

## **Disjoint Set ADT**

-The implementation of disjoint-set data structure using trees, particularly with the union by rank method, offers several advantages over other data structures like arrays

### **Efficient Union and Find Operations**

-In a tree-based implementation, the union operation is typically faster compared to array-based implementations.

-By linking smaller trees to larger ones based on their rank, the height of trees is kept low, resulting in faster find operations as well.

### **Balanced Trees**

-Union by rank ensures that the resulting trees remain balanced.

-Balanced trees prevent degenerate cases where the tree becomes essentially a linked list, ensuring that find operations remain efficient even in worst-case scenarios.

### **Optimized Find Operation**

-The find operation in a tree-based disjoint-set structure has a time complexity of  $O(\log n)$  in the worst case, where  $n$  is the number of elements.

-This is because path compression is applied during the find operation, which flattens the tree structure, reducing the height of the tree.

### **Memory Efficiency**

-Tree-based implementations typically require less memory compared to array-based implementations, especially when dealing with large sets of elements.

-This is because only the parent pointer, rank, and possibly some additional metadata are stored for each element, rather than allocating memory for an entire array.

### **Scalability**

-Tree-based implementations scale well with the number of elements in the disjoint sets.

-Even for large sets of elements, the time complexity of union and find operations remains logarithmic, making tree-based implementations suitable for a wide range of applications.

## **Flexibility**

-Tree-based disjoint-set implementations offer flexibility in terms of the underlying data structure.

-Different variations, such as using path compression or weighted union, can be applied based on the specific requirements of the application.

Overall, tree-based implementations with union by rank strike a balance between efficiency, memory usage, and scalability, making them a popular choice for implementing disjoint-set data structures in various algorithms and applications.

### **Algorithm 1: Creating a set**

**Input:** capacity, nodes(array), data (integer).

**Output:** newnode (pointer to a node).

1. Create a newnode.
2. If capacity is lesser than or equal to data
  - a.  $\text{new\_capacity} \leftarrow \text{data} + 1$  and dynamically resize the internal array (nodes) to accommodate the new data.
3. Initialise i to 0
4. Repeat till i is capacity
  - a.  $\text{new\_nodes}[i] \leftarrow \text{nodes}[i]$
  - b. Increment i
5. Initialise i to capacity
6. Repeat till i is new\_capacity
  - a.  $\text{new\_nodes}[i]$  is NULL
7. Delete nodes
8.  $\text{nodes} \leftarrow \text{new\_nodes}$
9.  $\text{capacity} \leftarrow \text{new\_capacity}$
10. If nodes[data] is NULL
  - a. Delete newnode
  - b. Return NULL
11.  $\text{newnode} \rightarrow \text{data} \leftarrow \text{data}$
12.  $\text{newnode} \rightarrow \text{parent} \leftarrow \text{newnode}$
13.  $\text{newnode} \rightarrow \text{rank} \leftarrow 0$ 
  - a. If nodes[data] is NULL
    - i.  $\text{nodes}[\text{data}] \leftarrow \text{newnode}$
  - b. Otherwise
    - i. Delete newnode
14. Return newnode

### **Algorithm 2: Finding root of the tree by path compression**

**Input:** capacity, nodes(array), ele (integer).

**Output:** root of the tree.

1. If ele lesser than 0 or ele greater than or equal to capacity or nodes[ele] is NULL

- a. Return -1
2. Create a pointer temp <- nodes[ele]
3. If temp is not temp->parent
  - a. root\_data <- find(temp->parent->data)
  - b. temp->parent <- nodes[root\_data]
4. Return temp->parent->data

### **Algorithm 3: Merge two sets by rank**

**Input:** nodes(array), ele\_1 and ele\_2 (integer).

**Output:** true (success) / false (failure).

1. root\_1 <- find(ele\_1)
2. root\_2 <- find(ele\_2)
3. If root\_1 is -1 and root\_2 is -1 (elements not found)
  - a. Return false
4. Otherwise if root\_1 is -1 (ele\_1 not found)
  - a. Return false
5. Otherwise if root\_2 is -1 (ele\_2 not found)
  - a. Return false
6. If root\_1 is root\_2 (both elements are in the same set)
  - a. Return false
7. If nodes[root\_1]->rank is lesser than nodes[root\_2]->rank
  - a. nodes[root\_1]->parent <- nodes[root\_2]
8. Otherwise if nodes[root\_1]->rank is greater than nodes[root\_2]->rank
  - a. nodes[root\_2]->parent <- nodes[root\_1]
9. Otherwise
  - a. nodes[root\_2]->parent <- nodes[root\_1]
  - b. nodes[root\_1]-> rank++
10. Return true

### **FUNCTIONAL REQUIREMENTS:**

1. create()
2. find()
3. merge()

### **TIME COMPLEXITY ANALYSIS FOR EACH ALGORITHM:**

#### **1. create() – O(n)**

$$f(n) = 4n + 19$$

$$g(n) = n \text{ and } c = 10$$

n	Cost of f(n)	Cost of c.g(n)
0	19	0
1	23	10
2	27	20

3	31	30
4	35	40

Therefore  $n_0 = 4$  for which  $f(n) \leq c \cdot g(n)$  is true for all  $n_0 < n$ .

Therefore  $f(n) = O(n)$

Analysis summary:

- Best Case:  $O(1)$
- Average Case:  $O(1)$
- Worst Case:  $O(n)$

Explanation:

- Best Case ( $O(1)$ ): This occurs when the data value (data) being inserted is already present in the nodes array (i.e.,  $\text{nodes}[\text{data}] \neq \text{NULL}$ ). In this case, the function simply checks for the existing value, prints an error message, and frees the allocated memory for newnode. All these operations are constant time ( $O(1)$ ).
- Average Case ( $O(1)$ ): Assuming a random distribution of data insertions, the average case complexity might be closer to constant time ( $O(1)$ ). This means that on average, most insertions might not trigger capacity expansion and can be handled in constant time.
- Worst Case ( $O(n)$ ): The worst case occurs when the data value (data) is significantly larger than the current capacity of the nodes array.

