Binary Search Tree

SWE2016-44

BST

Binary Search Tree is a node-based binary tree data structure.

Properties

1) **The left subtree of a node contains nodes with keys less than the node's key and the right subtree with keys greater than node's key.**

2) **The left and right subtree each must also be a binary search tree and there must be no duplicate nodes.**

BST

1) **Insertion**

2) **Searching**

3) **Delete**

Insertion in a BST

While doing insertion in BST the new key is always inserted at leaf.

Insertion in a BST

While doing insertion in BST the new key is always inserted at leaf.

We start searching a key from root till we hit a leaf node.

Insertion in a BST

While doing insertion in BST the new key is always inserted at leaf.

We start searching a key from root till we hit a leaf node.

Once a leaf node is found, the new node is added as a child of the leaf node.

Insertion in a 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;
}
```

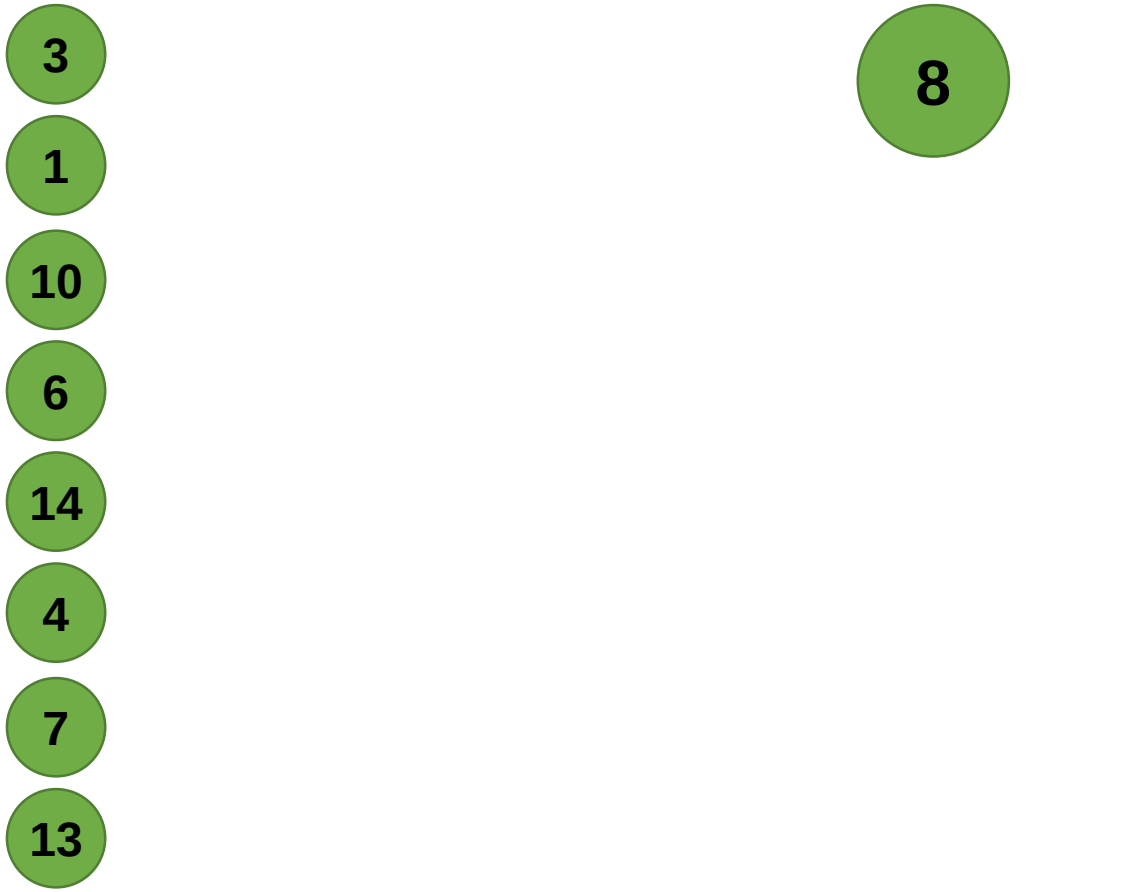
For example, let's create a BST by inserting multiple keys into it.

Insertion in a 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;
}
```

For example, let's create a BST by inserting multiple keys into it.

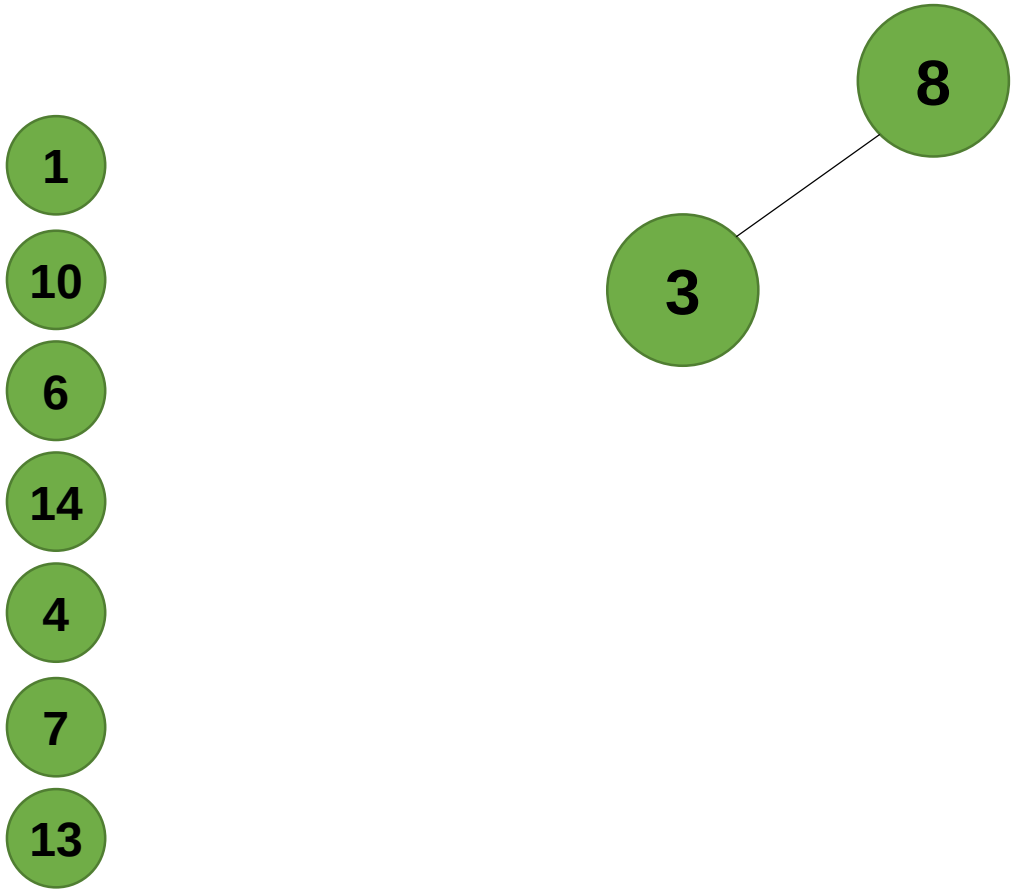
Insertion in a BST




```
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;
}
```

For example, let's create a BST by inserting multiple keys into it.

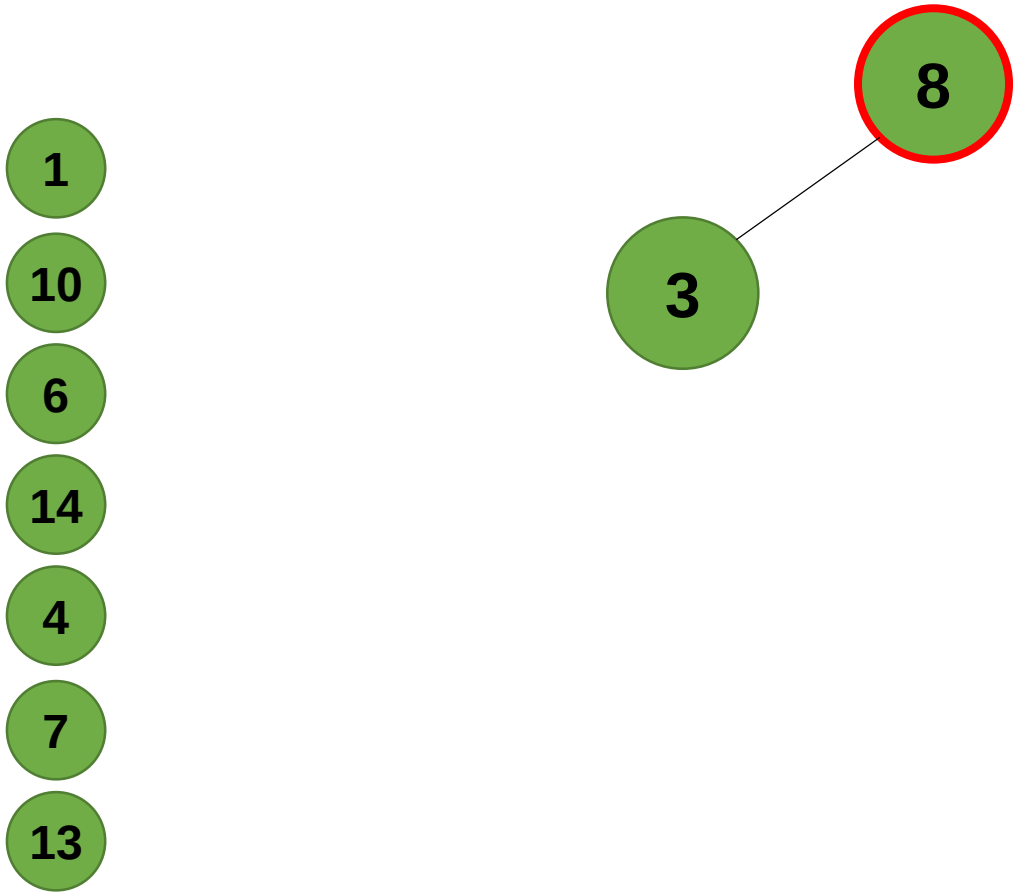
Insertion in a BST




```
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;
}
```

For example, let's create a BST by inserting multiple keys into it.

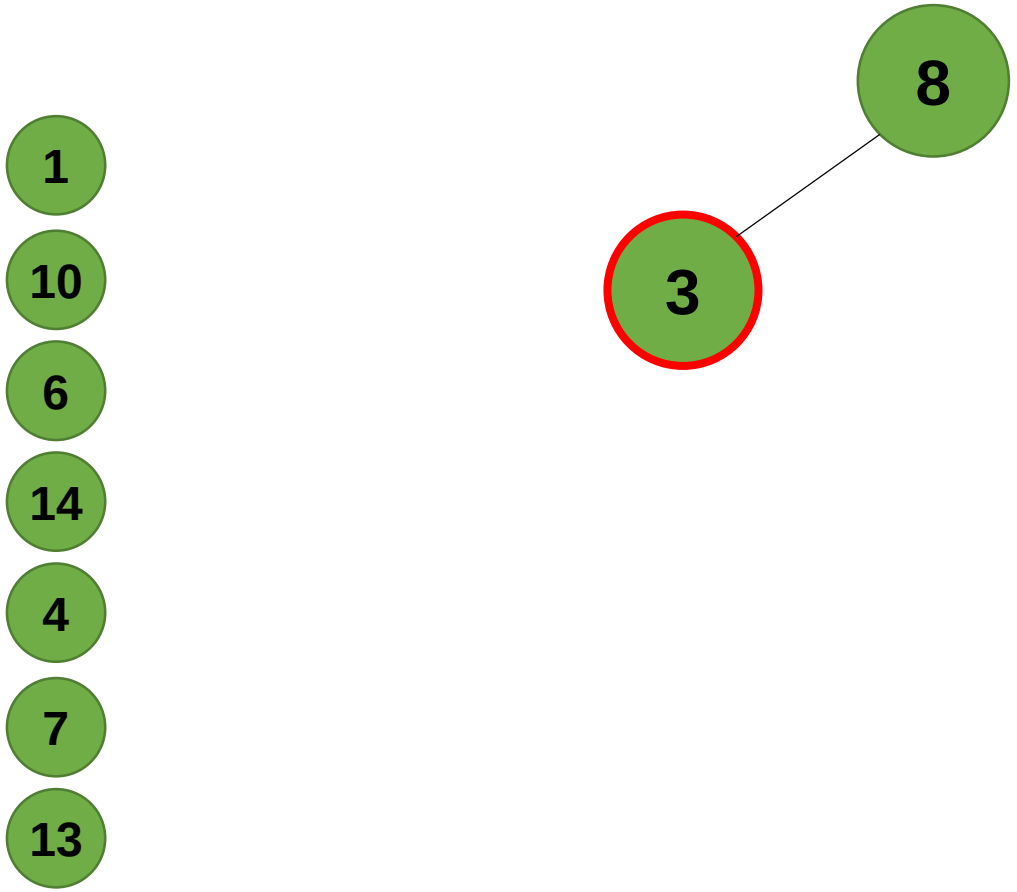
Insertion in a BST




```
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;
}
```

Make sure the keys greater than the node's key are added to the right and the smaller ones are added to the left.

