chapters - second half

Prof Bill - created: Dec 2016

These are my 1-page summaries of chapters in our textbook:

Data Structures and Algorithms
by Goodrich, et al
wiley.com/college/goodrich

First half, bump up maps and hash tables.
Midterm stuff:

- ➔ Ch 1, Ch 2 Java + OOP
- ➔ Ch 3, Ch 7 linked list + ArrayList + List ADT
- ➔ Ch 5 recursion
- ➔ Ch 6 stack and queue
- ➔ Ch 10 maps + hash tables

Here's the second half material that is covered in this file.
Second half:

- ➔ Ch 4 algorithm analysis
- ➔ Ch 8 trees
- ➔ Ch 9 priority queue + heaps
- ➔ Ch 14 graphs
- ➔ Ch 11 search tree

Ran out of time and didn't quite make it:

- ❖ Ch 12 search and sort

Optional

- ➔ Ch 13 text processing
- ➔ Ch 15 memory management

Missing:

- There's no GUI; use javafx, which is Ch 15 in Godfrey's book
- Streams API
- lambda exprs

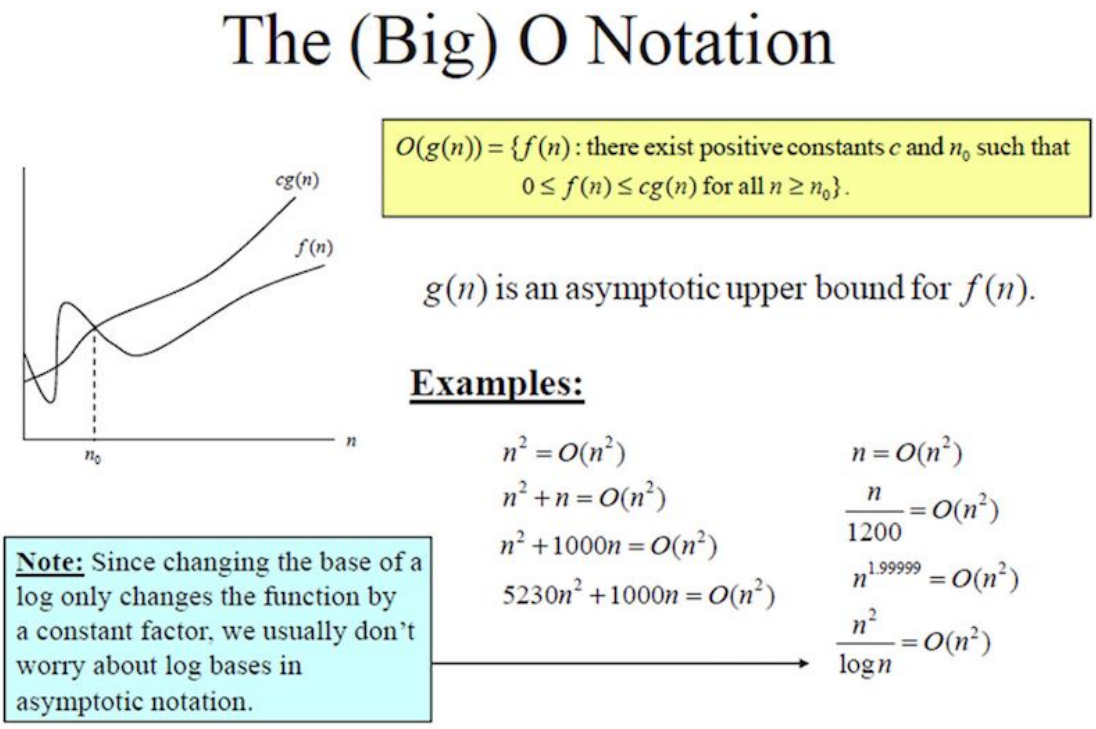
Ch 4 Algorithm Analysis

/ Big-Oh! */*

We are: “CSC*210*1 (40879) Data Structures & Algorithms”

data structure - a way of organizing and accessing data

algorithm - a step-by-step procedure for performing some task



Source: meherchilakalapudi.wordpress.com/category/data-structures-1asymptotic-analysis/

4.1 Experimental studies

Browser example: many many more websites, but no extra browser delay hash table!
Java - use **System.currentTimeMillis()** method to time algorithm.

