

1 Definition

- In mathematics, two sets are said to be disjoint sets if they have no elements in common. For example, $\{1, 2, 3, 5, 8\}$ and $\{4, 5, 6, 9\}$ are disjoint sets. The formal definition of disjoint set is $S = \{S_1, S_2, \dots, S_k\}$, $S_i \cap S_j = \emptyset$, $1 \leq i < j \leq k$.
where S is a collection of sets, and any two sets in S are nonoverlapping.
- **In computer science, a disjoint-set data structure is a data structure used to store and keep track of a set of elements partitioned into disjoint (non-overlapping) subsets.**
- In other words A disjoint-set data structure maintains a collection of disjoint dynamic sets $S = \{S_1, S_2, \dots, S_k\}$.
- Each set is identified by a representative, which is some member of the set.
- No matter how the representative is selected, we only care the representative fulfils the following:
 - a. It is different from other representatives
 - b. Each time we ask for the representative of a set, without modifying the set, we get the same answer (representative).
- For convenience, we choose one of the elements in the set as the representative. Figure 1 shows an example for the representative. We select the left most elements in each set as representative.

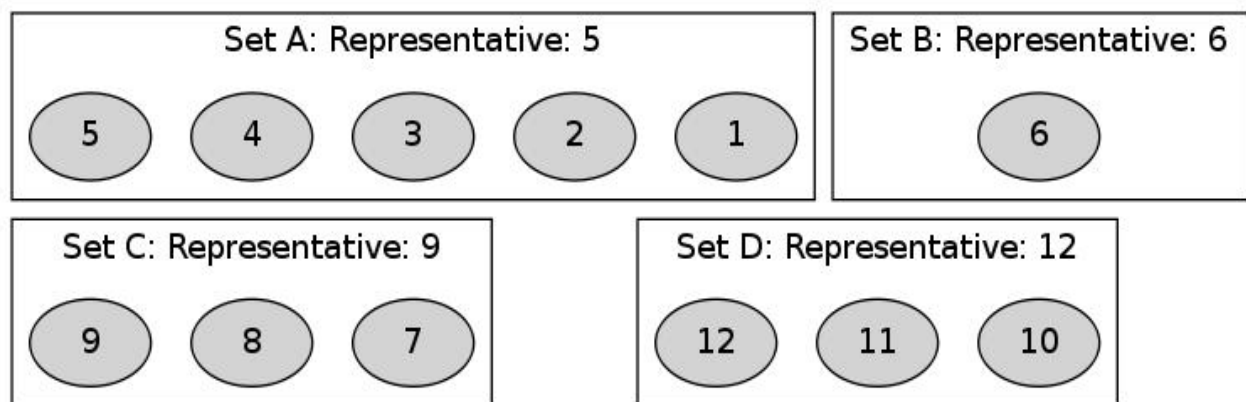


Figure 1: Disjoint and their representatives

1.1 Disjoint -Set operations

A disjoint-set data structure should support the following three operations:

- a) **MAKE-SET (x)** creates a new set whose only member is x . Since every elements should belong to one set, we create sets which own exactly one element at first.
- b) **FIND-SET (x)** returns the representative or a pointer to the representative of the set that contains element x . It is useful to determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- c) **UNION(x, y)** combine or unite or merge one set that contains x with other set that contains y into a single set. The two sets are assumed to be disjoint prior to the operation.

When we want to create a new disjoint sets with n elements, we need these three operations. At first we run the MAKE-SET operations n times to generate n disjoint sets which own exactly one element. Then we run the UNION operation a couple of times to merge sets and finally generate our disjoint sets data structure.

2 Linked-list Representation of Disjoint Set Data Structure (or List-based disjoint sets)

A simple way to implement disjoint-set data structure is to represent each set by a linked list. The idea is all elements in same set are linked together with a linked list. The representative is the first member of the linked list. Each object or element in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative. Each list maintains pointers *head*, to the representative, and *tail*, to the last object in the list. Within each linked list, the objects may appear in any order.

Let there are two disjoint sets $S_1 = \{b, c, e, h\}$ with c as representative and $S_2 = \{d, f, g\}$ with f as representative respectively. The representative of the resulting set is f .

