Union-Find

2023-Fall

국민대학교 최준수

Union-Find (Disjoint-Set) Data Structure

A data structure that stores a collection of disjoint (non-overlapping) sets.

- The universe consists of n elements, named 0, 1, …, $n-1$
- Each element is in exactly one set
  - Sets are disjoint (non-overlapping)
  - Initially, each set is a singleton (a set with exactly one element)
- Each set has a representative member (any element in the set will do)
- Operations:

Union/Find operations	
`void init(int n)`	initialize union-find data structure with n singleton sets ({0}, {1}, …, {n-1})
`void union(int p, int q)`	merge the set containing p and and the set containing q
`int find(int p)`	find a representative member of a set containing p (0 to n-1)
`boolean in_same_set(int p, int q)`	find out if p and q in the same sets?

Union-Find Data Structure

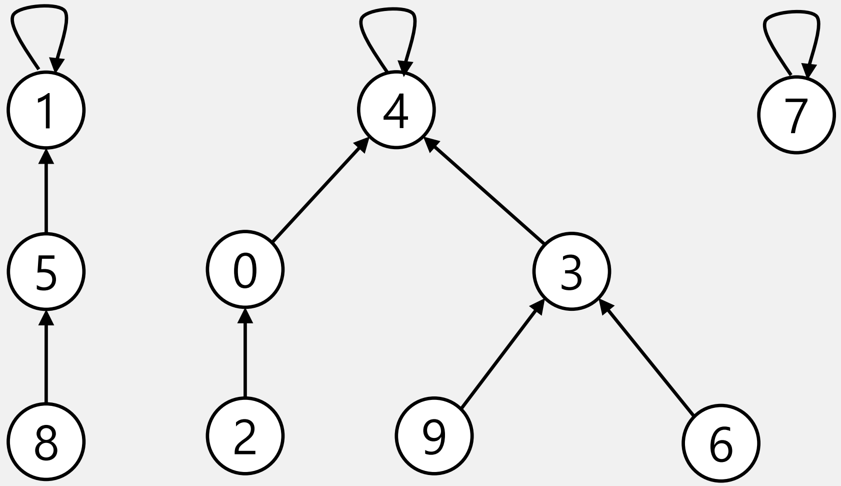
Goal : Design efficient data structure for union–find operations.

- Number of elements n can be huge.
- Number of union/find operations m can be huge.
- Union/find operations may be intermixed.

Union-Find Data Structure

Data Structure: disjoint-set forest

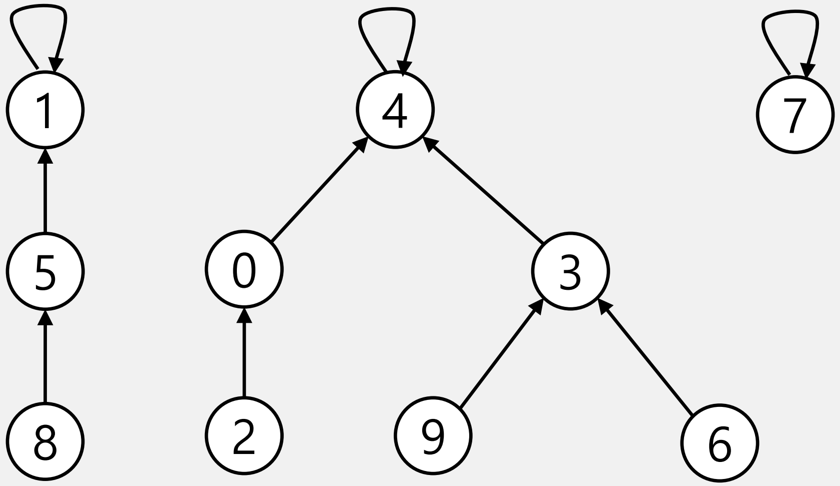
- Node of the forest
  - A pointer: used to make “parent pointer trees”
    - non-root node of a tree points points to its parent
    - root node points to itself or has a sentinel value (like NULL, -1, …)
  - Auxiliary information: a size
- Each tree represents a set stored in the forest
  - Members of the set being the nodes in the tree
  - Root nodes provides set representatives



Union-Find Data Structure

Data Structure:

- Integer array `pt[]` of length n.
- Interpretation: `pt[i]` is parent of i.
- Root of i is `pt[pt[pt[…pt[i]…]]]`.



