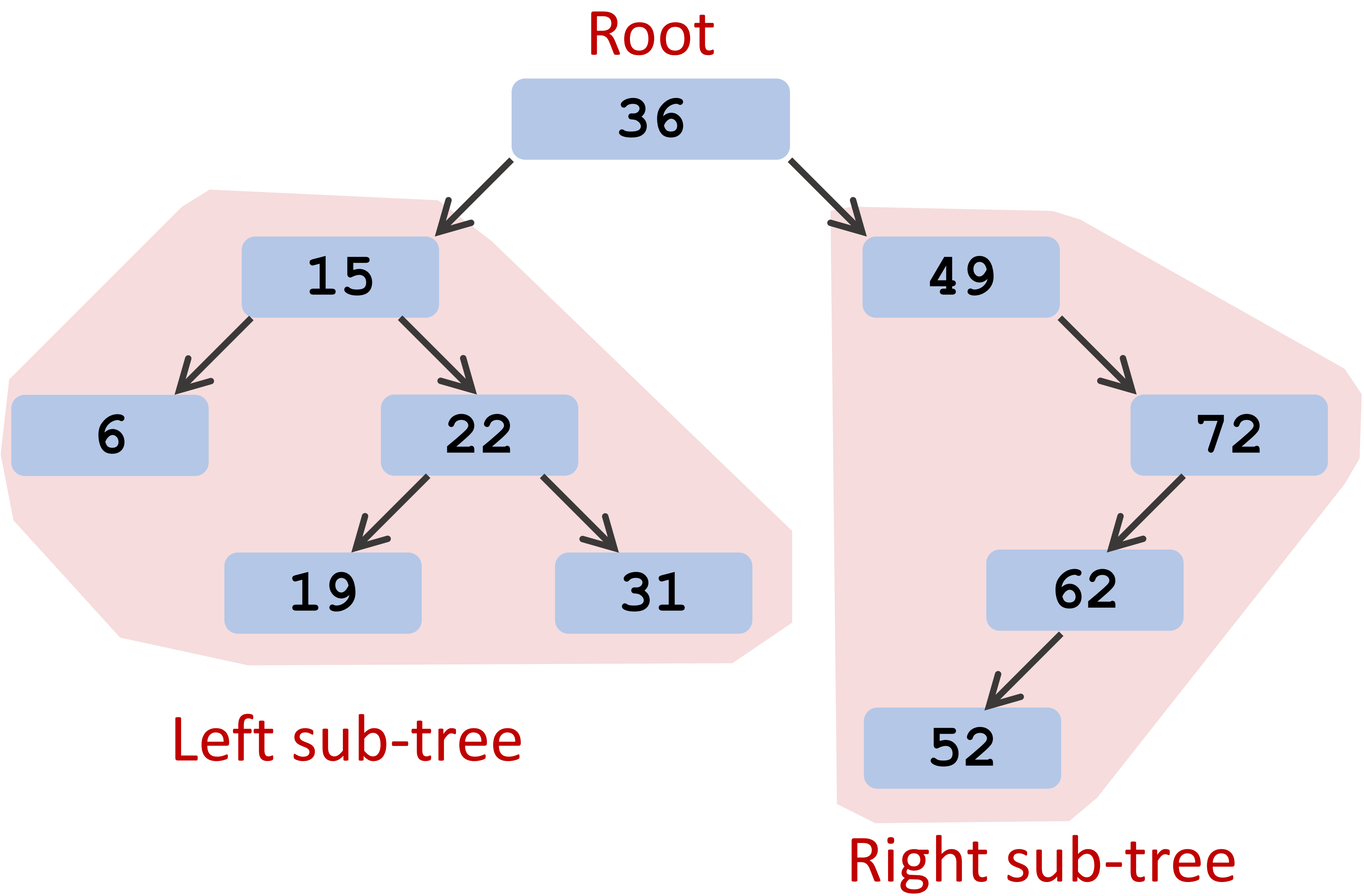
Binary Search Tree (1/2): Traversal, Search, and Insertion

Shusen Wang

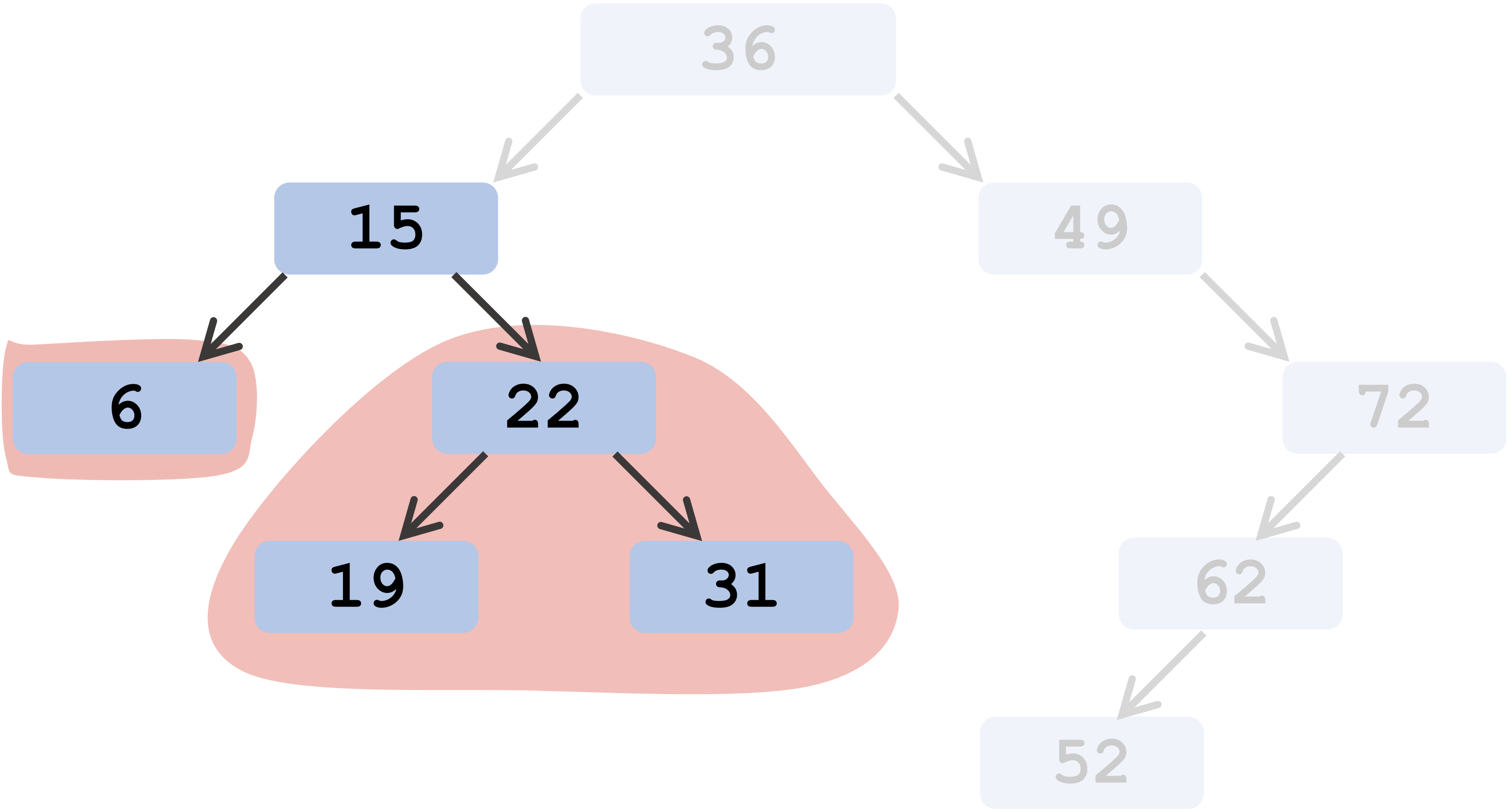
Binary Search Tree



Property:

- All the keys in the left sub-tree are smaller than the root's key.
- All the keys in the right sub-tree are greater than the root's key.

Binary Search Tree



Property:

- All the keys in the left sub-tree are smaller than the root's key.
- All the keys in the right sub-tree are greater than the root's key.

Traversal

Print all the keys

Tree:

```
        50
      /     \
    30        70
   /  \         \
20    40         80
```

Print:

```
20
30
40
50
70
80
```

Print all the keys

Tree:	Print:
50	50

Print all the keys

Tree:

```
     50
   /     \
 30       70
```

Print:

30

50

70

Print all the keys

Tree:

```
    50
   /  \
  30   70
 / \     \
20  40    80
```

Print:

20
30
40
50
70
80

Print all the keys

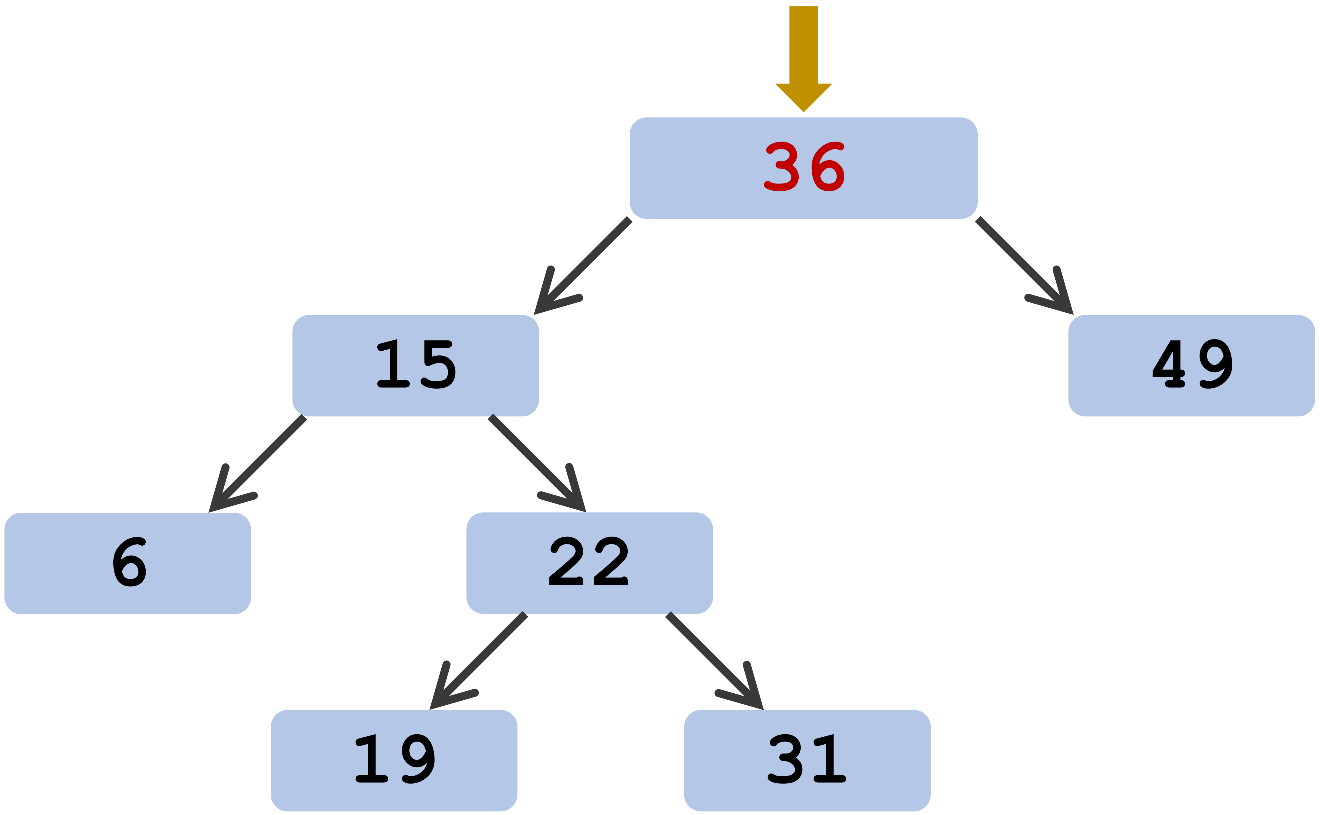
```
void traverse(struct vertex *root) {
    if (root != NULL) {
        traverse(root->left);
        cout << root->key << endl;
        traverse(root->right);
    }
}
```

