DATA STRUCTURES (ITPC-203)

Trees and Graphs - Conclusion

Dr. Sanga Chaki

Department of Information Technology

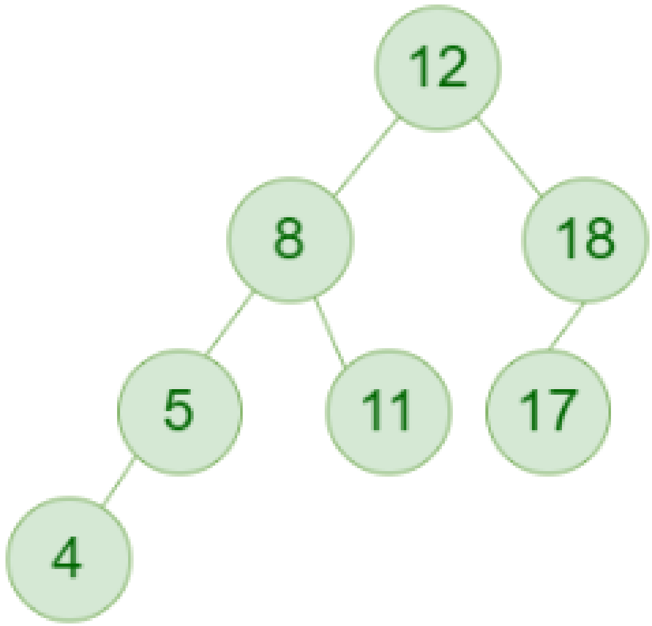
Dr. B. R. Ambedkar National Institute of Technology, Jalandhar

Contents

1. AVL Trees
2. m-way Search Trees
3. B Trees
4. Heaps
5. Heap Sort
6. Connected Component,
7. Spanning Trees,
8. Minimum Cost Spanning Trees: Prims and Kruskal algorithm.

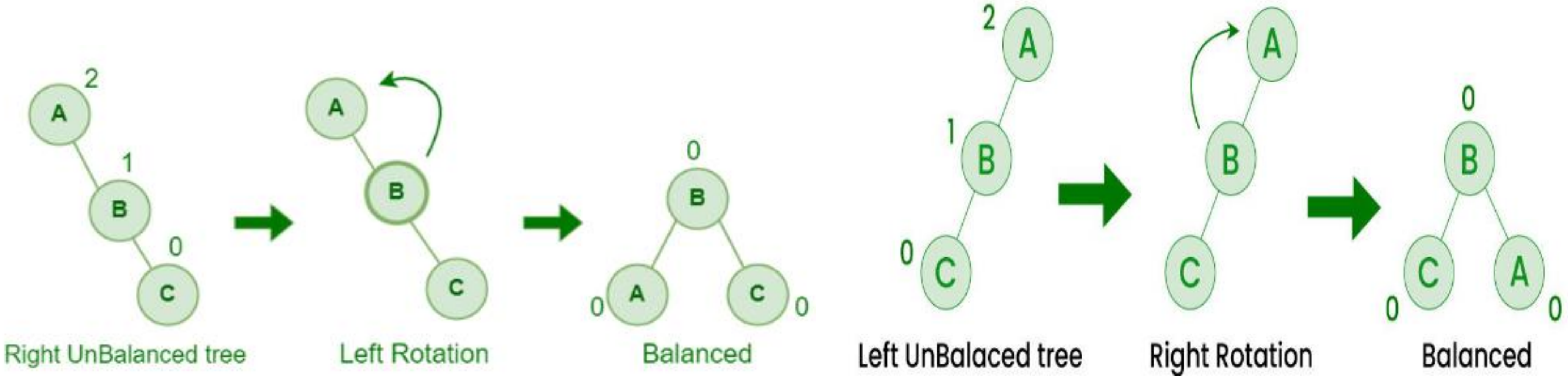
AVL Trees

1. An AVL tree defined as a self-balancing Binary Search Tree (BST) where
   - the difference between heights of left and right subtrees for any node cannot be more than one.
2. The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node.
3. This BST is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.
4. Operations: Insertion, Deletion, Searching



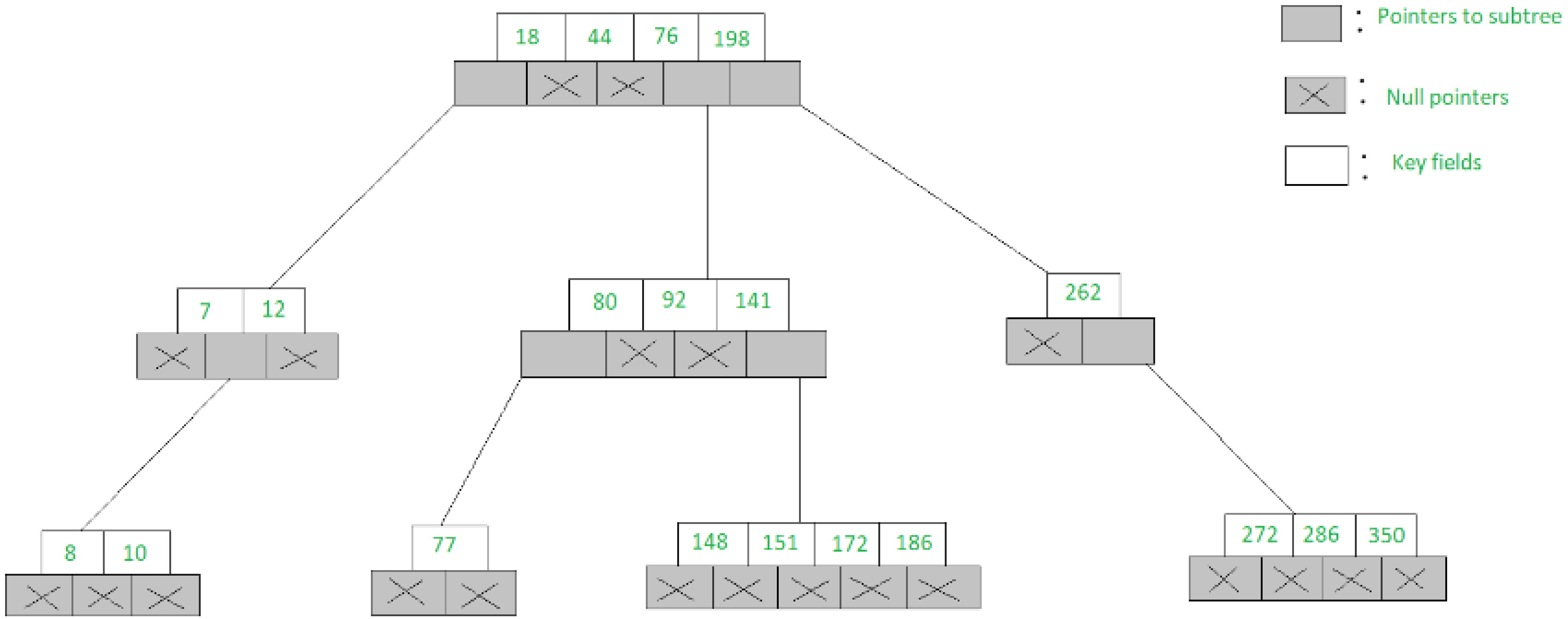
AVL Trees

1. In any case, the difference in balance between left and right subtree should not be more than one.
2. If it is more than one, we call the tree to be out of balance – rotate nodes to restore the balance.



m-way Search Trees

1. The m-way search trees are multi-way trees which are generalized versions of binary trees where each node contains multiple elements.
2. In an m-Way tree of order m, each node contains a maximum of m – 1 elements and m children.
3. What is the advantage/disadvantage?



B Trees (Balanced Tree)

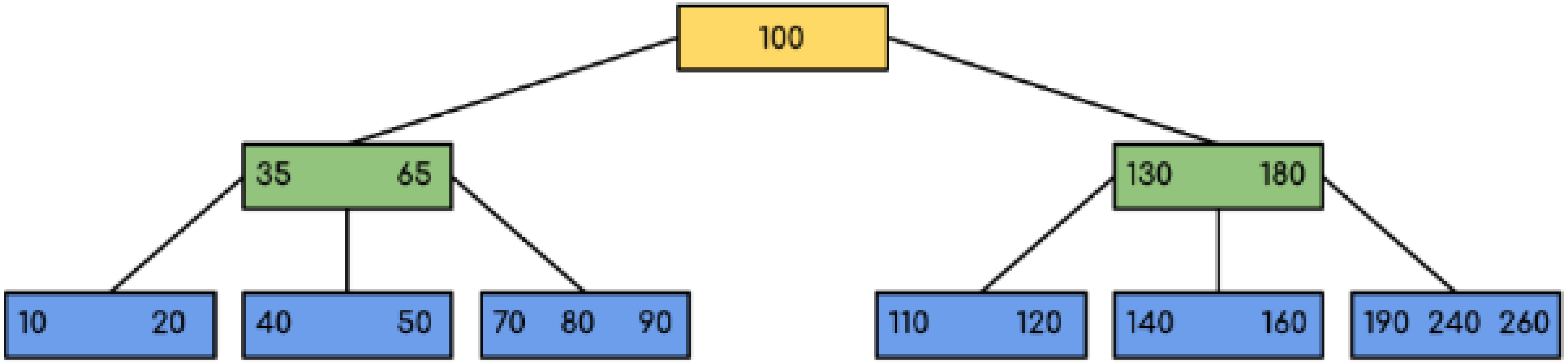
1. Used for storing and searching large amounts of data
2. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor
    1. Thus a shallower height.
3. This shallow height leads to less disk I/O, which results in faster search and insertion operations
4. A self-balancing tree data structure
5. Maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.
6. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

B Trees (Balanced Tree) - Properties

1. All leaves are at the same level.
2. B-Tree is defined by the term minimum degree ‘t‘. The value of ‘t‘ depends upon disk block size.
3. Every node except the root must contain at most t-1 keys. The root may contain a minimum of 1 key.
4. All nodes (including root) may contain at most (2*t – 1) keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(log n).
9. Insertion of a Node in B-Tree happens only at Leaf Node.

B Trees (Balanced Tree) - Example

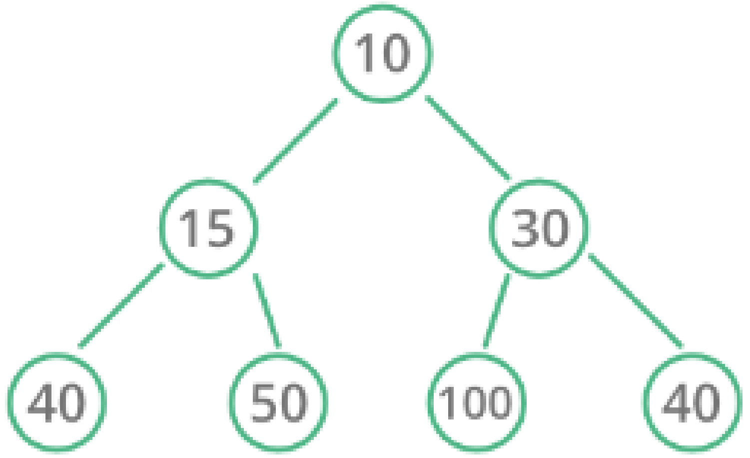
1. Following is an example of a B-Tree of minimum order 5



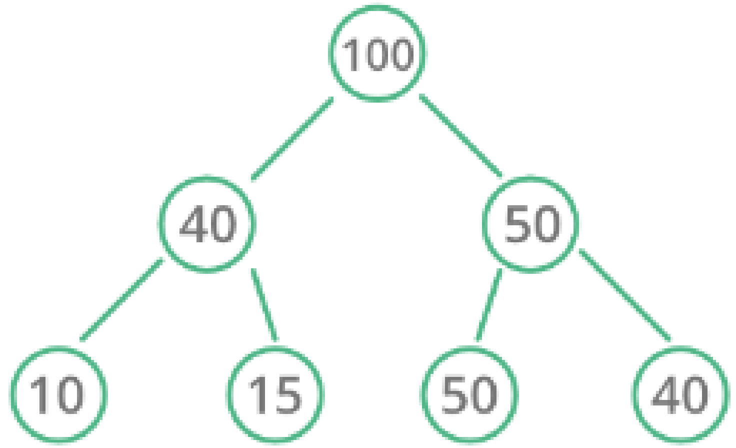
Heaps and Heap Sort

Heap

1. A Heap is a Tree-based data structure in which is a complete binary tree.
2. Generally, Heaps can be of two types:
   a) Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present in all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.
   b) Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.



