4.7 **Build Order:** You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project's dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.

EXAMPLE

Input:

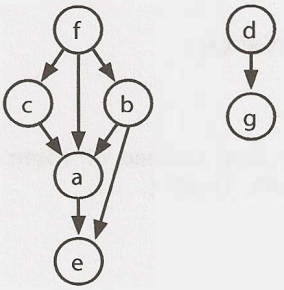
```
projects: a, b, c, d, e, f
dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)
```

Output: `f, e, a, b, d, c`

pg 110

SOLUTION

Visualizing the information as a graph probably works best. Be careful with the direction of the arrows. In the graph below, an arrow from d to g means that d must be compiled before g. You can also draw them in the opposite direction, but you need to consistent and clear about what you mean. Let's draw a fresh example.



In drawing this example (which is *not* the example from the problem description), I looked for a few things.

- I wanted the nodes labeled somewhat randomly. If I had instead put a at the top, with b and c as children, then d and e, it could be misleading. The alphabetical order would match the compile order.
- I wanted a graph with multiple parts/components, since a connected graph is a bit of a special case.
- I wanted a graph where a node links to a node that cannot immediately follow it. For example, f links to a but a cannot immediately follow it (since b and c must come before a and after f).
- I wanted a larger graph since I need to figure out the pattern.
- I wanted nodes with multiple dependencies.

Now that we have a good example, let's get started with an algorithm.

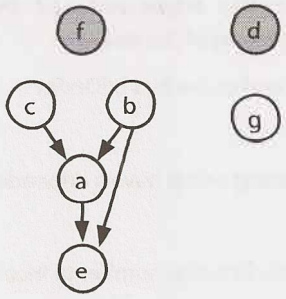
Solution #1

Where do we start? Are there any nodes that we can definitely compile immediately?

Yes. Nodes with no incoming edges can be built immediately since they don't depend on anything. Let's add all such nodes to the build order. In the earlier example, this means we have an order of f, d (or d, f).

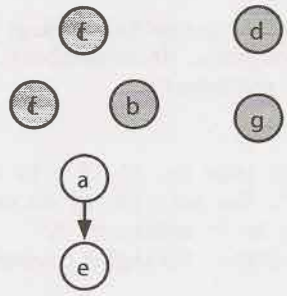
Once we've done that, it's irrelevant that some nodes are dependent on d and f since d and f have already been built. We can reflect this new state by removing d and f's outgoing edges.

```
build order: f, d
```



Next, we know that c, b, and g are free to build since they have no incoming edges. Let's build those and then remove their outgoing edges.

```
build order: f, d, c, b, g
```



Project a can be built next, so let's do that and remove its outgoing edges. This leaves just e. We build that next, giving us a complete build order.

```
build order: f, d, c, b, g, a, e
```

Did this algorithm work, or did we just get lucky? Let's think about the logic.

1. We first added the nodes with no incoming edges. If the set of projects can be built, there must be some "first" project, and that project can't have any dependencies. If a project has no dependencies (incoming edges), then we certainly can't break anything by building it first.
2. We removed all outgoing edges from these roots. This is reasonable. Once those root projects were built, it doesn't matter if another project depends on them.
3. After that, we found the nodes that *now* have no incoming edges. Using the same logic from steps 1 and 2, it's okay if we build these. Now we just repeat the same steps: find the nodes with no dependencies, add them to the build order, remove their outgoing edges, and repeat.
4. What if there are nodes remaining, but all have dependencies (incoming edges)? This means there's no way to build the system. We should return an error.

The implementation follows this approach very closely.

Initialization and setup:

1. Build a graph where each project is a node and its outgoing edges represent the projects that depend on it. That is, if A has an edge to B (A -> B), it means B has a dependency on A and therefore A must be built before B. Each node also tracks the number of *incoming* edges.
2. Initialize a `buildOrder` array. Once we determine a project's build order, we add it to the array. We also continue to iterate through the array, using a `toBeProcessed` pointer to point to the next node to be fully processed.

3. Find all the nodes with zero incoming edges and add those to a `buildOrder` array. Set a `toBeProcessed` pointer to the beginning of the array.

Repeat until `toBeProcessed` is at the end of the `buildOrder`:

1. Read node at `toBeProcessed`.
    - If `node` is `null`, then all remaining nodes have a dependency and we have detected a cycle.
2. For each child of node:
    - Decrement `child.dependencies` (the number of incoming edges).
    - If `child.dependencies` is zero, add `child` to end of `buildOrder`.
3. Increment `toBeProcessed`.

