Directed Acyclic Graphs & Topological Sort

CS16: Introduction to Data Structures & Algorithms

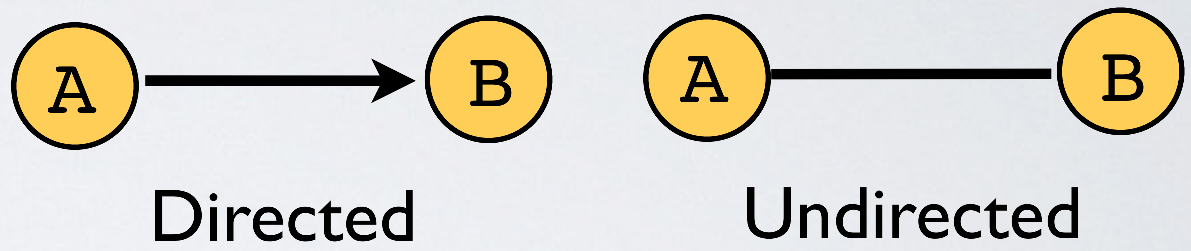
Summer 2021

A problem

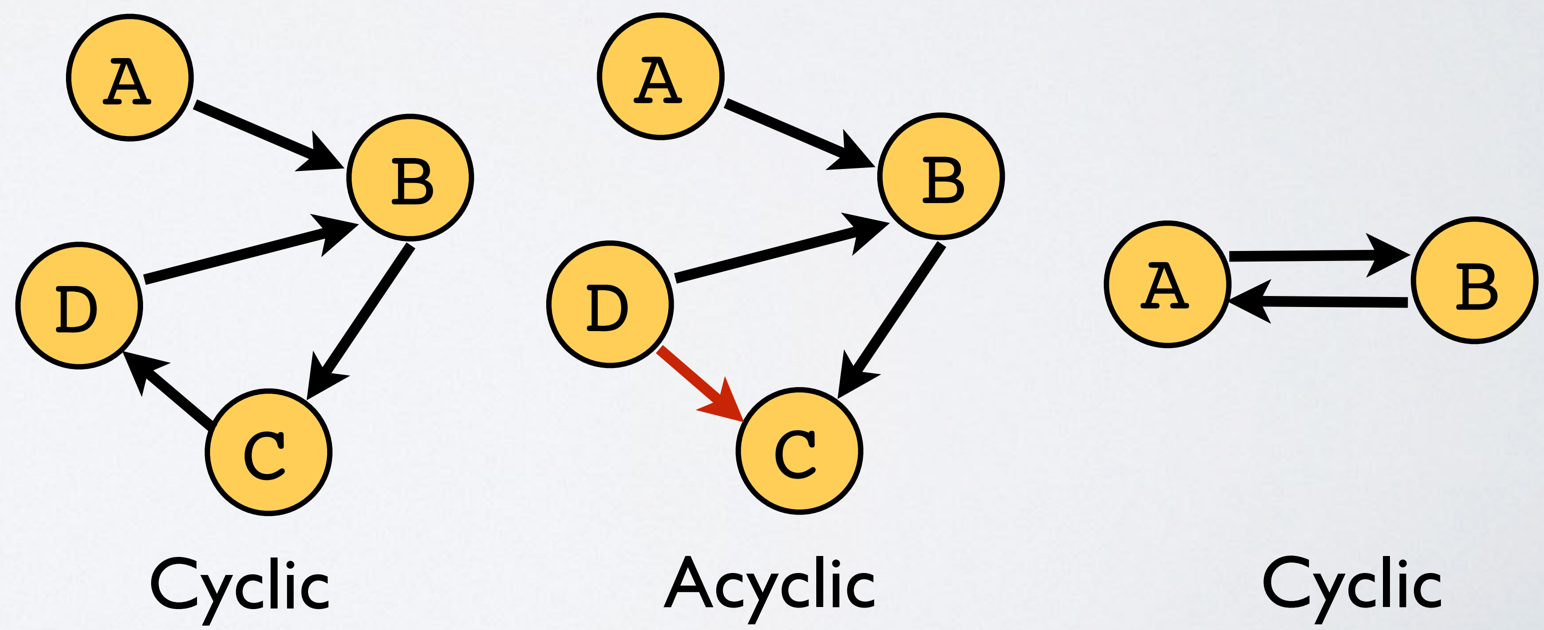
- We have a collection of **tasks** we want to accomplish
  - Some tasks depend on other tasks
  - Some are independent
- In what **order** should I do these tasks?
- Example: I make really good burritos
  - Need to chop an onion before sautéing it
  - But, can sauté onion and cook rice simultaneously
  - BAD: sauté onions, chop onions, cook rice
  - GOOD: chop onions, cook rice, sauté onions

Directed Acyclic Graphs

- A DAG is **directed** & **acyclic**
- Directed
  - edges have origin & destination…
  - ….represented by a directed arrow

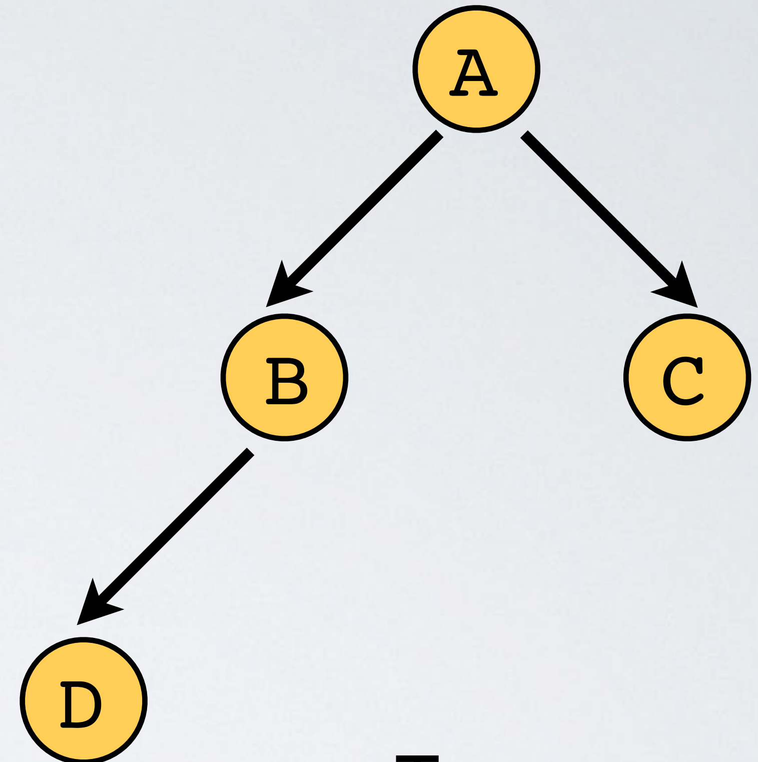


- Acyclic
  - No cycles!
  - Starting from any vertex, there is no path that leads back to the same vertex



Trees and DAGs

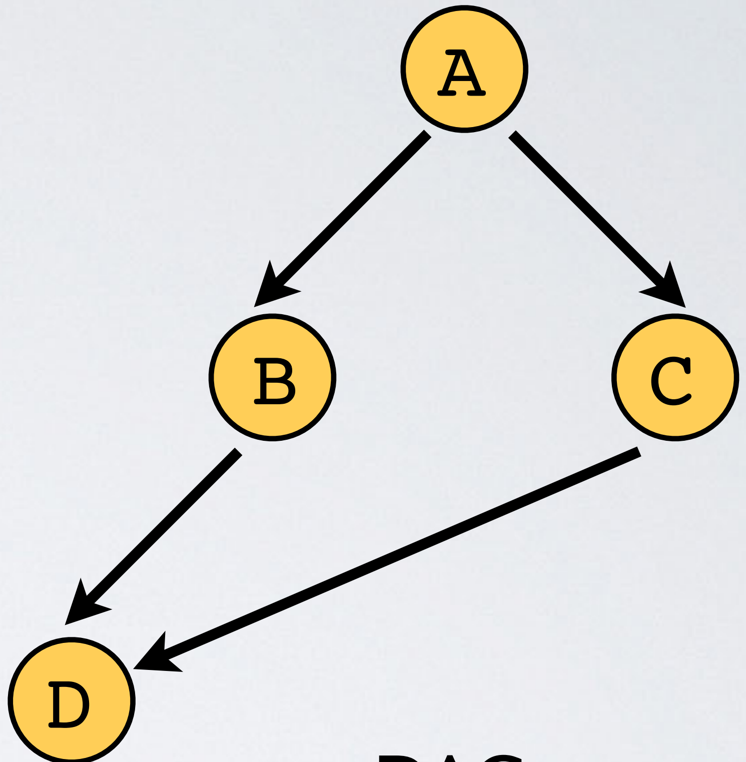
- All trees are DAGs
- **Not** all DAGs are trees!



Tree

Trees and DAGs

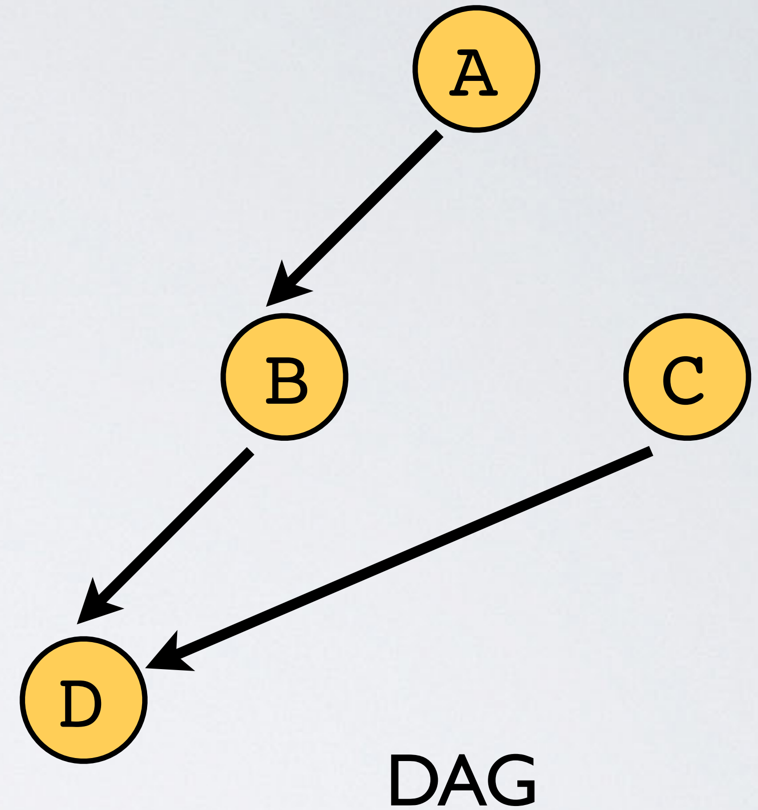
- All trees are DAGs
- **Not** all DAGs are trees!



DAG

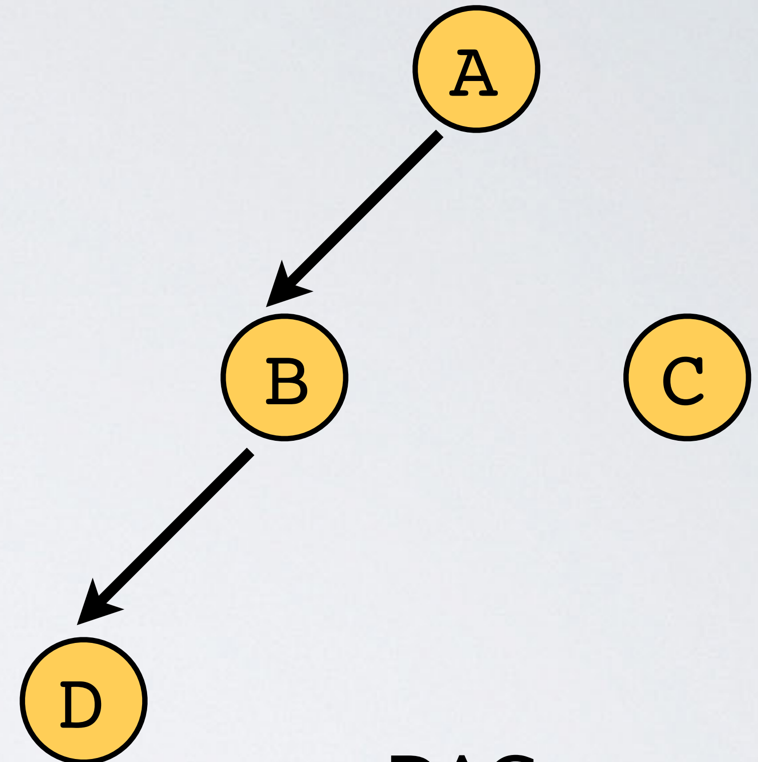
Trees and DAGs

- All trees are DAGs
- **Not** all DAGs are trees!



