Minimum Spanning Trees
via Disjoint Sets

Benjamin Sach
Minimum Spanning Trees
via Disjoint Sets

Benjamin Sach
In this lecture we will see an efficient data structure for maintaining a collection of disjoint sets.

We will then see how this data structure can be used to efficiently implement **Kruskal’s algorithm** which finds a minimum spanning tree in an undirected graph.
In this lecture we will see an efficient data structure for maintaining a collection of disjoint sets.

We will then see how this data structure can be used to efficiently implement **Kruskal’s algorithm** which finds a minimum spanning tree in an undirected graph.
Disjoint set data structures

We will be interested in a data structure which
stores a collection of disjoint sets

The elements of the sets are the numbers \( \{1, 2, \ldots, n\} \)

The following operations are supported:

- **MAKESET**\((x)\) - make a new set containing only \(x\)
  
  \(x\) cannot be a member of any existing set

- **UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

- **FINDSET**\((x)\) - returns the identity of the set containing \(x\)
  
  the identity of a set is any unique identifier of the set.

All we require from **FINDSET** is that
\[ \text{FINDSET}(x) = \text{FINDSET}(y) \]

if and only \(x\) and \(y\) are currently in the same set
Example

\textbf{\textsc{MakeSet}}(x) - make a new set containing only \( x \) (which is not already in a set)
\textbf{\textsc{Union}}(x, y) - merge the sets containing \( x \) and \( y \) into a single set
\textbf{\textsc{FindSet}}(x) - returns the \textit{identity} of the set containing \( x \)
Example

\text{\textsc{makeSet}}(x) - make a new set containing only \(x\) (which is not already in a set)

\text{\textsc{union}}(x, y) - merge the sets containing \(x\) and \(y\) into a single set

\text{\textsc{findSet}}(x) - returns the \textit{identity} of the set containing \(x\)

\textit{Let } n = 16 \textit{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}
Example

\textbf{MakeSet}(x) - make a new set containing only } x \text{ (which is not already in a set)}
\textbf{Union}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}
\textbf{FindSet}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{MakeSet}(3)
Example

\text{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\text{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\text{FINDSET}(x) - returns the } identity \text{ of the set containing } x

Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\text{MAKESET}(3)

\{3\}
**Example**

\( \text{MAKESET}(x) \) - make a new set containing only \( x \) (which is not already in a set)

\( \text{UNION}(x, y) \) - merge the sets containing \( x \) and \( y \) into a single set

\( \text{FINDSET}(x) \) - returns the *identity* of the set containing \( x \)

Let \( n = 16 \) so that the elements of the sets are the numbers \( \{1, 2, \ldots, 16\} \)

\[ \{3\} \]
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{MAKESET}(7)

\{3\}
Example

**MAKESET**(*x*) - make a new set containing only *x* (which is not already in a set)

**UNION**(*x*, *y*) - merge the sets containing *x* and *y* into a single set

**FINDSET**(*x*) - returns the *identity* of the set containing *x*

*Let* $n = 16$ *so that the elements of the sets are the numbers* \{1, 2, ..., 16\}

**MAKESET**(7)

\[
\{3\} \quad \{7\}
\]
Example

\textsc{MakeSet}(x) - make a new set containing only } x \text{ (which is not already in a set)

\textsc{Union}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set

\textsc{FindSet}(x) - returns the } identity \text{ of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\begin{align*}
\{3\} & \quad \{7\}
\end{align*}
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set

\textbf{FINDSET}(x) - returns the } \textit{identity} \text{ of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{MAKESET}(4)

\{3\} \quad \{7\}
Example

MAKESET(\(x\)) - make a new set containing only \(x\) (which is not already in a set)

UNION(\(x, y\)) - merge the sets containing \(x\) and \(y\) into a single set

FINDSET(\(x\)) - returns the identity of the set containing \(x\)

\[n = 16\] so that the elements of the sets are the numbers \(\{1, 2, \ldots, 16\}\)

MAKESET(4)

\[
\{3\} \quad \{7\} \quad \{4\}
\]
Example

\text{MAKESET}(x) - \text{make a new set containing only } x \text{ (which is not already in a set)}

\text{UNION}(x, y) - \text{merge the sets containing } x \text{ and } y \text{ into a single set}

\text{FINDSET}(x) - \text{returns the } \textit{identity} \text{ of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\{3\} \quad \{7\} \quad \{4\}
Example

**MAKE_SET(x)** - make a new set containing only $x$ (which is not already in a set)

**UNION(x, y)** - merge the sets containing $x$ and $y$ into a single set

**FIND_SET(x)** - returns the *identity* of the set containing $x$

---

Let $n = 16$ so that the elements of the sets are the numbers \{1, 2, \ldots, 16\}

---

**UNION(3, 7)**

\[
\{3\} \quad \{7\} \quad \{4\}
\]
Example

**MAKESET**(*x*) - make a new set containing only *x* (which is not already in a set)

**UNION**(*x*, *y*) - merge the sets containing *x* and *y* into a single set

**FINDSET**(*x*) - returns the *identity* of the set containing *x*

Let \( n = 16 \) so that the elements of the sets are the numbers \{1, 2, \ldots, 16\}

\[
\text{UNION}(3, 7)
\]

\[
\begin{align*}
\{3\} & \quad \{7\} & \quad \{4\} \\
\end{align*}
\]
Example

\textbf{MAKESET}(x) - make a new set containing only \(x\) (which is not already in a set)

\textbf{UNION}(x, y) - merge the sets containing \(x\) and \(y\) into a single set

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing \(x\)

Let \(n = 16\) so that the elements of the sets are the numbers \(\{1, 2, \ldots, 16\}\)

\textbf{UNION}(3, 7)

\[
\{3, 7\} \quad \{4\}
\]
**Example**

**MAKESET**($x$) - make a new set containing only $x$ (which is not already in a set)

**UNION**($x, y$) - merge the sets containing $x$ and $y$ into a single set

**FINDSET**($x$) - returns the *identity* of the set containing $x$

---

Let $n = 16$ so that the elements of the sets are the numbers \{1, 2, \ldots, 16\}

\[
\begin{align*}
\{3, 7\} & \quad \{4\}
\end{align*}
\]
Example

\textbf{MAKESET}(x) - make a new set containing only }$x$ (which is not already in a set)

\textbf{UNION}(x, y) - merge the sets containing }$x$ and }$y$ into a single set

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing }$x$

\textit{Let }$n = 16$ \textit{so that the elements of the sets are the numbers }$\{1, 2, \ldots, 16\}$

\textbf{MAKESET}(5)

\{3, 7\} \quad \{4\}
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the identity of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{MAKESET}(5)

\{5\} \quad \{3, 7\} \quad \{4\}
Example

**MAKESET**\((x)\) - make a new set containing only \(x\) (which is not already in a set)

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

**FINDSET**\((x)\) - returns the *identity* of the set containing \(x\)

---

*Let* \(n = 16\) *so that the elements of the sets are the numbers* \(\{1, 2, \ldots, 16\}\)

**MAKESET**\((5)\) \hspace{1cm} **MAKESET**\((9)\)

\[
\{5\} \hspace{2cm} \{3, 7\} \hspace{2cm} \{4\}
\]
Example

**MAKESET**(x) - make a new set containing only x (which is not already in a set)

**UNION**(x, y) - merge the sets containing x and y into a single set

**FINDSET**(x) - returns the identity of the set containing x

Let \( n = 16 \) so that the elements of the sets are the numbers \( \{1, 2, \ldots, 16\} \)

\[
\text{MAKESET}(5) \quad \text{MAKESET}(9)
\]

\[
\{5\} \quad \{9\} \quad \{3, 7\} \quad \{4\}
\]
Example

\textbf{MAKESET}(x) - make a new set containing only $x$ (which is not already in a set)
\textbf{UNION}(x, y) - merge the sets containing $x$ and $y$ into a single set
\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing $x$

\underline{Let $n = 16$ so that the elements of the sets are the numbers $\{1, 2, \ldots, 16\}$}

\textbf{MAKESET}(5) \quad \textbf{MAKESET}(9) \quad \textbf{MAKESET}(2)

$\{5\} \quad \{9\} \quad \{3, 7\} \quad \{4\}$
Example

\textbf{MAKESET}(x) - make a new set containing only \(x\) (which is not already in a set)
\textbf{UNION}(x, y) - merge the sets containing \(x\) and \(y\) into a single set
\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing \(x\)

\textit{Let } \(n = 16\) \textit{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{MAKESET}(5) \quad \textbf{MAKESET}(9) \quad \textbf{MAKESET}(2)

\{5\} \quad \{9\} \quad \{3, 7\} \quad \{2\} \quad \{4\}
Example

\textbf{MAKESET}(x) - make a new set containing only $x$ (which is not already in a set)

\textbf{UNION}(x, y) - merge the sets containing $x$ and $y$ into a single set

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing $x$

\textit{Let $n = 16$ so that the elements of the sets are the numbers $\{1, 2, \ldots, 16\}$}

\textbf{MAKESET}(5)  \textbf{MAKESET}(9)  \textbf{MAKESET}(2)  \textbf{MAKESET}(11)

\{5\}  \{9\}  \{3, 7\}  \{2\}  \{4\}
Example

\textsc{makeSet}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textsc{union}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textsc{findSet}(x) - returns the } identity \text{ of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textsc{makeSet}(5) \quad \textsc{makeSet}(9) \quad \textsc{makeSet}(2) \quad \textsc{makeSet}(11)

\{5\} \quad \{9\} \quad \{3, 7\} \quad \{2\} \quad \{4\} \quad \{11\}
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ } (which is not already in a set)

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the } identity \text{ of the set containing } x

Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{MAKESET}(5) \textbf{ MAKESET}(9) \textbf{ MAKESET}(2) \textbf{ MAKESET}(11) \textbf{ MAKESET}(16)

\{5\} \textbf{ } \{9\} \textbf{ } \{3, 7\} \textbf{ } \{2\} \textbf{ } \{4\} \textbf{ } \{11\}
Example

\texttt{MAKESET}(x) - make a new set containing only } x \texttt{ (which is not already in a set)

\texttt{UNION}(x, y) - merge the sets containing } x \texttt{ and } y \texttt{ into a single set

\texttt{FINDSET}(x) - returns the \emph{identity} of the set containing } x

\textit{Let } n = 16 \textit{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\texttt{MAKESET}(5) \texttt{ MAKESET}(9) \texttt{ MAKESET}(2) \texttt{ MAKESET}(11) \texttt{ MAKESET}(16)

\{5\} \quad \{9\} \quad \{3, 7\} \quad \{2\} \quad \{4\} \quad \{11\} \quad \{16\}
Example

\textbf{MAKESET}(x) - make a new set containing only $x$ (which is not already in a set)

\textbf{UNION}(x, y) - merge the sets containing $x$ and $y$ into a single set

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing $x$

\textit{Let } $n = 16 \textit{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}$

\begin{align*}
\{5\} & \quad \{9\} & \quad \{3, 7\} & \quad \{2\} & \quad \{4\} & \quad \{11\} & \quad \{16\}
\end{align*}
Example

**MAKESET**($x$) - make a new set containing only $x$ (which is not already in a set)

**UNION**(x, y) - merge the sets containing x and y into a single set

**FINDSET**($x$) - returns the *identity* of the set containing x

*Let $n = 16$ so that the elements of the sets are the numbers $\{1, 2, \ldots, 16\}$*

**UNION**(4, 9)

\[
\begin{align*}
\{5\} & \quad \{9\} & \quad \{3, 7\} & \quad \{2\} & \quad \{4\} & \quad \{11\} & \quad \{16\}
\end{align*}
\]
Example

**MAKESET**(x) - make a new set containing only x (which is not already in a set)

**UNION**(x, y) - merge the sets containing x and y into a single set

**FINDSET**(x) - returns the identity of the set containing x

Let \( n = 16 \) so that the elements of the sets are the numbers \{1, 2, \ldots, 16\}

\[ \text{UNION}(4, 9) \]

\[
\begin{align*}
\{5\} & \quad \{9\} & \quad \{3, 7\} & \quad \{2\} & \quad \{4\} & \quad \{11\} & \quad \{16\} \\
\text{merge these} & & & & & &
\end{align*}
\]
Example

\textbf{MAKESET}(x) - make a new set containing only } x (which is not already in a set)
\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}
\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{UNION}(4, 9)

\{5\} \quad \{4, 9\} \quad \{3, 7\} \quad \{2\} \quad \{11\} \quad \{16\}
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\{5\} \quad \{4, 9\} \quad \{3, 7\} \quad \{2\} \quad \{11\} \quad \{16\}
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{UNION}(2, 16)

\{5\} \quad \{4, 9\} \quad \{3, 7\} \quad \{2\} \quad \{11\} \quad \{16\}
Example

**MAKESET**($x$) - make a new set containing only $x$ (which is not already in a set)

**UNION**($x$, $y$) - merge the sets containing $x$ and $y$ into a single set

**FINDSET**($x$) - returns the identity of the set containing $x$

Let $n = 16$ so that the elements of the sets are the numbers \{1, 2, \ldots, 16\}

**UNION**($2$, $16$)

\[
\{5\} \quad \{4, 9\} \quad \{3, 7\} \quad \{2\} \quad \{11\} \quad \{16\}
\]

merge these
Example

\textbf{MAKE\textsc{Set}(x)} - make a new set containing only $x$ (which is not already in a set)

\textbf{UNION(x, y)} - merge the sets containing $x$ and $y$ into a single set

\textbf{FIND\textsc{Set}(x)} - returns the \textit{identity} of the set containing $x$

\textit{Let} $n = 16$ \textit{so that the elements of the sets are the numbers} \{1, 2, \ldots, 16\}

\textbf{UNION(2, 16)}

\{5\} \quad \{4, 9\} \quad \{3, 7\} \quad \{2, 16\} \quad \{11\}
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}
\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}
\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \textit{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\{5\} \quad \{4, 9\} \quad \{3, 7\} \quad \{2, 16\} \quad \{11\}
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{UNION}(7, 2)

\{5\} \quad \{4, 9\} \quad \{3, 7\} \quad \{2, 16\} \quad \{11\}
Example

**MAKESET**(\(x\)) - make a new set containing only \(x\) (which is not already in a set)

**UNION**(\(x, y\)) - merge the sets containing \(x\) and \(y\) into a single set

**FINDSET**(\(x\)) - returns the *identity* of the set containing \(x\)

*Let* \(n = 16\) *so that the elements of the sets are the numbers* \{1, 2, \ldots, 16\}

\[
\text{UNION}(7, 2)
\]

\{5\} \quad \{4, 9\} \quad \{3, 7\} \quad \{2, 16\} \quad \{11\}

merge these
Example

**MAKESET**(\(x\)) - make a new set containing only \(x\) (which is not already in a set)

**UNION**(\(x, y\)) - merge the sets containing \(x\) and \(y\) into a single set

**FINDSET**(\(x\)) - returns the *identity* of the set containing \(x\)

---

Let \(n = 16\) so that the elements of the sets are the numbers \(\{1, 2, \ldots, 16\}\)

\[\text{UNION}(7, 2)\]

\[
\begin{align*}
\{5\} & \quad \{4, 9\} & \quad \{2, 3, 7, 16\} & \quad \{11\}
\end{align*}
\]
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\{5\} \quad \{4, 9\} \quad \{2, 3, 7, 16\} \quad \{11\}
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the } identity \text{ of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots , 16\}

\textbf{UNION}(3, 5)

\{5\} \quad \{4, 9\} \quad \{2, 3, 7, 16\} \quad \{11\}
Example

**MAKESET**\((x)\) - make a new set containing only \(x\) (which is not already in a set)

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

**FINDSET**\((x)\) - returns the *identity* of the set containing \(x\)

Let \(n = 16\) so that the elements of the sets are the numbers \(\{1, 2, \ldots, 16\}\)

**UNION**\((3, 5)\)

\[
\{5\} \quad \{4, 9\} \quad \{2, 3, 7, 16\} \quad \{11\}
\]

merge these
Example

\text{MAKESET}(x)\ - \text{make a new set containing only } x \text{ (which is not already in a set)}

\text{UNION}(x, y)\ - \text{merge the sets containing } x \text{ and } y \text{ into a single set}

\text{FINDSET}(x)\ - \text{returns the identity of the set containing } x

\text{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\text{UNION}(3, 5)

\{4, 9\} \quad \{2, 3, 5, 7, 16\} \quad \{11\}
Example

\textbf{MAKESET}(x) - make a new set containing only \textit{x} (which is not already in a set)

\textbf{UNION}(x, y) - merge the sets containing \textit{x} and \textit{y} into a single set

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing \textit{x}

Let \( n = 16 \) so that the elements of the sets are the numbers \{1, 2, \ldots, 16\}

\{4, 9\} \quad \{2, 3, 5, 7, 16\} \quad \{11\}
Example

\text{MAKESET}(x)\ - \text{ make a new set containing only } x \ (\text{which is not already in a set})

\text{UNION}(x, y)\ - \text{ merge the sets containing } x \text{ and } y \text{ into a single set}

\text{FINDSET}(x)\ - \text{ returns the } \text{identity} \text{ of the set containing } x

Let \( n = 16 \) so that the elements of the sets are the numbers \( \{1, 2, \ldots, 16\} \)

\text{FINDSET}(2) \text{ returns } 3

\{4, 9\} \quad \{2, 3, 5, 7, 16\} \quad \{11\}
Example

**MAKESET(***x*** )** - make a new set containing only ***x*** (which is not already in a set)

**UNION(***x*** , ***y*** )** - merge the sets containing ***x*** and ***y*** into a single set

**FINDSET(***x*** )** - returns the *identity* of the set containing ***x***

Let $n = 16$ so that the elements of the sets are the numbers $\{1, 2, \ldots, 16\}$

- ***FINDSET(2)*** returns 3
- ***FINDSET(5)*** returns 3

$\{4, 9\} \quad \{2, 3, 5, 7, 16\} \quad \{11\}$
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

---

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textbf{FINDSET}(2) \text{ returns } 3

\textbf{FINDSET}(5) \text{ returns } 3

\textbf{FINDSET}(16) \text{ returns } 3

\{4, 9\} \quad \{2, 3, 5, 7, 16\} \quad \{11\}
Example

\textsc{MakeSet}(x) \ - \ make \ a \ new \ set \ containing \ only \ x \ (which \ is \ not \ already \ in \ a \ set)\\
\textsc{Union}(x, y) \ - \ merge \ the \ sets \ containing \ x \ and \ y \ into \ a \ single \ set\\
\textsc{FindSet}(x) \ - \ returns \ the \ identity \ of \ the \ set \ containing \ x\\

Let \( n = 16 \) so that the elements of the sets are the numbers \{1, 2, \ldots, 16\}\\

\textsc{FindSet}(2) \ returns \ 3\\
\textsc{FindSet}(5) \ returns \ 3\\
\textsc{FindSet}(16) \ returns \ 3\\

\{4, 9\} \quad \{2, 3, 5, 7, 16\} \quad \{11\}\\

\textsc{FindSet}(4) \ returns \ 9
Example

**MAKESET**(*x*) - make a new set containing only *x* (which is not already in a set)

**UNION**(*x*, *y*) - merge the sets containing *x* and *y* into a single set

**FINDSET**(*x*) - returns the *identity* of the set containing *x*

Let \( n = 16 \) so that the elements of the sets are the numbers \{1, 2, \ldots, 16\}

