Graph Traversal

Depth-First Search

- Using Stack

Breadth-First Search

- Using Queue

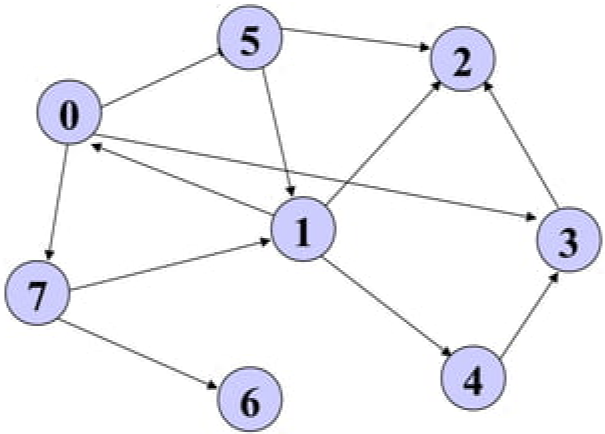
Overview

- Goal
  - To systematically visit the nodes of a graph
- A tree is a directed, acyclic, graph (DAG)
- If the graph is a tree,
  - DFS is exhibited by preorder, postorder, and (for binary trees) inorder traversals
  - BFS is exhibited by level-order traversal

Depth-First Search

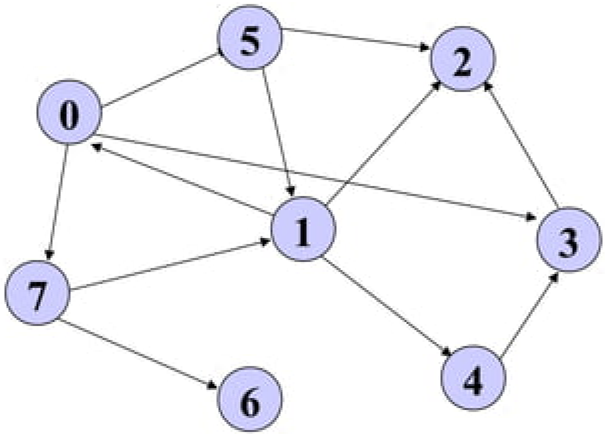
```
// recursive, preorder, depth-first search
void dfs (Node v) {
   if (v == null)
      return;

   if (v not yet visited)
      visit&mark(v);   // visit node before adjacent nodes

   for (each w adjacent to v)
      if (w has not yet been visited)
         dfs(w);

} // dfs
```

Example



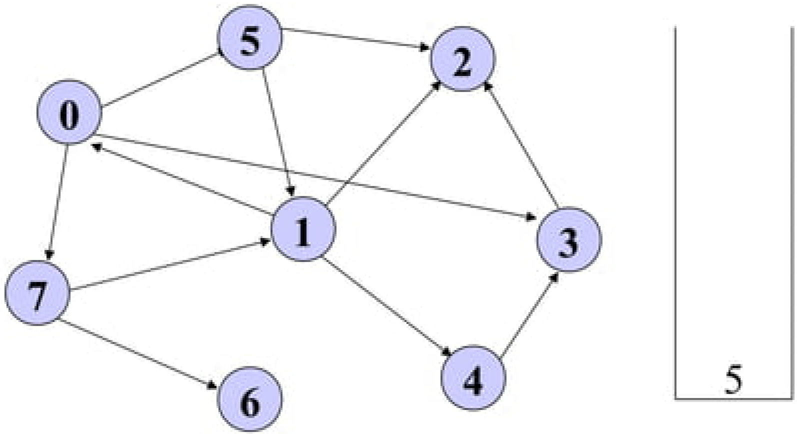
Policy: Visit adjacent nodes in increasing index order