Trees and DAGs

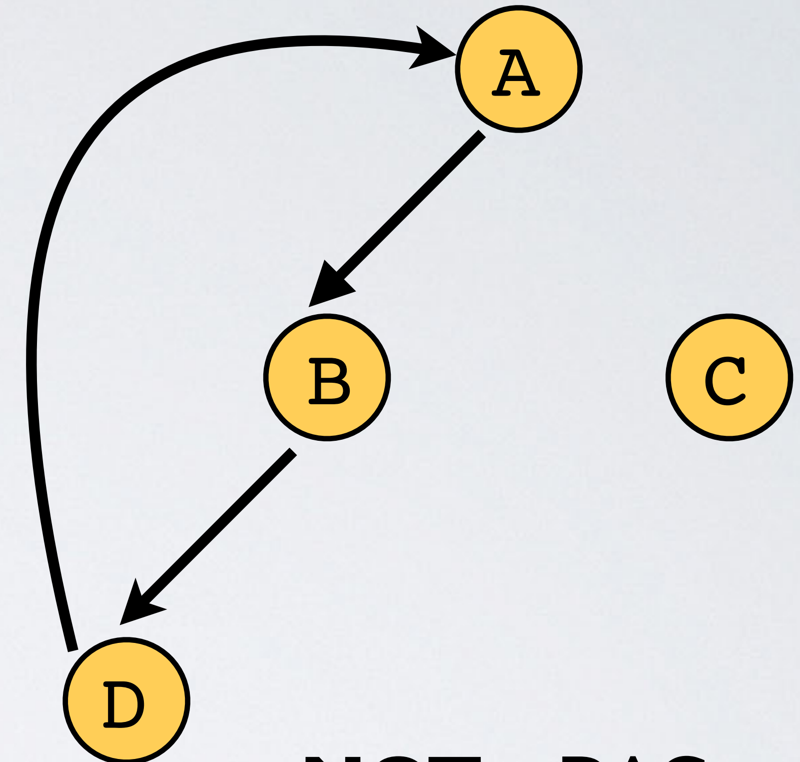
- All trees are DAGs
- **Not** all DAGs are trees!



DAG

Trees and DAGs

- All trees are DAGs
- **Not** all DAGs are trees!

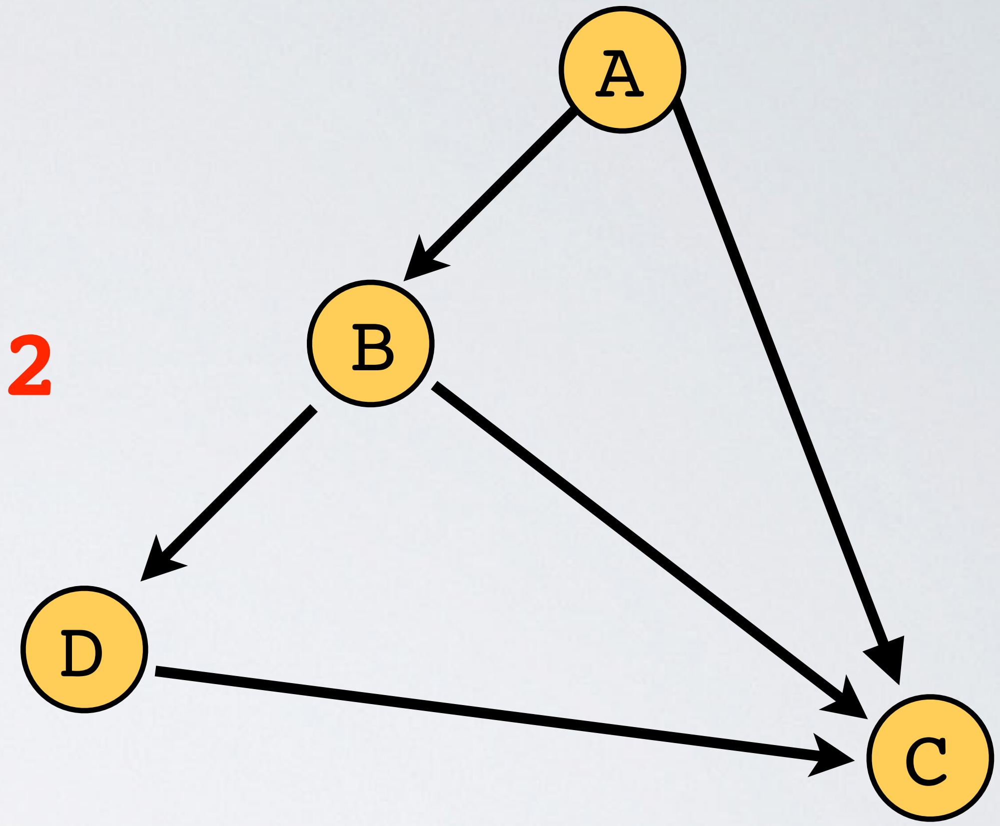


NOT a DAG

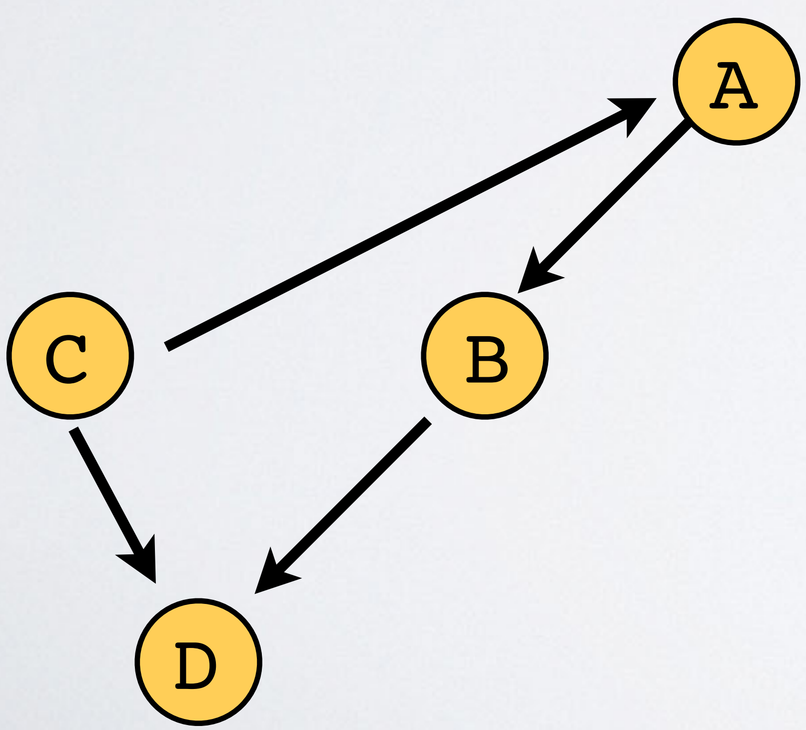
Which are DAGs?

1

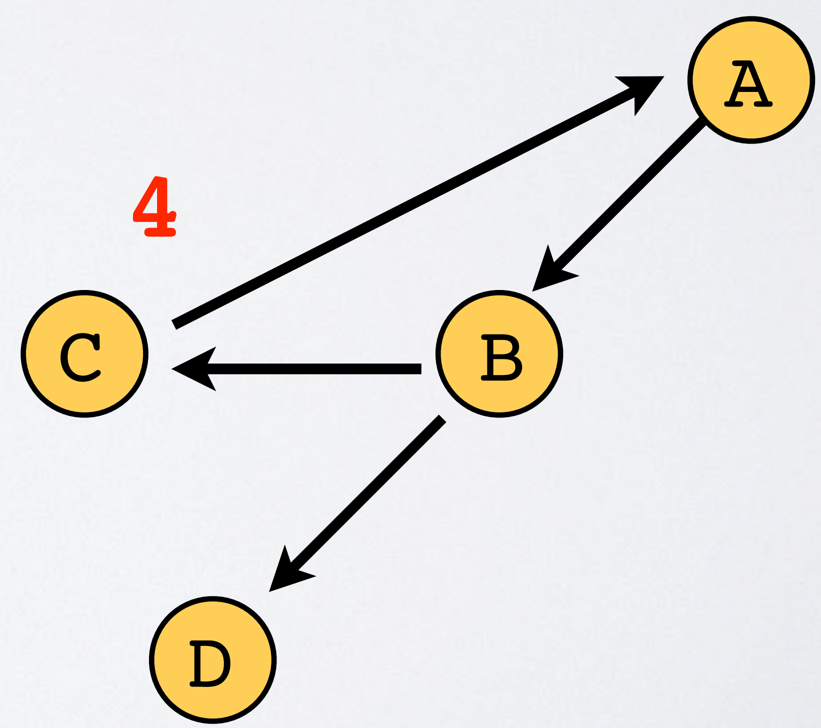
2



3



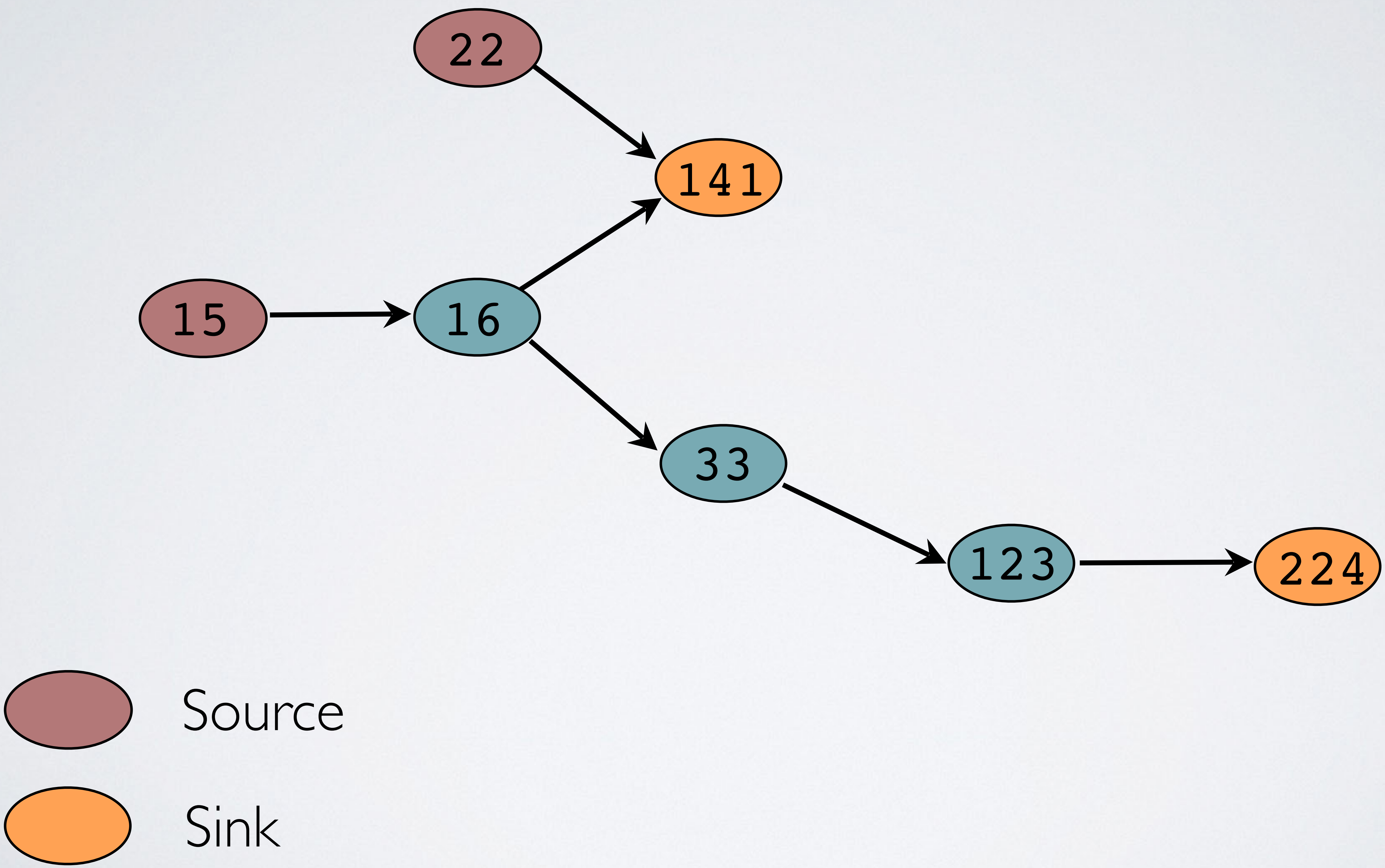
4



Directed Acyclic Graphs

- DAGs often used to model situations in which completing certain things depend on completing other things
  - ex: course prerequisites or small tasks in a big project
- Terminology
  - Sources: vertices with no incoming edges (no dependencies)
  - Sinks: vertices with no outgoing edges
  - In-degree of a vertex: number of incoming edges of the vertex
  - Out-degree of a vertex: number of outgoing edges of the vertex

