# Data Structures and Algorithms – COMS21103

2014/2015

# Minimum Spanning Trees

## via Disjoint Sets

Benjamin Sach

University of BRISTOL

# Data Structures and Algorithms – COMS21103

2014/2015

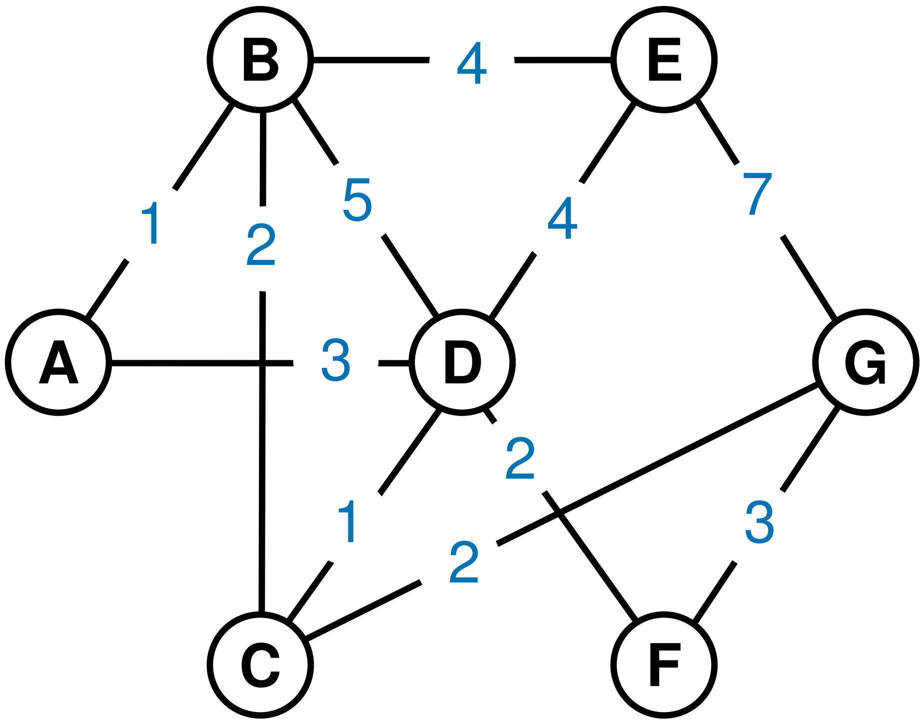
**Minimum Spanning Trees**

# via Disjoint Sets

Benjamin Sach

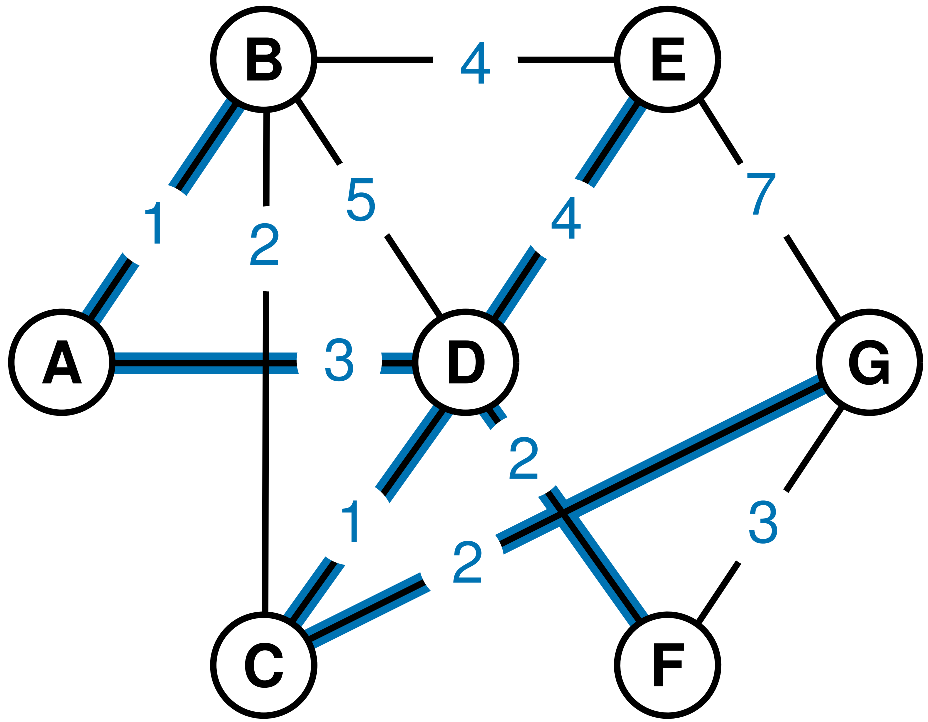
University of BRISTOL

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 numbers from* $\{1, 2, \ldots, n\}$

The following operations are supported:

$\textsc{MakeSet}(x)$ - make a new set containing only $x$

$x$ *cannot be a member of any existing 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 $x$

*the identity of a set is any unique identifier of the set.*

All we require from $\textsc{FindSet}$ is that $\textsc{FindSet}(x) = \textsc{FindSet}\,(y)$

if and only if $x$ and $y$ are currently in the same set

# 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 $x$

# 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* $\{1, 2, \ldots, 16\}$

# 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$(3)$

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(3)$

$$\{3\}$$

# 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* $\{1, 2, \ldots, 16\}$

$$\{3\}$$

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(7)$

$$\{3\}$$

# 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* $\{1, 2, \ldots, 16\}$

MAKESET$(7)$

$\{3\}$ $\{7\}$

# 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* $\{1, 2, \ldots, 16\}$

$\{3\}$ $\{7\}$

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(4)$

$\{3\}$ $\{7\}$

# 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* $\{1, 2, \ldots, 16\}$

$\text{MAKESET}(4)$

$\{3\} \quad \{7\} \quad \{4\}$

## 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* $\{1, 2, \ldots, 16\}$

$$\{3\} \qquad \{7\} \qquad \{4\}$$

# 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* $\{1, 2, \dots, 16\}$

Union$(3, 7)$

$\{3\}$ $\{7\}$ $\{4\}$

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{Union}(3, 7)$

$\{3\}$ $\{7\}$ $\{4\}$

merge these

# 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* $\{1, 2, \ldots, 16\}$

$\text{UNION}(3, 7)$

$\{3, 7\}$ $\{4\}$

# 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* $\{1, 2, \ldots, 16\}$

$$\{3, 7\} \qquad \{4\}$$

# 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$(5)$

$$\{3, 7\} \qquad \{4\}$$

University of BRISTOL

# 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* $\{1, 2, \ldots, 16\}$

$\text{MAKESET}(5)$

$\{5\}$ $\{3, 7\}$ $\{4\}$

# 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* $\{1, 2, \ldots, 16\}$

$\text{MAKESET}(5)$ $\text{MAKESET}(9)$

$\{5\}$ $\{3, 7\}$ $\{4\}$

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(5)$ $\textsc{MakeSet}(9)$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(5)$ $\textsc{MakeSet}(9)$ $\textsc{MakeSet}(2)$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(5)$ $\textsc{MakeSet}(9)$ $\textsc{MakeSet}(2)$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(5)$ $\textsc{MakeSet}(9)$ $\textsc{MakeSet}(2)$ $\textsc{MakeSet}(11)$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(5)$ $\textsc{MakeSet}(9)$ $\textsc{MakeSet}(2)$ $\textsc{MakeSet}(11)$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(5)$ $\textsc{MakeSet}(9)$ $\textsc{MakeSet}(2)$ $\textsc{MakeSet}(11)$ $\textsc{MakeSet}(16)$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{MakeSet}(5)$ $\textsc{MakeSet}(9)$ $\textsc{MakeSet}(2)$ $\textsc{MakeSet}(11)$ $\textsc{MakeSet}(16)$

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

# 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* $\{1, 2, \ldots, 16\}$

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

## 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$(4, 9)$

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

# 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$(4, 9)$

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

merge these

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{Union}(4, 9)$

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

# 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* $\{1, 2, \ldots, 16\}$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{Union}(2, 16)$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{Union}(2, 16)$

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

merge these

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{Union}(2, 16)$

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

# 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* $\{1, 2, \ldots, 16\}$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{Union}(7, 2)$

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

# 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* $\{1, 2, \ldots, 16\}$

Union$(7, 2)$

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

merge these

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{Union}(7, 2)$

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

# 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* $\{1, 2, \ldots, 16\}$

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

# 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* $\{1, 2, \ldots, 16\}$

$\textsc{Union}(3, 5)$

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

# 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$(3, 5)$

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

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* $\{1, 2, \ldots, 16\}$

Union$(3, 5)$

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

# 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* $\{1, 2, \ldots, 16\}$

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

# 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* $\{1, 2, \ldots, 16\}$

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

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

# 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 $3$

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

# 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 $3$

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

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 $3$

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

FindSet$(4)$ returns $9$

FindSet$(9)$ 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 $3$

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

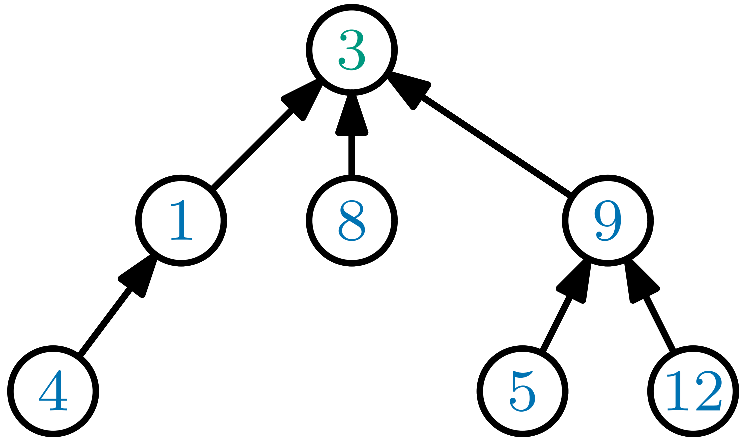
FINDSET$(4)$ returns $9$

FINDSET$(9)$ returns $9$

*In our data structure, the identity will be an element of the set*

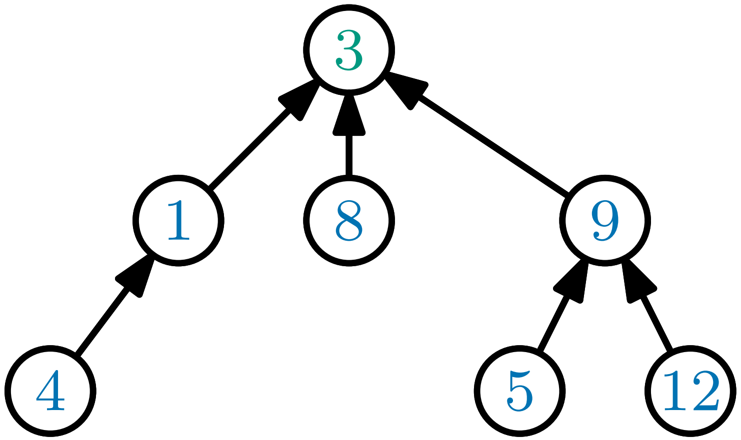
# Reverse Trees

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



# Reverse Trees

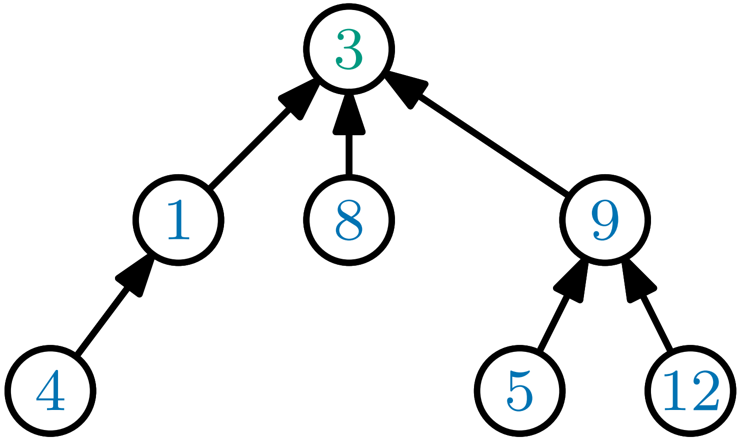
The data structure we will discuss stores each set as a reverse tree:




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:

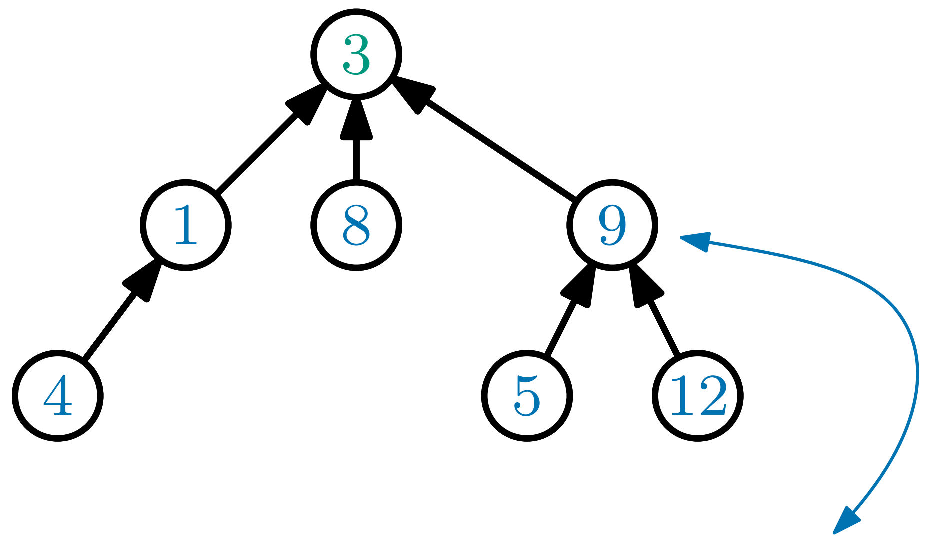



This reverse tree stores the set $\{1, 3, 4, 5, 8, 9, 12\}$

Each node stores an element from the set

# Reverse Trees

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

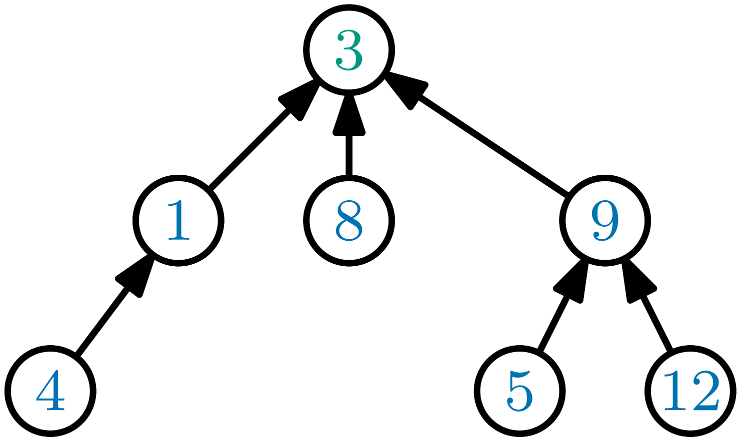



This reverse tree stores the set $\{1, 3, 4, 5, 8, 9, 12\}$

Each node stores an element from the set

# Reverse Trees

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

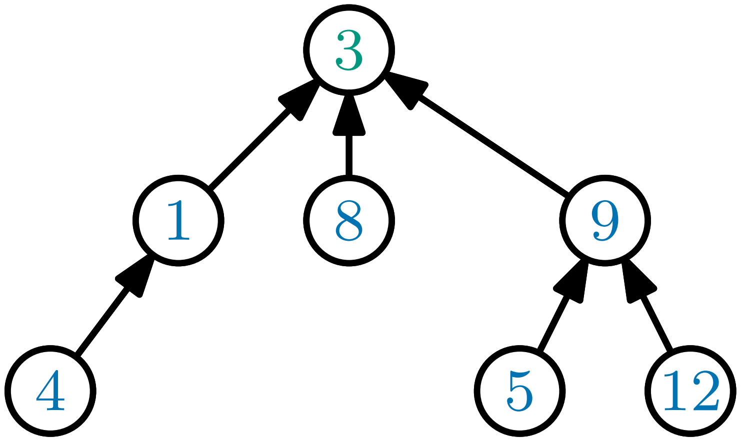



This reverse tree stores the set $\{1, 3, 4, 5, 8, 9, 12\}$

Each node stores an element from the set

# Reverse Trees

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



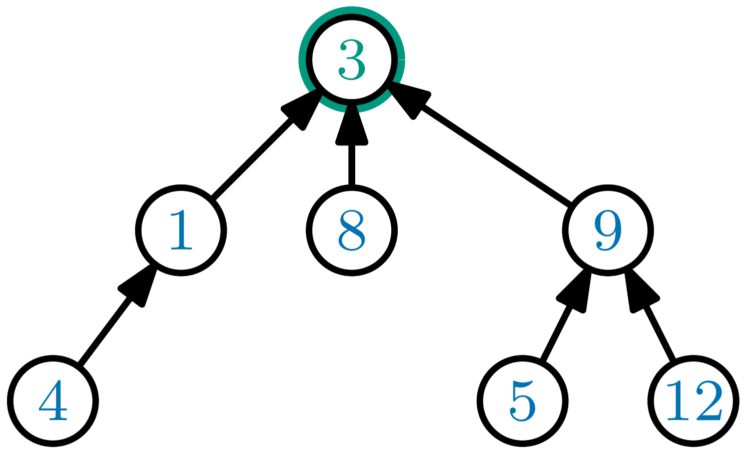

This reverse tree stores the set $\{1, 3, 4, 5, 8, 9, 12\}$

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:



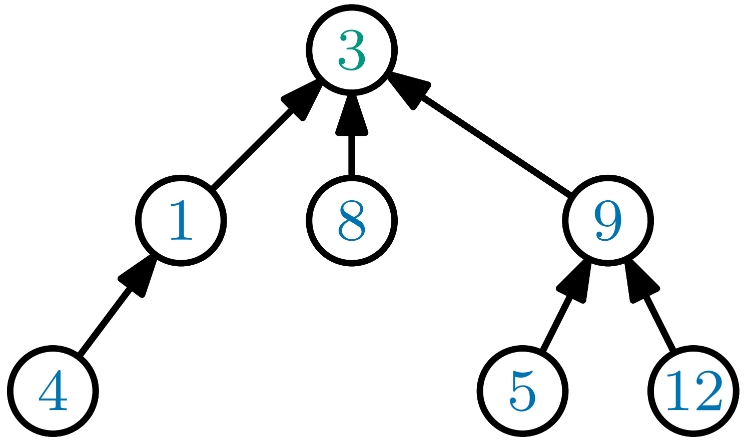

This reverse tree stores the set $\{1, 3, 4, 5, 8, 9, 12\}$

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:



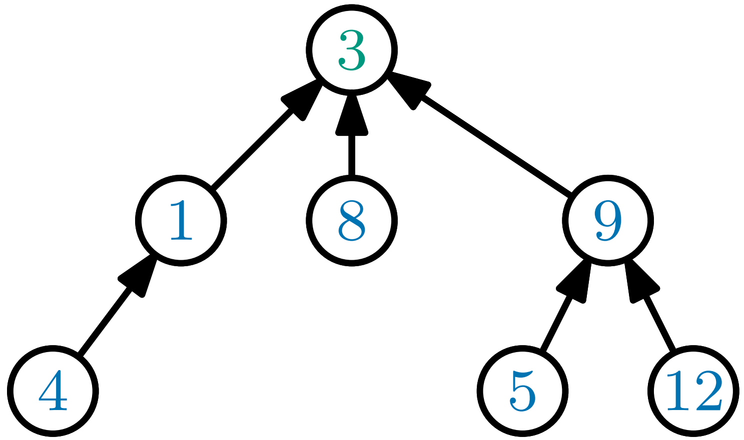

This reverse tree stores the set $\{1, 3, 4, 5, 8, 9, 12\}$

Each node stores an element from the set

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

# Reverse Trees

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




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

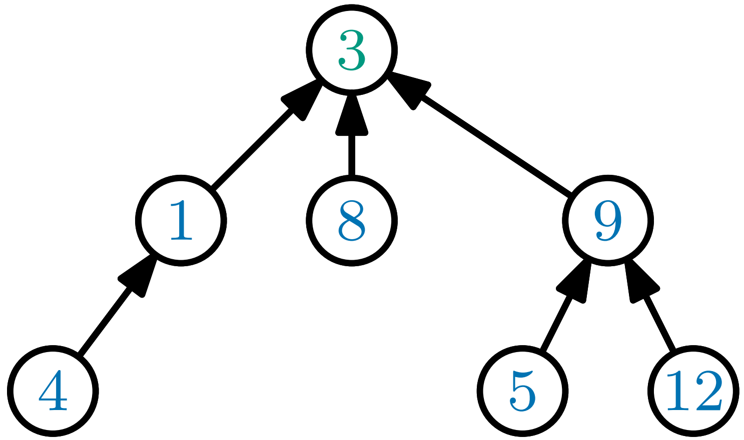
Each node stores an element from the set

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:




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

Each node stores an element from the set

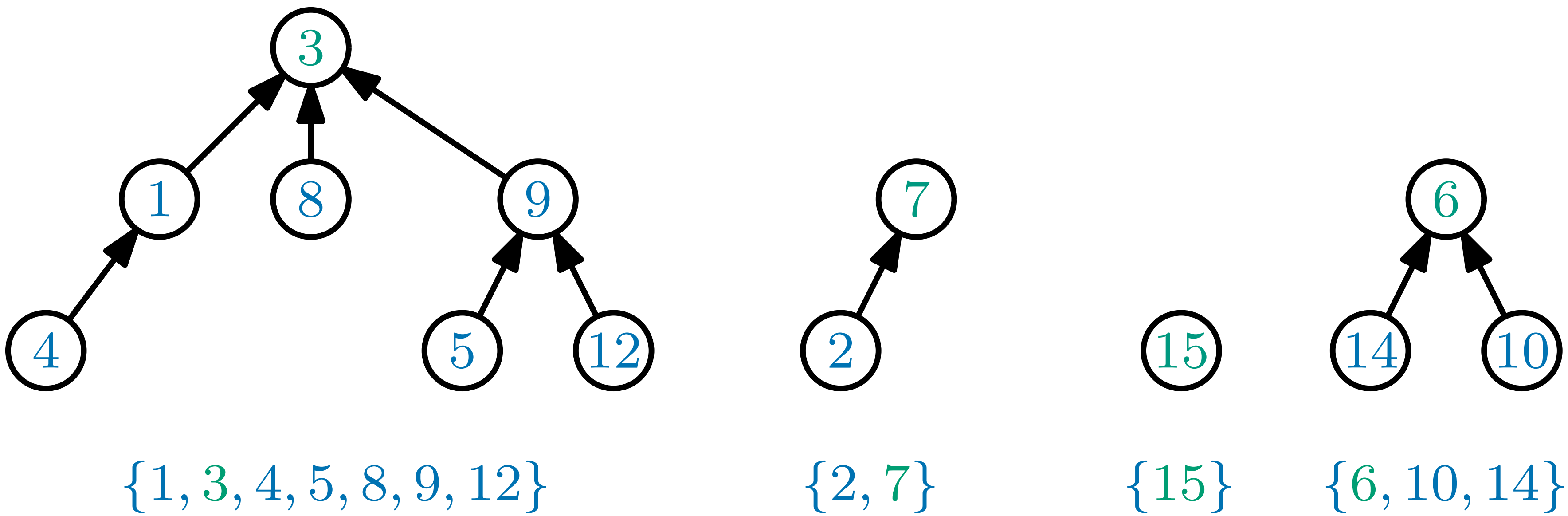
The identity of a set is element at the root (here $3$)

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

*- there will be no limit on the number of children each node can have*

# Reverse Forests

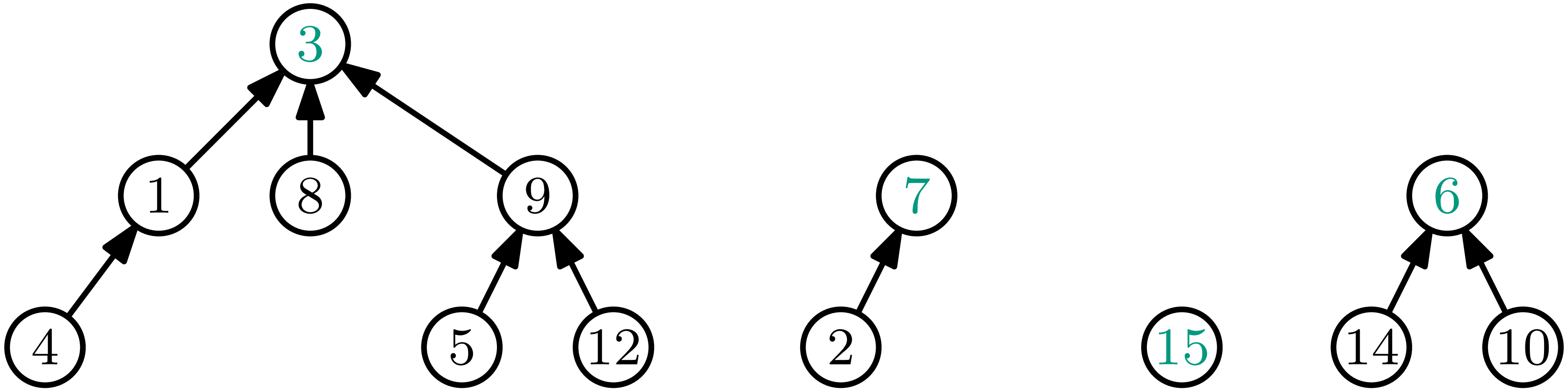
The data structure consists of a forest of reverse trees, one for each set




Each node stores an element from the set

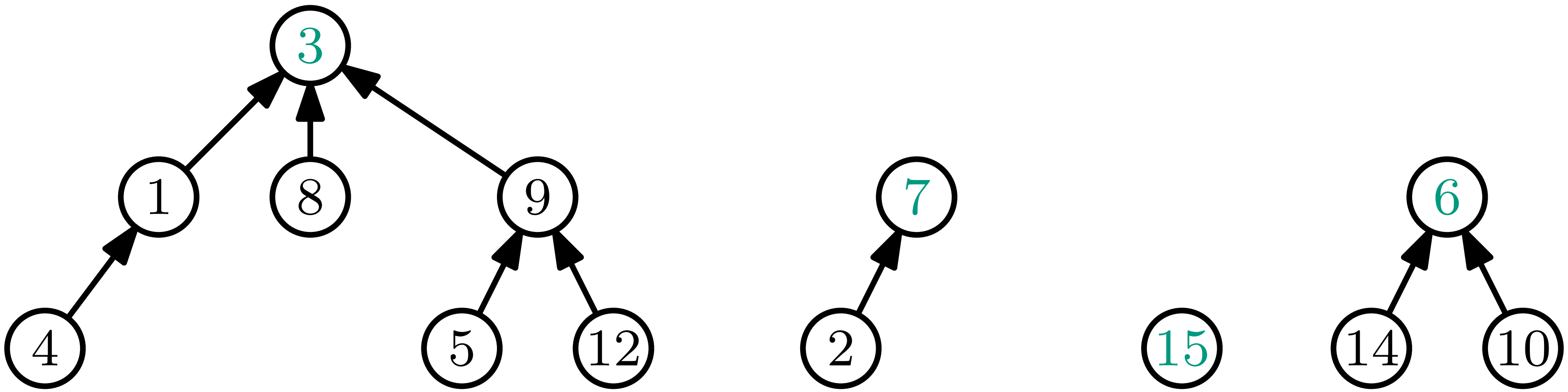
The identity of a set is element at the root

# How are these trees stored?




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

# How are these trees stored?

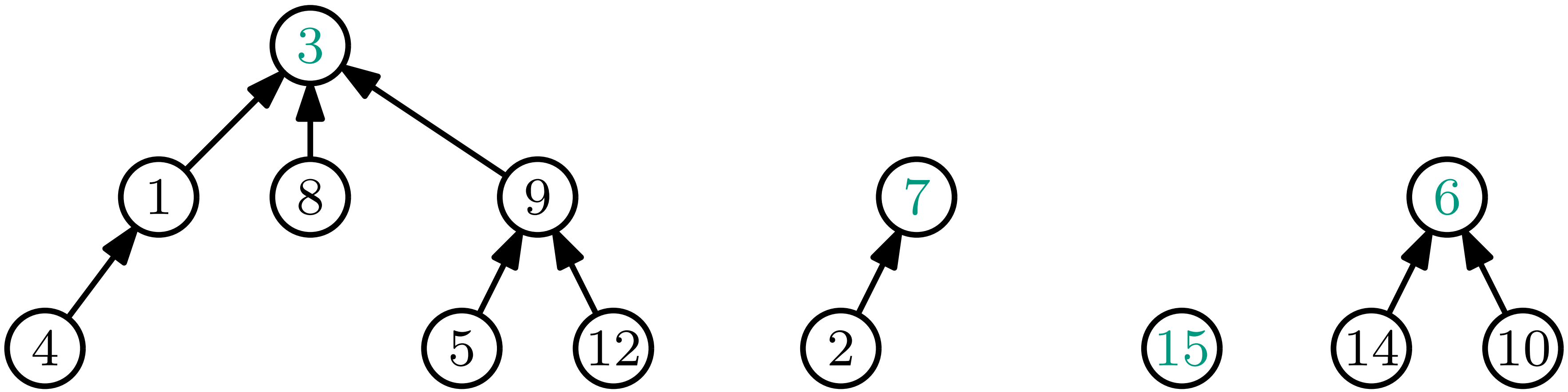


$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | $\cancel{11}$ | 12 | $\cancel{13}$ | 14 | 15 | $\cancel{16}$ |

# How are these trees stored?



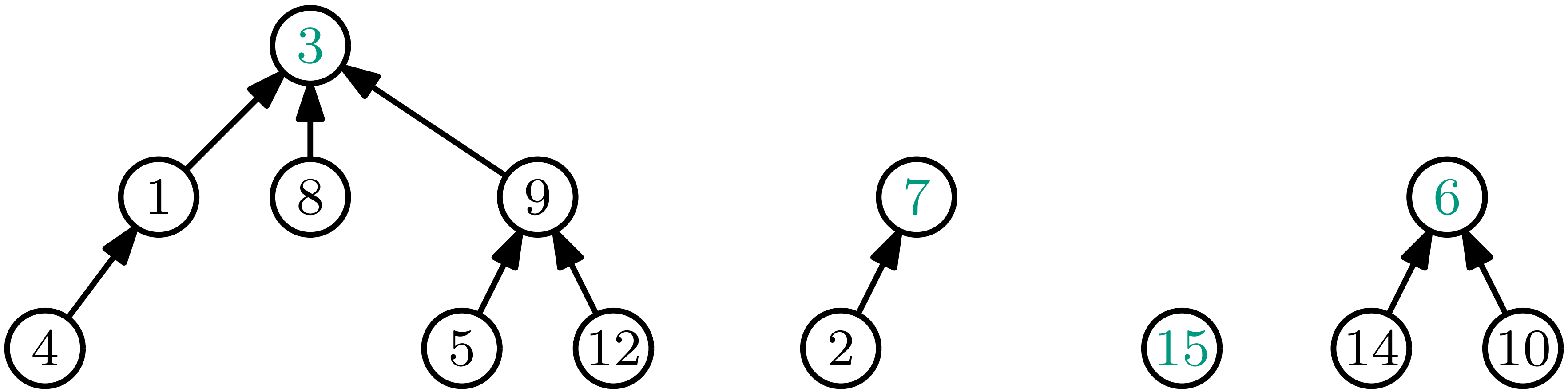
$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | $\cancel{11}$ | 12 | $\cancel{13}$ | 14 | 15 | $\cancel{16}$ |

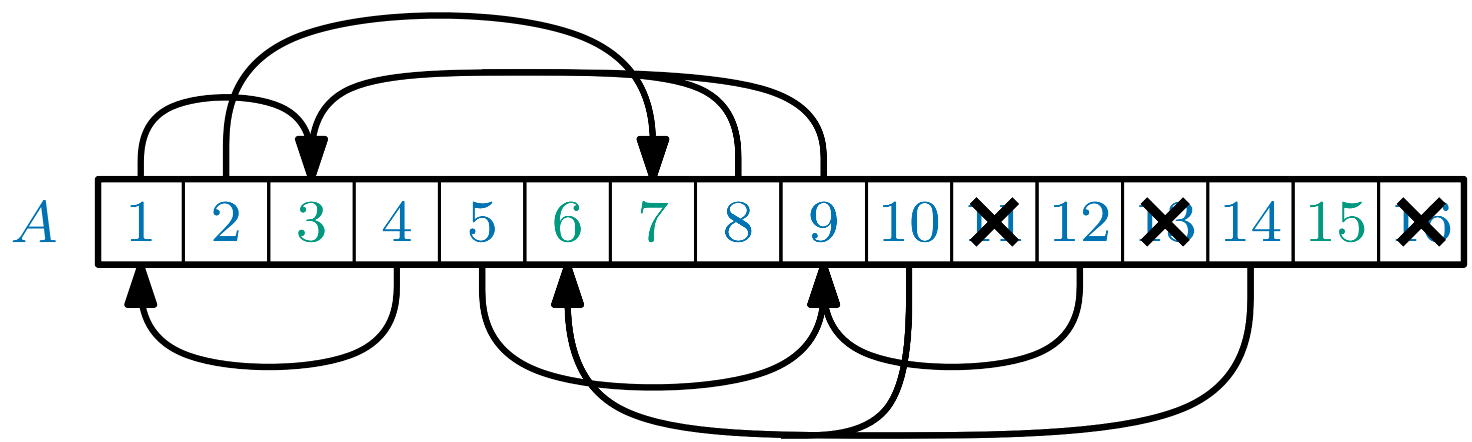
*recall that the elements of the sets are numbers from* $\{1, 2, \ldots, n\}$

