ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithm Theory

17 Union Find

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

CoNe
Freiburg

IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Union-Find Structures

- **Problem:**
  - Maintain a collection of disjoint sets while supporting the following operations:
  - e.make-set():
    - Creates a new set whose only memeber is e
  - e.find-set():
    - Return the set M_i containing e
  - union(M_i, M_j):
    - Unite the sets M_i and M_j into a new set

Union-Find Structures

- **Representation of set M_i:**
    - M_i is identified by a representative, which is some member of M_i.

Union-Find Structures

- **Operation using representative elements:**
- **e.make-set():**
  - Creates a new set whose only member is *e*. The representative is *e*.
- **e.find-set():**
  - Returns the name of representative of the set containing e.
- **e.union(f):**
  - Unites the sets M_e and M_f that e and f into a new set M and returns a member $M_e \cup M_f$ as the new representative of M. The sets M_e and M_f are erased.

Observations

- If n is the number of the make-set operations and m the total number of makes-set, find-set and union operations, then
  - $m \geq n$
  - after at most $(n - 1)$ union operations, only one set remains in the collection

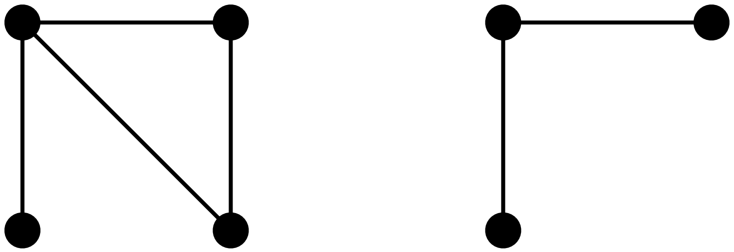
Application: Connected Components

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

Output: collection of the connected components of G

Algorithm: Connected-Components

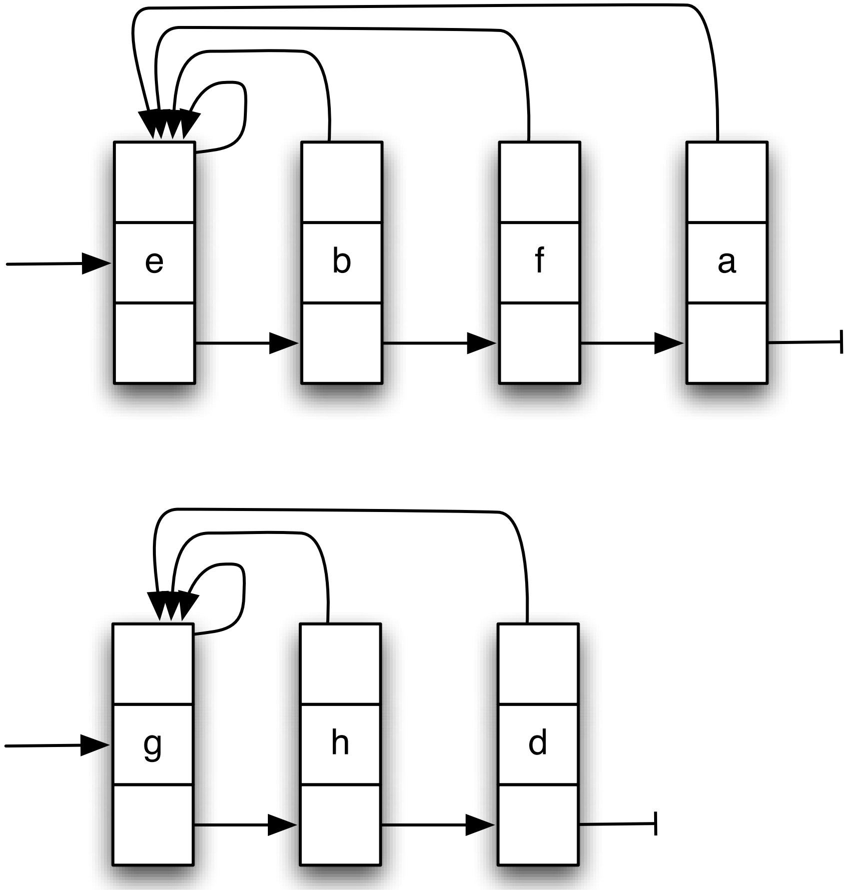
```
for all $v$ in $V$ do $v$.makeset()
for all $(u,v)$ in $E$ do
    if $u$.findset() $\neq$ $v$.findset()
        then $u$.union($v$)
```



