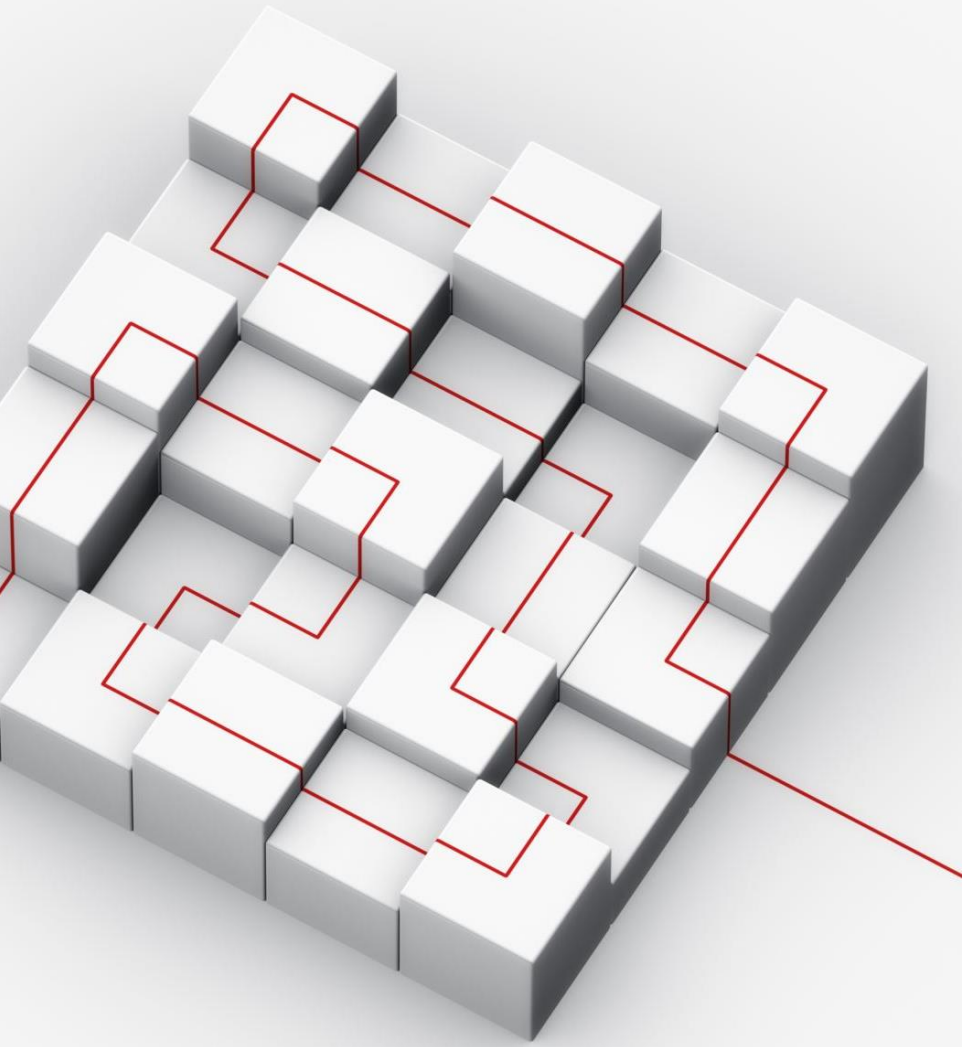
DATA STRUCTURES AND ALGORITHMS

DR SAMABIA TEHSIN

BS (AI)



Searching

Searching in data structure refers to the process of finding the required information from a collection of items stored as elements in the computer memory. These sets of items are in different forms, such as an array, linked list, graph, or tree. Another way to define searching in the data structures is by locating the desired element of specific characteristics in a collection of items.

Searching Methods

Searching in the data structure can be done by applying searching algorithms to check for or extract an element from any form of stored data structure.

Linear Search

Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.

Linear Search

Linear Search Algorithm

```
LinearSearch(array, key)
        for each item in the array
                if item == value
                        return its index
```

For searching operations in smaller arrays (<100 items).

Binary Search

- Binary Search is a searching algorithm for finding an element's position in a sorted array.
- In this approach, the element is always searched in the middle of a portion of an array.
- Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

Binary Search Working

Binary Search Algorithm can be implemented in two ways which are discussed below.

1. Iterative Method
2. Recursive Method

The recursive method follows the divide and conquer approach.