Min Heap



Max Heap

Operations of Heap Data Structure

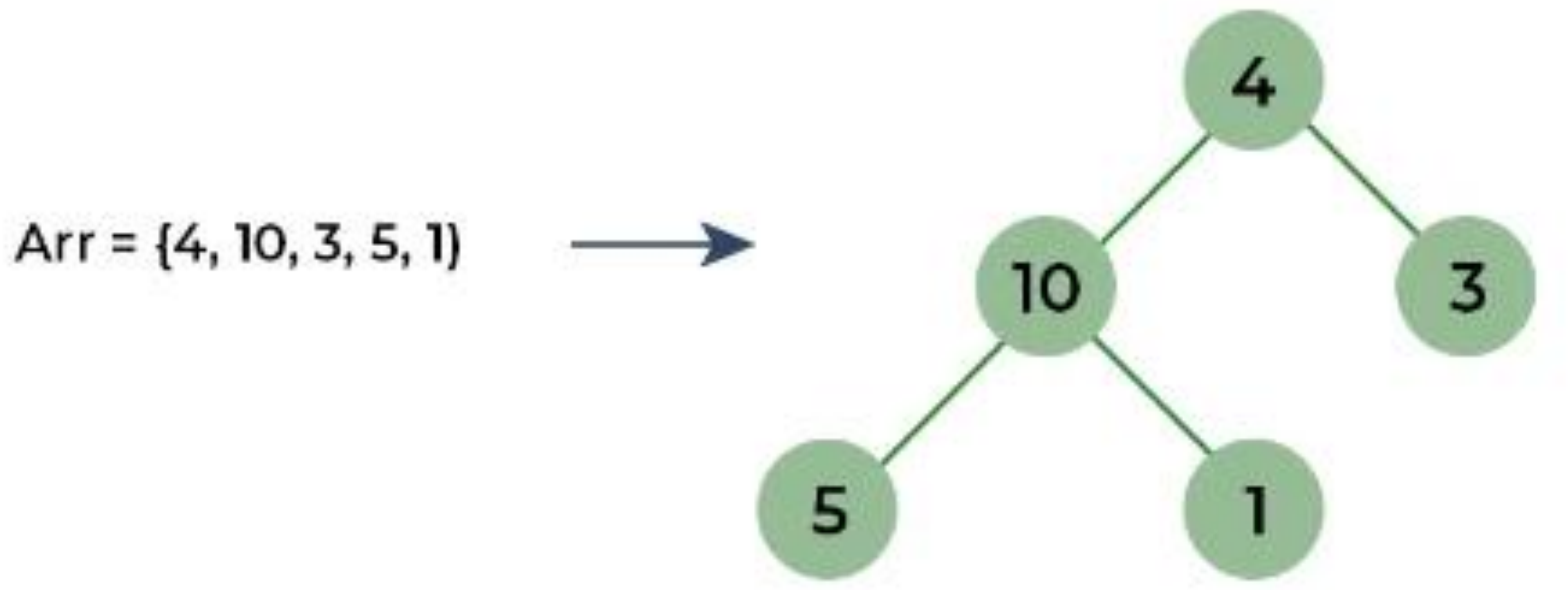
1. Heapify: a process of creating a heap from an array.
2. Insertion: process to insert an element in existing heap
3. Deletion: deleting the top element of the heap or the highest priority element, and then reorganizing the heap and returning the element
4. Peek: to check or find the first/top element of the heap.

Heap Sort

1. Heap sort is a comparison-based sorting technique based on Binary Heap data structure.
2. Steps:
   I. Build a maxheap from the given input array.
   II. Repeat the following steps until the heap contains only one element:
      a) Swap the root element of the heap (which is the largest element) with the last element of the heap.
      b) Remove the last element of the heap (which is now in the correct position).
      c) Heapify the remaining elements of the heap.
   III. The sorted array is obtained by reversing the order of the elements in the input array.

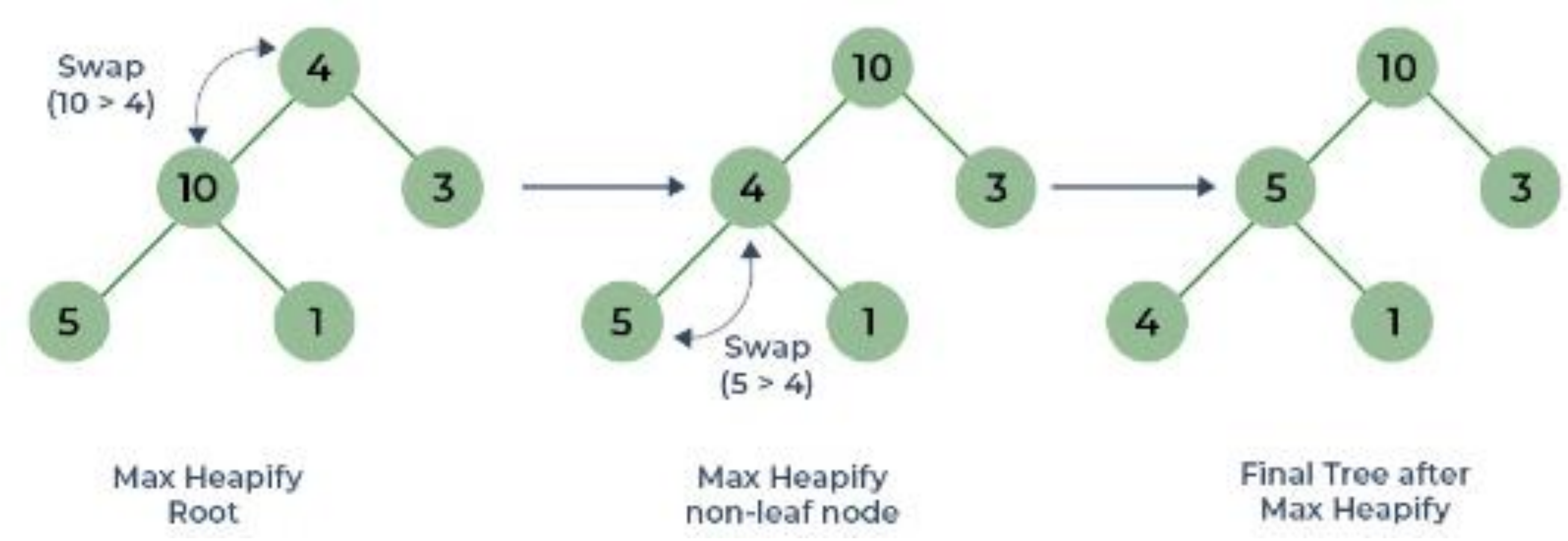
Heap Sort - Example

1. Consider the array: arr[] = {4, 10, 3, 5, 1}.
2. Build heap out of given array elements



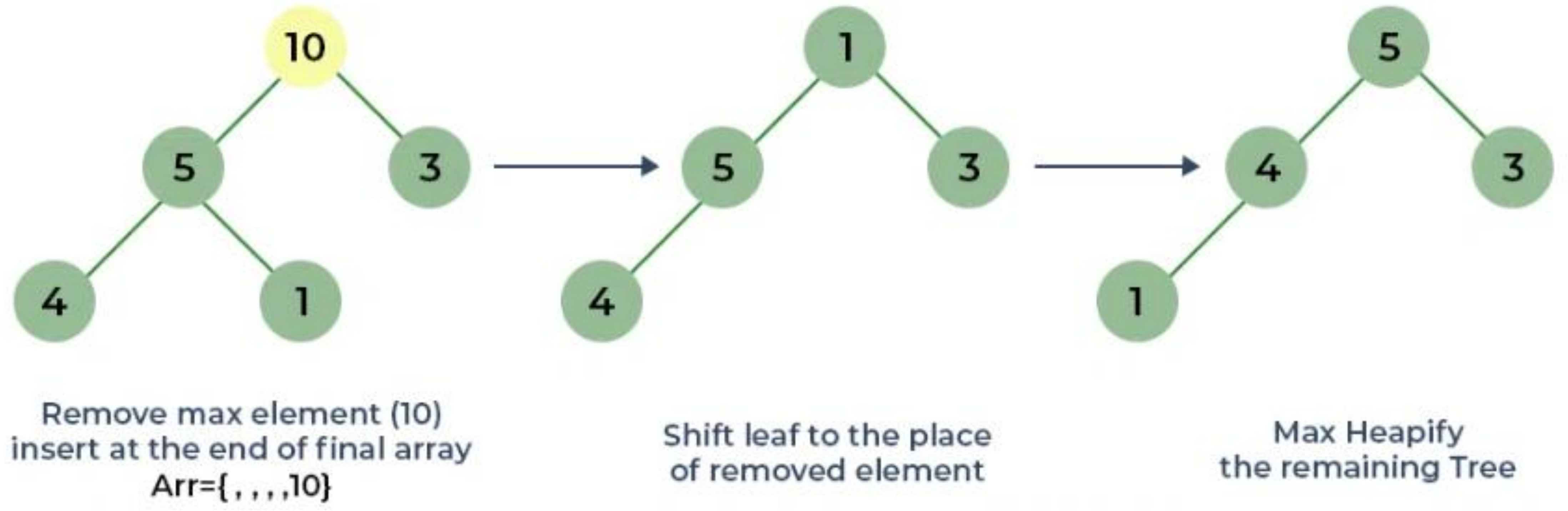
Heap Sort - Example

1. Build maxheap from the heap - parent node should always be greater than or equal to the child nodes – swap elements wherever necessary



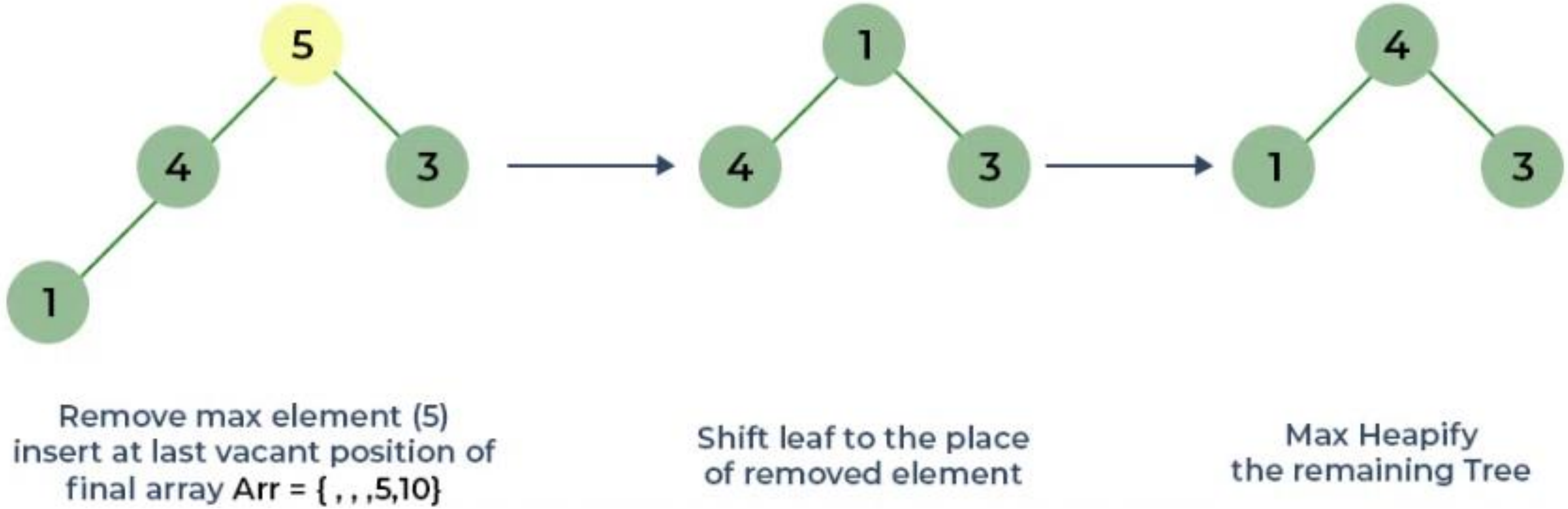
Heap Sort - Example

1. Remove the maximum element – store as last elements of sorted array
2. Swap the last element of the maxheap into the root
3. Consider the remaining elements and transform it into a max heap.



