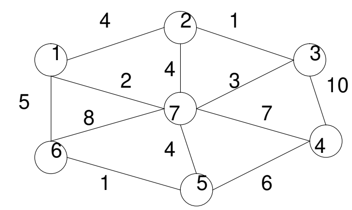
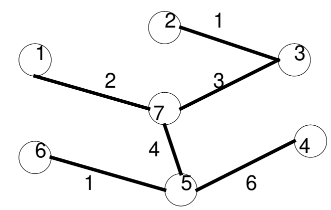
CSE 326
Lecture 22

Spring Quarter 1997

Minimum Spanning Tree

- Given a connected graph with weighted edges find the tree with minimum total weight which spans all the nodes of the graph.

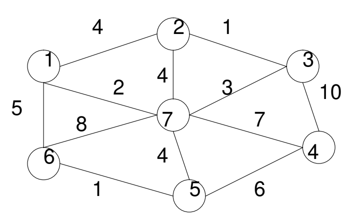




weight = 17

Kruskal's Algorithm (1956)

1. Start with all edges and an empty tree.
2. Repeat the following until a tree is formed.
   find the edge of minimum weight and remove it from the set of edges.
   if adding the edge does not form a cycle then do so.



Algorithm Design Problem

- Design an efficient program to implement Kruskal's algorithm.
- This is open for discussion.

Dynamic Equivalence

- Also commonly called "Disjoint Union/Find"
- We are given a set U = $\{a_1, a_2, a_3, \ldots, a_n\}$.
- We wish to maintain a partition of U, that is, a set of pairwise disjoint subsets of U. Each member of the partition has a unique name.

$$S_1, S_2, \ldots, S_m \subseteq U$$

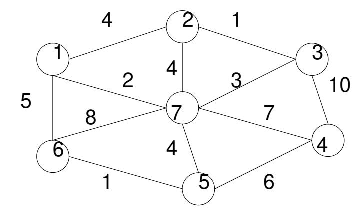
$$S_1 \cup S_2 \cup \ldots \cup S_m = U$$

$$S_i \cap S_j = \varnothing$$

- Union(s,t) creates a new set which is the union of the set named s and the set named t. The two sets are removed from the partition and the new one inserted. The new set has a unique name.
- Find(u) returns a unique name of the set S_i such that u is in S_i.

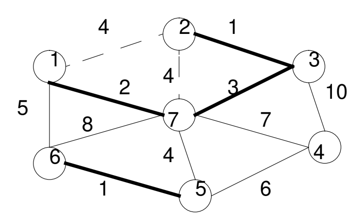
Example

- U = {1,2,3,4,5,6,7}
- Initial set of disjoint sets {1} {2} {3} {4} {5} {6} {7} . The name of each set is underlined.
- Find(2) = 2, Find(3) = 3
- Union(2,3) {1} {2,3} {4} {5} {6} {7}
- Find(5) = 5, Find(6) = 6
- Union(5,6) {1} {2,3} {4} {5, 6} {7}
- Find(1) = 1, Find(7) = 7
- Union(1,7) {1,7} {2,3} {4} {5,6}
- Find(3) = 2, Find(7) = 1
- Union(1,2) {1,7,2,3} {4} {5,6}
- Find(2) = 2, Find(7) = 2
- Find(1) = 2, Find(2) = 2



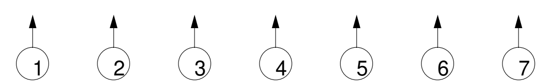
Example Continued

- Find(7) = 2, Find(5) = 5
- Union(2,5) {1,7,2,3,5,6} {4}
- Find(1) = 2, Find(6) = 2
- Find(5) = 2, Find(4) = 4
- Union(2,4) {1,7,2,3,5,6,4}

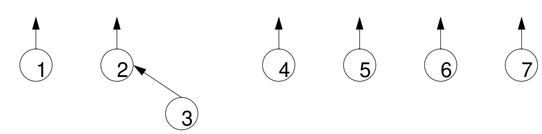


Up-Tree Implementation of Dynamic Equivalence

- A set is implemented as an up-tree connected by parent pointers, the root of the tree is the unique name of the set.
- A set of disjoint set is implemented as a set of up-trees.
- Example {1} {2} {3} {4} {5} {6} {7}

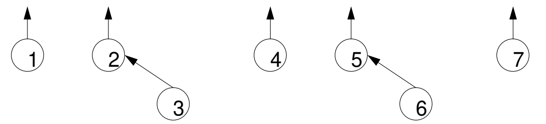


- Union(2,3) {1} {2,3} {4} {5} {6} {7}

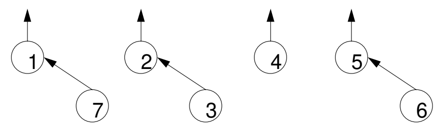


Up-Tree Forest

- Union(5,6)

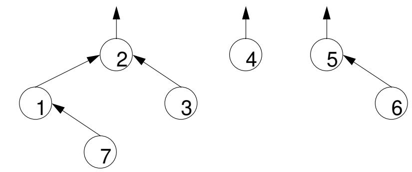


- Union(1,7)

