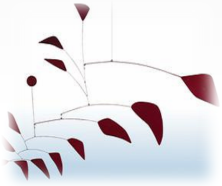
EECS 4101-5101
Advanced Data Structures



Shahin Kamali

Topic 5: Disjoint Sets

York University

Picture is from the cover of the textbook CLRS.

Objectives

- By the end of this module, you will be able to:
  - Explain the Disjoint Set abstract data type and its operations (queries).

Objectives

- By the end of this module, you will be able to:
  - Explain the Disjoint Set abstract data type and its operations (queries).
  - Recognize the application of Disjoint Sets as “black boxes" in algorithms like Kruskal’s minimum spanning tree algorithm, and use disjoint sets as black boxes for other practical algorithms.

Objectives

- By the end of this module, you will be able to:
  - Explain the Disjoint Set abstract data type and its operations (queries).
  - Recognize the application of Disjoint Sets as “black boxes" in algorithms like Kruskal's minimum spanning tree algorithm, and use disjoint sets as black boxes for other practical algorithms.
  - Describe various data structures for Disjoing Sets and compare and contrast their running times.

Objectives

- By the end of this module, you will be able to:
  - Explain the Disjoint Set abstract data type and its operations (queries).
  - Recognize the application of Disjoint Sets as “black boxes" in algorithms like Kruskal’s minimum spanning tree algorithm, and use disjoint sets as black boxes for other practical algorithms.
  - Describe various data structures for Disjoing Sets and compare and contrast their running times.
  - Describe the standard union-find data structure for disjoint sets using union-by-rank and path compression.

Disjoint Sets

- Disjoint set is an abstract data type for maintaining a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint, non-empty sets.
  - **Disjoint:** there is no common element between any two sets (if a is in S_i it cannot be in S_j where $i \neq j$).
  - **Dynamic:** sets can be modified by **make-set** and **union** operations
  - Each set is identified by a **representative element** of the set.

$$k = 4; \quad S_a = \{\underline{a}, b, m, n\}, S_c = \{\underline{c}, g, h\}, S_e = \{d, \underline{e}, f\}, S_q = \{\underline{q}\}$$

Disjoint Sets Operations

- **makeSet(x)**:
  - Create a new set $\{x\}$ whose only element is x.
  - By property 1 above, x cannot be an element of any other set.
  - By default, x is the representative of the new set.

$$k = 4; \quad S_a = \{\underline{a}, b, m, n\}, S_c = \{\underline{c}, g, h\}, S_e = \{d, \underline{e}, f\}, S_q = \{\underline{q}\}$$

Disjoint Sets Operations

- **makeSet(x)**:
  - Create a new set $\{x\}$ whose only element is x.
  - By property 1 above, x cannot be an element of any other set.
  - By default, x is the representative of the new set.

E.g., **makeSet($\{p\}$)**

$$k = 4; \quad S_a = \{\underline{a}, b, m, n\}, S_c = \{\underline{c}, g, h\}, S_e = \{d, \underline{e}, f\}, S_q = \{\underline{q}\}$$

$$S_p = \{\underline{p}\}$$

Disjoint Sets Operations

- **find(x)** (also called Find-Set(x)):
  - Return the representative element of the set containing x.

$$k = 4; \quad S_a = \{\underline{a}, b, m, n\}, S_c = \{\underline{c}, g, h\}, S_e = \{d, \underline{e}, f\}, S_q = \{\underline{q}\},$$

Disjoint Sets Operations

- **find(x)** (also called Find-Set(x)):
  - Return the representative element of the set containing x.

E.g., **find(b)** $\rightarrow a$

$$k = 4; \quad S_a = \{\underline{a}, b, m, n\}, S_c = \{\underline{c}, g, h\}, S_e = \{d, \underline{e}, f\}, S_q = \{\underline{q}\},$$

Disjoint Sets Operations

- **find(x)** (also called Find-Set(x)):
  - Return the representative element of the set containing x.

E.g., **find(b)** $\rightarrow a$
E.g., **find(c)** $\rightarrow c$

$k = 4; \quad S_a = \{\underline{a}, b, m, n\}, S_c = \{\underline{c}, g, h\}, S_e = \{d, \underline{e}, f\}, S_q = \{\underline{q}\},$

Disjoint Sets Operations

- **union(x, y)**:
  - Unite the sets containing x and y.
  - Suppose set S_x contains x and set S_y contains y.
  - $S \leftarrow S \cup \{S_x \cup S_y\} - S_x - S_y$
  - Assign a representative for $x \cup y$.
  - $union(x, y)$ is equivalent to $union(find(x), find(y))$.

$$k = 4; \quad S_a = \{\underline{a}, b, m, n\}, S_c = \{\underline{c}, g, h\}, S_e = \{d, \underline{e}, f\}, S_q = \{\underline{q}\},$$

Disjoint Sets Operations

- **union**(x, y):
  - Unite the sets containing x and y.
  - Suppose set S_x contains x and set S_y contains y.
  - $S \leftarrow S \cup \{S_x \cup S_y\} - S_x - S_y$
  - Assign a representative for $x \cup y$.
  - $union(x, y)$ is equivalent to $union(find(x), find(y))$.

E.g., Union$(b, d) \rightarrow$ merge S_a and S_e.

$$k = 4; \quad S_a = \{\underline{a}, b, m, n\}, S_c = \{\underline{c}, g, h\}, S_e = \{d, \underline{e}, f\}, S_q = \{\underline{q}\},$$

$$\rightarrow \quad S_c = \{\underline{c}, g, h\}, S_q = \{\underline{q}\}, S_a = \{\underline{a}, b, m, n, d, e, f\}$$

Disjoint Sets Operations

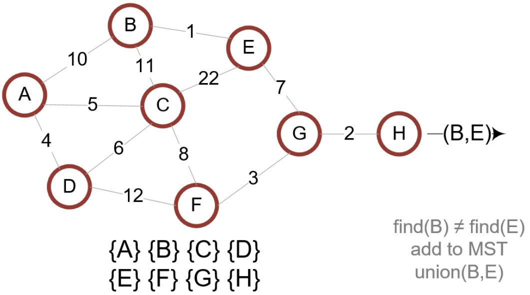
- **makeSet(x)**:
  - Create a new set $\{x\}$ whose only element is x.
  - By default, x is the representative of the new set.
- **find(x)** (also called Find-Set(x)):
  - Return the representative element of the set containing x.
- **union(x, y)**:
  - Unite the sets containing x and y.
  - Assign a representative for $x \cup y$.
  - $union(x, y)$ is equivalent to $union(find(x), find(y))$.