	0	1	2	3	4	5	6	7	8	9
pt[i]	4	1	0	4	4	1	3	7	5	3

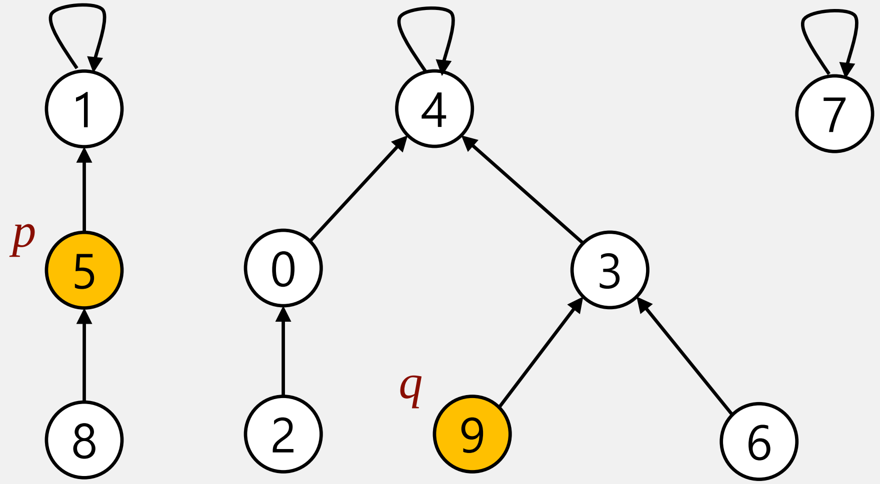
Union-Find Data Structure

Find

- What is the root of p?

In the same set?

- Do p and q have the same root?



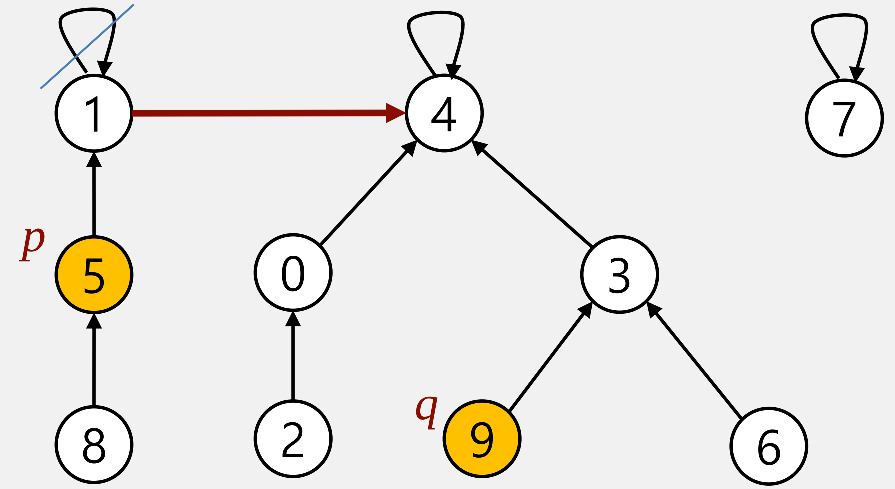
root of 5 is 1
root of 9 is 4
5 and 9 are not in the same set

Union/Find operations	
`int find(int p)`	find a representative member of a set containing p (0 to n-1)
`boolean in_same_set(int p, int q)`	find out if p and q in the same sets?

Union-Find Data Structure

Union

- Set the parent of p's root to the q's root.



