COMP251: Disjoint sets

Jérôme Waldispühl
School of Computer Science
McGill University

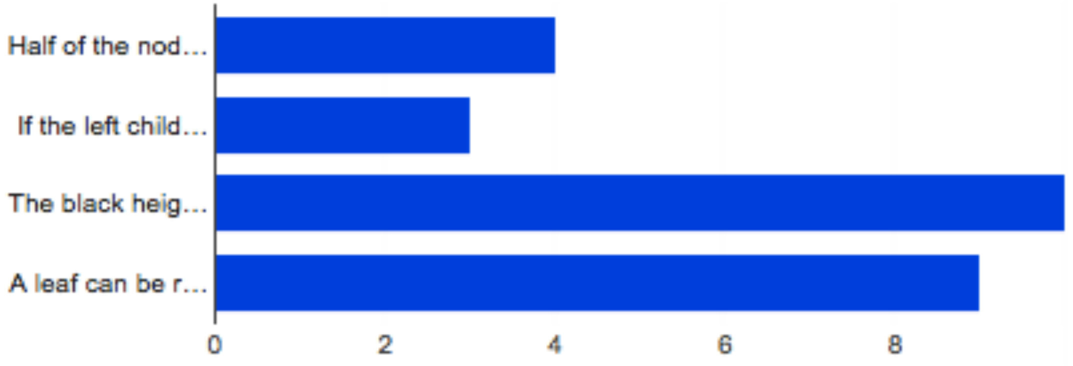
Based on slides from M. Langer (McGill)

Announces

- Assignment 1 is due on Wednesday February 1st.
- Submit your solution on MyCourse.
- Written solutions must be submitted in PDF format.
- Java files must compile & execute on SOCS machine!
- Use the forum: https://osqa.cs.mcgill.ca
- Use your SOCS login/password to login into the forum.
- If you do not have one, register at https://newuser.cs.mcgill.ca

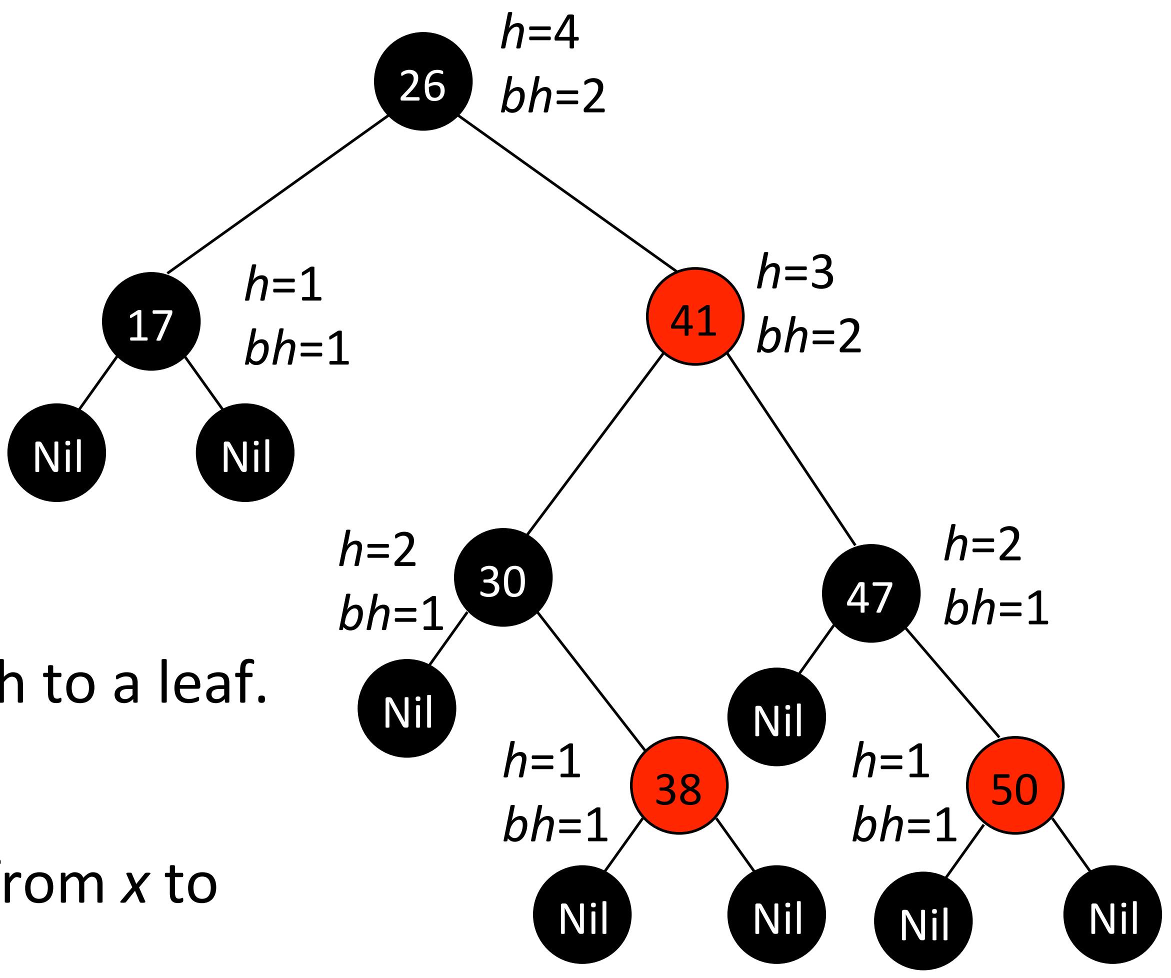
Let T be a red-black tree and x a node of this tree. Which of the following assertions are true?

- Half of the nodes on any path from x down to a leaf are black[*]. ✔
- If the left child of x is red, then its right child is red too. ✗
- The black height of the left and right sub-tree of x are identical. ✗
- A leaf can be red if and only if its parent node is black. ✗



[*] At least!

Height of a Red-black Tree



- Height h(x):

  #edges in a longest path to a leaf.

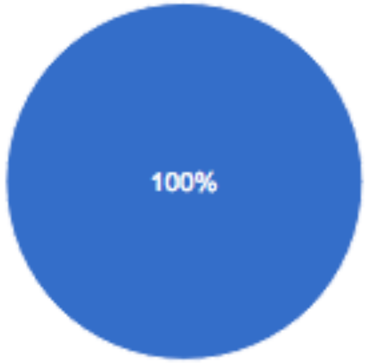
- Black-height bh(x):

  # black nodes on path from *x* to leaf, *not counting x*.

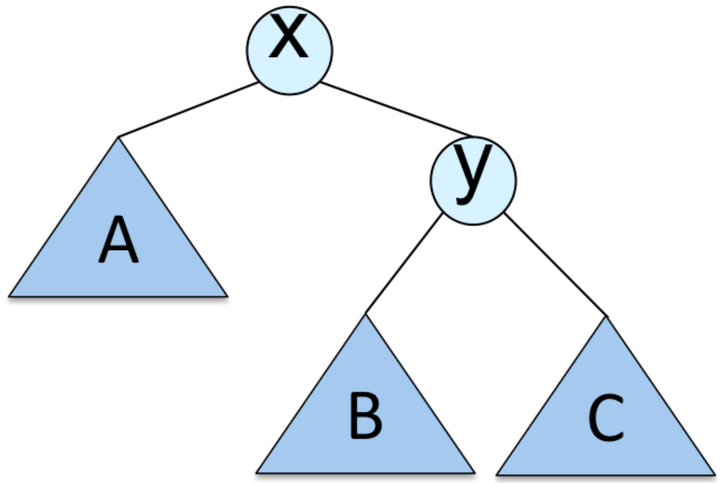
- Property: bh(x) ≤ h(x) ≤ 2 bh(x*)*

In RB tree insert, we assign the red color to the new node N being inserted. This allows to:

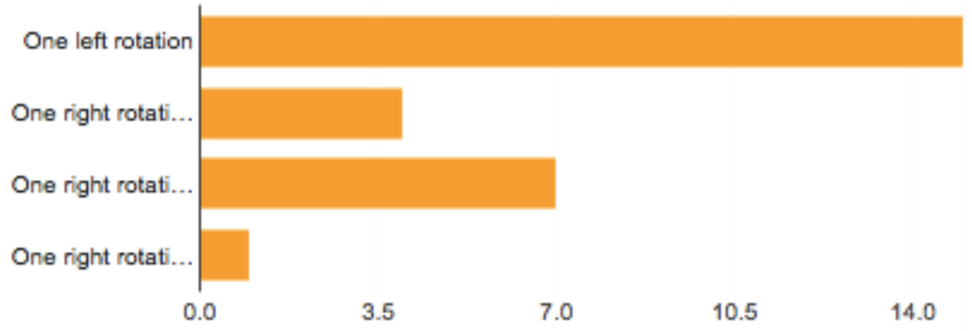
- preserve the black height of the paths traversing N. ✔
- avoid two consecutive black nodes.
- It does not matter. We can assign a black color instead and keep the same algorithm.



