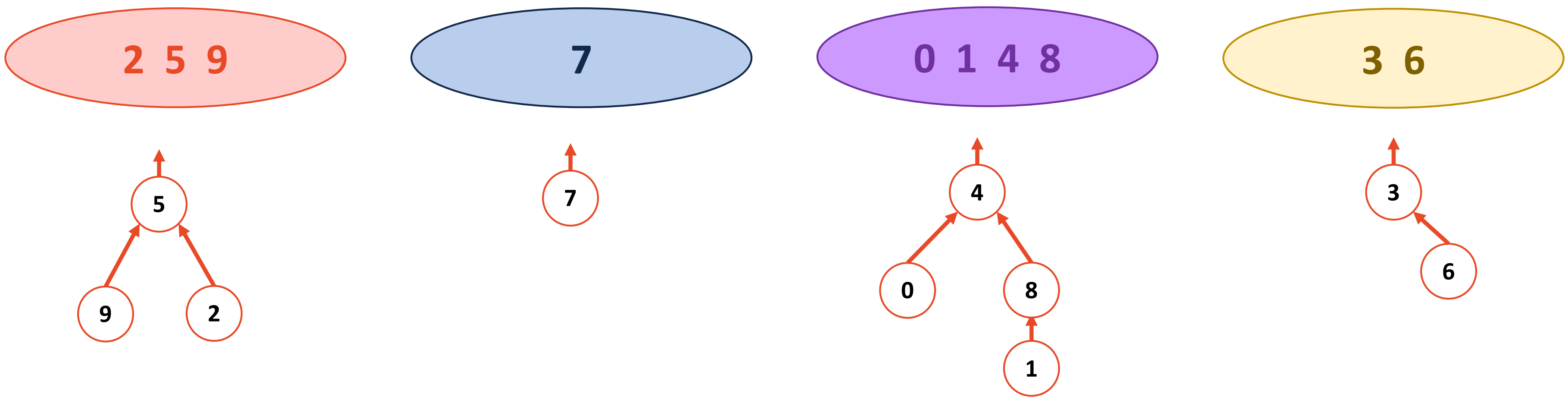
CS 225

Data Structures

November 6 – Disjoint Sets Finale + Graphs

G Carl Evans

Disjoint Sets



0	1	2	3	4	5	6	7	8	9
4	8	5	-1	-1	-1	3	-1	4	5

Disjoint Sets Find

```
int DisjointSets::find() {
  if ( s[i] < 0 ) { return i; }
  else { return _find( s[i] ); }
}
```

Running time?

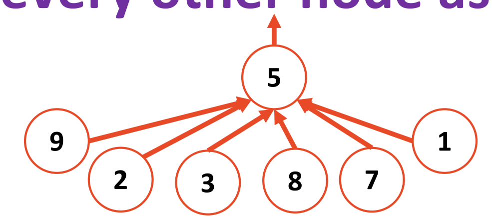
Structure: A structure similar to a linked list

Running time: O(h) < O(n)

What is the ideal UpTree?

Structure: One root node with every other node as it's child

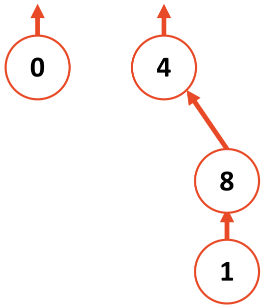
Running Time: O(1)



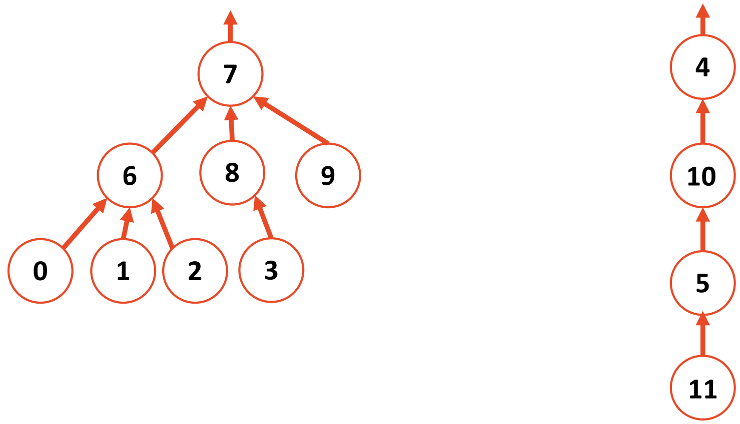
Disjoint Sets Union

```
void DisjointSets::union(int r1, int r2) {


}
```