Same-Component *(u,v)*:

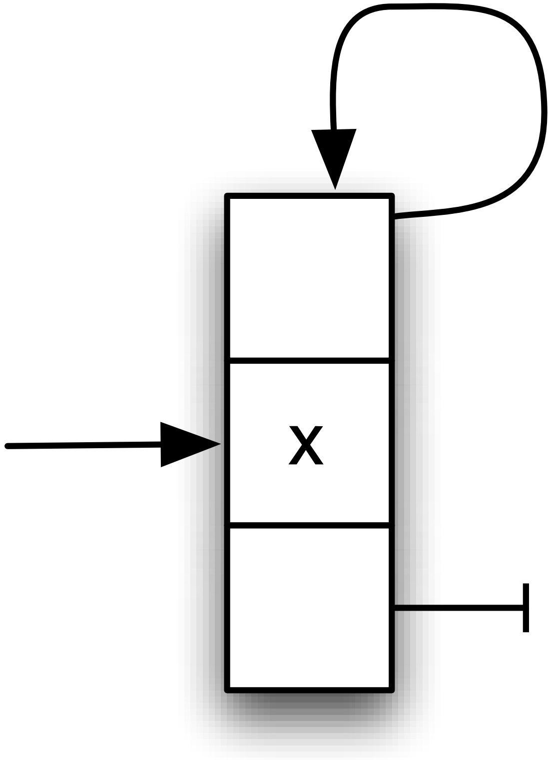
```
    if $u$.findset() = $v$.findset()
    then return true
```