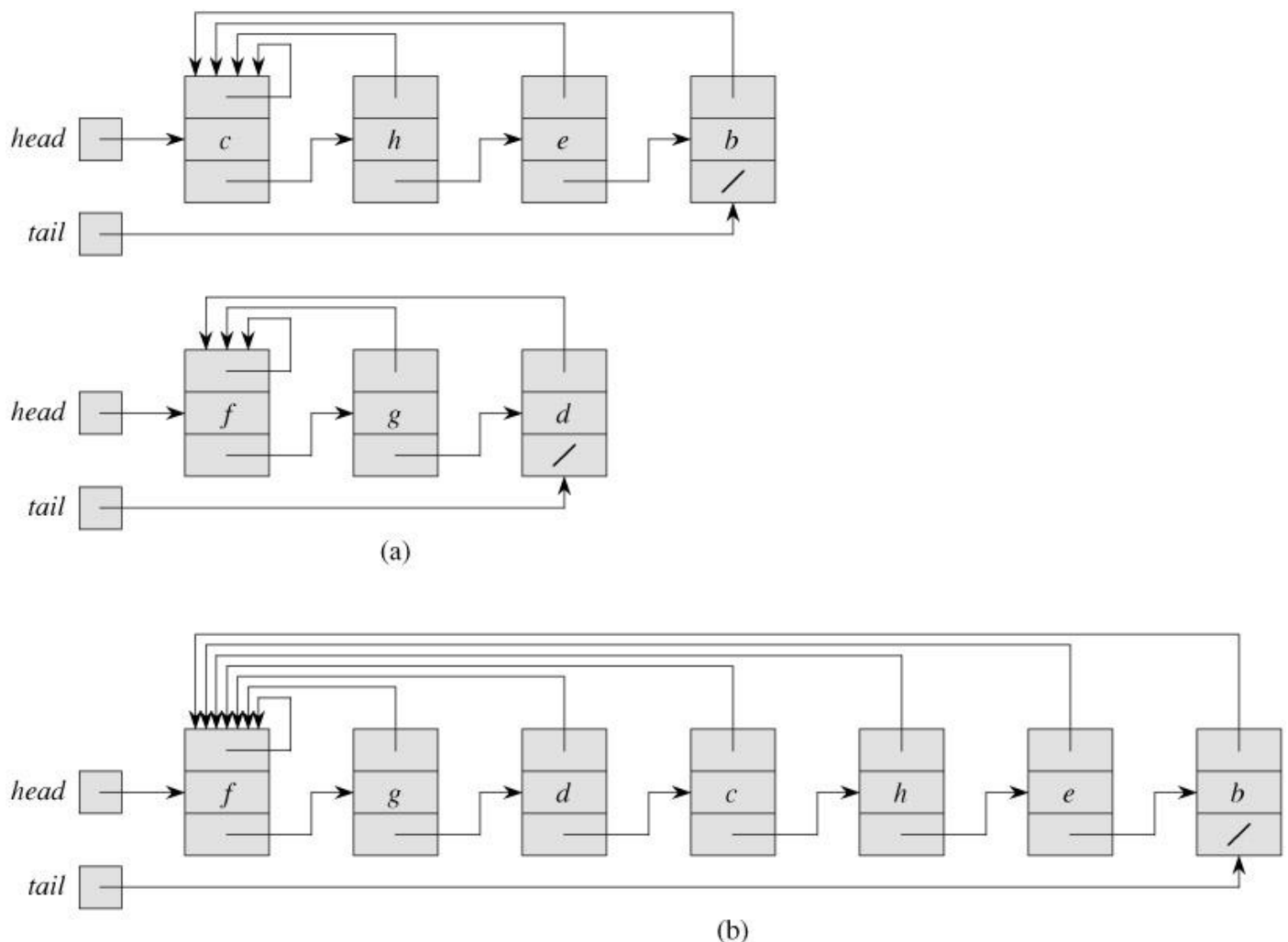


Figure 2: (a) Linked-list representations of two sets (b) An example of $UNION(b, g)$

2.1 Running Time Analysis

2.1.1 By using a simple implementation of UNION

- With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring $O(1)$ time. To carry out MAKE-SET(x), we create a new linked list whose only object is x . For FIND-SET(x), we just return the pointer from x back to the representative.
- The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET. The running time for UNION depends on the number of elements updated.

