# CSC263 Tutorial #10 Exercises

Disjoint Sets with Efficient PRINT Operation

**Alexander He Meng**

Prepared for March 28th, 2025

# CSC263H5 – Tutorial 11: Disjoint Sets with Efficient PRINT Operation

## Problem Statement

The goal is to augment the disjoint set data structure to efficiently implement the following operation:

**PRINT(x):** Print all elements in $S_x$ (the set containing $x$).

The target is to achieve $O(|S_x|)$ worst-case runtime for the `PRINT` operation, without affecting the usual $O(\alpha(n))$ amortized time for `MAKE-SET`, `FIND-SET`, and `UNION`.

## Data Structure Augmentation

Each set will be maintained as both:

- A tree for the standard disjoint set operations (`parent`, `rank`).
- A linked list that connects all elements of the set.

Each node will store:

$$\text{parent}[x],\ \text{rank}[x],\ \text{next}[x]$$

Each root node will additionally store:

$$\text{tail}[x]$$

The linked list will always start from the root of the set and will visit all elements exactly once through the `next` pointers.

## MAKE-SET(x)

`MAKE-SET` initializes the singleton set containing only $x$ and creates a trivial linked list where $x$ points to nothing.

```
def MAKE-SET(x):
    parent[x] = x
    rank[x] = 0
    next[x] = NIL    # x has no next node
    tail[x] = x      # tail of the list is itself
```

## FIND-SET(x)

The FIND-SET operation is the same as the classical disjoint set operation with path compression.

```
def FIND-SET(x):
    if parent[x] != x:
        parent[x] = FIND-SET(parent[x])
    return parent[x]
```

## UNION(x, y)

UNION merges both the disjoint set trees and the linked lists. The smaller ranked tree is attached under the larger ranked tree, and the corresponding linked lists are merged by connecting the tail of one list to the head of the other.

```
def UNION(w, z):
    x = FIND-SET(w)
    y = FIND-SET(z)
    if x == y:
        return

    if rank[x] < rank[y]:
        parent[x] = y
        tail[y].next = x
        tail[y] = tail[x]
    else:
        parent[y] = x
        tail[x].next = y
        tail[x] = tail[y]
        if rank[x] == rank[y]:
            rank[x] += 1
```

## PRINT(x)

`PRINT` outputs all elements in the set containing $x$ by traversing the linked list starting at the root.

```
def PRINT(x):
    r = FIND-SET(x)
    u = r
    while u != NIL:
        print(u)
        u = next[u]
```

## Explanation of Operations

- **MAKE-SET**: Creates a singleton set where the linked list contains only $x$.
- **FIND-SET**: Finds the representative of the set containing $x$ using path compression.
- **UNION**: Merges both the trees and their linked lists efficiently. Only one pointer adjustment is needed to connect the two linked lists, and the `tail` is updated.
- **PRINT**: Prints all members of the set in $O(|S_x|)$ time by simply traversing the linked list.

## Runtime Analysis

- **MAKE-SET**, **FIND-SET**, **UNION**: Each runs in $O(\alpha(n))$ amortized time using union by rank and path compression.
- **PRINT**: Runs in $O(|S_x|)$ time since it visits each element in the set exactly once.

## Correctness

The linked list always contains all elements of the set. Every union merges two lists correctly by appending one to the other, and no elements are lost or duplicated. The set structure used for `FIND-SET` and `UNION` remains valid and efficient due to the unchanged use of union by rank and path compression. The `PRINT` operation traverses the linked list directly and outputs the exact elements in the set.