Topic: Data Structures for ADT Disjoint Sets

Alg Approach: Data Structuring

Sometimes the main innovation is in the way the data is structured. An example is disjoint set operations.

Data: Elements $x_1, x_2, \cdots x_n$ (could be integers)

Organization: maintained as disjoint sets

$$S = \{ S_1, S_2, \cdots, S_k \}$$

- A set is non-empty
- A set is represented by one of its members
- The collection is dynamic (changes over time)

Operations:

MakeSet(x): // x is not already in a set
- makes a new set that just contains x
- x is representative.

Union(x, y) merge(x, y)
- Set containing x and set containing y

are unioned (one set)

- representative?

FindSet(x)

- returns a pointer (or name of) the representative of the unique set containing x

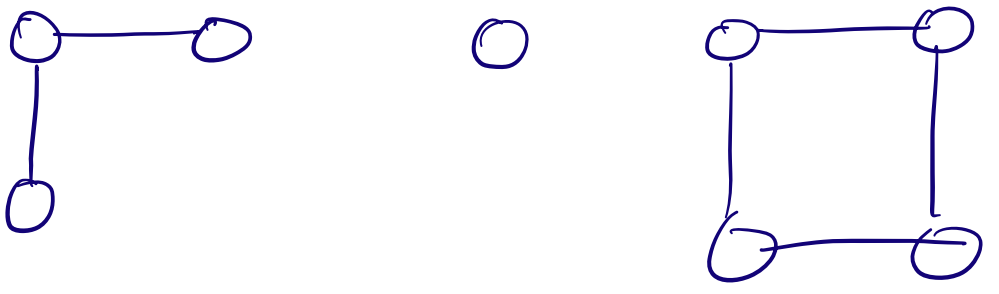
To analyze the running times of various implementations, we use two parameters:

n = number of MakeSet operations

m = number of MakeSet, Union, FindSet operations.

Eg Application: maintaining connectivity information about graph components.

(LANs, for example)



A note about persistent Data Structures and their analyses

We are dealing with a data type that has a "lifetime" over which a sequence of operations will be applied.

- different from "gcd(a, b)"

We can give the run-time complexity as a function of #elements, #find-ops, #union-ops

eg $T(e, f, u)$

But that makes it difficult to compare algs. running times.

For union-find, we assume

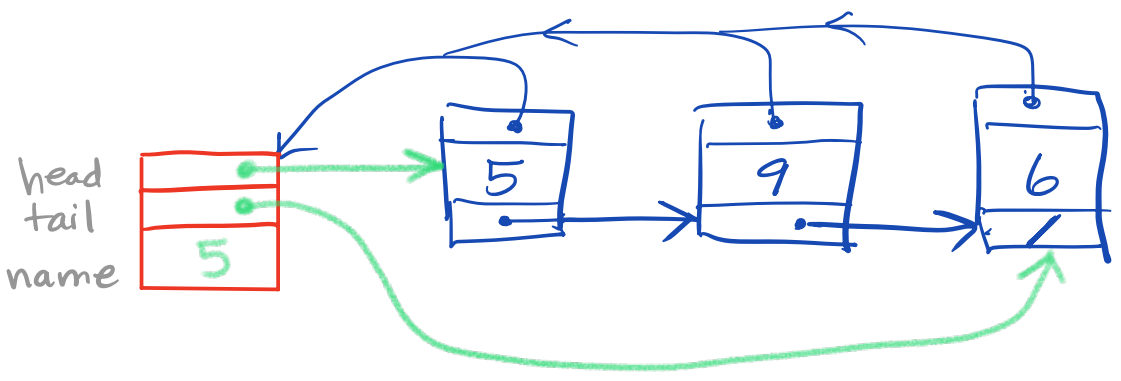
- m of the ops are make-Set (ie m is num elements)
- $\exists$ n ops of type find, union.
- Also we amortize the running time over all the ops.

