Announcements

• Quiz today - same as usual

• A4: Dijkstra’s Single Source Shortest Paths
  – We’ll cover Dijkstra’s algorithm Wednesday
  – Assignment released Wednesday
  – Due a week later, Wednesday 6/3

• Basic graph operation runtime - see end of Wednesday’s slides.
Goals

• Understand and be able to implement graph traversal/search algorithms:
  – Depth-first search
  – Breadth-first search
Look, a graph!
Look, a graph!
Look, a graph!
Look, a graph!
DAG

• A commonly-used flavor of graph: Directed Acyclic Graph (DAG).
• Definition: A graph that is directed and acyclic.
Which of the following two graphs are DAGs?

Graph 1:

Graph 2:
Breaking DAG

No DAG

DAG

1 → 3 → 2

1 → 3

2 → 3

3 → 1

0 1 1

0 0 0

0 1 0
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)
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;
Depth-First Search

```java
boolean[] visited;

• Node u is visited means: visited[u] is true
• To visit u means to: set visited[u] to true
```
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 explorable from u if there is a path (u, ..., v) in which all nodes of the path are unvisited.
boolean[ ] visited;

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

Suppose all nodes are unvisited.

Nodes explorable 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 exploratable from u if there is a path (u, ..., v) in which all nodes of the path are unvisited.

Suppose all nodes are unvisited.

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

Nodes exploratable 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 explorable 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 explorable from u if there is a path (u, ..., v)
  in which all nodes of the path are unvisited.

Green: visited
Blue: unvisited

Nodes explorable from node 1: {1, 0, 5}
Depth-First Search

boolean[ ] visited;

- Node \( u \) is visited means: \( \text{visited}[u] \) is true
- To visit \( u \) means to: set \( \text{visited}[u] \) to true
- \( v \) is explorable from \( u \) if there is a path \((u, \ldots, v)\)
in which all nodes of the path are unvisited.

**Green**: visited

**Blue**: unvisited

Nodes explorable from node 1:
\{1, 0, 5\}

Nodes explorable from 4: none
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 exploriable from u if there is a path (u, ..., v)
in which all nodes of the path are unvisited.

Green: visited
Blue: unvisited

Nodes exploriable from node 1:
{1, 0, 5}

Nodes exploriable from 4: none

Not even 4 itself, because it’s already been visited.
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

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

public static void dfs(int u) {

}
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
public static void dfs(int u) {
}
```

Let u be 1

The nodes explorable from 1 are 1, 0, 2, 3, 5
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

public static void dfs(int u) {

Let u be 1

The nodes explorable from 1 are 1, 0, 2, 3, 5
Depth-First Search

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

Let u be 1
The nodes explorable from 1 are 1, 0, 2, 3, 5
/** Visit all nodes that are explorable 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
/** Visit all nodes that are explorable 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

    Have to do DFS on all unvisited neighbors of u!
/** Visit all nodes that are explorable 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!
/** Visit all nodes that are explorable 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);
}
/** Visit all nodes that are explorable 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 ...
/** Visit all nodes that are explorable 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 ...
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
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 ...
/** Visit all nodes that are explorable 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 ...
/** Visit all nodes that are explorable 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);
}
/** Visit all nodes that are explorable 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);
}
/** Visit all nodes that are explorable 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 explorable along $e$ edges (in total). What is
• Worst-case runtime? $O(n+e)$
• Worst-case space? $O(n)$
/** Visit all nodes that are explorabale from u. Precondition: u is unvisited. */

```java
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.
Depth-First Search

/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

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

That’s all there is to basic DFS. You may have to change it to fit a particular situation.
Depth-First Search

/** Visit all nodes that are explorable 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);
}

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.

That’s all there is to basic DFS. You may have to change it to fit a particular situation.
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

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.

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.
Depth-First Search in OO fashion

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

Each node of the graph is an object of type Node.
Depth-First Search in OO fashion

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

    /** Visit all nodes that are explorable
     * from u. Precondition: u is unvisited */
    public void dfs() {
        visited= true;
    }
}
```

Each node of the graph is an object of type Node.
Depth-First Search in OO fashion

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

    /** Visit all nodes that are explorable
     * from u. Precondition: u is unvisited */
    public void dfs() {
        visited = true;
    }
}
```

Each node of the graph is an object of type Node

No need for a parameter. The object is the node.
public class Node {
    boolean visited;
    List<Node> neighbors;

    /** Visit all nodes that are explorable from u. Precondition: u is unvisited */
    public void dfs() {
        visited = true;
        for (Node n: neighbors) {
            if (!n.visited) n.dfs();
        }
    }
}
public class Node {
    boolean visited;
    List<Node> neighbors;

