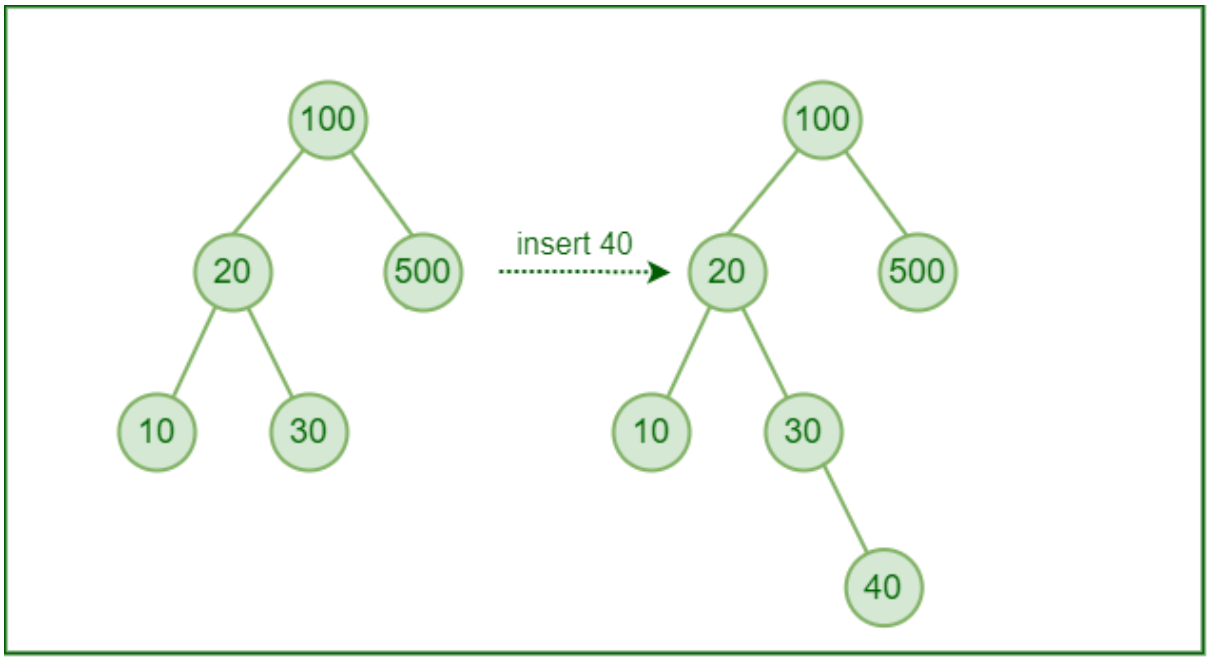
Insertion in Binary Search Tree (BST)

Given a **BST**, the task is to insert a new node in this **BST**.
Example:



Insertion in Binary Search Tree

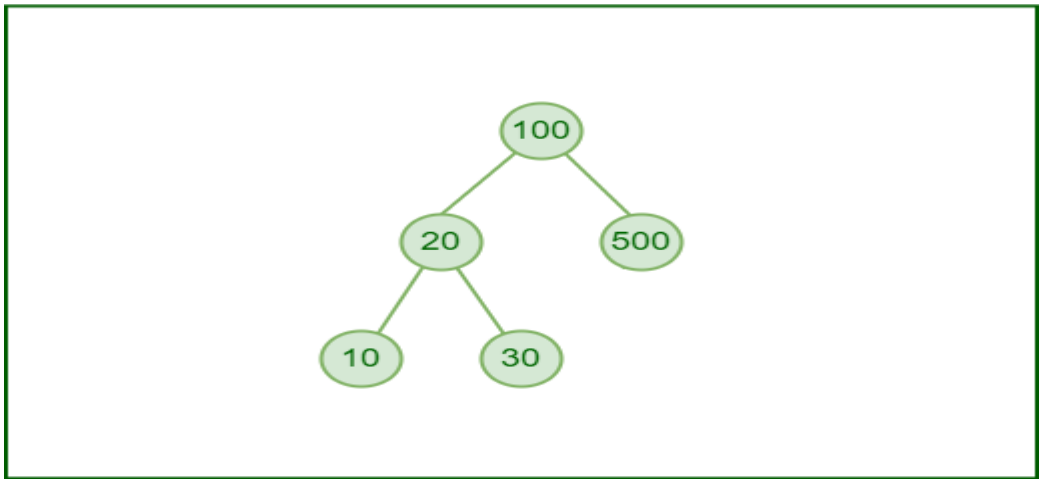
How to Insert a value in a Binary Search Tree:

A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:

- Check the value to be inserted (say **X**) with the value of the current node (say **val**) we are in:
    - If **X** is less than **val** move to the left subtree.
    - Otherwise, move to the right subtree.
- Once the leaf node is reached, insert **X** to its right or left based on the relation between **X** and the leaf node's value.

Follow the below illustration for a better understanding:

Illustration:
Consider the below tree:



Binary Search Tree

Let us try to insert a node with value **40** in this tree:
1st step: 40 will be compared with root, i.e., 100.

