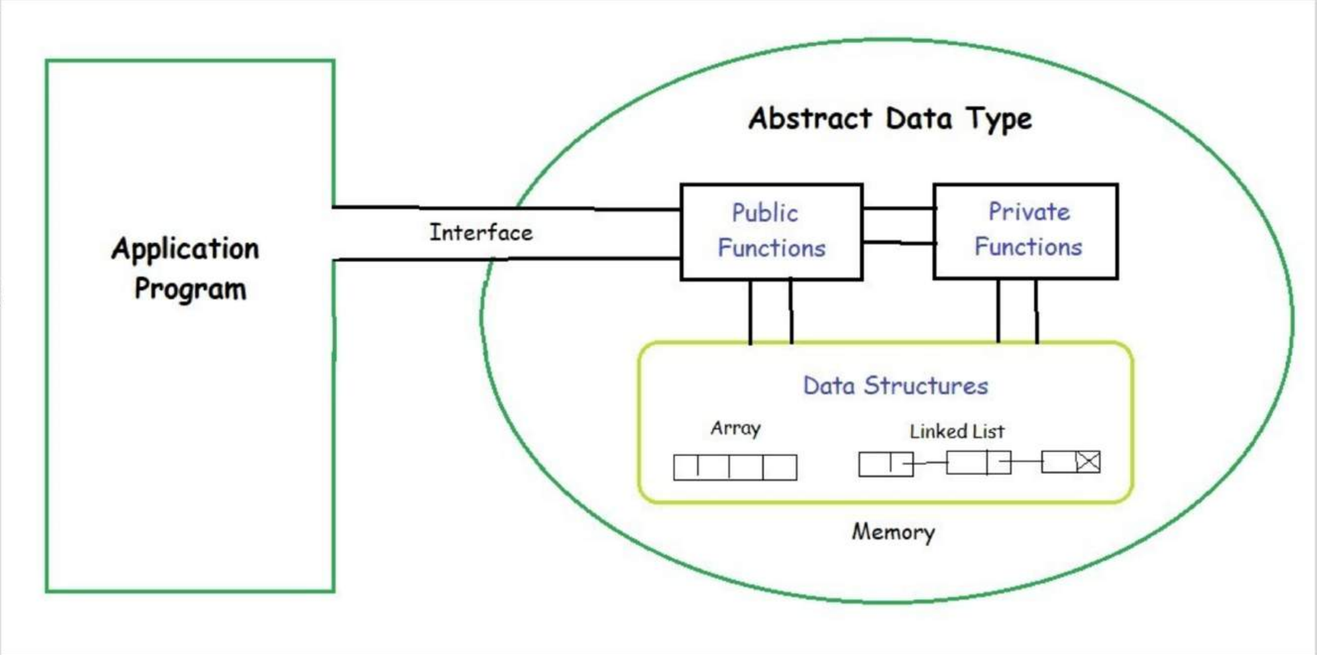
Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.



Stack ADT

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.

The program allocates memory for the data and address is passed to the stack ADT.

The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.

The stack head structure also contains a pointer to top and count of number of entries currently in stack.

push() – Insert an element at one end of the stack called top.

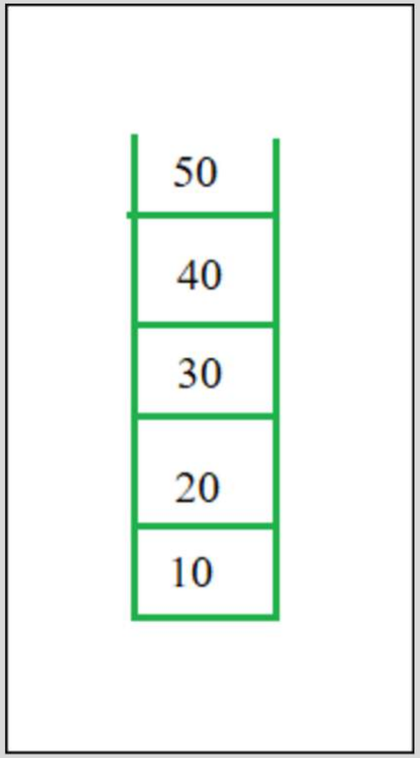
pop() – Remove and return the element at the top of the stack, if it is not empty.

peek() – Return the element at the top of the stack without removing it, if the stack is not empty.

size() – Return the number of elements in the stack.

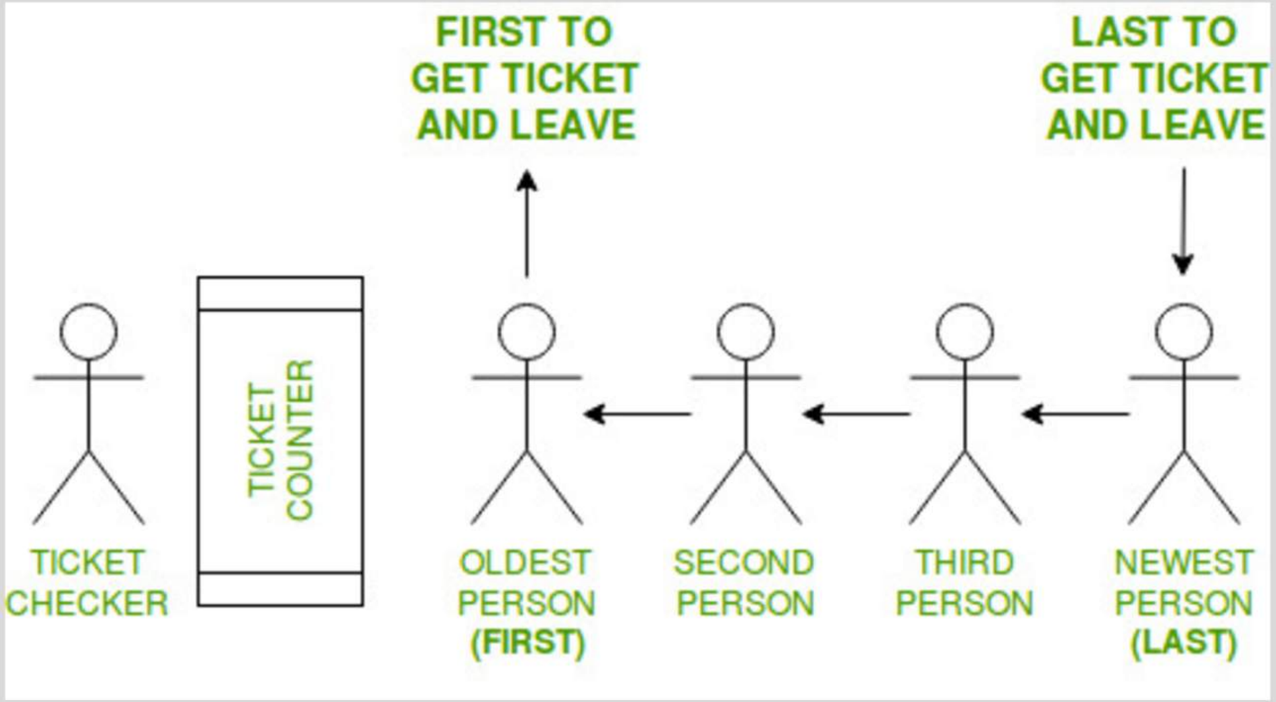
isEmpty() – Return true if the stack is empty, otherwise return false.

isFull() – Return true if the stack is full, otherwise return false.



Concrete data structure for a First In First out (FIFO) queue.

FIFO is an abbreviation for first in, first out. It is a method for handling data structures where the first element is processed first and the newest element is processed last.



In this example, following things are to be considered:

There is a ticket counter where people come, take tickets and go.

People enter a line (queue) to get to the Ticket Counter in an organized manner.

The person to enter the queue first, will get the ticket first and leave the queue.

