Graph Algorithms: Breadth-First Searching

Pontus Ekberg

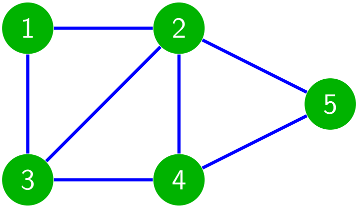
Uppsala University

(Based on previous material by Mohamed Faouzi Atig and Parosh Aziz Abdulla)

Graphs

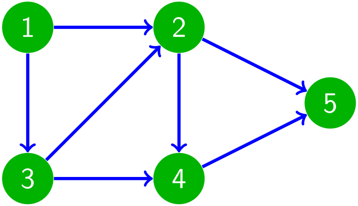
- A (directed) graph G is a pair (V, E) where:
  - V is a finite set of nodes (or vertices), and
  - $E \subseteq V \times V$ is a finite set of edges (or arcs)
- A graph $G = (V, E)$ is undirected if $(u, v) \in E$ implies that $(v, u) \in E$
- Applications: Modeling of
  - Social networks
  - World Wide Web
  - Computer networks
  - Road map
  - ...

Notations: Undirected Graphs



- Two nodes $u, v \in V$ are adjacent if there is an edge (u, v) in E
- An edge (u, v) is said to be incident to the nodes u and v
- A path is a sequence of nodes $u_1 u_2 \cdots u_n$ such that $(u_i, u_{i+1}) \in E$ for all $i \in \{1, 2, \ldots, n-1\}$. The node u_n is said to be reachable from u_1.

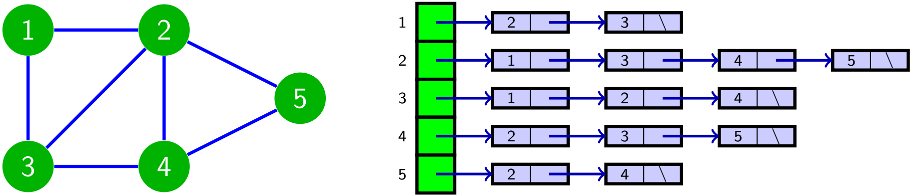
Notations: Directed Graphs



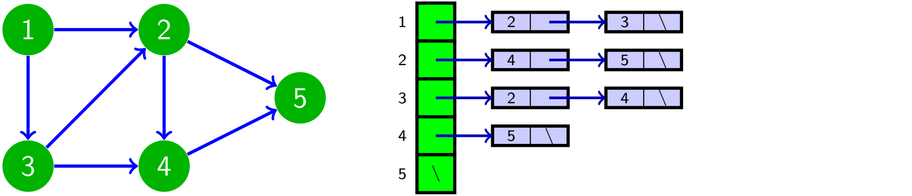
- An edge (u, v) is considered directed from u to v:
    - u is the tail and v is the head of this edge.
    - v is a direct successor of u.
    - u is a direct predecessor of v.
    - (u, v) is an input edge of v.
    - (u, v) is an output edge of u.
- A path is a sequence of nodes $u_1 u_2 \cdots u_n$ such that $(u_i, u_{i+1}) \in E$ for all $i \in \{1, 2, \ldots, n-1\}$. The node u_n is said to be reachable from u_1.

Representation of Graphs: Adjacency Lists

Undirected Graph:



Directed Graph:



Representation of Graphs: Adjacency Lists

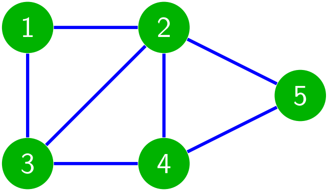
- An adjacency-list representation of a graph $G = (V, E)$ consists of:
    - An array Adj of $|V|$ lists, one per node.
    - For each u, the adjacency list $Adj[u]$ contains all the nodes v such that there is an edge $(u, v) \in E$

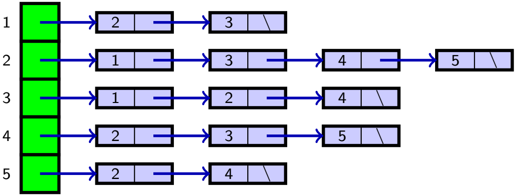
- Pseudocode Conventions:
    - We denote the node set of G by $G.V$ and its edge set by $G.E$
    - We consider the array Adj as an attribute of the graph G (i.e., To access to the array we use $G.Adj$)

- Some Observations:
    - The adjacency list allows to represent undirected and directed graphs.
    - The sum of the lengths of all the adjacency lists is $|E|$

Adjacency Lists: Complexity

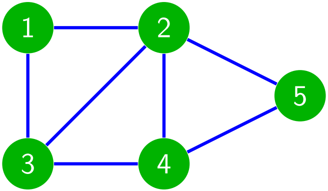
- Space complexity: $O(|V| + |E|)$
- Time complexity of checking if (u, v) is in E: $O(|V|)$
- Time complexity to list all nodes adjacent to u: $O(|V|)$





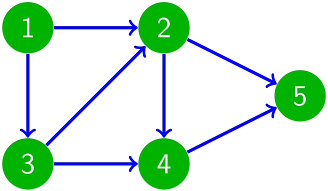
Representation of Graphs: Adjacency Matrix

