Trees and hierarchical orders, ordered trees, Search trees

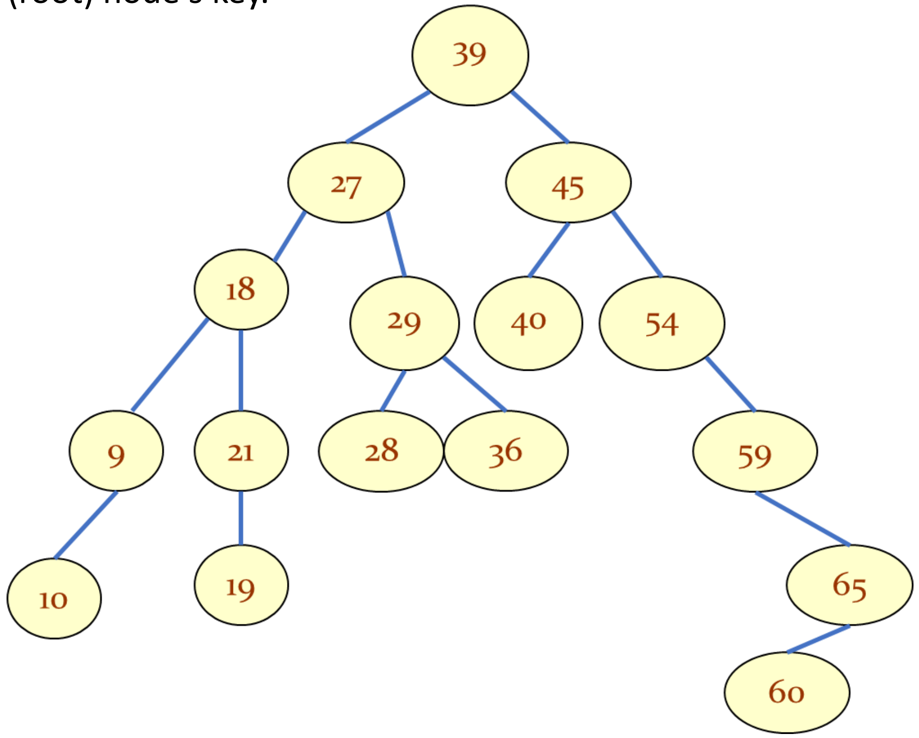
Binary search trees (BST)

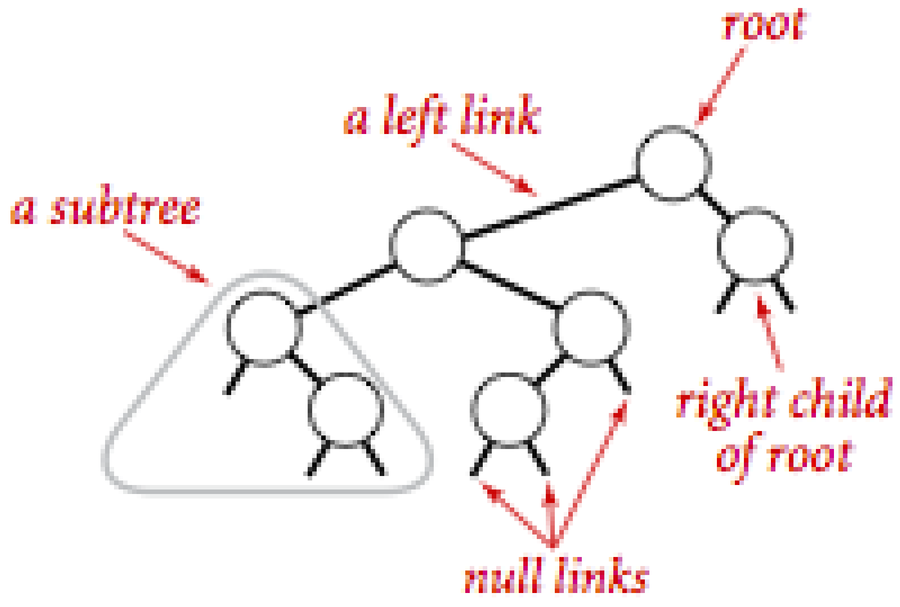
Binary Search Trees

- A binary search tree (BST), also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in order.
- In a BST, all nodes **in the left sub-tree have a value less than that of the root node**. **Correspondingly, all nodes in the right sub-tree have a value either equal to or greater than the root node.**
- The same rule is applicable to every sub-tree in the tree.
- Due to its efficiency in searching elements, BSTs are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