Heap Sort - Example

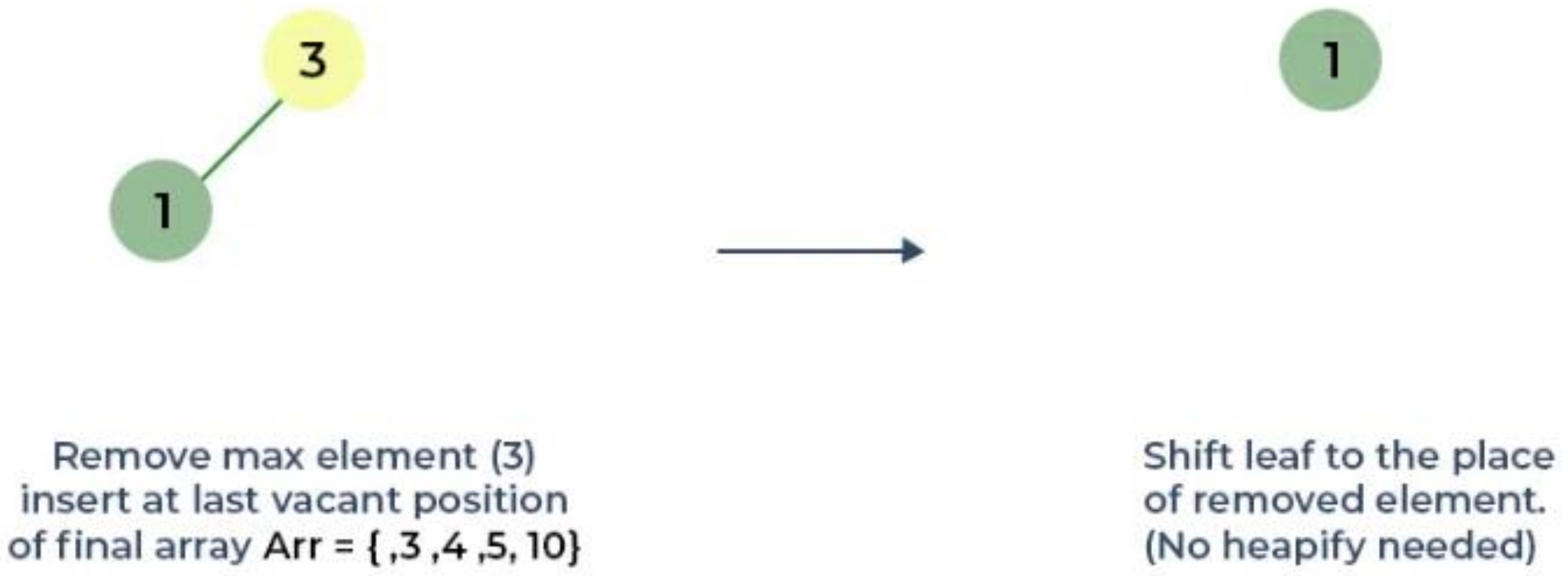
1. Repeat the above steps until only 1 node left



2. Remove 4 next

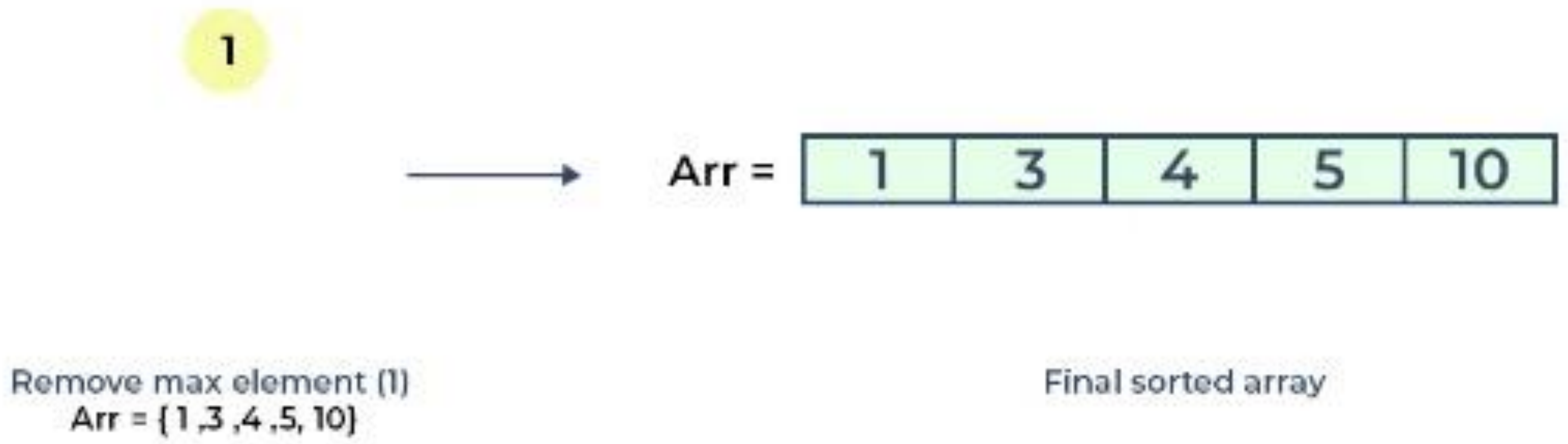
Heap Sort - Example

1. Repeat the above steps until only 1 node left – where Heapify is not needed anymore.



Heap Sort - Example

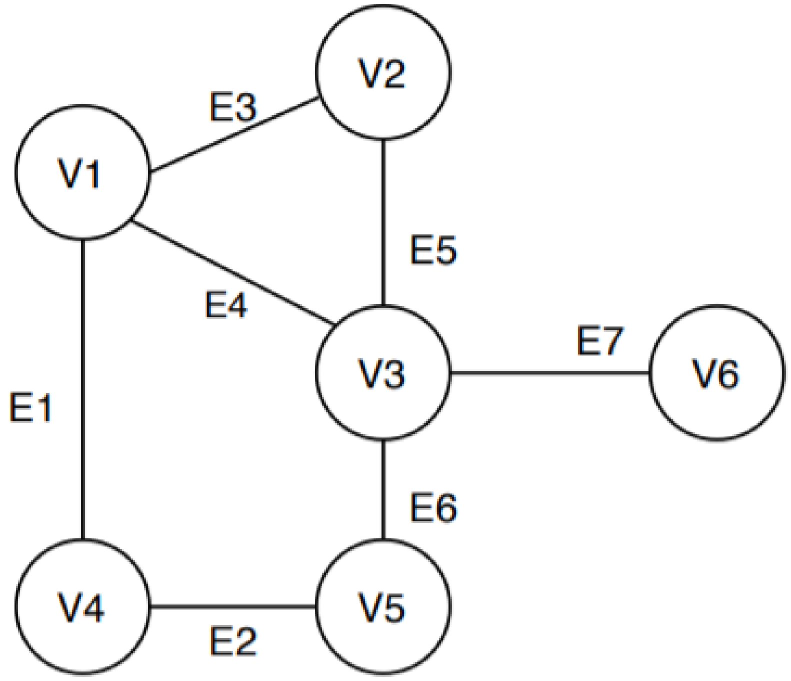
1. The final array is sorted.



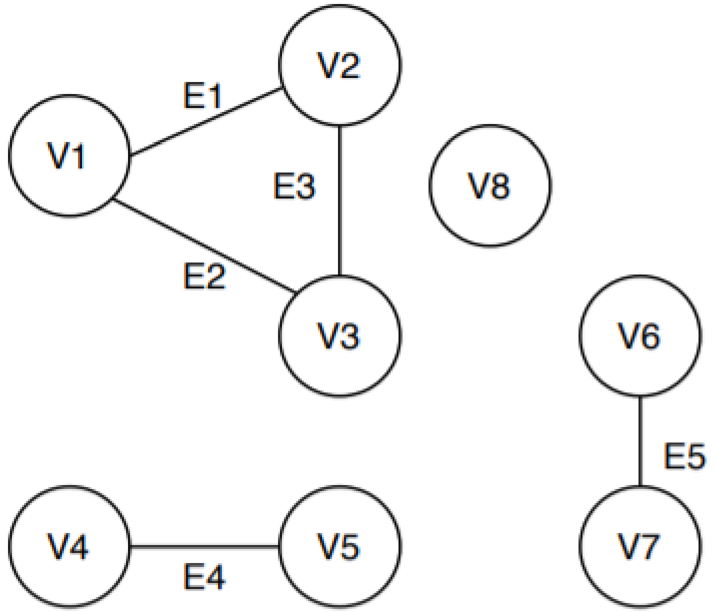
Connected Components in Graph

Connected Components in a Graph

1. A set of vertices in a graph that are linked to each other by paths
2. Traversal from one node to another is possible in a connected component of a graph



