Middlebury

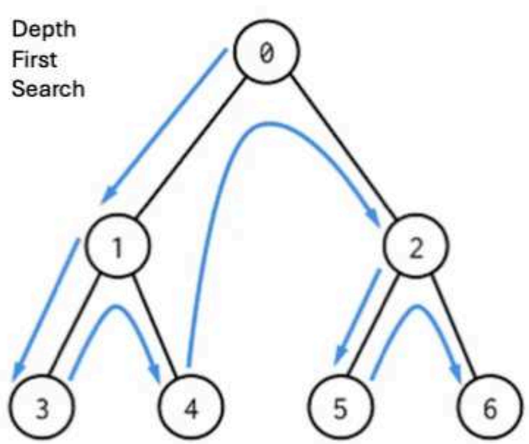
CSCI 201: Data Structures

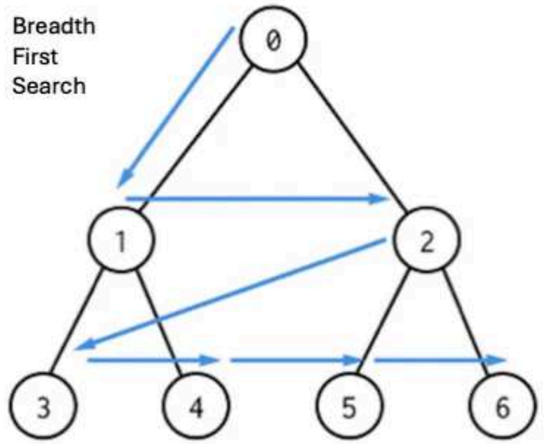
Spring 2025

Lecture 11F: Graph Searching

Goals for today:

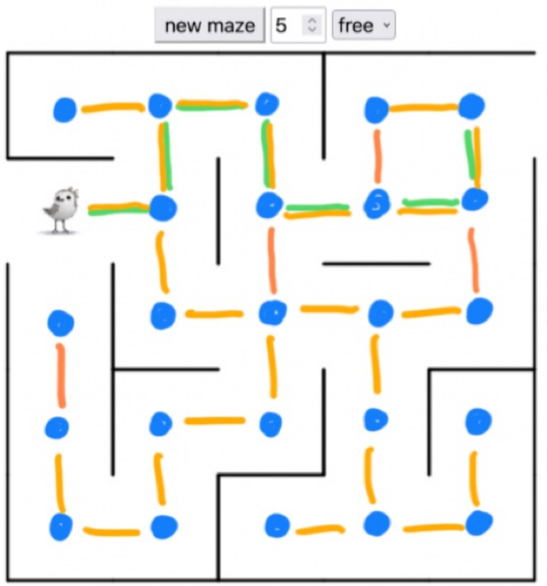
- Visit nodes in a graph using Depth-First Search (DFS).
- Visit nodes in a graph using Breadth-First Search (BFS).
- Implement DFS and BFS.
- Start Homework 10.





Very common topics in tech interviews.

A maze game:



- nodes
— edges
— shortest solution path

Click on "game" in the row for today's class at go/cs201

What is the relationship to graphs? What are the nodes? What are the edges?

Let's revisit our `getEdges` method from last class.

```java
public class Graph<Node> {

  Set<Edge> getEdges() {
    Set<Edge> edges = new HashSet<>();

    // loop through all nodes in the graph
    for (Node u : adj.keySet()) {

      // loop through all nodes adjacent to node u
      for (Node v : adj.get(u)) {

        // don't double-count this edge
        Edge edge = new Edge(u, v);
        if (!edges.contains(edge)) {
          edges.add(edge);
        }
      }
    }
    return edges;
  }
}
```

```java
public class Graph<Node> {

  List<Edge> getEdges() {
    List<Edge> edges = new ArrayList<>();

    // loop through all nodes in the graph
    for (Node u : adj.keySet()) {

      // loop through all nodes adjacent to node u
      for (Node v : adj.get(u)) {

        // don't double-count this edge
        Edge edge = new Edge(u, v);
        if (!edges.contains(edge)) {
          edges.add(edge);
        }
      }
    }
    return edges;
  }
}
```

