Union-Find-Algorithm

union-find: definitions

spec: Elements of sets chosen from $[0:N-1]$
algorithm maintains

- $Q \subseteq [1:N]$ set of all elements chosen so far. Initially $Q = \emptyset$.
- system (partition) P of Q

$$P = \{S_1, \dots, S_k\} \quad , \quad i \neq j \rightarrow S_i \cap S_j = \emptyset$$

$$\bigcup_{i=1}^{k} S_i = Q$$

For elements $x \in Q$ define $S(x)$ as the set $S \in P$ with $x \in S$

- representatives $r_i \in S_i$ serves as names of sets S_i. System of representatives:

$$R = \{r_1, \dots, r_k\}$$

union-find: definitions

spec: Elements of sets chosen from $[0:N-1]$

algorithm maintains

- $Q \subseteq [1:N]$ set of all elements chosen so far. Initially $Q = \emptyset$.
- system (partition) P of Q

$$P = \{S_1, \ldots, S_k\} \quad , \quad i \neq j \rightarrow S_i \cap S_j = \emptyset$$

$$\bigcup_{i=1}^{k} S_i = Q$$

For elements $x \in Q$ define $S(x)$ as the set $S \in P$ with $x \in S$

- representatives $r_i \in S_i$ serves as names of sets S_i. System of representatives:

$$R = \{r_1, \ldots, r_k\}$$

operations

- $make-set(x)$ with $x \notin Q$

$$Q' = Q \cup \{x\} \ , \ P' = P \cup \{\{x\}\}$$

- $union(x,y)$ with $x, y \in Q$

$$P' = P \setminus S(x) \setminus S(y) \cup \{S(x) \cup S(y)\}$$

- $find(x) = r$ with $x \in S(r)$ and $r \in R$

union-find: definitions

spec: Elements of sets chosen from $[0:N-1]$
algorithm maintains

- $Q \subseteq [1:N]$ set of all elements chosen so far. Initially $Q = \emptyset$.
- system (partition) P of Q

$$P = \{S_1, \ldots, S_k\} \quad , \quad i \neq j \rightarrow S_i \cap S_j = \emptyset$$

$$\bigcup_{i=1}^{k} S_i = Q$$

For elements $x \in Q$ define $S(x)$ as the set $S \in P$ with $x \in S$

- representatives $r_i \in S_i$ serves as names of sets S_i. System of representatives:

$$R = \{r_1, \ldots, r_k\}$$

operations

- $make-set(x)$ with $x \notin Q$

$$Q' = Q \cup \{x\} \ , \ P' = P \cup \{\{x\}\}$$

- $union(x,y)$ with $x, y \in Q$

$$P' = P \setminus S(x) \setminus S(y) \cup \{S(x) \cup S(y)\}$$

- $find(x) = r$ with $x \in S(r)$ and $r \in R$

problem size

- n: number of make-set operations; number of elements in all sets
- m number of union and find operations

$$n \leq m$$

disjoint set forest

one tree per set

class TE for tree elements. Nodes x are objects with components

- $x.p$: parent
- $x.r$: rank

notation:

$$
\begin{aligned}
p(x) &= x.p \text{ (parent)} \\
r(x) &= x.r \text{ (rank)}
\end{aligned}
$$

disjoint set forest

one tree per set

class TE for tree elements. Nodes x are objects with components

- $x.p$: parent
- $x.r$: rank

notation:

$$\begin{aligned} p(x) &= x.p \text{ (parent)} \\ r(x) &= x.r \text{ (rank)} \end{aligned}$$

convention:

- x is root of its tree iff $p(x) = x$, i.e. x points to itself
- roots serve as representatives

basic operations on trees

one tree per set

class TE for tree elements. Nodes x are objects with components

- $x.p$: parent
- $x.r$: rank

notation:

$$\begin{aligned} p(x) &= x.p \text{ (parent)} \\ r(x) &= x.r \text{ (rank)} \end{aligned}$$

convention:

- x is root of its tree iff $p(x) = x$, i.e. x points to itself
- roots serve as representatives

- test if $p(x) = x$: i.e. x points to itself

```
p(x)=x?: /x is representative, root */
x.p==x
```

basic operations on trees

one tree per set

class TE for tree elements. Nodes x are objects with components

- $x.p$: parent
- $x.r$: rank

notation:

$$\begin{aligned} p(x) &= x.p \text{ (parent)} \\ r(x) &= x.r \text{ (rank)} \end{aligned}$$

convention:

- x is root of its tree iff $p(x) = x$, i.e. x points to itself
- roots serve as representatives

- test if $p(x) = x$: i.e. x points to itself

```
p(x)=x?: /x is representative, root */
x.p==x
```

- make $y \in Q$ the parent of $x \in Q$:

```
p(x):=y :/*make y parent of x*/
x.p= y
```

basic operations on trees

one tree per set

class TE for tree elements. Nodes x are objects with components

- $x.p$: parent
- $x.r$: rank

notation:

$$\begin{aligned} p(x) &= x.p \text{ (parent)} \\ r(x) &= x.r \text{ (rank)} \end{aligned}$$

convention:

- x is root of its tree iff $p(x) = x$, i.e. x points to itself
- roots serve as representatives

- test if $p(x) = x$: i.e. x points to itself

```
p(x)=x?: /x is representative, root */
x.p==x
```

- make $y \in Q$ the parent of $x \in Q$:

```
p(x):=y :/*make y parent of x*/
x.p= y
```

- replace x by parent $p(x)$ of x

```
x:=p(x): /* replace x by p(x) */
x=x.p
```

basic operations on trees

one tree per set

class TE for tree elements. Nodes x are objects with components

- $x.p$: parent
- $x.r$: rank

notation:

$$\begin{aligned} p(x) &= x.p \text{ (parent)} \\ r(x) &= x.r \text{ (rank)} \end{aligned}$$

convention:

- x is root of its tree iff $p(x) = x$, i.e. x points to itself
- roots serve as representatives

- test if $p(x) = x$: i.e. x points to itself

```
p(x)=x?: /x is representative, root */
x.p==x
```