The code below implements this algorithm.

```
1  /* Find a correct build order. */
2  Project[] findBuildOrder(String[] projects, String[][] dependencies) {
3    Graph graph = buildGraph(projects, dependencies);
4    return orderProjects(graph.getNodes());
5  }
6
7  /* Build the graph, adding the edge (a, b) if b is dependent on a. Assumes a pair
8   * is listed in "build order". The pair (a, b) in dependencies indicates that b
9   * depends on a and a must be built before b. */
10 Graph buildGraph(String[] projects, String[][] dependencies) {
11   Graph graph = new Graph();
12   for (String project : projects) {
13     graph.createNode(project);
14   }
15
16   for (String[] dependency : dependencies) {
17     String first = dependency[0];
18     String second = dependency[1];
19     graph.addEdge(first, second);
20   }
21
22   return graph;
23 }
24
25 /* Return a list of the projects a correct build order.*/
26 Project[] orderProjects(ArrayList<Project> projects) {
27   Project[] order = new Project[projects.size()];
28
29   /* Add "roots" to the build order first.*/
30   int endOfList = addNonDependent(order, projects, 0);
31
32   int toBeProcessed = 0;
33   while (toBeProcessed < order.length) {
34     Project current = order[toBeProcessed];
35
36     /* We have a circular dependency since there are no remaining projects with
37      * zero dependencies. */
38     if (current == null) {
39       return null;
40     }
41
```

```
      /* Remove myself as a dependency. */
      ArrayList<Project> children = current.getChildren();
      for (Project child : children) {
         child.decrementDependencies();
      }

      /* Add children that have no one depending on them. */
      endOfList = addNonDependent(order, children, endOfList);
      toBeProcessed++;
   }

   return order;
}

/* A helper function to insert projects with zero dependencies into the order
 * array, starting at index offset. */
int addNonDependent(Project[] order, ArrayList<Project> projects, int offset) {
   for (Project project : projects) {
      if (project.getNumberDependencies() == 0) {
         order[offset] = project;
         offset++;
      }
   }
   return offset;
}

public class Graph {
   private ArrayList<Project> nodes = new ArrayList<Project>();
   private HashMap<String, Project> map = new HashMap<String, Project>();

   public Project getOrCreateNode(String name) {
      if (!map.containsKey(name)) {
         Project node = new Project(name);
         nodes.add(node);
         map.put(name, node);
      }

      return map.get(name);
   }

   public void addEdge(String startName, String endName) {
      Project start = getOrCreateNode(startName);
      Project end = getOrCreateNode(endName);
      start.addNeighbor(end);
   }

   public ArrayList<Project> getNodes() { return nodes; }
}

public class Project {
   private ArrayList<Project> children = new ArrayList<Project>();
   private HashMap<String, Project> map = new HashMap<String, Project>();
   private String name;
   private int dependencies = 0;

   public Project(String n) { name = n; }
```

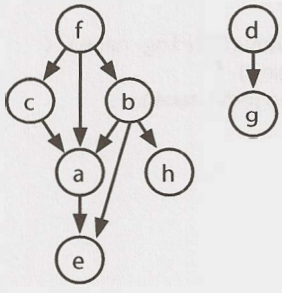
```
98
99    public void addNeighbor(Project node) {
100     if (!map.containsKey(node.getName())) {
101       children.add(node);
102       map.put(node.getName(), node);
103       node.incrementDependencies();
104     }
105   }
106
107   public void incrementDependencies() { dependencies++; }
108   public void decrementDependencies() { dependencies--; }
109
110   public String getName() { return name; }
111   public ArrayList<Project> getChildren() { return children; }
112   public int getNumberDependencies() { return dependencies; }
113 }
```

This solution takes O(P + D) time, where P is the number of projects and D is the number of dependency pairs.

Note: You might recognize this as the topological sort algorithm on page 632. We've rederived this from scratch. Most people won't know this algorithm and it's reasonable for an interviewer to expect you to be able to derive it.

Solution #2

Alternatively, we can use depth-first search (DFS) to find the build path.



Suppose we picked an arbitrary node (say b) and performed a depth-first search on it. When we get to the end of a path and can't go any further (which will happen at h and e), we know that those terminating nodes can be the last projects to be built. No projects depend on them.

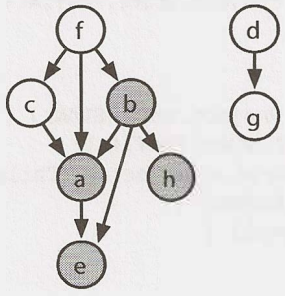
```
DFS(b)                              // Step 1
    DFS(h)                          // Step 2
        build order = ..., h        // Step 3
    DFS(a)                          // Step 4
        DFS(e)                      // Step 5
            build order = ..., e, h // Step 6
        ...                         // Step 7+
    ...
```

Now, consider what happens at node a when we return from the DFS of e. We know a's children need to appear after a in the build order. So, once we return from searching a's children (and therefore they have been added), we can choose to add a to the front of the build order.