Linked-List Representation

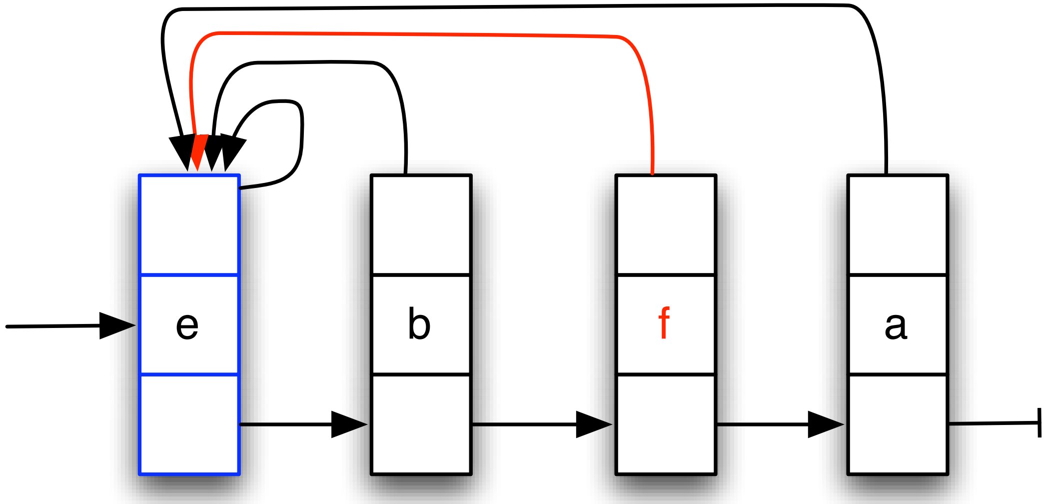


- *x.make-set()*
- *x.find-set()*
- *x.union(y)*

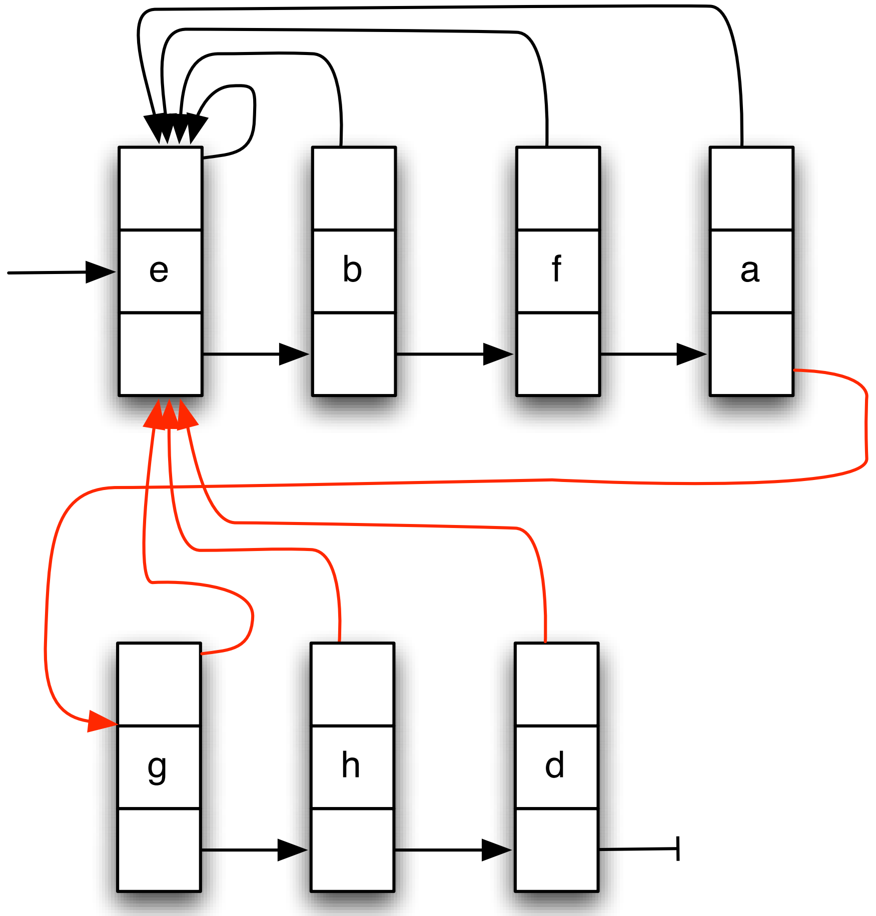
x.makeset()



f.find-set()



b.union(d)



