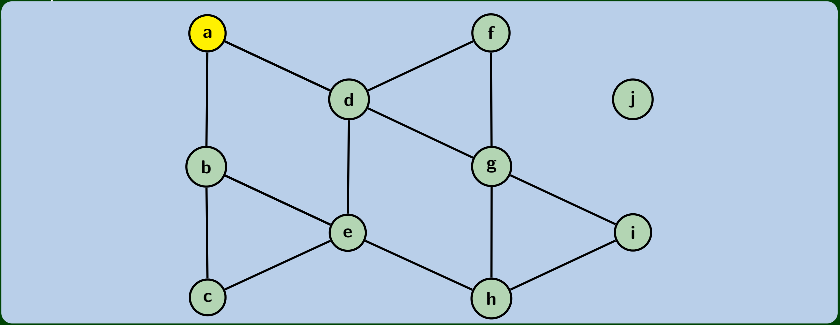
Searching a Graph (with a `SortedList`)

```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

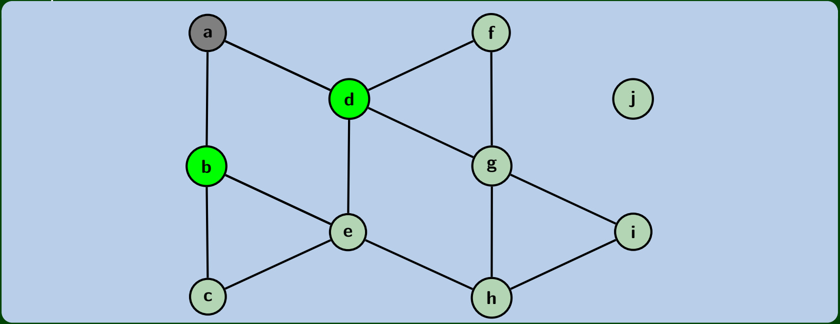


Searching a Graph (with a SortedList)


```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

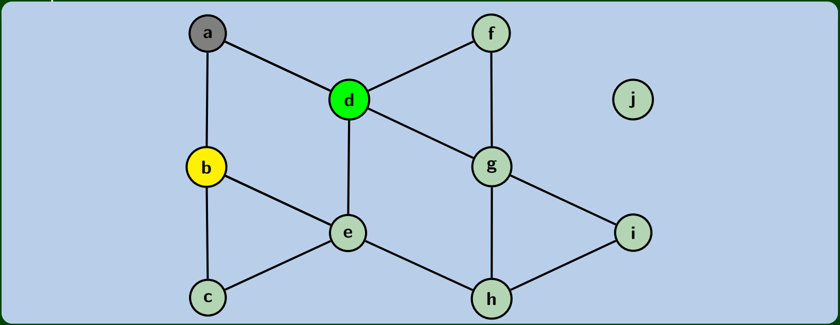
1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist



```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

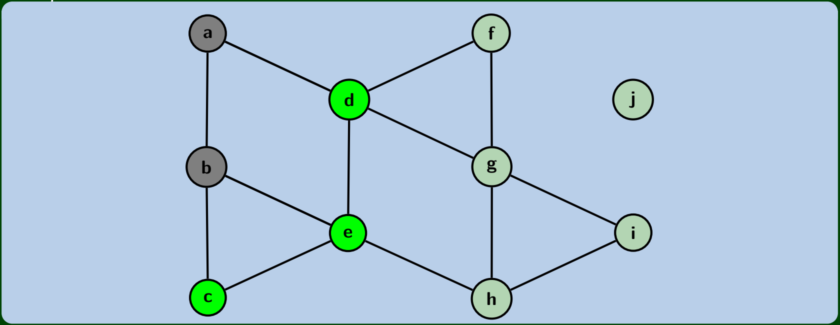


Searching a Graph (with a `SortedList`)


```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

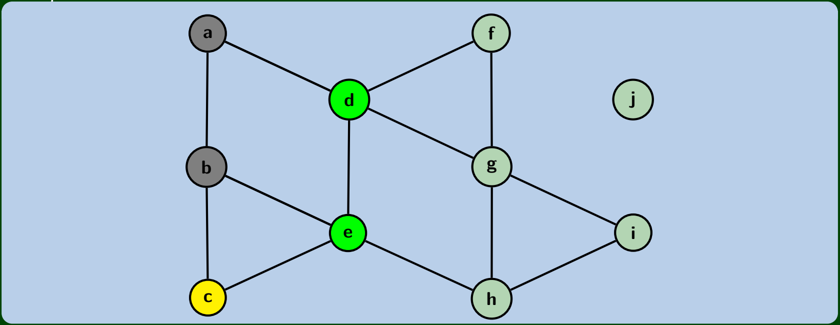


Searching a Graph (with a `SortedList`)

```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

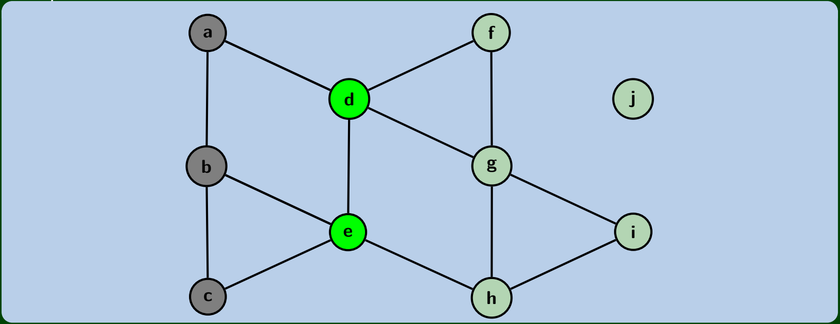
1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist



```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

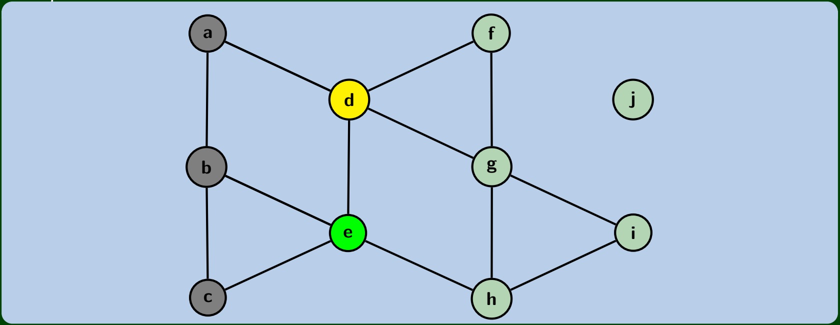
1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist



```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

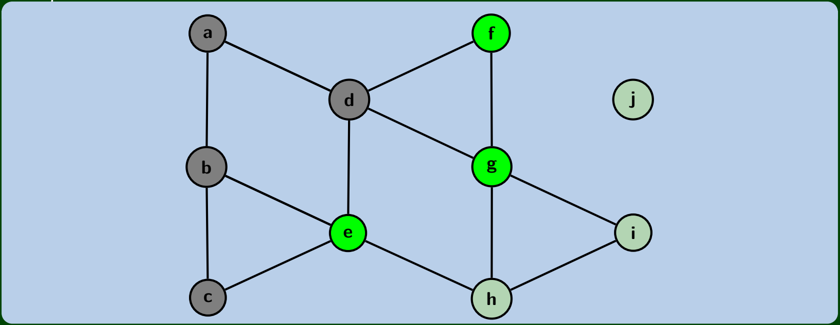


Searching a Graph (with a `SortedList`)

```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

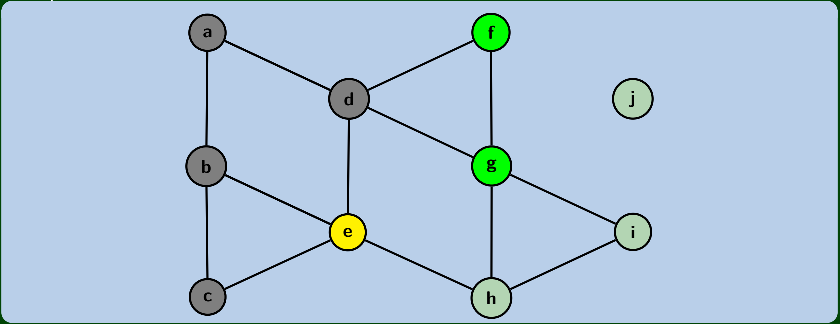
1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist



```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

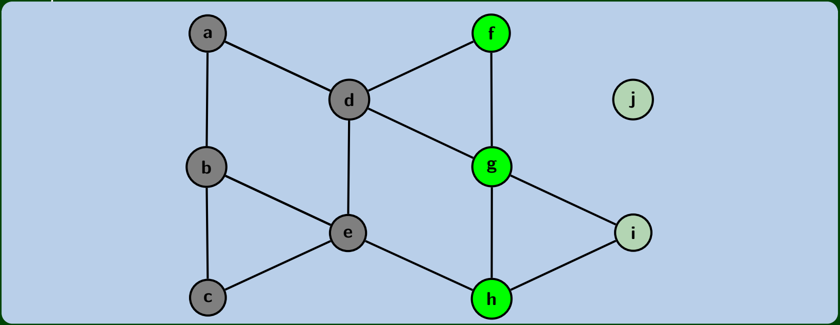


Searching a Graph (with a SortedList)


```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

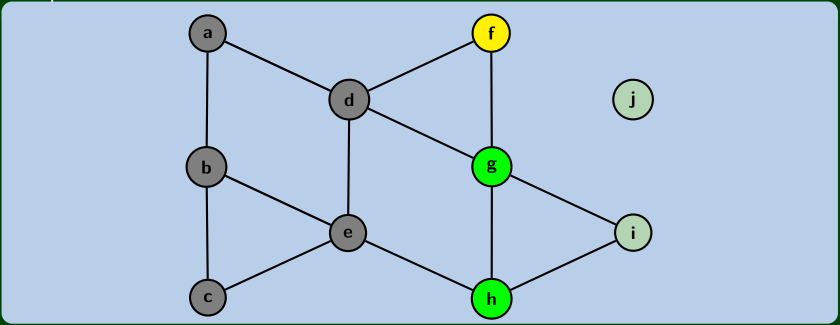
1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist



```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

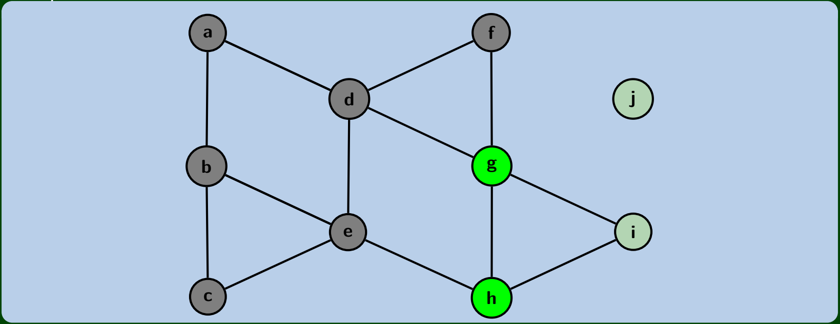


Searching a Graph (with a SortedList)

