并查集
Disjoint Sets (Union-Find)

钮鑫涛
Nanjing University
2024 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

DisjointSet ADT

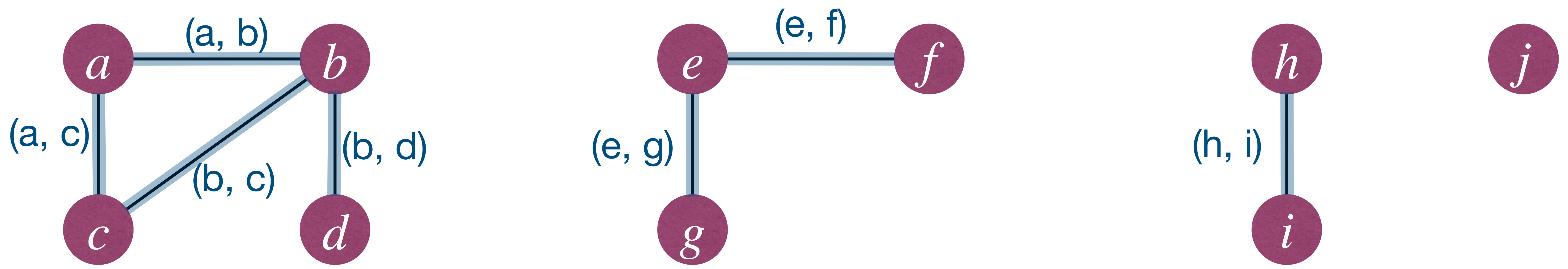
- A disjoint-set ADT (also known as Union-Find ADT) maintains a collections
  - $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of **sets** that are **disjoint** and **dynamic**.
- Each set S_i has a "**representative**" member (i.e., a "leader").
- DisjointSet ADT supports following operations:

Note:Does not support "remove" elements, or "split" sets.

  - `MakeSet(x)`: create a set containing only x, add the set to 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.

Sample application of DisjointSet ADT

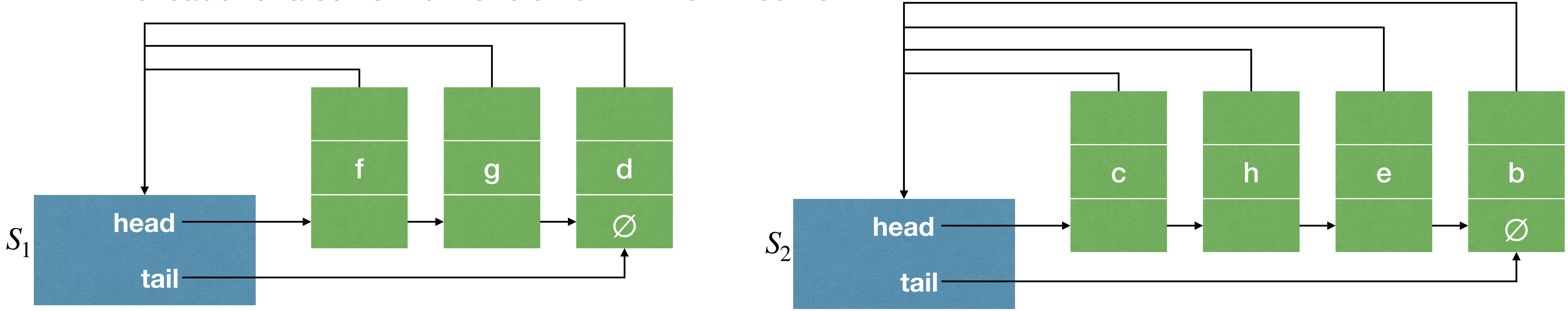
- Computing connected components



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

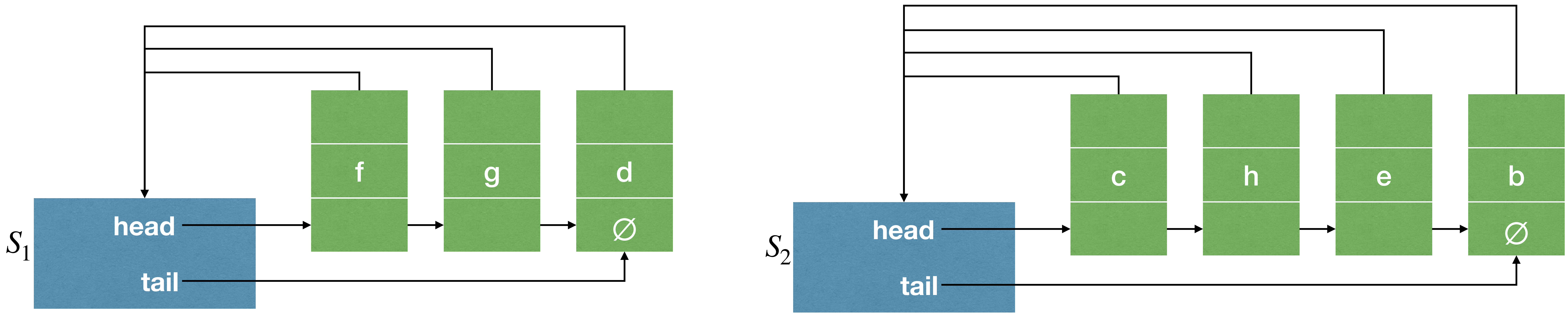
Linked-list implementation of DisjointSet

- 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.
  - The leader of a set is the first element in the linked-list.



Linked-list implementation of DisjointSet

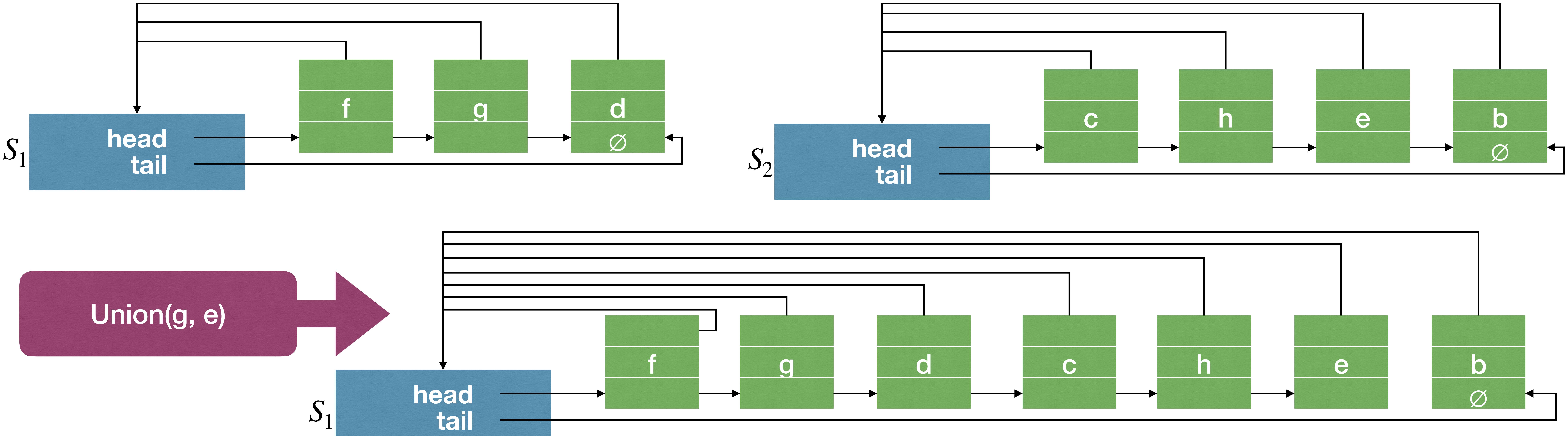
- Basic Idea: Use a linked list to store and represent a set.
- **`MakeSet(x)`**: Create a linked list containing only x. $\rightarrow \Theta(1)$
- **`Find(x)`**: Follow pointer from x back to the set object, then return pointer to the first element in the linked-list. $\rightarrow \Theta(1)$



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; - - - - $\Theta(1)$
  update set object pointers for elements originally in S_y. - - - - - Time depends on size of S_y.

