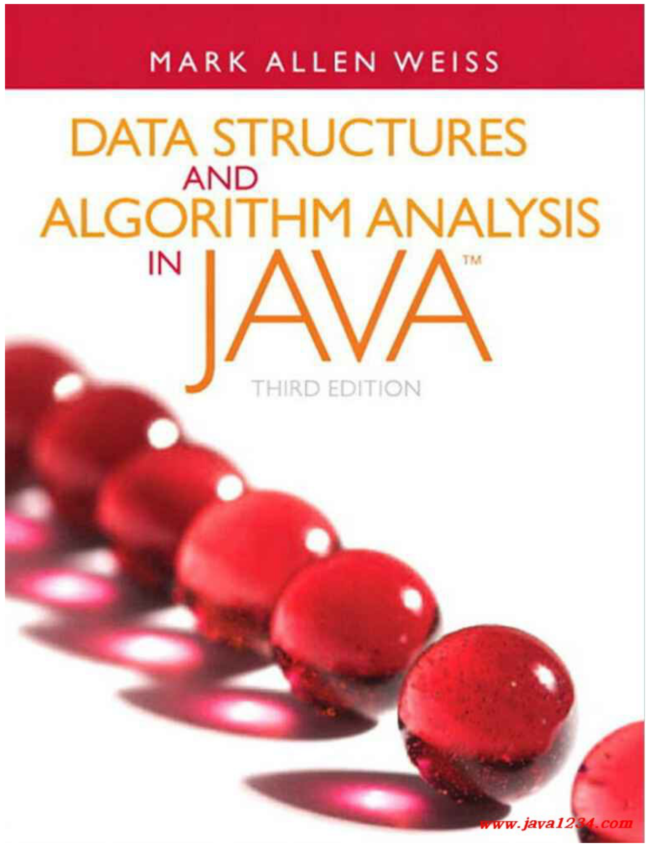
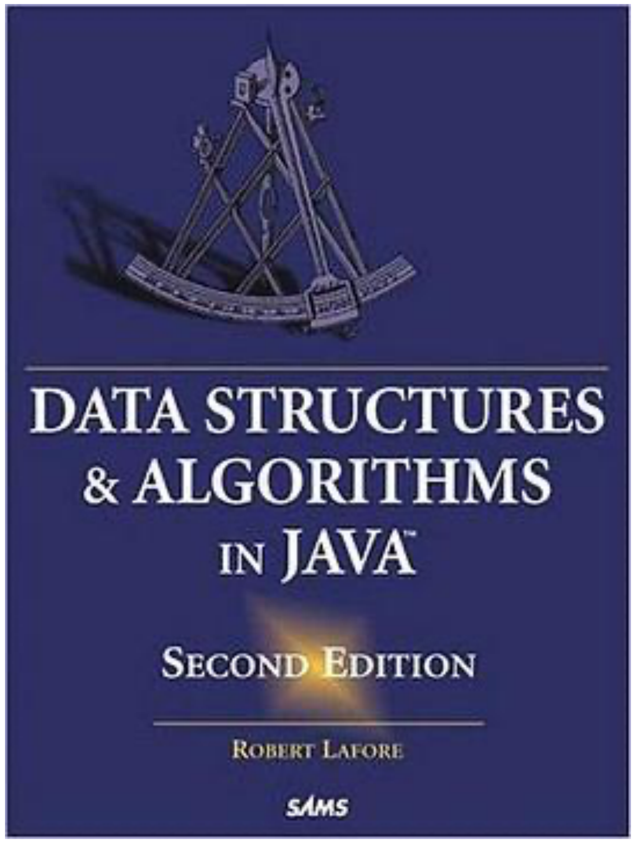
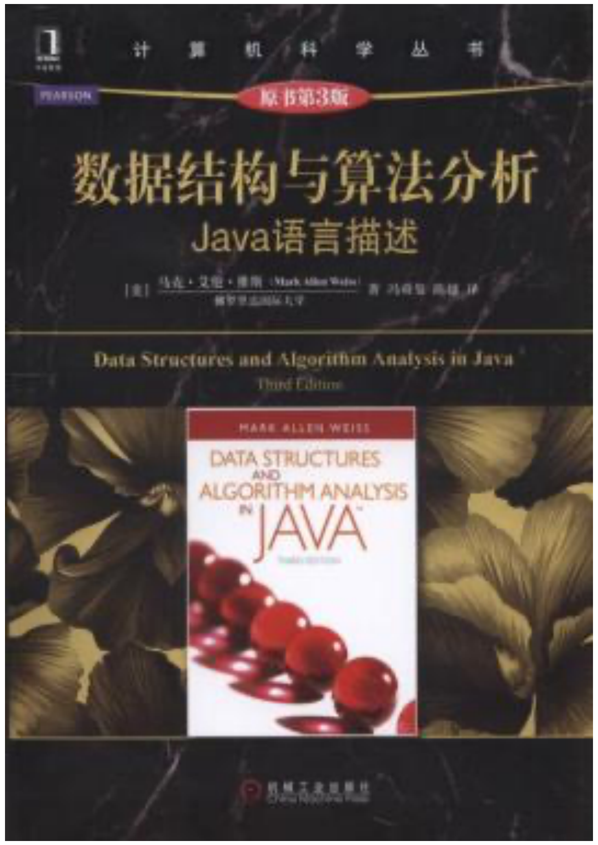
Topic 18 –
Minimum Spanning Tree







