CSE 220: Handout 29
Disjoint Sets

Motivation

- Suppose we have a database of people
- We want to figure out who is related to whom
- Initially, we only have a list of people, and information about relations is gained by updates of the form "alice is related to bob"
- Key property: If alice and bob are related, and bob and carol are related, then so are alice and carol
- Queries of the form: is alice related to bob?

Equivalence Relations

A binary relation R over a set S is called an equivalence relation if it has following properties

1. Reflexivity: for all element x, xRx

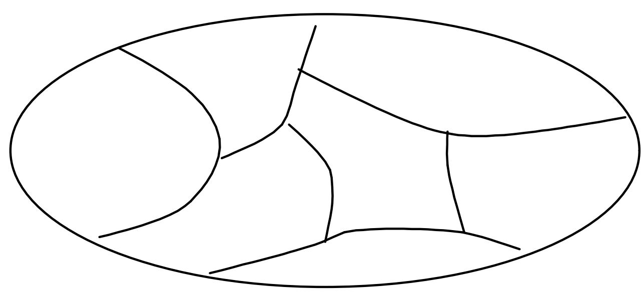
2. Symmetry: for all elements x and y, xRy if and only if yRx

3. Transitivity: for all elements x, y and z, if xRy and yRz then zRz

The relation "is related to" is an equivalence relation over the set of people

The Disjoint Sets view

- An equivalence relation R over a set S can be viewed as a partitioning of S into disjoint sets
- Each set of the partition is called an equivalence class of R (all elements that are related to each other according to R)

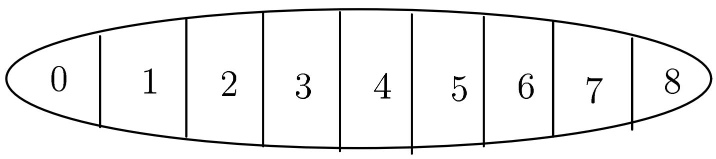


Union-Find Data Structure

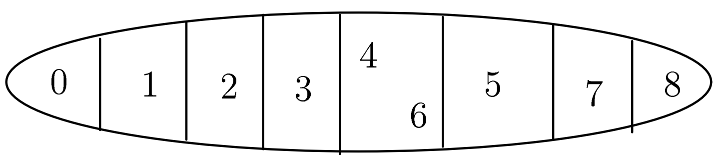
- The disjoint set is an abstract data type that supports two operations: `union` and `find`
- `union(x,y)` means merge the set containing x with the set containing y. In other words, declare x and y to be equivalent
- `find(x)` should return some representation of the set containing x.
- We can choose what `find(x)` returns as long as the following holds:
  `find(x) == find(y)` if and only if x and y are in the same set

Sample Sequence

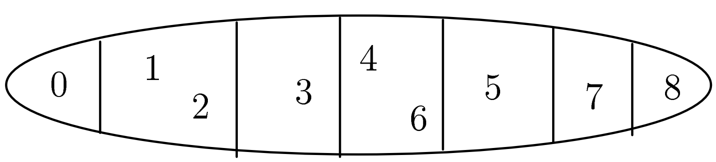
Initial



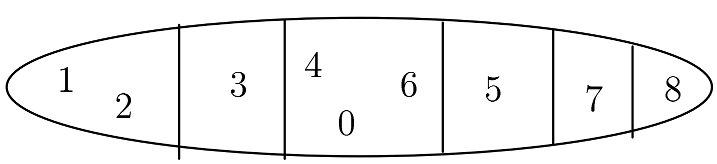
Union(4,6)



Union (1,2)