Union can be slow, even in amortized sense!



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!
- **Complication**: Need to maintain size for each set (but this is easy).

```
MakeSet(x0)
for i := 1 to n
    MakeSet(xi)
    Union(xi, x0)
```

- Complexity of this sequence of operations?
  - $\Theta(n)$ in total
  - $\Theta(1)$ on average.

Worst complexity of **any** sequence of $n+1$ ***MakeSet*** and then n ***Union***?

Linked-list implementation of DisjointSet

- Worst complexity of **any** sequence of $n+1$ ***MakeSet*** and then n ***Union***?
    - $O(n \lg n)$
- **Proof**:
    - The $n+1$ ***MakeSet*** op. take $O(n)$ time in total.
    - For ***Union*** op., cost dominated by set obj. pointer changes.
    - For each element, whenever its set obj. pointer changes, its set size **at least doubles**!
    - Each element's set obj. pointer changes $O(\lg n)$ times
    - Therefore, the Union op. take $O(n \lg n)$ time in total (there are $n+1$ elements).

"Average" cost of Union operation is reduced to $O(\lg n)$.

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 root of x to the root of y.

Time complexity depends on depth of x and y

Set $\{c, h, e, b\}$ with leader c

Set $\{f, d, g\}$ with leader f

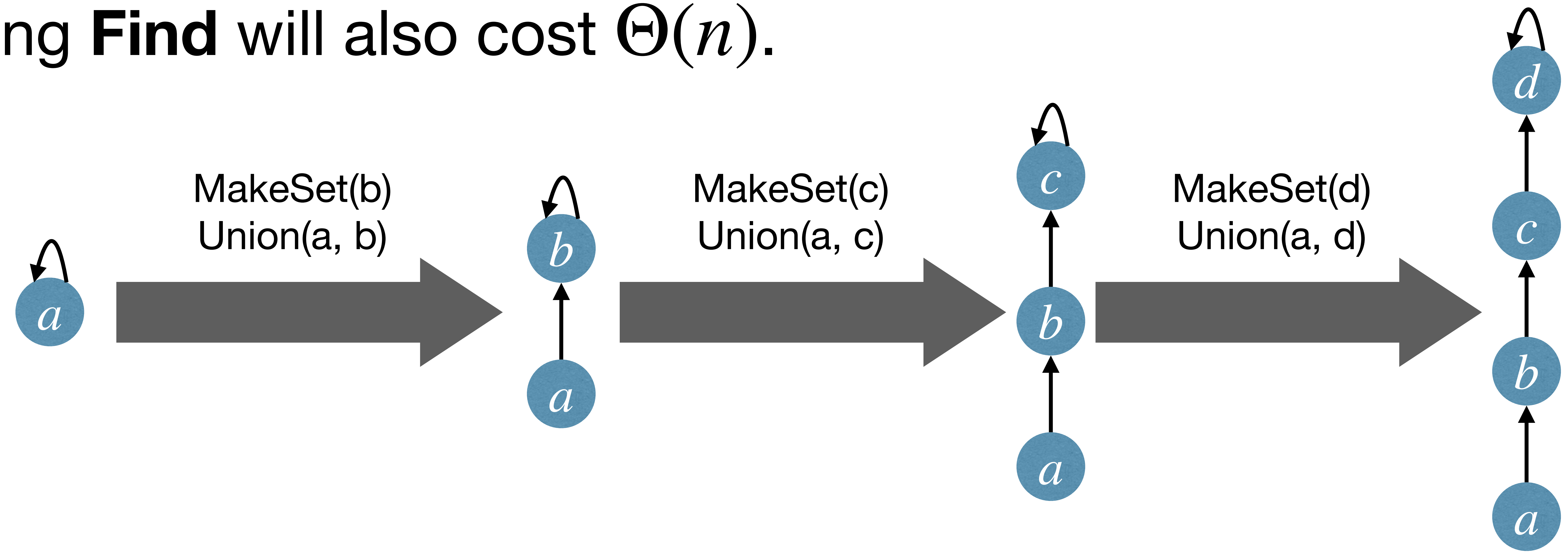
Linked-list vs Rooted-tree Implementation

- **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.

Rooted-tree implementation of DisjointSet

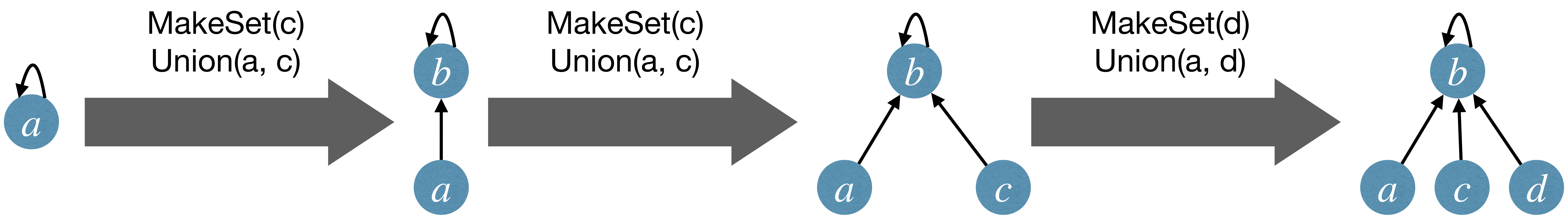
- **Worst case:** A sequence of n **Union** can cost $\Theta(n)$ on average; Many following **Find** will also cost $\Theta(n)$.