Binary Search Algorithm

Iteration Method

do until the pointers low and high meet each other.

```
mid = (low + high)/2
if (x == arr[mid])
        return mid
else if (x > arr[mid]) // x is on the right side
        low = mid + 1
else // x is on the left side
        high = mid - 1
```

Binary Search Algorithm

Recursive Method

```
binarySearch(arr, x, low, high)
   if low > high
         return False
   else
         mid = (low + high) / 2
         if x == arr[mid]
               return mid
         else if x > arr[mid] // x is on the right side
               return binarySearch(arr, x, mid + 1, high)
         else // x is on the right side
               return binarySearch(arr, x, low, mid - 1)
```

Depth First Search

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

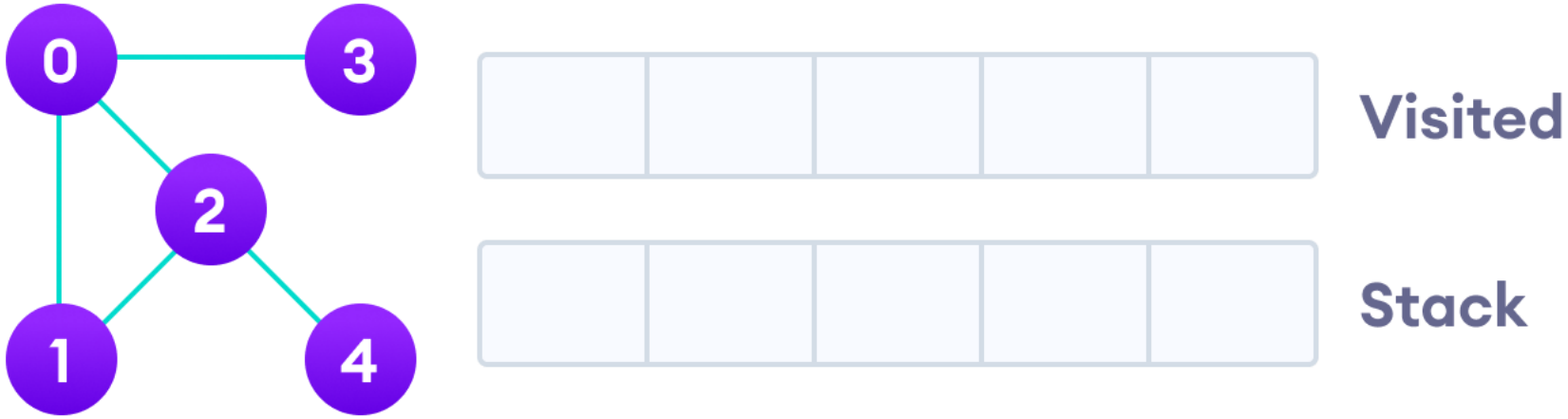
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