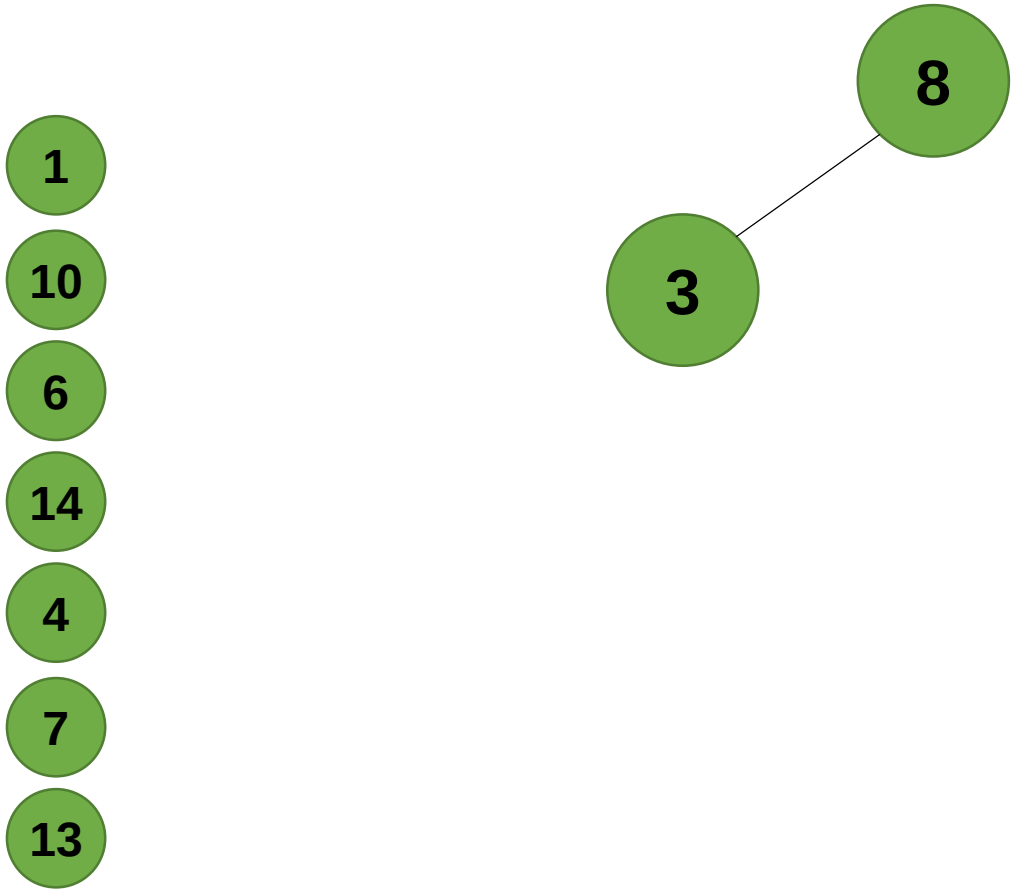
Insertion in a BST




```
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;
}
```

Make sure the keys greater than the node's key are added to the right and the smaller ones are added to the left.

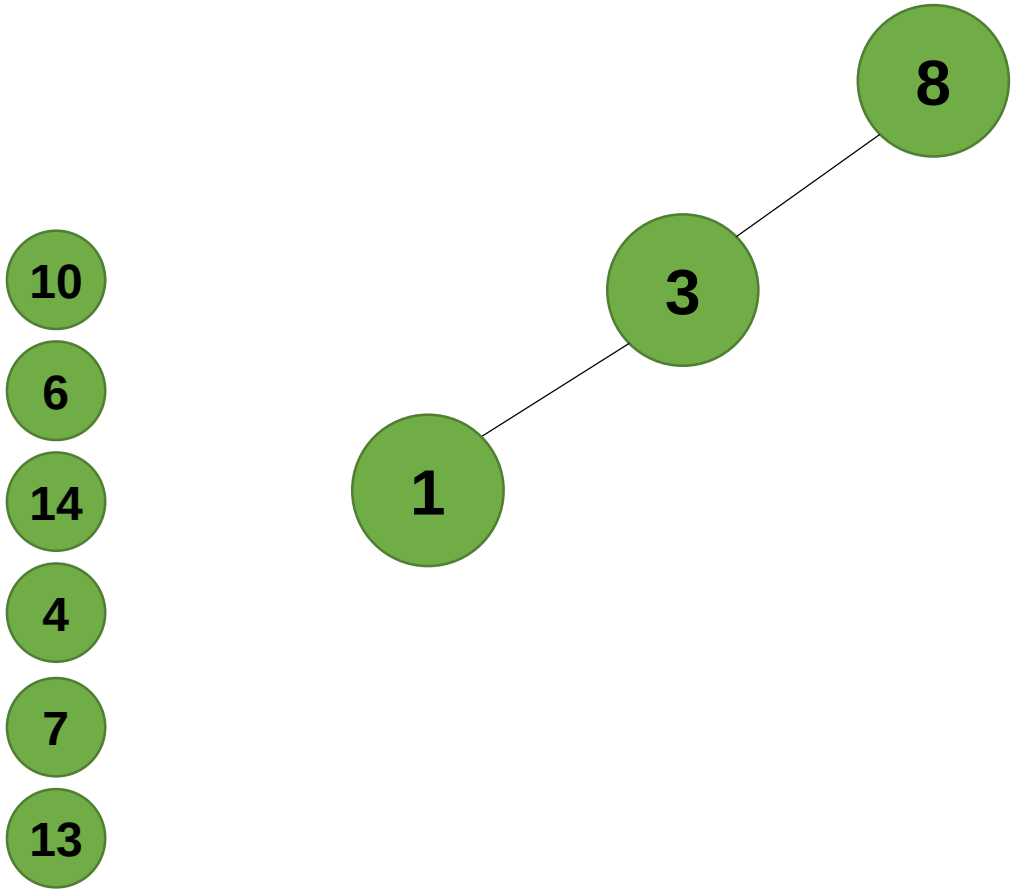
Insertion in a BST




```
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;
}
```

Make sure the keys greater than the node's key are added to the right and the smaller ones are added to the left.

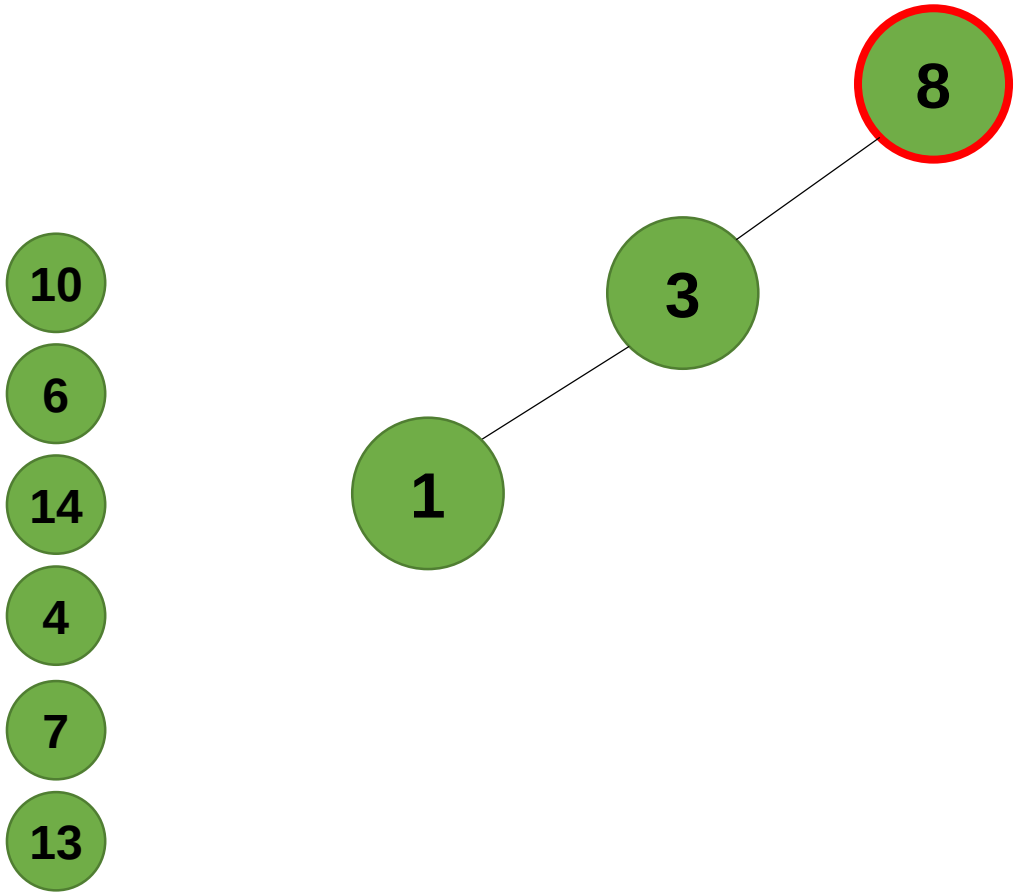
Insertion in a BST




```
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;
}
```

Make sure the keys greater than the node's key are added to the right and the smaller ones are added to the left.

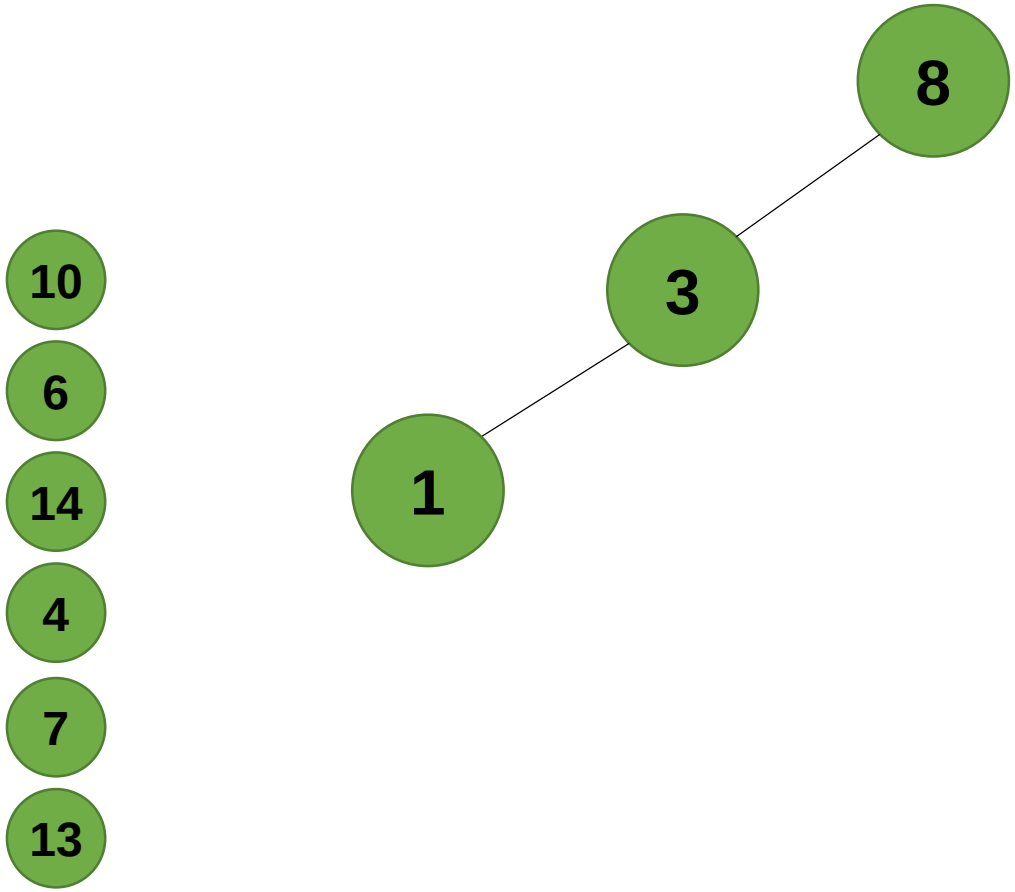
Insertion in a BST




```
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;
}
```

Keep adding the nodes according to the previous rule until no nodes are left.

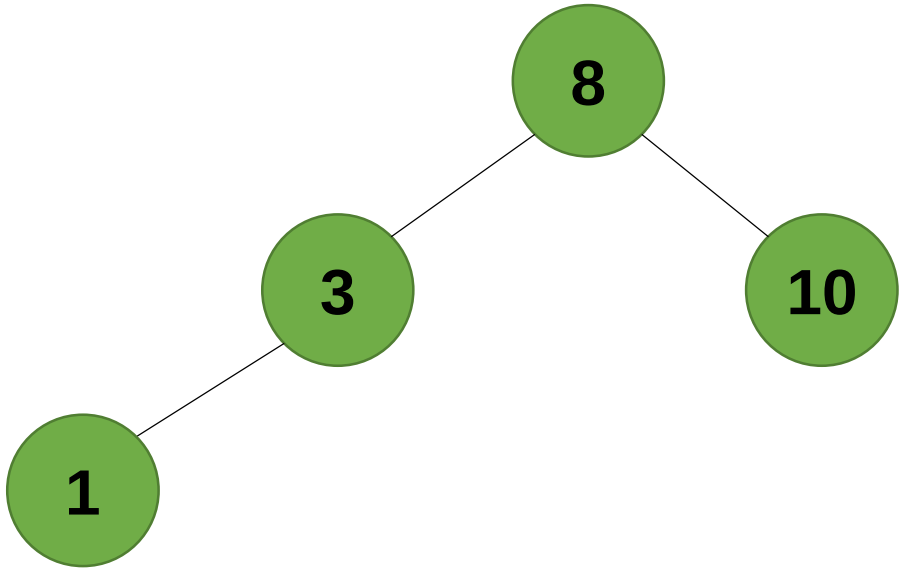
Insertion in a BST




```
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;
}
```

Keep adding the nodes according to the previous rule until no nodes are left.