- Suppose that we have n elements/objects $x_1, x_2, x_3, \dots, x_n$. Suppose also, that the total number of MAKE-SET, UNION, and FIND-SET operations is m . We execute the sequence of n MAKESET operations followed by $n - 1$ UNION operations shown in below:

Total number of operations	Operation	Number of elements updated	Total
n	MAKE-SET(x_1)	1	$n = O(n)$
	MAKE-SET(x_2)	1	
	MAKE-SET(x_3)	1	
	
 MAKE-SET(x_n)	.. 1	
$n-1$	UNION(x_1, x_2)	1	$\sum_{i=1}^{n-1} i = O(n^2)$
	UNION(x_2, x_3)	2	
	UNION(x_3, x_4)	3	
	
 UNION(x_{n-1}, x_n)	.. $n-1$	

- The total number of operations is $m = n + (n-1) = 2n - 1$.
- The i^{th} UNION operations updates i elements, which means we always append the longest list to the one-element set.
- A sequence of $2n - 1$ operations on n objects that takes $O(n^2)$ time in worst case, and $O(n)$ time per operation on average, using the linked-list set representation and the simple implementation of UNION.

2.1.2 By using weighted-union heuristic

- There is some improvement by this method over simple implementation of union.
- An improvement that can be made on the UNION operation is to always append the smaller set to the larger, resulting in fewer updates of representative pointers.
- Theorem:** Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, take $O(m + n \lg n)$ time.

Proof:

We first compute an upper bound on the number of times the element's head link has been updated. An element x is selected. Since we always append shorter list to longer list, x is always in the shorter list. The first time when x needs updated its representation, the result list must have had a least 2 members. Similarly, the next time when x is updated, the new result list must have had at least 4 members. Continuing on, we find out that when the new result list have at least k members, it means x has been updated $\lg k$ times where $k \leq n$. So the total number of updated for $k = n$ elements is $O(n * \lg k) = O(n \lg n)$. Since the running time for MAKE-SET and FIND-SET are both $O(1)$, and there are $O(m)$ of these operations, the total time for the entire generation is thus $O(m + n \lg n)$.

