Warning:

In order to find something, we must assume that variables / addresses always receive a value before they are accessed.

Complexity:

we havve:

$\mathcal{O}(\#\, edges + \#\, Vars)$	calls of	union*
$\mathcal{O}(\#\, edges + \#\, Vars)$	calls of	find
$\mathcal{O}(\#\, Vars)$	calls of	union

$\Longrightarrow$ We require efficient Union-Find data-structure :-)

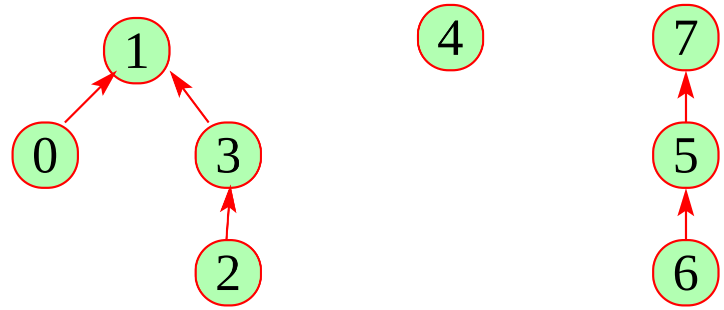
Idea:

Represent partition of U as directed forest:

- For $u \in U$ a reference $F[u]$ to the father is maintained;
- Roots are elements u with $F[u] = u$.

Single trees represent equivalence classes.

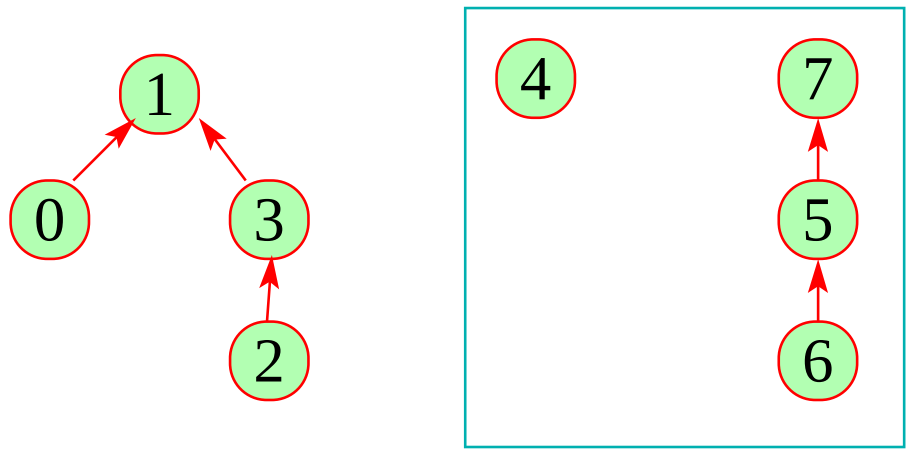
Their roots are their representatives ...



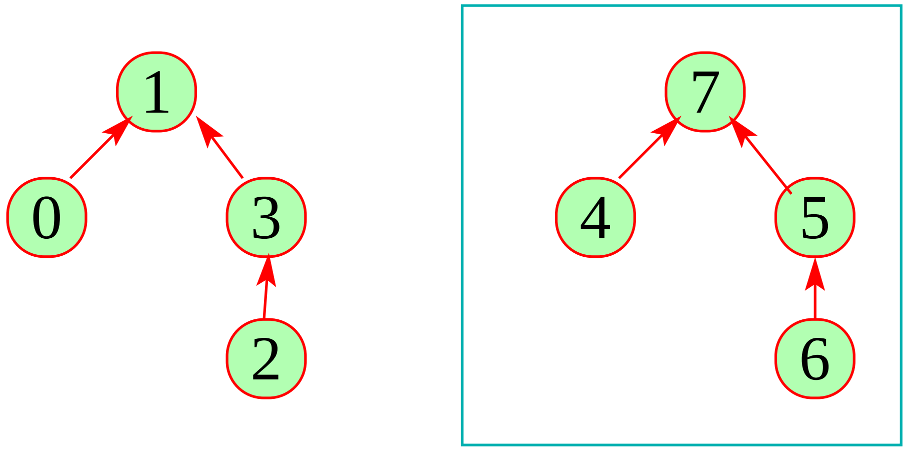
0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

$\longrightarrow$ find (π, u) follows the father references :-)

$\longrightarrow$ union (π, u_1, u_2) re-directs the father reference of one u_i ...



