E pluribus unum (Out of many, one)

— Official motto of the United States of America

John: *Who's your daddy? C'mon, you know who your daddy is! Who's your daddy? D'Argo, tell him who his daddy is!"*
D'Argo: *I'm your daddy.*

— *Farscape*, "Thanks for Sharing" (June 15, 2001)

9 Data Structures for Disjoint Sets (October 10 and 15)

In this lecture, we describe some methods for maintaining a collection of disjoint sets. Each set is represented as a pointer-based data structure, with one node per element. Each set has a 'leader' element, which uniquely identifies the set. (Since the sets are always disjoint, the same object cannot be the leader of more than one set.) We want to support the following operations.

- MAKESET(x): Create a new set $\{x\}$ containing the single element x. The element x must not appear in any other set in our collection. The leader of the new set is obviously x.
- FIND(x): Find (the leader of) the set containing x.
- UNION(A, B): Replace two sets A and B in our collection with their union $A \cup B$. For example, UNION(A, MAKESET(x)) adds a new element x to an existing set A. The sets A and B are specified by arbitrary elements, so UNION(x, y) has exactly the same behavior as UNION(FIND(x), FIND(y)).

Disjoint set data structures have lots of applications. For instance, Kruskal's minimum spanning tree algorithm relies on such a data structure to maintain the components of the intermediate spanning forest. Another application might be maintaining the connected components of a graph as new vertices and edges are added. In both these applications, we can use a disjoint-set data structure, where we keep a set for each connected component, containing that component's vertices.

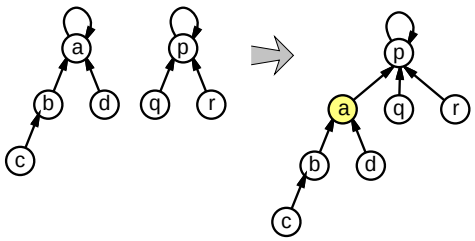
9.1 Reversed Trees

One of the easiest ways to store sets is using trees. Each object points to another object, called its *parent*, except for the leader of each set, which points to itself and thus is the root of the tree. MAKESET is trivial. FIND traverses the parent pointers up to the leader. UNION just redirects the parent pointer of one leader to the other. Notice that unlike most tree data structures, objects do *not* have pointers down to their children.

```
MAKESET(x):
  parent(x) ← x
```

```
FIND(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

```
UNION(x, y):
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  parent(ȳ) ← x̄
```



Merging two sets stored as trees. Arrows point to parents. The shaded node has a new parent.

MAKE-SET clearly takes $\Theta(1)$ time, and UNION requires only $O(1)$ time in addition to the two FINDs. The running time of FIND(x) is proportional to the depth of x in the tree. It is not hard to come up with a sequence of operations that results in a tree that is a long chain of nodes, so that FIND takes $\boxed{\Theta(n) \text{ time}}$ in the worst case.

However, there is an easy change we can make to our UNION algorithm, called *union by depth*, so that the trees always have logarithmic depth. Whenever we need to merge two trees, we always make the root of the *shallower* tree a child of the *deeper* one. This requires us to also maintain the depth of each tree, but this is quite easy.

```
MAKESET(x):
  parent(x) ← x
  depth(x) ← 0
```

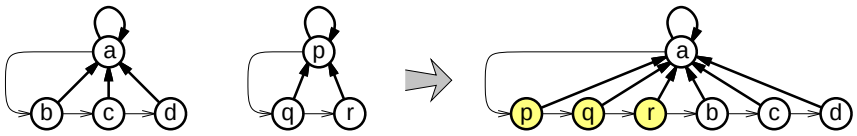
```
FIND(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

```
UNION(x, y)
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  if depth(x̄) > depth(ȳ)
      parent(ȳ) ← x̄
  else
      parent(x̄) ← ȳ
      if depth(x̄) = depth(ȳ)
          depth(ȳ) ← depth(ȳ) + 1
```

With this simple change, FIND and UNION both run in $\boxed{\Theta(\log n)}$ time in the worst case.

9.2 Shallow Threaded Trees

Alternately, we could just have every object keep a pointer to the leader of its set. Thus, each set is represented by a shallow tree, where the leader is the root and all the other elements are its children. With this representation, MAKESET and FIND are completely trivial. Both operations clearly run in constant time. UNION is a little more difficult, but not much. Our algorithm sets all the leader pointers in one set to point to the leader of the other set. To do this, we need a method to visit every element in a set; we will 'thread' a linked list through each set, starting at the set's leader. The two threads are merged in the UNION algorithm in constant time.



