```
# 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)
```