**FINDSET**(2) returns 3

**FINDSET**(5) returns 3

**FINDSET**(16) returns 3

\{4, 9\} \{2, 3, 5, 7, 16\} \{11\}

**FINDSET**(4) returns 9

**FINDSET**(9) returns 9
Example

\textbf{MAKESET}(x) - make a new set containing only } x \text{ (which is not already in a set)}

\textbf{UNION}(x, y) - merge the sets containing } x \text{ and } y \text{ into a single set}

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing } x

\textit{Let } n = 16 \text{ so that the elements of the sets are the numbers } \{1, 2, \ldots, 16\}

\textit{FINDSET}(2) \text{ returns 3}

\textit{FINDSET}(5) \text{ returns 3}

\textit{FINDSET}(16) \text{ returns 3}

\{4, 9\} \quad \{2, 3, 5, 7, 16\} \quad \{11\}

\textit{FINDSET}(4) \text{ returns 9}

\textit{FINDSET}(9) \text{ returns 9}

\textit{In our data structure, the identity will be an element of the set}
Reverse Trees

The data structure we will discuss stores each set as a reverse tree:
Reverse Trees

The data structure we will discuss stores each set as a reverse tree:

```
<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
```

This reverse tree stores the set \(\{1, 3, 4, 5, 8, 9, 12\}\)
Reverse Trees

The data structure we will discuss stores each set as a reverse tree:

This reverse tree stores the set \( \{1, 3, 4, 5, 8, 9, 12\} \)

Each node stores an element from the set
Reverse Trees

The data structure we will discuss stores each set as a reverse tree:

This reverse tree stores the set \( \{1, 3, 4, 5, 8, 9, 12\} \)

Each node stores an element from the set

The identity of a set is element at the root (here 3)
Reverse Trees

The data structure we will discuss stores each set as a reverse tree:

![Reverse Tree Diagram]

This reverse tree stores the set \( \{1, 3, 4, 5, 8, 9, 12\} \)

Each node stores an element from the set

The identity of a set is element at the root (here 3)
The data structure we will discuss stores each set as a reverse tree:

```
3
/   \
1  8  9
   /  \\  /
  4  5  12
```

This reverse tree stores the set \{1, 3, 4, 5, 8, 9, 12\}

Each node stores an element from the set

The identity of a set is element at the root (here 3)
Reverse Trees

The data structure we will discuss stores each set as a reverse tree:

This reverse tree stores the set \( \{1, 3, 4, 5, 8, 9, 12\} \)

Each node stores an element from the set

The identity of a set is element at the root (here 3)

In a reverse tree, each element stores a pointer to its parent but no pointers to its children
Reverse Trees

The data structure we will discuss stores each set as a reverse tree:

![Reverse Tree Diagram]

This reverse tree stores the set \(\{1, 3, 4, 5, 8, 9, 12\}\)

Each node stores an element from the set

The **identity** of a set is element at the root (here 3)

In a reverse tree, each element stores a pointer to its parent

- **no limit on the number of children each node can have**
Reverse Forests

The data structure consists of a forest of reverse trees, one for each set

Each node stores an element from the set

The identity of a set is element at the root
How are these trees stored?

{1, 3, 4, 5, 8, 9, 12}

{2, 7}

{15}

{6, 10, 14}
How are these trees stored?

The elements are stored in an array of length \( n \):
How are these trees stored?

The elements are stored in an array of length $n$:
How are these trees stored?

The elements are stored in an array of length $n$:
How are these trees stored?

The elements are stored in an array of length $n$:
How are these trees stored?

The elements are stored in an array of length $n$:
How are these trees stored?

The elements are stored in an array of length $n$: 

$A = \{1, 3, 4, 5, 8, 9, 12\}$  
$\{2, 7\}$  
$\{15\}$  
$\{6, 10, 14\}$
How are these trees stored?

The elements are stored in an array of length \( n \):
How are these trees stored?

The elements are stored in an array of length \( n \):
How are these trees stored?

{1, 3, 4, 5, 8, 9, 12}  {2, 7}  {15}  {6, 10, 14}

The elements are stored in an array of length $n$:

$A_{12} \rightarrow A_{2} \rightarrow A_{15} \rightarrow A_{14} \rightarrow A_{10}$
How are these trees stored?

The elements are stored in an array of length $n$:

This allows us to find any element $x$ in $O(1)$ time ($x$ is stored in $A[x]$)
The FINDSET operation

\[ \text{FINDSET}(x) \] - returns the identity of the set containing \( x \)

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
**The FINDSET operation**

**FINDSET**(x) - returns the *identity* of the set containing x

**Step 1:** Find the node storing element x

**Step 2:** Until you are at the root,
follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The \text{FINDSET} operation

\text{FINDSET}(x) - returns the \textit{identity} of the set containing \(x\)

\begin{itemize}
  \item \textbf{Step 1:} Find the node storing element \(x\)
  \item \textbf{Step 2:} Until you are at the root, follow the pointer to the parent of the current node
  \item \textbf{Step 3:} Output the element at the root
\end{itemize}
The **FINDSET** operation

**FINDSET**(*x*) - returns the *identity* of the set containing *x*

---

**Step 1:** Find the node storing element *x*

**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The `FINDSET` operation

\[ \text{FINDSET}(x) \] - returns the identity of the set containing \( x \)

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The \textbf{FINDSET} operation

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing \(x\)

\textbf{Step 1:} Find the node storing element \(x\)

\textbf{Step 2:} Until you are at the root, follow the pointer to the parent of the current node

\textbf{Step 3:} Output the element at the root
The **FINDSET** operation

\[ \text{FINDSET}(x) \] returns the *identity* of the set containing \( x \)

---

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The FINDSET operation

\( \text{FINDSET}(x) \) - returns the *identity* of the set containing \( x \)

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The **FINDSET** operation

\[ \text{FINDSET}(x) \] - returns the *identity* of the set containing \( x \)

\[
\begin{align*}
\text{FINDSET}(5) & \text{ returns 3} \\
\{1, 3, 4, 5, 8, 9, 12\} & \{2, 7\} & \{15\} & \{6, 10, 14\}
\end{align*}
\]

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The FINDSET operation

\[
\text{FINDSET}(x) \text{ - returns the identity of the set containing } x
\]

Step 1: Find the node storing element \( x \)

Step 2: Until you are at the root, follow the pointer to the parent of the current node

Step 3: Output the element at the root
The **FINDSET** operation

**FINDSET**\( (x) \) - returns the *identity* of the set containing \( x \)

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The \textsc{FindSet} operation

\textbf{FindSet}(x) - returns the identity of the set containing \(x\)

\begin{figure}
\centering
\includegraphics[width=\textwidth]{findset_diagram}
\caption{FindSet(1)}
\end{figure}

\begin{itemize}
\item \textbf{Step 1}: Find the node storing element \(x\)
\item \textbf{Step 2}: Until you are at the root,
\begin{itemize}
\item follow the pointer to the parent of the current node
\end{itemize}
\item \textbf{Step 3}: Output the element at the root
\end{itemize}
The FINDSET operation

$\text{FINDSET}(x)$ - returns the identity of the set containing $x$

**Step 1:** Find the node storing element $x$

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The FINDSET operation

\[ \text{FINDSET}(x) \] - returns the \textit{identity} of the set containing \( x \)

\textbf{Step 1:} Find the node storing element \( x \)

\textbf{Step 2:} Until you are at the root, follow the pointer to the parent of the current node

\textbf{Step 3:} Output the element at the root
The \textsc{FindSet} operation

\textsc{FindSet}(x) - returns the \textit{identity} of the set containing \( x \)

\textbf{Step 1:} Find the node storing element \( x \)

\textbf{Step 2:} Until you are at the root,

\hspace{1cm} follow the pointer to the parent of the current node

\textbf{Step 3:} Output the element at the root
The **FINDSET** operation

\[ \text{FINDSET}(x) \] - returns the *identity* of the set containing \( x \)

---

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

**Step 3:** Output the element at the root

---

\( \{1, 3, 4, 5, 8, 9, 12\} \)

\( \{2, 7\} \)

\( \{15\} \)

\( \{6, 10, 14\} \)
The FINDSET operation

\text{FINDSET}(x) - \text{returns the identity of the set containing } x

\begin{itemize}
  \item \textbf{Step 1:} Find the node storing element $x$
  \item \textbf{Step 2:} Until you are at the root,
    follow the pointer to the parent of the current node
  \item \textbf{Step 3:} Output the element at the root
\end{itemize}
The **FINDSET** operation

**FINDSET**(\(x\)) - returns the *identity* of the set containing \(x\)

**Step 1:** Find the node storing element \(x\)

**Step 2:** Until you are at the root,
follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The **FINDSET** operation

**FINDSET**(\(x\)) - returns the *identity* of the set containing \(x\)

**Step 1:** Find the node storing element \(x\)

**Step 2:** Until you are at the root,
follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The **FINDSET** operation

**FINDSET**(*x*) - returns the *identity* of the set containing *x*

**Step 1:** Find the node storing element *x*

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root

**FINDSET**(*15*) returns **15**
The **FINDSET** operation

**FINDSET**(*x*) - returns the *identity* of the set containing *x*

**Step 1:** Find the node storing element *x*

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root
The \textbf{FINDSET} operation

\textbf{FINDSET}(x) - returns the \textit{identity} of the set containing \textit{x}

\begin{itemize}
  \item \textbf{Step 1:} Find the node storing element \textit{x}
  \item \textbf{Step 2:} Until you are at the root, follow the pointer to the parent of the current node
  \item \textbf{Step 3:} Output the element at the root
\end{itemize}

