Disjoint Sets

Data Structures and Algorithms

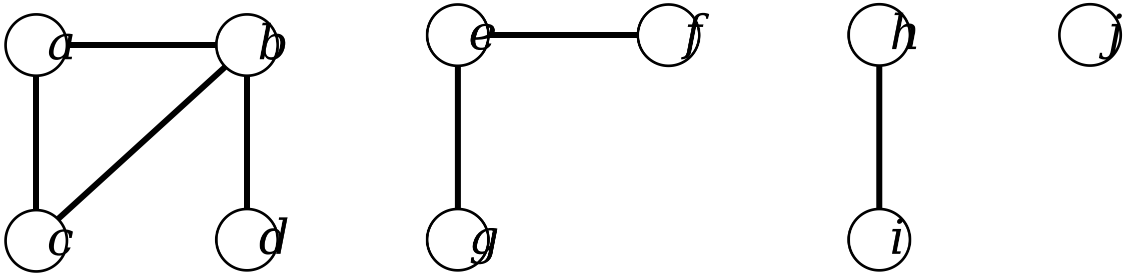
Nanjing University, Fall 2021
郑朝栋

DisjointSet ADT

- A **disjoint-set** ADT maintains a collections $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of *sets* that are *disjoint* and *dynamic*.
- Each set S_i has a "*representative*" member (i.e., a "*leader*").
- DisjointSet ADT supports following operations:
  - **MakeSet**(x): create a set containing only x, add the set to $\mathcal{S}$.
  - **Union**(x, y): find the sets containing x and y, say S_x and S_y;
    remove S_x and S_y from $\mathcal{S}$, add $S_x \cup S_y$ to $\mathcal{S}$.
  - **Find**(x): return a pointer to the leader of the set containing x.
- Does not support "remove" elements, or "split" sets.

Sample application of DisjointSet ADT

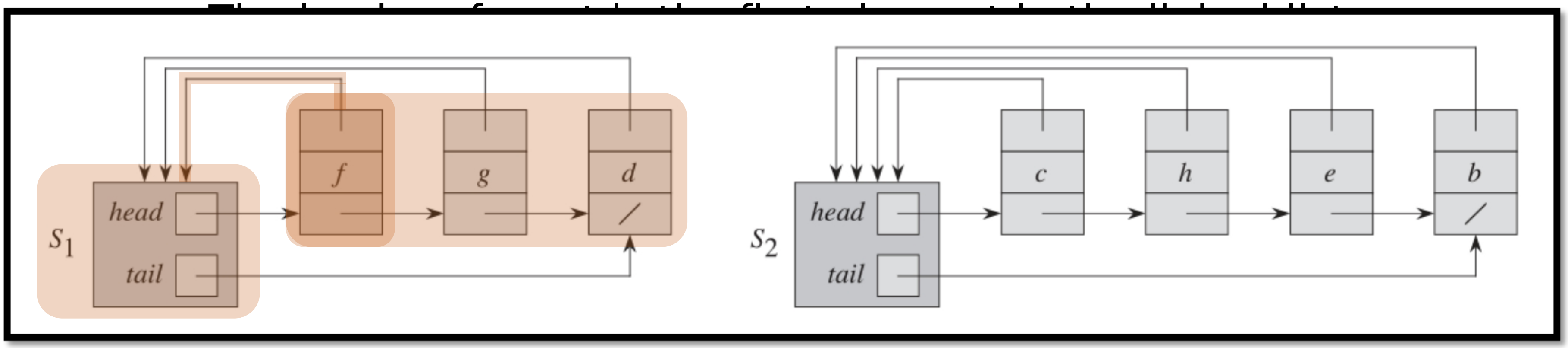
Computing connected components



Edge processed	Collection of disjoint sets									
MakeSet(·)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
Union(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Linked-list implementation of DisjointSet

- Basic Idea: Use a linked list to store and represent a set.
- Some more details:
  - A set object has pointers pointing to head and tail of the linked-list.
  - The linked-list contains the elements in the set.
  - Each element has a pointer pointing back to the set object.



Linked-list implementation of DisjointSet

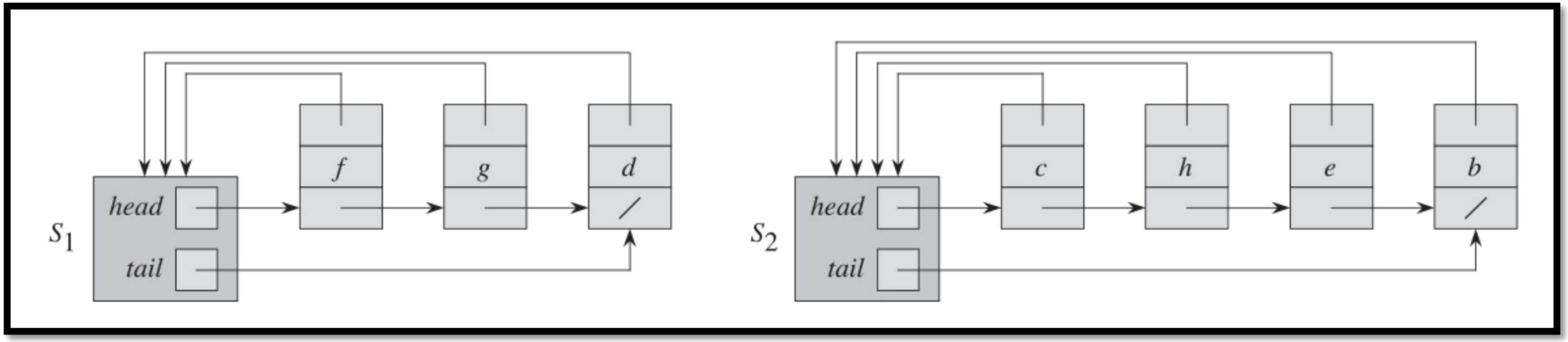
- Basic Idea: Use a linked list to store and represent a set.

