Shahin Kamali

Disjoin Sets and Union-Find Structures

CLRS 21.121.4

University of Manitoba
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 \texttt{make-set} and \texttt{union} operations
  - Each set is identified by a \textit{representative element} of the set.

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

\[
k = 4; \quad S_a = \{a, b, m, n\}, \quad S_c = \{c, g, h\}, \quad S_e = \{d, e, f\}, \quad S_q = \{q\}
\]
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., \( \text{makeSet}({p}) \)

\[
k = 4; \quad S_a = \{a, b, m, n\}, \quad S_c = \{c, g, h\}, \quad S_e = \{d, e, f\}, \quad S_q = \{q\} \\
S_p = \{p\}
Disjoint Sets Operations

- \textbf{find}(x) \text{ (also called Find-Set(x))}:
  - Return the representative element of the set containing \( x \).

\[ k = 4; \quad S_a = \{a, b, m, n\}, \quad S_c = \{c, g, h\}, \quad S_e = \{d, e, f\}, \quad S_q = \{q\}, \]
Disjoint Sets Operations

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

E.g., $\text{find}(b) \rightarrow a$

\[ k = 4; \quad S_a = \{a, b, m, n\}, \quad S_c = \{c, g, h\}, \quad S_e = \{d, e, f\}, \quad S_q = \{q\}, \]
Disjoint Sets Operations

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

E.g., \( \text{find}(b) \rightarrow a \)
E.g., \( \text{find}(c) \rightarrow c \)

\( k = 4; \quad S_a = \{a, b, m, n\}, \quad S_c = \{c, g, h\}, \quad S_e = \{d, e, f\}, \quad S_q = \{q\} \),
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 \( x \cup y \).
  - \( \text{union}(x, y) \) is equivalent to \( \text{union}(\text{find}(x), \text{find}(y)) \).

\[ k = 4; \quad S_a = \{a, b, m, n\}, \quad S_c = \{c, g, h\}, \quad S_e = \{d, e, f\}, \quad S_q = \{q\} \]
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 \(x \cup y\).
  - \(\text{union}(x, y)\) is equivalent to \(\text{union}(\text{find}(x), \text{find}(y))\).

