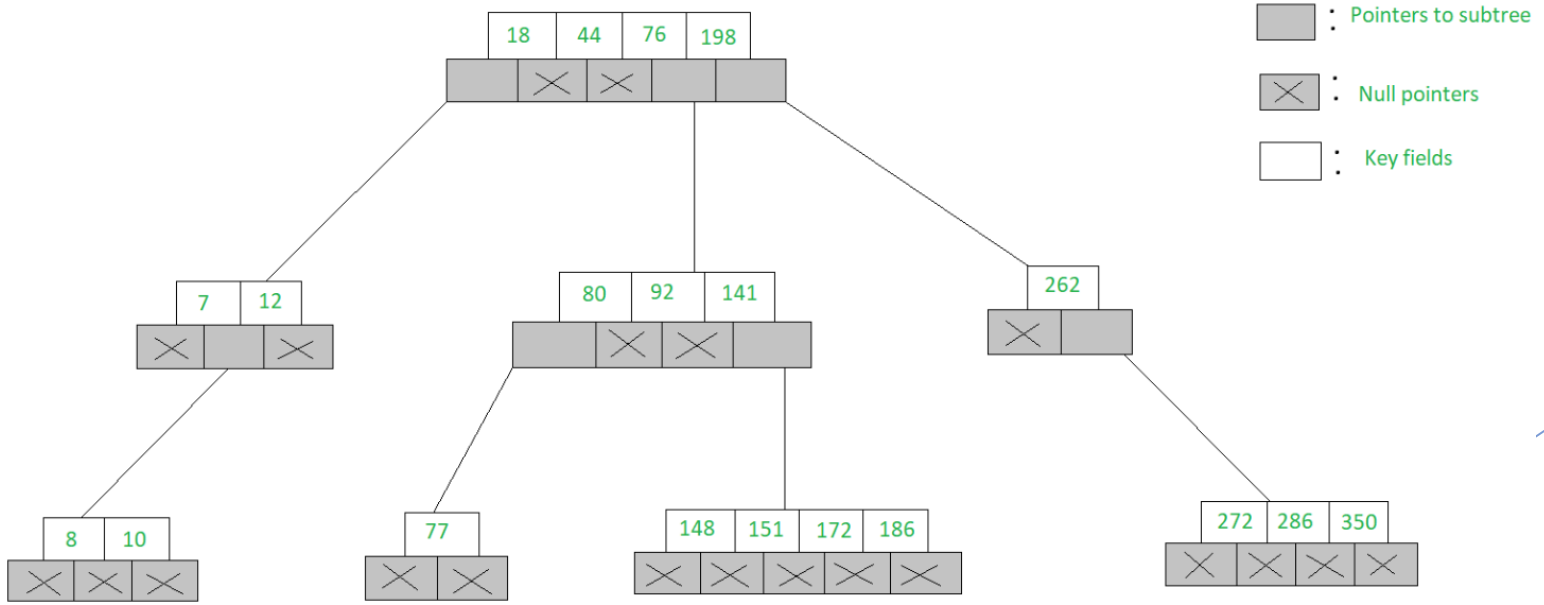
Searching in M-way Search Trees…

- Searching for a key 77 in an M-way search tree is similar to that of binary search tree



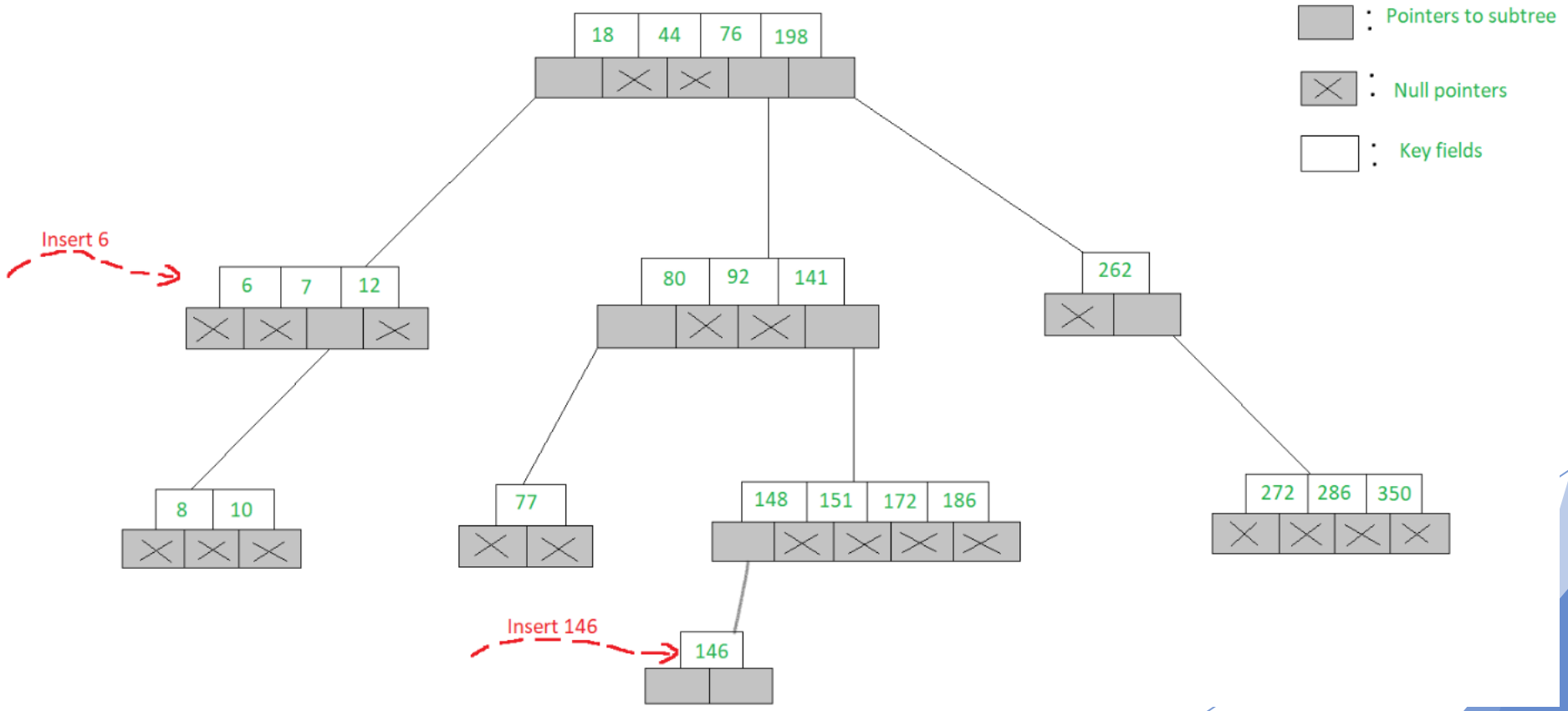
Insertion in M-way Search Trees

- The insertion in an M-way search tree is similar to binary trees but there should be no more than m-1 elements in a node
- If the node is full, then a child node will be created to insert the further elements
- Let's insert 2 new elements 6 and 146 in the below 5-way search tree



Insertion in M-way Search Trees…

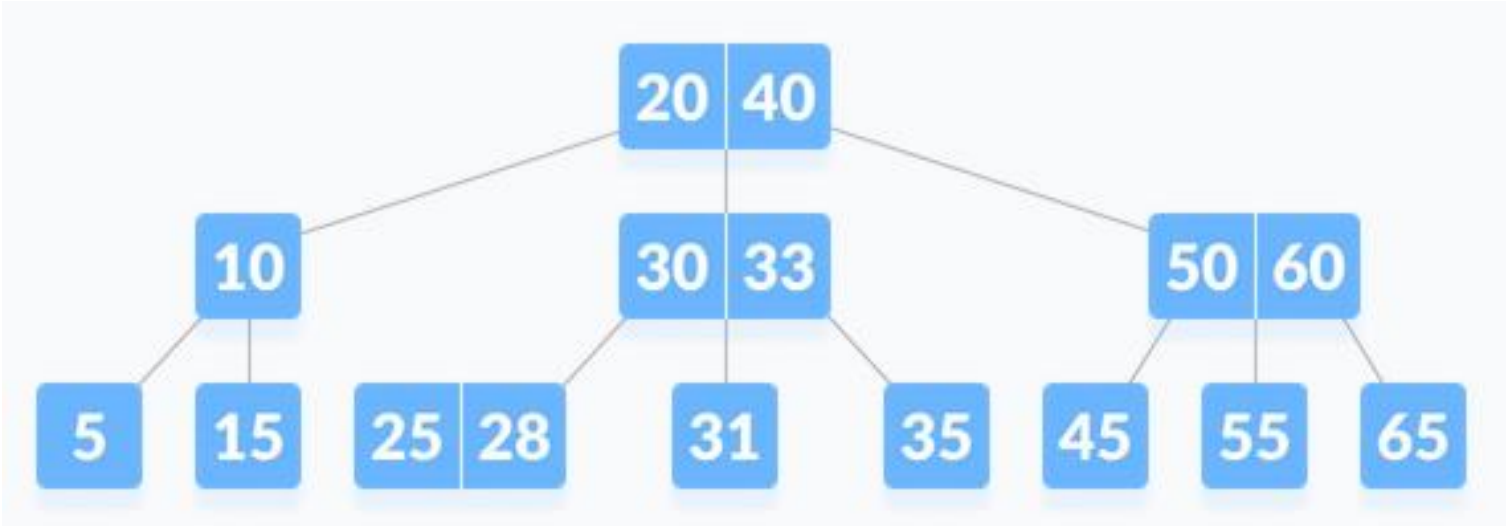
- Inserting 6 and 146 in the below 5-way search tree



B-Tree(B Tree)

What is B-Tree?