The person entering the queue next will get the ticket after the person in front of him

In this way, the person entering the queue last will the tickets last

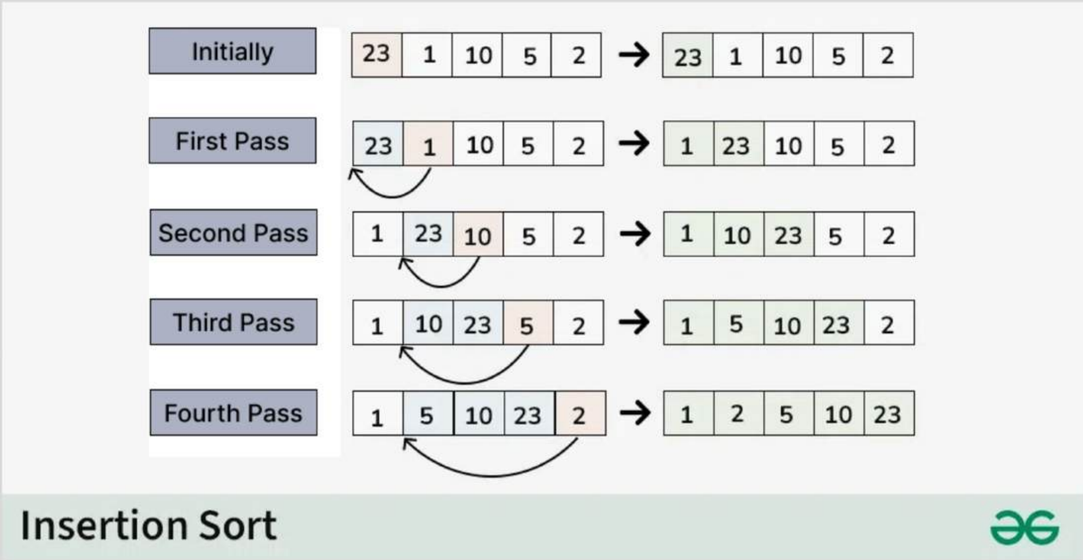
Therefore, the First person to enter the queue gets the ticket first and the Last person to enter the queue gets the ticket last.

Queue

```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args)
    {
        Queue<Integer> q = new LinkedList<>();

        // Adds elements {0, 1, 2, 3, 4} to queue
        for (int i = 0; i < 5; i++)
            q.add(i);

        // Display contents of the queue.
        System.out.println("Elements of queue-" + q);

        // To remove the head of queue.
        // In this the oldest element '0' will be removed
        int removedele = q.remove();
        System.out.println("removed element-" + removedele);

        System.out.println(q);

        // To view the head of queue
        int head = q.peek();
        System.out.println("head of queue-" + head);

        // Rest all methods of collection interface,
        // Like size and contains can be used with this
        // implementation.
        int size = q.size();
        System.out.println("Size of queue-" + size);
```

Two sorting algorithms
Insertion Sort Algorithm

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.
We start with second element of the array as first element in the array is assumed to be sorted.
Compare second element with the first element and check if the second element is smaller then swap them.
Move to the third element and compare it with the first two elements and put at its correct position
Repeat until the entire array is sorted.



```java
public class InsertionSort {
    /* Function to sort array using insertion sort */
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            int key = arr[i];
            int j = i - 1;

            /* Move elements of arr[0..i-1], that are
               greater than key, to one position ahead
               of their current position */
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    // Driver method
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6 };

        InsertionSort ob = new InsertionSort();
        ob.sort(arr);

        printArray(arr);
```

Illustration

arr = {23, 1, 10, 5, 2}

Initial:

- *Current element is* ***23***
- *The first element in the array is assumed to be sorted.*
- *The sorted part until* ***0th*** *index is :* ***[23]***

First Pass:

- *Compare* ***1*** *with* ***23*** *(current element with the sorted part).*
- *Since* ***1*** *is smaller, insert* ***1*** *before* ***23*** *.*
- *The sorted part until* ***1st*** *index is:* ***[1, 23]***

Second Pass:

- *Compare* ***10*** *with* ***1*** *and* ***23*** *(current element with the sorted part).*
- *Since* ***10*** *is greater than* ***1*** *and smaller than* ***23*** *, insert* ***10*** *between* ***1*** *and* ***23*** *.*
- *The sorted part until* ***2nd*** *index is:* ***[1, 10, 23]***

Third Pass:

- *Compare* ***5*** *with* ***1*** *,* ***10*** *, and* ***23*** *(current element with the sorted part).*
- *Since* ***5*** *is greater than* ***1*** *and smaller than* ***10*** *, insert* ***5*** *between* ***1*** *and* ***10***
- *The sorted part until* ***3rd*** *index is* ***:*** ***[1, 5, 10, 23]***

Fourth Pass:

- *Compare* ***2*** *with* ***1, 5, 10*** *, and* ***23*** *(current element with the sorted part).*
- *Since* ***2*** *is greater than* ***1*** *and smaller than* ***5*** *insert* ***2*** *between* ***1*** *and* ***5*** *.*
- *The sorted part until* ***4th*** *index is:* ***[1, 2, 5, 10, 23]***

Final Array:

- *The sorted array is:* ***[1, 2, 5, 10, 23]***

Advantages of Insertion Sort:
Simple and easy to implement.
Stable sorting algorithm.
Efficient for small lists and nearly sorted lists.
Space-efficient as it is an in-place algorithm.
Adoptive. the number of inversions is directly proportional to number of swaps. For example, no swapping happens for a sorted array and it takes O(n) time only.

Disadvantages of Insertion Sort:
Inefficient for large lists.
Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort Algorithm work?
QuickSort works on the principle of divide and conquer, breaking down the problem into smaller sub-problems.
There are mainly three steps in the algorithm:
Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

How the QuickSort algorithm works.

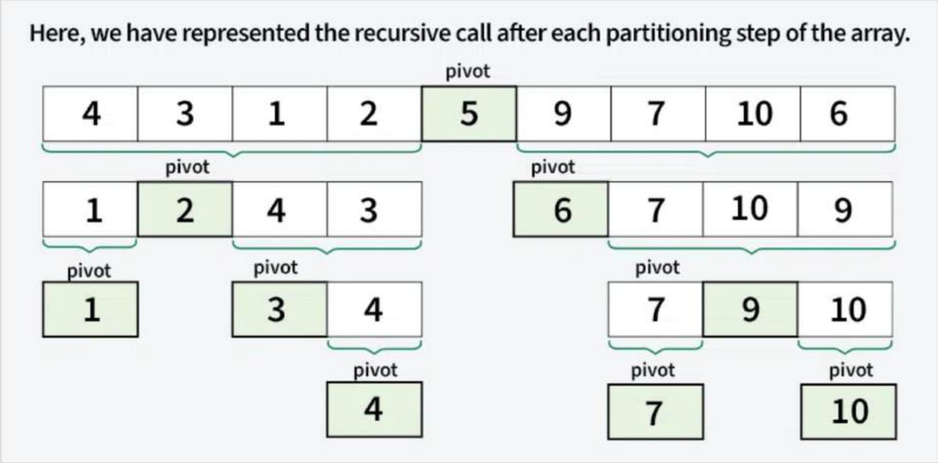
Choice of Pivot

There are many different choices for picking pivots.

Always pick the first (or last) element as a pivot. The below implementation is picks the last element as pivot. The problem with this approach is it ends up in the worst case when array is already sorted.

Pick a random element as a pivot. This is a preferred approach because it does not have a pattern for which the worst case happens.

Pick the median element is pivot. This is an ideal approach in terms of time complexity as we can find median in linear time and the partition function will always divide the input array into two halves. But it is low on average as median finding has high constants.



Partition Algorithm

The key process in quickSort is a partition(). There are three common algorithms to partition. All these algorithms have O(n) time complexity.

