5.3: Disjoint Sets

Frank Stajano

Thomas Sauerwald

Lent 2016
Disjoint Sets
Disjoint Sets (aka Union Find)

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains $x$
  - Behaviour: create a new set $\{x\}$ and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains $x$ (given pointer to $x$)
  - Behaviour: return the handle of the set that contains $x$

- **Handle Union(Handle h, Handle g)**
  - Precondition: $h \neq g$
  - Behaviour: merge two disjoint sets and return handle of new set $\{h, g\}$

Disjoint Sets Data Structure

- $h_0 = \text{MakeSet}(x)$
- $h_1 = \text{FindSet}(y)$
- $h_4 = \text{Union}(h_0, h_3)$
- $h_5 = \text{Union}(h_1, h_2)$
- $h_2 = h_3$
- $h_5 = h_4$

5.3: Disjoint Sets
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

\[ h_0 = \text{MakeSet}(x) \]
Disjoint Sets (aka Union Find)

**Disjoint Sets Data Structure**

- **Handle MakeSet(Item x)**
  
  Precondition: none of the existing sets contains \( x \)
  
  Behaviour: create a new set \{x\} and return its handle

\[ h_0 = \text{MakeSet}(x) \]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains \( x \)
  - Behaviour: create a new set \( \{x\} \) and return its handle

\[ h_0 = \text{MakeSet}(x) \]
Disjoint Sets (aka Union Find)

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \( \{x\} \) and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

Disjoint Sets Data Structure

\[
\text{h}_0 = \text{MakeSet}(x) \\
\text{h}_1 = \text{FindSet}(y) \\
\text{h}_4 = \text{Union}(\text{h}_0, \text{h}_3) \\
\text{h}_5 = \text{Union}(\text{h}_1, \text{h}_2) \\
\]

y

x

\( h_0 \)
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \( \{x\} \) and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

\[ h_1 = \text{FindSet}(y) \]
Disjoint Sets (aka Union Find)

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

Disjoint Sets Data Structure

\[
h_1 = \text{FindSet}(y)
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: \( h \neq g \)
  - Behaviour: merge two disjoint sets and return handle of new set

![Disjoint Sets Diagram](image)
Disjoint Sets (aka Union Find)

**Disjoint Sets Data Structure**

- **Handle MakeSet(Item x)**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  Precondition: h \neq g
  Behaviour: merge two **disjoint** sets and return handle of new set

\[ h_4 = \text{Union}(h_0, h_3) \]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: h \neq g
  - Behaviour: merge two *disjoint* sets and return handle of new set

```
h_4 = \text{Union}(h_0, h_3)
```
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: h \neq g
  - Behaviour: merge two disjoint sets and return handle of new set

\[ h_4 = \text{Union}(h_0, h_3) \]
Disjoint Sets (aka Union Find)

**Disjoint Sets Data Structure**

- **Handle `MakeSet(Item x)`**
  - **Precondition:** none of the existing sets contains `x`
  - **Behaviour:** create a new set `{x}` and return its handle

- **Handle `FindSet(Item x)`**
  - **Precondition:** there exists a set that contains `x` (given pointer to `x`)
  - **Behaviour:** return the handle of the set that contains `x`

- **Handle `Union(Handle h, Handle g)`**
  - **Precondition:** `h \neq g`
  - **Behaviour:** merge two disjoint sets and return handle of new set

\[
\begin{align*}
h_4 &= \text{Union}(h_0, h_3) \\
h_2 &\rightarrow h_1 & h_3 = h_4 \\
h_0 &\rightarrow y & x
\end{align*}
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: \( h \neq g \)
  - Behaviour: merge two disjoint sets and return handle of new set

\[
\begin{align*}
\text{h}_0 &= \text{MakeSet}(x) \\
\text{h}_1 &= \text{FindSet}(y) \\
\text{h}_4 &= \text{Union}(\text{h}_0, \text{h}_3) \\
\text{h}_5 &= \text{Union}(\text{h}_1, \text{h}_2) \\
\end{align*}
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set \( \{x\} \) and return its handle

- **Handle FindSet(Item x)**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  Precondition: \( h \neq g \)
  Behaviour: merge two disjoint sets and return handle of new set

\[
h_5 = \text{Union}(h_1, h_2)
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: \( h \neq g \)
  - Behaviour: merge two disjoint sets and return handle of new set

\[ h_5 = \text{Union}(h_1, h_2) \]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  
  Precondition: none of the existing sets contains $x$
  
