Disjoint Set ADT

Textbook Reading:

Chapter 4, Section 4.7, pp. 182-191

Implementing Sets

One method of maintaining disjoint sets would be to represent each subset by its characteristic vector, i.e., 1 (or true) to indicate an element is in the set and a 0 (or false), otherwise.

For example, if $S = \{0, 1, 2, \ldots, 9\}$ the sets $A = \{1, 3, 8, 9\}$ and $B = \{0, 1, 2, 3, 8\}$ could be represented by vectors

$$(0, 1, 0, 1, 0, 0, 0, 0, 1, 1) \text{ and } (1, 1, 1, 1, 0, 0, 0, 0, 1, 0)$$

The operations of intersection and union can be performed in linear time, i.e., time n, where n is the size of S. The operation of determining membership in a set, which we will refer to as the **find** operations can be perform in constant time, i.e., using a single comparison.

Implementing Disjoint Sets

When the collection of sets are disjoint then the intersection of any pair of set is empty, so we only need to implement the operation of **union** and **find**.

Disjoint Sets and Equivalence Relations

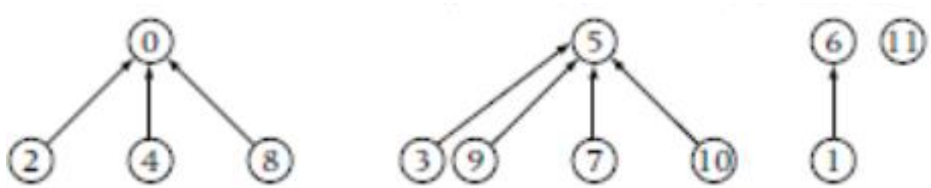
- When the disjoint sets form a partition, i.e., their union is the entire set S, then they correspond to a partition.
- We will apply disjoint sets to efficiently test for cycles when we get to minimum spanning trees and Kruskal's algorithm.

Compact way to represent disjoint sets using a single array

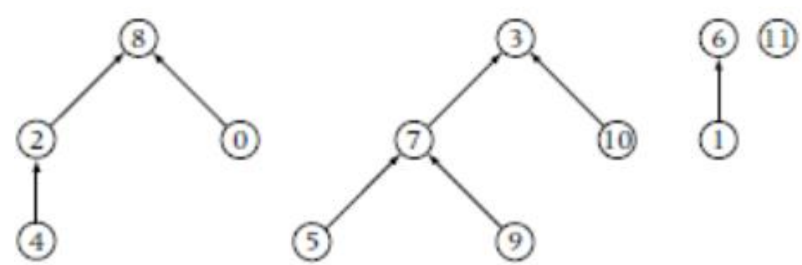
First represent the set the disjoint collections as the node sets of trees in a forest, then implement the forest using its parent array. Sets are identified by the root of each tree in the forest

Here's two possible implementations of the disjoint collection of sets

{{0,2,4,8}, {3,5,7,9,10}, {1,6}, {11}}



i	0	1	2	3	4	5	6	7	8	9	10	11
Parent[*i*]	–1	6	0	5	0	–1	–1	5	0	5	5	–1



i	0	1	2	3	4	5	6	7	8	9	10	11
Parent[*i*]	8	6	8	–1	2	7	–1	3	–1	7	3	–1

Implementation of Find

procedure *Find1*(*Parent*[0:n – 1],x,r)
Input: *Parent*[0:n – 1] (array representing disjoint subsets of S)
x (an element of S)
Output: r (the root of the tree corresponding to the subset containing x)

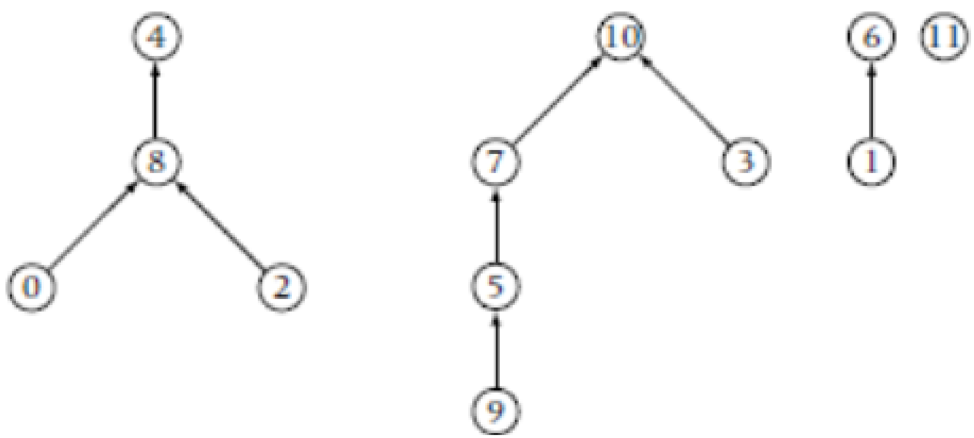
$r \leftarrow x$
while *Parent*[r] ≥ 0 **do**
  $r \leftarrow$ *Parent*[r]
endwhile
end *Find1*

