COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 16: April 10

Lecturer: Dr. Andrew Hines *Scribes: Xiang Zhao, Jiarui Qin*

1 What is a Graph?

Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as:

Theorem 1. A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes. Like a Tree ADT, a graph is a non-linear data type, but cycles and non-connected components are allowed.

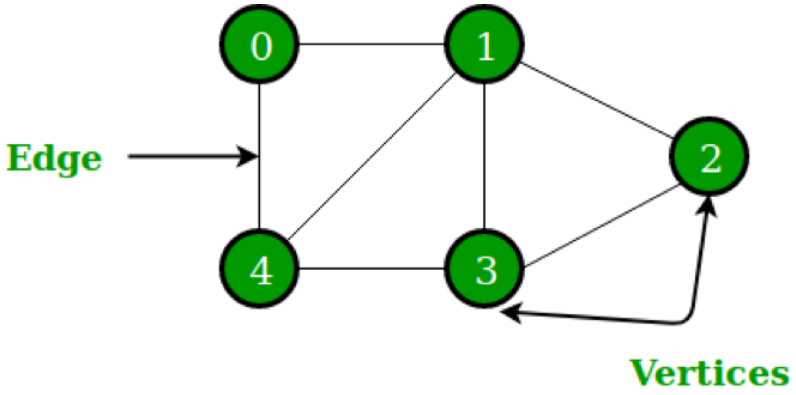


Figure 1: Vertices and Edges

Cite From Geeksforgeeks

In the above Graph, the set of vertices V = 0,1,2,3,4 and the set of edges E = 01, 12, 23, 34, 04, 14, 13.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, on Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale, etc.

2 Graph and its representations

1. The graph is a data structure that consists of the following two components:
   (a) A finite set of vertices also called nodes
   (b) A finite set of ordered pair of the form (u, v) called as an edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.
2. Graphs are used to represent many real-life:
   (a) Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.
   (b) Graphs are also used in social networks like LinkedIn, Facebook. For example, on Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, and locale.
3. The Graph ADT represents the mathematical concepts of:
   (a) Directed graph: a directed graph (or digraph) is a graph that is made up of a set of vertices connected by edges, where the edges have a direction associated with them.

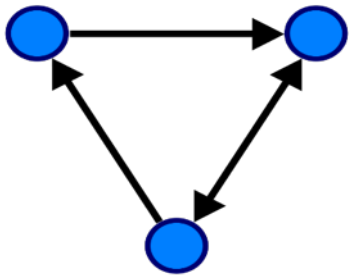


Figure 2: Directed Graph

Cite From Geeksforgeeks

   (b) Undirected graph: an undirected graph is made up of unordered pairs of vertices, which are usually called edges, arcs, or lines.

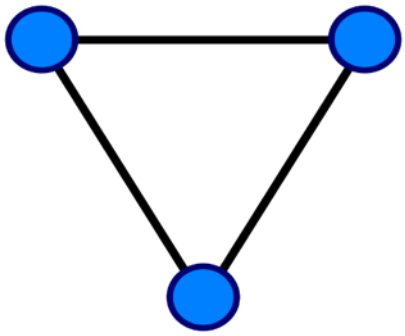


Figure 3: Undirected Graph

Cite From Geeksforgeeks

4. Operations of the Graph ADT:

   (a) Adjacent (G, x, y): tests whether the given there is an edge from the vertices x to y in the given graph G;

Algorithm 1: Psuedo-code for testing adjacent node

```
Input  : G, a, b
Output: True if a is adjacent to b otherwise False
if G[a][b] = 0 then
    return False;
else
    return True;
end
```

`adjacent(G, b, c)` = True
`adjacent(G, c, d)` = False

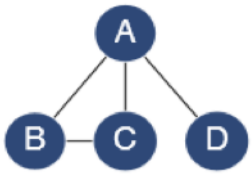


Figure 4: Adjacent Testing

Cite From Geeksforgeeks

(b) Neighbors (G, x): lists all vertices y such that there is an edge from the vertices x to y in the given graph G;