- B-tree is a special type of *self-balancing search tree* in which each node can contain more than one key and can have more than two children
- It is also known as a height-balanced m-way tree
- B Tree is a specialized m-way tree that can be widely used for disk access.
- A B-Tree of order m can have at most m-1 keys and m children



Why do you need a B-tree data structure?

- The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk accesses

- Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large and the access time increases

- B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses

B-tree Properties

- For each node *x*, the keys are stored in increasing order
- In each node, there is a boolean value *x.leaf* which is true if *x* is a leaf
- If *m* is the order of the tree, each internal node can contain at most *m - 1* keys along with a pointer to each child
- Each node except root can have at most *m* children and at least *m/2* children
- All leaves have the same depth (i.e. height-h of the tree)
- The root has at least 2 children and contains a minimum of 1 key

Search Operation in B-Tree

1) Starting from the root node, compare k with the first key of the node

   If *k* = *the first key of the node*, return the node and the index

2) If *k.leaf* = *true*, return *NULL* (i.e., not found)

3) If *k* < *the first key of the root node*, search the left child of this key recursively

4) If there is more than one key in the current node and *k* > *the first key*, compare *k* with the next key in the node

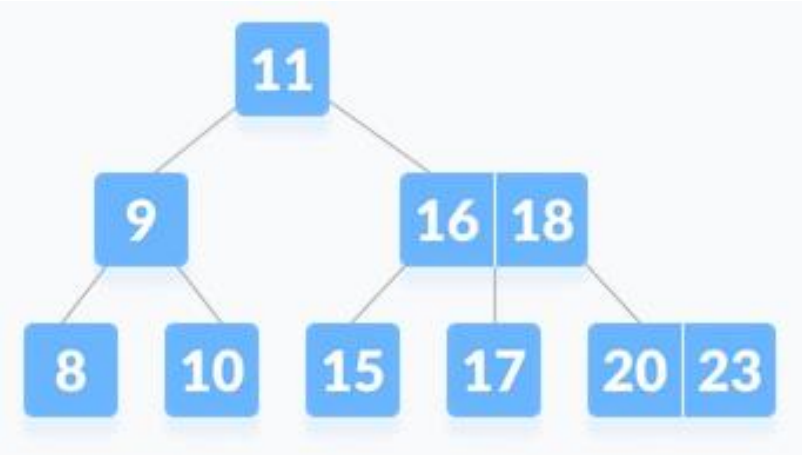
   If *k* < *next key*, search the left child of this key (i.e., *k* lies in between the first and the second keys)

   Else, search the right child of the key

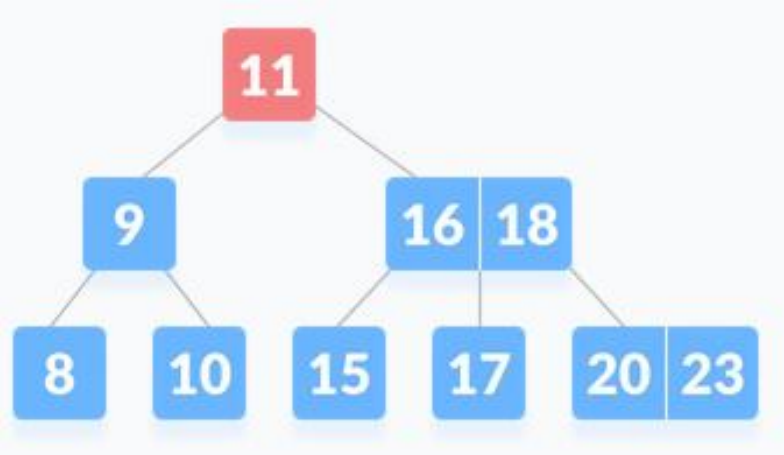
5) Repeat steps 1 to 4 until the leaf is reached

B-Tree Search Example

- Let us search key *k = 17* in the tree below of degree 3

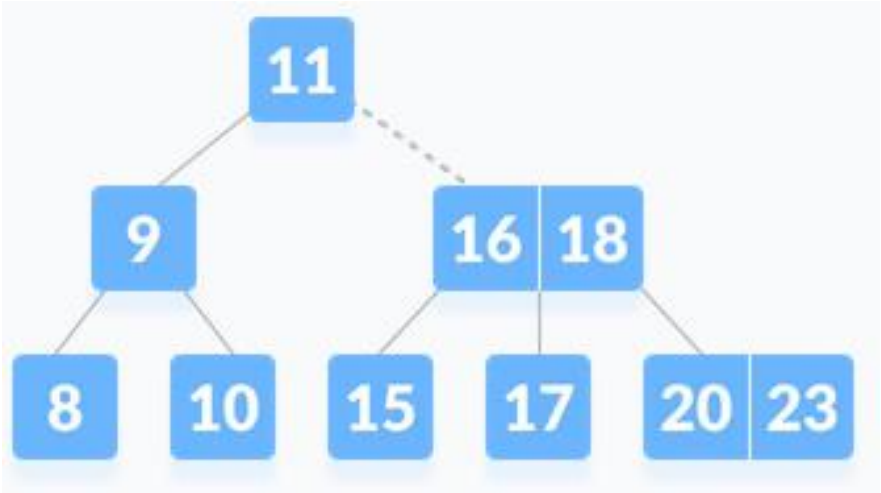


- *k* is not found in the root so, compare it with the root key

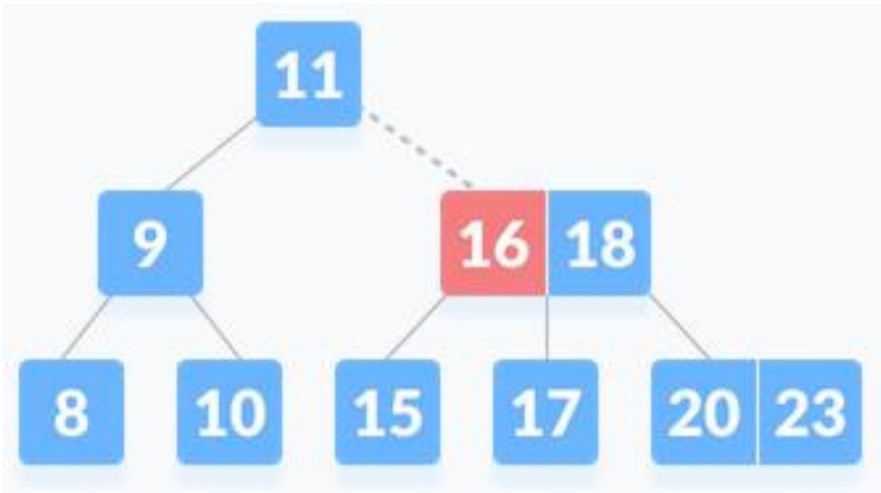


B-Tree Search Example…

- Since *k* > *11*, go to the right child of the root node

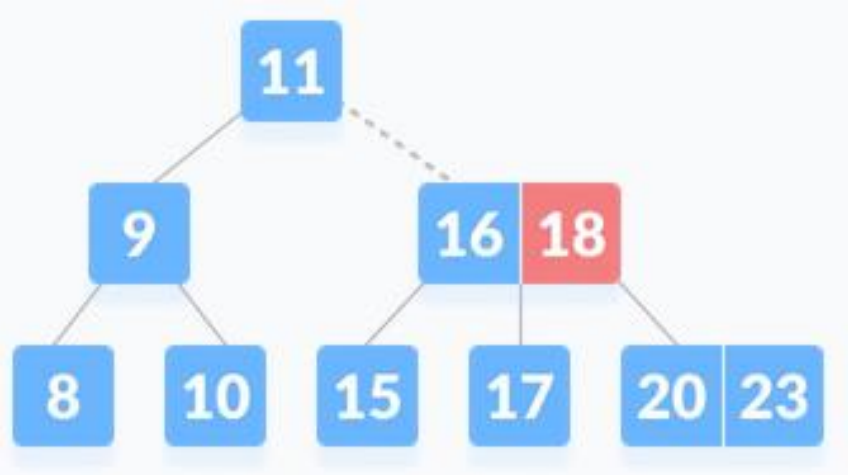


- Compare *k* with 16. Since *k* > *16*, compare *k* with the next key 18

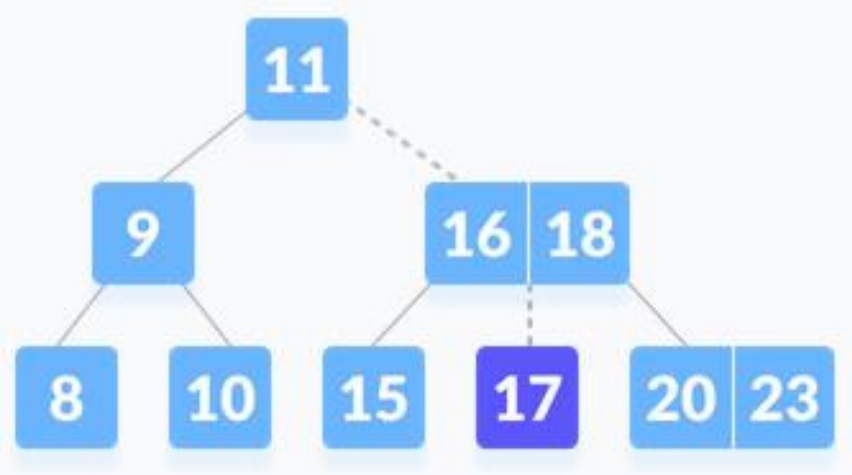


B-Tree Search Example…

- Since *k* < *18*, *k* lies between 16 and 18. Search in the right child of 16 or the left child of 18