Undirected Graph:



$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Directed Graph:



$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Representation of Graphs: Adjacency Matrix

- An adjacency-matrix representation of a graph $G = (V, E)$, with $V = \{1, 2, \ldots, |V|\}$ consists of:
  - A $(|V| \times |V|)$-matrix A such that for every $(i, j) \in |V| \times |V|$, we have:

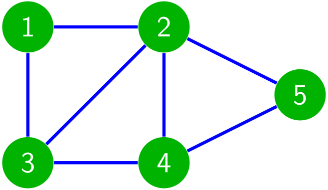
$$A[i,j] = \begin{cases} 1 & if\, (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Pseudocode Conventions:
  - We consider the matrix A as an attribute of the graph G (i.e., To access to the matrix we use $G.A$)

- Some Observations:
  - The adjacency matrix allows to represent undirected and directed graphs.

Adjacency Matrix: Complexity

- Space complexity: $O(|V|^2)$
- Time complexity of checking if (u, v) is in E: $O(1)$
- Time complexity to list all nodes adjacent to u: $O(|V|)$



$$
\begin{array}{c c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}
\end{array}
$$

Representations

- Adjacency List
  - Good for sparse graphs (i.e., $|E| \ll |V|^2$)
  - Less good for dense graphs (i.e., $|E| \approx |V|^2$)
  - Only consumes space to represent the existing edges
- Adjacency Matrix
  - Less good for sparse graphs (i.e., $|E| \ll |V|^2$)
  - Good for dense graphs (i.e., $|E| \approx |V|^2$)
  - Will always use $O(|V|)^2$ space to represent the edges.
  - Checking if an edge is in the graph can be done very efficiently.
  - Lower space and time overheads for dense graphs.

Graph Searching

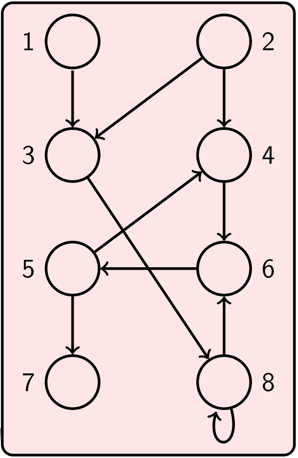
- Goal: Traversing all the nodes of a graph that are reachable from a given node v
- A node u is reachable from a node v if
  - $u = v$, or
  - v is adjacent to u, or
  - v is adjacent to a node w, and u is reachable from w.
- Different searching algorithms:
  - Breadth-first search
  - Depth-first search

Breadth-First Search

- One of the simplest algorithms for searching a graph.
- The archetype for many important graph algorithms
- Input: A graph $G = (V, E)$ and a node $s \in V$
- Output:
  - The set of nodes reachable from s
  - Compute the distance (smallest number of edges) from s to each reachable node
  - Produce a *breadth-first tree* with root s that contains all reachable nodes
- The algorithm works on both directed and undirected graphs

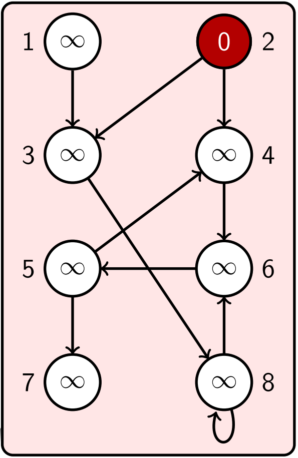
Breadth-First Search: Principle

- The algorithm discovers all nodes at distance k (smallest number of edges) from s before discovering any nodes at distance $k + 1$:
    - Initially, all the nodes are at distance ∞ from s (except s whose distance is 0)
    - Find all nodes S_1 at distance 1 from s
    - Find all nodes S_2 at distance 1 from S_1
    - Find all nodes S_3 at distance 1 from S_2
    - Etc.



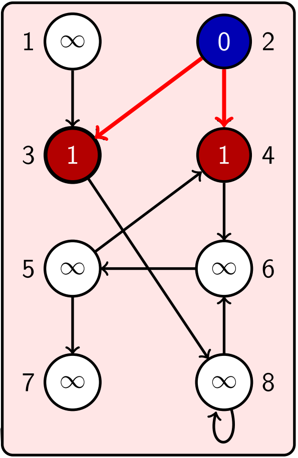
Breadth-First Search: Principle

- The algorithm discovers all nodes at distance k (smallest number of edges) from s before discovering any nodes at distance $k+1$:
  - Initially, all the nodes are at distance ∞ from s (except s whose distance is 0)
  - Find all nodes S_1 at distance 1 from s
  - Find all nodes S_2 at distance 1 from S_1
  - Find all nodes S_3 at distance 1 from S_2
  - Etc.