- make $y \in Q$ the parent of $x \in Q$:

```
p(x):=y :/*make y parent of x*/
x.p= y
```

- replace x by parent $p(x)$ of x

```
x:=p(x): /* replace x by p(x) */
x=x.p
```

- assign expression f to rank $r(x)$

```
r(x):=f /*assign f to rank of x*/
x.r= f
```

implementation of operations with balancing

using basic operations

implementation of operations with balancing

```
make-set(x):
x= new TE;
p(x):=x; r(x):=0
```

implementation of operations with balancing

using basic operations

implementation of operations with balancing

```
make-set(x):
x= new TE;
p(x):=x; r(x):=0
```

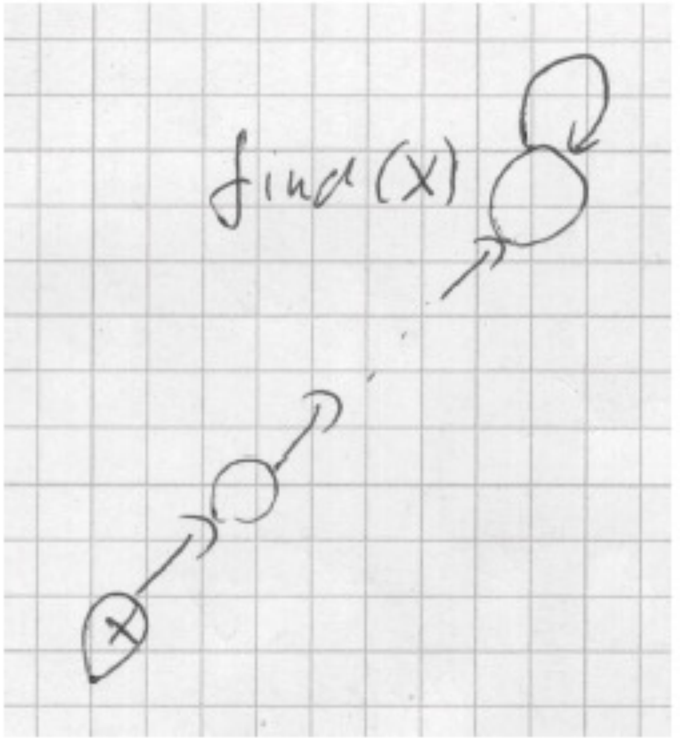

Figure 1: The root $find(x)$ of the tree containing x is found by chasing of parent-pointers

.

```
find(x):
while p(x) !=x {x:=p(x)};
return x
```

implementation of operations with balancing

using basic operations

implementation of operations with balancing

```
make-set(x):
x= new TE;
p(x):=x; r(x):=0
```

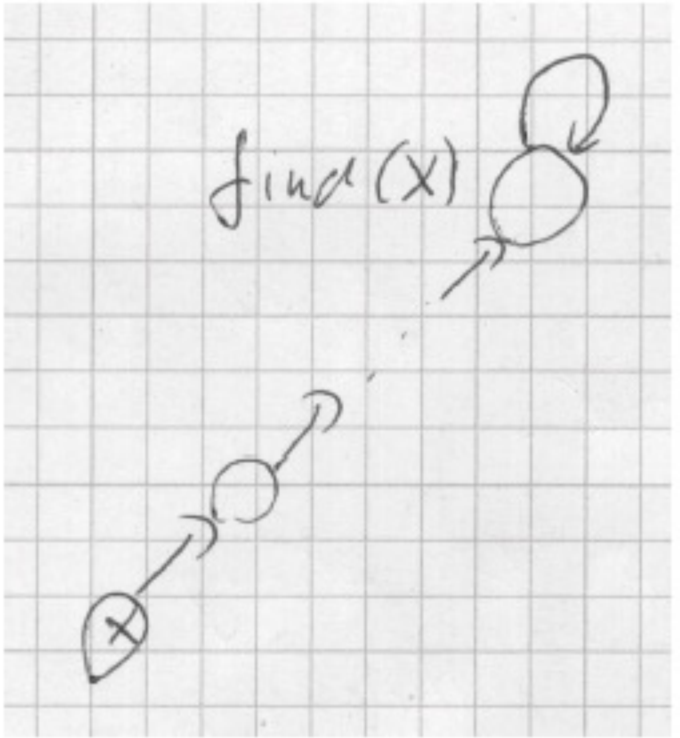

Figure 1: The root $find(x)$ of the tree containing x is found by chasing of parent-pointers

.

```
find(x):
while p(x) !=x {x:=p(x)};
return x
```

```
union(x,y): link(find(x), find(y))
```

linking trees with roots x and y

```
link(x,y):
if r(x)<r(y) {p(x):=y} /*make y predecessor of x*/;
if r(x)>r(y) {p(y):=x} /*make x predecessor of y*/;
if r(x)=r(y) {p(x):=y; r(y)=r(y)+1} /*increase rank of y*/
```

