Graphs II, Tries

Discussion 10

Announcements

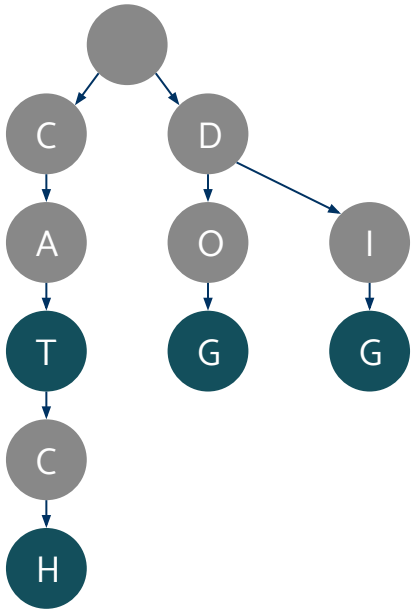
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	4/1 Project 2B/2C due				4/5 Lab 09 due	
					4/12 Lab 10 due	

Content Review

Tries

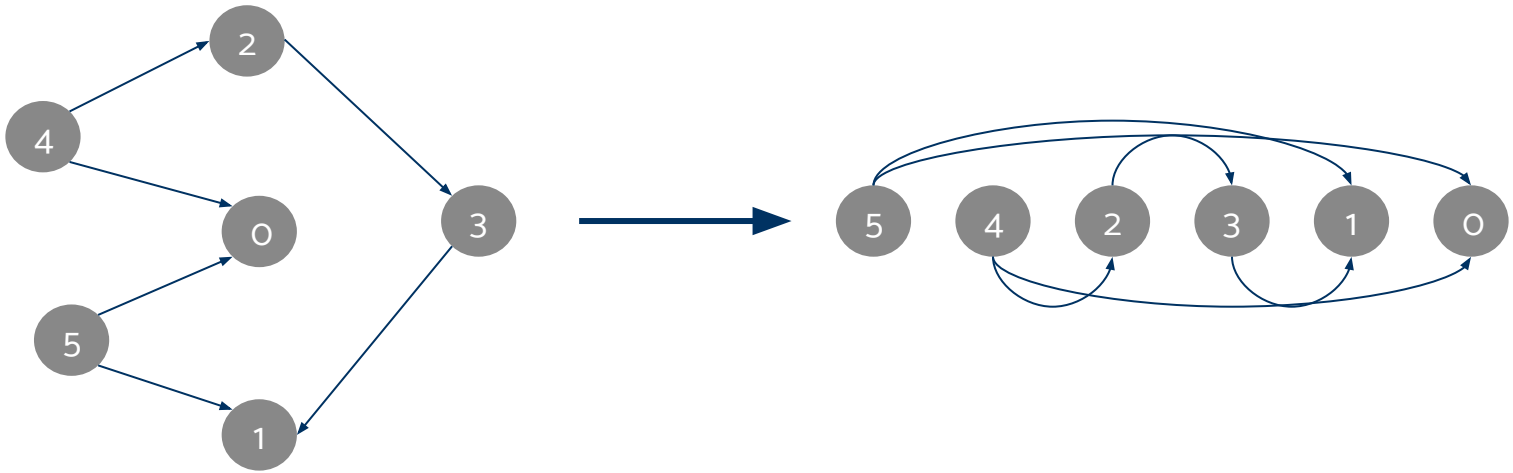
Tries are special trees mostly used for language tasks.

Each node in a trie is marked as being a word-end (a “key”) or not, so you can quickly check whether a word exists within your structure.



Topological Sort

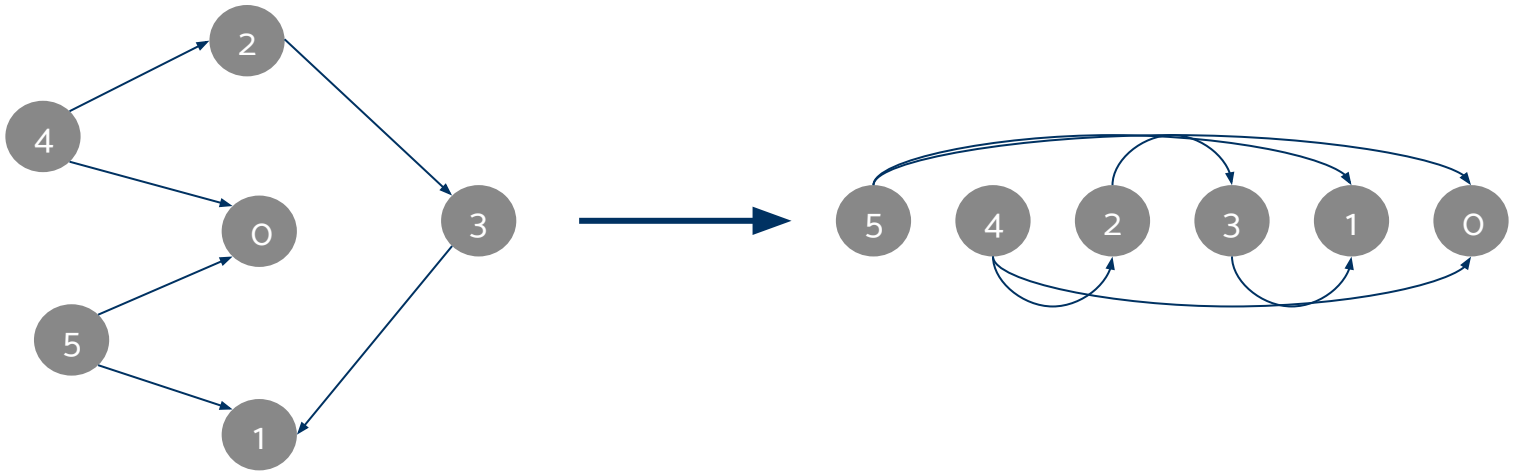
Topological Sort is a way of transforming a directed acyclic graph into a linear ordering of vertices, where for every directed edge u v, vertex u comes before v in the ordering.



Topological Sort

Key Ideas:

- Not having a topological sort indicates a that the graph has directed cycle (only works on DAGs)
- Most DAGs have multiple topological sorts
- Source node: a node that has no incoming edges
- Sink node: a node that has no outgoing edges



Graph Algorithm Runtimes

For a graph with V vertices and E edges:

Graph Algorithm	Runtime
DFS	O (V + E)
BFS	O (V + E)
Dijkstra's	O((V + E) log V)
A*	O((V + E) log V)
Prim's	O((V + E) log V)
Kruskal's	O(E log E)

Worksheet

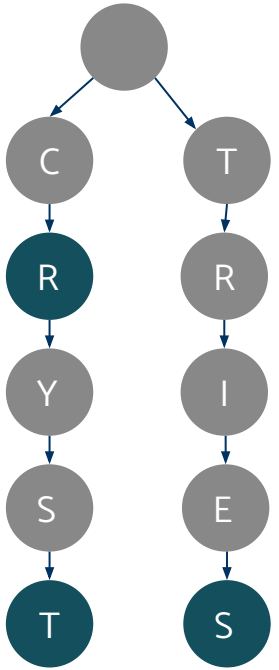
1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

For example, if a TrieSet t contains keys {"cryst", "tries", "cr"}:

t.longestPrefixOf("crystal") returns "cryst"

t.longestPrefixOf("crys") returns "crys"



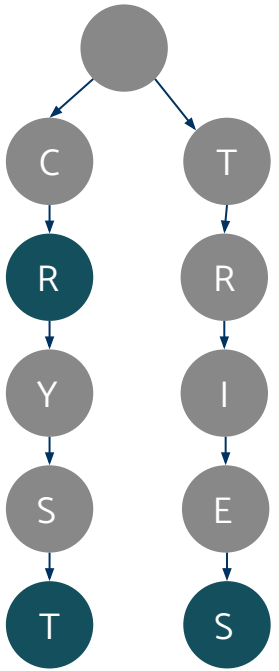
1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

The code uses the StringBuilder class to build strings character-by-character. To add a character to the end of the StringBuilder, use the append(char c) method.

