DIRECTED ACYCLIC GRAPHS AND TOPOLOGICAL SORT

CS16: Introduction to Data Structures & Algorithms

Outline

1) Directed Acyclic Graphs
2) Topological sort
    1) Run-Through
    2) Pseudocode
    3) Runtime Analysis

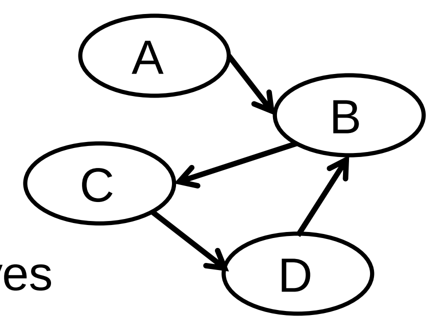
Directed Acyclic Graphs (DAGs)

- A **DAG** is a graph that has two special properties:
  - **Directed**: Each edge has an origin and a destination (visually represented by an arrow)

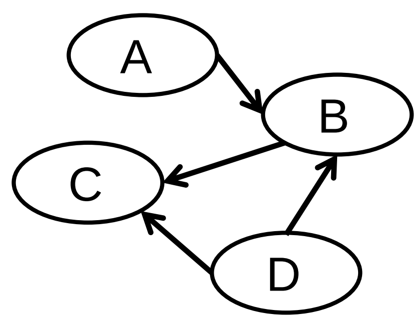
directed

not directed

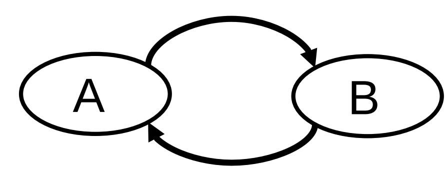
  - **Acyclic**: There is no way to start at a vertex and end up at the same vertex by traversing edges. There are no ‘cycles’
    - Which of these have cycles?



yes



no

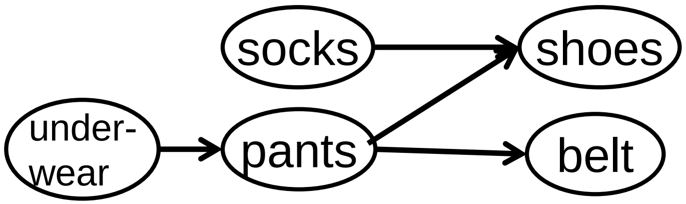


yes

Directed Acyclic Graphs (DAGs) (2)

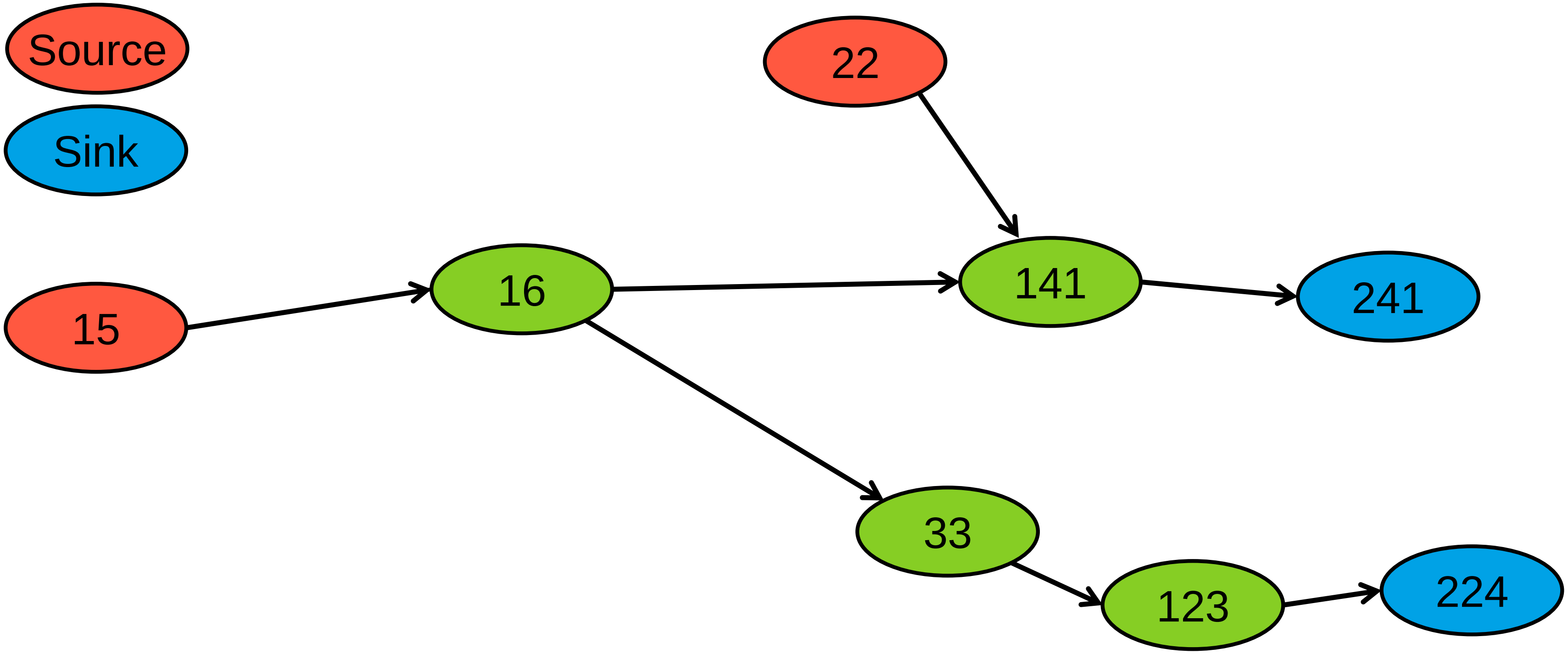
- DAGs are often used to model situations in which one element must come before another (course prerequisites, small tasks in a big project, etc.)

Ex: Getting Dressed



- **Sources** are vertices that have no incoming edges (no edges point to them)
  - “socks” and “underwear” are sources
- **Sinks** are vertices that have no outgoing edges (no edges have that vertex as their origin)
  - “shoes” and “belt”
- **In-degree** of a node is number of incoming edges
- **Out-degree** of a node is number of outgoing edges

Example DAG – Brown CS Course Prerequisites

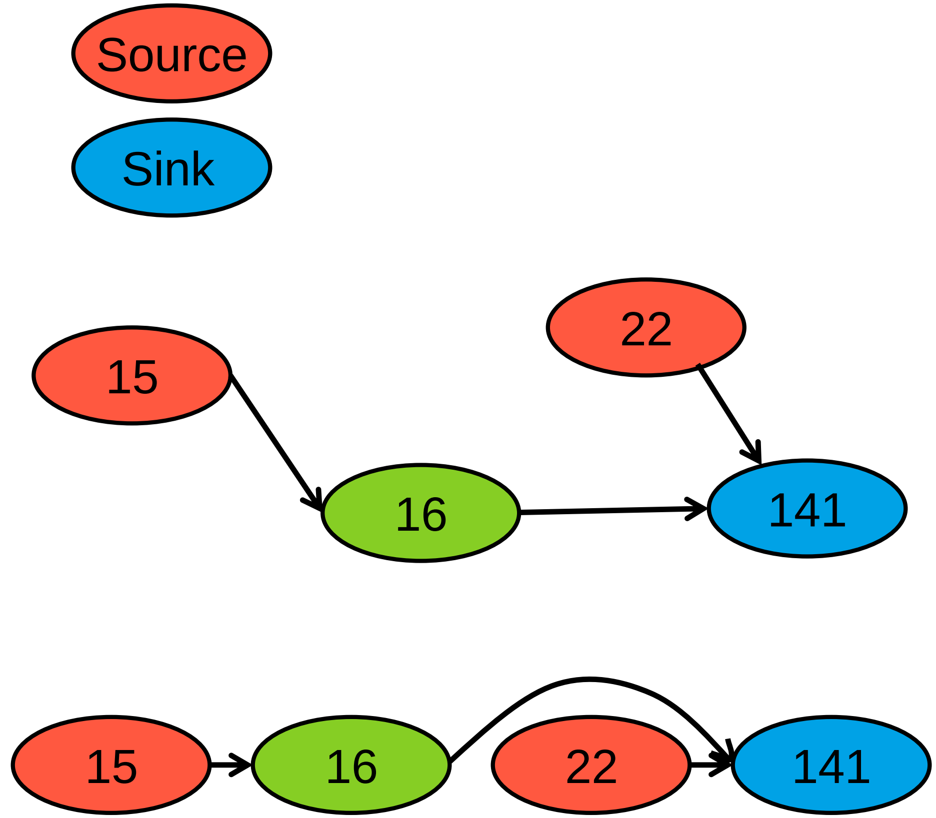


Intro to…

- *Imagine* that you are a CS concentrator trying to plan your courses for the next three years…
- How might you plan the order in which to take these courses?
- Topological sort! That’s how!

