(21.1) Disjoint Sets

{ Chap 21
21.1, 21.2, 21.3

A disjoint set ADT maintains a collection of dynamic sets

$$\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$$

which are pairwise disjoint: $S_i \cap S_j = \phi$ for $i \neq j$. Each set in the collection is identified by a distinguished member called its representative.

Ex.

$$\mathcal{S} = \{\{a, b, c\}, \{d, e\}, \{f\}, \{g, h\}\}$$

In most applications we don't care which member of a set is used as its representative.

We write S_x for the (unique) member of $\mathcal{S}$ containing x. Note x need not be the representative of S_x. In the above example

$$\mathcal{S} = \{S_a, S_d, S_f, S_g\}.$$

A disjoint set ADT supports the following operations.

MakeSet(x)
CREATES A NEW SET $\{x\}$ IN $\mathcal{S}$. x MAY NOT BELONG TO ANY EXISTING MEMBER OF $\mathcal{S}$ SINCE THE COLLECTION IS TO BE DISJOINT.

$$\mathcal{S} = \mathcal{S} \cup \{\{x\}\}$$

Union(x,y)
UNITES SETS S_x AND S_y. PRECONDITION: $S_x \neq S_y$. THIS OPERATION REDUCES THE NUMBER OF SETS IN $\mathcal{S}$ BY 1.

$$\mathcal{S} = \mathcal{S} - \{S_x\} - \{S_y\} \cup \{S_x \cup S_y\}.$$

FindSet(x)
RETURNS (A POINTER TO) THE REPRESENTATIVE OF THE SET S_x.

IN WHAT FOLLOWS WE WILL ANALYZE RUN TIMES OF SEQUENCES OF THESE OPERATIONS. WE ADOPT THE CONVENTIONS THAT

$$n = \#\text{MakeSet}$$

AND

$$m = (\#\text{MakeSet}) + (\#\text{Union}) + (\#\text{FindSet})$$

OPERATIONS. OBSERVE THAT SINCE Union REDUCES $|\mathcal{S}|$ BY 1, WE HAVE

$$(\#\text{union}) \leq n - 1.$$

WE ANALYZE SEQUENCES OF OPERATIONS SINCE, AS WE SHALL SEE, THE RUN TIME OF THESE OPERATIONS WILL DEPEND ON THE PARTICULAR HISTORY OF THE DATA STRUCTURE.

TWO QUESTIONS ARISE: (1) OF WHAT USE IS THIS NEW DATA STRUCTURE?, AND (2) HOW CAN IT BE IMPLEMENTED?

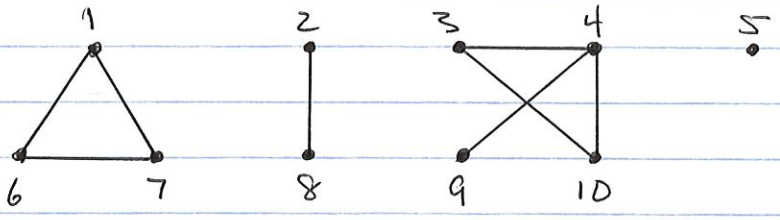
TO ANSWER (1) WE PRESENT THE FOLLOWING ALGORITHM WHICH DETERMINES THE CONNECTED COMPONENTS OF AN UNDIRECTED GRAPH $G = (V, E)$.

```
Components(G)
1.) for each v ∈ V
2.)     MakeSet(v)
3.) for each (u,v) ∈ E
4.)     if FindSet(u) ≠ FindSet(v)
5.)         Union(u,v)
```

```
SameComponent(u,v)
1.) if FindSet(u) == FindSet(v)
2.)     return true
3.) return false
```

ANOTHER USE OF THE DISJOINT SET DATA STRUCTURE IS TO DETERMINE A MWST, AS WE'LL SEE WHEN WE RETURN TO CHAP. 23.

Ex



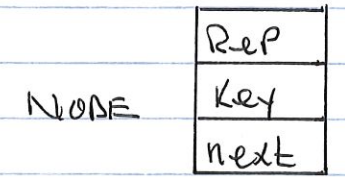
EDGE	$\mathcal{A}$
	{1} {2} {3} {4} {5} {6} {7} {8} {9} {10}
(1,7)	{1,7} {2} {3} {4} {5} {6} {8} {9} {10}
(2,8)	{1,7} {2,8} {3} {4} {5} {6} {9} {10}
(3,4)	{1,7} {2,8} {3,4} {5} {6} {9} {10}
(6,7)	{1,6,7} {2,8} {3,4} {5} {9} {10}
(9,4)	{1,6,7} {2,8} {3,4,9} {5} {10}
(4,10)	{1,6,7} {2,8} {3,4,9,10} {5}
(1,6)	" " " "
(3,10)	" " " "

SECTIONS 21.2 & 21.3 PRESENT TWO METHODS OF IMPLEMENTING A DISJOINT SET DATA STRUCTURE, WHICH ANSWERS QUESTION (2).

