INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 14

Instructor: James Pope

Email: jpope8@gmu.edu

Today

- Review Last Class
  - Union-find, weighted-union, path-compression
- Schedule
  - Bloom Filters
  - Review

Dynamic Connectivity

Sedgewick/Wayne 1.5

- The “is connected to” is an *equivalence* relationship.
  - *Reflexive*: a is connected to a
  - *Symmetric*: a is connected to b, b is connected to a
  - *Transitive*: if a is connected to b and b is connected to c, then a is connected to c
- Within the context of our weighted graph
  - Equivalence class is a connected component
  - Union two vertices combines their components
  - Find the component identifier of a vertex. Two vertices that are connected will have the same component identifier.

UF Quick-Find Example

Sedgewick/Wayner 1.5

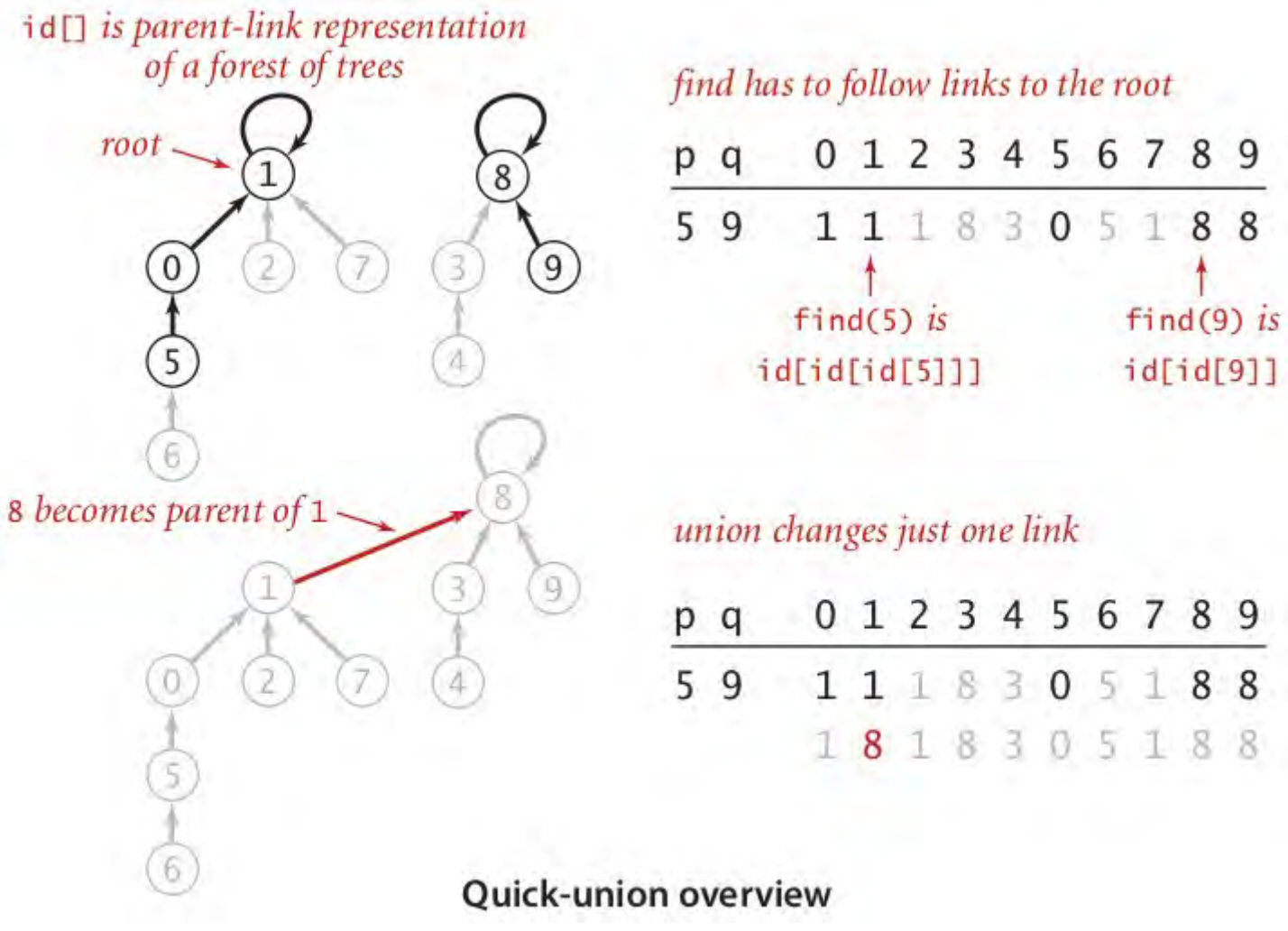
		id[]									
p	q	0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9
		0	1	2	3	3	5	6	7	8	9
3	8	0	1	2	3	3	5	6	7	8	9
		0	1	2	8	8	5	6	7	8	9
6	5	0	1	2	8	8	5	6	7	8	9
		0	1	2	8	8	5	5	7	8	9
9	4	0	1	2	8	8	5	5	7	8	9
		0	1	2	8	8	5	5	7	8	8
2	1	0	1	2	8	8	5	5	7	8	8
		0	1	1	8	8	5	5	7	8	8
8	9	0	1	1	8	8	5	5	7	8	8

UF Quick-Find Analysis

- Find operation is constant, just need to access array once (why this is called quick-find).
- Union operation is not constant
  - What did we have to do in order to union? How much did this cost us? O(N)
- Given some sequence of M operations (where M is approximately N), split evenly
  - 1/2 N x find() + 1/2 N x union()
  - 1/2 N x O(1) + 1/2 N x O(N)
  - UF with Quick-Find is quadratic $O(N^2)$
  - Cannot have more than N-1 union operations, Why?