Algorithm 2: Psuedo-code for finding all neighbour nodes

```
Input  : G, a
Output: All neighbour nodes of a
neighbours ← []
for i ← nodes in the graph do
    if G[a][i] = 1 then
        Add i into neighbours
    else
end
return neighbours
```

neighbours(G, a) = $\{b, c, d\}$
neighbours(G, b) = $\{a, c\}$
neighbours(G, d) = $\{a\}$

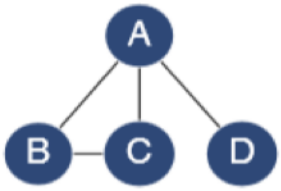


Figure 5: Neighbors

Cite From Geeksforgeeks

(c) Add vertex (G, x): adds the vertex x, if it is not there

Algorithm 3: Add new vertex into a graph

Input : G, *vertex*
Output: A new graph
for $v \leftarrow$ *all vertices in the graph* **do**
  $G[v][vertex] \leftarrow 1$ $G[vertex][v] \leftarrow 1$
end
return G

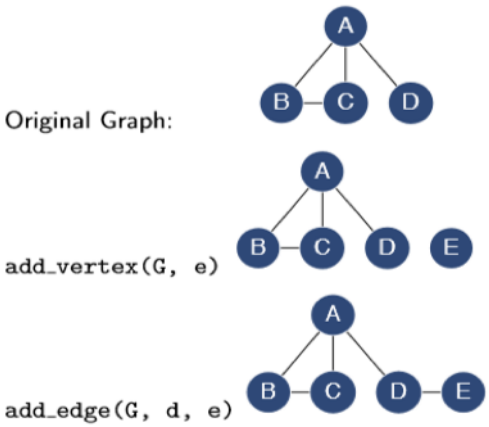


Figure 6: Add Vertex

Cite From Geeksforgeeks

(d) Remove vertex (G, x): removes the vertex x, if it is there;

Algorithm 4: Remove a vertex from a graph

Input : G, *vertex*
Output: A new graph
for $v \leftarrow$ *all vertices in a graph* **do**
  $G[v][vertex] \leftarrow 0$ $G[vertex][v] \leftarrow 0$
end
return G

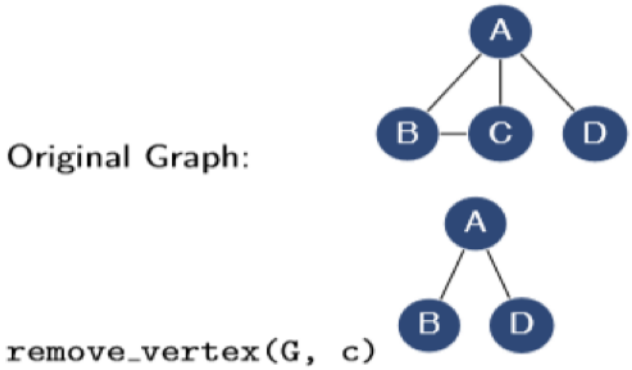


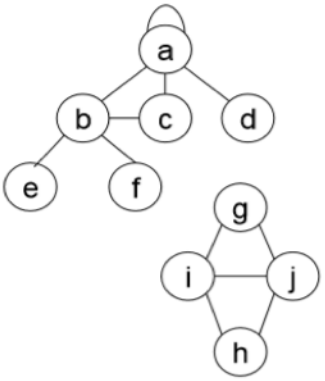
Figure 7: Remove Vertex

Cite From Geeksforgeeks

5. Computational Representation of Graphs

   Definition 2.1. Computational Representation of Graphs How to convert the visualization into a computer data structure. G (V, E): Graphs are a function of their Vertices and Edges

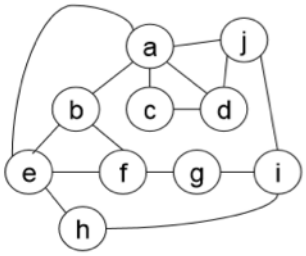
   Definition 2.2. Adjacency List An adjacency-list provide a compact representation for sparse graphs, i.e. those for which —E— is much less than $|V|^2$



a	a, b, c, d
b	a, c, e, f
c	a, b
d	a
e	b
f	b
g	i, j
h	i, j
i	g, h, j
j	g, h, i

Figure 8: Adjacent List
e Cite From Geeksforgeeks

As we can see from Adjacency-list Graph 1, the adjacency of a is a, b, c, d, so we put this element in one list called a. The rest examples are the same case.