Topological Sort

- Topological ordering
  - Ordering of vertices in a DAG
  - For each vertex v, all of v's "prerequisite" vertices are before v
- Topological sort
  - Given a DAG, produce a topological ordering!
- If you lined up all vertices in topological order, all edges would point to the right
- One DAG can have multiple valid topological orderings



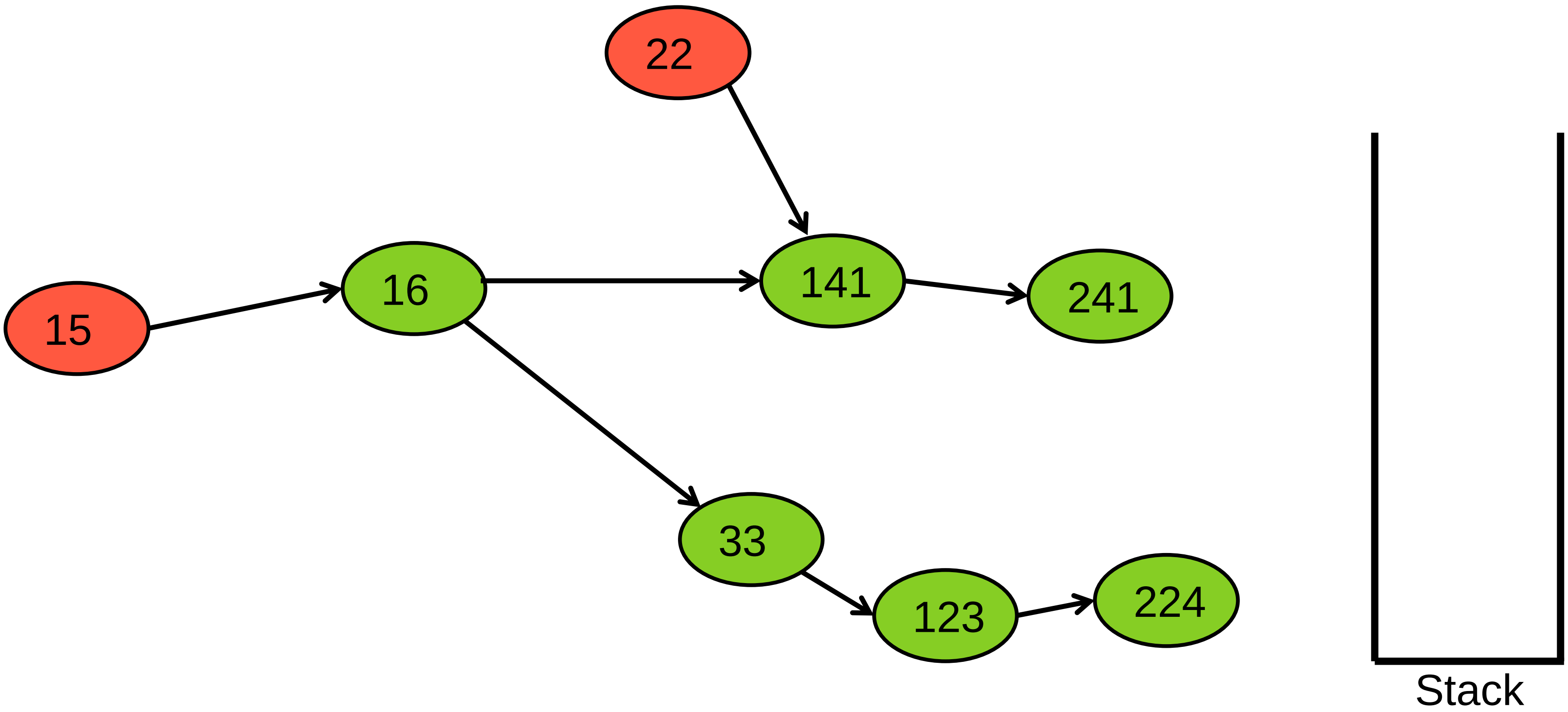
Valid Topological Orderings:
1. 15, 22, 16, 141
2. 15, 16, 22, 141
3. 22, 15, 16, 141

Top Sort: General Approach

- If a node is a source, there are no prerequisites, so we can visit it!
- Once we visit a node, we can delete all of its outgoing edges
- Deleting edges might create new sources, which we can now visit!

- Data Structures Needed:
  - DAG we're top-sorting
  - Set of all sources (represented by a stack)
  - List for our topological ordering

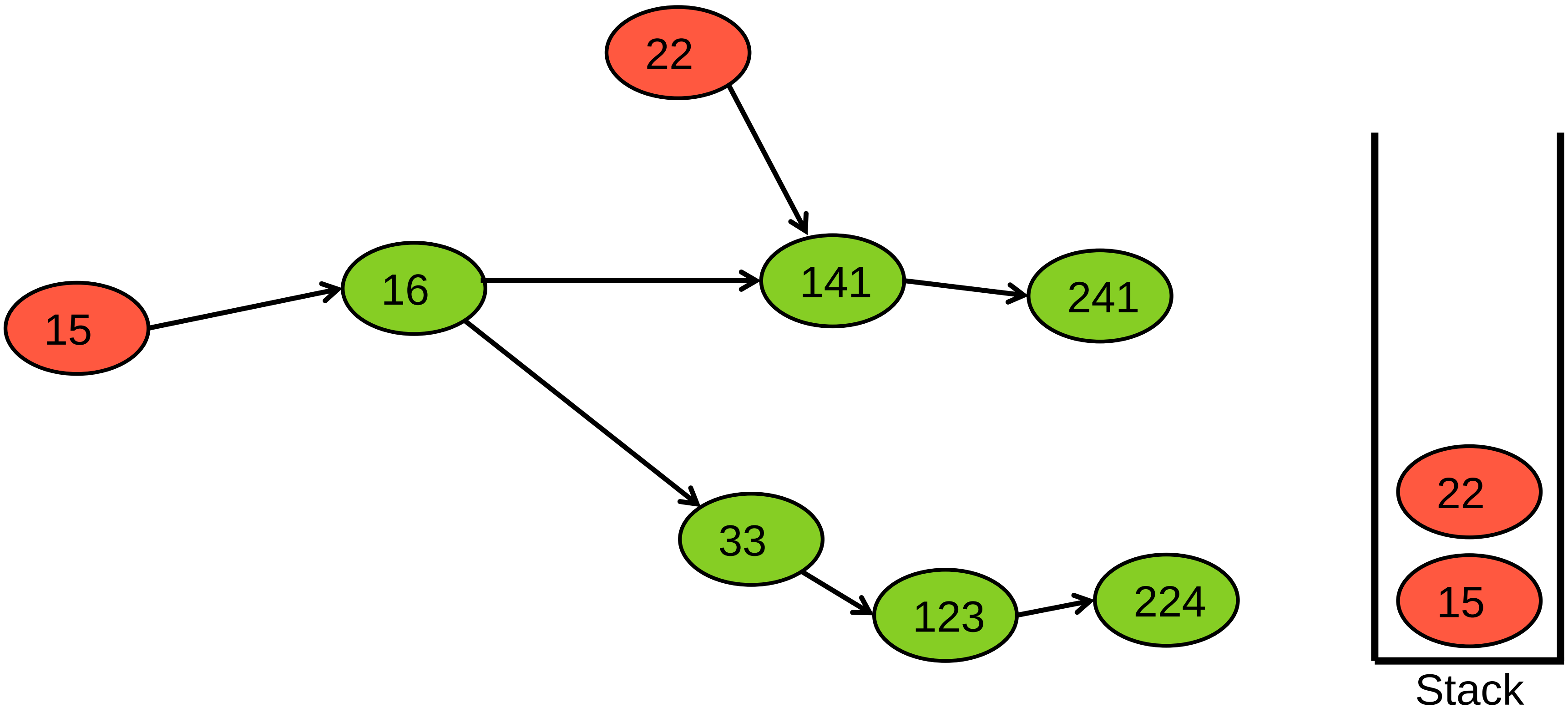
Topological Sort Run-Through



List:

Topological Sort Run-Through (2)

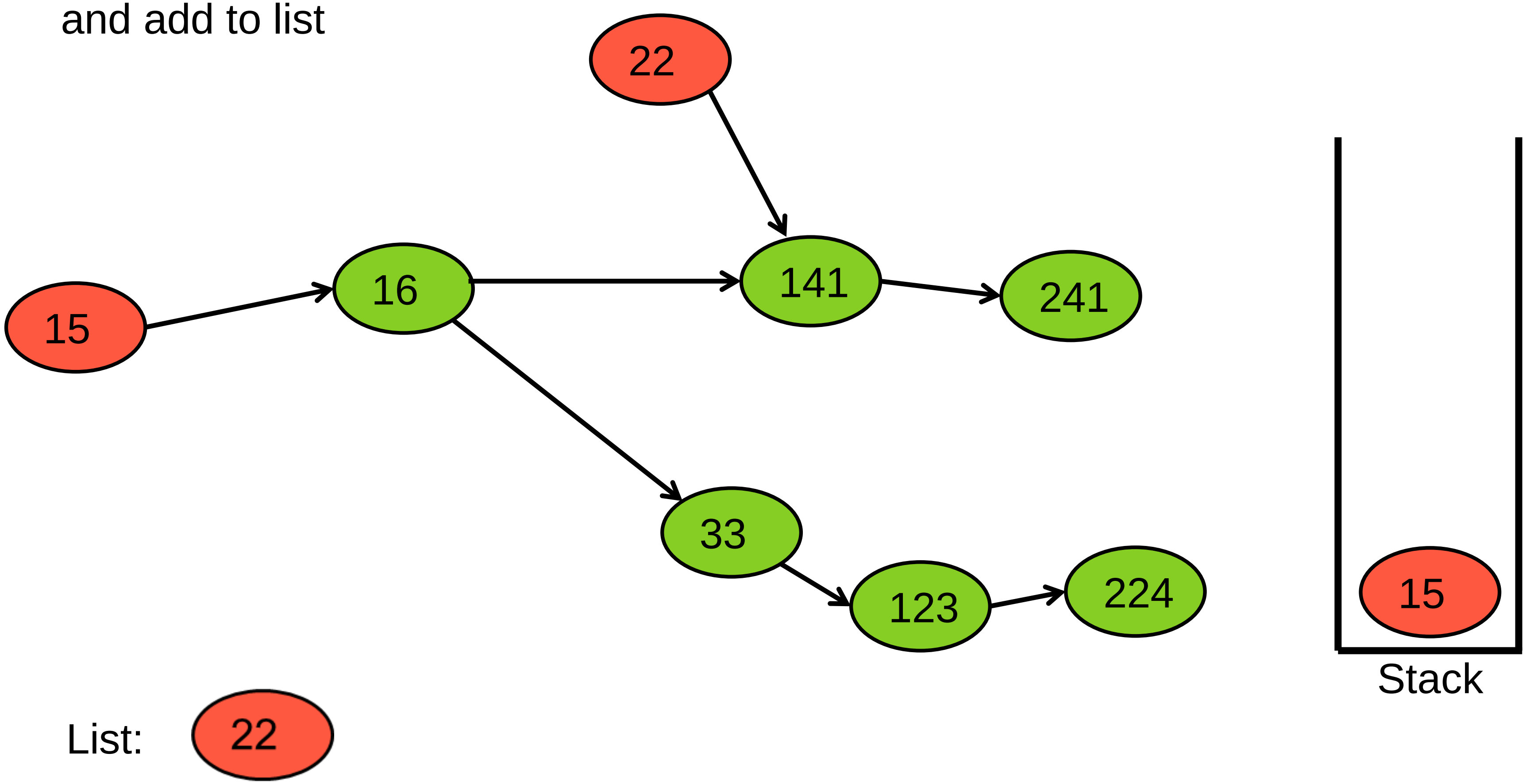
First: Populate Stack



List:

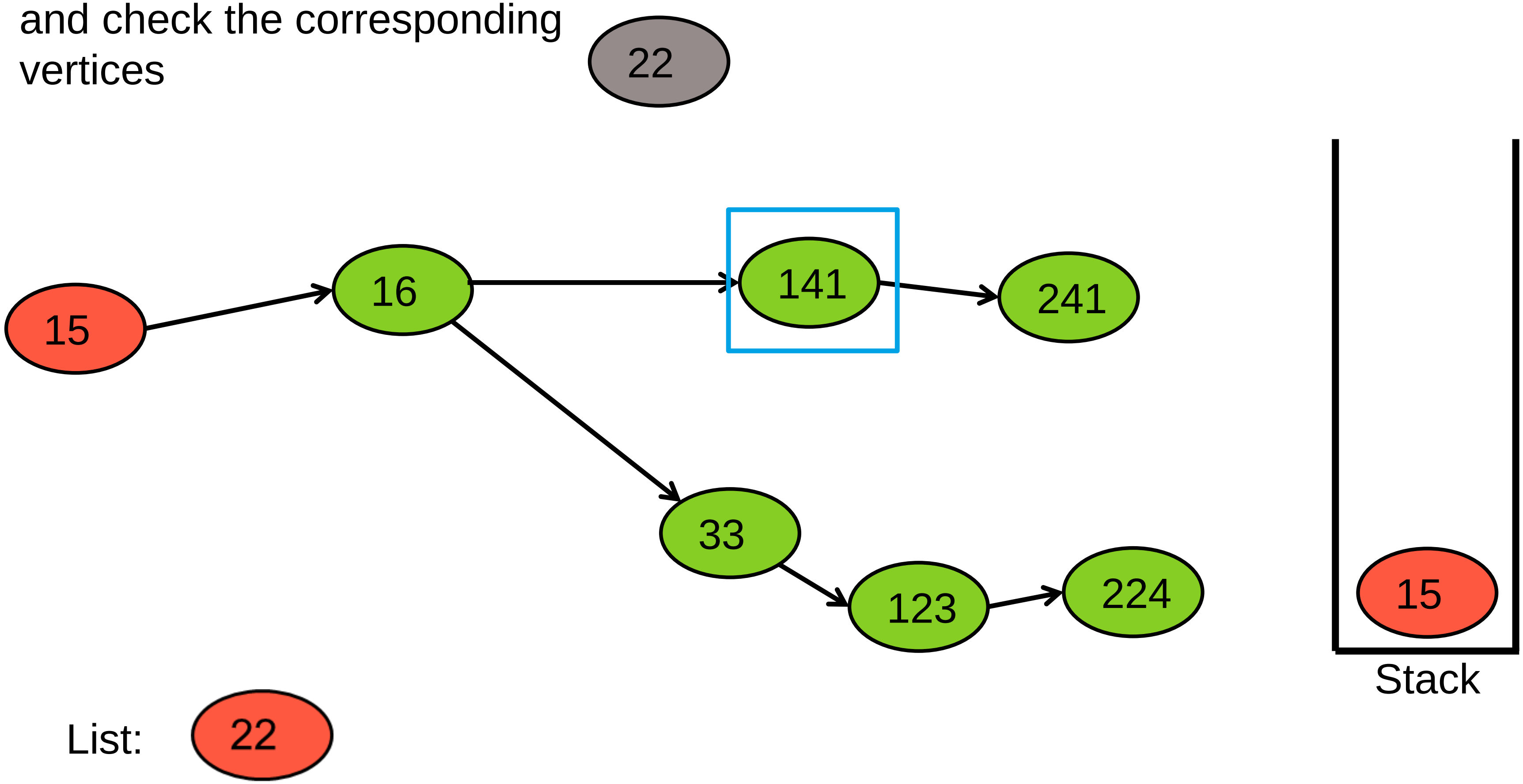
Topological Sort Run-Through (3)

Next: Pop element from stack and add to list



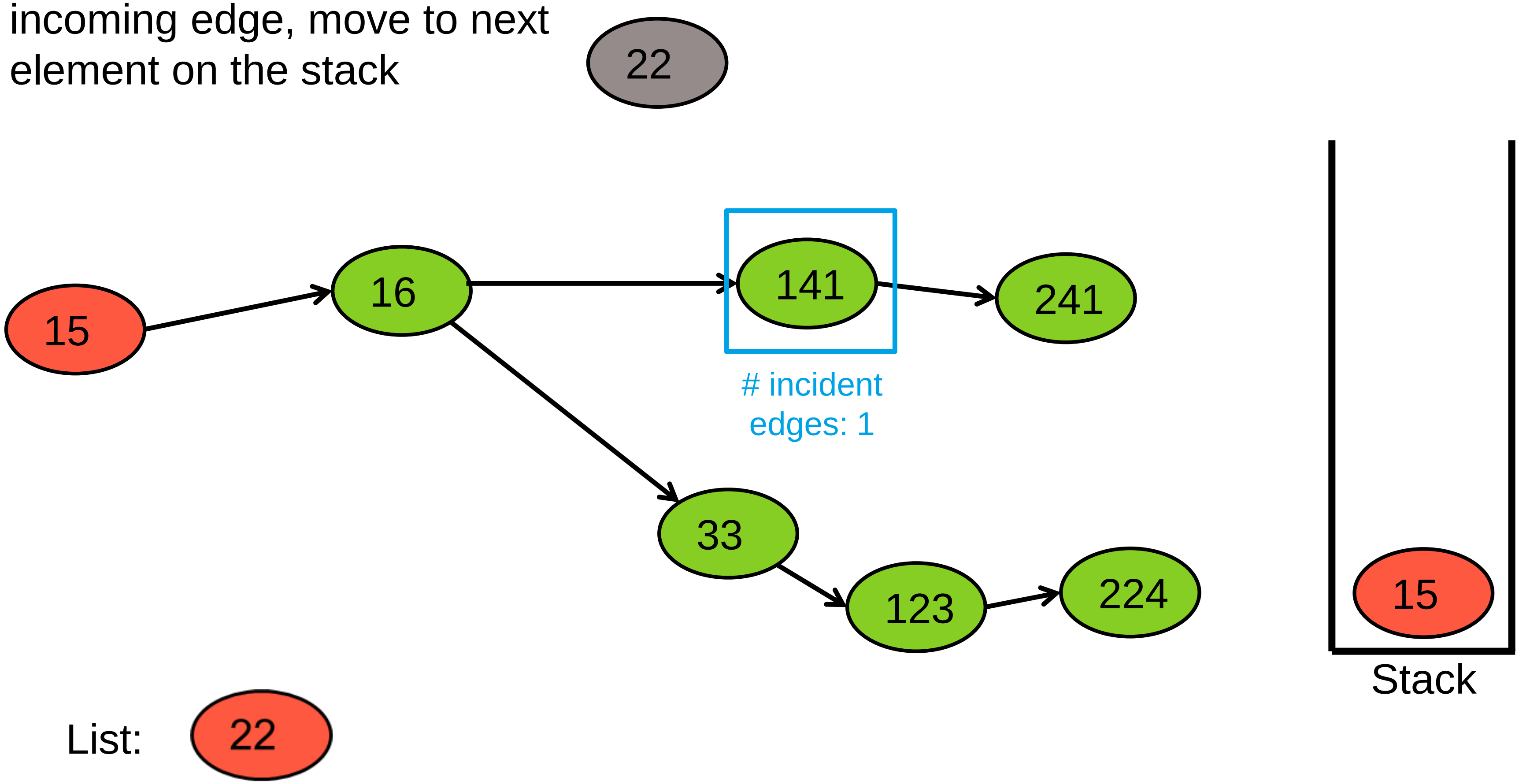
Topological Sort Run-Through (4)

Next: Remove outgoing edges and check the corresponding vertices



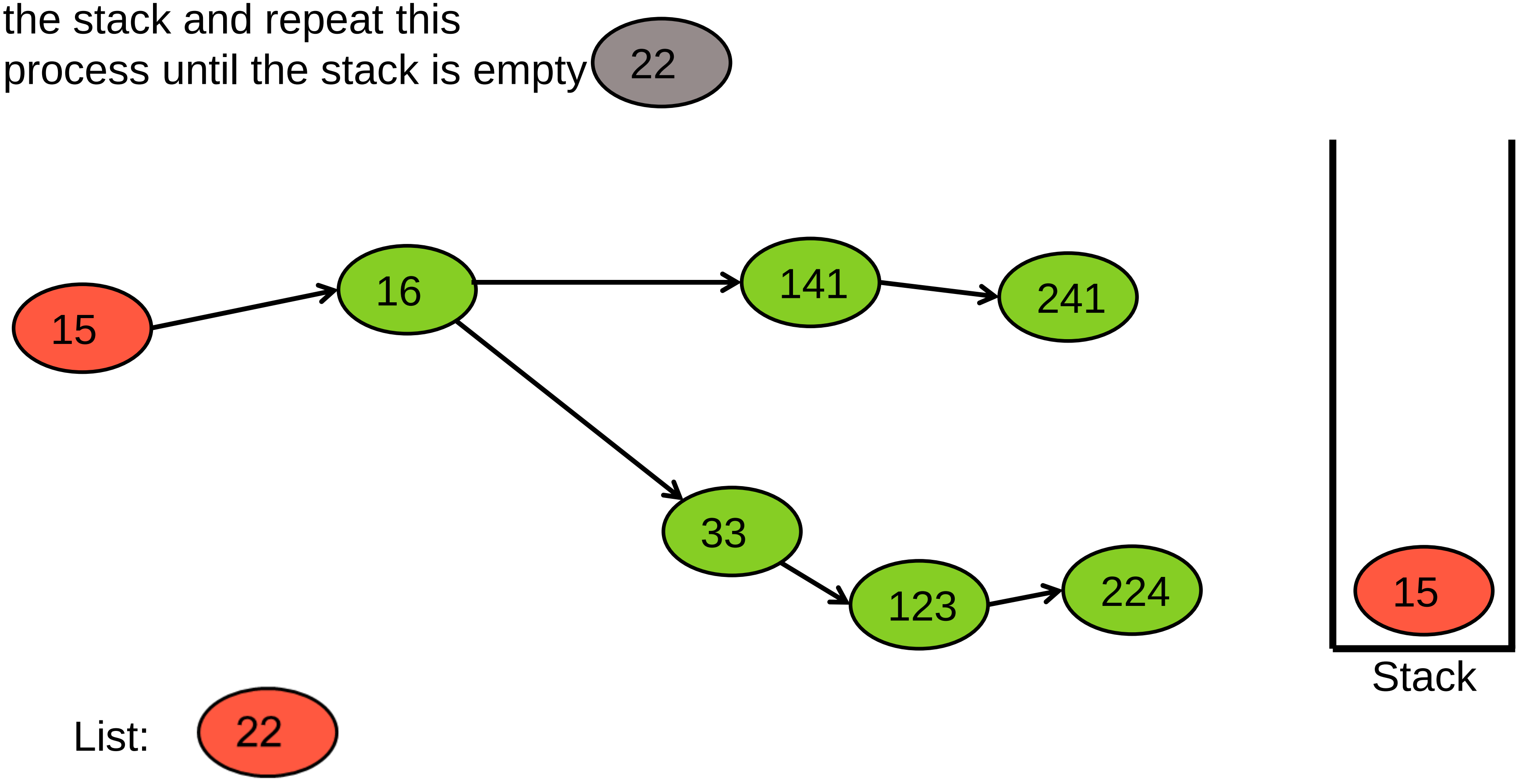
Topological Sort Run-Through (5)

