Kruskal's using Union-Find Operations

- we notice that the challenge in Kruskal's algo is that we have to constantly check for cycles when we add edges
- if we use DFS, we would have worst case:

  $O(|V|^2)x(V-1) = O(|V|^3)$
- this sucks for efficiency, which is why Kruskals is typically implemented using structures that support efficient union operations on sets

set: {a b c d}

subsets: {a} ~~{a b} {b c d}~~

not disjoint.

Union-Find Operations

- these operations work with disjoint subsets {a} {b} {c d}
  - ie: elements are only in one set at a time
- all operations work on a *Collection of Disjoint Subsets*
- the following set operations are supported:

```
makeset(x)
   - creates a new one element set
     containing {x}
find(x)
   - returns the subset containing x
union(x,y)
   - creates a new subset Sxy containing the
     subsets Sx and Sy. The sets Sx and Sy
     are removed from the collection, and
     Sxy is added
```

makeset{a} : creates S_a

makeset(b) : " S_b

eg: find(a) returns "S_a"

union(a, b) ⇒ ~~create~~ returns S_a:{a,b}

~~S_a~~ S_b:{}

Union-Find Example

- consider the following sequence of union-find operations:

```
let S be the set {1, 2, 3, 4, 5, 6, 7, 8}
for each element x in S
     makeset(x)
```

$S_1 = \{1\}$ $S_2 = \{2\}$ $S_3 = \{3\}$ . . .

```
union(2,7)
```

$S_2 = \{2, 7\}$ $S_7 = \{\}$

```
union(1,4)
```

$S_1 = \{1, 4\}$ $S_4 = \{\}$

```
y ← find(4)
```

returns "S_1"

```
union(y,3)
```

$S_1 = \{1, 3, 4\}$ $S_3 = \{\}$.

```
x ← find(1)
```

returns S_1

```
y ← find(2)
```

returns S_2

```
union(x, y)
```

$S_1 = \{1, 2, 3, 4, 7\}$

Restating Kruskal's

- here is a more typical way to state Kruskal's algorithm (using union-find operations)
- note:
  - we check for acyclicity by maintaining disjoint subsets of vertices in the solution tree
  - we can only add an edge if both its vertices are in disjoint subsets – otherwise we create a cycle

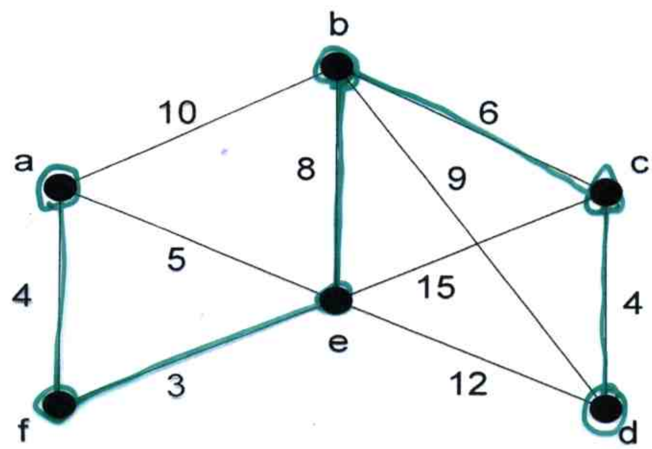
```
algorithm Kruskal(G)
              Graph
    Create a tree T ← Ø //T will contain the soln MST
    Create a priority queue PQ // candidate edges
    Create a collection C // contains disjoint subsets

    for each vertex v in G do      } create a bunch of
        C.makeset(v)               } 1-element subsets.

    for each edge e in G do        key:value = edge weight & edge
        PQ.add(e.weight, e)

    while T has fewer than n-1 edges do
  edge  (u,v) ← PQ.removeMin() // get next smallest edge
        cu ← C.find(u)
        cv ← C.find(v)             → are they in the same subset.
        if cv ≠ cu then            // is there a cycle?
            T.addEdge(v,u)
            C.union(cu, cv)
return tree T
```

Another Kruskal Example (using the union find stuff)



PQ (min-keyed)

~~3: ef~~
~~4: cd~~
~~4: af~~
~~5: ae~~
~~6: bc~~
~~8: be~~
9: bd
10: ab
12: de
15: ce

C

~~{a} {b} {c} {d} {e} {f}~~
~~{aef} {b} {cd}~~ ~~{ef}~~
~~{aef} {bcd}~~
{a b c d e f}

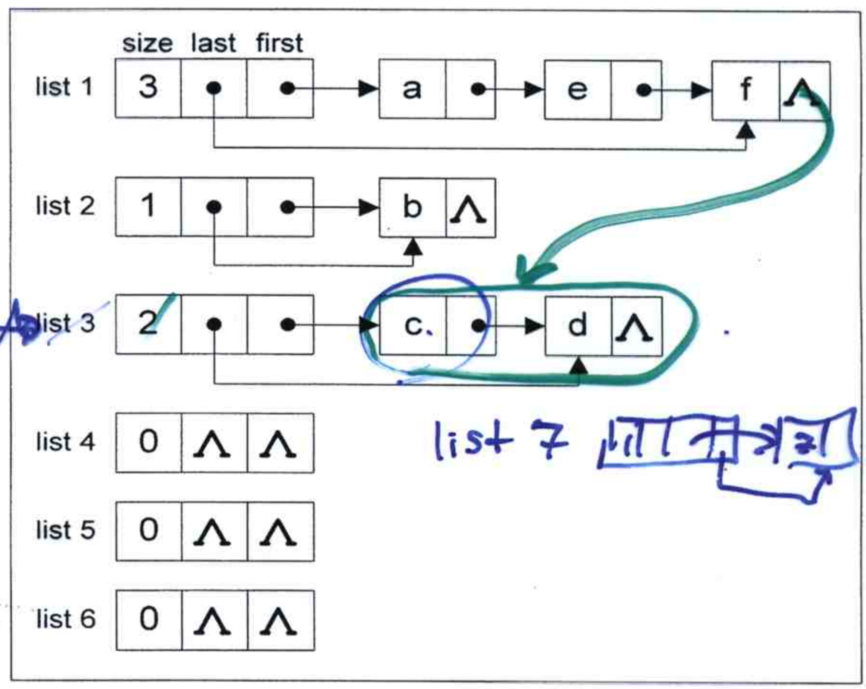
Implementing Disjoint Subsets (method 1)