- *k* is found



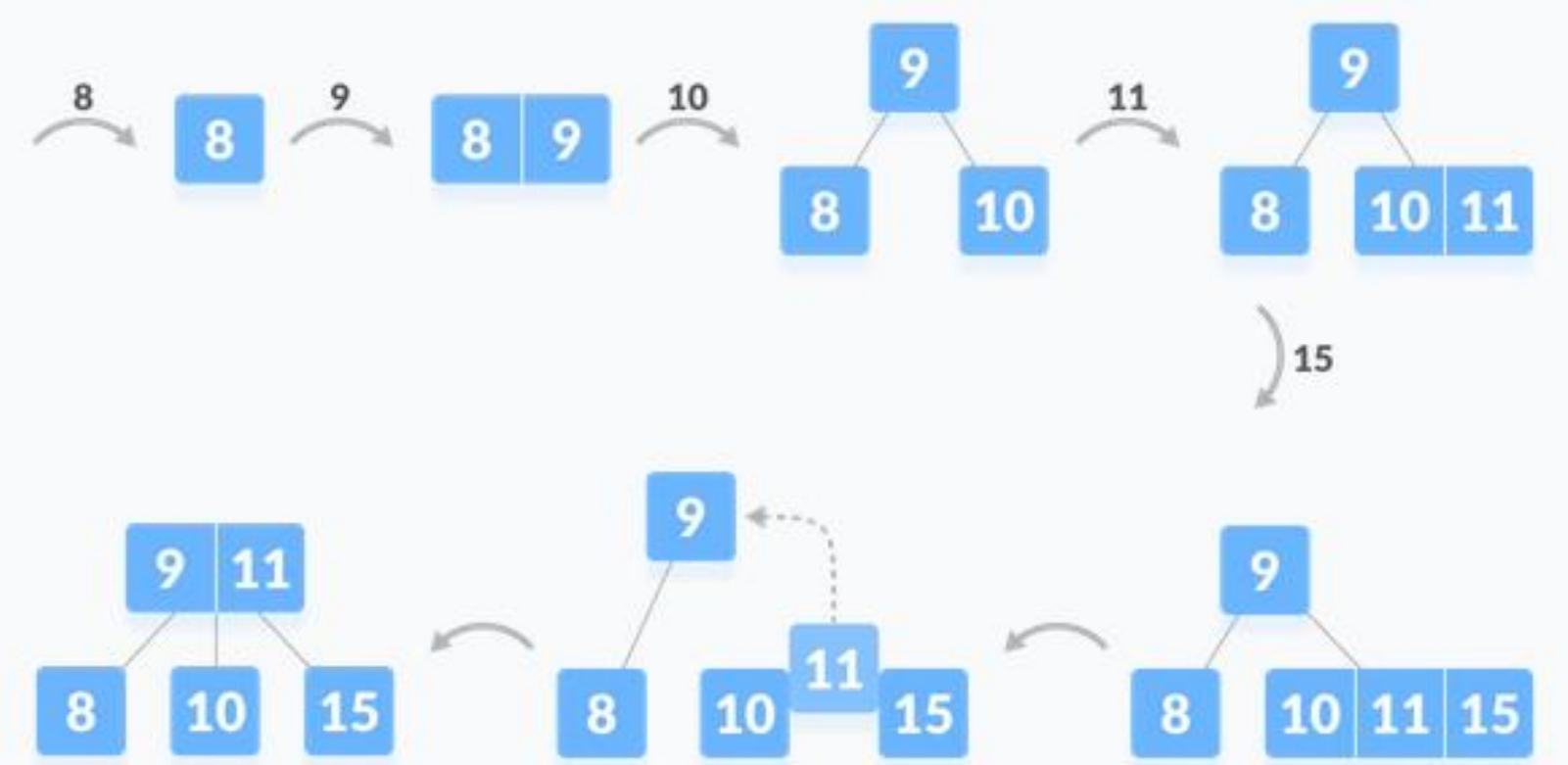
Insertion in B-Tree

- Inserting an element on a B-tree consists of two events: **searching the appropriate node** to insert the element and **splitting the node** if required. Insertion operation always takes place in the bottom-up approach

1) If the tree is empty, allocate a root node and insert the key
2) Update the allowed number of keys in the node
3) Search the appropriate node for insertion
4) If the node is full, follow the steps below
5) Insert the elements in increasing order
6) Now, there are elements greater than its limit. So, split at the median
7) Push the median key upwards and make the left keys as a left child and the right keys as a right child
8) If the node is not full, follow the steps below
9) Insert the node in increasing order

