CS 600.226: Data Structures

Michael Schatz

Oct 3 2018

Lecture 15. Graphs

Agenda

1. ***Questions on HW4***

2. ***Recap on Trees***

3. ***Graphs***

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Assignment 4: Stacking Queues

Out on: September 28, 2018
Due by: October 5, 2018 before 10:00 pm
Collaboration: None
Grading:

Packaging 10%,
Style 10% (where applicable),
Testing 10% (where applicable),
Performance 10% (where applicable),
Functionality 60% (where applicable)

Overview
The fourth assignment is mostly about stacks and dequeues. For the former you'll build a simple calculator application, for the latter you'll implement the data structure in a way that satisfies certain performance characteristics (in addition to the usual correctness properties).

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 1: Calculating Stacks (50%)

Your first task is to implement a basic RPN calculator that supports integer operands like 1, 64738, and -42 as well as the (binary) integer operators +, -, *, /, and %. Your program should be called `Calc` and work as follows:

- You create an empty `Stack` to hold intermediate results and then repeatedly accept input from the user. It doesn't matter whether you use the `ArrayStack` or the `ListStack` we provide, what does matter is that those specific types appear only once in your program.
- If the user enters a ***valid integer***, you ***push*** that integer onto the stack.
- If the user enters a ***valid operator***, you ***pop*** two integers off the stack, ***perform*** the requested operation, and ***push*** the result back onto the stack.
- If the user enters the symbol `?` (that's a question mark), you ***print*** the current state of the stack using its `toString` method followed by a new line.
- If the user enters the symbol `.` (that's a dot or full-stop), you ***pop*** the top element off the stack and ***print*** it (only the top element, not the entire stack) followed by a new line.
- If the user enters the symbol `!` (that's an exclamation mark or bang), you exit the program.

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

```
$ java Calc
?
[]
10
?
[10]
20 30
?
[30, 20, 10]
*
?
[600, 10]
+
?
[610]
.
610
!
$
```

```
$ java Calc
? 10 ? 20 30 ? *
? + ? . !
[]
[10]
[30, 20, 10]
[600, 10]
[610]
610
$
```

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 2: Hacking Growable Dequeues (50%)

Your second task is to implement a generic `ArrayDequeue` class as outlined in lecture. As is to be expected, `ArrayDequeue` must implement the `Dequeue` interface we provided on github.

- Your implementation must be done in terms of an array that grows by doubling as needed. It's up to you whether you want to use a basic Java array or the SimpleArray class you know and love; just in case you prefer the latter, we've once again included it on the github directory for this assignment. Your initial array must have a length of one slot only! (Trust us, that's going to make debugging the "doubling" part a lot easier.)
- Your implementation must support all `Dequeue` operations except insertion in (worst-case) constant time; insertion can take longer every now and then (when you need to grow the array), but overall all insertion operations must be constant amortized time as discussed in lecture.
- You should provide a `toString` method in addition to the methods required by the `Dequeue` interface. A new dequeue into which 1, 2, and 3 were inserted using `insertBack()` should print as [1, 2, 3] while an empty dequeue should print as []

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Bonus Problem (5 pts)

Develop an **algebraic specification** for the **abstract data type Queue**. Use `new`, `empty`, `enqueue`, `dequeue`, and `front` (with the meaning of each as discussed in lecture) as your set of operations. Consider unbounded queues only.

The difficulty is going to be modelling the FIFO (first-in-first-out) behavior accurately. You'll probably need at least one axiom with a case distinction using an if expression; the syntax for this in the Array specification for example.

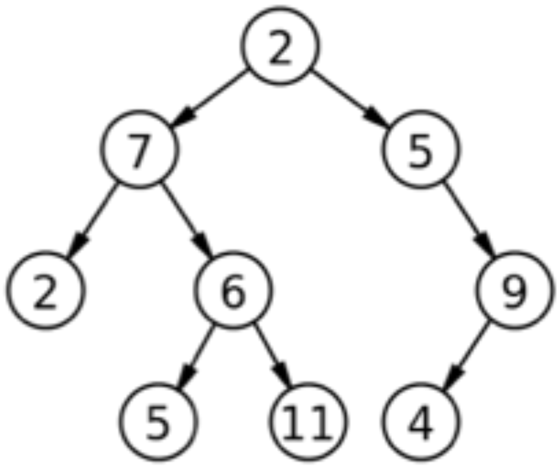
Doing this problem without resorting to Google may be rather helpful for the upcoming midterm. There's no need to submit the problem, but you can submit it if you wish; just include it at the end of your README file.

Agenda

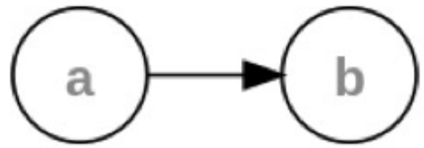
1. ***Questions on HW4***
2. ***Recap on Trees***
3. ***Graphs***

Trees

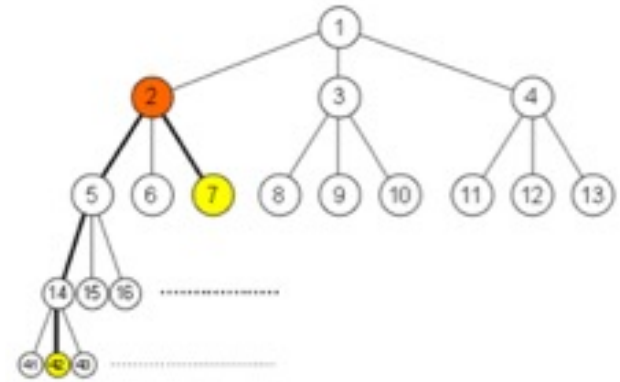
Types of Trees



Unordered
Binary tree



Linear
List

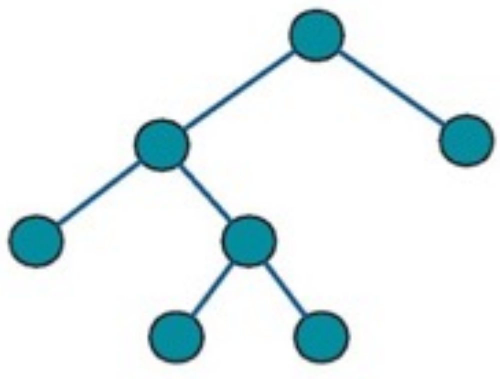


3-ary Tree
(k-ary tree has k children)

Single root node (no parent)
Each ***non-root*** node has at most 1 parent
Node may have 0 or more children

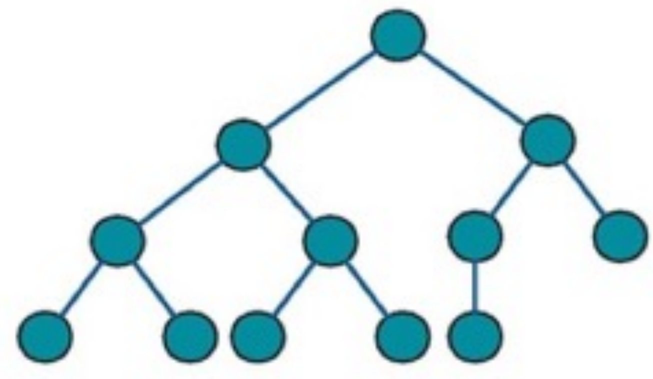
Internal node: has children; includes root unless tree is just root
Leaf node (aka external node): no children

Special Trees



Height of root = 0

Total Height = 3

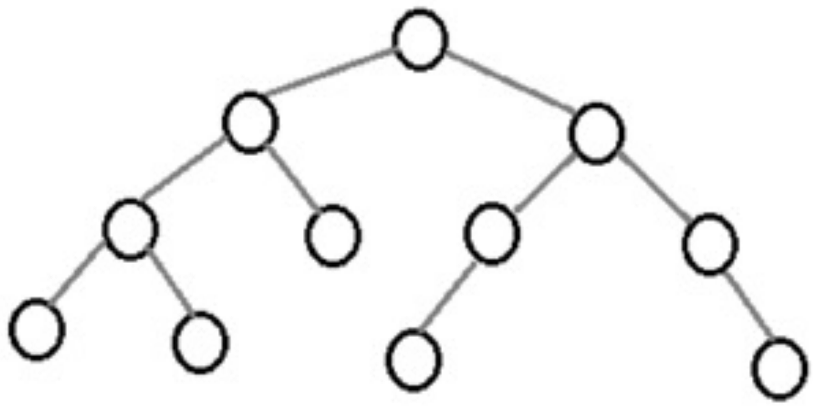


Full Binary Tree
Every node has 0 or 2 children

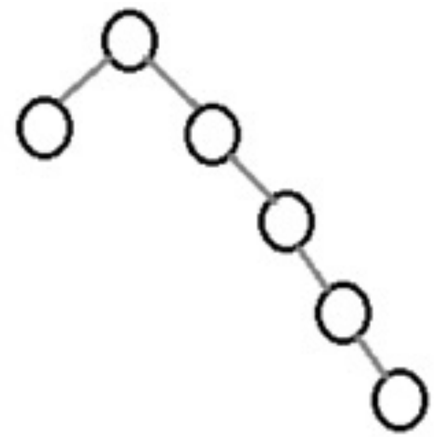
Complete Binary Tree
Every level full, except potentially the bottom level

What is the maximum number of leaf nodes in a complete binary tree?	2^h
What is the maximum number of nodes in a complete binary tree?	$2^{h+1}-1$
What fraction of the nodes in a complete binary tree are leaves?	about half

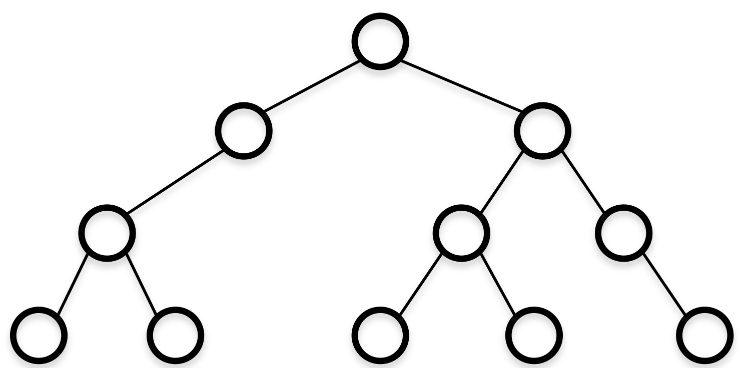
Balancing Trees



Balanced Binary Tree
Minimum possible height

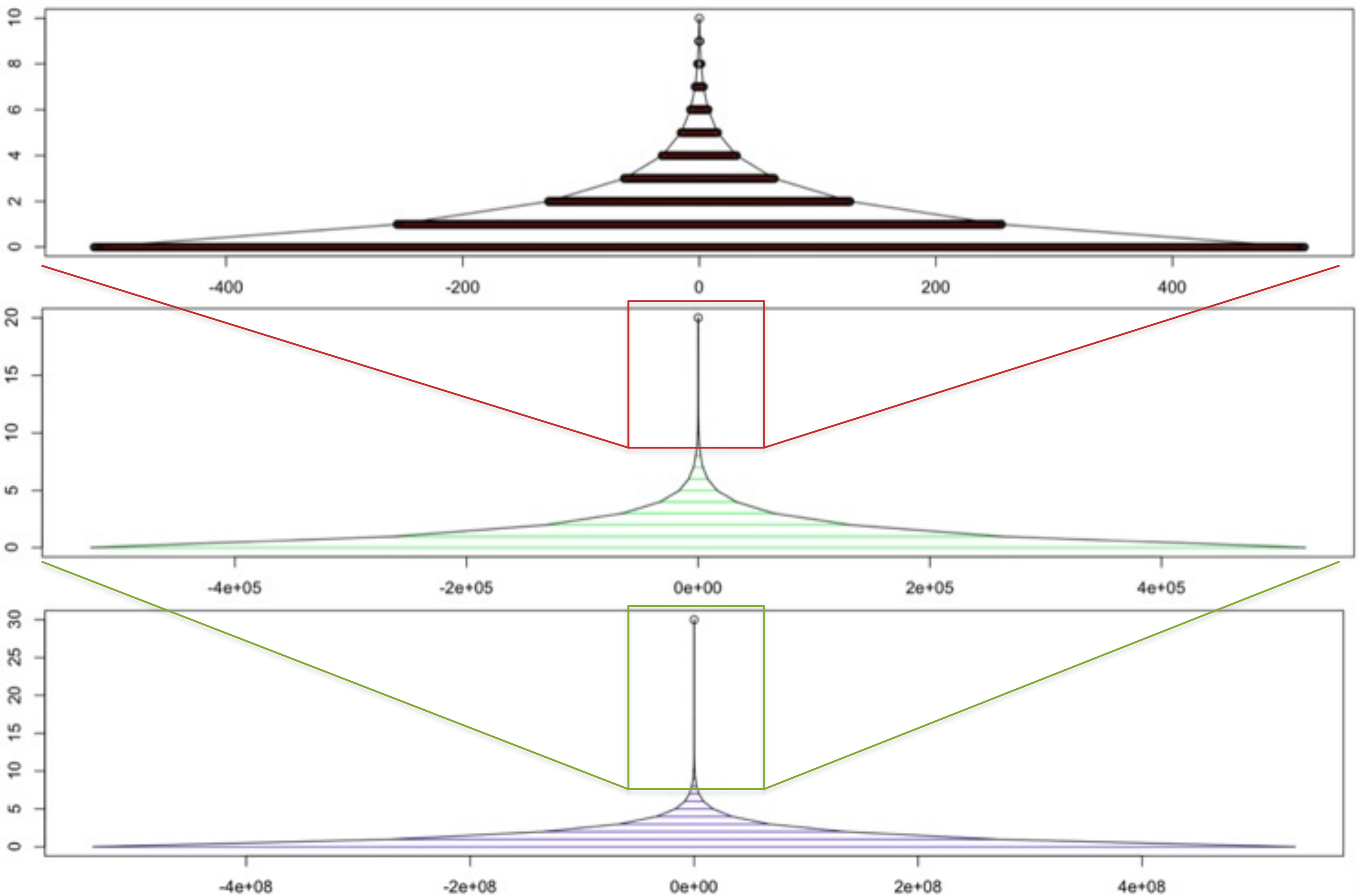


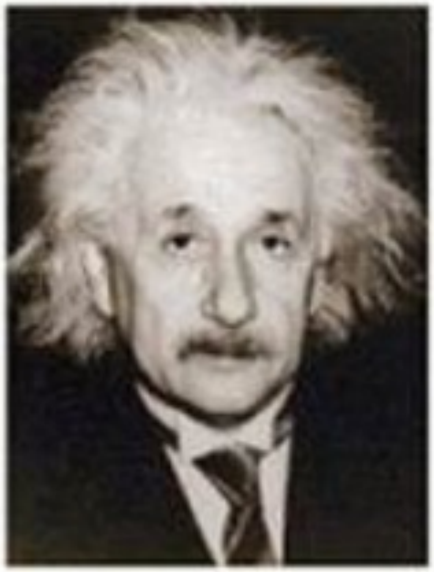
Unbalanced Tree
Non-minimum height



Balanced but not complete!

Tree Heights





"The most powerful force in the universe is **compound interest.**"

Albert Einstein

And 260 channels on TV for $79.95 mo. / 2 yrs.
Guaranteed for 2 yrs.
No Annual Contract.
optimum.
let's connect more
Learn more
Report Advertisement

Compound Interest

Did Albert Einstein declare compound interest to be 'the most powerful force in the universe'?

