Design and Analysis of Algorithms

L25: Kruskal's Algorithm
Minimum Cost Spanning Tree

Dr. Ram P Rustagi
Sem IV (2019-H1)
Dept of CSE, KSIT/KSSEM
rprustagi@ksit.edu.in

Resources

- Text book 2: Sec 4.1, 4.3, 4.4
- Text book 1: Sec 9.1-5.4 - Levitin
- R1: Introduction to Algorithms
    - Cormen et al.
- MIT Open CourseWare
    - https://ocw.mit.edu/courses/civil-and-environmental-engineering/1-204-computer-algorithms-in-systems-engineering-spring-2010/lecture-notes/MIT1_204S10_lec11.pdf
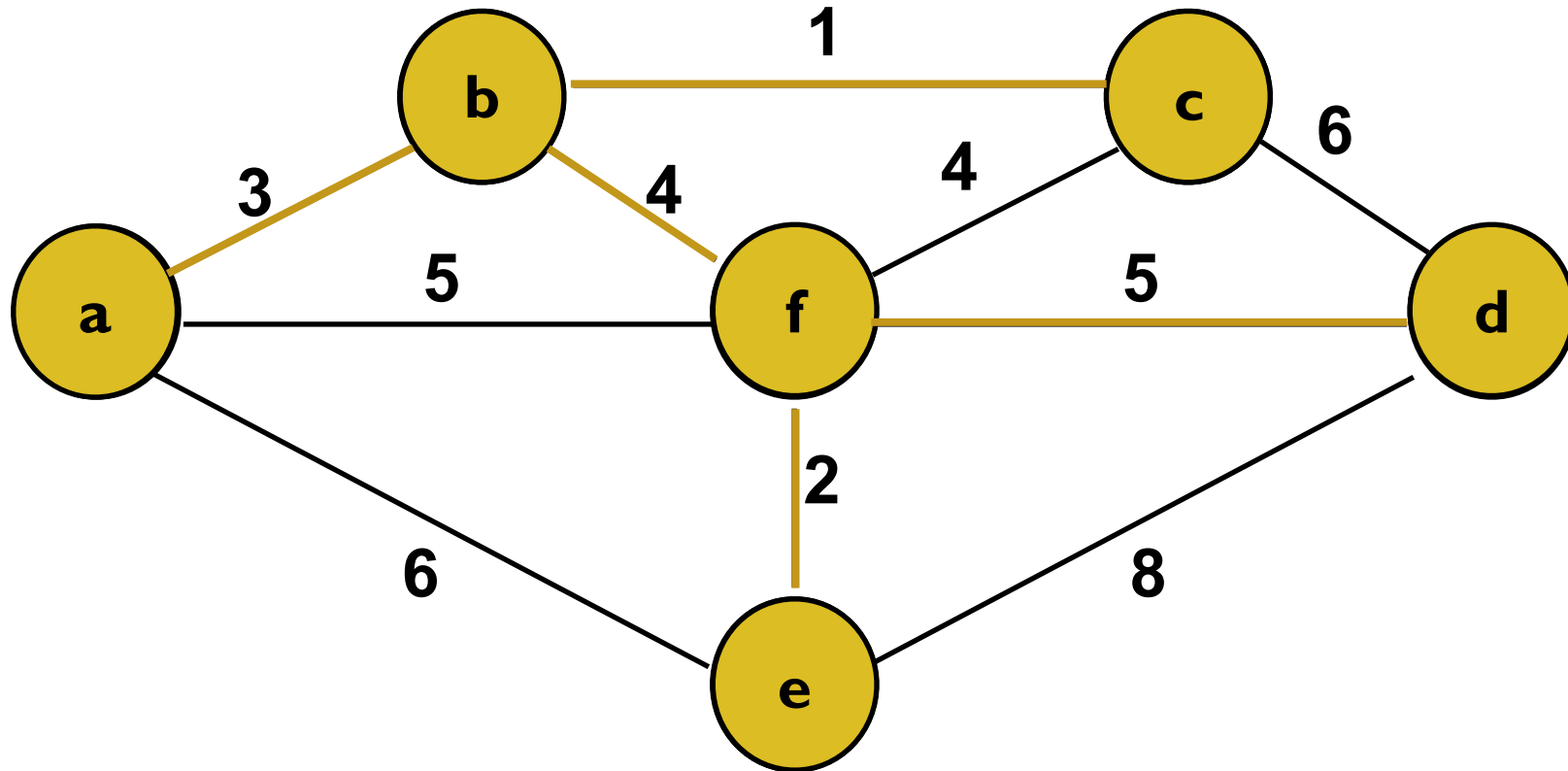
Kruskal's Algorithm