DFS: Start with Node 5



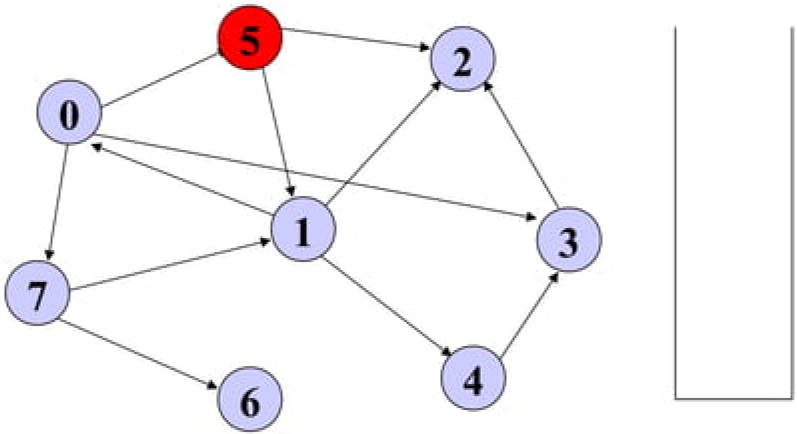
5 1 0 3 2 7 6 4

DFS: Start with Node 5



Push 5

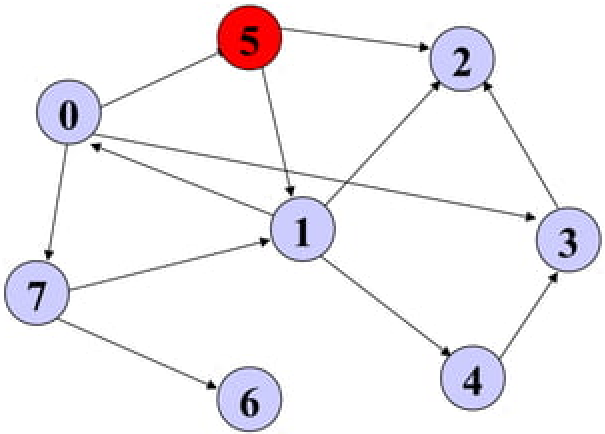
DFS: Start with Node 5

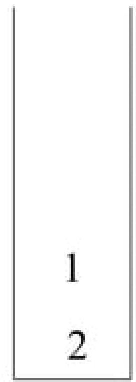


Pop/Visit/Mark 5

5

DFS: Start with Node 5

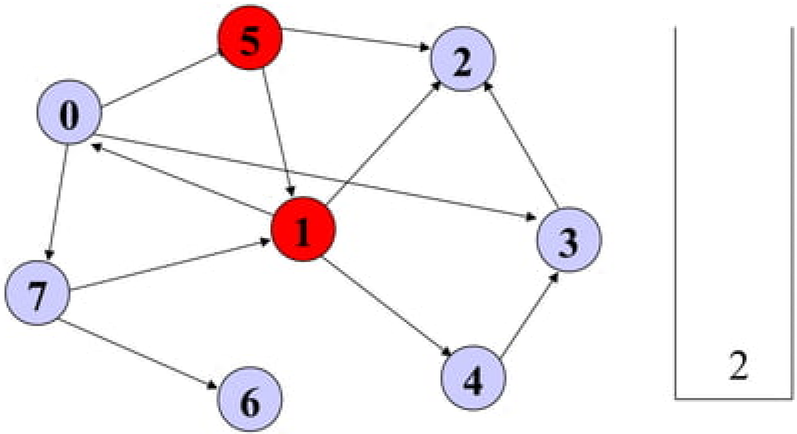




Push 2, Push 1

5

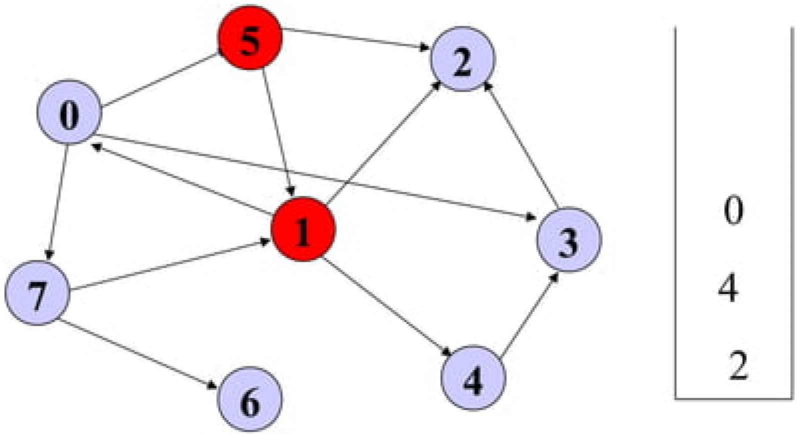
DFS: Start with Node 5



Pop/Visit/Mark 1

5 1