UF Quick-Union Overview

Sedgewick/Wayne 1.5

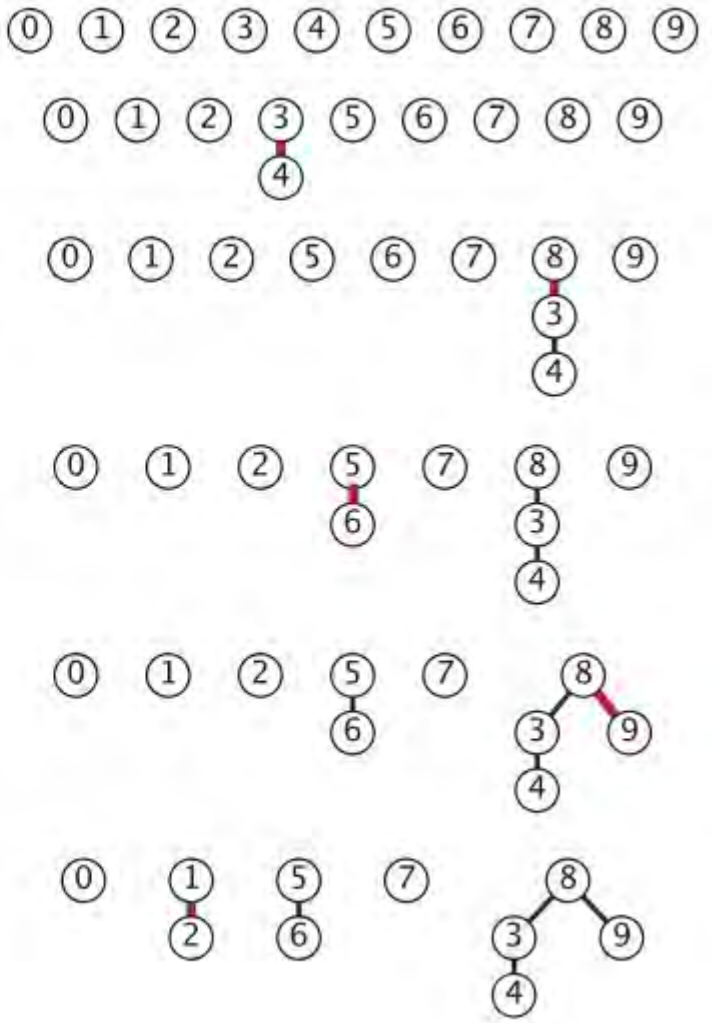


Quick-union overview

UF Quick-Union Example 1/2

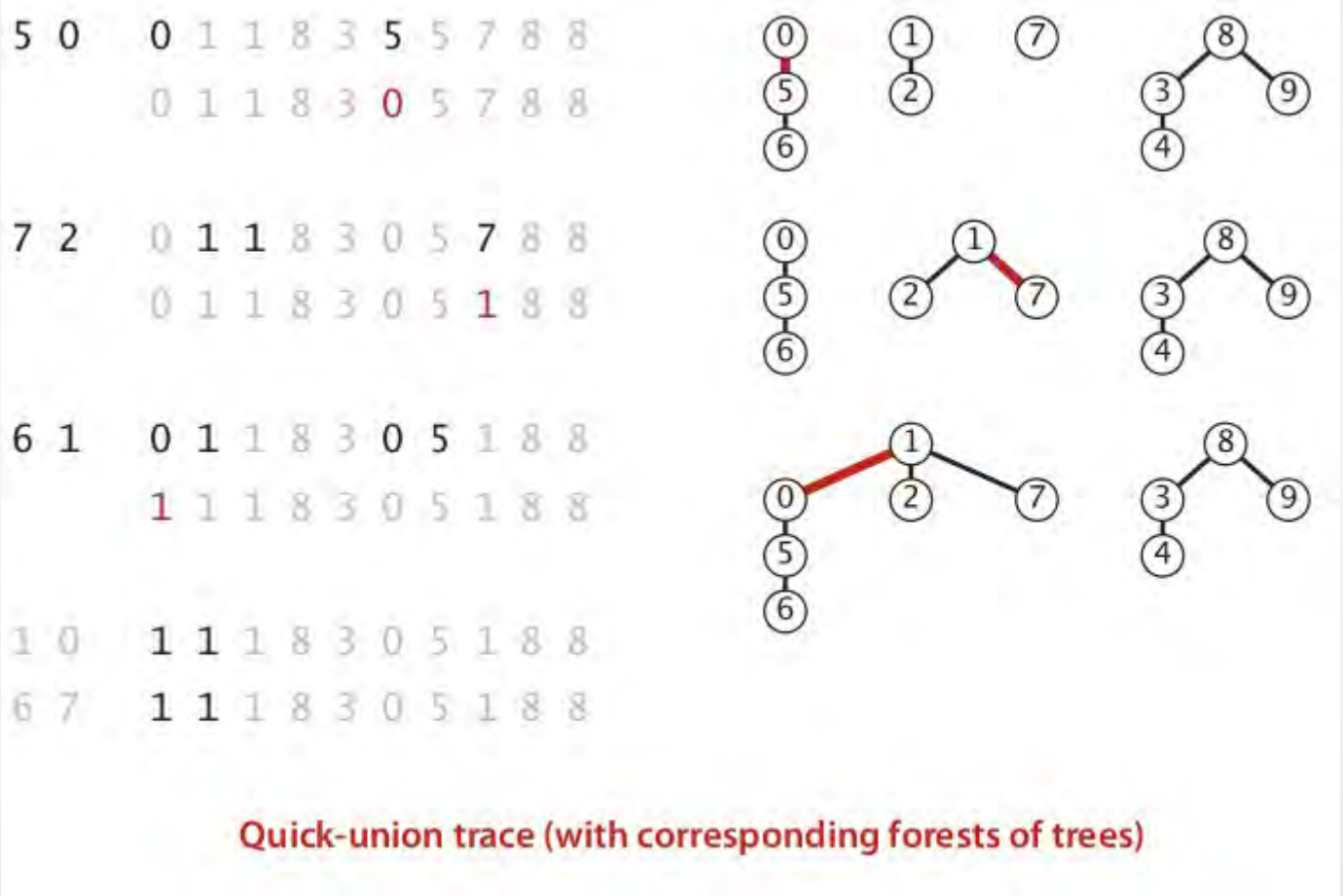
Sedgewick/Wayner 1.5

		id[]									
p	q	0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9
		0	1	2	3	3	5	6	7	8	9
3	8	0	1	2	3	3	5	6	7	8	9
		0	1	2	8	3	5	6	7	8	9
6	5	0	1	2	8	3	5	6	7	8	9
		0	1	2	8	3	5	5	7	8	9
9	4	0	1	2	8	3	5	5	7	8	9
		0	1	2	8	3	5	5	7	8	8
2	1	0	1	2	8	3	5	5	7	8	8
		0	1	1	8	3	5	5	7	8	8
8	9	0	1	1	8	3	5	5	7	8	8



