14-0: **Disjoint Sets**

- Maintain a collection of sets
- Operations:
  - Determine which set an element is in
  - Union (merge) two sets
- Initially, each element is in its own set
  - # of sets = # of elements

14-1: **Disjoint Sets**

- Elements will be integers (for now)
- Operations:
  - CreateSets(n) – Create n sets, for integers 0..(n-1)
  - Union(x,y) – merge the set containing x and the set containing y
  - Find(x) – return a representation of x's set
    - Find(x) = Find(y) iff x,y are in the same set

14-2: **Disjoint Sets**

- Implementing Disjoint sets
  - How should disjoint sets be implemented?

14-3: **Implementing Disjoint Sets**

- Implementing Disjoint sets (First Try)
  - Array of set identifiers:
    Set[i] = set containing element i
  - Initially, Set[i] = i

14-4: **Implementing Disjoint Sets**

- Creating sets:

14-5: **Implementing Disjoint Sets**

- Creating sets: (pseudo-Java)

```
void CreateSets(n) {
   for (i=0; i<n; i++) {
      Set[i] = i;
   }
}
```

14-6: **Implementing Disjoint Sets**

- Find:

14-7: **Implementing Disjoint Sets**

- Find: (pseudo-Java)

```
int Find(x) {
   return Set[x];
}
```

14-8: **Implementing Disjoint Sets**

- Union:

14-9: **Implementing Disjoint Sets**

- Union: (pseudo-Java)

```
void Union(x,y) {
   set1 = Set[x];
   set2 = Set[y];

   for (i=0; i < n; i=+)
     if (Set[i] == set2)
        Set[i] = set1;
}
```

14-10: **Disjoint Sets** $\Theta()$

- CreateSets
- Find
- Union

14-11: **Disjoint Sets** $\Theta()$

- CreateSets: $\Theta(n)$
- Find: $\Theta(1)$
- Union: $\Theta(n)$

14-12: **Disjoint Sets** $\Theta()$

- CreateSets: $\Theta(n)$
- Find: $\Theta(1)$
- Union: $\Theta(n)$

We can do better! (At least for Union ...)

14-13: **Implementing Disjoint Sets II**

- Store elements in trees
- All elements in the same set will be in the same tree
- Find(x) returns the element at the root of the tree containing x

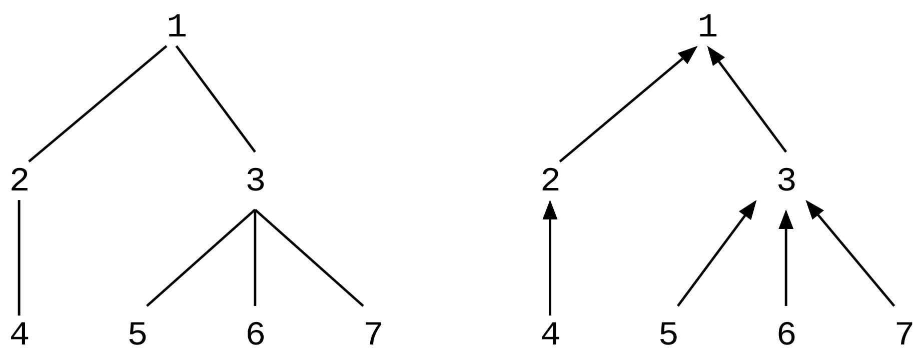
- How can we easily find the root of a tree containing x?

14-14: **Implementing Disjoint Sets II**

- Store elements in trees
- All elements in the same set will be in the same tree
- Find(x) returns the element at the root of the tree containing x
  - How can we easily find the root of a tree containing x?
  - Implement trees using *parent pointers* instead of *children pointers*

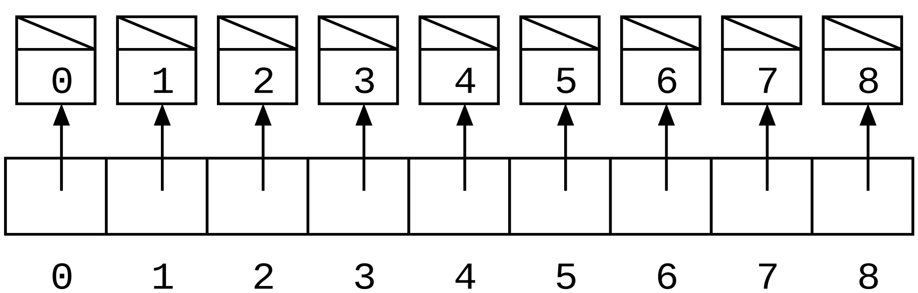
14-15: **Trees Using Parent Pointers**

- Examples:



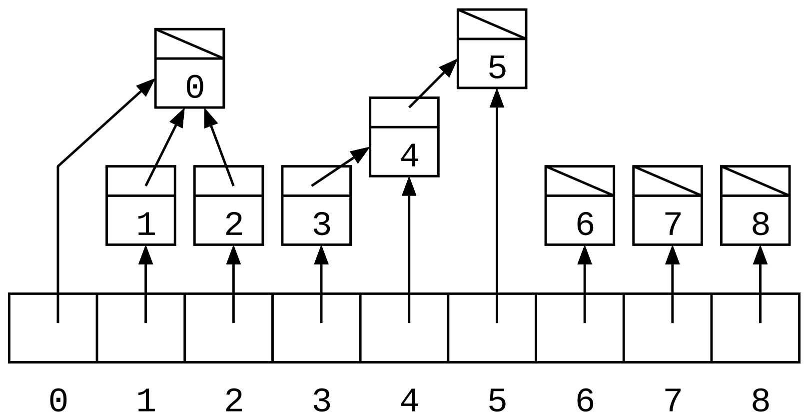

14-16: **Implementing Disjoint Sets II**

- Each element is represented by a node in a tree
- Maintain an array of pointers to nodes