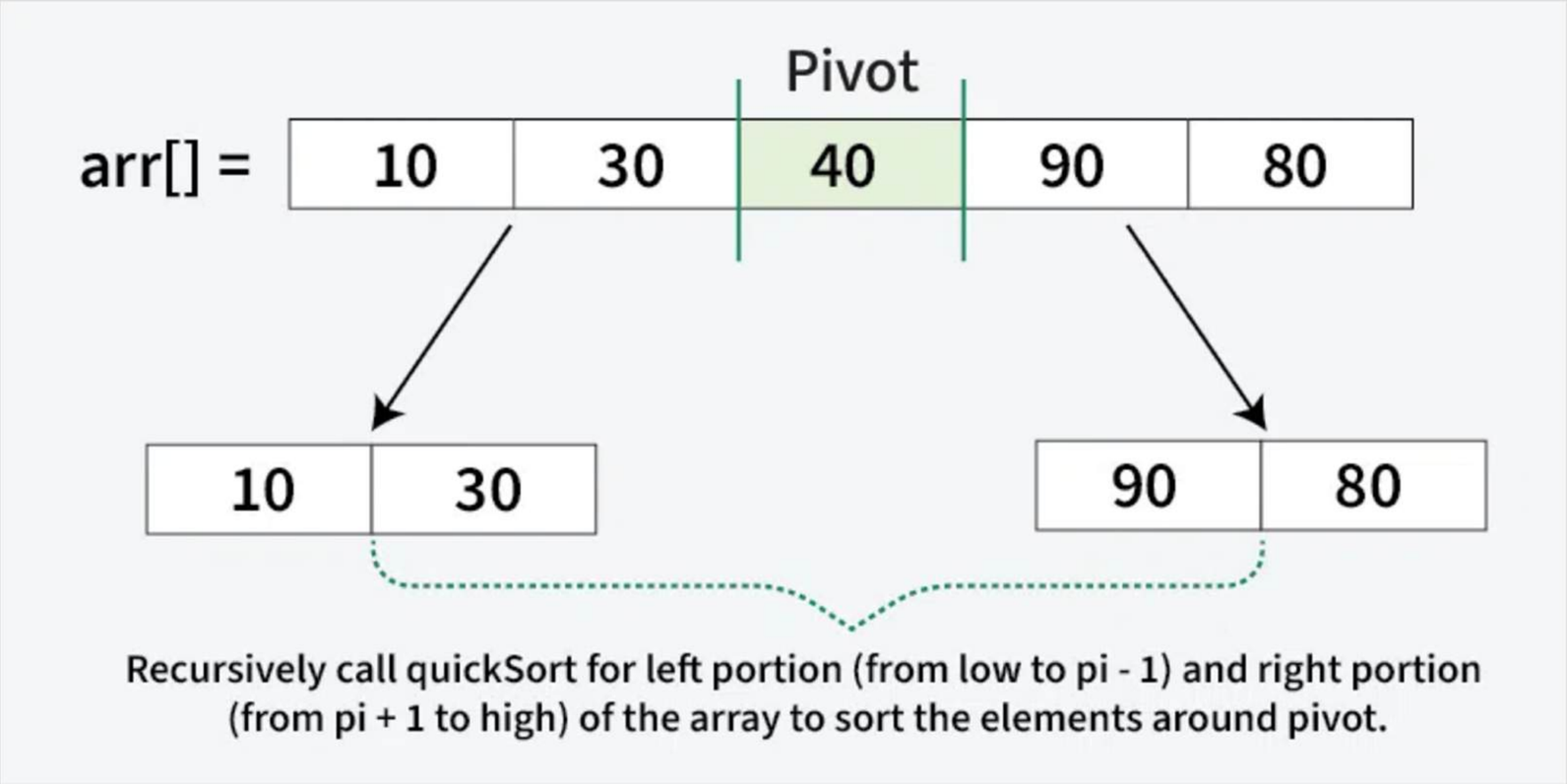
Naive Partition: Here we create copy of the array. First put all smaller elements and then all greater. Finally we copy the temporary array back to original array. This requires O(n) extra space.

Lomuto Partition: We have used this partition in this article. This is a simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this article because of its simplicity.

Hoare's Partition: This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned. Please refer Hoare's vs Lomuto for details.

Illustration of QuickSort Algorithm

In the previous step, we looked at how the partitioning process rearranges the array based on the chosen pivot. Next, we apply the same method recursively to the smaller sub-arrays on the left and right of the pivot. Each time, we select new pivots and partition the arrays again. This process continues until only one element is left, which is always sorted. Once every element is in its correct position, the entire array is sorted.



Below image illustrates, how the recursive method calls for the smaller sub-arrays on the left and right of the pivot:
Quick Sort is a crucial algorithm in the industry, but there are other sorting algorithms that may be more optimal in different cases. To gain a deeper understanding of sorting and other essential algorithms, check out our course Tech Interview 101 – From DSA to System Design . This course covers almost every standard algorithm and more.

Advantages of Quick Sort

It is a divide-and-conquer algorithm that makes it easier to solve problems.

It is efficient on large data sets.

It has a low overhead, as it only requires a small amount of memory to function.

It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.

Fastest general purpose algorithm for large data when stability is not required.

It is tail recursive and hence all the tail call optimization can be done.

Disadvantages of Quick Sort

It has a worst-case time complexity of O(n2), which occurs when the pivot is chosen poorly.

It is not a good choice for small data sets.

It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Shortest Path Algorithm in Computer Network

In between sending and receiving data packets from the sender to the receiver, it will go through many routers and subnets. So as a part of increasing the efficiency in routing the data packets and decreasing the traffic, we must find the shortest path. In this article, we are discussing the shortest path algorithms.

Shortest Path Routing

It refers to the algorithms that help to find the shortest path between a sender and receiver for routing the data packets through the network in terms of shortest distance, minimum cost, and minimum time.
It is mainly for building a graph or subnet containing routers as nodes and edges as communication lines connecting the nodes.
Hop count is one of the parameters that is used to measure the distance.
Hop count: It is the number that indicates how many routers are covered. If the hop count is 6, there are 6 routers/nodes and the edges connecting them.
Another metric is a geographic distance like kilometers.
We can find the label on the arc as the function of bandwidth, average traffic, distance, communication cost, measured delay, mean queue length, etc.

Dijkstra’s Algorithm

The Dijkstra’s Algorithm is a greedy algorithm that is used to find the minimum distance between a node and all other nodes in a given graph. Here we can consider node as a router and graph as a network. It uses weight of edge .ie, distance between the nodes to find a minimum distance route.

Algorithm:

1: Mark the source node current distance as 0 and all others as infinity.

2: Set the node with the smallest current distance among the non-visited nodes as the current node.

3: For each neighbor, N, of the current node:

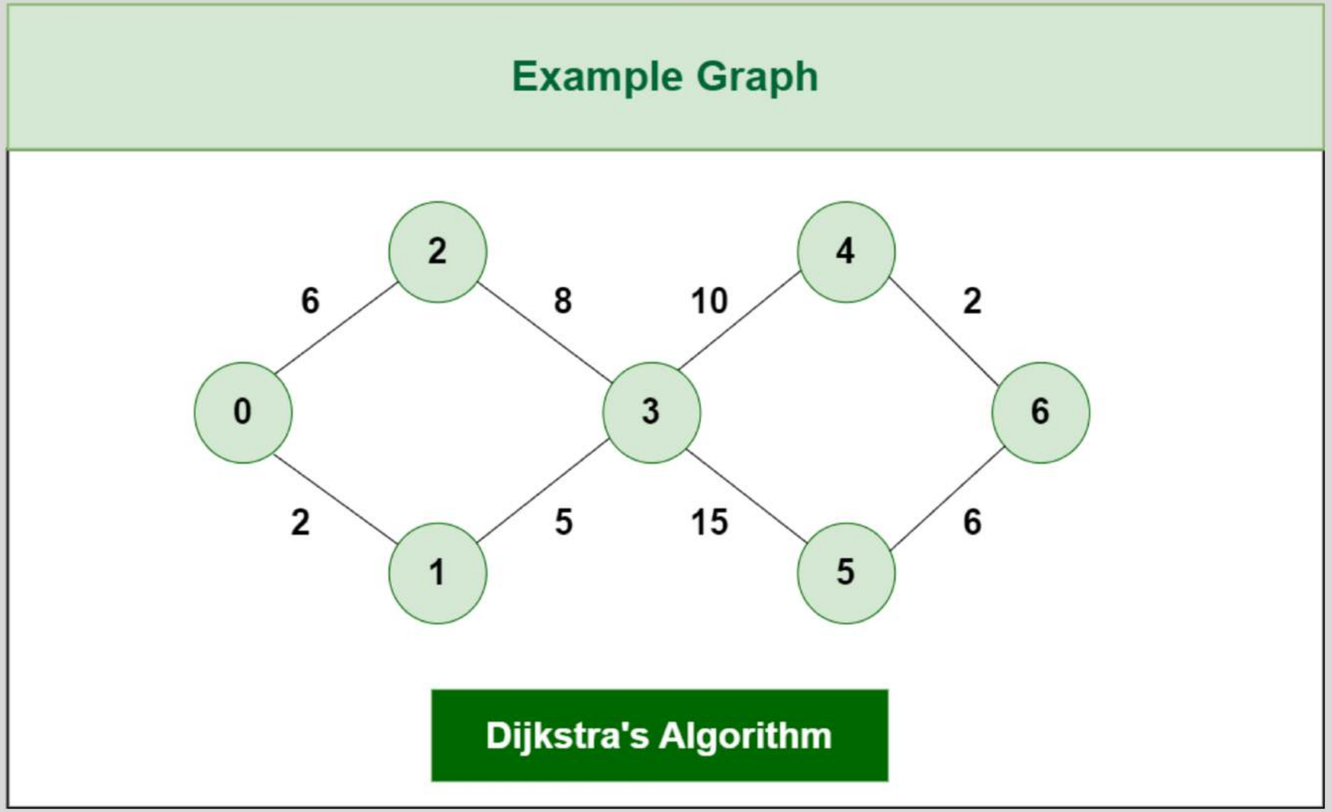
Calculate the potential new distance by adding the current distance of the current node with the weight of the edge connecting the current node to N.