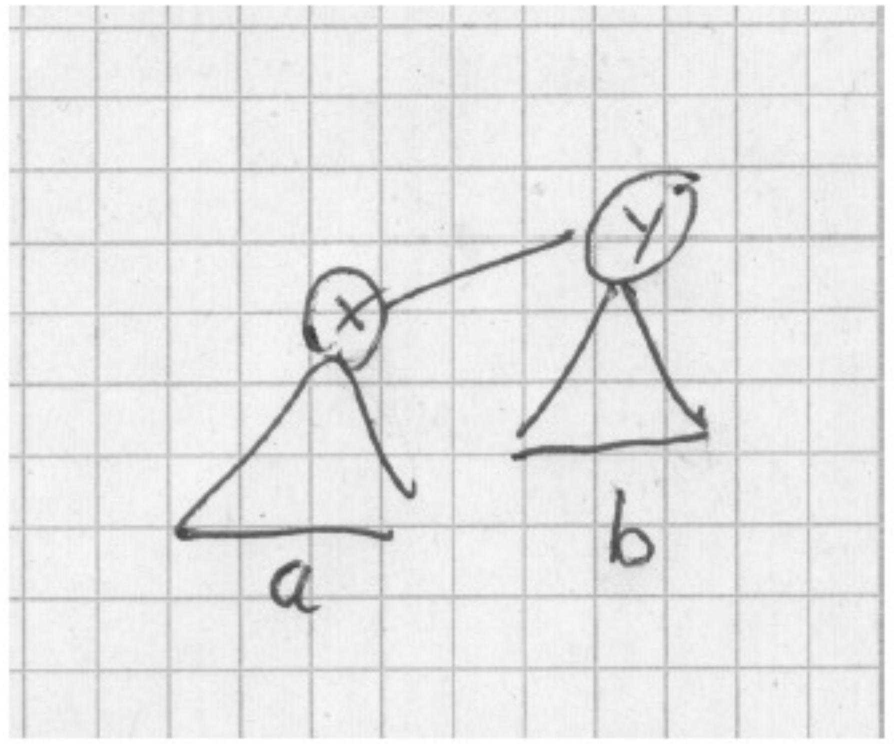

Figure 2: $link(x,y)$: if $r(x) \leq r(y)$ one makes x a son of y. If $r(x) = r(y)$ one increases $r(y)$ to $r'(y) = r(y) + 1$.

.

analysis of simple algorithm

Lemma 1. *if x is root of tree with n nodes, then*

$$r(x) \leq \lfloor \log n \rfloor \leq n - 1$$

Induction on n

$n = 0$ and right inequality: trivial

analysis of simple algorithm

Lemma 1. *if x is root of tree with n nodes, then*

$$r(x) \leq \lfloor \log n \rfloor \leq n-1$$

Induction on n

$n = 0$ and right inequality: trivial

$n-1 \rightarrow n$. Let a, b be number of nodes in trees with roots x, y

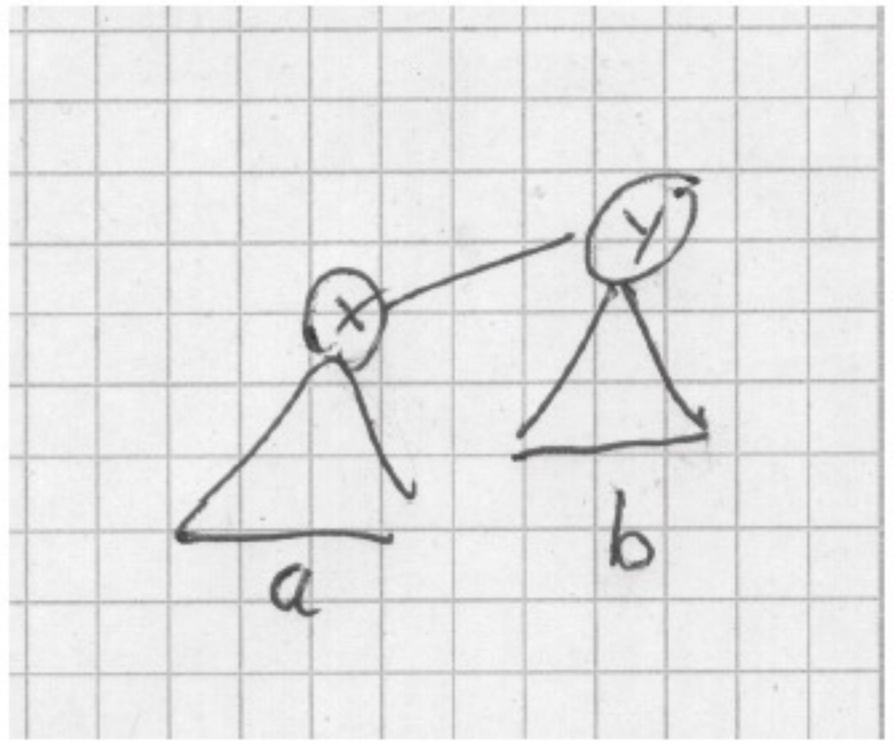

Figure 5: $link(x,y)$: if $r(x) \leq r(y)$ one makes x a son of y. The new tree has $a+b$ nodes.

.

- $r(x) < r(y)$

$$r(y) \leq \lfloor \log a \rfloor \leq \lfloor \log(a+b) \rfloor$$

analysis of simple algorithm

Lemma 1. *if x is root of tree with n nodes, then*

$$r(x) \leq \lfloor \log n \rfloor \leq n-1$$

Induction on n

$n = 0$ and right inequality: trivial

$n-1 \rightarrow n$. Let a, b be number of nodes in trees with roots x, y

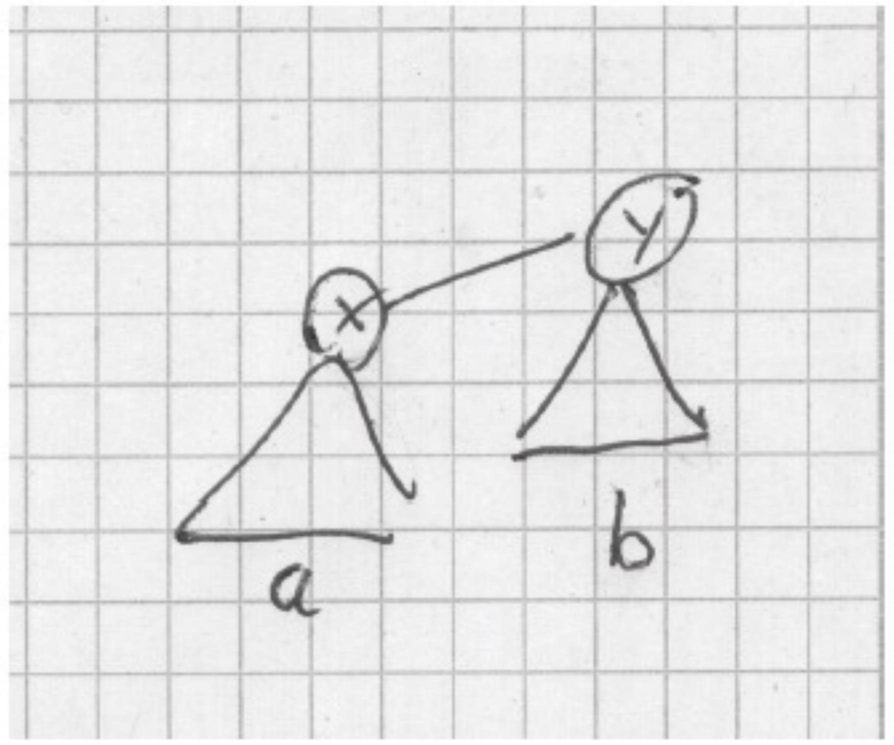

Figure 5: $link(x,y)$: if $r(x) \leq r(y)$ one makes x a son of y. The new tree has $a+b$ nodes.