- Sort the edges in nondecreasing order of lengths
- "Grow" tree one edge at a time
  - Produce MST through a series of expanding forests $F_1, F_2, ..., F_{n-1}$
- On each iteration,
  - Consider the next edge on the sorted list
    - If this edge creates a cycle,
      - Skip the edge
    - Else
      - Add the edge to spanning tree

Kruskal's Algorithm
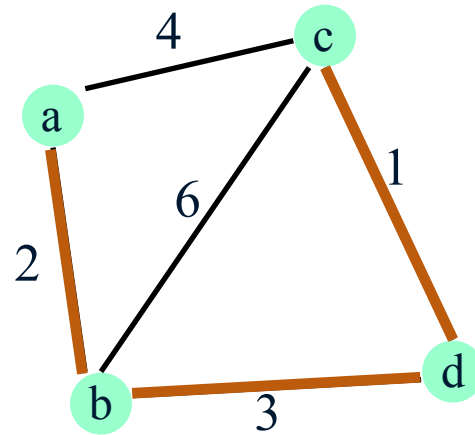
Example 2: Kruskal Algorithm

- Q: Construct an MST using Kruskal algo

Analysis of Kruskal's Algorithm

- Algorithm looks easier to implement
  - Just sort the edge in non-increasing order of weights and consider one edge at a time
- Cycle checking is harder to implement
  - A cycle occurs when the added edge connects vertices in the same connected component
- Using Union-Find algorithm to merge the connected components
- Time complexity:
  - `O(m*lg m)`, where `m` is number of edges.
  - The time spent is mostly on sorting.

Implementation of Kruskal's Algorithm

```
Algo Kruskal(G)
```
// i/p: A weighted connected graph $G= \langle V, E \rangle$

// o/p: Et, the set of edges composing an MST of G

sort E in non-decreasing order of edge weights i.e.
  $w(e_{i_1}) \leq w(e_{i_2}) \leq ... \leq w(e_{i_m})$

$E_T \leftarrow \varnothing;$ edgecount $\leftarrow 0;$ k $\leftarrow 0$

while edgecount $<$ $|V|$ $-$ 1 **do**

   k $\leftarrow$ k+1

   if E_T **U** $\{e_{i_k}\}$ **is acyclic**

      $E_T \leftarrow E_T$ **U** $\{e_{i_k}\}$

      edgecount $\leftarrow$ edgecount+1

// **end while**

return E_T

Union-Find Approach

- Using set based Union-Find approach
  - It is almost $O(n)$
- Union-Find approach
- Consider the set of n elements which are known.
- All these elements put in an array and their id can be the array index i.e.
  - $a[i] = x_i$ # i^{th} element is x_i.
- Elements are divided into groups (sets)
  - Initially, each element is a group by itself.
- Two kinds of operations:
  - Find the group to which element belongs
  - Merge the two groups

Time Complexity Analysis

- Sorting the edges
  `O(|E|lg |E|)`
- While loop : `|E|` times
  - Checking for cycle formation
    - Use Union Find approach for an edge
      `lg*n`
- Time complexity for cycle checking for all edges
  - `O(|E|lg*|V|)`
- Total time complexity
  `O(|E|lg |E|) + O(|E|lg*|V|)`
  `=O(|E|lg |E|)`

Union-Find Approach

- Two operations given below are performed in arbitrary order
  - *Find(`i`)*: return the group id containing element xi.
  - *Union(`A`,`B`)*: Combine the (set) group A with (set) group B to form a new group.
    - Give a unique name to this group. All elements of this new group will have this group id.
    - This could be one of earlier groups as well i.e.
      - The new names should conflict with other names.
- Goal: Design an efficient data structure that will support any sequence of these two operations.

Union-Find Approach

- Approach 1: Quick Find
  - Keep _Find(`i`)_ efficient. Since all elements are accessible at the `i`th index in array,
    - This can take `O(1)` time. Essentially, a trivial operation.
  - _Union(`A,B`)_ is expected to take more time.
    - Either change the id of all elements of A to that of B or vice versa.
    - Typically, take the smaller set and change group identity of these elements to that of larger set.
- Time complexity: `O(nlog₂n)`
  - Each time an element's group is changed, group size at least doubles.