UF Quick-Union Example 2/2

Sedgewick/Wayner 1.5

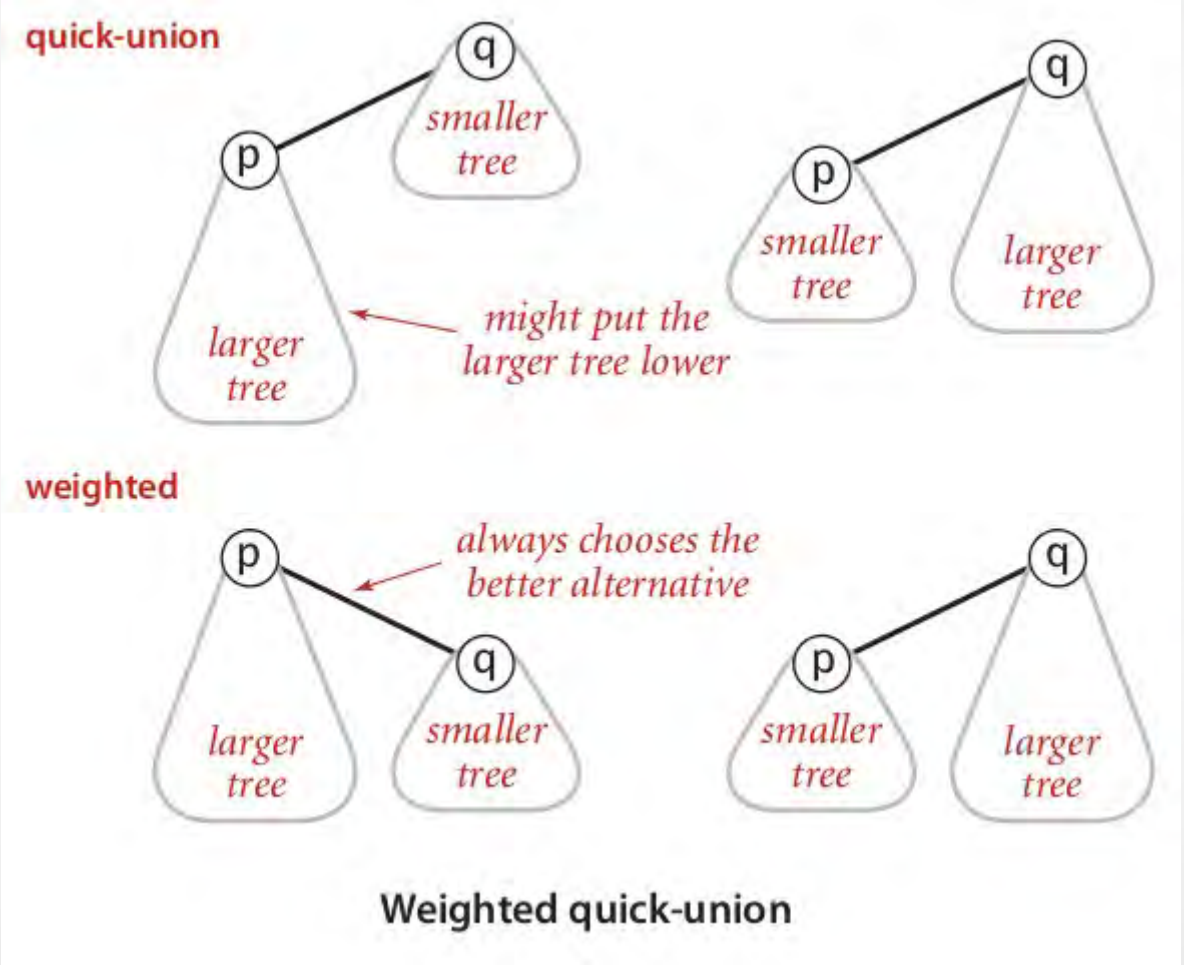


UF Quick-Union Analysis

- Find operation is worse case O(N). May get tree that is a linked list. How could this happen?
- Union operation calls find twice but otherwise is constant.
- Given some sequence of M operations (where M is approximately N), split evenly
  - 1/2 N x find() + 1/2 N x union()
  - 1/2 N x O(N) + 1/2 N x O(2N)
  - UF with Quick-Union is quadratic O(N^2)
- Not really any better than Quick-find
  - However, new idea based on quick-union

