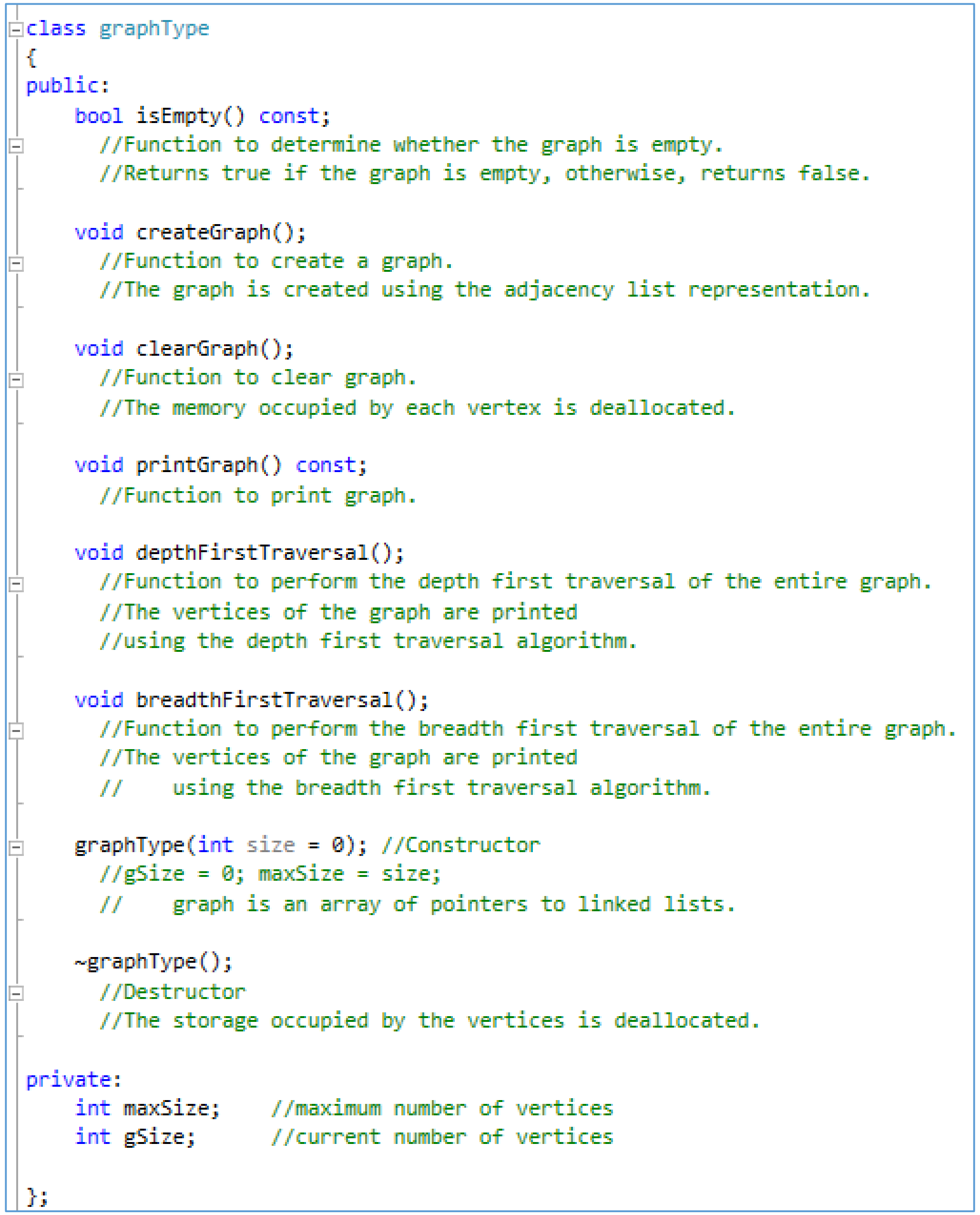
Graph Traversals
Depth-First Traversal
Breadth-First Traversal



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

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

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

    void printGraph() const;
      //Function to print graph.