- **Minimum Spanning Tree**
- Kruskal's algorithm
- Prim's algorithm
    - matrix representation
    - adjacency list representation

Minimum Spanning Tree (MST)

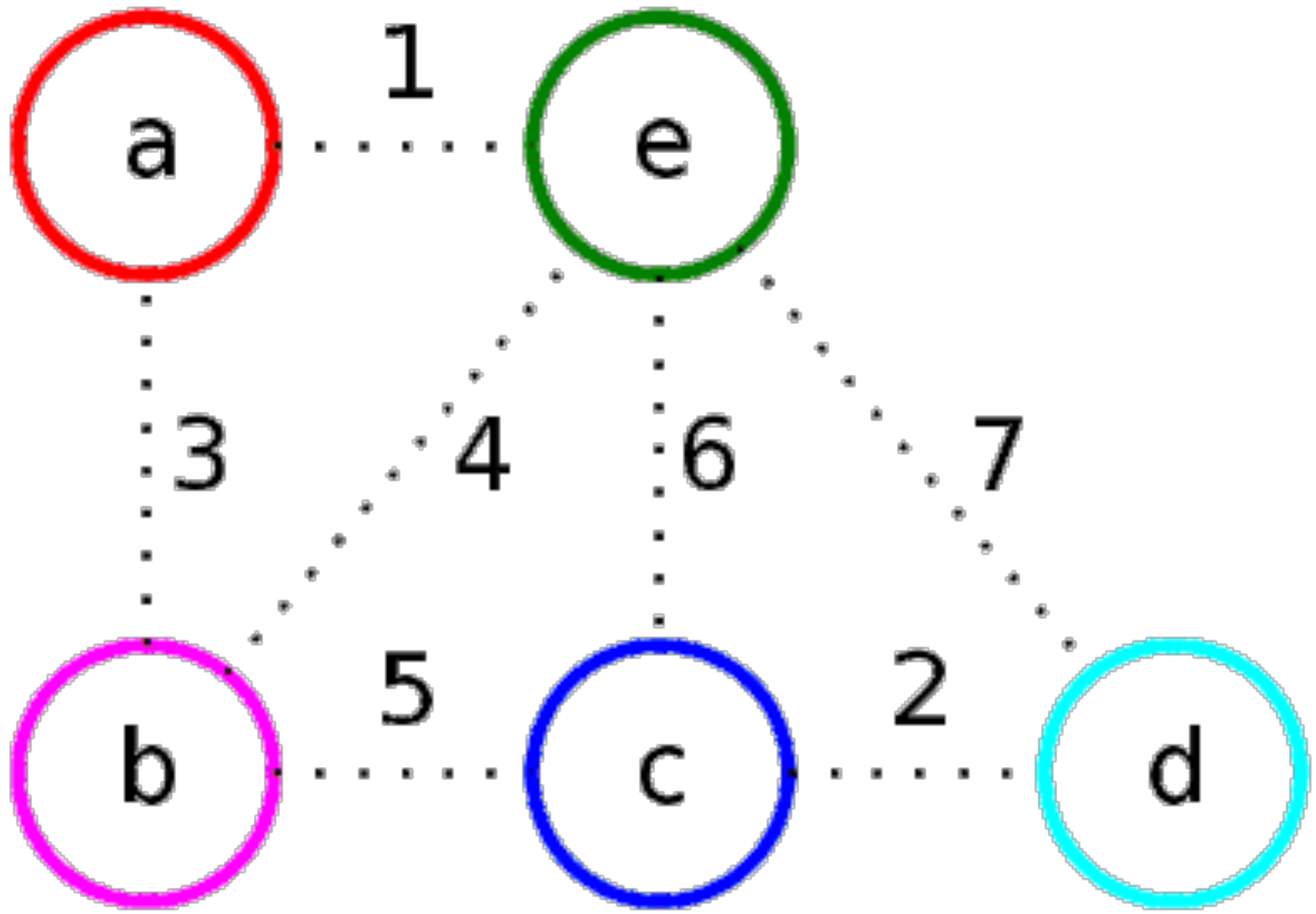
- Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees.
- A ***minimum spanning tree (MST)*** or **minimum weight spanning tree** for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