Next: Since 141 still has an incoming edge, move to next element on the stack



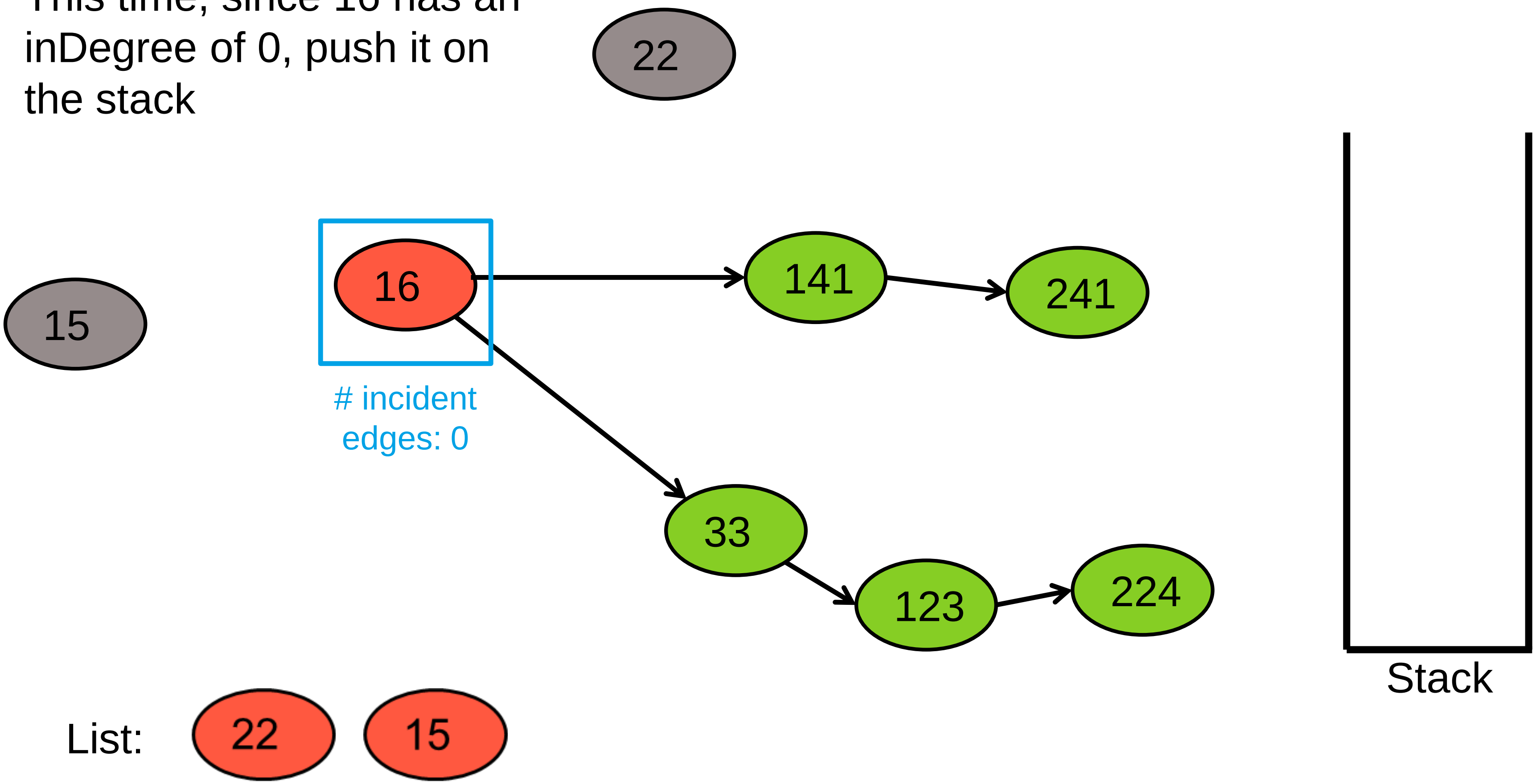
Topological Sort Run-Through (6)

Next: Pop the next element off the stack and repeat this process until the stack is empty



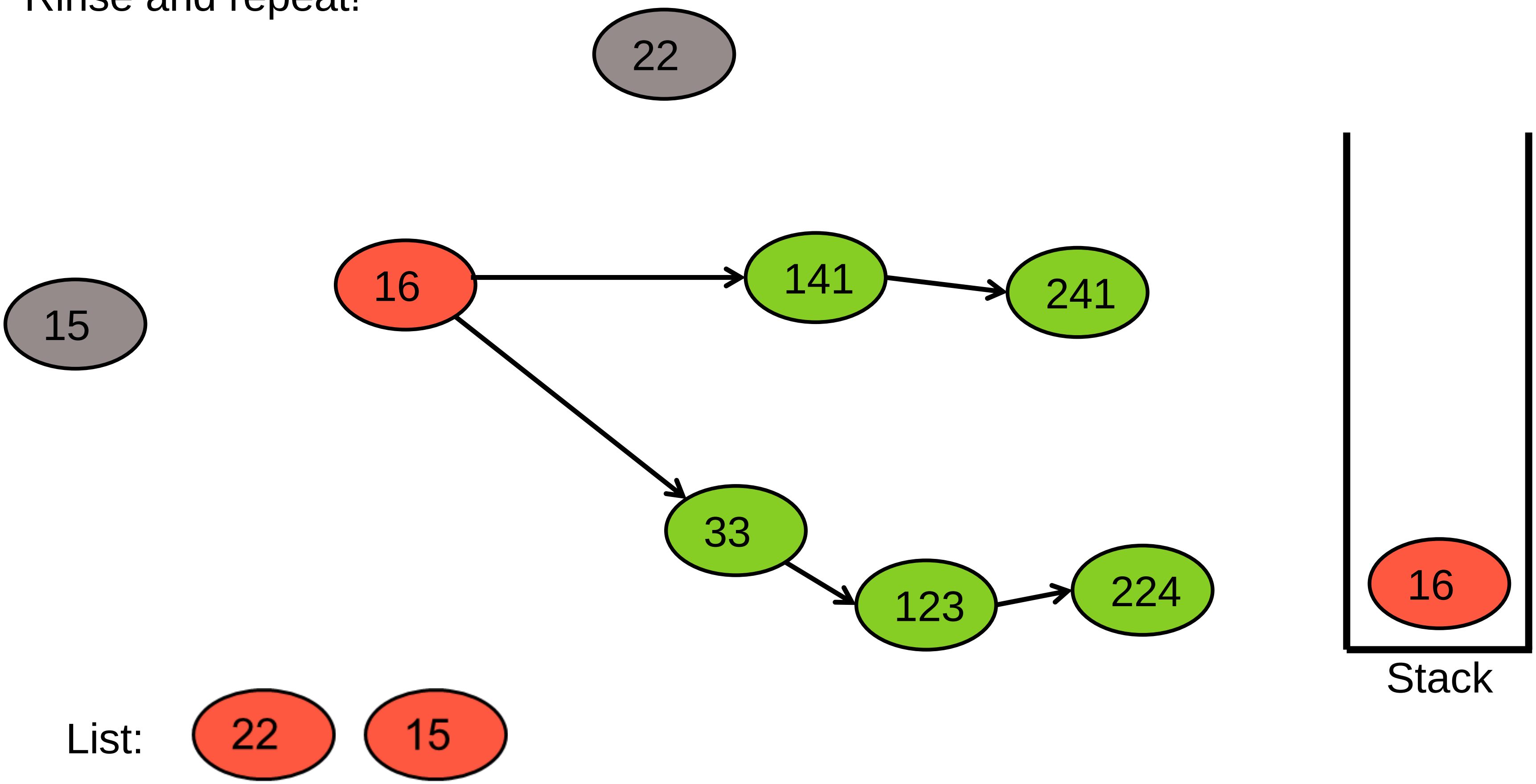
Topological Sort Run-Through (7)