$\Theta(1)$

- **MakeSet**(x): Create a linked list containing only x.

$\Theta(1)$

- **Find**(x): Follow pointer from x back to the set object, then return pointer to the first element in the linked-list.

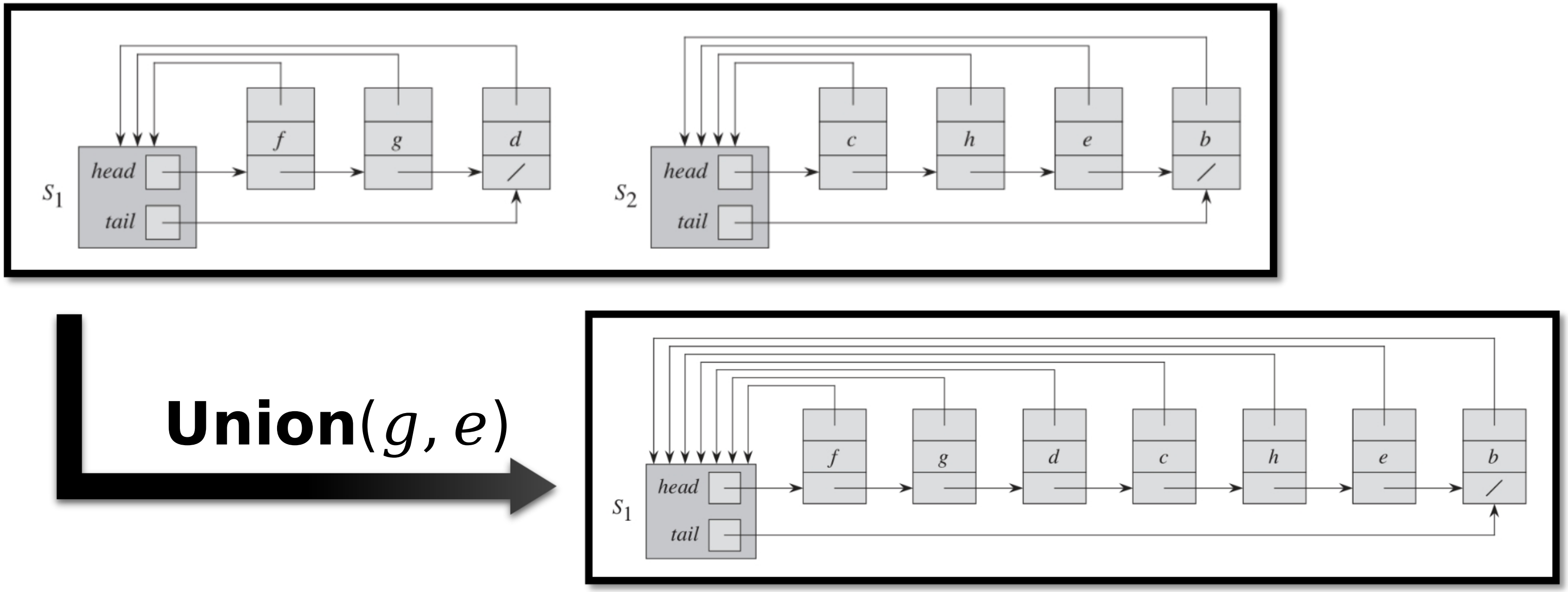


Linked-list implementation of DisjointSet

- Basic Idea: Use a linked list to s epresent a set. $\Theta(1)$
- **Union**(x, y): App S_y to list in S_x; destroy set object S
  object pointers for elements originally in S_y

Union can be slow, even in amortized sense!

Time depends on size of S_y.



Union(g, e)

Linked-list implementation of DisjointSet

- Basic Idea: Use a linked list to store and represent a set.
- **Union**(x, y): Append list in S_y to list in S_x; destroy set object S_y;
  update set object pointers for elements originally in S_y.

```
MakeSet($x_0$)
for (i=1 to n)
  MakeSet($x_i$)
  Union($x_i$,$x_0$)
```

Complexity of this sequence of operations?

$\Theta(n^2)$ in total.

Each **MakeSet** takes $\Theta(1)$ time,
but the average cost of **Union** reaches $\Theta(n)$.

Union operation is too expensive!

Linked-list implementation of DisjointSet

- **Improvement**: *Weighted-union* heuristic (or, *union-by-size*).
- **Basic Idea**: In **Union**, append the shorter list to the longer one!

```
MakeSet(x_0)
for (i=1 to n)
  MakeSet(x_i)
  Union(x_i,x_0)
```

n: Need to maintain size for each set (but this

Complexity of this sequence of operations?

Worst complexity of any sequence of $n+1$ MakeSet and then n Union? $O(n\lg n)$

Proof:

- The $n+1$ **MakeSet** o… e in total.
- For **Union** o… ed by set obj. pointer cha
- For … whenever its set obj. pointer change
  e at least doubles!
- Each element's set obj. pointer changes $O(\lg n)$ times
- The n **Union** op. take $O(n\lg n)$ time in total.