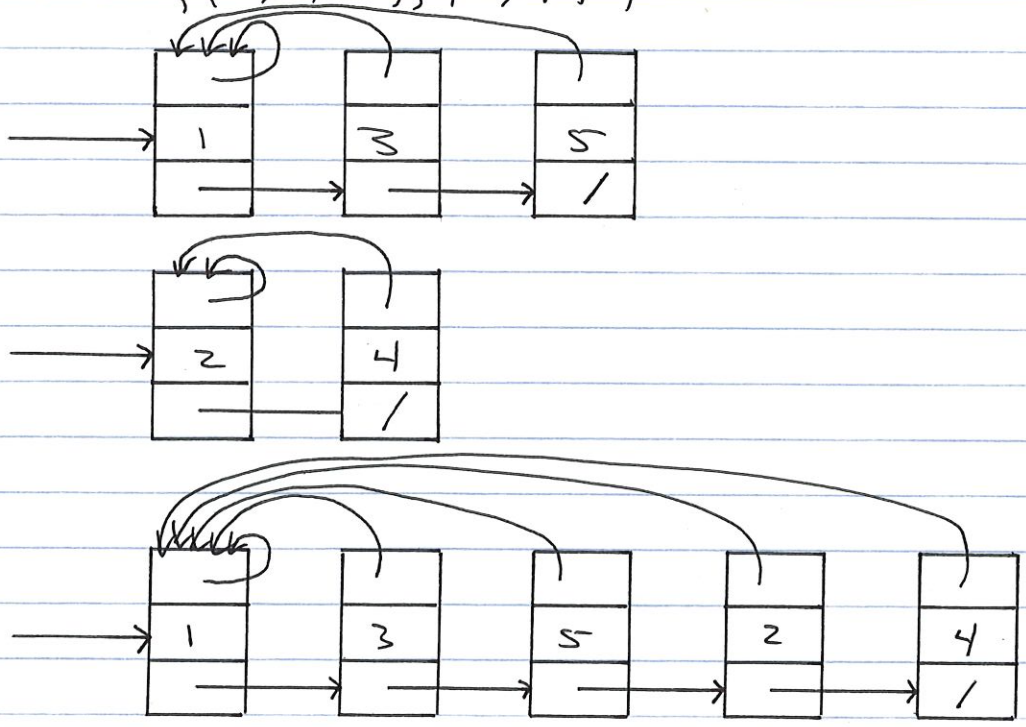
(21.2) Linked List Representation

The sets in $\mathcal{S}$ can be represented by linked lists where the first object in a list serves as that set's representative.

Each list node maintains a key field as well as pointers to the next node and the representative node.



Ex. $\mathcal{S} = \{\{1,3,5\},\{2,4\}\}$



Result of Union(3,4)

MakeSet(x)
1.) CREATE NEW list WITH SOLE OBJECT x

FindSet(x)
1.) return Rep[x]

Union(x,y)
1.) CONCATENATE list S_x AND S_y
2.) UPDATE Rep POINTERS IN S_y TO POINT TO Rep[x]

MakeSet AND FindSet RUN IN $\Theta(1)$ TIME WHILE Union TAKES $\Theta(|S_y|)$ TIME.

THE $\Theta(|S_y|)$ RUN TIME FOR Union CAUSES CERTAIN SEQUENCES OF OPERATIONS TO HAVE POOR TOTAL RUN TIME.

Ex.

MakeSet(x_1)
⋮
MakeSet(x_n)
} n

Union(x_2, x_1)
Union(x_3, x_2)
⋮
Union(x_n, x_{n-1})
} $n-1$

$m = n + (n-1) = 2n - 1$

THIS SEQUENCE CREATES A SINGLE LINKED LIST

THE n MakeSet OPERATIONS TAKE $\Theta(n)$ TIME. THE $n-1$ Union OPERATIONS PERFORM A TOTAL OF

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

REPRESENTATIVE UPDATES TAKING $\Theta(n^2)$ TIME. THE TOTAL RUN TIME OF THIS SEQUENCE IS THEREFORE $\Theta(n+n^2) = \Theta(n^2)$. IN TERMS OF $m = 2n-1$ THIS IS $\Theta(m^2)$ TIME TO PERFORM A TOTAL OF m OPERATIONS. IN OTHER WORDS $\Theta(m)$ TIME PER OPERATION, ON AVERAGE.

