CHAPTER 7

C++ Graphs

LESSON 7.1

Introduction to Graphs

In previous chapters, you learned various ways to represent and manipulate data. This chapter discusses how to implement and manipulate graphs, which have numerous applications in computer science.

In 1736, the following problem was posed. In the town of Ko¨nigsberg (now called Kaliningrad), the river Pregel (Pregolya) flows around the island Kneiphof and then divides into two. See Figure 7.1.1.

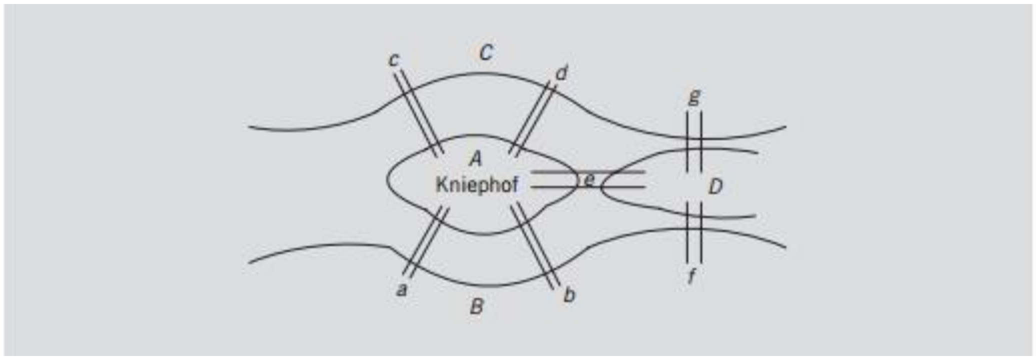

Figure 7.1.1 The Konigsberg bridge problem

The river has four land areas (A, B, C, D), as shown in the figure. These land areas are connected using seven bridges, as shown in Figure 7.1.1. The bridges are labeled a, b, c, d, e, f, and g. The Ko¨nigsberg bridge problem is as follows: Starting at one land area, is it possible to walk across all the bridges exactly once and return to the starting land area?

In 1736, Euler represented the Ko¨nigsberg bridge problem as a graph, as shown in Figure 7.1.2, and answered the question in the negative. This marked (as recorded) the birth of graph theory

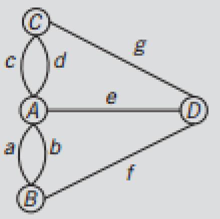

Figure 7.1.2 Graph representation of the Ko¨ nigsberg bridge problem

Over the past 200 years, graph theory has been applied to a variety of applications. Graphs are used to model electrical circuits, chemical compounds, highway maps, and so on. They are also used in the analysis of electrical circuits, finding the shortest route, project planning, linguistics, genetics, social science, and so forth. In this chapter, you learn about graphs and their applications in computer science.

Graphs Definition and Notations

To facilitate and simplify our discussion, we borrow a few definitions and terminology from set theory. Let X be a set. If a is an element of X, we write a ∈ X. (The symbol ''∈'' means ''belongs to.'') A set Y is called a subset of X if every element of Y is also an element of X. If Y is a subset of X, we write Y ⊆ X. (The symbol ''⊆'' means ''is a subset of.'') The intersection of sets A and B, written A ∩ B, is the set of all elements that are in A and B; that is, A ∩ B = {x | x ∈ A and x ∈ B}. (The symbol ''∩'' means ''intersection.'') The union of sets A and B, written A ∪ B, is the set of all elements that are in A or in B; that is, A∪ B ={x | x ∈ A or x ∈ B}. (The symbol ''∪'' means ''union.'') For sets A and B, the set A x B is the set of all ordered pairs of elements of A and B; that is, A x B ={(a, b) | a ∈ A, b ∈ B}. (The symbol ''x'' means ''Cartesian product.'').