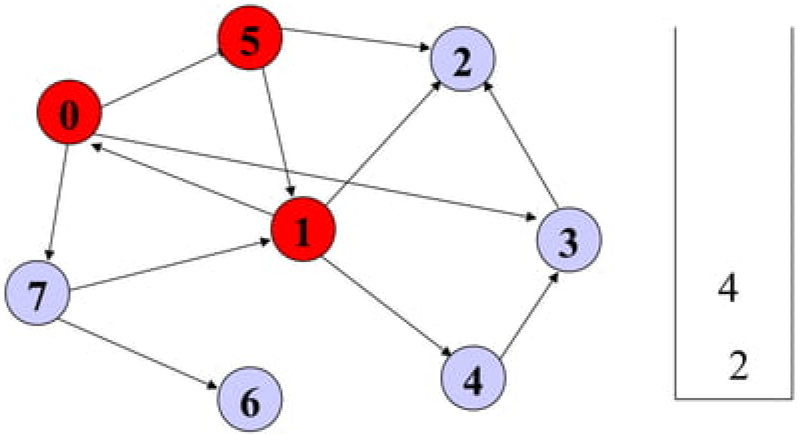
DFS: Start with Node 5



Push 4, Push 0

5 1

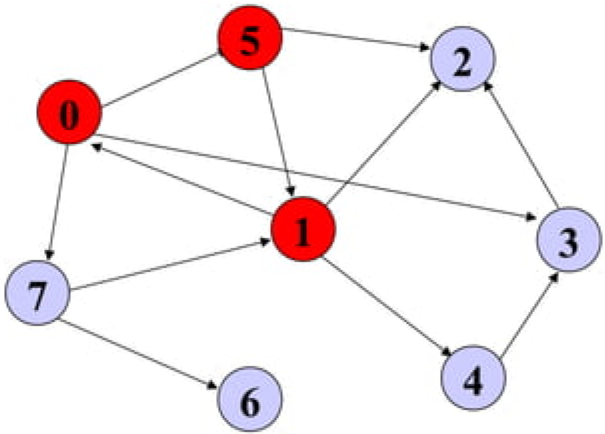
DFS: Start with Node 5

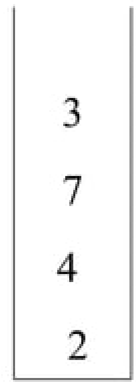


Pop/Visit/Mark 0

5 1 0

DFS: Start with Node 5

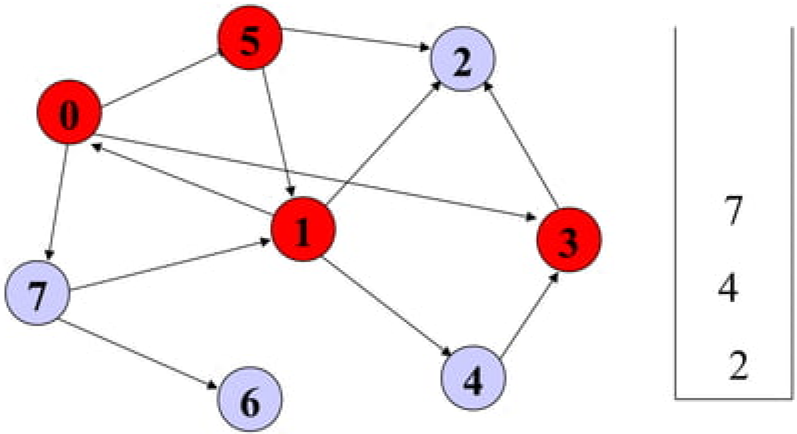




Push 7, Push 3

5 1 0

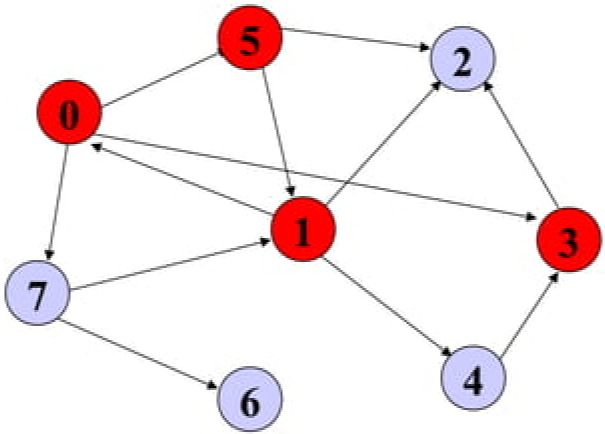
DFS: Start with Node 5

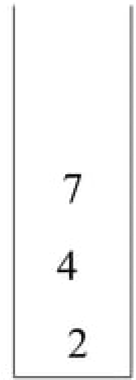


Pop/Visit/Mark 3

5 1 0 3

DFS: Start with Node 5