Binary Search Trees

- A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –
  - The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
  - The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

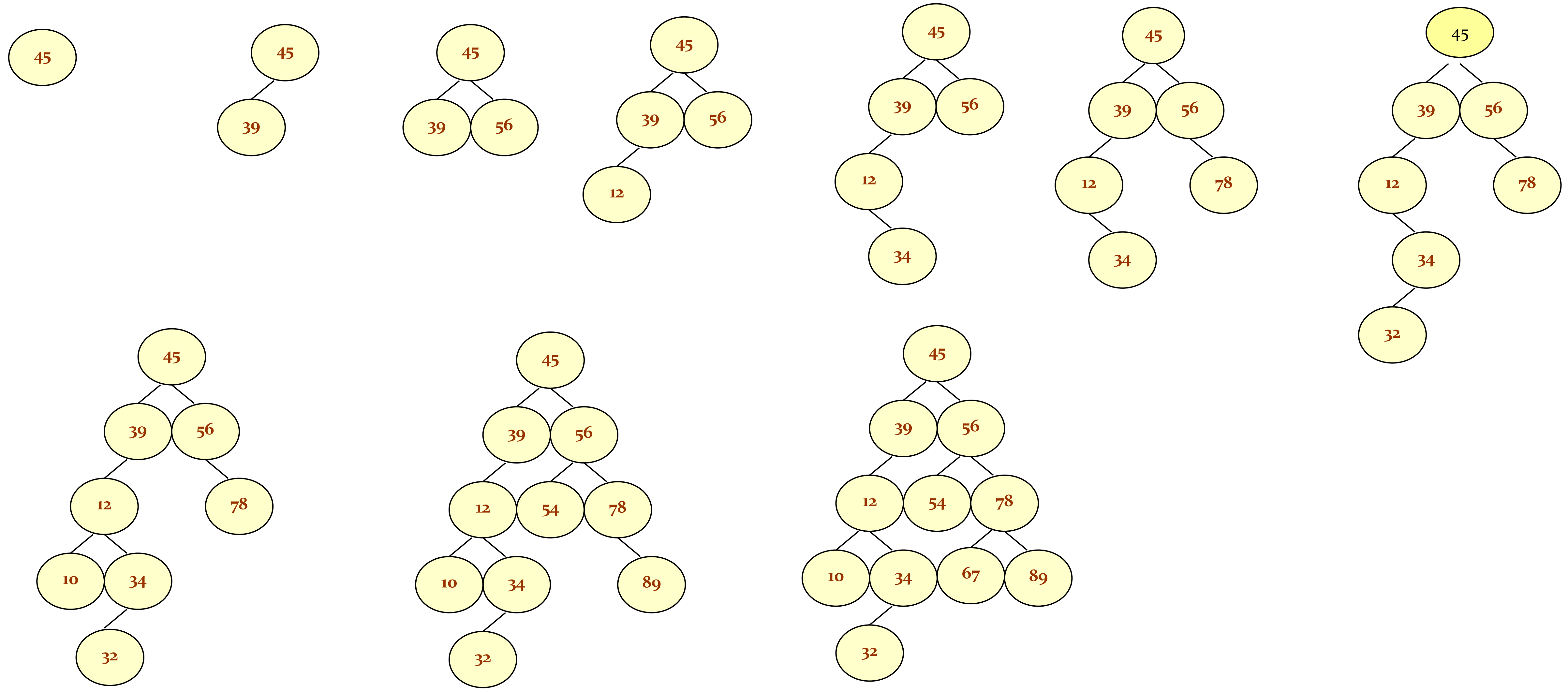




Anatomy of a binary tree

Creating a Binary Search from Given Values

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67



Algorithm to Insert a Value in a BST

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL, then
          Allocate memory for TREE
          SET TREE->DATA = VAL
          SET TREE->LEFT = TREE ->RIGHT = NULL
        ELSE
          IF VAL < TREE->DATA
             Insert(TREE->LEFT, VAL)
          ELSE
             Insert(TREE->RIGHT, VAL)
          [END OF IF]
        [END OF IF]
Step 2: End
```

Code to Insert a Value in a BST

