CSCI 104
Graph & Tree Search and Traversals Algorithms
Mark Redekopp
David Kempe
Sandra Batista
RECURSIVE TREE TRAVERSALS
Guiding Recursive Principle

- A useful principle when trying to develop recursive solutions is that the recursive code should handle only 1 element, which might be:
  1. An element in an array
  2. A node in a linked list
  3. A node in a tree
  4. One choice in a sequence of choices

- Then use recursion to handle the remaining elements

- And finally combine the solution(s) to the recursive call(s) with the one element being handled
Recursive Tree Traversals

- A traversal iterates over all nodes of the tree
  - Usually using a depth-first, recursive approach
- Three general traversal orderings
  - Pre-order [Process root then visit subtrees]
  - In-order [Visit left subtree, process root, visit right subtree]
  - Post-order [Visit left subtree, visit right subtree, process root]

```c
// Node definition
define struct TNode
{
    int val;
    TNode *left, *right;
};

Preorder(TNode* t)
{
    if t == NULL return
    process(t) // print val.
    Preorder(t->left)
    Preorder(t->right)
}
```

```c
Inorder(TNode* t)
{
    if t == NULL return
    Inorder(t->left)
    process(t) // print val.
    Inorder(t->right)
}
```

```c
Postorder(TNode* t)
{
    if t == NULL return
    Postorder(t->left)
    Postorder(t->right)
    process(t) // print val.
}
```

Example Traversals:
- Preorder: 60 20 10 30 25 50 80
- Inorder: 10 20 25 30 50 60 80
- Postorder: 10 25 50 30 20 80 60
Example 1: Count Nodes

- Write a recursive function to **count how many nodes** are in the binary tree
  - Only process 1 node at a time
  - Determine pre-, in-, or post-order based on whose answers you need to compute the result for your node
  - For in- or post-order traversals, determine how to use/combine results from recursion on children

```c
// Node definition
struct Tnode {
    int val;
    TNode *left, *right;
};

int count(TNode* root) {
    if( root == NULL ) ____________;
    else {
        ____________;
        ____________;
    }
}
```
Example 2: Prefix Sums

- Write a recursive function to **have each node store the sum of the values on the path from the root to each node.**
  - Only process 1 node at a time
  - Determine pre-, in-, or post-order based on whose answers you need to compute the result for your node

```plaintext
void prefixH(TNode* root, int psum)  
void prefix(TNode* root)  
  { prefixH(root, 0);  
  }  
void prefixH(TNode* root, int psum)  
  { if( root == NULL ) _________________;  
    else {  
  }  
  }
```
GENERAL GRAPH TRAVERSALS
BREADTH-FIRST SEARCH
Breadth-First Search

- Given a graph with vertices, V, and edges, E, and a starting vertex that we'll call u
- BFS starts at u (‘a’ in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to every other vertex
Breadth-First Search

- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to every other vertex

Depth 0: a
Depth 1: c,e
Breadth-First Search

- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to every other vertex
Breadth-First Search

- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to every other vertex

Depth 0: a
Depth 1: c,e
Depth 2: b,d,f,g
Depth 3: h
Developing the Algorithm

• Key idea: Must explore all nearer neighbors before exploring further-away neighbors
• From 'a' we find 'e' and 'c'
  – If we explore 'e' next and find 'f' who should we choose to explore from next: 'c' or 'f'?
• Must explore all vertices at depth i before any vertices at depth i+1
  – Essentially, the first vertices we find should be the first ones we explore from
  – What data structure may help us?
Developing the Algorithm

• Exploring all vertices in the order they are found implies we will explore all vertices at shallower depth before greater depth
  – Keep a first-in / first-out queue (FIFO) of neighbors found
• Put newly found vertices in the back and pull out a vertex from the front to explore next
• We don’t want to put a vertex in the queue more than once...
  – ‘mark’ a vertex the first time we encounter it
  – only allow unmarked vertices to be put in the queue
• May also keep a ‘predecessor’ structure that indicates how each vertex got discovered (i.e. which vertex caused this one to be found)
  – Allows us to find a shortest-path back to the start vertex
Breadth-First Search

