CSE 326: Data Structures

Disjoint Union/Find
plus
 Sorting Introduction

James Fogarty
Spring 2009
Disjoint Set ADT

- Data: set of pairwise **disjoint sets**.
- Required operations
  - **Union** – merge two sets to create their union
  - **Find** – determine which set an item appears in

- A common operation sequence:
  - Connect two elements if not already connected:
    
    if (Find(x) != Find(y)) then Union(x,y)
Union

• Union(x,y)
  take the union of two sets named x and y
  – \{3,5,7\} , \{4,2,8\}, \{9\}, \{1,6\}
  – Union(5,1)
    \{3,5,7,1,6\}, \{4,2,8\}, \{9\}
  – Union(9, 8)
    \{3,5,7,1,6\}, \{4,2,8,9\}
Find

- Find(x):
  return the name of the set containing x

- \{3,5,7,1,6\}, \{4,2,8\}, \{9\},

- Find(1) = 5

- Find(4) = 8

- Find(9) = 9
A Bad Case

Find(1): n steps!!
Weighted Union

- Instead of arbitrarily joining two roots, always point the smaller root to the larger root
Elegant Array Implementation

```
    1
   / \   \
  2   2
   \   /
    3

4
   / \   / \   /
  5   4  7   6
```

up weight

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


Weighted Union

//i and j are roots
W-Union(i,j : index){
    wi := weight[i];
    wj := weight[j];
    if wi < wj then
        up[i] := j;
        weight[j] := wi + wj;
    else
        up[j] := i;
        weight[i] := wi + wj;
}
Instead of a separate weight array, can re-use the empty parent reference
Example Again

1  2  3  …  n

Union(1,2)

2  3  …  n

Union(2,3)

1  

Union(n-1,n)

1  3  …  n

Find(1)  constant time
What’s the New Worst Case?

- When able to apply its criterion, weighted union does well

- For worst case, force arbitrary choices
Worst Case for Weighted Union

\[ \frac{n}{2} \text{ Weighted Unions} \]

\[ \frac{n}{4} \text{ Weighted Unions} \]
Example of Worst Cast (cont’)

After $n - 1 = n/2 + n/4 + \ldots + 1$ Weighted Unions

If there are $n = 2^k$ nodes then the worst case longest path from leaf to root has length $k$. 
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 hypothesis: Assume true for all $h' < h$.

\[
W(T_1) \geq W(T_2) > 2^{h-1}
\]

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

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

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

- $n \geq 2^h$, therefore $\log_2 n \geq h$

- Find($x$) in tree $T$ takes $O(\log n)$ time.

- Can we do better?
Path Compression

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

PC-Find(x)
Draw the result of Find(e):
Draw the result of Find(e):
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)
}
A Very Slow Growing Function

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

E.g. $log^* 2 = 1$
$log^* 4 = log^* 2^2 = 2$
$log^* 16 = log^* 2^{2^2} = 3$ (log log log 16 = 1)
$log^* 65536 = log^* 2^{2^{2^2}} = 4$ (log log log log 65536 = 1)
$log^* 2^{65536} = \ldots \ldots = 5$ (log log log log log 2^{65536} = 1)
Disjoint Union / Find with Weighted Union and PC

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

- Time complexity for $m \geq n$ operations on $n$ elements is $O(m \log^* n)$ (it’s actually better than $\log^* n$)
  - $\log^* n < 7$ for all reasonable $n$.
    Essentially constant time per operation
Course Overview

• We’ve seen a collection of ADTs
  – Stack
  – Queue
  – List
  – Priority Queue
  – Dictionary
  – Disjoint Sets
Course Overview

• We’ve seen a variety of data structures that take different approaches to those.