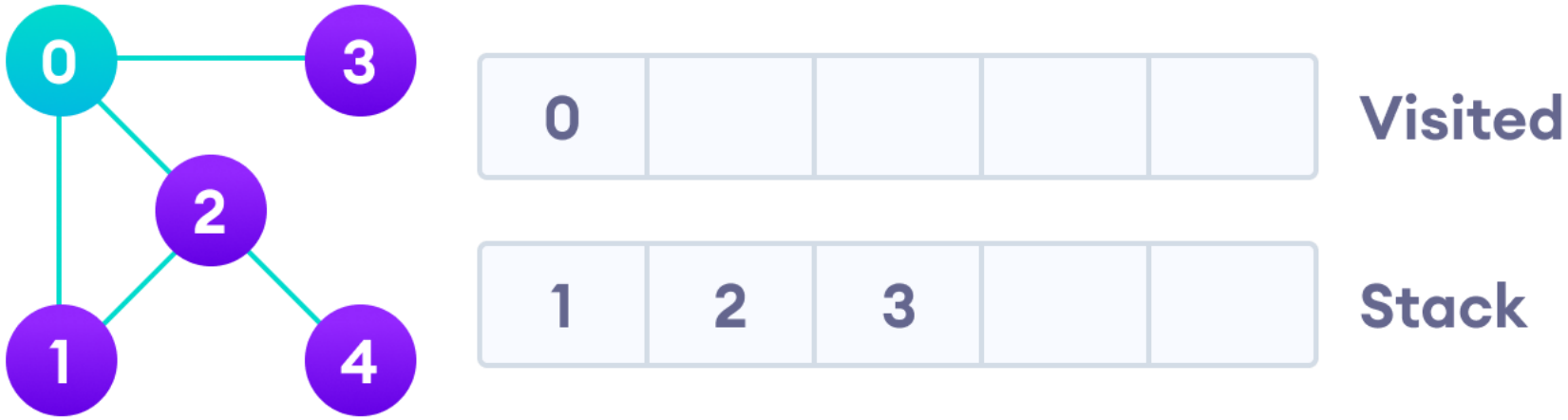
Depth First Search Algorithm

Undirected graph with 5 vertices



Depth First Search Algorithm

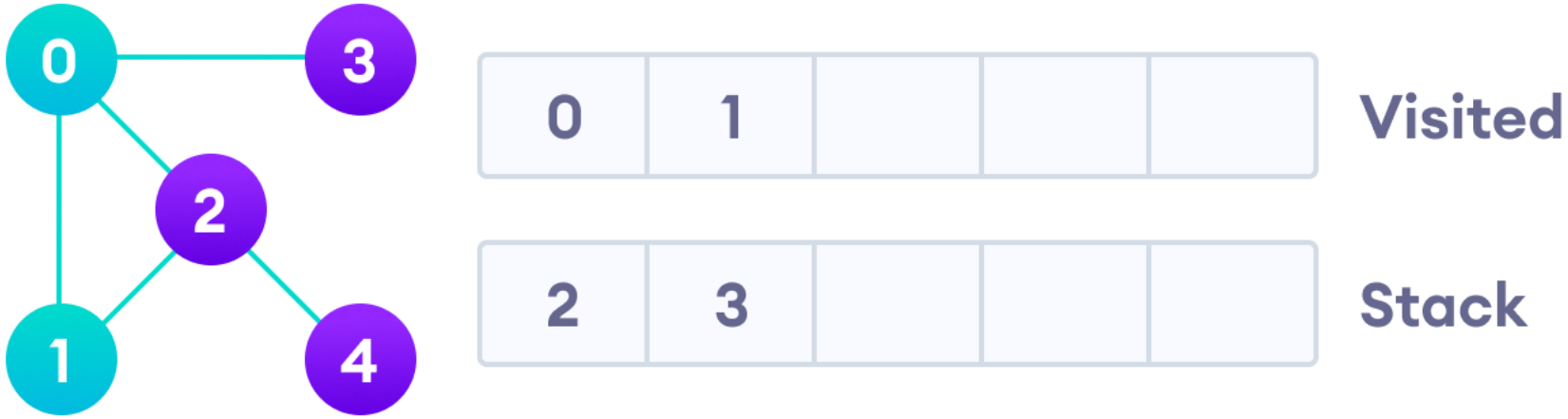
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Visit the element and put it in the visited list

Depth First Search Algorithm

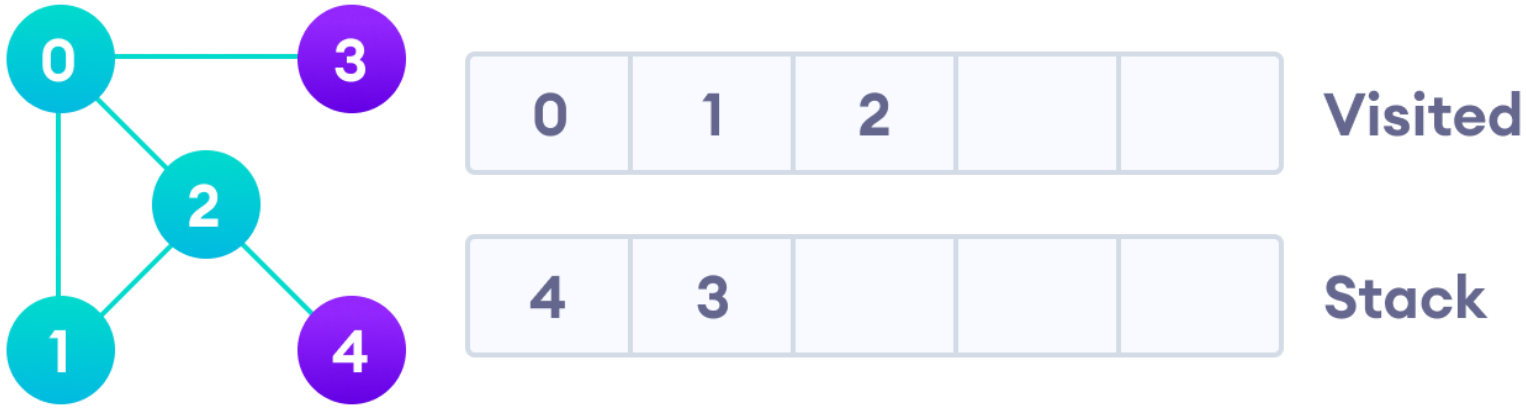
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



