Algorithm Theory

09 – Union-Find Data Structures

Dr. Alexander Souza

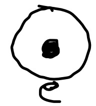
Winter term 11/12

Union-find data structures

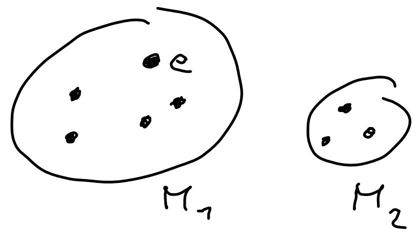
Problem:

Maintain a collection of disjoint sets while supporting the following operations:

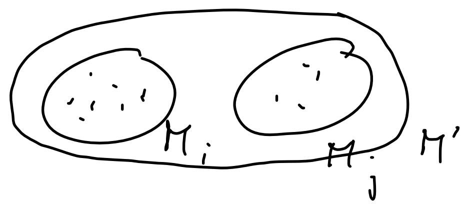
e.make-set(): Creates a new set whose only member is *e*.



e.find-set(): Returns the set M_i containing *e*.



union(M_i, M_j): Unites the sets M_i and M_j into a new set.



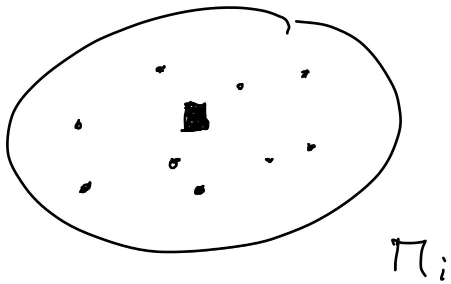
Union-find data structures

Printing the entire set M_i at a find-set operation is too costly

Instead

Representation of set M_i :

M_i is identified by a **representative**, which is some member of M_i.

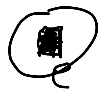


Union-find data structures

Operations using representatives:

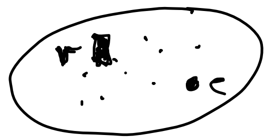
e.make-set():

Creates a new set whose only member is *e*. The representative is *e*.



e.find-set():

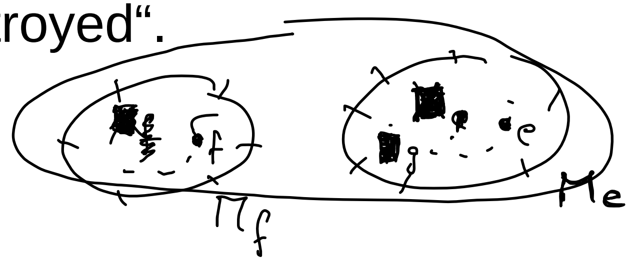
Returns the name of the representative of the set containing *e*.



e.union(f):

Unites the sets M_e and M_f that contain *e* and *f* into a new set *M* and returns a member of $M_e \cup M_f$ as the new representative of *M*.

The sets M_e and M_f are then „destroyed“.



Observations

- If *n* is the number of *make-set* operations and *m* the total number of *make-set*, *find-set* and *union* operations, then
  - $m >= n$
  - after $(n - 1)$ *union* operations, only one set remains in the collection