David Mikkelson
Apr 21, 2011

SHARE 85

SHARE
81

Claim: Albert Einstein once declared compound interest to be "the most powerful force in the universe."

? UNDETERMINED

Examples:

[Sangillo, 2006]

There's an urban legend that Albert Einstein once said compounding [interest] is the most powerful force in the universe.

Whether or not he really said it, that line has become my financial motto. I strongly suggest you adopt it.

[Hartgill, 1997]

Send us a rumor ›

Search snopes.com GO›

optimum.
let's connect more
And 260 channels on TV for
$79.95
mo. / 2 yrs.
Guaranteed for 2 yrs.
No Annual Contract.
Learn more
Report Advertisement

Follow snopes on ›

Get Snopes in your inbox
Your email GO›
We will never send you spam or share your email with nigerian royalty.

The Hot List
1 Hillary Clinton Freed Child Rapist
Hillary Clinton's role in a 40-year-old rape case became the ...
1.0M

Properties of logarithms

	Formula	Example
product	$\log_b(xy) = \log_b(x) + \log_b(y)$	$\log_3(243) = \log_3(9 \cdot 27) = \log_3(9) + \log_3(27) = 2 + 3 = 5$
quotient	$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$	$\log_2(16) = \log_2\left(\frac{64}{4}\right) = \log_2(64) - \log_2(4) = 6 - 2 = 4$
power	$\log_b(x^p) = p\log_b(x)$	$\log_2(64) = \log_2(2^6) = 6\log_2(2) = 6$
root	$\log_b \sqrt[p]{x} = \frac{\log_b(x)}{p}$	$\log_{10}\sqrt{1000} = \frac{1}{2}\log_{10} 1000 = \frac{3}{2} = 1.5$

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}.$$

Height with n leaves?	= lg (n)		= O(lg n)
Height with n nodes?	= n/2 leaves	= lg (n/2) = lg (n) – lg 2 = lg(n) – 1	= O(lg n)
3-ary with n leaves?	= $\log_3$ (n)		= O(lg n)
k-ary with n nodes?	= n/k leaves	= $\log_k$(n/k) = $\log_k$(n) – $\log_k$(k)	= O(lg n)

Tree Interface

```
public interface Tree<T>{
    Position <T> insertRoot(T t )
        throws TreeNotEmptyException;

    Position <T> insertChild(Position <T> p, T t)
        throws InvalidPositionException;

    boolean empty();

    Position <T> root()
        throws TreeEmptyException;

    Position <T>[] children(Position <T> p)
        throws InvalidPositionException , LeafException;

    Position <T> parent(Position<T> p)
        throws InvalidPositionException ;

    boolean leaf (Position <T> p)
        throws InvalidPositionException ;

    T remove(Position<T> p)
        throws InvalidPositionException, NotALeafException;
}
```

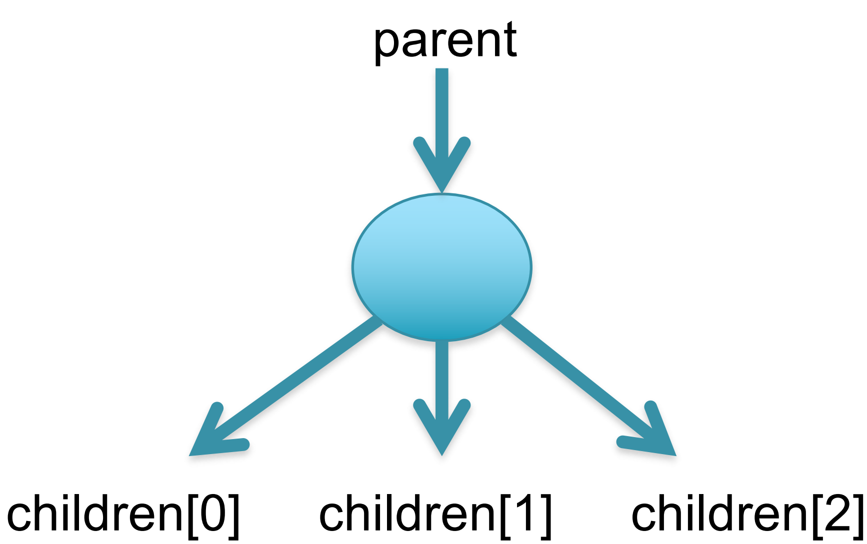
Why insertRoot?

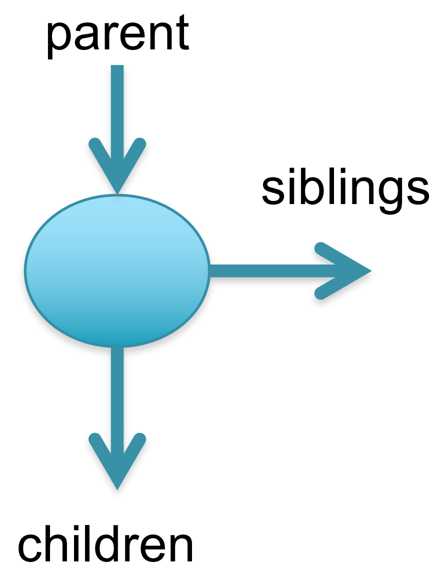
Why children()?

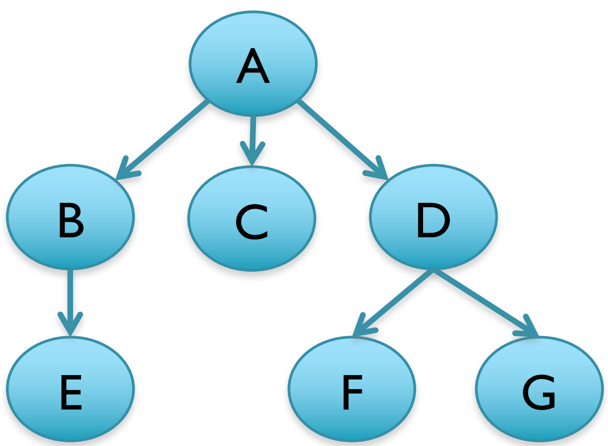
What should parent(root()) return?

What should remove(root()) do?

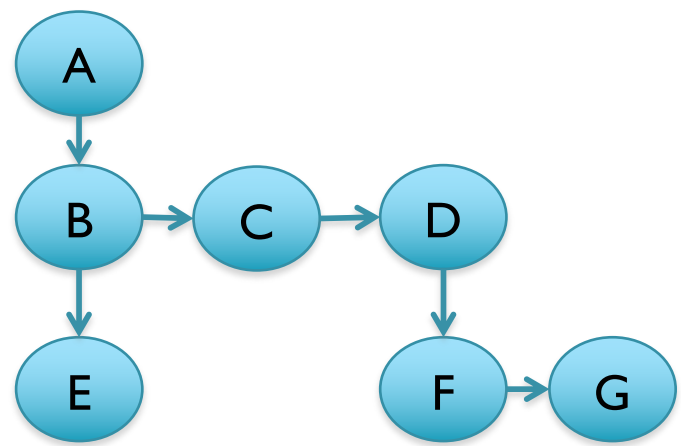
Tree Implementation





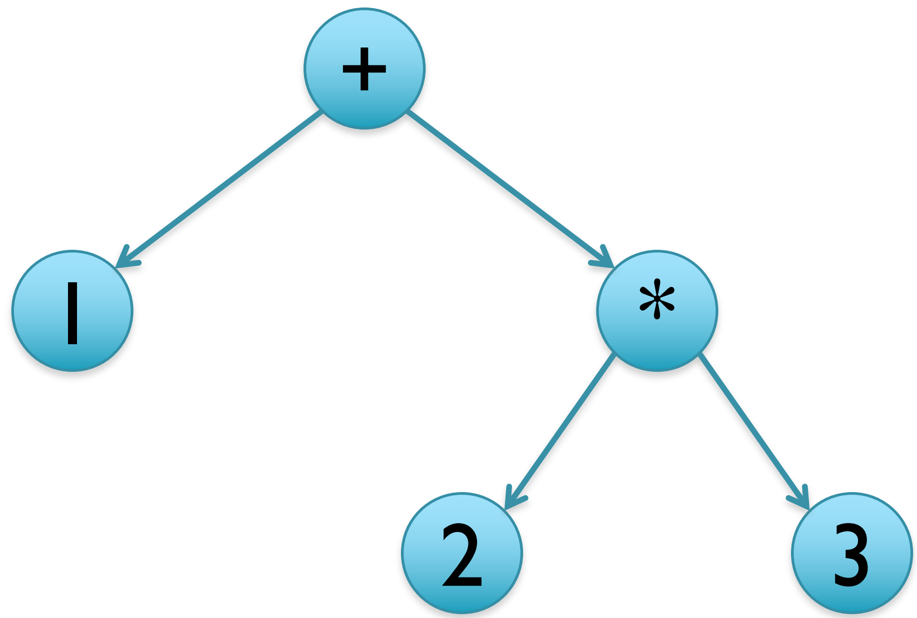


Simple Implementation
Overhead managing children[]

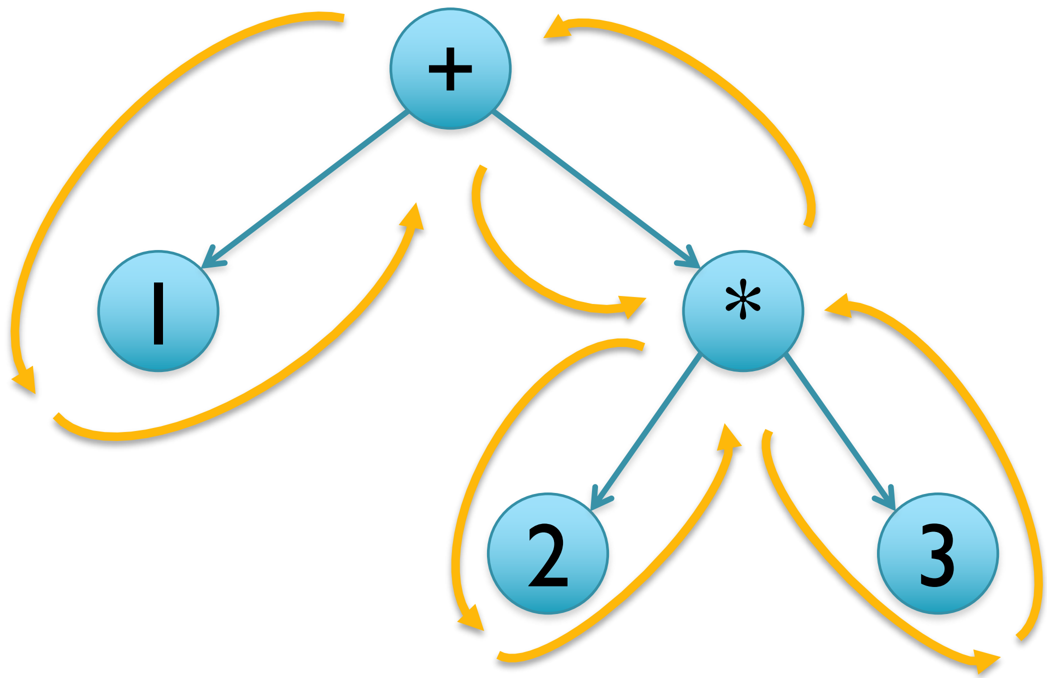


Less Space Overhead
(Except remove may require doubly linked lists)

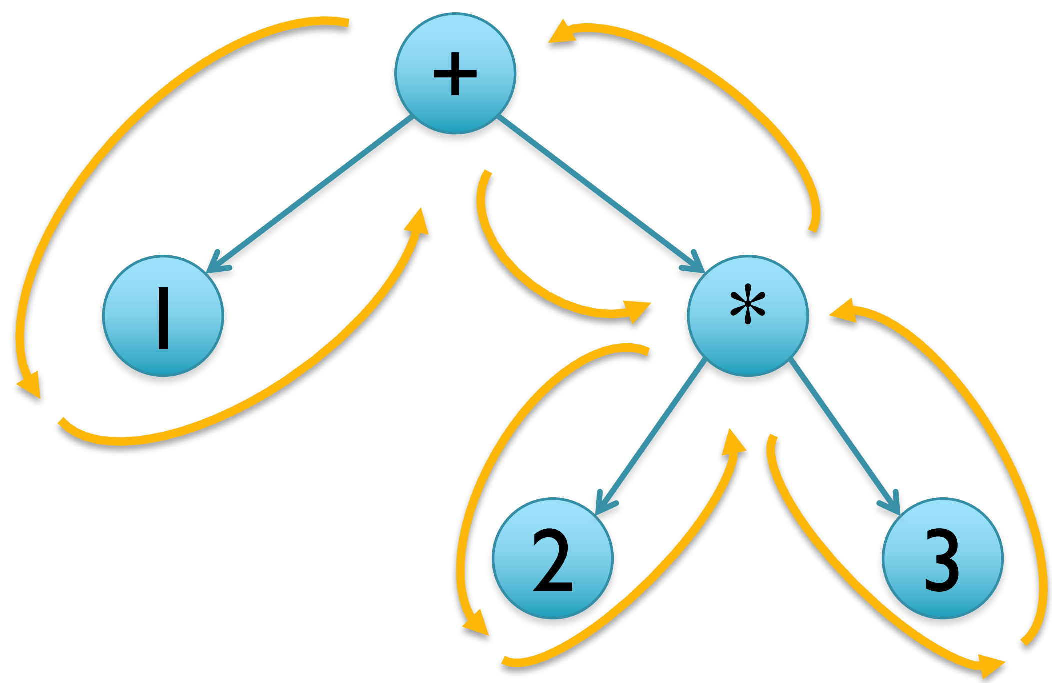
Tree Traversals



Tree Traversals



Tree Traversals



Note here we visit children from left to right, but could go right to left

Notice we visit internal nodes 3 times:
1: stepping down from parent
2: after visiting first child
3: after visiting second child

