Achtung:

- Den Wert Null haben wir nicht mit-modelliert.
  Dereferenzieren von Null kann darum nicht entdeckt werden :-(
- Destruktive Updates sind nur von Variablen möglich, nicht im Speicher!

  $\Longrightarrow$ keine Information, falls Speicher-Objekte nicht vorinitialisiert sind :-((
- Die Kanten-Effekte hängen jetzt von der ganzen Kante ab.

  Die Analyse lässt sich so nicht gegenüber der Referenz-Semantik als korrekt erweisen :-(

  Zur Korrektheit muss die konkrete Semantik mit zusätzlicher Information instrumentiert werden, die vermerkt, an welchem Programmpunkt eine Adresse erzeugt wurde.

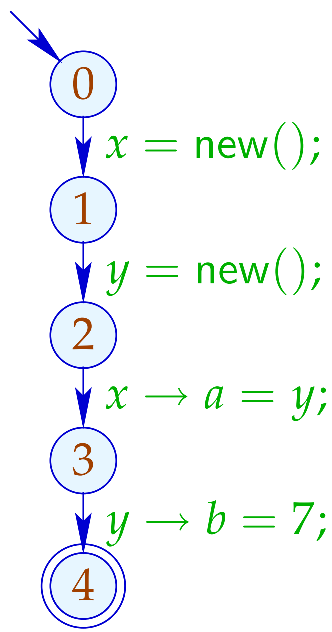
...

- Wir berechnen mögliche Points-to-Information.
- Daraus können wir May-Alias-Information gewinnen.
- Die Analyse kann jedoch ziemlich aufwendig sein (ohne viel raus zu kriegen :-(
- Separate Information für jeden Programmpunkt ist möglicherweise nicht nötig ??

Alias-Analyse 2. Idee:

Berechne für jede Variable und jede Adresse einen Wert, der die Werte an sämtlichen Programmpunkten sicher approximiert!

... im einfachen Beispiel:



x	$\{(0,1)\}$
y	$\{(1,2)\}$
$(0,1).a$	$\{(1,2)\}$
$(0,1).b$	$\emptyset$

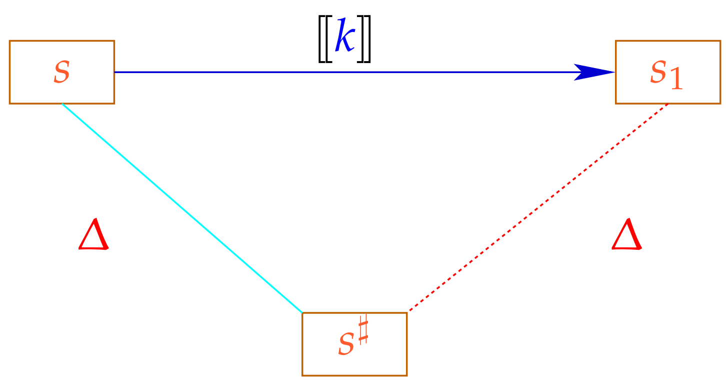
Jede Kante (u, lab, v) gibt Anlass zu Constraints:

lab	*Constraints*
$x = y;$	$\mathcal{P}[x] \supseteq \mathcal{P}[y]$
$x = \mathsf{new}();$	$\mathcal{P}[x] \supseteq \{(u, v)\}$
$x = R \to a;$	$\mathcal{P}[x] \supseteq \bigcup\{\mathcal{P}[f.a] \mid f \in \mathcal{P}[R]\}$
$R \to a = x;$	$\mathcal{P}[f.a] \supseteq (f \in \mathcal{P}[R])\,?\,\mathcal{P}[x] \,:\, \emptyset$ für alle $f \in Addr^\sharp$

Andere Kanten haben keinen Effekt :-)

Diskussion:

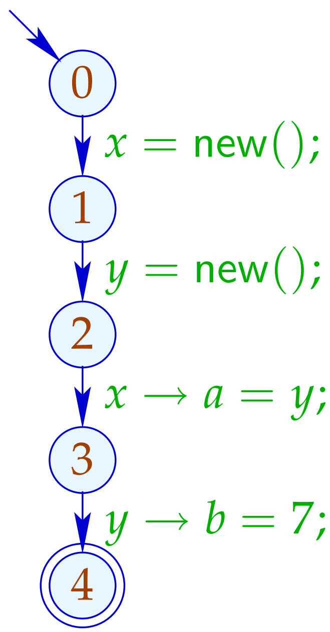
- Das resultierende Constraint-System ist $\mathcal{O}(k \cdot n)$ bei k abstrakten Adressen und n Kanten :-(
- Die Anzahl eventuell notwendiger Iterationen ist $\mathcal{O}(k)$ ...
- Die berechnete Information ist möglicherweise immer noch zu präzise !!?
- Zur Korrektheit einer Lösung $s^\sharp \in \mathit{States}^\sharp$ des Constraint-Systems zeigt man:



Alias-Analyse 3. Idee:

Berechne eine Äquivalenzrelation $\equiv$ auf Variablen x und Selektoren $y \to a$ mit $s_1 \equiv s_2$ falls an irgendeinem u s_1, s_2 die gleiche Adresse enthalten ...

... im einfachen Beispiel:



$$\equiv \; = \; \{\{x\}, \{y, x \to a\}, \{x \to b\}, \{y \to b\}\}$$

Diskussion:

- $\longrightarrow$ Wir berechnen eine Information für das ganze Programm.
- $\longrightarrow$ Die Berechnung dieser Information verwaltet Partitionen $\pi = \{P_1, \ldots, P_m\}$:-)
- $\longrightarrow$ Einzelne Mengen P_i identifizieren wir durch einen Repräsentanten $p_i \in P_i$.
- $\longrightarrow$ Die Operationen auf einer Partition π sind:

$$
\begin{array}{lcll}
\textsf{find}\,(\pi, p) & = & p_i & \text{falls } p \in P_i \\
 & // & \text{liefert den Repräsentanten} & \\
\textsf{union}\,(\pi, p_{i_1}, p_{i_2}) & = & \{P_{i_1} \cup P_{i_2}\} \cup \{P_j \mid i_1 \neq j \neq i_2\} & \\
 & // & \text{vereinigt repräsentierte Klassen} &
\end{array}
$$

$\longrightarrow$ Sind $x_1, x_2 \in Vars$ äquivalent, müssen auch $x_i \to a$ und $x_i \to b$ äquivalent sein :-)

$\longrightarrow$ Ist $P_i \cap Vars \neq \emptyset$, soll auch $p_i \in Vars$ gelten. Dann können wir union rekursiv anwenden :

$$
\begin{array}{lll}
\mathsf{union}^*\,(\pi, q_1, q_2) & = & \mathsf{let}\;\; p_{i_1} \;=\; \mathsf{find}\,(\pi, q_1) \\
& & \phantom{\mathsf{let}}\;\; p_{i_2} \;=\; \mathsf{find}\,(\pi, q_2) \\
& & \mathsf{in}\;\; \mathsf{if}\; p_{i_1} == p_{i_2}\; \mathsf{then}\; \pi \\
& & \quad\;\; \mathsf{else}\;\; \mathsf{let}\; \pi \;=\; \mathsf{union}\,(\pi, p_{i_1}, p_{i_2}) \\
& & \quad\;\; \phantom{\mathsf{else}}\;\; \mathsf{in}\;\; \mathsf{if}\; p_{i_1}, p_{i_2} \in Vars\; \mathsf{then} \\
& & \quad\;\; \phantom{\mathsf{else}\;\; \mathsf{in}}\;\; \mathsf{let}\; \pi = \mathsf{union}^*\,(\pi, p_{i_1} \to a, p_{i_2} \to a) \\
& & \quad\;\; \phantom{\mathsf{else}\;\; \mathsf{in}}\;\; \mathsf{in}\; \mathsf{union}^*\,(\pi, p_{i_1} \to b, p_{i_2} \to b) \\
& & \quad\;\; \phantom{\mathsf{else}}\;\; \mathsf{else}\; \pi
\end{array}
$$

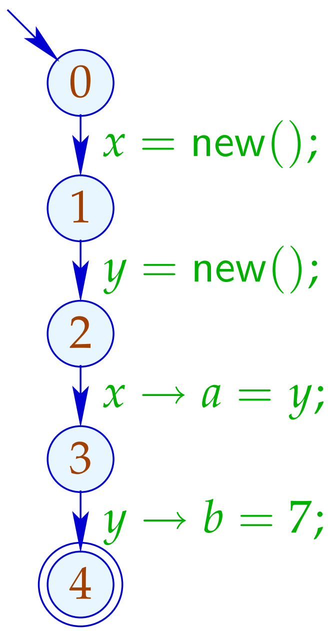
Die Analyse iteriert einmal über alle Kanten:

$$\pi = \{\{x\}, \{x \to a\}, \{x \to b\} \mid x \in Vars\};$$
$$\text{forall } k = (\_, lab, \_) \text{ do} \quad \pi = [\![lab]\!]^\sharp\, \pi;$$