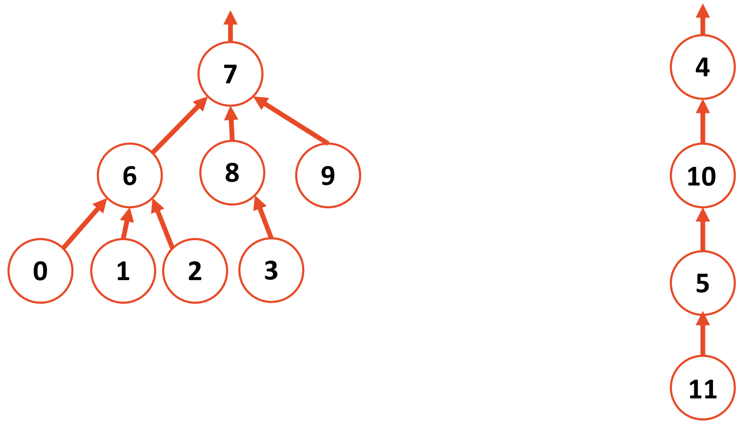


Disjoint Sets – Union



0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-1	10	7	-1	7	7	4	5

Disjoint Sets – Smart Union

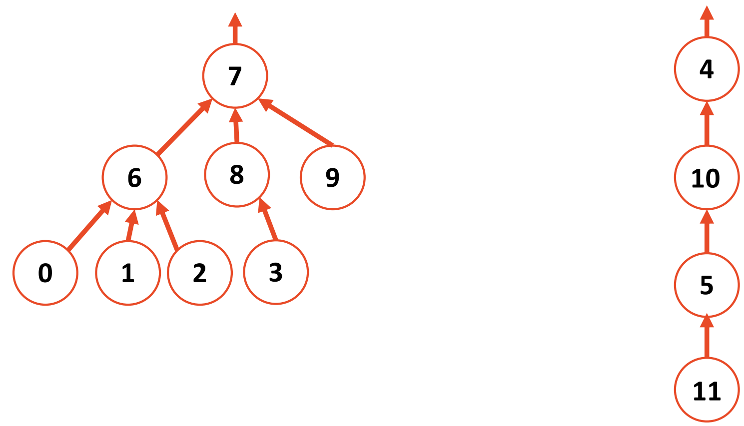


Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8		10	7		7	7	4	5

Idea: *Keep the height of the tree as small as possible.*

Disjoint Sets – Smart Union



Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	**6**	**6**	**8**	**-4**	**10**	**7**	**-3**	**7**	**7**	**4**	**5**

***Idea**: Keep the height of the tree as small as possible.*

Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	**6**	**6**	**8**	**-4**	**10**	**7**	**-8**	**7**	**7**	**4**	**5**

***Idea**: Minimize the number of nodes that increase in height*

Both guarantee the height of the tree is:

Disjoint Sets Find and Union

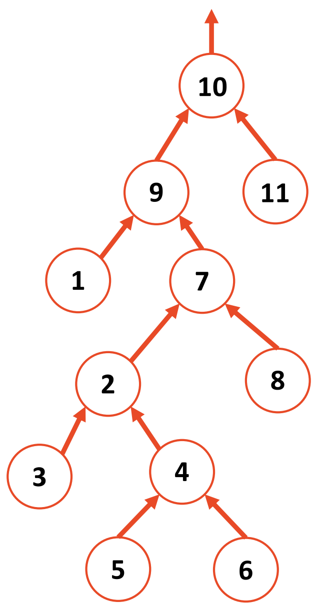
```
int DisjointSets::find(int i) {
  if ( arr_[i] < 0 ) { return i; }
  else { return _find( arr_[i] ); }
}
```

```
void DisjointSets::unionBySize(int root1, int root2) {
  int newSize = arr_[root1] + arr_[root2];

  // If arr_[root1] is less than (more negative), it is the larger set;
  // we union the smaller set, root2, with root1.
  if ( arr_[root1] < arr_[root2] ) {
    arr_[root2] = root1;
    arr_[root1] = newSize;
  }

  // Otherwise, do the opposite:
  else {
    arr_[root1] = root2;
    arr_[root2] = newSize;
  }
}
```

Path Compression



Disjoint Sets Find with Compression

```
int DisjointSets::find(int i) {
  // At root return the index
  if ( arr_[i] < 0 ) {
    return i;
  }

  // If not at the root recurse and on the return update parent
  // to be the root.
  else {
    int root = find( arr_[i] );
    arr_[i] = root;
    return root;
  }
}
```

Disjoint Sets Analysis

The **iterated log** function:

The number of times you can take a log of a number.

log*(n) =

$$\log^*(n) = \begin{cases} 0 & , n \le 1 \\ 1 + \log^*(\log(n)) & , n > 1 \end{cases}$$

What is **lg*(2^{65536})**?

Disjoint Sets Analysis

In an Disjoint Sets implemented with smart **unions** and path compression on **find**:

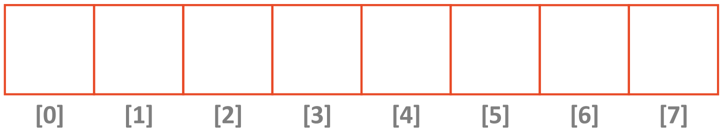
Any sequence of **m** **union** and **find** operations result in the worse case running time of O(____________),

where **n** is the number of items in the Disjoint Sets.

In Review: Data Structures

Array
- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
  - Hashing
  - Heaps
    - Priority Queues
  - UpTrees
    - Disjoint Sets

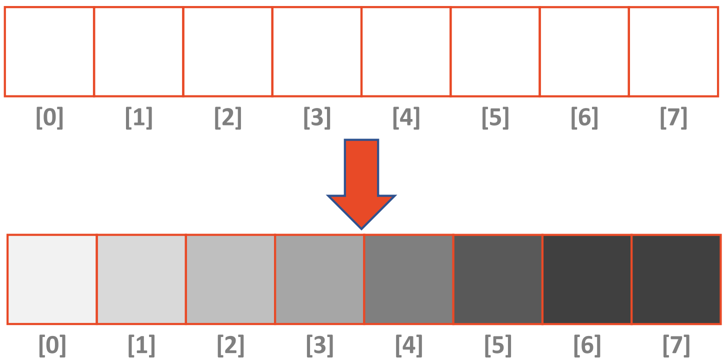
Linked
- Doubly Linked List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree



Array

- Constant time access to any element, given an index
  a[k] is accessed in O(1) time, no matter how large the array grows

- Cache-optimized
  Many modern systems cache or pre-fetch nearby memory values due the “Principle of Locality”. Therefore, arrays often perform faster than lists in identical operations.