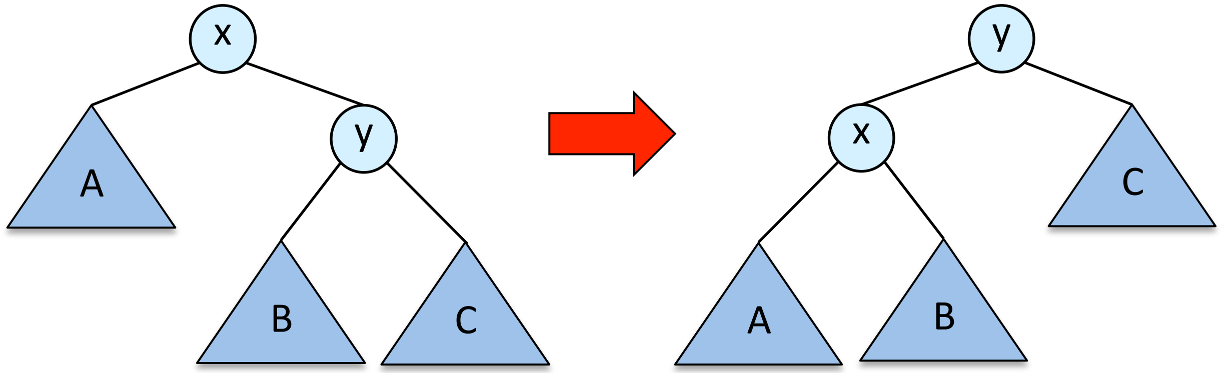
Consider the tree below and the node x and y. Which operations are allowed around the edge (x,y)?



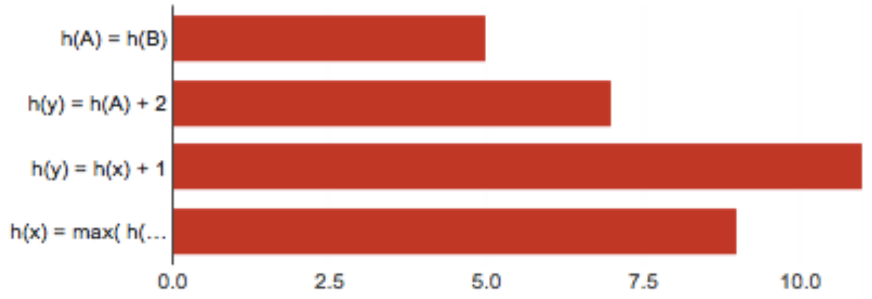
- One left rotation ✓
- One right rotation ✗
- One right rotation followed by left rotation ✗
- One right rotation followed by another right rotation ✗



Assume the tree below is a BST. We note h(A) (resp. h(B) and h(C)) the height of the subtree A (resp. B and C). We perform a left rotation at node x. Then...



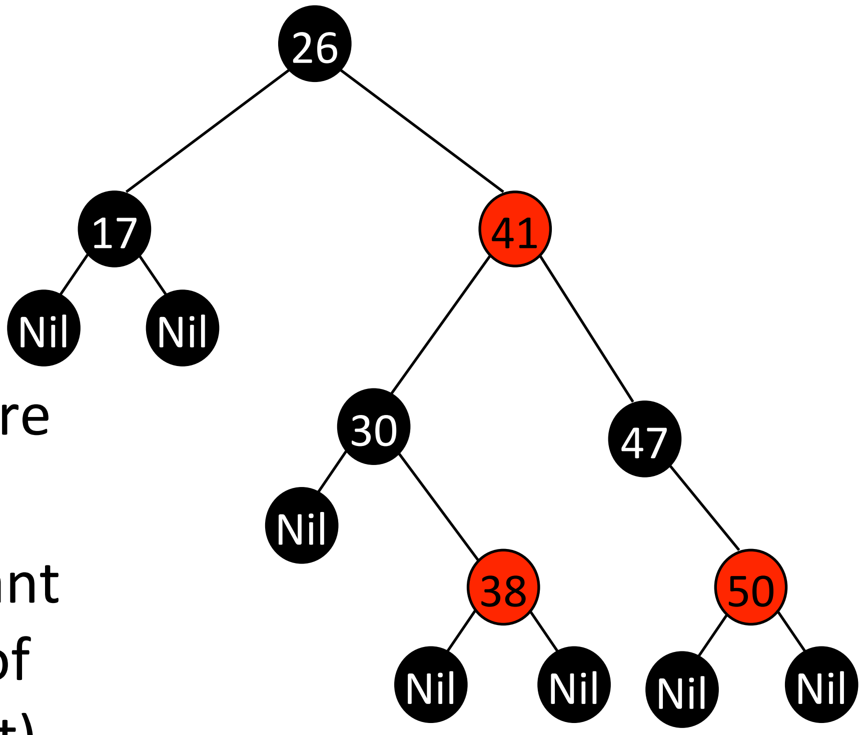
- h(A) = h(B) ✗
- h(y) = h(A) + 2 ✗
- h(y) = h(x) + 1 ✗
- h(x) = max(h(A), h(B)) ✗



Recap of the previous lecture

Properties:

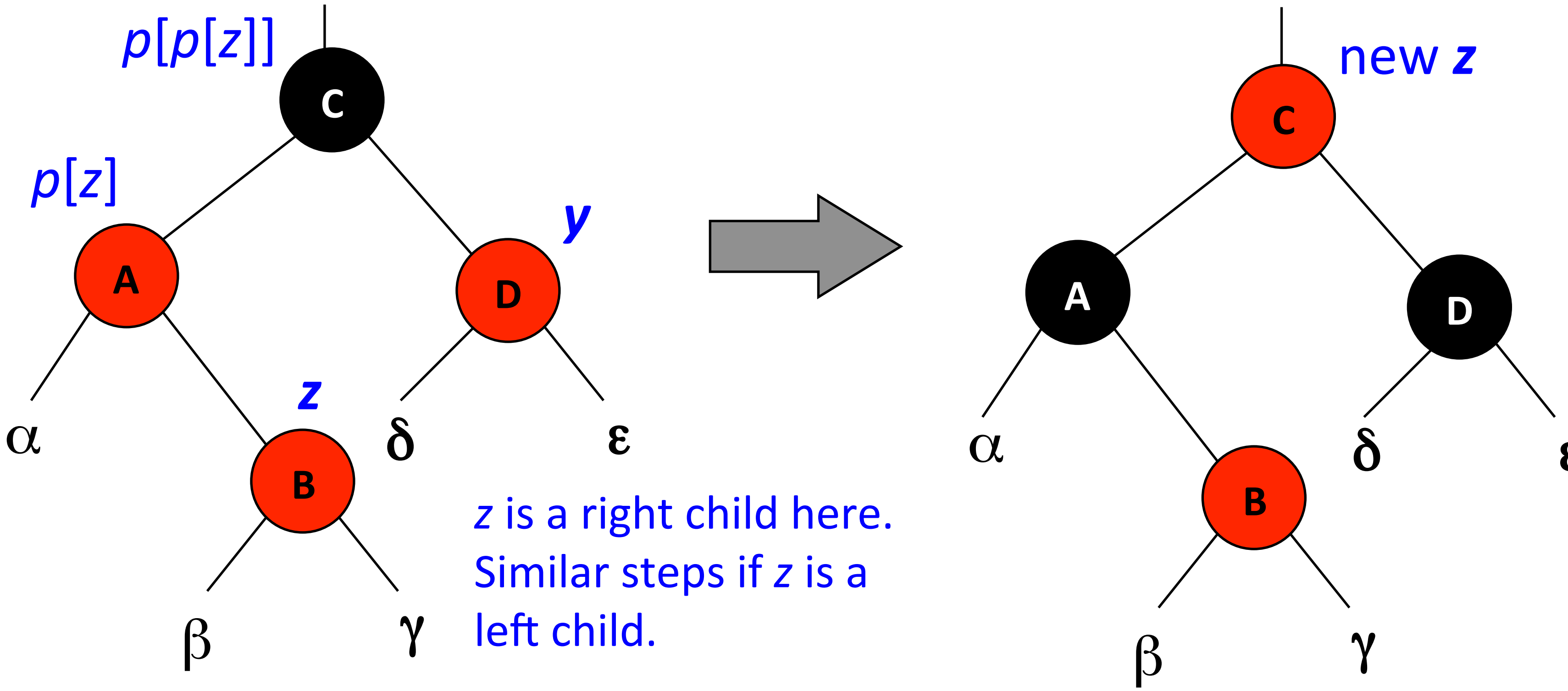
1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*) is black.
4. If a node is red, then its children are black.
5. All paths from a node to descendant leaves contain the same number of black nodes (i.e. same black height).



Insert node in a red-black tree:

1. BST insert.
2. Assign the color red to the new node.
3. Fix tree with recoloring of nodes and rotations

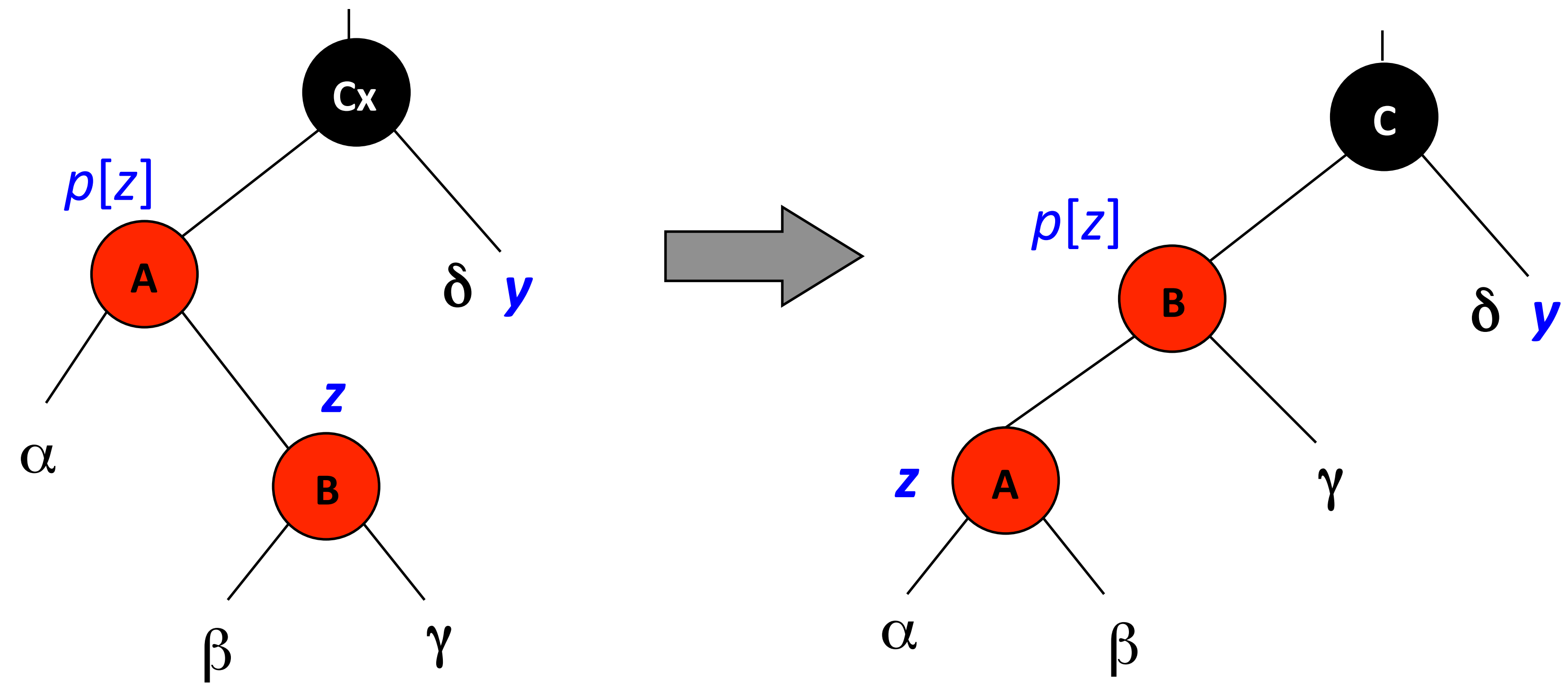
Case 1 – uncle *y* is red



z is a right child here. Similar steps if *z* is a left child.

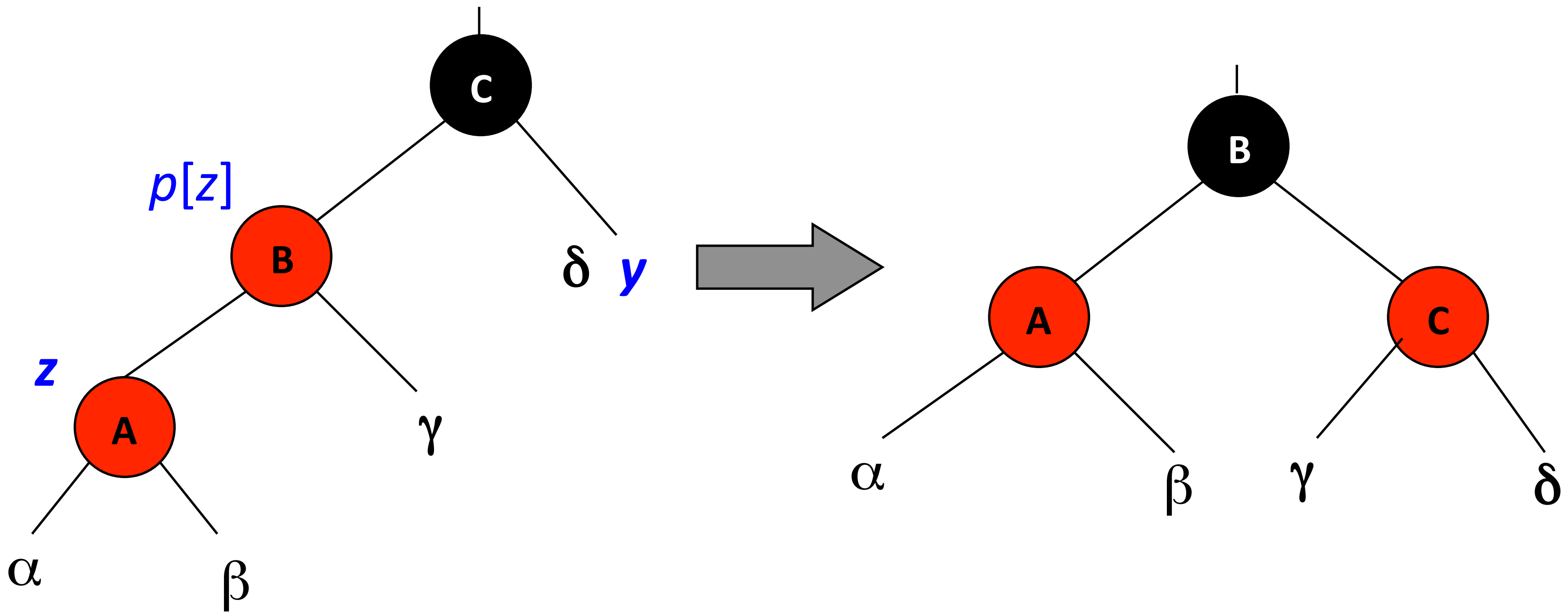
- *p*[*p*[*z*]] (*z*' s grandparent) must be black, since *z* and *p*[*z*] are both red and there are no other violations of property 4.
- Make *p*[*z*] and *y* black ⇒ now *z* and *p*[*z*] are not both red. But property 5 might now be violated.
- Make *p*[*p*[*z*]] red ⇒ restores property 5.
- The next iteration has *p*[*p*[*z*]] as the new *z* (i.e., *z* moves up 2 levels).