Average cost of **Union** operation is reduced to $O($

Rooted-tree implementation of DisjointSet

- Basic idea: Use a rooted-tree to represent a set; the root of a tree is the "leader" of that set.
- Some details: Each node has a pointer pointing to its parent; parent of a "leader" is the leader itself.
- **MakeSet**(x): Create a tree containing only (root) x; parent of x is x. $\Theta(1)$
- **Find**(x): Follow parent pointer from x back to the root, and return root.
- **Union**(x, y): Change the parent pointer of the r

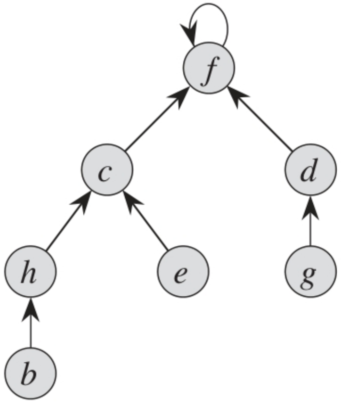
Time complexity depends on depth of x and y.

Linked-list vs Rooted-tree

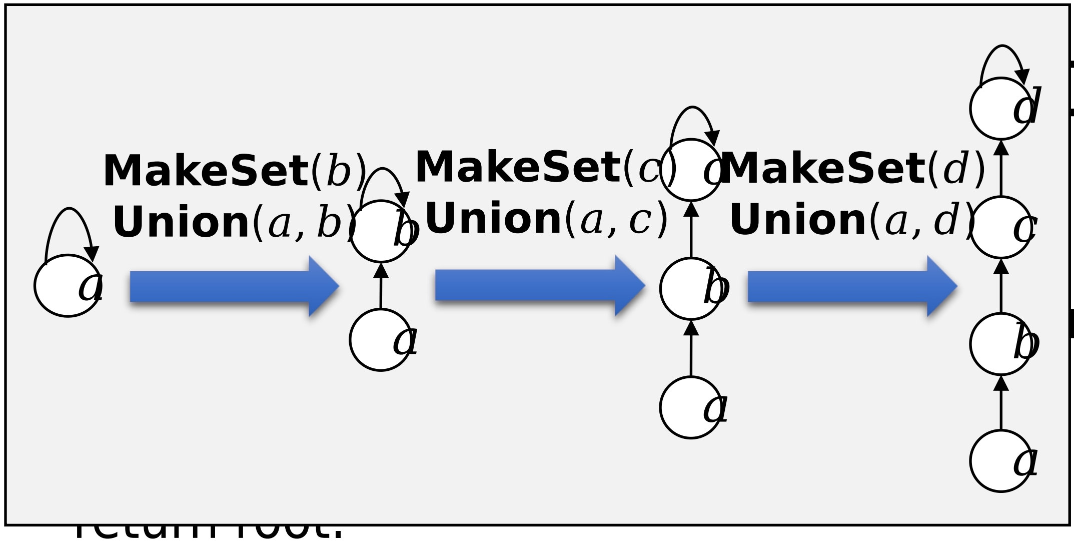
Implementation:

- **MakeSet** is fast in both cases.
- **Linked-list:** **Find** is fast, but **Union** is slow.
- **Rooted-tree:** **Find** is slow, but **Union** is fast.

(If **Union** always unions roots of trees.)



tation of



MakeSet(b) **Union**(a, b)

MakeSet(c) **Union**(a, c)

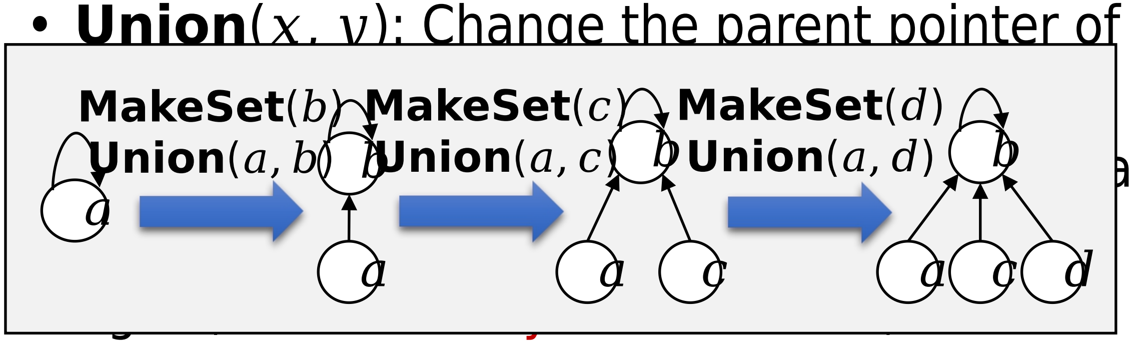
MakeSet(d) **Union**(a, d)

nly (root) x; parent of x

ck to the root, and

return root.

- **Union**(x, y): Change the parent pointer of th



MakeSet(b) **Union**(a, b)

MakeSet(c) **Union**(a, c)

MakeSet(d) **Union**(a, d)

```
MakeSet(x$_0$)
for (i=1 to n)
  MakeSet(x$_i$)
  Union(x$_0$,x$_i$)
Find(x$_0$)
```

avg

worst-case cost of **Union** and **Find** to $O(\lg n)$.
(Each time a node's depth increases, the tree size at least doubles. So size n tree has height $O(\lg n)$.)

- Alternatively, use union-by-height heuristic: In **Union**, let tree of larger height.

Can we do better?

- Union-by-height re worst-case cost of **Union** and **Find** to $O(\lg n)$.

Rooted-tree implementation of DisjointSet

Path-compression in **Find**

e containing only (root) x; parent of x

0.