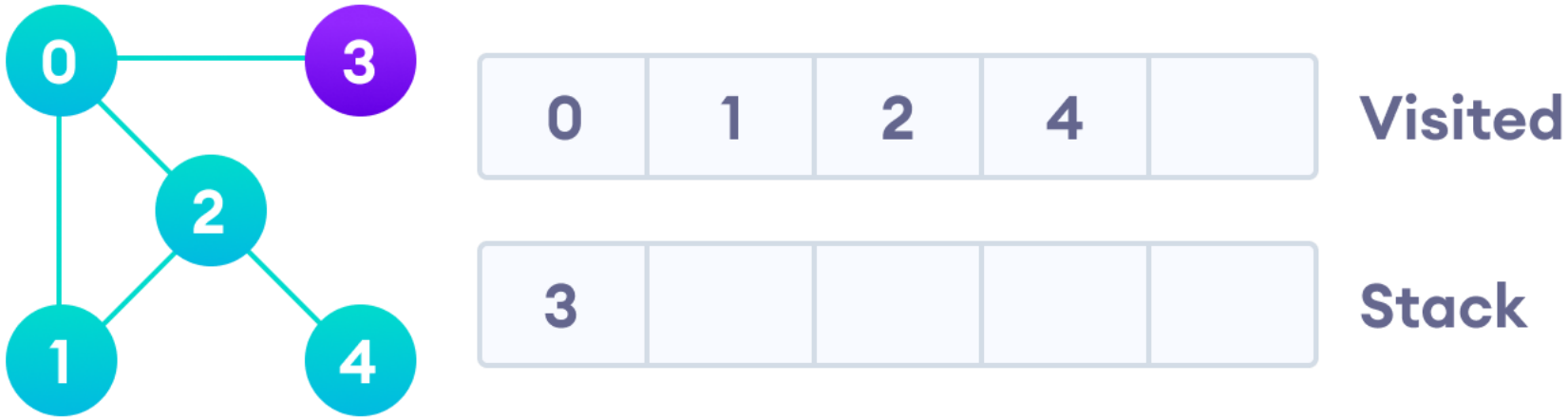
Visit the element at the top of stack

Depth First Search Algorithm

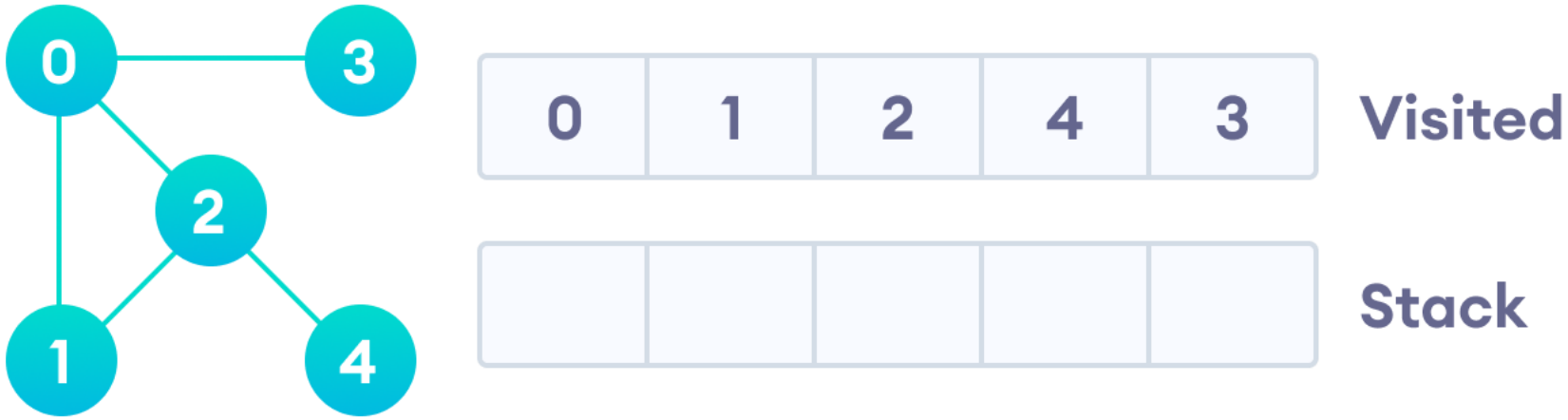
Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Depth First Search Algorithm



Depth First Search Algorithm



DFS Pseudocode (recursive implementation)

```
DFS(G, u)
   u.visited = true
   for each v ∈ G.Adj[u]
         if v.visited == false
               DFS(G,v)

init() {
   For each u ∈ G
         u.visited = false
   For each u ∈ G
         DFS(G, u)
}
```

In C++, the std::list refers to a storage container. The std:list allows you to insert and remove items from anywhere. The std::list is implemented as a doubly-linked list. This means list data can be accessed bi-directionally and sequentially.

