CS 61B Fall 2025

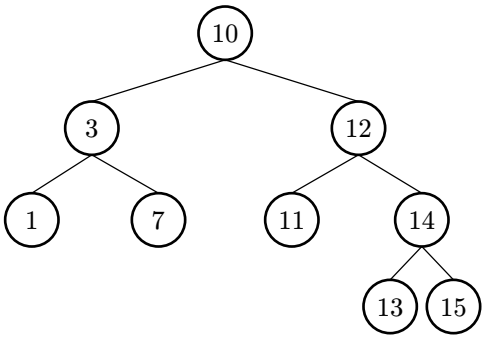
Tree Traversals, Graphs, and Shortest Paths

Exam-Level Discussion 08: October 20, 2025

1 Trees, Graphs, and Traversals

(a) Write the following traversals of the BST below.

Pre-order:
In-order:
Post-order:
Level-order (BFS):



Solution:

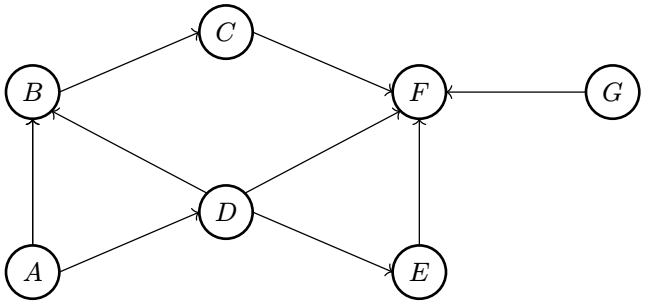
Pre-order: 10 3 1 7 12 11 14 13 15

In-order: 1 3 7 10 11 12 13 14 15

Post-order: 1 7 3 11 13 15 14 12 10

Level-order (BFS): 10 3 12 1 7 11 14 13 15

(b) Write the graph below as an adjacency matrix, then as an adjacency list. What would be different if the graph were undirected instead?



Solution:

Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 0 0 1 0 0 0 0
C 0 0 0 0 0 1 0
D 0 1 0 0 1 1 0
E 0 0 0 0 0 1 0
F 0 0 0 0 0 0 0
G 0 0 0 0 0 1 0
^ start node
```

List:

```
A: {B, D}
B: {C}
C: {F}
D: {B, E, F}
E: {F}
F: {}
G: {F}
```

For the undirected version of the graph, the representations look a bit more symmetric. For your reference, the representations are included below:

Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 1 0 1 1 0 0 0
C 0 1 0 0 0 1 0
D 1 1 0 0 1 1 0
E 0 0 0 1 0 1 0
F 0 0 1 1 1 0 1
G 0 0 0 0 0 1 0
^ start node
```

List:

```
A: {B, D}
B: {A, C, D}
C: {B, F}
D: {A, B, E, F}
E: {D, F}
F: {C, D, E, G}
G: {F}
```

(c) Write the order in which (1) DFS pre-order, (2) DFS post-order, and (3) BFS would visit nodes in the same directed graph above, starting from vertex A. Break ties alphabetically.

Pre-order:

ABCFDE (G)

Post-order:

FCBEDA (G)

BFS:

ABDCEF (G)

To compute DFS, we maintain a stack of nodes, and a visited set. As soon as we add something to our stack, we note it down for preorder. The top node in our stack represents the node we are currently on, and the marked set represents nodes that have been visited. After we add a node to the stack, we visit its lexicographically next unmarked child. If there is none, we pop the topmost node from the stack and note it down for postorder. *Note that there are two ways DFS could run: with restart or without; DFS with restart is the version where if we have exhausted our stack, and still have unmarked nodes left, we restart on the next unmarked node.*

Stack (bottom-top)	VisitedSet	Preorder	Postorder
A	{A}	A	-
AB	{AB}	AB	-
ABC	{ABC}	ABC	-
ABCF	{ABCF}	ABCF	-
ABC	{ABCF}	ABCF	F
AB	{ABCF}	ABCF	FC
A	{ABCF}	ABCF	FCB
AD	{ABCFD}	ABCFD	FCB
ADE	{ABCFDE}	ABCFDE	FCB
AD	{ABCFDE}	ABCFDE	FCBE
A	{ABCFDE}	ABCFDE	FCBED
-	{ABCFDE}	ABCFDE	FCBEDA

If DFS restarts on unmarked nodes, the following happens in the last line. Otherwise, we do not proceed further.