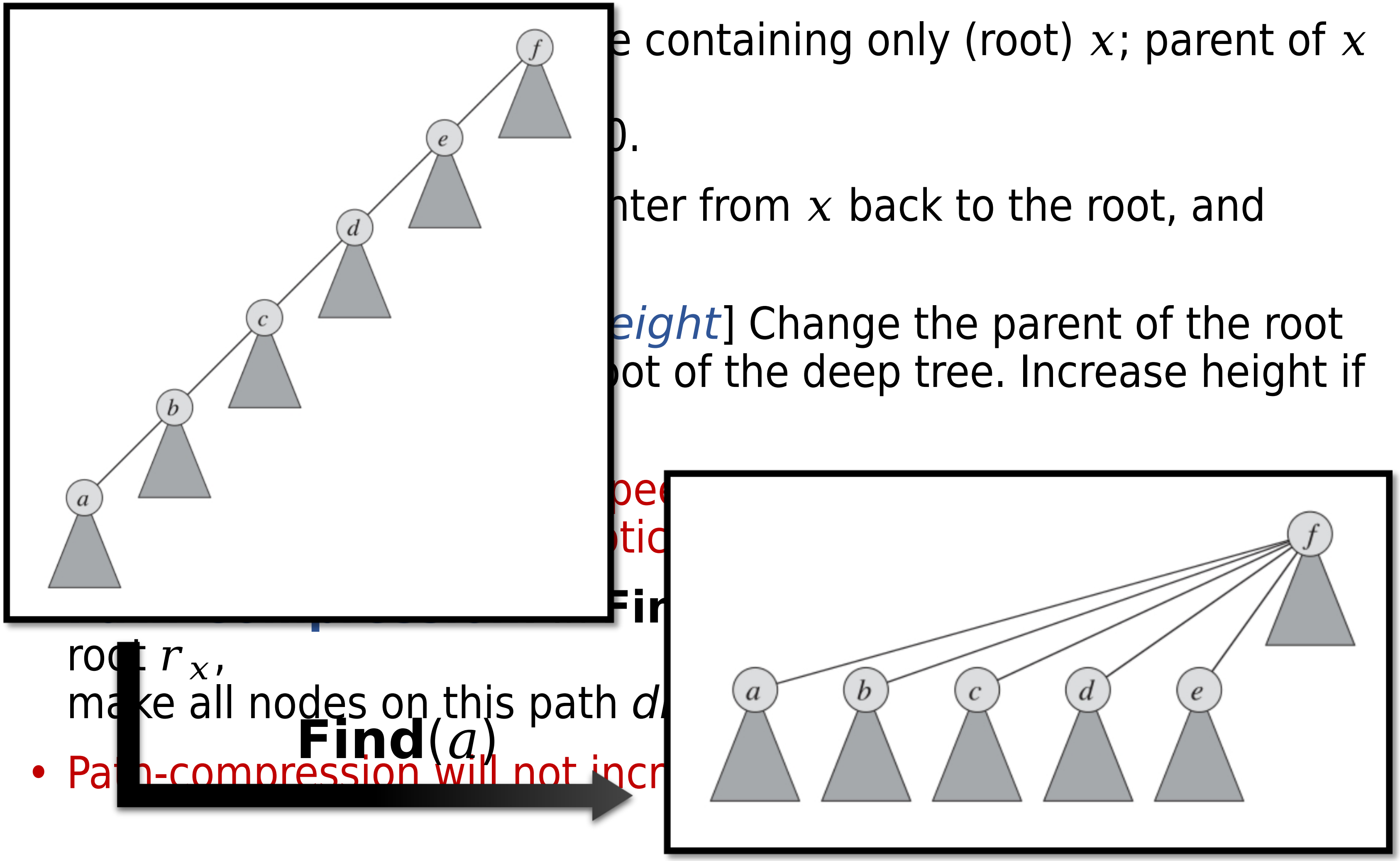
nter from x back to the root, and

eight] Change the parent of the root
oot of the deep tree. Increase height if

pee
otic

Fir
root r_x,
make all nodes on this path d

- Path-compression will not incr



Rooted-tree implementation of DisjointSet

Union-by-height and Path-compression

- ~~**MakeSet**(x): Create a tree containing only (root) x; parent of x is x.~~
  Height of the tree is set to 0.
- **Find**(x): [*path-compression*] Follow parent pointer from x back to root;
  let nodes along the path directly point to root; lastly return root.
- **Union**(x, y): [*union-by-height*] Change the parent of the root of the shallow tree to the root of the deep tree. Increase height if necessary.
- **Find** can now change heights! Maintaining heights becomes expensive!
- Maintain ***rank***, which is like "height ignoring path-compression."
  (rank is always an upper bound of height.)
- **MakeSet**(x): Create a tree containing only (root) x; parent of x is x.
  Rank of the node is set to 0.

Rooted-tree implementation of DisjointSet

Union-by-rank and Path-compression

- **MakeSet**(x): Create a tree containing only (root) x; parent of x is x.
  Rank of the node is set to 0.
- **Find**(x): [*path-compression*] Follow parent pointer from x back to root;
  let nodes along the path directly point to root; lastly return root.
- **Union**(x, y): [*union-by-rank*] Change the parent of the root with lower rank to the root with higher rank. Increase rank of new root if necessary.

- Very efficient implementation of DisjointSet,
  MakeSet is $O(1)$, **Find** and **Union** are both *almost $O(1)$ on average*.
- Analysis is highly non-trivial, but we'll do it!

Rooted-tree implementation with union-by-rank and path-compression

Performance Analysis

- **MakeSet**(x): Create a tree containing only (root) x; parent of x is x.
  Rank of the node is set to 0.
- **Find**(x): [*path-compression*] Follow parent pointer from x back to root;
  let nodes along the path directly point to root; lastly return root.
- **Union**(x, y): [*union-by-rank*] Change the parent of the root with lower rank to the root with higher rank. Increase rank of new root if necessary.