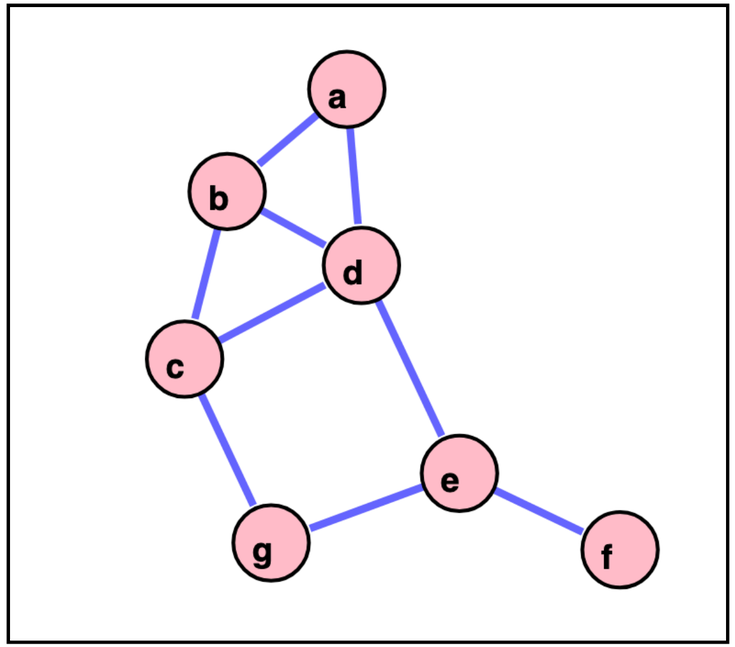
```java
// means we needed
class Edge {

  public int hashCode() {
    // edges (u, v) and (v, u)
    // should have the same hash table index
    ...
  }

  public boolean equals(Object otherObj) {
    // edges (u, v) and (v, u)
    // should be considered equal
    ...
  }
}
```

```java
// then we just need
class Edge {

  public boolean equals(Object otherObj) {
    // edges (u, v) and (v, u)
    // should be considered equal
    ...
  }
}
```

Another option if `Node` is `Comparable`.

```java
public class Graph<Node extends Comparable<Node>> {

  List<Edge> getEdges() {
    List<Edge> edges = new ArrayList<>();

    // loop through all nodes in the graph
    for (Node u : adj.keySet()) {

      // loop through all nodes adjacent to node u
      for (Node v : adj.get(u)) {

        // since adj(u) stores v
        // and adj(v) stores u
        if (u.compareTo(v) < 0) {
          edges.add(new Edge(u, v));
        }
      }
    }
    return edges;
  }
}
```

```java
// then we just need
class Edge {
  public Node u;
  public Node v;

  public Edge(Node u, Node v) {
    this.u = u;
    this.v = v;
  }
}
```



Consider three variants for storing adjacent nodes.

What is the complexity of checking if a node u is adjacent to v?

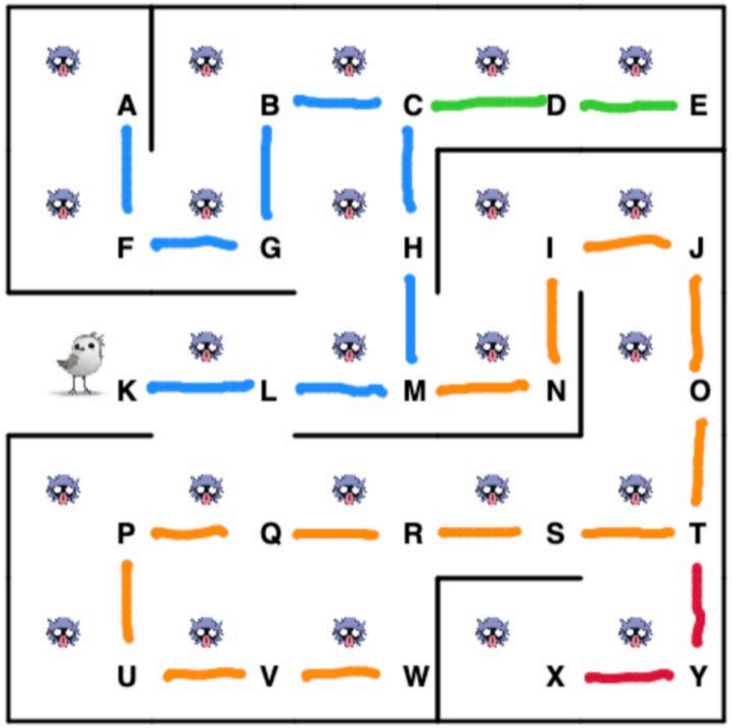
```java
public class Graph<Node> {

  HashMap<Node, TreeSet<Node>> adj;

  void addEdge(Node a, Node b) {
    if (!adj.containsKey(a)) {
      adj.put(a, new TreeSet<>());
    }
    if (!adj.containsKey(b)) {
      adj.put(b, new TreeSet<>());
    }
    adj.get(a).add(b);
    adj.get(b).add(a);
  }

  boolean areAdjacent(Node a, Node b) {
    return adj.get(a).contains(b);
  }
}
```

```java
public class Graph<Node> {

  HashMap<Node, HashSet<Node>> adj;

  void addEdge(Node a, Node b) {
    if (!adj.containsKey(a)) {
      adj.put(a, new HashSet<>());
    }
    if (!adj.containsKey(b)) {
      adj.put(b, new HashSet<>());
    }
    adj.get(a).add(b);
    adj.get(b).add(a);
  }

  boolean areAdjacent(Node a, Node b) {
    return adj.get(a).contains(b);
  }
}
```

```java
public class Graph<Node> {

  HashMap<Node, ArrayList<Node>> adj;

  void addEdge(Node a, Node b) {
    if (!adj.containsKey(a)) {
      adj.put(a, new ArrayList<>());
    }
    if (!adj.containsKey(b)) {
      adj.put(b, new ArrayList<>());
    }
    adj.get(a).add(b);
    adj.get(b).add(a);
  }

  boolean areAdjacent(Node a, Node b) {
    return adj.get(a).contains(b);
  }
}
```