Rooted-tree implementation of DisjointSet

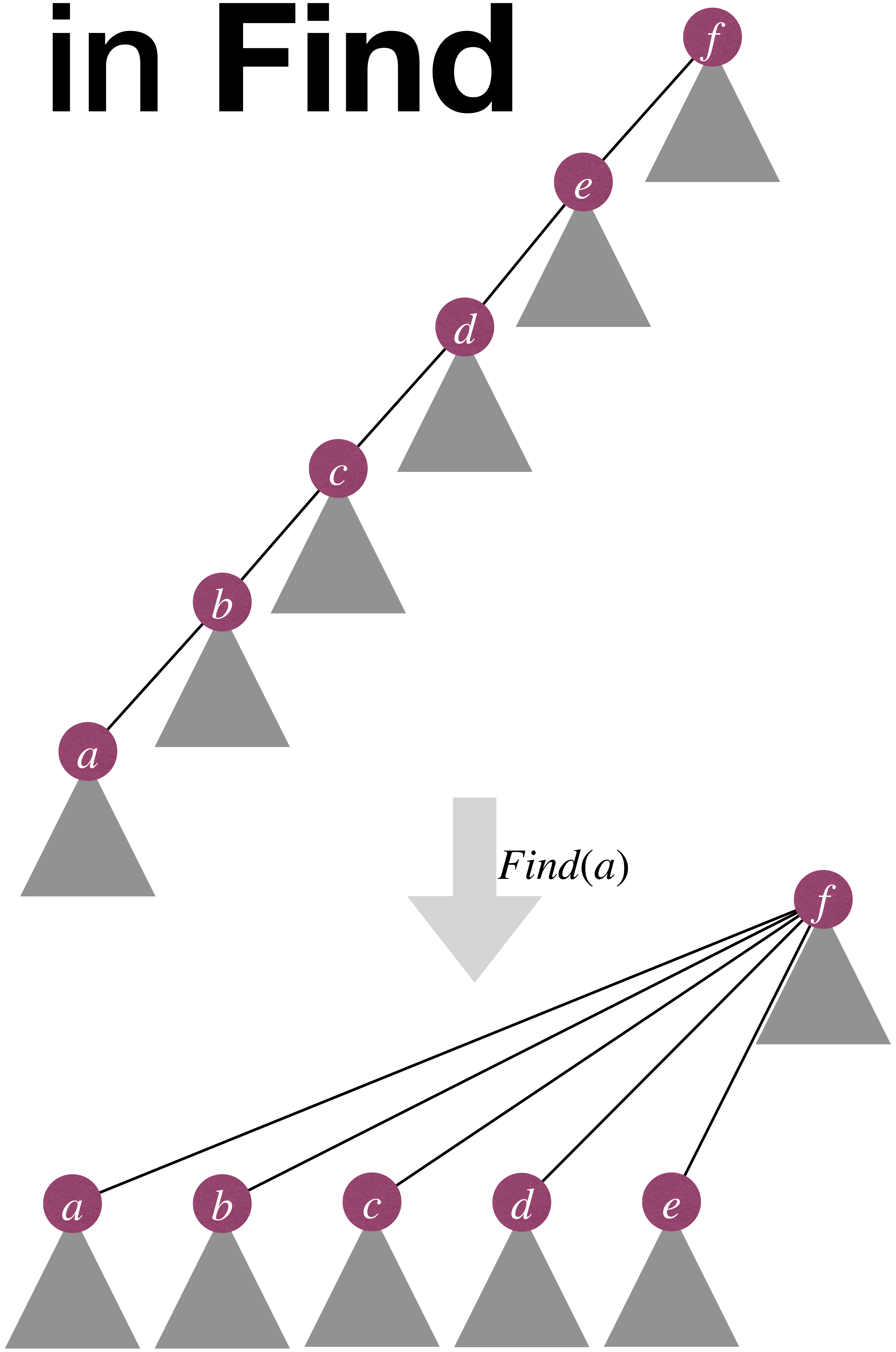
- Again, use **union-by-size** heuristic; reduce **worst-case** cost of **Union** and **Find** to $O(\lg n)$.
    - Each time a node's depth increases, the tree size at least doubles. So size n tree has height $O(\lg n)$.
- Alternatively, use **union-by-height** heuristic: In **Union**, let tree of smaller height be subtree of larger height.
    - **Union-by-height** reduces **worst-case** cost of **Union** and **Find** to $O(\lg n)$.
        - Easy proof via induction: height h tree has $\geq 2^h$ nodes.

can we do better?



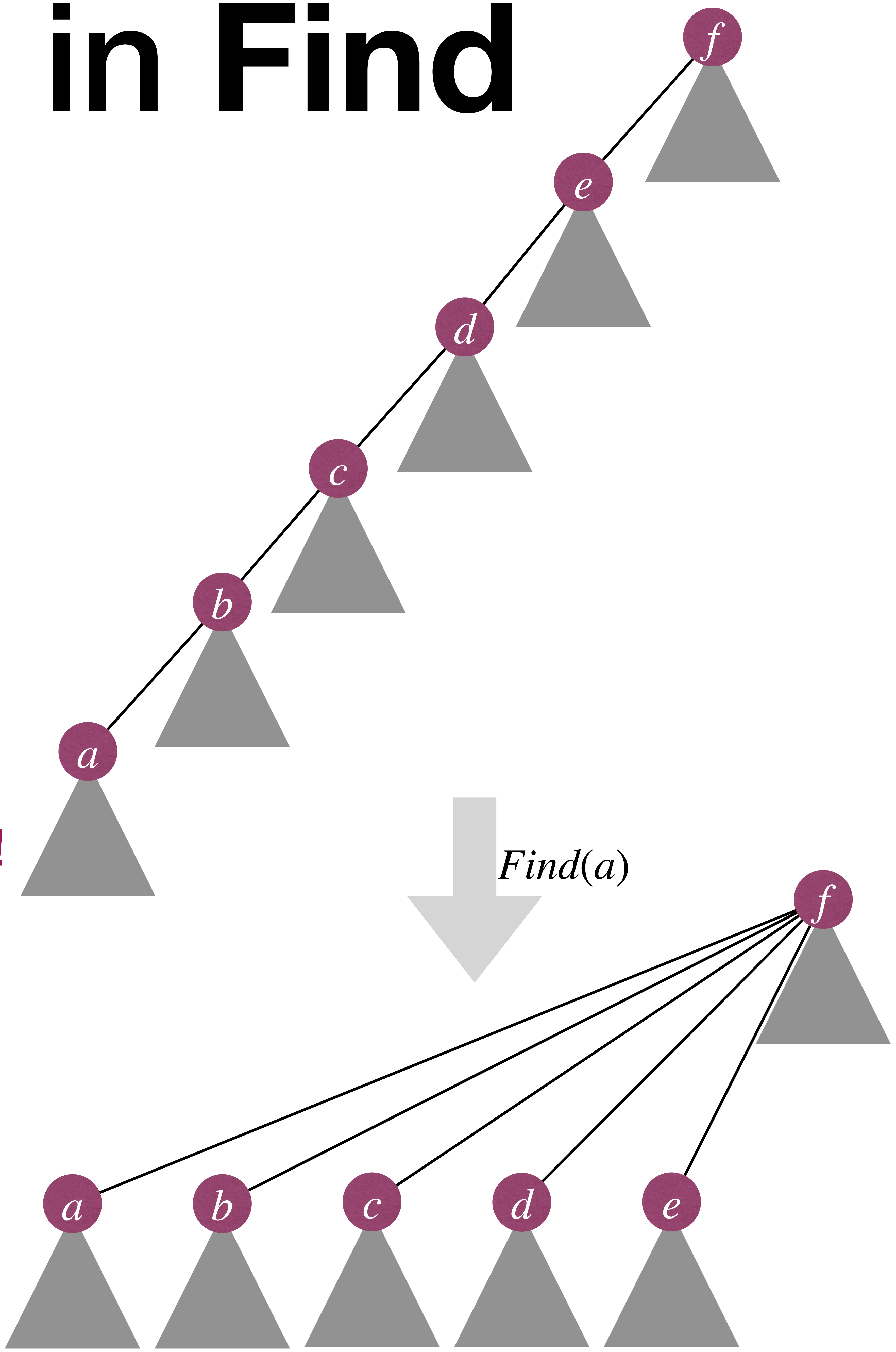
Path-compression in Find

- `MakeSet(x)`: Create a tree containing only (root) x; parent of x is x. Height of the tree is set to 0.
- `Find(x)`: Follow parent pointer from x back to the root, and 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.
- Do some work in **Find** to speed-up future **Find**, without increasing asymptotic cost of **Find**.
  - *Path-Compression*: In `Find(x)`, when traveling from x to root r_x, make all nodes on this path directly points to root r_x.
    - Path-compression will not increase cost of Find asymptotically.