Applications of Disjoint Sets

- Many applications in designing algorithms
- E.g., Kruskal's minimum spanning tree for a graph with n vertices and m edges.

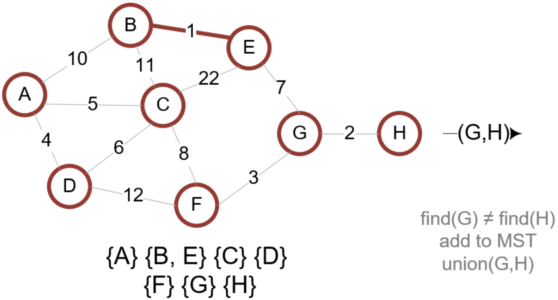
Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge e does not form a cycle in MST, add it to MST.



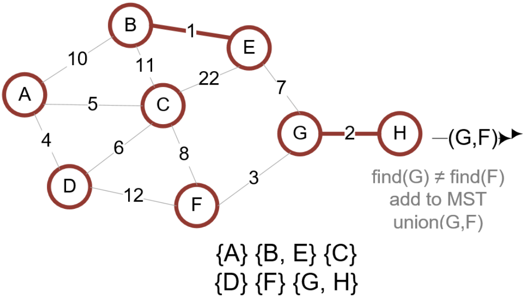
Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge e does not form a cycle in MST, add it to MST.



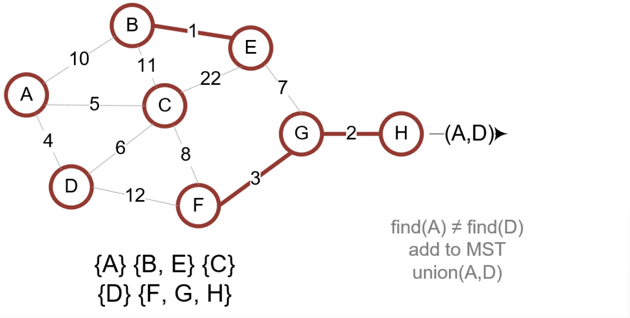
Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



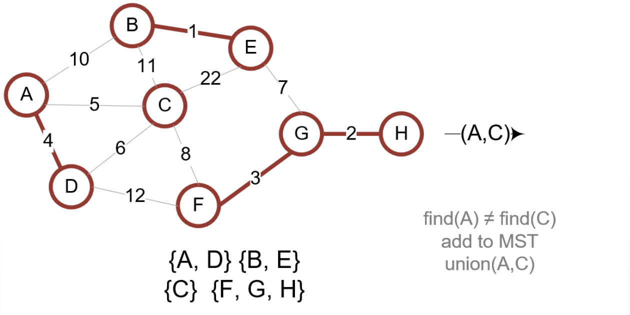
Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



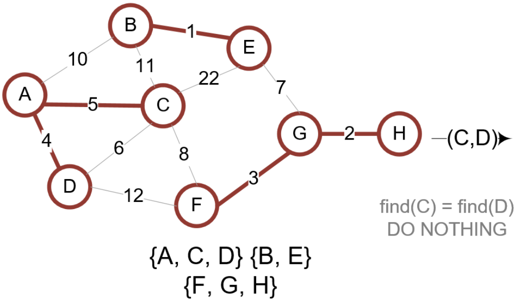
Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge *e* does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - *e* does not form a cycle iff its endpoints are in different components



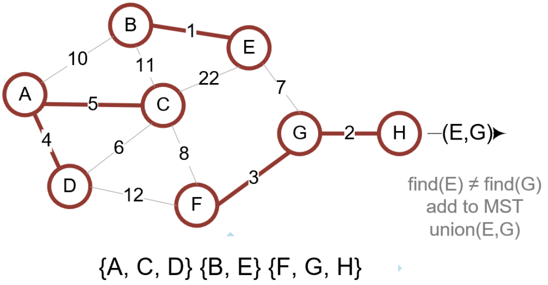
Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



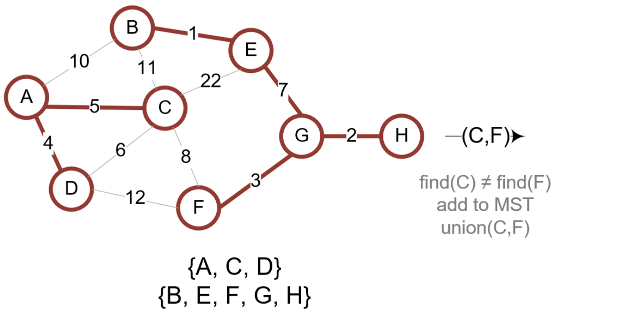
Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



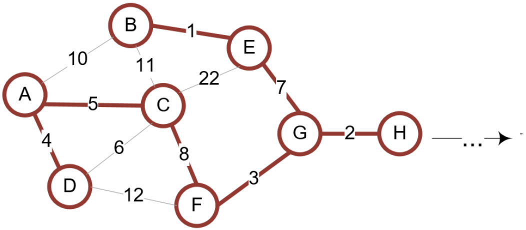
Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge e does not form a cycle in MST, add it to MST.
    - Maintain MST's connected component as disjoint sets of vertices
    - e does not form a cycle iff its endpoints are in different components



Kruskal's MST algorithm

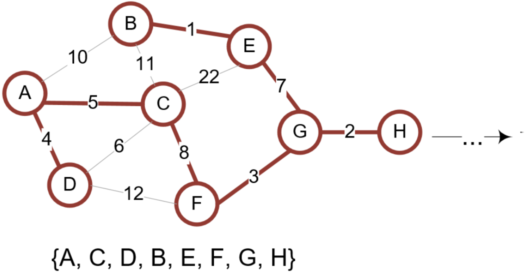
- Sort edges by their weights and process them one by one.
- If an edge *e* does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - *e* does not form a cycle iff its endpoints are in different components



{A, C, D, B, E, F, G, H}

Kruskal's MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components
  - The running time is $O(m \log m + mx)$, where $O(x)$ is the amortized running time of merge and find operations.



