# Parallel And Sequential Data Structures and Algorithms

## Depth-First Search (Continued) and Breadth-First Search

# Learning Objectives

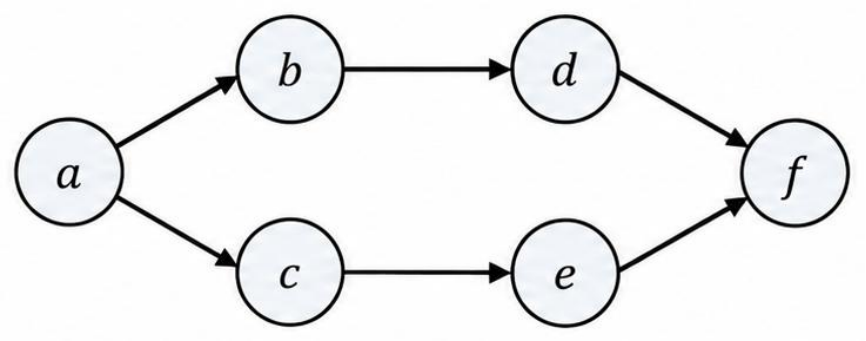
- Continue **DFS** from last lecture
  - Topological sorting
  - Edge classification
- Explore **Breadth-First Search (BFS)**
  - Explore BFS with **more breadth** than in 15-122
  - Develop a **parallel** version of breadth-first search and analyze its cost

# DFS Continued

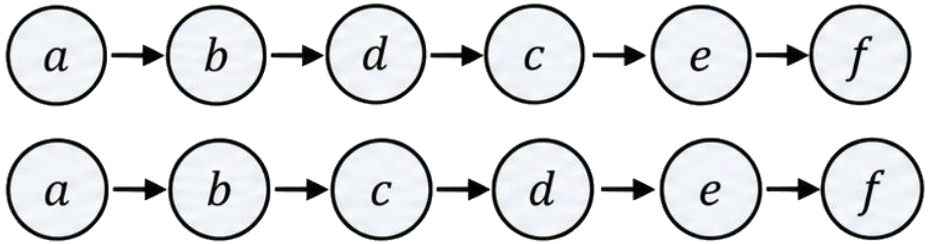
# Topological Sorting

**Problem:** Given a directed acyclic graph (DAG), find an **ordering of its vertices** such that for every edge $(u, v)$, $u$ appears before $v$

### Example: Course prerequisites




### Two example topological orders



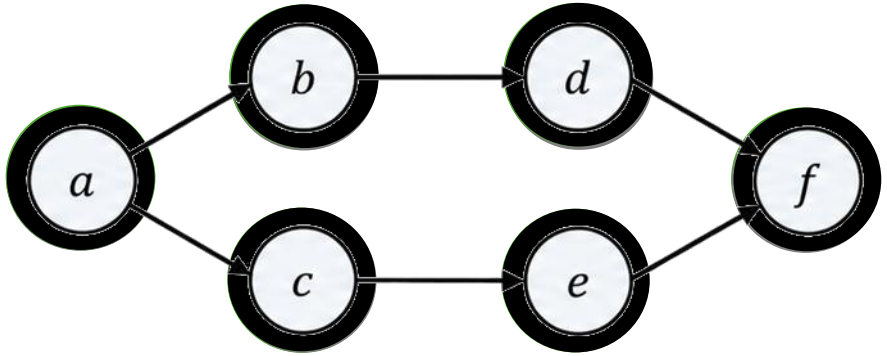

for every edge $(u, v)$, $u$ appears before $v$

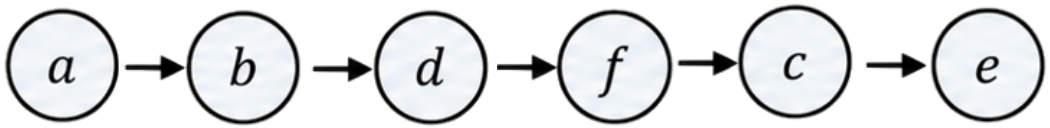
? What can a DFS tell us about the topological order?

# Topological Sorting

**Problem:** Given a directed acyclic graph (DAG), find an **ordering of its vertices** such that for every edge $(u, v)$, $u$ appears before $v$

**Example: Course prerequisites**



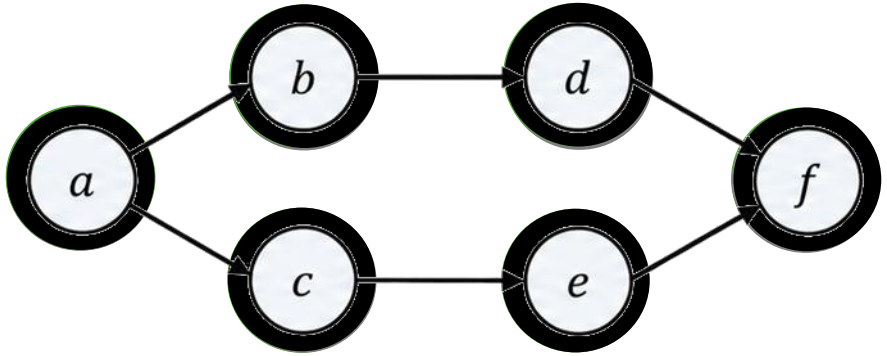



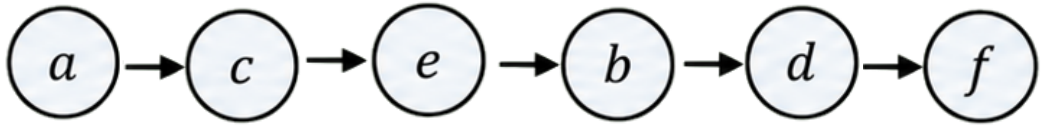

? Does the **visit order** of a DFS give a topological order? **NO**

# Topological Sorting

**Problem:** Given a directed acyclic graph (DAG), find an **ordering of its vertices** such that for every edge $(u, v)$, $u$ appears before $v$