  Behaviour: create a new set $\{x\}$ and return its handle

- **Handle FindSet(Item x)**
  
  Precondition: there exists a set that contains $x$ (given pointer to $x$)
  
  Behaviour: return the handle of the set that contains $x$

- **Handle Union(Handle h, Handle g)**
  
  Precondition: $h \neq g$
  
  Behaviour: merge two disjoint sets and return handle of new set

$h_5 = \text{Union}(h_1, h_2)$
Disjoint Sets (aka Union Find)

**Disjoint Sets Data Structure**

- **Handle MakeSet(Item x)**
  Precondition: none of the existing sets contains \( x \)
  Behaviour: create a new set \( \{x\} \) and return its handle

- **Handle FindSet(Item x)**
  Precondition: there exists a set that contains \( x \) (given pointer to \( x \))
  Behaviour: return the handle of the set that contains \( x \)

- **Handle Union(Handle h, Handle g)**
  Precondition: \( h \neq g \)
  Behaviour: merge two disjoint sets and return handle of new set

\[
\begin{align*}
  h_5 &= \text{Union}(h_1, h_2) \\
  h_3 &= h_4
\end{align*}
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: \( h \neq g \)
  - Behaviour: merge two disjoint sets and return handle of new set

\[ h_5 = \text{Union}(h_1, h_2) \]
First Attempt: List Implementation

Add extra pointer to the last element in each list ⇒ UNION-Operation

Add backward pointer to the list head from everywhere ⇒ FIND-SET takes constant time

Union \((h_1, h_2)\)

Need to find last element!

FindSet \((z_3)\)

\(h_4\)

\(z_1\)

\(z_2\)

\(z_3\)

\(z_4\)

Need to update all backward pointers!
First Attempt: List Implementation

**UNION-Operation**

\[ \text{Union}(h_1, h_2) \]

- \( h_1 \) \( h_2 \)
- \( x_1 \rightarrow x_2 \rightarrow x_3 \)
- \( y_1 \rightarrow y_2 \)

Need to find the last element!

FindSet(\( z_3 \))

Need to update all backward pointers!
**First Attempt: List Implementation**

**UNION-Operation**

- Add extra pointer to the last element in each list

**UNION**-

**FindSet**-Operation

- Add backward pointer to the list head from everywhere

**FindSet**-Operation

- Takes constant time

**Union**($h_1, h_2$)

Need to find last element!

```
|x1|→|x2|→|x3|
  ↓   ↓  ↓
  h1  h2

|y1|→|y2|
  ↓
  y1
```

Need to update all backward pointers!
First Attempt: List Implementation

**UNION-Operation**

- Add extra pointer to the last element in each list

```
Union(h_1, h_2)
Need to find last element!
```

```
\[
\begin{array}{c}
\text{Union}(h_1, h_2) \\
\hline
h_1 \\
\downarrow \\
x_1 \rightarrow x_2 \rightarrow x_3 \\
\hline
h_2 \\
\downarrow \\
y_1 \rightarrow y_2 \\
\end{array}
\]```

```
\text{Need to update all backward pointers!}
```
**First Attempt: List Implementation**

- **UNION-Operation**
  - Add extra pointer to the last element in each list

```
 Union(h₁, h₂)
```

Need to find last element!

```latex
\text{Union}(h₁, h₂)
```

Need to update all backward pointers!
First Attempt: List Implementation

**UNION-Operation**
- Add *extra pointer* to the last element in each list
⇒ UNION takes constant time