PSN. Action of Find

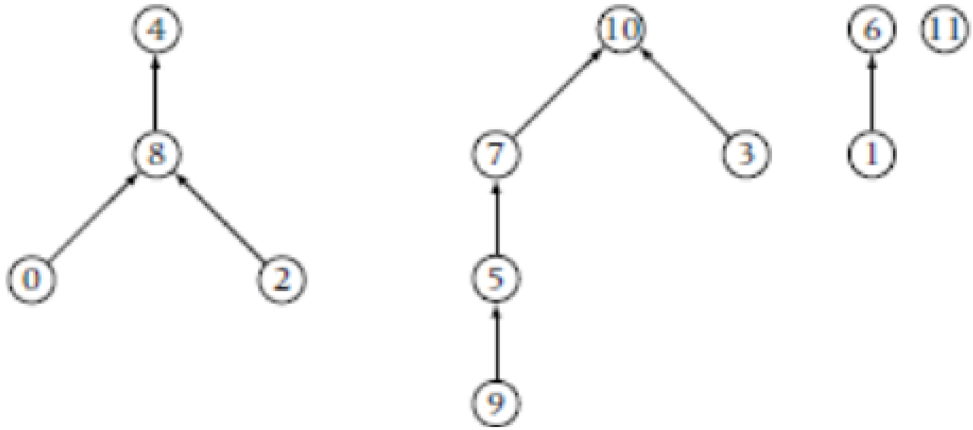
For the sample forest below

a) Show action of Find(9).

b) Show action of Find(2).



Solution



a) Parent(9) = 5, Parent(5) = 7, Parent(7) = 10, Parent(10) = -1. Find(9) = 10

b) Parent(2) = 8, Parent(8) = 4, Parent(4) = -1. Find(2) = 4

Implementing Union

- Find() will be efficient when the depth of the trees in the forest are small.
- When designing union we need to take this into account in order to make our disjoint set implementation efficient.
- It is less time consuming to compute the number of numbers of a tree than its depth. Therefore, we replace “depth” of the tree with the “number of nodes” in the tree.
- A tree with more nodes won’t necessary have smaller depth than a tree with fewer nodes.
- However, it turns out that this substitution is efficient and can be computed using a single addition, provided we keep track of the number of nodes in each tree at each stage.
- We achieve this by storing negative the number of nodes of a tree in the root of the tree instead of -1.

Pseudocode for Union

procedure *Union*(*Parent*[0:*n* – 1], *r*, *s*)
Input: *Parent*[0:*n* – 1] (an array representing disjoint sets)
r, *s* (roots of the trees representing two disjoint sets *A*,*B*)
Output: *Parent*[0:*n* – 1] (an array representing disjoint sets after forming $A \cup B$)

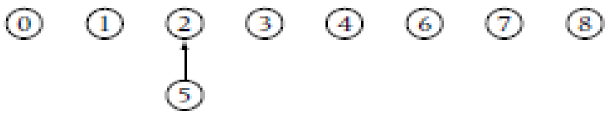
```
sum ← Parent[r] + Parent[s]
if  Parent[r] > Parent[s] then        //tree rooted at s has more vertices
        Parent[r] ← s                 //than tree rooted at r
        Parent[s] ← sum
else
        Parent[s] ← r
        Parent[r] ← sum
endif
end Union
```

Illustration of a sequence of union operations

Initial collection: {{0},{1},{2},{3},{4},{5},{6},{7},{8}}

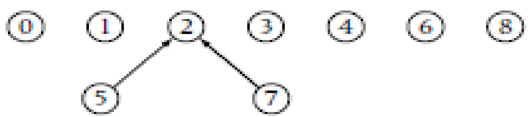
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	−1	−1	−1	−1	−1	−1	−1	−1	−1

Call *Union*[*Parent*[0:8],2,5] ⇒ {{0},{1},{2,5},{3},{4},{6},{7},{8}}



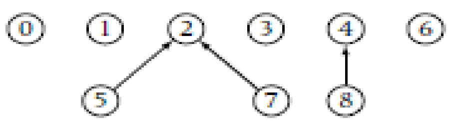
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	−1	−1	−2	−1	−1	2	−1	−1	−1