Search

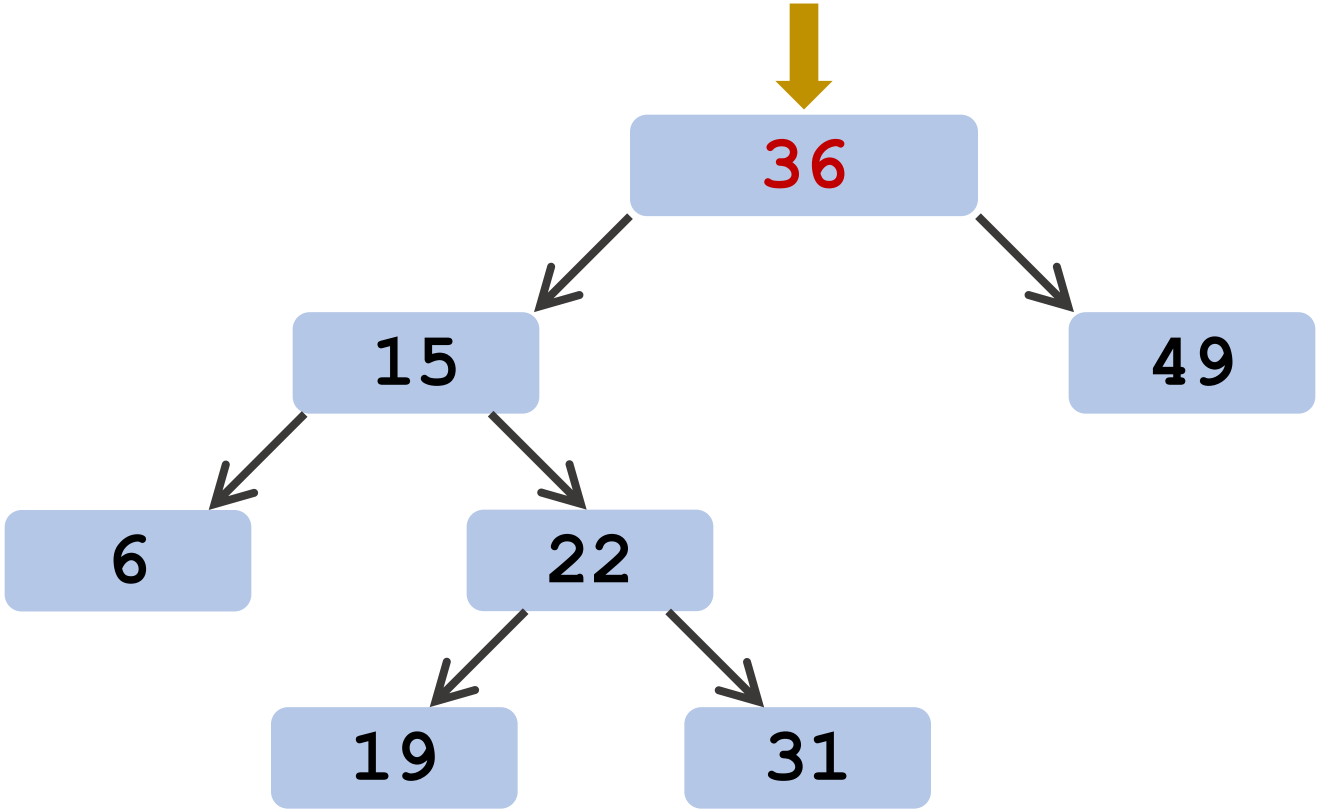
Search

- **Inputs: root** (of the tree) and **key** (to be matched).
- **Goal:** find the vertex which matches the input key.
- **Output: the vertex** (if found) or **`NULL`** (if not found).
- **Time complexity:** depth of the tree.

Search "key=36"

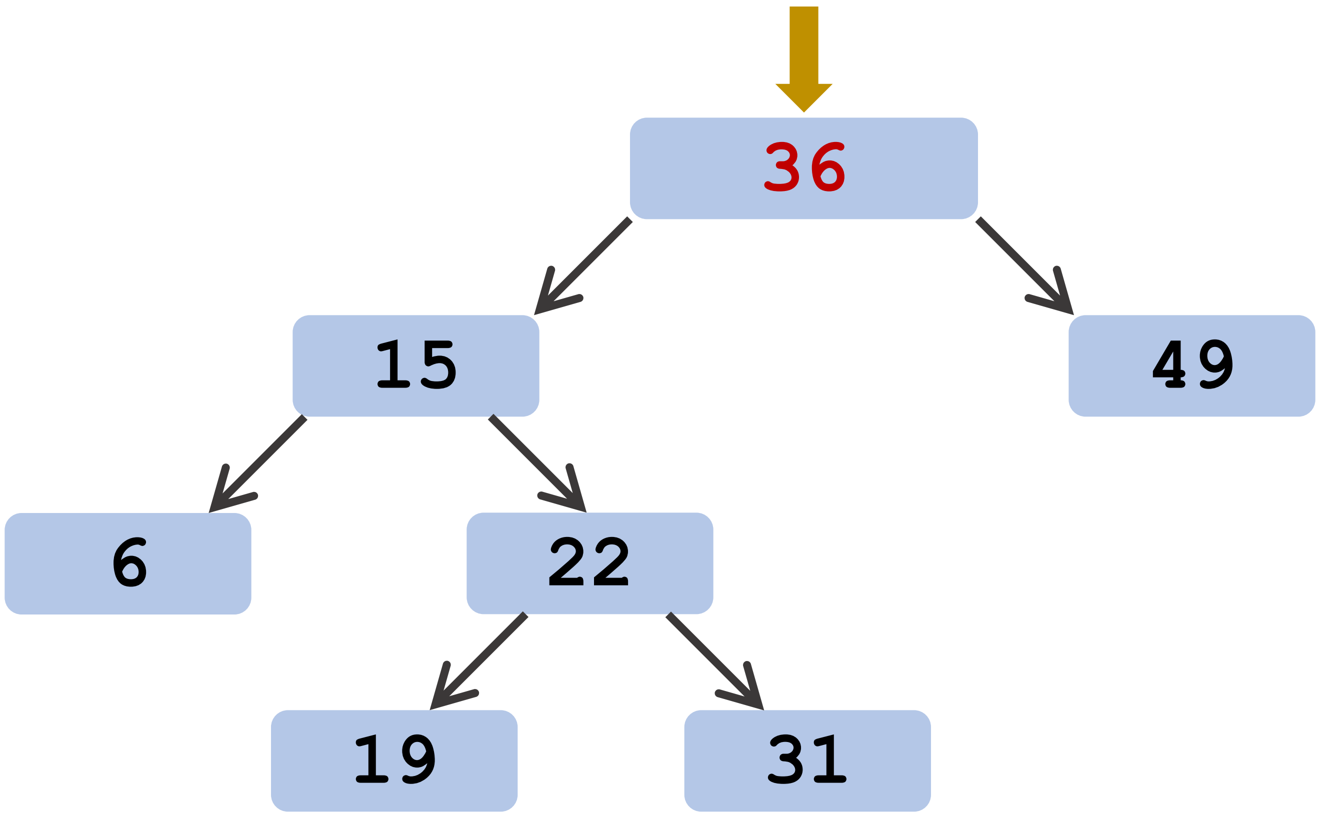


Search “key=36”

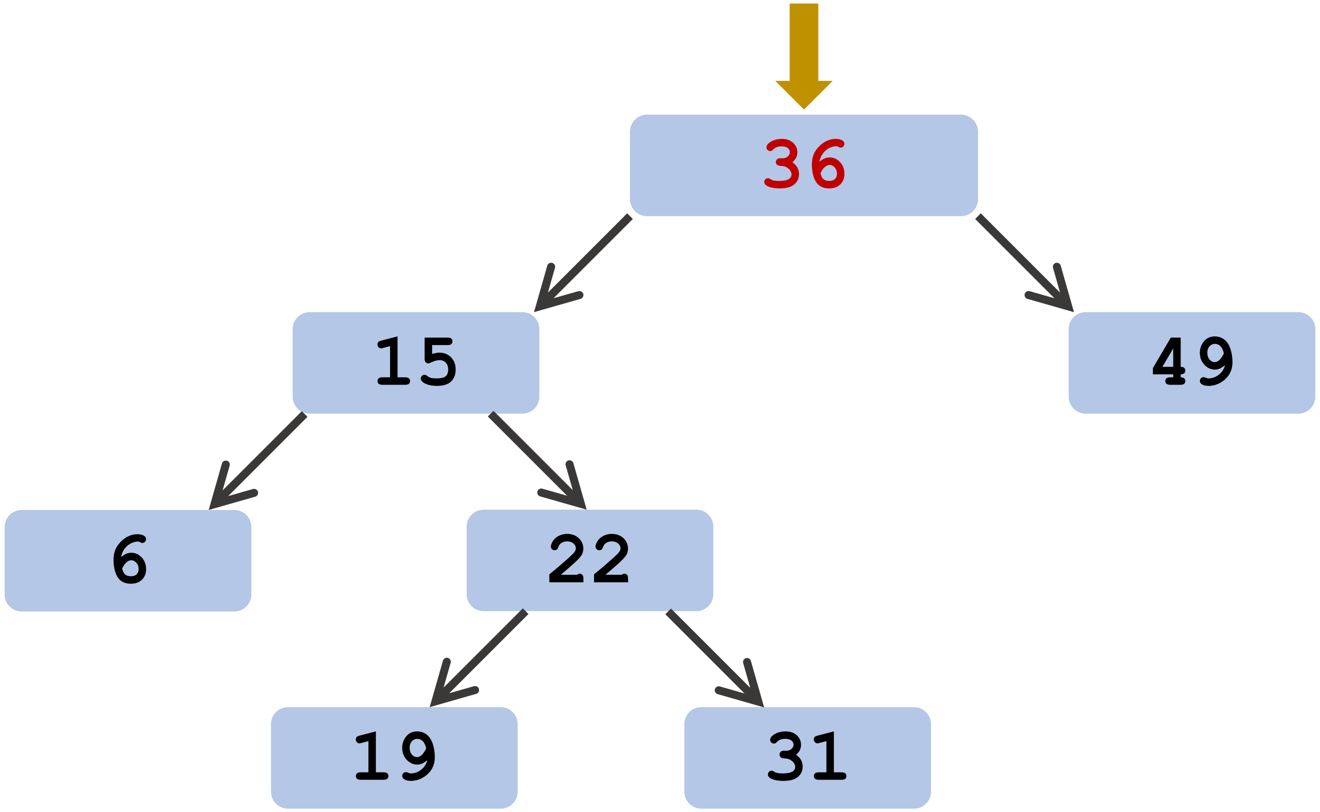


1. key == 36 ➔ Found!
2. Return the root

Search “key=22”

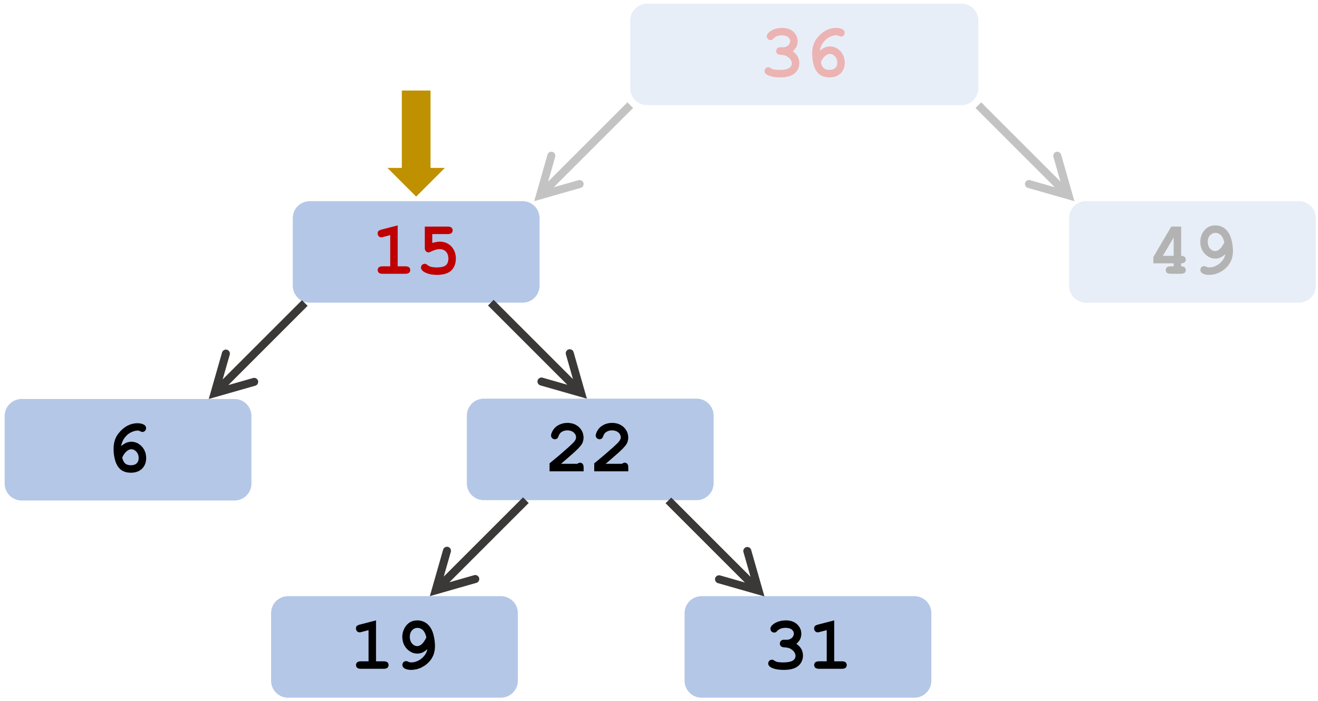


Search "key=22"