What is the worst-case time complexity of this operation?
The FINDSET operation

\text{FINDSET}(x) - \text{returns the identity of the set containing } x

\begin{itemize}
  \item \textbf{Step 1:} Find the node storing element \textit{x}
  \item \textbf{Step 2:} Until you are at the root, follow the pointer to the parent of the current node
  \item \textbf{Step 3:} Output the element at the root
\end{itemize}

What is the worst-case time complexity of this operation?
The FINDSET operation

\( \text{FINDSET}(x) \) - returns the \textit{identity} of the set containing \( x \)

\[ \{1, 3, 4, 5, 8, 9, 12\} \quad \{2, 7\} \quad \{15\} \quad \{6, 10, 14\} \]

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root

What is the worst-case time complexity of this operation?
The FINDSET operation

FINDSET\((x)\) - returns the identity of the set containing \(x\)

Step 1: Find the node storing element \(x\)  
\(O(1)\) time because \(x\) is stored in \(A[x]\)

Step 2: Until you are at the root, follow the pointer to the parent of the current node  
\(O(h)\) time

Step 3: Output the element at the root

What is the worst-case time complexity of this operation?
The **FINDSET** operation

**FINDSET**\( (x) \) - returns the *identity* of the set containing \( x \)

---

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root

What is the worst-case time complexity of this operation?

\[ O(1) \text{ time because } x \text{ is stored in } A[x] \]

\[ O(h) \text{ time} \]

\( h \) is the height of the tallest tree
The FINDSET operation

\[ \text{FINDSET}(x) \] - returns the *identity* of the set containing \( x \)

**Step 1:** Find the node storing element \( x \)

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root

What is the worst-case time complexity of this operation?
The **FINDSET** operation

**FINDSET**$(x)$ - returns the *identity* of the set containing $x$

---

**Step 1:** Find the node storing element $x$

**Step 2:** Until you are at the root, follow the pointer to the parent of the current node

**Step 3:** Output the element at the root

The overall worst-case time complexity is $O(h)$
The FINDSET operation

\( \text{FINDSET}(x) \) - returns the identity of the set containing \( x \)

Step 1: Find the node storing element \( x \)

Step 2: Until you are at the root, follow the pointer to the parent of the current node

Step 3: Output the element at the root

The overall worst-case time complexity is \( O(h) \)
The **MAKESET** operation

**MAKESET**($x$) - make a new set containing only $x$ (which is not already in a set)

Step 1: Make a new tree containing $x$ as the root
The MakeSet operation

\text{MakeSet}(x) - \text{make a new set containing only } x \text{ (which is not already in a set)}

\begin{itemize}
  \item \{1, 3, 4, 5, 8, 9, 12\}
  \item \{2, 7\}
  \item \{15\}
  \item \{6, 10, 14\}
\end{itemize}

\textbf{Step 1:} Make a new tree containing } x \text{ as the root
The \texttt{MAKESET} operation

\texttt{MAKESET}(x) - make a new set containing only \textit{x} (which is not already in a set)

\begin{itemize}
  \item \texttt{MAKESET}(11)
\end{itemize}

\begin{itemize}
  \item \texttt{MAKESET}(11) \rightarrow \{11\}
\end{itemize}

\begin{itemize}
  \item \texttt{MAKESET}(11) \rightarrow \{2, 7\} \texttt{MAKESET}(11) \rightarrow \{11\} \texttt{MAKESET}(11) \rightarrow \{15\} \texttt{MAKESET}(11) \rightarrow \{6, 10, 14\}
\end{itemize}

\textbf{Step 1:} Make a new tree containing \textit{x} as the root
The `MAKESET` operation

```
MAKESET(x) - make a new set containing only x (which is not already in a set)
```

**Step 1:** Make a new tree containing \( x \) as the root

(\textit{that's it})
The **MAKESET** operation

**MAKESET**(*x*) - make a new set containing only *x* (which is not already in a set)

Step 1: Make a new tree containing *x* as the root

*(that's it)*

What is the worst-case time complexity of this operation?
The **MAKESET** operation

**MAKESET** \((x)\) - make a new set containing only \(x\) (which is not already in a set)

![Tree diagram]

\{1, 3, 4, 5, 8, 9, 12\}  
\{2, 7\}  
\{11\}  
\{15\}  
\{6, 10, 14\}

\(O(1)\) time  
**Step 1:** Make a new tree containing \(x\) as the root  
(that's it)

*because \(x\) should be stored in \(A[x]\)*

**What is the worst-case time complexity of this operation?**
The \textsc{MakeSet} operation

\textsc{MakeSet}(x) - make a new set containing only $x$ (which is not already in a set)

\begin{itemize}
  \item \textsc{Step 1}: Make a new tree containing $x$ as the root
  \item \texttt{O(1)} time (that's it)
\end{itemize}

because $x$ should be stored in $A[x]$

\begin{itemize}
  \item \textit{What is the worst-case time complexity of this operation?}
  \item The overall worst-case time complexity is \texttt{O(1)}
\end{itemize}
The MAKE\textsc{Set} operation

\textbf{MAKE\textsc{Set}(x)} - make a new set containing only $x$ (which is not already in a set)

\begin{itemize}
  \item \{1, 3, 4, 5, 8, 9, 12\}
  \item \{2, 7\} \{11\} \{15\} \{6, 10, 14\}
\end{itemize}

\textbf{Step 1:} Make a new tree containing $x$ as the root (that's it)

because $x$ should be stored in $A[x]$

What is the worst-case time complexity of this operation?
The **UNION** operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

\[
\{1, 3, 4, 5, 8, 9, 12\} \quad \{2, 7\} \quad \{15\} \quad \{6, 10, 14\}
\]

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** Make \(r_x\) a child of \(r_y\) (*which merges the two trees*)
The **UNION** operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

![Tree Diagram]

Step 1: Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

Step 2: Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

Step 3: Make \(r_x\) a child of \(r_y\) *(which merges the two trees)*
The **UNION** operation

**UNION**($x, y$) - merge the sets containing $x$ and $y$ into a single set

---

**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**Step 3:** Make $r_x$ a child of $r_y$ (*which merges the two trees*)
The UNION operation

**UNION**(*x, y*) - merge the sets containing *x* and *y* into a single set

**Step 1:** Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing *x*

**Step 2:** Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing *y*

**Step 3:** Make \( r_x \) a child of \( r_y \) *(which merges the two trees)*
The \textit{UNION} operation

\textbf{UNION}(x, y) - merge the sets containing \textit{x} and \textit{y} into a single set

\begin{itemize}
  \item \textbf{Step 1:} Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing \textit{x}
  \item \textbf{Step 2:} Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing \textit{y}
  \item \textbf{Step 3:} Make \( r_x \) a child of \( r_y \) (\textit{which merges the two trees})
\end{itemize}
The **UNION** operation

**UNION** \((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

\[
\{1, 3, 4, 5, 8, 9, 12\}
\]

\[
\{2, 7\}
\]

\[
\{15\}
\]

\[
\{6, 10, 14\}
\]

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** Make \(r_x\) a child of \(r_y\) (*which merges the two trees*)
The UNION operation

**UNION**($x, y$) - merge the sets containing $x$ and $y$ into a single set

**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**Step 3:** Make $r_x$ a child of $r_y$ (which merges the two trees)
The **UNION** operation

**UNION**(*x, y*) - merge the sets containing *x* and *y* into a single set

**Step 1:** Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing *x*

**Step 2:** Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing *y*

**Step 3:** Make \( r_x \) a child of \( r_y \) (which merges the two trees)
The **UNION** operation

**UNION**($x, y$) - merge the sets containing $x$ and $y$ into a single set

Step 1: Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

Step 2: Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

Step 3: Make $r_x$ a child of $r_y$ (*which merges the two trees*)
The \textbf{UNION} operation

\textbf{UNION}(x, y) - merge the sets containing \textit{x} and \textit{y} into a single set

\begin{itemize}
  \item \textbf{Step 1:} Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing \textit{x}
  \item \textbf{Step 2:} Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing \textit{y}
  \item \textbf{Step 3:} Make $r_x$ a child of $r_y$ (\textit{which merges the two trees})
\end{itemize}
The **UNION** operation

**UNION**(x, y) - merge the sets containing x and y into a single set

**Step 1:** Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing x

**Step 2:** Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing y

**Step 3:** Make \( r_x \) a child of \( r_y \) (which merges the two trees)
The **UNION** operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

\[
\{1, 3, 4, 5, 8, 9, 12\}
\]

\[
\{2, 7, 15\}
\]

\[
\{6, 10, 14\}
\]

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** Make \(r_x\) a child of \(r_y\) (which merges the two trees)
The UNION operation

**UNION**(x, y) - merge the sets containing x and y into a single set

Step 1: Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing x

Step 2: Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing y

Step 3: Make \( r_x \) a child of \( r_y \) (which merges the two trees)
The UNION operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** Make \(r_x\) a child of \(r_y\) (which merges the two trees)
The **UNION** operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

Step 1: Compute \(r_x = FINDSET(x)\) - the root of the tree containing \(x\)

Step 2: Compute \(r_y = FINDSET(y)\) - the root of the tree containing \(y\)

Step 3: Make \(r_x\) a child of \(r_y\) (which merges the two trees)

*What is the worst-case time complexity of this operation?*
The **UNION** operation

**UNION**(x, y) - merge the sets containing x and y into a single set

Step 1: Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing x

Step 2: Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing y

Step 3: Make \( r_x \) a child of \( r_y \) *(which merges the two trees)*

What is the worst-case time complexity of this operation?
The UNION operation

**UNION**(x, y) - merge the sets containing x and y into a single set

Step 1: Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing x

Step 2: Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing y

Step 3: Make \( r_x \) a child of \( r_y \) (which merges the two trees)

What is the worst-case time complexity of this operation?

