Today's topics

- DFS
- Topological sort

References

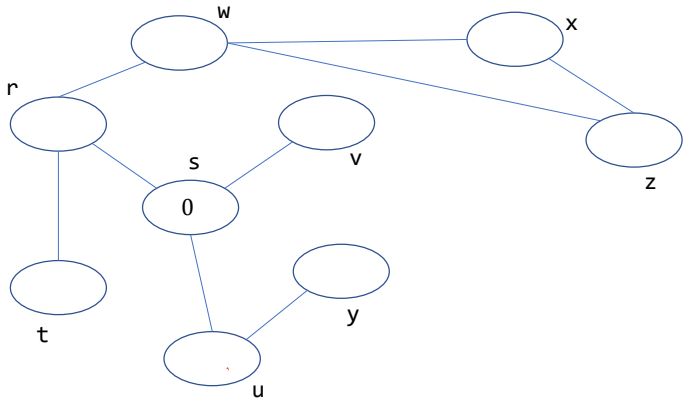
- Read chapter-20 of the CLRS book
- Read chapter-6 from Goodrich and Tamassia book
- https://en.wikipedia.org/wiki/Depth-first_search

Path finding

Path finding

Is there a path from vertex s to vertex z?

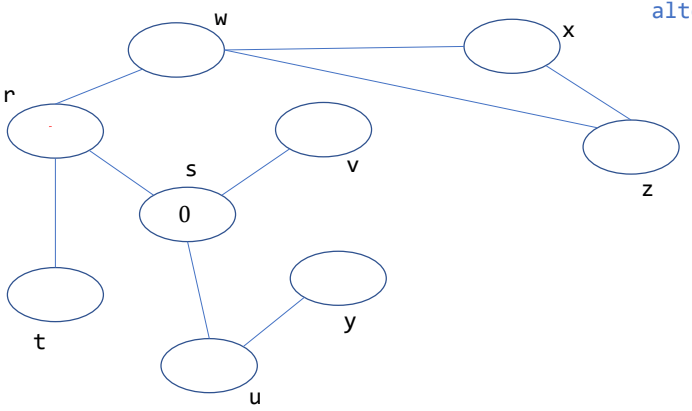
We can use BFS

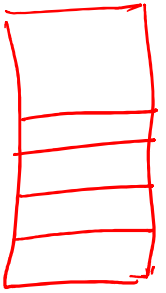


Path finding

Is there a path from vertex s to vertex z?

Is there any other alternative?





Depth-first search (DFS)

Depth-first search (DFS)

- Given a graph G = (V, E), DFS is a graph search algorithm that searches deeper into the graph whenever possible
- DFS visits all the vertices in the graph exactly once
- DFS at a given vertex s recursively explores all outgoing edges of s one-by-one using DFS

Depth-first search (DFS)

- Similar to BFS, in DFS algorithm marks the vertices as white, gray, and black
- Initially, all the vertices are marked as white
- When a white vertex v is visited for the first time, it is marked as gray
- DFS algorithm backtracks if it encounters a non-white vertex
- After visiting all the vertices that are reachable via v using DFS, the DFS algorithm finally marks v as black before leaving v

Depth-first search (DFS)

- DFS also timestamps each vertex to record the order in which the vertices are visited

- Each vertex v contains two timestamps
  - The first timestamp, discovery time (v.d), corresponds to the time when the vertex v is first discovered (or grayed)
  - The second timestamp, finish time (v.f), corresponds to the time when the algorithm leaves the vertex v (or blackens v)

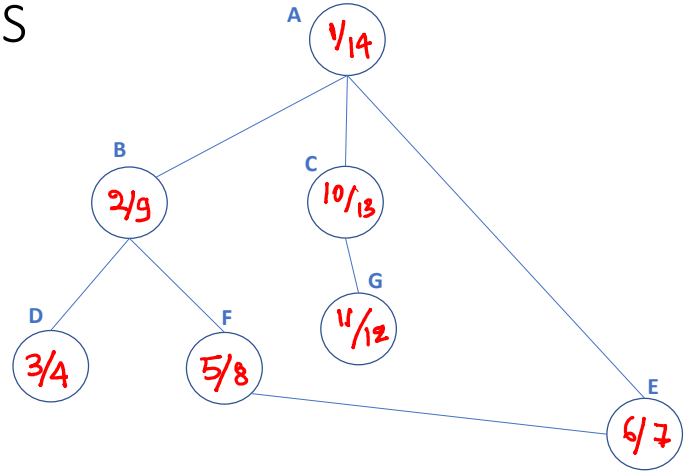
Depth-first search (DFS)

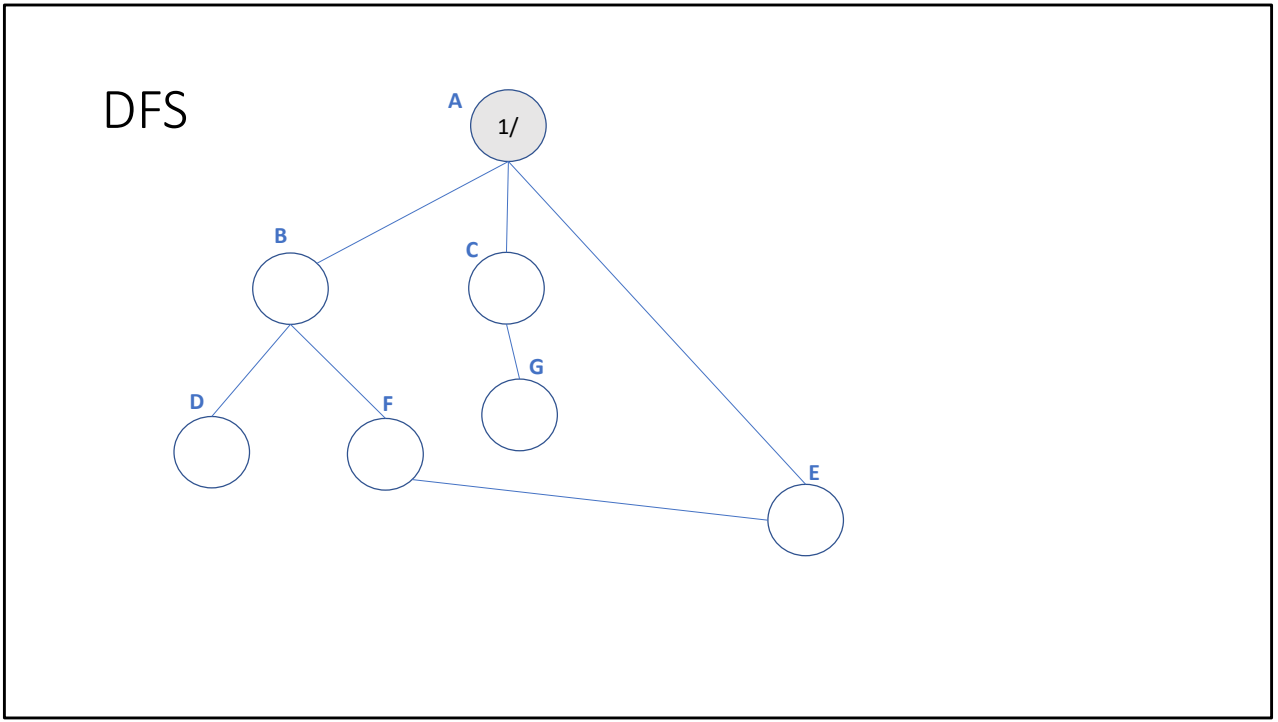
- Discovery edge
  - Whenever a vertex is marked gray, the corresponding edge is called a discovery edge or tree edge

- DFS at a given vertex v finds all vertices that are reachable via v
  - The discovery edges in the connected component of v form a spanning tree

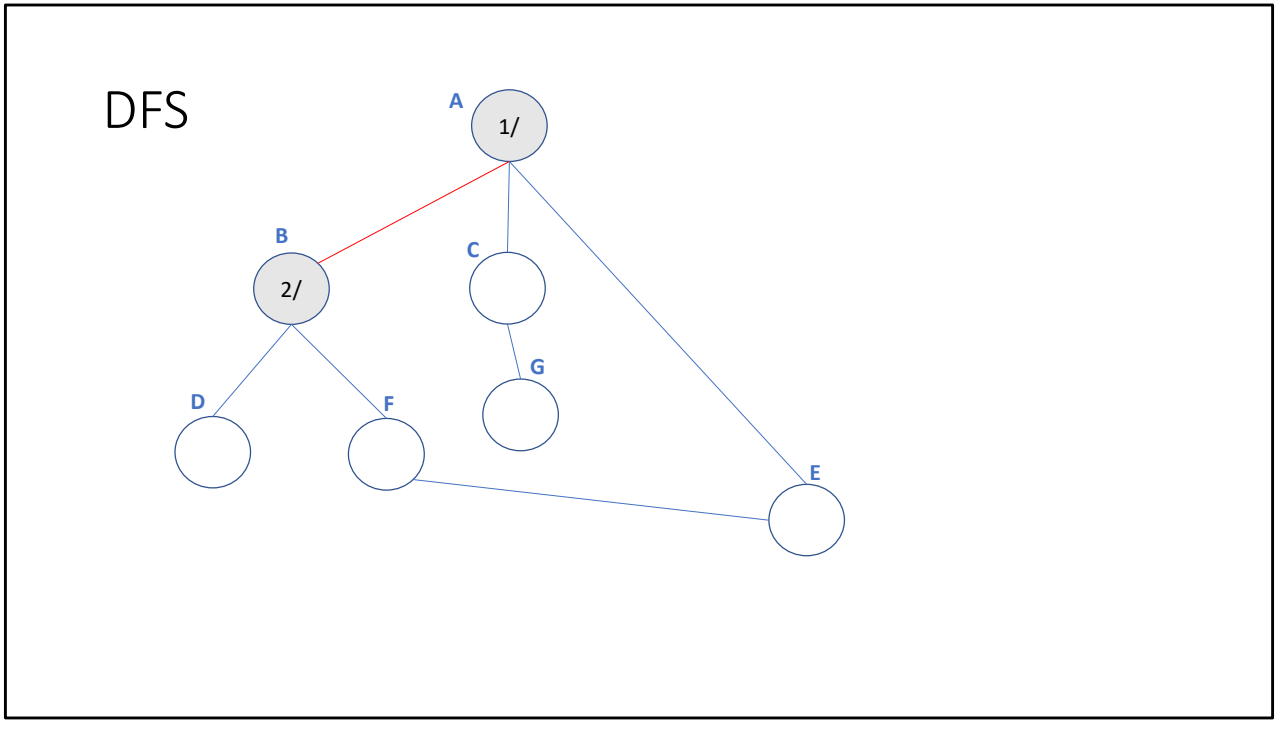
Counter = 1

DFS

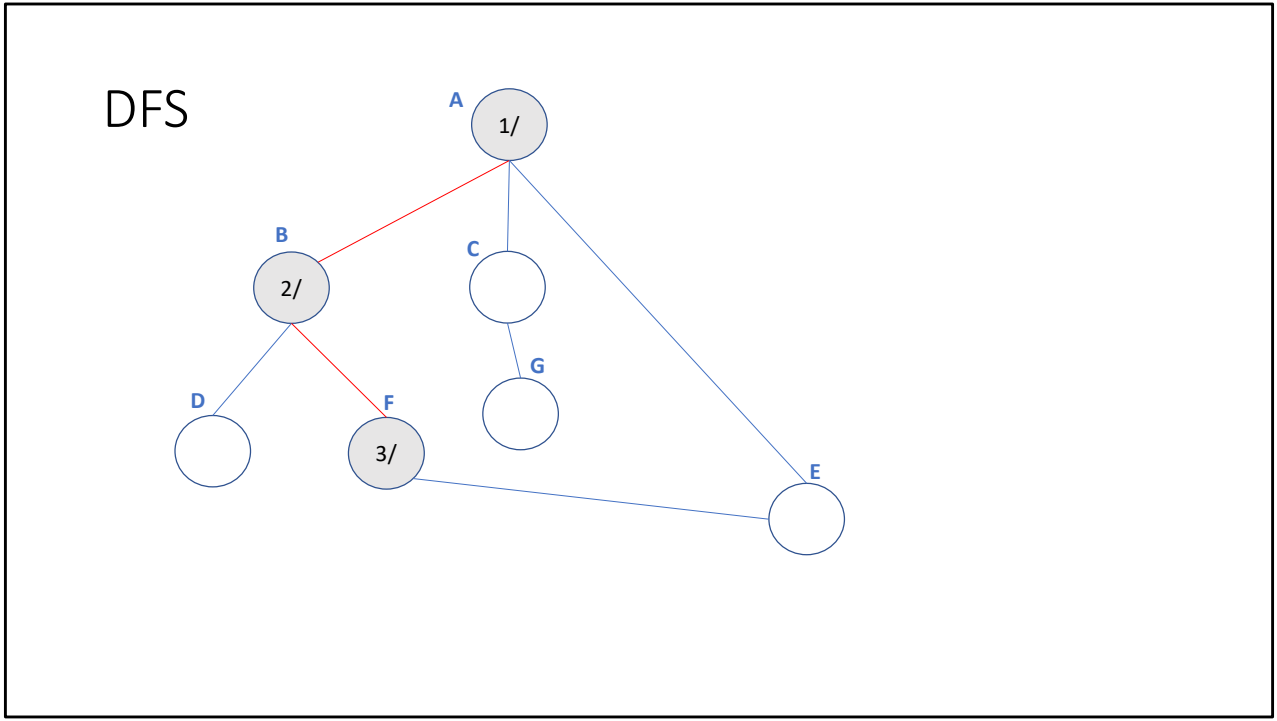




In the DFS algorithm, when a vertex is visited for the first time, it is marked as gray. A vertex is marked as black when we backtrack from the vertex. DFS also maintains a global counter that is incremented every time a vertex is marked as gray or black. A vertex contains two fields: discovery time and finish time. When a node is marked as gray, the discovery time is set to the current counter value. When a node is marked as black, the finish time is set to the current counter value. Initially, the counter value is one. In this example, we start from vertex A. Initially, vertex A is marked as gray, and the discovery time is set to the counter value, i.e., 1. The counter value is updated to 2. The DFS algorithm now picks one neighbor and explores it completely before moving on to the next neighbor.

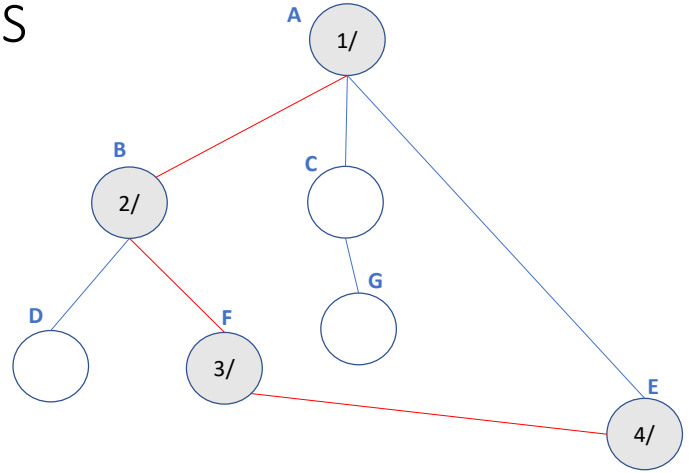


The algorithm picked neighbor B. The discovery time of B is set to the current counter value, i.e., 2. Afterward, the counter is set to 3. When a vertex is discovered for the first time, the corresponding edge is called a tree edge or a discovery edge. Tree edges are shown in red. The DFS will now explore one of the neighbors of B that hasn't been discovered yet.