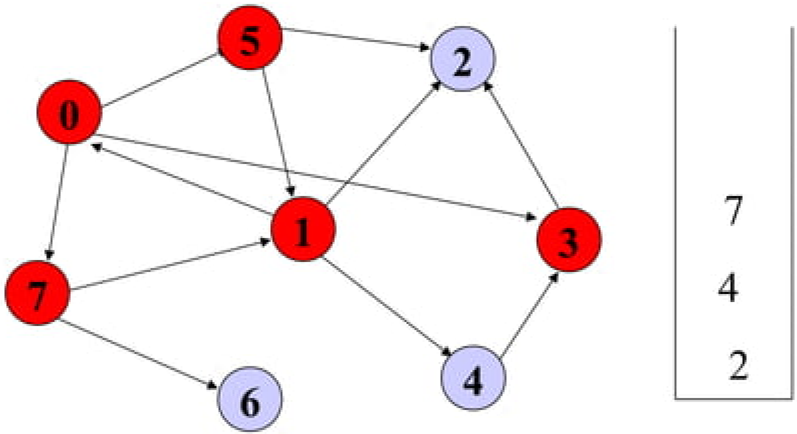




2 is already in stack

5 1 0 3

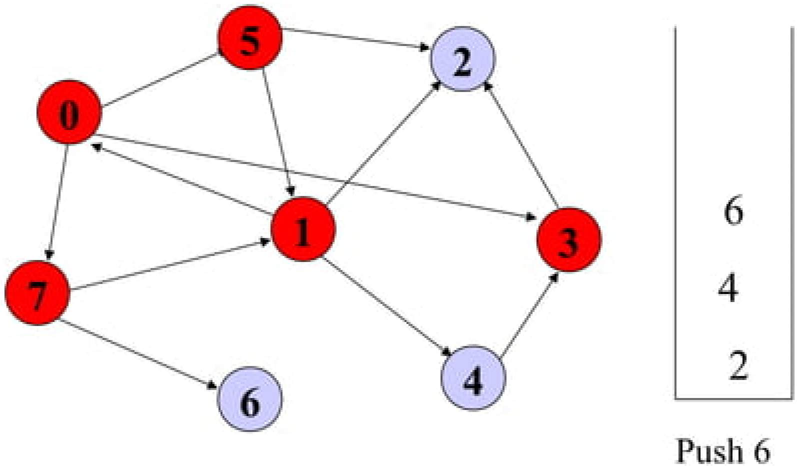
DFS: Start with Node 5



Pop/Mark/Visit 7

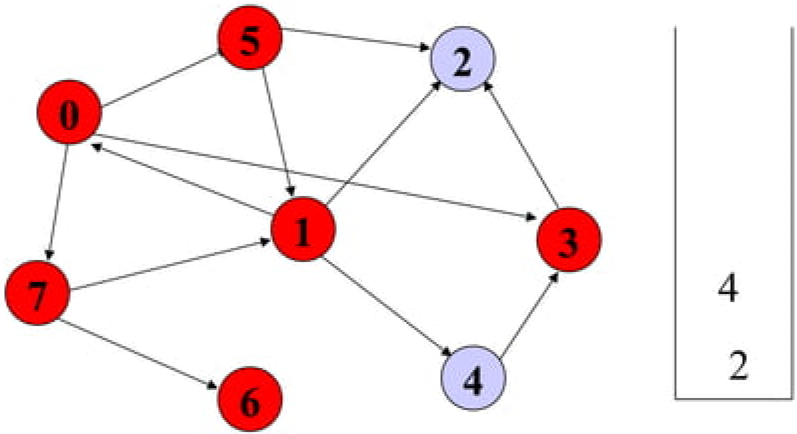
5 1 0 3 7

DFS: Start with Node 5



5 1 0 3 7

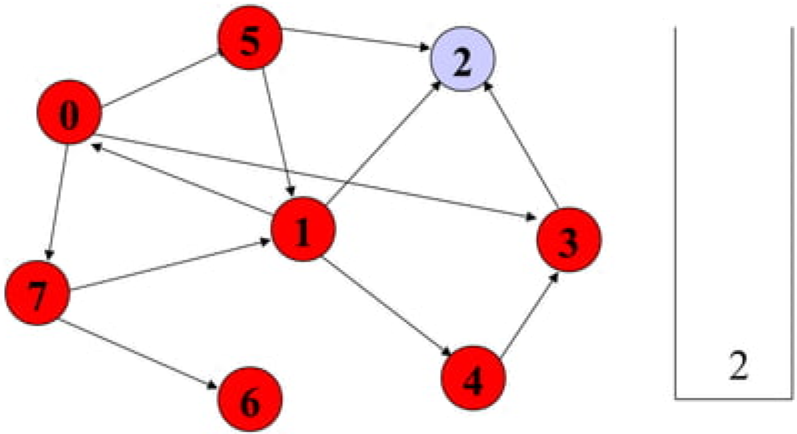
DFS: Start with Node 5



Pop/Mark/Visit 6

5 1 0 3 7 6

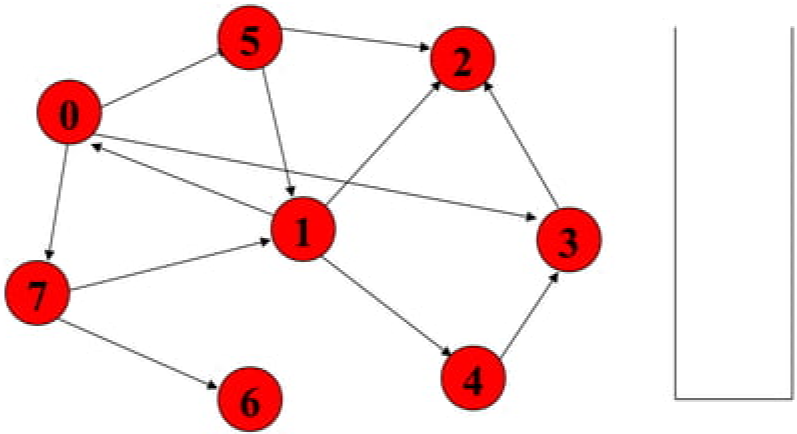
DFS: Start with Node 5



Pop/Mark/Visit 4

