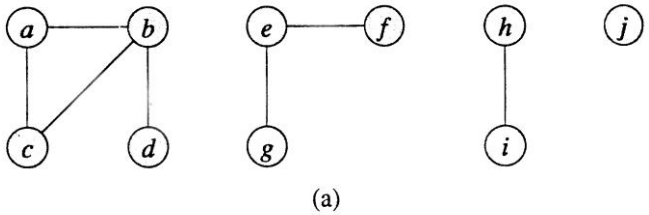
Disjoint Sets

Chapter 21

Disjoint Set Data Structure

- A way to maintain a collection S = { S1, S2, S3, ... } of disjoint dynamic sets
  - Each set has some representative element we use to identify the set
  - The Set data type will be useful to implement other algorithms coming up
  - STL or Collections classes include a “set” type
- Desirable operations
  - Make-Set(x)
    - Create a new set whose only member is item x
  - Union(x, y)
    - Union together set x and y and assume they are disjoint to begin with; there may be a new representative element
  - Find-Set(x)
    - Returns the set containing x

Application – Connected components of a graph



(a)

CONNECTED-COMPONENTS(*G*)

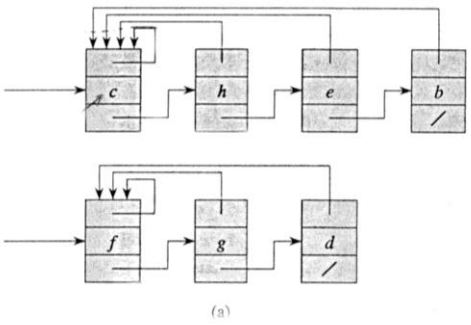
```
1 for each vertex v in the set V[G]
2        do MAKE-SET(v)
3  for each edge (u,v) in the set E[G]
4        do if FIND-SET(u)  != FIND-SET(v)
5             then UNION(u,v)
```

//determines whether two vertices are in the same connected component
SAME-COMPONENT (*u,v*)

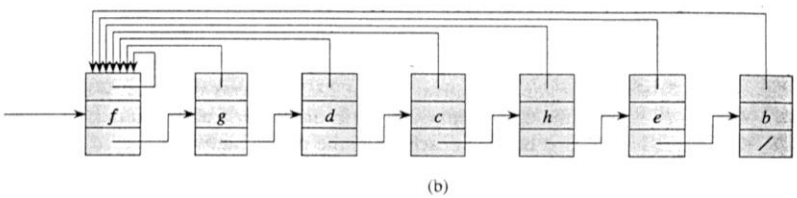
```
1  if FIND-SET(u) = FIND-SET(v)
2        then return TRUE
3        else return FALSE
```

Linked List Representation

- Each set is represented by its own linked list
- Each node links back to the head
- The head becomes the representative element



(a)



(b)

How to implement:

Make-Set
Find-Set
Union

Runtime?

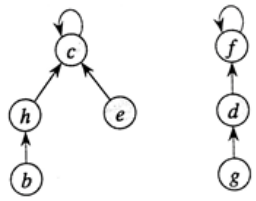
Common sequence could be $O(n^2)$

- Consider
  - Make-Set called for n numbers
  - Union called for each of the n sets

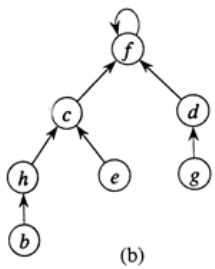
  - Can improve performance to O(nlgn) on Make, Union, Find operations if we union the shorter list to the longer one

Faster? implementation: Disjoint-set forest

- A forest is a collection of trees
- Worst case performance not better than linked-list but with two heuristics we can get a optimal amortized performance
- Use trees to represent sets; the root can be used to represent the set
- Make-set(x) : Create a tree with one node
- Find-set(x) : Follow links to root
- Union-set(x,y) : Make x as a child of y



(a)



(b)

Heuristics to improve the running time

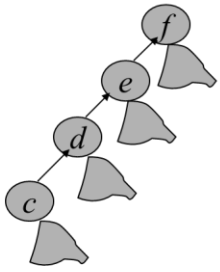
- Union by rank
  - Make the root of the tree with fewer nodes point to the root of the tree with more nodes
  - For faster computation we don't explicitly track the actual height
  - For each node, maintain a rank, which is an upper bound on the height of the node

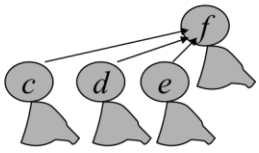
```
Make-Set(x)
x.p = x
x.rank  = 0

Union(x,y)
Link(Find-Set(x),Find-Set(y))
```

```
Link(x,y)
if x.rank > y.rank
  y.p = x
else
  x.p = y
    if x.rank == y.rank
      y.rank = x.rank+1
```

Heuristics to improve running time

- Path compression
  - Whenever we do a Find-Set operation (or union for that matter) make each node on the find path point directly to the root






```
Find-Set(x)
  if x != x.p
    x.p = Find-Set(x.p)
  return x.p
```

Heuristics and runtime

- Both improve runtime individually but together the improvement is even greater
- Union by rank
  - O(mlgm) where m is the total number of Make-Set, Union, and Find operations
- Path compression heuristic
  - O(n +f(lgn)) where f is the number of Find operations
- Both together
  - O(mα(n)) where α(n) is essentially a constant it grows so slowly