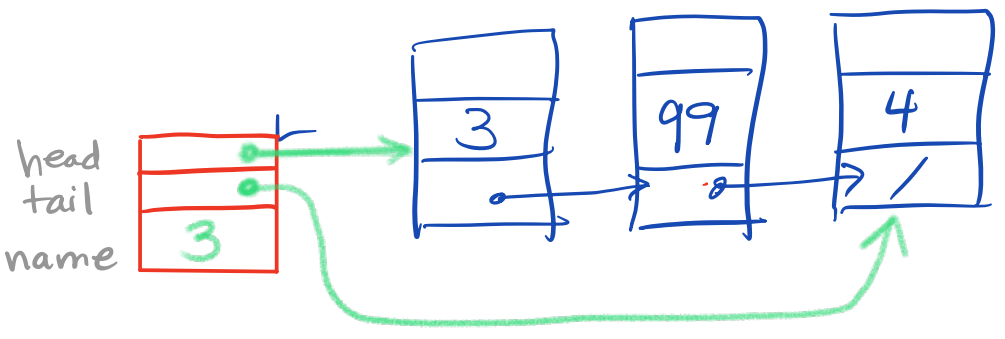
Amortizing is not the same as average case.

Take the entire run-time $T(n)$ over all n ops,

$\Rightarrow$ amortized running time is $\frac{T(n)}{n}$ ops

Representing Disjoint Sets as Linked Lists





MakeSet(x):



FindSet(x): return x→set→head

O(1).

```
if ( FindSet(x) ≠ Find set(y))
while temp ≠ NULL {
   temp = x→set→head
   temp→set = y→set
   temp = temp→next
}
```

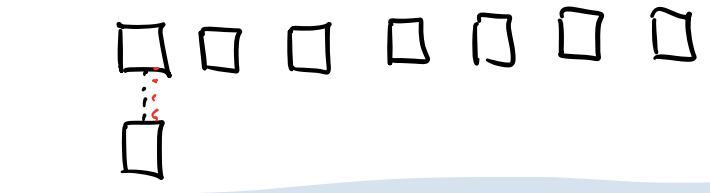
O(|y's set|)

Union(x,y): ?

Ideas

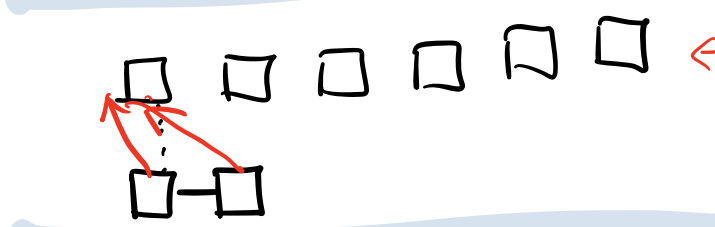
First, what if we do no optimization.

Worst case # links



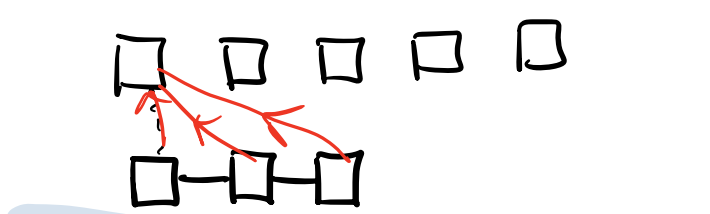
to reassign.

O(1)