Once all characters have been appended, the resulting String is returned by the toString() method.

```
StringBuilder sb = new StringBuilder();
sb.append('a');
sb.append('b');
System.out.println(sb.toString()); // "ab"
```



1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();
      Node curr = ____________________________;

      for (_____________________________________) {

            ___________________________________________

            ___________________________________________

            ___________________________________________

            ___________________________________________

            ___________________________________________

            ___________________________________________

      }

      return ____________________________________
}
```

```
public class TrieSet {
      private class Node {
            boolean isKey;
            Map<Character, Node> map;

            private Node() {
                  isKey = false;
                  map = new HashMap<>();
            }
      }

      private Node root;

      ...
}
```


1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```

```
public class TrieSet {
      private class Node {
            boolean isKey;
            Map<Character, Node> map;

            private Node() {
                  isKey = false;
                  map = new HashMap<>();
            }
      }

      private Node root;

      ...
}
```

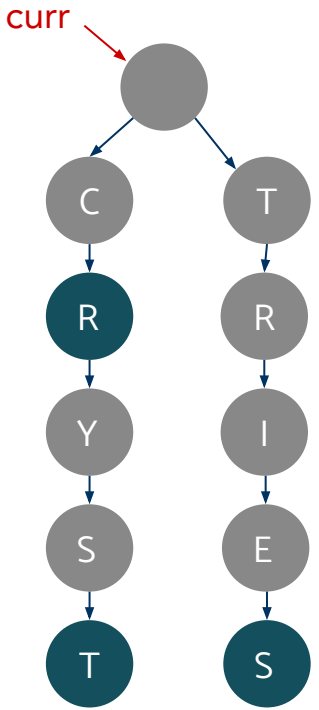
Traverse the trie until you reach a character in key that's missing!

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



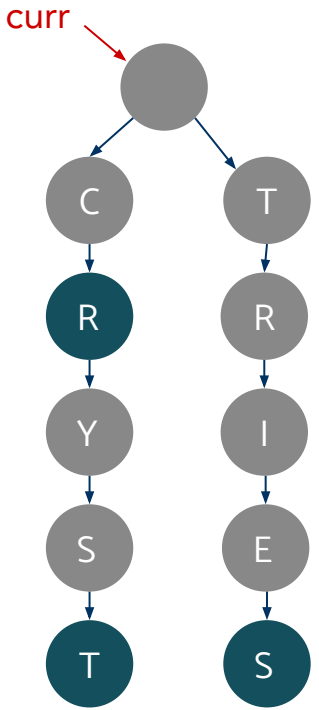
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i); // i = 0, c = 'c'
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



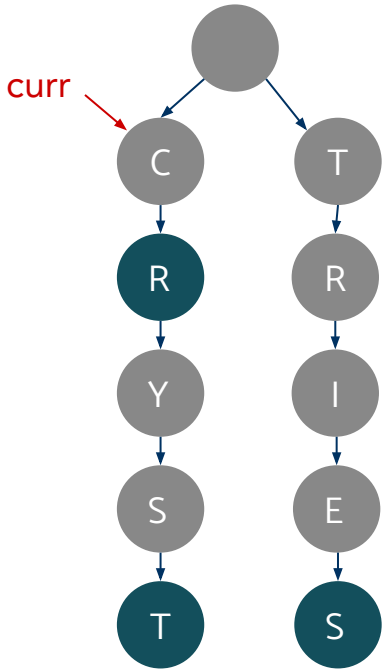

```
t.longestPrefixOf("crystal");
```

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



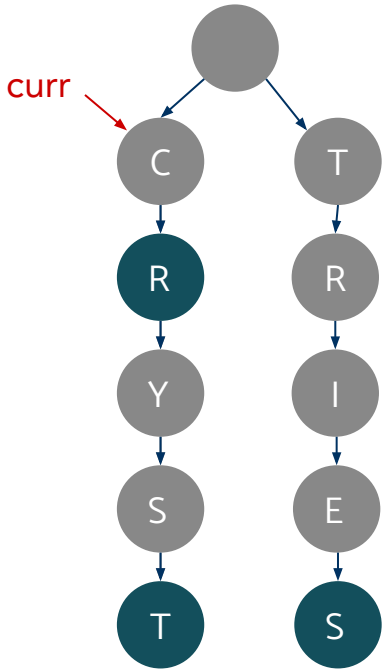
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c); // prefix = <'c'>
      }
      return prefix.toString();
}
```



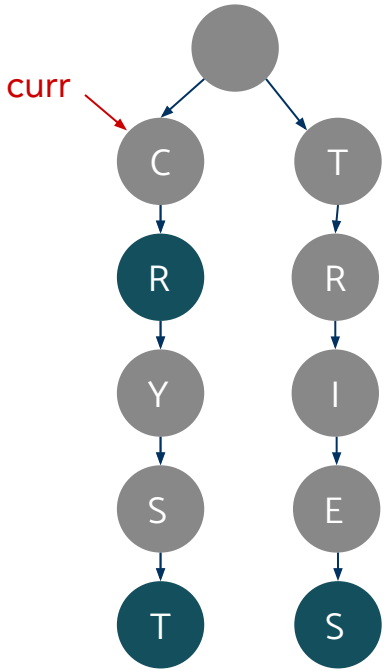

```
t.longestPrefixOf("crystal");
```

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i); // i = 1, c = 'r'
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



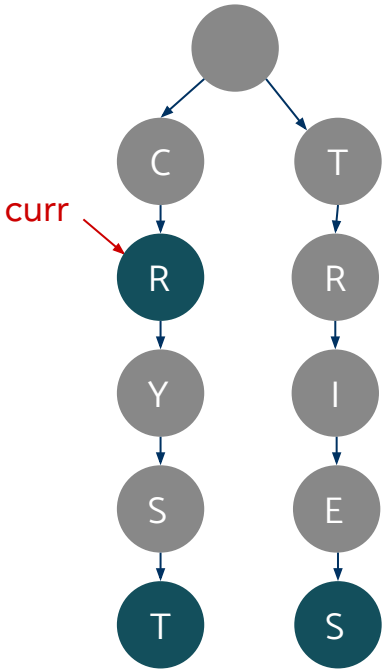
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