Algorithm:

BFS(G,u)
1 for each vertex v
3 Q = new Queue
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6 v = Q.front(); Q.dequeue()
7 foreach neighbor, w, of v:
8 if pred[w] == nil // w not found
9 Q.enqueue(w)
10 pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8         if pred[w] == nil // w not found
9             Q.enqueue(w)
10        pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1 for each vertex v
3 Q = new Queue
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6 v = Q.front(); Q.dequeue()
7 foreach neighbor, w, of v:
8 if pred[w] == nil // w not found
9 Q.enqueue(w)
10 pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G, u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u] = 0
5  while Q is not empty
6      v = Q.front(); Q.dequeue()
7      foreach neighbor, w, of v:
8          if pred[w] == nil // w not found
9             Q.enqueue(w)
10         pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G, u)
1 for each vertex v
3 Q = new Queue
4 Q.enqueue(u), d[u] = 0
5 while Q is not empty
6 v = Q.front(); Q.dequeue()
7 foreach neighbor, w, of v:
8 if pred[w] == nil // w not found
9 Q.enqueue(w)
10 pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u),  d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8         if pred[w] == nil // w not found
9             Q.enqueue(w)
10        pred[w] = v,  d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1 for each vertex v
3 Q = new Queue
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6 v = Q.front(); Q.dequeue()
7 foreach neighbor, w, of v:
8 if pred[w] == nil // w not found
9 Q.enqueue(w)
10 pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1. for each vertex v
3. Q = new Queue
4. Q.enqueue(u), d[u]=0
5. while Q is not empty
6. v = Q.front(); Q.dequeue()
7. foreach neighbor, w, of v:
8. if pred[w] == nil // w not found
9. Q.enqueue(w)
10. pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

**Algorithm:**

BFS(G,u)
1. for each vertex v
3. Q = new Queue
4. Q.enqueue(u), d[u]=0
5. while Q is not empty
6. v = Q.front(); Q.dequeue()
7. foreach neighbor, w, of v:
   8. if pred[w] == nil // w not found
   9. Q.enqueue(w)
10. pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

- Shortest paths can be found by walking predecessor value from any node backward

Example:
- Shortest path from a to h
  - Start at h
  - Pred[h] = b (so walk back to b)
  - Pred[b] = c (so walk back to c)
  - Pred[c] = a (so walk back to a)
  - Pred[a] = nil ... no predecessor, Done!!
Breadth-First Search Trees

- BFS (and later DFS) will induce a tree subgraph (i.e. acyclic, one parent each) from the original graph
  - BFS is tree of shortest paths from the source to all other vertices (in connected component)
Correctness

• Define
  – dist(s,v) = correct shortest distance
  – d[v] = BFS computed distance
  – p[v] = predecessor of v

• Loop invariant
  – What can we say about the nodes in the queue, their d[v] values, relationship between d[v] and dist[v], etc.?

BFS(G,u)
1 for each vertex v
2 \quad \text{pred}[v] = \text{nil}, d[v] = \text{inf}.
3 Q = \text{new Queue}
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6 \quad v = Q.front(); Q.dequeue()
7 \quad \text{foreach neighbor, w, of v:}
8 \quad \quad \text{if pred}[w] == \text{nil} \ // w \text{ not found}
9 \quad \quad \quad Q.enqueue(w)
10 \quad \text{pred}[w] = v, d[w] = d[v] + 1
Correctness