Dabei ist:

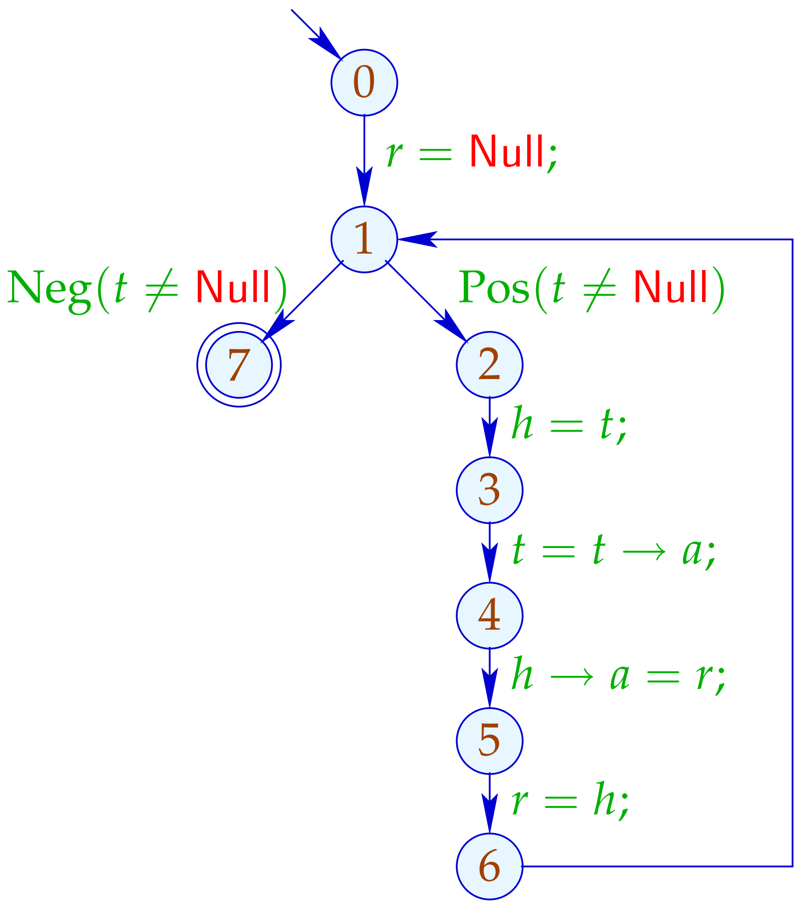
$$\begin{array}{lll}
[\![x = y;]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, y) \\
[\![x = R \to a;]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, R \to a) \\
[\![R \to a = x;]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, R \to a) \\
[\![lab]\!]^\sharp\, \pi & = & \pi \qquad\qquad \text{sonst}
\end{array}$$

... im einfachen Beispiel:



	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \ldots\}$
$(0,1)$	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \ldots\}$
$(1,2)$	$\{\{x\}, \{y\}, \{x \rightarrow a\}, \{y \rightarrow a\}, \ldots\}$
$(2,3)$	$\{\{x\}, \boxed{\{y, x \rightarrow a\}}, \{y \rightarrow a\}, \ldots\}$
$(3,4)$	$\{\{x\}, \{y, x \rightarrow a\}, \{y \rightarrow a\}, \ldots\}$

... im komplizierten Beispiel:



	$\{\{h\},\{r\},\{t\},\{h \to a\},\{t \to a\}\}$
$(2,3)$	$\{\boxed{\{h,t\}},\{r\},\boxed{\{h \to a, t \to a\}}\}$
$(3,4)$	$\{\boxed{\{h,t,h \to a, t \to a\}},\{r\}\}$
$(4,5)$	$\{\boxed{\{h,t,r,h \to a, t \to a\}}\}$
$(5,6)$	$\{\{h,t,r,h \to a, t \to a\}\}$

Achtung:

Um überhaupt etwas heraus zu kriegen, müssen wir annehmen, dass alle Variablen anfangs auf verschiedene Adressen zeigen.