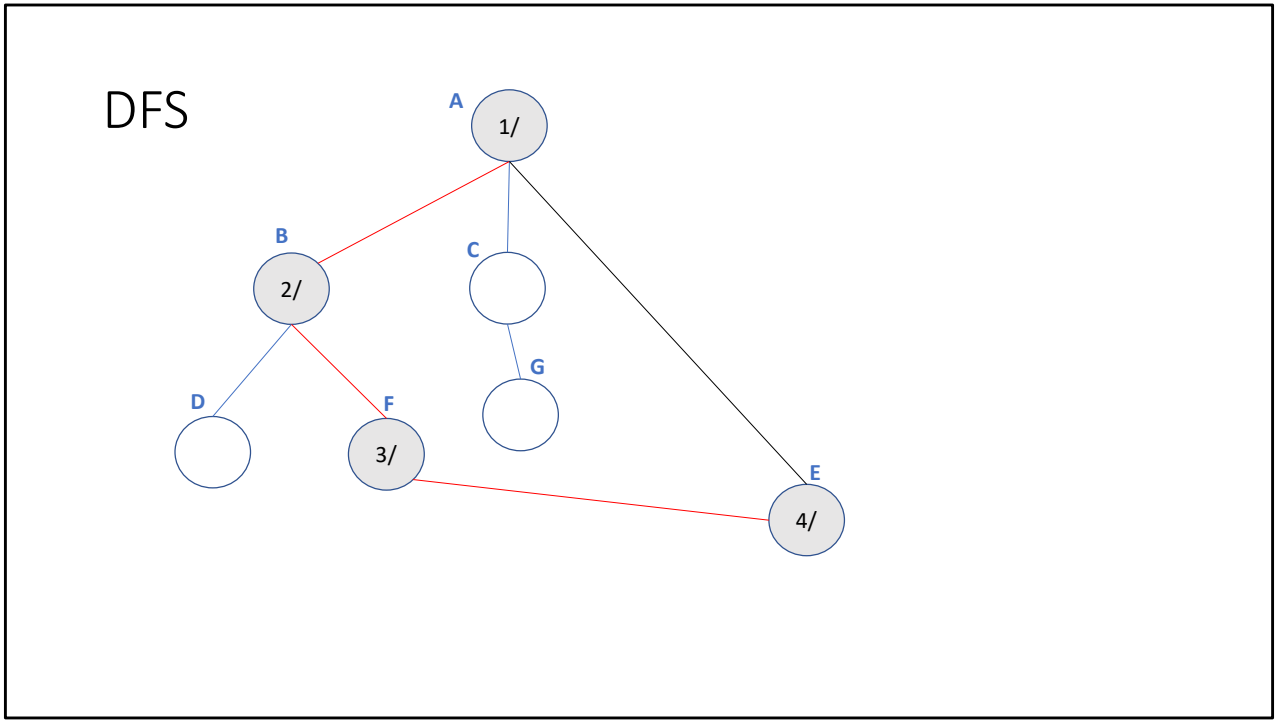


The algorithm picked 3. The discovery time is set to 3. Now DFS will explore neighbors of F.

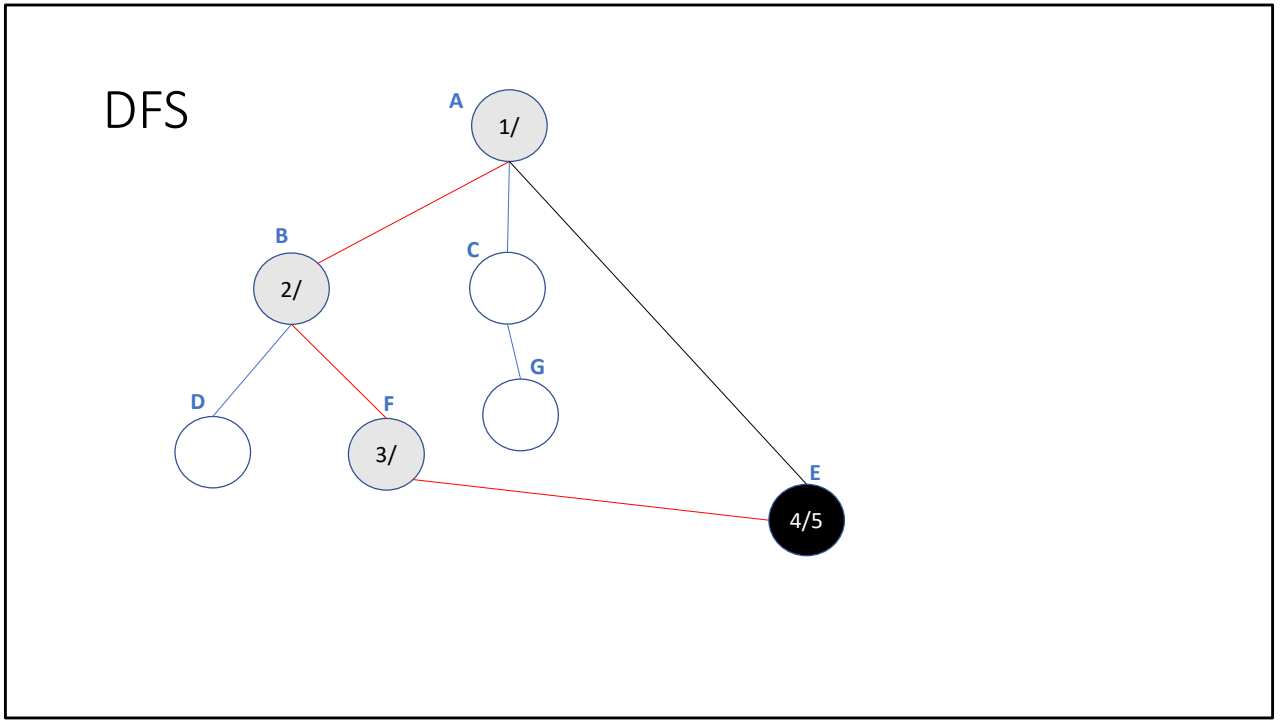
DFS



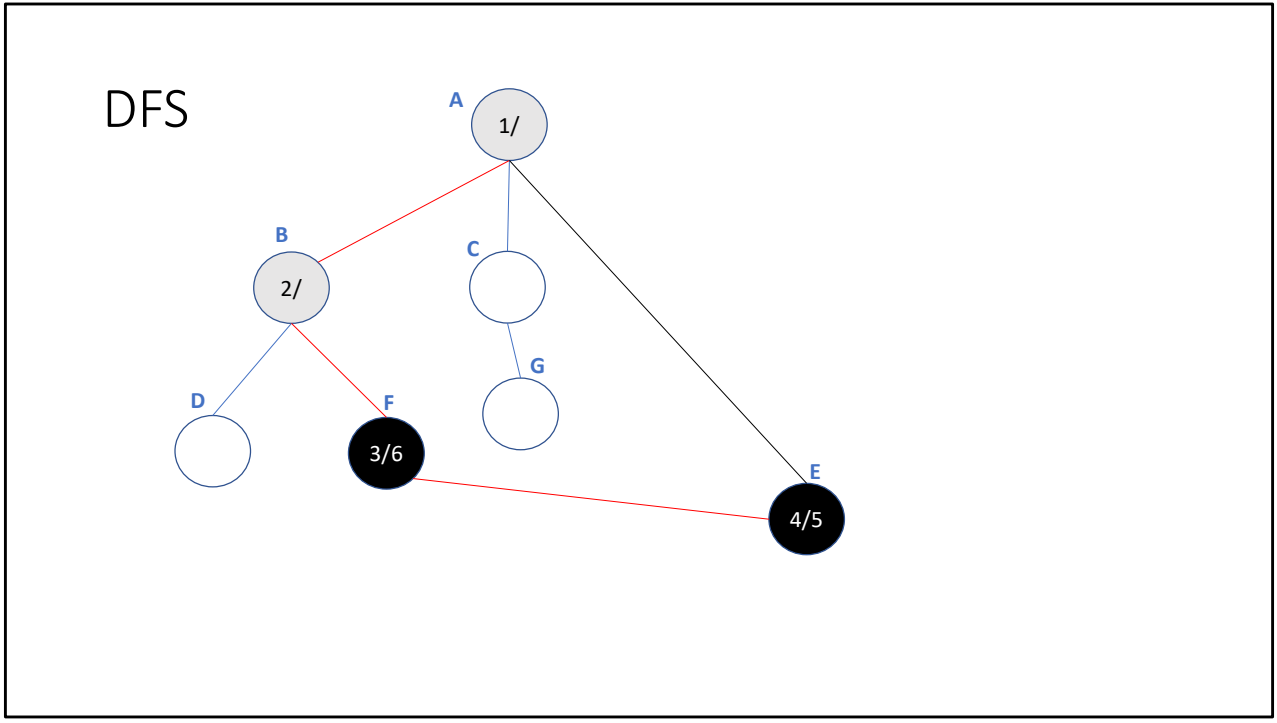
The algorithm has now picked E. The discovery time is set to 4. Now, the algorithm will explore the neighbors of E.



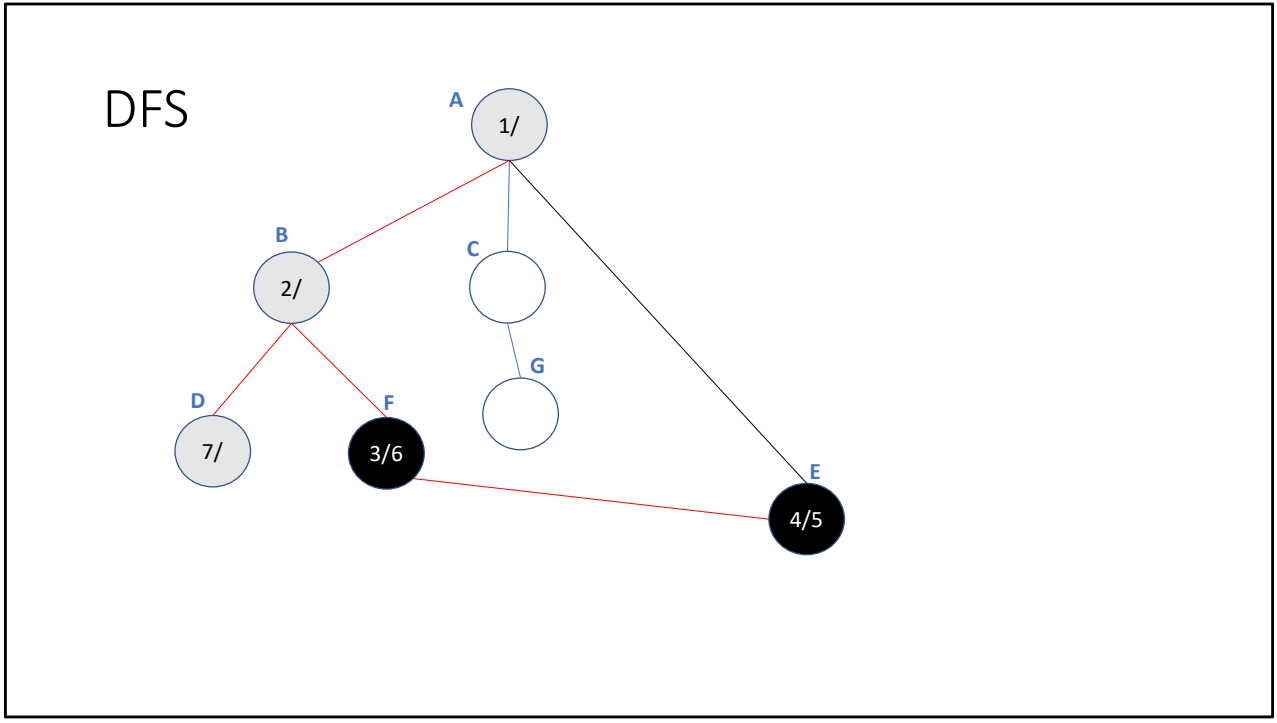
The neighbors of E (A and F) have already been visited. Edge (E, A) is not a tree edge because A has already been discovered. We have shown non-tree edges in black. The edges that are yet to be explored are blue. Now, we have reached a dead end, as there are no more vertices to explore. In that case, we can backtrack from E to a node from which we have reached E, i.e., F. We can maintain a stack or use the predecessor field in the vertex (similar to the one we maintained in BFS) for backtracking. However, a simpler approach would be to use recursion because it implicitly maintains a stack that can be used for backtracking. After backtracking from E, the DFS algorithm will track other undiscovered vertices of F. We mark a node as black just before the backtracking.



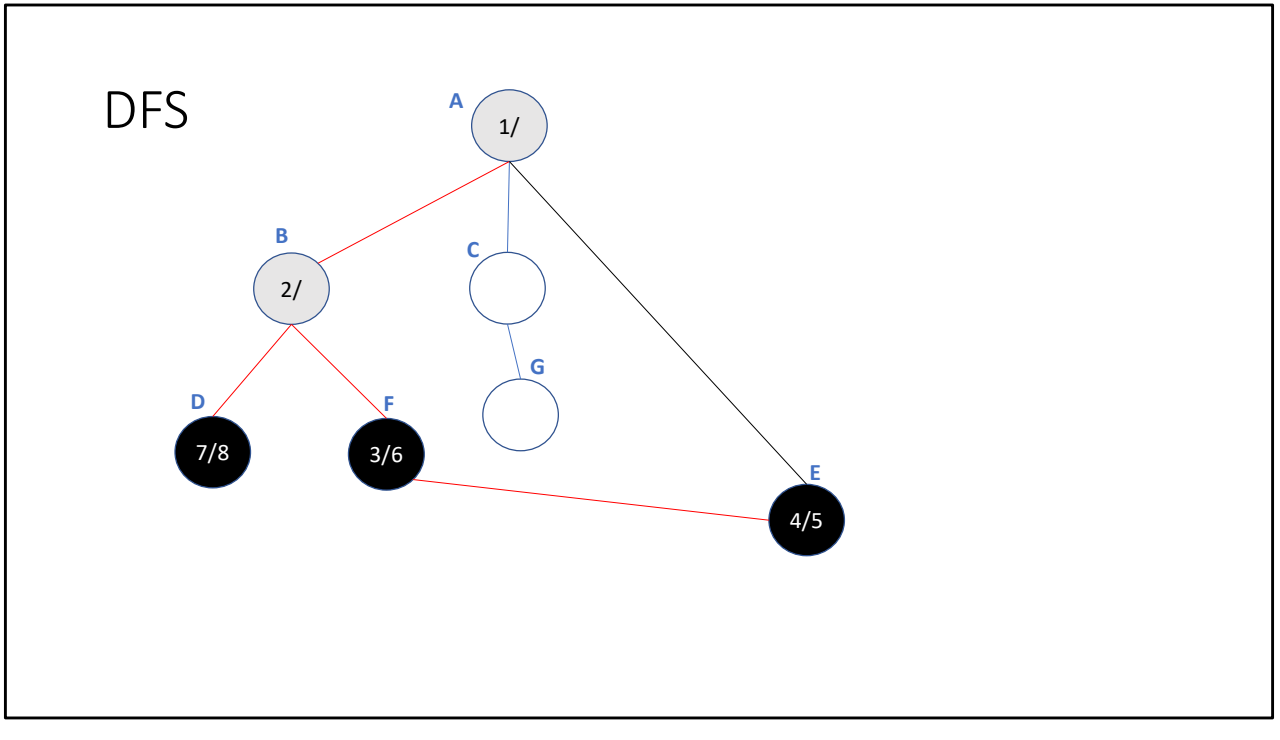
While marking a vertex as black, we also need to update the finish time of the vertex, which is set to the current counter value. The counter is incremented before backtracking to the predecessor.



