In [3]:
```python
class DisjointSet:
    """
    Disjoint Set (Union-Find) with:
    - MAKE-SET: Θ(1)
    - FIND-SET: amortized O(alpha(n))
    - UNION: amortized O(alpha(n))
    """
    def __init__(self):
        self.parent = {}  # parent[x] = parent of x
        self.rank = {}    # rank[x] = aproximate depth of tree rooted at x

    def MAKE_SET(self, x):
        """Θ(1): Create a set containing only x"""
        # x is its own parent (it's the root)
        self.parent[x] = x
        # rank starts at 0
        self.rank[x] = 0

    def FIND_SET(self, x):
        """O(α(n)): Find which set x belongs to"""
        # If x is not the root, recursivly find the root
        if self.parent[x] != x:
            # PATH COMPRESSION: make x point directly to the root
            self.parent[x] = self.FIND_SET(self.parent[x])
        return self.parent[x]

    def UNION(self, x, y):
        """O(α(n)): Union the sets containing x and y"""
        # Find the roots of both sets
        root_x = self.FIND_SET(x)
        root_y = self.FIND_SET(y)

        # If they're already in the same set, do nothing
        if root_x == root_y:
            return

        # UNION BY RANK: attach smaller tree under bigger tree
        if self.rank[root_x] < self.rank[root_y]:
            # x's tree is smaller, make y the parent
```

```python
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            # y's tree is smaller, make x the parent
            self.parent[root_y] = root_x
        else:
            # Same rank, pick one as parent and increse its rank
            self.parent[root_y] = root_x
            self.rank[root_x] += 1


# Example usage:
ds = DisjointSet()

# Make some sets
ds.MAKE_SET(1)
ds.MAKE_SET(2)
ds.MAKE_SET(3)
ds.MAKE_SET(4)

print(" FIND_SET(1): ",ds.FIND_SET(1))
print(" FIND_SET(2): ",ds.FIND_SET(2))
print(" FIND_SET(3): ",ds.FIND_SET(3))
print(" FIND_SET(4): ",ds.FIND_SET(4))

# Union some sets
print(" Now running UNION(1, 2)")
ds.UNION(1, 2)  # {1,2} {3} {4}
print(" Now running UNION(3, 4)")
ds.UNION(3, 4)  # {1,2} {3,4}

# Find which set elements belong to
print(" FIND_SET(1) after UNION(1, 2): ", ds.FIND_SET(1))  # Should be same as FIND_SET(2)
print(" FIND_SET(2) after UNION(1, 2): ", ds.FIND_SET(2))
print(" FIND_SET(3) after UNION(3, 4): ", ds.FIND_SET(3))  # Should be same as FIND_SET(4)
print(" FIND_SET(4) after UNION(3, 4): ", ds.FIND_SET(4))
print(" Now running UNION(2, 3)")
ds.UNION(2, 3)  # {1,2,3,4}
print(" FIND_SET(1) after UNION(2, 3): ", ds.FIND_SET(1))  # All should return same root now
print(" FIND_SET(2) after UNION(2, 3): ", ds.FIND_SET(2))
```

```python
print(" FIND_SET(3) after UNION(2, 3): ", ds.FIND_SET(3))
print(" FIND_SET(4) after UNION(2, 3): ", ds.FIND_SET(4))
```

```
FIND_SET(1):  1
FIND_SET(2):  2
FIND_SET(3):  3
FIND_SET(4):  4
Now running UNION(1, 2)
Now running UNION(3, 4)
FIND_SET(1) after UNION(1, 2):  1
FIND_SET(2) after UNION(1, 2):  1
FIND_SET(3) after UNION(3, 4):  3
FIND_SET(4) after UNION(3, 4):  3
Now running UNION(2, 3)
FIND_SET(1) after UNION(2, 3):  1
FIND_SET(2) after UNION(2, 3):  1
FIND_SET(3) after UNION(2, 3):  1
FIND_SET(4) after UNION(2, 3):  1
```

Explanations:

Since we flatten (Path Compression) the tree by connecting all point to root, the average cost becomes small.

Flatting operation however cost something O(1). it need to create the first point. but after that, we are doing x operations n times with a total O(xa(n)) and average being O(a(n)) with is small or at least grows very very slowly.

When merging (Union by Rank), we allways attached the smaller ree under the the larger tree, the cost of ransk is almost as the hight of the tree. witch gives: Hight <= log n with is O(log n)

the total runtime complexity is: MAKE: TETA(n) FIND AND UNION: O(xa(n)) or O(x+n)) since a(n) is allways less than 5 and almost constant