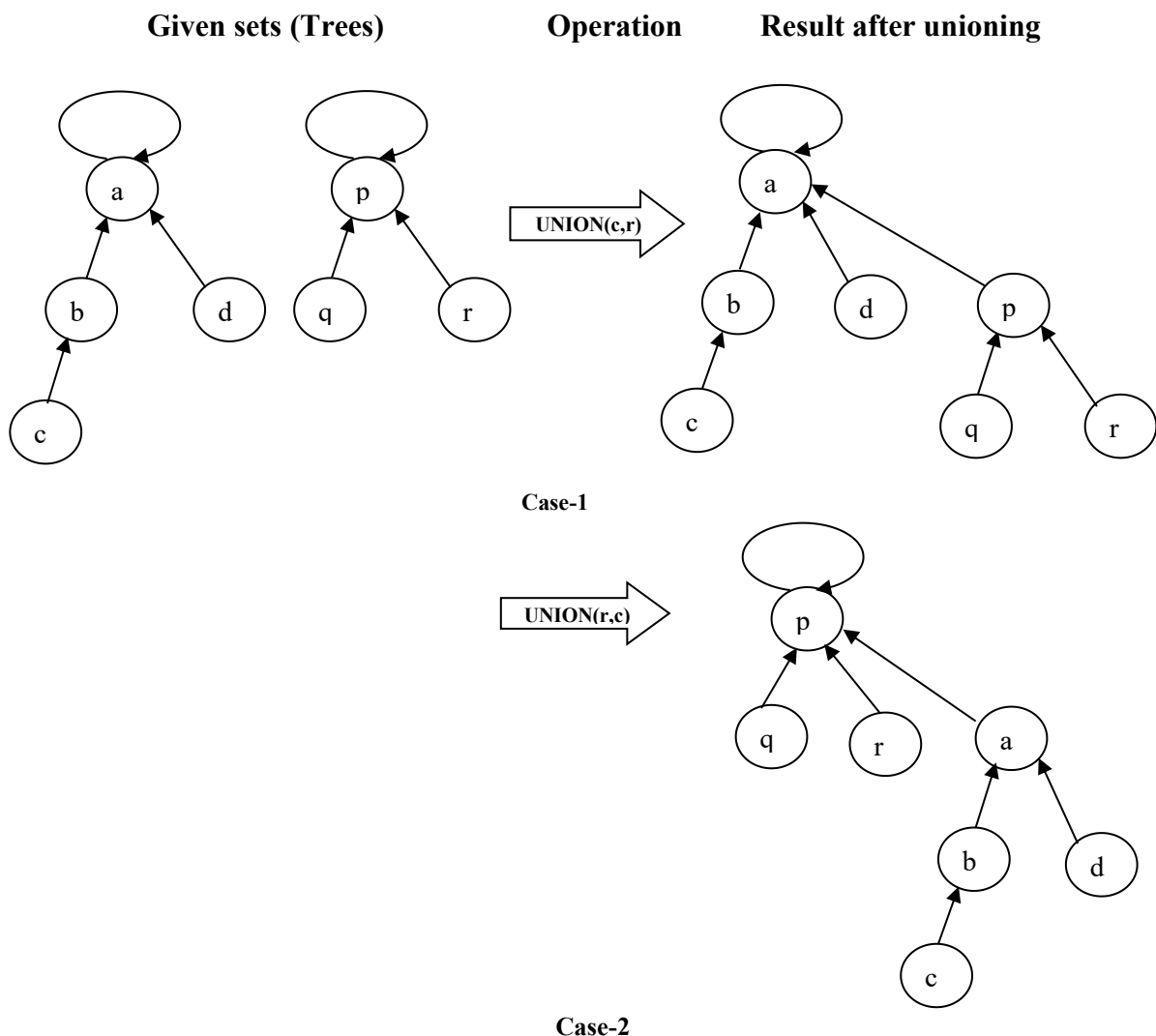
- To achieve faster performance, however, we will have to turn to a different data structure.

3 Tree-based disjoint sets (or Disjoint-set forests)

- In a disjoint-set forest, each set is stored as a tree. Each node represents a single element of the set and holds a pointer to its parent node, with the exception of root nodes, which points to itself. The root of a set's tree is used as its representative.
 - MAKE-SET (x)** - Create a new node. $O(1)$
 - FIND (x)** - Starting at x, traverse parent pointers up to the root. $O(n)$
 - UNION(x,y)** - Set root y's parent pointer to root x. $O(1)$
 - Here we first find out $\text{FIND}(x)$ and $\text{FIND}(y)$ which yields the roots of the trees containing x and y respectively. Let it be xRoot and yRoot respectively.

	Iterative version	Recursive version	
MAKE-SET(x) { parent[x] = x }	FIND-SET(x) { While (x \neq parent[x]) x \leftarrow parent[x] return x }	FIND-SET(x) { if (x == parent[x]) return x else return FIND- SET(parent[x]) }	UNION(x,y) { xRoot=FIND-SET(x) yRoot=FIND-SET(y) parent[yRoot] \leftarrow xRoot }
Time = $O(1)$	Time = $O(n)$		Time = $O(1)$

- A naive implementation, as outlined above, does not actually produce better results than the linked-list approach. However, we can make huge improvements on the tree-based structure.



- In case-1 figure, after UNION(c,r) operation, no depth increases which shows that later FIND operation will take same time as previous.
- In case-1 figure, after UNION(r,c) operation, depth of the tree increases which shows that later FIND operation may take more time than previous.
- The biggest problem we face with the naive implementation is the emergence of highly unbalanced trees due to the simplistic nature of the UNION operation as with this implementation we don't consider the depth of the trees.
- We can employ an optimization called UNION BY RANK to address this issue. It is analogous to the appending of the smaller set that we used with linked lists.