Zur Komplexität:

Wir haben:

$\mathcal{O}(\#\,\mathit{Kanten})$	Aufrufe von	union*
$\mathcal{O}(\#\,\mathit{Kanten})$	Aufrufe von	find
$\mathcal{O}(\#\,\mathit{Vars})$	Aufrufe von	union

$\Longrightarrow$ Wir benötigen effiziente Union-Find-Datenstruktur :-)

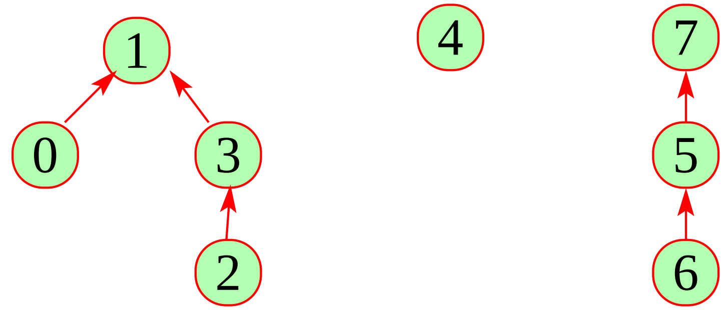
Idee:

Repräsentiere Partition von U als gerichteten Wald:

- Zu $u \in U$ verwalten wir einen Vater-Verweis $F[u]$.
- Elemente u mit $F[u] = u$ sind Wurzeln.

Einzelne Bäume sind Äquivalenzklassen.

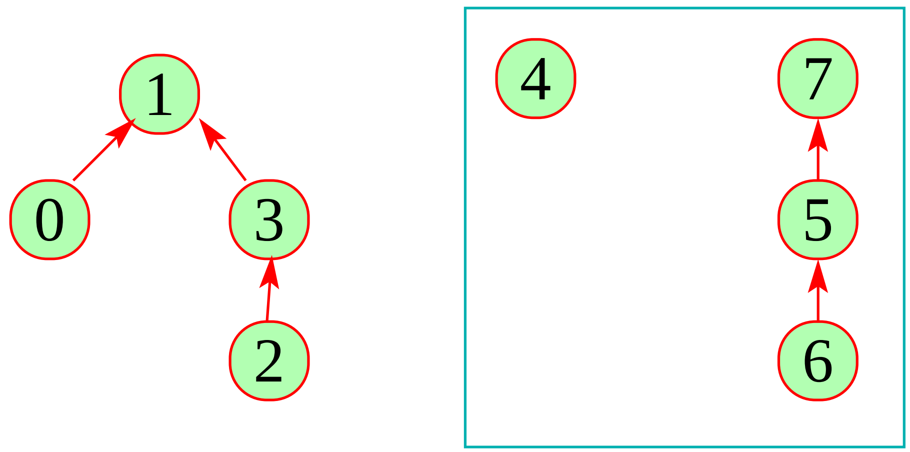
Ihre Wurzeln sind die Repräsentanten ...



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

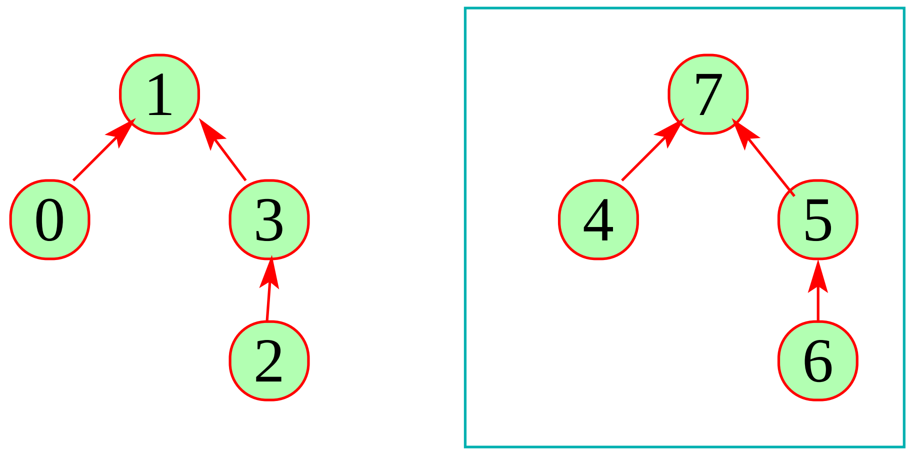
$\longrightarrow$ find (π, u) folgt den Vater-Verweisen :-)

$\longrightarrow$ union (π, u_1, u_2) hängt den Vater-Verweis eines u_i um ...