Union-Find Approach: Improved

- Approach 2: Quick Union
  - Make *Union(A,B)* efficient at the cost of *Find(i)*.
    - Make *Union* operation takes constant time and improve upon the time taken by *Find*.
    - Use the indirect addressing for union to make it in `O(1)` time.
    - Each entry in the array has following parts
      - Identity of element i.e. group id or value.
      - Pointer which is initially `Nil`.
      - Number of elements in teh group
    - *Union*(A,B) is performed by making the pointer of B to point to A.
    - After several union operations, data structures becomes like a tree.
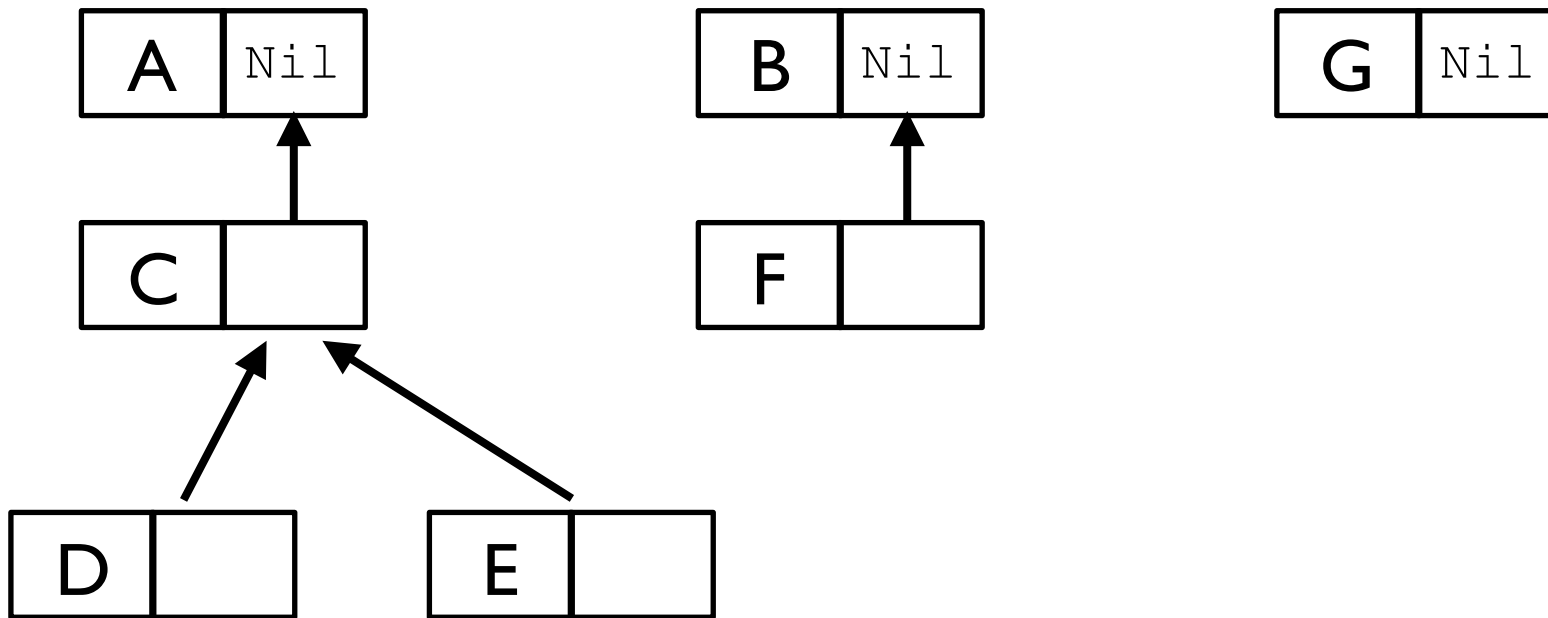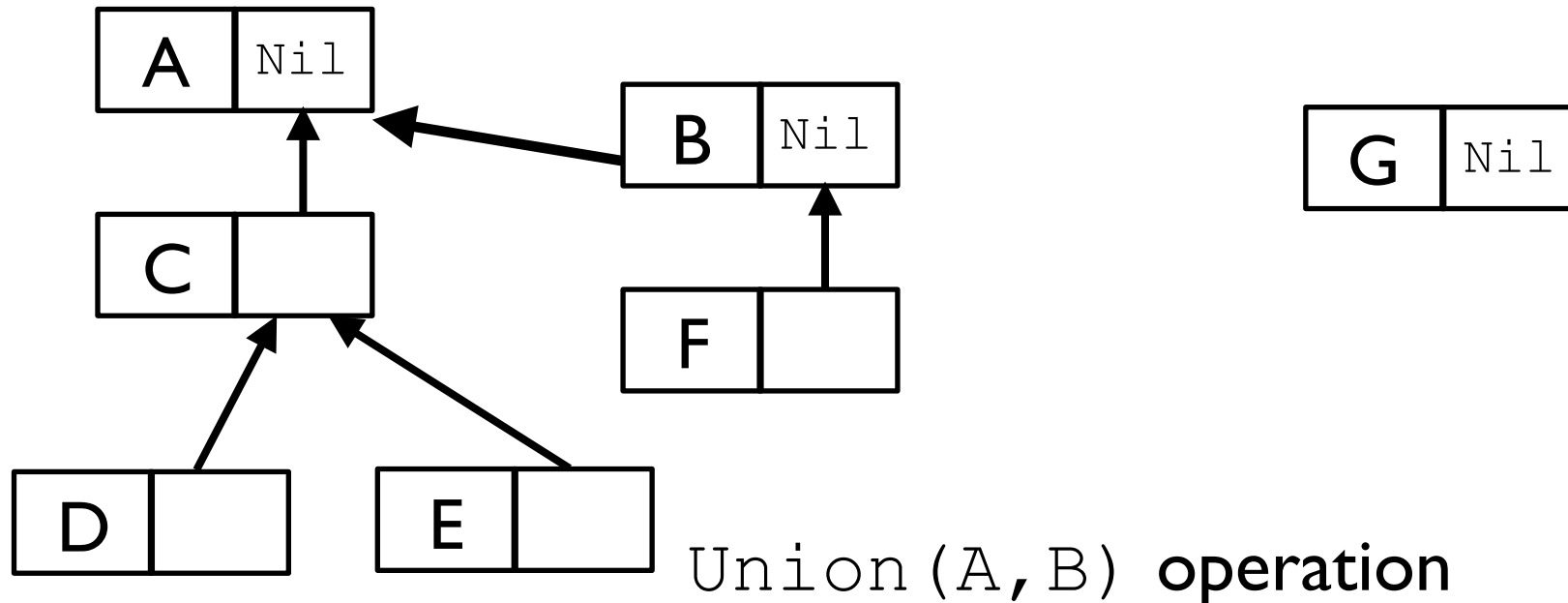
Union-Find Approach: Improved

