#### **Graph ADT**

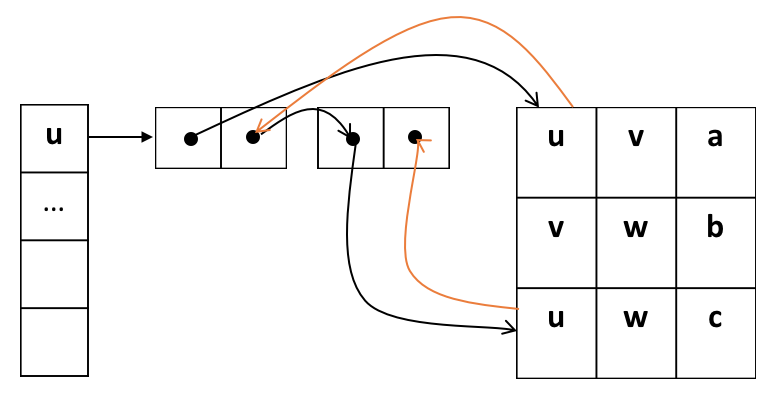
* **Data**: all vertices, all edges, and the structure to maintain relations between vertices and edges.
* **Functions**:
  + insert vertex/edge,
  + remove vertex/edge,
  + find incident edges, ￼
  + check if two vertices are adjacent, and
  + In case of directed graph find origin/destination.

##### **Question: Implementation 2 runs in either O(1) or O(n), while Implementation 1 runs in either O(1) or O(m). Which one is better?**

* + Depends on the data. If the graph is not connected Implementation 1 is better even though it seems bad when we look at the worst case running time.
  + Before we decide on the implementation we need to consider what our data set looks like. → O(m) is obviously better than O(n) because there are less edges than the number of vertices. However, depending on the manner in which we are accessing our graph, we may want one over the other (if there are significantly more O(1) accesses that would be O(m) in the other)

**Graph implementation 3 : ADJ List**

* + We will maintain a hash table of vertices, and every vertex in the table has a linked list of pointers which point to edges in the edge list.
  + Elements from the edge list will point back to the hash table.



* + The running time of **insert vertex** would be O(1) → we would just add a vertex to the hash table.
  + The running time of **remove vertex** would be O(deg(v)):
    - loop through the list of incident edges → each node v has deg(v) incident edges, so we have to go over deg(v) edges to remove a vertex.
  + To **check adjacent nodes**, we need to go through incident edges of one of the vertices. We will choose the vertex with smaller list. The running time is O(min(deg(v1), deg(v2))).
  + **Find incident edge**s takes O(deg(v)).
  + **InsertEdges**: O(1) + O(1) + O(1) = O(1)

#### 

#### **Summary of graph implementation run time**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Edge List** | **Adjacency Matrix** | **Adjacency List** |
| **space** | n+m |  | n+m |
| **insert vertex** | 1 😀 | n | 1 😀 |
| **remove vertex** | m | n | deg(v) 😀 |
| **insert edge** | 1 😀 | 1 😀 | 1 😀 |
| **remove edge** | 1 😀 | 1 😀 | 1 😀 |
| **incident edges** | m | n | deg(v) 😀 |
| **are adjacent** | m | 1 😀 | min(deg(v), deg(w)) |

* When removing a vertex deg(v) is always the best: if m = 0, deg(v) = 0; if the graph is simple, deg(v) = n-1.
* If we care about areAdjacent we are going to use adjacency matrix. On the other hand, if we care about incident vertices, we will use adjacency list. Insert/remove an edge takes O(1) because we are just adding/removing from the front of the linked list.
* Some possible cases:
* Sparse graphs: the graph is not connected → which implies . So in the case of sparse graph, we want to use adjacency list implementation.
* Dense graphs: the graph is almost fully connected → . So in the case of dense graph, we can use either adjacency list or adjacency matrix. It depends on the operations we need (are adjacent or insert vertex).