Path-compression in Find

- `MakeSet(x)`: Create a tree containing only (root) x; parent of x is x. Rank of the tree is set to 0.
- `Find(x)` [*path-compression*]: Follow parent pointer from x back to the root; let nodes along the path directly point to root; at last, return root.
- `Union(x,y)` [*union-by-rank*]: Change the parent of the root of the lower-rank tree to the root of the higher-rank tree. Increase rank if necessary.
- Find can now change **heights**! Maintaining heights becomes expensive!
- **Simple Solution**: just ignore the impact on "height" when doing path compression.
  - In such case, the "height" is referred to "**rank**".
  - Rank is always an upper bound of height.



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 the root; let nodes along the path directly point to root; at last, 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.

How efficient is the implementation of DisjointSet?
MakeSet is $O(1)$, ***Find*** and ***Union***?

Almost $O(1)$

*Performance analysis for rooted-tree implementation

with union-by-rank and path-compression

Slowly Growing Functions

- Consider the recurrence
  - $C(N) = \begin{cases} 0 & N \leq 1 \\ C(\lfloor f(N) \rfloor) + 1 & N > 1 \end{cases}$
  - In this equation, $C(N)$ represents the number of times, starting at N, that we must iteratively apply $f(N)$ until we reach 1 (or less).
  - We assume that $f(N)$ is a nicely defined function that reduces N. Call the solution to the equation $f^*(N)$.
    - When $f(N) = N - 2, f^*(N) = N/2$
    - When $f(N) = N/2,\ f^*(N) = \log N$
    - When $f(N) = \log N, f^*(N) = \log^* N$, this function grows extremely slow (e.g., $\log^* 2^{65536} = 5$.)

*Performance Analysis

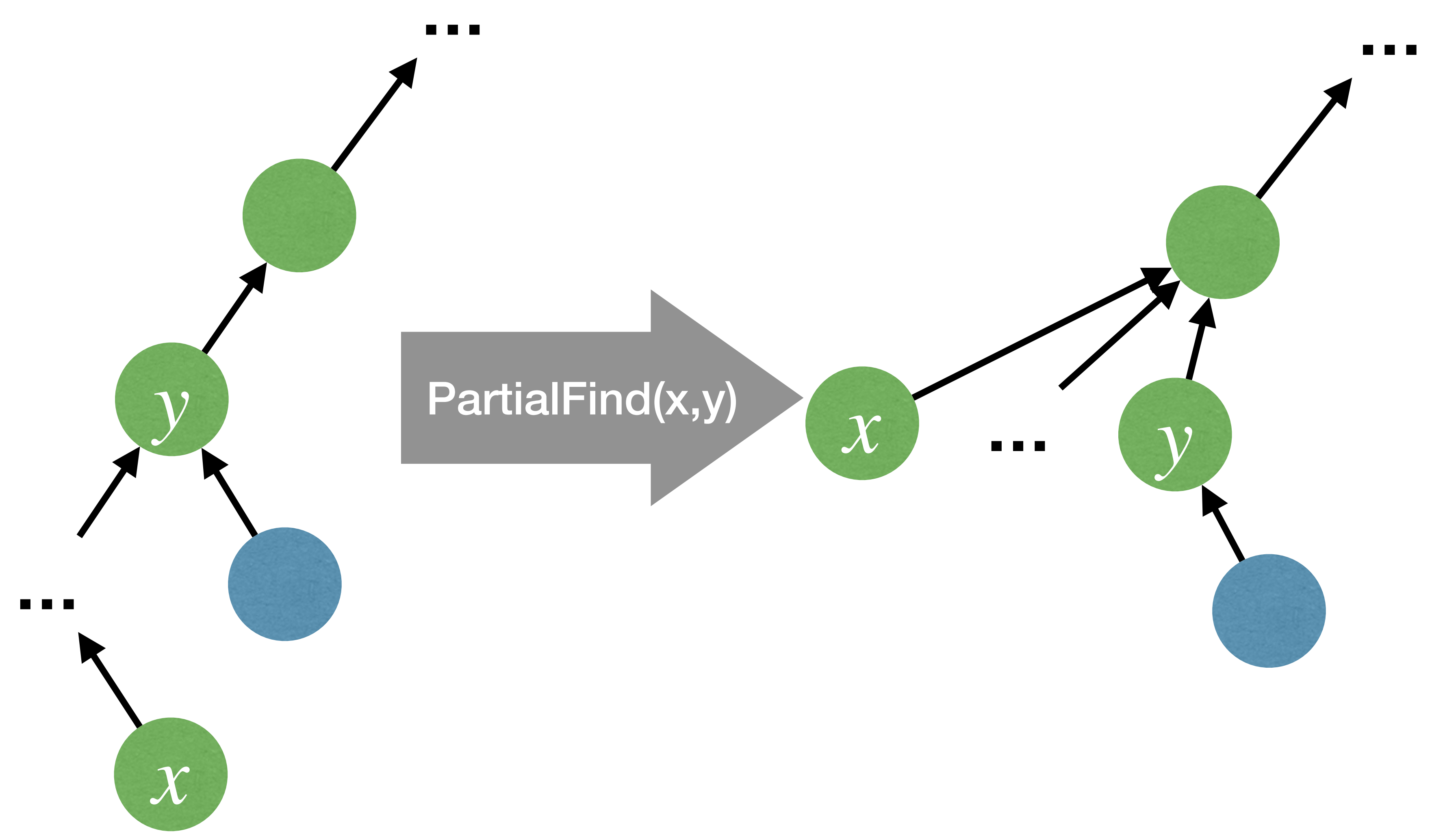
- **Goal**: Any sequence of ***MakeSet***, ***Find***, ***Union*** operations has low average cost.
- **Observation**:
  - (a) ***MakeSet*** can be moved to the beginning of operation sequence, without affecting the cost.
  - (b) ***MakeSet*** itself has low cost.
- **New Goal**: Starting from a forest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