`union(5, 9)`

	0	1	2	3	4	5	6	7	8	9
pt[i]	4	~~1~~	0	4	4	1	3	7	5	3

4

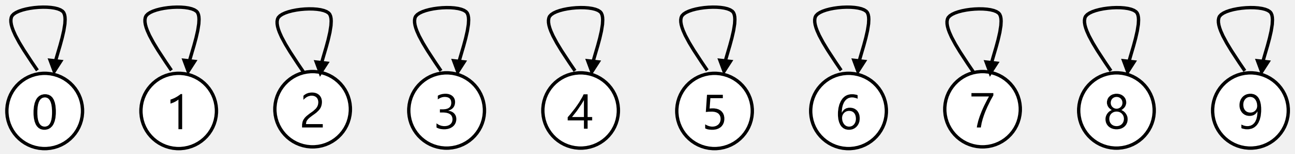
only one value changes

Union/Find operations	
`void union(int p, int q)`	merge the set containing p and and the set containing q

Union-Find Data Structure

Init

- Creates n singleton sets $\{0\}, \{1\}, \ldots, \{n-1\}$



	0	1	2	3	4	5	6	7	8	9
pt[i]	0	1	2	3	4	5	6	7	8	9

Union/Find operations	
`void init(int n)`	initialize union-find data structure with n singleton sets ($\{0\}, \{1\}, \ldots, \{n\text{-}1\}$)

Union-Find Data Structure: implementation

```
void init(int n)
{
    for(int i=0; i<n; i++)
        pt[i] = i;
}

int find(int i)
{
    while(pt[i] != i)
        i = pt[i];
    return i;
}

boolean in_same_set(int p, int q)
{
    return find(p) == find(q);
}

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

sets parent of each element to itself (create n singletons)

1. follows the chain of parent pointers from a query node i until it reaches a root element
2. returns the root element it reaches

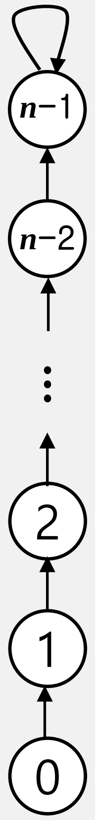
changes root of p to point to root of q

Union-Find Data Structure:

Shortcoming:

- Trees can get tall and `find()` operations are too expensive.

```
union(0, 1);
union(1, 2);
…
union(n-2, n-1);
find(0);
find(0);
…
find(0);
```

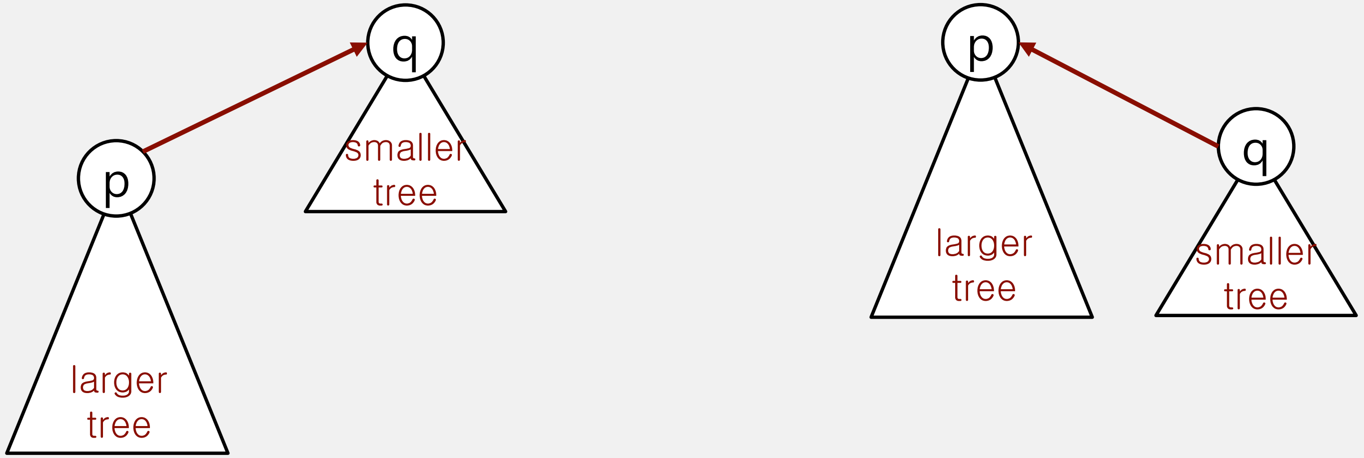


tree of height n-1
`find()` operates in O(n)

Improvement 1 (Weighting)

Weighting:

- Modify to avoid tall trees
- Keep track of size of each tree (number of elements)
- Balance by linking root of smaller tree to root of large tree.



Improvement 1 (Weighting)

Implementation of Weighting:

- maintain extra array `sz[i]` to keep the number of elements in the tree rooted at `i`.
- change the union function:
  - link root of smaller tree to the root of larger tree
  - update the `sz[]` array