Timing algorithms is unreliable and incomplete: 1) unreliable because of CPU load, 2) results limited to specific data trials, 3) you have to code the whole thing up to run it

➔ Answer: we need a theoretical way to measure algorithm performance!

/* String v. StringBuilder experiment - why the performance difference? */

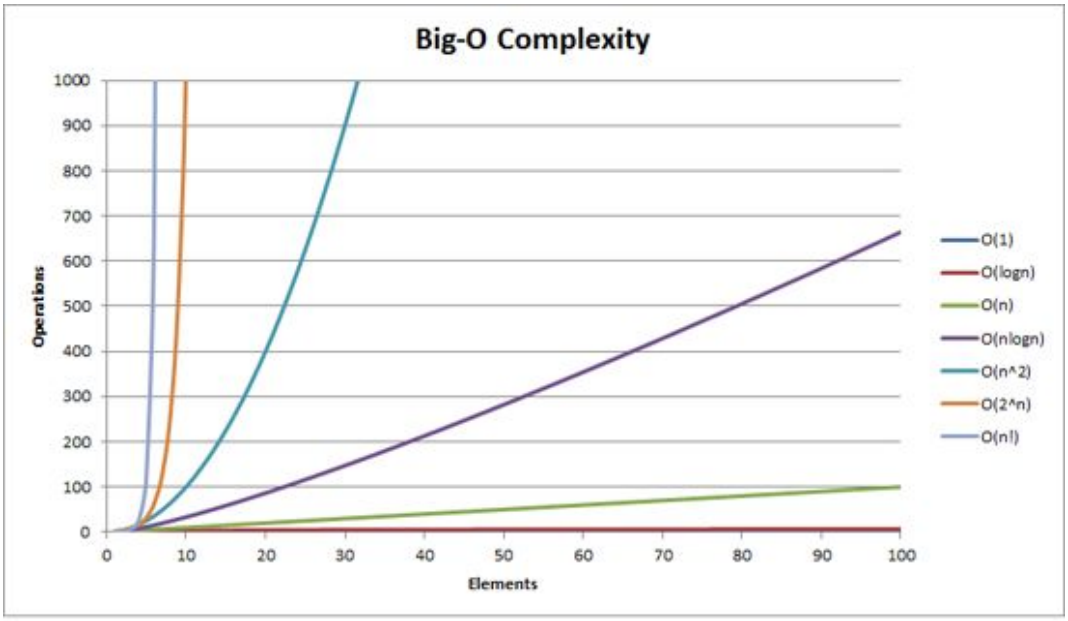
Analysis goals to estimate algorithm performance (fix experimental woes listed above):

1. independent of environment,
2. independent of specific data sets/inputs, and
3. with no coding needed

Usually focus on “worst case” performance in terms of input size = N; or “average case” but this is sometimes tough to analyze and specify

4.2 Seven functions

7 common analysis functions: constant, log, linear, n log n, quadratic, cubic, exponential
/* these are in order of how fast they grow */



Source: i.stack.imgur.com/WcBRI.png

4.3 Asymptotic analysis

Asymptotic means “as N grows to infinity”
Big-Oh! “f(n) is Big-Oh of g(n)” means:

```
f(n) <= c * g(n), for c > 0, n >= n0
```

Simplify Big-Oh to its most significant term: ignore constants, drop lower order terms
Big-Omega - grows at least as fast as g(n), an **asymptotic lower bound**, whereas Big-Oh is an asymptotic upper bound (link, link)
Big-Theta - identifies two functions that grow at same basic rate
Big-Oh is, by far, the most important one, and what we’ll focus on in class.

