Today’s Agenda:
✦ Planting and growing a forest of Up-Trees
rut
✦ Union-ing and Find-ing
✦ Extended example
✦ Implementing Union/Find
✦ Smart Union and Find
✦ Union-by-size/height and Path Compression
✦ Run Time Analysis – as tough as it gets!

Covered in Chapter 8 of the textbook

Recall from Last Time: Disjoint Set ADT
✦ Stores N unique elements. Two operations:
❖ Find: Given an element, return the name of its equivalence class (its set)
❖ Union: Given the names of two equivalence classes, merge them into one class

Example:
Initial Classes = \{1,4,8\}, \{2,3\}, \{6\}, \{7\}, \{10,9,5\}
Name of equiv. class underlined

Find(4) 8

Union(3,6)
Up-Tree Data Structure for Disjoint Sets

- Each equivalence class (or set) is an up-tree with its root as its representative member (= class name)
- All members of a given set are nodes in that set’s up-tree
- Hash table maps input data to a node e.g. input string \[ \rightarrow \] integer index

Up-trees are usually **not** binary!

Neat implementation trick for Up-Trees

- Forest of up-trees can easily be stored in an array (call it “up”)
- If node names are integers or characters, can use a very simple, perfect hash function: Hash(X) = X
- up [X] = parent of X;
  \[ = 0 \] if root

Array up:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R. Rao, CSE 373
Example of Find

Find: Just traverse to the root!

Find(f) = c
Find(e) = a

Runtime = ?

Array up:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Change a (from 0) to point to c (= 3)

Example of Union

Union: Just hang one root from the other!

Union(c,a)

Runtime = ?

Now:
Find(f) = c
Find(e) = c
A more detailed example

Initial Sets:

Union(b,e) ↓

A more detailed example...

Union(a,d) ↓
A more detailed example…

Union(a,b)

A more detailed example…

Union(d,e) – But (you say) d and e are not roots!
May be allowed in some implementations – do Find first to get roots
Since Find(d) = Find(e), union already done!

But: while we’re finding e, could we do something to
speed up Find(e) next time? (hold that thought!)
A more detailed example (continued)

Union(h,i)

A more detailed example...

Union(c,f)
A more detailed example

Implementation of Find and Union

```plaintext
int Find(int X, DisjSet up)
{
    // Assumes X = Hash(X_Element)
    // X_Element could be str/char etc.
    if (up[X] <= 0) // Root
        return X; //Return root = set name
    else
        //Find parent
        return Find(up[X], up);
}
void Union(DisjSet up, int X, int Y) {
    //Make sure X, Y are //roots
    assert(up[X] == 0);
    assert(up[Y] == 0);
    up[Y] = X;
}
Runtime of Find: $O(\text{max height})$
Runtime of Union: $O(1)$

Height depends on previous Unions
→ Best case: 1-2, 1-3, 1-4,... $O(1)$
→ Worst case: 2-1, 3-2, 4-3,... $O(N)$
Can we do better?
```
Let’s look back at our example…

Could we do a better job on this Union? What happened to e?

Speeding Up Union/Find: Union-by-Size

✦ For M Finds and N-1 Unions, worst case time is O(MN+N)

✧ Can we speed things up by being clever about growing our up-trees?

✦ Idea: In Union, always make root of larger tree the new root

✦ Why? Minimizes height of the new up-tree
Trick for Storing Size Information

- Instead of storing 0 in root, store up-tree size as **negative value** in root node.
- Why not positive value?
  - Would not know if array entry is size or parent pointer.

![Diagram](https://example.com/diagram.png)

Array up: 

```
0  1 (a)  2 (b)  3 (c)  4 (d)  5 (e)  6 (f)  7 (g)  8 (h)
```

```
-5  1  -2  1  2  3  1  -1
```

Union-by-Size Code

```c
void Union(DisjSet up, int X, int Y)
{
    //X, Y are roots
    //containing (-size) of up-trees
    assert(up[X] < 0);
    assert(up[Y] < 0);

    if (-up[X] > -up[Y]) {
        //update size of X and root of Y
        up[X] += up[Y];
        up[Y] = X;
    }
    else { //size of X < size of Y
        up[Y] += up[X];
        up[X] = Y;
    }
}
```

New run time of Union = ?