0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7



0	1	2	3	4	5	6	7

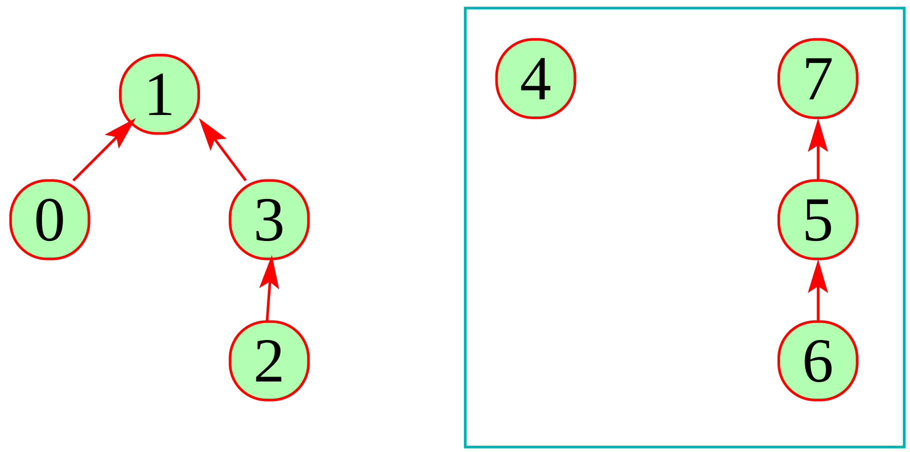
1	1	3	1	7	7	5	7

The Costs:

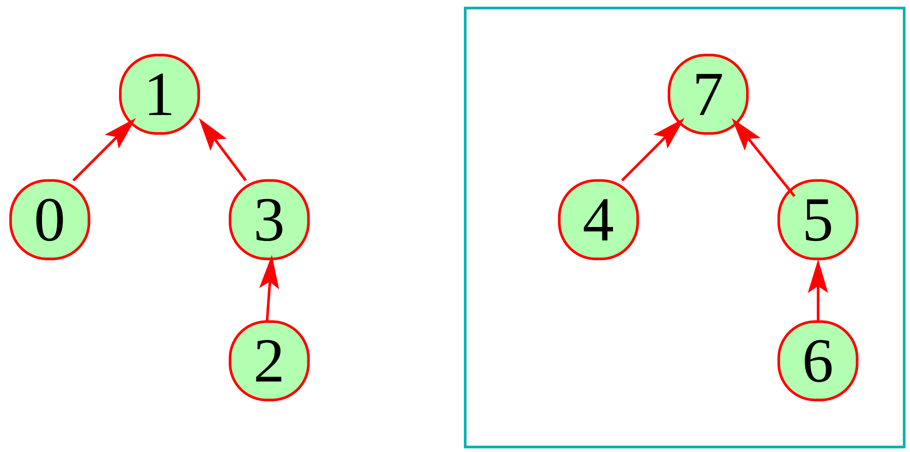
union	:	$\mathcal{O}(1)$	:-)
find	:	$\mathcal{O}(depth(\pi))$	:-(

Strategy to Avoid Deep Trees:

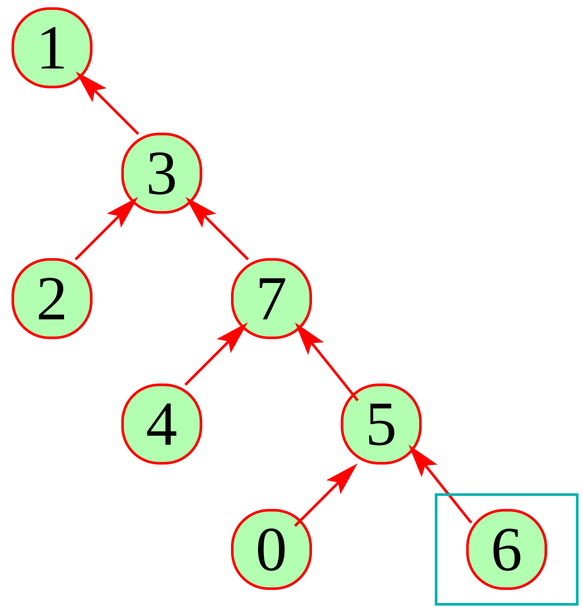
- Put the smaller tree below the bigger !
- Use find to compress paths ...