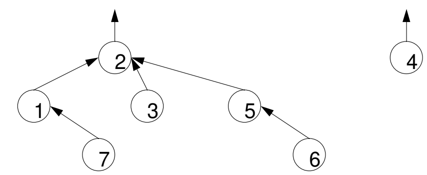


Up-Tree Forest

- Union(1,2)

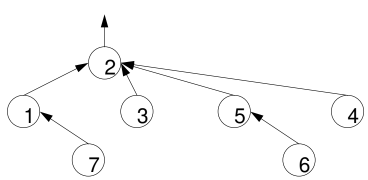


- Union(2,5)



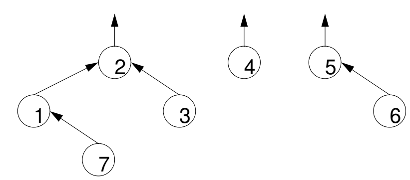
Up-Tree Forest

- Union(2,4)



Implementation Details

- The up-tree forest could be implemented with simple nodes with a key value field and a parent field.
- There is a very simple array implementation.



	1	2	3	4	5	6	7
up	2	0	2	0	0	5	1

Implementation of Operations

- Union(m,n) takes the union of the sets whose roots are m and n. We assume that m does not equal n.

  Union(m,n) { up[n] := m }

- Find(i) finds the root of the up-tree containing i.

  Find(i) {

  local int r;
  r := i
  while not(up[r] = 0) do r := up[r];
  return(r) }

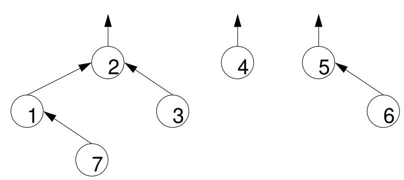
Time Complexity

- Union has O(1) worst case time.
- Find has O(n) worst case time.
- The time complexity of dynamic equivalence can be improved vastly by:

  1. Weighted union - always point the smaller up-tree to the larger one.

  2. Path compression - as part of a Find point all nodes on the path to the root directly to the root.

Weighted Union



	1	2	3	4	5	6	7
up	2	0	2	0	0	5	1
weight	-	4	-	1	2	-	-

- W-Union(m,n) {

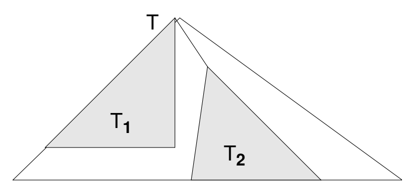
  if weight(m) $\leq$ weight(n) then
  {up[m] := n; weight[n] := weight[n] + weight[m]}
  else
  {up[n] := m; weight[m] := weight[m] + weight[n]}}

Complexity of Weighted Union / Find

- Lemma: With weighted unions an up-tree of height h has at least 2^h nodes.
- Proof by induction on h.

  Basis: h = 0. The only up-tree of height 0 is a single node and $2^0 = 1$.

  Inductive Step: Assume the lemma is true for all h' < h and h > 0. Let T be an up-tree of height h. Consider the weighted union step when a subtree of T first reached height h.



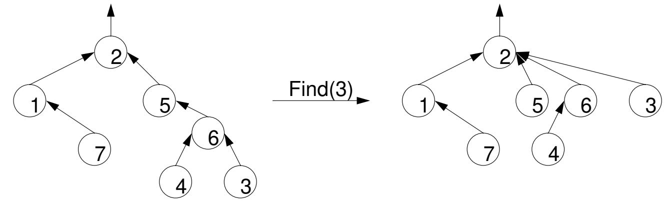
By the inductive hypothesis T_2 has at least 2^{h-1} nodes. By the weighted union operation T_1 has at least as many nodes as T_2. Thus, T has at least $2^{h-1} + 2^{h-1} = 2^h$ nodes.

Time Complexity of Weighted Union / Find

- W-Union(m,n) is O(1).
- Find(i) is O(log n).

Path Compression

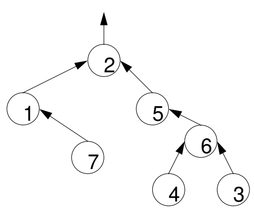
- On a Find point all nodes on the path to the root directly to the root.



Path Compression Code

- Find(i) {

```
local int r,t;
r := i
while not(up[r] = 0) do r := up[r];
t := up[i];
if not(t=0) then
  while not(t=r) do {up[i] := r; i := t; t := up[t]}
return(r) }
```



Complexity of Weighted Union and Path Compression

- Tarjan proved m weighted union and path compression find operations on a set of n elements has worst case time complexity O(m α(m,n)).

  α(m,n) is called "inverse ackermann's function". It is very, very slow growing. Indeed, $\alpha(m,n) \leq 4$ for all practical values of m and n.
- For all practical purposes implementing dynamic equivalence with weighted union and path compression find yields an essentially amortized constant time per operation data structure.
- In many practical cases neither weighted union nor path compression find are needed. When you need dynamic equivalence you have to decide if long paths can develop. If they can in your application then you need to introduce either weighted union or path compression find or both.