A graph G is a pair, G = (V, E), where V is a finite nonempty set, called the set of vertices of G and E ⊆ V x V . That is, the elements of E are pairs of elements of V. E is called the set of edges of G. G is called trivial if it has only one vertex.

Let V(G) denote the set of vertices, and E(G) denote the set of edges of a graph G. If the elements of E are ordered pairs, G is called a directed graph or digraph; otherwise, G is called an undirected graph. In an undirected graph, the pairs (u, v) and (v, u) represent the same edge.

Let G be a graph. A graph H is called a subgraph of G if V(H) ⊆ V(G) and E(H) ⊆ E(G); that is, every vertex of H is a vertex of G, and every edge in H is an edge in G.

A graph can be shown pictorially. The vertices are drawn as circles, and a label inside the circle represents the vertex. In an undirected graph, the edges are drawn using lines. In a directed graph, the edges are drawn using arrows.

Figure 7.1.3 shows some examples of undirected graphs.

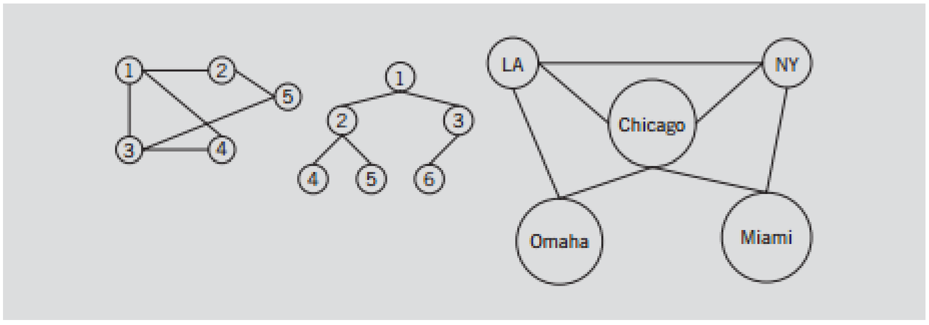

Figure 7.1.3 Various undirected graphs

Figure 7.1.4 shows some examples of directed graphs.

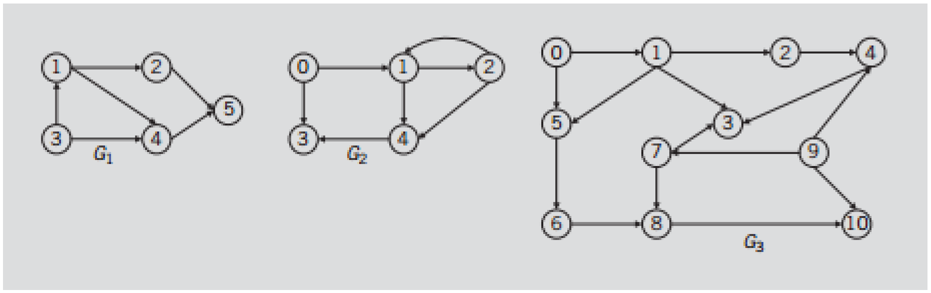

Figure 7.1.4 Various directed graphs

For the graphs of Figure 7.1.4, we have

V(G1) = {1, 2, 3, 4, 5}	E(G1) = {(1, 2), (1, 4), (2, 5), (3, 1), (3, 4), (4, 5)}
V(G2) = {0, 1, 2, 3, 4}	E(G2) = {(0, 1), (0, 3), (1, 2), (1, 4), (2, 1), (2, 4), (4, 3)}
V(G3) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}	E(G3) = {(0, 1), (0, 5), (1, 2), (1, 3), (1, 5), (2, 4), (4, 3), (5, 6), (6, 8), (7, 3), (7, 8), (8, 10), (9, 4), (9, 7), (9, 10)}