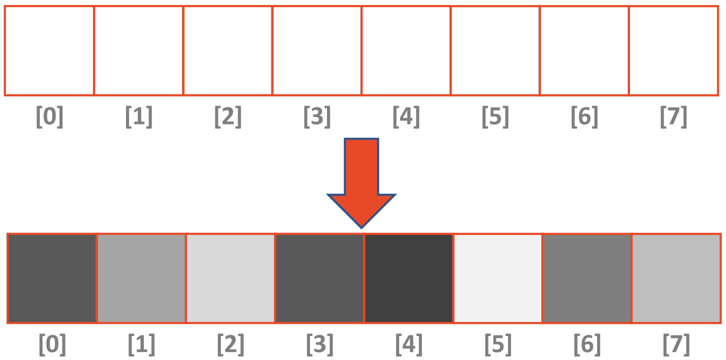


Array

Sorted Array

- Efficient general search structure
  Searches on the sort property run in O(lg(n)) with Binary Search

- Inefficient insert/remove
  Elements must be inserted and removed at the location dictated by the sort property, resulting shifting the array in memory – an O(n) operation

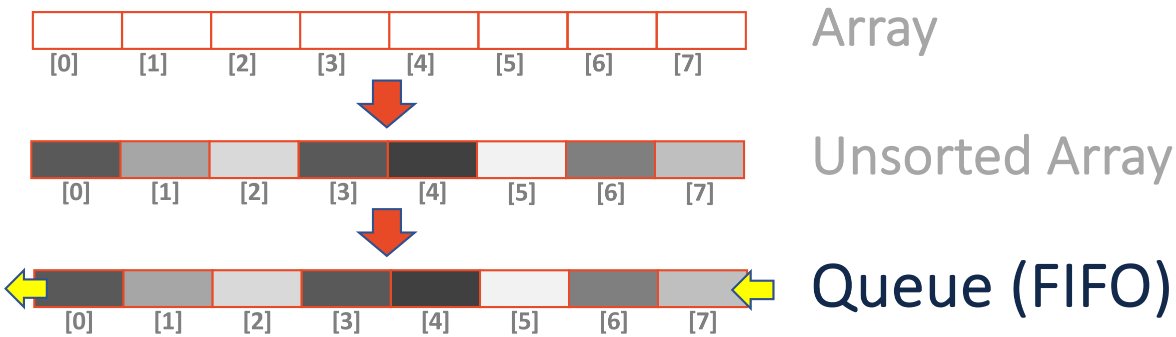


Array

Unsorted Array

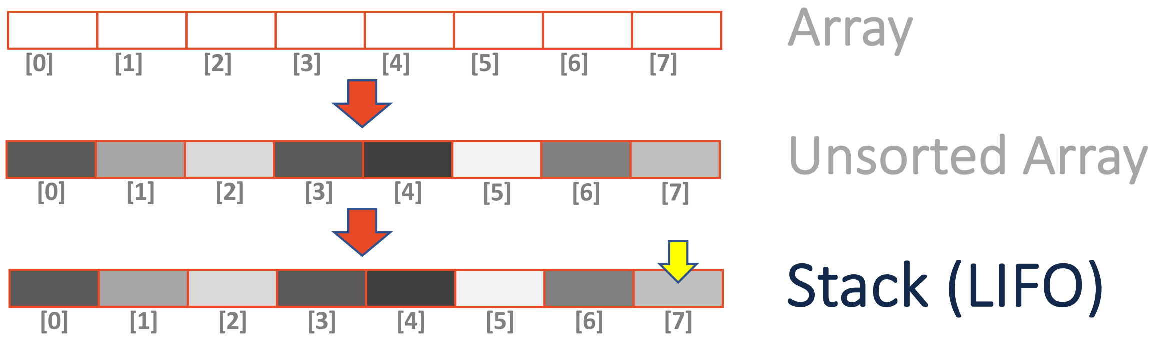
- Constant time add/remove at the beginning/end
  Amortized O(1) insert and remove from the front and of the array
  <u>Idea:</u> Double on resize

- Inefficient global search structure
  With no sort property, all searches must iterate the entire array; O(1) time



- First In First Out (FIFO) ordering of data
  Maintains an arrival ordering of tasks, jobs, or data

- All ADT operations are constant time operations
  enqueue() and dequeue() both run in O(1) time



- Last In First Out (LIFO) ordering of data
  Maintains a "most recently added" list of data

- All ADT operations are constant time operations
  push() and pop() both run in O(1) time

In Review: Data Structures

Array
- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
  - Hashing
  - Heaps
    - Priority Queues
  - UpTrees
    - Disjoint Sets