# How are these trees stored?

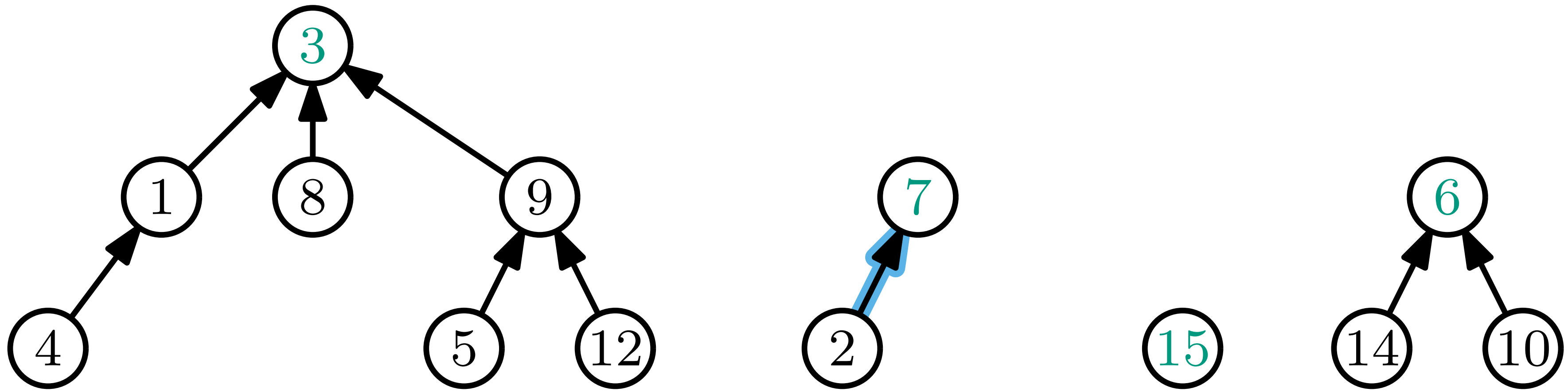



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

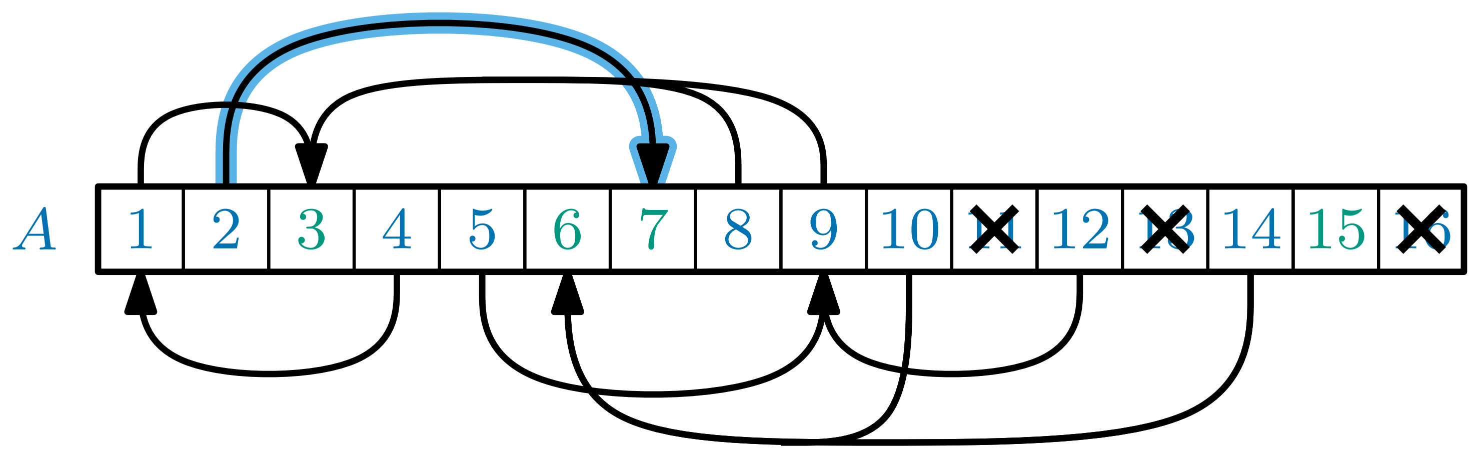


# How are these trees stored?

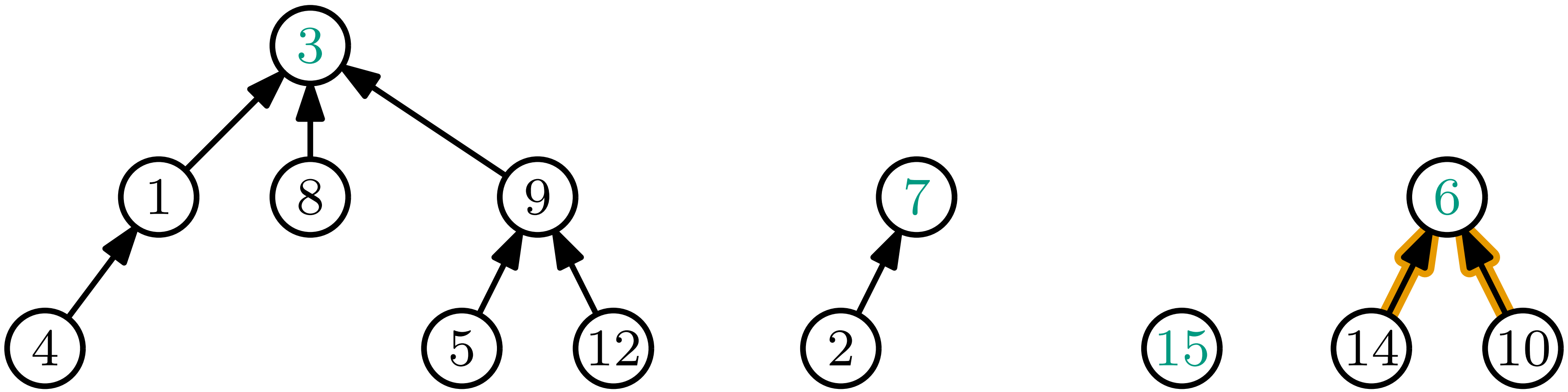



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

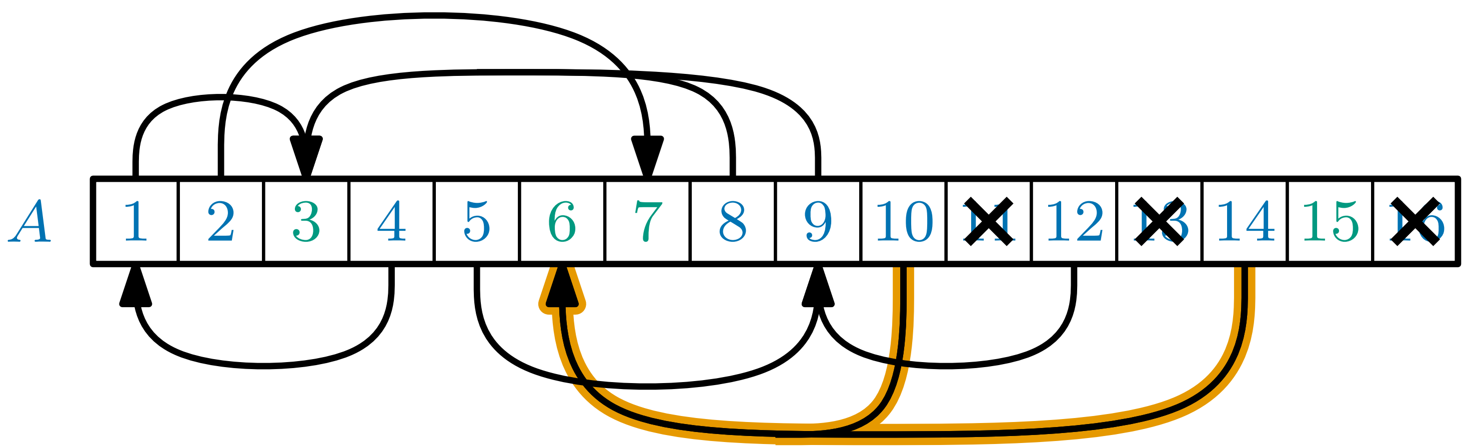


# How are these trees stored?

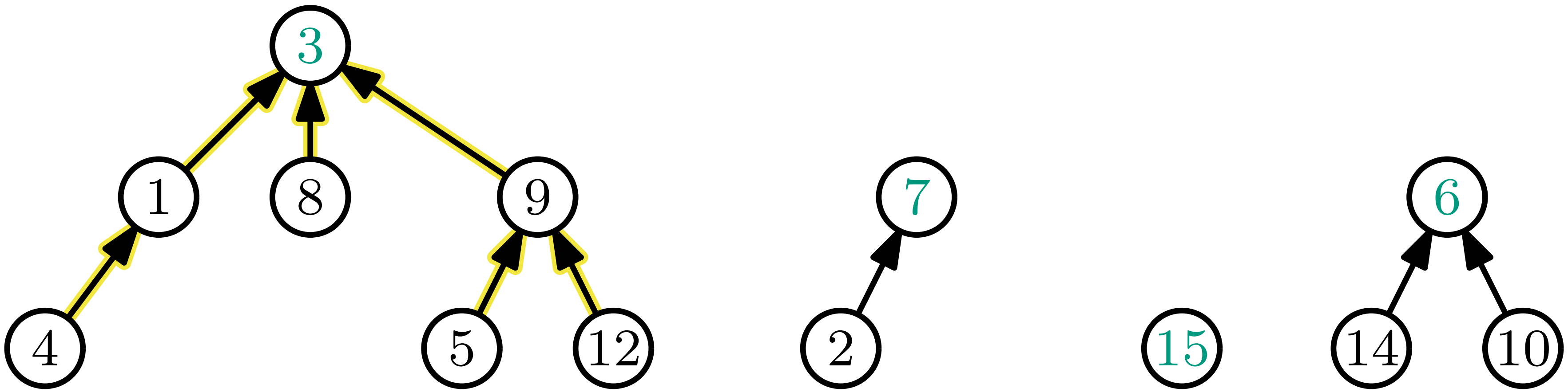



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

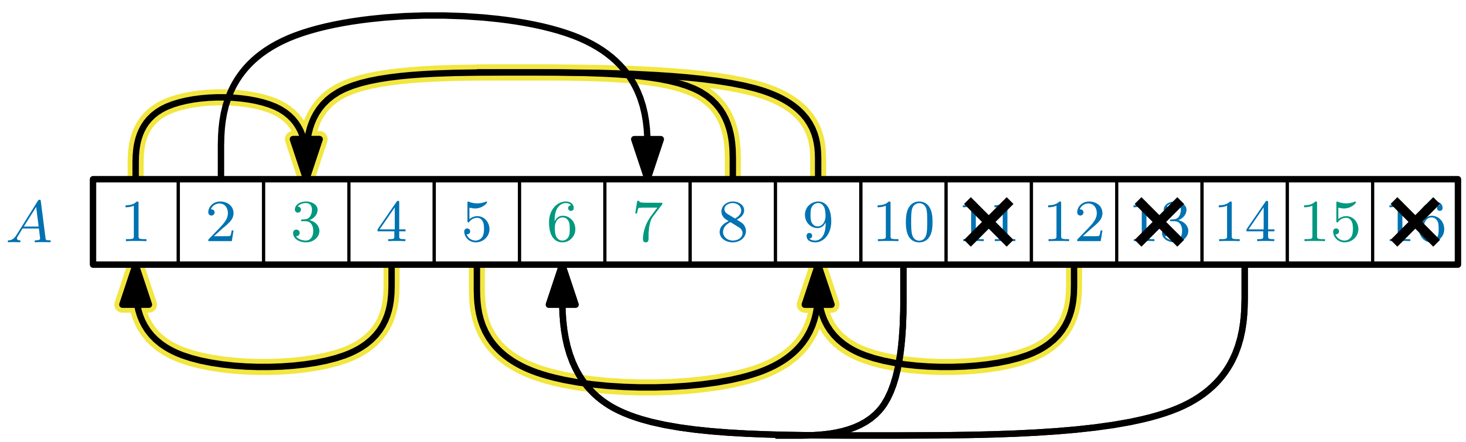


# How are these trees stored?

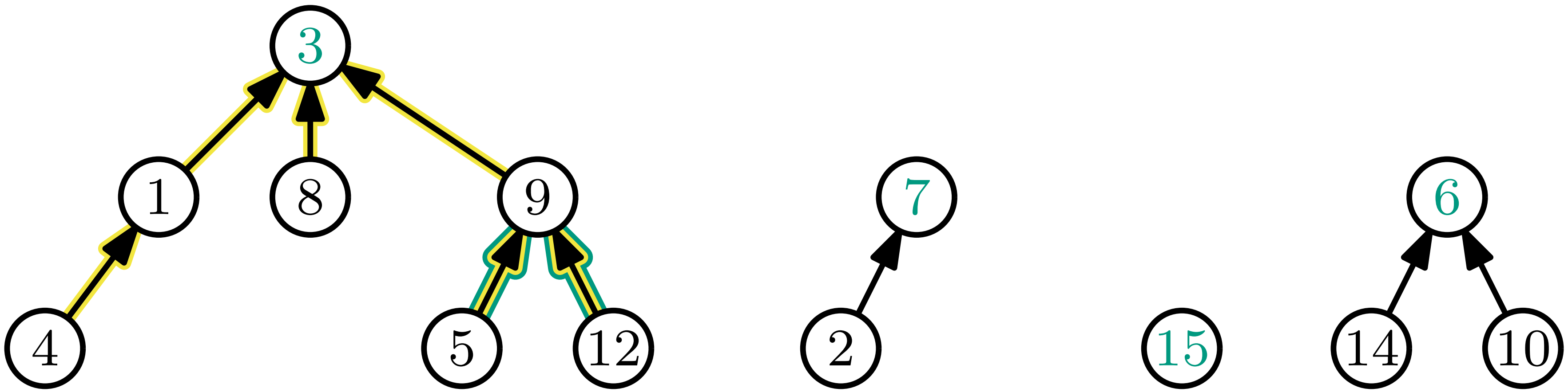


$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

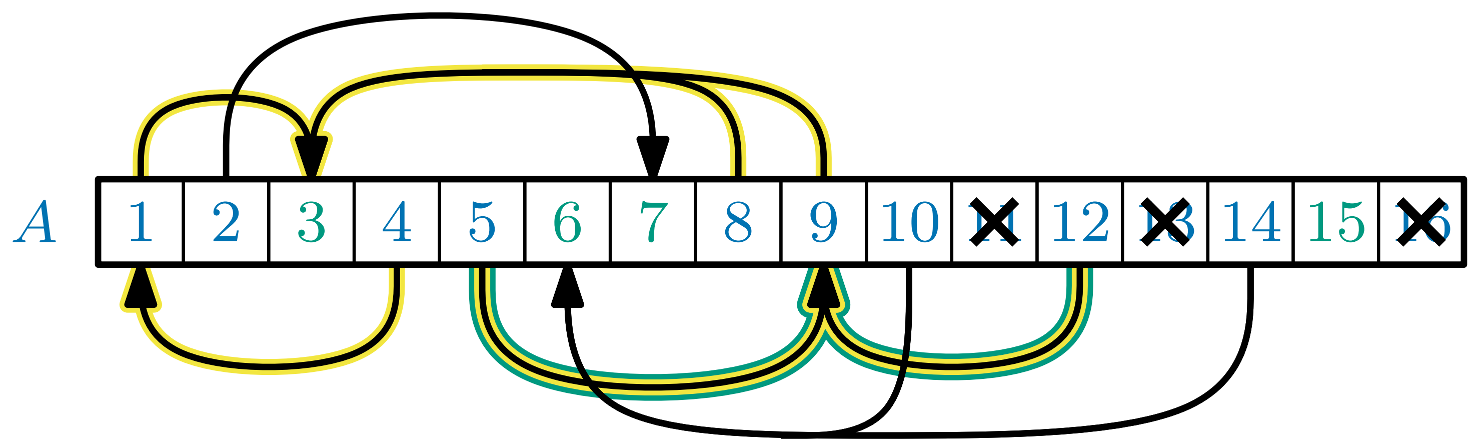


# How are these trees stored?

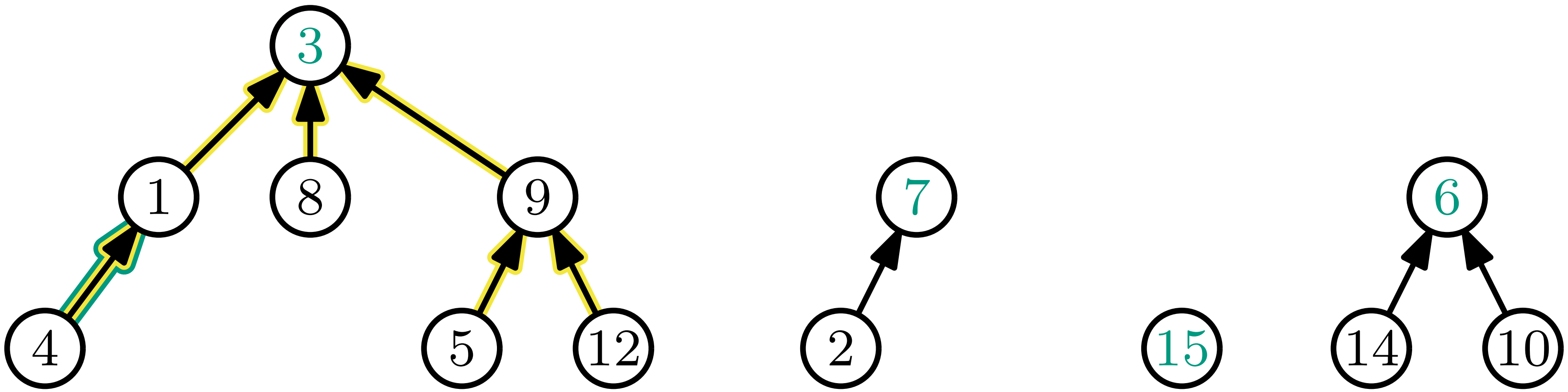



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

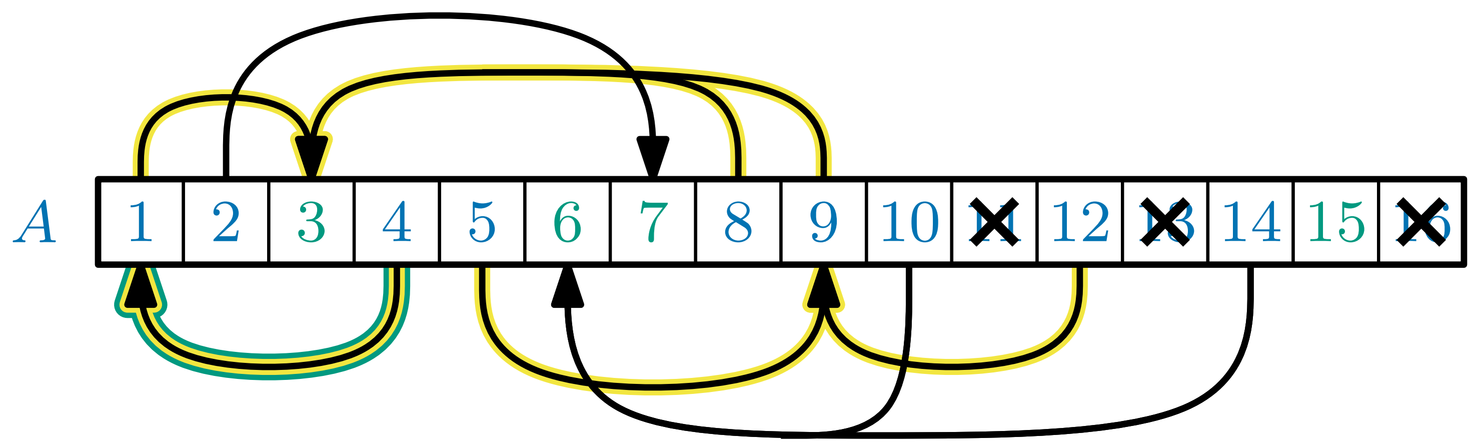


# How are these trees stored?

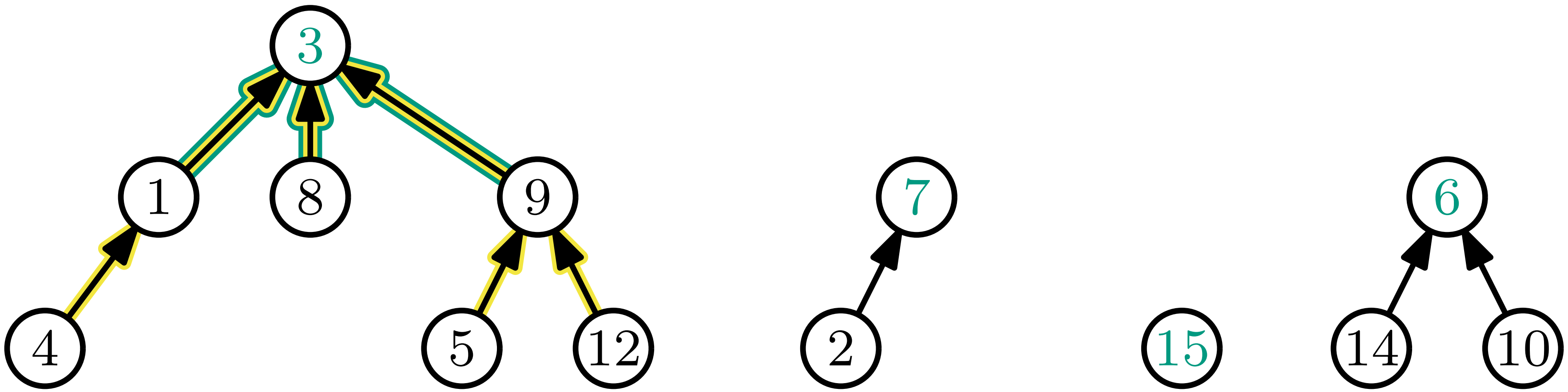



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

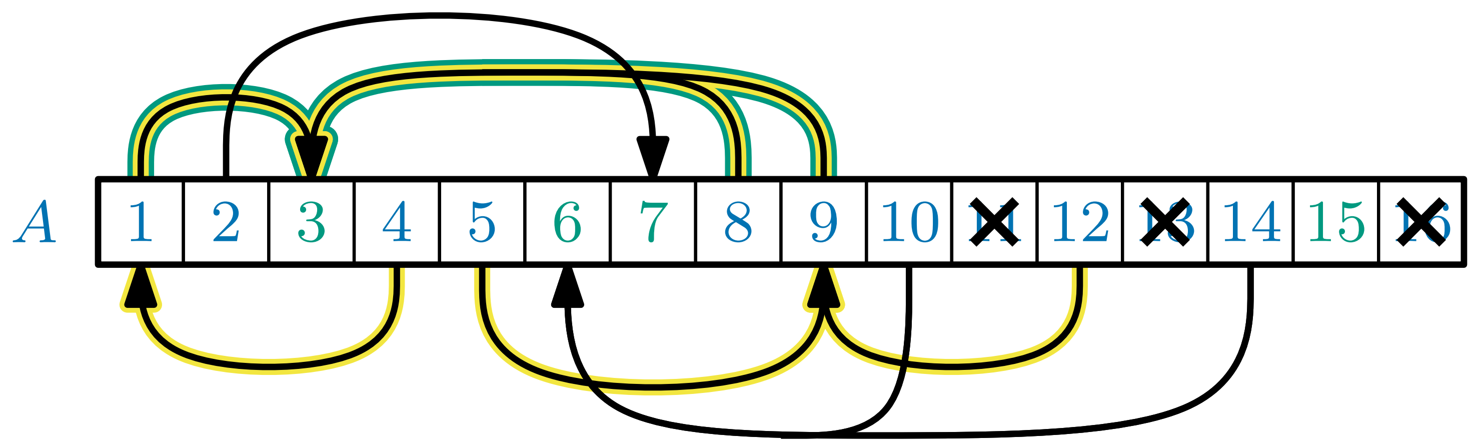


# How are these trees stored?

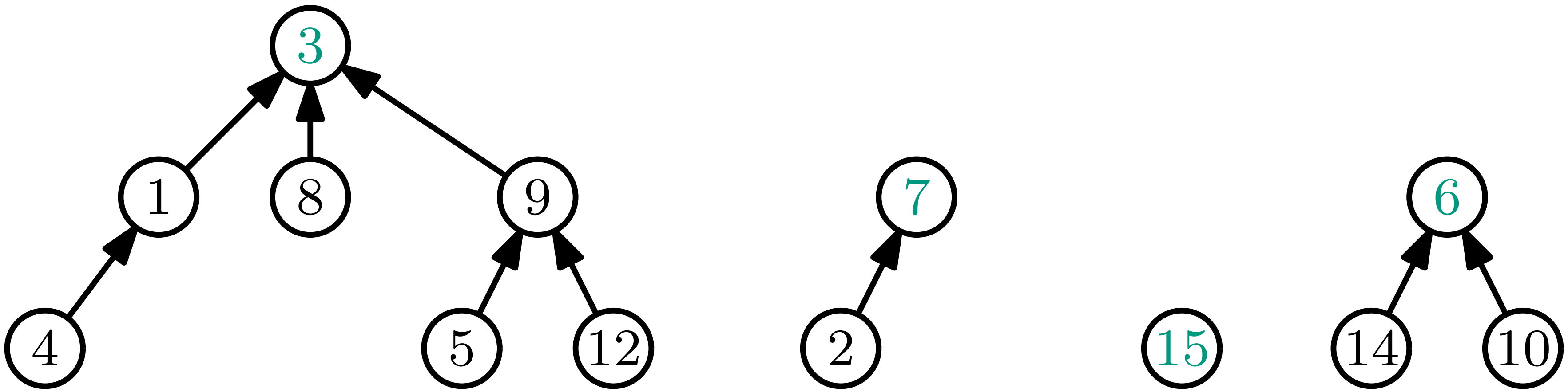



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

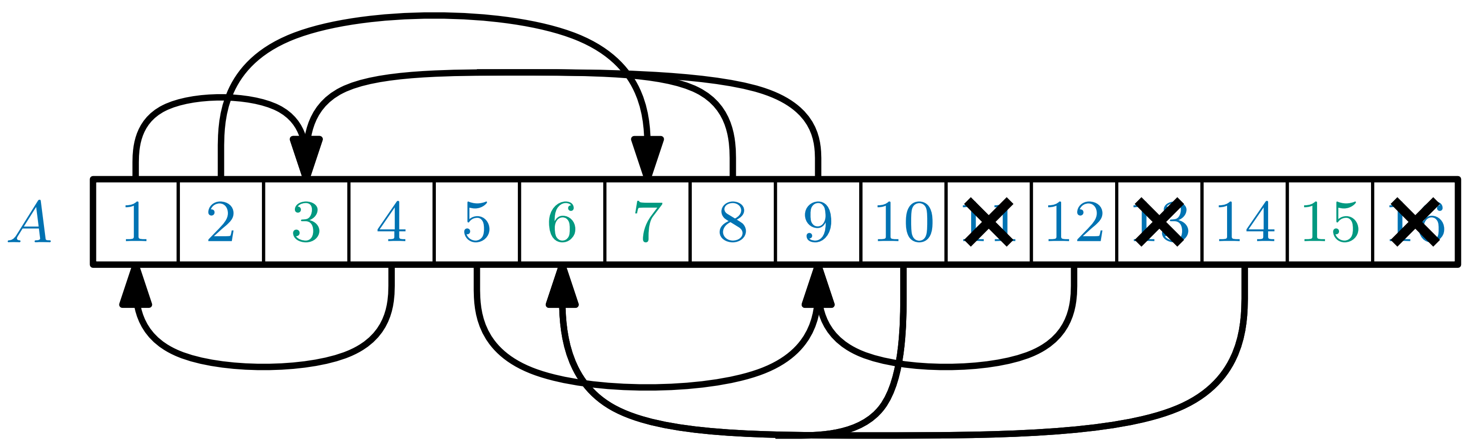


## How are these trees stored?

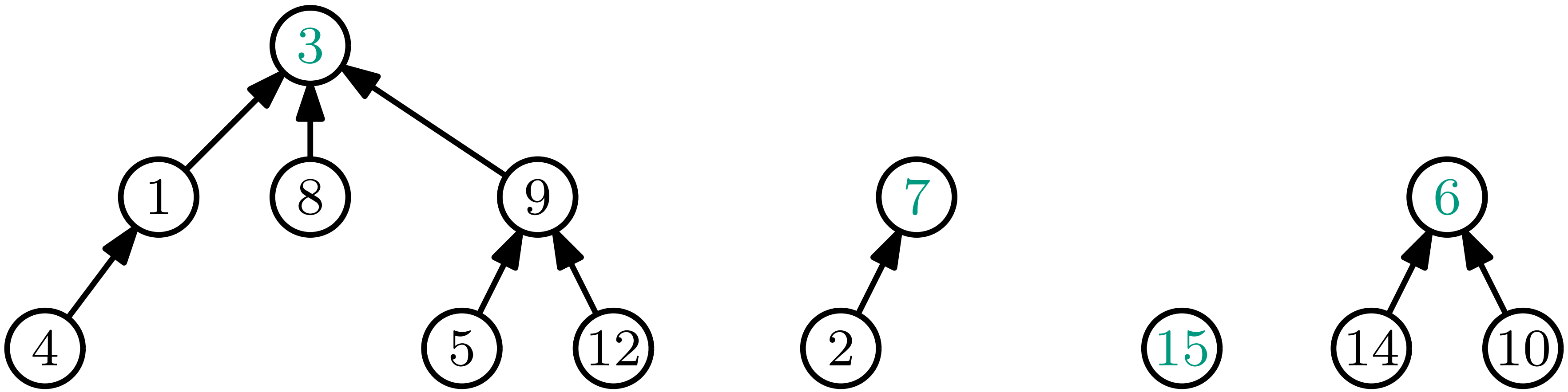



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

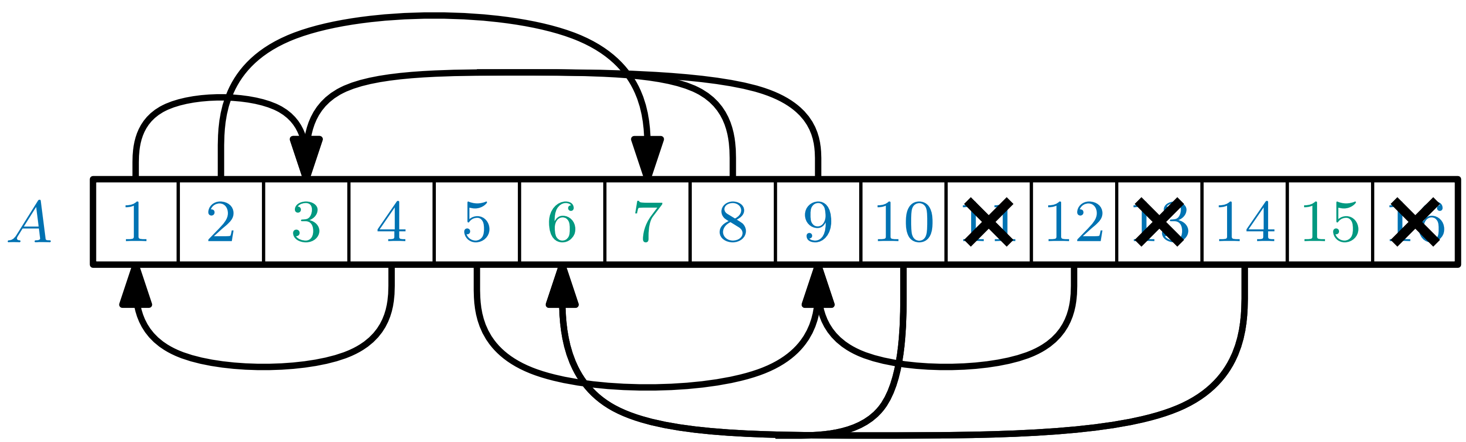


# How are these trees stored?




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

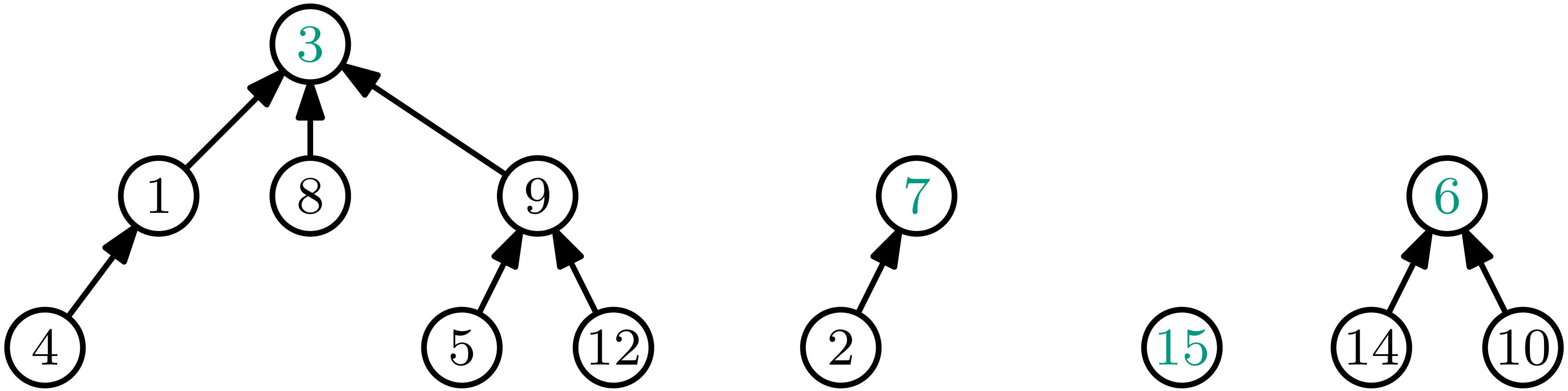
The elements are stored in an array of length $n$:




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

# The FINDSET operation

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




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

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

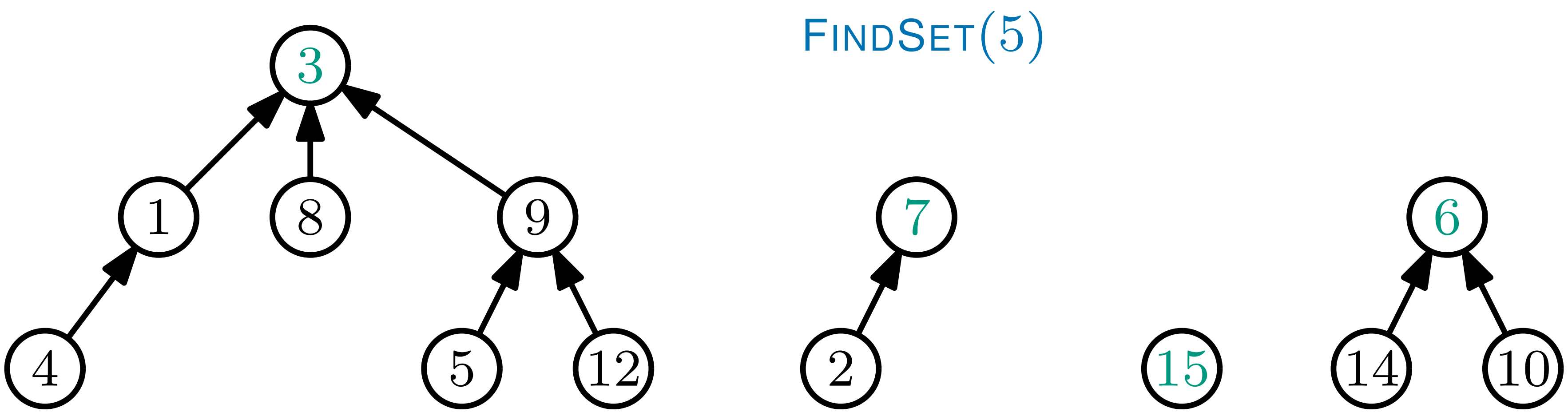
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FINDSET operation

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




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

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

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

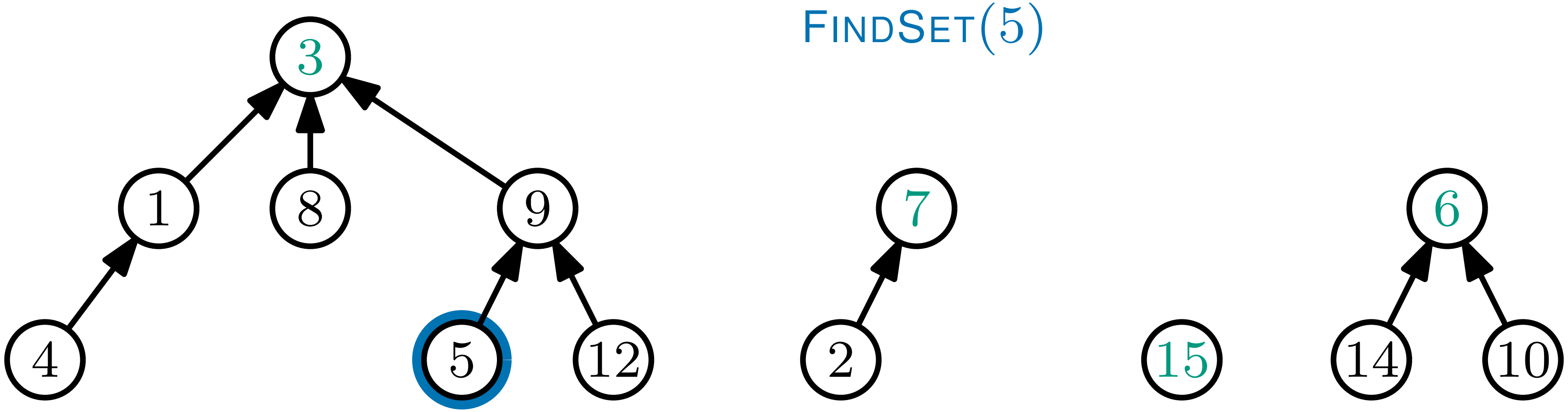
follow the pointer to the parent of the current node

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

# The FindSet operation

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

FindSet$(5)$




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

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

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

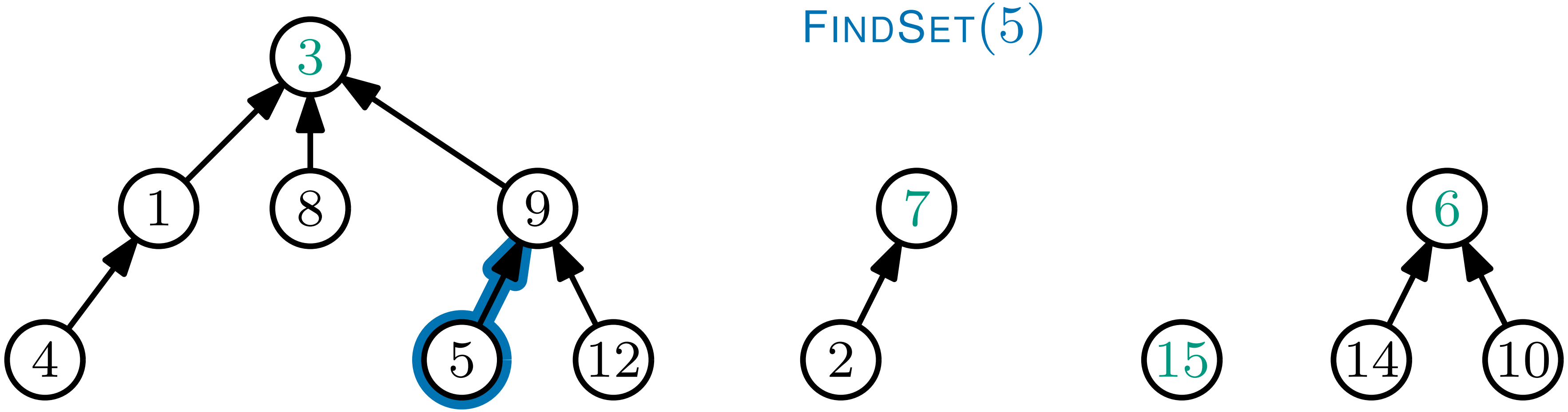
follow the pointer to the parent of the current node

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

# The FINDSET operation

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

FINDSET$(5)$




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

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

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

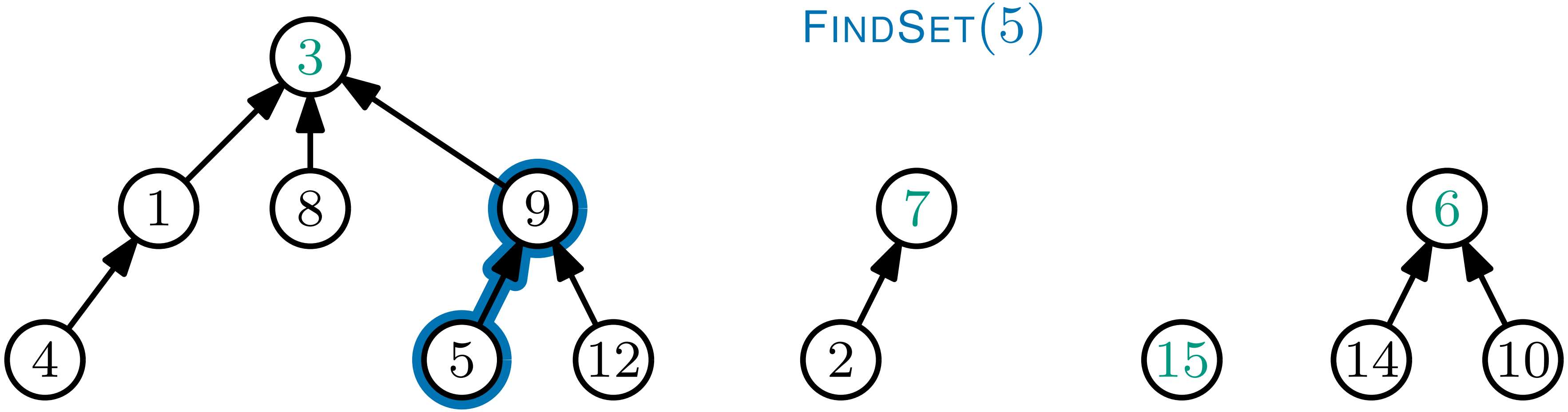
follow the pointer to the parent of the current node

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

# The FINDSET operation

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

FINDSET$(5)$




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

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

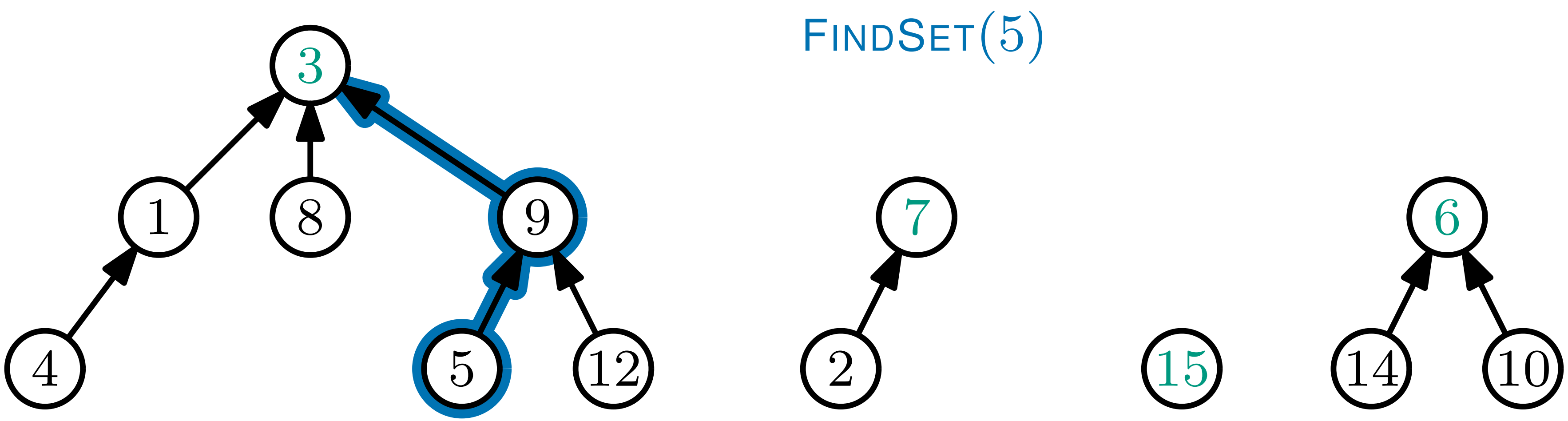
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FindSet operation

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




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

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

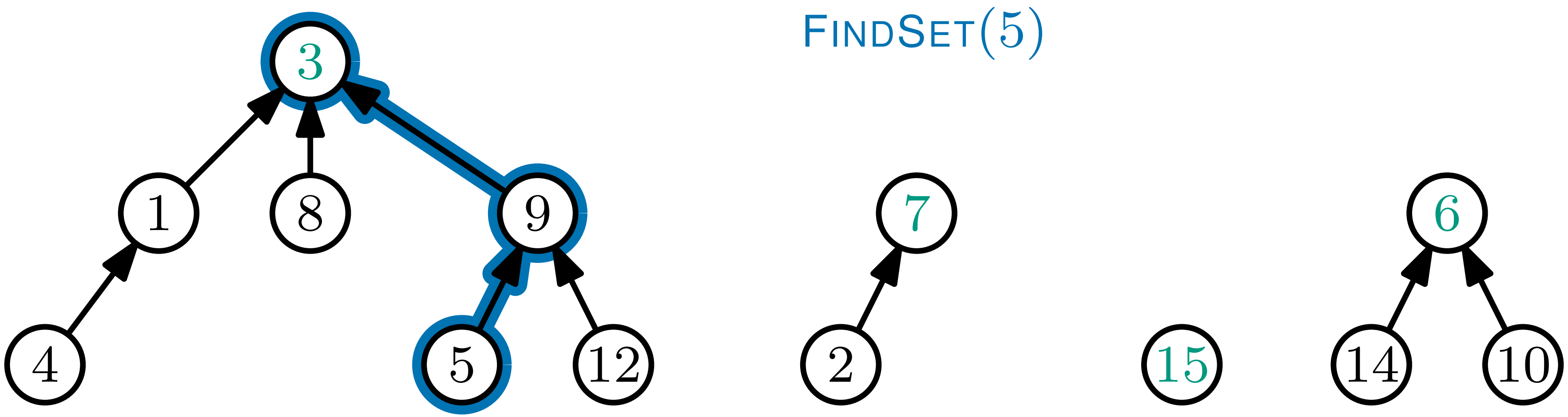
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FindSet operation

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




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

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

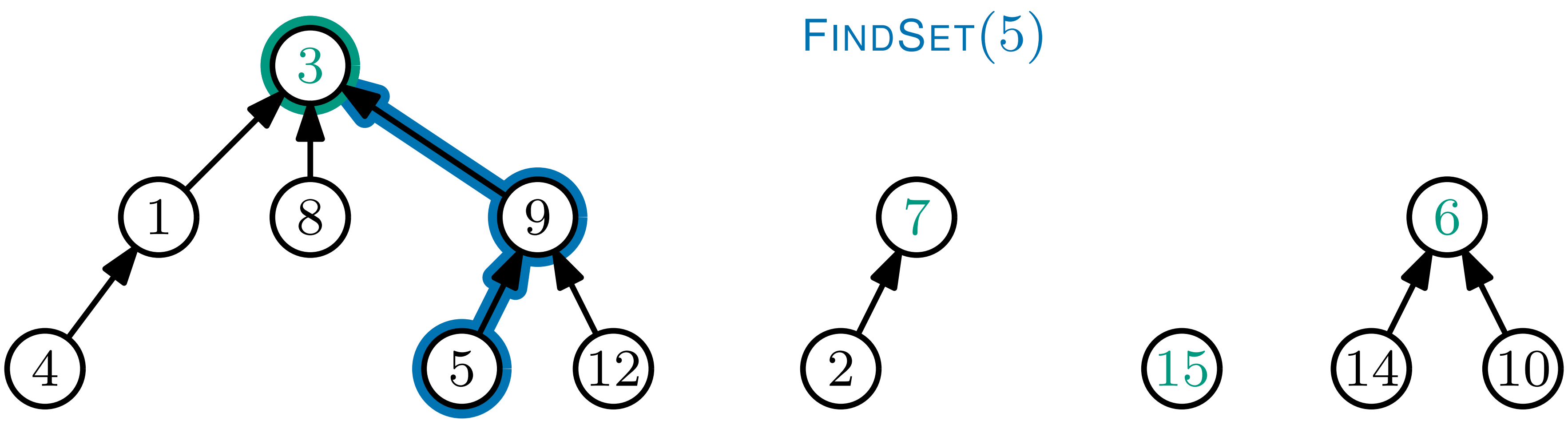
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FINDSET operation

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




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

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

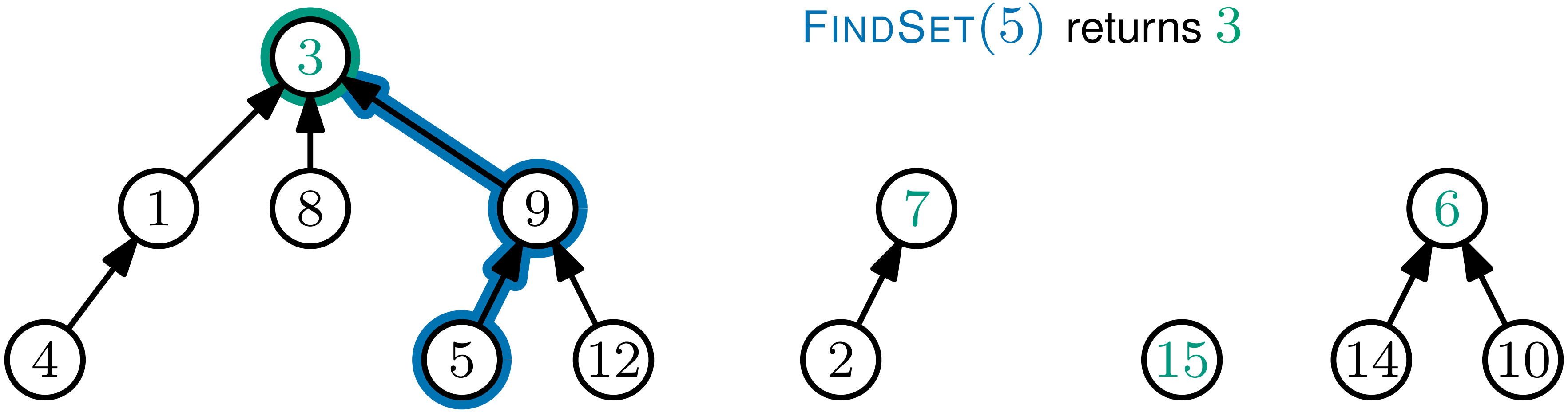
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FINDSET operation

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




FINDSET$(5)$ returns $3$

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

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

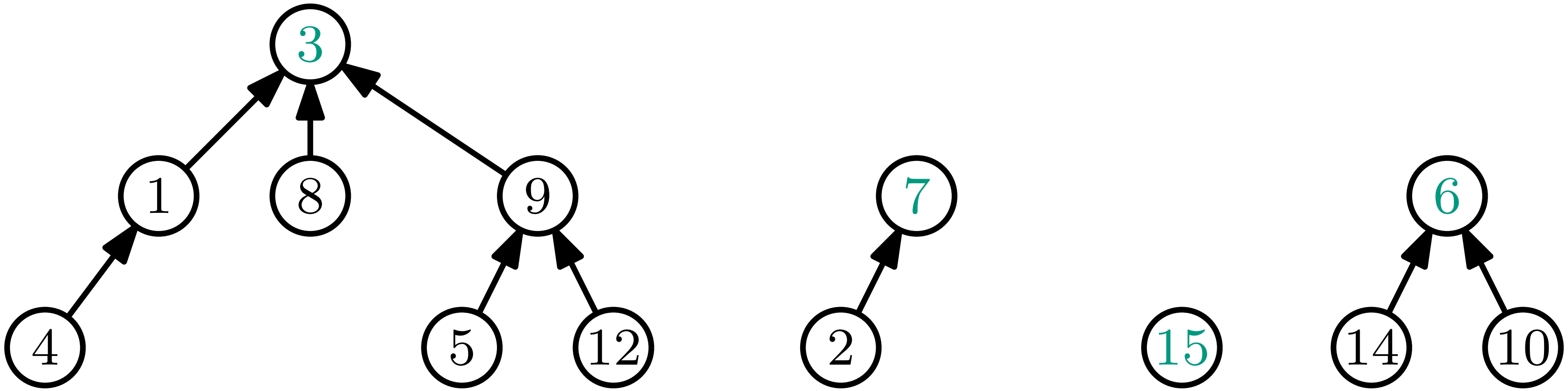
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FINDSET operation

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




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

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

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

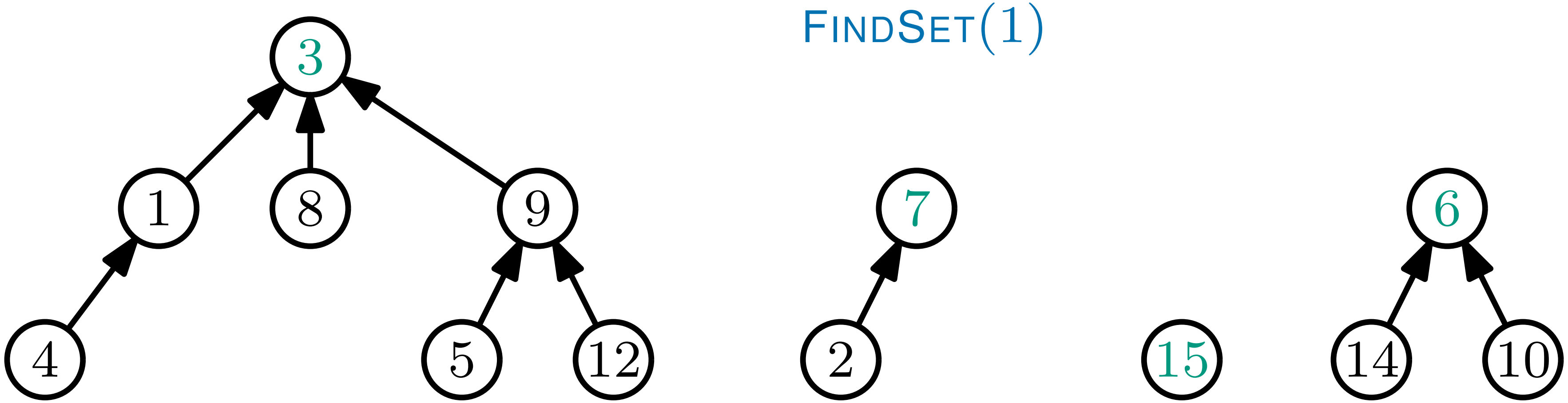
follow the pointer to the parent of the current node

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

# The FINDSET operation

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

FINDSET$(1)$




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

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

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

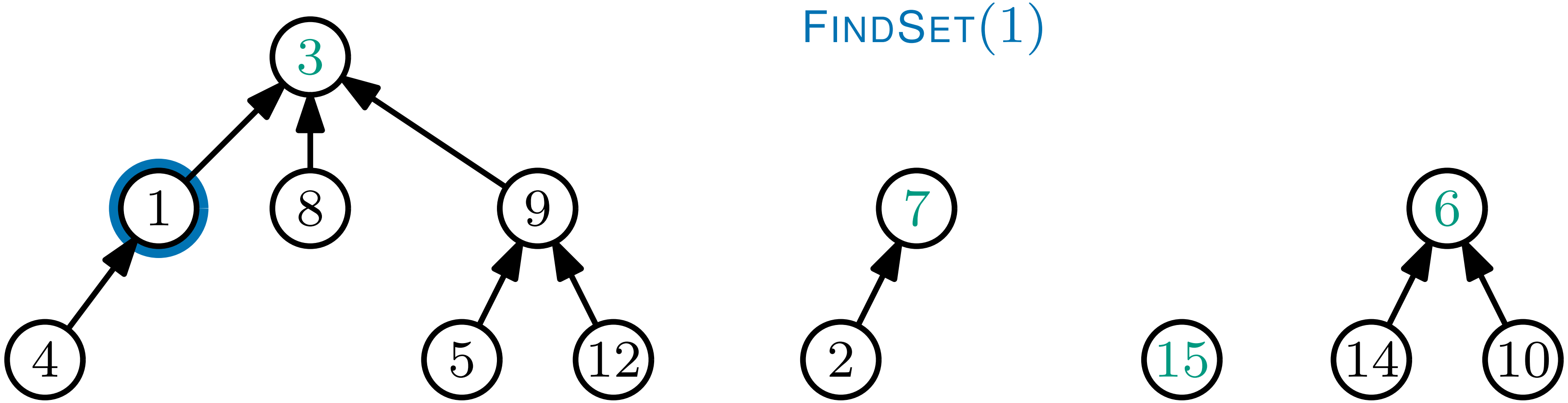
follow the pointer to the parent of the current node

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

# The FINDSET operation

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

FINDSET$(1)$




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

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

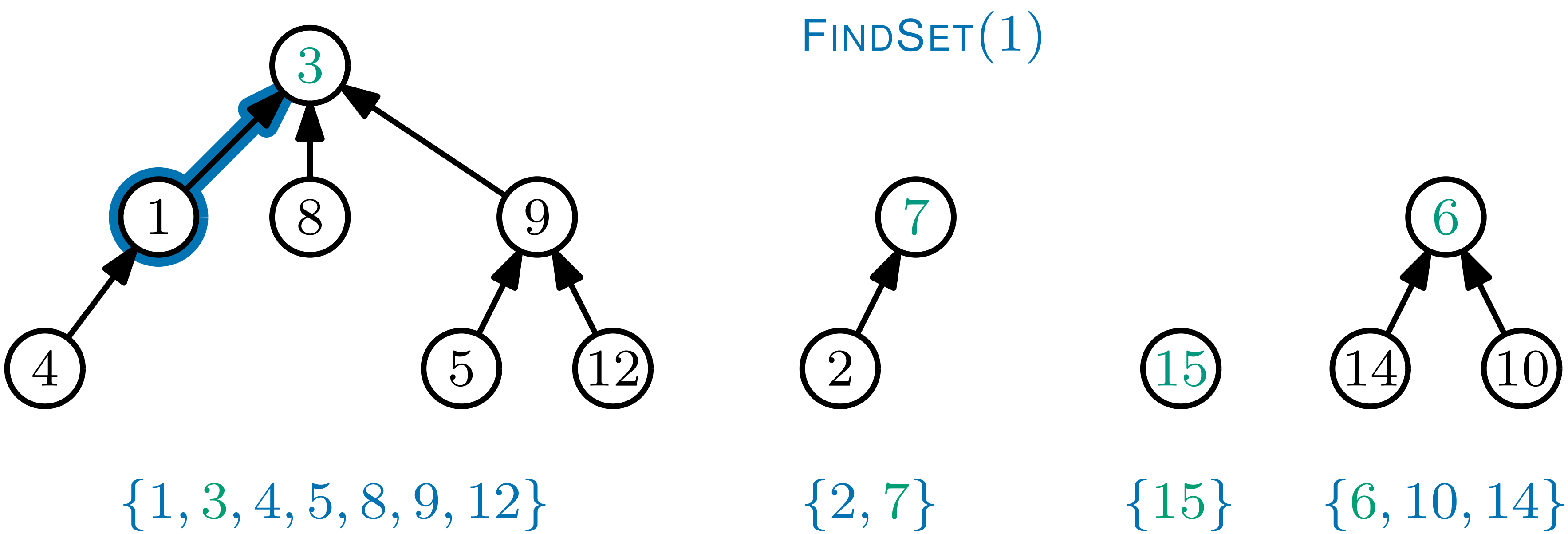
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FindSet operation

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




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

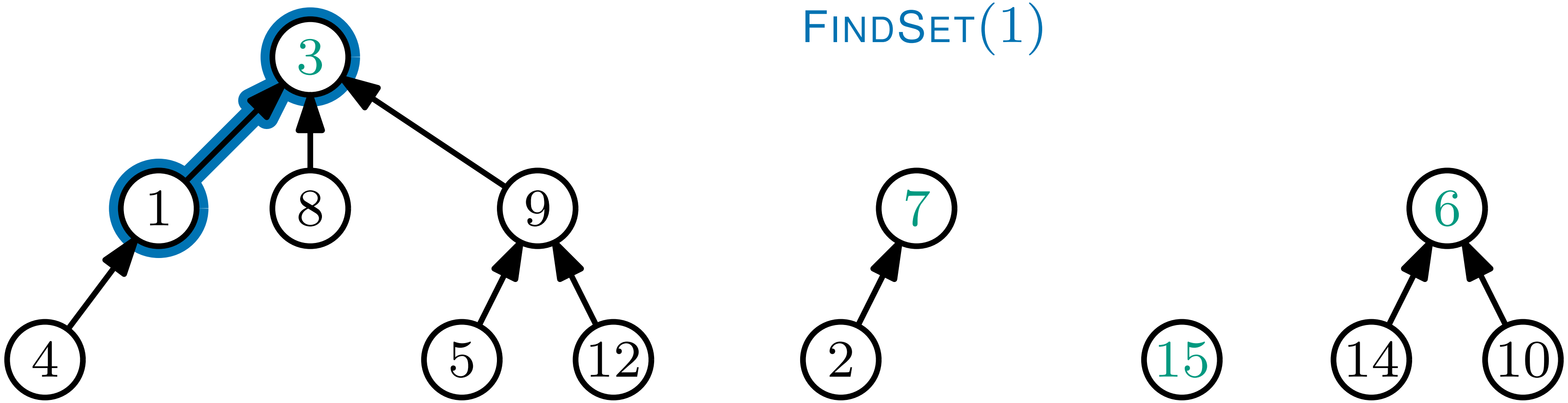
**Step 2:** Until you are at the root,

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 $x$




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

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

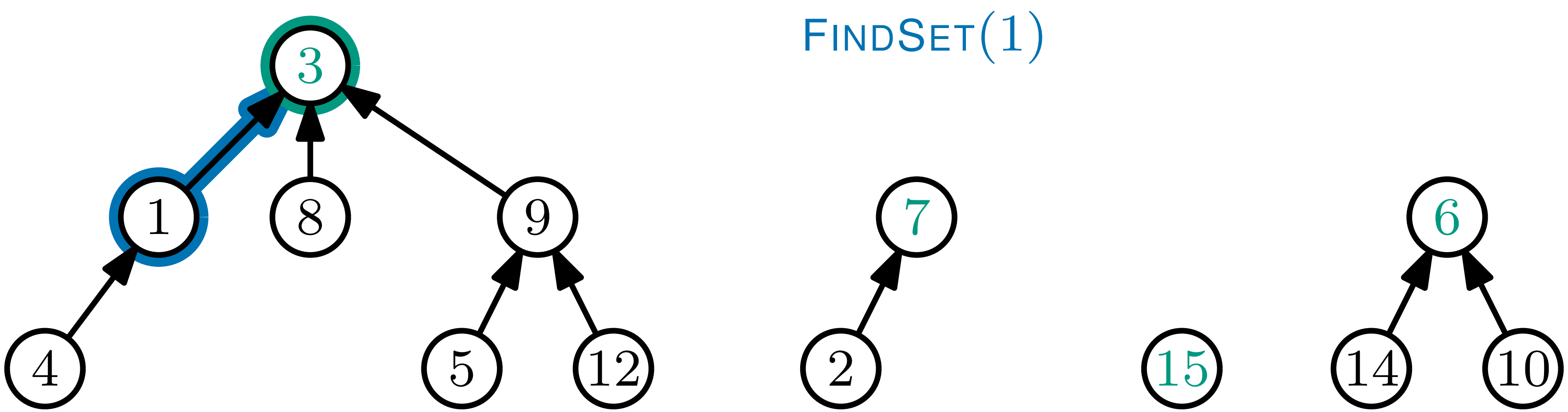
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FindSet operation