3.1 UNION BY RANK

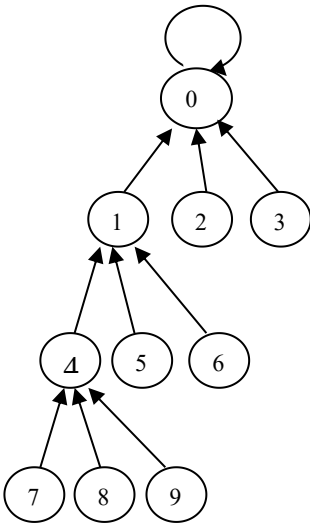
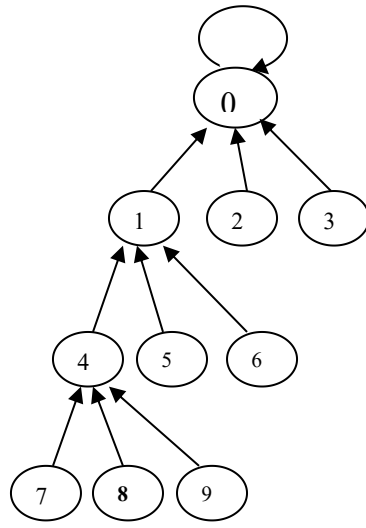
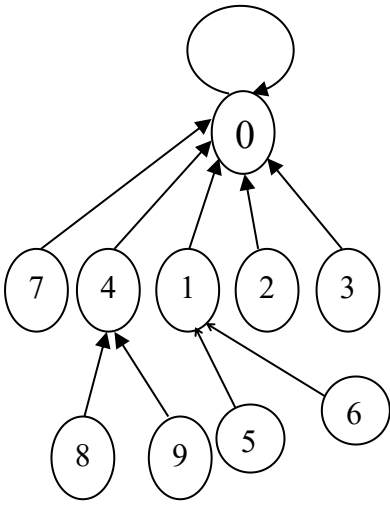
- In this heuristic when merging or unioning two sets (trees) together, we would like to attach the root of the smaller tree (tree with smaller depth) to the root of the larger tree (tree with larger depth).
- This requires us to maintain the depth of each tree, called rank of the tree.
- By keeping this rank the two roots can be compared and the one with a smaller rank can be attached to the root with a larger rank. After the union if the two roots being joined had equal ranks the new root nodes rank will be increased by 1.

MAKE-SET-RANK(x)	FIND-SET	UNION-RANK(x,y)
<pre>{ parent[x] = x rank[x] = 0 }</pre>	<p>same as above</p>	<pre>{ xRoot=FIND-SET(x) yRoot=FIND-SET(y) if (rank[xRoot] > rank[yRoot]) then parent[yRoot] ← xRoot else { parent[xRoot] ← yRoot if (rank[xRoot] = rank[yRoot]) then rank[yRoot] ← rank[yRoot] + 1 } }</pre>

- Using the technique of union by rank alone to improve disjoint-set forest performance produces an amortized run-time of $O(\log n)$ per operation.
- If we perform UNION BY RANK operation (to call the UNION_RANK(c,r) or UNION_RANK(r,c) procedure), on case-1 given trees, always it will give result same as the case-1 result. Analyze why? Because it compares the rank between the trees and the root of the higher rank tree is always be the parent and root of the lower rank tree is its child. In case same rank or rank of second tree is higher, second tree will be the parent and the first tree will be the child of it.

3.2 Path Compression

- The FIND method is simple, but we need one more trick to obtain the best possible speed.
- Suppose a sequence of UNION operations creates a tall tree, and we perform find repeatedly on its deepest leaf. Each time we perform FIND, we walk up the tree from leaf to root, perhaps at considerable expense. When we perform find the first time, why not move the leaf up the tree so that it becomes a child of the root? That way, next time we perform find on the same leaf, it will run much more quickly. Furthermore, why not do the same for every node we encounter as we walk up to the root?