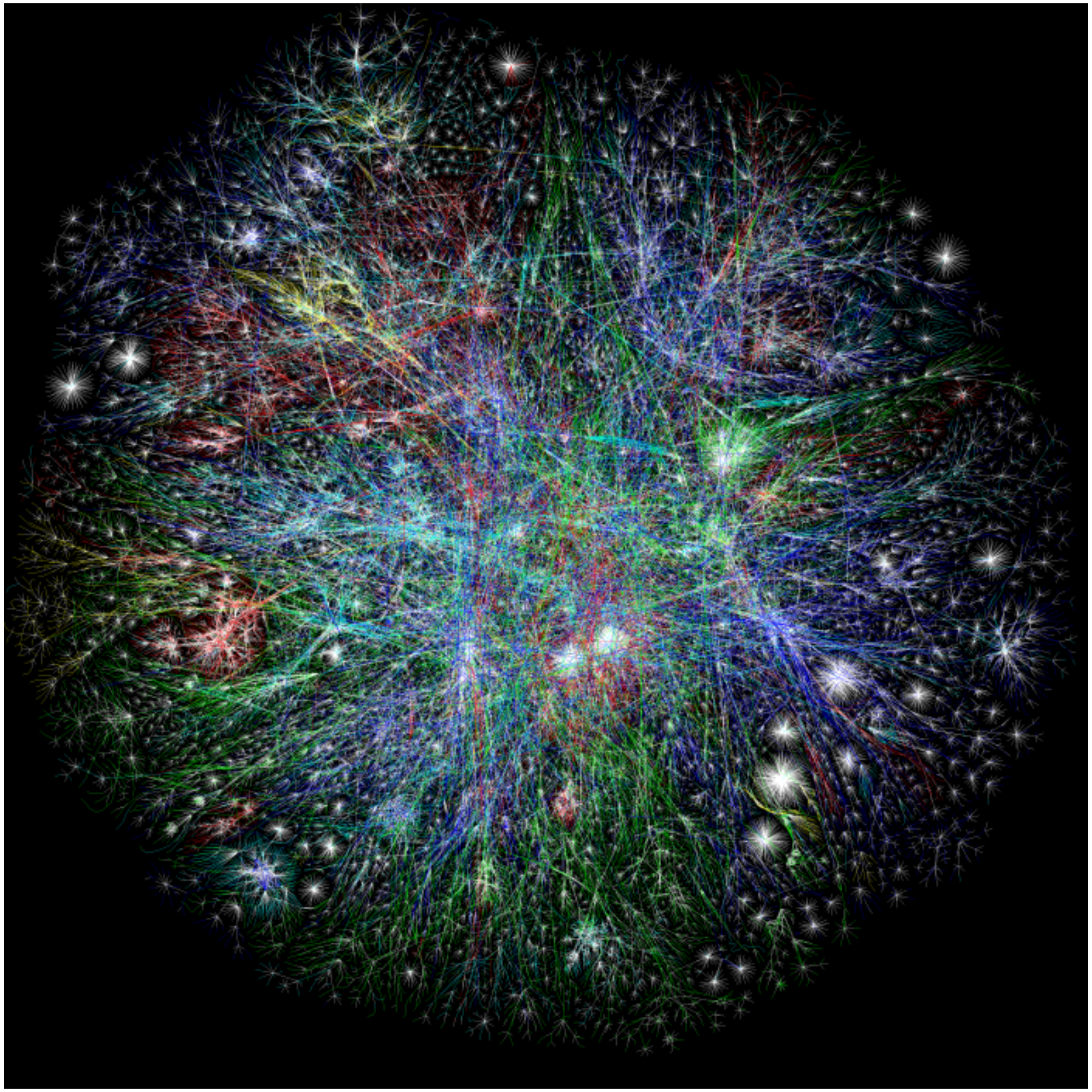
Linked
- Doubly Linked List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree

In Review: Data Structures

Array
- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
  - Hashing
  - Heaps
    - Priority Queues
  - UpTrees
    - Disjoint Sets

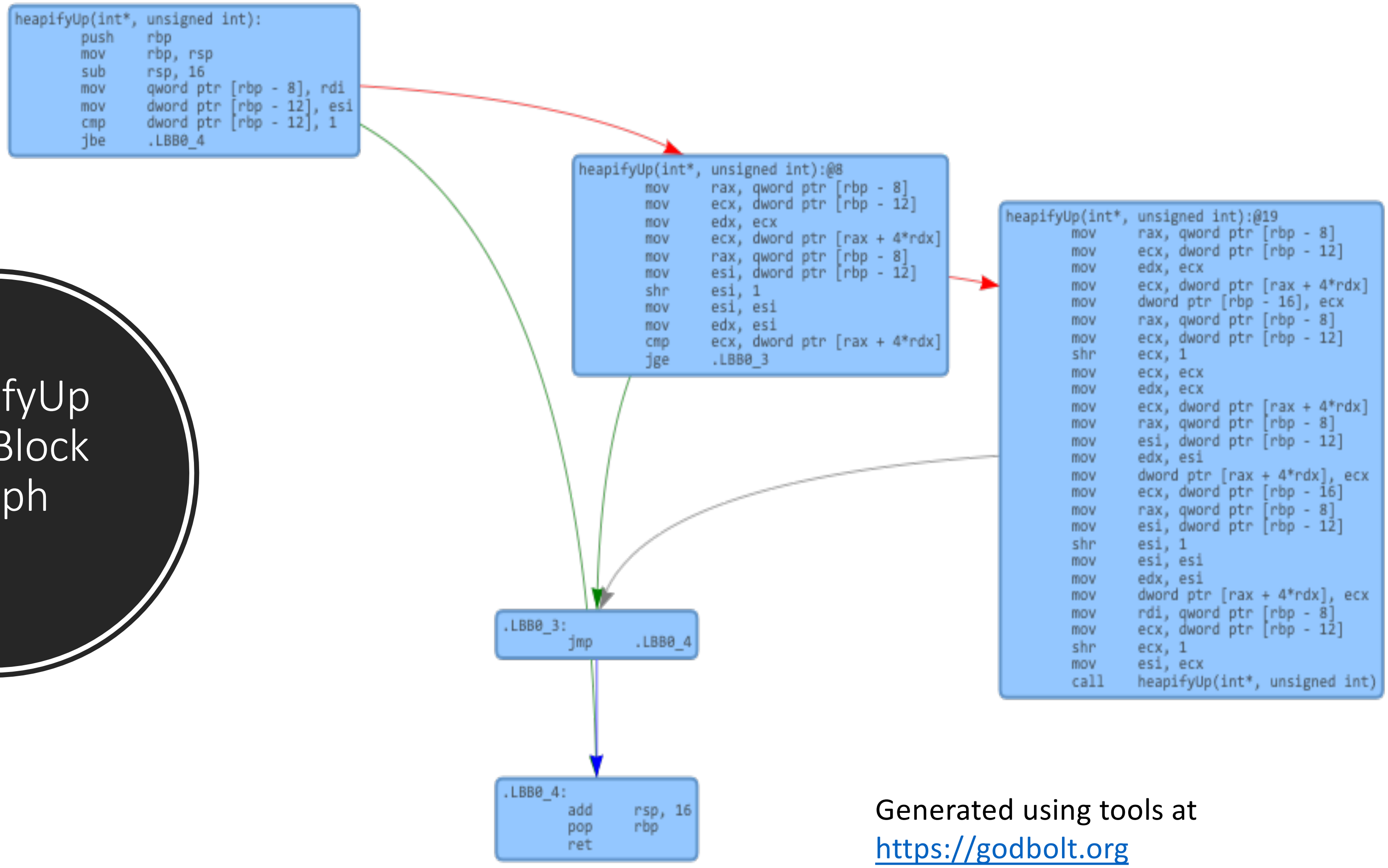
Graphs

Linked
- Doubly Linked List
- Skip List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree

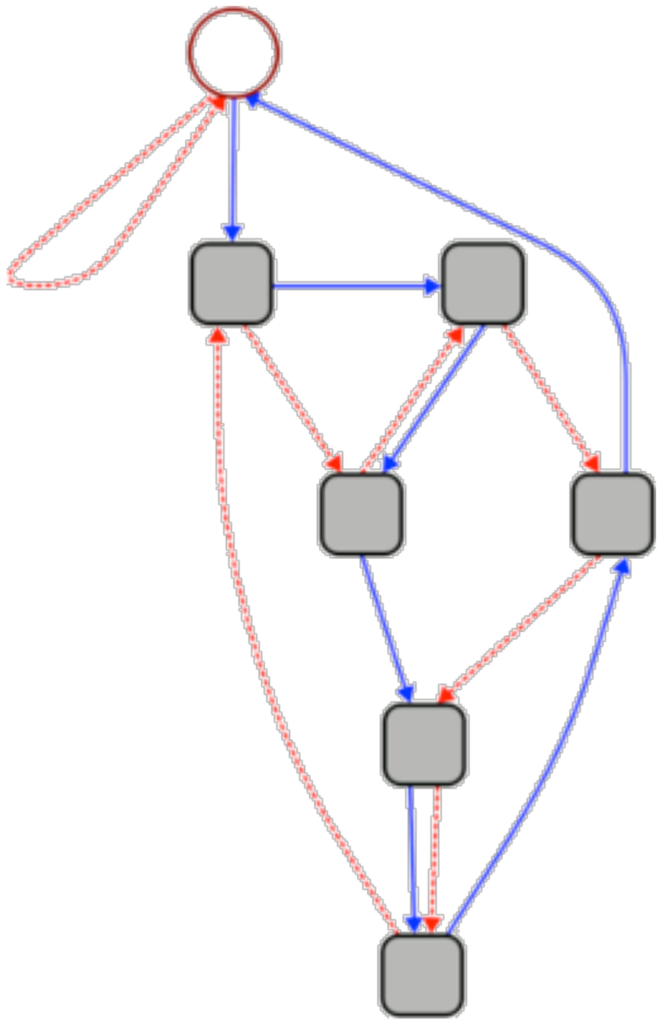


The Internet 2003
The OPTE Project (2003)
Map of the entire internet; nodes are routers; edges are connections.

HeapifyUp BasicBlock Graph



Generated using tools at https://godbolt.org

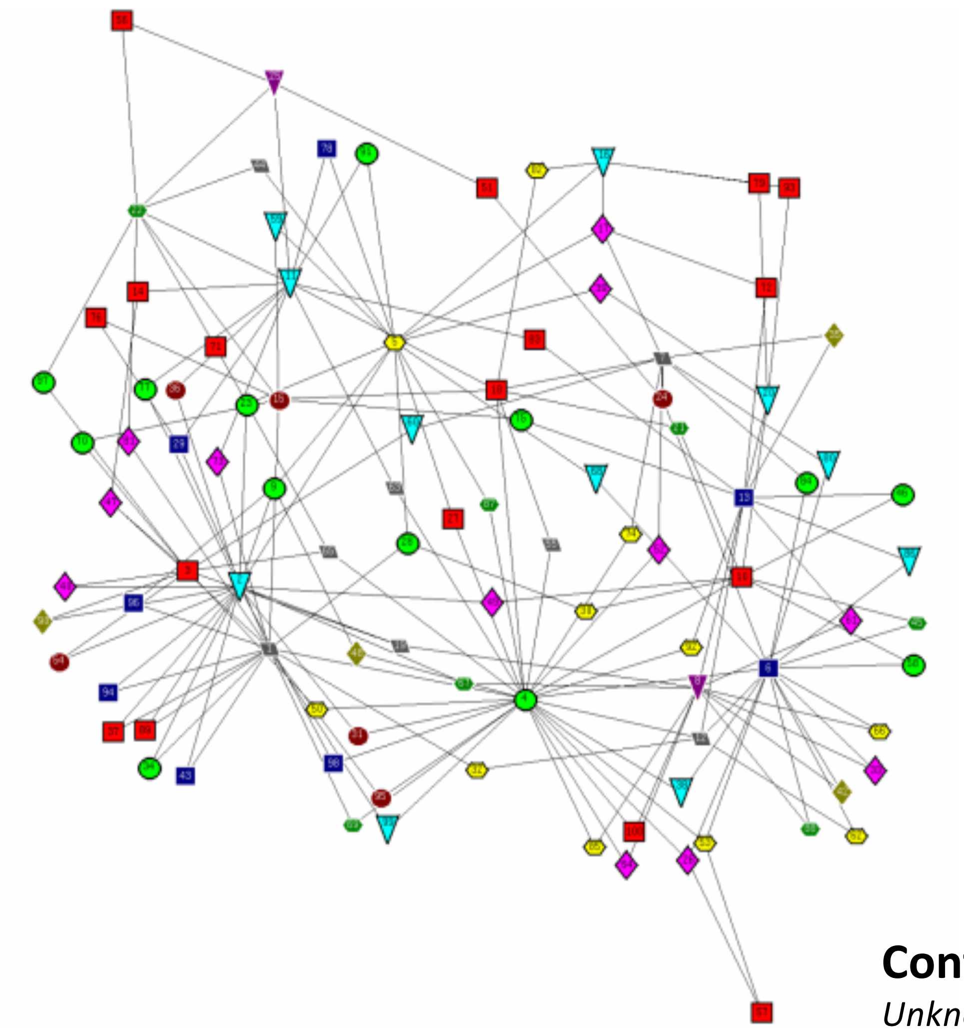


This graph can be used to quickly calculate whether a given number is divisible by 7.