```
Union(h₁, h₂)
```

- Need to find last element!
- Need to update all backward pointers!
**First Attempt: List Implementation**

---

**UNION-Operation**
- Add **extra pointer** to the last element in each list
  ⇒ UNION takes constant time

---

**FINDSET-Operation**

---

**Union**($h_1, h_2$)

- Need to find last element!
- Need to update all backward pointers!
First Attempt: List Implementation

**UNION-Operation**
- Add extra pointer to the last element in each list
  ⇒ UNION takes constant time

**FINDSET-Operation**

**Union**($h_1, h_2$)

**FindSet**($z_3$)
First Attempt: List Implementation

**UNION-Operation**
- Add *extra pointer* to the last element in each list
  ⇒ UNION takes constant time

---

**FINDSET-Operation**
- Add *backward pointer* to the list head from everywhere

---

**Union** \((h_1, h_2)\)

![Diagram of UNION operation]

**FindSet** \((z_3)\)

![Diagram of FINDSET operation]
**First Attempt: List Implementation**

**UNION-Operation**
- Add extra pointer to the last element in each list
  ⇒ UNION takes constant time

**FINDSET-Operation**
- Add backward pointer to the list head from everywhere

### UNION-Operation

\[ \text{Union}(h_1, h_2) \]

![Diagram of UNION operation]

### FINDSET-Operation

\[ \text{FindSet}(z_3) \]

![Diagram of FINDSET operation]
First Attempt: List Implementation

**UNION-Operation**
- Add extra pointer to the last element in each list
  - UNION takes constant time

**FINDSET-Operation**
- Add backward pointer to the list head from everywhere
  - FINDSET takes constant time

**Union**($h_1, h_2$)

**FindSet**($z_3$)

---

5.3: Disjoint Sets
First Attempt: List Implementation

**UNION-Operation**

- Add extra pointer to the last element in each list
  ⇒ UNION takes constant time

**FINDSET-Operation**

- Add backward pointer to the list head from everywhere
  ⇒ FINDSET takes constant time

---

**Union**

(\(h_1, h_2\))

\[ h_1 \]

\[ x_1 \rightarrow x_2 \rightarrow x_3 \]

\[ h_2 \]

\[ y_1 \rightarrow y_2 \]

---

**FindSet**

(\(z_3\))

\[ h_4 \]

\[ z_1 \rightarrow z_2 \rightarrow z_3 \rightarrow z_4 \]
First Attempt: List Implementation

**UNION-Operation**
- Add **extra pointer** to the last element in each list
  ⇒ UNION takes constant time

**FINDSET-Operation**
- Add **backward pointer** to the list head from everywhere
  ⇒ FINDSET takes constant time

---

**Union** \((h_1, h_2)\)

Need to find last element!

**FindSet** \((z_3)\)

Need to update all backward pointers!
First Attempt: List Implementation

**UNION-Operation**

- Add *extra pointer* to the last element in each list
  ⇒ UNION takes constant time

**FINDSET-Operation**

- Add *backward pointer* to the list head from everywhere
  ⇒ FINDSET takes constant time

**UNION($h_1, h_2$)**

![Diagram showing the effect of UNION operation on the linked lists.]

**FINDSET($z_3$)**

![Diagram showing the effect of FINDSET operation on the linked lists.]

Need to update all backward pointers!
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
First Attempt: List Implementation (Analysis)

\( d = \text{DisjointSet()} \)
\( h_0 = d.\text{MakeSet}(x_0) \)

\( h_1 = d.\text{MakeSet}(x_1) \)
\( h_0 = d.\text{Union}(h_1, h_0) \)

Cost for \( n \) \( \text{Union} \) operations:
\[
\sum_{i=1}^{n} i = \Theta(n^2)
\]

Better to append shorter list to longer

\( \Rightarrow \)

Weighted-Union Heuristic
First Attempt: List Implementation (Analysis)

\[
\begin{align*}
  d &= \text{DisjointSet}() \\
  h_0 &= d.\text{MakeSet}(x_0) \\
  h_1 &= d.\text{MakeSet}(x_1) \\
  h_0 &= d.\text{Union}(h_1, h_0)
\end{align*}
\]
First Attempt: List Implementation (Analysis)

\[
d = \text{DisjointSet}()
\]
\[
h_0 = d.\text{MakeSet}(x_0)
\]
\[
h_1 = d.\text{MakeSet}(x_1)
\]
\[
h_0 = d.\text{Union}(h_1, h_0)
\]

Cost for \( n \) UNION operations:
\[
\sum_{i=1}^{n} i = \Theta(n^2)
\]

Better to append shorter list to longer