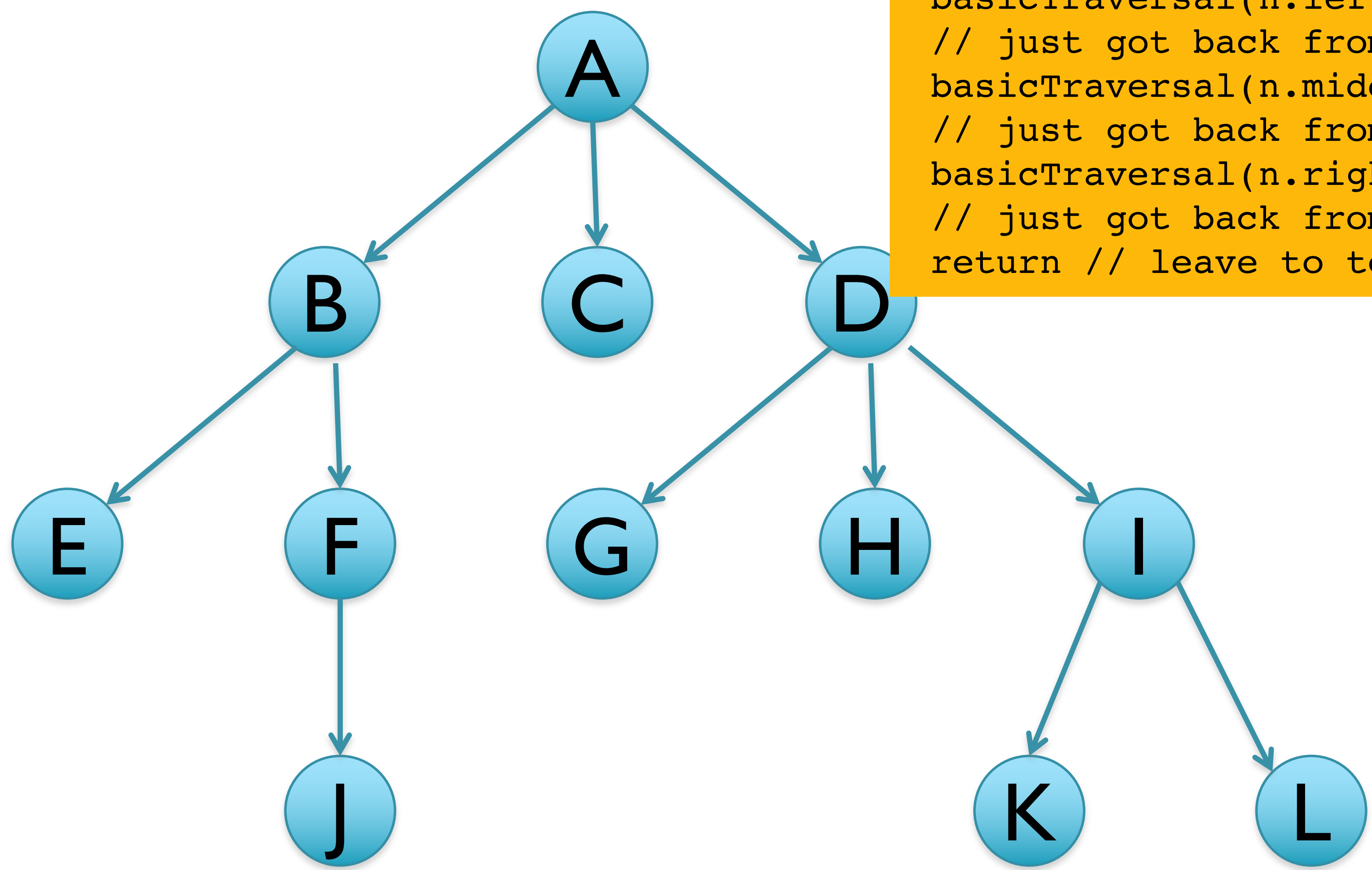
Different algs work at different times

1: preorder	+ 1 * 2 3
2: inorder	1 + 2 * 3
3: postorder	1 2 3 * +

Traversal Implementations

How to traverse?

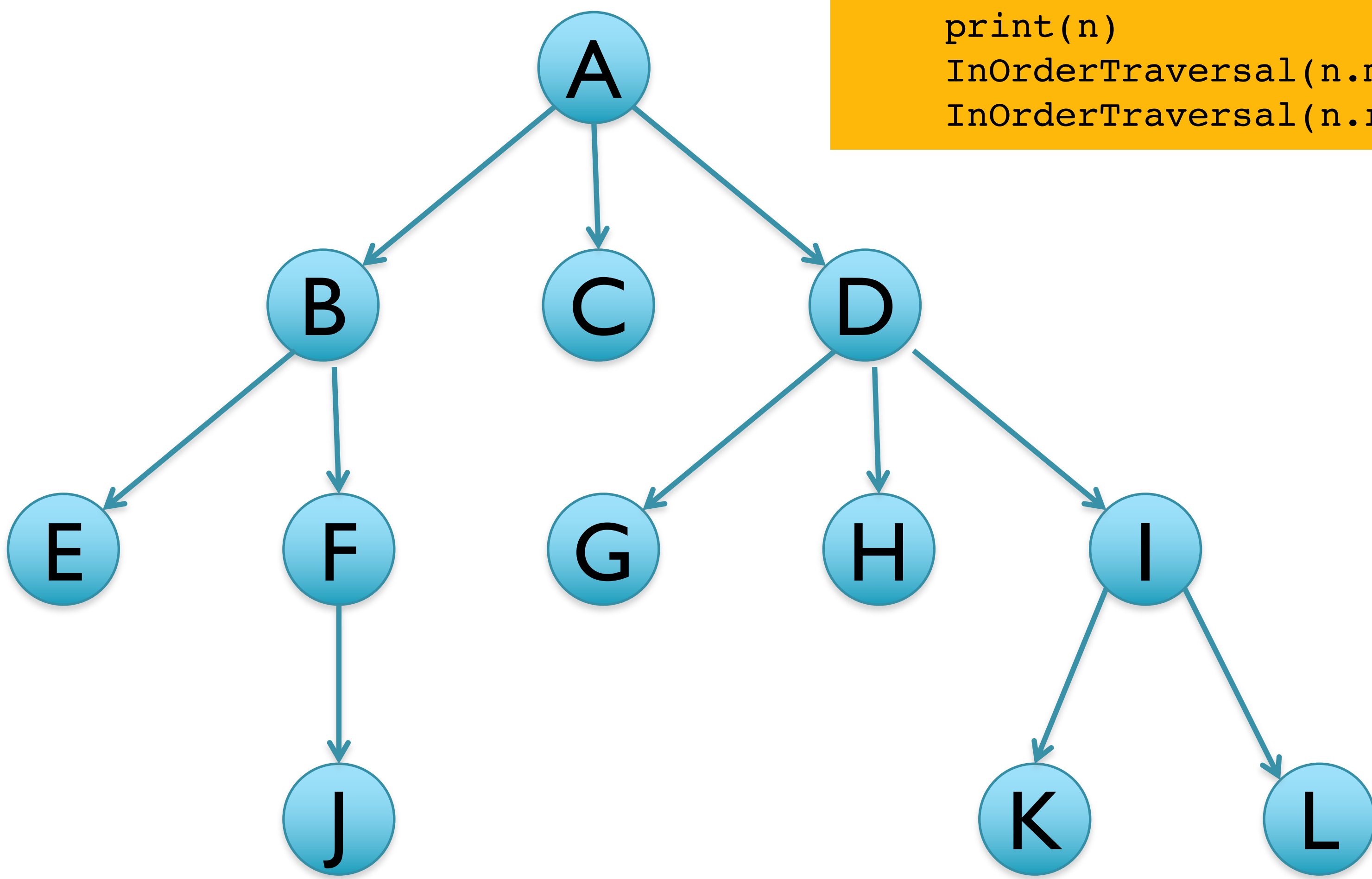
```
basicTraversal(Node n):
 // just entered from top
 basicTraversal(n.left)
 // just got back from left
 basicTraversal(n.middle)
 // just got back from middle
 basicTraversal(n.right)
 // just got back from right
 return // leave to top
```



InOrder Traversals

How to inorder print?

```
InOrderTraversal(Node n):
  if n is not null
    InOrderTraversal(n.left)
    print(n)
    InOrderTraversal(n.middle)
    InOrderTraversal(n.right)
```

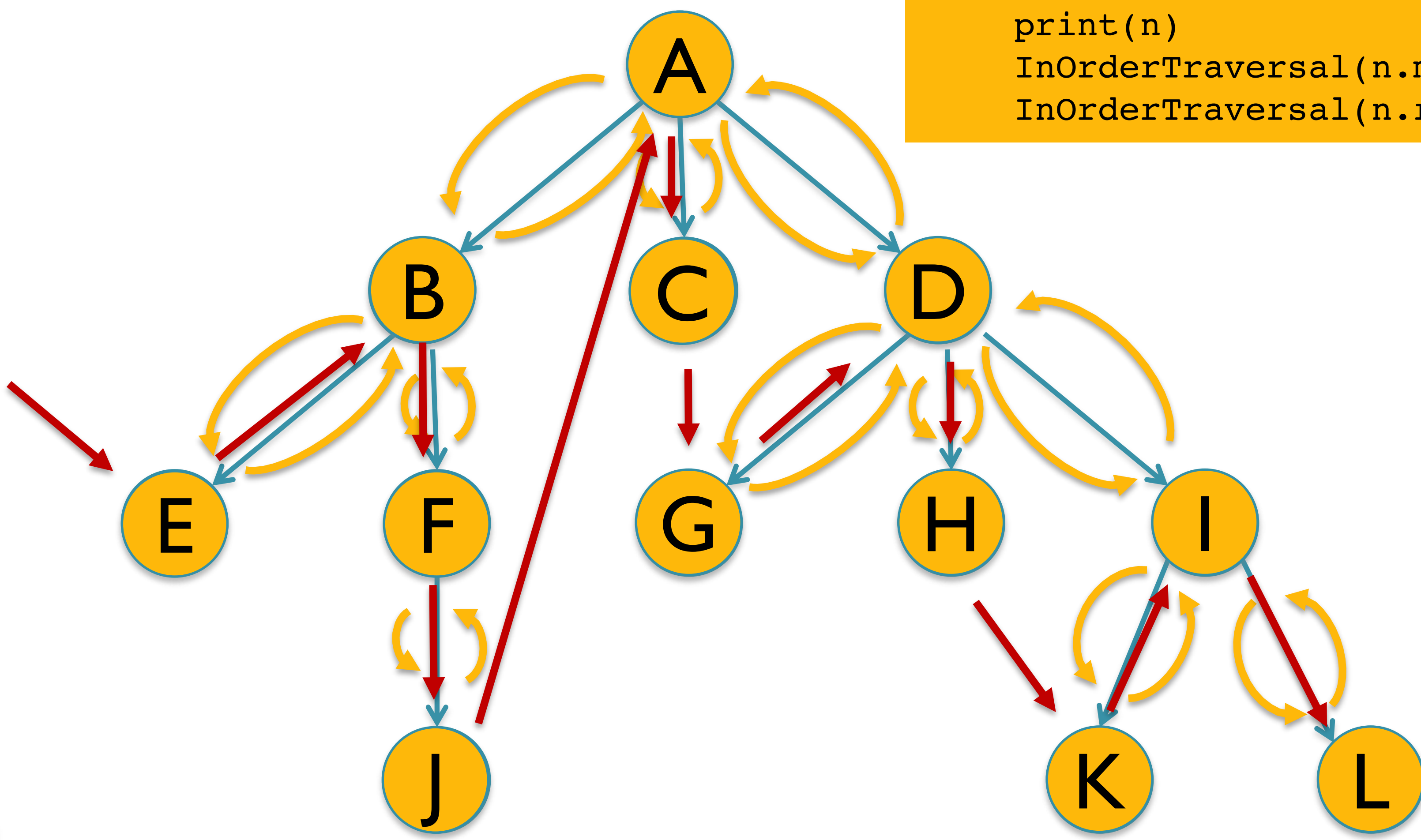


InOrder Traversals

What is the inorder print?

How to inorder print?

```
InOrderTraversal(Node n):
  if n is not null
    InOrderTraversal(n.left)
    print(n)
    InOrderTraversal(n.middle)
    InOrderTraversal(n.right)
```



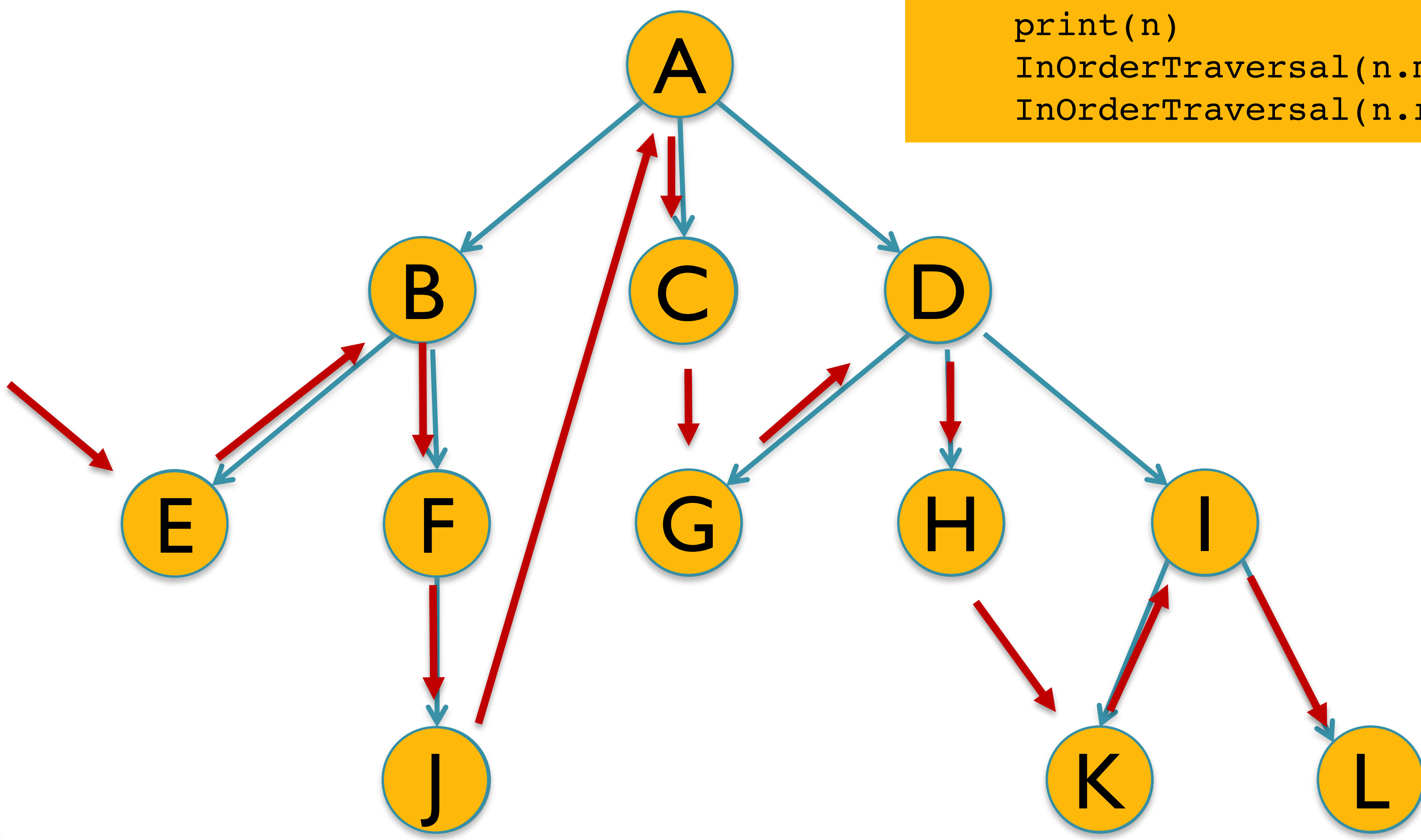
InOrder Traversals

What is the inorder print?

EBFJ AC GDHKIL

How to inorder print?