1. Start at the circle node at the top.
2. For each digit **d** in the given number, follow **d** blue (solid) edges in succession. As you move from one digit to the next, follow **1** red (dashed) edge.
3. If you end up back at the circle node, your number is divisible by 7.

3703

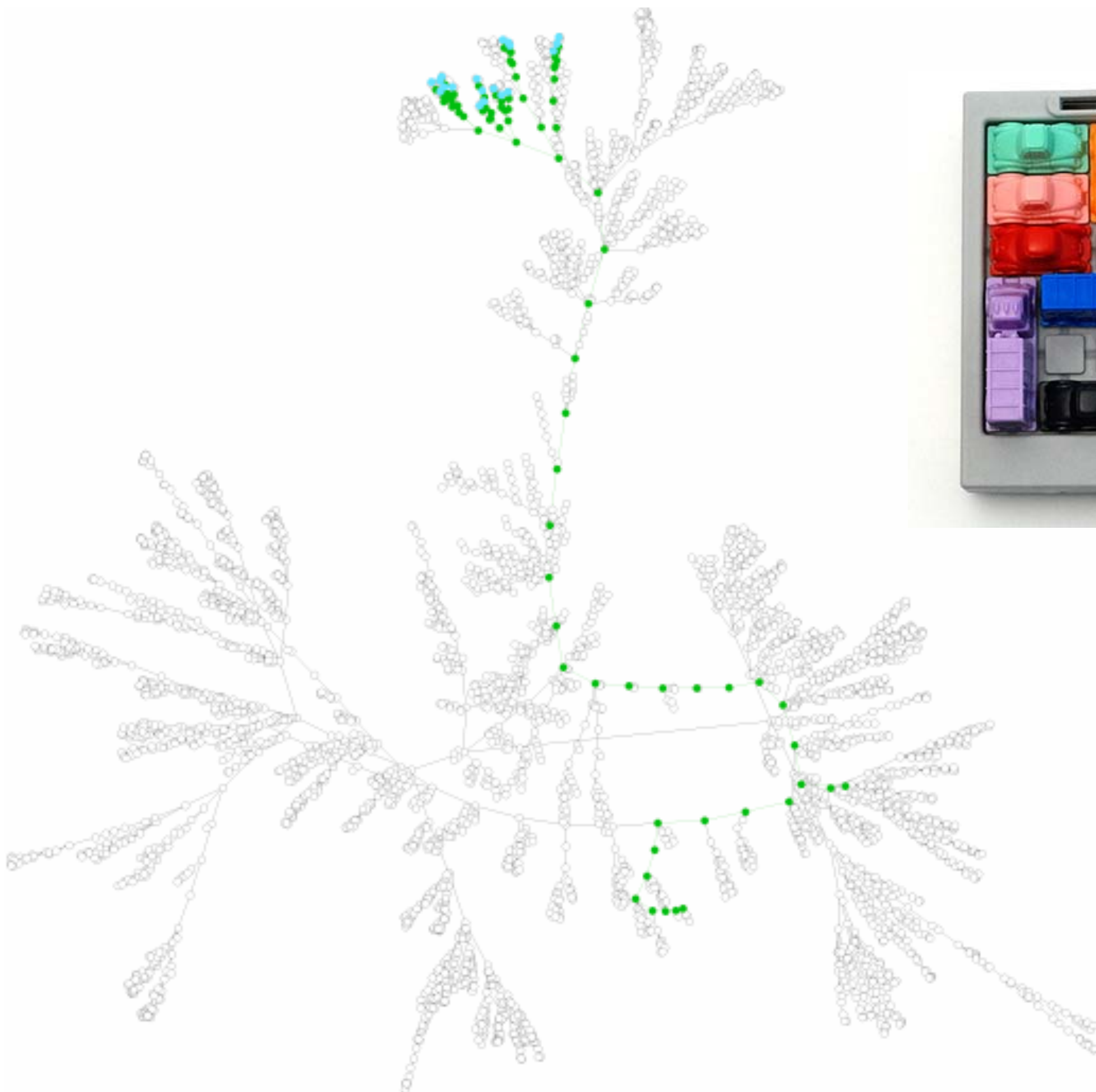
“Rule of 7”
Unknown Source
Presented by Cinda Heeren, 2016



