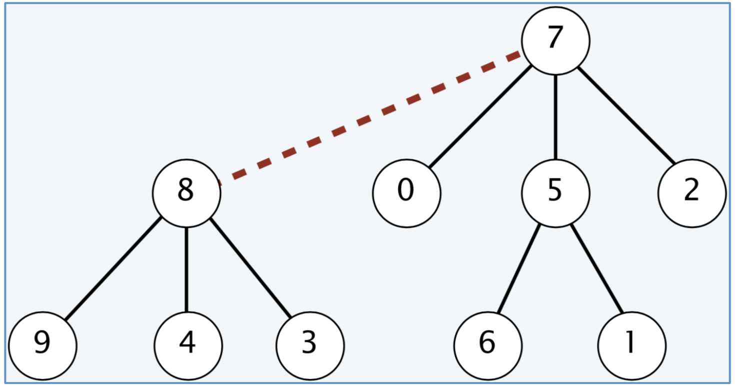
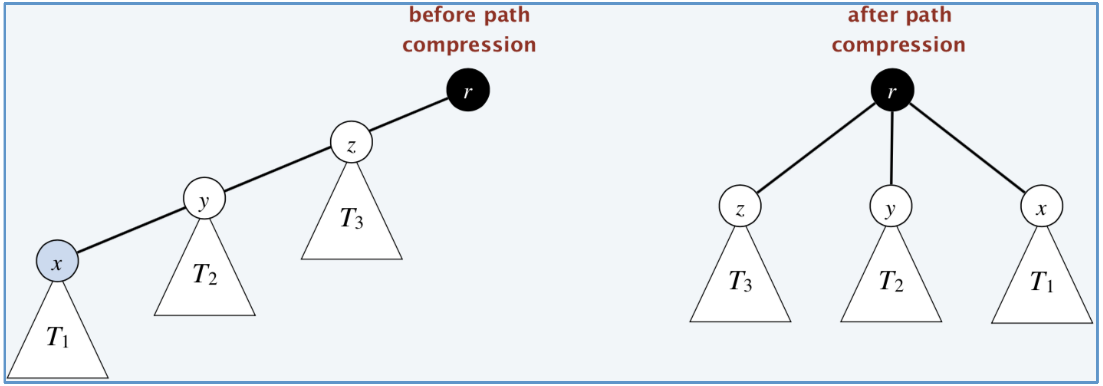
Disjoint Set Data Structure



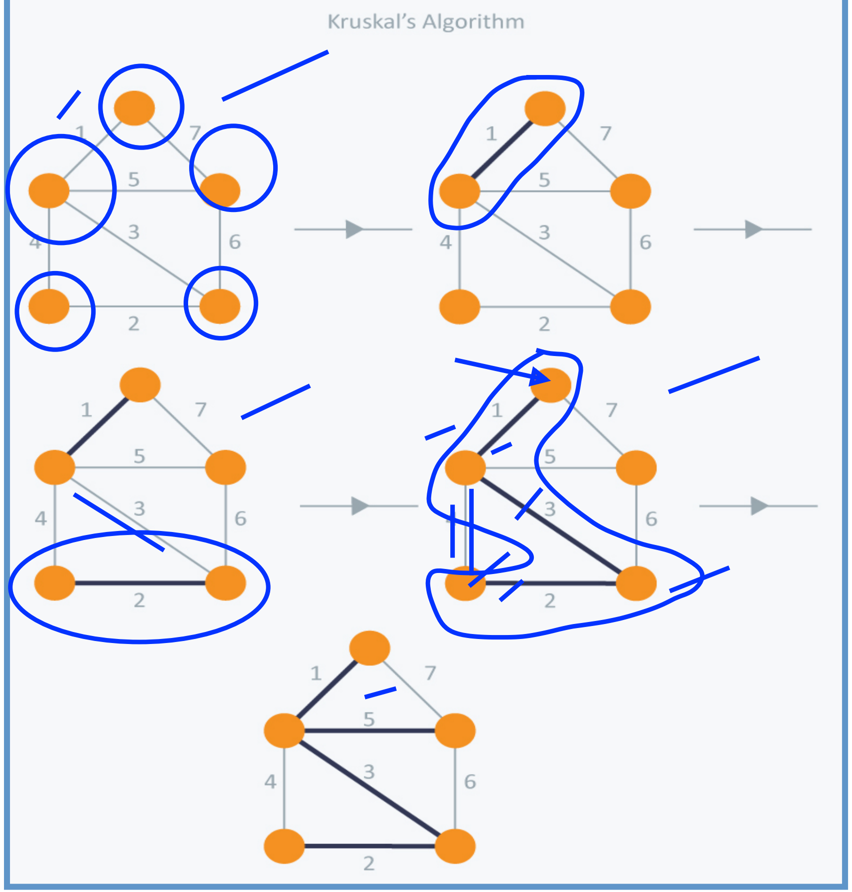


Partha P. Chakrabarti & Aritra Hazra
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

Disjoint-Set Data Structures: *Applications*

Minimum Spanning Tree of Graph (G)

```
Algorithm MST_Kruskal ( G = (V,E) ) {
    A = { };
    for each v in V do MAKE-SET(v);
    for each edge e = (u, v) in E ordered by
    increasing weight(u, v) do {
        if FIND-SET(u) ≠ FIND-SET(v) then {
            A = A + {(u, v)};
            UNION(FIND-SET(u), FIND-SET(v));
        }
    }
    return A;
}
```