Expensive Sequence of Operations

e_1.*make-set()*

e_2.*make-set()*

⋮

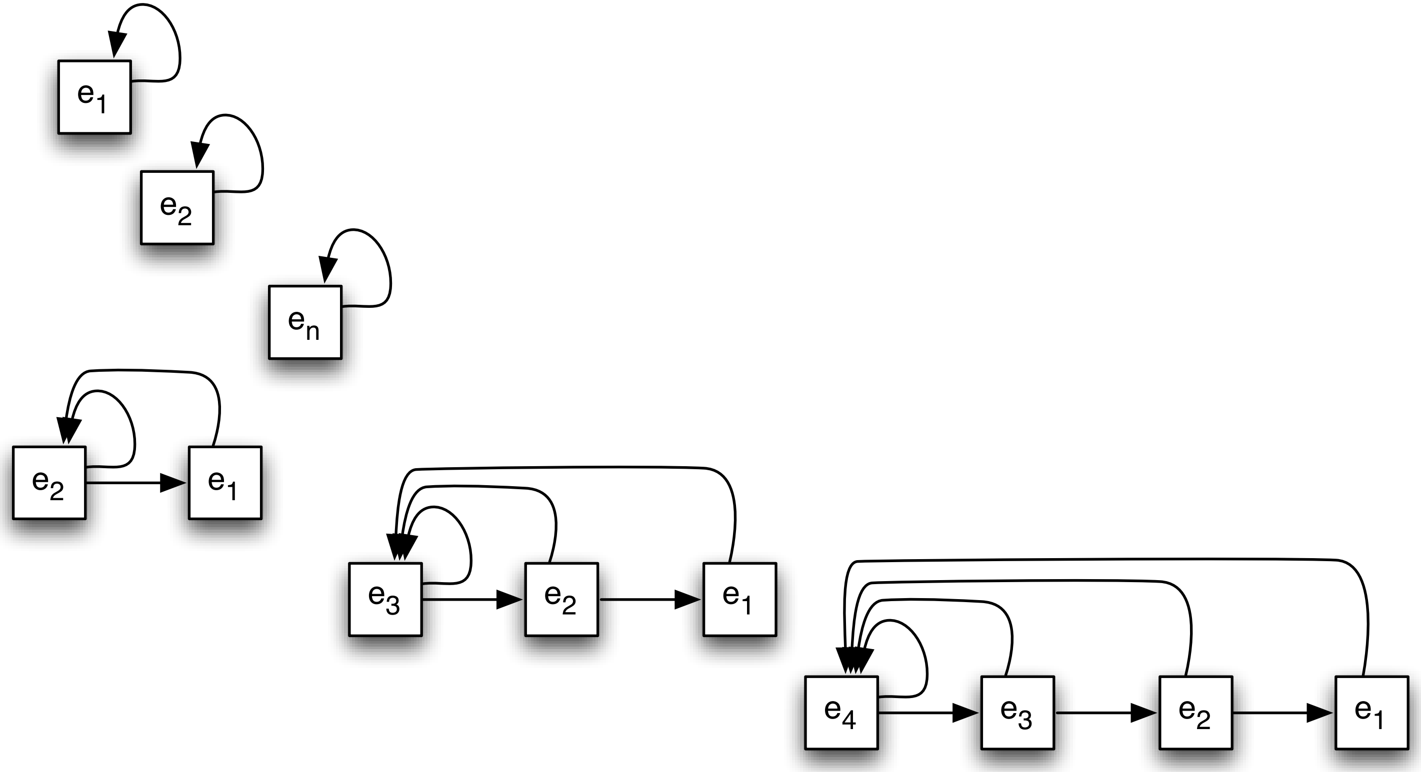
e_n.*make-set()*

e_2.*union*(e_1)

e_3.*union*(e_1)

⋮

e_n.*union*(e_{n-1})



The longer list is always is appended to the shorter list!

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

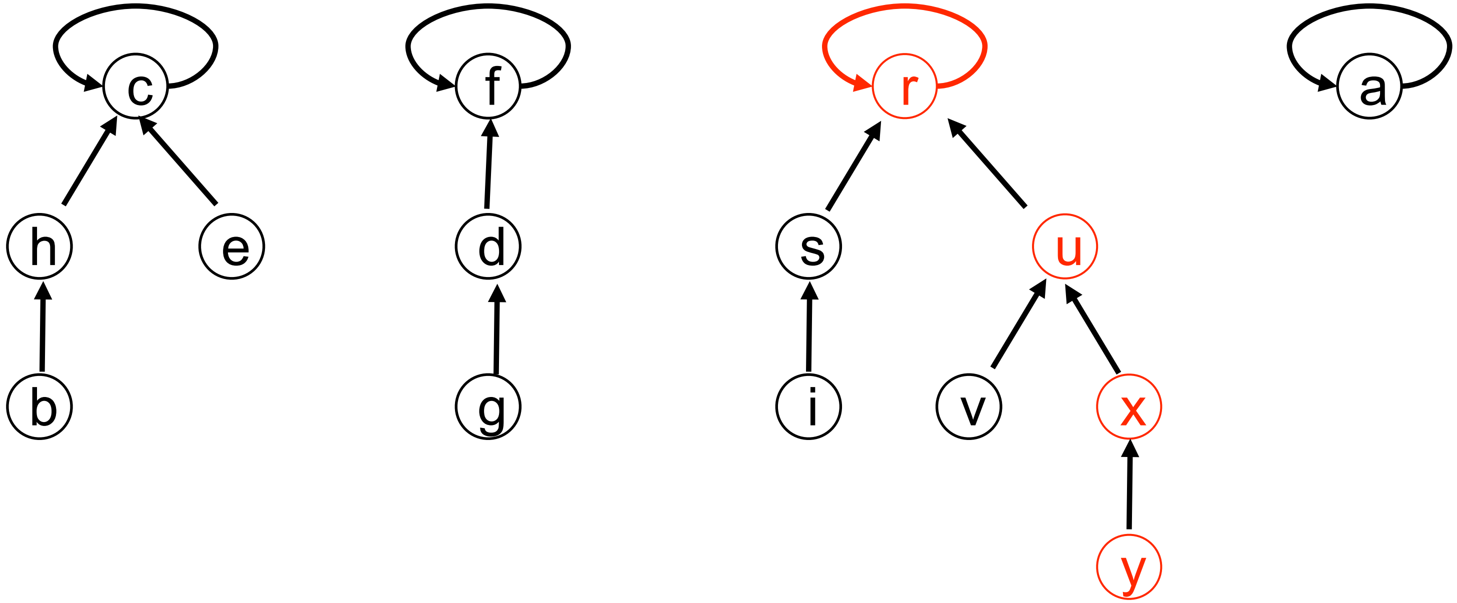
Running time of 2n -1 operations:

Improvement

- **Union by Size**
  - 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, where n are make-set() operations, is O(m+n log n)
- **Proof**
  - Consider element e.
  - Number of times e‘s pointer to the represented is updated: log n