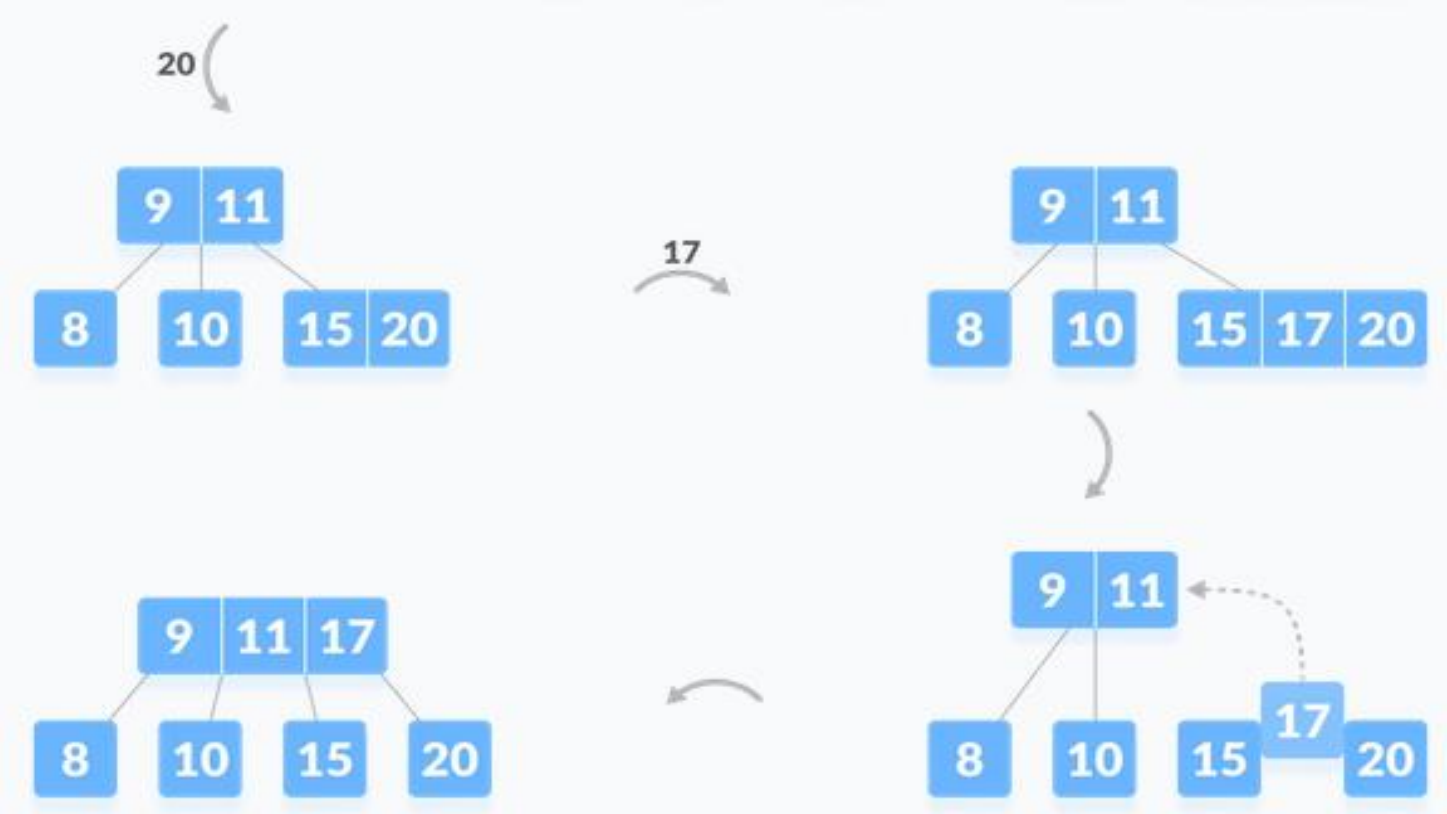
B-Tree Insertion Example

- The elements to be inserted are 8, 9, 10, 11, 15, 20, 17 in 3-way B tree



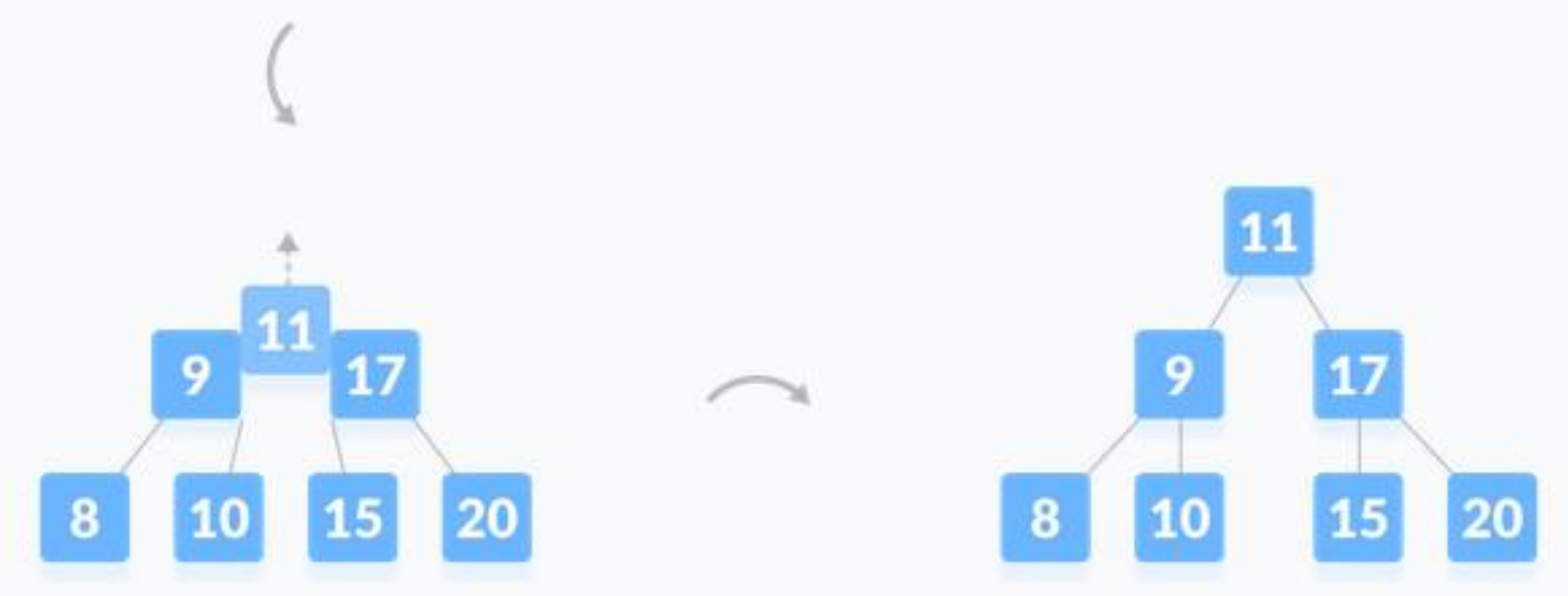
B-Tree Insertion Example…

- The elements to be inserted are 8, 9, 10, 11, 15, 20, 17 in 3-way B tree



B-Tree Insertion Example…

- The elements to be inserted are 8, 9, 10, 11, 15, 20, 17 in 3-way B tree



- *Note: Mention left biased & right biased for insertion in 4-way B tree*

Traversal in B-Tree

- Traversal is also similar to Inorder traversal of Binary Tree
- We start from the leftmost child, recursively print the leftmost child, then repeat the same process for the remaining children and keys
- In the end, recursively print the rightmost child

Review facts about a B-tree of degree m

- A node can have a maximum of m children. (i.e., 3)
- A node can contain a maximum of m - 1 keys. (i.e., 2)
- A node should have a minimum of [m/2] children. (i.e., 2)
- A node (except root node) should contain a minimum of [m/2] - 1 keys. (i.e., 1)
- All leaves have the same depth (i.e. height-h of the tree)

Application of B tree

- B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process

- Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case

B+ Tree

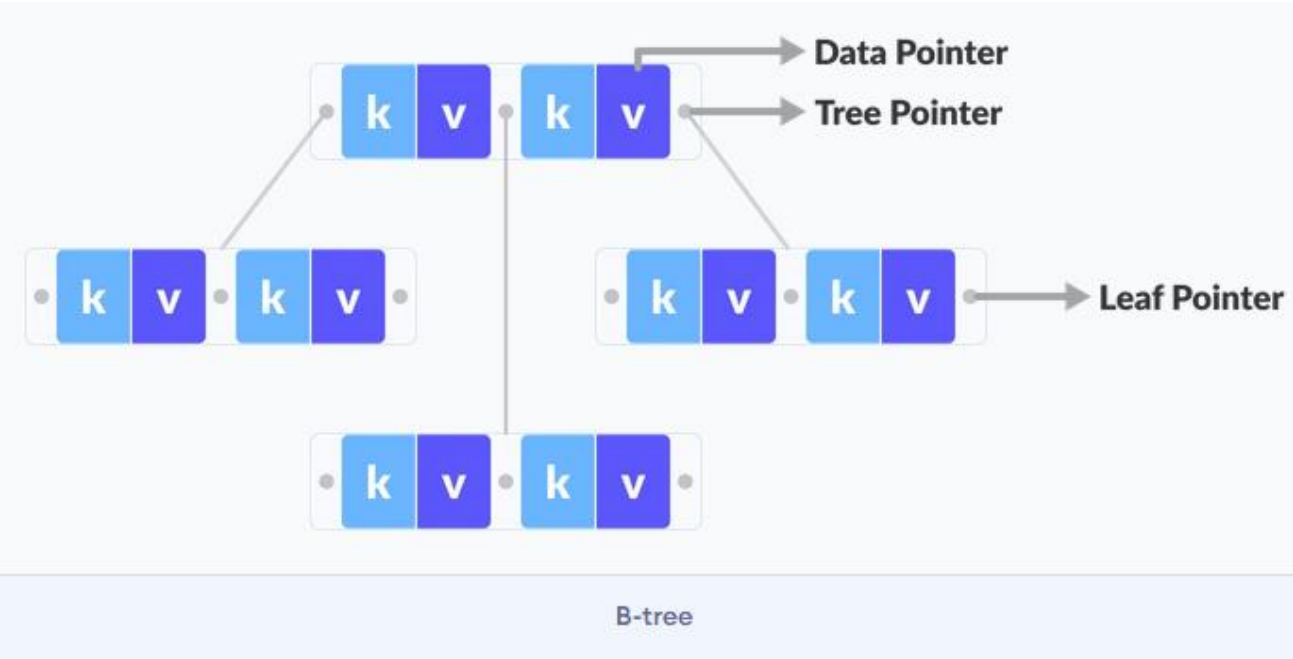
What is B+ Trees?

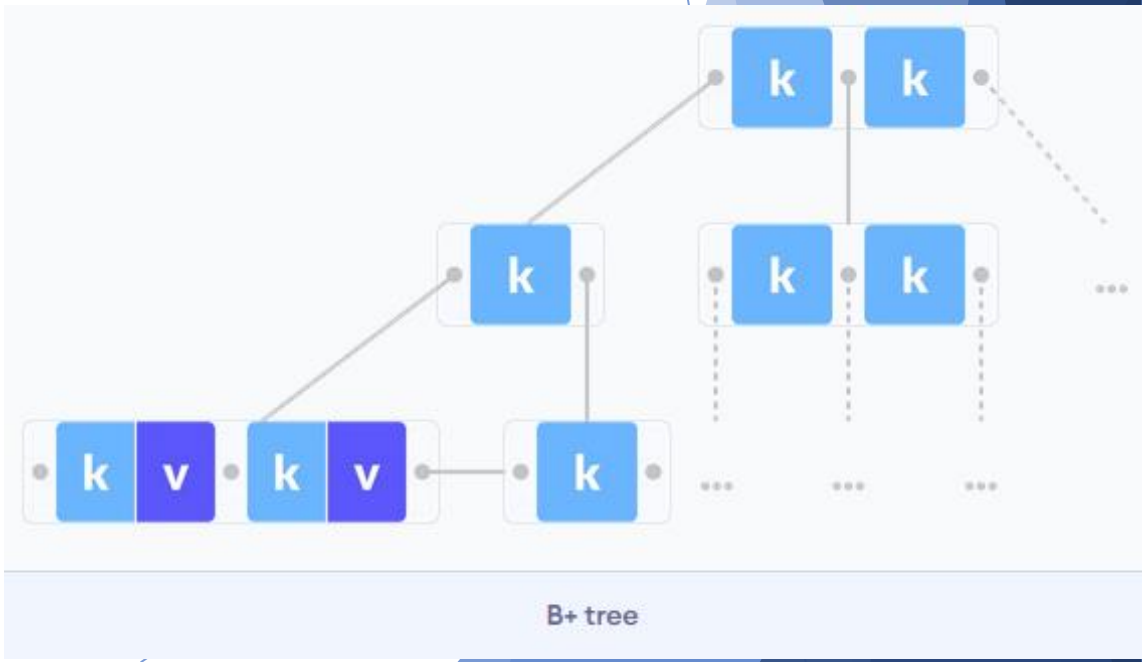
- A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a key
- While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; *only keys* are stored in the interior nodes
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient
- Typically, B+ trees are used to store large amounts of data that cannot be stored in the main memory
- With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory.

What is B+ Trees?...

- B+ trees store data only in the leaf nodes. All other nodes (internal nodes) are called *index nodes* or *i-nodes* and store index values

- This allows us to traverse the tree from the root down to the leaf node that stores the desired data item





Properties of a B+ Tree

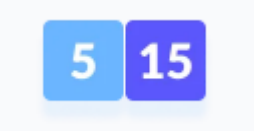
- All leaves are at the same level
- The root has at least two children
- Each node except root can have a maximum of *m* children and at least *m/2* children
- Each node can contain a maximum of *m - 1* keys and a minimum of *[m/2] - 1* keys

Insertion in B+ Tree

- The following steps are followed for inserting an element in B+ tree
  - Since every element is inserted into the leaf node, go to the appropriate leaf node
  - Insert the key into the leaf node
- **Case I**
  - If the leaf is not full, insert the key into the leaf node in increasing order
- **Case II**
  - If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way
  - Break the node at $m/2^{th}$ position
  - Add $m/2^{th}$ key to the parent node as well
  - If the parent node is already full, follow steps 2 to 3