If the potential new distance is smaller than the current distance of node N, update N's current distance with the new distance.

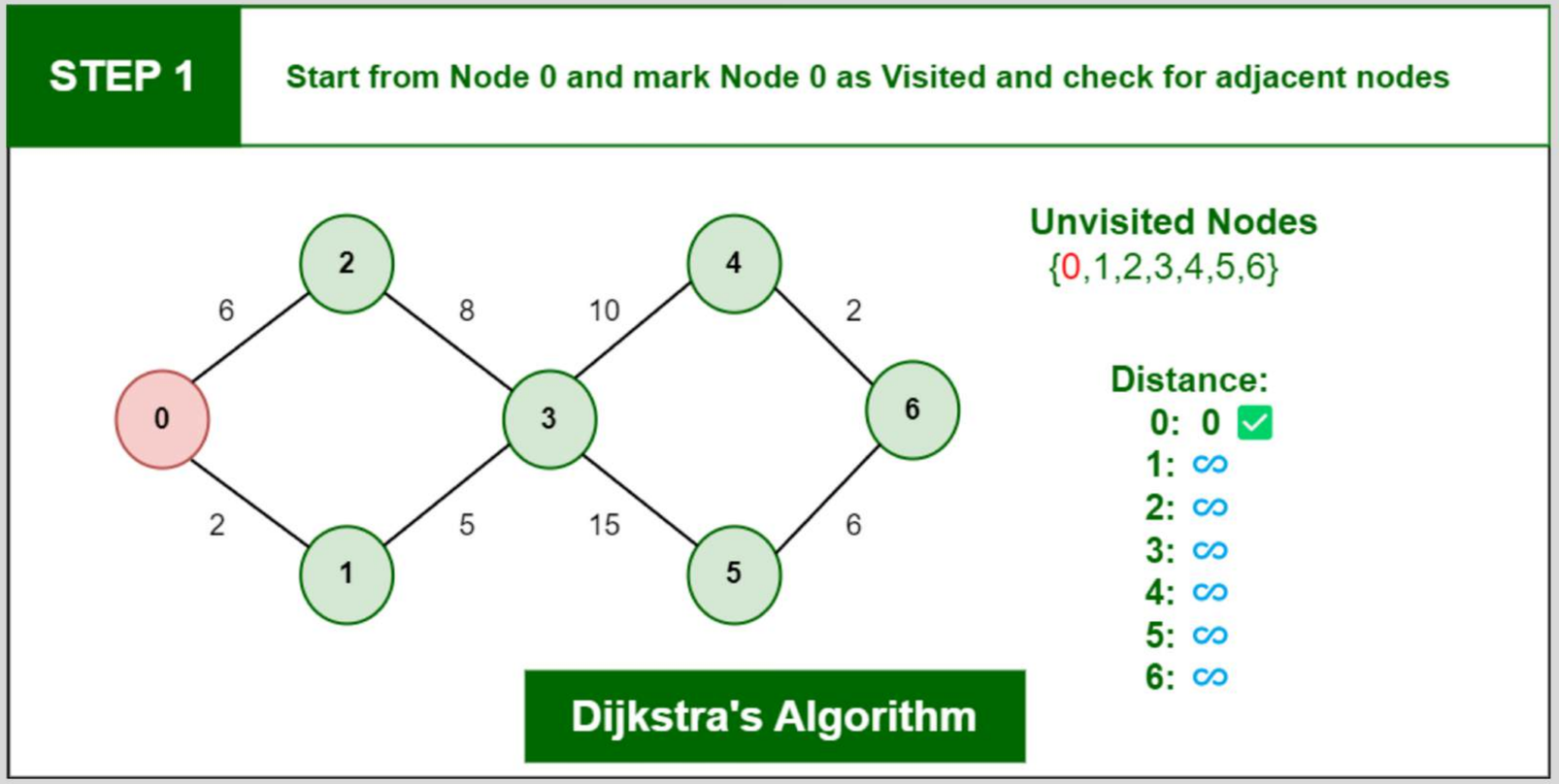
4: Make the current node as visited node.

5: If we find any unvisited node, go to step 2 to find the next node which has the smallest current distance and continue this process.

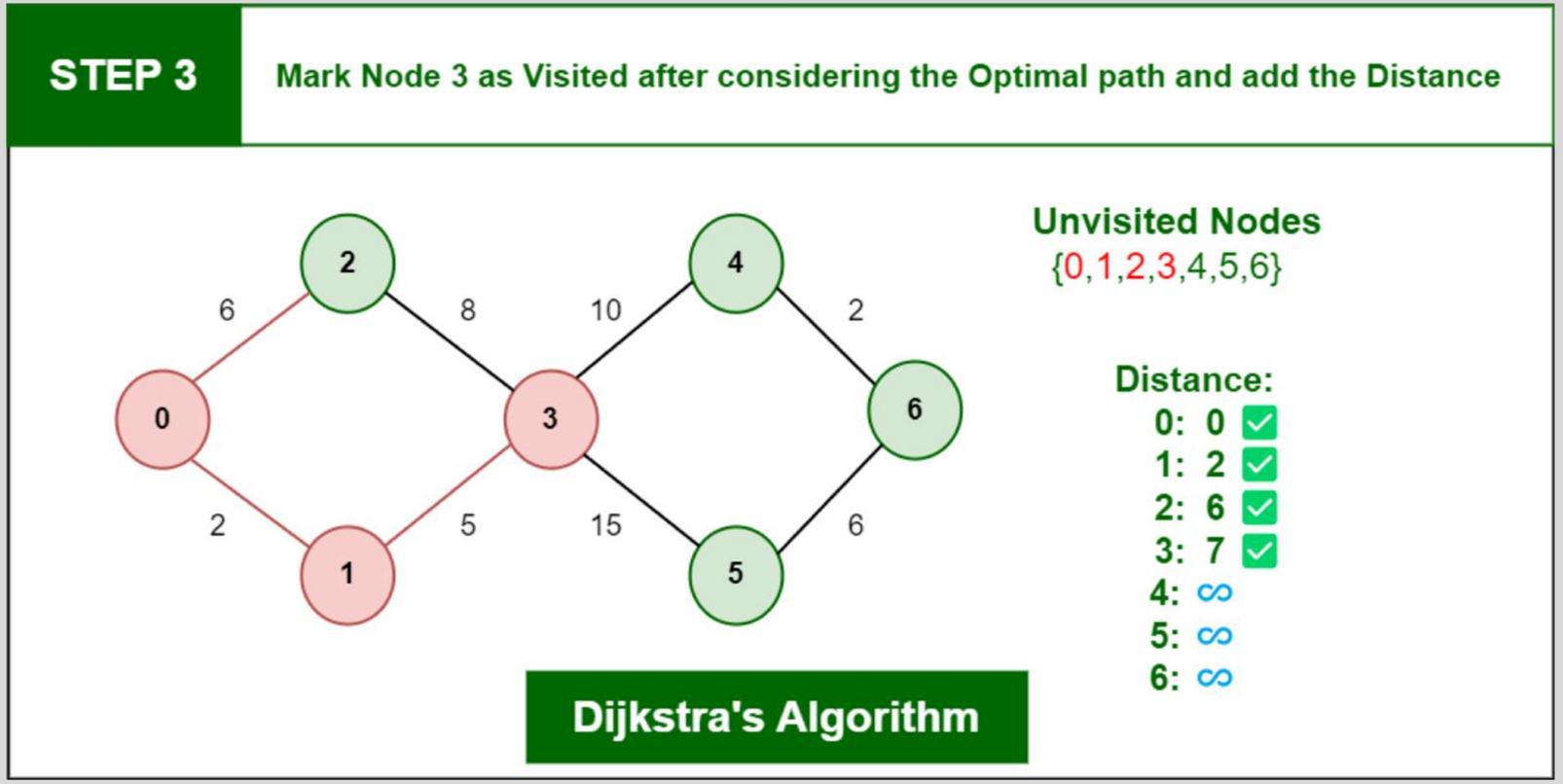
Example:
Consider the graph G:



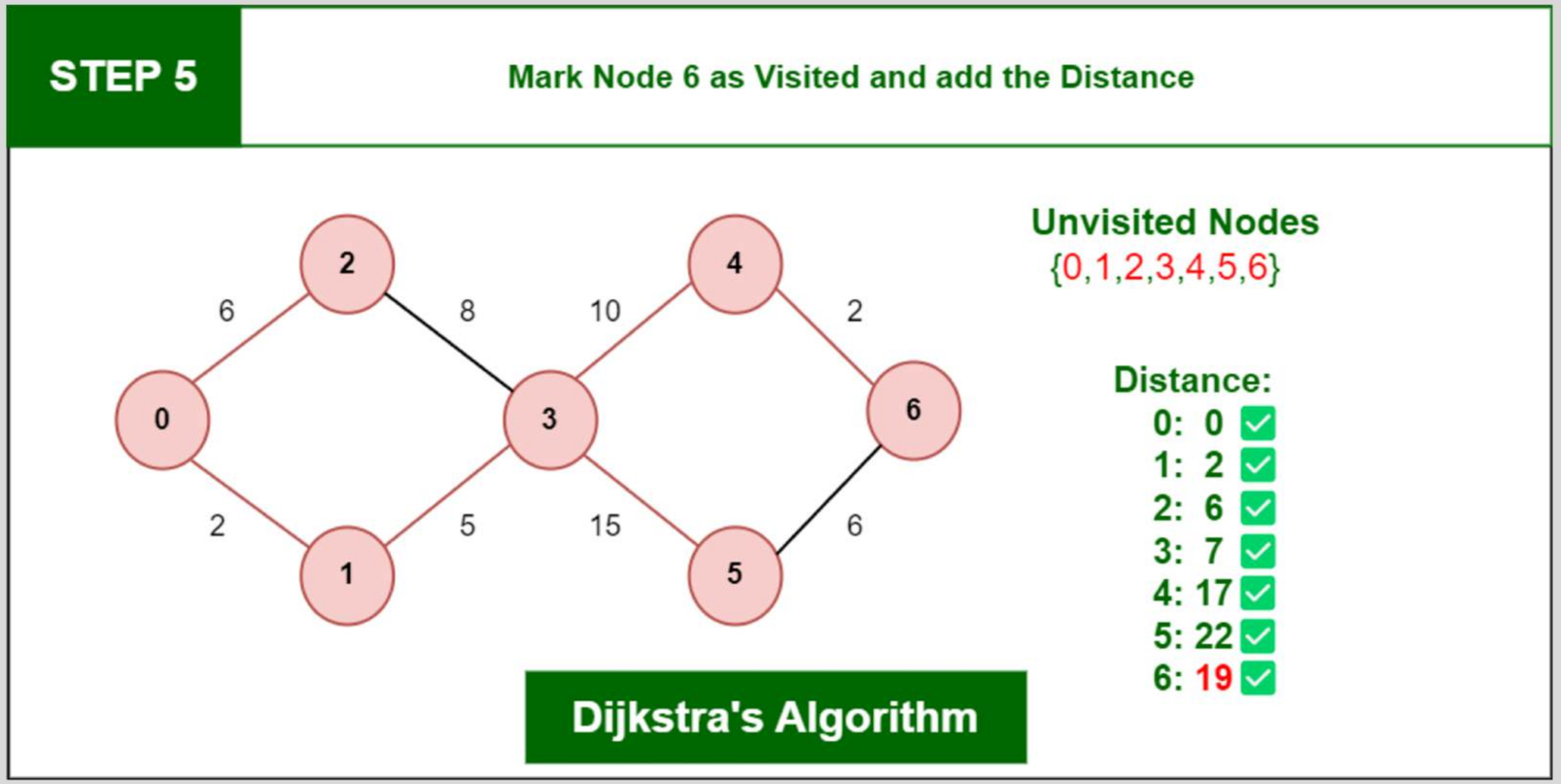
Now,we will start normalising graph one by one starting from node 0.



Nearest neighbour of 0 are 2 and 1 so we will normalize them first .

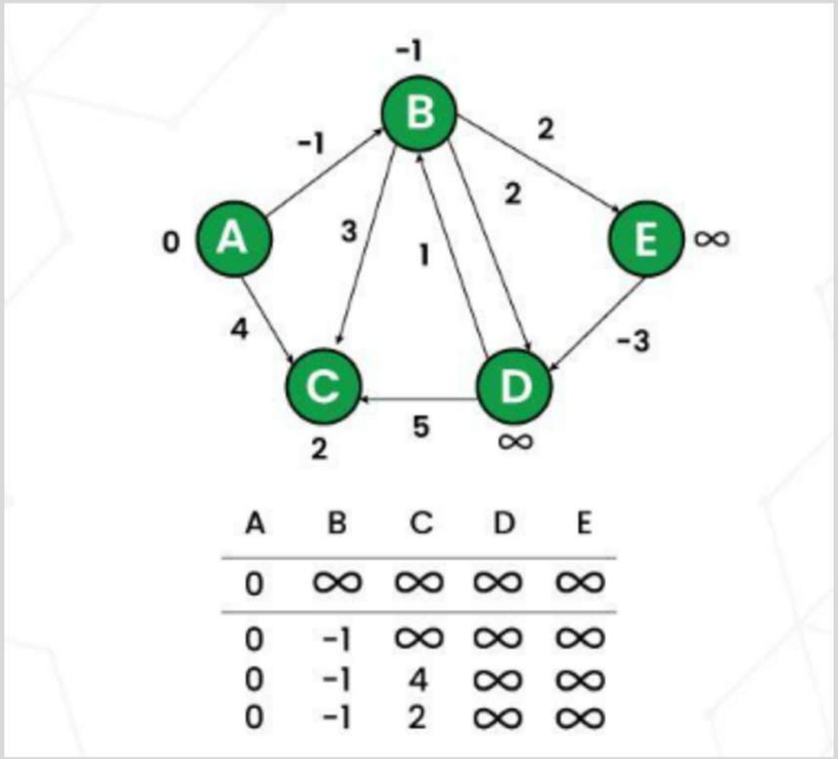


Similarly we will normalize other node considering it should not form a cycle and will keep track in visited nodes.



Bellman Ford’s Algorithm

The Bell man Ford’s algorithm is a single source graph search algorithm which help us to find the shortest path between a source vertex and any other vertex in a give graph. We can use it in both weighted and unweighted graphs. This algorithm is slower than Dijkstra's algorithm and it can also use negative edge weight.



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

Algorithm

1: First we Initialize all vertices v in a distance array dist[] as INFINITY.

2: Then we pick a random vertex as vertex 0 and assign dist[0] =0.