• Define
  – \text{dist}(s,v) = \text{correct shortest distance}
  – d[v] = \text{BFS computed distance}
  – p[v] = \text{predecessor of } v
• Loop invariant
  – All vertices with \text{p}[v] \neq \text{nil} (i.e. already in the queue or popped from queue) have \text{d}[v] = \text{dist}(s,v)
  – The distance of the nodes in the queue are sorted
    • If Q = \{v_1, v_2, ..., v_r\} then d[v_1] \leq d[v_2] \leq ... \leq d[v_r]
  – The nodes in the queue are from 2 adjacent layers/levels
    • i.e. d[v_k] \leq d[v_1] + 1
    • Suppose there is a node from a 3rd level (d[v_1] + 2), it must have been found by some, vi, where d[v_i] = d[v_1]+1

BFS(G,u)
1 for each vertex v
2 \text{pred}[v] = \text{nil}, \text{d}[v] = \text{inf}.
3 Q = \text{new Queue}
4 Q.\text{enqueue}(u), \text{d}[u]=0
5 while Q is not empty
6 v = Q.\text{front}(); Q.\text{dequeue}()
7 foreach neighbor, w, of v:
8 if \text{pred}[w] == \text{nil} // w not found
9 Q.\text{enqueue}(w)
10 \text{pred}[w] = v, \text{d}[w] = \text{d}[v] + 1
Breadth-First Search

• Analyze the run time of BFS for a graph with n vertices and m edges
  – Find $T(n,m)$
• How many times does loop on line 5 iterate?
• How many times loop on line 7 iterate?

BFS($G,u$)
1. for each vertex $v$
2. $\text{pred}[v] = \text{nil}$, $\text{d}[v] = \text{inf}$.
3. $Q = \text{new Queue}$
4. $Q.\text{enqueue}(u)$, $\text{d}[u]=0$
5. while $Q$ is not empty
6. $v = Q.\text{front}()$; $Q.\text{dequeue}()$
7. foreach neighbor, $w$, of $v$:
8. if $\text{pred}[w] == \text{nil}$ // $w$ not found
9. $Q.\text{enqueue}(w)$
10. $\text{pred}[w] = v$, $\text{d}[w] = \text{d}[v] + 1$
Breadth-First Search

• Analyze the run time of BFS for a graph with n vertices and m edges
  – Find T(n)

• How many times does loop on line 5 iterate?
  – N times (one iteration per vertex)

• How many times loop on line 7 iterate?
  – For each vertex, v, the loop executes \( \deg(v) \) times
    – \( \sum_{v \in V} \theta[1 + \deg(v)] \)
    – \( \theta(\sum_v 1) + \theta(\sum_v \deg(v)) \)
    – \( = \Theta(n) + \Theta(m) \)

• Total = \( \Theta(n+m) \)

```
BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u]=0
5  while Q is not empty
6      v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8         if pred[w] == nil // w not found
9             pred[w] = v, d[w] = d[v] + 1
```
Topological Search

DEPTH FIRST SEARCH MOTIVATING EXAMPLE
DFS Application: Topological Sort

- Breadth-first search doesn't solve all our problems.
- Given a graph of dependencies (tasks, prerequisites, etc.) **topological** sort creates a consistent ordering of tasks (vertices) where no dependencies are violated
- Many possible valid topological orderings exist
  - EE 109, EE 209, EE 354, EE 454, EE 457, CS104, PHYS 152, CS 201,...
  - CS 104, EE 109, CS 170, EE 209,...
Topological Sort

• Another example
  – Getting dressed

• More Examples:
  – Project management scheduling
  – Build order in a Makefile or other compile project
  – Cooking using a recipe
  – Instruction execution on an out-of-order pipelined CPU
  – Production of output values in a simulation of a combinational gate network

http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgorithms/topoSort.htm
Topological Sort

- Does breadth-first search work?
  - No. What if we started at CS 170...
  - We'd go to CS 201L before CS 104
- All parent nodes need to be completed before any child node
- BFS only guarantees *some* parent has completed before child
- Turns out a Depth-First Search will be part of our solution
Depth First Search

