DATA STRUCTURES FOR DISJOINT SETS

→ A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.
→ Each set is identified by a representative, which is a member of the set.
→ If we have two sets S_x and S_y, $x \neq y$ such that
$S_x = \langle 3, 4, 5, 6, 7 \rangle$
$S_y = \langle 1, 2 \rangle$
then these sets are called disjoint sets as there is no element common in both sets.

DISJOINT-SET OPERATIONS:

In dynamic-set implementations, an object 'x' represents each element of a set.

(1) MAKE-SET(x): It creates a new set whose only member is x (i.e. the representative of that set). Hence, x is not a member of any other set. (This is known as disjoint property)

(2) UNION(x, y):
Let 'x' be the representative of set S_x, and 'y' be the representative of set S_y.
Both S_x and S_y are disjoint sets.
The representative of resulting set is any member of $S_x \cup S_y$ (i.e. either from S_x or S_y)
→ Example:
$S_x = \{x, z\}$ → Here x is representative of set S_x.
$S_y = \{a, y, b\}$ → Here y is representative of set S_y.
$S = S_x \cup S_y$
$S = \{x, z, a, y, b\}$

(3) FIND-SET(x): It returns a pointer to the representative of the unique set containing x.
→ Example:
FIND-SET(x) → x
FIND-SET(y) → y
FIND-SET(z) → x
FIND-SET(a) → y
FIND-SET(b) → y

APPLICATIONS OF DISJOINT-SET DATA STRUCTURES

→ One applications of disjoint-set data structure is determining the connected components of an undirected graph.

→ The procedure CONNECTED-COMPONENTS of disjoint-set operations is to compute the connected components of a graph.

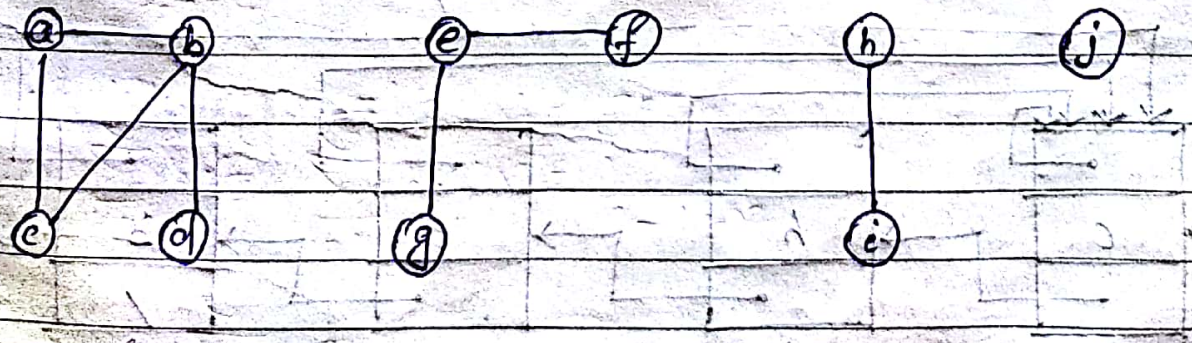
→ The procedure SAME-COMPONENT gives two vertices are in the same connected component.

→ The set of vertices of a graph G is denoted by V[G]. The set of edges is denoted by E[G].

```
CONNECTED-COMPONENTS(G)
1. for each vertex v ∈ V[G]
2.     do MAKE-SET(v)
3. for each edge (u,v) ∈ E[G]
4.     do if FIND-SET(u) ≠ FIND-SET(v)
5.         then UNION(u,v)
```

```
SAME-COMPONENT(u,v)
1. if FIND-SET(u) = FIND-SET(v)
2.     then return TRUE
3. else return FALSE
```

→ find the connected components in given graph



A graph with four connected components :