Depth-First Search ("backtracking")

- **Main idea:** Keep traversing edges until you "hit a wall," then go back to parent.
- Don't step into nodes we already visited.
- Resulting set of edges forms a **tree**: **connected** and **acyclic**

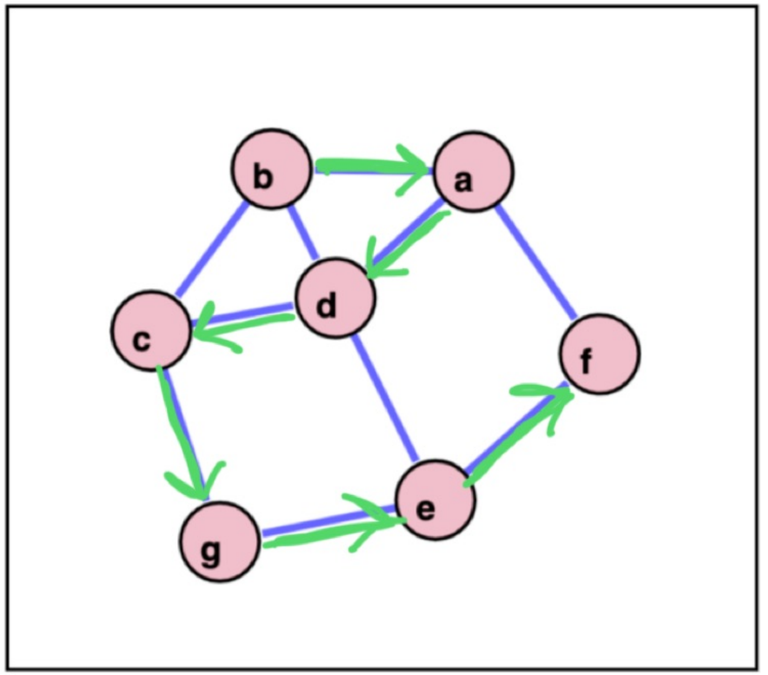


K - L - M - H - C - B - G
- F - A - D - E
- N - I - J - O - T
- S - R - Q - P - U - V
- W - Y - X

DFS uses a stack

Exercise: visit all nodes using DFS, starting at b. List the order in which nodes are traversed.

Assume we are using `TreeSet<Node>` to store adjacent nodes.

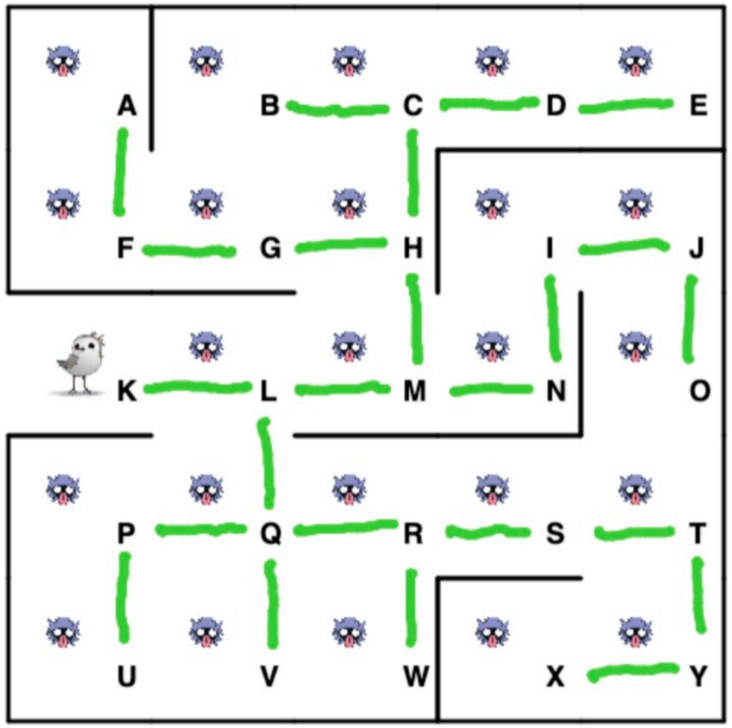


Choose next neighbor in alphabetical order

DFS starting at b: b, a, d, c, g, e, f

Breadth-First Search ("flooding")