Let G be an undirected graph. Let u and v be two vertices in G. Then u and v are called adjacent if there is an edge from one to the other; that is, (u, v) ∈ E. An edge incident on a single vertex is called a loop. If two edges, e1 and e2, are associated with the same pair of vertices {u, v}, then e1 and e2 are called parallel edges. A graph is called a simple graph if it has no loops and no parallel edges. Let e = (u, v) be an edge in G. We then say that edge e is incident on the vertices u and v. The degree of u, written deg(u) or d(u), is the number of edges incident with u. We make the convention that each loop on a vertex u contributes 2 to the degree of u. u is called an even (odd) degree vertex if the degree of u is even (odd). There is a path from u to v if there is a sequence of vertices u1, u2, . . ., un such that u = u1, un = v and (ui , ui+ 1) is an edge for all i = 1, 2, . . ., n - 1. Vertices u and v are called connected if there is a path from u to v. A simple path is a path in which all the vertices, except possibly the first and last vertices, are distinct. A cycle in G is a simple path in which the first and last vertices are the same. G is called connected if there is a path from any vertex to any other vertex. A maximal subset of connected vertices is called a component of G.

Let G be a directed graph, and let u and v be two vertices in G. If there is an edge from u to v, that is, (u, v) ∈ E, we say that u is adjacent to v and v is adjacent from u. The definitions of the paths and cycles in G are similar to those for undirected graphs. G is called strongly connected if any two vertices in G are connected.

Consider the directed graphs of Figure 7.1.4. In G1, 1-4-5 is a path from vertex 1 to vertex 5. There are no cycles in G1. In G2, 1-2-1 is a cycle. In G3, 0-1-2-4-3 is a path from vertex 0 to vertex 3; 1-5-6-8-10 is a path from vertex 1 to vertex 10. There are no cycles in G3.

LESSON 7.2

Graph Representation

To write programs that process and manipulate graphs, the graphs must be stored—that is, represented—in computer memory. A graph can be represented (in computer memory) in several ways. We now discuss two commonly used ways: adjacency matrices and adjacency lists

Adjacency Matrices

Let G be a graph with n vertices, where n > 0. Let V(G) = {v1, v2, . . ., vn}. The adjacency matrix AG is a two-dimensional n x n matrix such that the (i, j)th entry of AG is 1 if there is an edge from vi to vj ; otherwise, the (i, j)th entry is 0. That is,

$$A_G(i,j) = \begin{cases} 1 & \text{if}(v_i, v_j) \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

In an undirected graph, if (vi , vj) ∈ E(G), then (vj , vi) ∈ E(G), so AG(i, j) = 1 = AG(j, i). It follows that the adjacency matrix of an undirected graph is symmetric.

Consider the directed graphs of Figure 7.1.4. The adjacency matrices of the directed graphs G1 and G2 are as follows:

$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \ A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Adjacency Lists

Let G be a graph with n vertices, where n > 0. Let V(G) = {v1, v2, . . ., vn}. In the adjacency list representation, corresponding to each vertex, v, there is a linked list such that each node of the linked list contains the vertex, u, such that (v, u) ∈ E(G). Because there are n nodes, we use an array, A, of size n, such that A[i] is a reference variable pointing to the first node of the linked list containing the vertices to which vi is adjacent. Clearly, each node has two components, say vertex and link. The component vertex contains the index of the vertex adjacent to vertex i.

Consider the directed graphs of Figure 7.1.4. Figure 7.2.1 shows the adjacency list of the directed graph G2.

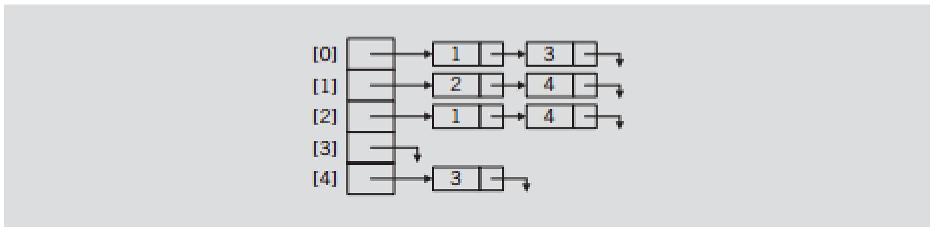

Figure 7.2.1 Adjacency list of graph G2 of Figure 7.1.4

Figure 7.2.2 shows the adjacency list of the directed graph G3.