This time, since 16 has an inDegree of 0, push it on the stack

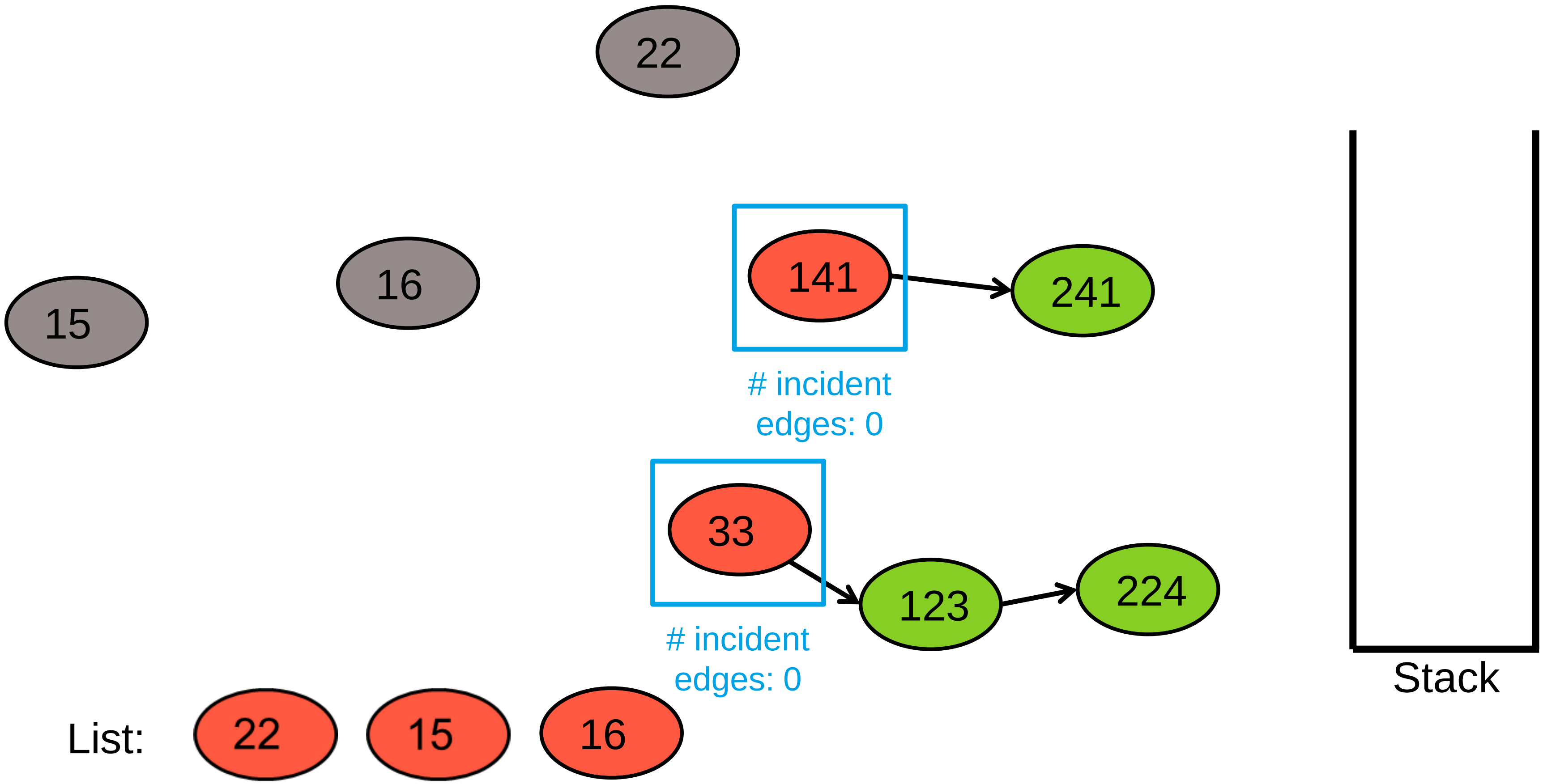


Topological Sort Run-Through (8)

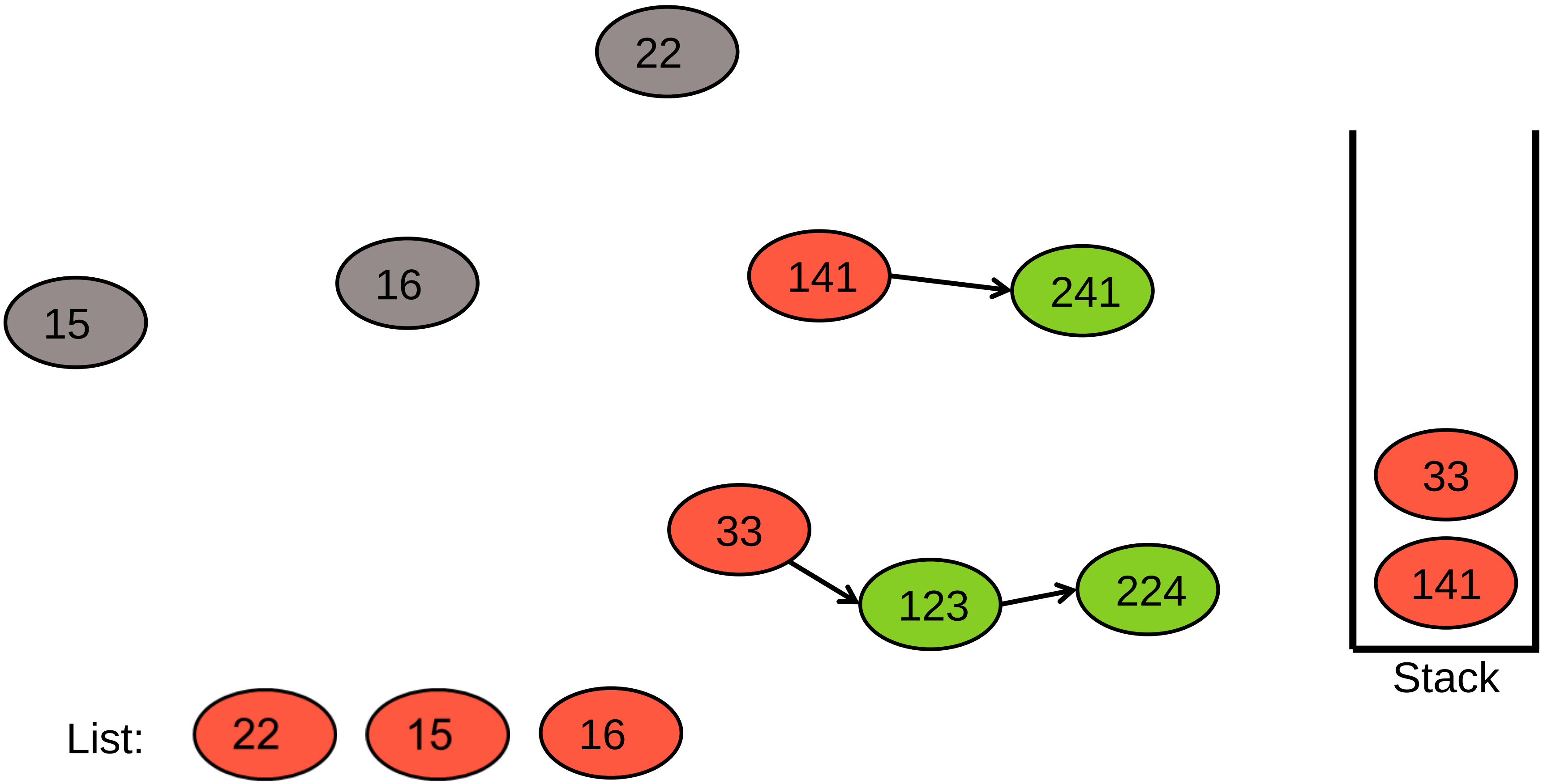
Rinse and repeat!



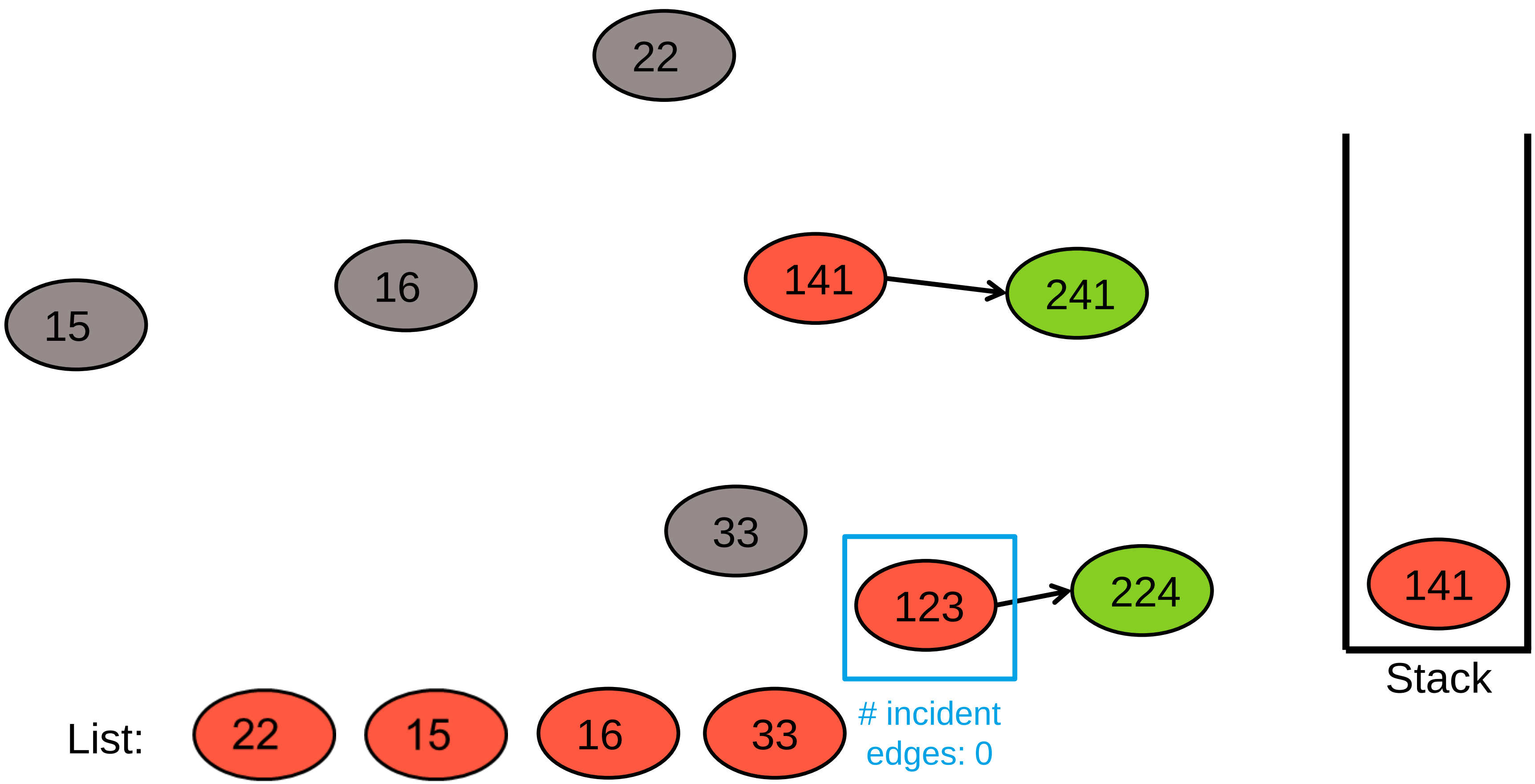
Topological Sort Run-Through (9)