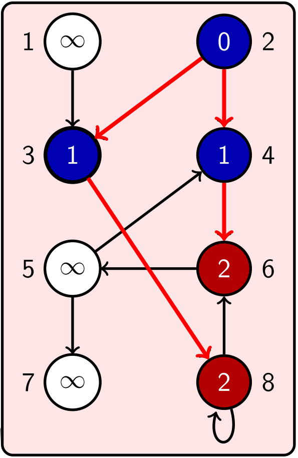
Breadth-First Search: Principle

- The algorithm discovers all nodes at distance k (smallest number of edges) from s before discovering any nodes at distance $k+1$:
  - Initially, all the nodes are at distance ∞ from s (except s whose distance is 0)
  - Find all nodes S_1 at distance 1 from s
  - Find all nodes S_2 at distance 1 from S_1
  - Find all nodes S_3 at distance 1 from S_2
  - Etc.



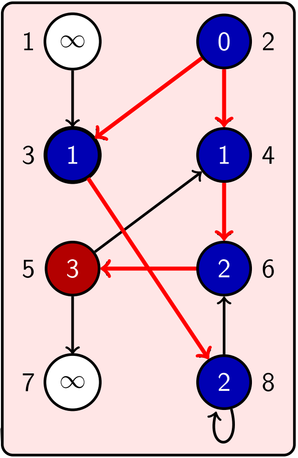
Breadth-First Search: Principle

- The algorithm discovers all nodes at distance k (smallest number of edges) from s before discovering any nodes at distance $k+1$:
  - Initially, all the nodes are at distance ∞ from s (except s whose distance is 0)
  - Find all nodes S_1 at distance 1 from s
  - Find all nodes S_2 at distance 1 from S_1
  - Find all nodes S_3 at distance 1 from S_2
  - Etc.



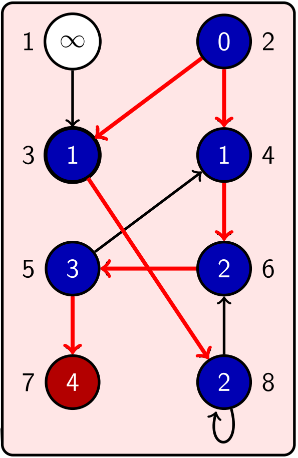
Breadth-First Search: Principle

- The algorithm discovers all nodes at distance k (smallest number of edges) from s before discovering any nodes at distance $k+1$:
  - Initially, all the nodes are at distance ∞ from s (except s whose distance is 0)
  - Find all nodes S_1 at distance 1 from s
  - Find all nodes S_2 at distance 1 from S_1
  - Find all nodes S_3 at distance 1 from S_2
  - Etc.



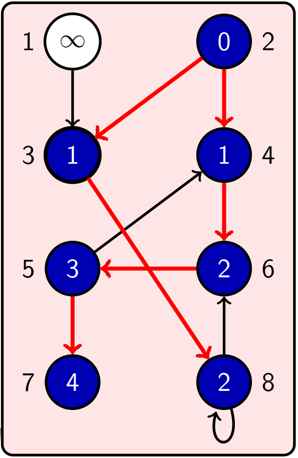
Breadth-First Search: Principle

- The algorithm discovers all nodes at distance k (smallest number of edges) from s before discovering any nodes at distance $k+1$:
  - Initially, all the nodes are at distance ∞ from s (except s whose distance is 0)
  - Find all nodes S_1 at distance 1 from s
  - Find all nodes S_2 at distance 1 from S_1
  - Find all nodes S_3 at distance 1 from S_2
  - Etc.



Breadth-First Search: Principle

- The algorithm discovers all nodes at distance k (smallest number of edges) from s before discovering any nodes at distance $k+1$:
  - Initially, all the nodes are at distance ∞ from s (except s whose distance is 0)
  - Find all nodes S_1 at distance 1 from s
  - Find all nodes S_2 at distance 1 from S_1
  - Find all nodes S_3 at distance 1 from S_2
  - Etc.



Breadth-First Search: Algorithm

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    do u.color ← WHITE
       u.d ← ∞
       u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ G.Adj[u]
           do if v.color = WHITE
                 then v.color ← RED
                      v.d ← u.d + 1
                      v.π ← u
                      ENQUEUE(Q, v)
       u.color ← BLUE
```

During the search, node u has the following attributes:

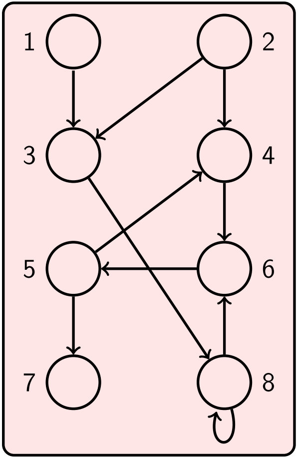
- $u.d$: distance from s to u.
- $u.color$:
  - *WHITE* : not discovered
  - *RED* : discovered but not analyzed
  - *BLUE* : finished, i.e., discovered and analyzed
- $u.\pi$: predecessor of u in the analysis.

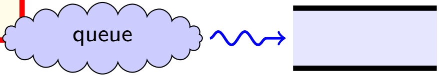
Q is a FIFO queue consists of the set of *RED* nodes

Breadth-First Search

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
                do if v.color = WHITE
                      then v.color ← RED
                           v.d ← u.d + 1
                           v.π ← u
                           ENQUEUE(Q, v)
           u.color ← BLUE
```

