SPANNING TREES

Spanning tree

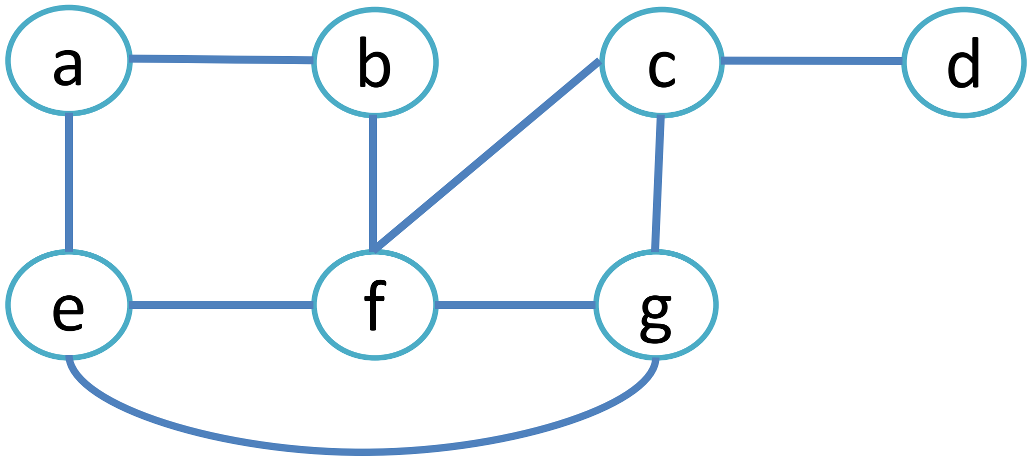
Let G be a simple graph.
A ***spanning tree*** of G is a subgraph of G that is a tree containing every vertex of G.

A simple graph G must be connected
→ Because there is a path in the spanning tree between any two vertices

Every connected simple graph has a spanning tree.

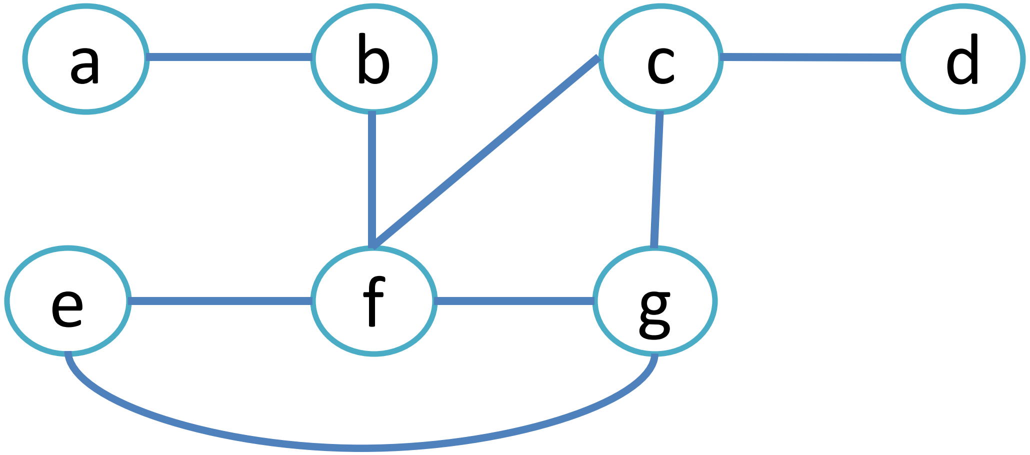
Example 1

Find a spanning tree of the simple graph *G*.



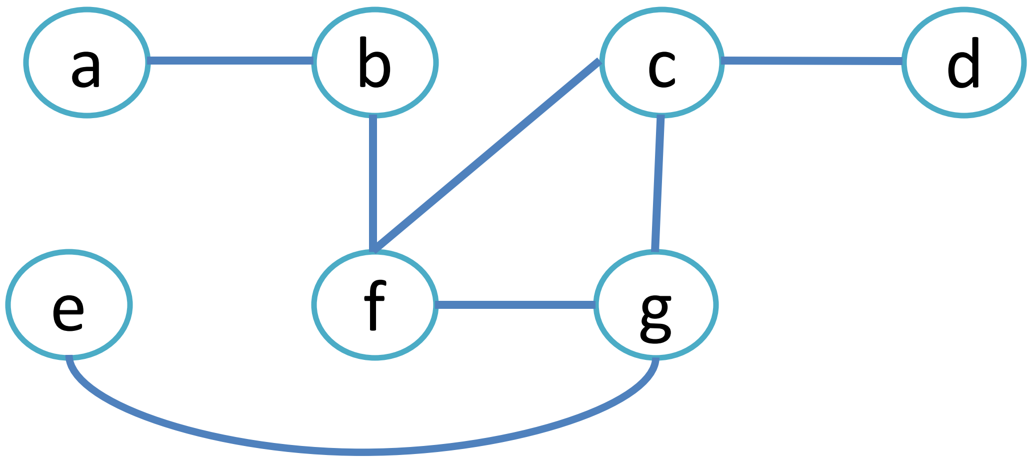
Example 1

Find a spanning tree of the simple graph G.



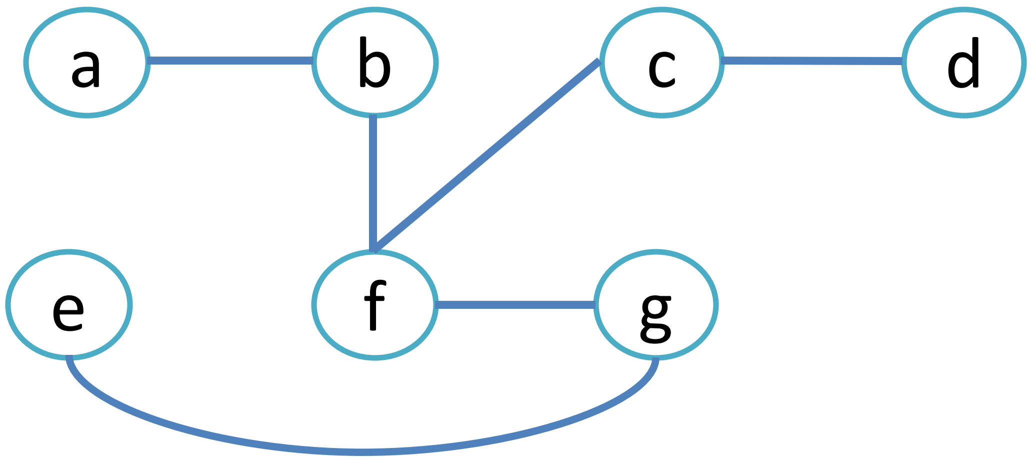
Example 1

Find a spanning tree of the simple graph G.



Example 1

Find a spanning tree of the simple graph *G*.

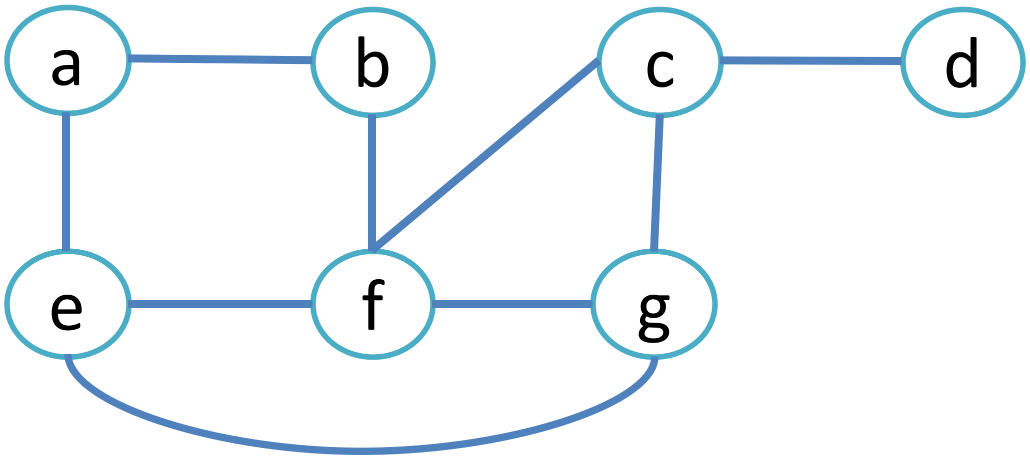


Other spanning trees

The example above is not the only spanning tree of graph G.

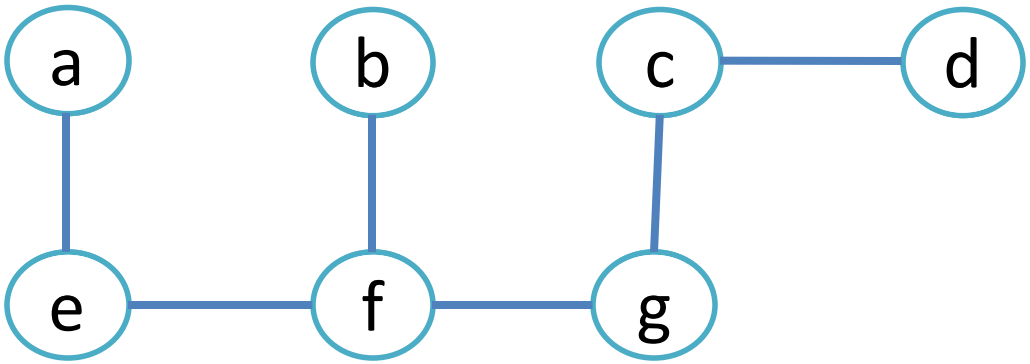
Original

- The example above is not the only spanning tree of graph G.



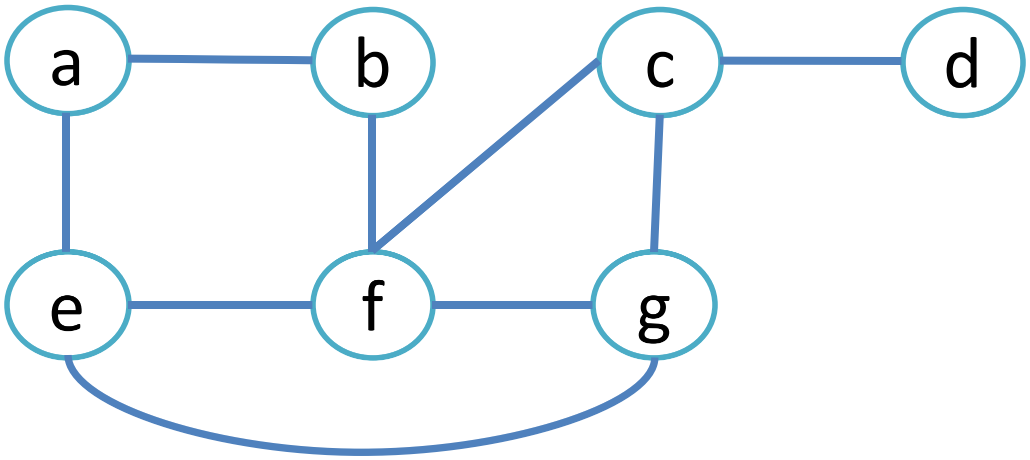
Other spanning trees

- The example above is not the only spanning tree of graph G.



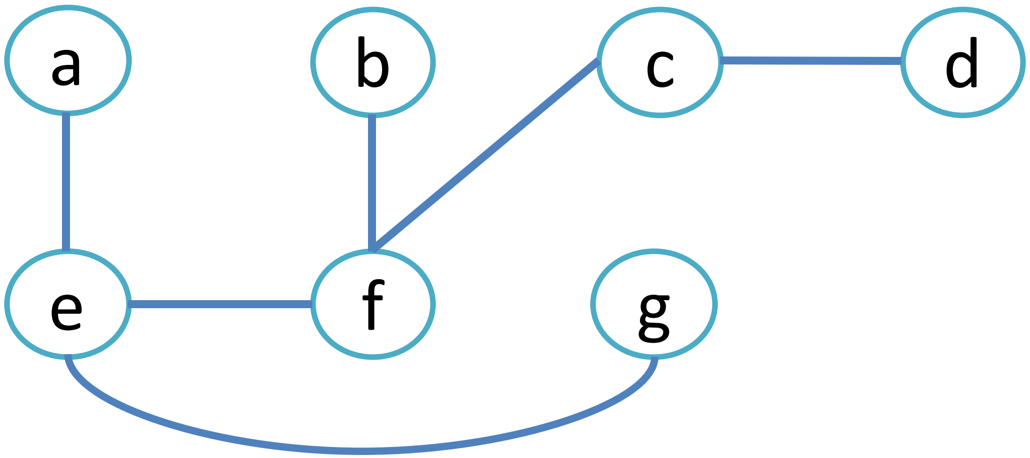
Original

- The example above is not the only spanning tree of graph G.



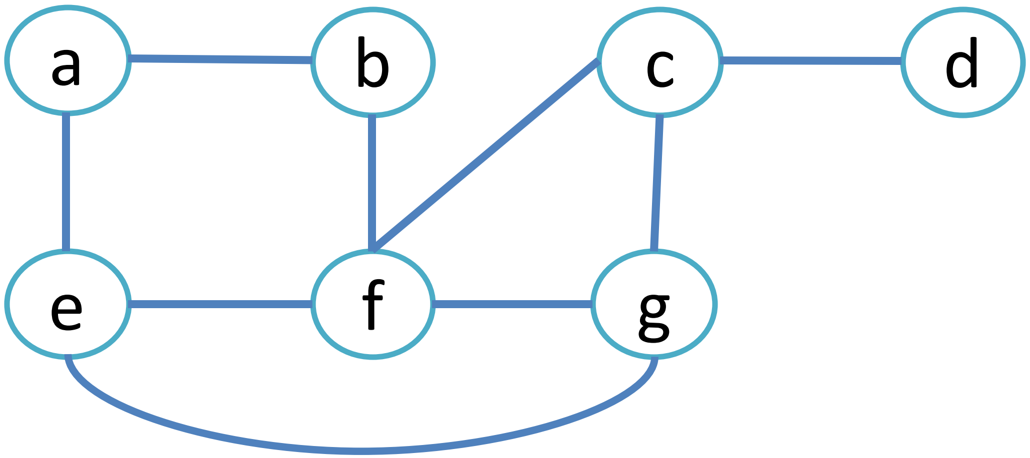
Other spanning trees

- The example above is not the only spanning tree of graph G.



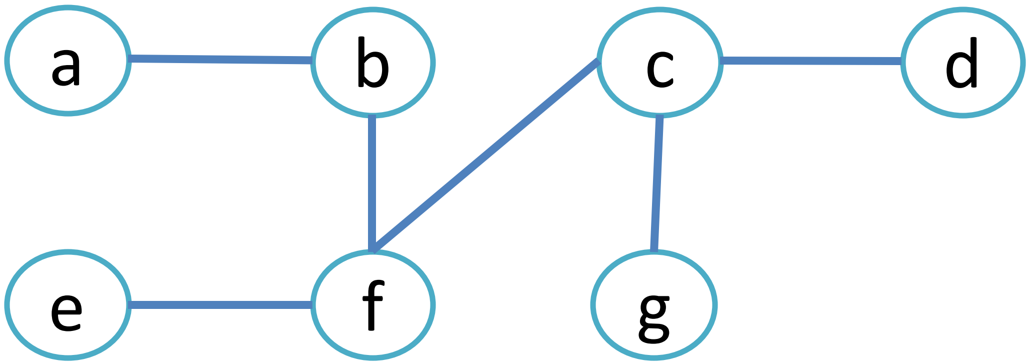
Original

- The example above is not the only spanning tree of graph G.



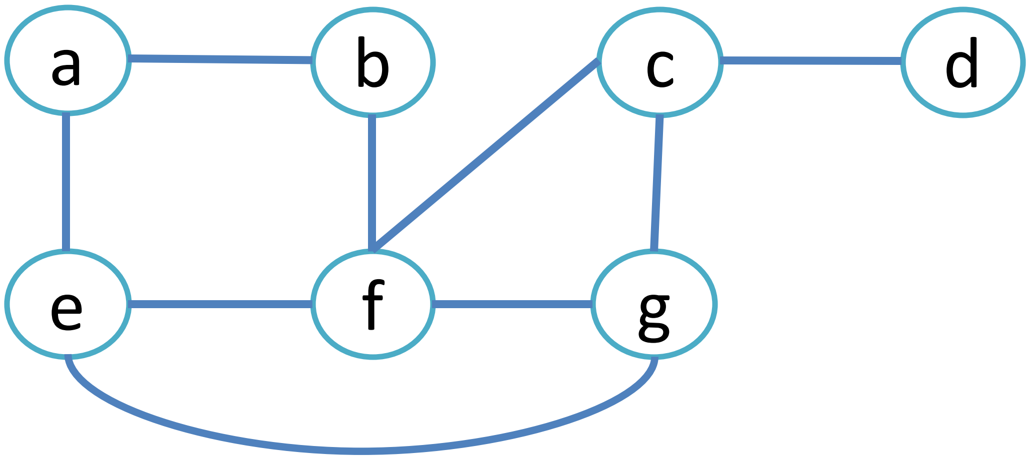
Other spanning trees

- The example above is not the only spanning tree of graph G.



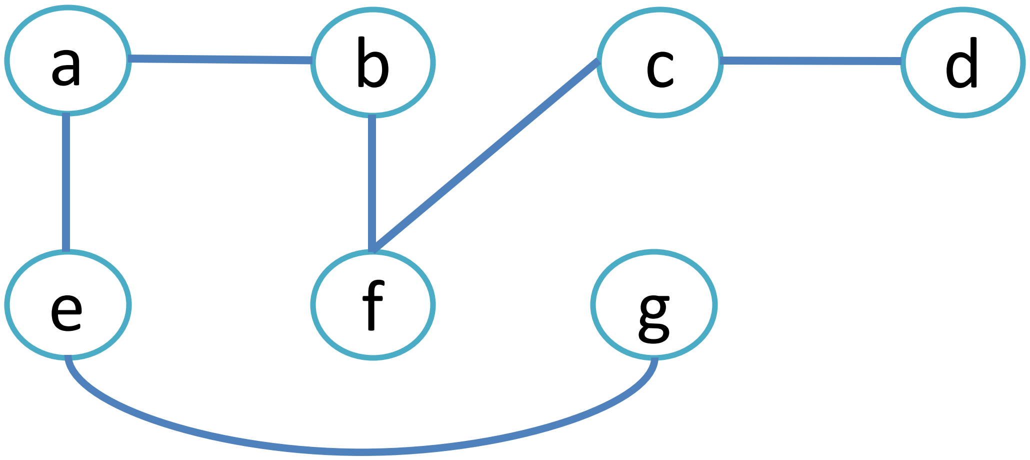
Original

- The example above is not the only spanning tree of graph G.



Other spanning trees

- The example above is not the only spanning tree of graph G.