Conflict-Free Final Exam Scheduling Graph

Unknown Source

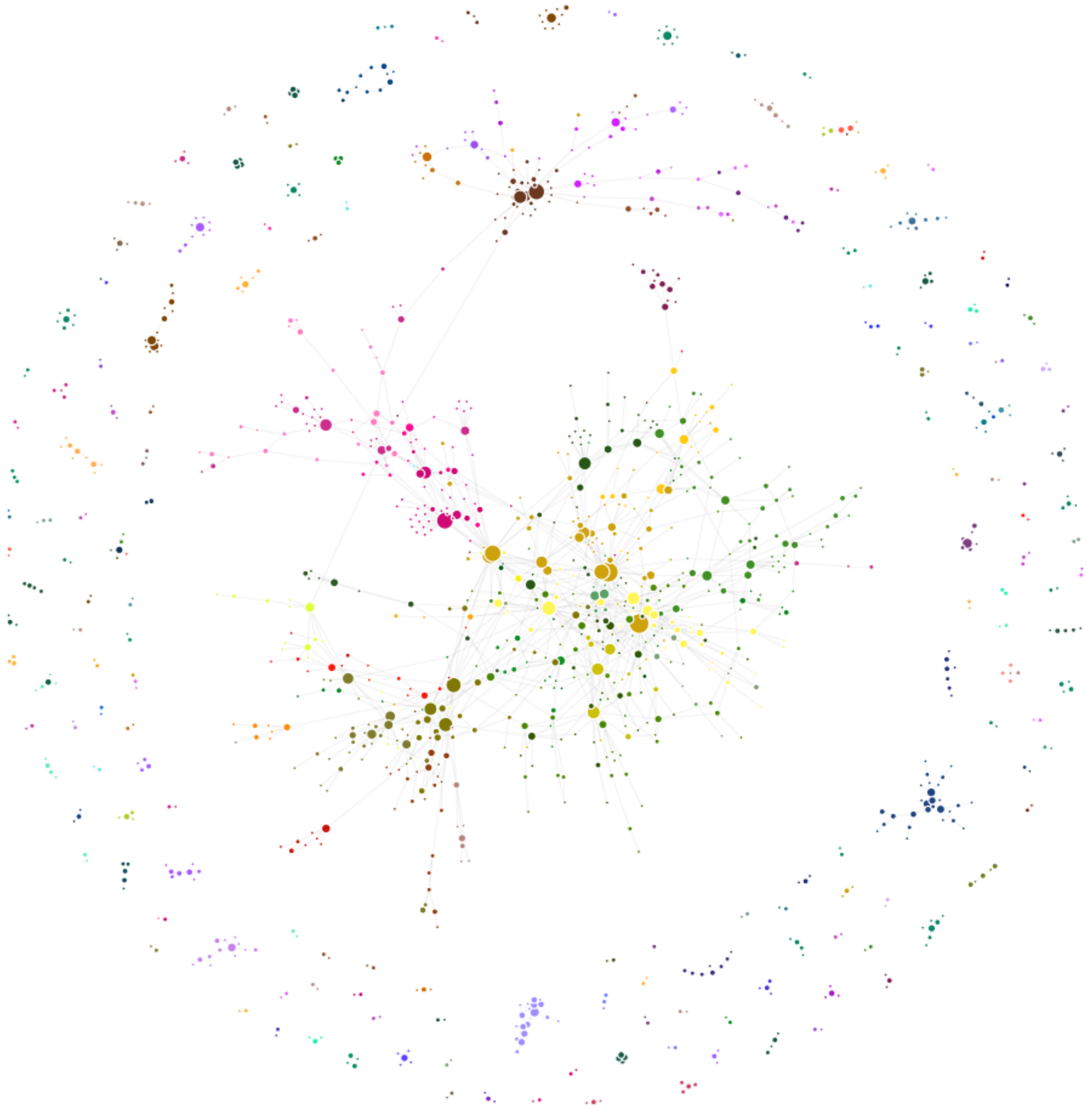
Presented by Cinda Heeren, 2016



“Rush Hour” Solution

Unknown Source

Presented by Cinda Heeren, 2016

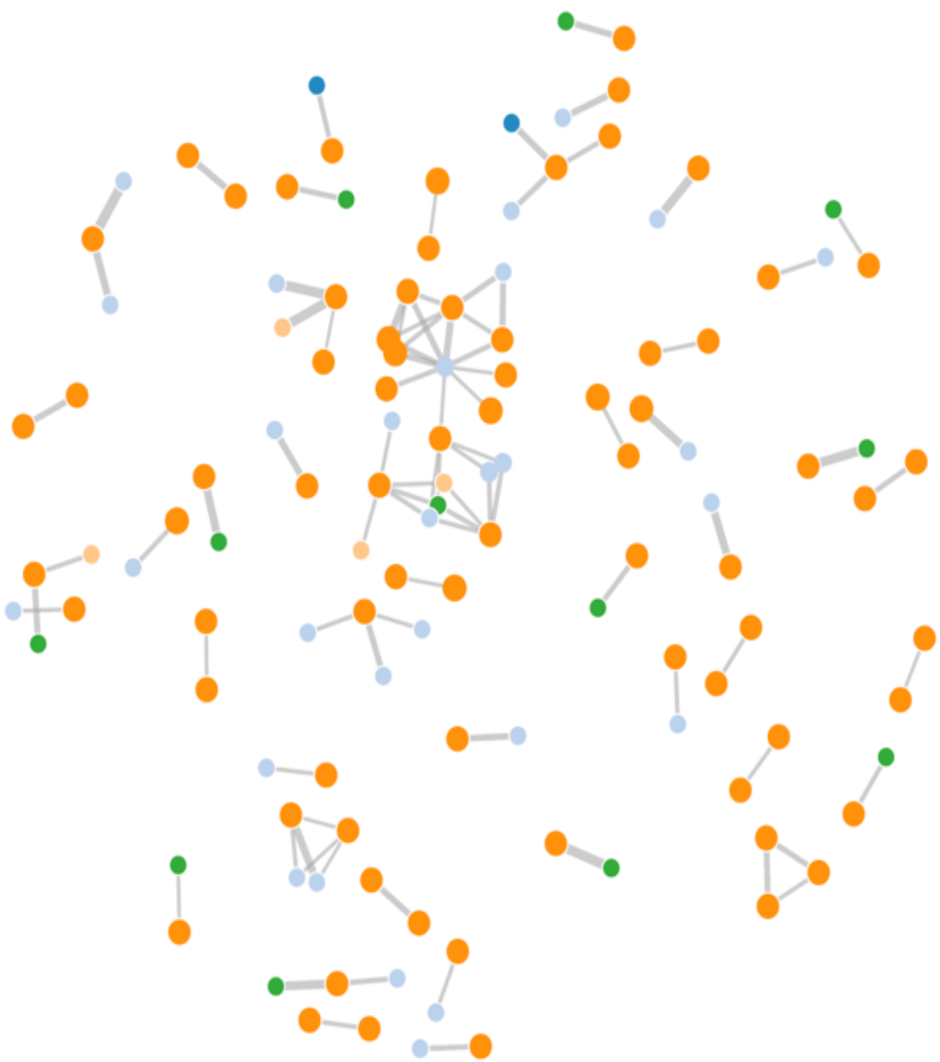


Class Hierarchy At University of Illinois Urbana-Champaign

A. Mori, W. Fagen-Ulmschneider, C. Heeren

Graph of every course at UIUC; nodes are courses, edges are prerequisites

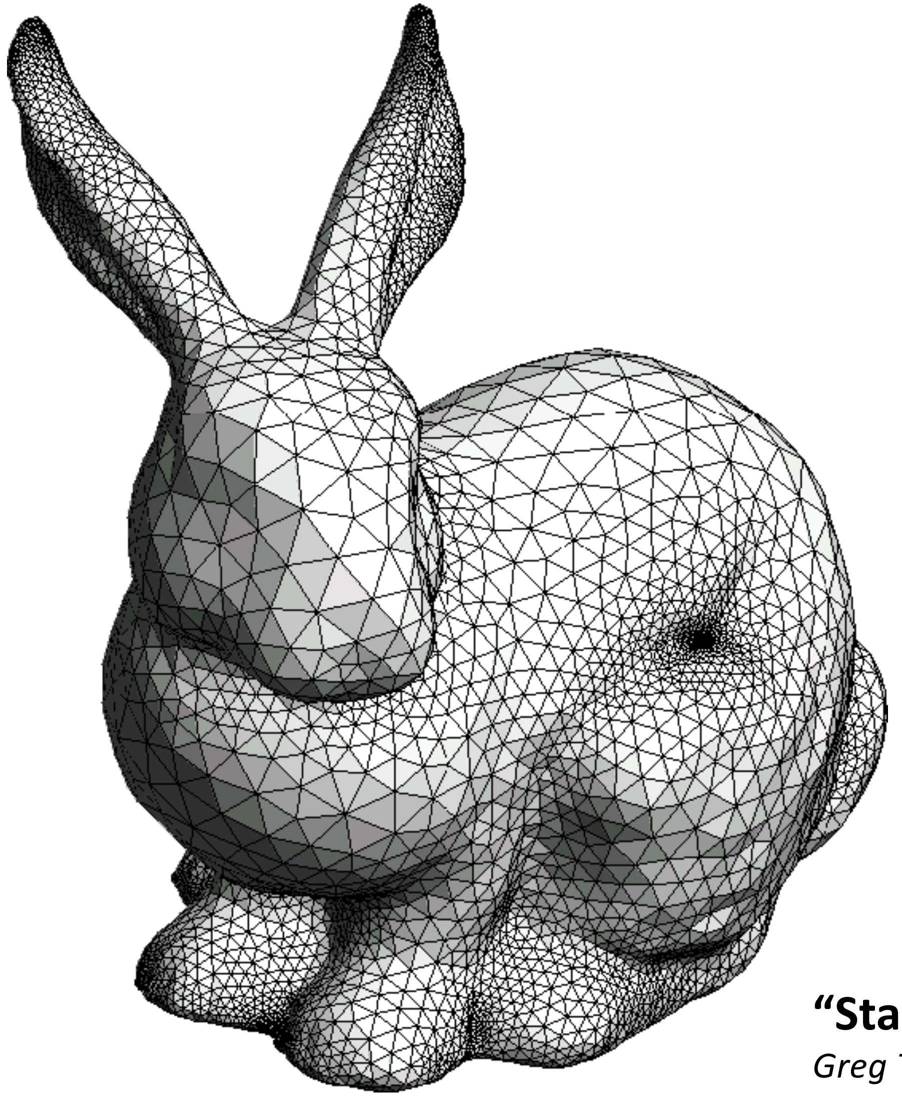