\[
\begin{array}{c}
\text{h}_0 \\
\downarrow \\
\text{x}_1 \\
\rightarrow \\
\text{x}_0
\end{array}
\]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]

Cost for \( n \) UNION operations:

\[ \sum_{i=1}^{n} i = \Theta(n^2) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]

\[ h_0 \]
\[ \downarrow \]

\[ x_2 \]

\[ h_2 \]

\[ x_1 \]

\[ \longrightarrow \]

\[ x_0 \]

Cost for \( n \) UNION operations:

\[ \sum_{i=1}^{n} i = \Theta(n^2) \]

better to append shorter list to longer

\( \Rightarrow \)

Weighted-Union Heuristic
First Attempt: List Implementation (Analysis)

\[
d = \text{DisjointSet}()
\]
\[
h_0 = d.\text{MakeSet}(x_0)
\]
\[
h_1 = d.\text{MakeSet}(x_1)
\]
\[
h_0 = d.\text{Union}(h_1, h_0)
\]
\[
h_2 = d.\text{MakeSet}(x_2)
\]
\[
h_0 = d.\text{Union}(h_2, h_0)
\]
First Attempt: List Implementation (Analysis)

\( d = \text{DisjointSet}() \)
\( h_0 = d.\text{MakeSet}(x_0) \)

\( h_1 = d.\text{MakeSet}(x_1) \)
\( h_0 = d.\text{Union}(h_1, h_0) \)
\( h_2 = d.\text{MakeSet}(x_2) \)
\( h_0 = d.\text{Union}(h_2, h_0) \)
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]

Cost for \( n \) \( \text{UNION} \) operations:

\[ \sum_{i=1}^{n} i = \Theta(n^2) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]

Cost for \( n \) UNION operations:

\[ \sum_{i=1}^{n} i = \Theta(n^2) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]
\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]

\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]

\[ h_0 = d.\text{Union}(h_1, h_0) \]

\[ h_2 = d.\text{MakeSet}(x_2) \]

\[ h_0 = d.\text{Union}(h_2, h_0) \]

\[ h_3 = d.\text{MakeSet}(x_3) \]

\[ h_0 = d.\text{Union}(h_3, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]

Cost for \( n \) UNION operations:
\[ \sum_{i=1}^{n} i = \Theta(n^2) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]
\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]

Cost for \( n \) \text{UNION} operations: \( \sum_{i=1}^{n} i = \Theta(n^2) \)
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]

Cost for \( n \) \text{UNION} operations: \( \sum_{i=1}^{n} i = \Theta(n^2) \)

better to append shorter list to longer \( \Rightarrow \) Weighted-Union Heuristic
Weighted-Union Heuristic

- Keep track of the length of each list
Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)
Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

**Theorem 21.1**

Amortized Analysis: Every operation has amortized cost $O(\log n)$, but there may be operations with total cost $\Theta(n)$. 

5.3: Disjoint Sets
Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

Weighted-Union Heuristic can be done easily without significant overhead

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are \texttt{MAKESET} operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.
Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are MAKESET operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

Amortized Analysis: Every operation has amortized cost \( \mathcal{O}(\log n) \), but there may be operations with total cost \( \Theta(n) \).
Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are \text{MAKESET} operations, takes \( O(m + n \cdot \log n) \) time.
Analysis of Weighted-Union Heuristic

Using the Weighted-Union heuristic, any sequence of $m$ operations, $n$ of which are MAKESET operations, takes $O(m + n \cdot \log n)$ time.

Proof:
Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are \textsc{MakeSet} operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

Proof:

- \( n \textsc{Make-Set} \) operations \( \Rightarrow \) at most \( n - 1 \textsc{Union} \) operations
Using the Weighted-Union heuristic, any sequence of $m$ operations, $n$ of which are MAKESET operations, takes $O(m + n \cdot \log n)$ time.

Proof:

- $n$ MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element $x$ and the number of updates of its backward pointer