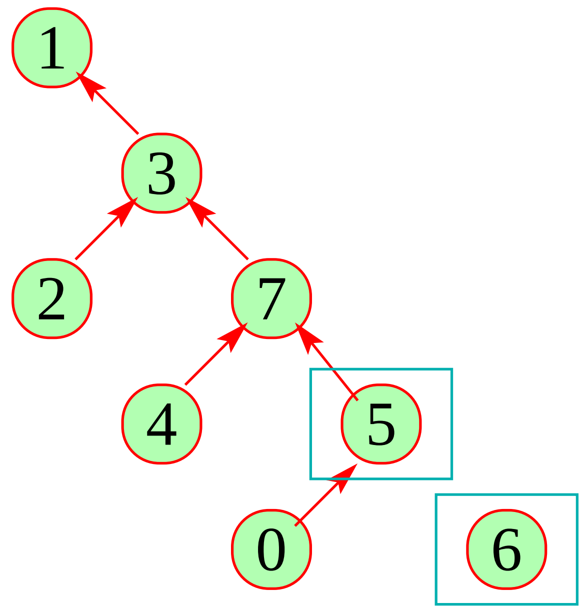
0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7



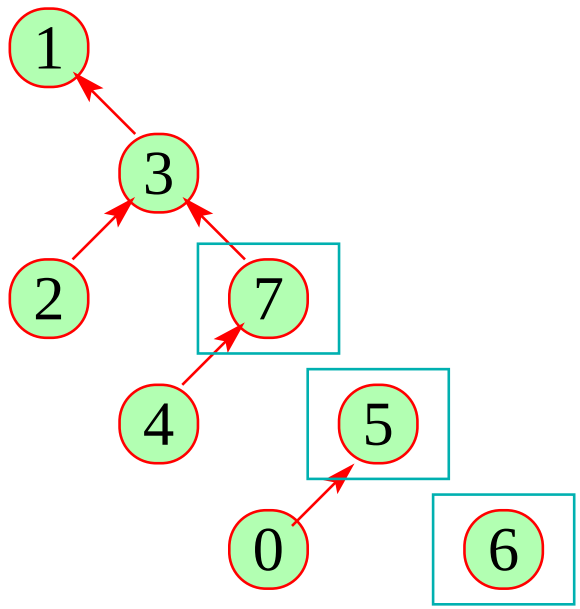
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7



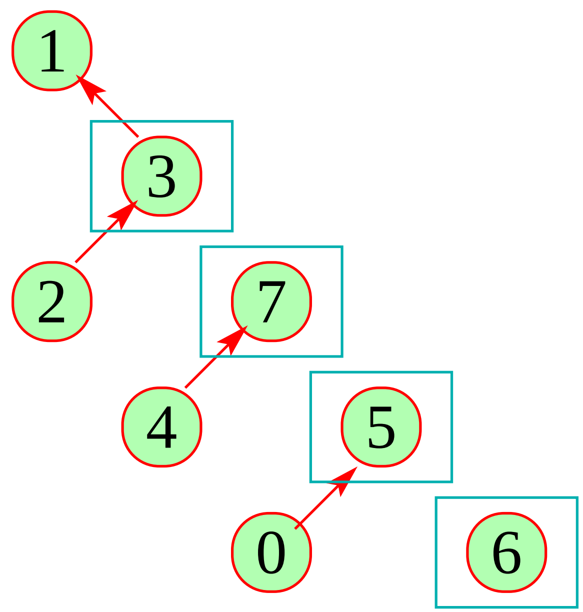
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