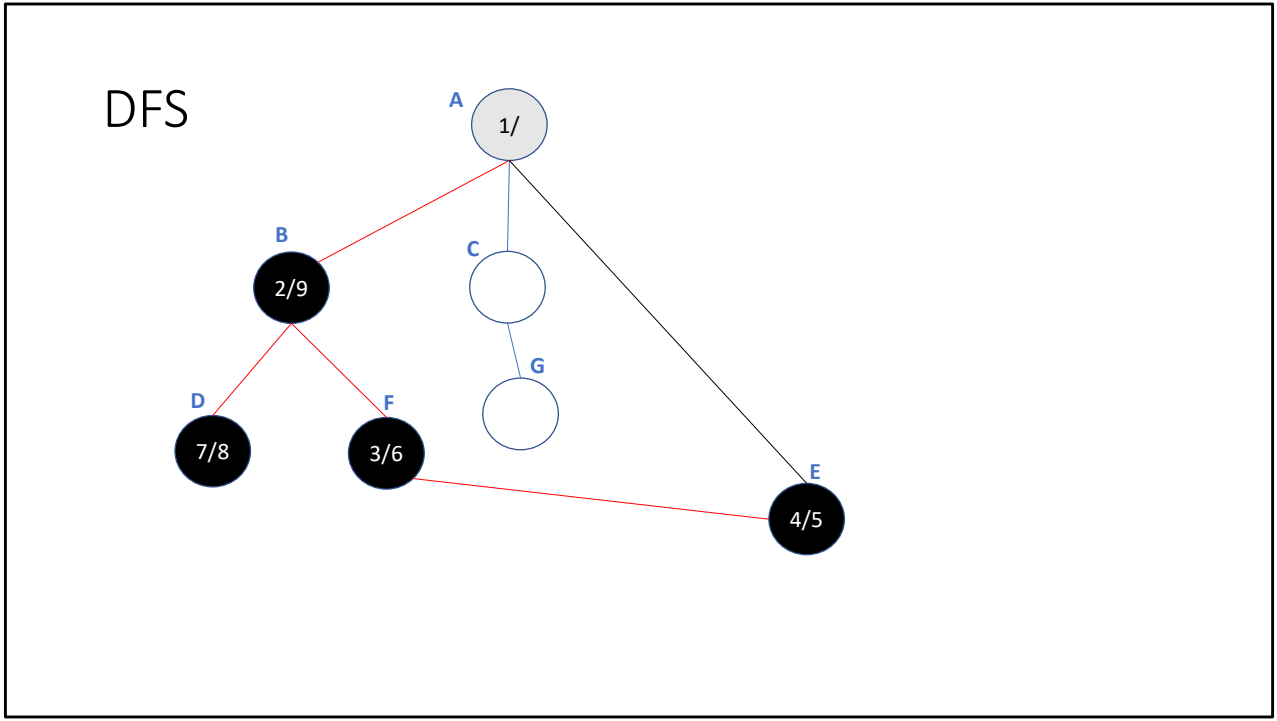
There are no other vertices to explore at F, so we updated the finish time and marked the vertex black. The DFS algorithm will now backtrack to B.



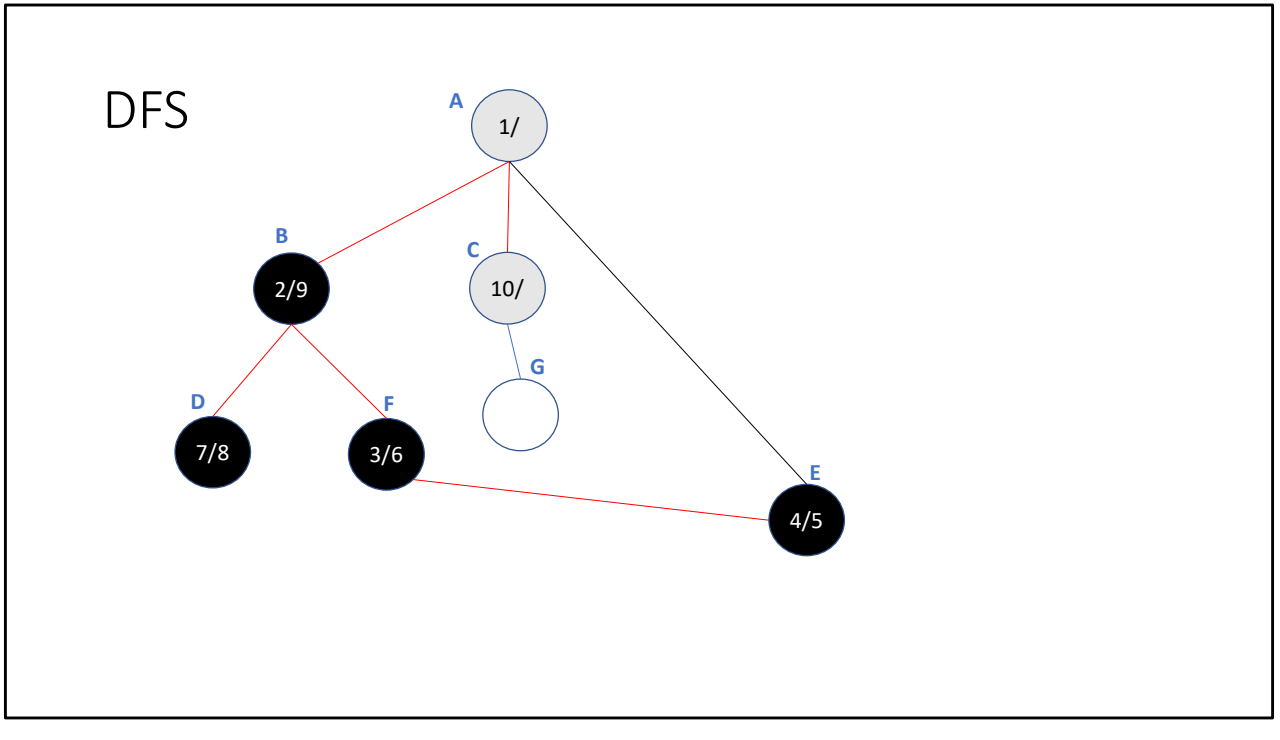
After backtracking, we reach vertex B. The vertex D is an unexplored neighbor of B. D is marked as gray.



There is no other vertex to explore at D. We mark D as black and backtrack to B.

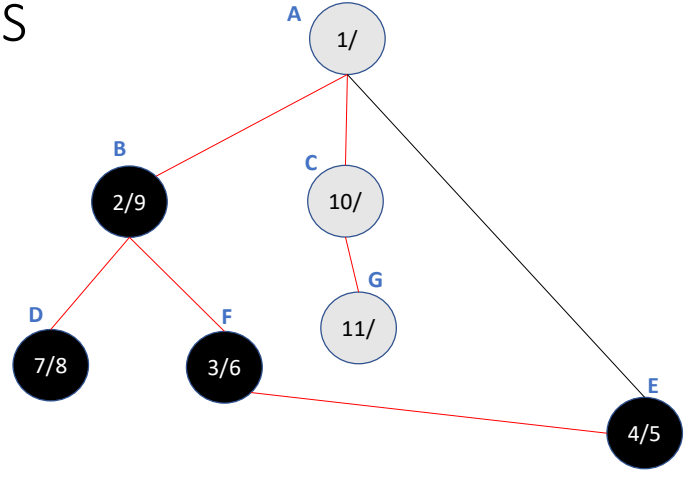


We need to backtrack from B as well because we have explored all undiscovered neighbors. After backtracking, we reach vertex A.



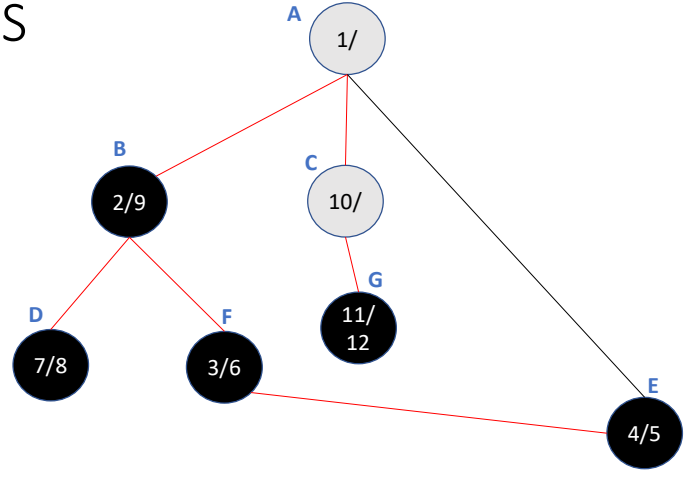
Neighbour C of A has not been explored yet. The DFS algorithm marks C as gray.

DFS

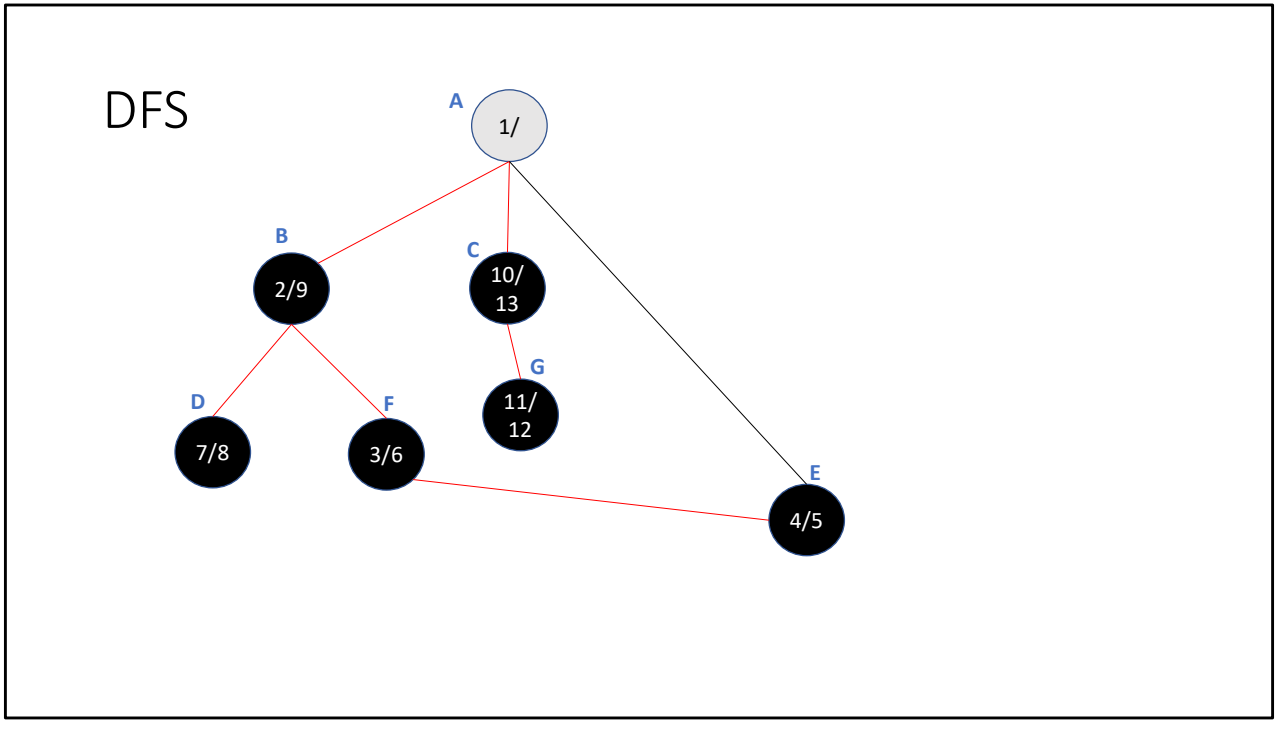


Neighbour G of C hasn’t been explored yet. The DFS algorithm marks G as gray.

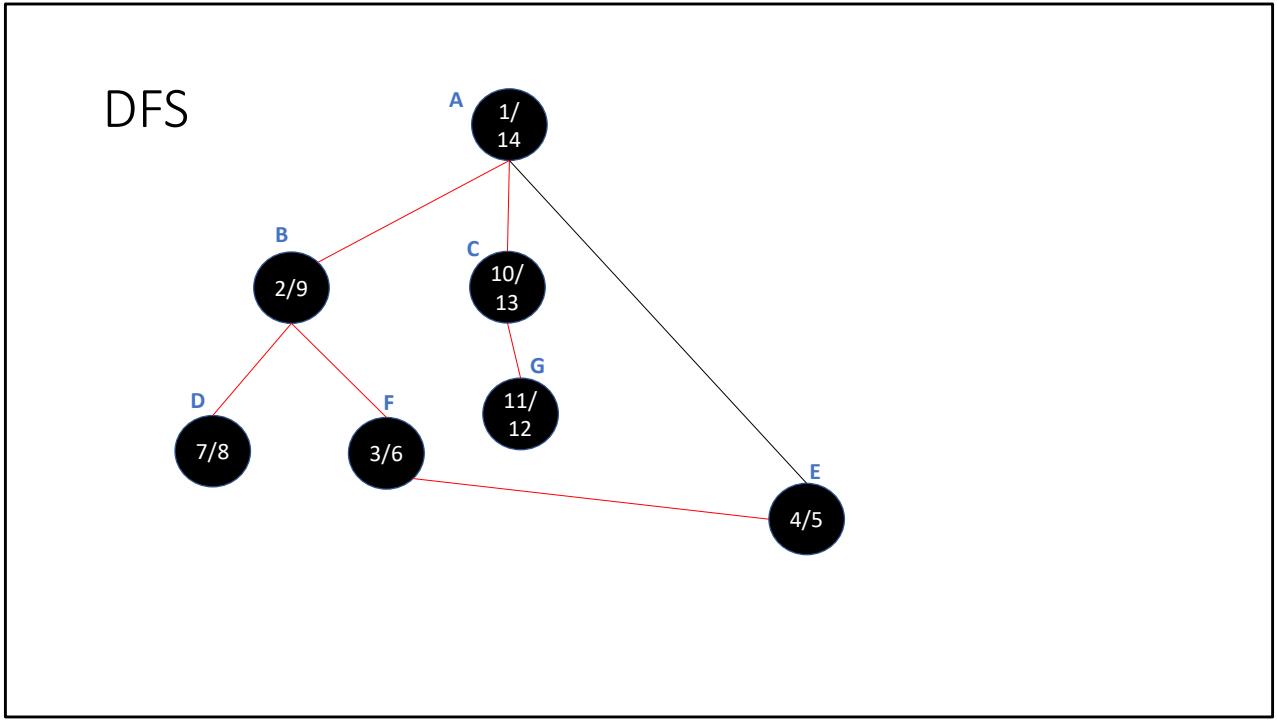
DFS



Backtracking from G to C because there is no other vertex to explore.



Backtracking from C to A.