Theorem 21.1

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are \textsc{Makeset} operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

Proof:

- \( n \textsc{Make-Set} \) operations \( \Rightarrow \) at most \( n - 1 \textsc{Union} \) operations
- Consider element \( x \) and the number of updates of its backward pointer
Analysis of Weighted-Union Heuristic

**Theorem 21.1**

Using the **Weighted-Union heuristic**, any sequence of \( m \) operations, \( n \) of which are **MAKESET** operations, takes \( O(m + n \cdot \log n) \) time.

**Proof:**

- \( n \) **MAKE-SET** operations \( \Rightarrow \) at most \( n - 1 \) **UNION** operations
- Consider element \( x \) and the number of updates of its backward pointer
Using the **Weighted-Union heuristic**, any sequence of \( m \) operations, \( n \) of which are \texttt{MAKESET} operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

Proof:

- \( n \texttt{MAKE-SET} \) operations ⇒ at most \( n - 1 \texttt{UNION} \) operations
- Consider element \( x \) and the number of updates of its backward pointer
Using the **Weighted-Union heuristic**, any sequence of \( m \) operations, \( n \) of which are \texttt{MAKESET} operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

**Proof:**

- \( n \) \texttt{MAKE-SET} operations \( \Rightarrow \) at most \( n - 1 \) \texttt{UNION} operations
- Consider element \( x \) and the number of updates of its backward pointer
- After each update of \( x \), its set increases by a factor of at least 2
Analysis of Weighted-Union Heuristic

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are \texttt{MAKESET} operations, takes \( O(m + n \cdot \log n) \) time.

**Theorem 21.1**

Proof:

- \( n \) \texttt{MAKE-SET} operations \( \Rightarrow \) at most \( n - 1 \) \texttt{UNION} operations
- Consider element \( x \) and the number of updates of its backward pointer
- After each update of \( x \), its set increases by a factor of at least 2
  \( \Rightarrow \) Backward pointer of \( x \) is updated at most \( \log_2 n \) times
Analysis of Weighted-Union Heuristic

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are \textsc{MakeSet} operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

Proof:

- \( n \) \textsc{MakeSet} operations \( \Rightarrow \) at most \( n - 1 \) \textsc{Union} operations
- Consider element \( x \) and the number of updates of its backward pointer
- After each update of \( x \), its set increases by a factor of at least 2
  \( \Rightarrow \) Backward pointer of \( x \) is updated at most \( \log_2 n \) times
- Other updates for \textsc{Union}, \textsc{MakeSet} & \textsc{FindSet} take \( \mathcal{O}(1) \) time per operation
Analysis of Weighted-Union Heuristic

Using the Weighted-Union heuristic, any sequence of $m$ operations, $n$ of which are \texttt{MAKESET} operations, takes $O(m + n \cdot \log n)$ time.

**Theorem 21.1**

Proof:

- $n$ \texttt{MAKE-SET} operations $\Rightarrow$ at most $n - 1$ \texttt{UNION} operations
- Consider element $x$ and the number of updates of its backward pointer
- After each update of $x$, its set increases by a factor of at least 2
  $\Rightarrow$ Backward pointer of $x$ is updated at most $\log_2 n$ times
- Other updates for \texttt{UNION}, \texttt{MAKE-SET} & \texttt{FIND-SET} take $O(1)$ time per operation

5.3: Disjoint Sets
Analysis of Weighted-Union Heuristic

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are \textsc{make-set} operations, takes \( O(m + n \cdot \log n) \) time.

**Theorem 21.1**

**Proof:**
- \( n \textsc{make-set} \) operations \( \Rightarrow \) at most \( n - 1 \textsc{union} \) operations
- Consider element \( x \) and the number of updates of its backward pointer
- After each update of \( x \), its set increases by a factor of at least 2
- \( \Rightarrow \) Backward pointer of \( x \) is updated at most \( \log_2 n \) times
