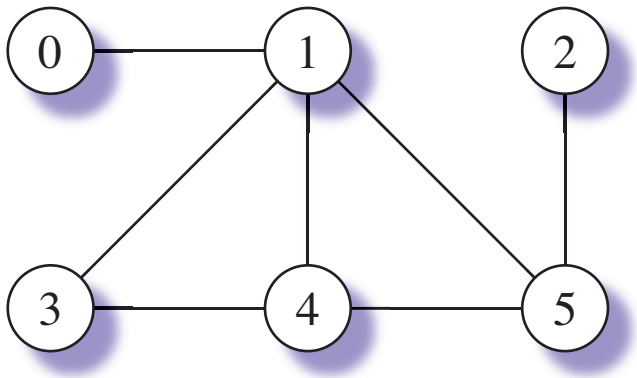
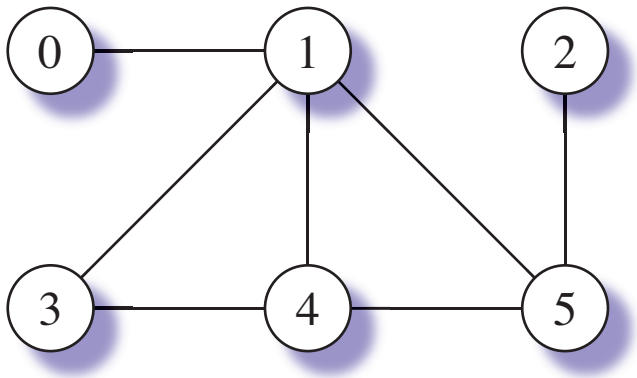
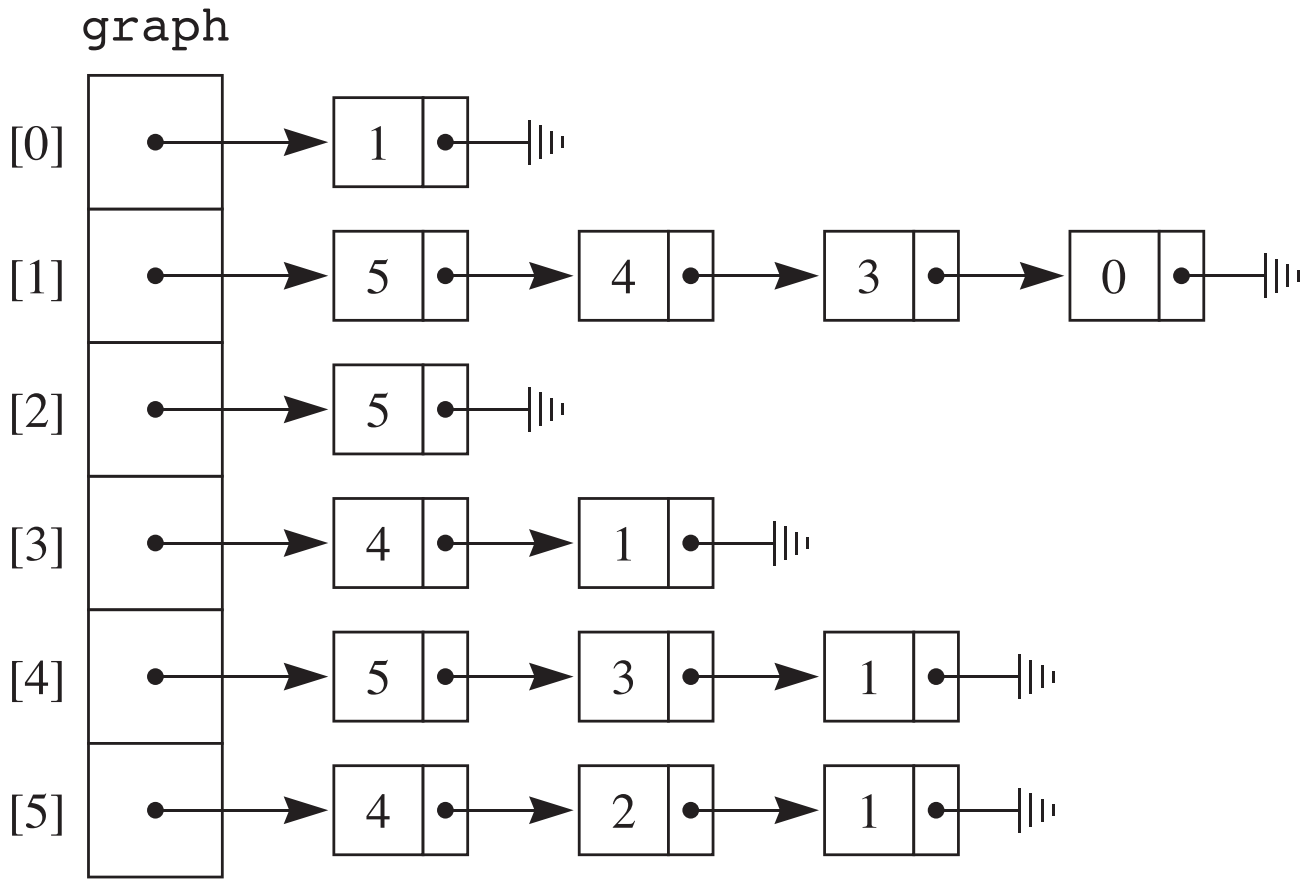
# Graphs



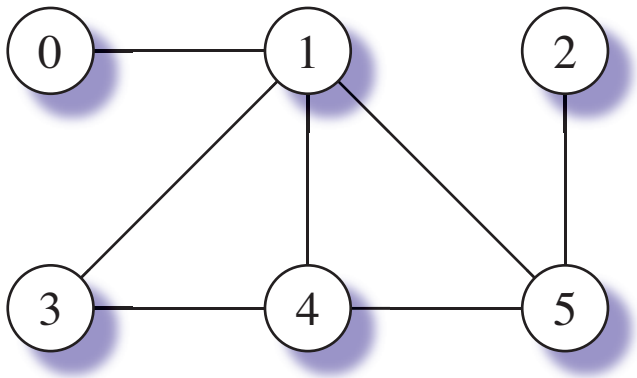

**(a)** A graph.




**(a)** A graph.



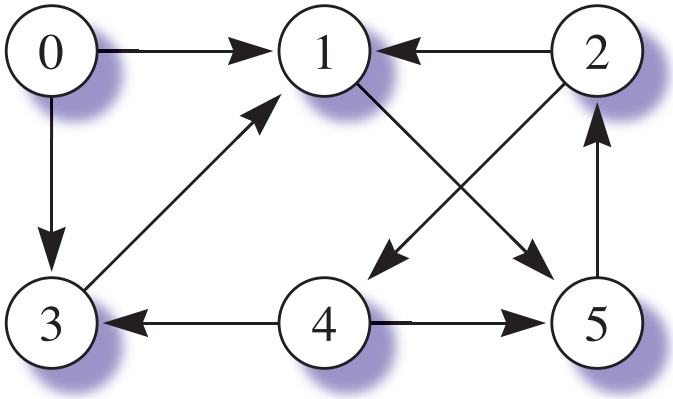

**(b)** The adjacency-list implementation of the graph.



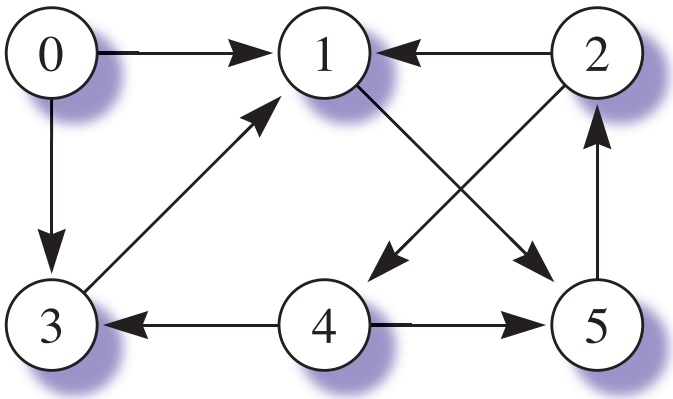

**(a)** A graph.

$$
\begin{array}{c}
\texttt{graph} \\
[0,] \\
[1,] \\
[2,] \\
[3,] \\
[4,] \\
[5,]
\end{array}
\begin{array}{c}
\begin{array}{cccccc} [,0] & [,1] & [,2] & [,3] & [,4] & [,5] \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

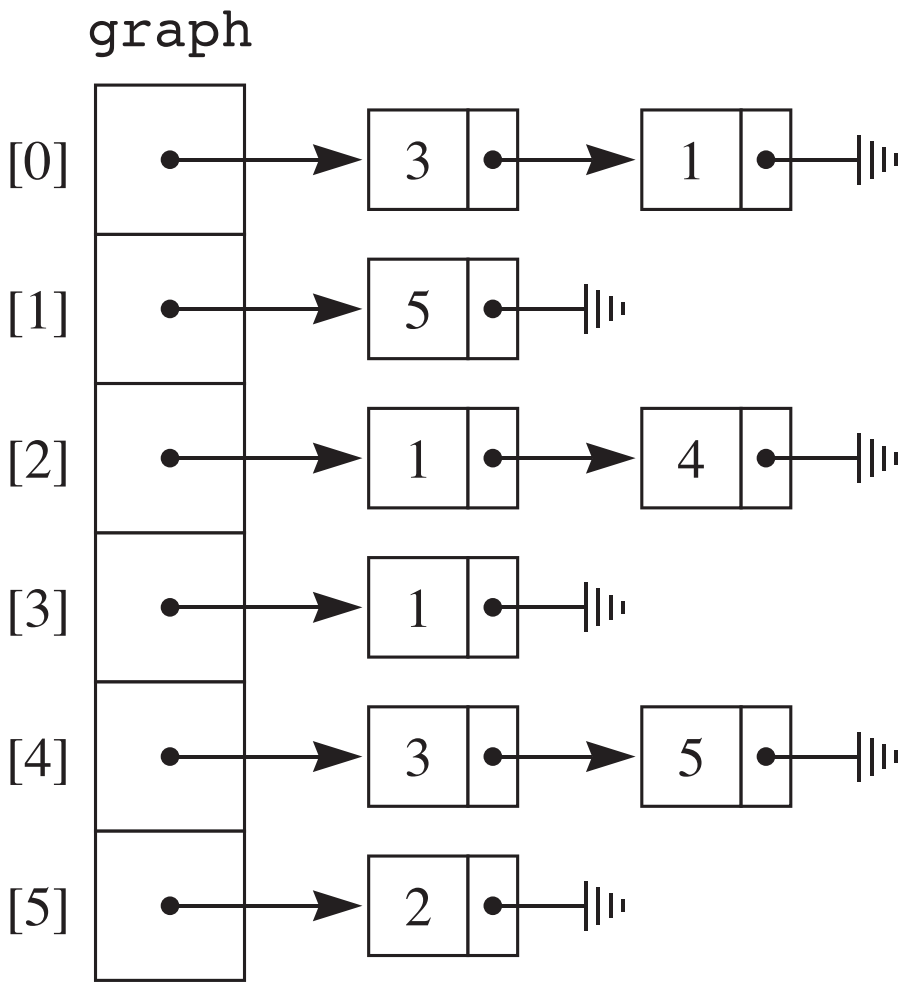
**(c)** The adjacency-matrix implementation of the graph.



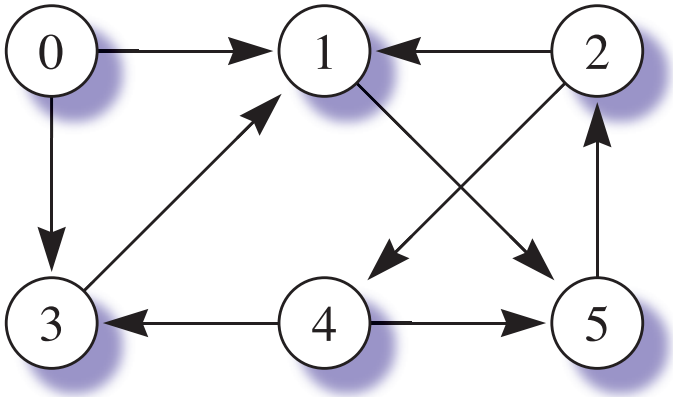

**(a)** A directed graph.




**(a)** A directed graph.




**(b)** The adjacency-list implementation of the digraph.




**(a)** A directed graph.

$$
\texttt{graph}\quad
\begin{array}{c}
\\ [0,] \\ [1,] \\ [2,] \\ [3,] \\ [4,] \\ [5,]
\end{array}
\begin{array}{c}
\begin{array}{cccccc} [,0] & [,1] & [,2] & [,3] & [,4] & [,5] \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

**(c)** The adjacency-matrix implementation of the digraph.

Demo insertEdge for graph and directed graph

```
enum ColorType {WHITE, GRAY, BLACK};
class GraphVertex {
public:
    ColorType color;
    int discovered;
    int finished;
    int distance;
    int predecessor;
};

class Graph {
private:
    int _numVertices;
    bool _isDigraph;
    int _time;
    ArrayT<GraphVertex> _vertex;
    ArrayT<shared_ptr<ListL<int>>> _graph;
    ArrayT<shared_ptr<ListLIterator<int>>> _graphIter;
```

```
public:
    Graph(bool isDigraph, int numVert);
    // Pre: 0 < numVert.
    // Post: The graph is allocated and initialized to have
    // zero edges and numVert vertices.

    void breadthFirstSearch(int s, ostream &os);
    // Post: A breadth-first search of this graph beginning
    // at vertex s is output to os.
    // Post: Discovered time, finished time, distance from s,
    // and predecessor vertex for each vertex is output to os.

    void depthFirstSearch(int u, ostream &os);
    // Post: A depth-first search of this graph beginning
    // at vertex s is output to os.
    // Post: Discovered time, finished time, and predecessor
    // vertex for each vertex is output to os.
```

```
void insertEdge(int from, int to);
// Pre: 0 <= from < _numVertices, and 0 <= to < _numVertices.
// Post: If the edge <from, to> is not already in the graph,
// it is installed; otherwise, the graph is unchanged.

int numEdges();
// Post: The number of edges is returned.

void removeEdge(int from, int to);
// Pre: 0 <= from < _numVertices, and 0 <= to < _numVertices.
// Post: If the edge <from, to> is in the graph, it is removed;
// otherwise, the graph is unchanged.
```

```
void writeAdjacencyLists(ostream &os);
// Post: The adjacency lists for each vertex with a nonempty list
// is output to os.

void writeComponents(ostream &os);
// Post: A list of all the connected components is output to os,
// with a count of how many components exist.

void writePath(int from, int to, ostream &os);
// Pre: 0 <= from < _numVertices, and 0 <= to < _numVertices.
// Post: If there is a path from vertex "from" to vertex "to" a
// path of smallest distance and its length is output to os;
// otherwise, a statement that no path exists is output.
```

```
private:
    void bfs(int s, ostream &os);
    // Post: A breadth-first search of this graph beginning at
    // vertex s is performed without initialization.

    void dfs(int u, ostream &os);
    // Pre: color for each vertex is well-defined and the color
    // of vertex u is WHITE.
    // Post: A depth-first search of this graph beginning at
    // vertex u is performed with the color of each vertex visited,
    // including the color of u, set to BLACK.
    // Post: Vertex u is output to os.

    void initGraph();
    // Post: Every vertex of the graph is initialized in preparation
    // for a breadth-first or depth-first search.
```

```
    void writePathHelper(int from, int to, ostream &os);

    void writeVerticesPostBreadth(ostream &os);
    // Post: Discovered time, finished time, predecessor vertex,
    // and distance from s,for each vertex is output to os.

    void writeVerticesPostDepth(ostream &os);
    // Post: Discovered time, finished time, and predecessor
    // vertex for each vertex is output to os.
};
```

```
// ========= Constructor =========
Graph::Graph(bool isDigraph, int numVert):
    _time(0),
    _numVertices(numVert),
    _isDigraph(isDigraph),
    _vertex(numVert),
    _graph(numVert),
    _graphIter(numVert) {
    for (int i = 0; i < numVert; i++) {
        _graph[i] = make_shared<ListL<int>>();
        _graphIter[i] = make_shared<ListLIterator<int>>();
        _graphIter[i]->setIterListL(_graph[i]);
    }
}
```

```
// ========= insertEdge =========
void Graph::insertEdge(int from, int to) {
    if ((from < 0) || (_numVertices <= from) ||
          (to < 0) || (_numVertices <= to)) {
        cerr << "insertEdge precondition violated: "
             << "from or to out of range." << endl;
        cerr << "from == " << from << "  to == " << to << endl;
        throw -1;
    }
    if (!_graph[from]->contains(to)) {
        _graph[from]->prepend(to);
    }
    if (!_isDigraph && !_graph[to]->contains(from)) {
        _graph[to]->prepend(from);
    }
}
```

# Graph traversals

- Breadth-first search.
- Depth-first search.

Neither of these traversals are unique with adjacency lists.

Figure 11.8

```
// ========= initGraph =========
void Graph::initGraph() {
    _time = 0;
    for (int i = 0; i < _numVertices; i++) {
        _vertex[i].color = WHITE;
        _vertex[i].discovered = -1;
        _vertex[i].finished = -1;
        _vertex[i].distance = -1;
        _vertex[i].predecessor = -1;
    }
}
```

# Breadth-first search

```
// ========= Breadth first search =========
void Graph::breadthFirstSearch(int s, ostream &os) {
    initGraph();
    os << endl;
    bfs(s, os);
    writeVerticesPostBreadth(os);
}

void Graph::bfs(int s, ostream &os) {
    cerr << "Graph::bfs: Exercise for the student." << endl;
    throw -1;
}
```

(**a**) The C++ implementation.

```
bfs(s, os)
    Set the color of s to GRAY.
    Set the distance of s to 0.
    Declare a local queue of integers, initialized to the empty queue.
    Set the discovered time of s to ++_time.
    queue.enqueue(s)
    while (queue is not empty)
        u = queue.dequeue()
        Stream u to os.
        Set the finished time of u to ++_time.
        for (each vertex v adjacent to u)
            if (the color of v is WHITE)
                Set the color of v to GRAY.
                Set the distance of v to the distance of u + 1.
                Set the predecessor of v to u.
                Set the discovered time of v to ++_time.
                queue.enqueue(v)
        Set the color of u to BLACK.
```

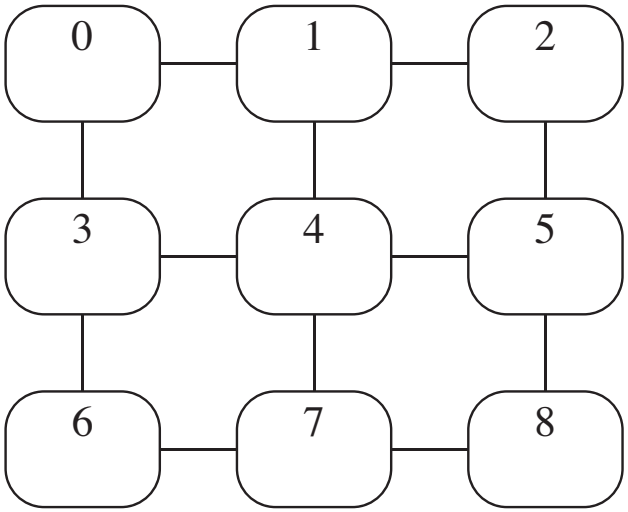
(**b**) The algorithm for `bfs()`.

