Competitive Programming

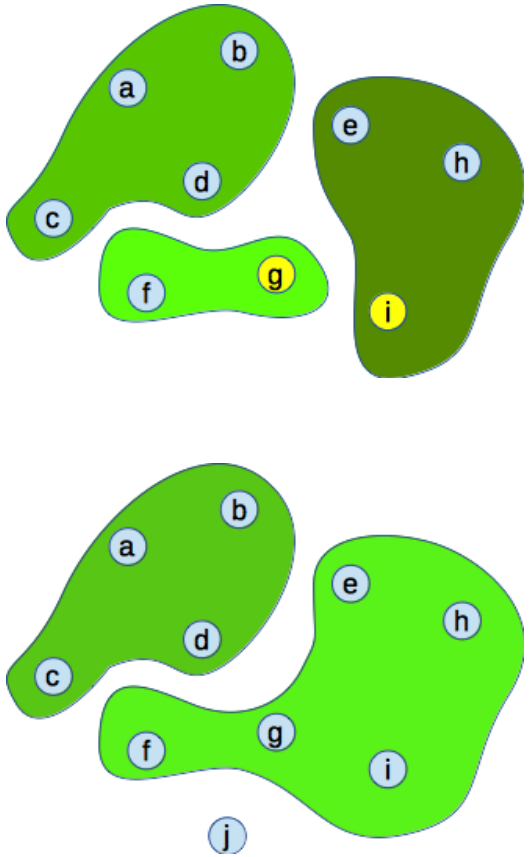
Disjoint Set Data Structure

Data Structures in Competitive Programming

- There are a (surprisingly small) number of data structures essential to competitive programming.
- You should:
  - Be familiar with them
  - Know the asymptotic performance of all their standard operations.
  - Be able to implement them if needed
  - Avoid implementing them if at all possible
  - … instead, use an existing, standard implementation

Disjoint Set

- We want a data structure that can:
  - Keep up with a number of disjoint sets
  - Quickly tell is two elements are in the same set.
  - Quickly merge two sets into a larger set.
  - Make new elements (singleton sets) as needed.

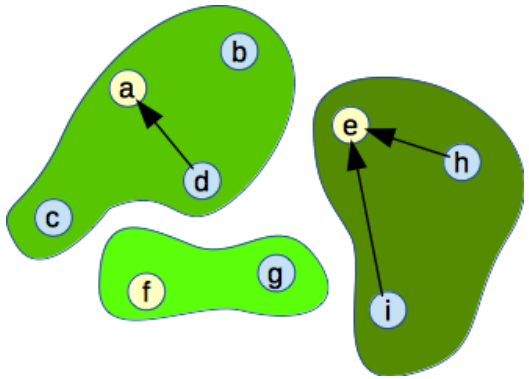


What is it Good For?

- Keep in mind, we're **not** building a container
  - We won't ask it to enumerate sets.
  - We won't be able to (efficiently) take sets apart after we merge them.
- This will be useful as part of other algorithms.
  - For example:
    - Connected components
    - Minimum Spanning Tree

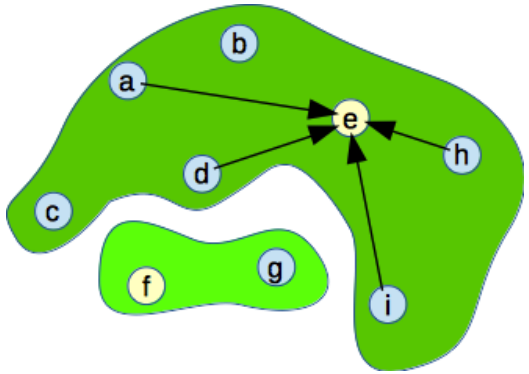
Implementing Disjoint Set

- Each set will have a *representative* element
  - Given an element x, we can quickly find the representative for its set.
  - We'll have an operation for this
    find(x) : return the representative for the set containing x.
  - This will let us quickly test for the same set.
  - if find(x) == find(y)
    - Then, x and y are in the same set.