**Example: Course prerequisites**




**Key Idea**: The visit order doesn't help. Look at the **finish order**.

When a vertex finishes, **all vertices reachable from it** have finished

# Application 4: Topological Sorting

**Problem:** Given a DAG, return a topological ordering of its vertices

```
fun topological_sort(adj : UnweightedAdjList):
  n = |adj|
  color = [WHITE for _ in 0...n-1]
  order = []

  fun dfs(u : int):
    color[u] ← GRAY
    for v in adj[u]:
      if color[v] == WHITE:
        dfs(v)

    order ← [u] + order  // FINISH(U)
    color[u] ← BLACK

  for u in 0...n-1:
    if color[u] == WHITE:
      dfs(u)

  return order
```

**Cost:** $O(n^2 + m)$ since `[u] + order` will copy the whole array

# Application 4: Topological Sorting

**Problem:** Given a DAG, return a topological ordering of its vertices

```
fun topological_sort(adj : UnweightedAdjList):
  n = |adj|
  color = [WHITE for _ in 0...n-1]
  order = []

  fun dfs(u : int):
    color[u] ← GRAY
    for v in adj[u]:
      if color[v] == WHITE:
        dfs(v)

    order.append(u)  // FINISH(U)
    color[u] ← BLACK

  for u in 0...n-1:
    if color[u] == WHITE:
      dfs(u)

  return reverse(order)
```

**Cost:** $O(n+m)$ using a dynamic array with $O(1)$ amortized append

# Even More General DFS

# DFS Start and Finish Times

**Motivation**: Three-colour DFS tells us which vertices are currently being explored, but after DFS finishes, every vertex is BLACK. What if we want to **preserve information** about the structure and nesting of recursive explorations?

- We can generalize the colours by recording two **timestamps**:
  - `start[u]`: When DFS first discovered vertex $u$
  - `finish[u]`: When DFS finished exploring $u$
- This generalizes the three-colour scheme:
  - **WHITE**: `start[u] == NONE`
  - **GRAY**: `start[u] != NONE` and `finish[u] == NONE`
  - **BLACK**: `finish[u] != NONE`