Example 4.7, page 165 - *5n^4 + 3n^3 + 2n^2 + 4n + 1 is O(n^4)*
Justify: 5n^4 + 3n^3 + 2n^2 + 4n + 1 <= (5+ 3 + 2 + 4 + 1) n^4 = cn^4.
Summary: Big-Oh is simplified down to the highest order polynomial.
Summary: Big-Oh eliminates all smaller-order factors: n^2 + log n is O(n^2).

Important question: Does my algorithm run in polynomial or exponential time?
This table shows why this is so critical.

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Source: www.cpp.edu/~ftang/courses/CS240/lectures/img/alg-tab.jpg

Goodrich doesn't cover P versus NP issue:

- en.wikipedia.org/wiki/P_versus_NP_problem
- danielmiessler.com/study/pvsnp/#gs.VkVM7LM
- www.ida.liu.se/opendsa/OpenDSA/Books/Everything/html/NPComplete.html

4.4 Simple Justification Techniques

Justification is more hand-waving than a rigorous mathematical proof.

My own hand-waving: Often, we are estimating search times. Array, no search = O(1); Linked list, touch each node = O(N); Binary search = halve the nodes we touch each iteration = O(log n); Bubble sorting, loop inside a loop = O(n^2)

We said that hash table is special because it runs in O(1). Remember though that is average case. Worst case is still O(N).

Code frags 4.5, 4.6 - 3 disjoint set example; one solution is O(n^3), another O(n^2)

Big-O Notation	Examples of Algorithms
O(1)	Push, Pop, Enqueue (if there is a tail reference), Dequeue, Accessing an array element
O(log(n))	Binary search
O(n)	Linear search
O(n log(n))	Heap sort, Quick sort (average), Merge sort
$O(n^2)$	Selection sort, Insertion sort, Bubble sort
$O(n^3)$	Matrix multiplication
$O(2^n)$	Towers of Hanoi

Source: images.slideplayer.com/16/5041115/slides/slide_6.jpg

Ch 8 Trees

/ the basics! */*

8.1 General Trees

Definitions:

- ❏ **tree** - an ADT that stores elements hierarchically
- ❏ **root** - the top/first node in the tree
- ❏ **parent** - the node above this node in the tree; everyone but the root has a parent
- ❏ **child** - each node has 0 or more child nodes below it
- ❏ **leaf** - a node with no children; node sits at the bottom of the tree

More (less important) definitions:

- ❖ **siblings** - two or more nodes that share the same parent
- ❖ **internal node** - has one or more children
- ❖ **external node** - has no children, aka a leaf
- ❖ **ancestor** - “node u is an ancestor of node v if u = v or u is an ancestor of v’s parent” (recursive definition!), or node is connected above this node
- ❖ **descendant** - if u is an ancestor v, then v is a descendant of u, node is connected below this node
- ❖ **subtree** - a subtree at node v include v as root and all v’s descendants
- ❖ **ordered** - tree is ordered if there is an order to the children of nodes

Tree ADT

- Each node in the tree is a “position”; for any position: getElement(); /* Position interface defined on p 274, Code Frag 7.7, just that one method */
- For the tree overall: root(), parent(p), children(p), numchildren(p)
- Util methods: size(), isEmpty(), iterator(), positions()
- Position util methods: isInternal(p), isExternal(p), isRoot(p)

In Java:

```
public interface Tree<T> implements Iterable<T> { … }
```

depth of a node/position = num ancestors of p, see recursive method at Code Frag 8.3
height of a node/position = height of a leaf is 0; height of internal node is max all children’s heights + 1 (recursive again!)

8.2 Binary Trees

Binary tree - each node has at most 2 children
full binary tree - each node has 0 or 2 children, also “proper”
Binary Tree ADT add these methods: left(p), right(p), sibling(p)

Nice height-related properties of full binary tree, h=height:

1. 2h+1 <= n <= 2^(h+1) - 1
2. h+1 <= num edges <= 2^h
3. h <= num internal nodes <= 2^h - 1
4. log(n+1) - 1 <= h <= (n-1) / 2