Case 2 – *y* is black, *z* is a right child



- Left rotate around *p*[*z*], *p*[*z*] and *z* switch roles ⇒ now *z* is a left child, and both *z* and *p*[*z*] are red.
- Takes us immediately to case 3.

Case 3 – *y* is black, *z* is a left child



- Make *p*[*z*] black and *p*[*p*[*z*]] red.
- Then right rotate on *p*[*p*[*z*]]. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- *p*[*z*] is now black $\Rightarrow$ no more iterations.

AVL & Red-Black trees

- Both are BST trees with additional properties.
- These properties allow to guarantee that the BST are balanced.
- Insertion is achieved with a classical BST insert followed by operations to restore the tree properties (i.e. AVL or RB).
- Rotation is the basic operation to modify trees while preserving the BST properties.
- We can use these data structures (i.e. AVL and RB trees) to design a sorting algorithms with guaranteed O(log n) running time.

Problem

Let $G=(V,E)$ be undirected graph, and $A, B \in V$ two nodes of G.

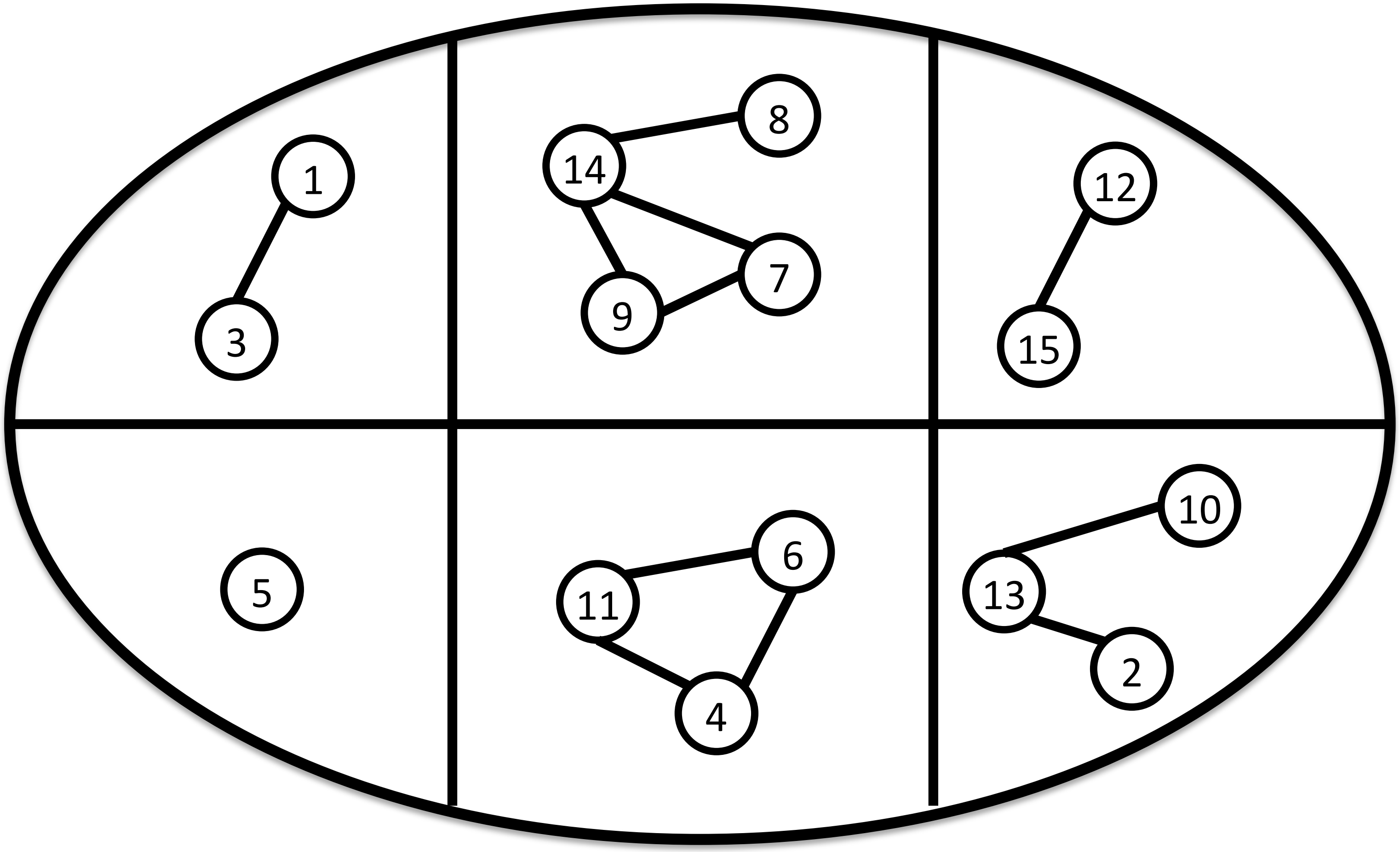
Question: Is there a path between A and B?

But we are not interested in knowing the path between A and B.

Is there a faster way to solve this problem (faster than DFS or BFS)?

Connected components

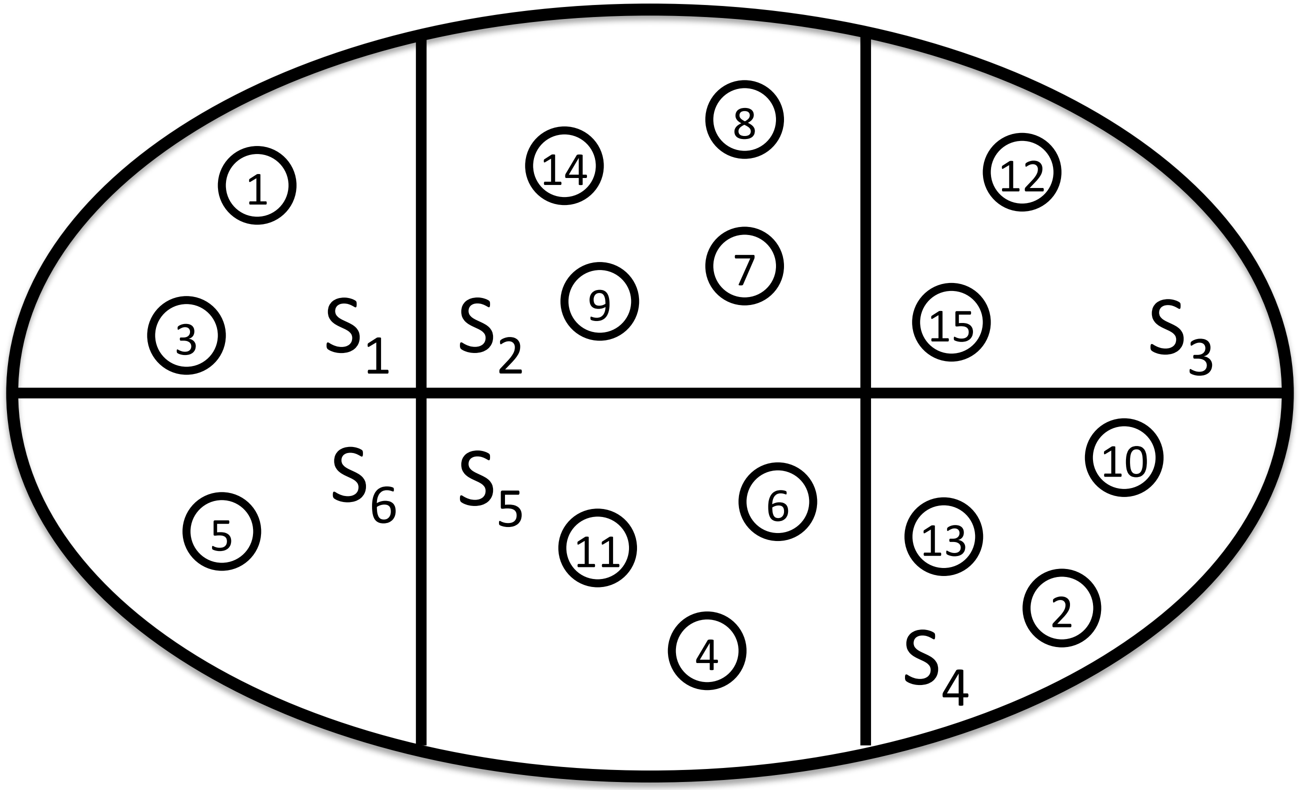
Connected component: Set of nodes connected by a path.



