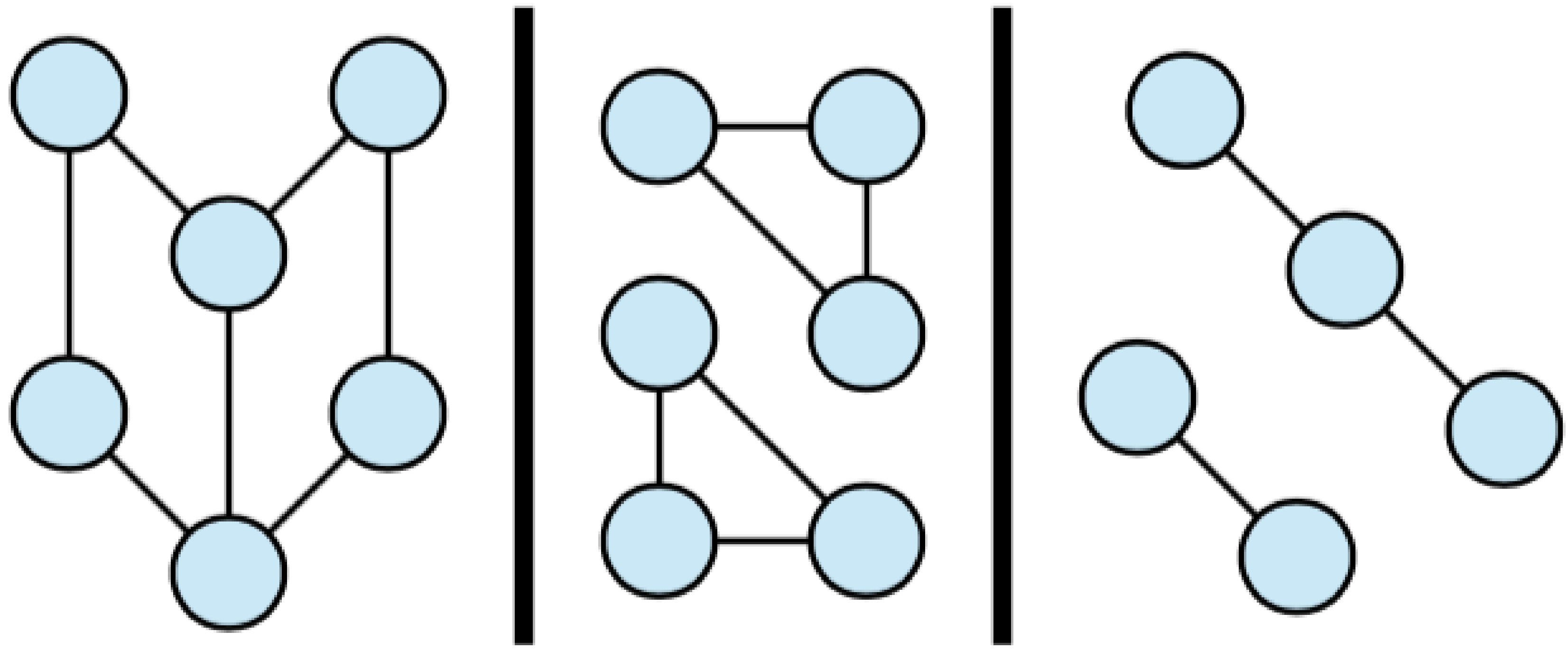
Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Kazan, Russia

Connected Component
Graph Traversals

Connected Component

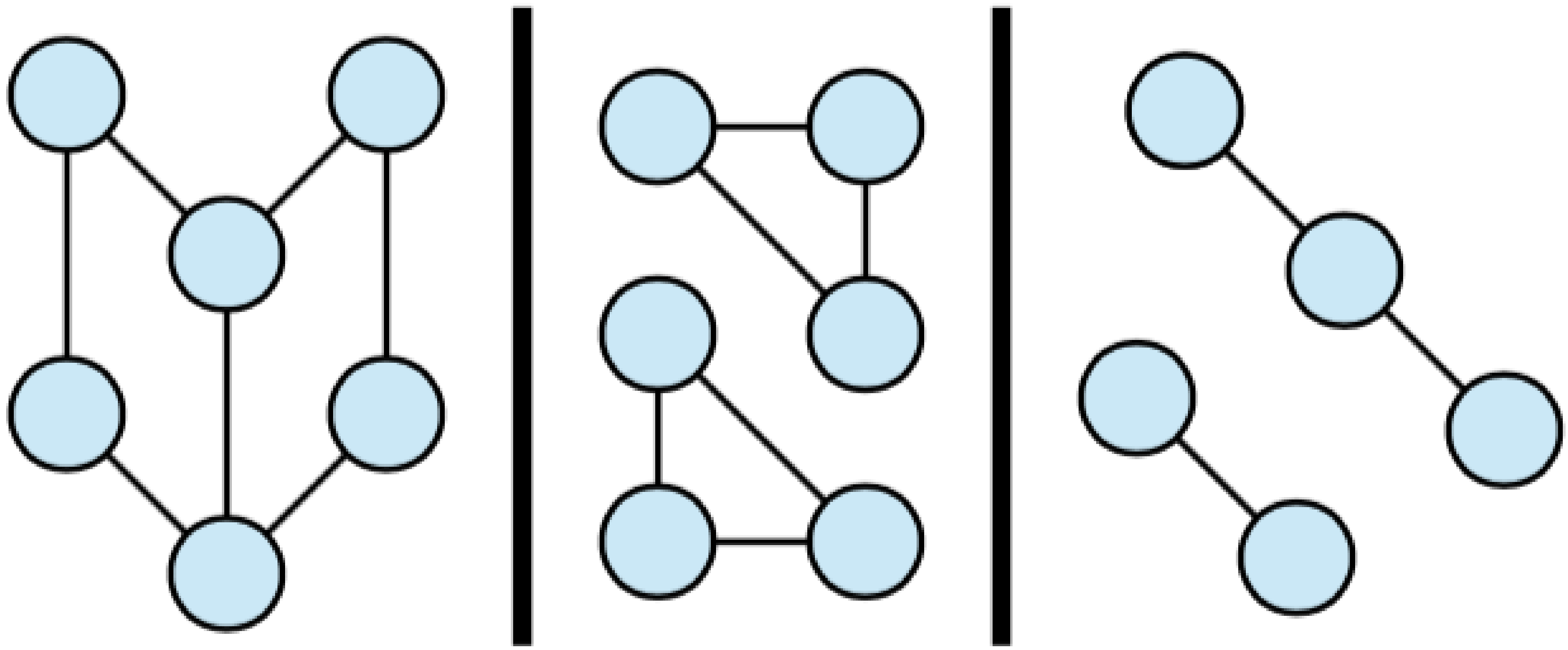
- Let G be an undirected graph.
- Two nodes u and v are called connected if there is a path from u to v in G ($u \longleftrightarrow v$)
- Now consider the following graphs



G1 G2 G3

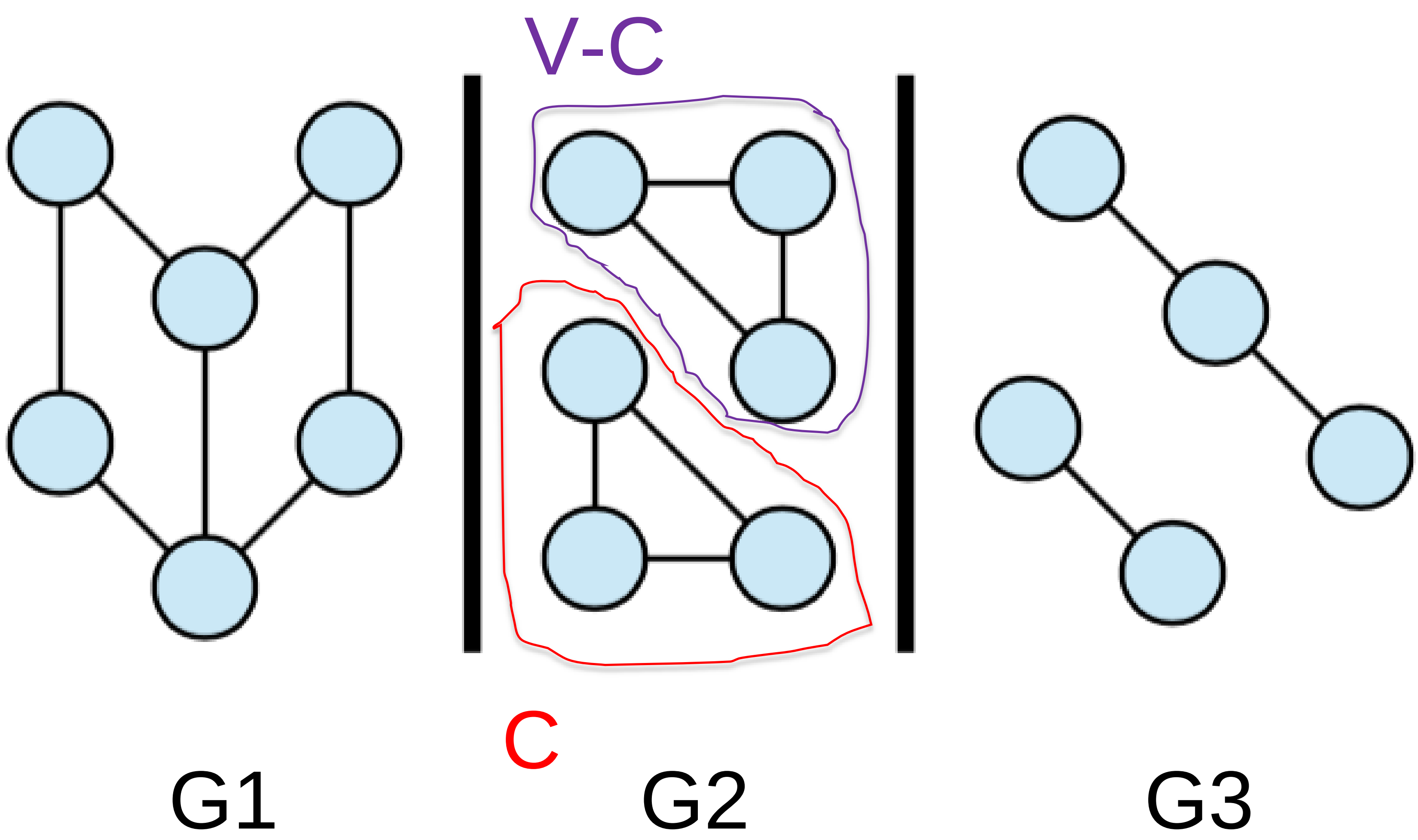
G1 seems like it is one big piece.

G2 and G3 are in multiple pieces.



G1 G2 G3

Let G = (V, E) be an undirected graph. A connected component of G is a nonempty set of nodes C (that is, C ⊆ V), such that
(1) For any u, v ∈ C, we have u ↔ v.
(2) For any u ∈ C and v ∈ V – C, we have u !↔ v



Let G = (V, E) be an undirected graph. A connected component of G is a nonempty set of nodes C (that is, $C \subseteq V$), such that

(1) For any $u, v \in C$, we have $u \leftrightarrow v$.

(2) For any $u \in C$ and $v \in V - C$, we have $u\ !\!\leftrightarrow v$

Graph Traversals

Traversing a Graph

- Visit every edge and vertex in a systematic way

- Why do this?

"One of the fundamental operations in a graph is finding vertices that can be reached from a specified vertex."

For example, imagine trying to find out how many cities in Russia can be reached by a passenger train from Kazan

Traversing a Graph

- There are two ways to traverse a graph:

  Depth-First Search (DFS)

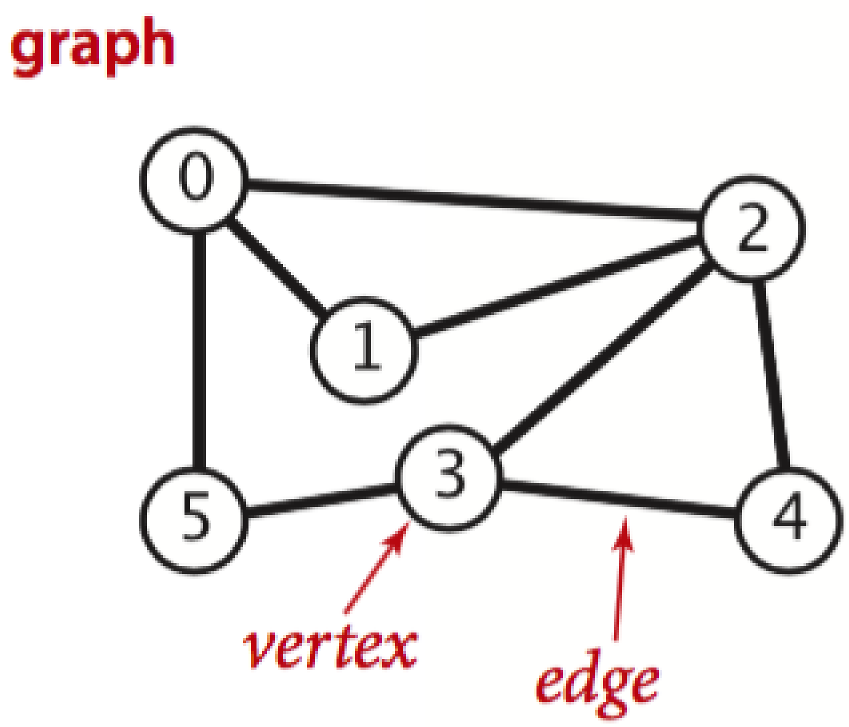
  Breadth-First Search (BFS)

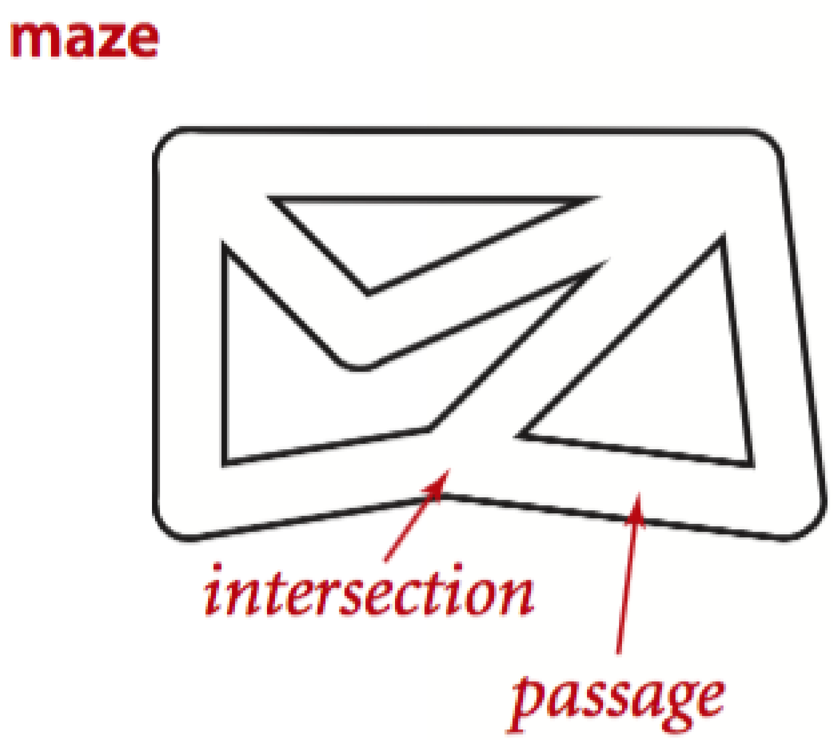
- Both will eventually reach all connected nodes
- The difference is

  DFS uses a stack

  BFS uses a queue

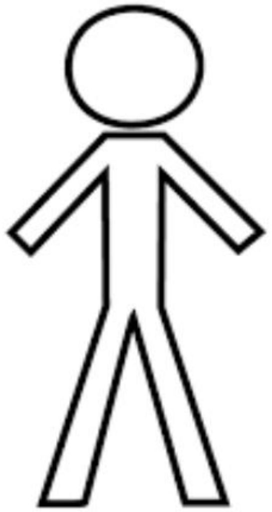
Searching in a Maze



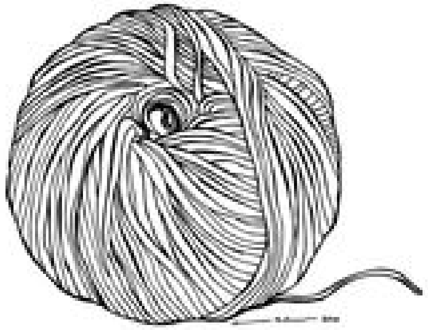


Depth First Search DFS (1)

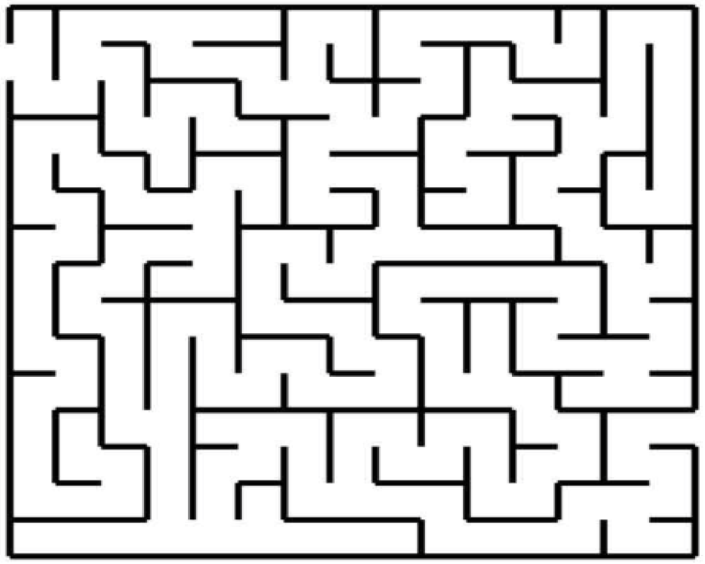
- Searching in a maze



You

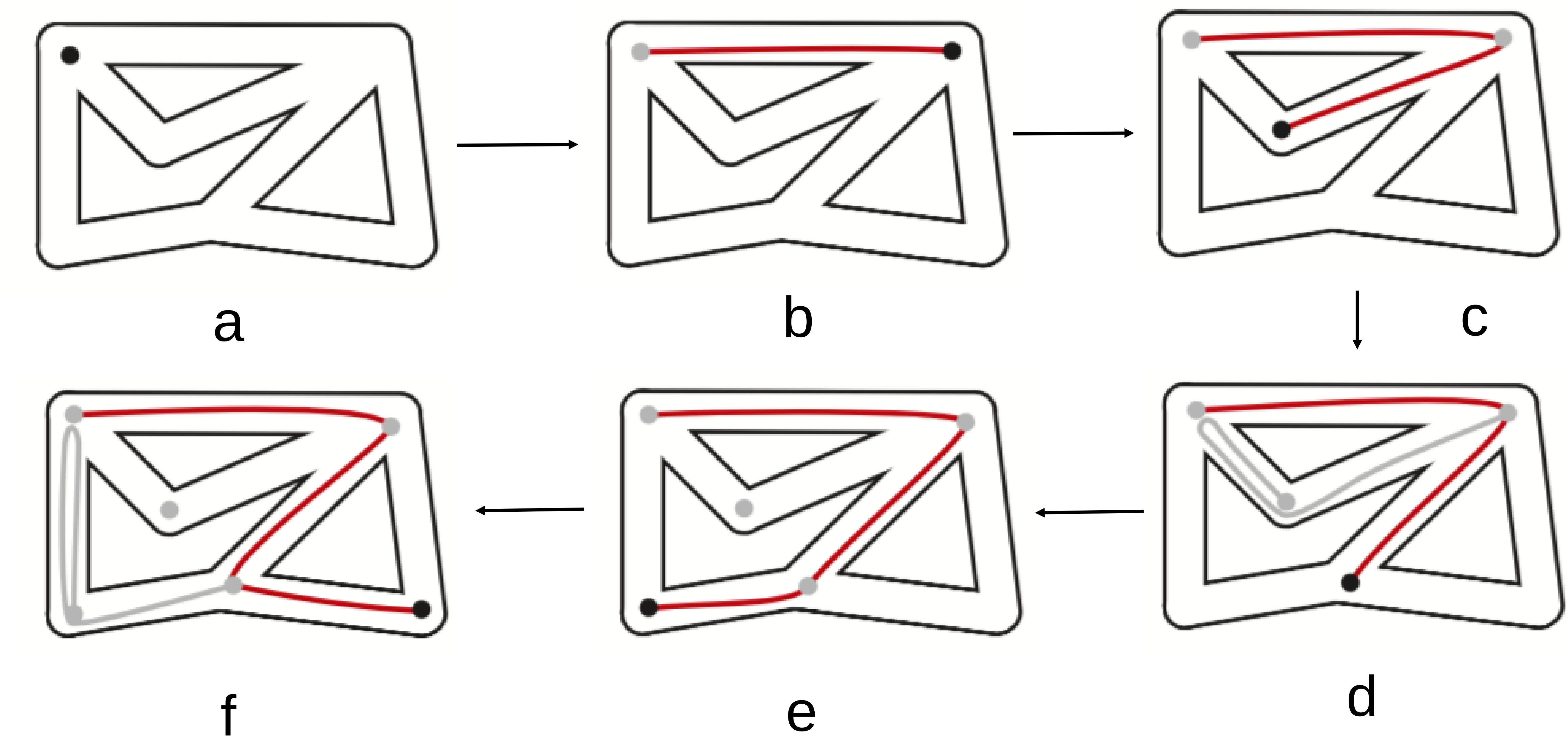


String



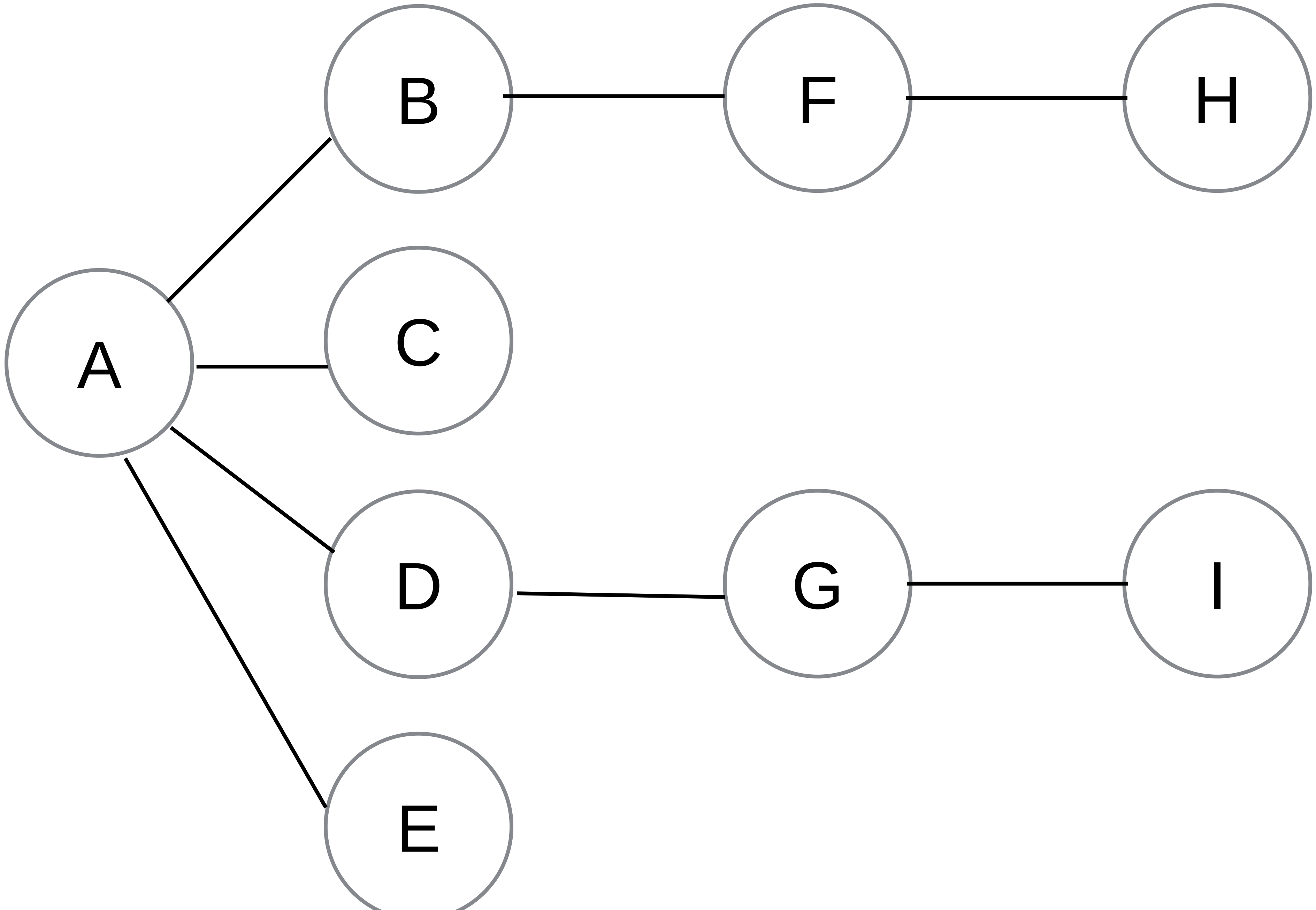
Maze

DFS (2)



Tremaux Exploration

DFS with a Stack



DFS with a Stack (2)

- Pick a starting point - in this case vertex A, and do three things

  1. visit this vertex

  2. push it on a stack

  3. mark it visited (so you won’t visit it again)

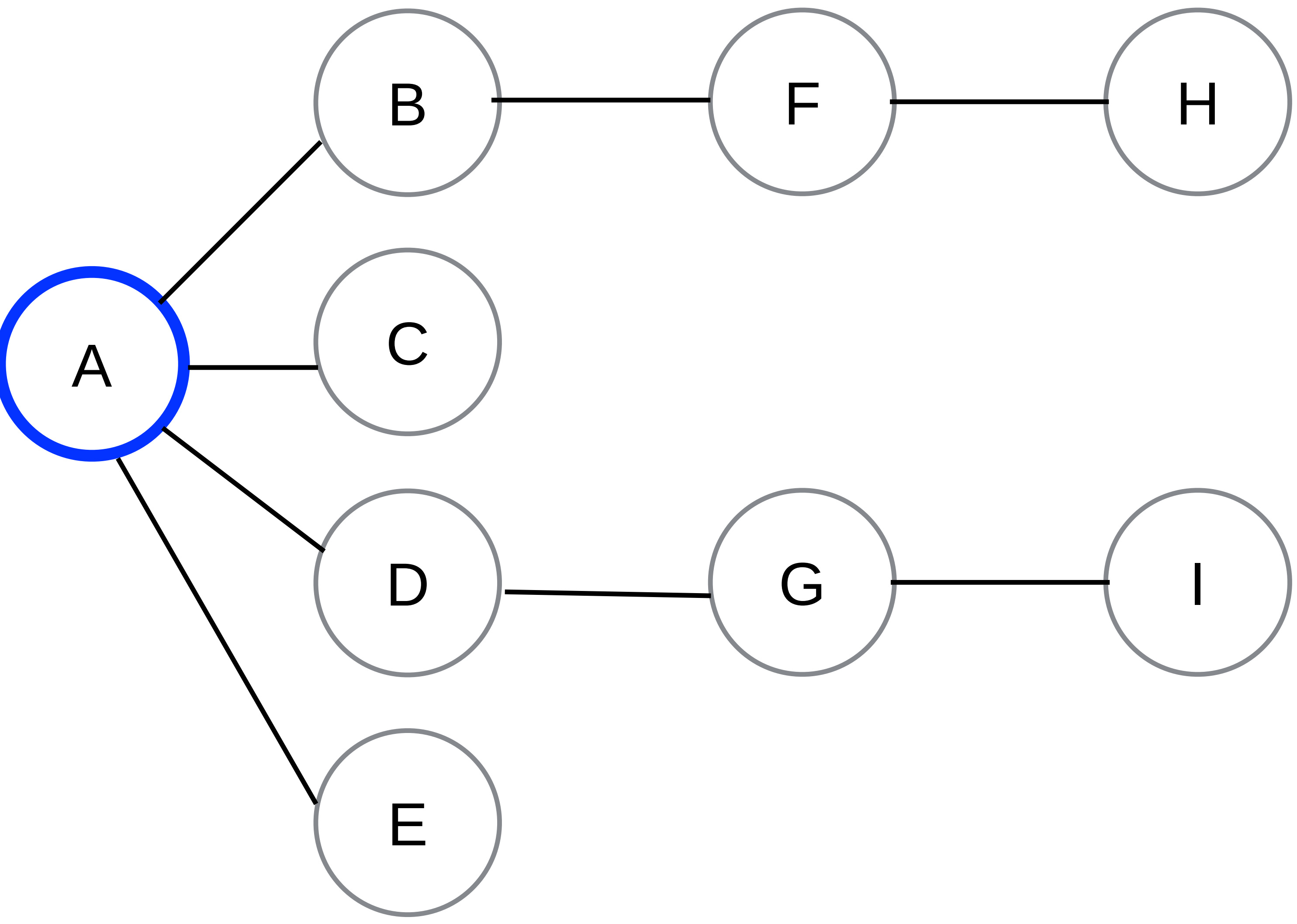
DFS with a Stack (3)

- Pick a starting point - in this case vertex A, and do three things

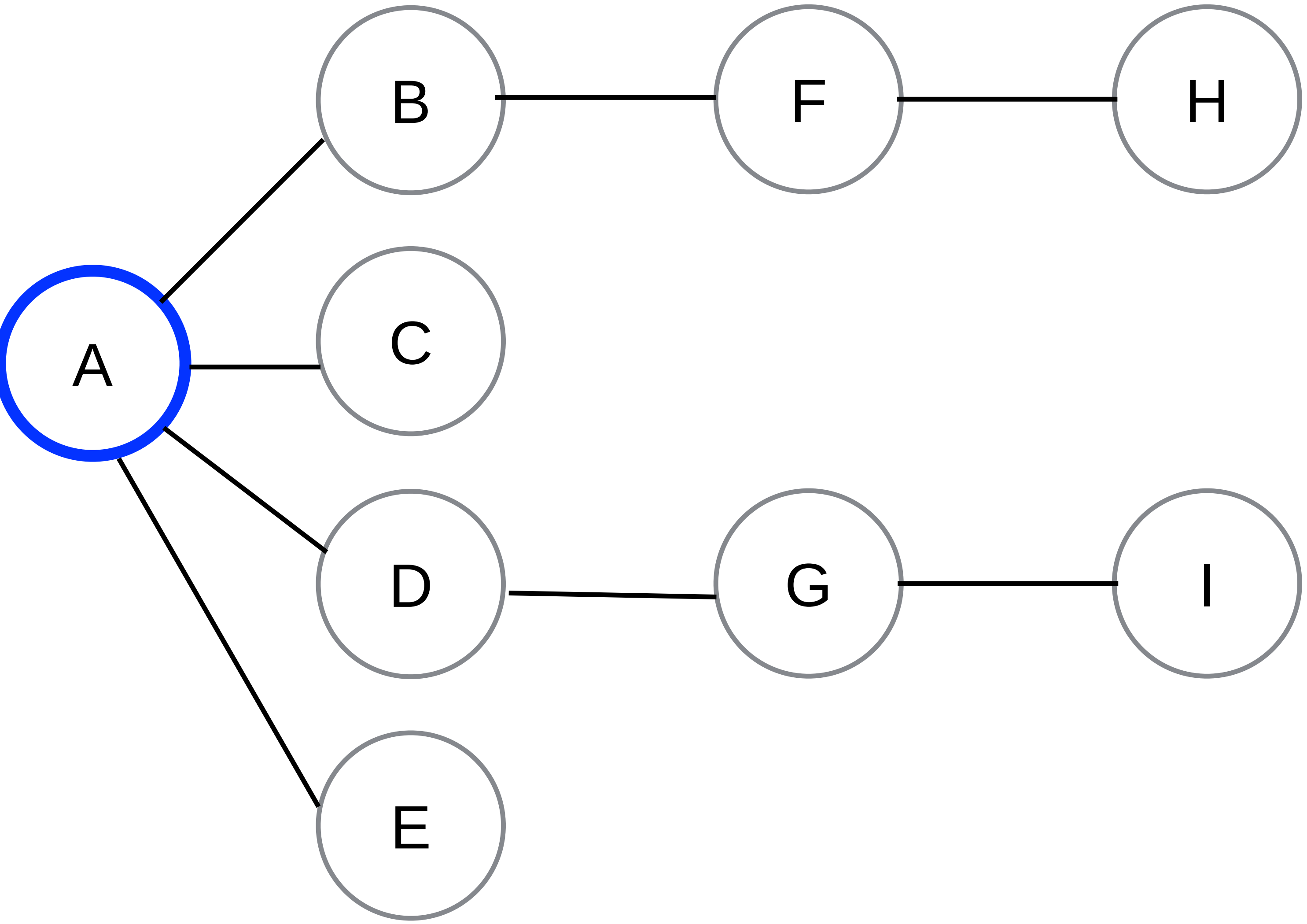
1. visit this vertex
2. push it on a stack
3. mark it visited (so you won't visit it again)

Visit is abstract, just like BST

How can you mark a vertex as visited?