Disjoint-Set Data-Type and Operations

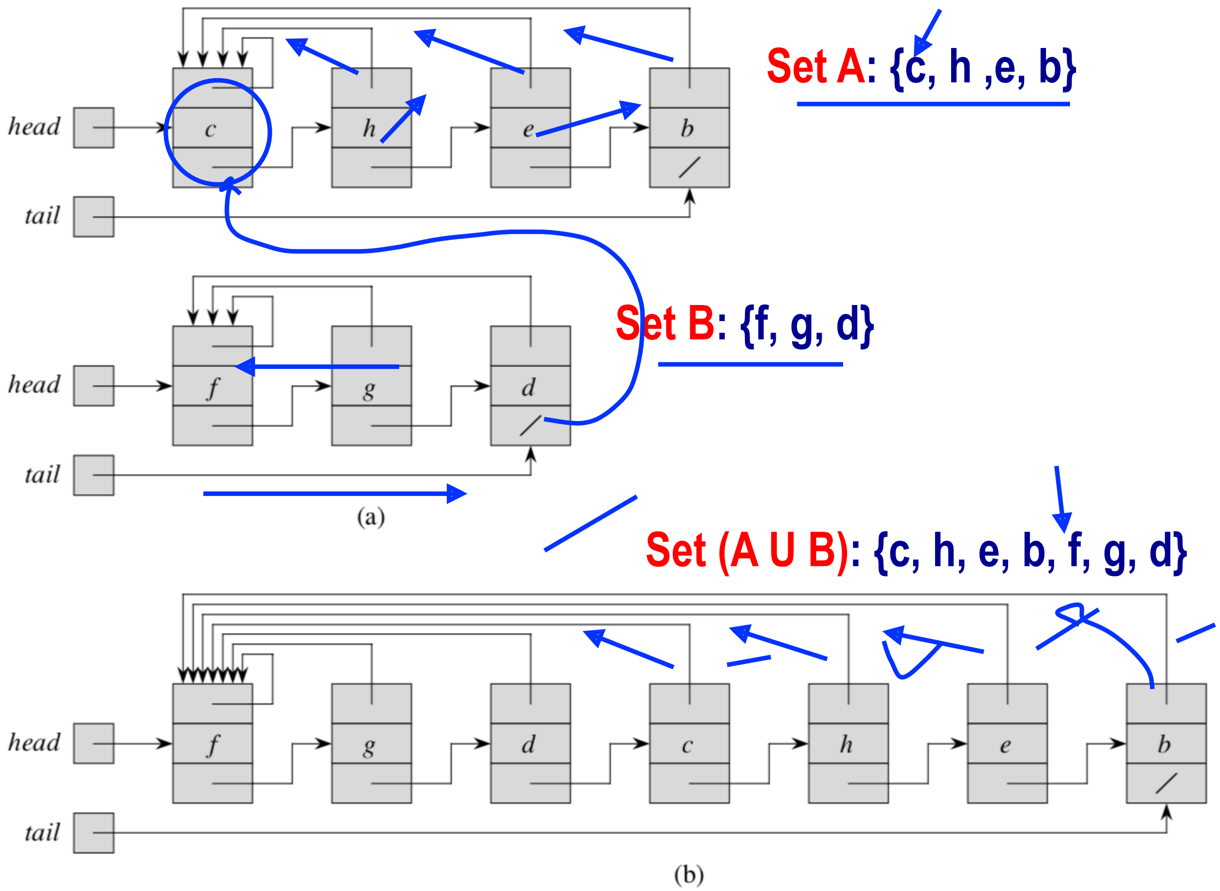
- **Primary Operations:**
  - MAKE-SET(*x*): create a new set containing only element *x*
  - FIND-SET(*x*): return a canonical element in the set containing *x*
  - UNION(*x*, *y*): replace the sets containing *x* and *y* with their union
- **Performance parameters:**
  - *m* = number of calls to FIND-SET and UNION operations
  - *n* = number of elements = number of calls to MAKE-SET
- **Application: Dynamic connectivity** over initially empty graph
  - ADD-NODE(*u*): add node *u* *(1 MAKE-SET operation)*
  - ADD-EDGE(*u*, *v*): add an edge between nodes *u* and v *(1 UNION operation)*
  - IS-CONNECTED(*u*, *v*): is there a path between *u* and *v* ? *(2 FIND-SET operations)*

disjoint sets = connected components

Disjoint-Set Operations: *Implementation* (1)