Insertion in a BST



6

14

4

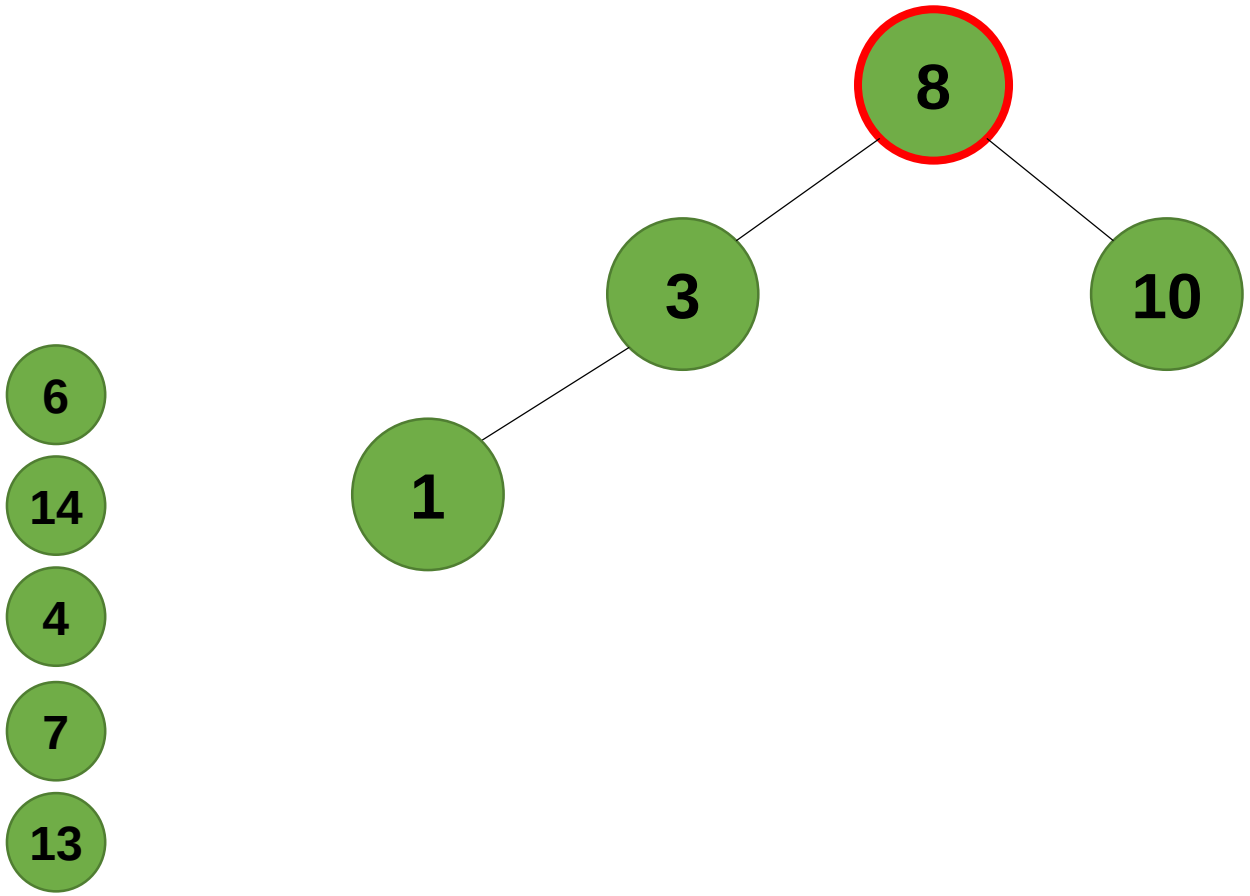
7

13

```
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;
}
```

Keep adding the nodes according to the previous rule until no nodes are left.

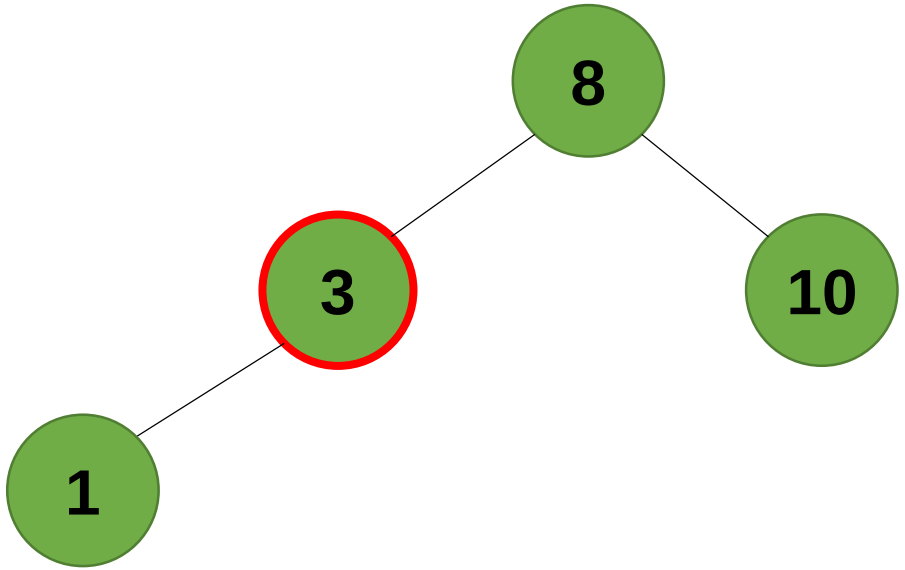
Insertion in a BST




```
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;
}
```

Keep adding the nodes according to the previous rule until no nodes are left.

Insertion in a BST

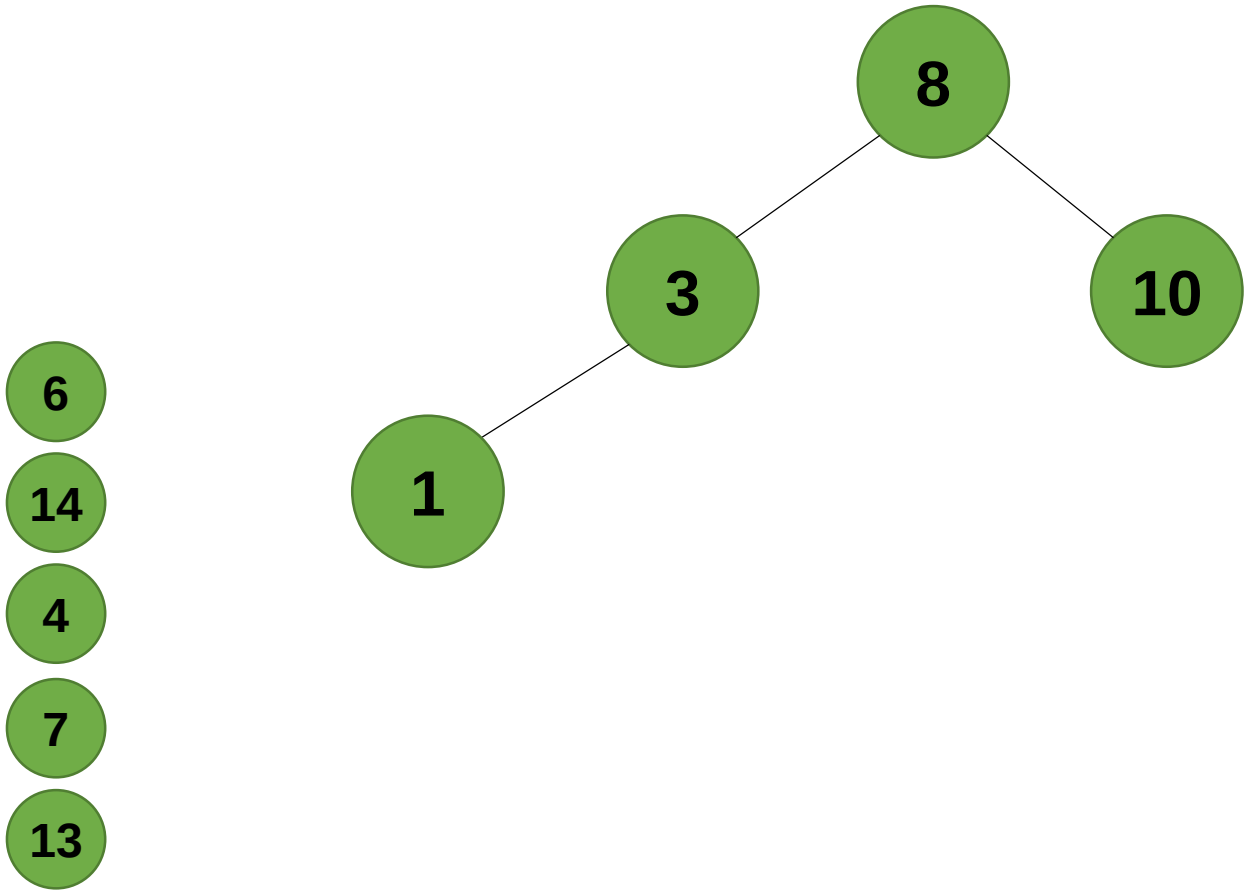


6
14
4
7
13

```
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;
}
```

Keep adding the nodes according to the previous rule until no nodes are left.

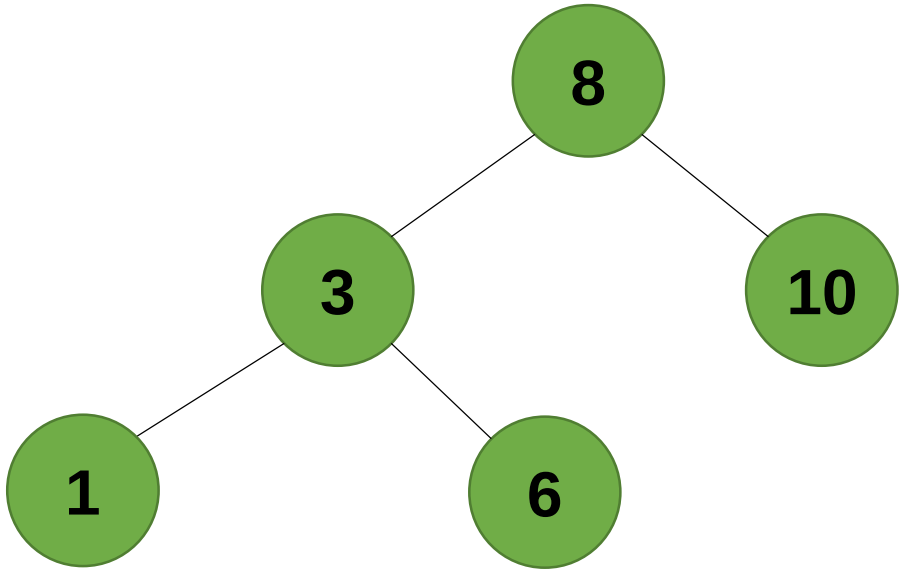
Insertion in a BST




```
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;
}
```

Keep adding the nodes according to the previous rule until no nodes are left.

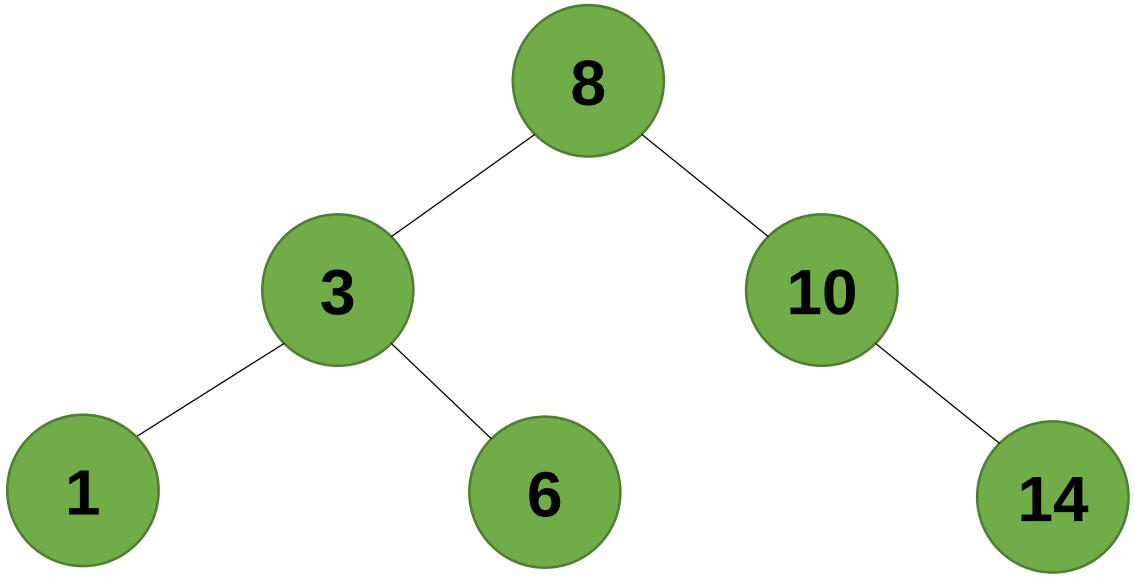
Insertion in a BST




```
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;
}
```

Keep adding the nodes according to the previous rule until no nodes are left.