There is nothing to backtrack at A. The algorithm stops at this point after marking A as black and updating its finish time.

DFS

```
DFS(G)

// G is a graph (V, E)
// s is the source vertex
// each vertex contains four
fields: color, d, f, π
// d is discovery time
// f is finish time
// π is predecessor in the DFS
forest

// Output: compute the discovery
time, finish time, and predecessor
in the DFS forest for each vertex
```

```
1. DFS(G)
2. for each vertex u ∈ G.V
3.     u.color = WHITE
4.     u.π = NIL
5. time = 0
6. for each vertex u ∈ G.V
7.     if u.color == WHITE
8.        DFS-VISIT(G, u)

9. DFS-VISIT(G, u)
10.time = time + 1
11.u.d = time
12.u.color = GRAY
13.for each vertex v ∈ G.Adj[u]
14.    if v.color == WHITE
15.       v.π = u
16.       DFS-VISIT(G, v)
17.time = time + 1
18.u.f = time
19.u.color = BLACK
```

Time complexity

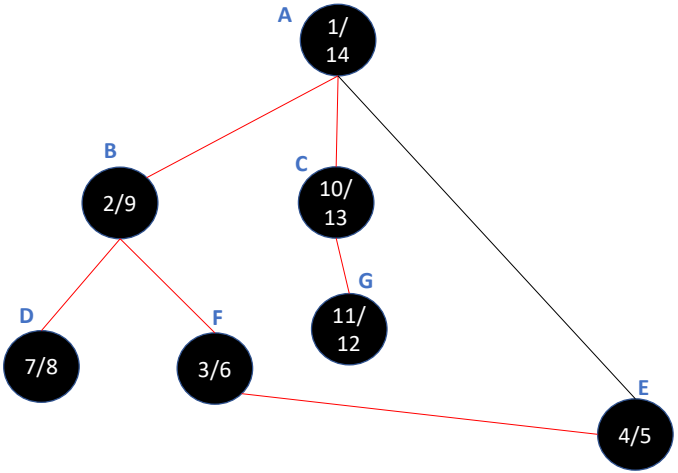
$O(|V| + |E|)$

The justification is given on the next slide.

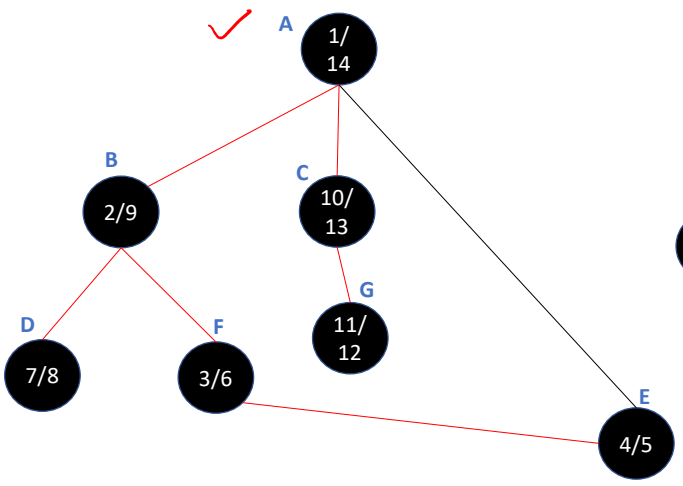
Time complexity

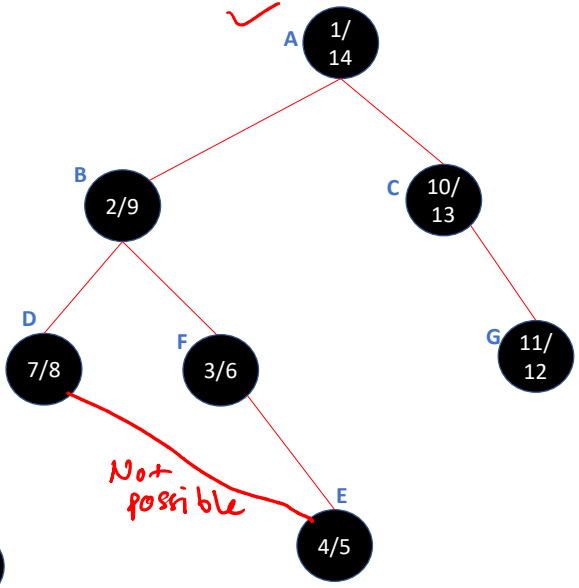
- DFS-visit is called at line-8,16 for every white vertex
- The vertex is marked gray at the start of DFS-visit at line-12
- A vertex never gets whitened after it changes its color
- Therefore DFS-visit is called |V| times, one for each vertex
- At line-13, the adjacency list of a vertex is traversed
  - Therefore, line-13 executes the sum of the degree of all vertices time, which is O(|E|)
  - The for loop body at lines 14-16 does a constant number of operations
- Therefore, the time complexity is O(|V|+|E|)

DFS-tree

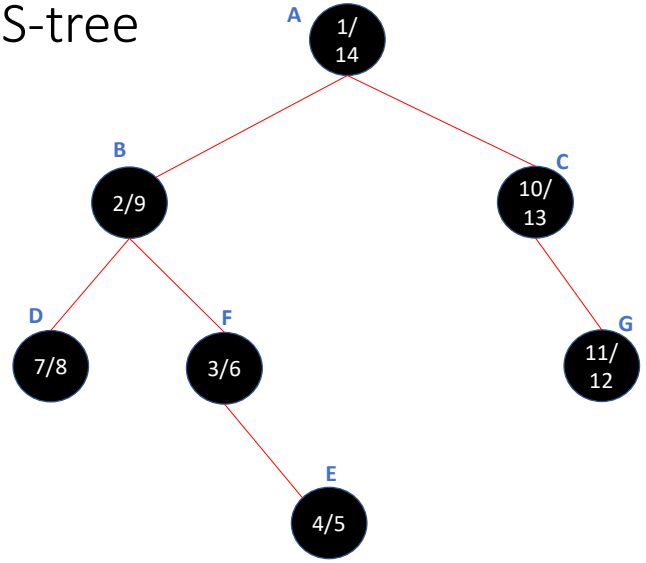


DFS-tree


A
1/
14
B
2/9
C
10/
13
D
7/8
F
3/6
G
11/
12
E
4/5

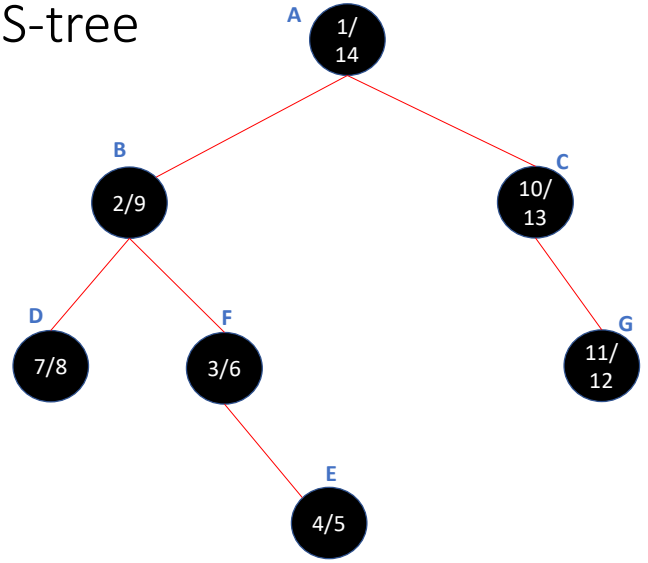

A
1/
14
B
2/9
C
10/
13
D
7/8
F
3/6
G
11/
12
Not
possible
E
4/5

DFS-tree



Why do discovery edges form a tree?

DFS-tree

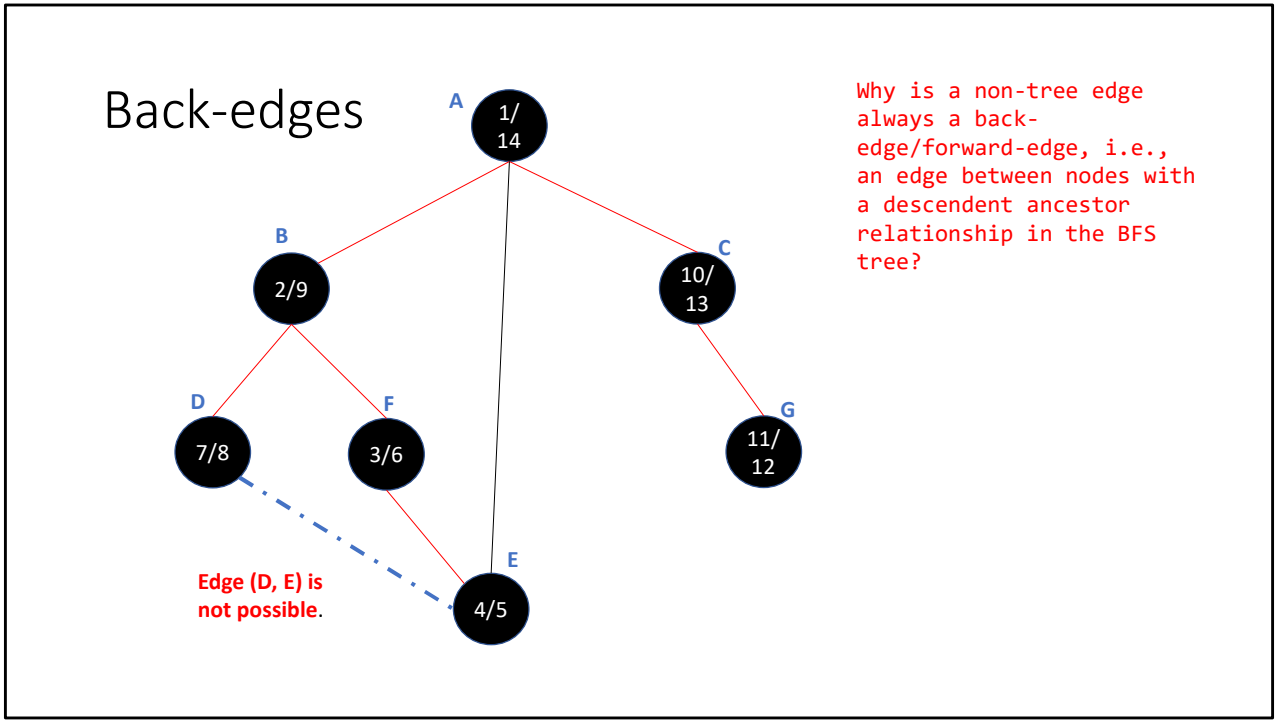


Why do discovery edges form a tree?

A vertex is discovered only once; therefore, each vertex has a single predecessor except the source vertex, which doesn't have any predecessor. Thus the number of edges is n-1.

There is a path from each vertex to the source vertex. Therefore, all vertices are connected.

Hence, it is a tree.



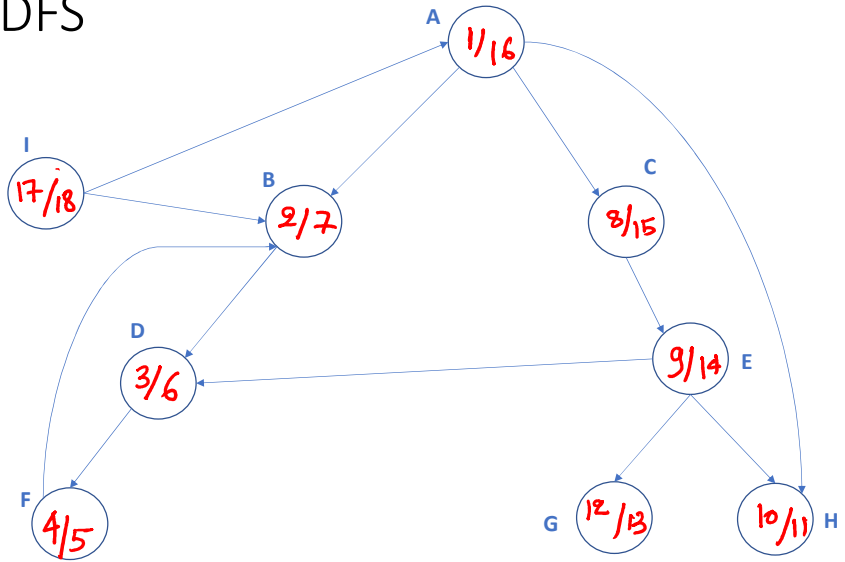
There can't be a non-tree edge between nodes that don't have an ancestor/descendent relationship. Suppose there is an edge between D and F (D and F don't have an ancestor/descendent relationship). When we arrive at D, only the ancestors of D can be gray. Therefore, E is either black or white. If E is white, the DFS algorithm will explore vertex E right now; thus, edge (D, E) will become a tree edge. Let's say E is black. In this case, when the DFS algorithm reaches E, D must have been white because D has not been blackened yet, and only an ancestor can be gray. Therefore, the DFS algorithm must have traversed edge (E, D) then, making it a tree edge. Therefore, non-tree edges are not possible for edges whose endpoints don't have an ancestor-descendent relationship.

DFS on directed graph

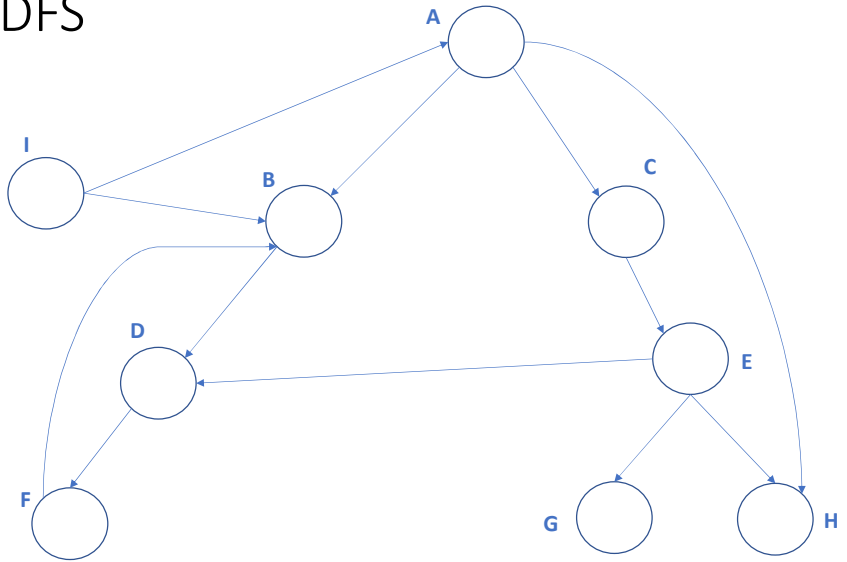
DFS on a directed graph

- The algorithm for a directed graph is the same as an undirected graph
- The tree generated during DFS on a directed graph for a given vertex v generates a directed DFS tree
  - In a directed DFS tree, there is a path from v to all other vertices that are reachable via v; however, the opposite is not true