```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

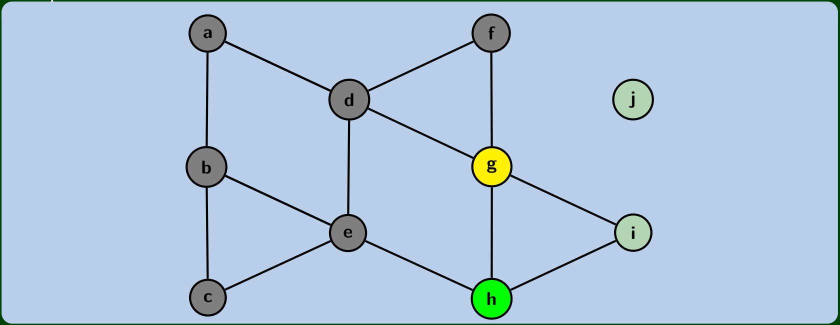


Searching a Graph (with a `SortedList`)


```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

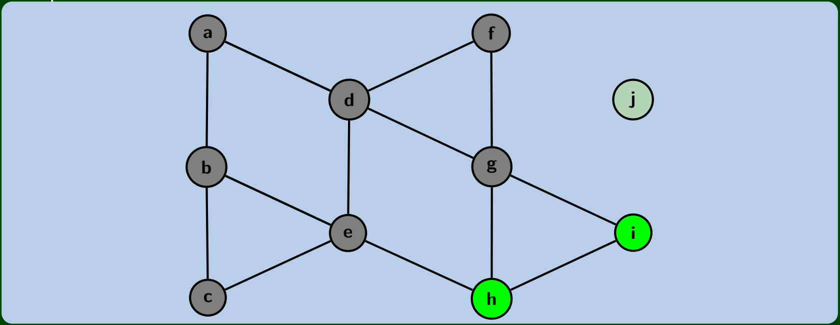


Searching a Graph (with a SortedList)

```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

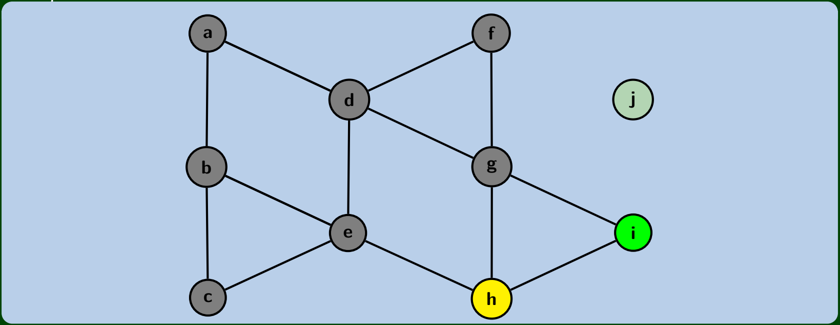
1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist



```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

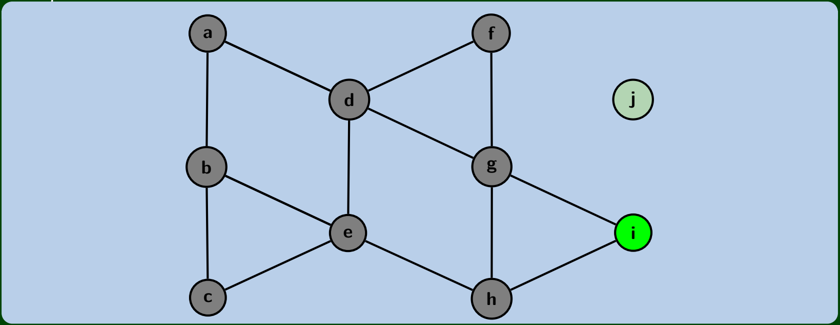


Searching a Graph (with a SortedList)

```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

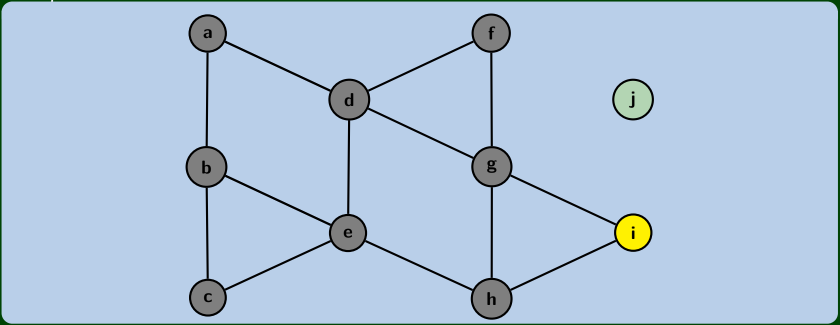
1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist



```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist

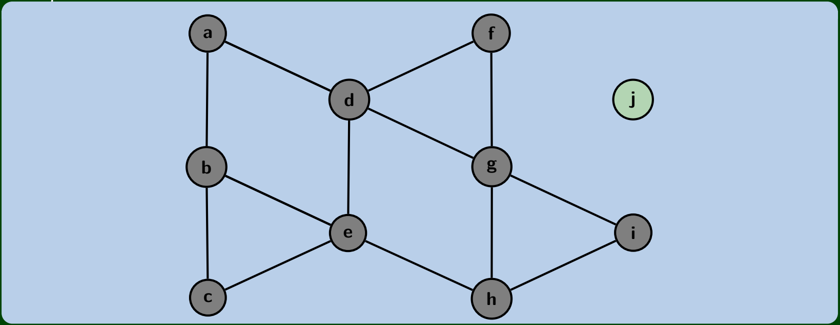


Searching a Graph (with a `SortedList`)