Event	Stack
Visit A	A



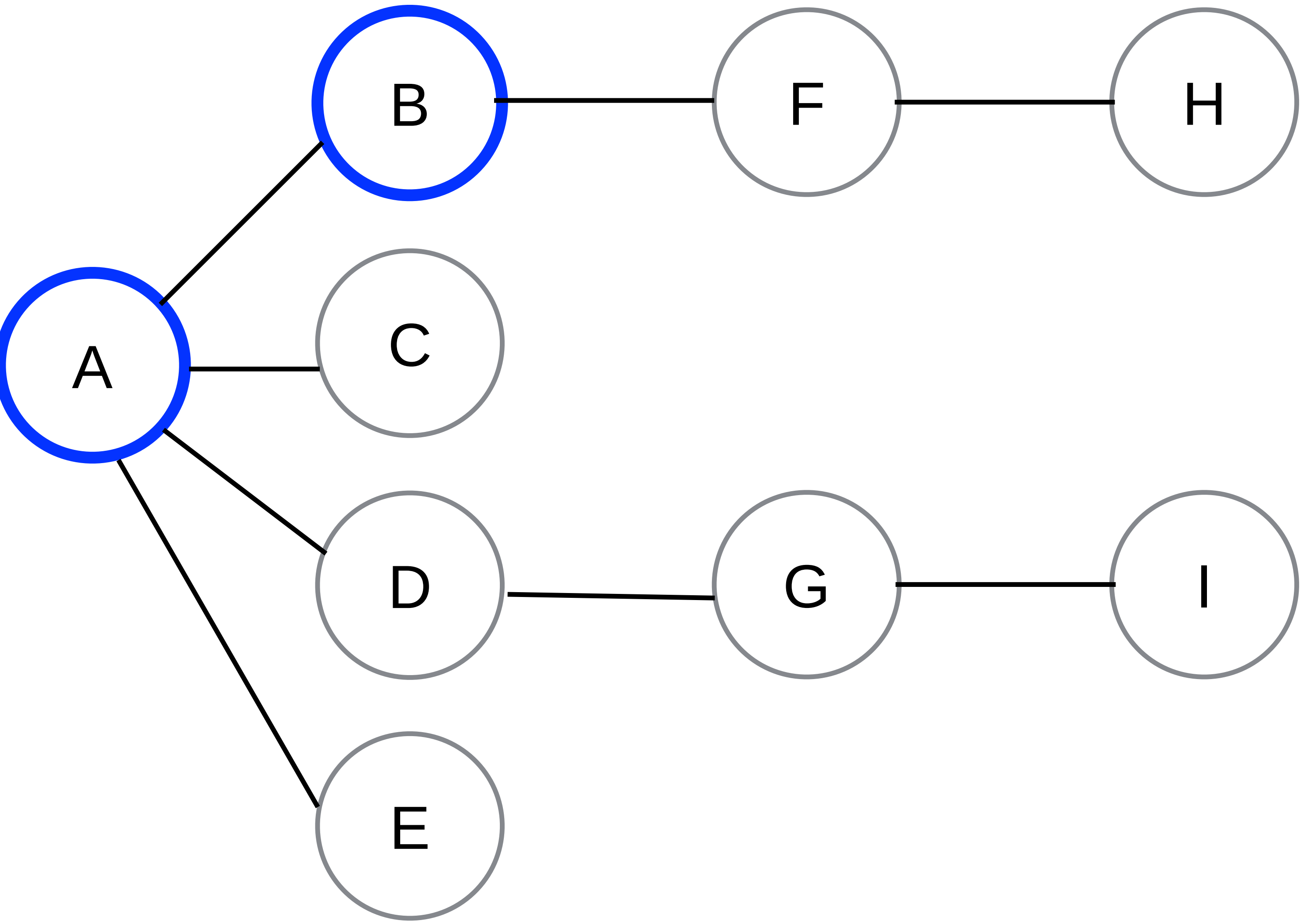
Event	Stack
Visit A	A

Next, go to a vertex adjacent to A, which hasn't been yet visited

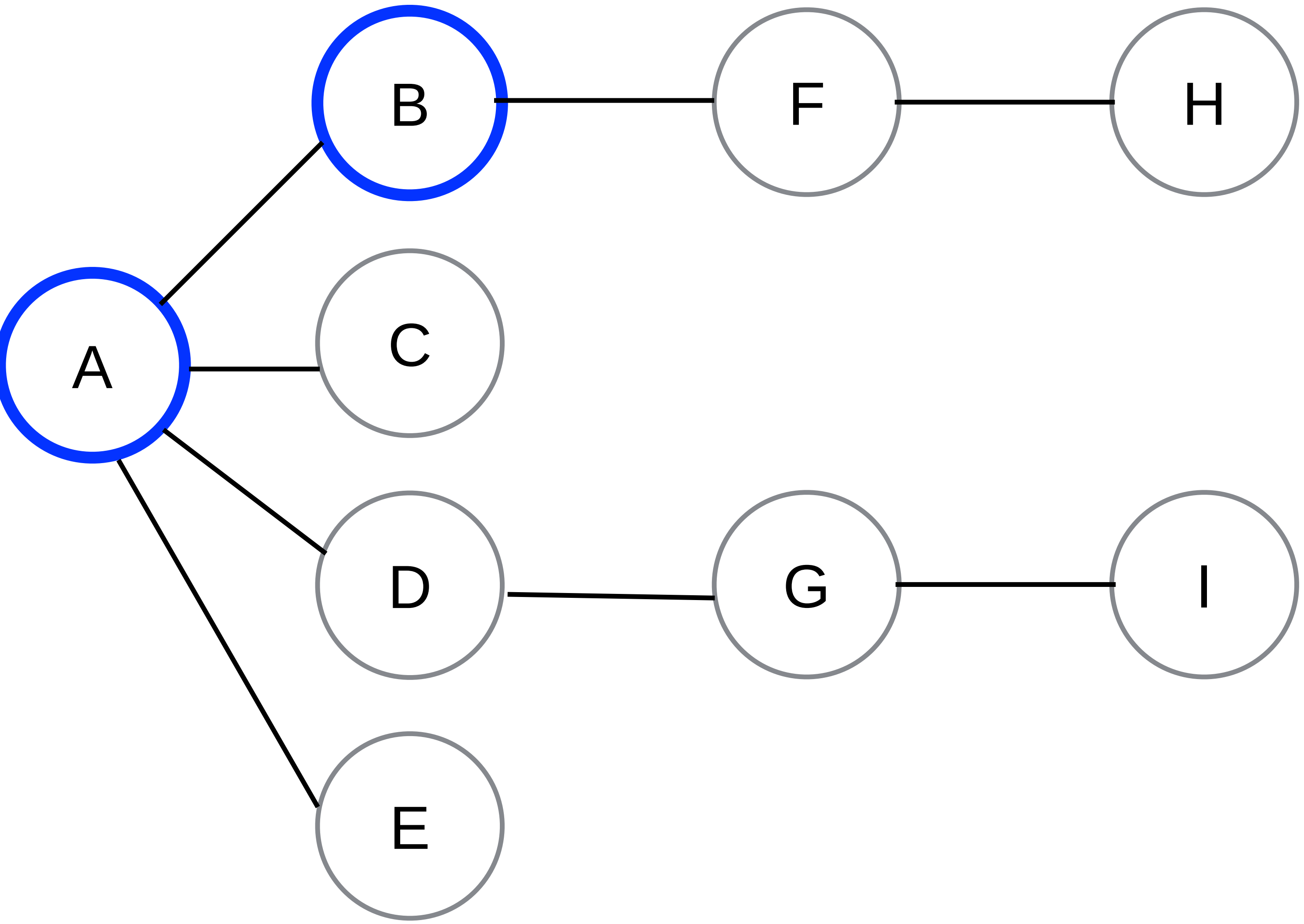
For this example, let's go to B

Visit B, mark it, and push it on the stack
Let's call this **Rule 1:**

"If possible, visit an unvisited adjacent vertex, mark it, and push it on the stack"



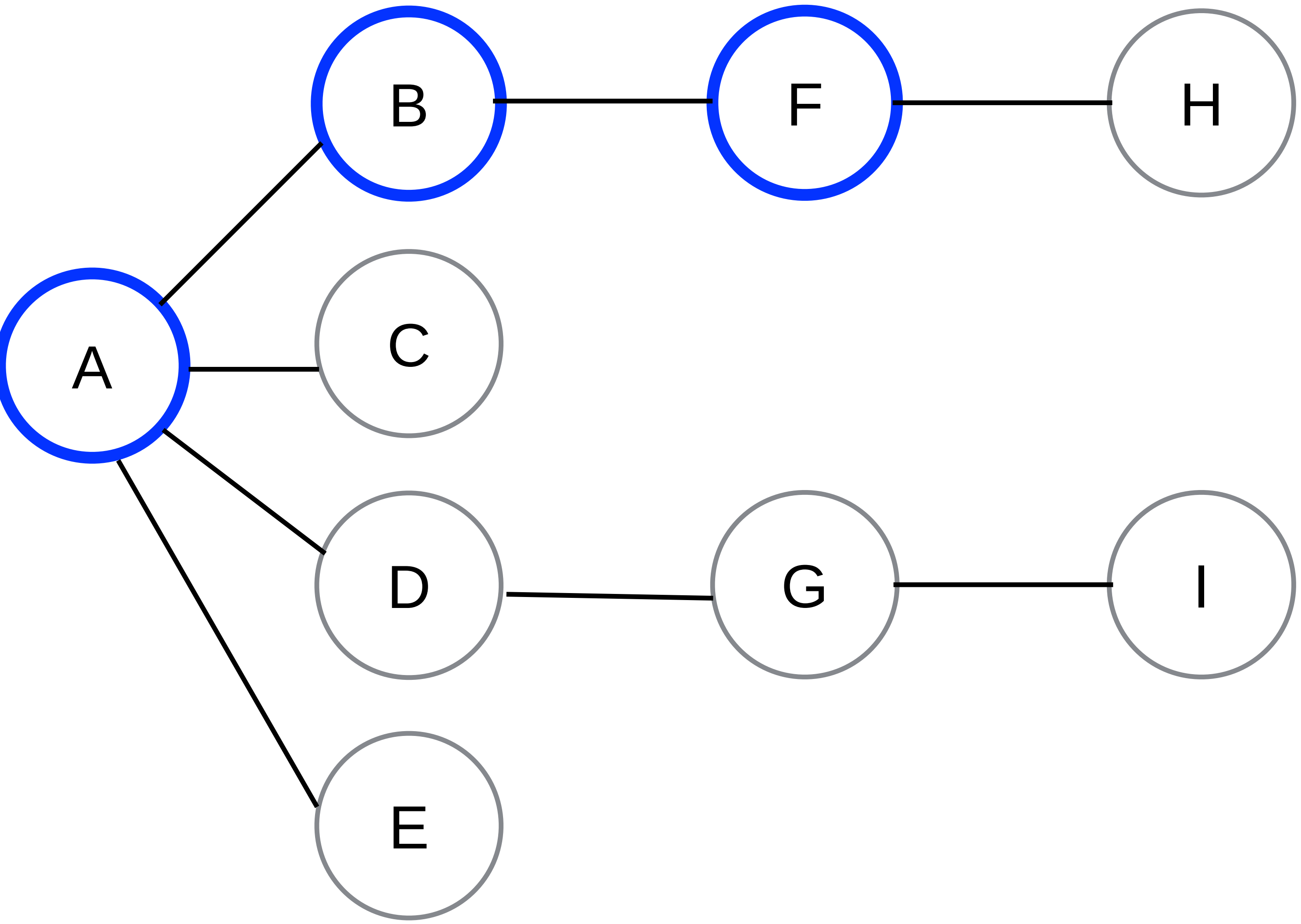
Event	Stack
Visit A	A
Visit B	AB



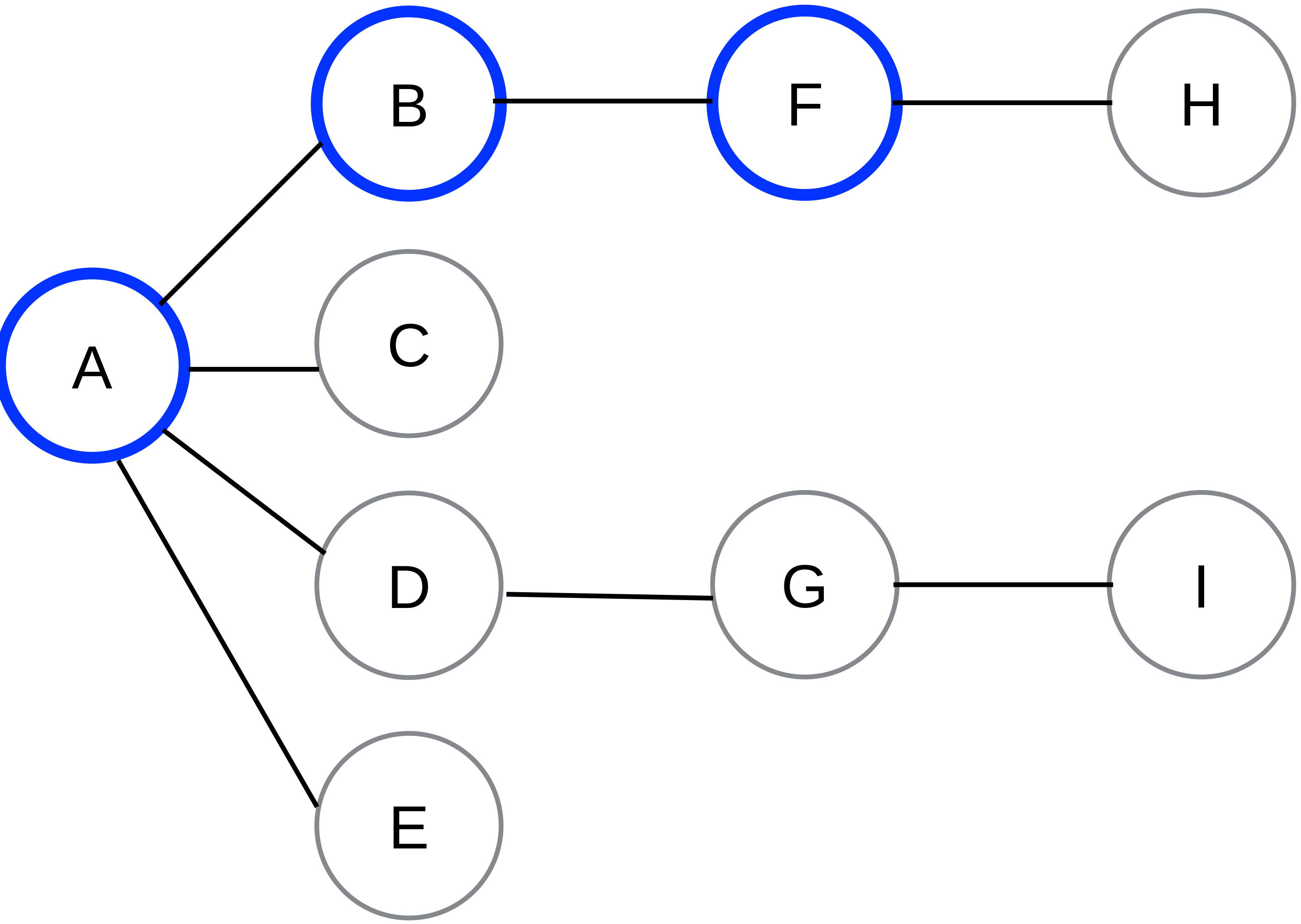
Event	Stack
Visit A	A
Visit B	AB

While at B, apply Rule 1 again.

"If possible, visit an unvisited adjacent vertex, mark it, and push it on the stack"

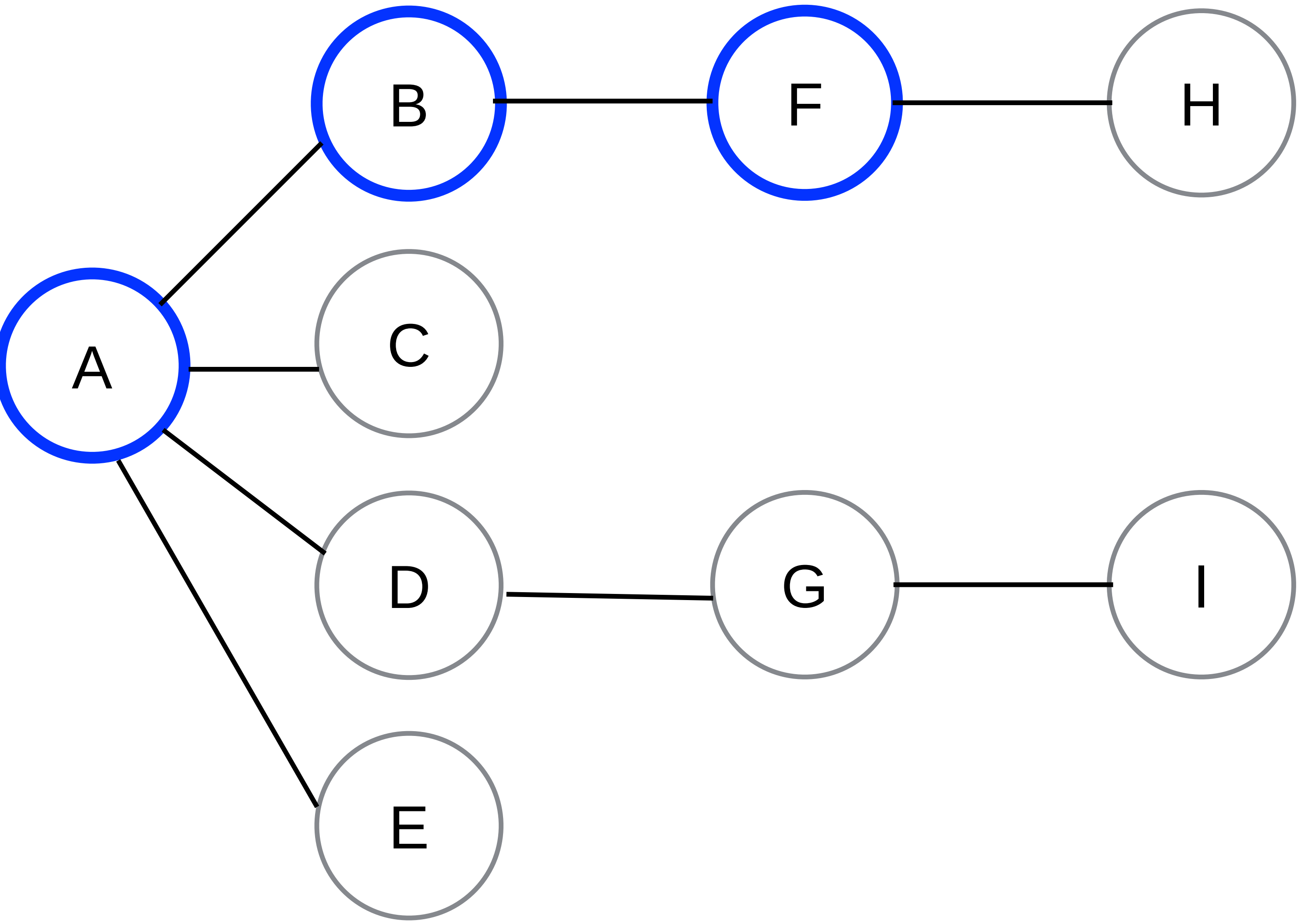


Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF



Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF

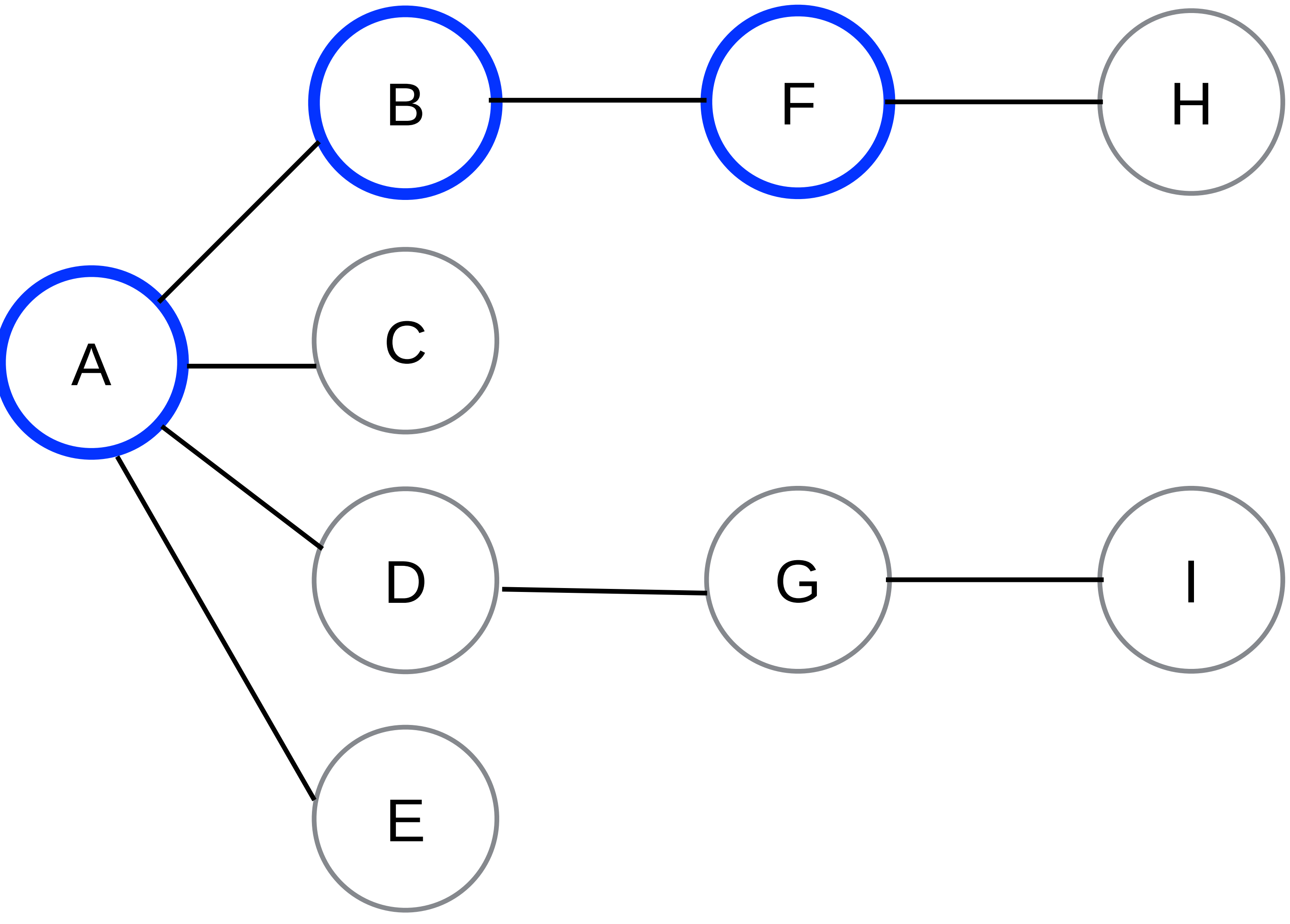
What if we had picked edge “BA”?



Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF

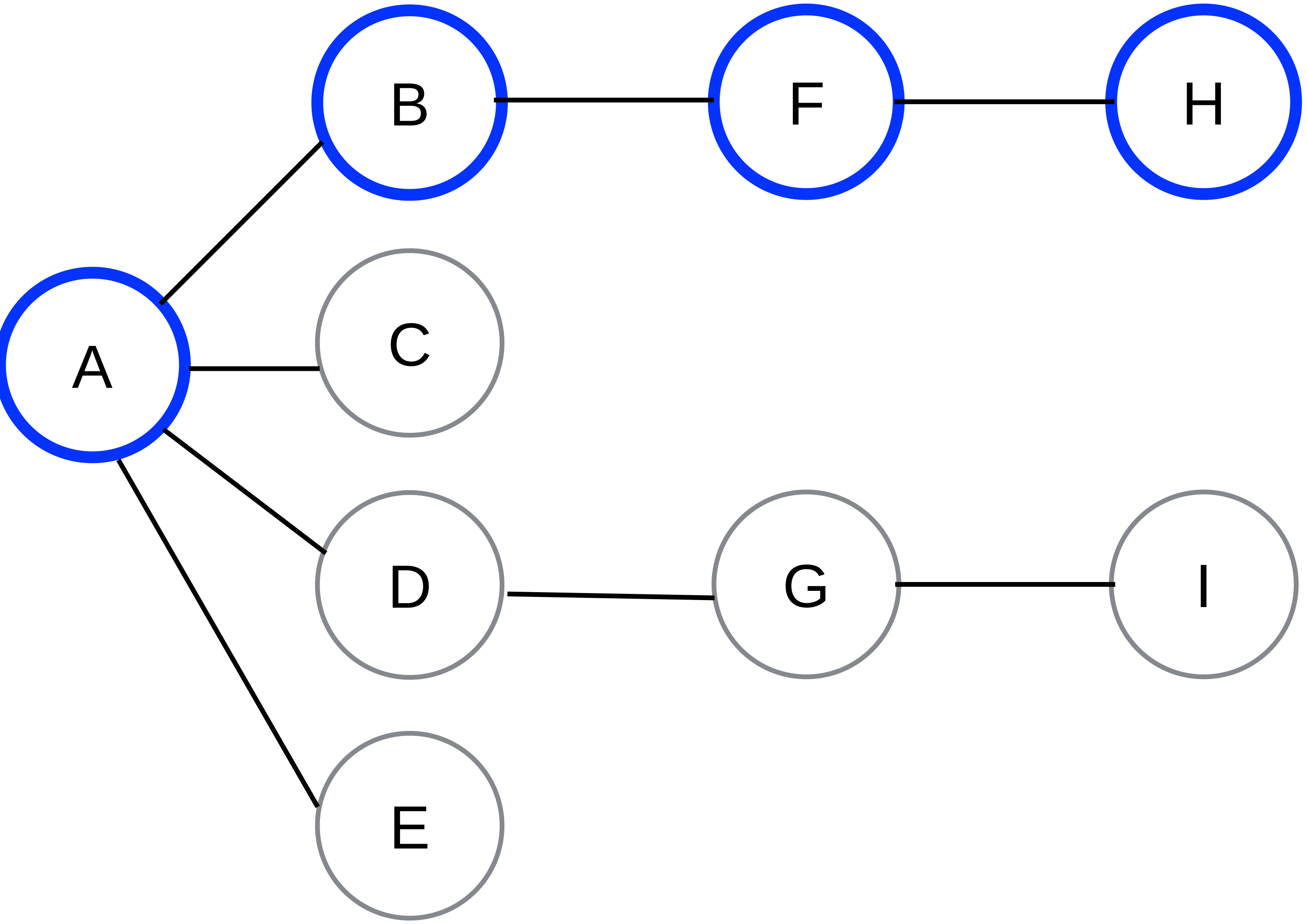
What if we had picked edge “BA”?