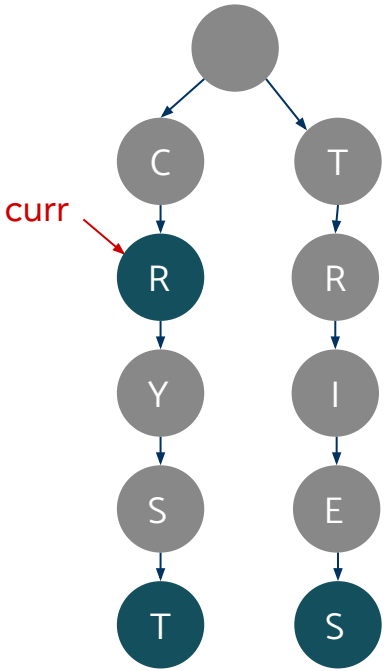

```
t.longestPrefixOf("crystal");
```

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c); // prefix = <'c', 'r'>
      }
      return prefix.toString();
}
```



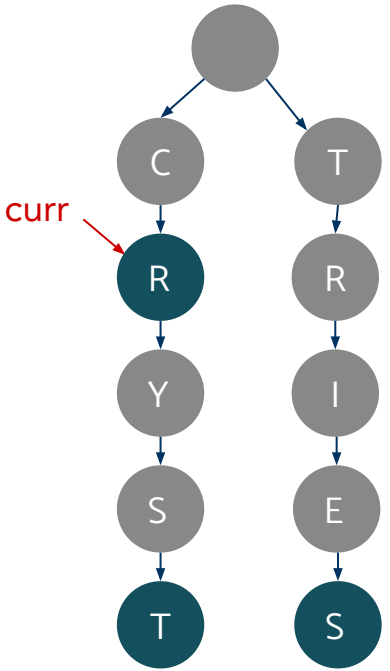

```
t.longestPrefixOf("crystal");
```

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i); // i = 2, c = 'y'
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



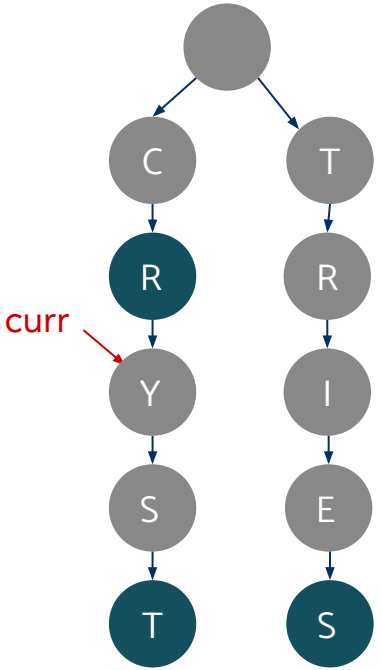
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



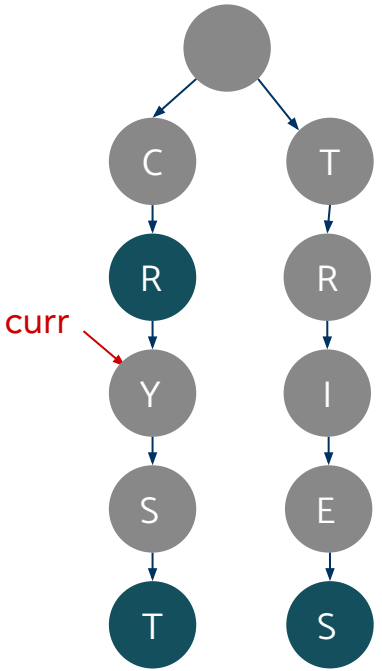
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c); // prefix = <'c', 'r', 'y'>
      }
      return prefix.toString();
}
```



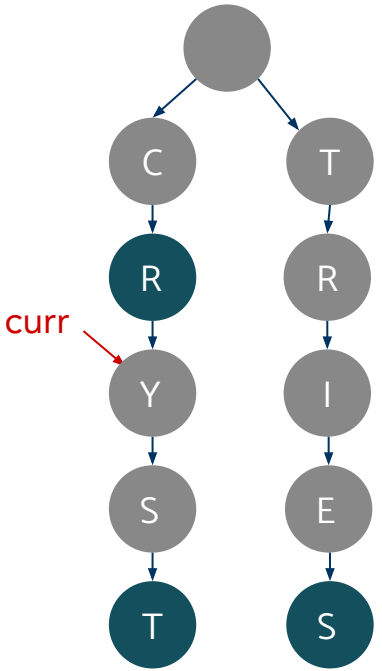

```
t.longestPrefixOf("crystal");
```

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i); // i = 3, c = 's'
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



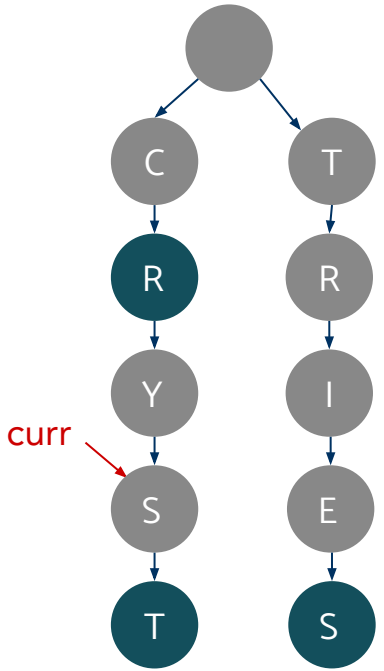
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



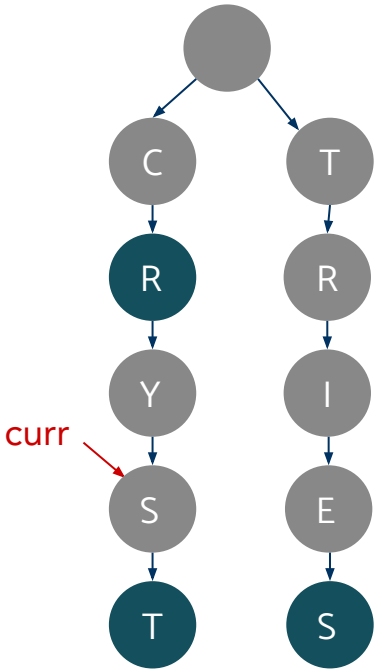
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c); // prefix = <'c', 'r', 'y', 's'>
      }
      return prefix.toString();
}
```



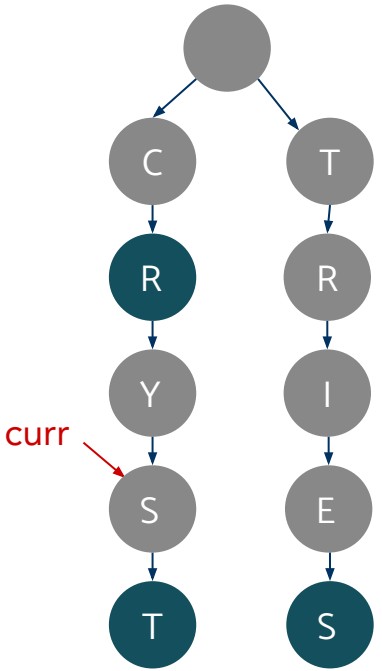