\( O(h) \) time
The **UNION** operation

**UNION**(x, y) - merge the sets containing x and y into a single set

---

![Diagram showing trees and sets

1. **Step 1**: Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing x
2. **Step 2**: Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing y
3. **Step 3**: Make \( r_x \) a child of \( r_y \) (which merges the two trees)

What is the worst-case time complexity of this operation?
The UNION operation

\textbf{UNION}(x, y) - merge the sets containing \textit{x} and \textit{y} into a single set

\begin{itemize}
  \item \textbf{Step 1}: Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \textit{x}
  \item \textbf{Step 2}: Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \textit{y}
  \item \textbf{Step 3}: Make \(r_x\) a child of \(r_y\) (which merges the two trees)
\end{itemize}

What is the worst-case time complexity of this operation?

It's \(O(h)\) again.
How high does the sycamore grow?

Unfortunately, every **UNION** operation could increase $h$ by one…

Consider the following sets:

{1}  {2}  {3}  {4}  {5}  …
How high does the sycamore grow?

Unfortunately, every UNION operation could increase $h$ by one...

Consider the following sets:

Now perform UNION(1, 2)
How high does the sycamore grow?

Unfortunately, every **UNION** operation could increase \( h \) by one...

Consider the following sets:

Now perform **UNION(1, 2)**
How high does the sycamore grow?

Unfortunately, every \texttt{UNION} operation could increase $h$ by one…

Consider the following sets:
How high does the sycamore grow?

Unfortunately, every \textsc{union} operation could increase $h$ by one…

Consider the following sets:

Now perform \textsc{union}(1, 3)
How high does the sycamore grow?

Unfortunately, every UNION operation could increase $h$ by one...

Consider the following sets:

Now perform $\text{UNION}(1, 3)$
How high does the sycamore grow?

Unfortunately, every \texttt{UNION} operation could increase $h$ by one...

Consider the following sets:
How high does the sycamore grow?

Unfortunately, every \texttt{UNION} operation could increase $h$ by one...

Consider the following sets:

\begin{align*}
{1, 2, 3} & \quad {4} & \quad {5} \\
\end{align*}

Now perform \texttt{UNION}(1, 4)
How high does the sycamore grow?

Unfortunately, every **UNION** operation could increase $h$ by one…

Consider the following sets:

$$\{1, 2, 3, 4\} \quad \{5\}$$

Now perform **UNION**$(1, 4)$
How high does the sycamore grow?

Unfortunately, every **UNION** operation could increase $h$ by one...

Consider the following sets:

\{1, 2, 3, 4\} \quad \{5\}
How high does the sycamore grow?

Unfortunately, every \texttt{UNION} operation could increase $h$ by one...

Consider the following sets:

\begin{equation*}
\{1, 2, 3, 4\} \quad \{5\}
\end{equation*}

Now perform \texttt{UNION}(1, 5)
How high does the sycamore grow?

Unfortunately, every \texttt{UNION} operation could increase $h$ by one...

Consider the following sets:

$$\{1, 2, 3, 4, 5\}$$

Now perform \texttt{UNION}(1, 5)
How high does the sycamore grow?

Unfortunately, every \texttt{UNION} operation could increase \( h \) by one…

Consider the following sets:

\[
\{1, 2, 3, 4, 5, \ldots\}
\]

Now perform \texttt{UNION}(1, 5)…

So in the worst case the height of the tallest tree is \( n \)
How high does the sycamore grow?

Unfortunately, every \textit{UNION} operation could increase $h$ by one...

Consider the following sets:

$$\{1, 2, 3, 4, 5, \ldots\}$$
How high does the sycamore grow?

Unfortunately, every UNION operation could increase $h$ by one…

Consider the following sets:

$$\{1, 2, 3, 4, 5, \ldots\}$$

In the worst case the height of the tallest tree is $n$. 
How high does the sycamore grow?

Unfortunately, every UNION operation could increase $h$ by one...

Consider the following sets:

$\{1, 2, 3, 4, 5, \ldots\}$

In the worst case the height of the tallest tree is $n$

so UNION and FIND run in $O(n)$ time
How high does the sycamore grow?

Unfortunately, every \texttt{UNION} operation could increase $h$ by one…

Consider the following sets:

\{1, 2, 3, 4, 5, \ldots\}

Can we improve this?

In the worst case the height of the tallest tree is $n$ 
so \texttt{UNION} and \texttt{FIND} run in $O(n)$ time
What's bad about the UNION operation?

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

\[
\begin{array}{c}
1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 \\
\{1, 2, 3, 4\} & \rightarrow & 5 & \{5\}
\end{array}
\]

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** Make \(r_x\) a child of \(r_y\) (which merges the two trees)
What’s bad about the UNION operation?

**UNION**(*x, y*) - merge the sets containing *x* and *y* into a single set

Step 1: Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing *x*

Step 2: Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing *y*

Step 3: Make \( r_x \) a child of \( r_y \) *(which merges the two trees)*

When we performed **UNION**(1, 5), we made a \( r_x \) the child of \( r_y \)
What's bad about the UNION operation?

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

Step 1: Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

Step 2: Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

Step 3: Make \(r_x\) a child of \(r_y\) *(which merges the two trees)*

When we performed **UNION**(1, 5), we made a \(r_x\) the child of \(r_y\) this increases the height by one
What's bad about the UNION operation?

\textbf{UNION}(x, y) - merge the sets containing \textcolor{blue}{x} and \textcolor{red}{y} into a single set

\begin{itemize}
  \item \textcolor{blue}{r}_x \quad \textcolor{red}{r}_y
\end{itemize}

When we performed \textbf{UNION}(1, 5),
we made a \textcolor{blue}{r}_x the child of \textcolor{red}{r}_y
this increases the height by one

If instead we made \textcolor{red}{r}_y the child of \textcolor{blue}{r}_x…

\textbf{Step 1}: Compute \( \textcolor{blue}{r}_x = \textsc{FindSet}(x) \) - the root of the tree containing \textcolor{blue}{x}

\textbf{Step 2}: Compute \( \textcolor{red}{r}_y = \textsc{FindSet}(y) \) - the root of the tree containing \textcolor{red}{y}

\textbf{Step 3}: Make \textcolor{blue}{r}_x a child of \textcolor{red}{r}_y \textit{(which merges the two trees)}
What's bad about the UNION operation?

**UNION**($x$, $y$) - merge the sets containing $x$ and $y$ into a single set

Step 1: Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

Step 2: Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

Step 3: Make $r_x$ a child of $r_y$ *(which merges the two trees)*

When we performed **UNION**($1$, $5$), we made a $r_x$ the child of $r_y$
this increases the height by one

If instead we made $r_y$ the child of $r_x$...

\{1, 2, 3, 4, 5\}
What's bad about the UNION operation?

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

\[
\begin{align*}
    r_x & \rightarrow 4 \\
    r_y & \rightarrow 5
\end{align*}
\]

\{1, 2, 3, 4, 5\}

When we performed **UNION**(1, 5),
we made a \(r_x\) the child of \(r_y\)
this increases the height by one
If instead we made \(r_y\) the child of \(r_x\)...
the height is unchanged

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** Make \(r_x\) a child of \(r_y\) *(which merges the two trees)*
What's bad about the **UNION** operation?

**UNION**($x, y$) - merge the sets containing $x$ and $y$ into a single set

1. Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$
2. Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$
3. Make $r_x$ a child of $r_y$ (which merges the two trees)

When we performed **UNION**(1, 5),
we made a $r_x$ the child of $r_y$
this increases the height by one
If instead we made $r_y$ the child of $r_x$ . . .
the height is unchanged

How can we generalise this?
What’s bad about the \textsc{Union} operation?

\textsc{Union}(x, y) - merge the sets containing $x$ and $y$ into a single set

When we performed \textsc{Union}(1, 5),
we made a $r_x$ the child of $r_y$
this increases the height by one
If instead we made $r_y$ the child of $r_x$…
the height is unchanged

How can we generalise this?

\textbf{Step 1:} Compute $r_x = \text{FindSet}(x)$ - the root of the tree containing $x$

\textbf{Step 2:} Compute $r_y = \text{FindSet}(y)$ - the root of the tree containing $y$

\textbf{Step 3:} Make $r_x$ a child of $r_y$ \textit{(which merges the two trees)}

\textbf{Key Idea} always make the shorter tree the child of the taller tree
An improved UNION operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** If \(h(x) \leq h(y)\) make \(r_x\) a child of \(r_y\)

Else make \(r_y\) a child of \(r_x\)
An improved UNION operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

1. **Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)
2. **Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)
3. **Step 3:** If \(h(x) \leq h(y)\) make \(r_x\) a child of \(r_y\)
   - Else make \(r_y\) a child of \(r_x\)

Let \(h(x)\) be the height of the tree containing \(x\) (and \(h(y)\) for \(y\))
An improved UNION operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** If \(h(x) \leq h(y)\) make \(r_x\) a child of \(r_y\)

Else make \(r_y\) a child of \(r_x\)

Let \(h(x)\) be the height of the tree containing \(x\) (and \(h(y)\) for \(y\))
An improved UNION operation

$\text{UNION}(x, y)$ - merge the sets containing $x$ and $y$ into a single set

Step 1: Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

Step 2: Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

Step 3: If $h(x) \leq h(y)$ make $r_x$ a child of $r_y$

Else make $r_y$ a child of $r_x$
An improved UNION operation

\textsc{Union}(x, y) - merge the sets containing \(x\) and \(y\) into a single set

\begin{align*}
\text{Step 1:} & \quad \text{Compute } r_x = \text{FindSet}(x) - \text{the root of the tree containing } x \\
\text{Step 2:} & \quad \text{Compute } r_y = \text{FindSet}(y) - \text{the root of the tree containing } y \\
\text{Step 3:} & \quad \text{If } h(x) \leq h(y) \text{ make } r_x \text{ a child of } r_y \\
& \quad \text{Else make } r_y \text{ a child of } r_x
\end{align*}