Theorem 1

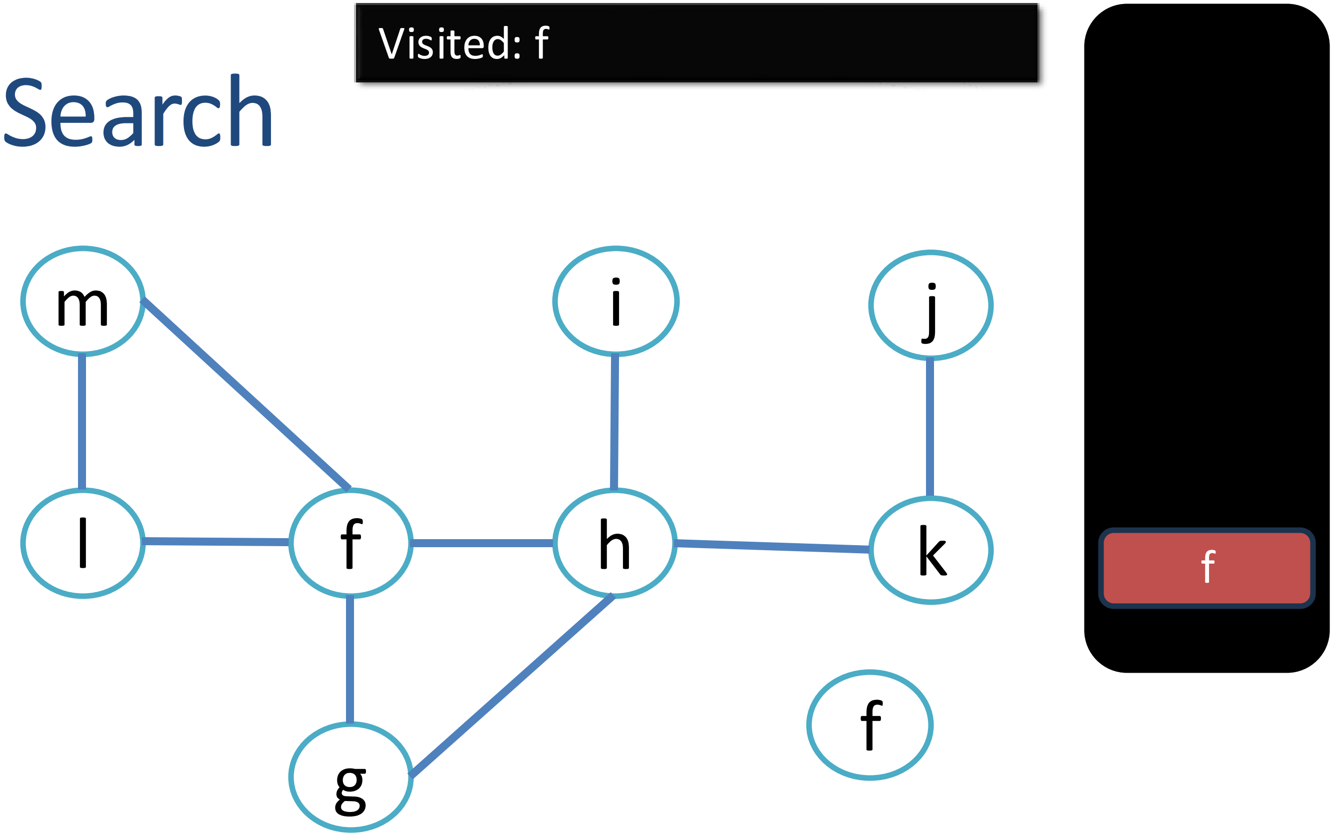
A simple graph is connected if and only if it has a spanning tree

```
DFS-iterative (G, s): //Where G is graph and s is source vertex
        let S be stack
        S.push( s ) //Inserting s in stack
        mark s as visited.
        while (S is not empty): //Pop a vertex from stack to visit
                next v = S.top( )
                S.pop( )
                //Push all the neighbours of v in stack that are not visited
                for all neighbours w of v in Graph G:
                        if w is not visited : S.push( w )
                                mark w as visited
```