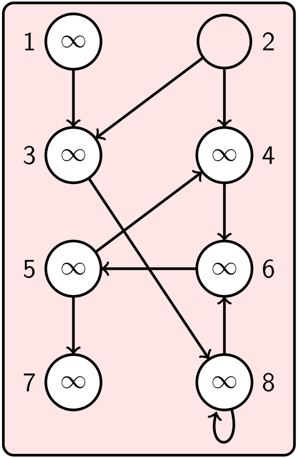


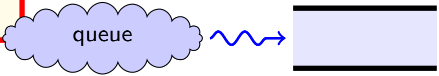


Breadth-First Search

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```

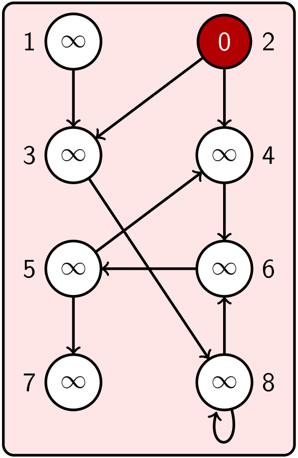


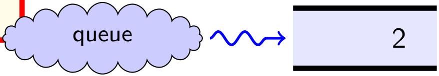


Breadth-First Search

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
      do u.color ← WHITE
         u.d ← ∞
         u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
      do u ← DEQUEUE(Q)
         for each v ∈ G.Adj[u]
             do if v.color = WHITE
                   then v.color ← RED
                        v.d ← u.d + 1
                        v.π ← u
                        ENQUEUE(Q, v)
         u.color ← BLUE
```

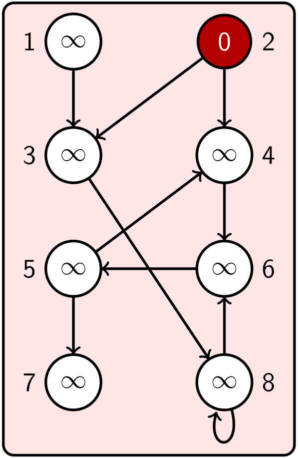


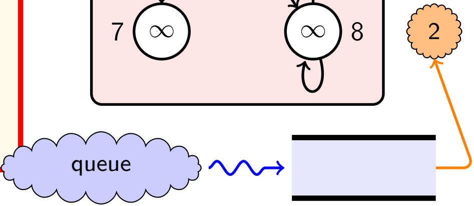


Breadth-First Search

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```

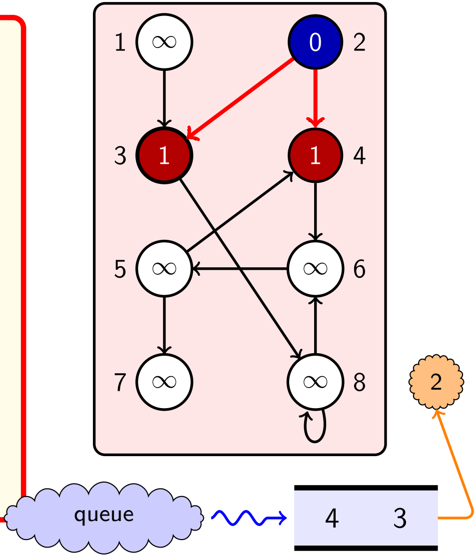




Breadth-First Search

BFS(G, s)

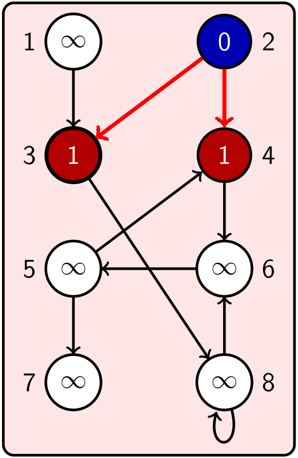
```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```