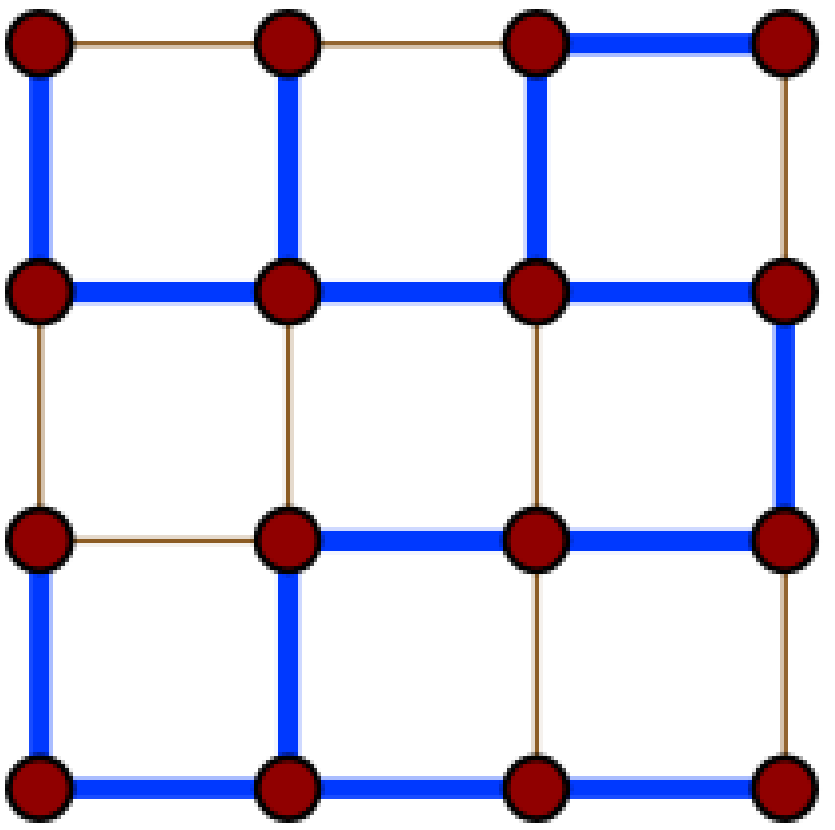
One connected component



Four Connected Components

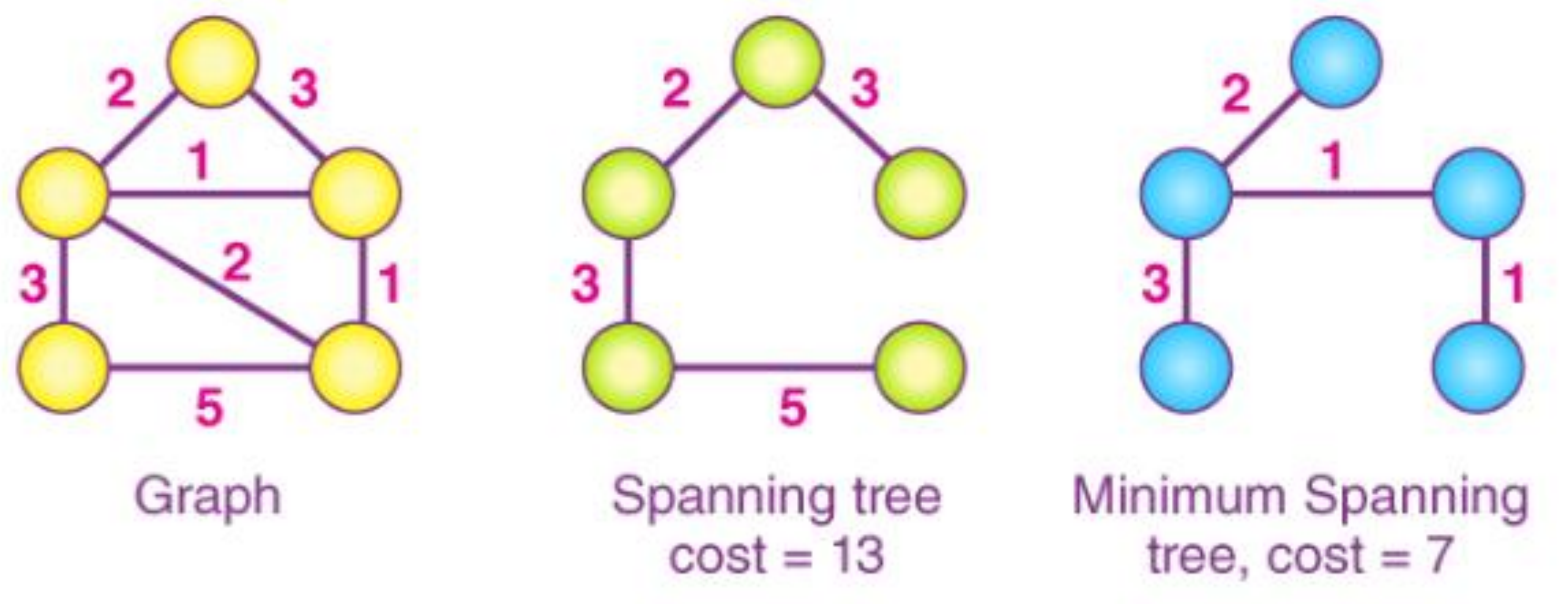
Spanning Trees

1. The spanning tree is a subgraph of an undirected connected graph.
2. It includes all the vertices in the graph and the least number of edges that can connect every vertex without forming a loop or cycle.
3. Alternate definition: A spanning tree T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G.
4. A graph may have several spanning trees
5. A graph that is not connected will not contain a spanning tree
6. If all of the edges of G are also edges of a spanning tree T of G, then G is a tree and is identical to T - a tree has a unique spanning tree and it is itself



A spanning tree (blue heavy edges) of a grid graph

Different Spanning Trees for same Graph - Example



1. For this given graph, two possible spanning trees are shown
2. This type of graph with values associated with the edges are called **edge-weighted graphs -** Weights represent cost of travelling that edge.
3. If we want to optimize (minimize) cost, we would choose the second spanning tree – The minimum spanning tree

Minimum Cost Spanning Trees (MST)

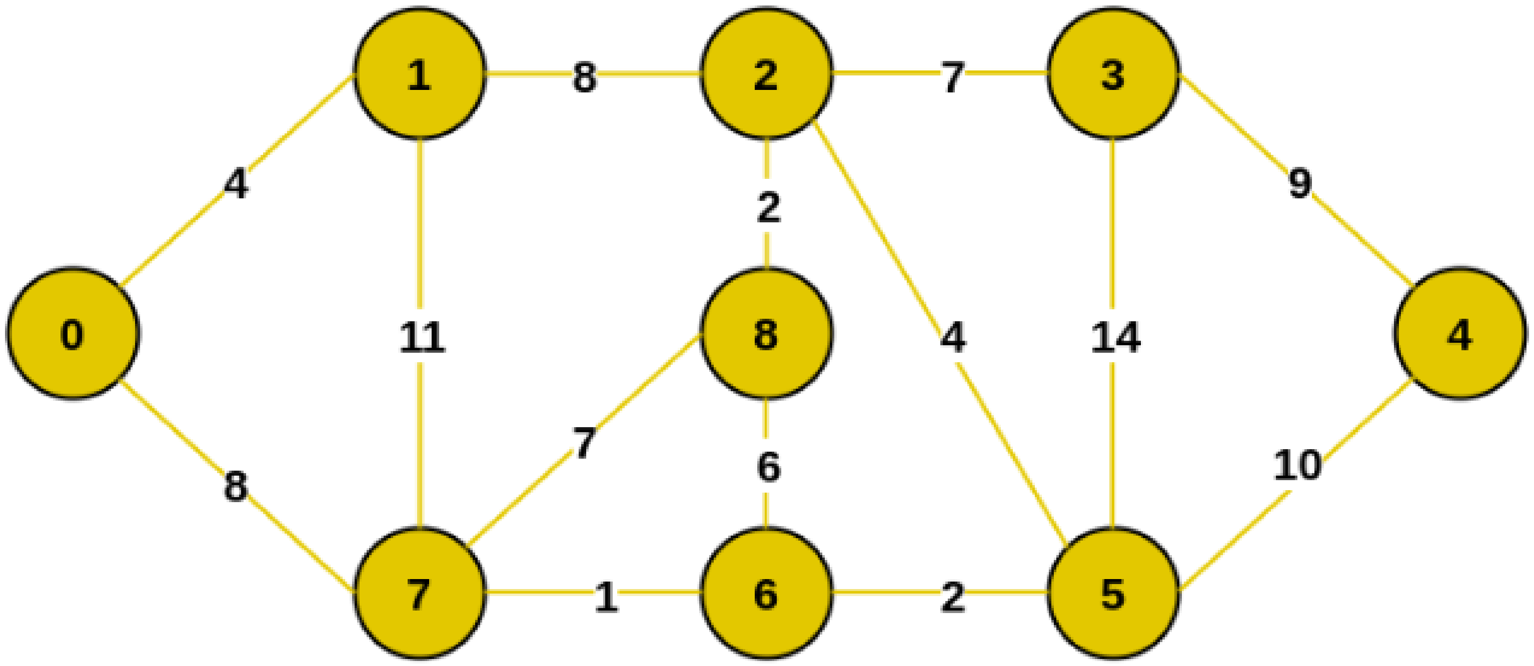
1. A minimum spanning tree (MST) is
   a) a subset of the edges of a connected, edge-weighted graph
   b) that connects all the vertices together without any cycles
   c) and with the minimum possible total edge weight.
2. It is a way of finding the most economical way to connect a set of vertices.
3. Applications of Minimum Spanning Tree
   - It helps in finding the route or paths on the map
   - For water-supply networks and also help in creating the networks like telecommunication etc
4. We can evaluate the minimum spanning tree from a graph using two algorithms:
   - Prim's Algorithm
   - Kruskal's Algorithm

Prim’s Algorithm

1. Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
2. Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
3. Step 3: Find edges connecting any tree vertex with the fringe vertices.
4. Step 4: Find the minimum among these edges.
5. Step 5: Add the chosen edge to the MST if it does not form any cycle.
6. Step 6: Return the MST and exit