Question: Given 2 nodes A & B, are they in the same component?

Partition

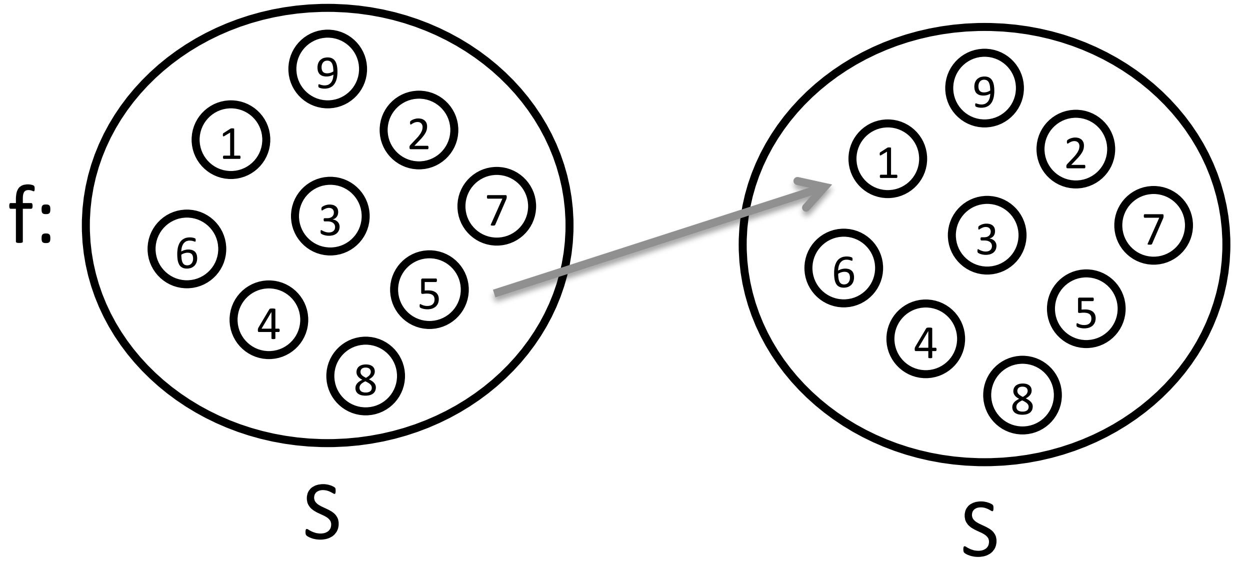
Generalization: Set of object partitioned into disjoint subsets.



$$S = S_1 \cup S_2 \cup \ldots \cup S_n \left\{ \begin{array}{c} S_i \neq \varnothing \ \forall i \in \{1,\ldots,n\} \\ S_i \cap S_j = \varnothing \ iff \ i \neq j \end{array} \right.$$

Map vs. Relation

Map or function



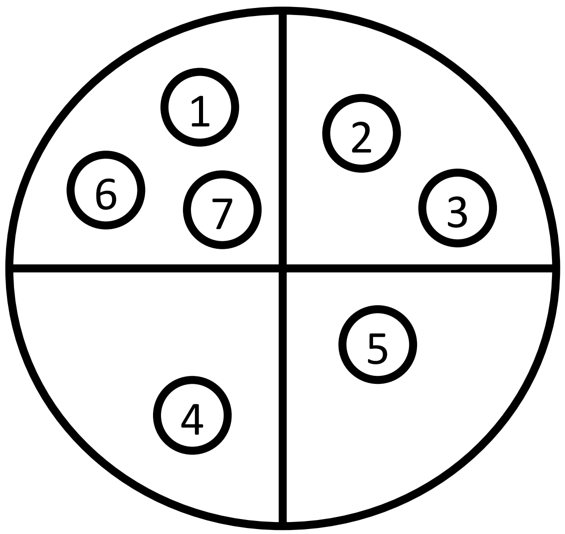
$\{(a,f(a)\}$

Relation $R \subseteq \{ (a,b) : a,b \in S \}$

$$
\begin{array}{c|ccccc|}
 & & & b & & \\
\hline
 & 0 & 1 & 1 & 0 & 1 \\
 & 0 & 1 & 0 & 1 & 0 \\
a & 1 & 0 & 0 & 1 & 0 \\
 & 1 & 1 & 0 & 0 & 1 \\
 & 0 & 1 & 1 & 0 & 0 \\
\hline
\end{array}
$$

Any boolean matrix defines a relation

Equivalence relation



$$\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
2 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
7 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
\end{array}$$

i is equivalent to j if they belong to the same set.

(more constrained that general relation)

Equivalence relation

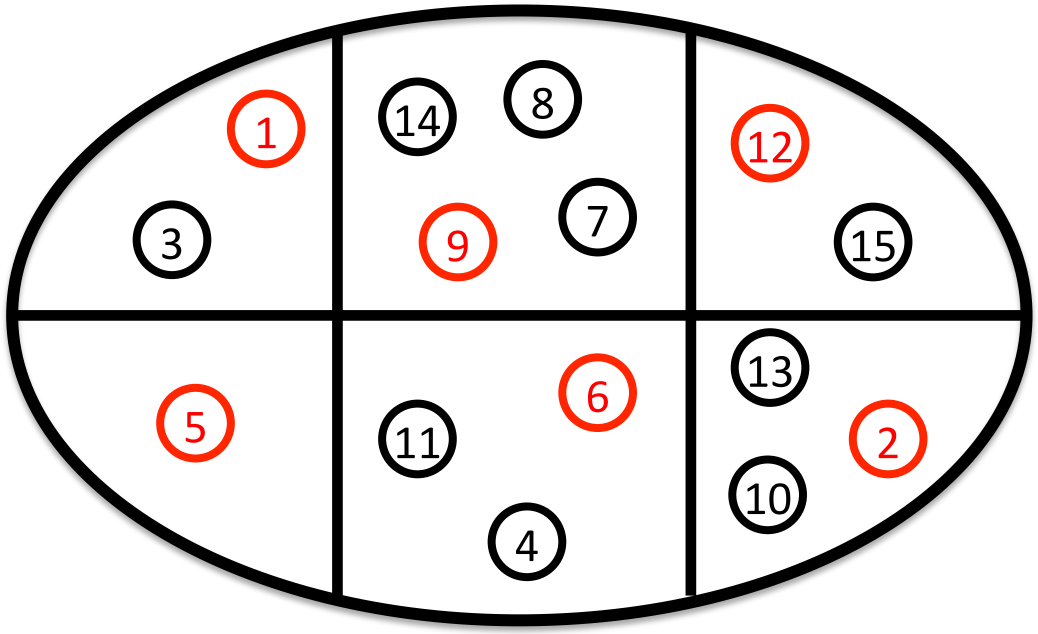
- Reflexivity $\forall a \in S, (a,a) \in R$
- Symmetry $\forall a,b \in S, (a,b) \in R \Rightarrow (b,a) \in R$
- Transitivity $\forall a,b,c \in S, (a,b) \in R \ and \ (b,c) \in R \Rightarrow (a,c) \in R$

Example:

For any undirected graph, the connections define an equivalence relation on vertices.