Merging two sets stored as threaded trees.
Bold arrows point to leaders; lighter arrows form the threads. Shaded nodes have a new leader.

```
MAKESET(x):
  leader(x) ← x
  next(x) ← x
```

```
FIND(x):
  return leader(x)
```

```
UNION(x, y):
  x̄ ← FIND(x)
  ȳ ← FIND(y)

  y ← ȳ
  leader(y) ← x̄
  while (next(y) ≠ NULL)
      y ← next(y)
      leader(y) ← x̄

  next(y) ← next(x̄)
  next(x̄) ← ȳ
```

The worst-case running time of UNION is a constant times the size of the *larger* set. Thus, if we merge a one-element set with another n-element set, the running time can be $\Theta(n)$. Generalizing this idea, it is quite easy to come up with a sequence of n MAKESET and $n-1$ UNION operations that requires $\Theta(n^2)$ time to create the set $\{1, 2, \ldots, n\}$ from scratch.

```
WORSTCASESEQUENCE(n):
  MAKESET(1)
  for i ← 2 to n
      MAKESET(i)
      UNION(1, i)
```

We are being stupid in two different ways here. One is the order of operations in WORSTCASESEQUENCE. Obviously, it would be more efficient to merge the sets in the other order, or to use some sort of divide and conquer approach. Unfortunately, we can't fix this; we don't get to decide how our data structures are used! The other is that we always update the leader pointers in the larger set. To fix this, we add a comparison inside the UNION algorithm to determine which set is smaller. This requires us to maintain the size of each set, but that's easy.

```
MAKEWEIGHTEDSET(x):
  leader(x) ← x
  next(x) ← x
  size(x) ← 1
```

```
WEIGHTEDUNION(x, y)
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  if size(x̄) > size(ȳ)
      UNION(x̄, ȳ)
      size(x̄) ← size(x̄) + size(ȳ)
  else
      UNION(ȳ, x̄)
      size(x̄) ← size(x̄) + size(ȳ)
```

The new WEIGHTEDUNION algorithm still takes $\Theta(n)$ time to merge two n-element sets. However, in an amortized sense, this algorithm is much more efficient. Intuitively, before we can merge two large sets, we have to perform a large number of MAKEWEIGHTEDSET operations.

Theorem 1. *A sequence of m* MAKEWEIGHTEDSET *operations and n* WEIGHTEDUNION *operations takes $O(m + n \log n)$ time in the worst case.*

Proof: Whenever the leader of an object x is changed by a WEIGHTEDUNION, the size of the set containing x increases by at least a factor of two. By induction, if the leader of x has changed k times, the set containing x has at least 2^k members. After the sequence ends, the largest set contains at most n members. (Why?) Thus, the leader of any object x has changed at most $\lfloor \lg n \rfloor$ times.

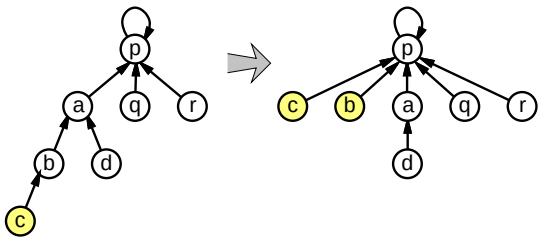
Since each WEIGHTEDUNION reduces the number of sets by one, there are $m - n$ sets at the end of the sequence, and at most n objects are *not* in singleton sets. Since each of the non-singleton objects had $O(\log n)$ leader changes, the total amount of work done in updating the leader pointers is $O(n \log n)$. □

The aggregate method now implies that each WEIGHTEDUNION has amortized cost $O(\log n)$.

9.3 Path Compression

Using unthreaded tress, FIND takes logarithmic time and everything else is constant; using threaded trees, UNION takes logarithmic amortized time and everything else is constant. A third method allows us to get both of these operations to have *almost* constant running time.

We start with the original unthreaded tree representation, where every object points to a parent. The key observation is that in any FIND operation, once we determine the leader of an object x, we can speed up future FINDs by redirecting x's parent pointer directly to that leader. In fact, we can change the parent pointers of all the ancestors of x all the way up to the root; this is easiest if we use recursion for the initial traversal up the tree. This modification to FIND is called *path compression*.