• This course is about both:
  – Commonly used abstractions
  – Commonalities in strategies for effectively implementing those abstractions
Course Overview

• We now shift away from ADTs and cover two topics in some depth:
  – Sorting
  – Graphs and Associated Algorithms

• The general theme remains:
  – These particular problems are important
  – The strategies we use to attack them are too
Sorting

• Input
  – an array $A$ of data records
  – a key value in each data record
  – a comparison function which imposes a consistent ordering on the keys

• Output
  – reorganize the elements of $A$ such that
    • For any $i$ and $j$, if $i < j$ then $A[i] \leq A[j]$
Consistent Ordering

- The comparison function must provide a consistent *ordering* on the set of possible keys
  - You can compare any two keys and get back an indication of \( a < b \), \( a > b \), or \( a = b \)

- The comparison functions must be consistent
  - If \( \text{compare}(a, b) \) says \( a < b \), then \( \text{compare}(b, a) \) must say \( b > a \)
  - If \( \text{compare}(a, b) \) says \( a = b \), then \( \text{compare}(b, a) \) must say \( b = a \)
Why Sort?

• Allows binary search of an N-element array in $O(\log N)$ time
• Allows $O(1)$ time access to $k$th largest element in the array for any $k$
• People tend to like their output sorted

• Sorting algorithms are a frequently used and heavily studied family of algorithms in computer science
Space

• How much space does an algorithm require to sort a collection of items?
  – Is copying needed?
    • In-place sorting algorithms: no copying or $O(1)$ additional temp space
  – External memory sorting:
    • Data so large it cannot fit in memory
Stability

A sorting algorithm is **stable** if:

– Items in the input with the same value end up in the same order as when they began.

<table>
<thead>
<tr>
<th>Input</th>
<th>Stable or Unstable Sort?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams 1</td>
<td>Adams 1</td>
</tr>
<tr>
<td>Black 2</td>
<td>Smith 1</td>
</tr>
<tr>
<td>Brown 4</td>
<td>Black 2</td>
</tr>
<tr>
<td>Jackson 2</td>
<td>Jackson 2</td>
</tr>
<tr>
<td>Jones 4</td>
<td>Washington 2</td>
</tr>
<tr>
<td>Smith 1</td>
<td>White 3</td>
</tr>
<tr>
<td>Thompson 4</td>
<td>Wilson 3</td>
</tr>
<tr>
<td>Washington 2</td>
<td>Brown 4</td>
</tr>
<tr>
<td>White 3</td>
<td>Jones 4</td>
</tr>
<tr>
<td>Wilson 3</td>
<td>Thompson 4</td>
</tr>
</tbody>
</table>

[Sedgewick]
Stability

A sorting algorithm is **stable** if:

– Items in the input with the same value end up in the same order as when they began.

<table>
<thead>
<tr>
<th>Input</th>
<th>Stable Sort</th>
<th>Unstable sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams 1</td>
<td>Adams 1</td>
<td>Adams 1</td>
</tr>
<tr>
<td>Black 2</td>
<td>Smith 1</td>
<td>Smith 1</td>
</tr>
<tr>
<td>Brown 4</td>
<td>Black 2</td>
<td>Washington 2</td>
</tr>
<tr>
<td>Jackson 2</td>
<td>Jackson 2</td>
<td>Jackson 2</td>
</tr>
<tr>
<td>Jones 4</td>
<td>Washington 2</td>
<td>Black 2</td>
</tr>
<tr>
<td>Smith 1</td>
<td>White 3</td>
<td>White 3</td>
</tr>
<tr>
<td>Thompson 4</td>
<td>Wilson 3</td>
<td>Wilson 3</td>
</tr>
<tr>
<td>Washington 2</td>
<td>Brown 4</td>
<td>Thompson 4</td>
</tr>
<tr>
<td>White 3</td>
<td>Jones 4</td>
<td>Brown 4</td>
</tr>
<tr>
<td>Wilson 3</td>
<td>Thompson 4</td>
<td>Jones 4</td>
</tr>
</tbody>
</table>

[Sedgewick]
How fast is the algorithm?