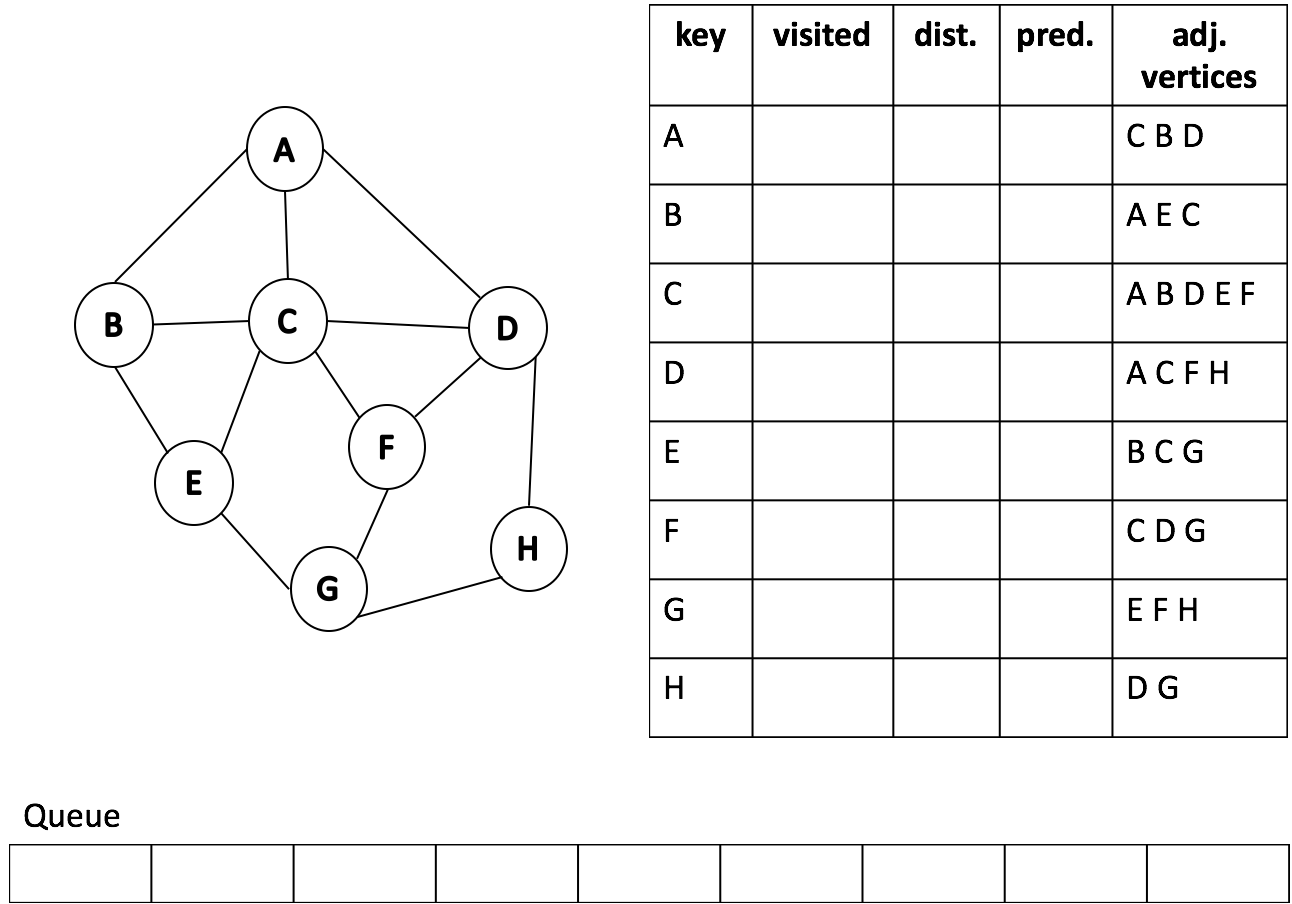
#### **Traversal**

* Objective: Visit every vertex and every edge exactly once
* Purpose: Search for interesting substructures in the graph
* We’ve done it on trees, but it was easier

|  |  |
| --- | --- |
| **Trees** | **Graphs** |
| 1. Ordered → we always go from parents to children. 2. Obvious start → we start at the root node. 3. Notion of completeness → we are done when we reach leaf nodes. | 1. Unordered → no notion of children nodes, just neighbours. 2. No obvious start → we can start anywhere. 3. No notion of completeness → we need to know when we have visited all nodes. |