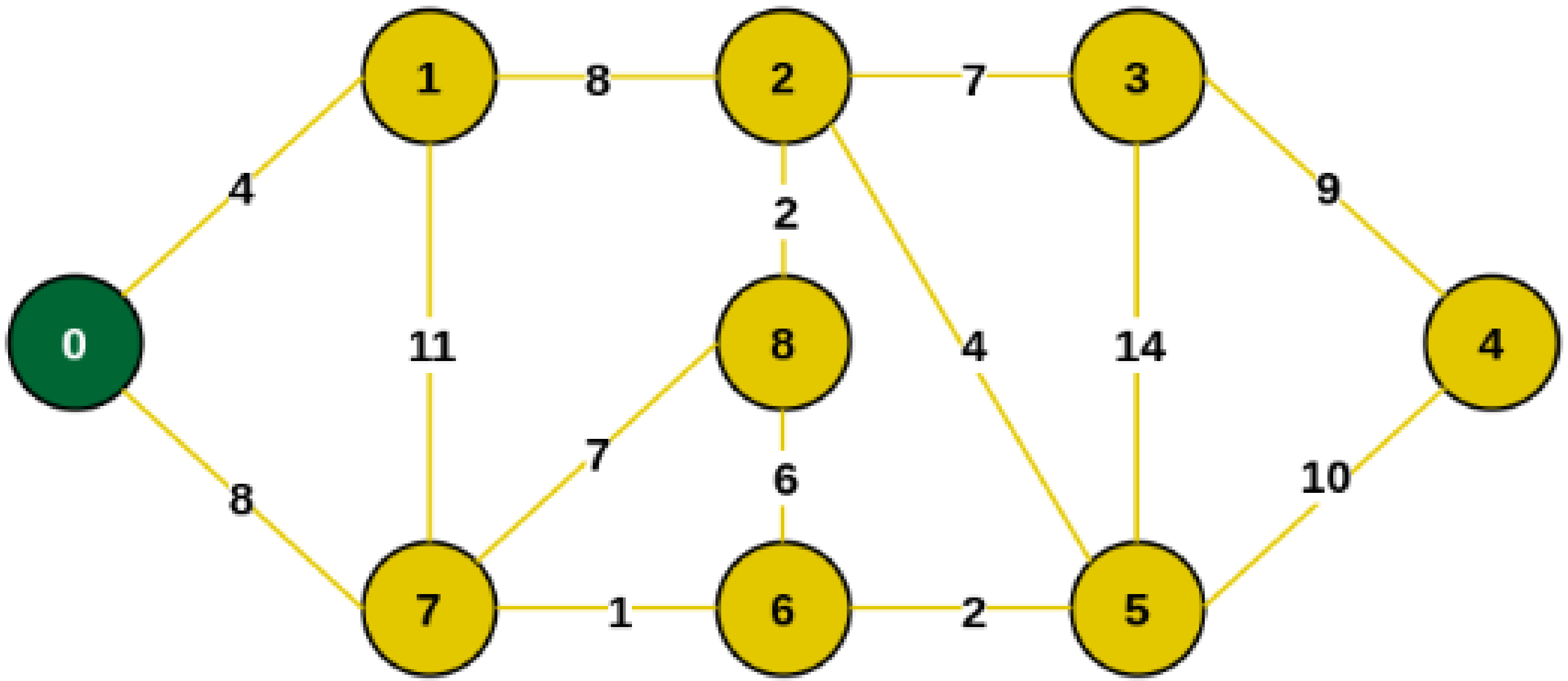
Prim's Algorithm

1. Start with the given graph



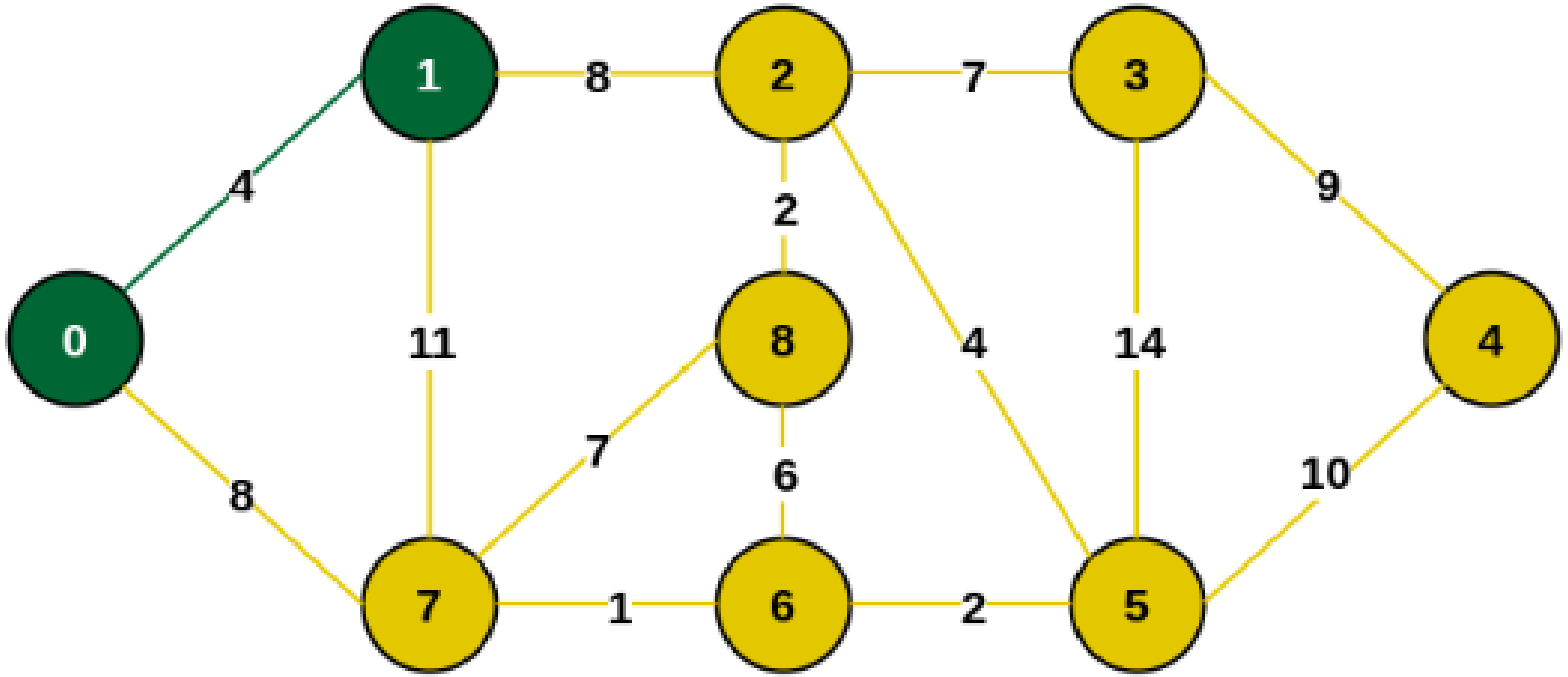
Prim’s Algorithm

1. Step 1: Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex 0 as the starting vertex.



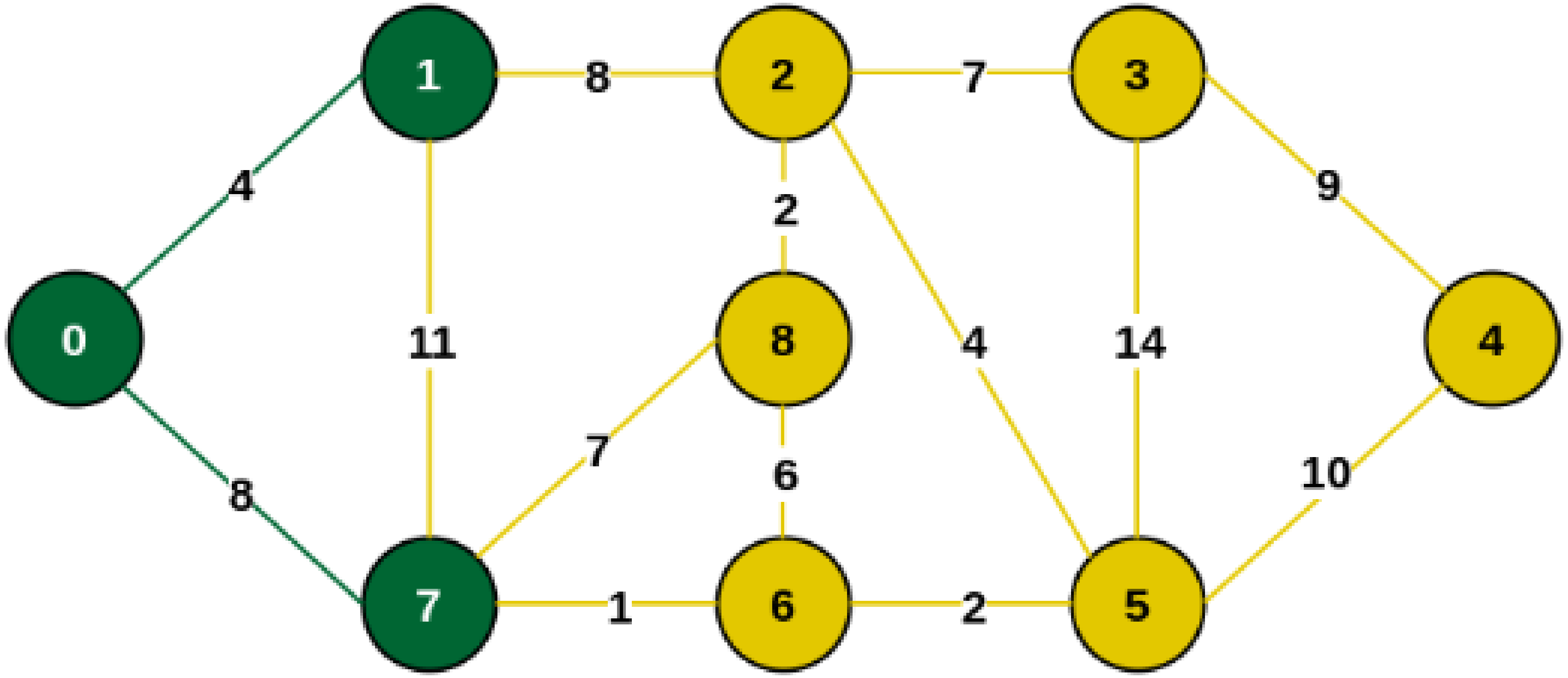
Prim's Algorithm

1. Step 2: All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.



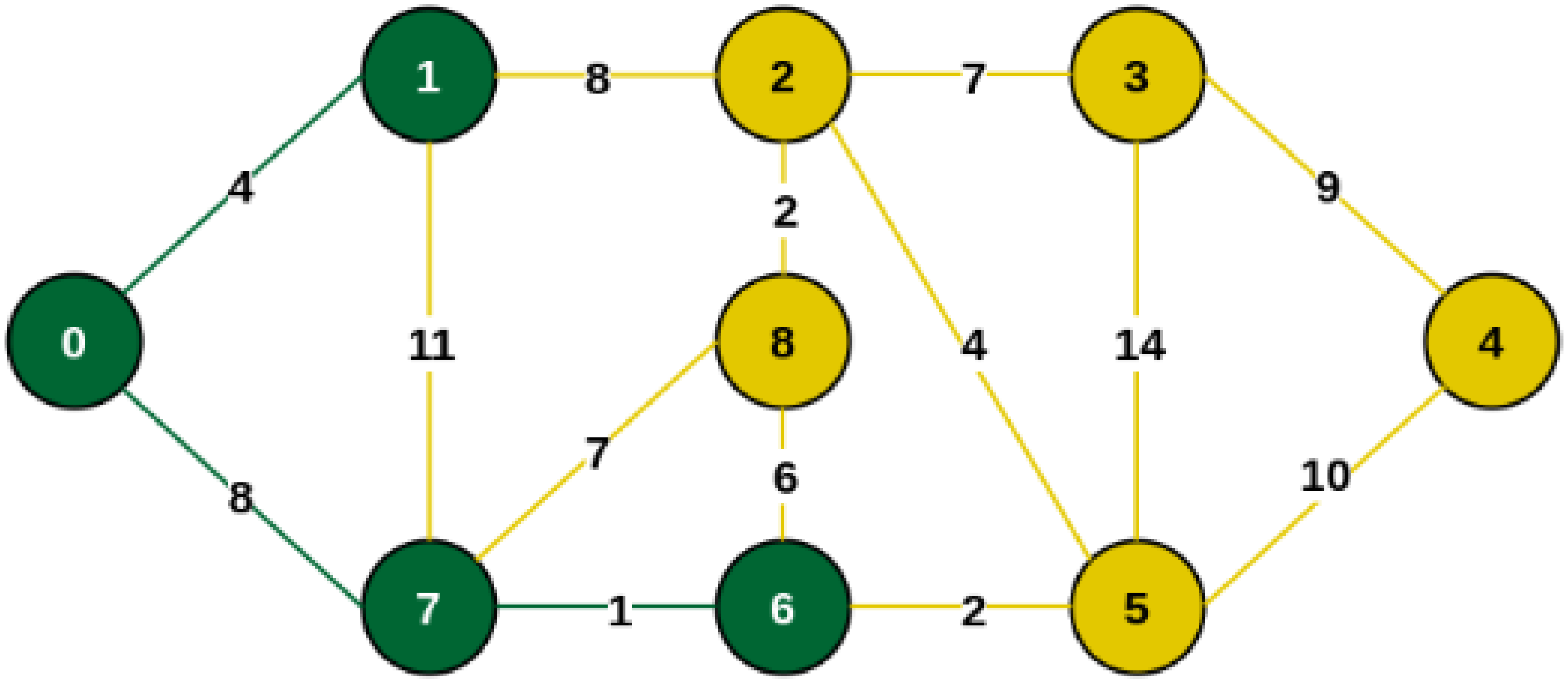
Prim’s Algorithm

1. Step 3: The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].