Function	Description
insert()	This function inserts a new item before the position the iterator points.
push_back()	This functions add a new item at the list's end.
push_front()	It adds a new item at the list's front.
pop_front()	It deletes the list's first item.
size()	This function determines the number of list elements.
front()	To determines the list's first items.
back()	To determines the list's last item.
reverse()	It reverses the list items.
merge()	It merges two sorted lists.

DFS Code

```
class Graph {
  int numVertices;
  list<int> *adjLists;
  bool *visited;
   public:
  Graph(int V);
  void addEdge(int src, int dest);
  void DFS(int vertex);
};

// Initialize graph
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
  visited = new bool[vertices];
}

// Add edges
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_front(dest);
}
```

```
// DFS algorithm
void Graph::DFS(int vertex) {
  visited[vertex] = true;
  list<int> adjList = adjLists[vertex];

  cout << vertex << " ";

  list<int>::iterator i;
  for (i = adjList.begin(); i != adjList.end(); ++i)
    if (!visited[*i])
      DFS(*i);
}

int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 3);
  g.DFS(2);
  return 0;
}
```

Breadth first search

BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

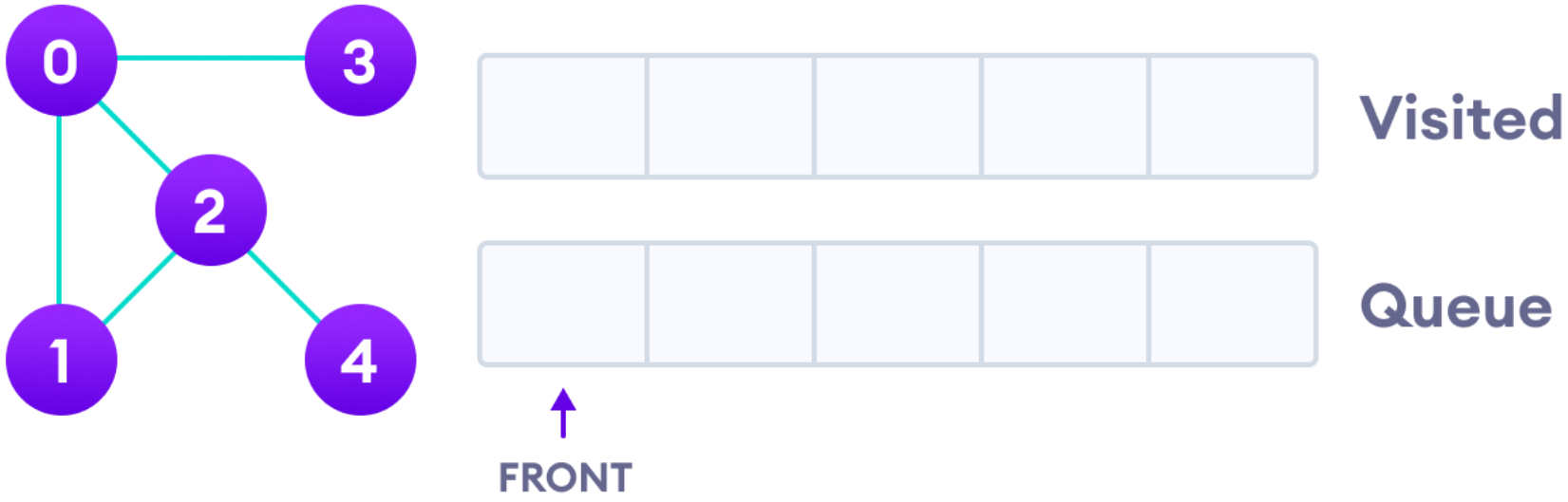
1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