- **Goal:** Any sequence of **MakeSet**, **Find**, **Union** op. has low avg. cost.
- **Observation:** (a) **MakeSet** can be moved to the beginning of op. sequence, without affecting cost. (b) **MakeSet** itself has low cost.
- **New Goal:** Starting from a forest containint n nodes,
  any sequence of **Find** and **Union** op. has low avg. cost

Rooted-tree implementation with union-by-rank and path-compression

Performance Analysis

- **MakeSet**(x): Create a tree containing only (root) x; parent of x is x.
  Rank of the node is set to 0.
- **Find**(x): [*path-compression*] Follow parent pointer from x back to root;
  let nodes along the path directly point to root; lastly return root.
- **Union**(x, y): [*union-by-rank*] Change the parent of the root with lower rank to the root with higher rank. Increase rank of new root if necessary.

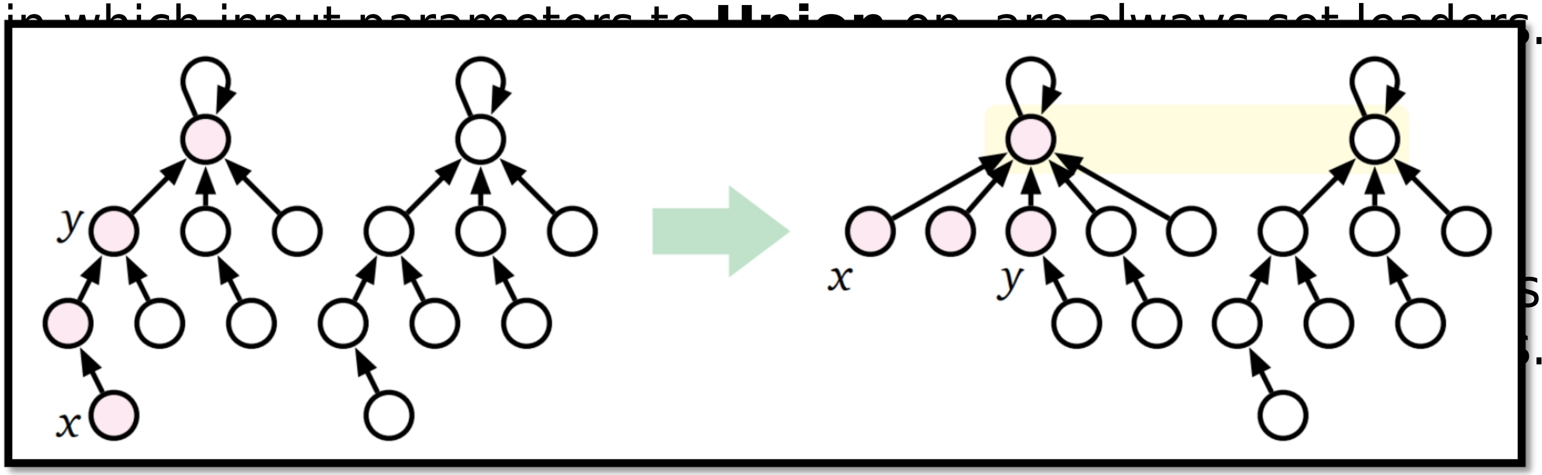
- **Goal:** Starting from a forest containint n nodes,
  any sequence of **Find** and **Union** op. has low avg. cost.
- **Observation:** Cost[**Union**(x, y)] = Cost[**Find**(x)] + Cost[**Find**(y)] + $O(1)$.
- **New Goal:** Starting from a forest containint n nodes,
  any sequence of **Find** and **Union** op. has low avg. cost,
  in which input parameters to **Union** op. are always set leaders

Rooted-tree implementation with union-by-rank and path-compression

Performance Analysis

- **Find**(x): [*path-compression*] Follow parent pointer from x back to root;
  let nodes along the path directly point to root; lastly return root.
- **PartialFind**(x, y): [*y is ancestor of x*] Follow parent pointer from x back to y; let nodes along the path point to y's parent; return parent of y.

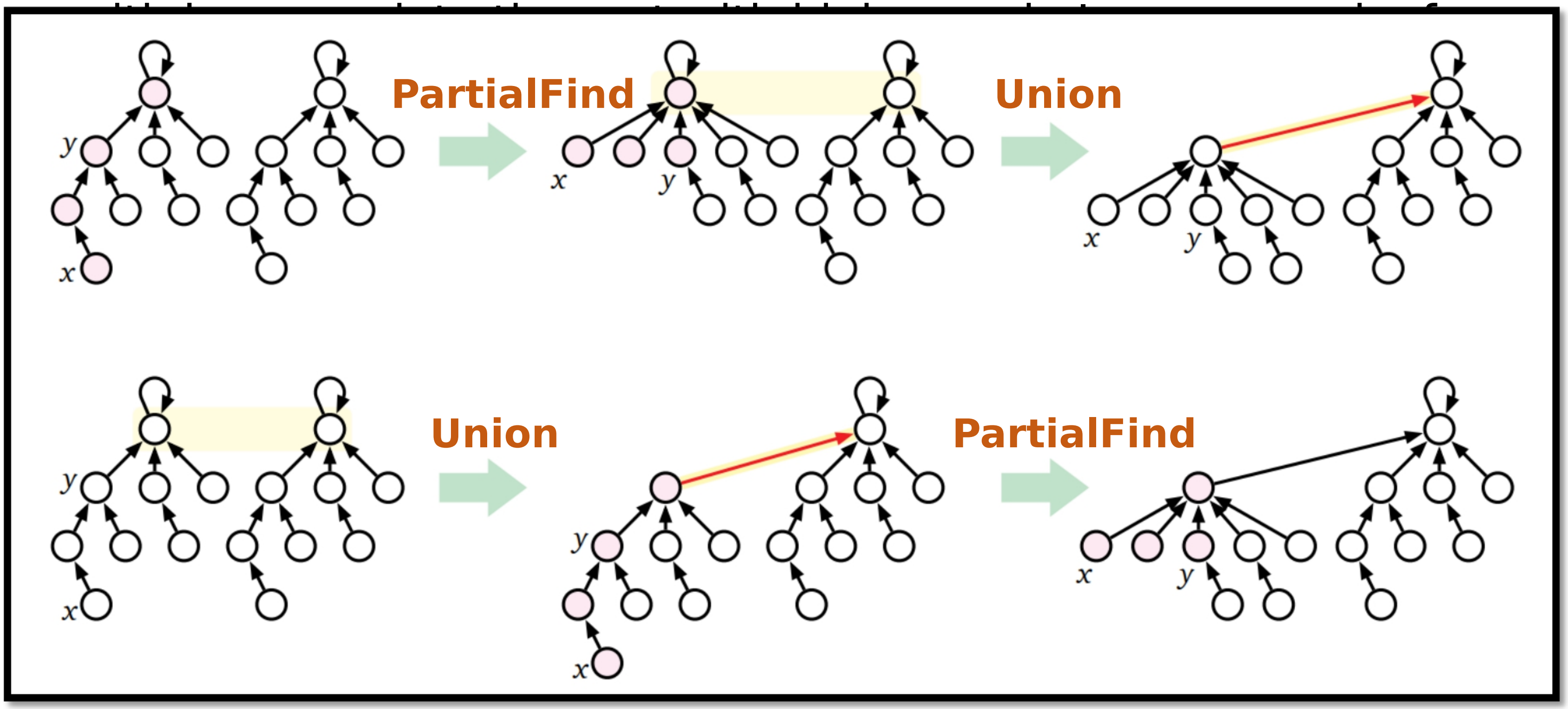
- **Goal:** Starting from a forest containint n nodes,
  any sequence of **Find** and **Union** op. has low avg. cost,
  in which input parameters to **Union** op. are always set leaders.