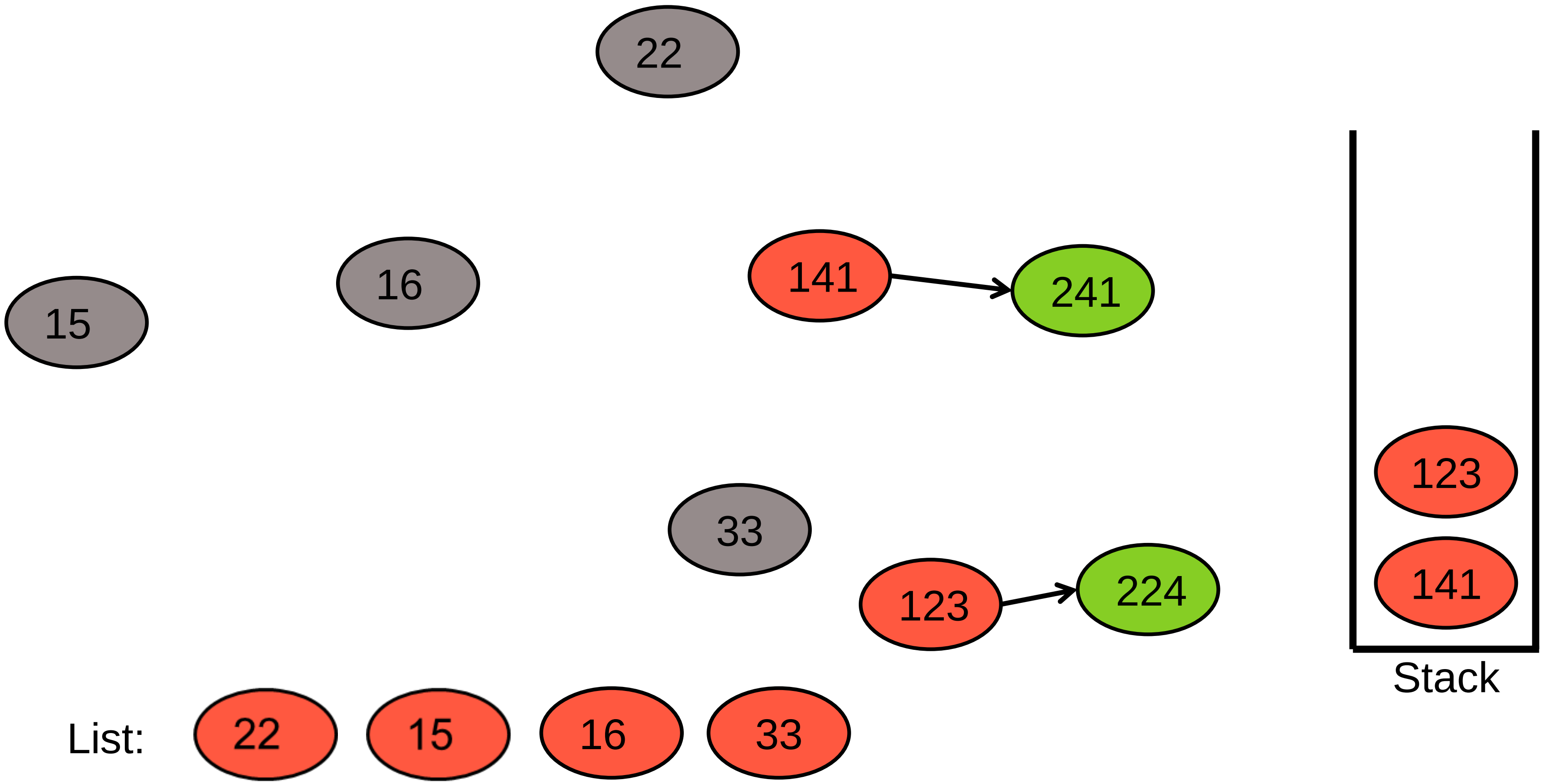
Topological Sort Run-Through (10)



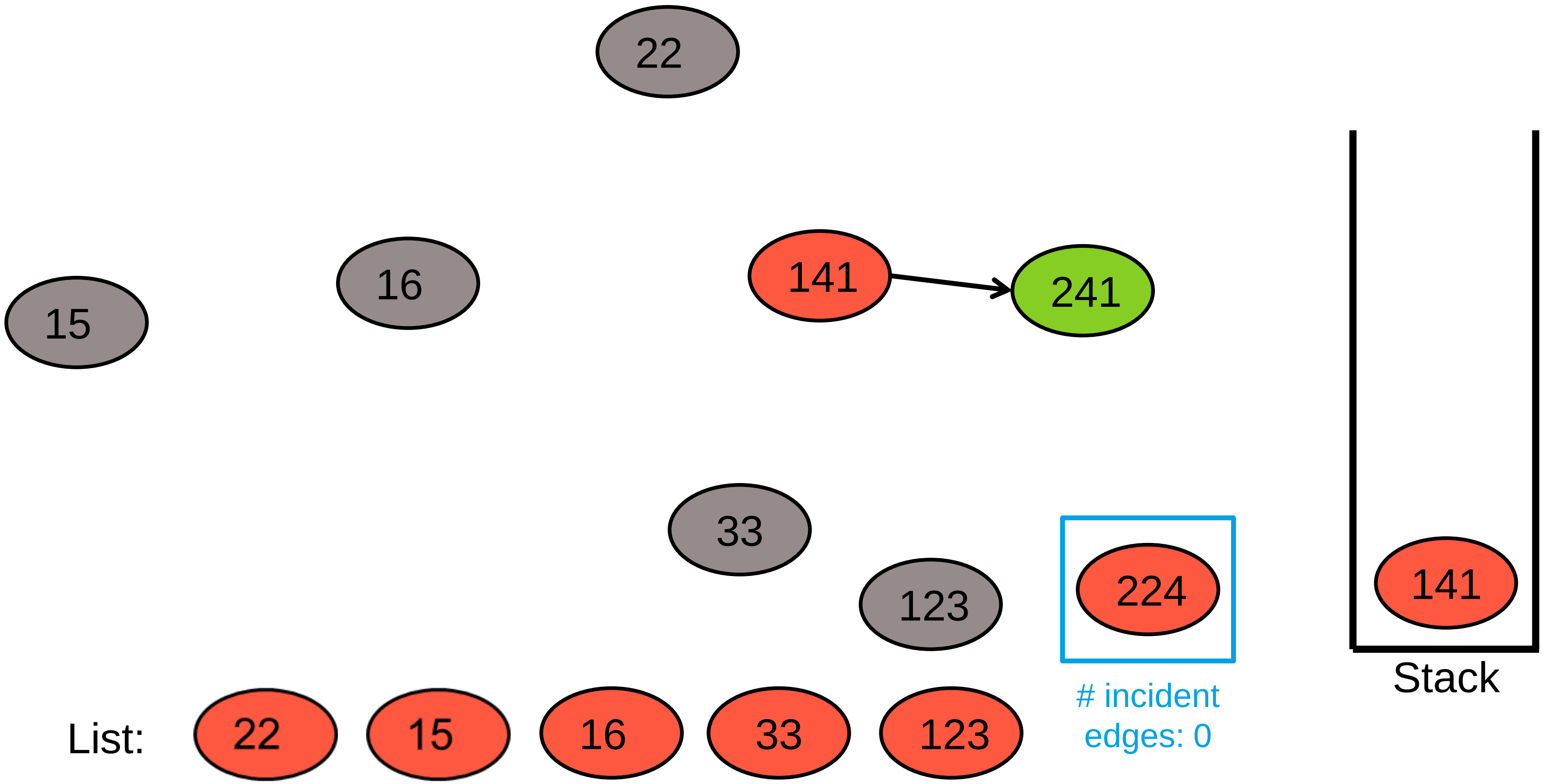
Topological Sort Run-Through (12)



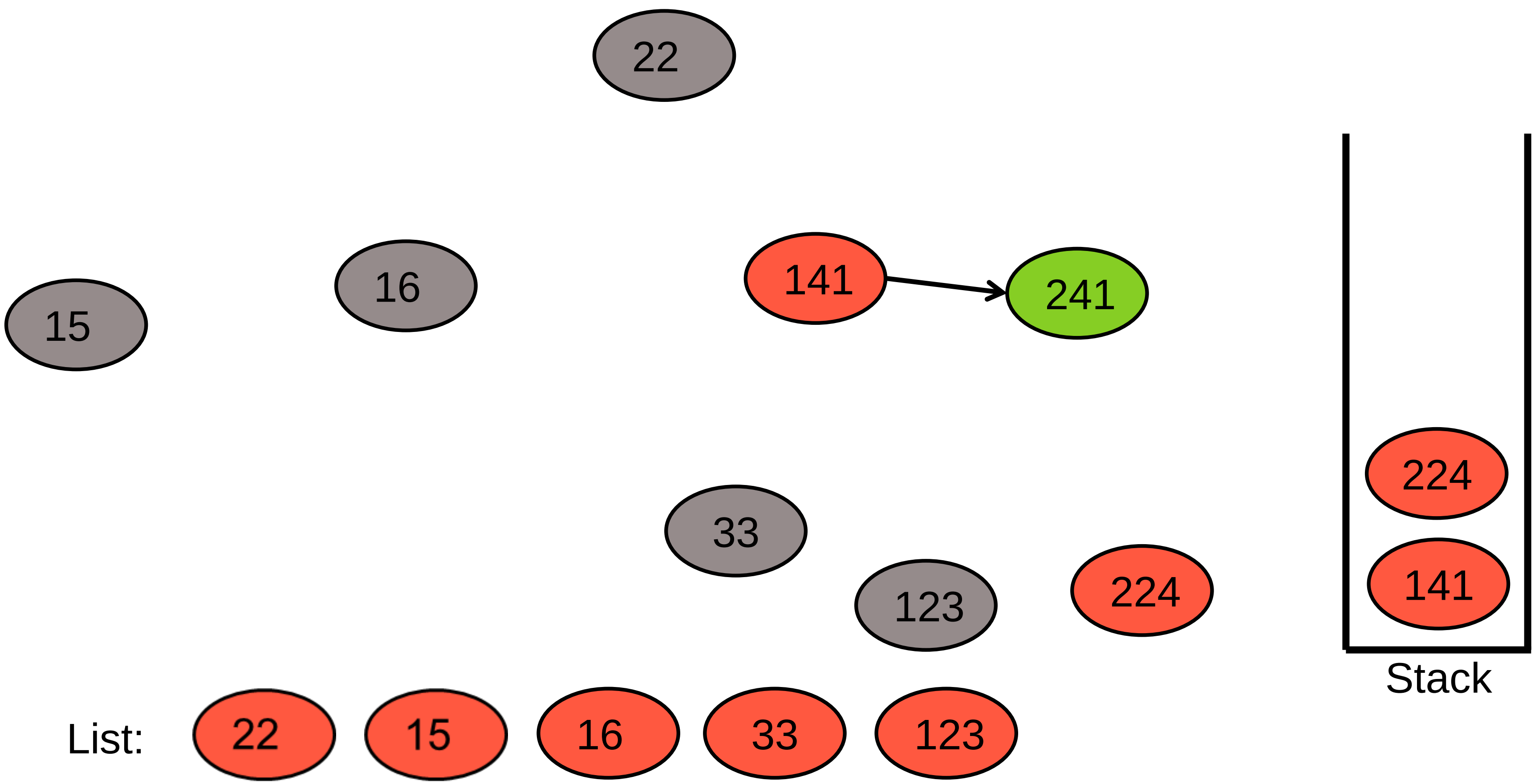
Topological Sort Run-Through (13)