Disjoint Sets Review

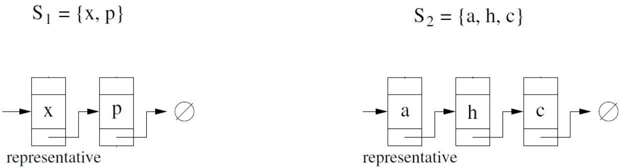
- **Disjoint set** is an abstract data type for maintaining a set of dosjoint sets
  - make-set(x): create a new set with a single item x (which is not in any of the existing sets).
  - find(x): returns the representative item of the set that includes x.
  - union(x,y): removes the sets in which x and y belong to and adds a new set which is the union of deleted sets

Disjoint Sets Review

- **Disjoint set** is an abstract data type for maintaining a set of dosjoint sets
  - make-set(x): create a new set with a single item x (which is not in any of the existing sets).
  - find(x): returns the representative item of the set that includes x.
  - union(x,y): removes the sets in which x and y belong to and adds a new set which is the union of deleted sets
- Disjoint sets have many applications in design of algorithms (e.g., Kruskal's MST algorithm)

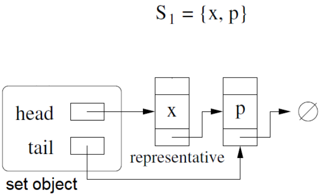
Data Structures for Disjoint Sets

- Linked lists for disjoint sets:
  - Each set is stored as a linked-list.
  - The representative element is the first element in the list.



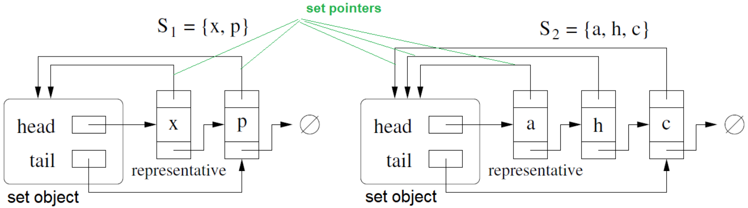
Data Structures for Disjoint Sets

- Linked lists for disjoint sets:
  - Each set is stored as a linked-list.
  - The representative element is the first element in the list.
  - In a 'set object', store head/tail pointers to the first/last elements.



Data Structures for Disjoint Sets

- Linked lists for disjoint sets:
  - Each set is stored as a linked-list.
  - The representative element is the first element in the list.
  - In a ‘set object’, store head/tail pointers to the first/last elements.
  - Each node stores a **set pointer** to the set object.



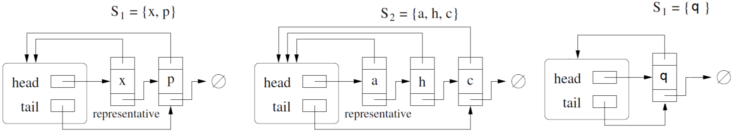
Linked lists for disjoint sets

- makeSet(x):
  - Create a list containing one node.
  - takes $O(1)$
  - O(1) time

Linked lists for disjoint sets

- makeSet(x):
  - Create a list containing one node.
  - takes $O(1)$
  - O(1) time

makeSet(q)



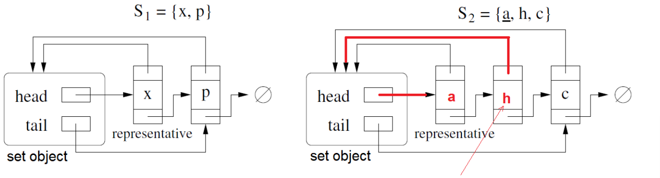
Linked lists for disjoint sets

- find(x):
    - follow the set-pointer to find the set object and get the representative element.

Linked lists for disjoint sets

- find(x):
  - follow the set-pointer to find the set object and get the representative element.

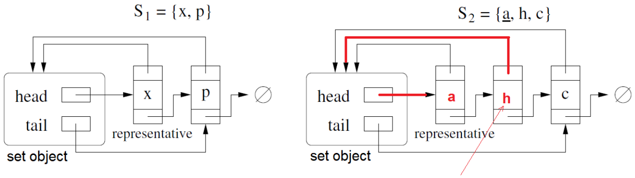
find(h) $\to$ a



Linked lists for disjoint sets

- find(x):
  - follow the set-pointer to find the set object and get the representative element.
  - We assume we're given a reference to x.
  - It takes O(1) time

find(h) $\rightarrow$ a



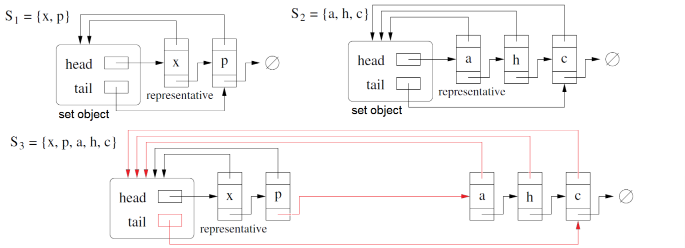
Linked lists for disjoint sets

- union(x,y):
  - Append y's list to the end of x's list.
  - find(x) becomes the representative of the new set.
  - Use head pointer from x's list and tail pointer from y's list.
  - Requires updating the **set pointer** for each node in y's list, i.e., $\Theta(n)$ time per operation in the worst case (when y has size $\Theta(n)$).