- The definition of a sorted array $A$ says:
  for any $i < j$, $A[i] \leq A[j]$

- You at least check each element:
  • Complexity is at least: $O(n)$

- You could end up checking each element against every other element
  • Complexity could be as bad as: $O(n^2)$

The big question is: How close to $O(n)$ can you get?
Sorting: *The Big Picture*

Given *n* comparable elements in an array, sort them in an increasing order.

Simple algorithms: $O(n^2)$
- Insertion sort
- Selection sort
- Bubble sort

Improved algorithms: $O(n^{1+q})$
- Shell sort

Fancier algorithms: $O(n \log n)$
- Heap sort
- Binary tree sort
- Merge sort
- Quick sort (avg case)

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: $O(n)$
- Bucket sort
- Radix sort

Handling huge data sets
- External sorting
Selection Sort: Idea

1. Find the smallest element, put it $1^{st}$
2. Find the next smallest element, put it $2^{nd}$
3. Find the next smallest, put it $3^{rd}$
4. And so on …
Try it out: Selection Sort

• 31, 16, 54, 4, 2, 17, 6
• 2, 16, 54, 4, 31, 17, 6
• 2, 4, 54, 16, 31, 17, 6
• 2, 4, 6, 16, 31, 17, 54
• 2, 4, 6, 16, 17, 31, 54
• 2, 4, 6, 16, 17, 31, 54
• 2, 4, 6, 16, 17, 31, 54
• 2, 4, 6, 16, 17, 31, 54

Not shown: lots of scanning over the remainder of the array
Selection Sort: Code

```c
void SelectionSort (Array a[0..n-1]) {
    for (i=0; i<n; ++i) {
        j = Find index of smallest entry in a[i..n-1]
        Swap(a[i],a[j])
    }
}
```

**Runtime:**

- worst case : $O(n^2)$
- best case : $O(n^2)$
- average case : $O(n^2)$
Bubble Sort Idea

• Take a pass through the array
  – If a pair of neighboring elements are out of sort order, swap them.

• Take passes until no swaps are needed at any point in the pass.
Try it out: Bubble Sort

- 31, 16, 54, 4, 2, 17, 6
- 16, 31, 54, 4, 2, 17, 6
- 16, 31, 4, 54, 2, 17, 6
- 16, 31, 4, 2, 54, 17, 6
- 16, 31, 4, 2, 17, 54, 6
- 16, 31, 4, 2, 17, 6, 54
- 16, 4, 31, 2, 17, 6, 54
- 16, 4, 2, 31, 17, 6, 54
- 16, 4, 2, 17, 31, 6, 54
- 16, 4, 2, 17, 6, 31, 54
- 4, 16, 2, 17, 6, 31, 54
- 4, 2, 16, 17, 6, 31, 54
- 4, 2, 16, 6, 17, 31, 54
- 2, 4, 16, 6, 17, 31, 54
- 2, 4, 6, 16, 17, 31, 54
- 2, 4, 6, 16, 17, 31, 54
Bubble Sort: Code

```c
void BubbleSort (Array a[0..n-1]) {
    swapPerformed = 1
    while (swapPerformed) {
        swapPerformed = 0
        for (i=0; i<n-1; i++) {
            if (a[i+1] < a[i]) {
                Swap(a[i], a[i+1])
                swapPerformed = 1
            }
        }
    }
}
```

Runtime:
- worst case : $O(n^2)$
- best case : $O(n)$
- average case : $O(n^2)$
Insertion Sort: Idea

1. One element is by definition sorted
2. Sort first 2 elements.
3. Insert 3\textsuperscript{rd} element in order.
   • (First 3 elements are now sorted.)
4. Insert 4\textsuperscript{th} element in order
   • (First 4 elements are now sorted.)
5. And so on…
How to do the insertion?