And… num external nodes = num internal nodes + 1

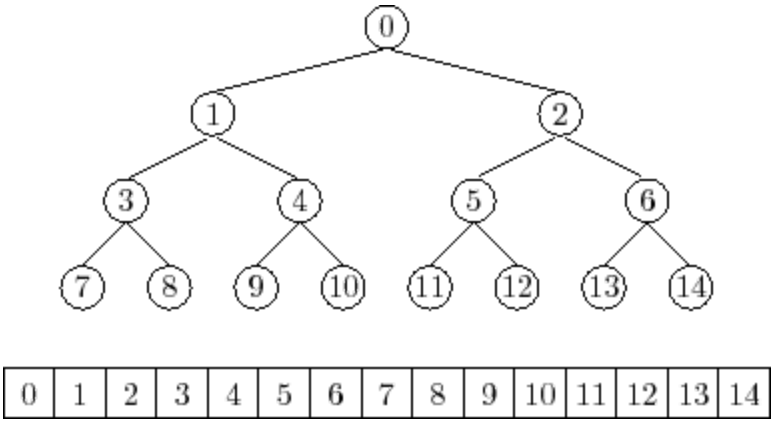
8.3 Implementing Trees

Interface (method signatures) => Abstract class (common code) => concrete class
Linked binary tree uses nodes. Binary tree node has left and right pointers.
Factory method pattern to create nodes, instead of ctor:

```
Node<E> createNode( E e, Node<E> parent, Node<E> left, Node<E> right) { … }
```

Array-based binary tree (cool!) At any array position i:

- left child(i) = 2i + 1
- right child(i) = 2i + 2
- parent(i) = (i-1) / 2



8.4 Tree Traversal

Preorder = node, left right; Inorder = left, node, right; Postorder = left, right, node.
Binary search tree = log(n) search
Breadth-first traversal - use queue

Ch 9 Priority Queue

/ Heaps of fun! */*

Priority Queue (PQ) is like a queue, but not FIFO. Order is based on some other property of the key or value in PQ. It's priority! Example: jobs waiting for CPU time; passengers waiting for airline seats, time-base events in a simulation, etc.

9.1 PQ ADT

ADT for PriorityQueue<K,V> is more like a Map than a Queue.

`Entry<K,V> insert(k,v)` - add entry to PQ
`Entry<K,V> min()` - return min entry in PQ
`Entry<K,V> removeMin()` - remove the min entry in PQ and return it
`int size()` - num entries in PQ
`boolean isEmpty()`

9.2 Implementing PQ

We need a way to order entries into PQ. Like sorting. Two ways:

- ➔ **Comparable<T>** interface - defines **compareTo(T o)** method; easy way to remember, method returns (this - o); docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
- ➔ **Comparator<T>** class - an algorithm in a box; similar method but with 2 params, **compare(T o1, To2)**, easy to remember, returns (o1 - o2); docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

There are two **Collections.sort()** methods in JCF… one for Comparable, and one for Comparator: docs.oracle.com/javase/8/docs/api/java/util/Collections.html

We know this paradigm now… interface, abstract class, class.

- ➔ Unsorted list performance: O(1) insert, O(n) min
- ➔ Sorted list performance: O(n) insert, O(1) min /* always first in list */

9.3 Heaps

This is the $$$.
Heap order property - every node is greater than its parent, except the root (of course)

- en.wikipedia.org/wiki/Binary_heap - nice Wikipedia summary

- www.cs.usfca.edu/~galles/JavascriptVisual/Heap.html - nice simulation, helpful to visualize this structure to understand it

insert(k, v) pseudo-code - add to end, bubble up

E = new entry for (k,v)
add E to end of heap
while (not root && E.parent < E)
swap E and it's parent

removeMin() pseudo-code - remove root, move last entry to root, sink down

remove root
move last entry E to root
while(E > it's children)
swap E with smallest child

Array representation is key/ubiquitous. Performance analysis: insert is O(log n); removeMin is O(log n). Rationale - heap height is <= log(n), where n = # entries in heap
/ Note - we won't cover "9.3.4 Bottom-up Heap Construction" */*
Java Collections Framework - java.util.PriorityQueue