- 40 is less than 100.
- So move to the left subtree of 100. The root of the left subtree is 20.

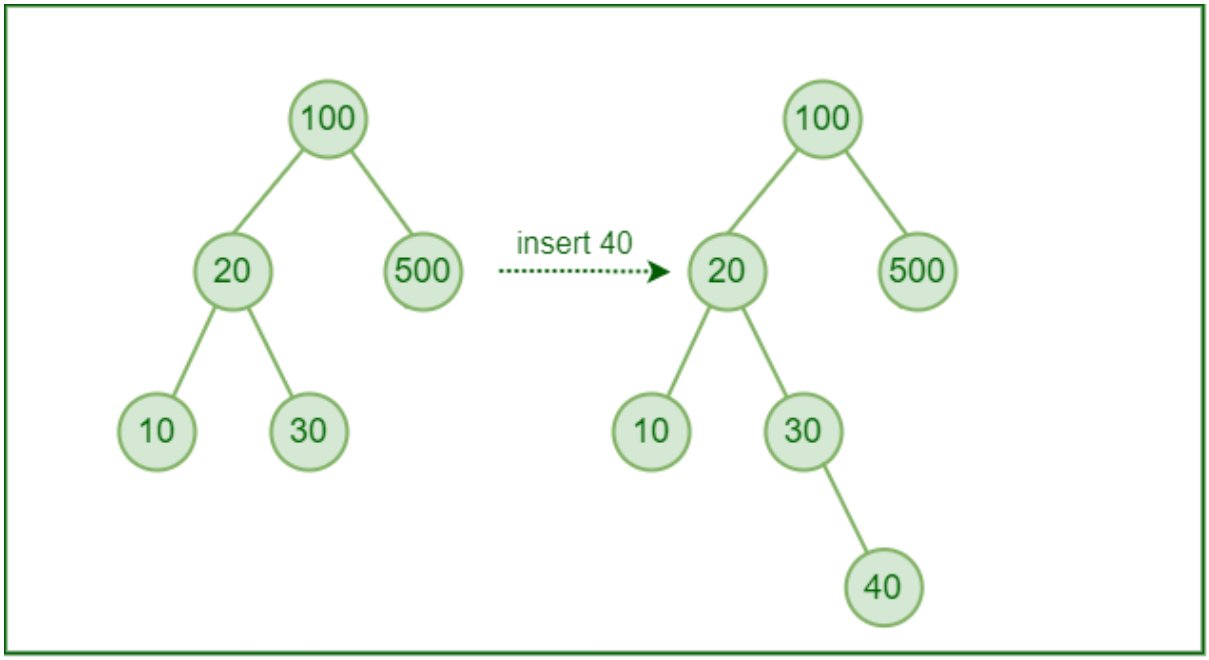
2nd step: 40 is now compared with 20.

- It is greater than 20.
- So move to the right subtree of 20 whose root is 30.

3rd step: 30 is a leaf node.

- So we have to insert 40 to the left or right of 30.
- As 40 is greater than 30, insert 40 to the right of 30.

The new tree will look like the following:



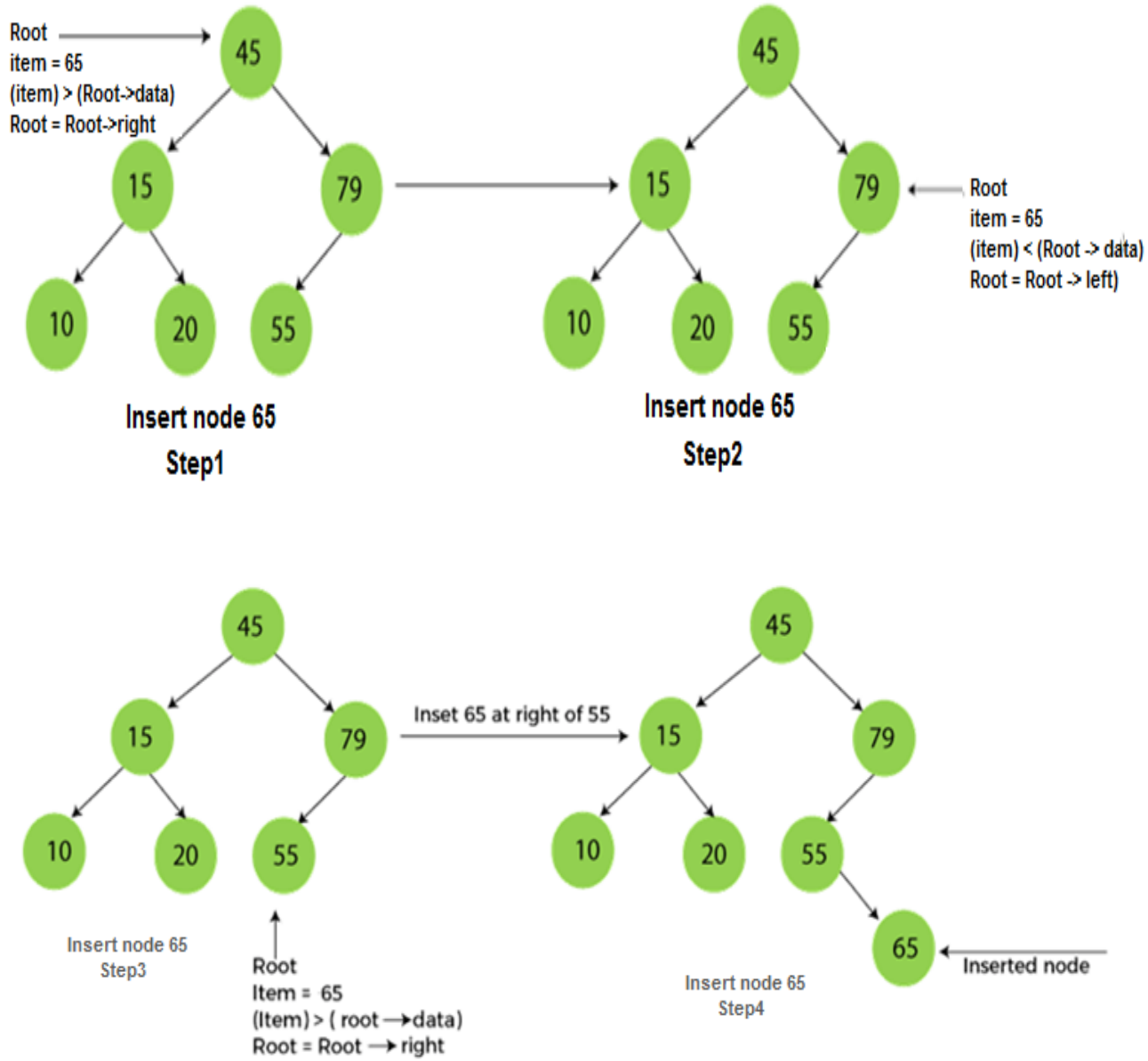
Insertion in Binary Search Tree

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search

for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.



Insertion in Binary Search Tree using Recursion:

Below is the implementation of the insertion operation using recursion.

```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 ", root->key);
        inorder(root->right);
    }
}

// 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;
}
```

```
// Driver Code
intmain()
{
   /* Let us create following BST
            50
         /     \
        30      70
       / \    / \
     20   40  60   80 */
   structnode* root = NULL;
   root = insert(root, 50);
   insert(root, 30);
   insert(root, 20);
   insert(root, 40);
   insert(root, 70);
   insert(root, 60);
   insert(root, 80);

   // Print inorder traversal of the BST
   inorder(root);

   return0;
}
```

Output
20 30 40 50 60 70 80

Time Complexity:

- The worst-case time complexity of insert operations is **O(h)** where **h** is the height of the Binary Search Tree.
- In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become **n** and the time complexity of insertion operation may become **O(n).**

Auxiliary Space: The auxiliary space complexity of insertion into a binary search tree is **O(1)**

Insertion in Binary Search Tree using Iterative approach:

Instead of using recursion, we can also implement the insertion operation iteratively using a **while loop**. Below is the implementation using a while loop.

```
// C++ Code to insert node and to print inorder traversal
```

```
// using iteration

#include <bits/stdc++.h>
usingnamespacestd;

// BST Node
classNode {
public:
    intval;
    Node* left;
    Node* right;
    Node(intval)
        : val(val)
        , left(NULL)
        , right(NULL)
    {
    }
};

// Utility function to insert node in BST
voidinsert(Node*& root, intkey)
{
    Node* node = newNode(key);
    if(!root) {
        root = node;
        return;
    }
    Node* prev = NULL;
    Node* temp = root;
    while(temp) {
        if(temp->val> key) {
            prev = temp;
            temp = temp->left;
        }
        elseif(temp->val< key) {
            prev = temp;
            temp = temp->right;
        }
    }
    if(prev->val> key)
        prev->left = node;
    else
        prev->right = node;
}

// Utility function to print inorder traversal
voidinorder(Node* root)
{
    Node* temp = root;
```

```
    stack<Node*>st;
    while(temp != NULL || !st.empty()) {
        if(temp != NULL) {
            st.push(temp);
            temp = temp->left;
        }
        else{
            temp = st.top();
            st.pop();
            cout<< temp->val<<" ";
            temp = temp->right;
        }
    }
}

// Driver code
intmain()
{
    Node* root = NULL;
    insert(root, 30);
    insert(root, 50);
    insert(root, 15);
    insert(root, 20);
    insert(root, 10);
    insert(root, 40);
    insert(root, 60);

    // Function call to print the inorder traversal
    inorder(root);

    return0;
}
```

Output

10 15 20 30 40 50 60

The **time complexity** of **inorder traversal** is **O(n)**, as each node is visited once. The **Auxiliary space** is **O(n)**, as we use a stack to store nodes for recursion.