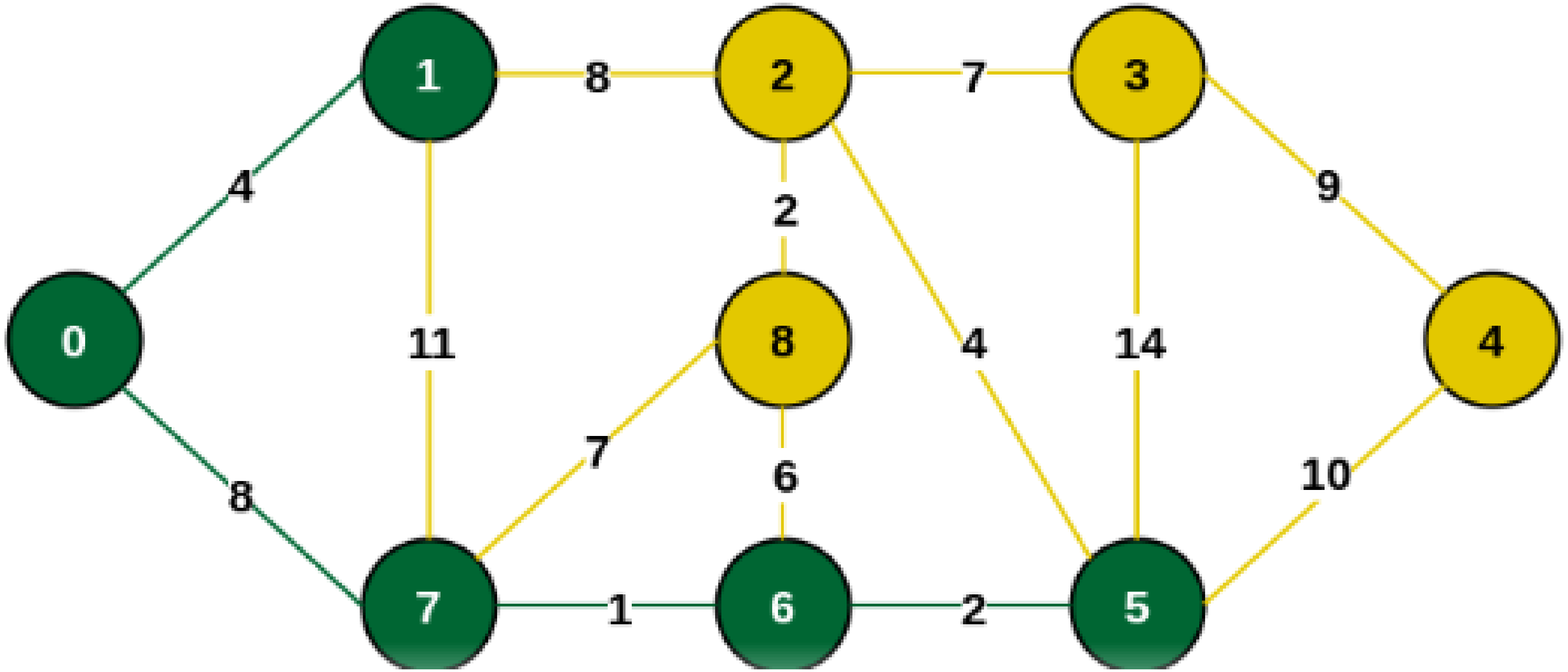
Prim’s Algorithm

1. Step 4: The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).
2. Why we haven’t considered edge {1, 7}?



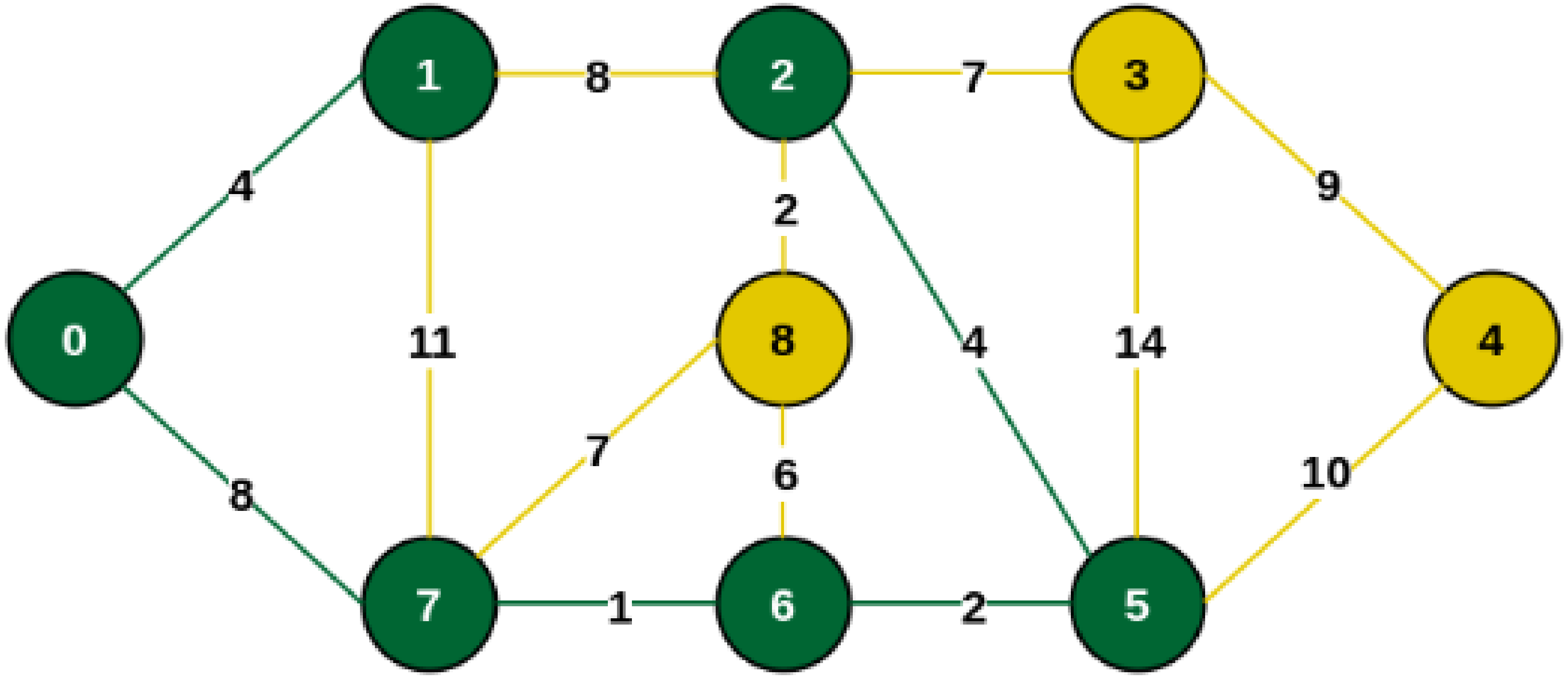
Prim’s Algorithm

1. Step 5: The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.



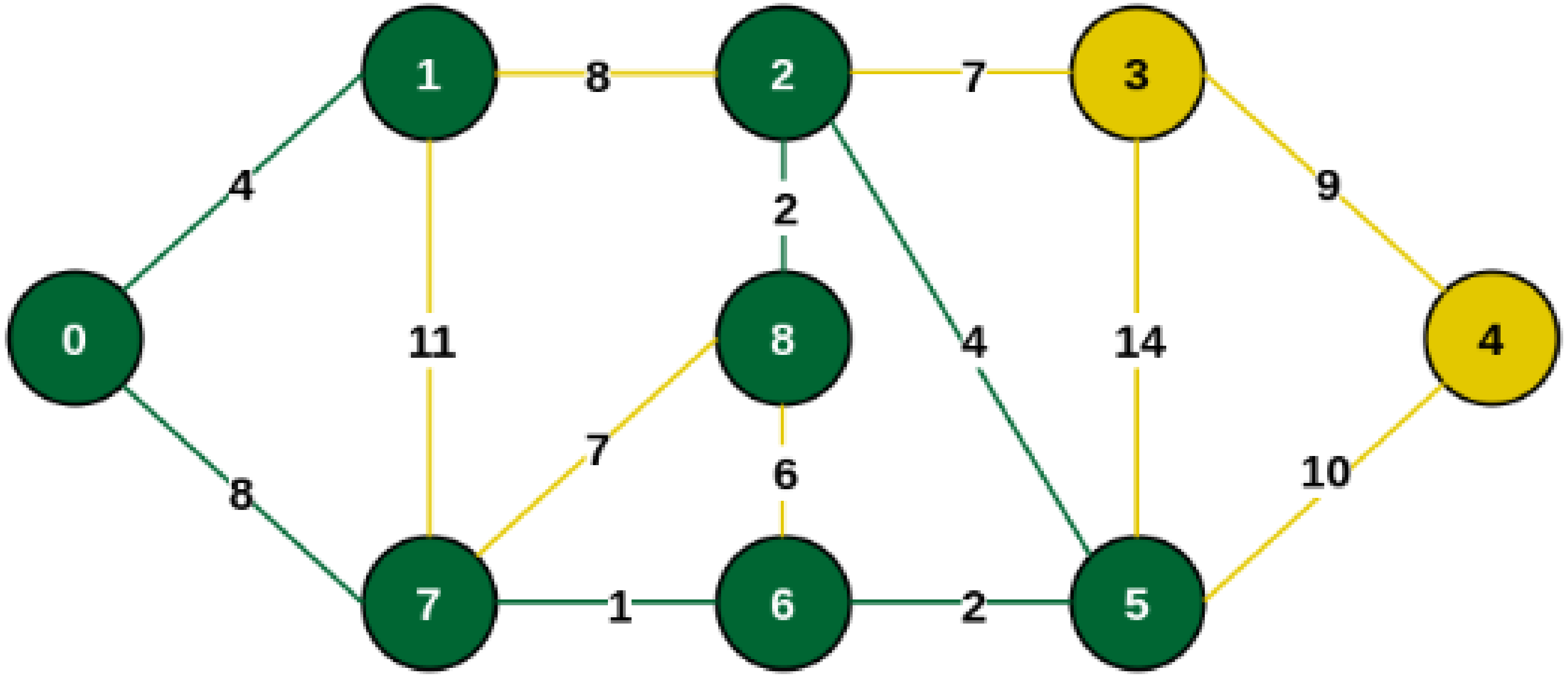
Prim’s Algorithm

1. Step 6: Among the current connecting edges, the edge {5, 2} has the minimum weight. So include that edge and the vertex 2 in the MST.



