
G
O
D
O
O
O
D

Disjoint set Data Structure

/ Union-Find Data Structure

— as an abstract data type (ADT)

— implementation so that it's efficient for my algo

ADT — Data type

integers — user defined data type

** In some cases, actual data type is not so important ✓ it helps in my algo

Example :- For your algo, you need a FIFO queue of items.

I need certain operations

- → insert items in queue
- → remove items from queue (FIFO)
- → Is the queue empty?
- — —

FIFO queue

ADT that supports these operations

It is convenient that we agree on the operations without specifying the data type of the items.

& design my algo

Define a data structure → Implementation view

(Not the only view)

↓

Operational view ✓

FIFO queue: insert, remove...

implementation view ✓

- arrays
- linked list

Users of the data structure doesn't need to know about implementation view.

Operational view :→ A general ABSTRACT DATA TYPE

more general

Ex. Stack pop push

ADT

Operations

↑ implementation

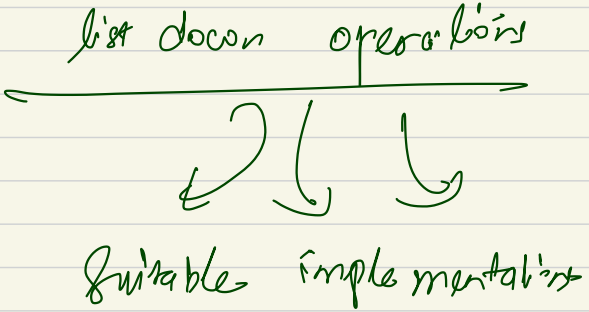
- arrays
- linked list.

While designing an algo, if you realize that your needs correspond to a part of ADT you immediately use that.

Modular process

Algo

Using the ADT (without worrying about implementation details)



* We cannot always ignore these.

Disjoint set / Union find as an ADT.

↓

Nice example of not so straight-forward data structure that helps in improving efficiency of Algos

Settings ⇒ There are n elements $x_1, \ldots x_n$ divided into disjoint sets.

Initially each element is a group in itself

Operations ⇒

Disjoint set data structure

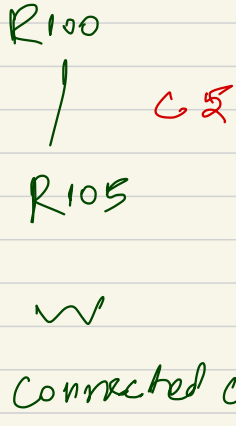
- **MakeSet(x) ⇒** Make a singleton set with x i'd - "x"
- **FindSet(x) ⇒** or **Find(x)** Given x, find the id of the set it belongs to
- **Union(x,y) ⇒** Create a single set of two sets containing x & y. New set can have id of any of those

Example of an algo :- (Min. Spanning Tree Algo
Kruskal's Algo

Setting Think of a social network N, (Facebook / Twitter)
defined by a set of people V
& a set of edges that define
relationship betn pair of people E (in no part^r order)

V R1 R2 — — R130

E: (R2, R3) (R5, R10) — — — [No part^r order]



Write an algo that finds out all the connected components, and for each person $x \in V$, outputs the connected component x belongs to

R2 C1

R100 C5

⋮ ⋮

* You can't make any assm about ordering of the Co'n, traversal could be used)
- You get to scan E only once

Algo (that uses my ADT)

```
for each x∈V do
        makeSet(x)
for each (x,y) ∈ E do
        if find(x) ≠ find(y)
                union(x,y)
for each x∈V
        output - Person "x" belongs to component
                 find(x).
```

m edges n people

2m - find(x) n union

O(n) makeSet O(m) find O(n) Union

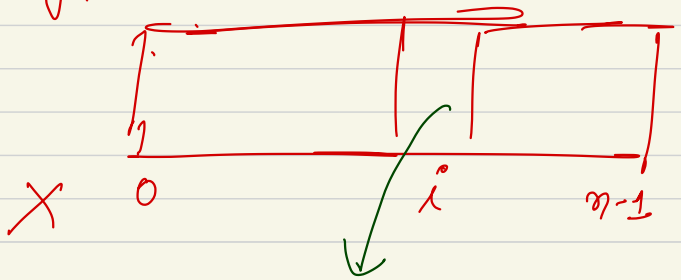
You can implement Unionfind data structure in any way & use how but you also want efficiency

Implementation view:=>

Simplest implementation

n people

Use an array?