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




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

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

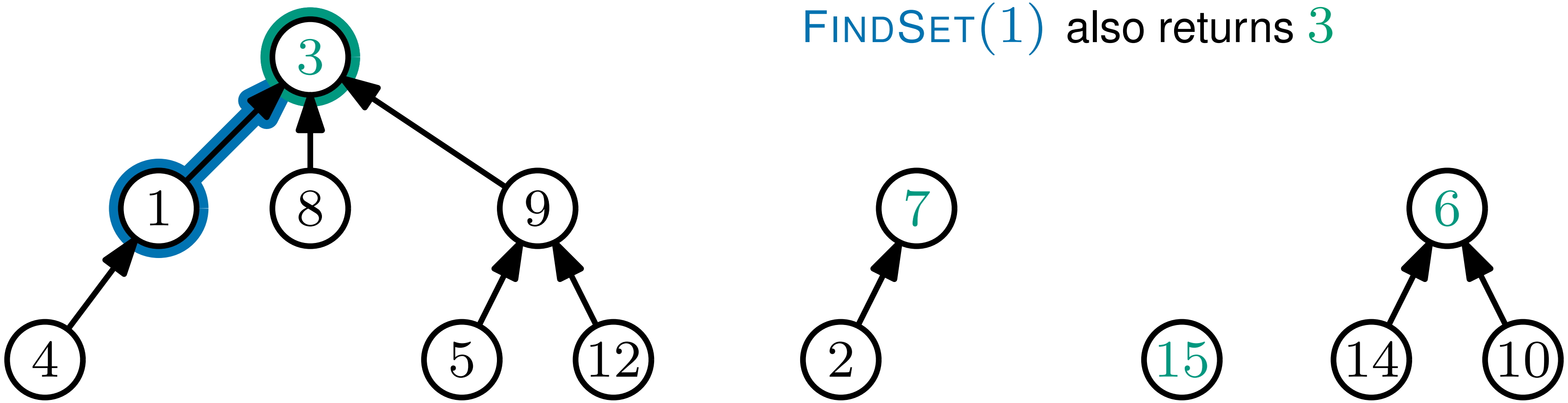
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FINDSET operation

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




FINDSET$(1)$ also returns $3$

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

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

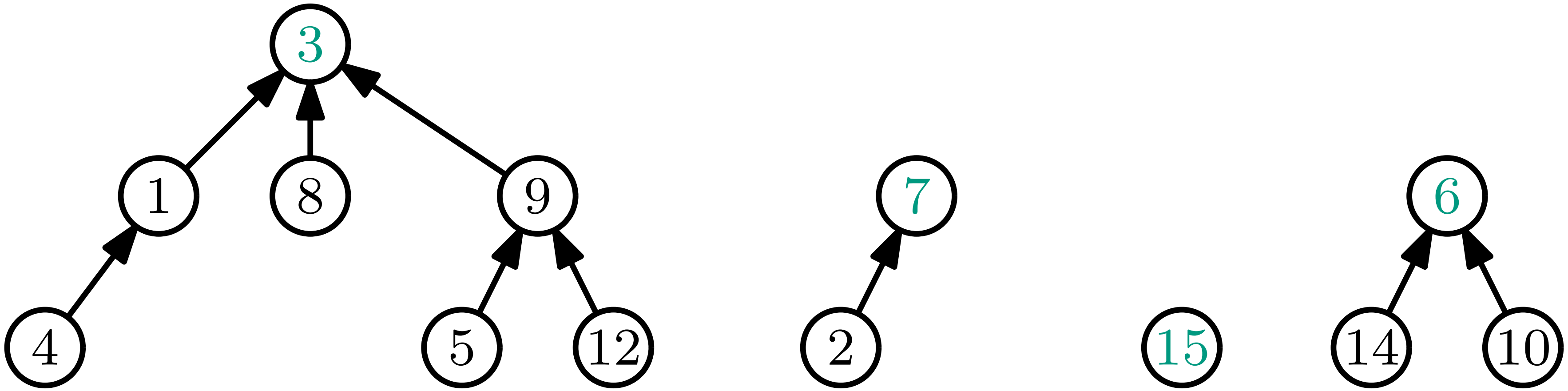
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FINDSET operation

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




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

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

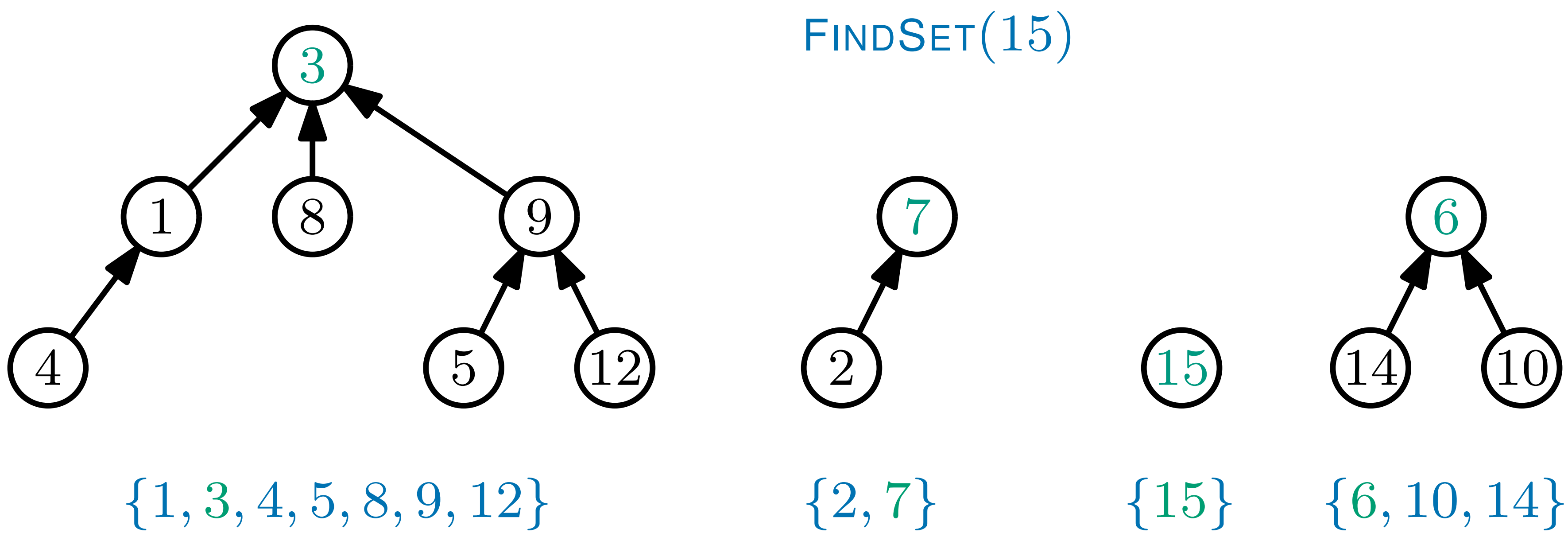
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# 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,

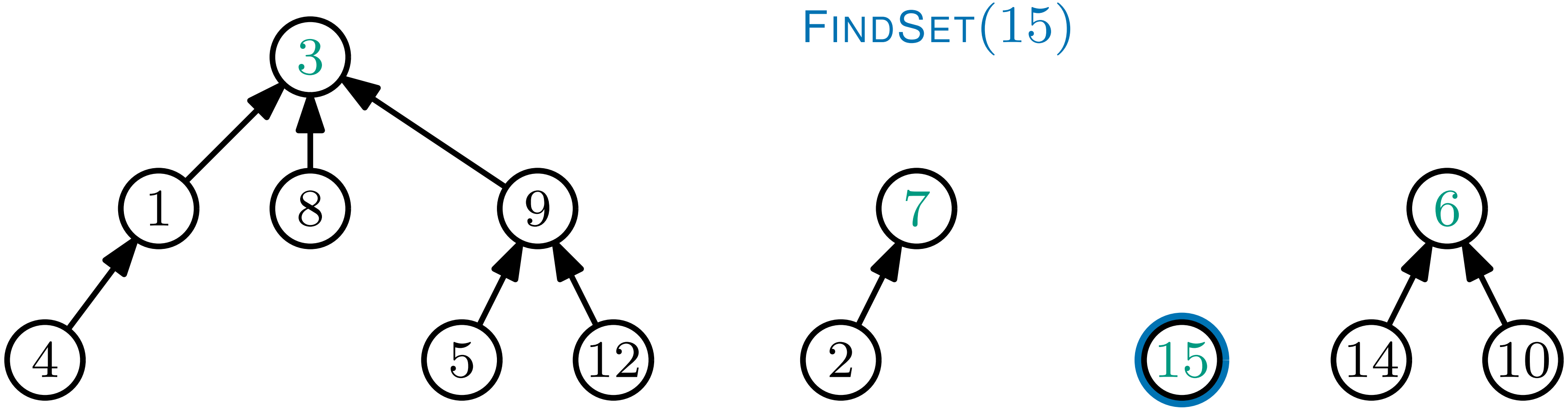
follow the pointer to the parent of the current node

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

# The FindSet operation

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

FindSet$(15)$




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

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

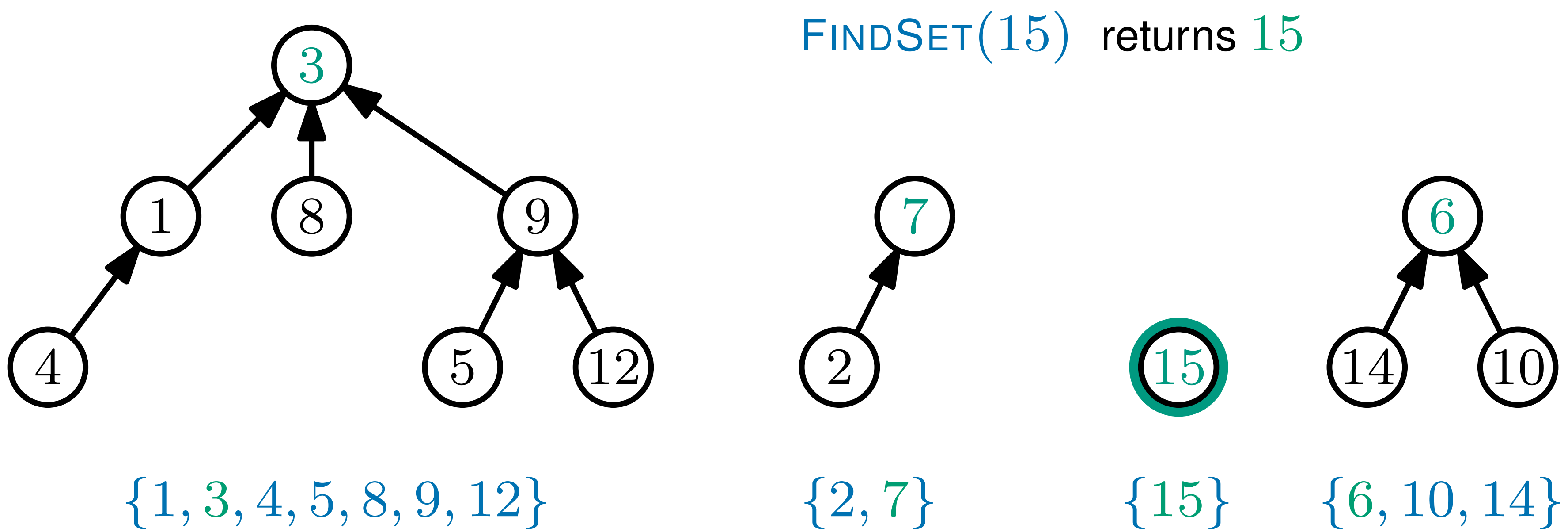
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FindSet operation

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




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

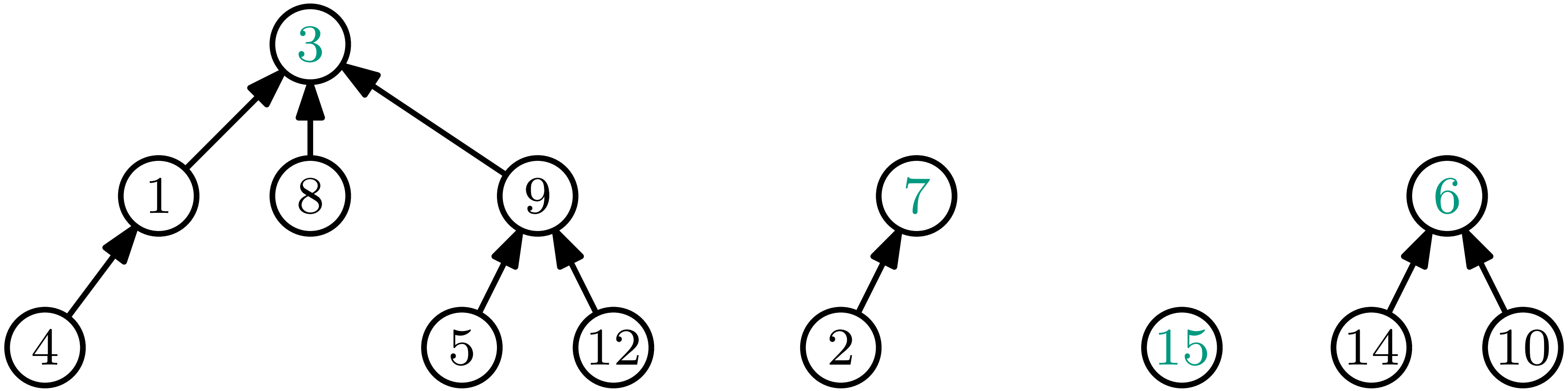
**Step 2:** Until you are at the root,

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 $x$



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

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

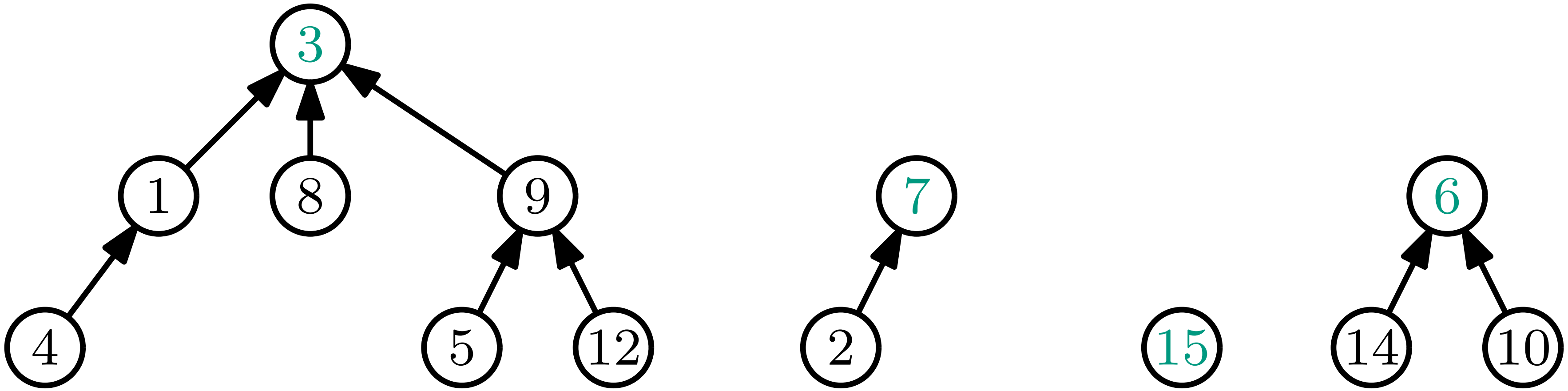
**Step 2:** Until you are at the root,

follow the pointer to the parent of the current node

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

# The FINDSET operation

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



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

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

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

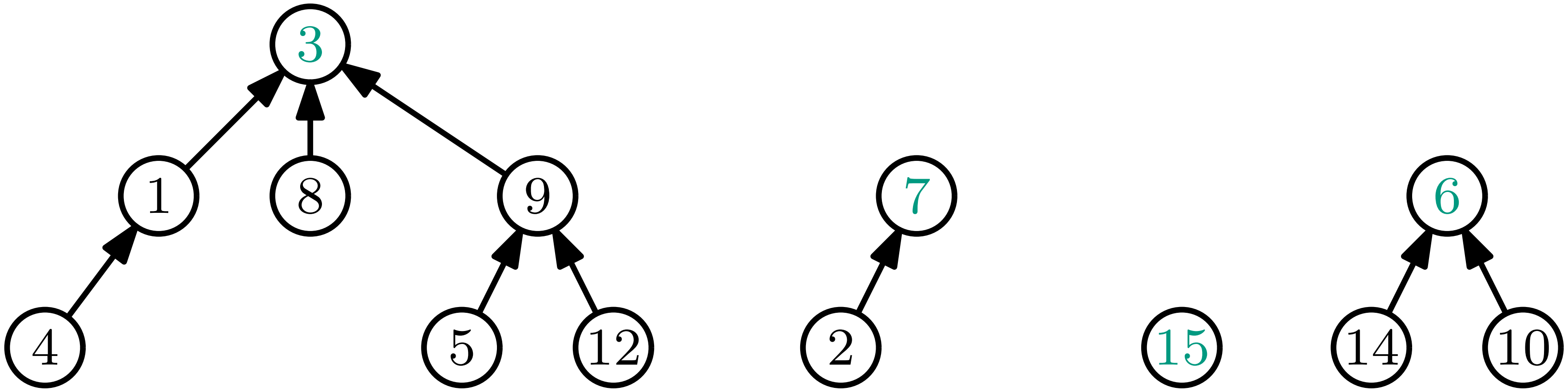
follow the pointer to the parent of the current node

**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 $x$



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

$O(1)$ time

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

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

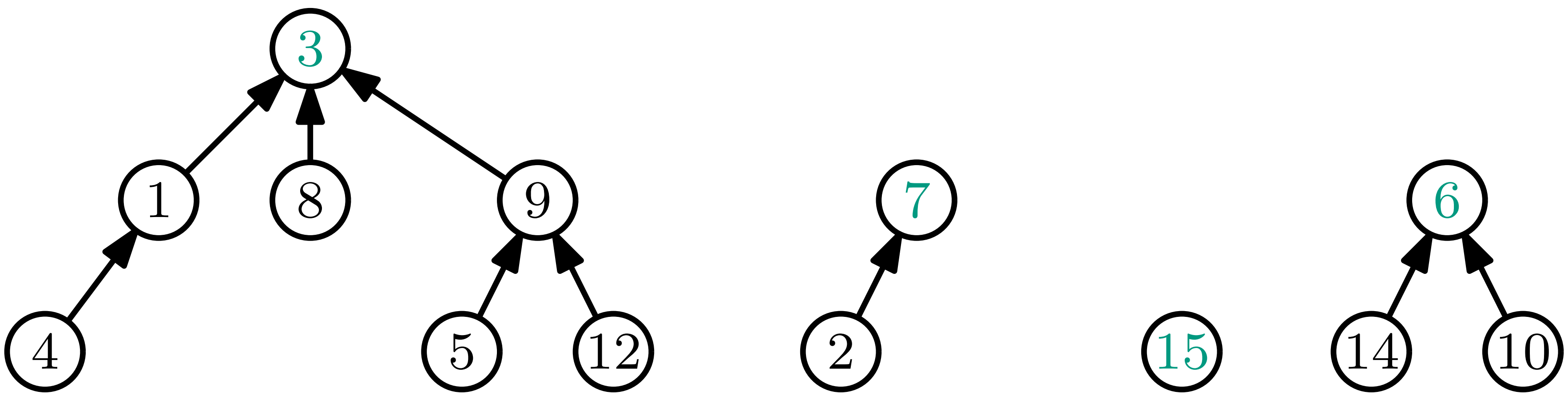
follow the pointer to the parent of the current node

**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 $x$




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

$O(1)$ time

*because $x$ is stored in $A[x]$*

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

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

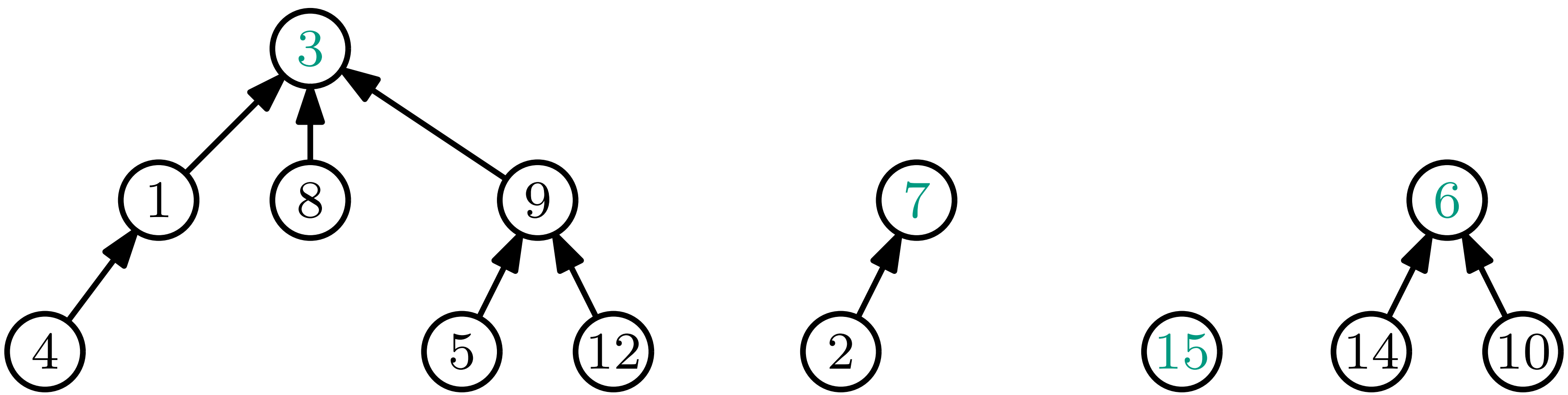
follow the pointer to the parent of the current node

**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 $x$



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

$O(1)$ time

*because $x$ is stored in $A[x]$*

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

$O(h)$ time

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

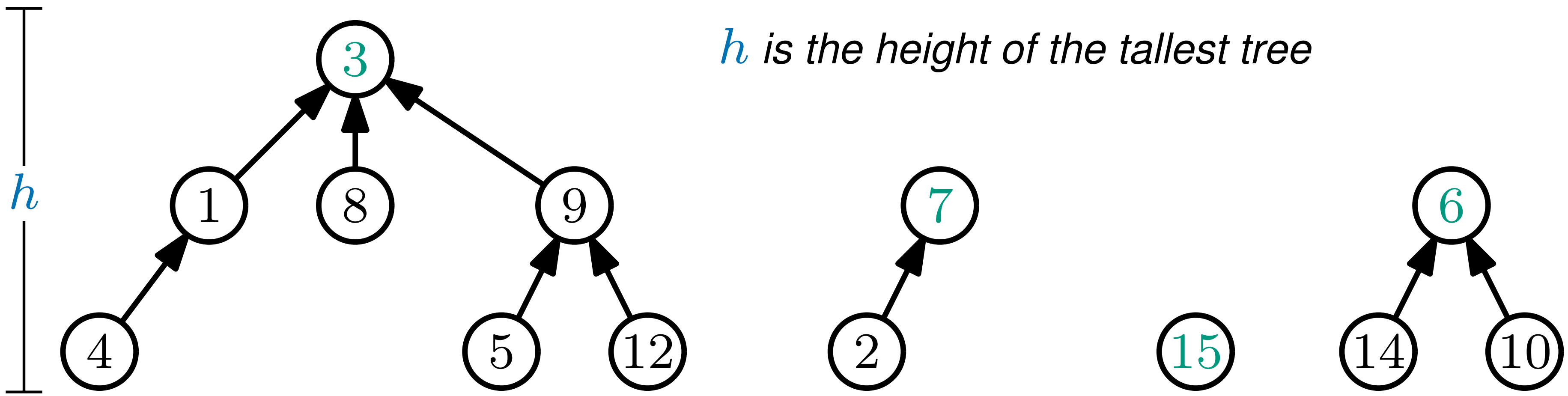
follow the pointer to the parent of the current node

**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 $x$




*$h$ is the height of the tallest tree*

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

$O(1)$ time

*because $x$ is stored in $A[x]$*

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

$O(h)$ time

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

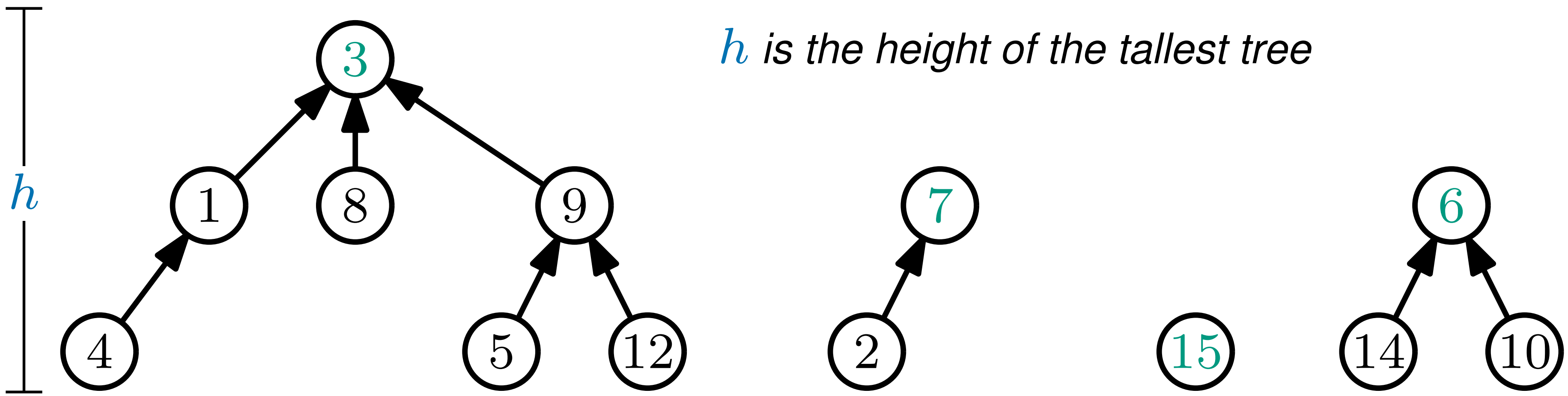
follow the pointer to the parent of the current node

**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 $x$




*$h$ is the height of the tallest tree*

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

$O(1)$ time

*because $x$ is stored in $A[x]$*

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

$O(h)$ time

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

follow the pointer to the parent of the current node

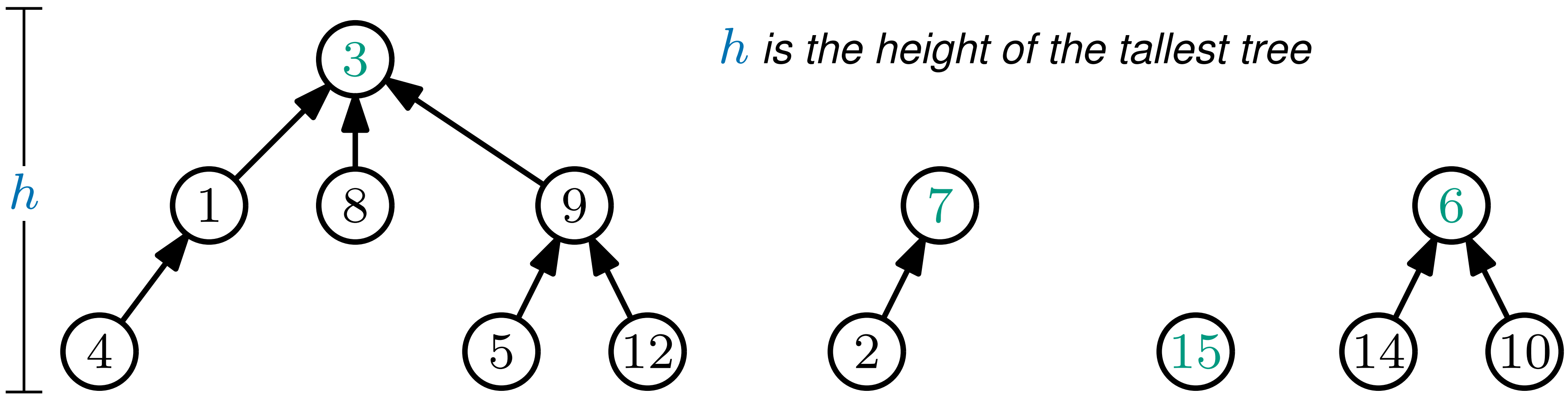
**Step 3:** Output the element at the root

$O(1)$ time

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

# The FINDSET operation

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




*$h$ is the height of the tallest tree*

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

$O(1)$ time

*because $x$ is stored in $A[x]$*

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

$O(h)$ time

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

follow the pointer to the parent of the current node

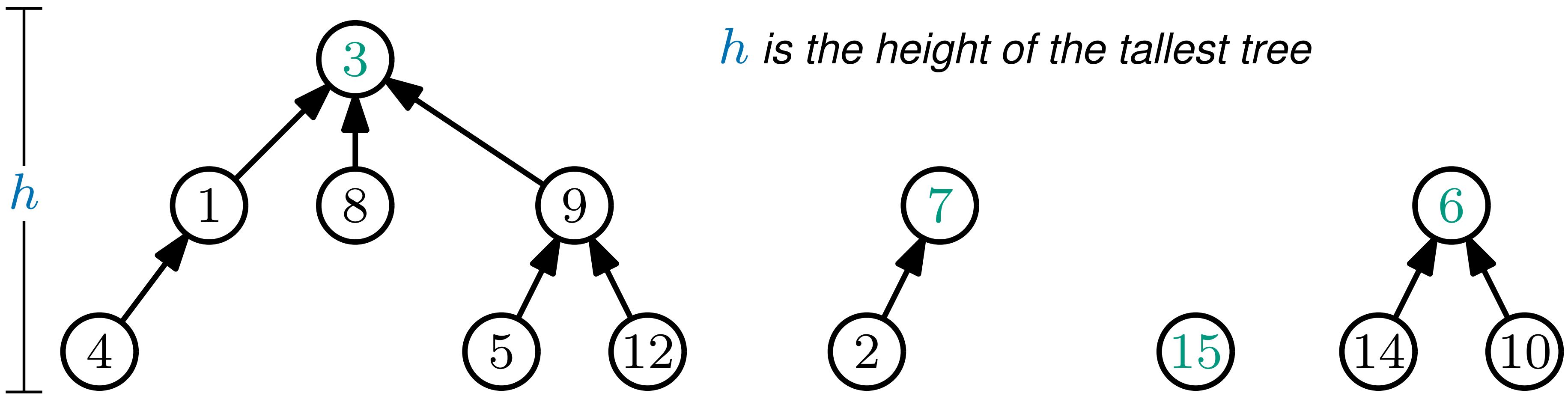
**Step 3:** Output the element at the root

$O(1)$ time

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

# The FindSet operation

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




$h$ *is the height of the tallest tree*

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

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

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

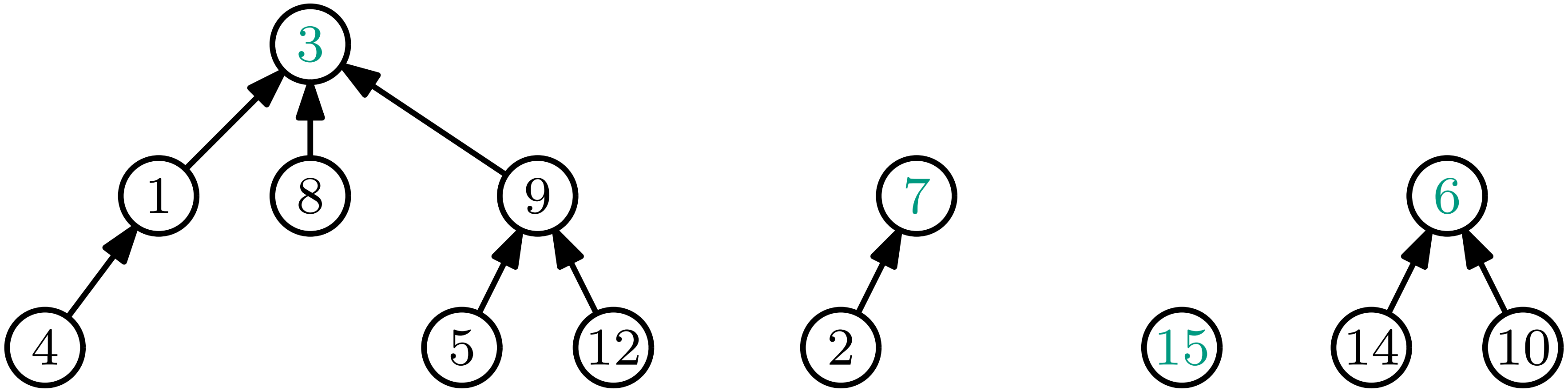
follow the pointer to the parent of the current node

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

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)



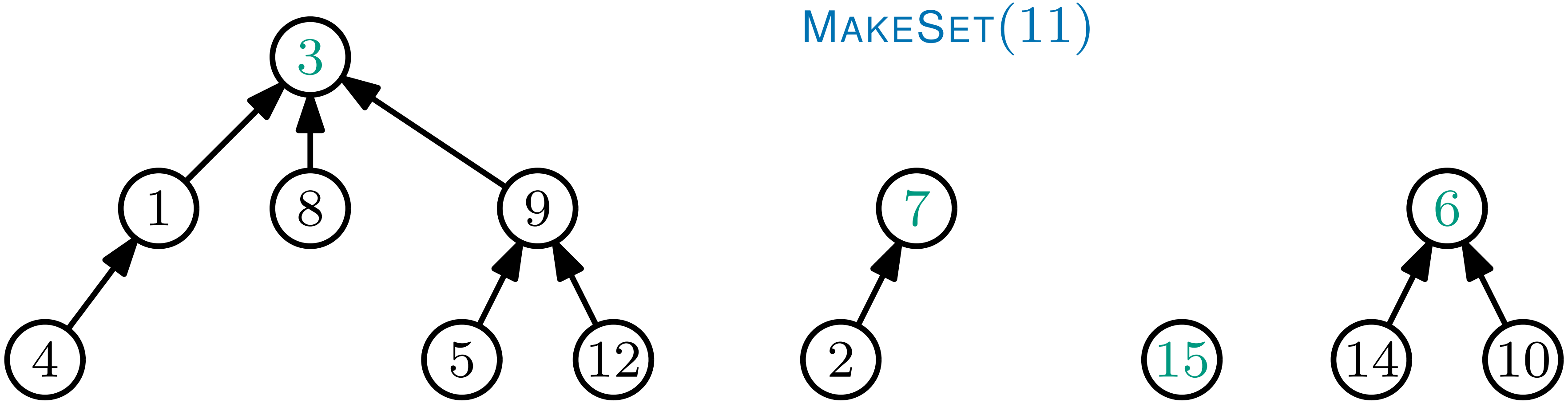
$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

# The MAKESET operation

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

MAKESET$(11)$

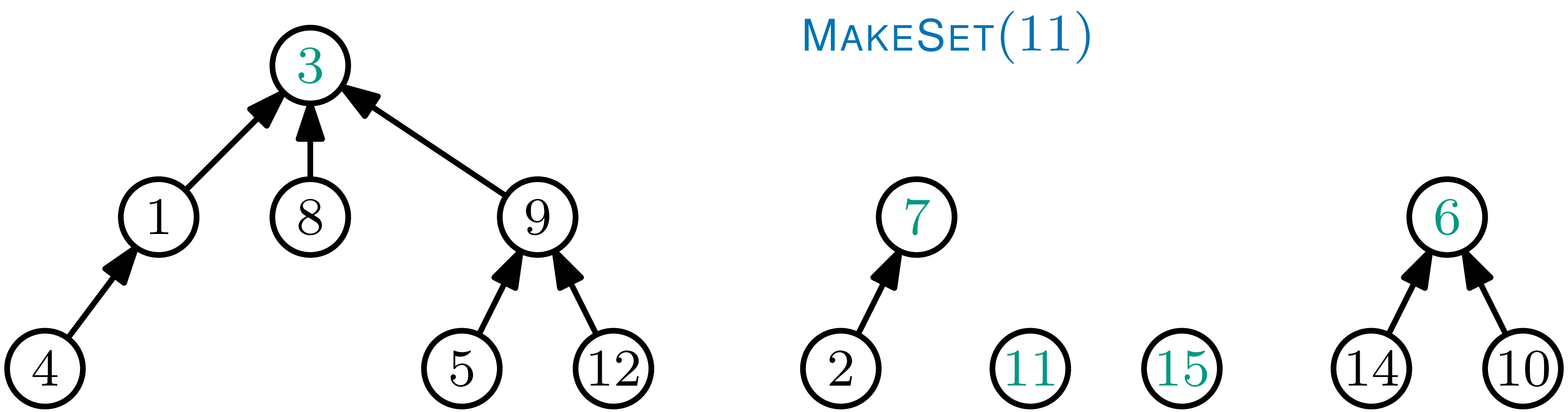



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{15\}$ $\{6, 10, 14\}$

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

# The MAKESET operation

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

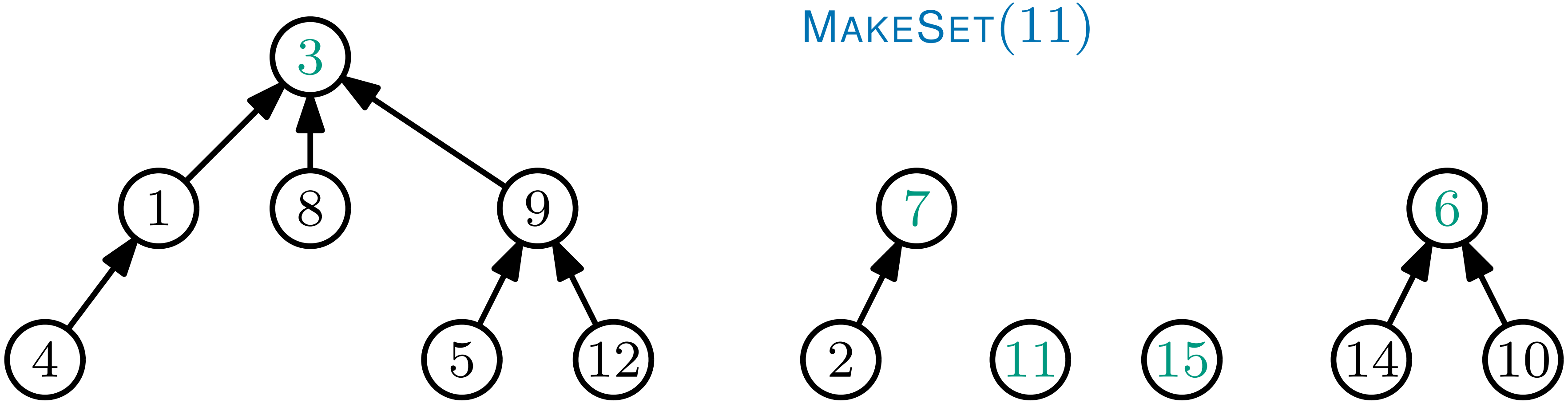



$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{11\}$ $\{15\}$ $\{6, 10, 14\}$

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

# The MAKESET operation

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



MAKESET$(11)$

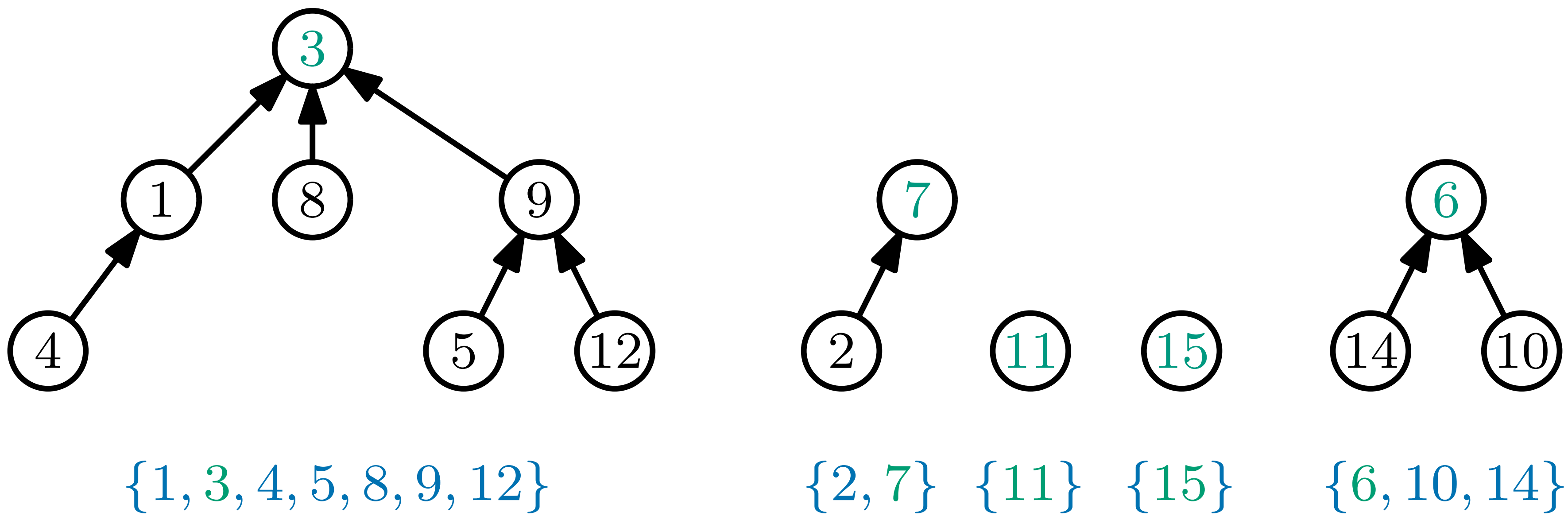
$\{1, 3, 4, 5, 8, 9, 12\}$ $\{2, 7\}$ $\{11\}$ $\{15\}$ $\{6, 10, 14\}$

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

*(that's it)*

# The MakeSet operation

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



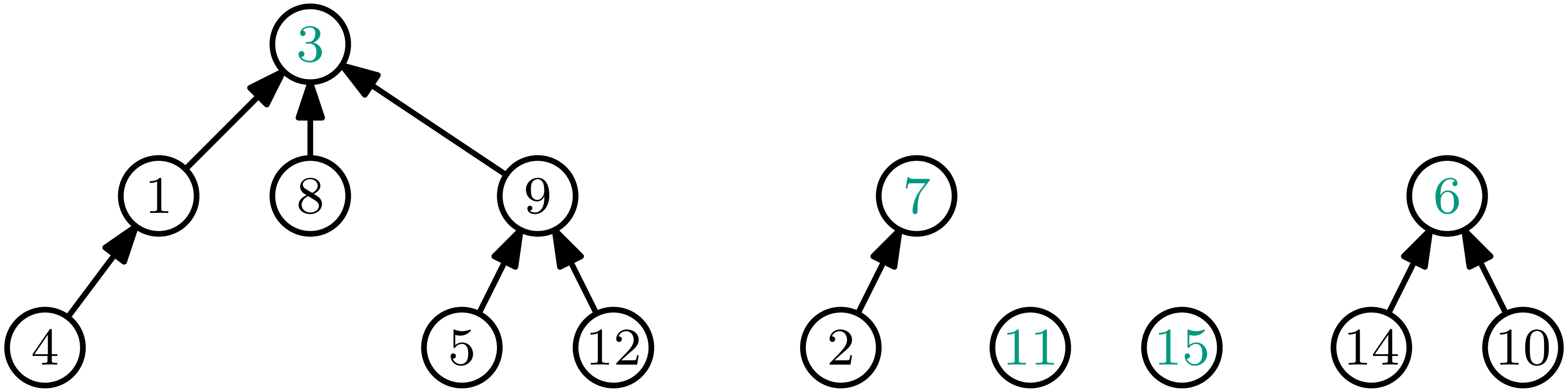

**Step 1:** Make a new tree containing $x$ as the root

*(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)




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

$O(1)$ time → **Step 1:** Make a new tree containing $x$ as the root

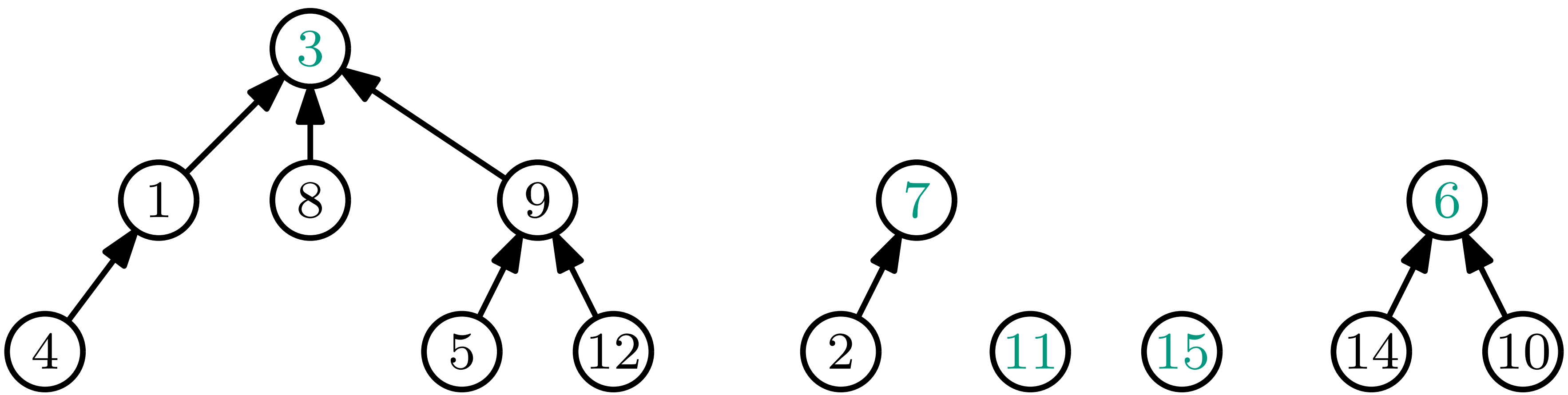
*(that's it)*

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

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

# The MAKESET operation

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




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

$O(1)$ time → **Step 1:** Make a new tree containing $x$ as the root

*(that's it)*

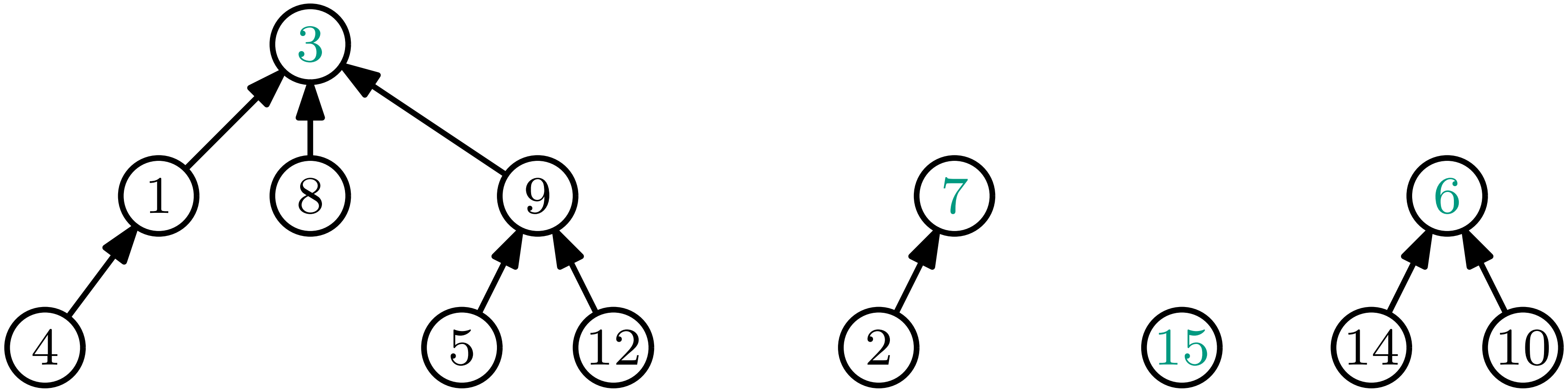
*because $x$ should be stored in $A[x]$*

