Disjoint Sets

What are Disjoint Sets?

- A set with no duplicate items and each item only belongs in one set.
- A set is a collection of items.
- EG:
  - *1* = {*a*, *c*, *d*} (Items *a*, *c* & *d* belong to set *1*)
  - *2* = {*b*, *e*} (Items *b* & *e* belong to set *2*)
- Used to solve Union-Find Problems

Data Structure

A disjoint set data structure support the following operations:

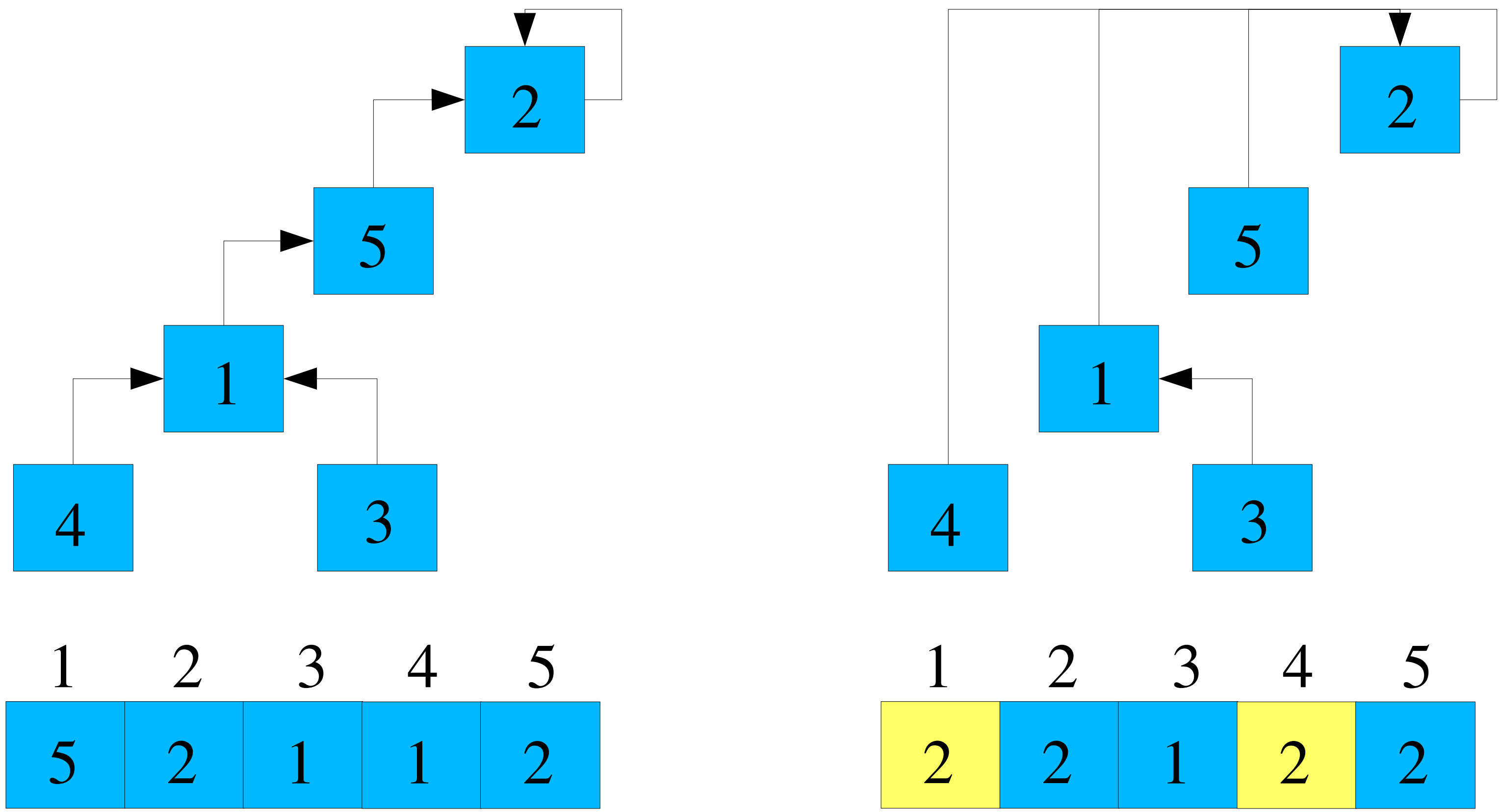
- New-Set (x) Creates a new set $\{x\}$
- Union (x, y) Combines the set that x is in with the set that y is in
- Find-Set (x) Finds which set x is in. (Must obey Find-Set (x) = Find-Set (y))