#### **BFS**

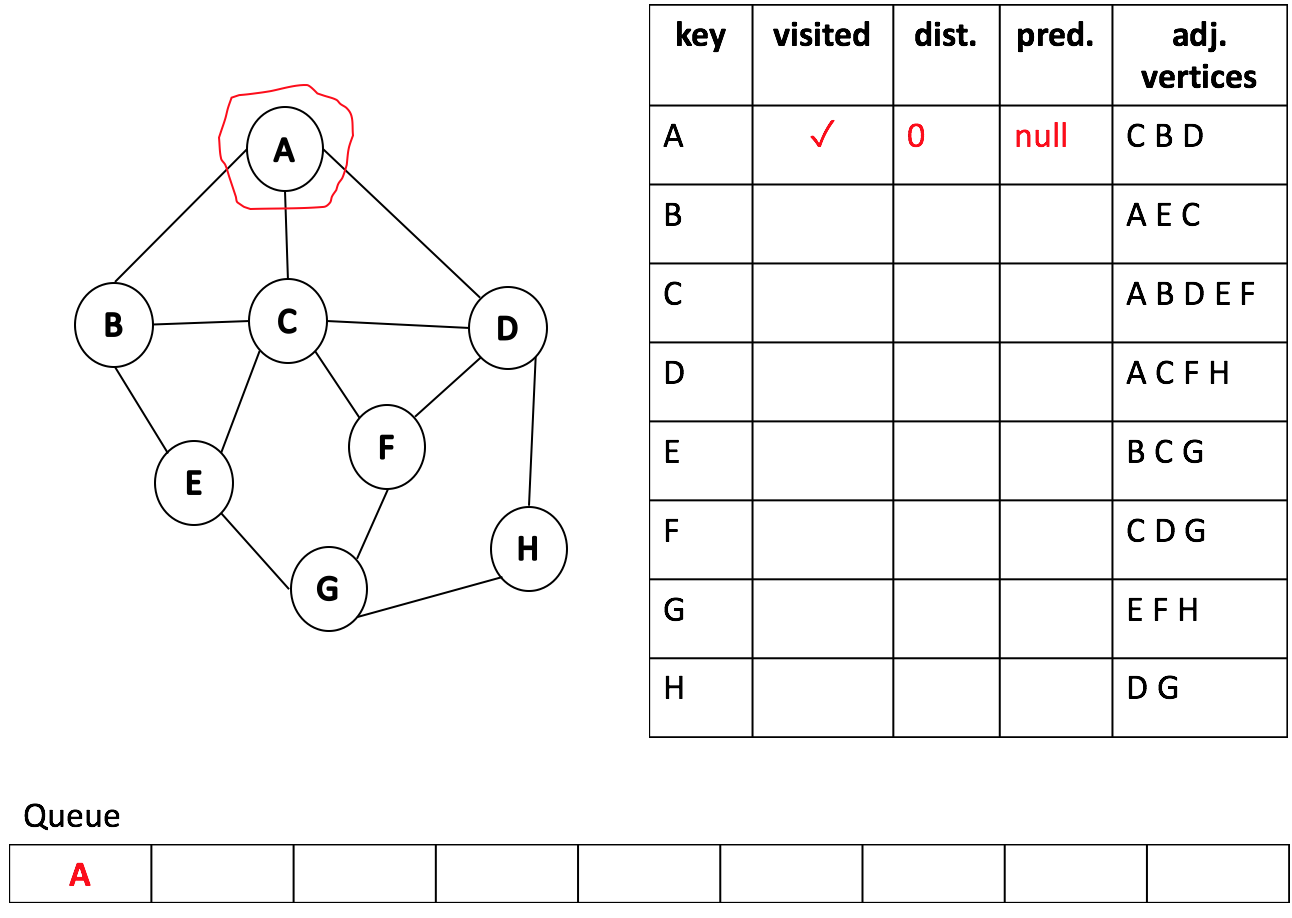
* Setup:
  + Maintain a queue
  + Maintain a table of vertices with following features:
    - Boolean flag - visited
    - Distance from the start
    - Predecessor
    - List of adjacent vertices