Suppose my sequence is:

16, 31, 54, 78, 32, 17, 6

And I’ve already sorted up to 78. How to insert 32?
Example

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>18</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>18</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>18</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>18</th>
<th>15</th>
<th>23</th>
<th>16</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>18</th>
<th>15</th>
<th>23</th>
<th>16</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>18</th>
<th>15</th>
<th>23</th>
<th>16</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>18</th>
<th>23</th>
<th>16</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>18</th>
<th>23</th>
<th>16</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>16</th>
<th>18</th>
<th>23</th>
<th>17</th>
<th>14</th>
</tr>
</thead>
</table>

[Diagram of example process]
Example

1 2 3 7 8 9 10 12 15 16 18 17 23 14
1 2 3 7 8 9 10 12 15 16 17 18 23 14
1 2 3 7 8 9 10 12 15 16 17 18 14 23
1 2 3 7 8 9 10 12 15 16 17 14 18 23
1 2 3 7 8 9 10 12 15 16 14 17 18 23
1 2 3 7 8 9 10 12 15 14 16 17 18 23
1 2 3 7 8 9 10 12 14 15 16 17 18 23

43
Insertion Sort: Code

```c
void InsertionSort (Array a[0..n-1]) {
    for (i=1; i<n; i++) {
        for (j=i; j>0; j--) {
            if (a[j] < a[j-1])
                Swap(a[j],a[j-1])
            else
                break
        }
    }
}
```

Note: can instead minimize copying by creating and then moving a “hole”, as with a binary heap.

**Runtime:**

- worst case : $O(n^2)$
- best case : $O(n)$
- average case : $O(n^2)$
Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6
- 16, 31, 54, 4, 2, 17, 6
- 16, 31, 4, 54, 2, 17, 6
- 16, 4, 31, 54, 2, 17, 6
- 4, 16, 31, 54, 2, 17, 6
- 4, 16, 31, 2, 54, 17, 6
- 4, 16, 2, 31, 54, 17, 6
- 4, 2, 16, 31, 54, 17, 6
- 2, 4, 16, 31, 17, 54, 6
- 2, 4, 16, 17, 31, 54, 6
- 2, 4, 16, 17, 31, 6, 54
- 2, 4, 16, 17, 6, 31, 54
- 2, 4, 16, 6, 17, 31, 54
- 2, 4, 6, 16, 17, 31, 54
- 2, 4, 6, 16, 17, 31, 54
Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6
- 16, 31, 54, 4, 2, 17, 6
- 16, 31, 4, 54, 2, 17, 6
- 16, 4, 31, 54, 2, 17, 6
- 4, 16, 31, 54, 2, 17, 6
- 4, 16, 31, 2, 54, 17, 6
- 4, 16, 2, 31, 54, 17, 6
- 4, 2, 16, 31, 54, 17, 6
- 2, 4, 16, 31, 17, 54, 6
- 2, 4, 16, 17, 31, 54, 6
- 2, 4, 16, 17, 31, 6, 54
- 2, 4, 16, 17, 6, 31, 54
- 2, 4, 16, 6, 17, 31, 54
- 2, 4, 6, 16, 17, 31, 54
- 2, 4, 6, 16, 17, 31, 54

Cut out these assignments by creating and moving the “hole”
Shell Sort: Idea

A small element at end of list takes a long time to percolate to front.

Idea: take bigger steps at first to percolate faster.

1. Choose offset $k$:
   a. Insertion sort over array: $a[0]$, $a[k]$, $a[2k]$, $a[3k]$, …
   b. Insertion sort over array: $a[1]$, $a[1+k]$, $a[1+2k]$, $a[1+3k]$, …
   d. Do this until all elements touched

2. Choose smaller offset $m$ less than $k$, and do another set of insertion sort passes, stepping by $m$ through the array.