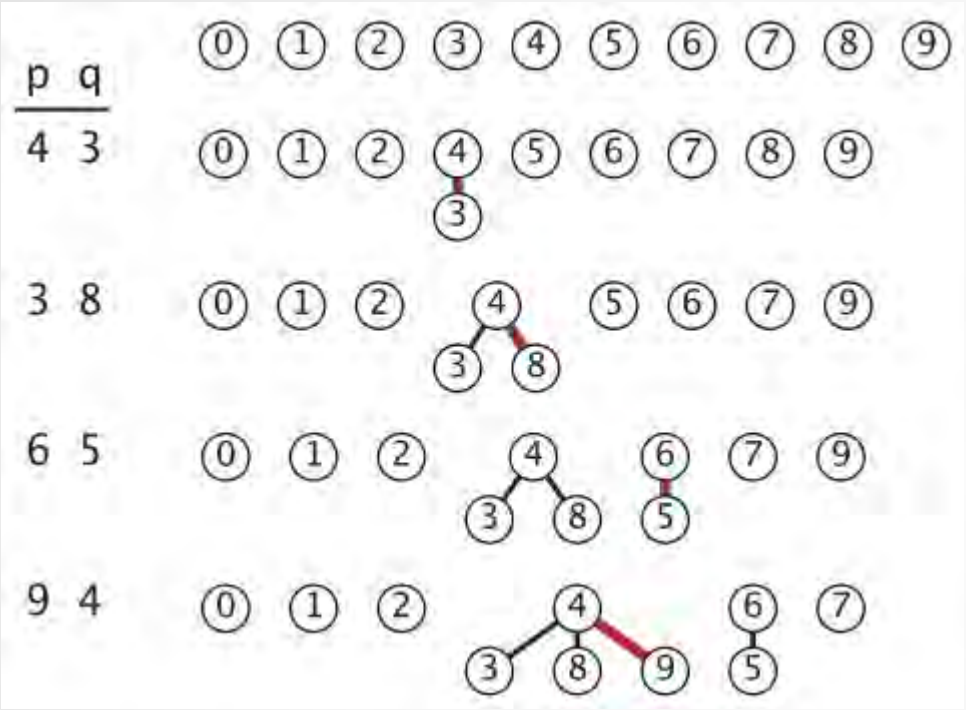
UF Weighted Quick-Union Overview

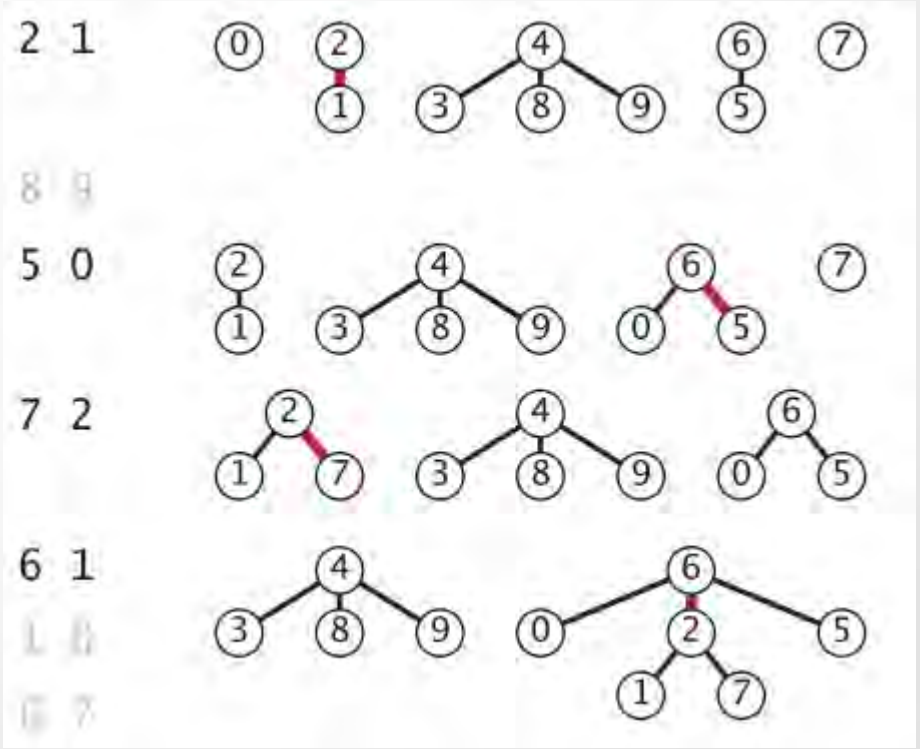
Sedgewick/Wayne 1.5



UF Weighted Quick-Union Example 1/2

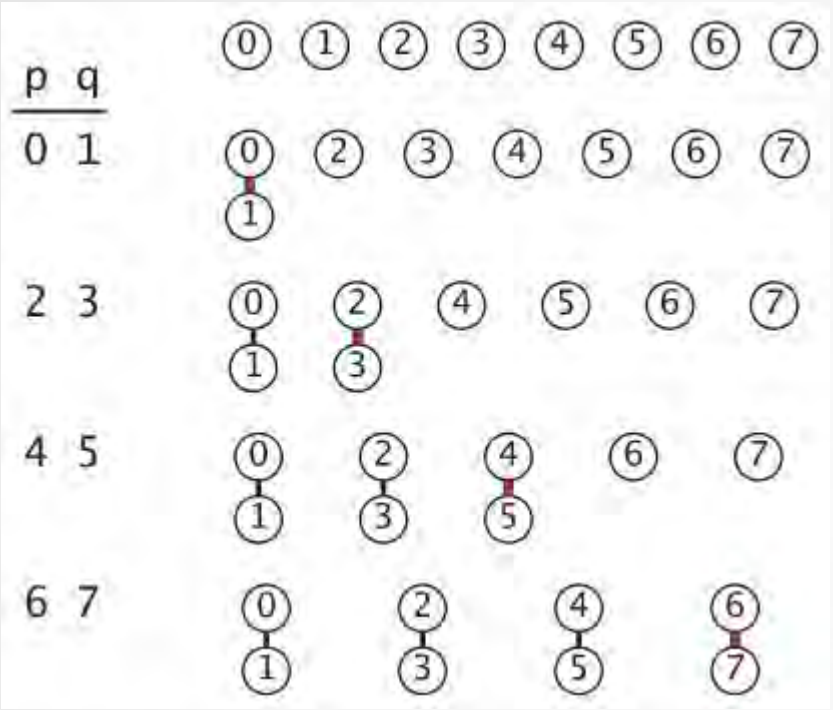
Sedgewick/Wayner 1.5

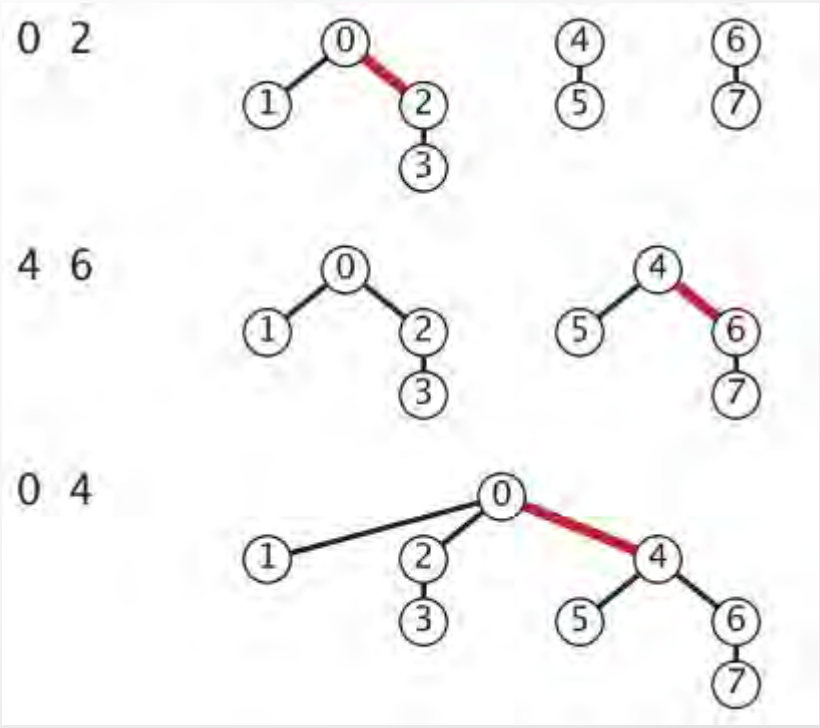




UF Weighted Quick-Union Example 2/2

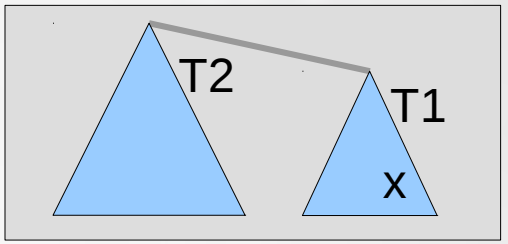
Sedgewick/Wayner 1.5





UF Weighted Quick-Union Analysis

- Find and union are determined by the maximum height of any tree. Due to the weighted heuristic:
  - Depth of any node is at most lg(N)
    - Size doubles, height increases by one
  - Thus, find and union expected logarithmic



- Given some sequence of M operations (where M is approximately N), split evenly
  - 1/2 N x find() + 1/2 N x union()
  - 1/2 N x O(lg(N)) + 1/2 N x O(2lg(N))
  - UF with Weighted Quick-Union is quadratic O(N lg(N))

UF Weighted QU w/Path Compression

- Path-compression Heuristic: When performing find that traverses the tree, modify the tree to make it shorter.
  - Does not cost any extra and takes constant time, only one or two extra lines of code, still correct operations
  - Analysis is beyond scope but takes *iterated logarithmic* time (similar growth to *inverse Ackermann Function*)
- Iterated logarithm (denoted lg*(N)): given positive number N, the *number of times* necessary to take the logarithm so that the number is reduced <= 1.
  - $lg^*(2^4) = lg(lg(lg(2^4))) = 3$
  - $lg^*(2^{16}) = 4$
  - $lg^*(2^{65536}) = 5$, practically never more than 5, constant

Union-Find Summary

Sedgwick 1.5

- Typical to implement weighted union and path compression. Both heuristics are easy to implement.
  - Weighted union-find with path compression is nearly optimal as a constant time algorithm has been proven not to exist.

Algorithm	Constructor	union	find
Quick-Find	N	N	1
Quick-Union	N	Tree height	Tree height
Weighted QU	N	lg(N)	lg(N)
Weighted QU w/PC	N	~1*	~1*
Impossible	N	1	1

* Very, very , very nearly one, amortized, close to best that can be done for problem

Questions?