- *How many edges does a minimum spanning tree has?*
    - A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

- Minimum Spanning Tree
- **Kruskal's algorithm**
- Prim's algorithm
    - matrix representation
    - adjacency list representation

Kruskal's algorithm

Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6



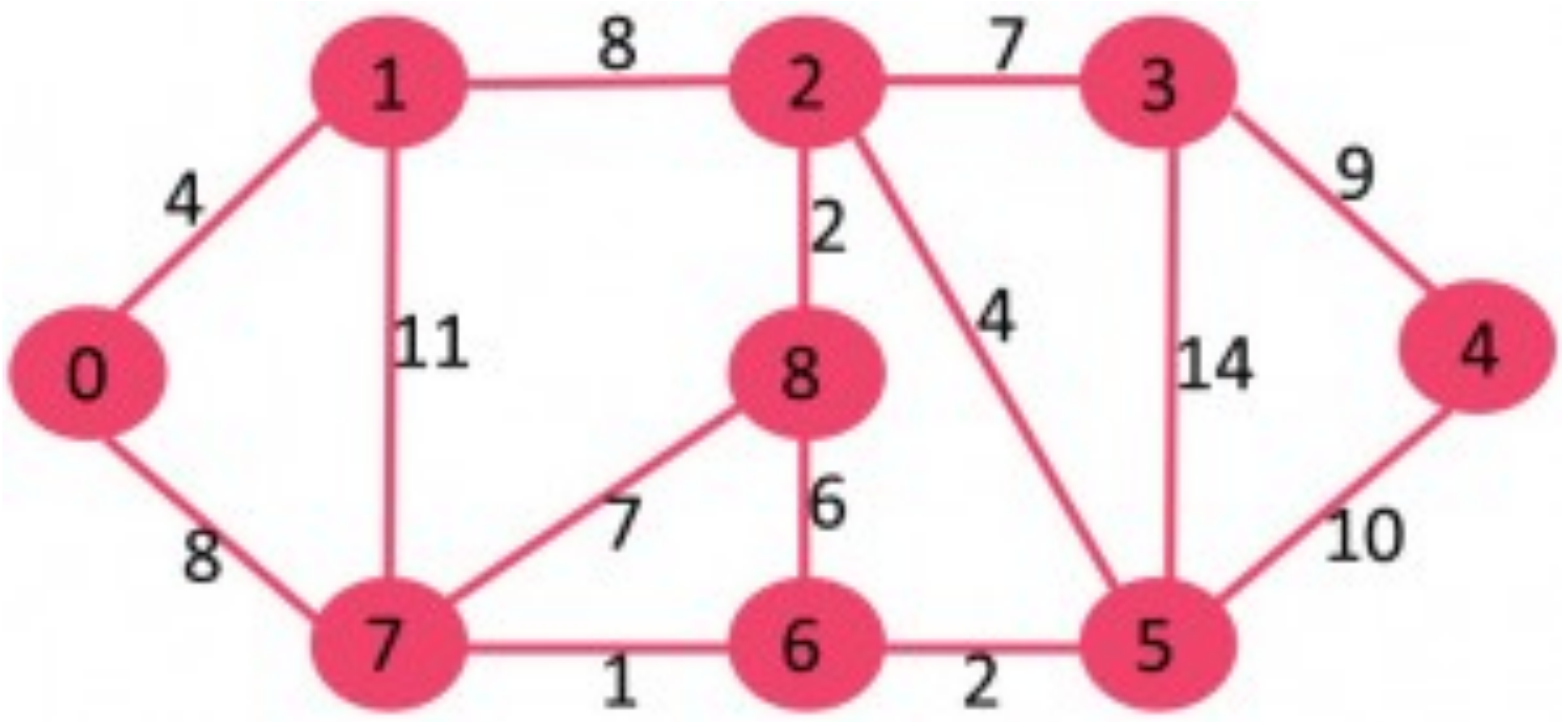
Kruskal's algorithm

- ***1.*** *Sort all the edges 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 cycle is not formed, include this edge. Else, discard it.*
- ***3.*** *Repeat step#2 until there are (V-1) edges in the spanning tree.*