Rooted-tree implementation with union-by-rank and path-compression

Performance Analysis

- **PartialFind**(x, y): [*y is ancestor of x*] Follow parent pointer from x back to y; let nodes along the path point to y's parent; return parent of y.
- **Union**(x, y): [*union-by-rank*] Change the parent of the root



Rooted-tree implementation with union-by-rank and path-compression

Performance Analysis

- **PartialFind**(x, y): [*y is ancestor of x*] Follow parent pointer from x back to y; let nodes along the path point to y's parent; return parent of y.
- **Union**(x, y): [*union-by-rank*] Change the parent of the root with lower rank to the root with higher rank. Increase rank of new root if necessary.

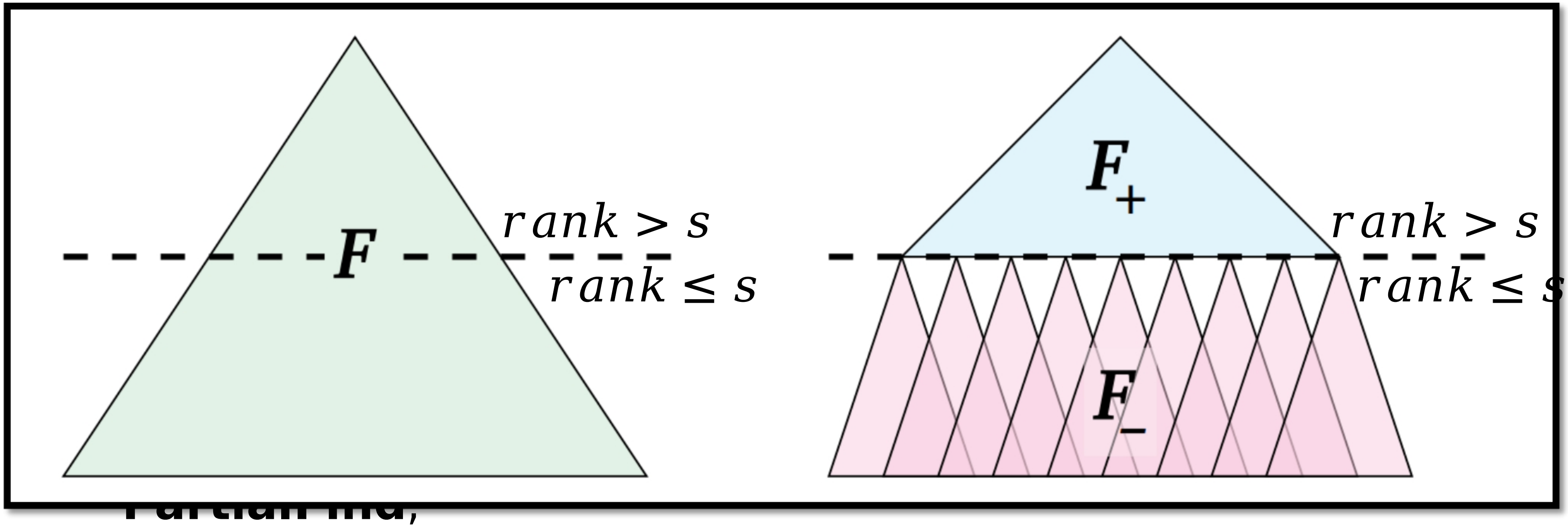
- **Goal:** Starting from a forest containint n nodes, any sequence of **PartialFind** and **Union** op. has low avg. cost, in which every **Union** occurs before any **PartialFind**, and input parameters to **Union** op. are always set leaders.
- **Observation:** Each **Union** op. only costs $O(1)$.
- **New Goal:** Starting from a forest containint n nodes, any sequence of m **PartialFind** op. has low avg. cost.
- **Observation:** Cost of **PartialFind** is dominated by pointer assignments.
- **New Goal:** Starting from a forest containint n nodes,

Rooted-tree implementation with union-by-rank and path-compression

Performance Analysis

- **PartialFind**(x, y): [*y is ancestor of x*] Follow parent pointer from x back to y; let nodes along the path point to y's parent; return parent of y.
- **Goal:** Starting from a forest containint n nodes, any sequence of m **PartialFind** op. has *low total pointer assignments*.