Implementation

- Array with size max item
  - Array [x] points to another item in set. If it points to itself, then x is the value of the set.
  - If items are text, you can use a hash table. Key = item & value = set
- Make-Set (x): array[x] = x
- Find (x): Find (array [x]) until array [x] = x
- Union (x, y): array [find (x)] = array [find (y)]

Optimizing Find

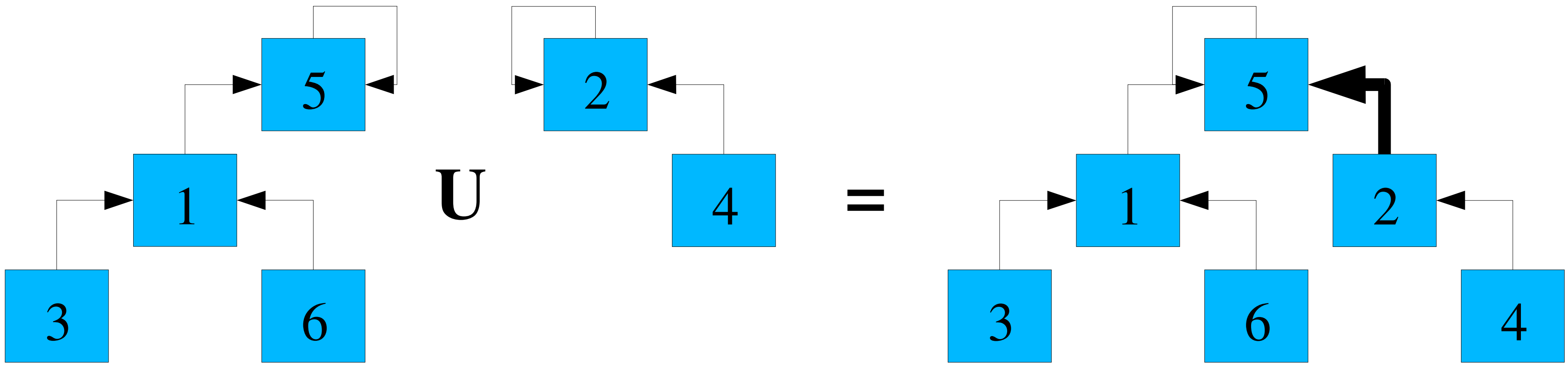
- The find operation is 0 (log *n*). n = size of set
- To speed up operation, use “compression”.
  - Caches the set, so future calls are O (*1*)



Optimizing Unions

- Unions combine sets.
- Union (*x*, *y*) causes *x*'s root to point to *y*'s root
- To minimize depth of trees, we store the depth of a tree, and add the shallower tree to the root of the deeper tree.
- If $depth_x = depth_y$, choose any root as the new root and increase the new root's depth by 1.
- Union's efficiency is O (*log* n), but on average it is O (1).