```
t.longestPrefixOf("crystal");
```

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i); // i = 4, c = 't'
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



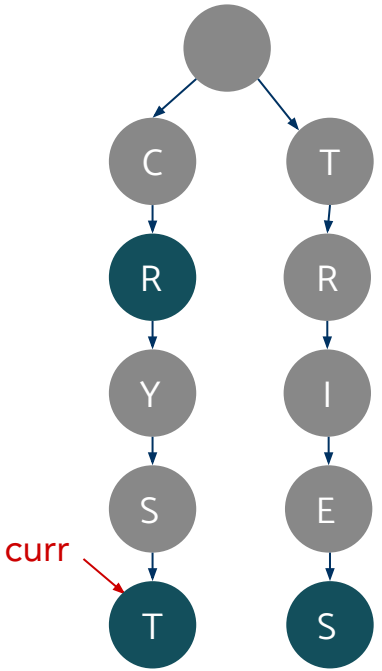
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



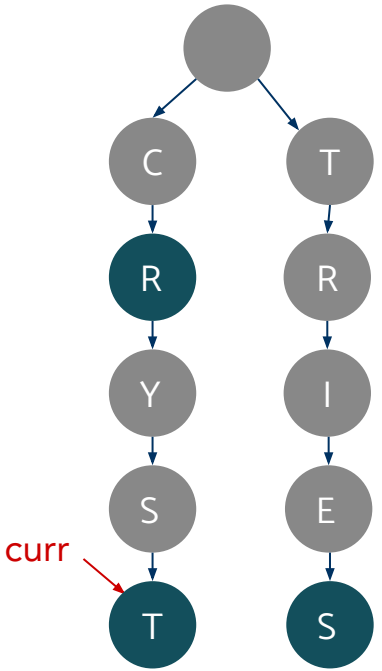
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c); // prefix = <'c', 'r', 'y', 's', 't'>
      }
      return prefix.toString();
}
```



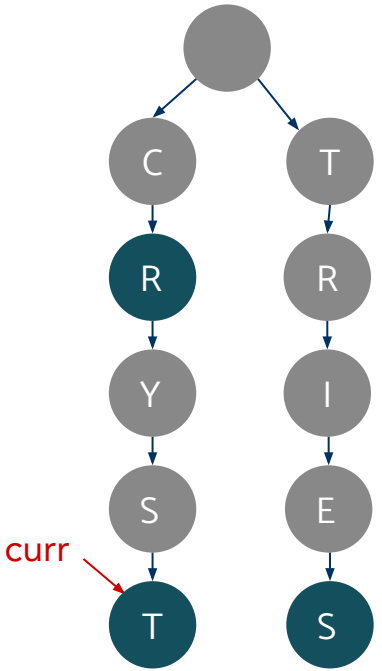

```
t.longestPrefixOf("crystal");
```

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i); // i = 5, c = 'a'
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



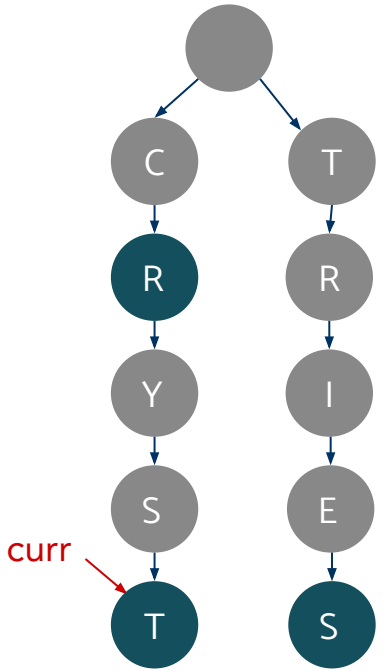

```
t.longestPrefixOf("crystal");
```

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString();
}
```



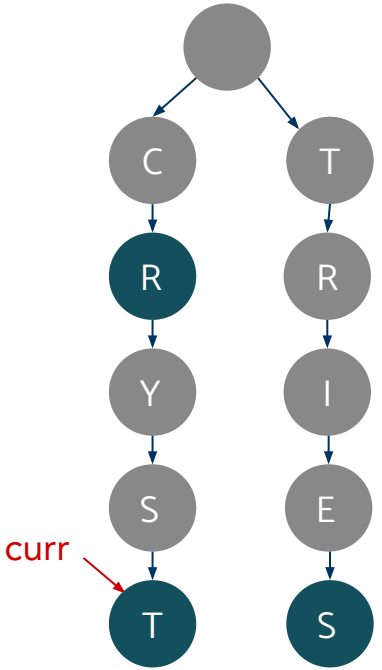
`t.longestPrefixOf("crystal");`

1 Longest Prefix

Fill in the longestPrefixOf(String word) method below such that it returns the longest prefix of word that is also a prefix of a key in the trie.

```
public String longestPrefixOf(String word) {
      int n = word.length();
      StringBuilder prefix = new StringBuilder();

      Node curr = root;
      for (int i = 0; i < n; i++) {
            char c = word.charAt(i);
            if (!curr.map.containsKey(c)) {
                  break;
            }
            curr = curr.map.get(c);
            prefix.append(c);
      }
      return prefix.toString(); // returns “cryst”
}
```