Linked lists for disjoint sets

- union(x,y):
  - Append y's list to the end of x's list.
  - find(x) becomes the representative of the new set.
  - Use head pointer from x's list and tail pointer from y's list.
  - Requires updating the **set pointer** for each node in y's list, i.e., $\Theta(n)$ time per operation in the worst case (when y has size $\Theta(n)$).

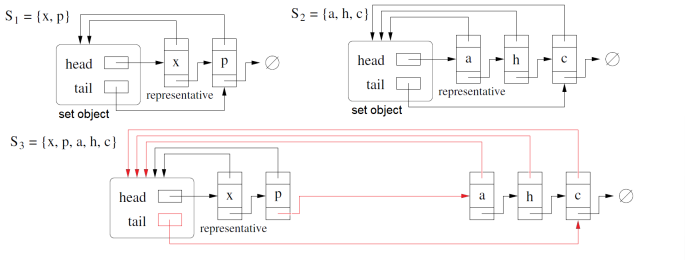
union(p,h)



Linked lists for disjoint sets

- union(x,y):
  - Append y's list to the end of x's list.
  - find(x) becomes the representative of the new set.
  - Use head pointer from x's list and tail pointer from y's list.
  - Requires updating the **set pointer** for each node in y's list, i.e., $\Theta(n)$ time per operation in the worst case (when y has size $\Theta(n)$).
  - What is the **amortized cost** of performing $n-1$ union operations?

union(p,h)



Review of Amortized Analysis

- Amortized analysis considers the average cost per operation for a sequence of m operations.

Review of Amortized Analysis

- Amortized analysis considers the average cost per operation for a sequence of m operations.
- In many data structures, there are many different sequences of operations
  - We often consider the **worst-case amortized time**, i.e., the average cost of an operation for the worst-case sequence
  - Sometimes people look at expected amortized time which considers the average cost for a random sequence (we do not talk about it in this course).

Linked lists for disjoint sets

- What is the amortized cost of performing $n-1$ union operations?
- The following example is a worst-case sequence which provides a lower bound.
  - makeSet(x_i) for $i \in \{1, 2 \ldots, n\}$
  - union(x_i, x_{i-1}) for $i \in \{n, n-1, \ldots 2\}$, that is:
    - union(x_{n-1}, x_n): update 1 set-pointers
    - union(x_{n-2}, x_n): update 2 set-pointers
    - ...
    - union(x_{n-i}, x_n): $\rightarrow$ update i set-pointers
    - ...
    - union(x_1, x_n): updated $n-1$ set-pointers

Linked lists for disjoint sets

- What is the amortized cost of performing $n-1$ union operations?
- The following example is a worst-case sequence which provides a lower bound.
  - makeSet(x_i) for $i \in \{1, 2 \ldots, n\}$
  - union(x_i, x_{i-1}) for $i \in \{n, n-1, \ldots 2\}$, that is:
    - union(x_{n-1}, x_n): update 1 set-pointers
    - union(x_{n-2}, x_n): update 2 set-pointers
    - ...
    - union(x_{n-i}, x_n): $\rightarrow$ update i set-pointers
    - ...
    - union(x_1, x_n): updated $n-1$ set-pointers
- Total set-pointer updates: $1 + 2 + 3 + \ldots + n - 1 \in \Omega(n^2)$.
  - Amortized cost of the update operation is $\Omega(n)$ in the worst case.

Linked lists for disjoint sets

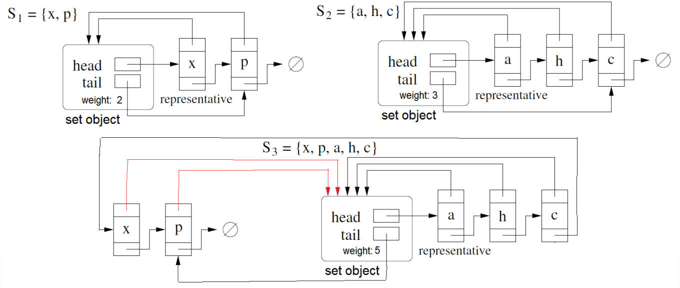
- What is the amortized cost of performing $n-1$ union operations?
- The following example is a worst-case sequence which provides a lower bound.
  - makeSet(x_i) for $i \in \{1, 2 \ldots, n\}$
  - union(x_i, x_{i-1}) for $i \in \{n, n-1, \ldots 2\}$, that is:
    - union(x_{n-1}, x_n): update 1 set-pointers
    - union(x_{n-2}, x_n): update 2 set-pointers
    - ...
    - union(x_{n-i}, x_n): $\rightarrow$ update i set-pointers
    - ...
    - union(x_1, x_n): updated $n-1$ set-pointers
- Total set-pointer updates: $1 + 2 + 3 + \ldots + n - 1 \in \Omega(n^2)$.
  - Amortized cost of the update operation is $\Omega(n)$ in the worst case.
  - This is a worst-case amortized time; there are sequences formed m unions for which the amortized cost is constant.