0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7



0	1	2	3	4	5	6	7

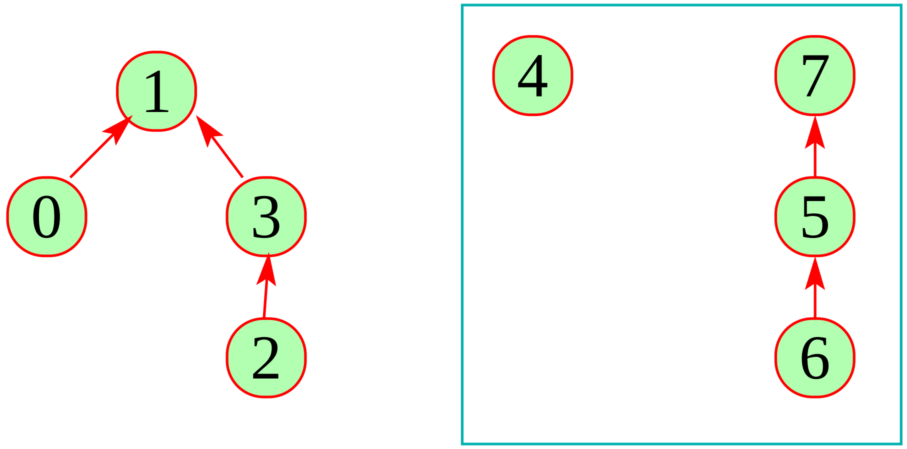
1	1	3	1	7	7	5	7

Die Kosten:

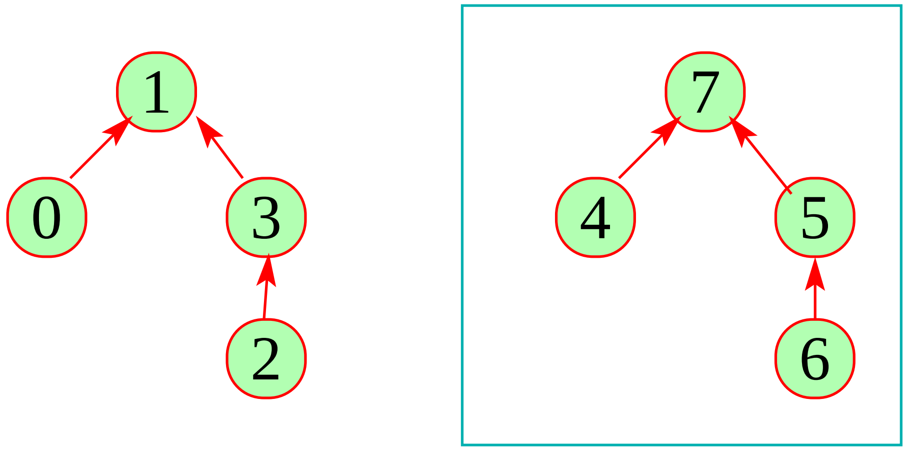
union	:	$\mathcal{O}(1)$	:-)
find	:	$\mathcal{O}(depth(\pi))$	:-(

Strategie zur Vermeidung tiefer Bäume:

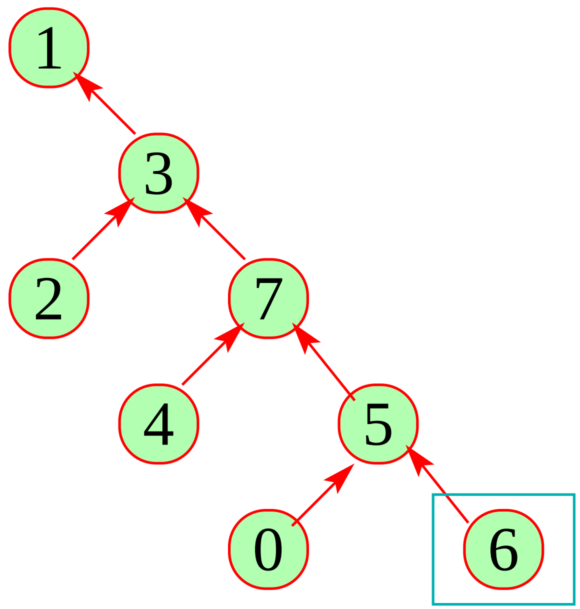
- Hänge den kleineren Baum unter den größeren !
- Benutze find , um Pfade zu komprimieren ...