- API: add (insert); peek (min); remove (removeMin); size; isEmpty

9.4 Sorting with PQ

EZ. Successive calls to removeMin() sorts the list.

```
S = new empty List
for i = 1 to PQ.size
   E = PQ.removeMin()
   S.addEnd( E)
```

Heap Sort - performance is O(n log n). N entries inserted at O(log n) each.
Use array. Max heap. Do remove() "in place". Example at Figure 9.8, page 389.
Nice summary at Wikipedia: https://en.wikipedia.org/wiki/Heapsort

9.5 Adaptable PQ

Add three methods to PQ ADT:

- remove(e) - not just min
- replaceKey(e, k) - change key of entry already in the heap
- replaceValue(e, v) - change value of entry already in the heap

Ch 11 Search Trees

11.1 Binary Search Trees

Featuring Guest Lecturer: ***Brandon P!***

Binary search tree property… for every node:

1. Left child is < node
2. Right child > node

ADT for BinarySearchTree<K, V> - put(k); v get(k); remove(k)
Inorder traversal of BST results in sorted list.

Pseudo-code for (recursive) **search**:

```
TreeSearch( Node n, Key k)
    if k == key(n), then return n
    else if k < key(n), then return TreeSearch( left(n), k)
    else return TreeSearch( right(n), k)
```

Pseudo-code for **insert**:

```
TreeInsert( Node n, Key k)
    if n == null, then new node is root, return // empty
BST
    if k == key(n), then return   // key already in BST
    else if k < key(n)   // less than means go left
        if left(n) == null, then add new node as left child
        else TreeInsert( left(n), k)
    else    // must be greater than, go right
        if( right(n) == null, then add node as left child
        else TreeInsert( right(n), k)
```

Performance of search and insert: O(h), where h is height of the tree.
Performance is **not** O(log n) like a heap... because tree is not full, so longest path in BST can be much longer than log n.
This lagging performance is the rationale for the rest of Chapter 11… balanced trees.
/* We'll worry about deletion later! */

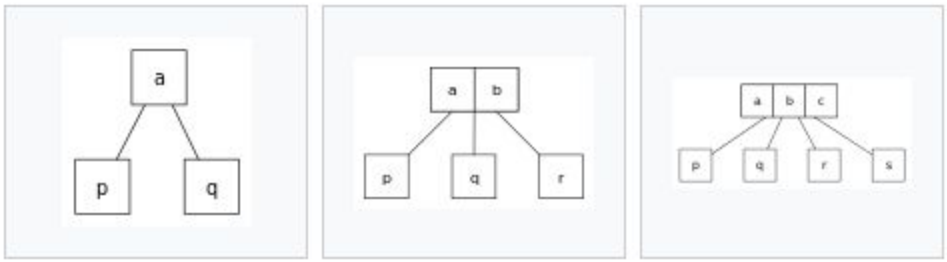
Nice animation: www.cs.usfca.edu/~galles/visualization/BST.html

11.5 (2,4) Trees

Featuring Guest Lecturer: ***Brett K!***

Also called 2,3,4 Trees: en.wikipedia.org/wiki/2%E2%80%933%E2%80%934_tree
General (n,m) Trees are covered in section 15.3 of our book.
These are also called B-Trees: https://en.wikipedia.org/wiki/B-tree

Goal: O(log n) search by guaranteeing height of your tree is <= log(num nodes)
Three types of nodes:



2-node 3-node 4-node

Values within each node are in sorted order.
Children follow the BST property: left < node, right >node
So for the nodes above:

- 2-node: p < a, q > a
- 3-node: p < a, a < q < b, r > b
- 4-node: p < a, a < q < b, b < r < c, s > c

Insert

Always insert at leaf. If 4 values in node, then split node into 2 nodes and bubble up inner value. Repeat above if 4 values in node.

Search

```
Search( Tree, key)
    For node v, compare the key with the keys k1,k2,k3 stored at v.
    If k is found then FOUND.
    Otherwise return to step 1 using the child vi such that ki-1 <= ki.
    If ki child is null then NOT FOUND
```

/* We'll worry about deletion later! */
Animation... set max degree = 4: www.cs.usfca.edu/~galles/visualization/BTree.html

11.2 Balanced Trees

Binary Search Tree (BST) can get out of balance. This is why BST is **not** O(log n).