Fig: Representation for Union Find problem

- The element at root of the Tree is the identity of group.
- To find the group of an element, follow the path till the root of the tree.

Union-Find Approach: Improved

`Union(A,B)` operation

- Take the tree with smaller number of nodes to point to root of the tree of larger number of nodes.
- This requires storing the number of elements in the tree at the root as well.
- On Union operation, update the pointer of smaller tree and count at the larger tree.
 - Break the tie arbitrarily

Union-Find Approach: Improved

- Basic idea: Balance and collapse the tree
  - Union operation still takes $O(1)$ time
    - Changing the pointer $O(1)$ time
    - Updating the count $O(1)$ time
  - Thus, $O(1) + O(1) = O(1)$
    - i.e. Union operation takes $O(1)$ time

Union-Find Approach: Quick Union

- Theorem: When balancing is used, the tree of height h will contain at least 2^h nodes.
- Proof outline
  - First union operation results in tree of height 1 with two elements.
  - Consider A is of height h_A and B is of height h_B.
    - Let A is larger tree. Thus, on merging B, root of B points to root of A.
    - If $h_A > h_B$, then A's height remains h_A i.e. unchanged
    - Otherwise, height of tree becomes $h_B + 1$
    - Thus, with increase in height of 1, the size (number of nodes) of tree has at least doubled.
- Thus, time taken for a _Find(_X_)_ operation is $O(\log_2 n)$