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

The overall worst-case time complexity is $O(1)$

# 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 $x$

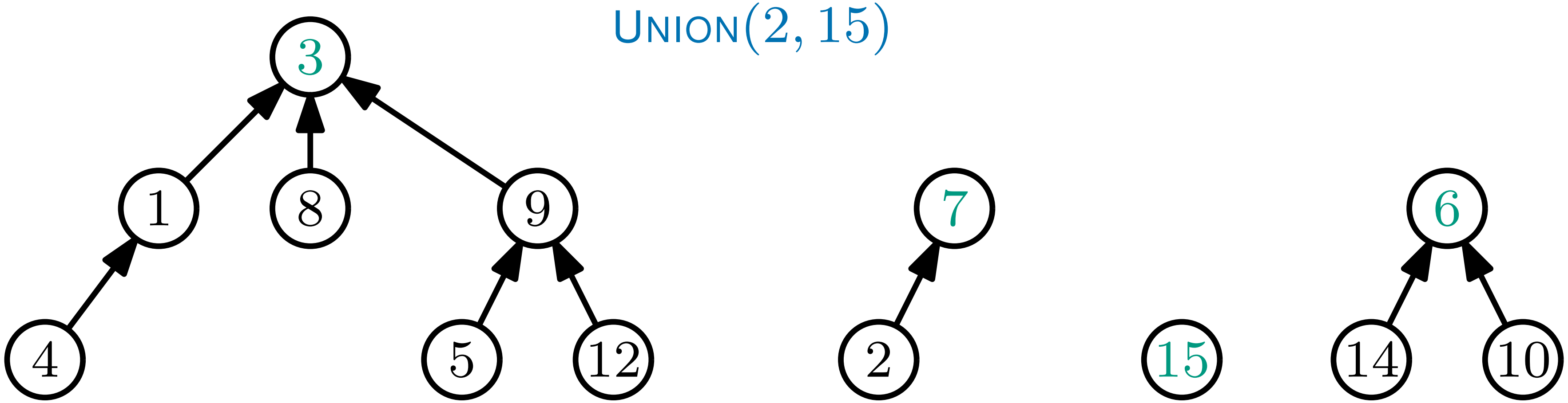
**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**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

UNION$(2, 15)$



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

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

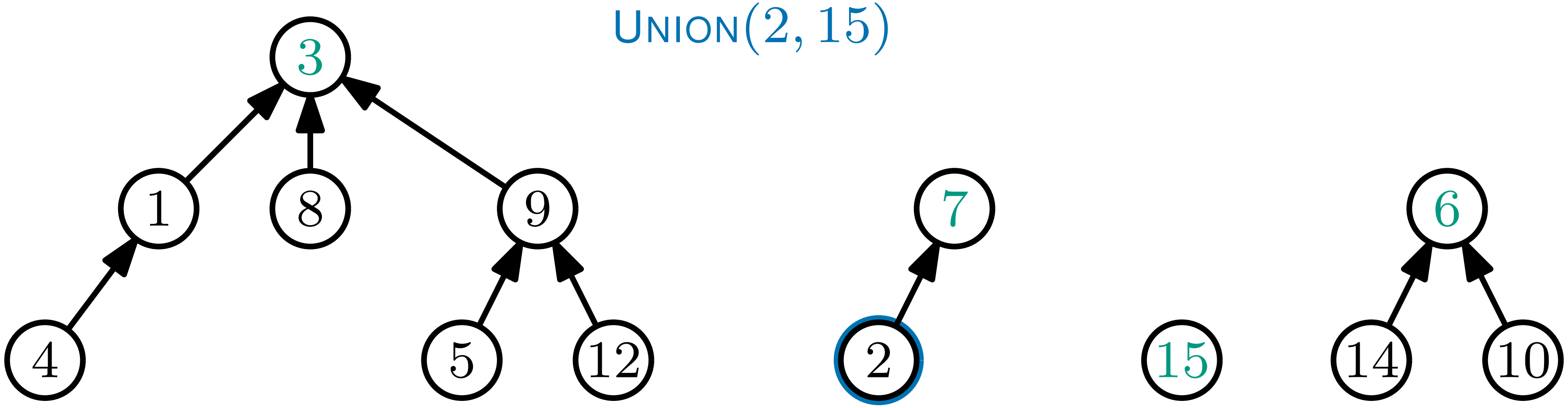
**Step 2:** Compute $r_y =$ FINDSET$(y)$ - the root of the tree containing $y$

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

# The UNION operation

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

UNION$(2, 15)$




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

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

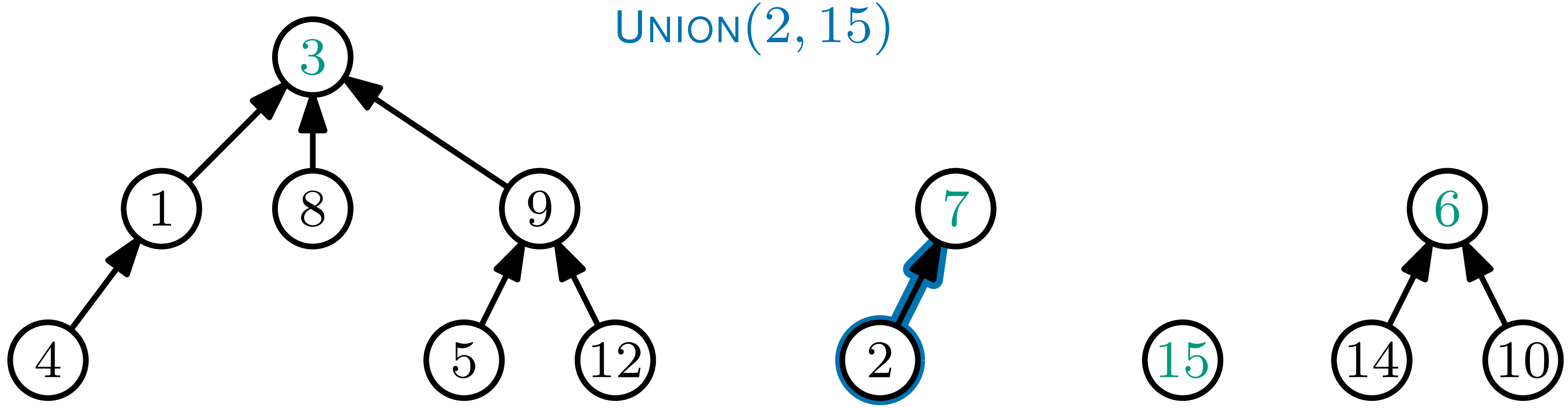
**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**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

UNION$(2, 15)$




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

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

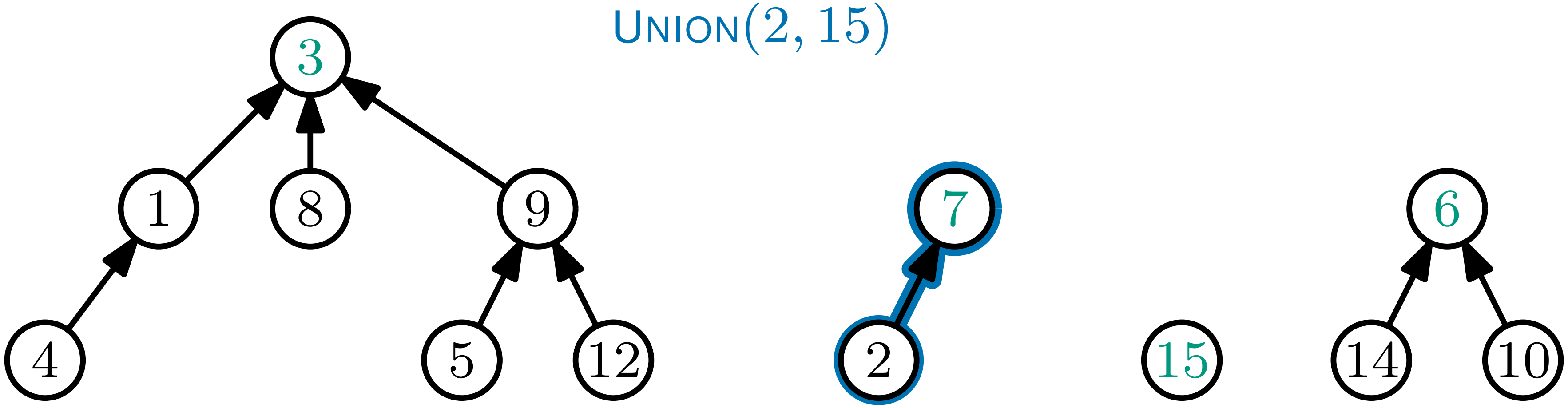
**Step 2:** Compute $r_y =$ FINDSET$(y)$ - the root of the tree containing $y$

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

# The UNION operation

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

UNION$(2, 15)$




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

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

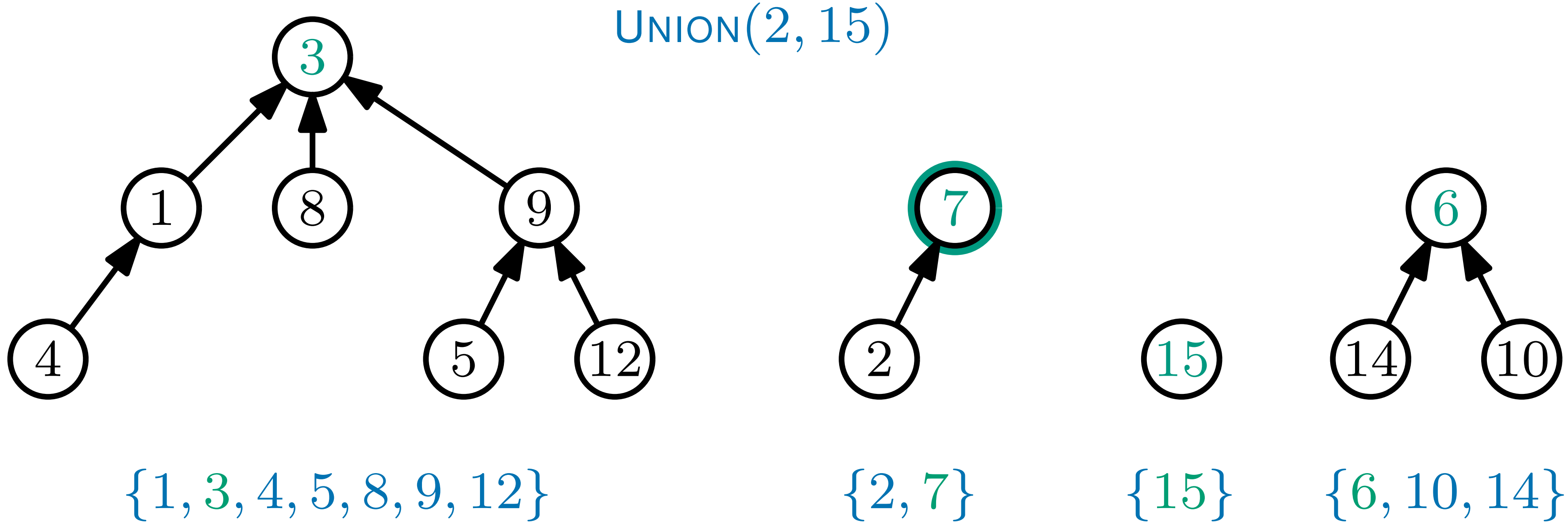
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

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

# The UNION operation

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

UNION$(2, 15)$




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

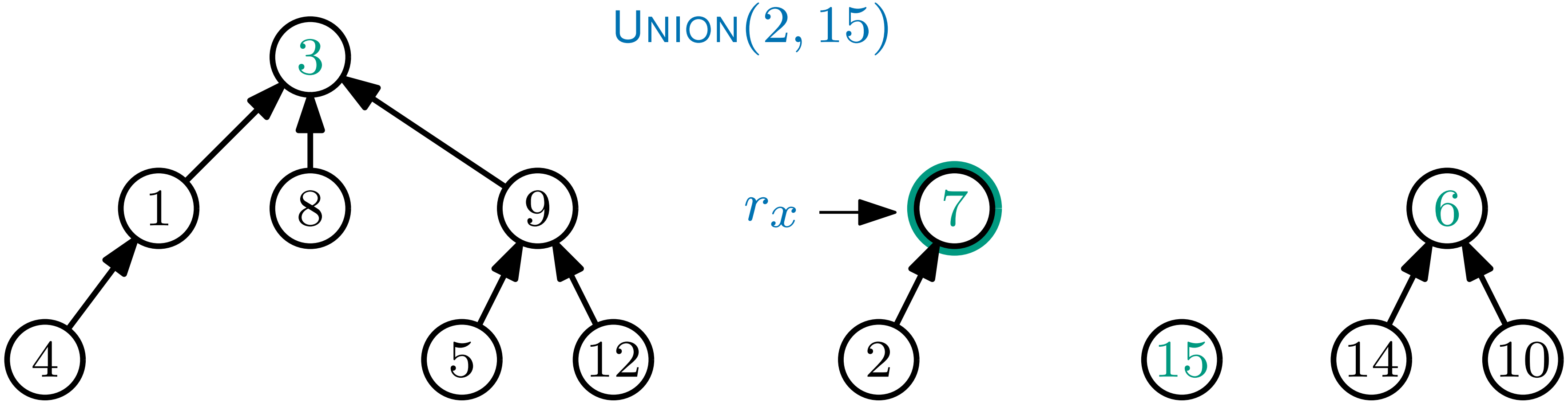
**Step 2:** Compute $r_y =$ FINDSET$(y)$ - the root of the tree containing $y$

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

# The UNION operation

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

$\textsc{Union}(2, 15)$




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

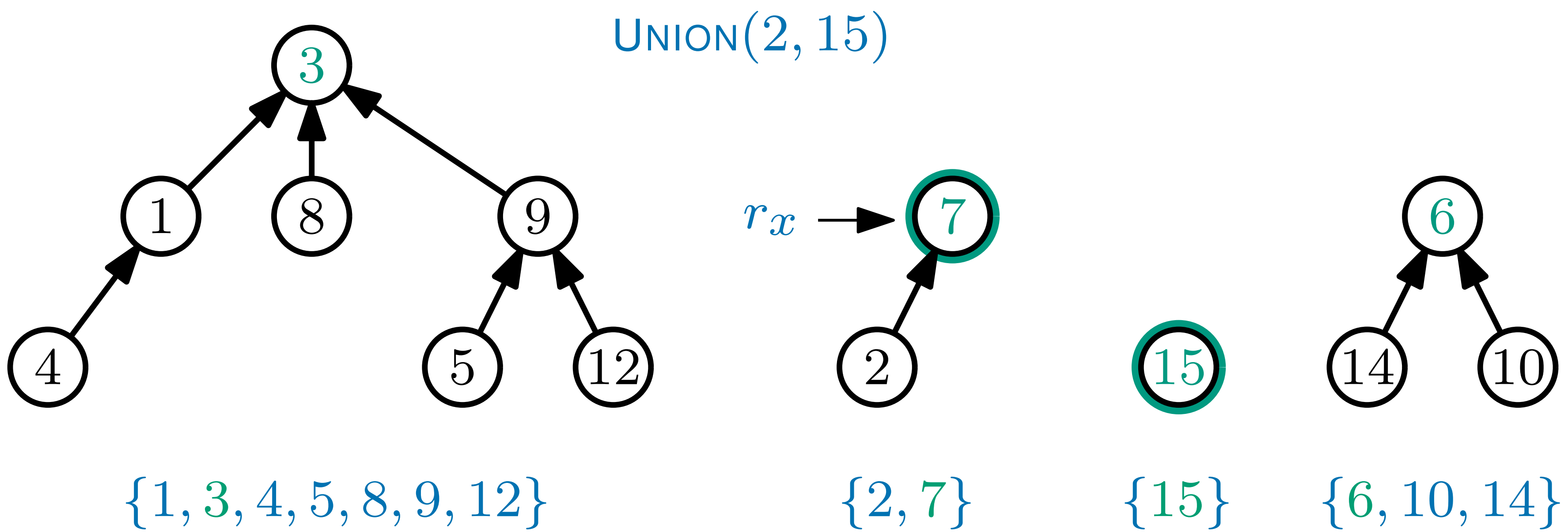
**Step 1:** Compute $r_x = \textsc{FindSet}(x)$ - the root of the tree containing $x$

**Step 2:** Compute $r_y = \textsc{FindSet}(y)$ - 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



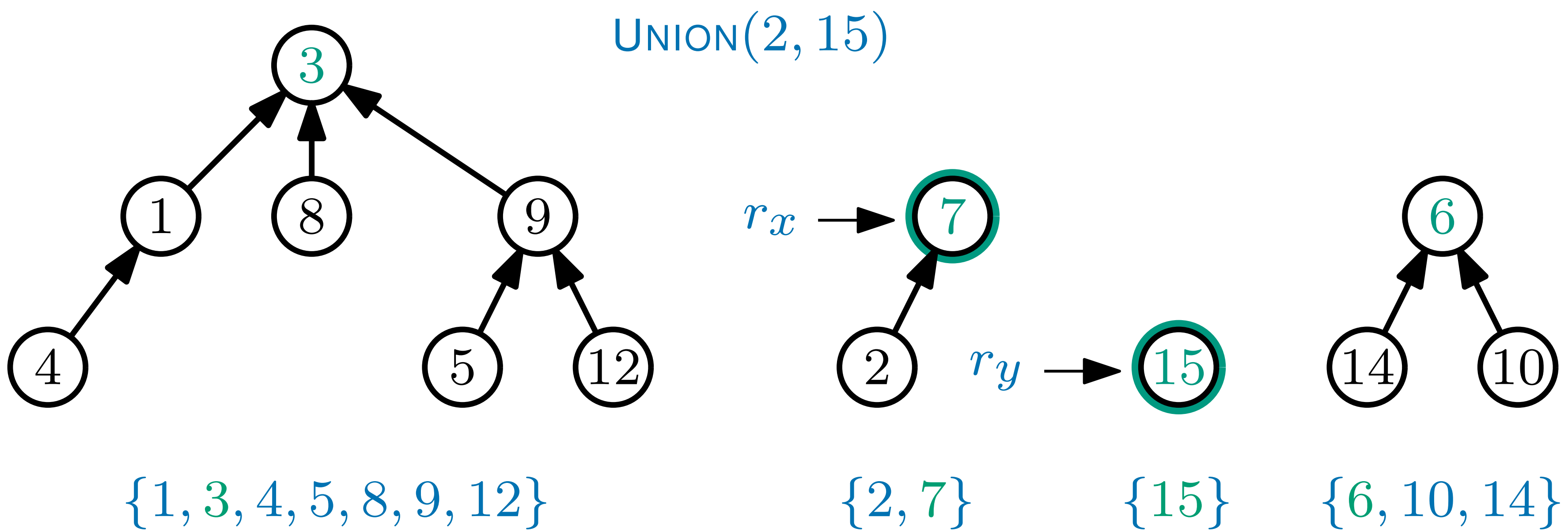

**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**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



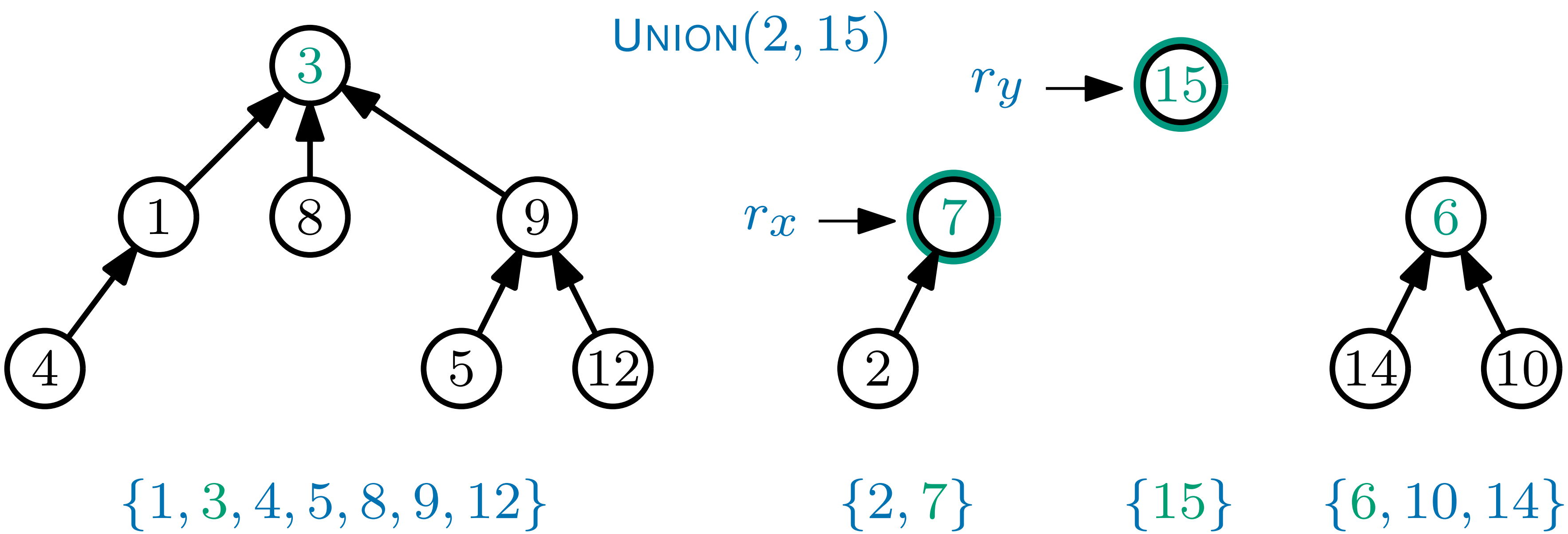

**Step 1:** Compute $r_x =$ FINDSET$(x)$ - the root of the tree containing $x$

**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)*

# The UNION operation

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



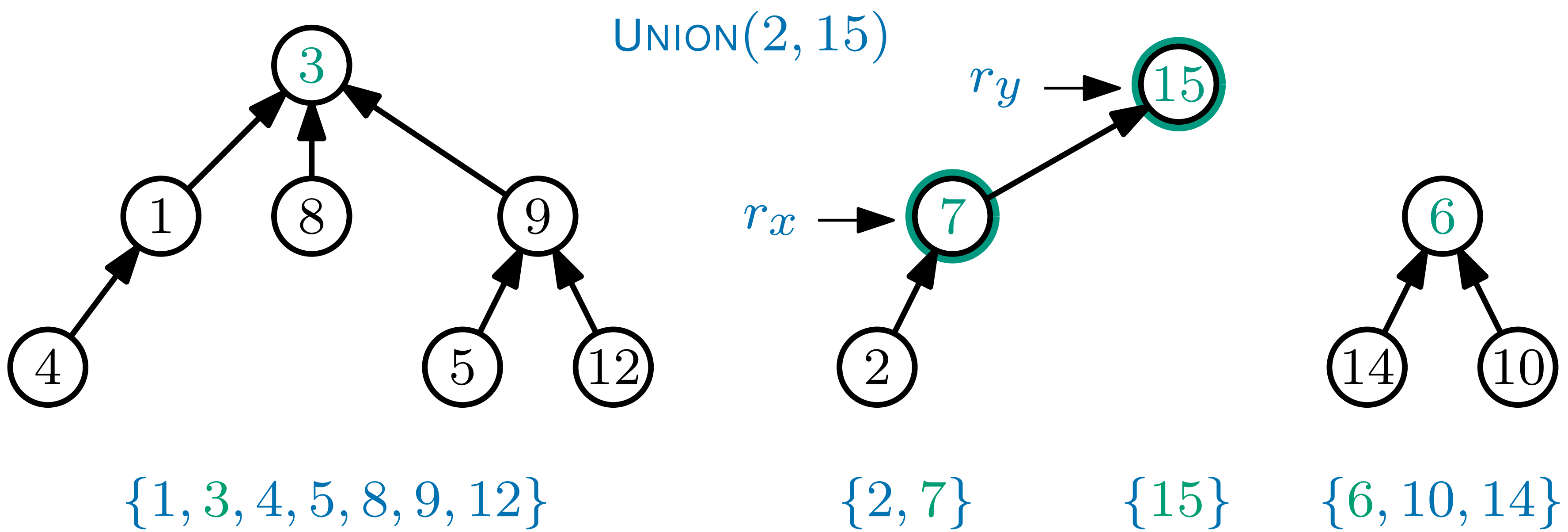

**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**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



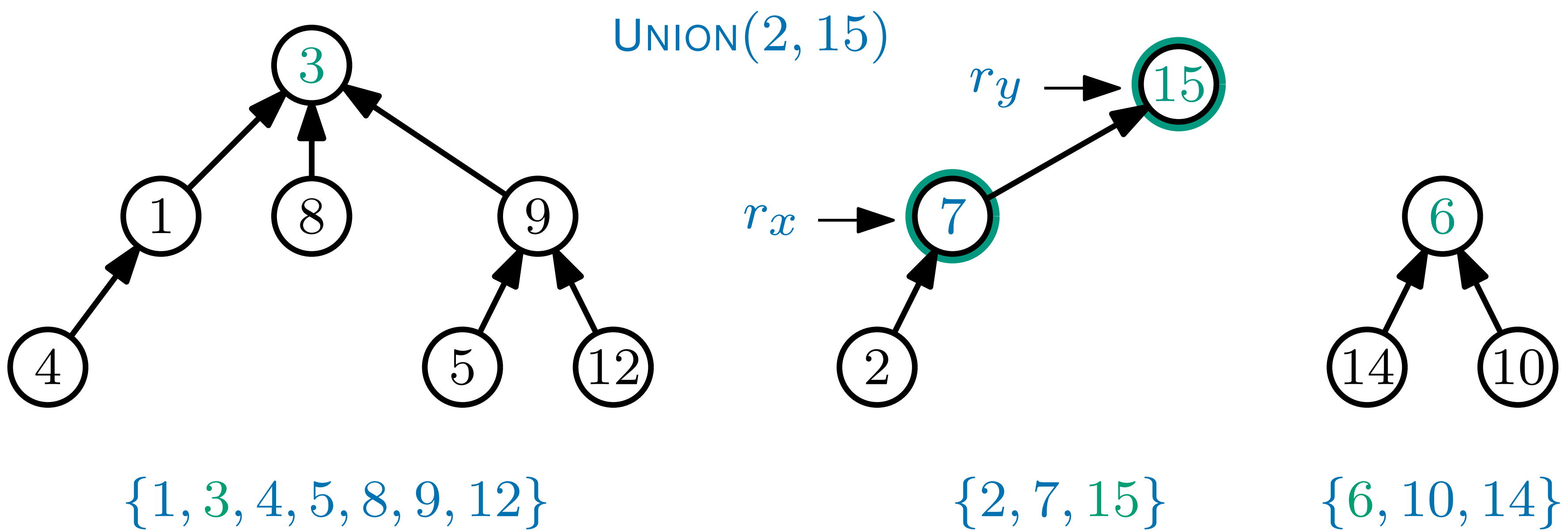

**Step 1:** Compute $r_x =$ FINDSET$(x)$ - the root of the tree containing $x$

**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)*

# The UNION operation

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



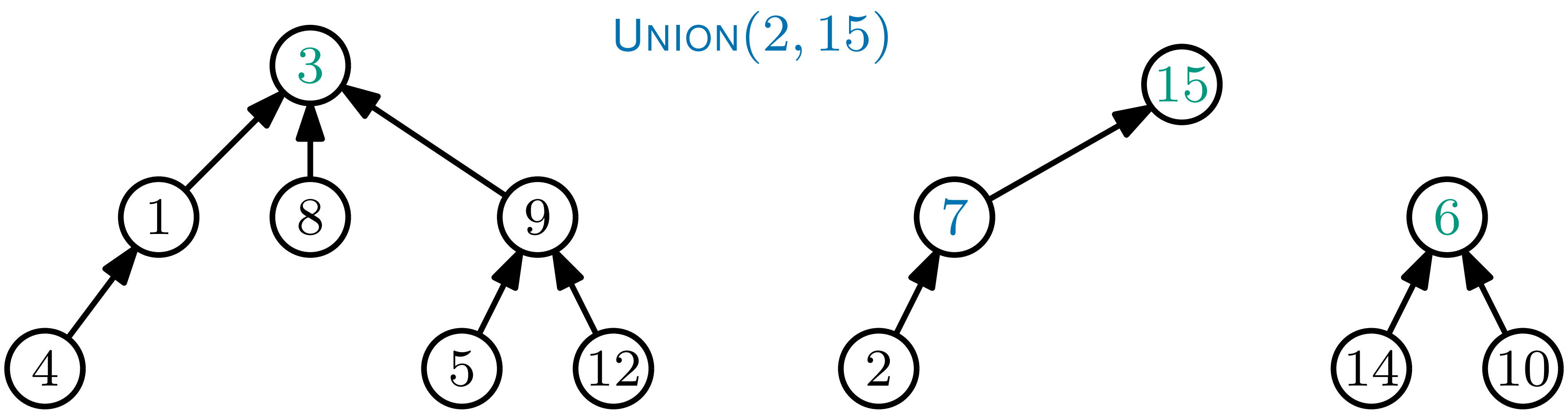

**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**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




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

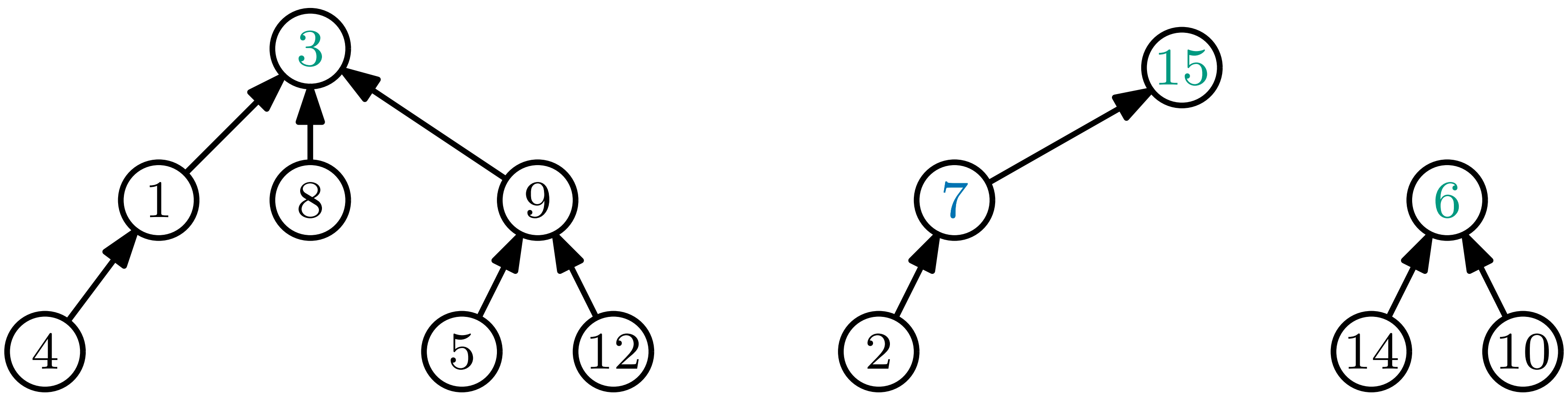
**Step 1:** Compute $r_x =$ FINDSET$(x)$ - the root of the tree containing $x$

**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)*

# The UNION operation

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




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

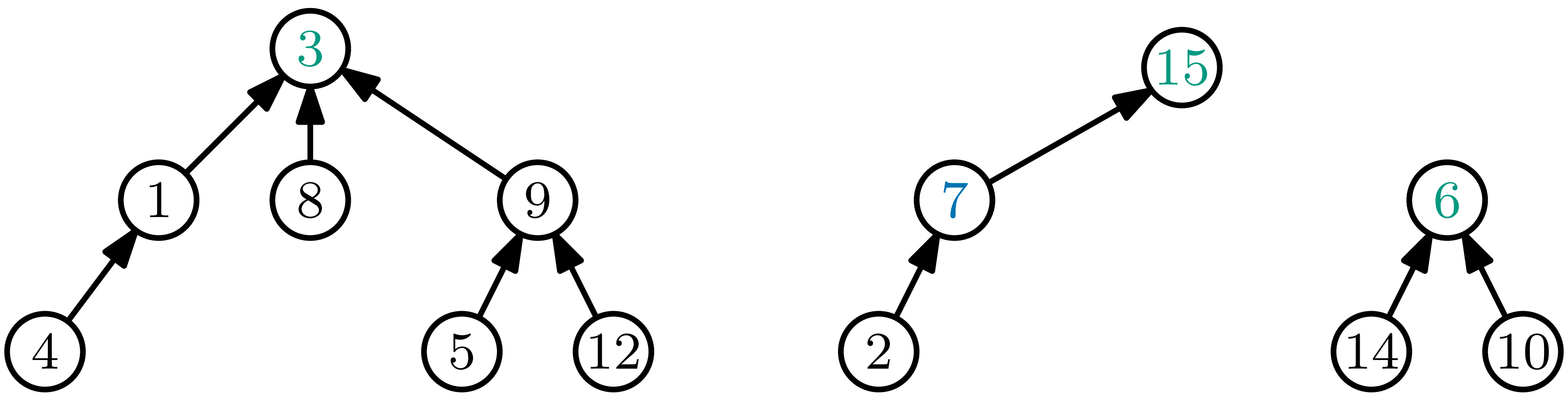
**Step 1:** Compute $r_x = \text{FINDSET}(x)$ - the root of the tree containing $x$

**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




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

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

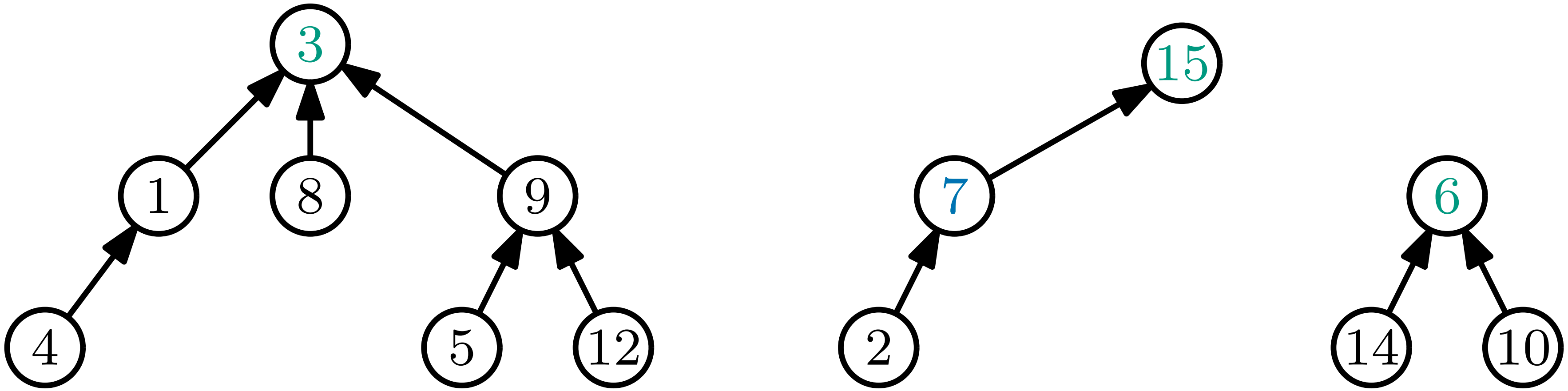
**Step 2:** Compute $r_y = \textsc{FindSet}(y)$ - the root of the tree containing $y$

**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




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

$O(h)$ time

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

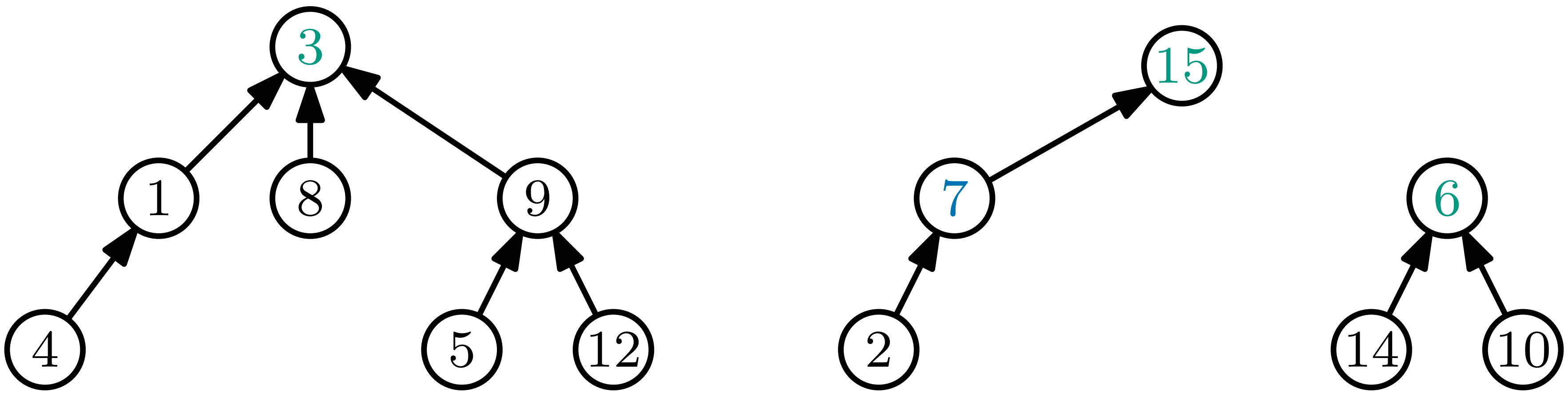
**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**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




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

$O(h)$ time

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

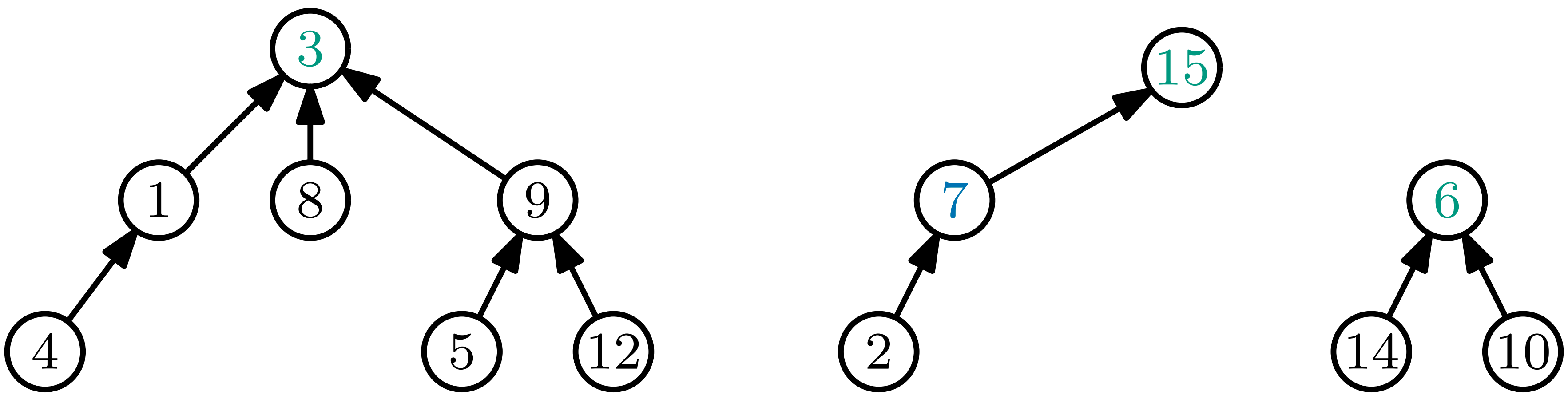
**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**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

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




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

$O(h)$ time

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

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

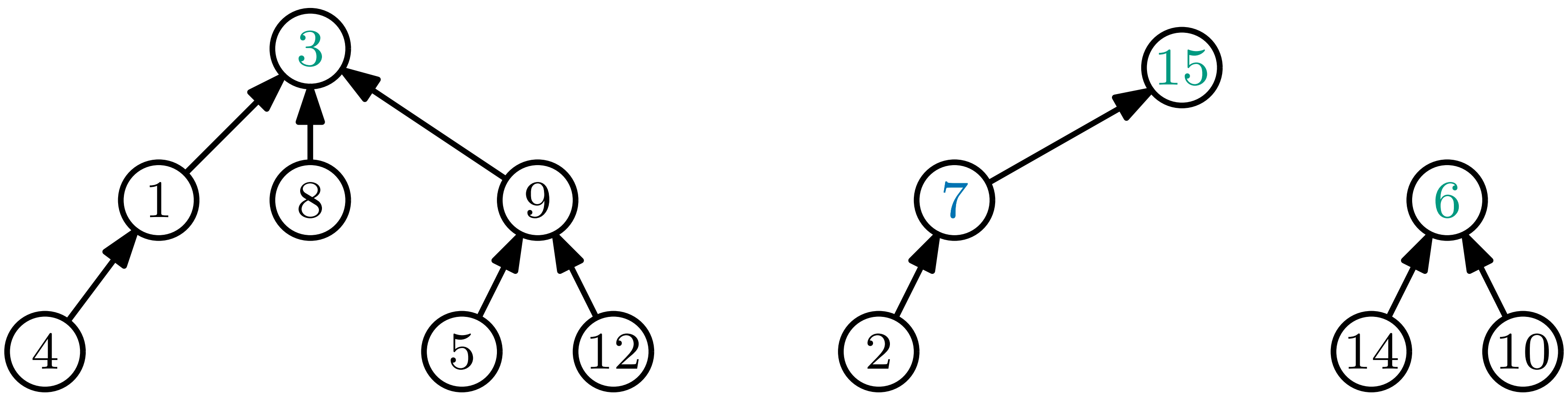
$O(1)$ time

**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

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




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

$O(h)$ time

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

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

$O(1)$ time

**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?*

it's $O(h)$ again

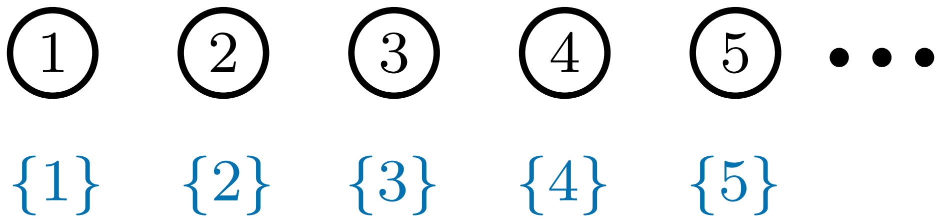
# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

## How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one. . .

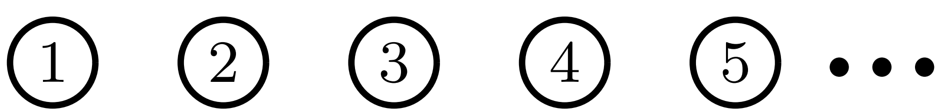
Consider the following sets:



## How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:



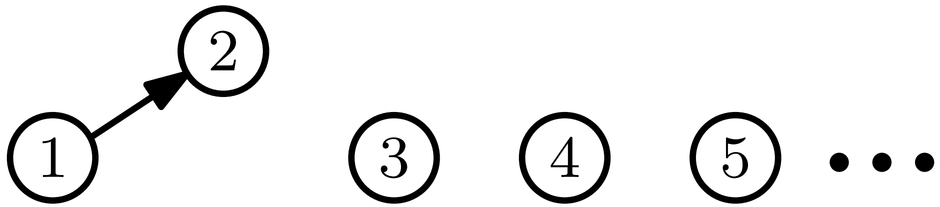

$\{1\}$ $\{2\}$ $\{3\}$ $\{4\}$ $\{5\}$

Now perform UNION$(1, 2)$

# How high does the sycamore grow?

Unfortunately, every UNION operation could
increase the tallest tree height, $h$ by one...