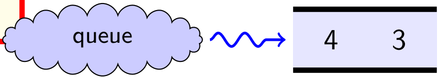


Breadth-First Search

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
      do u.color ← WHITE
         u.d ← ∞
         u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
      do u ← DEQUEUE(Q)
         for each v ∈ G.Adj[u]
             do if v.color = WHITE
                   then v.color ← RED
                        v.d ← u.d + 1
                        v.π ← u
                        ENQUEUE(Q, v)
         u.color ← BLUE
```

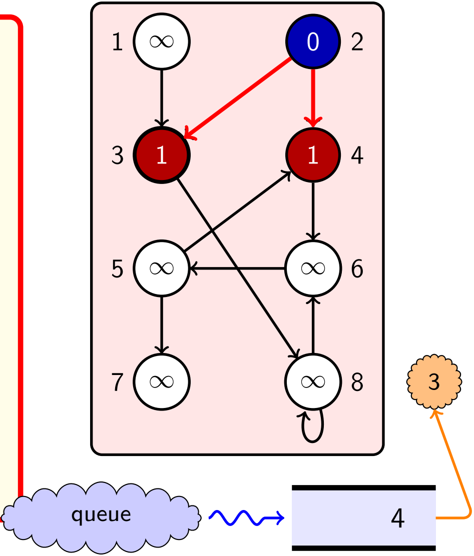




Breadth-First Search

BFS(G, s)

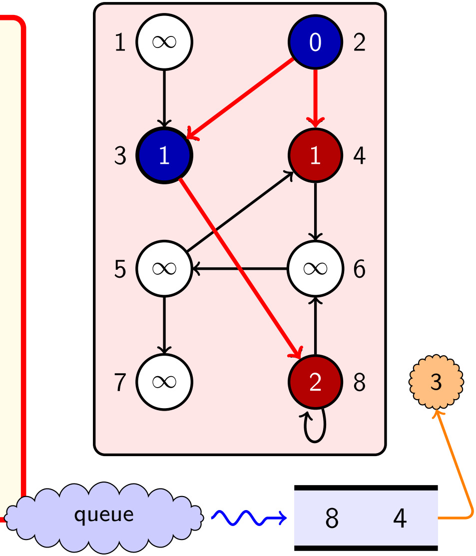
```
for each vertex u ∈ G.V − {s}
      do u.color ← WHITE
         u.d ← ∞
         u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
      do u ← DEQUEUE(Q)
         for each v ∈ G.Adj[u]
             do if v.color = WHITE
                   then v.color ← RED
                        v.d ← u.d + 1
                        v.π ← u
                        ENQUEUE(Q, v)
         u.color ← BLUE
```



Breadth-First Search

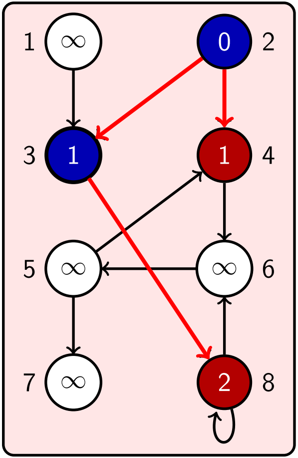
BFS(G, s)

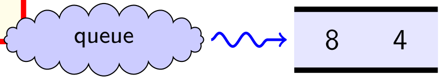
```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
            u.d ← ∞
            u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
            for each v ∈ G.Adj[u]
                  do if v.color = WHITE
                          then v.color ← RED
                                  v.d ← u.d + 1
                                  v.π ← u
                                  ENQUEUE(Q, v)
            u.color ← BLUE
```



Breadth-First Search

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
      do u.color ← WHITE
         u.d ← ∞
         u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
      do u ← DEQUEUE(Q)
         for each v ∈ G.Adj[u]
             do if v.color = WHITE
                   then v.color ← RED
                        v.d ← u.d + 1
                        v.π ← u
                        ENQUEUE(Q, v)
         u.color ← BLUE
```

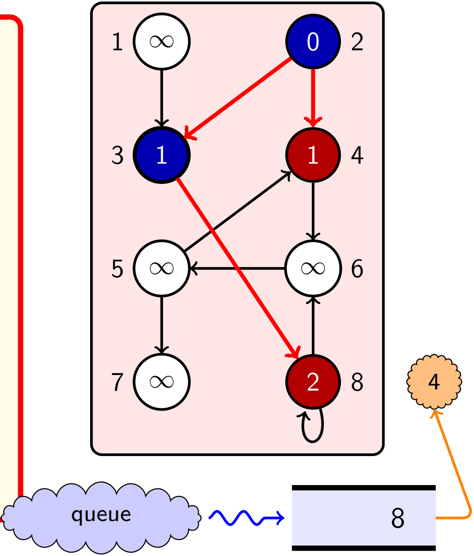




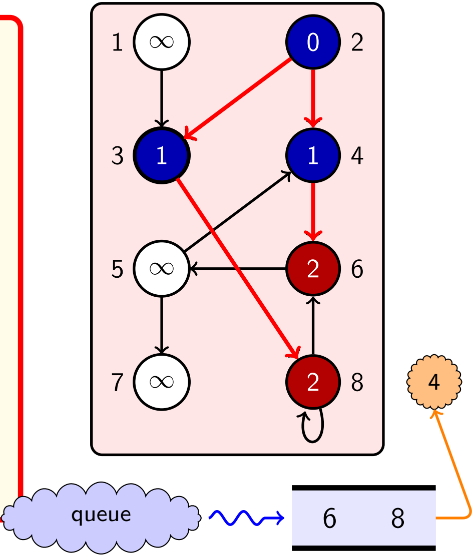
Breadth-First Search

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```

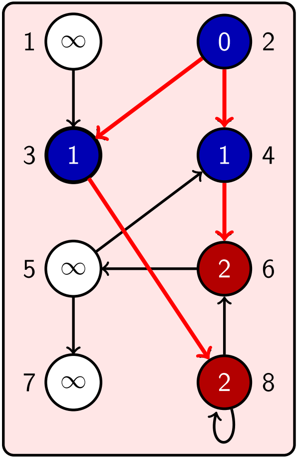


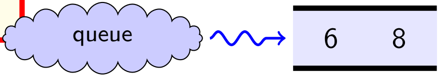
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
                do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```