Thus you visit each vertex just once (put it in stack) but you visit each edge twice!

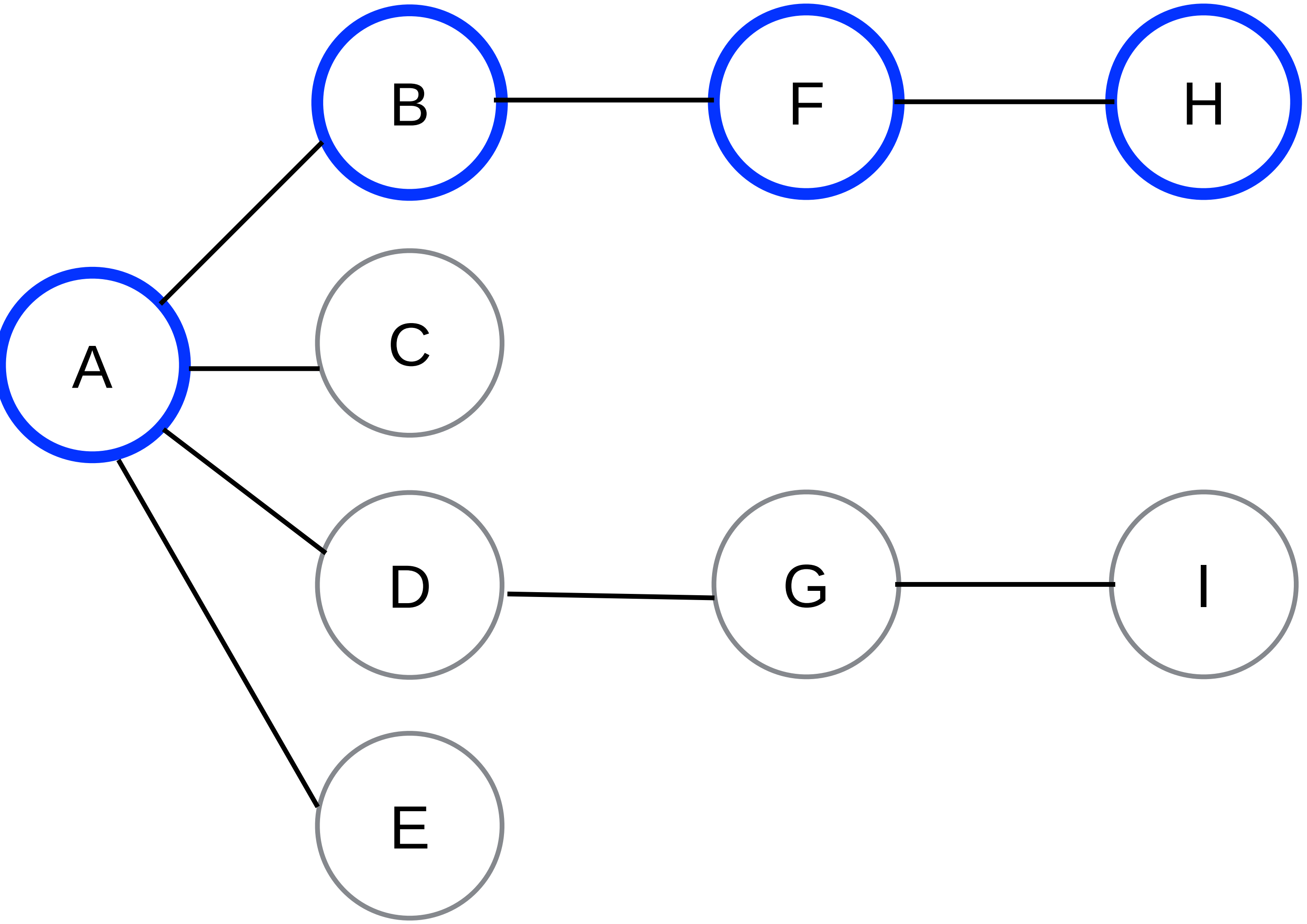


Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF

While at F, apply Rule 1 again.



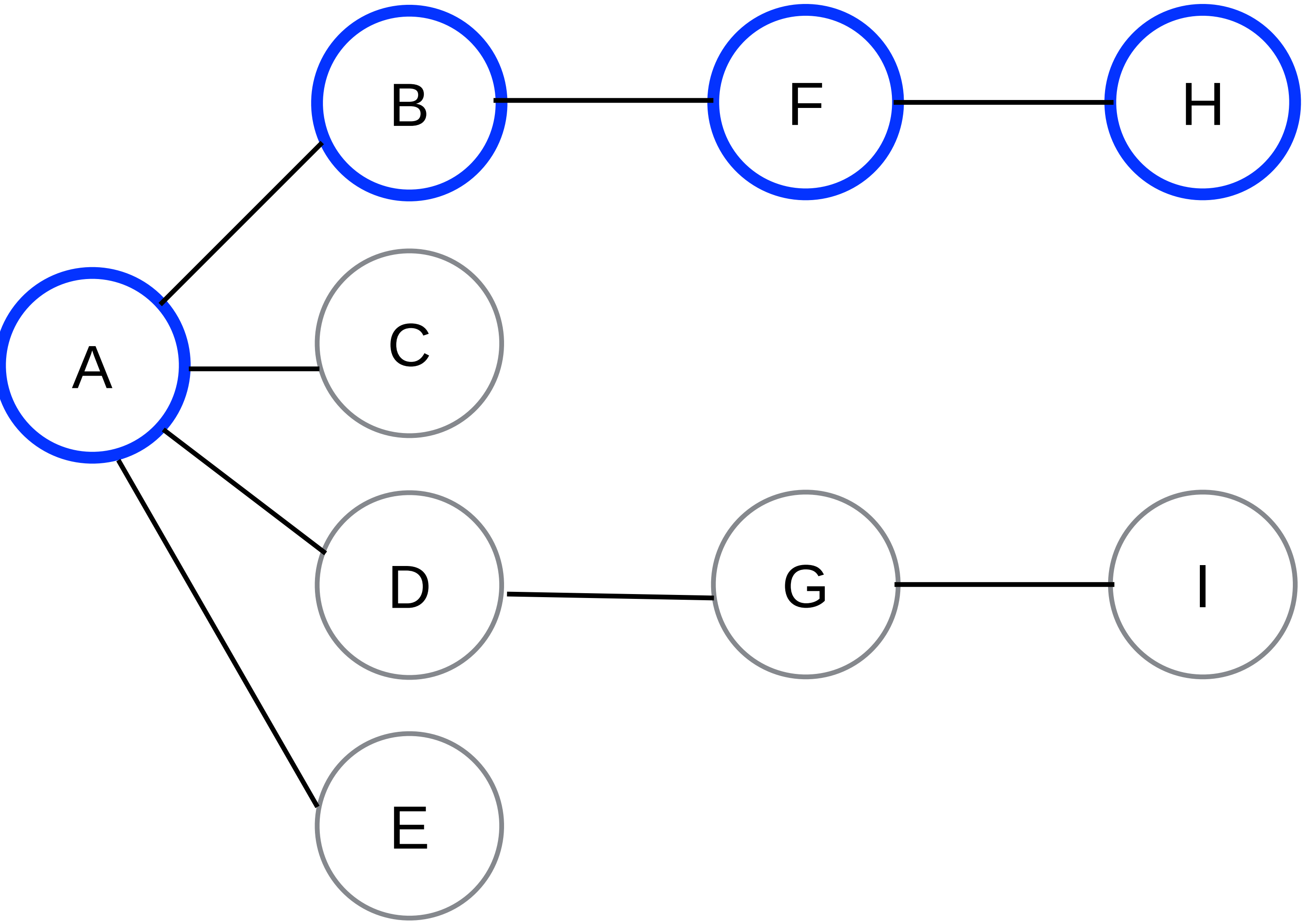
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH



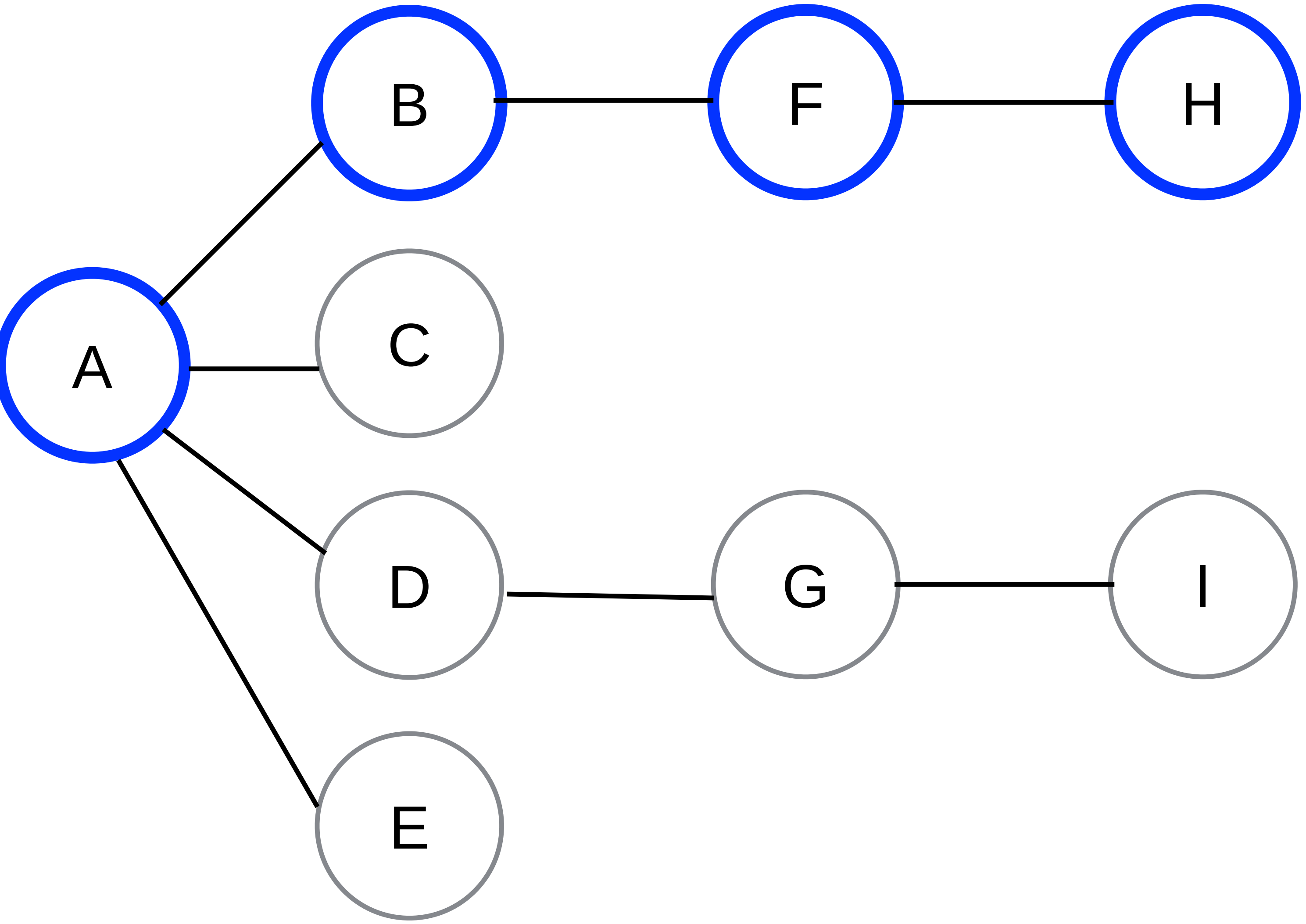
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH

At this point (at H), there are no unvisited adjacent vertices (HF leads back to F)
So we need to do something else

Rule 2:
"If you cannot follow Rule 1, then, if possible, pop a vertex off the stack"



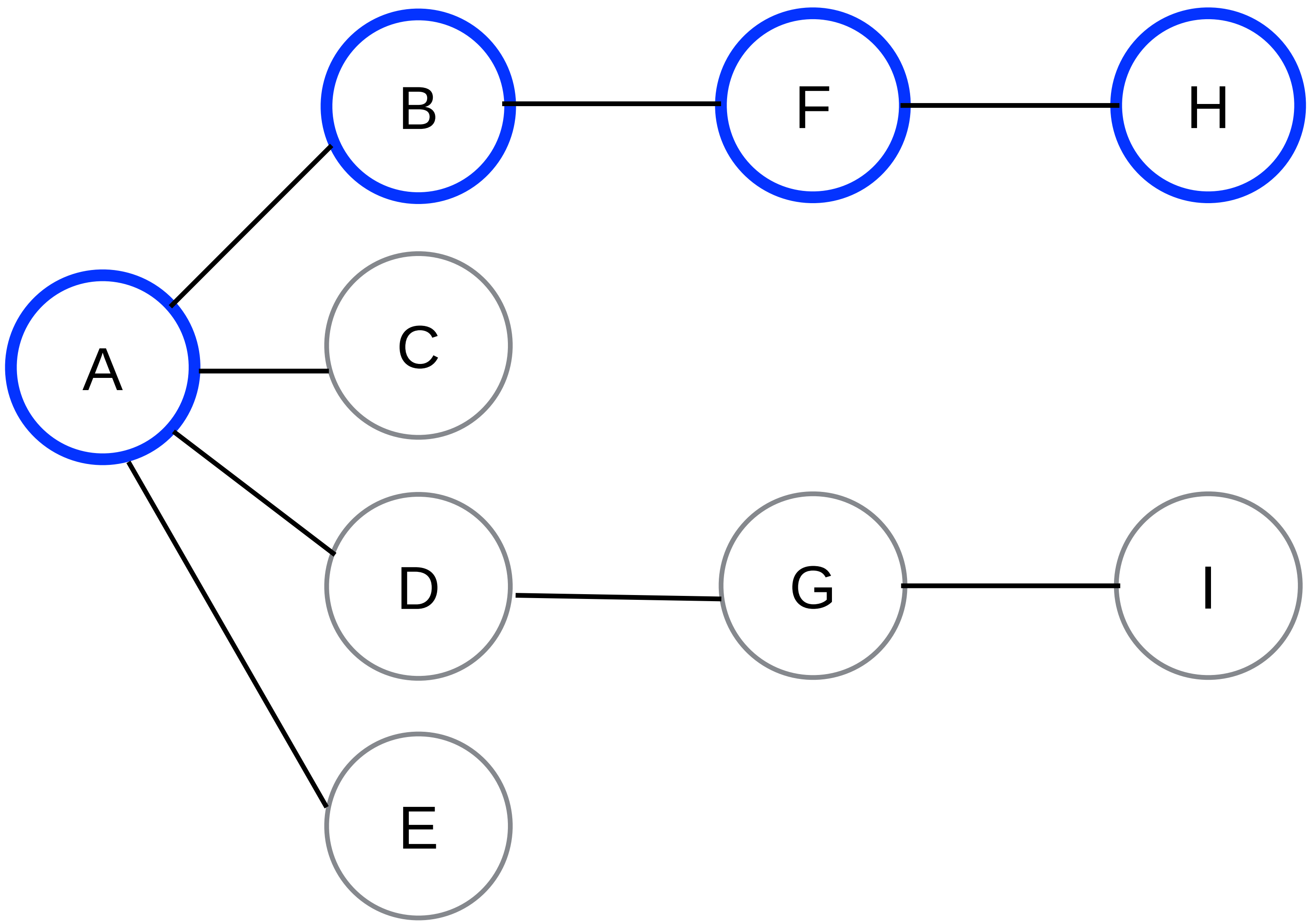
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF



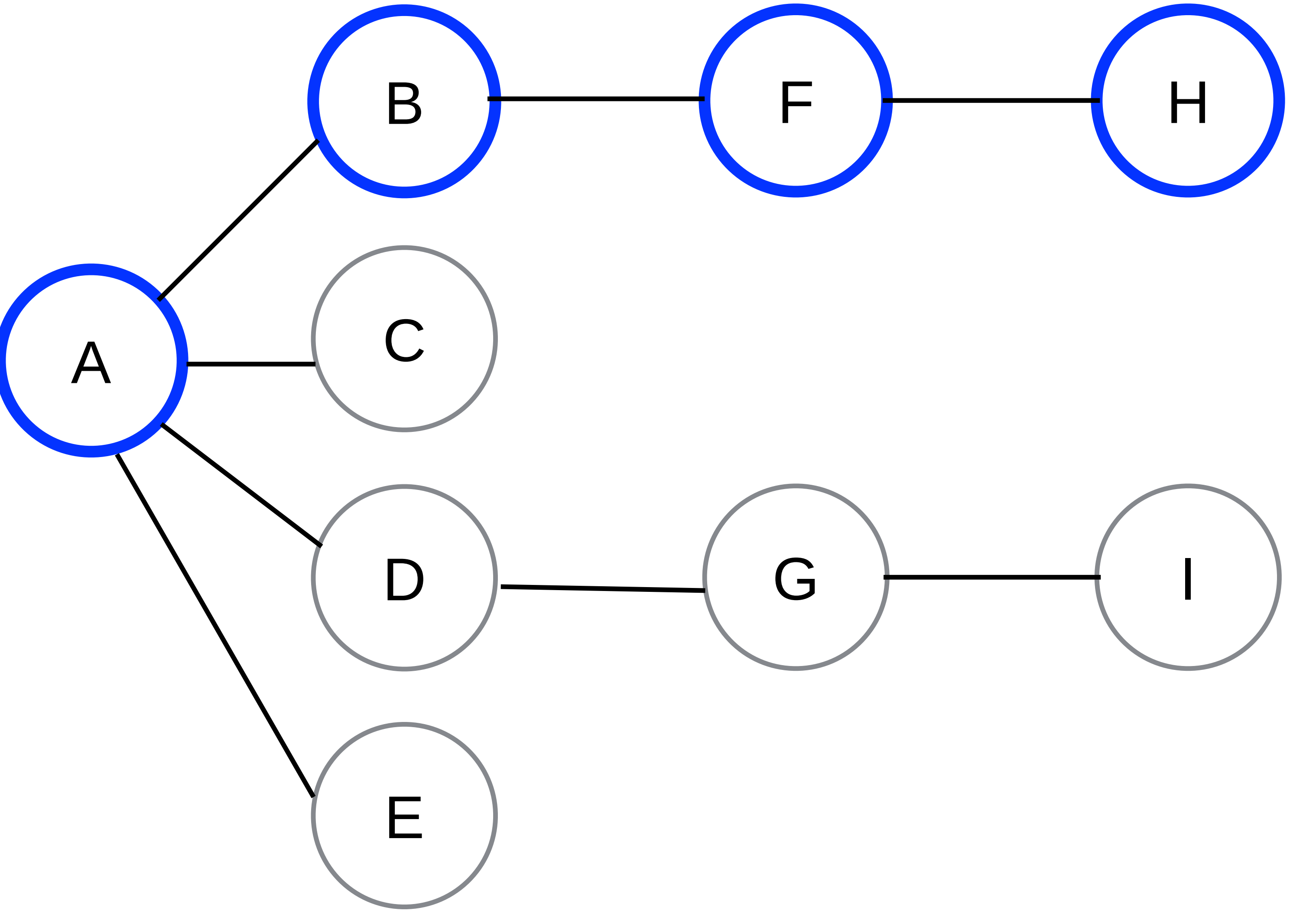
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF

We are back at F

No more unvisited adjacent vertices, so pop it off, too

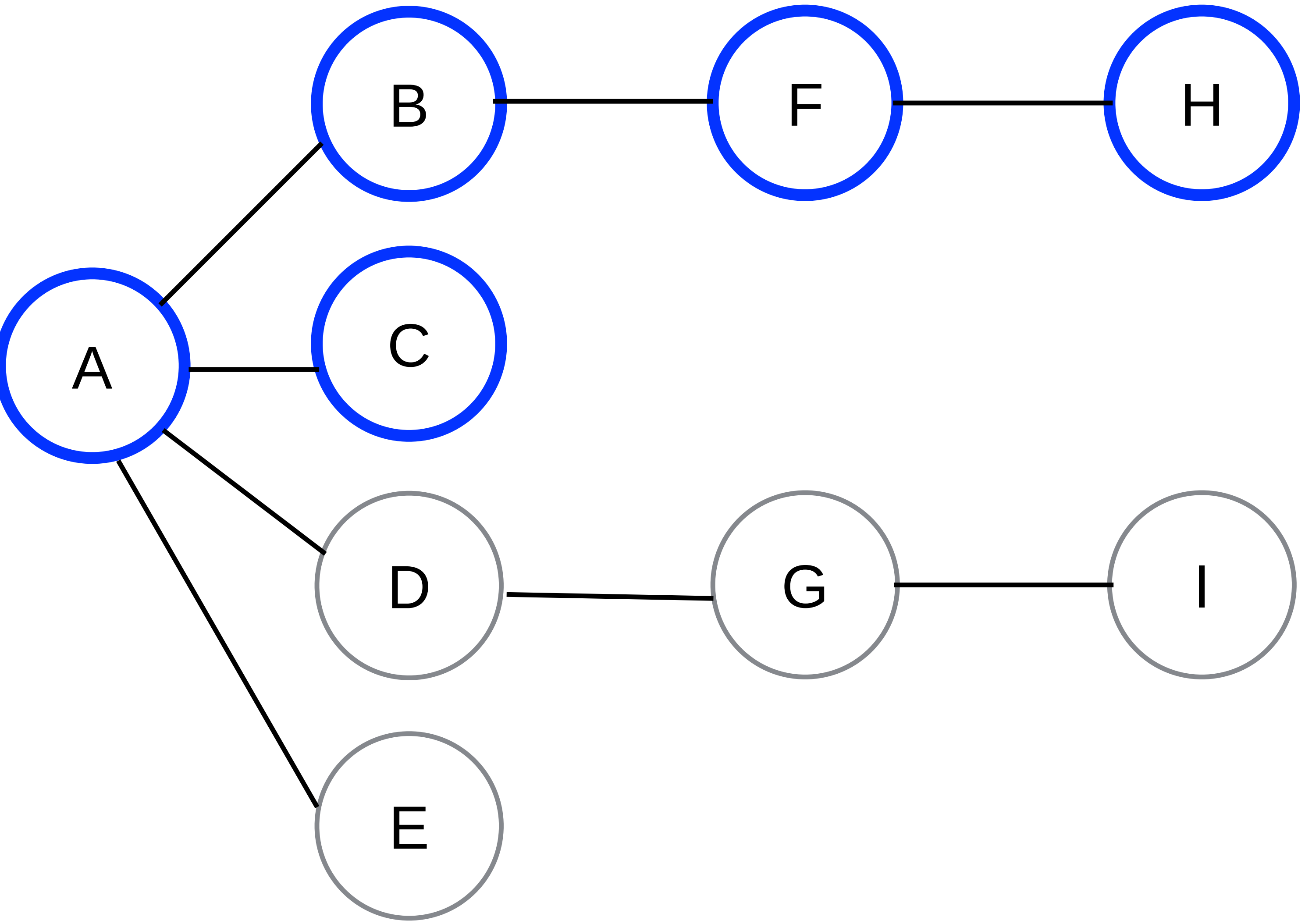


Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB

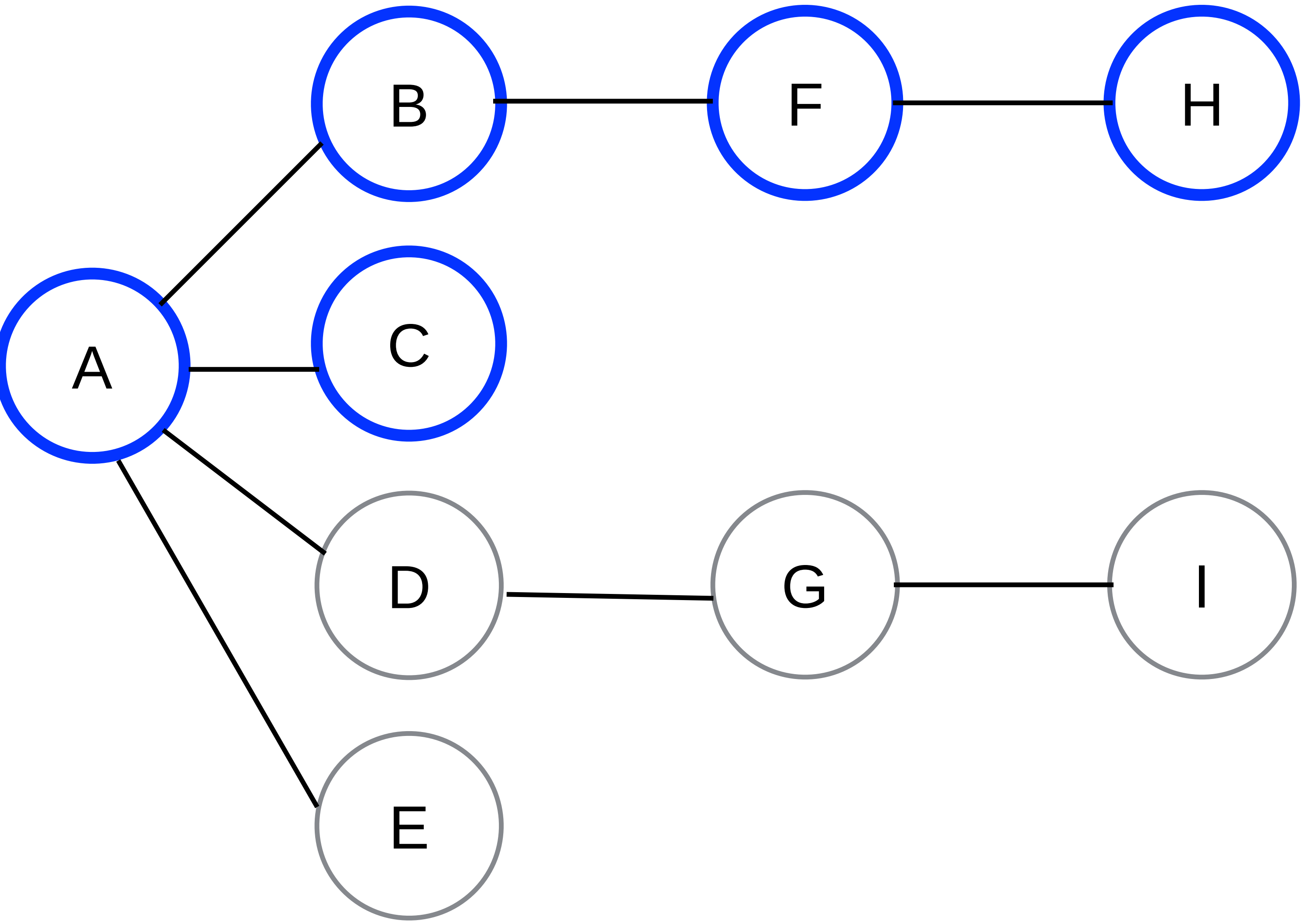


Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A

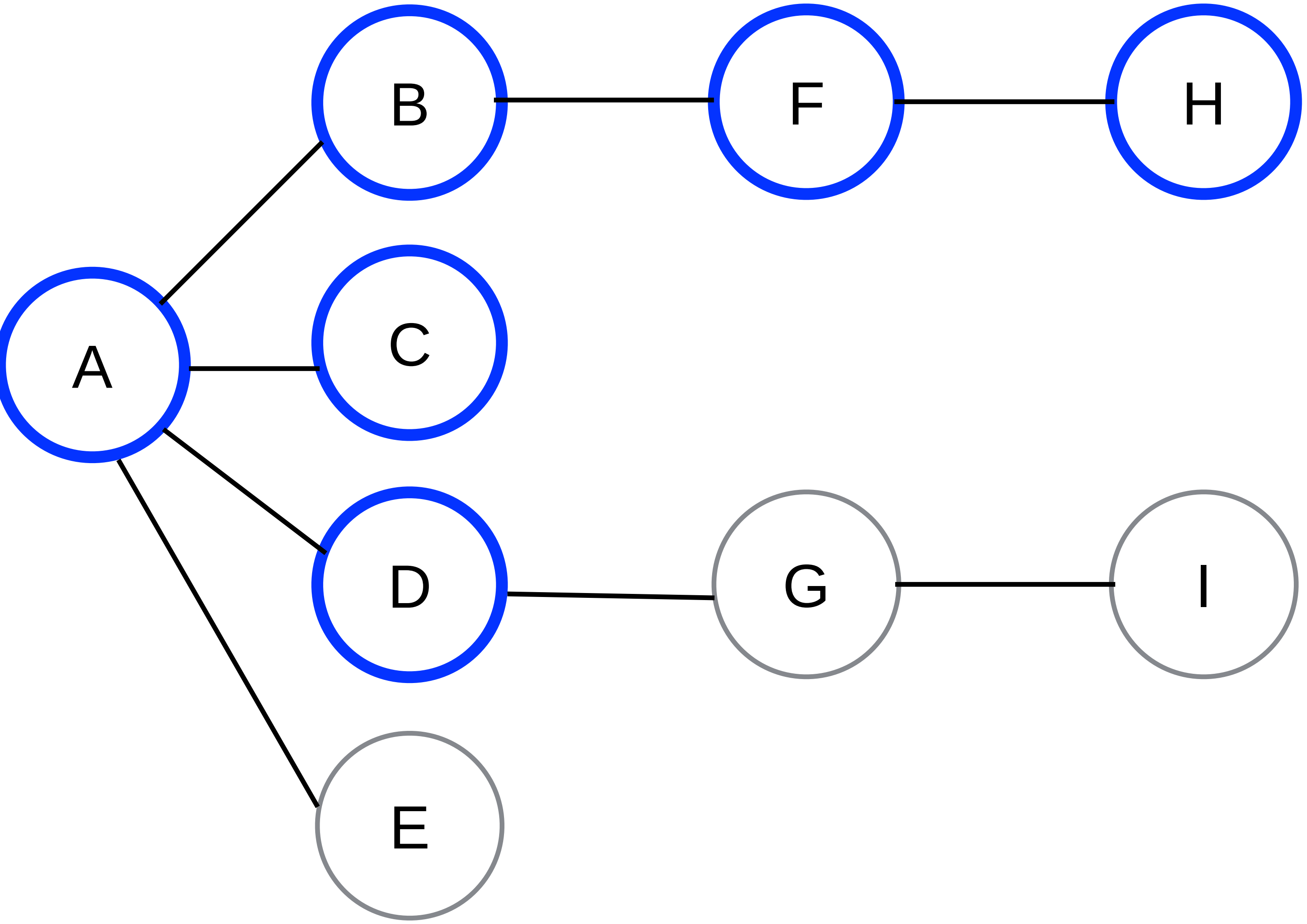
- We are back at A
- Pick the next adjacent vertex and repeat