Quick Union of 2 Trees
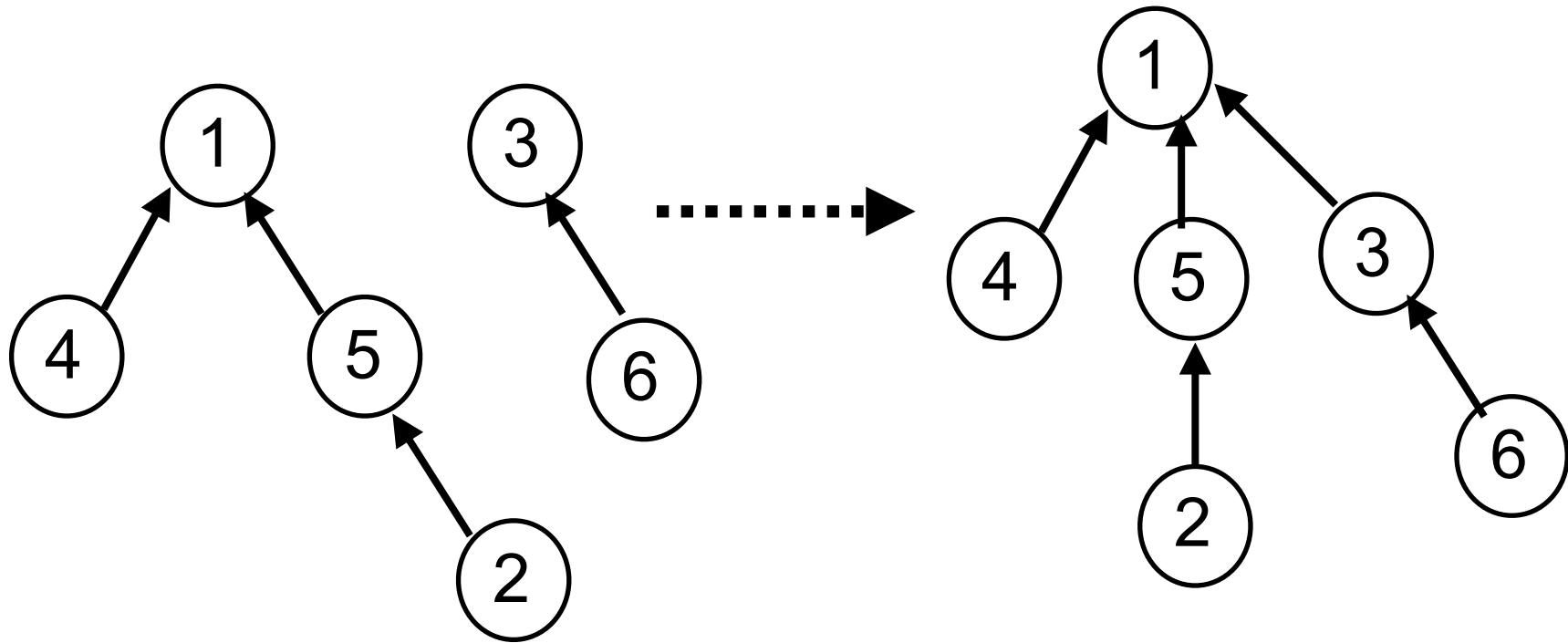
a: Forest representation of
two sets used by quick union

b: Result of quick union

Union-Find Approach: Quick Union

- Further improvement
- Any time we do a *Find* operation, change the pointers of all the nodes in the path to directly point to the root of the tree.
    – This is called **path compression**.
- Traversing the path takes only double the number of steps, and thus
    – Time complexity of *Find* remains the same.
- Time complexity with path compression for `m` operations is given by