```
search(v) {
   worklist = [v];
   seen = {v};
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not seen) {
            worklist.add(w);
            seen = seen ∪ {w};
         }
      }
   }
```

We have two ways of fixing this:

1. insist that the worklist take care of duplicates, and
2. avoid feeding duplicates to the worklist



Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

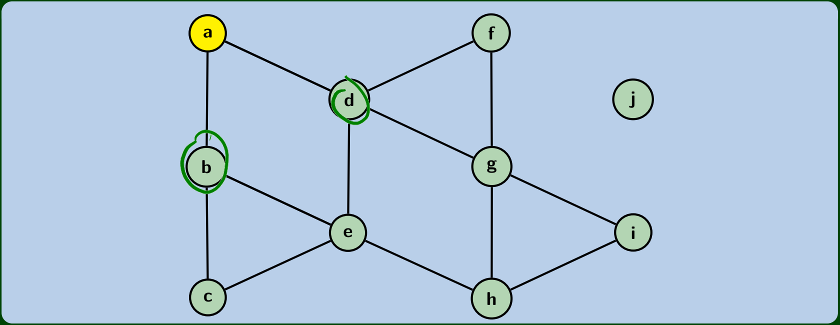
worklist

↓↑

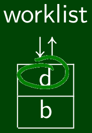
a

When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

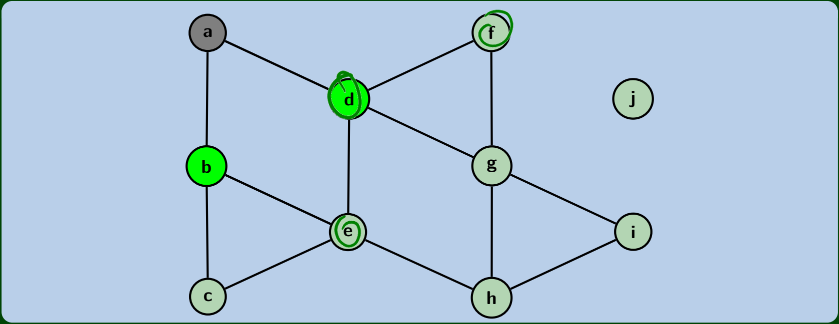


Okay, but we'd never actually use a `SortedList`. How about a `Stack`?



When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively



Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

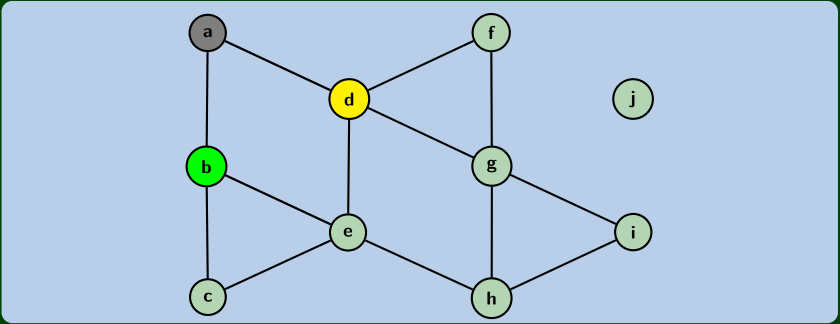
worklist

↓↑

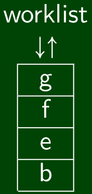
d
b

When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

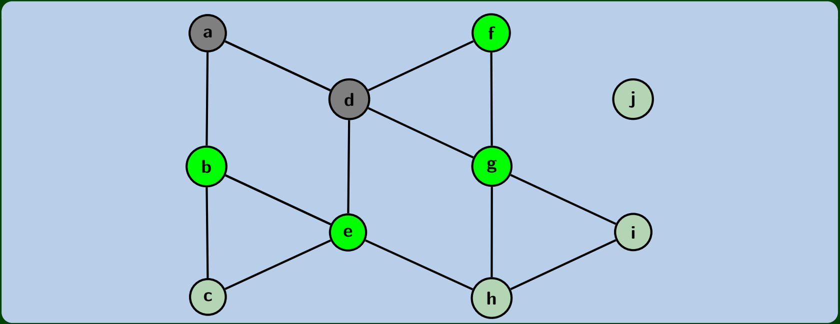


Okay, but we'd never actually use a `SortedList`. How about a `Stack`?



When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively



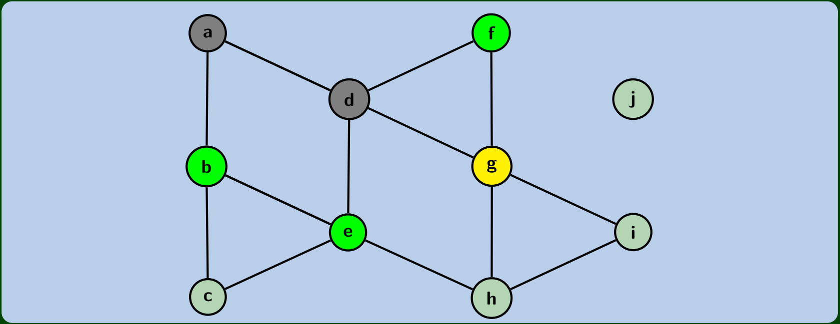
Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

worklist
↓↑

g
f
e
b

When we use a `Stack`:

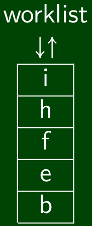
1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

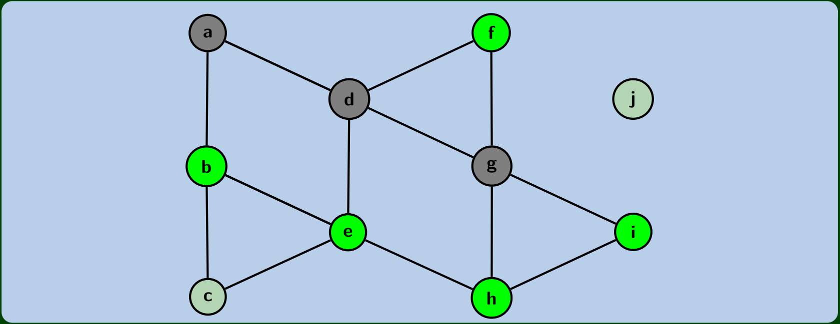


Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively





Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

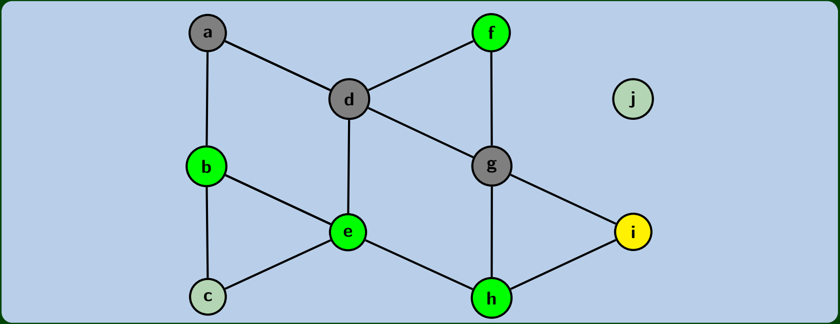
worklist

↓↑

i
h
f
e
b

When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively



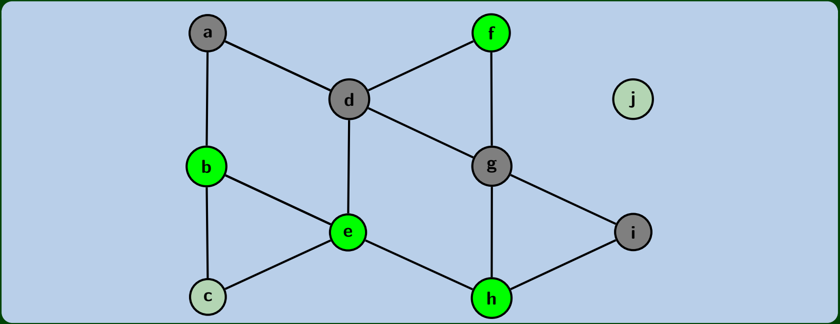
Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

worklist
↓↑

h
f
e
b

When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively



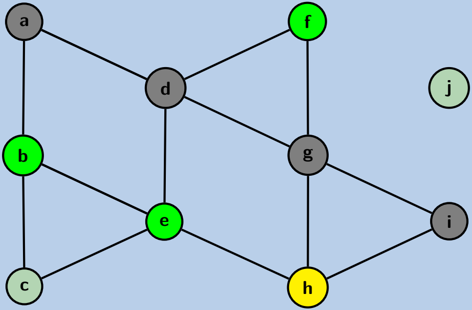
Okay, but we'd never actually use a SortedList. How about a Stack?

worklist
↓↑

h
f
e
b

When we use a Stack:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively



Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

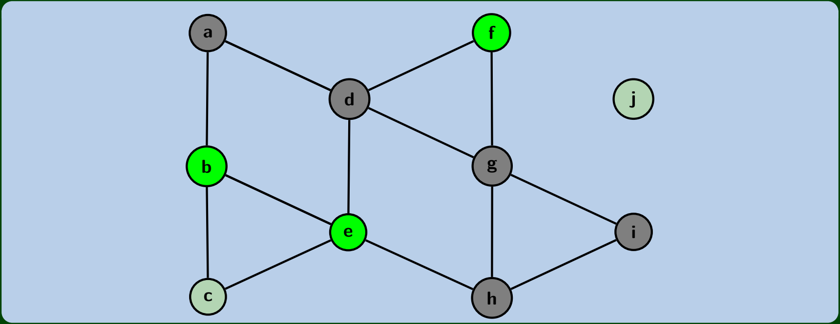
When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

worklist

↓↑

f
e
b



Searching a Graph (with a Stack)

Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

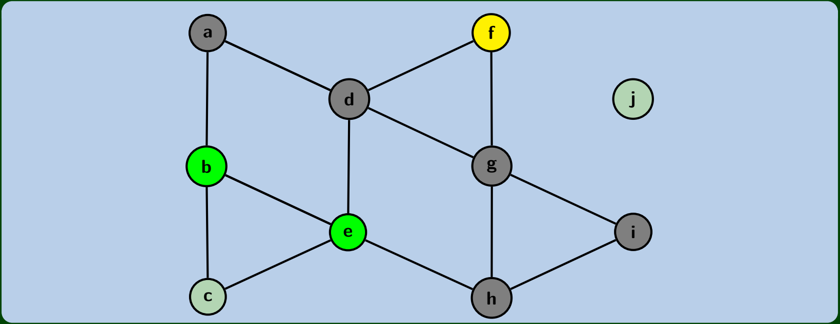
When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

worklist

↓↑

f
e
b



Okay, but we'd never actually use a SortedList. How about a Stack?

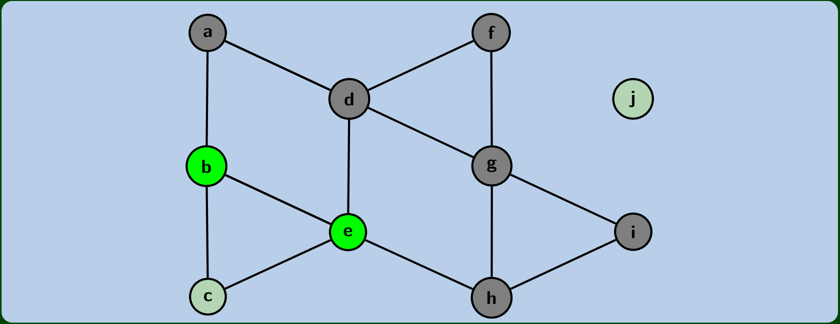
worklist

↓↑

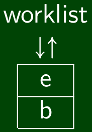
e
b

When we use a Stack:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

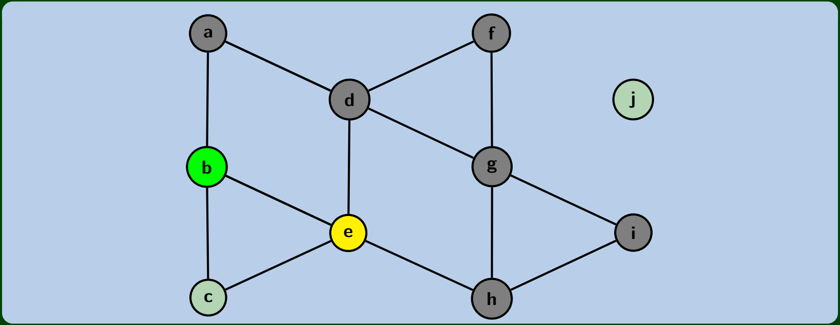


Okay, but we'd never actually use a `SortedList`. How about a `Stack`?



When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

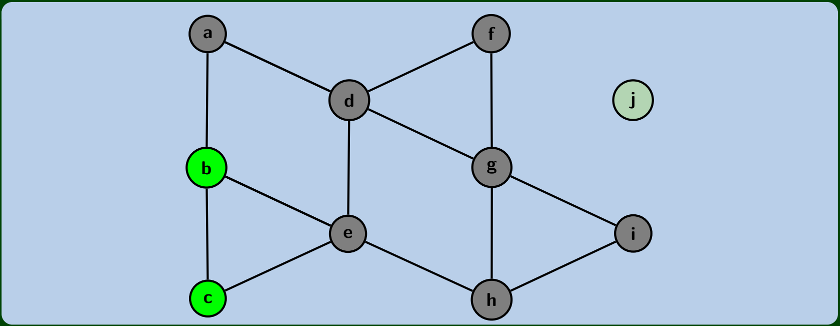


Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

worklist
↓↑
c
b

When we use a `Stack`:

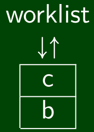
1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

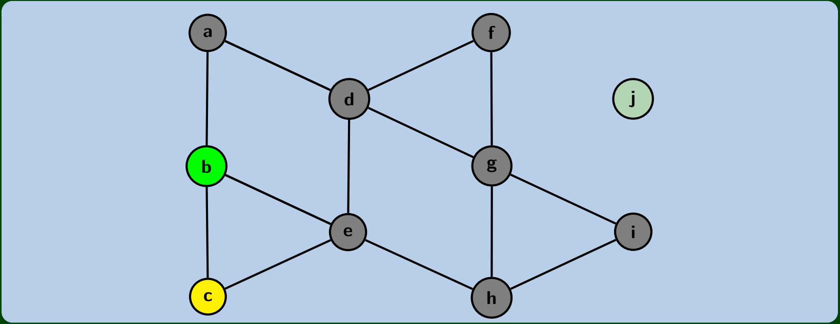


Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively





Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

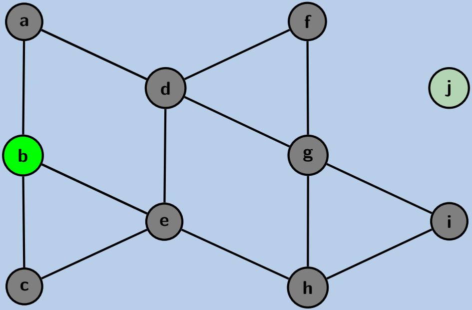
When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

worklist

↓↑

b

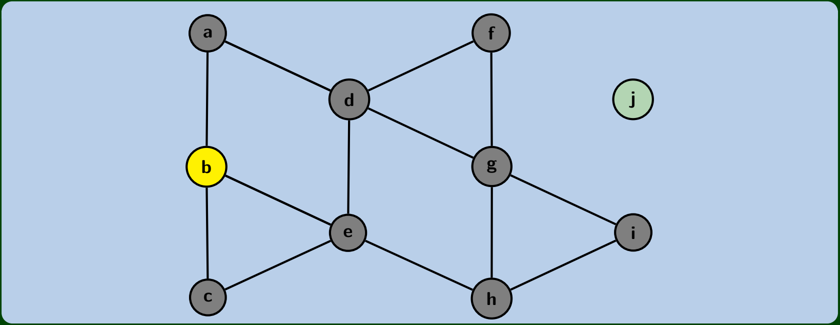


Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

worklist
↓↑
b

When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively



Searching a Graph (with a `Stack`)

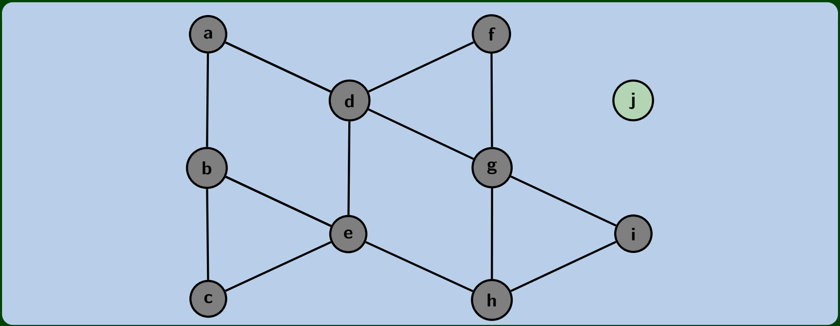
Okay, but we'd never actually use a `SortedList`. How about a `Stack`?

worklist

$\downarrow\uparrow$

When we use a `Stack`:

1. This algorithm is called DFS (depth-first search)
2. We follow a path as far as possible, then back up
3. It can be written recursively

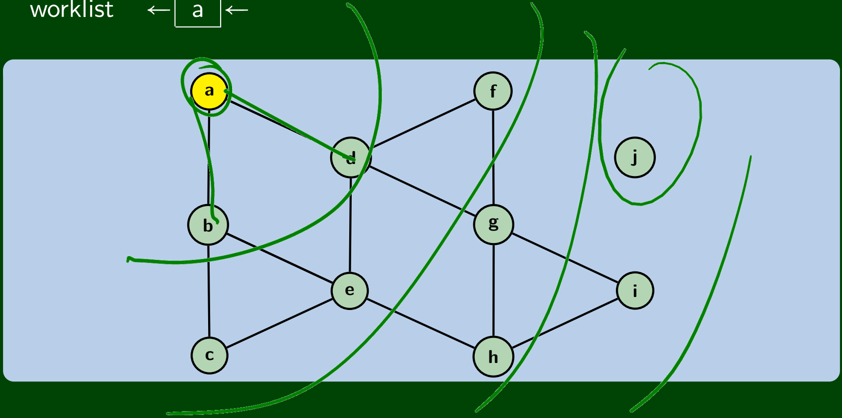


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← a ←

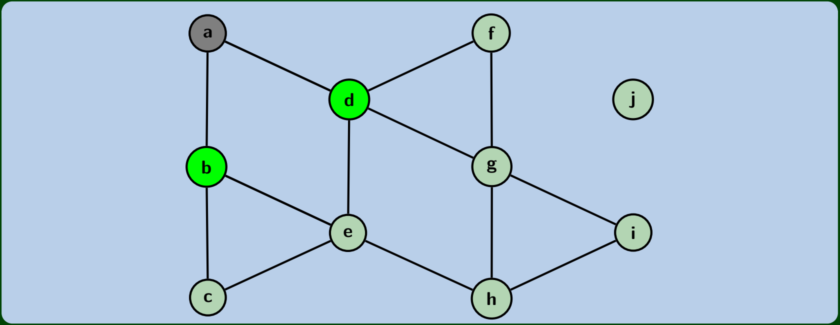


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist	←	b	d	←

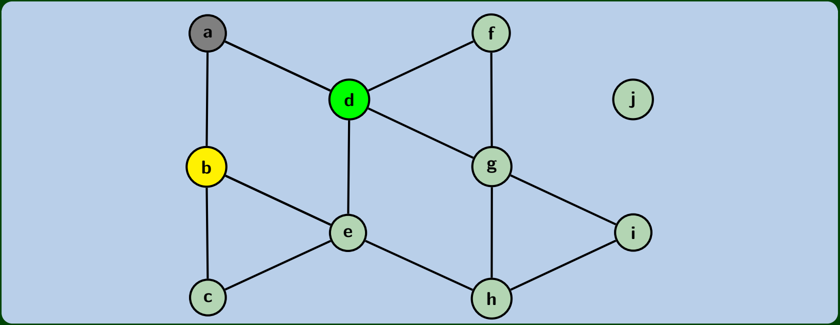


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← | b | d | ←

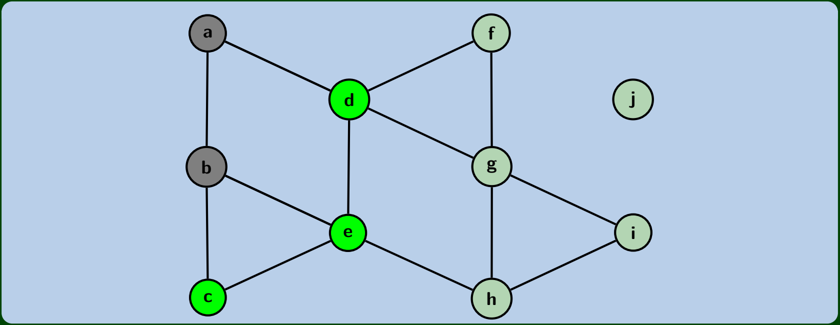


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← | d | c | e | ←

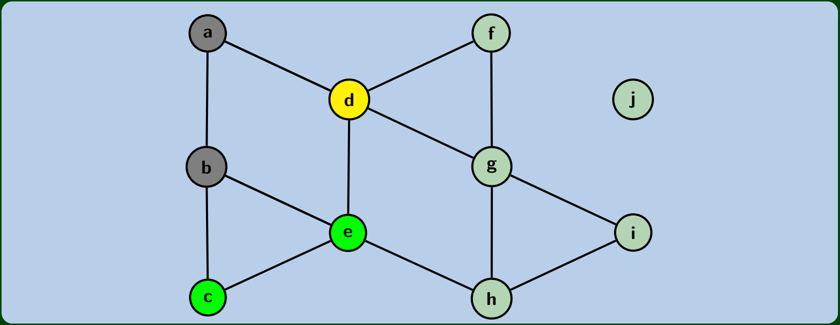


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist	←	d	c	e	←

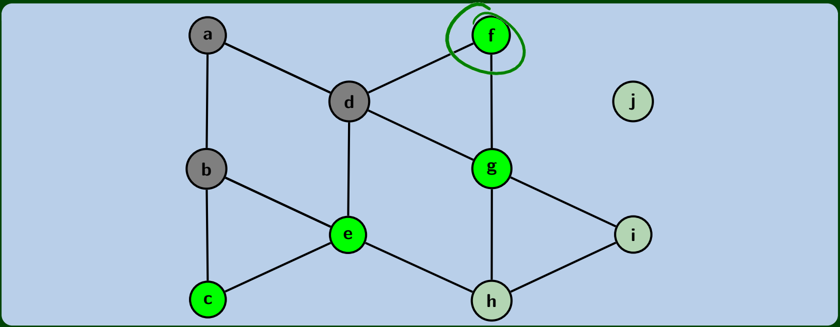


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← | c | e | f | g | ←

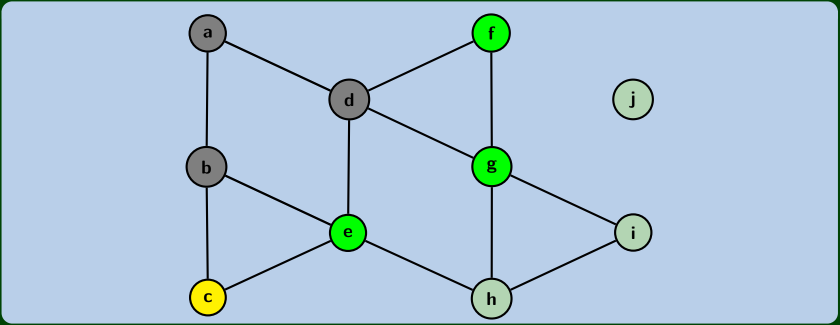


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist	←	c	e	f	g	←



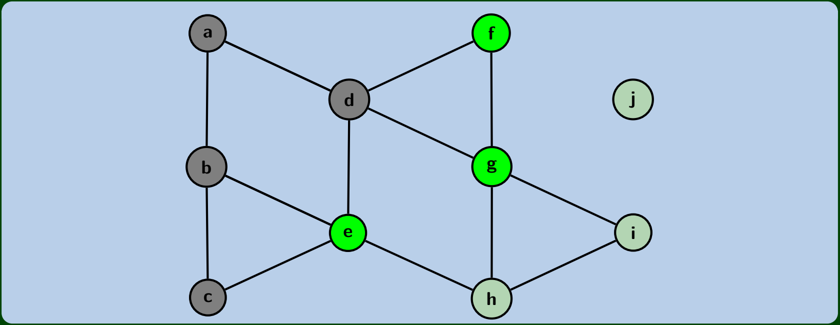
Searching a Graph (with a `Queue`)

Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ←	e	f	g	←



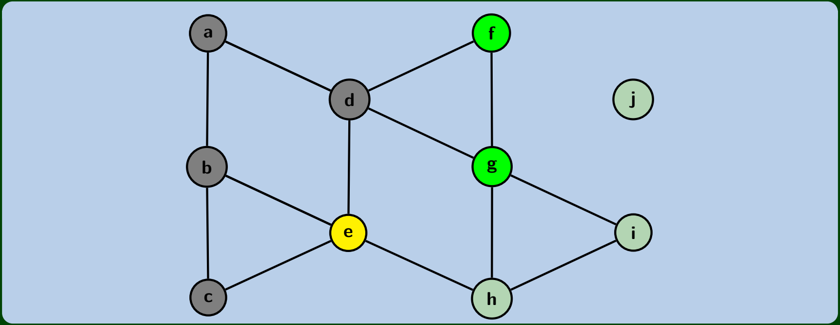
Searching a Graph (with a `Queue`)

Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← | e | f | g | ←

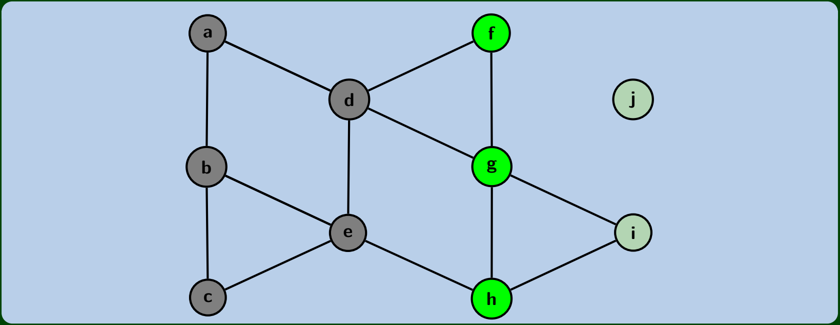


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← | f | g | h | ←

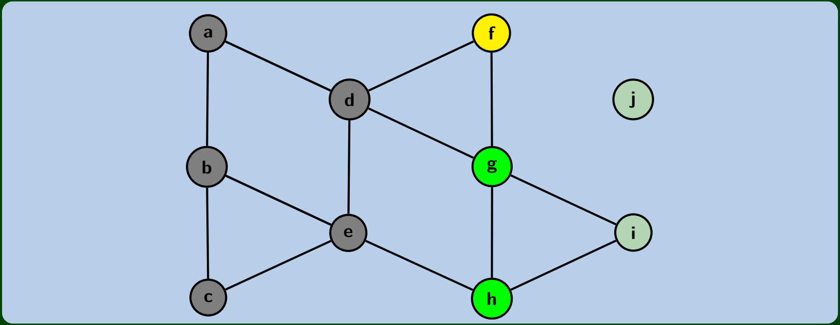


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← | f | g | h | ←

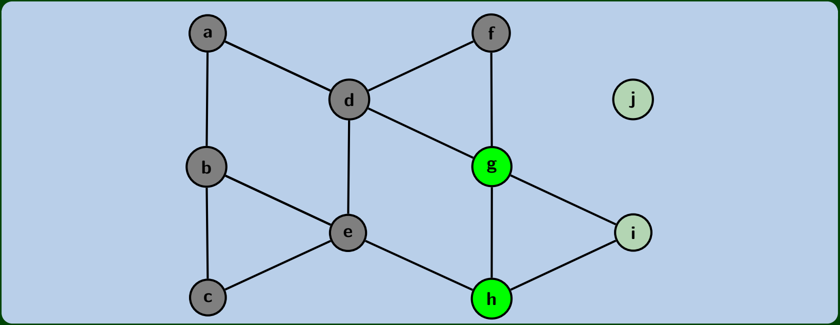


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist	←	g	h	←

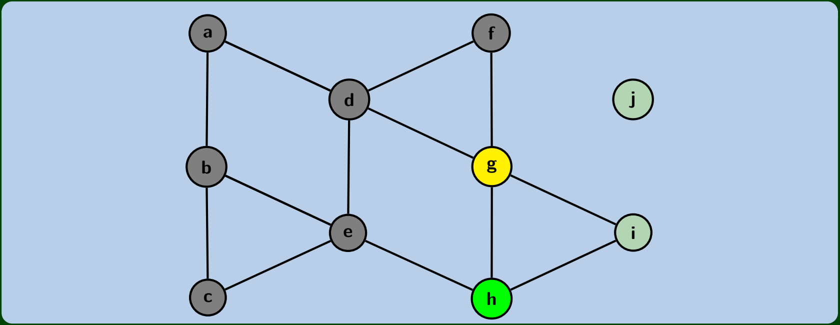


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist	←	g	h	←

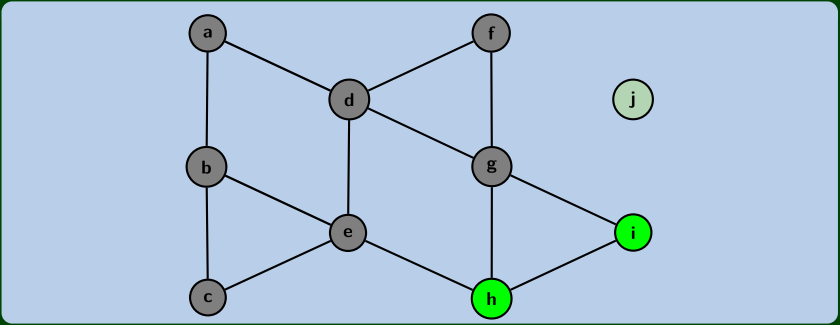


Any reason we shouldn't use a Queue?

When we use a Queue:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← | h | i | ←

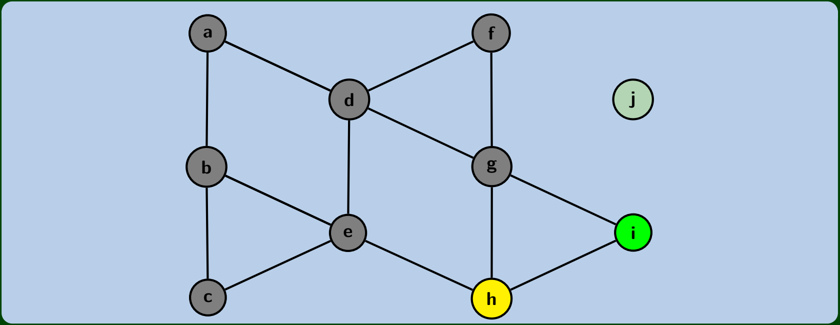


Any reason we shouldn't use a `Queue`?

When we use a `Queue`:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← | h | i | ←

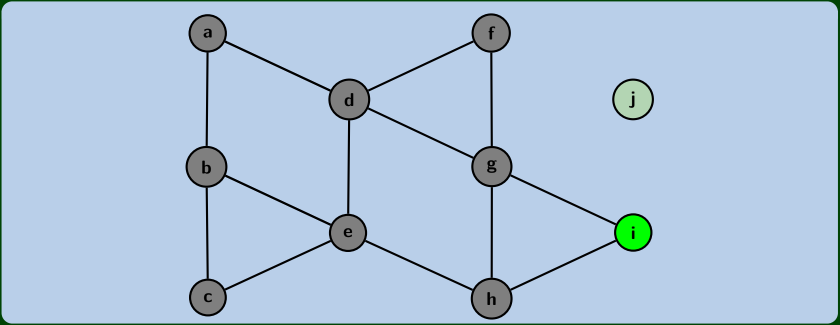


Any reason we shouldn't use a Queue?

When we use a Queue:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← i ←

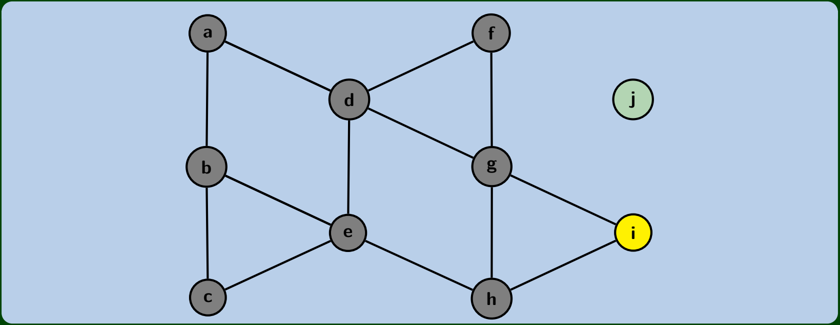


Any reason we shouldn't use a Queue?

When we use a Queue:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ← i ←



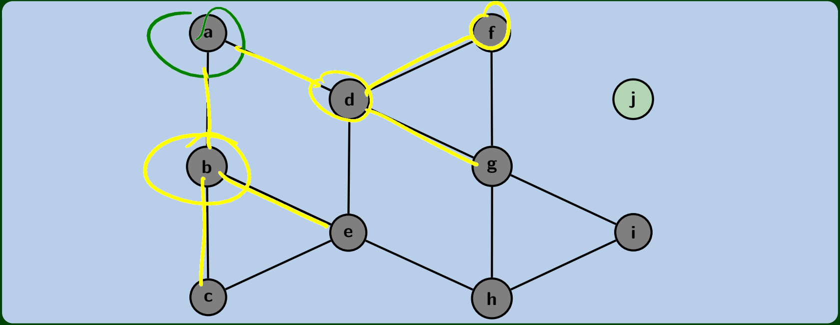
Any reason we shouldn't use a Queue?

When we use a Queue:

1. This algorithm is called BFS (breadth-first search)
2. We increase our horizon away from the starting node

worklist ←

{a:null, d:a, b:a, c:b, e:b,
f:d, g:d}



Use A Dictionary!