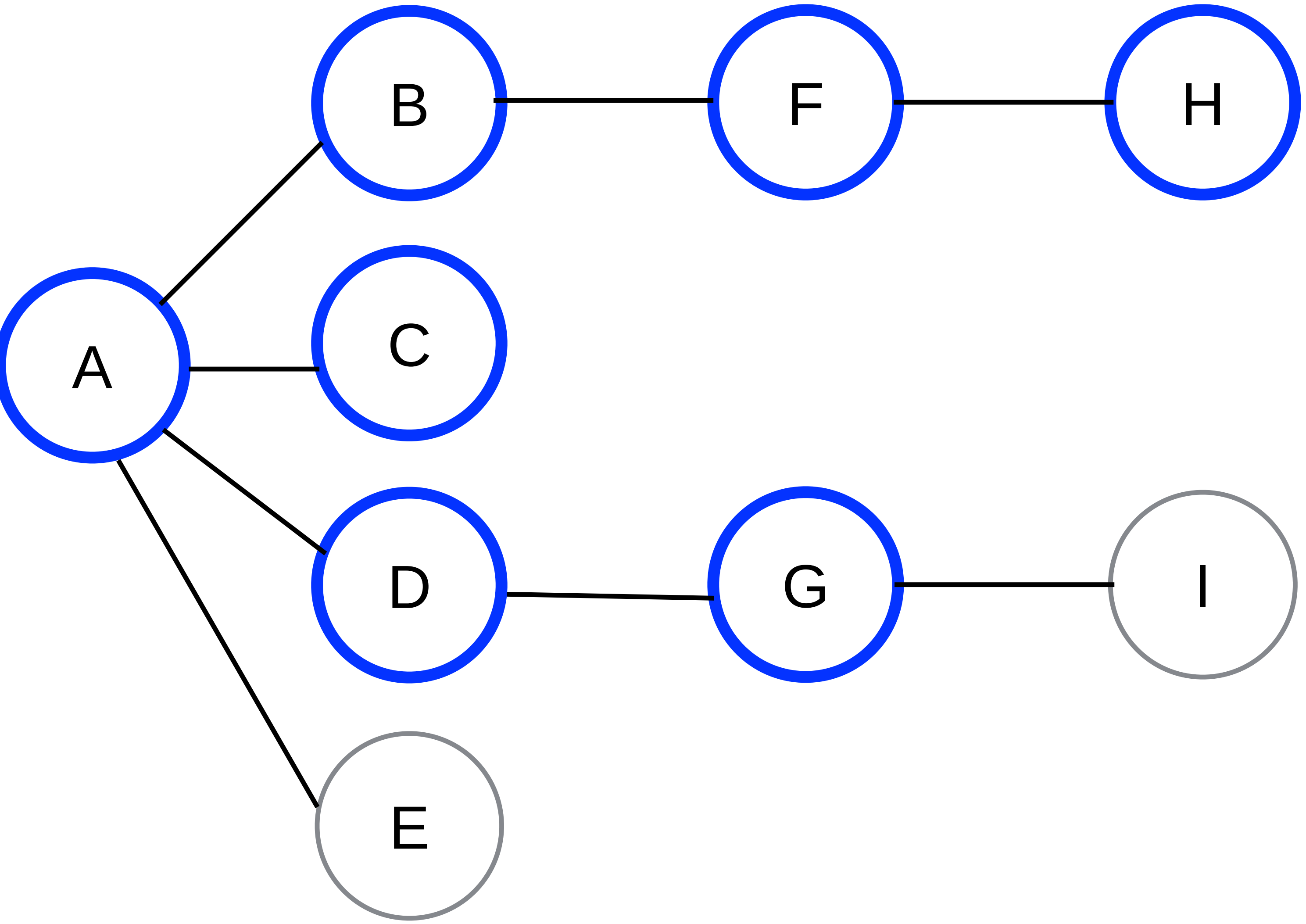
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC



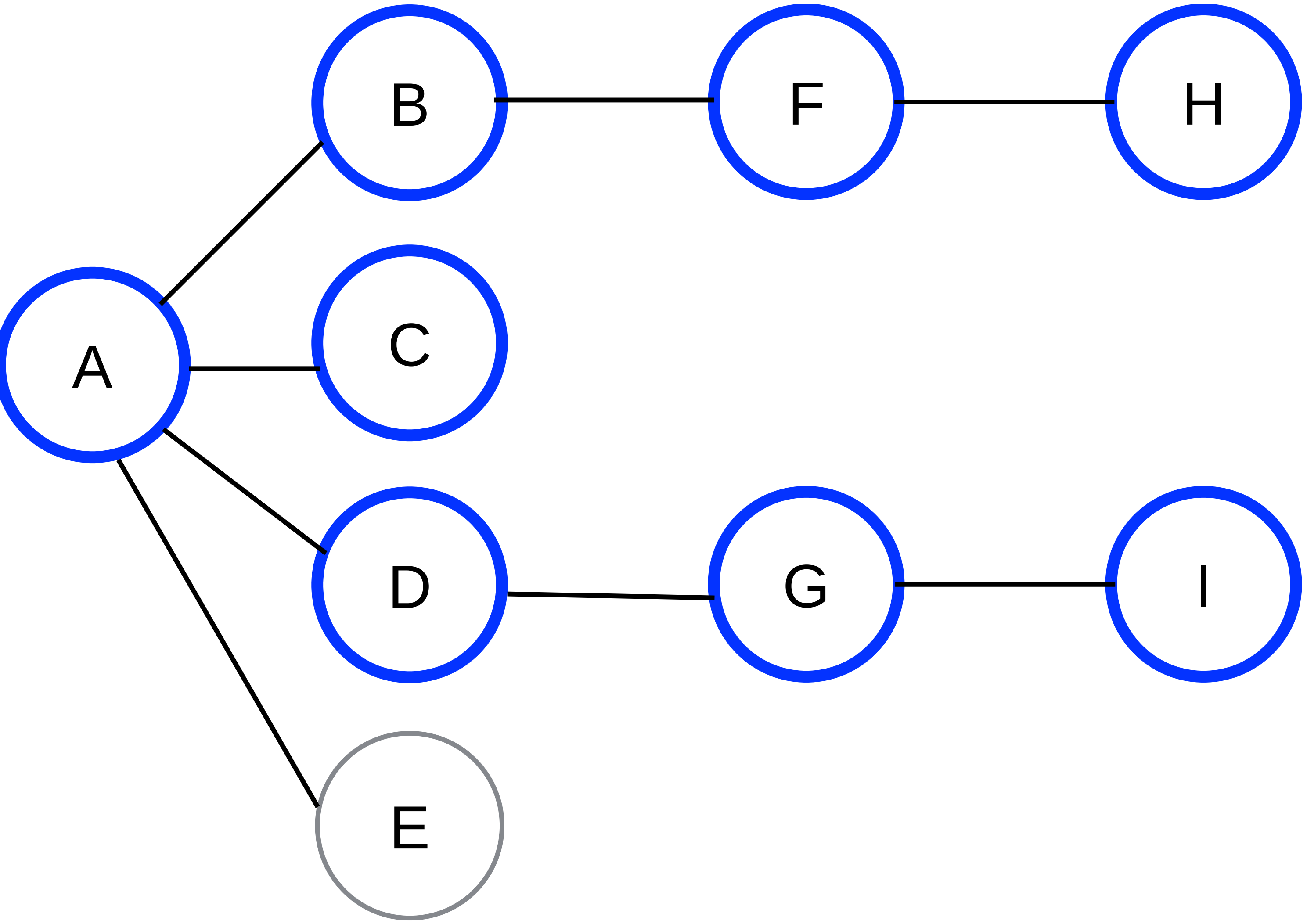
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A



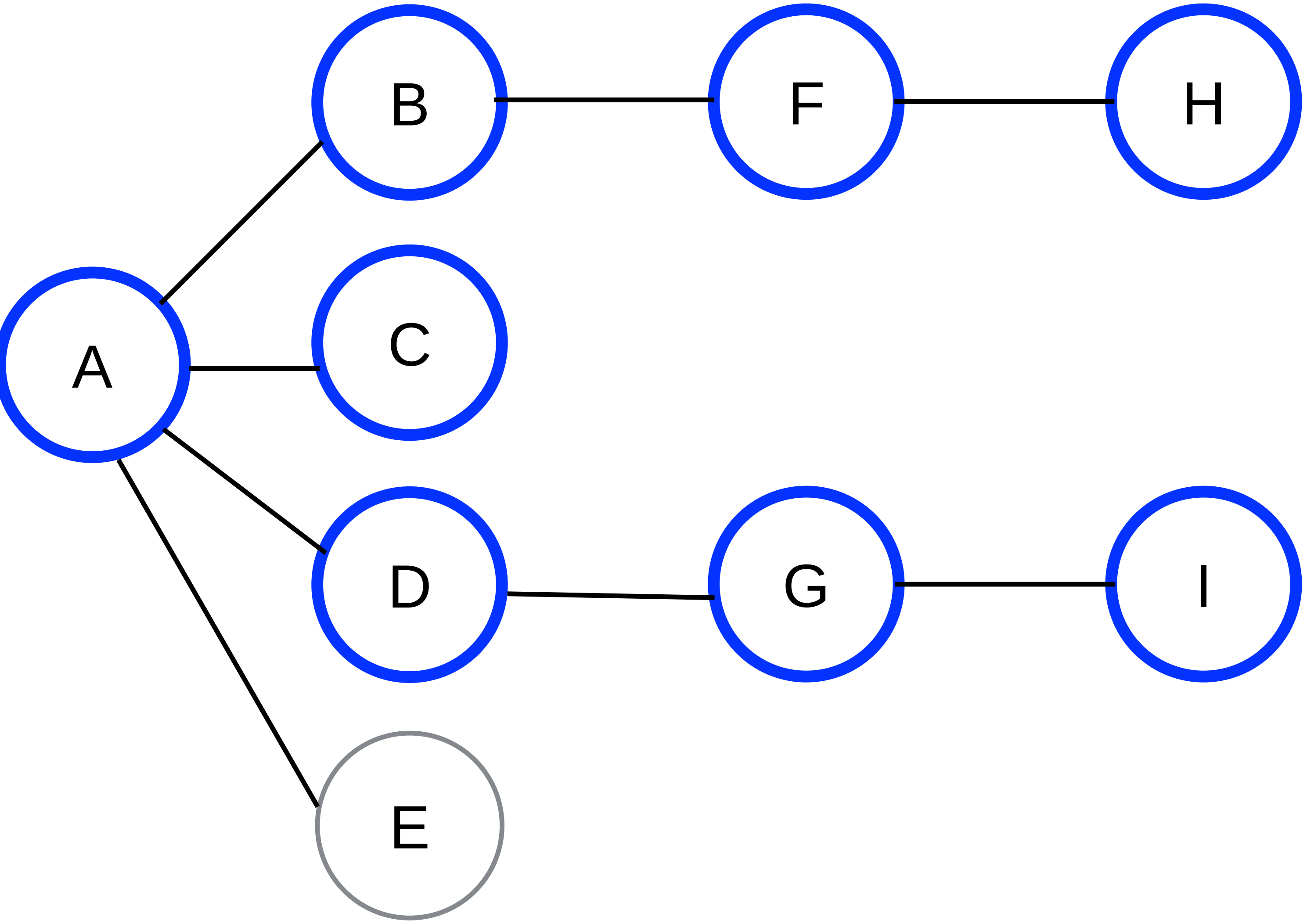
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD



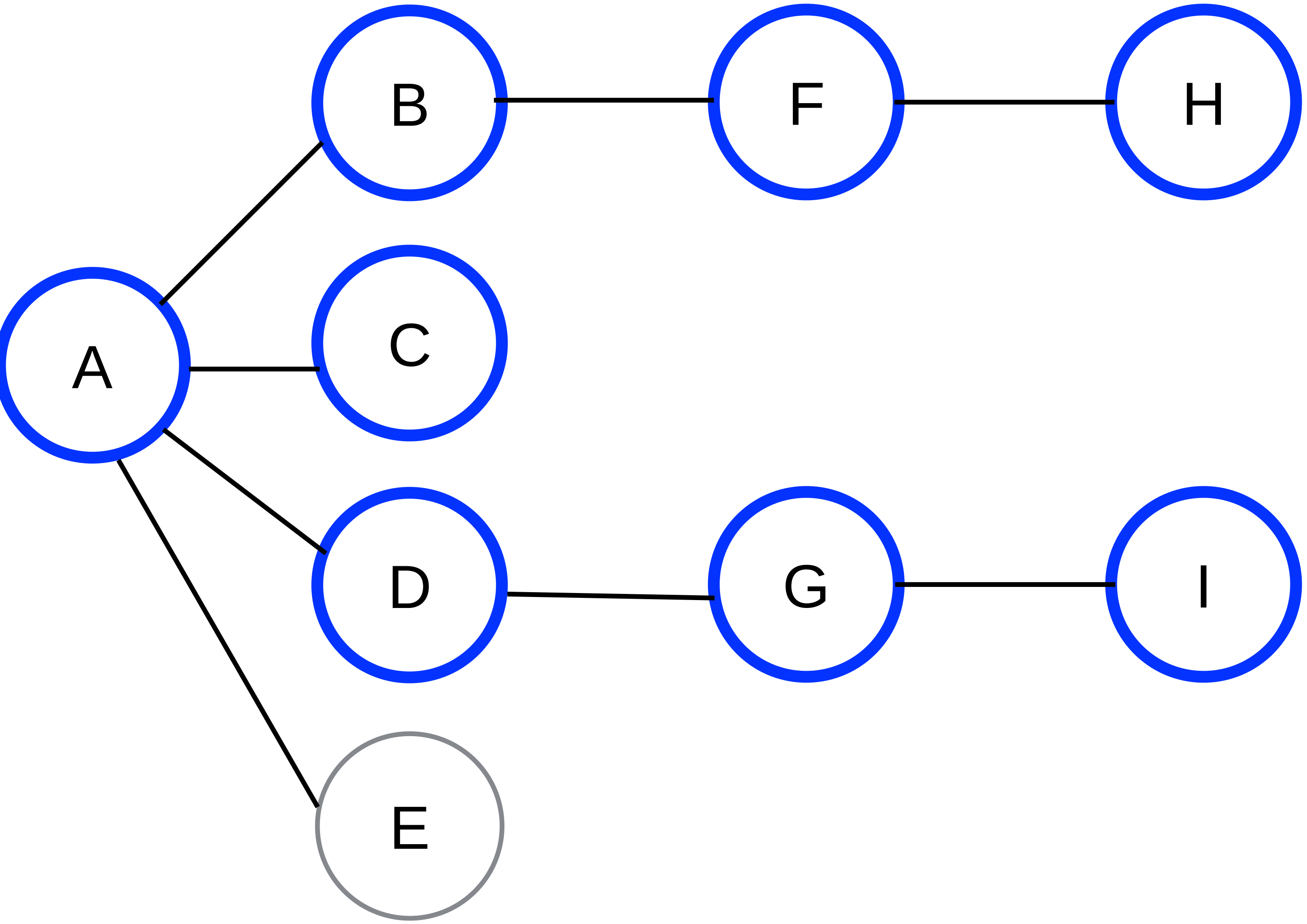
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG



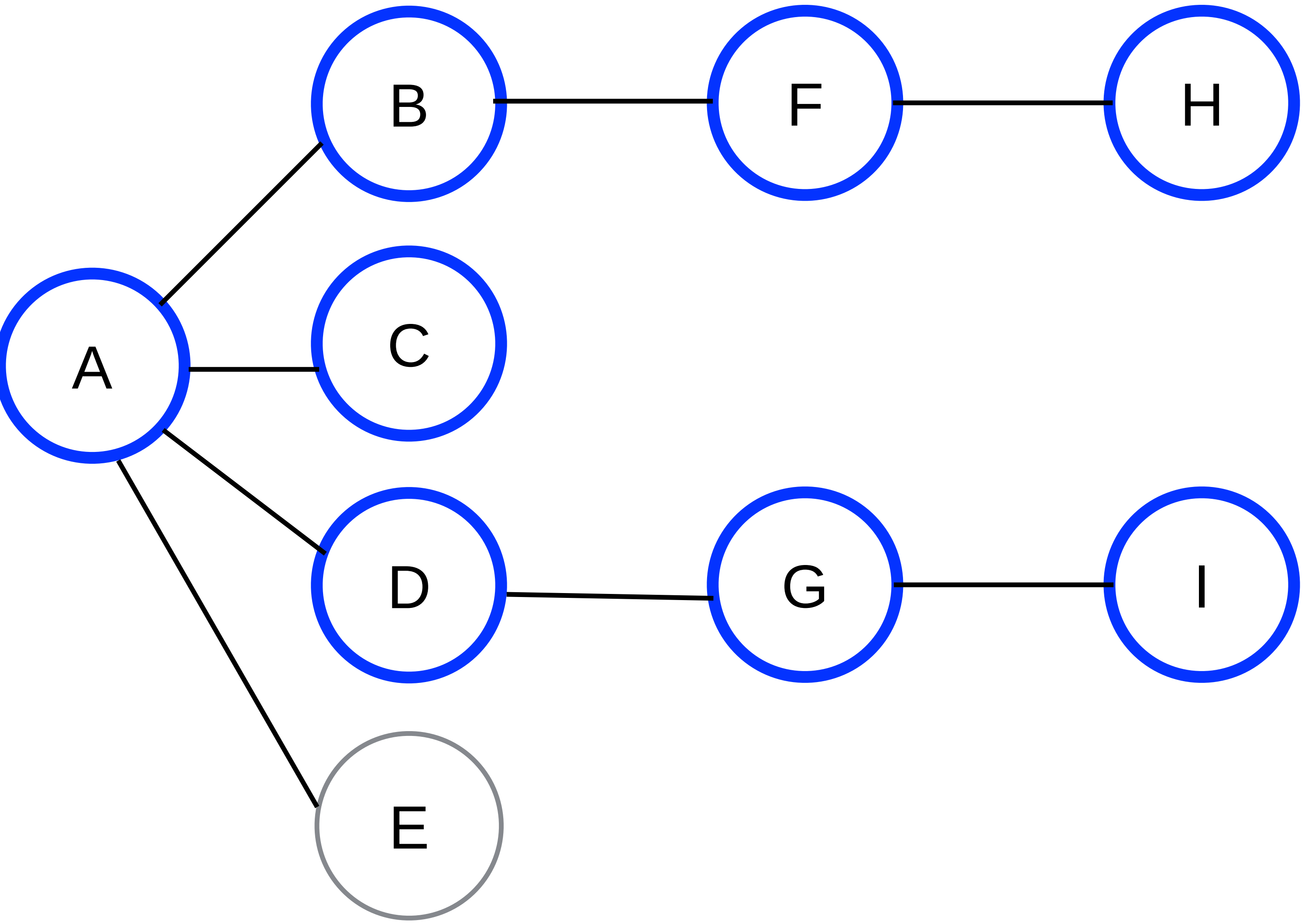
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI



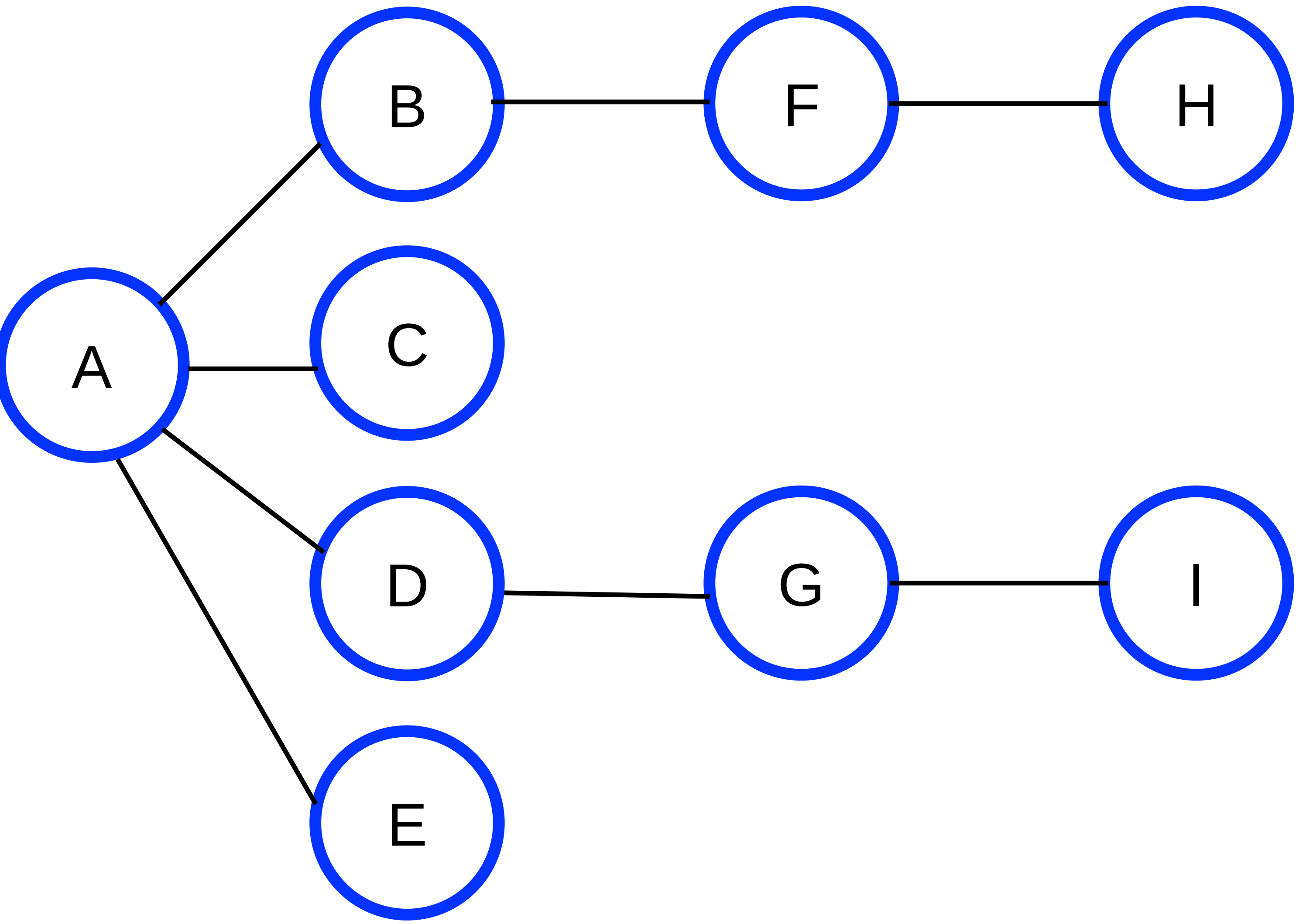
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG



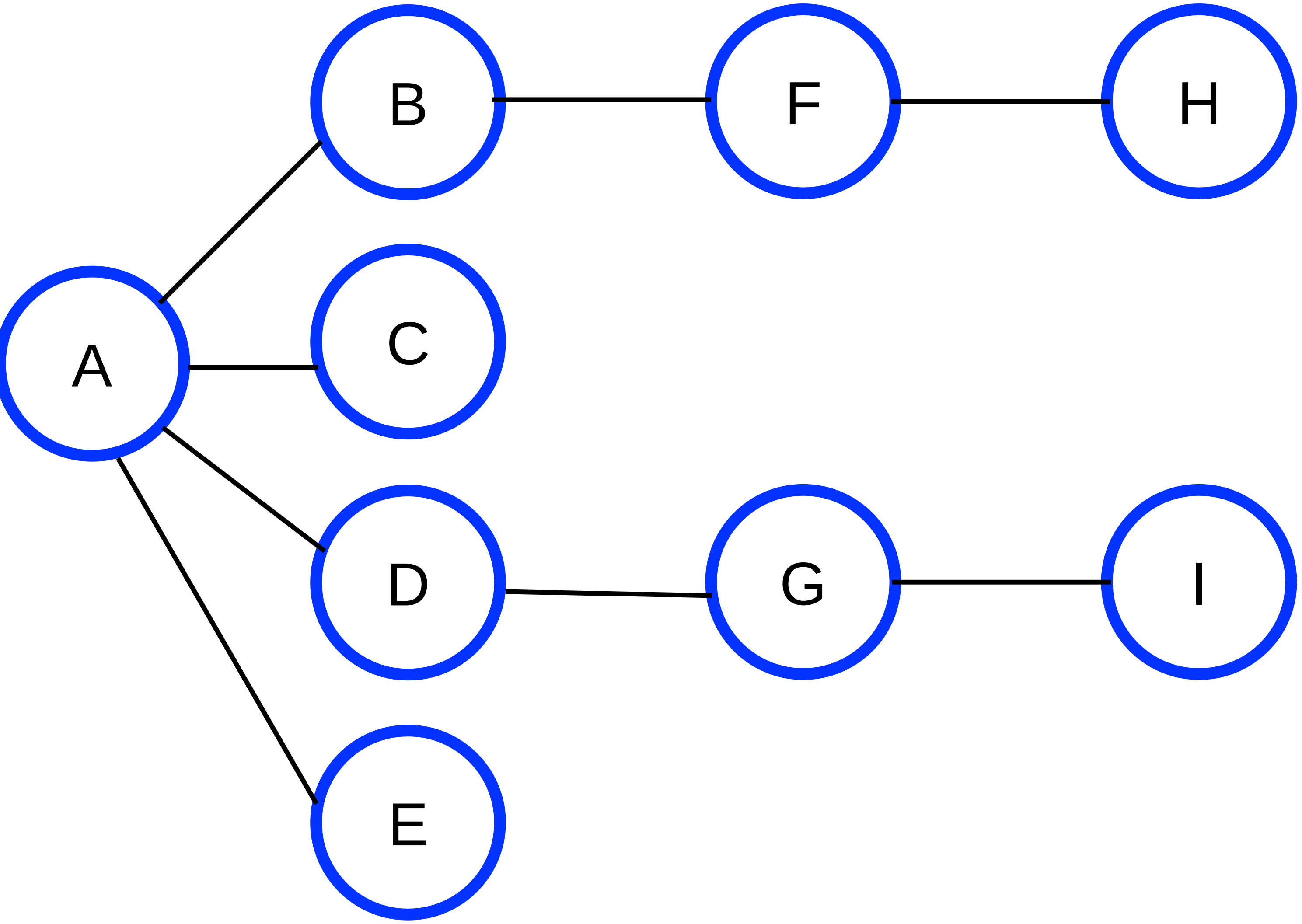
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD



Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A

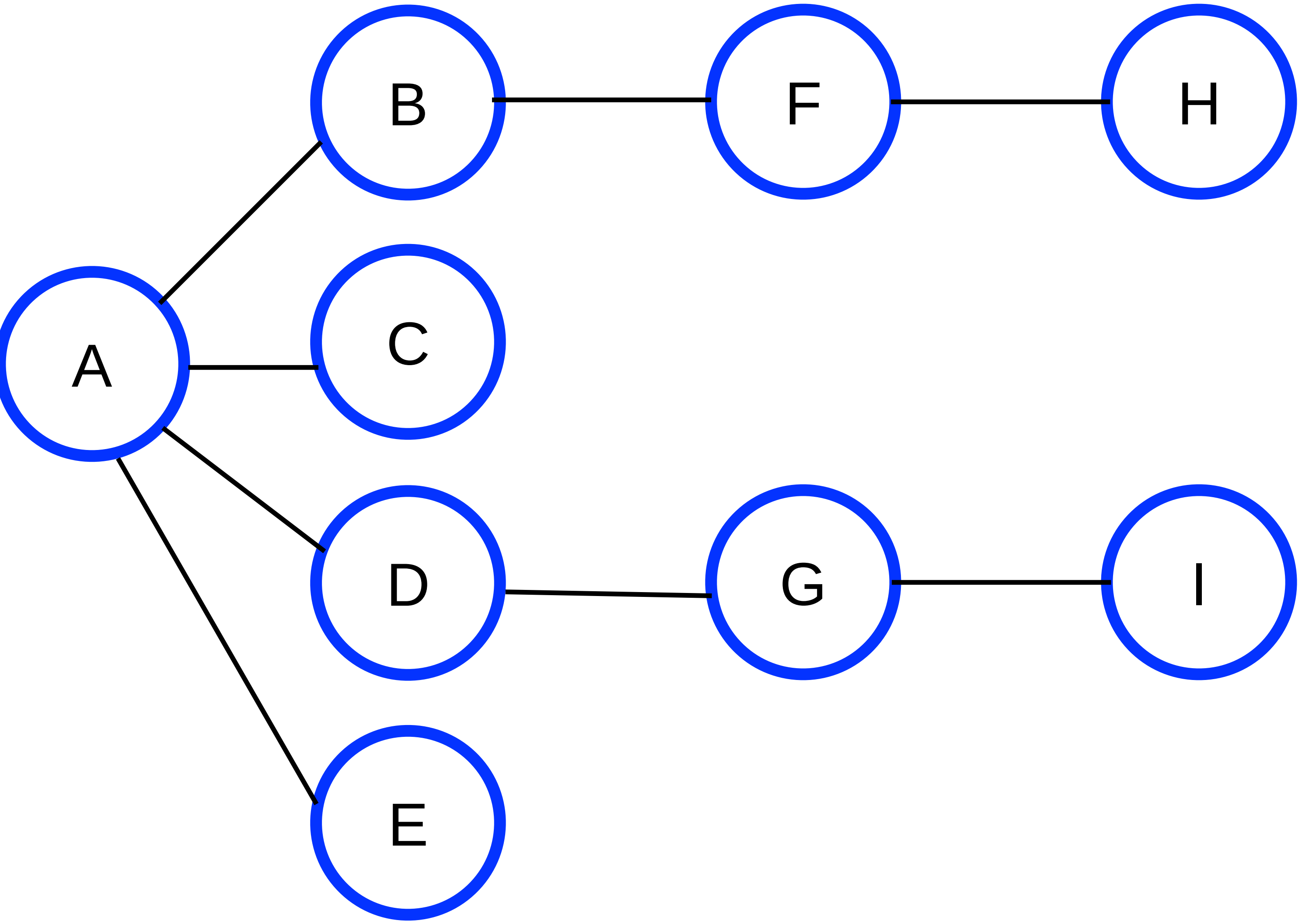


Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE



- At this point, A has no more adjacent unvisited vertices left
- We pop it off the stack