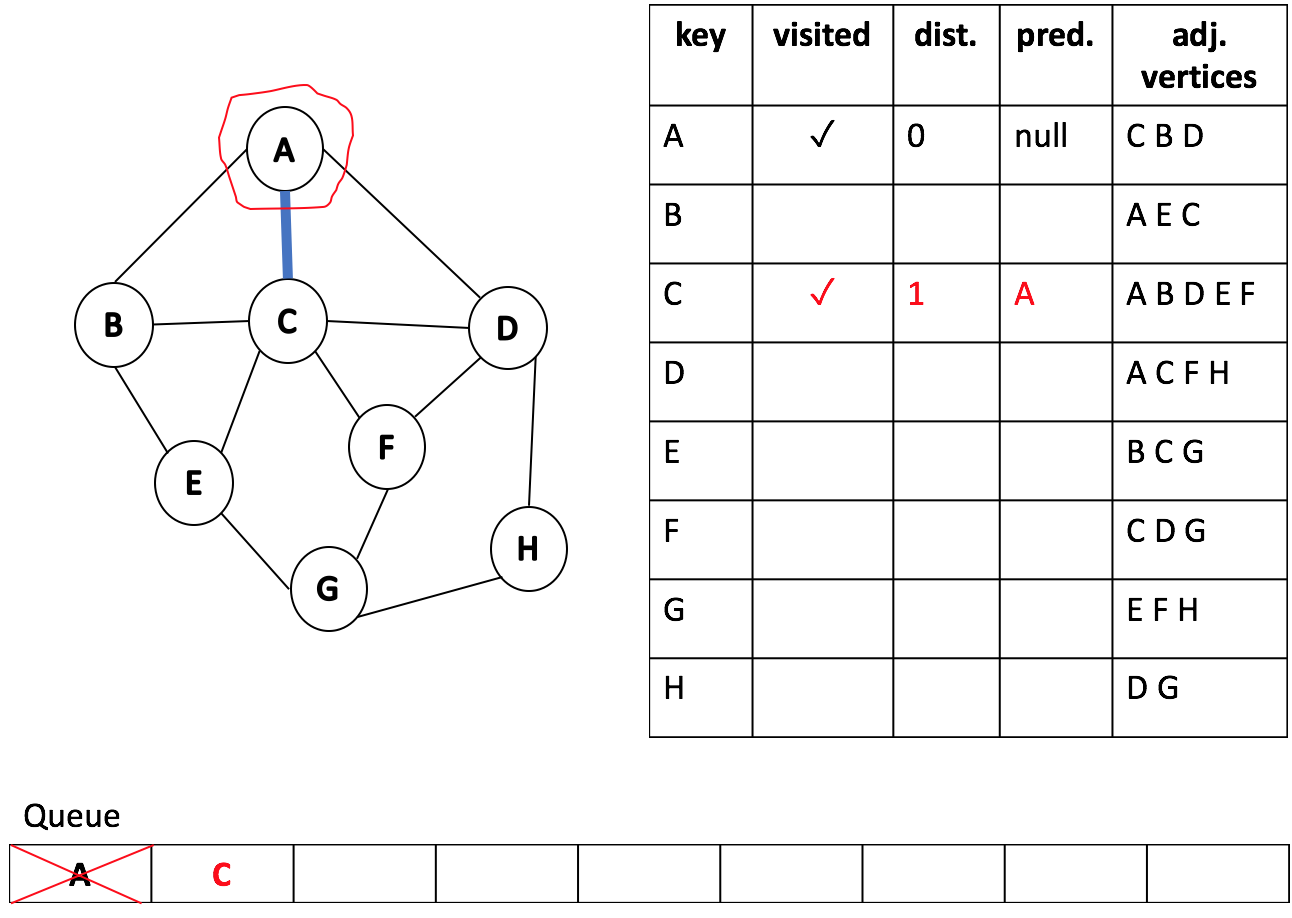
* Algorithm:

1. Add the starting point
2. While the queue is not empty
   1. Dequeue **v**
   2. For all of the unlabelededges adjacent to **v**
      * If an adjacent edge “discovers” a new vertex **t**:
        + Label the edge a “discovery edge”
        + Enqueue **t**, update the information of **t** (distance = dist(**v**) + 1, predecessor = **v**)
      * If an adjacent edge is between two visited vertices
        + Label the edge a “cross edge”