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Figure 9: Adjacent List

Cite From Geeksforgeeks

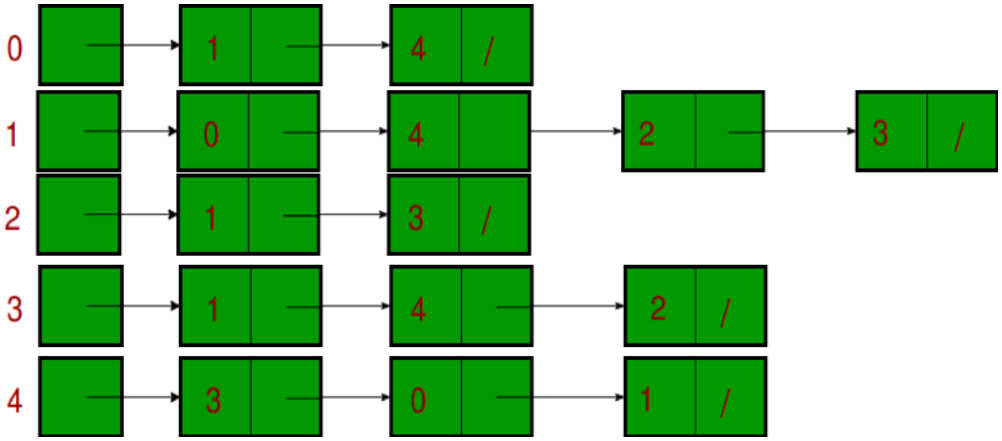


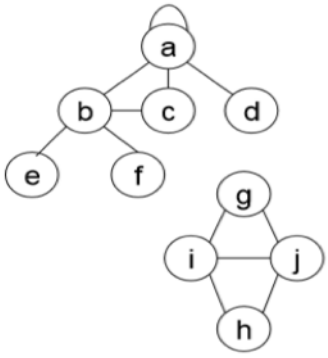
Figure 10: Adjacent List

Cite From Geeksforgeeks

As we can see from Adjacency-list Graph 3, an array of lists is used. Size of the array is equal to the number of vertices. Let the array be array []. An entry array [i] represents the list of vertices adjacent to the ith vertex.

This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is adjacency list representation of the above graph.

Definition 2.3. Adjacency Matrix An adjacency-matrix is better when a graph is dense, i.e. $|E|$ is close to $|V|^2$

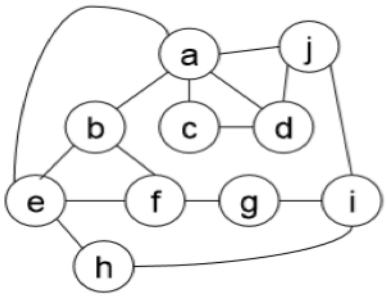


	a	b	c	d	e	f	g	h	i	j
a	1	1	1	1						
b	1		1		1	1				
c	1	1								
d	1									
e		1								
f		1								
g									1	1
h									1	1
i							1	1		1
j							1	1	1	

Figure 11: Adjacent Matrix

Cite From Geeksforgeeks

As can be seen from Adjacency-matrix Graph 1, this is a Boolean here. For instance, if there is adjacency between a and b in the graph, we put 1 here, if not we put nothing here.



	a	b	c	d	e	f	g	h	i	j
a		1	1	1	1					1
b	1				1	1				
c	1			1						
d	1		1							1
e	1	1				1		1		
f		1			1		1			
g						1			1	
h					1				1	
i							1	1		1
j	1			1					1	