    `O(mlog*n),` where `log*n` is iterated logarithm function

Quick Union of 2 Trees
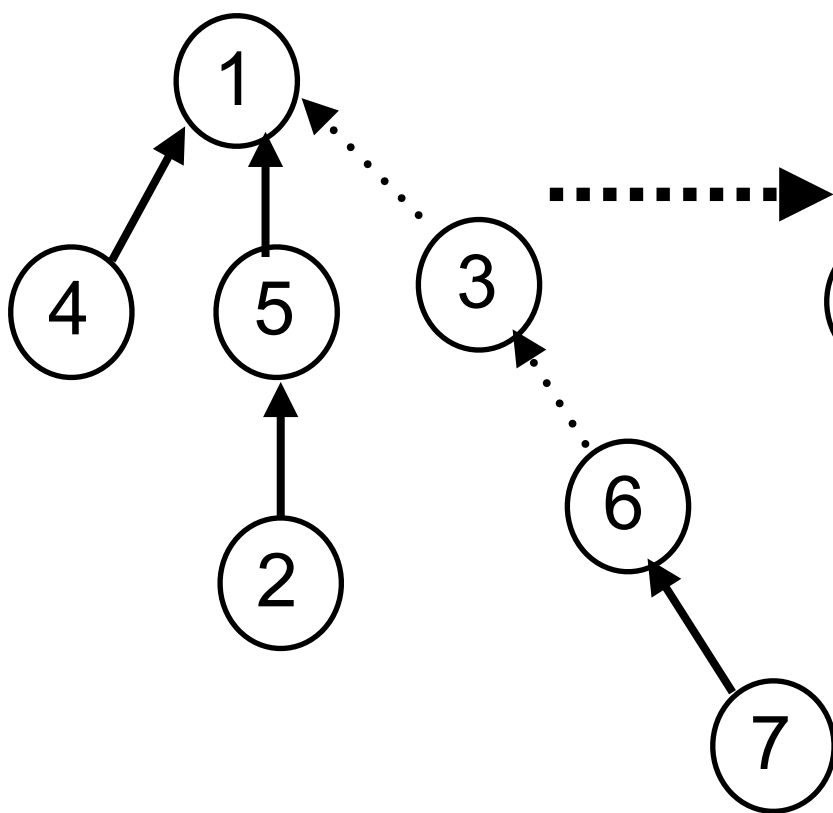
a: Representation of Tree before finding node 6
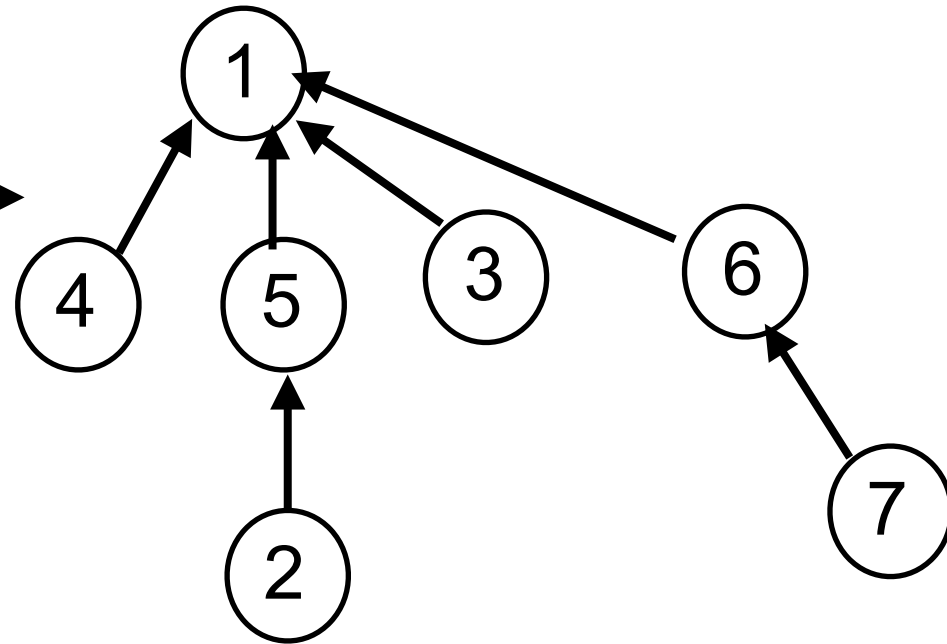
b: Result of path compression All nodes in the path from 6 to root 1 point to root 1

Iterated Logarithm

- Iterated logarithm function is defined as

```
log*n=1+log*(⌈log₂n⌉)
log*2=1 (Given)
log*4=log*2²
```

$$=1+\log^*(\lceil \log_2 2^2 \rceil)$$
$$=1+\log^*2=1+1 = 2$$

```
log*16=log*2⁴
```

$$=1+\log^*(\lceil \log_2 2^4 \rceil)$$
$$=1+\log^*4=1+2 = 3$$

```
log*65536=log*2¹⁶
```

$$=1+\log^*(\lceil \log_2 2^{16} \rceil)$$
$$=1+\log^*16=1+3 = 4$$

$$\log^* 2^{65536}=1+\log^*65536 = 5 \text{ (very large n)}$$

Summary

- Kruskal Algo
  - Sort the edges in non-decreasing order of weights
  - take one edge at a time and check for cycle
  - Cycle check is done by using Union-Find algo
  - Time complexity: `O(|E|lg|E|)`