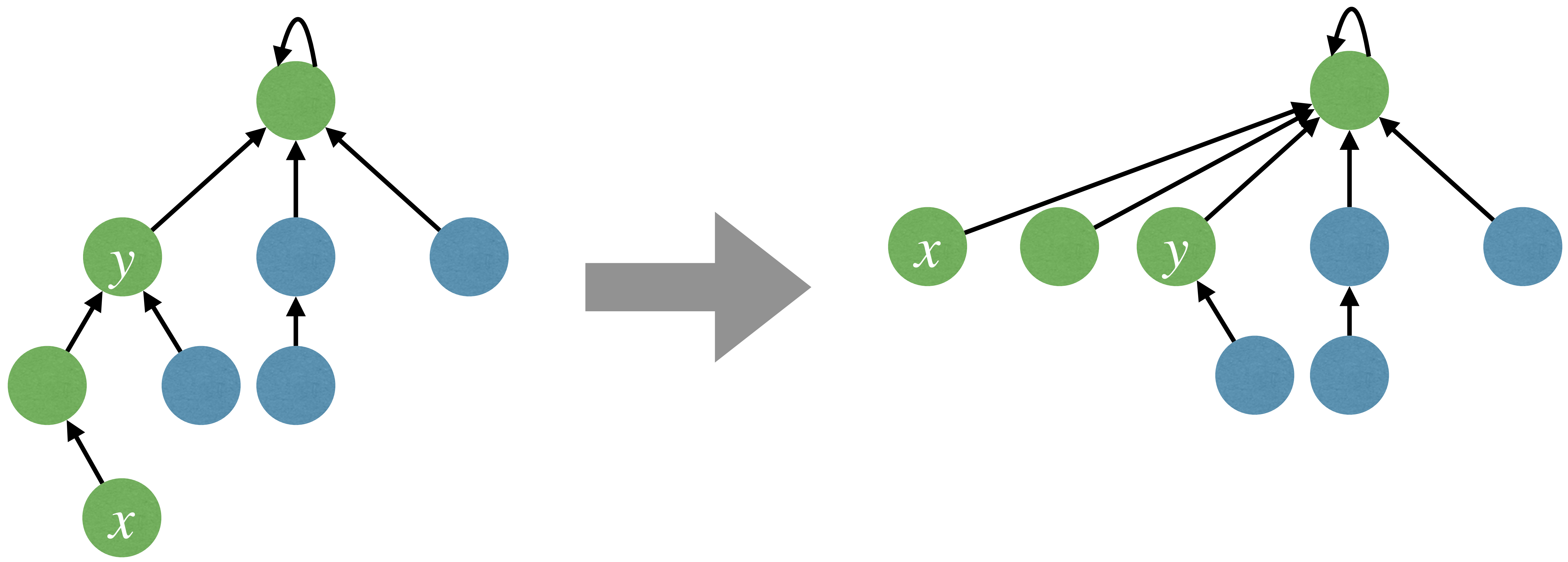
Performance Analysis

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

Performance Analysis

- `Find(x)` [*path-compression*]: Follow parent pointer from x back to the root; let nodes along the path directly point to root; at last, 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; at last, return parent of y.





Every ***Find*** operation can be replaced by a ***PartialFind*** operation.

Performance Analysis

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

- **Observation**: Every ***Find*** operation can be replaced by a ***PartialFind*** operation.

- **New Goal**: Starting from a forest containing n nodes, any sequence of ***PartialFind*** and ***Union*** operations has low average cost, in which input parameters to ***Union*** operation are always set leaders.

Performance Analysis

Performance Analysis

- **Goal**: Starting from a forest containing n nodes, any sequence of ***PartialFind*** and ***Union*** operations has low average cost, in which input parameters to ***Union*** operation are always set leaders.

- **Observation**: We can push all ***Union*** operation to the beginning.

  - Relative order among all ***Union*** operation is preserved.

- **New Goal**: Starting from a forest containing n nodes, any sequence of ***PartialFind*** and ***Union*** operations has low average cost, in which every ***Union*** occurs before any ***PartialFind***, and input parameters to ***Union*** operation are always set leaders.

Performance Analysis

- **Goal**: Starting from a forest containing n nodes, any sequence of ***PartialFind*** and ***Union*** operations has low average cost, in which every ***Union*** occurs before any ***PartialFind***, and input parameters to ***Union*** operation are always set leaders.
- **Observation**: Each ***Union*** operation only costs $O(1)$.
- **New goal:** Starting from a forest containing n nodes, any sequence of m ***PartialFind*** operations has low average cost.
- **Observation**: Cost of ***PartialFind*** is **dominated** by pointer assignments (that is, the number of **parent changes**).
- **New goal:** Starting from a forest containing n nodes, any sequence of m ***PartialFind*** operations has low total pointer assignments.

Performance Analysis

- **Goal:** Starting from a forest containing n nodes, any sequence of m ***PartialFind*** operations has low total pointer assignments.

- $T(m, n, r)$: **worst number of pointer assignments** in any sequence of m ***PartialFind***, starting from a size n forest where each node has rank at most r.

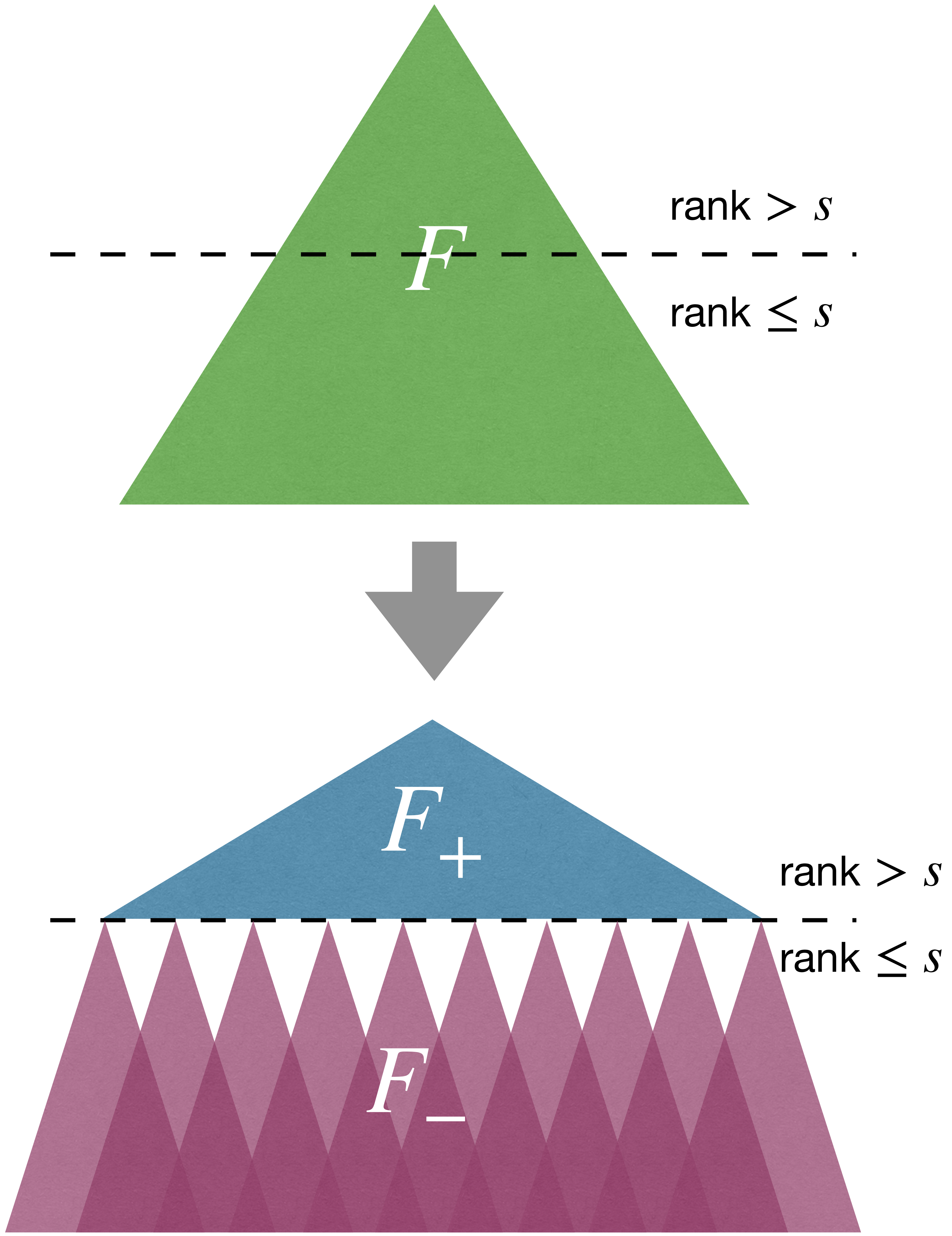
- **Goal**: $T(m, n, r)$ is small.