Figure 12: Adjacent Matrix

Cite From Geeksforgeeks

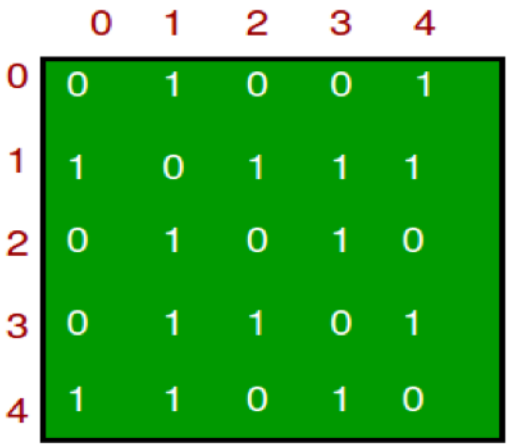


Figure 13: Adjacent Matrix

Cite From Geeksforgeeks

As can be seen from the Adjacency-matrix Graph 3, Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j. Adjacency matrix for the undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w.

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex u to vertex v are efficient and can be done $O(1)$.

Cons: Consumes more space $O(V^2)$. Even if the graph is sparse (contains a smaller number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

6. DFS and BFS

   Depth-first search (DFS) is a method for exploring a tree or graph. In a DFS, we can determine whether two nodes x and y have a path between them by looking at the children of the starting node and recursively determining if a path exist.

While Bread-first search (BFS) will traverse through a graph one level of children at a time, Depth-first search will traverse down a single path, one child node at a time.

The depth-first algorithm sticks with one path, following that path down a graph structure until it ends. The breadth-first search approach, however, evaluates all the possible paths from a given node equally, checking all potential vertices from one node together, and comparing them simultaneously.

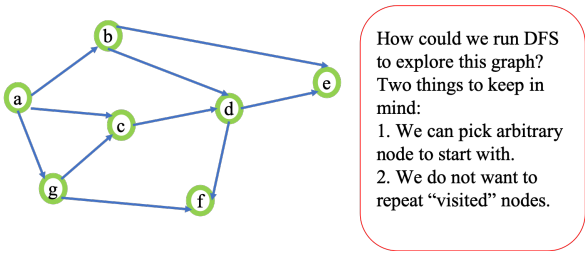


Figure 14: DSF

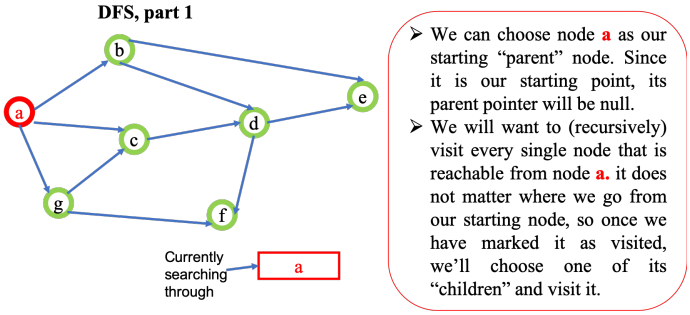


Figure 15: DFS 1

We can choose node a as our starting *parent* node. Since it is our starting point, its parent pointer will be *null*.

We will want to recursively visit every single node that is reachable from node *a*. it does not matter where we go from our starting node, so once we have marked it as visited, we'll choose one of its *children* and visit it.

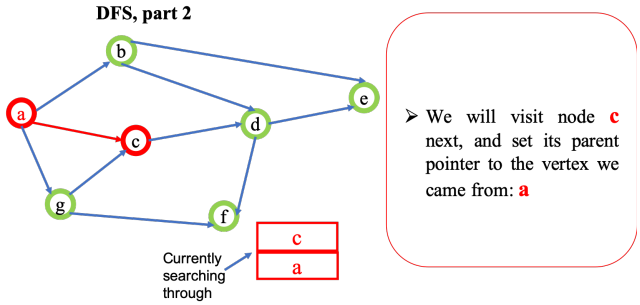


Figure 16: DFS 2

We will visit node next c, and set its parent pointer to the vertex we came from:

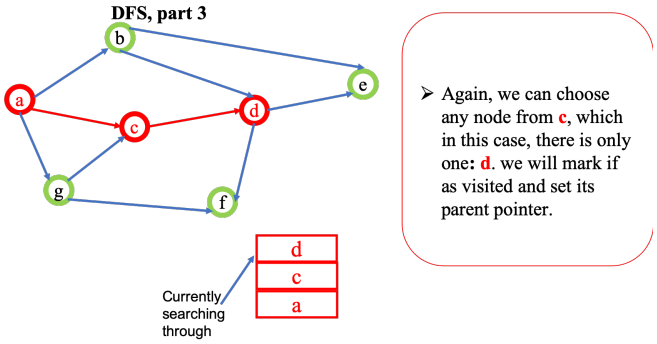


Figure 17: DFS 3

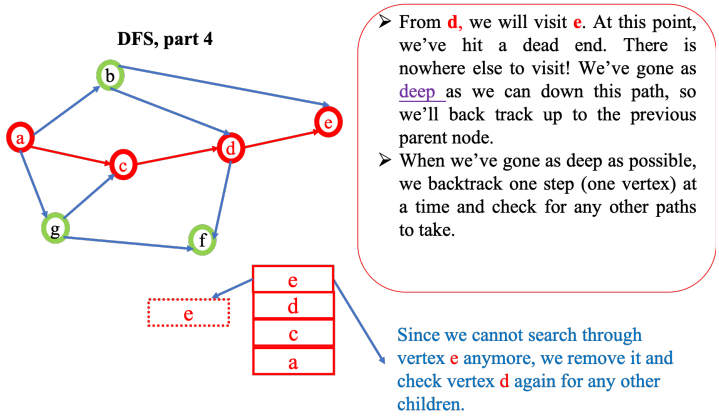


Figure 18: DFS 4

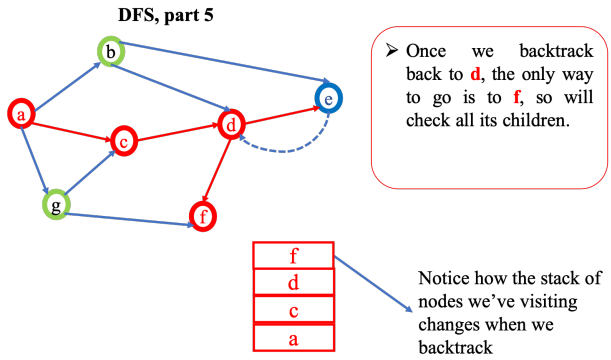


Figure 19: DFS 5

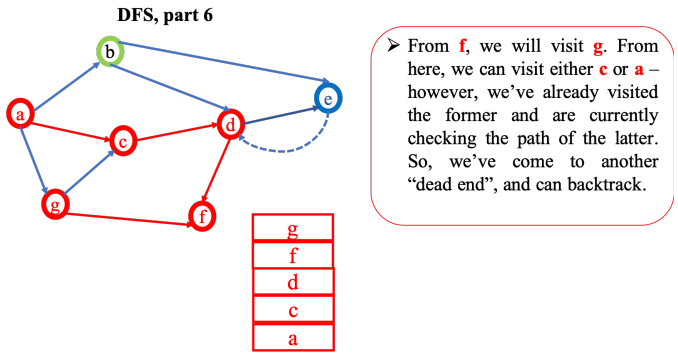


Figure 20: DFS 6

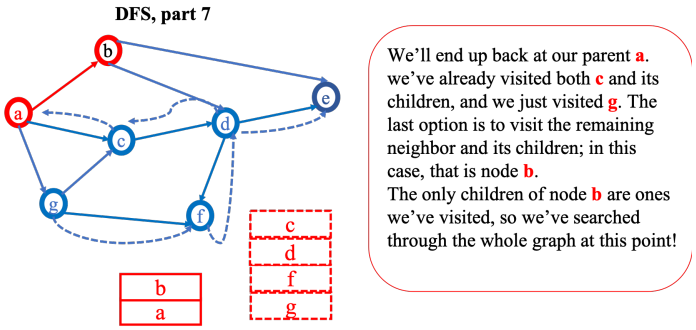


Figure 21: DFS 7