```c
// C program to demonstrate insert // operation in binary search tree.
#include <stdio.h>
#include <stdlib.h>

struct node {
   int key;
   struct node *left, *right;
};

// A utility function to create a new BST 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;
}

// A utility function to do inorder traversal of BST
void inorder(struct node* root)
{
   if (root != NULL) {
      inorder(root->left);
      printf("%d \n", root->key);
      inorder(root->right);
} }
```

```c
/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
   /* If the tree is empty, return a new node */
   if (node == NULL)
      return newNode(key);

   /* Otherwise, recur down the tree */
   if (key < node->key)
      node->left = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);
    /* return the (unchanged) node pointer */
   return node;
}
```

```c
// Driver Code
int main()
{ struct node* root = NULL;
   root = insert(root, 50);
   insert(root, 30); /*    insert(root, 20);    insert(root, 40);   insert(root, 70); */

   // print inoder traversal of the BST
   inorder(root);

   return 0;
}
```

Code to Insert a Value in a BST

```python
# Python program to demonstrate to insert operation in binary search tree
# A utility class that represents an individual node in a BST


class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


# A utility function to insert a new node with the given key
def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if root.val == key:
            return root
        elif root.val < key:
            root.right = insert(root.right, key)
        else:
            root.left = insert(root.left, key)

    return root
```

```python
# A utility function to do inorder tree traversal
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val)
        inorder(root.right)


# Driver program to test the above functions
# Let us create the following BST
#    50
#  /     \
# 30     70
#  / \ / \
# 20 40 60 80


r = Node(50)
r = insert(r, 30)
r = insert(r, 20)
r = insert(r, 40)
r = insert(r, 70)
r = insert(r, 60)
r = insert(r, 80)


# Print inoder traversal of the BST
inorder(r)
```

Searching for a Value in a BST

- The search function is used to find whether a given value is present in the tree or not.
- The function first checks if the BST is empty. If it is, then the value we are searching for is not present in the tree, and the search algorithm terminates by displaying an appropriate message.
- If there are nodes in the tree, then the search function checks to see if the key value of the current node is equal to the value to be searched.
- If not, it checks if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node.
- In case the value is greater than the value of the node, it should be recursively called on the right child node.

Algorithm to Search a Value in a BST

```
searchElement(TREE, VAL)
Step 1:
     IF TREE->DATA = VAL OR TREE = NULL, then
        Return TREE
     ELSE
        IF VAL < TREE->DATA
           Return searchElement(TREE->LEFT, VAL)
        ELSE
           Return searchElement(TREE->RIGHT, VAL)
        [END OF IF]
      [END OF IF]
Step 2: End
```

Code to Search a Value in a BST

```
// A sample C function to check if a given node exists in a binary search tree or not
int search(struct node* root, int value)
{
    // while is used to traverse till the end of tree
    while (root != NULL){

        // checking condition and passing right subtree & recusing
        if (value > root->val)
            root = root->right;