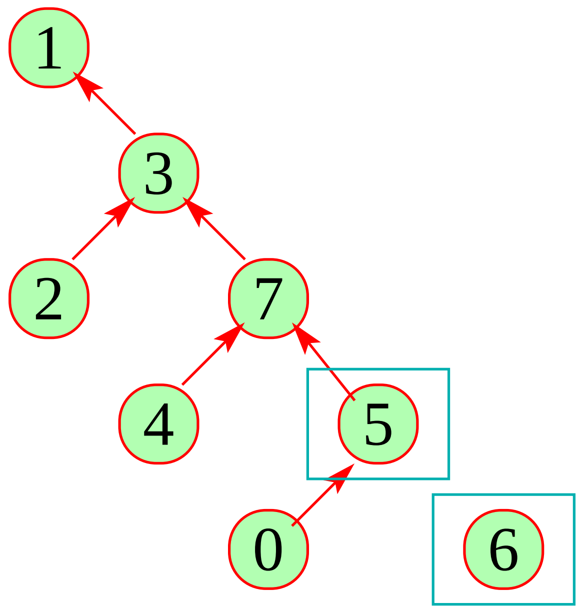
0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7



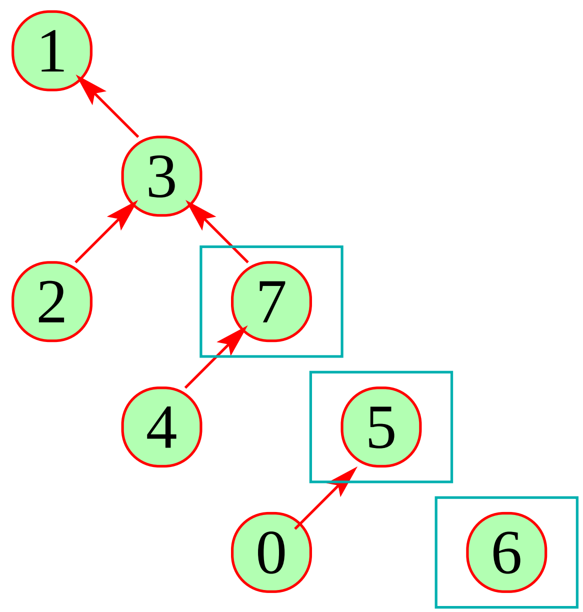
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7



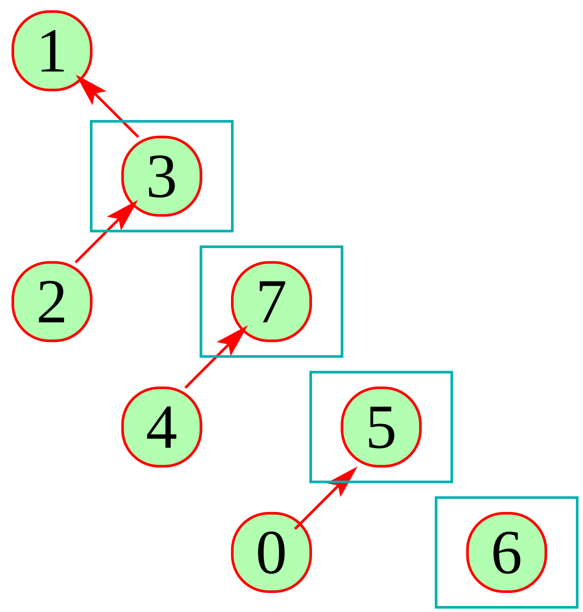
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



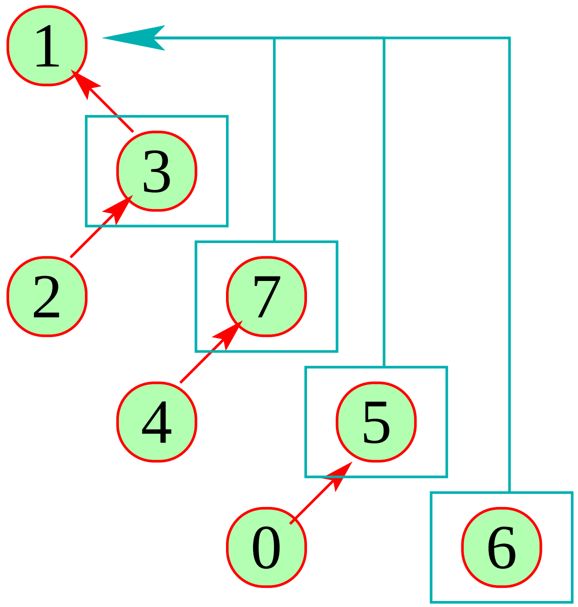
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