```
t.longestPrefixOf(“crystal”);
```

2 A Tree Takes on Graphs

Your friend at Stanford has come to you for help on their homework! For each of the following statements, determine whether they are true or false; if false, provide counterexamples.

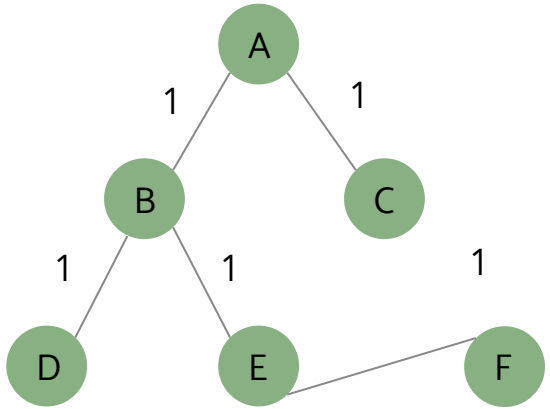
2A A Tree Takes on Graphs

"A graph with edges that all have the same weight will always have multiple MSTs."

2A A Tree Takes on Graphs

"A graph with edges that all have the same weight will always have multiple MSTs."

False! A tree will only have one way to be connected, so it is its own MST

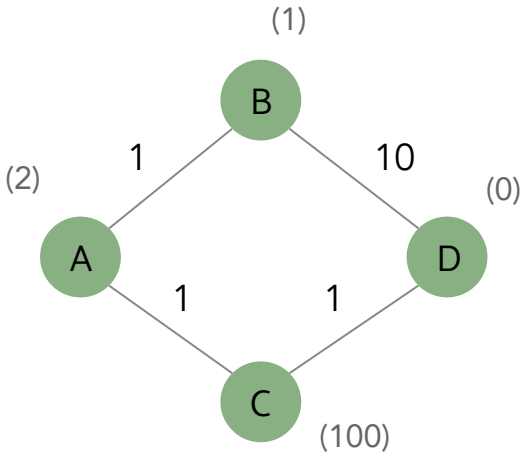


2B A Tree Takes on Graphs

"No matter what heuristic you use, A* search will always find the correct shortest path."

2B A Tree Takes on Graphs

"No matter what heuristic you use, A* search will always find the correct shortest path."



False! A* sets the priority to node v to be distTo[v] + h(v). Because our heuristic for C is very inaccurate, A* prefers exploring B and D before ever going to C.

A* returns A → B → D rather than A → C → D, even though the second is shorter!

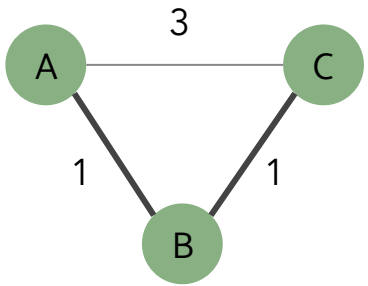
2C A Tree Takes on Graphs

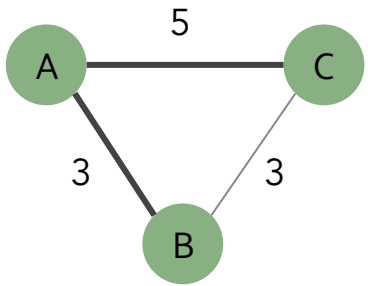
"If you add a constant factor to each edge in a graph, Dijkstra's algorithm will return the same shortest paths tree."

2C A Tree Takes on Graphs

"If you add a constant factor to each edge in a graph, Dijkstra's algorithm will return the same shortest paths tree."

False! After adding 2 to all of the edges in the graph, the shortest paths tree from A changes





Extra challenge: What about multiplying each edge weight by a constant factor?

3A Class Enrollment

You're planning your CS classes for the upcoming semesters, but it's hard to keep track of all the prerequisites! Let's figure out a valid ordering of the classes you're interested in. A valid ordering is an ordering of classes such that every prerequisite of a class is taken before the class itself. Assume we're taking one CS class per semester.

The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 161: CS 61C, CS 70
- CS 170: CS 61B, CS 70

3A Class Enrollment

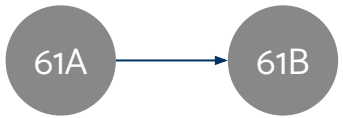
The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70

3A Class Enrollment

The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70