Solution: When tree starts to get out of whack, rebalance by rotating tree nodes.
(2,4) Tree is (sort of) an example of this, but it's not a binary tree.
Chapter 11 has 3 examples of binary trees that use rotation to stay balanced.

Section 11.3 AVL Trees
Use rotations to maintain **Height-Balance Property**:

> *For every internal position p of T, the heights of children of p differ at most by 1*

Animation: www.cs.usfca.edu/~galles/visualization/AVLtree.html

Section 11.4 Splay Trees
Splay tree definition

> *A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.*
>
> - en.wikipedia.org/wiki/Splay_tree

So, recently accessed nodes are closer to the top and found quicker the next time.
splay - given a node x of tree T, we splay x by moving it to the root of T.

Animation: www.cs.usfca.edu/~galles/visualization/SplayTree.html

Section 11.6 Red-Black Trees
Binary tree where nodes in the tree are colored red or black.
Rotation follows these properties:

- ➔ Root property - The root is black.
- ➔ External property - Every external node is black.
- ➔ Red property - The children of a red node are black.
- ➔ Depth property - All external nodes have the same **black depth**, defined as the number of proper ancestors that are black.

Animation: www.cs.usfca.edu/~galles/visualization/RedBlack.html

Ch 14 Graphs

/ Vertices + Edges = Excellent! */*

Graph G =

1. V, set of **vertices**, and
2. E, collection of **edges**, that connect pairs of verts

2 flavors of graphs: **directed** (digraph) and **undirected** (edges)
Examples: computer circuits (directed), 6 degrees of Kevin Bacon or Erdos number (undirected), airline flight info (directed), servers on the internet (undirected)

Definitions galore…
two vertices are **adjacent** if they are connected by an edge
an edge is **incident** to a vertex if the vertex is one of its endpoints
the **degree** of a vertex is its number of incident edges

path - a sequence of vertex-edge that starts at one vertex and ends at another
cycle - path that starts and ends at the same vertex
simple path/cycle - each vertex appears only once
a directed graph is **acyclic** iff it has no cycles, then it's a **"dag"**
reachable - u reaches v if there is a path between the two vertices
a graph is **connected** if there is a path between any two vertices
subgraph of G is a graph whose vertices and edges are a subset of G
spanning subgraph is a subgraph that includes all vertices
a **forest** is a graph without cycles
a **tree** is a connected forest, a connected graph w/out cycles
spanning tree is a spanning subgraph without cycles

Graph ADT… generic and for un- and directed graphs (section 14.1.1, page 618)

- Info: numVertices(); vertices(); numEdges(); edges()
- Creation: insertVertex(x); insertEdge(u, v, x)
- Removal: removeVertex(v); removeEdge(e)
- Connectivity: getEdge(u, v); endVertices(e); opposite(v, e)
- Vertex info: outDegree(v); inDegree(v); outgoingEdges(v); incomingEdges(v)

/* We'll have our own Graph210 interface, which is a simpler, non-generic, undirected graph with weighted edges */

14.2 Data Structures

Data structures:

- **Edge List:** unordered list of edges
- **Adjacency List:** add to Edge List, vertex list with unordered list of their edges
- **Adjacency Map:** Same as list above, but use map: Edge get(Vertex v)
- **Adjacency Matrix:** NxN matrix, where N= # vertices, stored edges in matrix

Most common: Adj List, unless you know graph is very dense, then use Adj Matrix.
Show each structure for this graph:

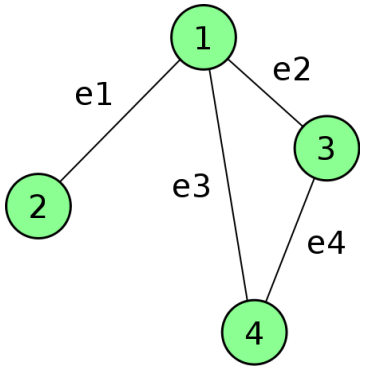


Table 14.1, page 619… Big-Oh performance of Graph ADT for various data structures.

of vertices u and v; if no such edge exists, the slot will store null.