- Other updates for \textsc{union}, \textsc{make-set} & \textsc{find-set} take \( O(1) \) time per operation
How to Improve?

**Basic Idea:** Update Backward

**Operations:**

- **MAKESET:** $O(1)$
- **FINDSET:** $O(n)$
- **UNION:** $O(1)$

**Doubly-Linked List**
How to Improve?

Doubly-Linked List
- \textbf{\textsc{MakeSet}}: \(\mathcal{O}(1)\)
- \textbf{\textsc{FindSet}}: \(\mathcal{O}(n)\)
- \textbf{\textsc{Union}}: \(\mathcal{O}(1)\)

Weighted-Union Heuristic
- \textbf{\textsc{MakeSet}}: \(\mathcal{O}(1)\)
- \textbf{\textsc{FindSet}}: \(\mathcal{O}(1)\)
- \textbf{\textsc{Union}}: \(\mathcal{O}(\log n)\) (amortized)
How to Improve?

Basic Idea: Update Backward Pointers only during FIND-SET

Doubly-Linked List
- **MAKESET**: $\mathcal{O}(1)$
- **FINDSET**: $\mathcal{O}(n)$
- **UNION**: $\mathcal{O}(1)$

Weighted-Union Heuristic
- **MAKESET**: $\mathcal{O}(1)$
- **FINDSET**: $\mathcal{O}(1)$
- **UNION**: $\mathcal{O}(\log n)$ (amortized)
Disjoint Sets via Forests

Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer $p$ to its parent (for root $x$, $x.p = x$)
Disjoint Sets via Forests

\{b, c, e, h\}

Forest Structure
- Set is represented by a rooted tree with root being the representative
- Every node has pointer .p to its parent (for root x, x.p = x)
Set is represented by a rooted tree with root being the representative

Every node has pointer .p to its parent (for root x, x.p = x)

**UNION**: Merge the two trees
Disjoint Sets via Forests

\[ \{b, c, e, h\} \quad \{d, f, g\} \]

Forest Structure
- Set is represented by a rooted tree with root being the representative
- Every node has pointer \( p \) to its parent (for root \( x, x.p = x \))
- \texttt{UNION}: Merge the two trees

Rank may be just an upper bound on the height!
Disjoint Sets via Forests

Set is represented by a rooted tree with root being the representative
Every node has pointer \( p \) to its parent (for root \( x \), \( x.p = x \))
**UNION**: Merge the two trees

Forest Structure

\[
\{b, c, e, h\} \quad \{d, f, g\} \quad \{b, c, d, e, f, g, h\}
\]
Disjoint Sets via Forests

\{b, c, e, h\} \quad \{d, f, g\} \quad \{b, c, d, e, f, g, h\}

Forest Structure

- Set is represented by a **rooted tree** with root being the representative
- Every node has pointer $p$ to its parent (for root $x$, $x.p = x$)
- **UNION**: Merge the two trees

Append tree of smaller height $\leadsto$ Union by Rank
Disjoint Sets via Forests

\{b, c, e, h\} \quad \{d, f, g\} \quad \{b, c, d, e, f, g, h\}

Forest Structure
- Set is represented by a rooted tree with root being the representative
- Every node has pointer \( p \) to its parent (for root \( x \), \( x.p = x \))
- \textsc{Union}: Merge the two trees

Append tree of smaller height \( \leadsto \) Union by Rank

\begin{itemize}
  \item \textbf{5.3: Disjoint Sets}
  \item T.S.
\end{itemize}
Disjoint Sets via Forests

- Set is represented by a **rooted tree** with root being the representative.
- Every node has **pointer .p** to its parent (for root \( x \), \( x.p = x \)).
- **UNION**: Merge the two trees.

Append tree of smaller height \( \sim \) **Union by Rank**.

\[
\{b, c, e, h\} \quad \{d, f, g\} \quad \{b, c, d, e, f, g, h\}
\]

Rank may be just an upper bound on the height!
Disjoint Sets via Forests

Set is represented by a rooted tree with root being the representative.

Every node has a pointer $p$ to its parent (for root $x$, $x.p = x$).

**UNION**: Merge the two trees.

Append tree of smaller height $\leadsto$ Union by Rank.

**Forest Structure**

- Rank may be just an upper bound on the height!

U

N

I

O

N

Union by Rank

$\{b, c, e, h\}$  $\{d, f, g\}$  $\{b, c, d, e, f, g, h\}$
Path Compression during FINDSET

FindSet(b):

0: `FindSet(x)`
1: `if x \neq x.p`
2: `x.p = FindSet(x.p)`
3: `return x.p`
Path Compression during **FINDSET**

**FindSet**(*b*):