```
void init(int n)
{
    for(int i=0; i<n; i++)
    {
        pt[i] = i;
        sz[i] = 1;
    }
}
```

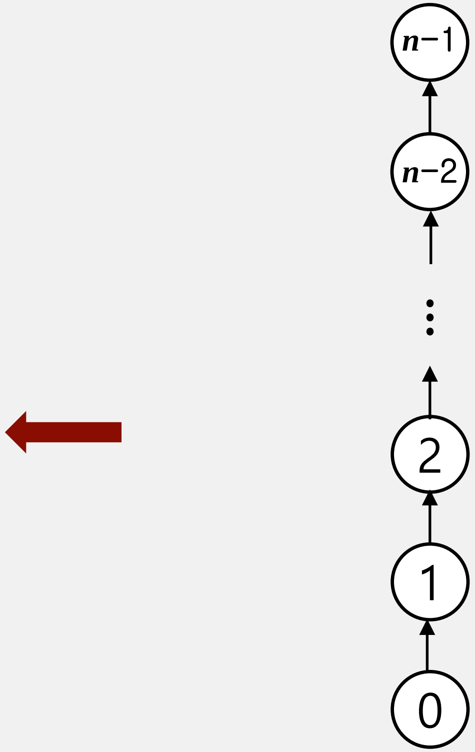
```
void union(int p, int q)
{
    int i = find(p);
    int j = find(q);

    if(i != j)
        if(sz[i] < sz[j]) { pt[i] = j; sz[j] += sz[i]; }
        else              { pt[j] = i; sz[i] += sz[j]; }
}
```

Improvement 1 (Weighting)

Best case:

```
union(0, 1);
union(1, 2);
…
union(n-2, n-1);
```

tree of height 1
`find()` operates in a constant time

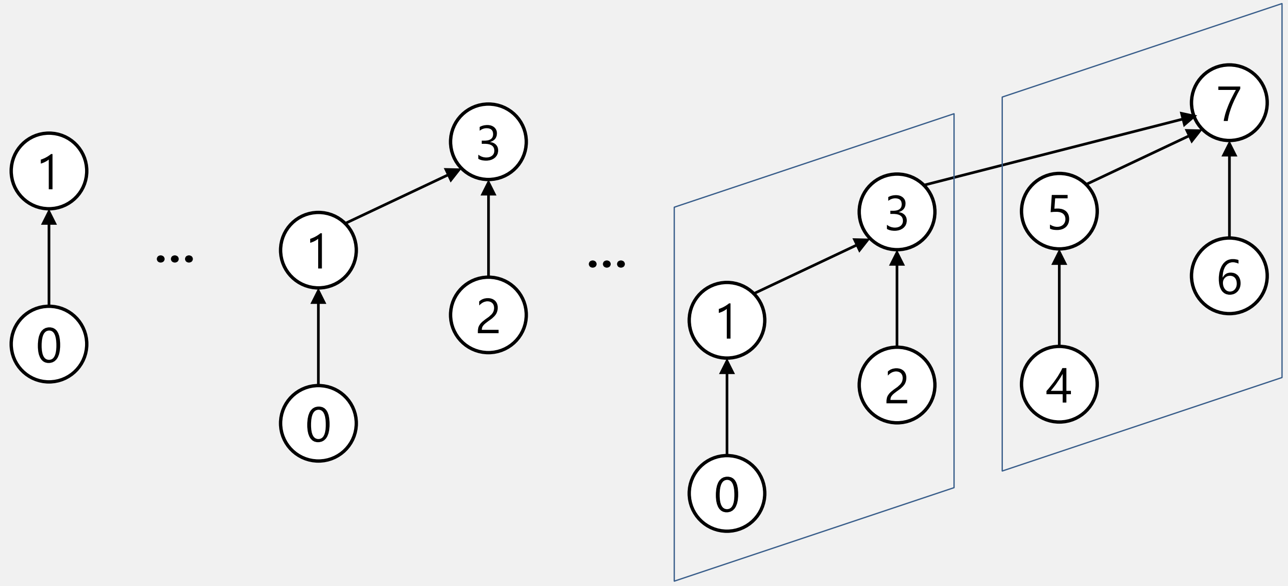


Improvement 1 (Weighting)

Worst case:

- Every union merges two sets with the same size

