Disjoint set structures

- Last time
  - Splay Trees
- Today
  - Disjoint set structures

Disjoint set structures

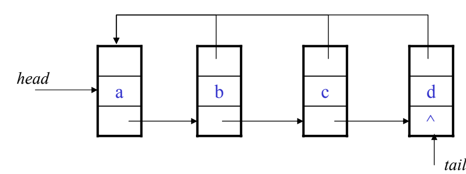
- Definition
  - A collection of disjoint dynamic sets
  - Each set is identified by a representative which is some member of the set
- Three functions
  - *MakeSet(x)* – create a new set whose only member is x
  - *FindSet(x)* –find the set that contains *x;* return the representative
  - *Union(x, y)* – unites the corresponding sets that contains *x* and *y* respectively and choose a representative for the combined set
- Used to implement partition

Goal

- Assumption
  - Assume *m* operations, MakeSet, FindSet, and Union
  - *n* of which are MakeSet operations
- Goal:
  - Find efficient structures and algorithms for all these operations

Linked List Representation

- Represent each set as a link list
  - The first object is the representative
  - A pointer, *head*, pointing to the representative
  - A pointer, *tail*, pointing to the last object of the list
    - For easy union
  - Each object has two pointers
    - *next*: points to the next object in the list
    - *rep*: points to the representative



Union for Linked-list

```
Union(x, y) // add list x to the tail of y
{
   cur = x.head;

   while (cur != null) {
      cur.rep = y;
      cur = cur.next;
   }

   y.tail.next = x.head;
   y.tail = x.tail;
}
```

What's the problem

- MakeSet and FindSet take constant time
- For Union, we have to update the *rep* pointer of every node on one set.
  - Worst case: $\Theta(n^2)$
    - MakeSet(x_1)
    - MakeSet(x_2)
    - …
    - MakeSet(x_n)
    - Union(x_1, x_2)
    - Union(x_2, x_3)
    - …
    - Union(x_{n-1}, x_n)

A weighted-union heuristic

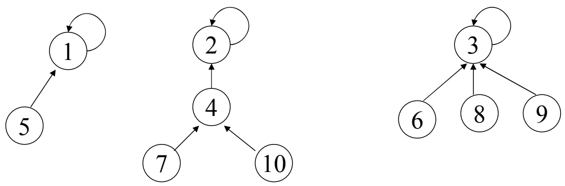
- Maintain the *length* for each list
- Always join the shorter list to the longer list
- Theorem: Using the linked-list representation of disjoint set and the weighted-union heuristic, a sequence of m operations, n of which are MakeSet, takes $O(m+n\lg n)$ time

Union for Linked-list

```
WeightedUnion(x, y)
{
   if (x.length > y.length) {
     Union(y, x);
     x.length += y.length;
   } else {
     Union(x, y);
     y.length += x.length
   }
}
```

Disjoint-set forest

- Represent each set as a rooted tree
  - Each member points only to its parent
  - The root is the representative
  - The root's parent is itself



Implementation: disjoint-set forest

```
FindSet(x)
{
  r = x;
  while (r.parent != r)
    r = r.parent;
  return r;
}
```

$\Theta(n)$ in worst case

```
MakeSet(x)
{
   x.parent = x;
}
```

$\Theta(1)$

```
Union(x, y)
{
   Link(FindSet(x), FindSet(y));
}
```

$\Theta(n)$ in worst case

```
Link(x, y)
{
   x.parent = y;
}
```

$\Theta(1)$

$\Theta(n^2)$ in worst case for m operations when $m \in \Theta(n)$

Improvements: two heuristics

- Union by rank
  - Rank: for each node, its rank is an upper bound on its height
  - Union: the root with smaller rank is made pointed to the root with larger rank
- Path compression
  - Make each node on the *find path* directly point to the root
  - *find path*: the path FindSet goes through

Union by rank

```
Link(x,y)
{
   if (x.rank > y.rank) {
     y.parent = x;
   } else {
     x.parent = y;
     if (x.rank == y.rank)
       y.rank++;
   }
}
```

$\Theta(1)$

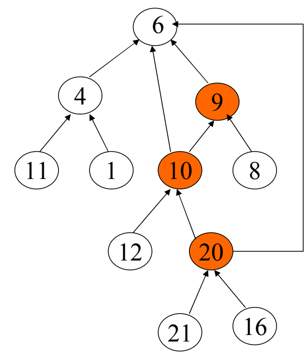
Path compression

- Squash the path when doing FindSet(), so the next FindSet() will be likely quicker (path compression).
  - first pass to find the root
  - second pass change the pointers along the path to the root and make them all point to the root

```
FindSet(x)
{
    r = x;
    while (r.parent <> r)
        r = r.parent;

    i = x;
    while (i <> r) {
        j = i.parent;
        i.parent = r;
        i = j;
    }
    return r;
}
```

An example



Efficiency with the two heuristics combined

- A sequence of *m* operations, *n* of which are MakeSet, takes O(*m* $\alpha(n)$) time
  - Practically, $\alpha(n) <= 4$

A quickly growing function

$$A_k(j) = \begin{cases} j+1 & \text{if} \quad k=0 \\ A_{k-1}^{(j+1)}(j) & \text{if} \quad k \geq 1 \end{cases}$$

$$A_k^0(j) = j$$

$$A_k(j) \quad \text{means} \quad A_k^1(j)$$

$$A_k^{(j)}(i) = A_k(A_k^{(j-1)}(i))$$

$$A_1(1) = 3$$
$$A_2(1) = 7$$
$$A_3(1) = 2047$$
$$A_4(1) >> 10^{80}$$

Definition of *α(n)*

$$\alpha(n) = \begin{cases} 0 & for \quad 0 \le n \le 2 \\ 1 & for \quad n = 3 \\ 2 & for \quad 4 \le n \le 7 \\ 3 & for \quad 8 \le n \le 2047 \\ 4 & for \quad 2048 \le n \le A_4(1) \end{cases}$$