5 1 0 3 7 6 4

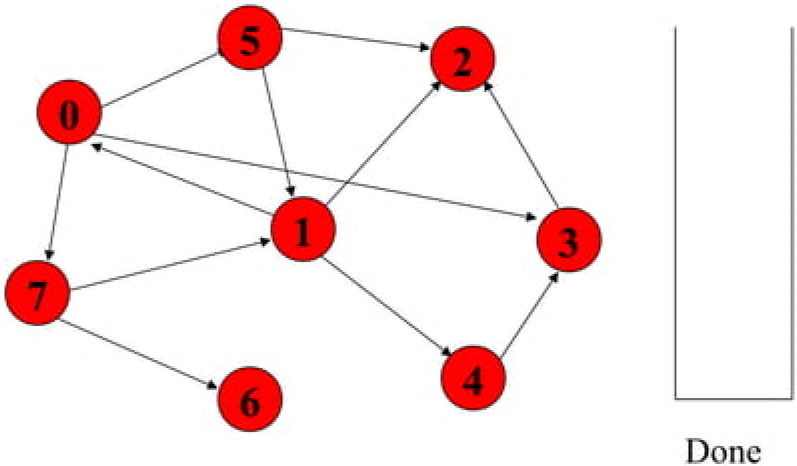
DFS: Start with Node 5



Pop/Mark/Visit 2

5 1 0 3 7 6 4 2

DFS: Start with Node 5



5 1 0 3 7 6 4 2

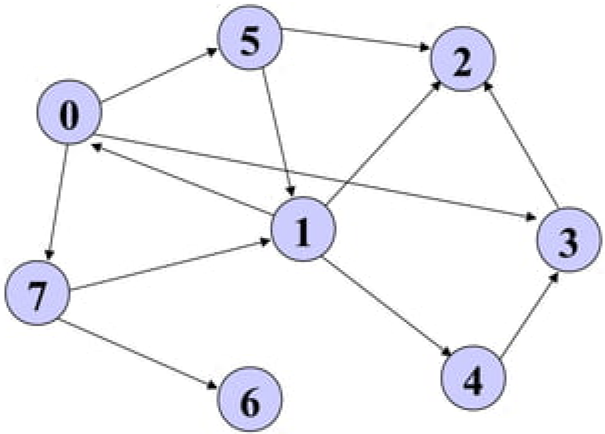
Breadth-first Search

- Ripples in a pond
- Visit designated node
- Then visited unvisited nodes a distance i away, where $i = 1, 2, 3$, etc.
- For nodes the same distance away, visit nodes in systematic manner (eg. increasing index order)

