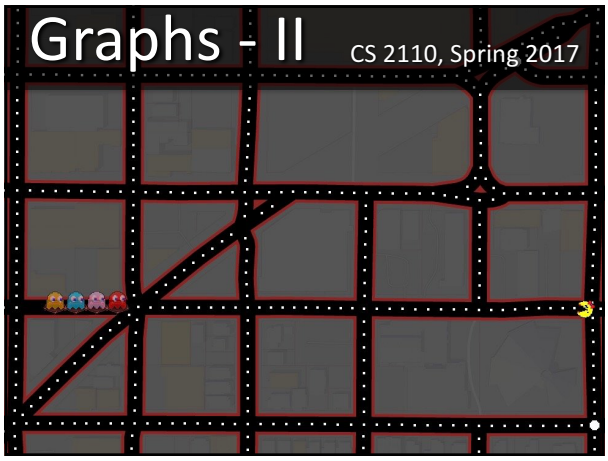
Graphs - II

CS 2110, Spring 2017



Announcements

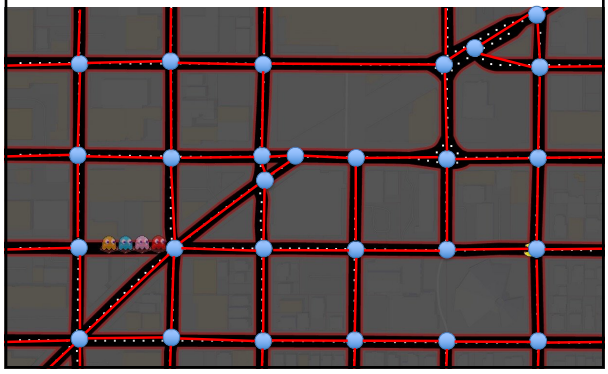
1. **Lunch** with instructors: Still many slots open!
   - See pinned Piazza note @275
2. **Hidden Figures**: Thursday at 8:30 and 11:00!
   - If you haven't signed up on the CMS schedule do so now!
   - so far: 8:30 114/330
     11:00 147/330

Announcements

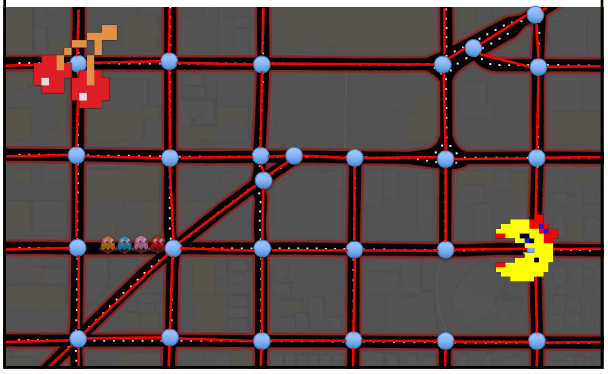
3. For next Tuesday's lecture, you **MUST** watch the tutorial on the shortest path algorithm beforehand:
   http://www.cs.cornell.edu/courses/cs2110/2017sp/online/shortestPath/shortestPath.html
   - Tuesday's lecture **will assume** that you understand it. Watch the tutorial once or twice and execute the algorithm on a small graph.
4. Information about **Prelim 2** is now on the Exams page of the course website and P2Conflict will be available on the CMS today (Tuesday).

Look, a graph!

Look, a graph!



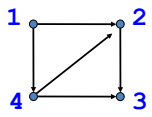
Look, a graph!



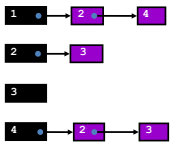
Graph Algorithms

- Search
  - Depth-first search
  - Breadth-first search
- Shortest paths
  - Dijkstra's algorithm
- Minimum spanning trees
  - Prim's algorithm
  - Kruskal's algorithm

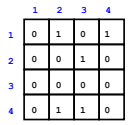
Representations of Graphs



Adjacency List



Adjacency Matrix



	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

Adjacency Matrix or Adjacency List?