Cache Problem

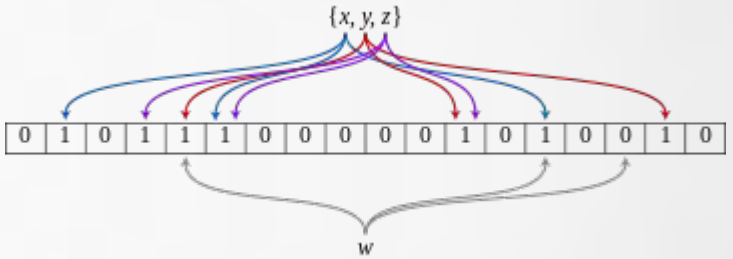
- Database key/value pairs (e.g. BigTable, Hbase/HDFS, Cassandra). Clients make requests to read a value using the key.
  - The database has to search multiple disks to find value
  - Significant performance improvement if we can cache all the keys and disk location in memory
  - Given: 1k memory and 100 keys, each key 100 bytes
    - Has your boss just given you an impossible task!
- Cache problem: Have a long task (disk/network access) ideally avoided using cache. Limited memory, need to store more keys than we have available memory.
  - Want to filter disk accesses through memory cache

Bloom Filter

- *Probabilistic data structure*
  - May return incorrect answer to operation/query
- Bloom filter is a logical set (no remove operation)
    - Add an element (key) to the set
    - Membership, determine if element is in the set
- The add operation will always work. Testing membership may not always work. The bloom filter may indicate an element is in the set when it was never added (false positive)! Bloom filters never generate false negatives.
  - False positive: Truth=false, Answer= true
  - False negative: Truth=true, Answer=false

Bloom Filter Overview

- Uses k independent hash functions to determine which bit in a bitset of fixed size m bits should be set for a given key. The filter will have n keys.
- The add operation takes the key and hashes to k different bit positions, sets those bits to 1
- The membership operation takes key and hashes to k different bit positions
  - Returns true if all bit positions are 1, returns false otherwise



An example of a Bloom filter, representing the set { x, y, z }. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set { x, y, z }, because it hashes to one bit-array position containing 0. For this figure, m = 18 and k = 3.

From: https://en.wikipedia.org/wiki/Bloom_filter

Bloom Filter Add Operation

- Can use prime numbers to implicitly define different hash functions. Can also use cryptographic hash functions (e.g. MD5).

```
public static final int PRIME1 = 0x797A8D77;
public static final int PRIME2 = 0x932688C1;
public static final int PRIME3 = 0xAD07A2D1;
public static final int PRIME4 = 0xF70E5939;

// Add with k=4 hash functions
public int add(Object element) {
    int hash1 = element.hashCode() ^ PRIME1;
    int hash2 = element.hashCode() ^ PRIME2;
    int hash3 = element.hashCode() ^ PRIME3;
    int hash4 = element.hashCode() ^ PRIME4;

    int m = this.getM();
    this.setBit( hash1 % m, true );
    this.setBit( hash2 % m, true );
    this.setBit( hash3 % m, true );
    this.setBit( hash4 % m, true );
}
```

Bloom Filter Membership Operation

- All bits must be set to true. This means that the element may be a member of the set. If operation returns false, element is definitely not in the set.

```
// Memebership with k=4 hash functions
public int isMember(Object element) {
    int hash1 = element.hashCode() ^ PRIME1;
    int hash2 = element.hashCode() ^ PRIME2;
    int hash3 = element.hashCode() ^ PRIME3;
    int hash4 = element.hashCode() ^ PRIME4;

    int m = this.getM();
    boolean bit1Set = this.getBit( hash1 % m );
    boolean bit2Set = this.getBit( hash2 % m );
    boolean bit3Set = this.getBit( hash3 % m );
    boolean bit4Set = this.getBit( hash4 % m );

    return bit1Set && bit2Set && bit3Set && bit4Set;
}
```

Bloom Filter Summary

- The false positive (FP) rate can be controlled by parameters k, m, and n.
  - See http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html
  - The more elements added, the higher the false positives rate becomes
  - Can rehash to keep FP low by using more memory
- Does not work if the application cannot handle false positives. For applications that can handle false positives (generally FP means a little extra, needless work), the approach works very well.

Review
Copy/Paste From First Lecture

This Course

- Program Design
  - Determine solutions to common problems
  - Evaluate those solutions
- Data Structures
  - Identify and understand useful data structures used in those solutions

- How and why should we evaluate solutions?

What we will cover

- Algorithm Related
  - Algorithm Analysis
  - Searching/Sorting
  - Hashing
- Programming
  - Generic Classes
  - Recursion
- Data Structures
  - Dynamic Lists
  - Linked Lists
  - Stacks and Queues
  - Trees
  - Graphs

What is a data structure?

- “A data structure is a representation of data and the operations allowed on the data” (Weiss, Chapter 6).
- Other textbooks will also refer to these (data/fields and operations on them) as “Abstract Data Types (ADTs)” (Lafore).

Data Representation	Operations
int 32 bit, two's complement	+,-,*,/,%
String, char array	length, charAt, indexOf
Stack, array or linked list	push, pop, peek, size
ordered array, binary heap	delMax, insert, size

Dynamic Arrays

- Java: ArrayLists
- Data structure like a paper and pencil list
- Need to copy things down/over to add items
- Need more paper if not enough space

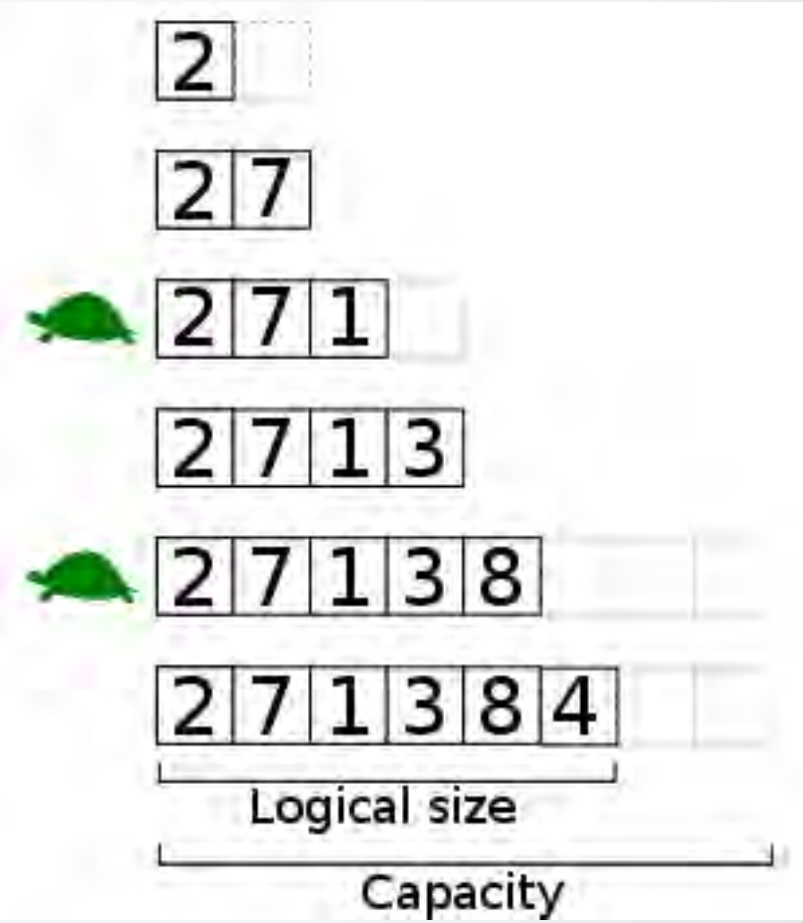


Image Source: http://en.wikipedia.org/wiki/File:Dynamic_array.svg

Linked List

- Data structure along the lines of a treasure hunt
  - Start at the beginning
  - At each “stop” find an item and the clue to the next place to look