Path compression during FIND(c). Shaded nodes have a new parent.

```
FIND(x)
  if x ≠ parent(x)
      parent(x) ← FIND(parent(x))
  return parent(x)
```

If we use path compression, the 'depth' field we used earlier to keep the trees shallow is no longer correct, and correcting it would take way too long. But this information still ensures that FIND runs in $\Theta(\log n)$ time in the worst case, so we'll just give it another name: *rank*.

```
MAKESET(x):
  parent(x) ← x
  rank(x) ← 0
```

```
UNION(x, y)
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  if rank(x̄) > rank(ȳ)
      parent(ȳ) ← x̄
  else
      parent(x̄) ← ȳ
      if rank(x̄) = rank(ȳ)
          rank(ȳ) ← rank(ȳ) + 1
```

Ranks have several useful properties that can be verified easily by examining the UNION and FIND algorithms. For example:

- If an object x is not a set leader, then the rank of x is strictly less than the rank of its parent.
- Whenever parent(x) changes, the new parent has larger rank than the old parent.
- The size of any set is exponential in the rank of its leader: $\text{size}(\overline{x}) \geq 2^{\text{rank}(\overline{x})}$. (This is easy to prve by induction hint hint.)
- In particular, since there are only n objects, the highest possible rank is $\lfloor \lg n \rfloor$.

We can also derive a bound on the number of nodes with a given rank r. Only set leaders can change their rank. When the rank of a set leader $\overline{x}$ changes from $r-1$ to r, mark all the nodes in that set. At least 2^r nodes are marked. The next time these nodes get a new leader $\overline{y}$, the rank of $\overline{y}$ will be at least $r+1$. Thus, any node is marked at most once. There are n nodes altogether, and any object with rank r marks 2^r of them. Thus, there can be $\boxed{\text{at most } n/2^r \text{ objects of rank } r}$.

Purely as an accounting tool, we will also partition the objects into several numbered *blocks*. Specifically, each object x is assigned to block number $\lg^*(\text{rank}(x))$. In other words, x is in block b if and only if

$$2 \uparrow\uparrow (b-1) < \text{rank}(x) \le 2 \uparrow\uparrow b,$$

where $2 \uparrow\uparrow b$ is the *tower* function[1]

$$2 \uparrow\uparrow b = \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right\}^{b} = \begin{cases} 1 & \text{if } b = 0 \\ 2^{2\uparrow\uparrow(b-1)} & \text{if } b > 0 \end{cases}$$

Since there are at most $n/2^r$ objects with any rank r, the total number of objects in block b is at most

$$\sum_{r=2\uparrow\uparrow(b-1)+1}^{2\uparrow\uparrow b} \frac{n}{2^r} < \sum_{r=2\uparrow\uparrow(b-1)+1}^{\infty} \frac{n}{2^r} = \frac{n}{2^{2\uparrow\uparrow(b-1)}} = \boxed{\frac{n}{2 \uparrow\uparrow b}}.$$

Every object has a rank between 0 and $\lfloor \lg n \rfloor$, so $\boxed{\text{there are } \lg^* n \text{ blocks}}$, numbered from 0 to $\lg^* \lfloor \lg n \rfloor = \lg^* n - 1$.

Theorem 2. *If we use both union-by-rank and path compression, the worst-case running time of a sequence of m operations, n of which are* MakeSet *operations, is $O(m \log^* n)$.*

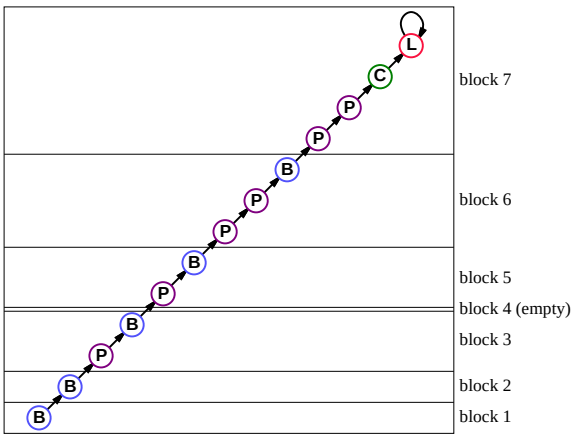
Proof: Since each MakeSet and Union operation takes constant time, it suffices to show that any sequence of m Find operations requires $O(m \log^* n)$ time in the worst case.

The cost of Find(x_0) is proportional to the number of nodes on the *find path* from x_0 up to its leader (before path compression). To count up the total cost of all Finds, we use an accounting method—each object $x_0, x_1, x_2, \ldots, x_l$ on the find path pays a \$1 tax into one of several different bank accounts. After all the Find operations are done, the total amount of money in these accounts will tell us the total running time.

- The leader x_l pays into the *leader* account.
- The child of the leader x_{l-1} pays into the *child* account.
- Any other object x_i in a different block from its parent x_{i+1} pays into the *block* account.
- Any other object x_i in the same block as its parent x_{i+1} pays into the *path* account.

During any Find operation, one dollar is paid into the leader account, at most one dollar is paid into the child account, and at most one dollar is paid into the block account for each of the $\lg^* n$ blocks. Thus, when the sequence of m operations ends, those three accounts share a total of at most $2m + m \lg^* n$ dollars. The only remaining difficulty is the path account.