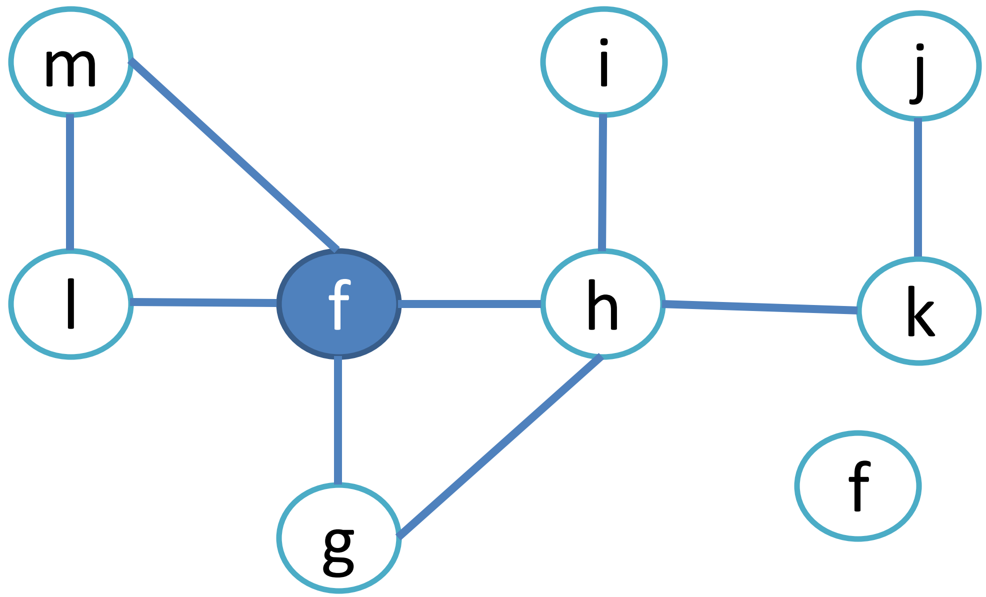
```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

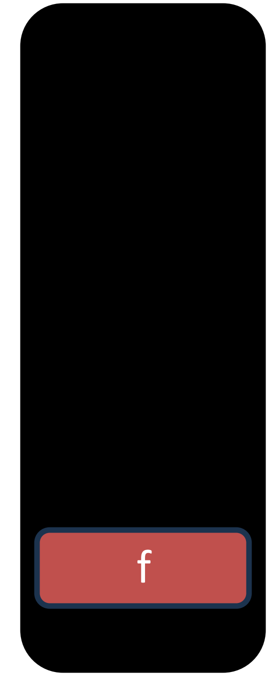
Depth-first Search

Visited: f



Depth-first Search



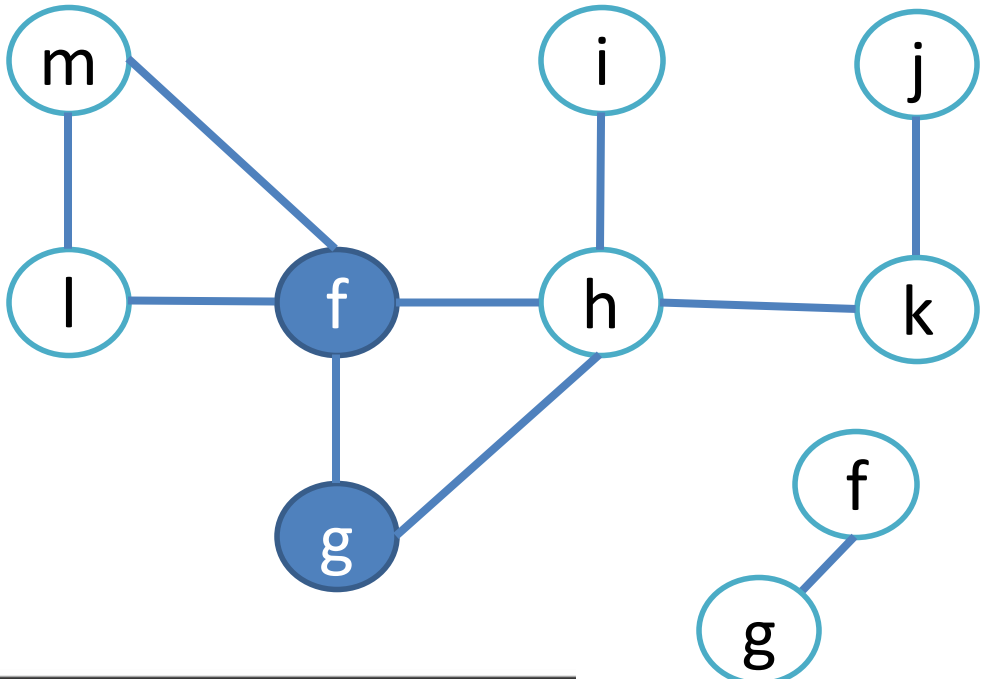


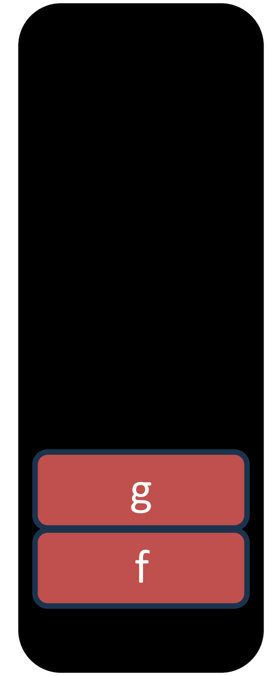
visit(f)

Procedure *visit*(v : vertex of G)

for each vertex w adjacent to v and not yet in T **do**
    add vertex w and edge $\{v, w\}$ to T
    visit(w)
end for

Depth-first Search



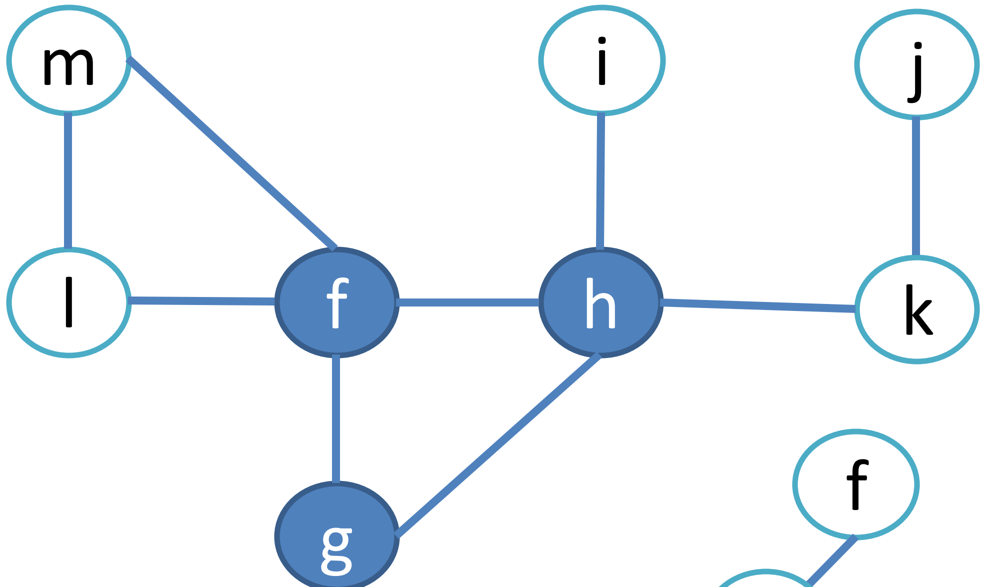


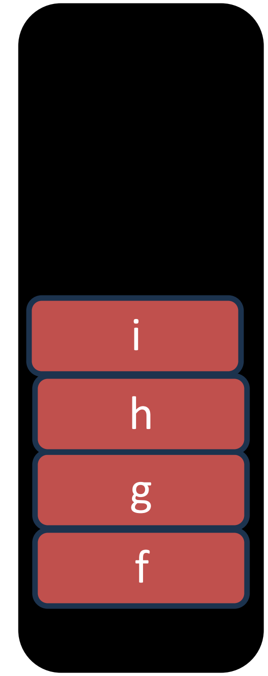
Procedure *visit*(v : vertex of G)

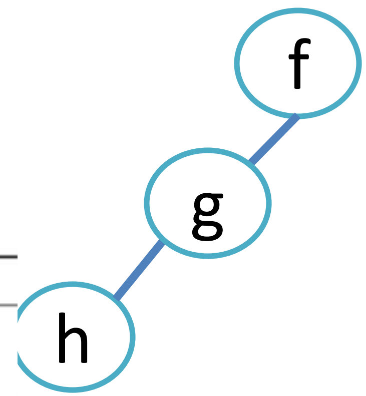
```
for each vertex w adjacent to v and not yet in T do
    add vertex w and edge {v, w} to T
    visit(w)
end for
```

add vertex g and edge (f, g) to T
visit(g)

Depth-first Search





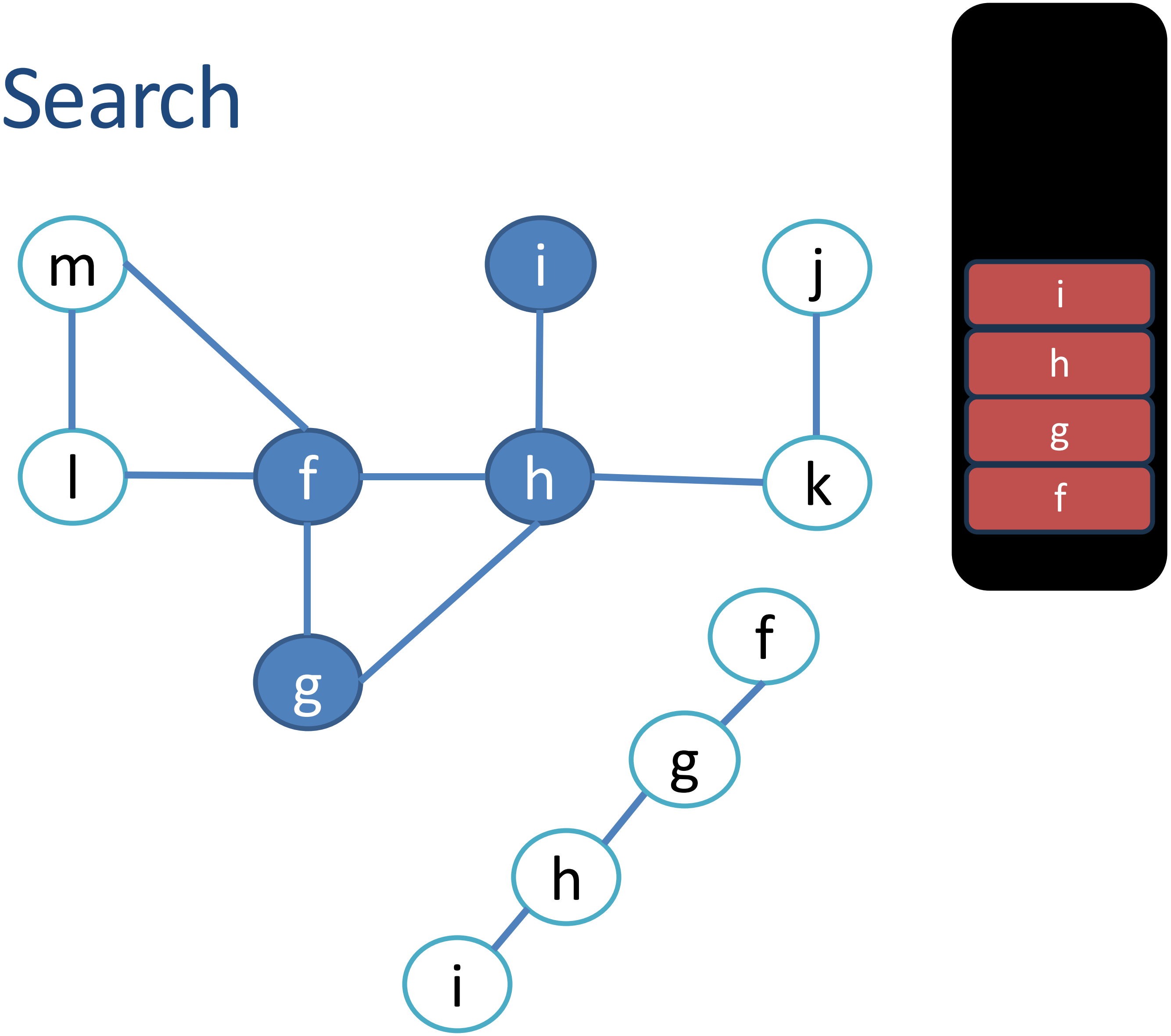


Procedure *visit*(v : vertex of G)

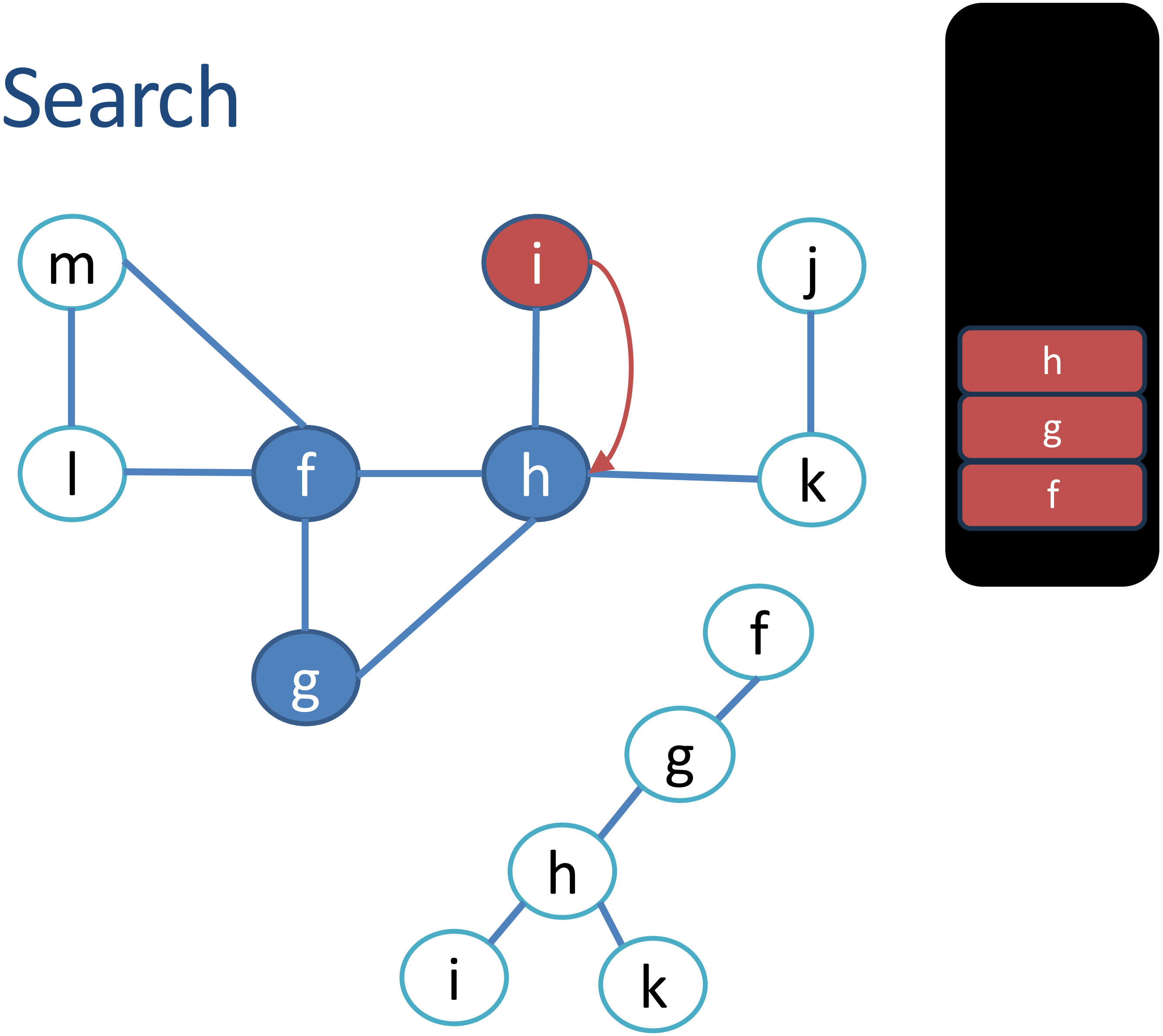
for each vertex w adjacent to v and not yet in T **do**
    add vertex w and edge $\{v, w\}$ to T
    visit(w)
end for

add vertex h and edge (g, h) to T
visit(h)

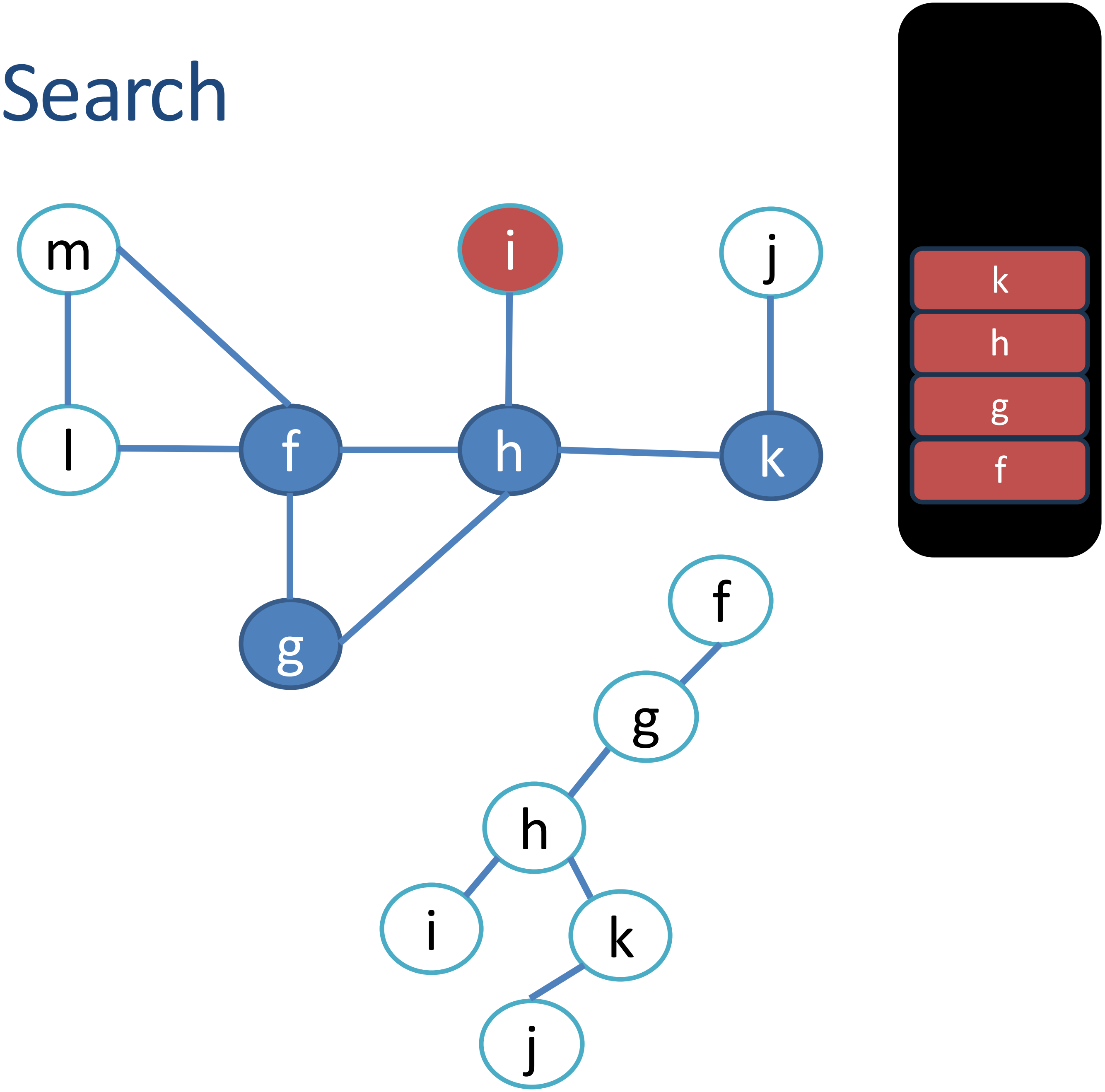
Depth-first Search



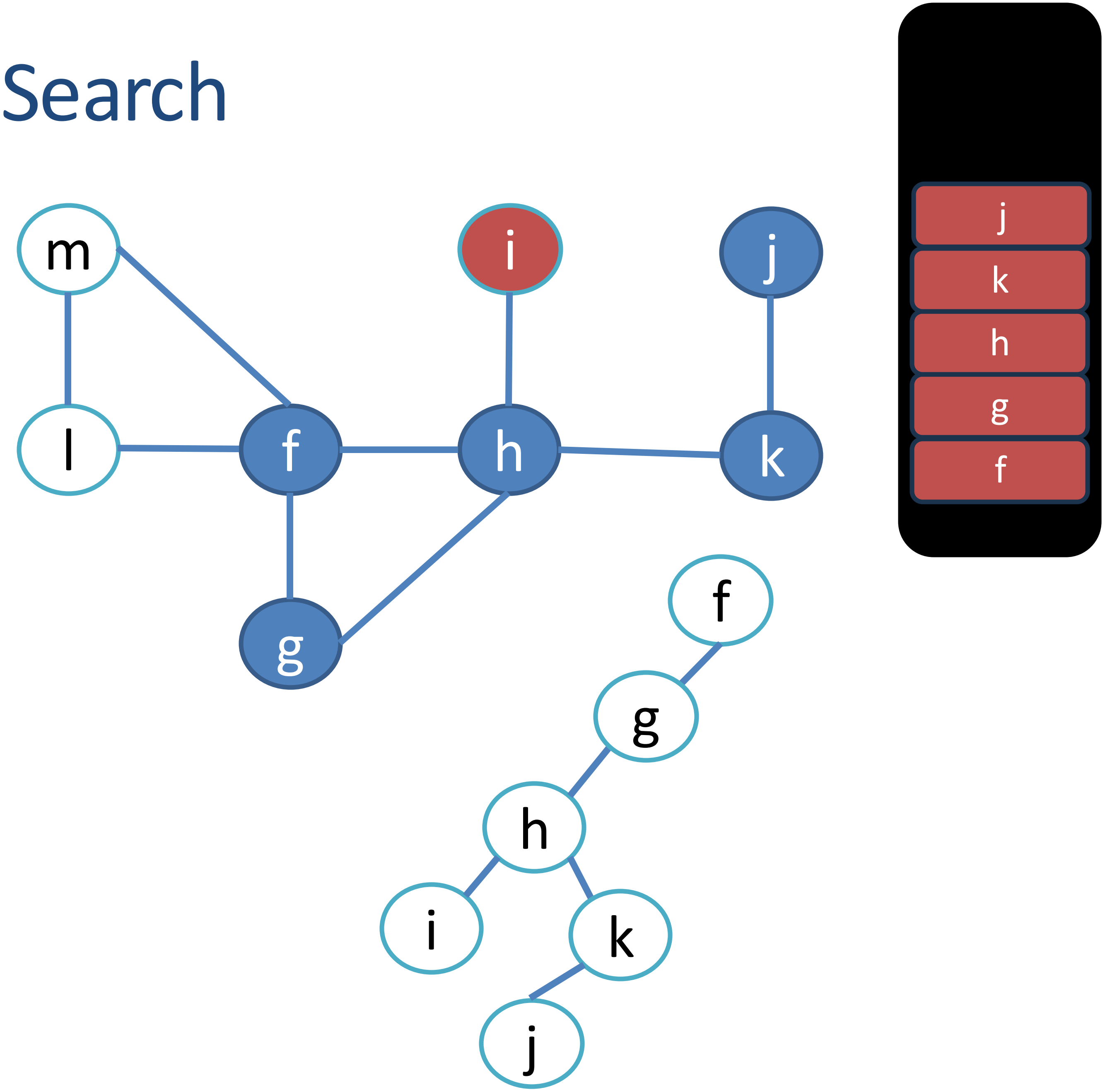
Depth-first Search



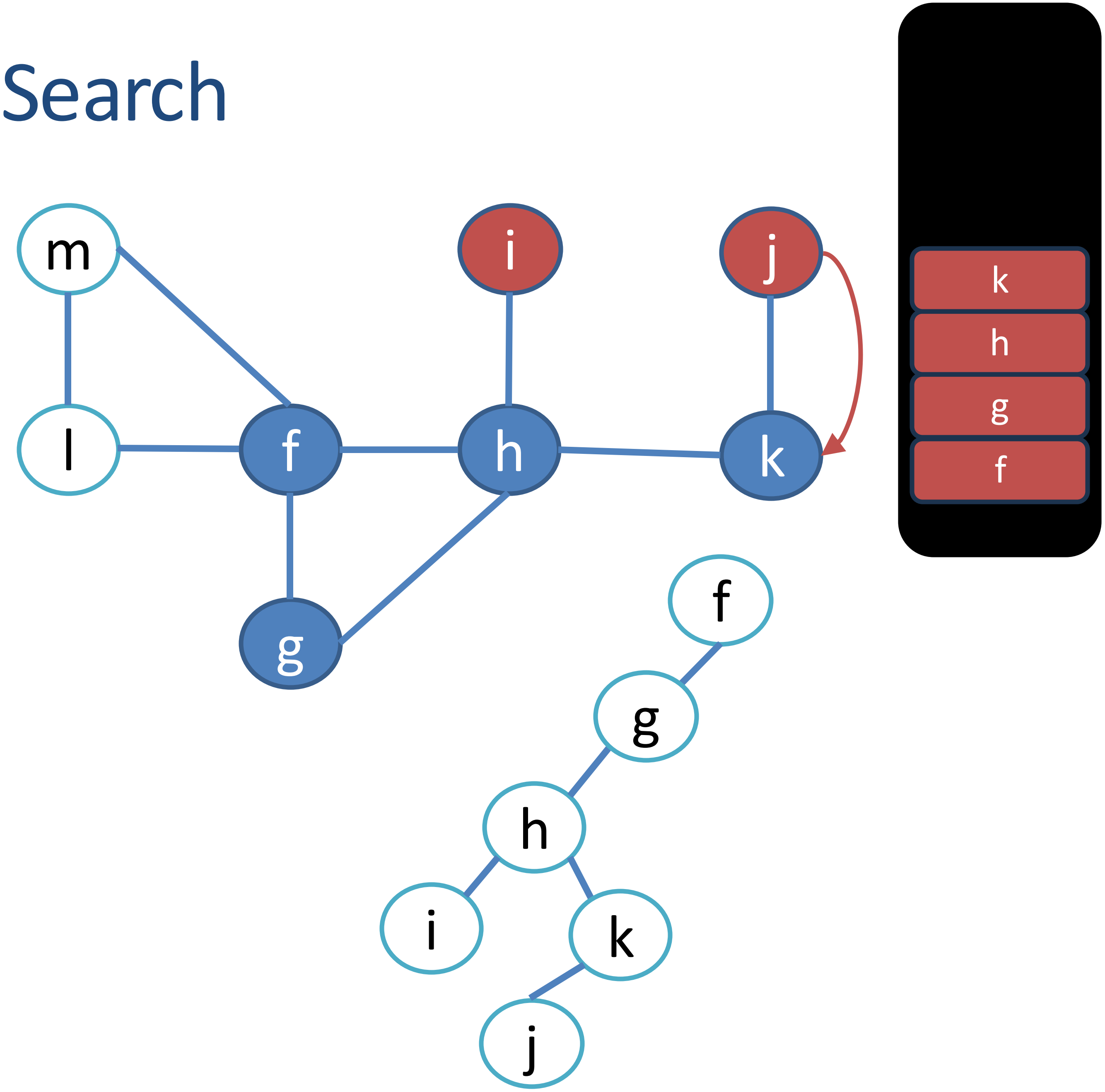
Depth-first Search



