## Disjoint Set data structure

→ Many times the efficiency of an algorithm depends on the data structure used in the algorithm.

→ A wise choice in the structure used in solving a problem can reduce the time of execution

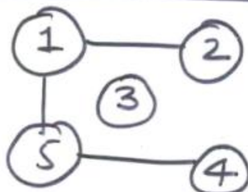
## BIG EXAMPLE (1)

" In a room there are N persons, and we will define two persons are friends if they are directly or indirectly friends"

[take example of any social networking site like orkut or friendster]

A group of friends is a group of persons where any two persons in the group are friends. Given the list of persons that are directly friends find the number of groups of friends and number of persons in each group"

like




{1, 2, 4, 5} {3}

Two groups.

→ direct or indirect friends.

(1)

→ {Dynamic sets}

*→ In mathematics, a set is understood as a collection of clearly distinguishable entities (called elements)

*→ once defined the set does not change

*→ In computer science, a dynamic set is understood to be a set that can change over time by adding or removing elements

→ {Abstract data type: Disjoint sets}

State: Collection of disjoint dynamic sets.

↳ the No. of sets and their composition can change but they must always be disjoint

↳ Each set has a representative element that serves as the name of set.

representative Element

e.g. $S = \{a, b, c\}$ can be represented by a

Operations -:

Make_set(x): creates a singleton set $\{x\}$ and adds it to the collection of sets

Union(x,y): replaces x's set $S_x$ and y's set $S_y$ with $S_x \cup S_y$

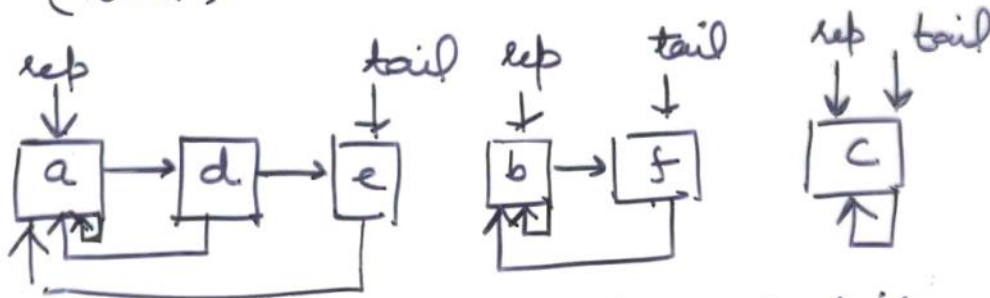
find_set(x): return (a pointer to) the representative of the set containing x

# Solution to Big Example (1)

```
Read N
for (each person x from 1 to N) make_set(x)
for(each pair of friends (x,y))
    if ( find_set(x) != find_set(y))
        union(x,y)
```

## linked list representation -:

- → Store the set elements in a linked list. Each list node has a pointer to the next list node
- → the first list node is the set representative <u>rep</u>
- → Each list node also has a pointer to the set representative <u>rep</u>
- → Keep external pointer to first list node (rep) and last list node (tail)




<u>Make_set(x)</u>: make a new linked list containing just a node for x ( $O(1)$ time )

<u>find_set(x)</u>: given (pointer to) linked-list node containing x, follow rep pointer to head of list ( $O(1)$ time )

<u>Union(x,y)</u>: append list containing x to end of list containing y and update all rep pointers in the old list of x to point to the rep of Y ( $O($ sizeof (x's old list) $)$ )

## Time analysis

Consider:

Makeset_set($x_1$), make_set($x_2$) ... make_set($x_n$)
union($x_1, x_2$), union($x_2, x_3$) ... union($x_{n-1}, x_n$)

$m = 2n-1$ operations

Total time $= O(n^2)$ beacause union update $1, 2, \ldots n-1$ elements

Remedy: Linked list with weighted union.

→ Always append smaller list to larger list

→ Need to keep count of number of elements in the list (weight) in rep node.