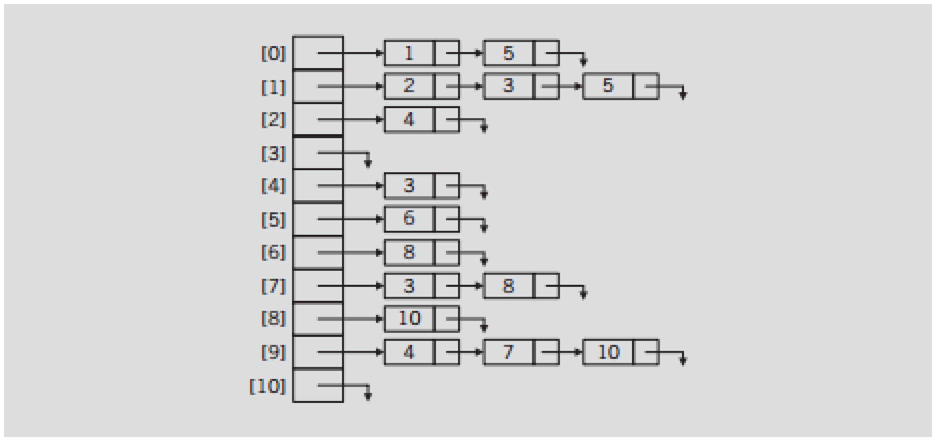

Figure 7.2.2 Adjacency list of graph G3 of Figure 7.1.4

LESSON 7.3

Operations on Graphs

Now that you know how to represent graphs in computer memory, the next obvious step is to learn the basic operations on a graph. The operations commonly performed on a graph are as follows:

1. Create the graph. That is, store the graph in computer memory using a particular graph representation.
2. Clear the graph. This operation makes the graph empty.
3. Determine whether the graph is empty.
4. Traverse the graph.
5. Print the graph.

We will add more operations on a graph when we discuss a specific application or a particular graph later in this chapter.

How a graph is represented in computer memory depends on the specific application. For illustration purposes, we use the adjacency list (linked list) representation of graphs. Therefore, for each vertex v the vertices adjacent to v (in a directed graph, also called the immediate successors) are stored in the linked list associated with v.

To manage the data in a linked list, we use the class unorderedLinkedList discussed in linked list chapter.

The labeling of the vertices of a graph depends on a specific application. If you are dealing with the graph of cities, you could label the vertices by the names of the cities. However, to write algorithms to manipulate a graph as well as to simplify the algorithm, there must be some ordering to the vertices. That is, we must specify the first vertex, the second vertex, and so on. Therefore, for simplicity, throughout this chapter we assume that the n vertices of the graphs are numbered 0, 1, . . ., n - 1. Moreover, it follows that the class we will design to implement the graph algorithm will not be a template.

Graphs as ADT

In this section, we describe the class to implement graphs as an abstract data type (ADT) and provide the definitions of the functions to implement the operations on a graph.