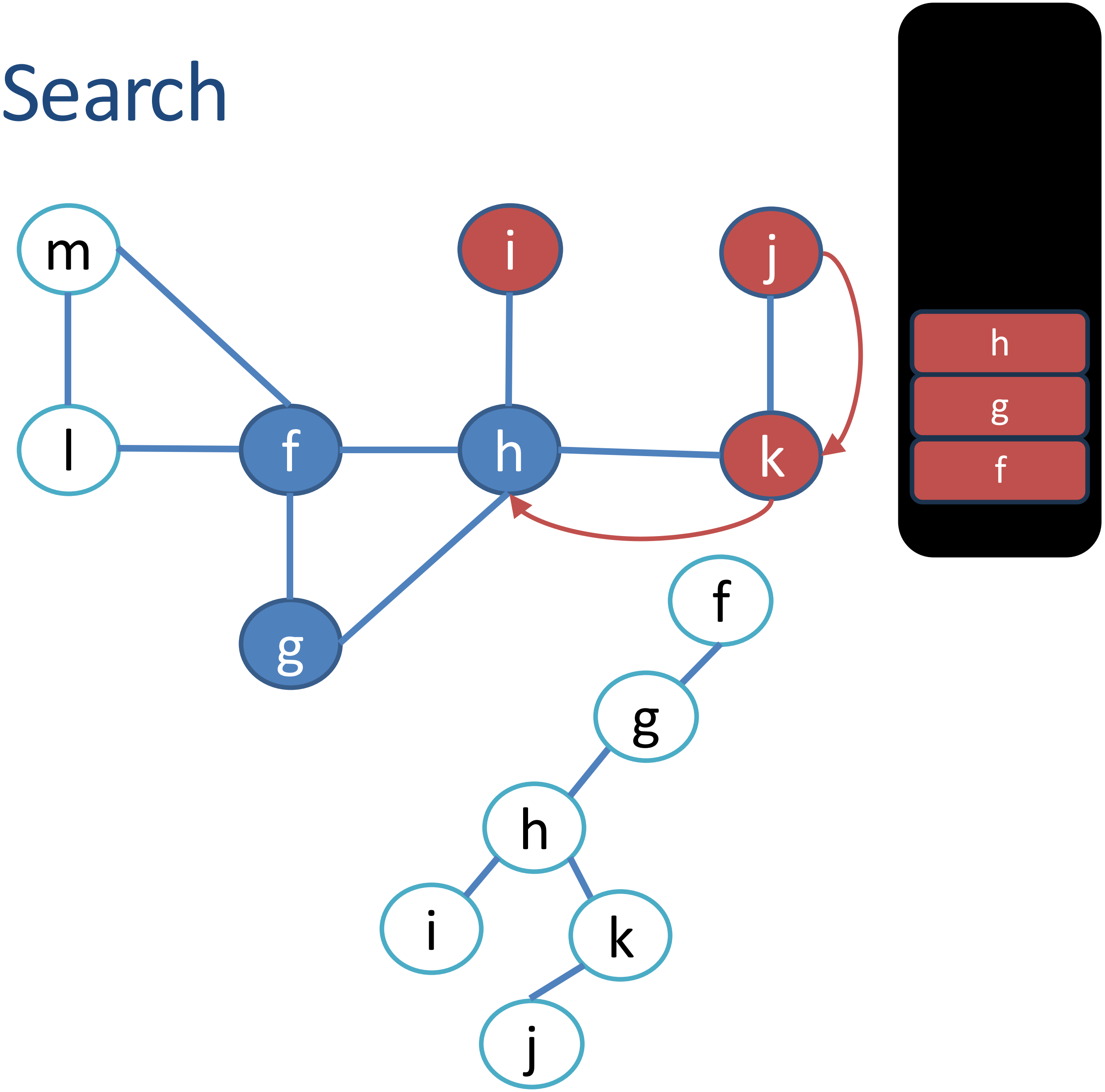
Depth-first Search



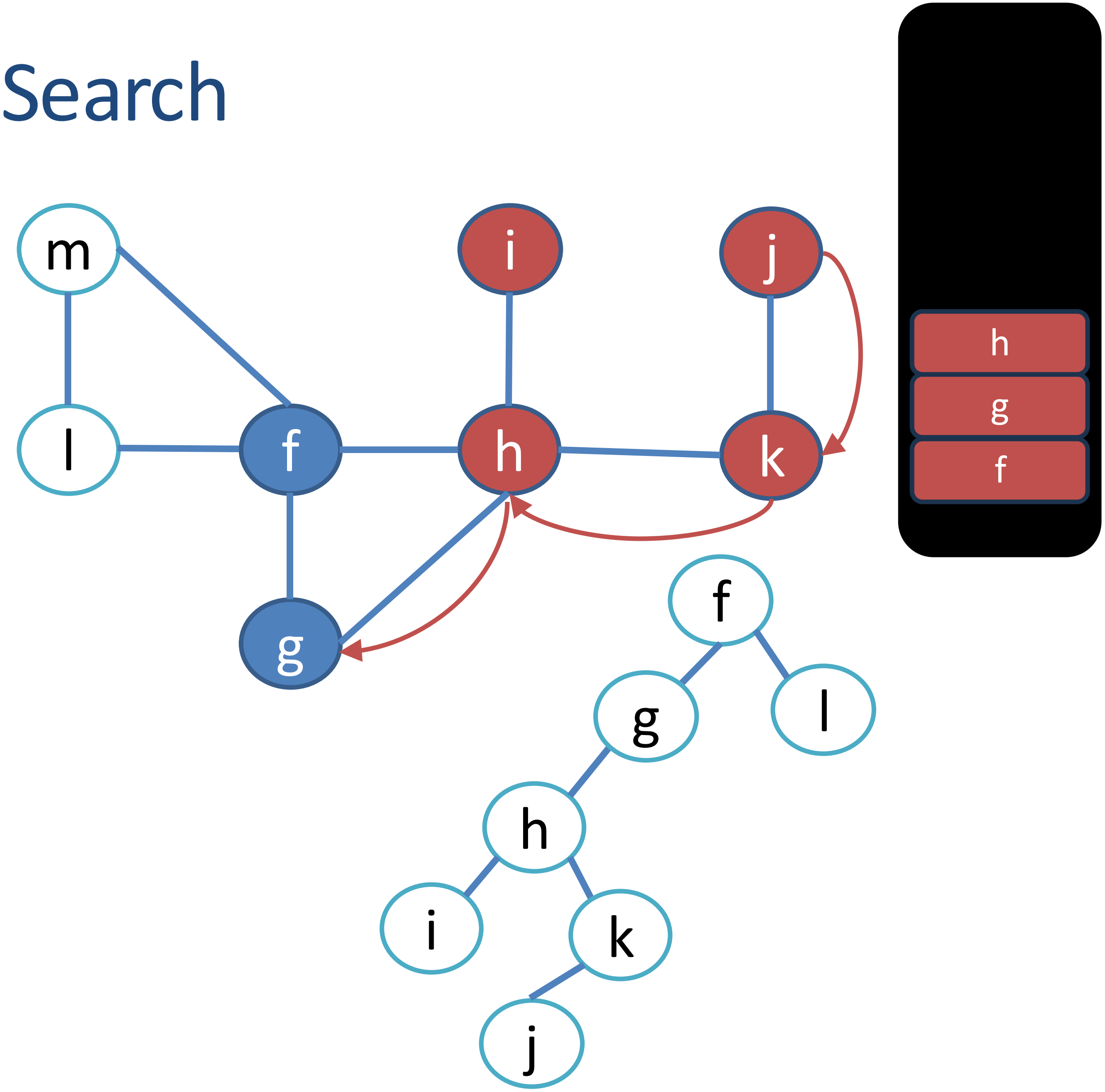
Depth-first Search



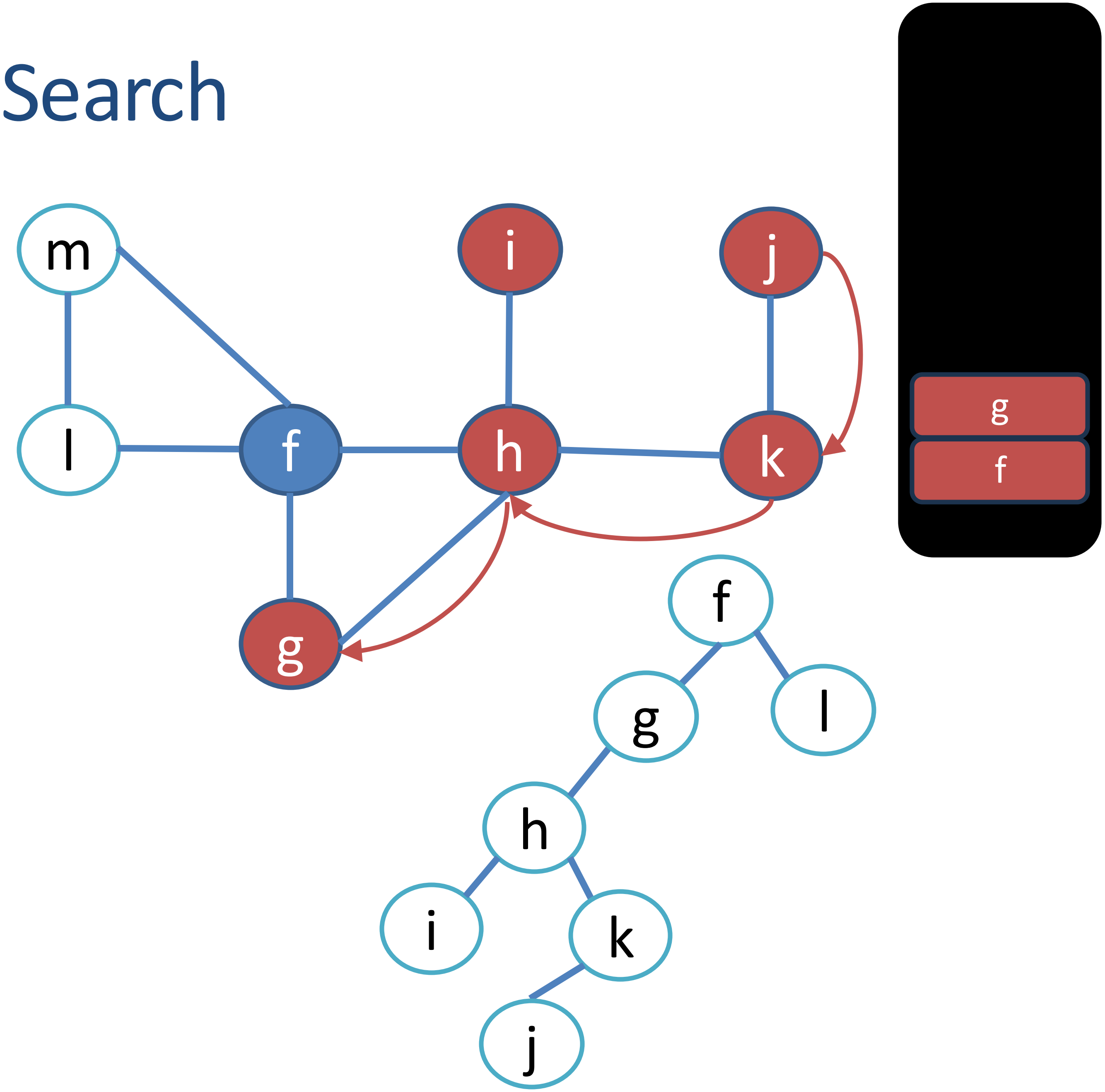
Depth-first Search



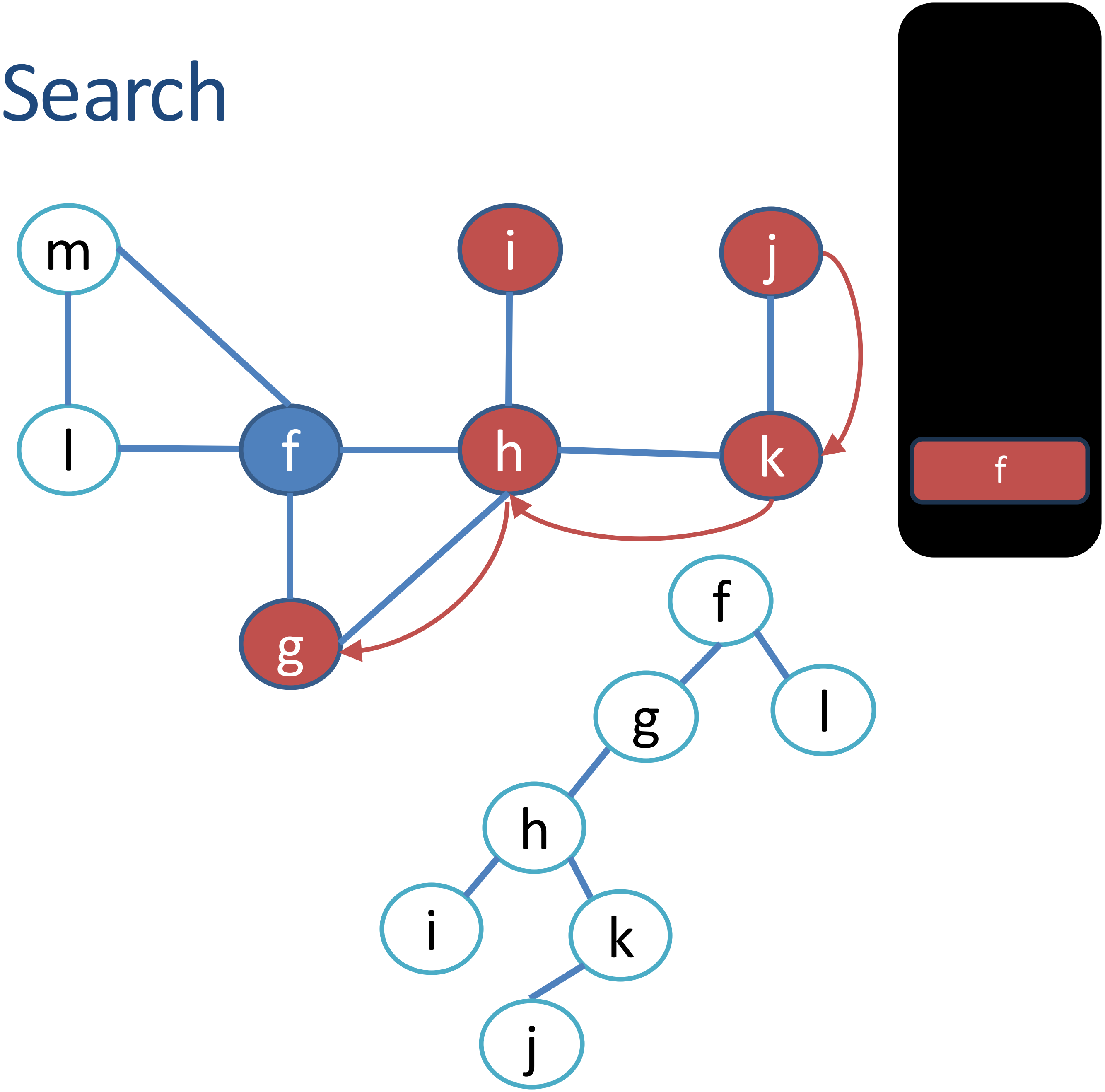
Depth-first Search



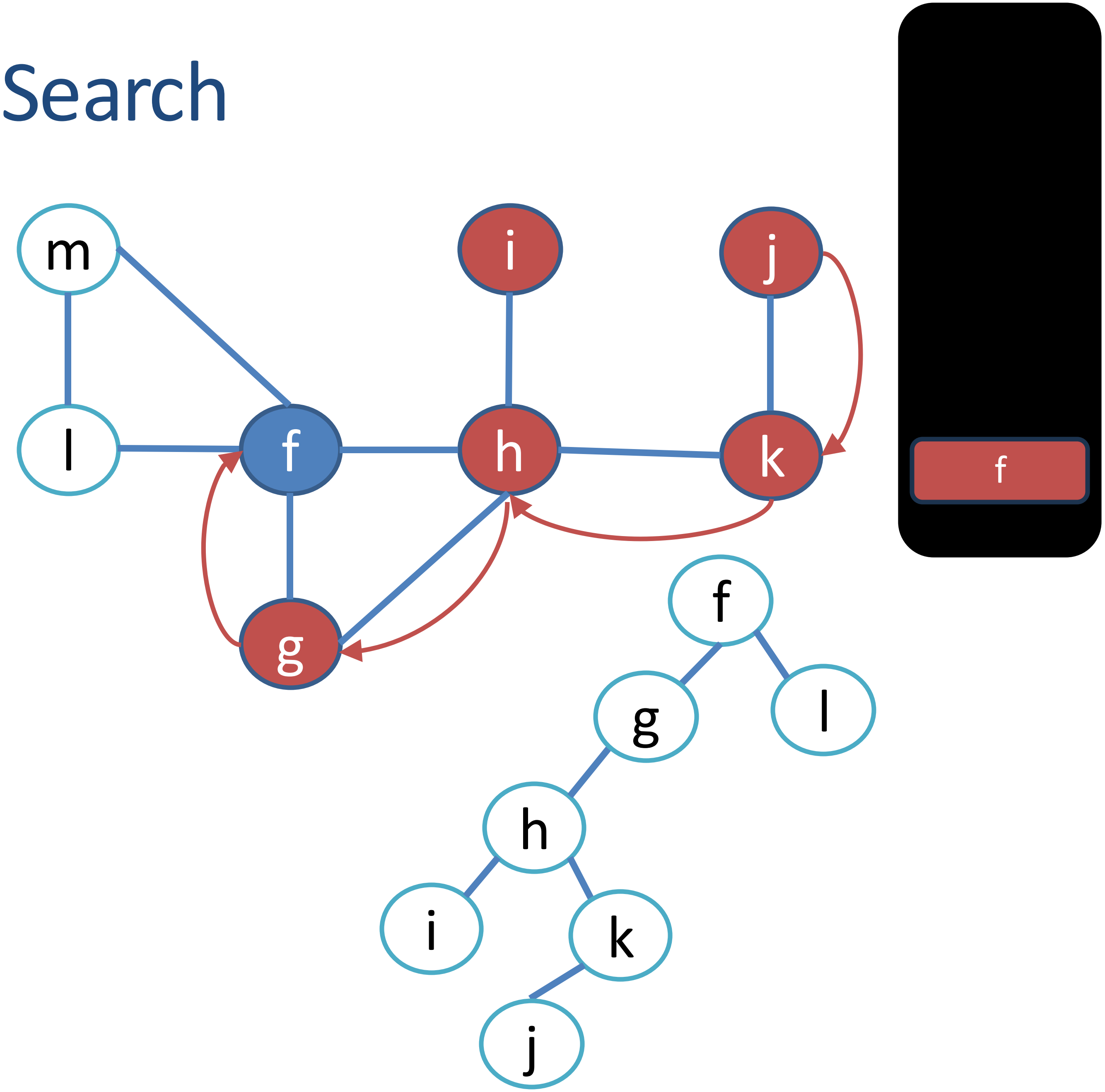
Depth-first Search



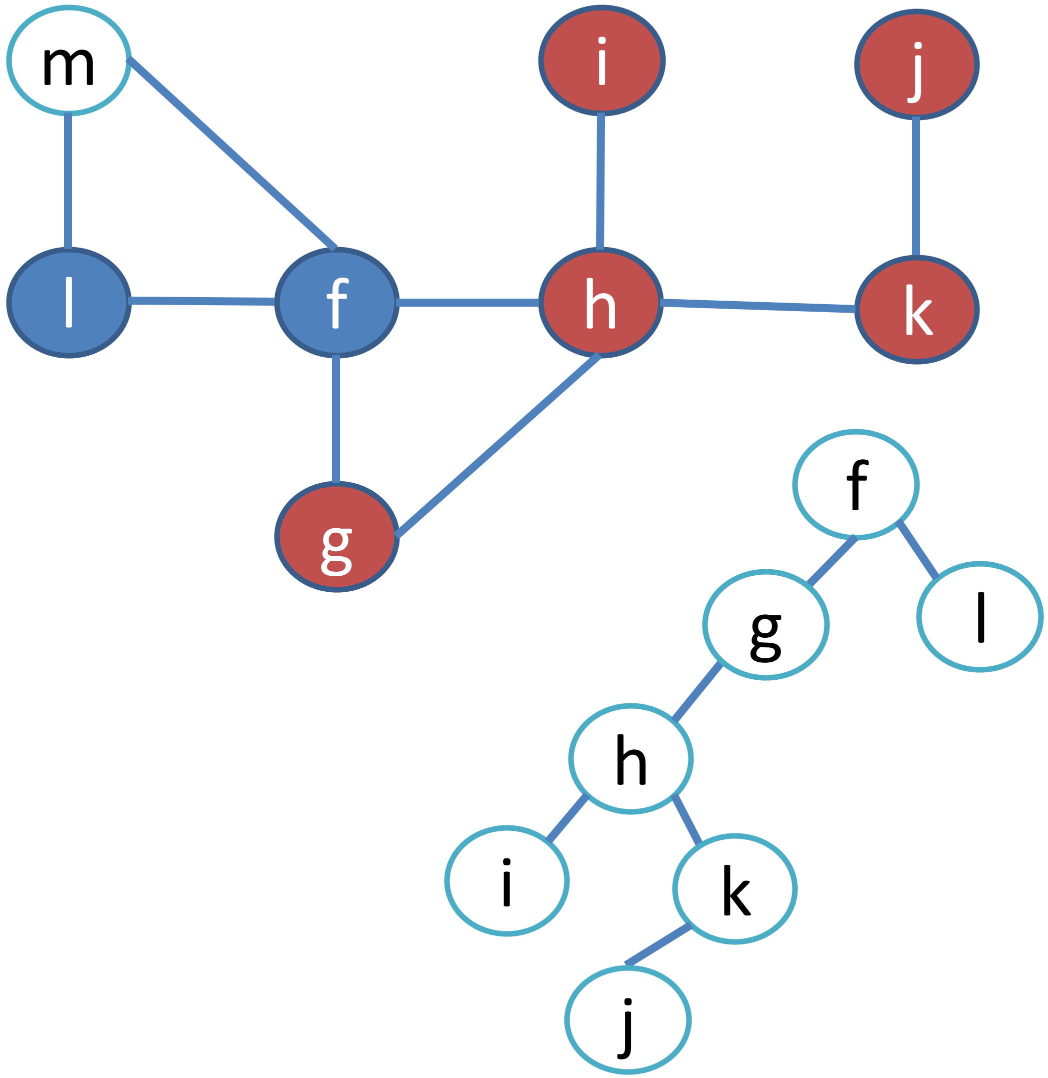
Depth-first Search

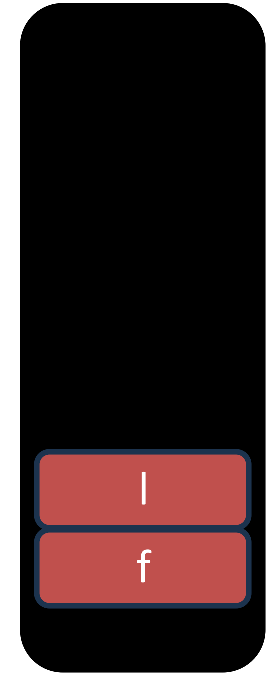


Depth-first Search

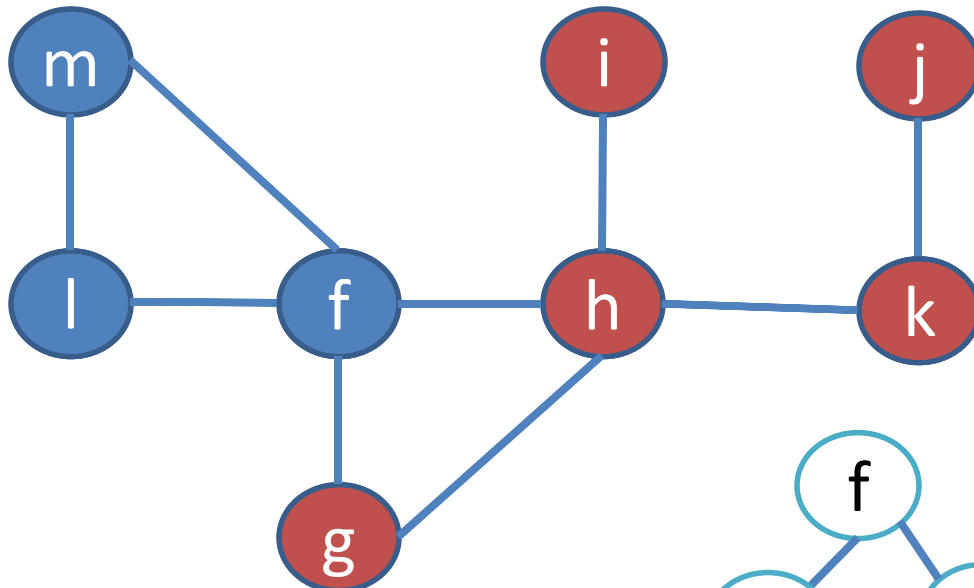


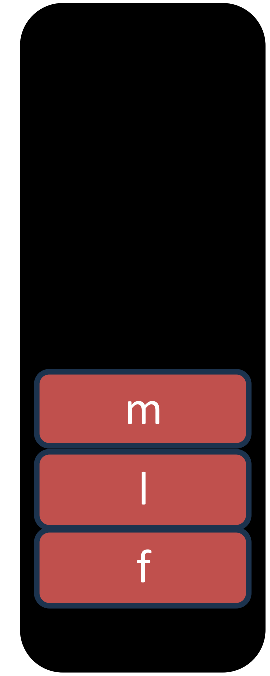
Depth-first Search

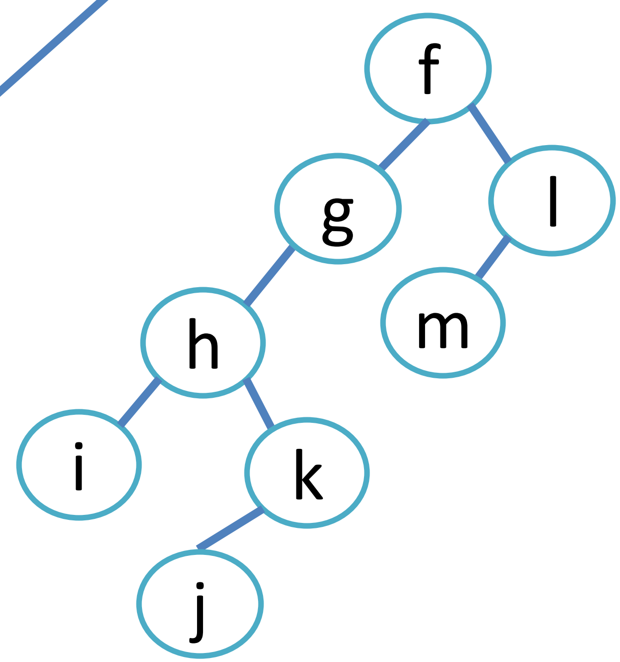




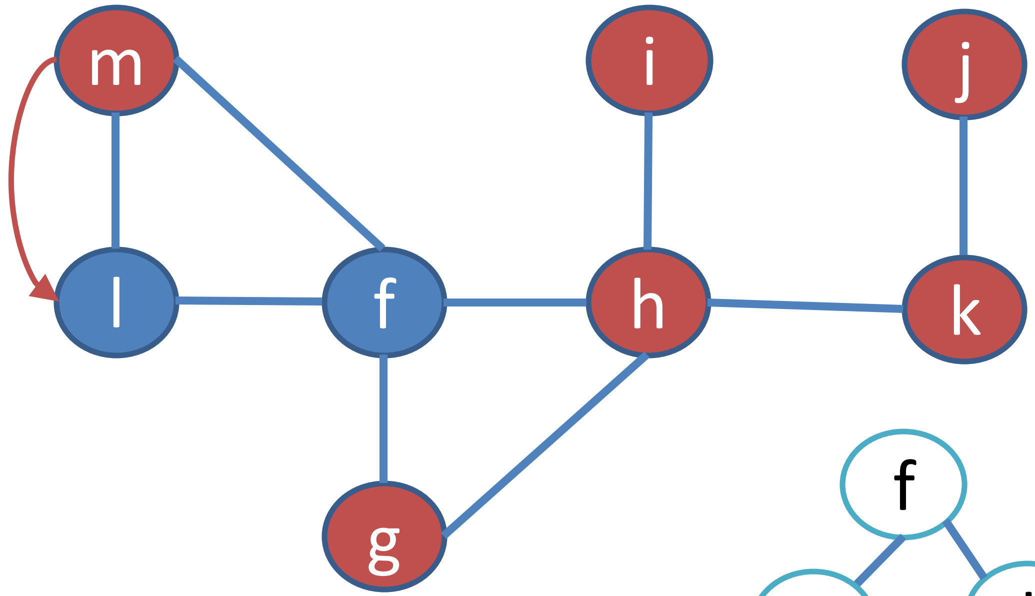
Depth-first Search

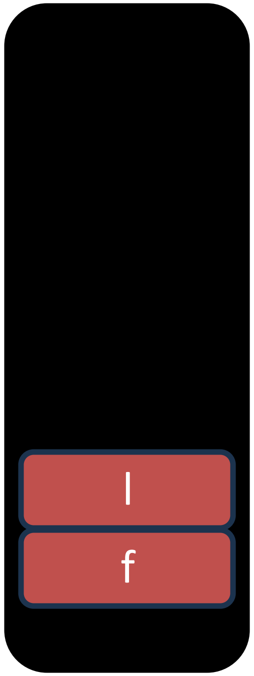


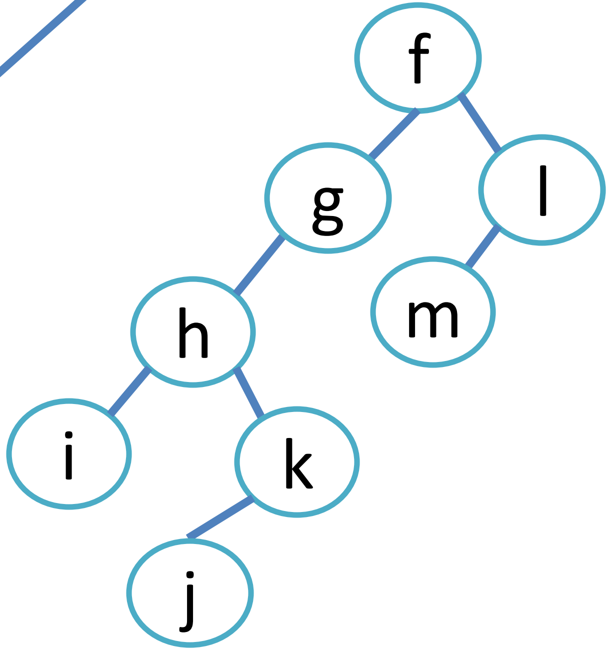




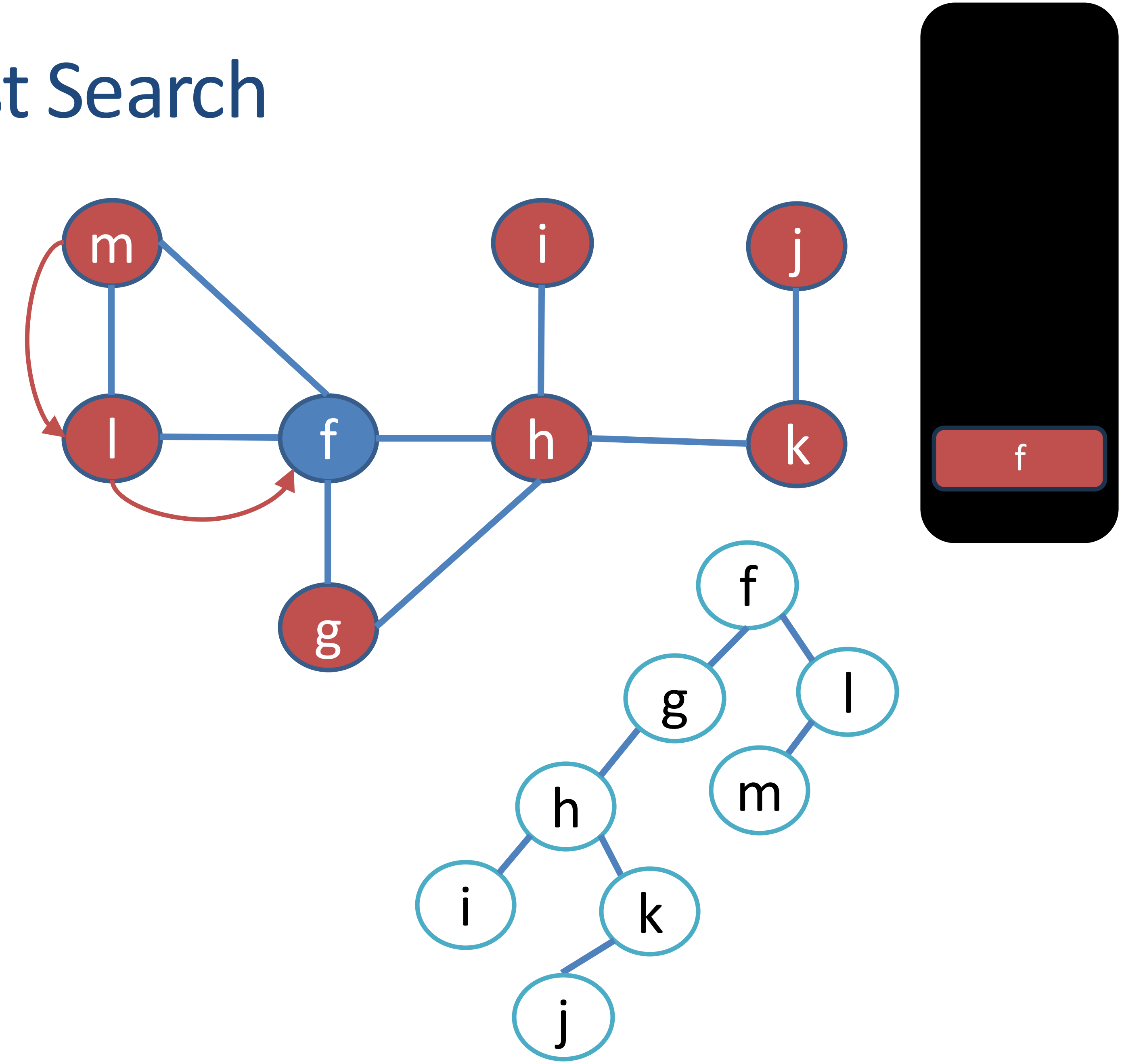
Depth-first Search



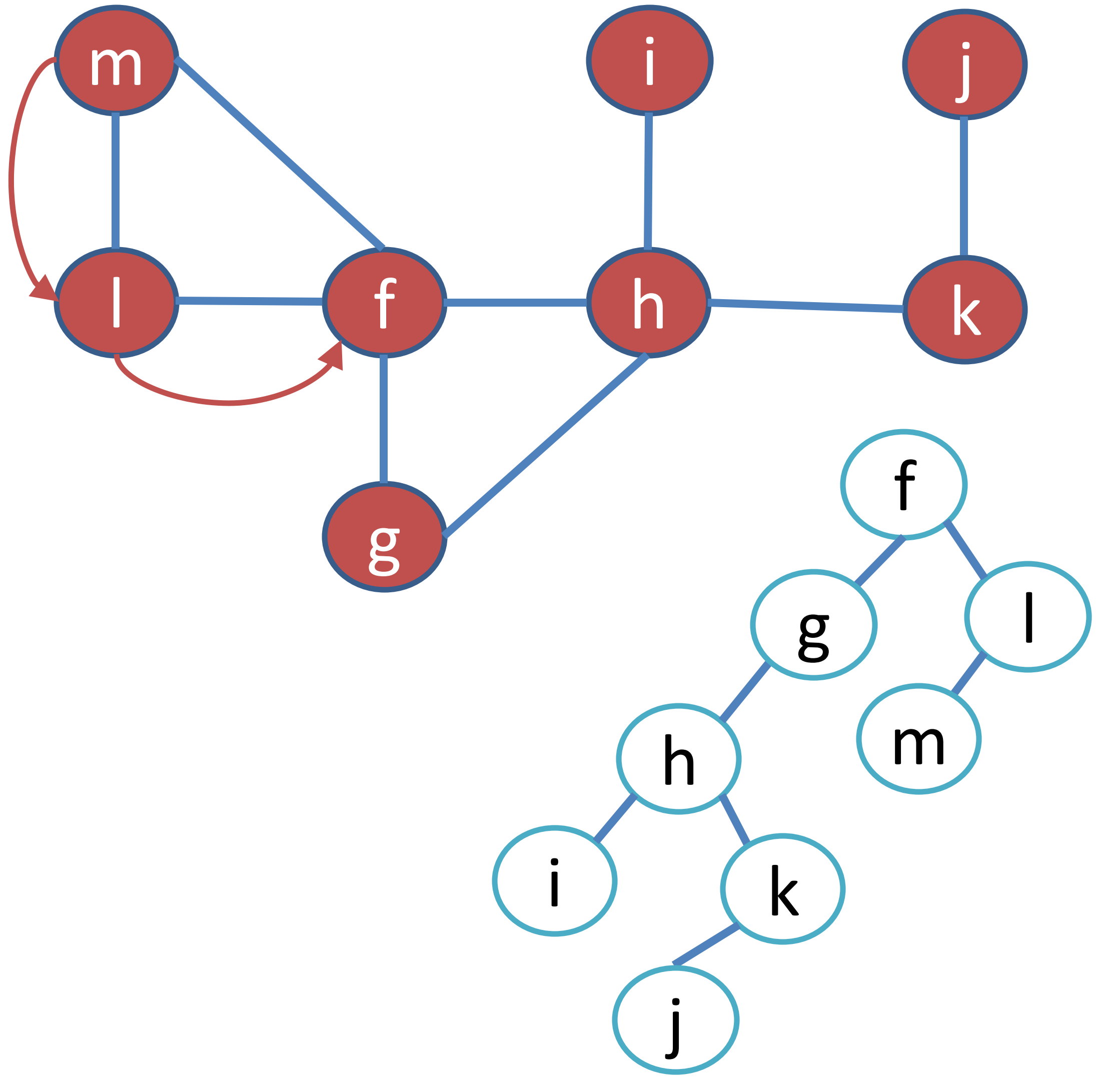




Depth-first Search



Depth-first Search



[redacted]

Anaysis

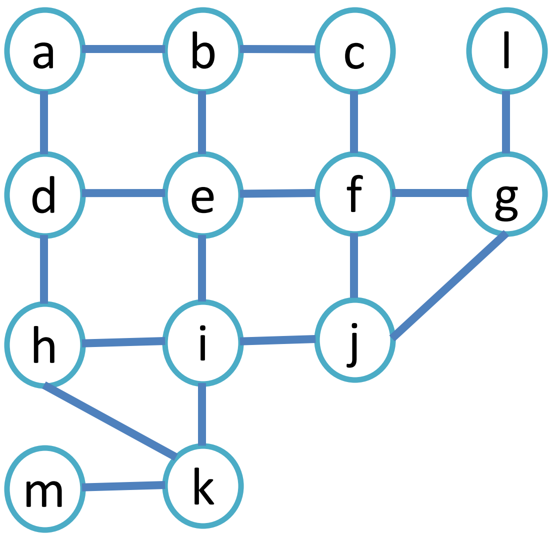
- The time complexity of Depth First Search (DFS) is O(V + E), where V is the number of vertices and E is the number of edges.
- DFS visits each vertex and explores each edge **once** (in an adjacency list representation).
- **Time Complexity:**
- You visit each node once: O(V)
- For each node, you look at its neighbors (edges): O(E)
- **Total:** O(V+E)

Breadth-first search