New run time of Find = ?
Union-by-Size: Analysis

- Finds are $O(\text{max up-tree height})$ for a forest of up-trees containing $N$ nodes
- Number of nodes in an up-tree of height $h$ using union-by-size is $\geq 2^h$
- Pick up-tree with max height
- Then, $2^{\text{max height}} \leq N$
- max height $\leq \log N$
- Find takes $O(\log N)$

Union-by-Height

- Textbook describes alternative strategy of Union-by-height
- Keep track of height of each up-tree in the root nodes
- Union makes root of up-tree with greater height the new root
- Same results and similar implementation as Union-by-Size
  - Find is $O(\log N)$ and Union is $O(1)$
Speeding Up Find: Path Compression

- If we do $M$ Finds on same element $\Rightarrow O(M \log N)$ time
  - Can we modify Find to have side-effects so that next Find will be faster?
- Path Compression: Point everything along path of a Find to root
- Reduces height of entire access path to 1: Finds get faster!
  - Déjà vu? Idea similar to the one behind your old friend – splay tree…

**A P.C. example with more meat…**

R. Rao, CSE 373

21

R. Rao, CSE 373

22
R. Rao, CSE 373

Trivial modification of original Find
New running time of Find = ?

int Find(int X, DisjSet up)
{
    // Assumes X = Hash(X_Element)
    // X_Element could be str/char etc.

    if (up[X] <= 0) // Root
        return X; //Return root = set name
    else
        //Find parent
        return up[X] = Find(up[X], up);
}

• Trivial modification of original Find
• New running time of Find = ?

Make all nodes along access path point to root

R. Rao, CSE 373

Find still takes O(max up-tree height) worst case
But what happens to the tree heights over time?
What is the amortized run time of Find if we do M Finds?
Analysis of P.C. with Union-by-Size

- R. E. Tarjan (of the up-trees fame) showed that:
  - When both P.C. and Union-by-Size are used, the worst case run time for a sequence of $M$ operations (Unions or Finds) is $\Theta(M \alpha(M,N))$
- What is $\alpha(M,N)$?
  - $\alpha(M,N)$ is the inverse of Ackermann’s function
- What is Ackermann’s function?

Digression: Them slow-growing functions…

- **How fast does log $N$ grow?** log $N = 4$ for $N = 16 = 2^4$
  - Grows quite slowly
- **Let** $\log^{(k)} N = \log (\log (\log \ldots (\log N)))$ (k logs)
- **Let** $\log^* N = \text{minimum } k \text{ such that } \log^{(k)} N \leq 1$
- **How fast does log$^*$ $N$ grow?** log$^* N = 4$ for $N = 65536 = 2^{22}$
  - Grows very slowly
- Ackermann created a really explosive function $A(i, j)$ whose inverse $\alpha(M, N)$ grows very, very slowly (slower than log$^* N$)
- **How slow does $\alpha(M, N)$ grow?** $\alpha(M, N) = 4$ for $M \geq N$ far larger than the number of atoms in the universe ($2^{300}$)!!
Analysis of P.C. with Union-by-Size

- R. E. Tarjan (of the up-trees fame) showed that:
  - When both P.C. and Union-by-Size are used, the worst case run time for a sequence of M operations (Unions or Finds) is $\Theta(M \alpha(M, N))$
  - $\alpha(M, N) \leq 4$ for all practical choices of M and N

- Textbook proves weaker result of $O(M \log^* N)$ time
  - 7 pages and 8 Lemmas! (Check it out but no need to know the proof)

- Amortized run time per operation = total time/(# operations)
  - $= \Theta(M \alpha(M, N))/M = \Theta(\alpha(M, N)) \approx \Theta(1)$ for all practical purposes (constant time!)

Summary of Disjoint Set and Union/Find

- Disjoint Set data structure arises in many applications where objects of interest fall into different equivalence classes or sets
  - Cities on a map, electrical components on chip, computers in a network, people related to each other by blood, etc.

- Two main operations: Union of two classes and Find class name for a given element

- Up-Tree data structure allows efficient array implementation
  - Unions take $O(1)$ worst case time, Finds can take $O(N)$
  - Union-by-Size reduces worst case time for Find to $O(\log N)$
  - Union-by-Size plus Path Compression allows further speedup

  - Any sequence of M Union/Find operations results in $O(1)$ amortized time per operation (for all practical purposes)
Next Class: CSE 373 gets graphic…
(Algo-rhythms on Graphs)

To Do:
Finish Homework #4 (due next class)
Finish reading chapter 8
Start reading chapter 9