```
search(v) {
   worklist = [v];
   from = new Dictionary();
   from.put(v, null);
   while (worklist.hasWork()) {
      v = worklist.next();
      doSomething(v);
      for (w : v.neighbors()) {
         if (w not in from) {
            worklist.add(w);
            from.put(w, v);
         }
      }
   }
   return from;
}
```

```
findPath(v, w) {
   from = search(v);
   path = [];
   curr = w;
   while (curr != null) {
      path.add(0, curr);
      curr = from[curr];
   }
   return path;
}
```

Runtime

- Both algorithms visit all nodes in the connected component: $|V|$
- Both algorithms can visit a node once for each edge in the graph: $|E|$

So, BFS and DFS are $\mathcal{O}(|V|+|E|)$ (this is called "graph linear").

Space

- DFS: If the longest path has length p and the largest number of neighbors is n, then DFS stores at most pn vertices
- BFS: Consider a tree. BFS will hold the entire bottom level which is $\mathcal{O}(|V|)$.

Trade-Offs

- DFS has better space usage, but it might find a circuitous path
- BFS will always find the shortest path to a node, but it will use more memory

Iterative Deepening

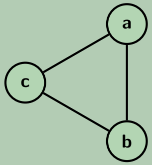
Iterative Deepening is a DFS that **bounds** the depth:

```
int depth = 1;
while (there are nodes to explore) {
   dfs(v, depth);
   depth++;
}
```

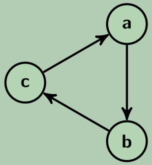
Since **most of the vertices are "leaves"**, this actually doesn't waste much time!

- Undirected vs. Directed (do the edges have arrows?)

Undirected

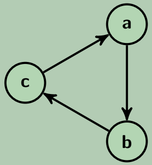


Directed

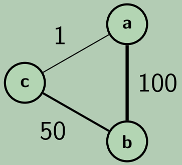


- Weighted vs. Unweighted (do the edges have weights?)

Unweighted & Directed

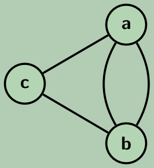


Weighted & Undirected

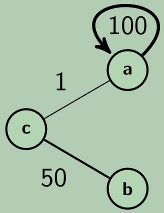


- Simple vs. Multi (loops on vertices? multiple edges?)

Multi-graph



Graph with Loops



These generalizations are all useful in different domains. We're going to talk a lot more about them over the next few lectures.

Next lecture, we'll be working mostly with **directed graphs**.

A Word about Sparsity

Back to counting edges. In a graph without multiple edges, if there are n vertices, there can be anywhere from 0 to n^2 edges.

This is a very wide range. A graph with fewer edges is called **sparse** and one with closer to n^2 is called **dense**.

We already saw that graph traversal was $\mathcal{O}(|E|+|V|)$:

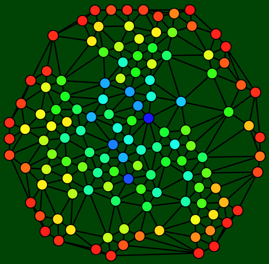
- On a sparse graph, that's $\mathcal{O}(|V|)$
- On a dense graph, that's $\mathcal{O}(|V|^2)$.

Sparsity makes a **huge** difference!

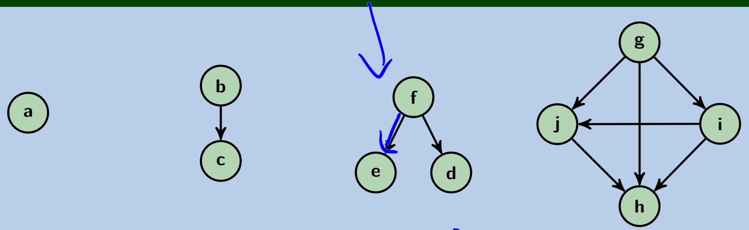
CSE 332

Data Abstractions

Graphs 2: Representing Graphs Topological Sort



A **Directed** Graph is a Thingy. . .



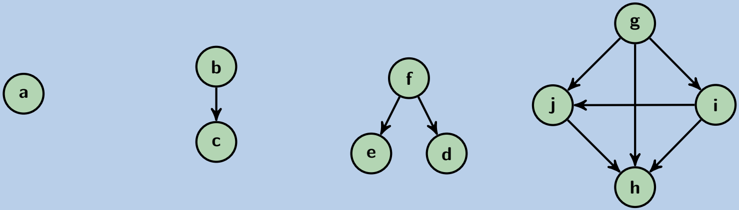
$V = \{a\}$, $E = \varnothing$

$V = \{b, c\}$,
$E = \{(b, c)\}$

$V = \{e, d, f\}$
$E = \{(f, e), (f, d)\}$

Let's extend our terminology for **directed graphs**!

A Directed Graph is a Thingy. . .



$V = \{a\},\ E = \varnothing$

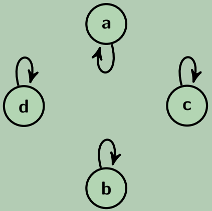
$V = \{b, c\},$
$E = \{(b, c)\}$

$V = \{d, e, f\},$
$E = \{(f, e), (f, d)\}$

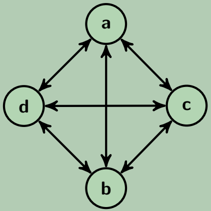
$V = \{g, h, i, j\},$
$E = \{(g, h), (h, i), (g, j),$
$(i, h), (j, h), (i, j)\}$

Let's extend our terminology for **directed graphs**!

A Lonely Graph



Complete Directed Graph

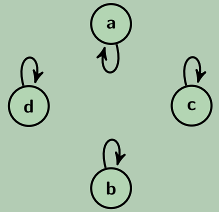


Some Questions

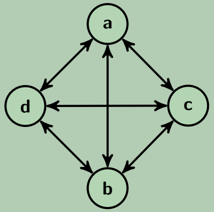
- How many edges can a **directed** graph with $|V| = n$ have?
- How many edges can a **directed** graph with $|V| = n$ and possible loops have?

More Graphs

A Lonely Graph



Complete Directed Graph



Some Questions

- How many edges can a **directed** graph with $|V| = n$ have?

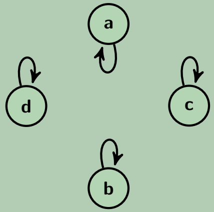
$$|E| = n(n-1)$$

don't allow loops

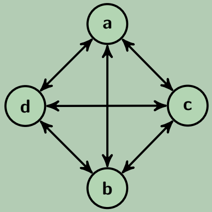
n^2 if allow loops

- How many edges can a **directed** graph with $|V| = n$ and possible loops have?

A Lonely Graph



Complete Directed Graph



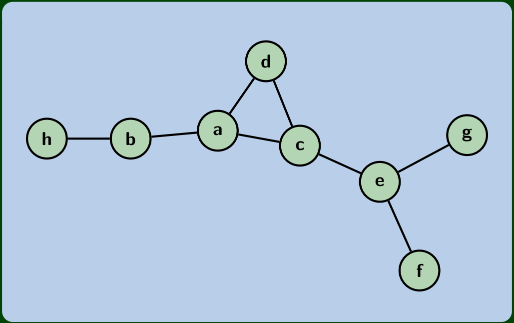
Some Questions

- How many edges can a **directed** graph with $|V| = n$ have?

$$|E| = n(n-1)$$

- How many edges can a **directed** graph with $|V| = n$ and possible loops have?

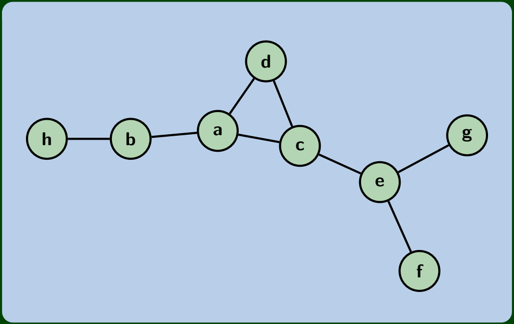
$$|E| = n^2$$



Definition (Degree)

The **degree** of a vertex in a graph is the number of vertices adjacent to it. In the above graph, we have:

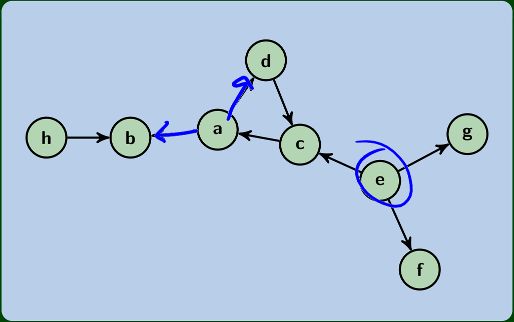
a	b	c	d	e	f	g	h



Definition (Degree)

The **degree** of a vertex in a graph is the number of vertices adjacent to it. In the above graph, we have:

a	b	c	d	e	f	g	h
3	2	3	2	3	1	1	1

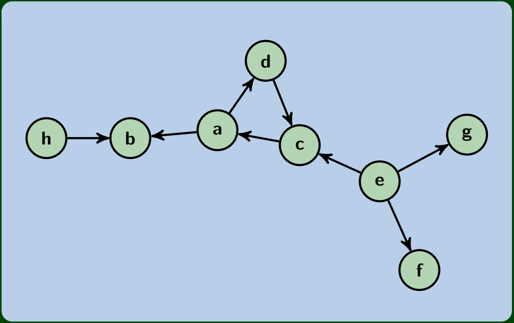


Definition (In & Out Degree)

The **in-degree** of a vertex, v, in a graph is $|\{(x,v) \mid (x,v) \in E, x \in V\}|$.
The **out-degree** of a vertex, v, in a graph is $|\{(v,x) \mid (x,v) \in E, x \in V\}|$.

	a	b	c	d	e	f	g	h
In-Degree	1				0			0
Out-Degree	2				3			1

Burgers? Now?

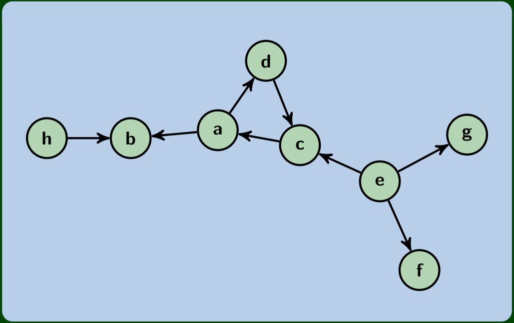


Definition (In & Out Degree)

The **in-degree** of a vertex, v, in a graph is $|\{(x,v) \mid (x,v) \in E, x \in V\}|$.
The **out-degree** of a vertex, v, in a graph is $|\{(v,x) \mid (x,v) \in E, x \in V\}|$.

	a	b	c	d	e	f	g	h
In-Degree	1	2	2	1	0	1	1	0
Out-Degree								

Burgers? Now?

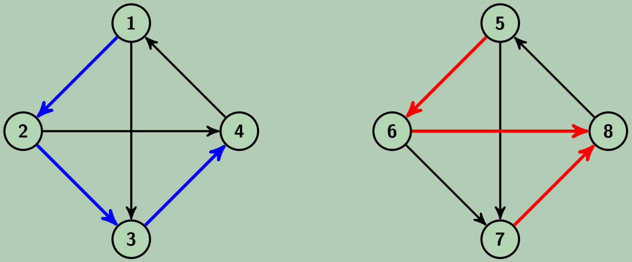


Definition (In & Out Degree)

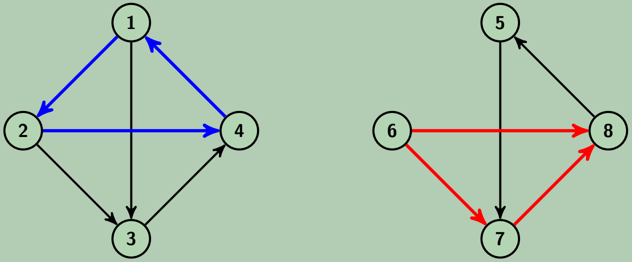
The **in-degree** of a vertex, v, in a graph is $|\{(x,v) \mid (x,v) \in E, x \in V\}|$.
The **out-degree** of a vertex, v, in a graph is $|\{(v,x) \mid (x,v) \in E, x \in V\}|$.

	a	b	c	d	e	f	g	h
In-Degree	1	2	2	1	0	1	1	0
Out-Degree	2	0	1	1	3	0	0	1

Paths?



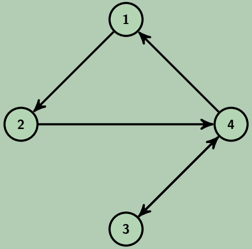
Cycle



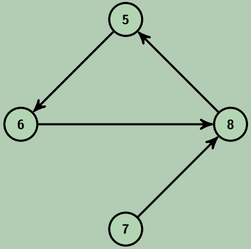
Making A Connection!

Definition (**Strongly** Connected Directed Graph)

We say a directed graph is **strongly connected** iff for every pair of vertices, $u, v \in V$, there is a path from u to v.



Strongly Connected!

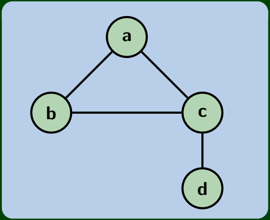


Not Strongly Connected!

Definition (**Weakly** Connected Directed Graph)

We say a directed graph is **weakly connected** iff the underlying undirected graph is connected.

That is, if we "undirected the edges", if the graph is connected, then the digraph is weakly connected.



Adjacency Matrix

	a	b	c	d
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0

Adjacency List

a: b → c →

b: a → c →

c: a → b → d →

d: c →

Adjacency Matrix Analysis

Adjacency Matrix

	a	b	c	d
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0

Adjacency List

a: $\boxed{b} \longrightarrow \boxed{c} \longrightarrow$

b: $\boxed{a} \longrightarrow \boxed{c} \longrightarrow$

c: $\boxed{a} \longrightarrow \boxed{b} \longrightarrow \boxed{d} \longrightarrow$

d: $\boxed{c} \longrightarrow$

Adjacency Matrix Properties

How long to...

- Get a vertex's out-edges? $\mathcal{O}(|V|)$
- Get a vertex's in-edges? $\mathcal{O}(|V|)$
- Check if an edge exists? $\mathcal{O}(1)$
- Insert an edge? $\mathcal{O}(1)$
- Delete an edge? $\mathcal{O}(1)$

Space Requirements: $\mathcal{O}(|V|^2)$

Adjacency Matrices are reasonable for dense graphs, but not otherwise.

Adjacency List Analysis

Adjacency Matrix

	a	b	c	d
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0

Adjacency List

a: $\boxed{b} \longrightarrow \boxed{c} \longrightarrow$

b: $\boxed{a} \longrightarrow \boxed{c} \longrightarrow$

c: $\boxed{a} \longrightarrow \boxed{b} \longrightarrow \boxed{d} \longrightarrow$

d: $\boxed{c} \longrightarrow$

Adjacency List Properties

How long to...

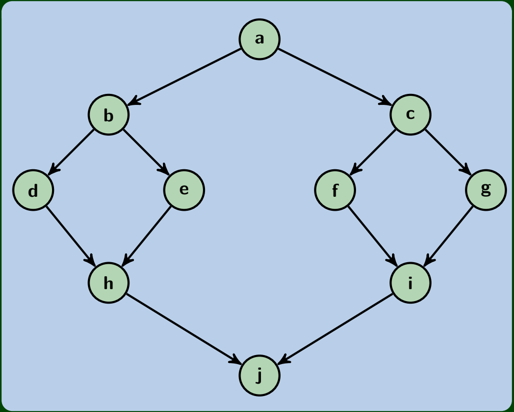
- Get a vertex's out-edges? $\mathcal{O}(d)$
- Get a vertex's in-edges? $\mathcal{O}(|E|)$
  - To fix this, keep a **second** adjacency list going the other way
- Check if an edge exists? $\mathcal{O}(d)$
- Insert an edge? $\mathcal{O}(1)$
- Delete an edge? $\mathcal{O}(d)$

Space Requirements: $\mathcal{O}(|V|+|E|)$

Adjacency Lists should be your goto choice.

Definition (DAG)

A **DAG** is a **directed**, **acyclic** graph.



By "acyclic", we mean in the **directed** sense.

DAGs vs. Trees?

Is there a tree that isn't a DAG?

Is there a DAG that isn't a tree?

DAGs vs. Trees?

All trees are DAGs (remember, trees must be acyclic and connected!).

Not all DAGs are trees. See previous slide. Also, DAGs don't have to be connected!

Why DAGs?

They come up a lot in practice. Cycles can be icky. Examples:

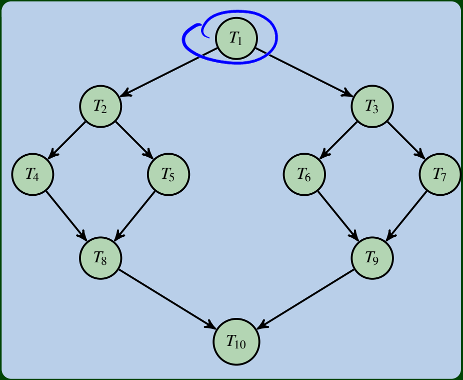
- Any sort of scheduling problem (scheduling your courses, scheduling fork-join threads, . . .)
- Causal Structures (Baysian Networks)
- Genealogy
- . . .

Topological Sort

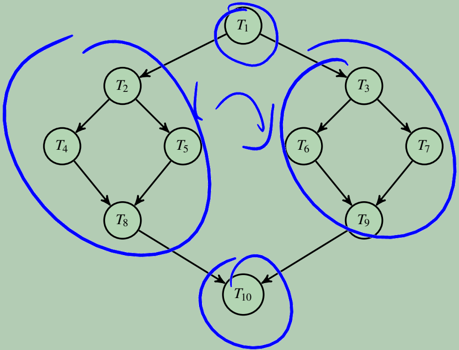
Given a DAG ($G = (V, E)$), output all the vertices in an order such that no vertex appears before any vertex that has an edge to it.

"Output an order to process the graph that meets all dependencies"

This is how we can allocate work in the `ForkJoin` model!



How Many Valid Topological Sorts?



- $T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}$
- $T_1, T_2, T_4, T_3, T_5, T_6, T_7, T_8, T_9, T_{10}$
- $T_1, T_2, T_5, T_4, T_3, T_6, T_7, T_8, T_9, T_{10}$
- $T_1, T_3, T_6, T_7, T_9, T_2, T_5, T_4, T_8, T_{10}$
- ...

Implementing Topological Sort

Throw all the **in-degrees** in a priority queue. `removeMin()` repeatedly.

- This works, but it's **too slow**.
- Insight: `PriorityQueues` must deal with negative numbers; indegree will never be negative!
- Instead: Split ready vs. not ready (0 vs. non-zero) sets
- The "ready set" is a worklist!

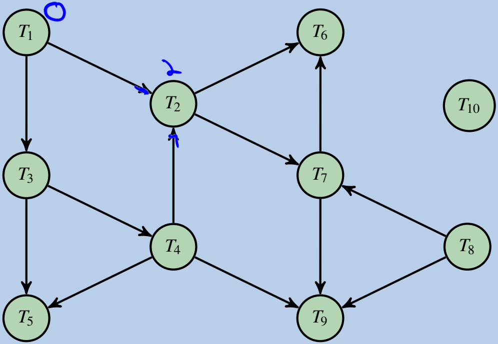
Setup

```
output = []
deps = {}
worklist = []
for (v : vertices) {
   deps[v] = in-degree(v);
   if (deps[v] == 0) {
      worklist.add(v);
   }
}
```

Do Work