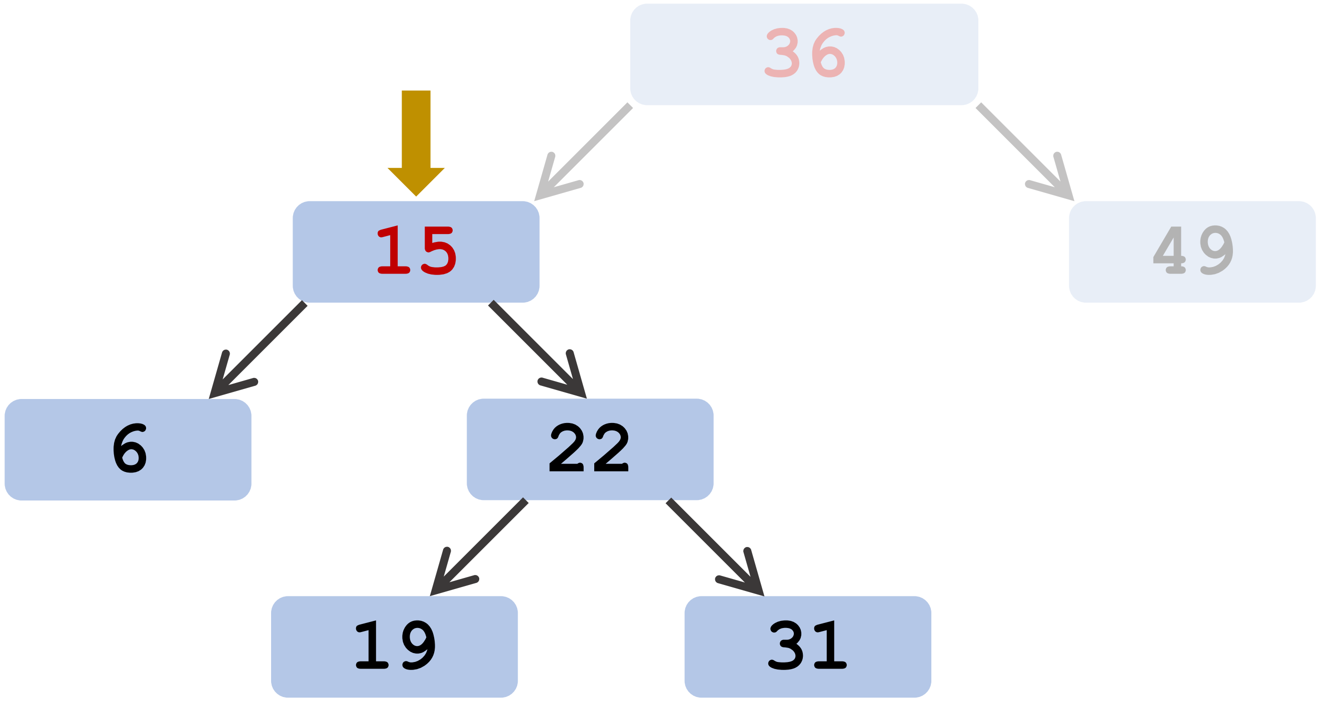
1. key < 36 ➔ Go to left.

Search “key=22”



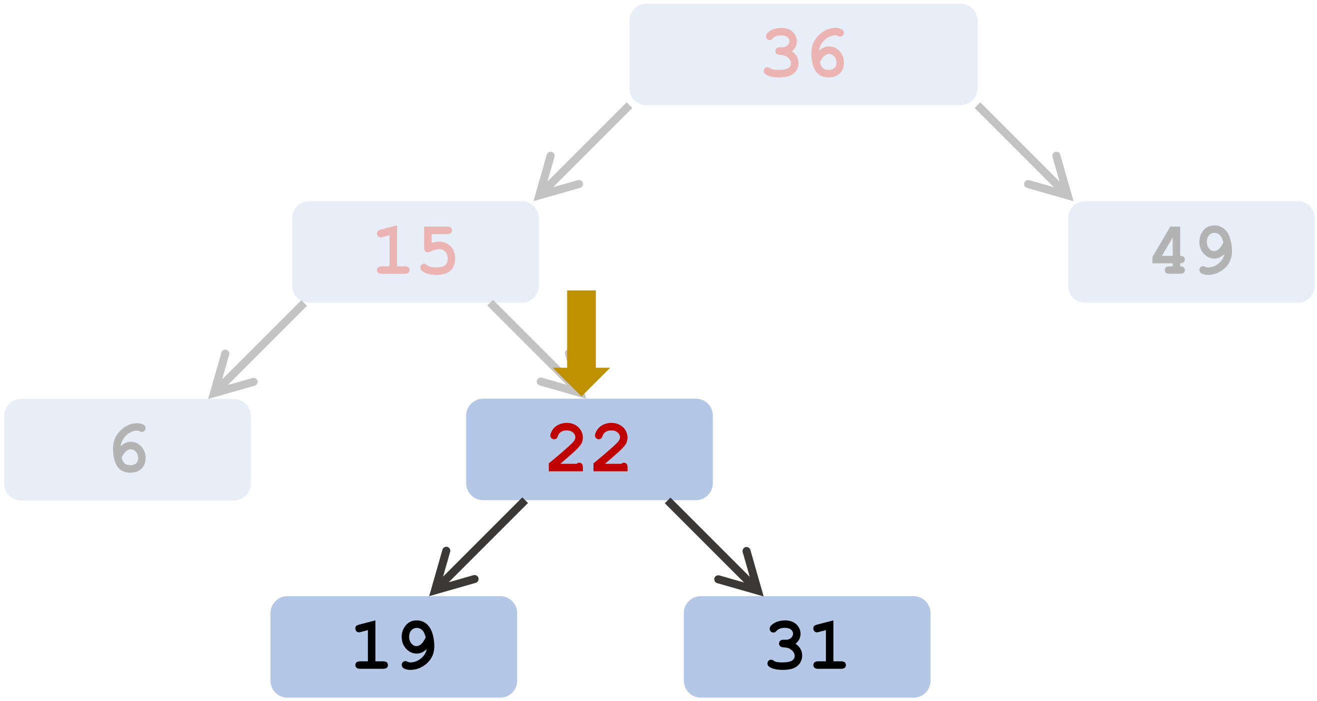
1. key < 36 ➔ Go to left.

Search “key=22”



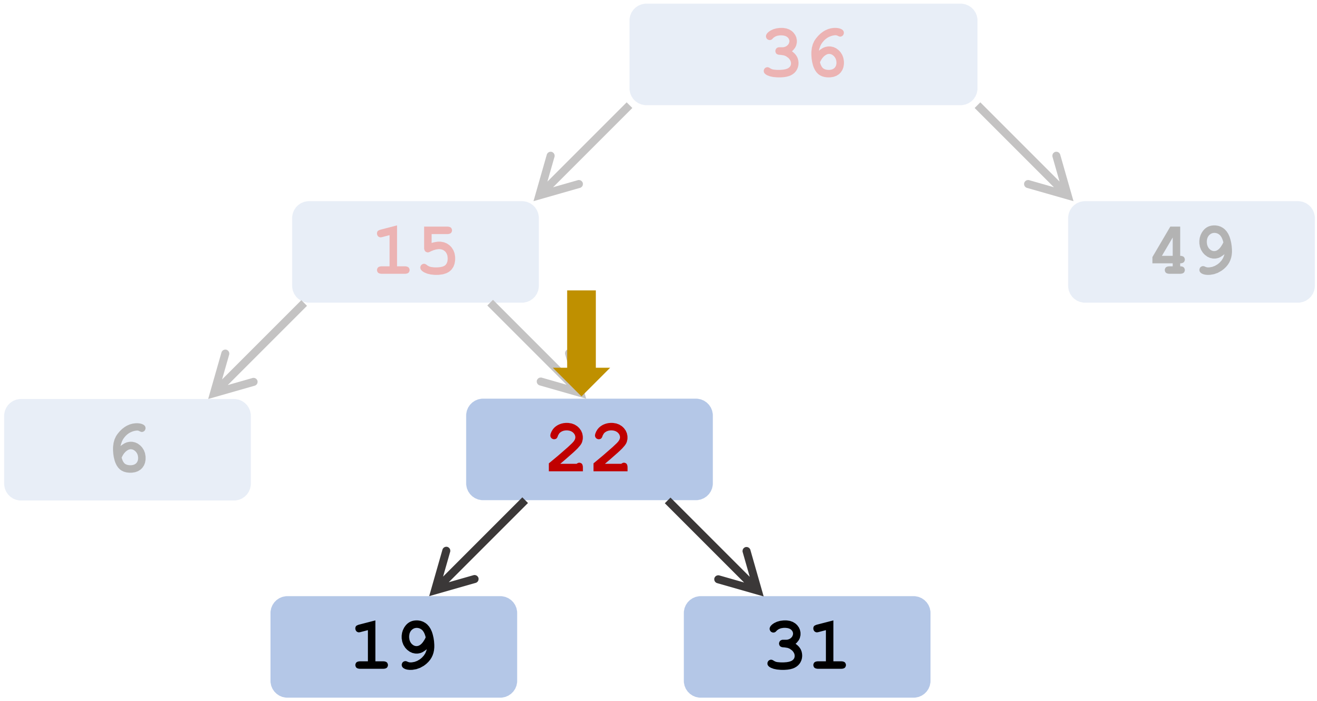
1. key < 36 ➔ Go to left.
2. key > 15 ➔ Go to right.

Search “key=22”



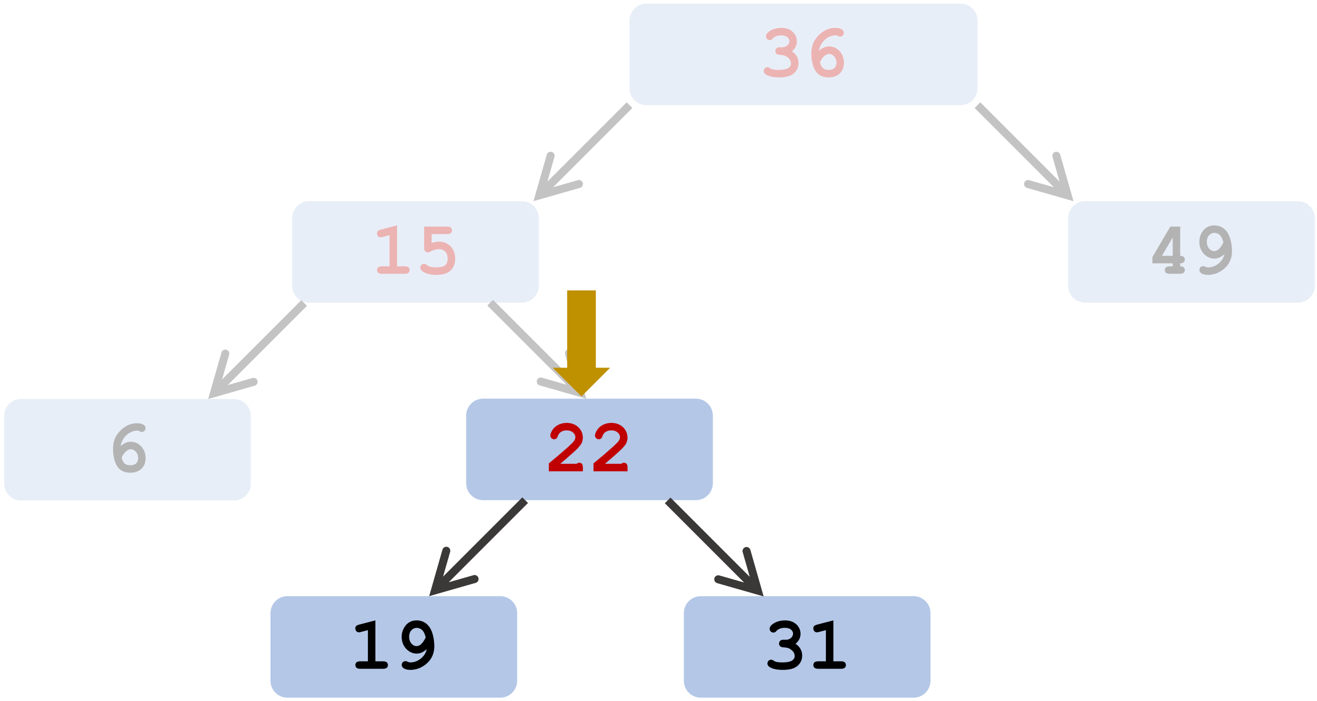
1. key < 36 ➔ Go to left.
2. key > 15 ➔ Go to right.