A summary of the performance of these structures is given in Table 14.1.

Method	Edge List	Adj. List	Adj. Map	Adj. Matrix
numVertices( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
numEdges( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
outDegree(v) inDegree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
outgoingEdges(v) incomingEdges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

for the methods of the graph ADT, us-

Ch 14 Graphs, cont.

14.3 Graph traversals

graph traversal - systematically visit all vertices and edges of a graph
Useful for determining if graph is connected, or detecting cycles

Depth-First Search (DFS)
DFS pseudo-code:

```
DFS( G, v)
    mark v visited
    for each vertex u adjacent to v
        if u is not visited, then DFS( G, u)
```

Very nice animation: www.cs.usfca.edu/~galles/visualization/DFS.html
Performance: O(V + E)

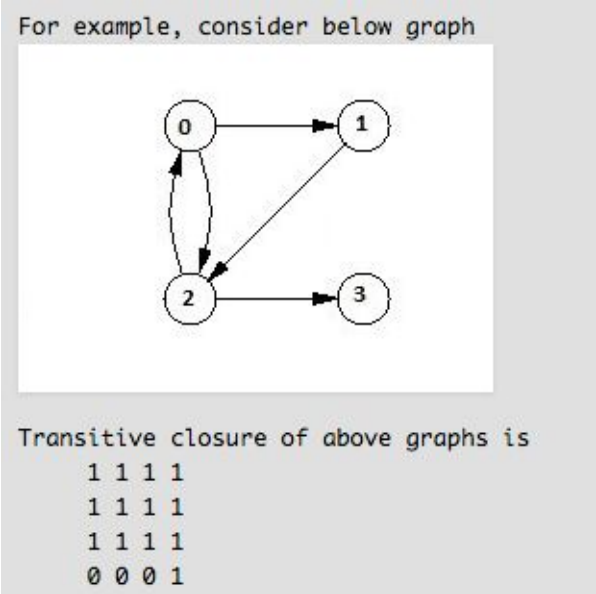
Breadth-First Search (BFS)
BFS pseudo-code:

```
BFS( G, v)
    knownSet.add(v)  // add v to known set
    Q.enqueue(v)   // add v to search queue
    while Q is not empty
        u = Q.dequeue()
        for each vertex w adjacent to u
            if ! knownSet.contains(w), then
                knownSet.add(w)
                Q.enqueue(w)
```

Animation: www.cs.usfca.edu/~galles/visualization/BFS.html
Performance: O(V + E)

14.4 Transitive Closure

For a directed graph, is there a path from each pair of vertices? **Reachable**.
Nice summary here: www.geeksforgeeks.org/transitive-closure-of-a-graph/
Reachability matrix: 1 if reachable from vertex, 0 if not



One (easy) solution: Depth-first search from each vertex, track reachability each time

Floyd-Warshall algorithm - grow graph, adding edges to show transitive closure

- if edges (u, v) and (v, w), then add (u, w) if not already there
- do this #vertices times

Tally weights during Floyd-Warshall to get shortest paths between all vertex pairs.
en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Animation: www.cs.usfca.edu/~galles/visualization/Floyd.html

14.5 DAGS

Directed Acyclic Graph = DAG

Topological ordering - number edges in any path in increasing order, this numbering is not unique, DAG only!

