Disjoint Sets ADT

A **disjoint sets** data structure keeps track of multiple sets which do not share any elements. Here's the ADT:

UnionFind ADT

find(**x**)	Returns a number representing the set that **x** is in.
union(**x**, **y**)	Updates the sets so whatever sets **x** and **y** were in are now considered the same sets.

Example

```
list = [1, 2, 3, 4, 5, 6];
UF uf = new UF(list); // State: {1}, {2}, {3}, {4}, {5}, {6}
uf.find(1);           // Returns 1
uf.find(2);           // Returns 2
uf.union(1, 2);       // State: {1, 2}, {3}, {4}, {5}, {6}
uf.find(1);           // Returns 1
uf.find(2);           // Returns 1
uf.union(3, 5);       // State: {1, 2}, {3, 5}, {4}, {6}
uf.union(1, 3);       // State: {1, 2, 3, 5}, {4}, {6}
uf.find(3);           // Returns 1
uf.find(6);           // Returns 6
```

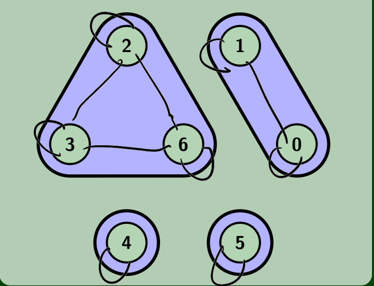
Implementation 1: A List of LinkedLists

Data Structure

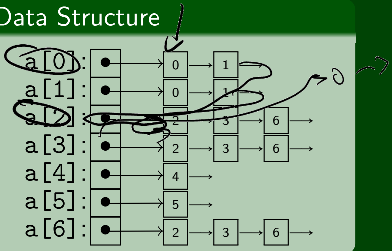
Type: `List<LinkedList<Integer>>`

Idea: A mapping from **id** → a list of **id**s in the same set

Pictorial View



Data Structure



`find(x)`

```
find(x) {
   return a[x].front;
}
```

`union(x, y)`

```
union(x, y) {
   ...
}
```

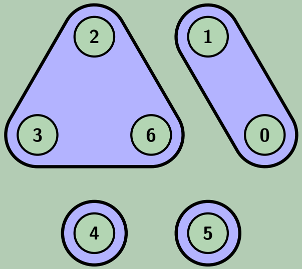
Implementation 1.5: A List of LinkedLists

Data Structure

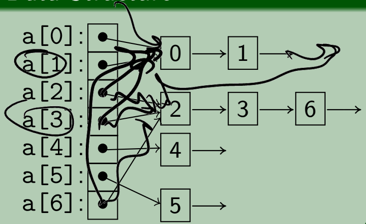
Type: `List<LinkedList<Integer>>`

Idea: A mapping from **id** → a list of **id**s in the same set

Pictorial View



Data Structure



find(x)

```
find(x) {
   return a[x].front;
}
```

union(x, y)

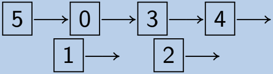
```
union(x, y) {
   curr = a[x].head;
   a[y].tail.next = curr;
   while (curr != null && curr.next != null) {
      a[curr.data] = a[y].head
      curr = curr.next;
   }
}
```

Implementation 3: IMPLICIT Lists Unioned-By-Size

We started with a **list of linked lists**. Then, we realized that we could use **references to the same linked list** to save memory.

We can do even better. The idea is to use an "implicit list".

Example (Explicit List)



Example (Implicit List)

3	-1	-1	4	-1	0
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

5 → 0 → 3 → 4

1 →

2 →

Implementation 3: IMPLICIT Lists Unioned-By-Size

We started with a **list of linked lists**. Then, we realized that we could use **references to the same linked list** to save memory.

We can do even better. The idea is to use an “implicit list”.

Example (Explicit List)

$5 \to 0 \to 3 \to 4 \to$

$1 \to \quad 2 \to$

Example (Implicit List)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
3	-1	-1	4	-1	0

If you’ve already taken CSE 351, you’ve seen this idea already! When implementing `malloc`, you store a **free list**. You can save a lot of memory (which in `malloc` is important...) by using the unused **data fields** to store the **pointers**.

0	1	2	3	4	5	6
5	-2	-3	1	2	-2	-4

$6 \to 4 \to 2$

$3 \to 1$

$0 \to 5$

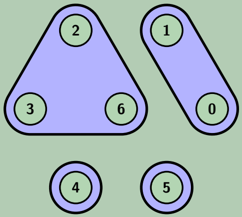
Implementation 3: IMPLICIT Lists Unioned-By-Size

Data Structure

Type: An array

Idea: Each index has either the value of the "next" thing in its set or a negative number representing the size of the set

Pictorial View



Data Structure

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
-2	0	6	~~-2~~ 6	-1	-1	-3

- "Non-canonicals" store "pointers"
- "Canonicals" store -size

Implementation

```
init(x) { a[x] = −1 }
```

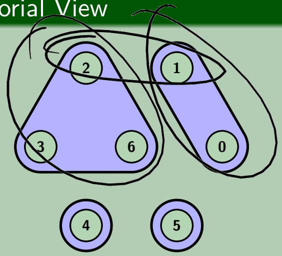
Implementation 3: IMPLICIT Lists Unioned-By-Size

Data Structure

Type: An array

Idea: Each index has either the value of the "next" thing in its set or a negative number representing the size of the set

Pictorial View



Data Structure

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
-5 ~~-2~~	0	6	2	-1	-1	~~3~~ 0

- "Non-canonicals" store "pointers"
- "Canonicals" store -size

Implementation

```
init(x) { a[x] = −1 }
find(x) {
   while(a[x] >= 0) {
      x = a[x]
   }
   return x
}
```

Implementation 4: Implicit Lists Unioned-By-Size with Path Compression

OLD find(x)

```
find(x) {
   while(a[x] >= 0) {
      x = a[x]
   }
   return x
}
```

NEW find(x)

```
find(x) {
   if (a[x] < 0) {
      return x
   }
   a[x] = find(a[x])
   return a[x]
}
```

In Words: Once we've **found** a node. . . save it.

Amortized Analysis of m find Operations?

$f(1) \to 2 \to 3 \to 4$

Consider what we know:

- We know the worst case height of a tree is $\lg(n)$.
- We know it's difficult to make a tree of large height.
- We know that as soon as we access a path in a tree, it flattens the whole path

This **feels** like it should be better than $\lg(n)$, and it is.

We can use facts to show this, but its outside the scope of this lecture. Instead, we'll just talk about two bounds.

Inverse Ackermann function

But it gets better. . .

Upper Bound 2: `find`(x) is amortized $\mathcal{O}(\alpha(n))$

2STACK

The Ackermann function grows even more quickly than $\cancel{\lg^*(n)}$.

It turns out $\alpha(n)$, the **inverse Ackermann function** is also an upper bound. . .

Interestingly, **it is also a lower bound** for the disjoint data structures problem! We can't do better than the algorithm we came up with! (Just like with sorting!)