WHY ARE WE INTERESTED IN THE RUN TIME OF SEQUENCES OF OPERATIONS? BECAUSE IN A DISJOINT SET DATA STRUCTURE, THE RUN TIME OF AN OPERATION (e.g. Union) DEPENDS ON THE PARTICULAR HISTORY OF OPERATIONS PERFORMED.

ANALYZING THE RUN TIME OF A SEQUENCE, THEN AVERAGING OVER ALL OPERATIONS PERFORMED, IS CALLED <u>AMORTIZED ANALYSIS</u>. THIS IS OFTEN DONE IN SITUATIONS WHERE HISTORY MATTERS.

GOING BACK TO THE PREVIOUS EXAMPLE, WE SEE THE PROBLEM WITH THAT SEQUENCE IS THAT WE ALWAYS APPEND THE LONGER LIST TO THE SHORTER ONE, WHICH INCREASES THE NUMBER OF REPRESENTATIVE UPDATES.

WE CAN IMPROVE PERFORMANCE BY STORING THE LENGTH OF EACH LIST, THEN ALWAYS APPEND THE SHORTER LIST TO THE LONGER, CAUSING FEWER REPRESENTATIVE UPDATES. WE CALL THIS THE WEIGHTED UNION HEURISTIC.

THEOREM
USING THE WEIGHTED UNION HEURISTIC WITH LINKED LIST REPRESENTATION OF DISJOINT SETS, ANY SEQUENCE OF m MakeSet, Union, AND FindSet OPERATIONS, n OF WHICH ARE MakeSets, TAKES TIME $O(m + n \lg n)$.

PROOF: P. 566

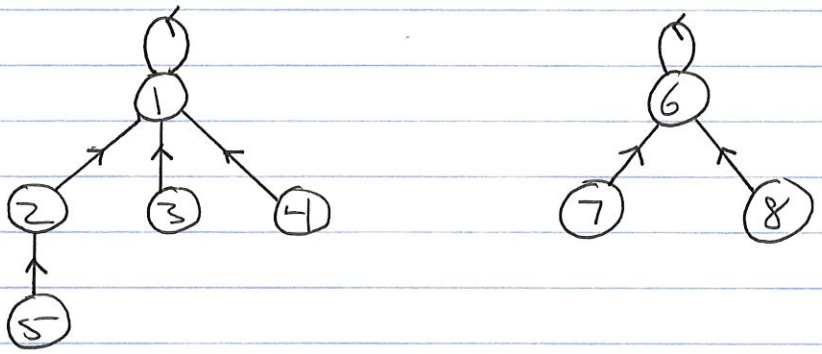
NOTE THAT THE PREVIOUS EXAMPLE TAKES TIME $\Theta(n) = \Theta(m)$ WITH WEIGHTED UNION HEURISTIC, WHICH IS SOMEWHAT BETTER THAN THAT GUARANTEED BY THE ABOVE THEOREM. AVERAGING OVER ALL m OPERATIONS IN THIS EXAMPLE GIVES $\Theta(1)$ AMORTIZED TIME.

(21.3) Disjoint Set Forests

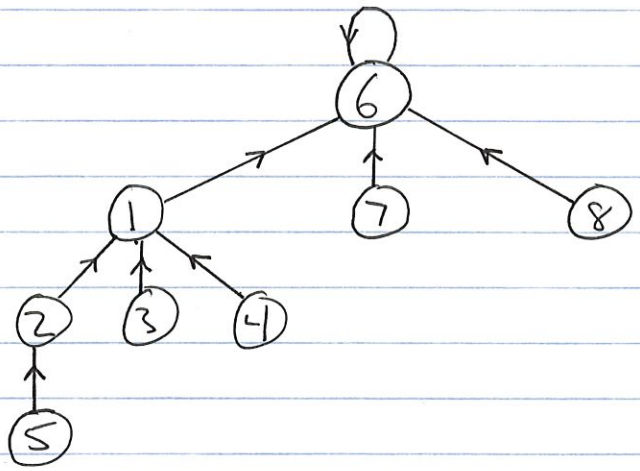
Another implementation strategy is to represent each set in $\mathcal{S}$ by a rooted tree, i.e. $\mathcal{S}$ is a **Disjoint Set Forest**.

Each node points only to its parent. The root of each tree is its own parent, and is its set's representative.

Ex



Result of Union(3, 8):



Convention: Union(x, y) causes Rep[x] to point to Rep[y].