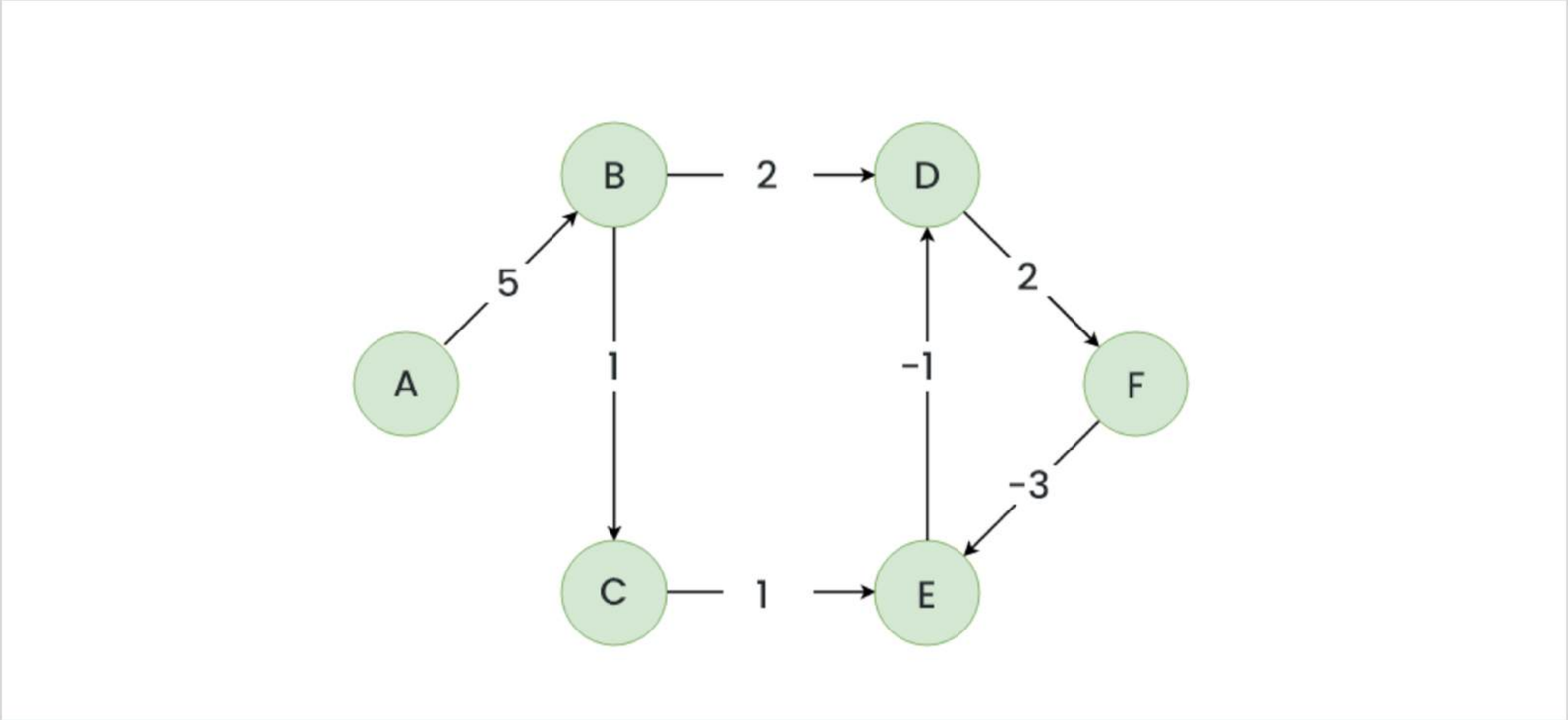
3: Then iteratively update the minimum distance to each node (dist[v]) by comparing it with the sum of the distance from the source node (dist[u]) and the edge weight (weight) N-1 times.

4: To identify the presence of negative edge cycles, with the help of following cases do one more round of edge relaxation.

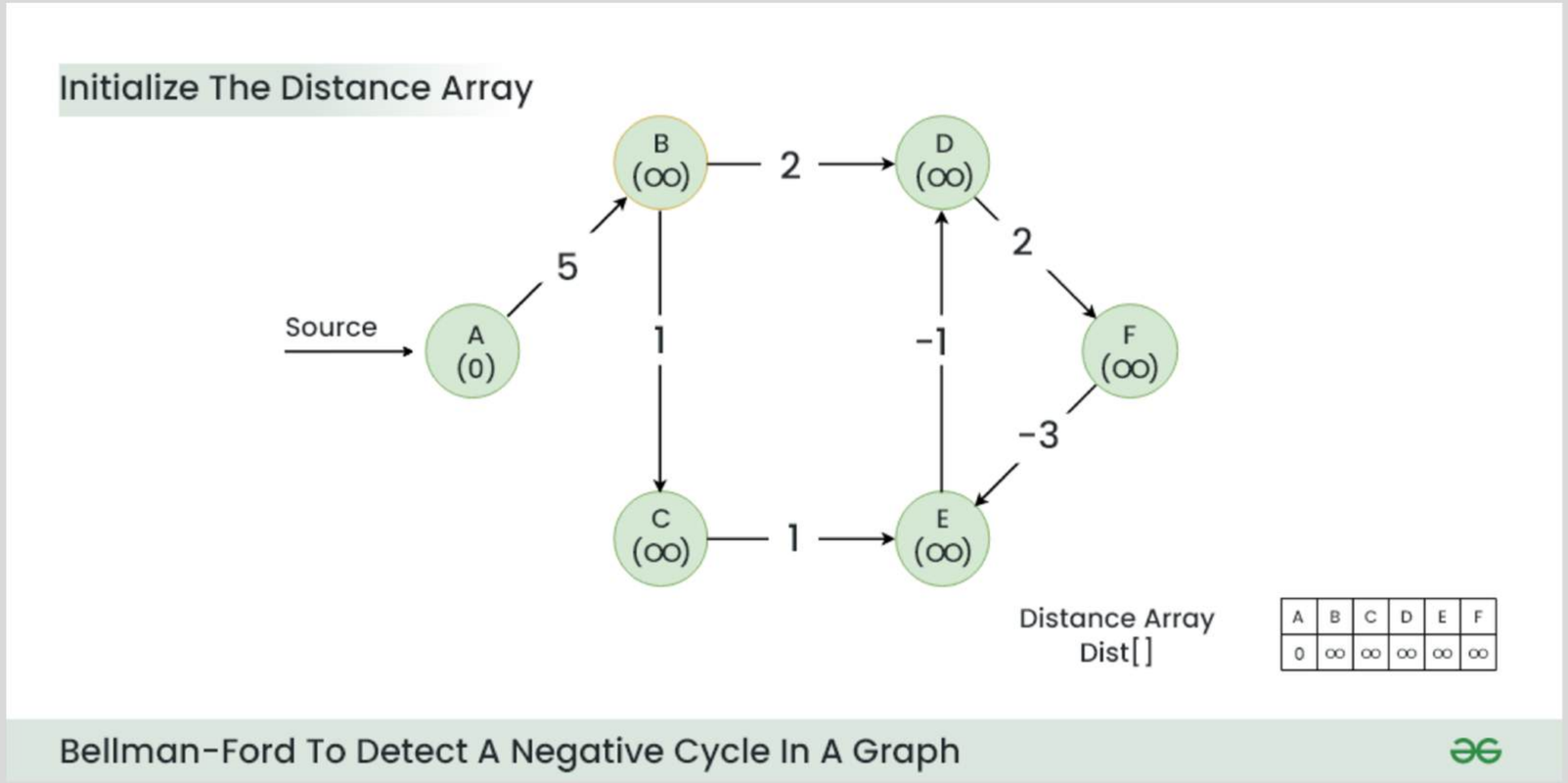
We can say that a negative cycle exists if for any edge uv the sum of distance from the source node (dist[u]) and the edge weight (weight) is less than the current distance to the largest node(dist[v])

It indicates the absence of negative edge cycle if none of the edges satisfies case1.