- Explores ALL children before completing a parent
  - Note: BFS completes a parent before ANY children
- For DFS let us assign:
  - A start time when the node is first found
  - A finish time when a node is completed
- If we look at our nodes in reverse order of finish time (i.e. last one to finish back to first one to finish) we arrive at a...
  - Topological ordering!!!
DFS Algorithm

- DFS visits and completes all children before completing (and going on to a sibling)
- Process:
  - Visit a node
  - Mark as visited (started)
  - For each visited neighbor, visit it and perform DFS on all of their children
  - Only then, mark as finished
- Let's trace recursive DFS!!
- If cycles in the graph, ensure we don’t get caught visiting neighbors endlessly
  - Use some status (textbooks use "colors" but really just some integer)
  - White = unvisited,
  - Gray = visited but not finished
  - Black = finished

DFS-All (G)
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5.   if u.color == WHITE then
6.     DFS-Visit (G, u, finish_list)
7. return finish_list

DFS-Visit (G, u)
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3.   if v.color = WHITE then
4.     DFS-Visit (G, v)
5. u.color = BLACK
6. finish_list.append(u)

Source: "Introduction to Algorithms", Cormer, Leiserson, Rivest
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)
Depth First-Search

DFS-All (G)
1  for each vertex u
2   u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5     if u.color == WHITE then
6       DFS-Visit (G, u, finish_list)
7  return finish_list

DFS-Visit (G, u,l)
1   u.color = GRAY
2  for each vertex v in Adj(u) do
3     if v.color = WHITE then
4       DFS-Visit (G, v)
5   u.color = BLACK
6  l.append(u)
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)
Depth First-Search

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)

Finish_list:
- h

**Diagram:**
- Nodes: g, b, f, d, e, c, a, h
- Edges: g→f, f→h, h→u, u→d, d→a, a→c, c→e, e→d, d→b

**DFS-Visit(G,h,l):**
- h

**DFS-Visit(G,f,l):**
- h

**DFS-Visit(G,d,l):**
- h

**DFS-Visit(G,a,l):**
- h
Depth First-Search

DFS-All \((G)\)
1. for each vertex \(u\)
2. \(u\).color = WHITE
3. finish_list = empty_list
4. for each vertex \(u\) do
5. if \(u\).color == WHITE then
6. DFS-Visit \((G, u, \text{finish_list})\)
7. return finish_list

DFS-Visit \((G, u, l)\)
1. \(u\).color = GRAY
2. for each vertex \(v\) in Adj\((u)\) do
3. if \(v\).color = WHITE then
4. DFS-Visit \((G, v)\)
5. \(u\).color = BLACK
6. \(l\).append\((u)\)
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6  DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u,l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4  DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
  h

DFS-Visit(G,g,l):
DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h, g
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h, g, f

DFS-Visit(G,f,l):
- h, g, f

DFS-Visit(G,d,l):
- h, g, f

DFS-Visit(G,a,l):
- h, g, f
**Depth First-Search**

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6.   DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3.   if v.color = WHITE then
4.     DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)

Finish_list:
- h
- g
- f
- d

DFS-Visit(G,d,l):
- d
- f
- h
- g

DFS-Visit(G,a,l):
- a
- c
- e
- b

DFS-Visit(G,a,l):
- u
- g
- f
- d
- c
- e
- b
- a
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h,
- g,
- f,
- d

DFS-Visit(G,a,l):
- h,
- g,
- f,
- d
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Vist (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h, g, f, d

DFS-Visit(G,c,l):

DFS-Visit(G,a,l):
**Depth First-Search**

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)

**Finish_list:**
- h, g, f, d

**Graph Diagram:**
- Nodes: a, b, c, d, e, f, g, h, u
- Edges: a→b, a→c, b→d, c→d, c→e, e→f, f→g, g→h, h→u

**DFS-Visit(G,e,l):**
- No edges from e to other vertices

**DFS-Visit(G,c,l):**
- No edges from c to other vertices

**DFS-Visit(G,a,l):**
- No edges from a to other vertices
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list: h, g, f, d, e