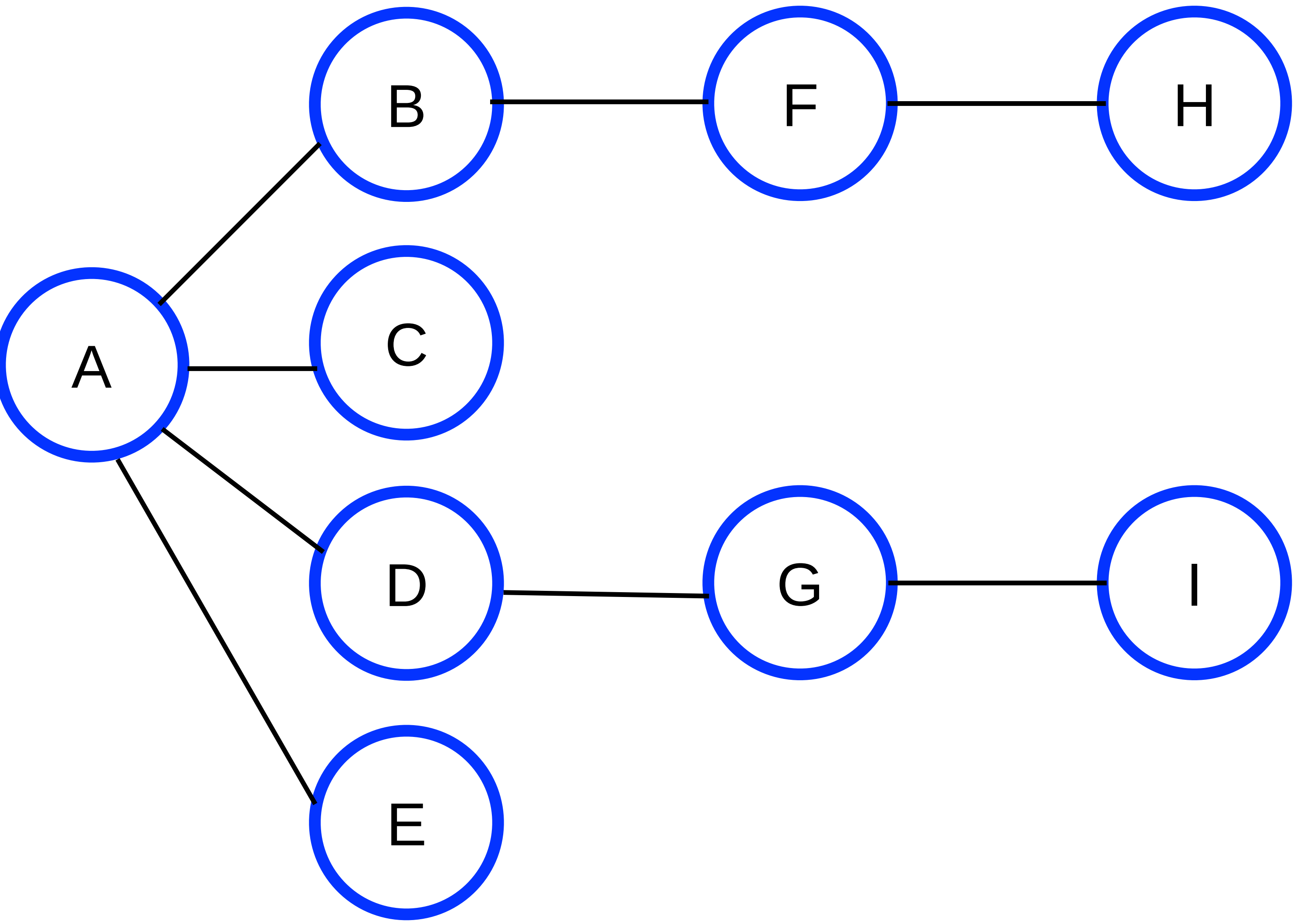
Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A



Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	

- This brings us to **Rule 3:**

“If you cannot follow Rule 1 or Rule 2, you are done”



Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	

Order: ABFHCDGIE

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

DFS

- Notice that,
  - DFS tries to get as far away from the starting point as quickly as possible
  - An returns only when it reaches a dead end
  - Thus the name, **Depth First Search**

DFS Implementation

```
// dfs.java
// demonstrates depth-first search
// to run this program: C>java DFSApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
   public StackX()            // constructor
      {
      st = new int[SIZE];    // make array
      top = -1;
      }
   public void push(int j)   // put item on stack
      { st[++top] = j; }
```

DFS Implementation (2)

```
   public int pop()           // take item off stack
      { return st[top--]; }
   public int peek()          // peek at top of stack
      { return st[top]; }
   public boolean isEmpty()   // true if nothing on stack
      { return (top == -1); }
   }  // end class StackX

////////////////////////////////////////////////////////////////

class Vertex
   {
   public char label;         // label (e.g. 'A')
   public boolean wasVisited;
```

DFS Implementation (3)

```
// ------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }

// ------------------
   }  // end class Vertex

////////////////////////////////////////////////////////////////

class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private StackX theStack;
```

DFS Implementation (4)

```
// ------------------
   public Graph()                    // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                        // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      theStack = new StackX();
      }  // end constructor

// ------------------
```

DFS Implementation (5)

```
public void addVertex(char lab)
   {
   vertexList[nVerts++] = new Vertex(lab);

    }

// -------------------
  public void addEdge(int start, int end)
     {
     adjMat[start][end] = 1;
     adjMat[end][start] = 1;
     }

// -------------------
  public void displayVertex(int v)
     {
     System.out.print(vertexList[v].label);
     }
```

DFS Implementation (6)

```
// -------------------
   public void dfs()  // depth-first search
      {                                       // begin at vertex 0
      vertexList[0].wasVisited = true;        // mark it
      displayVertex(0);                       // display it
      theStack.push(0);                       // push it

      while( !theStack.isEmpty() )            // until stack empty,
         {
         // get an unvisited vertex adjacent to stack top
         int v = getAdjUnvisitedVertex( theStack.peek() );
         if(v == -1)                          // if no such vertex,
            theStack.pop();
```

DFS Implementation (7)

```
    else                                // if it exists,
       {
       vertexList[v].wasVisited = true;  // mark it
       displayVertex(v);                 // display it
       theStack.push(v);                 // push it
       }
    }  // end while

    // stack is empty, so we're done
    for(int j=0; j<nVerts; j++)          // reset flags
       vertexList[j].wasVisited = false;
 }  // end dfs
```

DFS Implementation (8)

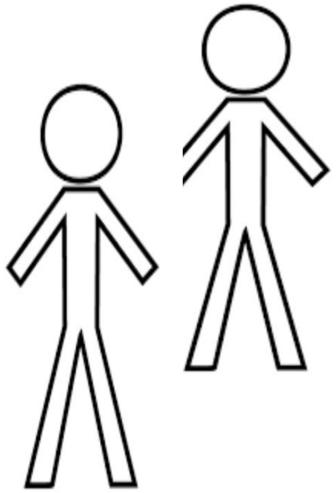
```
// ------------------
   // returns an unvisited vertex adj to v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;
      return -1;
      }  // end getAdjUnvisitedVert()

// ------------------


}  // end class Graph

////////////////////////////////////////////////////////////////
```

DFS Implementation (9)

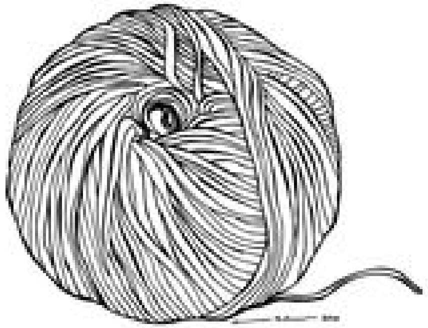
```
class DFSApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (start for dfs)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4

      theGraph.addEdge(0, 1);     // AB
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(3, 4);     // DE

      System.out.print("Visits: ");
      theGraph.dfs();             // depth-first search
      System.out.println();
      }  // end main()
   }  // end class DFSApp
```

Breadth First Search BFS (1)

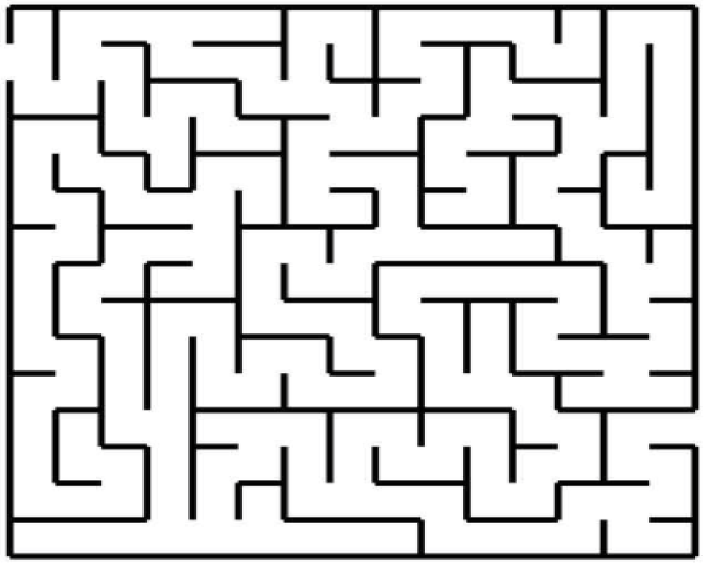
- Searching in a maze



Group of Searchers

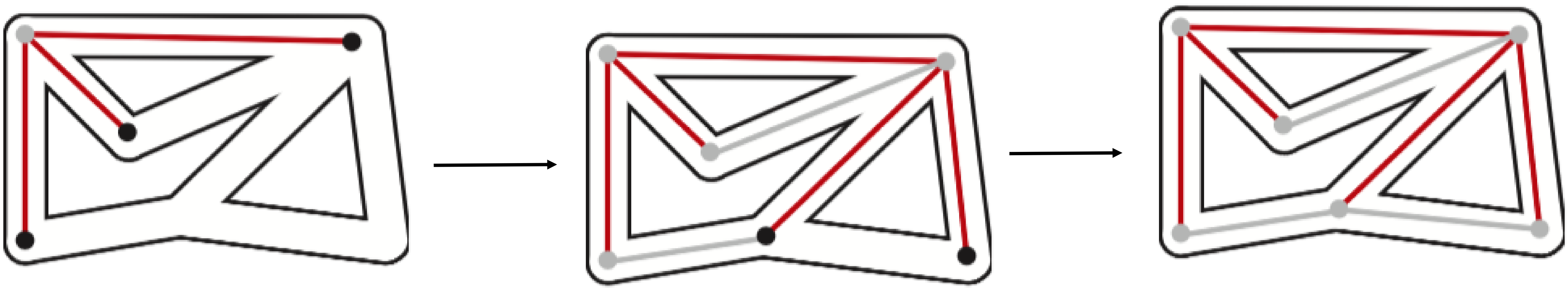


String



Maze

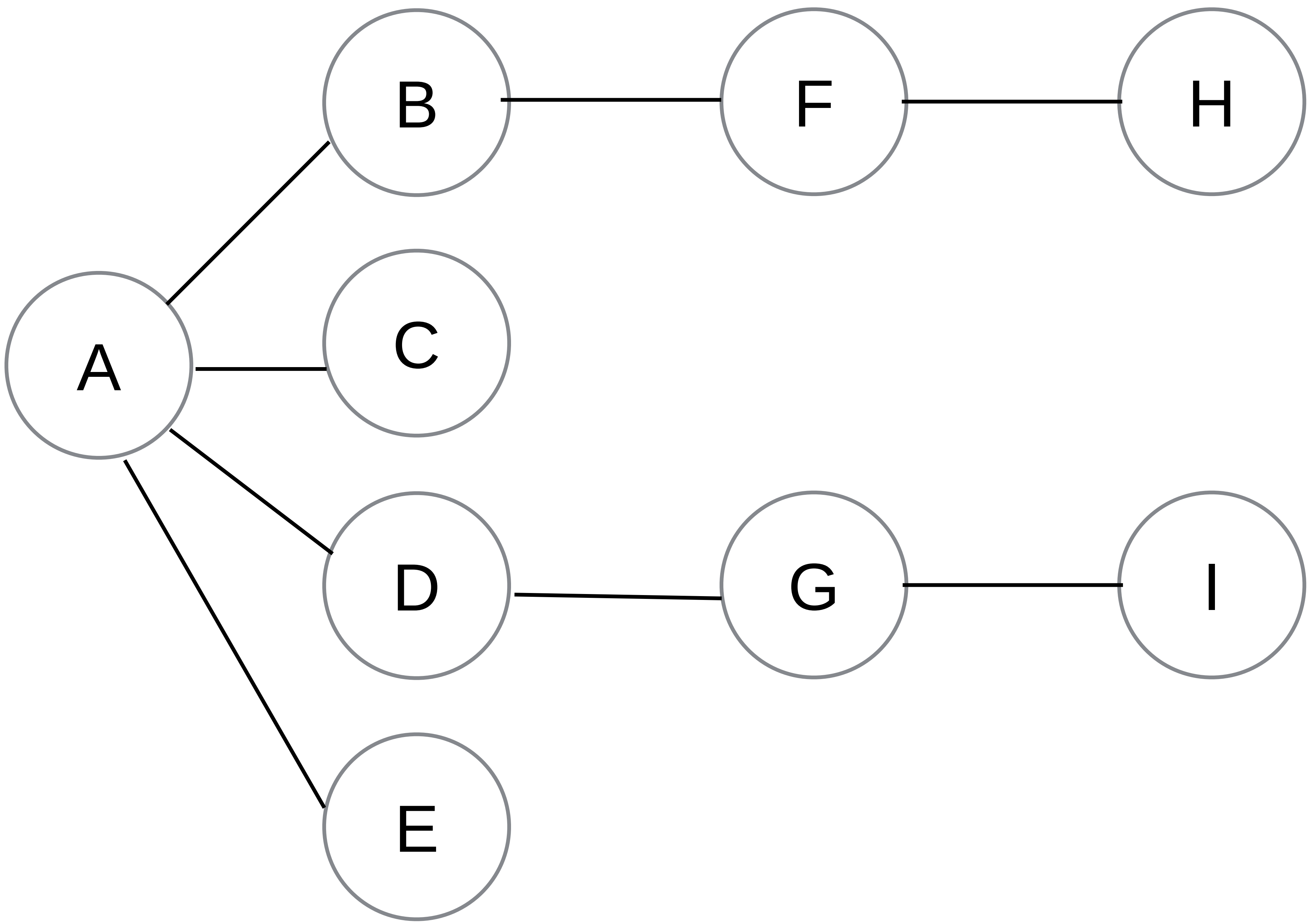
BFS(2)



a b c

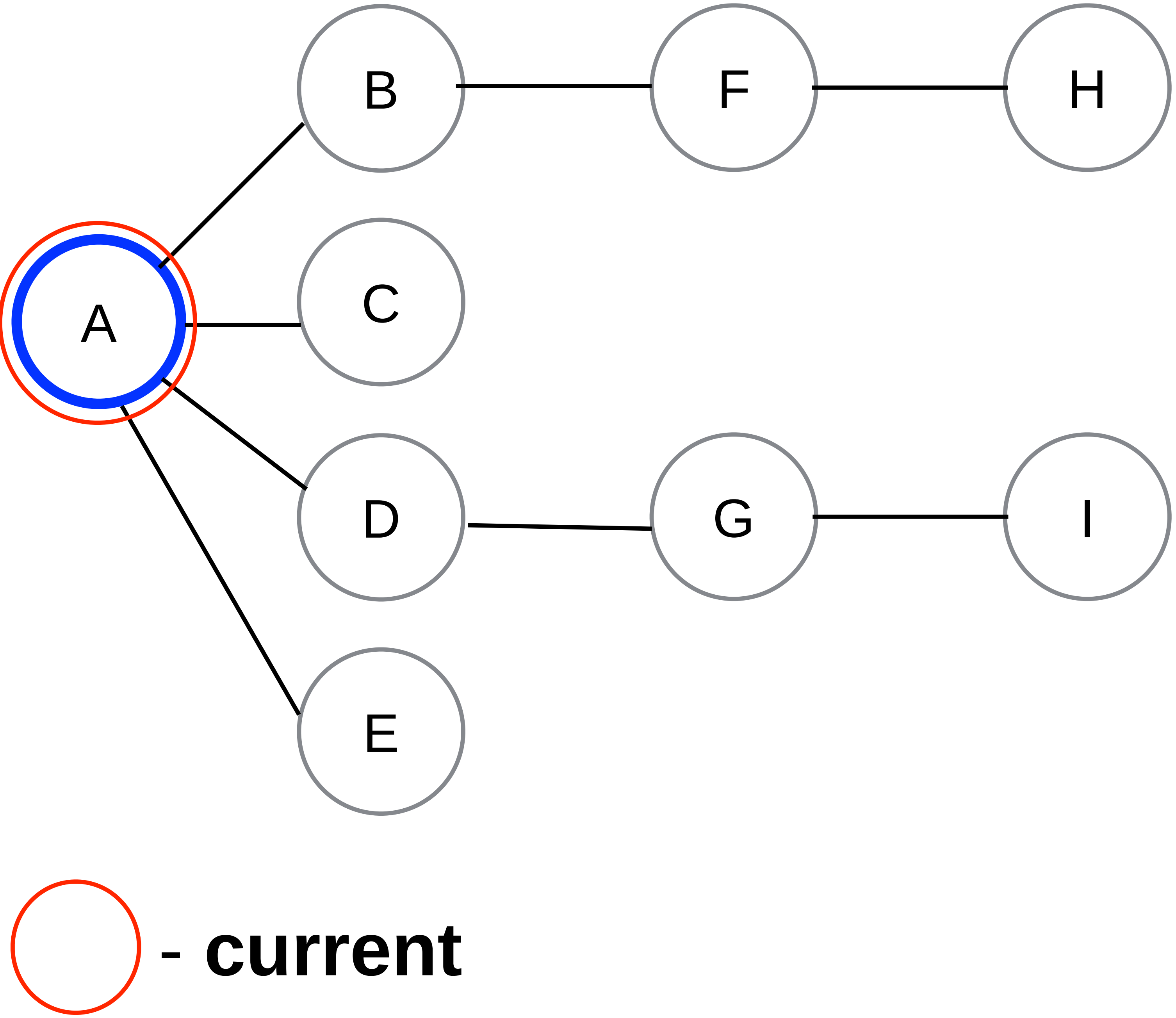
BFS Maze Exploration

BFS with a Queue

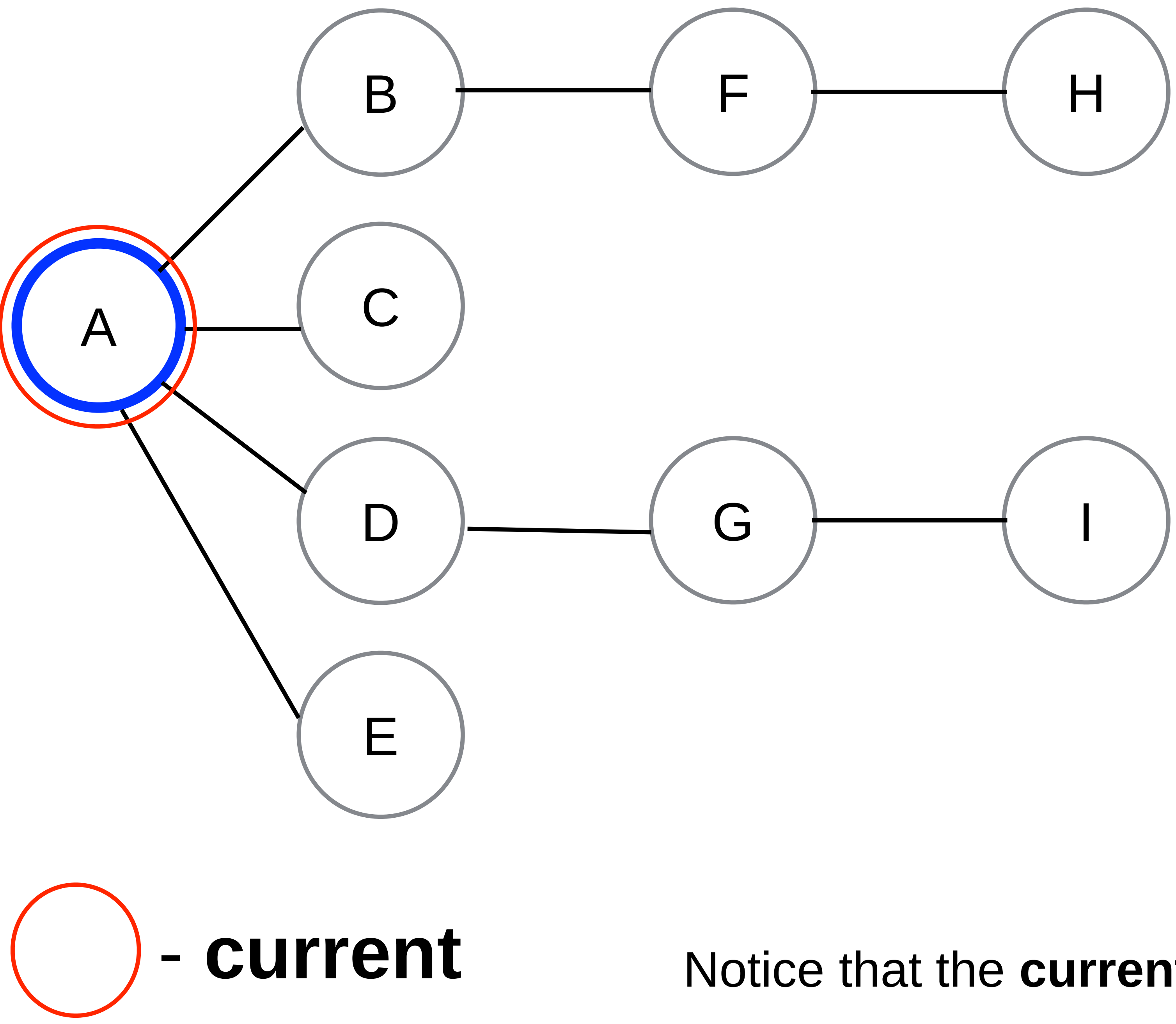


BFS with a Queue (2)

- Start with a vertex, visit it, and call it **current**
- Let’s start with vertex A



Event	Queue
Visit A	

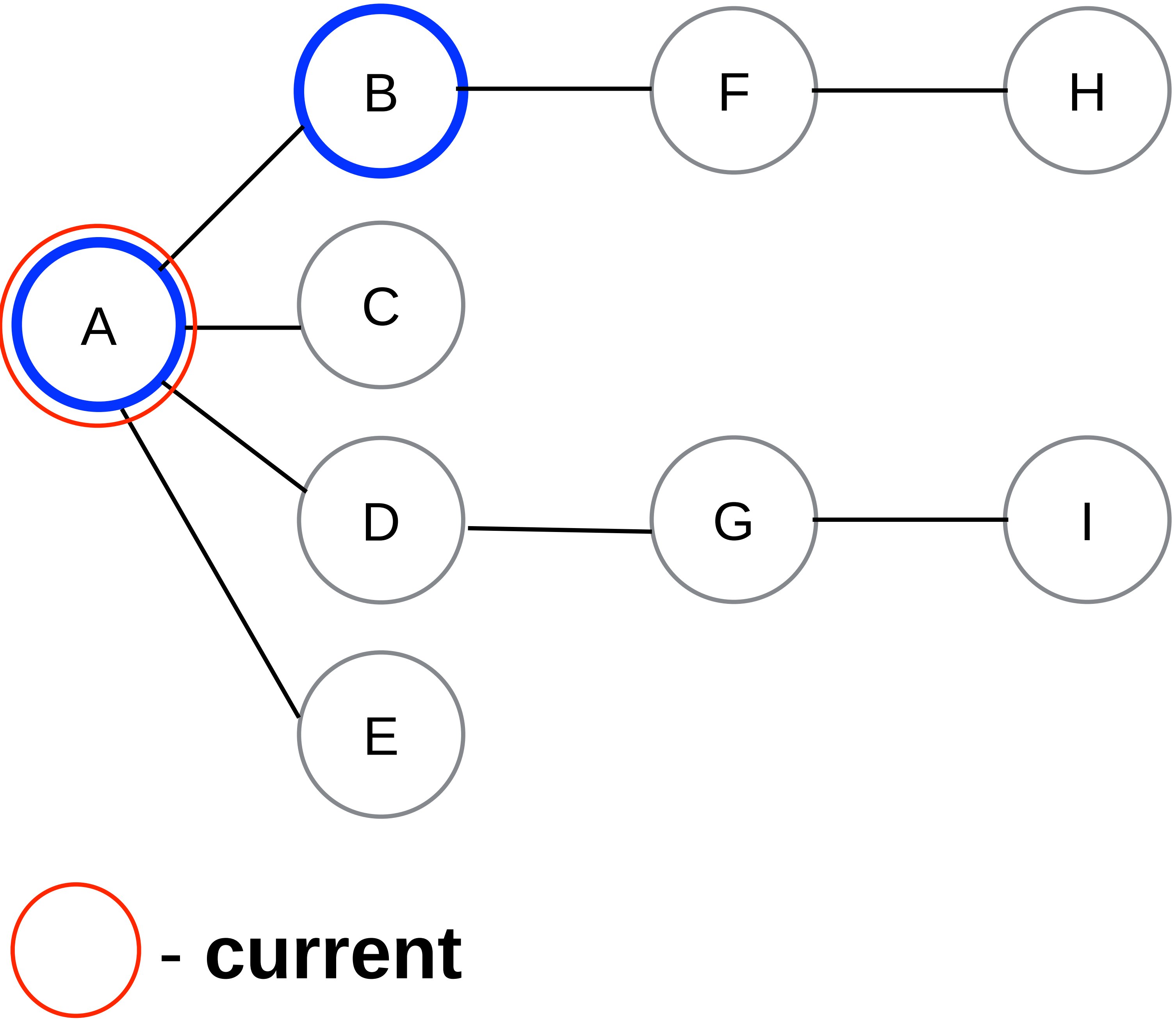


Event	Queue
Visit A	

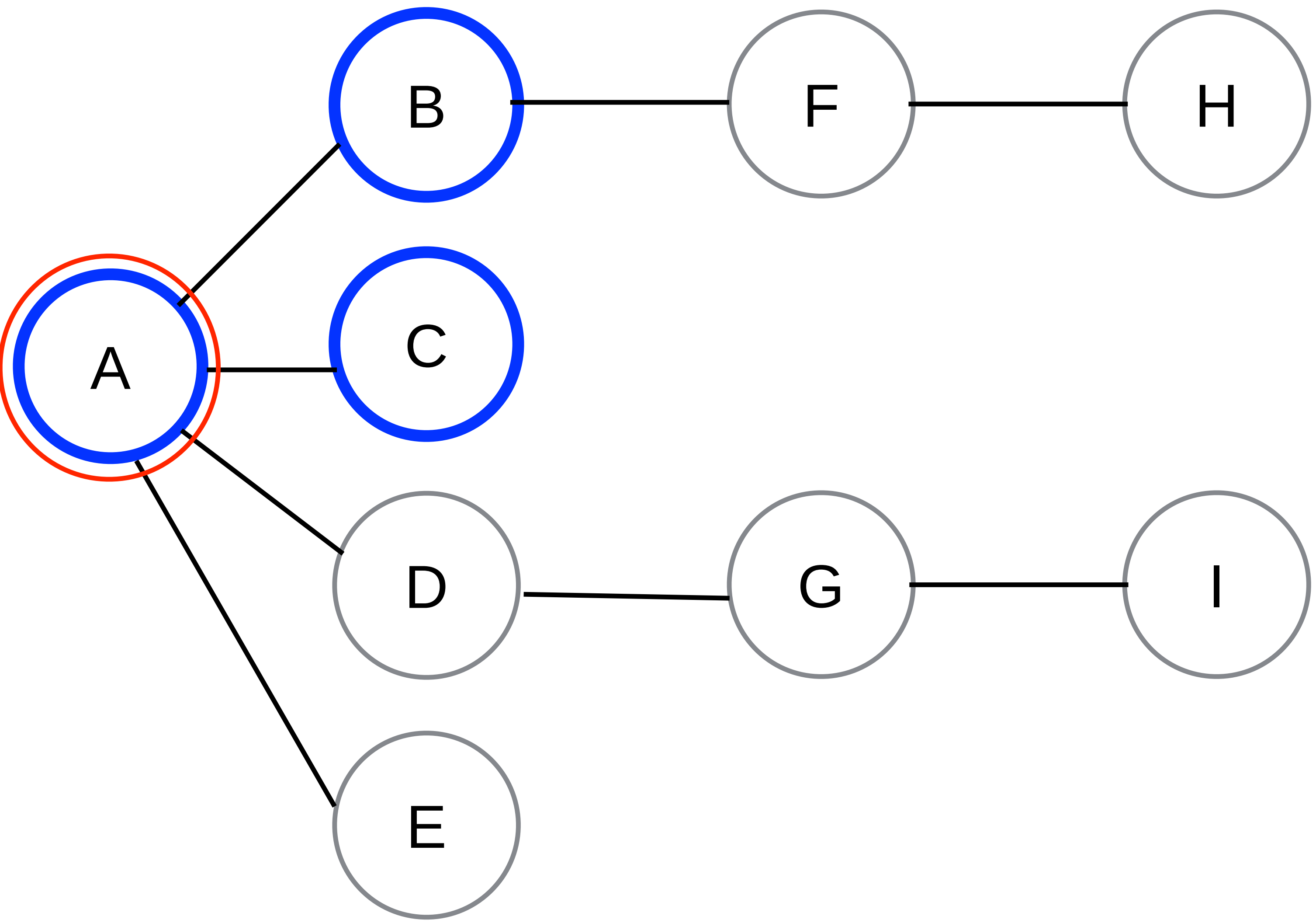
Notice that the **current** is not inserted into the queue