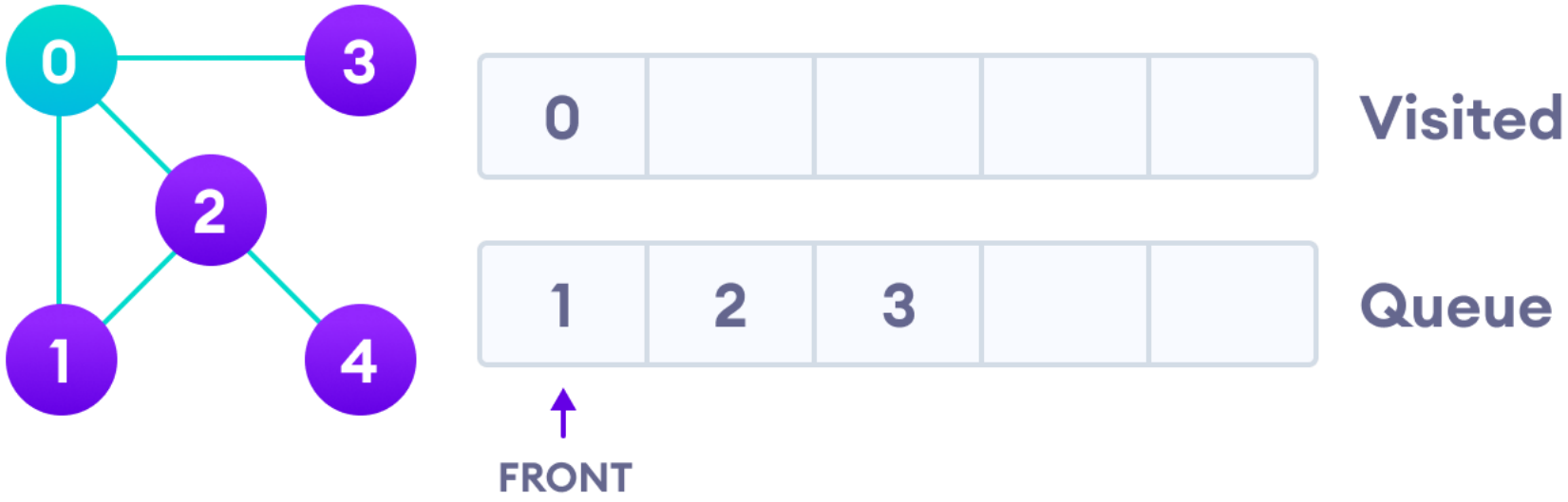
The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