```
while (worklist.hasWork()) {
   v = worklist.next();
   output.add(v);
   for (w : neighbors(v)) {
      deps[w] -= 1
      if (deps[w] == 0) {
         worklist.add(w);
      }
   }
}
```

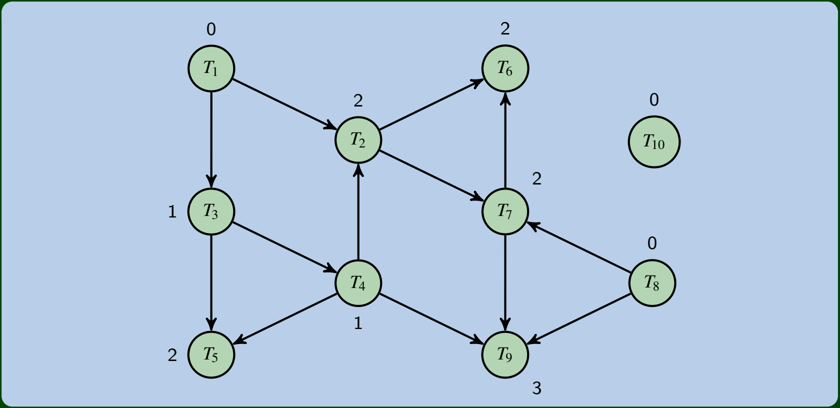
worklist ←



`output`

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]

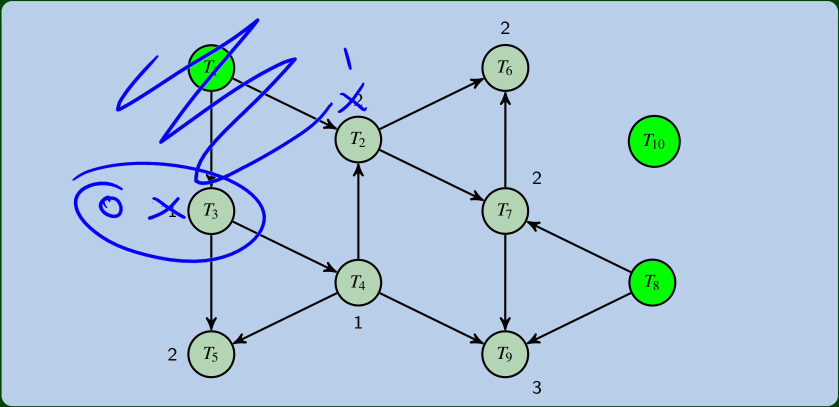
worklist ←



`output`

`o[0]`	`o[1]`	`o[2]`	`o[3]`	`o[4]`	`o[5]`	`o[6]`	`o[7]`	`o[8]`	`o[9]`

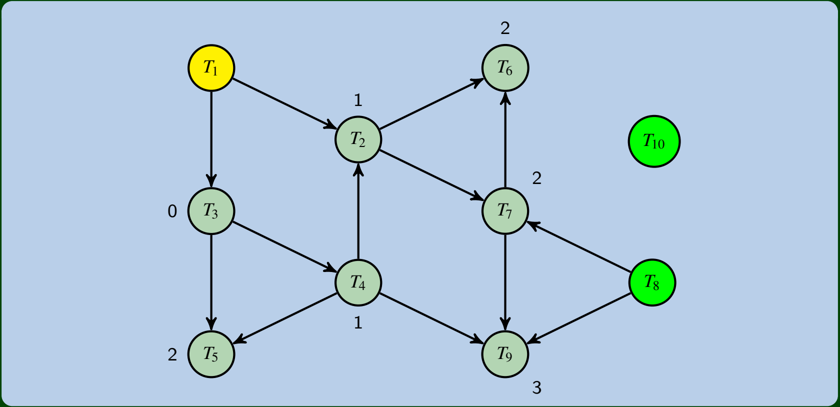
worklist ← | T_1 | T_8 | T_{10} | ←



output

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]

worklist ← | T_8 | T_{10} | ←



output

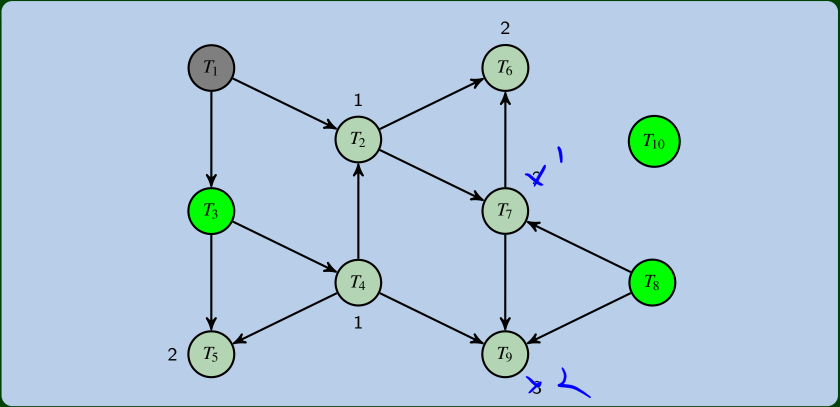
o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]
T_1									

Topologically Sorting A DAG (with a Queue)

worklist $\leftarrow$

$\cancel{T_8}$	T_{10}	T_3

$\leftarrow$

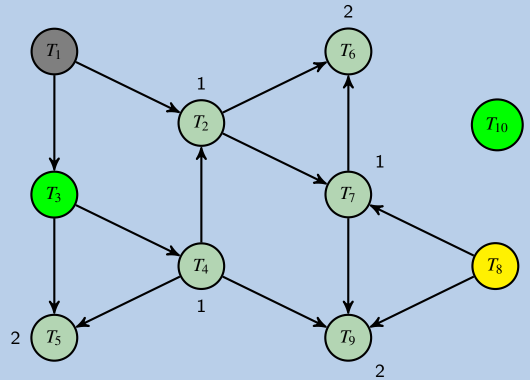


output

T_1									
o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]

Topologically Sorting A DAG (with a `Queue`)

worklist ← T_{10} | T_3 ←

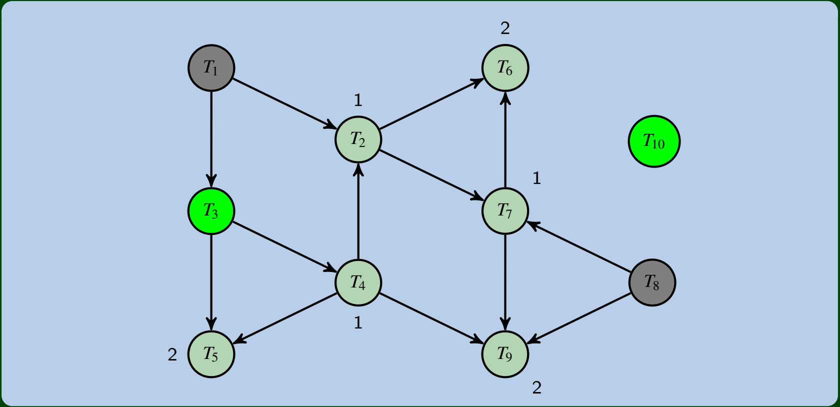


`output`

T_1	T_8								
o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]

Topologically Sorting A DAG (with a Queue)

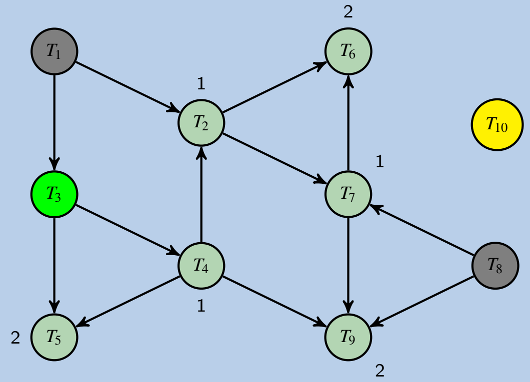
worklist $\leftarrow$ | T_{10} | T_3 | $\leftarrow$



output

T_1	T_8								
o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]

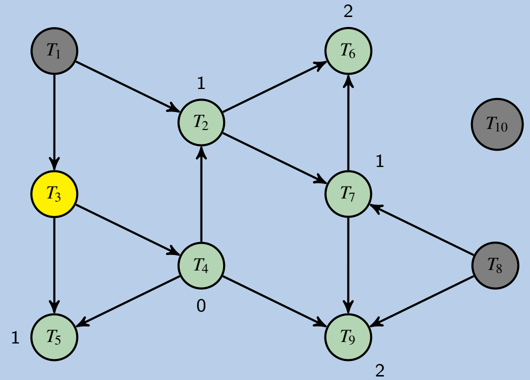
worklist ← T_3 ←



output

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]
T_1	T_8	T_{10}							

worklist ←

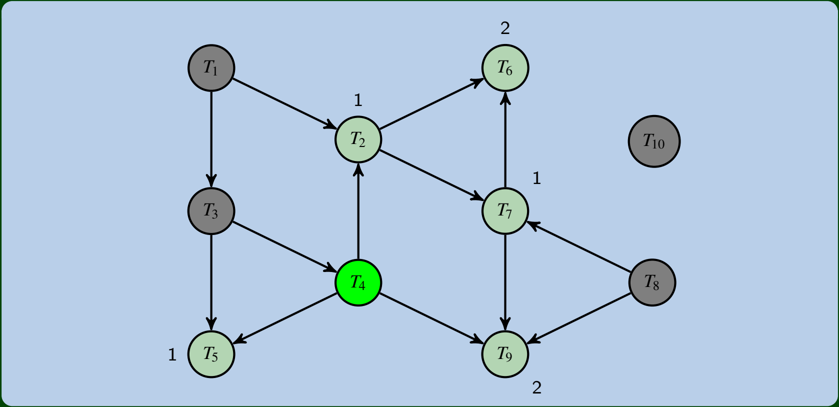


output

T_1	T_8	T_{10}	T_3						
o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]

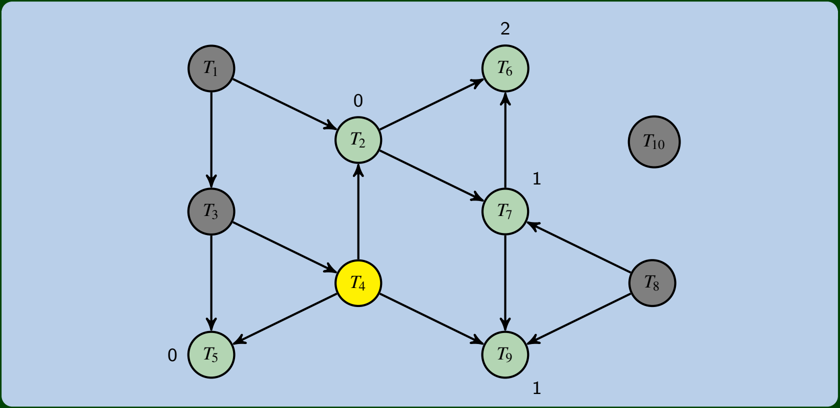
Topologically Sorting A DAG (with a `Queue`)

worklist ← T_4 ←



output	T_1	T_8	T_{10}	T_3						
	o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]

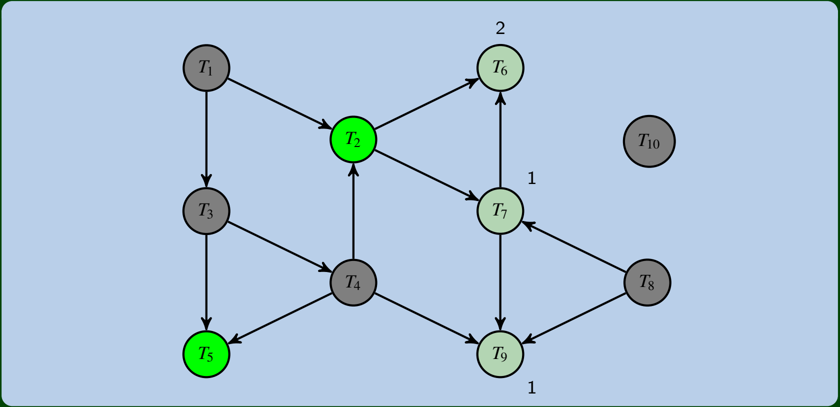
worklist ←



`output`	T_1	T_8	T_{10}	T_3	T_4					
	`o[0]`	`o[1]`	`o[2]`	`o[3]`	`o[4]`	`o[5]`	`o[6]`	`o[7]`	`o[8]`	`o[9]`

Topologically Sorting A DAG (with a Queue)

worklist ← | T_2 | T_5 | ←

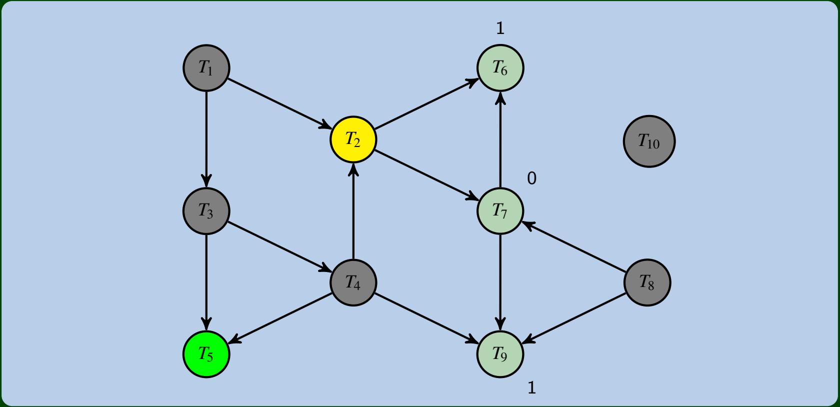


output

