Computational Optimization
ISE 407

Lecture 17

Dr. Ted Ralphs

References for Today's Lecture

- Sections 17.2–17.5, R. Sedgewick, *Algorithms in C++, Part 5*.
- AMO Sections 2.3
- CLRS Section 22.1

Connectivity Relations

- So far, we have only considered sets of items that are related to each other through some kind of ordering (if at all).
- In other words, two items x and y are only related by their relative positions in the ordered list.
- We will now generalize this idea by considering additional *connectivity relationships* between items.
- To do so, we will specify that there is a direct link between certain pairs of items.
- This will allow us to ask questions such as the following.
  - Is x connected "directly" to y?
  - Is x connected to y "indirectly," i.e., through a sequence of direct connections?
  - What is the set of of all items connected to x, directly or indirectly?
  - What is the shortest number of connections needed to get from x to y?

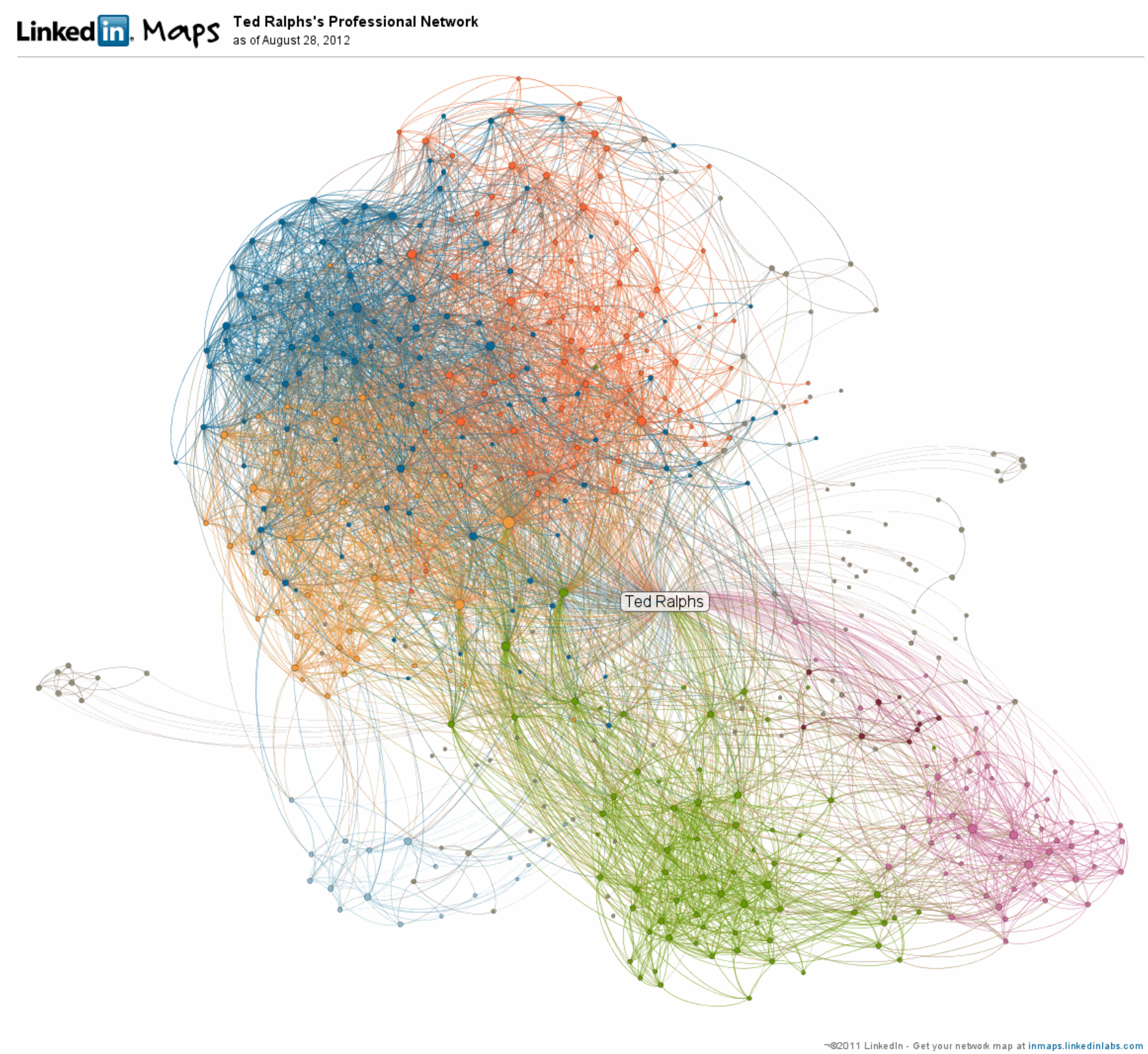
Graphs

- A *graph* is an abstract object used to model such connectivity relations.
- A *graph* consists of a list of items, along with a set of connections between the items.
- The study of such graphs and their properties, called *graph theory*, is hundreds of years old.
- Graphs can be visualized easily by creating a physical manifestation.
- There are several variations on this theme.
  - The connections in the graph may or may not have an *orientation* or a *direction*.
  - We may not allow more than one connection between a pair of items.
  - We may not allow an item to be connected to itself.