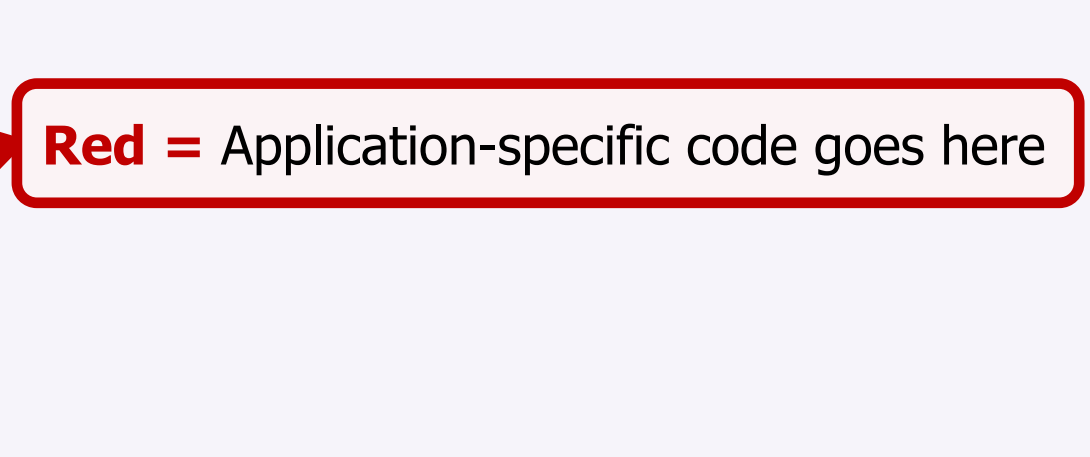
# DFS Start and Finish Times

```
fun dfs_application(adj : UnweightedAdjList, ...):
  n = |adj|
  start = [NONE for _ in 0...n-1]
  finish = [NONE for _ in 0...n-1]
  time = 0

  fun dfs(u : int):
    start[u] ← time++
    // DISCOVER(u)
    for v in adj[u]:
      // EXAMINE(u,v)
      if start[v] == NONE:
        dfs(v)
        // RETURN(u,v)

    // FINISH(u)
    finish[u] ← time++

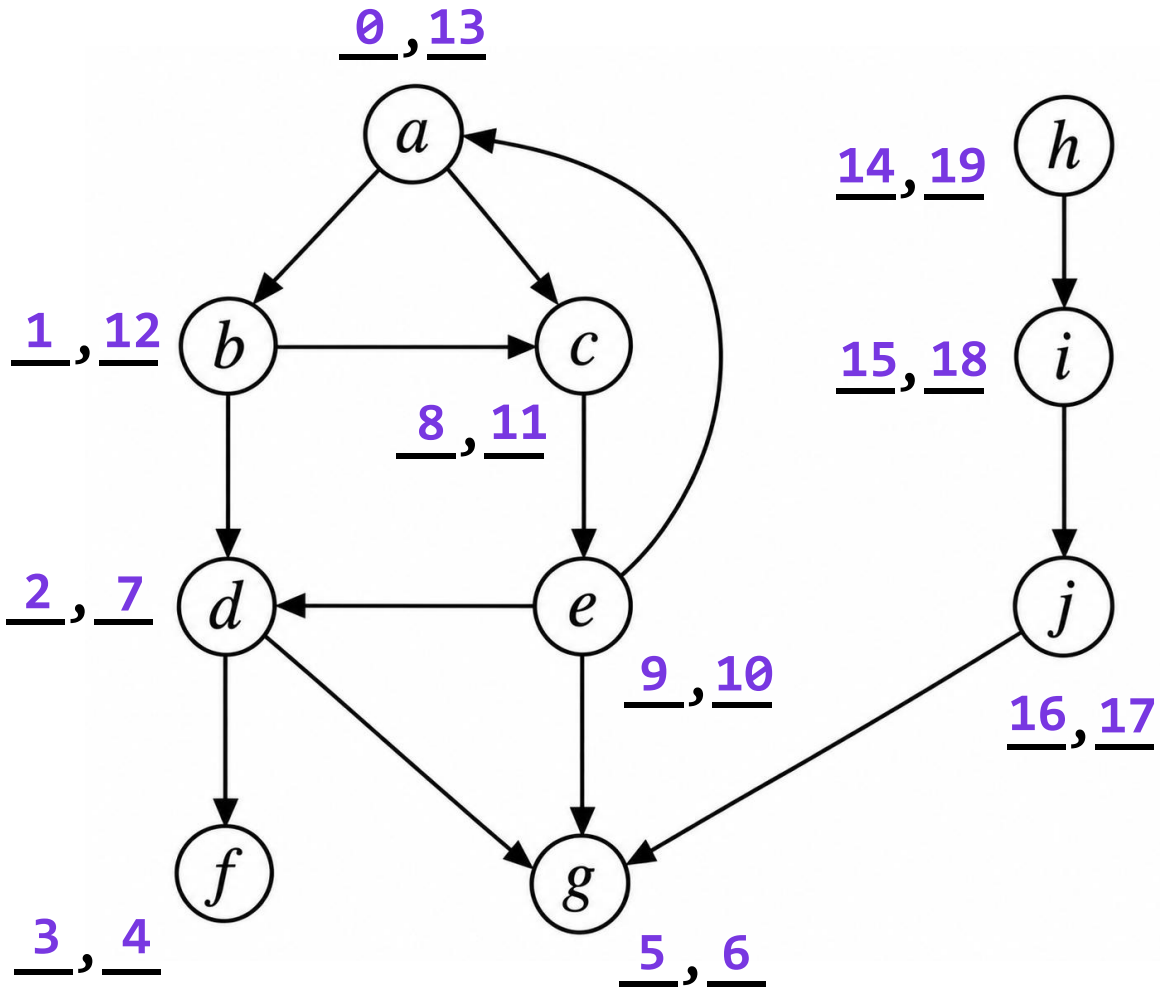
  // Initiate DFS from appropriate vertices
```



# Vertex Numbering

Record two **timestamps**:

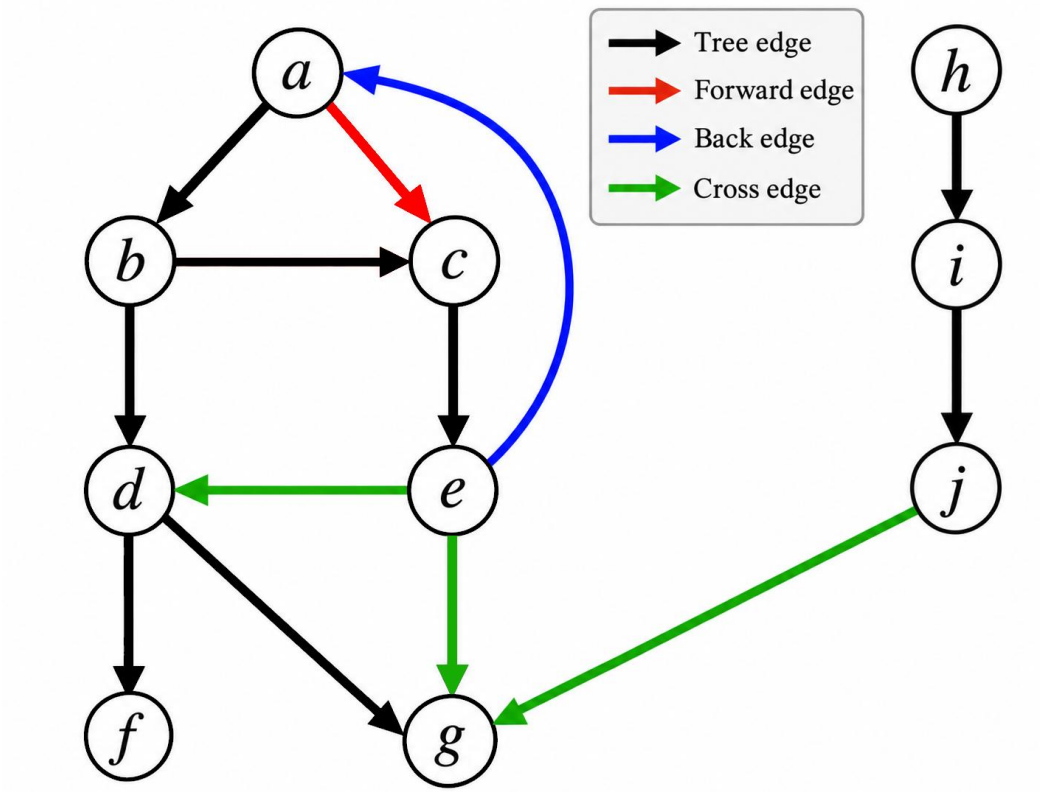
- `start[u]`: When DFS first discovered vertex $u$
- `finish[u]`: When DFS finished exploring $u$