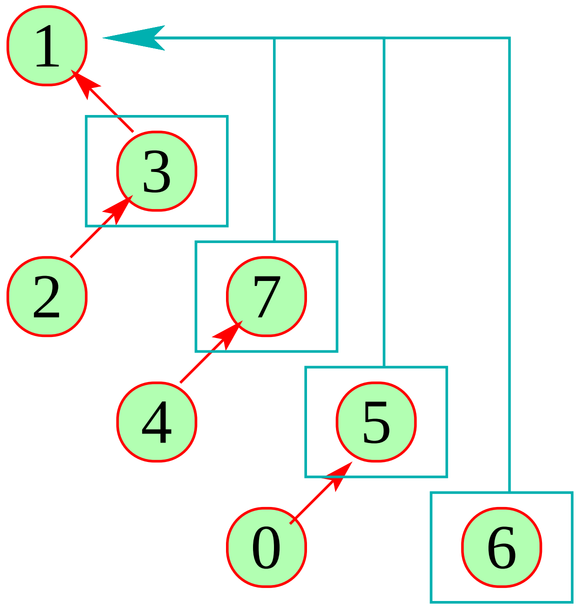
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	1	7	1	1

Robert Endre Tarjan, Princeton

Note:

- By this data-structure, n union- und m find operations require time $\mathcal{O}(n + m \cdot \alpha(n,n))$

  // α the inverse Ackermann-function :-)

- For our application, we only must modify union such that roots are from *Vars* whenever possible.
- This modification does not increase the asymptotic run-time. :-)

Summary:

The analysis is extremely fast — but may not find very much.

Background 3: Fixpoint Algorithms

Consider: $x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n$

Observation:

RR-Iteration is inefficient:

→ We require a complete round in order to detect termination :-(

→ If in some round, the value of just one unknown is changed, then we still re-compute all :-(

→ The practical run-time depends on the ordering on the variables :-(

Idea: Worklist Iteration

If an unknown x_i changes its value, we re-compute all unknowns which depend on x_i. Technically, we require:

→ the lists $Dep\, f_i$ of unknowns which are accessed during evaluation of f_i. From that, we compute the lists:

$$I[x_i] = \{x_j \mid x_i \in Dep\, f_j\}$$

i.e., a list of all x_j which depend on the value of x_i;

→ the values $D[x_i]$ of the x_i where initially $D[x_i] = \bot$;

→ a list W of all unknowns whose value must be recomputed ...

The Algorithm:

$$
\begin{array}{l}
W = [x_1, \ldots, x_n]; \\
\textsf{while}\ (W \neq [\,])\ \{ \\
\qquad \begin{array}{lcl}
x_i & = & \textsf{extract}\ W; \\
t & = & f_i\ \textsf{eval}; \\
t & = & D[x_i] \sqcup t; \\
\end{array} \\
\qquad \textsf{if}\ (t \neq D[x_i])\ \{ \\
\qquad\qquad \begin{array}{lcl}
D[x_i] & = & t; \\
W & = & \textsf{append}\ I[x_i]\ W; \\
\end{array} \\
\qquad \} \\
\}
\end{array}
$$

$$
\text{where :} \quad \mathit{eval}\ x_j \quad = \quad D[x_j]
$$

Example:

$$x_1 \supseteq \{a\} \cup x_3$$
$$x_2 \supseteq x_3 \cap \{a, b\}$$
$$x_3 \supseteq x_1 \cup \{c\}$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

Example:

$$x_1 \supseteq \{a\} \cup x_3$$
$$x_2 \supseteq x_3 \cap \{a, b\}$$
$$x_3 \supseteq x_1 \cup \{c\}$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

$D[x_1]$	$D[x_2]$	$D[x_3]$	W
$\emptyset$	$\emptyset$	$\emptyset$	$\boxed{x_1}, x_2, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_2}, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_3}$
$\{a\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_1}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_3}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_2}$
$\{a, c\}$	$\{a\}$	$\{a, c\}$	[ ]

Theorem

Let $x_i \sqsupseteq f_i(x_1, \ldots, x_n)$, $i = 1, \ldots, n$ denote a constraint system over the complete lattice $\mathbb{D}$ of hight $h > 0$.

(1) The algorithm terminates after at most $h \cdot N$ evaluations of right-hand sides where

$$N = \sum_{i=1}^{n} (1 + \#(Dep\ f_i)) \qquad // \quad \text{size of the system} \quad \text{:-)}$$

(2) The algorithm returns a solution.
If all f_i are monotonic, it returns the least one.

Proof:

Ad (1):

Every unknown x_i may change its value at most h times :-)

Each time, the list $I[x_i]$ is added to W .