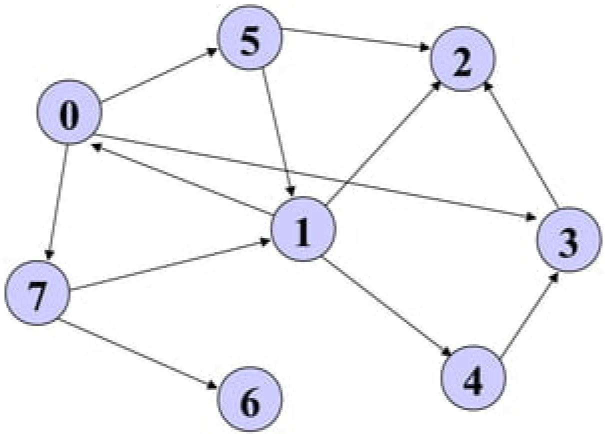
Breadth-First Search

```
// non-recursive, preorder, breadth-first search
void bfs (Node v) {
   if (v == null)
      return;

   enqueue(v);
   while (queue is not empty) {
      dequeue(v);
      if (v has not yet been visited)
         mark&visit(v);

      for (each w adjacent to v)
         if (w has not yet been visited && has not been queued)
            enqueue(w);
   } // while

} // bfs
```