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

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

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

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

private:
    int maxSize;    //maximum number of vertices
    int gSize;      //current number of vertices

};
```

Operations on Graphs

- Commonly performed operations
  - Create graph
    - Store graph in computer memory using a particular graph representation
  - Clear graph
    - Makes graph empty
  - Determine if graph is empty
  - Traverse graph
  - Print graph

Graphs as ADTs

- Function `createGraph`
  - Implementation
    - Depends on how data input into the program
- Function `clearGraph`
  - Empties the graph
    - Deallocates storage occupied by each linked list
    - Sets number of vertices to zero

Graph Traversals

- Processing a graph
  - Requires ability to traverse the graph
- Traversing a graph
  - Similar to traversing a binary tree
    - A bit more complicated
- Two most common graph traversal algorithms
  - Depth first traversal
  - Breadth first traversal

Graph Traversals (cont.)

Breadth-first search and Depth-first search work for either directed or undirected graphs.

- Must mark visited vertices so you do not go into an infinite loop!
- Initialize an array to false (0)to keep track unvisited / visited nodes

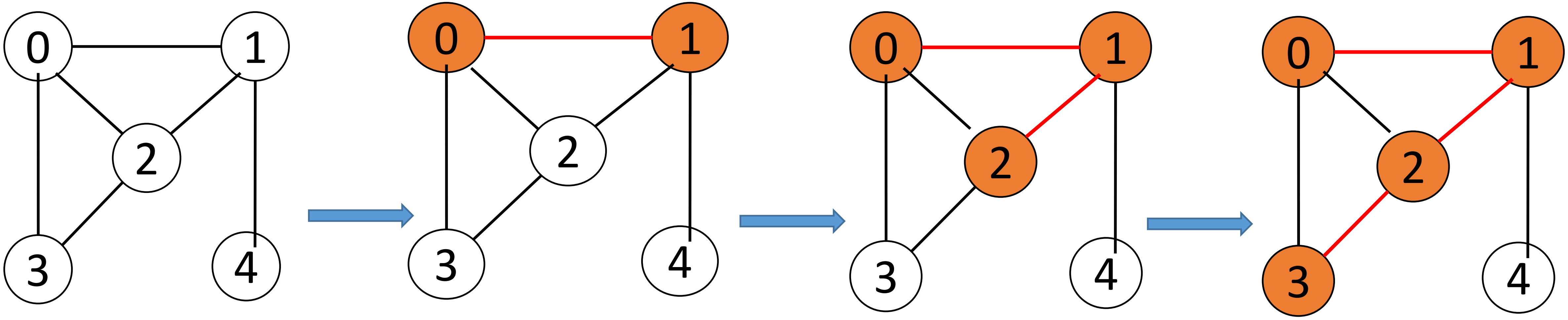
```
While some element of unvisited is false
    a. Select an unvisited vertex v
    b. Use BFS or DFS to visit all vertices reachable from v
End while
```

Depth-First Traversal

Depth First Traversal

- Similar to binary tree preorder traversal – Goes as far as possible from a vertex before backing up.
- General algorithm

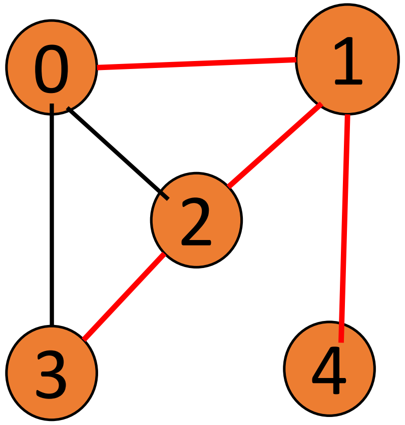
```
dft(vertex v) {
  visit v;
  for each neighbor w adjacent to  v
    if (w has not been visited) {
      dft(w);
    }
}
```



b) Node 1 is visited

c) Node 2 is visited

d) Node 3 is visited

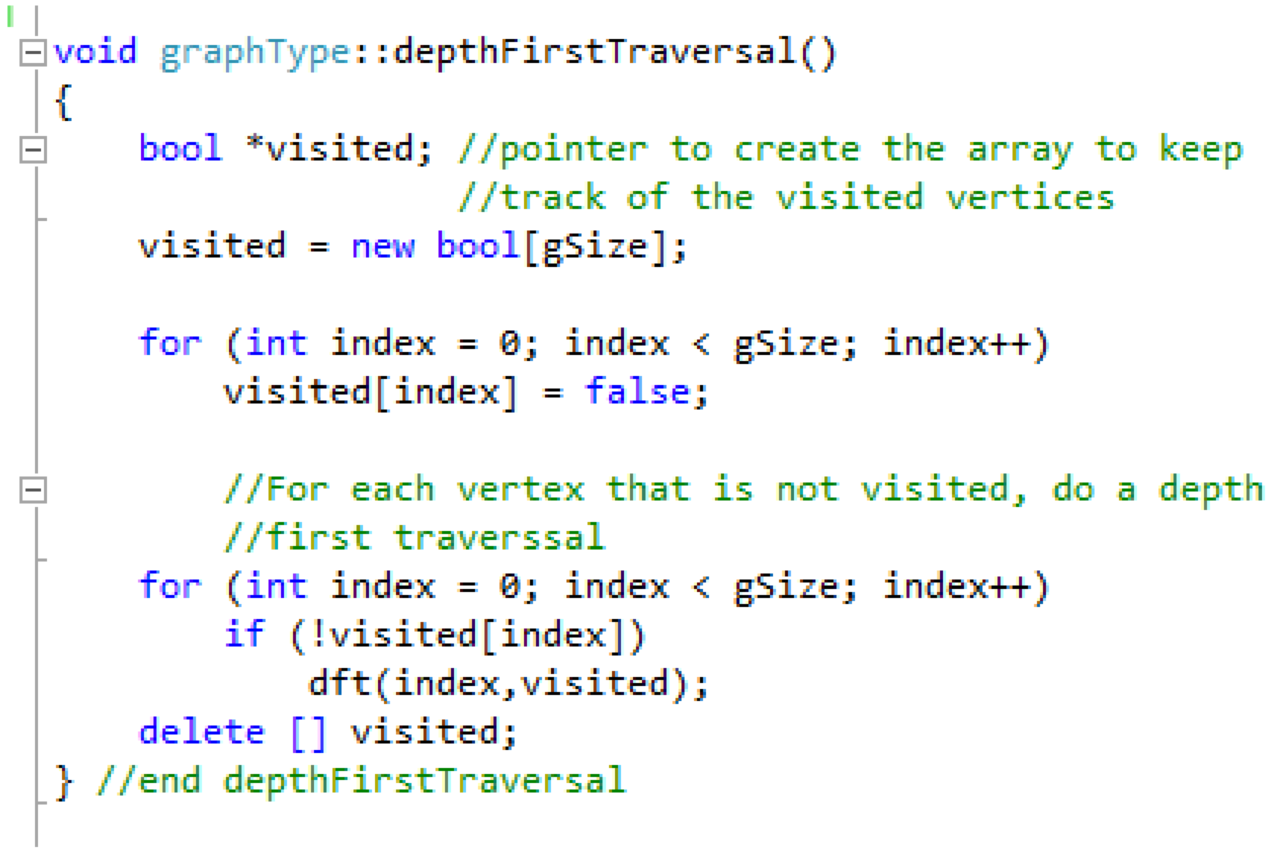


e) Node 4 is visited

- Suppose we start from vertex 0. First visit 0, then any of its neighbors, say 1. Now 1 visited (b).
- Vertex 1 has three neighbors: 0, 2 and 4. We only visit either 2 or 4. Let visit 2(c).
- Vertex 2 has three neighbors: 0, 1, 3. Visit un-visited node 3 (d).
- Since all neighbors of 3 have been visited, backtrack to 2.
- Since all neighbors of 2 have been visited, backtrack to 1.
- Vertex 4 is adjacent to 1, and un-visited. Visit 4 (e). Backtrack to 0. All nodes have been visited. The search ended.

Depth First Traversal (cont’d.)

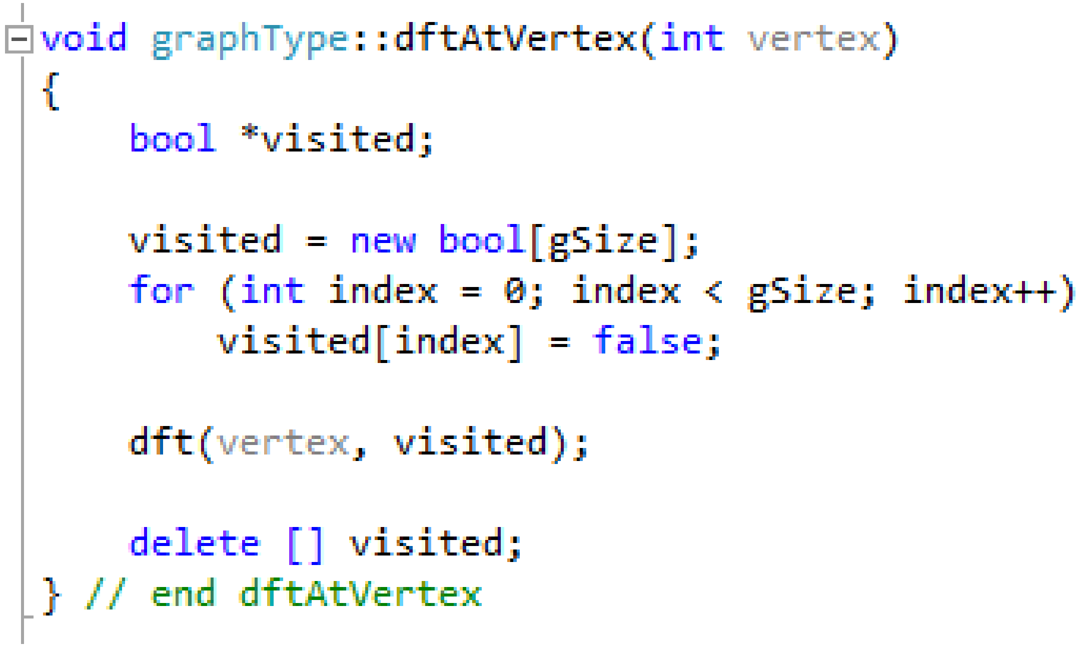
- **Function** `depthFirstTraversal`
  - Implements depth first traversal of the graph