- **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.

Performance Analysis

- Fix forest F of n nodes with max rank r, and a sequence C of m **PartialFind** on F.
- $T'(F, C)$: total number 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 $\leq s$.
- Let $|F_+| = n_+$, and $|F_-| = n_-$
- m_+: number of operations in C that involve any node in F_+.
- m_- : $m - m_+$

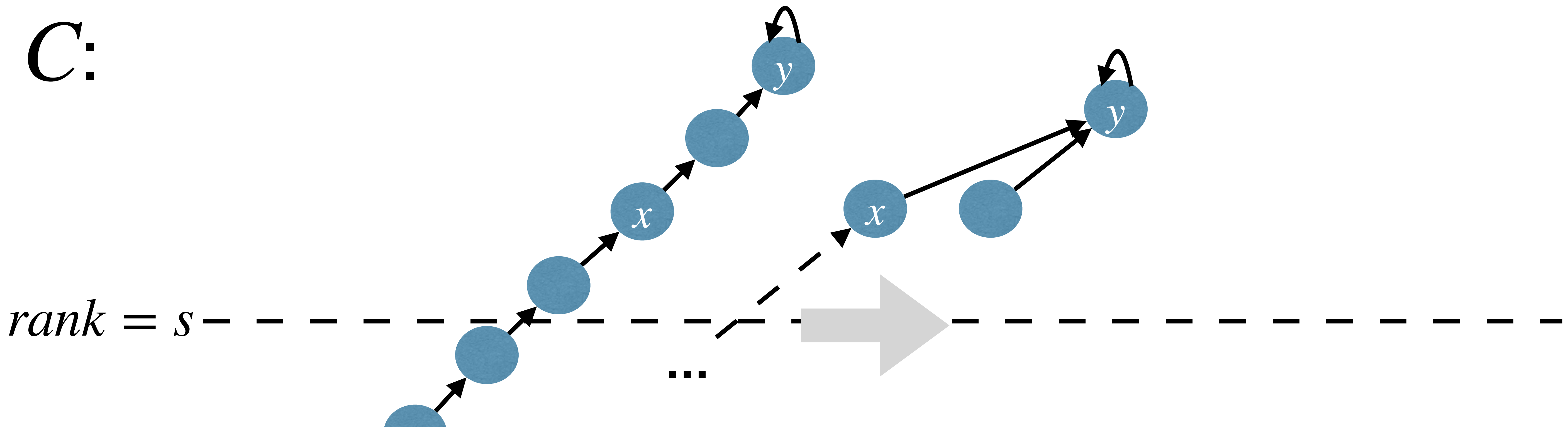


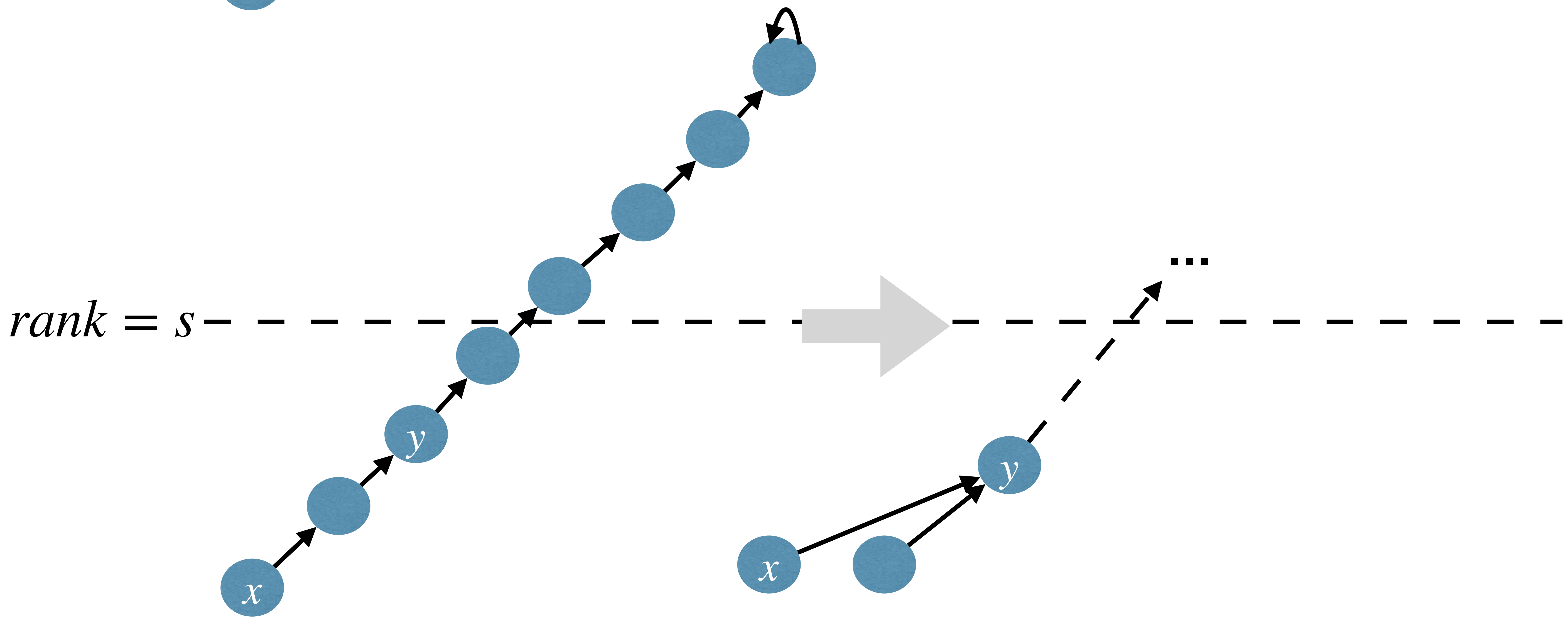
Performance Analysis

- Consider a `PartialFind(x, y)` in C:

- If $rank(x) > s$: the operation is a ***PartialFind*** operation in F_+.