    /** Visit all nodes that are explorable
     * from u. Precondition: u is unvisited */
    public void dfs() {
        visited = true;
        for (Node n: neighbors) {
            if (!n.visited) n.dfs();
        }
    }
}
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID);  // Not Java!
    // inv: all nodes that have to be visited are
    //       explorable from some node in s
    while ( ) {
    }
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID); // Not Java!
    // inv: all nodes that have to be visited are
    //      explorable from some node in s
    while (s is not empty) {
        // Code...
    }
}
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID);  // Not Java!
    // inv: all nodes that have to be visited are
    //     explorable 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) {
            // Further code...
        }
    }
}
Depth-First Search written iteratively

```java
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID);  // Not Java!
    // inv: all nodes that have to be visited are
    //      explorable 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;
        }
    }
}
```
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID);  // Not Java!
    // inv: all nodes that have to be visited are
    //      explorable 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 explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID);    // Not Java!
    // inv: all nodes that have to be visited are
    //      explorable 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);
        }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID); // Not Java!
    // inv: all nodes that have to be visited are
    //       explorable 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);
        }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID);  // Not Java!
    // inv: all nodes that have to be visited are
    //       explorable 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 explorable 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);
        }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();
    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);
        }
    }
}
/** Visit all nodes explorable 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 explorable 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
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s = new Stack();
    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
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u =  s . p o p () ;
        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

0
2
5
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```
public static void dfs(int u) {
    Stack s = new Stack();
    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

Stack s

0
2
5

```
/** Visit all nodes explorables 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

Stack s

2
5

2 5
/** Visit all nodes explorable 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

Stack s
Depth-First Search written iteratively

/** Visit all nodes explorable 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

Stack s

2 5

Call dfs(1)
/** Visit all nodes explorable 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);
        }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s = new Stack();
    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

Stack s
/** Visit all nodes explorable 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

Stack s

1
2
3
4
5
6
/** Visit all nodes explorable 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.

Stack s

3
5
5
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u =  s . p o p ( ) ;
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s . p u s h ( v ) ;
        }
    }
}

Call dfs(1)

Iteration 3

Stack s

3
5
5
/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();
    while (s is not empty) {
        int 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 3

Stack s
Depth-First Search written iteratively

/** Visit all nodes explorable 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 3

Stack s
/* Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s = new Stack();
    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 3

Stack s
/** Visit all nodes explorables from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s = new Stack();
    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 4

Stack s
Depth-First Search written iteratively

/** Visit all nodes explorable 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)
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();
    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 4

Stack s

5
5
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
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 5

Stack `s`
/** Visit all nodes explorable 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 6

Stack s
/** Visit all nodes explorables from u. Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s = new Stack();
    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!

```java
/** Visit all nodes explorable 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
    //      explorable 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);
        }
    }
}
```
That’s DFS!

/** Visit all nodes explorable 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
    // explorable 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 explorable 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
    //      explorable 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);
        }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();  // Not Java!
    // inv: all nodes that have to be visited are
    //      explorable 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);
        }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void bfs(int u) {
    Queue q = "u"; // Not Java!
    // inv: all nodes that have to be visited are explorable 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 explorable 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
    // explorable 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
        }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
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)
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q = new Queue(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);
    }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void bfs(int u) {
    Queue q = new Queue(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);
        }
    }
}
Breadth-First Search

/** Visit all nodes explorable 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 exploriable 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)

Queue q
/** Visit all nodes explorable 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);
        }
    }
}
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q = new (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);
        }
    }
}
Breadth-First Search

/** Visit all nodes explorable 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);
        }
    }
}
/** Visit all nodes explorabale from u. Pre: u is unvisited. */

```java
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);
        }
    }
}
```
Breadth-First Search

/** Visit all nodes explorable 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);
        }
    }
}
```java
/** Visit all nodes explorable 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);
        }
    }
}
```
** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
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);
        }
    }
}
```

Breadth-First Search

Call bfs(1)

Iteration 2

Queue q

1 2 3 4 5 6 7
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q = new Queue(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);
        }
    }
}
Breadth-First Search

/** Visit all nodes explorable 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 3 5
Queue q
Breadth-First Search

/** Visit all nodes explorable 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
...

Queue q

7 3 5
Some working code for DFS

- [https://codeboard.io/projects/97448](https://codeboard.io/projects/97448)
- Sample graph constructed by the code:

- Suggested exercises:
  - Run DFS by hand
  - Run BFS by hand
  - Code BFS
Questions to Ponder

• 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)