Search “key=22”



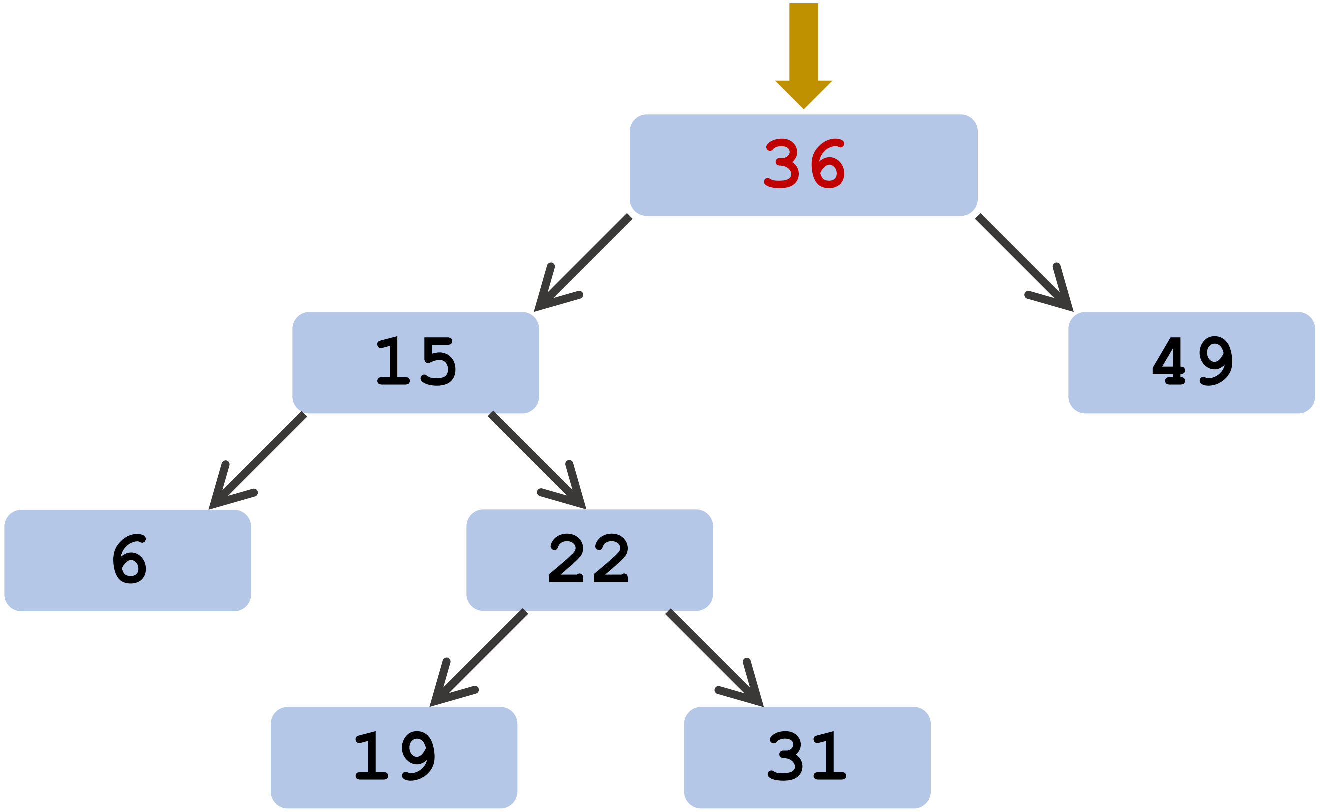
1. key < 36 ➔ Go to left.
2. key > 15 ➔ Go to right.
3. key == 22 ➔ Found!

Search “key=22”

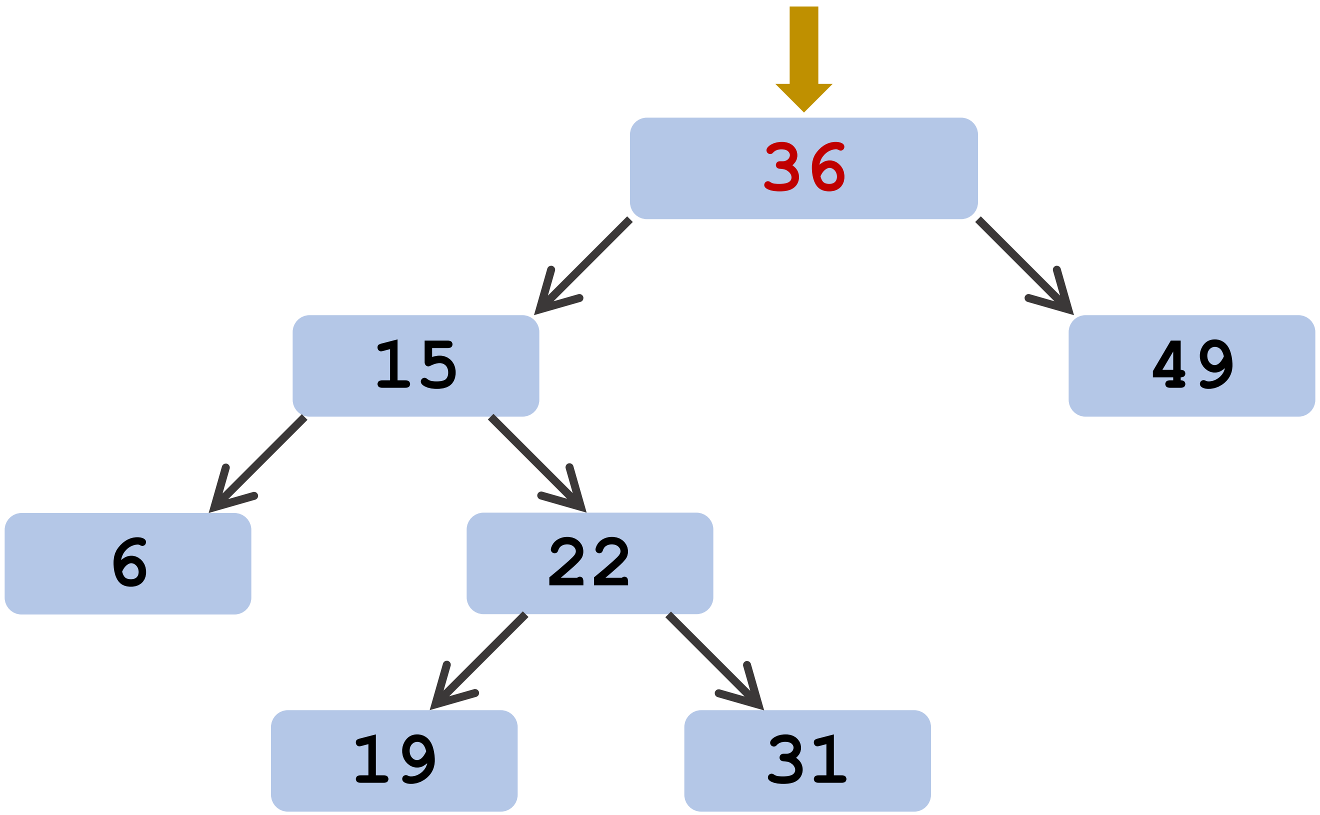


1. key < 36 ➔ Go to left.
2. key > 15 ➔ Go to right.
3. key == 22 ➔ Found!
4. Return the node 22

Search “key=40”

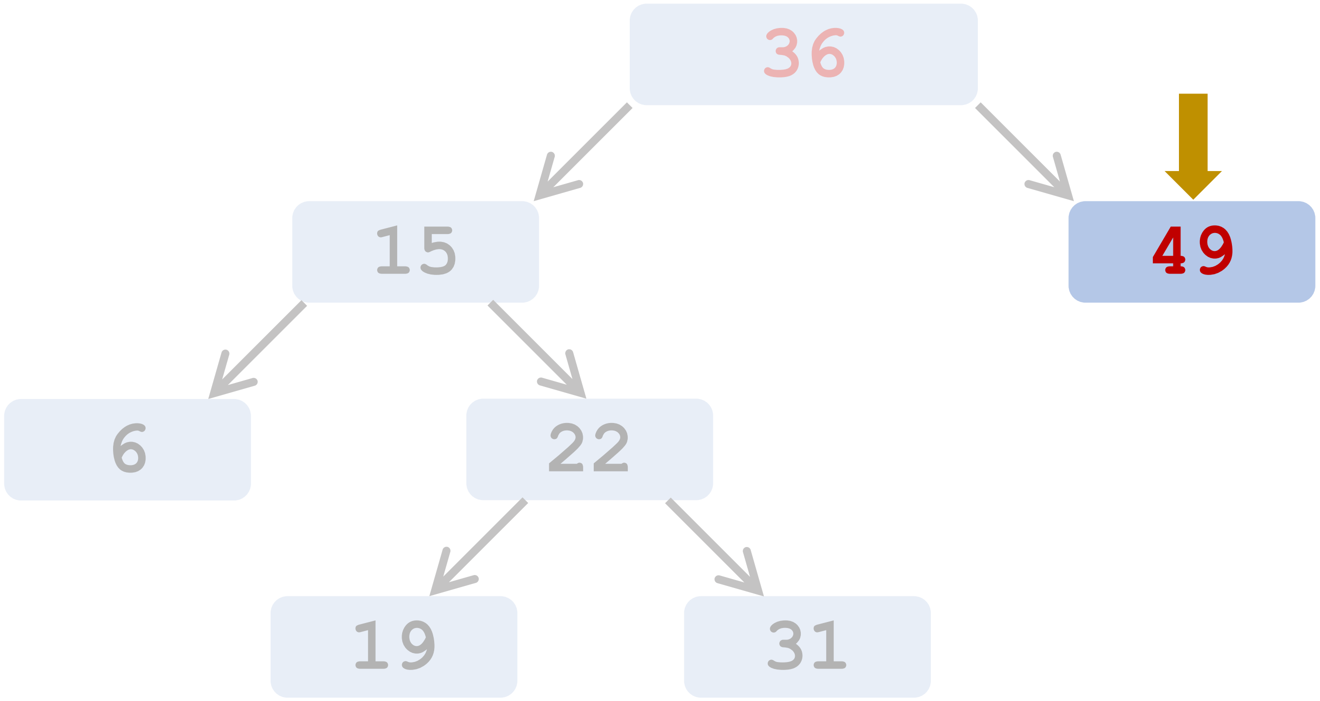


Search “key=40”



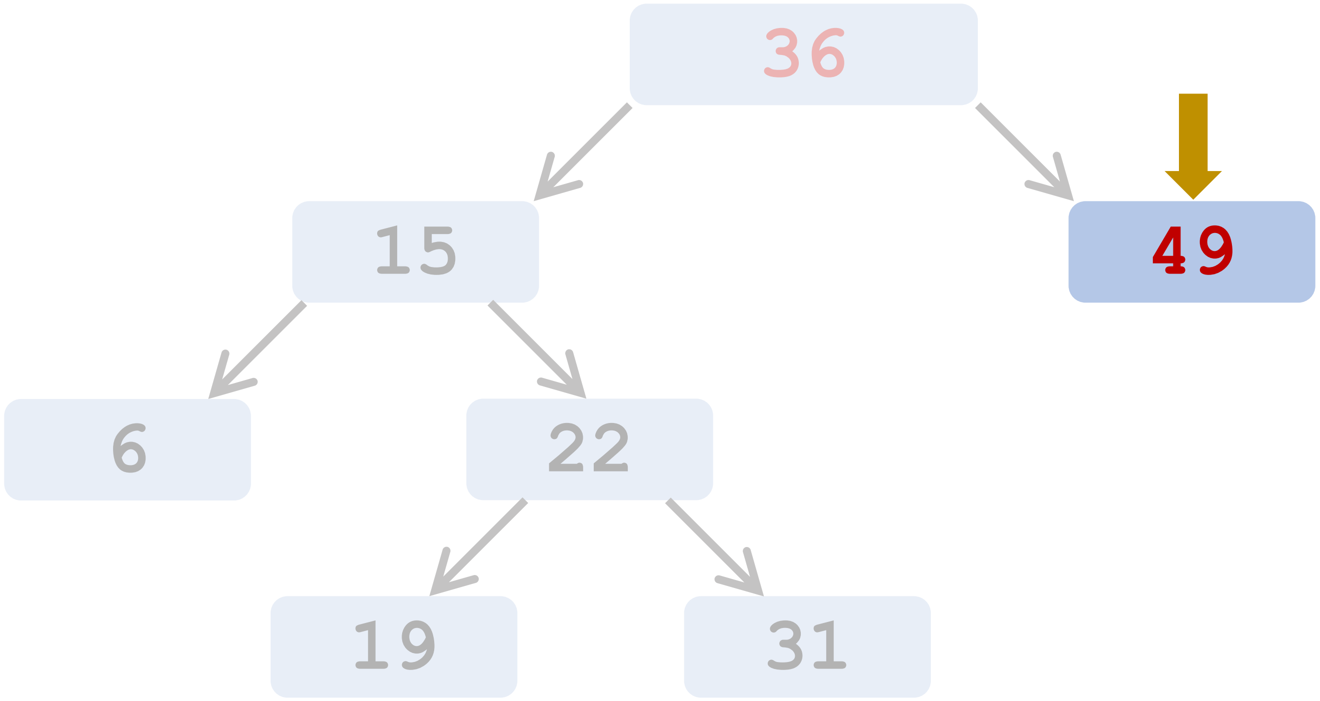
1. key > 36 ➔ Go to right.