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]
T_1	T_8	T_{10}	T_3	T_4					

Topologically Sorting A DAG (with a `Queue`)

worklist ← T_5 ←

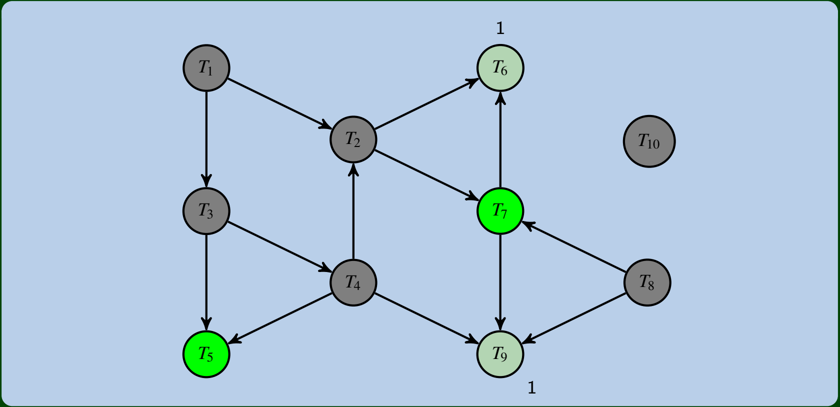


output

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]
T_1	T_8	T_{10}	T_3	T_4	T_2				

Topologically Sorting A DAG (with a `Queue`)

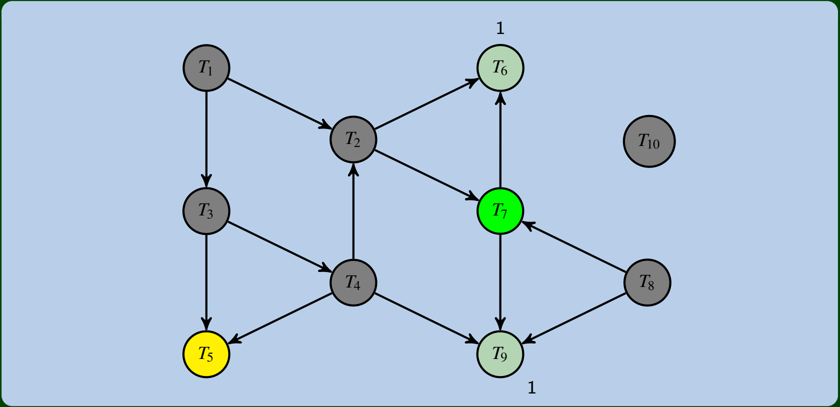
worklist ← | T_5 | T_7 | ←



`output`

T_1	T_8	T_{10}	T_3	T_4	T_2				
o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]

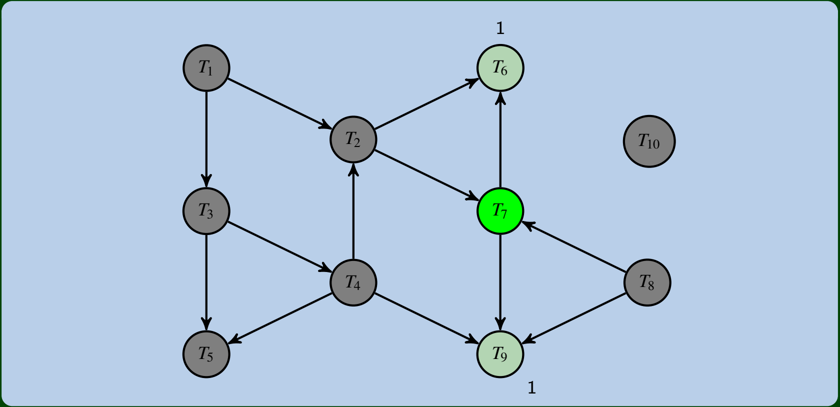
worklist ← T_7 ←



output

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]
T_1	T_8	T_{10}	T_3	T_4	T_2	T_5			

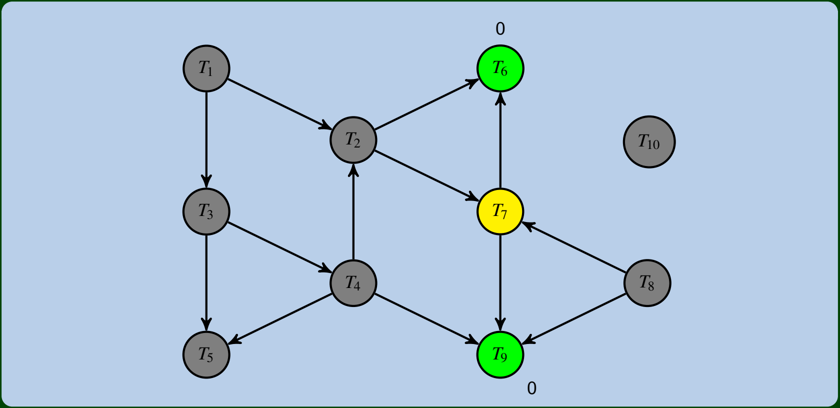
worklist ← T_7 ←



output

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]
T_1	T_8	T_{10}	T_3	T_4	T_2	T_5			

worklist ←

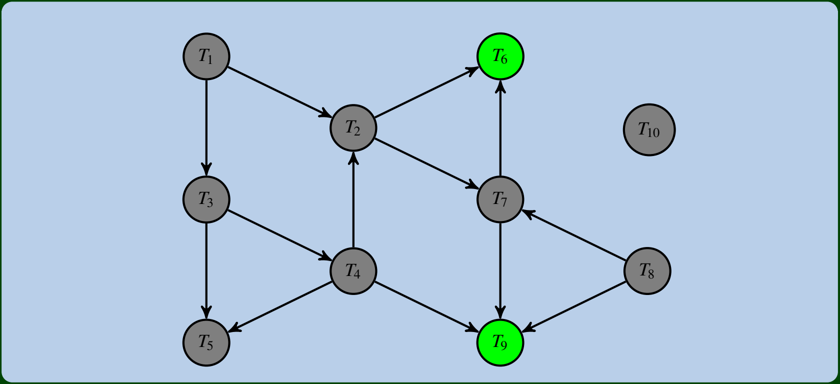


output

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]
T_1	T_8	T_{10}	T_3	T_4	T_2	T_5	T_7		

Topologically Sorting A DAG (with a `Queue`)

worklist ← | T_6 | T_9 | ←

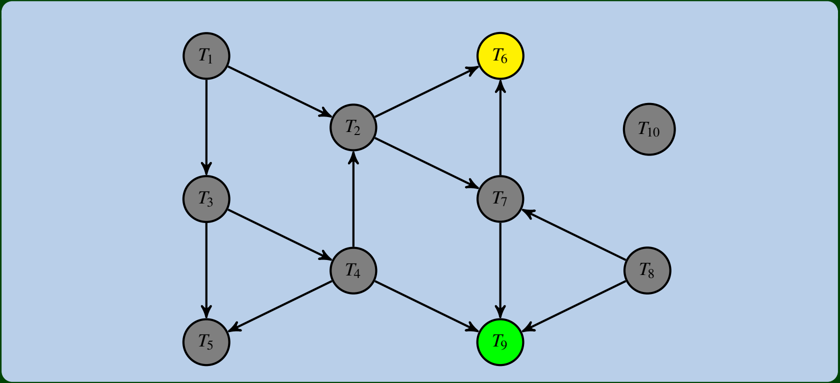


`output`

T_1	T_8	T_{10}	T_3	T_4	T_2	T_5	T_7		
`o[0]`	`o[1]`	`o[2]`	`o[3]`	`o[4]`	`o[5]`	`o[6]`	`o[7]`	`o[8]`	`o[9]`

Topologically Sorting A DAG (**with a `Queue`**)

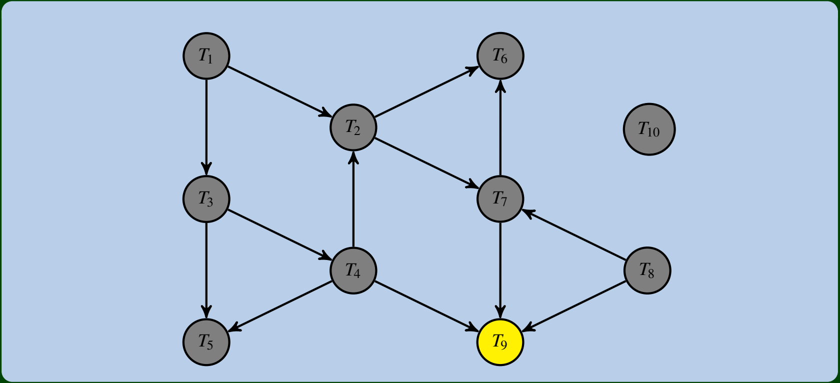
worklist $\leftarrow \boxed{T_9} \leftarrow$



`output`

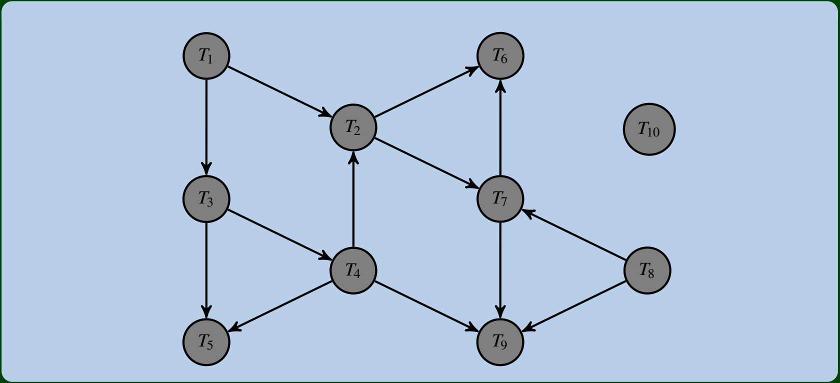
T_1	T_8	T_{10}	T_3	T_4	T_2	T_5	T_7	T_6	
`o[0]`	`o[1]`	`o[2]`	`o[3]`	`o[4]`	`o[5]`	`o[6]`	`o[7]`	`o[8]`	`o[9]`

worklist ←



`output`	T_1	T_8	T_{10}	T_3	T_4	T_2	T_5	T_7	T_6	T_9
	`o[0]`	`o[1]`	`o[2]`	`o[3]`	`o[4]`	`o[5]`	`o[6]`	`o[7]`	`o[8]`	`o[9]`

Topologically Sorting A DAG (with a `Queue`)



What happens if there is a cycle?

Our worklist will be empty before we've processed all of the vertices. (e.g., "there are no nodes ready to print next, but we haven't gone through all of them)
In this case: our algorithm should throw a "not a DAG exception".

Runtime?

- Setup: We follow every edge for every vertex: $\mathcal{O}(|V|+|E|)$
- We add/remove each vertex from the work list once: $\mathcal{O}(|V|)$
- We decrement each indegree until zero (once for each edge): $\mathcal{O}(|E|)$
- So, overall, it's graph linear!