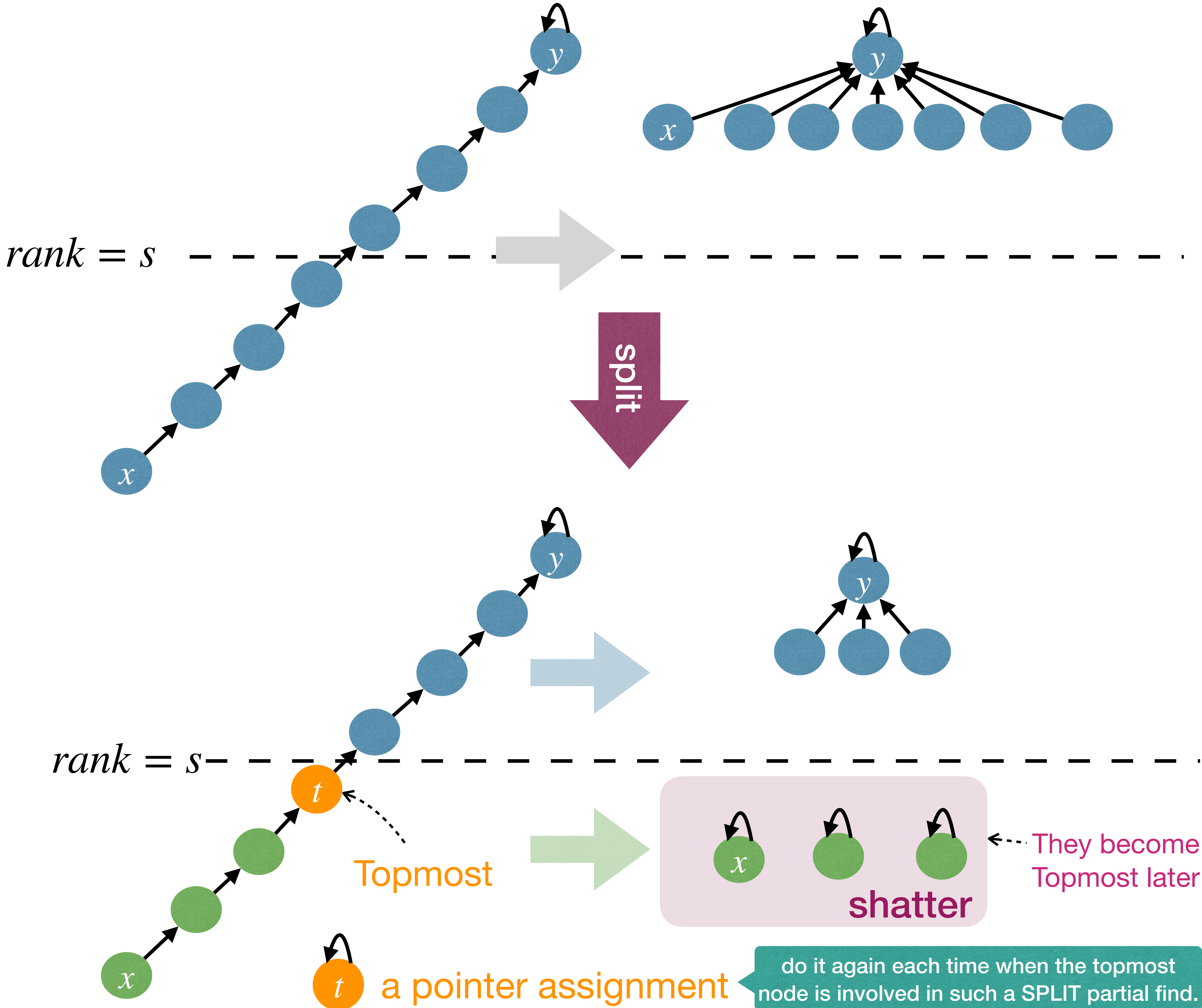
- If $rank(y) \leq s$: the operation is a ***PartialFind*** operation in F_-.