Call *Union*[*Parent*[0:8],2,7] ⇒ {{0},{1},{2,5,7},{3},{4},{6},{8}}



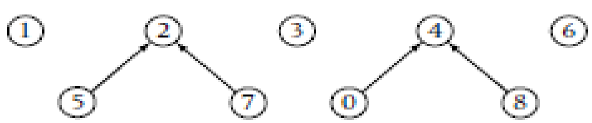
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	−1	−1	−3	−1	−1	2	−1	2	−1

Call *Union*[*Parent*[0:8],4,8] ⇒ {{0},{1},{2,5,7},{3},{4,8},{6}}



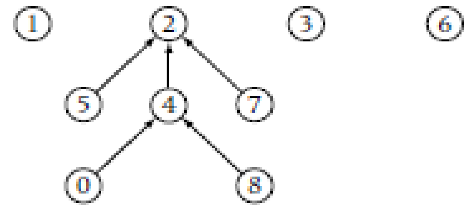
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	−1	−1	−3	−1	−2	2	−1	2	4

Call *Union*[*Parent*[0:8],0,4] ⇒ {{0,4,8},{1},{2,5,7},{3},{6}}



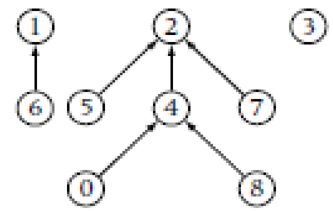
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	4	−1	−3	−1	−3	2	−1	2	4

Call *Union*[*Parent*[0:8],2,4] ⇒ {{0,2,4,5,7,8},{1},{3},{6}}



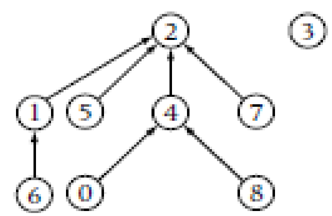
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	4	−1	−6	−1	2	2	−1	2	4

Call *Union*[*Parent*[0:8],1,6] ⇒ {{0,2,4,5,7,8},{1,6},{3}}



i	0	1	2	3	4	5	6	7	8
Parent[*i*]	4	−2	−6	−1	2	2	1	2	4

Call *Union*[*Parent*[0:8],1,2] ⇒ {{0,1,2,4,5,6,7,8},{3}}



i	0	1	2	3	4	5	6	7	8
Parent[*i*]	4	2	−8	−1	2	2	1	2	4

...

Complexity Analysis

The worst-case complexity in making an intermixed sequence of n calls to *Union*, and m calls to *Find1* is

$$O(m \log n).$$

Improving the Complexity – Collapsing Rule

Once we've found the root, traverse the path a second time and set the parent of each node to be the root. This doubles the computing time of find, but gives a better complexity for an intermixed sequence of union and find operations.

procedure *Find2*(*Parent*[0:n – 1],x,r)
Input: *Parent*[0:n – 1] (an array representing disjoint subsets of S)
x (an element of S)
Output: r (the root of the tree corresponding to subset containing x)

$r \leftarrow x$
while $Parent[r] \geq 0$ **do**
  $r \leftarrow Parent[r]$
endwhile
$y \leftarrow x$
while $y \neq r$ **do**
  $Temp \leftarrow Parent[y]$
  $Parent[y] \leftarrow r$
  $y \leftarrow Temp$
endwhile
end *Find2*

Complexity Analysis using Collapsing Rule

The worst-case complexity in making an intermixed sequence of n calls to *Union*, and m calls to *Find2* is

$$O(m\ \alpha(m,n))$$

where $\alpha(m,n)$ is an extremely slow-growing function. In fact, it grows so slowly that, for all practical purposes, we can regard $\alpha(m,n)$ as a constant function of m and n. The function $\alpha(m,n)$ is related to a functional inverse of the extremely fast-growing Ackermann's function $A(m,n)$ given by the recurrence relation:

$$A(0,n) = n + 1$$

$$A(m + 1,0) = A(m, 1)$$

$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

An Application of Disjoint Set ADT

Suppose we are given a forest *F* in a graph and wish to add an edge e = {*u*,*v*} as long as no cycle is formed.

How can we efficiently test whether a cycle is formed? This will be applied to the implementation of Kruskal's algorithm later in the course.

Solution: Construct a collection of disjoint sets corresponding to the node sets of the trees in *F*.

Perform the operations

call to *Find2*(*Parent*[0:n – 1],*u*,*r*)

call to *Find2*(*Parent*[0:n – 1],*v*,*s*)

If $r = s$ then *u* and *v* belong to the same set in the collection of disjoint sets, which implies they belong to the same tree, which in turn implies that a cycle is formed; otherwise a cycle is not formed. Thus,

if $r = s$ then REJECT edge *e*

else

add edge e to current forest

perform the union of sets containing *r* and *s*, respectively

Why can’t a bicycle stand by itself?

Because it is too tired.