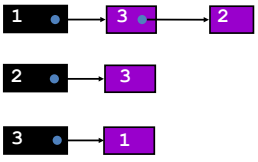
- Definitions:
  - n = number of vertices
  - e = number of edges
  - $d(u)$ = degree of u = number of edges leaving u
- Adjacency Matrix
  - Uses space $O(n^2)$
  - Can iterate over all edges in time $O(n^2)$
  - Can answer "Is there an edge from u to v?" in $O(1)$ time
  - Better for dense graphs (lots of edges)
- Adjacency List
  - Uses space $O(e + n)$
  - Can iterate over all edges in time $O(e + n)$
  - Can answer "Is there an edge from u to v?" in $O(d(u))$ time
  - Better for sparse graphs (fewer edges)

Breaking DAG

Which of the following two graphs are DAGs?

Directed Acyclic Graph

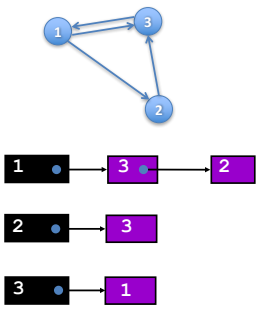
Graph 1:

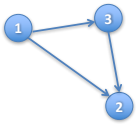


Graph 2:

	1	2	3
1	0	1	1
2	0	0	0
3	0	1	0

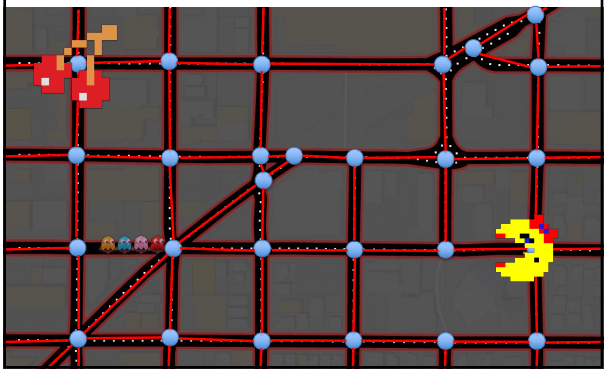
Breaking DAG





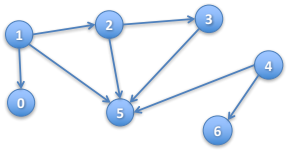
	1	2	3
1	0	1	1
2	0	0	0
3	0	1	0

Back to Important Things:



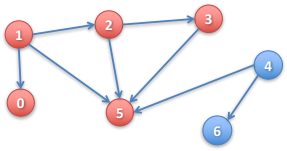
Depth-First Search

- Given a graph and one of its nodes u
  (say node 1 below)



Depth-First Search

- Given a graph and one of its nodes u
  (say node 1 below)
- We want to "visit" each node reachable from u
  (nodes 1, 0, 2, 3, 5)



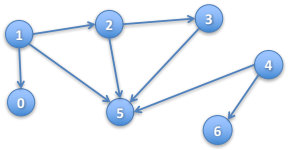
There are many paths to some nodes.

How do we visit all nodes efficiently, without doing extra work?

Depth-First Search

boolean[ ] visited;

- Node u is visited means: visited[u] is true
- To visit u means to: set visited[u] to true
- v is REACHABLE* from u if there is a path (u, …, v)
  *in which all nodes of the path are unvisited.

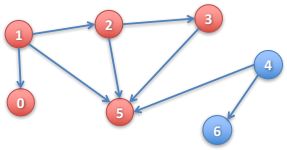


Suppose all nodes are unvisited.

Depth-First Search

boolean[ ] visited;

- Node u is visited means: visited[u] is true
- To visit u means to: set visited[u] to true
- v is REACHABLE* from u if there is a path (u, …, v)
  *in which all nodes of the path are unvisited.