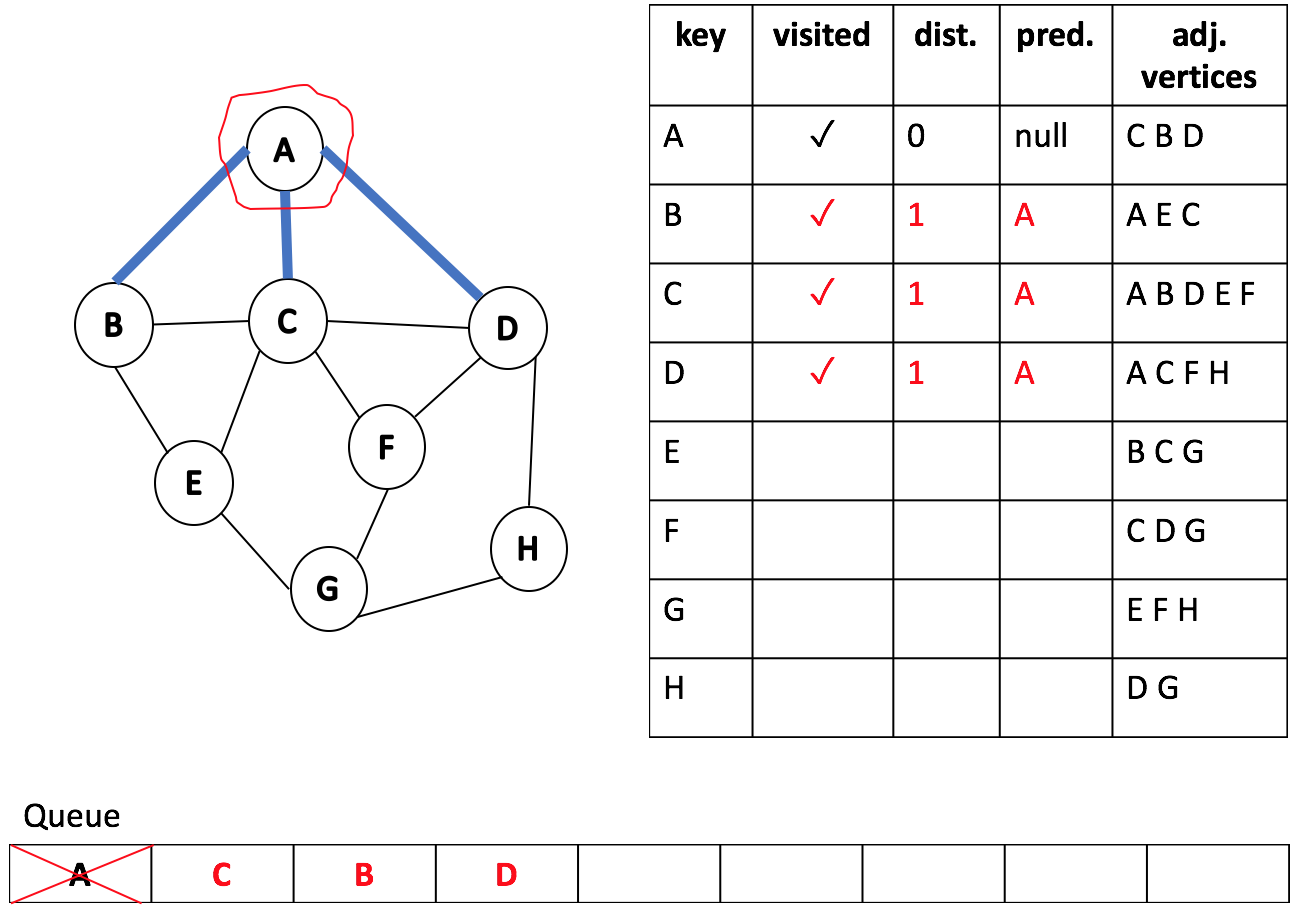
* **Example:**
  + A as a starting point:



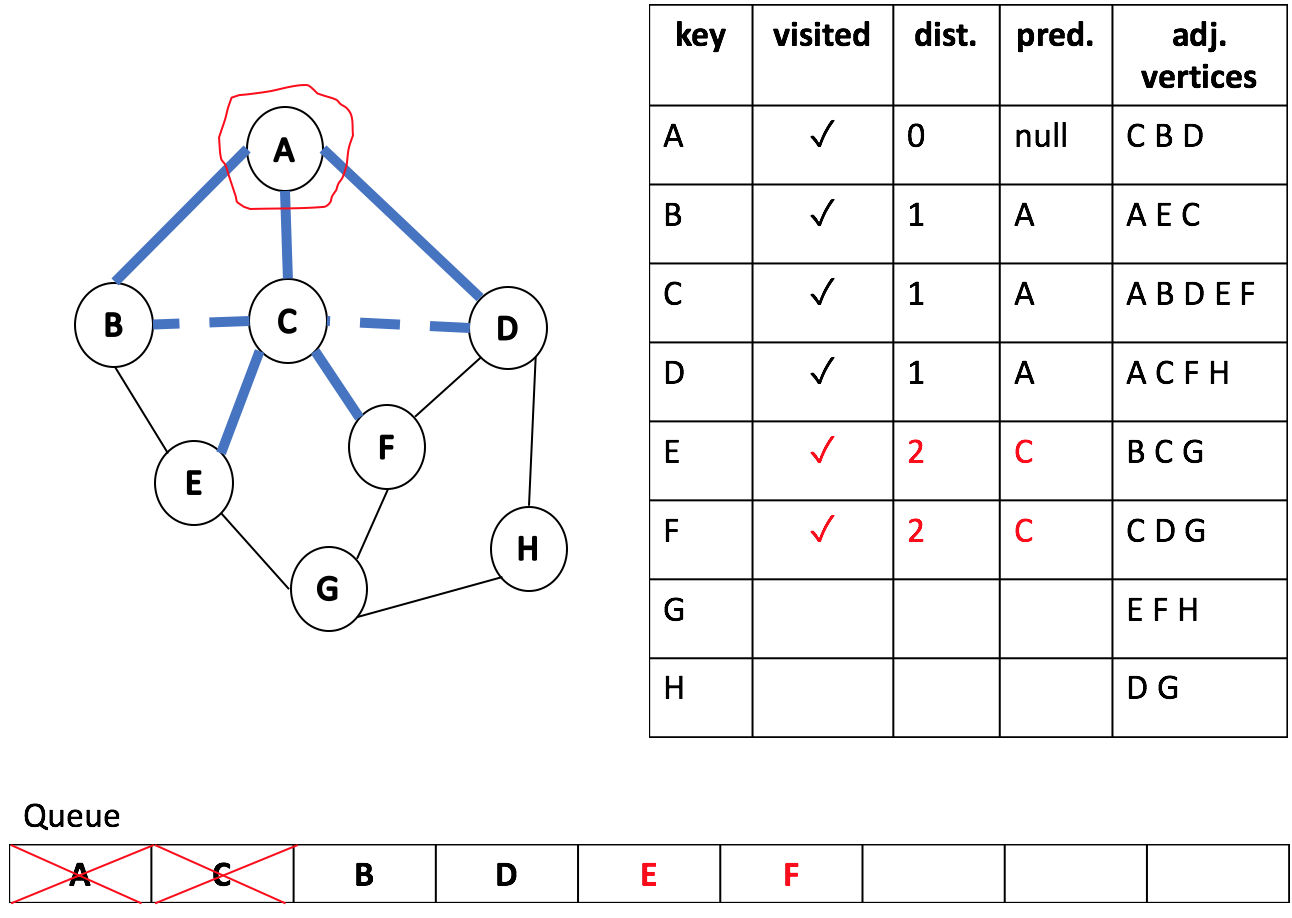
* + dequeue A and examine vertices C, B, and D.
  + First examine C. It hasn’t been visited, so we add a discovery edge, update visited flag, set distance to parent’s distance plus vertex distance which is one in this case, set predecessor to A, and add C to the queue.