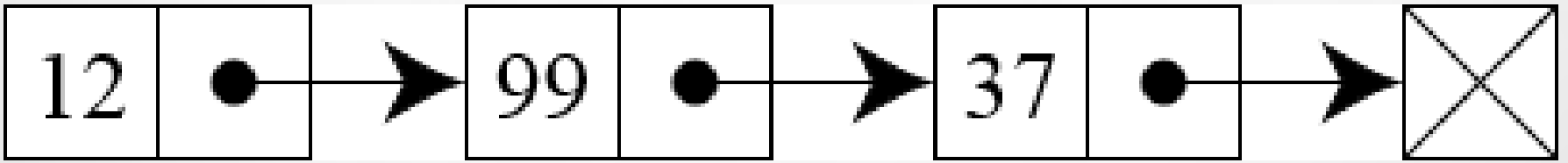
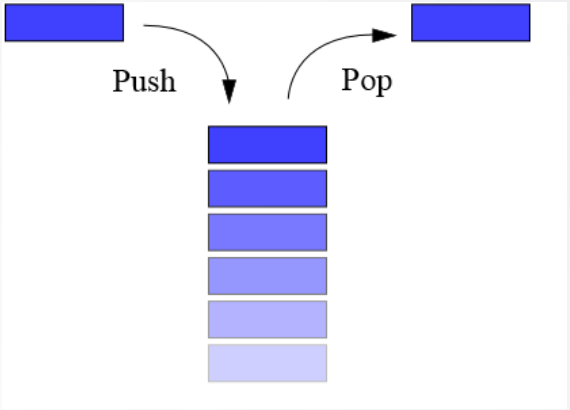


Image Source: http://en.wikipedia.org/wiki/File:Singly-linked-list.svg

Stacks and Queues

- Stack
  - Data structure that works like a... stack (e.g. a stack of paper)
- Queue
  - Data structure that works like a... queue (or a “line” if you aren't British)



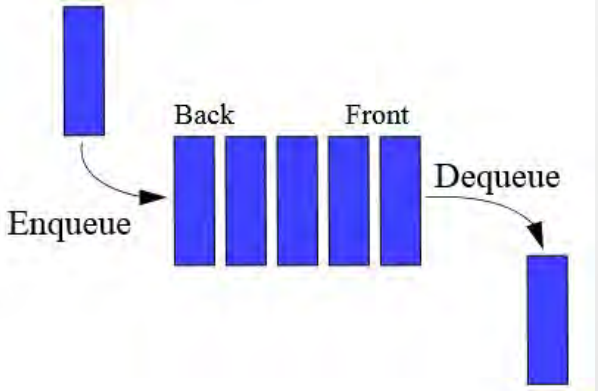
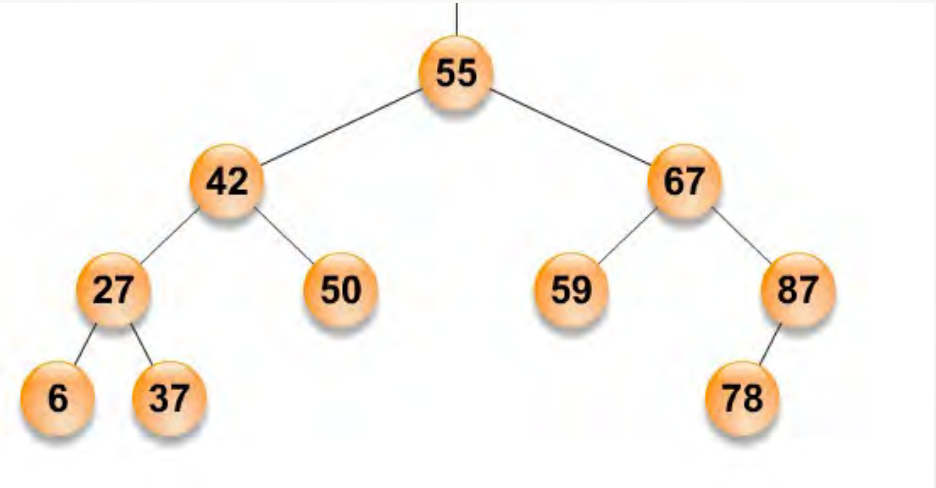


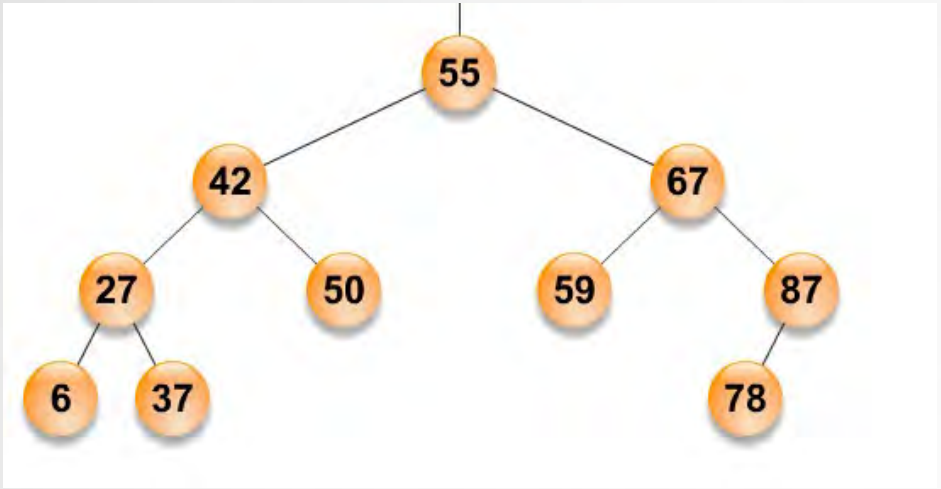
Image Source: http://en.wikipedia.org/wiki/File:Data_stack.svg and http://en.wikipedia.org/wiki/File:Data_Queue.svg