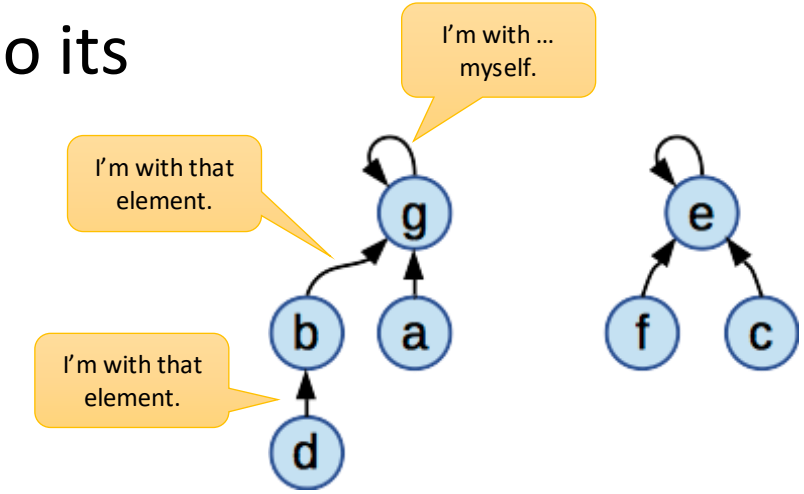
Implementing Disjoint Set

- We can quickly merge sets.
  - We'll have an operation for that
    union(x, y) : merge the sets
    containing elements x and y
- There are lots of different ways
  to design this data structure.
  - With different performance tradeoffs.
- But, one of them will be really, really fast … and simple.



Disjoint Set Forests

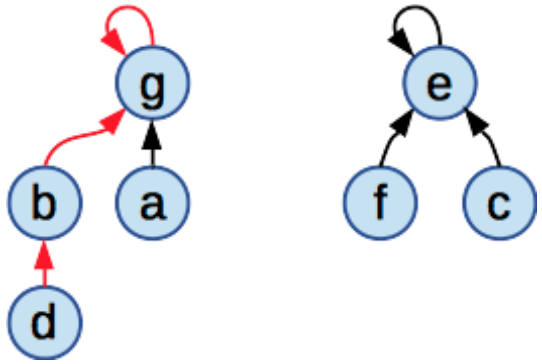
- Each element has a pointer to its representative.
  - What if we permit multiple levels of indirection?
  - The representative pointer doesn't have to point directly to the representative.

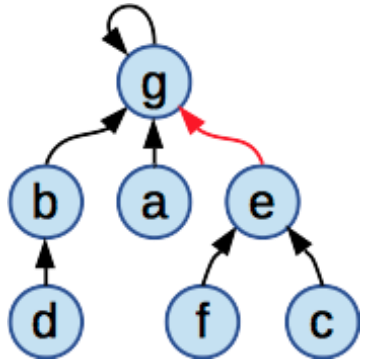


- Really, we're building a forest of trees.
  - The representative pointer is like a parent pointer.
  - Root nodes would just point to themselves.
  - All other nodes would point (directly or indirectly) to their representative.

Disjoint Set Forests

- find() won't be constant-time
  - May need to follow a chain of pointers to the representative.
- For union()
  - find() the representative for each set.
  - Then, a constant-time change to one pointer.

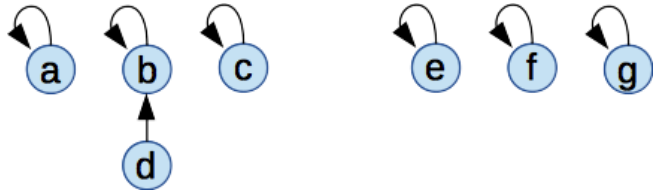




A Short Example

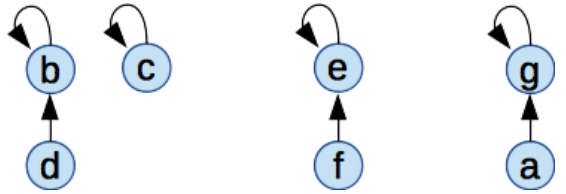
- The makeSet() operation creates singleton sets for all our elements

  - We can create more elements as we go, if needed.
- union(b, d) will make one node a child of another.

