TOPOLOGICAL ORDER

In college, before taking a particular course, students usually must take all applicable prerequisite courses. For example, before taking the Programming II course, students must take the Programming I course. However, certain courses can be taken independently of each other. The courses within a department can be represented as a directed graph. A directed edge from, say, vertex u to vertex v means that the course represented by the vertex u is a prerequisite of the course represented by the vertex v. It would be helpful for students to know, before starting a major, the sequence in which they can take courses so that before taking a particular course, they take all its prerequisite courses. This section describes an algorithm that can be used to output the vertices of a directed graph in such a sequence. Let us first introduce some terminology.

Let G be a directed graph and $V(G) = \{v_1, v_2, \ldots, v_n\}$, where $n > 0$. A **topological ordering** of $V(G)$ is a linear ordering $v_{i1}, v_{i2}, \ldots, v_{in}$ of the vertices such that if v_{ij} is a predecessor of v_{ik}, $j \neq k$, $1 \leq j \leq n$, and $1 \leq k \leq n$, then v_{ij} precedes v_{ik}, that is, $j < k$ in this linear ordering.

This section describes an algorithm that outputs the vertices of a directed graph in topological order. We assume that the graph has no cycles. We leave it for the reader, as an exercise, to modify the algorithm for the graphs that have cycles.

Because the graph has no cycles:

- There exists a vertex u in G such that u has no predecessor.
- There exists a vertex v in G such that v has no successor.

Suppose that the array `topologicalOrder` (of size n, the number of vertices) is used to store the vertices of G in topological order. Thus, if a vertex, say, `u`, is a successor of the vertex v and `topologicalOrder[j] = v` and `topologicalOrder[k] = u`, then `j < k`.

The topological sort algorithm can be implemented using either the depth first traversal or the breadth first traversal. This section discusses how to implement topological ordering using the breadth first traversal. The Programming Exercise at the end of this section describes how to implement the topological sort using the depth first traversal.