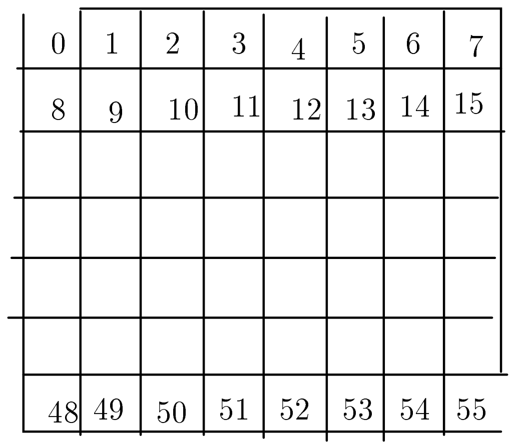


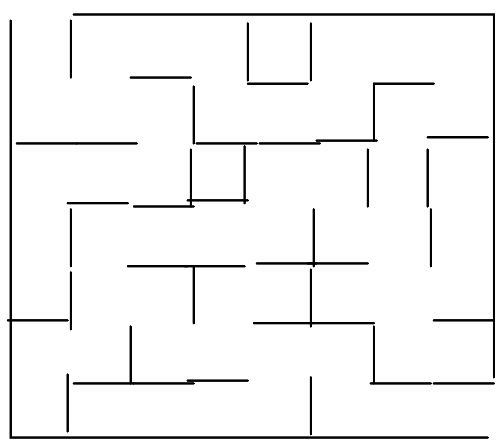
Union (0,4)



Application: Maze generation

Goal: Knock down "enough" walls randomly so that cells 0 and 55 are connected





Algorithm

```
Initialize set S of all 56 cells
as union/find ADT

while ( S.find(0) != S.find(55) ) {
   choose randomly a pair (i,j)
              of adjacent cells;
   if (find(i) != find(j)) {
       knock down the wall
             between cells i and j;
       S.union(i,j);
      }
}
```

A Simple Implementation

- We will assume that elements are 0 to $n-1$

- Maintain an array A: for each element i, $A[i]$ is the name of the set containing i

- `find(i)` returns $A[i]$: $O(1)$

- `union(i,j)` requires scanning entire array: $O(n)$

```
for (k=0;k<n;k++)
     if (A[k]==A[j]) A[k]=A[i]
```

- In better implementations, `union` is less expensive (`find` will no longer be $O(1)$, but overall performance will be improved)

Forest Data Structure

- Maintain the set S as a collection of trees, one per partition
- Initially, there are n trees, each containing a single element
- `find(i)` returns the root of the tree containing i
- `union(i,j)` merges the trees containing i and j
- Typical tree traversal not required, so no need for pointers to children, instead we need a pointer to parent
- Parent pointers can be stored in an array: `parent[i]` (set to -1 if i is root)

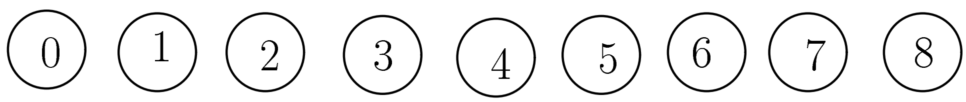
Tree-based Implementation

```
Initialization:
   for (i=0; i<n; i++)
      parent[i] = -1;

find(i):
   // traverse to the root
   for (j=i;
        parent[j]>=0;
        j=parent[j]);
   return j

union(i,j):
   root1 = find(i);
   root2 = find(j);
   if (root1 != root2)
      parent[root2] = root1;
```

Initially for $n = 9$

0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	-1

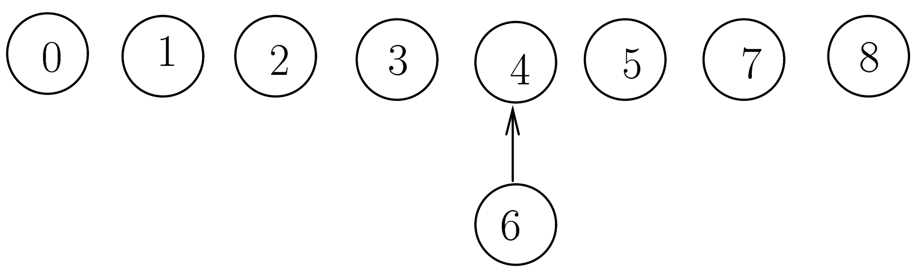
The forest view



After union(4,6)