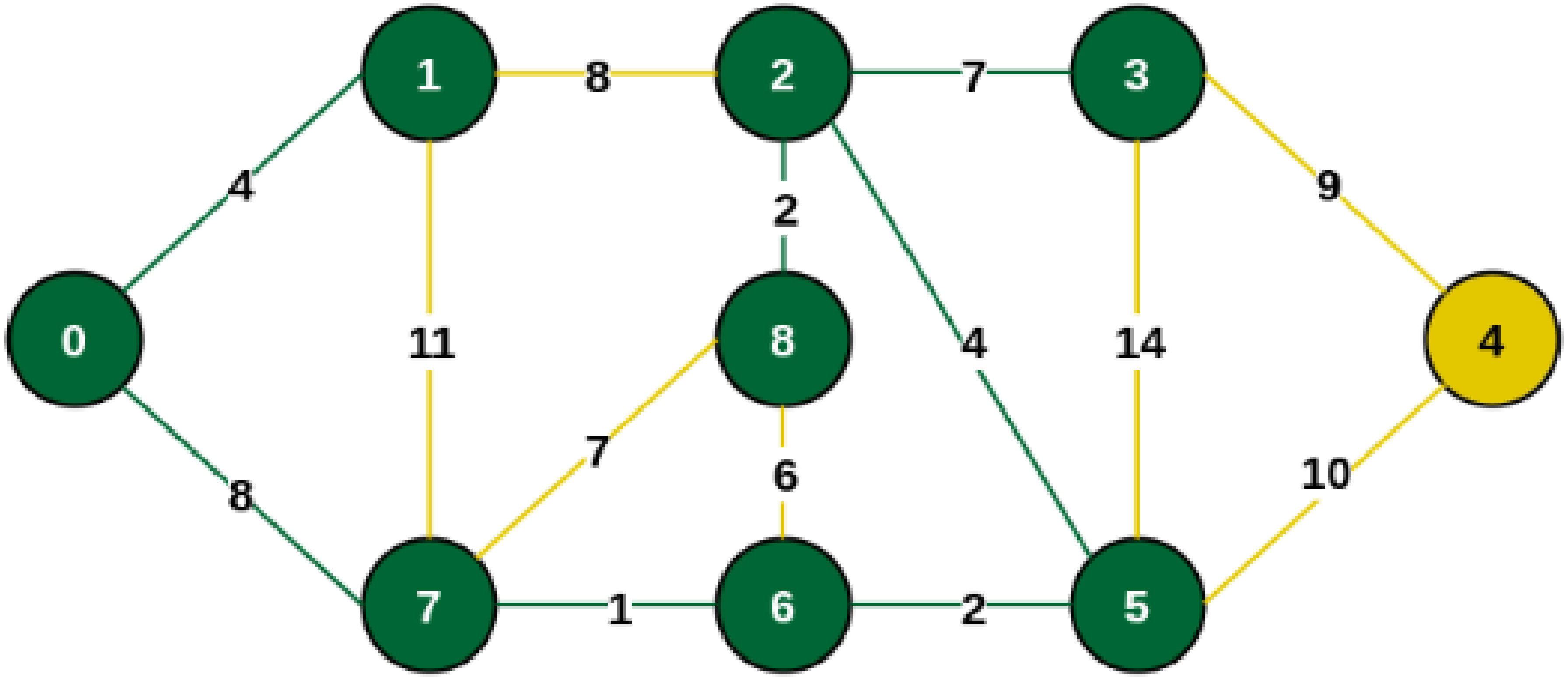
Prim’s Algorithm

1. Step 7: The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.



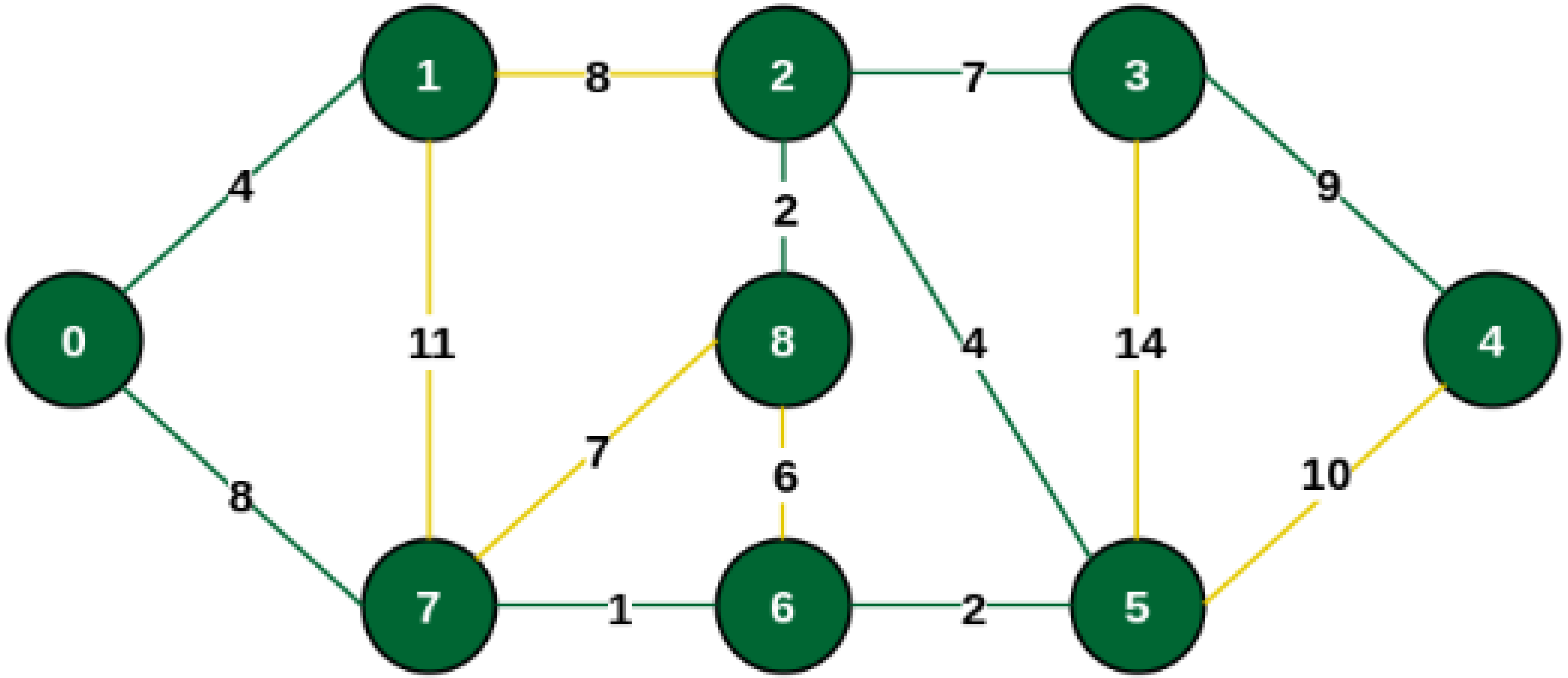
Prim’s Algorithm

1. Step 8: See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.



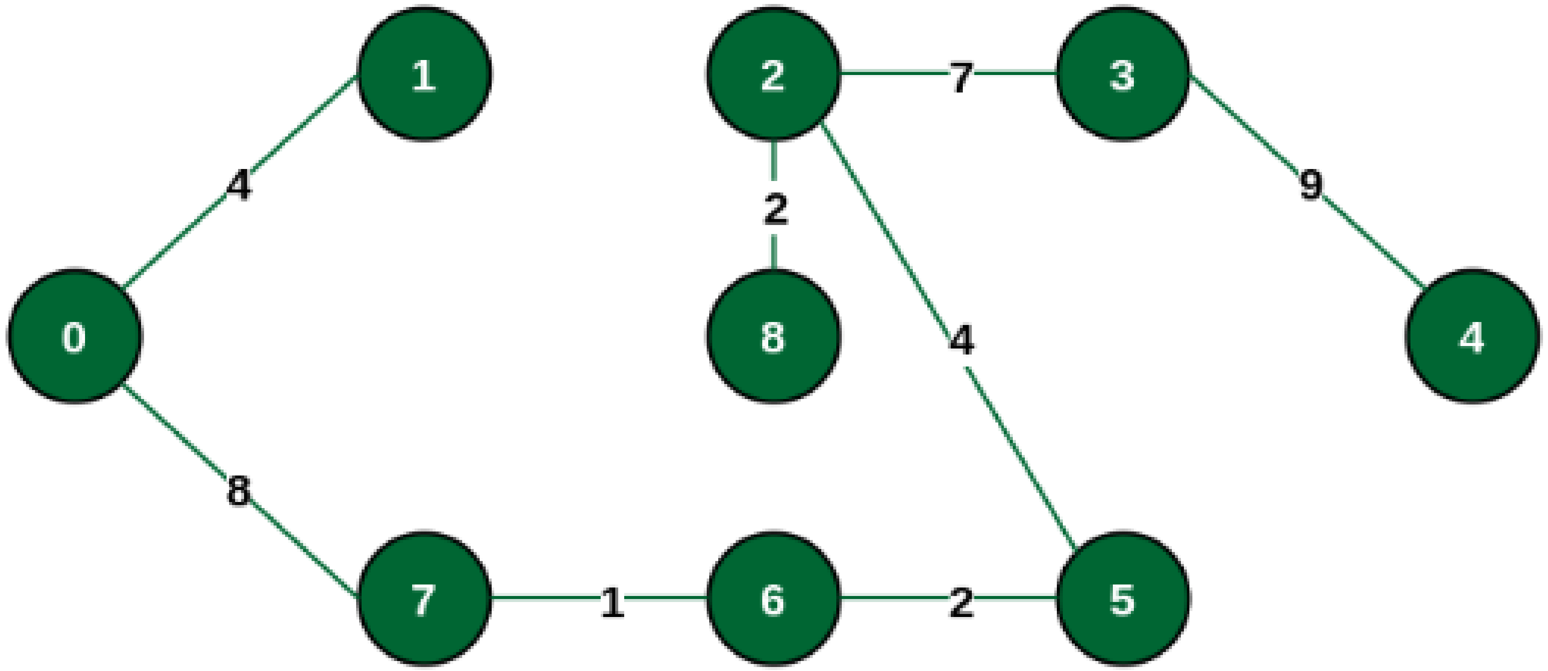
Prim’s Algorithm

1. Step 9: Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4 is {3, 4}.



Prim’s Algorithm

1. The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = 37.



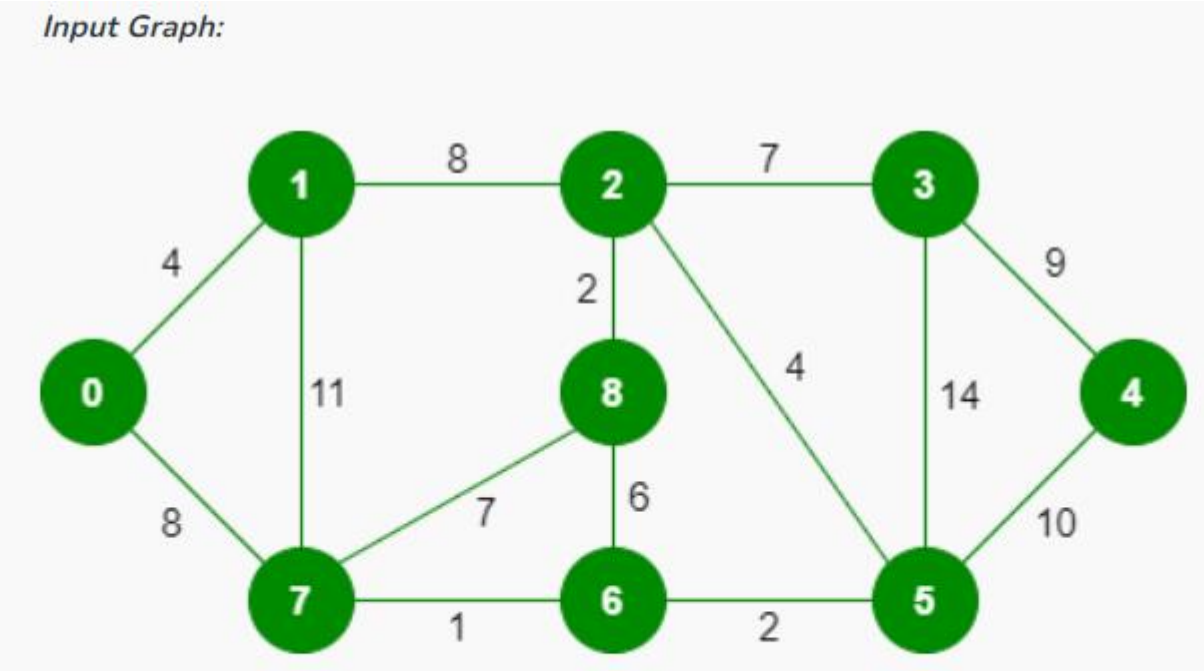
The final structure of MST

Kruskal's Algorithm for MST

1. Sort all the edges of the input graph in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.
4. Note: Step 2 uses the Union-Find algorithm to detect cycles.