MakeSet CREATES A TREE WITH JUST ONE NODE WHICH POINTS TO ITSELF. Union CAUSES THE ROOT OF ONE TREE TO POINT TO THE ROOT OF ANOTHER. BOTH OPERATIONS HAVE COST $\Theta(1)$.

FindSet FOLLOWS PARENT POINTERS UNTIL THE ROOT IS REACHED. THE NODES VISITED ON THIS PATH TOWARD THE ROOT CONSTITUTE A FIND PATH. THE COST OF FindSet IS Θ(length-of-findPath).

IT IS POSSIBLE USING THESE OPERATIONS TO CREATE A TREE WHICH IS SIMPLY A LINEAR LINKED LIST.

EX.

MakeSet(x_1)
⋮
MakeSet(x_n)
Union(x_n, x_{n-1})
Union(x_{n-1}, x_{n-2})
⋮
Union(x_3, x_2)
Union(x_2, x_1)

THIS SEQUENCE IS NOT BAD IN ITSELF (RUN TIME $\Theta(2n-1) = \Theta(m)$), BUT IT CREATES A TREE IN WHICH FindSet will RUN slowly (i.e. IN $\Theta(n)$ TIME.)

TWO HEURISTICS ARE USED TO IMPROVE THE RUN TIME OF SEQUENCES WHICH INCLUDE MANY FindSet OPERATIONS.

Union By RANK

FOR EACH NODE x, MAINTAIN A VALUE rank[x], WHICH IS AN UPPER BOUND ON THE HEIGHT OF x.

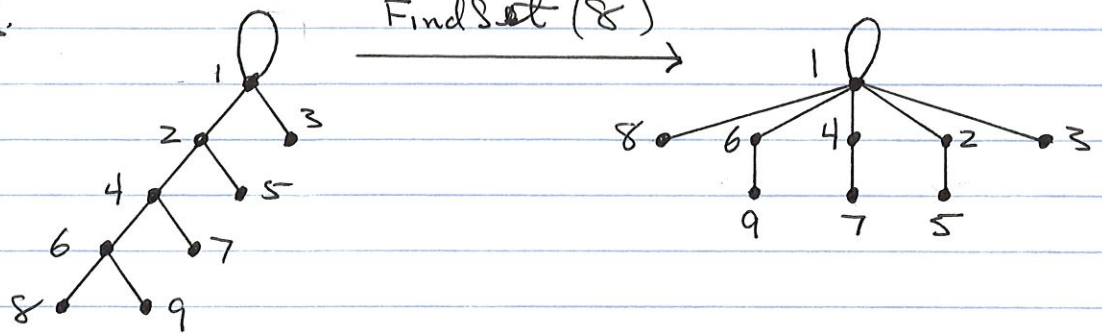
$$height(x) \leq rank[x].$$

MODIFY Union SO THAT THE ROOT WITH SMALLER RANK IS MADE TO POINT TO THE ROOT WITH LARGER RANK.

PATH COMPRESSION

THIS IS A MODIFICATION OF FindSet WHICH CAN REDUCE THE HEIGHT OF A TREE. IT INVOLVES TWO PASSES: PERFORM ONE PASS UP THE FIND PATH TO FIND THE ROOT, THEN MAKE ANOTHER PASS DOWN THE FIND PATH CAUSING EACH NODE ALONG THAT PATH TO POINT TO THE ROOT.

Ex.



FindSet(8) ALTERS THE TREE AS ABOVE, THEN RETURNS 1.

MakeSet(x)

1.) $P[x] = x$
2.) $rank[x] = 0$

Union(x, y) (USING UNION-BY-RANK)

1.) Link(FindSet(x), FindSet(y))

Link(x, y) (Pre: x, y BOTH ROOT NODES)

1.) if $rank[x] > rank[y]$.
2.) $\quad P[y] = x$
3.) else
4.) $\quad P[x] = y$
5.) $\quad$ if $rank[x] = rank[y]$
6.) $\quad\quad rank[y]$ ++

NOTE: AS LONG AS ONLY MakeSet AND Link OPERATIONS ARE PERFORMED, $rank[x]$ is ACTUALLY EQUAL TO $height(x)$. THIS CAN BE PROVED BY INDUCTION ON THE NUMBER OF Link OPERATIONS PERFORMED.

FindSet(x) (USING PATH COMPRESSION)

1.) if $x \neq P[x]$
2.) $P[x] = \text{FindSet}(P[x])$
3.) return $P[x]$.

