Announcements (5/21/08)

- Homework due beginning of class on Friday.

- Reading for this lecture: Chapter 8.

CSE 326: Data Structures
Disjoint Set Union/Find
(part 2)

Brian Curless
Spring 2008

Decision Tree

The leaves contain all the possible orderings of a, b, c.

Alternate Explanation of the
Comparison-based Sorting Bound

At each decision point, one child has $\leq \frac{1}{2}$ of the options remaining, the other has $\geq \frac{1}{2}$ remaining.

Worst case: we always end up with $\geq \frac{1}{2}$ remaining.

Best outcome, in the worst case: we always end up with exactly $\frac{1}{2}$ remaining.

Thus, in the worst case, the best we can hope for is halving the space $d$ times (with $d$ comparisons), until we have an answer, i.e., until the space is reduced to size = 1.

The space starts at $N!$ in size, and halving $d$ times means multiplying by $1/2^d$, giving us a lower bound on the worst case:

$$\frac{N!}{2^d} = 1 \implies N! = 2^d \implies d = \log_2(N!) \in \Omega(N \log N)$$
Implementation: Take 1

**Approach:**
- Each set is a doubly-linked list (with pointer to last element).
- Store set name with object.

**Find:** get set name of object
- Worst case complexity?

**Union:** put one list on the end of the other, update set names of objects to be all the same
- Worst case complexity?

Implementation: Take 2

**Approach:**
- Each set is a doubly-linked list (with pointer to last element).
- Front of list is set identifier.

**Find:** traverse linked list until reaching the front
- Worst case complexity?

**Union:** put one list on the end of the other
- Worst case complexity?

Union/Find Trade-off

- Known result:
  - Find and Union cannot both be done in worst-case $O(1)$ time with any data structure.
- We will instead aim for good *amortized* complexity.
- For $m$ operations on $n$ elements:
  - Target complexity: $O(m)$ *i.e.* $O(1)$ amortized

Tree-based Approach

We'll build on the “fast union” approach (linked list, with head node as set name, no set names explicitly stored in nodes).

**Improvements:**
- Instead of linked lists, use a forest of trees (one tree per set).
- Root of each tree is the set name.
- Allow large fanout. Why is this good?
Up-Tree for DS Union/Find

Observation: we will only traverse these trees upward from any given node to find the root.

Idea: reverse the pointers (make them point up from child to parent). The result is an up-tree.

Initial state

1 2 3 4 5 6 7

Intermediate state

1 2 3 4 5 6 7

Roots are the names of each set.

Find Operation

Find(x) follow x to the root and return the root.

Find(6) = 7

Union Operation

Union(i, j) - assuming i and j roots, point i to j.

Simple Implementation

• Array of indices

Up[x] = -1 means x is a root.
Implementation

```c
int Find(int x) {
    while(up[x] >= 0) {
        x = up[x];
    }
    return x;
}

void Union(int x, int y) {
    up[y] = x;
}
```

runtime for Union: runtime for Find:

Amortized complexity is no better.

A Bad Case

![A Bad Case Diagram]

Two Big Improvements

Can we do better?  Yes!

1. Union-by-size
   • Improve \texttt{Union} so that \texttt{Find} only takes worst case time of $\Theta(\log n)$.

2. Path compression
   • Improve \texttt{Find} so that, with Union-by-size, \texttt{Find} takes amortized time of almost $\Theta(1)$

Union-by-Size

Union-by-size
   – Always point the smaller tree to the root of the larger tree

S-Union(1,7)

![Union-by-Size Diagram]
Example Again

Analysis of Union-by-Size

- Theorem: With union-by-size an up-tree of height $h$ has size at least $2^h$.
- Proof by induction
  - Base case: $h = 0$. The up-tree has one node, $2^0 = 1$
  - Inductive hypothesis: Assume true for $h-1$
  - Inductive step: Then true for $h$.
  - Observation: tree gets taller only as a result of a union.

$$T = \text{S-Union}(T_1, T_2)$$

Worst Case for Union-by-Size

- What is worst case complexity of Find(x) in an up-tree forest of $n$ nodes?
- (Amortized complexity is no better.)

$$n/2 \text{ Unions-by-size}$$

$$n/4 \text{ Unions-by-size}$$
Example of Worst Cast (cont’)

After \( n - 1 = n/2 + n/4 + \ldots + 1 \) Unions-by-size

If there are \( n = 2^k \) nodes then the longest path from leaf to root has length \( k \).

Array Implementation

Can store separate size array:

<table>
<thead>
<tr>
<th>up</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Better, store sizes in the up array:

```
1 2 3 4 5 6 7
up 2 1 -1 7 7 5 -4
```

Negative up-values correspond to sizes of roots.

Elegant Array Implementation

Code for Union-by-Size

```
S-Union(i,j){
    // Collect sizes
    si = -up[i];
    sj = -up[j];

    // verify i and j are roots
    assert(si >=0 && sj >=0)

    // point smaller sized tree to
    // root of larger, update size
    if (si < sj) {
        up[i] = j;
        up[j] = -(si + sj);
    } else {
        up[j] = i;
        up[i] = -(si + sj);
    }
}
```
Path Compression

• To improve the amortized complexity, we’ll borrow an idea from splay trees:
  – When going up the tree, *improve nodes on the path*!
• On a Find operation point all the nodes on the search path directly to the root. This is called “path compression.”

Self-Adjustment Works

Your turn

Draw the result of Find(e):

Code for Path Compression Find

```java
PC-Find(i) {
  j = i;

  // find root
  while (up[j] >= 0) {
    j = up[j];
    root = j;
  }

  // compress path
  if (i != root) {
    parent = up[i];
    while (parent != root) {
      up[i] = root;
      i = parent;
      parent = up[parent];
    }
  }
  return(root)
}
```
Complexity of Union-by-Size + Path Compression

- Worst case time complexity for...
  - ...a single Union-by-size is:
  - ...a single PC-Find is:

- Time complexity for \( m \geq n \) operations on \( n \) elements has been shown to be \( O(m \log^* n) \).
  [See Weiss for proof.]
  - Amortized complexity is then \( O(\log^* n) \)
  - What is \( \log^* \)?

\[
\log^* n \leq 5 \text{ for all reasonable } n.
\]

The Tight Bound

In fact, Tarjan showed the time complexity for \( m \geq n \) operations on \( n \) elements is:

\[
\Theta(m \alpha(m, n))
\]

Amortized complexity is then \( \Theta(\alpha(m, n)) \).

What is \( \alpha(m, n) \)?
- Inverse of Ackermann’s function.
- For reasonable values of \( m, n \), grows even slower than \( \log^* n \). So, it’s even “more constant.”

Proof is beyond scope of this class. A simple algorithm can lead to incredibly hardcore analysis!