```
union(0, 1);
union(2, 3);
…
union(n/2, n-1);
```

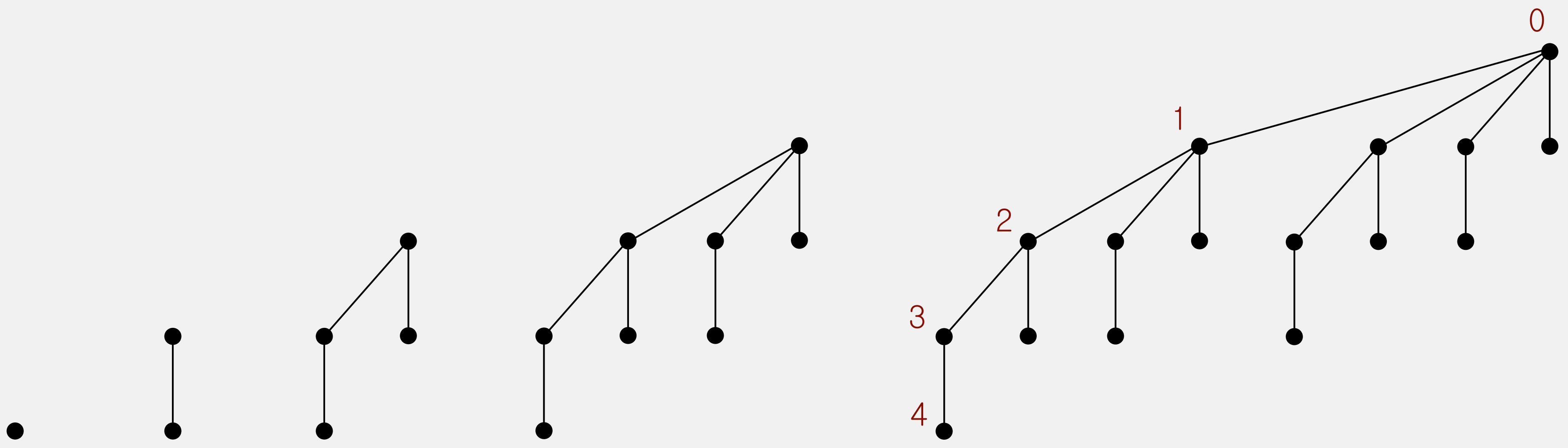


Improvement 1 (Weighting)

Worst case:

- Every union merges two sets with the same size

```
union(0, 1);
union(2, 3);
…
union(n/2, n-1);
```



depth of any node is at most $\lg n$

($\lg n = \log_2 n$)

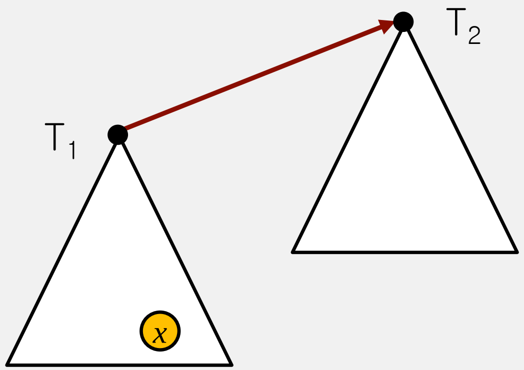
Improvement 1 (Weighting)

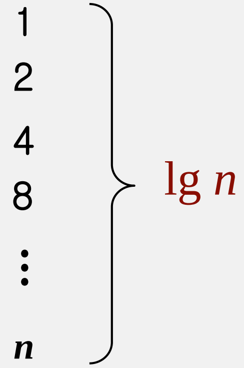
Running Time:

- Find: takes time proportional to depth of a node p.
- Union: takes constant time, given roots.

Proposition: Depth of any node x is at most $\lg n$.

- Increases the depth of a node x by 1 when
  - tree T_1 containing x is merged into another tree T_2
  - the size of T_1 and T_2 are same.





Improvement 1 (Weighting)

Running Time:

- Find: takes time proportional to depth of a node p.
- Union: takes constant time, given roots.

Proposition: Depth of any node x is at most $\lg n$.

algorithm	init	find	union	in–same–set
UF-Weighting	n	$\lg n$	$\lg n$	$\lg n$

Improvement 2 (Path Compression)

`find()` with path compression:

- makes every node between x and the root point to the root.