- For all u ∈ V, there is a path of length 0 from u to u.
- For all u,v ∈ V, There is a path from u to v, iff there is a path from v to u.
- For all u,v,w ∈ V, if there is a path from u to v and a path from v to w, then there is a path from u to w.

Disjoint set ADT



Each set in the partition as a representative member.

- find(i) returns the representative of the set that contains i.
- sameset(i,j) returns the boolean value find(i)==find(j)
- union(i,j) merges the sets containing i and j.

Union of disjoint sets

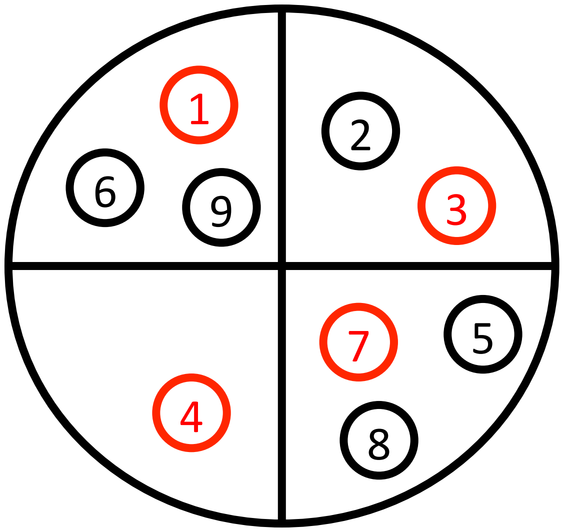
union(i,j) merges the sets containing i and j.

- Does nothing if i and j are already in the same set.
- Otherwise, we merge the set and need a policy to decide who will be the representative of the new merged set.

Quick find

Rep[]

1	1
2	3
3	3
4	4
5	7
6	1
7	7
8	7
9	1

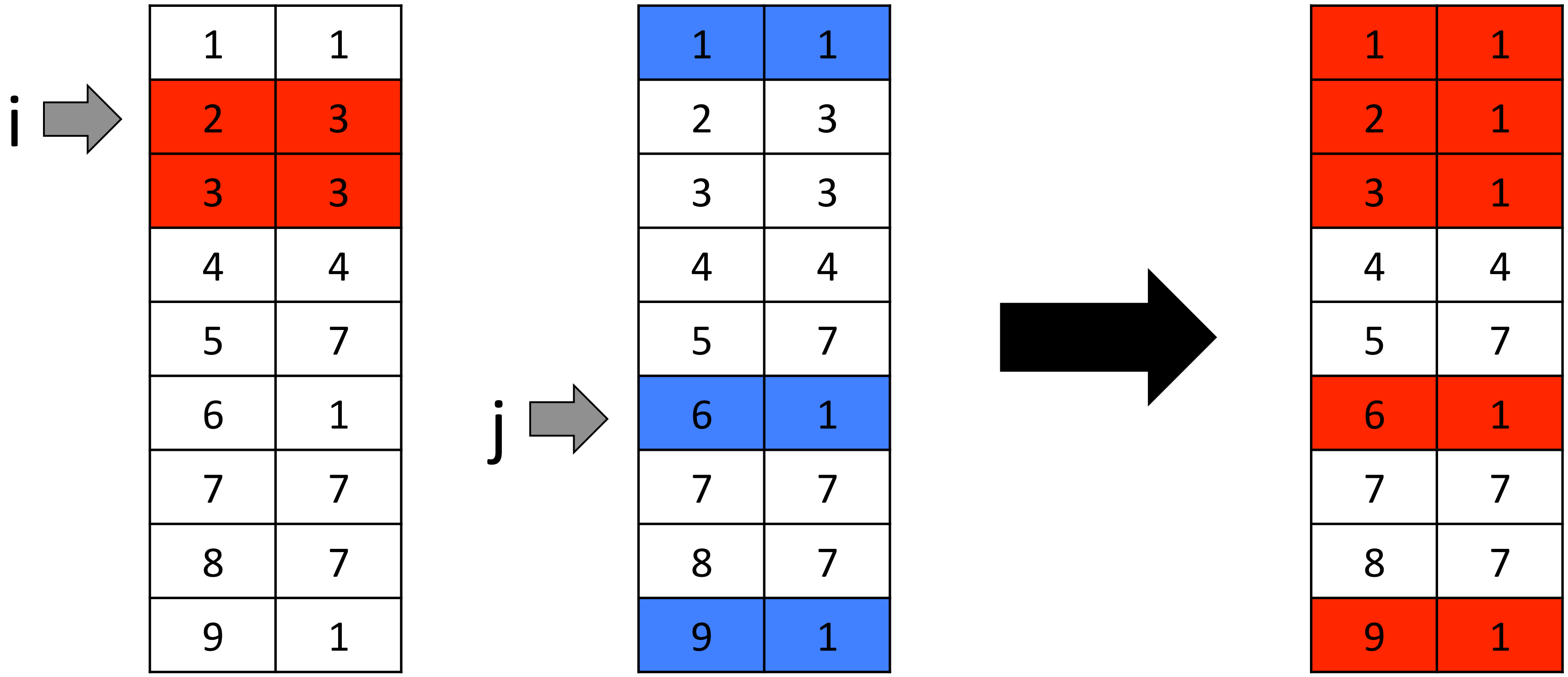


Let Rep[i] $\in$ { 1, 2, … , n } be the representative of the set containing i.

Quick fund & union

- find(i) { return rep[i]; }
- union(i,j) merges the sets containing i and j.

Example: union(2,6)