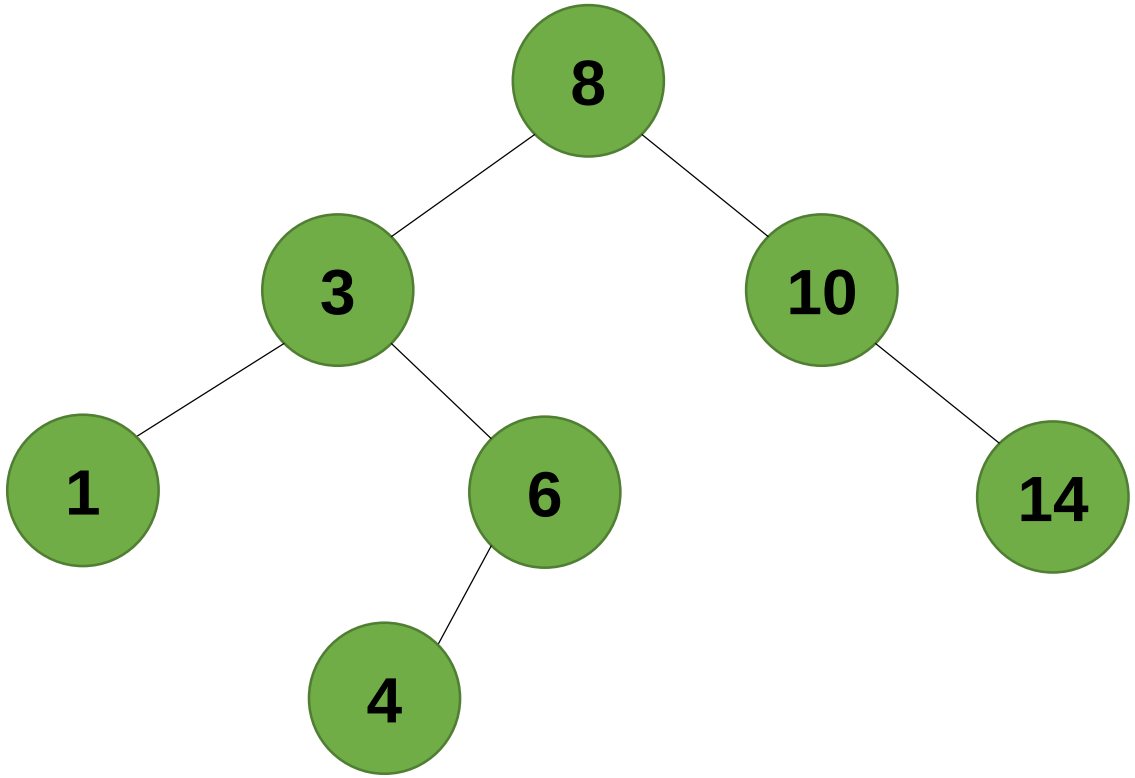
Insertion in a BST




```
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;
}
```

Similarly add the rest of the nodes and you will get the BST needed.

Insertion in a BST



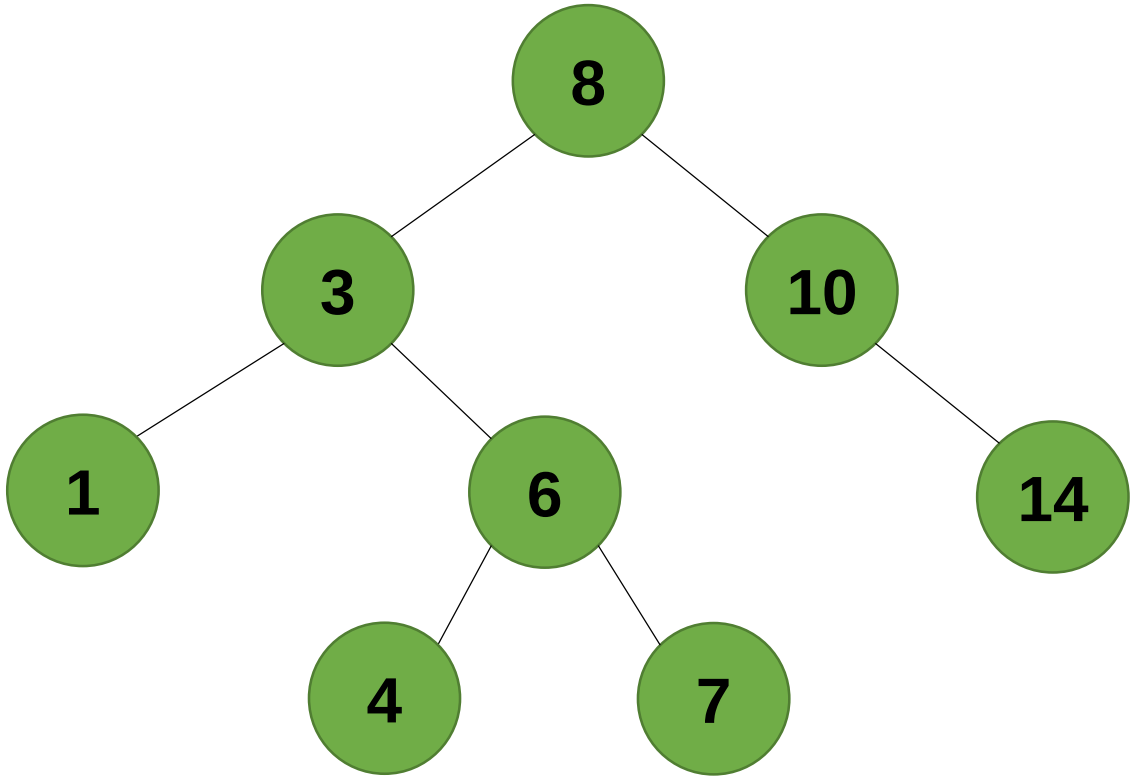
7

13

```
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;
}
```

Similarly add the rest of the nodes and you will get the BST needed.

Insertion in a BST

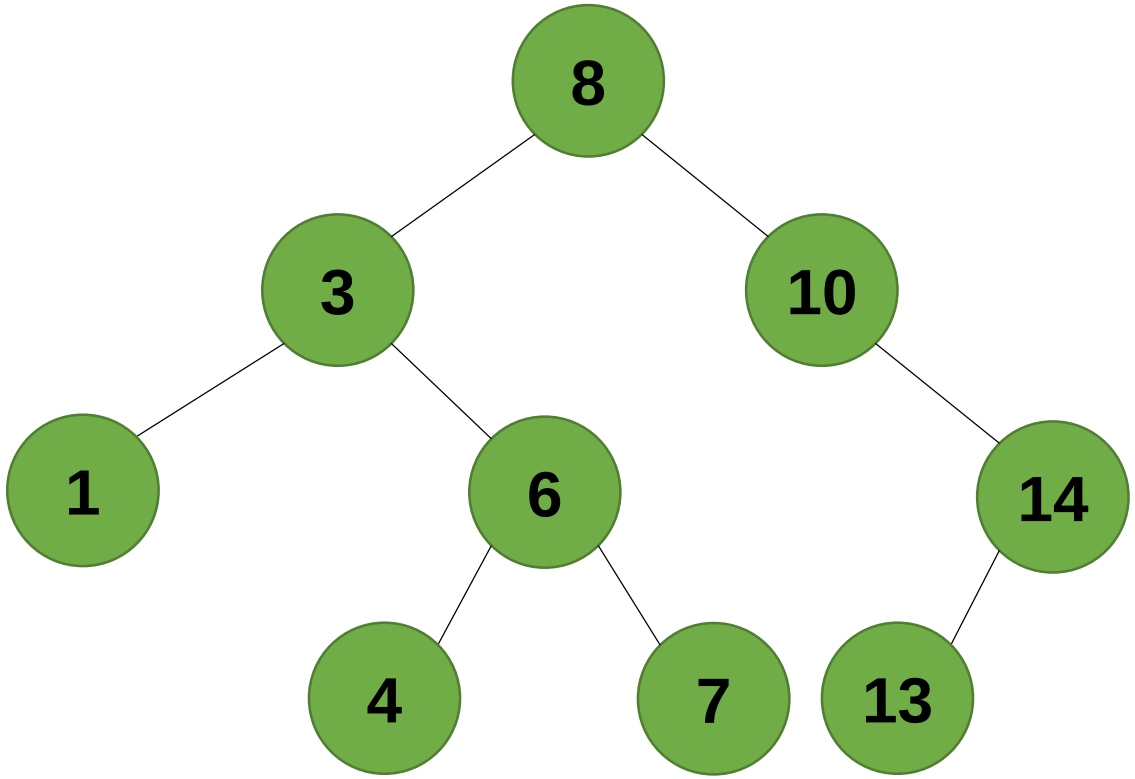


13

```
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;
}
```

Similarly add the rest of the nodes and you will get the BST needed.

Insertion in a 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;
}
```

Similarly add the rest of the nodes and you will get the BST needed.

Searching in a BST

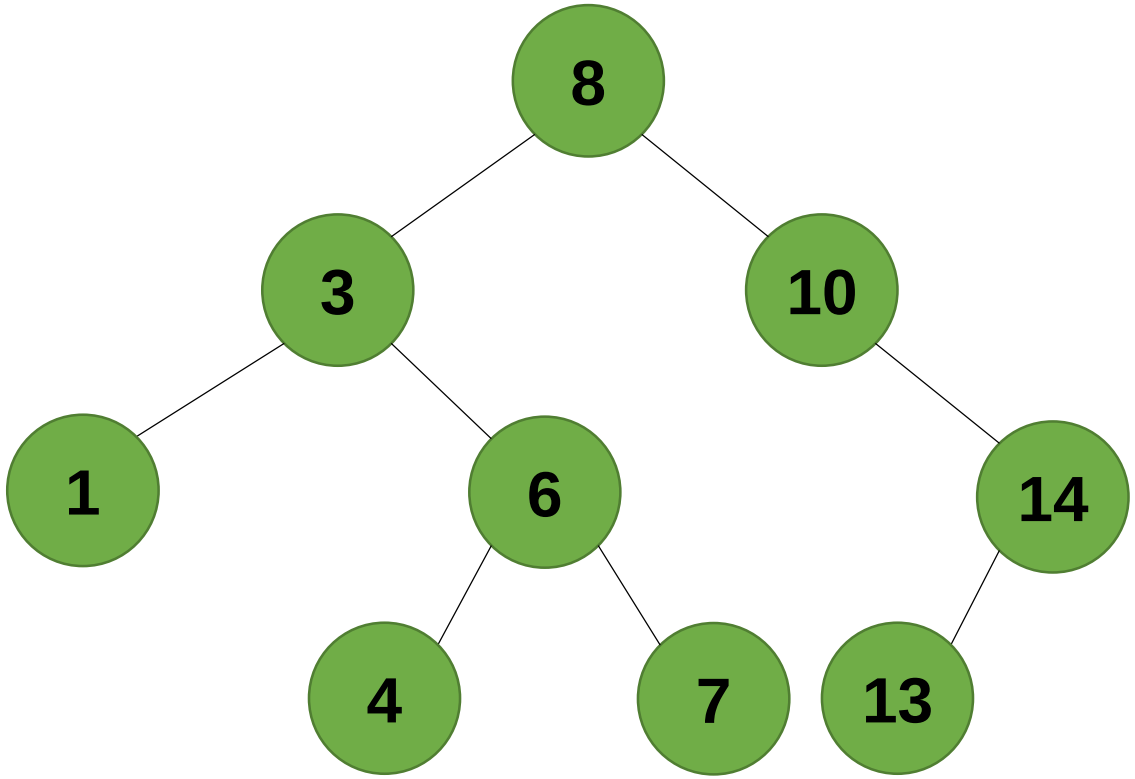
To search a given key in Binary Search Tree, we first compare it with root.

Searching in a BST

To search a given key in Binary Search Tree, we first compare it with root.

If the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node.

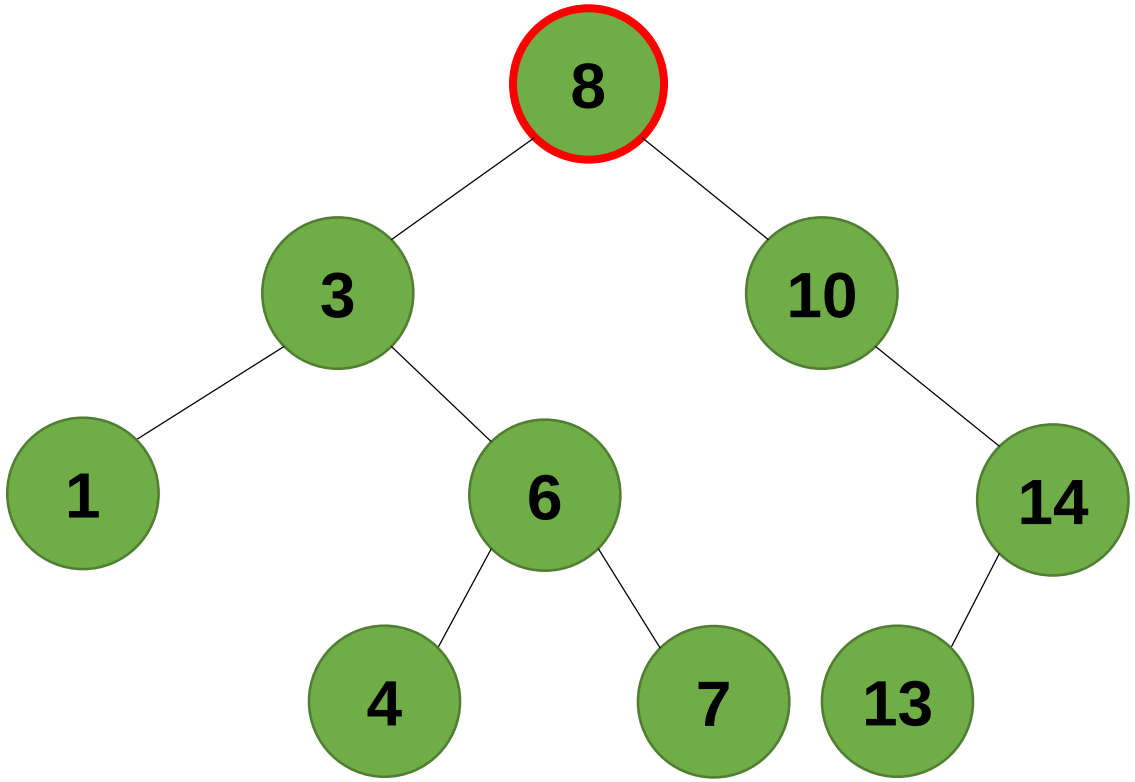
Searching in a BST