```plaintext
0: **FindSet**(*x*)
1:    **if**  *x* ≠ *x*.p
2:        *x*.p = **FindSet**(*x*.p)
3:    **return**  *x*.p
```
Path Compression during **FINDSET**

**FindSet**($b$):

0: **FindSet**($x$)
1: if $x \neq x.p$
2: $x.p = \text{FindSet}(x.p)$
3: return $x.p$

---

5.3: Disjoint Sets
FindSet\((b)\):
Path Compression during FINDSET

**FindSet(b):**

```
0: FindSet(x)
1: if x != x.p
2: x.p = FindSet(x.p)
3: return x.p
```
Path Compression during **FINDSET**

**FindSet**($b$):

```
0: FindSet($x$)
1: if $x \neq x.p$
2: $x.p = \text{FindSet}(x.p)$
3: return $x.p$
```
Path Compression during **FINDSET**

**FindSet(b):**

```
0: FindSet(x)
1: if x \neq x.p
2: x.p = FindSet(x.p)
3: return x.p
```
Path Compression during **FINDSET**

**FindSet**(b):

```
FindSet(x):
  0: FindSet(x)
  1: if x \neq x.p
  2: x.p = FindSet(x.p)
  3: return x.p
```
Path Compression during `FINDSET`

**FindSet**($b$):

0: \textbf{FindSet}($x$)
1: \textbf{if} $x \neq x.p$
2: \hspace{1em} $x.p = \textbf{FindSet}(x.p)$
3: \textbf{return} $x.p$
Path Compression during **FINDSET**

**FindSet** \((b)\):

```
FindSet(x):
0: FindSet(x)
1: if \(x \neq x.p\)
2: \(x.p = \text{FindSet}(x.p)\)
3: return \(x.p\)
```
Path Compression during FINDSET

FindSet(b):

FindSet(x):

0: FindSet(x)
1: if x ≠ x.p
2: x.p = FindSet(x.p)
3: return x.p
Path Compression during `FINDSET`

\[ \text{FindSet}(b) : \]

\[
\begin{array}{c}
  b \\
  h \\
  c \\
\end{array}
\]

0: `FindSet(x)`
1: \textbf{if} \( x \neq x.p \)
2: \( x.p = \text{FindSet}(x.p) \)
3: \textbf{return} \( x.p \)
Path Compression during **FINDSET**

**FindSet(b):**

\[ \text{FindSet}(b) : \]

\[
\begin{array}{c}
\text{b} \\
\hline
\text{h}
\end{array}
\]

0: **FindSet**(*x*)
1: if *x* ≠ *x*.\(p\)
2: \(x.\text{p} = \text{FindSet}(x.\text{p})\)
3: return \(x.\text{p}\)
Path Compression during \textbf{FINDSET}

\textbf{FindSet}(b):

\begin{itemize}
  \item \textbf{FindSet}(x)
  \item \textbf{if} \hspace{0.5em} x \neq x.p
  \item \hspace{1em} x.p = \textbf{FindSet}(x.p)
  \item \textbf{return} \hspace{0.5em} x.p
\end{itemize}
**FindSet**($b$):

```
0: FindSet($x$)  
1:   if $x \neq x.p$  
2:     $x.p = \text{FindSet}(x.p)$  
3:   return $x.p$
```
Path Compression during FINDSET

**FindSet**\((b)\):

\[
\begin{align*}
0: & \text{FindSet}(x) \\
1: & \quad \textbf{if} \ x \neq x.p \\
2: & \quad x.p = \text{FindSet}(x.p) \\
3: & \quad \textbf{return} \ x.p
\end{align*}
\]
Path Compression during `FINDSET`

**FindSet** *(b)*:

```
FindSet(b):
```

```
b
```

```
0: FindSet(x)
1: if x ≠ x.p
2: x.p = FindSet(x.p)
3: return x.p
```
Path Compression during **FINDSET**

**FindSet(b):**

```
FindSet(x)
1:   if x != x.p
2:       x.p = FindSet(x.p)
3:   return x.p
```
Path Compression during FINDSET

FindSet(b):

0: FindSet(x)
1: if x ≠ x.p
2: x.p = FindSet(x.p)
3: return x.p
Path Compression during FINDSET

FindSet\((b)\):