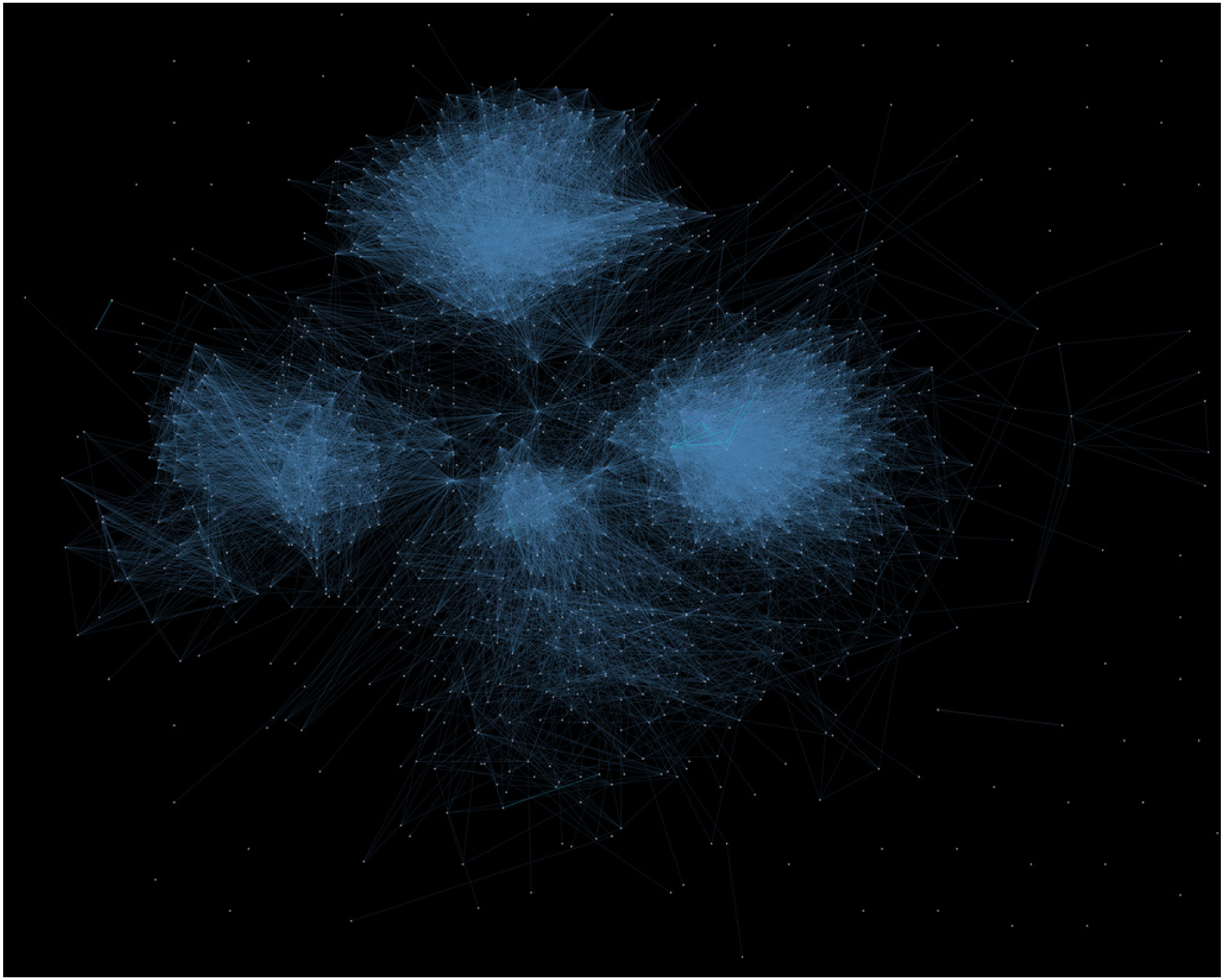
Applications of Graphs

- Maps
- Social Networks
- World Wide Web
- Circuits
- Scheduling