[1]The arrow notation $a \uparrow\uparrow b$ was introduced by Don Knuth in 1976.



Different nodes on the find path pay into different accounts: B=block, P=path, C=child, L=leader. Horizontal lines are boundaries between blocks. Only the nodes on the find path are shown.

So consider an object x_i in block b that pays into the path account. This object is not a set leader, so its rank can never change. The parent of x_i is also not a set leader, so after path compression, x_i acquires a new parent—namely x_l—whose rank is strictly larger than its old parent x_{i+1}. Since rank(parent(x)) is always increasing, the parent of x_i must eventually lie in a different block than x_i, after which x_i will never pay into the path account. Thus, x_i can pay into the path account at most once for every rank in block b, or less than $2 \uparrow\uparrow b$ times overall.

Since block b contains less than $n/(2 \uparrow\uparrow b)$ objects, these objects contribute less than n dollars to the path account. There are $\lg^* n$ blocks, so the path account receives less than $n \lg^* n$ dollars altogether.

Thus, the total amount of money in all four accounts is less than $2m + m \lg^* n + n \lg^* n = O(m \lg^* n)$, and this bounds the total running time of the m FIND operations. □

The aggregate method now implies that each FIND has $\boxed{\text{amortized cost } O(\log^* n)}$, which is significantly better than its $\boxed{\text{worst-case cost } \Theta(\log n)}$.

9.4 Ackermann's Function and Its Inverse

But this amortized time bound can be improved even more! Just to *state* the correct time bound, I need to introduce a certain function defined by Wilhelm Ackermann in 1928. The function can be[2] defined by the following two-parameter recurrence.

$$\boxed{A_i(n) = \begin{cases} 2 & \text{if } n = 1 \\ 2j & \text{if } i = 1 \text{ and } n > 1 \\ A_{i-1}(A_i(n-1)) & \text{otherwise} \end{cases}}$$

Clearly, each $A_i(n)$ is a monotonically increasing function of n, and these functions grow faster and faster as the index i increases—$A_2(n)$ is the power function 2^n, $A_3(n)$ is the tower function $2 \uparrow\uparrow n$, $A_4(n)$ is the *wower* function $2 \uparrow\uparrow\uparrow n = \underbrace{2 \uparrow\uparrow 2 \uparrow\uparrow \cdots \uparrow\uparrow 2}_{n}$ (so named by John Conway), *et cetera ad infinitum*.

[2]Ackermann didn't define his function this way—I'm actually describing a different function defined 35 years later by R. Creighton Buck—but the exact details of the definition are surprisingly irrelevant!

i	$A_i(n)$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$i=1$	$2n$	2	4	6	8	10
$i=2$	$2{\uparrow}n$	2	4	8	16	32
$i=3$	$2 \uparrow\uparrow n$	2	4	16	65536	2^{65536}
$i=4$	$2 \uparrow\uparrow\uparrow n$	2	4	65536	$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\Big\}65536$	$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\Big\}2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\Big\}65536$
$i=5$	$2 \uparrow\uparrow\uparrow\uparrow n$	2	4	$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\Big\}65536$	$2^{2^{\cdots 2}}\Big\}^{2^{\cdots 2}}\Big\}^{\cdot^{\cdot^{\cdot^{2^{\cdots 2}}}}}\Big\}65536\Big\}2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\Big\}65536$	⟪*Yeah, right.*⟫

Small(!!) values of Ackermann's function.

The *functional inverse* of Ackermann's function is defined as follows:

$$\boxed{\alpha(m,n) = \min\{i \mid A_i(\lfloor m/n \rfloor) > \lg n\}}$$

For all practical values of n and m, we have $\alpha(m,n) \leq 4$; nevertheless, if we increase m and keep n fixed, $\alpha(m,n)$ is eventually bigger than any fixed constant.

Bob Tarjan proved the following surprising theorem. The proof of the upper bound[3] is very similar to the proof of Theorem 2, except that it uses a more complicated 'block' structure. The proof of the matching lower bound[4] is, unfortunately, way beyond the scope of this class.[5]

Theorem 3. *Using both union by rank and path compression, the worst-case running time of a sequence of m operations, n of which are* MakeSets*, is $\Theta(m\alpha(m,n))$. Thus, each operation has amortized cost $\Theta(\alpha(m,n))$. This time bound is optimal: any pointer-based data structure needs $\Omega(m\alpha(m,n))$ time to perform these operations.*

[3] R. E. Tarjan. Efficiency of agood but not linear set union algorithm. *J. Assoc. Comput. Mach.* 22:215–225, 1975.

[4] R. E. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *J. Comput. Syst. Sci.* 19:110–127, 1979.

[5] But if you like this sort of thing, google for "Davenport-Schinzel sequences".