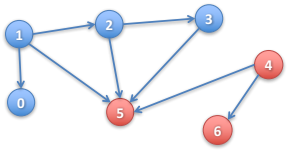
Suppose all nodes are unvisited.

Nodes REACHABLE* from node **1**:
{1, 0, 2, 3, 5}

Depth-First Search

boolean[ ] visited;

- Node u is visited means: visited[u] is true
- To visit u means to: set visited[u] to true
- v is REACHABLE* from u if there is a path (u, …, v)
  *in which all nodes of the path are unvisited.



Suppose all nodes are unvisited.

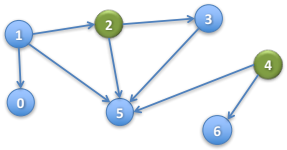
Nodes REACHABLE* from node **1**:
{1, 0, 2, 3, 5}

Nodes REACHABLE* from **4**: {4, 5, 6}

Depth-First Search

boolean[ ] visited;

- Node u is visited means: visited[u] is true
- To visit u means to: set visited[u] to true
- v is REACHABLE* from u if there is a path (u, …, v)
  *in which all nodes of the path are unvisited.

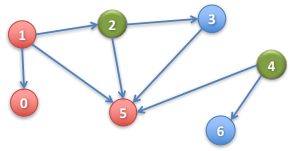


Green: visited
Blue: unvisited

Depth-First Search

boolean[ ] visited;

- Node u is visited means: visited[u] is true
- To visit u means to: set visited[u] to true
- v is REACHABLE* from u if there is a path (u, …, v)
  *in which all nodes of the path are unvisited.



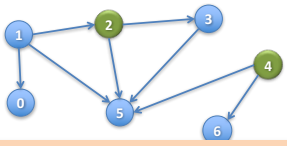
Green: visited
Blue: unvisited

Nodes REACHABLE* from node **1**: {1, 0, 5}

Depth-First Search

boolean[ ] visited;

- Node u is visited means: visited[u] is true
- To visit u means to: set visited[u] to true
- v is REACHABLE* from u if there is a path (u, …, v)
  *in which all nodes of the path are unvisited.



Green: visited
Blue: unvisited

Nodes REACHABLE * from node **1**: {1, 0, 5}

Nodes REACHABLE * from **4**: none

Not even **4** itself, because it's already been visited!

Depth-First Search

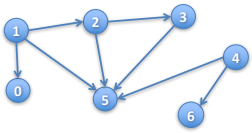
```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {


}
```

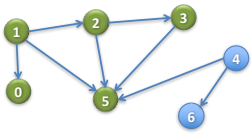
Let u be **1**

The nodes REACHABLE* from 1 are 1, 0, 2, 3, 5

Start



End



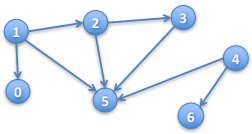
Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {


}
```

Let u be **1**

The nodes REACHABLE* from 1 are 1, 0, 2, 3, 5



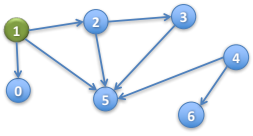
Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;


}
```

Let u be **1**

The nodes REACHABLE* from 1 are 1, 0, 2, 3, 5



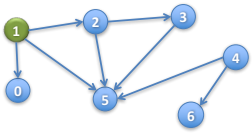
Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;