3A Class Enrollment

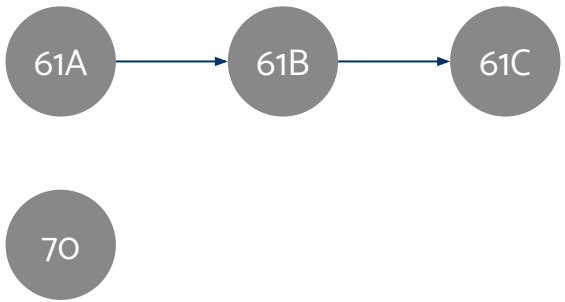
The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70

3A Class Enrollment

The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

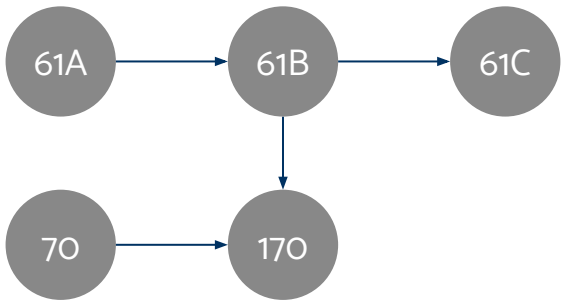
- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70



3A Class Enrollment

The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

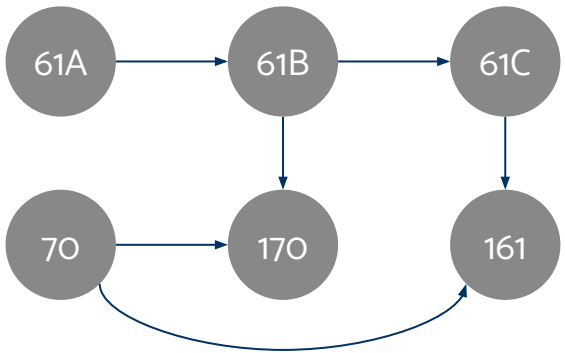
- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70



3A Class Enrollment

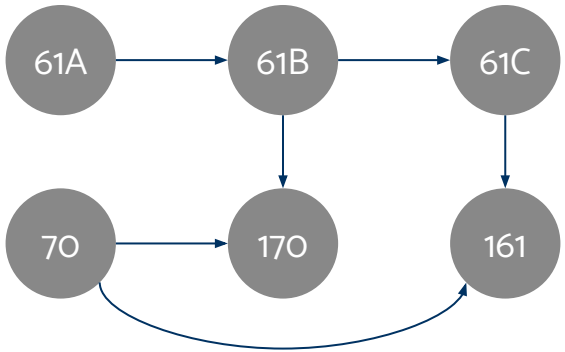
The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70



3B Class Enrollment

Suppose we added a new prerequisite where the student must take CS 161 before CS 170 and CS 170 before CS 61C. Is there still a valid ordering of classes such that no prerequisites are broken? If no, explain.

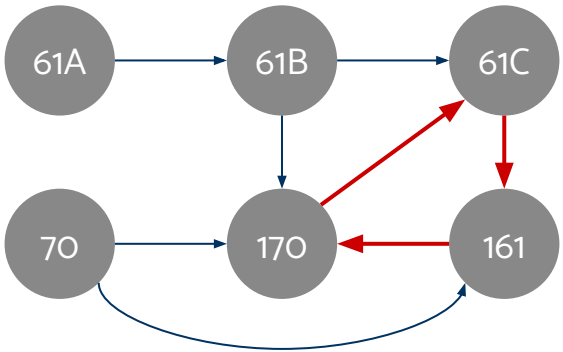


3B Class Enrollment

Suppose we added a new prerequisite where the student must take CS 161 before CS 170 and CS 170 before CS 61C. Is there still a valid ordering of classes such that no prerequisites are broken? If no, explain.

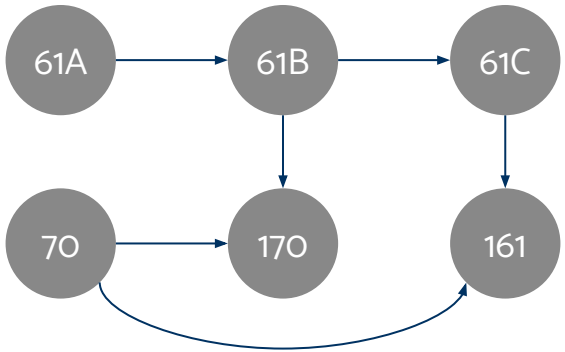
No.

Graph is no longer acyclic because of CS 61C, CS 170, and CS 161. There is no way to order these classes and satisfy all prerequisite constraints!



3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.



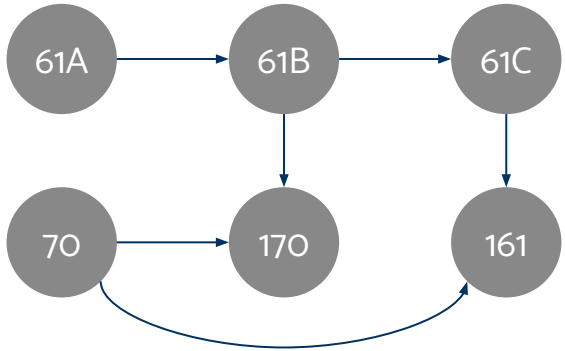
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



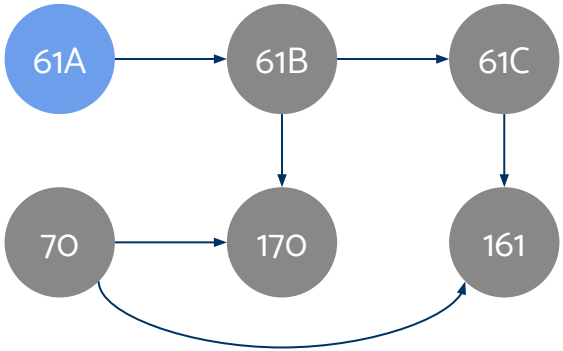
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 61A

Postorder:

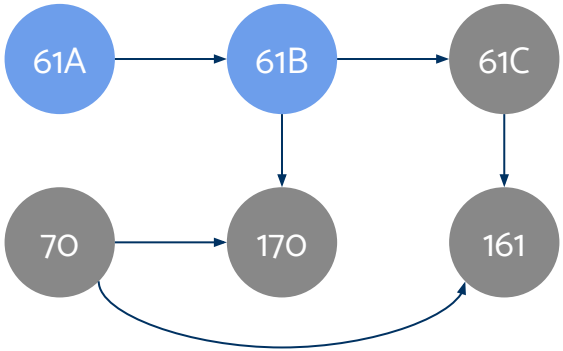
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 61A 61B

Postorder:

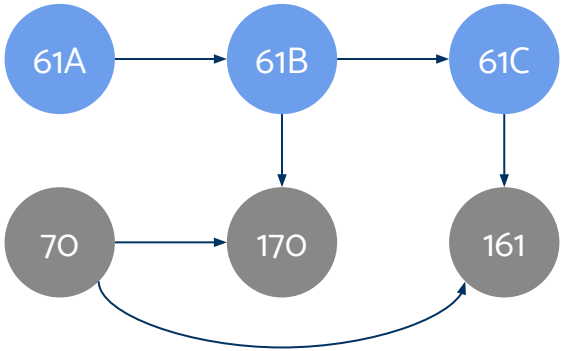
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 61A 61B 61C

Postorder:

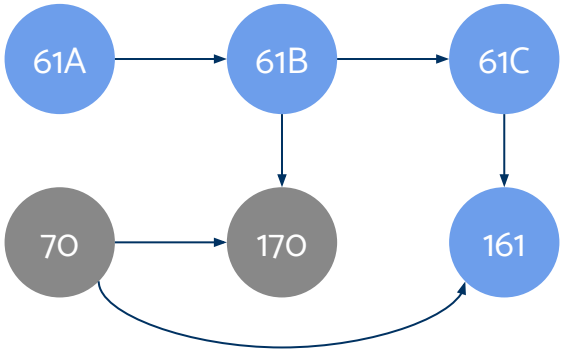
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 61A 61B 61C 161

Postorder:

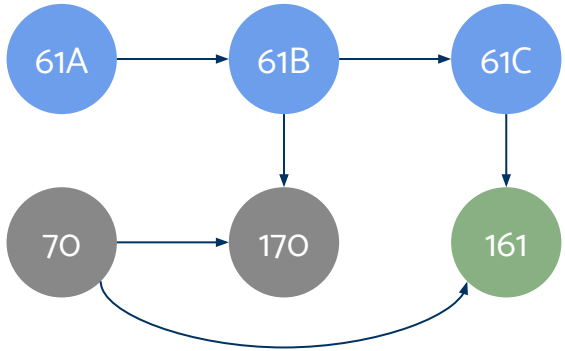
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 61A 61B 61C
Postorder: 161

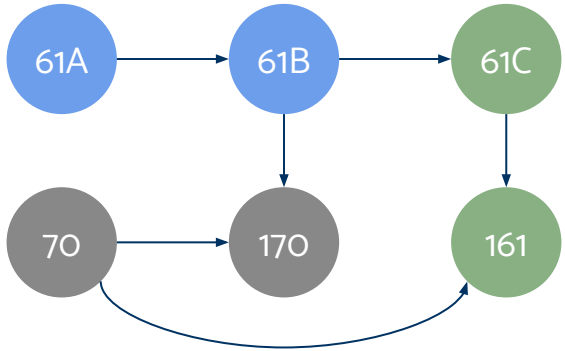
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 61A 61B

Postorder: 161 61C

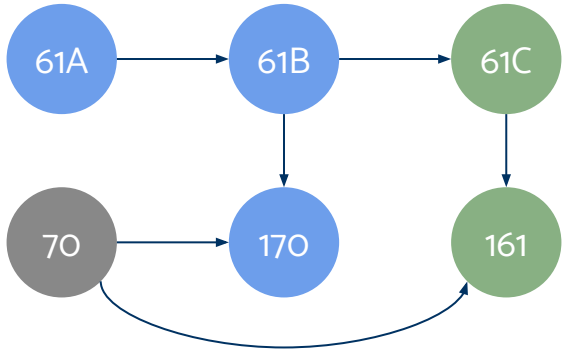
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 61A 61B 170

Postorder: 161 61C