X[i] stores the identity of the group containing ith element

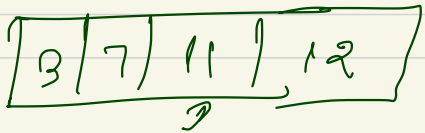
Find (a): O(1)

↳ Simply look at the index of array

makeSet
Find()
Union

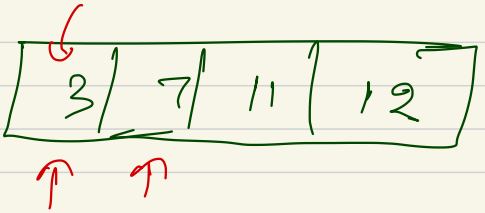
Define a group-id to each element

Ex. 3 7 11 12



Union

3 7 11 12

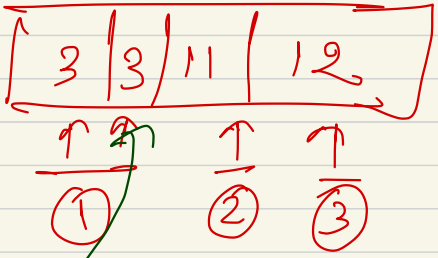


(3, 7)

Union (a, y)

m. Find → O(m)

n. Union → O(n²)



Find (7) = 3

O(m+n²)

Not very efficient

Change the identity of each element in B to A

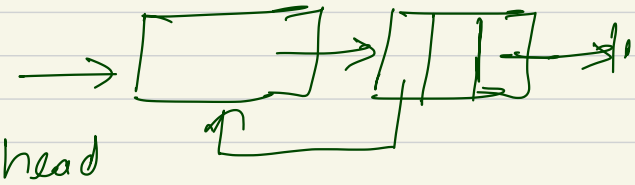
Set containing y

Set containing x

Complexity: Worst case :→ O(n) complexity

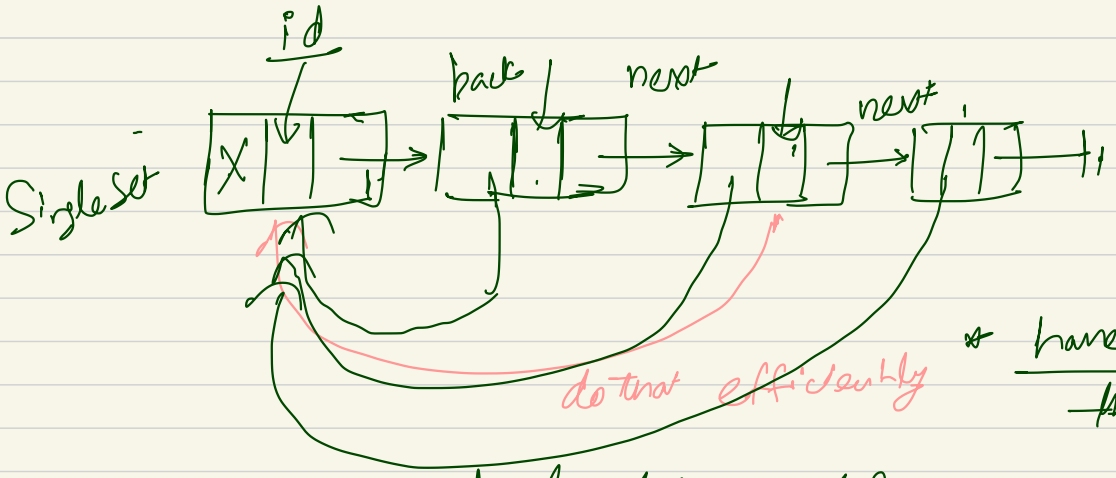
Linked List Data Structure

Each set is denoted by a linked list



makeSet - O(1)