Breadth-First Search

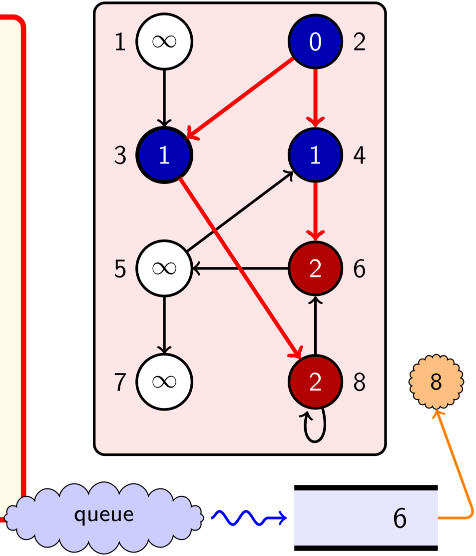
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
                do if v.color = WHITE
                        then v.color ← RED
                             v.d ← u.d + 1
                             v.π ← u
                             ENQUEUE(Q, v)
           u.color ← BLUE
```





Breadth-First Search

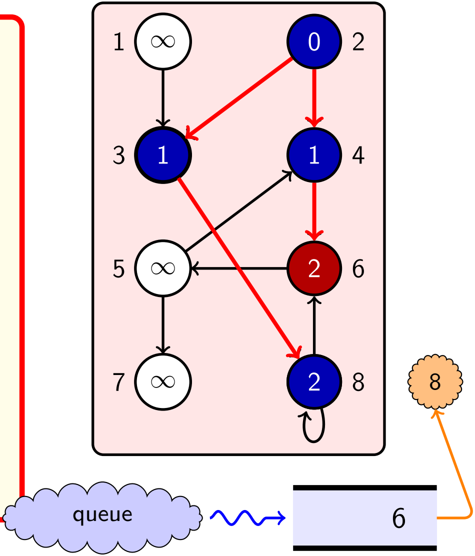
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```



Breadth-First Search

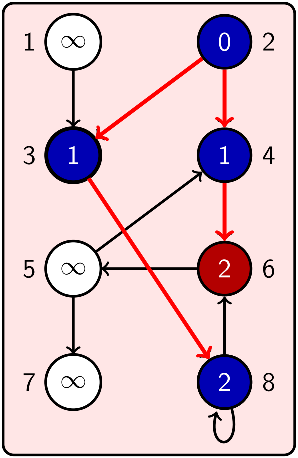
BFS(G, s)

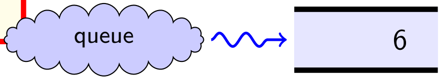
```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                      then v.color ← RED
                           v.d ← u.d + 1
                           v.π ← u
                           ENQUEUE(Q, v)
           u.color ← BLUE
```



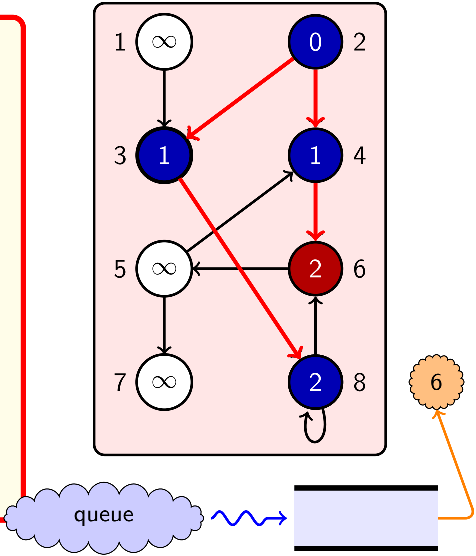
Breadth-First Search

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```





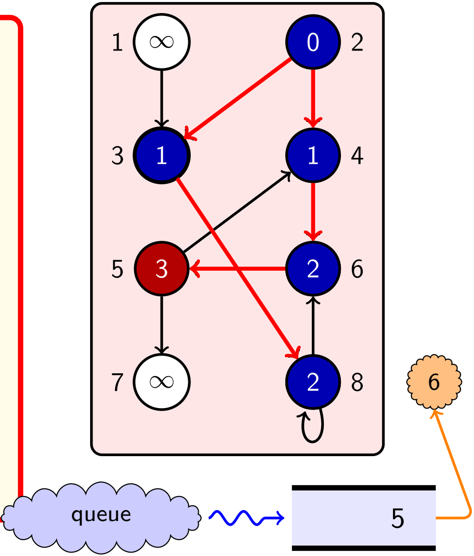
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
      do u.color ← WHITE
         u.d ← ∞
         u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
      do u ← DEQUEUE(Q)
         for each v ∈ G.Adj[u]
             do if v.color = WHITE
                   then v.color ← RED
                        v.d ← u.d + 1
                        v.π ← u
                        ENQUEUE(Q, v)
         u.color ← BLUE
```



Breadth-First Search

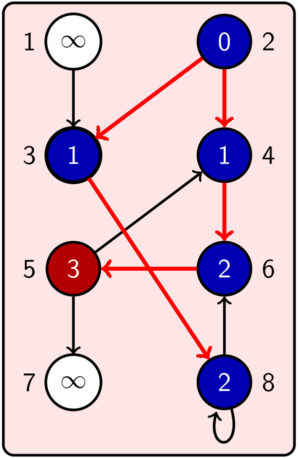
BFS(G, s)