- Communication Networks
- Matching and Assignment

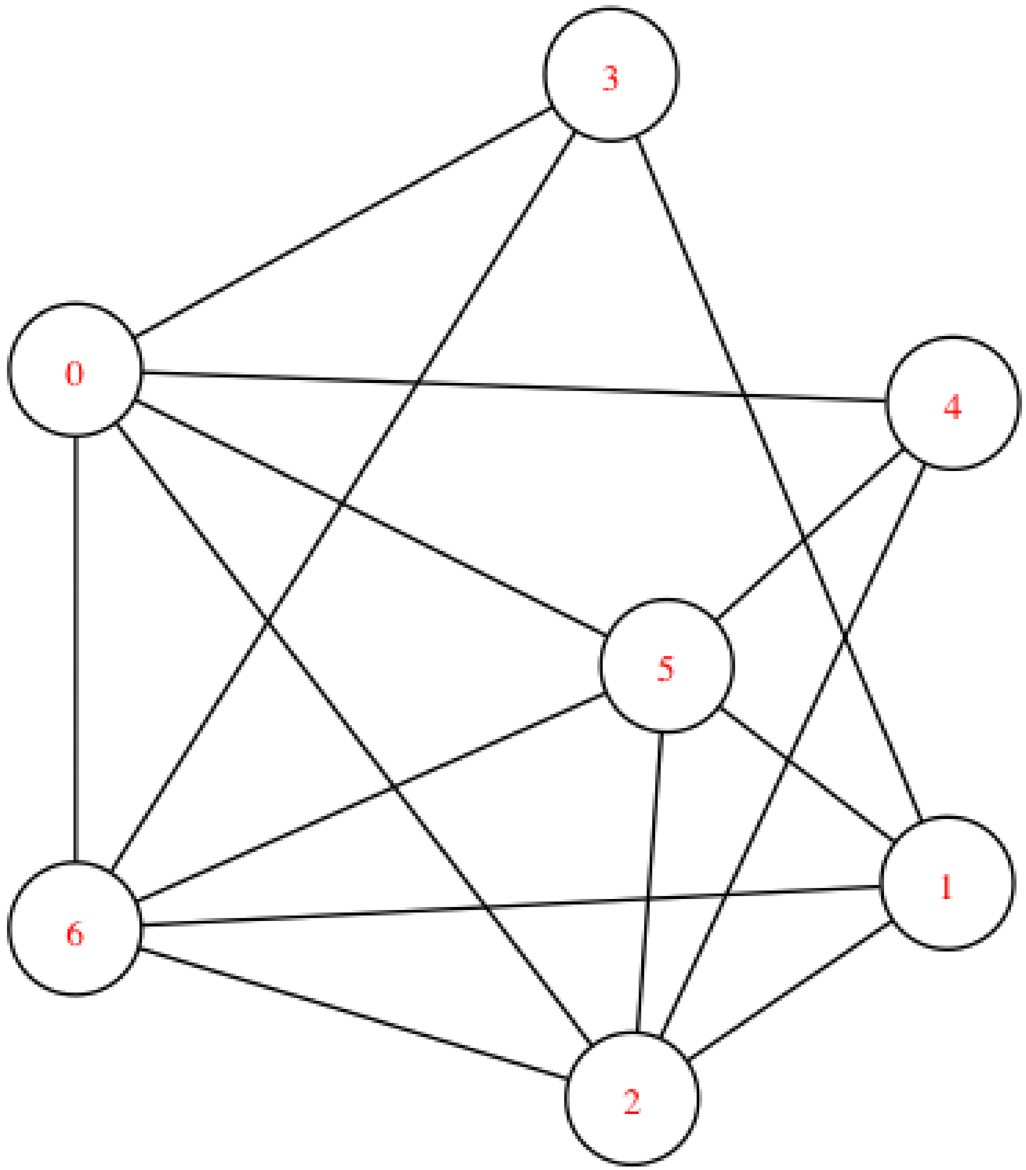
Example Graph (Social Network)