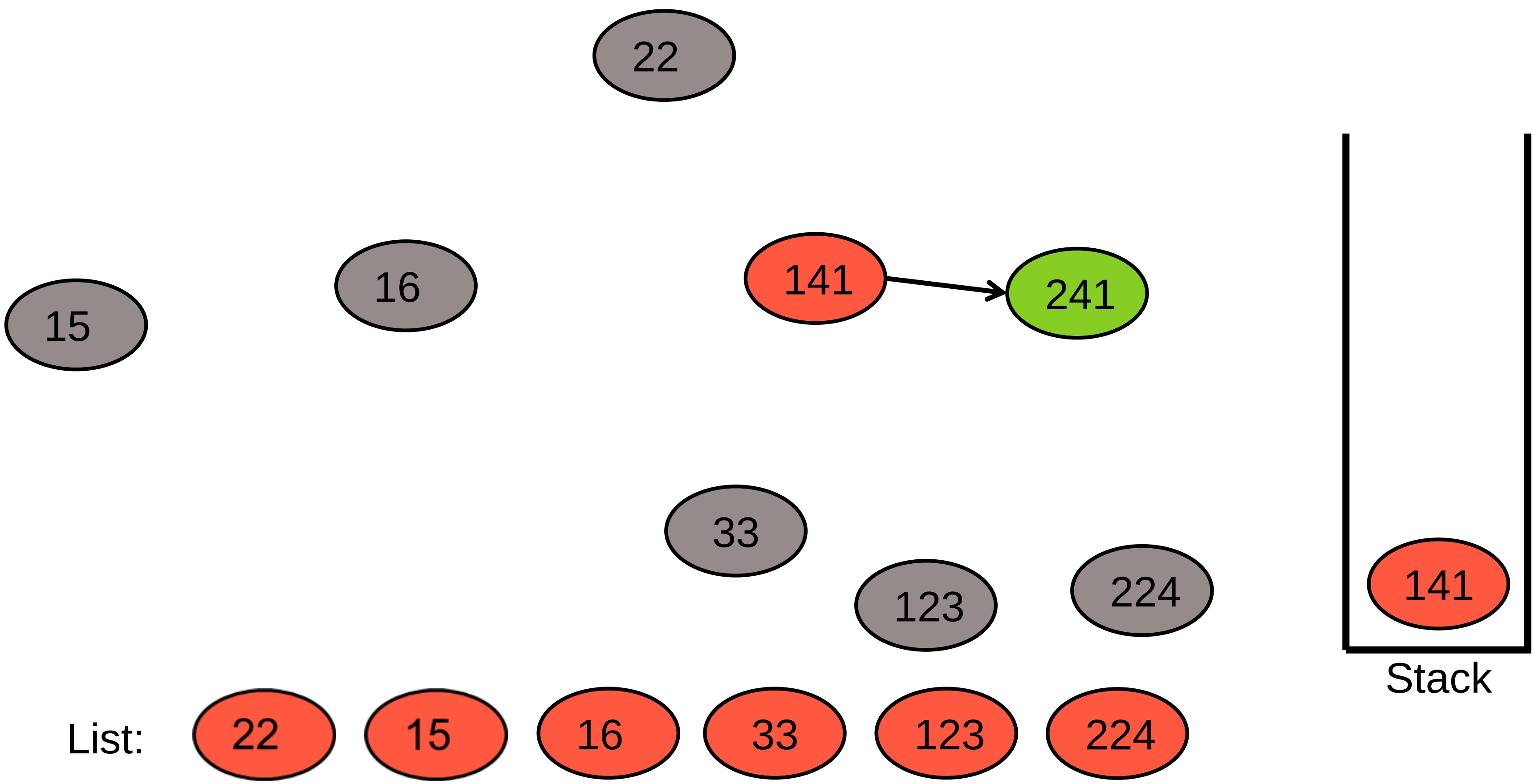
Topological Sort Run-Through (14)



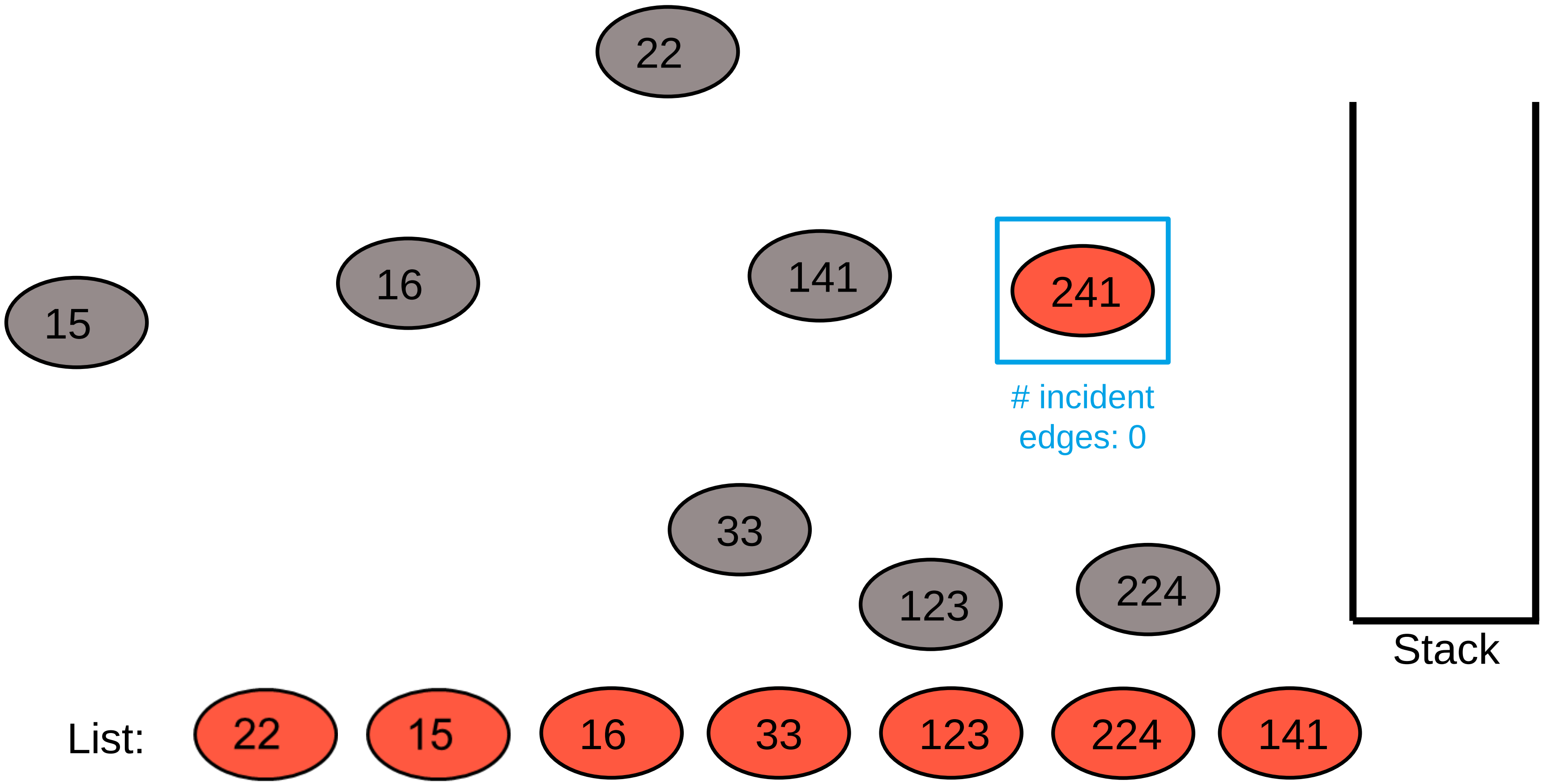
Topological Sort Run-Through (15)



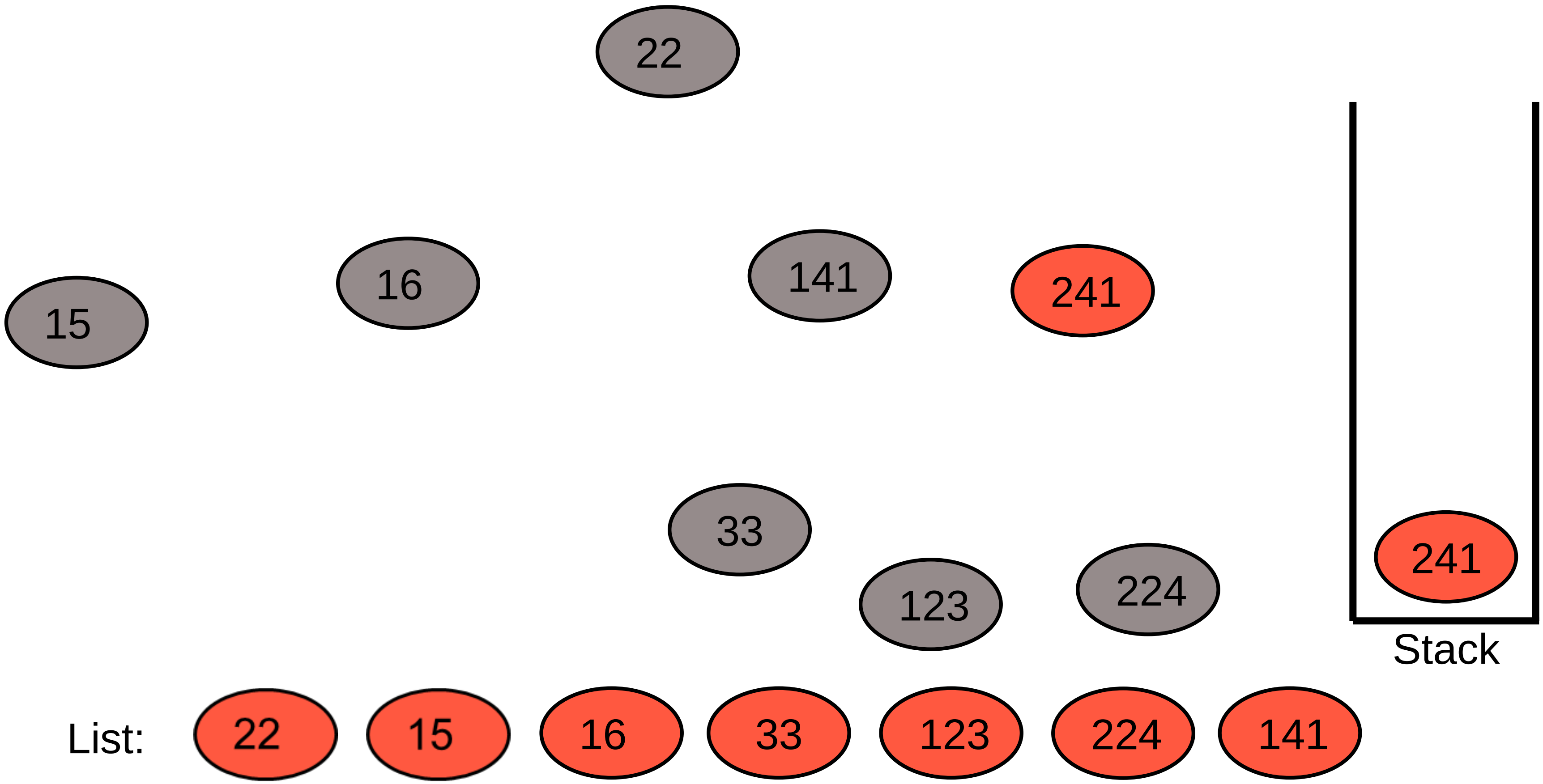
Topological Sort Run-Through (16)