Stack (bottom-top)	VisitedSet	Preorder	Postorder
G	{ABCFDEG}	ABCFDEG	FCBEDAG

For BFS, we use a queue instead of a stack. BFS does not have the notion of in-order and post-order, so we only visit it when we remove it from the queue.

2 Graph Conceptuals

(a) Answer the following questions as either **True** or **False** and provide a brief explanation:

1. If a graph with n vertices has $n-1$ edges, it **must** be a tree.

   Solution: False. The graph **must** be connected.

2. Every edge is looked at exactly twice in each full run of DFS on a connected, undirected graph.

   Solution: True. Say an edge connects u and v. Both u and v will look at the other one through this edge when it's their turn.

3. In BFS, let $d(v)$ be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe (recall that the fringe in BFS is a queue), $|d(u) - d(v)|$ is **always less than** 2.

   Solution: True. Suppose this was not the case. Then, we could have a vertex 2 edges away and a vertex 4 edges away in the fringe at the same time. But, the only way to have a vertex 4 edges away is if a vertex 3 edges away was removed from the fringe. We see this could never occur because the vertex 2 edges away would be removed before the vertex 3 edges away!

(b) Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm.

Solution: We do a depth first search traversal through the graph. While we recurse, if we visit a node that we visited already, then we've found a cycle. Assuming integer labels, we can use something like a `visited` boolean array to keep track of the elements that we've seen, and while looking through a node's neighbors, if `visited` gives true, then that indicates a cycle.

However, since the graph is undirected, if an edge connects vertices `u` and `v`, then `u` is a neighbor of `v`, and `v` is a neighbor of `u`. As such, if we visit `v` after `u`, our algorithm will claim that there is a cycle since `u` is a visited neighbor of `v`. To address this case, when we visit the neighbors of `v`, we should ignore `u`. To implement this in code, we could add the parent as another parameter in the method call. In the worst case, we have to explore at most V edges before finding a cycle (number of edges doesn't matter). So, this runs in $\Theta(V)$.

Pseudocode is provided below (for a disconnected graph, we should call `find_cycle` on each component).

```
find_cycle(v, parent=-1):
    visited[v] = true
    for (v, w) in G:
        if !visited[w]:
            if find_cycle(w, v):
                return True
        else if w != parent:
            return True
    return False
```

3 Sticky Flights

Your airline company has been contracted to fly a large shipment of honey from Honeysville to the 61Bees in Goldenhive City. However, the airplane doesn't have enough fuel capacity to fly directly to Goldenhive City so it will stop at at least one of n airports along the way to refuel. Refueling takes an hour, and if the airport is one of $k < n$ airports, your airplane will be grounded for six hours due to curfews (refueling is included in the six hours). The 61Bees want their honey as soon as possible so please design an algorithm to find the route that will allow your airplane to reach Goldenhive City in the least amount of hours.

Hint: Think of the n airports as a graph, where the paths between them are edges of weight equivalent to the number of hours it takes to fly from airport A to airport B. You may assume that the amount of time it takes to fly from A to B is equal to the amount of time it takes to fly from B to A.

Solution: Since we want to find a path of minimum time (weight), using a Shortest Paths Tree algorithm would make sense for this problem. The problem states that we can represent the airports as a graph, so we first create the graph. We have one node for each of the n airports and for all airports directly reachable from a particular airport, we create an undirected edge with the flight time (in hours) between the two airports as the edge weight. From here, there are multiple approaches we can take to adjust the graph.

1. We can increase the edge weights by the associated refueling or grounding time. Think of undirected edges as two directed edges pointing in opposite directions; we can increase the weight of the directed edge by the refueling/grounding time of the node it points to.

2. We can attach the additional weights to the nodes themselves and modify Dijkstra's algorithm to take into account both edge and node weight. This approach will end up looking very similar to A$*$.

3. For this option, we must create a directed graph rather than an undirected graph. We can split airports into two nodes. We attach all incoming edges to the original node to one node ("left" node) and all outgoing edges to the other ("right" node). Finally we connect the two nodes with a directed edge going from the left node to the right node of weight equal to the refueling/grounding time.

Once the graph is prepared, we run Dijkstra's algorithm starting at the node corresponding to Honeysville and terminate once the node corresponding to Goldenhive City is popped off the fringe. The `distTo` value of Goldenhive City is the minimum time and backtracking from Goldenhive City to Honeysville gives the shortest path.