A Facebook Graph



Example of an Abstract Graph



Undirected Graphs: Terminology and Notation

- In an *undirected graph*, the "items" are usually called *vertices* (sometimes also called *nodes*).
- The set of vertices is denoted V and the vertices are indexed from 0 to $n-1$, where $n = |V|$.
- The connections between the vertices are unordered pairs called *edges*.
- The set of edges is denoted E and $m = |E| \leq n(n-1)/2$.
- An undirected graph $G = (V, E)$ is then composed of a set of vertices V and a set of edges $E \subseteq V \times V$.
- If $e = \{i, j\} \in E$, then
  - i and j are called the *endpoints* of e,
  - e is said to be *incident* to i and j, and
  - i and j are said to be *adjacent* vertices and are also called *neighbors*.

Directed Graphs: Terminology and Notation

- In a *directed graph*, the “items” are traditionaally called *nodes*.
- The set of nodes is denoted N and are indexed from 0 to $n-1$, where $n = |N|$.
- The connections between the nodes are ordered pairs called *arcs*.
- The set of arcs is denoted A and $m = |A| \leq n(n-1)$.
- A directed graph $G = (N, A)$ is then composed of a set of nodes N and a set of arcs $A \subseteq N \times N$.
- If $a = \{i, j\} \in A$, then
  - i is the *tail* of a and j is the *head*.
  - a is said to be *incident from* i and *to* j,
  - i and j are said to be *adjacent* nodes, and
  - j is an *out-neighbor* or i and i is an *in-neightbor* of j.

More Terminology

- Let $G = (V, E)$ be an undirected graph.
- A *subgraph* of G is a graph composed of an edge set $E' \subseteq E$ along with all incident vertices.
- A subset V' of V, along with all incident edges is called an *induced subgraph*.
- A *path* in G is a sequence of vertices such that each vertex is adjacent to the vertex preceding it in the sequence.
- A path is *simple* if no vertex occurs more than once in the sequence.
- A *cycle* is a path that is simple except that the first and last vertices are the same.
- A *tour* is a cycle that includes all the vertices.
- Similar concepts exist for directed graphs.

Network Representation

- Our goal is to develop "efficient" algorithms $\rightarrow$ reasonable computation time.
- The main factors affecting efficiency are
  - The underlying algorithm
  - Data structure for storing the network
- The same algorithm may behave much differently with different graph data structure.
- What information do we need to store?
  - network topology (structure of nodes and arcs)
  - associated data (costs, capacities, supplies/demands)
- What are the important operations we might need to perform with a network data structure?

Data Structures

- We first consider the general case of a directed graph.
- Common data structures
  - Node-Arc Incidence Matrix
  - Node-Node Adjacency Matrix
  - Adjacency List
  - Forward Star (Reverse Star)

(Node-Arc) Incidence Matrix

- $n \times m$ matrix denoted $\mathcal{N}$.
- One row for each node and one column for each arc.
- For each arc (i, j), put $+1$ in row i and -1 in row j.

	$(1,2)$	$(1,3)$	$(2,3)$	$(2,4)$	$(3,2)$	$(3,4)$	$(3,5)$	$(4,5)$
1								
2								
3								
4								
5								

(Node-Arc) Incidence Matrix

- What is the size of the matrix?
- How many entries are non-zero?
- What information do we get by reading across a row?
- Is this a space efficient representation?
- How about other operations?