        // checking condition and passing left subtree & recusing
        else if (value < root->val)
            root = root->left;
        else
            return 1; // if the value is found return 1
    }
    return 0;
}
```

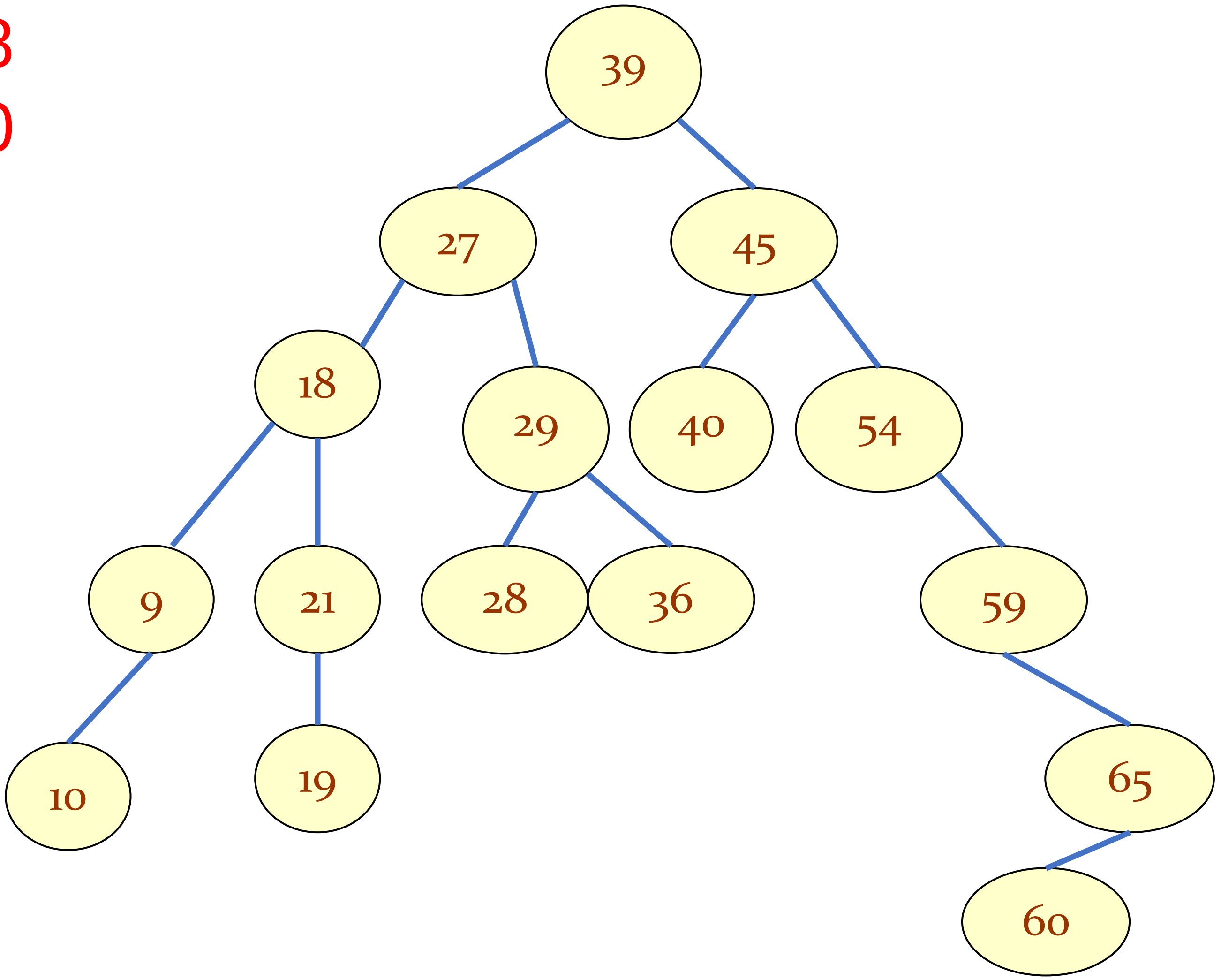
Code to Search a Value in a BST

```
# findval method to compare the value with nodes
   def findval(self, lkpval):
      if lkpval < self.data:
         if self.left is None:
            return str(lkpval)+" Not Found"
         return self.left.findval(lkpval)
       else if lkpval > self.data:
            if self.right is None:
               return str(lkpval)+" Not Found"
            return self.right.findval(lkpval)
        else:
            print(str(self.data) + ' is found')
```

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data

# Print the tree
   def PrintTree(self):
      if self.left:
         self.left.PrintTree()
      print( self.data),
      if self.right:
         self.right.PrintTree()
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
print(root.findval(7))
print(root.findval(14))
```