```
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);
}
```

Let’s search the key ‘4’ from the BST we made earlier.

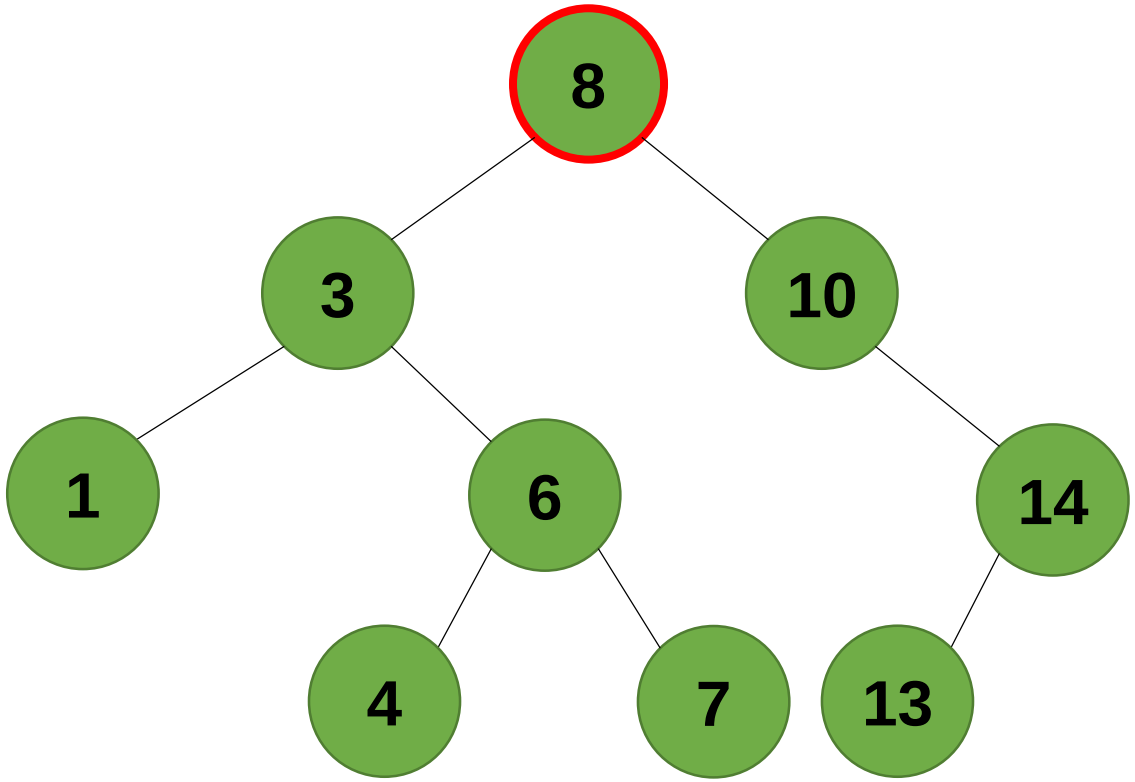
Searching in a BST




```
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);
}
```

Starting from the root, keep moving downwards until you find the key required.

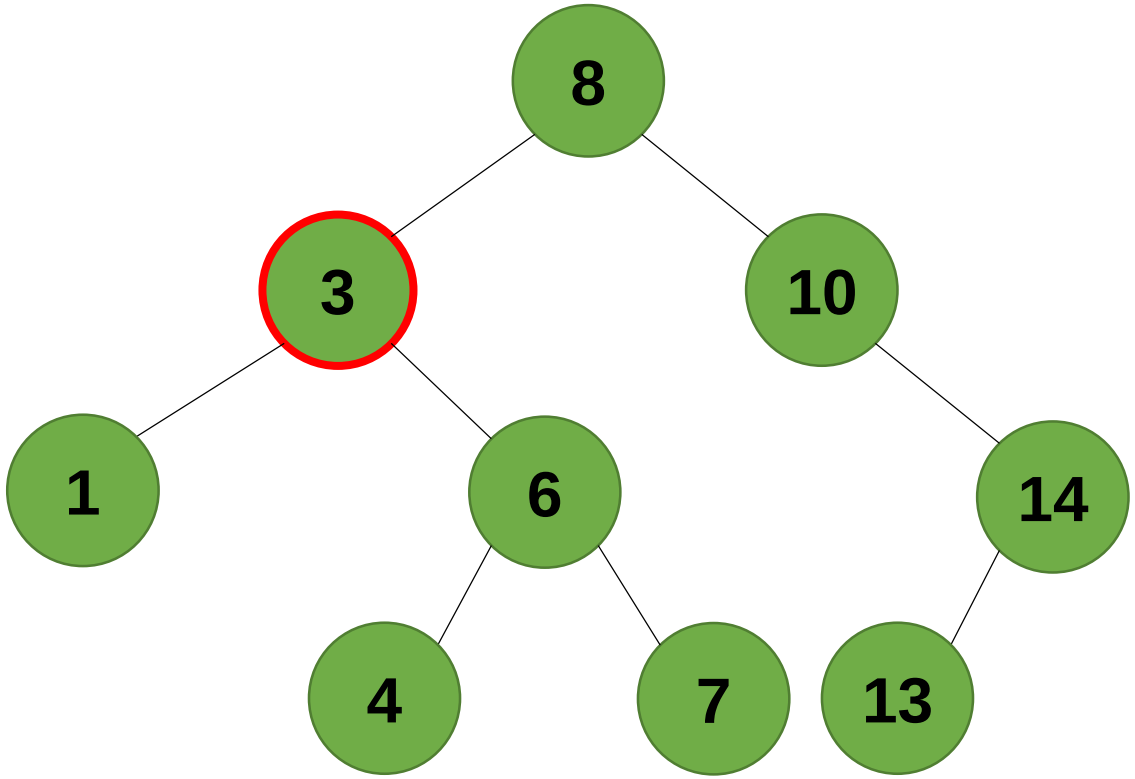
Searching in a BST




```
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);
}
```

Move to the right child if the key is greater than the selected node and left if it is smaller.

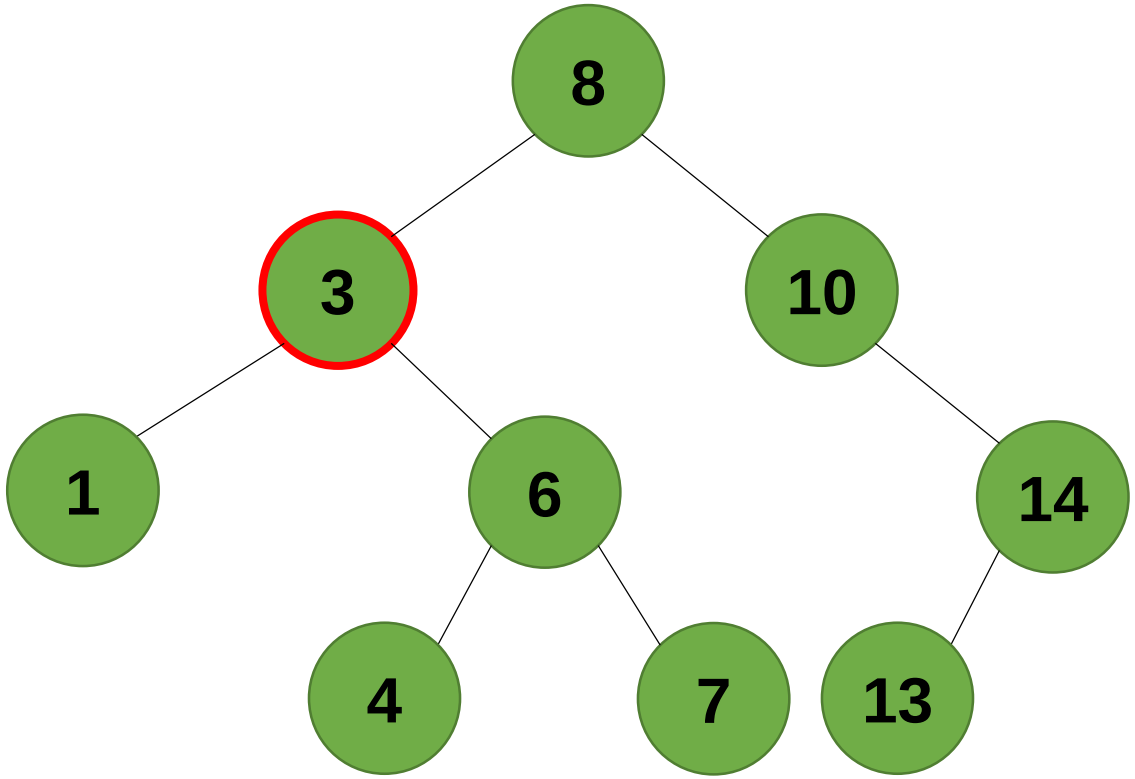
Searching in a BST




```
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);
}
```

Move to the right child if the key is greater than the selected node and left if it is smaller.

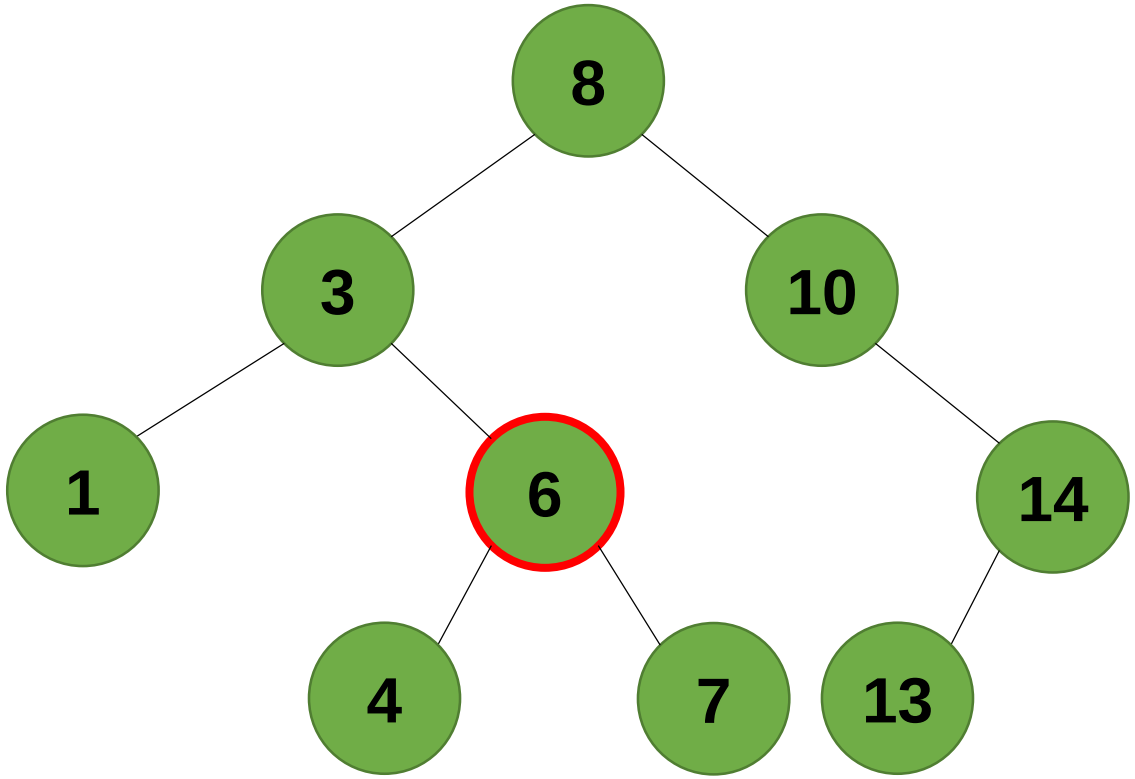
Searching in a BST