(Node-Node) Adjacency Matrix

- $n \times n$ matrix denoted $\mathcal{H}$
- one row for each node and one column for each node
- entry $h_{ij} = 1$ if arc $(i, j) \in A$ (0 otherwise)

	1	2	3	4	5
1					
2					
3					
4					
5					

(Node-Node) Adjacency Matrix

- What is the size of the matrix?
- How many entries are non-zero?
- What data structures might we use to store arc costs and capacities?
- Is this a space efficient representation?
- What operations are most efficient with this data structure?

Adjacency List

- The adjacency list of node i, $A(i)$, is a list of the nodes j for which $(i, j) \in A$
- Textbook approach is to store $A(i)$ as a *linked list*, which allows efficient addition and deletion, in principle.
- This results in one linked list of length $|A(i)|$ for each node.
- The overall graph is stored as an array of these linked lists.
- The node data structure of the linked list can be used store additional fields, such as arc cost and capacity.
- Is this a space efficient representation?
- What operations are most efficient with this data structure?

Aside: Adjacency List Implementations

- There are many, different subtle variants on the textbook method described on the previous slide.
- Which one is the most appropriate depends on how the data structure will be used.
  - Is the set of nodes fixed or might nodes come and go?
  - Are the nodes identified by keys that we want to be able to look up?
  - Is the set of arcs fixed or might arcs come and go?
  - What auxiliary data must be stored?
- In many cases, storing the list of nodes or the list of edges as dictionaries makes sense.
- In others, it may be better to use a list data structure (dynamic arrays).
- It may or may not make sense to have a separate data structure for storing auxiliary data (recall "struct of arrays" versus array of structs")

Forward Star

- Stores node adjacency list of each node in one large array
- Associates a unique sequence number with each arc using a specific order starting with arcs outgoing from node 1, then node 2, etc.
- Stores tail information about each arc in **tail** array, head information in **head** array, etc.
- Maintains a pointer for each node that indicates the smallest numbered arc in the arc list for that node.
- What are the advantages of this representation?

Reverse Star

- Similar to a forward start except that arcs are sequenced starting with arcs incoming from node 1.
- The two representations can be maintained side-by-side if necessary.

Miscellaneous Issues

- Parallel Arcs
  - Why would we need parallel arcs?
  - Which representation(s) could accommodate them?
- Undirected Network
  - What needs to change?
    - Node-Arc Incidence Matrix
    - Node-Node Adjacency Matrix
    - Adjacency List

Summary of Representations

Representation	Storage Space	Features
Incidence Matrix	nm	1. Space inefficient 2. Expensive to manipulate 3. MCFP constraint matrix
Adjacency Matrix	kn^2	1. Suited to dense networks 2. Easy to implement
Adjacency List	$k_1n + k_2m$	1. Space efficient 2. Efficient to manipulate 3. Suited to dense and sparse
Forward Star	$k_3n + k_4m$	1. Space efficient 2. Efficient to manipulate 3. Suited to dense and spare

Table 1: From Ahuja et al. Figure 2.25

Basic Graph Interface in Python

```
class Graph:
    def __init__:
        self.nodes = {}
        self.edges = {}

    def add_node(v)
    def add_edge(v, w)
    def get_node_list()
```

Node Class

```
class Node:
    def __init__(self, name):
        self.name = i
        self.neighbors = {}

    def get_neighbors(self):
        return self.neighbors
```

A Client Function for Printing a Graph

- Here's an example of a standard way in which the graph interface class is used.
- Here, we print out a graph by enumerating all the edges incident to each vertex.

```
def print(G):
    for n in G.get_node_list():
        print n, ":",
        for i in n.get_neighbors():
            print i
        print
```

Trees

- A *tree* is a set of items organized into a hierarchical structure (think of a family tree).
- We can think of this as a special case of a graph, and so we call the items *nodes*.
- Each node has a single designated *parent* and one or more *children*.
- There is a single designated node, called the *root*, with no parent.
- Any node with no children is called a *leaf*.
- Any node with children is called *internal*.
- A tree in which all nodes have 2 or fewer children is called a *binary tree*.
- Storing a list of items in a tree structure allows us to represent additional relationships among the items in the list.
- Trees occur naturally in a wide variety of applications.