Searching for a Value in a BST

Search 28
Search 20

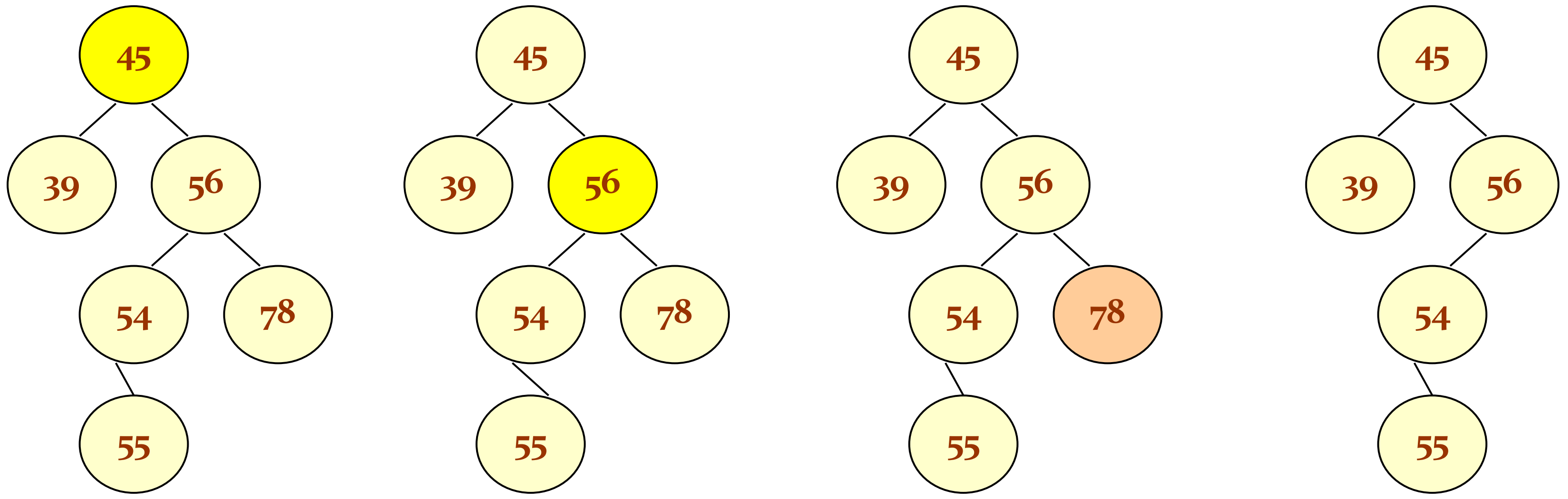


Deleting a Value from a BST

- The delete function deletes a node from the binary search tree.
- Deletion operation needs to keep the property of BSTs.
- The deletion of a node involves any of the three cases.

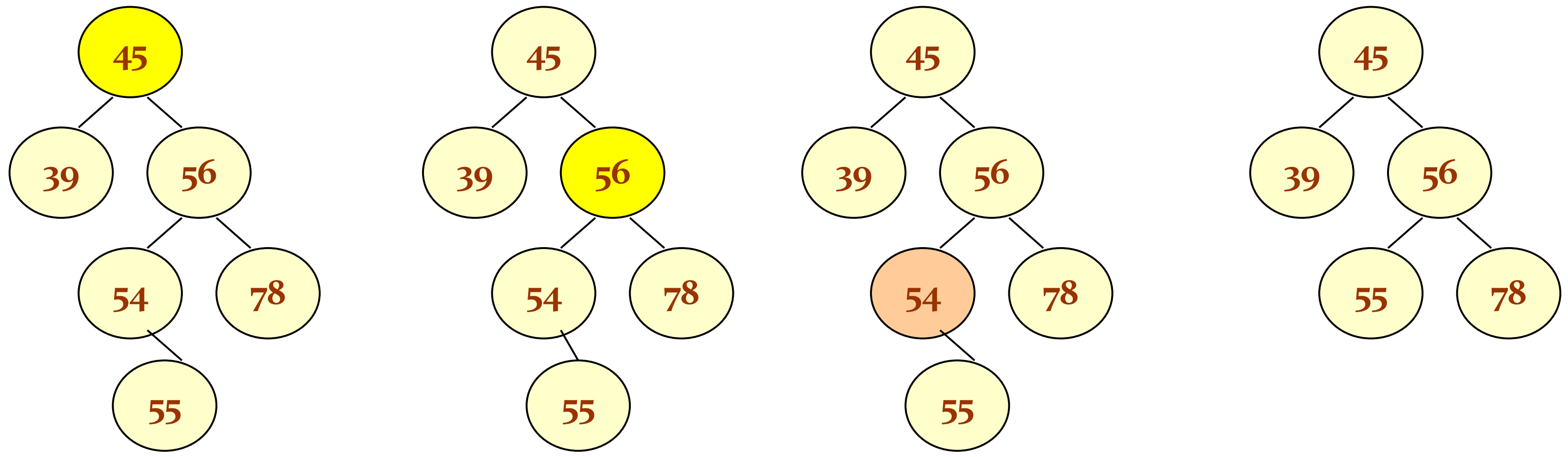
***Case 1*: Deleting a node that has no children.**

For example, deleting node 78 in the tree below.



