Prim's Implementation - Continued.

Lets assume that there is an array Entry[1...n]. For each $v \in V - S$, Entry[v] points to priority queue entry $(v, \delta(v), c(v, \delta(v)))$ of v.

For $v \in S$, Entry[v] is null.

$T \leftarrow \emptyset$

Entry[s] $\leftarrow$ null // Take $s = 1$.

For each $v \in Adj[s]$
  Set Entry[v] to $(v, s, c(v,s))$
  and insert into priority queue.

For each $v \in V \setminus \{s\}$,
  If Entry[v] = null (priority queue entry for v has not been created), set Entry[v] to $(v, -, \infty)$ and insert into priority queue.

For $i \leftarrow 1$ to $n-1$ do

Do Extract-min on priority queue.

Suppose this yields $(v, u, c(v,u))$.

For each $w \in Adj[v]$

If $Entry[w] \neq null$

Suppose $Entry[w]$ points to $(w, -, c)$. If $c(v,w) < c$, set $Entry[w]$ to $(w, v, c(w,v))$ and update priority queue.

$Entry[v] \leftarrow null$

Add (u,v) to T.

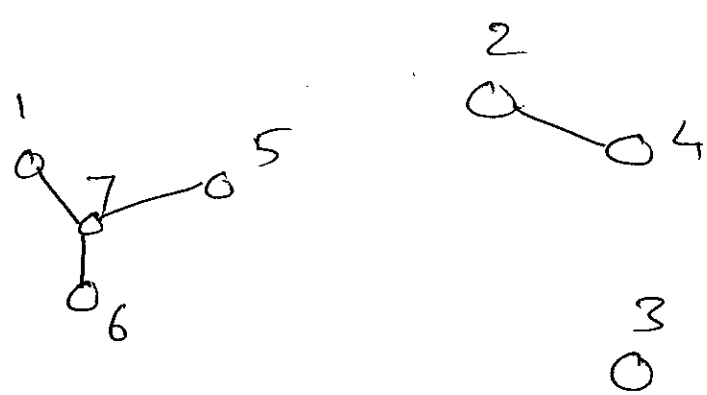
Running time is $O(n+m)$ plus time for $n-1$ inserts, $n-1$ extract mins, and m (why?) update keys.

Since each operation on a priority queue can be performed in $O(\log n)$ time, running time is $O(m \log n)$.

(We assume $n = O(m)$, which holds because the input graph is connected.)

<u>Kruskal's implementation</u>:

We need a data structure that maintains connected components. Suppose the graph constructed by Kruskal so far is:



This graph has 3 connected components:

$\{1, 7, 5, 6\}$, $\{2, 4\}$, $\{3\}$

Suppose algo examines edge (5,6) at this point. It needs to know if 5 and 6 are in the same connected component. When it discovers they are, it doesn't add edge (5,6). So connected components remain unchanged.

Suppose next edge examined is (5,2). Algorithm will add (5,2).
The connected components change — the components containing 5 and 2 have to be maged.

$\{1, 7, 5, 6, 2, 4\}$ $\{3\}$

A Union-Find data structure is one that maintains disjoint sets under the union operation and that supports the find operation: given an element,

return the set that it contains.

For the name of a set, we'll use one of ~~the~~ its elements. The choice is arbitrary but ~~is~~ should be consistent across all elements of the set.

Here are the operations supported by the Union-Find Structure:

Make Union Find (V) - Create a data structure with each element of V in a separate set.

Find (u) - Return name of set containing u

Union (A, B) - Merge the sets A and B into a single set.

With these primitives, here is Kruskal again.
We can assume that an edge $e = (u, v)$ is represented as a triple $(u, v, c(u,v))$.

Suppose $E \leftarrow e_1, e_2, \ldots, e_m$ in order of increasing costs.

```
MakeUnionFind(V)
    T ← ∅
For i ← 1 to m do
    Suppose e_i = (u, v, c(u,v)).
    If Find(u) ≠ Find(v),
        Add e_i to T.
        Union(Find(u), Find(v)).
    endif
endfor
```

Running time is $O(m \log m)$ plus time for $2m$ find operations, $n-1$ union operations (why?), and one MakeUnionFind operation on V.

How to implement the data structure.

Lets first consider a first, array based, implementation. There is an array Set[1..n]. Set(v) identifies the set to which v belongs.

In the example, before components are merged, the array would look like:

1	2	3	4	5	6	7
1	2	3	2	1	1	1

After the two components are merged, the array looks like:

