Graphs
(a) A graph.

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

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

\[
\begin{align*}
\text{graph} \quad & [0,] \quad [1,] \quad [2,] \quad [3,] \quad [4,] \quad [5,] \\
[0,] & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\
[1,] & \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \\
[2,] & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
[3,] & \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \\
[4,] & \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \\
[5,] & \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}
\end{align*}
\]
(a) A directed graph.

(b) The adjacency-list implementation of the digraph.
(a) A directed graph.

Figure 11.2

(c) The adjacency-matrix implementation of the digraph.
Demo insertEdge for graph and directed graph
enum ColorType {WHITE, GRAY, BLACK};
typedef struct {
    ColorType color;
    int discovered;
    int finished;
    int distance;
    int predecessor;
} graphVertex;

class Graph {
private:
    int _numVertices;
    bool _isDigraph;
    int _time;
    ArrayT<graphVertex> _vertex;
    ArrayT<ListL<int>> _graph;
    ArrayT<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.

    ~Graph();
    // Post: The graph is deallocated.

    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) :
    _isDigraph(isDigraph),
    _numVertices(numVert),
    _vertex(numVert),
    _graph(numVert),
    _graphIter(numVert) {
    for (int i = 0; i < numVert; i++) {
        _graphIter[i].setIterListL(&_graph[i]);
    }
}
```
void Graph::insertEdge(int from, int to) {
    if (((from < 0) || (_numVertices <= from) ||
         (to < 0) || (_numVertices <= to)) {  // insertEdge precondition violated: from or to out of range.
        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.
// ========== 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
```cpp
// ========= 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().
Figure 11.10  
(a) Initial graph.

(b) Initialization of while.

(c) First execution of while.

(d) Second execution of while.

(e) Third execution of while.

(f) Fourth execution of while.
Figure 11.11 (continued) A trace of a breadth-first search of a graph.

(g) Fifth execution of while.

(h) Sixth execution of while.

(i) Seventh execution of while.

(j) Eighth execution of while.

(k) Ninth execution of while.
Review ListL Iterator Pattern
Without iterator

```cpp
// ======== toStream ========
template<class T>
void ListL<T>::toStream(ostream &os) const {
    os << "(");
    for (LNode<T> *p = _head; p != nullptr; p = p->next) {
        if (p->next != nullptr) {
            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;
    << 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().
Figure 11.13

At race of a depth-first search of a graph. The trace continues in the next figure.

(a) Initial graph.
(b) Initialization of for.
(c) Call dfs(3).
(d) Call dfs(4).
(e) Call dfs(5).
(f) Call dfs(8).
(g) Call $\text{dfs}(7)$.

(h) Return from $\text{dfs}(7)$.

(i) Return from $\text{dfs}(8)$.

(j) Call $\text{dfs}(2)$.

(k) Call $\text{dfs}(1)$.

(l) Call $\text{dfs}(0)$. 
Figure 11.14 (continued) A trace of a depth-first search of a graph. The trace continues in the next figure.
(s) Return from dfs(6).
Write components
Figure 11.16

A graph with three connected components.
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
The directed graph from Figure 11.2 and the breadth-first searches from each of its vertices.

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)) {
        cerr << "minimumDistance precondition violated: from or to out of range.
    from == " << from << " to == " << to << endl;
    throw -1;
    }
    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().
writePathHelper(0, 4, os)

writePathHelper(0, 2, os)

writePathHelper(0, 5, os)

writePathHelper(0, 1, os)

writePathHelper(0, 0, os)

(a) Down the call chain.

writePathHelper(0, 4, os)

writePathHelper(0, 2, os)

writePathHelper(0, 5, os)

writePathHelper(0, 1, os)

writePathHelper(0, 0, os)

(b) Streaming to os up the call chain.