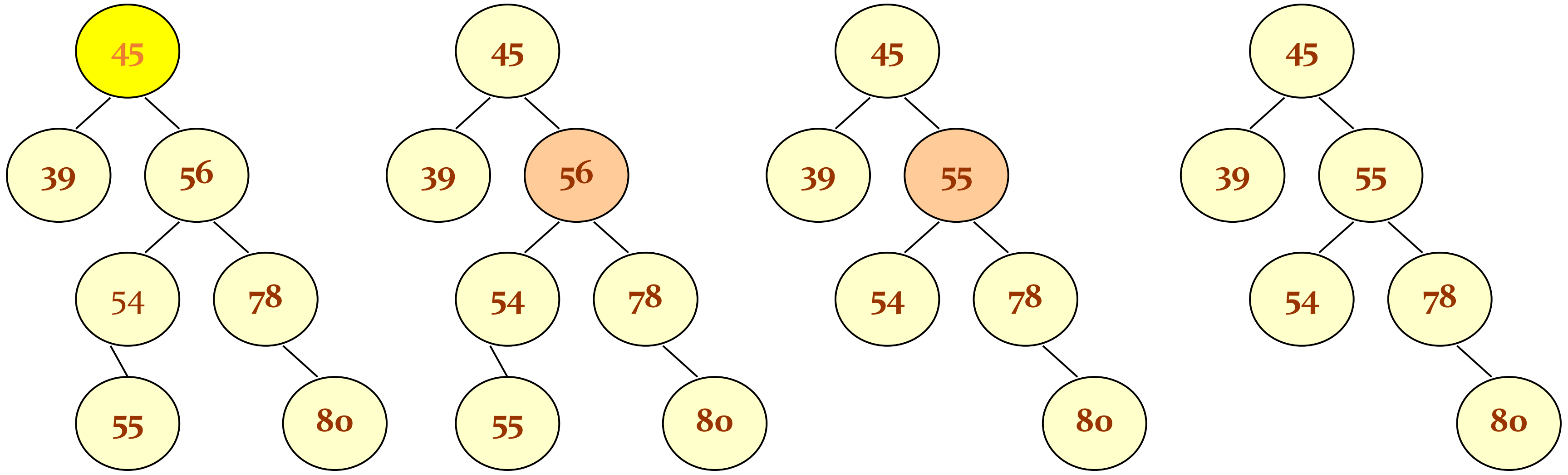
Deleting a Value from a BST

- ***Case 2:*** **Deleting a node with one child (either left or right).**
- To handle the deletion, **Copy the child to the node and delete the node**.
- Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent.
- Correspondingly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.



Deleting a Value from a BST

- ***Case 3:* Deleting a node with two children.**
- To handle this case of deletion, replace the node's value with its **in-order predecessor** (largest value in the left sub-tree) or **in-order successor** (smallest value in the right sub-tree).
- **The in-order predecessor or the successor can then be deleted using any of the above cases.**



Algorithm to Delete from a BST

```
Delete (TREE, VAL)
Step 1: IF TREE = NULL, then
          Write “VAL not found in the tree”
        ELSE IF VAL < TREE->DATA
            Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE->DATA
          Delete(TREE->RIGHT, VAL)
        ELSE IF TREE->LEFT AND TREE->RIGHT
            SET TEMP = findLargestNode(TREE->LEFT)
            SET TREE->DATA = TEMP->DATA
            Delete(TREE->LEFT, TEMP->DATA)
        ELSE
           SET TEMP = TREE
           IF TREE->LEFT = NULL AND TREE ->RIGHT = NULL
               SET TREE = NULL
           ELSE IF TREE->LEFT != NULL
               SET TREE = TREE->LEFT
           ELSE
               SET TREE = TREE->RIGHT
           [END OF IF]
             FREE TEMP
        [END OF IF]
Step 2: End
```

Code to Delete from a BST

```c
/* Given a binary search tree and a key, this function deletes the key and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL)
        return root;
     // If the key to be deleted is smaller than the root's key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);
     // If the key to be deleted is greater than the root's key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
     // if key is same as root's key, then This is the node to be deleted
    else {
        // node 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;
        }
         // node with two children Get the inorder successor  // (smallest in the right subtree)
        struct node* temp = minValueNode(root->right);
         // Copy the inorder successor's content to this node
        root->key = temp->key;
         // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

```c
struct node* minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current && current->left != NULL)
        current = current->left;

    return current;
}
```

Code to Delete from a BST

```
def deleteNode(root, key):