Let \(h(x)\) be the height of the tree containing \(x\) (and \(h(y)\) for \(y\))

\[\text{Union}(8, 14)\]
An improved UNION operation

**UNION**(*x, y*) - merge the sets containing *x* and *y* into a single set

![Tree Diagram]

Let *h(x)* be the height of the tree containing *x* (and *h(y)* for *y*)

**Step 1:** Compute *r_x* = **FINDSET**(*x*) - the root of the tree containing *x*

**Step 2:** Compute *r_y* = **FINDSET**(*y*) - the root of the tree containing *y*

**Step 3:** If *h(x) ≤ h(y)* make *r_x* a child of *r_y*

Else make *r_y* a child of *r_x*
An improved UNION operation

**UNION**(*x, y*) - merge the sets containing *x* and *y* into a single set

Step 1: Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing *x*

Step 2: Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing *y*

Step 3: If \(h(x) \leq h(y)\) make \(r_x\) a child of \(r_y\)

Else make \(r_y\) a child of \(r_x\)

Let \(h(x)\) be the height of the tree containing *x* (and \(h(y)\) for *y*)
An improved UNION operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

Let \(h(x)\) be the height of the tree containing \(x\) (and \(h(y)\) for \(y\))

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** If \(h(x) \leq h(y)\) make \(r_x\) a child of \(r_y\)

Else make \(r_y\) a child of \(r_x\)
An improved UNION operation

**UNION**(\(x, y\)) - merge the sets containing \(x\) and \(y\) into a single set

\[
\begin{align*}
\text{Step 1:} & \quad \text{Compute } r_x = \text{FINDSET}(x) \quad \text{- the root of the tree containing } x \\
\text{Step 2:} & \quad \text{Compute } r_y = \text{FINDSET}(y) \quad \text{- the root of the tree containing } y \\
\text{Step 3:} & \quad \text{If } h(x) \leq h(y) \text{ make } r_x \text{ a child of } r_y \\
& \quad \text{Else make } r_y \text{ a child of } r_x
\end{align*}
\]

Let \(h(x)\) be the height of the tree containing \(x\) (and \(h(y)\) for \(y\))
An improved UNION operation

**UNION**\((x, y)\) - merge the sets containing \(x\) and \(y\) into a single set

Let \(h(x)\) be the height of the tree containing \(x\) (and \(h(y)\) for \(y\))

**Step 1:** Compute \(r_x = \text{FINDSET}(x)\) - the root of the tree containing \(x\)

**Step 2:** Compute \(r_y = \text{FINDSET}(y)\) - the root of the tree containing \(y\)

**Step 3:** If \(h(x) \leq h(y)\) make \(r_x\) a child of \(r_y\)

Else make \(r_y\) a child of \(r_x\)
An improved UNION operation

**UNION**\( (x, y) \) - merge the sets containing \( x \) and \( y \) into a single set

Step 1: Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing \( x \)

Step 2: Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing \( y \)

Step 3: If \( h(x) \leq h(y) \) make \( r_x \) a child of \( r_y \)

Else make \( r_y \) a child of \( r_x \)
An improved UNION operation

$\text{UNION}(x, y)$ - merge the sets containing $x$ and $y$ into a single set

**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leq h(y)$ make $r_x$ a child of $r_y$

Else make $r_y$ a child of $r_x$

Let $h(x)$ be the height of the tree containing $x$ (and $h(y)$ for $y$)
An improved UNION operation

\textbf{UNION}(x, y) - merge the sets containing \( x \) and \( y \) into a single set

\begin{itemize}
  \item \textbf{Step 1:} Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing \( x \)
  \item \textbf{Step 2:} Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing \( y \)
  \item \textbf{Step 3:} If \( h(x) \leq h(y) \) make \( r_x \) a child of \( r_y \)
    \hspace{1cm} Else make \( r_y \) a child of \( r_x \)
\end{itemize}

Let \( h(x) \) be the height of the tree containing \( x \) (and \( h(y) \) for \( y \))

\textbf{UNION}(8, 14)
An improved UNION operation

$\text{UNION}(x, y)$ - merge the sets containing $x$ and $y$ into a single set

**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leq h(y)$ make $r_x$ a child of $r_y$

Else make $r_y$ a child of $r_x$

Let $h(x)$ be the height of the tree containing $x$ (and $h(y)$ for $y$)
An improved **UNION** operation

**UNION**($x, y$) - merge the sets containing $x$ and $y$ into a single set

Let $h(x)$ be the height of the tree containing $x$ (and $h(y)$ for $y$)

**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leq h(y)$ make $r_x$ a child of $r_y$

 Else make $r_y$ a child of $r_x$
An improved UNION operation

**UNION**(x, y) - merge the sets containing x and y into a single set

Let \( h(x) \) be the height of the tree containing x (and \( h(y) \) for y)

**Step 1:** Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing x

**Step 2:** Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing y

**Step 3:** If \( h(x) \leq h(y) \) make \( r_x \) a child of \( r_y \)

Else make \( r_y \) a child of \( r_x \)
An improved `UNION` operation

`UNION(x, y)` - merge the sets containing `x` and `y` into a single set

Let `h(x)` be the height of the tree containing `x` (and `h(y)` for `y`)

**Step 1**: Compute `r_x = FINDSET(x)` - the root of the tree containing `x`

**Step 2**: Compute `r_y = FINDSET(y)` - the root of the tree containing `y`

**Step 3**: If `h(x) ≤ h(y)` make `r_x` a child of `r_y`

Else make `r_y` a child of `r_x`
An improved UNION operation

**UNION**($x, y$) - merge the sets containing $x$ and $y$ into a single set

#### Step 1:
Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

#### Step 2:
Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

#### Step 3:
If $h(x) \leq h(y)$ make $r_x$ a child of $r_y$
Else make $r_y$ a child of $r_x$

Let $h(x)$ be the height of the tree containing $x$ (and $h(y)$ for $y$)
An improved UNION operation

**UNION***(x, y) - merge the sets containing x and y into a single set

Let h(x) be the height of the tree containing x (and h(y) for y)

**Step 1:** Compute \( r_x = \text{FINDSET}(x) \) - the root of the tree containing x

**Step 2:** Compute \( r_y = \text{FINDSET}(y) \) - the root of the tree containing y

**Step 3:** If \( h(x) \leq h(y) \) make \( r_x \) a child of \( r_y \)

Else make \( r_y \) a child of \( r_x \)

This still takes \( O(h) \) time
An improved UNION operation

$\text{UNION}(x, y)$ - merge the sets containing $x$ and $y$ into a single set

Step 1: Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

Step 2: Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

Step 3: If $h(x) \leq h(y)$ make $r_x$ a child of $r_y$

Else make $r_y$ a child of $r_x$

This still takes $O(h)$ time . . . but the height only increases when $h(x) = h(y)$
Now big is $h$ now?
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

We begin by proving that any tree of height $h$ contains at least $2^h$ nodes
Now big is $h$ now?

Claim The height, $h$, of the tallest tree is $O(\log n)$

We begin by proving that any tree of height $h$ contains at least $2^h$ nodes

Proof by induction on tree height $i$, 
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

We begin by proving that any tree of height $h$ contains at least $2^h$ nodes

**Proof** by induction on tree height $i$,

**Base Case** ($i = 0$) Any tree of height 0 represents a single element set (so contains $2^0 = 1$ node)
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

We begin by proving that any tree of height $h$ contains at least $2^h$ nodes

**Proof** by induction on tree height $i$,

**Base Case** ($i = 0$) Any tree of height 0 represents a single element set
   (so contains $2^i = 1$ node)

**Inductive Step**
   Assume every tree of height $(i - 1)$ contains at least $2^{i-1}$ nodes
Now big is \( h \) now?

**Claim** The height, \( h \), of the tallest tree is \( O(\log n) \)

We begin by proving that any tree of height \( h \) contains at least \( 2^h \) nodes.

**Proof** by induction on tree height \( i \),

- **Base Case** \((i = 0)\) Any tree of height 0 represents a single element set (so contains \( 2^i = 1 \) node).

- **Inductive Step**
  
  Assume every tree of height \((i - 1)\) contains at least \( 2^{i-1} \) nodes.

  A tree of height \( i \) is only created when two trees of height \((i - 1)\) merge (as we previously observed).
Now big is $h$ now?

Claim The height, $h$, of the tallest tree is $O(\log n)$

We begin by proving that any tree of height $h$ contains at least $2^h$ nodes

Proof by induction on tree height $i$,

Base Case ($i = 0$) Any tree of height 0 represents a single element set

(so contains $2^i = 1$ node)

Inductive Step

Assume every tree of height $(i - 1)$ contains at least $2^{i-1}$ nodes

A tree of height $i$ is only created when two trees of height $(i - 1)$ merge

(as we previously observed)

Therefore, a tree of height $i$ contains at least $2 \cdot 2^{i-1} = 2^i$ nodes
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

So we have that any tree of height $h$ contains at least $2^h$ nodes.
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

So we have that any tree of height $h$ contains at least $2^h$ nodes

Assume that there is a tree with height $h \geq \log_2 n + 1$
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

So we have that any tree of height $h$ contains at least $2^h$ nodes

Assume that there is a tree with height $h \geq \log_2 n + 1$

This tree contains at least $2^{\log_2 n + 1} > n$ nodes
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

So we have that any tree of height $h$ contains at least $2^h$ nodes

Assume that there is a tree with height $h \geq \log_2 n + 1$

This tree contains at least $2^{\log_2 n + 1} > n$ nodes

*and each node contains a distinct element*
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

So we have that any tree of height $h$ contains at least $2^h$ nodes

Assume that there is a tree with height $h \geq \log_2 n + 1$

This tree contains at least $2^{\log_2 n + 1} > n$ nodes

*and each node contains a distinct element*

Which is a contradiction because the elements are members of the set

$$\{1, 2, 3, 4, \ldots, n\}$$
Now big is $h$ now?