## 2. find() - $O(\log n)$

$$f(n) = \log n + 5$$

$$g(n) = \log n \text{ and } c = 6$$

n	Cost of $f(n)$	Cost of $c \cdot g(n)$
0	-	-
1	5	0
2	5.301	1.806
3	5.477	2.862
4	5.602	3.612
5	5.698	4.193
6	5.778	4.668
7	5.845	5.070
8	5.903	5.418
9	5.954	5.725
10	6	6
11	6.041	6.248

Therefore  $n_0 = 10$  for which  $f(n) \leq c \cdot g(n)$  is true for all  $n_0 < n$ .

Therefore  $f(n) = O(\log n)$

### Analysis summary:

- Best Case:  $O(1)$
- Average Case:  $O(\log n)$
- Worst Case:  $O(\log n)$

### Explanation:

- Best Case ( $O(1)$ ): This occurs when the element (ele) is the root node itself (i.e.,  $\text{temp} \rightarrow \text{parent} == \text{temp}$ ). In this case, the function directly returns  $\text{temp} \rightarrow \text{parent} \rightarrow \text{data}$  which is  $\text{temp} \rightarrow \text{data}$ , avoiding the recursive call. All operations involved are constant time ( $O(1)$ ).
- Average Case ( $O(\log n)$ ): The find function would typically need to make a logarithmic number of recursive calls (one for each level) to reach the root from a given node. Each call itself is constant time ( $O(1)$ ). This results in an average-case time complexity of  $O(\log n)$ .
- Worst Case ( $O(\log n)$ ): The function might need to traverse all the way up to the root from the given node, resulting in  $h$  recursive calls (where  $h$  is the height of the tree). Each call takes constant time ( $O(1)$ ). This leads to a worst-case time complexity of  $O(\log n)$ .

### 3. merge() - $O(\log n)$

$$f(n) = 2 \log n + 11$$

$$g(n) = \log n \text{ and } c = 12$$

n	Cost of $f(n)$	Cost of $c.g(n)$
0	-	-
1	11	0
2	11.602	3.612
3	11.954	5.725
4	12.204	7.224
5	12.397	8.387
6	12.556	9.337
7	12.690	10.141
8	12.806	10.837
9	12.908	11.450
10	13	12
11	13.082	12.496
12	13.158	12.950
13	13.227	13.367
14	13.292	13.753

Therefore  $n_0 = 13$  for which  $f(n) \leq c.g(n)$  is true for all  $n_0 < n$ .

Therefore  $f(n) = O(\log n)$

### Analysis summary:

- Best Case:  $O(1)$
- Average Case:  $O(\log n)$
- Worst Case:  $O(\log n)$

### Explanation:

- Best Case ( $O(1)$ ): If the elements are already roots (single-node trees), merging them is very fast (constant time).
- Average Case ( $O(\log n)$ ): Finding the roots of elements takes a logarithmic amount of time on average. Merging based on rank is still constant time. This leads to an overall average complexity of  $O(\log n)$ .
- Worst Case ( $O(\log n)$ ): In the worst case, the elements might belong to very unbalanced (skewed) trees. Finding the root in such trees can take linear time (proportional to the height  $h$  of the tree). This leads to worst-case time complexity of  $O(\log n)$ .

## **APPLICATIONS:**

Disjoint-set ADT (Union-Find data structure) finds various applications in computer science due to its efficient management of disjoint sets (non-overlapping sets). Here are some key areas where it's used:

### **1. Kruskal's Algorithm:**

- This algorithm finds the minimum spanning tree (MST) of a weighted undirected graph. It works by iteratively processing edges in ascending order of weight. Disjoint-set is used to efficiently:
- Track connected components: Initially, each vertex represents a separate component. When an edge connects vertices from different components, the find operation helps determine which components to merge using the union operation.
- Avoid cycles: The find operation ensures that merging edges wouldn't create cycles in the MST.

### **2. Image Segmentation:**

- In image processing, disjoint-set helps identify connected regions of similar pixels (e.g., objects in an image). Here's how:
- Initially, each pixel represents a separate set.
- Adjacent pixels with similar properties (e.g., color) are considered for merging.
- The find operation determines if adjacent pixels belong to the same region.
- If not, the union operation merges them into a single set, representing a connected object.

### **3. Percolation Simulation:**

- This technique models systems with connected and disconnected regions, like studying fluid flow through porous materials. Disjoint-set can be used to:
- Track connected clusters: Initially, each site in the grid is a separate set.
- When a site becomes open (fluid can flow through), it's connected to its open neighbors using the union operation.
- The find operation helps determine if two sites belong to the same connected cluster, indicating fluid flow between them.

### **4. Network Connectivity:**

- In network management, disjoint-set can be used to:
- Track connected components in a network: Each device or router can initially represent a separate set.
- When a connection is established between devices, the union operation merges their sets, representing a connected network segment.
- The find operation helps determine if two devices are part of the same network segment.

### **5. Equivalence Checking:**

- This involves determining if two expressions or objects are equivalent based on certain rules. Disjoint-set can represent equivalence classes, and the union operation merges equivalent objects.

### **Additional Applications:**

- Social network analysis: Grouping users based on interactions.
- Code optimization: Identifying common subexpressions in compiler optimization.
- Game development: Tracking connected entities in a game world.