Directed Acyclic Graphs — Example

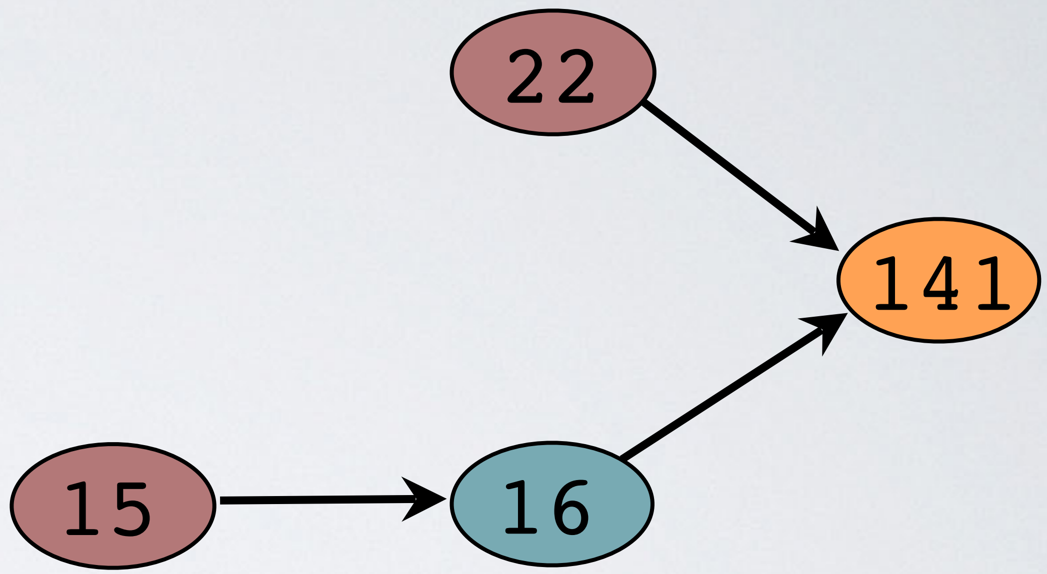


Topological Sort

- Imagine you are a CS concentrator
- You need to plan your courses for next 3 years
- How can you do that taking into account pre-requisites?
  - Represent courses w/ a DAG
  - Use topological sort!
    - Produces topological ordering of a DAG

Topological Sort

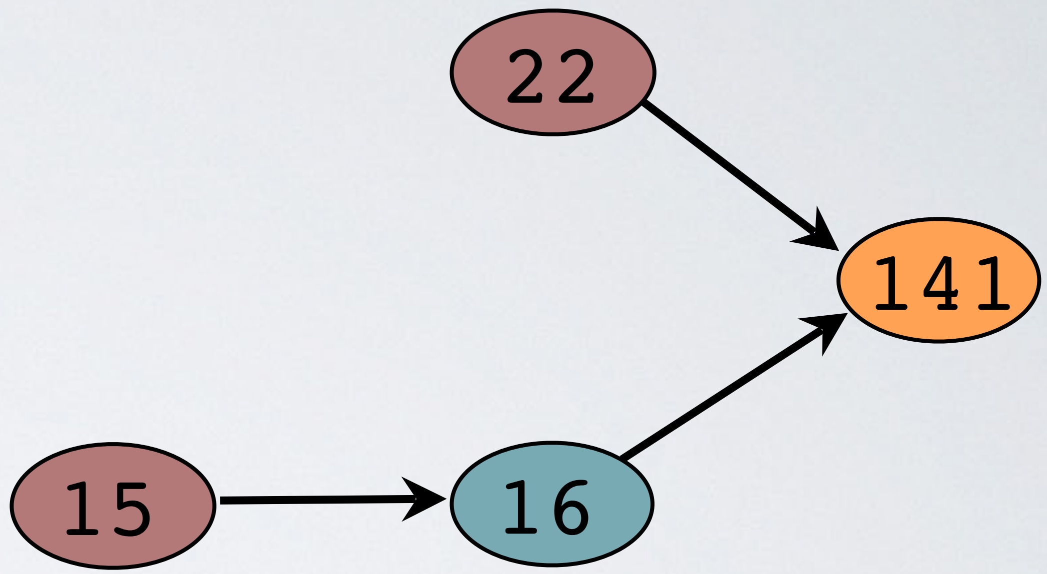
- Topological Ordering
  - ordering of vertices in DAG…
  - …such that for each vertex v…
  - …all of v's prereqs come before it in the ordering
- Topological Sort
  - Algorithm that produces topological ordering given a DAG



- Valid topological orderings
  - `15,16,22,141`

Topological Sort

- Topological Ordering
  - ordering of vertices in DAG…
  - …such that for each vertex v…
  - …all of v's prereqs come before it in the ordering
- Topological Sort
  - Algorithm that produces topological ordering given a DAG

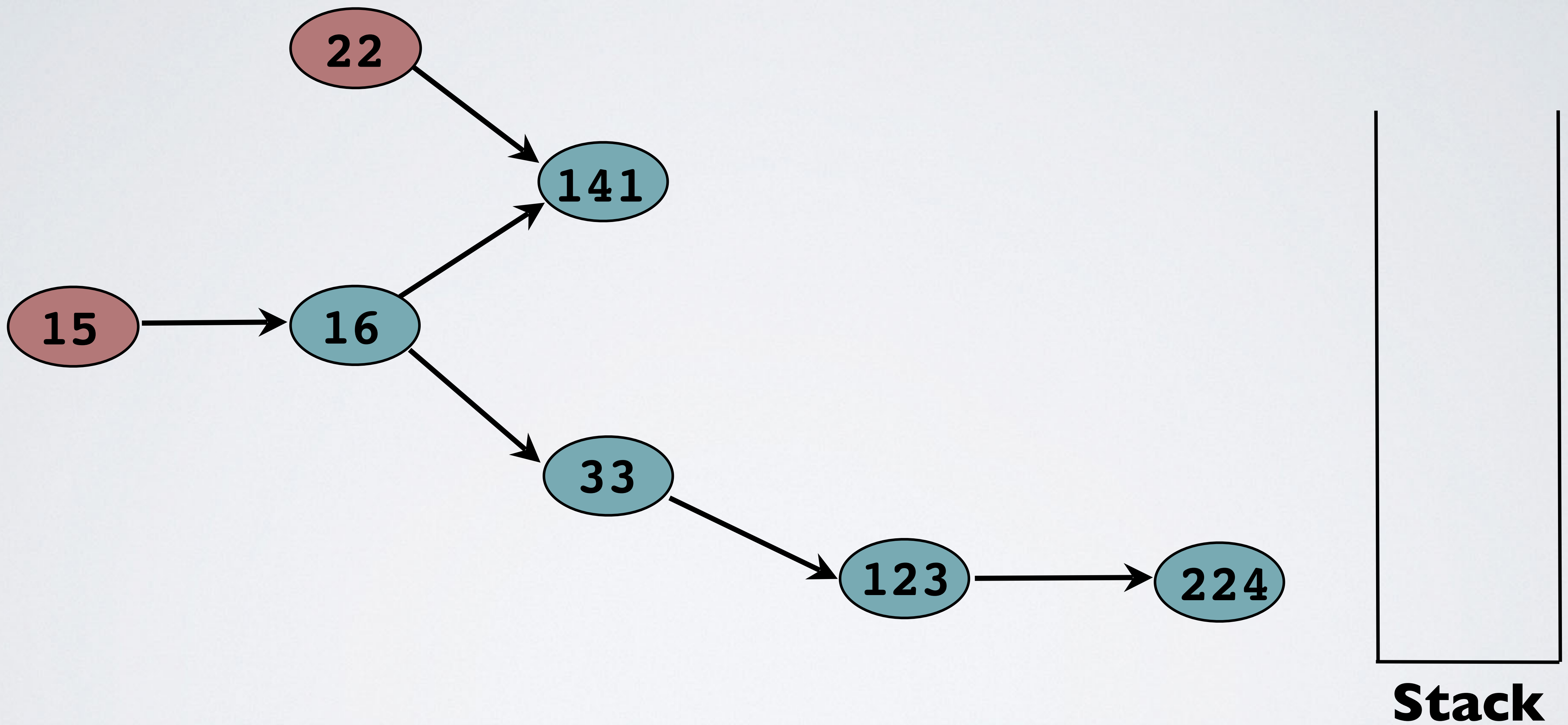


- Valid topological orderings
  - `15,16,22,141`
  - `22,15,16,141`
  - `15,22,16,141`

Topological Sort—General Strategy

- If vertex has no prerequisites (i.e., is a source), we can visit it!
- Once we visit a vertex,
  - all of it's outgoing edges can be deleted
  - because that prerequisite has been satisfied
- Deleting edges might create new sources
  - which we can now visit
- Data Structures needed
  - DAG to top-sort
  - A structure to keep track of sources
  - A list to keep track of the resultant topological ordering

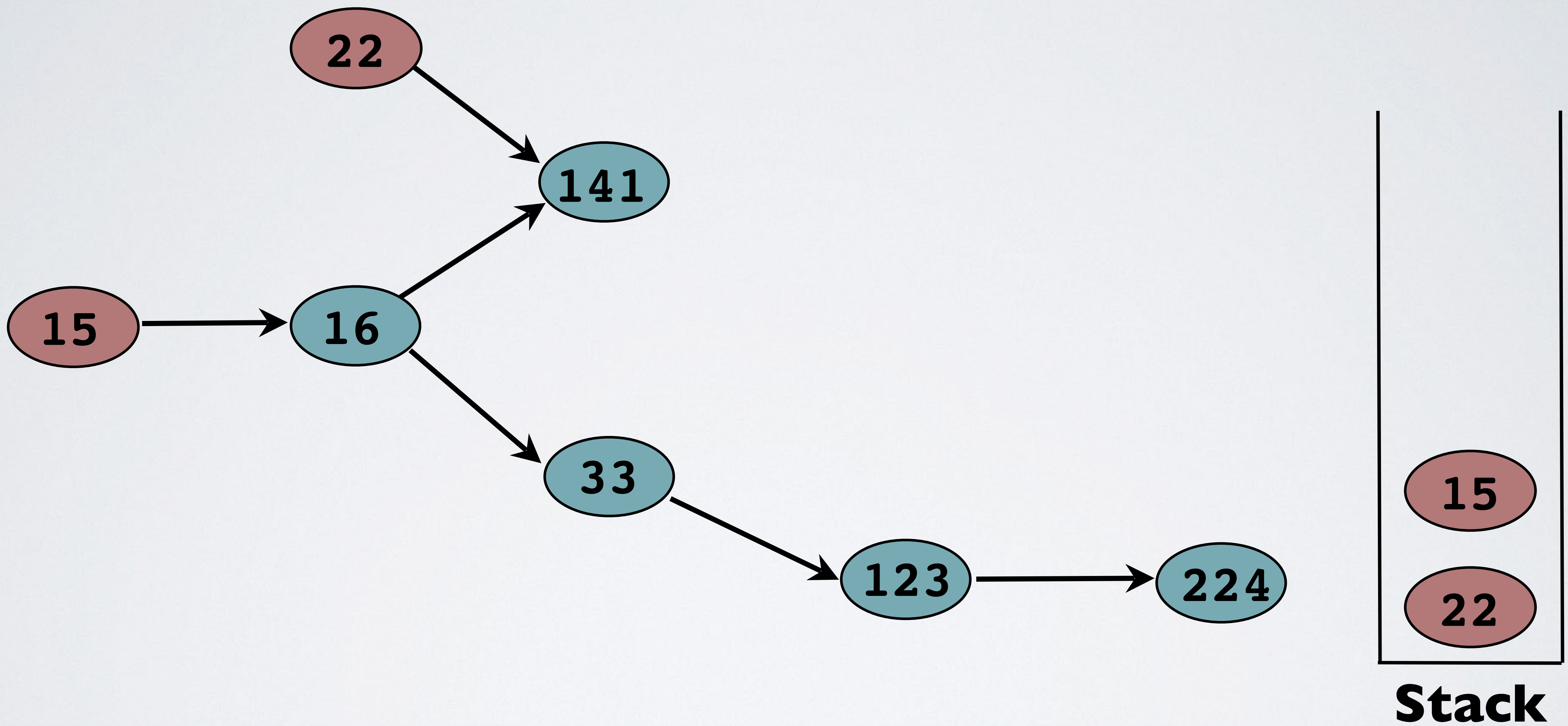
Topological Sort—Simulation



List:

Topological Sort—Simulation

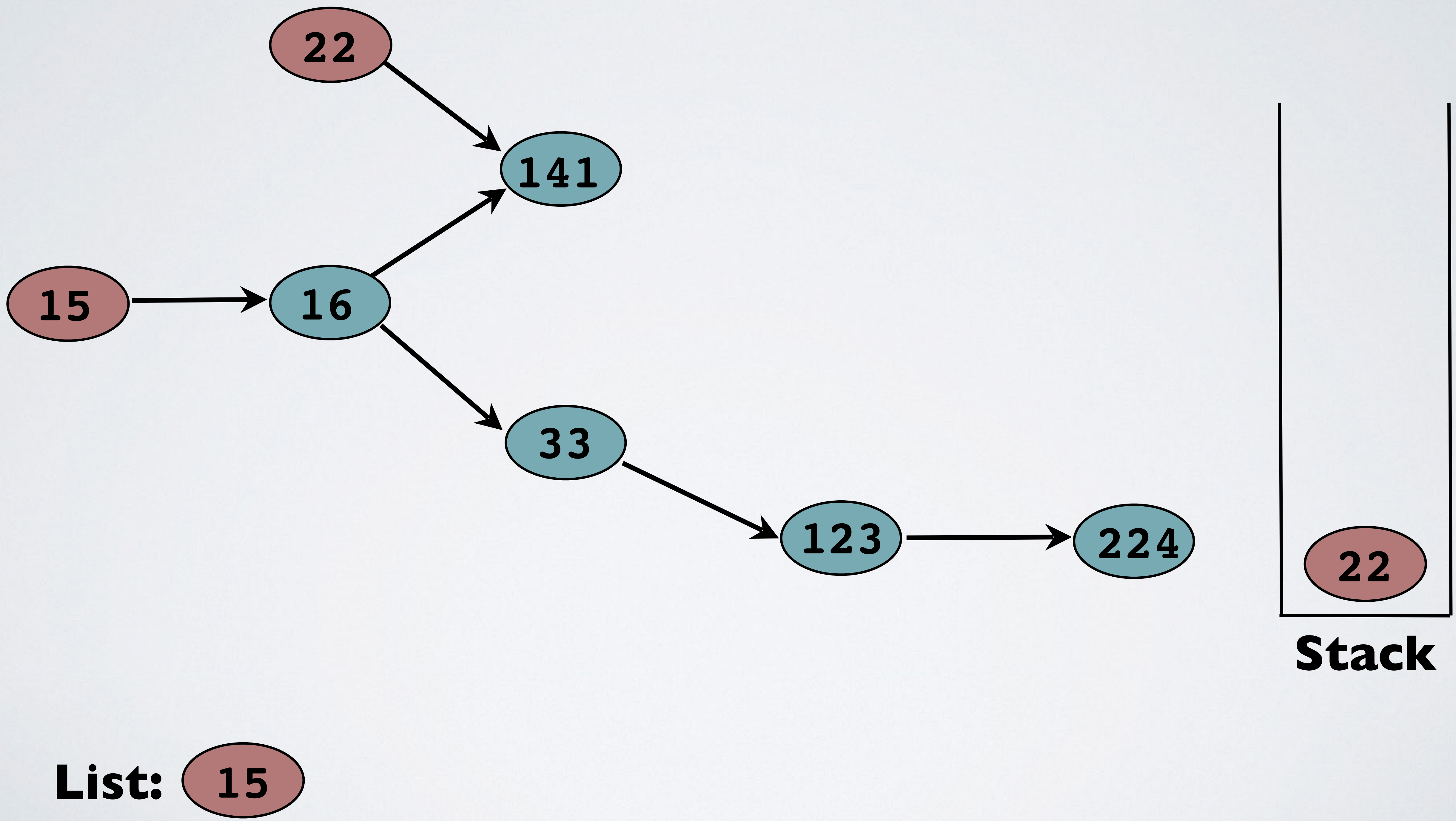
Populate Stack with source vertices



List:

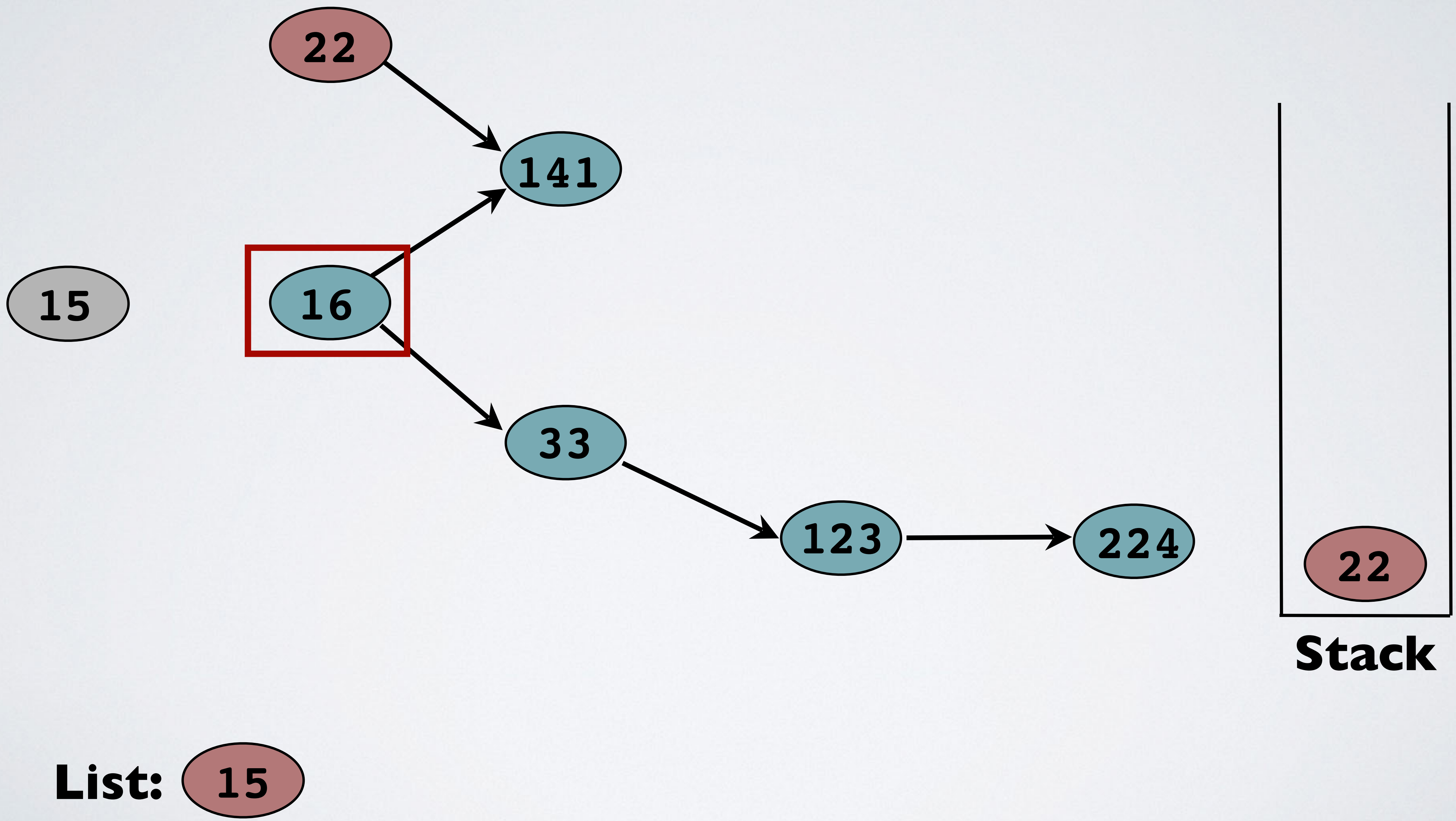
Topological Sort—Simulation

Pop from stack and add to list



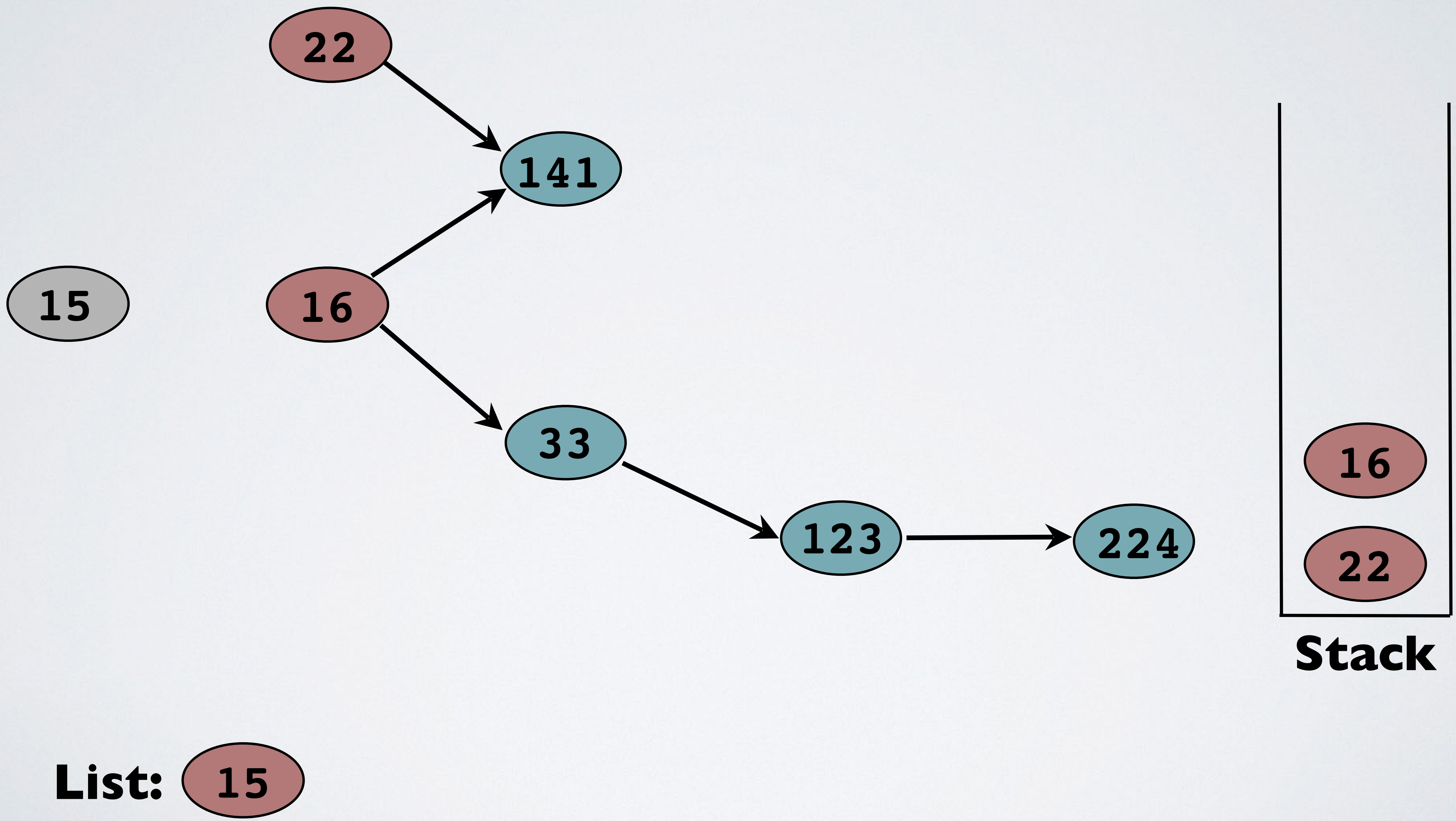
Topological Sort—Simulation

Remove outgoing edges & check corresponding vertices



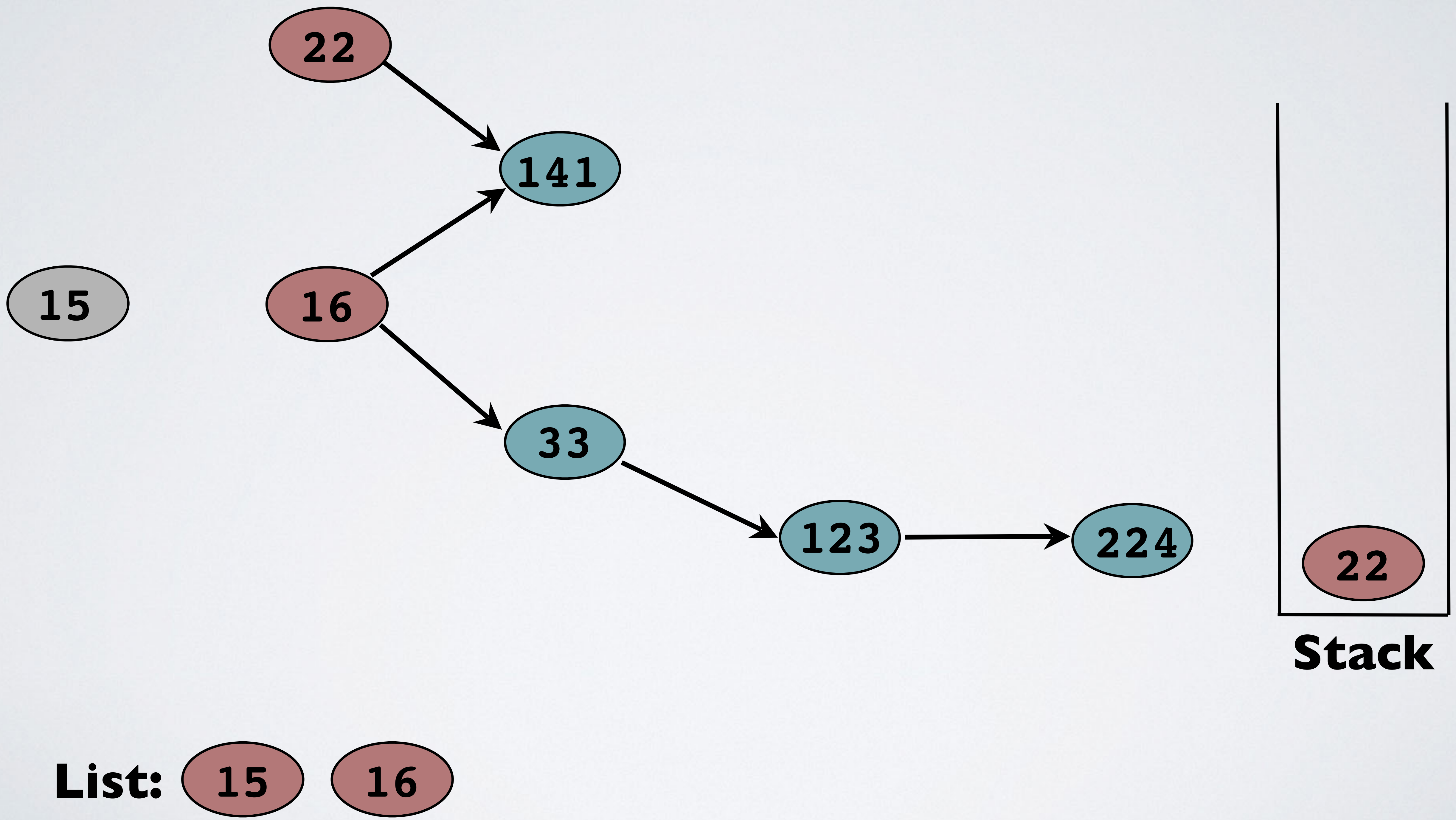
Topological Sort—Simulation

16 has no more incoming edges so push it on the stack



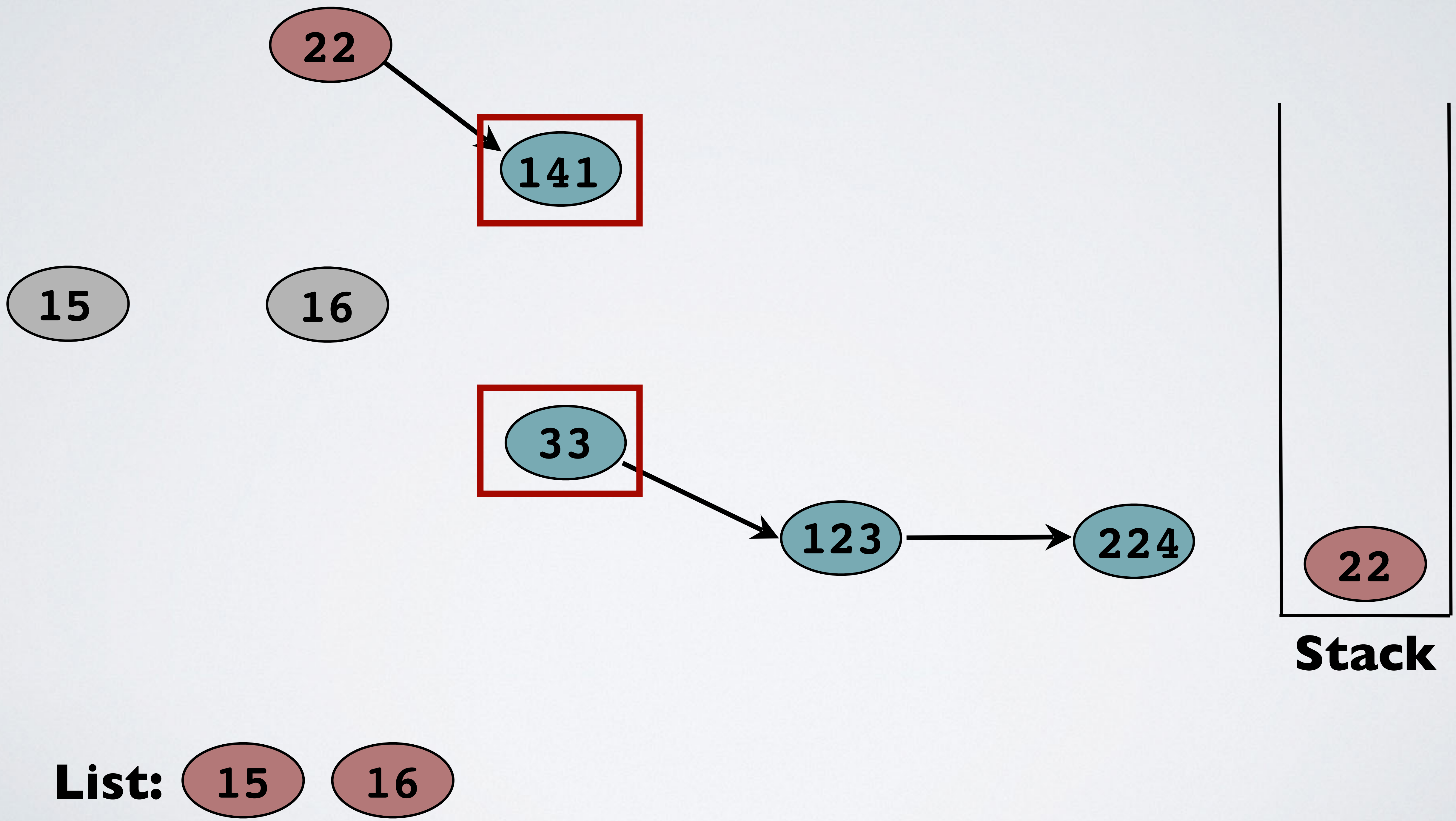
Topological Sort—Simulation

Pop from the stack and add to list



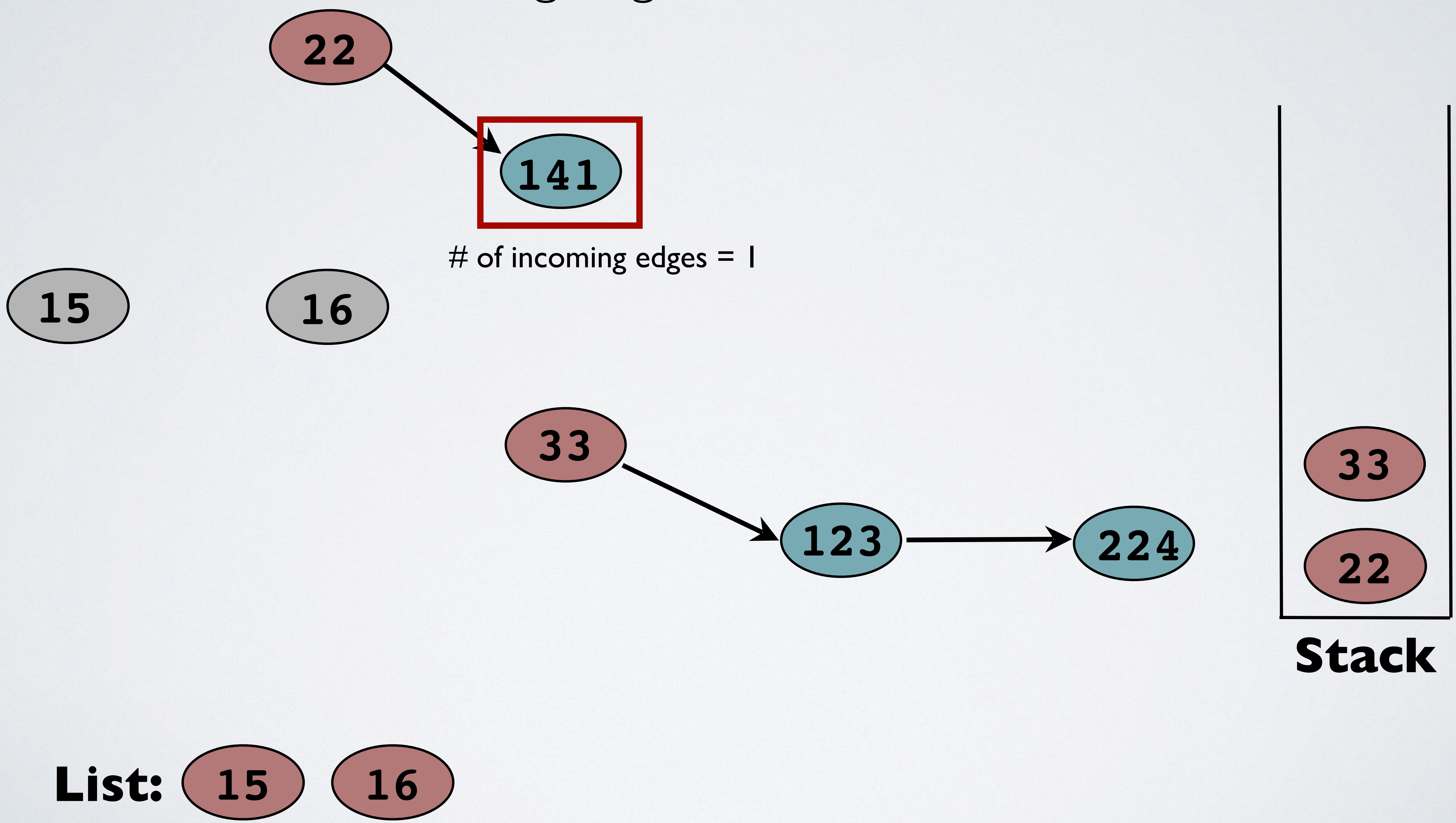
Topological Sort—Simulation

Remove outgoing edges & check the corresponding vertices



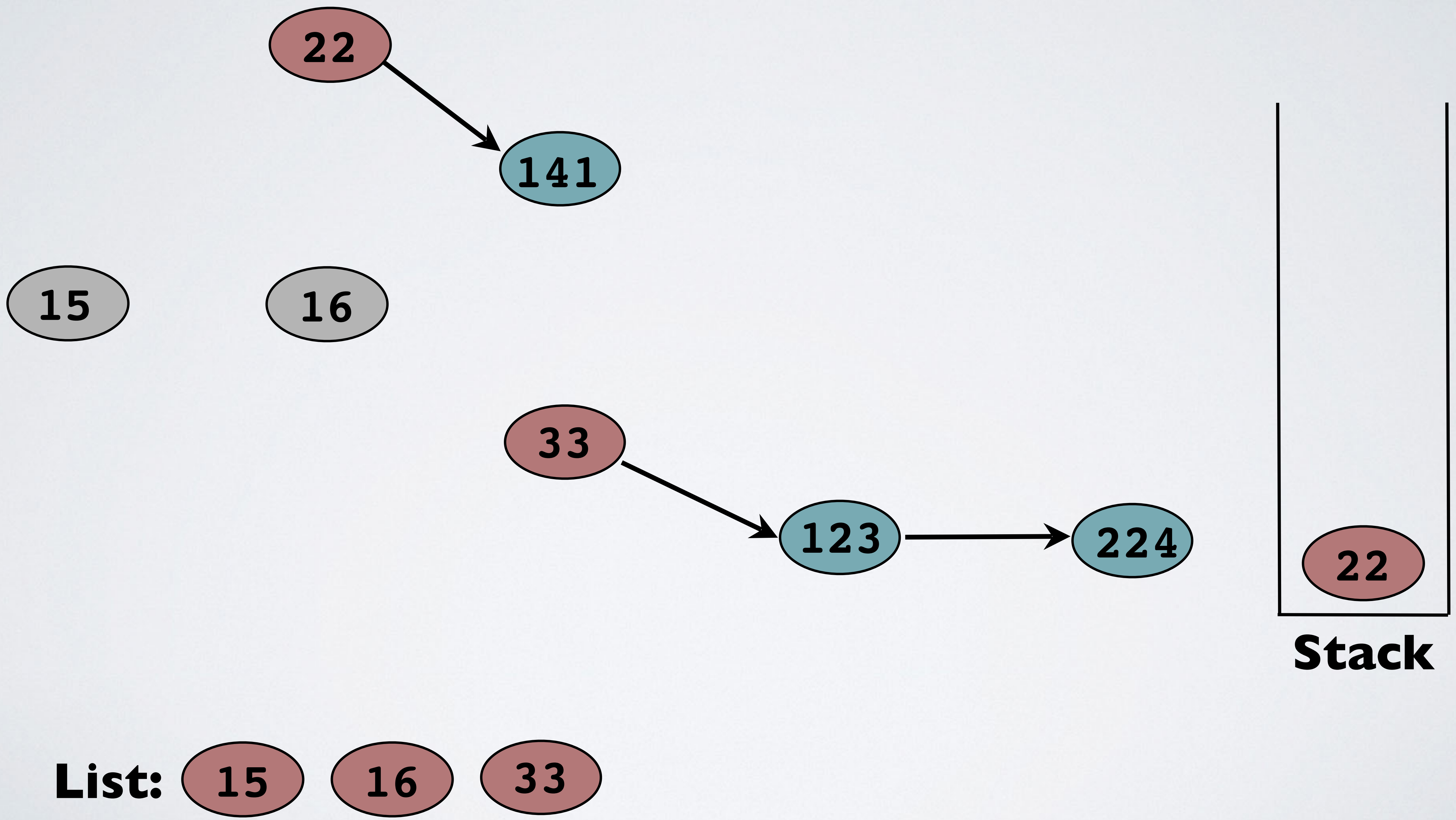
Topological Sort—Simulation

33 has no more incoming edges so push it onto the stack
141 still has an incoming edge