Pseudo-code:

```
TopologicalSort( G)
    // init in-degree array
    create array, indegree[v] = 0
    for each edge (u,v)
      increment indegree[v]

    // load stack with 0 in-degree vertices
    create vertex stack VStack
    for each vertex v
      if indegree[v] == 0 then VStack.push(v)

    // process vertices on stack
    create vertex list VList
    while not VStack.isEmpty() {
      v = VStack.pop()
      VList.add( v)   // add to end of list
      for all outgoing edges (v, w)
        decrement indegree[w]
        if indegree[w] == 0 then S.push( w)
    }
    // done - order of verts in VList is topological ordering
```

Animation: www.cs.usfca.edu/~galles/visualization/TopoSortIndegree.html

14.6 Shortest Paths

weighted graph - each edge has a weight (duh)
shortest path - min weight sum of edges between two vertices
distance array - d[v] = distance from start vertex to v, equals inf if no path

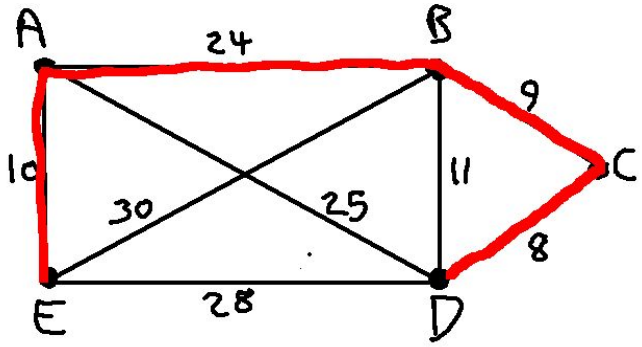
Dikstra's Algorithm - similar to Prim's; grow path from start node and then tree

```
Dijkstra( G, start)
  init D distance array: D[v] = inf; D[start] = 0
  init known array: known[v] = false
  init path array: path[v] = -1
  create PQ priority queue: add all D[v] to PQ

  while not PQ.isEmpty()
    u = PQ.removeMin()   // process min distance vertex, D[u]
    known[u] = true

    for each edge (u, v)
      if known[v] is false
        if D[u] + weight(u, v) < D[v] {
          // update v with new, shorter distance
          D[v] = D[u] + weight(u, v)
          path[v] = u
          change D[v] key in PQ   // prob delete and re-insert
        }
```

Animation: www.cs.usfca.edu/~galles/visualization/Dijkstra.html

14.7 Minimum Spanning Trees

spanning tree - a tree that contains every vertex in a connected graph (remember - tree means no cycles!); it's a list of edges
min spanning tree - the spanning tree where the sum of edge weights is smallest



Prim's Algorithm
In English: Pick a vertex to be root of the tree. Find the min weight edge connected to the tree. Add that edge's vertex. Repeat until all vertices are in the tree.
Similar to Dijkstra's Algorithm for finding shortest path.
Animation: www.cs.usfca.edu/~galles/visualization/Prim.html

Pseudo-code:

```
PrimsMST( G, startv)
      create distance array, D[#vertices] = inf
      create parent array, parent[#vertices] = -1
      create known array, known[#vertices] = false

      D[startv] = 0   // start vertex is tree root
      add each D[i] to PriorityQueue PQ
      while PQ not empty
            u = PQ.removeMin()
            known[u] = true
            for each edge connected to u, (u, v)
                  if ! known[v]  and weight of edge < D[v]
                        D[v] = weight of edge
                        parent[v] = u
                        change D[v] key in PQ   // remove, and re-add to PQ
      for each vertex, v
            add edge ( v, parent[v]) to MST list
```

Kruskal's Algorithm

In English:

sort the edges by weight
for each edge
add edge to MST if it doesn't create a cycle

Animation: www.cs.usfca.edu/~galles/visualization/Kruskal.html

Pseudo-code:

```
KruskalsMST( G)
      place each vertex in its own disjoint set
      sort all edges in G by weight
      for each edge (u,v)
            ds1 = find disjoint set of u
            ds2 = find disjoint set of v
            if ds1 != ds2
                  add edge to MST list
                  union( ds1, ds2)   // merge 2 disjoint sets into 1
```

What's are disjoint sets? What is find()? Union?

Answer: Disjoint set is a collection of sets whose members don't intersect.

We use a nifty representation of disjoint sets (an array) to efficiently determine if adding edge would create a cycle. A lot of people use disjoint sets, eh...

en.wikipedia.org/wiki/Disjoint-set_data_structure