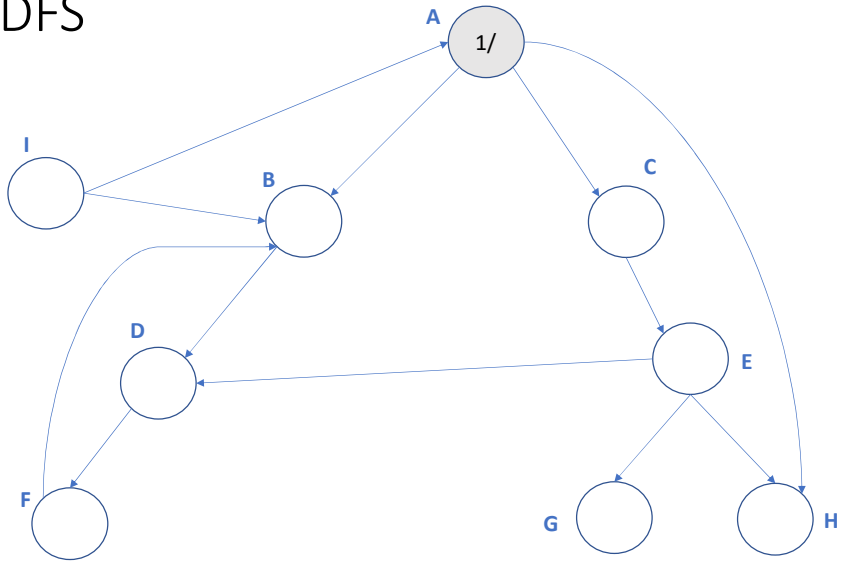
DFS



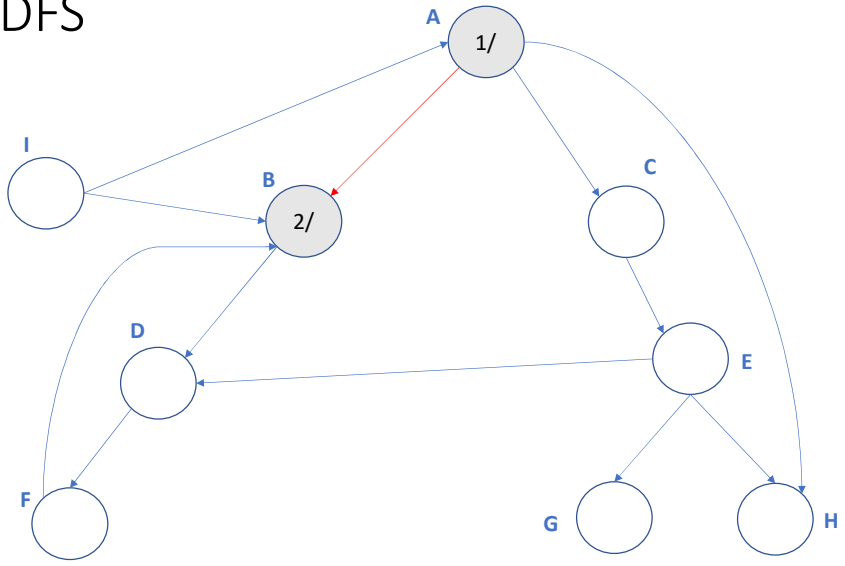
DFS



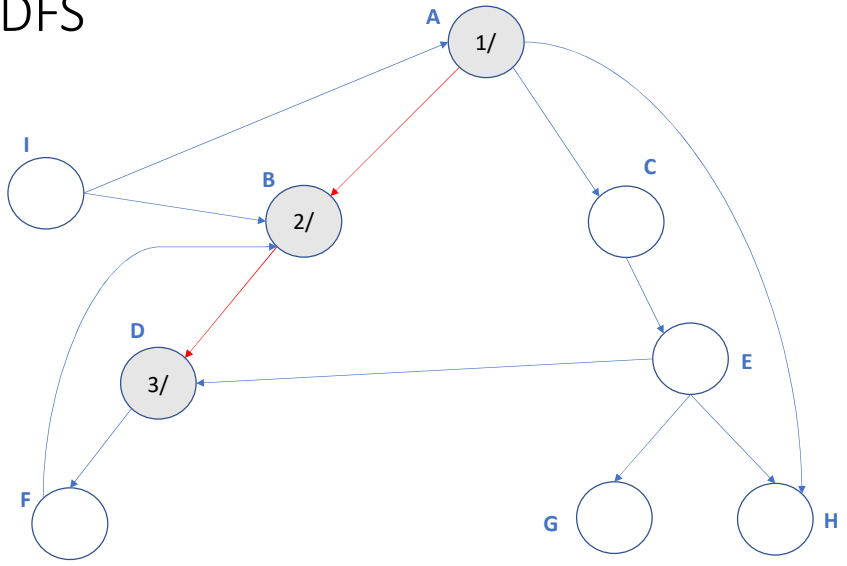
DFS



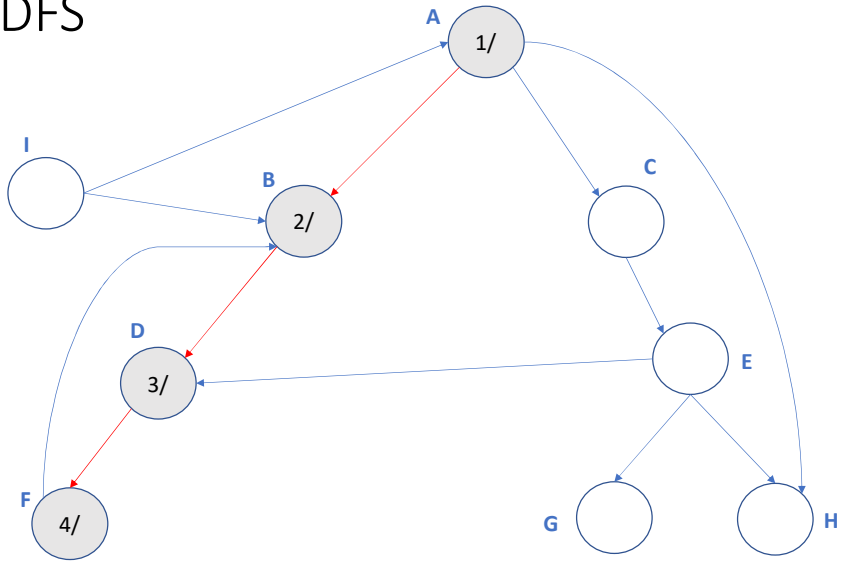
DFS



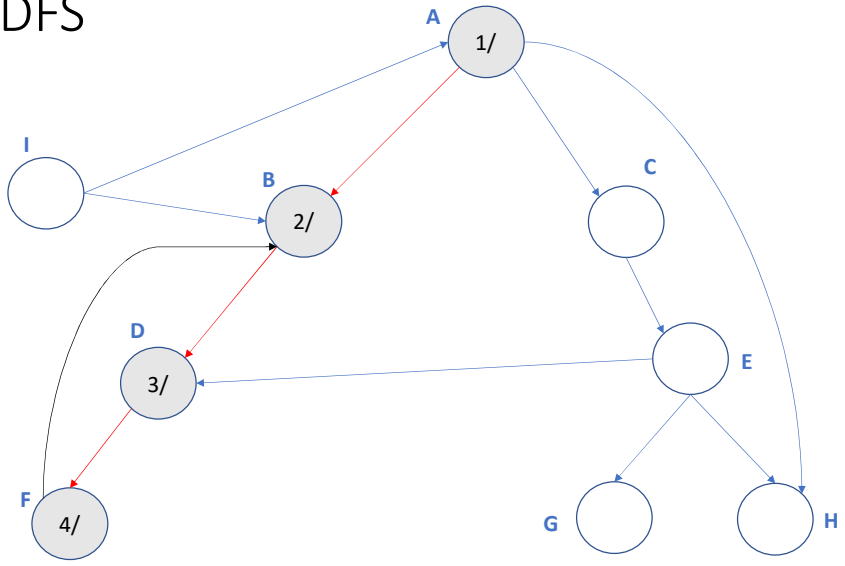
DFS



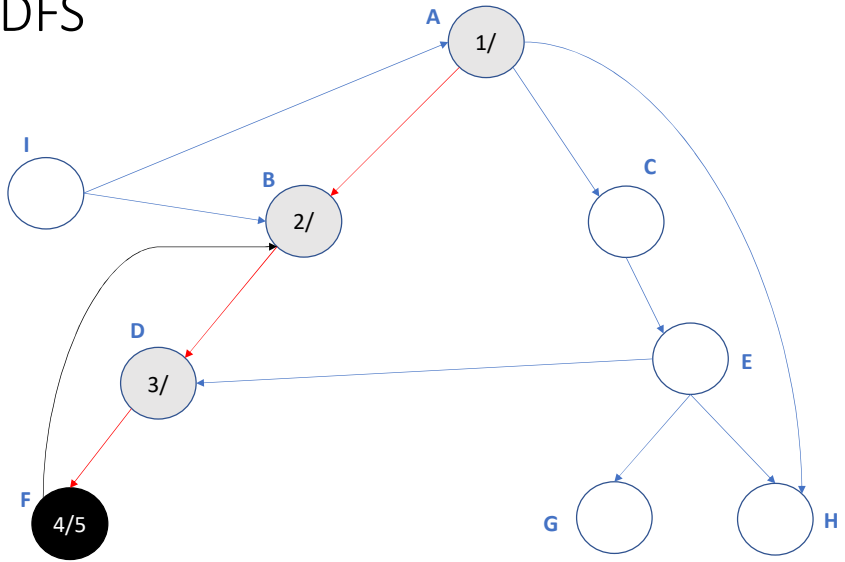
DFS



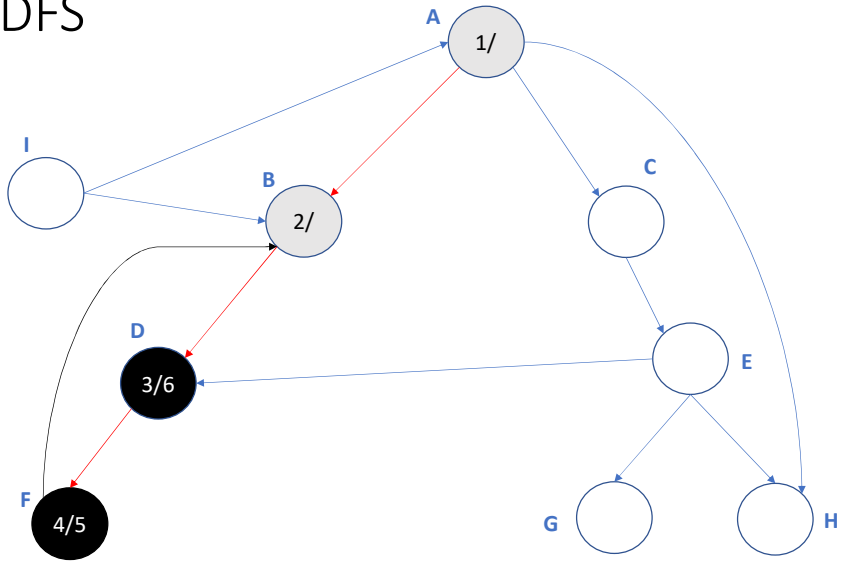
DFS



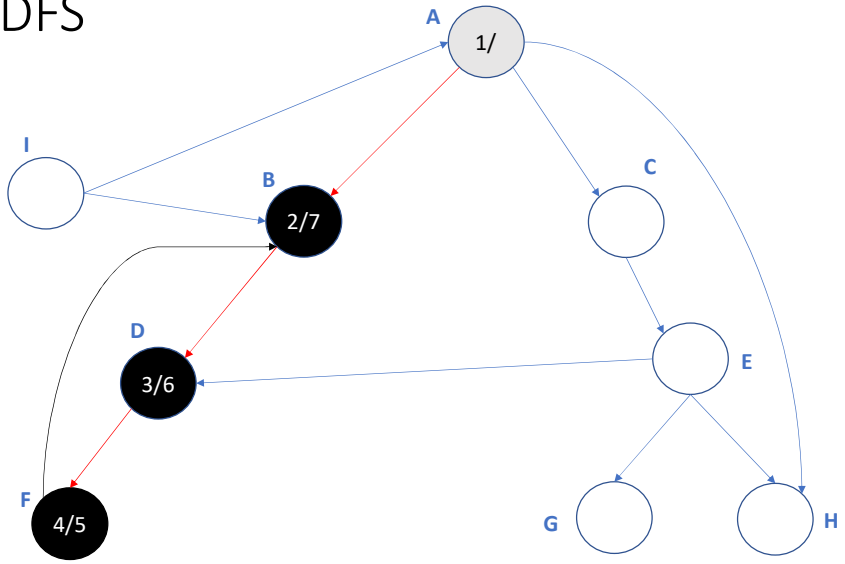
DFS



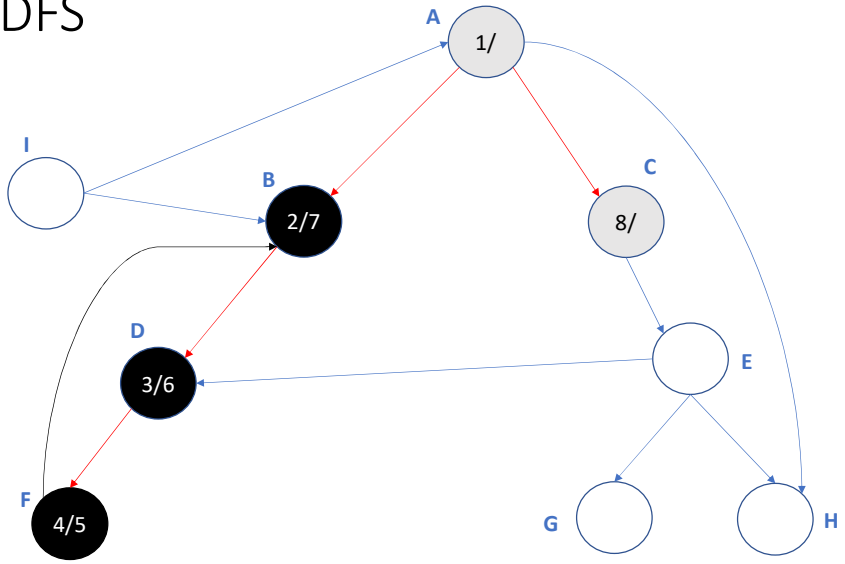
DFS



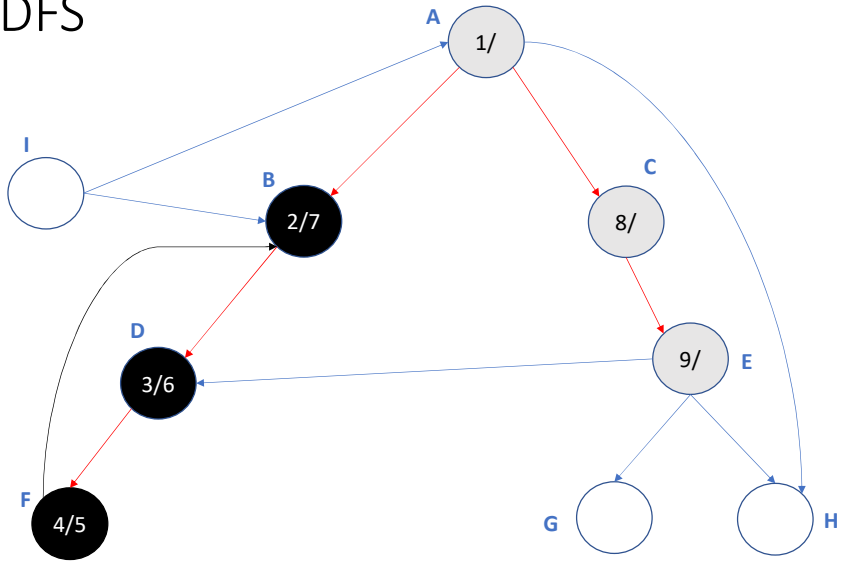
DFS



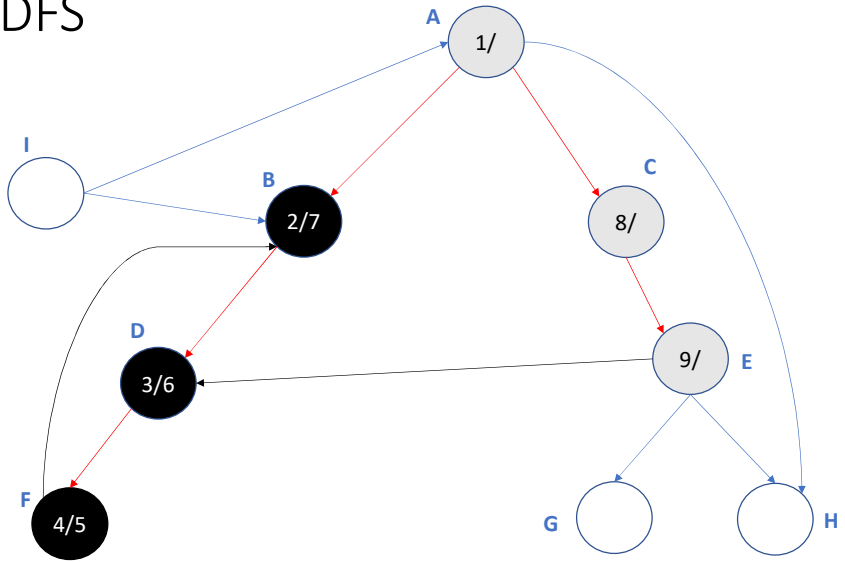
DFS



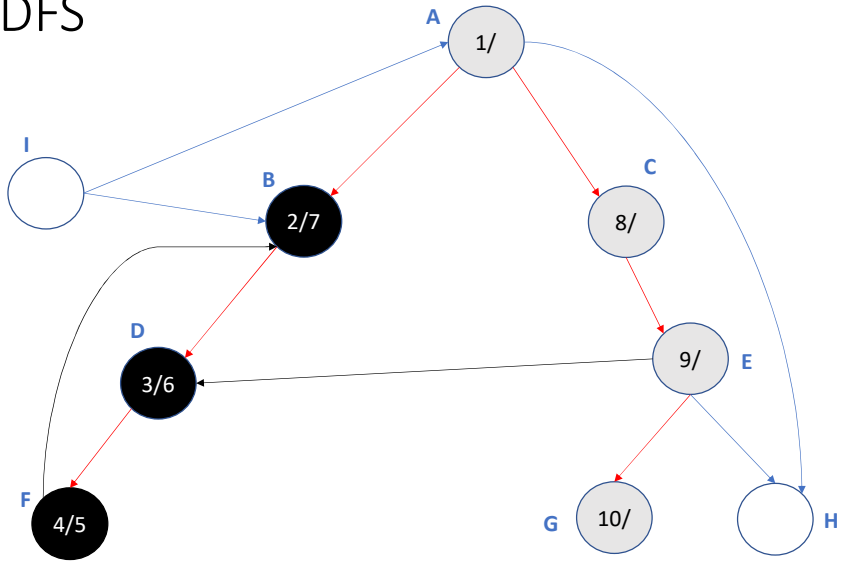
DFS



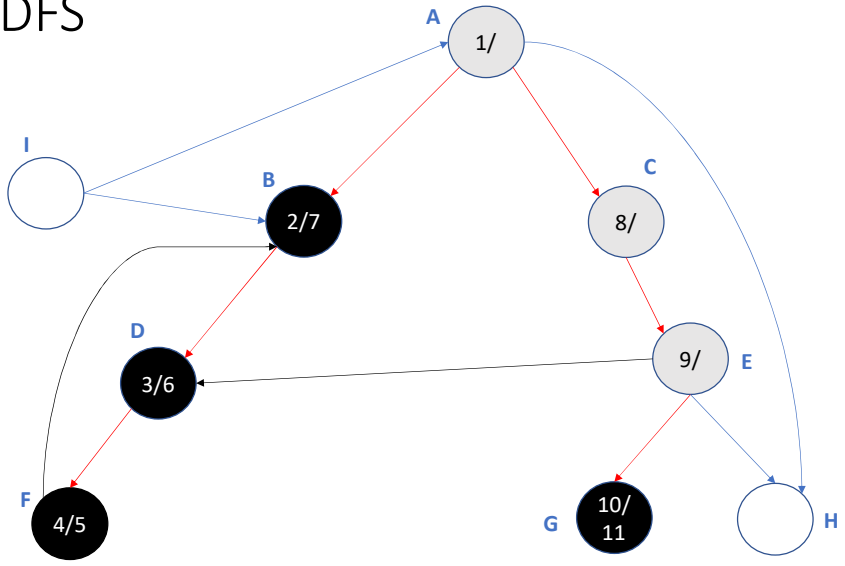
DFS



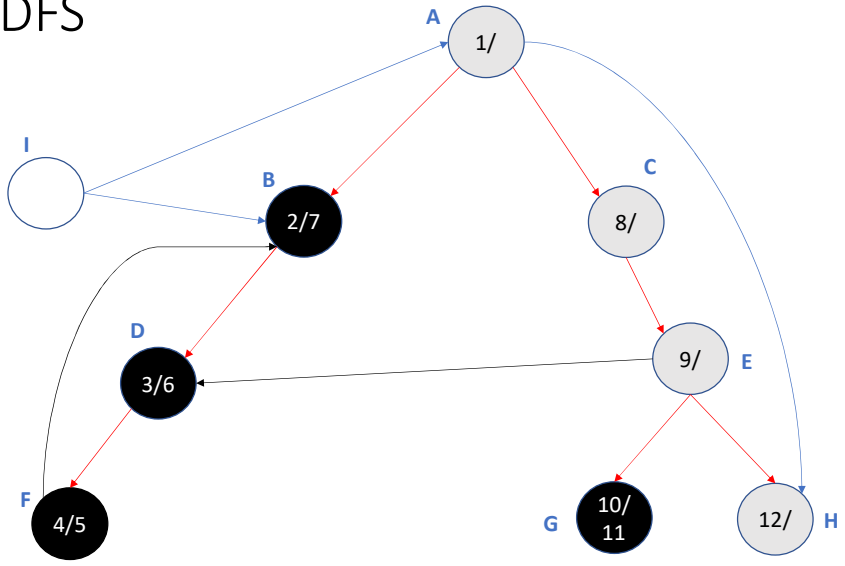
DFS