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

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

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

$\{A\} \{B\} \{C\} \{D\} \\
\{E\} \{F\} \{G\} \{H\}$

$\text{find}(B) \neq \text{find}(E) \Rightarrow \text{add to MST}$

$\text{union}(B, E)$
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.

```
(A) {B, E} {C} {D} {F} {G} {H}
```

```
find(G) \neq \text{find(H)}
add to MST
union(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.

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

find($G$) $\neq$ find($F$)
add to MST
union($G, F$)
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

\[
\begin{align*}
\{A\} & \quad \{B, E\} \quad \{C\} \\
\{D\} & \quad \{F, G, H\}
\end{align*}
\]

\[
\text{find}(A) \neq \text{find}(D) \\
\text{add to MST} \\
\text{union}(A, D)
\]
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

\[
\begin{align*}
\text{find}(A) &\neq \text{find}(C) \\
\text{add to MST} \\
\text{union}(A,C)
\end{align*}
\]

\{A, D\}, \{B, E\}, \{C\}, \{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.

\[
\text{find}(C) = \text{find}(D) \\
\text{DO NOTHING}
\]

\{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.

$\text{find}(E) \neq \text{find}(G)$

union$(E, G)$

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

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

[edge $(C, F)$ added to MST, with find$(C) \neq$ find$(F)$]

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.

{A, C, D, B, E, F, G, H}
Disjoint Sets Review

- **Disjoint set** is an abstract data type for maintaining a set of disjoint 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 disjoint 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.

\[ S_1 = \{ x, p \} \]

\[ S_2 = \{ a, h, c \} \]
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.

\[
S_1 = \{x, p\} \quad S_2 = \{a, h, c\}
\]
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.

\[ S_1 = \{x, p\} \]
\[ S_2 = \{a, h, c\} \]
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)**

\[ S_1 = \{ x, p \} \]
\[ S_2 = \{ a, h, c \} \]
\[ S_1 = \{ q \} \]
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.

\[ \text{find}(h) \rightarrow a \]
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

\[
\text{find(h)} \rightarrow a
\]
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 \( \Theta(n) \)).
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.

- \text{makeSet}(x_i) \text{ for } i \in \{1, 2 \ldots, n\}
- \text{union}(x_i, x_{i-1}) \text{ for } i \in \{n, n - 1, \ldots, 2\}, \text{ that is:}
  - \text{union}(x_n, x_{n-1}): \text{update 1 set-pointers}
  - \text{union}(x_{n-1}, x_{n-2}): \text{update 2 set-pointers}
  - \ldots
  - \text{union}(x_{n-i+1}, x_{n-i}): \rightarrow \text{update } i \text{ set-pointers}
  - \ldots
  - \text{union}(x_2, x_1): \text{updated } n - 1 \text{ 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, x_{n-1})`: update 1 set-pointers
  - `union(x_{n-1}, x_{n-2})`: update 2 set-pointers
  - $\ldots$
  - `union(x_{n-i+1}, x_{n-i})`: $\rightarrow$ update $i$ set-pointers
  - $\ldots$
  - `union(x_2, x_1)`: 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 worstcase.
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($x_n, x_{n-1}$): update 1 set-pointers
    - union($x_{n-1}, x_{n-2}$): update 2 set-pointers
    - \ldots
    - union($x_{n-i+1}, x_{n-i}$): update $i$ set-pointers
    - \ldots
    - union($x_2, x_1$): 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 worstcase.
    - This is a worst-case amortized time, e.g., for a sequence of $m$
      operations formed by $m$ make-sets, 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, x_{n-1})`: update 1 set-pointers
    - `union(x_{n-1}, x_{n-2})`: update 2 set-pointers
    - ...
    - `union(x_{n-i+1}, x_{n-i})`: update $i$ set-pointers
    - ...
    - `union(x_2, x_1)`: 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, e.g., for a sequence of $m$ operations formed by $m$ make-sets, 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 $\Theta(n)$. 

- COMP 3170 - Analysis of Algorithms & Data Structures
Review of Linked lists for Disjoint Set

- We want to maintain a set of disjoint sets so that make-set, find, and union operations can be performed efficiently.
  - make-set(x) creates a new set \{x\}.
  - find(x) returns the (representative item of) the set that \(x\) belongs to.
  - union(x,y) merges the sets that \(x, y\) belong to.
Review of Linked lists for Disjoint Set

- We want to maintain a set of disjoint sets so that make-set, find, and union operations can be performed efficiently.
  - make-set(x) creates a new set \{x\}.
  - find(x) returns the (representative item of) the set that x belongs to
  - union(x,y) merges the sets that x, y belong to.

Disjoint set via linked lists: each set is represented by a linked list
  - Each node has a set-pointer to the set object, which makes find(x) run in constant time
  - For union(x,y), we append one list to the end of another
    - This requires updating all set pointers of the appended list
We want to maintain a set of disjoint sets so that make-set, find, and union operations can be performed efficiently.

- make-set(x) creates a new set \{x\}.
- find(x) returns the (representative item of) the set that x belongs to.
- union(x,y) merges the sets that x, y belong to.

Disjoint set via linked lists: each set is represented by a linked list

- Each node has a set-pointer to the set object, which makes find(x) run in constant time.
- For union(x,y), we append one list to the end of another.
  - This requires updating all set pointers of the appended list.
- We saw in the last class that if we append the second list to the first one, there are *worst-case scenarios* such that even *amortized cost* of each operation 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.
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.
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.

Ignoring the cost of set-pointer updates, the cost of union for other pointer updates and also find and make-set are constants \( \rightarrow \Theta(m) \) cost for \( m \) operation.
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.

- Ignoring the cost of set-pointer updates, the cost of union for other pointer updates and also find and make-set are constants $\rightarrow \Theta(m)$ cost for $m$ operation

- Union by Weight has a cost of $O(n \log n + m)$ for a sequence of $m$ operations on a universe of size $n$
  - The amortized cost per operation is $O(n \log n/m + 1) = O(\log n)$