Linked lists for disjoint sets

- What is the amortized cost of performing $n-1$ union operations?
- The following example is a worst-case sequence which provides a lower bound.
  - makeSet(x_i) for $i \in \{1, 2 \ldots, n\}$
  - union(x_i, x_{i-1}) for $i \in \{n, n-1, \ldots 2\}$, that is:
    - union(x_{n-1}, x_n): update 1 set-pointers
    - union(x_{n-2}, x_n): update 2 set-pointers
    - ...
    - union(x_{n-i}, x_n): $\rightarrow$ update i set-pointers
    - ...
    - union(x_1, x_n): updated $n-1$ set-pointers
- Total set-pointer updates: $1 + 2 + 3 + \ldots + n - 1 \in \Omega(n^2)$.
  - Amortized cost of the update operation is $\Omega(n)$ in the worst case.
  - This is a worst-case amortized time; there are sequences formed m unions for which the amortized cost is constant.
- **If we simply append the second list to the end of the first list, the (worst-case) amortized time for union is $\Theta(n)$.**

Linked lists & Union by Weight

- What if we append the smallest list to the end of the larger list?
- In the set object, in addition to head and tail pointers, maintain a **weight** field which indicates the number of items in that list (set).
  - Make-set and find are as before, i.e., they take constant time per operation
  - For union, we compare the weights and append the smaller list to the end of the larger list



Linked lists & Union by Weight

- Consider a single node u of the list. We count the number of times the set-pointer is updated for that node.
- Each time the pointer of u is updated, that means that the set of u is merged with a larger set
  - The weight of the set of u is at least doubled after the merge.
- If there are n items in all sets, the weight of each set is at most n.
  - Each update for set-pointer of u doubles the weight of its list, and this weight cannot be more than n
  - Hence, there are at most $\lceil \log n \rceil$ set-pointer updates per item, i.e., a total of $O(n \log n)$ set-pointer updates in total.

Linked lists & Union by Weight

- There are at most $\lceil \log n \rceil$ set-pointer updates per item, i.e., a total of $O(n \log n)$ set-pointer updates.
- In addition to the cost of set-pointer updates, the cost of each operation for other pointer updates is constants $\rightarrow \Theta(m)$ cost for m operations

Linked lists & Union by Weight

- There are at most $\lceil \log n \rceil$ set-pointer updates per item, i.e., a total of $O(n \log n)$ set-pointer updates.
- In addition to the cost of set-pointer updates, the cost of each operation for other pointer updates is constants $\rightarrow \Theta(m)$ cost for m operations
- Union by Weight has a cost of $O(n \log n + m)$ for a sequence of m operations on a universe of size n
  - Assuming $m \geq n$, the amortized cost per operation is $O(n \log n / m + 1) = O(\log n)$

Linked lists & Union by Weight

- There are at most $\lceil \log n \rceil$ set-pointer updates per item, i.e., a total of $O(n \log n)$ set-pointer updates.
- In addition to the cost of set-pointer updates, the cost of each operation for other pointer updates is constants $\rightarrow \Theta(m)$ cost for m operations
- Union by Weight has a cost of $O(n \log n + m)$ for a sequence of m operations on a universe of size n
  - Assuming $m \geq n$, the amortized cost per operation is $O(n \log n / m + 1) = O(\log n)$
- **Union by weight (appending smaller list to the end of larger one) improves the amortized time complexity from $\Theta(n)$ to $O(\log n)$.**

Disjoint Sets Review

- **Disjoint set** is an abstract data type for maintaining a set of dosjoint sets
  - make-set(x): create a new set with a single item x (which is not in any of the existing sets).
  - find(x): returns the representative item of the set that includes x.
  - union(x,y): removes the sets in which x and y belong to and adds a new set which is the union of deleted sets

Disjoint Sets Review

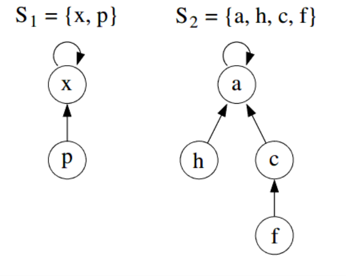
- **Disjoint set** is an abstract data type for maintaining a set of dosjoint sets
  - make-set(x): create a new set with a single item x (which is not in any of the existing sets).
  - find(x): returns the representative item of the set that includes x.
  - union(x,y): removes the sets in which x and y belong to and adds a new set which is the union of deleted sets
- Disjoint sets have many applications in design of algorithms (e.g., Kruskal's MST algorithm)

Disjoint Sets Review

- **Disjoint set** is an abstract data type for maintaining a set of dosjoint sets
  - make-set(x): create a new set with a single item x (which is not in any of the existing sets).
  - find(x): returns the representative item of the set that includes x.
  - union(x,y): removes the sets in which x and y belong to and adds a new set which is the union of deleted sets
- Disjoint sets have many applications in design of algorithms (e.g., Kruskal's MST algorithm)
- Maintaining a list for each set and union-by-weight (appending smaller list to the end of larger one) gives an amortized time of $O(\log n)$ per operation.

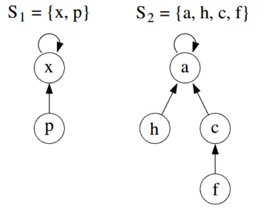
Disjoint Set Forests

- A data structure for disjoint sets which is based on trees instead of lists.
  - Each set is stored as a rooted tree
  - Each node points to its parent
  - The root points to itself
  - The representative element is the root



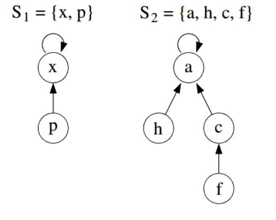
Disjoint Set Forests

- MakeSet(x) takes $O(1)$ time:
  - Create a new tree containing one node x
  - parent(x) $\rightarrow x$



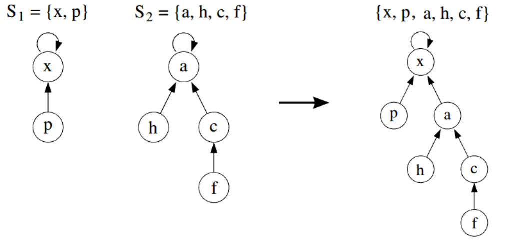
Disjoint Set Forests

- MakeSet(x) takes $O(1)$ time:
    - Create a new tree containing one node x
    - parent(x) $\rightarrow x$
- Find(x):
    - Follow parent pointers to the root and return it.
        - $y \leftarrow x$
        - while $y \neq parent(y)$
        - $\quad y \leftarrow parent(y)$
        - return y
    - Time proportional to the tree's height



