Recursive Binary Contain

We did this in binary search tree;

In binary search tree, we did this through itteratively method;

Now, we are trying to solve this recursively.

In [ ]:
```
def __r_contains(self, current_node, value):

    # With the other contains method we only passed it a value ; we dindot need to pass it a node


    # Thats why we have double underscore in front of the function name; This implies ; "we dont want end user to c

    # So for that we are going to have another method; called r_contains, that doesnot have double underscore


def r_contains(self, value):
    # With this we only need to pass it the value; not the node ;

    # THen after from where we will call __doubble underscore function for recursive contains with the root value

    return self.__r_contains(self.root,value)
```

🔒 `__r_contains(self, current_node, value)` – ***Private Recursive Helper***

- The **double underscore (`__`)** prefix is a naming convention in Python indicating that a method is **intended to be private** and **should not be accessed directly** by the user of the class.
- It performs the actual recursive logic, which requires both the current node and the value to search.
- This method is "internal" – it's part of how the class works, but not part of the interface you expect other programmers to use.

🔓 `r_contains(self, value)` – ***Public Interface***

- This method is part of the **public API** of the class.
- It simplifies usage by hiding the recursive details from the user.
- The user only needs to pass in the `value`, and this method **initializes** the recursion by calling `__r_contains` with the root node.

🔍 Recursive Contains Method: Naming Convention in Python

This code demonstrates a common Python convention for recursive methods using **public and private naming** to separate interface from implementation.

🧩 Code Snippet (Overview)

```python
def __r_contains(self, current_node, value):
    # Private recursive method that takes the current node
    # Not intended to be called directly by the user
    pass

def r_contains(self, value):
    # Public method that initializes the recursion
    # Calls the private method starting from the root node
    return self.__r_contains(self.root, value)
```

🔒 `__r_contains(self, current_node, value)` – *Private Recursive Helper*

- The **double underscore (__)** prefix is a Python convention to indicate the method is **private**.
- Handles the actual **recursive logic** for searching.
- Requires both the current node and the value.
- Hidden from external use — not part of the class's public interface.

🔓 `r_contains(self, value)` – *Public Interface Method*

- This is the **user-facing method**.
- Accepts only the `value` to search for — easier and cleaner to use.
- Internally calls the private method:
  `self.__r_contains(self.root, value)`
- Abstracts away the recursion details from the user.

✅ Benefits of This Pattern

Benefit	Description

Benefit	Description
Encapsulation	Keeps internal logic hidden and protected from accidental misuse
Ease of Use	User only needs to provide the value, not deal with tree nodes
Clean Structure	Separates the interface (public method) from the implementation (private)

☕ Analogy

> `r_contains(value)` is like pressing a **button on a coffee machine**.
> `__r_contains(node, value)` is the **inner mechanism** that grinds, heats, and brews — all hidden behind the scenes.

In [1]:
```
def __r_contains(self, current_node, value):
    if current_node == None:
        return False
    if value == current_node.value:
        return True
    if value < current_node.value:
        return __r_contains(current_node.left, value)
    if value > current_node.value:
        return __r_contains(current_node.right, value)


def r_contains(self, value):

    return self.__r_contains(self.root, value)
```

In [17]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self):
        self.root = None
```

```python
    def insert(self, value):
        new_node = Node(value)
        if self.root == None:
            self.root = new_node
            return True
        temp = self.root
        while (True):
            if new_node.value < temp.value:
                if temp.left == None:
                    temp.left = new_node
                    return True
                temp = temp.left
            else:
                if temp.right == None:
                    temp.right = new_node
                    return True
                temp = temp.right

    def __r_contains(self, current_node, value):
        if current_node == None:
            return False
        if value == current_node.value:
            return True
        if value < current_node.value:
            return self.__r_contains(current_node.left, value)
        if value > current_node.value:
            return self.__r_contains(current_node.right, value)


    def r_contains(self, value):
        return self.__r_contains(self.root, value)

my_Tree = BinaryTree()
my_Tree.insert(47)
my_Tree.insert(57)
my_Tree.insert(37)
my_Tree.insert(67)
my_Tree.insert(77)

print(my_Tree.r_contains(67))
```

```
print(my_Tree.r_contains(57))

print(my_Tree.r_contains(87))
```

```
True
True
False
```

Now recursive Insert method

In [20]:

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

class BinaryTree:
    def __init__(self):
        self.root = None

    def __r_insert(self, current_node,value):
        new_node = Node(value)
        if current_node == None:
            current_node = new_node
        if value < current_node.value:
            return self.__r_insert(current_node.left, value)
        if value > current_node.value:
            return self.__r_insert(current_node.right, value)

    def r_insert(self,value):
        self.__r_insert(self.root,value)

    def __r_contains(self, current_node, value):
        if current_node == None:
            return False
        if value == current_node.value:
            return True
        if value < current_node.value:
            return self.__r_contains(current_node.left, value)
```

```
        if value > current_node.value:
            return self.__r_contains(current_node.right, value)


    def r_contains(self, value):
        return self.__r_contains(self.root, value)

my_Tree = BinaryTree()
my_Tree.r_insert(47)
my_Tree.r_insert(57)
my_Tree.r_insert(37)
my_Tree.r_insert(67)
my_Tree.r_insert(77)

print(my_Tree.r_contains(67))

print(my_Tree.r_contains(57))

print(my_Tree.r_contains(87))
```

```
False
False
False
```