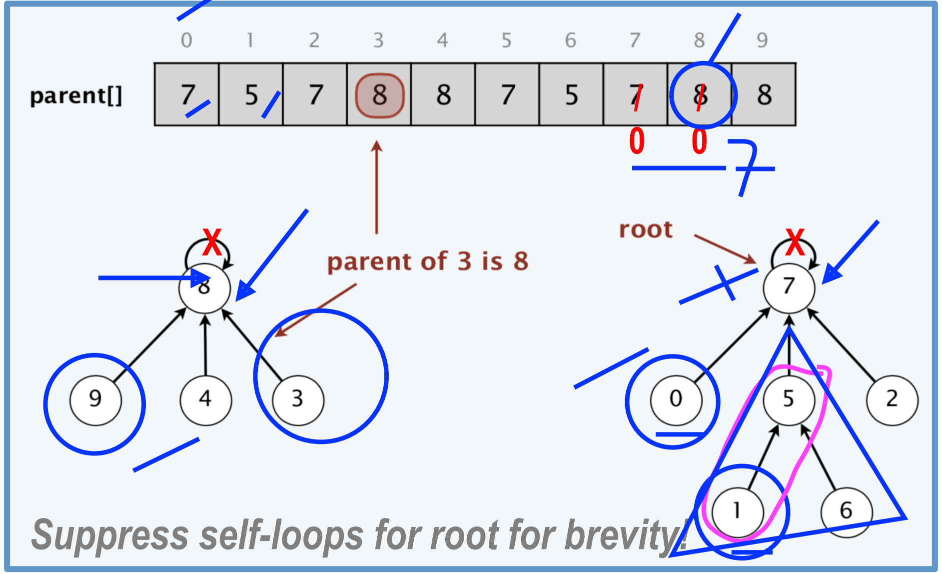
Linked List Implementation

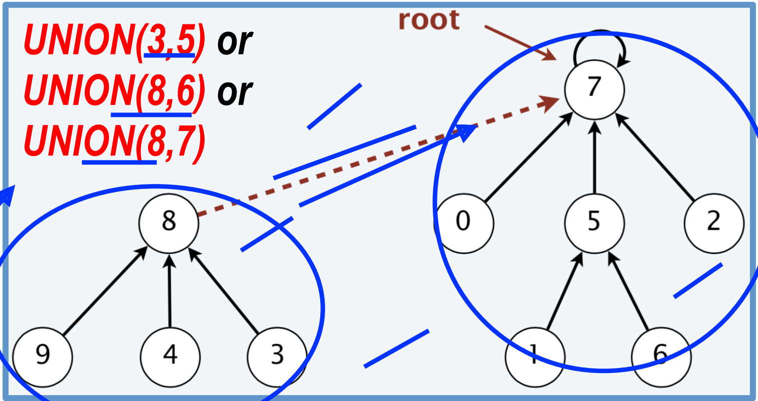
- **MAKE-SET(*x*)**: O(1)
  - need to create only one node created with appropriate pointers
- **FIND-SET(x)**: O(n)
  - need to traverse entire linked list to find x
- **UNION(x,y)**: O(n)
  - need to point back all back-pointers of second list to head of first list



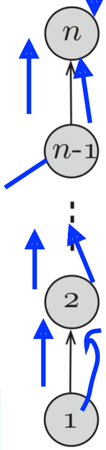
Disjoint-Set Operations: *Implementation* (2)

- **Array Representation**
  - Represent each set as tree of elements
  - Allocate an array of `parent[]` of length n
  - `parent[i]=j` (parent of element `i` is `j`)





- **Analysis of Operations:**
  - Total zeros in array = Disjoint-sets
  - FIND-SET(x): O(n) worst-case
  - UNION(x,y): O(n) worst-case
    - UNION(FIND-SET(x), FIND-SET(y))
    - O(n) due to FIND-SET operation



Solution: **Smart Union-Find Algorithms !!**

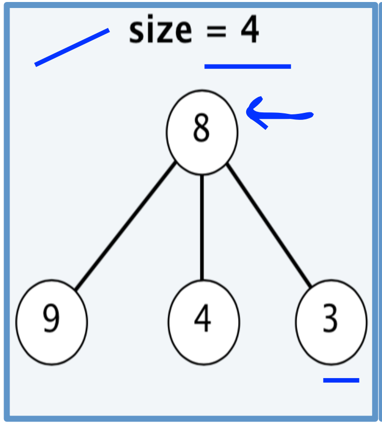
Smart Disjoint-Set Operations: *Union-by-Size*

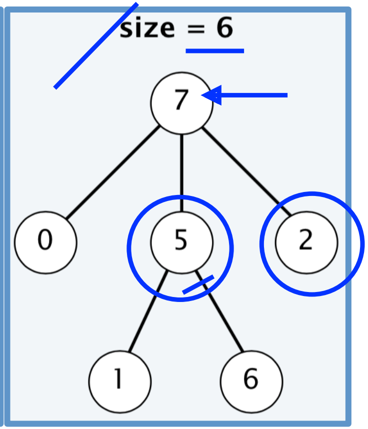
- **Union-by-Size**
  - Maintain a tree size (number of nodes) for each root node
  - Link root of smaller tree to root of larger tree (break tries arbitrarily)

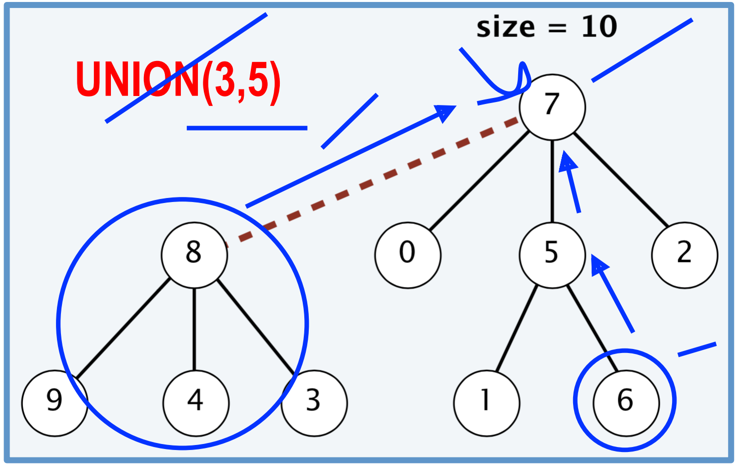
```
FIND-SET(x) {
   while(x is not parent)
      x ← parent[x];
   return x;
}
```

```
MAKE-SET(x) {
   parent[x] ← 0;
   size[x] ← 1;
   return x;
}
```

```
UNION(x,y) {
   r ← FIND-SET(x);
   s ← FIND-SET(y);
   if(r == s) return r;
   else if(size[r] > size[s]) {
      parent[s] ← r;
      size[r] = size[r] + size[s];
      return r;
   }
   else {
      parent[r] ← s;
      size[s] = size[r] + size[s];
      return s;
   }
}
```







Analysis of Union-by-Size Heuristic (1)

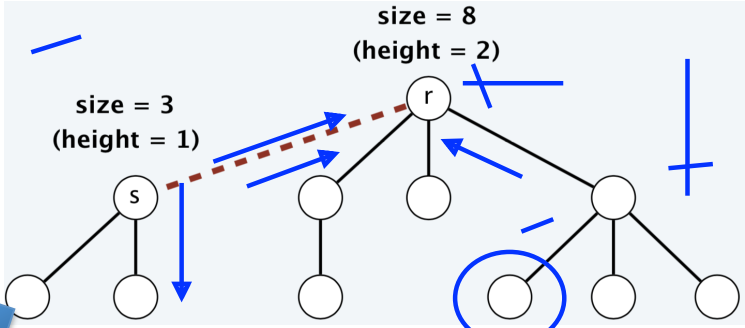
Property: Using union-by-size, for every root node r, *we have* $size[r] \geq 2^{height(r)}$