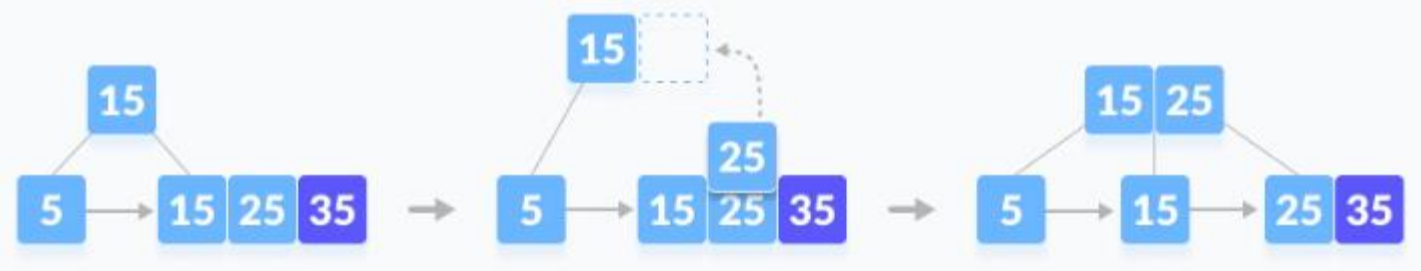
B+ Tree Insertion Example

- The elements to be inserted are 5,15, 25, 35, 45 in a 3-way B+ tree



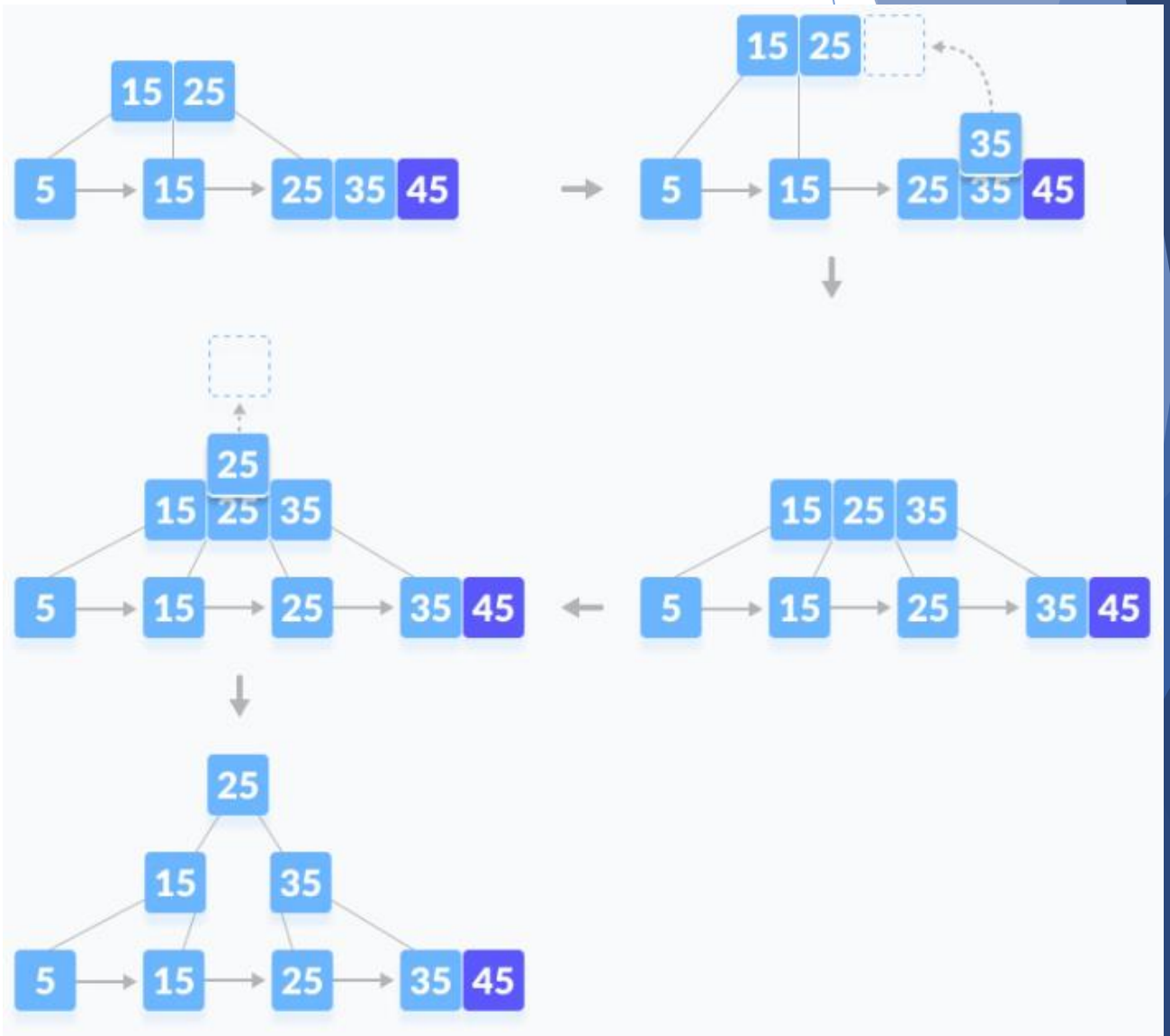
B+ Tree Insertion Example…

- The elements to be inserted are 5,15, 25, 35, 45 in a 3-way B+ tree



B+ Tree Insertion Example…

- The elements to be inserted are 5,15, 25, 35, 45 in a 3-way B+ tree



Advantages of B+ trees

Records can be fetched in equal number of disk accesses

Height of the tree remains balanced and less as compared to B tree

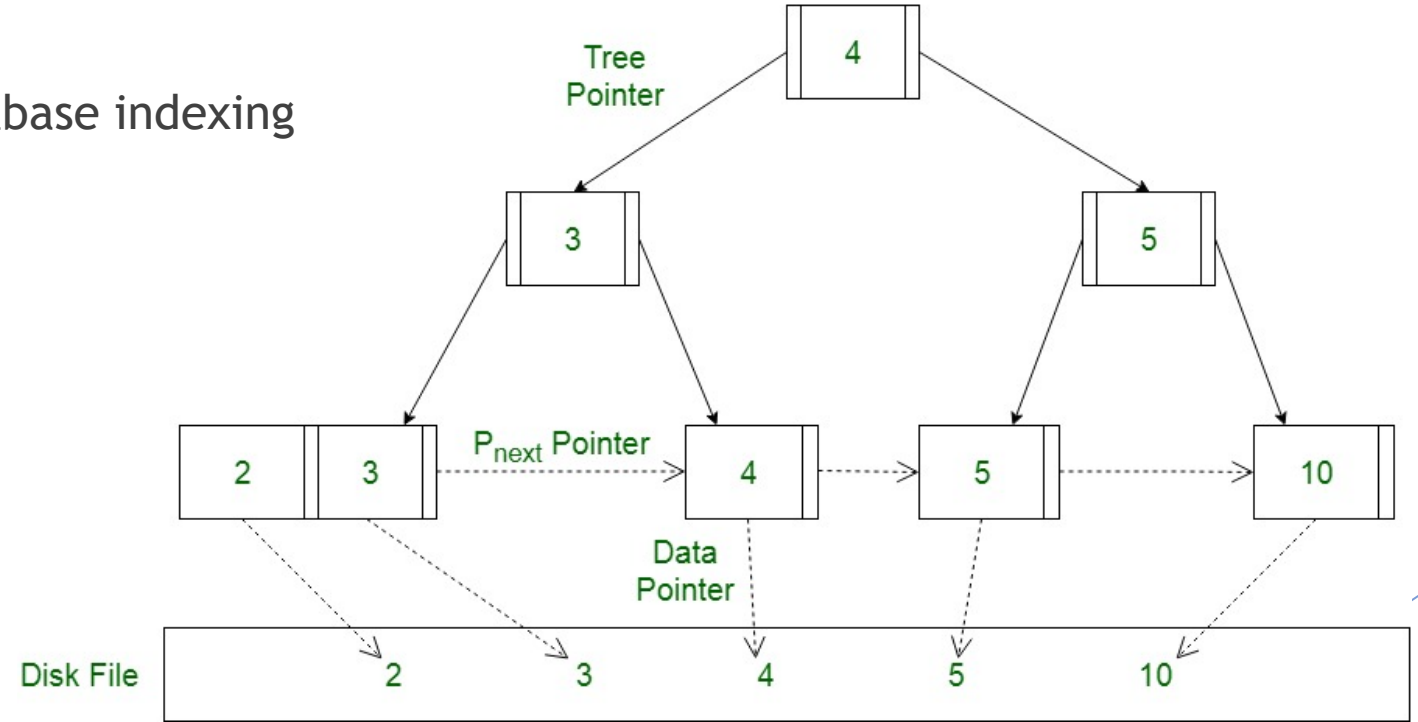
We can access the data stored in a B+ tree sequentially as well as directly

Keys are used for indexing

Faster search queries as the data is stored only on the leaf nodes

Application of B+ tree

- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)
- Database indexing



Comparison Between B Trees and B+ Trees

B Tree	B+ Tree
Search keys are not repeated	Stores redundant search key
Data is stored in internal or leaf nodes	Data is stored only in leaf nodes
Searching takes more time as data may be found in a leaf or non-leaf node	Searching data is very easy as the data can be found in leaf nodes only
Deletion of internal nodes are so complicated and time consuming	Deletion will never be a complexed process since element will always be deleted from the leaf nodes
Leaf nodes can not be linked together	Leaf nodes are linked together to make the search operations more efficient
The structure and operations are complicated	The structure and operations are simple

Trie

What is Trie?