We extend the definition of the `class graphType` (using inheritance) to implement the breadth first topological ordering algorithm. Let us call this `class topologicalOrderType`. Next, we give the definition of the class that includes the functions to implement the topological ordering algorithm:

```
class topologicalOrderType: public graphType
{
public:
    void bfTopOrder();
      //output the vertices in breadth first topological order.

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

We leave the UML class diagram of the `class` `topologicalOrderType` and the inheritance hierarchy as an exercise. Next, we discuss how to implement the function `bfTopOrder`.

Breadth First Topological Ordering

Recall that the breadth first traversal algorithm is similar to traversing a binary tree level by level, and so the root node (which has no predecessor) is visited first. Therefore, in the breadth first topological ordering, we first find a vertex that has no predecessor vertex and place it first in the topological ordering. We next find the vertex, say, `v`, all of whose predecessors have been placed in the topological ordering and place `v` next in the topological ordering. To keep track of the number of vertices of a vertex, we use the array `predCount`. Initially, `predCount[j]` is the number of predecessors of the vertex v_j. The queue used to guide the breadth first traversal is initialized to those vertices v_k such that `predCount[k]` is zero. In essence, the general algorithm is:

1. Create the array `predCount` and initialize it so that `predCount[i]` is the number of predecessors of the vertex v_i.
2. Initialize the queue, say, `queue`, to all those vertices v_k so that `predCount[k]` is zero. (Clearly, `queue` is not empty because the graph has no cycles.)
3. `while` the queue is not empty
   - 3.1. Remove the front element, `u`, of the queue.
   - 3.2. Put `u` in the next available position, say, `topologicalOrder[topIndex]`, and increment `topIndex`.
   - 3.3. For all the immediate successors `w` of `u`:
     - 3.3.1. Decrement the predecessor count of `w` by `1`.
     - 3.3.2. `if` the predecessor count of `w` is zero, add `w` to `queue`.

Consider the graph *G* of Figure T-1.

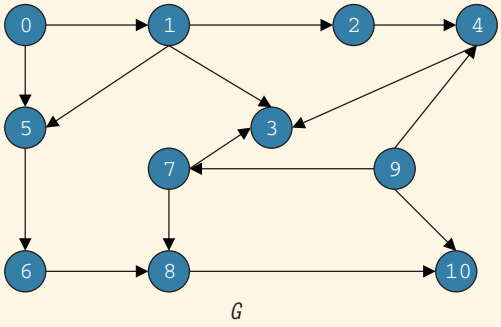

FIGURE T-1 Graph *G*

The graph *G* has no cycles. The vertices of *G* in breadth first topological ordering are:

Breadth First Topological Order: `0 9 1 7 2 5 4 6 3 8 10`

Next, we illustrate the breadth first topological ordering of the graph *G*.

After Steps 1 and 2 execute, the arrays **`predCount`**, **`topologicalOrder`**, and **`queue`** are as shown in Figure T-2. (Notice that for simplicity, we show only the elements of the queue.)

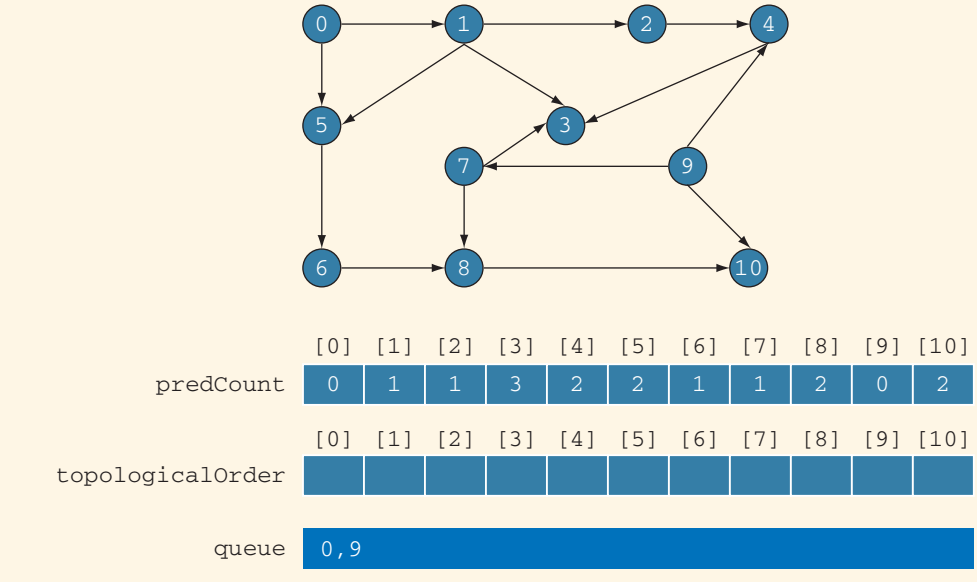

FIGURE T-2 Arrays `predCount`, `topologicalOrder`, and `queue` after Steps 1 and 2 execute

Step 3 executes as long as the queue is nonempty.