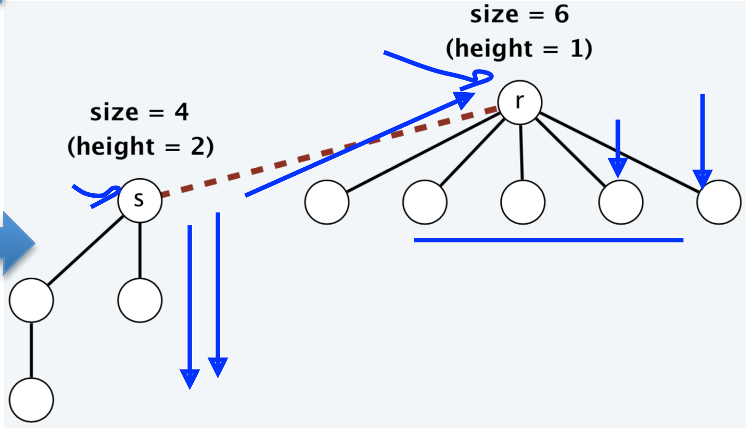
Proof: [by induction on number of links]

- **Base case**: singleton tree has size 1 and height 0
- **Inductive hypothesis**: assume true after first i links
- Tree rooted at r changes only when a smaller (or equal) size tree rooted at s is linked into r
- **Case 1.** [$height(r) > height(s)$]

  $size'[r] > size[r] \geq 2^{height(r)} = 2^{height'(r)}$
- **Case 2.** [$height(r) \leq height(s)$]

  $size'[r] = size[r] + size[s] \geq 2\ size[s] \geq 2 \times 2^{height(s)}$
  $= 2^{height(s)+1} = 2^{height'(r)}$





Analysis of Union-by-Size Heuristic (2)

- **Theorem:** Using union-by-size, any UNION or FIND-SET operation takes $O(\log_2 n)$ time in the worst case, where n is the number of elements
- **Proof:**
  - The running time of each operation is bounded by the tree height
  - Using union-by-size, a tree with n nodes can have height at most $\log_2 n$
  - By the previous property, the height is $\leq \lfloor \log_2 n \rfloor$

- The UNION operation takes $O(1)$ time except for its two calls to FIND-SET
  - FIND-SET required to find out the set representative (which is the root)

- m number of UNION and FIND-SET operations takes a total of $O(m \log_2 n)$ time

Smart Disjoint-Set Operations: *Union-by-Rank*

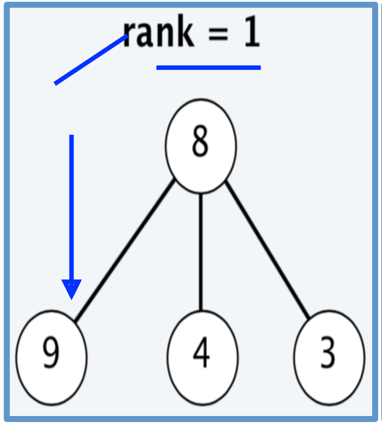
- Union-by-Rank
  - Maintain an integer rank for each node, initially 0
  - Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1

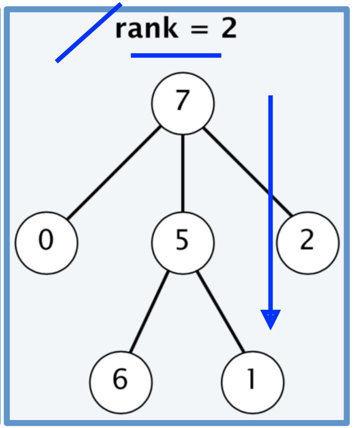
rank = height

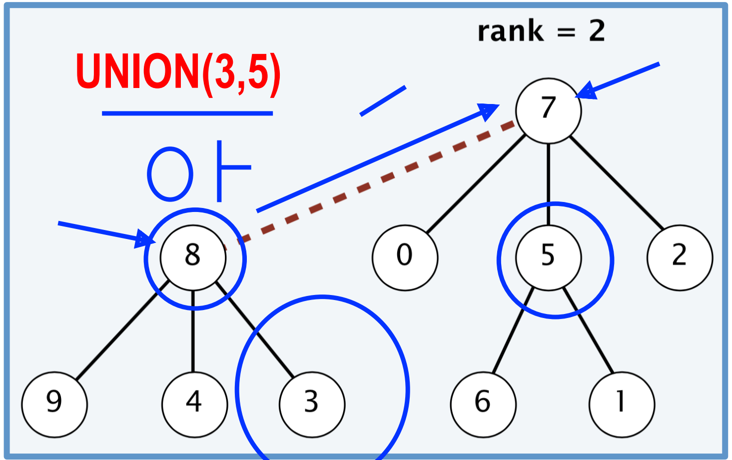
```
FIND-SET(x) {
   while(x is not parent)
      x ← parent[x];
   return x;
}
```

```
MAKE-SET(x) {
   parent[x] ← 0;
   rank[x] ← 0;
   return x;
}
```

```
UNION(x,y) {
   r ← FIND-SET(x);
   s ← FIND-SET(y);
   if (r == s) return r;
   else if (rank[r] ≥ rank[s]) {
      parent[s] ← r;
      if(rank[r] == rank[s])
         rank[r] = rank[r] + 1;
      return r;
   }
   else {
      parent[r] ← s;
      return s;
   }
}
```