Topological Sort Run-Through (17)

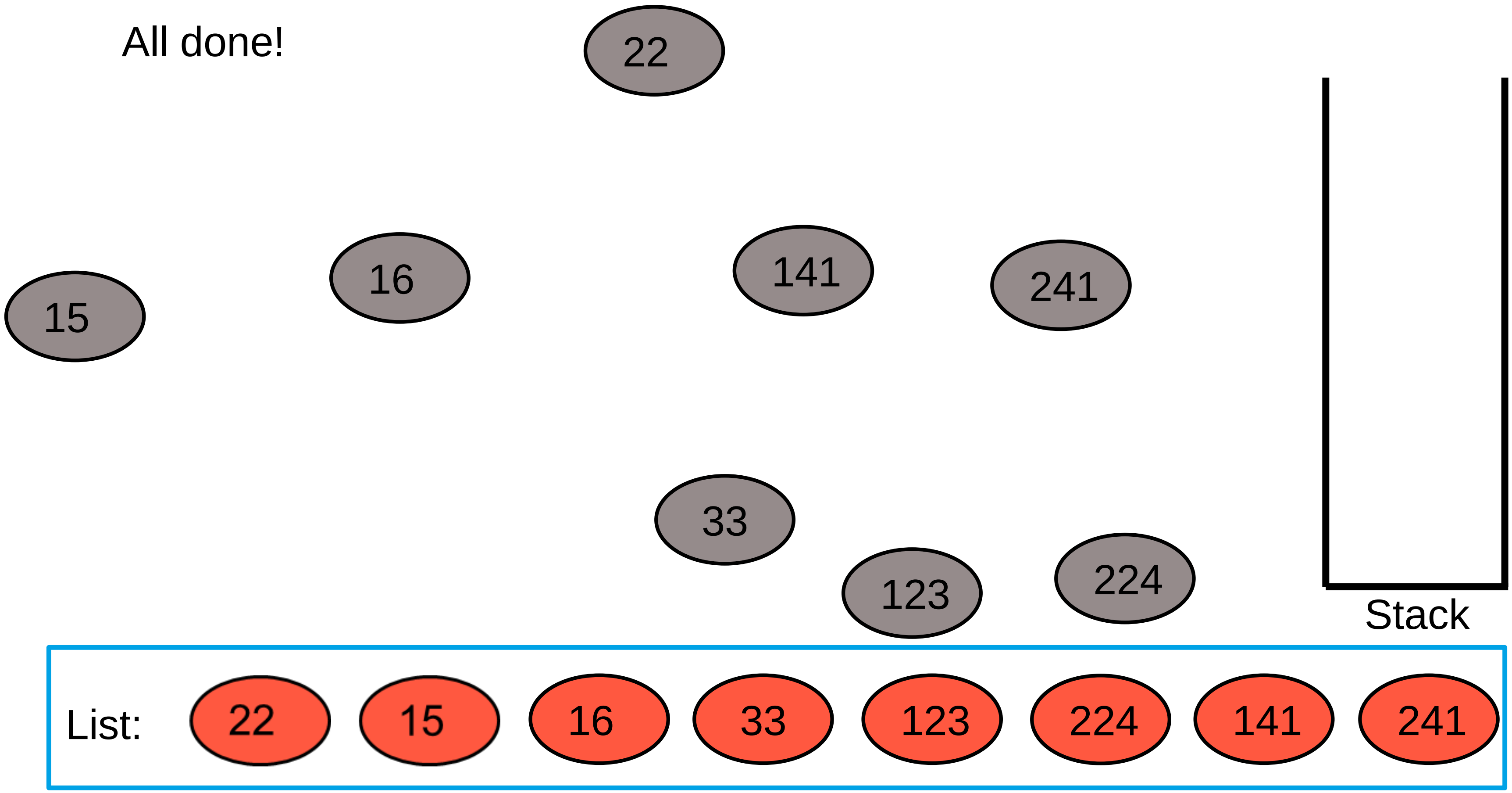


Topological Sort Run-Through (18)



Topological Sort Run-Through (19)

All done!



Top Sort Pseudocode

```
function topological_sort(G):
//Input: A DAG G
//Output: A list of the vertices of G in topological order

   S = Stack()
   L = List()
   for each vertex in G:
      if vertex has no incident edges:
         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 has no incident edges:
            S.push(w)
   return L
```

Top Sort Runtime

- So, what's the runtime?
- Let's consider the major steps:
  1. Create a set of all sources.
  2. While the set isn't empty,
     - Remove a vertex from the set and add it to the sorted list
     - For every edge from that vertex:
       - Delete the edge from the graph
       - Check all of its destination vertices and add them to the set if they have no incoming edges

Top Sort Runtime

- So, what's the runtime?
- Let's consider the major steps:
  1. Create a set of all sources.
  2. While the set isn't empty,
     - Remove a vertex from the set and add it to the sorted list
     - For every edge from that vertex:
       - Delete the edge from the graph
       - Check all of its destination vertices and add them to the set if they have no incoming edges

Step 1 requires looping through all of the vertices to find those with no incident edges – *O(|V|)*

Top Sort Runtime

- So, what's the runtime?
- Let's consider the major steps:
  1. Create a set of all sources.
  2. While the set isn't empty,
     - Remove a vertex from the set and add it to the sorted list
     - For every edge from that vertex:
       - Delete the edge from the graph
       - Check all of its destination vertices and add them to the set if they have no incoming edges

Step 1 requires looping through all of the vertices to find those with no incident edges – *O(|V|)*

Step 2 also requires looping through every vertex, but also looks at edges. Since you only visit the edges that begin at that vertex, every edge gets visited only once. Thus, the runtime for this section is *O(|V| + |E|)*.

Top Sort Runtime

- So, what's the runtime?
- Let's consider the major steps:
  1. Create a set of all sources.
  2. While the set isn't empty,
     - Remove a vertex from the set and add it to the sorted list
     - For every edge from that vertex:
       - Delete the edge from the graph
       - Check all of its destination vertices and add them to the set if they have no incoming edges

Step 1 requires looping through all of the vertices to find those with no incident edges – *O(|V|)*

Step 2 also requires looping through every vertex, but also looks at edges. Since you only visit the edges that begin at that vertex, every edge gets visited only once. Thus, the runtime for this section is *O(|V| + |E|)*.

- Overall, this makes the algorithm run in *O(|V|)* + *O(|V| + |E|)* = *O(2*|V| + |E|)* = ***O(|V| + |E|)*** time.

Top Sort Pseudocode – $O(|V| + |E|)$

```
function topological_sort(G):
//Input: A DAG G
//Output: A list of the vertices of G in topological order

   S = Stack()
   L = List()
   for each vertex in G:                              O(|V|)
      if vertex has no incident edges:
         S.push(vertex)
   while S is not empty:                              O(|V| + |E|)
      v = S.pop()
      L.append(v)
      for each outgoing edge e from v:
         w = e.destination
         delete e
         if w has no incident edges:
            S.push(w)
   return L
```

Top Sort Variations

- What if we're not allowed to remove edges from the input graph?

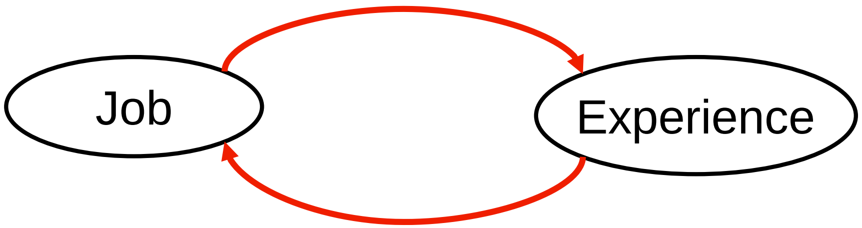
- That's okay! Just use decorations.
  - In the beginning: decorate each vertex with its in-degree
  - Instead of removing an edge, just decrement the in-degree of the destination node. When the in-degree reaches 0, push it onto the stack!

Top Sort Variations (2)

- Do we need to use a stack in topological sort?
  - Nope! Any data structure would do: queue, list, set, etc…
- Different data structures produce different valid orderings. But why do they all work?
  - A node is only added to the data structure when it's degree reaches 0 – i.e. when all of its "prerequisite" nodes have been processed and added to the final output list. This is an invariant throughout the course of the algorithm, so a valid topological order is always guaranteed!

Top Sort: Why only on DAGs?

- When is there no valid topological ordering?



  - I need experience to get a job…I need a job to get experience…I need experience to get a job…I need a job to get experience…Uh oh!
  - If there is a **cycle** there is no valid topological ordering!
- In fact, we can actually use topological sort to see if a graph contains a cycle
  - If there are still edges left in the graph at the end of the algorithm, that means there must be a cycle.