Now follow this rule

Rule 1: Visit the next unvisited vertex (if there is one) that is adjacent to the **current** vertex, mark it, and insert it into the queue

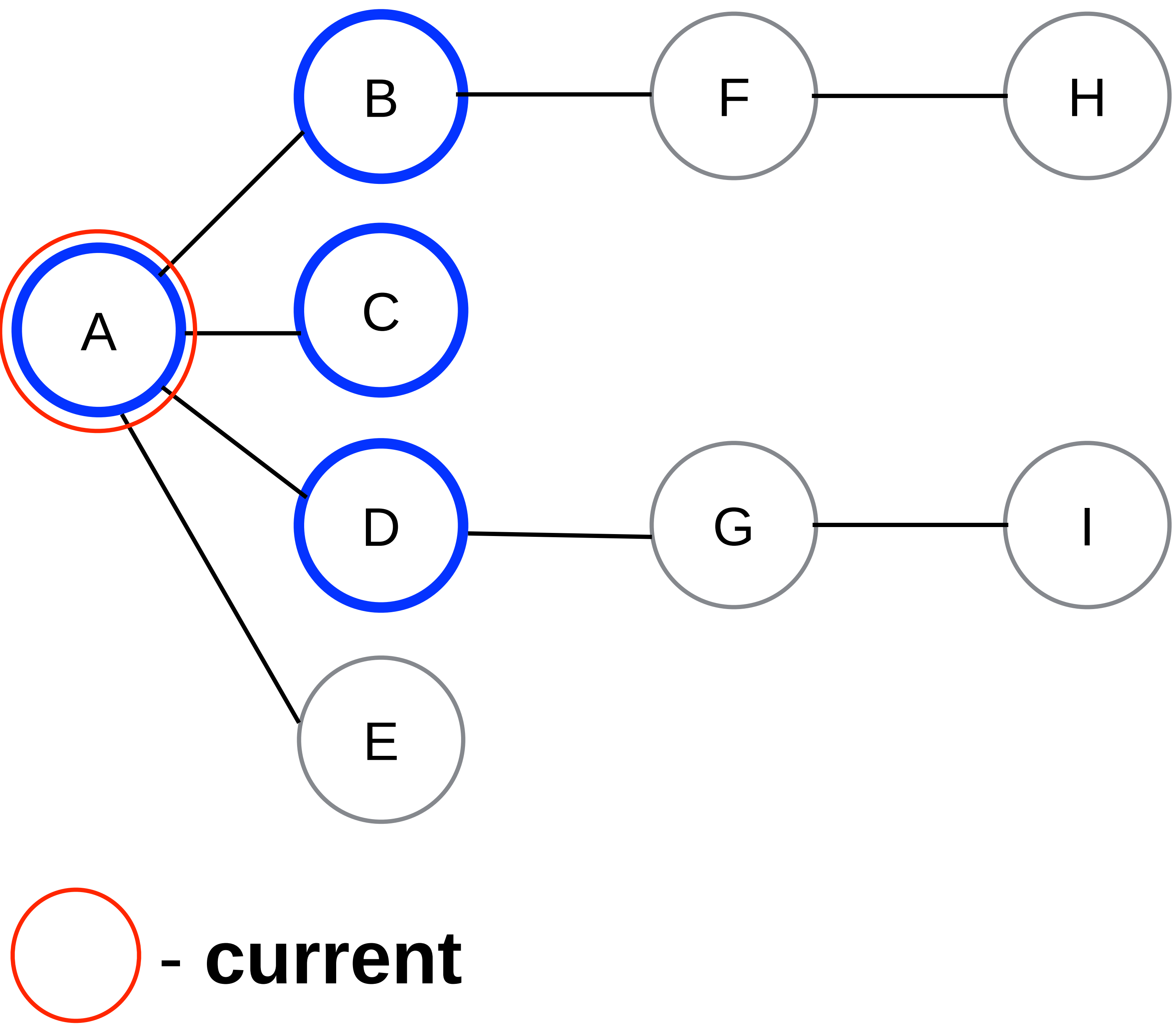


Event	**Queue**
Visit A	
Visit B	B

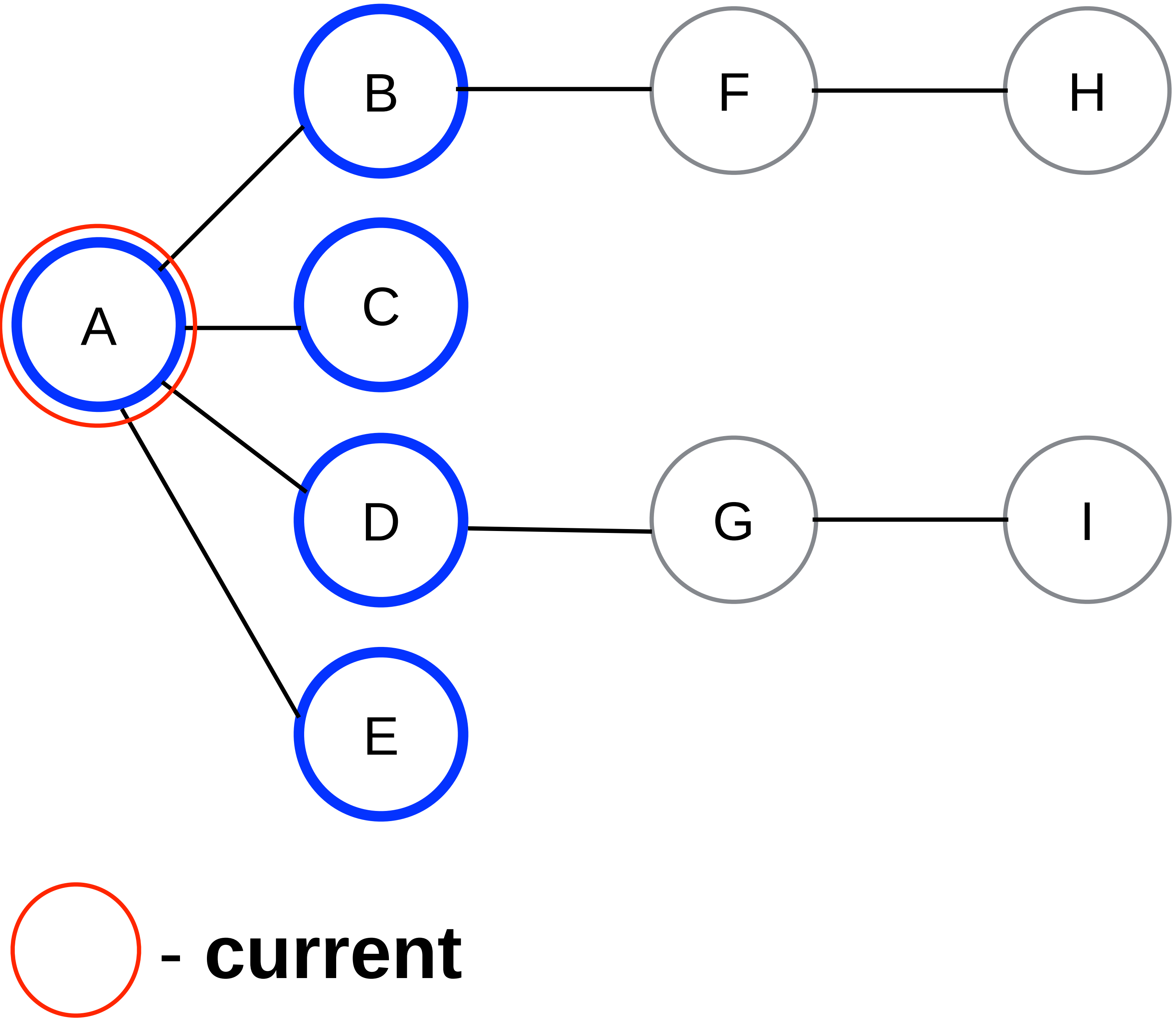


Event	**Queue**
Visit A	
Visit B	B
Visit C	BC

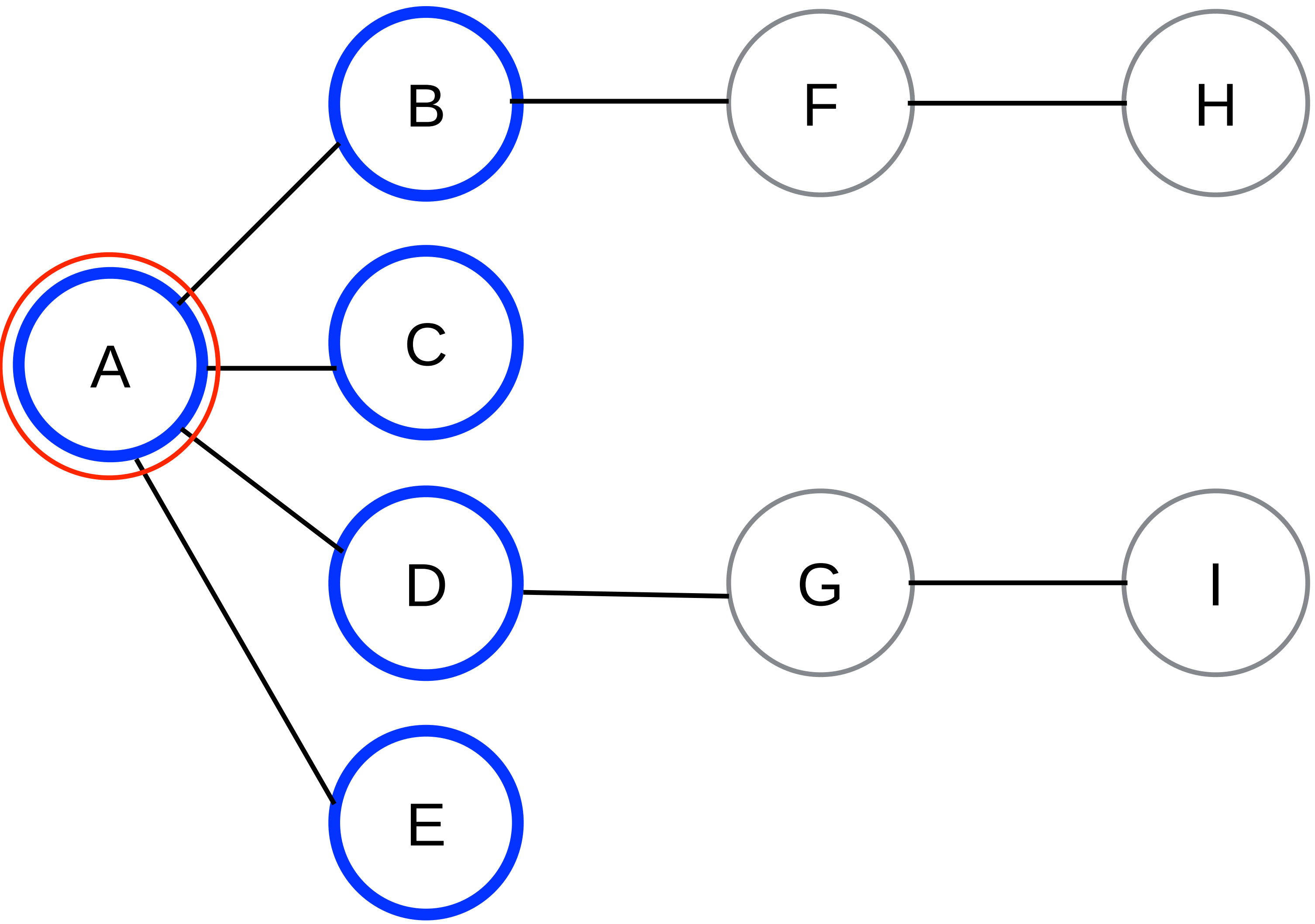
- current



Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD



Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE

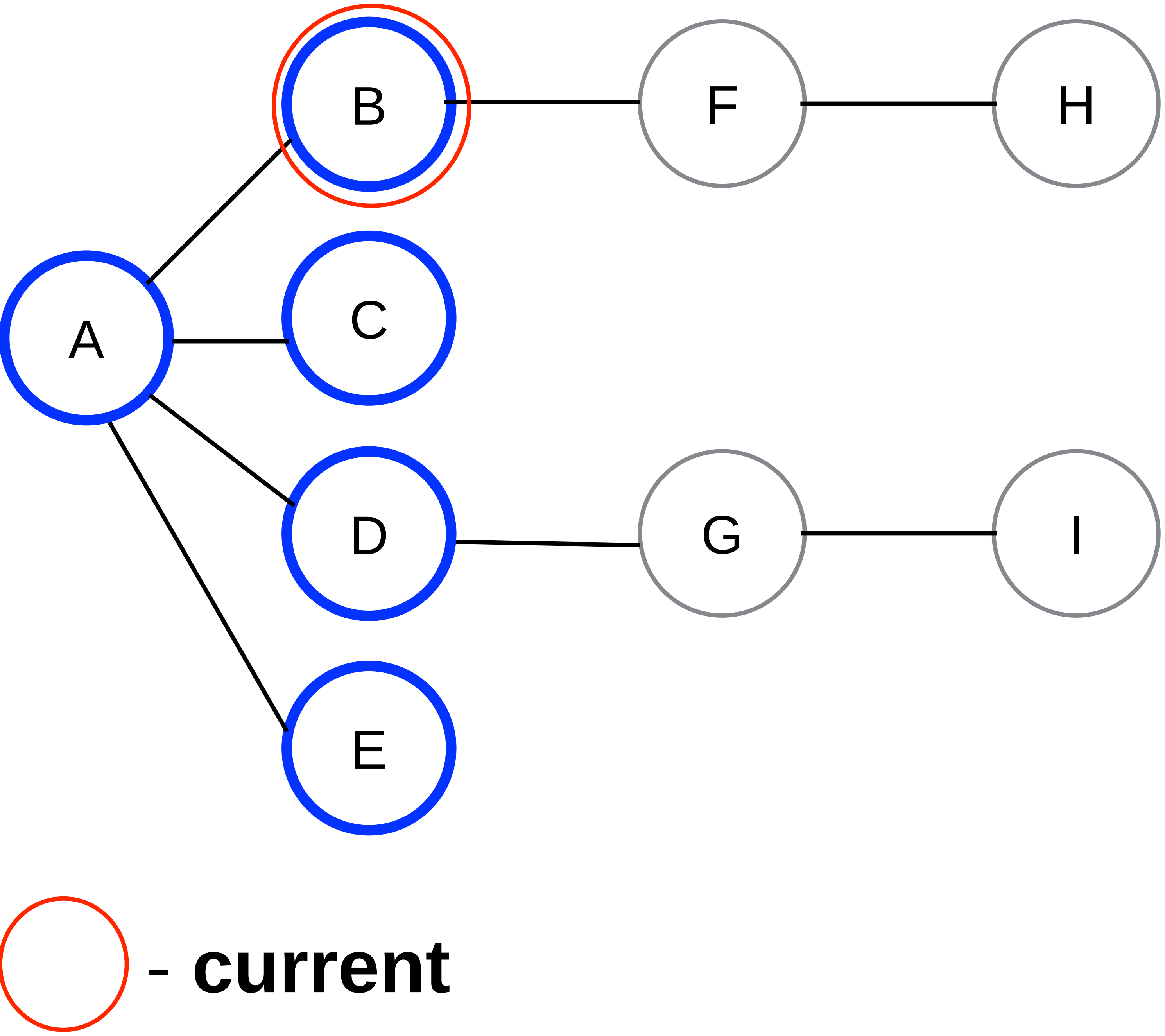


Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE

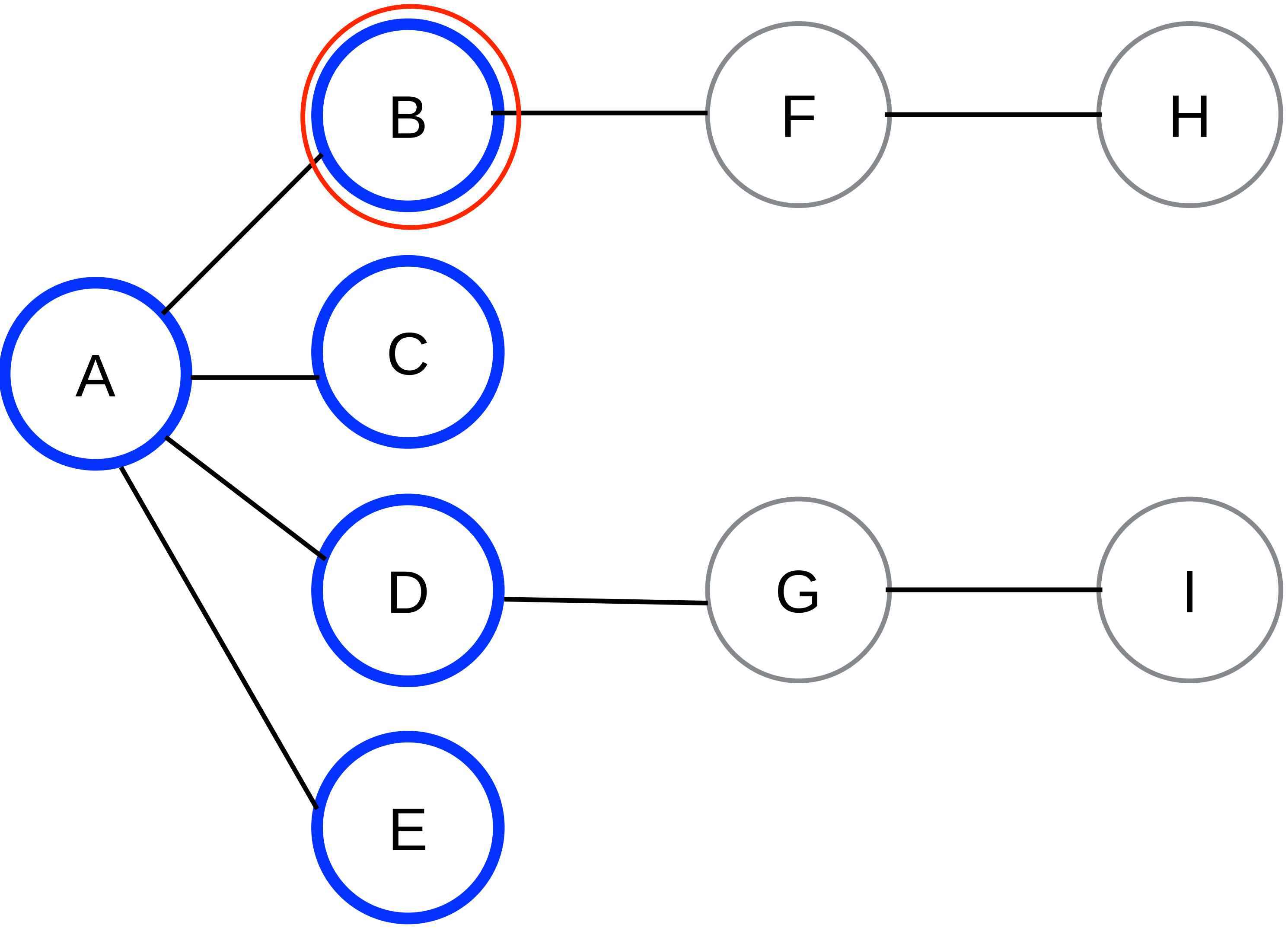
- **current**

At this point A (**the current**) has no more unvisited adjacent vertex
So, follow **Rule 2**:

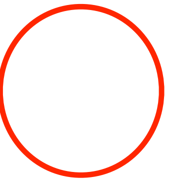
If you can’t carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it **current** vertex



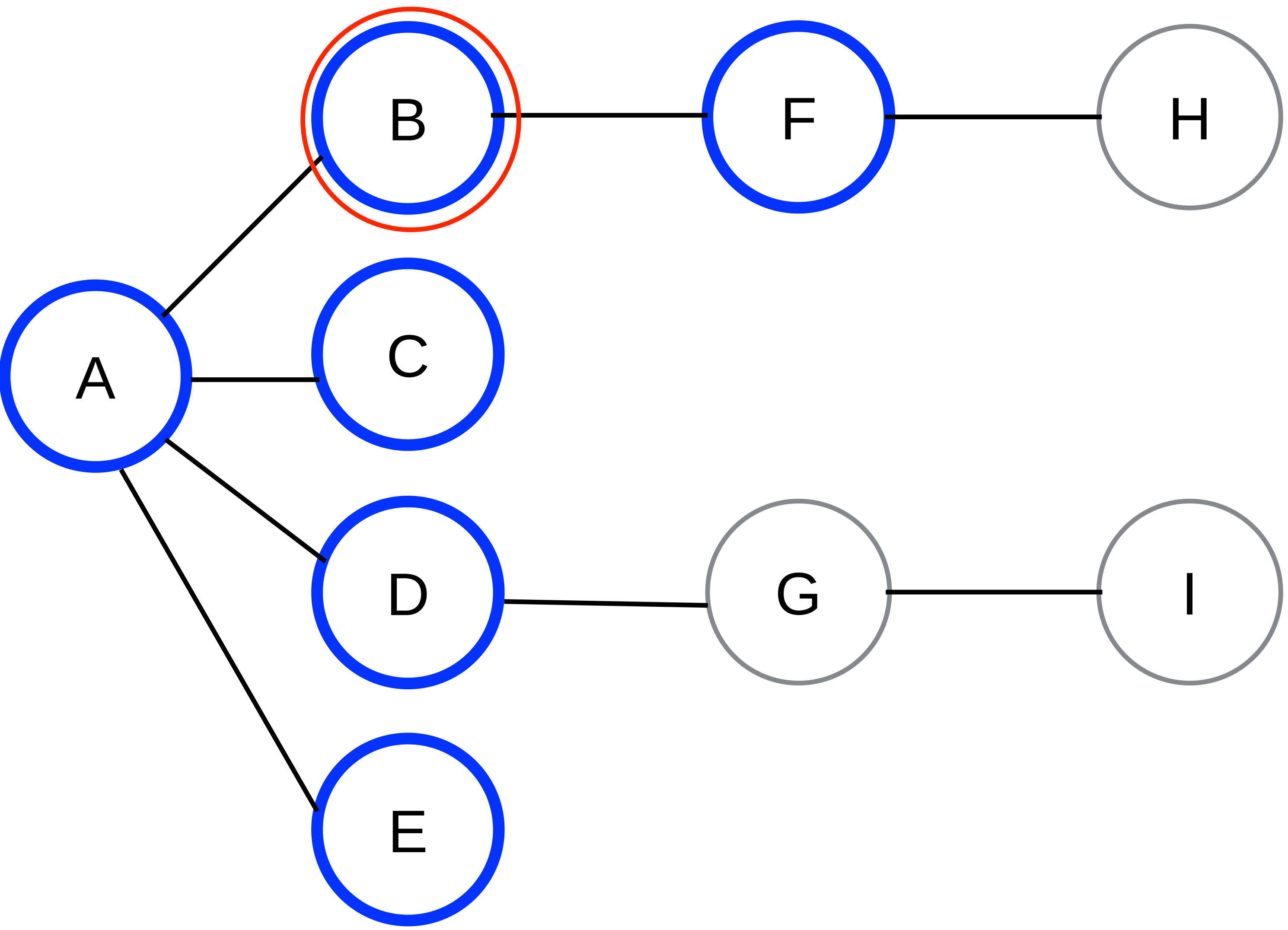
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE



Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE

 - **current**

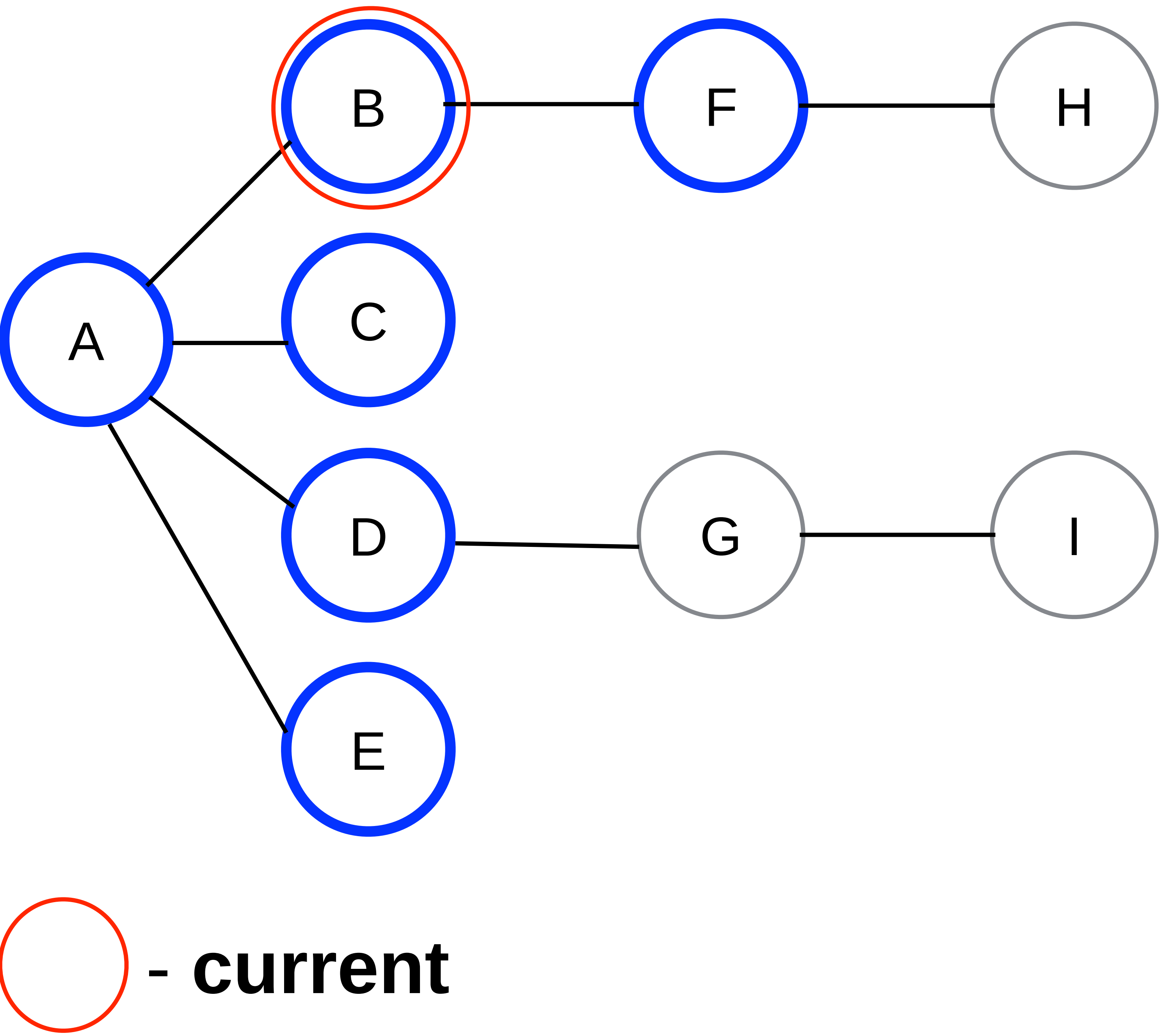
Repeat Rule 1 for the new **current**



Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF

- **current**

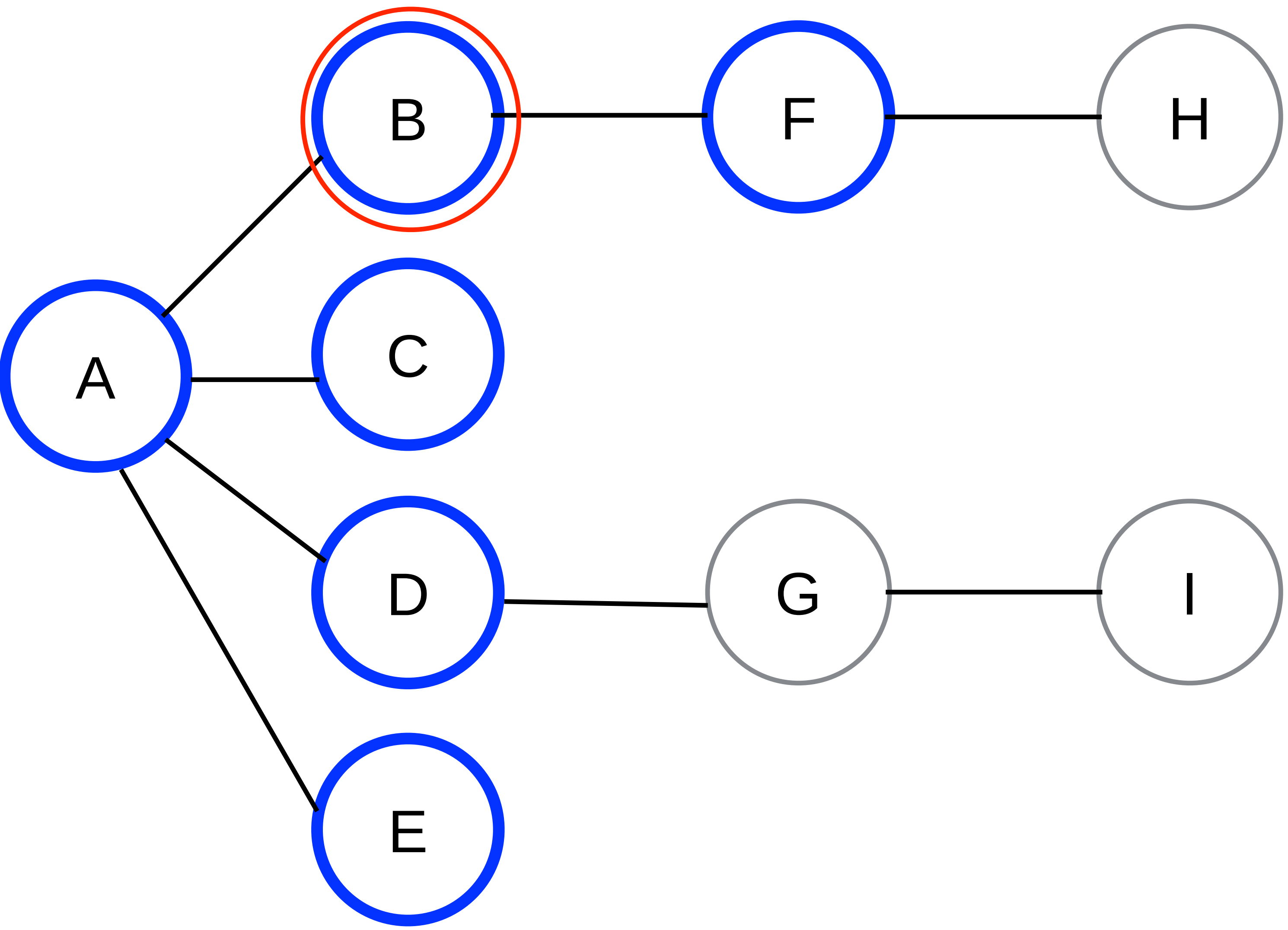
Will we follow BA?



Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF

Will we follow BA?

Yes! But it will take us back to A, which is already visited!



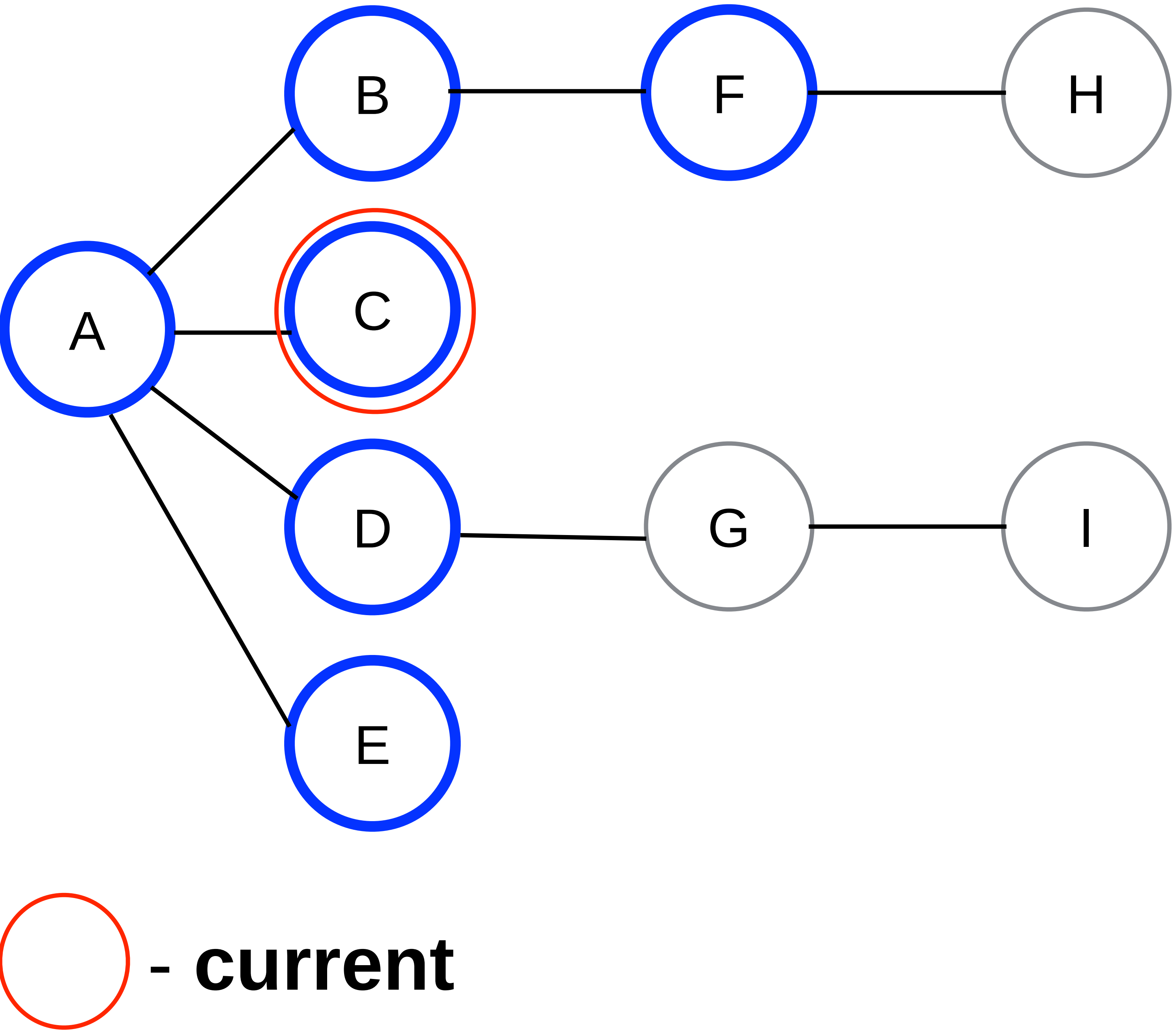
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF

- **current**

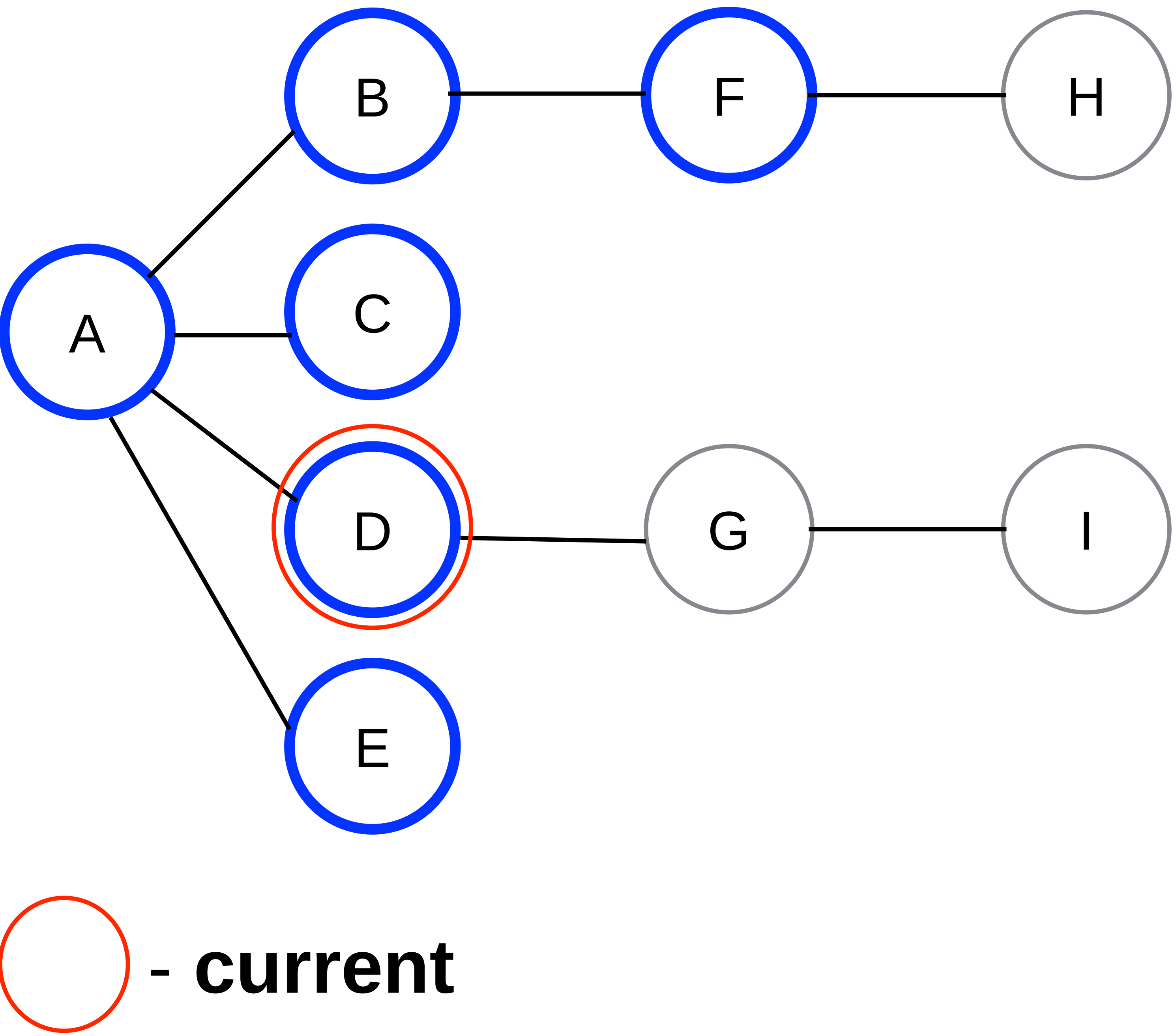
Will we follow BA?

Yes! But it will take us back to A, which is already visited!

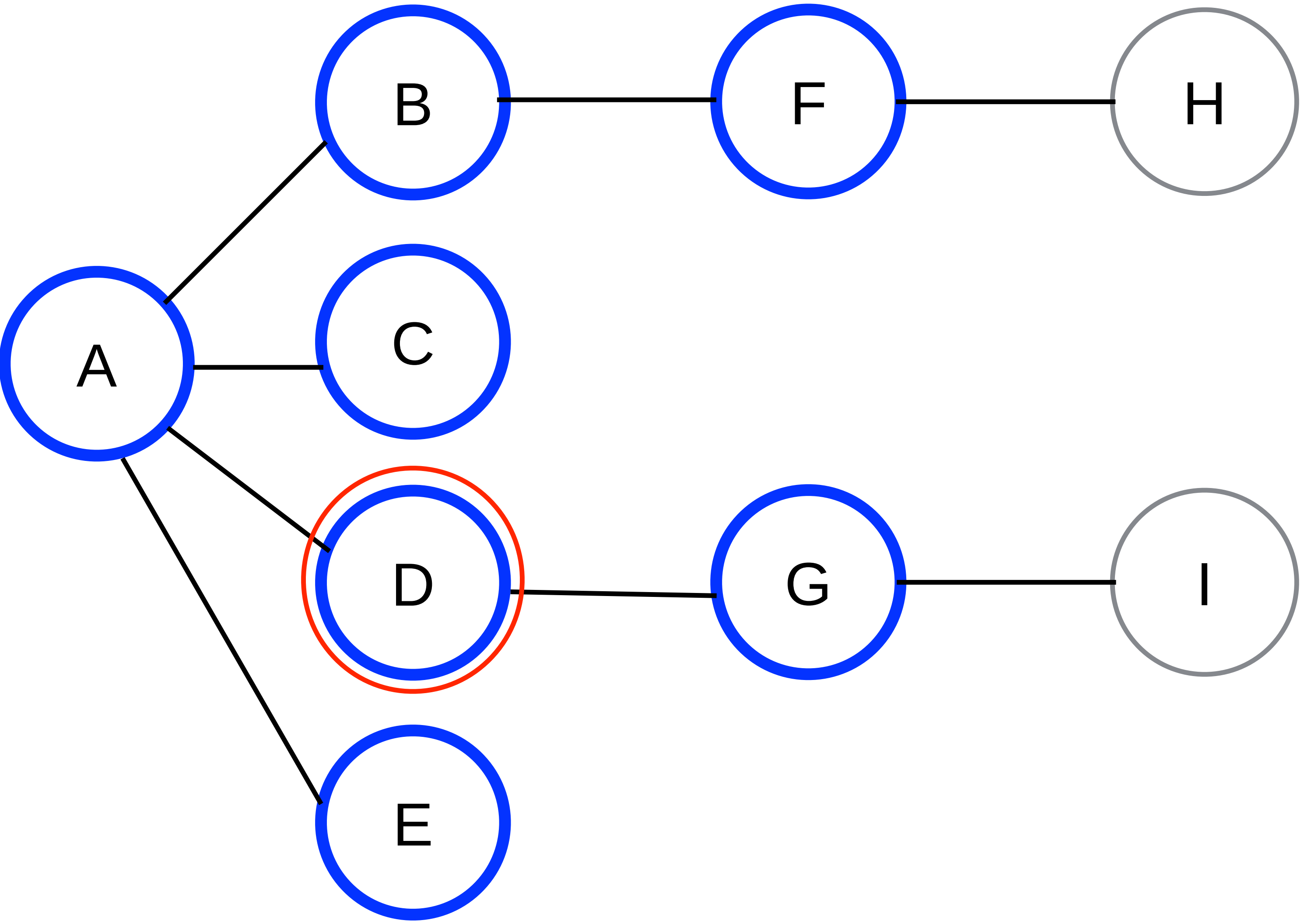
Thus each, vertex is visited once, and each edge twice!



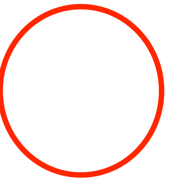
Event	**Queue**
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF



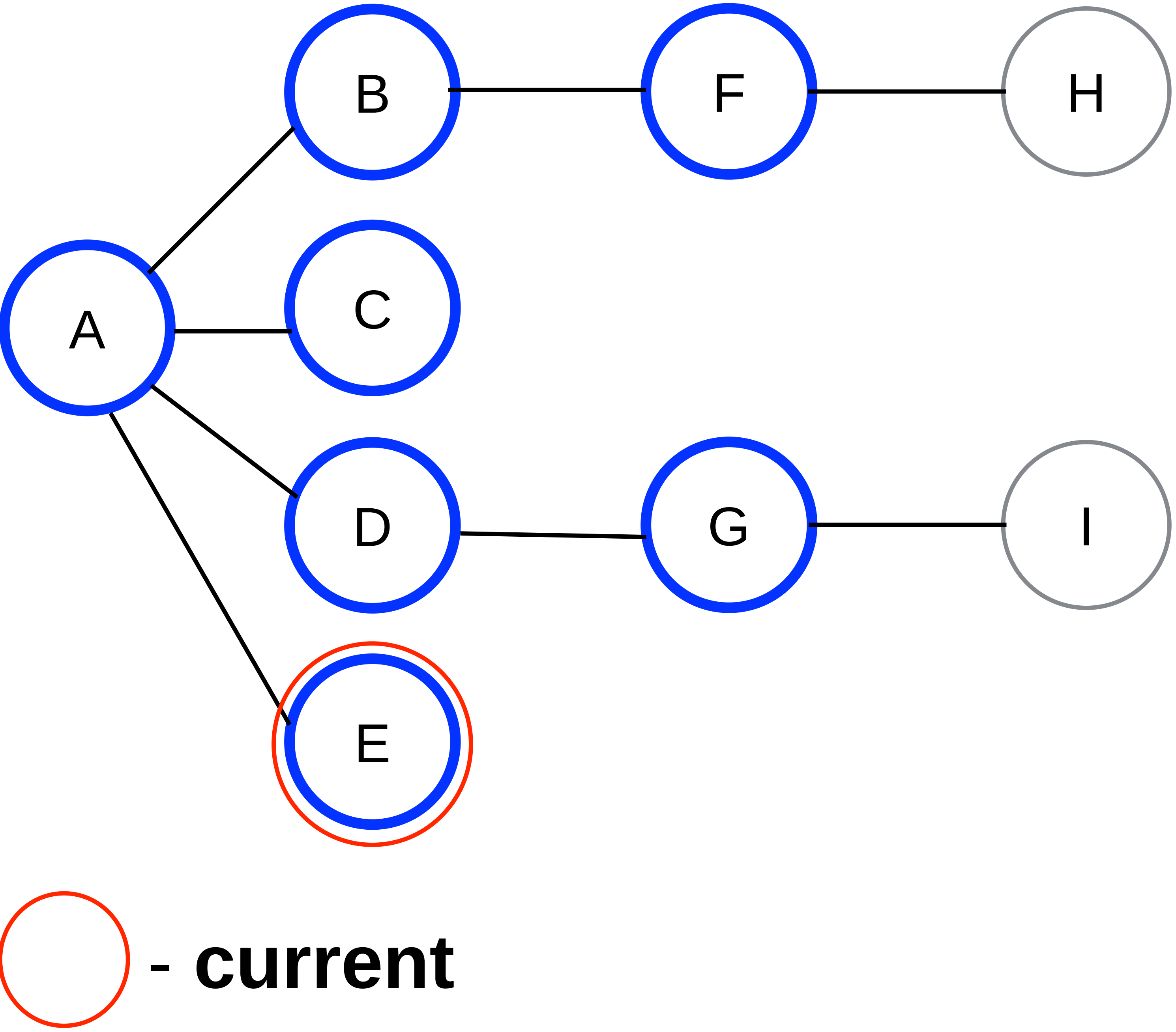
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF



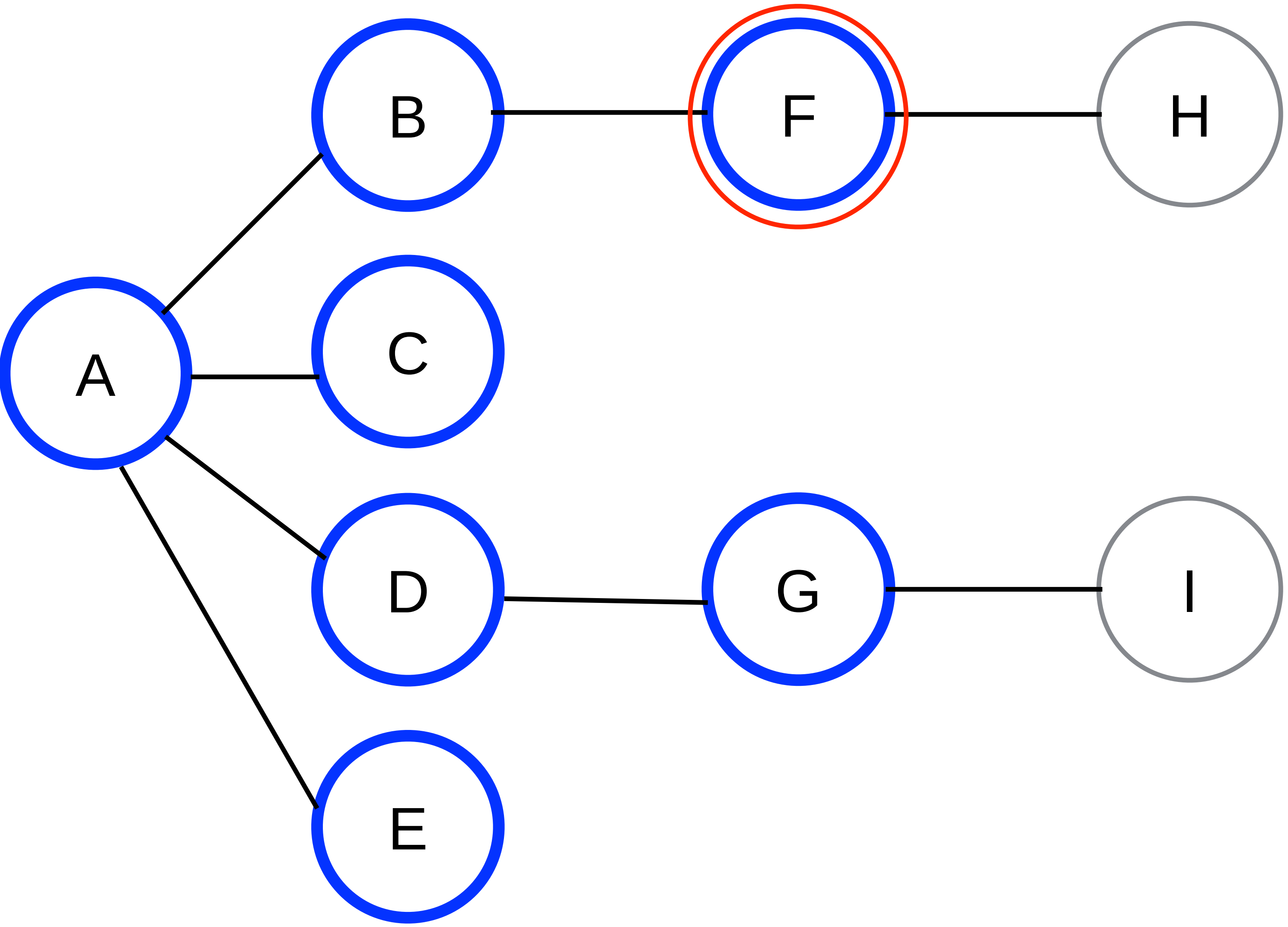
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG



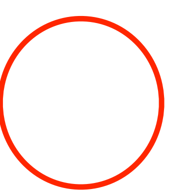
- **current**



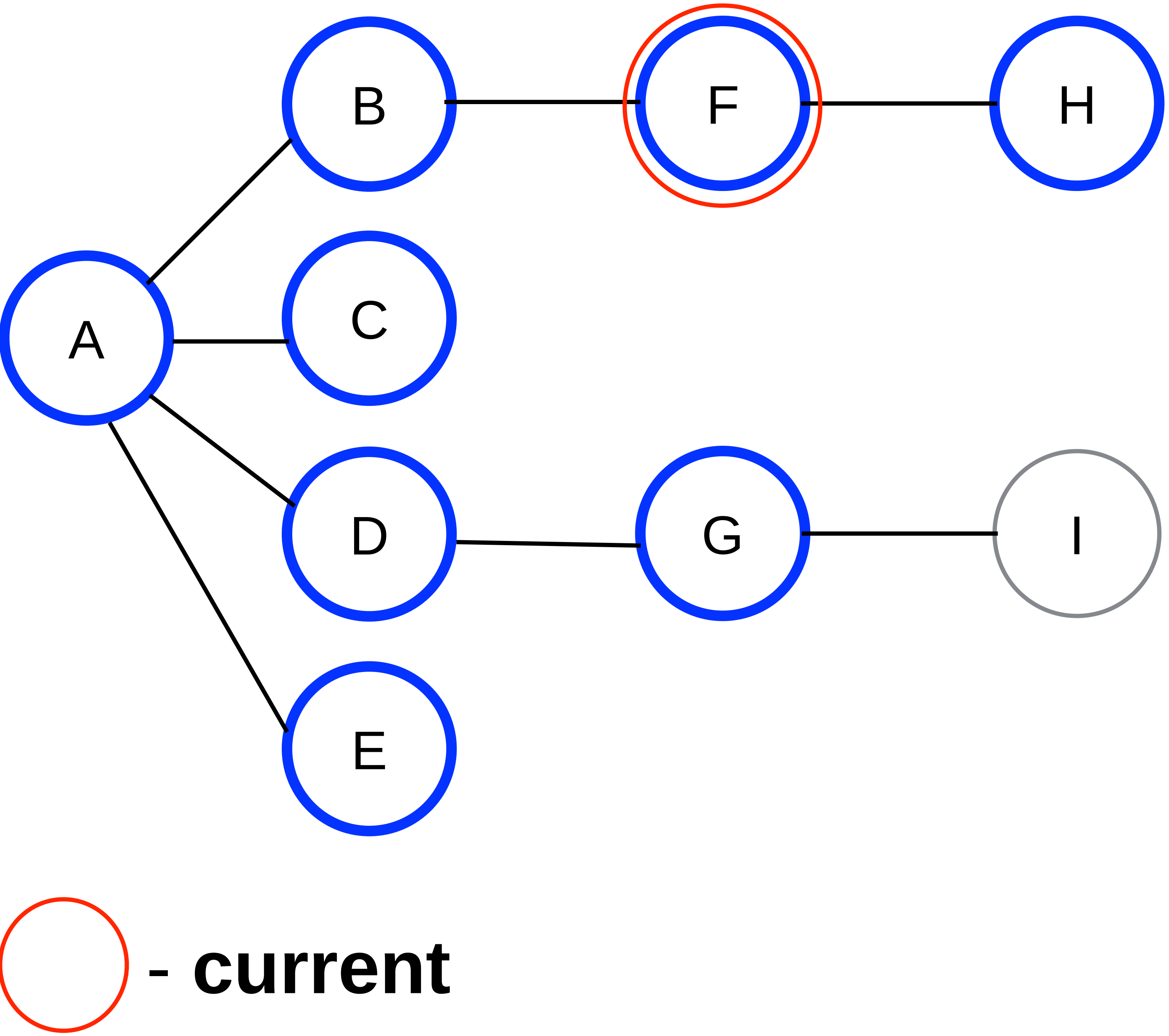
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG



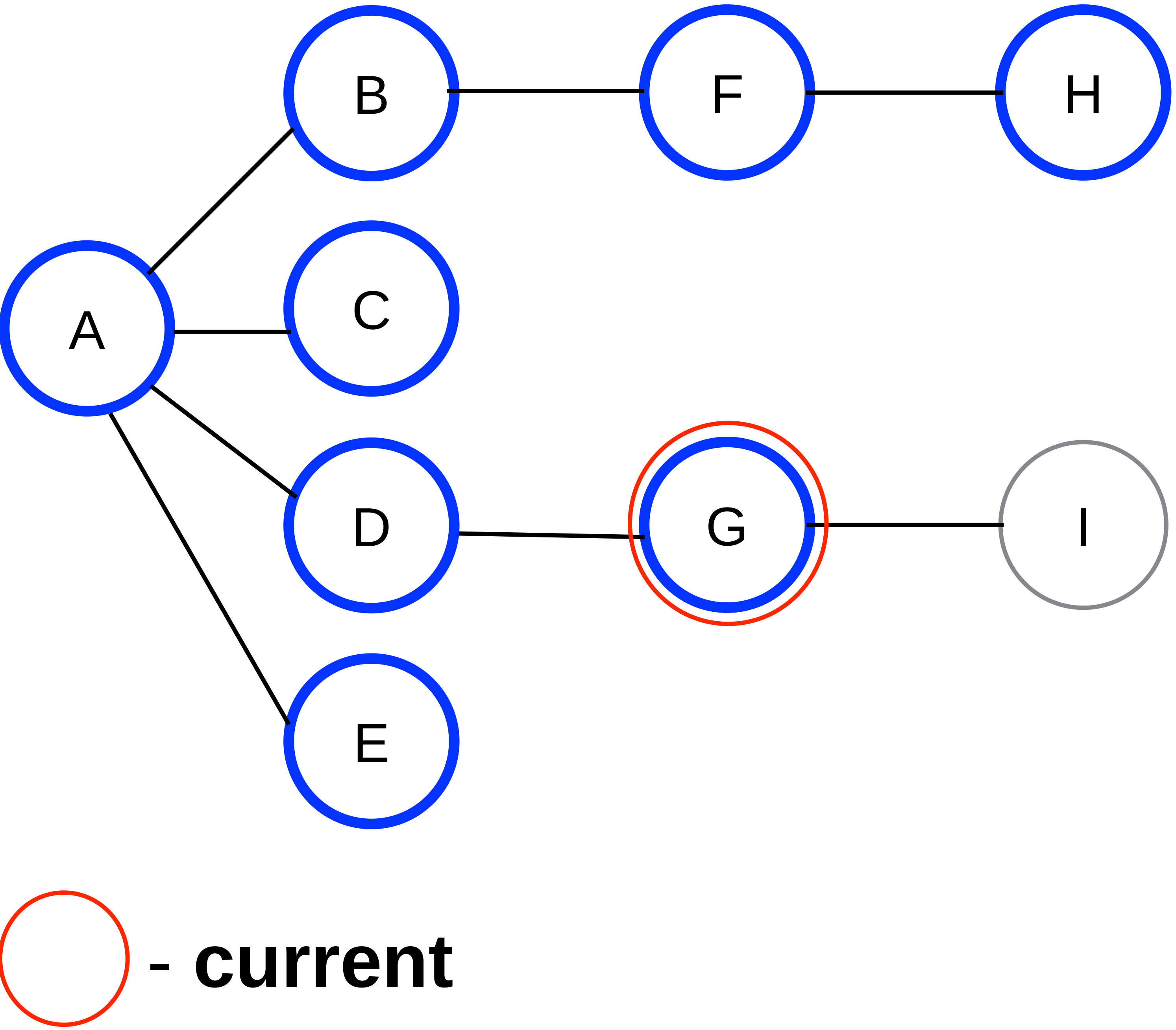
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G