```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
    while Q is not empty:
        v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

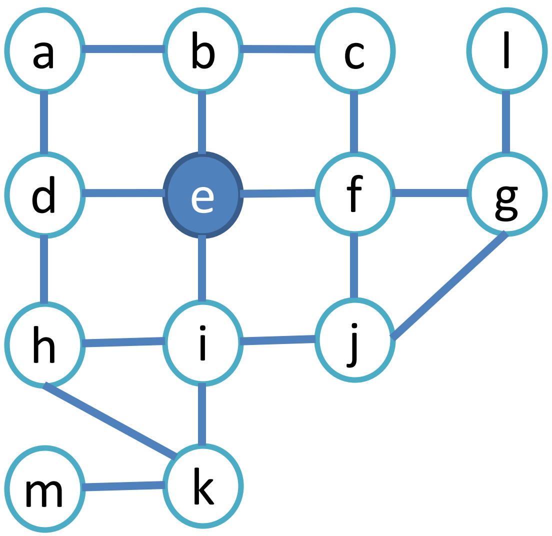
Breadth-first Search

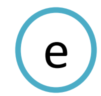
[redacted]

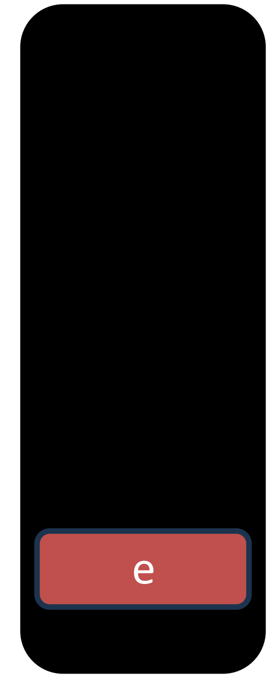



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                        Q.enqueue(w)
```

Breadth-first Search





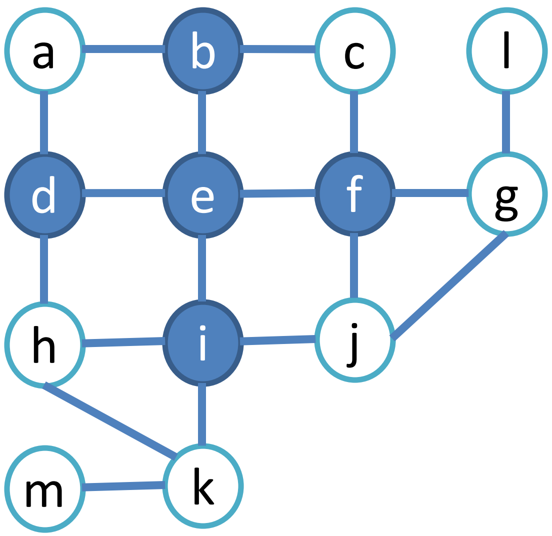


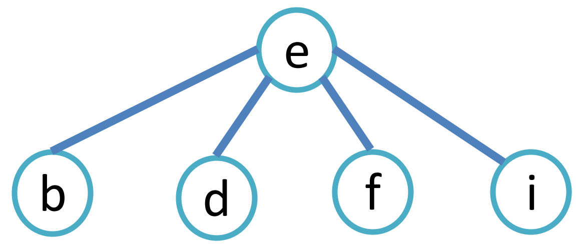