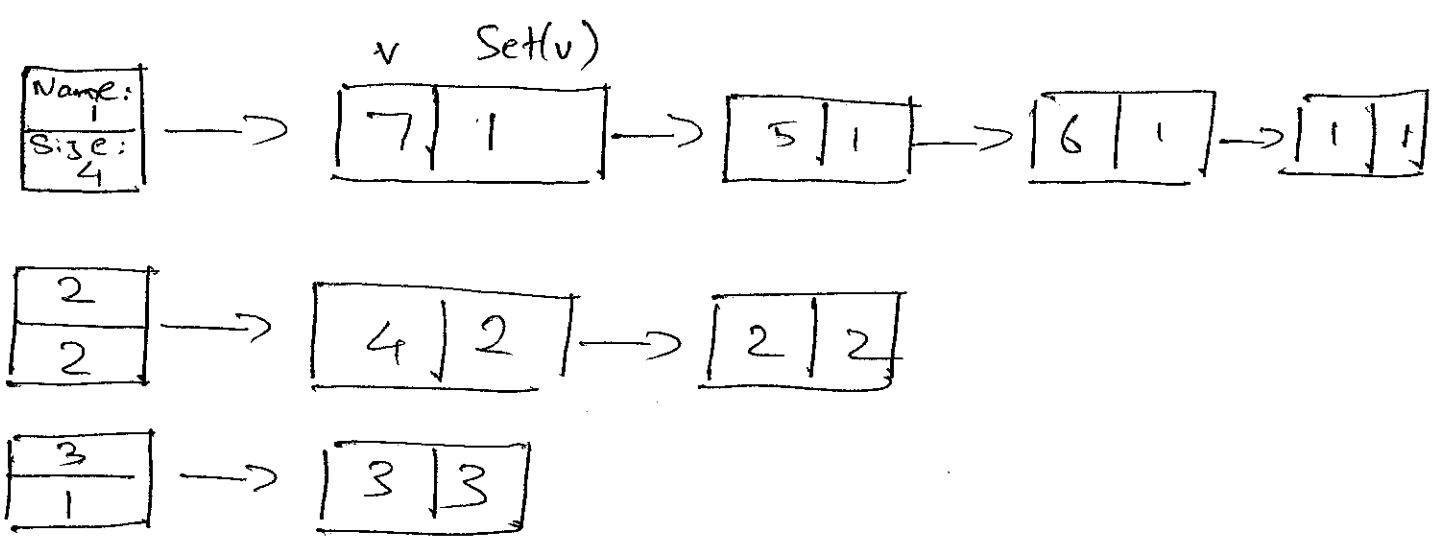
1	2	3	4	5	6	7
1	1	3	1	1	1	1

For such a data structure, ~~Union~~ MakeUnionFind(v) takes $O(n)$ time, Find() takes $O(1)$ time.

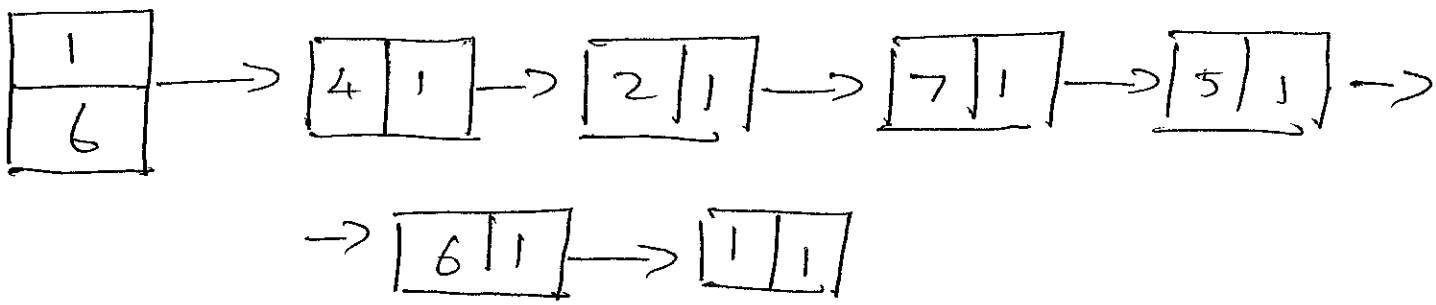
The problem is that each union operation requires us to scan the entire array, and thus takes $\Omega(n)$ time.

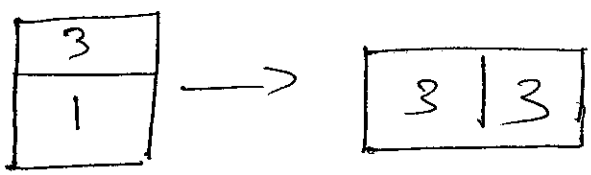
A better idea is to represent sets using linked lists. Corresponding to each $v \in V$, there is an object which has a Set field. Set(v) identifies the set to which v belongs. This object should be accessible from v.

In the example, just before the merge:



After the merge:





In a union operation, we merge the smaller set into the larger set. We can use the size field of the lists to determine the smaller set.

In this implementation,

MakeUnionFind(V) takes $O(n)$ time.

Find() takes $O(1)$ time.

Union(A, B) could still take $\Omega(n)$ time in worst case. For

example when $|A| = |B| = \frac{n}{2}$.

The key is to look at the combined running time of all $n-1$ unions.

~~(why are there n-1?)~~ Note that a single union operation takes time equal to the size of the smaller set (ignoring constant factors). <u>Charge</u> this running time to the elements of the smaller set, assigning a charge of 1 to each element.

Thus, running time for all the unions $= \sum_{v \in V}$ Total Charge to v

$= \sum_{v \in V}$ ~~No. of~~

Notice that each time v is changed, the size of the set in which v resides doubles (at least).

After v is changed for i^{th} time, size of set in which v resides is at least 2^i.

Now, we must have

$$2^{(\text{No. of times } v \text{ is changed})} \leq n.$$

So No. of times v is changed $\leq \log_2 n$.

We conclude that running time for $n-1$ unions $= O(n \log_2 n)$.

We also conclude that our implementation of Kruskal takes

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

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