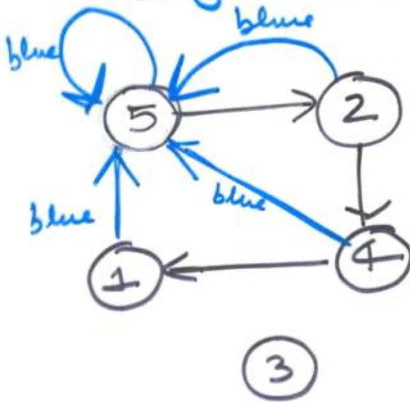
→ How many times must the rep pointer for an arbitrary node x be updated

*→ first time the rep pointer of x is updated, the new set has at least 2 elements

*→ the second time rep pointer of x is updated, the new set has at least 4 elements

*→ max. size of set $= n$.
rep element of X may be updated at most $\log n$ times. → Total time for all unions $= n\log n$

# Implementation of solution of Big Example (1) Through Linked list

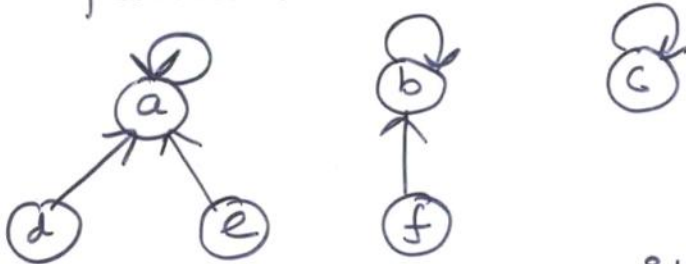



← Blue arrow : pointer to representative element

← black arrow : pointer to next element in the list

# Disjoint Sets in tree representation
(Disjoint set forest)

→ Use a collection of trees, one per set
→ the rep is the root
→ Each node has a pointer to it's parent in the tree




**make_set** : make a tree with one node
$O(1)$ time

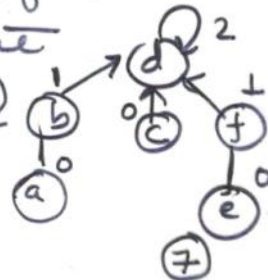
**find_set** : follow parent pointer to root
$O(h)$ time $h$ = height of tree

**Union(x,y)**: make the root of x's tree a child of the root of y's tree
$O(1)$ time

→ with a weighted union, smaller tree becomes child of larger tree

**Rank** : gives upper bound on the height of tree
→ It's approximately the log of the number of nodes in the tree

**Ex** ms(a), ms(b), ms(c), ms(d), ms(e), ms(f)
U(a,b), U(c,d), U(e,f), U(a,c), U(a,e)



make_set(x):

parent(x) := x
rank(x) := 0 // used for weighted union.

union (x,y)

r := find_set(x) ; s = find_set(y)
if rank(r) > rank(s) then parent(s) := r
else parent(r) := s
if rank(r) = rank(s) then rank(s)++

find_set (x)

if x ≠ parent(x) then
parent (x) := find_set (parent(x))
return parent(x)

Unroll recursion:

First follow parent pointers up the tree
Then go back down the path, making every node on the path a child of root

# Implementation of solution of Big-Example(1) using trees

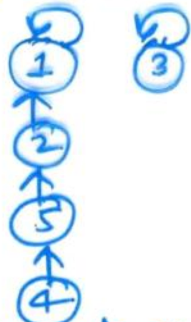
→ step 1: Nobody is anybody friend

→ step 2 ÷ 1 and 2 are friends union(1,2)

MODIFIED

→ step3 –: 5 and 4 are friends, union(5,4)

→ step4 –: 5 and 1 are friends union(5,1).



Two heuristics:
- union by rank
- path compression

each node x keep an integer value rank[x]
rank[x] = bigger than or equal to no. of edges in the longest path between node x and sub-leaf.

Let P[x] = parent of x.

make_set(x): P[x] = x; rank[x] = 0;

union(x,y): Px = find_set(x); Py = find_set(y)
if (rank[Px] > rank[Py]) P[Py] = Px
Else P[Px] = Py.
if (rank[Px] == rank[Py]) rank[Py] = rank[Py]+1