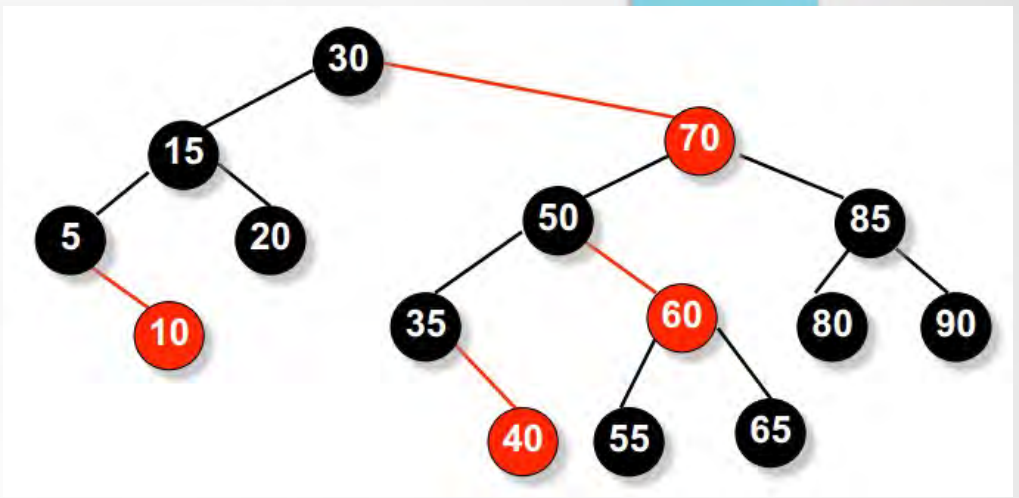
Trees

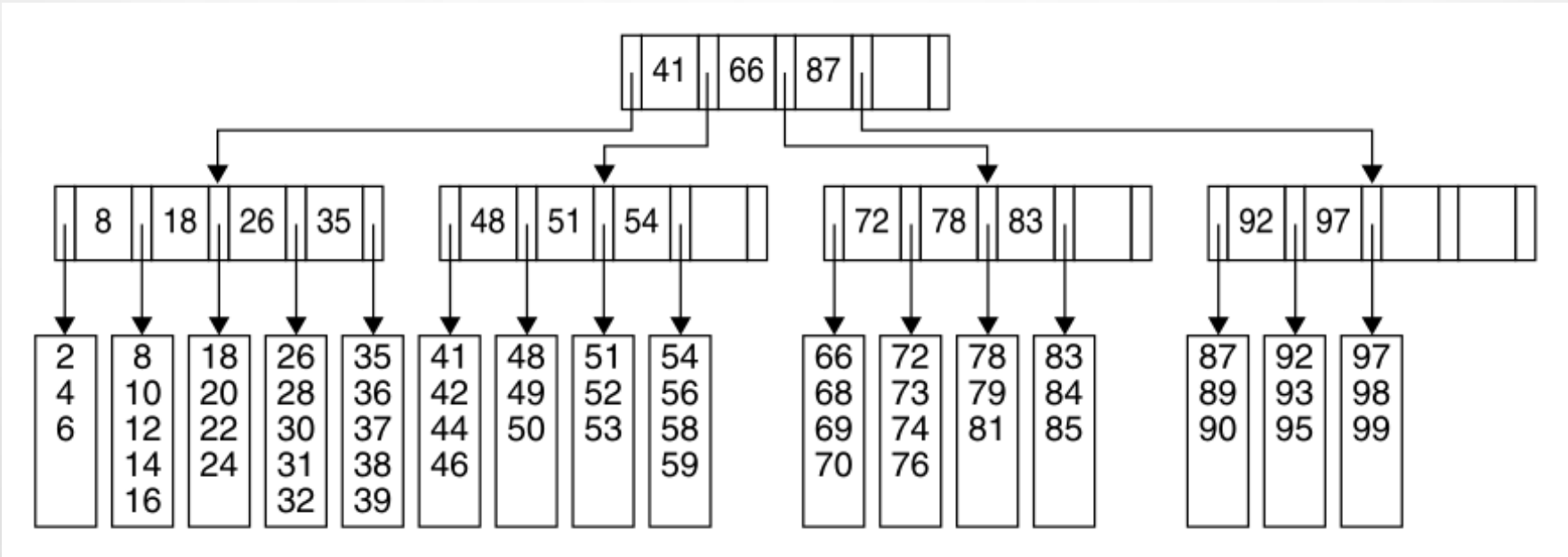
- Data structure which looks like an upside down tree (or the root system of a tree)
  - Nodes have parents and children
  - No loops



Types of Trees

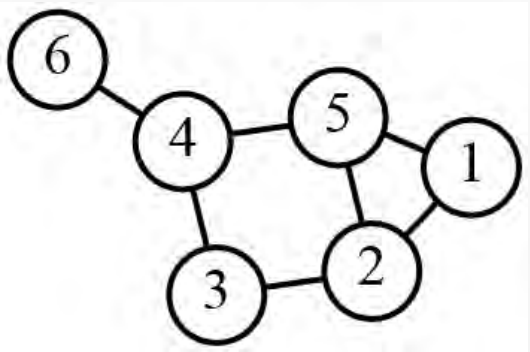






Graphs

- Data structure which works kinda like a road map
  - Locations = nodes
  - Roads (or “how to get from one location to another) = edges (or “how to get from one node to another)
  - Linked Lists and Trees are graphs too...



Data Structure Important Questions

- How does it work?
  - How do you find things?
  - How do you put things in?
  - What are the “rules”?
- What are the benefits / trade offs?
  - How fast is it?
  - How big is it?

What we will cover

- Algorithm Related
  - Algorithm Analysis
  - Searching/Sorting
  - Hashing
- Programming
  - Generic Classes
  - Recursion
- Data Structures
  - Dynamic Lists
  - Linked Lists
  - Stacks and Queues
  - Trees
  - Graphs

Generic Classes

- Basically: another technique for how to program so that you can reuse your code
  - In Java there are generic “collections” classes for each of the data structures I just mentioned
  - We'll be writing our own from scratch to see how they work
- I *expect* that you've already seen generics in Java in your previous classes
  - But maybe not in so much detail

Recursion

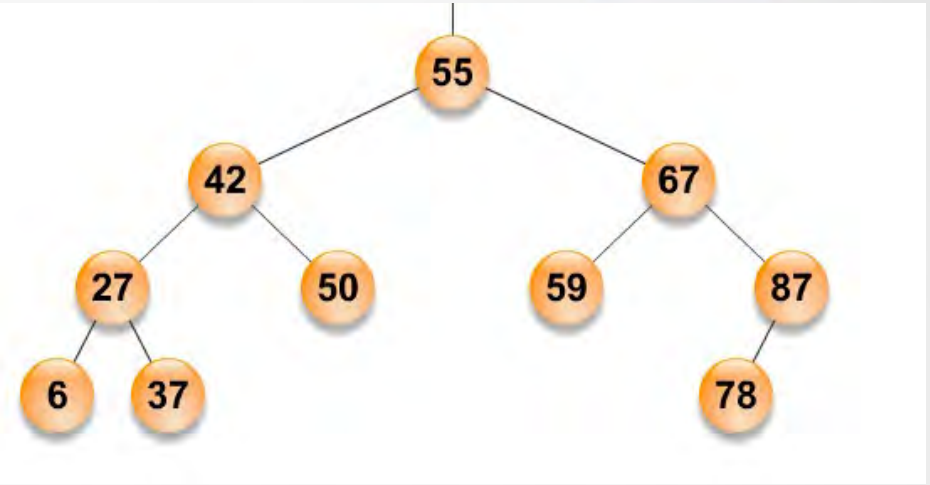
- Call a function/method from inside the same function/method
- Idea: keep doing the same thing, reducing the problem
  - smaller subset of the problem
  - one step closer to the answer
- Key components
  - recursive case (when to keep going)
  - base case (when to stop)

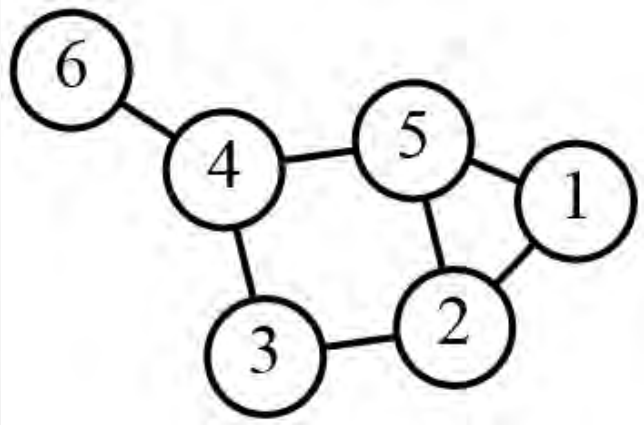
What we will cover

- Algorithm Related
  - Algorithm Analysis
  - Searching/Sorting
  - Hashing
- Programming
  - Generic Classes
  - Recursion
- Data Structures
  - Dynamic Lists
  - Linked Lists
  - Stacks and Queues
  - Trees
  - Graphs

Searching and Data Structures

- Visit every item?
- Visit only some items?
- Searching a List...
  - 1, 3, 6, 24, 26, 38
- Searching in a Tree...
- Searching in a Graph...