14-17: **Implementing Disjoint Sets II**

- Each element is represented by a node in a tree
- Maintain an array of pointers to nodes




14-18: **Implementing Disjoint Sets II**

- Find:

14-19: **Implementing Disjoint Sets II**

- Find:
  - Follow parent pointers, until root is reached.
    - Root is node with `null` parent pointer.
    - (alternately, root points to itself)
  - Return element at root

14-20: **Implementing Disjoint Sets II**

- Find: (pseudo-Java)

```
int Find(x) {
   Node tmp = Sets[x];
   while (tmp.parent != null)
      tmp = tmp.parent;
   return tmp.element;
}
```

14-21: **Implementing Disjoint Sets II**

- Union(x,y)

14-22: **Implementing Disjoint Sets II**

- Union(x,y)
  - Calculate:
    - Root of x's tree, rootx
    - Root of y's tree, rooty
  - Set parent(rootx) = rooty

14-23: **Implementing Disjoint Sets II**

- Union(x,y) (pseudo-Java)

```
void Union(x,y) {
   rootx = Find(x);
   rooty = Find(y);
   Sets[rootx].parent = Sets[rooty];
}
```

14-24: **Removing pointers**

- We don't need any pointers
- Instead, use index into set array

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

14-25: **Removing pointers**

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

- Union(2,3), Union(6,8), Union(0,2), Union(2,6)

14-26: **Removing pointers**

- Union(2,3), Union(6,8), Union(0,2), Union(2,8)

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

14-27: **Implementing Disjoint Sets III**

- Find: (pseudo-Java)

```
int Find(x) {
   while (Parent[x] >= 0)
      x = Parent[x]
   return x
}
```

14-28: **Implementing Disjoint Sets II**

- Union(x,y) (pseudo-Java)

```
void Union(x,y) {
   rootx = Find(x);
   rooty = Find(y);
   Link(rootx, rooty);
}

Link(x,y) {
   Parent[x] = y;
}
```

14-29: **Efficiency of Disjoint Sets II**

- So far, we haven't done much to improve the run-time efficiency of Disjoint sets.
- Two improvements will make a huge difference:
  - Union by rank
  - Path compression

14-30: **Union by Rank**

- Merging sets:
  - We want to avoid long chains of elements
  - When merging two sets, which should become the parent, and why?

14-31: **Union by Rank**

- Merging sets:
  - We want to avoid long chains of elements
  - When merging two sets, which should become the parent, and why?
    - The tree with the largest height should be the parent.
    - Keep track of an estimate of the height of each tree (until we add path compression, the estimate will be exact)

14-32: **Union by Rank**

- For each node, keep a rank, which is an estimate of the depth of the tree rooted at that node
- Initially, rank for each node is 0
- How should ranks be used / updated?

14-33: **Union by Rank**

```
union(x,y) {
   rootx = Find(x);
   rooty = Find(y);
   Link(rootx,rooty);
}

Link(x,y) {
  Parent[x] = y
}
```

14-34: **Union by Rank**

```
union(x,y) {
  rootx = Find(x);
  rooty = Find(y);
  Link(rootx,rooty);
}

Link(x,y) {
  if (rank[x] > rank[y]);
    Parent[y] = x;
  else
    Parent[x] = y;
    if (rank[x] == rank[y]);
      rank[y]++;
}
```

14-35: **Union by Rank**

- For each node, we need either the rank or the parent – not both
- We can use the same array to store both pieces of information
  - If a node $x$ is not a root, Parent$[x]$ = parent of x
  - If a node $x$ is a root, Parent$[x]$ = 0 - height of tree
- Assuming we don't allow 0 to be a set, if Parent[x] is positive, then x is not a root. If Parent[x] is 0 or negative, then x is a root
- (note – text does not do this! Roots point to themselves, rank is separate)

14-36: **Path Compression**

- After each call to `Find(x)`, change `x`'s parent pointer to point directly at root
- Also, change all parent pointers on path from `x` to root

14-37: **Implementing Disjoint Sets III**

- Find: (pseudo-Java)

```
int Find(x) {
   if (Parent[x] < 0)
      return x;
   else {
       Parent[x] = Find(Parent[x]);
       return Parent[x];
   }
}
```

14-38: **Disjoint Set $\Theta$**

- Time to do a Find / Union proportional to the depth of the trees

- "Union by Rank" tends to keep tree sizes down
- "Path compression" causes Find and Union to flatten trees
- Union / Find take roughly time O(1) on average

14-39: **Disjoint Set $\Theta$**

- Technically, $m$ Find/Unions on $n$ sets take time $O(m \lg^* n)$
- $\lg^* n$ is the number of times we need to take $\lg$ of $n$ to get to 1.
  - $\lg 2 = 1$, $\lg^* 2 = 1$
  - $\lg(\lg 4) = 1$, $\lg^* 4 = 2$
  - $\lg(\lg(\lg 16)) = 1$, $\lg^* 16 = 3$
  - $\lg(\lg(\lg(\lg 65536))) = 1$, $\lg^* 65536 = 4$
  - $\ldots$
  - $\lg^* 2^{65536} = 5$
- # of atoms in the universe $\approx 10^{80} \ll 2^{65536}$
- $\lg^* n <= 5$ for all practical values of $n$