# Trace of breadth-first search from vertex 6 with the following adjacency lists

```
0: (1, 3)

1: (0, 2, 4)

2: (1, 5)

3: (0, 4, 6)

4: (1, 3, 5, 7)

5: (2, 4, 8)

6: (3, 7)

7: (4, 6, 8)

8: (5, 7)
```

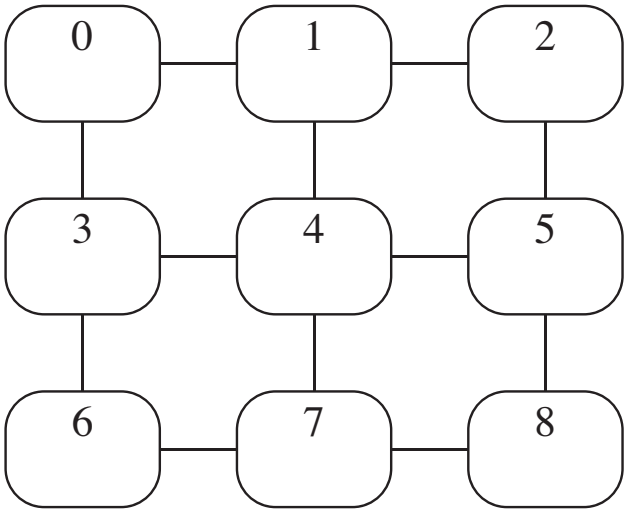
os:

queue:



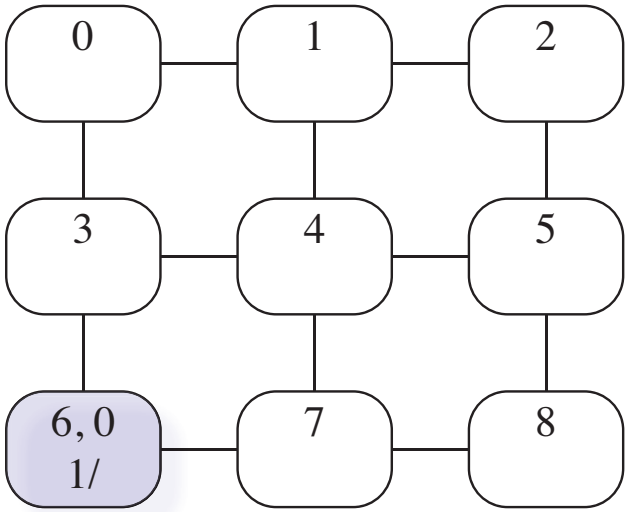

**(a)** Initial graph.

os:
queue:

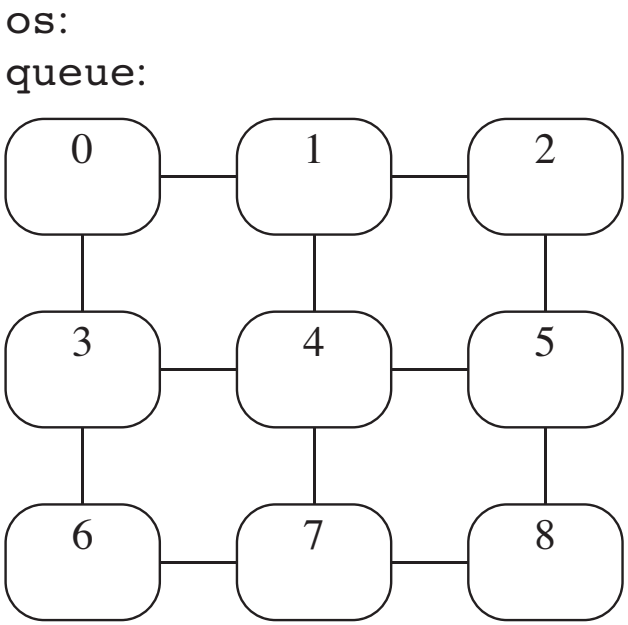



**(a)** Initial graph.

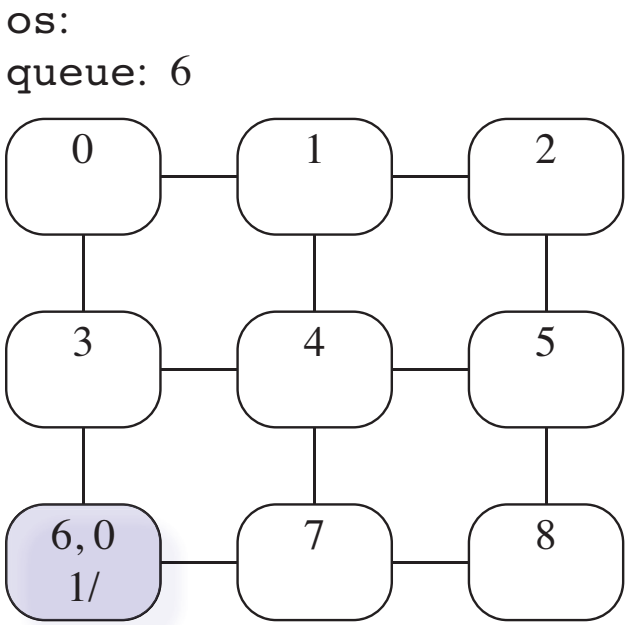
os:
queue: 6



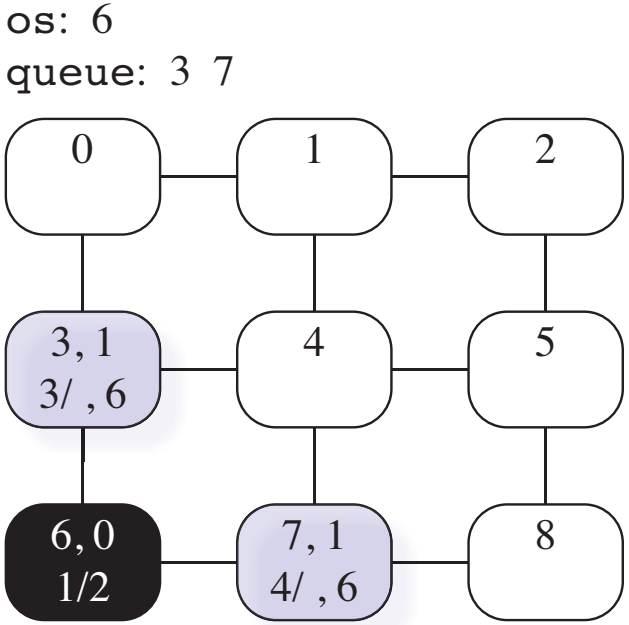

**(b)** Initialization of `while`.




**(a)** Initial graph.

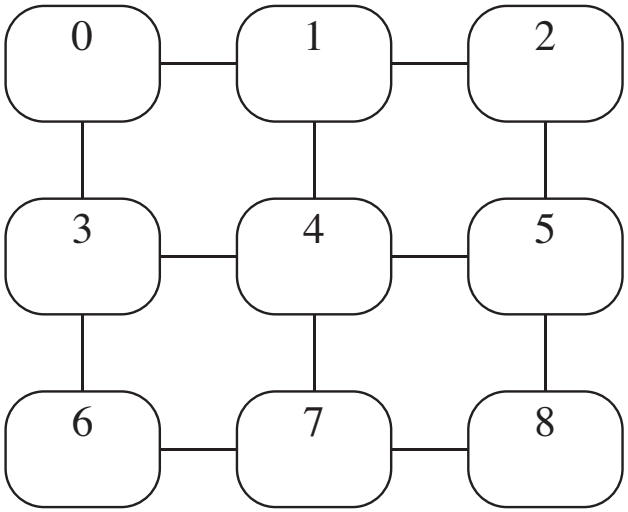



**(b)** Initialization of `while`.



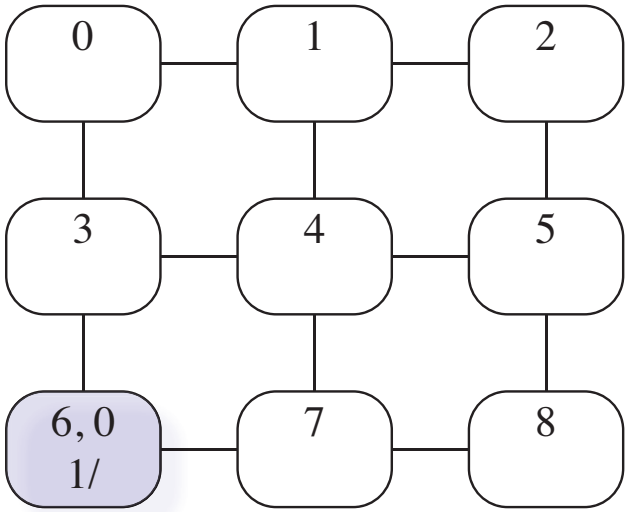

**(c)** First exection of `while`.

os:
queue:



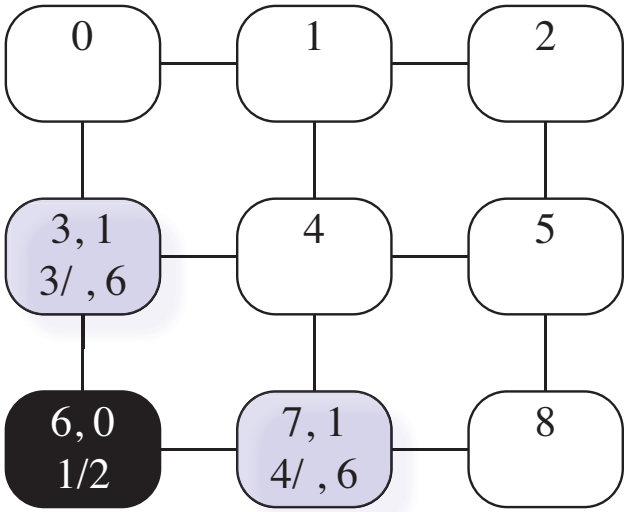

**(a)** Initial graph.

os:
queue: 6



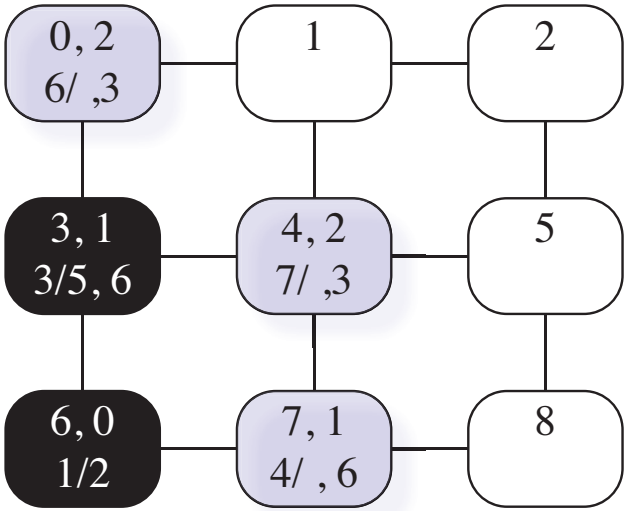

**(b)** Initialization of `while`.

os: 6
queue: 3 7



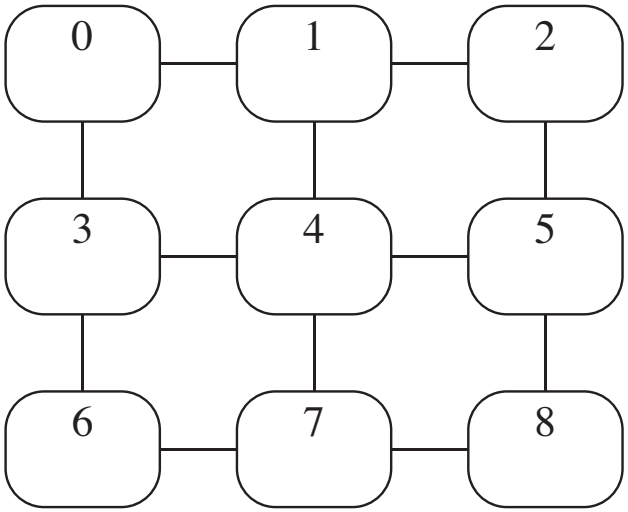

**(c)** First exection of `while`.

os: 6 3
queue: 7 0 4



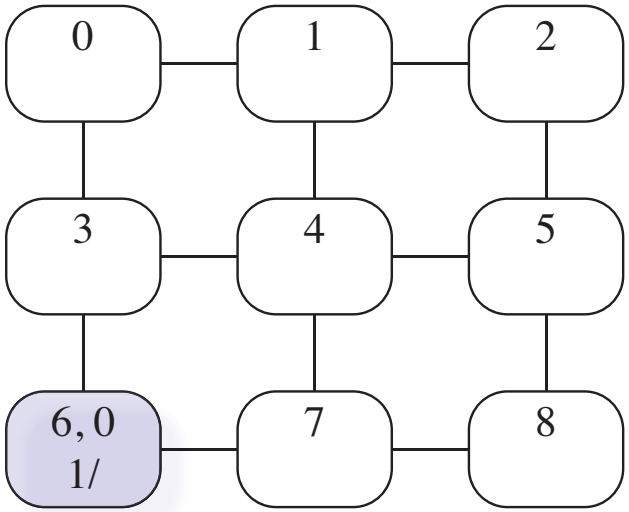

**(d)** Second exection of `while`.

os:
queue:



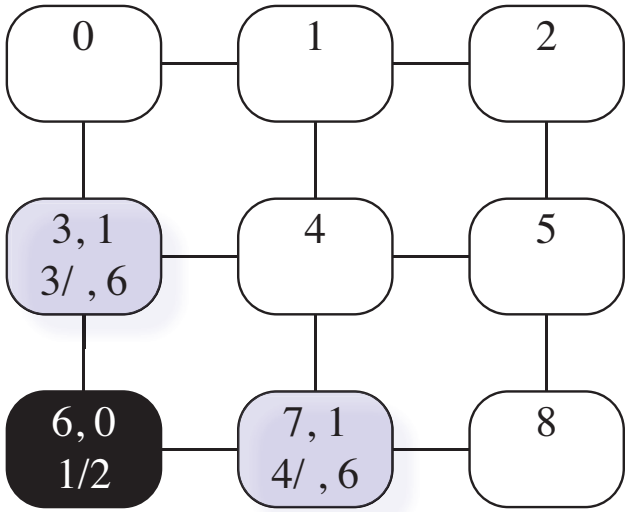

**(a)** Initial graph.

os:
queue: 6



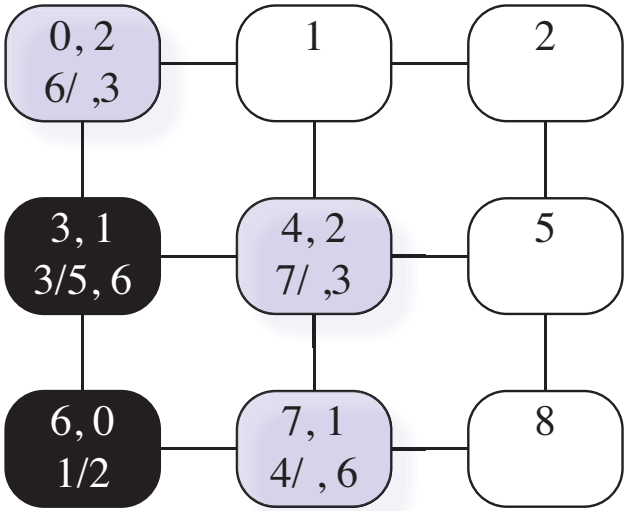

**(b)** Initialization of `while`.

os: 6
queue: 3 7



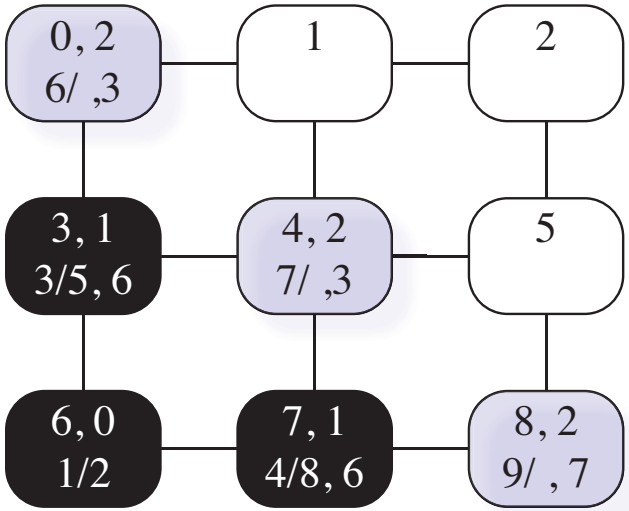

**(c)** First exection of `while`.

os: 6 3
queue: 7 0 4



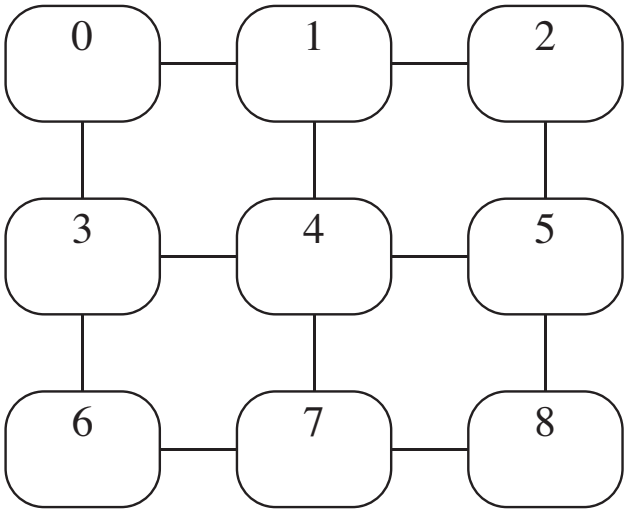

**(d)** Second exection of `while`.

os: 6 3 7
queue: 0 4 8



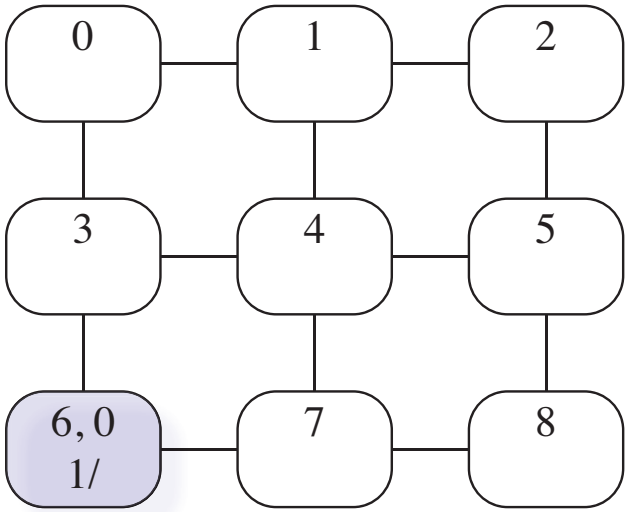

**(e)** Third exection of `while`.

os:
queue:



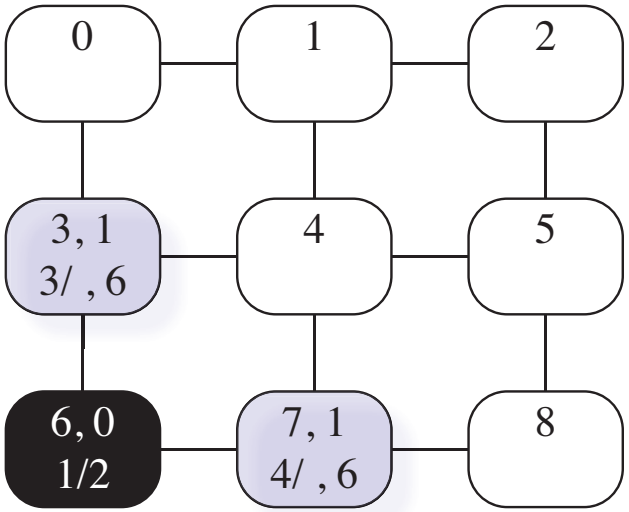

**(a)** Initial graph.

os:
queue: 6



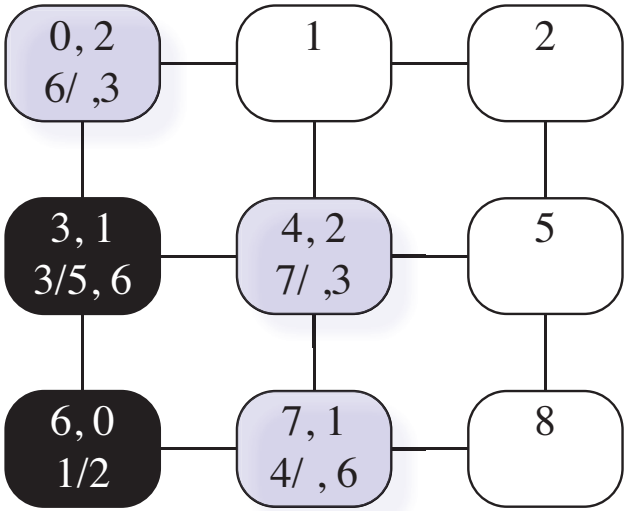

**(b)** Initialization of `while`.

os: 6
queue: 3 7



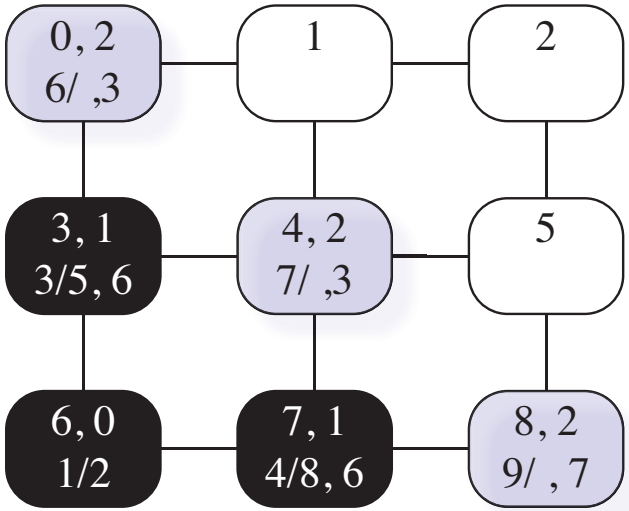

**(c)** First exection of `while`.

os: 6 3
queue: 7 0 4



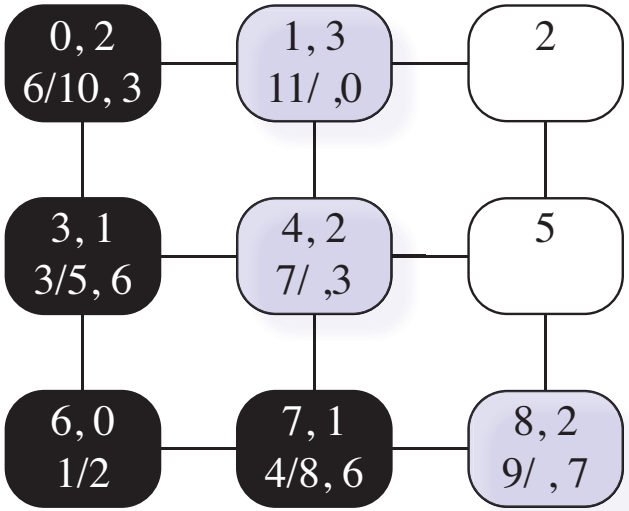

**(d)** Second exection of `while`.

os: 6 3 7
queue: 0 4 8



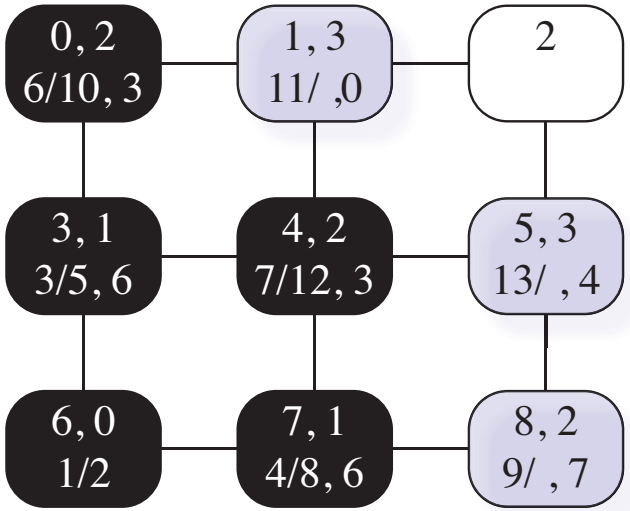

**(e)** Third exection of `while`.

os: 6 3 7 0
queue: 4 8 1



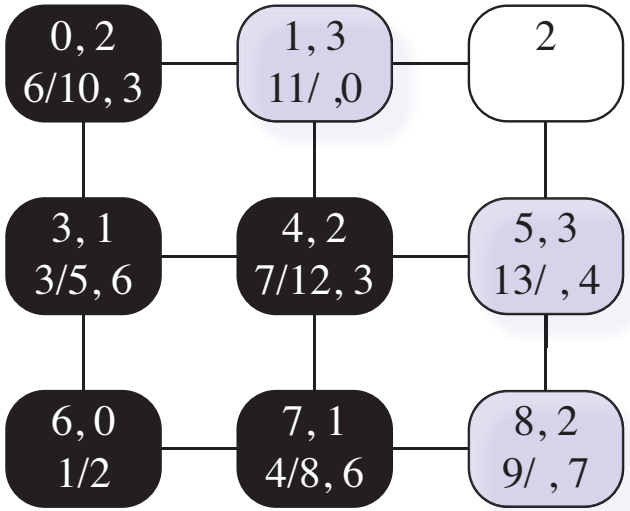

**(f)** Fourth exection of `while`.

os: 6 3 7 0 4
queue: 8 1 5



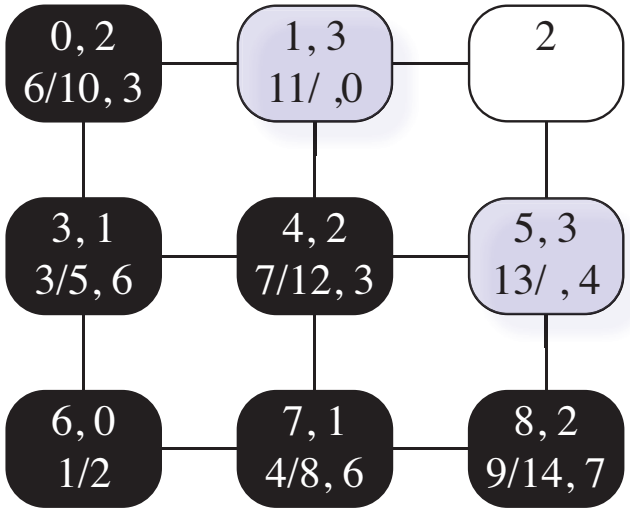

**(g)** Fifth exection of `while`.

os: 6 3 7 0 4
queue: 8 1 5



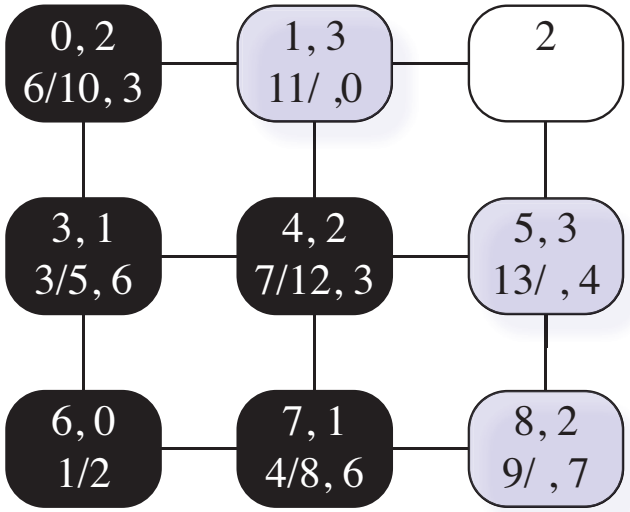

**(g)** Fifth exection of `while`.

os: 6 3 7 0 4 8
queue: 1 5



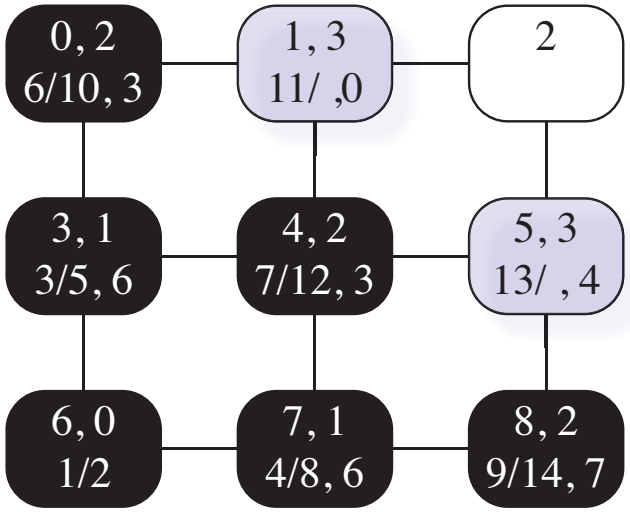

**(h)** Sixth exection of `while`.

os: 6 3 7 0 4
queue: 8 1 5



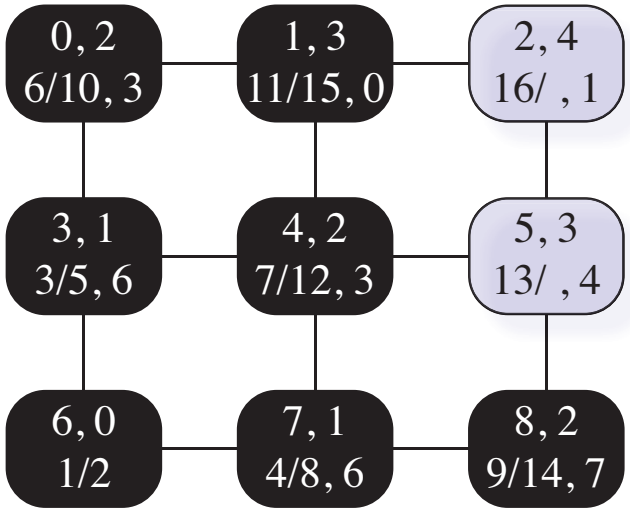

**(g)** Fifth exection of `while`.

os: 6 3 7 0 4 8
queue: 1 5



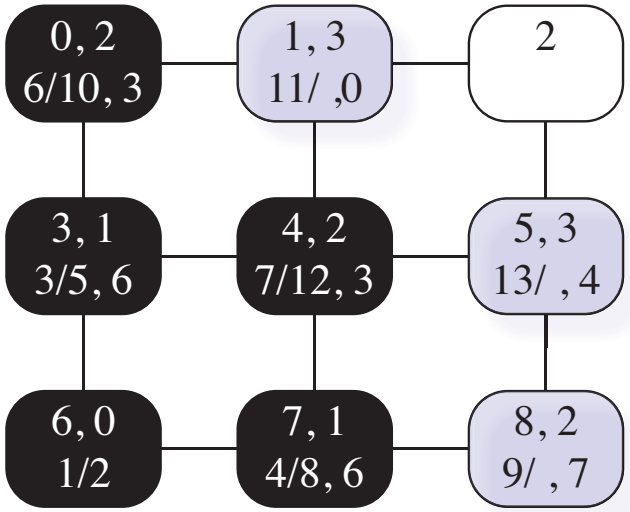

**(h)** Sixth exection of `while`.

os: 6 3 7 0 4 8 1
queue: 5 2



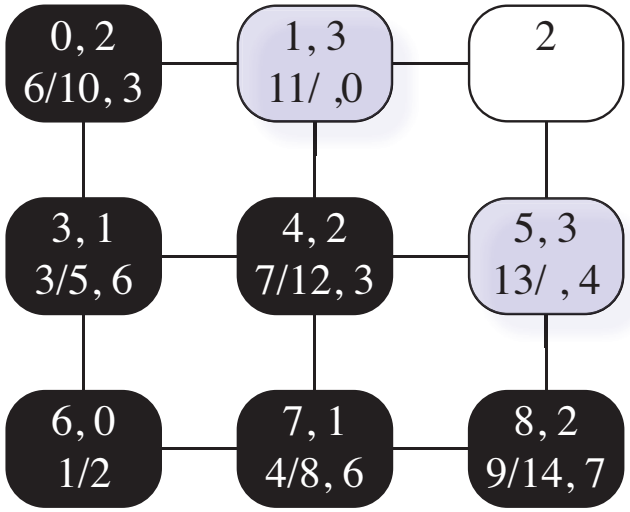

**(i)** Seventh exection of `while`.

os: 6 3 7 0 4
queue: 8 1 5



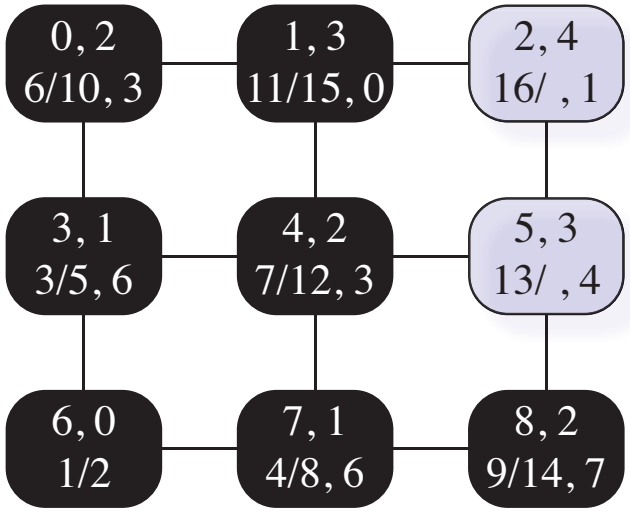

**(g)** Fifth exection of `while`.

os: 6 3 7 0 4 8
queue: 1 5



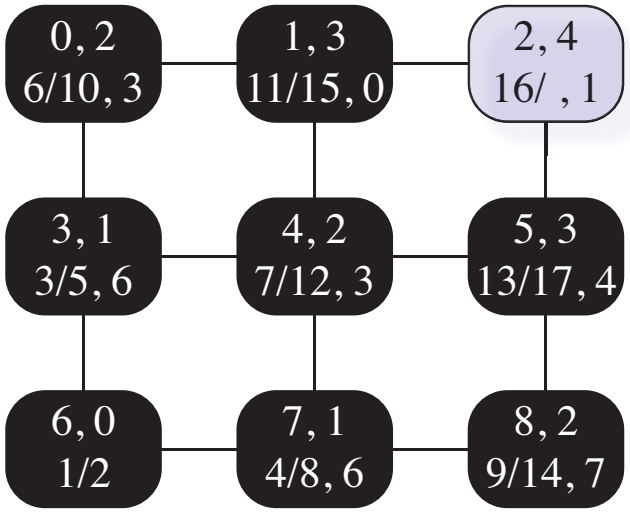

**(h)** Sixth exection of `while`.

os: 6 3 7 0 4 8 1
queue: 5 2



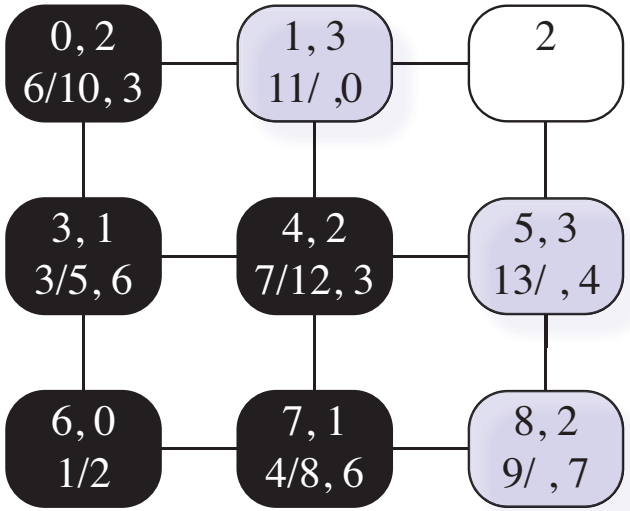

**(i)** Seventh exection of `while`.

os: 6 3 7 0 4 8 1 5
queue: 2



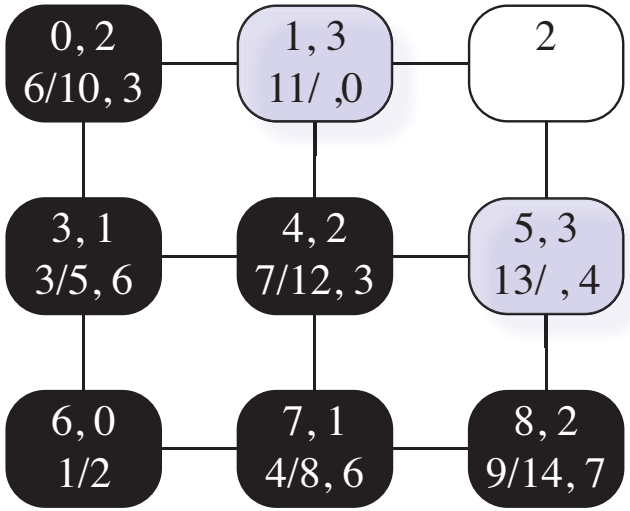

**(j)** Eighth exection of `while`.

os: 6 3 7 0 4
queue: 8 1 5



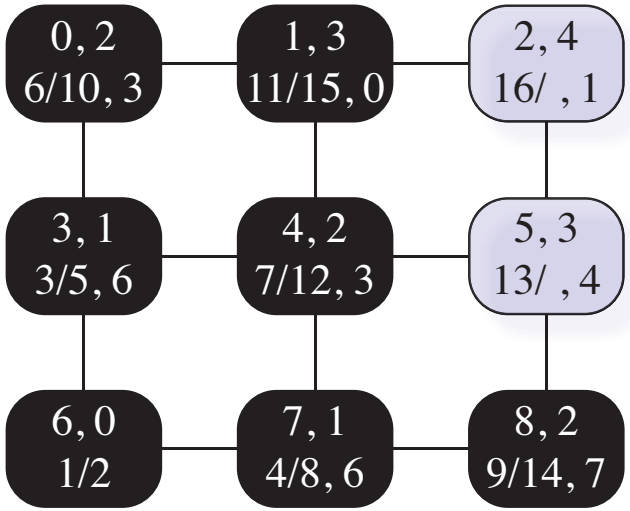

**(g)** Fifth exection of `while`.

os: 6 3 7 0 4 8
queue: 1 5



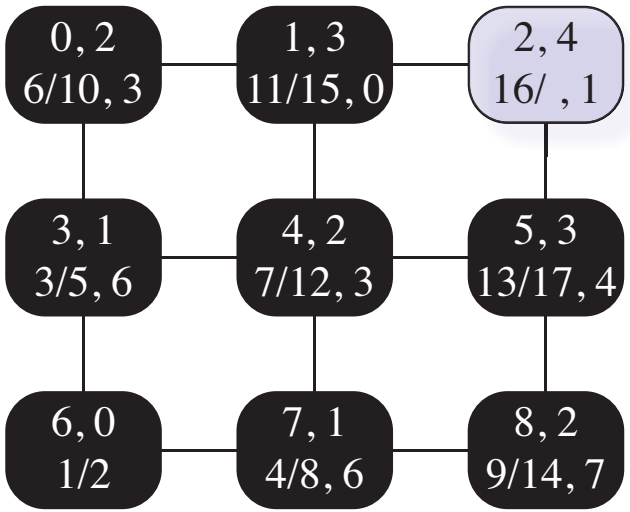

**(h)** Sixth exection of `while`.

os: 6 3 7 0 4 8 1
queue: 5 2



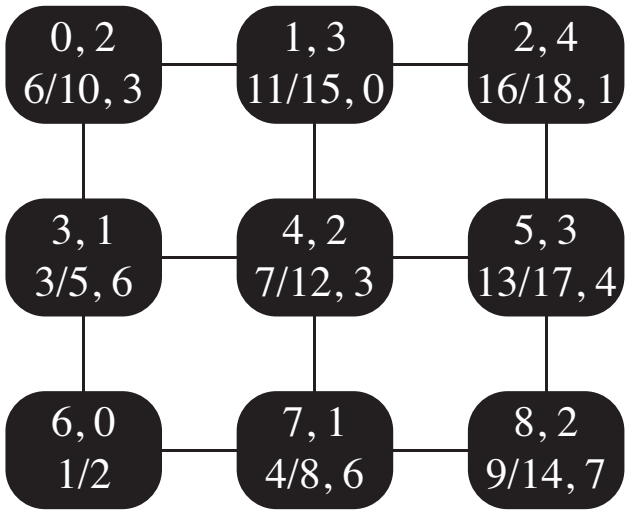

**(i)** Seventh exection of `while`.

os: 6 3 7 0 4 8 1 5
queue: 2




**(j)** Eighth exection of `while`.

os: 6 3 7 0 4 8 1 5 2
queue:




**(k)** Ninth exection of `while`.

# Review ListL Iterator Pattern

# Without iterator

```
// ========= toStream =========
template<class T>
void ListL<T>::toStream(ostream &os) const {
    os << "(";
    for (auto p = _head; p; p = p->_next) {