Search “key=40”



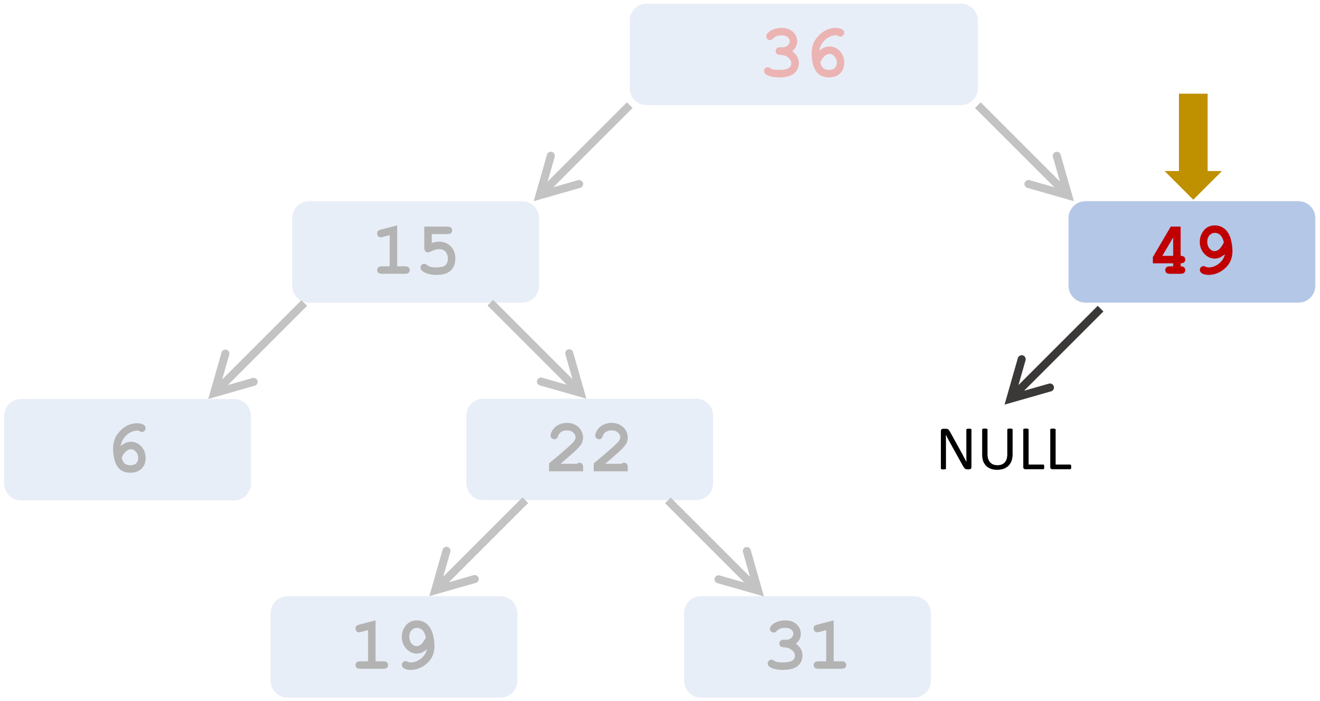
1. key > 36 ➔ Go to right.

Search “key=40”



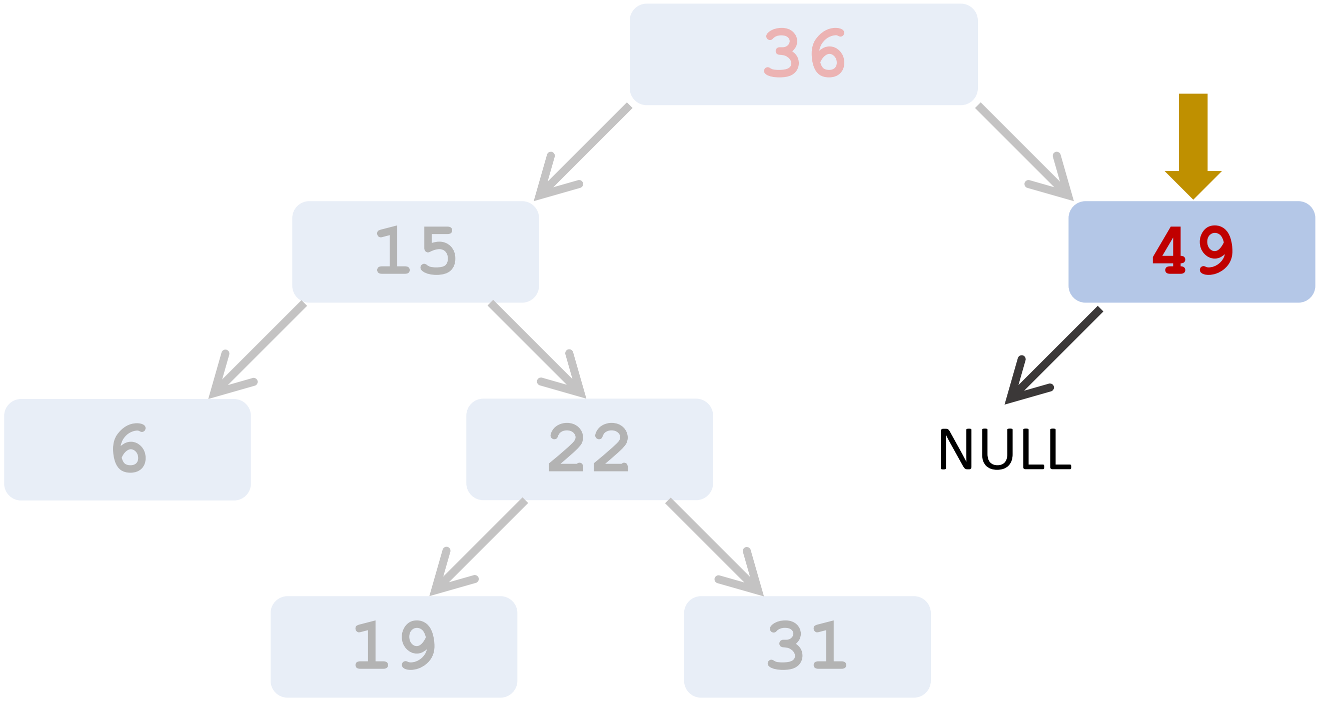
1. key > 36 ➔ Go to right.
2. key < 49 ➔ Go to left.

Search “key=40”



1. key > 36 ➔ Go to right.
2. key < 49 ➔ Go to left.
3. The left child is NULL.

Search "key=40"



1. key > 36 ➔ Go to right.
2. key < 49 ➔ Go to left.
3. The left child is NULL.
4. Return NULL.

```
struct vertex* search(struct vertex* root, int key) {
    // empty tree or the key is present at root
    if (root == NULL || root->key == key)
        return root;
    // key is in the right subtree
    if (key > root->key)
        return search(root->right, key);
    // key is in the left subtree
    else
        return search(root->left, key);
}
```

Insertion

Insert

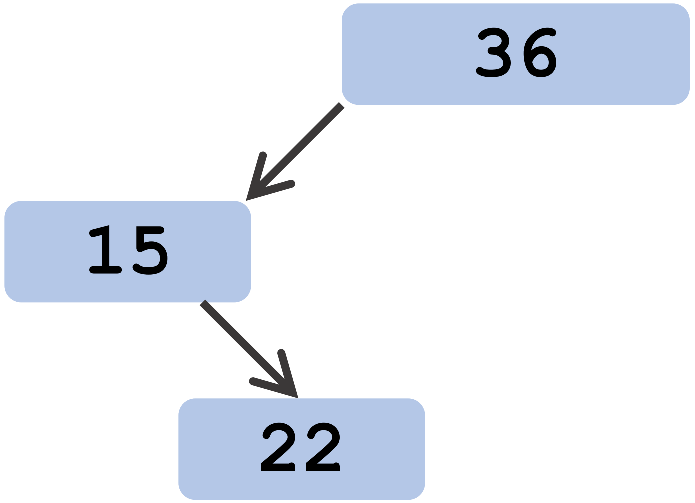
- **Inputs:** **root** (of the tree) and **key** (to be matched).
- **Goal:** create a new vertex and insert it into the correct position.
- **Time complexity:** depth of the tree.

Insert

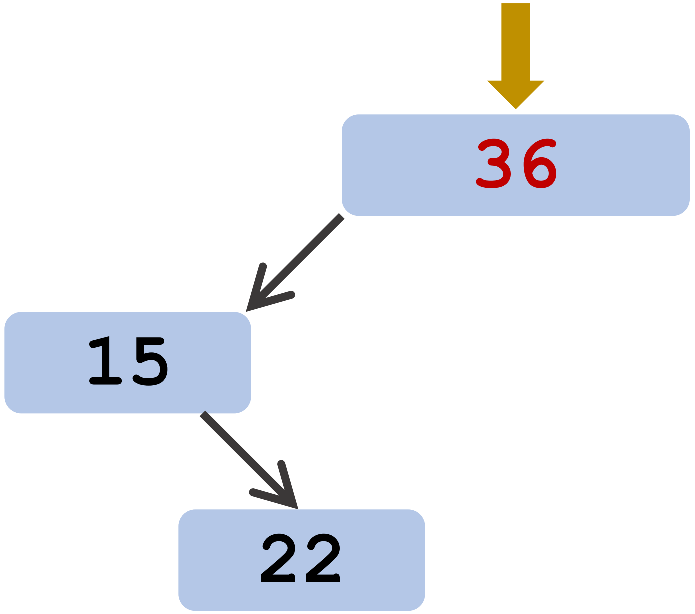
Question: What if the tree is empty (i.e., **`root==NULL`**)?

- Create a new vertex and make it the root.
- Return the root.

Insert "key=49"

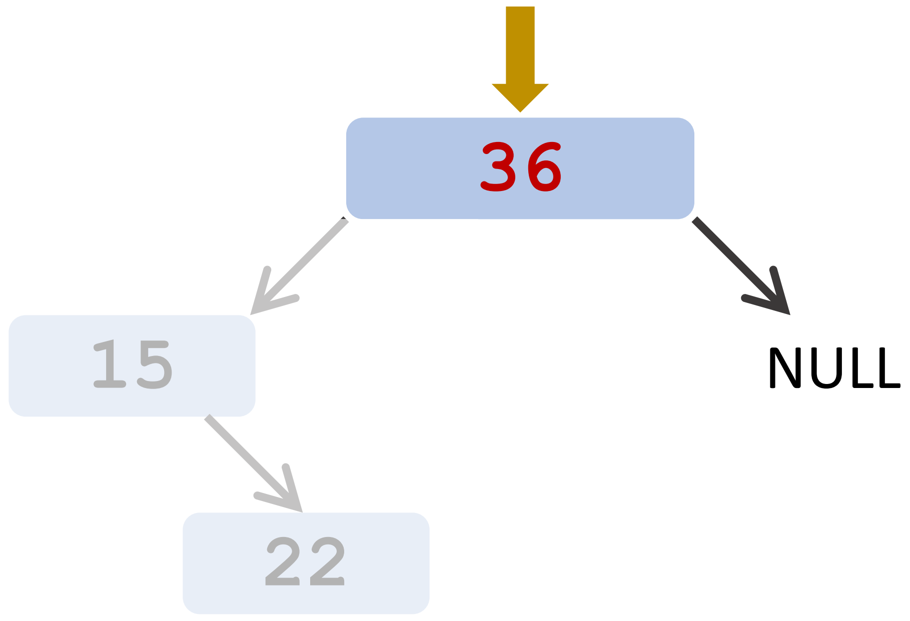


Insert "key=49"



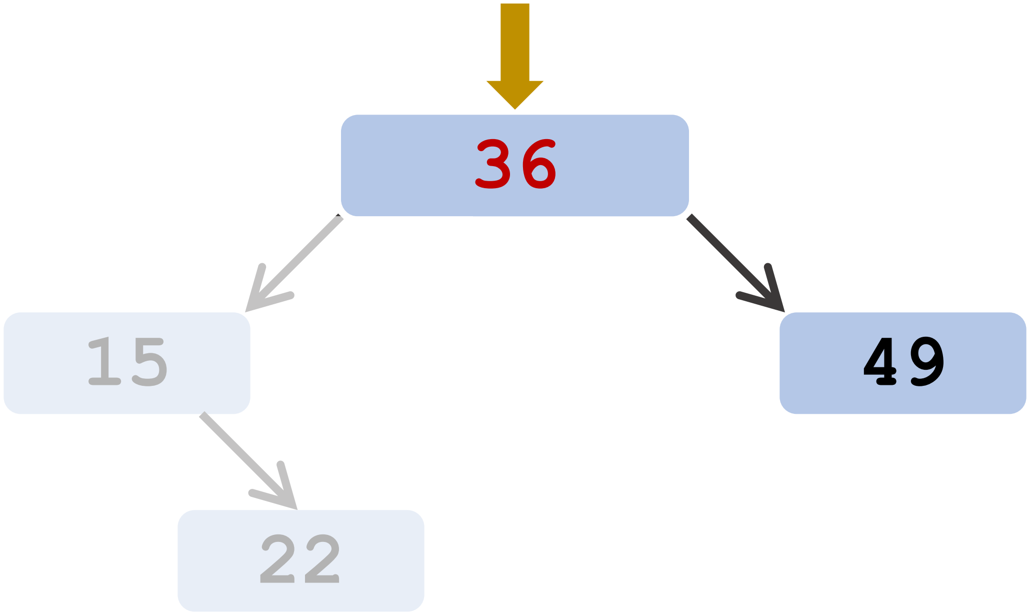
1. Create new node 49
2. key > 36 ➔ Go to right.