BFS: Start with Node 5

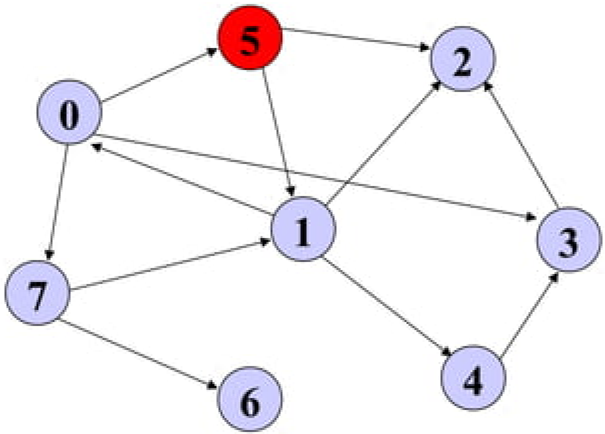


BFS: Start with Node 5



5

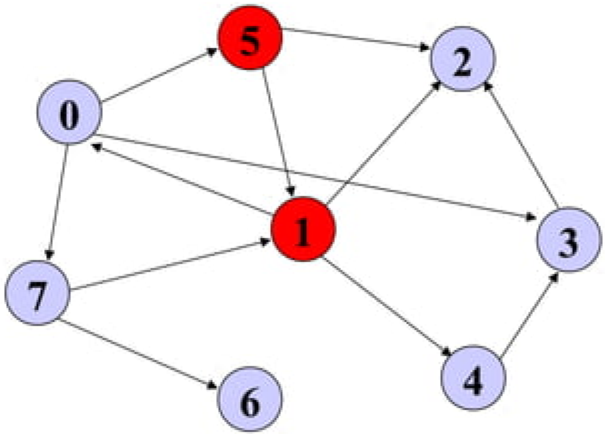
BFS: Node one-away



5

~~5~~ 1 2

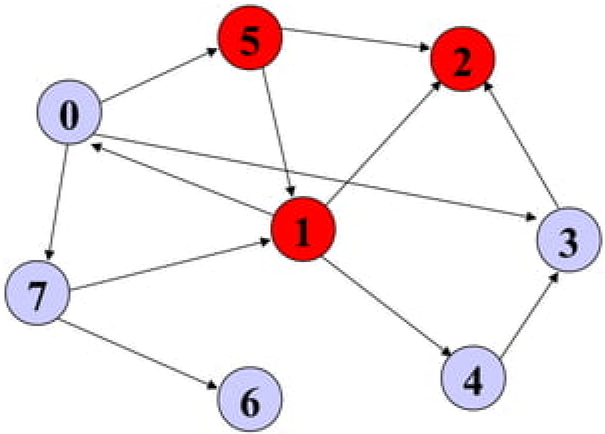
BFS: Visit 1 and add its adjacent nodes



5 1

~~5~~ ~~1~~ 2 0 4

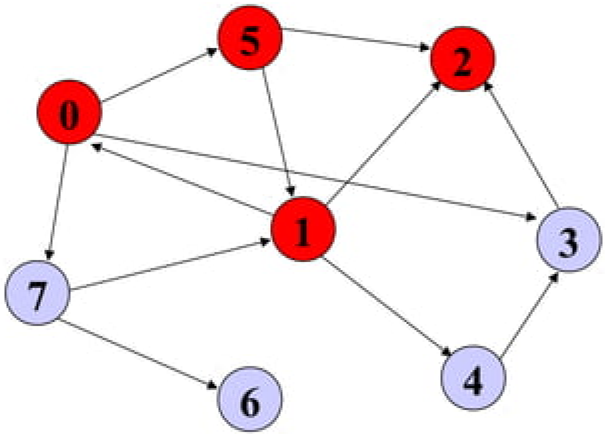
BFS: Visit 2 and add its adjacent nodes



5 1 2

~~5~~ ~~1~~ ~~2~~ 0 4

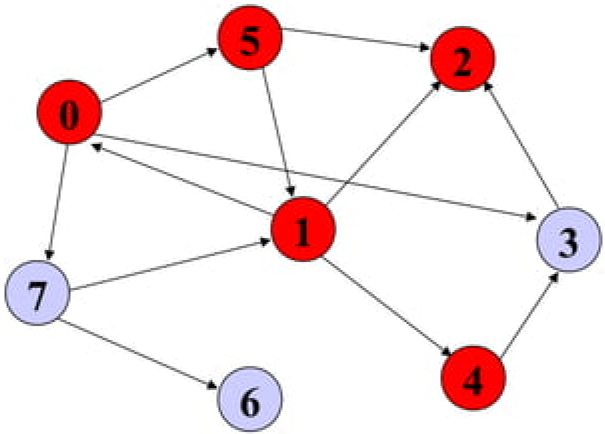
BFS: Visit 0 and add its adjacent nodes



5 1 2 0

~~5~~ ~~1~~ ~~2~~ ~~0~~ 4 3 7

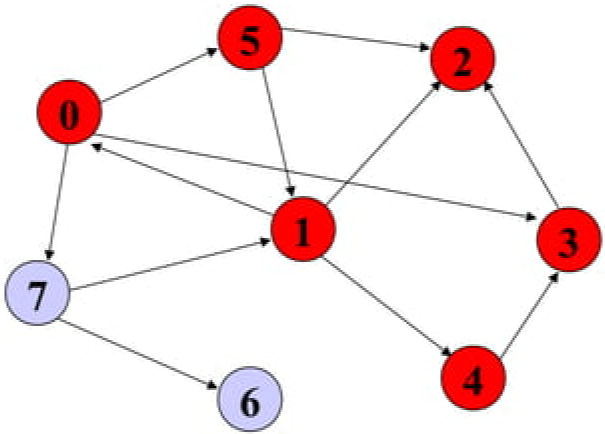
BFS: Visit 4 and add its adjacent nodes



5 1 2 0 4

~~5~~ ~~1~~ ~~2~~ ~~0~~ ~~4~~ 3 7

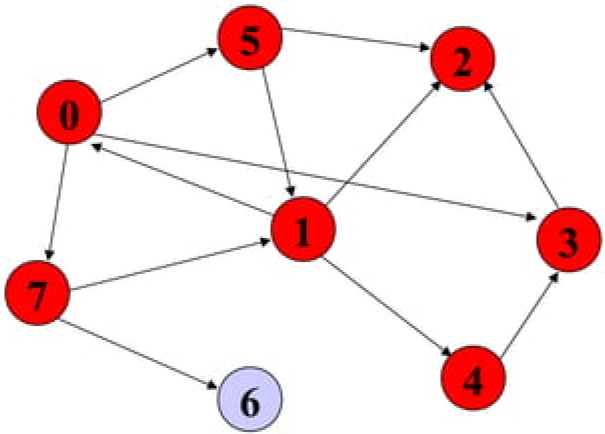
BFS: Visit 3 and add its adjacent nodes



5 1 2 0 4 3

~~5~~ ~~1~~ ~~2~~ ~~0~~ ~~4~~ ~~3~~ 7

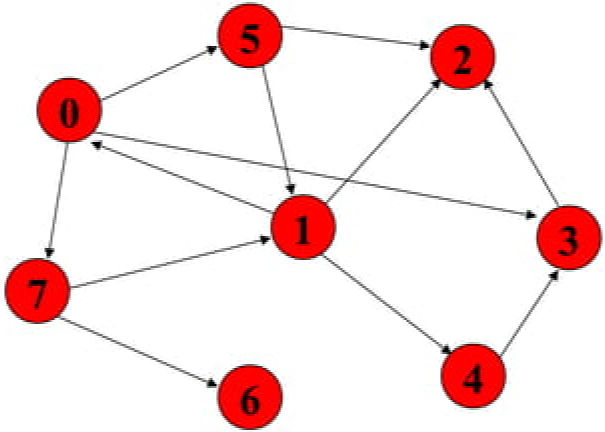
BFS: Visit 7 and add its adjacent nodes



5 1 2 0 4 3 7

~~5~~ ~~1~~ ~~2~~ ~~0~~ ~~4~~ ~~3~~ ~~7~~ 6

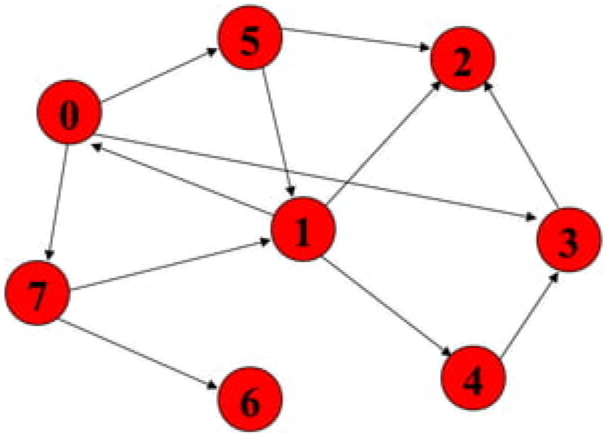
BFS: Visit 6 and add its adjacent nodes



5 1 2 0 4 3 7 6

~~5~~ ~~1~~ ~~2~~ ~~0~~ ~~4~~ ~~3~~ ~~7~~ ~~6~~

BFS Traversal Result



5 1 2 0 4 3 7 6

Applications of BFS

- Computing Distances: Given a source vertex *x*, compute the distance of all vertices from *x*.
- Checking for cycles in a graph: Given an undirected graph G, report whether there exists a cycle in the graph or not. (Note: won't work for directed graphs)
- Checking for bipartite graph: Given a graph, check whether it is bipartite or not? A graph is said to be bipartite if there is a partition of the vertex set V into two sets V_1 and V_2 such that if two vertices are adjacent, either both are in V_1 or both are in V_2.
- Reachability: Given a graph G and vertices *x* and *y*, determine if there exists a path from *x* to *y*.

Applications of DFS

- Computing Strongly Connected Components: A directed graph is strongly connected if there exists a path from every vertex to every other vertex. Trivial Algo: Perform DFS **n** times. Efficient Algo: Single DFS.
- Checking for Biconnected Graph: A graph is biconnected if removal of any vertex does not affect connectivity of the other vertices. Used to test if network is robust to failure of certain nodes. A single DFS traversal algorithm.
- Topological Ordering: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (x, y), vertex x comes before y in the ordering