```
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
```

Depth First Traversal (cont'd.)

- Function `depthFirstTraversal`
  - Performs a depth first traversal of entire graph
- Function `dftAtVertex`
  - Performs a depth first traversal at a given vertex



```
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

Breadth First Traversal

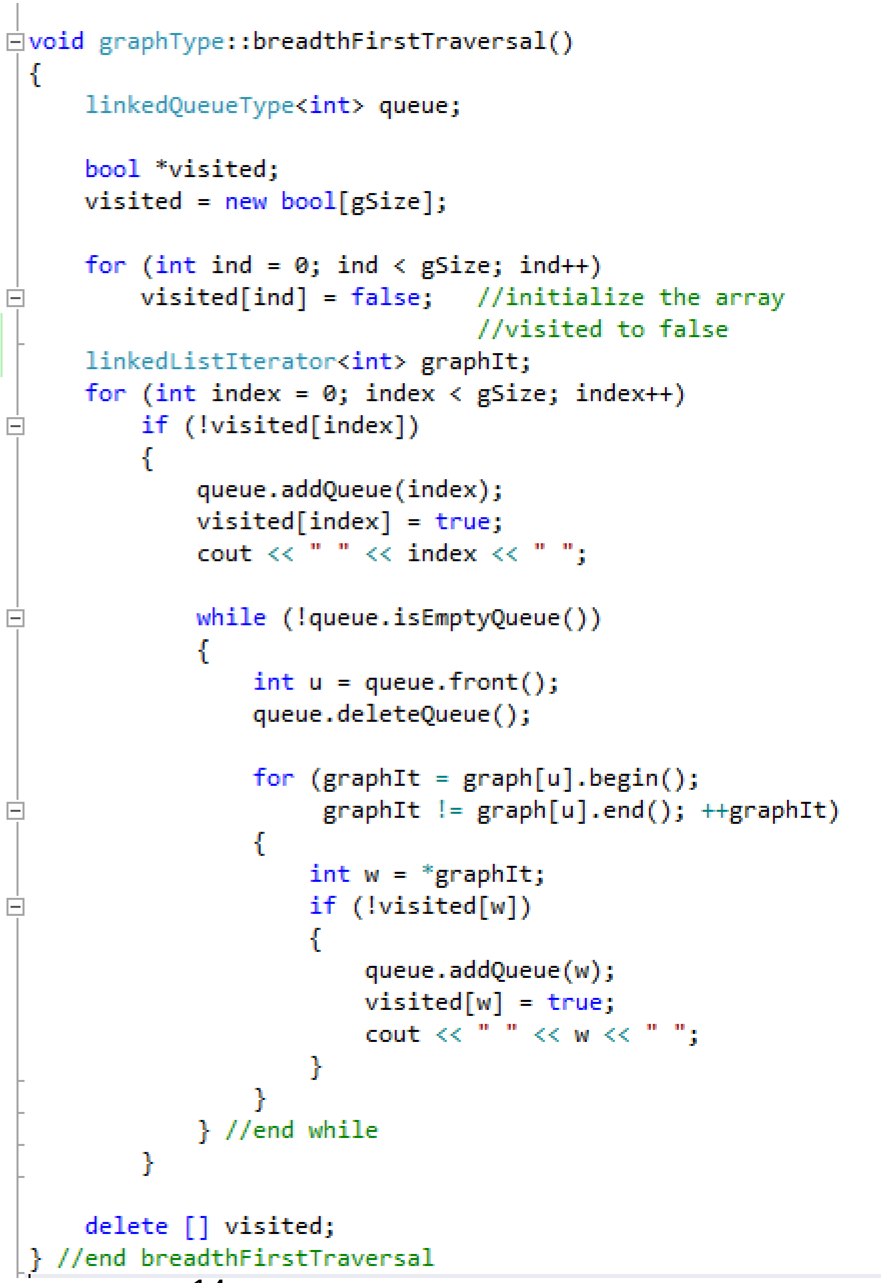
- Similar to traversing binary tree level-by-level
    - Visits all vertices adjacent to a vertex before going forward. BFT visits nodes by level.
        - Start from a given vertex of v; visit all neighbors of first neighbor of v
        - Then visit all neighbors of second neighbor x of v …