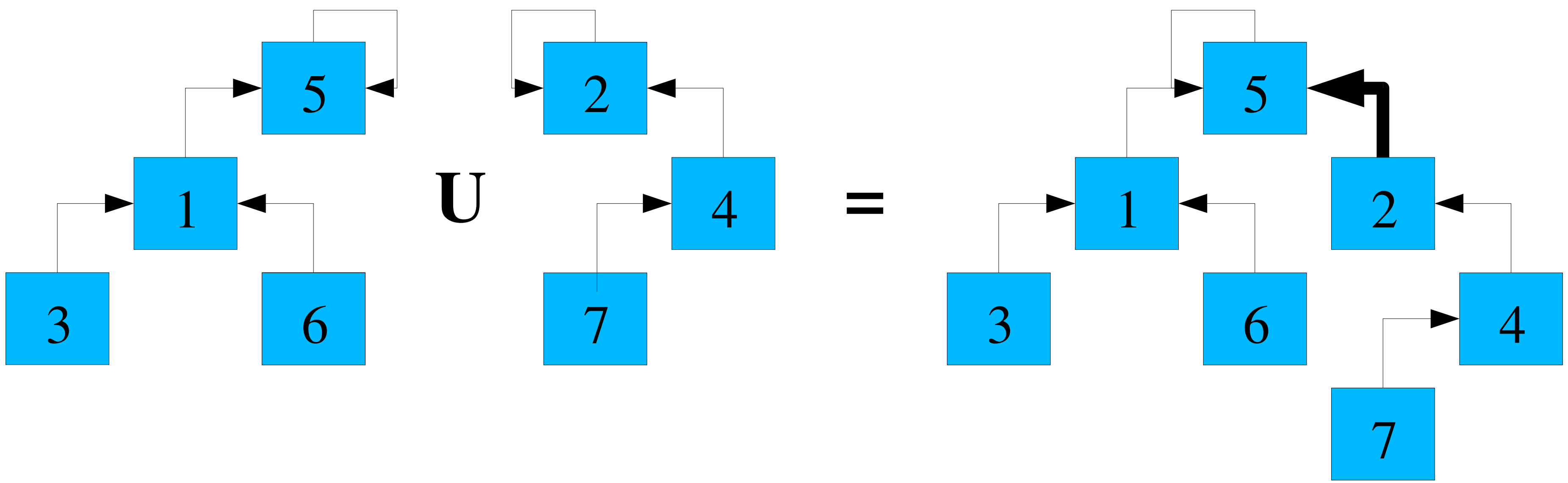
Union Example



	1	2	3	4	5	6
Sets Array	5	2	1	2	5	1
Depth Array	1	1	0	0	2	0

	1	2	3	4	5	6
Sets Array	5	5	1	2	5	1
Depth Array	1	1	0	0	2	0

Union Example 2



Sets Array

1	2	3	4	5	6	7
5	2	1	2	5	1	4
1	1	0	0	2	0	0

1	2	3	4	5	6	7
5	5	1	2	5	1	4
1	1	0	0	3	0	0

Depth Array

Pseudocode

Array *sets*, *depth* with size MAX

```
New-Set (x) :   sets [x] = x
                depth [x] =  0


Find-Set (x):   if x not = sets [x] then
                    sets [x] = Find-Set (x)
                endif
                return sets [x]
```

Pseudocode Cont.

```
Union (x, y):   x = Find-Set (x)
                y = Find-Set (y)
                if depth [x] > depth [y] then
                    sets [y] = x
                endif
                else then
                    sets [x] = y
                    if depth [x] = depth [y] then
                        depth [y] = depth [y] + 1
                    endif
                endelse
```

Other Data Structures

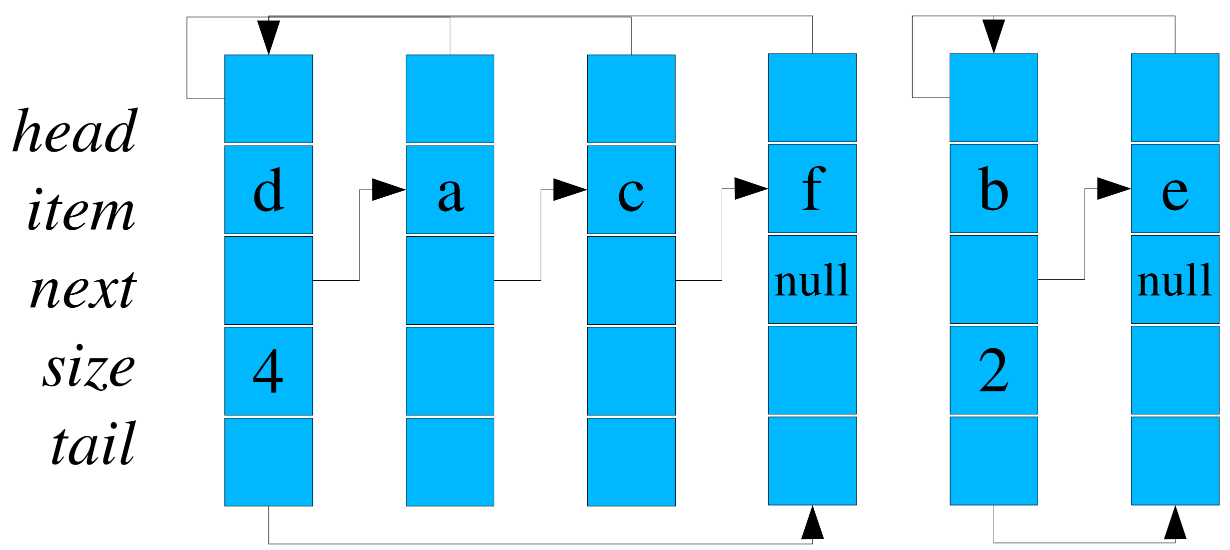
- Arrays are static
- Dynamic Structures:
  - Linked List
  - Disjoint-Set Forest