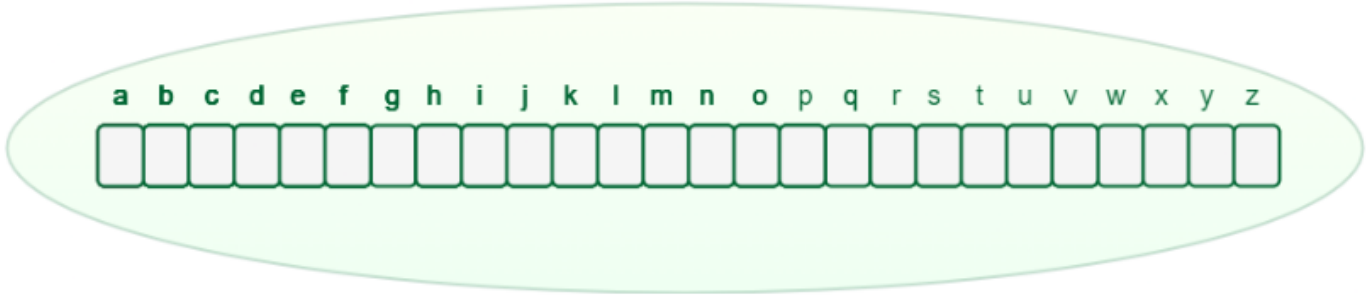
- A Trie is an advanced data structure that is sometimes also known **digital tree** or **prefix tree**
- The word Trie is derived from reTRIEval, which means finding something or obtaining it
- It is a tree that stores the data in an ordered and efficient way, and we generally use trie's to store strings
- It is known as a prefix tree as all descendants of a node have a *common prefix of the string* associated with that node
- Every Trie node consists of a *character pointer array* or hashmap and a *flag* to represent if the word is ending at that node or not

Why use Trie Data Structure?

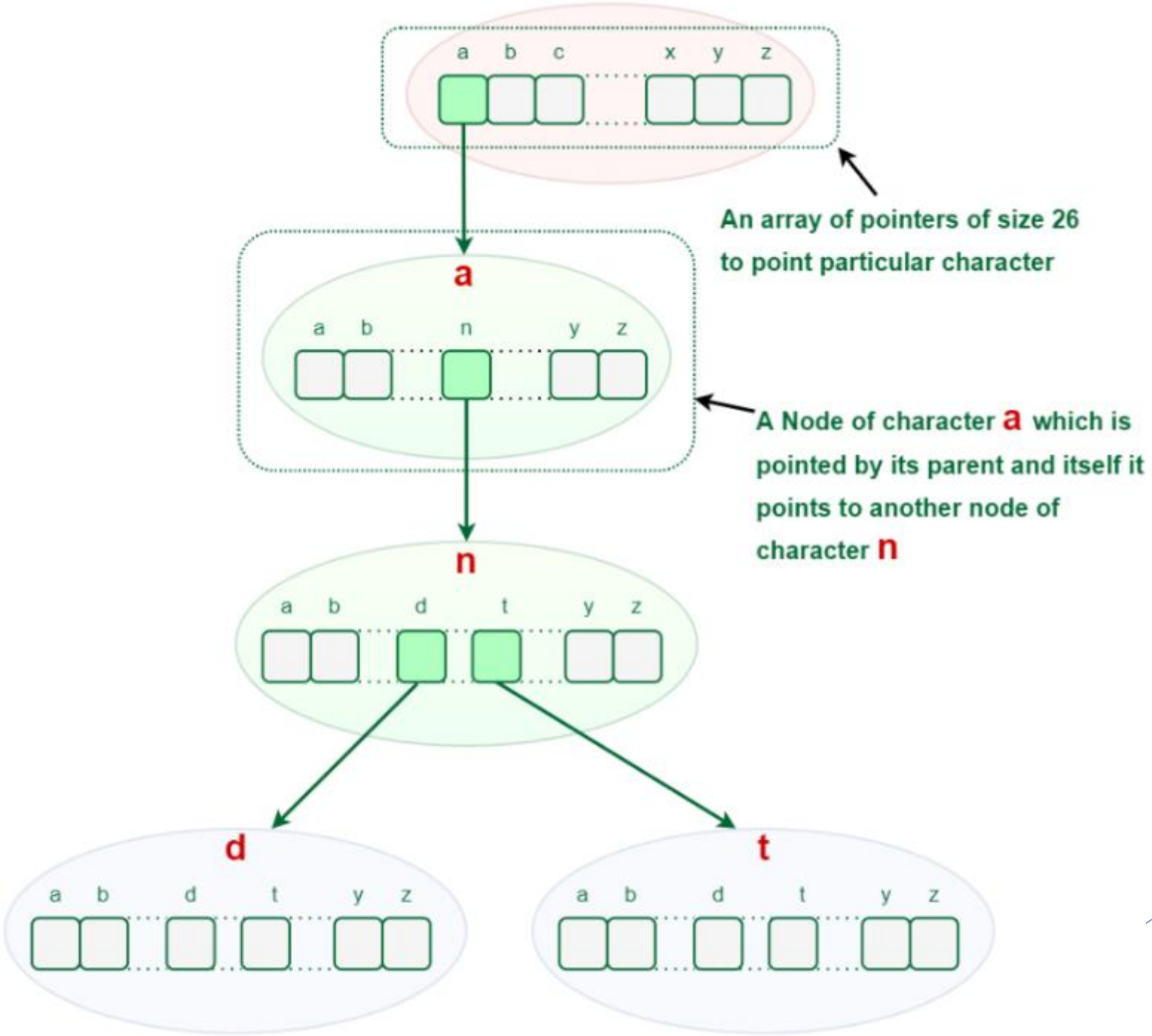
- Hash tables are generally considered one of the fastest ways to retrieve values from a data structure. Trie's are much more efficient than hash tables and also they possess several advantages over the same. mainly:
  - There *won't be any collisions* hence making the worst performance better than a hash table that is not implemented properly
  - *No need* for hash functions
  - Lookup time for a string in trie is O(k) where *k = length of the word*
  - It can take even less than O(k) time when the word is not there in a trie

Properties of a Trie

- Each node of a Trie represents a string and each edge represents a character
- Every node consists of hashmaps or an array of pointers, with each index representing a character and a flag to indicate if any string ends at the current node.
- Trie data structure can contain any number of characters including alphabets, numbers, and special characters. For our class, we will discuss strings with characters a-z. Therefore, only 26 pointers needed for every node, where the 0th index represents ‘a’ and the 25th index represents ‘z’ characters
- The key length determines Trie depth



An array of pointers inside every Trie node


a b c x y z
An array of pointers of size 26
to point particular character
a
a b n y z
A Node of character a which is
pointed by its parent and itself it
points to another node of
character n
n
a b d t y z
d
a b d t y z
t
a b d t y z

Representation of Trie Node

- Every Trie node consists of a character pointer array or hashmap and a flag to represent if the word is ending at that node or not
- But if the words contain only lower-case letters (i.e. a-z), then we can define Trie Node with an array instead of a hashmap

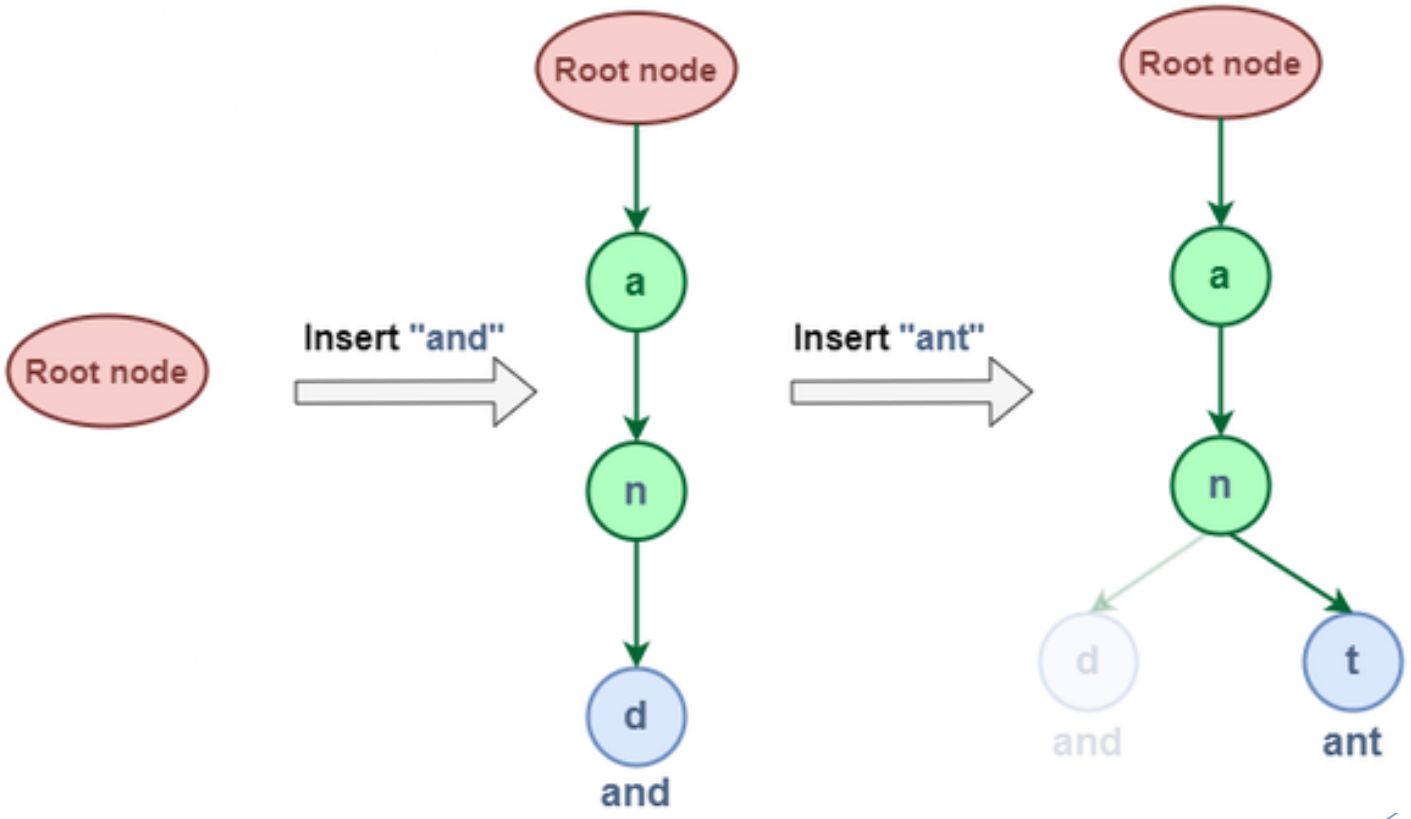
```
struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];

    // This will keep track of number of strings that are
    // stored in the Trie from root node to any Trie node
    int wordCount = 0;
};
```