Breadth First Traversal

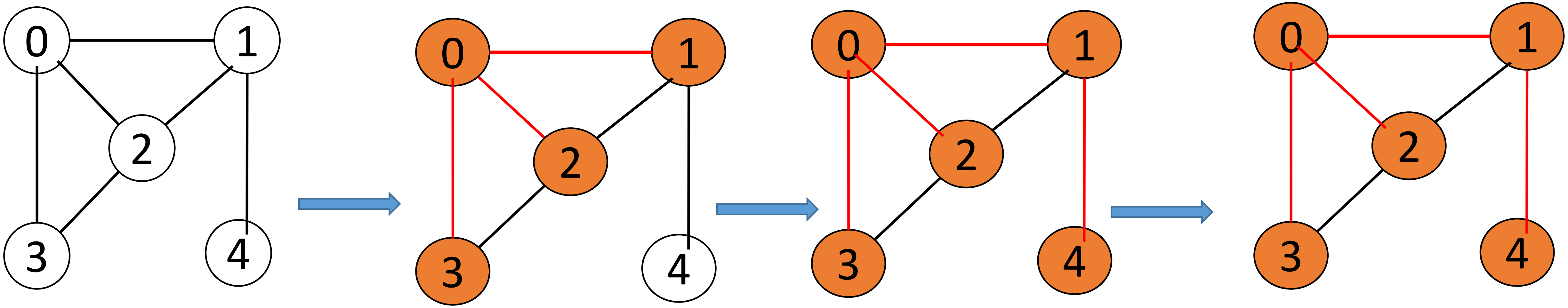
- General search algorithm
  - Breadth first search algorithm with a queue

```
for each vertex v in the graph
       if v is not visited
              add v to the queue // start bf search at v
Mark v as visited
While the queue is not empty
       remove vertex u from the queue
       retrieve the vertices adjacent to u
       for each vertex w that is adjacent to u
              add w to the queue
              mark w as visited
```

- C++ function implements breadFirstTraversal 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
```



b) Node 1 is visited

c) Node 2 is visited

d) Node 3 is visited

- Suppose we start from vertex 0. First visit 0 and all of its neighbors: 1, 2, 3 (b).
- Vertex 1 has three neighbors: 0, 2 and 4. Since 0 and 2 already visited, we only visit 4 (c).
- Vertex 2 has three neighbors, 0, 2, and 3 which have all been visited.
- Vertex 3 has three neighbors, 0, 2, and 4, which have all been visited.
- Vertex 4 has one neighbors which has been visited. All nodes have been visited. The search ended.

Time Complexity

- Since each edge and each vertex is visited only once, the time complexity of the Depth-first search and Breadth-first search functions is O(|V| + |E|), where |E| denotes the number of edges and |V| denotes the number of vertices.

The End!