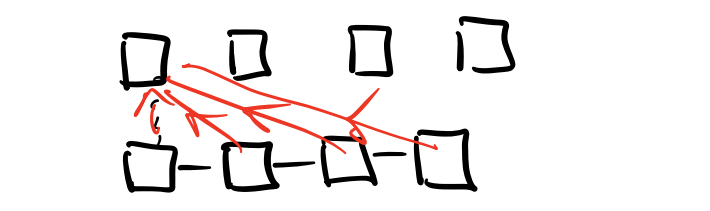


← Singleton

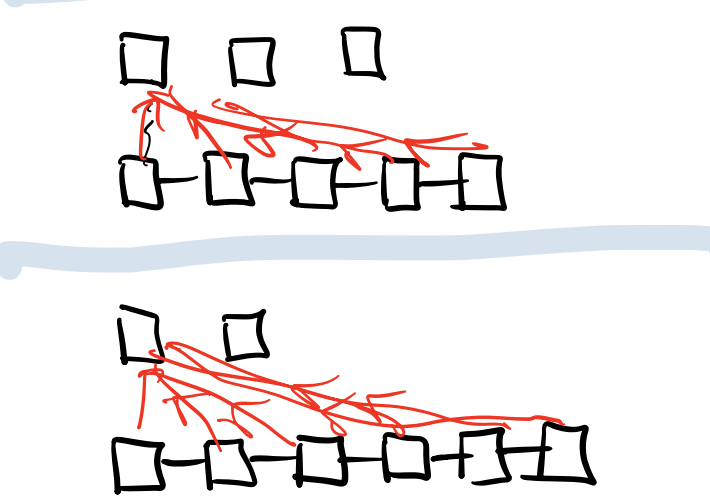
2



3

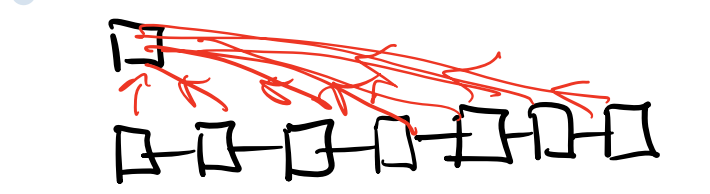


4



5

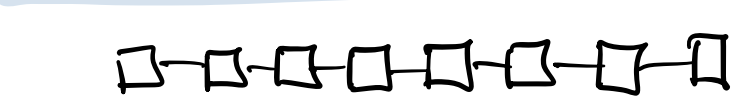
6



7

$\frac{(n-1)(n-2)}{2}$

$O(n^2)$.



What is total worst-case running time for m ops, n if which are MakeSet? $O(n^2)$ for m ops

in worst case, $m \in O(n)$ so amortized over m ops

How can we do better?

is $O(n)$ per operation.

heuristic: when unioning,
choose the smaller list to reassign
"set" pointers to larger list.

Weighted-Union heuristic

Theorem 21.1 (Cormen, Leiserson, Rivest + Stein — Introduction to Algorithms)

Linked-list rep'n + weighted-union heuristic for m ops, n makeSets, takes $O(m + n \lg n)$ time

Worst case is when $m \approx c \cdot n$

Then $O(m + n \lg n) \approx O(n \lg n)$ and amortizing over $c \cdot n$ op's, for a constant c, yields $O(\lg n)$ time/op

Proof: We prove an upper bound on the number of times x's parent pointer can be reset.

After all m ops, whats the largest set-size x can be in? n

For x to have its pointer reset, x must be in the smaller of the two sets being unioned....

So what happens to set-size of x's set each time it's pointer is updated?

doubles (or more).

How many times can that happen before the set size is n?

$\lceil \lg n \rceil$

Proof: The max number of union ops is $n-1$.

$\forall x$, x's pointer is updated only if the size of the set containing x doubles.

This occurs $\leq \lceil \lg n \rceil$ times during the m ops, as max set size is n.

$\therefore$ pointer updates $\in O(n \lg n)$

$\therefore$ total running time for m ops is $O(m + n \lg n)$

$\therefore$ worst case amortized running time is when

$m \approx n$ and amortized running time (time per operation, on average) is $O(\lg n)$. ▨

For <u>dynamic</u> data structures, that have a sequence of operations applied to them, we analyze the worst case running time for m operations and then <u>amortize</u> (average) that running time over the m ops (ie divide by m).

Result is called the amortized running time per operation.

This is different from "Average case analysis"

amortized = Worst case, amortized over # of ops

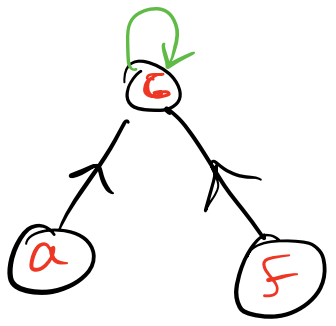
"Average" running time = Expected running time

given an assumed distribution of the input cases.