```
0: FindSet\((x)\)
1: if \(x \neq x.p\)
2: \(x.p = \text{FindSet}(x.p)\)
3: return \(x.p\)
```

Maintaining the exact height would be costly, hence rank is only an **upper bound**!
Theorem 21.14

Any sequence of $m$ MAKESET, UNION, FINDSET operations, $n$ of which are MAKESET operations, can be performed in $O(m \cdot \alpha(n))$ time.
Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of \( m \) \textsc{MakeSet}, \textsc{Union}, \textsc{FindSet} operations, \( n \) of which are \textsc{MakeSet} operations, can be performed in \( O(m \cdot \alpha(n)) \) time.

\[
\alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80}
\end{cases}
\]
Combining Union by Rank and Path Compression

**Theorem 21.14**

Any sequence of $m$ MAKESET, UNION, FINDSET operations, $n$ of which are MAKESET operations, can be performed in $O(m \cdot \alpha(n))$ time.

\[ \alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80} 
\end{cases} \]

More than the number of atoms in the universe!
Combining Union by Rank and Path Compression

**Theorem 21.14**

Any sequence of $m$ MAKESET, UNION, FINDSET operations, $n$ of which are MAKESET operations, can be performed in $O(m \cdot \alpha(n))$ time.

\[
\alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80} 
\end{cases}
\]

$log^*(n)$, the iterated logarithm, satisfies $\alpha(n) \leq log^*(n)$, but still $log^*(10^{80}) = 5.$
Combining Union by Rank and Path Compression

**Theorem 21.14**

Any sequence of \( m \) MAKESET, UNION, FINDSET operations, \( n \) of which are MAKESET operations, can be performed in \( \mathcal{O}(m \cdot \alpha(n)) \) time.

In practice, \( \alpha(n) \) is a small constant

\[
\alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80} 
\end{cases}
\]
Combining Union by Rank and Path Compression

Any sequence of \( m \) \texttt{MAKESET}, \texttt{UNION}, \texttt{FINDSET} operations, \( n \) of which are \texttt{MAKESET} operations, can be performed in \( O(m \cdot \alpha(n)) \) time.

In practice, \( \alpha(n) \) is a small constant

\[ \alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80}.
\]
Simulating the Effects of Union by Rank and Path Compression

1. Initialise singletons 1, 2, ..., 300
2. For every 1 \leq i \leq 300, pick a random 1 \leq r \leq 300, r \neq i and perform UNION(FINDSET(i), FINDSET(r))
3. Perform j \in \{0, 100, 200, 300, 600, 900, 1200, 1500, 1800\} many additional FINDSET(r), where 1 \leq r \leq 300 is random

Experimental Setup 5.3: Disjoint Sets T.S. 12
Simulating the Effects of Union by Rank and Path Compression

---

**Experimental Setup**

1. Initialise singletons 1, 2, \ldots, 300
2. For every $1 \leq i \leq 300$, pick a random $1 \leq r \leq 300$, $r \neq i$ and perform $\text{UNION}(\text{FINDSET}(i), \text{FINDSET}(r))$
Simulating the Effects of Union by Rank and Path Compression

Experimental Setup

1. Initialise singletons 1, 2, . . . , 300
2. For every $1 \leq i \leq 300$, pick a random $1 \leq r \leq 300$, $r \neq i$ and perform $	ext{UNION}(	ext{FINDSET}(i), \text{FINDSET}(r))$
3. Perform $j \in \{0, 100, 200, 300, 600, 900, 1200, 1500, 1800\}$ many additional $	ext{FINDSET}(r)$, where $1 \leq r \leq 300$ is random
Union by Rank without Path Compression

Average Height: 2.12
Union by Rank with Path Compression

Average Height: 1.75
Union by Rank with Path Compression (100 additional FINDSET)

Average Height: 1.53
Union by Rank with Path Compression (200 additional FINDSET)

Average Height: 1.35
Union by Rank with Path Compression (300 additional FINDSET)

Average Height: 1.22

5.3: Disjoint Sets
Union by Rank with Path Compression (600 additional FINDSET)

Average Height: 1.08
Union by Rank with Path Compression (900 additional FINDSET)

Average Height: 1.02
Union by Rank with Path Compression (1200 additional FINDSET)

Average Height: 1.01
Union by Rank with Path Compression (1500 additional FINDSET)

Average Height: 1.00
Union by Rank with Path Compression (1800 additional FINDSET)

Average Height: 0.98
Union by Rank with Path Compression (1800 additional FINDSET)

Coupon Collecting Time: $300 \cdot \ln(300) \approx 1711$

Average Height: 0.98
<table>
<thead>
<tr>
<th></th>
<th>Union by Rank</th>
<th>Union by Rank &amp; Path Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>300 MAKESET &amp; 300 UNION</strong></td>
<td>2.12</td>
<td>1.75</td>
</tr>
<tr>
<td>100 extra FINDSET</td>
<td>2.12</td>
<td>1.53</td>
</tr>
<tr>
<td>200 extra FINDSET</td>
<td>2.12</td>
<td>1.35</td>
</tr>
<tr>
<td>300 extra FINDSET</td>
<td>2.12</td>
<td>1.22</td>
</tr>
<tr>
<td>600 extra FINDSET</td>
<td>2.12</td>
<td>1.08</td>
</tr>
<tr>
<td>900 extra FINDSET</td>
<td>2.12</td>
<td>1.02</td>
</tr>
<tr>
<td>1200 extra FINDSET</td>
<td>2.12</td>
<td>1.01</td>
</tr>
<tr>
<td>1500 extra FINDSET</td>
<td>2.12</td>
<td>1.00</td>
</tr>
<tr>
<td>1800 extra FINDSET</td>
<td>2.12</td>
<td>0.98</td>
</tr>
</tbody>
</table>