Trees in Action

- File system
- Philogenic Trees
- Family Trees
- Call Trees
- Web page

Additional Terminology

- The *level* of a node in the tree is the number of recursive calls to `parent()` needed to reach the root.
- The *depth* of the tree is the maximum level of any of its nodes.
- A *balanced tree* is one in which all leaves are at levels k or $k-1$, where k is the depth of the tree.
- Additional terms
  - Edge
  - Path
  - Siblings
  - Subtree

Tree Data Structures

- The tree ADT can be thought of as a list ADT with additional structure.
- One of the most important roles of the additional structure is to allow for the list to be traversed easily in various orders.
- We may also want to be able to be able to query the relationships of a given node to others (parent/sibling/child).

Tree ADT

```python
class Tree:
    def __init__(self, root):
        self.root = root
    def add_node(self, key, data, parent)
    def get_children(self, key) # return list of children
    def get_parent(self, key)
    def traverse(self, order) # print nodes in order
    def __contains__(self, key)
    def __iter__(self) # iterate over nodes in order
```

Additional Functionality

- Later, we'll want to be able to "splice" nodes into the tree at particular places.
- We'll also want to be able to do certain "rotations" in which we change the parent/child relationships in a systematic way.
- The goal of these operations will be to maintain a certain structure in the tree.
- This will make certain kinds of additions, deletions, and traversals efficient so we can implement additional operations.

Iterating Over the Nodes in a Tree

- Iterating over the nodes of a tree consists of visiting the nodes in a specified order, starting at the root node.
- The methods we consider here can be implemented using the standard API, i.e., we "discover" the nodes one by one as neighbors of previously discovered node.
- As we encounter each node, we put all of its children on the list to be visited.
- The order in which we take nodes off this list determined the *search order*.
  - Depth-first: Last in, first out. This means that we visit the node in the list at the deepest level first.
  - Breadth-first: First in, first out. This means we visit the node in the list at the shallowest level first.

Iterating in Depth-First Order (Recursive)

Here is a recursive implementation of a depth-first search.

```
def dfs_r(self, root):
    for i in self.get_children(root):
        print i
        self.dfs_r(i)

def dfs(self)
    print self.root
    self.dfs_r(root)
```

Iterating in Depth-First Order (Recursive)

We can also do depth-first search with a stack

```
def dfs(self):
    s = Stack()
    s.push(self.root)
    while s.isEmpty() != True:
        current = s.pop()
        print current
        for i in self.get_children(current):
            s.push(i)
```

Iterating in Breadth-First Order

To get breadth first search, we can simply replace the stack with a queue:

```
def dfs(self):
    s = Queue()
    s.enqueue(self.root)
    while s.isEmpty() != True:
        current = s.dequeue()
        print current
        for i in self.get_children(current):
            s.enqueue(i)
```

Binary Trees

- In many applications, the trees that arise are binary by nature.
- The call tree in quicksort or mergesort is an example.
- When we know that there will be at most two children of a given node, we call them the *right* and *left* children.
- We can specialize the ADT by adding methods to access the right and left children directly.
  - `get_parent(index)`: return the parent of node `index`.
  - `get_right(index)`: return the "right" child of node `index`.
  - `get_left(index)`: return the "left" child of node `index`.

Binary Tree ADT

```
class BinaryTree(Tree):

    def get_left(self, index):
        get_children(index)[0]

    def get_right(self, index):
        get_children(index)[1]
```

Iterating Over the Nodes of a Binary Tree (Pre-order)

When doing a depth first search, if we print each node before searching either of the children recursively, this produces an "pre-order" traversal.

```
def depth(self, root):
    print root
    self.depth(self.get_left(root))
    self.depth(self.get_right(root))
```

Iterating Over the Nodes of a Binary Tree (In-order)

Alternatively, if we print each node in between searching the left and right subtrees, this produces an "in-order" traversal.

```
def depth(self, root):
    self.depth(self.get_left(root))
    print root
    self.depth(self.get_right(root))
```

Iterating Over the Nodes of a Binary Tree (Post-order)

Finally, if we print each node after searching both the left and right subtrees, this produces a "post-order" traversal.

```
def depth(self, root):
    self.depth(self.get_left(root))
    self.depth(self.get_right(root))
    print root
```

Running Time of Iterating Nodes

- What is the running time of these methods of iterating over the nodes?