Linked List

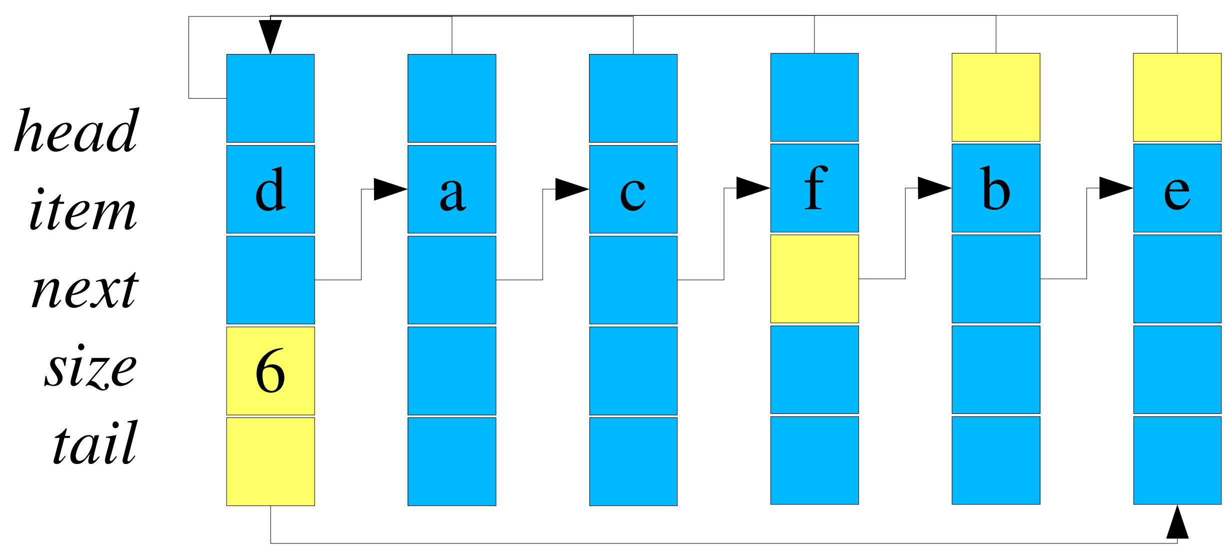
- Items have fields: *head*, *tail*, *next*, *size*
- Find-Set (x) returns *head* [x]
- New-Set (x) *head* & *tail* = x; *next* =*null; size = 1*
- Union (x, y):

```
x = Find-Set (x)
y = Find-Set (y)
if size[x] < size[y] then x <-> y endif
next [tail[x]] = y
tail [x] = tail [y]
size [x] = size [x] + size [y]
while y not = null do
    head [y] = x
    y = next [y]
endwhile
```

Linked List Representation



Union (a, b) =



Disjoint Set Forest

- Each set is a tree with the root representing the set.
- Items have fields: *parent*, *depth*.
- Slightly modify code used for arrays to use the item's fields instead.

Uses in Graph Theory

Graph Connectivity:

- If the vertices are items and an edge represents a union, x will be connected to y if Find-Set (x) = Find-Set (y)
- If you are constantly checking connectivity (ie: Kruskal), using Find-Set ($O(1)$) is more efficient than DFS ($O(n)$).