```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

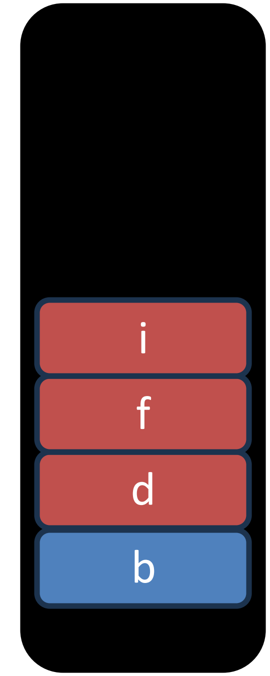
Q = {e}

L = {}

Breadth-first Search

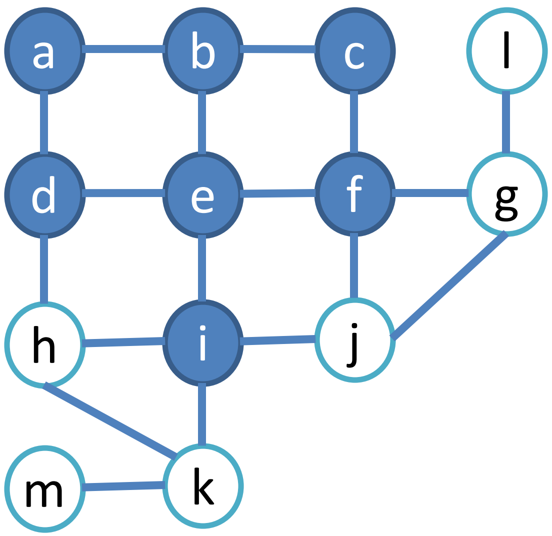


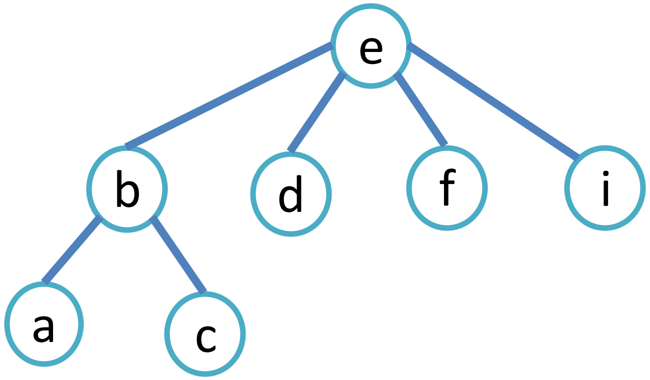


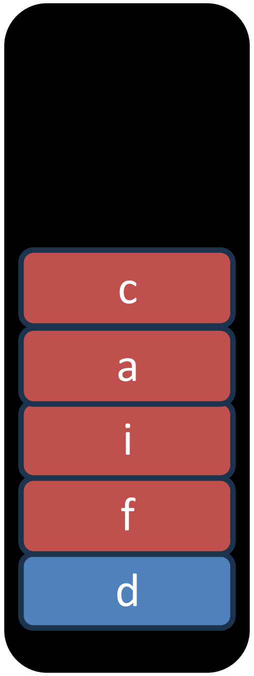



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

Breadth-first Search

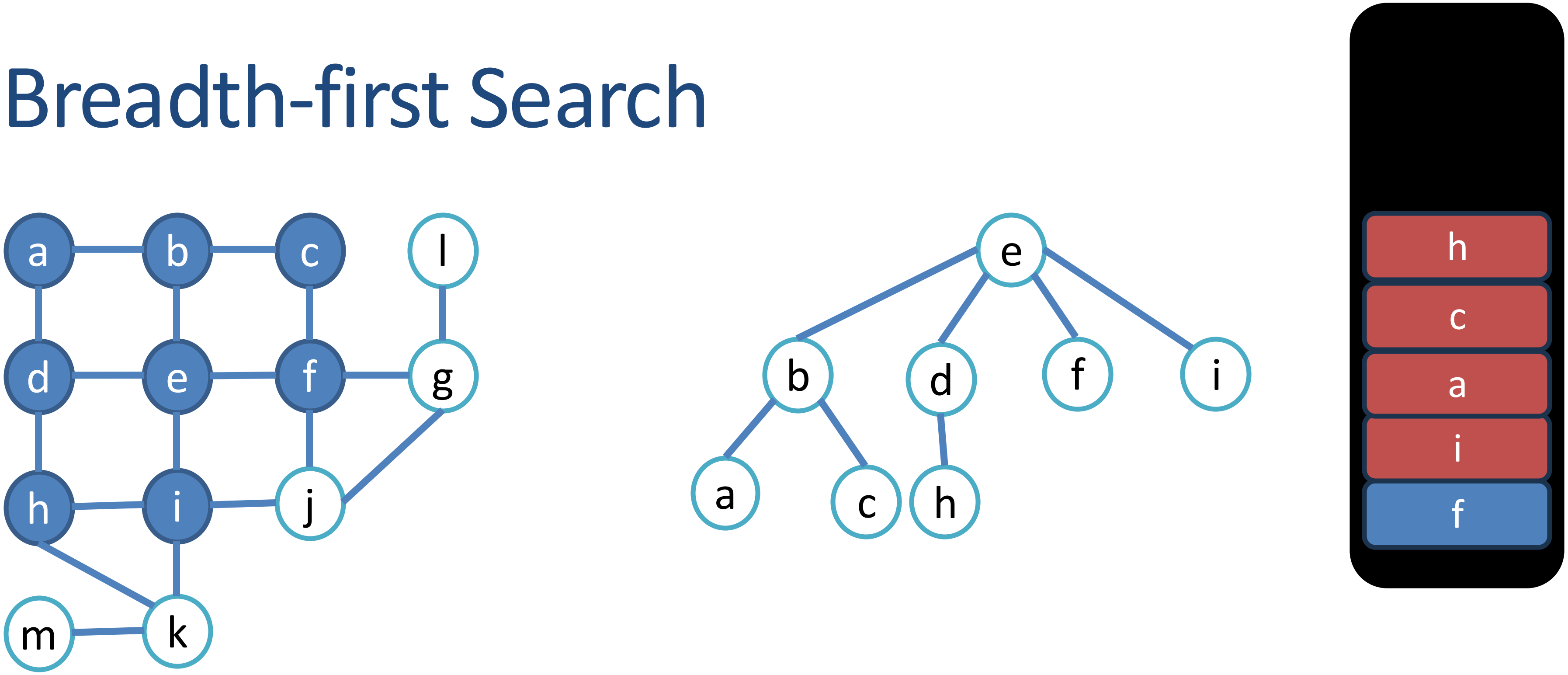






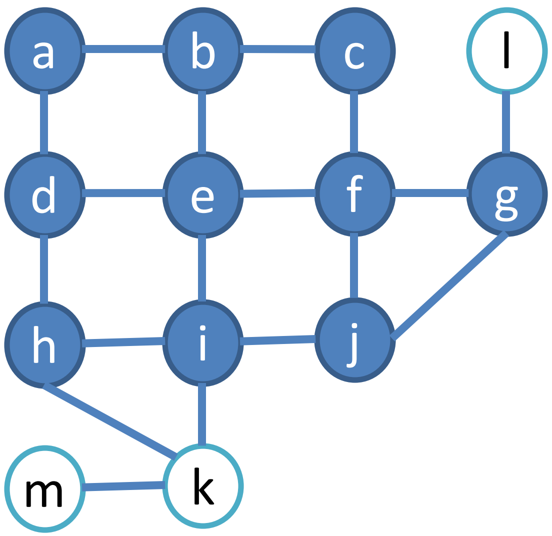

```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

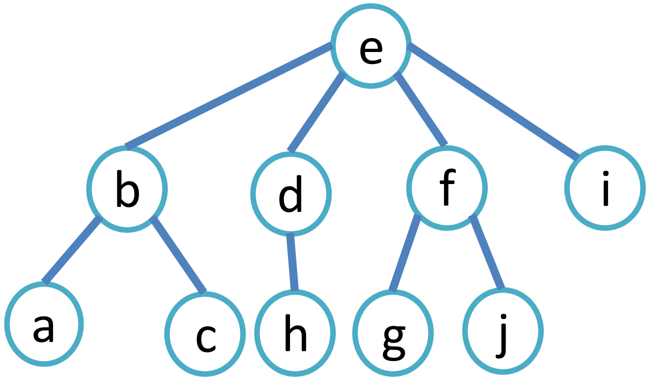
Breadth-first Search

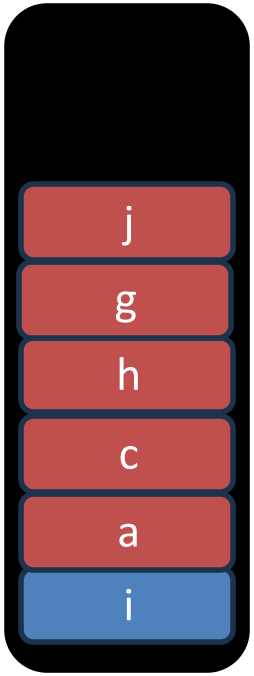



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

Breadth-first Search

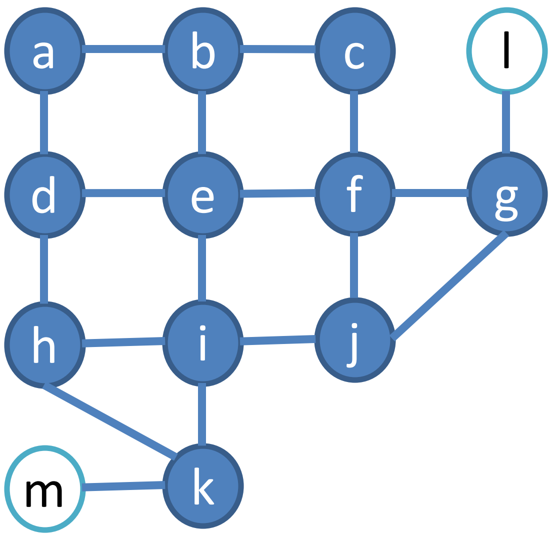


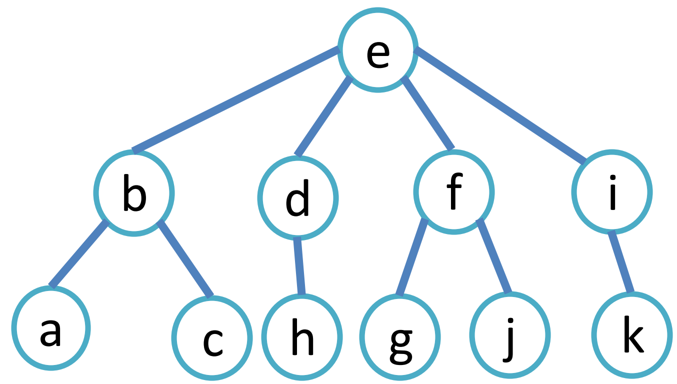


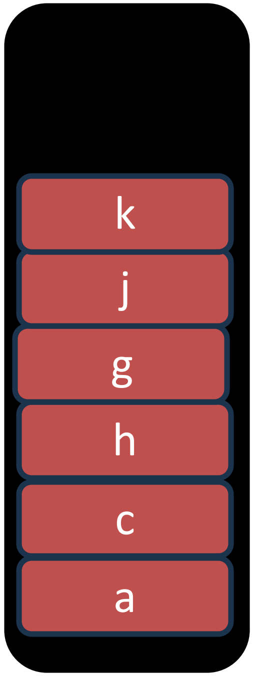



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                        Q.enqueue(w)
```

Breadth-first Search

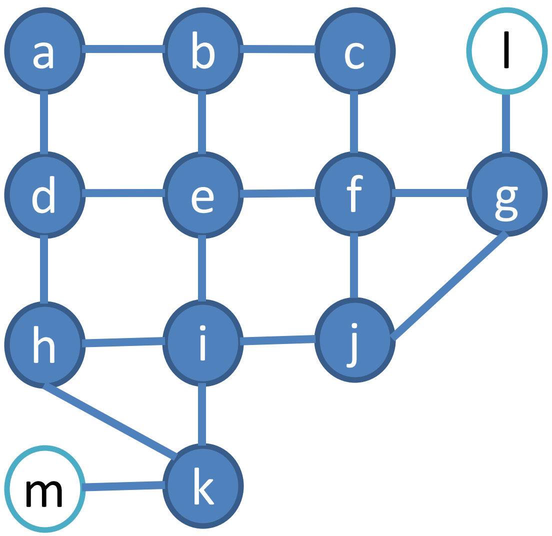


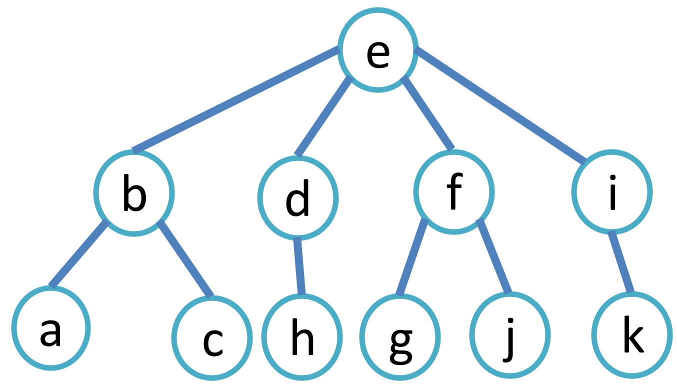


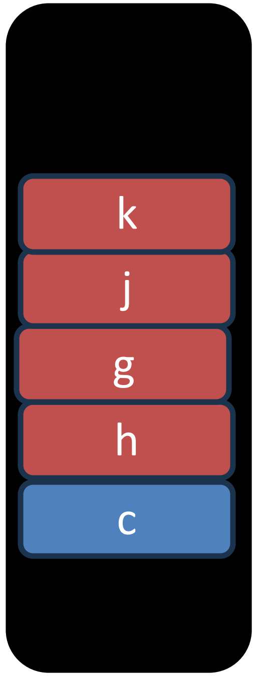



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

Breadth-first Search



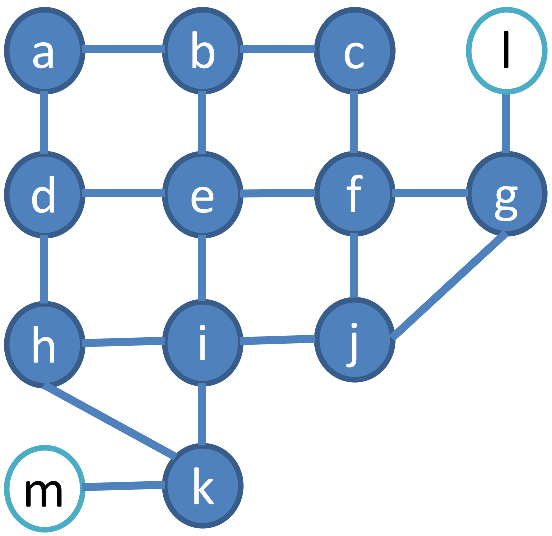


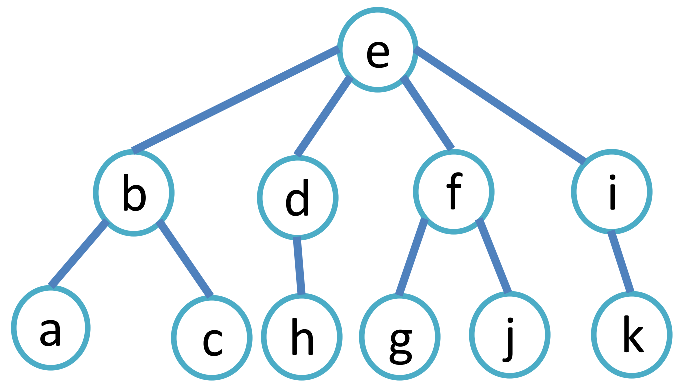


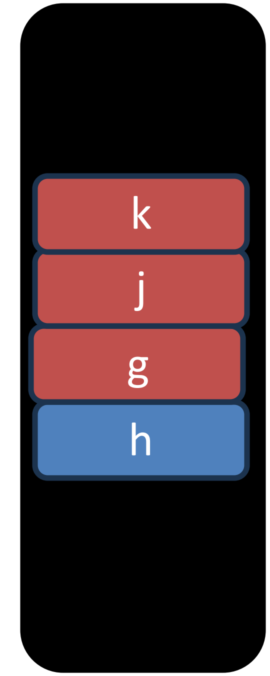
BFS (graph G, start vertex s)

    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)