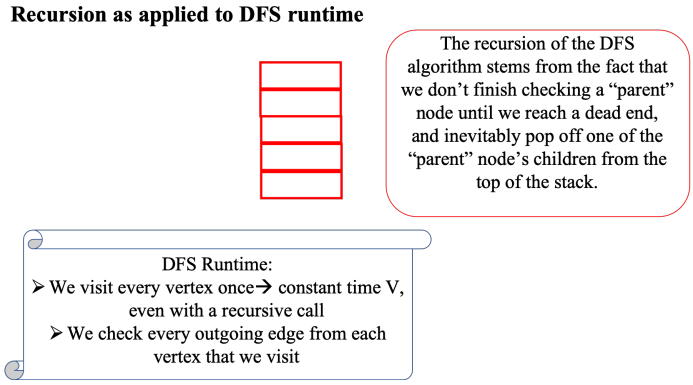


Figure 22: DFS 8

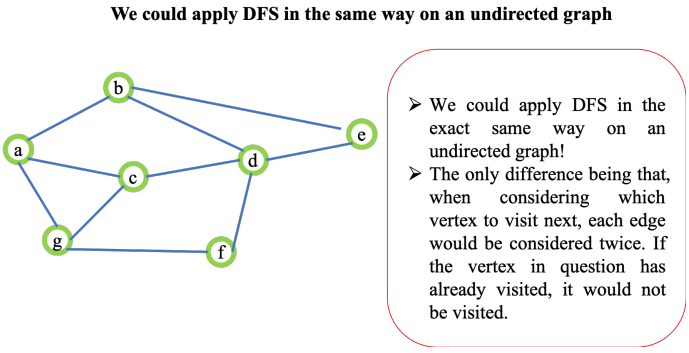


Figure 23: DFS 9

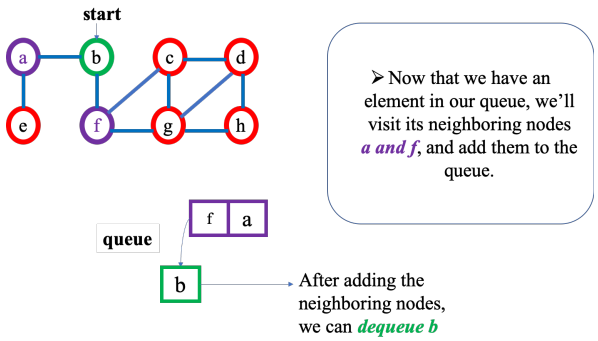


Figure 26: BFS 2

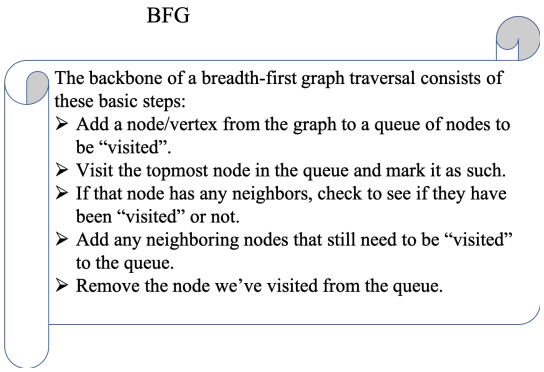


Figure 24: BFS

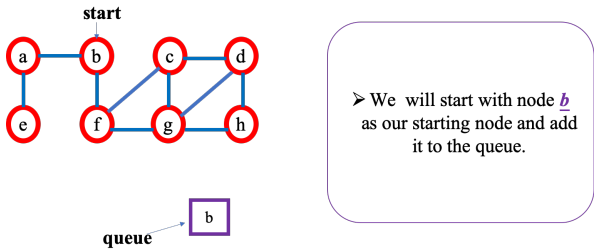


Figure 25: BFS 1