{a, b, c, d}, {e, f, g}, {h, i}, {j}

Edge processed	collection of disjoint sets
Initial sets	{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}
(b,d)	{a} {b,d} {c} {e} {f} {g} {h} {i} {j}
(e,g)	{a} {b,d} {c} {e,g} {f} {h} {i} {j}
(a,c)	{a,c} {b,d} {e,g} {f} {h} {i} {j}
(h,i)	{a,c} {b,d} {e,g} {f} {h,i} {j}
(a,b)	{a,b,c,d} {e,g} {f} {h,i} {j}
(e,f)	{a,b,c,d} {e,f,g} {h,i} {j}
(b,c)	{a,b,c,d} {e,f,g} {h,i} {j}

LINKED-LIST REPRESENTATION OF DISJOINT-SETS

→ Disjoint-set data structure can be implemented by a linked list to represent each set.

→ The first object in each linked list is known as representative.

→ Each object in the linked list contains

(1) A set member.

(2) A pointer to the next ~~object~~ set member.

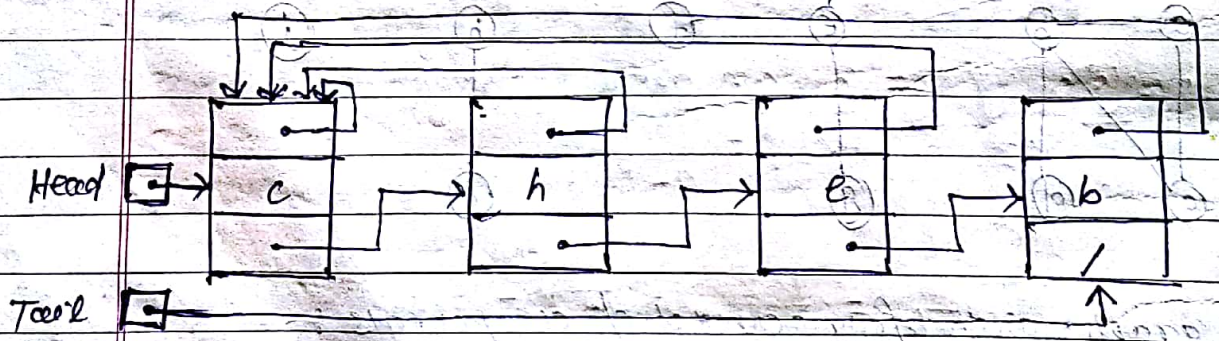
(3) A pointer back to the representative.

→ Each linked list contains two pointers:

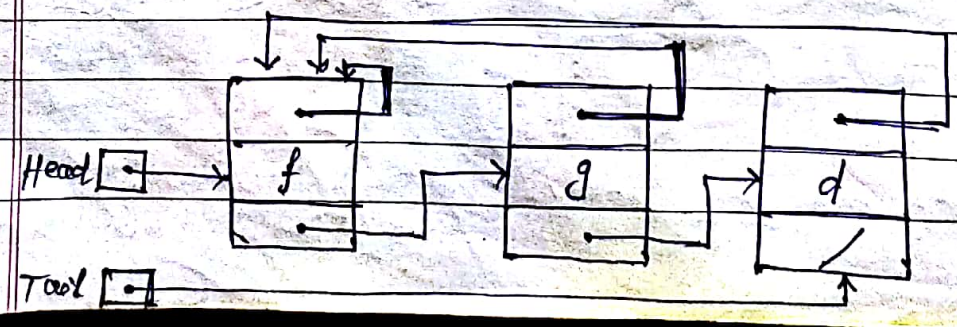
(1) Head: It keeps address of the representative.

(2) Tail: It keeps address of last object in the list.

→ Linked list representation of one set contains objects b, c, e, and h with c as its representative.