```
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);
}
```

Move to the right child if the key is greater than the selected node and left if it is smaller.

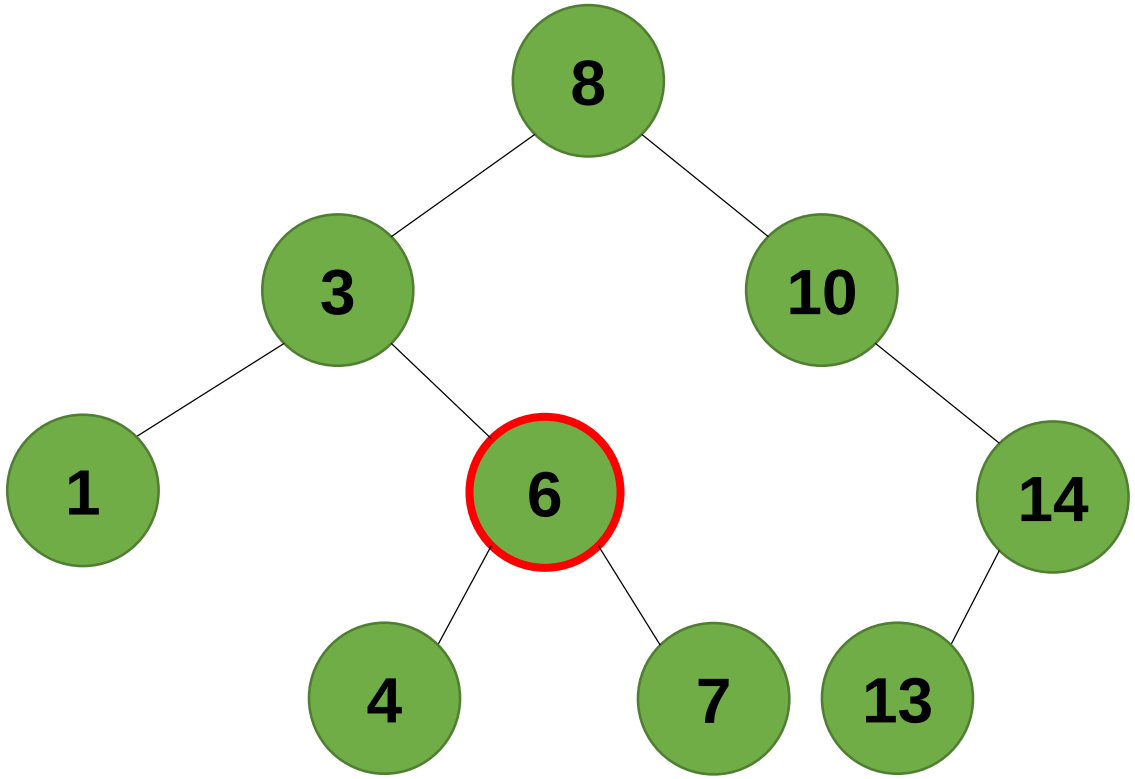
Searching in a BST




```
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);
}
```

Move to the right child if the key is greater than the selected node and left if it is smaller.

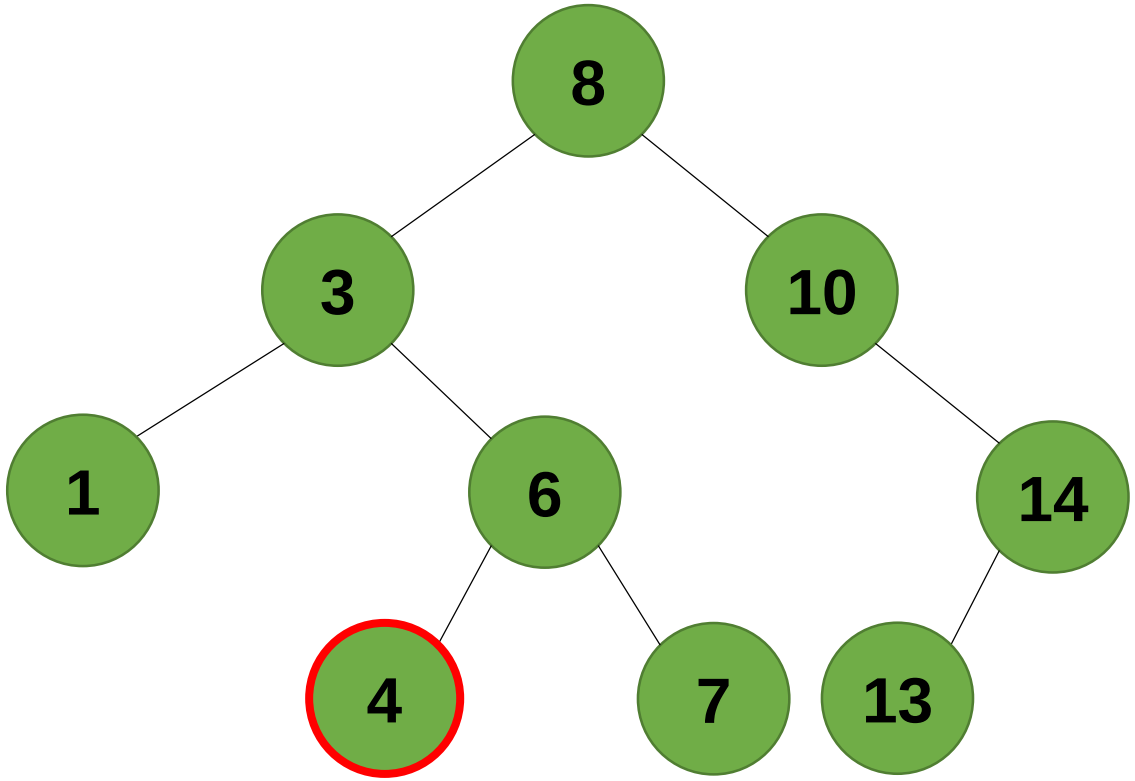
Searching in a BST




```
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);
}
```

Move to the right child if the key is greater than the selected node and left if it is smaller.

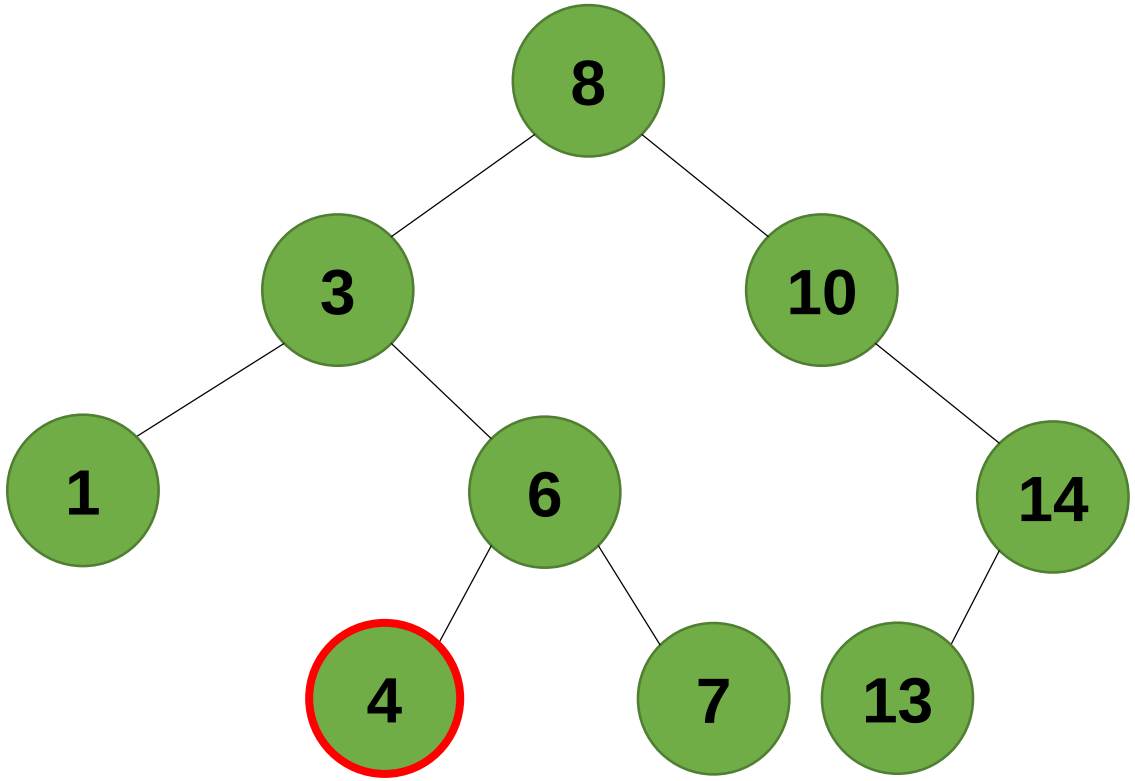
Searching in a BST




```
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);
}
```

Move to the right child if the key is greater than the selected node and left if it is smaller.

Searching in a 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);
}
```

Move to the right child if the key is greater than the selected node and left if it is smaller.

Insertion and Searching in a BST

Complexity:

h is height of Binary Search Tree.

Insertion and Searching in a BST

Complexity:

h is height of Binary Search Tree.

The worst case time complexity: O(h)

Insertion and Searching in a BST

Complexity:

h is height of Binary Search Tree.

The worst case time complexity: O(h)

If the height of a skewed tree becomes n, the time complexity is O(n).

Insertion and Searching in a BST

Complexity:

h is height of Binary Search Tree.

The worst case time complexity: O(h)

If the height of a skewed tree becomes n, the time complexity is O(n).

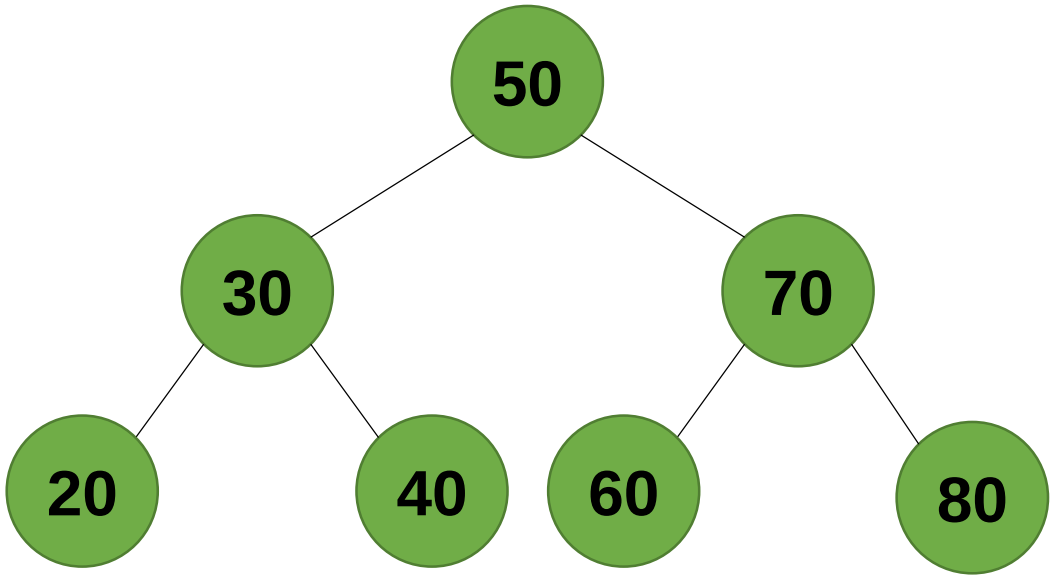
On average, h ≈ log n ➔ O(log n)

Deletion in a BST

Deletion in BST has been divided into 3 cases.