Singleton set

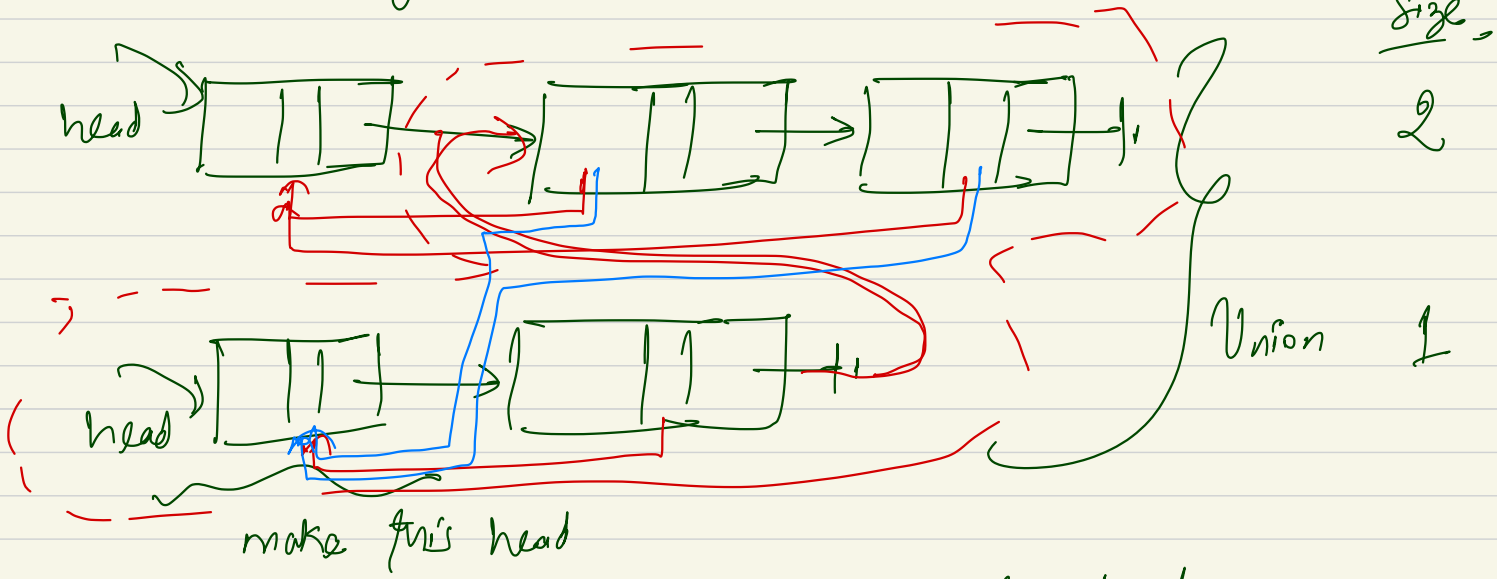


* head node stores the id of the set

* have a back pointer that directly points to the head

↓ find(x) : O(1)

Union (x, y)



* add one list to the tail of another (B)
(A)
(single pointer update)

* head & tail pointers of each list

* Change back pointers of all the nodes in list A.

worst case
$O(n)$ ←

⇒ $O(m+n^2)$?? Inefficient

Can we do better?

1. Keep # of nodes in each list (in head)
2. During union, add the smaller list to end of the larger list

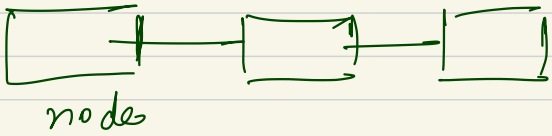
→ will that help?

Worst case - smaller list can be $n/2$ elts.

→ Changing $\frac{n}{2}$ back-pointers

Let's look at number of changes we're doing.

Look at a node, find an upper bound on the total # of times its back-pointer is updated.



Change its backpointer - when it is appended at the end of a longer list

first time its back pointer was updated ⇒ size of the resultan linked list = atleast 2

2nd time

3rd

Kth time

$K = O(\log n)$

4

8

2^K

Single node: almost $O(\log n)$ updates

n nodes : $\longrightarrow$ $O(n \log n)$ updates

m find + n Union

$O(m + n \log n)$ — Linked list

$O(m + n^2)$ — Static Array.

Worst Case: Each operation adding a list to the tail of another list

$\rightarrow O(n)$ time

In a series of operations, the worst case complexity for any one operation may be high, but on an average, complexity

be low.

This average over a sequence gives a more reasonable estimate.

Amortized Analysis : Independently worst case may be high

Sequence of operations: average analysis

for a node → O(log n) amortized complexity for bac-pointer update.