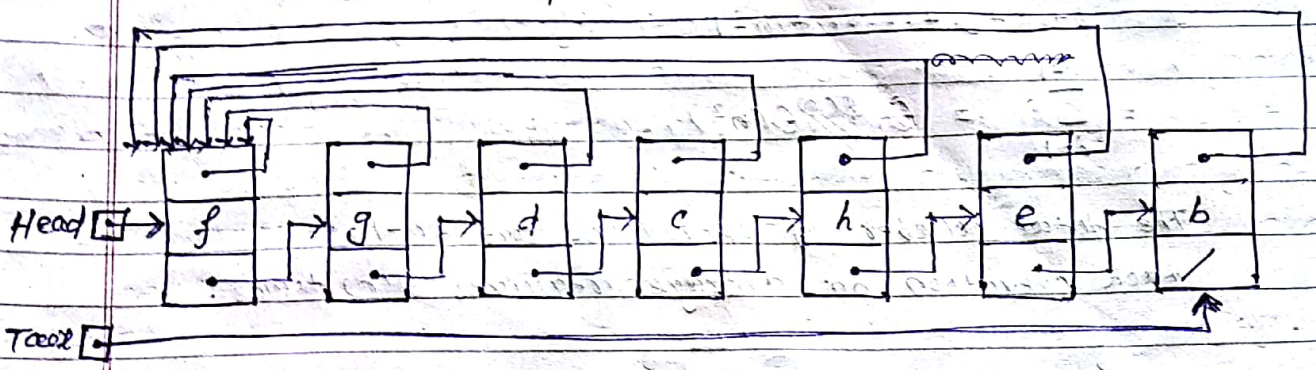


→ Linked list representation of another set contains objects d, f and g, with f as the representative.



96

→ UNION(e, g), the representative of the resulting set is f.



COMPLEXITY:

→ The linked list representation of both MAKE-SET and FIND-SET takes O(1) time.

→ MAKE-SET(x) creates a new linked list whose only object is x.

→ FIND-SET(x) returns the pointer from x back to the representative.

→ In UNION(x, y) operation, we append x's list onto the end of y's list. We use tail pointer for y's list to quickly find where to append x's list. The representative of the new set is the element that was originally the representative of set y.

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
⋮	⋮
UNION(x_{n-1}, x_n)	n-1

→ We have n objects such as $x_1, x_2, x_3, \ldots, x_n$. So, we execute the sequence of n MAKE-SET operations.

→ 'n' objects having n-1 UNION operations.

→ So, total operation = $2n$ = $n + n - 1 = 2n - 1$

→ 'n' MAKE-SET operations takes $\theta(n)$ time because each MAKE-SET operation takes 1 unit of time.

The time required to (n-1) UNION operations are:

$1 + 2 + 3 + \cdots + n - 1$

$= \sum_{i=1}^{n-1} i = \Theta(n^2)$

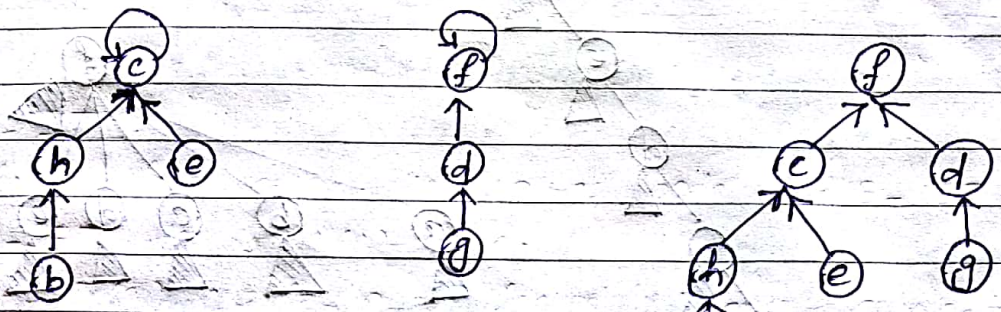
The total number of operations are $2n-1$.

Each operation on average requires $\Theta(n)$ times.

DISJOINT-SET FORESTS

→ For faster implementation of disjoint sets, we represent sets by rooted trees. Each node containing one member and each tree representing one set.

→ In a disjoint set forest, each member points only to its parent. The root node of the tree contains the representative and is its own parent.



set {b, c, e, h} with c as the representative

set {d, f, g} with f as the representative

UNION(e, g)

→ We perform three disjoint-set operations.