http://waf.cs.illinois.edu/discovery/class_hierarchy_at_illinois/



MP Collaborations in CS 225

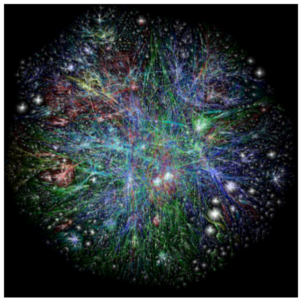
Unknown Source

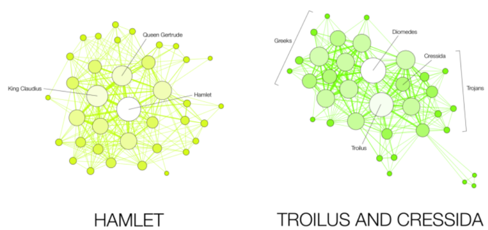
Presented by Cinda Heeren, 2016

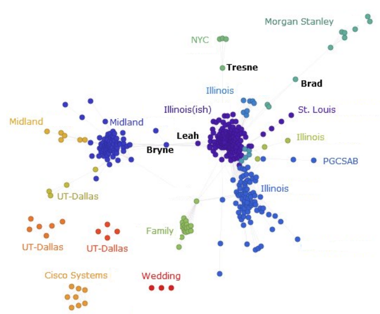


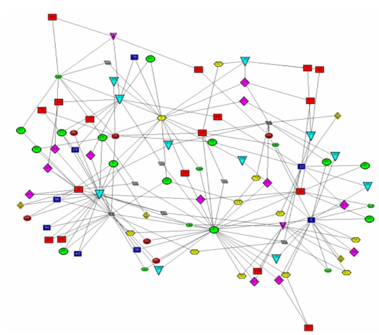
“Stanford Bunny”
Greg Turk and Mark Levoy (1994)

Graphs









To study all of these structures:

1. A common vocabulary
2. Graph implementations
3. Graph traversals
4. Graph algorithms