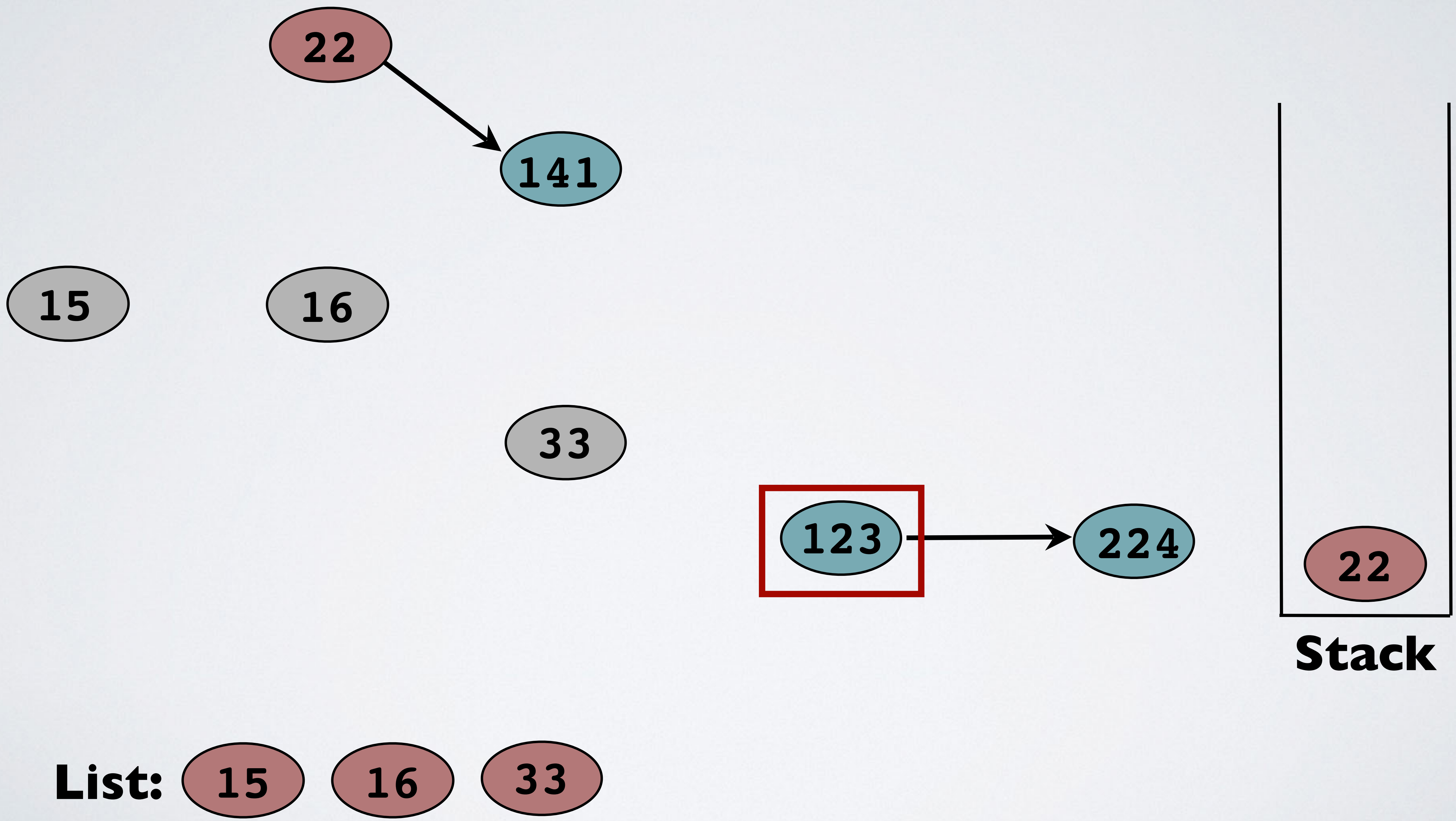


Topological Sort—Simulation

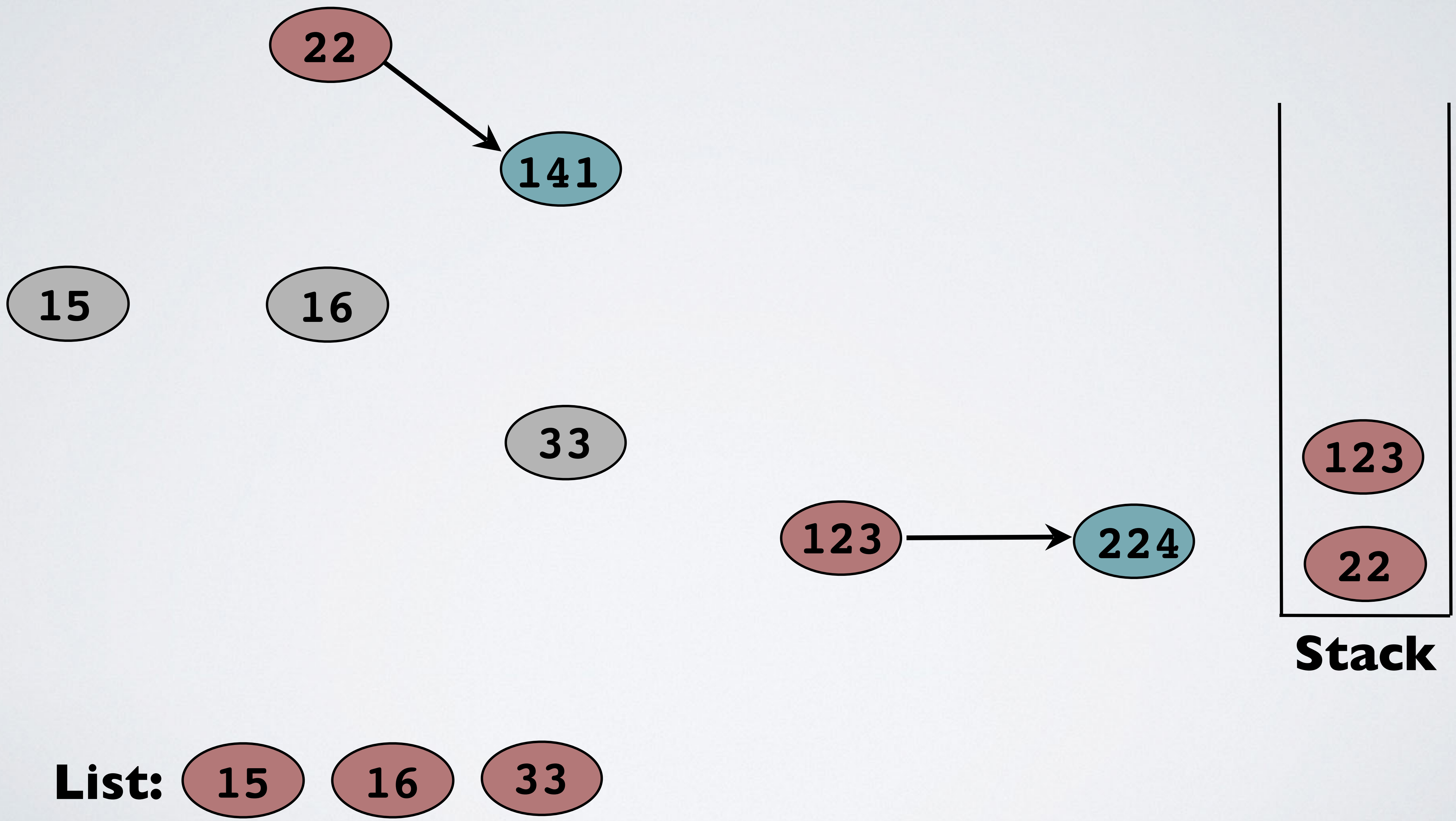
Pop from the stack & repeat!