Once we return from a, and complete the DFS of b's other children, then everything that must appear after b is in the list. Add b to the front.

```
DFS(b)                                  // Step 1
    DFS(h)                              // Step 2
        build order = ..., h            // Step 3
    DFS(a)                              // Step 4
        DFS(e)                          // Step 5
            build order = ..., e, h     // Step 6
        build order = ..., a, e, h      // Step 7
    DFS(e) -> return                    // Step 8
    build order = ..., b, a, e, h       // Step 9
```

Let's mark these nodes as having been built too, just in case someone else needs to build them.



Now what? We can start with any old node again, doing a DFS on it and then adding the node to the front of the build queue when the DFS is completed.

```
DFS(d)
    DFS(g)
        build order = ..., g, b, a, e, h
    build order = ..., d, g, b, a, e, h

DFS(f)
    DFS(c)
        build order = ..., c, d, g, b, a, e, h
    build order = f, c, d, g, b, a, e, h
```

In an algorithm like this, we should think about the issue of cycles. There is no possible build order if there is a cycle. But still, we don't want to get stuck in an infinite loop just because there's no possible solution.

A cycle will happen if, while doing a DFS on a node, we run back into the same path. What we need therefore is a signal that indicates "I'm still processing this node, so if you see the node again, we have a problem."

What we can do for this is to mark each node as a "`partial`" (or "`is visiting`") state just before we start the DFS on it. If we see any node whose state is `partial`, then we know we have a problem. When we're done with this node's DFS, we need to update the state.

We also need a state to indicate "I've already processed/built this node" so we don't re-build the node. Our state therefore can have three options: `COMPLETED`, `PARTIAL`, and `BLANK`.

The code below implements this algorithm.

```
1  Stack<Project> findBuildOrder(String[] projects, String[][] dependencies) {
2    Graph graph = buildGraph(projects, dependencies);
3    return orderProjects(graph.getNodes());
4  }
```

```
5
6  Stack<Project> orderProjects(ArrayList<Project> projects) {
7     Stack<Project> stack = new Stack<Project>();
8     for (Project project : projects) {
9        if (project.getState() == Project.State.BLANK) {
10          if (!doDFS(project, stack)) {
11             return null;
12          }
13       }
14    }
15    return stack;
16 }
17
18 boolean doDFS(Project project, Stack<Project> stack) {
19    if (project.getState() == Project.State.PARTIAL) {
20       return false; // Cycle
21    }
22
23    if (project.getState() == Project.State.BLANK) {
24       project.setState(Project.State.PARTIAL);
25       ArrayList<Project> children = project.getChildren();
26       for (Project child : children) {
27          if (!doDFS(child, stack)) {
28             return false;
29          }
30       }
31       project.setState(Project.State.COMPLETE);
32       stack.push(project);
33    }
34    return true;
35 }
36
37 /* Same as before */
38 Graph buildGraph(String[] projects, String[][] dependencies) {...}
39 public class Graph {}
40
41 /* Essentially equivalent to earlier solution, with state info added and
42  * dependency count removed. */
43 public class Project {
44    public enum State {COMPLETE, PARTIAL, BLANK};
45    private State state = State.BLANK;
46    public State getState() { return state; }
47    public void setState(State st) { state = st; }
48    /* Duplicate code removed for brevity */
49 }
```

Like the earlier algorithm, this solution is O(P+D) time, where P is the number of projects and D is the number of dependency pairs.

By the way, this problem is called **topological sort**: linearly ordering the vertices in a graph such that for every edge (a, b), a appears before b in the linear order.

4.8 First Common Ancestor: Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

pg 110

SOLUTION

If this were a binary search tree, we could modify the `find` operation for the two nodes and see where the paths diverge. Unfortunately, this is not a binary search tree, so we must try other approaches.

Let's assume we're looking for the common ancestor of nodes p and q. One question to ask here is if each node in our tree has a link to its parents.

Solution #1: With Links to Parents

If each node has a link to its parent, we could trace p and q's paths up until they intersect. This is essentially the same problem as question 2.7 which find the intersection of two linked lists. The "linked list" in this case is the path from each node up to the root. (Review this solution on page 221.)

```
1  TreeNode commonAncestor(TreeNode p, TreeNode q) {
2    int delta = depth(p) - depth(q); // get difference in depths
3    TreeNode first = delta > 0 ? q : p; // get shallower node
4    TreeNode second = delta > 0 ? p : q; // get deeper node
5    second = goUpBy(second, Math.abs(delta)); // move deeper node up
6
7    /* Find where paths intersect. */
8    while (first != second && first != null && second != null) {
9      first = first.parent;
10     second = second.parent;
11   }
12   return first == null || second == null ? null : first;
13 }
14
15 TreeNode goUpBy(TreeNode node, int delta) {
16   while (delta > 0 && node != null) {
17     node = node.parent;
18     delta--;
19   }
20   return node;
21 }
22
23 int depth(TreeNode node) {
24   int depth = 0;
25   while (node != null) {
26     node = node.parent;
27     depth++;
28   }
29   return depth;
30 }
```

This approach will take `O(d)` time, where d is the depth of the deeper node.