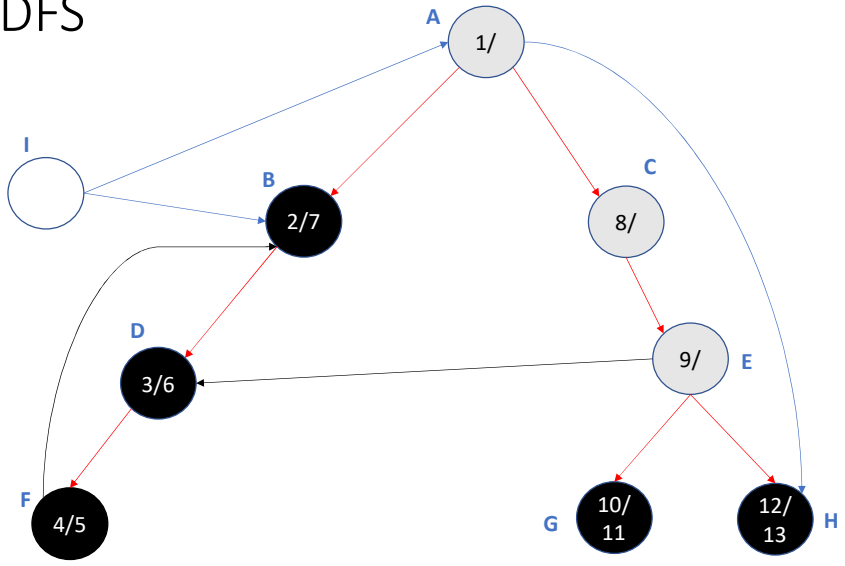
DFS



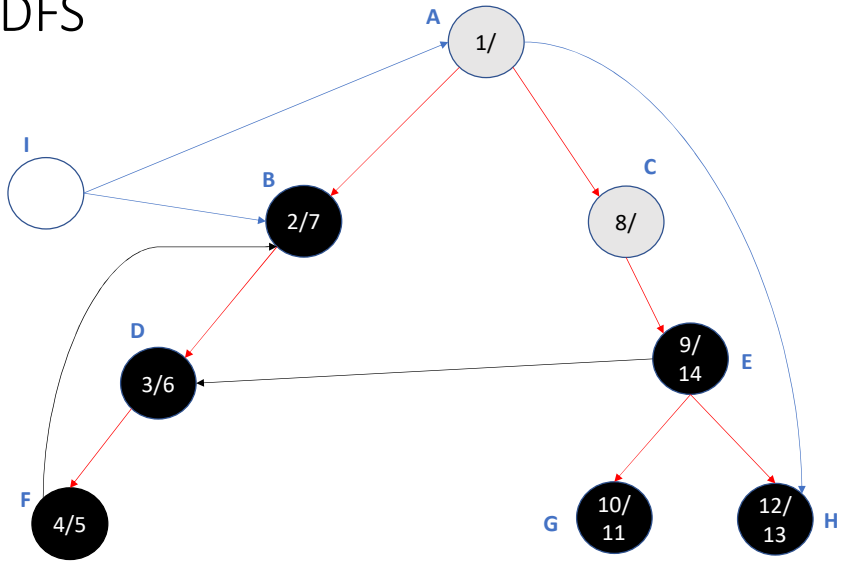
DFS



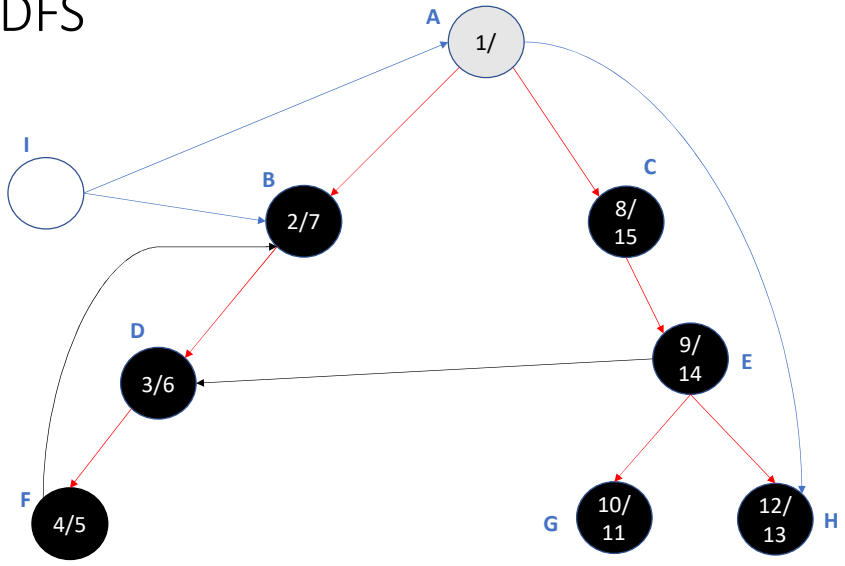
DFS



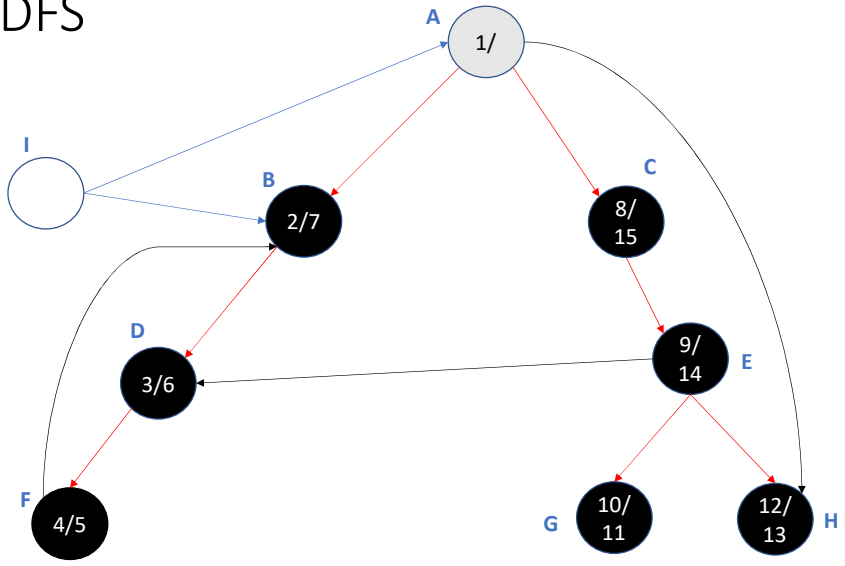
DFS



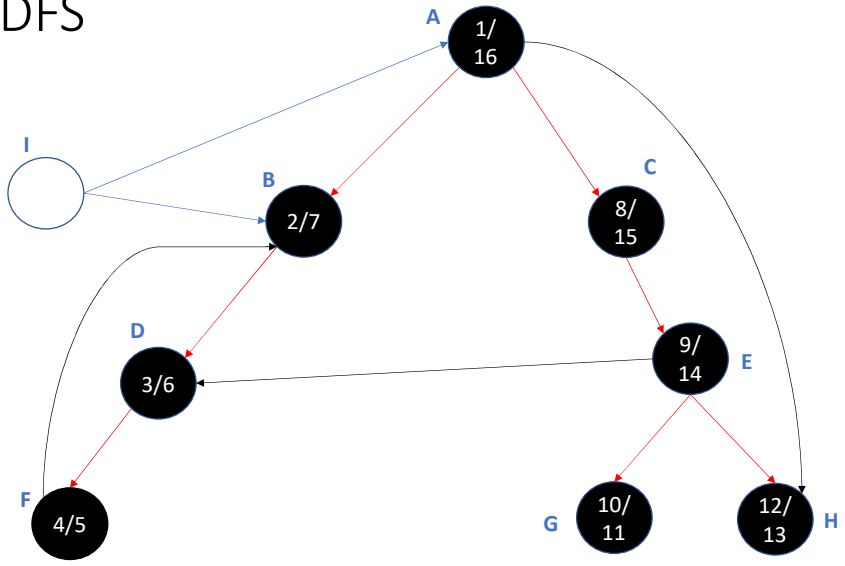
DFS



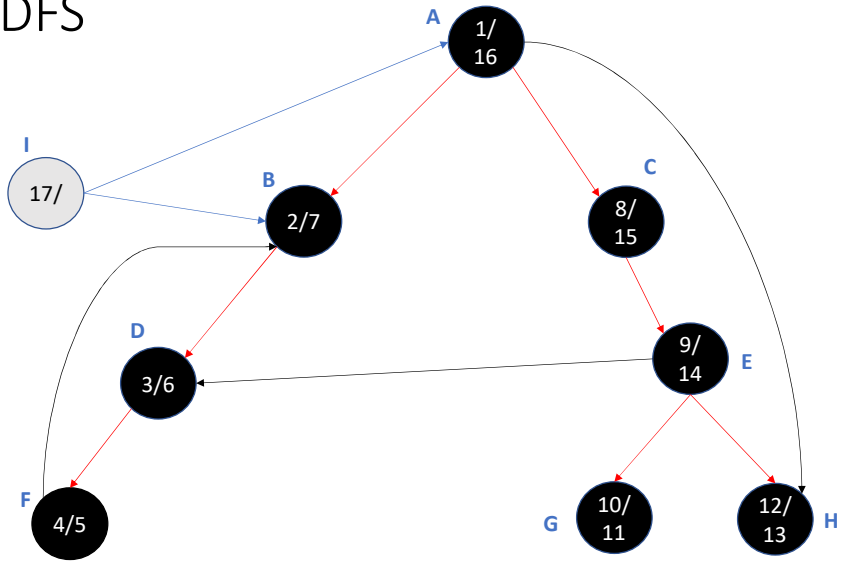
DFS



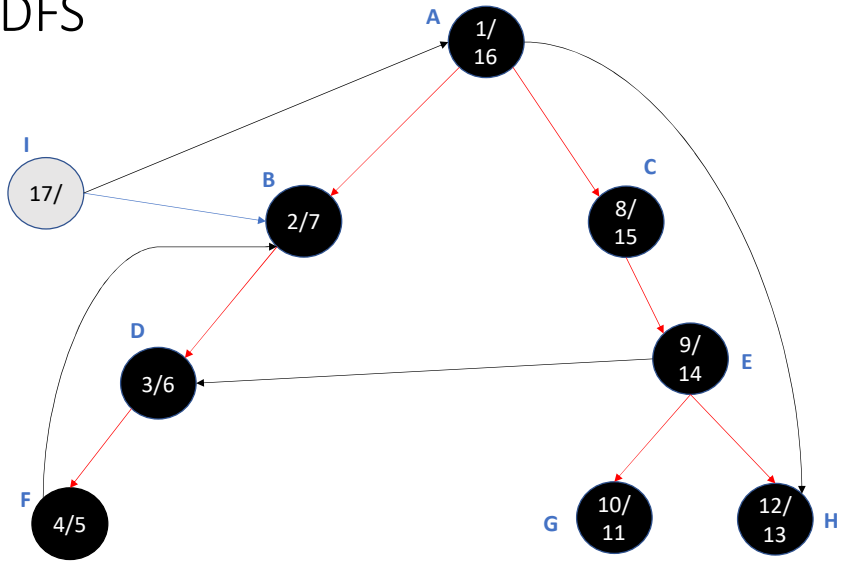
DFS



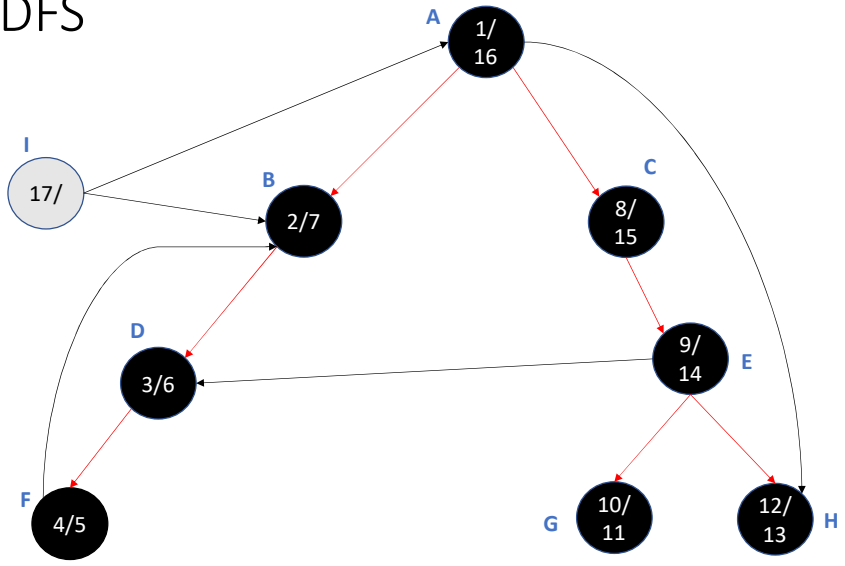
DFS



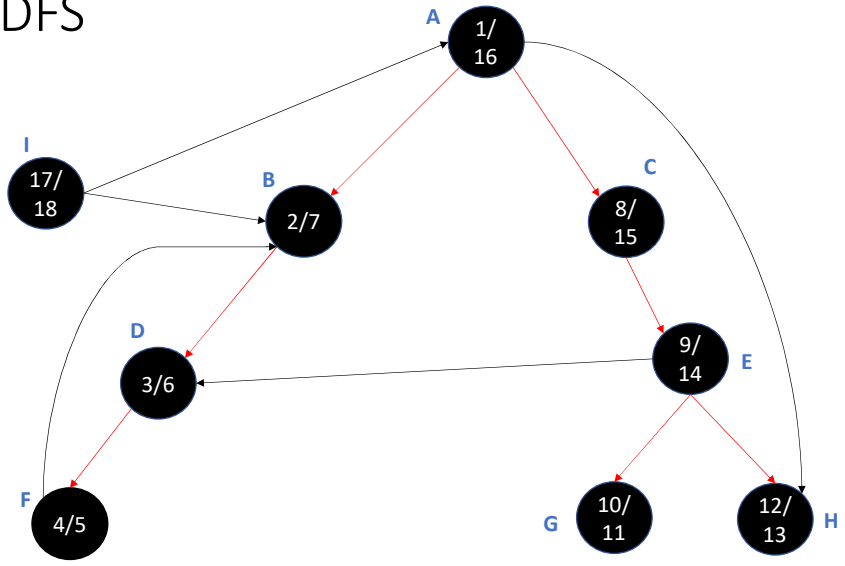
DFS



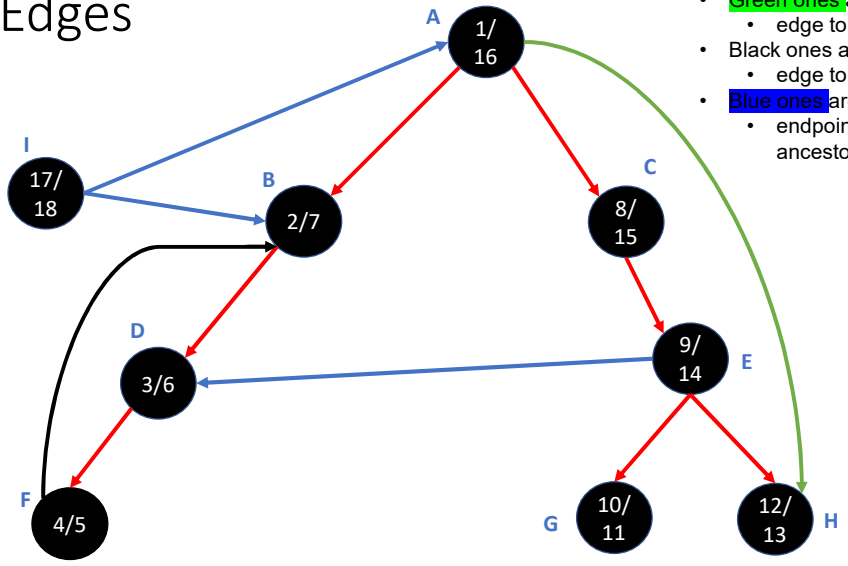
DFS



DFS



Edges

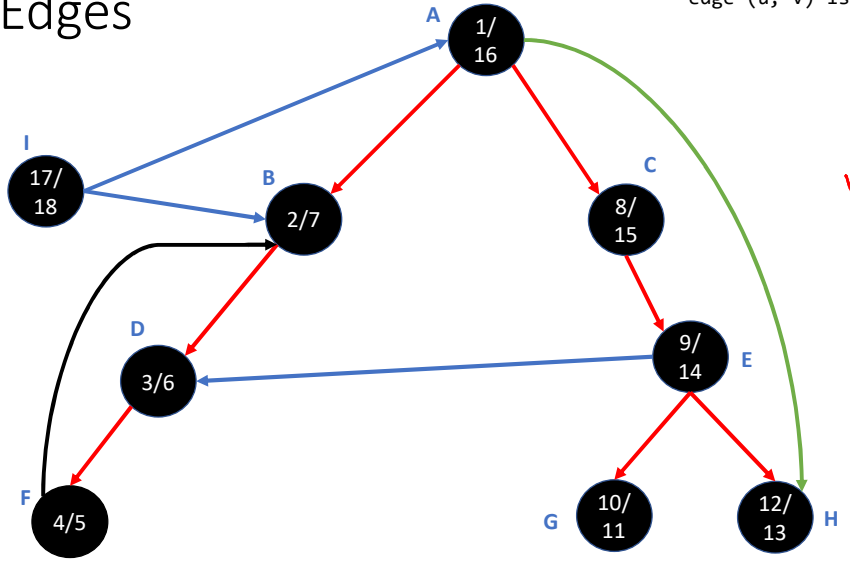


- Edge classification
  - Red ones are **tree edges**
  - Green ones are forward edges
    - edge to a descendent node in the tree