**Claim** The height, $h$, of the tallest tree is $O(\log n)$

So we have that any tree of height $h$ contains at least $2^h$ nodes.

Assume that there is a tree with height $h \geq \log_2 n + 1$

This tree contains at least $2^{\log_2 n + 1} > n$ nodes

and each node contains a distinct element

Which is a contradiction because the elements are members of the set

$$\{1, 2, 3, 4, \ldots, n\}$$

So as $h$ is $O(\log n)$,

the operations **UNION** and **FINDSET** run in $O(\log n)$ time.
Disjoint Set Summary

We have seen a data structure which
stores a collection of disjoint sets

The elements of the sets are the numbers \( \{1, 2, \ldots, n\} \)

The following operations are supported:

- **MAKESET**\( (x) \) - make a new set containing only \( x \)
  - \( x \) cannot be a member of any existing set

- **UNION**\( (x, y) \) - merge the sets containing \( x \) and \( y \) into a single set

- **FINDSET**\( (x) \) - returns the identity of the set containing \( x \)
  - the identity of a set is any unique identifier of the set.

The operations **UNION** and **FINDSET** take \( O(\log n) \) time.

The operation **MAKESET** runs in \( O(1) \) time.
Minimum Spanning Trees

In a connected, undirected graph $G$, a spanning tree is a subgraph $T$ such that

Every vertex $v \in V$ is in $T$
and $T$ is a tree (it contains no cycles)
Minimum Spanning Trees

In a connected, undirected graph $G$, a spanning tree is a subgraph $T$ such that

- Every vertex $v \in V$ is in $T$
- and $T$ is a tree (it contains no cycles)

This graph is an example of a spanning tree.
In a connected, undirected graph $G$, a *spanning tree* is a subgraph $T$ such that

- Every vertex $v \in V$ is in $T$.
- $T$ is a tree (it contains no cycles).

The weight of a spanning tree is the sum of the weights of its edges.
In a connected, undirected graph $G$, a **spanning tree** is a subgraph $T$ such that

- Every vertex $v \in V$ is in $T$
- and $T$ is a tree (*it contains no cycles*)

The weight of a spanning tree is the sum of the weights of its edges.

For the given graph, a spanning tree with weight 23 is shown.
Minimum Spanning Trees

In a connected, undirected graph $G$, a spanning tree is a subgraph $T$ such that

- Every vertex $v \in V$ is in $T$
- and $T$ is a tree (it contains no cycles)

The weight of a spanning tree is the sum of the weights of its edges.

$T$ is a minimum spanning tree if no other spanning tree has a lower weight.
Minimum Spanning Trees

In a connected, undirected graph $G$, a spanning tree is a subgraph $T$ such that

1. Every vertex $v \in V$ is in $T$.
2. $T$ is a tree (it contains no cycles).

The weight of a spanning tree is the sum of the weights of its edges.

$T$ is a minimum spanning tree if no other spanning tree has a lower weight.

Example:

- A spanning tree with weight 23 (not a minimum spanning tree).
- The weight of a spanning tree is the sum of the weights of its edges.
Minimum Spanning Trees

In a connected, undirected graph $G$, a *spanning tree* is a subgraph $T$ such that

- Every vertex $v \in V$ is in $T$ and $T$ is a tree (it contains no cycles)
- The weight of a spanning tree is the sum of the weights of its edges

_a minimum spanning tree with weight 13_

$T$ is a minimum spanning tree if no other spanning tree has a lower weight
Kruskal's algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}

Step 1: For each \( v \in V \), \texttt{MAKESET}(v)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph...

using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

Step 1: For each \( v \in V \), MAKESET(\( v \))

Step 2: Sort the edges in order of increasing weight
Kruskal's algorithm

Kruskal's algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)

\[ \text{If } \text{FINDSET}(u) \neq \text{FINDSET}(v) \text{ then } \]

\[ \text{UNION}(u, v) \text{ and add } (u, v) \text{ to } T \]
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}

**Step 1:** For each \( v \in V \), MAKESET(\( v \))

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E \) (in order)

If FINDSET(\( u \)) \( \neq \) FINDSET(\( v \)) then

UNION(\( u, v \)) and add (\( u, v \)) to \( T \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)

If \( \text{FINDSET}(u) \neq \text{FINDSET}(v) \) then

\( \text{UNION}(u, v) \) and add \( (u, v) \) to \( T \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \texttt{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E\) (in order)

If \( \texttt{FINDSET}(u) \neq \texttt{FINDSET}(v) \) then

\[ \texttt{UNION}(u, v) \text{ and add } (u, v) \text{ to } T \]
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}

Step 1: For each \( v \in V \), MAKESET\((v)\)

Step 2: Sort the edges in order of increasing weight

Step 3: For each \((u, v) \in E\) (in order)
If FINDSET\((u) \neq FINDSET(v)\) then UNION\((u, v)\) and add \((u, v)\) to \(T\)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph...

using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), MAKESET\((v)\)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E\) (in order)

If \(\text{FINDSET}(u) \neq \text{FINDSET}(v)\) then

\[\text{UNION}(u, v)\]

and add \((u, v)\) to \(T\)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from $\{1, 2, 3, \ldots, |V|\}$

Step 1: For each $v \in V$, `MAKESET(v)`

Step 2: Sort the edges in order of increasing weight

Step 3: For each $(u,v) \in E$ (in order)
If `FINDSET(u) \neq FINDSET(v)` then
`UNION(u,v)` and add $(u,v)$ to $T$
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)
If \( \text{FINDSET}(u) \neq \text{FINDSET}(v) \) then
\text{UNION}(u, v) and add \( (u, v) \) to \( T \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \texttt{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E\) (in order)
If \( \texttt{FINDSET}(u) \neq \texttt{FINDSET}(v) \) then \( \texttt{UNION}(u, v) \) and add \((u, v)\) to \( T \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E\) (in order)

If \( \text{FINDSET}(u) \neq \text{FINDSET}(v) \) then

\( \text{UNION}(u, v) \) and add \((u, v)\) to \( T \)
Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), `MAKESET(v)`

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)

If `FINDSET(u) \neq FINDSET(v)` then

\[ \text{UNION}(u, v) \text{ and add } (u, v) \text{ to } T \]
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E\) (in order)

If \( \text{FINDSET}(u) \neq \text{FINDSET}(v) \) then

\[ \text{UNION}(u, v) \] and add \((u, v)\) to \( T \)
Kruskal's algorithm

Kruskal's algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}

Step 1: For each \( v \in V \), \text{MAKESET}(v)

Step 2: Sort the edges in order of increasing weight

Step 3: For each \((u, v) \in E\) (in order)
If \text{FINDSET}(u) \neq \text{FINDSET}(v) then
\text{UNION}(u, v) and add \((u, v)\) to \(T\)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), MAKESET(\( v \))

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)

If FINDSET(\( u \)) \neq FINDSET(\( v \)) then

UNION(\( u, v \)) and add \( (u, v) \) to \( T \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)

If \( \text{FINDSET}(u) \neq \text{FINDSET}(v) \) then

\( \text{UNION}(u, v) \) and add \( (u, v) \) to \( T \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}

Step 1: For each \( v \in V \), \text{MAKESET}(v)

Step 2: Sort the edges in order of increasing weight

Step 3: For each \((u, v) \in E\) \(\text{in order}\)

If \(\text{FINDSET}(u) \neq \text{FINDSET}(v)\) then

\(\text{UNION}(u, v)\) and add \((u, v)\) to \(T\)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)
If \( \text{FINDSET}(u) \neq \text{FINDSET}(v) \) then
\[ \text{UNION}(u, v) \] and add \( (u, v) \) to \( T \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)

If \( \text{FINDSET}(u) \neq \text{FINDSET}(v) \) then

\[ \text{UNION}(u, v) \text{ and add } (u, v) \text{ to } T \]
Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}

\textbf{Step 1:} For each \(v \in V\), \text{MAKESET}(v)

\textbf{Step 2:} Sort the edges in order of increasing weight

\textbf{Step 3:} For each \((u, v) \in E\) (in order)
   
   If \text{FINDSET}(u) \neq \text{FINDSET}(v)\) then
   
   \text{UNION}(u, v) and add \((u, v)\) to \(T\)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), MAKESET(\( v \))

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \( (u, v) \in E \) (in order)

If FINDSET(\( u \)) \( \neq \) FINDSET(\( v \)) then

UNION(\( u, v \)) and add \( (u, v) \) to \( T \)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph . . .

using a disjoint set data structure where the elements are from \(\{1, 2, 3, \ldots, |V|\}\)

**Step 1:** For each \(v \in V\), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E\) (in order)

If \(\text{FINDSET}(u) \neq \text{FINDSET}(v)\) then

\(\text{UNION}(u, v)\) and add \((u, v)\) to \(T\)
Kruskal's algorithm finds a minimum spanning tree in a connected, undirected graph using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}.