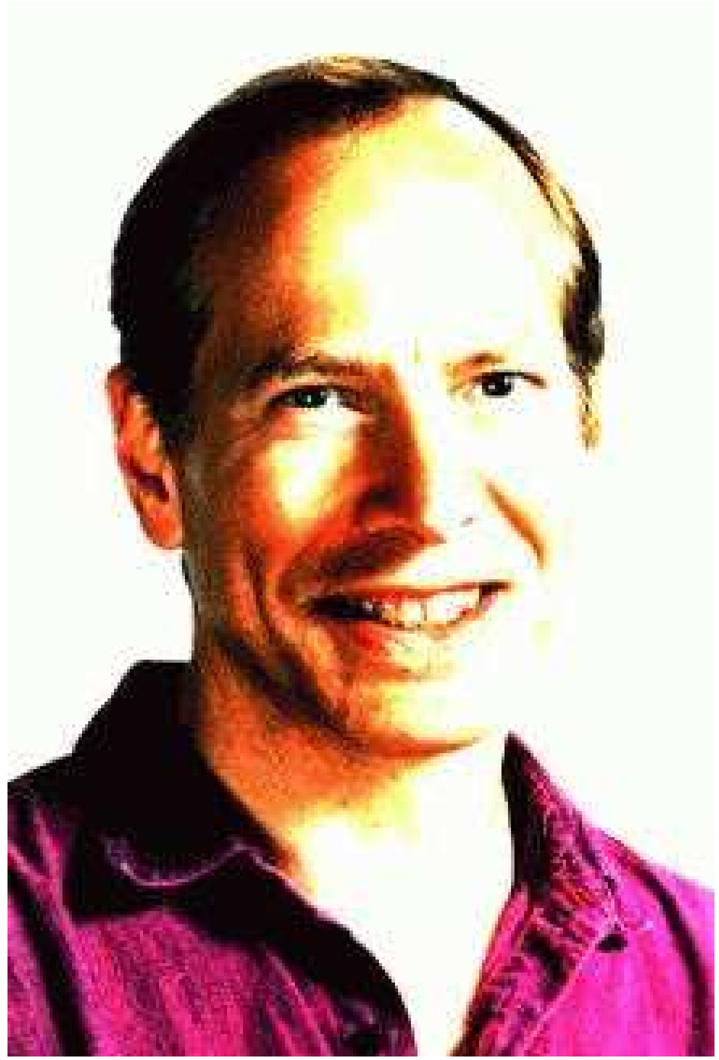
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



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

Beachte:

- Mit dieser Datenstruktur dauern n union- und m find-Operationen $\mathcal{O}(n + m \cdot \alpha(n, n))$

  // α die inverse Ackermann-Funktion :-)

- Für unsere Anwendung müssen wir union nur so modifizieren, dass an den Wurzeln nach Möglichkeit Elemente aus *Vars* stehen.

- Diese Modifikation vergrößert die asymptotische Laufzeit nicht :-)

Fazit:

Die Analyse ist blitzschnell — findet aber nicht sehr viel heraus.

Exkurs 3: Fixpunkt-Algorithmen

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

Beobachtung:

RR-Iteration ist ineffizient:

→ Wir benötigen eine ganze Runde, um Terminierung festzustellen :-(

→ Ändert sich in einer Runde der Wert nur einer Variable, berechnen wir trotzdem alle neu :-(

→ Die praktische Laufzeit hängt von der Reihenfolge der Variablen ab :-(

Idee: **Workset-Iteration**

Ändert eine Variable x_i ihren Wert, werten wir alle Variablen neu aus, die von x_i abhängen. Technisch benötigen wir:

→ die Mengen $\mathit{Dep}\, f_i$ der Variablen, auf die die Auswertung von f_i zugreift. Daraus berechnen wir:

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

d.h. die Menge der x_j, die von x_i abhängen.

→ die Werte $D[x_i]$ der x_i, wobei anfangs $D[x_i] = \bot$;

→ Eine Menge W der Variablen, deren Wert neu berechnet werden muss ...