0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	4	-1	-1

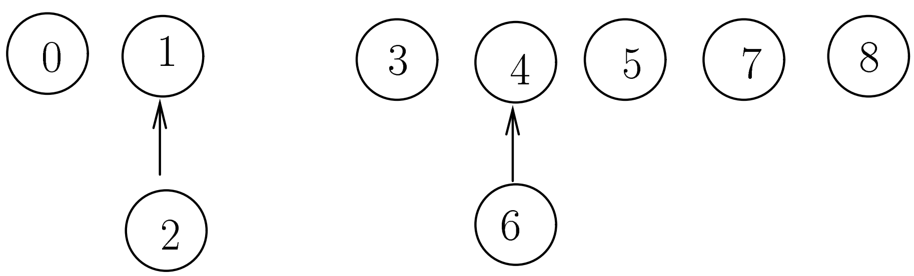
The forest view



After union(1,2)

0	1	2	3	4	5	6	7	8
-1	-1	1	-1	-1	-1	4	-1	-1

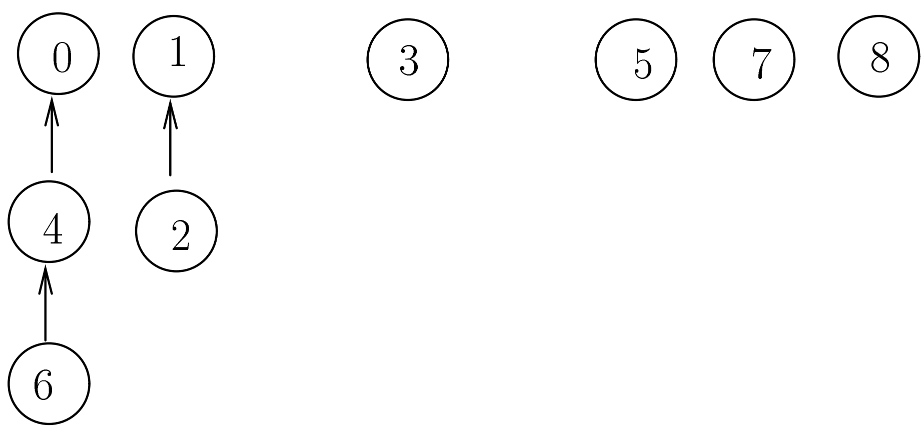
The forest view



After union(0,4)

0	1	2	3	4	5	6	7	8
-1	-1	1	-1	0	-1	4	-1	-1

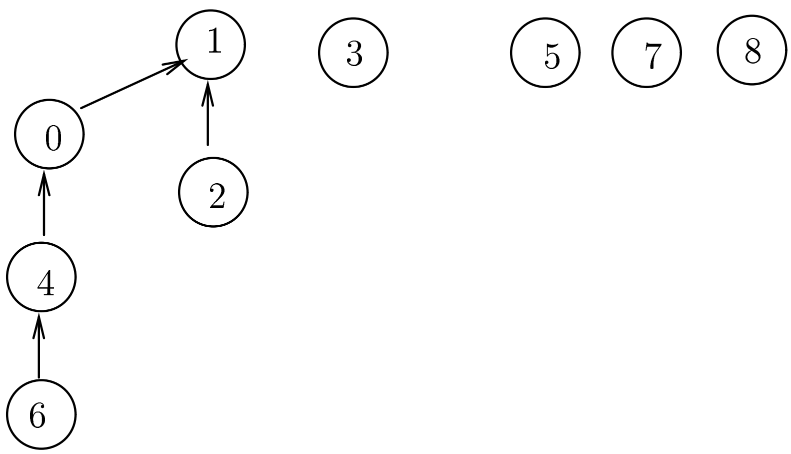
The forest view



After union(2,4)

0	1	2	3	4	5	6	7	8
1	-1	1	-1	0	-1	4	-1	-1

The forest view



Running Time Analysis

- Running time of `find(i)` is proportional to the height of the tree containing node i