  - Note that $m \geq n$ since we need $m$ operations to make a universe of size $n$. 
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.

Ignoring the cost of set-pointer updates, the cost of union for other pointer updates and also find and make-set are constants \(\Theta(m)\) cost for \(m\) operation.

Union by Weight has a cost of \(O(n \log n + m)\) for a sequence of \(m\) operations on a universe of size \(n\).

- The amortized cost per operation is \(O(n \log n / m + 1) = O(\log n)\).
- Note that \(m \geq n\) since we need \(m\) operations to make a universe of size \(n\).

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 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

\[
S_1 = \{x, p\} \quad S_2 = \{a, h, c, f\}
\]
Disjoint Set Forests

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

\[
\begin{align*}
S_1 &= \{x, p\} \\
S_2 &= \{a, h, c, f\}
\end{align*}
\]
Disjoint Set Forests

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

$S_1 = \{x, p\}$
$S_2 = \{a, h, c, f\}$
Disjoint Set Forests

- Union(x, y) (first approach):
  - Set root of y’s tree to point to the root of x’s tree.
    - root_{x} ← find(x)
    - root_{y} ← find(y)
    - parent(root_{y}) ← root_{x}.
  - Time is proportional to tree’s height

\[
S_1 = \{x, p\} \quad S_2 = \{a, h, c, f\} \quad \{x, p, a, h, c, f\}
\]
Disjoint Set Forests

- Union($x,y$) (first approach):
  - Set root of $y$’s tree to point to the root of $x$’s tree.
    - $\text{root}_x \leftarrow \text{find}(x)$
    - $\text{root}_y \leftarrow \text{find}(y)$
    - $\text{parent}(	ext{root}_y) \leftarrow \text{root}_x$.
  - 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:
    - $\text{make-set}(x_i)$ for $i \in \{1, \ldots, n\}$
    - $\text{union}(x_i, x_1)$ for $i \in \{2, \ldots, n\}$.
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

\[
\text{root}_x \leftarrow \text{find}(x); \quad \text{root}_y \leftarrow \text{find}(y) \\
\text{if } \text{rank}(	ext{root}_x) > \text{rank}(	ext{root}_y) \\
\quad \text{parent}(	ext{root}_y) \leftarrow \text{root}_x \\
\text{else} \\
\quad \text{parent}(	ext{root}_x) \leftarrow \text{root}_y \\
\text{if } \text{rank}(	ext{root}_x) = \text{rank}(	ext{root}_y) \\
\quad \text{rank}(	ext{root}_y) \leftarrow \text{rank}(	ext{root}_y) + 1
\]
Union by Rank

- If $\text{rank}(x) = h$, the tree rooted at $x$ has at least $2^h$ nodes.
Union by Rank

- If \( \text{rank}(x) = h \), 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 \( \text{rank}(x) = h \), 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.
If $\text{rank}(x) = h$, 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 \( \text{rank}(x) = h \), 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
  - \text{Find}(x) \text{ involves finding a path from } x \text{ to the root of its tree}
  - For each node on the path, updated its pointer to point directly to the root:

\[ \text{Time complexity remains the same asymptotically, i.e., } O(\log n). \]

For any \( y \) that used to lie on the path from \( x \) to the root, any subsequent call to \text{find}(y) \text{ takes } O(1) \text{ time, the amortized time is significantly improved.}
Path Compression

- A simple, effective add on to union by rank
  - $\text{Find}(x)$ involves finding a path from $x$ to the root of its tree
  - For each node on the path, updated its pointer to point directly to the root:
    
    ```
    \text{if } x \neq \text{parent}(x)
    \hspace{1em} \text{parent}(x) \leftarrow \text{find}(\text{parent}(x))
    \hspace{1em} \text{return parent}(x)
    ```

  - For any $y$ that used to lie on the path from $x$ to the root, any subsequent call to $\text{find}(y)$ takes $O(1)$ time, the amortized time is significantly improved.
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, updated its pointer to point directly to the root:
    \[
    \text{if } x \neq \text{parent}(x) \\
    \text{parent}(x) \leftarrow \text{find(parent}(x)) \\
    \text{return parent}(x)
    \]
  - 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 \(\text{find}(y)\) takes \(O(1)\) time. The amortized time is significantly improved.
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:
    ```python
    if x != parent(x):
        parent(x) ← 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., $O(\log n)$.
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, updated its pointer to point directly to the root:
    
    \[
    \text{if } x \neq \text{parent}(x) \\
    \text{parent}(x) \leftarrow \text{find}(	ext{parent}(x)) \\
    \text{return parent}(x)
    \]
  
  - 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 $\text{find}(y)$ 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

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 the inverse Ackermann function. For any practical reason, $\alpha(n) \leq 4$.

In practice (not in theory) you can support disjoint operations in constant time.
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

\[
\begin{align*}
  f^{(0)}(n) &= n = 2^0 n, \\
  f^{(1)}(n) &= f(f^{(0)}(n)) = 2(n) = 2^1 n, \\
  f^{(2)}(n) &= f(f^{(1)}(n)) = 2(2^1 n) = 2^2 n, \\
  &\vdots \\
  f^{(i)}(n) &= f(f^{(i-1)}(n)) = 2(2^{i-1} n) = 2^i n,
\end{align*}
\]
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
  $$f^{(0)}(n) = n = 2^0 n,$$
  $$f^{(1)}(n) = f(f^{(0)}(n)) = 2(n) = 2^1 n,$$
  $$f^{(2)}(n) = f(f^{(1)}(n)) = 2(2^1 n) = 2^2 n,$$
  $$\ldots$$
  $$f^{(i)}(n) = f(f^{(i-1)}(n)) = 2(2^{i-1} n) = 2^i n,$$

- E.g., if $f(n) = 2^n$, then
  $$f^{(0)}(n) = n$$
  $$f^{(1)}(n) = f(f^{(0)}(n)) = f(n) = 2^n$$
  $$f^{(2)}(n) = f(f^{(1)}(n)) = f(2^n) = 2^{2^n}$$
  $$\ldots$$
  $$f^{(i)}(n) = f(f^{(i-1)}(n)) = 2^{2^{\ldots^{2^n}} \text{ } \text{ } \text{ } i \text{ times}}$$
For any $k \geq 0$ and $j \geq 1$, let

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

$A_k(j)$ is strictly increasing in both $j$ and $k$.

For $j > 0$, $A_1(j) = 2j + 1$.

For $j > 0$, $A_2(j) = 2j + 1(j+1) - 1$.

$A_3(1) = A_2(A_2(1)) = 2^8 - 1 = 255$.

$A_4(1) = A_2(A_3(1)) = 2^{255} - 1 > 2^{10^8}$

$A_4(1)$ is by far larger than the number of atoms in the universe.
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 \( n \)

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

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).
\( \alpha(n) \) 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 \( A_k(n) \): 
  \[
  \alpha(n) = \min \{ k \mid A_k(1) \geq n \}
  \]
  - \( \alpha(n) \) 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

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!
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!