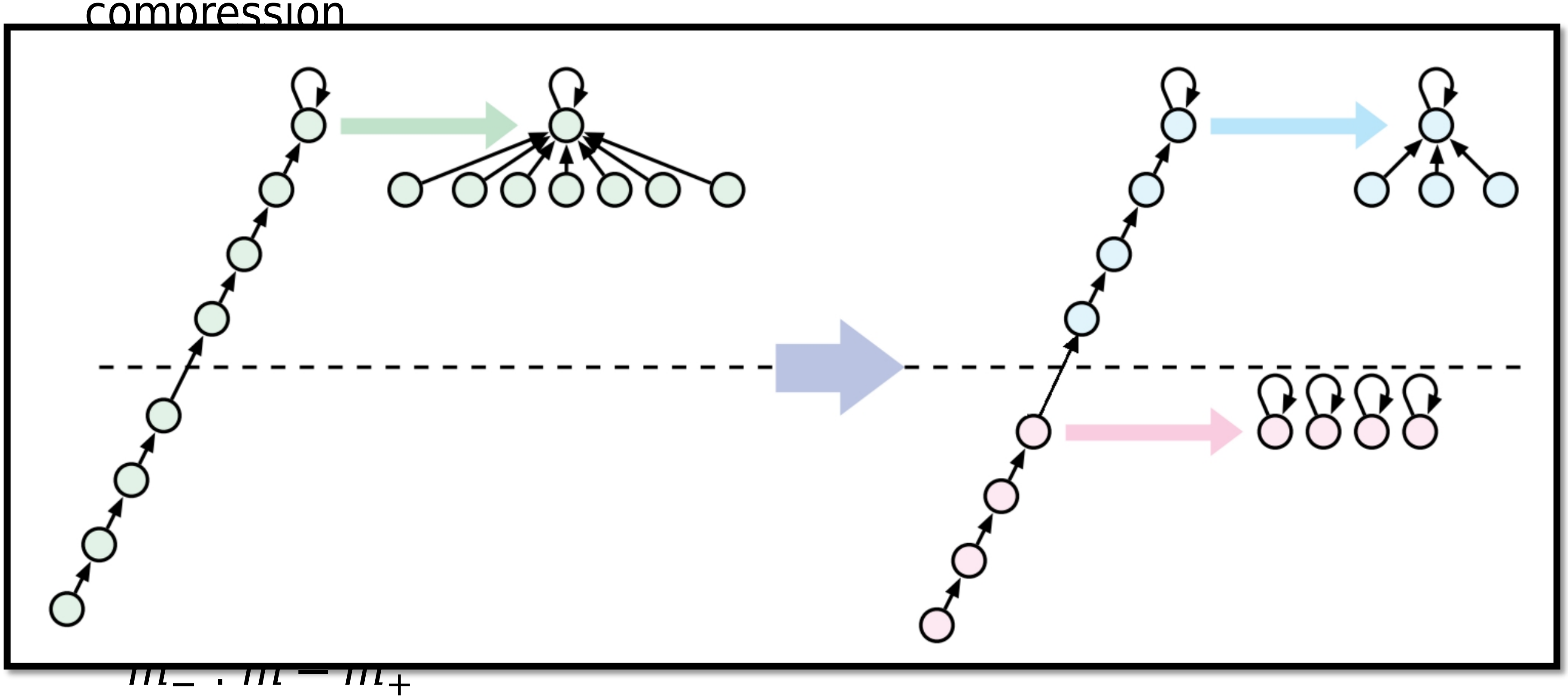
- $T(m, n, r)$: worst # of ptr. assignments in any seq. of m **PartialFind**, starting from a size n forest where each node has rank at most r.
- **Goal:** $T(m, n, r)$ is small.
- **Claim:** $T(m, n, r) \leq nr$.
- **Proof:** Each node can change parent at most r times, since each new parent has higher rank than the old one.



Partial find,
starting from a size n forest where each node has rank at most r.

- Fix forest F of n nodes with max rank r, and a seq. C of m **PartialFind** on F.
- $T'(F, C)$: total # of ptr. assignments occurred in C.
- Let s be an arbitrary positive rank, partition F into F_- and F_+.
- [High Forest] F_+: containing nodes with rank $> s$;
  [Low Forest] F_-: containing nodes with rank $\le s$.
- Let $|F_+| = n_+$, and $|F_-| = n_-$
- m_+: number of ops. in C that involve any node in F_+

Rooted-tree implementation with union-by-rank and path-compression



$m_- : m - m_+$

- Consider a **PartialFind**(x, y) in C:
- If $rank(x) > s$: the op. is a **PartialFind** op. in F_+.
- If $rank(y) \leq s$: the op. is a **PartialFind** op. in F_-.
- If $rank(x) \leq s$ and $rank(y) > s$:
  Split the op. into (a) a **PartialFind** op. in F_+; (b) some *shatter* op. in F_-;
  and (c) a pointer assignment for the “topmost” node in F

Consider a **PartialFind**(x, y) in C:

- If $rank(x) > s$: the op. is a **PartialFind** op. in F_+.
- If $rank(y) \leq s$: the op. is a **PartialFind** op. in F_-.
- If $rank(x) \leq s$ and $rank(y) > s$:
  Split the op. into (a) a **PartialFind** op. in F_+; (b) some *shatter* op. in F_-;
  and (c) a pointer assignment for the "topmost" node in F_-.