Insert "key=49"



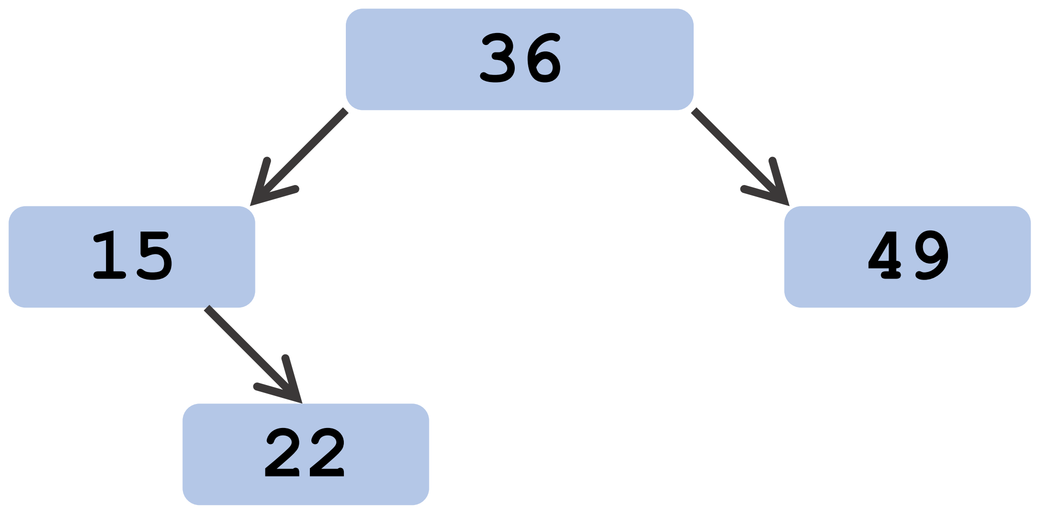
1. Create new node **49**
2. key > 36 ➔ Go to right.
3. Right child is NULL

Insert “key=49”

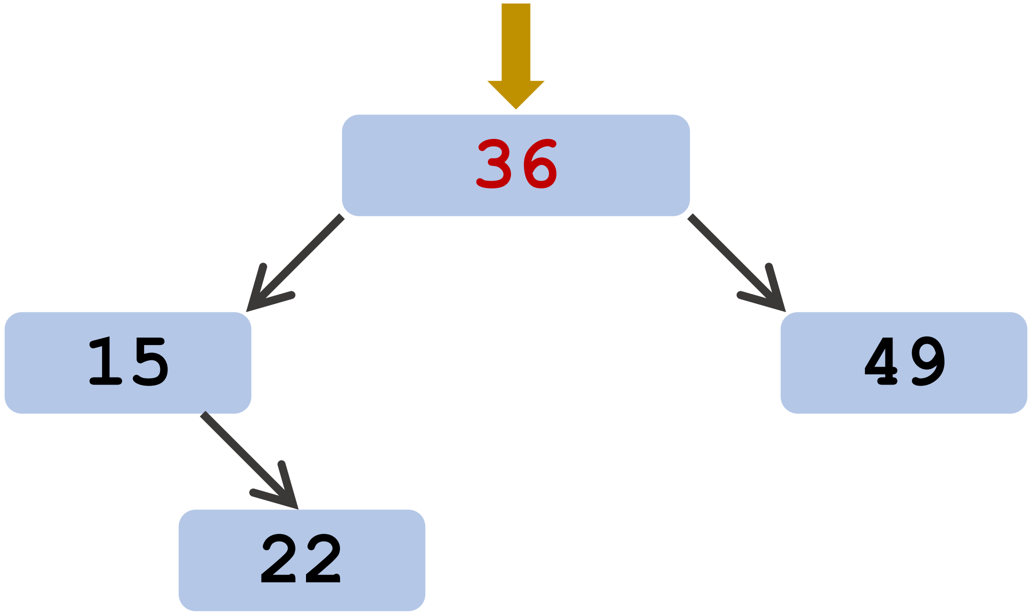


1. Create new node 49
2. key > 36 ➔ Go to right.
3. Right child is NULL ➔
   Make “49” the right child.

Insert “key=31”

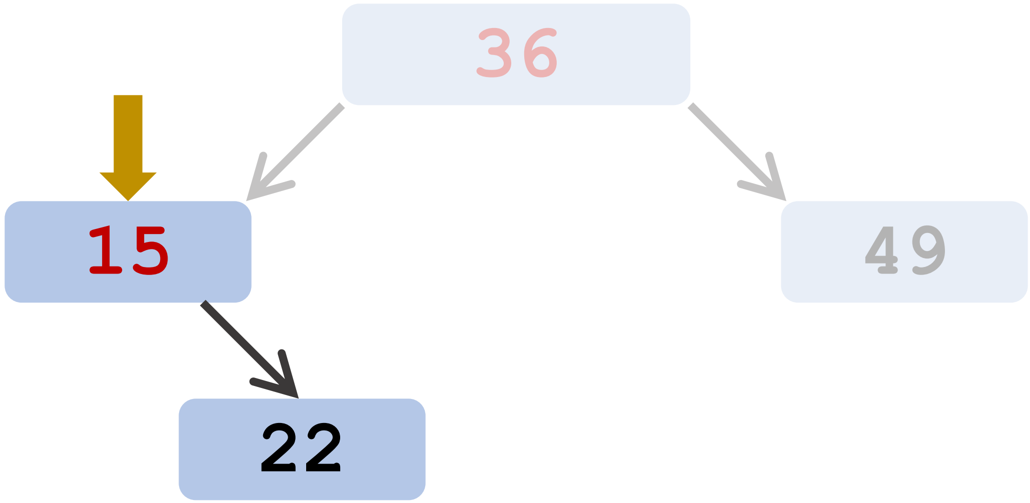


Insert "key=31"



1. Create new node 31
2. key < 36 ➔ Go to left.

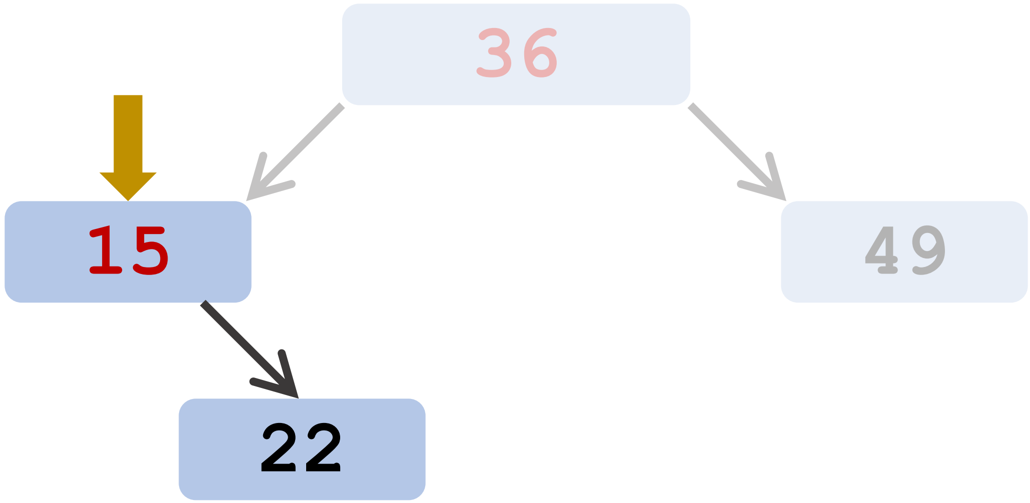
Insert “key=31”



1. Create new node 

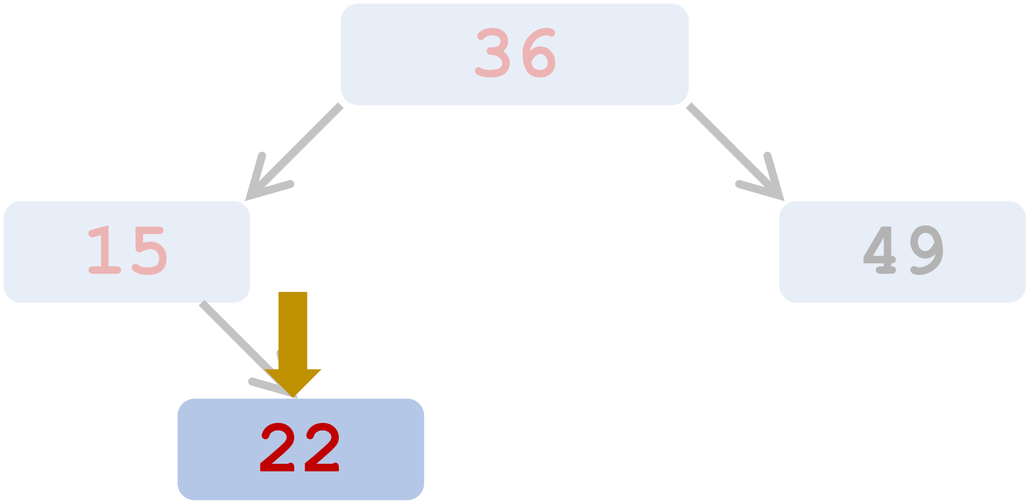
2. key < 36 ➔ Go to left.

Insert "key=31"



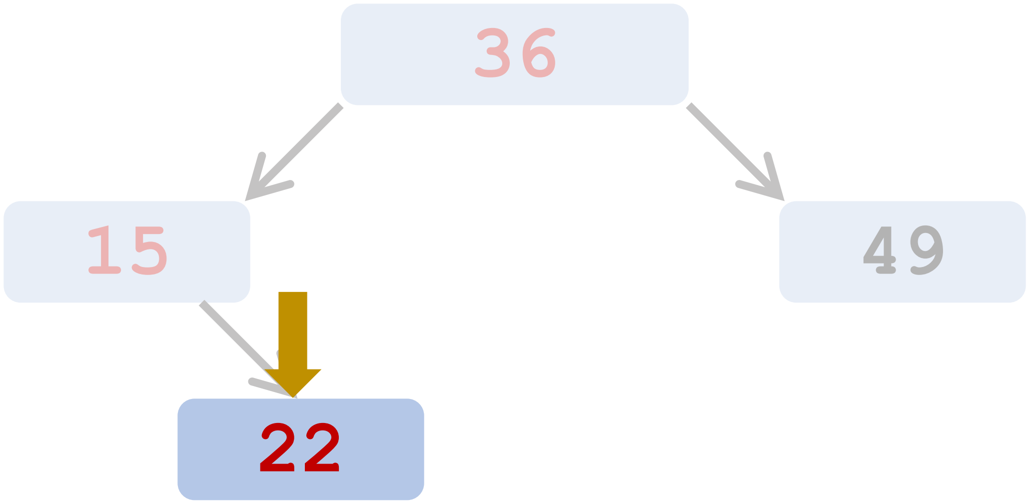
1. Create new node 31
2. key < 36 ➔ Go to left.
3. key > 15 ➔ Go to right.

Insert “key=31”



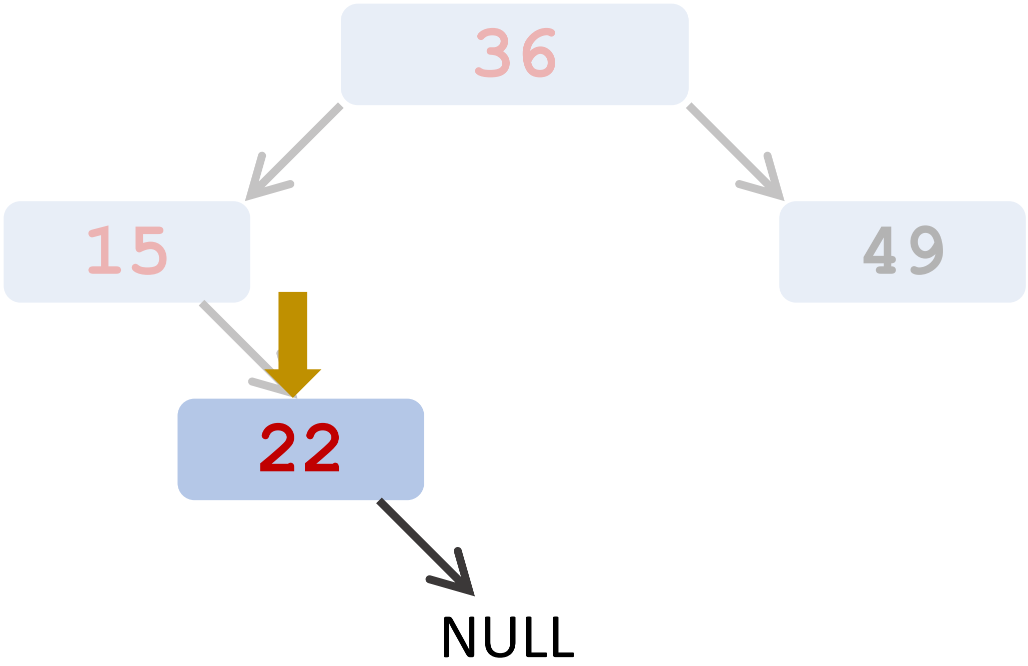
1. Create new node 31
2. key < 36 ➔ Go to left.
3. key > 15 ➔ Go to right.

Insert "key=31"



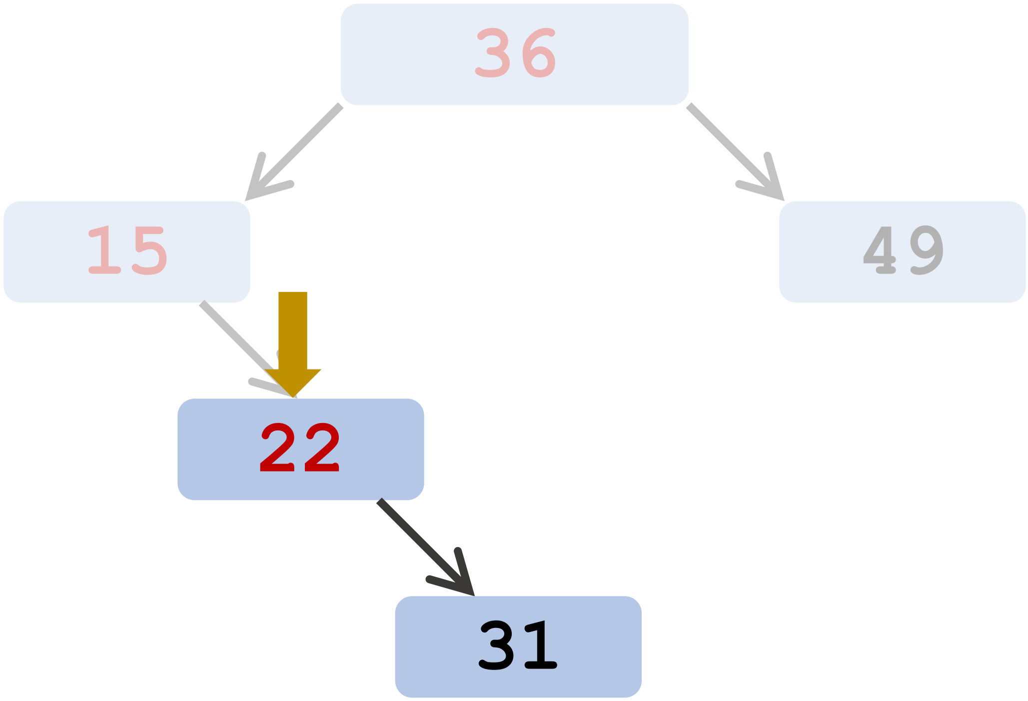
1. Create new node 31
2. key < 36 ➔ Go to left.
3. key > 15 ➔ Go to right.
4. key > 22 ➔ Go to right.