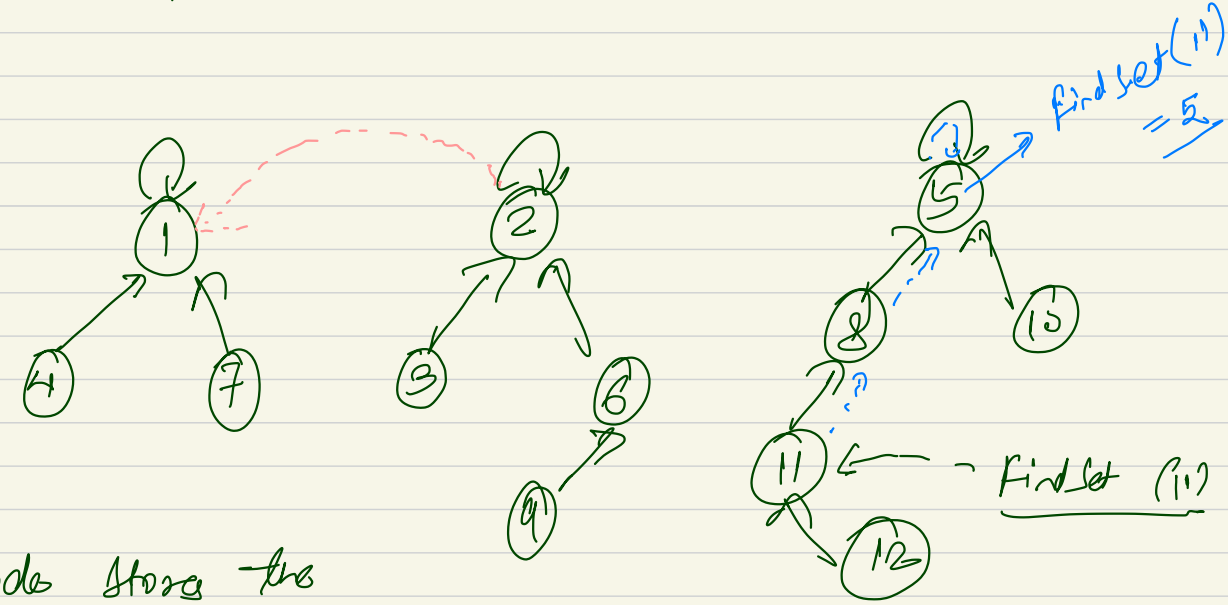
O(n log n)

Another popular implementation — via rooted trees.

→ Each set is a rooted tree (inverted tree)

Id of a set → Id of root

5 sets



* Each node stores the element of the set, and a parent pointer to the parent node.
* The parent pointer of the root points to itself.

Makeset (x) :→

O(1)

Find Set (x) :→ follow parent pointers to the root

Union (x, y) :→ Hang root of one tree as child of another.

↓

if you've the set IDs already

O(1) Oper"

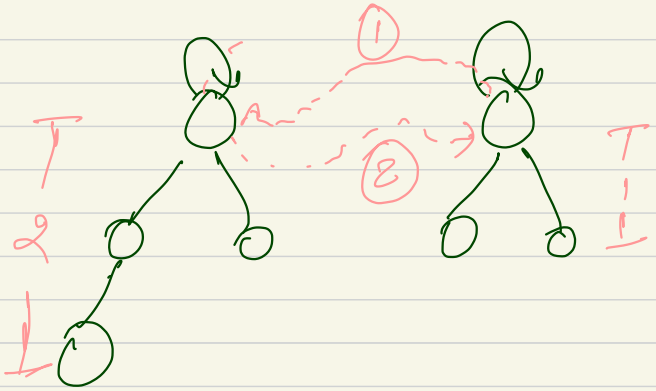
Worst case: this can be O(n)

Complexity

O(mn + n) not efficient

Can I do better (balancing)?

Linked list



addy a set A to the tail of set B

set A is smaller

follow 1 : height = 2

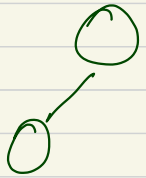
follow 2 : height = 3

⇒ When a union is performed, the pointer of the smaller group is set to to point to root of the larger group.

If we use this balancing, any tree of height h must have atleast 2^h elements.

Proof ⇒ Ind^n

Base Case: height 1



2 elements

Union

A B

(larger)