- $r(x) < r(y)$

$$r(y) \leq \lfloor \log a \rfloor \leq \lfloor \log(a+b) \rfloor$$

- $r(x) > r(y)$ similar
- $r(x) = r(y)$

$$\begin{aligned} r'(y) &= r(y)+1 \\ &= r(x)+1 \\ &\leq \lfloor \log\ \min\{a,b\} \rfloor + 1 \\ &\leq \lfloor \log \frac{a+b}{2} \rfloor + 1 \\ &= \lfloor \log(a+b) - 1 \rfloor + 1 \\ &= \lfloor \log(a+b) \rfloor \end{aligned}$$

analysis of simple algorithm

Lemma 2.

$$r(x) \geq h(x)$$

Induction on number of nodes in tree with root x

$n = 1$ trivial

analysis of simple algorithm

Lemma 2.

$$r(x) \geq h(x)$$

Induction on number of nodes in tree with root x

$n = 1$ trivial

$n - 1 \to n$

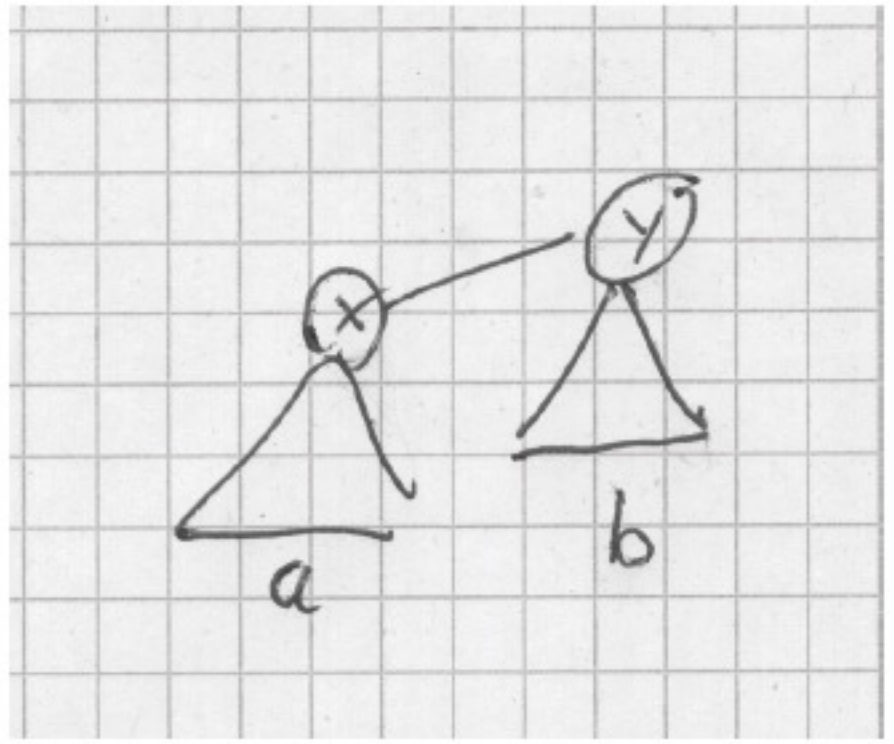

Figure 6: $link(x, y)$: if $r(x) \leq r(y)$ one makes x a son of y. The new tree has height $h'(y) = \max\{h(x) + 1, h(y)\}$.

- $r(x) < r(y)$

$$h'(y) = \max\{h(y), h(x) + 1\}$$

$$r'(y) = r(y) \geq r(x) + 1 \geq h(x) + 1$$

analysis of simple algorithm

Lemma 2.

$$r(x) \geq h(x)$$

Induction on number of nodes in tree with root x

$n = 1$ trivial

$n-1 \to n$

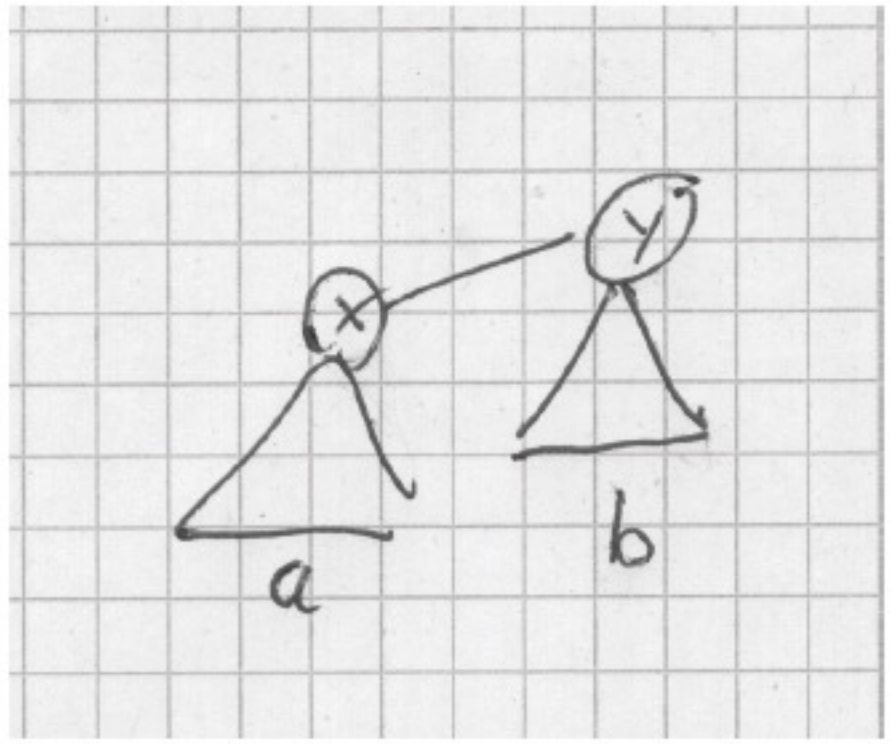

Figure 6: $link(x,y)$: if $r(x) \leq r(y)$ one makes x a son of y. The new tree has height $h'(y) = \max\{h(x)+1, h(y)\}$.