Breadth-first Search

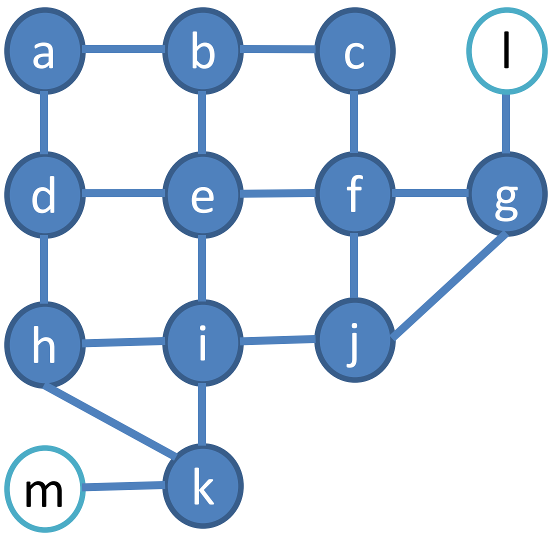


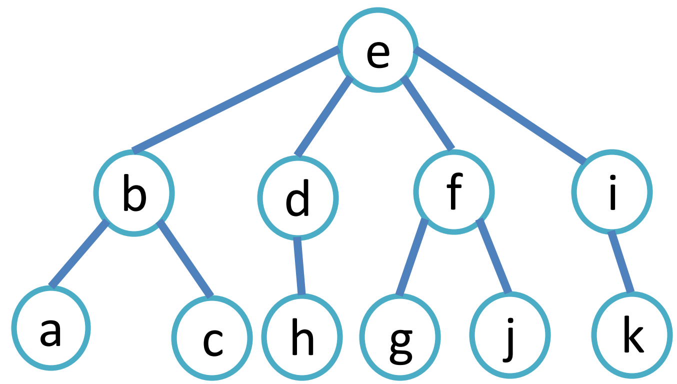


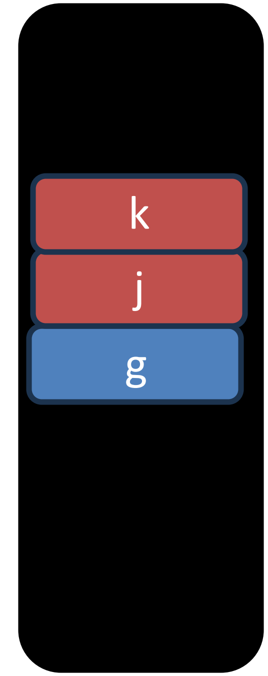



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

Breadth-first Search

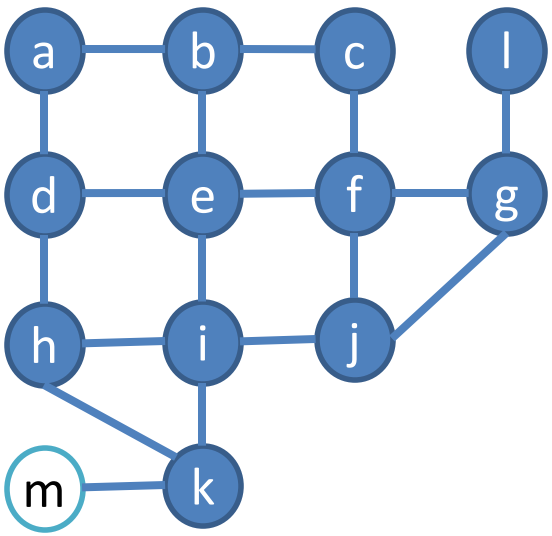


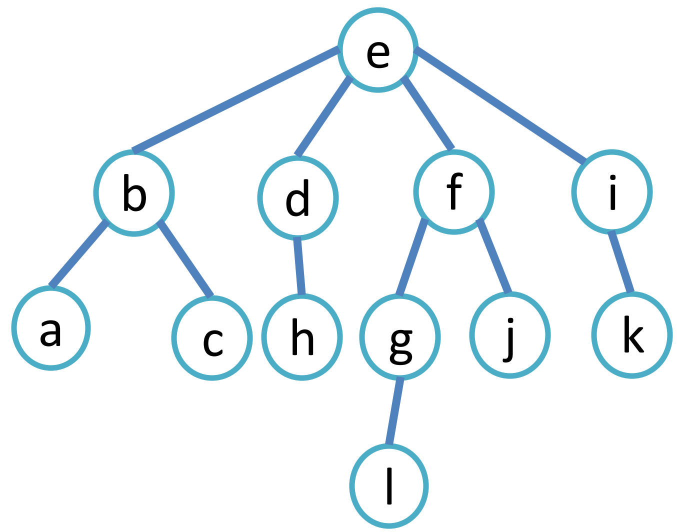


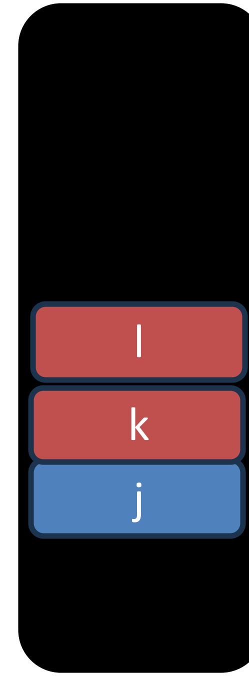



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

Breadth-first Search

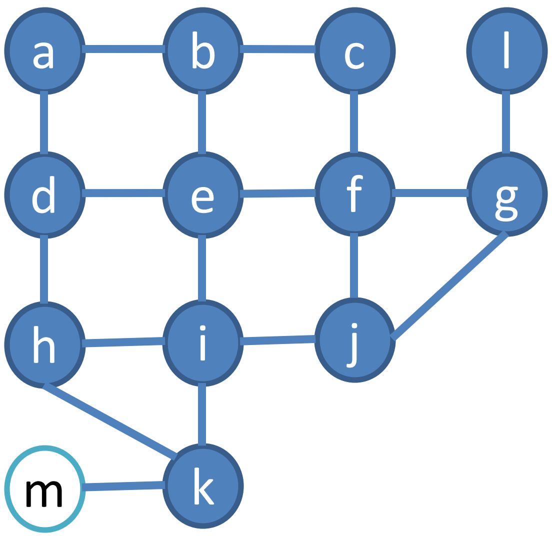


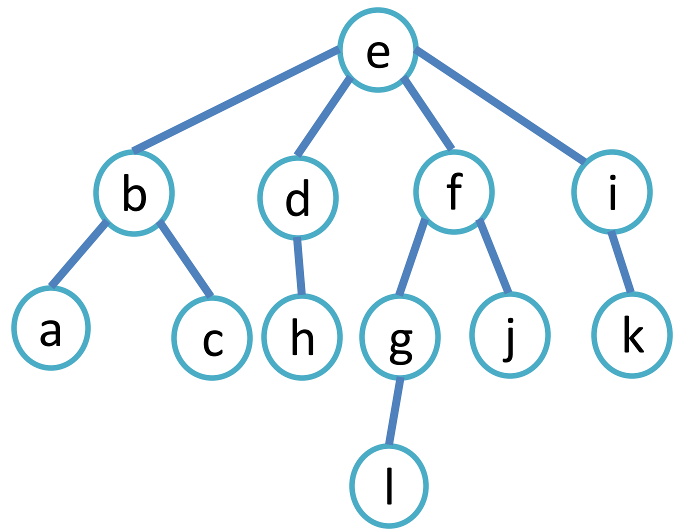


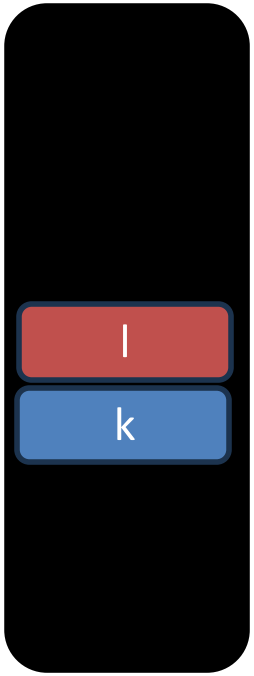



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                        Q.enqueue(w)
```

Breadth-first Search

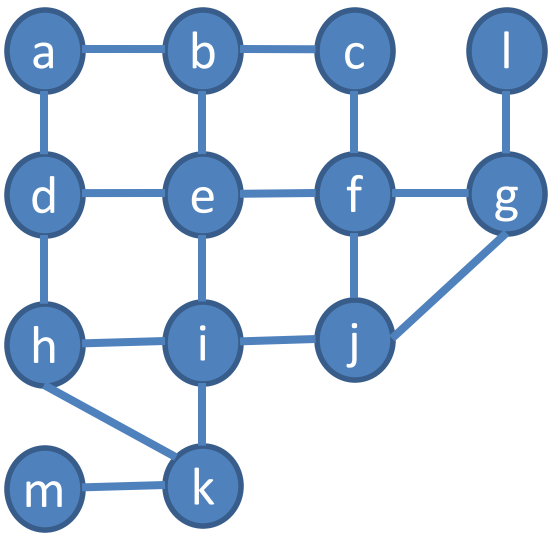


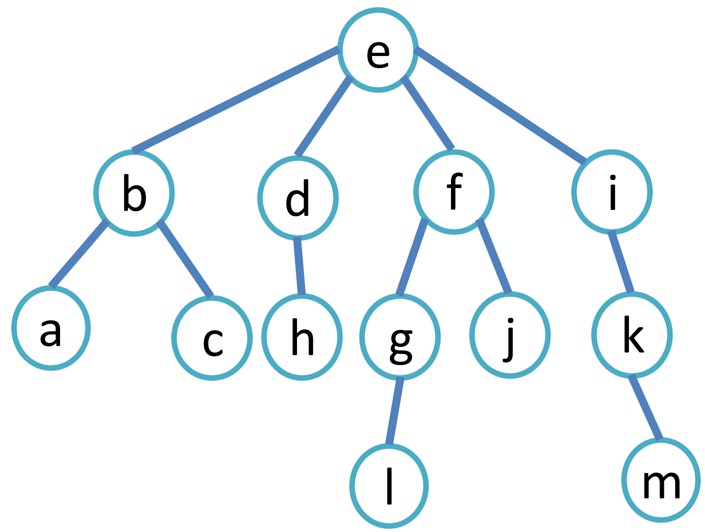


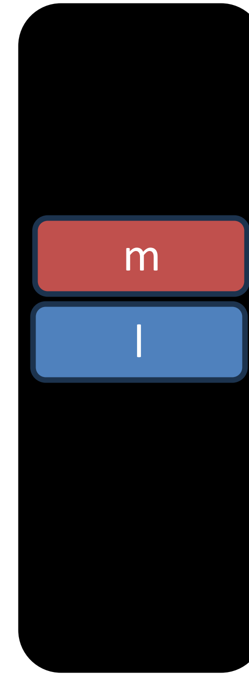



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

Breadth-first Search

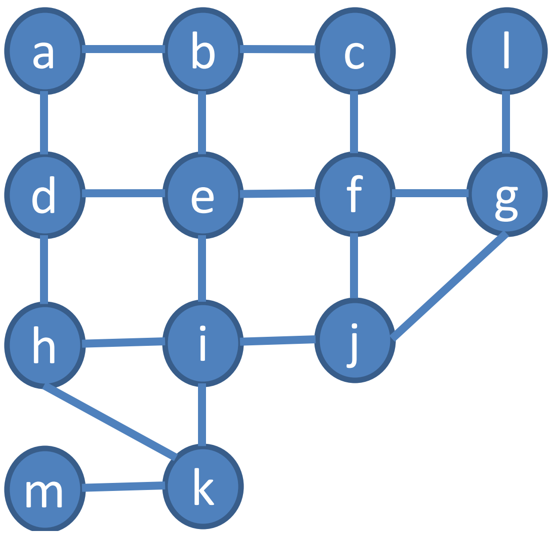


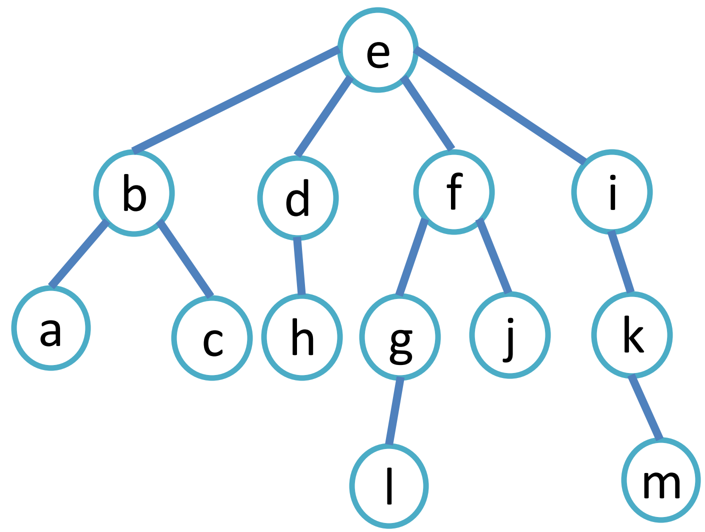


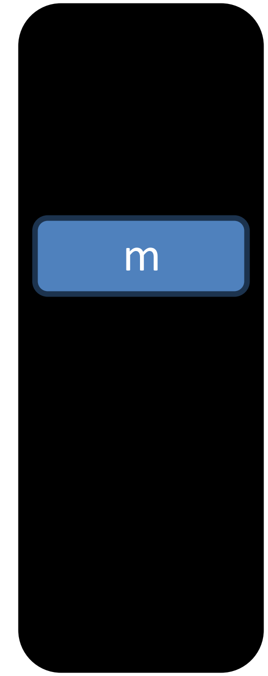



```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

Breadth-first Search







Algorithm 2 *BFS*(G : connected graph with vertices $v_1, v_2, \ldots, v_n$)

```
T = tree consisting only of vertex v1
L = empty list
put v1 in the list L of unprocessed vertices
while L is not empty do
    remove the first vertex, v, from L
    for each neighbor w of v do
        if w is not in L and not in T then
            add w to the end of the list L
            add w and edge {v, w} to T
        end if
    end for
end while
```

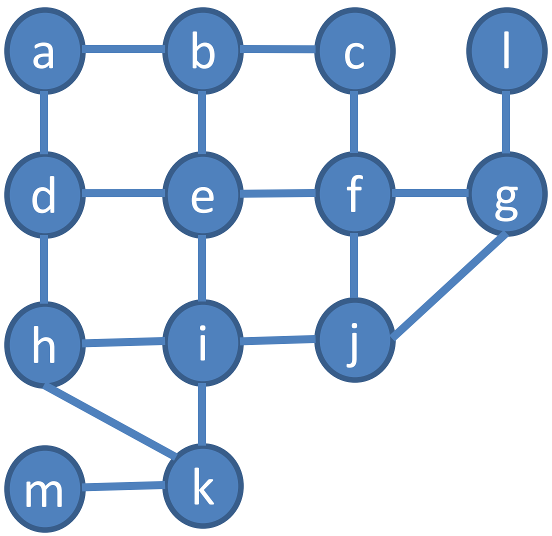
L = {k}

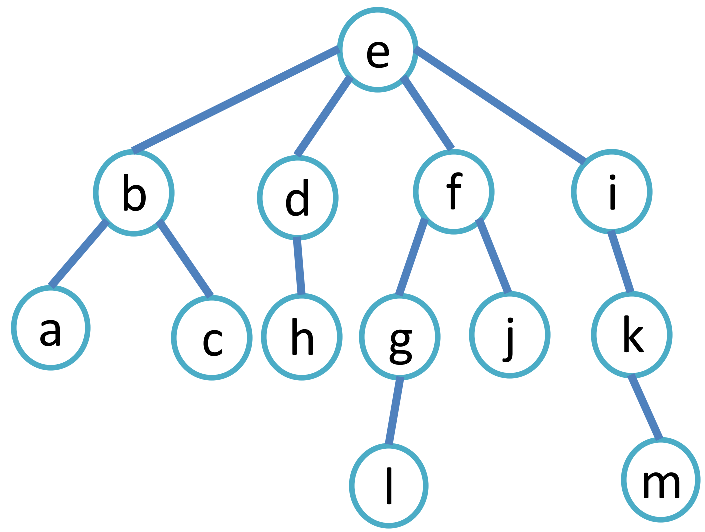
L = {}

L = {}

T = {(e, b), (e, d), (e, f), (e, i), (b, a), (b, c), (d, h), (f, g), (f, j), (i, k), (g, l), (k, m)}

Breadth-first Search





[redacted]

```
BFS (graph G, start vertex s)
    initially all nodes are unexplored
    let Q = queue, initialized with s
        while Q is not empty:
            v = Q.dequeue()
            if v is not yet explored:
                label v as explored
                foreach edge (v,w):
                    Q.enqueue(w)
```

Analysis

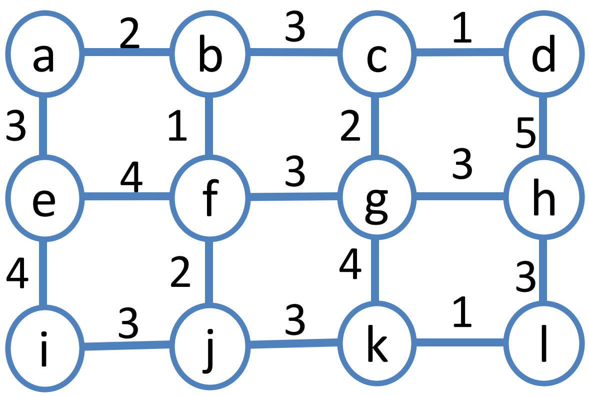
- Runs in O(V + E) time using a stack (LIFO)
- Like DFS, BFS explores all vertices and edges once, assuming an **adjacency list** representation.
- Every vertex is enqueued and dequeued exactly once.

Minimum spanning trees

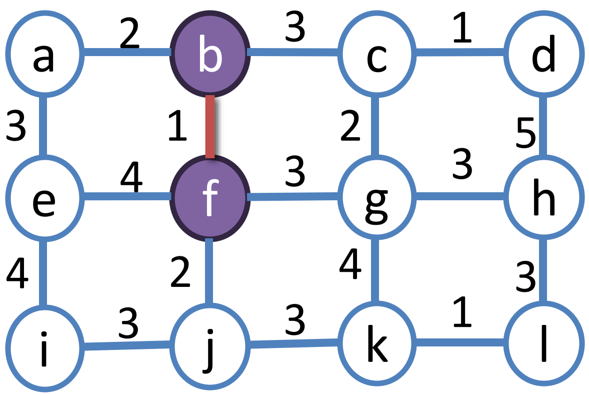
Prim’s algorithm

- Begin by choosing any edge with smallest weight, putting it into the spanning tree.
- Successively add to the tree edges of minimum weight that are incident to a vertex already in the tree, never forming a simple circuit with those edges already in the tree.
- Stop when n − 1 edges have been added

Example

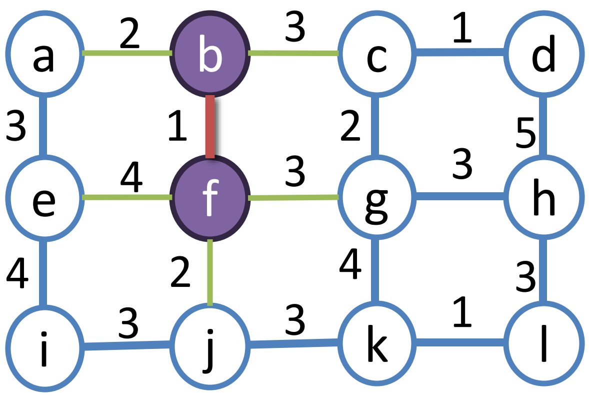


Example



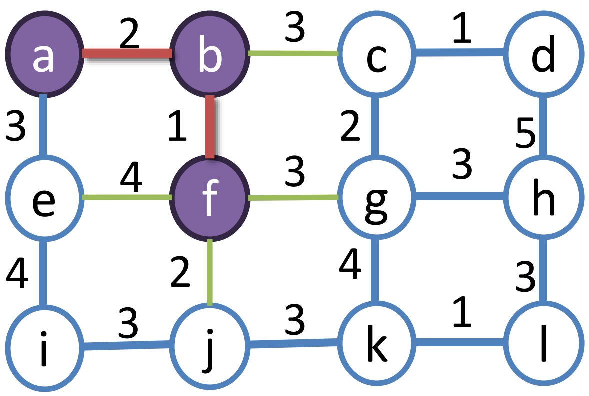
{b, f} 1

Example



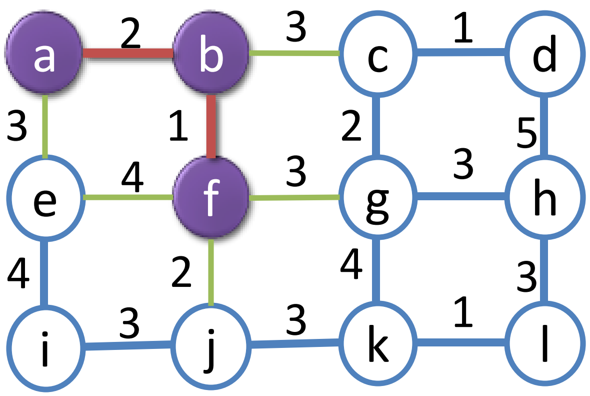
{b, f} 1

Example



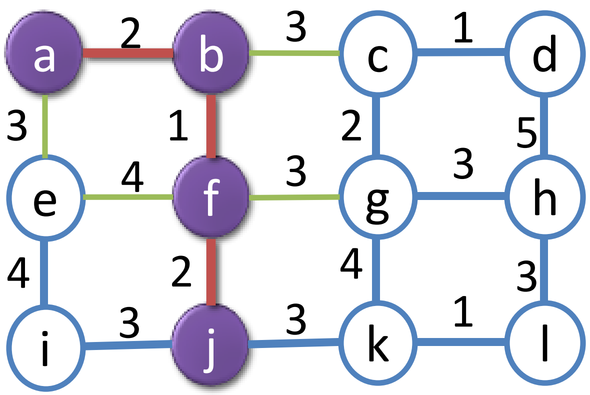
{b, f}	1
{a, b}	2

Example



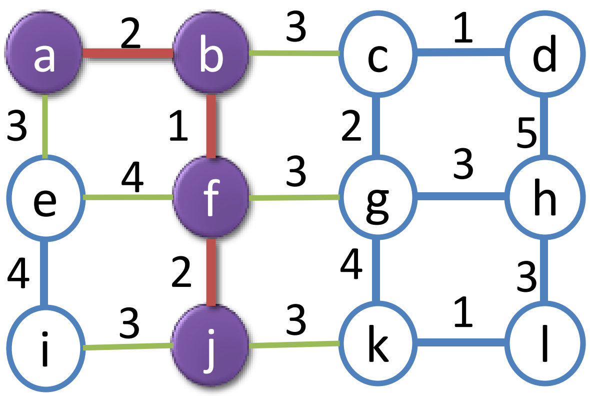
{b, f}	1
{a, b}	2

Example



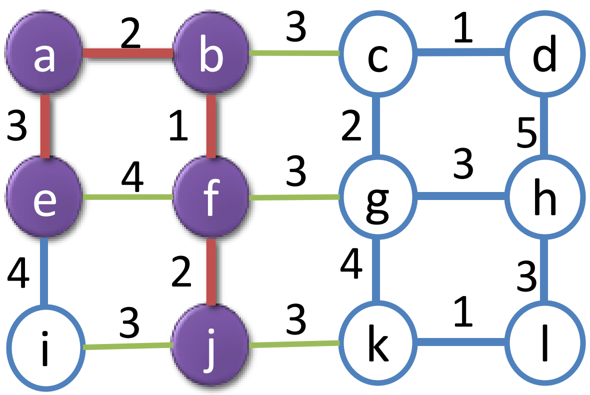
{b, f}	1
{a, b}	2
{f, j}	2

Example



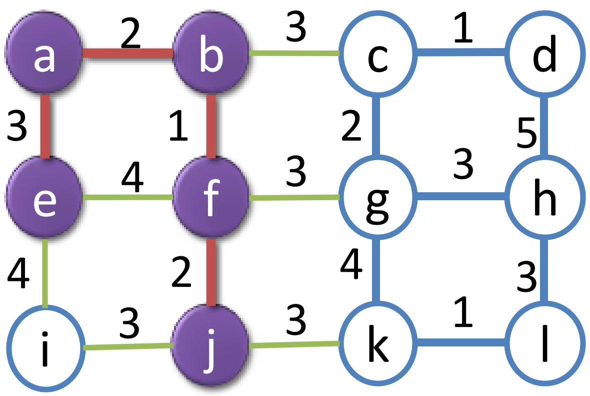
{b, f}	1
{a, b}	2
{f, j}	2

Example



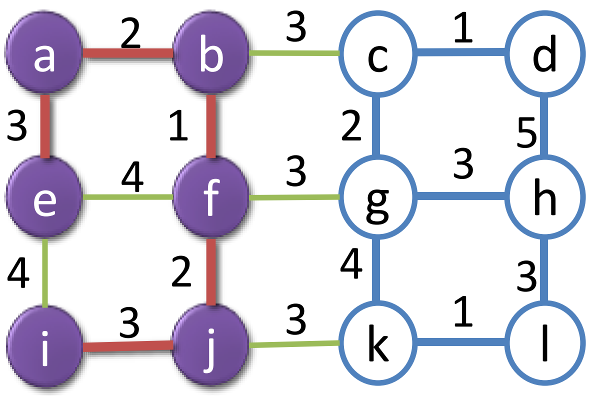
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3

Example



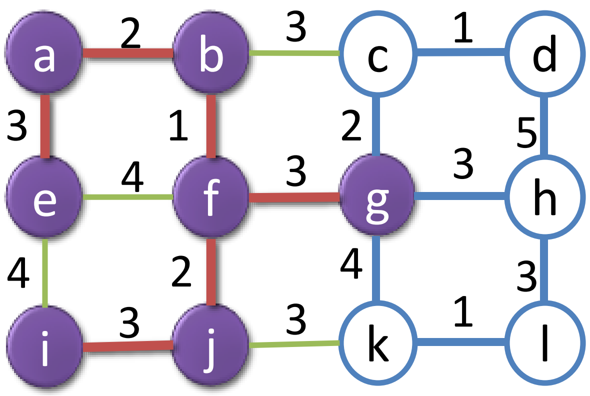
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3

Example



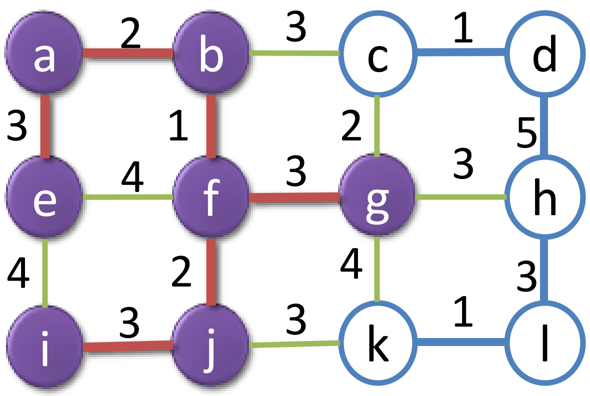
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3

Example



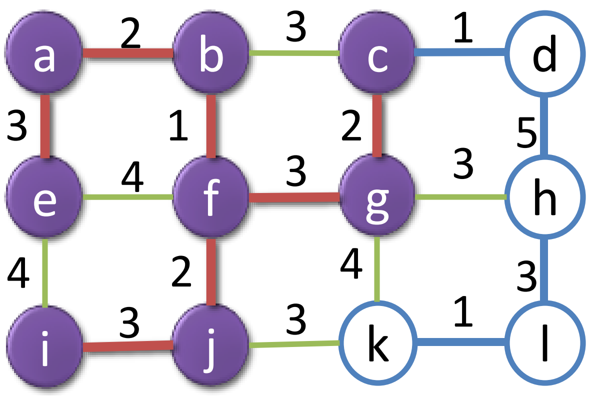
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3