DFS-Visit(G,e,l):
DFS-Visit(G,c,l):
DFS-Visit(G,a,l):
Depth First-Search

DFS-All (G)
1  for each vertex u
2    u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5    if u.color == WHITE then
6      DFS-Visit (G, u, finish_list)
7  return finish_list

DFS-Visit (G, u, l)
1    u.color = GRAY
2    for each vertex v in Adj(u) do
3      if v.color == WHITE then
4        DFS-Visit (G, v)
5    u.color = BLACK
6    l.append(u)

Finish_list:
h, g, f, d, e, c

DFS-Visit(G,c,l):
DFS-Visit(G,a,l):
Depth First-Search

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)

Finish_list:
- h
- g
- f
- d
- e
- c
- a

DFS-Visit(G,a,l):
Depth First-Search

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)

Finish_list:
- h,
- g,
- f,
- d,
- e,
- c,
- a

DFS-Visit(G,b,l):
- May iterate through many complete vertices before finding b to launch a new search from

Diagram of graph with vertices and edges.
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6    DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3   if v.color = WHITE then
4     DFS-Visit (G, v)
5   u.color = BLACK
6 l.append(u)

Finish_list:
- h, g, f, d, e, c, a, b

DFS-Visit(G,b,l):
Depth First-Search

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)

Finish_list:
- h, g, f, d, e, c, a, b
With Cycles in the graph

ANOTHER EXAMPLE (IF TIME)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)

DFS-Visit(G,c):
DFS-Visit(G,a):
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)

DFS-Visit(G,g):
DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit(G, u, finish_list)

DFS-Visit(G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit(G, v)
5 u.color = BLACK
6 finish_list.append(u)
Toposort(G)
1 for each vertex u
2 \(\text{u.color} = \text{WHITE}\)
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == \text{WHITE} then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 \(\text{u.color} = \text{GRAY}\)
2 for each vertex v in Adj(u) do
3 if v.color = \text{WHITE} then
4 DFS-Visit (G, v)
5 \(\text{u.color} = \text{BLACK}\)
6 finish_list.append(u)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6   DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4   DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)

DFSQ:

<table>
<thead>
<tr>
<th>DFS-Visit(G,e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS-Visit(G,f)</td>
</tr>
<tr>
<td>DFS-Visit(G,g)</td>
</tr>
<tr>
<td>DFS-Visit(G,h)</td>
</tr>
<tr>
<td>DFS-Visit(G,b)</td>
</tr>
<tr>
<td>DFS-Visit(G,c)</td>
</tr>
<tr>
<td>DFS-Visit(G,a)</td>
</tr>
</tbody>
</table>
Depth First-Search

Toposort(G)
1 for each vertex u
2  u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5   if u.color == WHITE then
6     DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1  u.color = GRAY
2  for each vertex v in Adj(u) do
3    if v.color = WHITE then
4       DFS-Visit (G, v)
5  u.color = BLACK
6  finish_list.append(u)
Depth First-Search

Toposort(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
**Depth First-Search**

Toposort(G)
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. finish_list.append(u)

DFSQ:
```python
d e f g h b c
```

DFS-Visit(G,a):
Toposort(G)
1  for each vertex u
2    u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5    if u.color == WHITE then
6      DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1    u.color = GRAY
2  for each vertex v in Adj(u) do
3    if v.color = WHITE then
4      DFS-Visit (G, v)
5    u.color = BLACK
6    finish_list.append(u)

DFSQ: d e f g h b c a
ITERATIVE VERSION
Depth First-Search

DFS (G,s)
1   for each vertex u
2       u.color = WHITE
3   st = new Stack
4   st.push_back(s)
5   while st not empty
6       u = st.back()
7       if u.color == WHITE then
8           u.color = GRAY
9           foreach vertex v in Adj(u) do
10              if v.color == WHITE
11                st.push_back(v)
12       else if u.color != WHITE
13           u.color = BLACK
14       st.pop_back()
Depth First-Search