- Whenever an element has to update its pointer it becomes the element of a set with a size which as as least doubled
- because of the weighted union heuristic
- Every element experiences at most log n union pointer changes

Disjoint Set Forests



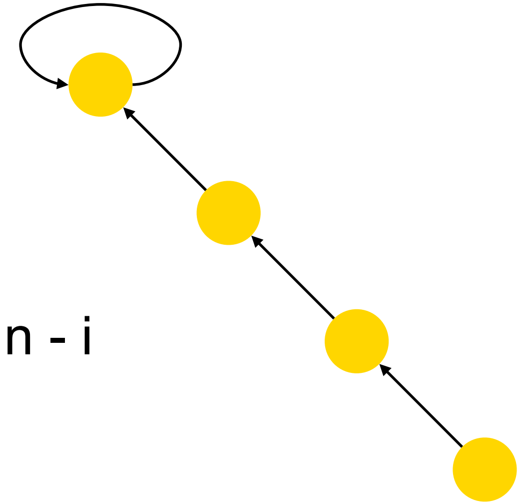
- a.make-set(): as before
- y.find-set(): Follow path upwards
- d.union(e): Make the representative of one set (e.g. f) the parent of the representative of the other set

Example

m = total number of operations(≥ 2n)

```
    for $i = 1$ to $n$ do $e_i$.make-set( )
    for $i = n$ to $2$ do $e_{i-1}$.union($e_i$ )
    for $i = 1$ to $f$ do $e_n$.find-set( )
```

i – th step



running time of f find-set operations: $O(f \ n)$

Union by Size

Additional variable:

e.size **= (#nodes in the subtree rooted *e*)**

e.make-set()

```
1  e.parent = e
2  e.size = 1
```

e.union(f)

```
1  Link(e.find-set( ), f.find-set( ))
```

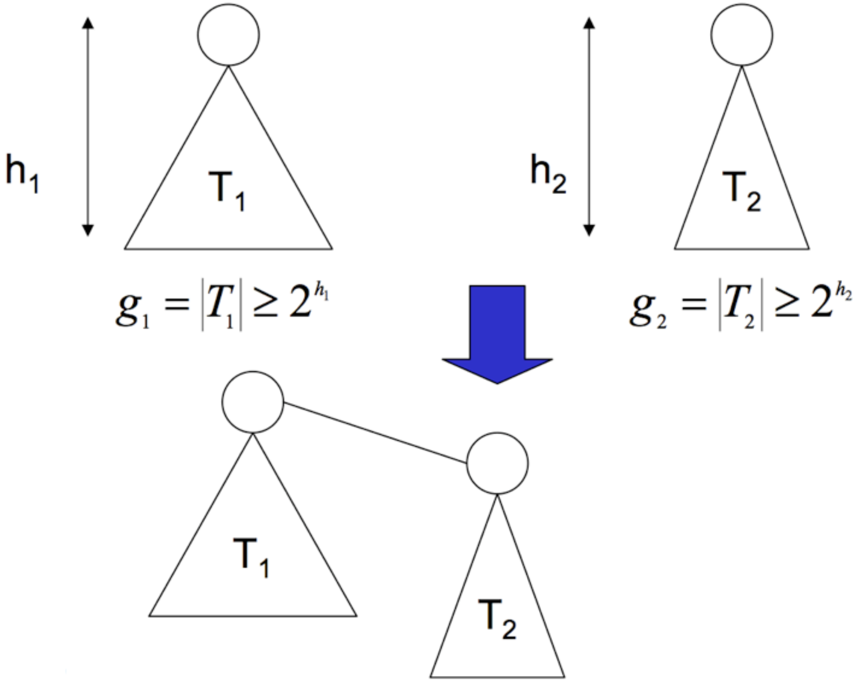
Union by Size

Link(e,f)

```
if $e.size \ge f.size$
    then $f.parent = e$
         $e.size = e.size + f.size$
    else /* $e.size < f.size$ */
         $e.parent = f$
         $f.size = e.size + f.size$
```

Union by Size

- **Theorem**
  - The method union-by-size maintains the following invariant: A tree of height h contains at least 2^h
- **Proof**