  - Black ones are **back edges**
    - edge to an ancestor node in the tree
  - Blue ones are **cross edges**
    - endpoints don't have ancestor/descendent relationship

Edges

for vertex v, v.d is the discovery time
for vertex v, v.f is the finish time
edge (u, v) is a **back edge** if

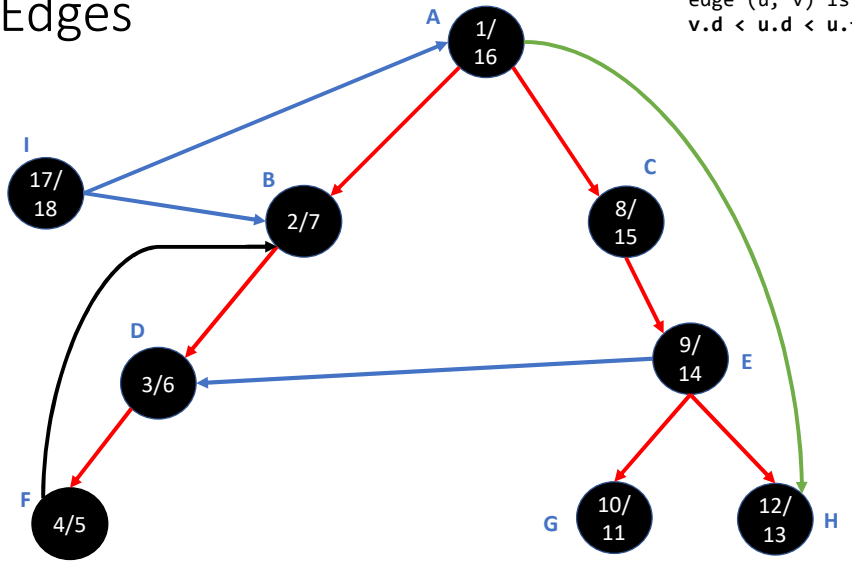
$v.d < u.d < u.f < v.f$

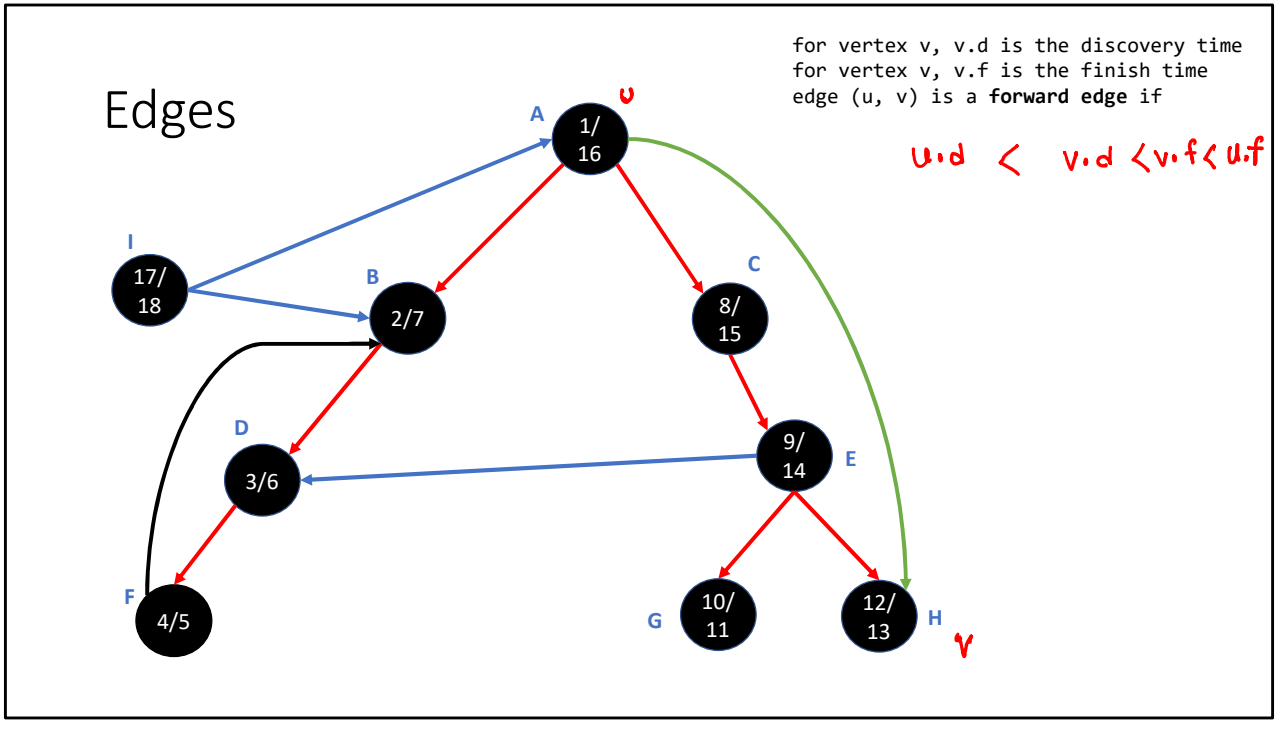


This is the condition for a back edge.

Edges

for vertex v, v.d is the discovery time
for vertex v, v.f is the finish time
edge (u, v) is a **back edge** if
v.d < u.d < u.f < v.f

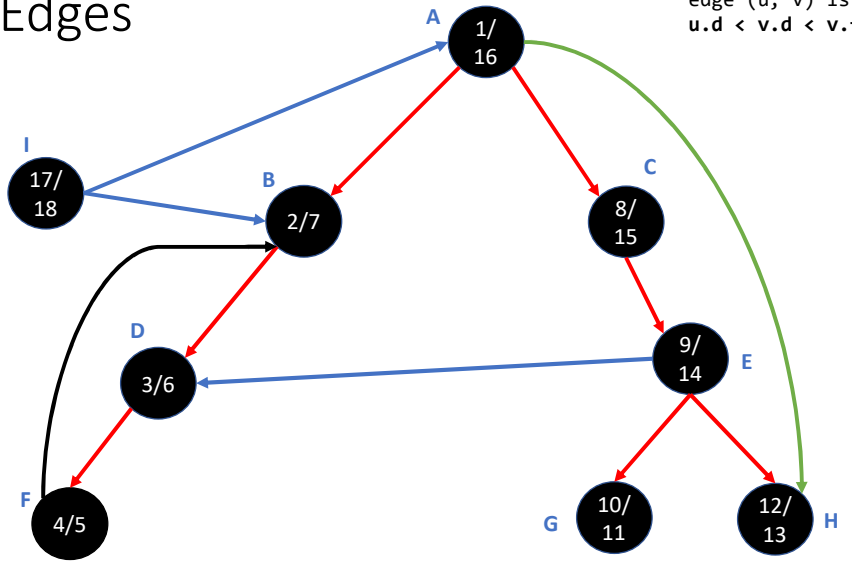




This is the condition for a forward edge.