DFS (G,s)
1  for each vertex u
2    u.color = WHITE
3  st = new Stack
4  st.push_back(s)
5  while st not empty
6    u = st.back()
7    if u.color == WHITE then
8      u.color = GRAY
9      foreach vertex v in Adj(u) do
10         if v.color == WHITE
11            st.push_back(v)
12      else if u.color != WHITE
13         u.color = BLACK
14     st.pop_back()
Depth First-Search

DFS (G,s)
1   for each vertex u
2     u.color = WHITE
3   st = new Stack
4   st.push_back(s)
5   while st not empty
6     u = st.back()
7     if u.color == WHITE then
8       u.color = GRAY
9     foreach vertex v in Adj(u) do
10        if v.color == WHITE
11           st.push_back(v)
12     else if u.color != WHITE
13       u.color = BLACK
14     st.pop_back()

st: a c e c f
Depth First-Search

DFS (G, s)
1   for each vertex u
2     u.color = WHITE
3   st = new Stack
4   st.push_back(s)
5   while st not empty
6     u = st.back()
7     if u.color == WHITE then
8       u.color = GRAY
9     foreach vertex v in Adj(u) do
10        if v.color == WHITE
11           st.push_back(v)
12     else if u.color != WHITE
13       u.color = BLACK
14     st.pop_back()

st: a c e c f d g
Depth First-Search

DFS (G, s)
1  for each vertex u
2    u.color = WHITE
3  st = new Stack
4  st.push_back(s)
5  while st not empty
6    u = st.back()
7    if u.color == WHITE then
8      u.color = GRAY
9      foreach vertex v in Adj(u) do
10         if v.color == WHITE
11            st.push_back(v)
12      else if u.color != WHITE
13        u.color = BLACK
14    st.pop_back()
Depth First-Search

DFS \((G,s)\)
1. for each vertex \(u\)
2. \(u.color = WHITE\)
3. \(st = new\ Stack\)
4. \(st.push\_back(s)\)
5. while \(st\) not empty
6. \(u = st.back()\)
7. if \(u.color == WHITE\) then
8. \(u.color = GRAY\)
9. foreach vertex \(v\) in Adj\((u)\) do
10. if \(v.color == WHITE\) then
11. \(st.push\_back(v)\)
12. else if \(u.color != WHITE\) then
13. \(u.color = BLACK\)
14. \(st.pop\_back()\)
Depth First-Search

DFS (G,s)
1. for each vertex u
2. u.color = WHITE
3. st = new Stack
4. st.push_back(s)
5. while st not empty
6. u = st.back()
7. if u.color == WHITE then
8. u.color = GRAY
9. foreach vertex v in Adj(u) do
10. if v.color == WHITE
11. st.push_back(v)
12. else if u.color != WHITE
13. u.color = BLACK
14. st.pop_back()
Depth First-Search

DFS (G,s)
1  for each vertex u
2    u.color = WHITE
3  st = new Stack
4  st.push_back(s)
5  while st not empty
6    u = st.back()
7    if u.color == WHITE then
8      u.color = GRAY
9      foreach vertex v in Adj(u) do
10         if v.color == WHITE
11            st.push_back(v)
12    else if u.color != WHITE
13      u.color = BLACK
14    st.pop_back()

st: a c e c f d g c h b c d
Depth First-Search

DFS (G,s)
1  for each vertex u
2    u.color = WHITE
3  st = new Stack
4  st.push_back(s)
5  while st not empty
6    u = st.back()
7    if u.color == WHITE then
8      u.color = GRAY
9      foreach vertex v in Adj(u) do
10         if v.color == WHITE
11            st.push_back(v)
12      else if u.color != WHITE
13        u.color = BLACK
14    st.pop_back()
Depth First-Search