     # Base Case
    if root is None:
        return root

     # If the key to be deleted is smaller than the root's
    # key then it lies in  left subtree
    if key < root.key:
        root.left = deleteNode(root.left, key)

     # If the kye to be delete is greater than the root's key
    # then it lies in right subtree
    elif(key > root.key):
        root.right = deleteNode(root.right, key)

     # If key is same as root's key, then this is the node to be deleted
    else:

         # Node with only one child or no child
        if root.left is None:
            temp = root.right
            root = None
            return temp

        elif root.right is None:
            temp = root.left
            root = None
            return temp

         # Node with two children: Get the inorder successor
        # (smallest in the right subtree)
        temp = minValueNode(root.right)

         # Copy the inorder successor's content to this node
        root.key = temp.key

         # Delete the inorder successor
        root.right = deleteNode(root.right, temp.key)

   return root
```

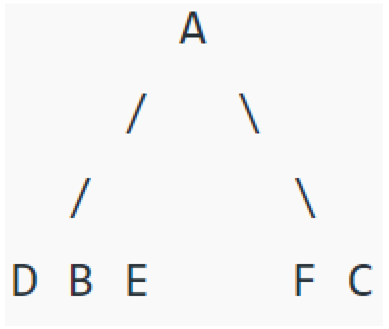
```
def minValueNode(node):
    current = node

    # loop down to find the leftmost leaf
    while(current.left is not None):
        current = current.left

    return current
```

Construct Tree from given traversals

Let us consider the below traversals:

- **Inorder sequence: D B E A F C**
- **Preorder sequence: A B D E C F**

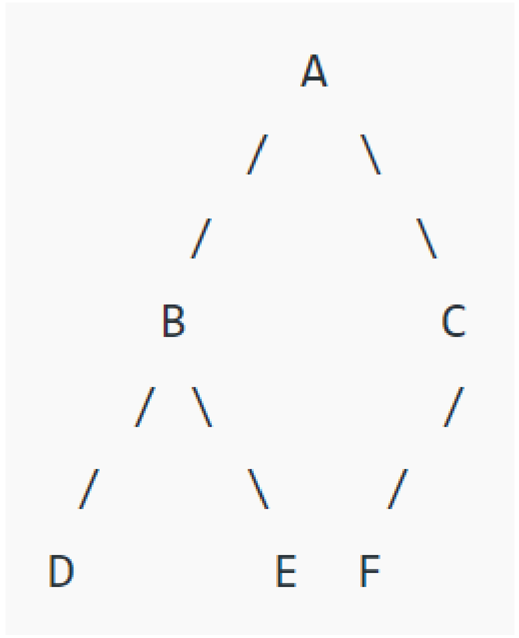
- Inorder sequence: D B E **A** F C
- Preorder sequence: **A** B D E C F



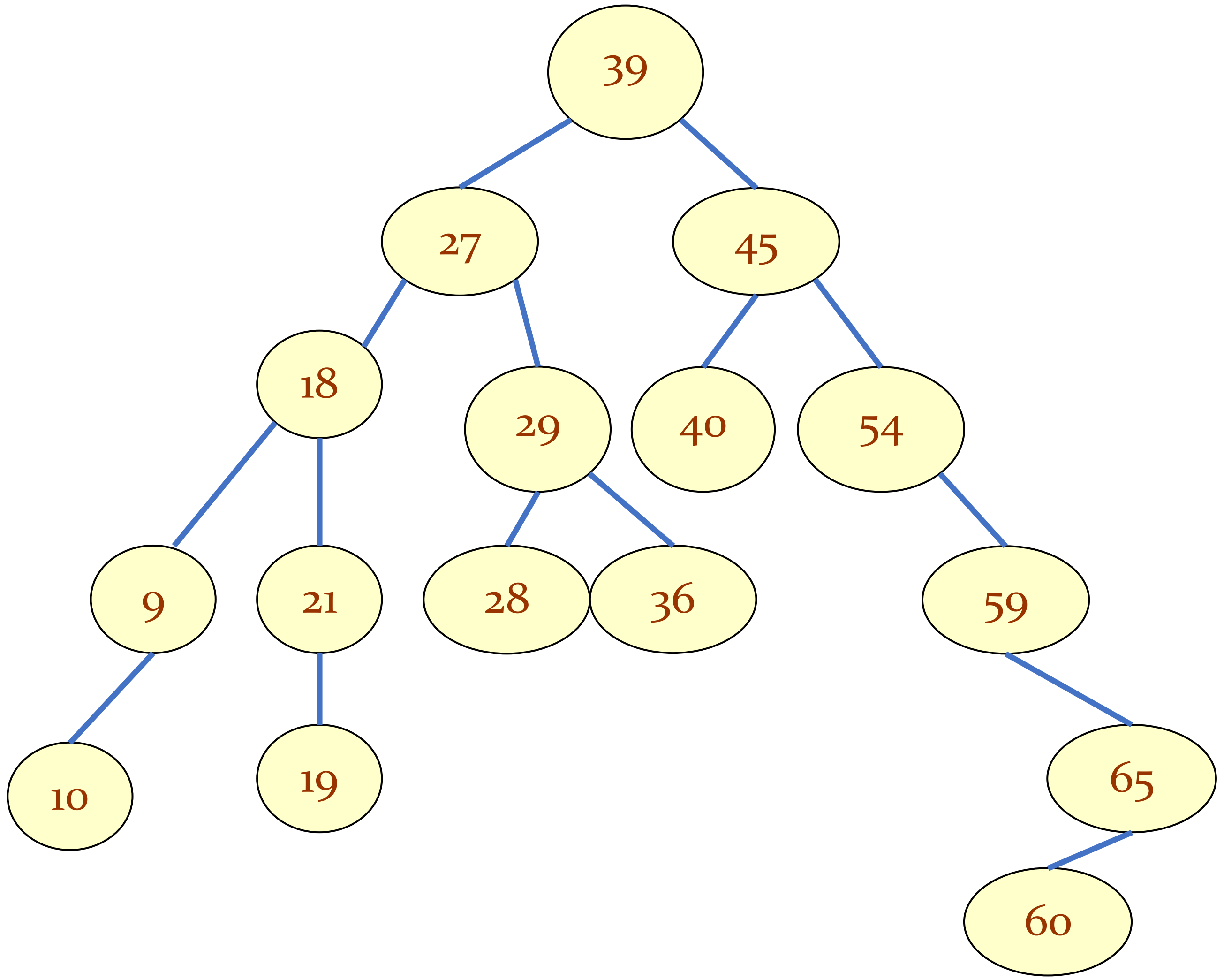
- Inorder sequence: D B E **A** F C
- Preorder sequence: **A** B D E C F

- Inorder sequence: D B E **A** F C
- Preorder sequence: **A** B D E C F

We recursively follow the above steps and get the following tree.



One more BST



Let's do ALL traversals as well

What is your conclusion ?

Finding the Largest Node in a BST

- The basic property of a BST states that the larger value will occur in the right sub-tree.
- If the right sub-tree is NULL, then the value of root node will be largest as compared with nodes in the left sub-tree.
- So, to find the node with the largest value, we will find the value of the rightmost node of the right sub-tree.
- If the right sub-tree is empty then we will find the value of the root node.