Insert “key=31”



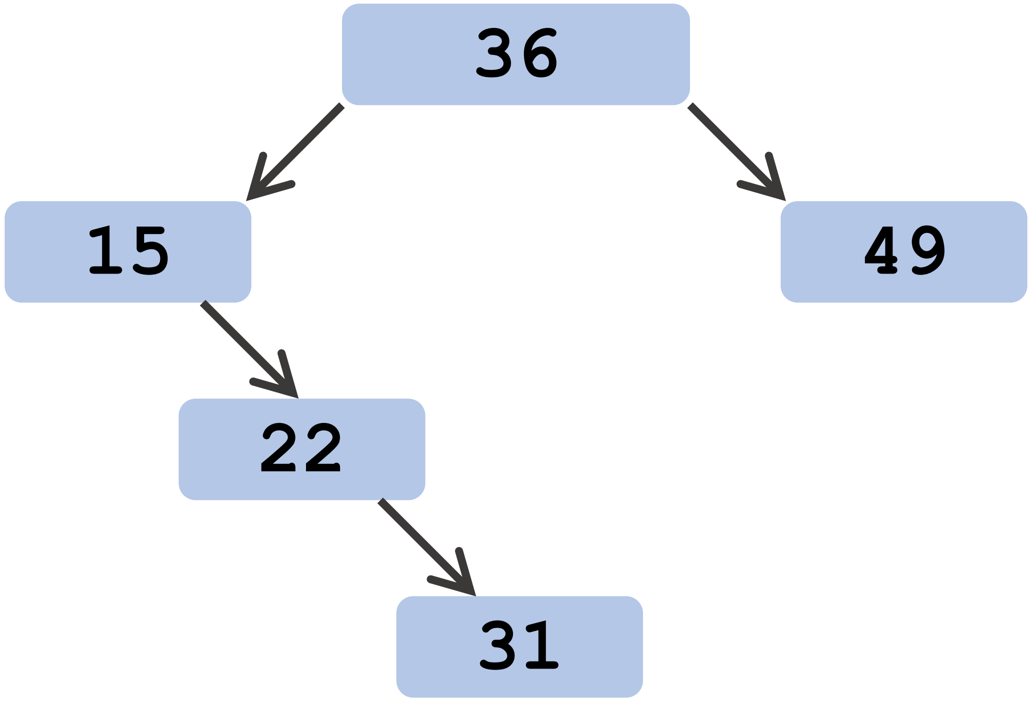
1. Create new node 31
2. key < 36 ➔ Go to left.
3. key > 15 ➔ Go to right.
4. key > 22 ➔ Go to right.
5. Right child is NULL

Insert “key=31”

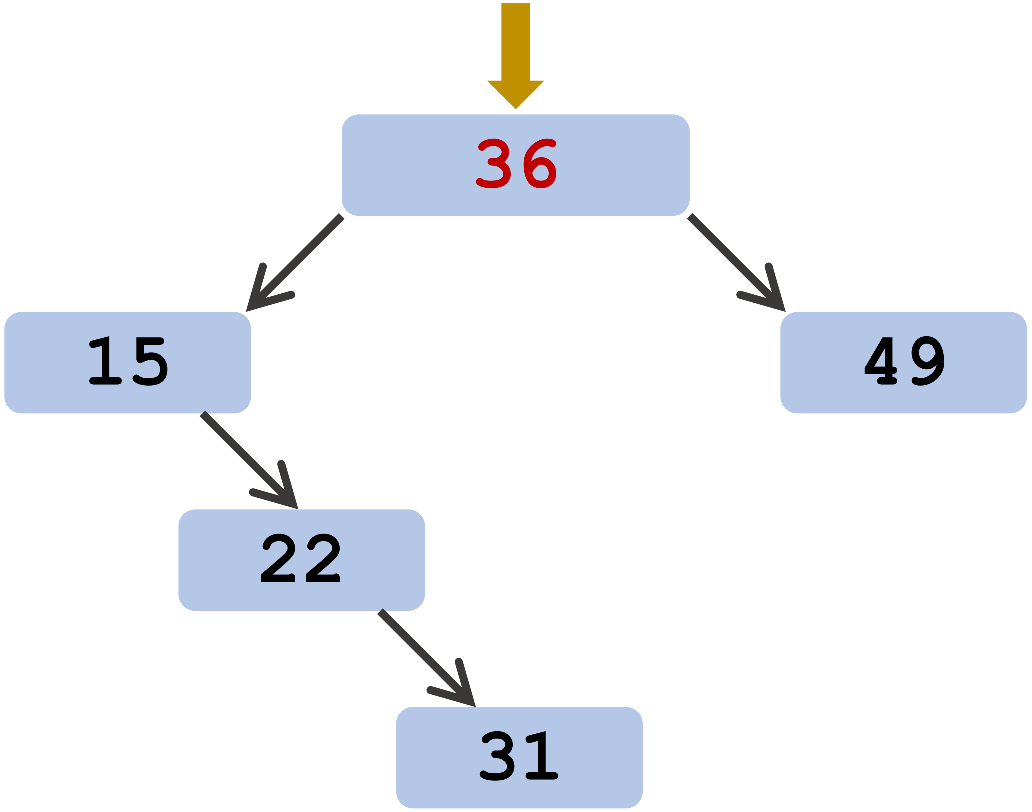


1. Create new node 31
2. key < 36 ➔ Go to left.
3. key > 15 ➔ Go to right.
4. key > 22 ➔ Go to right.
5. Right child is NULL ➔ Make “31” the right child.

Insert “key=6”

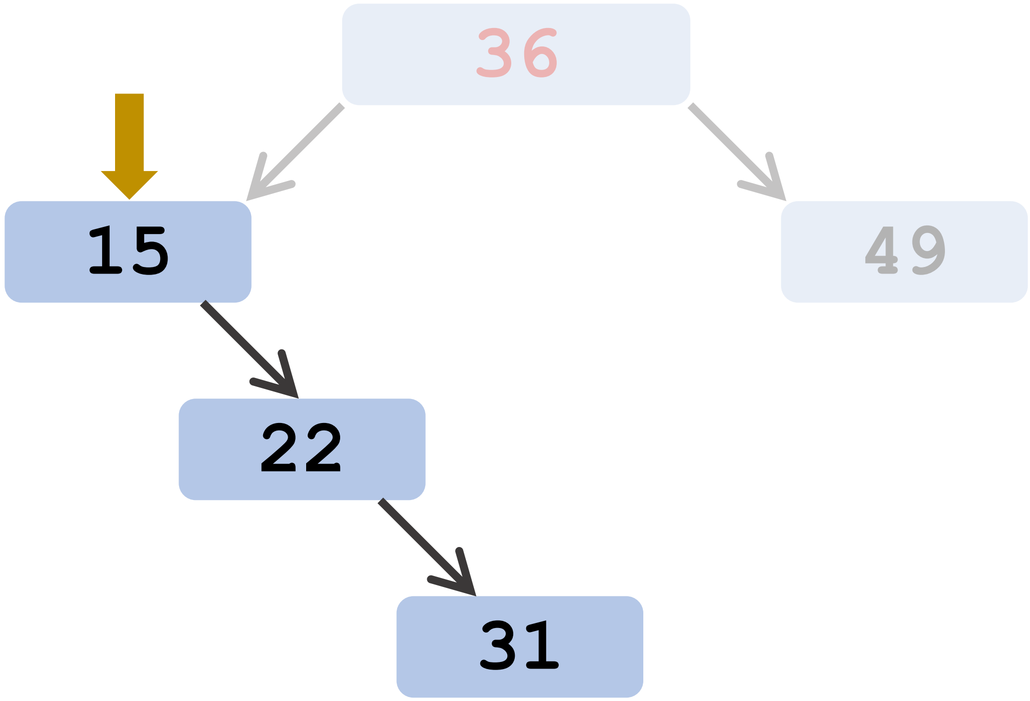


Insert “key=6”



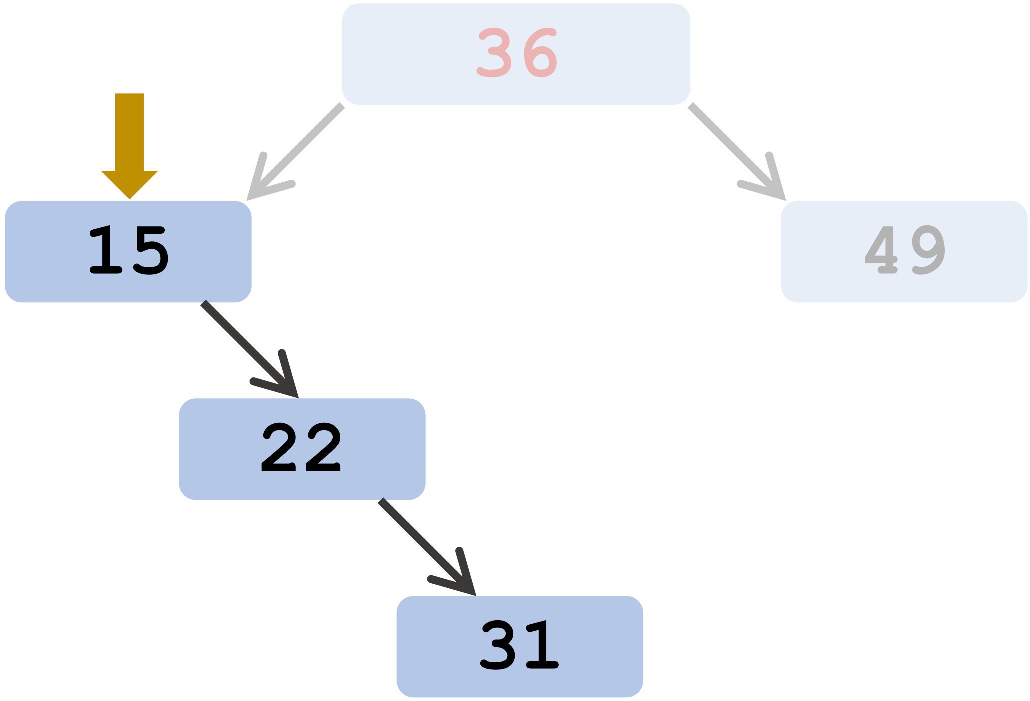
1. Create new node 6
2. key < 36 ➔ Go to left.

Insert "key=6"



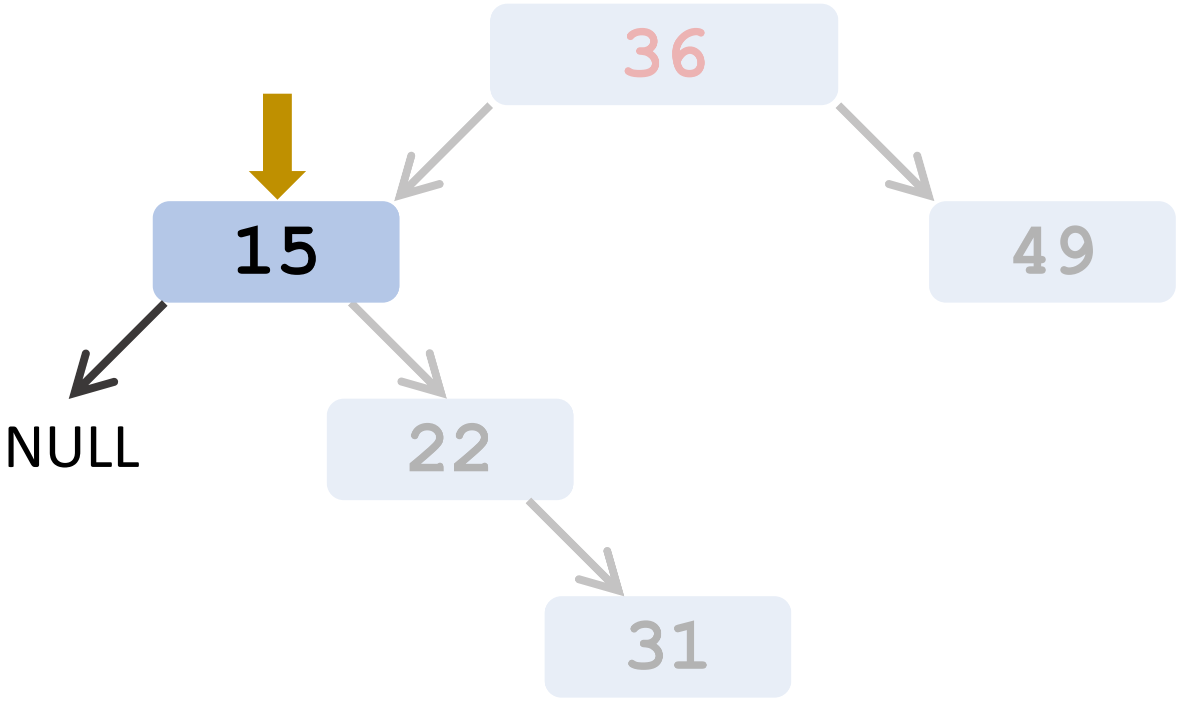
1. Create new node 6
2. key < 36 ➔ Go to left.