DFS (G,s)

1. for each vertex u
2. \( u.color = \text{WHITE} \)
3. st = new Stack
4. st.push_back(s)
5. while st not empty
6. \( u = \text{st.back}() \)
7. if \( u.color == \text{WHITE} \) then
8. \( u.color = \text{GRAY} \)
9. foreach vertex v in Adj(u) do
10. if v.color == WHITE
11. \( \text{st.push_back(v)} \)
12. else if u.color != WHITE
13. \( u.color = \text{BLACK} \)
14. \( \text{st.pop_back()} \)
DFS (G,s)
1. for each vertex u
2. u.color = WHITE
3. st = new Stack
4. st.push_back(s)
5. while st not empty
6. u = st.back()
7. if u.color == WHITE then
8. u.color = GRAY
9. foreach vertex v in Adj(u) do
10. if v.color == WHITE
11. st.push_back(v)
12. else if u.color != WHITE
13. u.color = BLACK
14. st.pop_back()
Depth First-Search

DFS (G,s)
1  for each vertex u
2    u.color = WHITE
3  st = new Stack
4  st.push_back(s)
5  while st not empty
6    u = st.back()
7    if u.color == WHITE then
8      u.color = GRAY
9      foreach vertex v in Adj(u) do
10         if v.color == WHITE
11            st.push_back(v)
12      else if u.color != WHITE
13         u.color = BLACK
14    st.pop_back()
Depth First-Search

DFS (G,s)
1  for each vertex u
2    u.color = WHITE
3  st = new Stack
4  st.push_back(s)
5  while st not empty
6    u = st.back()
7    if u.color == WHITE then
8      u.color = GRAY
9      foreach vertex v in Adj(u) do
10         if v.color == WHITE
11             st.push_back(v)
12         else if u.color != WHITE
13             u.color = BLACK
14        st.pop_back()
BFS vs. DFS Algorithm

- BFS and DFS are more similar than you think
  - Do we use a FIFO/Queue (BFS) or LIFO/Stack (DFS) to store vertices as we find them

```plaintext
BFS-Visit (G, start_node)
1   for each vertex u
2       u.color = WHITE
3       u.pred = nil
4   bfsq = new Queue
5   bfsq.push_back(start_node)
6   while bfsq not empty
7       u = bfsq.pop_front()
8       if u.color == WHITE
9           u.color = GRAY
10      foreach vertex v in Adj(u) do
11         bfsq.push_back(v)
```

```plaintext
DFS-Visit (G, start_node)
1   for each vertex u
2       u.color = WHITE
3       u.pred = nil
4   st = new Stack
5   st.push_back(start_node)
6   while st not empty
7       u = st.top(); st.pop()
8       if u.color == WHITE
9           u.color = GRAY
10      foreach vertex v in Adj(u) do
11         st.push_back(v)
```
SOLUTIONS
Example 1: Count Nodes

• Write a recursive function to **count how many nodes** are in the binary tree
  – Only process 1 node at a time
  – Determine pre-, in-, or post-order based on whose answers you need to compute the result for your node
  – For in- or post-order traversals, determine how to use/combine results from recursion on children

```c
// Node definition
struct Tnode {
    int val;
    TNode *left, *right;
};

int count(TNode* root) {
    if (root == NULL) return 0;
    else {
        return 1 + count(root->left) + count(root->right);
    }
}
```
Example 2: Prefix Sums

- Write a recursive function to **have each node store the sum of the values on the path from the root to each node.**
  - Only process 1 node at a time
  - Determine pre-, in-, or post-order based on whose answers you need to compute the result for your node

```c
void prefixH(TNode* root, int psum)
void prefix(TNode* root)
{
    prefixH(root, 0);
}
void prefixH(TNode* root, int psum)
{
    if (root == NULL) return;
    else {
        root->val += psum;
        prefixH(root->left, root->val);
        prefixH(root->right, root->val);
    }
}
```