WE TRACE FindSet ON THE PRECEDING EXAMPLE:

FindSet(8): 1.) $8 \neq P[8]$
2.) $P[8] \leftarrow \text{FindSet}(6)$
3.) return 1

FindSet(6) 1.) $6 \neq P[6]$
2.) $P[6] \leftarrow \text{FindSet}(4)$
3.) return 1

FindSet(4) 1.) $4 \neq P[4]$
2.) $P[4] \leftarrow \text{FindSet}(2)$
3.) return 1

FindSet(2) 1.) $2 \neq P[2]$
2.) $P[2] \leftarrow \text{FindSet}(1)$
3.) return 1

FindSet(1) 1.) $1 = P[1]$
2.)
3.) return 1

NOTE THAT FindSet(x) MAY REDUCE THE HEIGHT OF THE TREE CONTAINING x, WHILE IT DOES NOT ALTER ANY RANKS. THUS FindSet OPERATIONS MAINTAIN THE INEQUALITY

$$height(x) \leq rank[x]$$

FOR ALL NODES x.

THE UNION-BY-RANK AND PATH COMPRESSION HEURISTICS WORK TOGETHER TO KEEP THE HEIGHTS OF TREES SMALL, WHICH LOWERS THE COST OF FindSet.

THEOREM

USING THE UNION-BY-RANK AND PATH COMPRESSION HEURISTICS, ANY SEQUENCE OF m MakeSet, Union, AND FindSet OPERATIONS, n OF WHICH ARE MakeSets, CAN BE PERFORMED IN TIME

$$O(m \cdot \alpha(n))$$

WHERE $\alpha(n)$ IS AN EXTREMELY SLOW GROWING FUNCTION (DEFINED IN 21.4). IN ANY CONCEIVABLE APPLICATION $\alpha(n) \leq 4$, SO THE ABOVE RUN TIME IS EFFECTIVELY LINEAR, i.e. $O(m)$.

AMORTIZING THIS COST OVER ALL OPERATIONS GIVES $O(1)$ TIME PER OPERATION ON AVERAGE, WHICH IS VERY EFFICIENT.

BACK TO SEC. 23.2

RECALL THAT KRUSKAL'S ALGORITHM GROWS SEVERAL TREES SIMULTANEOUSLY UNTIL THEY MERGE INTO A SINGLE MWST. THE IMPLEMENTATION BELOW USES A DISJOINT SET DATA STRUCTURE TO MAINTAIN A COLLECTION OF VERTEX SETS. EACH SET IN THIS COLLECTION IS THE VERTEX SET OF ONE OF THE TREES UNDER CONSTRUCTION.

NOTE THE SIMILARITY OF THIS ALGORITHM TO THE ALGORITHM FOR DETERMINING CONNECTED COMPONENTS FROM 21.1.

KRUSKAL (G, w)

1.) $A = \emptyset$
2.) for each $v \in V$
3.) MakeSet(v)
4.) SORT E BY WEIGHT w (NON-DECREASING)
5.) for each $(u,v) \in E$ (IN SORTED ORDER)
6.) if FindSet$(u) \neq$ FindSet(v)
7.) $A = A \cup \{(u,v)\}$
8.) Union(u,v)
9.) return A

ON EACH ITERATION OF LOOP 5-8 WE PICK THE LIGHTEST EDGE JOINING TWO DISTINCT TREES AND ADD THAT TO A. I.E. AMONGST ALL EDGES WHOSE ADDITION TO A WOULD NOT CREATE A CYCLE, WE PICK ONE OF MINIMUM WEIGHT. THE RESULTING SET $A \subseteq E$ CONSTITUTES THE EDGES OF A MWST IN G.

Run Time

Assume we use the disjoint set forest implementation with union-by-rank and path compression heuristics, which is the asymptotically fastest known.

Initialization of A in line 1 costs $O(1)$ time. Loop 2–3 does $|V|$ MakeSet operations and loop 5–8 $O(|E|)$ FindSet and Union operations. Thus the disjoint set operations have total cost $O((|V|+|E|)\cdot\alpha(|V|))$.

Now since G is connected (otherwise there is no spanning tree) we have $|E| \geq |V| - 1$, whence $|V| \leq |E| + 1 = O(|E|)$, so the disjoint set operations cost $O(|E|\alpha(|V|))$

The sort on line 4 costs $O(|E| \lg |E|)$ time (assuming we use MergeSort or HeapSort, or a similarly efficient sort.)

Also note that $\alpha(|V|) = O(\lg|V|) = O(\lg|E|)$, so the Run time of Kruskal is

$$O(|E| \lg |E|).$$

But also since G is simple, $|E| < |V|^2$, so $\lg|E| = O(\lg|V|)$, and the Run time of Kruskal can be expressed as

$$O(|E| \lg |V|)$$

which is the same as Prim.