```
findLargestElement (TREE)
Step 1: IF TREE = NULL OR TREE->RIGHT = NULL, then
                Return TREE
             ELSE
                Return findLargestElement(TREE->RIGHT)
           [END OF IF]
Step 2: End
```

```
struct node* minValueNode(struct node* node)
{
   struct node* current = node;

   /* loop down to find the leftmost leaf */
   while (current && current->right != NULL)
      current = current->right;

   return current;
}
```

Finding the Smallest Node in a BST

- The basic property of a BST states that the smaller value will occur in the left sub-tree.
- If the left sub-tree is NULL, then the value of root node will be smallest as compared with nodes in the right sub-tree.
- So, to find the node with the smallest value, we will find the value of the leftmost node of the left sub-tree.
- However, if the left sub-tree is empty then we will find the value of the root node.

```
findSmallestElement (TREE)
Step 1: IF TREE = NULL OR TREE->LEFT = NULL, then
                    Return TREE
          ELSE
                    Return findSmallestElement(TREE->LEFT)
          [END OF IF]
Step 2: End
```

Balanced BST

Balanced BST,   i.e. height of left and right subtrees are equal or not much differences at any node

Example: a full binary tree of n nodes

The search in can be done in log(n) time, O(log n).

Depth of recursion is O(log n)

Time complexity   O(log n)

Space complexity O(log n)

A BST is not balanced in general !

We will discuss on AVL tree section