Graphs
Chemical Bonds
PANFLUTE FLOWCHART

do you need one

YES

no you don't

NO

no panflute
A graph is a mathematical structure for representing relationships.
A graph is a mathematical structure for representing relationships.
A **graph** is a mathematical structure for representing relationships.
A graph is a mathematical structure for representing relationships.

A graph consists of a set of nodes connected by edges.
A **graph** is a mathematical structure for representing relationships.

A graph consists of a set of **nodes** connected by **edges**.
A graph is a mathematical structure for representing relationships.

A graph consists of a set of nodes connected by edges.
Some graphs are *directed*. 
Some graphs are *undirected*. 
How can we represent graphs in C++?
We can represent a graph as a map from nodes to the list of nodes each node is connected to.
Representing Graphs

• The approach we just saw is called an **adjacency list** in comes in a number of different forms:

  Map<string, Vector<string>>
  HashMap<string, HashSet<string>>
  Map<string, Set<string>>
  Vector<Vector<int>>

• The core idea is that we have some kind of mapping associating each node with its outgoing edges.
Representing Graphs

The approach we just saw is called an *adjacency list* in comes in a number of different forms:

- `Map<string, Vector<string>>`
- `HashMap<string, HashSet<string>>`
- `Map<string, Set<string>>`
- `Vector<Vector<int>>`

The core idea is that we have some kind of mapping associating each node with its outgoing edges.

Question to ponder: where have you seen this before?
Other Graph Representations

This representation is called an **adjacency matrix**.

For those of you in Math 51: if $A$ is an adjacency matrix for a graph $G$, what is the significance of the matrix $A^2$?
Many problems work on an implicit graph.
You’ll find graphs just about everywhere you look. They’re an extremely versatile and powerful abstraction to be aware of.
Going forward, unless stated otherwise, assume we’re using an adjacency list.
Traversing Graphs
Iterating over a Graph

• In a singly-linked list, there’s pretty much one way to iterate over the list: start at the front and go forward!

• In a binary search tree, there are many traversal strategies:
  • An *inorder traversal* that produces all the elements in sorted order.
  • A *postorder traversal* used to delete all the nodes in the BST.

• There are many ways to iterate over a graph, each of which have different properties.
One Search Strategy
**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.
Core idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
Core idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.
Core idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
Core idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
Core idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.
Implementing this Idea
Visit nodes in ascending order of distance from the start node $E$.  

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.  

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Queue: $D$, $B$

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Queue: $B$

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node \( E \).

Queue: \( \text{B} \)

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. 

Load newly-discovered nodes into a queue.

Queue: $B$, $A$, $G$
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.

Queue: $A$, $G$, $C$
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. 

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node \( E \).

Load newly-discovered nodes into a queue.

Queue: \( A, G, C \)
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.

Queue: $C$, $H$
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node \( E \).

Queue: \( H, F \)

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Queue: $F$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. 

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$. 

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node E.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node \( E \).

Queue: \( I \)

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Visit nodes in ascending order of distance from the start node $E$.

Load newly-discovered nodes into a queue.
Breadth-First Search

- The Queue-based search strategy we just saw is called *breadth-first search* (or just **BFS** for short).
- In pseudocode:

```plaintext
bfs-from(node v) {
    make a queue of nodes, initially seeded with v.
    while the queue isn't empty:
        dequeue a node curr.
        process the node curr.
        for each node adjacent to curr:
            if that node has never been enqueued:
                enqueue that node.
}
```
BFS Efficiency

```plaintext
bfs-from(node v) {
    make a queue of nodes, initially seeded with v.
    while the queue isn't empty:
        dequeue a node curr.
        process the node curr.
        for each node adjacent to curr:
            if that node has never been enqueued:
                enqueue that node.
}
```
BFS Efficiency

bfs-from(node v) {
    make a queue of nodes, initially seeded with v.
    while the queue isn't empty:
        dequeue a node curr.
        process the node curr.
        for each node adjacent to curr:
            if that node has never been enqueued:
                enqueue that node.
}

Whenever we process a node, we do
1. some fixed amount of work to process the node, then
2. some amount of work proportional to the number of edges touching that node.
BFS Efficiency

- Suppose our graph has \( n \) nodes and \( m \) edges.
  - These letters are the standard conventions for talking about graphs.
- Average work done per node: \( O(1) \) baseline work, plus \( O(m/n) \) work processing edges.
- Number of nodes: \( n \).
- Total work done: \( n \cdot (O(m/n) + O(1)) = O(m + n) \).

Whenever we process a node, we do
1. some fixed amount of work to process the node, then
2. some amount of work proportional to the number of edges touching that node.
BFS Efficiency

- The amount of work done to run breadth-first search is $O(m + n)$, assuming the graph is represented as an adjacency list.
  - Great question to ponder: how fast is breadth-first search if we use an adjacency matrix?
- The work done is proportional to the number of objects (nodes and edges) that make up the graph.
- We say that BFS is a **linear-time** graph algorithm.
A Nifty BFS Trick
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: E
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue:
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue:
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: \( D \) \( B \)
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: B
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: B
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: B A G
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: B A G
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: \( A \) \( G \)
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: A G C
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: A G C
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: A \ G \ C
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: G  C
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: C
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: C H
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: \( H \)
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: H
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: \( \text{Queue: } H, F \)
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: $\{H, F\}$
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: F
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: \( F \)
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: $F$ $I$
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: I
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: $I$
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue:  \( I \)
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue:
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.
Run BFS, but have each node store a pointer back to the node that first discovered it.

Start at any node and follow pointers until you reach E. What path are you tracing out?
Shortest Path Routing

- Breadth-first search can be used to find a shortest path from each node back to the start node.
- The tree you get when you do this is called a breadth-first search tree and has lots of fun properties and cool applications.
- Want to learn more? Take CS161!
Next Time

- **Minimum Spanning Trees**
  - Linking Things Cheaply
- **Applications of MSTs**
  - Oh, there are many!