Brief Midterm Postmortem

• Heaps
• Hash tables
• Bubble sort
• Properties of Sorting Algorithms
• Merging
Analysis of Weighted Union

• With weighted union an up-tree of height $h$ has weight at least $2^h$.

• Proof by induction
  – Basis: $h = 0$. The up-tree has one node, $2^0 = 1$
  – Inductive step: Assume true for all $h' < h$.

Minimum weight up-tree of height $h$ formed by weighted unions

- $W(T_1) \geq W(T_2) \geq 2^{h-1}$
- Weighted union
- Induction hypothesis

$W(T) \geq 2^{h-1} + 2^{h-1} = 2^h$
Analysis of Weighted Union

• Let T be an up-tree of weight n formed by weighted union. Let h be its height.

  • \( n \geq 2^h \)
  • \( \log_2 n \geq h \)
  • \( \text{Find}(x) \text{ in tree } T \) takes \( O(\log n) \) time.
  • Can we do better?
Worst Case for Weighted Union

n/2 Weighted Unions

n/4 Weighted Unions
Example of Worst Cast (cont’)

After \( n - 1 = \frac{n}{2} + \frac{n}{4} + \ldots + 1 \) Weighted Unions

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

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

```
<table>
<thead>
<tr>
<th>up weight</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>7</th>
<th>7</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
```
Weighted Union

\[
W-\text{Union}(i,j : \text{index})\
//i \text{ and } j \text{ are roots}\\
\text{wi} := \text{weight}[i];\\
\text{wj} := \text{weight}[j];\\
\text{if } \text{wi} < \text{wj} \text{ then}\\
\quad \text{up}[i] := j;\\
\quad \text{weight}[j] := \text{wi} + \text{wj};\\
\text{else}\\
\quad \text{up}[j] := i;\\
\quad \text{weight}[i] := \text{wi} + \text{wj};\\
\]
Path Compression

• On a Find operation point all the nodes on the search path directly to the root.
Self-Adjustment Works

PC-Find(x)
Student Activity

Draw the result of Find(e):

```
  a  \
 /   \
|     |
b     d
|     |
i     e
|     |
h     c
     |
g
```

11/18/2009 CSE 373 Fall 2009 -- Dan Suciu
Path Compression Find

PC-Find(i : index) {
    r := i;
    while up[r] ≠ 0 do //find root//
        r := up[r];
    if i ≠ r then  //compress path//
        k := up[i];
        while k ≠ r do
            up[i] := r;
            i := k;
            k := up[k]
    return(r)
}

Interlude: A Really Slow Function

**Ackermann’s function** is a really big function \( A(x, y) \) with inverse \( \alpha(x, y) \) which is really small

How fast does \( \alpha(x, y) \) grow?

\[
\alpha(x, y) = 4 \text{ for } x \text{ far larger than the number of atoms in the universe } (2^{300})
\]

\( \alpha \) shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences
A More Comprehensible Slow Function

\[ \log^* x = \text{number of times you need to compute } \log \text{ to bring value down to at most 1} \]

E.g. \( \log^* 2 = 1 \)
\[ \begin{align*}
\log^* 4 &= \log^* 2^2 = 2 \\
\log^* 16 &= \log^* 2^{2^2} = 3 \\
\log^* 65536 &= \log^* 2^{2^{2^2}} = 4 \\
\log^* 2^{65536} &= \ldots \ldots \ldots = 5
\end{align*} \]

Take this: \( \alpha(m,n) \) grows even slower than \( \log^* n \) !!
Disjoint Union / Find with Weighted Union and PC

• Worst case time complexity for a W-Union is $O(1)$ and for a PC-Find is $O(\log n)$.

• Time complexity for $m \geq n$ operations on $n$ elements is $O(m \log^* n)$
  – $\log^* n < 7$ for all reasonable $n$. Essentially constant time per operation!

• Using “ranked union” gives an even better bound theoretically.
Amortized Complexity

• For disjoint union / find with weighted union and path compression.
  – average time per operation is essentially a constant.
  – worst case time for a PC-Find is $O(\log n)$.

• An individual operation can be costly, but over time the average cost per operation is not.
Find Solutions

Recursive

Find(up[] : integer array, x : integer) : integer {
  //precondition: x is in the range 1 to size/
  if up[x] = 0 then return x
  else return Find(up,up[x]);
}

Iterative

Find(up[] : integer array, x : integer) : integer {
  //precondition: x is in the range 1 to size/
  while up[x] ≠ 0 do
    x := up[x];
  return x;
}