Iteration 1 of Step 3: After Step 3.1 executes, the value of `u` is `0`. Step 3.2 stores the value of `u`, which is `0`, in the next available position in the array `topologicalOrder`. Notice that `0` is stored at position `0` in this array. Step 3.3 reduces the predecessor count of all the successors of `0` by `1`, and if the predecessor count of any successor node of `0` reduces to `0`, that node is added to `queue`. The successor nodes of node `0` are nodes `1` and `5`. The predecessor count of node `1` reduces to `0`, and the predecessor count of node `5` reduces to `1`. Node `1` is added to `queue`. After the first iteration of Step 3, the arrays `predCount`, `topologicalOrder`, and `queue` are as shown in Figure T-3.

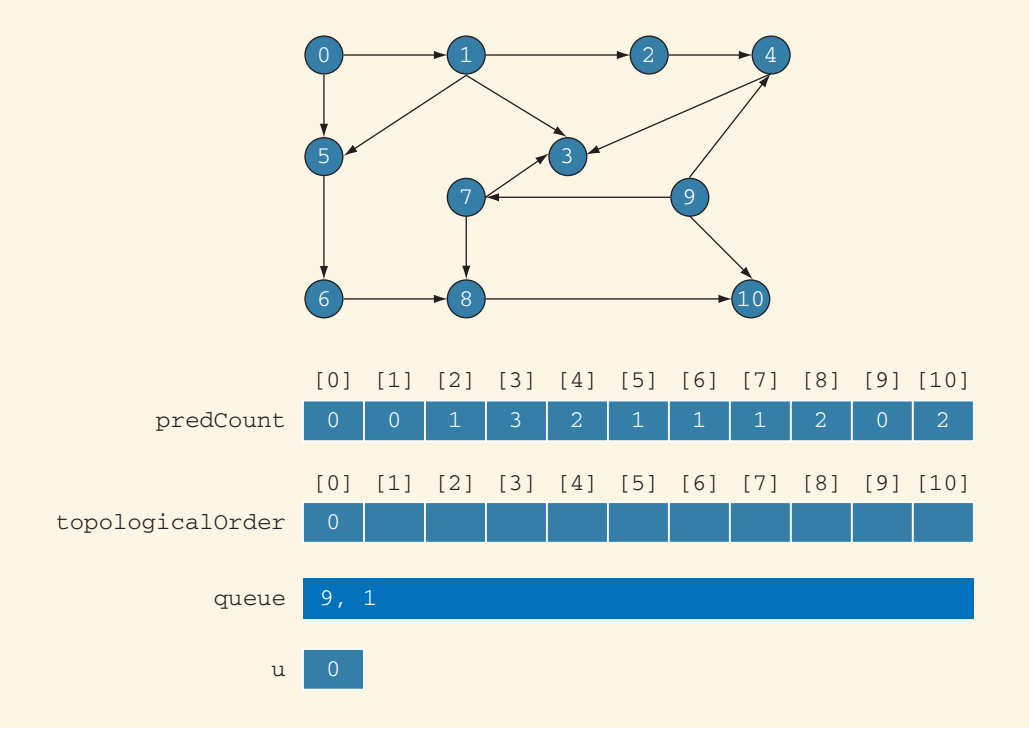

FIGURE T-3 Arrays `predCount`, `topologicalOrder`, and `queue` after the first iteration of Step 3

Iteration 2 of Step 3: The queue is nonempty. After Step 3.1 executes, the value of `u` is `9`. Step 3.2 stores the value of `u`, which is `9`, in the next available position in the array `topologicalOrder`. Notice that `9` is stored at position `1` in this array. Step 3.3 reduces the predecessor count of all the successors of `9` by `1`, and if the predecessor count of any successor node of `9` reduces to `0`, that node is added to `queue`. The successor nodes of node `9` are nodes `4`, `7`, and `10`. The predecessor count of node `7` reduces to `0` and the predecessor count of nodes `4` and `10` reduces to `1`. Node `7` is added to `queue`. After the second iteration of Step 3, the arrays `predCount`, `topologicalOrder`, and `queue` are as shown in Figure T-4.

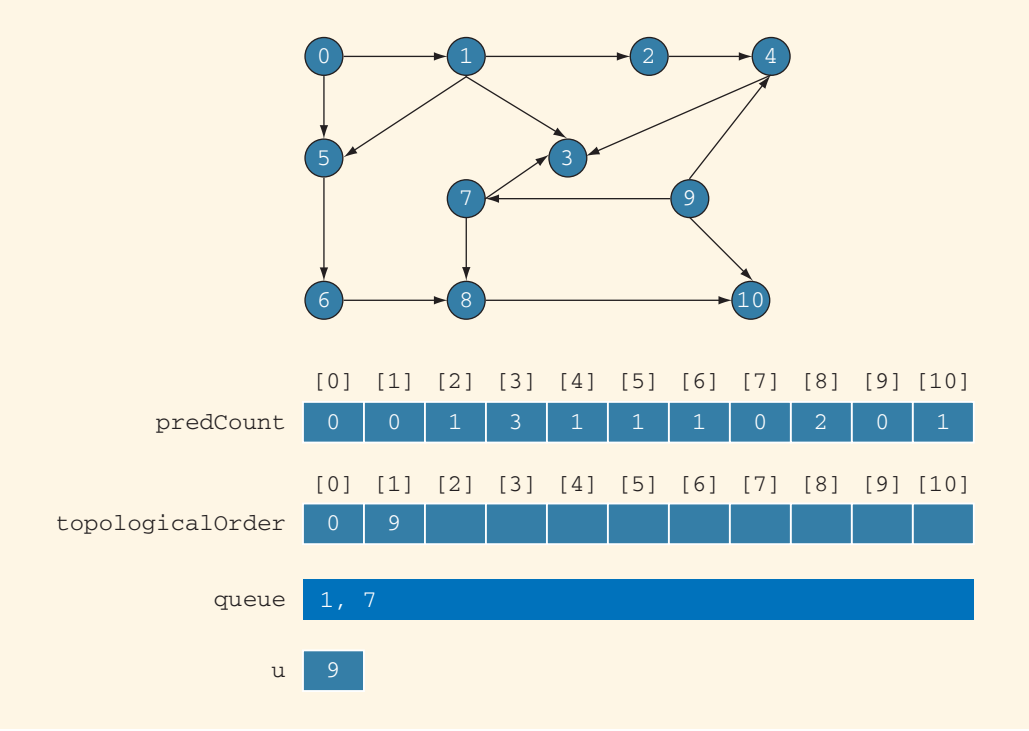