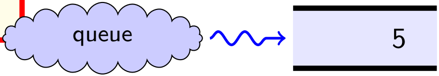
```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
                do if v.color = WHITE
                      then v.color ← RED
                           v.d ← u.d + 1
                           v.π ← u
                           ENQUEUE(Q, v)
           u.color ← BLUE
```



Breadth-First Search

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
                do if v.color = WHITE
                      then v.color ← RED
                           v.d ← u.d + 1
                           v.π ← u
                           ENQUEUE(Q, v)
           u.color ← BLUE
```

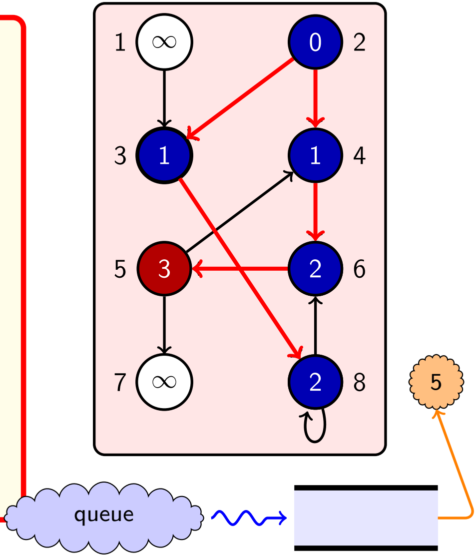




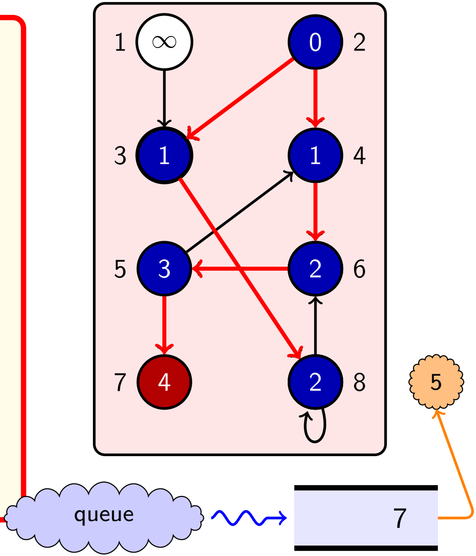
Breadth-First Search

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```



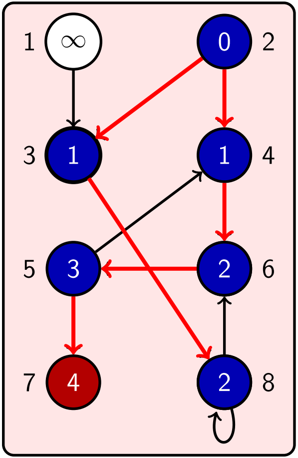
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
                do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```

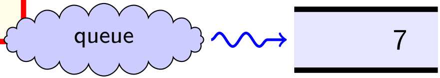


Breadth-First Search

BFS(G, s)

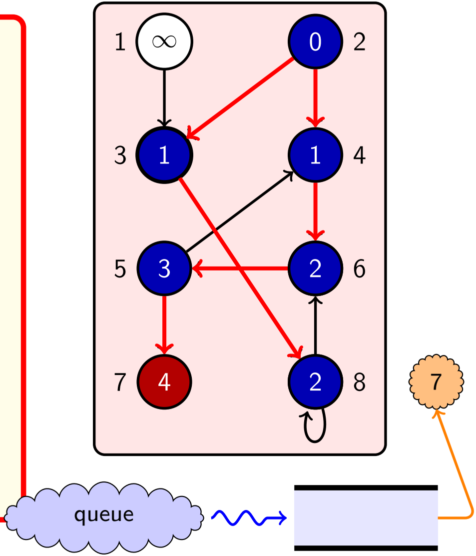
```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
            u.d ← ∞
            u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
            for each v ∈ G.Adj[u]
                do if v.color = WHITE
                    then v.color ← RED
                         v.d ← u.d + 1
                         v.π ← u
                         ENQUEUE(Q, v)
            u.color ← BLUE
```





Breadth-First Search

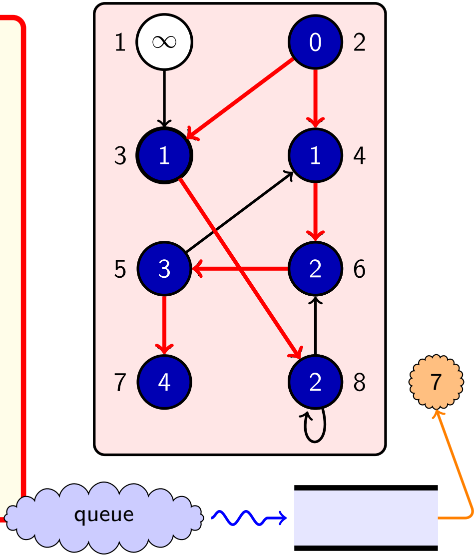
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
                do if v.color = WHITE
                      then v.color ← RED
                           v.d ← u.d + 1
                           v.π ← u
                           ENQUEUE(Q, v)
           u.color ← BLUE
```