# Classification of DFS Edges

**Definition (DFS Edges):**

- **Tree edges** occur when a vertex is first visited.
- **Forward edges** are non-tree edges that go from a vertex to one of its descendants in the tree.
- **Back edges** go from a vertex to one of its ancestors in the tree.
- **Cross edges** connect two vertices neither of which is an ancestor of the other



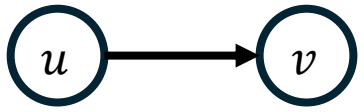
# Classification via Timestamps

**Theorem (DFS Interval Property):** For any pair of vertices $u, v$, their intervals $(s_u, f_u), (s_v, f_v)$ are either disjoint or one contains the other

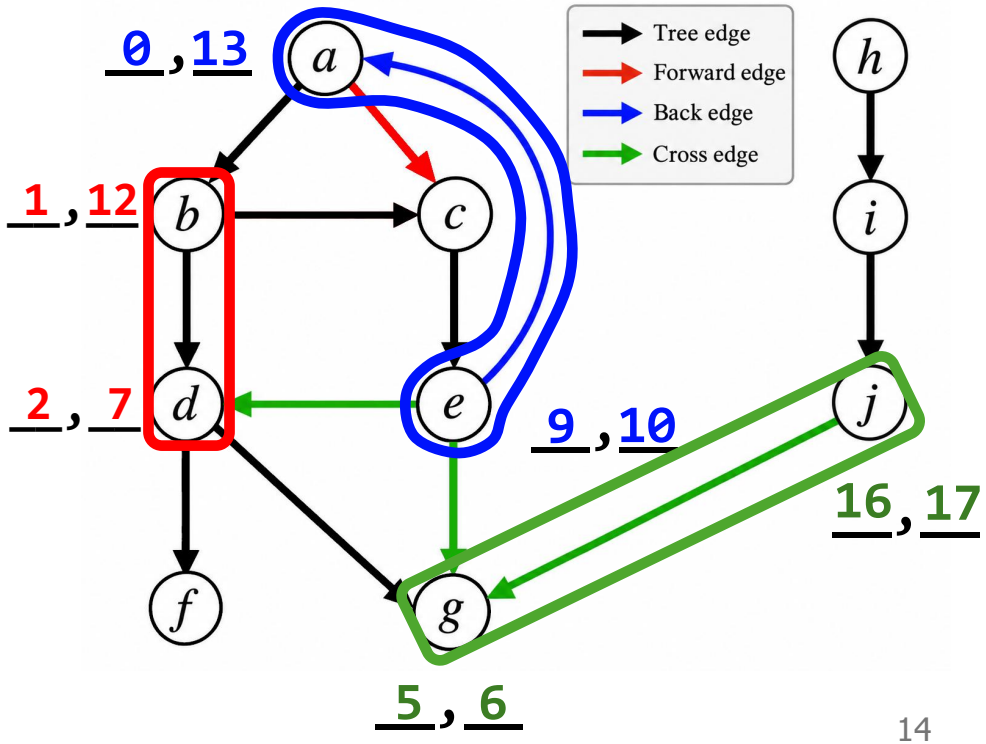
For an edge $(u, v)$, the timestamps give us three possible relationships:

- **Tree or Forward**: $s_u < s_v < f_v < f_u$
- **Back**: $s_v < s_u < f_u < f_v$
- **Cross**: $f_v < s_u$

? Why is $f_u < s_v$ impossible?