Consider the following sets:



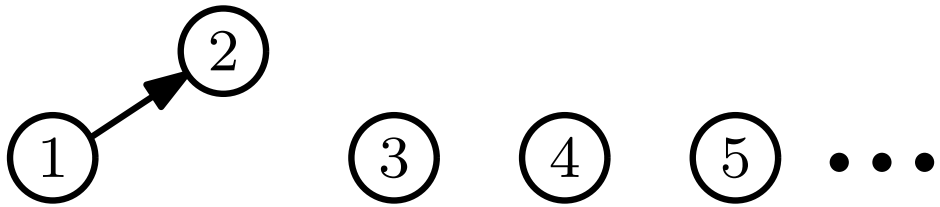

$\{1, 2\}$ $\{3\}$ $\{4\}$ $\{5\}$

Now perform UNION$(1, 2)$

# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:

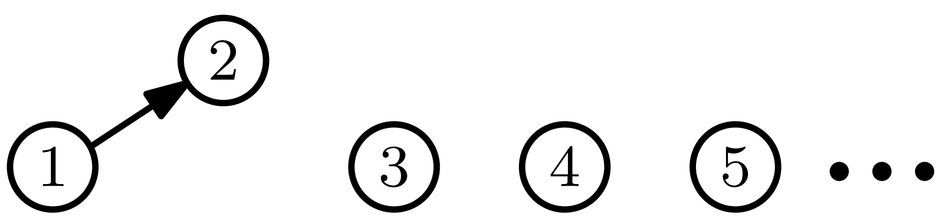


$\{1, 2\}$ $\{3\}$ $\{4\}$ $\{5\}$

# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:



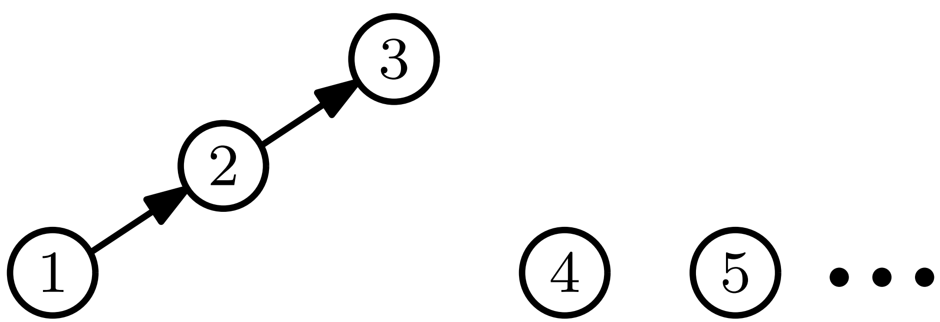

$\{1, 2\}$ $\{3\}$ $\{4\}$ $\{5\}$

Now perform UNION$(1, 3)$

# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:



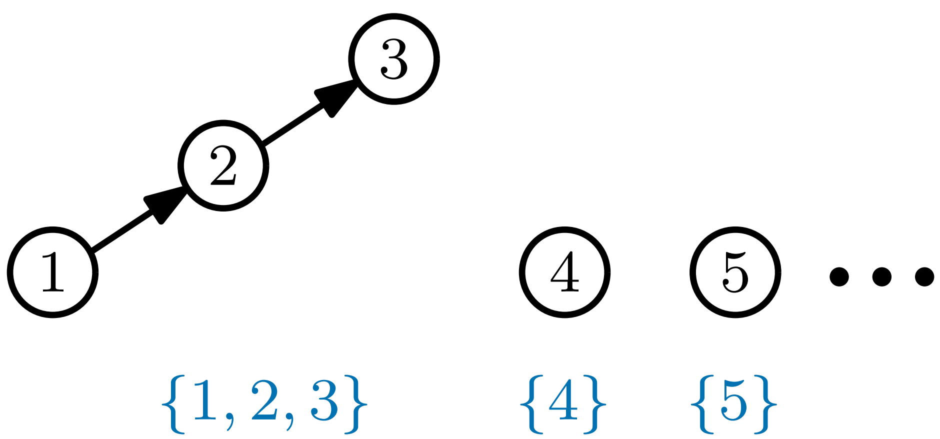
$\{1, 2, 3\}$ $\{4\}$ $\{5\}$

Now perform UNION$(1, 3)$

# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

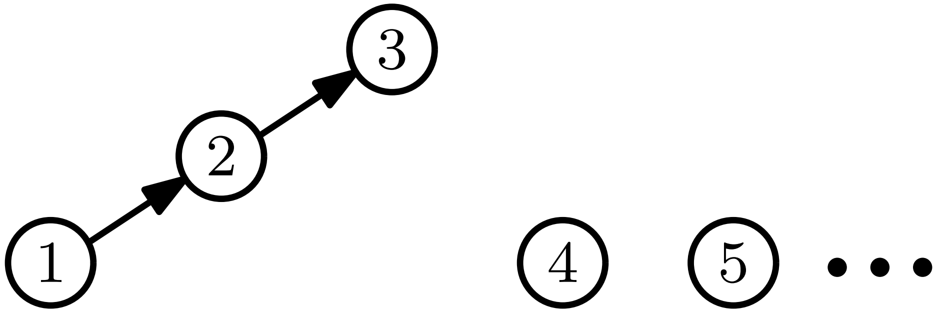
Consider the following sets:



## How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:



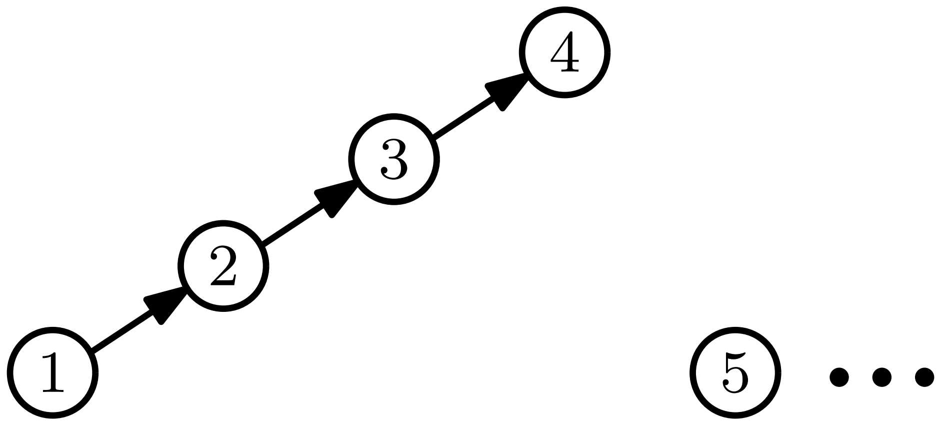

$\{1, 2, 3\}$ $\{4\}$ $\{5\}$

Now perform UNION$(1, 4)$

## How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:



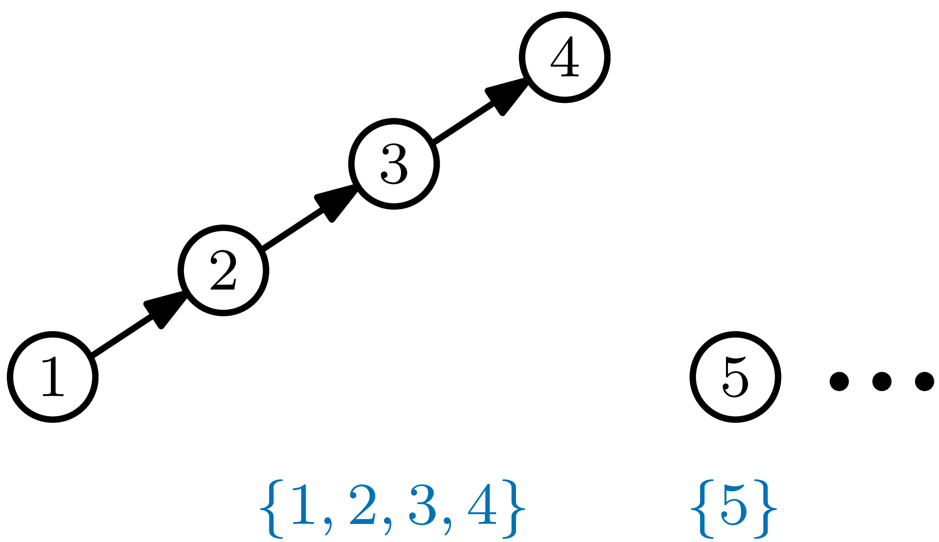

$\{1, 2, 3, 4\}$ $\{5\}$

Now perform UNION$(1, 4)$

# How high does the sycamore grow?

Unfortunately, every UNION operation could
increase the tallest tree height, $h$ by one...

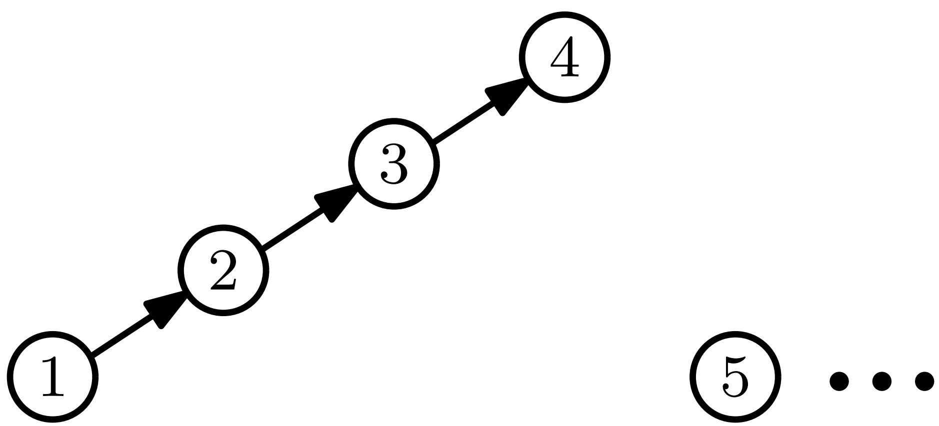
Consider the following sets:



# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:



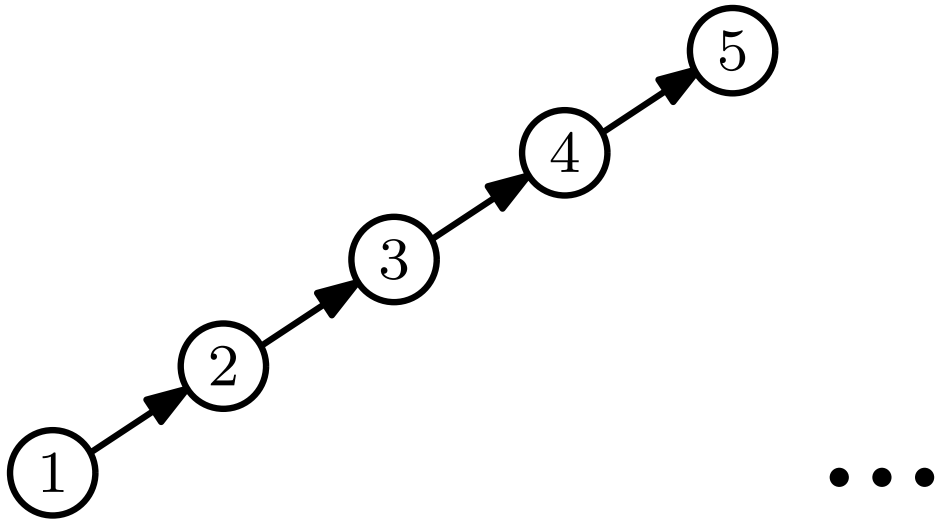

$\{1, 2, 3, 4\}$ $\{5\}$

Now perform UNION$(1, 5)$

# How high does the sycamore grow?

Unfortunately, every UNION operation could
increase the tallest tree height, $h$ by one...

Consider the following sets:



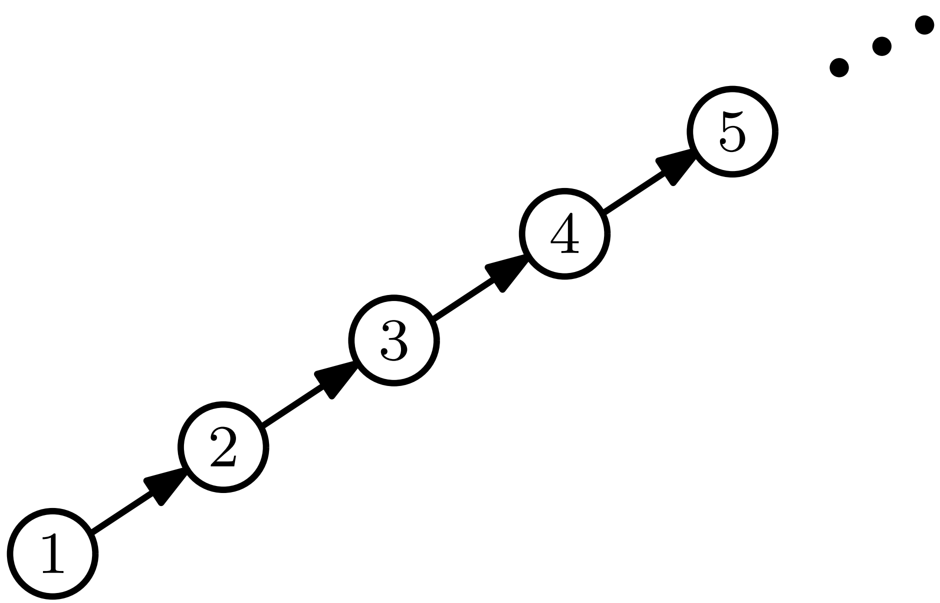
$\{1, 2, 3, 4, 5\}$

Now perform UNION$(1, 5)$

# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:



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

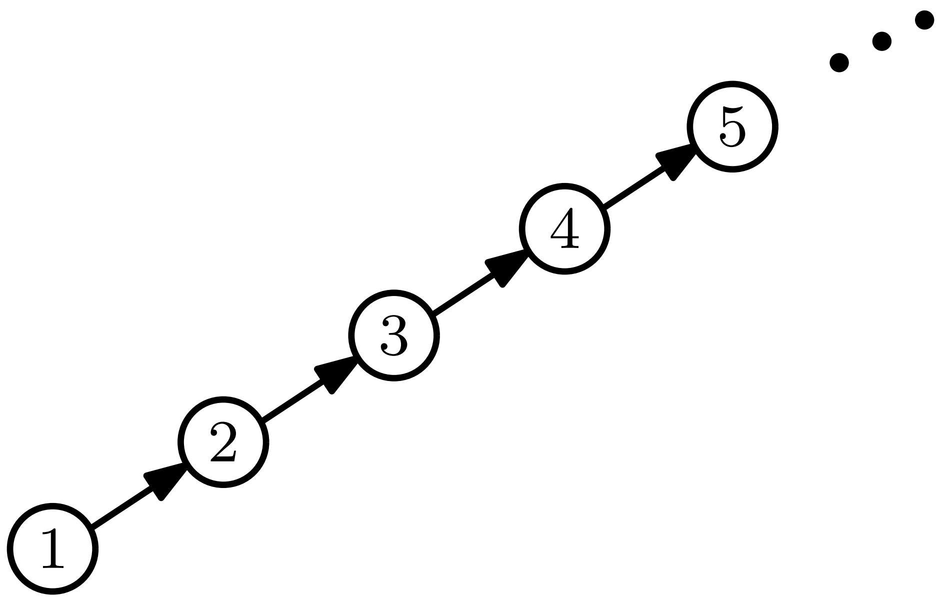
Now perform UNION$(1, 5)$...

So in the worst case the height of the tallest tree is $n$

# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:

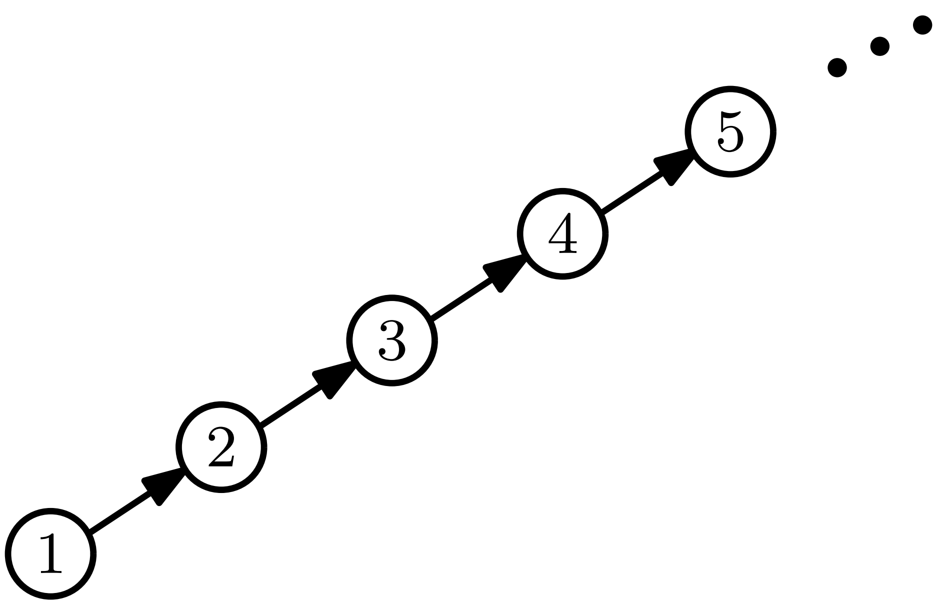



$\{1, 2, 3, 4, 5, \ldots\}$

# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:




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

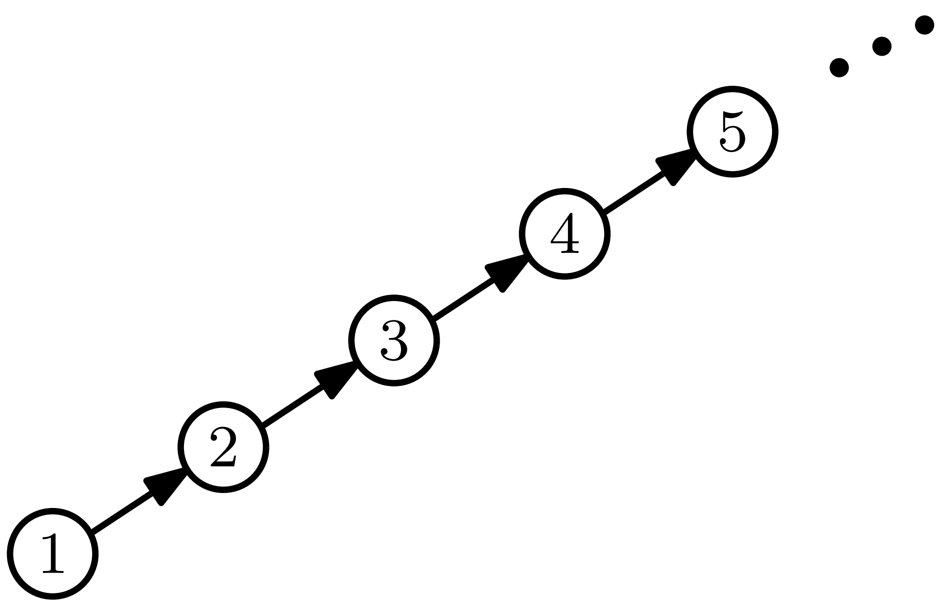
In the worst case the height of the tallest tree is $n$

# How high does the sycamore grow?

Unfortunately, every UNION operation could

increase the tallest tree height, $h$ by one...

Consider the following sets:




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

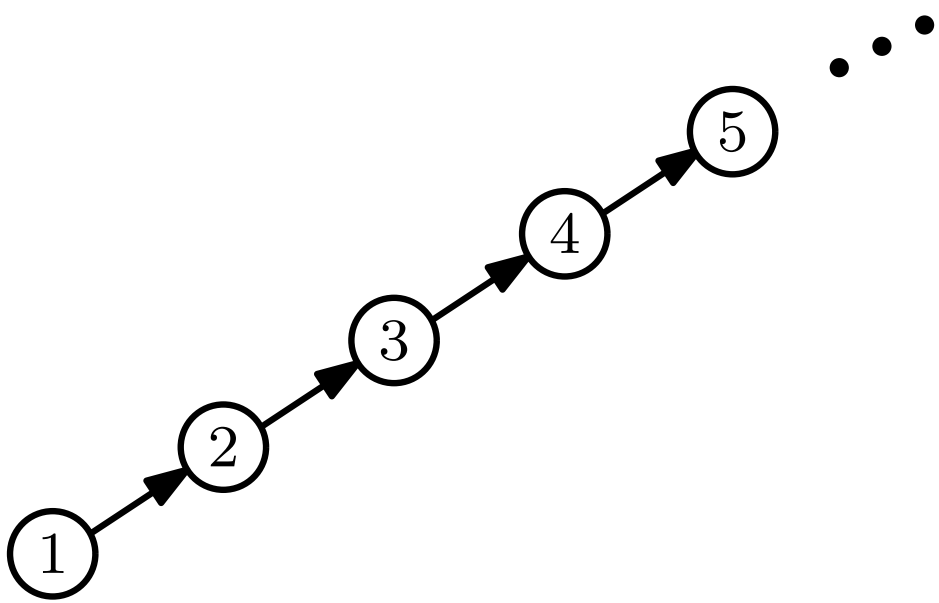
In the worst case the height of the tallest tree is $n$

so UNION and FIND run in $O(n)$ time

# How high does the sycamore grow?

Unfortunately, every UNION operation could increase the tallest tree height, $h$ by one...

Consider the following sets:




*Can we improve this?*

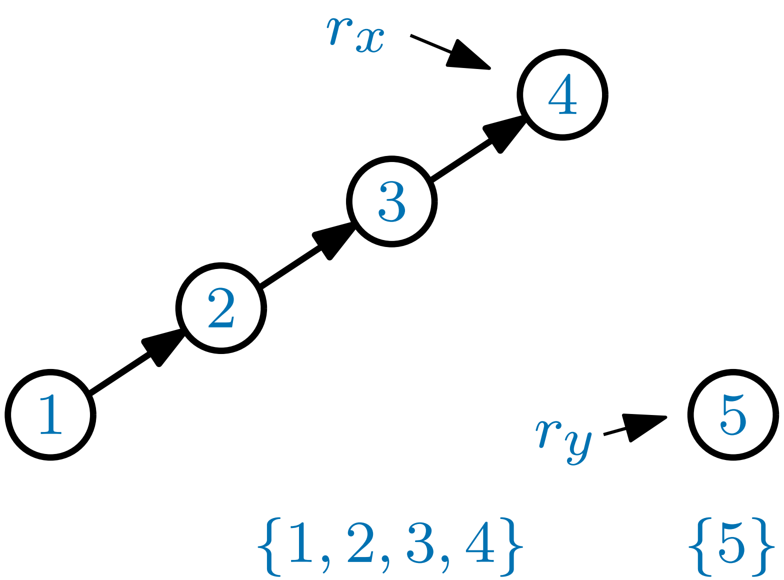
$$\{1, 2, 3, 4, 5, \ldots\}$$

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

so UNION and FIND run in $O(n)$ time

# What's bad about the UNION operation?

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



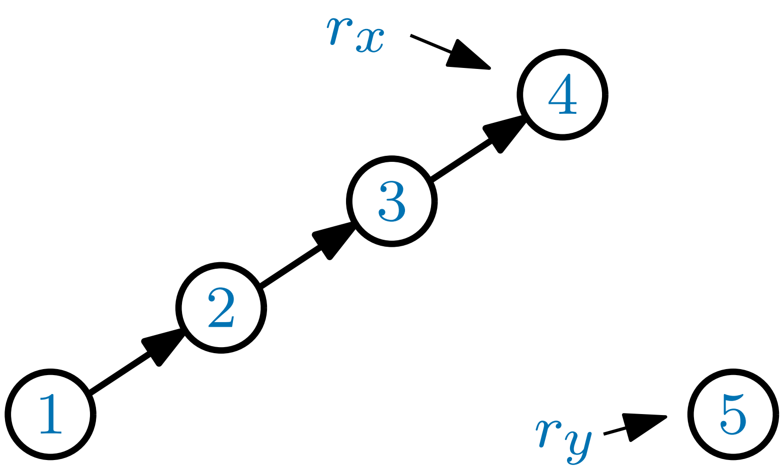

**Step 1:** Compute $r_x$ = FINDSET$(x)$ - the root of the tree containing $x$

**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's bad about the UNION operation?

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




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

When we performed UNION$(1, 5)$,

we made a $r_x$ the child of $r_y$

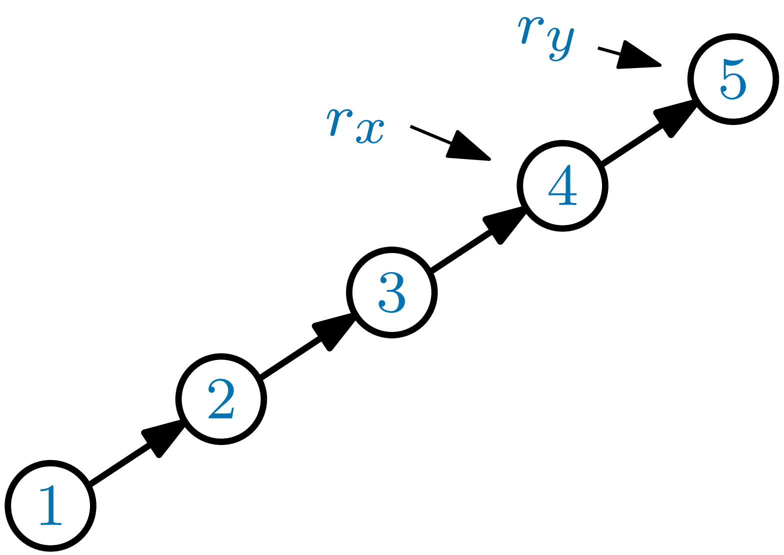
**Step 1:** Compute $r_x$ = FINDSET$(x)$ - the root of the tree containing $x$

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

**Step 3:** 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




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

When we performed UNION$(1, 5)$,

we made a $r_x$ the child of $r_y$

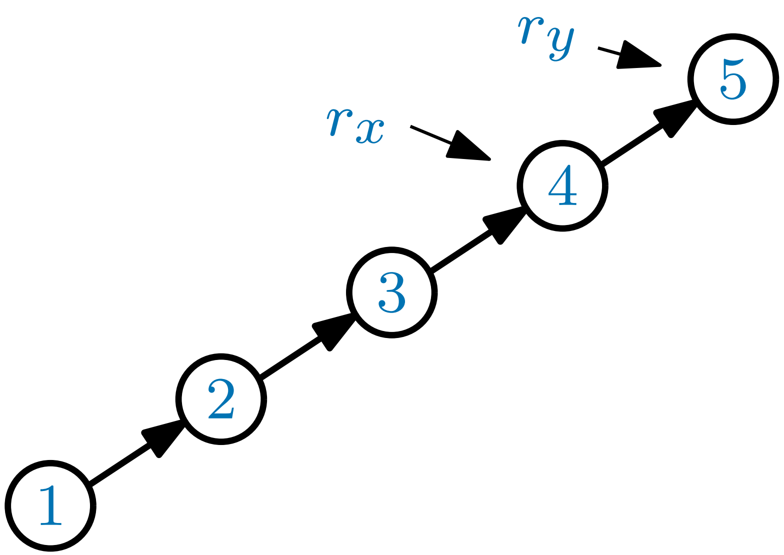
**Step 1:** Compute $r_x$ = FINDSET$(x)$ - the root of the tree containing $x$

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

**Step 3:** 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




When we performed UNION$(1, 5)$,

we made a $r_x$ the child of $r_y$

this increases the height by one

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

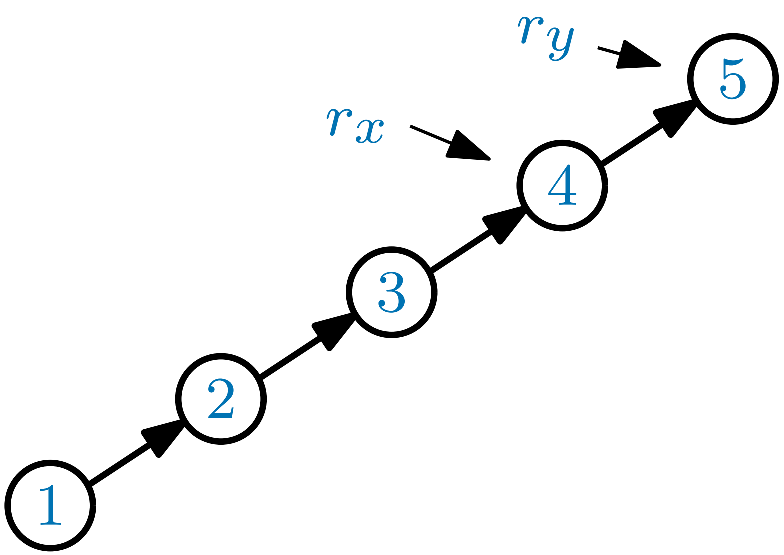
**Step 1:** Compute $r_x =$ FINDSET$(x)$ - the root of the tree containing $x$

**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's bad about the UNION operation?

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




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

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 $r_x$...

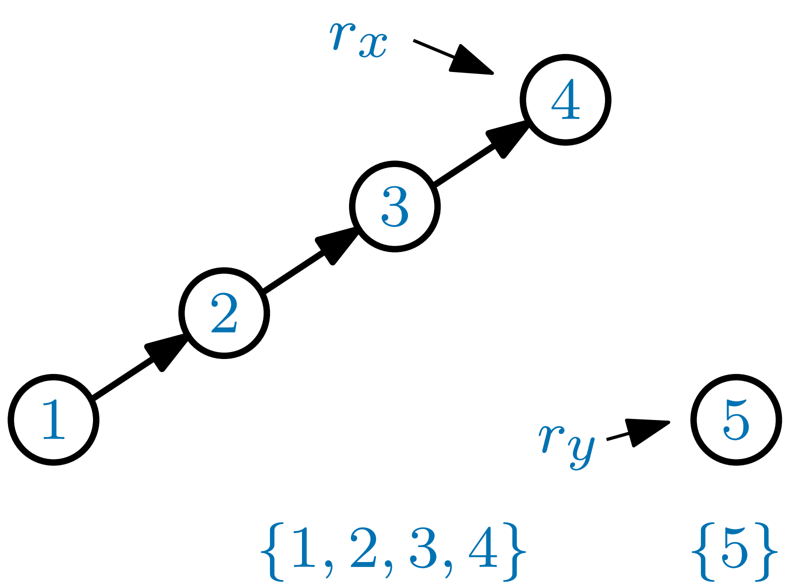
**Step 1:** Compute $r_x$ = FINDSET$(x)$ - the root of the tree containing $x$

**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's bad about the 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 $r_x$...

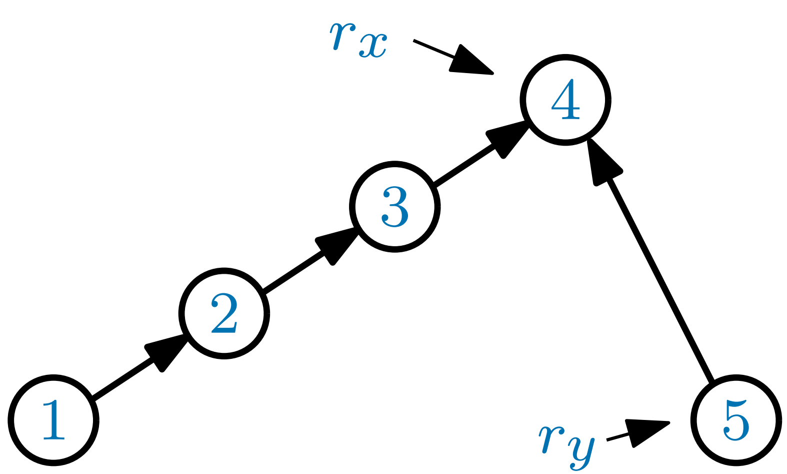
**Step 1:** Compute $r_x = \textsc{FindSet}(x)$ - the root of the tree containing $x$

**Step 2:** Compute $r_y = \textsc{FindSet}(y)$ - 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




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

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 $r_x$...

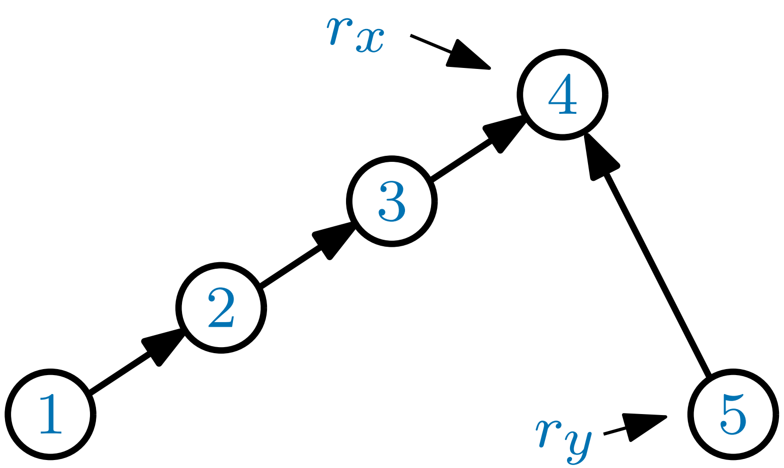
**Step 1:** Compute $r_x$ = FINDSET$(x)$ - the root of the tree containing $x$

**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's bad about the UNION operation?

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




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

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

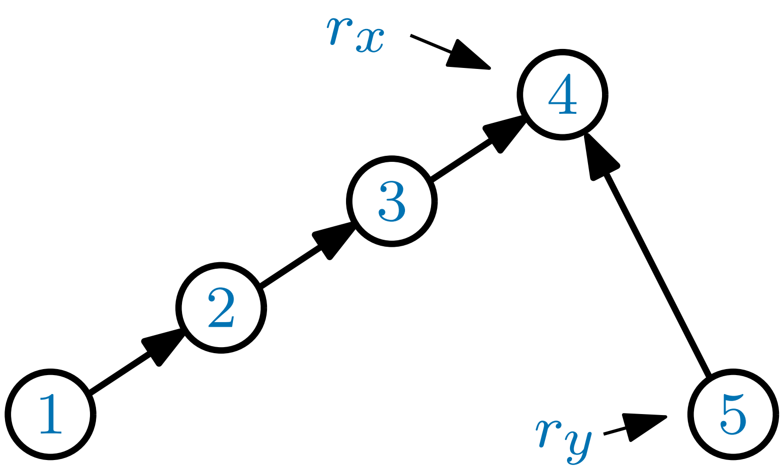
**Step 1:** Compute $r_x =$ FINDSET$(x)$ - the root of the tree containing $x$

**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's bad about the UNION operation?

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




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

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

How can we generalise this?

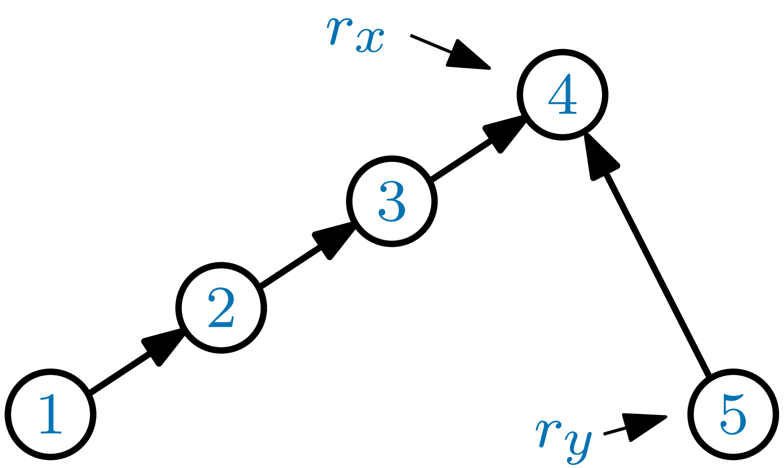
**Step 1:** Compute $r_x$ = FINDSET$(x)$ - the root of the tree containing $x$

**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's bad about the UNION operation?

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




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

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

How can we generalise this?

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

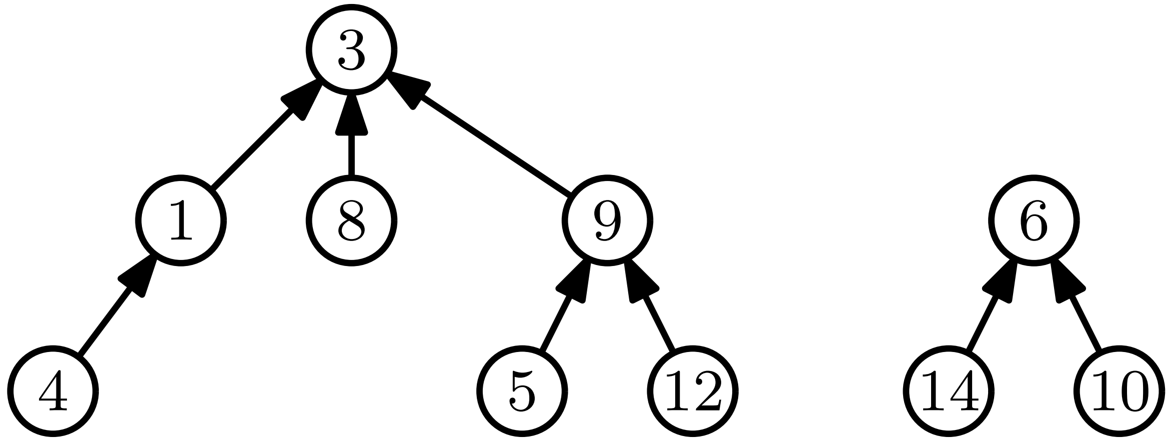
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

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

**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



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 $x$

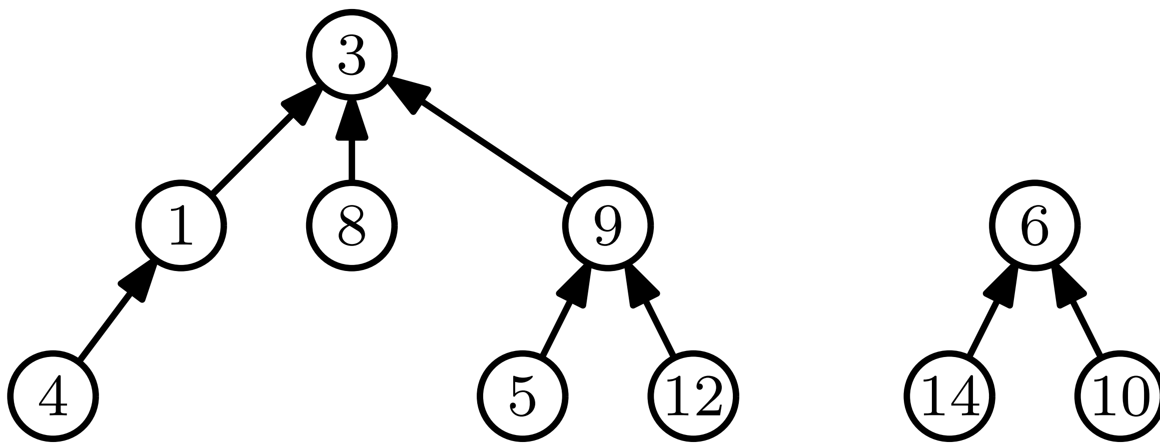
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

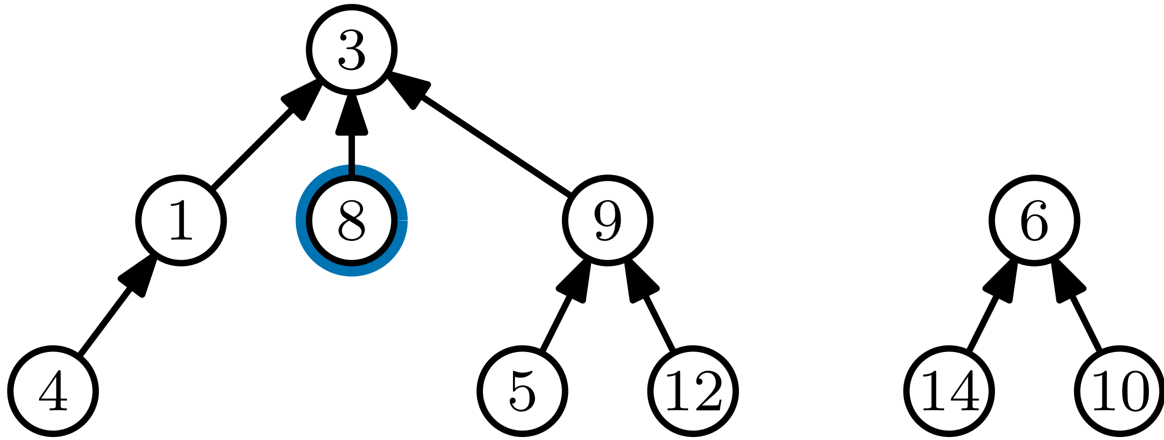
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

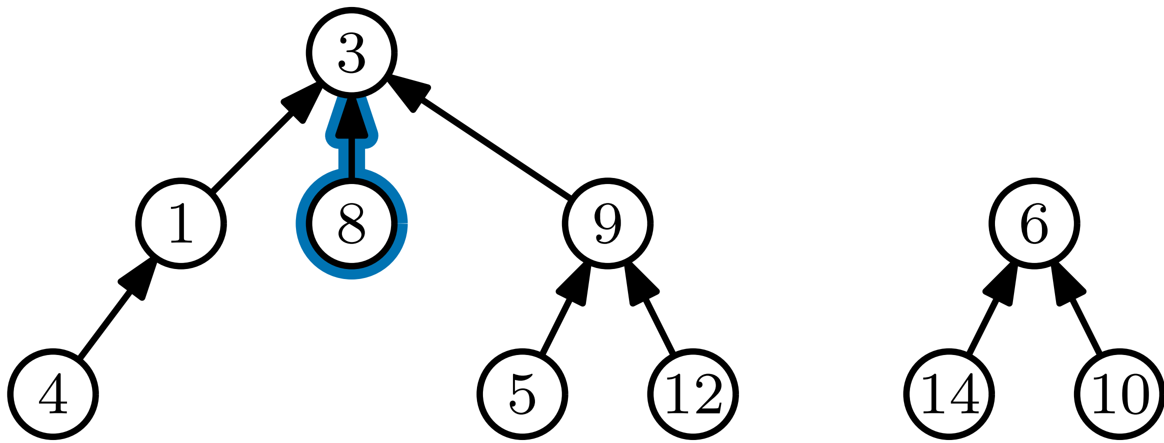
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

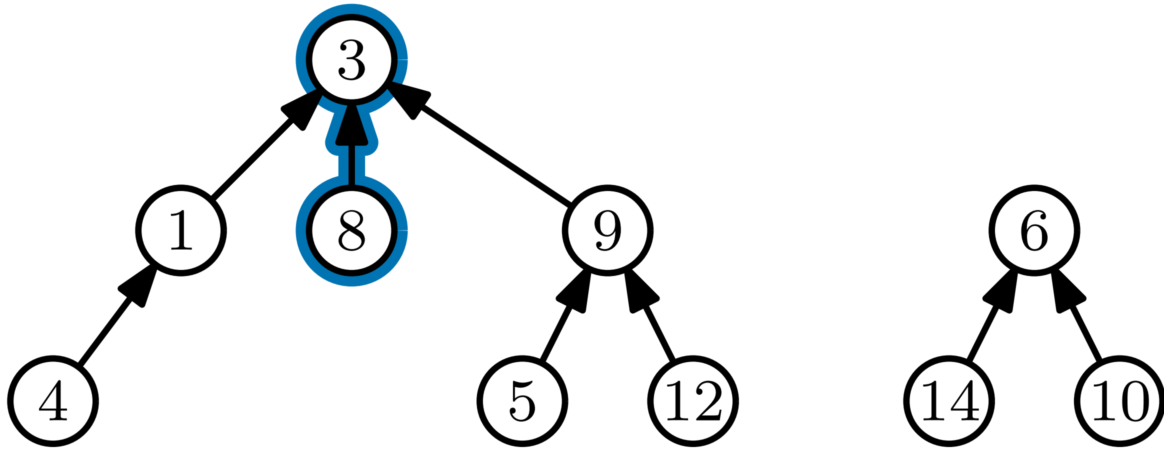
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

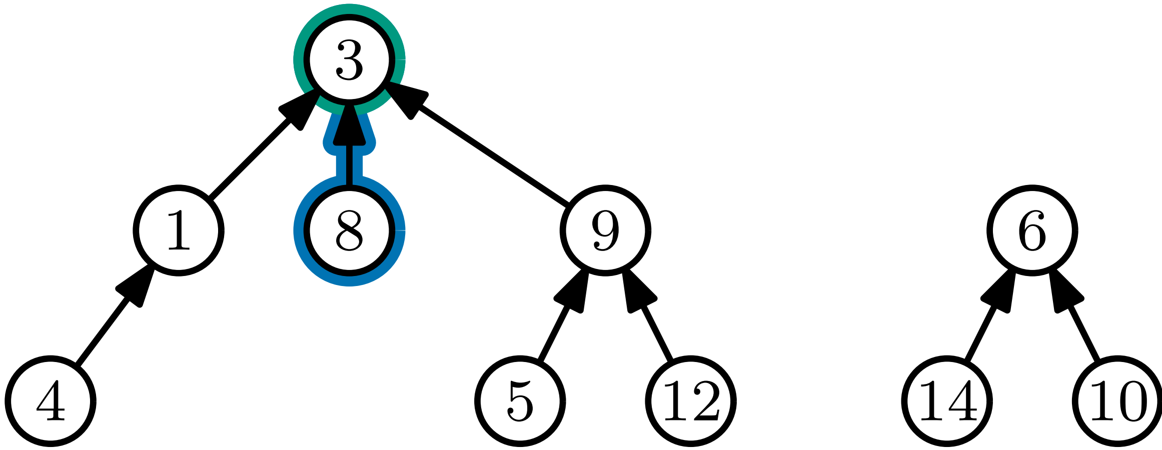
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

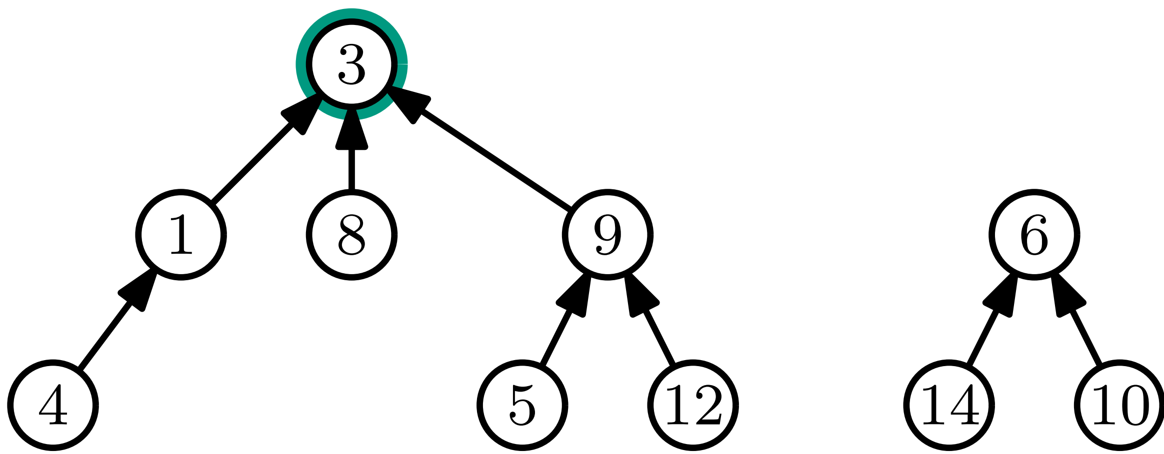
**Step 2:** Compute $r_y =$ FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

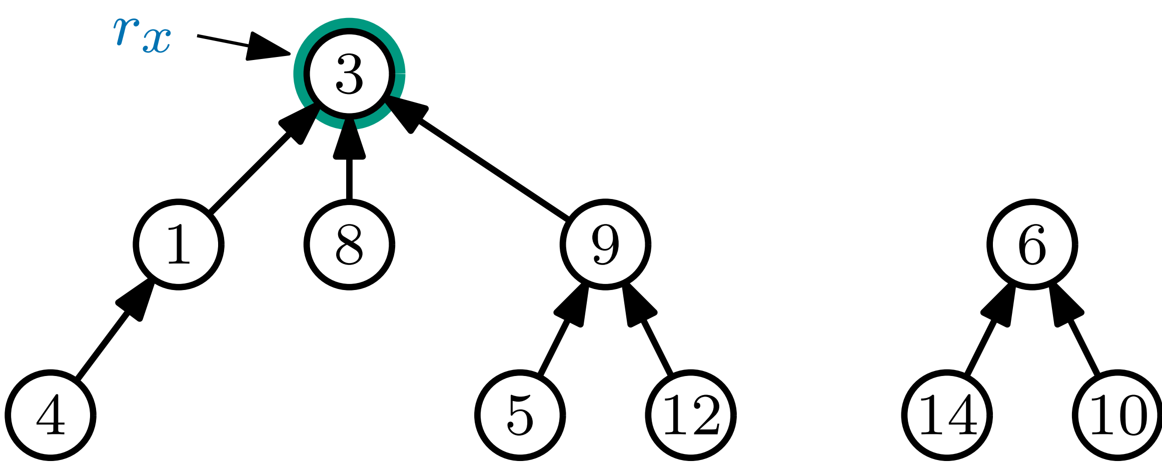
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

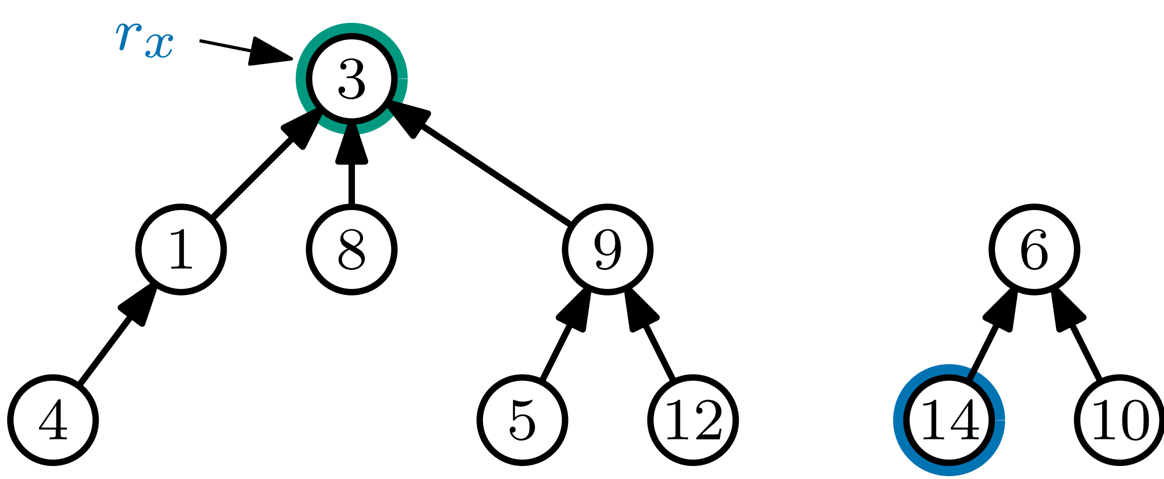
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