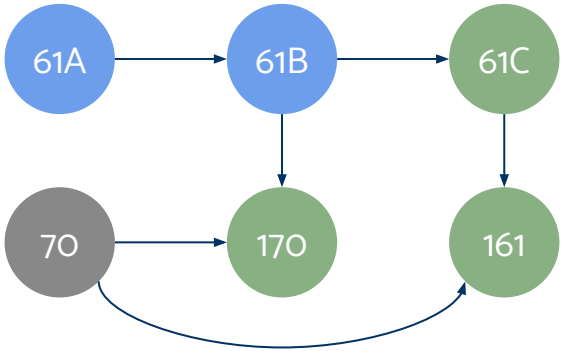
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 61A 61B

Postorder: 161 61C 170

3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order

Stack: 61A
Postorder: 161 61C 170 61B

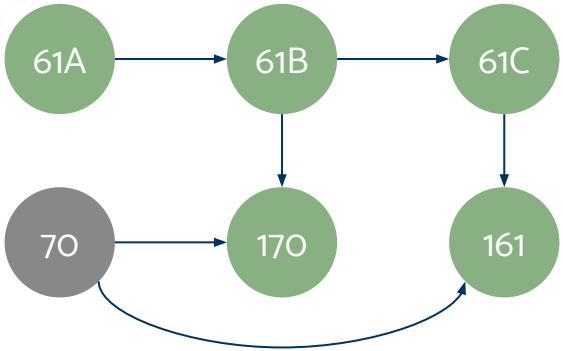
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack:

Postorder: 161 61C 170 61B 61A

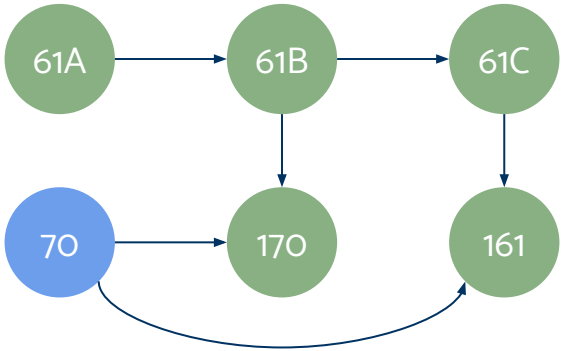
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack: 70

Postorder: 161 61C 170 61B 61A

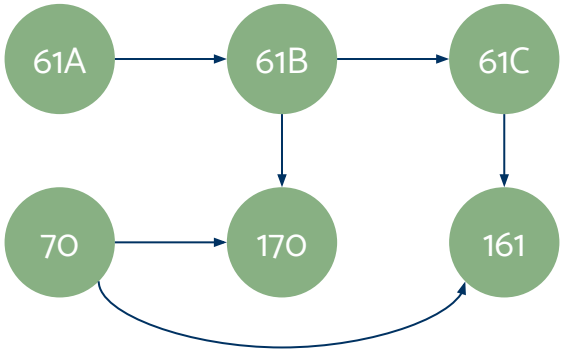
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Stack:

Postorder: 161 61C 170 61B 61A 70

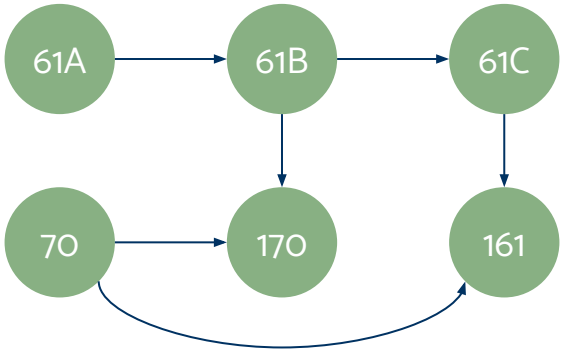
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

Reverse the postorder to get a topologically sorted order



Valid/Topologically Sorted Order:
70 61A 61B 170 61C 161

Stack:
Postorder: 161 61C 170 61B 61A 70

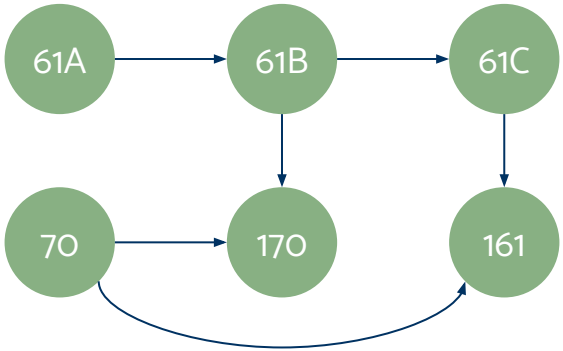
3C Class Enrollment

With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

DFS from every vertex with no incoming edges and compute the postorder

don't reset the marked nodes!

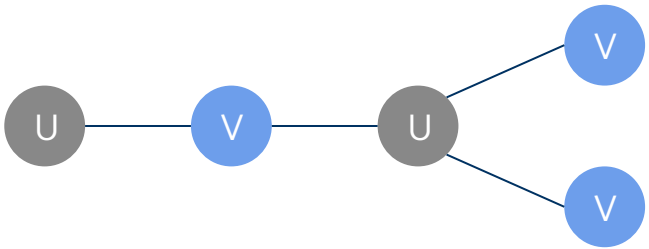
Reverse the postorder to get a topologically sorted order

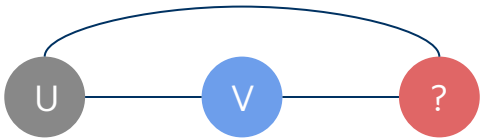


Also works if you DFS from 70 before 61A
61A 61B 61C 70 170 161

4A Graph Algorithm Design

An undirected graph is said to be bipartite if all of its vertices can be divided into two disjoint sets *U* and *V* such that every edge connects an item in *U* to an item in *V*. For example below, the graph on the left is bipartite, whereas on the graph on the right is not. Provide an algorithm which determines whether or not a graph is bipartite. What is the runtime of your algorithm? *Hint: Can you modify an algorithm we already know (ie. graph traversal)?*



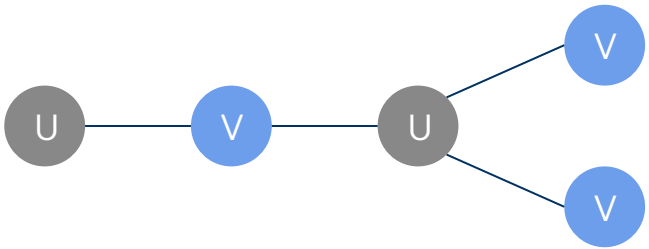


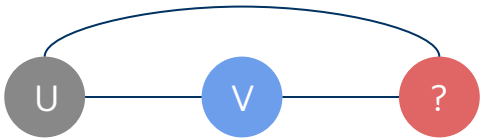
4A Graph Algorithm Design

BFS or DFS! If a node is in set U, then all neighbors must be in set V.

- If we try to label a neighbor as u but they've already been labeled v → not bipartite
- If we successfully mark every node → bipartite

Runtime: Θ(V + E), same as BFS or DFS





4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, then give an example of a graph where this algorithm may not traverse in DFS order.