(1) A MAKE-SET operation simply creates a tree with just one node.

(2) A FIND-SET operation by following parent pointers until we find the root of the tree. The nodes visited on this path toward the root constitute the find path.

(3) A UNION operation causes the root of one tree to point to the point to the root of the other.

Heuristics to improve the running time:

(1) UNION BY RANK:

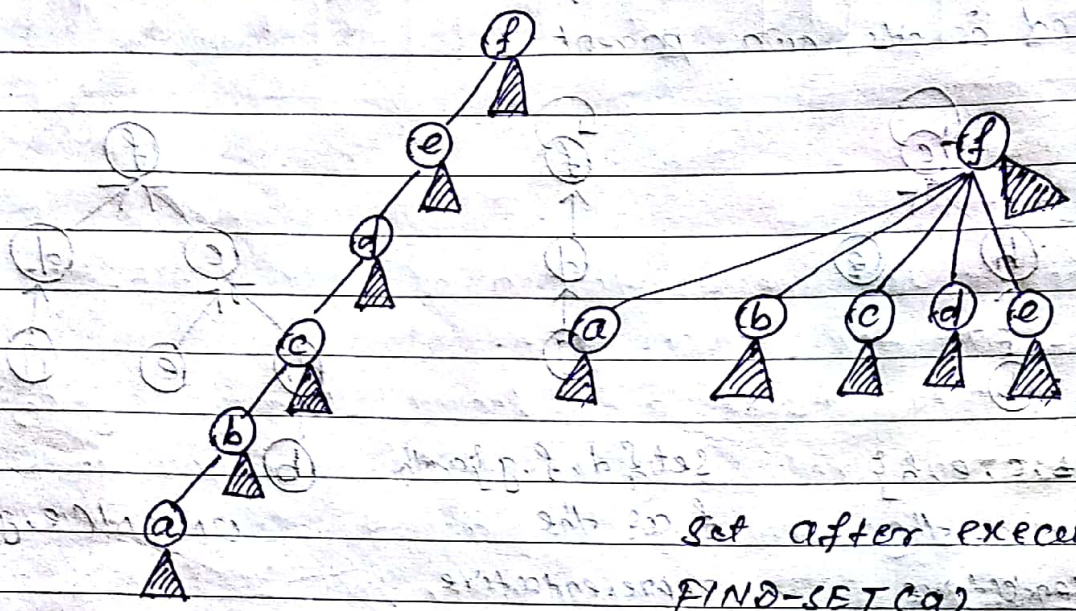
→ Make the root of the tree with fewer nodes point to the root of the tree with more nodes.

→ For each node we maintain a rank that is an upper bound on the height of the node.

→ In union by rank, the root with smaller rank is made to point point to the root with larger rank during a UNION operation.

(2) PATH COMPRESSION:

We use path compression during ~~find~~ FIND-SET operations to make each node on the find path point directly to the root. path compression doesnot change any ranks.



A tree representing a set prior to executing FIND-SET(a)

set after executing FIND-SET(a)

Pseudocode for Disjoint-set forests:

→ For each node x, we maintain the integer value rank[x], which is an upper bound on the height of x. i.e. Leaf node having rank 0.

→ When we apply UNION of two trees, there are two cases, depending upon the roots having equal rank or not.

→ If the roots have ~~equal~~ ~~rank~~ unequal rank, we make the root of higher rank the parent of the root of lower rank, but ranks themselves remain unchanged.

→ If the roots have equal ranks, we arbitrarily choose one of the roots as the parent and increment its rank.

MAKE-SET(x)

1. p[x] ← x
2. rank[x] ← 0

LINK(x, y)

1. If rank[x] > rank[y]
2. then p[y] ← x
3. else p[x] ← y
4. If rank[x] = rank[y]
5. then rank[y] ← rank[y] + 1

UNION(x, y)

1. LINK(FIND-SET(x), FIND-SET(y))

FIND-SET(x)

1. If x ≠ p[x]
2. then p[x] ← FIND-SET(p[x])
3. return p[x]