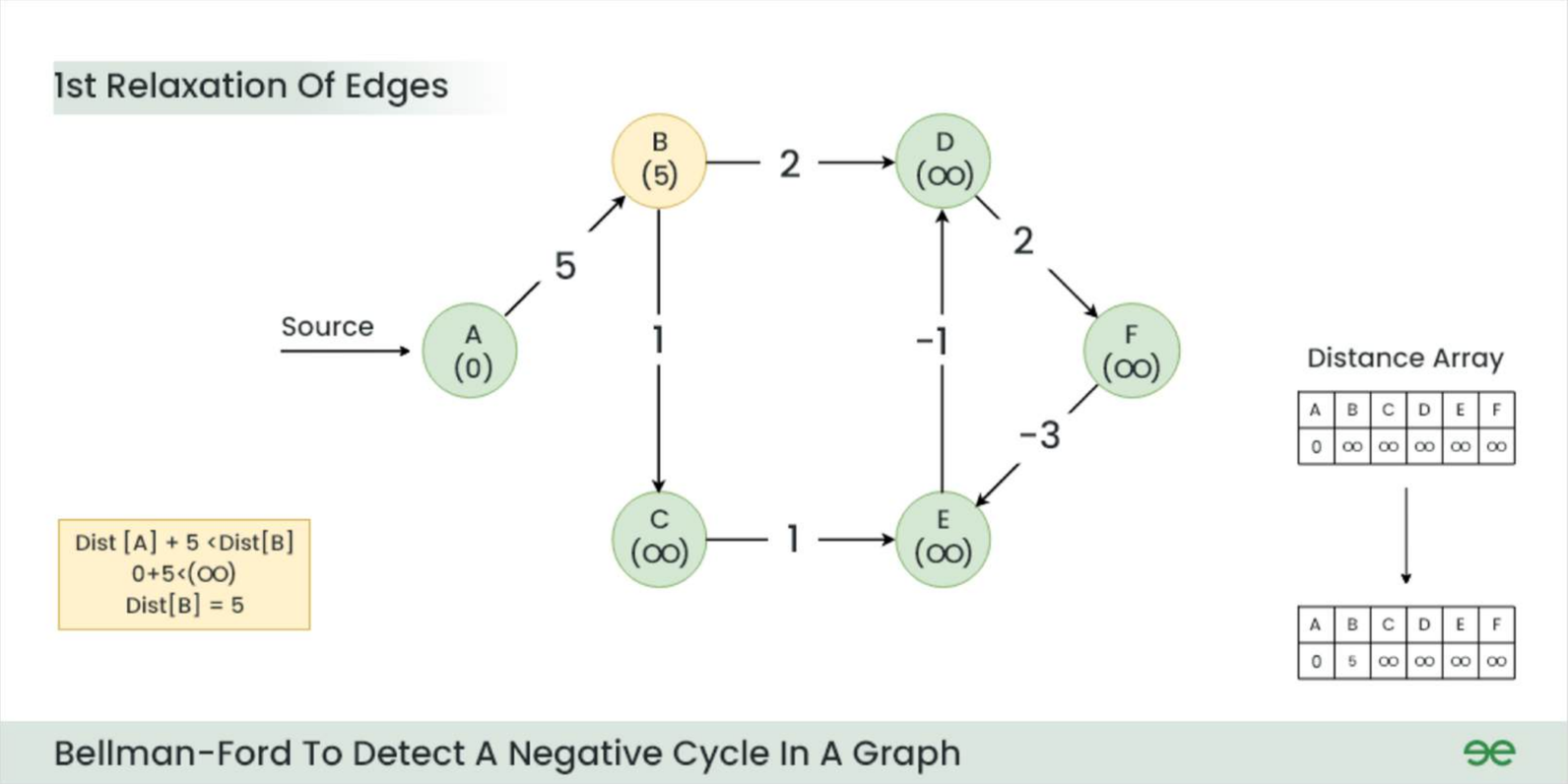
Example: Bellman ford detecting negative edge cycle in a graph.
Consider the Graph G:



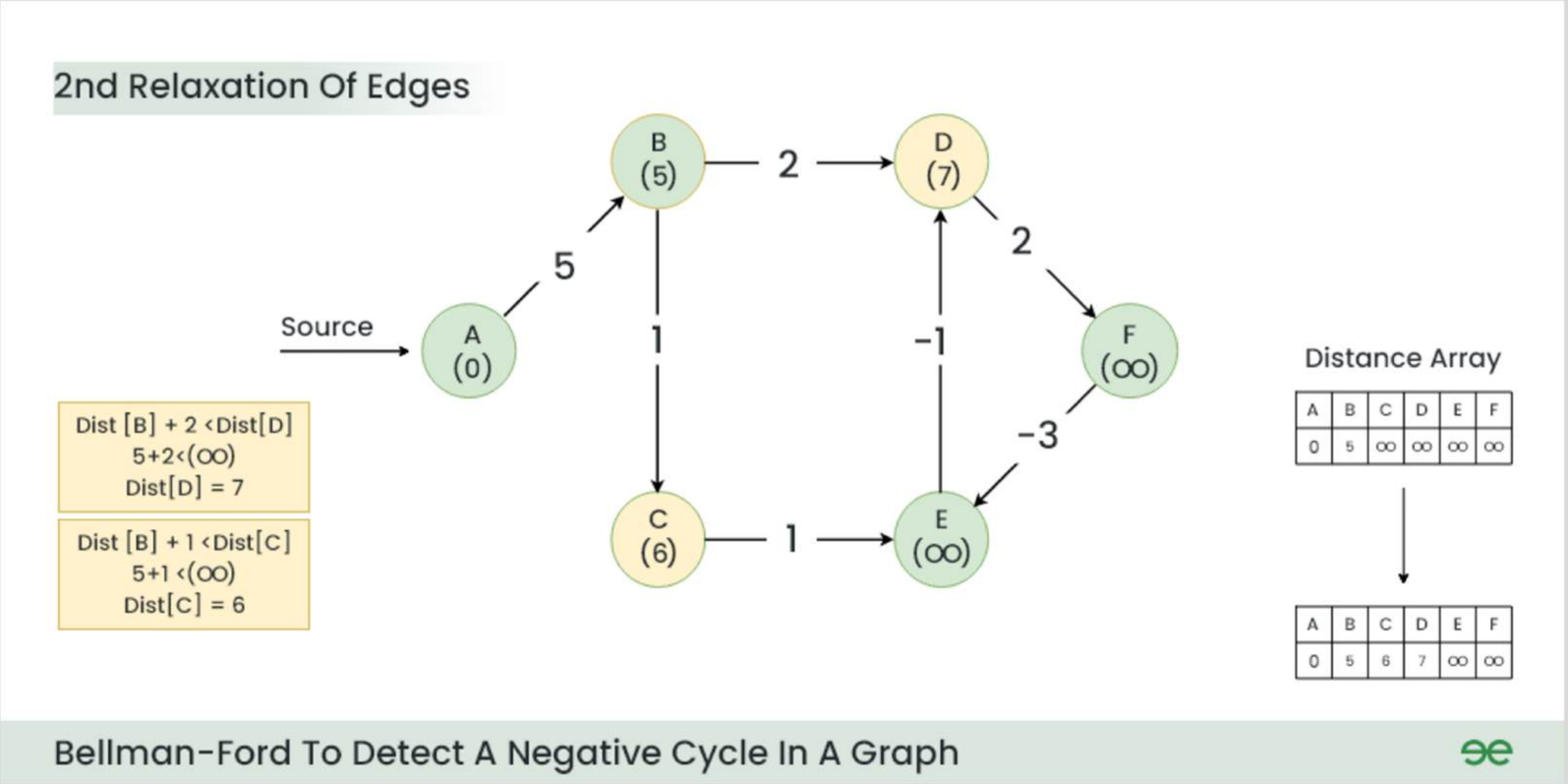
Step 1:


Initialize The Distance Array
B
(∞)
2
D
(∞)
2
5
Source
A
(0)
1
-1
F
(∞)
-3
C
(∞)
1
E
(∞)
Distance Array
Dist[ ]
A B C D E F
0 ∞ ∞ ∞ ∞ ∞
Bellman-Ford To Detect A Negative Cycle In A Graph