🧠 Python Naming Conventions: `_`, `__`, and Method Design

✅ `r_insert(self, value)` – Public Method

- **No leading underscores** means it's part of the **public interface**.
- This is the method intended to be used by the end user, like:
  `tree.r_insert(42)`

🔒 `__r_insert(self, current_node, value)` – Private Helper Method

- **Double leading underscores (`__`)** trigger **name mangling**.
- Python internally renames `__r_insert to _ClassName__r_insert`, e.g., `_BinaryTree__r_insert`.

- It's used to prevent accidental access and override in subclasses.
- Meant to signal: "**this is internal, don't touch it outside the class**."

💡 Why use both?

This is a **design pattern** to:

1. Make the API user-friendly (`r_insert(value)`)
2. Encapsulate the recursive logic internally (`__r_insert(current_node, value)`)

🔹 Summary Table

Name	Convention	Intended Use
`r_insert`	Public	Called by users of the class
`_r_insert`	Protected (soft)	Internal use, but can still be accessed
`__r_insert`	Private (strict)	Strongly internal – Python mangles the name

---

🔁 Why `r_insert` Has No `return`, but `r_contains` Does

📌 `r_insert(value)` — *Modifies the Tree (No Return)*

- Its **purpose is to change the tree** by inserting a new node.
- It doesn't need to return anything to the caller — it's a **side-effect** operation.
- You call it like:
```
tree.r_insert(42)  # no need to store a result
```

✅ `r_contains(value)` — *Checks if Value Exists (Returns True/False)*

- Its **purpose is to return a value** — a `bool`: whether or not the value exists in the tree.
- This is a **query operation**, not a modification.
- You use the result:

```
if tree.r_contains(42):
    print("Found!")
```

🔍 Summary

Method	Purpose	Returns?	Why?
`r_insert`	Insert a node (modify)	❌ No	Only modifies internal structure
`r_contains`	Check if value exists	✅ Yes	Returns True/False

==

🤔 Why do we need `current_node.left = ...`?

✅ Because `__r_insert()` returns an updated subtree (Node)

When we insert a value recursively, we might be creating a new `Node` or modifying a subtree. That change needs to be **attached** to the current node's `.left` or `.right` pointer.

Example:

```
current_node.left = self.__r_insert(current_node.left, value)
```

Here's what this line means:

- You're asking: "Insert this value into the **left subtree**."
- `__r_insert()` will return the **new or unchanged subtree**, and

- You assign it back to `current_node.left` — ensuring the tree structure is preserved.

❌ What if we don’t do the assignment?

If you write:

```
self.__r_insert(current_node.left, value)
```

Then:

- The recursive call still creates or updates nodes.
- But you **throw away the result** — so `current_node.left` stays unchanged.
- This breaks the tree! Nodes get created, but they're not linked to anything.

🔁 Analogy

Think of `current_node.left = ...` like **reconnecting wires** after an upgrade.

You’re replacing the old left child with a new version of it (possibly unchanged, possibly updated with a new node added deeper).

✅ TL;DR:

Code	Effect
`current_node.left = ...`	Correctly attaches the result to the tree structure
`self.__r_insert(current_node.left, value)`	Does the work, but discards the result — tree is broken

In [23]:

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

```python
class BinaryTree:
    def __init__(self):
        self.root = None

    def __r_insert(self, current_node, value):
        if current_node is None:
            return Node(value)
        if value < current_node.value:
            current_node.left = self.__r_insert(current_node.left, value)
        elif value > current_node.value:
            current_node.right = self.__r_insert(current_node.right, value)
        return current_node

    def r_insert(self, value):
        if self.root is None:
            self.root = Node(value)
        self.root = self.__r_insert(self.root, value)

    def __r_contains(self, current_node, value):
        if current_node == None:
            return False
        if value == current_node.value:
            return True
        if value < current_node.value:
            return self.__r_contains(current_node.left, value)
        if value > current_node.value:
            return self.__r_contains(current_node.right, value)


    def r_contains(self, value):
        return self.__r_contains(self.root, value)

my_Tree = BinaryTree()
my_Tree.r_insert(47)
my_Tree.r_insert(57)
my_Tree.r_insert(37)
my_Tree.r_insert(67)
my_Tree.r_insert(77)

print(my_Tree.r_contains(67))
```

```
print(my_Tree.r_contains(57))

print(my_Tree.r_contains(87))
```

```
True
True
False
```