Insert “key=6”



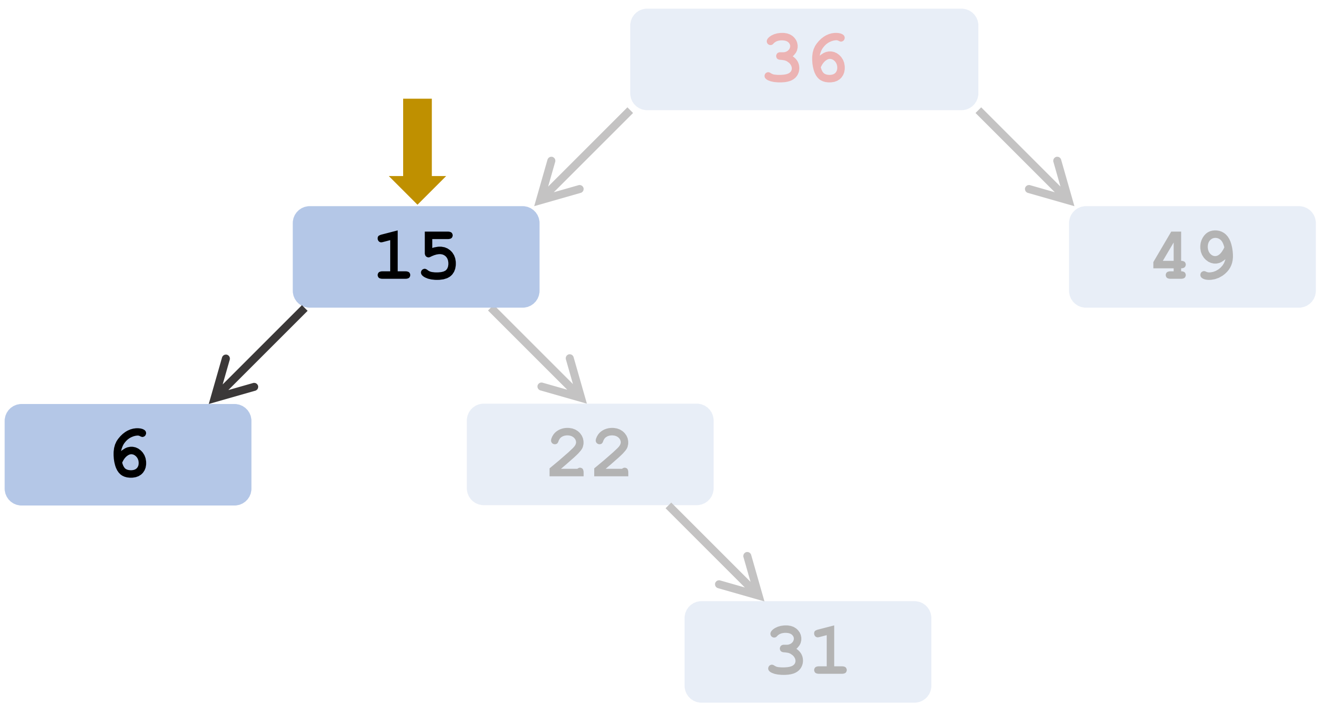
1. Create new node 6
2. key < 36 ➔ Go to left.
3. key < 15 ➔ Go to left.

Insert “key=6”



1. Create new node 6
2. key < 36 ➔ Go to left.
3. key < 15 ➔ Go to left.
4. Left child is NULL

Insert “key=6”



1. Create new node 6
2. key < 36 ➔ Go to left.
3. key < 15 ➔ Go to left.
4. Left child is NULL ➔
   Make “6” the left child.

```c
struct vertex* insert(struct vertex* root, int key) {
    if (root == NULL) { // the tree is empty
        struct vertex* r = newNode(key);
        return r; // new root
    }
    else { // recur down the tree
        if (key < root->key)
            root->left  = insert(root->left, key);
        else if (key > root->key)
            root->right = insert(root->right, key);
        return root; // the root is unchanged
    }
}
```

```
struct vertex* insert(struct vertex* root, int key) {
    if (root == NULL) { // the tree is empty
        struct vertex* r = newNode(key);
        return r; // new root
    }
    else { // recur down the tree
        if (key < root->key)
            root->left  = insert(root->left, key);
        else if (key > root->key)
            root->right = insert(root->right, key);
        return root; // the root is unchanged
    }
}
```

Questions

Inorder Traversal

This is what we learned in this class.

```
void inorder(struct vertex *root) {
    if (root != NULL) {
        inorder(root->left);
        cout << root->key << endl;
        inorder(root->right);
    }
}
```

Inorder Traversal

Tree:

```
        50
      /    \
    30      70
   /  \       \
 20    40      80
```

Inorder print:

```
20
30
40
50
70
80
```

Preorder Traversal

```
void preorder(struct vertex *root) {
    if (root != NULL) {
        cout << root->key << endl;
        preorder(root->left);
        preorder(root->right);
    }
}
```

Preorder Traversal

Tree:

```
        50
      /    \
    30      70
   /  \       \
 20    40      80
```

Preorder print:

50

30

20

40

70

80

Postorder Traversal

```
void postorder(struct vertex *root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        cout << root->key << endl;
    }
}
```

Postorder Traversal

Tree:

```
        50
      /     \
    30        70
   /  \         \
 20    40        80
```

Postorder print:

20

40

30

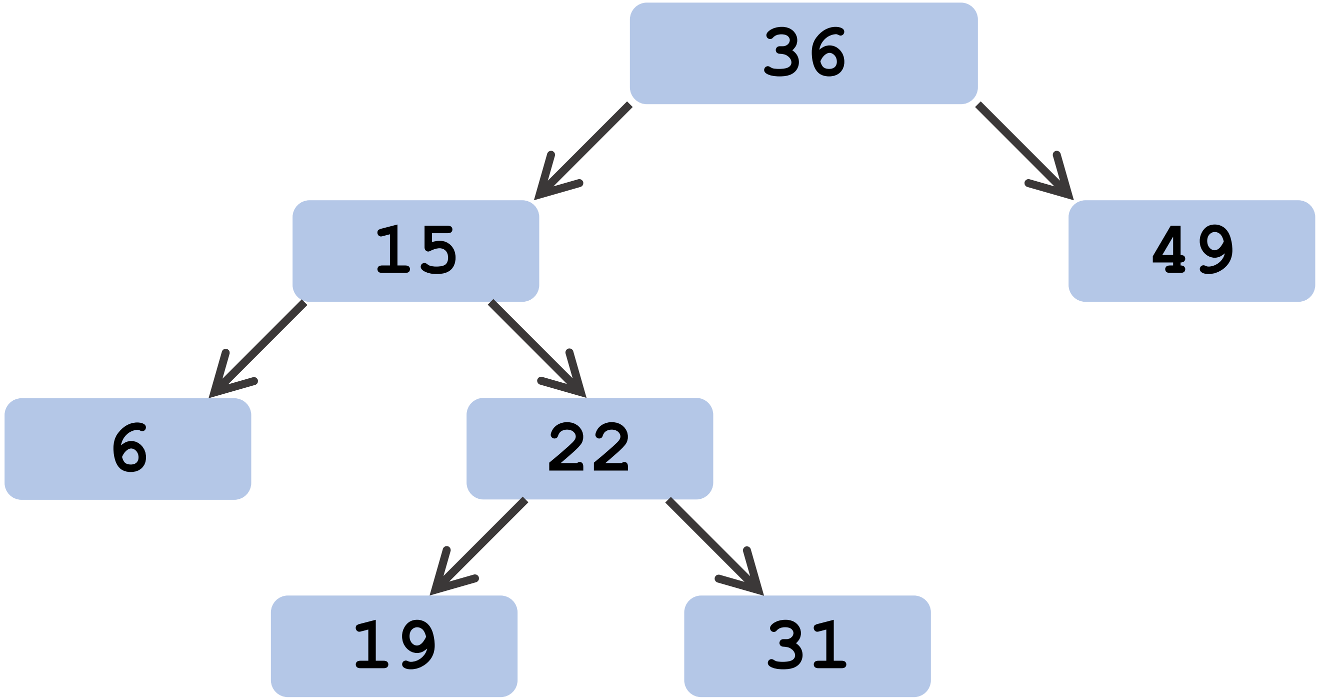
80

70

50

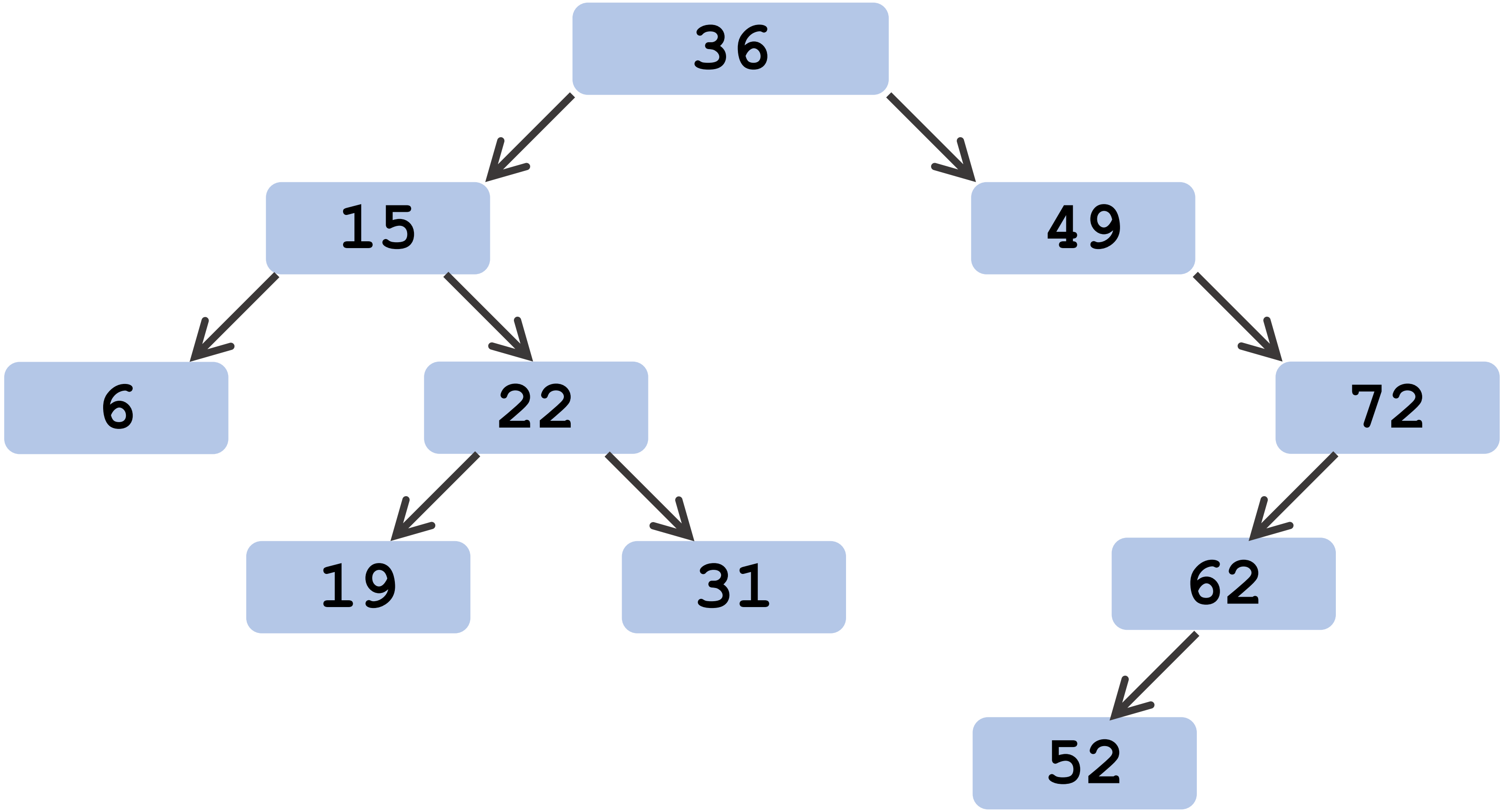
Question 1: Traversal

What are the results of inorder, preorder, and postorder print?



Question 2: Traversal

What are the results of inorder, preorder, and postorder print?



Question 3: Insertion

- Initially, the binary search tree is empty.
- The following keys are inserted sequentially:

  19, 89, 64, 8, 9, 6, 4, 66, 76.

- Draw the tree after all the insertions.

Thank You!