1) **Node to be deleted is leaf.**
2) **Node to be deleted has only one child.**
3) **Node to be deleted has two children.**

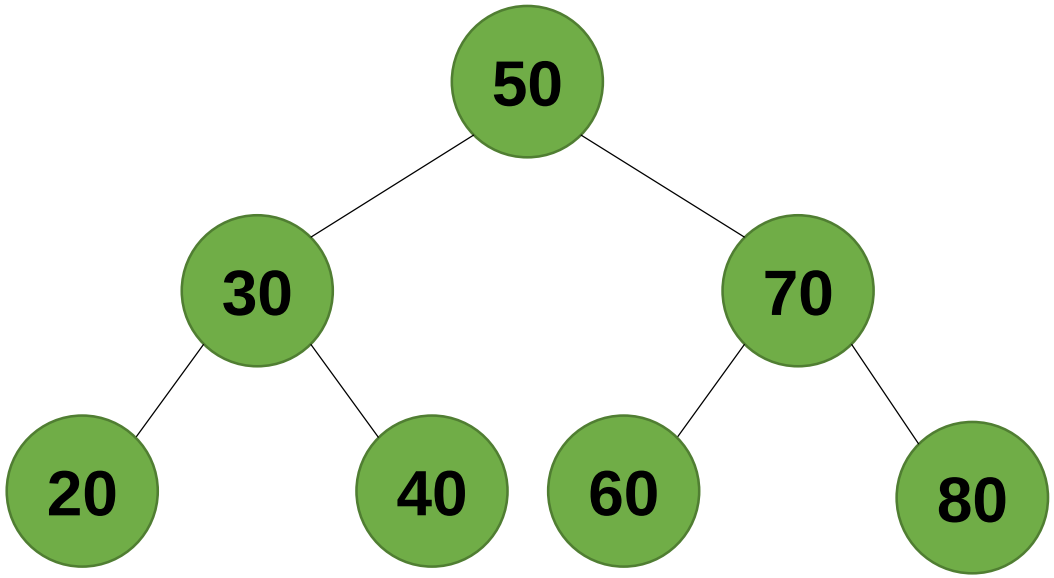
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

Case 1: Node to be deleted is leaf: Let's delete the node with value '20'

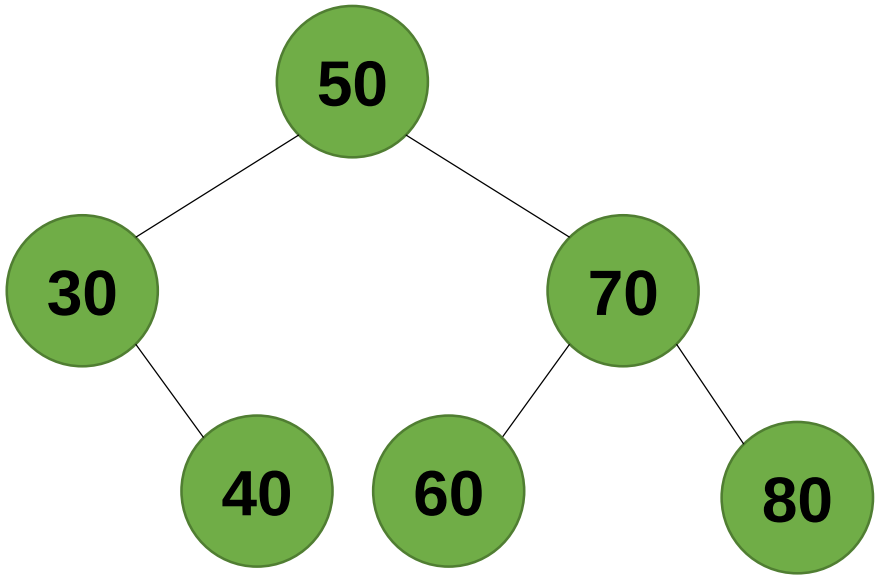
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

      root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

To delete a leaf node simply remove it from the tree.

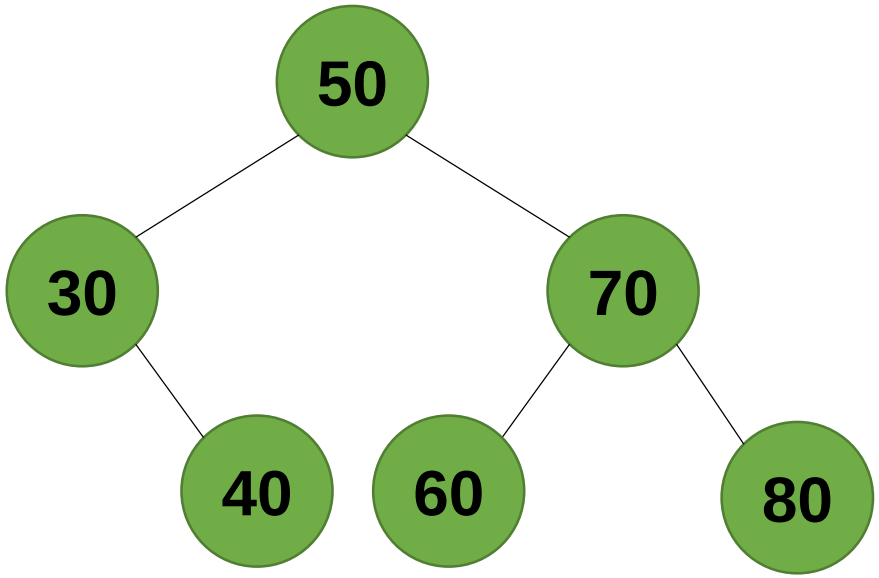
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

      root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

To delete a leaf node simply remove it from the tree.

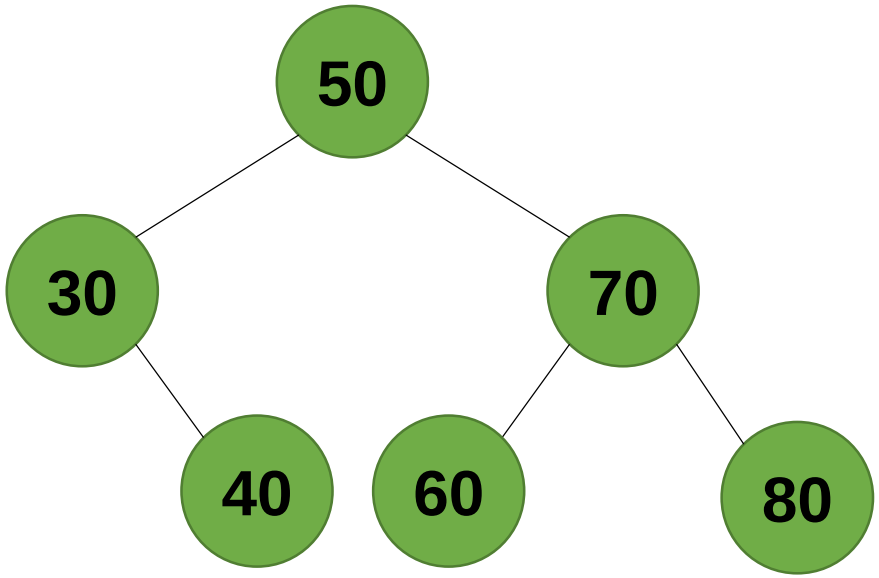
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

      root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

Since it was a leaf node, we deleted it from the tree without making any other changes.

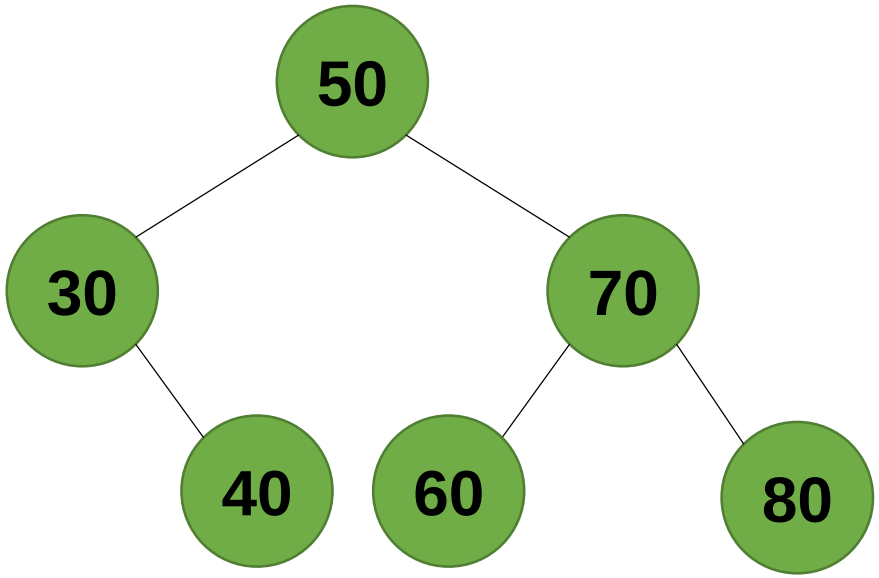
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

Case 2: Node to be deleted has only one child: Let's delete the node with value '30'

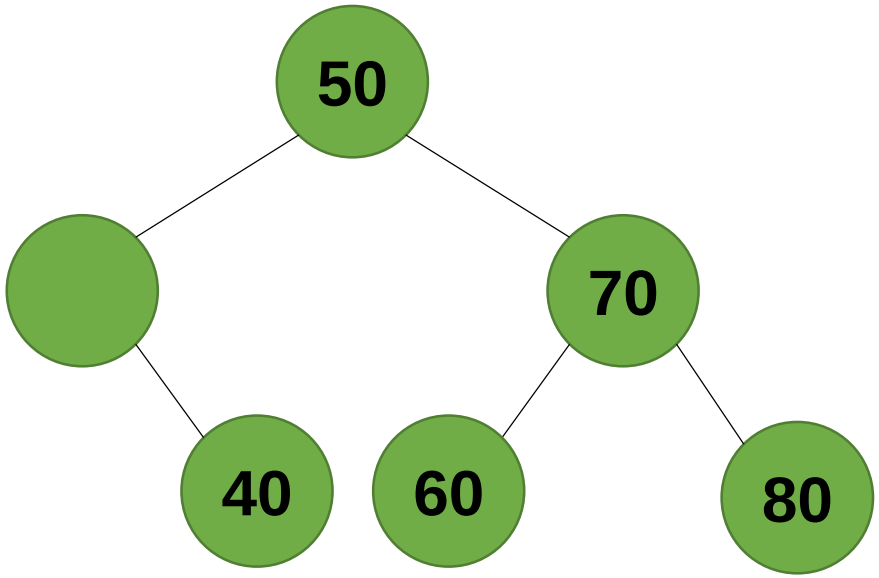
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

To delete this node, copy the child to the node and delete the child.

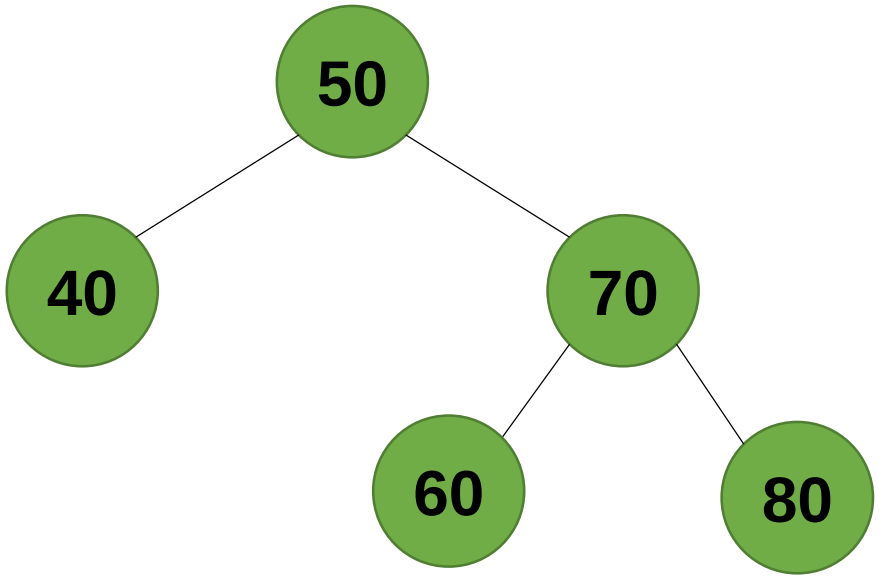
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

To delete this node, copy the child to the node and delete the child.

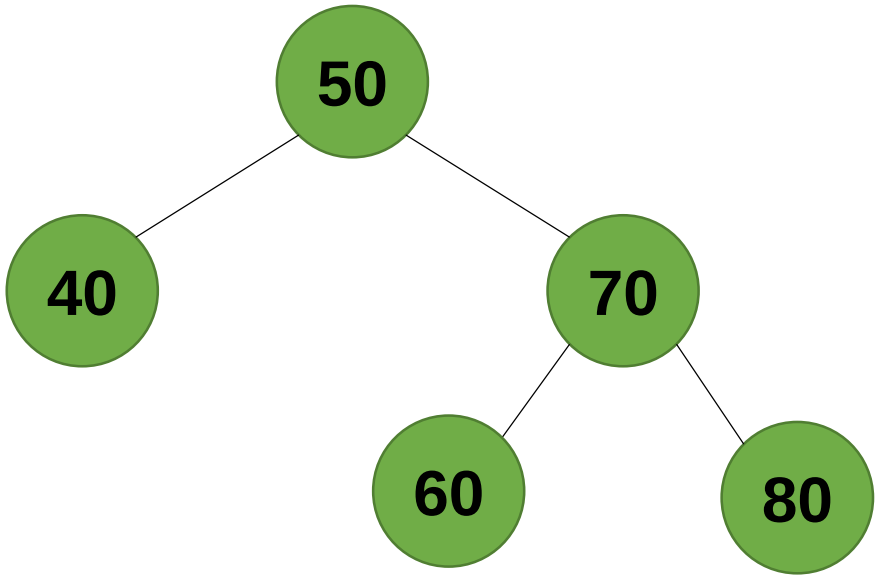
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

To delete this node, copy the child to the node and delete the child.