Insertion in Trie

- Every character of the input key is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next-level trie nodes
- The key character acts as an index to the array children
- If the input key is new or an extension of the existing key, construct non-existing nodes of the key, and mark the end of the word for the last node by incrementing the wordCount of the last node
- If the input key is a prefix of the existing key in Trie, Simply mark the last node of the key as the end of a word

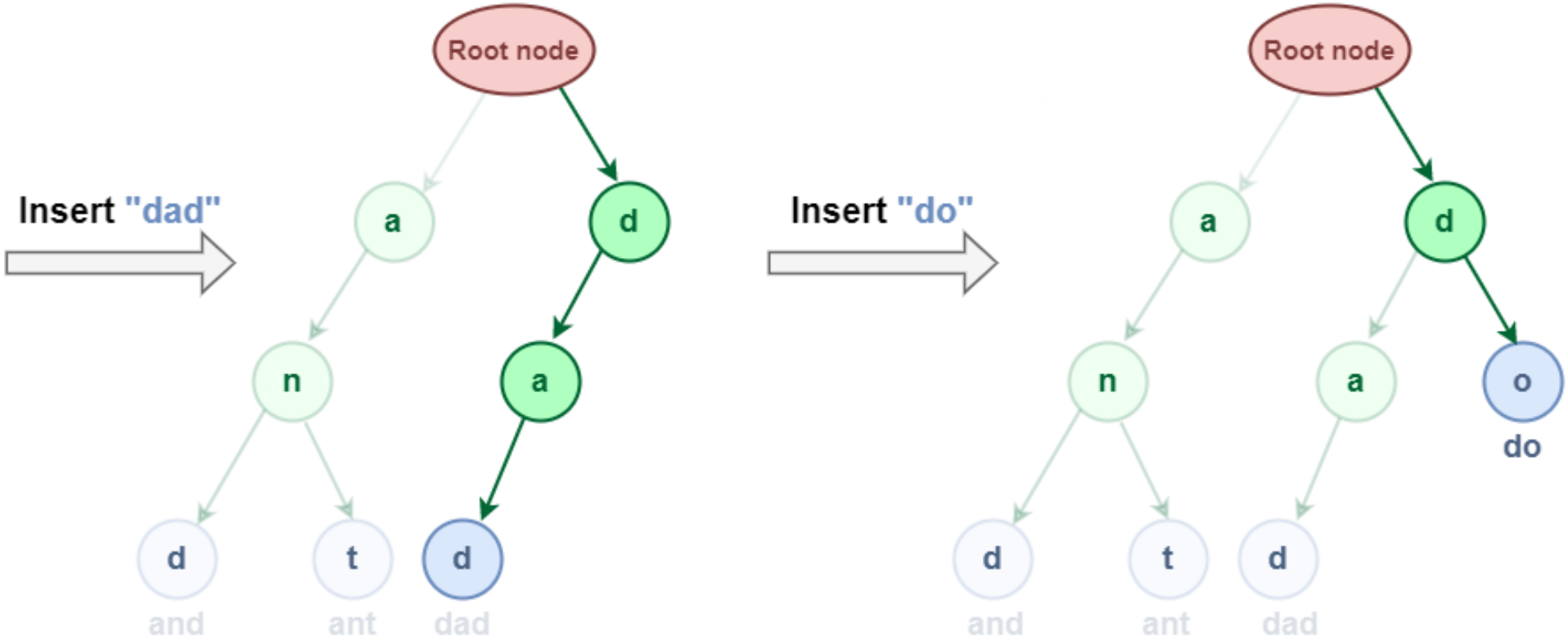
Insertion in Trie(Example)

- Let us try to Insert “and” & “ant” in this Trie



Insertion in Trie(Example)

- Let us try to Insert “dad” & “do” in this Trie

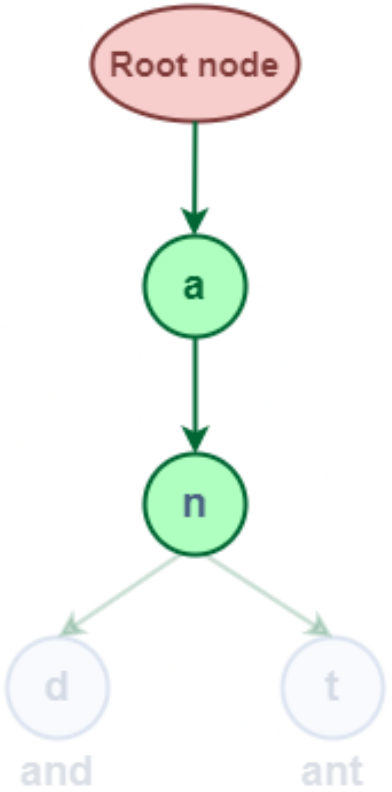


Searching in Trie

- Search operation in Trie is performed in a similar way as the insertion operation but the only difference is that whenever we find that the array of pointers in curr node does not point to the current character of the word then **return false** instead of creating a new node for that current character of the word
- There are two search approaches in the Trie data structure:
  - Find whether the given word exists in Trie
  - Find whether any word that starts with the given prefix exists in Trie
- There is a similar search pattern in both approaches
- The first step in searching a given word in Trie is to convert the word to characters and then compare every character with the trie node from the root node
- If the current character is present in the node, move forward to its children. Repeat this process until all characters are found.

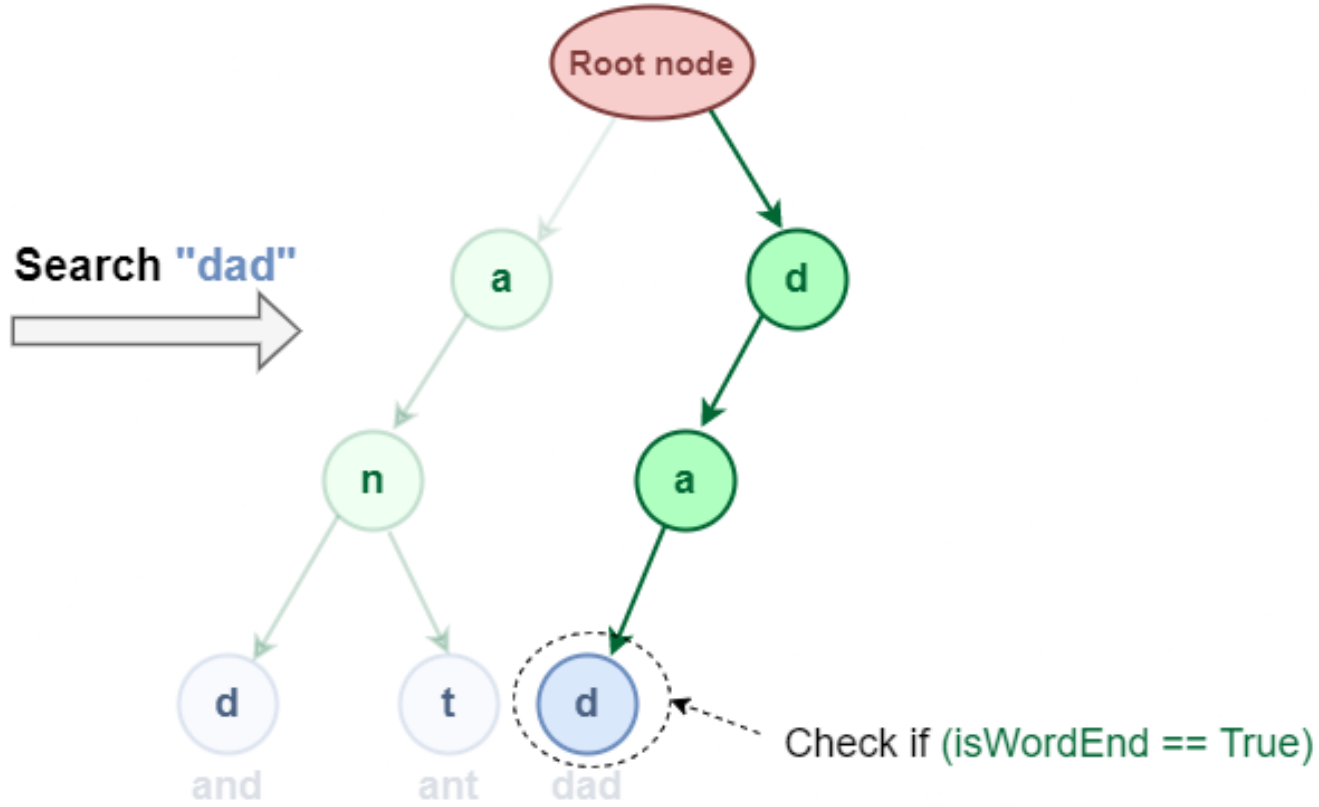
Searching Prefix in Trie

- Search for the prefix “an” in the Trie Data Structure



Searching Complete Word in Trie

- It is similar to prefix search but additionally, we have to check if the word is ending at the last character of the word or not