- Running time of an operation is proportional to the maximum height of a tree in the forest

- This can be $O(n)$ in the worst case (but not always)

- Goal: Modify `union` to ensure that heights stay small

Union by Size

- Maintain sizes of all trees, and during `union` make smaller tree the subtree of the larger one
- Implementation: for each root node i, instead of setting `parent[i]` to -1, set it to $-k$ if tree rooted at i has k nodes
- Modify the code for `union` on p. 11

```
union(i,j):
   root1 = find(i);
   root2 = find(j);
   if (root1 != root2)
     if (parent[root1] <= parent[root2]) {
       // first tree has more nodes
       parent[root1] += parent[root2];
       parent[root2] = root1; }
     else { // second tree has more nodes
       parent[root2] += parent[root1];
       parent[root1] = root2; }
```

Sample Execution

0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	-1

Union (4, 6)

-1	-1	-1	-1	-2	-1	4	-1	-1

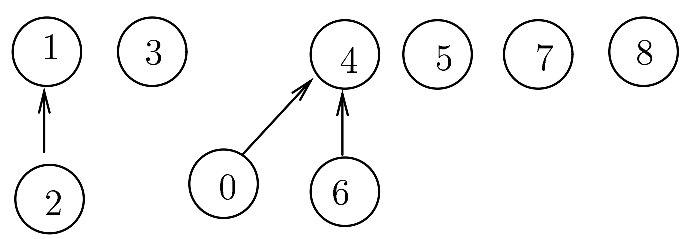
Union (1, 2)

-1	-2	1	-1	-2	-1	4	-1	-1

Union (0,4)

4	-2	1	-1	-3	-1	4	-1	-1

The forest view

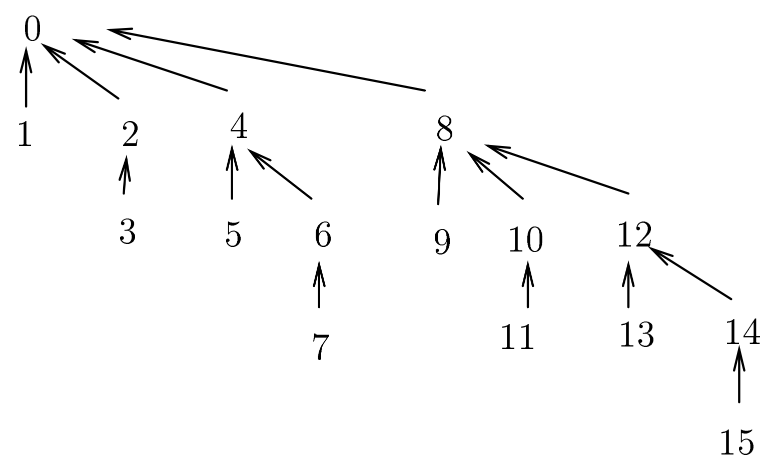


Running Time Analysis

- Claim: In union-by-size strategy, depth of a node cannot be more than $log\ n$
- Each time the depth of a node i increases by 1, the size of the tree containing i is at least doubled.
- Thus, running time of each operation is $log\ n$
- Average over a sequence of operations is almost constant: that is, a sequence of M union/find operations takes a total of $O(M)$ time on average
- Alternative to union-by-size strategy: maintain heights, and during union, make a tree with smaller height a subtree of the other

The worst-case scenario

- Worst-case scenario for 16 union operations



- Improving union won't change performance for this sequence of operations (why?: ties must be broken one or the other)

- Can we optimize `find`?

Path Compression

- During `find(i)`, as we traverse the path from i to root, update parent entries for all these nodes to the root

- This reduces the heights of all these nodes

- Pay now, and reap benefits later!
  Subsequent `find` may do less work

- Updated code for `find`

```
find (i) {
  if (parent[i] < 0)
     return i;
  else return parent[i] = find (parent[i]);
}
```