Lecture 21: Union and Find between Up-Trees

✦ Today’s Agenda:
  ➤ Planting and growing a forest of Up-Trees
  ➤ 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

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 ➔ integer index
  \{a,d,g,b,e\} \{c,f\} \{h\}
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:
- \begin{array}{cccccccc}
  0 & 1 & 0 & 1 & 2 & 3 & 1 & 0 \\
\end{array}

R. Rao, CSE 373  Some of the material on these slides are courtesy of: S. Wolfman, CSE 326, 2000
Example of Find

Find: Just traverse to the root!

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

Runtime = ?

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…

Union(h,i)
Implementation of Find and Union

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

Runtime of Union: O(1)
Can we do better?

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

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;
  }
}
```

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 $\geq 2^h$
  - Base case: $h = 0$, tree has 2$^0$ = 1 node
  - Induction hypothesis: Assume true for $h < h'$
  - Induction Step: New tree of height $h'$ was formed via union of two trees of height $h' - 1$
    - Each tree then has $\geq 2^{h' - 1}$ nodes by the induction hypothesis
    - So, total nodes $\geq 2^{h' - 1} + 2^{h' - 1} = 2^h$
      - True for all $h$
✦ Pick up-tree with max height
✦ Then, $2^{\text{max height}} \leq N$
✦ max height $\leq \log N$
✦ Find takes $O(\log N)$
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…

```c
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);
}
```

- Make all nodes along access path point to root
- Trivial modification of original Find
- New running time of Find $= ?$

A P.C. example with more meat…

Find(e)  

How to P.C. – Path Compression Code

```c
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);
}
```

- Find still takes $O(\text{max up-tree height})$ 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 \( \log^{(k)} N = \log (\log (\log \ldots (\log N))) \) \( (k \text{ logs}) \)
- Let \( \log^* N \) = minimum \( k \) such that \( \log^{(k)} N \leq 1 \)
- How fast does \( \log^* N \) grow?
  \( \log^* N = 4 \) for \( N = 65536 = 2^{2^2} \)
  - 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 (\( 2^{300} \))!!

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