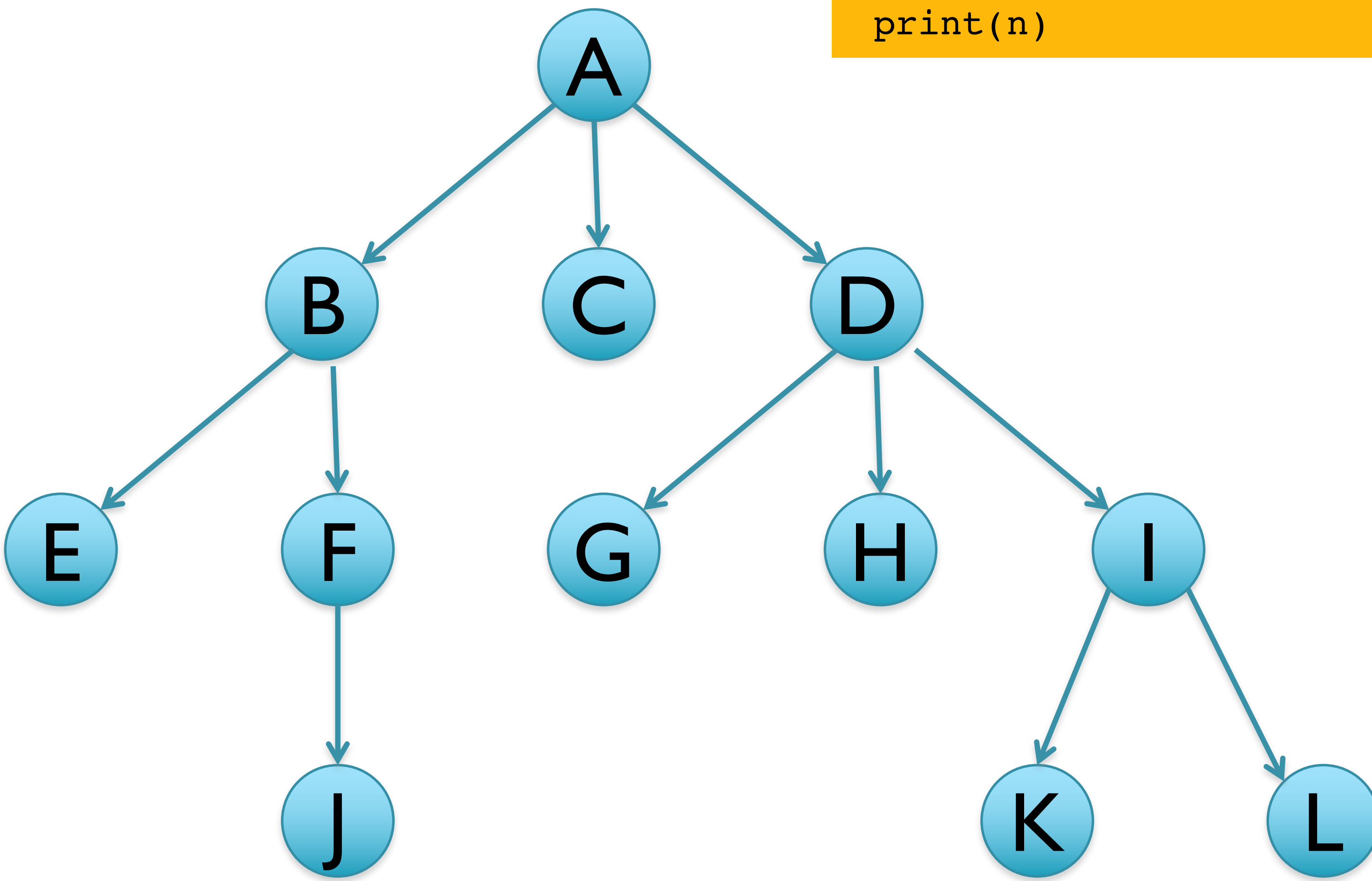
```
InOrderTraversal(Node n):
  if n is not null
    InOrderTraversal(n.left)
    print(n)
    InOrderTraversal(n.middle)
    InOrderTraversal(n.right)
```



PostOrder Traversals

How to postorder print?

```
PostOrderTraversal(Node n):
 for c in x.children:
    PostOrderTraversal(c)
 print(n)
```

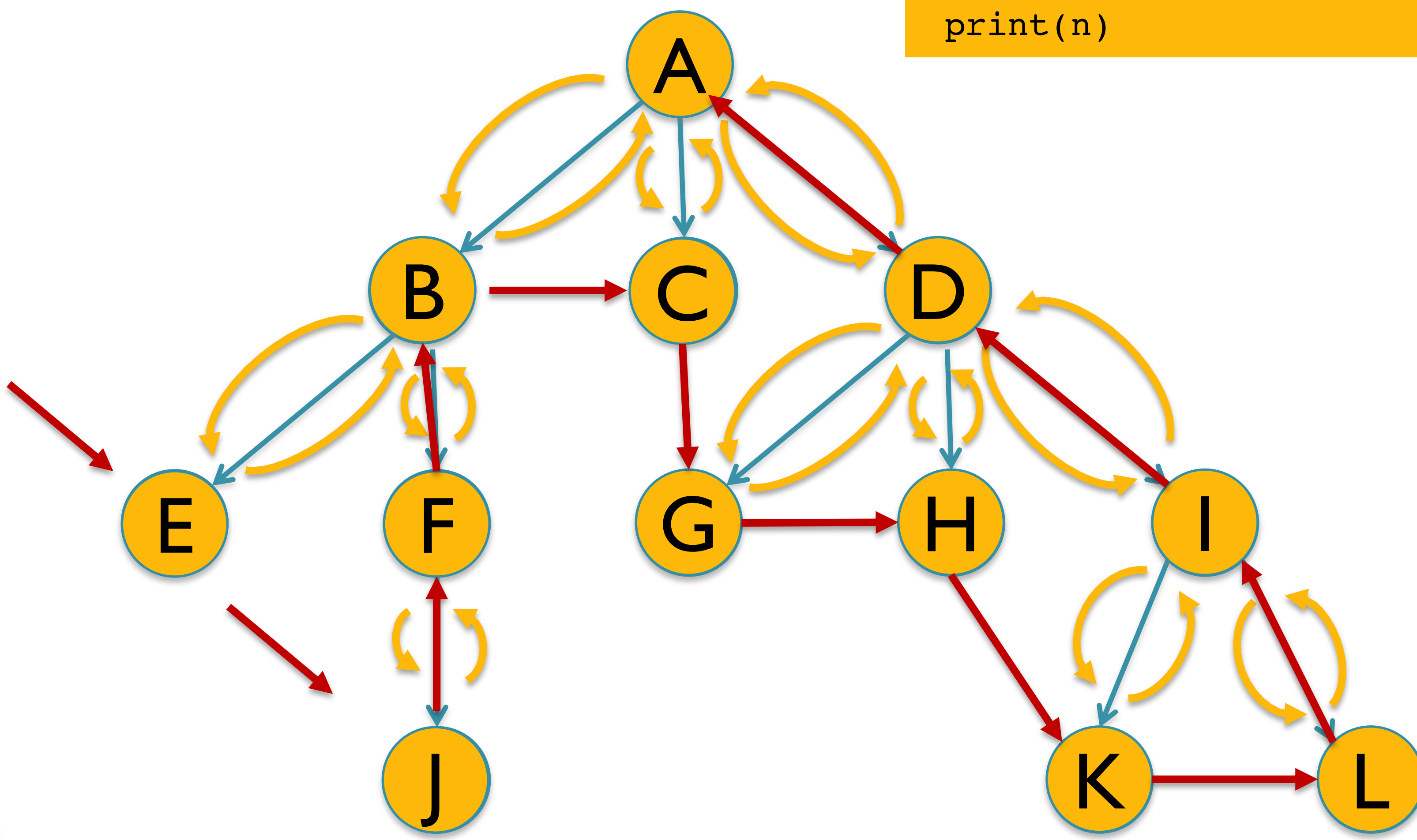


PostOrder Traversals

What is the postorder print?

How to postorder print?

```
PostOrderTraversal(Node n):
 for c in x.children:
     PostOrderTraversal(c)
 print(n)
```



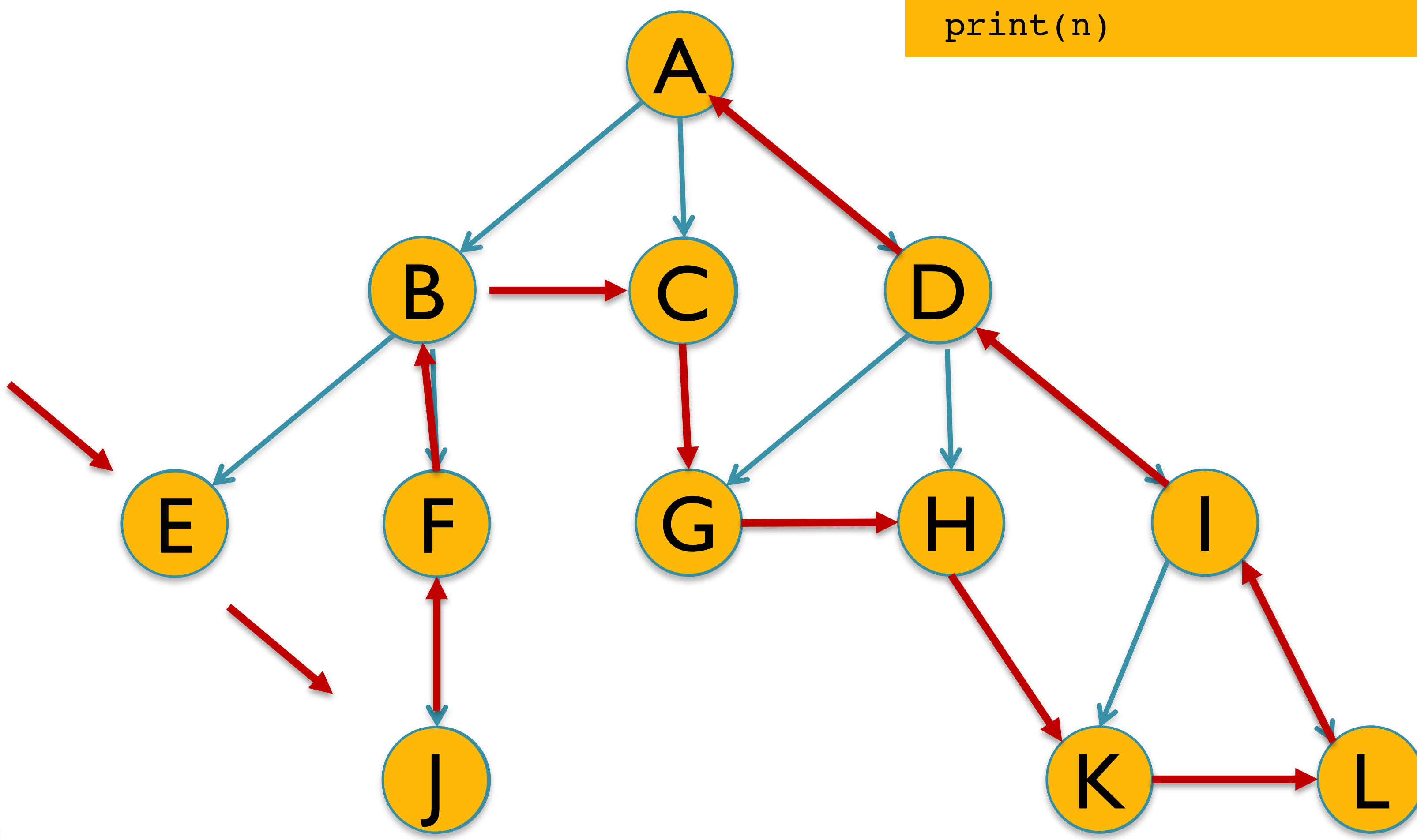
PostOrder Traversals

What is the postorder print?

EJFB C GHKLIDA

How to postorder print?

```
PostOrderTraversal(Node n):
 for c in x.children:
     PostOrderTraversal(c)
 print(n)
```

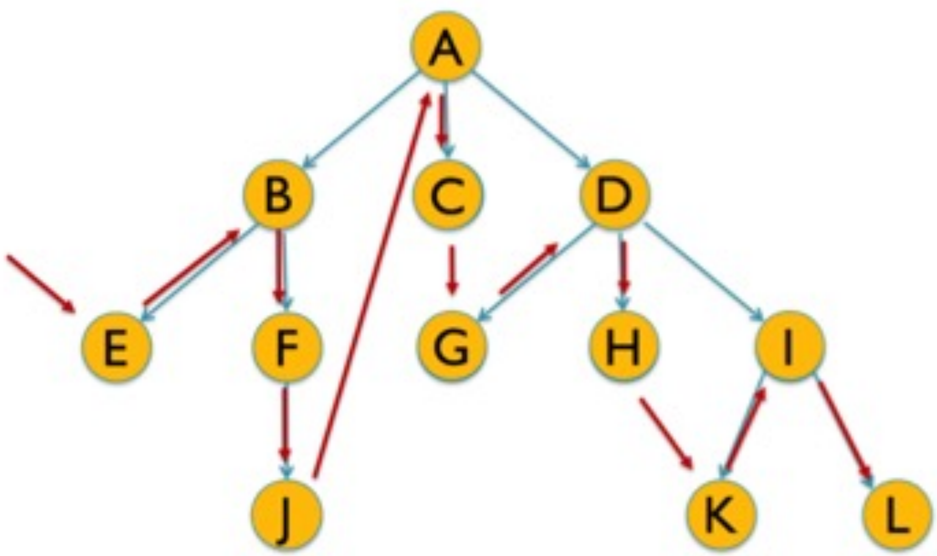


InOrder vs PostOrder

What is the inorder print?

EBFJ AC GDHKIL

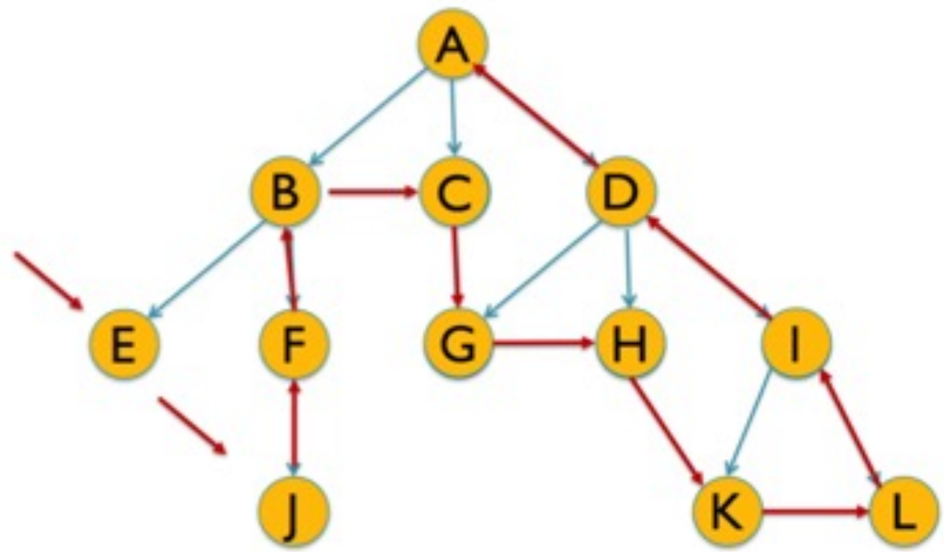
```
InOrderTraversal(Node n):
  if n is not null
     InOrderTraversal(n.left)
     print(n)
     InOrderTraversal(n.middle)
     InOrderTraversal(n.right)
```



What is the postorder print?