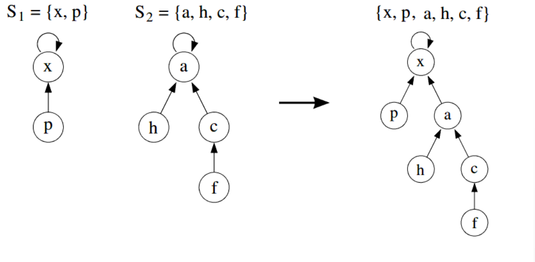
Disjoint Set Forests

- Union(x,y) (first approach):
  - Set root of y's tree to point to the root of x's tree.
    - $\text{root}_x \leftarrow \text{find}(x)$
    - $\text{root}_y \leftarrow \text{find}(y)$
    - $\textit{parent}(\text{root}_y) \leftarrow \text{root}_x$.
  - Time is proportional to tree's height



Disjoint Set Forests

- Union(x,y) (first approach):
  - Set root of y's tree to point to the root of x's tree.
    - $\text{root}_x \leftarrow \text{find}(x)$
    - $\text{root}_y \leftarrow \text{find}(y)$
    - $\mathit{parent}(\text{root}_y) \leftarrow \text{root}_x$.
  - Time is proportional to tree's height
- Tree's height can be $\Theta(n)$ for a universe of size n
  - In the worst case, each operation takes $\Theta(n)$.



Amortized cost of first approach

- What is the amortized cost when performing m operations?

Amortized cost of first approach

- What is the amortized cost when performing m operations?
  - If we simply make the second tree point to the first one, it takes $\Theta(n)$ in the worst case:
  - Consider the following worst-case sequence of operations:
    - make-set(x_i) for $i \in \{1, \ldots, n\}$
    - union(x_i, x_1) for $i \in \{2, \ldots, n\}$.

Amortized cost of first approach

- What is the amortized cost when performing m operations?
  - If we simply make the second tree point to the first one, it takes $\Theta(n)$ in the worst case:
  - Consider the following worst-case sequence of operations:
    - make-set(x_i) for $i \in \{1, \ldots, n\}$
    - union(x_i, x_1) for $i \in \{2, \ldots, n\}$.
  - After the i'th union, set of x_1 is a tree of height i.
  - The total time for the $2n - 1$ operations is $\sum_{i=1}^{n-1} i = n(n-1)/2$, I.e., the amortized cost is $\Theta(n)$.

Amortized cost of first approach

- What is the amortized cost when performing m operations?
  - If we simply make the second tree point to the first one, it takes $\Theta(n)$ in the worst case:
  - Consider the following worst-case sequence of operations:
    - make-set(x_i) for $i \in \{1, \ldots, n\}$
    - union(x_i, x_1) for $i \in \{2, \ldots, n\}$.
  - After the i'th union, set of x_1 is a tree of height i.
  - The total time for the $2n - 1$ operations is $\sum_{i=1}^{n-1} i = n(n-1)/2$, I.e., the amortized cost is $\Theta(n)$.
  - After forming this bad tree, the worst-case sequence of operations continues with $m - 2n + 1$ find(x) operation where x is the only leaf of the tree.

Amortized cost of first approach

- What is the amortized cost when performing m operations?
  - If we simply make the second tree point to the first one, it takes $\Theta(n)$ in the worst case:
  - Consider the following worst-case sequence of operations:
    - make-set(x_i) for $i \in \{1, \ldots, n\}$
    - union(x_i, x_1) for $i \in \{2, \ldots, n\}$.
  - After the i'th union, set of x_1 is a tree of height i.
  - The total time for the $2n-1$ operations is $\sum_{i=1}^{n-1} i = n(n-1)/2$, I.e., the amortized cost is $\Theta(n)$.
  - After forming this bad tree, the worst-case sequence of operations continues with $m - 2n + 1$ find(x) operation where x is the only leaf of the tree.

Observation

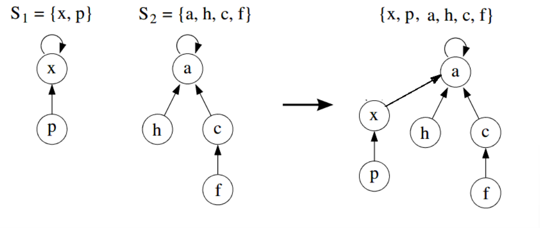
Having the second tree point to the first one for union results in the worst-case trees of height n and amortized time of $\Theta(n)$ for each operation.

Reducing the Height of Trees

- Two strategies for bounding tree heights:
  - union by rank
  - path compression

Union by Rank

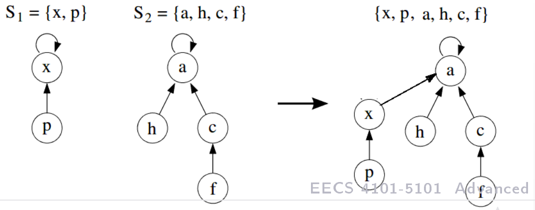
- Attempt to attach the shorter tree to the root of the taller one
  - Similar to union-by-weight on lists
- Maintain the **rank** as an **upper bound** for the height of each tree.
  - The rank increased when both trees have the same rank



Union by Rank

- Attempt to attach the shorter tree to the root of the taller one
  - Similar to union-by-weight on lists
- Maintain the **rank** as an **upper bound** for the height of each tree.
  - The rank increased when both trees have the same rank

  $\text{root}_x \leftarrow \text{find}(x);\ \text{root}_y \leftarrow \text{find}(y)$
  if $\text{rank}(\text{root}_x) > \text{rank}(\text{root}_y)$
      $\textit{parent}(\text{root}_y) \leftarrow \text{root}_x$
  else
      $\textit{parent}(\text{root}_x) \leftarrow \text{root}_y$
  if $\text{rank}(\text{root}_x) = \text{rank}(\text{root}_y)$
      $\text{rank}(\text{root}_y) \leftarrow \text{rank}(\text{root}_y) + 1$



Union by Rank

- If $rank(x) = h$, the tree rooted at x has at least 2^h nodes.