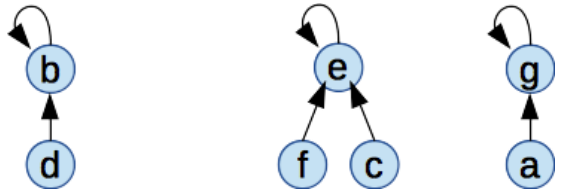


A Short Example

- After a union(e, f) and union(g, a):

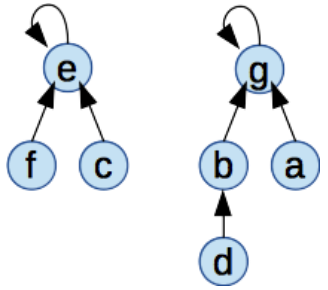


- union(f, c) will need to find the representative for f first.

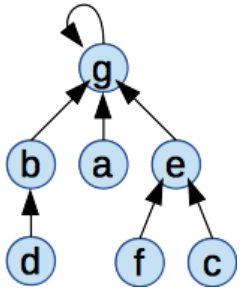


- union(a, d) will have to find the representative of both a and d.

A Short Example

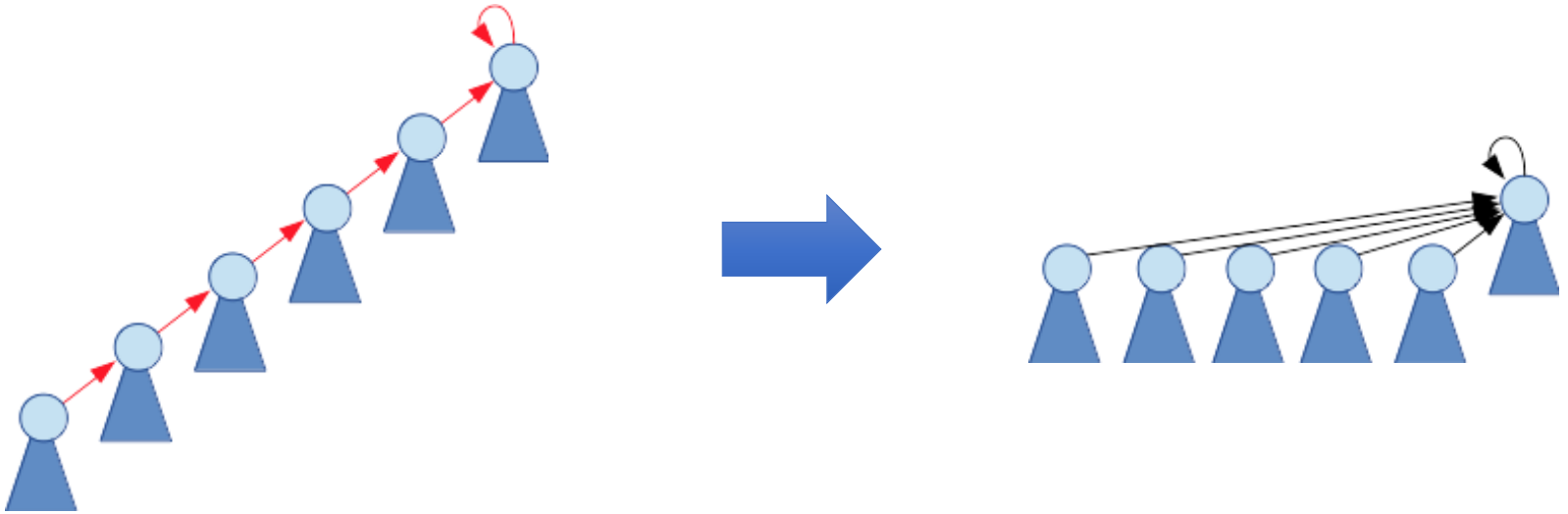


- After a union(b, c)



Path Compression

- We to prevent long chains of reference pointers.
  - We can’t keep this from happening,
  - But, when it does, we can keep it from happening again.
- This is called *path compression*.
  - It will happen automatically in the find() operation.



Implementing Path Compression

- It's easy do path compression as part of find().
  - Here's a recursive implementation, without path compression:

```
find( x )
   if x.parent != x
      return find( x.parent )
   return x
```

  - … and with path compression:

```
find( x )
   if x.parent != x
      x.parent = find( x.parent )
   return x.parent
```

A faster find()

- Recursion can be fast …
- … but an iterative solution could be even faster (better locality)

```
find( x )
   while x.parent != x
      x = x.parent
   return x
```

- For path compression, we can make two trips.