If $v$ has not started, $u$ would `dfs` $v$, hence $u$ hasn't finished yet



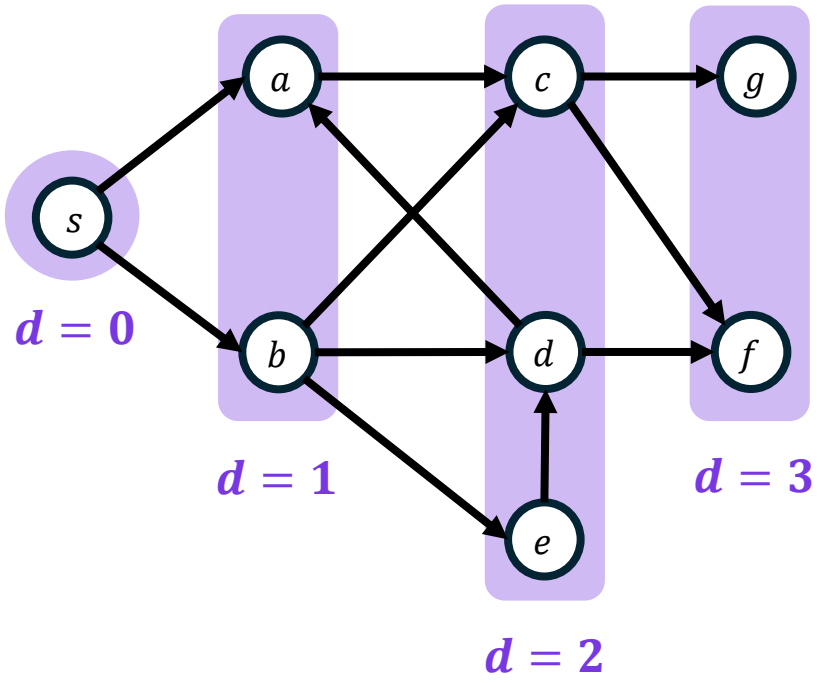
# Breadth-First Search

# Breadth-First Search (BFS)

**Problem:** We want to find the **shortest path** between two vertices.

**Recall BFS from 15-122**

- **Issue**: DFS might explore a very long path
- **Idea**: Explore vertices in increasing order of distance from the source.
  - First, vertices 0 edges away.
  - Then, vertices 1 edge away.
  - Then, vertices 2 edges away, and so on.



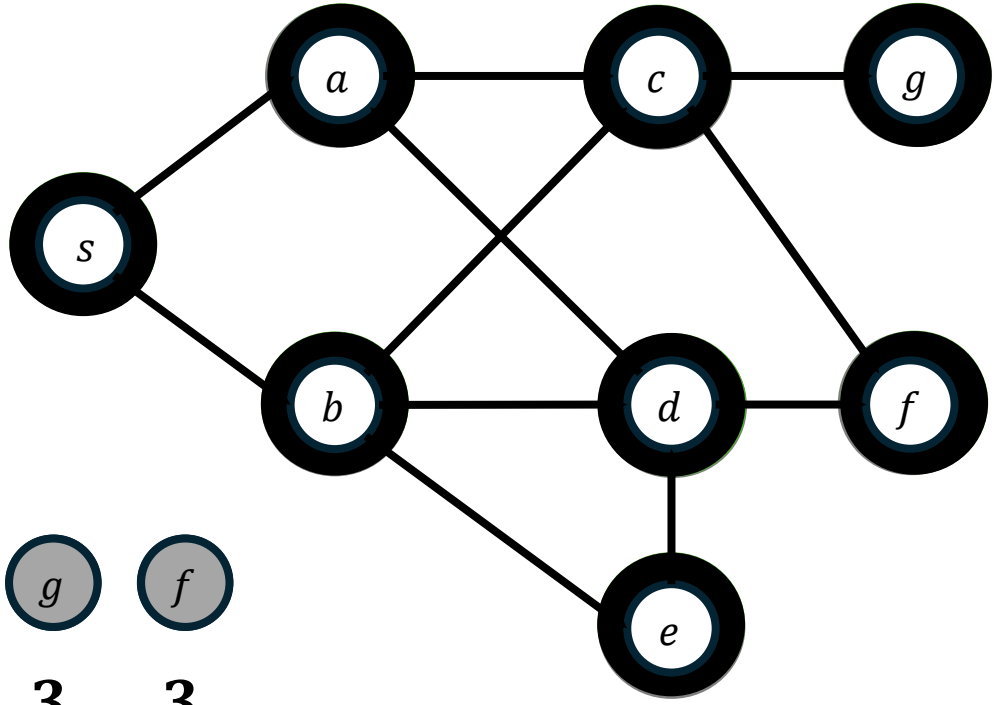

**Conclusion:** The first time BFS discovers a vertex, it has found a shortest path to that vertex

# How Do We Explore Breadth-First?

**A FIFO queue** lets us explore vertices in the order they are discovered

**As seen in 15-122:**

- Initially, **enqueue** the source vertex $s$
- Repeatedly **dequeue** the next vertex $u$ and examine its outgoing edges.
- Whenever we discover an **unvisited** neighbour $v$, mark it visited and **enqueue** it.




