```
# With Binary Trees each node has a maximum of 2 children.
# A 2-3-4 Tree is a Tree that can contain more then 2 children.
# All non-leaf nodes have 1 more child than pieces of data
# The 2-3-4 refers to :
# 1. A Node with 1 piece of Data -> 2 Children
# 2. A Node with 2 pieces of Data -> 3 Children
# 3. A Node with 3 pieces of Data -> 4 Children

# Empty nodes are not allowed
# Each node can contain 3 pieces of data
# Each nodes values are positioned in ascending order
# All child node values on the left of a node are less than the parent
# All child node values on the right of a node are greater than the parent
# Duplicate values aren't allowed
# Leaves are all on the bottom

class Data:

    def __init__(self, value):
        self.value = value

    def get_data_value(self):
        print(f"{self.value} ")


class Node:
    def __init__(self):
        self.num_values = 0
        self.parent = None
        self.child_list = []  # Holds Node Children
        self.value_list = []  # List of Values
        # Initialize Lists
        for j in range(4):
            self.child_list.append(None)
        for k in range(3):
            self.value_list.append(None)

    # Connect the child to the node
    def connect_child(self, child_num, child):
        self.child_list[child_num] = child
        # If not null it is parent
        if child:
            child.parent = self

    # Disconnect and return child
    def disconnect_child(self, child_num):
        # Store child for returning and delete by setting to null
        temp = self.child_list[child_num]
        self.child_list[child_num] = None
        return temp

    # Check for child list to find if it is a leaf
    def is_leaf(self):
        return not self.child_list[0]
```

```
# Can't contain more than 3 values
def is_full(self):
    return self.num_values == 3

# Cycle through 3 possible values looking for a match
def find_item(self, key):
    for j in range(3):
        # If not found return -1 else return the value
        if not self.value_list[j]:
            break
        elif self.value_list[j].value == key:
            return j
    return -1

# Slide
def insert_item(self, new_item):
    # Assume node isn't full and increment
    self.num_values += 1
    # Create new item key
    new_key = new_item.value

    # Cycle through values starting on the right
    for j in reversed(range(3)):
        # If a null value go left
        if self.value_list[j] is None:
            pass
        # If not null
        else:
            # Get the other key
            other_key = self.value_list[j].value
            # If the new key is smaller
            if new_key < other_key:
                # Shift to right
                self.value_list[j + 1] = self.value_list[j]
            else:
                # Otherwise insert it
                self.value_list[j + 1] = new_item
                # Return index to new value
                return j + 1
    # Insert new value
    self.value_list[0] = new_item
    return 0

def remove_item(self):
    # Assume node isn't empty and save value
    temp = self.value_list[self.num_values - 1]
    # Remove by setting to null and decrement
    self.value_list[self.num_values - 1] = None
    self.num_values -= 1  # one less item
    return temp

def display_node(self):
    for j in range(self.num_values):
```

```
        self.value_list[j].get_data_value()


class Tree234:
    def __init__(self):
        self.root = Node()  # root node

    def find(self, key):
        # Start searching at root
        curr_node = self.root
        while True:
            # Cycle through the values in the node looking for it
            child_number = curr_node.find_item(key)
            # If found return it
            if child_number != -1:
                return child_number
            # If it is a leaf we can't search in a child below
            # so it isn't here
            elif curr_node.is_leaf():
                return -1
            # Search in the child node below
            else:
                curr_node = self.get_next_child(curr_node, key)

    def split(self, the_node):
        # Assume the node is full
        node_2 = the_node.remove_item()  # remove items from
        node_3 = the_node.remove_item()  # this node
        child_2 = the_node.disconnect_child(2)  # remove children
        child_3 = the_node.disconnect_child(3)  # from this node

        # Make new right node
        new_right = Node()

        # If Root make new root and assign as parent
        if the_node == self.root:
            self.root = Node()
            parent = self.root
            self.root.connect_child(0, the_node)
        # Otherwise node isn't root
        else:
            # So get the parent
            parent = the_node.parent

        # Insert the node in the parent
        item_index = parent.insert_item(node_3)
        # Get number of values in parent
        n = parent.num_values

        # Move parents connected nodes one child to right
        j = n - 1
        while j > item_index:
            temp = parent.disconnect_child(j)
            parent.connect_child(j + 1, temp)
```

```
        j -= 1
    # Connect new right node to the parent
    parent.connect_child(item_index + 1, new_right)

    # Insert node in new right and connect children
    new_right.insert_item(node_2)
    new_right.connect_child(0, child_2)
    new_right.connect_child(1, child_3)

# New data is always inserted in leaves
# Slide If 15 is inserted well look for the proper position
# in a leaf, shift larger numbers right if needed and insert
def insert(self, value):
    # Start at root
    curr_node = self.root
    # Create a Data object
    temp = Data(value)

    while True:
        # Check if there is room for the data
        if curr_node.is_full():
            # If no room we need to split up values
            self.split(curr_node)
            # Save the parent
            curr_node = curr_node.parent
            # Get the next child
            curr_node = self.get_next_child(curr_node, value)
        # If node is a leaf break out of loop and insert with
        # the last line
        elif curr_node.is_leaf():
            break
        # If node is not full and not a leaf go to lower level
        else:
            curr_node = self.get_next_child(curr_node, value)
    curr_node.insert_item(temp)  # Insert new item

# Gets correct node based on the value
def get_next_child(self, node, value):
    # Get number of values in the node
    num_values = node.num_values

    # Search in each value of the node
    for j in range(num_values):

        # If less return left child
        if value < node.value_list[j].value:
            return node.child_list[j]
    else:  # If greater return the right
        return node.child_list[j+1]

def print_tree(self):
    self.rec_print_tree(self.root, 0, 0)

def rec_print_tree(self, the_node, level, child_number):
```

```python
        print('Row :', level, 'Child :', child_number)
        # Print Node data
        the_node.display_node()

        # Recursively call this function for each child of this node
        num_values = the_node.num_values
        for j in range(num_values + 1):
            next_node = the_node.child_list[j]
            if next_node:
                self.rec_print_tree(next_node, level + 1, j)
            else:
                return


tree = Tree234()
tree.insert(23)
tree.insert(55)
tree.insert(11)
tree.insert(42)
tree.insert(74)
tree.insert(5)
tree.insert(9)
tree.insert(13)
tree.insert(23)
tree.insert(30)
tree.insert(44)
tree.insert(47)
tree.insert(63)
tree.insert(67)
tree.insert(72)
tree.print_tree()
if tree.find(67) != -1:
    print("Found Value")
else:
    print("Not Found")
```