EJFB C GHKLIDA

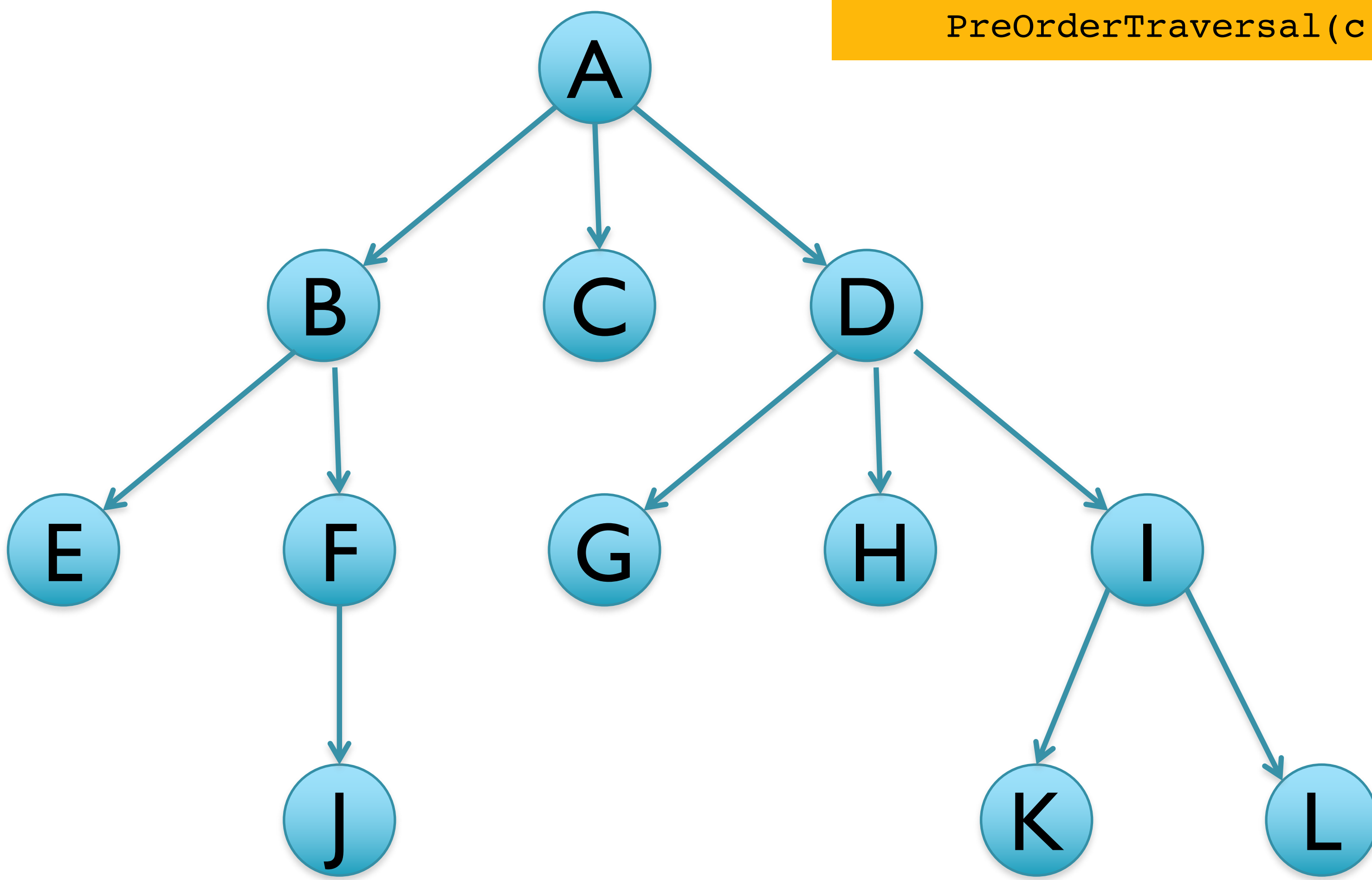
```
PostOrderTraversal(Node n):
  for c in x.children:
     PostOrderTraversal(c)
  print(n)
```



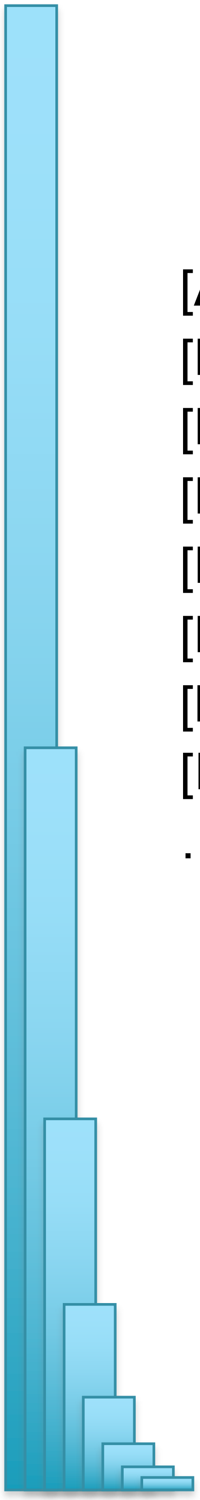
PreOrder Traversals

How to preorder print?

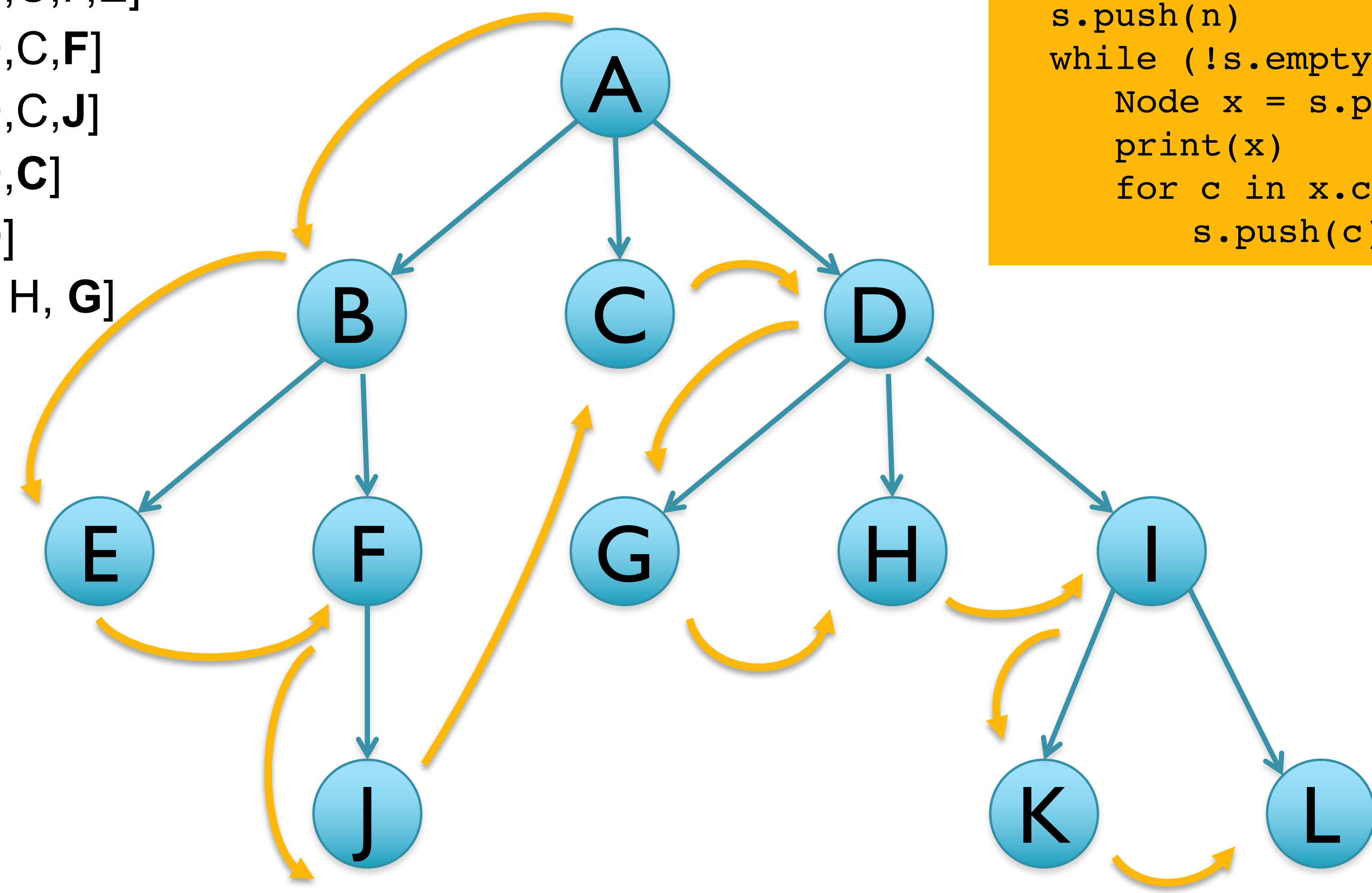
```
PreOrderTraversal(Node n):
 print(n)
 for c in x.children:
     PreOrderTraversal(c)
```



PreOrder Traversals



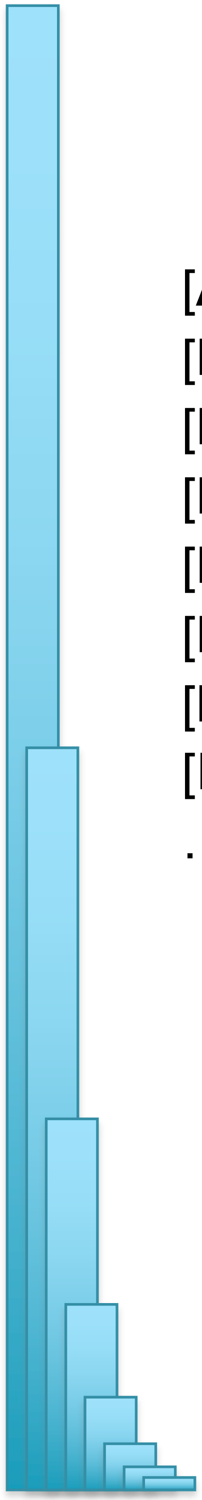
[**A**]
[D,C,**B**]
[D,C,F,**E**]
[D,C,**F**]
[D,C,**J**]
[D,**C**]
[**D**]
[I, H, **G**]
…



How to preorder print?

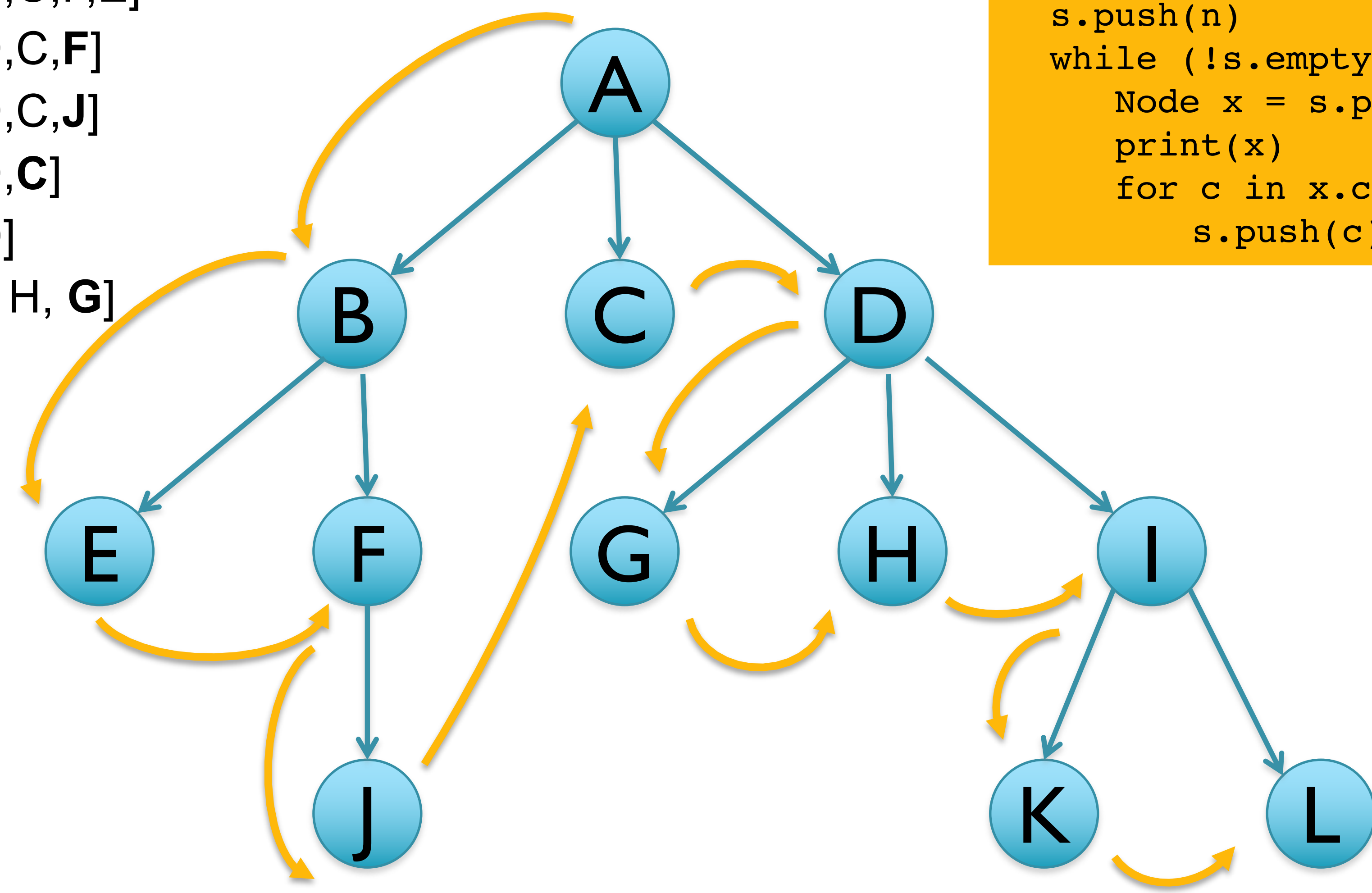
```
PreOrderTraversal(Node n):
  Stack s
  s.push(n)
  while (!s.empty()):
     Node x = s.pop()
     print(x)
     for c in x.children:
        s.push(c)
```