Sl. no	Tree	Operation	After operation
1		<p>Perform the simple FIND operation on node 7 that is to call the following procedure</p> <p style="text-align: center;">FIND (7)</p>	<p>There will be no change in the structure of the tree.</p> <p>It will return the root node 0, which is the representative of the tree.</p>
2		<p>Perform the FIND operation with path compression on node 7 that is to call the following procedure</p> <p style="text-align: center;">FIND-PATH (7)</p>	<p>Here the structure of the given tree will change. With the FIND- PATH (7) walk, all the nodes on the path node 7 to root node 0 will be connected directly to the root.</p> 

- In the example above, FIND(7) walks up the tree from 7, discovers that 0 is the root, and then makes 0 the parent of 4 and 7, so that future FIND operations on 4, 7, or their children will be faster. This technique is called **"path compression."**
- The key observation is that in any FIND operation, once we determine the leader of an object x, we can speed up future FINDs by redirecting x's parent pointer directly to that leader. In fact, we can change the parent pointers of all the ancestors of x all the way up to the root; this is easiest if we use recursion for the initial traversal up the tree. This modification to FIND is called path compression.

- **FIND-SET WITH PATH COMPRESSION**

MAKE-SET Same as above	FIND-SET-PATH(x) { if (x ≠ parent[x]) then parent[x] ← FIND-SET(parent[x]) return parent[x] }	UNION Same as above
---------------------------	--	------------------------

All together all the tree based disjoint data structure algorithms are given below. Remember it.

	Iterative version	Recursive version	
MAKE-SET(x) { parent[x] = x }	FIND-SET(x) { While (x ≠ parent[x]) x ← parent[x] return x }	FIND-SET(x) { if (x == parent[x]) return x else return FIND-SET(parent[x]) }	UNION(x,y) { xRoot=FIND-SET(x) yRoot=FIND-SET(y) parent[yRoot]←xRoot }
MAKE-SET-RANK(x) { parent[x] = x rank[x] = 0 }	FIND-SET-PATH(x) { if (x ≠ parent[x]) then parent[x] ← FIND-SET-PATH(parent[x]) return parent[x] }		UNION-RANK(x,y) { xRoot=FIND-SET(x) yRoot=FIND-SET(y) if (rank[xRoot] > rank[yRoot]) then parent[yRoot] ← xRoot else { parent[xRoot] ← yRoot if (rank[xRoot] = rank[yRoot]) then rank[yRoot] ← rank[yRoot] + 1 } }
If the union by rank and path compression both can be combined then only changes will happen in the first two lines of UNION-RANK(x,y), the rest part will be as usual.			
xRoot=FIND-SET-PATH(x) yRoot=FIND-SET-PATH(y)			

4 Applications of Disjoint-set Data Structure

- a) Keeping track of connectivity on an undirected graph
 - Is there a path between these two nodes?
 - Would adding an edge between these two nodes produce a cycle?
- b) Separate an undirected graph into connected components
- c) Kruskal's algorithm for finding minimal spanning trees

QUESTIONS AND ANSWERS

Q1. Write a non-recursive/iterative version of FIND-SET with path compression.

[5]

Solution

Algorithm

```
FIND-SET-PATH(x)
{
    y ← parent[x]
    parent[x] ← x

    while( x ≠ y )
    {
        z ← parent[y]
        parent[y] ← x
        x ← y
        y ← z
    }

    root ← x
    y ← parent[x]
    parent[x] ← x

    while( x ≠ y )
    {
        z ← parent[y]
        parent[y] ← root
        x ← y
        y ← z
    }

    return root
}
```

The first while loop traverses the path from x to the root of the tree while reversing all parent pointers along the way. The second while loop returns along this path and sets all parent pointers along the way to point to the root.

Q2. Give a simple proof that operations on a disjoint-set forest with union by rank but without path compression run in $O(m \lg n)$ time.

Solution

Clearly, each MAKE-SET and UNION operation takes $O(1)$ time. Because the rank of a node is an upper bound on its height, each find path has length $O(\lg n)$, which in turn implies that each FIND-SET takes $O(\lg n)$ time. Thus, any sequence of m MAKE-SET, UNION, and FIND-SET operations on n elements takes $O(m \lg n)$.

Q3. State and explain union by rank algorithm with a suitable example.

Solution: Refer the above study material (page-5)

Q4. State and explain weighted-union heuristic theorem.

Solution: Refer the above study material (page-3)

Q5. What do you mean by FIND-SET(x). Explain.

Solution: Refer the above study material (page-1)

=====+++=Any error is found mail to anilkumarswain@gmail.com=+++=====