Application: Connected components

Input: graph $G = (V,E)$

Output: collection of the connected components of G

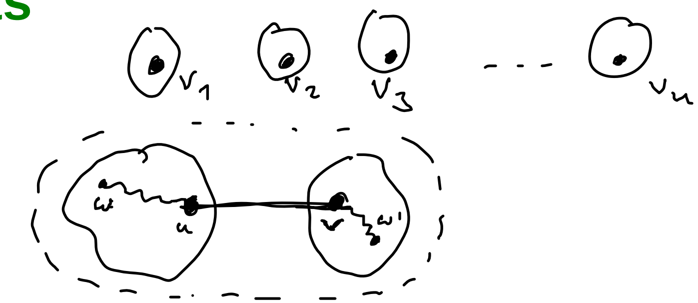
Algorithm: Connected-Components

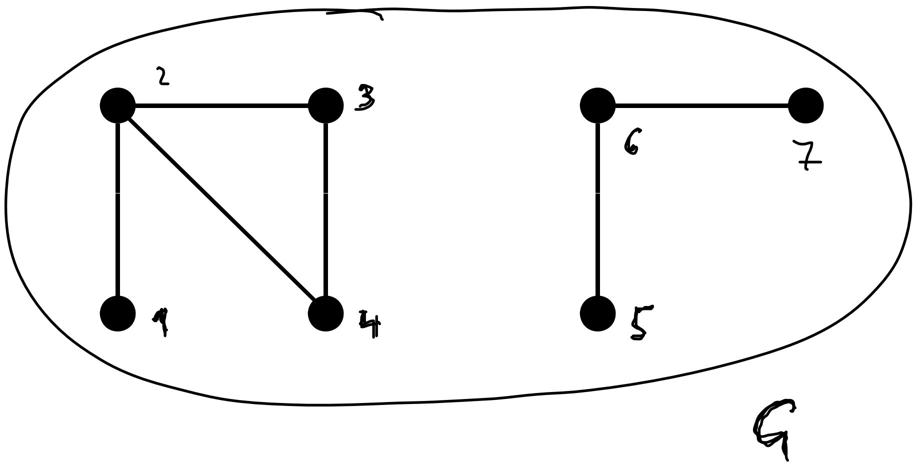
for all v **in** V **do** v.*make-set()*

for all (u,v) **in** E **do**

  if u.*find-set()* $\neq$ v.*find-set()*

    then u.*union*(v)





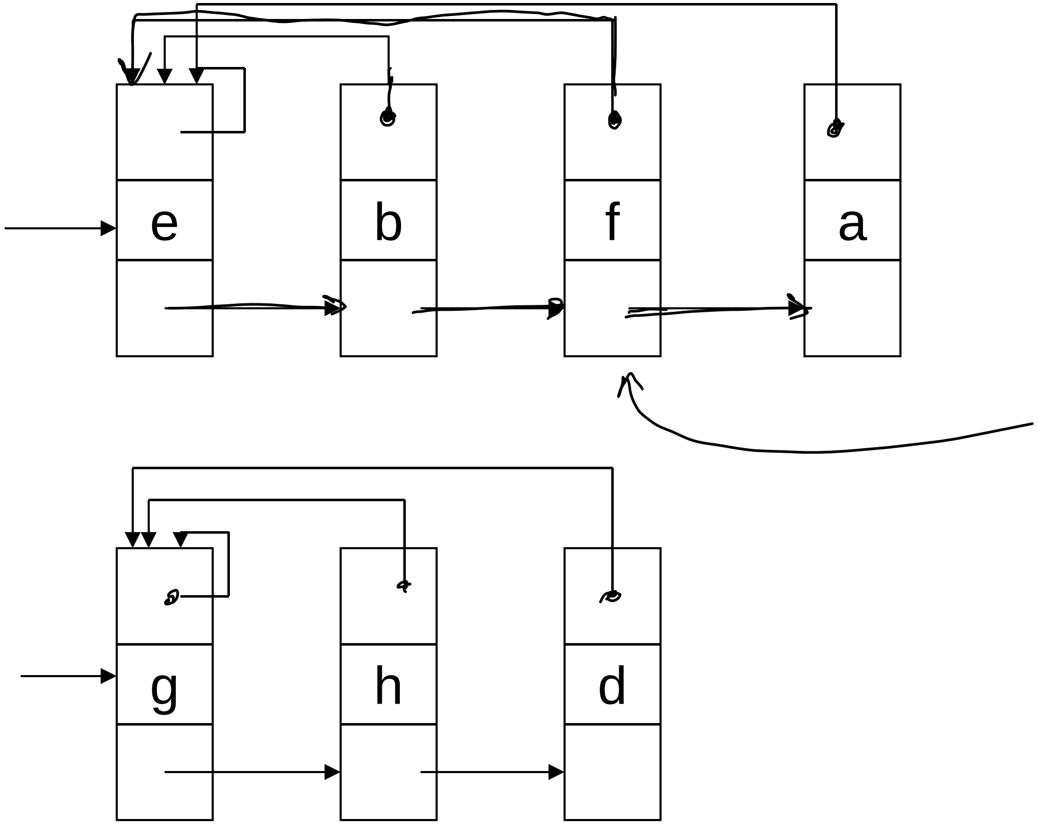
Same-Component (u,v):