resultant tree

$h = \max\{h(A), h(B)+1\}$

Induction step: Assume true both A & B Allow the property

$\geq 2^{h(A)}$ & $\geq 2^{h(B)}$

Case 1: $h(A)$ is larger

$h = \max\{h(A), h(B)+1\}$

$\Rightarrow$ The final set tree has the height as that of A

but has more elements

$n(A) \geq 2^{h(A)}$

$n(new) \geq n(A)$

$\geq 2^{h(A)}$

$= 2^{h(new)}$

Case 2: $h(B)+1$ is larger

$n(B) \geq 2^{h(B)}$

$n(new) \geq 2 \cdot n(B)$

$\geq 2^{h(B)+1} = 2^{h(new)}$

→ both cases: new tree will contain at least 2^h elements

↳ Proof

⇒ Using balanced binary trees have atleast 2^h elements

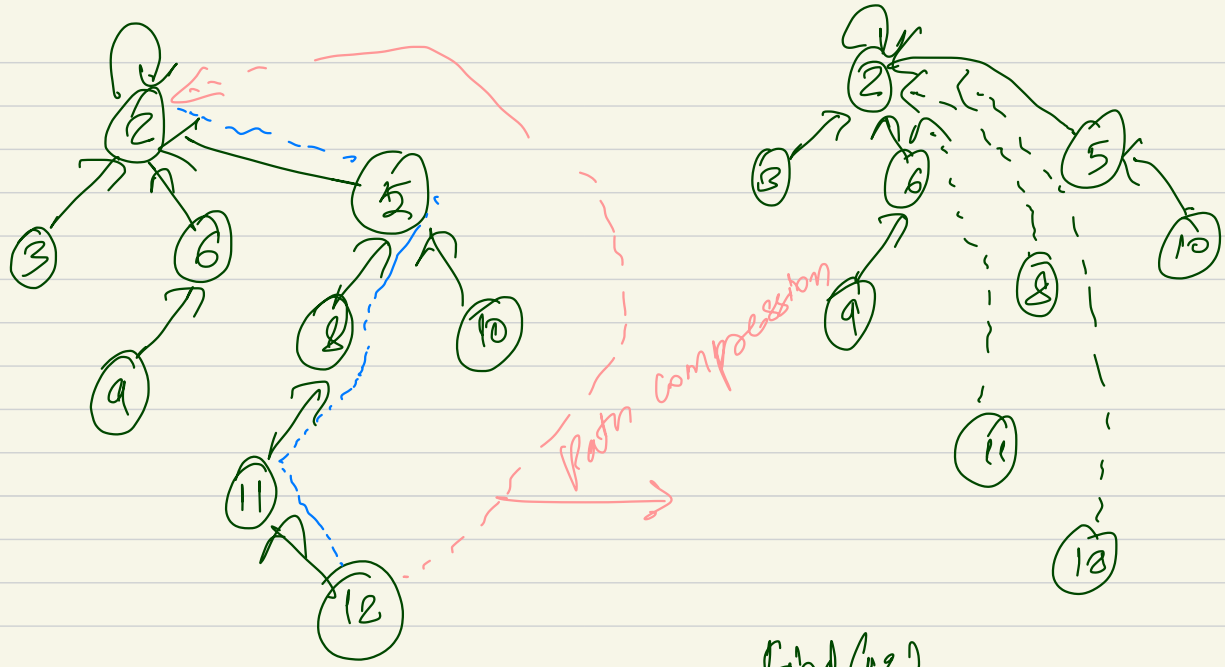
size $O(\log n)$ n elements

⇒ find operation can be done in $O(\log n)$ time

$O(n + m \cdot \log n)$ } quite similar to linked list

We can do better than this?

Idea Path Compression ⇒ In a find operation, for each node v that a find visits, reset the parent pointer from v to point to the root.



Analysis shows that using path compression for a sequence of m operations of find $\geq n$ has a complexity

$$O(m \log^* n)$$

$\log^* n$ is recursively defined as

$$\log^* 1 = \log^* 2 = 1$$

for $n > 2$

$$\log^* n = 1 + \log^* (\lceil \log_2 n \rceil)$$

Ex $\log^* 4 = 1 + \log^* 2 = 2$

$\log^* 14 = 1 + \log^* 4 = 3$

$\log^* 60000 = 1 + \log^* 16 = 4$

$< 2^{16}$

for any number $n \leq 2^{65536}$

$\boxed{\log^* n \leq 5}$ → as good as constant for all practical purposes

⇒ Path compression helps in achieving

$O(m \log^* n + n)$ complexity

m Finds
n Unions

$\Leftarrow$ $O(m+n)$ for any practical input.

```
makeSet()
    for each singleton element, x do
        x.parent = x
        x.size = 1
```

// For union by size, the element'll store the size.

Algo Union (x, y) // Assume x & y are individual roots

if $x.size < y.size$

$x.parent = y$

$y.size = y.size + x.size$

else

$y.parent = x$

$x.size = y.size + x.size$

```
a = find(x)
b = find(y)
Union(a,b)
```

Algo Find (x)

$r = x$

while $r.parent \neq r$

$r = r.parent$ // find the root

$z = x$

while $z.parent \neq z$

$w = z$

$z = z.parent$

$w.parent = r$ // path compression

ADT — Disjoint Sets / Union Find

MakeSet(x)

Find(x)

Union(x,y)

Oper^n View

Imp^n View — Array, Linked List, Trees

Problem in Trees — Problem in array

n MakeSet

m finds & n Union

Tree based implementation is the best

$\Rightarrow O(m+n)$

Amortized Analysis

Sequence of n operations, worst case appears once in a while but any analysis gives a more realistic picture.

For an algo that needs

Kruskal's algo