3. Repeat for smaller offsets until last pass uses offset = 1

[Named after the algorithm’s inventor, Donald Shell.]
Try it out: Shell Sort

• Offset: 3
  • 31, 16, 54, 4, 2, 17, 6
• 4, 16, 54, 31, 2, 17, 6
• 4, 16, 54, 6, 2, 17, 31
• 4, 2, 54, 6, 16, 17, 31
• 4, 2, 17, 6, 16, 54, 31
• Offset: 2
  • 4, 2, 16, 6, 17, 54, 31

• Offset: 1
  • 2, 4, 16, 6, 17, 54, 31
• 2, 4, 6, 16, 17, 54, 31
• 2, 4, 6, 16, 17, 31, 54
• 2, 4, 6, 16, 17, 31, 54
Shell Sort: Code

```c
void ShellSort (Array a[0..n-1]) {
    determine good offsets based on n
    for (i=0, i<numOffsets; i++) {
        for (j=0, j<offsets[i]; j++) {
            insertionSkipSort(a, j, offsets[i])
        }
    }
}

void InsertionSkipSort (Array a[0..n-1],
                        Int start, Int offset) {
    Do insertion sort on array
    a[start], a[start+offset], a[start+2*offset],...
}
```
Shell Sort Offsets

The key to good Shell sort performance: **good offsets**.

Shell started the offset at \(\text{ceil}(n/2)\) and halved the offset each time. Turns out to be not so good.

Sedgewick proposed this offset sequence:
- Lowest offset is 1.
- Others are: \(1 + 3 \cdot 2^i + 4^{i+1}\) for \(i \geq 0\)
- Looks like: 1, 8, 23, 77, 281, 1073, 4193, ...
- (Put in the offset array in reverse order to work with pseudocode on previous slide.)

Result:
- Worst case complexity is \(O(n^{4/3})\)
- Average case is believed to be \(O(n^{7/6})\)
Heap Sort: Sort with a Binary Heap

Runtime: $O(n \log n)$
Heap Sort: Sort with a Binary Heap

Use a max-heap, do it in-place

Runtime: $O(n \log n)$
<table>
<thead>
<tr>
<th>Step</th>
<th>Array</th>
<th>Array</th>
<th>Array</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31, 16, 54, 4, 2, 17, 6</td>
<td>2, 16, 6, 4, 17, 31, 54</td>
<td>16, 2, 6, 4, 17, 31, 54</td>
<td>16, 4, 6, 2, 17, 31, 54</td>
</tr>
<tr>
<td>2</td>
<td>BuildHeap</td>
<td>54, 16, 31, 4, 2, 17, 6</td>
<td>16, 4, 6, 2, 17, 31, 54</td>
<td>2, 4, 6, 16, 17, 31, 54</td>
</tr>
<tr>
<td>3</td>
<td>6, 16, 31, 4, 2, 17, 54</td>
<td>6, 16, 31, 4, 2, 17, 54</td>
<td>6, 4, 2, 16, 17, 31, 54</td>
<td>2, 4, 6, 16, 17, 31, 54</td>
</tr>
<tr>
<td>4</td>
<td>31, 16, 6, 4, 2, 17, 54</td>
<td>31, 16, 6, 4, 2, 17, 54</td>
<td>31, 16, 17, 4, 2, 6, 54</td>
<td>2, 4, 6, 16, 17, 31, 54</td>
</tr>
<tr>
<td>5</td>
<td>31, 16, 17, 4, 2, 6, 54</td>
<td>31, 16, 17, 4, 2, 6, 54</td>
<td>6, 4, 2, 16, 17, 31, 54</td>
<td>4, 2, 6, 16, 17, 31, 54</td>
</tr>
<tr>
<td>6</td>
<td>6, 16, 17, 4, 2, 31, 54</td>
<td>6, 16, 17, 4, 2, 31, 54</td>
<td>2, 4, 6, 16, 17, 31, 54</td>
<td>2, 4, 6, 16, 17, 31, 54</td>
</tr>
<tr>
<td>7</td>
<td>17, 16, 6, 4, 2, 31, 54</td>
<td>17, 16, 6, 4, 2, 31, 54</td>
<td>2, 4, 6, 16, 17, 31, 54</td>
<td>2, 4, 6, 16, 17, 31, 54</td>
</tr>
</tbody>
</table>

Don’t let all the work near the top of the heap deceive you, this is the only N log N we saw today.