Performance Analysis

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

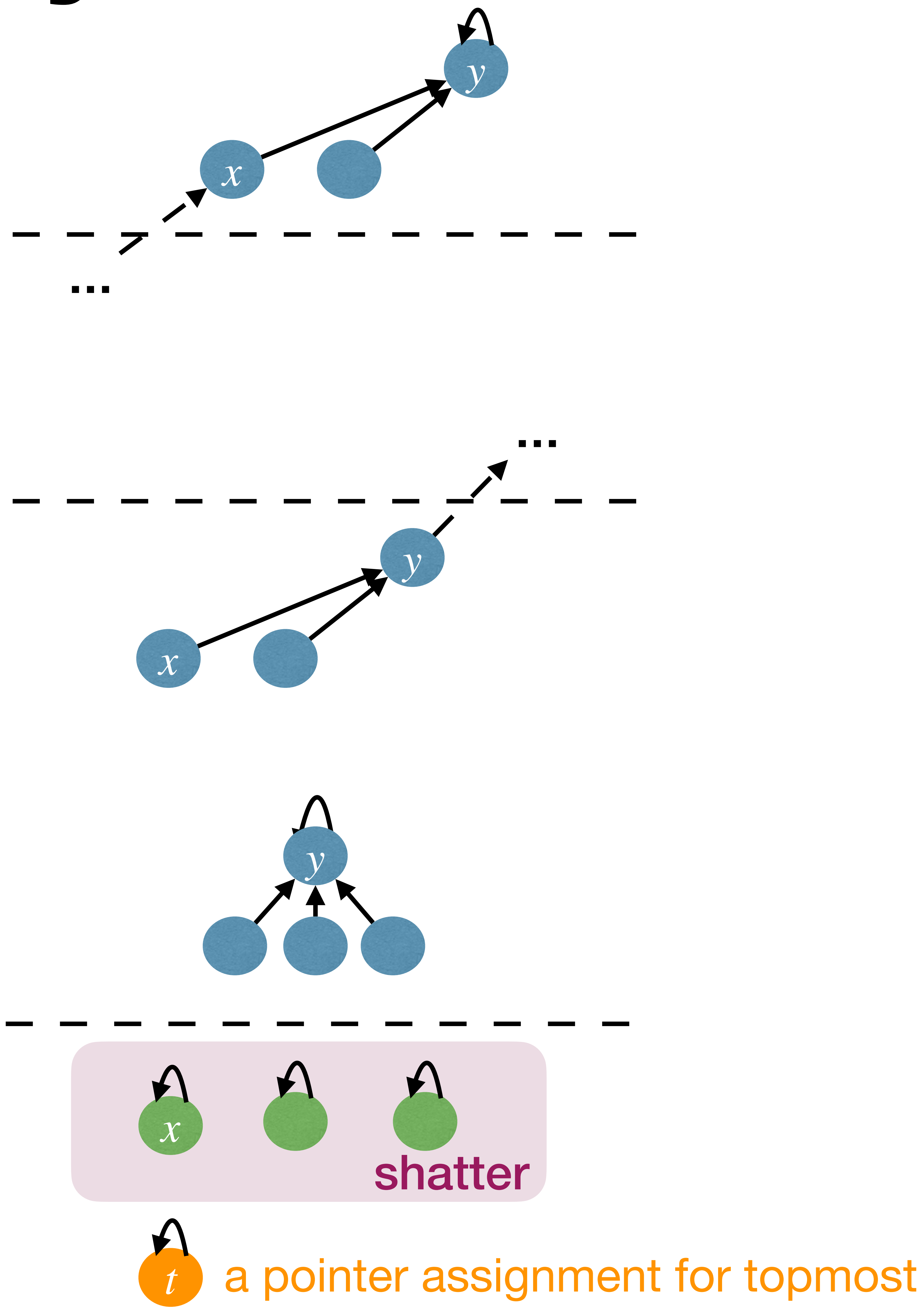


Performance Analysis

- We have converted C into:
  - (a) C_+: ops involving nodes only in F_+;
  - (b) C_-: ops involving nodes only in F_-;
  - (c) shatter operations; and
  - (d) pointer assignments for "topmost" nodes in F_-.
- **Observations**:
  - Each node get shattered at most once (then be "topmost" node in F_-).
  - There are at most m_+ pointer assignments for "topmost" nodes in F_-.

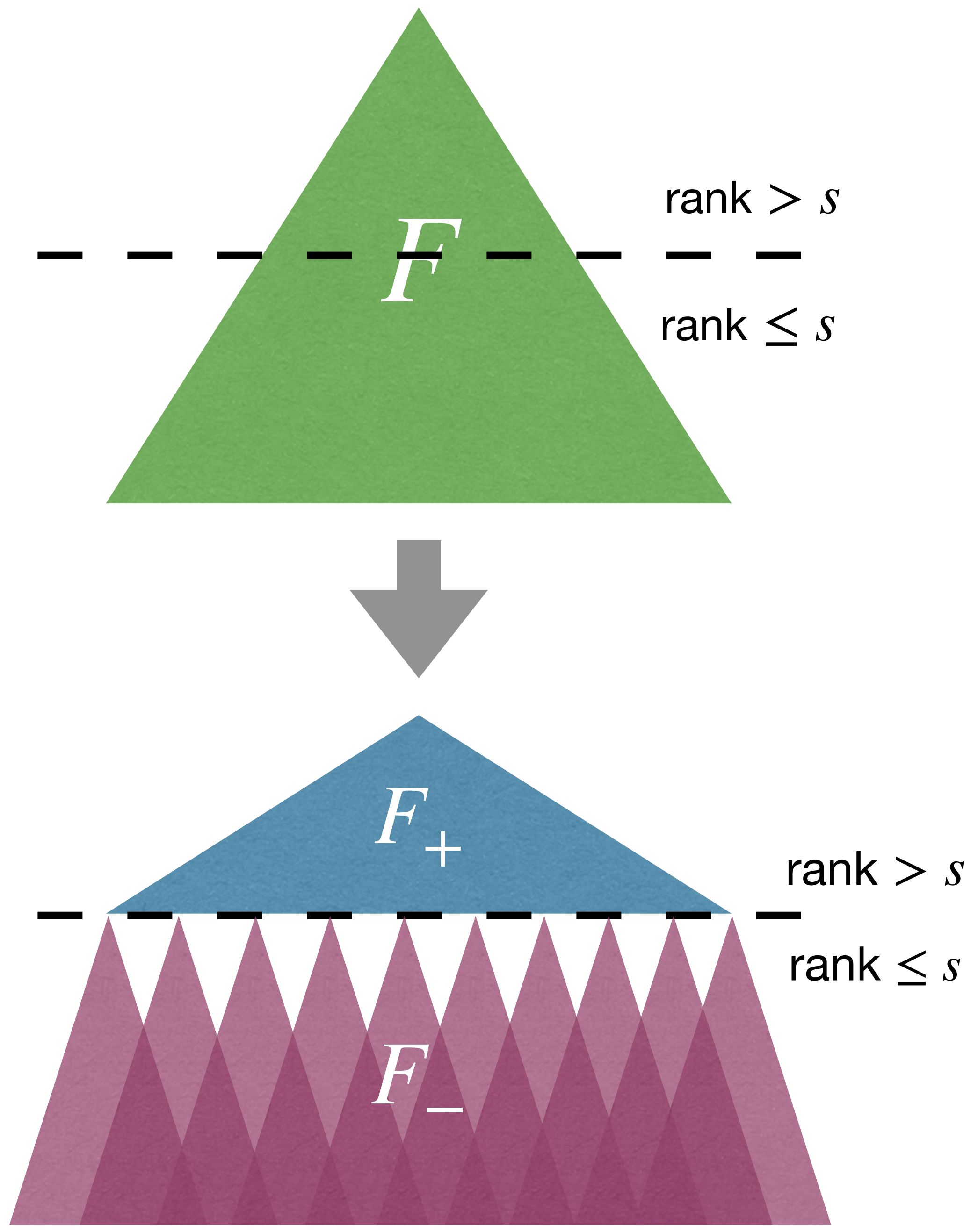
#operations in C that involve any node in F_+.

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



Performance Analysis

- $T'(F, C) \leq T'(F_+, C_+) + T'(F_-, C_-) + n + m_+$
  - 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 $T'(F, C)$.
  - **Previous Claim**: $T(m, n, r) \leq nr$.
  - Recall that $T(m, n, r)$ is the worst number of pointer assignment in any sequence of ***PartialFind***, starting from a size n forest where each node has rank at most r.



Performance Analysis

- **Claim**: $T(m, n, r) \leq nr.$
- **Claim**: There are at most $n/2^i$ nodes of rank i in any size n forest.

Note: one rank i tree has $\geq 2^i$ nodes (by induction)

  - $T'(F_+, C_+) \leq n_+ \cdot r \leq \left(\sum_{i>s} \frac{n}{2^i} \right) \cdot r = \frac{nr}{2^s}$
  - Fix $s = \lg r$, then $T'(F, C) \leq T'(F_-, C_-) + 2n + m_+$, or equivalently $T'(F, C) - m \leq \left(T'(F_-, C_-) - m_- \right) + 2n$
  - $T''(m, n, r) \leq T''(m, n, \lg r) + 2n$, where $T''(m, n, r) = T(m, n, r) - m$
  - $T''(m, n, r) \leq 2n \lg^* r$. That is: $T(m, n, r) \leq m + 2n \lg^* r$

Actual performance is even better!

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

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 parameters 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 $O(\lg^* n)$.

In amortized sense!

Further reading

- [CLRS] Ch.21(excluding 21.4)
- [Weiss] Ch.8 (8.6)
- Lecture notes by Jeff Erickson
  - http://jeffe.cs.illinois.edu/teaching/algorithms/notes/11-unionfind.pdf