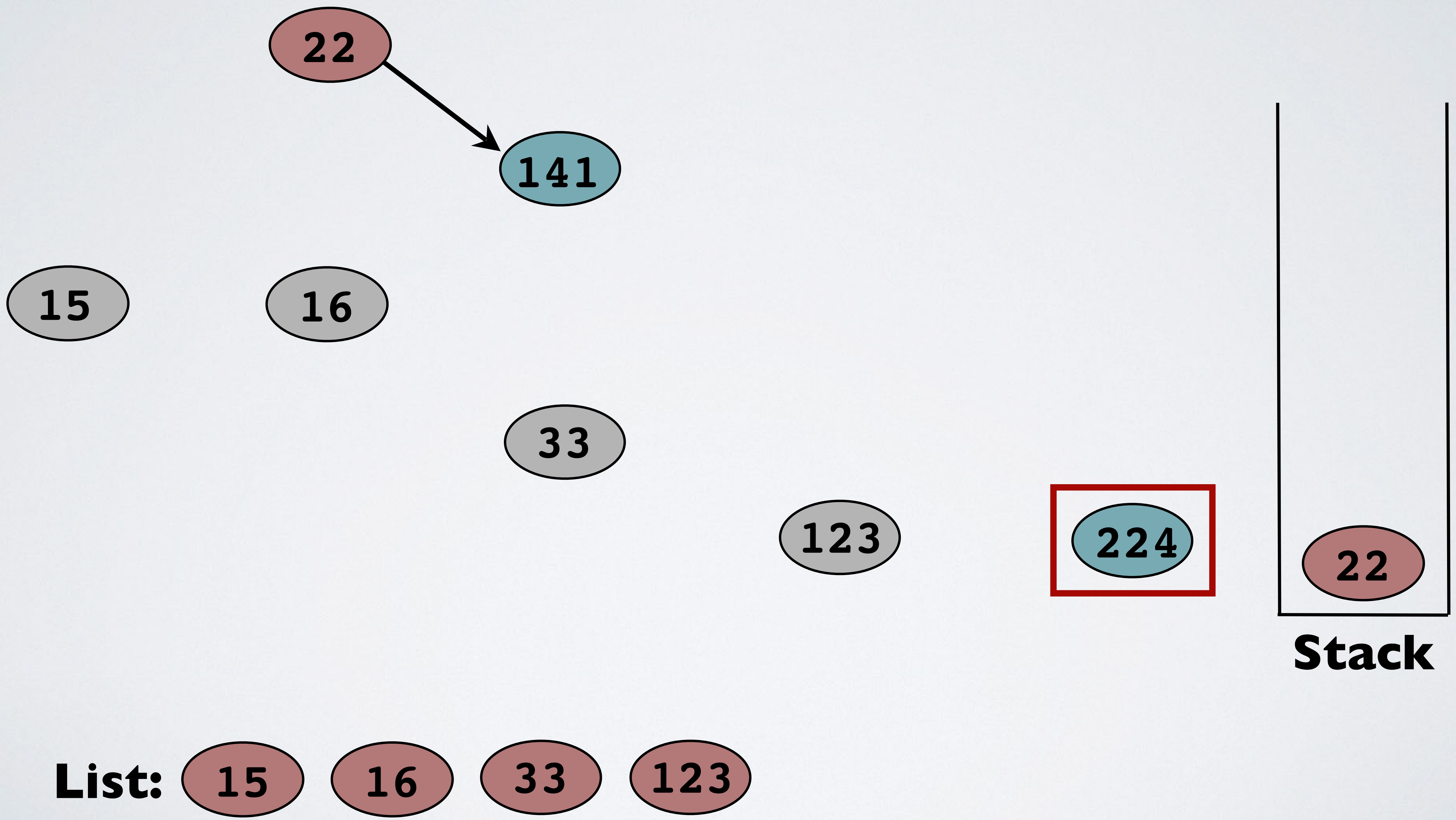
Topological Sort—Simulation



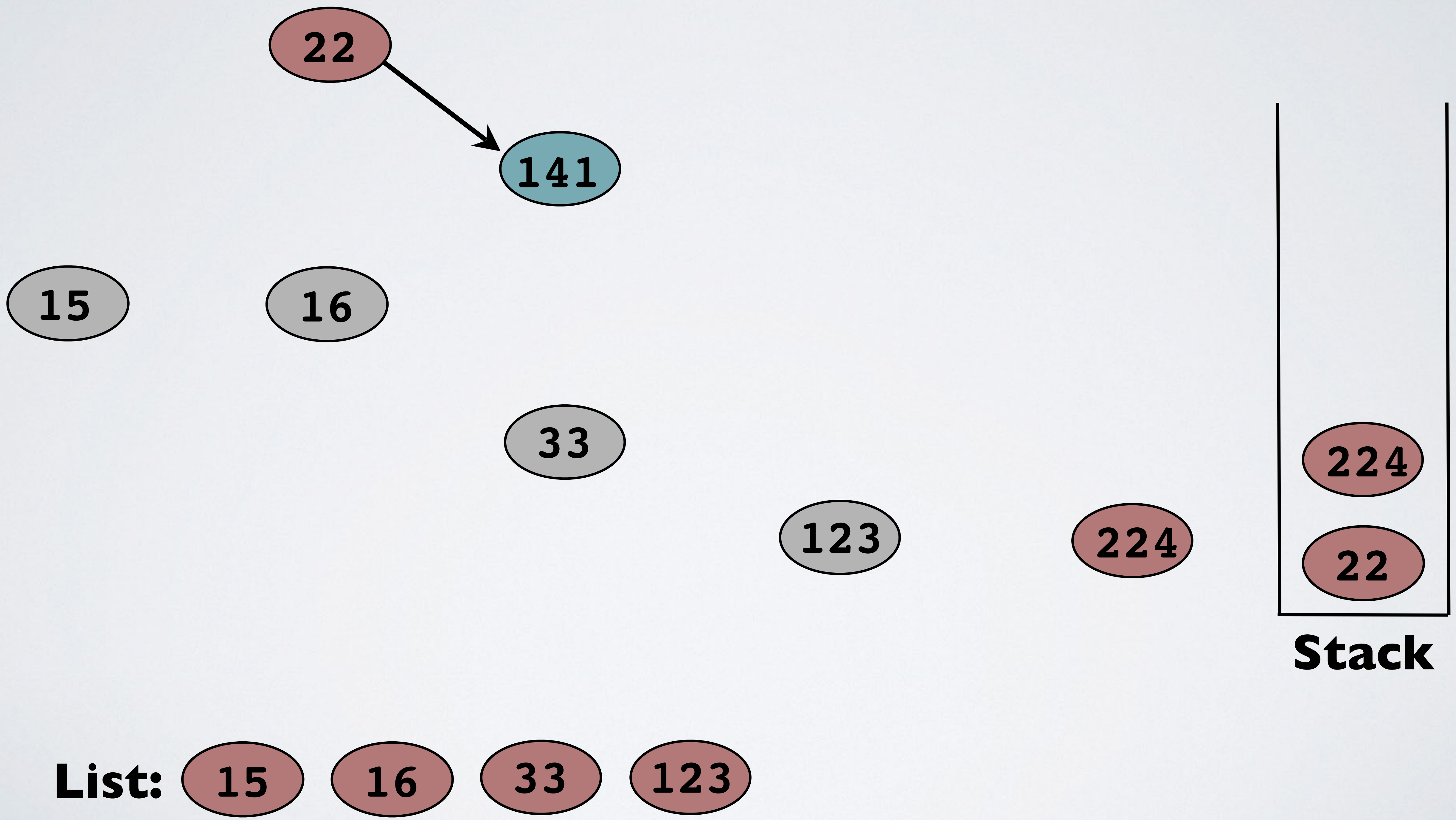
Topological Sort—Simulation



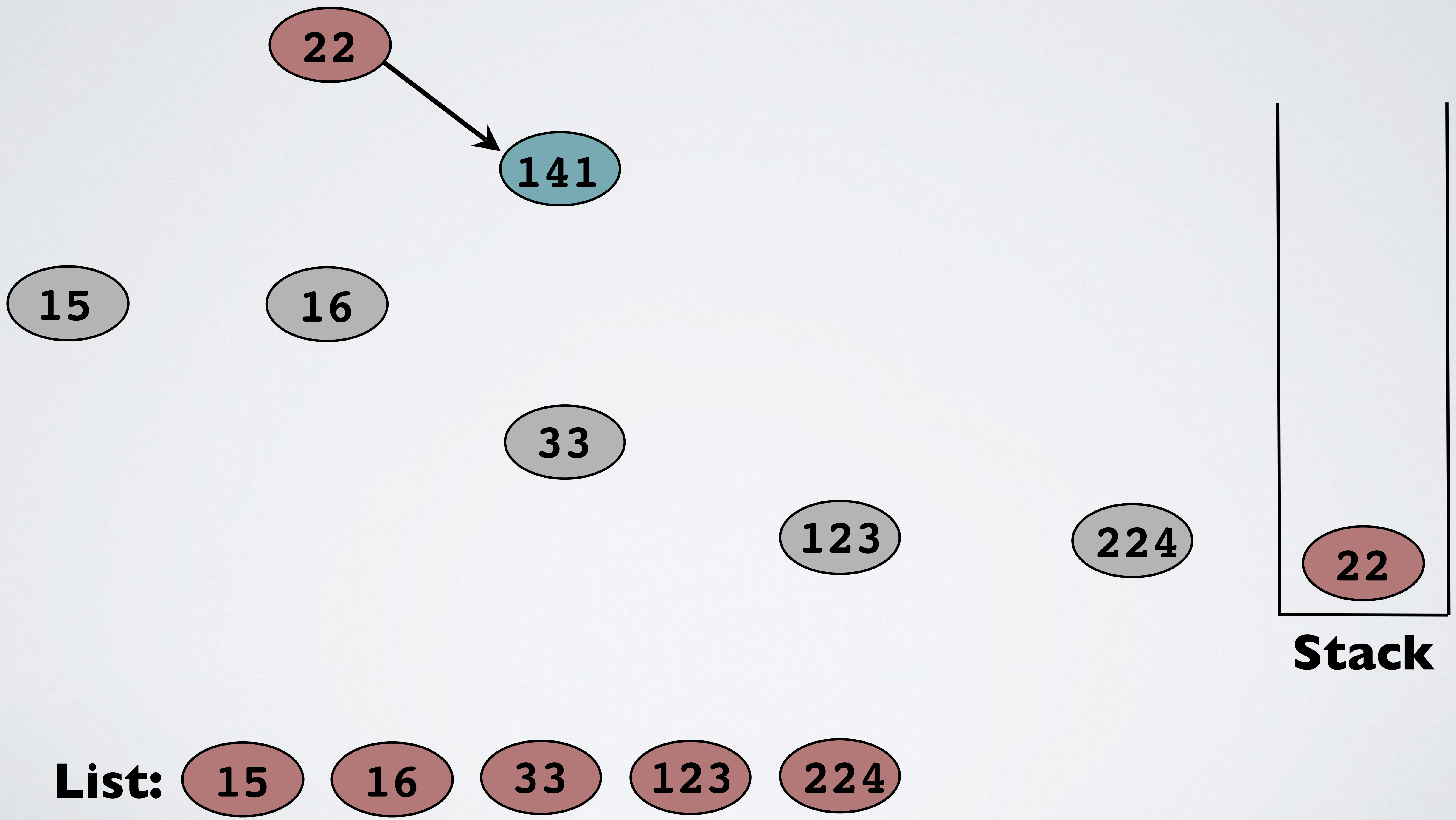
Topological Sort—Simulation



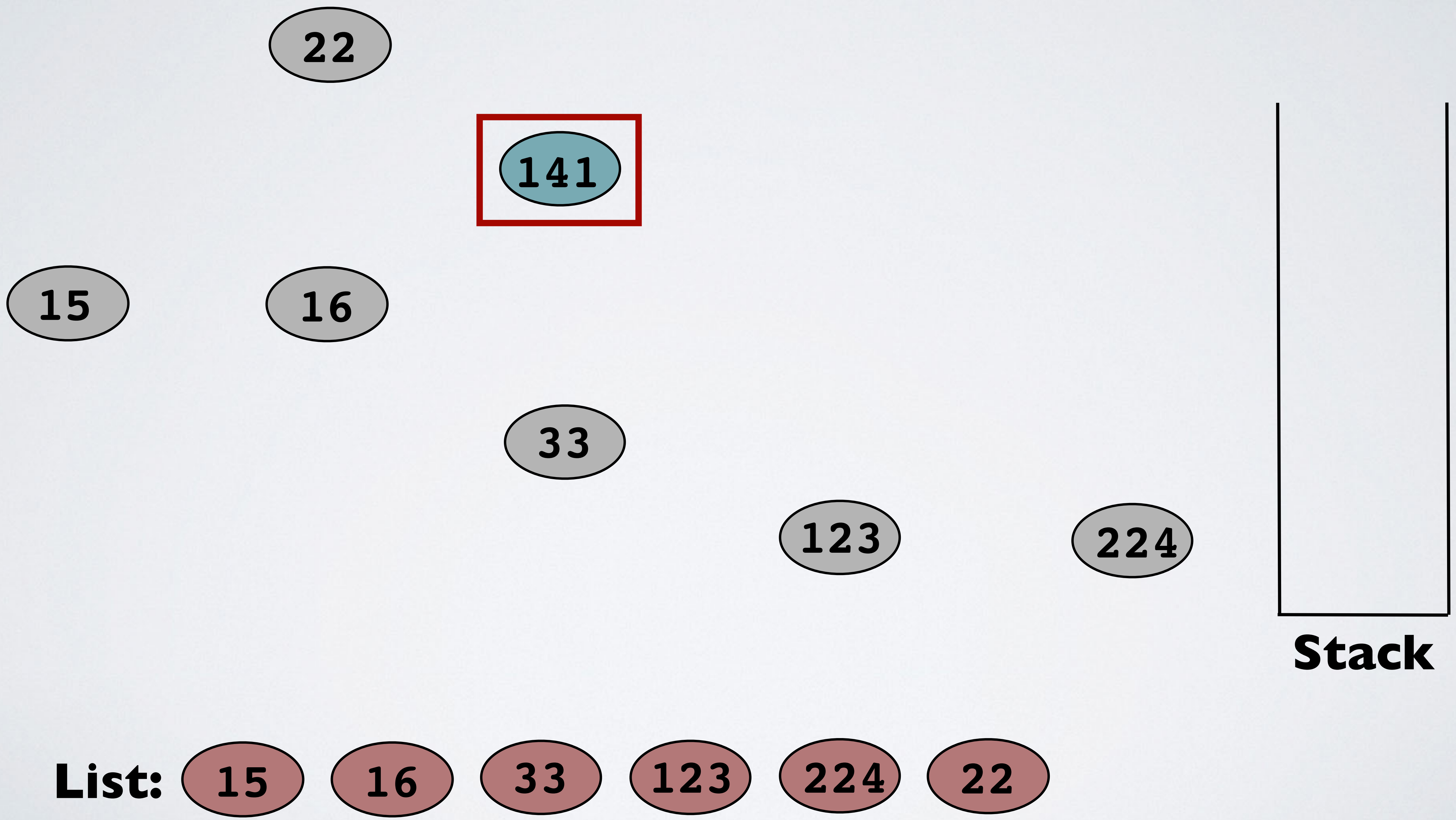
Topological Sort—Simulation



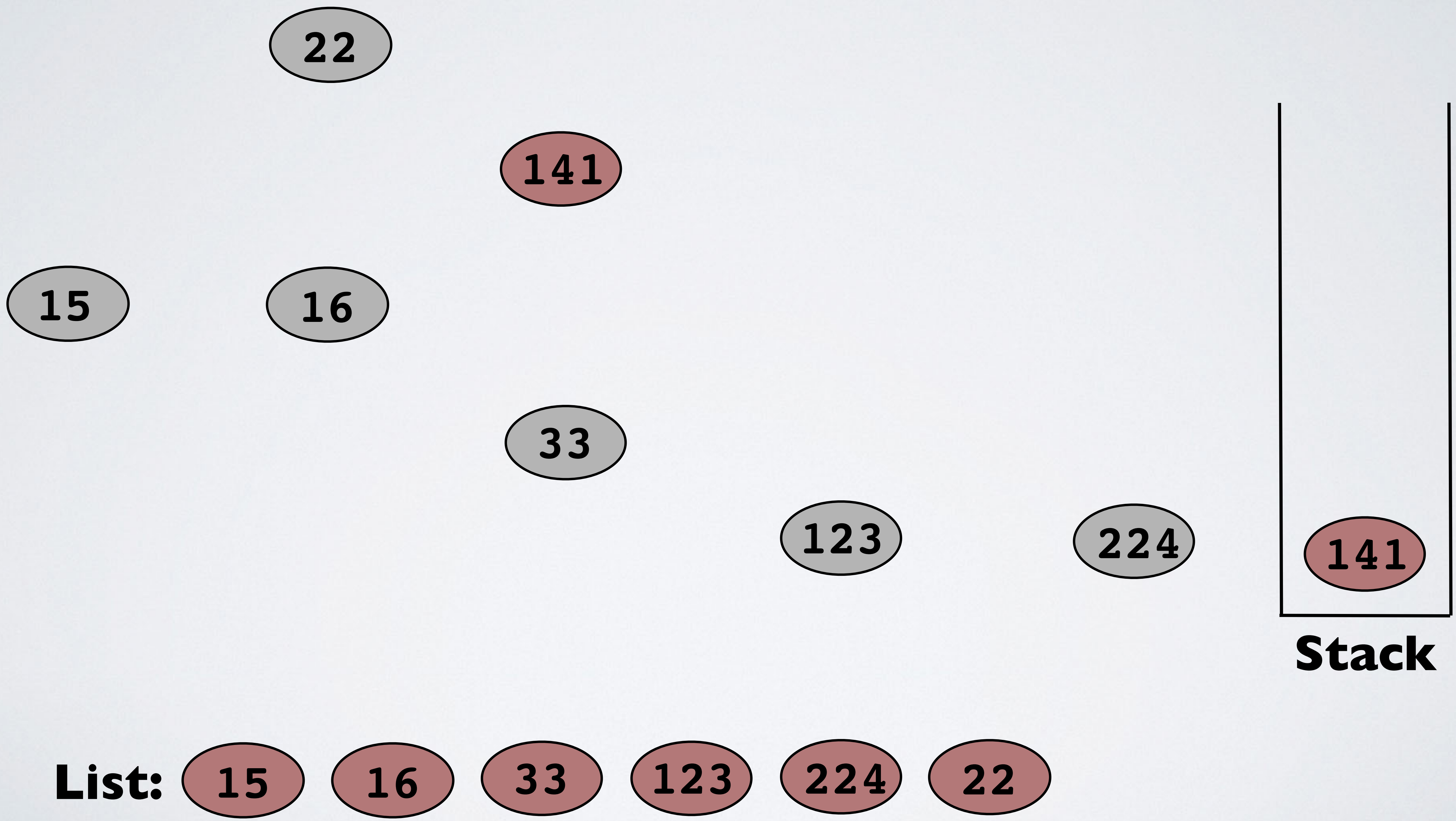
Topological Sort—Simulation



Topological Sort—Simulation

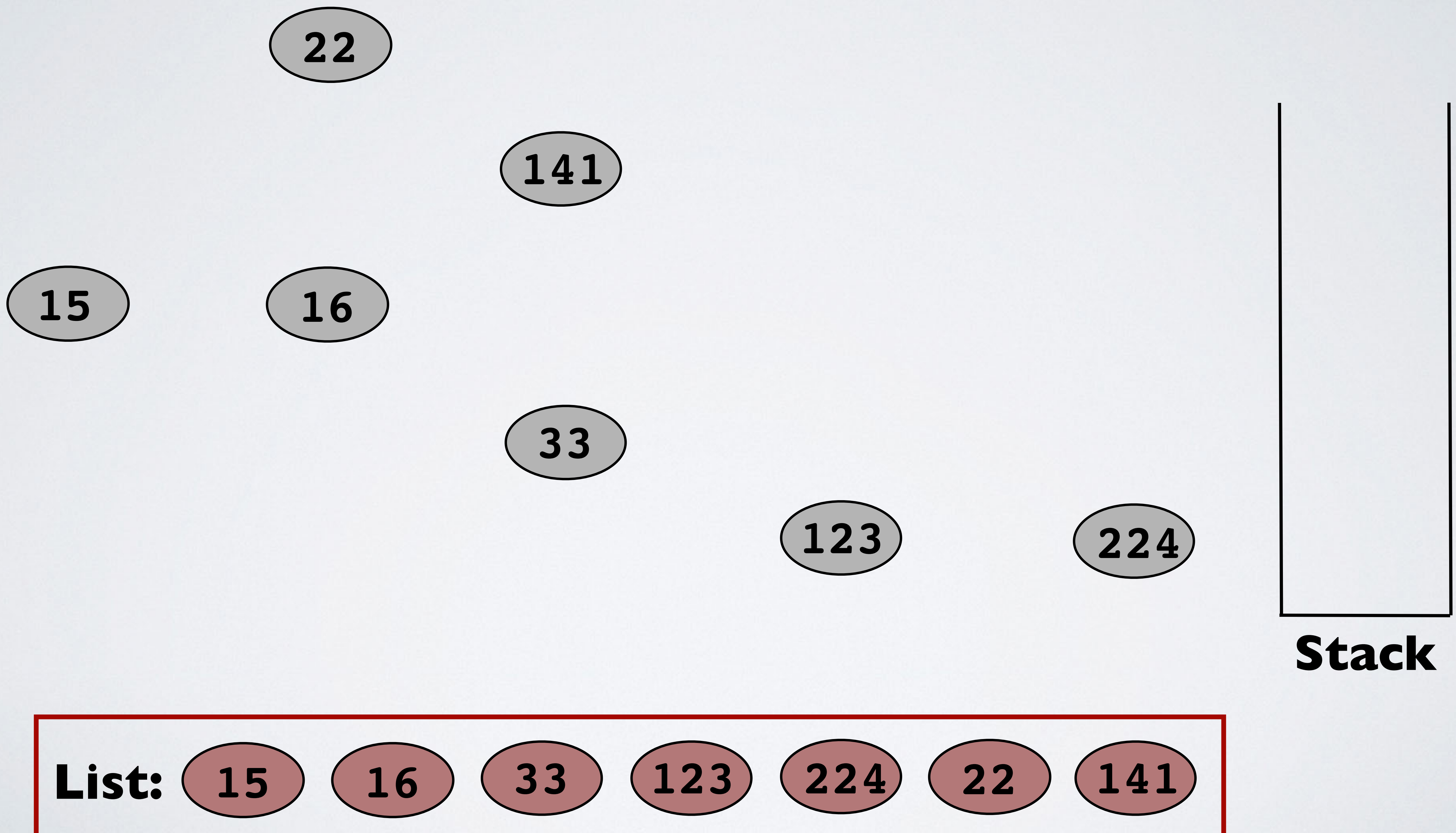


Topological Sort—Simulation



Topological Sort—Simulation

We're done!



Topological Sort Pseudo-code

```
function top_sort(graph g):
    // Input: A DAG g
    // Output: A list of vertices of g, in topological order
    s = Stack()
    l = List()
    for each vertex in g:
        if vertex is source:
            s.push(vertex)
    while s is not empty:
        v = s.pop()
        l.append(v)
        for each outgoing edge e from v:
            w = e.destination
            delete e
            if w is a source:
                s.push(w)
    return l
```

Topological Sort Runtime

```
function top_sort(graph g):
    // Input: A DAG g
    // Output: A list of vertices of g, in topological order
    s = Stack()
    l = List()
    for each vertex in g:
        if vertex is source:
            s.push(vertex)
    while s is not empty:
        v = s.pop()
        l.append(v)
        for each outgoing edge e from v:
            w = e.destination
            delete e
            if w is a source:
                s.push(w)
    return l
```

Looping through every vertex to find sources is O(|V|)

Topological Sort Runtime

```
function top_sort(graph g):
    // Input: A DAG g
    // Output: A list of vertices of g, in topological order
    s = Stack()
    l = List()
    for each vertex in g:
        if vertex is source:
            s.push(vertex)
    while s is not empty:
        v = s.pop()
        l.append(v)
        for each outgoing edge e from v:
            w = e.destination
            delete e
            if w is a source:
                s.push(w)