1	1
2	3
3	3
4	4
5	7
6	1
7	7
8	7
9	1

1	1
2	3
3	3
4	4
5	7
6	1
7	7
8	7
9	1

1	1
2	1
3	1
4	4
5	7
6	1
7	7
8	7
9	1

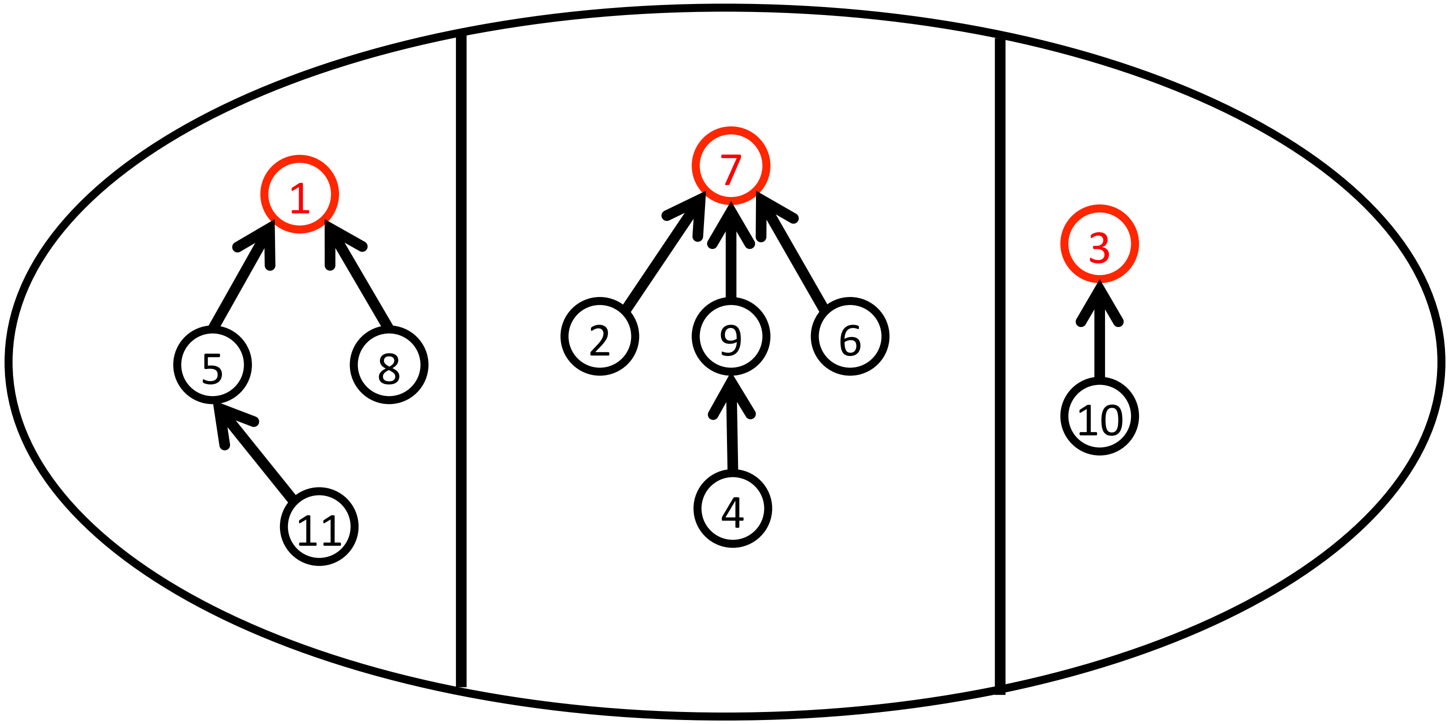
Quick find & union

```
union(i,j) {
  if rep[i] != rep[j] {
     prevrepi = rep[i];
     for (k=1; k<=n; k++) {
        if rep[k] == prevrepi {
           rep[k] = rep[j];
        }
     }
  }
}
```

- store value of rep[i] because it may change during the execution of the algorithm.
- O(n) running time… slow.

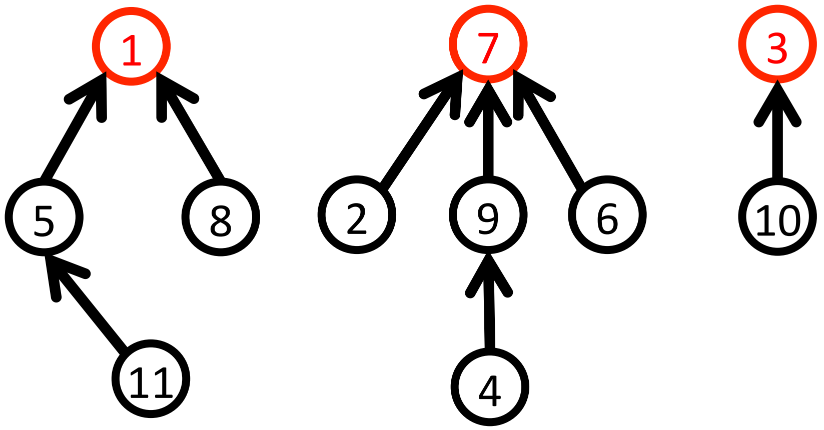
Tree representation & forests

- Represent the disjoint sets by a forest of rooted trees.
- Roots are the representative (i.e. find(i) == findroot(i)).
- Each node points to its parent.



Array representation

	p[]
1	1
2	7
3	3
4	9
5	1
6	7
7	7
8	1
9	7
10	3
11	5



- Non-root nodes hold index of their parent.
- Root nodes store their own value.

Find & Union

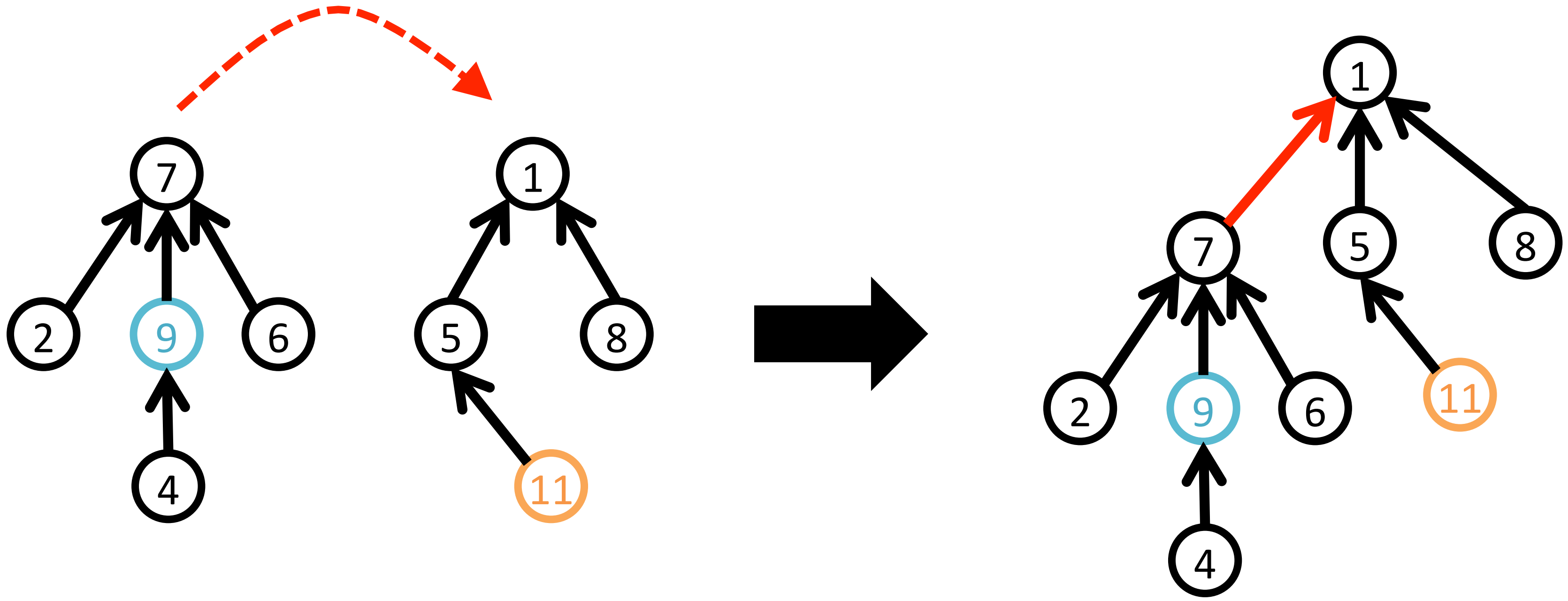
```
find(i) {
   if p[i] == i {
      return i;
   } else {
      return find(p[i]);
   }
}

union(i,j) {
   if find(i) != find(j) {
      p[find(i)] = find(j);
   }
}
```

Remark: Arbitrarily merge the set on i into the set of j.

Union example

union(9,11)

Root of the tree of 11 becomes the parent of the root of the tree of 9.