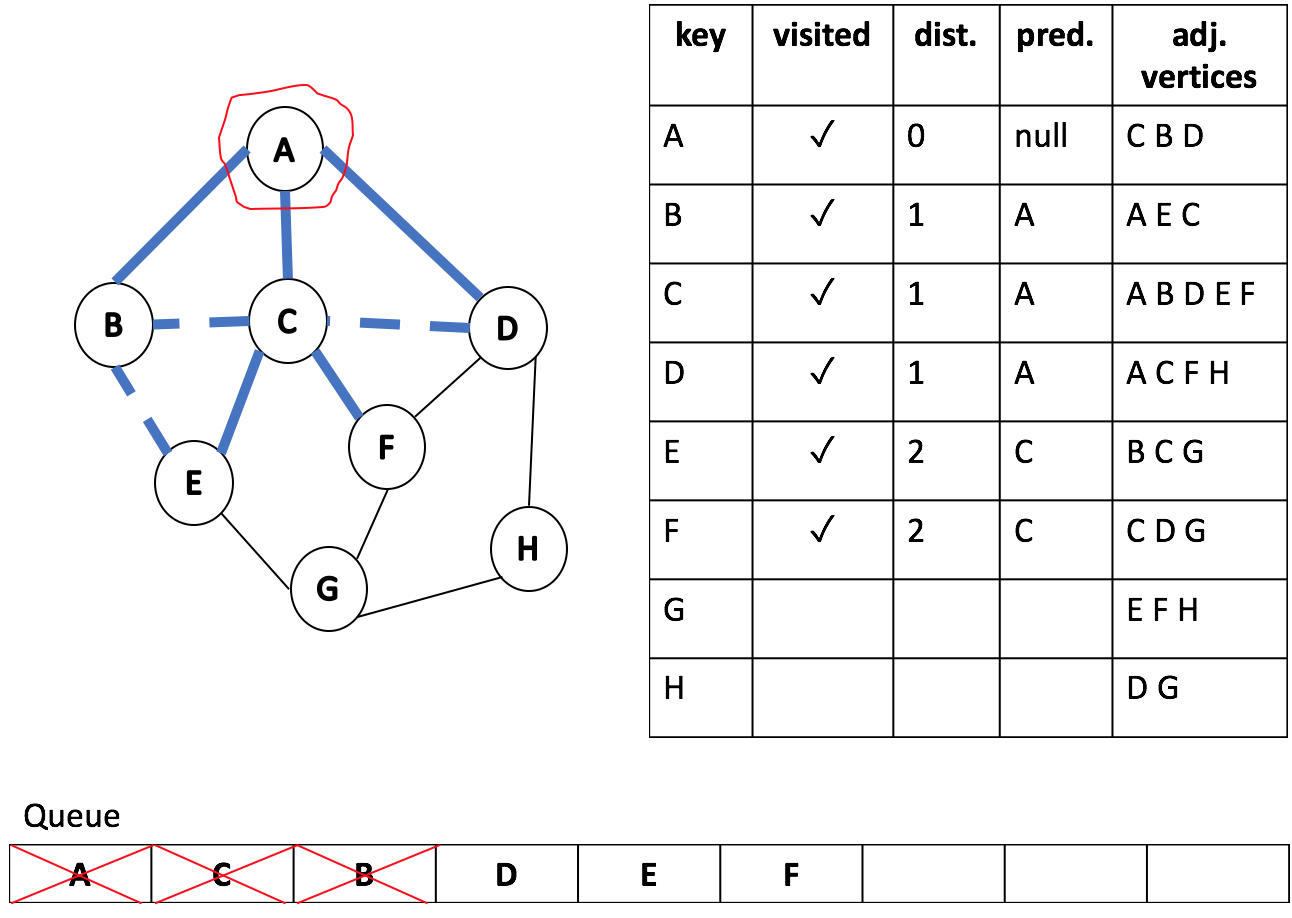
* + We do the same for B and D.