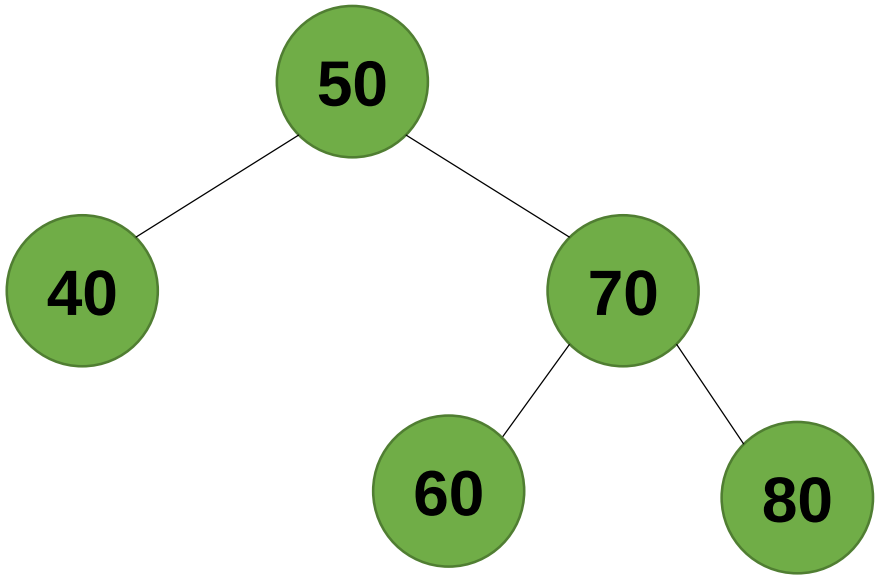
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

The node had only one child so we copied its child's value to it and deleted its child.

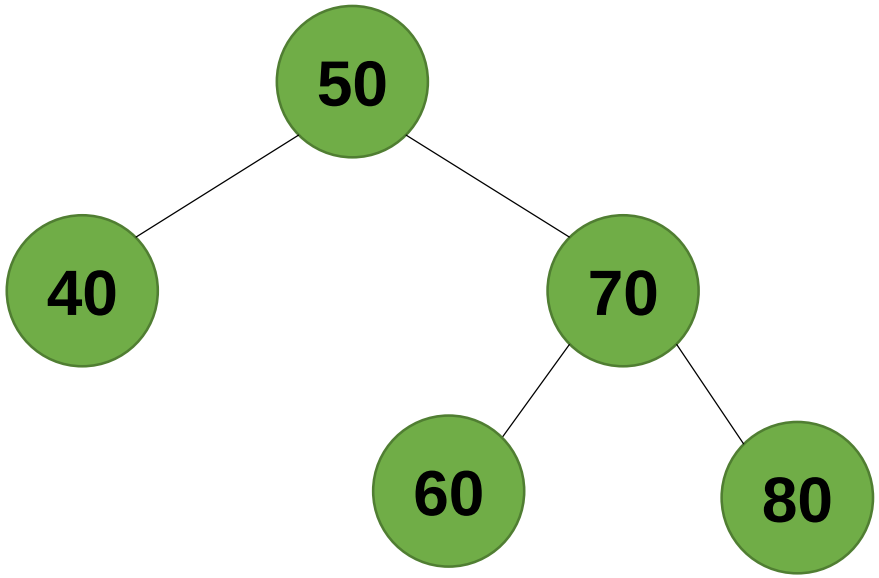
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

Case 3: Node to be deleted has two children: Let's delete the root node with value '50'

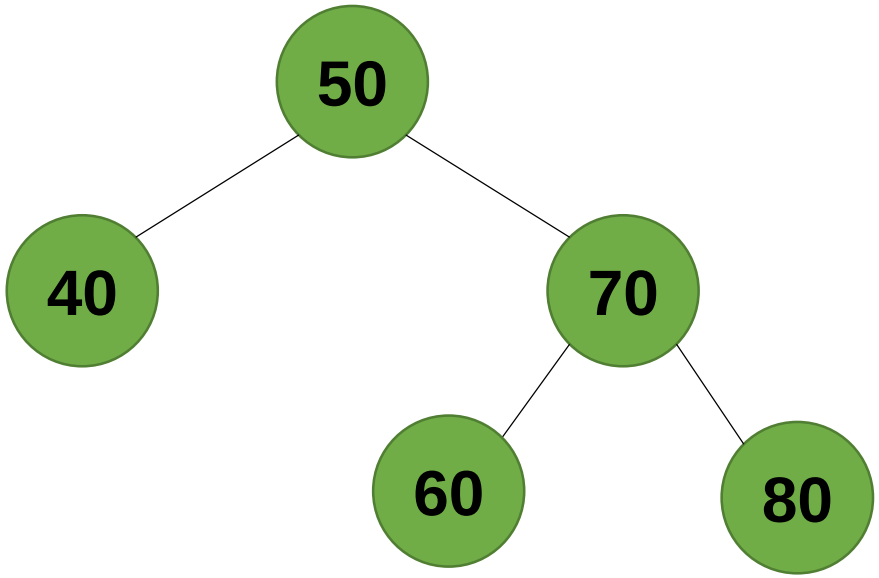
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }
        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

      root->right = deleteNode(root->right, temp->key);

      }
      return root;
}
```

First, find inorder successor of the node.

Deletion in a BST

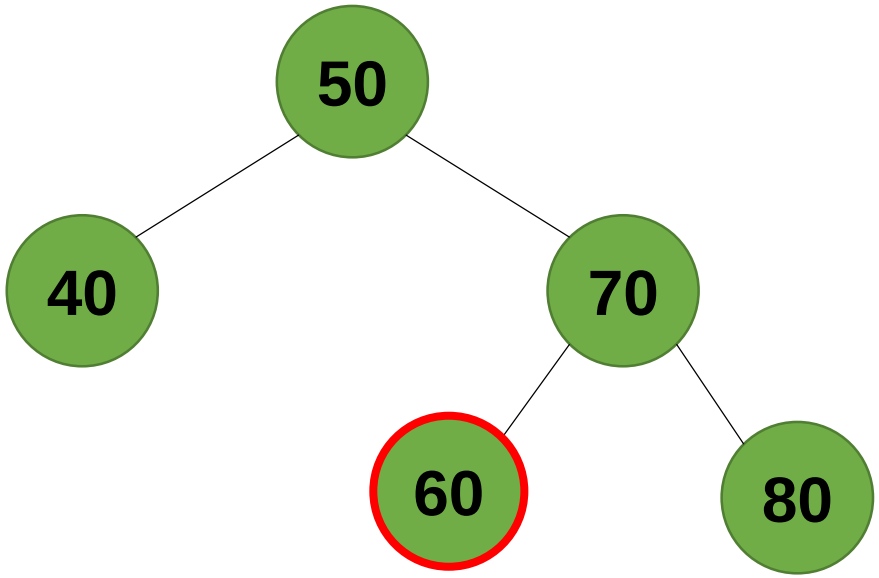


40 50 60 70 80: Inorder Traversal

First, find inorder successor of the node.

```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    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;
        }
        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

      root->right = deleteNode(root->right, temp->key);

      }
      return root;
}
```

Deletion in a BST

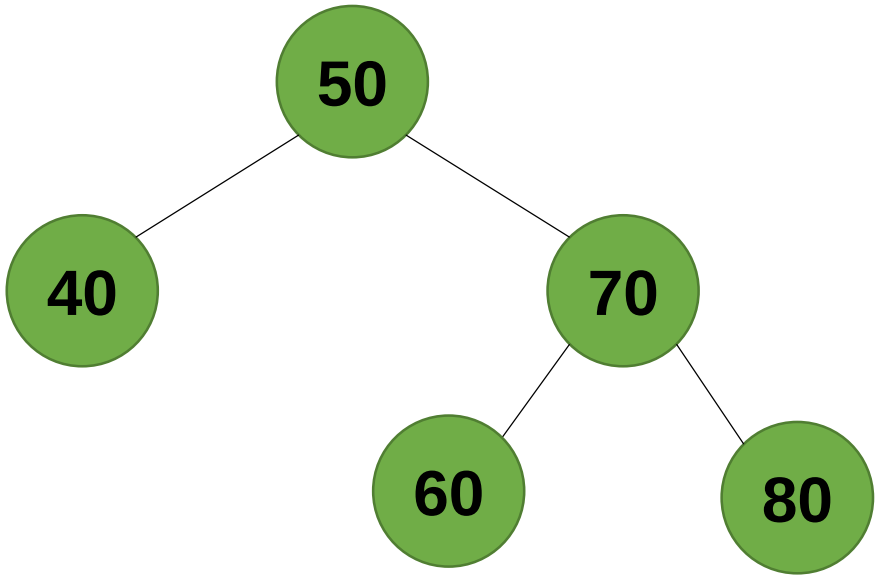


40 50 60 70 80: Inorder Traversal

First, find inorder successor of the node.

```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    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;
        }
        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

      root->right = deleteNode(root->right, temp->key);

      }
      return root;
}
```

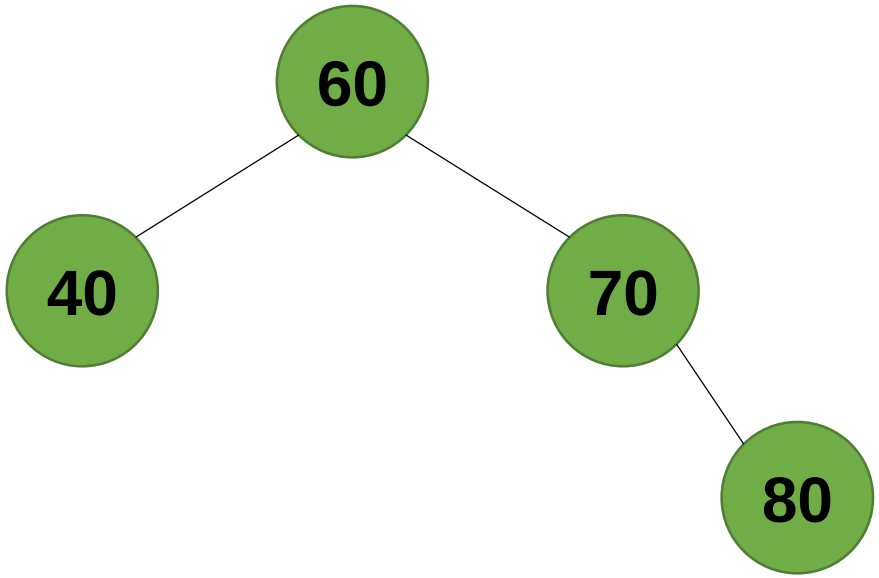
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

Now copy contents of the inorder successor to the node and delete the inorder successor.

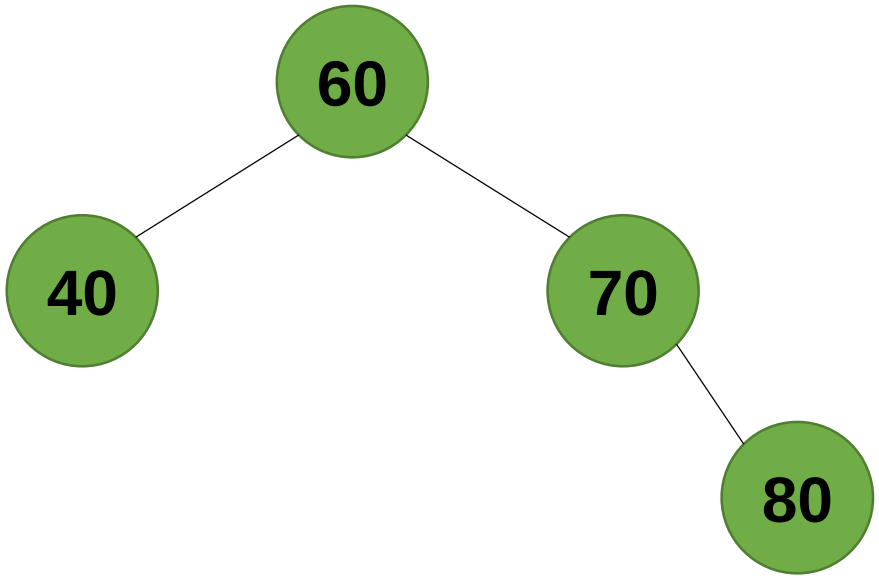
Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

Now copy contents of the inorder successor to the node and delete the inorder successor.

Deletion in a BST




```
struct node* deleteNode(struct node* root, int ke
{

    if (root == NULL) return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    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;
        }

        struct node* temp = minValueNode(root->right);

            root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }
    return root;
}
```

The inorder predecessor can also be used in the same manner.

Deletion in a BST

Complexity:

h is height of Binary Search Tree.

Deletion in a BST

Complexity:

h is height of Binary Search Tree.

The worst case time complexity: O(h)

If the height of a skewed tree becomes n, the time complexity is O(n).

On average, h ≈ log n ➔ O(log n)

BST vs Hash Table

Hash Table supports following operations in $\Theta(1)$ time:

1. **Search**
2. **Insert**
3. **Delete**

For a *Self Balancing Binary Search Tree* the time complexity for these oprations is:

$\mathbf{O(\log n)}$

Advantages of BST over Hash Table

1. **Can get all keys in sorted order by just doing Inorder Traversal of BST**

Advantages of BST over Hash Table

1. **Can get all keys in sorted order by just doing Inorder Traversal of BST**
2. **Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs**

Advantages of BST over Hash Table

1. **Can get all keys in sorted order by just doing Inorder Traversal of BST**
2. **Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs**
3. **BSTs are easy to implement compared to hashing**

Advantages of BST over Hash Table

1. Can get all keys in sorted order by just doing Inorder Traversal of BST
2. Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs
3. BSTs are easy to implement compared to hashing
4. With Self Balancing BSTs, all operations are guaranteed to work in $O(\log n)$ time

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms”,* September 29, 2004

- https://www.geeksforgeeks.org