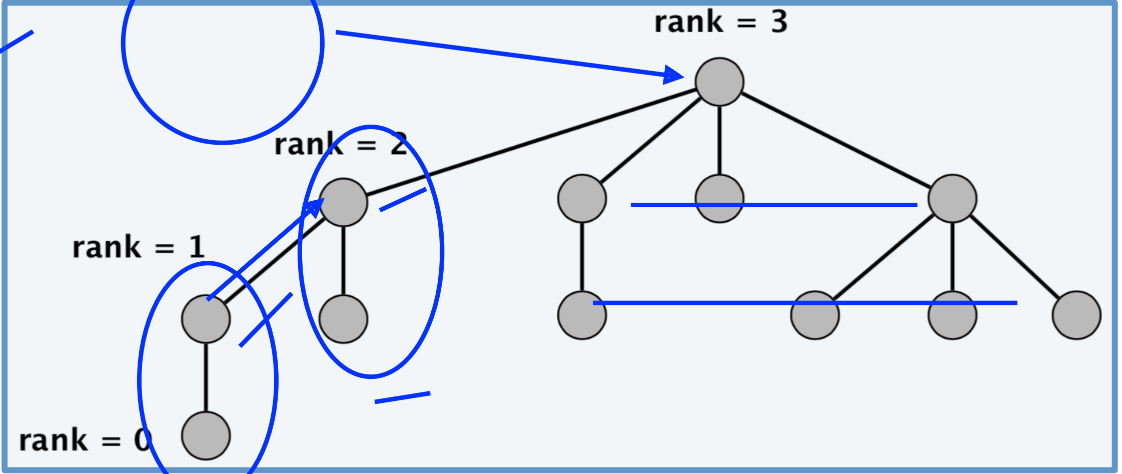


Analysis of Union-by-Rank Heuristic (1)

Property-1: If *x* is not a root node, then *rank*[*x*] < *rank*[*parent*[*x*]]
Proof: A node of rank *k* is created only by linking two roots of rank *k* – 1.

Property-2: If *x* is not a root node, then *rank*[*x*] will never change again
Proof: Rank changes only for roots; a non-root never becomes a root.

Property-3: If *parent*[*x*] changes, then *rank*[*parent*[*x*]] strictly increases.
Proof: The parent can change only for a root, so before linking *parent[x]* = *0*. After *x* is linked using union-by-rank to new root *r* we have *rank*[*r*] > *rank*[*x*].

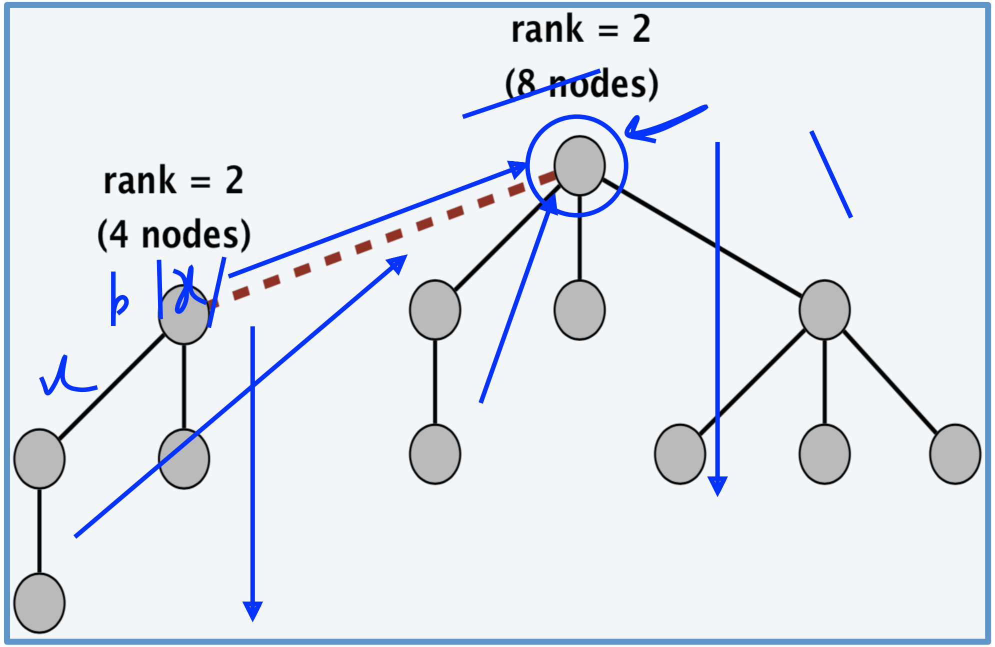


Analysis of Union-by-Rank Heuristic (2)

Property-4: Any root node of rank k has $\geq 2^k$ nodes in its tree

Proof: [by induction on k]

- **Base case:** true for $k = 0$
- **Inductive hypothesis:** assume true for $k - 1$
- A node of rank k is created only by linking two roots of rank $k - 1$
- By inductive hypothesis, each of two sub-tree has $\geq 2^{k-1}$ nodes
  => resulting tree has $\geq 2^k$ nodes



Property-5: The highest rank of a node is $\leq \lfloor \log_2 n \rfloor$

Proof: Immediately concluded from Property-1 and Property-4

Analysis of Union-by-Rank Heuristic (3)