```
int find(int x)
{
    int parent;
    int root = x;
    while(pt[root] != root)
        root = pt[root];

    while(pt[x] != root)
    {
        parent = pt[x];
        pt[x] = root;
        x = parent;
    }

    return root;
}
```

two-pass (two-loops) implementation

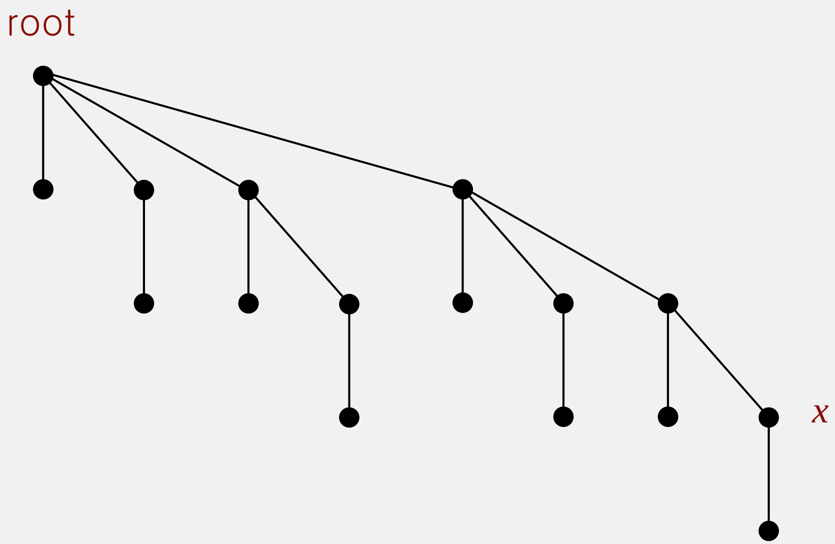
find root node of a query node x

makes every node between x and the root node to the root

Improvement 2 (Path Compression)

`find()` with path compression:

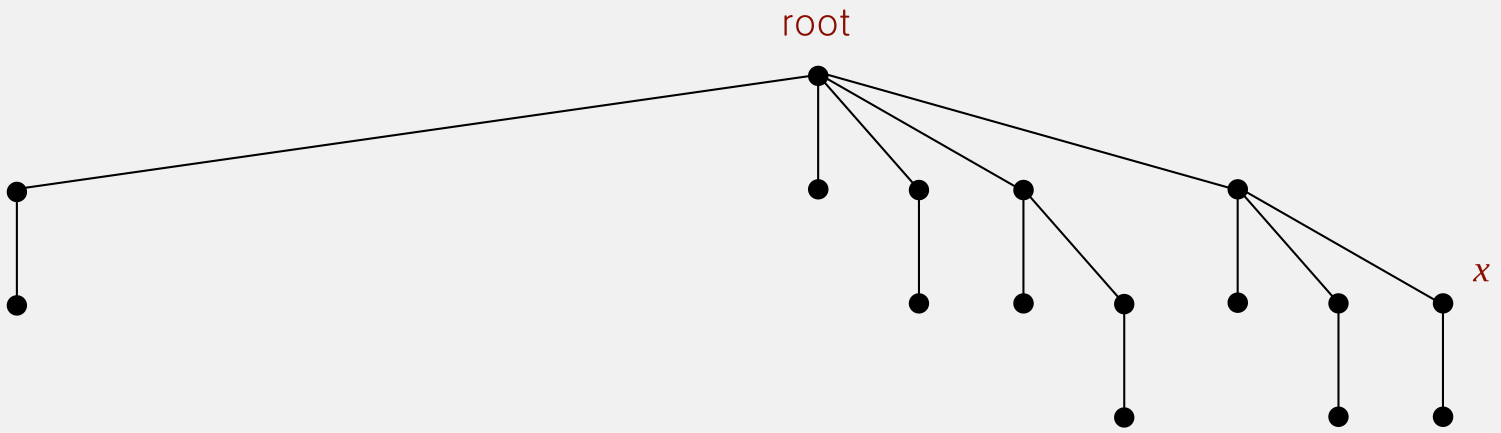
- makes every node between x and the root point to the root.



Improvement 2 (Path Compression)

`find()` with path compression:

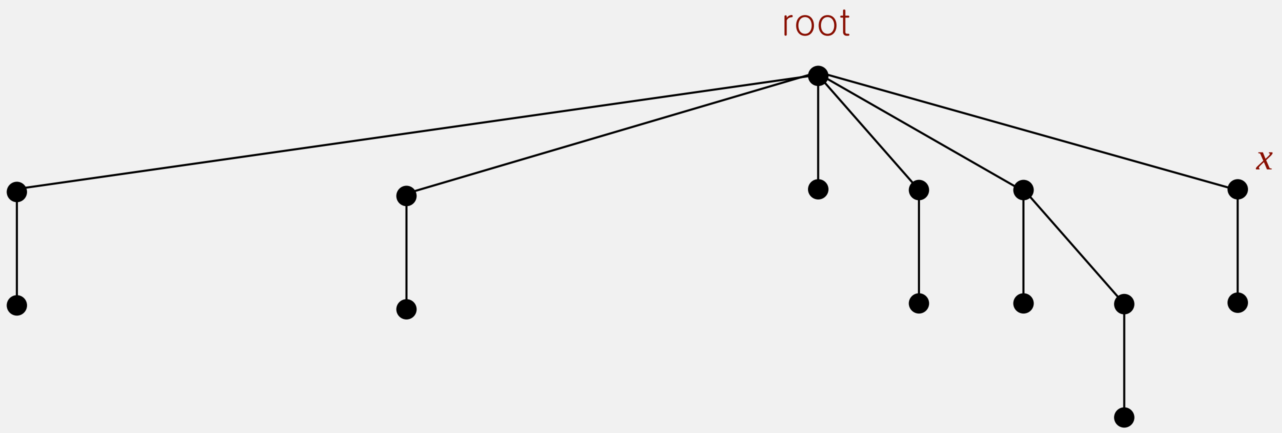
- makes every node between x and the root point to the root.



Improvement 2 (Path Compression)

`find()` with path compression:

- makes every node between x and the root point to the root.



Improvement 2 (Path Compression)

`find()` with path compression:

- makes every node between x and the root point to the root.

```
int find(int i)
{
    while(pt[i] != i)
    {
        id[i] = id[id[i]];
        i = pt[i];
    }
    return i;
}
```

one-pass (one-loop) implementation

make every other node in path point to its grandparent

Improvement 2 (Path Compression)

Analysis (Amortized):

- [Hopcroft–Ullman, Tarjan]
  - Starting from an empty data structure, any sequence of m union–find operations on n elements makes $\leq c\ (n + m \lg^* n)$ array accesses.
    - Analysis can be improved to $n + m\ \alpha(m, n)$

n	$\lg^* n$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

$\lg^* n$: iterated $\lg$ function

- number of times the $\lg$ must be iteratively applied before the result is less than or equal to 1

$\alpha(n)$: inverse Ackermann function

- grows extraordinarily slowly
- 4 or less for any n that can actually be written in the physical universe
- almost constant

Union-Find Data Structure:

Analysis:

- Number of elements n can be huge.
- Number of union/find operations m can be huge.
- Union/find operations may be intermixed.

algorithm	Worst-case
UF-Weighting	$n + m \lg n$
UF-Weighting, Path compression	$n + m \lg^* n$ $n + m\ \alpha(m, n)$