We have converted C into:

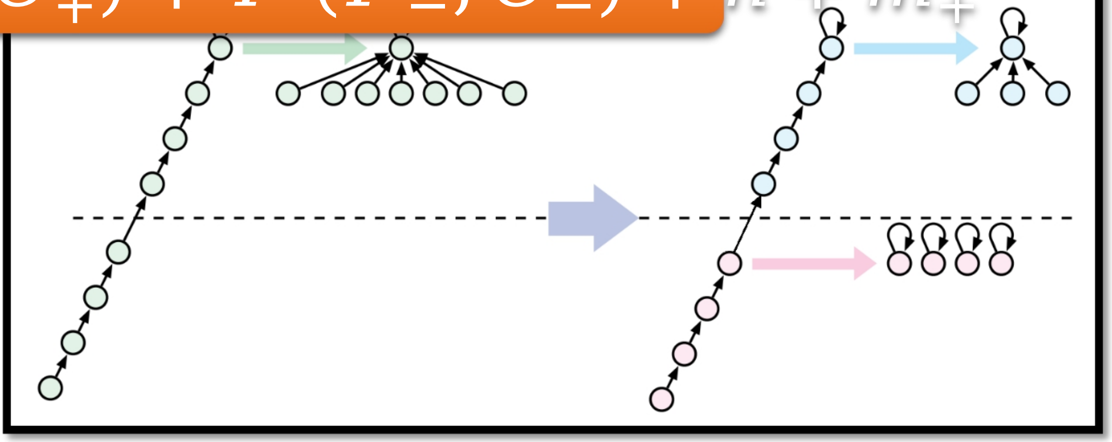
(a) C_+: ops involving nodes only in F_+; (b) C_-: ops involving nodes only in F

(c) shatter ops; and (d) pointer assignments for "topmost" nodes in F_-.

Observation: each node get shattered at most once (then be "topmost" n

Observation: there are at most m_+ pointer assignments for "topmost" nodes in F_-

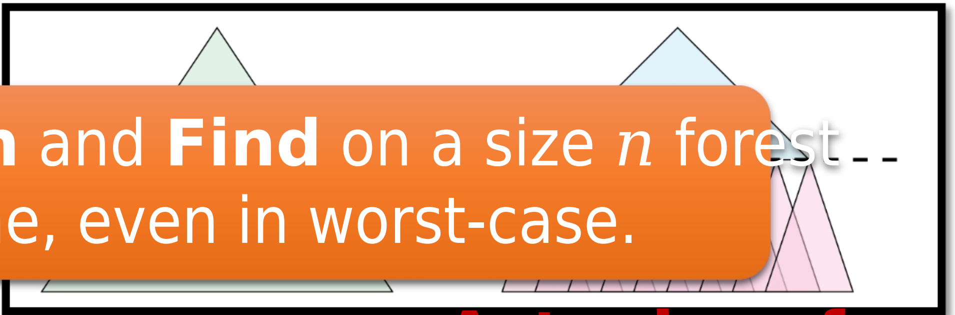
$$T'(F, C) \leq T'(F_+, C_+) + T'(F_-, C_-) + n + m_+$$



Rooted tree implementation wit

Pe

Any sequence of m **Union** and **Find** on a size n forest takes $O(m + 2n\lg^* n)$ time, even in worst-case.



Actual performa
is even better!

- $T'(F, C) \leq T'(F_+, C_+) + T'(F_-, C_-) + n + m_+$
- Nodes in F_+ has rank at least $s + 1$ and at most r;
  Nodes in F_- has rank at most s.
- **Strategy**: obtain a bound of $T'(F_+, C_+)$ to get recurrence of $T'(F, C)$.
- Claim: $T(m, n, r) \leq nr$.
  (Recall $T(m, n, r)$: worst # of ptr. assignments in any seq. of m **PartialFind**, starting from a size n forest where each node has rank at most r.)
- Claim: There are at most $n/2^i$ nodes of rank i in any size n forest.
- $T'(F_+, C_+) \leq n_+ \cdot r \leq \left(\sum_{i>s} \frac{n}{2^i}\right) \cdot r = \frac{nr}{2^s}$
- Fix $s = \lg r$, then $T'(F, C) \leq T'(F_-, C_-) + 2n + m_+$,
  or equivalently, $T'(F, C) - m \leq (T'(F_-, C_-) - m_-) + 2n$
- $T''(m, n, r) \leq T''(m, n, \lg r) + 2n$, where $T''(m, n, r) = T(m, n, r) - m$
- $T''(m, n, r) \leq 2n\lg^* r$. That is: $T(m, n, r) \leq m + 2n\lg^* r$.

Summary

- DisjointSet ADT: **MakeSet**(x), **Union**(x, y), and **Find**(x).
- Linked-list based implementation:
  - Use a linked-list to denote a set, first element in list is leader.
  - **Union** is slower, **Find** is fast.
  - With *union-by-size*, **Union** has *average* cost $O(\lg n)$.
- Rooted-tree based implementation:
  - Use a rooted-tree to denote a set, root of the tree is leader.
  - **Union** is fast (if input arg. are leaders), **Find** is slower.
  - With *union-by-size* or *union-by-height*,
    Union and **Find** has *worst-case cost* $O(\lg n)$.
  - With *union-by-rank* and *path-compression*,
    Union and **Find** has *average cost* $O(\lg^* n)$.
    (More careful analysis leads to an even better bound!)

Reading

- [CLRS] Ch.21 (excluding 21.4)
- Compared with CLRS, following material presents a simpler analysis for the performance of the DisjointSet data structure when both union-by-rank and path-compression are used.
- [Weiss] Ch.8 (8.6)
- Lecture notes by Jeff Erickson: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/11-unionfind.pdf