}
```

Let u be **1** (visited)

The nodes to be visited are 0, 2, 3, 5



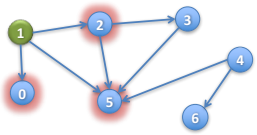
Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

Let u be **1** (visited)

The nodes to be visited are 0, 2, 3, 5

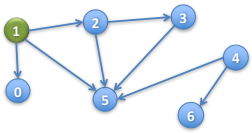
Have to do DFS on all unvisited neighbors of u!



Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

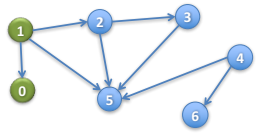
Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1 …



Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

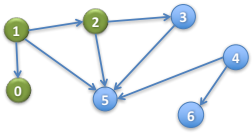
Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1, 0 …



Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

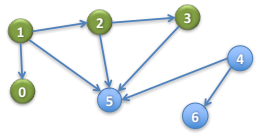
Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1, 0, 2 …



Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

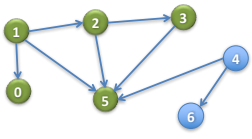
Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1, 0, 2, 3 …



Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1, 0, 2, 3, 5



Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

Suppose n nodes are REACHABLE* along e edges (in total). What is
- Worst-case runtime? O(n+e)
- Worst-case space? O(n)

Depth-First Search

```
/** Visit all nodes that are REACHABLE*
from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

That's all there is to basic DFS. You may have to change it to fit a particular situation.

If you don't have this spec and you do something different, it's probably wrong.

Example: Use different way (other than array visited) to know whether a node has been visited

Example: We really haven't said what data structures are used to implement the graph

Depth-First Search in OO fashion

```
public class Node {
    boolean visited;
    List<Node> neighbors;

    /** Visit all nodes that are REACHABLE*
       * from u. Precondition: u is unvisited */
    public void dfs() {
        visited= true;
        for (Node n: neighbors) {
            if (!n.visited) n.dfs();
        }
    }
}
```

Each node of the graph is an object of type Node

No need for a parameter. The object is the node.

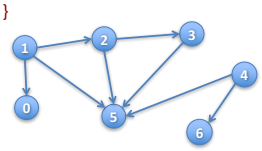
Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);   // Not Java!
    // inv: all nodes that have to be visited are
    //       REACHABLE* from some node in s
    while ( s is not empty ) {
        u= s.pop();   // Remove top stack node, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1)



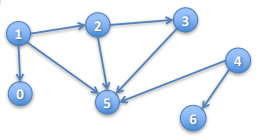
1

Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 0



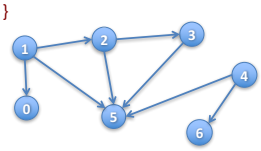
1

Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 0

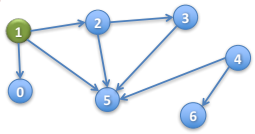


Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 0

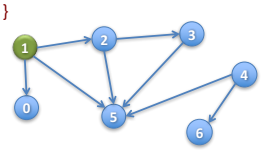


Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited.  */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 0

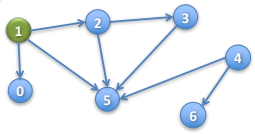


0
2
5
Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 1

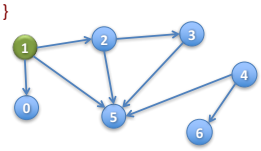


0
2
5
Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited.*/
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 1

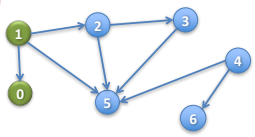


2
5
Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 1

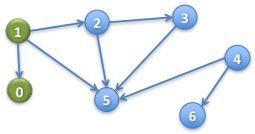


2
5
Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 2

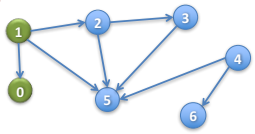


2
5
Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 2

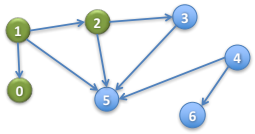


5
Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 2



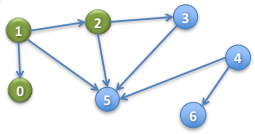
5
Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. . */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1) Iteration 2

Yes, 5 is put on the stack twice, once for each edge to it. It will be visited only once.



3
5
5
Stack s

Depth-First Search written iteratively

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. . */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

That's DFS!

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);   // Not Java!
    // inv: all nodes that have to be visited are
    //      REACHABLE* from some node in s
    while ( s is not empty ) {
        u= s.pop();   // Remove top stack node, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Want to see a magic trick?