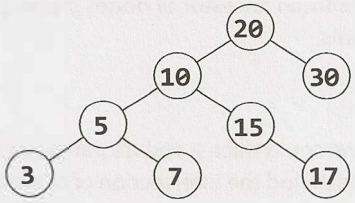
Solution #2: With Links to Parents (Better Worst-Case Runtime)

Similar to the earlier approach, we could trace p's path upwards and check if any of the nodes cover q. The first node that covers q (we already know that every node on this path will cover p) must be the first common ancestor.

Observe that we don't need to re-check the entire subtree. As we move from a node x to its parent y, all the nodes under x have already been checked for q. Therefore, we only need to check the new nodes "uncovered", which will be the nodes under x's sibling.

For example, suppose we're looking for the first common ancestor of node `p = 7` and node `q = 17`. When we go to `p.parent` (5), we uncover the subtree rooted at 3. We therefore need to search this subtree for q.

Next, we go to node 10, uncovering the subtree rooted at 15. We check this subtree for node 17 and—voila—there it is.



To implement this, we can just traverse upwards from p, storing the `parent` and the `sibling` node in a variable. (The `sibling` node is always a child of `parent` and refers to the newly uncovered subtree.) At each iteration, `sibling` gets set to the old `parent`'s sibling node and `parent` gets set to `parent.parent`.

```
1  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2    /* Check if either node is not in the tree, or if one covers the other. */
3    if (!covers(root, p) || !covers(root, q)) {
4      return null;
5    } else if (covers(p, q)) {
6      return p;
7    } else if (covers(q, p)) {
8      return q;
9    }
10
11   /* Traverse upwards until you find a node that covers q. */
12   TreeNode sibling = getSibling(p);
13   TreeNode parent = p.parent;
14   while (!covers(sibling, q)) {
15     sibling = getSibling(parent);
16     parent = parent.parent;
17   }
18   return parent;
19 }
20
21 boolean covers(TreeNode root, TreeNode p) {
22   if (root == null) return false;
23   if (root == p) return true;
24   return covers(root.left, p) || covers(root.right, p);
25 }
26
27 TreeNode getSibling(TreeNode node) {
28   if (node == null || node.parent == null) {
29     return null;
30   }
31
32   TreeNode parent = node.parent;
```

```
33    return parent.left == node ? parent.right : parent.left;
34  }
```

This algorithm takes O(t) time, where t is the size of the subtree for the first common ancestor. In the worst case, this will be O(n), where n is the number of nodes in the tree. We can derive this runtime by noticing that each node in that subtree is searched once.

Solution #3: Without Links to Parents

Alternatively, you could follow a chain in which p and q are on the same side. That is, if p and q are both on the left of the node, branch left to look for the common ancestor. If they are both on the right, branch right to look for the common ancestor. When p and q are no longer on the same side, you must have found the first common ancestor.

The code below implements this approach.

```
1   TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2     /* Error check - one node is not in the tree. */
3     if (!covers(root, p) || !covers(root, q)) {
4       return null;
5     }
6     return ancestorHelper(root, p, q);
7   }
8
9   TreeNode ancestorHelper(TreeNode root, TreeNode p, TreeNode q) {
10    if (root == null || root == p || root == q) {
11      return root;
12    }
13
14    boolean pIsOnLeft = covers(root.left, p);
15    boolean qIsOnLeft = covers(root.left, q);
16    if (pIsOnLeft != qIsOnLeft) { // Nodes are on different side
17      return root;
18    }
19    TreeNode childSide = pIsOnLeft ? root.left : root.right;
20    return ancestorHelper(childSide, p, q);
21  }
22
23  boolean covers(TreeNode root, TreeNode p) {
24    if (root == null) return false;
25    if (root == p) return true;
26    return covers(root.left, p) || covers(root.right, p);
27  }
```

This algorithm runs in O(n) time on a balanced tree. This is because `covers` is called on 2n nodes in the first call (n nodes for the left side, and n nodes for the right side). After that, the algorithm branches left or right, at which point `covers` will be called on $2n/2$ nodes, then $2n/4$, and so on. This results in a runtime of O(n).

We know at this point that we cannot do better than that in terms of the asymptotic runtime since we need to potentially look at every node in the tree. However, we may be able to improve it by a constant multiple.

Solution #4: Optimized

Although Solution #3 is optimal in its runtime, we may recognize that there is still some inefficiency in how it operates. Specifically, `covers` searches all nodes under `root` for p and q, including the nodes in each subtree (`root.left` and `root.right`). Then, it picks one of those subtrees and searches all of its nodes. Each subtree is searched over and over again.