        if (p->_next) {
            os << p->_data << ", ";
        } else {
            os << p->_data;
        }
    }
    os << ")";
}
```

# With iterator

```
template<class T>
void ListL<T>::toStream4(ostream &os) const {
    ListLIterator<T> iter;
    iter.setIterListL(this);
    os << "(";
    for (iter.first(); !iter.isDone(); iter.next()) {
        if (iter.hasNext()) {
            os << iter.currentItem() << ", ";
        }
        else {
            os << iter.currentItem();
        }
    }
    os << ")";
}
```

# Depth-first search

```
// ========= Depth first search =========
void Graph::depthFirstSearch(int s, ostream &os) {
    initGraph();
    os << endl;
    dfs(s, os);
    writeVerticesPostDepth(os);
}

void Graph::dfs(int u, ostream &os) {
    cerr << "Graph::dfsVisit: Exercise for the student."
         << endl;
    throw -1;
}
```

(**a**) The C++ implementation.

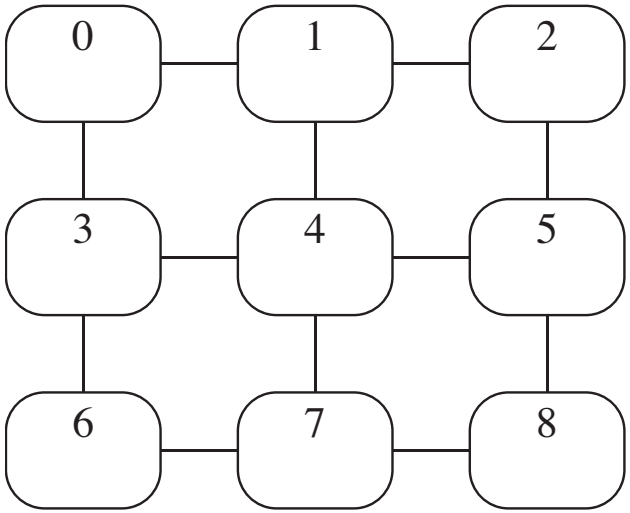
```
dfs(u, os)
    Stream u to os.
    Set the discovered time of u to ++_time.
    Set the color of u to GRAY.
    for (each vertex v adjacent to u)
        if (the color of v is WHITE)
            Set the predecessor of v to u.
            dfs(v, os)
    Set the color of u to BLACK.
    Set the finished time of u to ++_time.
```

(**b**) The algorithm for `dfs()`.

# Trace of depth-first search from vertex 6 with the following adjacency lists

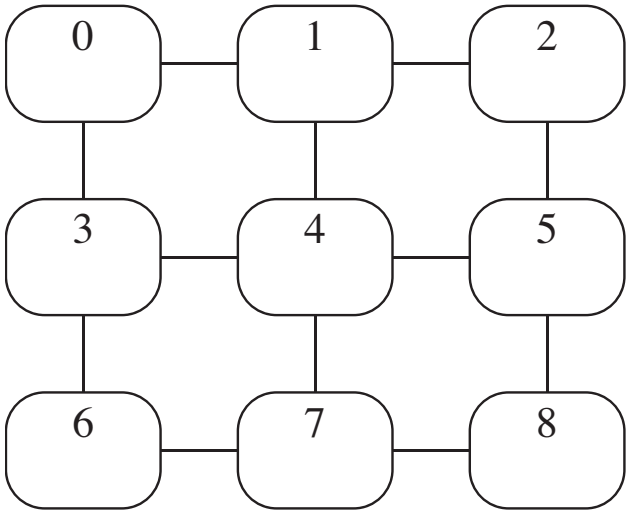
```
0: (1, 3)

1: (0, 2, 4)

2: (1, 5)

3: (4, 0, 6)

4: (5, 3, 7, 1)

5: (8, 4, 2)

6: (3, 7)

7: (4, 6, 8)

8: (5, 7)
```

os:



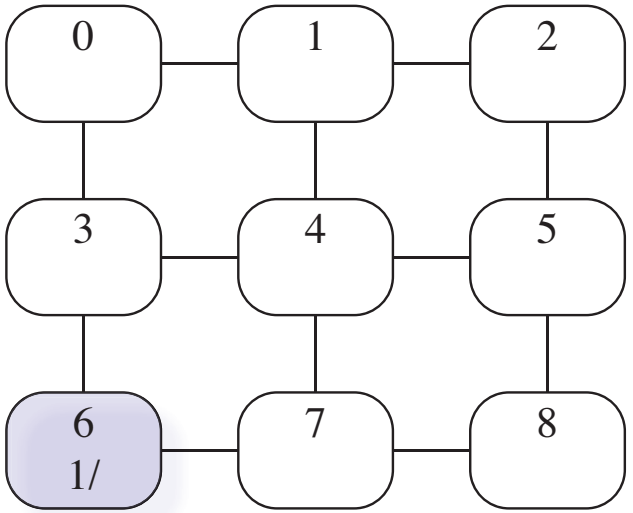

**(a)** Initial graph.

os:

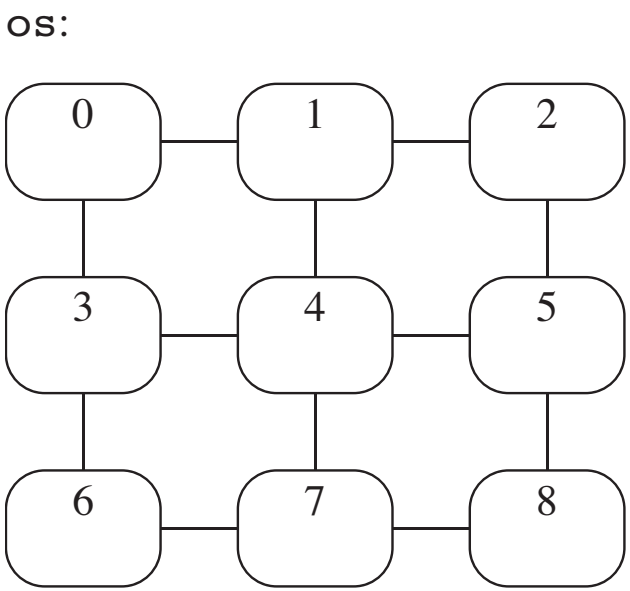



**(a)** Initial graph.

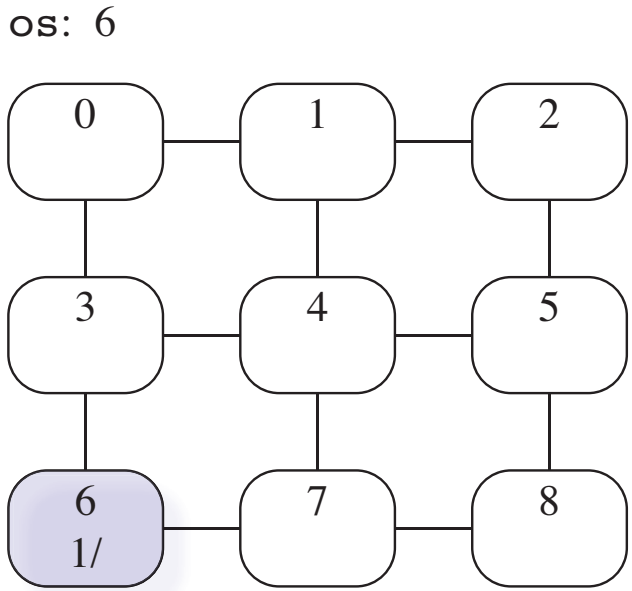
os: 6



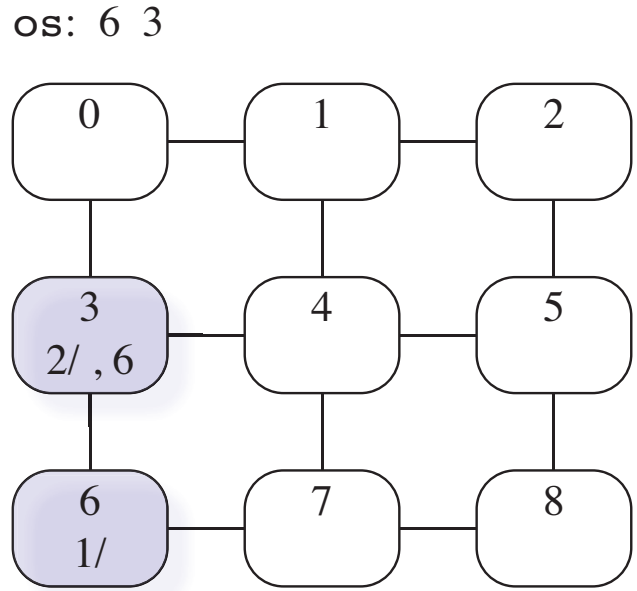

**(b)** Initialization of `for`.




**(a)** Initial graph.

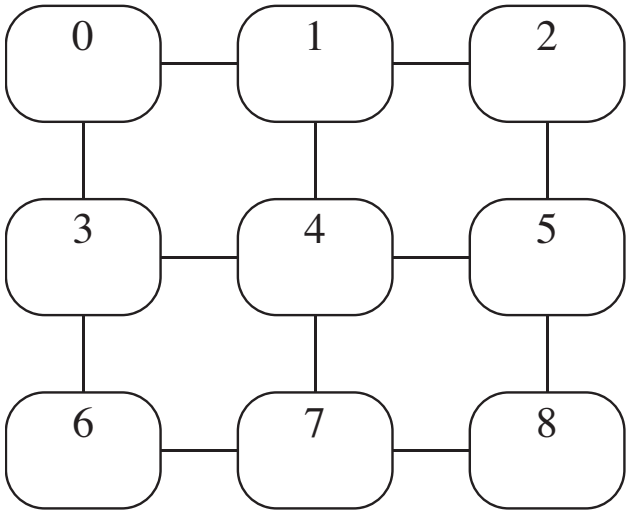



**(b)** Initialization of `for`.



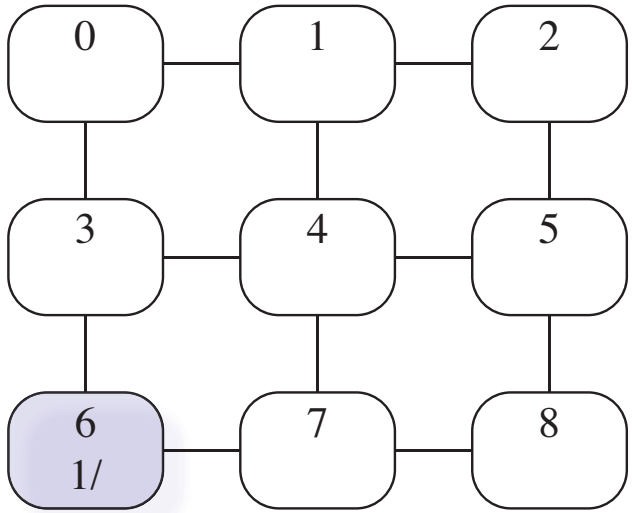

**(c)** Call `dfs(3)`.

os:



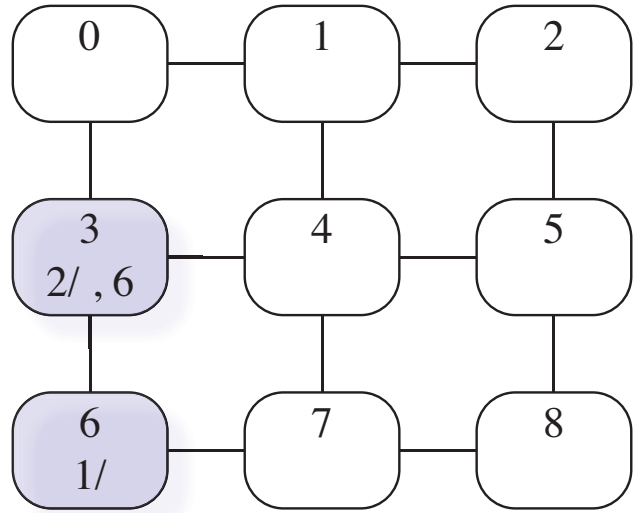

**(a)** Initial graph.

os: 6



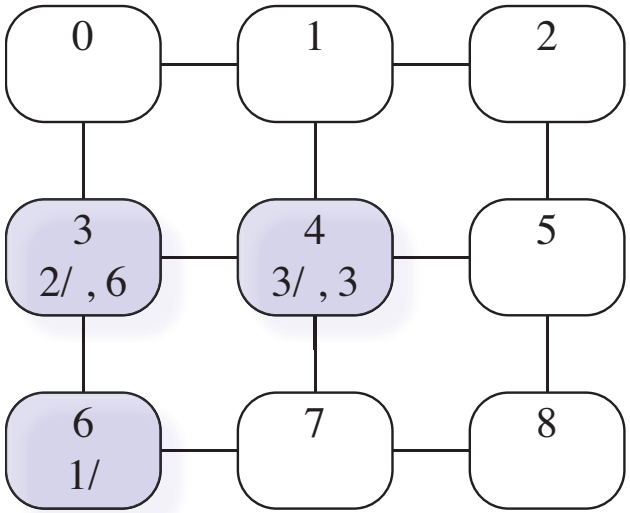

**(b)** Initialization of `for`.

os: 6 3



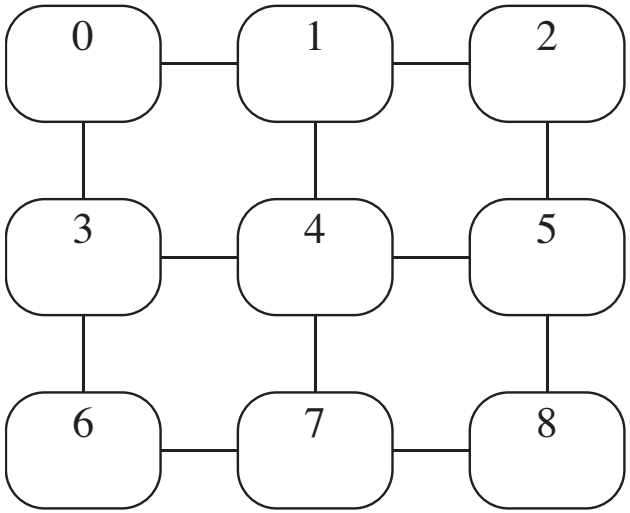

**(c)** Call `dfs(3)`.

os: 6 3 4



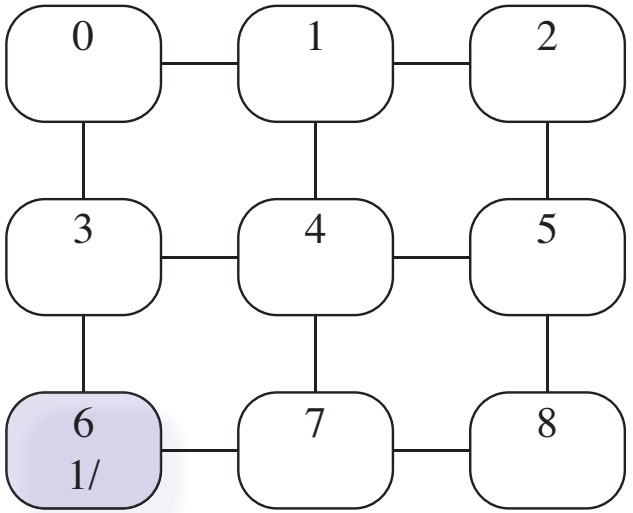

**(d)** Call `dfs(4)`.

os:



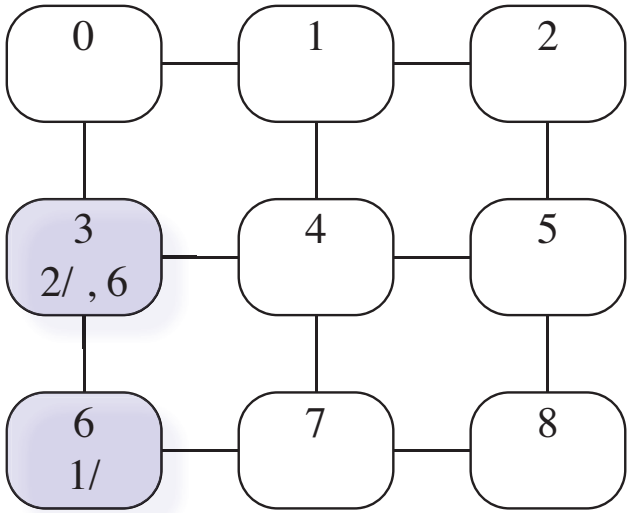

**(a)** Initial graph.

os: 6



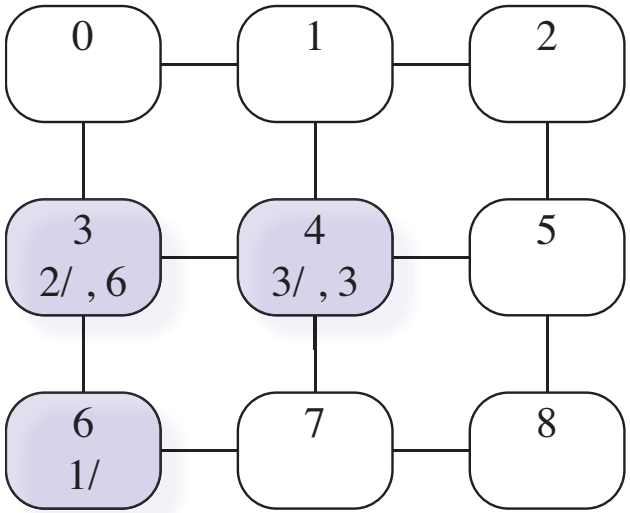

**(b)** Initialization of `for`.

os: 6 3



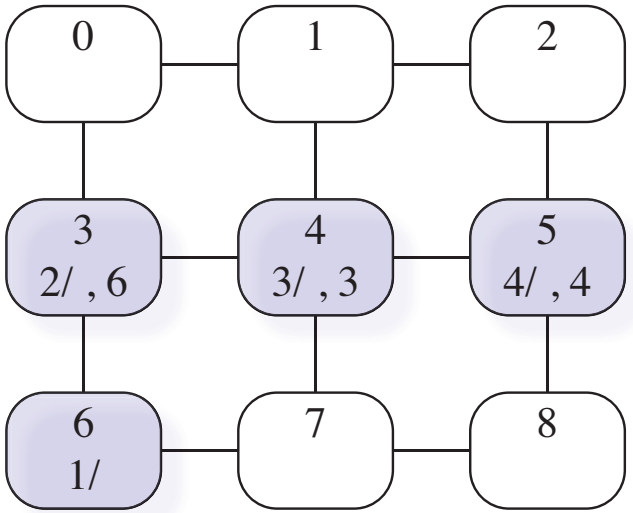

**(c)** Call `dfs(3)`.

os: 6 3 4



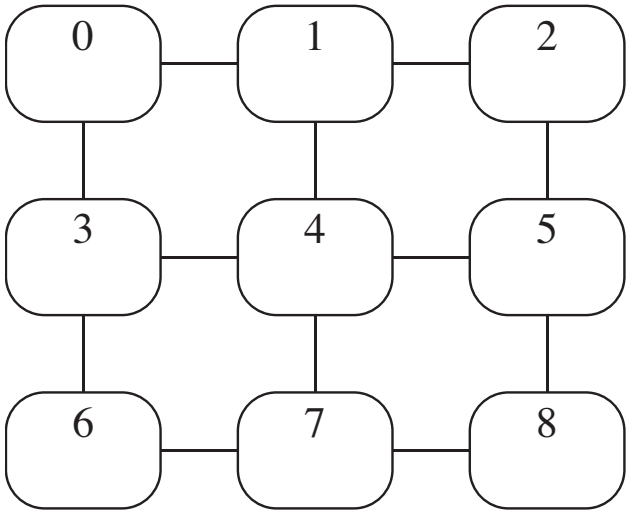

**(d)** Call `dfs(4)`.

os: 6 3 4 5



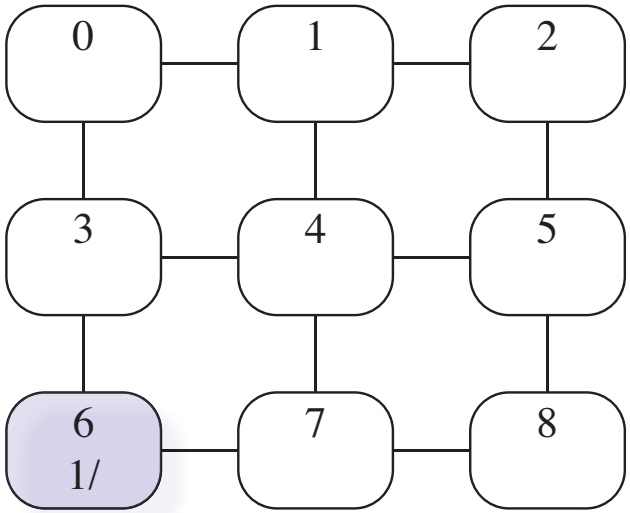

**(e)** Call `dfs(5)`.

os:



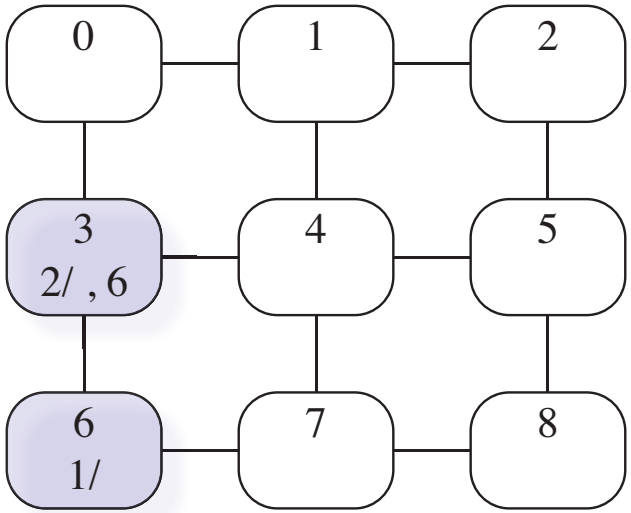

**(a)** Initial graph.

os: 6



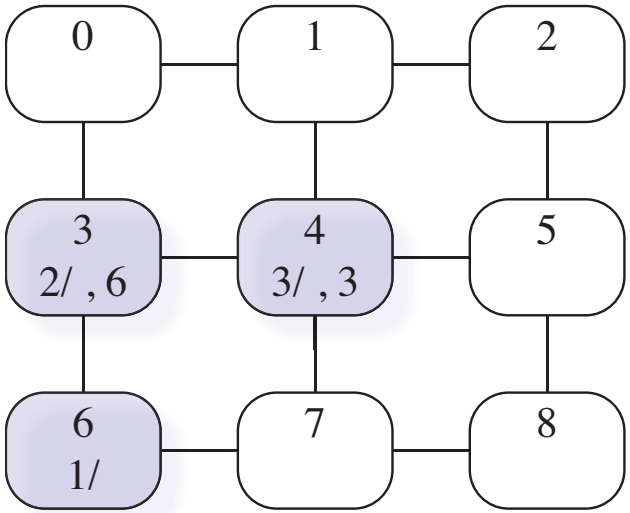

**(b)** Initialization of `for`.

os: 6 3



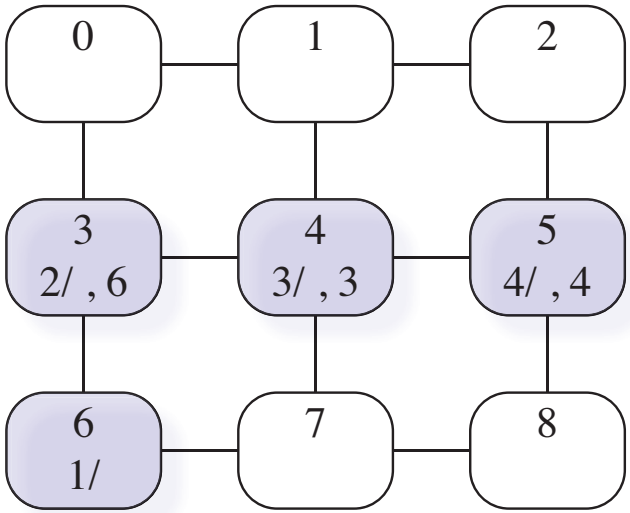

**(c)** Call `dfs(3)`.

os: 6 3 4



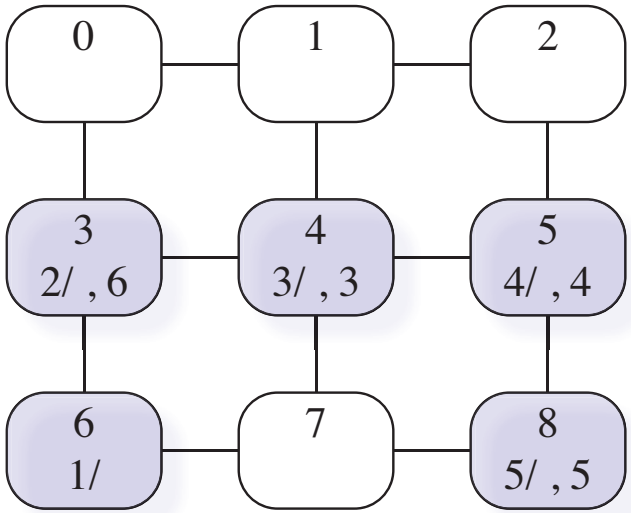

**(d)** Call `dfs(4)`.

os: 6 3 4 5



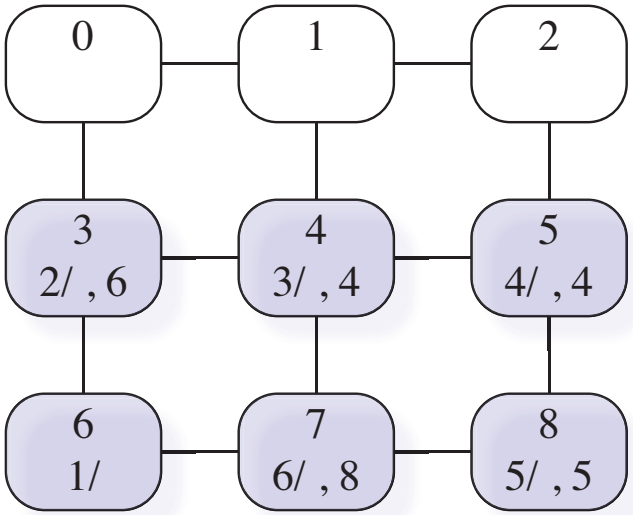

**(e)** Call `dfs(5)`.

os: 6 3 4 5 8



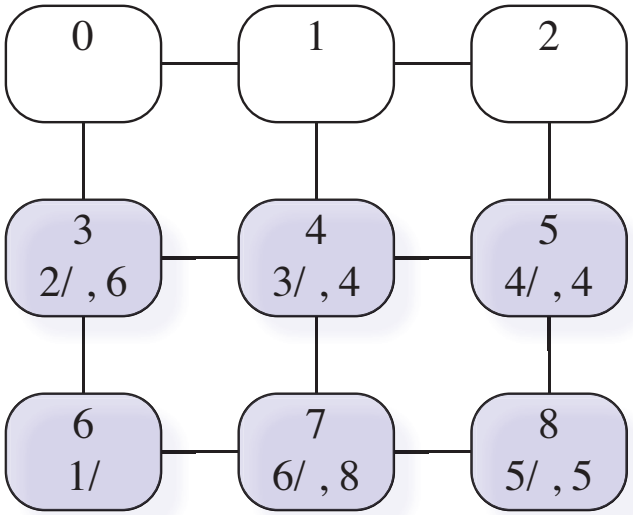

**(f)** Call `dfs(8)`.

`os: 6 3 4 5 8 7`



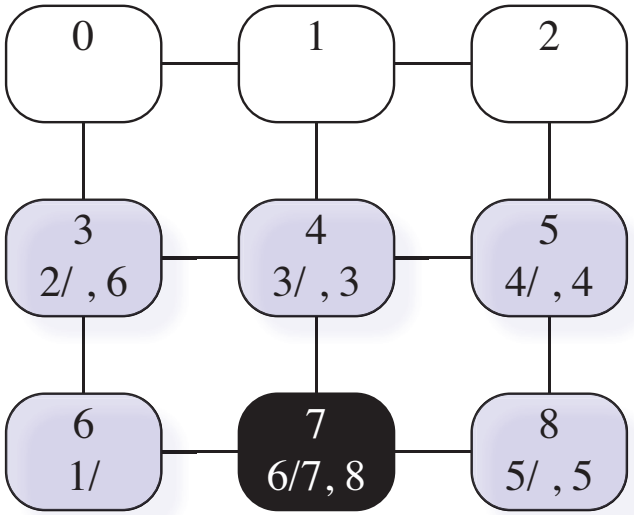

**(g)** Call `dfs(7)`.

os: 6 3 4 5 8 7



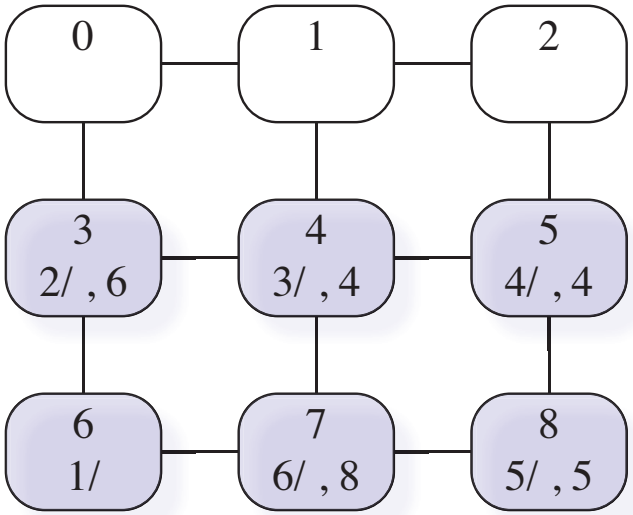

**(g)** Call `dfs(7)`.

os: 6 3 4 5 8 7



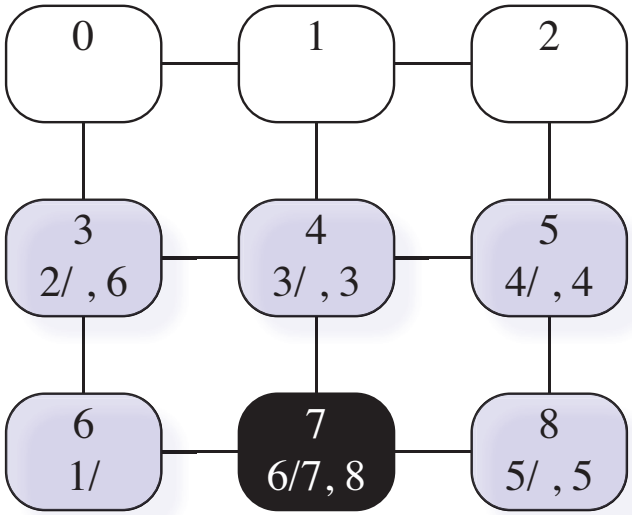

**(h)** Return from `dfs(7)`.

`os: 6 3 4 5 8 7`



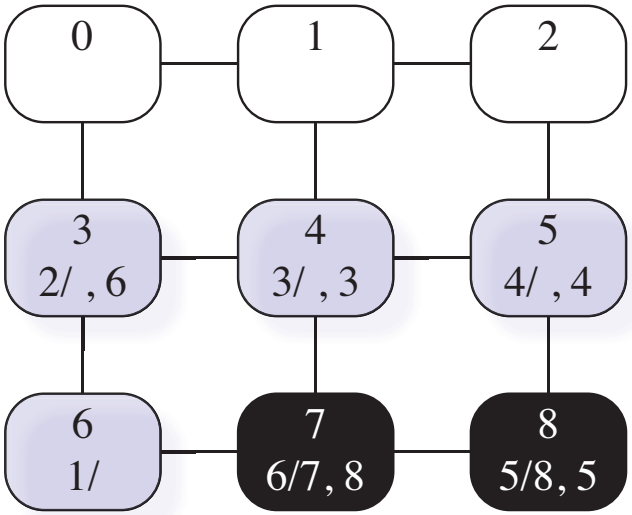

**(g)** Call `dfs(7)`.

`os: 6 3 4 5 8 7`



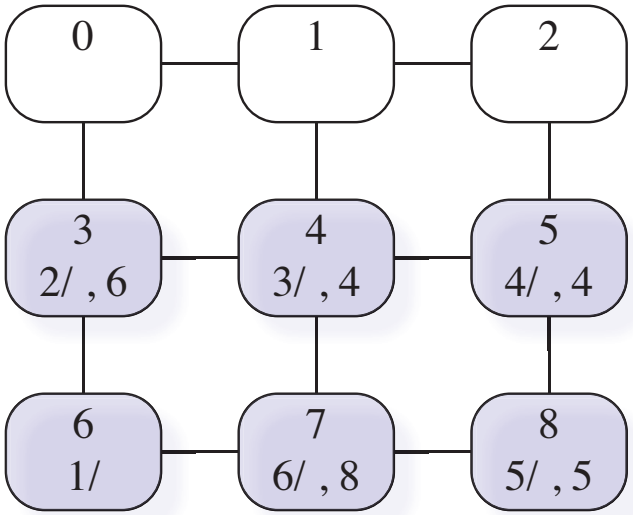

**(h)** Return from `dfs(7)`.

`os: 6 3 4 5 8 7`



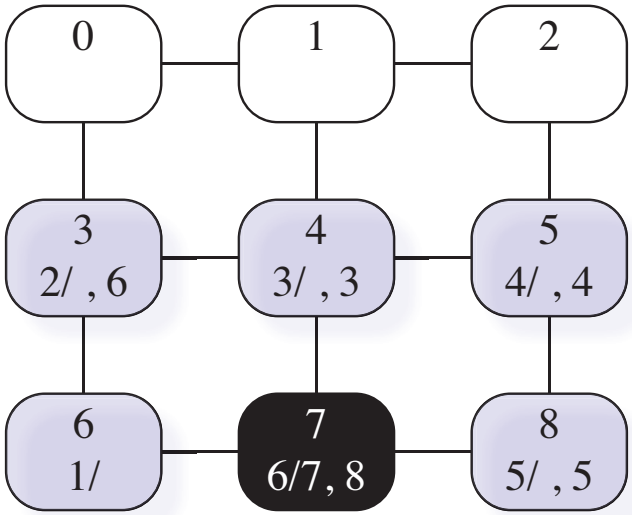

**(i)** Return from `dfs(8)`.

`os: 6 3 4 5 8 7`



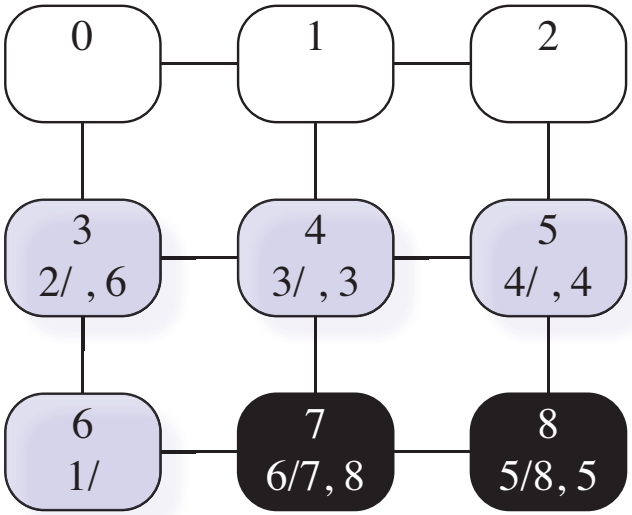

**(g)** Call `dfs(7)`.

`os: 6 3 4 5 8 7`



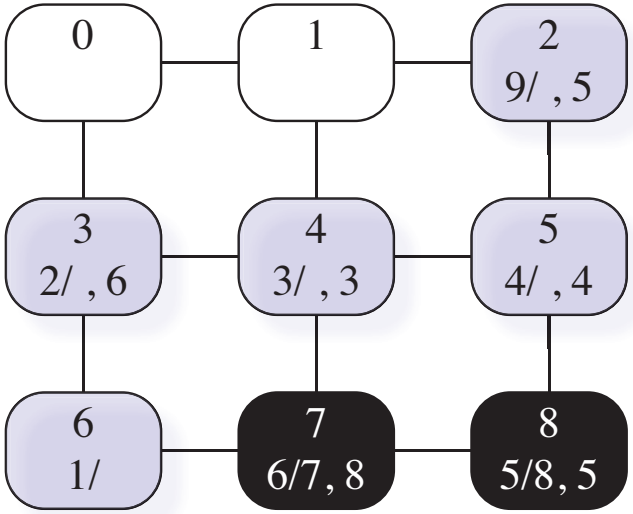

**(h)** Return from `dfs(7)`.

`os: 6 3 4 5 8 7`



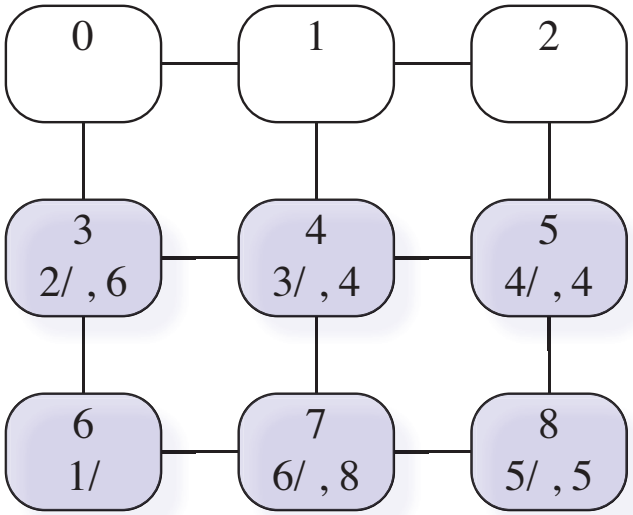

**(i)** Return from `dfs(8)`.

`os: 6 3 4 5 8 7 2`



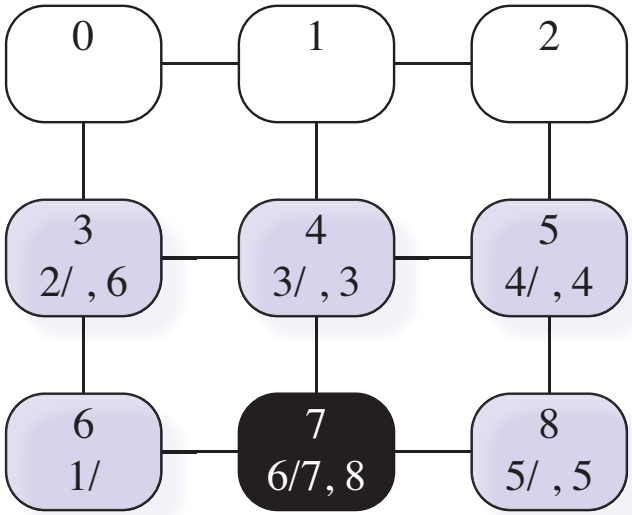

**(j)** Call `dfs(2)`.

os: 6 3 4 5 8 7



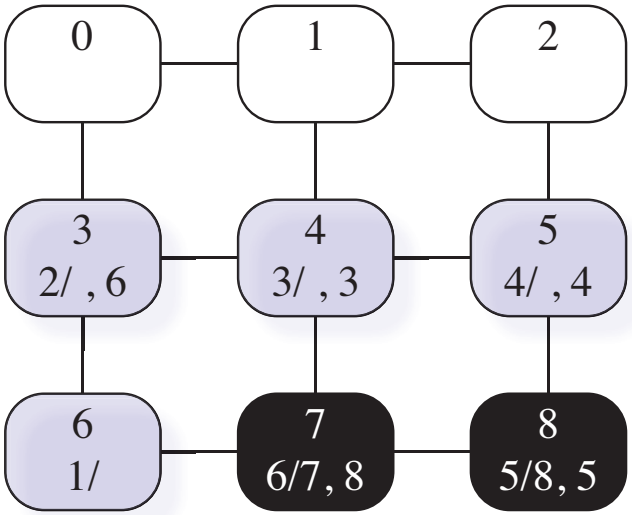

**(g)** Call `dfs(7)`.

os: 6 3 4 5 8 7



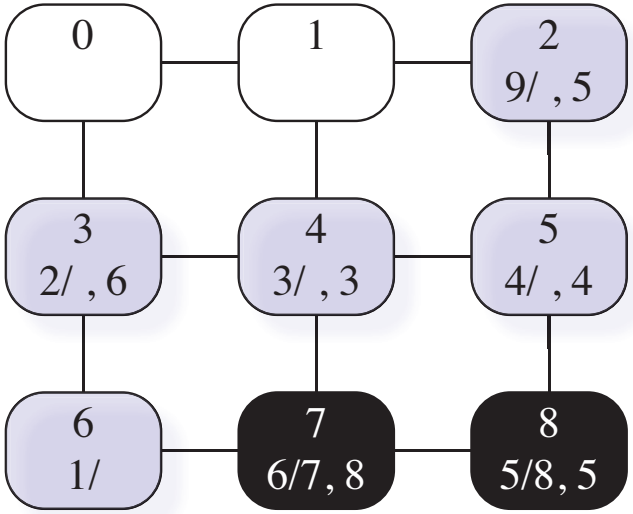

**(h)** Return from `dfs(7)`.

os: 6 3 4 5 8 7



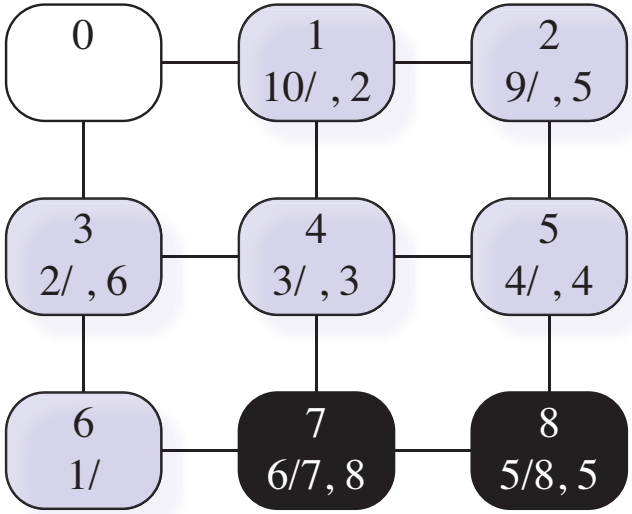

**(i)** Return from `dfs(8)`.

os: 6 3 4 5 8 7 2



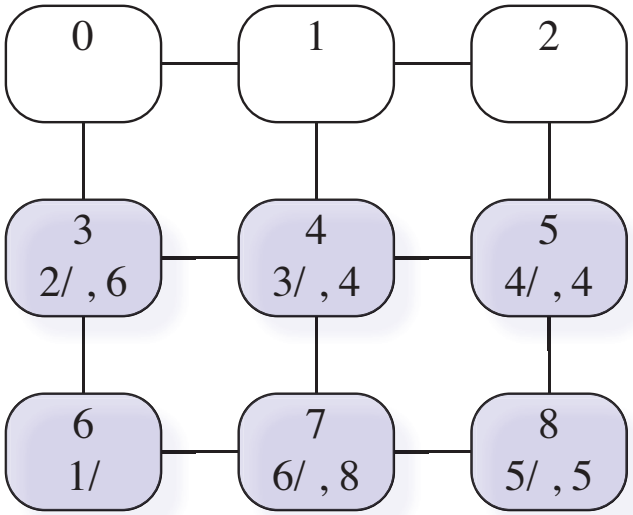

**(j)** Call `dfs(2)`.

os: 6 3 4 5 8 7 2 1



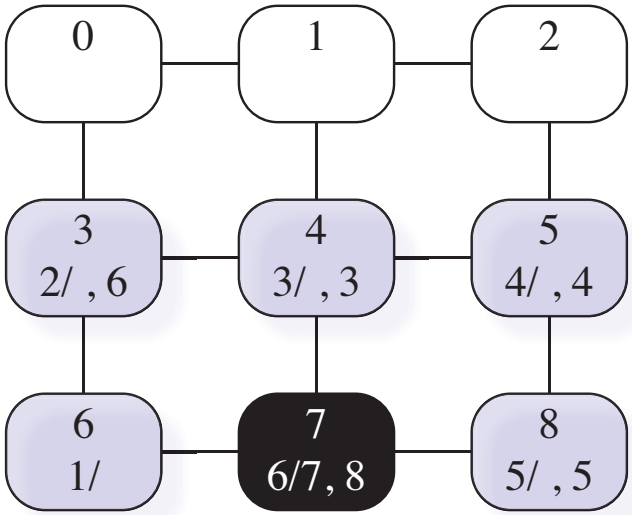

**(k)** Call `dfs(1)`.

os: 6 3 4 5 8 7



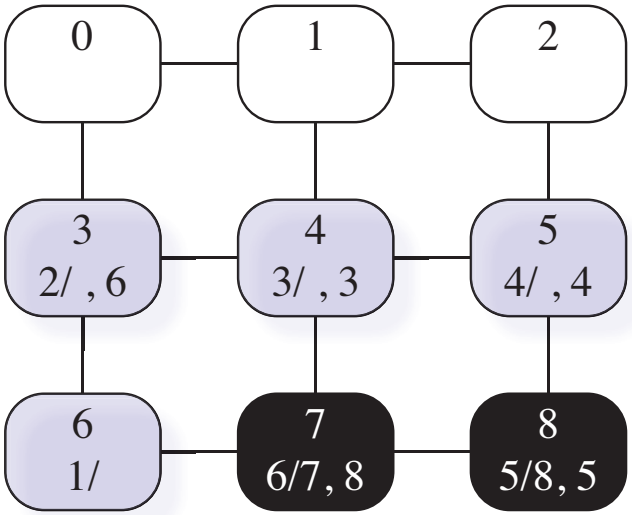

**(g)** Call `dfs(7)`.

os: 6 3 4 5 8 7



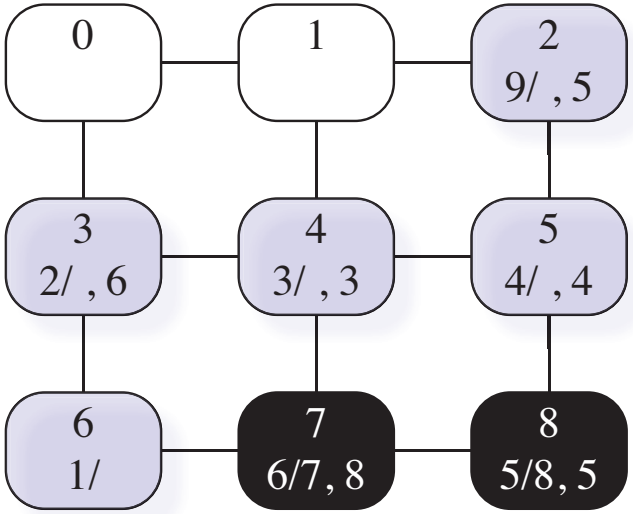

**(h)** Return from `dfs(7)`.

os: 6 3 4 5 8 7



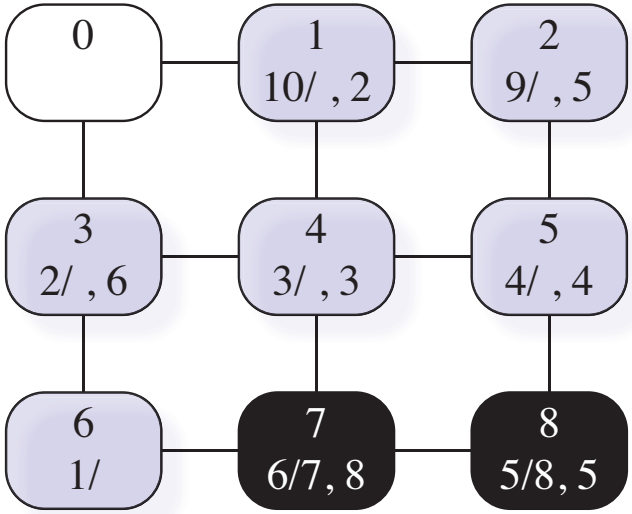

**(i)** Return from `dfs(8)`.

os: 6 3 4 5 8 7 2



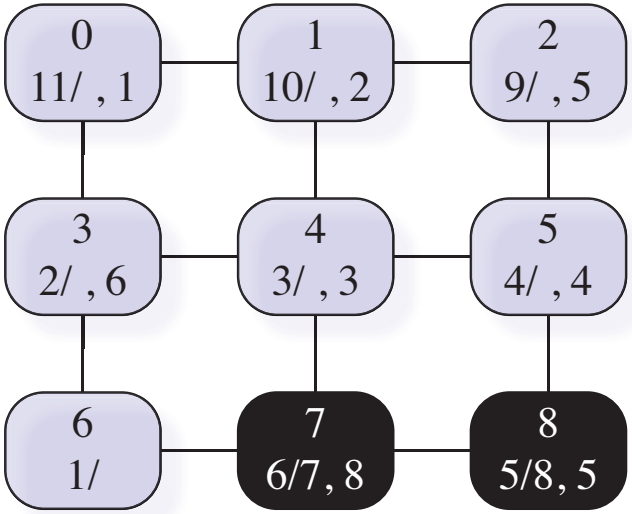

**(j)** Call `dfs(2)`.

os: 6 3 4 5 8 7 2 1



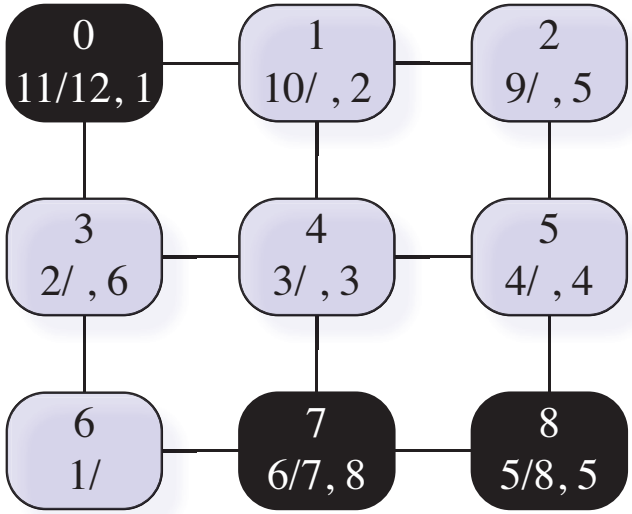