Kruskal’s Algorithm

1. Start with the given graph

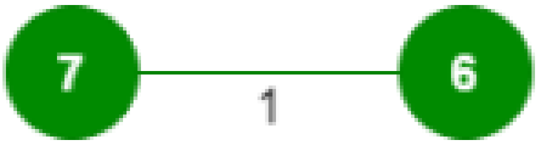


Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

2. The graph contains 9 vertices and 14 edges.
3. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

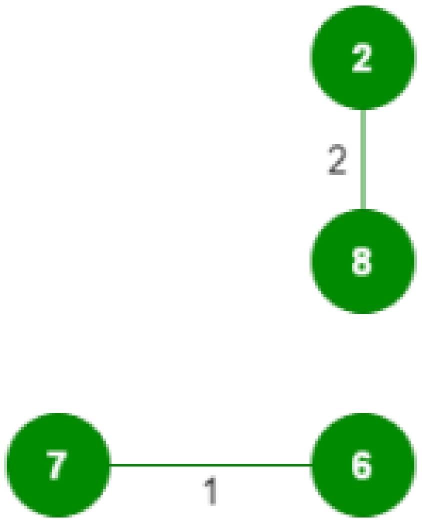
Kruskal’s Algorithm

1. Pick edge 7-6. No cycle is formed, include it.



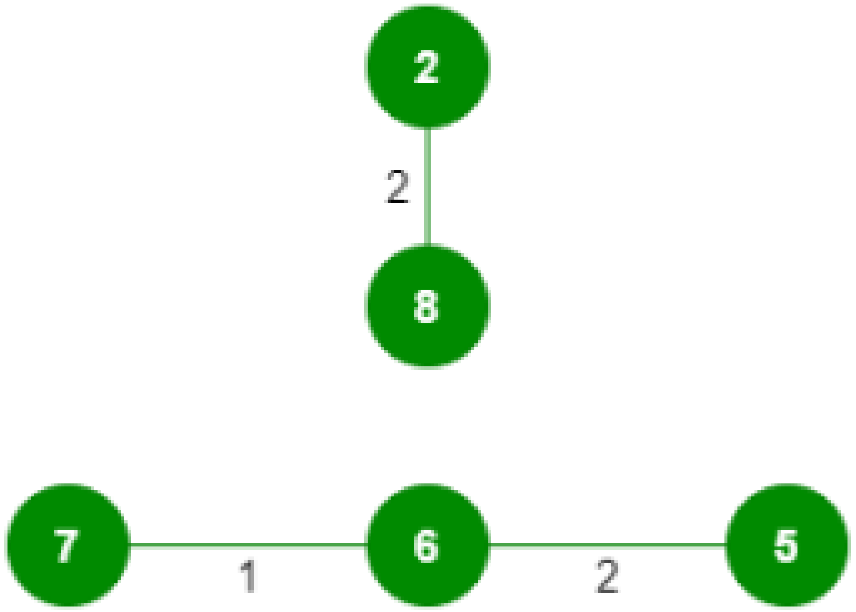
Kruskal’s Algorithm

1. Pick edge 8-2. No cycle is formed, include it.



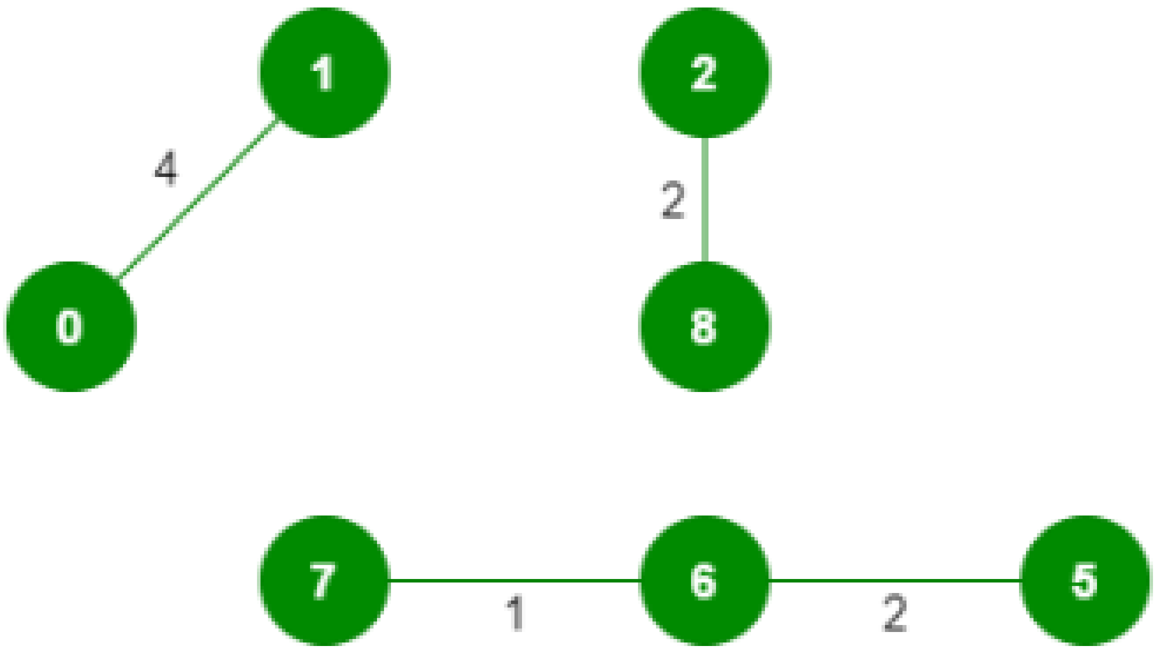
Kruskal’s Algorithm

1. Pick edge 6-5. No cycle is formed, include it.



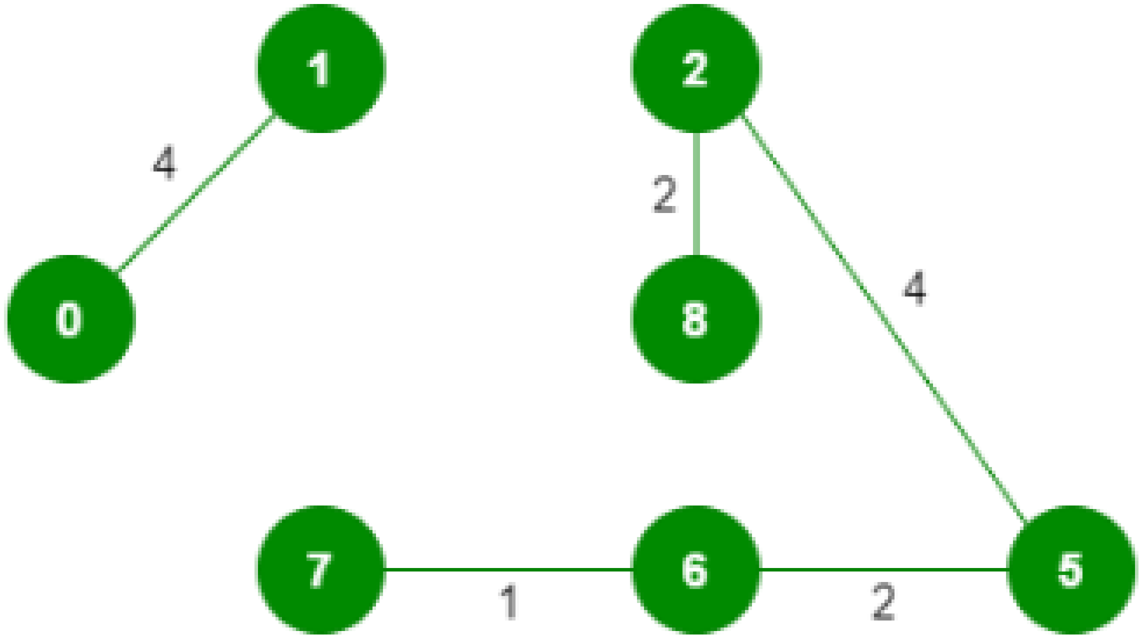
Kruskal’s Algorithm

1. Pick edge 0-1. No cycle is formed, include it.



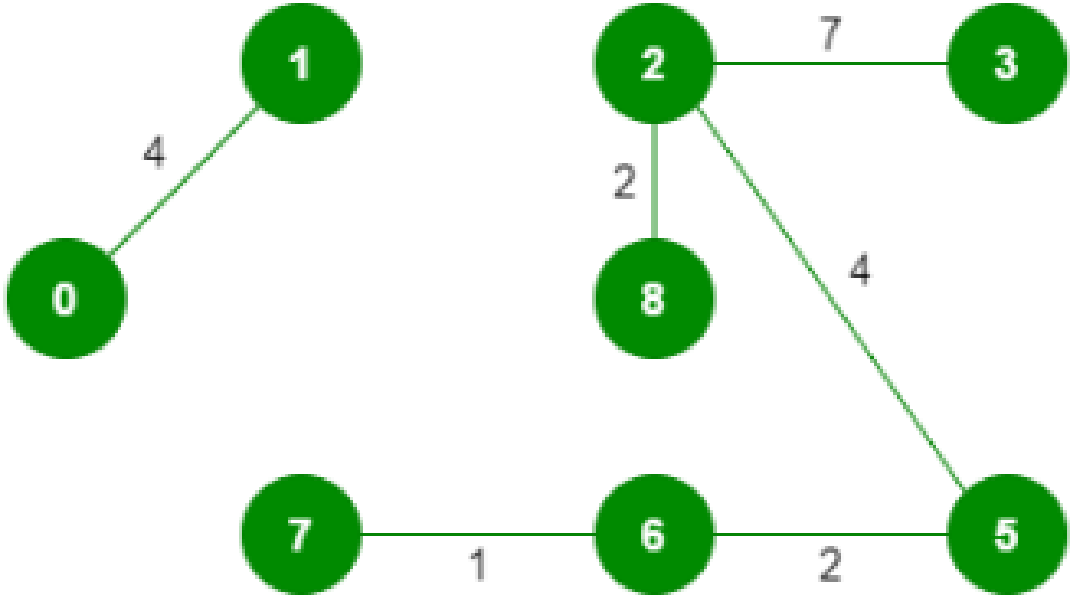
Kruskal’s Algorithm

1. Pick edge 2-5. No cycle is formed, include it.



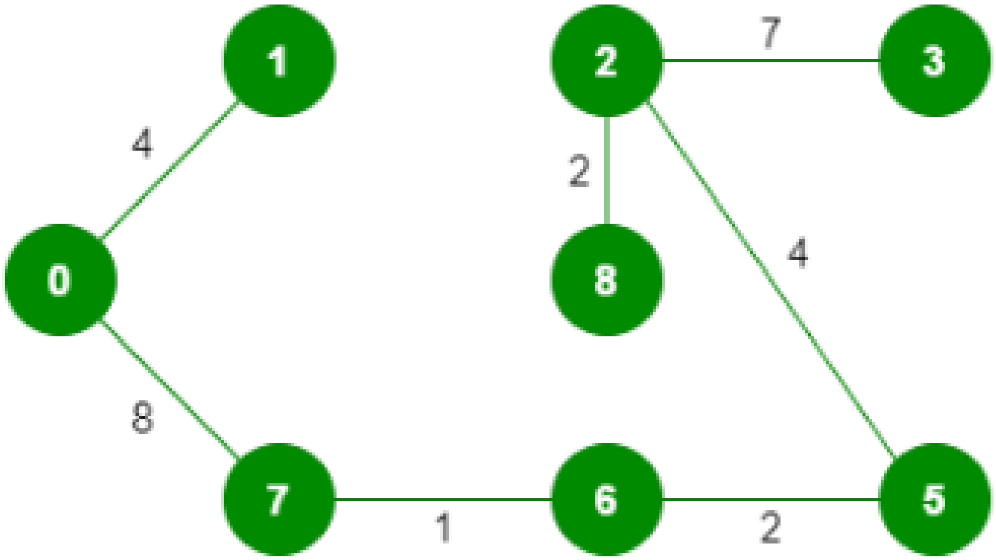
Kruskal's Algorithm

1. Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.



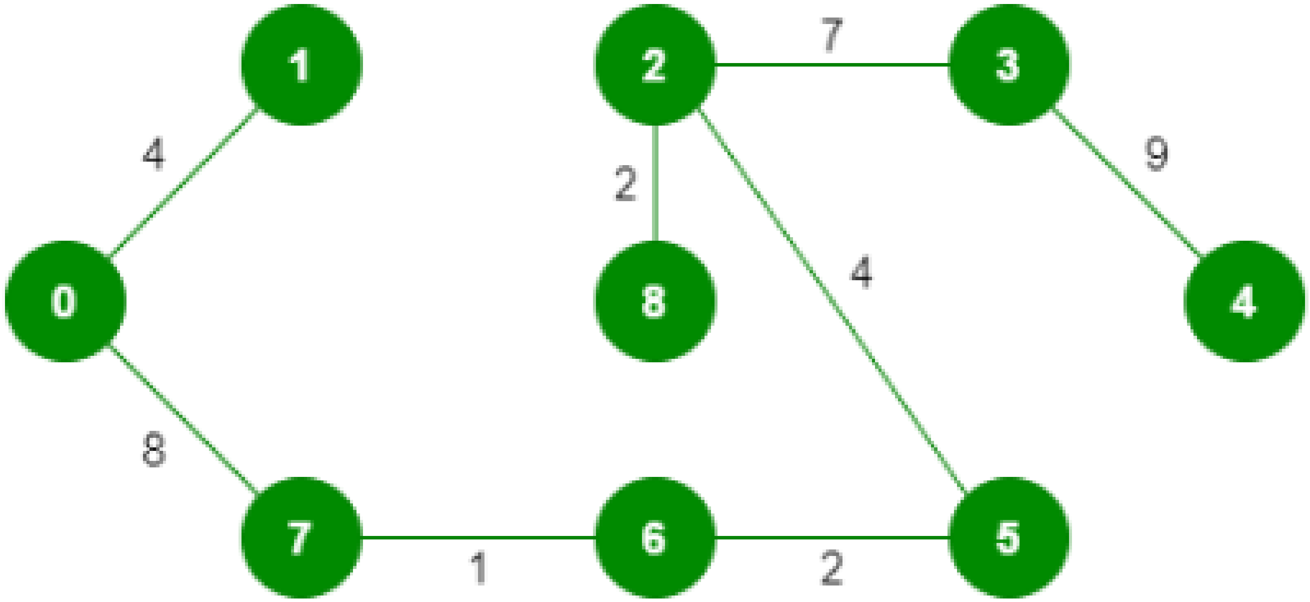
Kruskal’s Algorithm

1. Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.



Kruskal’s Algorithm

1. Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.



2. Note: Since the number of edges included in the MST equals to (V – 1), so the algorithm stops here