- **Main idea:** Visit neighbors one "level" at a time.
- Don't step into nodes we already visited.
- Resulting set of edges forms a **tree**: **connected** and **acyclic**

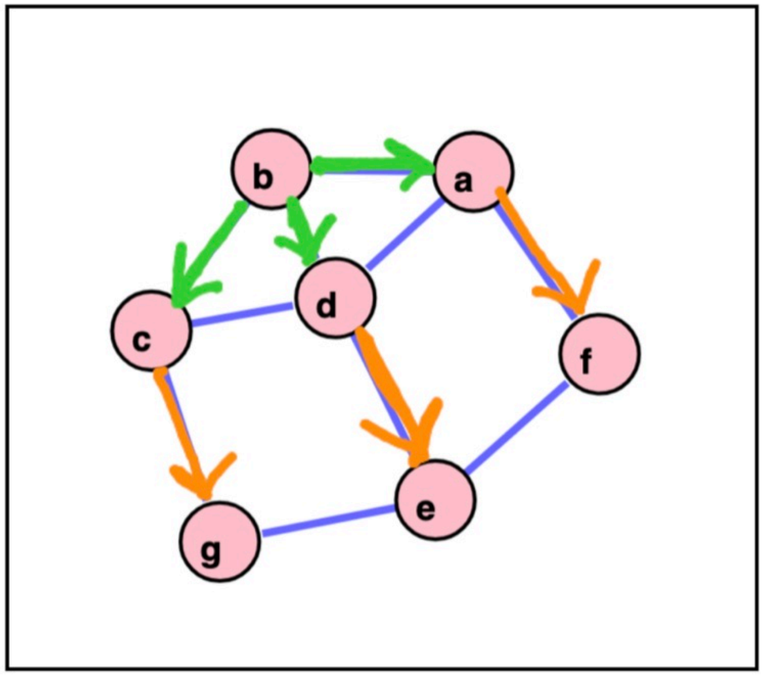


K - L - M - Q - H - N - P - R - V
- C - G - I - U - S - W
- B - D - F - J - T - E
- A - O - Y - X

BFS uses a Queue

Exercise: visit all nodes using BFS, starting at b. List the order in which nodes are traversed.

Assume we are using `TreeSet<Node>` to store adjacent nodes.



b - a - c - d - f - g - e

BFS: b, a, c, d, f, g, e

Implementing DFS and BFS in Java

Notice we are using `TreeSet` for adjacencies: neighboring nodes will be traversed in *order*.

```java
public class Graph<Node> {

  HashMap<Node, TreeSet<Node>> adj;

  public ArrayList<Node> dfs(Node root) {
    ArrayList<Node> order = new ArrayList<>();
    HashSet<Node> visited;
    dfsHelper(root, order, visited);
    return order;
  }

  private void dfsHelper(Node u,
                         ArrayList<Node> order,
                         HashSet<Node> visited) {
    visited.add(u);
    order.add(u);

    // TODO what lines could go here
    // to visit all the adjacent nodes of u?
  }
}
```

```java
public class Graph<Node> {

  HashMap<Node, TreeSet<Node>> adj;

  public ArrayList<Node> bfs(Node root) {
    ArrayDeque<Node> queue = new ArrayDeque<>();
    ArrayList<Node> order = new ArrayList<>();
    HashSet<Node> visited = new HashSet<>();

    queue.add(root);
    visited.add(root);
    while (!queue.isEmpty()) {
      Node u = queue.poll();
      order.add(u);

      // TODO what lines could go here
      // to visit all the adjacent nodes of u?
    }
    return order;
  }
```

```java
// possible implementation:
for (Node v : adj.get(u)) {
  if (!visited.contains(v)) {
    dfsHelper(v, order, visited);
  }
}
```

```java
// possible implementation:
for (Node v : adj.get(u)) {
  if (!visited.contains(v)) {
    visited.add(v);
    queue.add(v);
  }
}
```

Lab today:

- Start Homework 10, due Thursday 5/8
- Submit what you have so far to Gradescope at the end of the lab period