Thus, the total number of evaluations is:

$$\begin{array}{ll} \leq & n + \sum_{i=1}^{n} (h \cdot \# (I[x_i])) \\ = & n + h \cdot \sum_{i=1}^{n} \# (I[x_i]) \\ = & n + h \cdot \sum_{i=1}^{n} \# (Dep\ f_i) \\ \leq & h \cdot \sum_{i=1}^{n} (1 + \# (Dep\ f_i)) \\ = & h \cdot N \end{array}$$

Ad (2):

We only consider the assertion for monotonic f_i .

Let D_0 denote the least solution. We show:

- $D_0[x_i] \sqsupseteq D[x_i]$ (all the time)
- $D[x_i] \not\sqsupseteq f_i\,\mathsf{eval} \implies x_i \in W$ (at exit of the loop body)
- On termination, the algo returns a solution :-))

Discussion:

- In the example, fewer evaluations of right-hand sides are required than for RR-iteration :-)
- The algo also works for non-monotonic f_i :-)
- For monotonic f_i, the algo can be simplified:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{;}$$

- In presence of widening, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t = D[x_i] \sqcup\!\!\!\!\_ \; t;}$$

- In presence of Narrowing, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t = D[x_i] \sqcap\!\!\!\!\!\bar{\ } \; t;}$$

Warning:

- The algorithm relies on explicit dependencies among the unknowns.

  So far in our applications, these were obvious. This need not always be the case :-(

- We need some strategy for extract which determines the next unknown to be evaluated.

- It would be ingenious if we always evaluated first and then accessed the result ... :-)

$\Longrightarrow$ recursive evaluation ...

Idea:

→ If during evaluation of f_i, an unknown x_j is accessed, x_j is first solved recursively. Then x_i is added to $I[x_j]$:-)

$$
\begin{aligned}
\textsf{eval}\ x_i\ x_j \quad = \quad & \textsf{solve}\ x_j; \\
& I[x_j] = I[x_j] \cup \{x_i\}; \\
& D[x_j];
\end{aligned}
$$

→ In order to prevent recursion to descend infinitely, a set *Stable* of unknown is maintained for which solve just looks up their values :-)

Initially, $\mathit{Stable} = \emptyset$ ...

The Function solve :

$$
\begin{array}{lll}
\text{solve } x_i & = & \text{if } (x_i \notin \mathit{Stable}) \; \{ \\
 & & \quad \mathit{Stable} = \mathit{Stable} \cup \{x_i\}; \\
 & & \quad t = f_i \, (\text{eval } \; x_i); \\
 & & \quad t = D[x_i] \sqcup t; \\
 & & \quad \text{if } (t \neq D[x_i]) \; \{ \\
 & & \qquad W = I[x_i]; \quad I[x_i] = \emptyset; \\
 & & \qquad D[x_i] = t; \\
 & & \qquad \mathit{Stable} = \mathit{Stable} \backslash W; \\
 & & \qquad \text{app } \; \text{solve } \; W; \\
 & & \quad \} \\
 & & \}
\end{array}
$$

Helmut Seidl, TU München ;-)

Example:

Consider our standard example:

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

A trace of the fixpoint algorithm then looks as follows:

solve x_2	eval x_2 x_3	solve x_3	eval x_3 x_1	solve x_1	eval x_1 x_3	solve x_3
						stable!
					$I[x_3] = \{x_1\}$	
					$\Rightarrow \quad \emptyset$	
				$D[x_1] = \{a\}$		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a\}$			
		$D[x_3] = \{a, c\}$				
		$I[x_3] = \emptyset$				
		solve x_1	eval x_1 x_3	solve x_3		
				stable!		
			$I[x_3] = \{x_1\}$			
			$\Rightarrow \quad \{a, c\}$			
		$D[x_1] = \{a, c\}$				
		$I[x_1] = \emptyset$				
		solve x_3	eval x_3 x_1	solve x_1		
				stable!		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a, c\}$			
		ok				
	$I[x_3] = \{x_1, x_2\}$					
	$\Rightarrow \quad \{a, c\}$					
$D[x_2] = \{a\}$						

→ Evaluation starts with an interesting unknown x_i (e.g., the value at *stop*)

→ Then automatically all unknowns are evaluated which influence x_i :-)

→ The number of evaluations is often smaller than during worklist iteration ;-)

→ The algorithm is more complex but does not rely on pre-computation of variable dependencies :-))

→ It also works if variable dependencies during iteration change !!!

$\Longrightarrow$ interprocedural analysis