**Step 1:** For each \( v \in V \), \text{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E\) (in order)
If \text{FINDSET}(u) \neq \text{FINDSET}(v)\) then
\text{UNION}(u, v) and add \((u, v)\) to \(T\)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

Step 1: For each \( v \in V \), \text{MAKESET}(v)

Step 2: Sort the edges in order of increasing weight

Step 3: For each \( (u, v) \in E \) (in order)
If \( \text{FINDSET}(u) \neq \text{FINDSET}(v) \) then
\[ \text{UNION}(u, v) \text{ and add } (u, v) \text{ to } T \]
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in an connected, undirected graph... using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}

Step 1: For each \(v \in V\), MAKESET(\(v\))

Step 2: Sort the edges in order of increasing weight

Step 3: For each \((u, v) \in E\) (in order)
If \(\text{FINDSET}(u) \neq \text{FINDSET}(v)\) then \(\text{UNION}(u, v)\) and add \((u, v)\) to \(T\)
Kruskal's algorithm

Kruskal's algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

**Step 1:** For each \( v \in V \), \texttt{MAKESET}(v)

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each \((u, v) \in E\) (in order)
If \texttt{FINDSET}(u) \neq \texttt{FINDSET}(v) then
\texttt{UNION}(u, v) and add \((u, v)\) to \(T\)
Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \{1, 2, 3, \ldots, |V|\}

Step 1: For each \( v \in V \), **MAKESET**(\( v \))

Step 2: Sort the edges in order of increasing weight

Step 3: For each \((u, v) \in E\) (in order)

If **FINDSET**(\( u \)) \( \neq \) **FINDSET**(\( v \)) then

**UNION**(\( u, v \)) and add \((u, v)\) to \( T \)

If we implement the operations as we have seen, they run in \( O(\log |V|) \) time
Kruskal’s algorithm

Kruskal's algorithm finds a minimum spanning tree in a connected, undirected graph... using a disjoint set data structure where the elements are from \( \{1, 2, 3, \ldots, |V|\} \)

Step 1: For each \( v \in V \), \text{MAKESET}(v)

Step 2: Sort the edges in order of increasing weight

Step 3: For each \((u, v) \in E\) (in order)

If \text{FINDSET}(u) \neq \text{FINDSET}(v)\) then

\text{UNION}(u, v) \text{ and add } (u, v) \text{ to } T

If we implement the operations as we have seen, they run in \( O(\log |V|) \) time

Therefore the overall running time becomes \( O(|E| \log |V|) \)
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal

*(here we have omitted the proof that Kruskal always outputs a spanning tree)*
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal

(*here we have omitted the proof that Kruskal always outputs a spanning tree*)

Let $M$ be any minimum spanning tree such that $M \neq K$
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal
\[(\text{here we have omitted the proof that Kruskal always outputs a spanning tree})\]

Let $M$ be any minimum spanning tree such that $M \neq K$

$K$ must contain at least one edge not in $M$
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal

(here we have omitted the proof that Kruskal always outputs a spanning tree)

Let $M$ be any minimum spanning tree such that $M \neq K$

$K$ must contain at least one edge not in $M$

(because every spanning tree contains $|V| - 1$ edges)
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal

\textit{(here we have omitted the proof that Kruskal always outputs a spanning tree)}

Let $M$ be any minimum spanning tree such that $M \neq K$

$K$ must contain at least one edge not in $M$

\textit{(because every spanning tree contains $|V| - 1$ edges)}

We will argue that there is another minimum spanning tree, $M_2$

\textit{with one more edge in common with $T$}
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal

(here we have omitted the proof that Kruskal always outputs a spanning tree)

Let $M$ be any minimum spanning tree such that $M \neq K$

$K$ must contain at least one edge not in $M$

(because every spanning tree contains $|V| - 1$ edges)

We will argue that there is another minimum spanning tree, $M_2$

with one more edge in common with $T$

The proof that $K$ is a minimum spanning tree then follows from repeatedly applying this argument
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal

*(here we have omitted the proof that Kruskal always outputs a spanning tree)*

Let $M$ be any minimum spanning tree such that $M \neq K$

$K$ must contain at least one edge not in $M$

*(because every spanning tree contains $|V| - 1$ edges)*

We will argue that there is another minimum spanning tree, $M_2$

with one more edge in common with $T$

The proof that $K$ is a minimum spanning tree then follows from

repeatedly applying this argument

E.g. If there is a minimum spanning tree with 7 edges in common with $K$
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal

*(here we have omitted the proof that Kruskal always outputs a spanning tree)*

Let $M$ be any minimum spanning tree such that $M \neq K$

$K$ must contain at least one edge not in $M$

*(because every spanning tree contains $|V| - 1$ edges)*

We will argue that there is another minimum spanning tree, $M_2$

*with one more edge in common with $T$*

The proof that $K$ is a minimum spanning tree then follows from repeatedly applying this argument

E.g. If there is a minimum spanning tree with 7 edges in common with $K$

then there is a minimum spanning tree with 8 edges in common with $K$
Correctness Sketch

Let $K$ be the spanning tree outputted by Kruskal

*(here we have omitted the proof that Kruskal always outputs a spanning tree)*

Let $M$ be any minimum spanning tree such that $M \neq K$

$K$ must contain at least one edge not in $M$

*(because every spanning tree contains $|V| - 1$ edges)*

We will argue that there is another minimum spanning tree, $M_2$

*with one more edge in common with $T$*

The proof that $K$ is a minimum spanning tree then follows from repeatedly applying this argument

E.g. If there is a minimum spanning tree with 7 edges in common with $K$

then there is a minimum spanning tree with 8 edges in common with $K$

so there is a minimum spanning tree with 9 edges in common with $K$...
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal
and $M$ be any minimum spanning tree such that $M \neq K$
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal
and $M$ be any minimum spanning tree such that $M \neq K$

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

If we were to add $e$ to $M$, we would introduce a cycle.
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal
and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

If we were to add $e$ to $M$ we would introduce a cycle.

because $M$ is a spanning tree.
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

If we were to add $e$ to $M$ we would introduce a cycle. 

because $M$ is a spanning tree

There must be an edge $f$ in this potential cycle which is not in $K$. 
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

If we were to add $e$ to $M$ we would introduce a cycle. *because $M$ is a spanning tree*.

There must be an edge $f$ in this potential cycle which is not in $K$. *because $K$ is a tree, so contains no cycles*.
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

If we were to add $e$ to $M$ we would introduce a cycle.

because $M$ is a spanning tree.

There must be an edge $f$ in this potential cycle which is not in $K$.

because $K$ is a tree, so contains no cycles.

Further, Kruskal’s algorithm must have considered $e$ before $f$. 
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

If we were to add $e$ to $M$ we would introduce a cycle. 

because $M$ is a spanning tree

There must be an edge $f$ in this potential cycle which is not in $K$.

because $K$ is a tree, so contains no cycles

Further, Kruskal's algorithm must have considered $e$ before $f$.

otherwise $f$ would be in $K$.
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$

If we were to add $e$ to $M$ we would introduce a cycle. *because $M$ is a spanning tree*

There must be an edge $f$ in this potential cycle which is not in $K$ *because $K$ is a tree, so contains no cycles*

Further, Kruskal’s algorithm must have considered $e$ before $f$ *otherwise $f$ would be in $K*

As Kruskal considers edges in weight order, the weight of $e$ is at most the weight of $f$
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal
and $M$ be any minimum spanning tree such that $M \neq K$

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$

If we were to add $e$ to $M$ we would introduce a cycle.  

*because $M$ is a spanning tree*

There must be an edge $f$ in this potential cycle which is not in $K$

*because $K$ is a tree, so contains no cycles*

Further, Kruskal’s algorithm must have considered $e$ before $f$

*otherwise $f$ would be in $K$*

As Kruskal considers edges in weight order,
the weight of $e$ is at most the weight of $f$

Let $M_2$ be $M$ with $e$ added and $f$ removed
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$. 
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$

Let $M_2$ be $M$ with $e$ added and $f$ removed
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal
and $M$ be any minimum spanning tree such that $M \neq K$

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$

Let $M_2$ be $M$ with $e$ added and $f$ removed

the weight of $e$ is at most the weight of $f$ . . .
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

Let $M_2$ be $M$ with $e$ added and $f$ removed.

The weight of $e$ is at most the weight of $f$... so $M_2$ is also a minimum spanning tree.
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

Let $M_2$ be $M$ with $e$ added and $f$ removed.

The weight of $e$ is at most the weight of $f$.

So $M_2$ is also a minimum spanning tree.

$e$ is in $K$ but $f$ is not in $K$. . .
How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

Let $M_2$ be $M$ with $e$ added and $f$ removed.

the weight of $e$ is at most the weight of $f$.

so $M_2$ is also a minimum spanning tree.

$e$ is in $K$ but $f$ is not in $K$.

so $M_2$ has one more edge in common with $T$. 


How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal and $M$ be any minimum spanning tree such that $M \neq K$.

Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$.

Let $M_2$ be $M$ with $e$ added and $f$ removed.

The weight of $e$ is at most the weight of $f$.

So $M_2$ is also a minimum spanning tree.

$e$ is in $K$ but $f$ is not in $K$.

So $M_2$ has one more edge in common with $T$.

As we said before, the proof that $K$ is a minimum spanning tree then follows from repeatedly applying this argument.
Summary

We first saw a **data structure** which stores a collection of disjoint sets

*The elements of the sets are the numbers \( \{1, 2, \ldots, n\} \)*

The operations **UNION** and **FINDSET** run in \( O(\log n) \) time

and the operation **MAKESET** runs in \( O(1) \) time.

We then revisited Kruskal’s algorithm

which finds a minimum spanning tree in a connected, undirected graph

and runs in \( O(|E| \log |V|) \) time

*when implemented using the above data structure*

Prims algorithm for finding a minimum spanning tree in a connected, undirected graph

also runs in \( O(|E| \log |V|) \) time

*when the priority queue is implemented using a binary heap*