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **QUEUE:** | $s$ | $a$ | $b$ | $c$ | $d$ | $e$ | $g$ | $f$ |
| **DISTANCE:** | **0** | **1** | **1** | **2** | **2** | **2** | **3** | **3** |

# Sequential BFS Implementation

```
fun bfs(adj : UnweightedAdjList, s : int) -> sequence<int>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]   // pretend that ∞ is an int

  Q : queue<int> = empty          // assume we have a queue implementation
  Q.enqueue(s)
  dist[s] ← 0
  while not Q.is_empty():
    u = Q.dequeue()
    for v in adj[u]:
      if dist[v] == ∞:
        dist[v] ← dist[u] + 1
        Q.enqueue(v)

  return dist
```

**Cost:** (Sequential) BFS costs $O(n + m)$ work (and span)

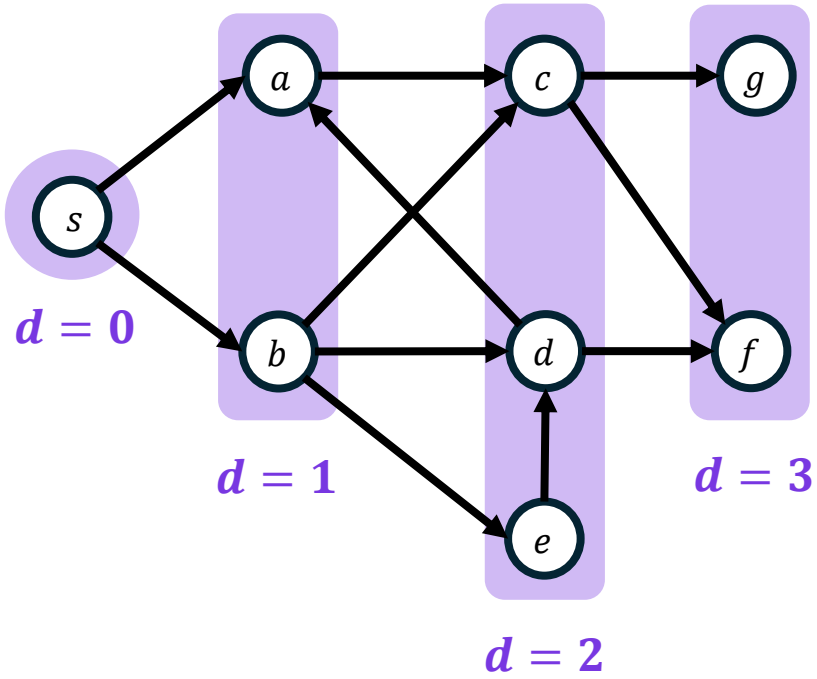
The queue is **completely sequential**. No room for parallelism!

# BFS in Layers

**Recall:** Our original idea for BFS was to explore vertices **layer by layer**

The FIFO queue does this implicitly:

| **Q:** | $s$ | $a$ | $b$ | $c$ | $d$ | $e$ | $g$ | $f$ |
|---|---|---|---|---|---|---|---|---|
| **Dist:** | **0** | **1** | **1** | **2** | **2** | **2** | **3** | **3** |



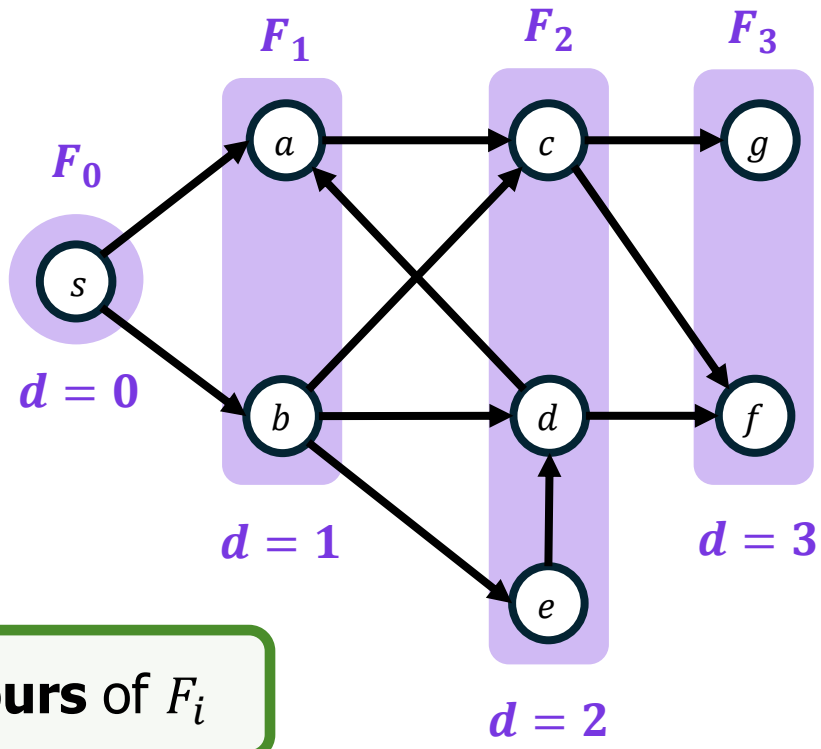

**Fact:** At any point in time, the queue contains vertices of at most two distances: the current layer and the next layer

# BFS in Layers

**Key Idea:** Make the **layers explicit** instead of using a queue

Instead of processing the vertices from a queue one by one:

- Maintain the current **frontier** $F_i$ containing all vertices at distance $i$
- Obtain the vertices in the next frontier $F_{i+1}$ from $F_i$