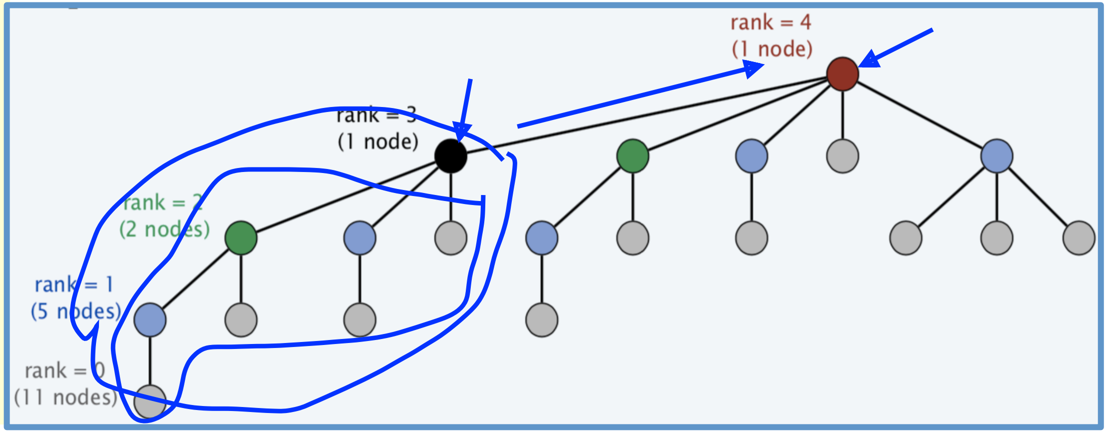
Property-6: For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank k

Proof:

- Any root node of rank k has $\geq 2^k$ descendants. [by Property-4]
- Any non-root node of rank k has $\geq 2^k$ descendants because:
  - it had this property just before it became a non-root [by Property-4]
  - its rank does not change once it became a non-root [by Property-2]
  - its set of descendants does not change once it became a non-root
- Different nodes of rank k cannot have common descendants [by Property-1]

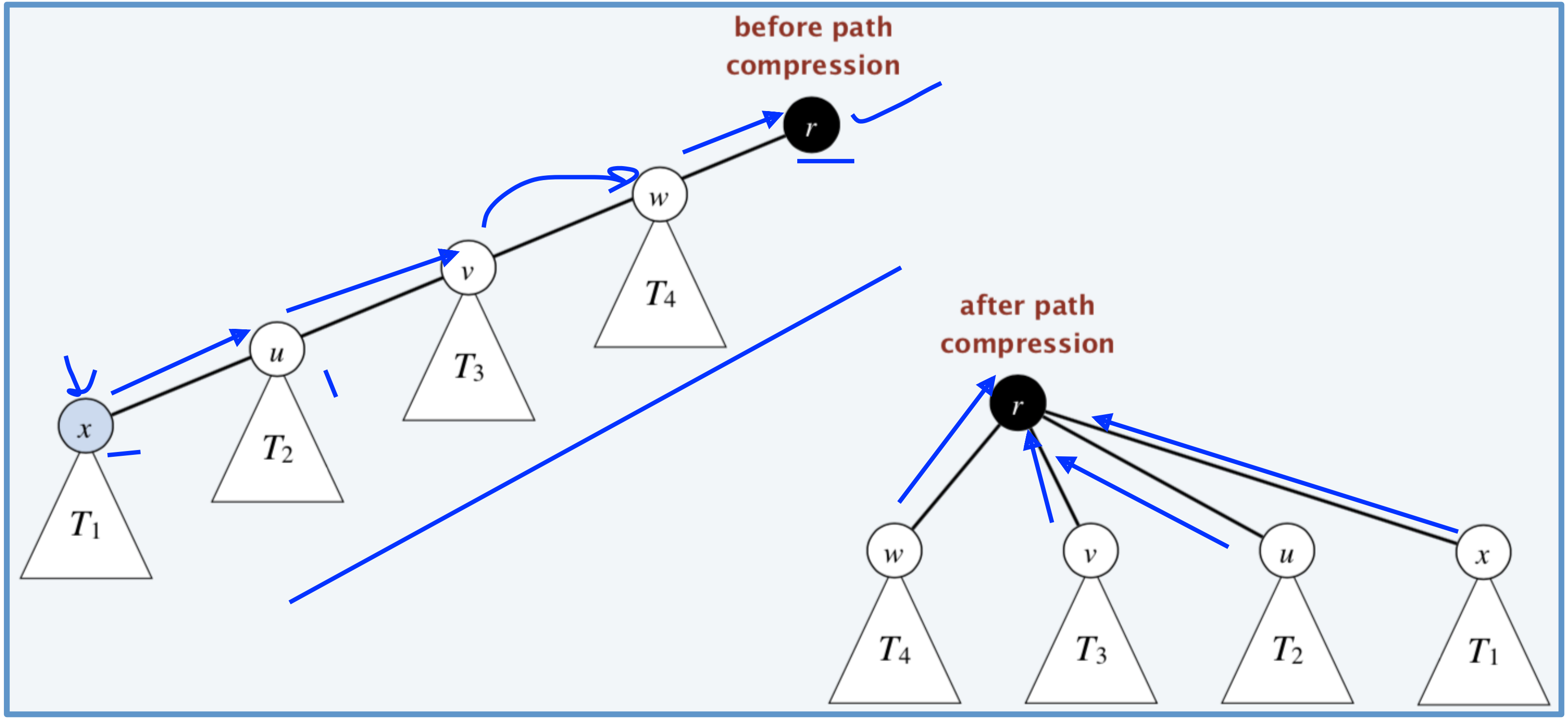
Theorem: Using union-by-rank, any UNION or FIND-SET operation takes $O(\log_2 n)$ time in the worst case, where n is the number of elements.