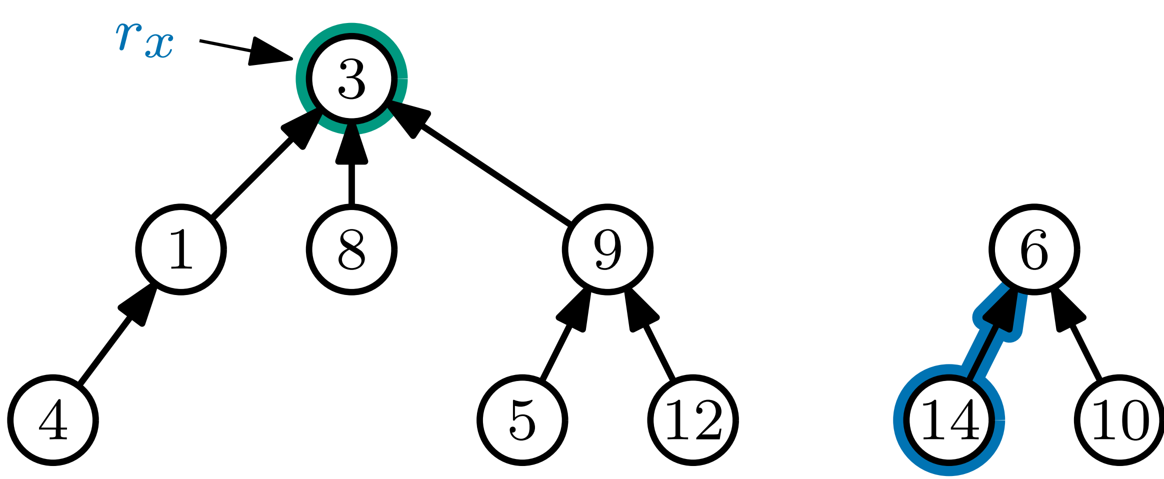
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

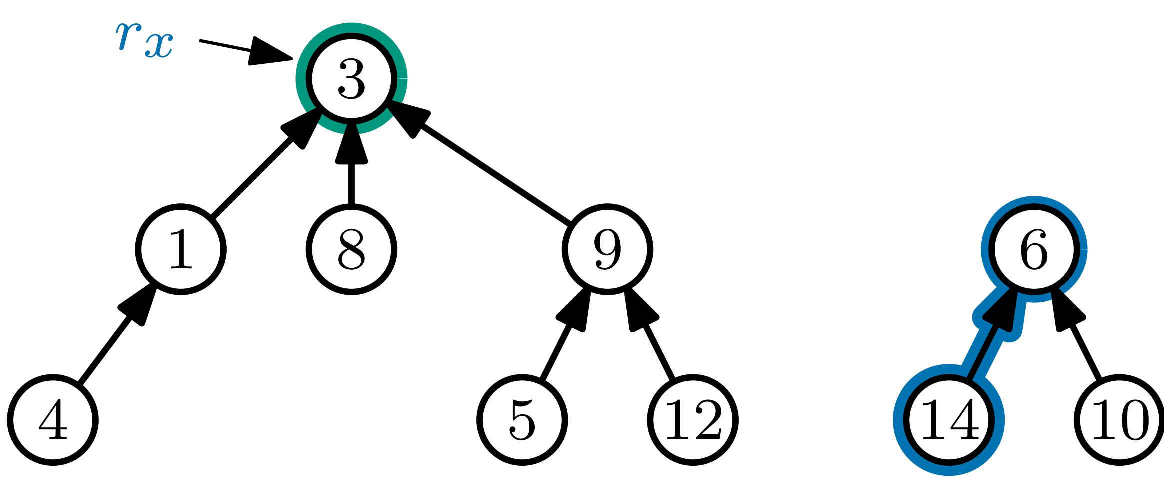
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

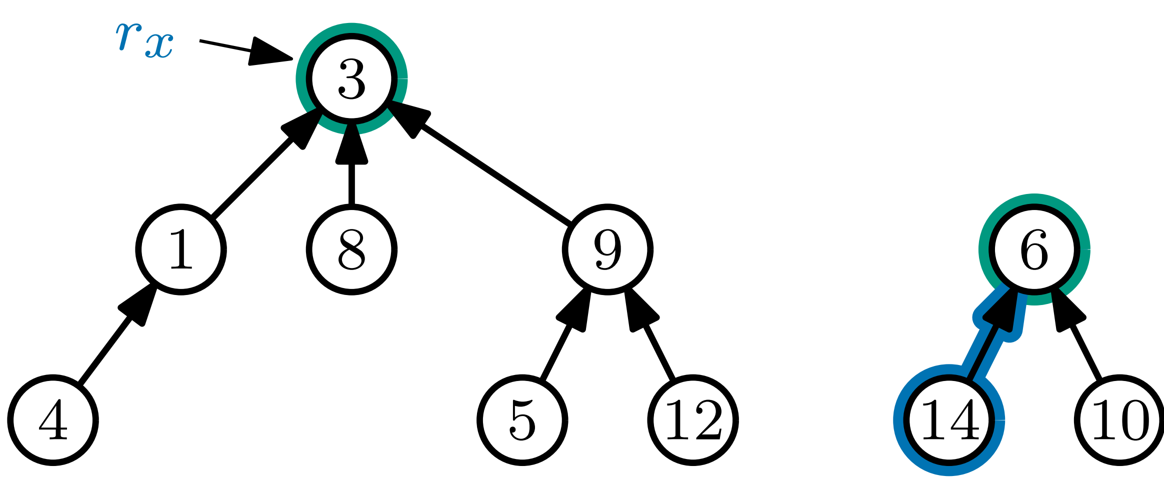
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

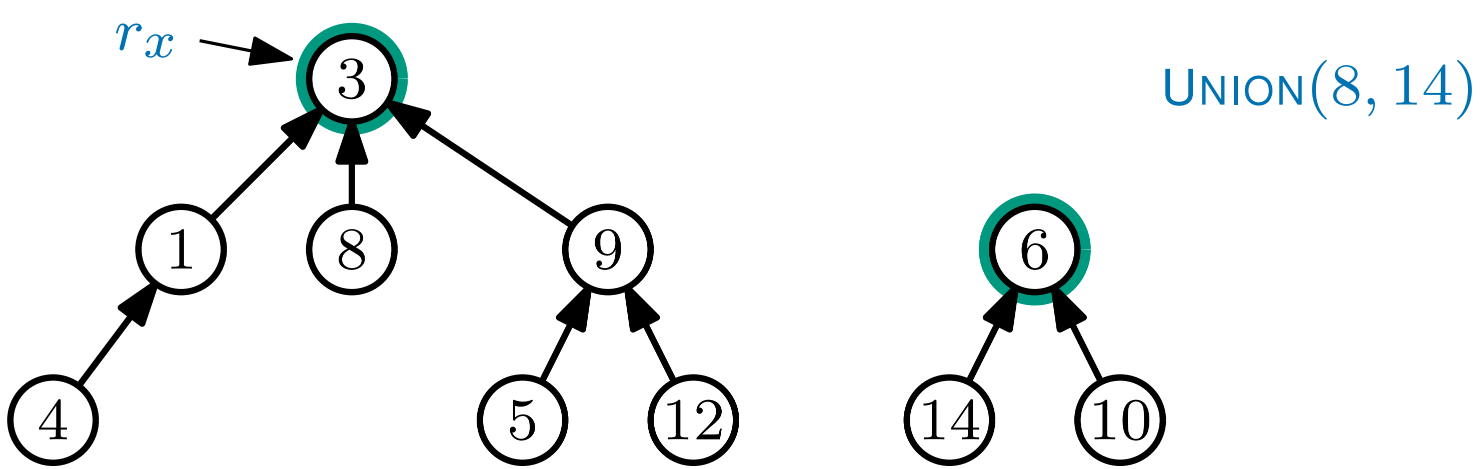
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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 $x$

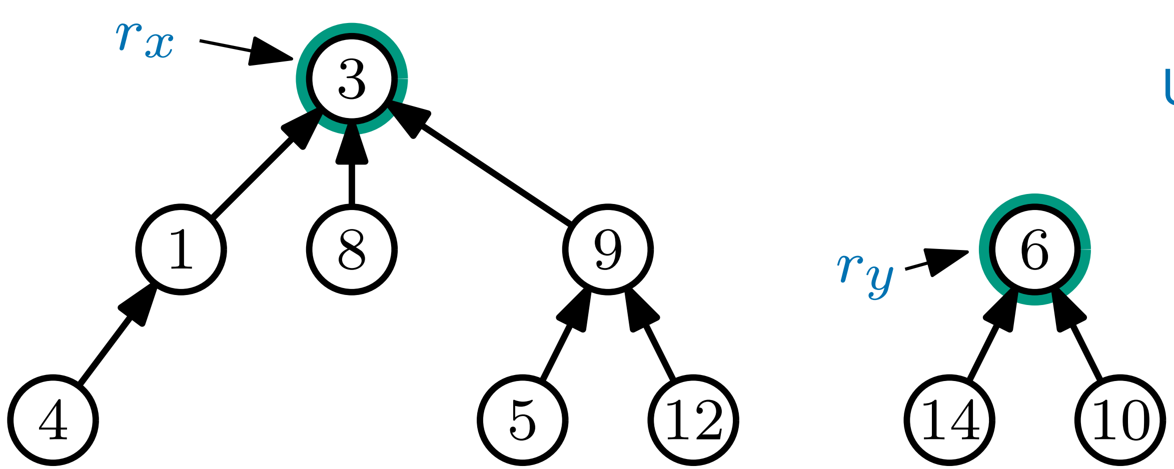
**Step 2:** Compute $r_y$ = FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




UNION$(8, 14)$

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 $x$

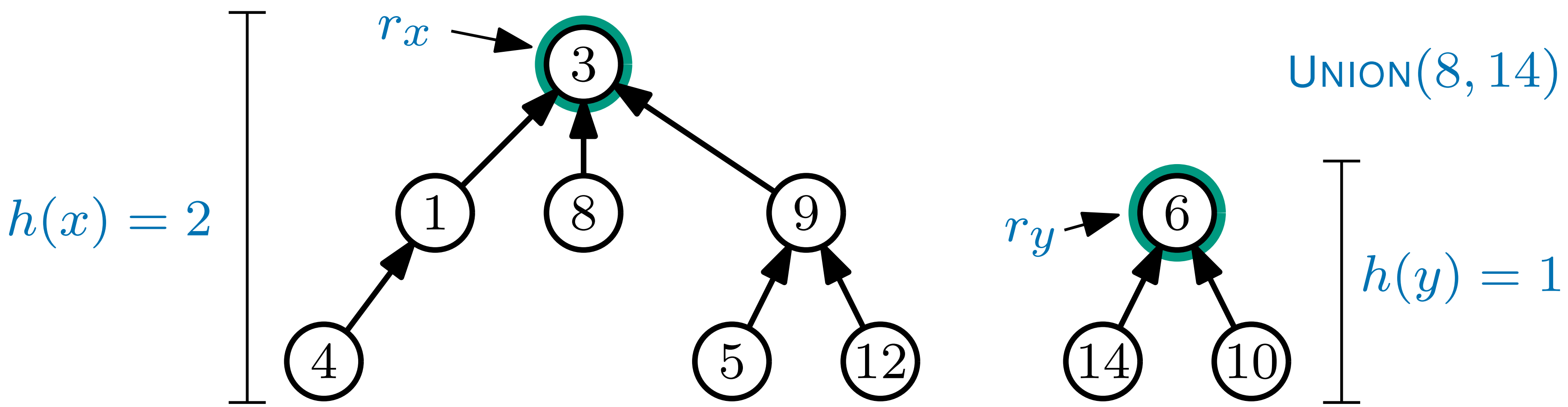
**Step 2:** Compute $r_y =$ FINDSET$(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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 $x$

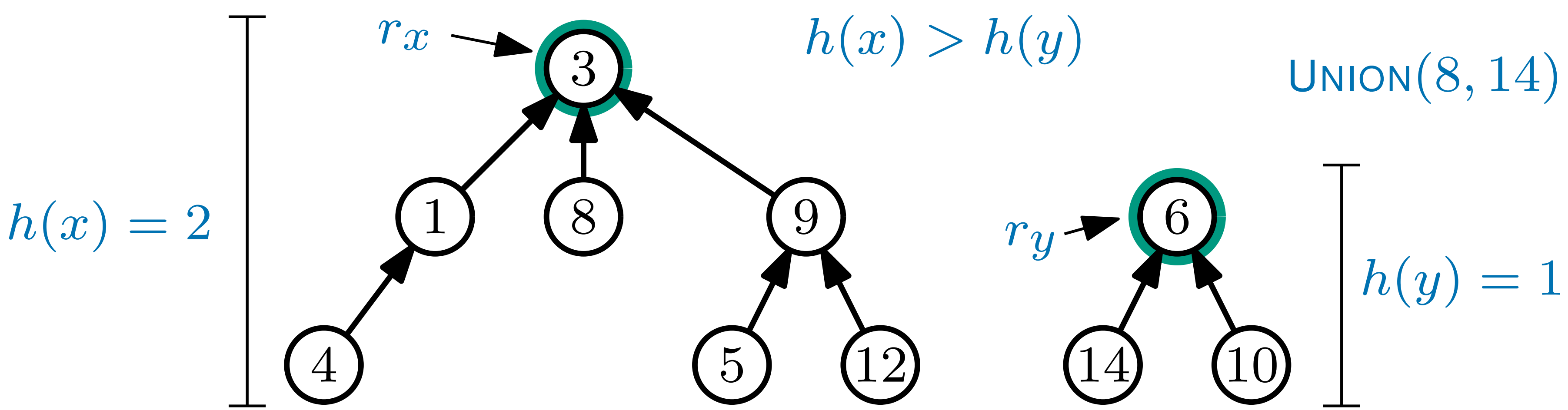
**Step 2:** Compute $r_y =$ FINDSET$(y)$ - the root of the tree containing $y$

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

Else make $r_y$ a child of $r_x$

# An improved $\textsc{Union}$ operation

$\textsc{Union}(x, y)$ - 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 $r_x = \textsc{FindSet}(x)$ - the root of the tree containing $x$

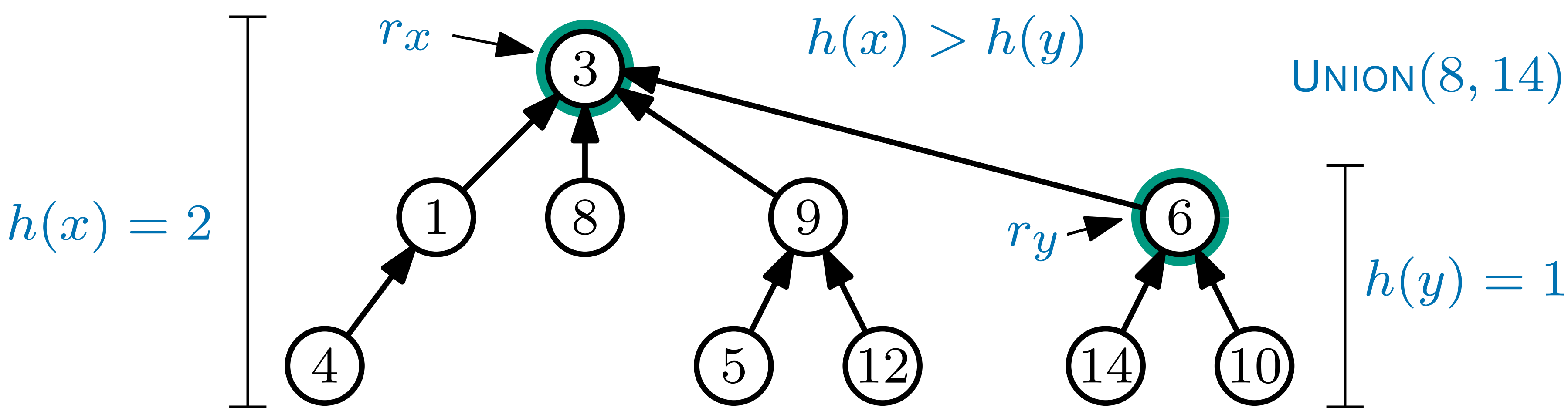
**Step 2:** Compute $r_y = \textsc{FindSet}(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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




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

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

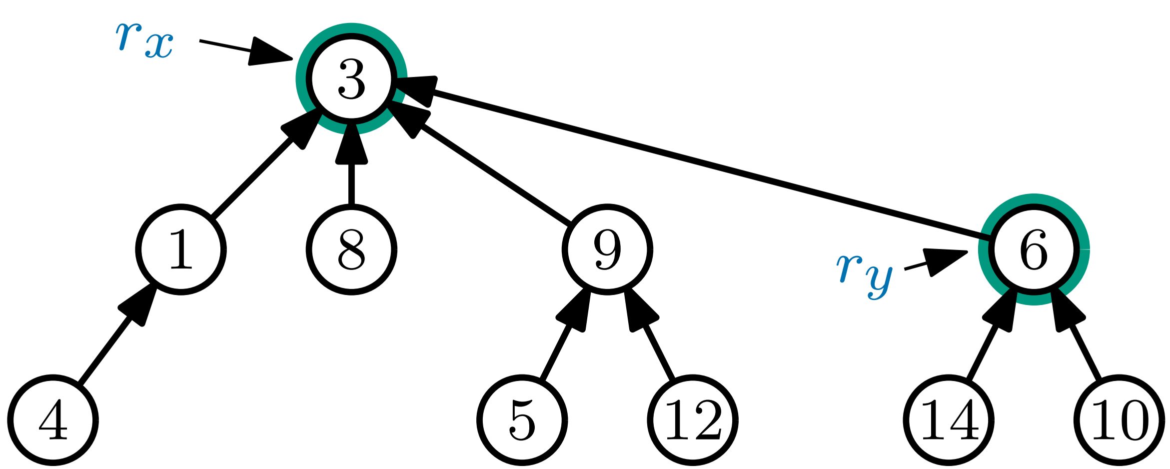
**Step 2:** Compute $r_y = \textsc{FindSet}(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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 $x$

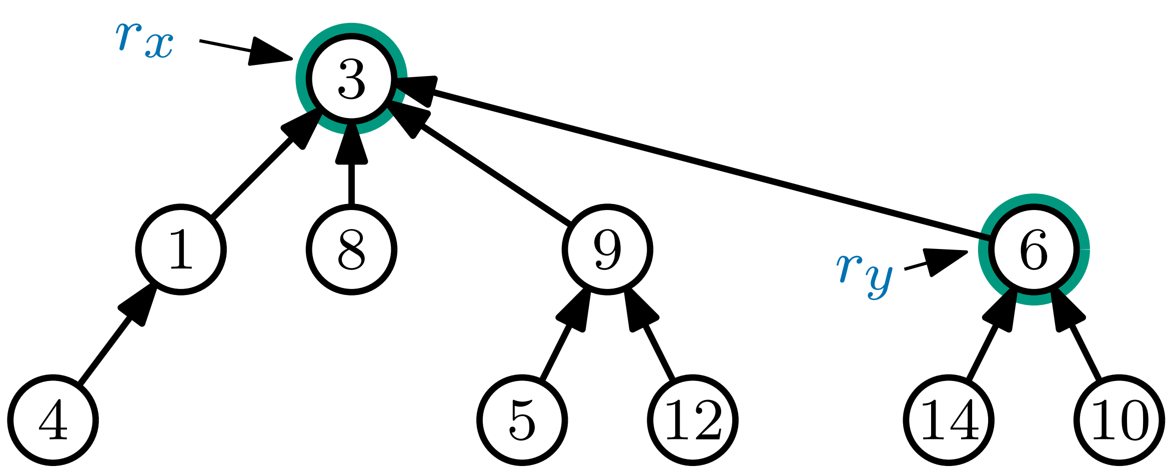
**Step 2:** Compute $r_y = \text{FINDSET}(y)$ - the root of the tree containing $y$

**Step 3:** If $h(x) \leqslant h(y)$ 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 $y$

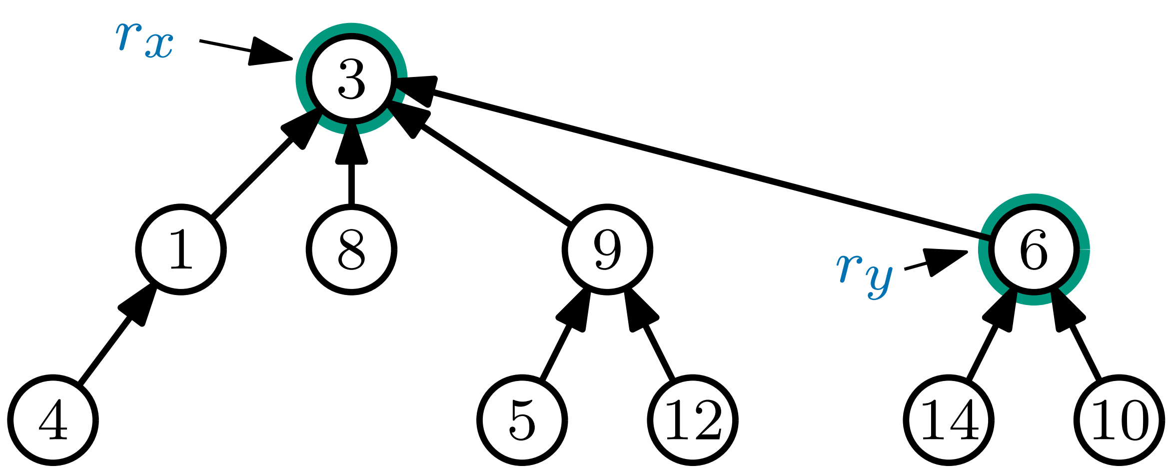
**Step 3:** If $h(x) \leqslant h(y)$ make $r_x$ a child of $r_y$

Else make $r_y$ a child of $r_x$

This still takes $O(h)$ time

# An improved UNION operation

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 $h(x) \leqslant h(y)$ 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 $h$ (created by the data structure)

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 $h$ (created by the data structure)

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 $h$ (created by the data structure)
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 $\rightarrow$ $5$
(so contains $2^i = 1$ 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 $h$ (created by the data structure)

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

5

(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 $h$ (created by the data structure)

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) 5

**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 $h$ (created by the data structure)
contains at least $2^h$ nodes

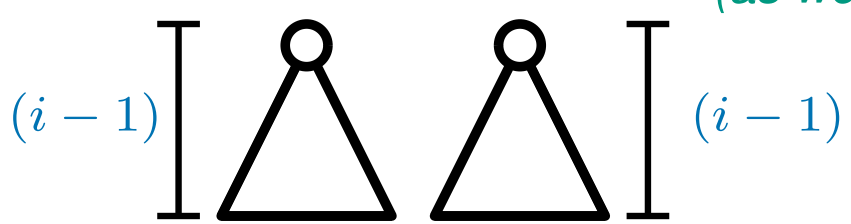
**Proof** by induction on tree height $i$,

**Base Case** ($i = 0$) Any tree of height $0$ represents a single element set
(so contains $2^i = 1$ node) 5

**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 $h$ (created by the data structure)
contains at least $2^h$ nodes

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

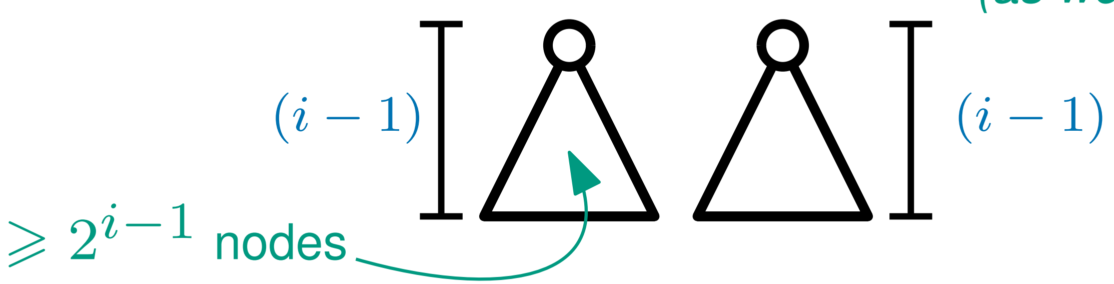
**Base Case** ($i = 0$) Any tree of height $0$ represents a single element set
(so contains $2^i = 1$ node)

5

**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 $h$ (created by the data structure)
contains at least $2^h$ nodes

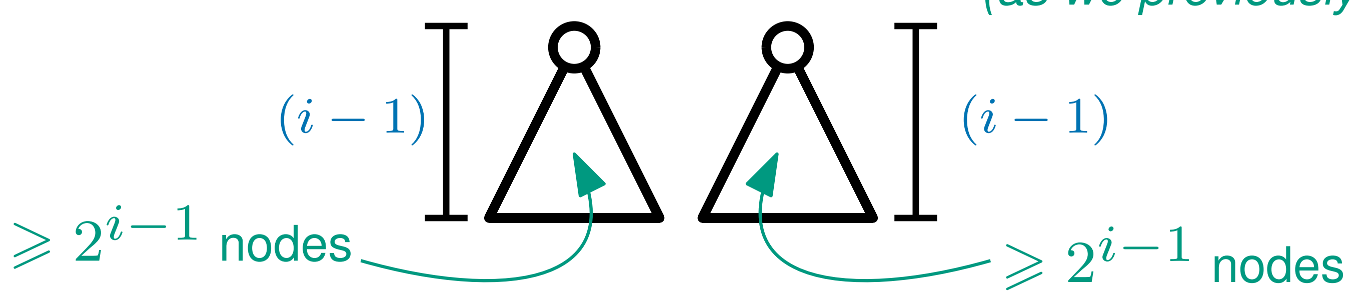
**Proof** by induction on tree height $i$,

**Base Case** ($i = 0$) Any tree of height $0$ represents a single element set $5$
(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 $h$ (created by the data structure)

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

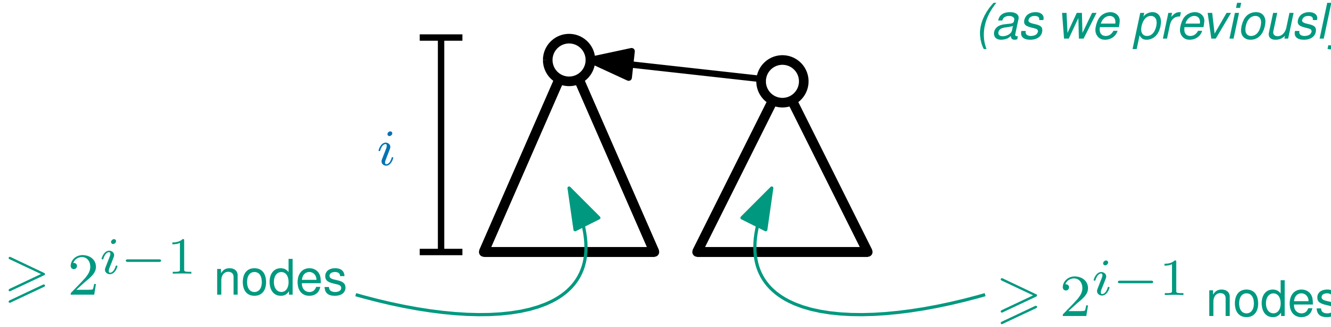
(so contains $2^i = 1$ node) 5

**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 $h$ (created by the data structure)
contains at least $2^h$ nodes

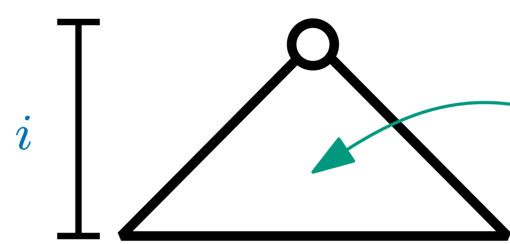
**Proof** by induction on tree height $i$,

**Base Case** ($i = 0$) Any tree of height $0$ represents a single element set
(so contains $2^i = 1$ node) 5

**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)*




$\geqslant 2 \cdot 2^{i-1} = 2^i$ nodes

## Now big is $h$ now?

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

We begin by proving that
any tree of height $h$ (created by the data structure)
contains at least $2^h$ nodes

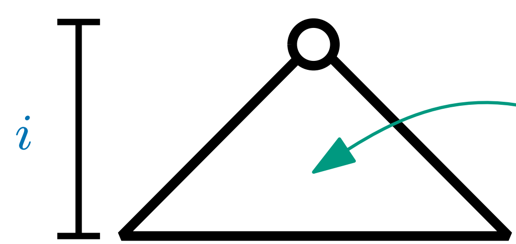
**Proof** by induction on tree height $i$,

**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)*




$\geqslant 2 \cdot 2^{i-1} = 2^i$ nodes

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 $O(\log n)$

**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

## Now big is $h$ now?

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

**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

Now assume (for a contradiction) that there is a tree with height $h \geqslant \log_2 n + 1$

# Now big is $h$ now?

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

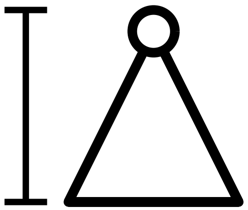
**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

Now assume (for a contradiction) that there is a tree with height $h \geqslant \log_2 n + 1$

$\geqslant \log_2 n + 1$



# Now big is $h$ now?

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

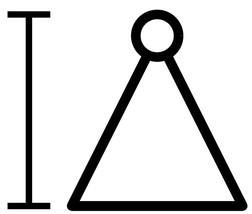
**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

Now assume (for a contradiction) that there is a tree with height $h \geqslant \log_2 n + 1$

$\geqslant \log_2 n + 1$



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 $O(\log n)$

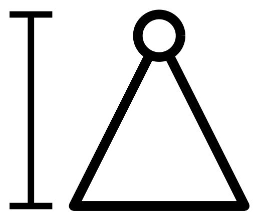
**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

Now assume (for a contradiction) that there is a tree with height $h \geqslant \log_2 n + 1$

$\geqslant \log_2 n + 1$



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

*and each node represents a distinct element*

# Now big is $h$ now?

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

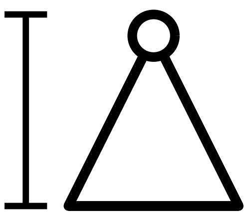
**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

Now assume (for a contradiction) that there is a tree with height $h \geqslant \log_2 n + 1$

$\geqslant \log_2 n + 1$



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

*and each node represents 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 $O(\log n)$

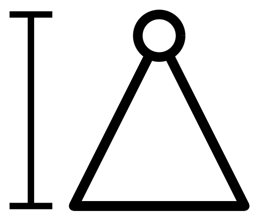
**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

Now assume (for a contradiction) that there is a tree with height $h \geqslant \log_2 n + 1$

$\geqslant \log_2 n + 1$



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

*and each node represents a distinct element*

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

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

So, $h \leqslant \log_2 n$,

# Now big is $h$ now?

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

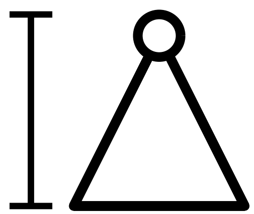
**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

Now assume (for a contradiction) that there is a tree with height $h \geqslant \log_2 n + 1$

$\geqslant \log_2 n + 1$



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

*and each node represents a distinct element*

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

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

So, $h \leqslant \log_2 n$,

Recall that the operations UNION and FINDSET run in $O(h)$ time

# Now big is $h$ now?

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

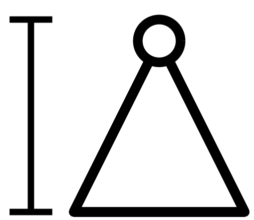
**We have proven that**

any tree of height $h$ (created by the data structure)

contains at least $2^h$ nodes

Now assume (for a contradiction) that there is a tree with height $h \geqslant \log_2 n + 1$

$\geqslant \log_2 n + 1$



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

*and each node represents a distinct element*

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

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

So, $h \leqslant \log_2 n$,

Recall that the operations UNION and FINDSET run in $O(h)$ time

As, $h \leqslant \log_2 n$ they both only take $O(\log n)$ time.

# Disjoint Set Summary

We have seen a data structure which

stores a collection of disjoint sets

*The elements of the sets are numbers from* $\{1, 2, \ldots, n\}$

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 $x$

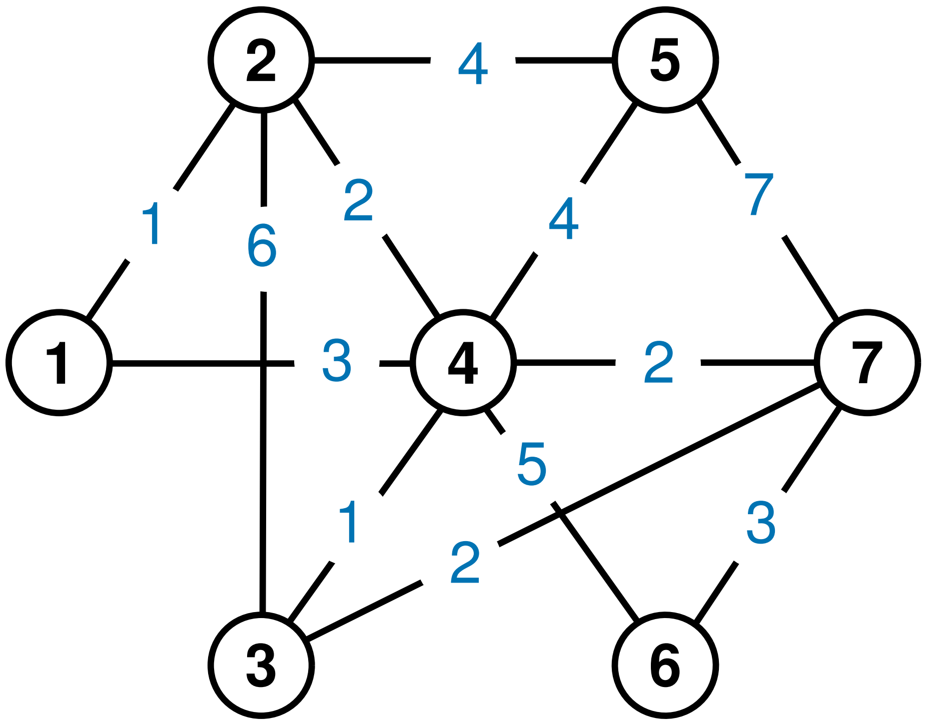
*the identity of a set is any unique identifier of the set.*

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

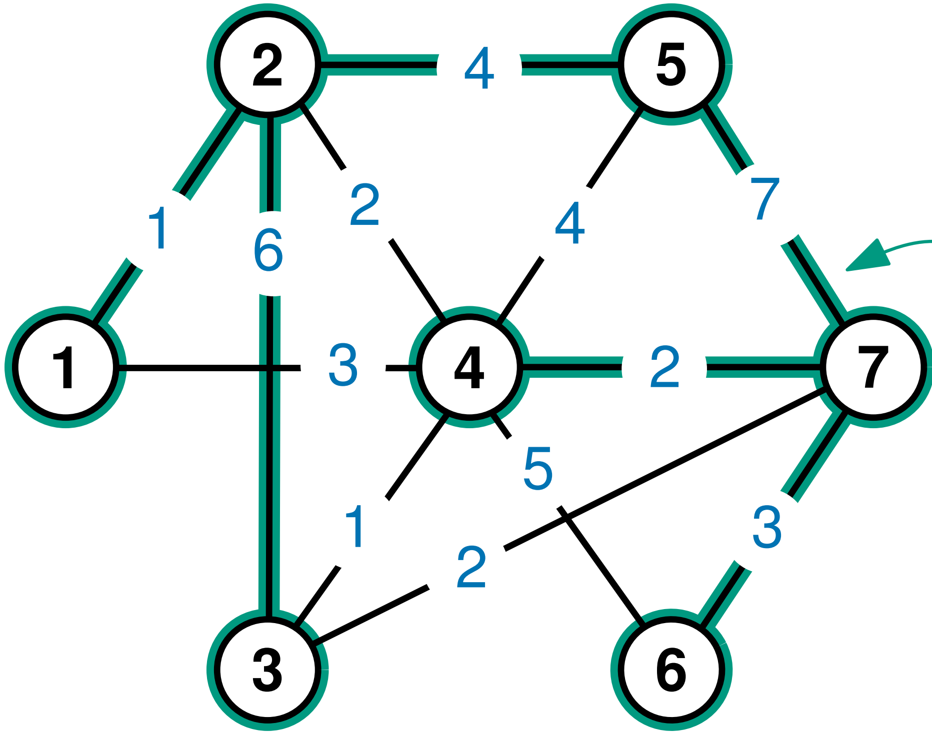



Every vertex $v \in V$ is in $T$

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



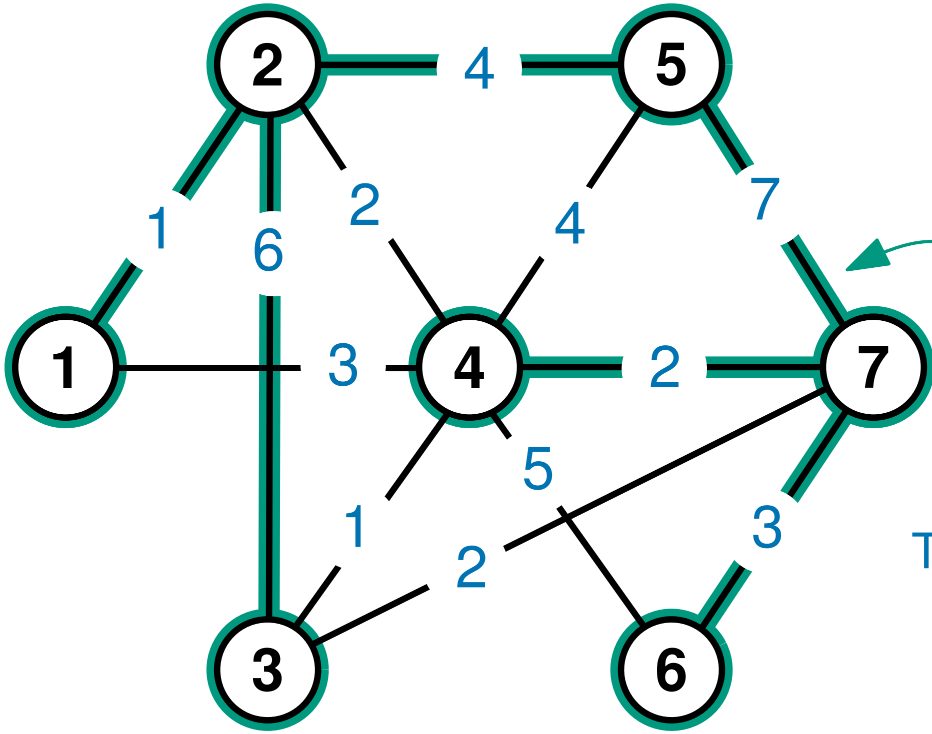

Every vertex $v \in V$ is in $T$

and $T$ is a tree *(it contains no cycles)*

a spanning tree

# 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)*

a spanning tree

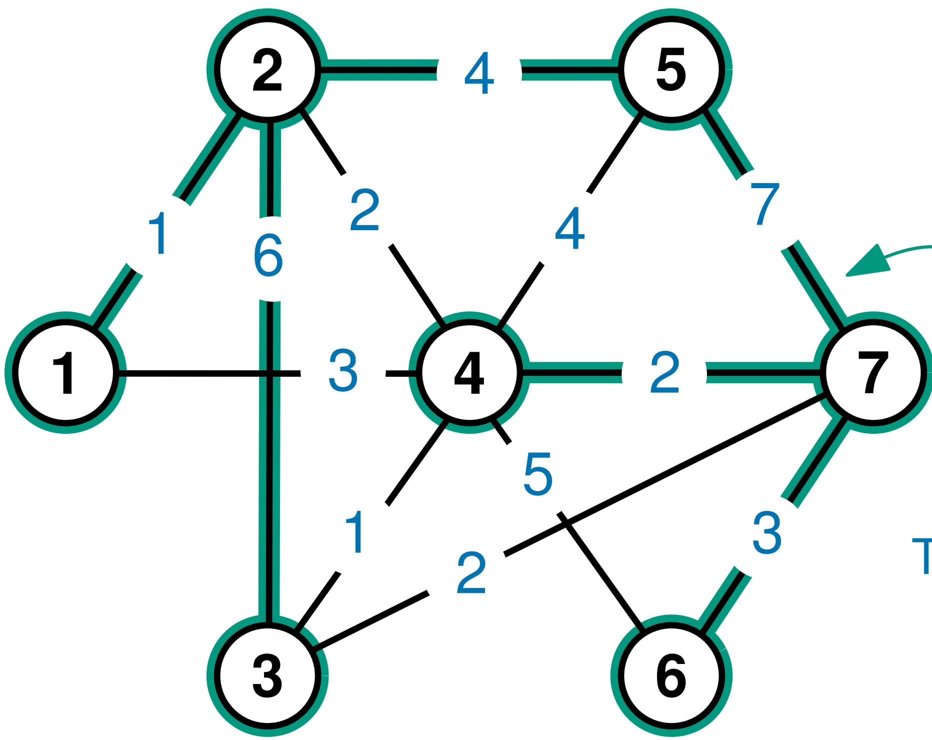
The weight of a spanning tree
is the sum of the weights of its edges

# 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)*




a spanning tree

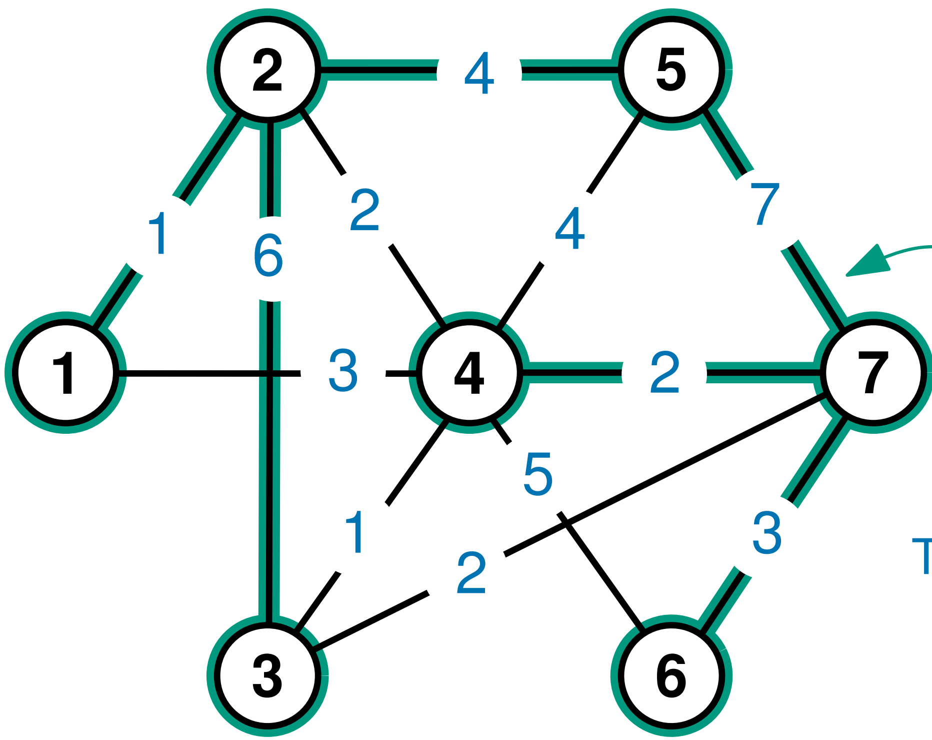
with weight $23$

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)*

a spanning tree

with weight $23$

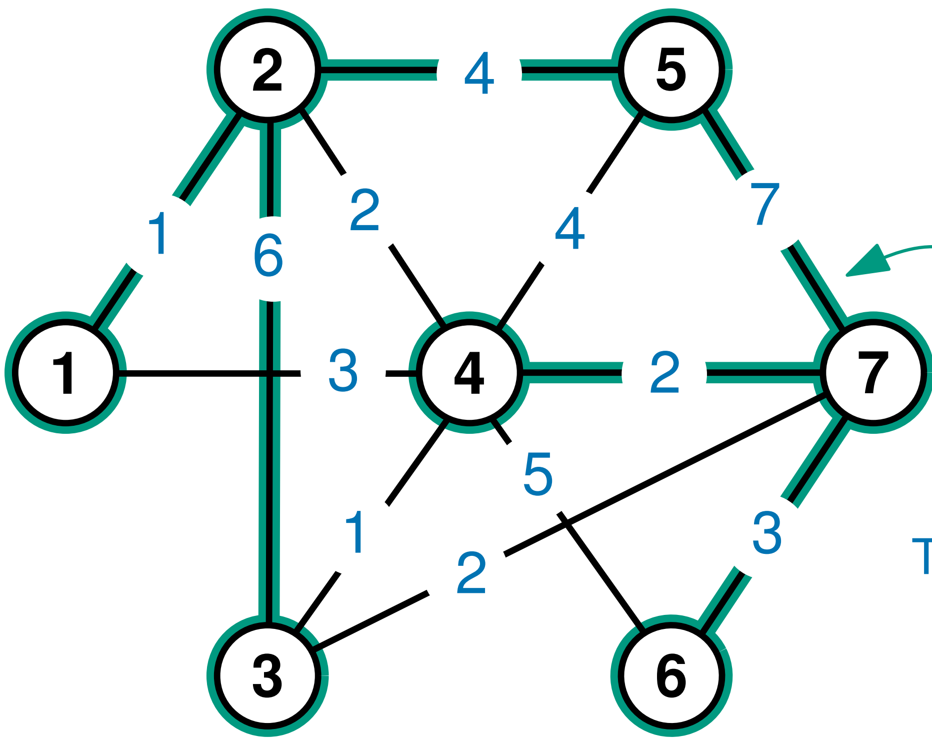
The weight of a spanning tree

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




Every vertex $v \in V$ is in $T$

and $T$ is a tree *(it contains no cycles)*

a spanning tree

with weight $23$

*(not a minimum spanning tree)*

The weight of a spanning tree

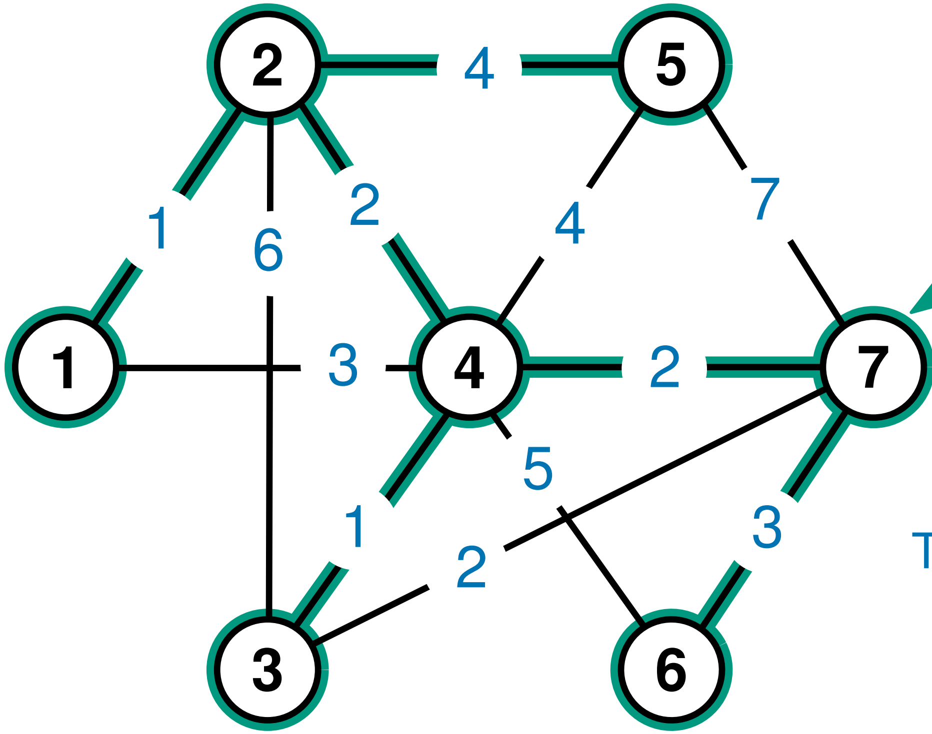
is the sum of the weights of its edges

$T$ is a minimum spanning tree if no other spanning tree has a lower weight

University of BRISTOL

# 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)*

a minimum spanning tree with weight $13$

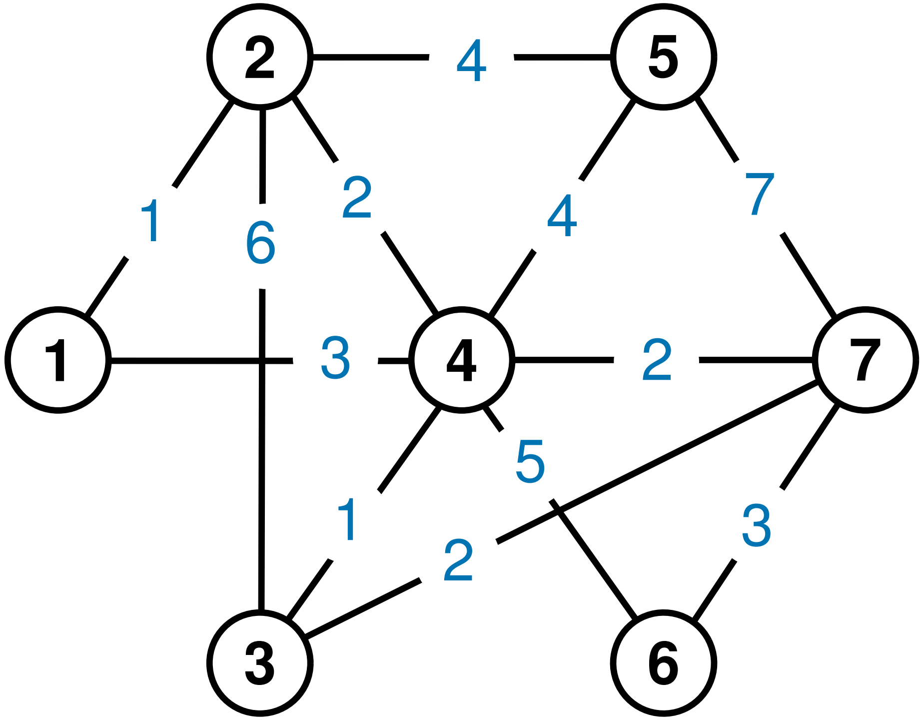
The weight of a spanning tree is the sum of the weights of its edges

$T$ is a minimum spanning tree if no other spanning tree has a lower weight

# Kruskal's algorithm

Kruskal's algorithm finds a minimum spanning tree in an connected, undirected graph...