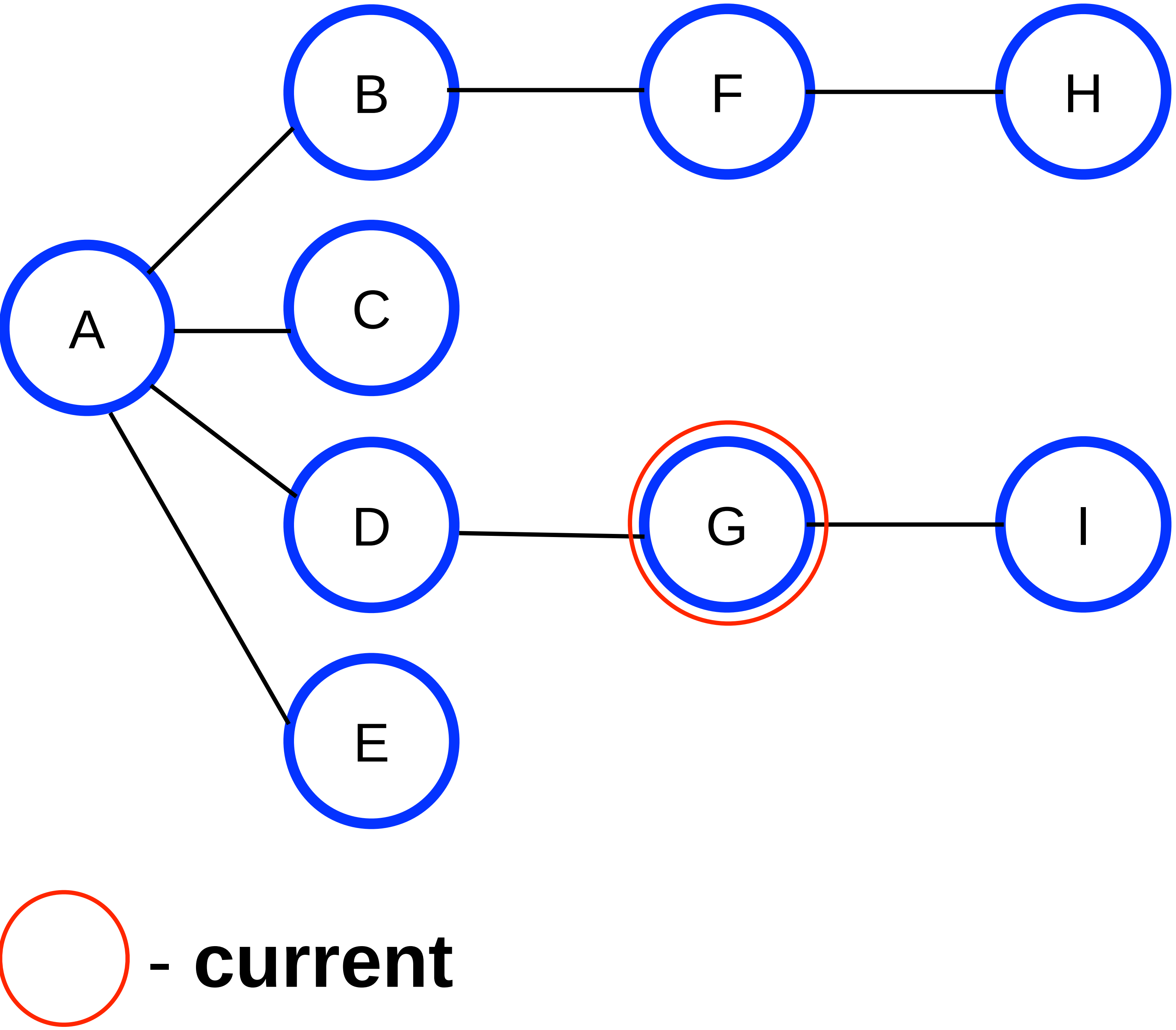
- current



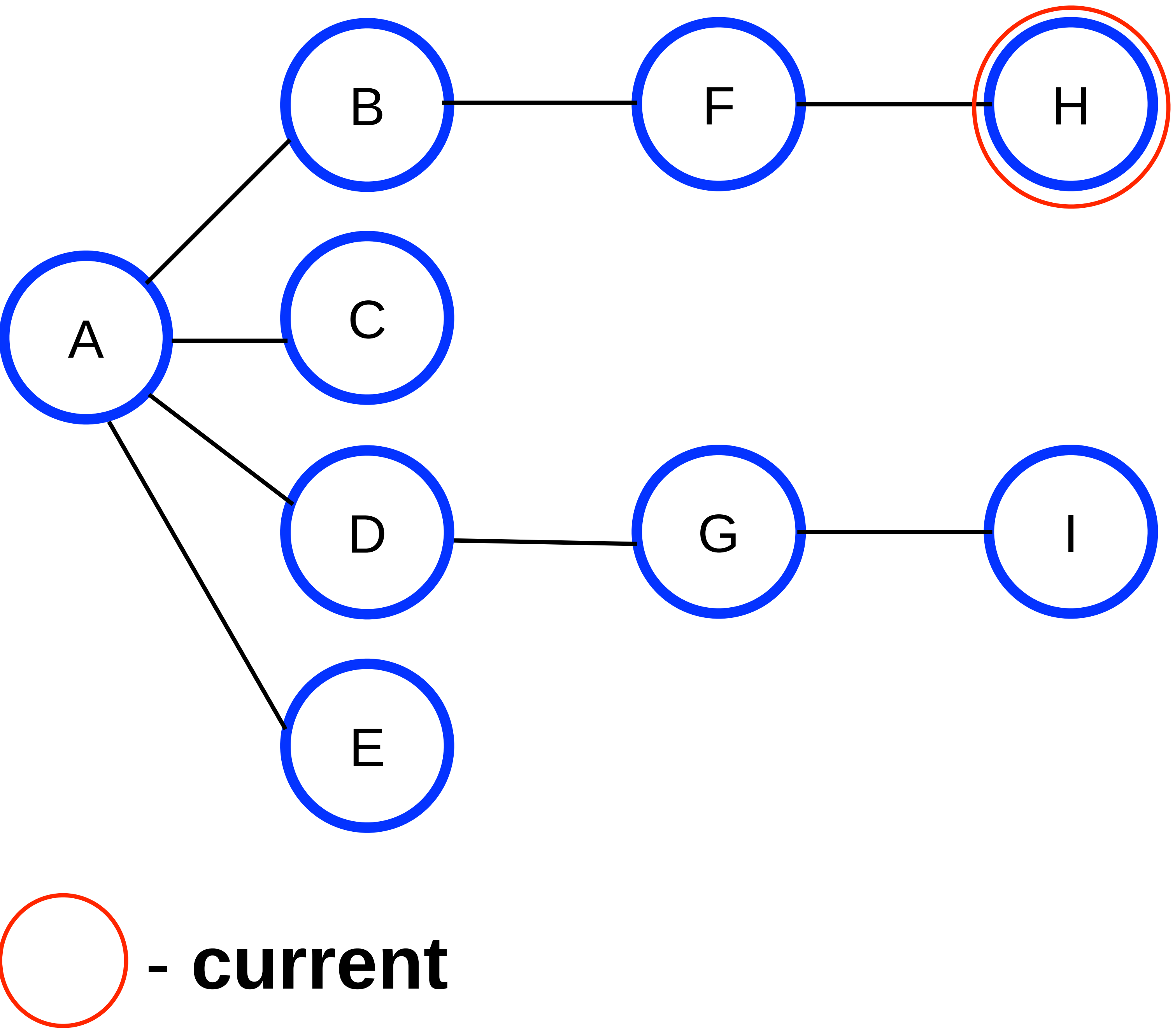
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G
Visit H	GH



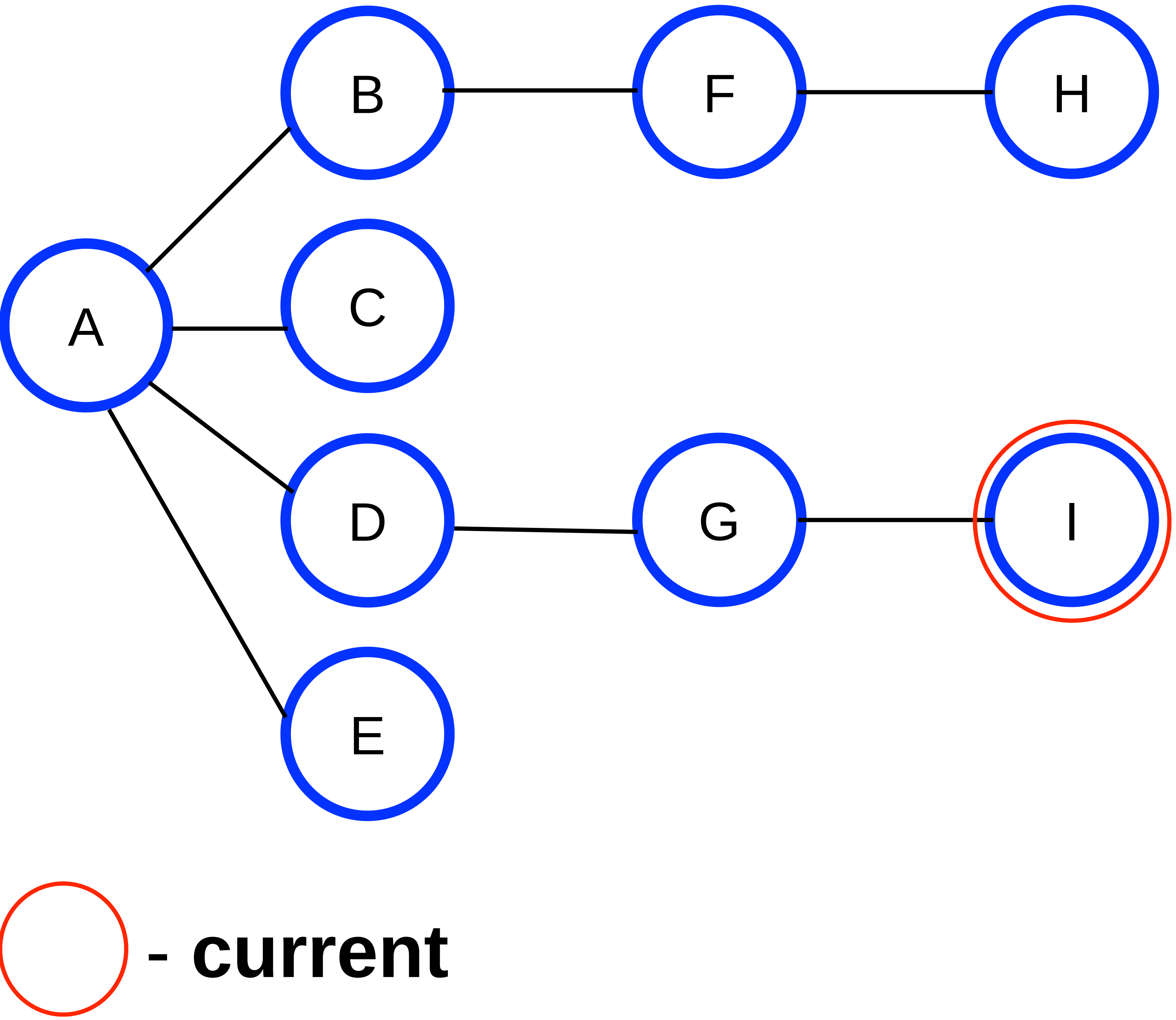
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G
Visit H	GH
Dequeue (G)	H



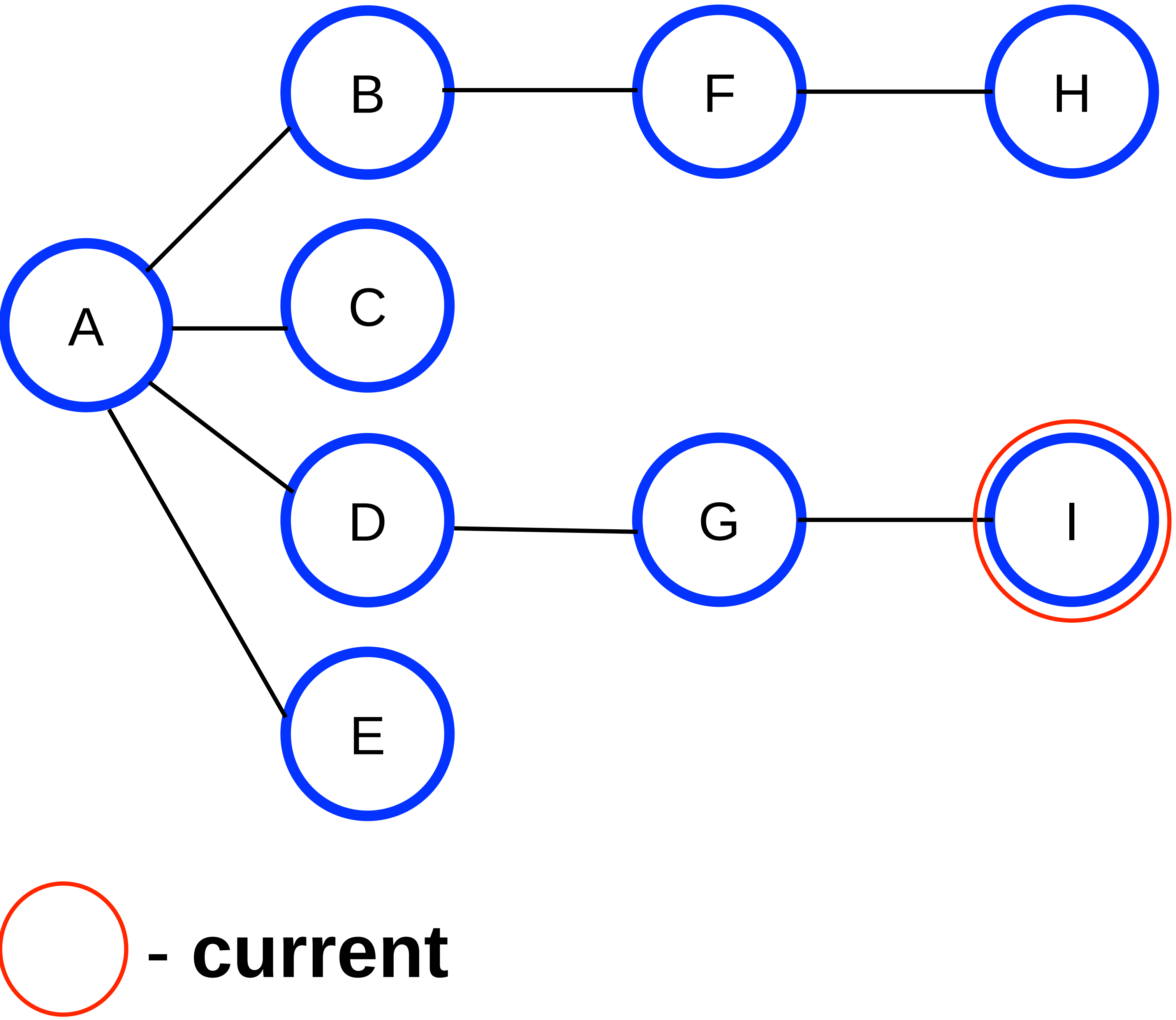
Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G
Visit H	GH
Dequeue (G)	H
Visit I	HI



Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G
Visit H	GH
Dequeue (G)	H
Visit I	HI
Dequeue (H)	I

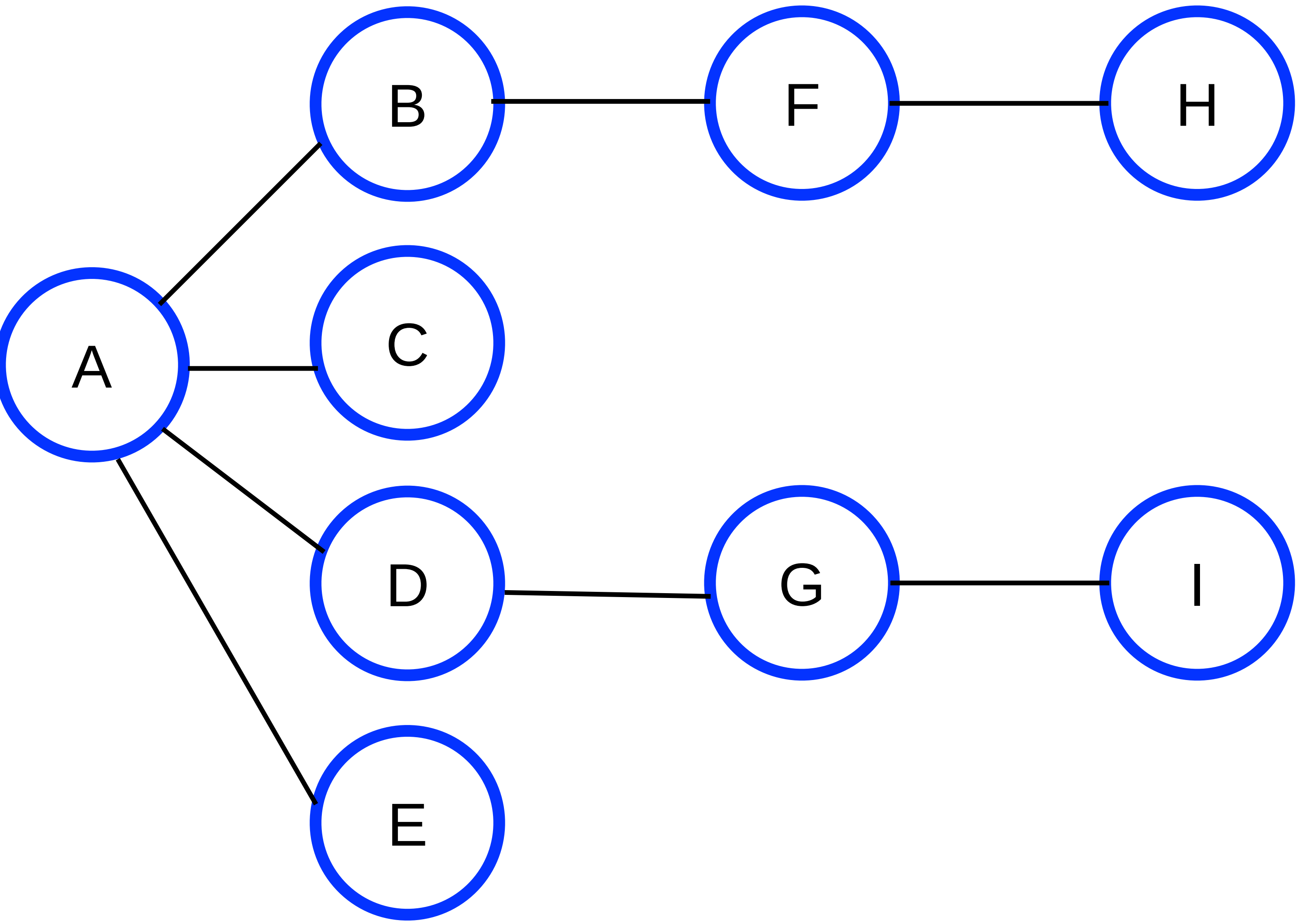


Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G
Visit H	GH
Dequeue (G)	H
Visit I	HI
Dequeue (H)	I
Dequeue (I)	

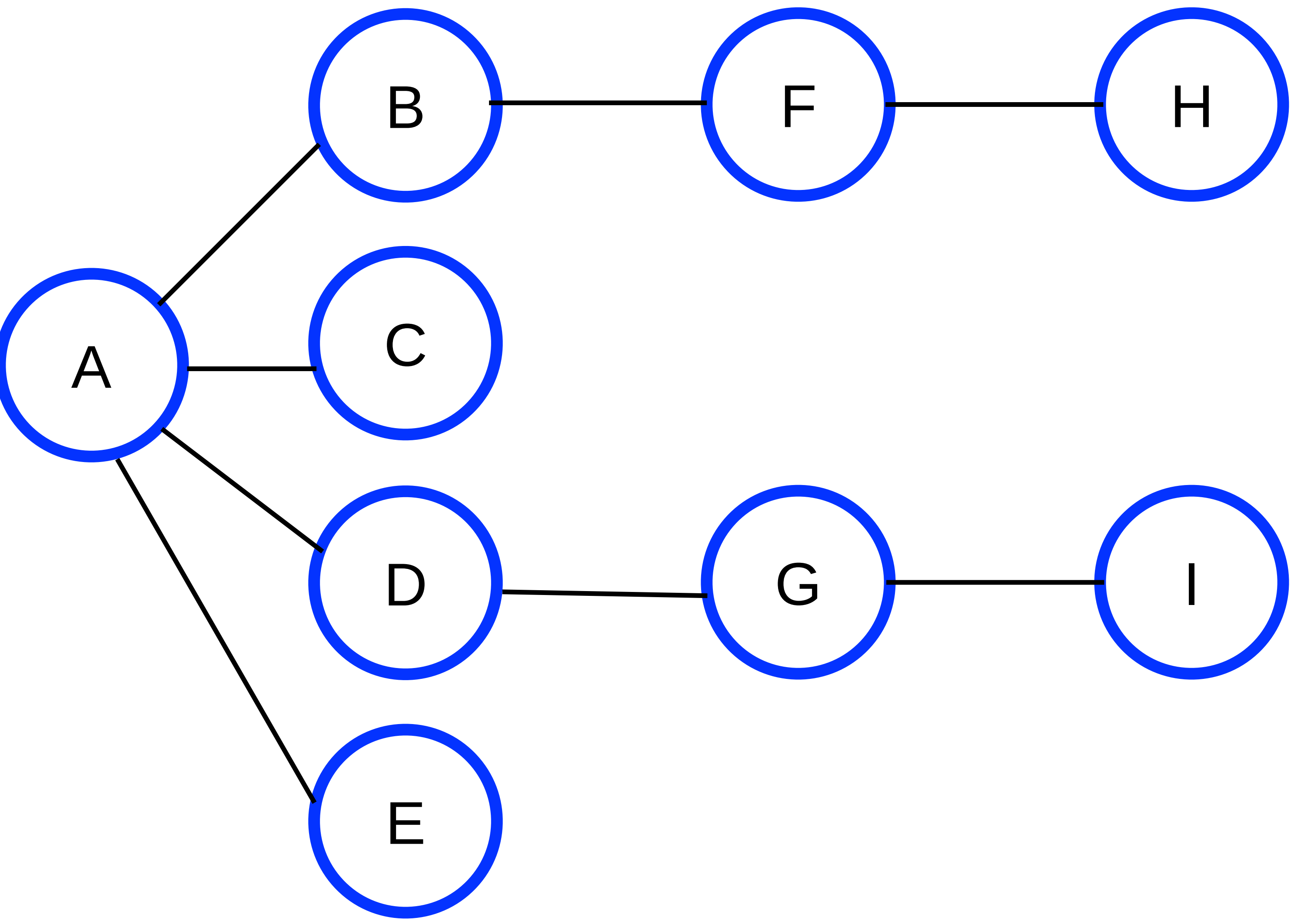


Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G
Visit H	GH
Dequeue (G)	H
Visit I	HI
Dequeue (H)	I
Dequeue (I)	

Now the queue is empty, so it is time for **Rule 3:**
"If you can't carry out Rule 2 because the queue is empty, you are finished"



Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G
Visit H	GH
Dequeue (G)	H
Visit I	HI
Dequeue (H)	I
Dequeue (I)	
Done	



Order: ABCDEFGHI

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

Event	**Queue**
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Dequeue (B)	CDE
Visit F	CDEF
Dequeue (C)	DEF
Dequeue (D)	EF
Visit G	EFG
Dequeue (E)	FG
Dequeue (F)	G
Visit H	GH
Dequeue (G)	H
Visit I	HI
Dequeue (H)	I
Dequeue (I)	
Done	

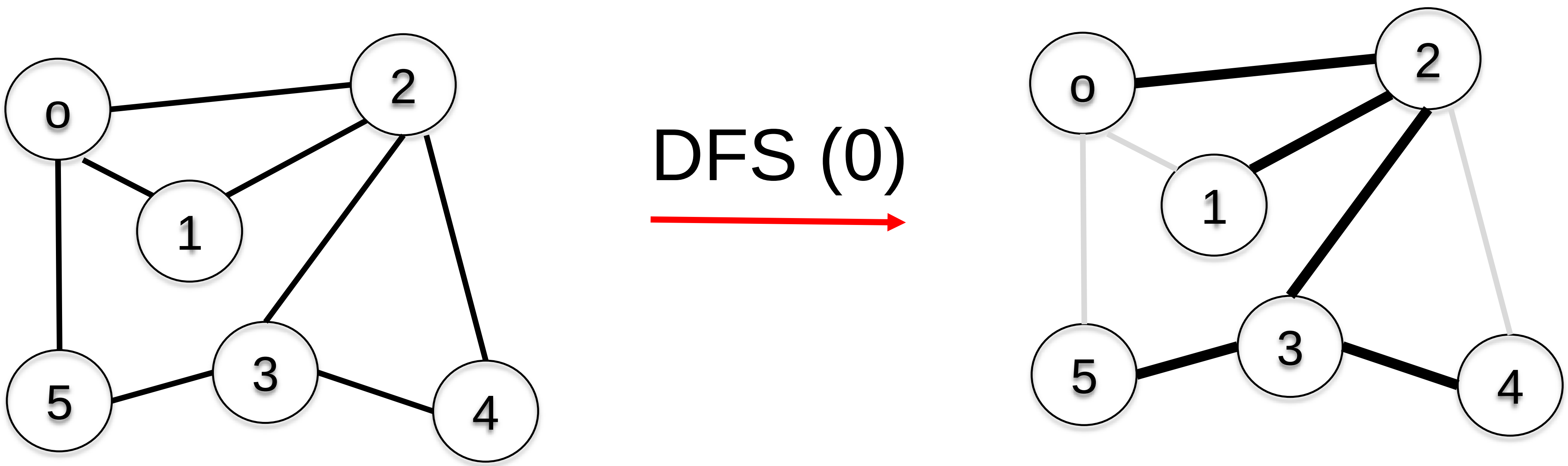
Breadth First Search

- Notice that,
  - BFS tries to stay as close as possible to the starting point
  - Thus the name, **Breadth First Search**
- Implementation of BFS is left as an exercise

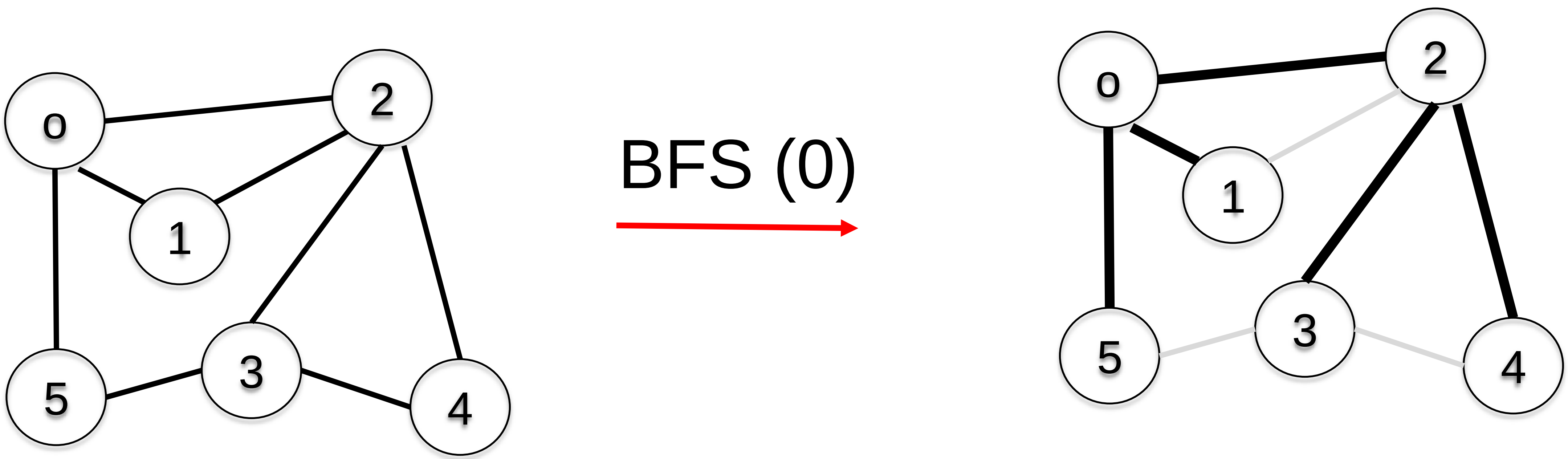
DFS & BFS

- Can be used to find:
  - whether there is a path between two vertices
  - whether a graph is connected
  - whether there is a cycle
  - connected components of a graph (slight modification or extension)

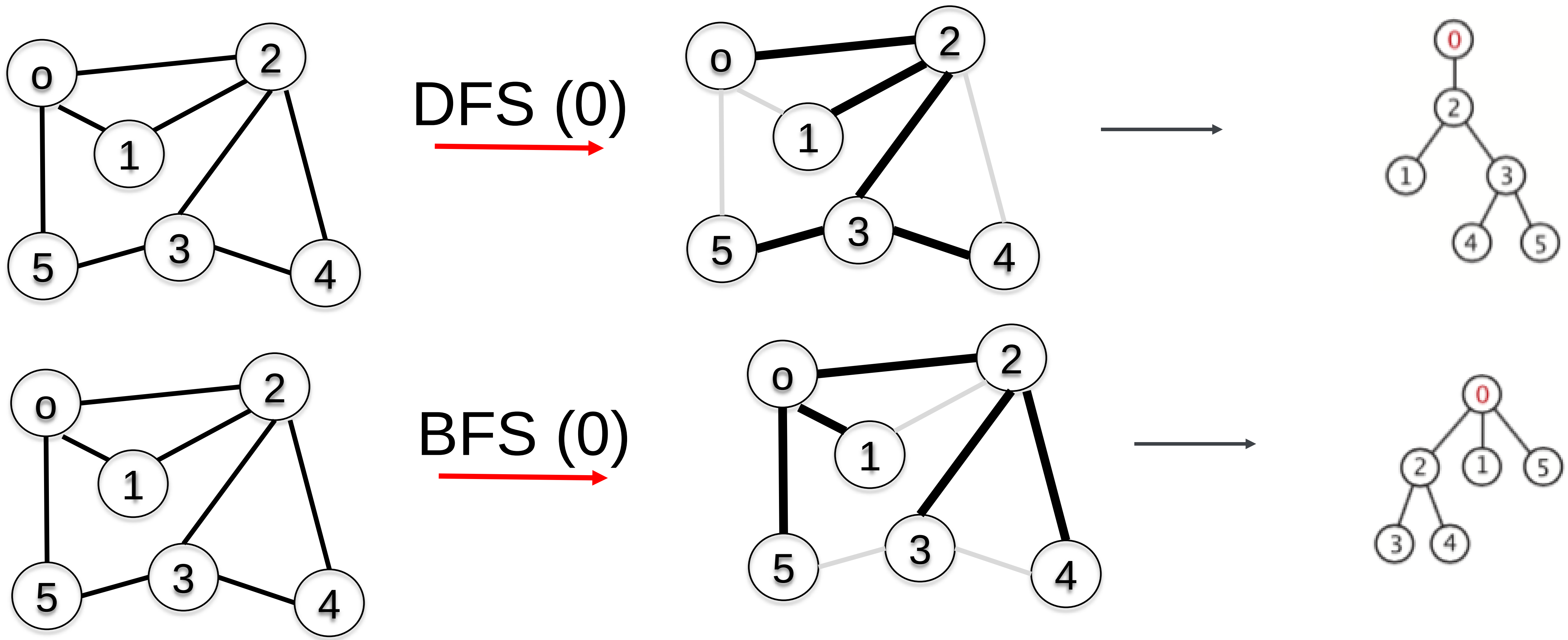
Final Remarks (1)



Final Remarks (2)



Final Remarks (3)



DFS finds a path, whereas BFS finds the shortest path

However, note that the graph is: unweighted (or same weight)