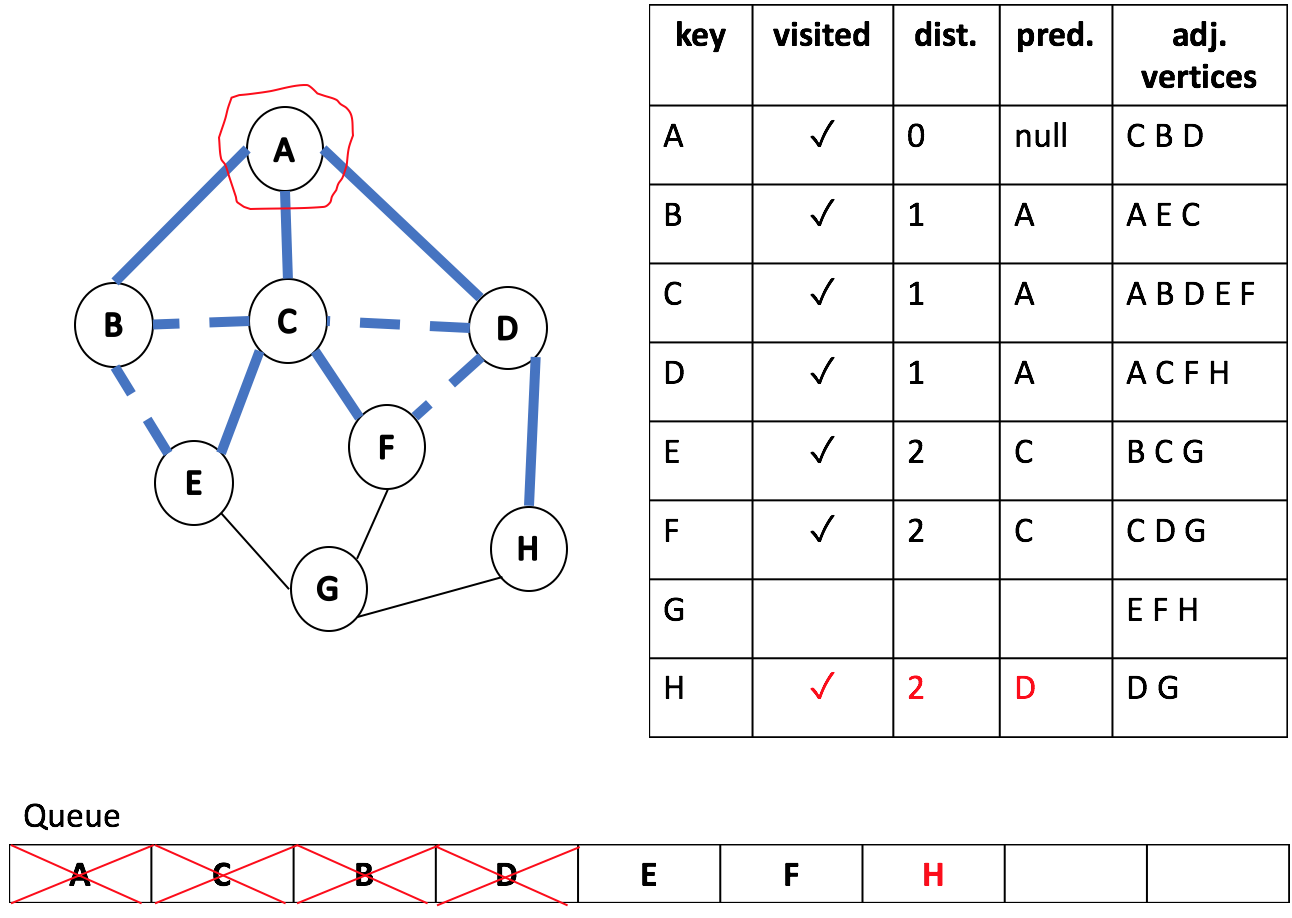
* + Repeat this process until the queue is empty.
  + Dequeue C and examine its adjacent edges. A has been visited and the edge is labeled as discovery edge, so it is just ignored. B and D have been visited from another node, so we add a cross edge but do not update anything. E and F have not been discovered yet, therefore we add discovery edges to the graph, add vertices to the queue, and update the table.