The following class defines a graph as an ADT:

```
class graphType
{
public:
    bool isEmpty() const;
      //Function to determine whether the graph is empty.
      //Postcondition: Returns true if the graph is empty;
      //    otherwise, returns false.

    void createGraph();
      //Function to create a graph.
      //Postcondition: The graph is created using the
      //    adjacency list representation.

    void clearGraph();
      //Function to clear graph.
      //Postcondition: The memory occupied by each vertex
      //    is deallocated.

    void printGraph() const;
      //Function to print graph.
      //Postcondition: The graph is printed.

    void depthFirstTraversal();
      //Function to perform the depth first traversal of
      //the entire graph.
      //Postcondition: The vertices of the graph are printed
      //    using the depth first traversal algorithm.


    void dftAtVertex(int vertex);
      //Function to perform the depth first traversal of
      //the graph at a node specified by the parameter vertex.
      //Postcondition: Starting at vertex, the vertices are
      //    printed using the depth first traversal algorithm.

    void breadthFirstTraversal();
      //Function to perform the breadth first traversal of
      //the entire graph.
      //Postcondition: The vertices of the graph are printed
      //    using the breadth first traversal algorithm.

    graphType(int size = 0);
      //Constructor
      //Postcondition: gSize = 0; maxSize = size;
      //    graph is an array of pointers to linked lists.

    ~graphType();
      //Destructor
      //The storage occupied by the vertices is deallocated.

protected:
    int maxSize;    //maximum number of vertices
    int gSize;      //current number of vertices
    unorderedLinkedList<int> *graph; //array to create
                                     //adjacency lists
private:
    void dft(int v, bool visited[]);
      //Function to perform the depth first traversal of
      //the graph at a node specified by the parameter vertex.
      //This function is used by the public member functions
      //depthFirstTraversal and dftAtVertex.
      //Postcondition: Starting at vertex, the vertices are
      //    printed using the depth first traversal algorithm.
};
```

A graph is empty if the number of vertices is 0—that is, if gSize is 0. Therefore, the definition of the function isEmpty is as follows:

```
bool graphType::isEmpty() const
{
    return (gSize == 0);
}
```

The definition of the function createGraph depends on how the data is input into the program. For illustration purposes, we assume that the data to the program is input from a file. The user is prompted for the input file. The data in the file appears in the following form:

```
5
0 2 4 ... -999
1 3 6 8 ... -999
...
```

The first line of input specifies the number of vertices in the graph. The first entry in the remaining lines specifies the vertex, and all of the remaining entries in the line (except the last) specify the vertices that are adjacent to the vertex. Each line ends with the number –999.

Using these conventions, the definition of the function createGraph is as follows:

```
void graphType::createGraph()
{
    ifstream infile;
    char fileName[50];

    int vertex;
    int adjacentVertex;

    if (gSize != 0)  //if the graph is not empty, make it empty
        clearGraph();

    cout << "Enter input file name: ";
    cin >> fileName;
    cout << endl;

    infile.open(fileName);

    if (!infile)
    {
        cout << "Cannot open input file." << endl;
        return;
    }
    infile >> gSize;   //get the number of vertices

    for (int index = 0; index < gSize; index++)
    {
        infile >> vertex;
        infile >> adjacentVertex;

        while (adjacentVertex != -999)
        {
            graph[vertex].insertLast(adjacentVertex);
            infile >> adjacentVertex;
        } //end while
    } // end for

    infile.close();
} //end createGraph
```

The function clearGraph empties the graph by deallocating the storage occupied by each linked list and then setting the number of vertices to 0.

```
void graphType::clearGraph()
{
    for (int index = 0; index < gSize; index++)
        graph[index].destroyList();

    gSize = 0;
} //end clearGraph
```

The definition of the function printGraph is given next:

```
void graphType::printGraph() const
{
    for (int index = 0; index < gSize; index++)
    {
        cout << index << " ";
        graph[index].print();
        cout << endl;
    }

    cout << endl;
} //end printGraph
```

The definitions of the constructor and the destructor are as follows:

```
    //Constructor
graphType::graphType(int size)
{
    maxSize = size;
    gSize = 0;
    graph = new unorderedLinkedList<int>[size];
}

    //Destructor
graphType::~graphType()
{
    clearGraph();
}
```

LESSON 7.4

Graph Traversals

Processing a graph requires the ability to traverse the graph. This section discusses the graph traversal algorithms.