Union by Rank

- If $rank(x) = h$, the tree rooted at x has at least 2^h nodes.
  - Use induction; for the base, we know when $h = 0$, the tree contains $1 = 2^0$ nodes.

Union by Rank

- If $rank(x) = h$, the tree rooted at x has at least 2^h nodes.
  - Use induction; for the base, we know when $h = 0$, the tree contains $1 = 2^0$ nodes.
  - Choose any $h > 0$ and consider the union operation in which the rank is increased from $h - 1$ to h.
  - At the time of union, both trees had rank $h - 1$
  - By induction hypothesis, they each included at least 2^{h-1} nodes.
  - Then the resulting tree has at least $2 \cdot 2^{h-1} = 2^h$ nodes.

Union by Rank

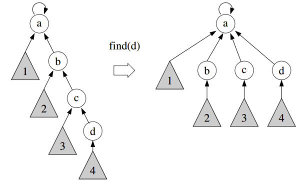
- If $rank(x) = h$, the tree rooted at x has at least 2^h nodes.
  - Use induction; for the base, we know when $h = 0$, the tree contains $1 = 2^0$ nodes.
  - Choose any $h > 0$ and consider the union operation in which the rank is increased from $h - 1$ to h.
  - At the time of union, both trees had rank $h - 1$
  - By induction hypothesis, they each included at least 2^{h-1} nodes.
  - Then the resulting tree has at least $2 \cdot 2^{h-1} = 2^h$ nodes.
  - The number of nodes is **at least** 2^h since after the union, the number of nodes can be increased further.

Union by Rank

- If $rank(x) = h$, the tree rooted at x has at least 2^h nodes.
  - Use induction; for the base, we know when $h = 0$, the tree contains $1 = 2^0$ nodes.
  - Choose any $h > 0$ and consider the union operation in which the rank is increased from $h - 1$ to h.
  - At the time of union, both trees had rank $h - 1$
  - By induction hypothesis, they each included at least 2^{h-1} nodes.
  - Then the resulting tree has at least $2 \cdot 2^{h-1} = 2^h$ nodes.
  - The number of nodes is **at least** 2^h since after the union, the number of nodes can be increased further.
- Since the number of nodes is at least 2^h, the height of the trees is $O(\log n)$
  - Union, find operations when we use union by rank is $O(\log n)$.

Path Compression

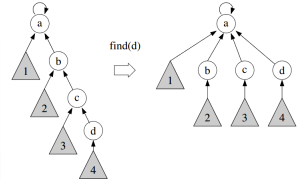
- A simple, effective add on to union by rank
  - Find(x) involves finding a path from x to the root of its tree
  - For each node on the path, update its pointer to point directly to the root.



Path Compression

- A simple, effective add on to union by rank
  - Find(x) involves finding a path from x to the root of its tree
  - For each node on the path, update its pointer to point directly to the root.

    if $x \neq$ parent(x)
        parent(x) $\leftarrow$ find(parent(x))
    return parent(x)

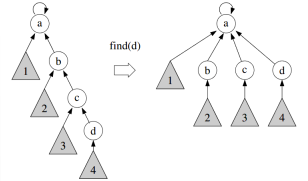


Path Compression

- A simple, effective add on to union by rank
  - Find(x) involves finding a path from x to the root of its tree
  - For each node on the path, update its pointer to point directly to the root.

    if $x \neq \text{parent}(x)$
    $\quad \text{parent}(x) \leftarrow \text{find}(\text{parent}(x))$
    return $\text{parent}(x)$

- For each visited node, the additional work is updating one pointer.

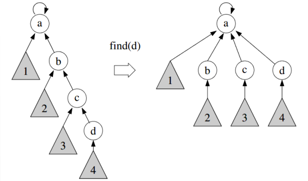


Path Compression

- A simple, effective add on to union by rank
  - Find(x) involves finding a path from x to the root of its tree
  - For each node on the path, update its pointer to point directly to the root.

    if $x \neq \text{parent}(x)$
        $\text{parent}(x) \leftarrow \text{find}(\text{parent}(x))$
    return $\text{parent}(x)$

- For each visited node, the additional work is updating one pointer.
  - Time complexity remains the same asymptotically, i.e., $O(\log n)$.

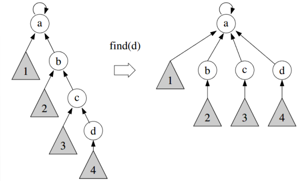


Path Compression

- A simple, effective add on to union by rank
  - Find(x) involves finding a path from x to the root of its tree
  - For each node on the path, update its pointer to point directly to the root.

    if $x \neq$ parent(x)
        parent(x) $\leftarrow$ find(parent(x))
    return parent(x)

- For each visited node, the additional work is updating one pointer.
  - Time complexity remains the same asymptotically, i.e., $O(\log n)$.
- For any y that used to lie on the path from x to the root, any subsequent call to *find*(y) takes O(1) time
  - The amortized time is significantly improved.



Disjoint set data structure

- Maintain a set of disjoint forests
  - Apply union-by rank after union operation (attach the tree with smaller rank to the one with higher rank)
  - Apply path compression after find operation (update the pointer of any node on the Find path to point to the root)
    - Note that the height might change after path compression; hence we use term rank as an upper bound for height

Disjoint set data structure

- Maintain a set of disjoint forests
  - Apply union-by rank after union operation (attach the tree with smaller rank to the one with higher rank)
  - Apply path compression after find operation (update the pointer of any node on the Find path to point to the root)
    - Note that the height might change after path compression; hence we use term rank as an upper bound for height
- The amortized time for performing any operation is $O(\alpha(n))$ where $\alpha(n)$ is a very, very, very slow growing function of n similar to inverse Ackermann function.
  - For any practical reason, $\alpha(n) \leq 4$.
  - In practice (not in theory) you can support disjoint operations in constant time.

$\alpha(n)$ Description

- Let $f^{(i)}(n)$ denote $f(n)$ iteratively applied i times to the initial value of n.