FIGURE T-4 Arrays `predCount`, `topologicalOrder`, and `queue` after the second iteration of Step 3

Iteration 3 of Step 3: The queue is nonempty. After Step 3.1 executes, the value of `u` is `1`. Step 3.2 stores the value of `u`, which is `1`, in the next available position in the array `topologicalOrder`. Notice that `1` is stored at position `2` in this array. Step 3.3 reduces the predecessor count of all the successors of `1` by `1`, and if the predecessor count of any successor node of `1` reduces to `0`, that node is added to `queue`. The successor nodes of node `1` are nodes `2`, `3`, and `5`. The predecessor count of nodes `2` and `5` reduces to `0`, and the predecessor count of node `3` reduces to `2`. Nodes `2` and `5`, in this order, are added to `queue`. After the third iteration of Step 3, the arrays `predCount`, `topologicalOrder`, and `queue` are as shown in Figure T-5.

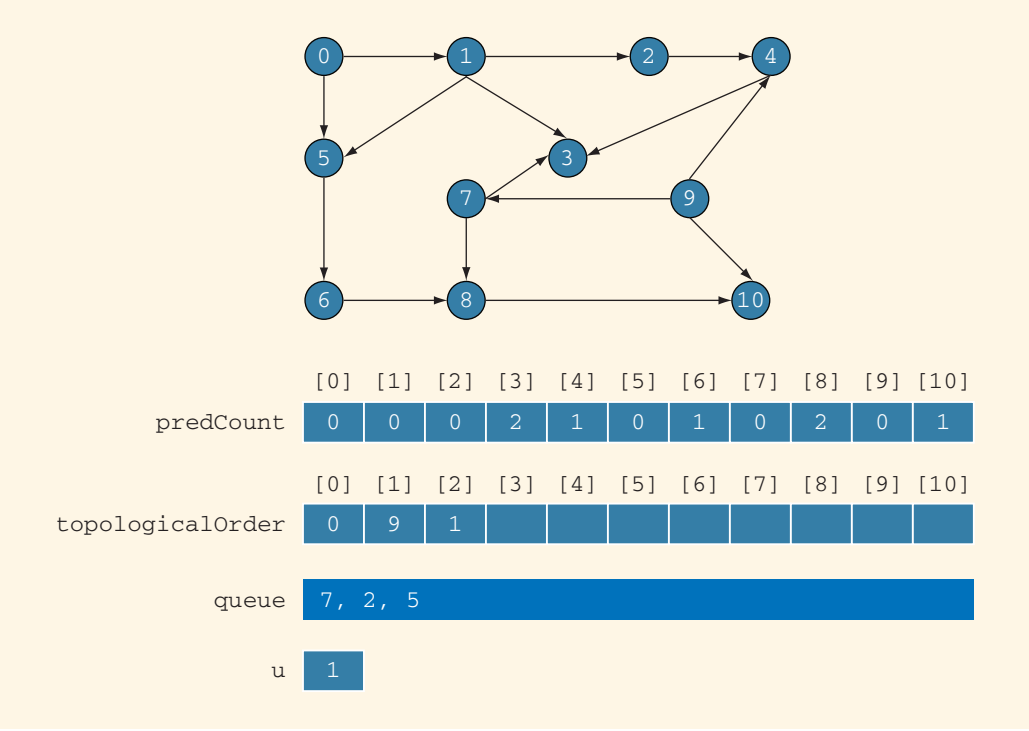

FIGURE T-5 Arrays `predCount`, `topologicalOrder`, and `queue` after the third iteration of Step 3

If you repeat Step 3 eight more times, the arrays **`predCount`**, **`topologicalOrder`**, and **`queue`** are as shown in Figure T-6.

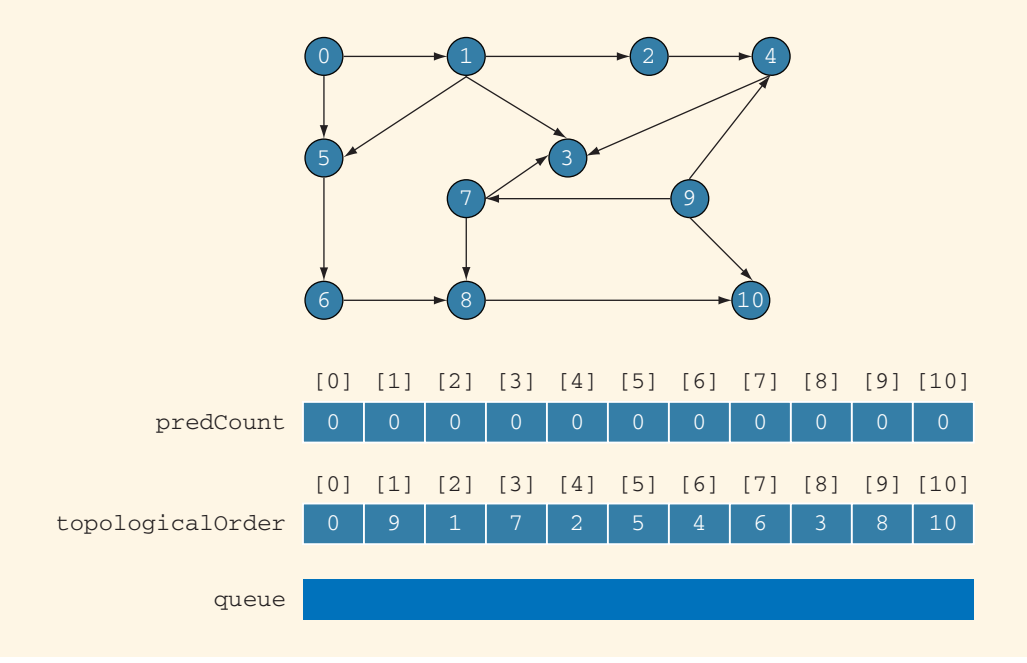

FIGURE T-6 Arrays `predCount`, `topologicalOrder`, and `queue` after Step 3 executes eight more times

In Figure T-6, the array `topologicalOrder` shows the breadth first topological ordering of the nodes of the graph *G*.

The following C++ function implements this breadth first topological ordering algorithm:

```
void topologicalOrderType::bfTopOrder()
{
    linkedQueueType<int> queue;

    int *topologicalOrder; //pointer to the array to store
                           //breadth first topological ordering
    topologicalOrder = new int[gSize];

    int topIndex = 0;

    linkedListIterator<int> graphIt; //iterator to traverse a
                                     //linked list

    int *predCount;     //pointer to the array to store the
                        //predecessor count of a vertex.
    predCount = new int[gSize];

    for (int ind = 0; ind < gSize; ind++)
        predCount[ind] = 0;
```

```
    //Determine the predecessor count of all the vertices.
for (int ind = 0; ind < gSize; ind++)
{
    for (graphIt = graph[ind].begin();
         graphIt != graph[ind].end(); ++graphIt)
    {
        int w = *graphIt;
        predCount[w]++;
    }
}

    //Initialize queue: If the predecessor count of
    //vertex is 0, put this node into the queue.
for (int ind = 0; ind < gSize; ind++)
    if (predCount[ind] == 0)
       queue.addQueue(ind);

while (!queue.isEmptyQueue())
{
    int u = queue.front();
    queue.deleteQueue();
    topologicalOrder[topIndex++] = u;

       //Reduce the predecessor count of all the successors
       //of u by 1. If the predecessor count of a vertex
       //becomes 0, put this vertex into the queue.
    for (graphIt = graph[u].begin();
         graphIt != graph[u].end(); ++graphIt)
    {
        int w = *graphIt;
        predCount[w]--;
        if (predCount[w] == 0)
           queue.addQueue(w);
    }
}//end while

   //output the vertices in breadth first topological order
for (int ind = 0; ind < gSize; ind++)
    cout << topologicalOrder[ind] << " ";

cout << endl;

delete [] topologicalOrder;
delete [] predCount;
}//bfTopOrder
```

We leave the definition of the constructor as an exercise.

QUICK REVIEW

1. Let G be a graph and $V(G) = \{v_1, v_2, \ldots, v_n\}$, where $n > 0$. A topological ordering of $V(G)$ is a linear ordering $v_{i1}, v_{i2}, \ldots, v_{in}$ of the vertices such that if v_{ij} is a predecessor of v_{ik}, $j \neq k$, $1 \leq j \leq n$, and $1 \leq k \leq n$, then v_{ij} precedes v_{ik}, that is, $j < k$ in this linear ordering.

EXERCISE

1. List the nodes of the graph of Figure T-7 in a breadth first topological ordering.

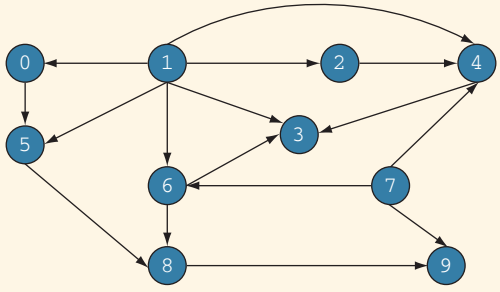

FIGURE T-7 Graph for Exercise 1

PROGRAMMING EXERCISES

1. Write a program to test the breadth first topological ordering algorithm.
2. (**Depth first topological ordering**) Let G be a graph and $V(G) = \{v_1, v_2, \ldots, v_n\}$, where $n > 0$. Recall that a topological ordering of $V(G)$ is a linear ordering $v_{i1}, v_{i2}, \ldots, v_{in}$ of the vertices such that if v_{ij} is a predecessor of v_{ik}, $j \neq k$, $1 \leq j \leq n$, and $1 \leq k \leq n$, then v_{ij} precedes v_{ik}, that is, $j < k$ in this linear ordering. Suppose that G has no cycles. The following algorithm, a **depth first topological order**, lists the nodes of the graph in a topological ordering.

   In a depth first topological ordering, we first find a vertex that has no successors (such a vertex exists because the graph has no cycles), and place it last in the topological order. After we have placed all the successors of a vertex in topological order, we place the vertex in the topological order before any of its successors. Clearly, in the depth first topological ordering, first we find the vertex to be placed in `topologicalOrder[n - 1]`, then `topologicalOrder[n - 2]`, and so on.

   Write the definitions of the C++ functions to implement the depth first topological ordering. Add these functions to the `class` `topologicalOrderType`, which is derived from the `class` `graphType`. Also, write a program to test your depth first topological ordering.