Proof: The running time of UNION and FIND-SET is bounded by the tree height $\leq \lfloor \log_2 n \rfloor$ [by Property-5]

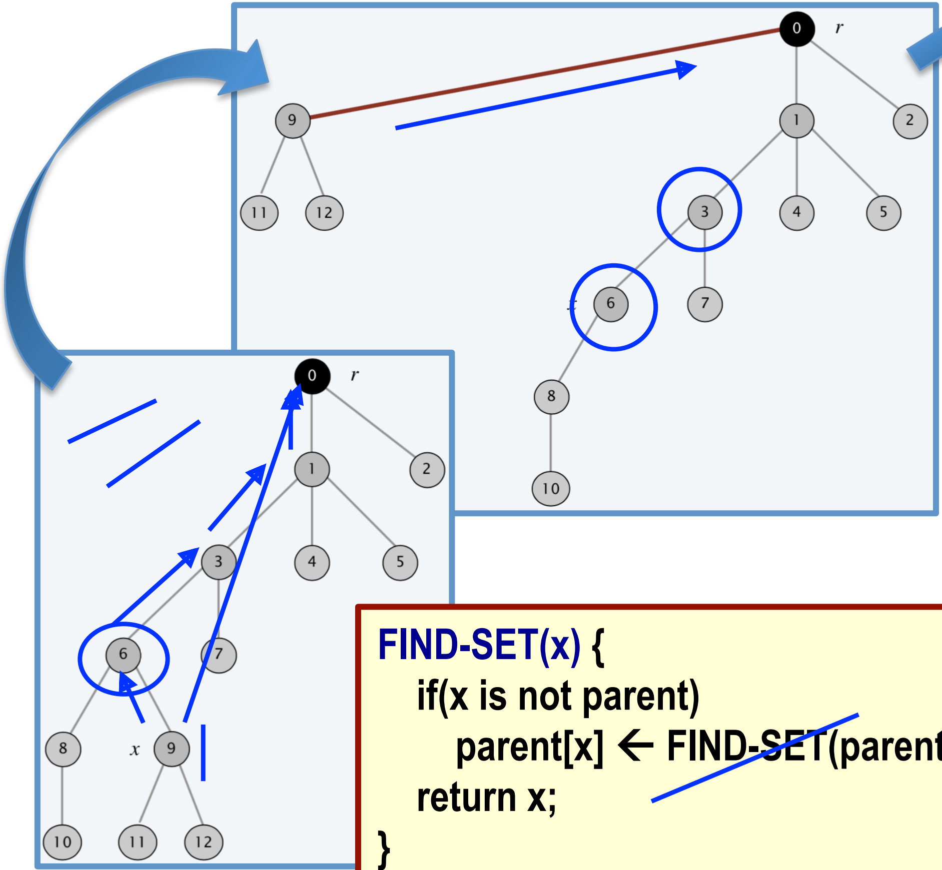


Smart Disjoint-Set Operations: *Path Compression*

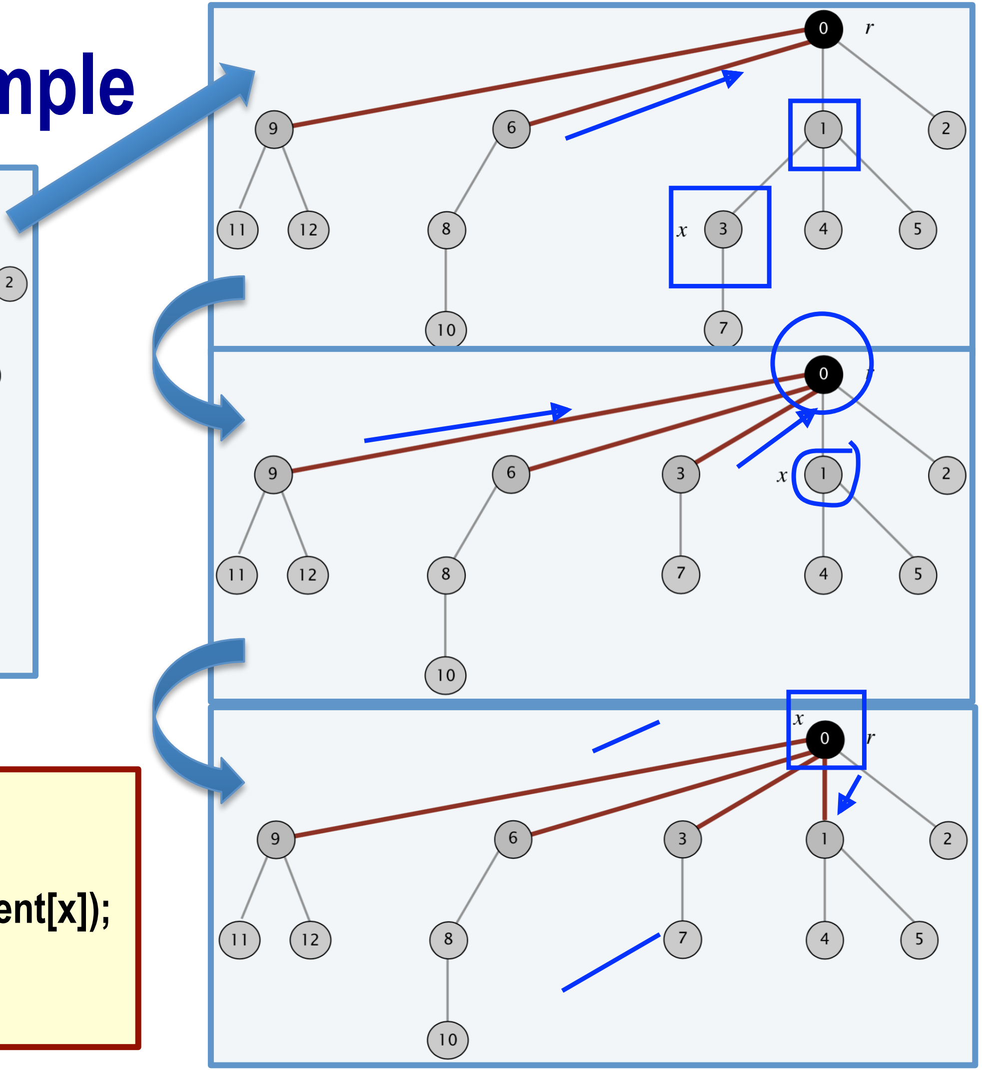
- When finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*



Path Compression: Example




```
FIND-SET(x) {
    if(x is not parent)
        parent[x] ← FIND-SET(parent[x]);
    return x;
}
```



Properties of Union-by-Rank + Path Compression (1)

Property-0: The tree roots, node ranks, and elements within a tree are the same with or without path compression.