    return l
```

Looping through every vertex to find sources is `O(|V|)`

Stack will hold each vertex once

At each iteration we only visit outgoing edges from popped vertex. So every edge visited once.

Total runtime: `O(|V|+|E|)`

Topological Sort Variations

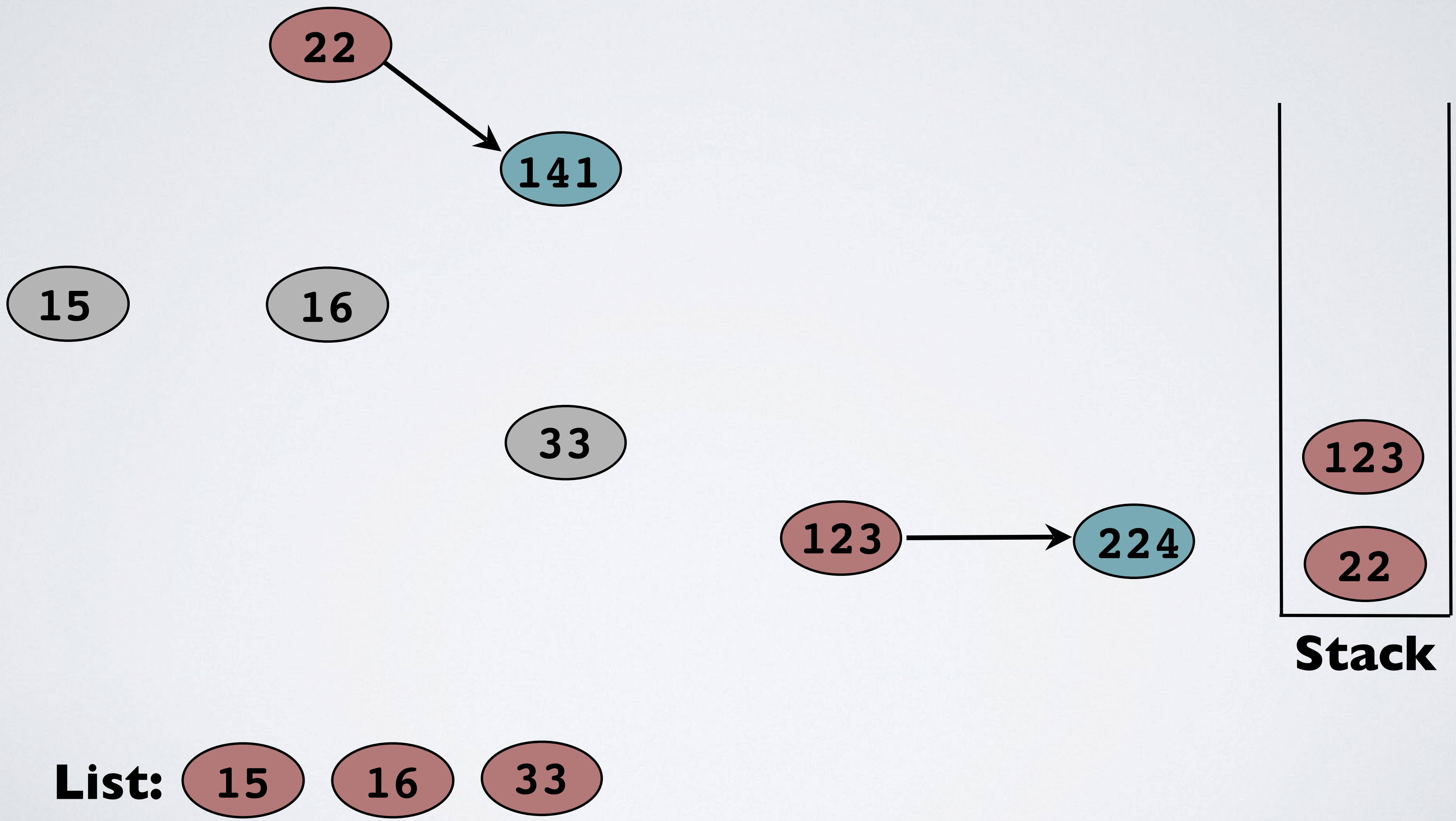
- What if we're not allowed to modify original DAG?
  - How do we delete edges?
  - Use decorations!
- Start by decorating each vertex with its in-degree
  - Instead of deleting edge
  - decrement in-degree of destination vertex by `1`
  - then push vertex on stack when in-degree is `0`!

Topological Sort Pseudo-code

```
function top_sort(graph g):
    // Input: A DAG g
    // Output: A list of vertices of g, in topological order
    s = Stack()
    l = List()
    for each vertex in g:
        if vertex is source:
            s.push(vertex)
    while s is not empty:
        v = s.pop()
        l.append(v)
        for each outgoing edge e from v:
            w = e.destination
            delete e
            if w is a source:
                s.push(w)
    return l
```

What would happen if we used a different data structure?

Topological Sort—Simulation

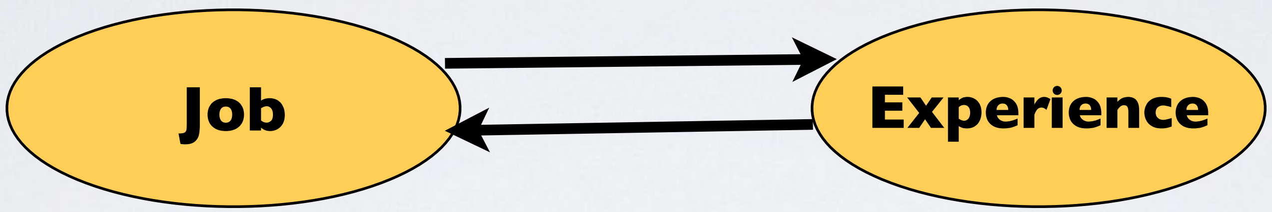


Topological Sort Variations

- Do we need to use a stack?
  - No! Any data structure like a list or queue would work
  - All we're doing is keeping track of sources
- Different structures might yield different topological orderings
  - Why do they all work ?
  - Vertices are only added to structure when they become a source
    - i.e., when all of its "prerequisites" have been visited
  - This *invariant* is maintained throughout algorithm…
  - …and guarantees a valid topological ordering!

Top Sort: Why only on DAGs ?

- If the graph has a cycle…



- …we don't have a valid topological ordering
- We can use top sort to check if a DAG has a cycle
- Run top sort on graph
  - if there are edges left at the end but no more sources
  - then there must be a cycle