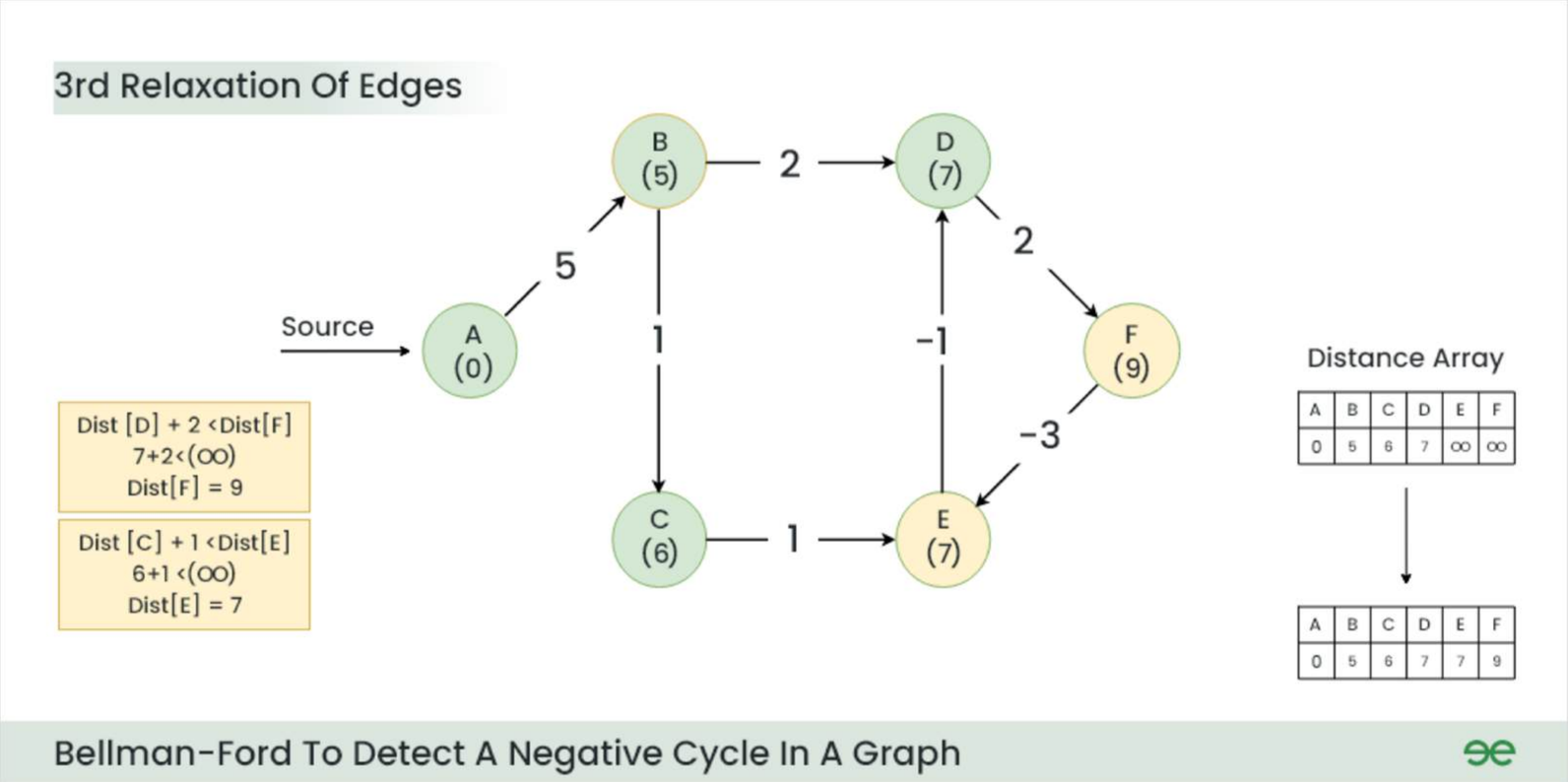
Step 2:


1st Relaxation Of Edges
B
(5)
2
D
(∞)
5
2
Source
A
(0)
1
-1
F
(∞)
-3
C
(∞)
1
E
(∞)
Dist [A] + 5 <Dist[B]
0+5<(∞)
Dist[B] = 5
Distance Array
A B C D E F
0 ∞ ∞ ∞ ∞ ∞
A B C D E F
0 5 ∞ ∞ ∞ ∞
Bellman-Ford To Detect A Negative Cycle In A Graph

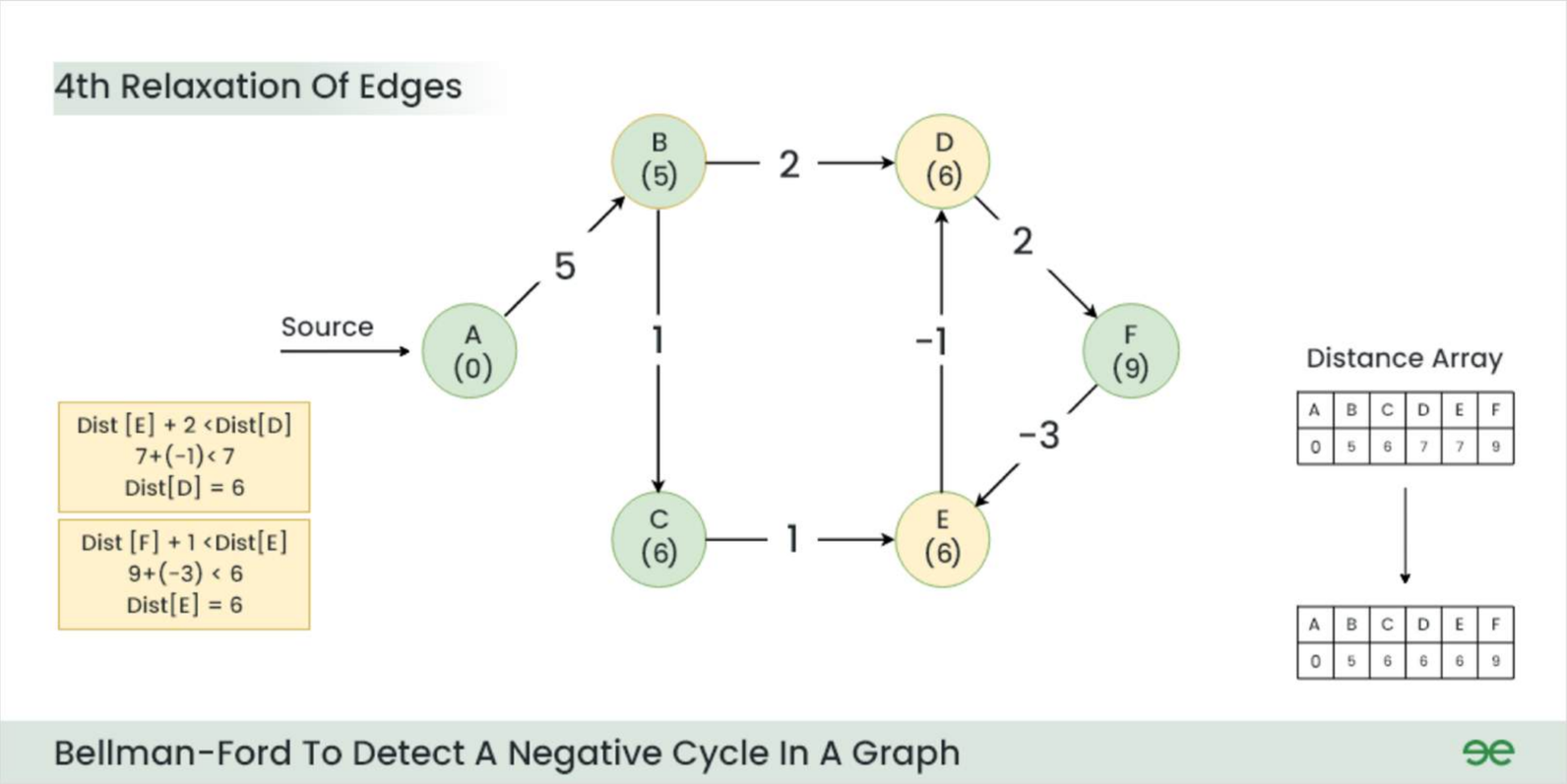
Step 3:


2nd Relaxation Of Edges
B
(5)
2
D
(7)
5
2
Source
A
(0)
1
-1
F
(∞)
-3
C
(6)
1
E
(∞)
Dist [B] + 2 <Dist[D]
5+2<(∞)
Dist[D] = 7
Dist [B] + 1 <Dist[C]
5+1 <(∞)
Dist[C] = 6
Distance Array
A B C D E F
0 5 ∞ ∞ ∞ ∞
A B C D E F
0 5 6 7 ∞ ∞
Bellman-Ford To Detect A Negative Cycle In A Graph

Step 4:


3rd Relaxation Of Edges
B
(5)
2
D
(7)
5
2
Source
A
(0)
1
-1
F
(9)
-3
C
(6)
1
E
(7)
Dist [D] + 2 <Dist[F]
7+2<(∞)
Dist[F] = 9
Dist [C] + 1 <Dist[E]
6+1 <(∞)
Dist[E] = 7
Distance Array
A B C D E F
0 5 6 7 ∞ ∞
A B C D E F
0 5 6 7 7 9
Bellman-Ford To Detect A Negative Cycle In A Graph

Step 5:



Outcome: The graph contains a negative cycle in the path from node D to node F and then to node E.