Example



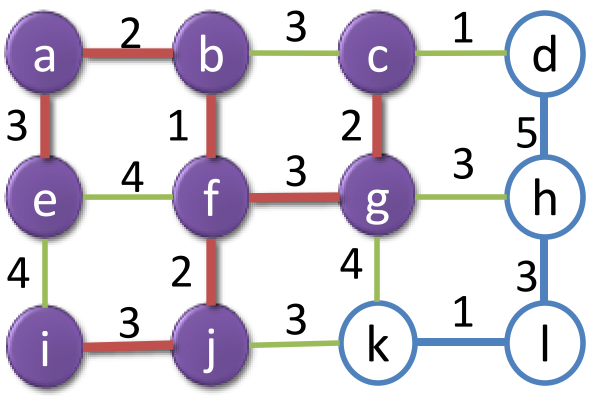
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3

Example



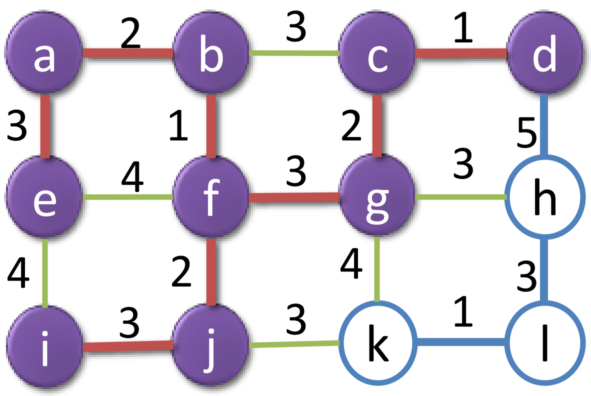
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2

Example



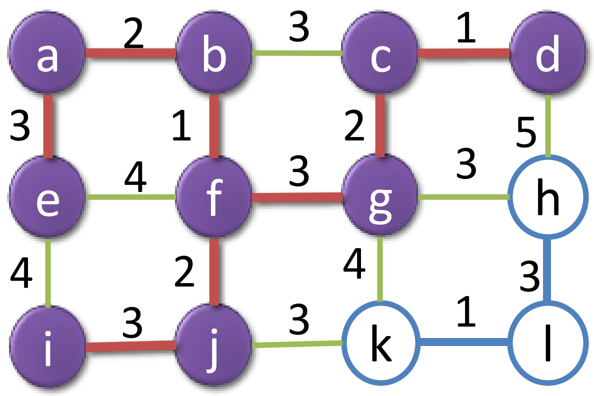
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2

Example



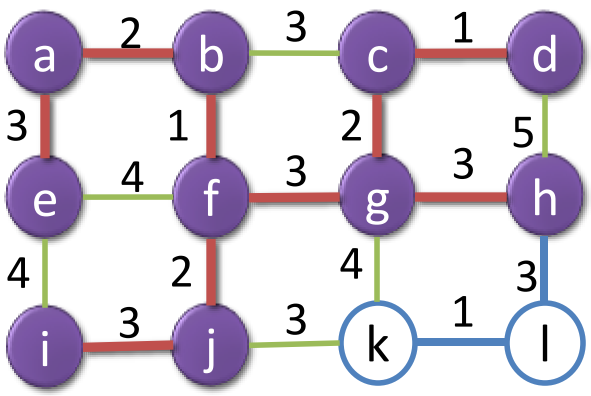
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2
{c, d}	1

Example



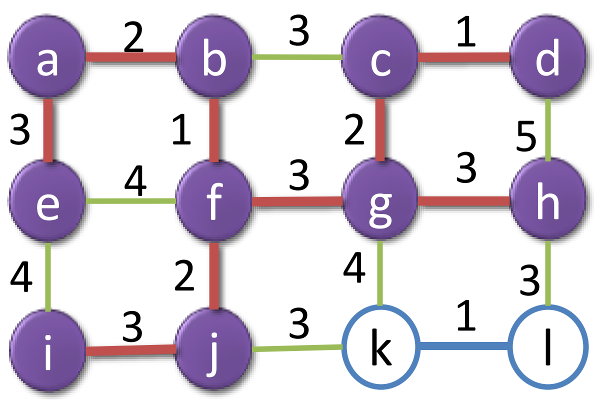
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2
{c, d}	1

Example



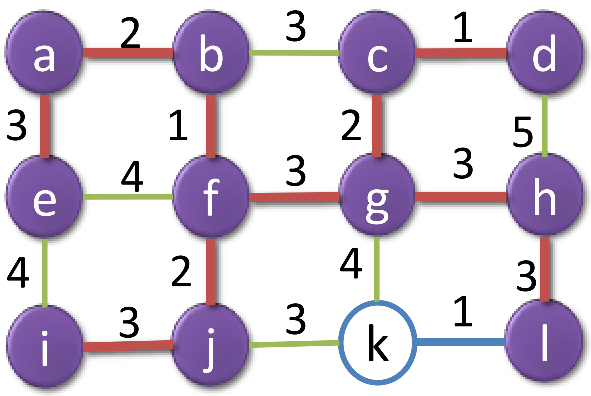
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2
{c, d}	1
{g, h}	3

Example



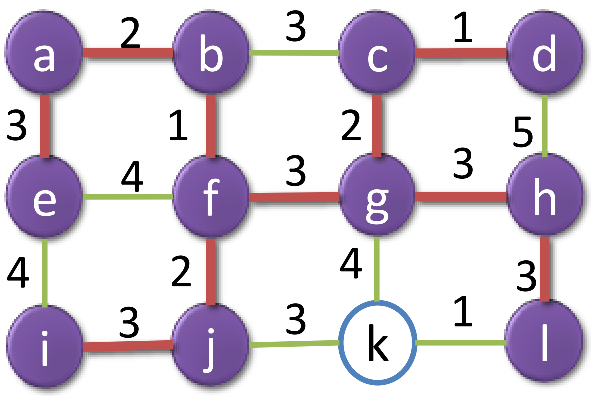
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2
{c, d}	1
{g, h}	3

Example



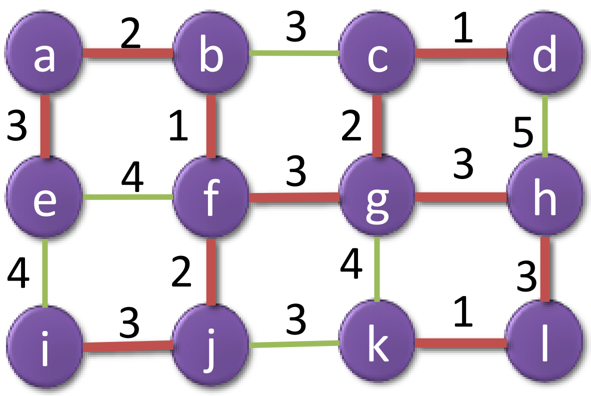
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2
{c, d}	1
{g, h}	3
{h, i}	3

Example



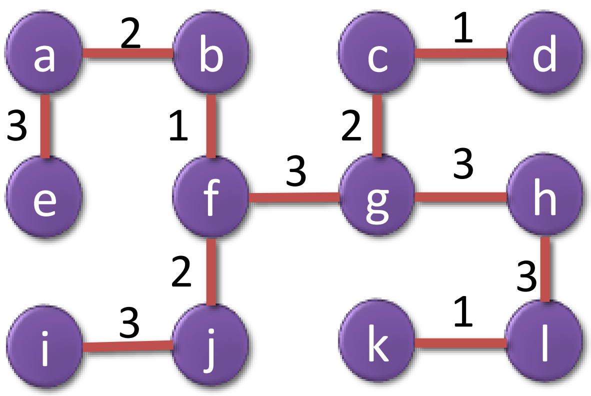
{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2
{c, d}	1
{g, h}	3
{h, i}	3

Example



{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2
{c, d}	1
{g, h}	3
{h, i}	3
{k, l}	1

Example



{b, f}	1
{a, b}	2
{f, j}	2
{a, e}	3
{i, j}	3
{f, g}	3
{c, g}	2
{c, d}	1
{g, h}	3
{h, i}	3
{k, l}	1
Total	**24**

Analysis

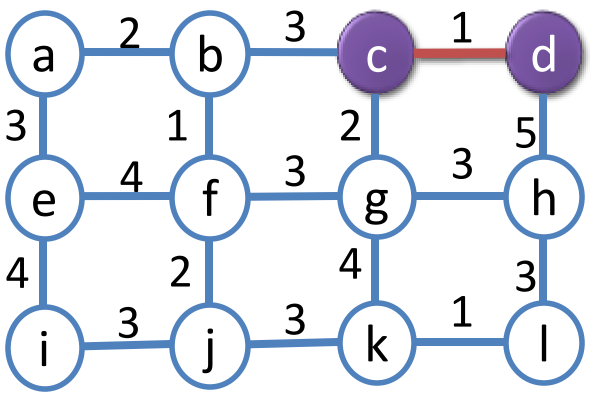
- Using a **binary min-heap** and an **adjacency list** to represent the graph, here's how the complexity arises:
- **Operations in Prim's (Min-Heap):**
  1. **Insert or update key of a vertex** in the min-heap → **log V**
  2. **Extract the minimum** from the min-heap → **log V**
- **Key Steps:**
  - You extract **V** vertices (each once):
    - V * log V → total time for **extract-min**
  - You examine all **E** edges (in adjacency list):
    - For each edge, you may perform a **decrease-key** operation → **log V**
    - So E * log V time for **decrease-key** operations

Kruskal’s Algorithm

Kruskal's Algorithm

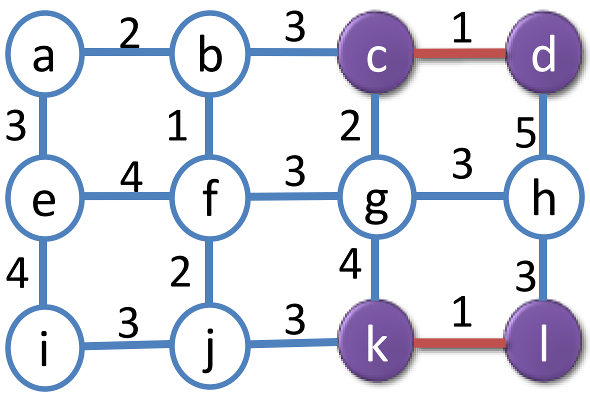
- Begin by choosing an edge in the graph with minimum weight.
- Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen.
- Stop after $n - 1$ edges have been selected.

Kruskal's Algorithm



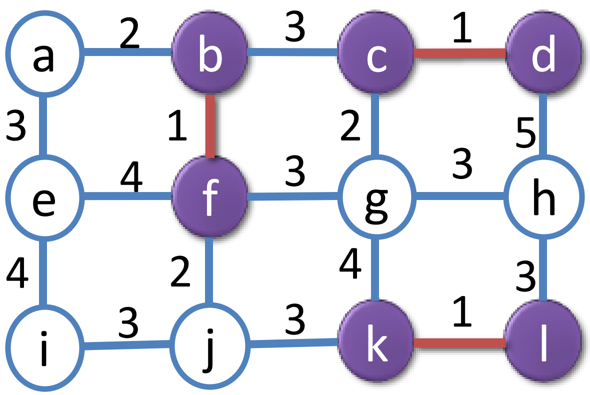
{c, d} 1

Kruskal's Algorithm



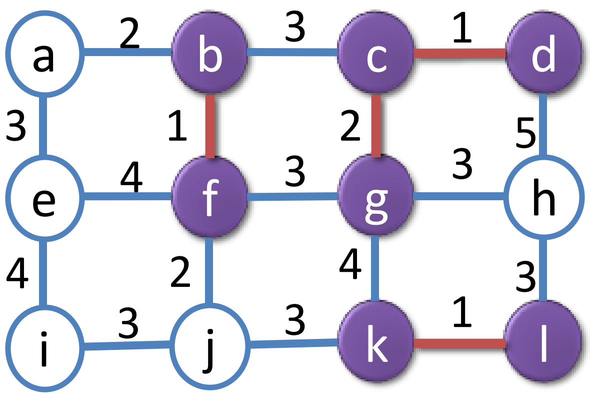
{c, d}	1
{k, l}	1

Kruskal's Algorithm



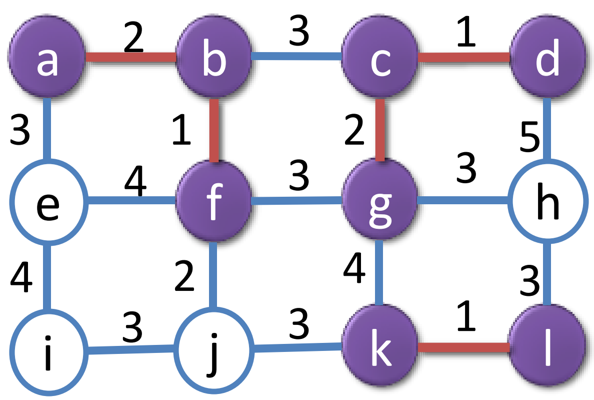
{c, d}	1
{k, l}	1
{b, f}	1

Kruskal's Algorithm



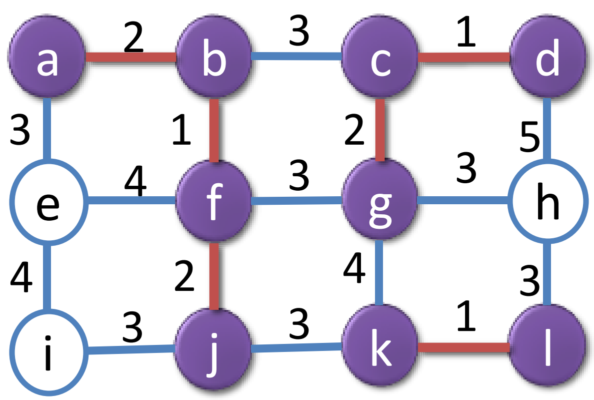
{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2

Kruskal's Algorithm



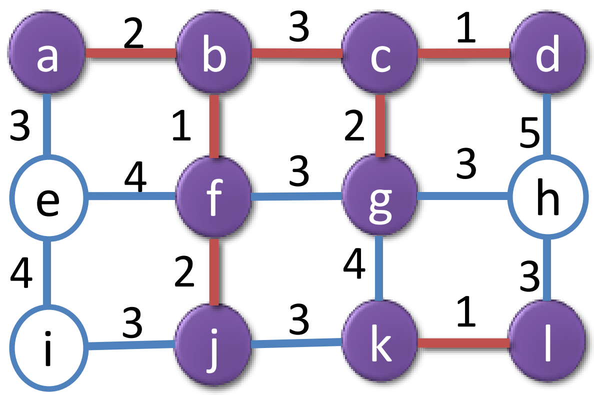
{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2
{a, b}	2

Kruskal's Algorithm



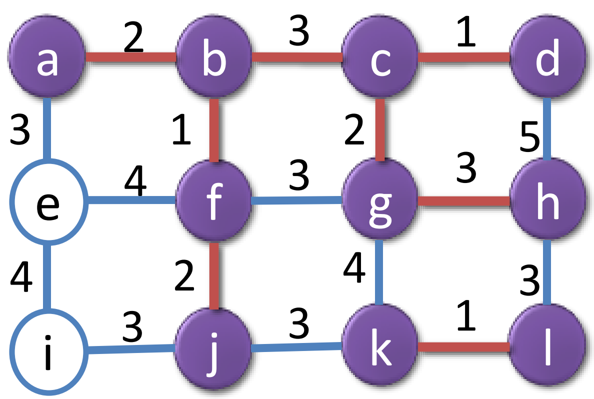
{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2
{a, b}	2
{f, j}	2

Kruskal’s Algorithm



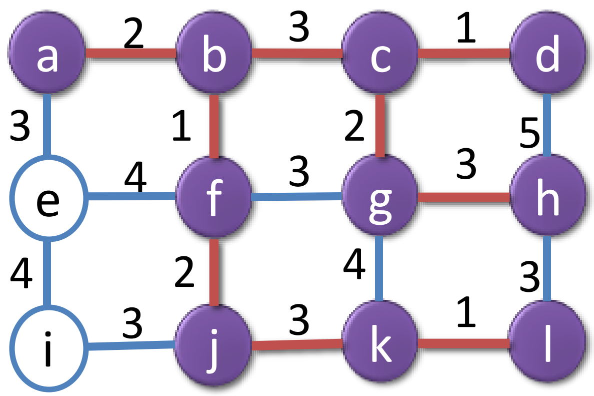
{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2
{a, b}	2
{f, j}	2
{b, c}	3

Kruskal's Algorithm



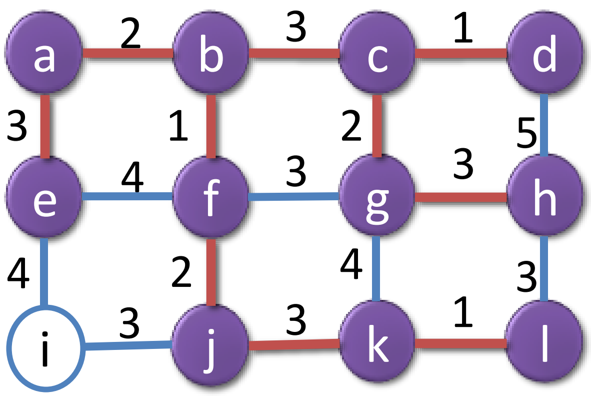
{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2
{a, b}	2
{f, j}	2
{b, c}	3
{g, h}	3

Kruskal's Algorithm



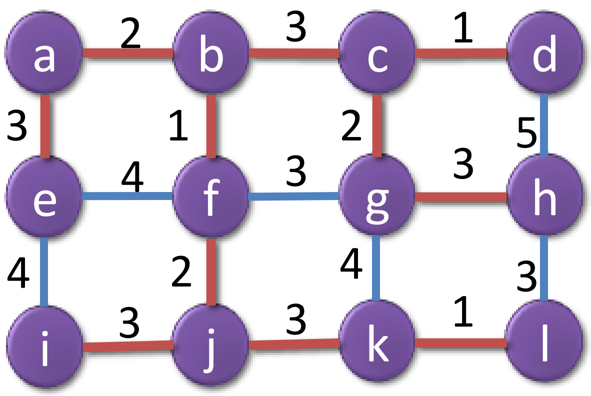
{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2
{a, b}	2
{f, j}	2
{b, c}	3
{g, h}	3
{j, k}	3

Kruskal's Algorithm



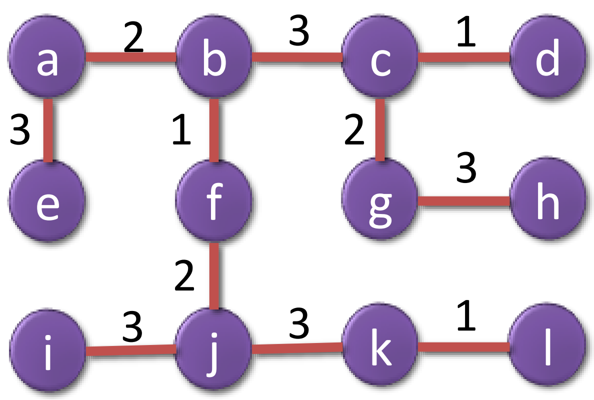
{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2
{a, b}	2
{f, j}	2
{b, c}	3
{j, k}	3
{g, h}	3
{a, e}	3

Kruskal's Algorithm



{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2
{a, b}	2
{f, j}	2
{b, c}	3
{j, k}	3
{g, h}	3
{a, e}	3
{i, j}	3

Kruskal's Algorithm



{c, d}	1
{k, l}	1
{b, f}	1
{c, g}	2
{a, b}	2
{f, j}	2
{b, c}	3
{j, k}	3
{g, h}	3
{a, e}	3
{i, j}	3

Analysis

End of Lecture