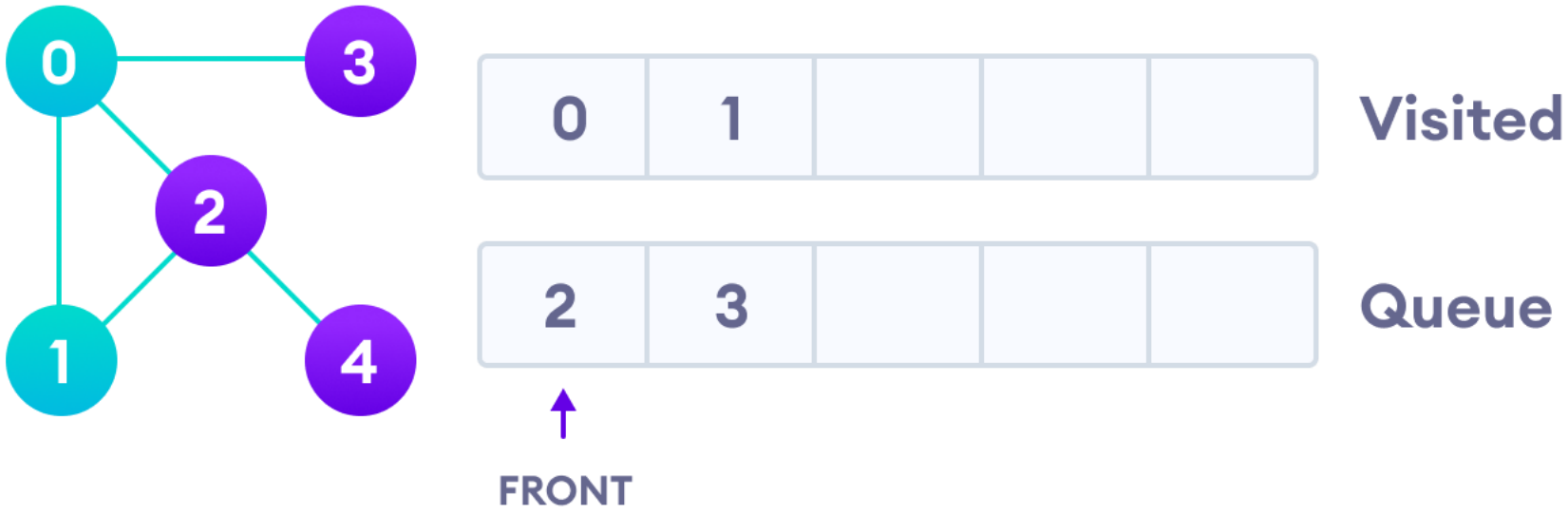
BFS example



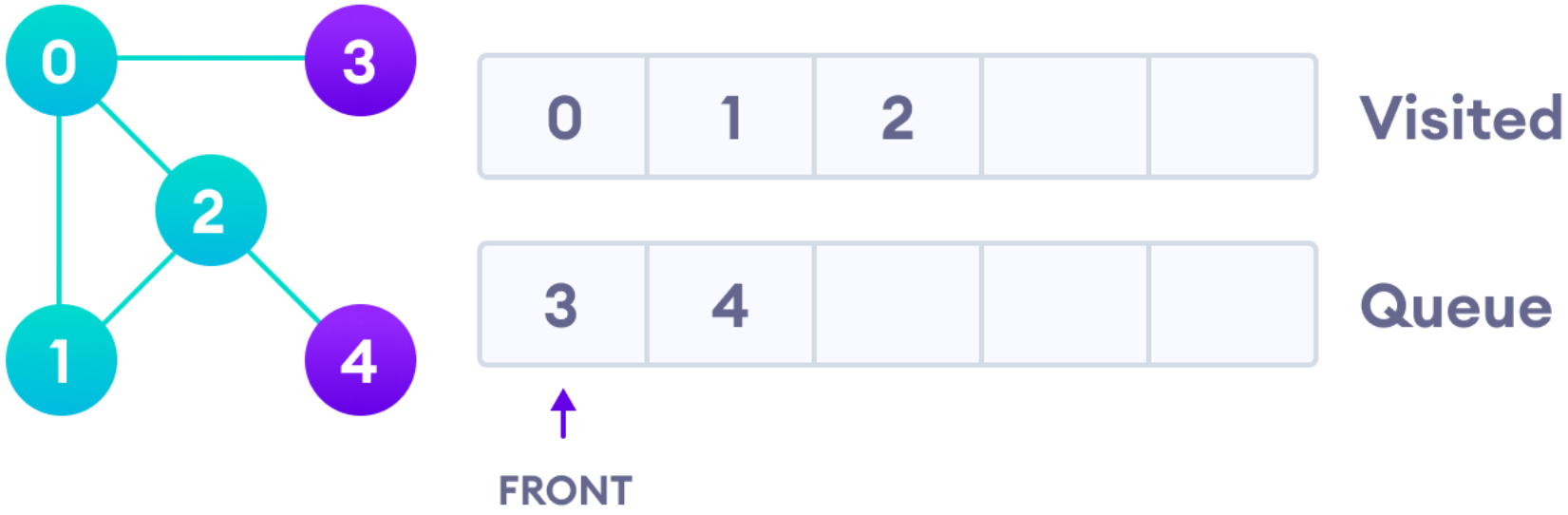
BFS example



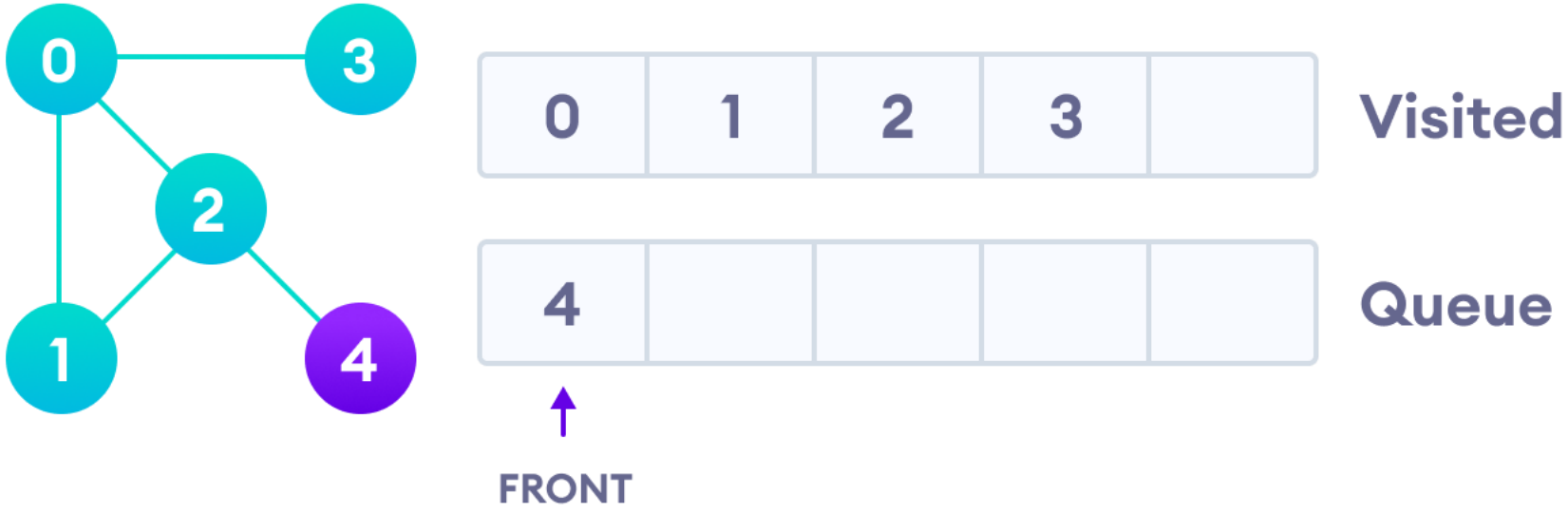
BFS example



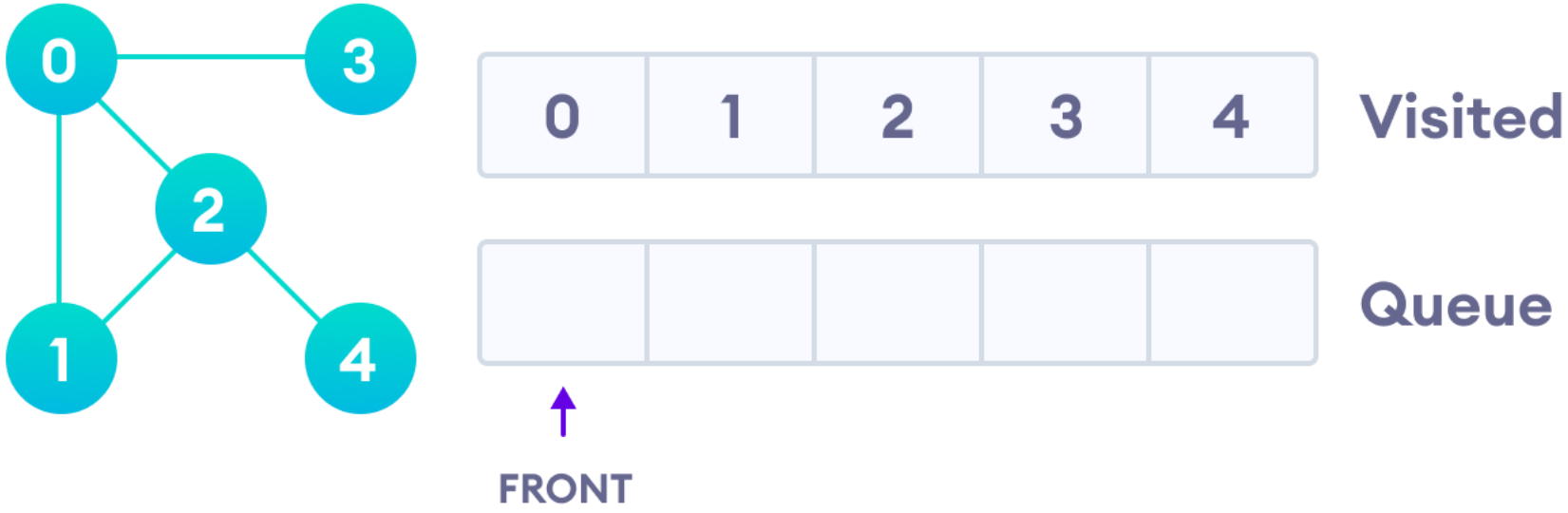
Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



BFS example



BFS example



BFS

BFS pseudocode

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
  remove the head u of Q
  mark and enqueue all (unvisited) neighbours of u
```

Credits and Acknowledgements

https://www.gatevidyalay.com

https://www.programiz.com/