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases}$$

$\alpha(n)$ Description

- Let $f^{(i)}(n)$ denote $f(n)$ iteratively applied i times to the initial value of n.

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases}$$

  - E.g., if $f(n) = 2n$, then

$$f^{(0)}(n) = n = 2^0 n,$$
$$f^{(1)}(n) = f(f^{(0)}(n)) = 2(n) = 2^1 n,$$
$$f^{(2)}(n) = f(f^{(1)}(n)) = 2(2^1 n) = 2^2 n,$$
$$\ldots$$
$$f^{(i)}(n) = f(f^{(i-1)}(n)) = 2(2^{i-1} n) = 2^i n,$$

$\alpha(n)$ Description

- Let $f^{(i)}(n)$ denote $f(n)$ iteratively applied i times to the initial value of n.

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases}$$

  - E.g., if $f(n) = 2n$, then

    $f^{(0)}(n) = n = 2^0 n,$
    $f^{(1)}(n) = f(f^{(0)}(n)) = 2(n) = 2^1 n,$
    $f^{(2)}(n) = f(f^{(1)}(n)) = 2(2^1 n) = 2^2 n,$
    ...
    $f^{(i)}(n) = f(f^{(i-1)}(n)) = 2(2^{i-1} n) = 2^i n,$
  - E.g., if $f(n) = 2^n$, then

    $f^{(0)}(n) = n$
    $f^{(1)}(n) = f(f^{(0)}(n)) = f(n) = 2^n$
    $f^{(2)}(n) = f(f^{(1)}(n)) = f(2^n) = 2^{2^n}$
    ...
    $f^i(n) = f(f^{(i-1)}(n)) = 2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}} \}i \text{ times}$

$\alpha(n)$ Description (cntd.)

- For any $k \geq 0$ and $j \geq 1$, let

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k > 0 \end{cases}$$

$\alpha(n)$ Description (cntd.)

- For any $k \geq 0$ and $j \geq 1$, let

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k > 0 \end{cases}$$

- Function $A_k(j)$ is strictly increasing in both j and k
  - For $j > 0$, $A_1(j) = 2j + 1$.
  - For $j > 0$, $A_2(j) = 2^{j+1}(j+1) - 1$.
  - $A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2^{11} - 1 = 2047$
  - $A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) >> A_2(2047) = 2^{2048}(2048) - 1 > 2^{2048} >> 10^{80}$
  - $A_4(1)$ is by far larger than the number of atoms in the universe.

$\alpha(n)$ Description (cntd.)

- $\alpha(n)$ is the inverse of $A_k(n)$: $\alpha(n) = min\{k | A_k(1) \geq n\}$
  - $\alpha(n)$ is the lowest value of k for which $A_k(1)$ is at least n

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2047 \\ 4 & \text{for } 2048 \leq n \leq A_4(1) \end{cases}$$

$\alpha(n)$ Description (cntd.)

- $\alpha(n)$ is the inverse of $A_k(n)$: $\alpha(n) = min\{k | A_k(1) \geq n\}$
  - $\alpha(n)$ is the lowest value of k for which $A_k(1)$ is at least n

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2047 \\ 4 & \text{for } 2048 \leq n \leq A_4(1) \end{cases}$$

  - For any practical purpose, $\alpha(n) \leq 4$.
  - Theoretically, however, $\alpha(n) \in \omega(1)$, i.e., for every constant c, there is a very huge n such that $\alpha(n) \geq c$.

$\alpha(n)$ Description (cntd.)

- $\alpha(n)$ is the inverse of $A_k(n)$: $\alpha(n) = min\{k | A_k(1) \geq n\}$
  - $\alpha(n)$ is the lowest value of k for which $A_k(1)$ is at least n

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2047 \\ 4 & \text{for } 2048 \leq n \leq A_4(1) \end{cases}$$

  - For any practical purpose, $\alpha(n) \leq 4$.
  - Theoretically, however, $\alpha(n) \in \omega(1)$, i.e., for every constant c, there is a very huge n such that $\alpha(n) \geq c$.
- Recall that the worst-case amortized time for performing an operation (make-set, union, find) is $\alpha(n)$.
  - This bound is tight, i.e., we cannot do better than $\alpha(n)$.

$\alpha(n)$ Description (cntd.)

- $\alpha(n)$ is the inverse of $A_k(n)$: $\alpha(n) = min\{k|A_k(1) \geq n\}$
  - $\alpha(n)$ is the lowest value of k for which $A_k(1)$ is at least n

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2047 \\ 4 & \text{for } 2048 \leq n \leq A_4(1) \end{cases}$$

  - For any practical purpose, $\alpha(n) \leq 4$.
  - Theoretically, however, $\alpha(n) \in \omega(1)$, i.e., for every constant c, there is a very huge n such that $\alpha(n) \geq c$.
- Recall that the worst-case amortized time for performing an operation (make-set, union, find) is $\alpha(n)$.
  - This bound is tight, i.e., we cannot do better than $\alpha(n)$.
- $\alpha(n)$ is the smallest super-constant function that appears in algorithm analysis (there are smaller ones like $\alpha(\alpha(n))$ which don't appear in analysis of practical algorithms).

Disjoint Set Summary

- Disjoint sets maintain a set of disjoint sets with support of make-set(x), find(x), and union(x,y).

Disjoint Set Summary

- Disjoint sets maintain a set of disjoint sets with support of make-set(x), find(x), and union(x,y).
- The right data structure for disjoint sets is a forest of trees (one tree per set).
  - In case of a union, apply union by rank
  - In case of a find, apply path compression

Disjoint Set Summary

- Disjoint sets maintain a set of disjoint sets with support of make-set(x), find(x), and union(x,y).
- The right data structure for disjoint sets is a forest of trees (one tree per set).
  - In case of a union, apply union by rank
  - In case of a find, apply path compression
- The amortized cost per operation for this data structure is $\Theta(\alpha(n))$ which is very slowly growing
  - This is the best that is possible!