Edges

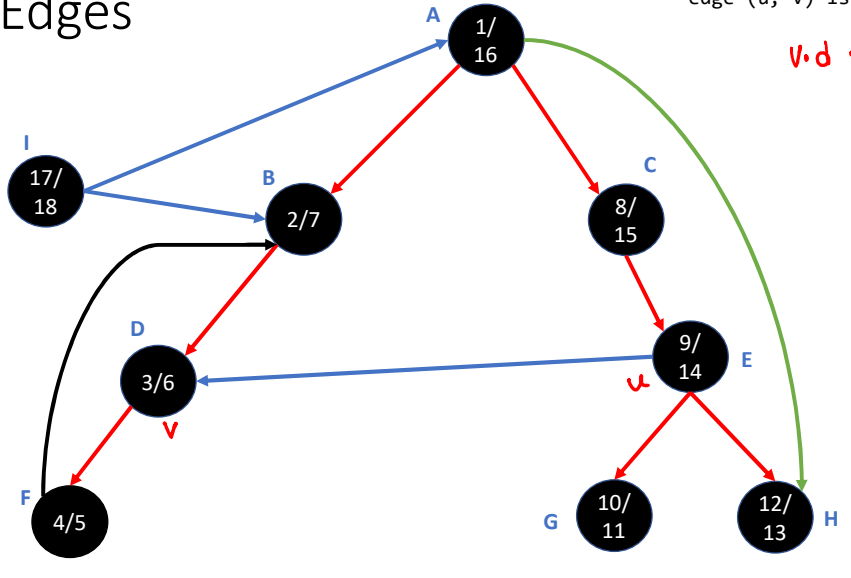
for vertex v, v.d is the discovery time
for vertex v, v.f is the finish time
edge (u, v) is a **forward edge** if
u.d < v.d < v.f < u.f



Edges

for vertex v, v.d is the discovery time
for vertex v, v.f is the finish time
edge (u, v) is a **cross edge** if

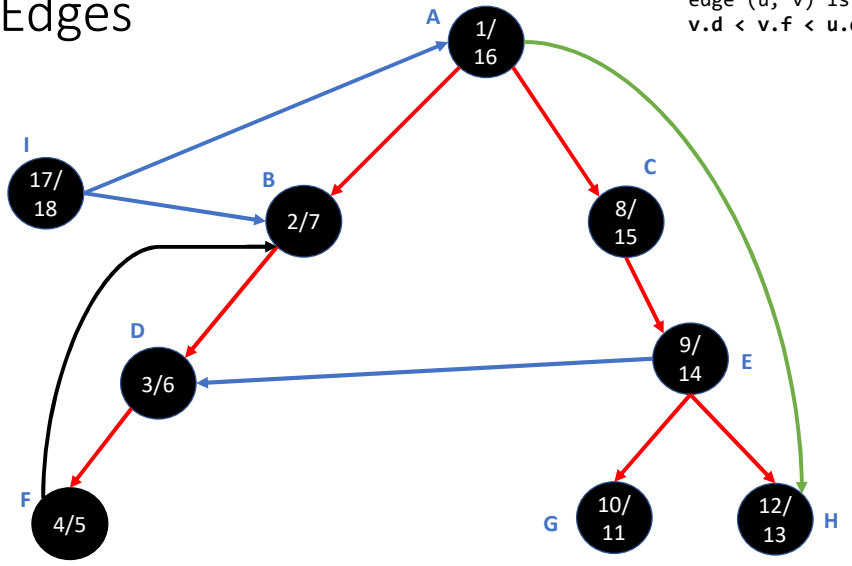
$v.d < v.f < u.d < u.f$



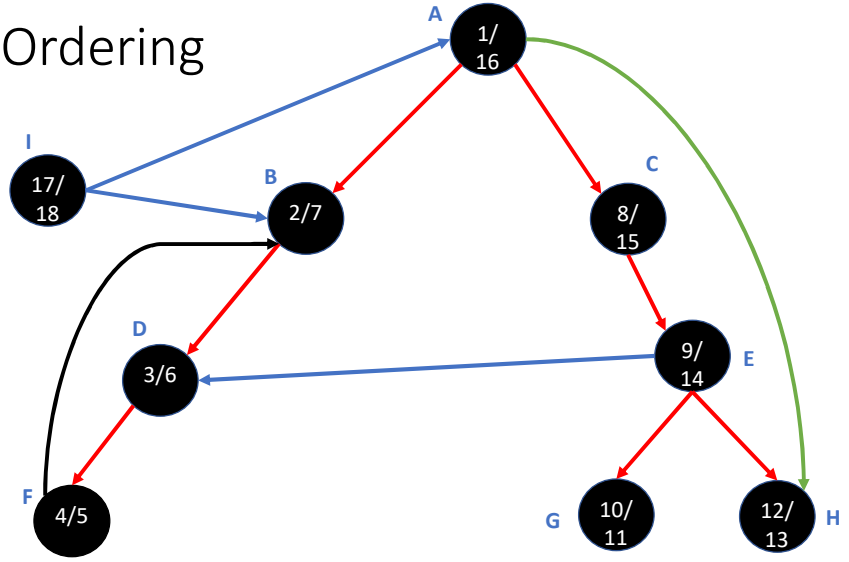
This is the condition for a cross edge.

Edges

for vertex v, v.d is the discovery time
for vertex v, v.f is the finish time
edge (u, v) is a **cross edge** if
v.d < v.f < u.d < u.f



Ordering

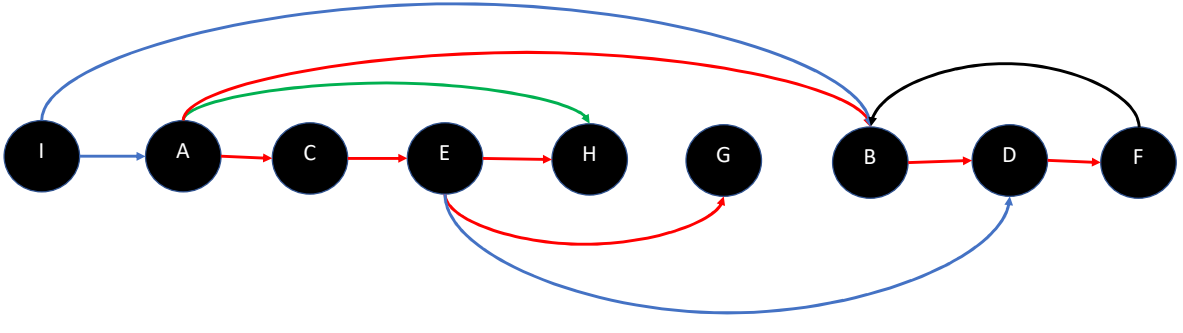


Ordering

- Arrange vertices in the reverse order in which they are blackened

Ordering

- Arrange vertices in the reverse order in which they are blackened



The explanation is given on the following slides.

Ordering

- Let's arrange vertices in the reverse order in which they are blackened

- In the DFS tree, a descendent node always finishes (gets blackened) before its ancestor node, therefore
  - A tree edge (ancestor to descendent) will always be from left to right
  - A forward edge (ancestor to descendent) will always be from left to right

Ordering

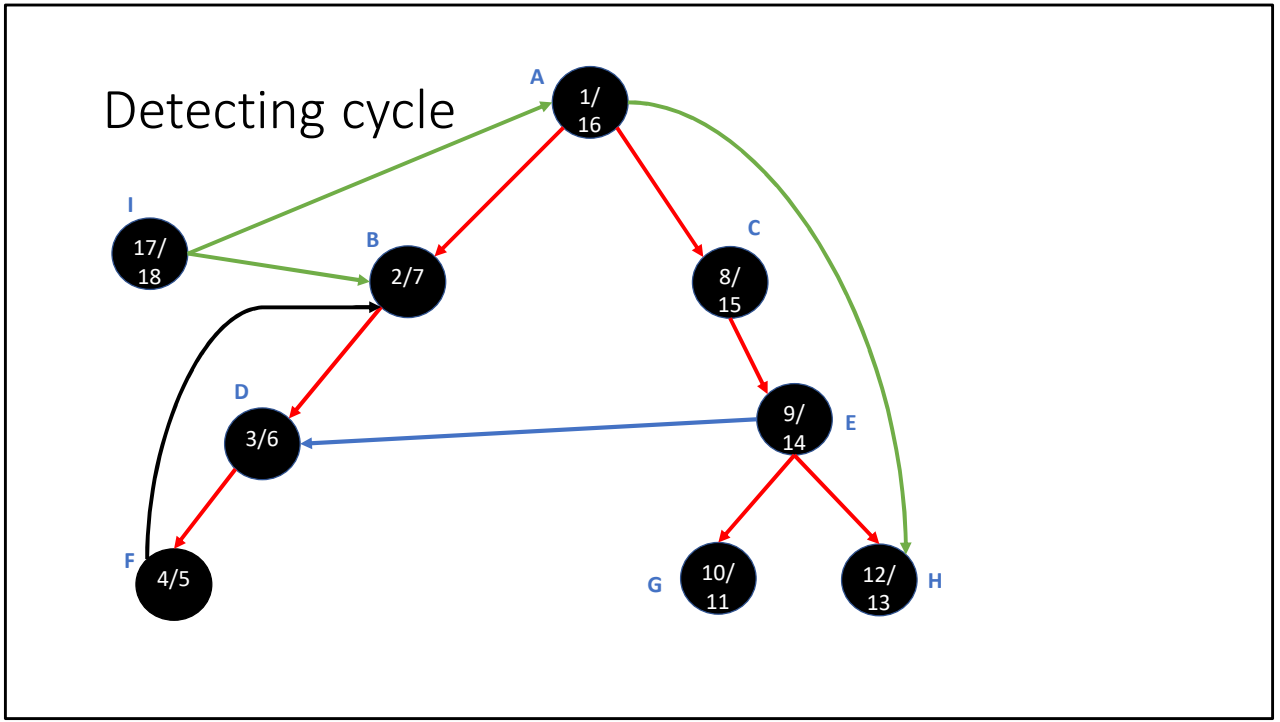
- For a cross edge **(u, v),** there is no ancestor/descendent relationship between the **u** and **v**, therefore
  - **v** must be black
    - If **v** is gray, then it is one of the ancestors
    - If **v** is white, then **v** is discovered for the first time; thus, it is a tree edge
  - Consequently, **v** is blackened before **u**
  - It means that the cross edge will also be from left to right

- For a back edge **(u, v)**, the destination vertex is an ancestor in the tree; therefore, the back edge will be from right to left
  - Because **u** is reachable from all ancestor vertices, including **v**, and **v** is reachable via **u** using the back edge, therefore a back edge must create a cycle

Ordering

- If we redraw the graph, which has only tree edges, forward edges, and cross edges in the reverse order in which they are blackened, then all the edges will be from left to right, and therefore there will be no cycle

Detecting cycle in directed graph



A back edge in a directed graph indicated the presence of a cycle. A back edge is between a descendent and an ancestor in the DFS tree. The ancestor is marked black only when all its descendants have been marked black. When we encounter a back edge (u, v), where u is the descendent of v, it means u hasn't been blackened yet, and so it v. Thus, to check if an edge is a back edge, we just need to check the color of the destination vertex. If the color of the destination vertex is gray, it means it's a back edge.

Detecting cycle

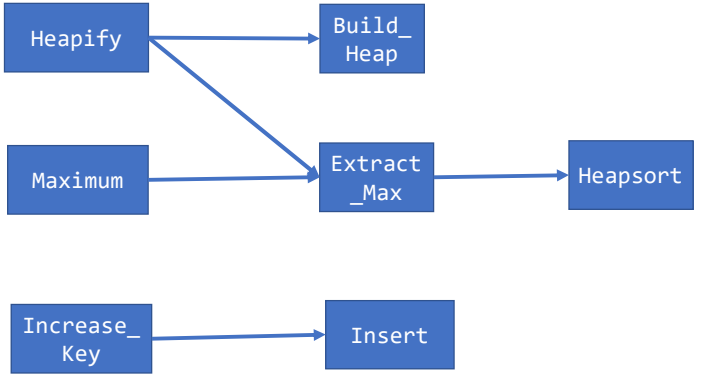
- During DFS, if we encounter a destination vertex that is neither white (tree edge) nor black (cross or forward edge), then the graph has a cycle

Topological sort

Example

- Build_Max_Heap
- Max_Heap_Extract_Max
- Max_Heap_Insert
- Max_Heap_Maximum
- Max_Heapify
- Heapsort
- Max_Heap_Increase_Key

Example



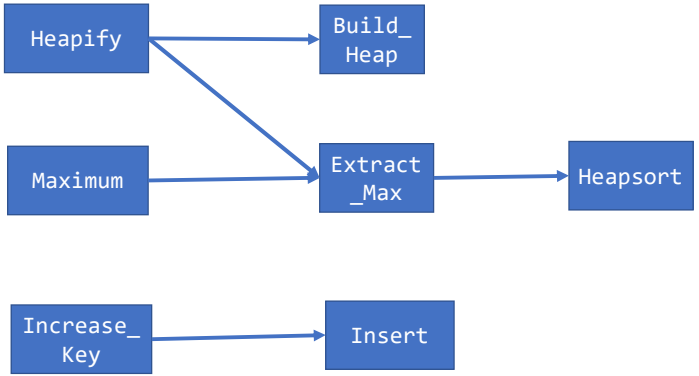
Is this a valid topological order?

Heapify, Build_Heap, Extract_Max, Heapsort, Maximum, Increase_Key, Insert

No

An edge represents dependency. In this example, Build_Heap implementation depends on Heapify. Similarly, Heapsort implementation depends on Extract_Max, and so on. If we can order vertices so that if a vertex v comes before another vertex u in the ordering means v must doesn't depend on u, then it's a topologically sorted order.

Example

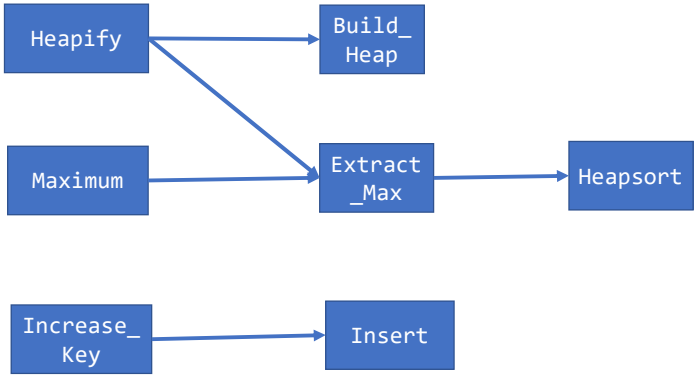


Is this a valid topological order?

Heapify, Maximum, Build_Heap, Extract_Max, Heapsort, Increase_Key, Insert

yes

Example



Is this a valid topological order?

Maximum, Heapify, Build_Heap, Extract_Max, Heapsort, Increase_Key, Insert

Yes