Sorting

- How to sort things efficiently
  - so we can find them!
- Lots of sorts (we won't cover them all)
  - Insertion
  - Bucket
  - Merge
  - Quick
  - Heap
  - …

Hashing

- Basically, how to “map” things to a fixed set of locations
  - The bucket sort is a type of hash
    - 26 letters in the alphabet
    - Papers are “mapped” to each “letter pile” so I can quickly look up people with “A” names
  - What do you do when multiple things “map” to the same location?
- We'll do much better hashing

Algorithm Analysis

- Algorithm

  = how you do things
- Algorithm Analysis

  = analyze how well you do things
- Data structures need algorithms to:
  - insert data
  - find data that (may) already exist
  - delete data
- What defines well?

Review
Copy/Paste From Lectures

List Summary

- Though arrays are limited in functionality, constants for arrays are much faster

Operation Implementation	get set	add remove end	insert remove begin	insert remove middle	search	Grow Shrink
Array	1	-	-	-	-	No
Static Array	1	1	N	N	N	No
Dynamic Array	1	1*	N	N	N	Yes
Single Linked List	N	N	1	N**	N	Yes
Doubly Linked List	N	1	1	N**	N	Yes
? ***	1	-	-	-	1	Yes

* Amortized analysis
** Have to search first N, then insert is constant
*** Hash Tables, will cover later

Stacks and Queues Summary

Stack

Operation Implementation	push	pop	peek	isEmpty	size
Dynamic Array	1	1	1	1	1
Doubly Linked List	1	1	1	1	1

Easiest to implement Dynamic Array, constants are better

Queue

Operation Implementation	enqueue	dequeue	peek	isEmpty	size
Dynamic Array	1	1	1	1	1
Doubly Linked List	1	1	1	1	1

For Queue, Dynamic Array is called a “Circular Queue”
Easiest to implement Singly (double-ended) Linked List

Priority Queue Summary

- Binary heap supports insertion and deletion of the max (min) item in logarithmic worst-case time. Uses an array, easy to implement, and elegant. Often best choice.

Operation Implementation	insert	delMax	findMax
Unordered Array	1	N	N
Ordered Array	N	1	1
Binary Heap	lg(N)	lg(N)	1
???*	1	1	1

Impossible: Lower bounds (omega) for compare sorting is N lg N. If O(1), then heap sort O(N).

Sorting Summary

- Quicksort is fastest in practice but not stable and naive implementation may result in $O(N^2)$ worse case.
- Mergesort is stable but not in place. Easy to implement and guarantees O(N lg N).
- Heap sort is in place and guarantees O(N lg N) performance. Poor cache relative to merge/quick (compares with values far apart).

Operation Implementation	worst	average	best	in place O(1)	stable	remarks
Selection Sort	N^2	N^2	N^2	yes	no	never use
Insertion Sort	N^2	N^2	N	yes	yes	small n
Merge Sort	N lg N	N lg N	N lg N	no	yes	extra memory
Quick Sort	N^2	N lg N	N lg N	yes*	no	fast practice
Heap Sort	N lg N	N lg N	N lg N	yes	no	poor cache
???	N lg N	N lg N	N lg N	yes	yes	Unknown

* Depending on variant, will assume O(lg(N)) ~ O(1)

Symbol Table Summary

- Rule of thumb: Generally use hash table unless guaranteed performance or need ordered operations

Implementation	Worse-Case		Average-Case		Order Ops	remarks
	Search	Insert	Search	Insert		
Unordered List	N	N	N	N	No	
Ordered Array	lg N	N	lg N	N	Yes	
BST	N	N	lg N	lg N	Yes	Easy
AVL	lg N	lg N	lg N	lg N	Yes	Easy
Red-Black	lg N	lg N	lg N	lg N	Yes	Often Used*
HT Chaining	N	N	N / M	N / M	No	Often Used*
HT Probing	N	N	1	1	No	

* Good constants and relatively easy to implement, used in many libraries

Graph Data Structure Summary

- Rule of thumb: When dealing with a dense graph, use an adjacency matrix. Use adjacency list when dealing with sparse graphs. When you are not sure, use an adjacency list (most applications have sparse graphs).

Data Structure Implementation	memory	add edge	v adjacent to w	iterate v's neighbors
Edge List	E	1	E	E
Adjacency Matrix	V^2	1	1	V
Adjacency List	V + E	1	degree(v)	degree(v)

Graph/Problem Algorithm Summary

Problem	Graph	Weighted Graph	Digraph	Weighted Digraph
Single-source Paths	BFS,DFS	BFS,DFS	BFS,DFS	BFS,DFS
Single-source Shortest Paths	BFS	Dijkstra*, Bellman-Ford**	BFS	Dijkstra*, Bellman-Ford**
All-pairs Shortest Paths		Dijkstra*, Floyd-Warshall**		Dijkstra*, Floyd-Warshall**
Connected Components (CC)	BFS,DFS	BFS,DFS		
Minimum Spanning Tree		Prim Kruskal		

* Applicable when edge weights are non-negative.

** Applicable when no negative cycles.

Union-Find Summary

- Typical to implement weighted union and path compression. Both heuristics are easy to implement.
  - Weighted union-find with path compression is nearly optimal as a constant time algorithm has been proven not to exist.

Algorithm	Constructor	union	find
Quick-Find	N	N	1
Quick-Union	N	Tree height	Tree height
Weighted QU	N	lg(N)	lg(N)
Weighted QU w/PC	N	~1*	~1*
Impossible	N	1	1

* Very, very , very nearly one, amortized, close to best that can be done for problem

Final Overview

- Cumulative, however heavily weighted towards material after the midterm.
  - Short answers can be drawn from entire course
  - Problems will be from after the midterm
- Problems may include
  - BST, AVL insert/rotation cases
  - Convert 2-3 Tree <-> Red-Black Tree
  - B Tree Parameter Computations
  - Huffman's Algorithm, Convert Code Table <-> Trie
  - Linear Probing, Separate Chaining, Load Factor
  - DFS/BFS, Prim, Kruskal, Dijkstra
  - Union Find, QF, QU

Free Question Time!