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, then give an example of a graph where this algorithm may not traverse in DFS order.

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

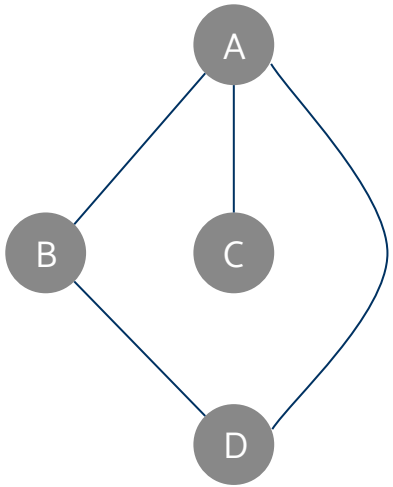
```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    if the vertex was not already marked:
        mark the vertex you just popped
        for each neighbor of the vertex:
            if neighbor not marked:
                push neighbor onto the fringe
```

Identify the bug, then give an example of a graph where this algorithm may not traverse in DFS order.

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```



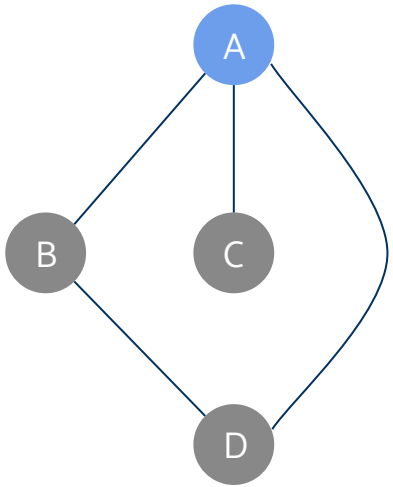
Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**



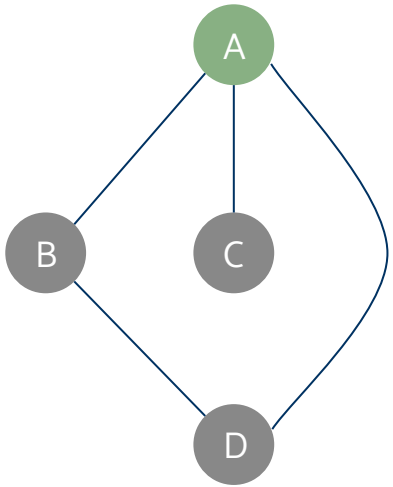
Fringe: A

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**



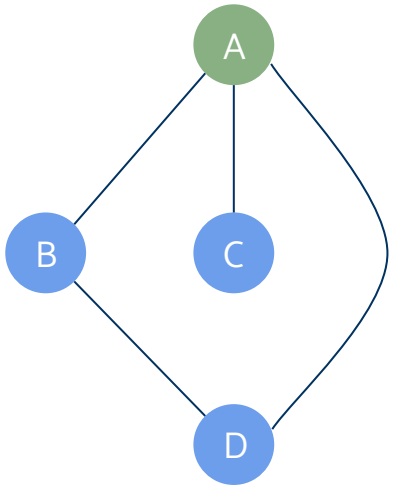
Fringe:

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**



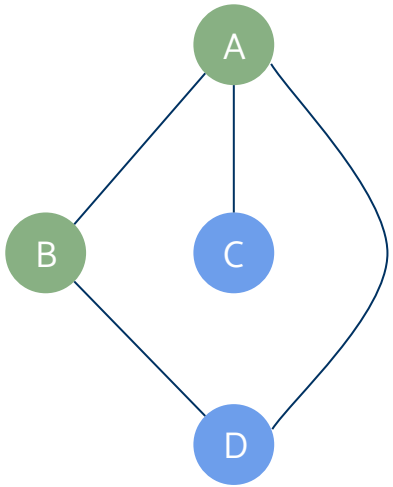
Fringe: D C B

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**



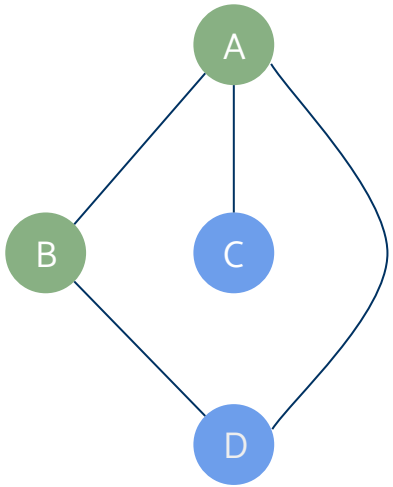
Fringe: D C

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**



Fringe: D C

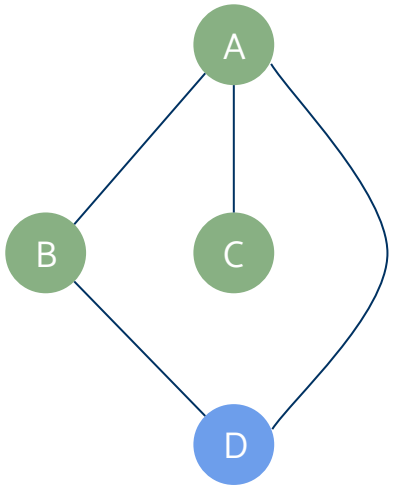
C and D are already marked!

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**



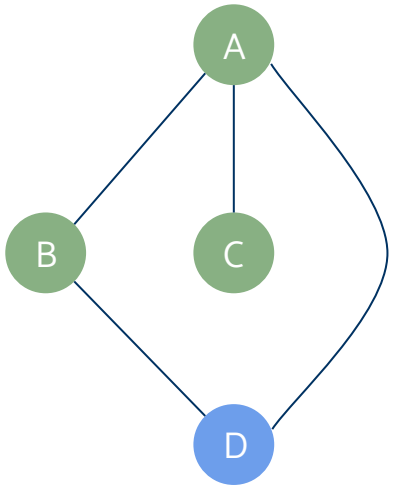
Fringe: D

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**



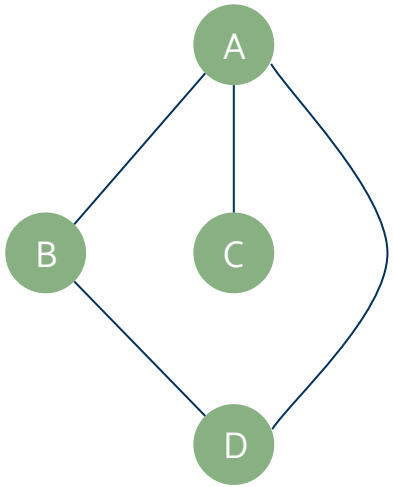
Fringe: D

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**

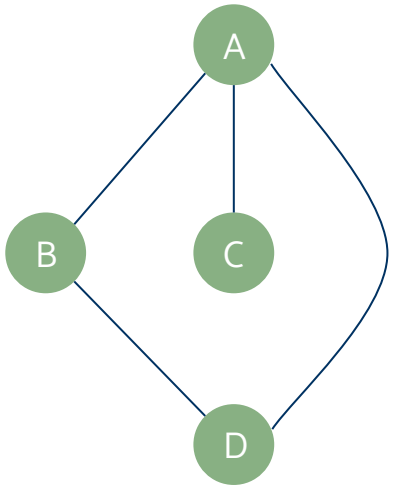


Fringe:

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```



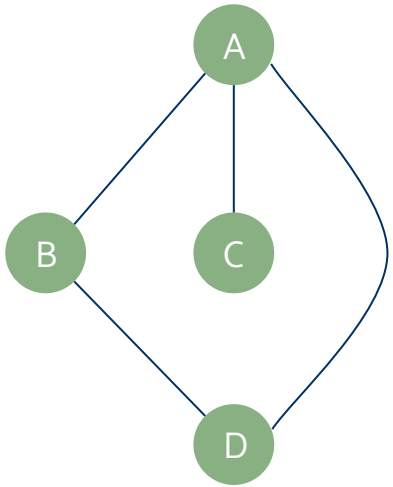
Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**

Fringe:

4B Graph Algorithm Design

Consider the following implementation of DFS, which contains an error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```



Identify the bug, **then give an example of a graph where this algorithm may not traverse in DFS order.**

This DFS implementation visited A → B → C → D.
A correct implementation should visit A → B → D → C.

4C Graph Algorithm Design *Extra*

Provide an algorithm that finds the shortest cycle (in terms of the number of edges used) in a directed graph in O(EV) time and O(E) space, assuming E > V.