Deletion in Trie

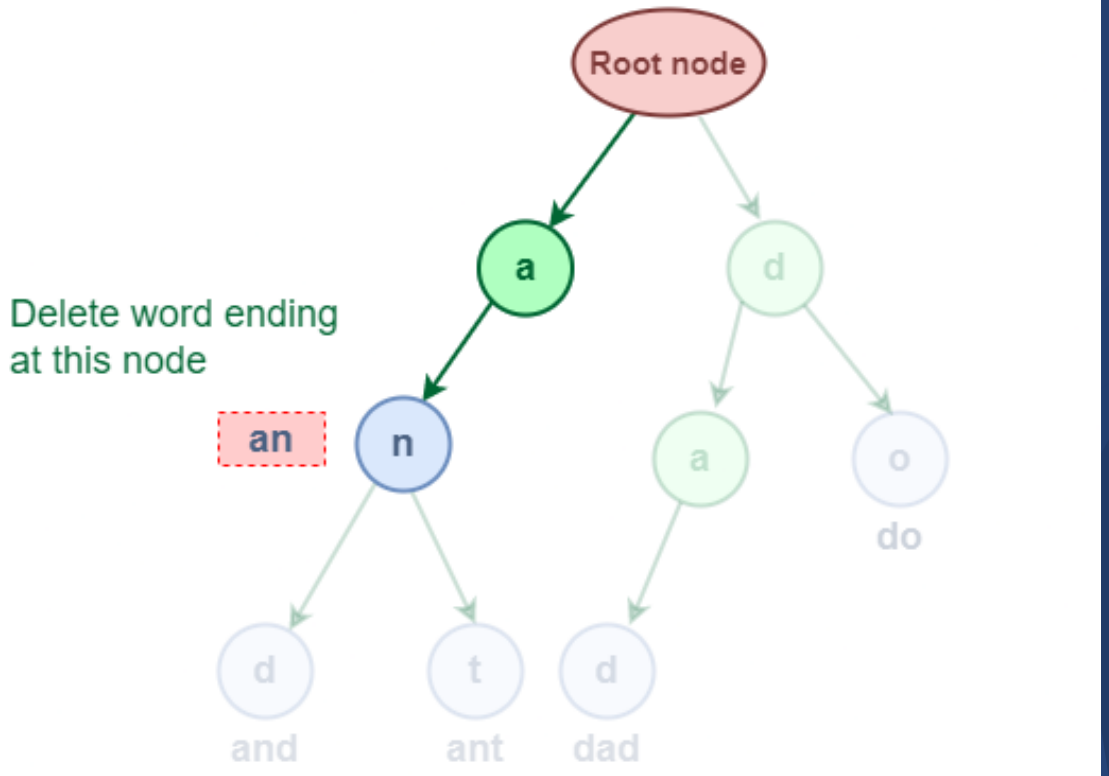
- This operation is used to delete strings from the Trie data structure

- There are three cases when deleting a word from Trie
  1) The deleted word is a prefix of other words in Trie
  2) The deleted word shares a common prefix with other words in Trie
  3) The deleted word does not share any common prefix with other words in Trie

1. The deleted word is a prefix of other words in Trie

- As shown in the following figure, the deleted word “an” share a complete prefix with another word “and” and “ant”

- An easy solution to perform a delete operation for this case is to just *decrement the wordCount by 1* at the ending node of the word

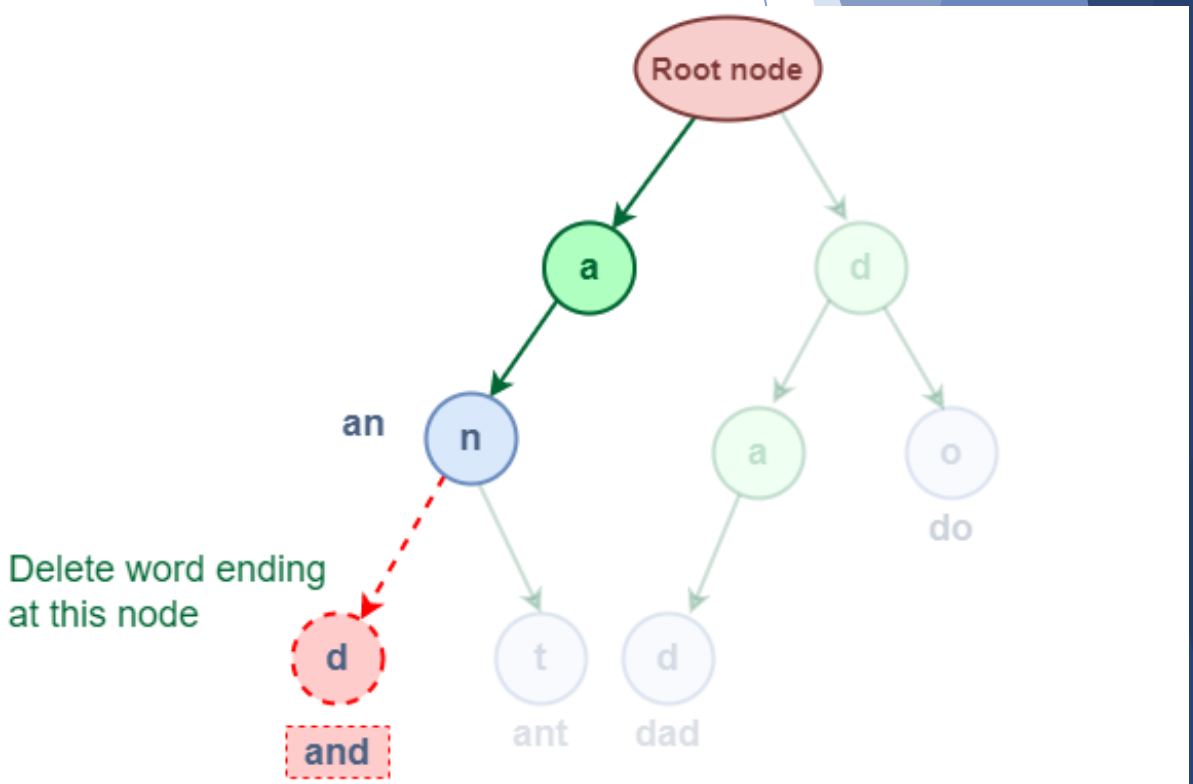


Case 1: The deleted word is prefix of other words

2. The deleted word shares a common prefix with other words in Trie

- As shown in the following figure, the deleted word “and” has some common prefixes with other words “ant”. They share the prefix “an”

- The solution for this case is to delete all the nodes *starting from the end of the prefix* to the *last character of the given word*

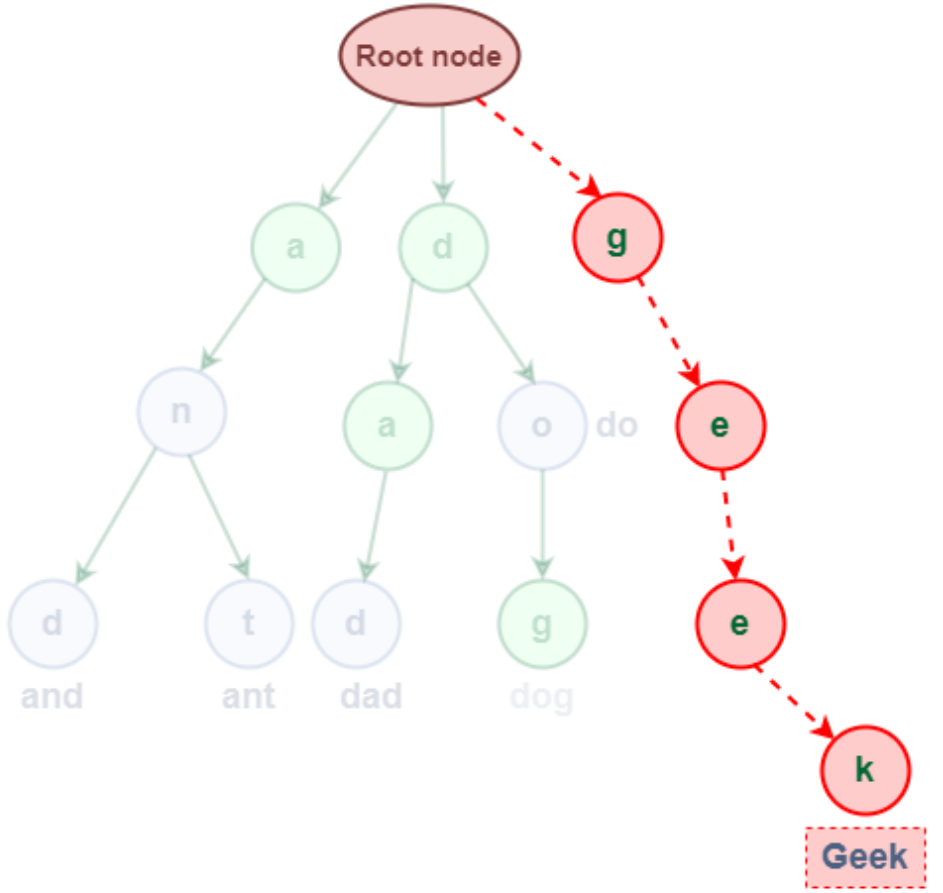


Case 2: The deleted word share common prefix with other words

3. The deleted word does not share any common prefix with other words in Trie

- As shown in the following figure, the word “geek” does not share any common prefix with any other words

- The solution for this case is just to delete all the nodes



Case 3: The deleted word does not share any common prefix with other words

Applications of Trie

- **Autocomplete Feature:** Autocomplete provides suggestions based on what you type in the search box. Trie data structure is used to implement autocomplete functionality

- **Spell Checkers:** If the word typed does not appear in the dictionary, then it shows suggestions based on what you typed.
  Trie stores the data dictionary and makes it easier to build an algorithm for searching the word from the dictionary and provides the list of valid words for the suggestion

- **Longest Prefix Matching Algorithm(Maximum Prefix Length Match):** This algorithm is used in networking by the routing devices in IP networking.
  To speed up the lookup process, Multiple Bit trie schemes were developed that perform the lookups of multiple bits faster


Any
Questions?