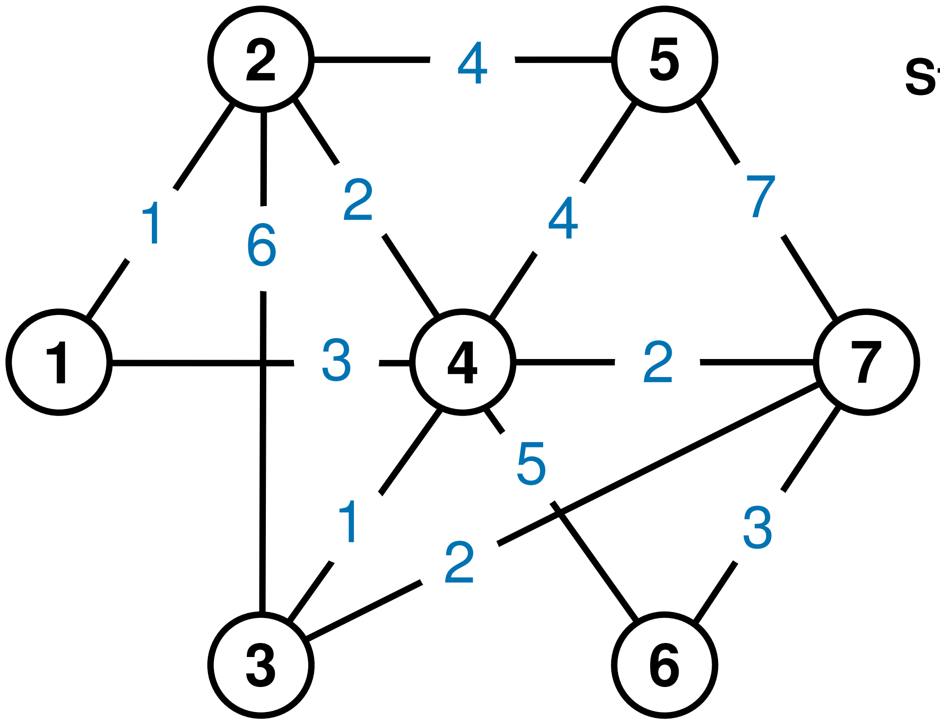
using a disjoint set data structure where the elements are from $\{1, 2, 3, \ldots, |V|\}$



# 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 $\{1, 2, 3, \ldots, |V|\}$

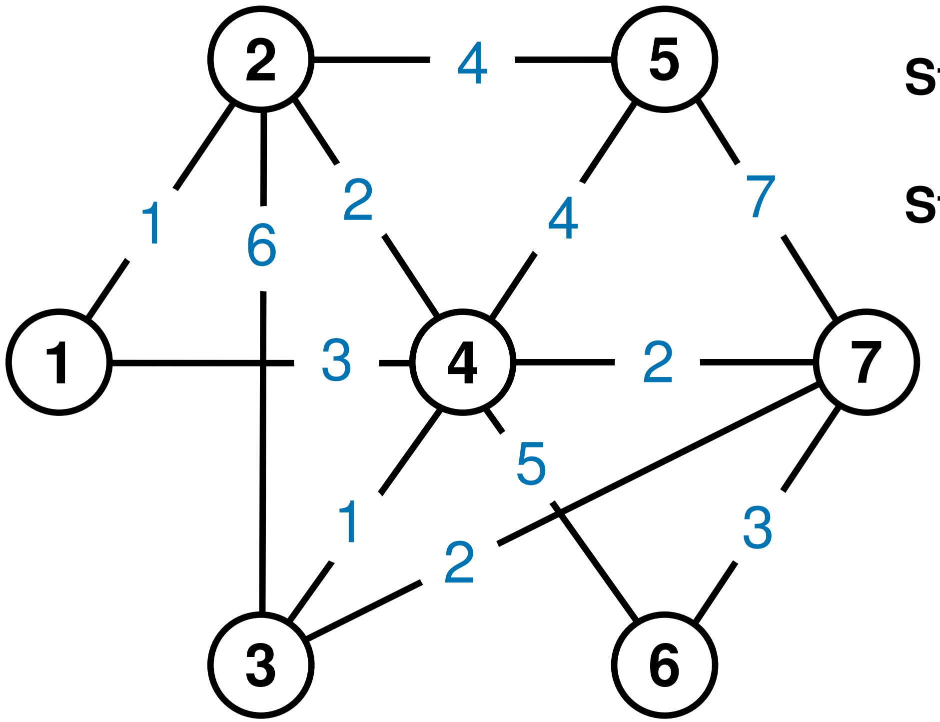



**Step 1:** For each $v \in V$, MAKESET$(v)$

# 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 $\{1, 2, 3, \ldots, |V|\}$



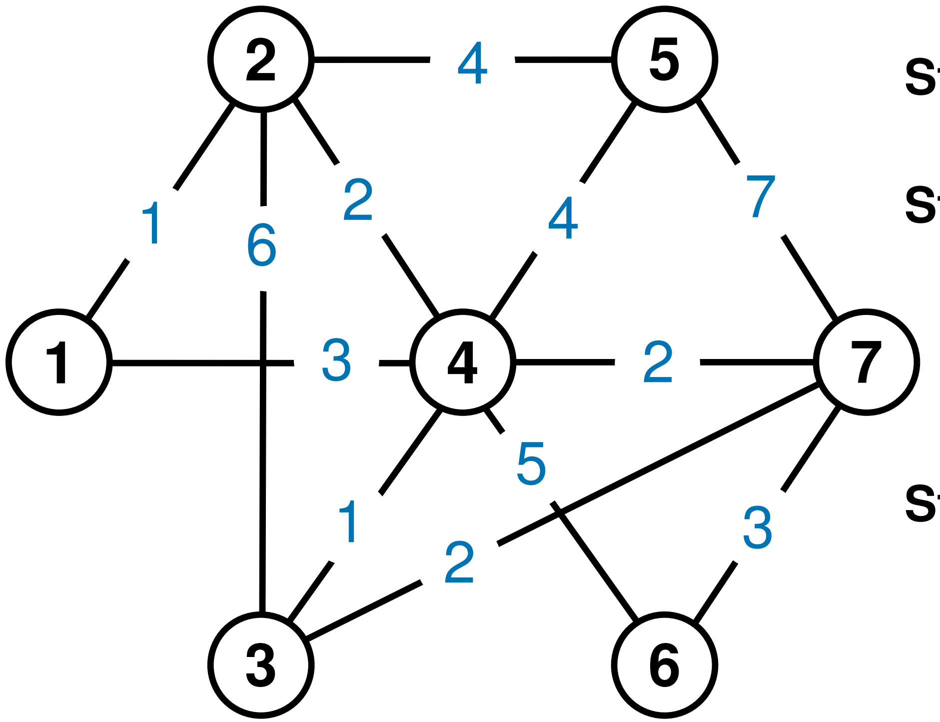

**Step 1:** For each $v \in V$, $\textsc{MakeSet}(v)$

**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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

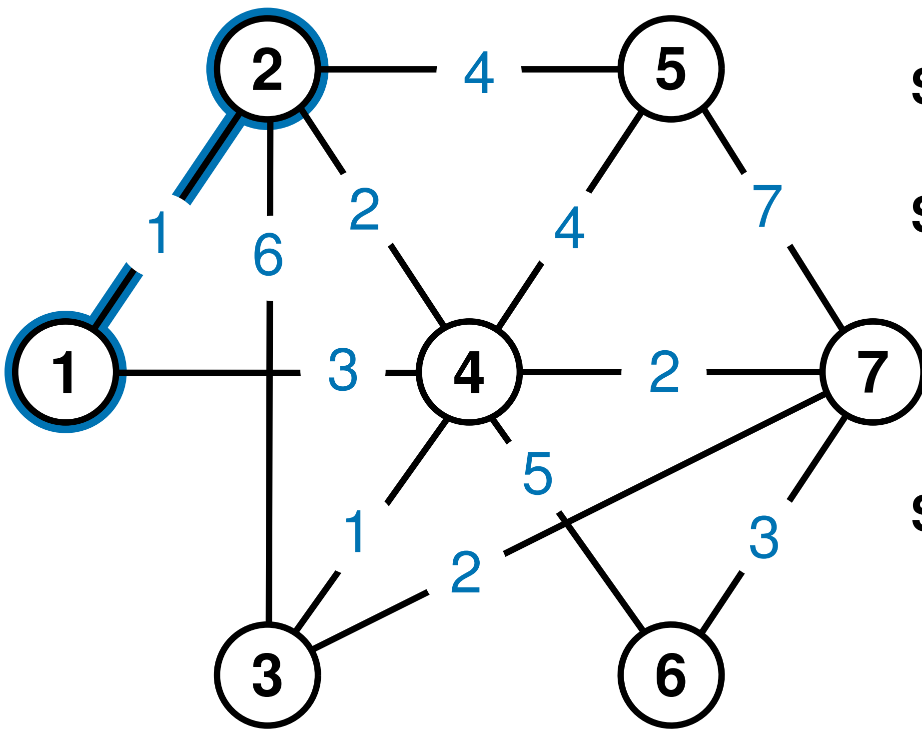
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

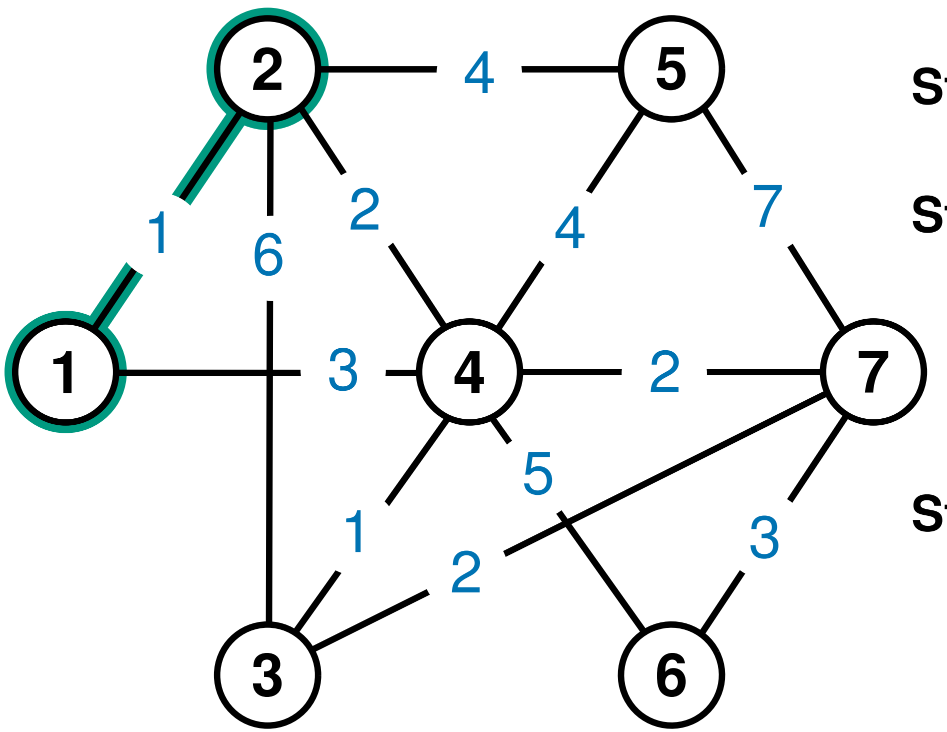
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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)*

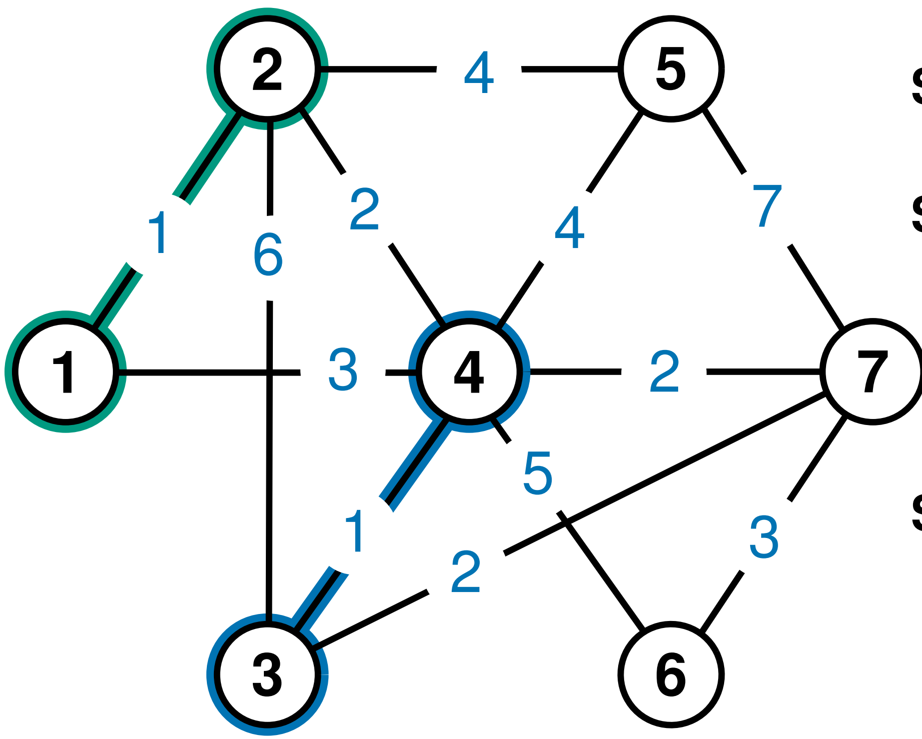
If FINDSET$(u) \neq$ FINDSET$(v)$ then

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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

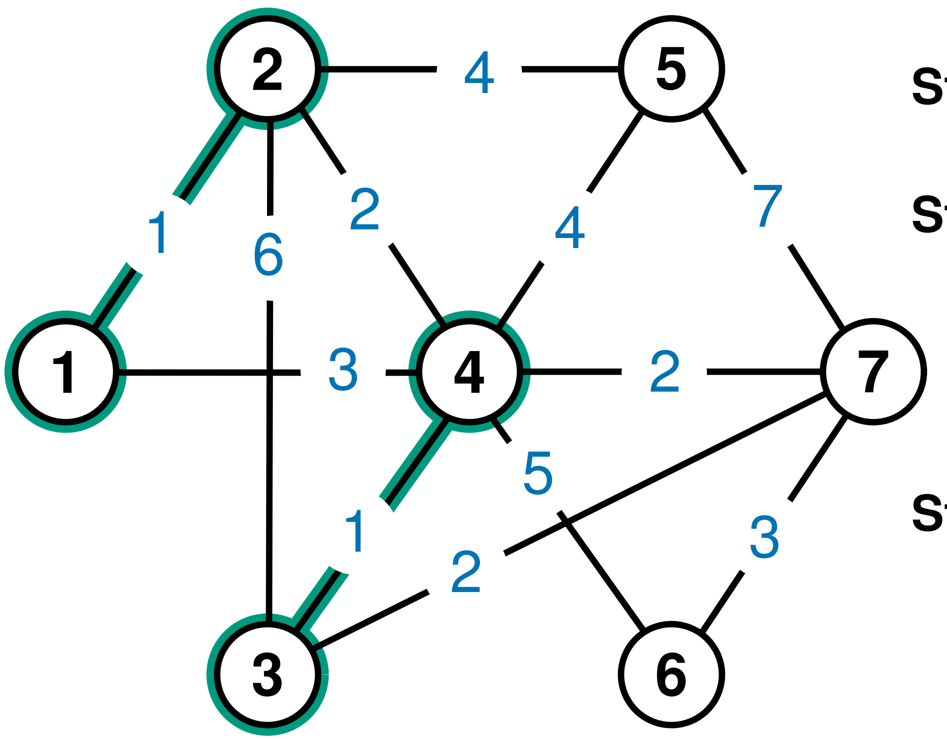
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

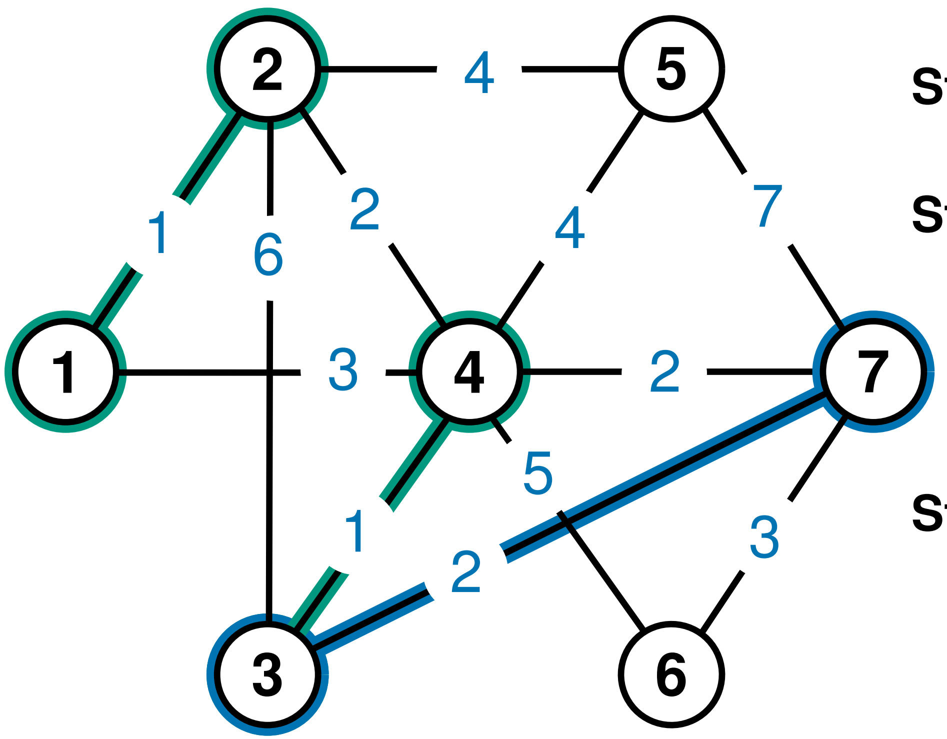
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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)*

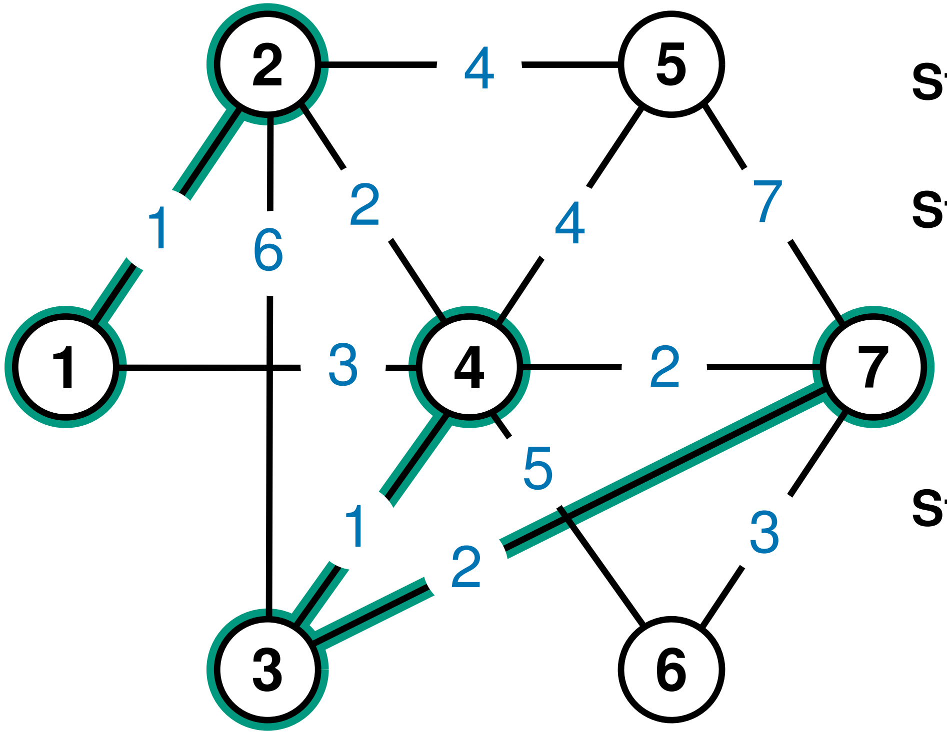
If FINDSET$(u) \neq$ FINDSET$(v)$ then

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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

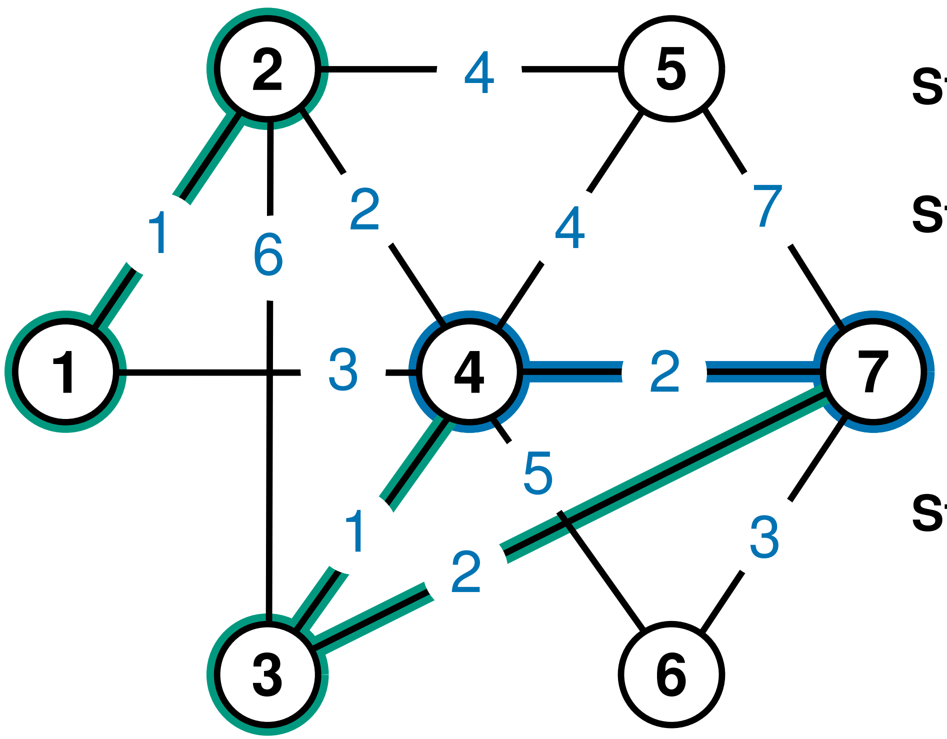
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

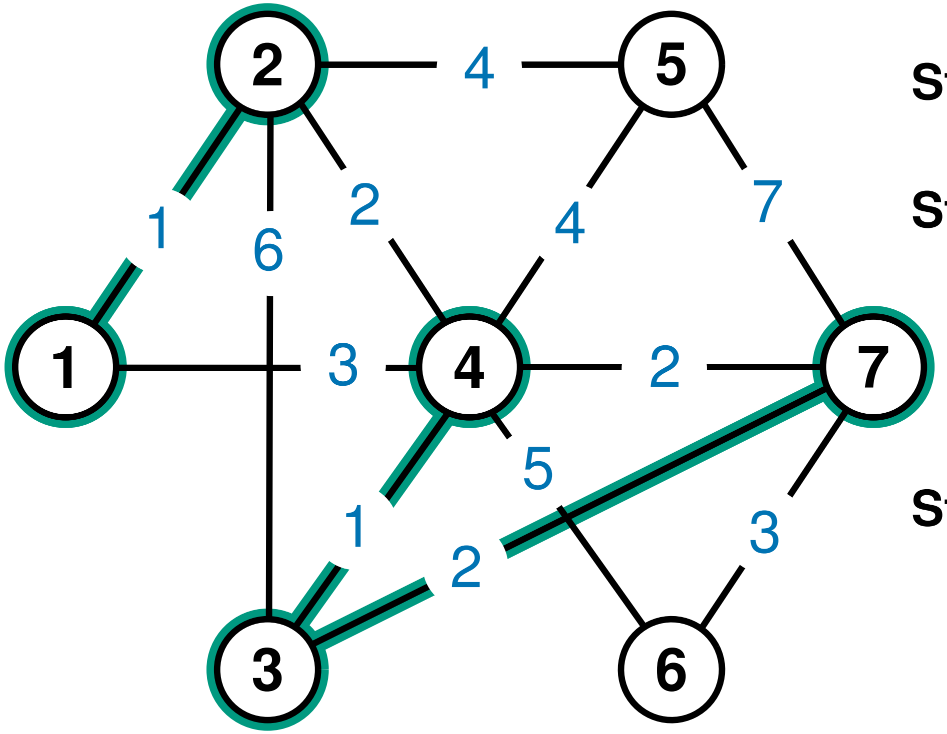
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

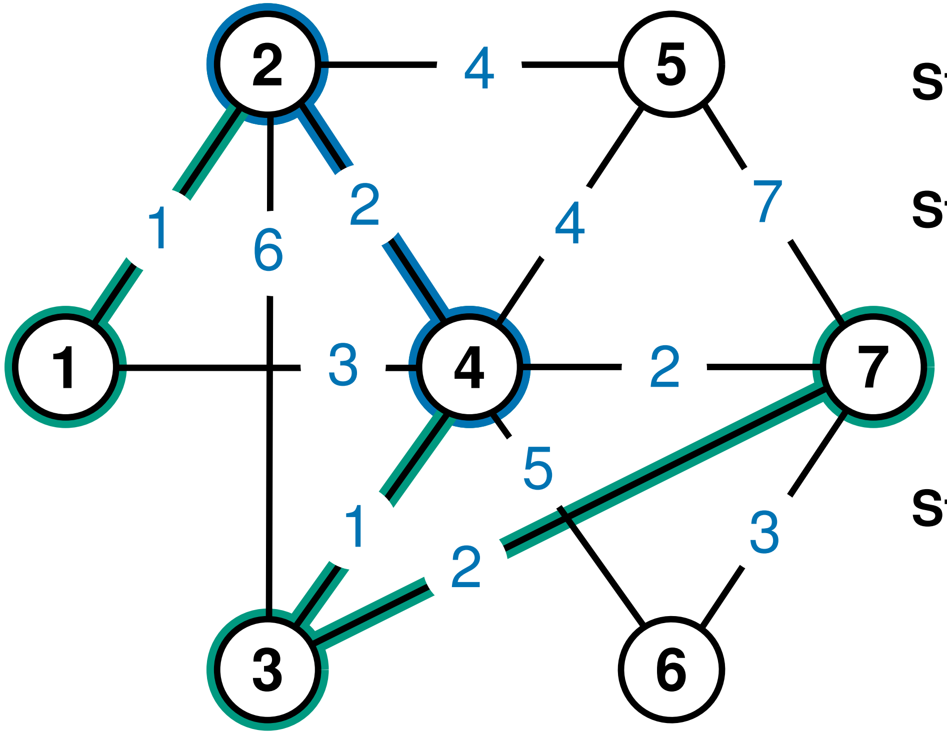
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

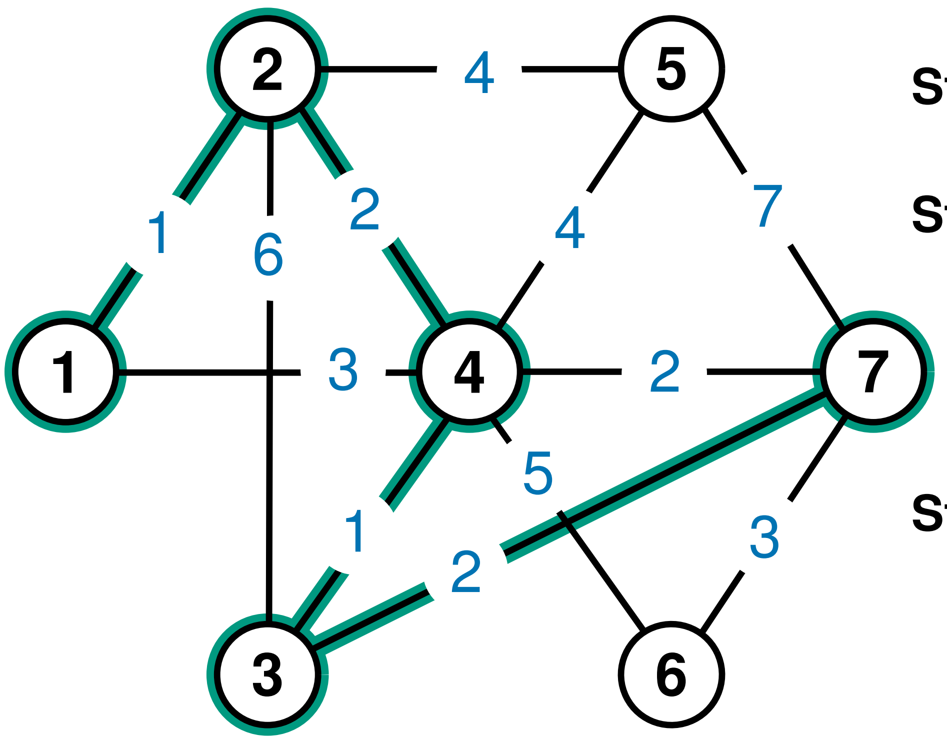
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

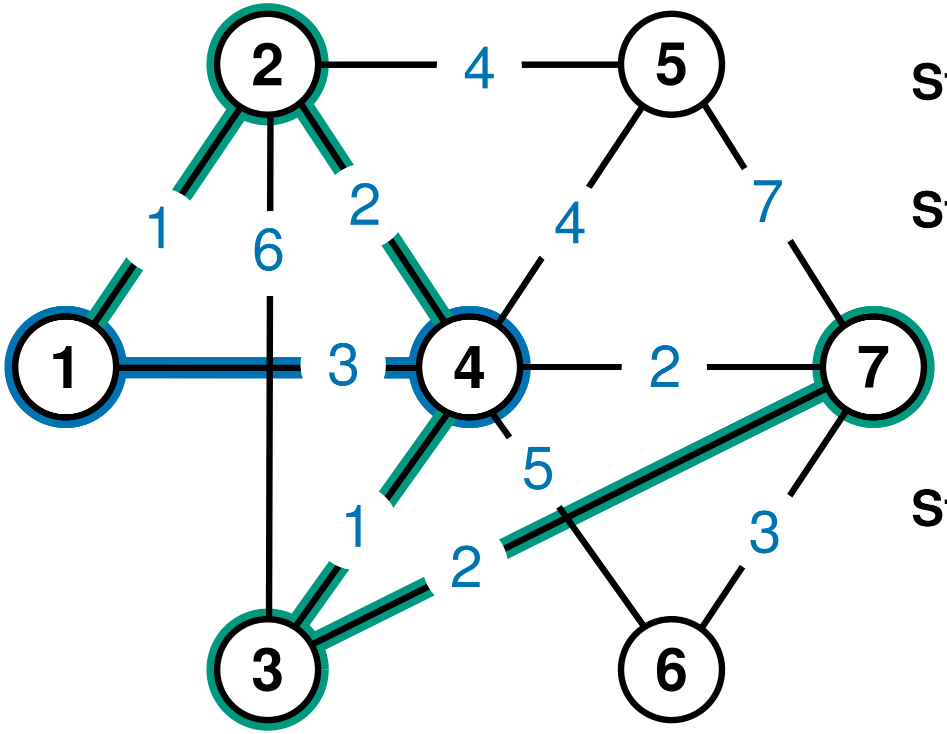
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

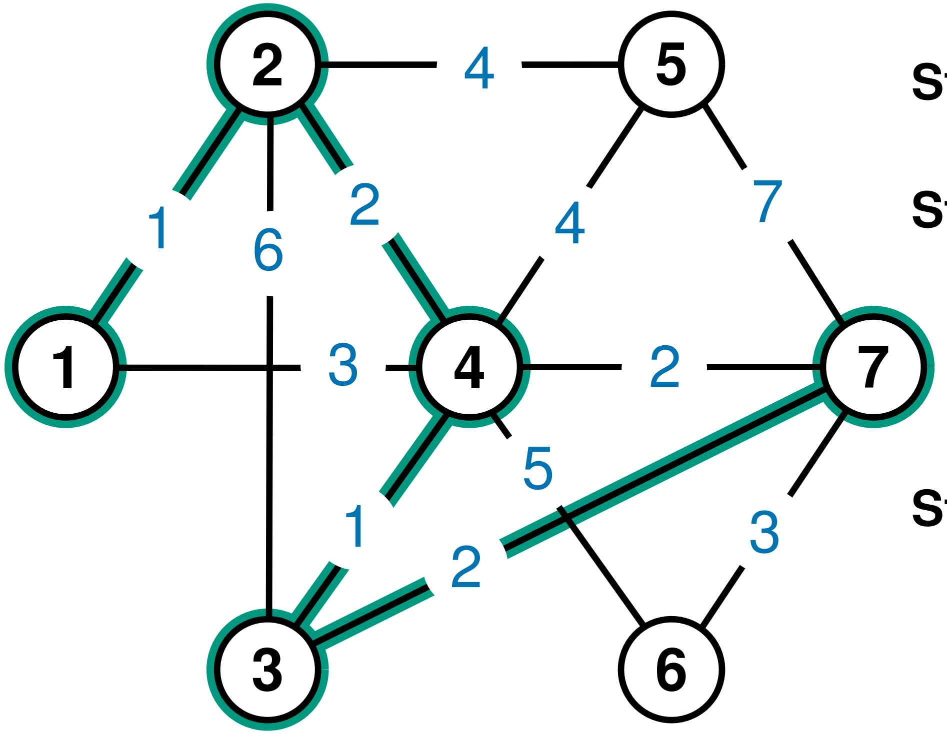
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $\{1, 2, 3, \dots, |V|\}$




**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)*

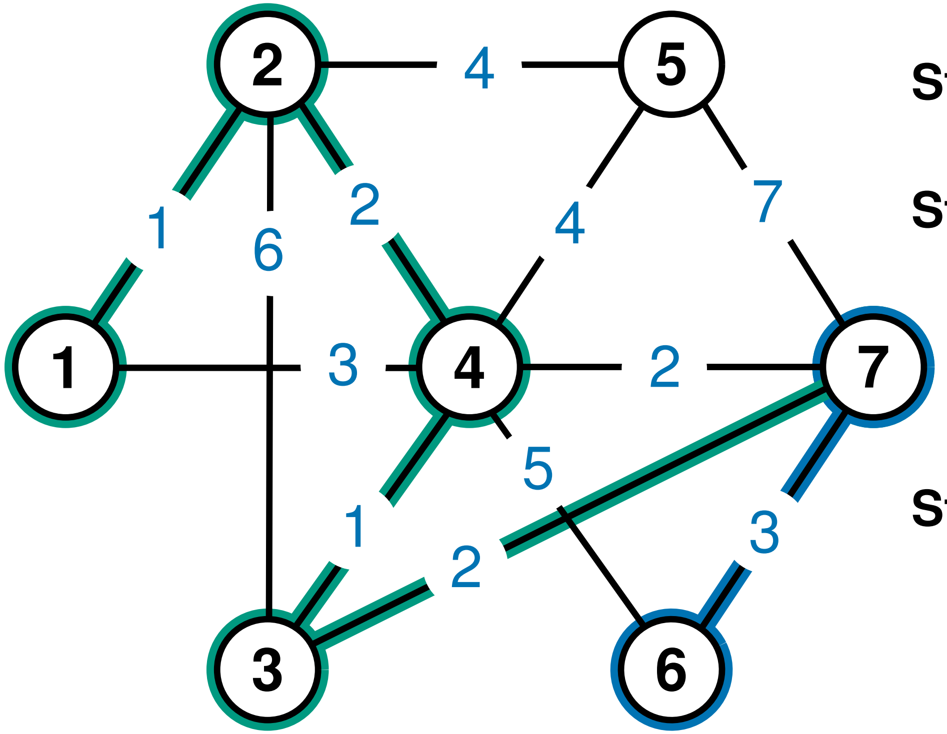
If $\text{FINDSET}(u) \neq \text{FINDSET}(v)$ then

$\text{UNION}(u, v)$ 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)*

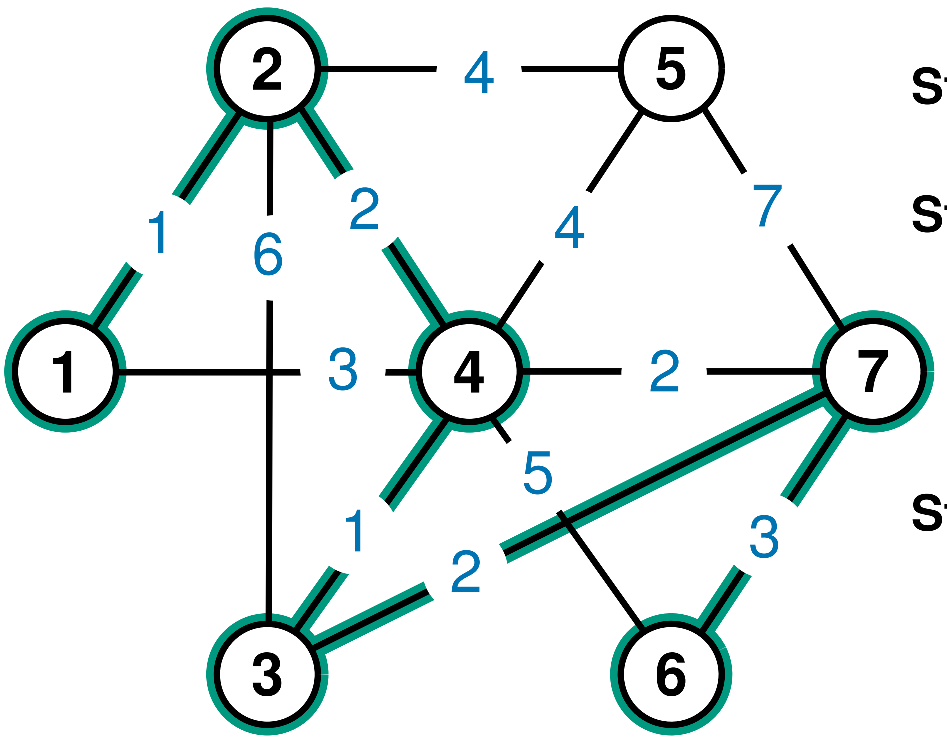
If FINDSET$(u) \neq$ FINDSET$(v)$ then

UNION$(u, v)$ and add $(u, v)$ to $T$

# Kruskal's algorithm

Kruskal's algorithm finds a minimum spanning tree in an connected, undirected graph...

using a disjoint set data structure where the elements are from $\{1, 2, 3, \ldots, |V|\}$




**Step 1:** For each $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

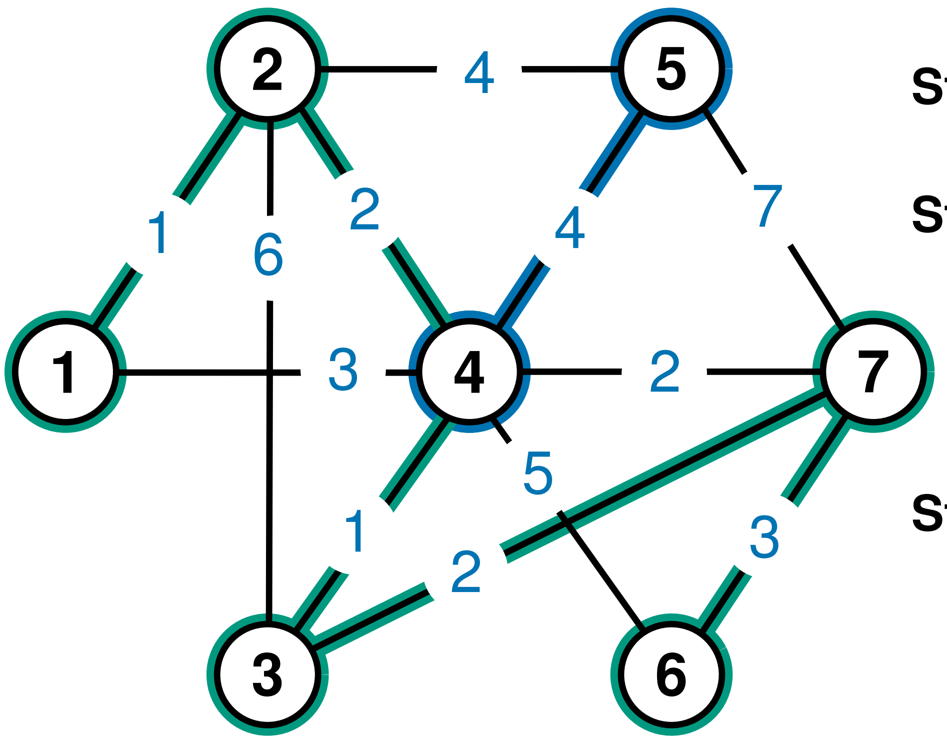
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

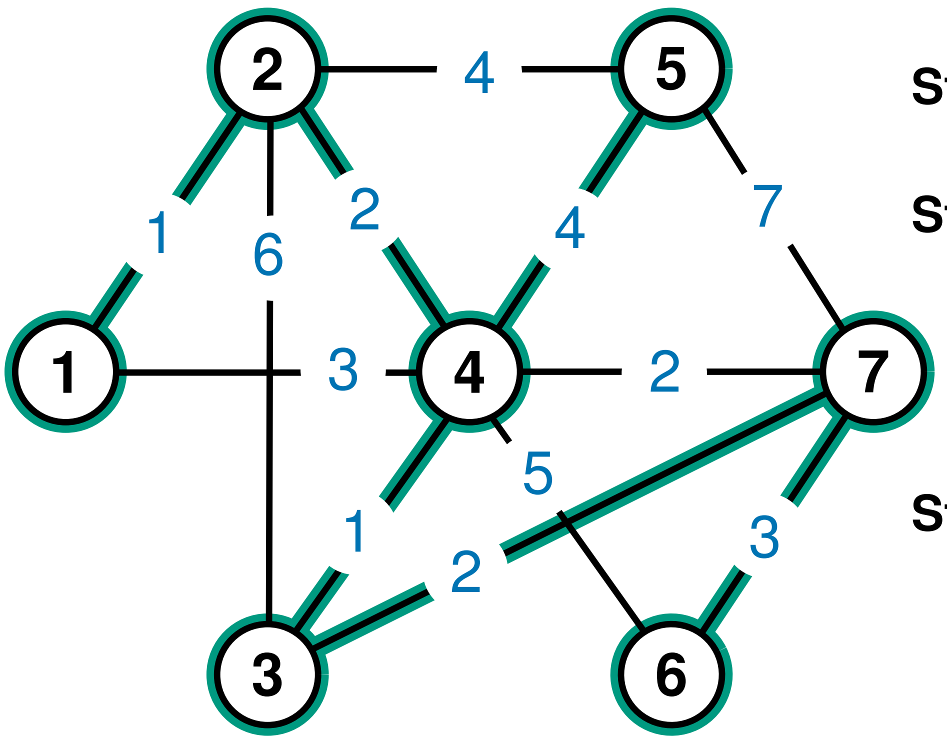
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

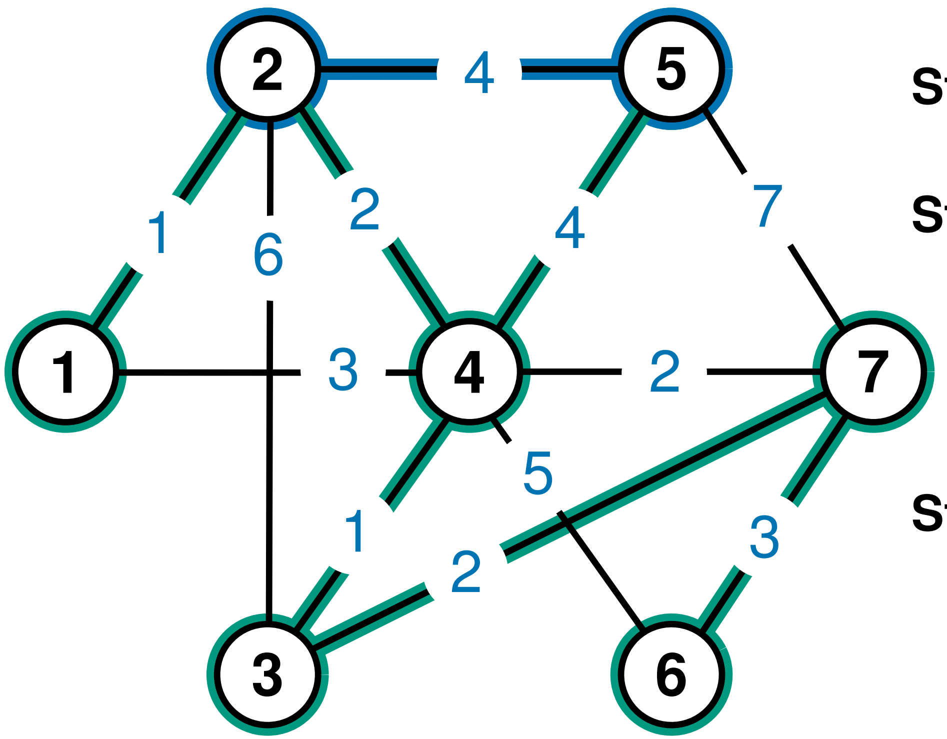
$\textsc{Union}(u, v)$ and add $(u, v)$ to $T$