Worst case

union(1,2)

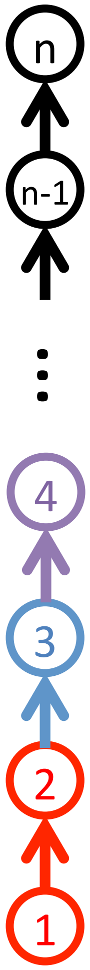
union(1,3)

union(1,4)

…

union(1,n)

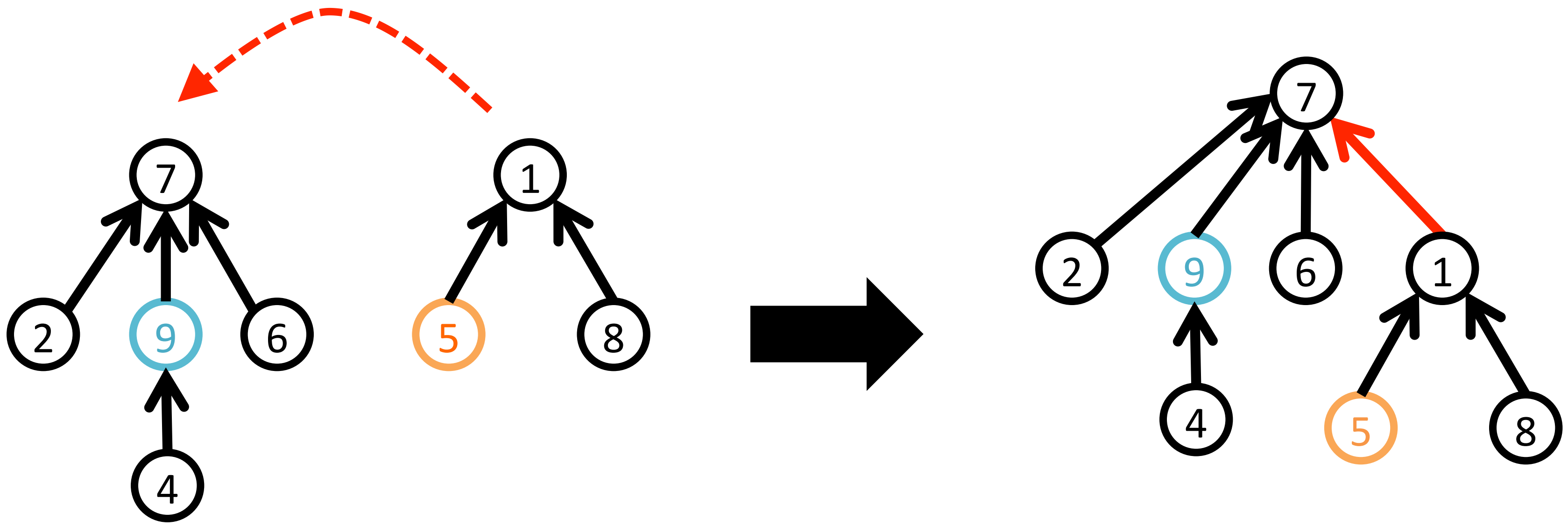
Then find(1) is O(n)!



Union by size

Heuristic to control the height to the trees after merging.

Idea: Merge tree with smaller number of nodes into the tree with the largest number of nodes (In practice, we can also use rank which is an upper bound on the height of nodes).

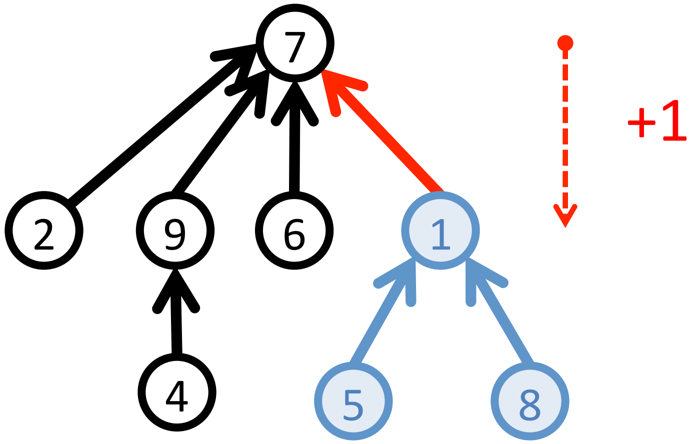


Union by size

Claim: The depth of any node is at most log n.

Proof:

- If union causes the depth of a node to increase, then this node must belong to the smallest tree.



- Thus, the size of the (merged) tree containing this node will at least double.
- But we can double the size of a tree at most log n times.

Union by height

Idea: Merge tree with smaller height into tree with larger height.

Claim: The height of trees obtained by union-by-height is at most log n.

Corollary: An union-by-height tree of height h has at least $n_h \geq 2^h$ nodes.

Proof (Corollary):

- Base case: a tree of height 0 has one node.
- Induction: (hypothesis) $n_h \geq 2^h$. Show $n_{h+1} \geq 2^{h+1}$.

Running time

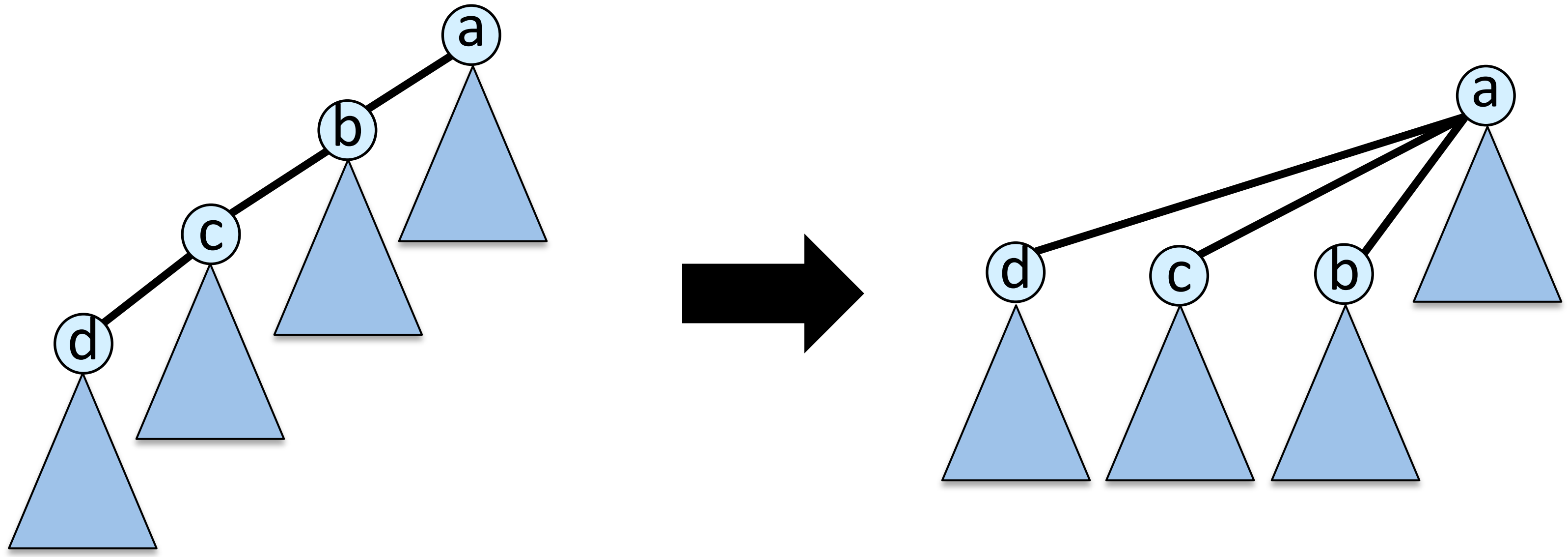
		find(i)	union(i,j)
	Quick find	O(1)	O(n)
Quick Union	Union by size	O(log n)	O(log n)
	Union by height	O(log n)	O(log n)

$\uparrow$ Quick union makes 2 calls to find.

Note: These are worst case complexities.

Path compression

- Find path = nodes visited during the execution of find() on the trip to the root.
- Make all nodes on the find path direct children of root.



Path compression

```
find(i) {
   if p[i] == i {
      return i;
   } else {
      p[i] = find(p[i]);
      return p[i];
   }
}
```

Running time

- Use union by size and path compression.
- Worst case running time is O(log n).
- However, we can show that m union or find operations take O(m $\alpha(n)$).

What is $\alpha(n)$?

n	$\alpha(n)$
0 - 2	0
3	1
4 - 7	2
8 - 2047	3
2048 – $A_4(1)$	4

Where $A_4(1) >> 10^{80}$!!