Let $S = \{e_1, e_2, \ldots, e_n\}$

$\Rightarrow$ Elements

At any point, $S$ may be partitioned into $\{S_1, S_2, \ldots, S_k\}$

Where these $S_i$ are disjoint and $\bigcup_{i=1}^{k} S_i = S$.

In the instances where this is used, initially

$S_i = \{e_i\}$ $i = 1 \ldots n$.

i.e. each set contains only one element. As the structure evolves, sets are combined (never broken) until at the end, in some cases, there is only $S$.

Each set is represented by one of its elements
(Choice depends on the algorithm designer)

There are 3 main operations within this data structure

One of these, called MAKE-SET($x$), is done only in the beginning. This process creates a set $\{x\}$ with $x$ as its only member; $x$ is the representative of $\{x\}$ depicted by $\mathbb{2}$.

We do this for each element of $S$. 

\[CS\ 6363: \ \text{Lecture} \#16\]

\[\text{Data Structures for Dynamic Disjoint Sets (UNION-FIND)} \ (\text{Ch} \ 21)\]
The second operation is called **FIND-SET(x)**; \(x \in S\)
and it returns the representative of \(S_x\), the set that contains \(x\) at that point. [This step involves a lot of "pointer" chasing and will need speeding up.]

The third operation is **UNION(x,y)**, \(x,y \in S\)

This operation replaces \(S_x\) and \(S_y\) (sets that contain \(x\) and \(y\) respectively) into a combined set \(S_{xy} = S_x \cup S_y\), and the representative of the new (UNION set) is either rep of \(S_x\) or rep of \(S_y\) before this operation.

We will use **Disjoint Forest Disjoint-Set Forest representation** [Linked list is not efficient]

An example of the use of this is to find Connected Components of an undirected graph and we use a "numerical" example to show the process. (See page 563, 569 of book.)
Given a graph:

\[ \begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{d} & \text{e} & \text{f} \\
\end{array} \]

\[ \begin{array}{ccc}
\text{g} & \text{h} & \text{i} \\
\text{j} & & \\
\end{array} \]

\[ S = \text{Set of nodes} \]

MAKE-SET(x) : for \( x = a, b, \ldots \)  

Result:  
\[ \begin{array}{ccc}
\text{a} & \text{b} & \text{j} \\
\end{array} \]

Now we "process" each edge one by one. For example:

\[ a \rightarrow b \]

\[ \text{FIND-SET(a)} = a \rightarrow \text{UNION}(a, b) \]

\[ \text{FIND-SET(b)} = b \]

\[ \begin{array}{ccc}
\text{a} & \text{c} & \text{j} \\
\text{b} & & \\
\end{array} \]

\[ a \rightarrow c \]

\[ \text{FIND-SET(a)} = a \rightarrow \text{UNION}(a, c) \]

\[ \text{FIND-SET(c)} = c \]

Less points → Make the root (rep) of the structure with fewer number of nodes point to root of structure with larger number of nodes.
They are four disjoint trees. So original graph has 4 different components each of which is connected.

If we want to know if nodes x and y are in same component: We do \text{FIND-SET}(x) \text{ and } \text{FIND-SET}(y)

- If they return same node, \(x, y\) in same component
- Else they are in different ones.
Use in Kruskal Algorithm A.

Recall this algorithm considers edge in increasing order of weights. At each step, we color the next edge blue if this does not create a cycle; else we color it red (do not include intra).

Checking this is where we use UNION-FIND data structure. We illustrate on an example for MST.

Recall edges arranged in increasing order of weight: CF, DE*, FH, AB*, EH, DF, CD*, EG, BE*, AC, GI, HI, BS, GH

Kruskal Alg A with UNION-FIND:

Step 1: MAKE-SET; S = \{V\};

\[ \text{A} \quad \text{B} \quad \ldots \quad \text{I} \]

Process edges in that order.

CF: FIND-SET(C) = C

\[ (F) = F \]

COLOR (F) blue

UNION (CF)

1

\[ \begin{array}{c}
\text{C} \\
\text{F}
\end{array} \]
DE
\[ \text{FIND-SET}(D) = D \]
\[ \text{FIND-SET}(E) = E \]
\[ \text{UNION}(D, E) \]
\[ \text{COLOR}(DE) \text{ blue} \]

FH
\[ \text{FIND-SET}(F) = C \]
\[ \text{FIND-SET}(H) = H \]
\[ \text{UNION}(F, H) \]
\[ \text{COLOR}(FH) \text{ blue} \]

To keep structure shallow to reduce pointer chasing during FIND-SET in future.

AB
\[ \text{FIND-SET}(A) = A \]
\[ \text{FIND-SET}(B) = B \]
\[ \text{UNION}(A, B) \]
\[ \text{COLOR}(AB) \text{ blue} \]
\[ \text{EH} \]
\[
\text{FIND-SET}(E) = D \\
\text{COLOR}(EH) \text{ blue}
\]
\[
\text{FIND-SET}(H) = C \\
\text{UNION}(EH)
\]
\[
\text{Some write UNION}(D,C)
\]

Root of "smaller" set to root of "larger" set to keep tree structure shallow.

\[ \text{DF} \]
\[
\text{FIND-SET}(D) = C \\
\text{COLOR}(DF) \text{ red}
\]
(else will create a blue cycle!)

\[
\text{FIND-SET}(F) = C
\]
I hope you can finish this.
There is one more trick to keep the structure shallow called **PATH COMPRESSION**.
This is done only at **FIND-SET** operation.

Suppose at some point we have **FIND-SET(x)**, and this looks at "parental" chain starting at x

Say it looks like:

```
  j
  V
  o
  o
  o
  x
```

Before:  

```
  j
  V
  o
  o
  o
  x
```

After:  

```
  j
  V
  o
  o
  o
  x
```

DO NOT PERFORM Path Compression

generate during **UNION** !!!
If we use both strategies for keeping structure:

Shallow, Complexity is $O(|E| \alpha^*(\frac{|E|}{|V|}))$

$\alpha^*$ is inverse Ackerman function, a very slowly growing function.

Hence Complexity of Kruskal A: $\Theta(|E| \log |E|)$

$\sqrt{|E|} = O(|V|^2)$

$|E| \log |E| = \Theta(|E| \log |V|)$

Compared favorably with Prim.

Now for proof of correctness of these algorithms → Next Lecture