Union by Size

Case 1: The height of the new tree is equal of the height of T_1

$$g_1 + g_2 \geq g_1 \geq 2^{h_1}$$

Case 2 : The new tree T has a greater height

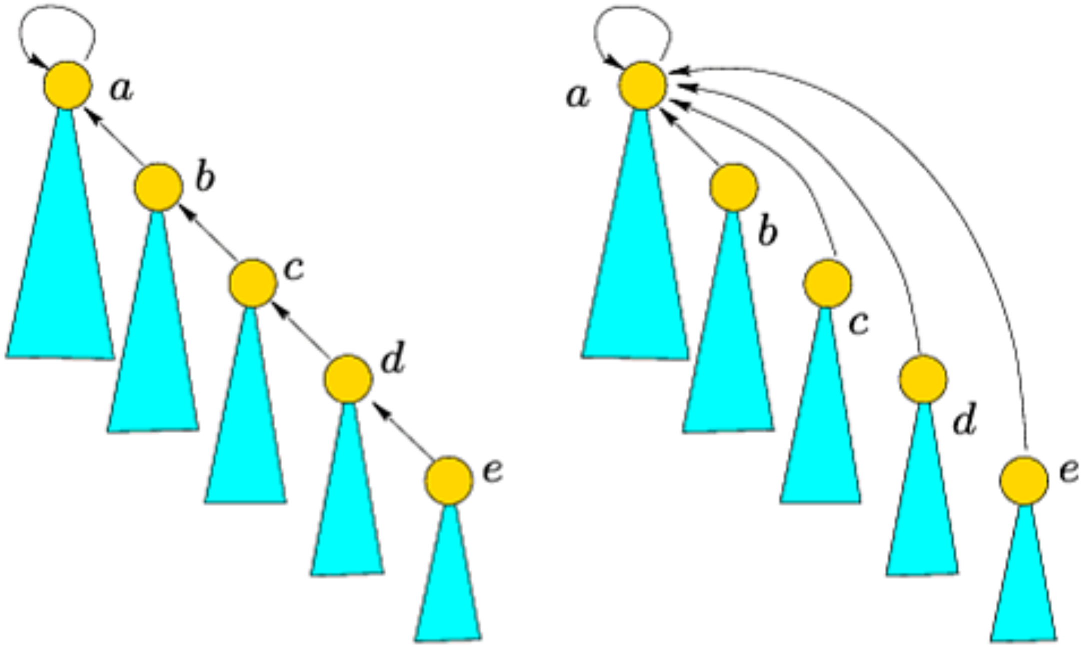
Height of T**:** $h_2 + 1$

$$g = g_1 + g_2 \geq 2^{h_2} + 2^{h_2} = 2^{h_2+1}$$

Consequence

The running time of find-set operation is O(log n), where n is the number of make-set operations.

Path Compression during Find-Set Operations



```
e.find-set()
  if $e \neq e.parent$
    then $e.parent = e.parent.find\text{-}set(\,)$
  return
```

Analysis of Running Time

m total number of operations,

f *of which are find-set* operations and

n of which are make-set operations

→ at most $n-1$ *union*-operations

Union by size:

$O(n + f \log n)$

Find-set operations with path compression:

if $f < n$, $\Theta(n + f \log n)$

if $f \geq n$, $\Theta(f \log_{1+f/n} n)$

Analysis of the Running Time

Theorem (Union by size with path compression)

Using the combined union-by-size and path-compression heuristic, the running time of m disjoint-set operation of n elements is $\Theta(m * \alpha(m,n))$

where $\alpha(m,n)$ is the inverse of the Ackermann‘s function

Ackermann Function und Inverse

Ackermann‘s function

$A(1,j) = 2^j$ for $j \geq 1$

$A(i,1) = A(i-1,2)$ for $i \geq 2$

$A(i,j) = A(i-1, A(i, j-1))$ for $i,j \geq 2$

Inverse Ackermann‘s function

$$\alpha(m,n) = \min\{i \geq :\ A(i, \lfloor m/n \rfloor) > \log n\}$$

Ackermann‘s function and its Inverse

$$A(i, \lfloor m/n \rfloor) \geq A(i,1)$$

$$A(2,1) = A(1,2) = 2^2 = 4$$

$$A(3,1) = A(2,2) = A(1, A(2,1)) = 2^4 = 16$$

$$A(4,1) = A(3,2) = A(2, A(3,1)) = A(2,16)$$

$$\geq 2^{2^{2^{2^{2}}}} = 2^{65536}$$

$$\alpha(m,n) \leq 4, \text{ für } \log n < 2^{65536}$$

ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithm Theory

17 Union Find

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

CoNe
Freiburg

IIF
INSTITUT FÜR
INFORMATIK
FREIBURG