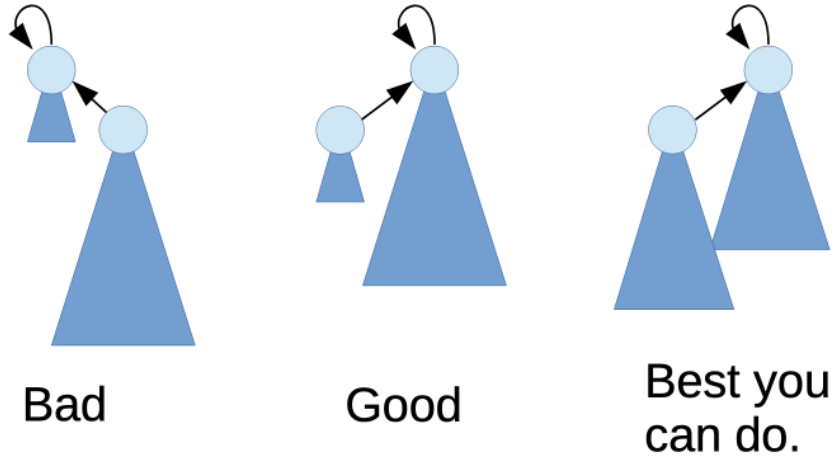
```
find( x )
   r = x
   while r.parent != r
      r = r.parent

   while x.parent != x
      next = x.parent
      x.parent = r
      x = next

   return r
```

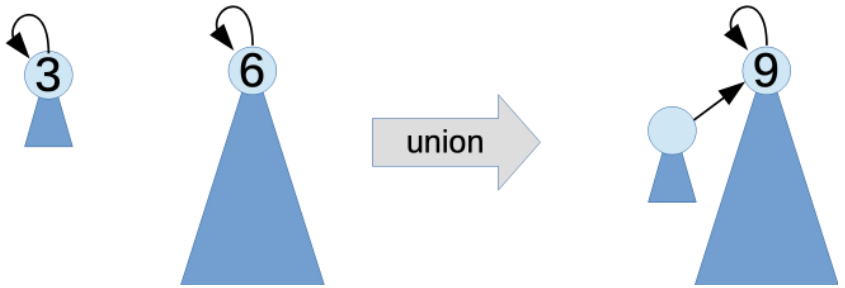
Union By Rank

- For the best asymptotic performance
  - it helps to union smaller sets into larger ones
  - rather than the other way around
  - This helps to make short, fat trees.
- At the root, we can store an upper bound on the height of the tree.
  - This is called the rank
- This probably isn't needed for competitive programming
  - Performance is already pretty good without it.



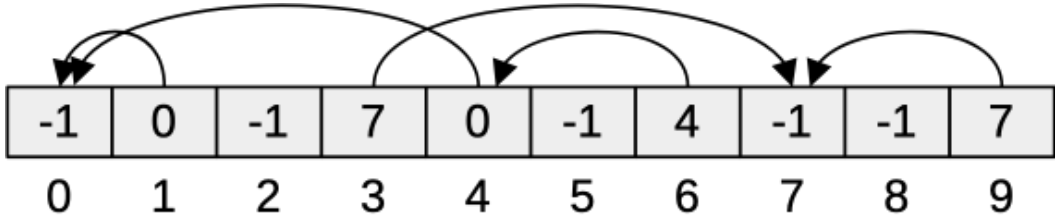
Augmenting Disjoint Set

- We can use similar tricks to maintain additional information about each set.
  - Like the size of each set.



Array Implementation

- Array implementation can be easier to work with
  - Each element can hold the index of its parent.
  - Maybe -1 to indicate no parent



  - Easier to initialize.

Analysis

- These operations can still take some time:
  - The cost of find() is linear in the path it has to follow.
  - union() makes two calls to find()
  - … but the rest is all constant-time operations.
- So, the way to slow this data structure down is to create really tall trees.
  - But, it takes a lot of cheap operations to do this.
  - Amortized cost of find() ends up being practically constant (technically a very slow-growing function of n)