**(k)** Call `dfs(1)`.

os: 6 3 4 5 8 7 2 1 0



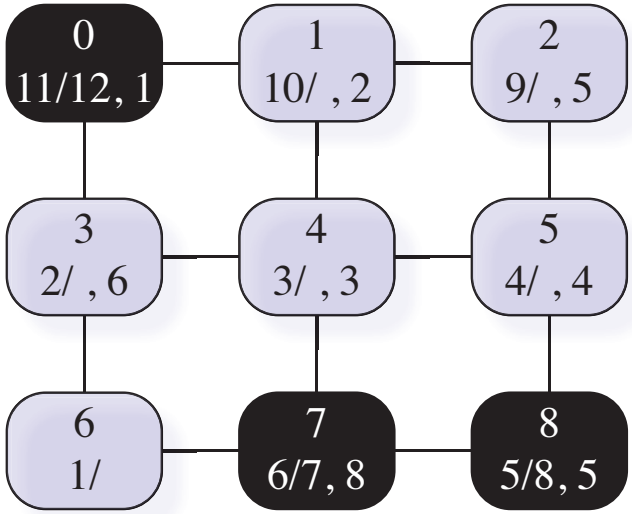

**(l)** Call `dfs(0)`.

`os:` 6 3 4 5 8 7 2 1 0



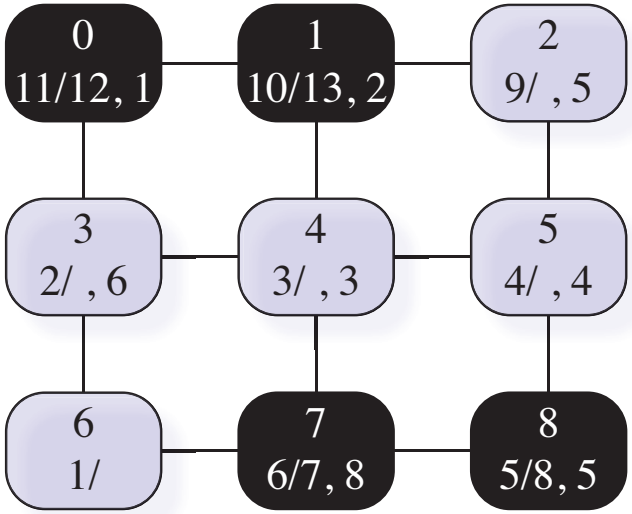

**(m)** Return from `dfs(0)`.

`os: 6 3 4 5 8 7 2 1 0`



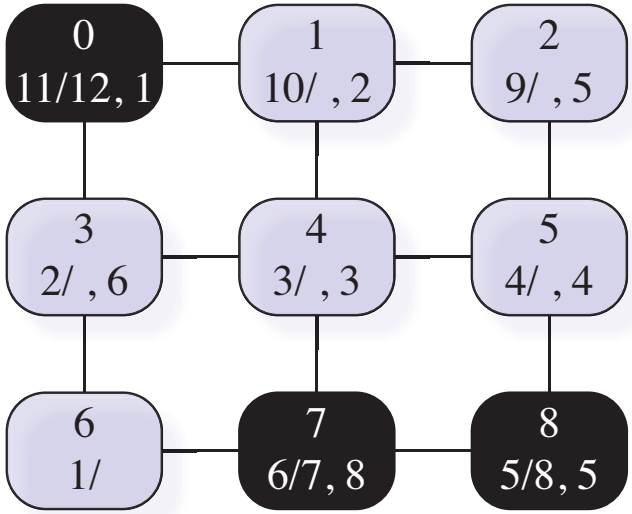

**(m)** Return from `dfs(0)`.

`os: 6 3 4 5 8 7 2 1 0`



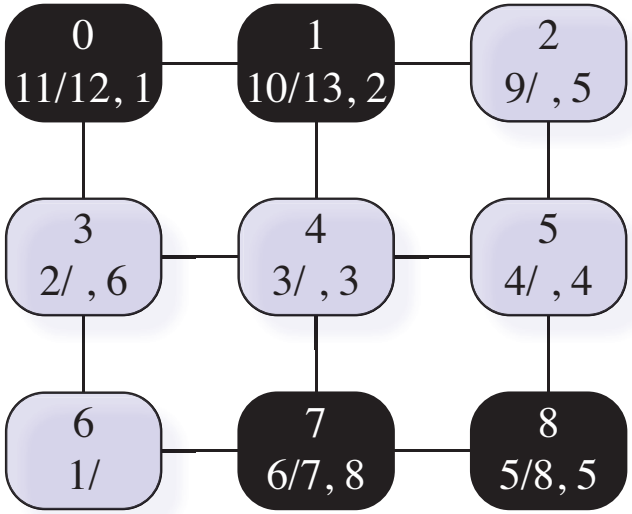

**(n)** Return from `dfs(1)`.

os: 6 3 4 5 8 7 2 1 0



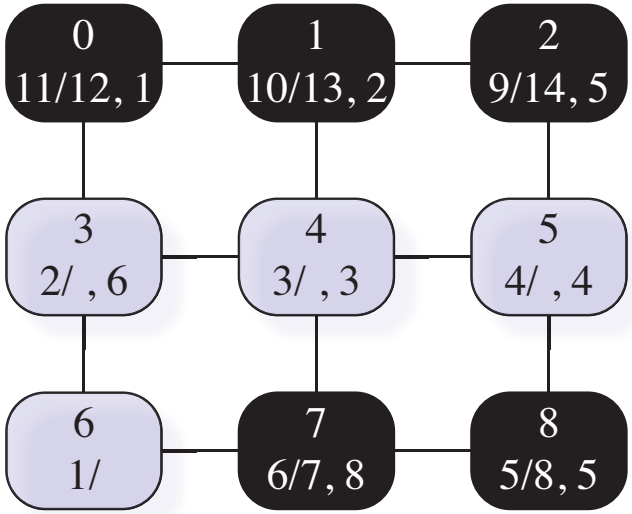

**(m)** Return from `dfs(0)`.

os: 6 3 4 5 8 7 2 1 0



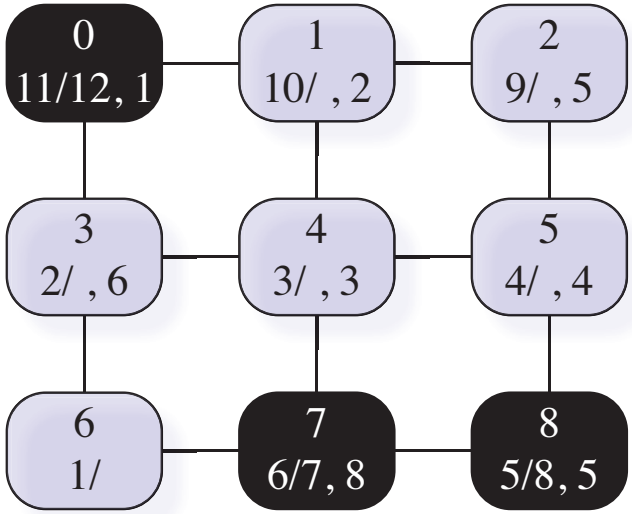

**(n)** Return from `dfs(1)`.

os: 6 3 4 5 8 7 2 1 0



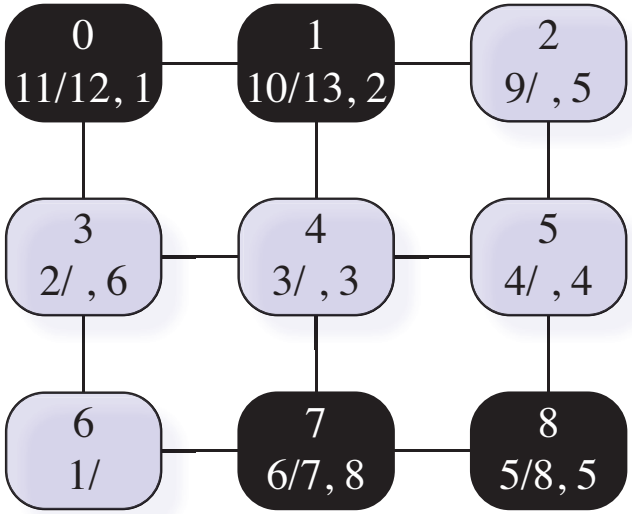

**(o)** Return from `dfs(2)`.

os: 6 3 4 5 8 7 2 1 0



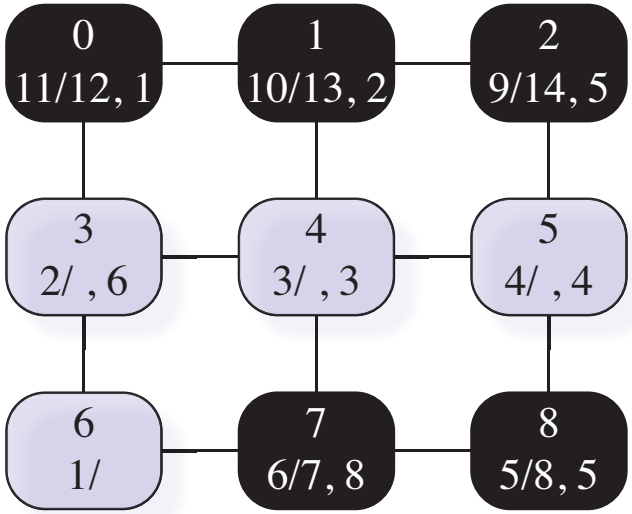

**(m)** Return from `dfs(0)`.

os: 6 3 4 5 8 7 2 1 0



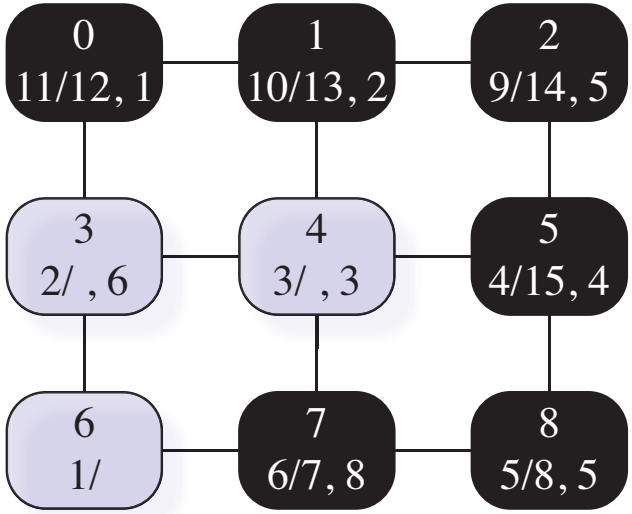

**(n)** Return from `dfs(1)`.

os: 6 3 4 5 8 7 2 1 0



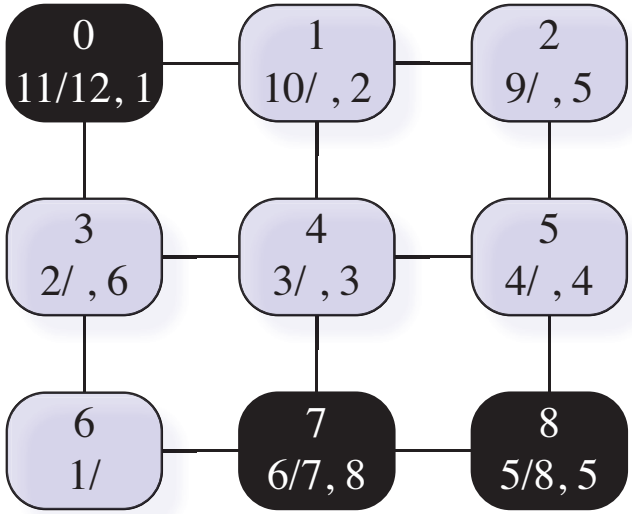

**(o)** Return from `dfs(2)`.

os: 6 3 4 5 8 7 2 1 0



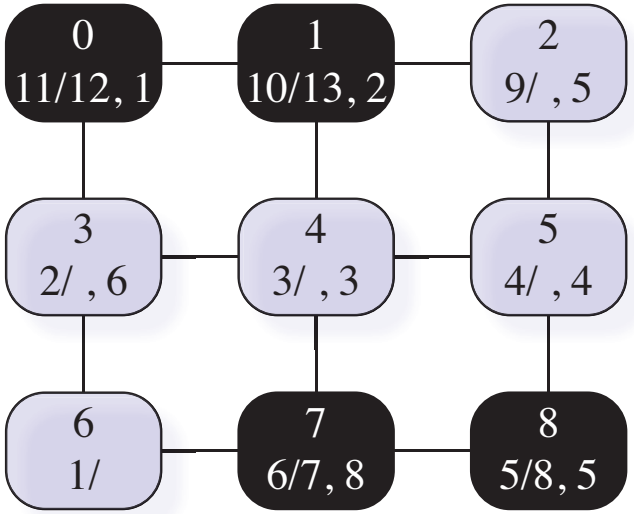

**(p)** Return from `dfs(5)`.

os: 6 3 4 5 8 7 2 1 0



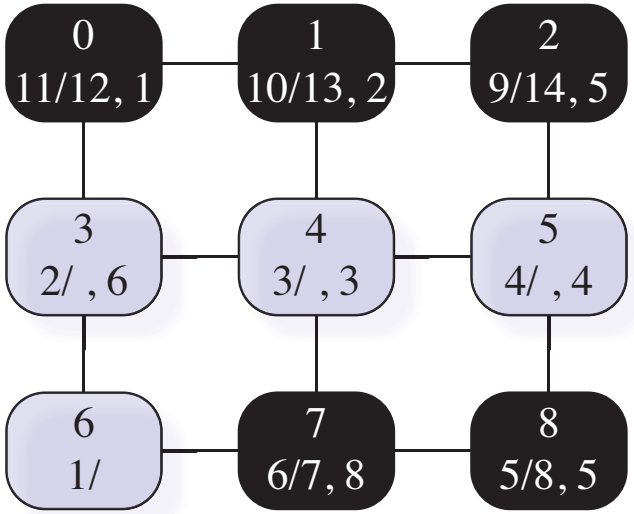

**(m)** Return from `dfs(0)`.

os: 6 3 4 5 8 7 2 1 0



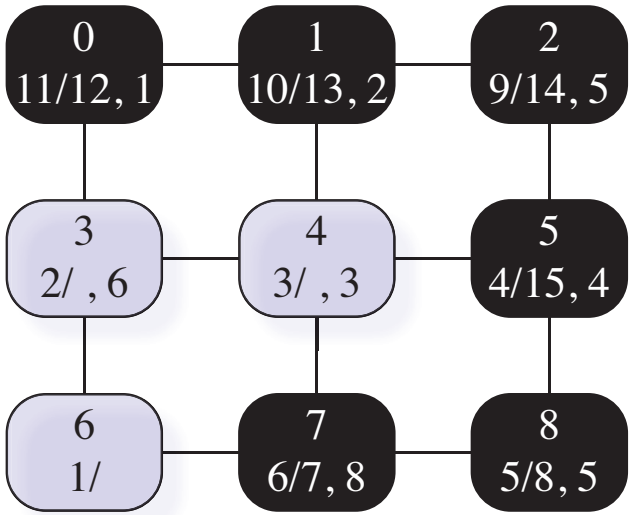

**(n)** Return from `dfs(1)`.

os: 6 3 4 5 8 7 2 1 0



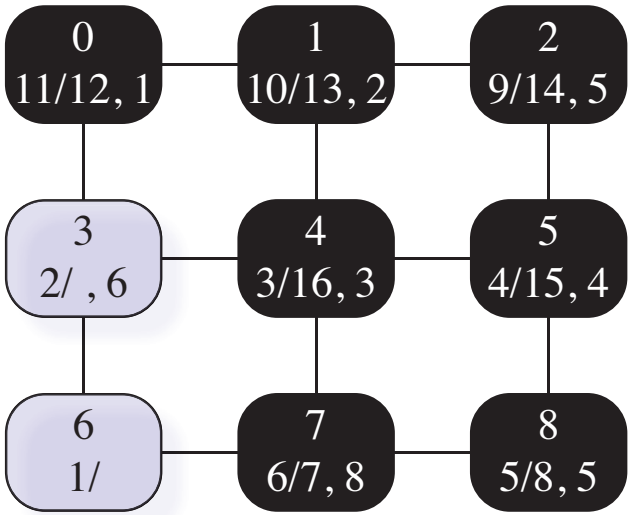

**(o)** Return from `dfs(2)`.

os: 6 3 4 5 8 7 2 1 0



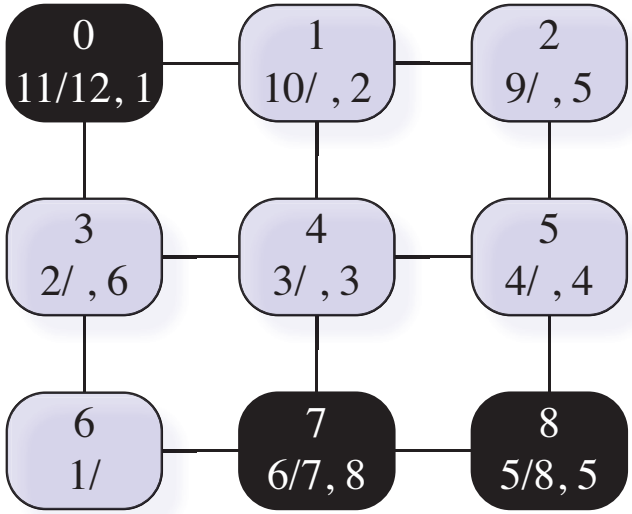

**(p)** Return from `dfs(5)`.

os: 6 3 4 5 8 7 2 1 0



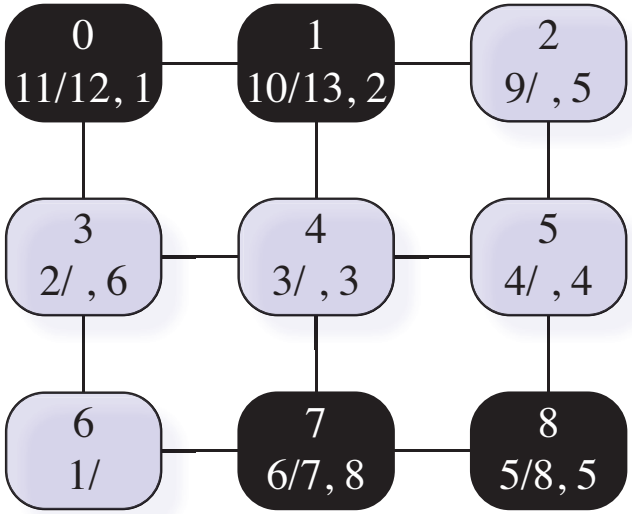

**(q)** Return from `dfs(4)`.

os: 6 3 4 5 8 7 2 1 0



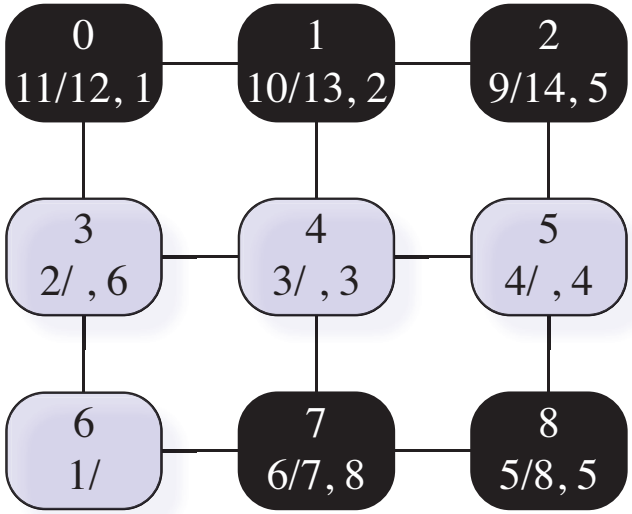

**(m)** Return from `dfs(0)`.

os: 6 3 4 5 8 7 2 1 0



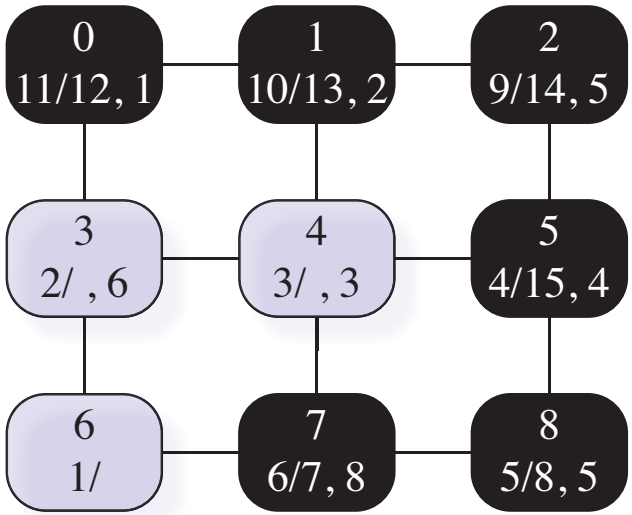

**(n)** Return from `dfs(1)`.

os: 6 3 4 5 8 7 2 1 0



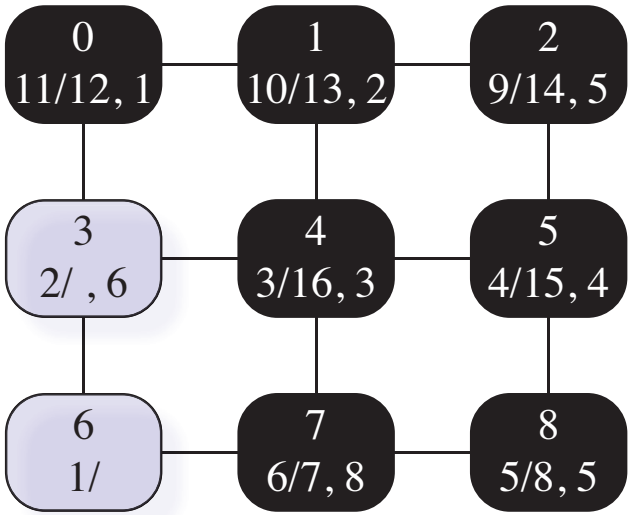

**(o)** Return from `dfs(2)`.

os: 6 3 4 5 8 7 2 1 0



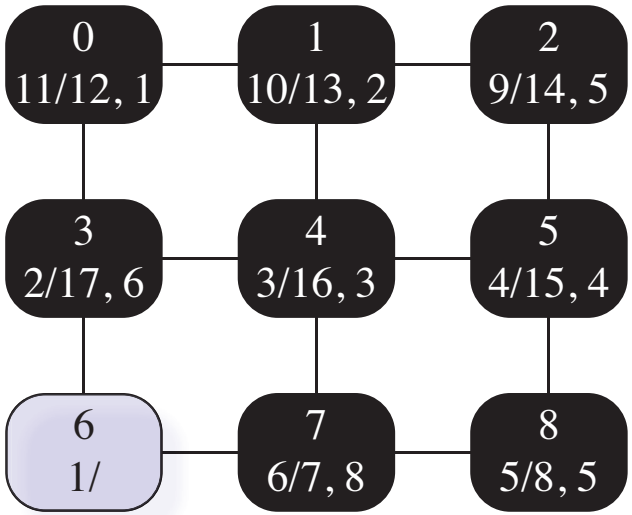

**(p)** Return from `dfs(5)`.

os: 6 3 4 5 8 7 2 1 0



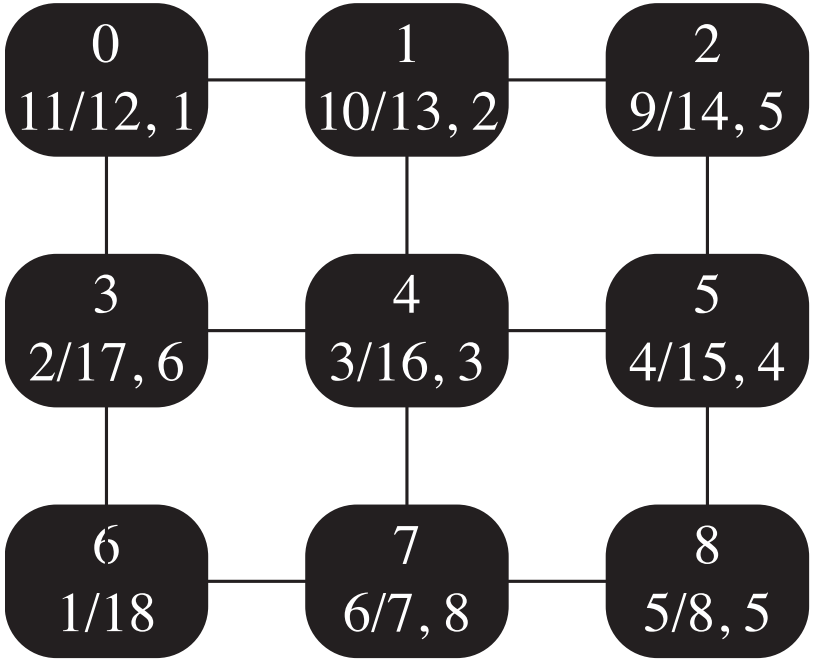

**(q)** Return from `dfs(4)`.

os: 6 3 4 5 8 7 2 1 0




**(r)** Return from `dfs(3)`.

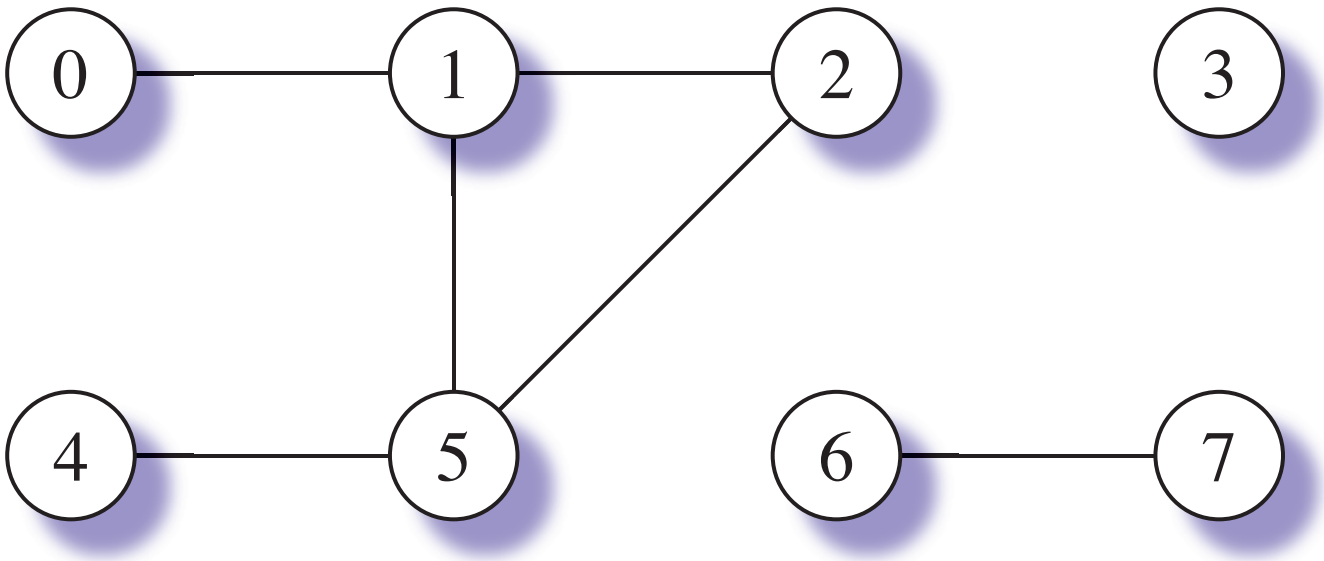
`os:` 6 3 4 5 8 7 2 1 0




(s) Return from `dfs(6)`.