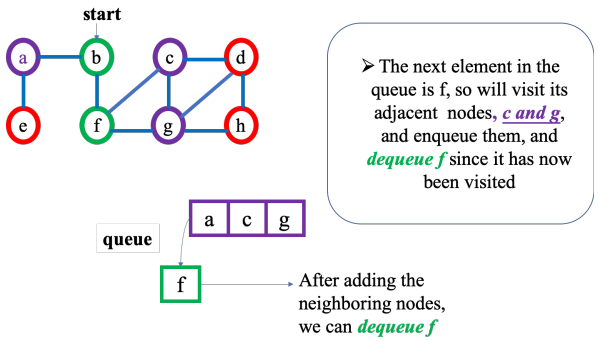


Figure 27: BFS 3

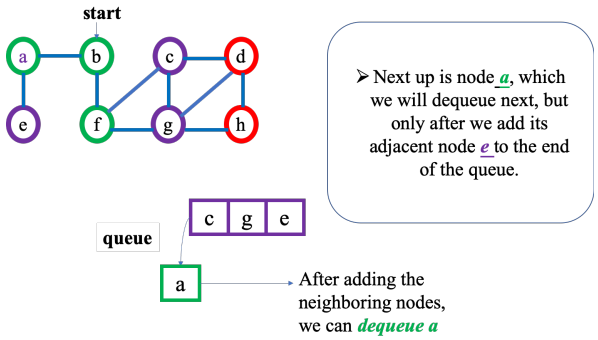


Figure 28: BFS 4

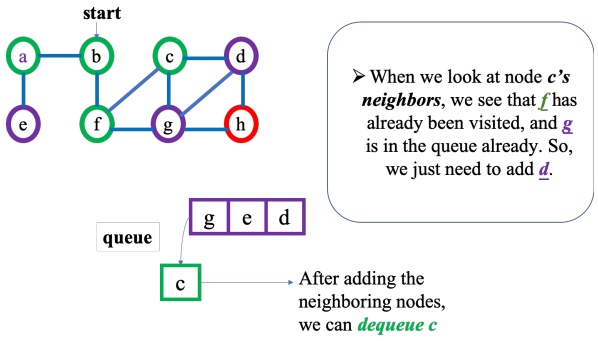


Figure 29: BFS 5

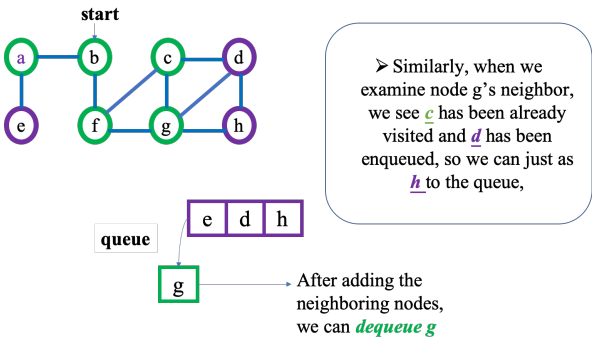


Figure 30: BFS 6

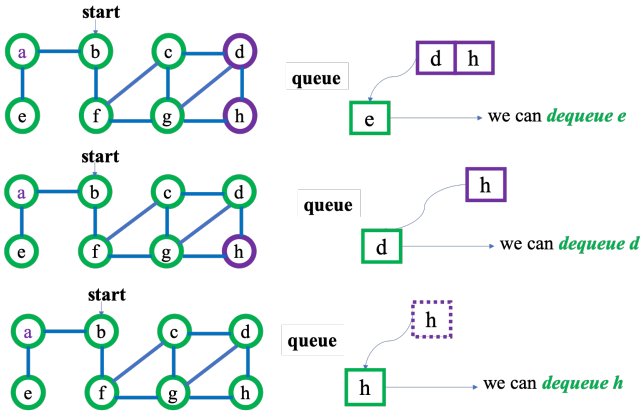


Figure 31: BFS 7