Traversing a graph is similar to traversing a binary tree, except that traversing a graph is a bit more complicated. Now a binary tree has no cycles and starting at the root node we can traverse the entire tree. On the other hand, a graph might have cycles and we might not be able to traverse the entire graph from a single vertex (for example, if the graph is not connected). Therefore, we must keep track of the vertices that have been visited. We must also traverse the graph from each vertex (that has not been visited) of the graph. This ensures that the entire graph is traversed.

The two most common graph traversal algorithms are the depth-first traversal and breadth-first traversal, which are described next. For simplicity, we assume that when a vertex is visited, its index is output. Moreover, each vertex is visited only once. We use the bool array visited to keep track of the visited vertices.

Depth-First Traversal

The depth-first traversal is similar to the preorder traversal of a binary tree. The general algorithm is as follows:

```
for each vertex, v, in the graph
    if v is not visited
        start the depth first traversal at v
```

Consider the graph G3 of Figure 7.1.4. It is shown here again as Figure 7.4.1 for easy reference.

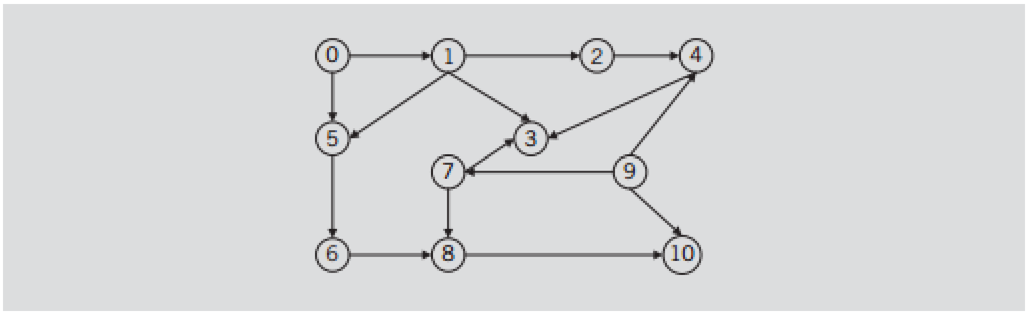

Figure 7.4.1 Directed graph G3

The depth-first ordering of the vertices of graph G3 in Figure 7.4.1 is as follows:

0 1 2 4 3 5 6 8 10 7 9

For the graph of Figure 7.4.1, the depth-first search starts at the vertex 0. After visiting all the vertices that can be reached starting at the vertex 0, the depth-first search starts at the next vertex that is not visited. There is a path from the vertex 0 to every other vertex except the vertices 7 and 9. Therefore, when the depth-first search starts at the vertex 0, all the vertices except 7 and 9 are visited before these vertices. After completing the depth-first search that started at the vertex 0, the depth-first search starts at the vertex 7 and then at the vertex 9.

Note that there is no path from the vertex 7 to the vertex 9. Therefore, after completing the depth-first search that started at the vertex 7, the depth-first search starts at the vertex 9.

The general algorithm to do a depth-first traversal at a given node, v, is as follows:

```
1. mark node v as visited
2. visit the node
3. for each vertex u adjacent to v
       if u is not visited
           start the depth first traversal at u
```

Clearly, this is a recursive algorithm. We use a recursive function, dft, to implement this algorithm. The vertex at which the depth-first traversal is to be started, and the bool array visited, are passed as parameters to this function.

```
void graphType::dft(int v, bool visited[])
{
    visited[v] = true;
    cout << " " << v << " ";  //visit the vertex

    linkedListIterator<int> graphIt;

        //for each vertex adjacent to v
    for (graphIt = graph[v].begin(); graphIt != graph[v].end();
                                     ++graphIt)
    {
        int w = *graphIt;
        if (!visited[w])
            dft(w, visited);
    } //end while
} //end dft
```

In the preceding code, note that the statement

linkedListIterator<int> graphIt;

declares graphIt to be an iterator. In the for loop, we use it to traverse a linked list (adjacency list) to which the pointer graph[v] points. Next, let us look at the statement

int w = *graphIt;

The expression *graphIt returns the label of the vertex, adjacent to the vertex v, to which graphIt points.