Depth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);   // Not Java!
    // inv: all nodes that have to be visited are
    //      REACHABLE* from some node in s
    while ( s is not empty ) {
        u= s.pop();   // Remove top stack node, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);   // Not Java!
    // inv: all nodes that have to be visited are
    //      REACHABLE* from some node in s
    while ( q is not empty ) {
        u= q.popFirst();   // Remove first node in queue, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);   // Add to end of queue
        }
    }
}
```

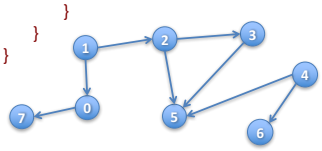
Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);   // Not Java!
    // inv: all nodes that have to be visited are
    //      REACHABLE* from some node in s
    while ( q is not empty ) {
        u= q.popFirst();   // Remove first node in queue, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);   // Add to end of queue
        }
    }
}
```

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1)



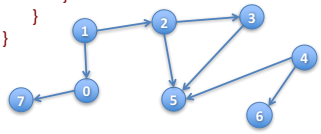
1

Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 0



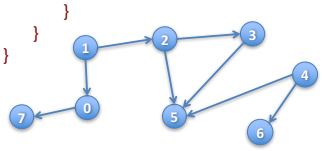
1

Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 0

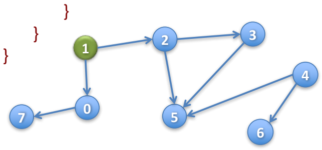


Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 0

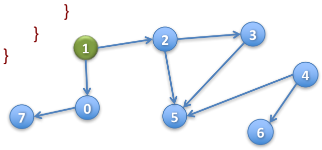


Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 0

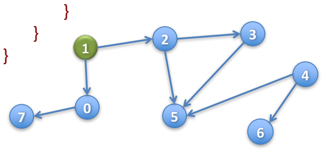


0 2
Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 1

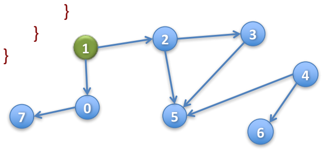


0 2
Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 1

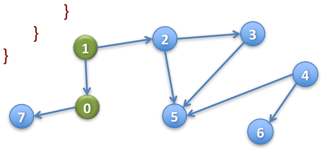


2
Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 1

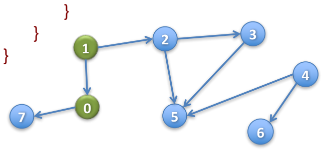


2
Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 1

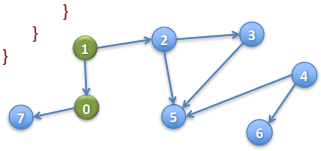


2 7
Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. . */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 2

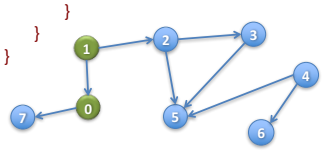


2 7
Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. . */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 2

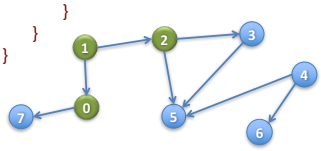


7
Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 2



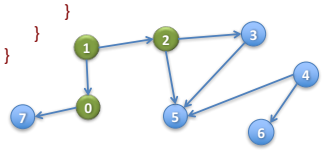
7
Queue q

Breadth-First Search

```
/** Visit all nodes REACHABLE* from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);
    while q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1) Iteration 2

Breadth first:
(1) Node u
(2) All nodes 1 edge from u
(3) All nodes 2 edges from u
(4) All nodes 3 edges from u
…



7 3 5
Queue q

Some food for thought:

- BFS(root) on a tree corresponds to which tree traversal?
- Write out the order nodes are visited in this undirected graph, when calling:
  - BFS(5)
  - DFS(5)
  - DFS(0)

(if there are ties, visit the lower # first)