University of BRISTOL

# 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

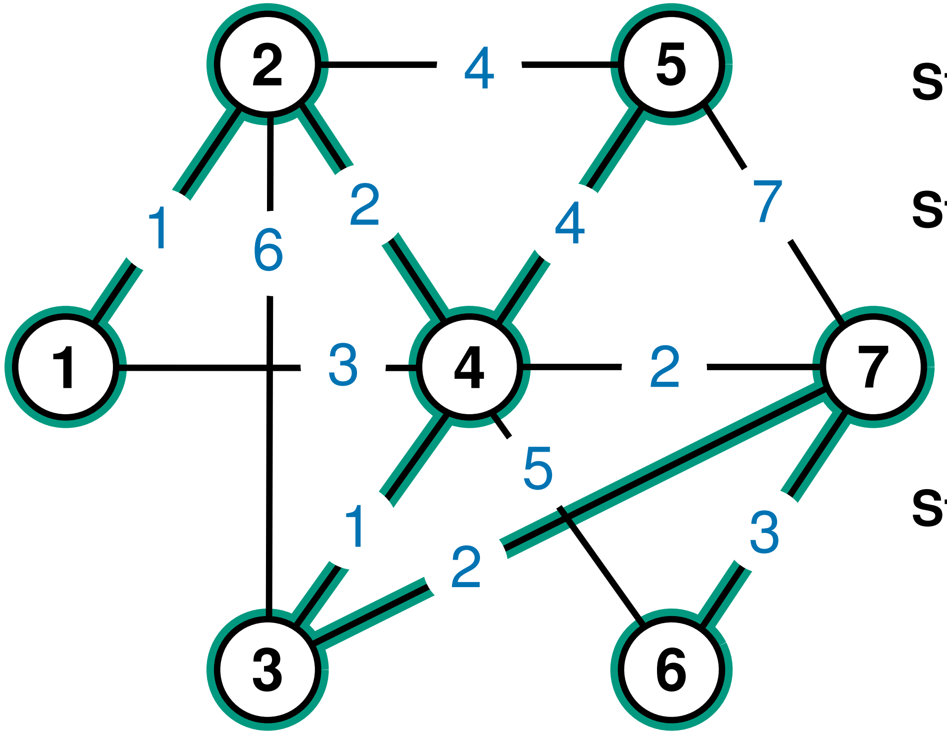
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

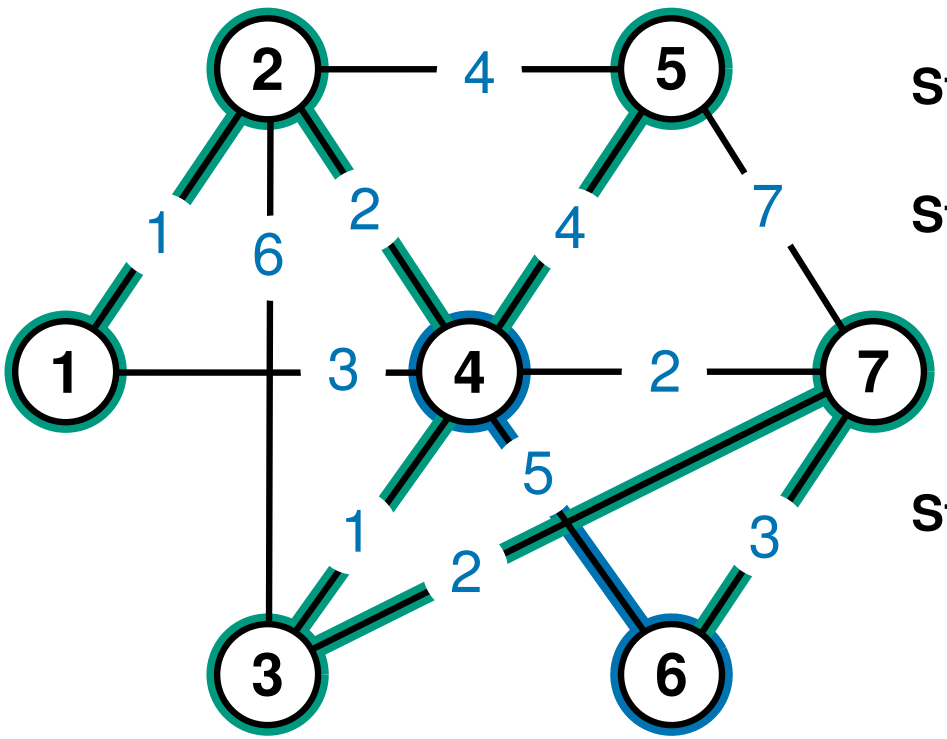
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

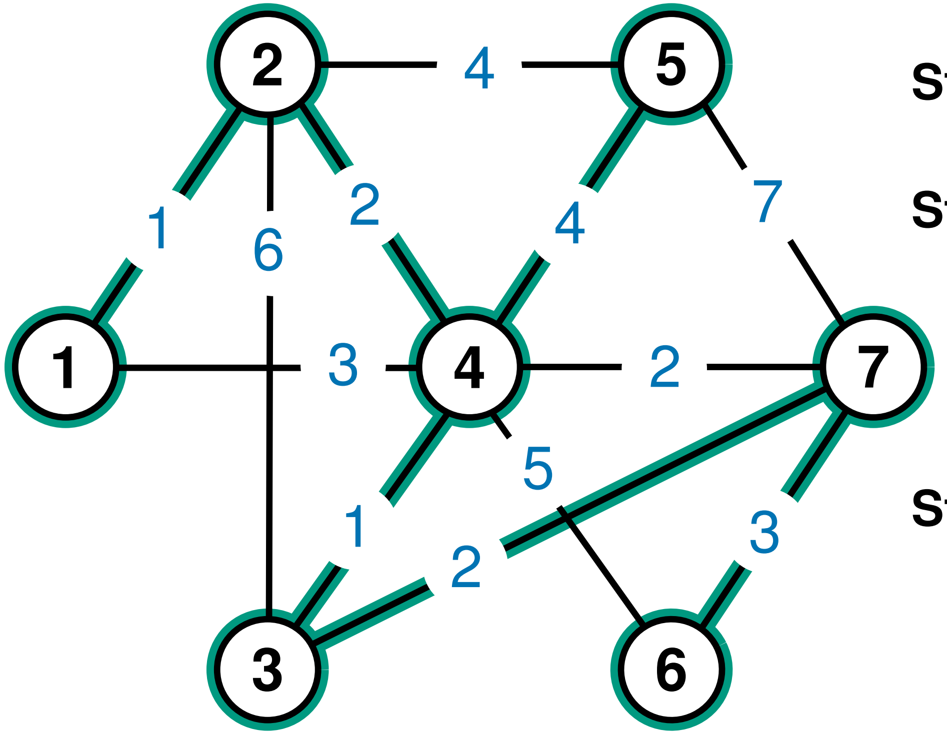
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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)*

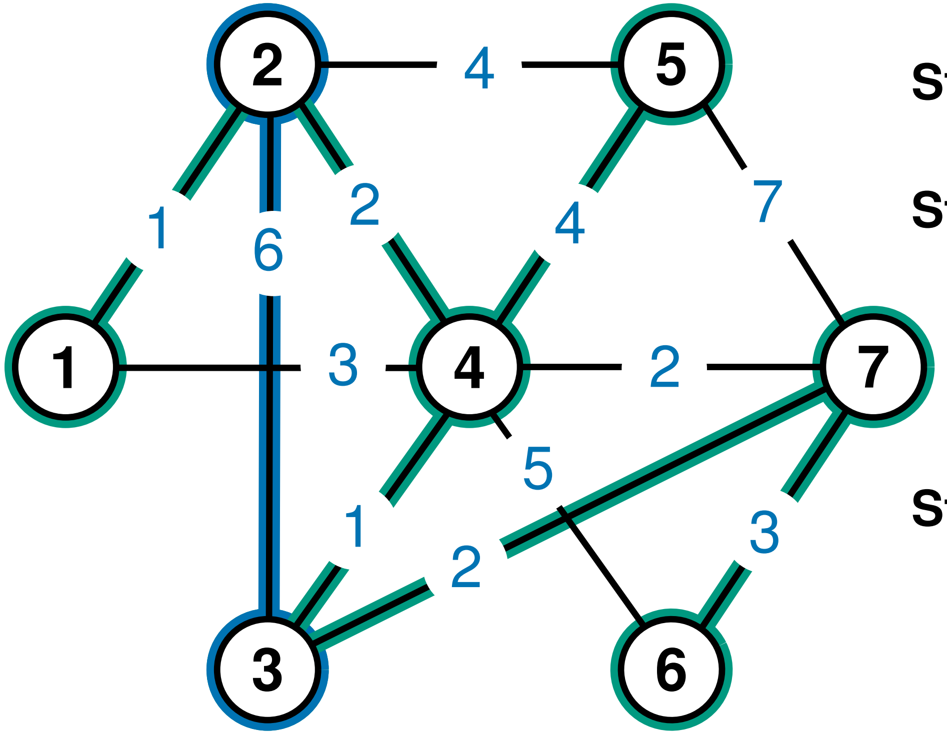
If FINDSET$(u) \neq$ FINDSET$(v)$ then

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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

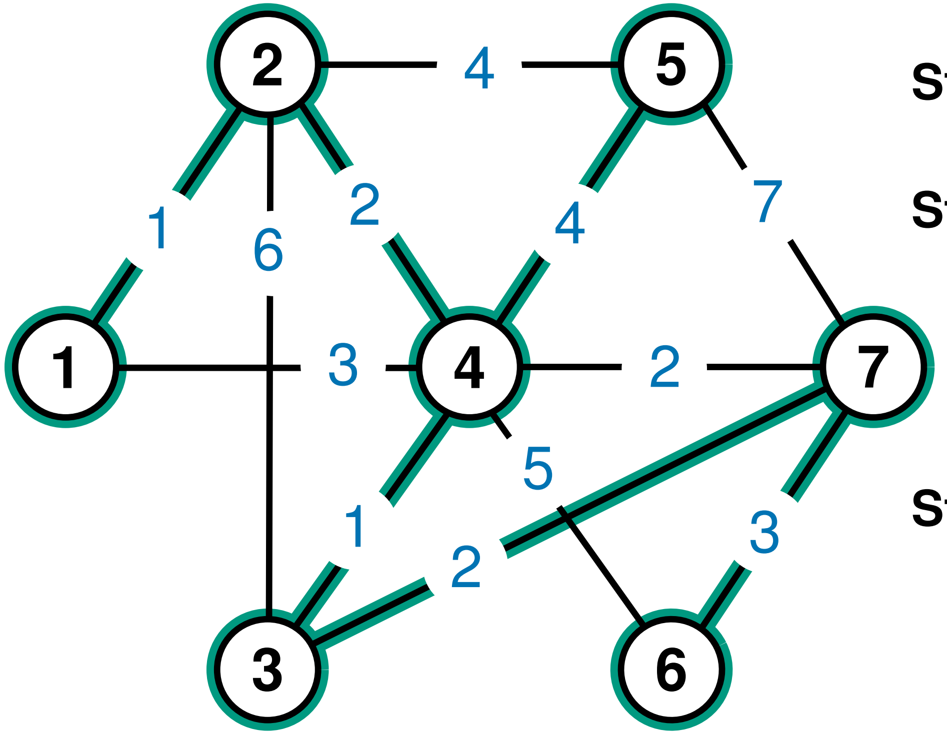
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $\{1, 2, 3, \dots, |V|\}$




**Step 1:** For each $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

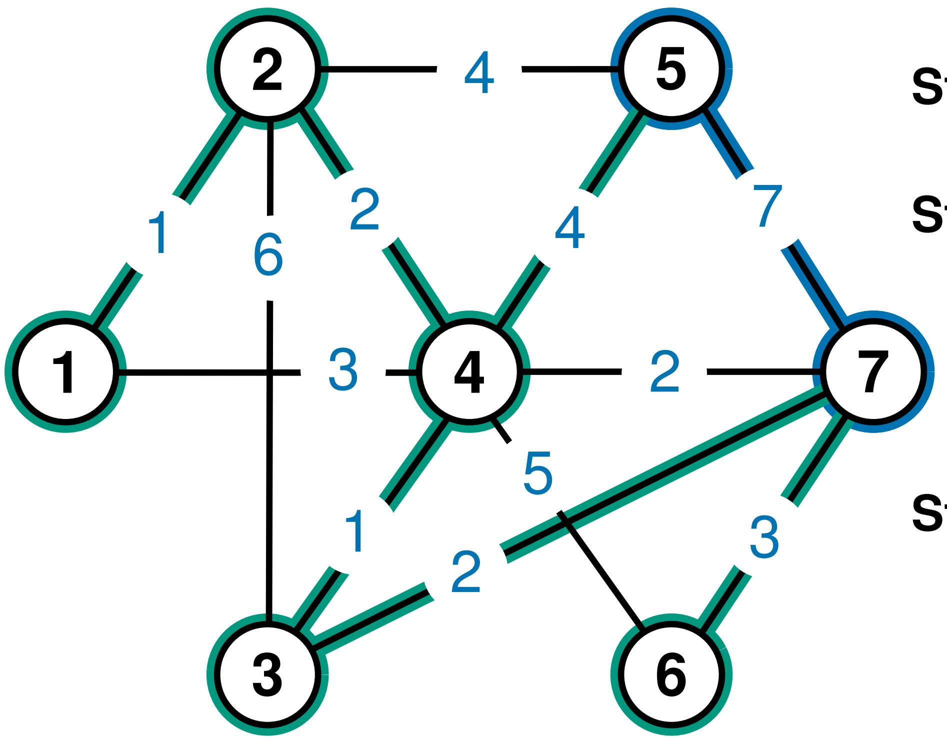
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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)*

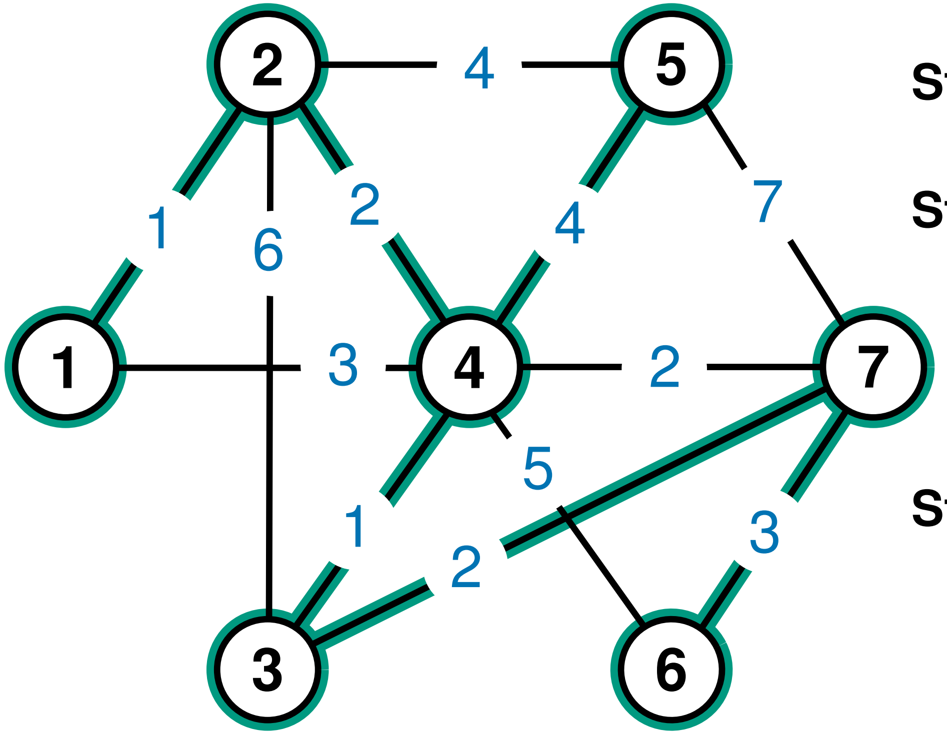
If $\text{FINDSET}(u) \neq \text{FINDSET}(v)$ then

$\text{UNION}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

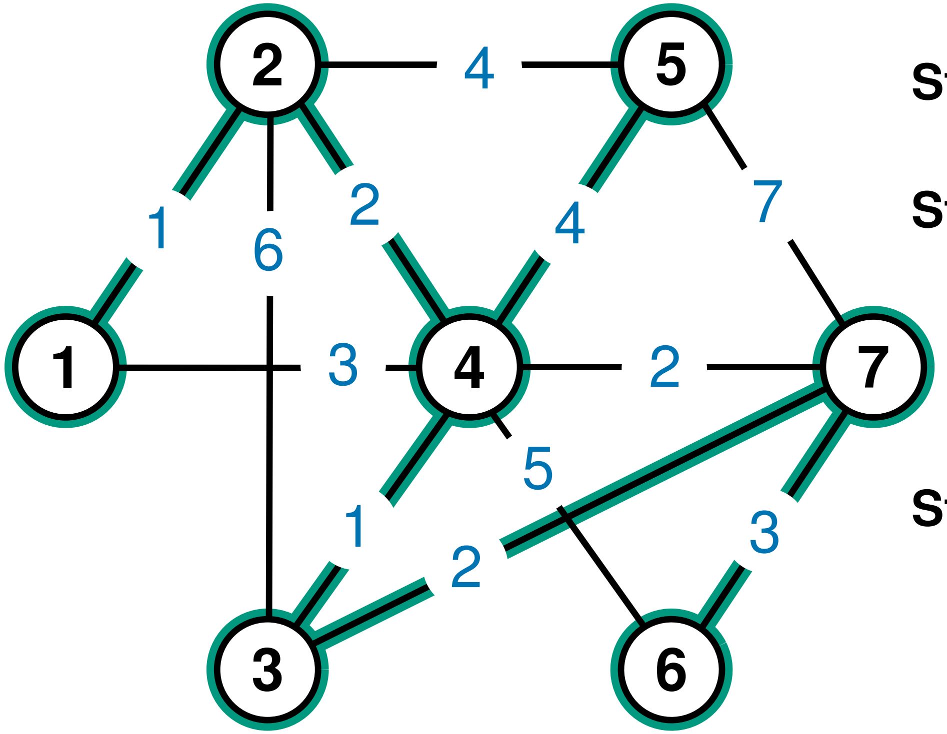
If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ 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 $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

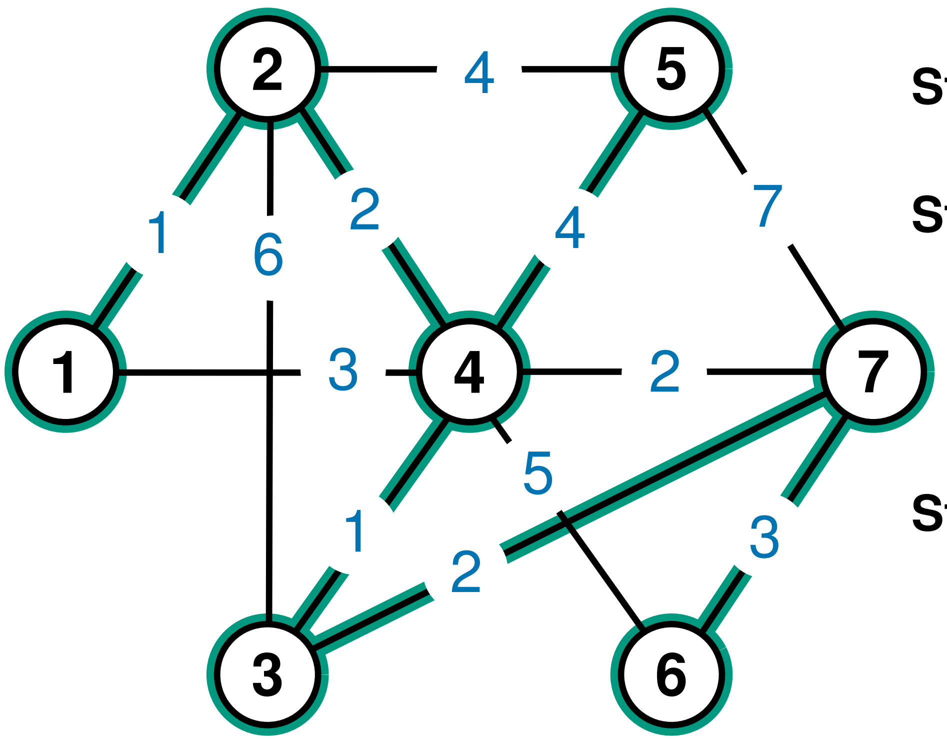
$\textsc{Union}(u, v)$ and add $(u, v)$ to $T$

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 an connected, undirected graph...

using a disjoint set data structure where the elements are from $\{1, 2, 3, \ldots, |V|\}$




**Step 1:** For each $v \in V$, $\textsc{MakeSet}(v)$

**Step 2:** Sort the edges in order of increasing weight

**Step 3:** For each $(u, v) \in E$ *(in order)*

If $\textsc{FindSet}(u) \neq \textsc{FindSet}(v)$ then

$\textsc{Union}(u, v)$ and add $(u, v)$ to $T$

If we implement the operations as we have seen, they run in $O(\log |V|)$ time

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

*(here we have omitted the proof that Kruskal always outputs a spanning tree)*

Let $M$ be any minimum spanning tree such that $M \neq K$

We will argue that there is *another* minimum spanning tree, $M_2$

*with one more edge 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 $M \neq K$

We will argue that there is *another* minimum spanning tree, $M_2$

*with one more edge in common with* $K$

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 $M \neq K$

We will argue that there is *another* minimum spanning tree, $M_2$

*with one more edge in common with* $K$

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 $M \neq K$

We will argue that there is *another* minimum spanning tree, $M_2$

*with one more edge in common with* $K$

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 $M \neq K$

We will argue that there is *another* minimum spanning tree, $M_2$
*with one more edge in common with* $K$

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 $K\ldots$

## 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 $M \neq K$

We will argue that there is *another* minimum spanning tree, $M_2$

*with one more edge in common with* $K$

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 $K$...

so there is a minimum spanning tree with $10$ 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 $M \neq K$

We will argue that there is *another* minimum spanning tree, $M_2$

*with one more edge in common with* $K$

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 $K$...

so there is a minimum spanning tree with $10$ edges in common with $K$

so there is a minimum spanning tree with $11$ edges in common w

# 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 $M$

lowest weight

# 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 $M$

lowest weight

# 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 $M \neq K$

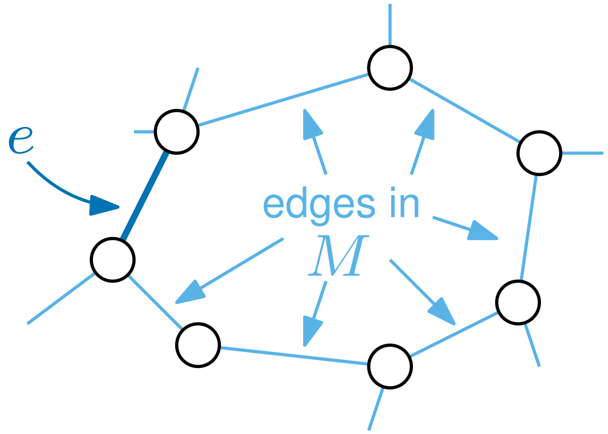
Let $e$ be the lightest edge (breaking ties arbitrarily) that is in $K$ but not in $M$

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 $M$

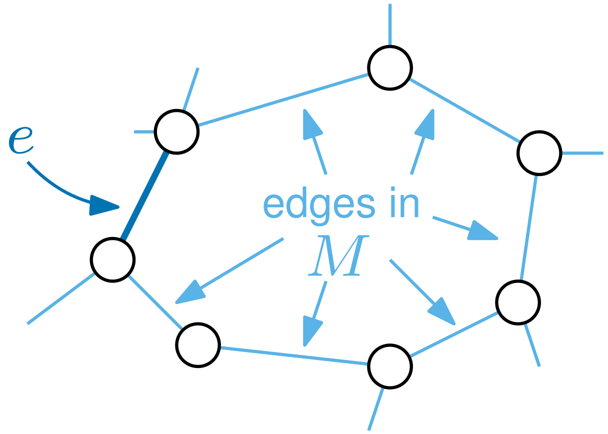



If we were to add $e$ to $M$ we would introduce a cycle.

## 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 $M$

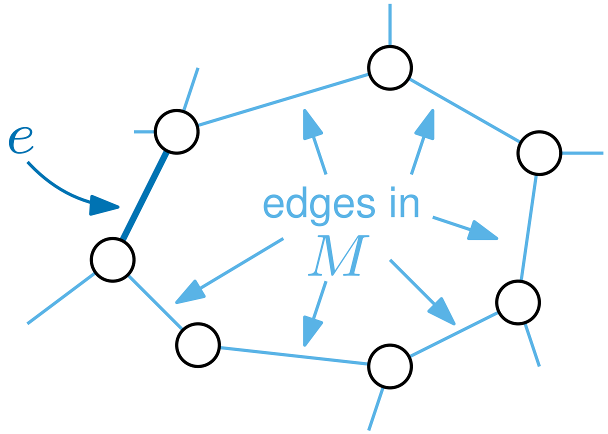



If we were to add $e$ to $M$ we would introduce a cycle.
*because $M$ is a spanning tree*

# 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 $M$



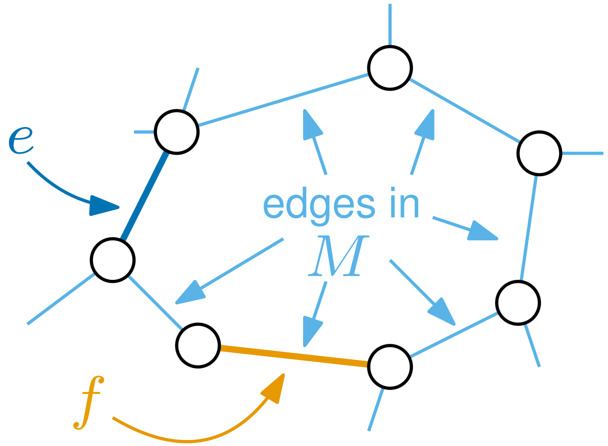

If we were to add $e$ to $M$ we would introduce a cycle.
*because $M$ is a spanning tree*

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 $M$



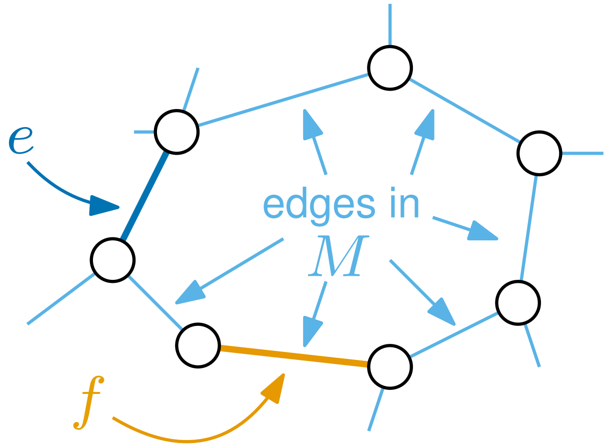

If we were to add $e$ to $M$ we would introduce a cycle.
*because $M$ is a spanning tree*

There must be an edge $f$ in this ‘potential cycle’
which is not in $K$

# 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 $M$



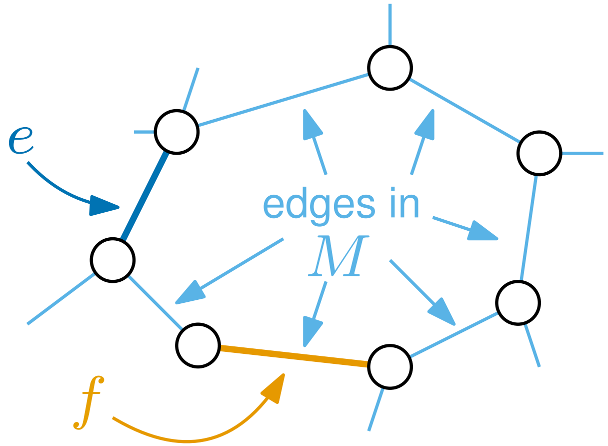

If we were to add $e$ to $M$ we would introduce a cycle.
*because $M$ is a spanning tree*

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 $M$



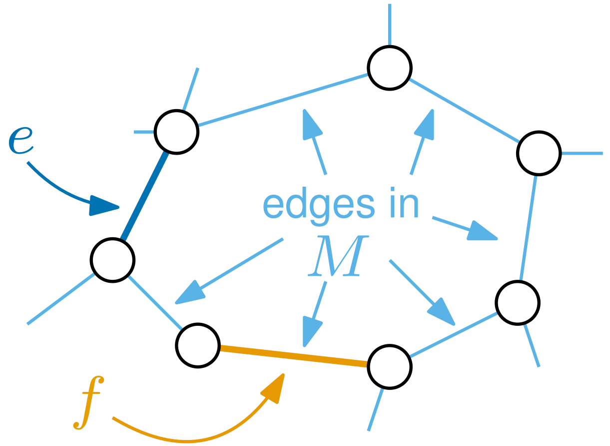

If we were to add $e$ to $M$ we would introduce a cycle.
*because $M$ is a spanning tree*

There must be an edge $f$ in this 'potential cycle' which is not in $K$
*because $K$ is a tree, so contains no cycles*

Let $M_2$ be $M$ with $e$ added and $f$ removed

# How do we make $M_2$?

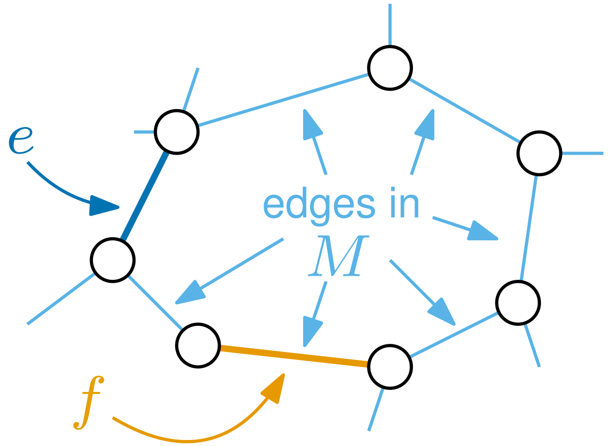
Let $K$ be the spanning tree outputted by Kruskal
and $M$ be any minimum spanning tree such that $M \neq K$



# How do we make $M_2$?

Let $K$ be the spanning tree outputted by Kruskal
and $M$ be any minimum spanning tree such that $M \neq K$

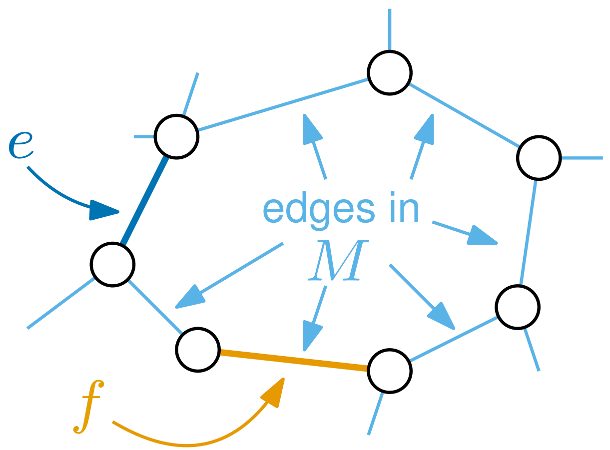
Let $M_2$ be $M$ with $e$ added and $f$ removed



# 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 $M_2$ be $M$ with $e$ added and $f$ removed

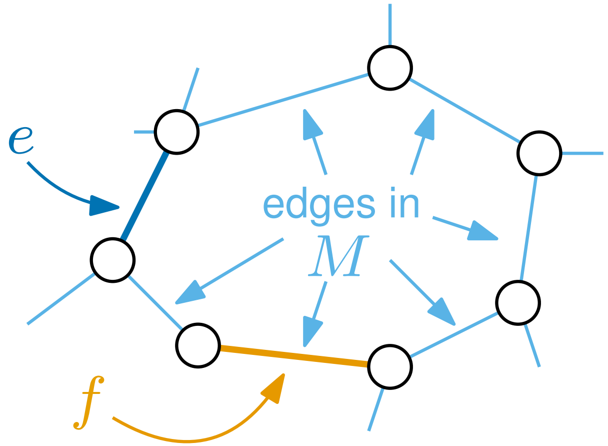



$M_2$ is a spanning tree

# 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 $M_2$ be $M$ with $e$ added and $f$ removed



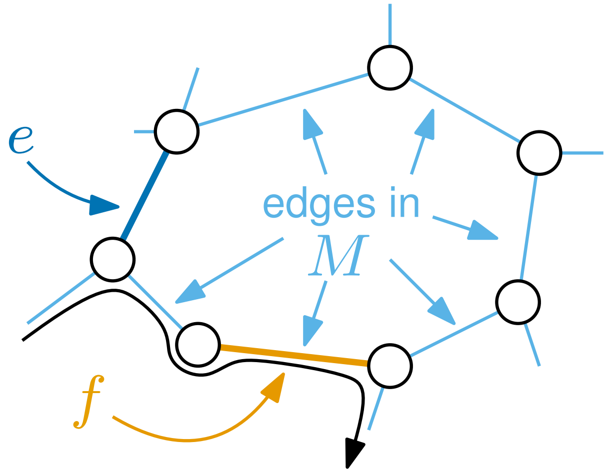

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

# 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 $M_2$ be $M$ with $e$ added and $f$ removed




$M_2$ is a spanning tree

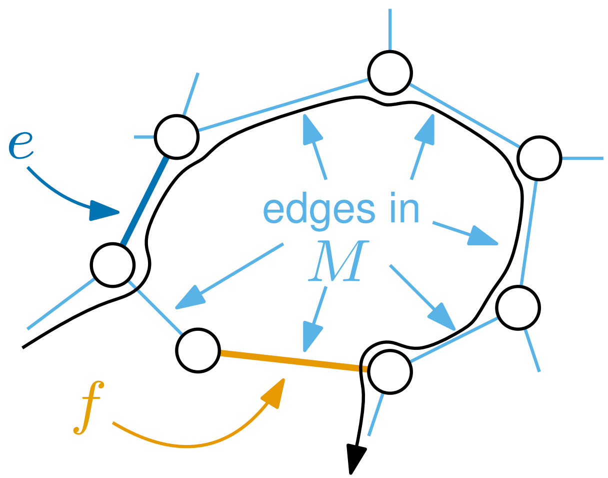
(reroute any path in $M$ that used $f$ via $e$)

# 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 $M_2$ be $M$ with $e$ added and $f$ removed




$M_2$ is a spanning tree

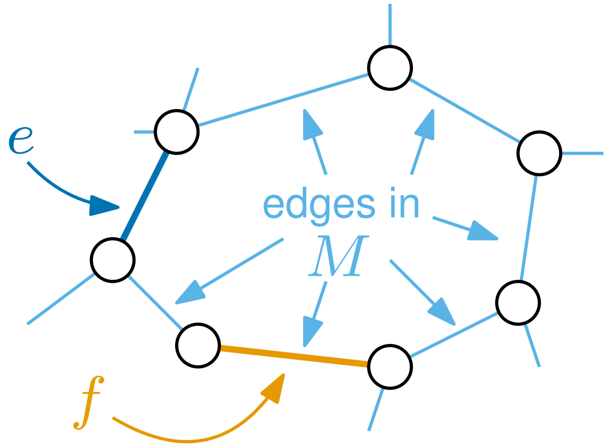
(reroute any path in $M$ that used $f$ via $e$)

# 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 $M_2$ be $M$ with $e$ added and $f$ removed



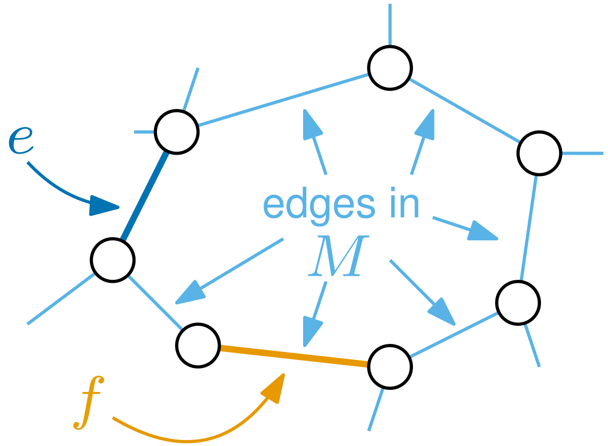

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

# 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 $M_2$ be $M$ with $e$ added and $f$ removed




(adding $e$ introduces exactly one cycle)

$M_2$ is a spanning tree

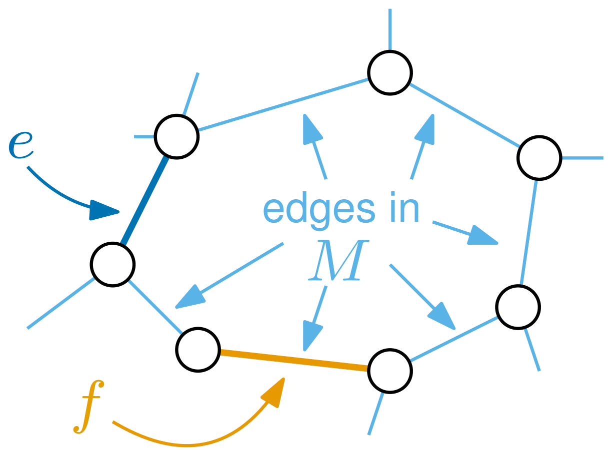
(reroute any path in $M$ that used $f$ via $e$)

# 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 $M_2$ be $M$ with $e$ added and $f$ removed




(adding $e$ introduces exactly one cycle)

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

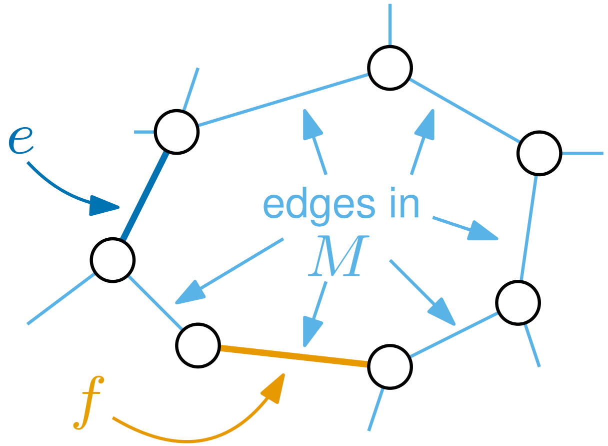
the weight of $e$ is at most the weight of $f$...

# 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 $M_2$ be $M$ with $e$ added and $f$ removed




(adding $e$ introduces exactly one cycle)

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

the weight of $e$ is at most the weight of $f$...

**Post lecture addition**

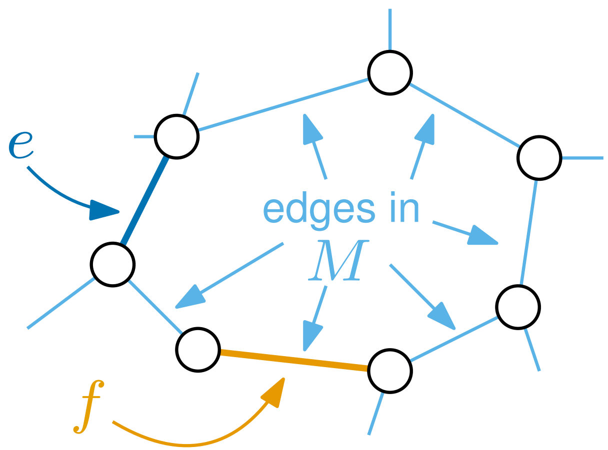
I have added a proof of this claim at the end of the slides

# 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 $M_2$ be $M$ with $e$ added and $f$ removed




(adding $e$ introduces exactly one cycle)

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

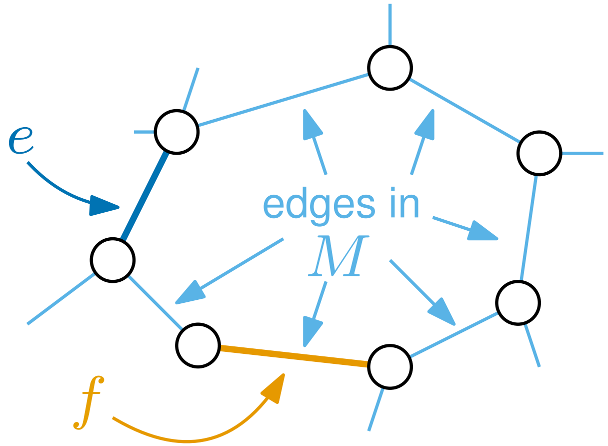
the weight of $e$ is at most the weight of $f$...

# 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 $M_2$ be $M$ with $e$ added and $f$ removed




(adding $e$ introduces exactly one cycle)

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

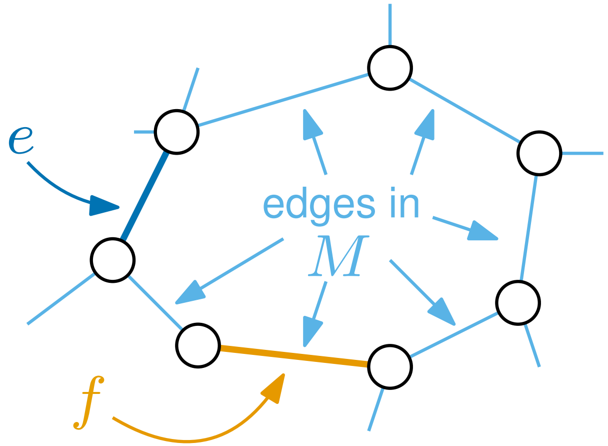
the weight of $e$ is at most the weight of $f$...

so $M_2$ is 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 $M_2$ be $M$ with $e$ added and $f$ removed




(adding $e$ introduces exactly one cycle)

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

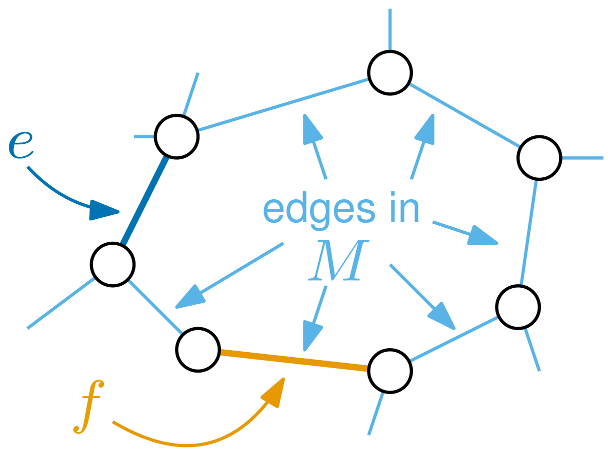
the weight of $e$ is at most the weight of $f$...
so $M_2$ is a *minimum* spanning tree.

$e$ 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 $M_2$ be $M$ with $e$ added and $f$ removed




(adding $e$ introduces exactly one cycle)

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

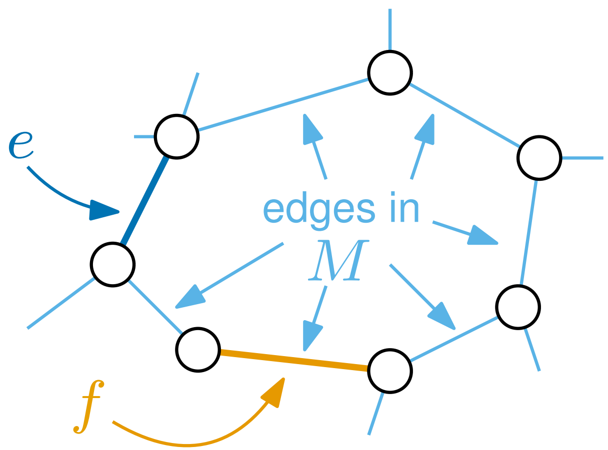
the weight of $e$ is at most the weight of $f$...
so $M_2$ is a *minimum* spanning tree.

$e$ is in $K$ but $f$ is not in $K$...
*so $M_2$ has one more edge in common with $K$*
*(than $M$ has 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 $M \neq K$

Let $M_2$ be $M$ with $e$ added and $f$ removed




(adding $e$ introduces exactly one cycle)

$M_2$ is a spanning tree

(reroute any path in $M$ that used $f$ via $e$)

the weight of $e$ is at most the weight of $f$...
so $M_2$ is a *minimum* spanning tree.

$e$ is in $K$ but $f$ is not in $K$...
*so $M_2$ has one more edge in common with $K$*
*(than $M$ has in common with $K$)*

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 numbers from* $\{1, 2, \ldots, n\}$

The operations UNION and FINDSET run in $O(\log n)$ time

and the operation MAKESET runs in $O(1)$ time.

We then saw Kruskal's algorithm

which finds a minimum spanning tree in an 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*

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

## Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$

This is because Kruskal considers edges in increasing weight order
so every edge in $K'$ is no heavier than $f$ which is in turn lighter than $e$

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$
This is because Kruskal considers edges in increasing weight order
so every edge in $K'$ is no heavier than $f$ which is in turn lighter than $e$

Every edge in $K'$ is in $K$ *(Kruskal's doesn't remove edges)*

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$
This is because Kruskal considers edges in increasing weight order
so every edge in $K'$ is no heavier than $f$ which is in turn lighter than $e$

Every edge in $K'$ is in $K$ *(Kruskal's doesn't remove edges)*

Every edge in $K'$ is in $M$

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$

This is because Kruskal considers edges in increasing weight order
so every edge in $K'$ is no heavier than $f$ which is in turn lighter than $e$

Every edge in $K'$ is in $K$ *(Kruskal's doesn't remove edges)*

Every edge in $K'$ is in $M$

Otherwise, there is an edge in $K'$ which is lighter than $e$ and contained in $K$ but not in $M$.
This would contradict the definition of $e$.

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$
This is because Kruskal considers edges in increasing weight order
so every edge in $K'$ is no heavier than $f$ which is in turn lighter than $e$

Every edge in $K'$ is in $K$ *(Kruskal's doesn't remove edges)*

Every edge in $K'$ is in $M$

Otherwise, there is an edge in $K'$ which is lighter than $e$ and contained in $K$ but not in $M$.
This would contradict the definition of $e$.

Kruskal's algorithm rejects $f$ because it would create a cycle in $K'$

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$
This is because Kruskal considers edges in increasing weight order
so every edge in $K'$ is no heavier than $f$ which is in turn lighter than $e$

Every edge in $K'$ is in $K$ *(Kruskal's doesn't remove edges)*

Every edge in $K'$ is in $M$

Otherwise, there is an edge in $K'$ which is lighter than $e$ and contained in $K$ but not in $M$.
This would contradict the definition of $e$.

Kruskal's algorithm rejects $f$ because it would create a cycle in $K'$

**But** $f$ *is* in $M$ and every edge in $K'$ is in $M$

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$

This is because Kruskal considers edges in increasing weight order
so every edge in $K'$ is no heavier than $f$ which is in turn lighter than $e$

Every edge in $K'$ is in $K$ *(Kruskal's doesn't remove edges)*

Every edge in $K'$ is in $M$

Otherwise, there is an edge in $K'$ which is lighter than $e$ and contained in $K$ but not in $M$.
This would contradict the definition of $e$.

Kruskal's algorithm rejects $f$ because it would create a cycle in $K'$

**But** $f$ *is* in $M$ and every edge in $K'$ is in $M$

So $M$ contains a cycle,

# Why is the weight of $e$ at most the weight of $f$?

*(this slide was added after the lecture)*

For a contradiction, assume that $f$ is lighter than $e$.

Consider pausing Kruskal's algorithm at the moment
just after edge $f$ has been considered *(and rejected)*.

Let $K'$ be the set of edges chosen by Kruskal's algorithm at this moment.

Every edge in $K'$ is lighter than $e$
This is because Kruskal considers edges in increasing weight order
so every edge in $K'$ is no heavier than $f$ which is in turn lighter than $e$

Every edge in $K'$ is in $K$ *(Kruskal's doesn't remove edges)*

Every edge in $K'$ is in $M$

Otherwise, there is an edge in $K'$ which is lighter than $e$ and contained in $K$ but not in $M$.
This would contradict the definition of $e$.

Kruskal's algorithm rejects $f$ because it would create a cycle in $K'$

**But** $f$ *is* in $M$ and every edge in $K'$ is in $M$

So $M$ contains a cycle, *Contradiction!*