Next, we give the definition of the function depthFirstTraversal to implement the depth-first traversal of the graph.

```
void graphType::depthFirstTraversal()
{
    bool *visited; //pointer to create the array to keep
                   //track of the visited vertices
    visited = new bool[gSize];

    for (int index = 0; index < gSize; index++)
        visited[index] = false;

        //For each vertex that is not visited, do a depth
        //first traverssal
    for (int index = 0; index < gSize; index++)
        if (!visited[index])
            dft(index,visited);
    delete [] visited;
} //end depthFirstTraversal
```

The function depthFirstTraversal performs a depth-first traversal of the entire graph. The definition of the function dftAtVertex, which performs a depth-first traversal at a given vertex, is as follows:

```
void graphType::dftAtVertex(int vertex)
{
    bool *visited;

    visited = new bool[gSize];

    for (int index = 0; index < gSize; index++)
        visited[index] = false;

    dft(vertex, visited);

    delete [] visited;
} // end dftAtVertex
```

Breadth-First Traversal

The breadth-first traversal of a graph is similar to traversing a binary tree level-by-level (the nodes at each level are visited from left to right). All the nodes at any level, i, are visited before visiting the nodes at level i + 1.

The breadth-first ordering of the vertices of the graph G3 in Figure 7.4.1 is as follows:

0 1 5 2 3 6 4 8 10 7 9

For the graph G3, we start the breadth traversal at vertex 0. After visiting the vertex 0, we visit the vertices that are directly connected to it and are not visited, which are 1 and 5.

Next, we visit the vertices that are directly connected to 1 and are not visited, which are 2 and 3. After this, we visit the vertices that are directly connected to 5 and are not visited, which is 6. After this, we visit the vertices that are directly connected to 2 and are not visited, and so on.

As in the case of the depth-first traversal, because it might not be possible to traverse the entire graph from a single vertex, the breadth-first traversal also traverses the graph from each vertex that is not visited. Starting at the first vertex, the graph is traversed as much as possible; we then go to the next vertex that has not been visited. To implement the breadth-first search algorithm, we use a queue. The general algorithm is as follows:

```
1. for each vertex v in the graph
       if v is not visited
           add v to the queue //start the breadth first
           search at v
2. Mark v as visited
3. while the queue is not empty
       i.   Remove vertex u from the queue
       ii.  Retrieve the vertices adjacent to u
       iii. for each vertex w that is adjacent to u if w is not visited
           1. Add w to the queue
           2. Mark w as visited
```

The following C++ function, breadthFirstTraversal, implements this algorithm:

```
void graphType::breadthFirstTraversal()
{
    linkedQueueType<int> queue;

    bool *visited;
    visited = new bool[gSize];

    for (int ind = 0; ind < gSize; ind++)
        visited[ind] = false;    //initialize the array
                                 //visited to false

    linkedListIterator<int> graphIt;

    for (int index = 0; index < gSize; index++)
        if (!visited[index])
        {
            queue.addQueue(index);
            visited[index] = true;
            cout << " " << index << " ";

            while (!queue.isEmptyQueue())
            {
                int u = queue.front();
                queue.deleteQueue();

                for (graphIt = graph[u].begin();
                     graphIt != graph[u].end(); ++graphIt)
                {
                    int w = *graphIt;
                    if (!visited[w])
                    {
                        queue.addQueue(w);
                        visited[w] = true;
                        cout << " " << w << " ";
                    }
                }
            } //end while
        }

    delete [] visited;
} //end breadthFirstTraversal
```

As we continue to discuss graph algorithms, we will be writing C++ functions to implement specific algorithms, and so we will derive (using inheritance) new classes from the class graphType.