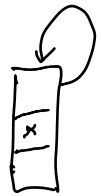
  if u.*find-set()* = v.*find-set()*

  then return *true*

  else return *false*

Linked-list representation

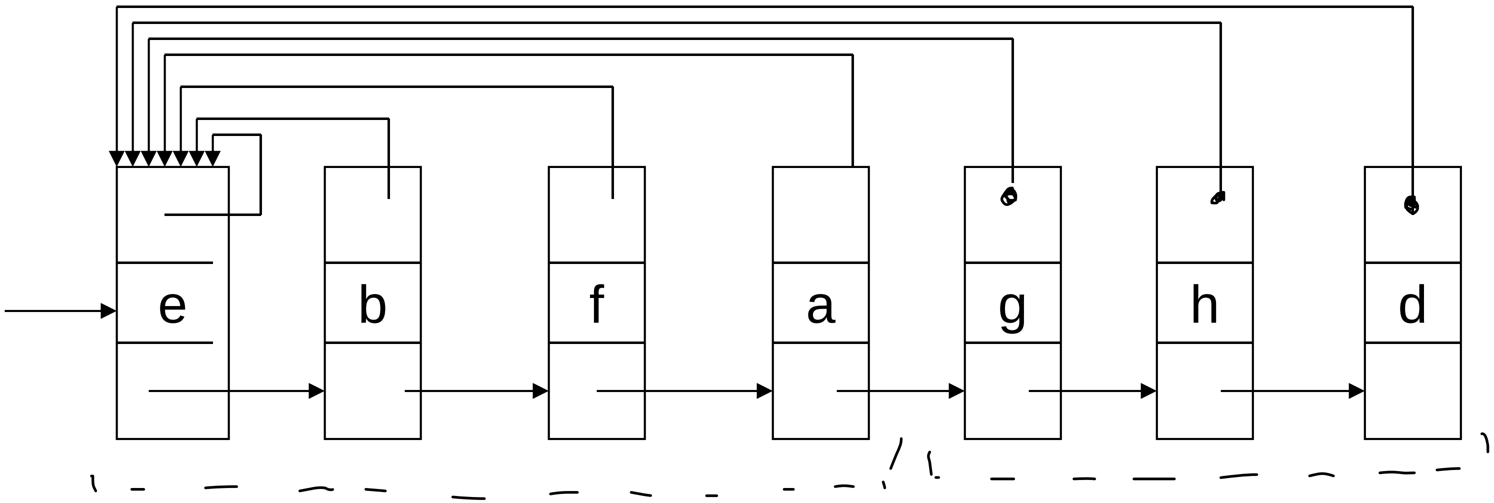




- *x.make-set()*
- *x.find-set()*
- *x.union(y)*

Linked-list representation

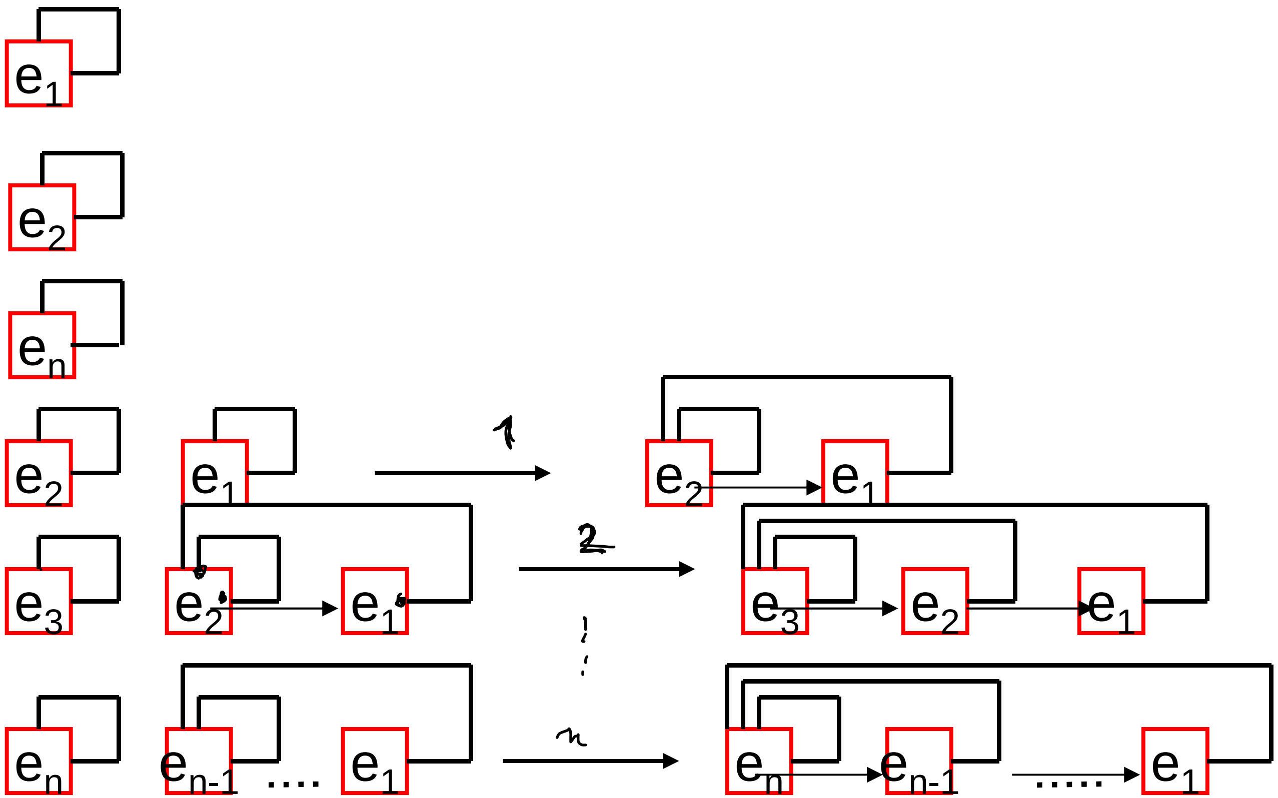
b.union(d)



„Bad“ sequence of operations

$n \begin{cases} e_1.\textit{make-set}() \\ e_2.\textit{make-set}() \\ \vdots \\ e_n.\textit{make-set}() \end{cases}$

$n-1 \begin{cases} e_2.\textit{union}(e_1) \\ e_3.\textit{union}(e_2) \\ \vdots \\ e_n.\textit{union}(e_{n-1}) \end{cases}$



The longer list is always appended to the shorter list!

Pointer updates for the i-th operation $e_i.\textit{union}(e_{i-1})$:

Running time of $2n -1$ operations:

$$n + \sum_{i=1}^{n-1} i = n + \frac{n \cdot (n-1)}{2} = \Theta(n^2)$$

Improvement

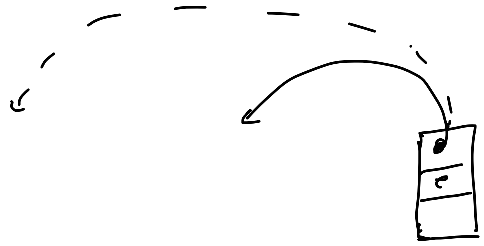
Weighted-union heuristic

Always append the smaller list to the longer list.
(Maintain the length of a list as a parameter).

Theorem

Using the weighted-union heuristic, the running time of a sequence of *m*
make-set, find-set, and *union* operations, *n* of which are *make-set()* operations,
is $O(m + n \log n)$.

Proof



Consider element *e*.

Number of times *e*'s pointer to the representative is updated: log *n*

1.

2.

3.