1. Quick Find (optimizes the find operation)
   - uses a set of linked lists to store subsets
   - maintains an index array to identify which subset an element belongs to
   - ***find(x) is fast*** because it is simply a lookup in the index array to get the subset
   - ***union(x, y) is not so fast*** because we append y's list to x's list, but then we have to update all of y's entries in the index array

used for find(x).

index

element	subset
a	list 1
b	list 2
c	1* ~~list 3~~
d	1 ~~list 3~~
e	list 1
f	list 1

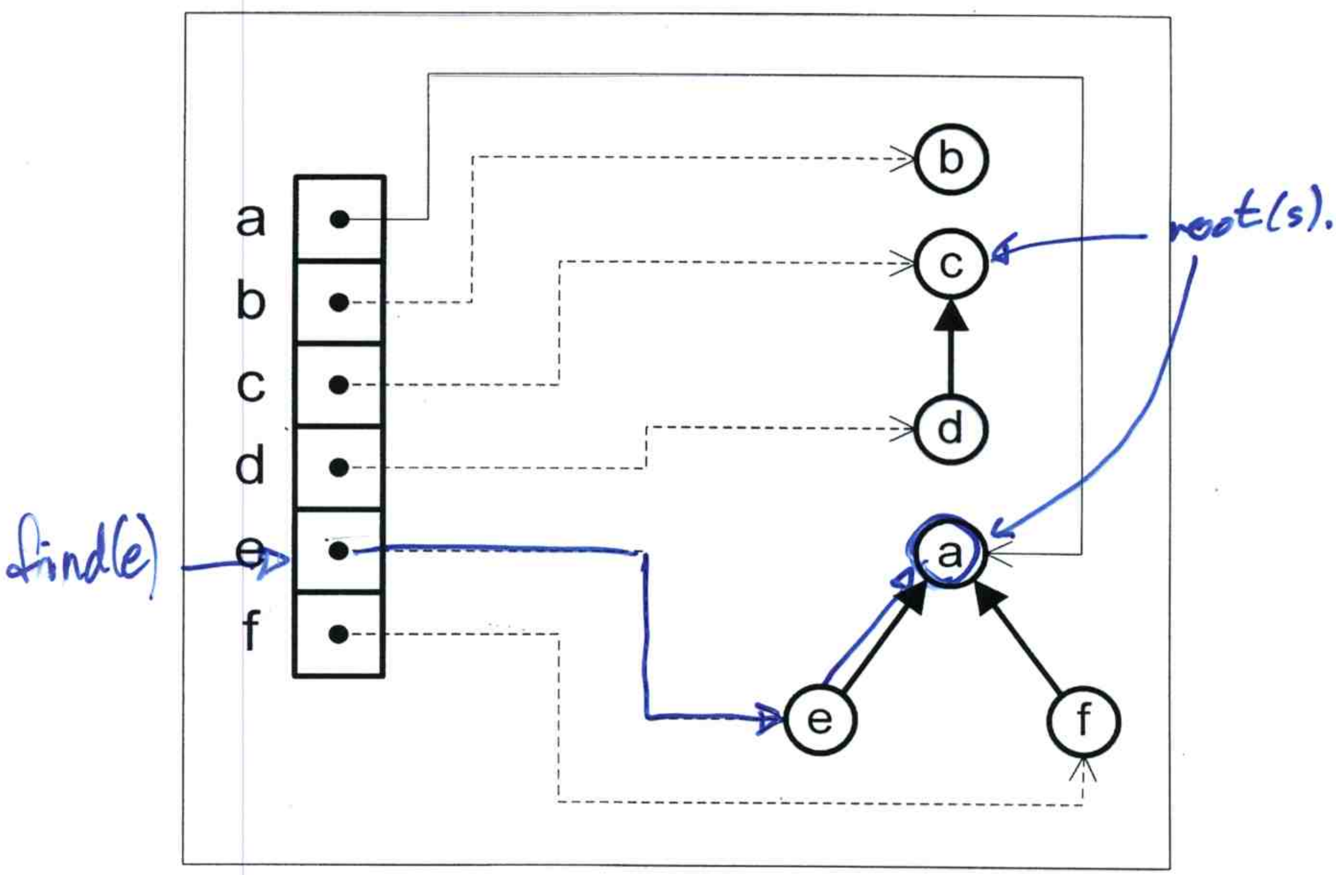


find(c).

each list is a subset.

Implementing Disjoint Subsets (method 2)

1. Quick Union (optimizes the union operation)
   - uses a *set of rooted trees* to store subsets
   - maintains an *array of pointers to tree nodes* to identify which subset an element belongs to
   - ***union(x, y) is fast*** because we simply connect the root of y to the root of x
   - ***find(x) is not so fast*** because we traverse X's tree from node x to the root



union

union (d, e)

→ find d ⇒ set c

→ find e ⇒ set a

connect set a to root of set c