**Definition (Frontier):** $F_i := \{v : \mathrm{dist}(s, v) = i\}$

**Observe:** $F_{i+1}$ is precisely the **undiscovered neighbours** of $F_i$

# Layered BFS Implementation

```
fun bfs(adj : UnweightedAdjList, s : int) -> sequence<int>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]

  dist[s] ← 0
  frontier = [s]
  d = 0

  while |frontier| > 0:
    next_frontier = []
    for u in frontier:
      for v in adj[u]:
        if dist[v] == ∞:
          dist[v] ← d+1
          next_frontier.append(v)

    frontier = next_frontier
    d ← d+1

  return dist
```

**Cost:** (Layered) BFS costs $O(n + m)$ work (and span)

The next frontier is a **function of the previous frontier**

But it's still completely **sequential**…

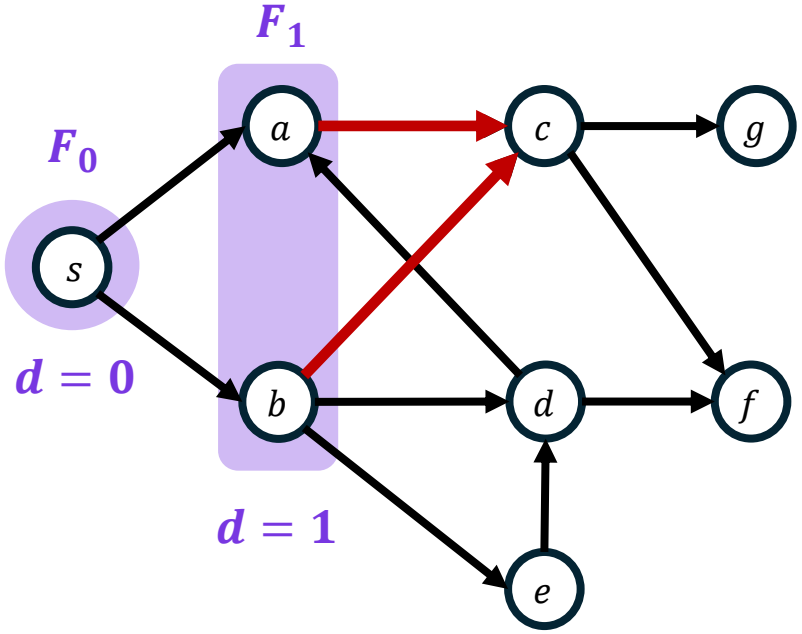
# Parallelizing BFS

**Idea:** Vertices in the frontier can examine their outgoing edges **in parallel**

- Can not be trivially parallelized!

```
if dist[v] == ∞:
  dist[v] ← d
  next_frontier.append(v)
```

- Main issue: **Duplicates**

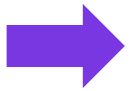



**Challenge**: How can we construct the next frontier in parallel without **duplicate discoveries** or **data races**?

# Parallel Implementation

- The only change is the **next frontier computation**

```
next_frontier = []
for u in frontier:
  for v in adj[u]:
    if dist[v] == ∞:
      dist[v] ← d
      next_frontier.append(v)
```



```
neighbours = flatten(map(fn u => adj[u], frontier))
unvisited = filter(fn v => dist[v] == ∞, neighbours)
next_frontier = remove_duplicates(unvisited)

parallel for v in next_frontier:
  dist[v] ← d
```

? How do we implement `remove_duplicates`?

```
fun remove_duplicates(S : sequence<T>) -> sequence<T>:
  return unique(sort(S))
```

**Note**: The cost of `remove_duplicates` is $O(|S| \log |S|)$ work. This is **more work** than sequential BFS!

# Parallel BFS Implementation

```
fun bfs(adj : UnweightedAdjList, s : int) -> sequence<int>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]
  dist[s] ← 0
  frontier = [s]
  d = 0

  while |frontier| > 0:
    neighbours = flatten(map(fn u => adj[u], frontier))
    unvisited = filter(fn v => dist[v] == ∞, neighbours)
    next_frontier = remove_duplicates(unvisited)

    parallel for v in next_frontier:
      dist[v] ← d+1

    frontier = next_frontier
    d ← d+1

  return dist
```

# Parallel BFS Cost Analysis

**Definition:** We define the **volume** of a frontier:

$$\|F_i\| \coloneqq \sum_{u \in F_i} \left(1 + d_G^+(u)\right) = |F_i| + \sum_{u \in F_i} d_G^+(u)$$

- The volume of $F_i$ is the number of vertices $|F_i|$ and the total number of **outgoing edges** $\sum d_G^+(u)$ of those vertices

**Lemma:**

$$\sum_{0 \le i \le D} \|F_i\| = O(n + m)$$

*Proof: Each reachable vertex is in **exactly one frontier.** Each edge is therefore in at most one frontier (or two for an undirected graph).*

# Parallel BFS Cost Analysis

**Theorem**: Let $D$ be the largest distance to any vertex reachable from $s$. Parallel BFS costs $O\big((m+n)\log(n+m)\big)$ work and $O\big((D+1)\log^2(n+m)\big)$ span.