PreOrder Traversals



[**A**]
[D,C,**B**]
[D,C,F,**E**]
[D,C,**F**]
[D,C,**J**]
[D,**C**]
[**D**]
[I, H, **G**]
…

Stack leads to a Depth-First Search



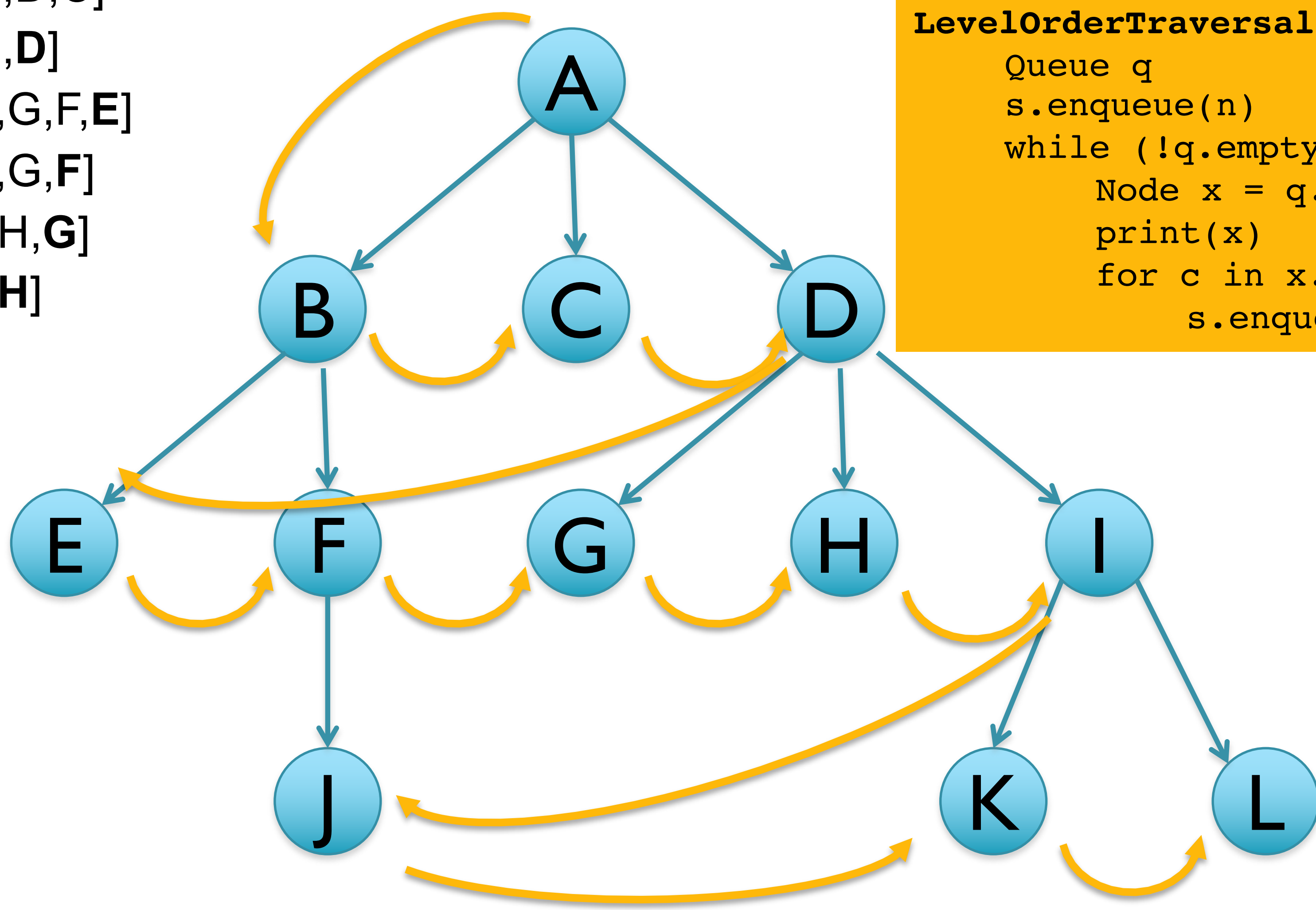
How to preorder print?

```
PreOrderTraversal(Node n):
  Stack s
  s.push(n)
  while (!s.empty()):
     Node x = s.pop()
     print(x)
     for c in x.children:
        s.push(c)
```

Level Order Traversals

[**A**]
[D,C,**B**]
[F,E,D,**C**]
[F,E,**D**]
[I,H,G,F,**E**]
[I,H,G,**F**]
[J,I,H,**G**]
[J,I,**H**]
...

Queue leads to a Breadth-First Search



How to level order print?

(A) (B C D) (E F G H I) (J K L)

```
LevelOrderTraversal(Node n):
    Queue q
    s.enqueue(n)
    while (!q.empty()):
        Node x = q.dequeue()
        print(x)
        for c in x.children:
            s.enqueue(c)
```

Multiple Traversals

```
public abstract class Operation<T> {
    void pre(Position<T> p) {}
    void in(Position<T> p) {}
    void post(Position<T> p) {}
}

public interface Tree<T> {
     ...
     traverse(Operation<T> o);
    ...
}

  // Tree implementation pseudo-code:
  niceTraversal(Node n, Operation o):
     if n is not null:
         o.pre(n)
         niceTraversal(n.left, o)
         o.in(n)
         niceTraversal(n.right, o)
         o.post(n)
}
```

Abstract class simplifies the use of function objects - functors

Client extends Operation<T> but overrides just the methods that are needed ☺

Implementation (1)

```
public class TreeImplementation<T> implements Tree<T> {
...
  private static class Node<T> implements Position<T> {
    T data;
    Node<T> parent;
    ArrayList<Node<T>> children;

    public Node(T t) {
        this.children = new ArrayList<Node<T>>();
        this.data = t;
    }

    public T get() {
        return this.data;
    }

    public void put(T t) {
        this.data = t;
    }
  }
}
```

Constructor ensures children, data are initialized correctly

What other fields might we want to include? (Hint: Position<>)

Should set the “owner” field to point to this Tree so Position<> can be checked

Implementation (2)

```
public Position<T> insertRoot(T t) throws InsertionException {
    if (this.root != null) {
        throw new InsertionException();
    }
    this.root = new Node<T>(t);
    this.elements += 1;
    return this.root;
}


public Position<T> insertChild(Position<T> pos, T t)
                              throws InvalidPositionException {
    Node<T> p = this.convert(pos);
    Node<T> n = new Node<T>(t);
    n.parent = p;
    p.children.add(n);
    this.elements += 1;
    return n;
}
```

convert?

convert method (a private helper) takes a position, validates it, and then returns the Node<T> object hiding behind the position

Implementation (3)

```
public boolean empty() {
    return this.elements == 0;
}

public int size() {
    return this.elements;
}

public boolean hasParent(Position<T> p) throws
                                InvalidPositionException {
    Node<T> n = this.convert(p);
    return n.parent != null;
}

public boolean hasChildren(Position<T> p) throws
                                   InvalidPositionException {
    Node<T> n = this.convert(p);
    return !n.children.isEmpty();
}
```

Traversal

```
private void recurse(Node<T> n, Operation<T> o) {
    if (n == null) { return; }
    o.pre(n);
    for (Node<T> c: n.children) {
        this.recurse(c, o);
        // figure out when to call o.in(n)
    }
    o.post(n);
}

public void traverse(Operation<T> o) {
    this.recurse(this.root, o);
}
```

Private helper method working with Node<T> rather than Position<T>

Just make sure we start at root

When should we call o.in()?

We don't want to call the in method after we visit the last child.
We do want to call the in method even for a node with no children

More to come…

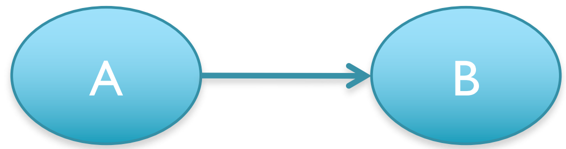
DRAFT

Graphs

Graphs are Everywhere!

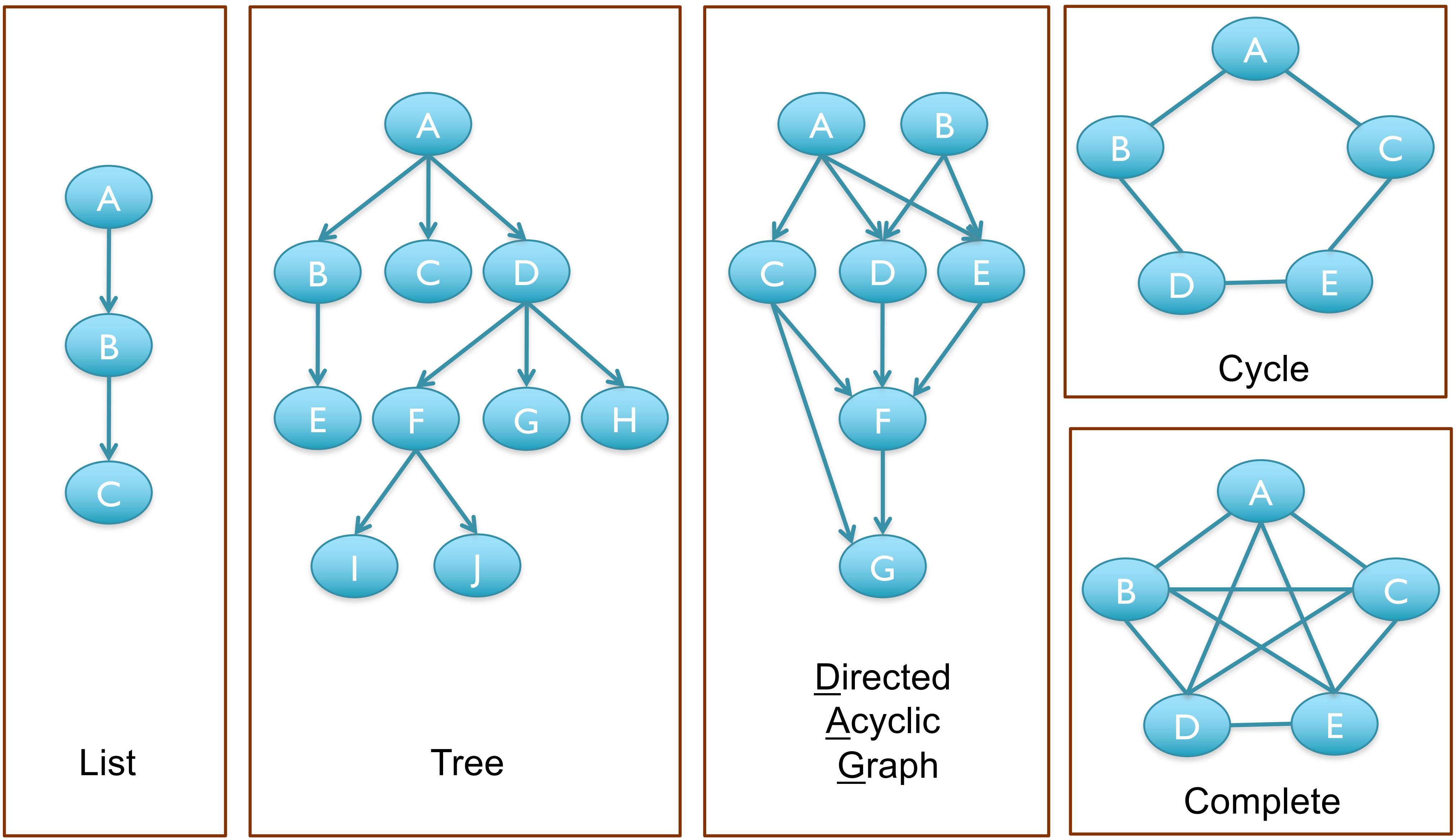
Computers in a network, Friends on Facebook, Roads & Cities on GoogleMaps, Webpages on Internet, Cells in your body, …

Graphs

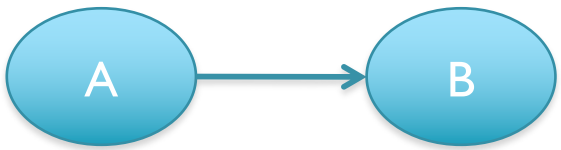


- Nodes aka vertices
  - People, Proteins, Cities, Genes, Neurons, Sequences, Numbers, …
- Edges aka arcs
  - A is connected to B
  - A is related to B
  - A regulates B
  - A precedes B
  - A interacts with B
  - A activates B
  - …

Graph Types

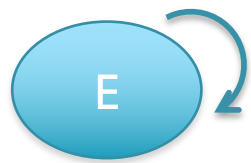


Definitions (1)



Directed Edge

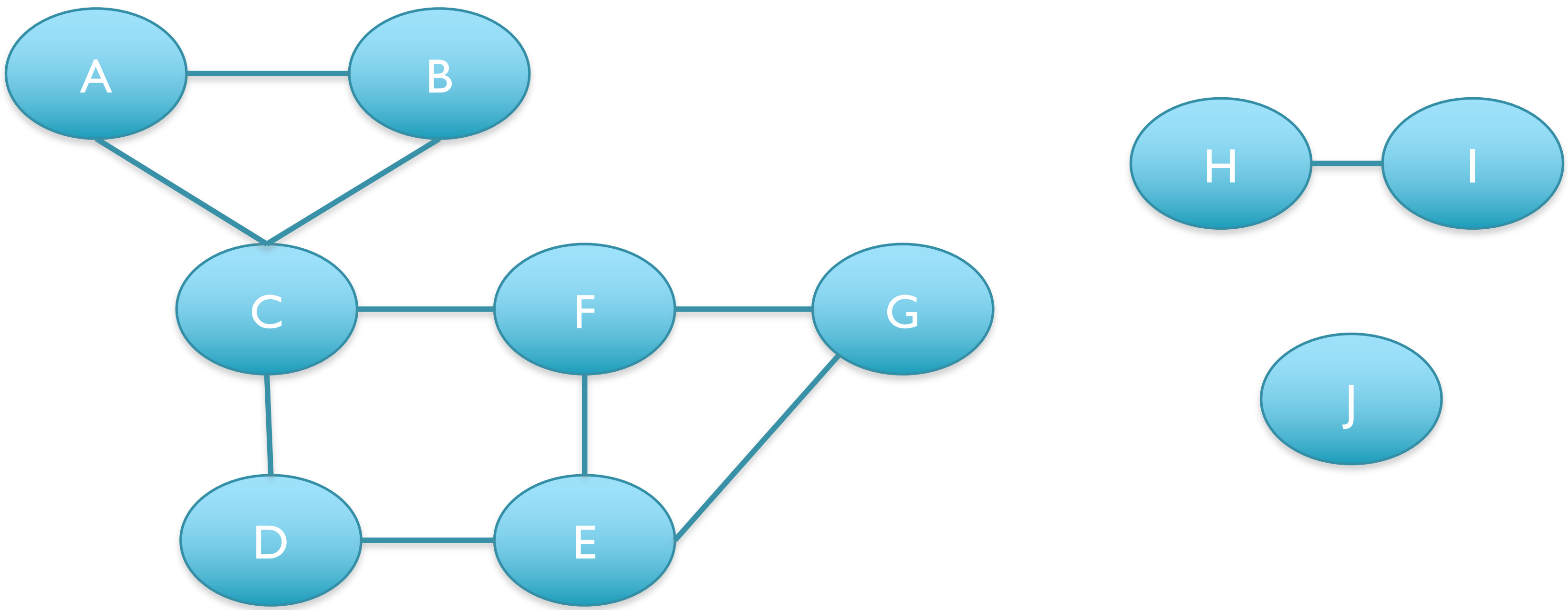
Undirected Edge



Self Edge
(Unusual but usually allowed)

- A and B are ***adjacent***
- An edge connected to a vertex is ***incident*** on that vertex
- The number of edges incident on a vertex is the ***degree*** of that vertex
  - For directed graphs, separately report ***indegree*** and ***outdegree***
- A ***multigraph*** allows multiple edges between the same pair of nodes, a ***simple*** graph does not allow multiple edges (most common)