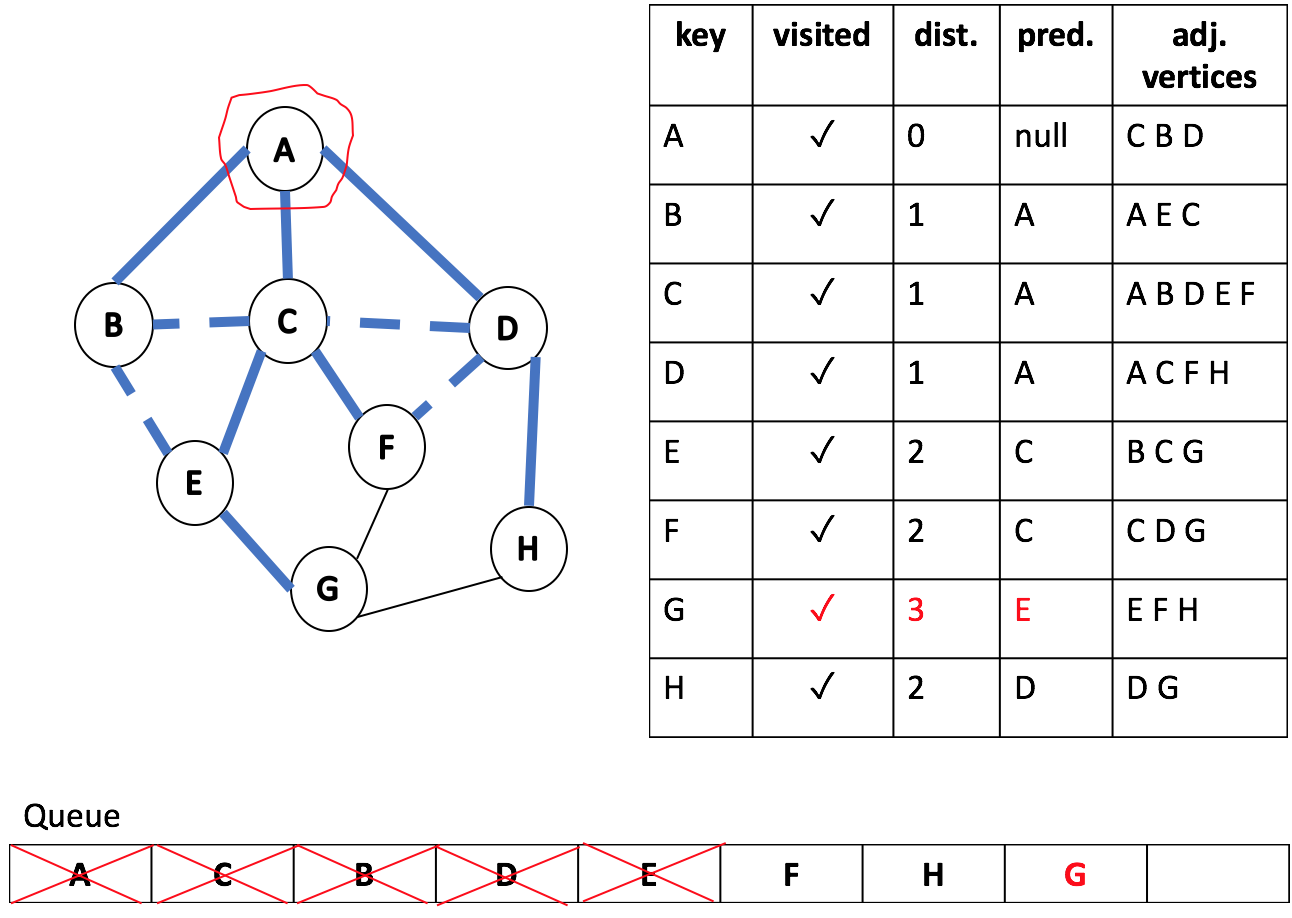
* + Next, dequeue B and add a cross edge to E.



* + Dequeue D, add cross edge to F, and update H.



* + Then, dequeue E and update G.



* + Dequeue F and add cross edge to G; Dequeue H and add cross edge to G.
  + Finally dequeue G and we are done.