*Proof: There are* $(D+1)$ *frontier calculations*

- *The total size of* `neighbours` *is* $\sum d_G^+(u) \leq \|F_i\|$
- ***`flatten`*** *and* ***`filter`*** *cost* $O(\|F_i\|)$ *work and* $O(\log \|F_i\|)$ *span*
- *The cost is dominated by* ***`remove_duplicates`*** *which sorts and hence costs* $O(\|F_i\| \log\|F_i\|)$ *work and* $O(\log^2 \|F_i\|)$ *span*
- *Summing over every layer, the work is:*

$$\sum_{0\leq i\leq D} O(\|F_i\| \log \|F_i\|) = O\big((n+m)\log(n+m)\big)$$

```
neighbours = flatten(map(fn u => adj[u], frontier))
unvisited = filter(fn v => dist[v] == ∞, neighbours)
next_frontier = remove_duplicates(unvisited)
```

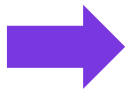
# Functional Parallel BFS

Our parallel BFS used parallel mutation. Can we do without?

**Idea**: use a **persistent dictionary** instead of a mutable sequence to store distances

```
neighbours = flatten(map(fn u => adj[u], frontier))
unvisited = filter(fn v => dist[v] == ∞, neighbours)
next_frontier = remove_duplicates(unvisited)
```

```
parallel for v in next_frontier:
  dist[v] ← d+1
```



```
type dist_dict = treap_dict<int, int>

neighbours = flatten(map(fn u => adj[u], frontier))
unvisited = filter(fn v=>dist.find(v)==NONE, neigh…)
next_frontier = remove_duplicates(unvisited)
```

```
new_dists = dist_dict.from_sequence(
              map(fn v => (v, d+1), next_frontier)
next_dist = union(dist, new_dists)
```

- A vertex is undiscovered exactly when its key is absent from `dist`.
- `union` adds an **entire frontier's distances in parallel**

# Functional Parallel BFS Implementation

```
type dist_dict = treap_dict<int, int>

fun bfs(adj: UnweightedAdjList, s: int) -> dist_dict:

  fun explore(frontier: sequence<int>, dist: dist_dict, d: int) -> dist_dict:
    if |frontier| == 0: return dist

    neighbors = flatten(map(fn u => adj[u], frontier))
    unvisited = filter(fn v => dist.find(v) == NONE, neighbors)
    next_frontier = remove_duplicates(unvisited)

    new_dists = dist_dict.from_sequence(map(fn v => (v, d+1), next_frontier))
    next_dist = union(dist, new_dists)

    return explore(next_frontier, next_dist, d+1)

  return explore([s], {s:0}, 0)
```

# Functional Parallel BFS Cost Analysis

**Theorem**: Functional parallel BFS also costs $O\big((m+n)\log(n+m)\big)$ work and $O\big((D+1)\log^2(n+m)\big)$ span[1].

*Proof: The cost is the same as the imperative BFS except for* ***`from_sequence`*** *and* ***`union`***

- ***`from_sequence`*** *costs* $O(|F_{i+1}|\log|F_{i+1}|)$ *work and* $O(\log^2|F_{i+1}|)$ *span (sorting)*
- *Recall* `union(A,B)` *for* $|A| \leq |B|$ *costs at most* $O(|A|\log|B|)$ *work and* $O(\log|B|)$ *span*
- *On level i, the cost of union is therefore* $O(|F_{i+1}|\log n)$ *work and* $O(\log n)$ *span*
- *Summing over all levels, since* $\sum|F_{i+1}| \leq n$*, the total extra work is* $O(n\log n)$ *and the total extra span is* $O\big((D+1)\log^2(n)\big)$*, which is within cost bounds*

```
new_dists = dist_dict.from_sequence(map(fn v => (v, d+1), next_frontier))
next_dist = union(dist, new_dists)
```

[1]Assuming we use a deterministic balanced BST instead of a treap. Otherwise, the bounds hold w.h.p.

# BFS: From Sequential to Parallel

- We managed to parallelize BFS and keep it purely functional!

| Version | Key idea | Work | Span |
|---|---|---|---|
| **Classic BFS** | FIFO queue | $O(n+m)$ | $O(n+m)$ |
| **Layered BFS** | Explicit frontiers | $O(n+m)$ | $O(n+m)$ |
| **Imperative parallel BFS** | Parallel frontier construction | $O\big((n+m)\log(n+m)\big)$ | $O\big((D+1)\log^2(n+m)\big)$ |
| **Functional parallel BFS** | Persistent BST dictionary | $O\big((n+m)\log(n+m)\big)$ | $O\big((D+1)\log^2(n+m)\big)$ |

**Key takeaway**: BFS has sequential dependencies **between frontiers**, but substantial parallelism **within each frontier**. Quite parallel on real-world graphs with low diameter.

# Summary

## DFS: A Reusable Traversal

- **Reachability and connectivity**: Track visited vertices and recur.
- **Cycle detection**: Distinguish currently active vertices from finished vertices.
- **Topological sorting**: Utilize the DFS finishing order as a customization point.
- **Timestamps**: Record traversal history to classify edges and DFS tree ancestry.

## BFS: Sequential to Parallel

- **Shortest paths**: Explore vertices in increasing order of distance.
- **Explicit frontiers**: Replace the FIFO queue with distance layers.
- **Parallel BFS**: Process each frontier using parallel sequence operations.
- **Functional BFS**: Replace mutable distances with persistent BST dictionary.

**DFS** provides a versatile template for many graph algorithms but is (mostly) sequential.
**BFS** finds shortest paths and exposes parallelism by organizing traversal into frontiers.