- $r(x) < r(y)$

$$h'(y) = \max\{h(y), h(x)+1\}$$

$$r'(y) = r(y) \geq r(x)+1 \geq h(x)+1$$

- $r(x) > r(y)$ similar
- $r(x) = r(y)$

$$h'(y) = \max\{h(y), h(x)+1\}$$

$$h'(y) = h(y) \to h'(y) \leq r(y) < r(y)+1 = r'(y)$$

$$h'(y) = h(x)+1 \to h'(y) = h(x)+1 \leq r(x)+1 = r(y)+1 = r'(y)$$

run time

time for operations

- make-set: $O(1)$
- union: $O(1)$
- find:

$$\begin{aligned} O(h(find(x)) &= O(r(find(x)) \quad \text{(lemma 2)} \\ &= O(logn) \quad \text{(lemma 1)} \end{aligned}$$

$\boxed{\text{total runtime: } O(n+m\log n)}$

run time

time for operations

- make-set: $O(1)$
- union: $O(1)$
- find:

$$
\begin{aligned}
O(h(find(x)) &= O(r(find(x)) \quad \text{(lemma 2)} \\
&= O(logn) \quad \text{(lemma 1)}
\end{aligned}
$$

$\boxed{\text{total runtime: } O(n+m\log n)}$

improved by path compression

- programming: utterly simple
- run time: THE most famous analysis of an algorithm (Tarjan 1975)

find with path compression

spec:

input x at depth t.

$$x = x_t \quad , \quad i \geq 0 \rightarrow p(x_i) = x_{i-1} \quad , \quad p(x_0) = x_0$$

output: x_0

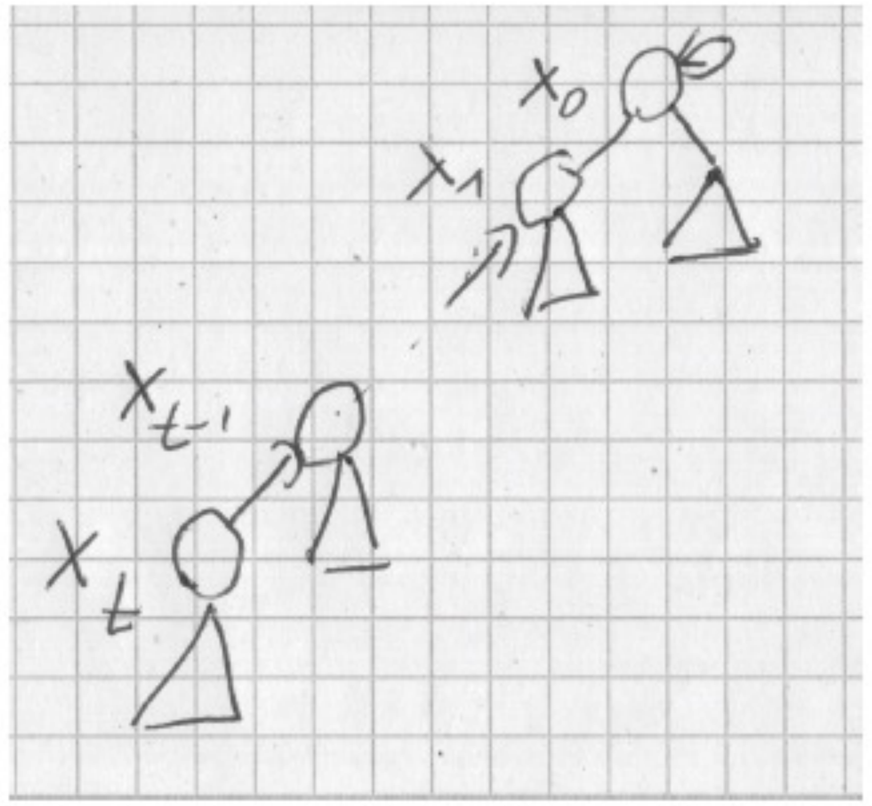

Figure 7: parent chasing from $x = x_t$ touches elements $x_{t-1}, \ldots, x_0$

find with path compression

spec:

input x at depth t.

$$x = x_t \quad , \quad i \geq 0 \to p(x_i) = x_{i-1} \quad , \quad p(x_0) = x_0$$

output: x_0

side effect:

$$p'(x_i) = x_0 \text{ for } i \in [1:t]$$

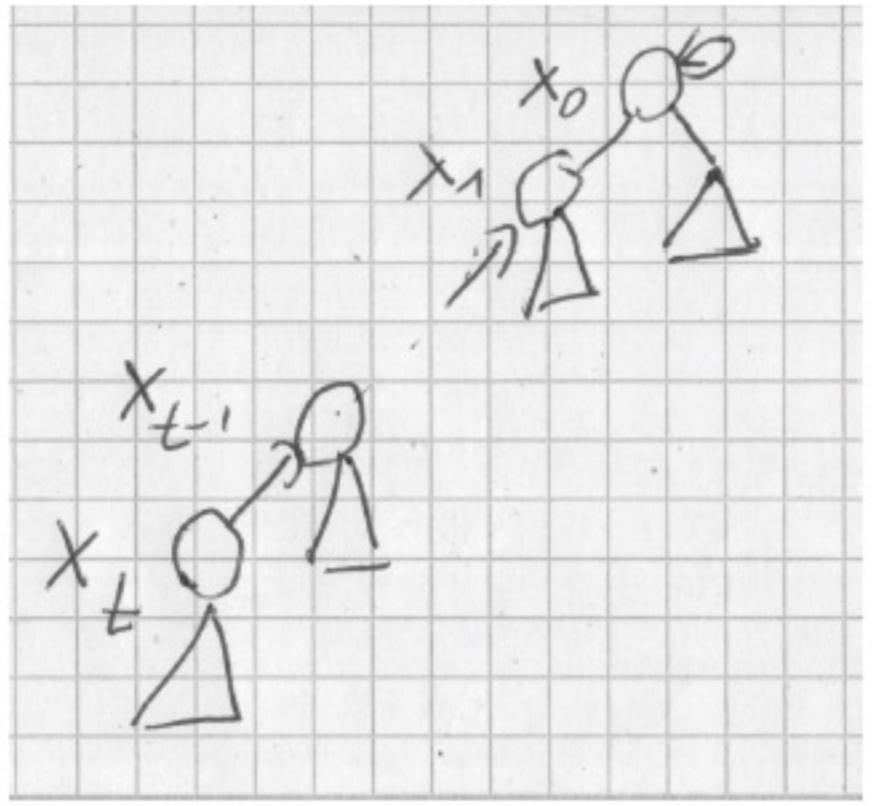

Figure 7: parent chasing from $x = x_t$ touches elements $x_{t-1}, \ldots, x_0$

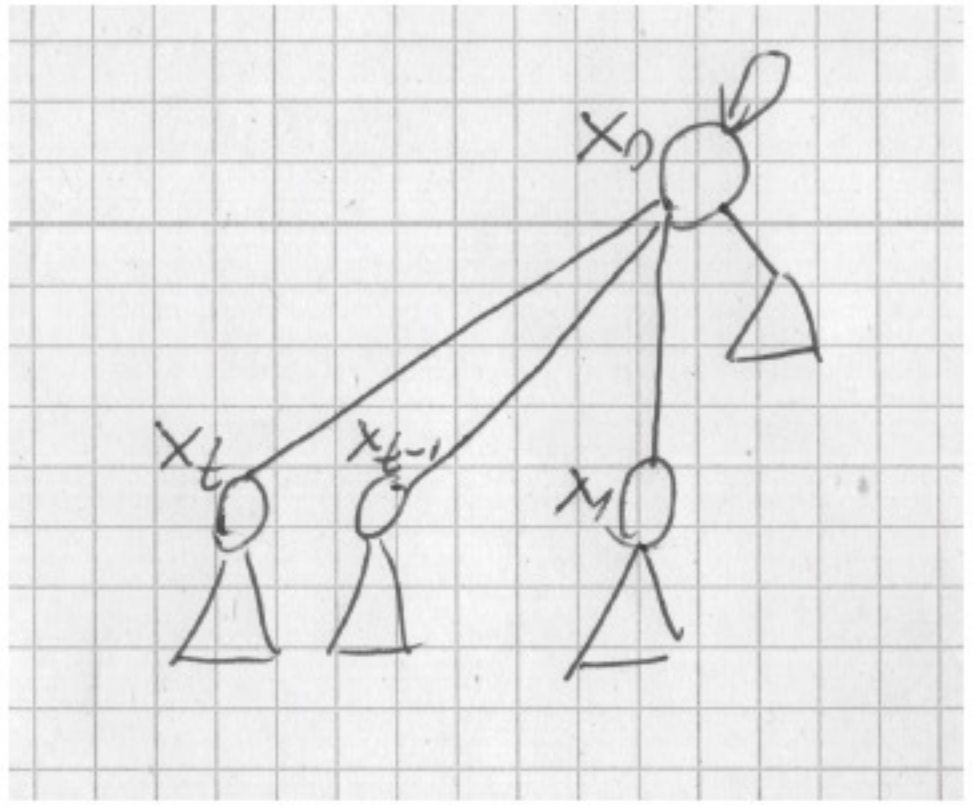

Figure 8: after path compression all nodes $x_t, \ldots, x_1$ are sons of the root x_0

find with path compression

implementation:

```
find(x):if x != p(x)
{p(x):= find(p(x)} /*recursive call with side effect*/
return p(x)
```

correctness:

$t = 0$ trivial

find with path compression

implementation:

```
find(x):if x != p(x)
{p(x):= find(p(x)} /*recursive call with side effect*/
return p(x)
```

correctness:

$t = 0$ trivial

$t \to t+1$

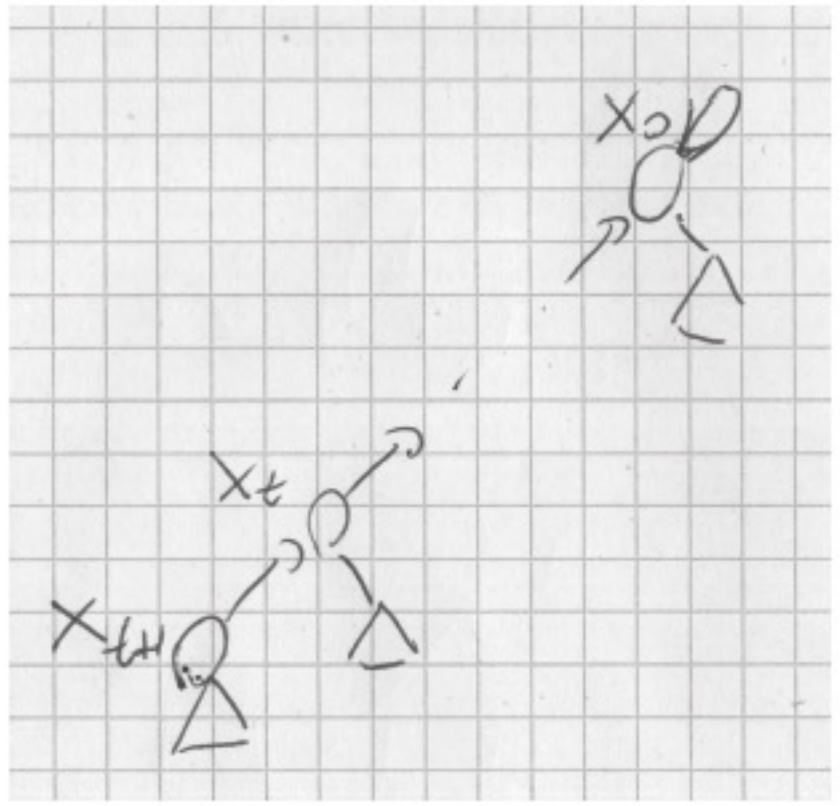

Figure 9: The path from x_{t+1} to x_0 needs to be compressed

find with path compression

implementation:

```
find(x):if x != p(x)
{p(x):= find(p(x)} /*recursive call with side effect*/
return p(x)
```

correctness:

$t = 0$ trivial

$t \to t+1$

$$p(x_{t+1}) = x_t$$

$find(x_t)$ called.

side effect:

$$p'(x_i) = x_0 \text{ for } i \in [1:t]$$

returns $find(x_t) = x_0$

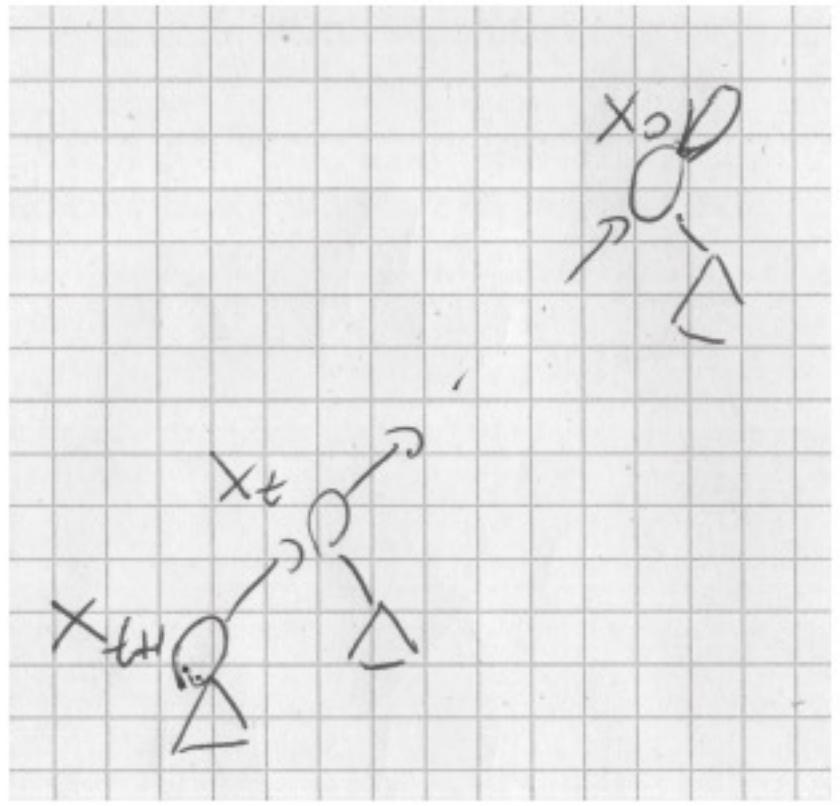

Figure 9: The path from x_{t+1} to x_0 needs to be compressed

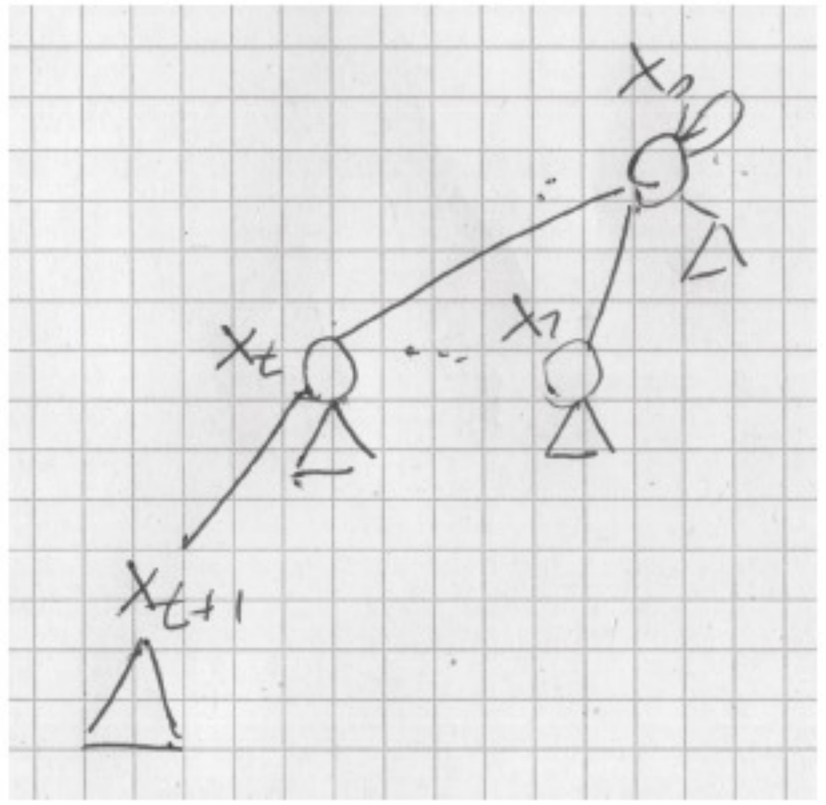

Figure 10: The path from x_t to x_0 is compressed by the recursive call $find(x_t)$

find with path compression

implementation:

```
find(x):if x != p(x)
{p(x):= find(p(x)} /*recursive call with side effect*/
return p(x)
```

correctness:

$t = 0$ trivial

$t \rightarrow t+1$

$$p(x_{t+1}) = x_t$$

$find(x_t)$ called.