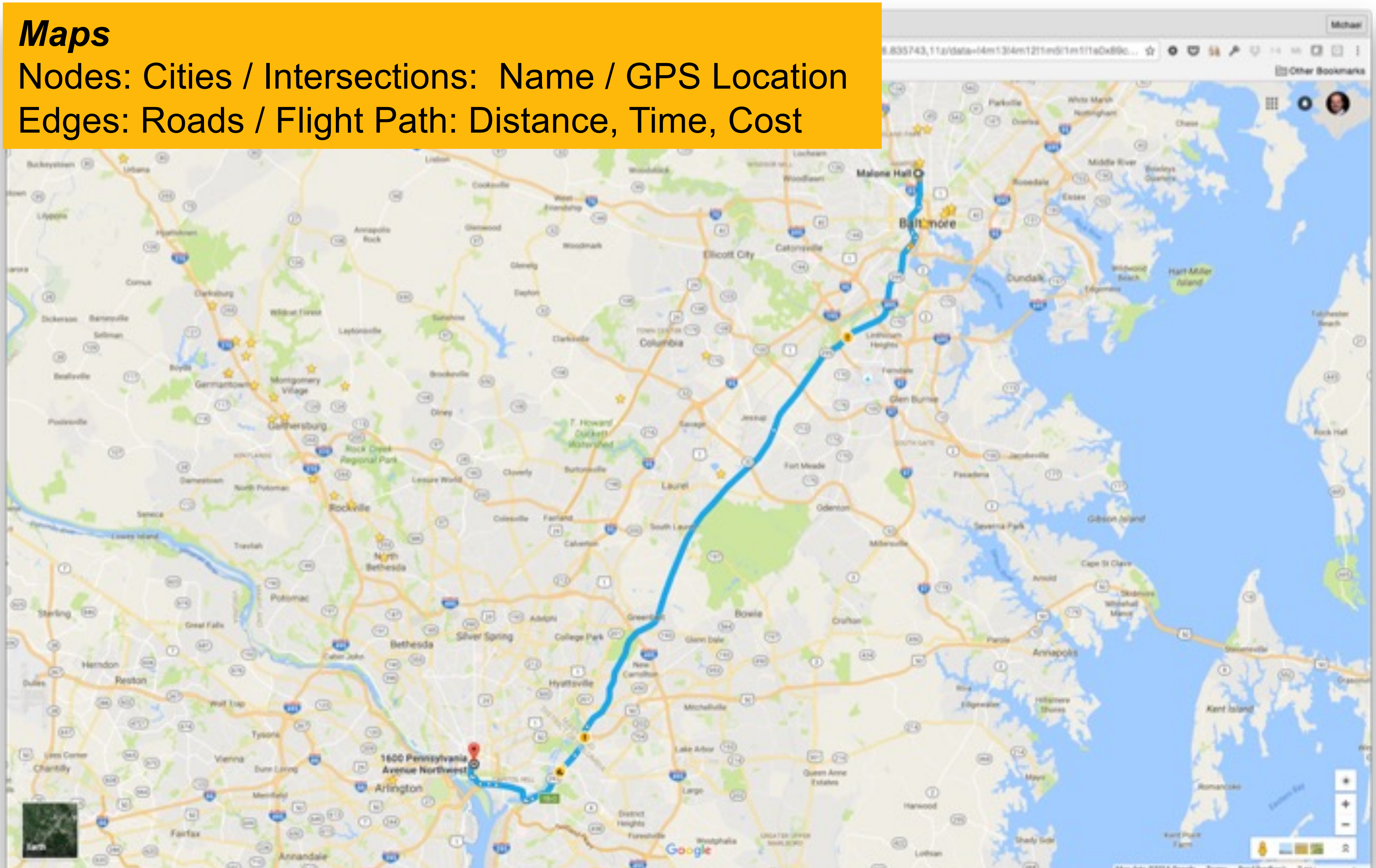
Definitions (2)



- A ***path*** is a sequence of edges e_1, e_2, … e_n in which each edge starts from the vertex the previous edge ended at
- A path that starts and ends at the same node is a ***cycle***
- The number of edges in a path is called the ***length*** of the path
- A graph is ***connected*** if there is a path between every pair of nodes, otherwise it is ***disconnected*** into >1 ***connected components***

The Road to the White House

Maps
Nodes: Cities / Intersections: Name / GPS Location
Edges: Roads / Flight Path: Distance, Time, Cost



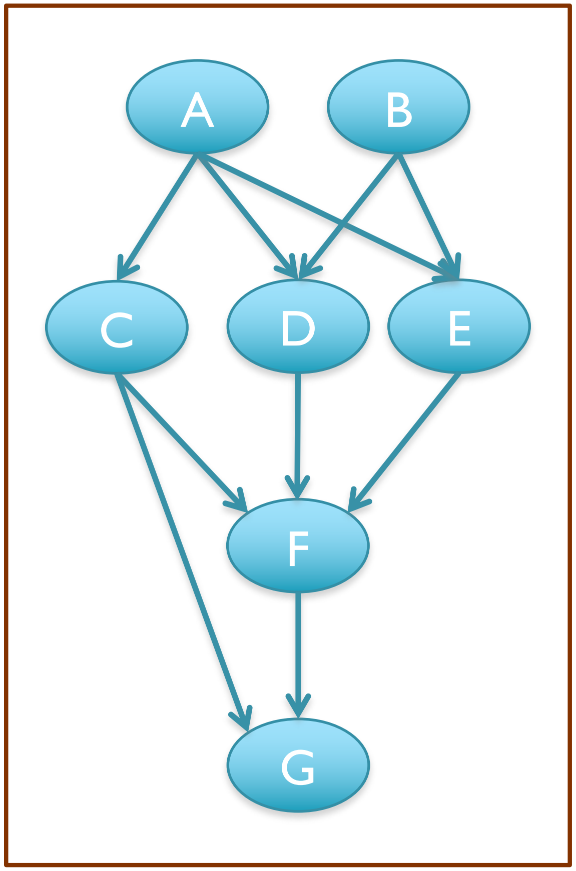
Graph Interface

Separate generic types for vertices <V> and edges <E>

```
public interface Graph<V,E> {
    ...
    Position<V> insertVertex(V v);

    Position<E> insertEdge(Position<V> from, Position<V> to, E e)
            throws InvalidPositionException, InsertionException;

    V removeVertex(Position<V> p)
            throws InvalidPositionException, RemovalException;

    E removeEdge(Position<E> p)
                              throws InvalidPositionException;

    Iterable<Position<V>> vertices();

    Iterable<Position<E>> edges();

    Iterable<Position<E>> incomingEdges(Position<V> p)
                              throws InvalidPositionException;

    Iterable<Position<E>> outgoingEdges(Position<V> p)
                              throws InvalidPositionException;
}
```

Can iterate through all the nodes OR iterate through all the edges as different algorithms may require one or the other

Representing Graphs

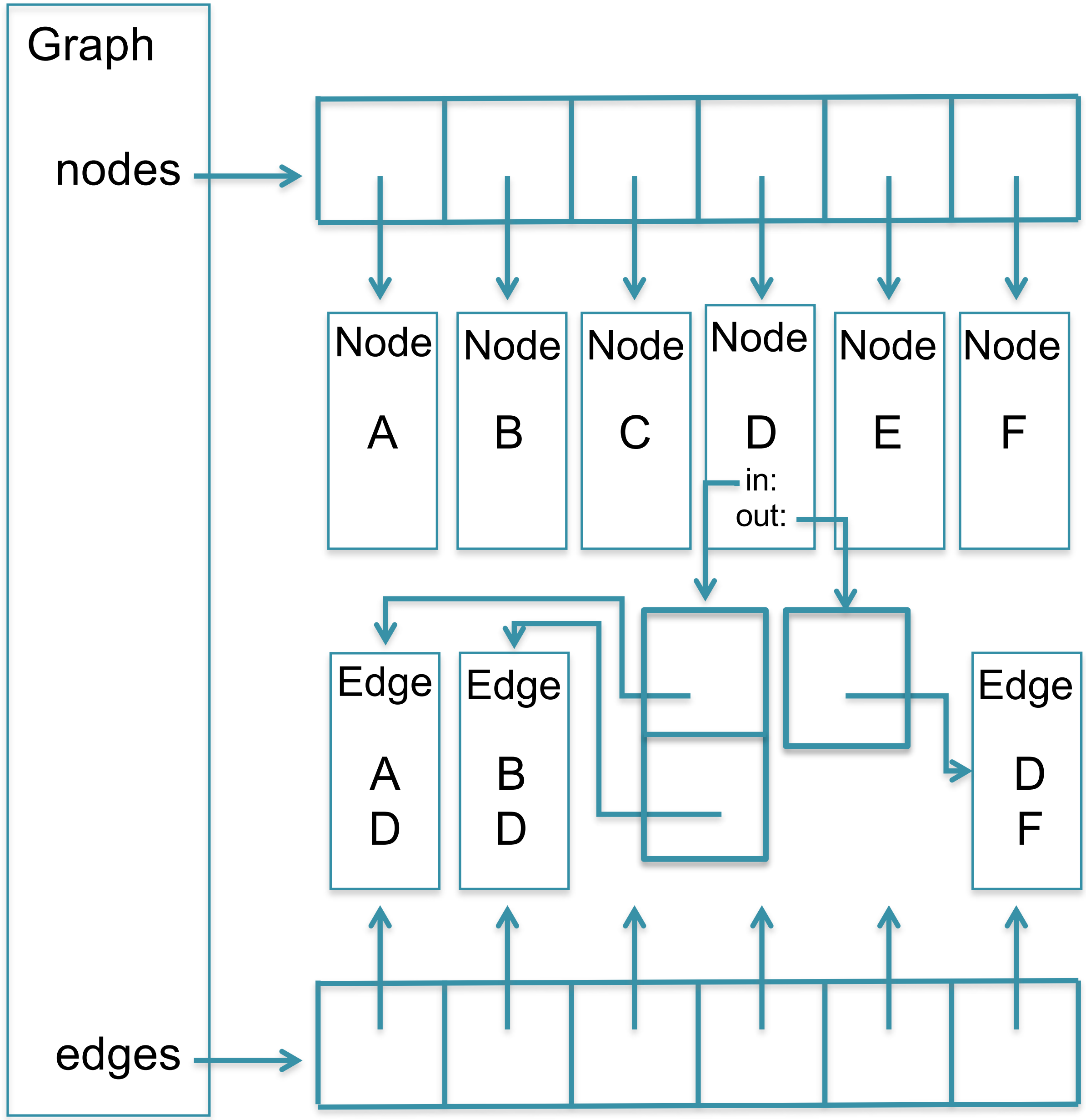


Incidence List

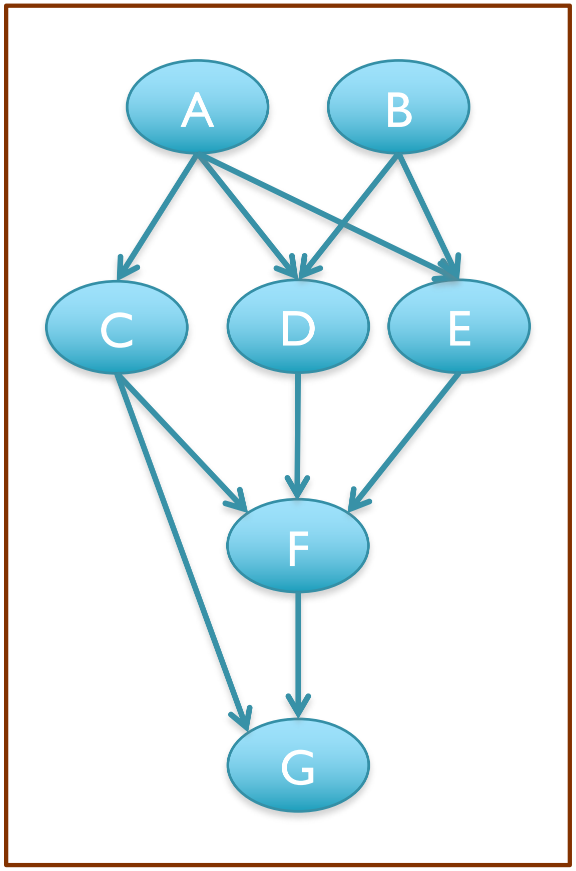
Good for sparse graphs

Compact storage

A: C, D, E
B: D, E
C: F, G
D: F
E: F
G:



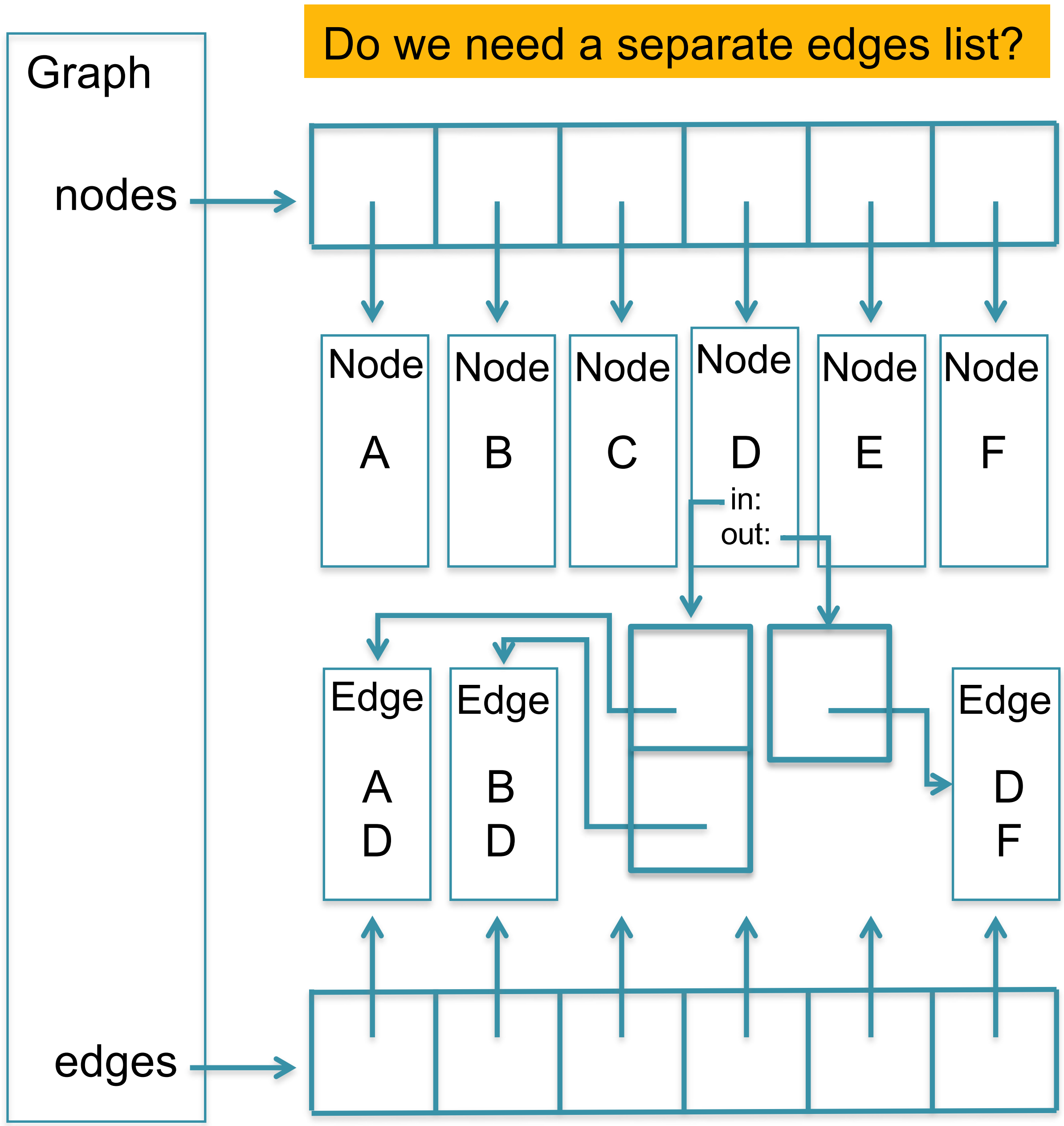
Representing Graphs



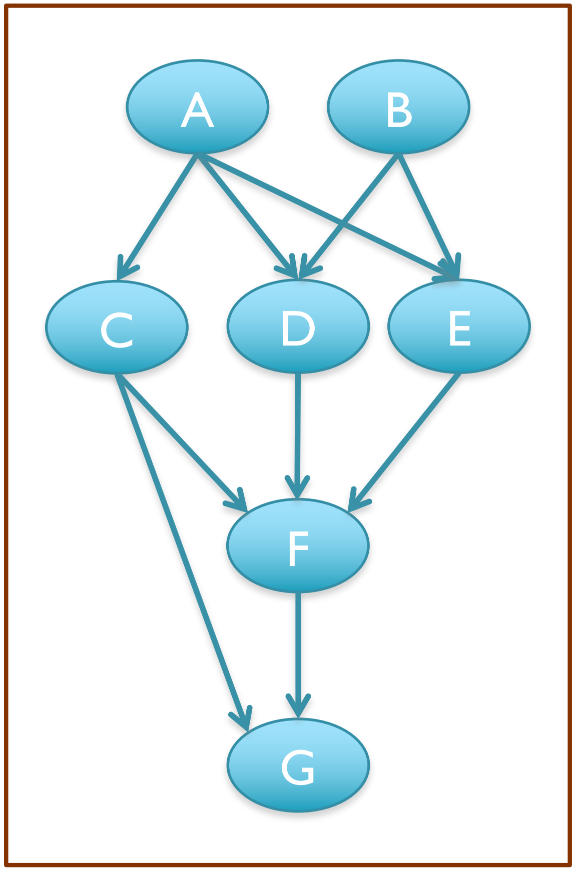
Incidence List

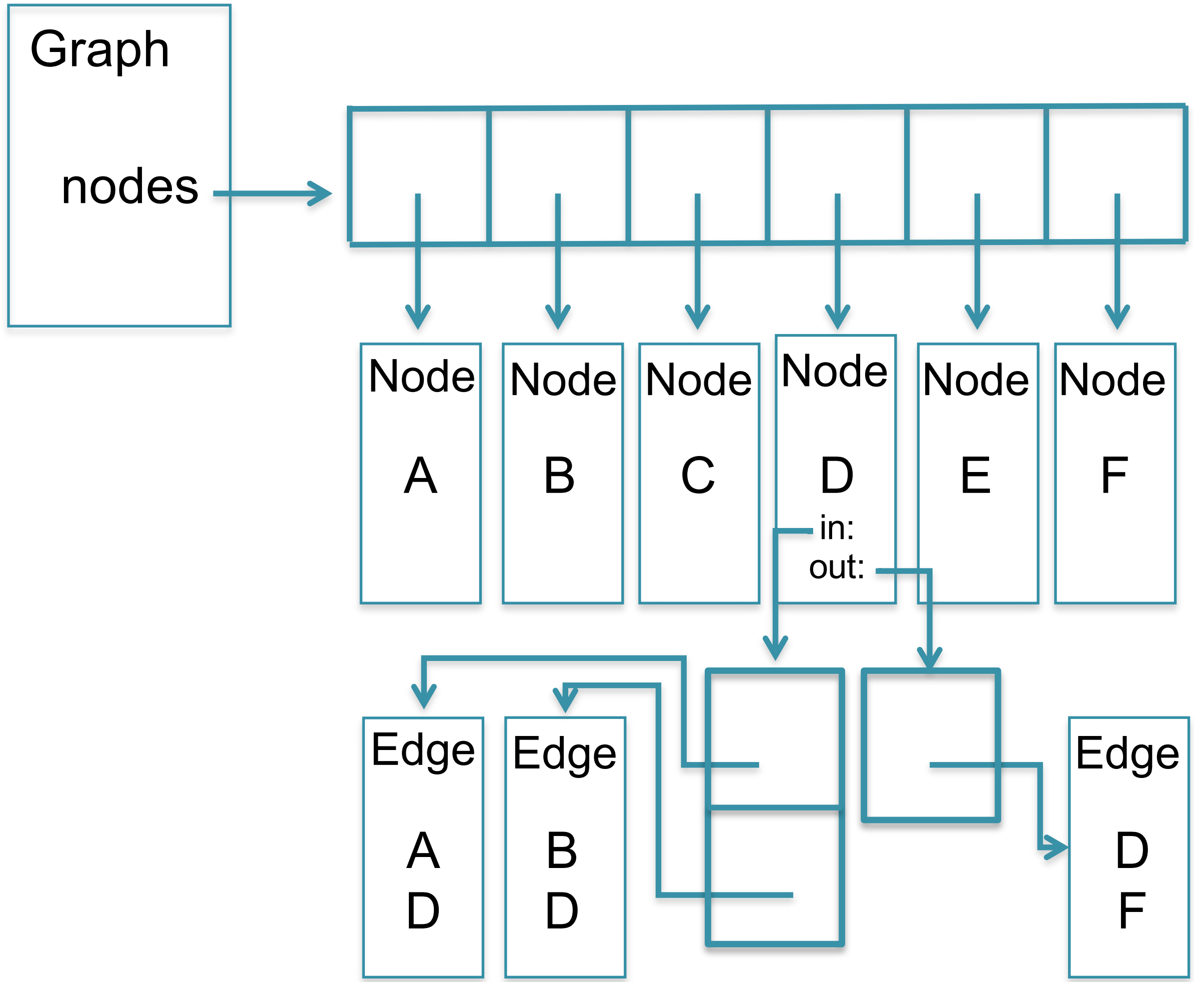
Good for sparse graphs
Compact storage

A: C, D, E
B: D, E
C: F, G
D: F
E: F
G:



Representing Graphs





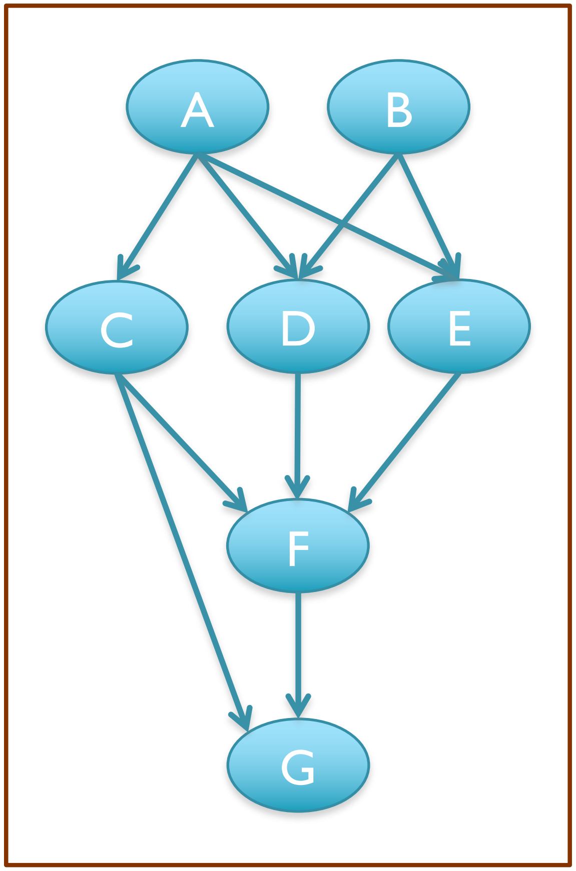
Incidence List

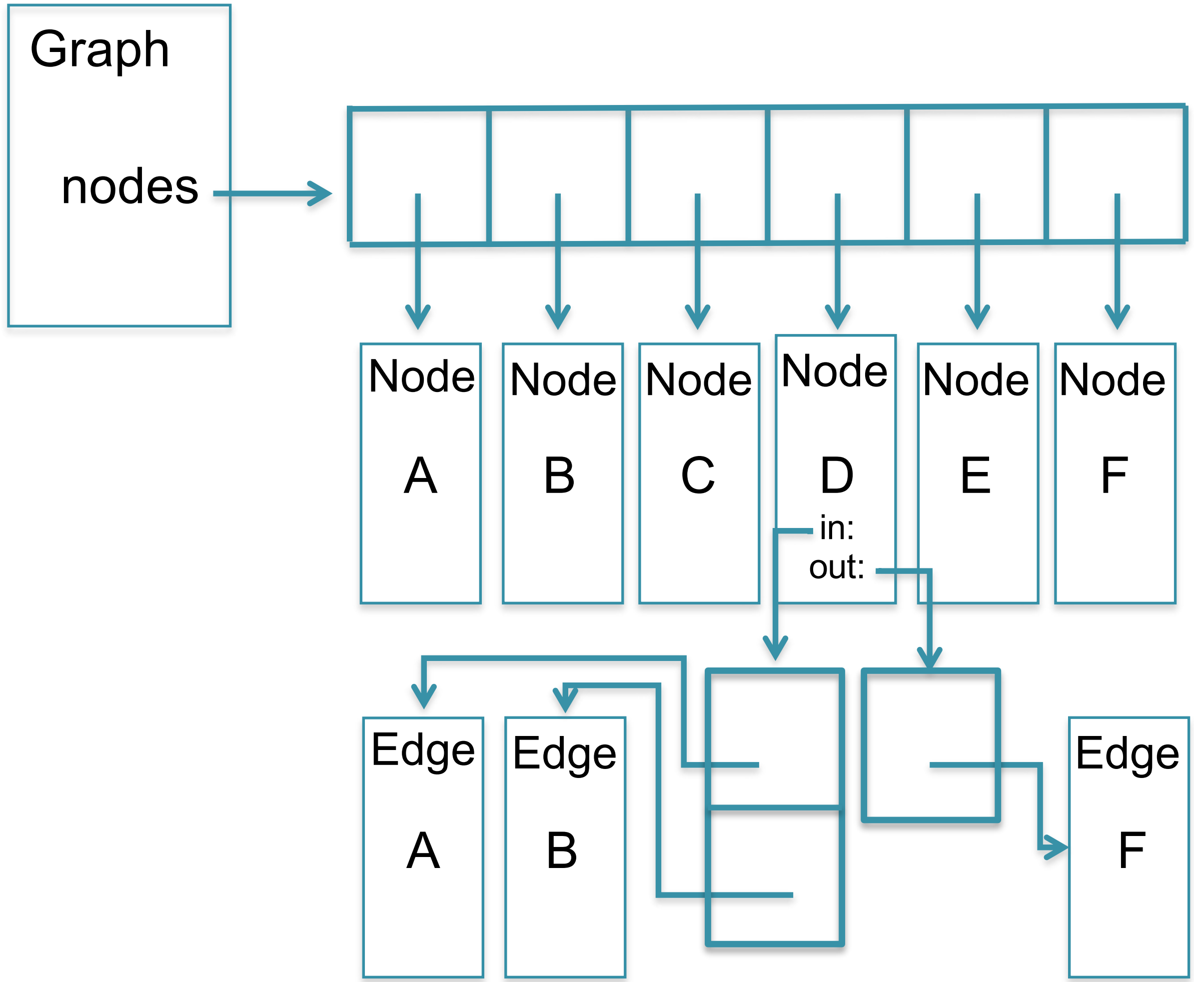
Good for sparse graphs
Compact storage

A: C, D, E
B: D, E
C: F, G
D: F
E: F
G:

Note the labels in the edges are really references to the corresponding node objects!

Representing Graphs





Incidence List

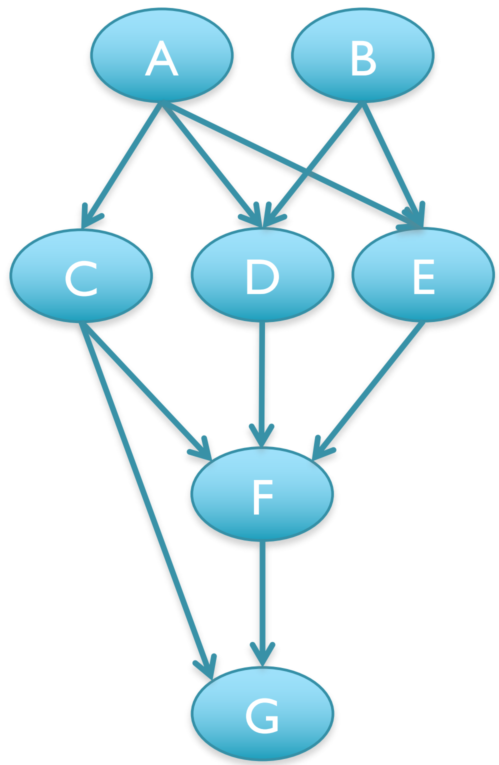
Good for sparse graphs

Compact storage

A: C, D, E
B: D, E
C: F, G
D: F
E: F
G:

Often possible for the Edge to only contain a reference to the "other" node

Representing Graphs



Incidence List
Good for sparse graphs
Compact storage

A: C, D, E
B: D, E
C: F, G
D: F
E: F
G:

Complexity Analysis

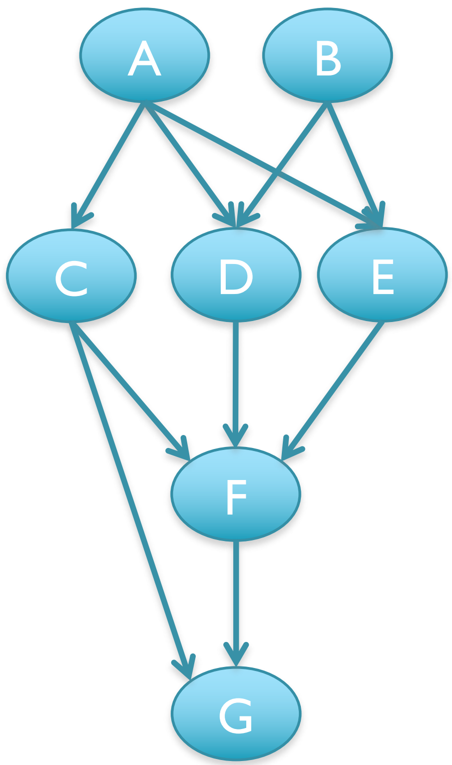
If n is the number of vertices, and m is the number of edges, we need O(n + m) space to represent the graph

When we insert a vertex, allocate one object and two empty edge lists: O(1)

When we insert an edge we allocate one object and insert the edge into appropriate lists for the incident vertices: O(1)

Remove a node?	O(1); Only after edges removed
Remove an edge?	O(d) where d is max degree; O(n) worse case
Find/check edge between nodes?	O(d); O(n) worst case

Representing Graphs



Adjacency Matrix
Good for dense graphs
Fast, Fixed storage: N^2 bits or N^2 weights

	A	B	C	D	E	F	G
A			1	1	1		
B				1	1		
C						1	1
D						1	
E						1	
F							1
G							

Incidence List
Good for sparse graphs
Compact storage: ~8 bytes/edge

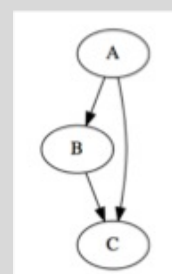
A: C, D, E
B: D, E
C: F, G
D: F
E: F
G:

Edge List
Easy, good if you (mostly) need to iterate through the edges
~16 bytes / edge

A,C B,C C,F
A,D B,D C,G
A,E B,E D,F
E,F F,G

Tools
Graphviz: http://www.graphviz.org/
Gephi: https://gephi.org/
Cytoscape: http://www.cytoscape.org/

```
digraph G {
    A->B
    B->C
    A->C
}
$ dot -Tpdf -o g.pdf g.dot
```



Next Steps

1. Work on HW4

2. Check on Piazza for tips & corrections!