Breadth-First Search

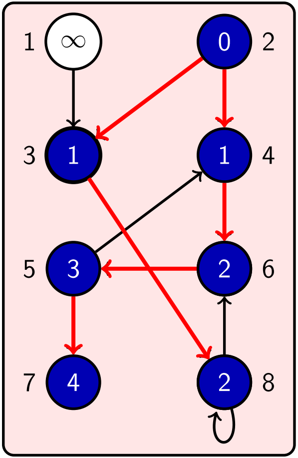
BFS(G, s)

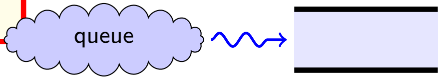
```
for each vertex u ∈ G.V − {s}
        do u.color ← WHITE
           u.d ← ∞
           u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        do u ← DEQUEUE(Q)
           for each v ∈ G.Adj[u]
               do if v.color = WHITE
                     then v.color ← RED
                          v.d ← u.d + 1
                          v.π ← u
                          ENQUEUE(Q, v)
           u.color ← BLUE
```



Breadth-First Search

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
      do u.color ← WHITE
         u.d ← ∞
         u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
      do u ← DEQUEUE(Q)
         for each v ∈ G.Adj[u]
             do if v.color = WHITE
                   then v.color ← RED
                        v.d ← u.d + 1
                        v.π ← u
                        ENQUEUE(Q, v)
         u.color ← BLUE
```





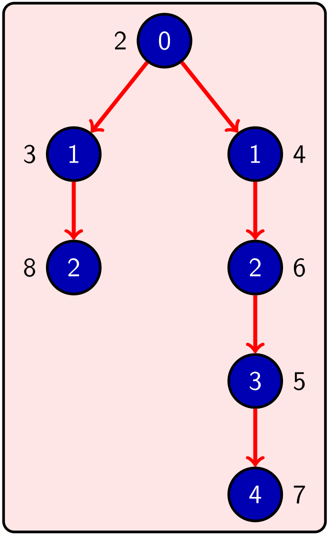
Breadth-First Search: Complexity

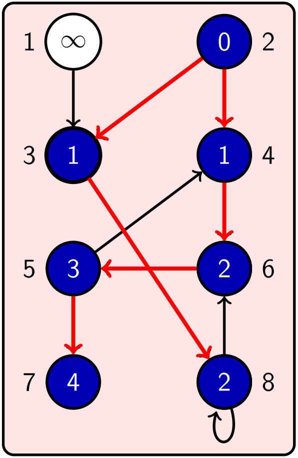
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    do u.color ← WHITE
       u.d ← ∞
       u.π ← NIL
s.color ← RED
s.d ← 0
s.π ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ G.Adj[u]
           do if v.color = WHITE
                 then v.color ← RED
                      v.d ← u.d + 1
                      v.π ← u
                      ENQUEUE(Q, v)
       u.color ← BLUE
```

- Initialization costs $O(|V|)$
- The operations of enqueueing and dequeueing take $O(1)$ time
- Each node is enqueued at most once (when the color changes from *WHITE* to *RED*). Consequently, each node dequeued at most once.
- The **while** loop is executed at most $|V|$ times
- The adjacency list of each node is scanned at most once (when the node is dequeued). The sum of lengths of adjacency list is $O(|E|)$
- Total time $= O(|V| + |E|)$

Breadth-First Tree

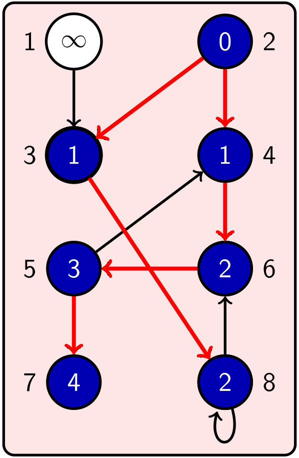
The predecessor subgraph of G is defined by $G_\pi = (V_\pi, E_\pi)$ where $V_\pi = \{v \in V \mid v.\pi \neq NIL\} \cup \{s\}$ and $E_\pi = \{(v.\pi, v) \mid v \in V_\pi \setminus \{s\}\}$





Printing the Shortest Path

PRINT-PATH(G, s, v)
```
if v = s
    then print s
    else if v.π = NIL
            then print v not reachable
            else PRINT-PATH(G, s, v.π)
                 print v
```



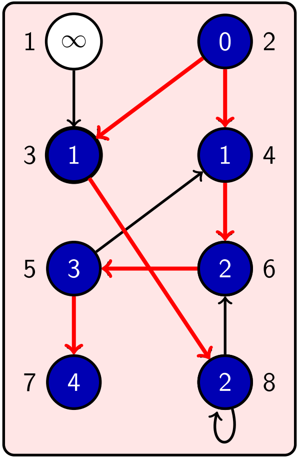
Printing the Shortest Path

PRINT-PATH(G, s, v)

```
if v = s
    then print s
    else if v.π = NIL
            then print v not reachable
            else PRINT-PATH(G, s, v.π)
                 print v
```

PRINT-PATH$(G, 2, 1)$

1 not reachable



Printing the Shortest Path

```
PRINT-PATH(G, s, v)
if v = s
    then print s
    else if v.π = NIL
            then print v not reachable
            else PRINT-PATH(G, s, v.π)
                 print v
```

PRINT-PATH$(G, 2, 8)$

2 3 8