side effect:

$$p'(x_i) = x_0 \text{ for } i \in [1:t]$$

returns $find(x_t) = x_0$

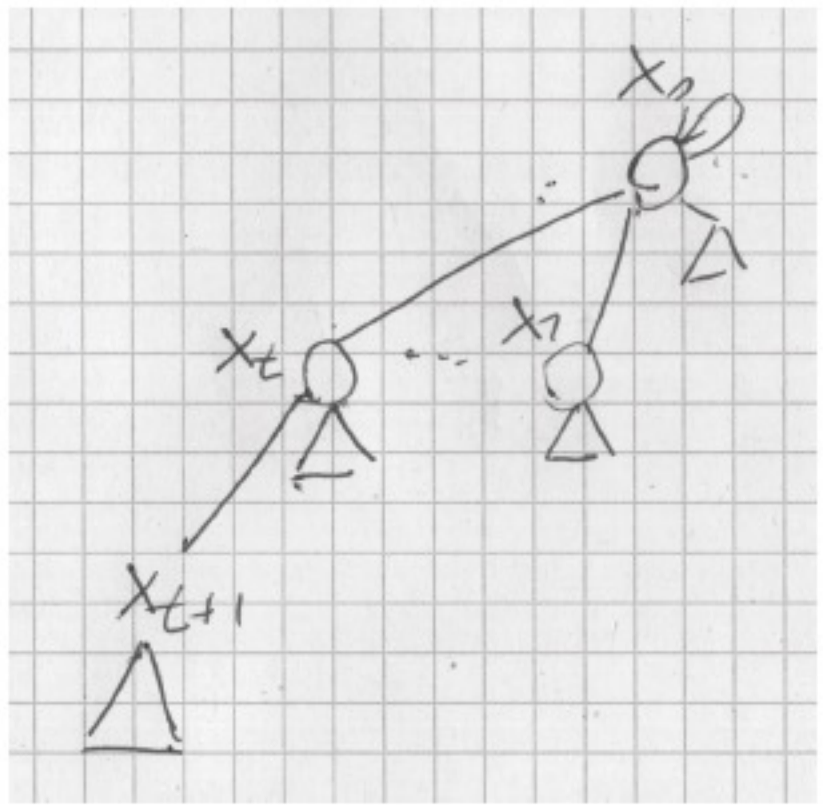

Figure 10: The path from x_t to x_0 is compressed by the recursive call $find(x_t)$

$p(x_{t+1}) := x_0$ executed

side effect:

$$p''(x_i) = x_0 \text{ for } i \in [1:t+1]$$

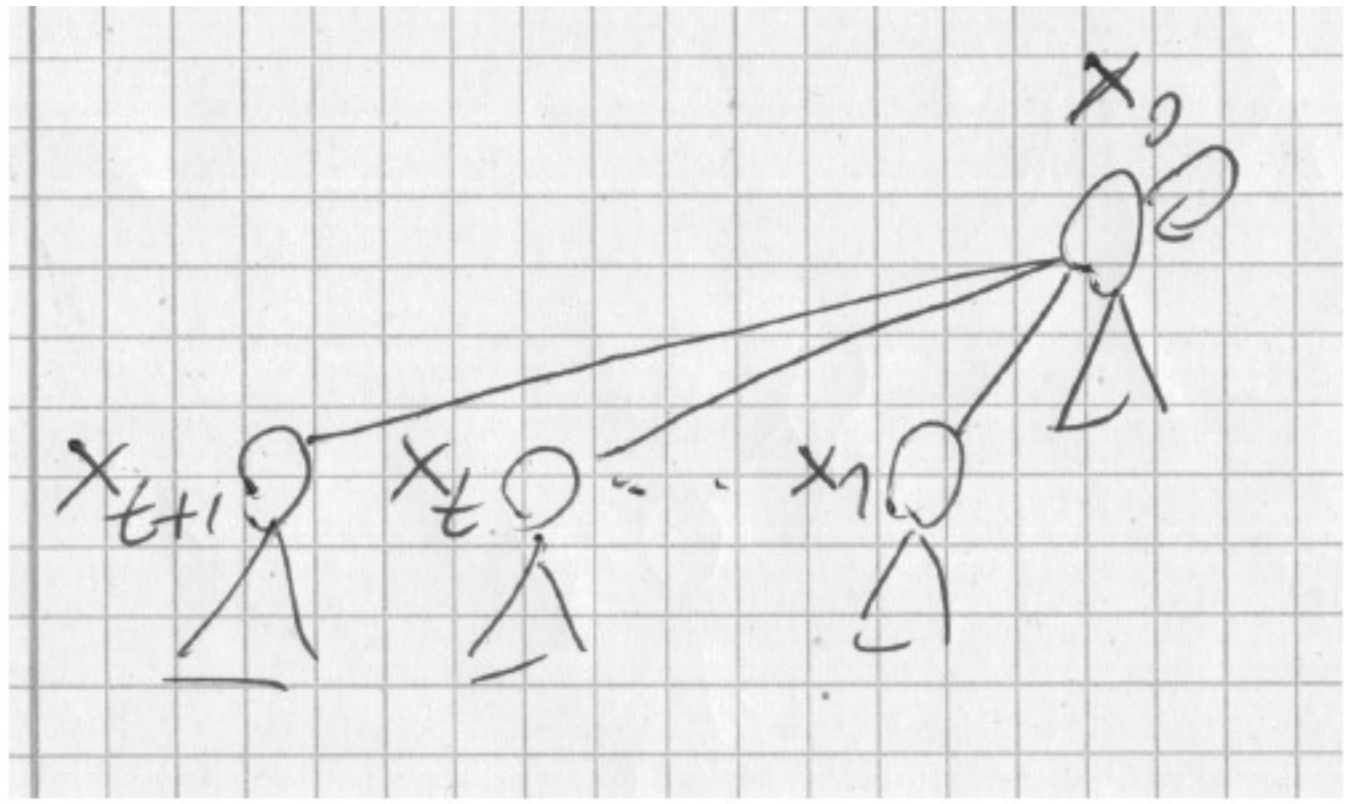

Figure 11: assignment $p(x_{t+1}) := x_0$ has completed the path compression