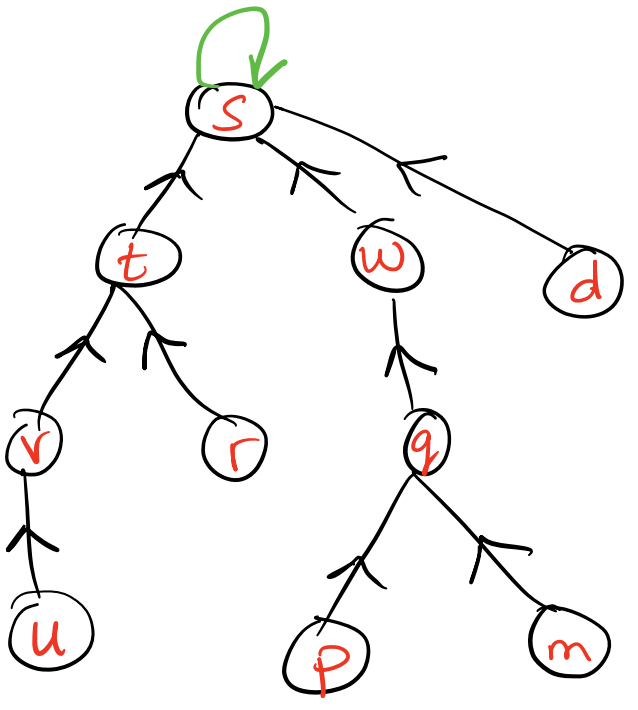
Cover Sept 17, 2026.

ADT Disjoint Sets

Represented by a Forest



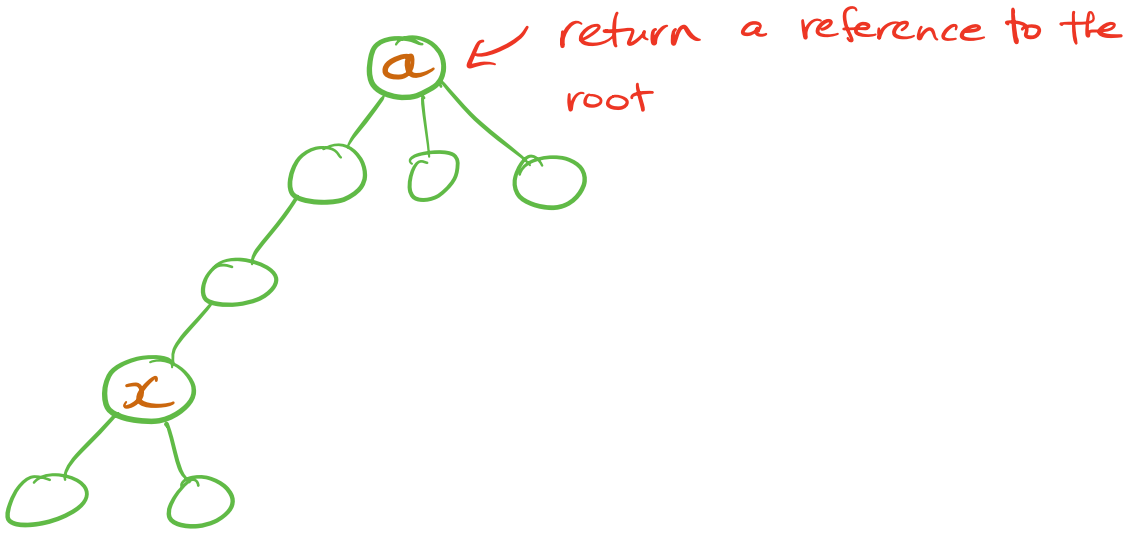




MakeSet(x)



FindSet(x)



Union(x, y)