Property-1: If x is not a root node, then $rank[x] < rank[parent[x]]$

Proof: Path compression can make x point to only an ancestor of $parent[x]$

Property-2: If x is not a root node, then $rank[x]$ will never change again

Property-3: If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.

Proof: Path compression doesn't change any ranks, but it can change parents

If $parent[x]$ doesn't change during a path compression the inequality continues to hold

if $parent[x]$ changes, then $rank[parent[x]]$ strictly increases

Property-4: Any root node of rank k has $\geq 2^k$ nodes in its tree

Property-5: The highest rank of a node is $\leq \lfloor \log_2 n \rfloor$

Property-6: For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank k

Properties of Union-by-Rank + Path Compression (2)

- **Definitions:**
  - $\log^* n = 0$, when $n \le 1$
    $= \text{MIN}\{ i \ge 0 \mid \underbrace{\log_2 \log_2 \ldots \log_2 n}_{i \text{ times}} \le 1\}$, when $n \ge 2$
  - $\log^* n = 0$, when $n \le 1$
    $= 1 + \log^* (\log_2 n)$, otherwise
  - *Ackerman Function*, $F(j) = 1$, when $j = 0$
    $= 2^{F(j-1)}$, when $j \ge 1$

Recursive Definition

Inverse Ackerman Function

Rank	Groups
1	0
2	1
[3, 4]	2
[5, 16]	3
[17, 65536]	4
[65537, 2^{65536}]	5

Property-7: The largest group number is $\le \log^* (\log_2 n) = \log^* n - 1$

Proof: Since largest possible rank is $\lfloor \log_2 n \rfloor$, hence the result

Property-8: Number of nodes in a particular group g is given by, $n_g < n/F(g)$

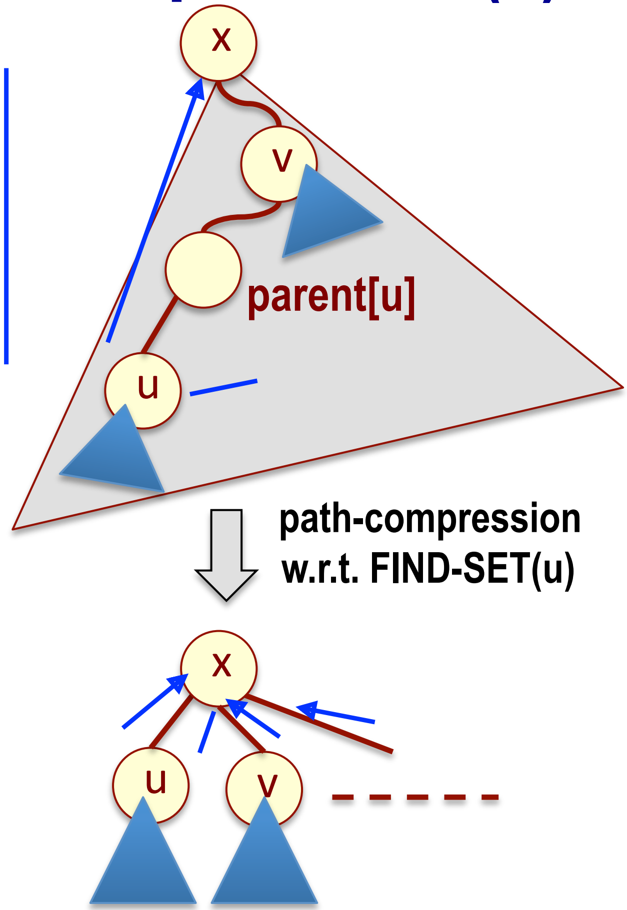
Proof: $n_g < \sum_{r=F(g-1)+1}^{F(g)} n/2^r < 2n/2^{F(g-1)+1} = n/2^{F(g-1)} = n/F(g)$

[since, $n/2^r + n/2^{r+1} + n/2^{r+2} + \ldots + n/2^{r+k}$
$< (n/2^r) \sum_0^\infty (1/2^k) = 2n/2^r$]

Analysis of Union-by-Rank with Path Compression (1)

- **Case-1:** If v is root (= x), a child of root or if parent[v] is in a different rank group; then we charge ONE unit of time to FIND-SET operation
- **Case-2:** If v ≠ x, and both v and parent[u] are in the same group, then we charge ONE unit of time to node v

- **Observation-1:** Ranks of nodes in a path from u to x increases monotonically
  - After x is found to be the root, we do path compression
  - If later on, x becomes a child of another node and v & x are in different groups, no more node charges on v in later FIND-SET operations



Analysis of Union-by-Rank with Path Compression (2)

- **Observation-2:** If a node v is in group g (g > 0), v can be moved and charged at most [F(g) – F(g-1)] times before it acquires a parent in a higher group.

- **Complexity Analysis:**
  - Time Complexity = (Number of nodes in group g) x (Movement charges across groups) x (Movement charges with groups) = (n/F(g)) x (log* n) x [F(g) – F(g-1)]

    ≤ n log* n   [since, (n/F(g))x[F(g) – F(g-1)] ≤ n]

- **Theorem:** The time complexity required to process m UNION and FIND-SET operations using union-by-rank with path-compression heuristic is O(m log* n) in the worst case
  - which may be also said as O(m), as log*n ≤ 5 practically

    (as otherwise n is more than the number of atoms in universe!!)

Thank you