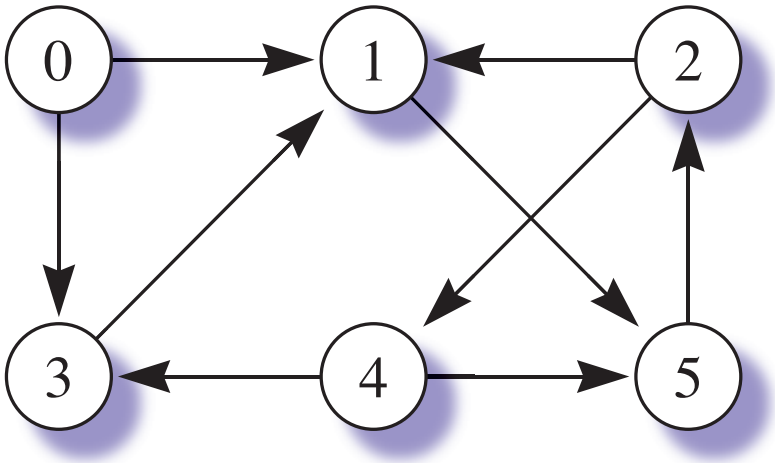
# Write components

# Figure 11.16



```
writeComponents(os)
    Set numComponents to 0.
    Initialize _graph.
    for (each vertex u of _graph)
        if (the color of u is WHITE)
            Stream "Connected component:" to os.
            Search _graph without initialization starting at vertex u.
            Stream endl to os.
            Increment numComponents.
    Stream numComponents to os.
```

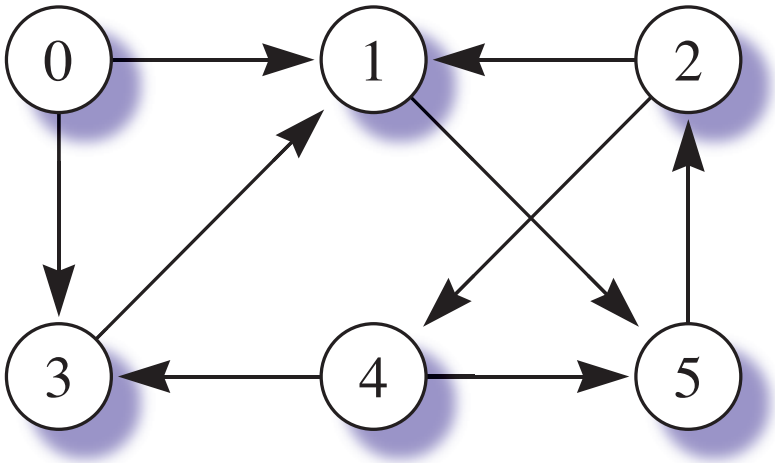
# Write path




**(a)** A directed graph.

From 0: 0 3 1 5 2 4
From 1: 1 5 2 4 3
From 2: 2 1 4 5 3
From 3: 3 1 5 2 4
From 4: 4 3 5 1 2
From 5: 5 2 1 4 3

**(b)** Breadth-first searches.




**(a)** A directed graph.

From 0: 0 3 1 5 2 4
From 1: 1 5 2 4 3
From 2: 2 1 4 5 3
From 3: 3 1 5 2 4
From 4: 4 3 5 1 2
From 5: 5 2 1 4 3

**(b)** Breadth-first searches.

Predecessor of vertex 0: –1
Predecessor of vertex 1: 0
Predecessor of vertex 2: 5
Predecessor of vertex 3: 0
Predecessor of vertex 4: 2
Predecessor of vertex 5: 1

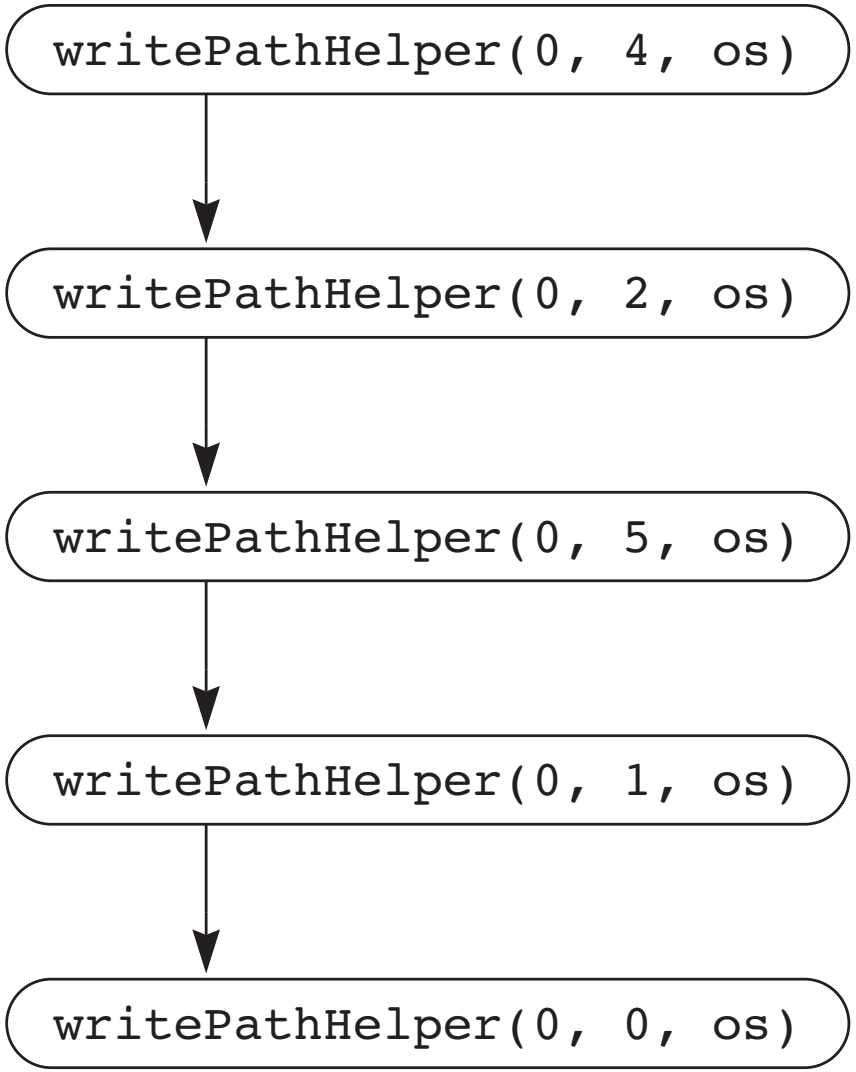
**(c)** The predecessor vertices for the search from vertex 0.

```
// ========= writePath =========
void Graph::writePath(int from, int to, ostream &os) {
    if ((from < 0) || (_numVertices <= from) ||
        (to < 0) || (_numVertices <= to)) {
      ...
    initGraph();
    os << "\nBreadth-first search from " << from << ": ";
    bfs(from, os);
    os << "\nPath from " << from << " to " << to << " is: ";
    writePathHelper(from, to, os);
    if (_vertex[to].distance != -1) {
        os << "\nDistance = " << _vertex[to].distance << endl;
    }
}

void Graph::writePathHelper(int from, int to, ostream &os) {
    cerr << "writePathHelper: Exercise for the student." << endl;
    throw -1;
}
```

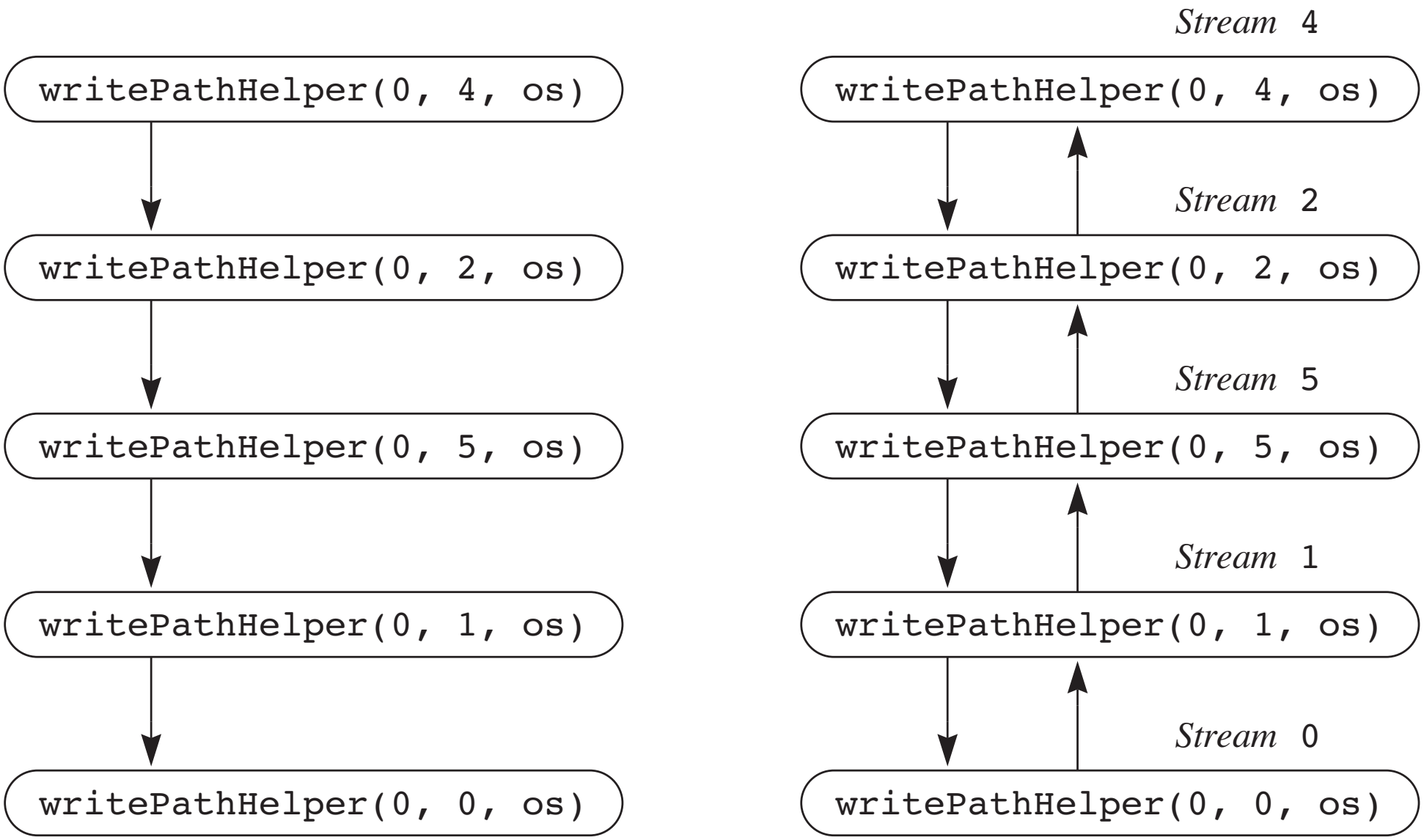
(**a**) The C++ implementation.

```
writePathHelper(from, to, os)
    if (to == from)
        Stream from to os.
    else if (the predecessor of to equals -1)
        Stream "No path exists" to os.
    else
        writePathHelper(from, the predecessor of to, os)
        Stream to to os.
```

(**b**) The algorithm for `writePathHelper()`.




**(a)** Down the call chain.




**(a)** Down the call chain.

**(b)** Streaming to `os` up the call chain.