- The algorithm is a ***Greedy Algorithm***.
  - The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far.

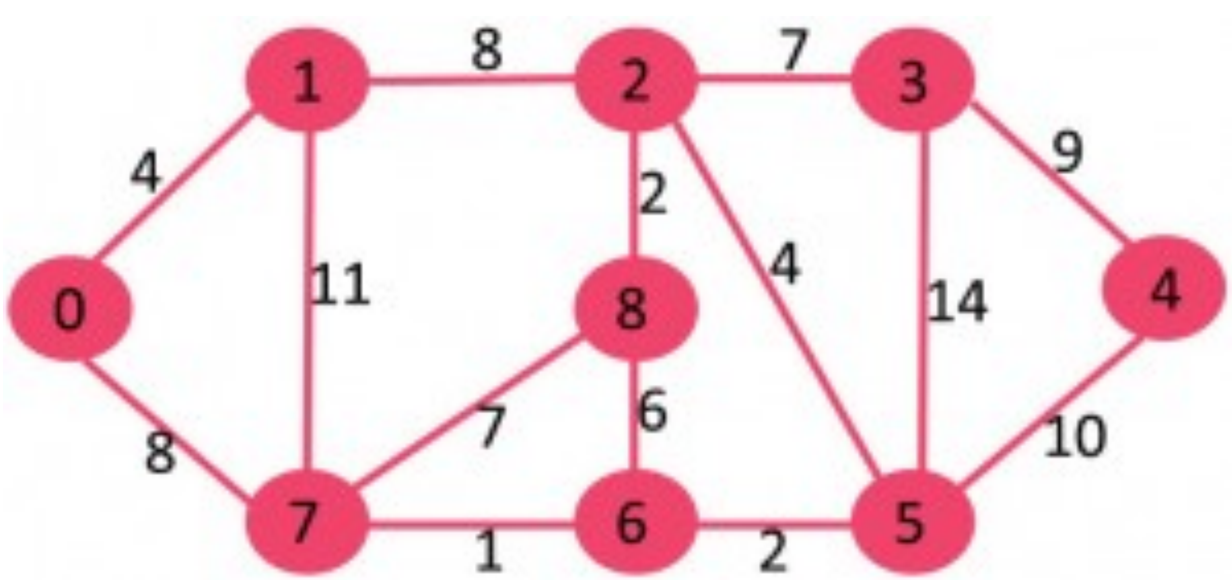
Example

- Consider the below input graph.



Example

- The graph contains 9 vertices and 14 edges.
- So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.
- **Step 1.**
  - After sorting:

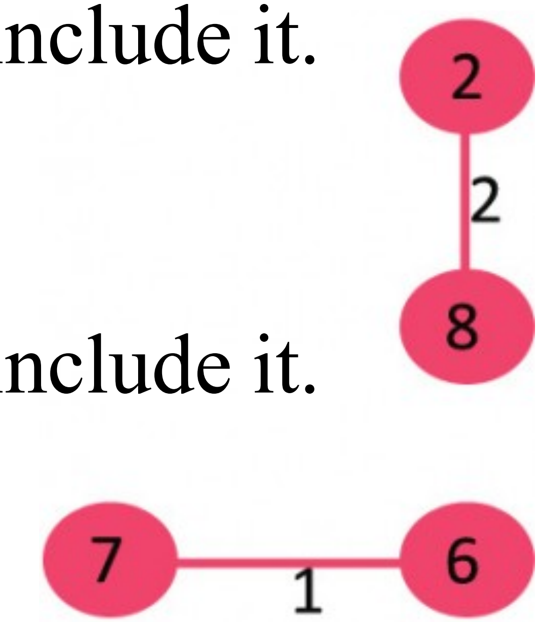


Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

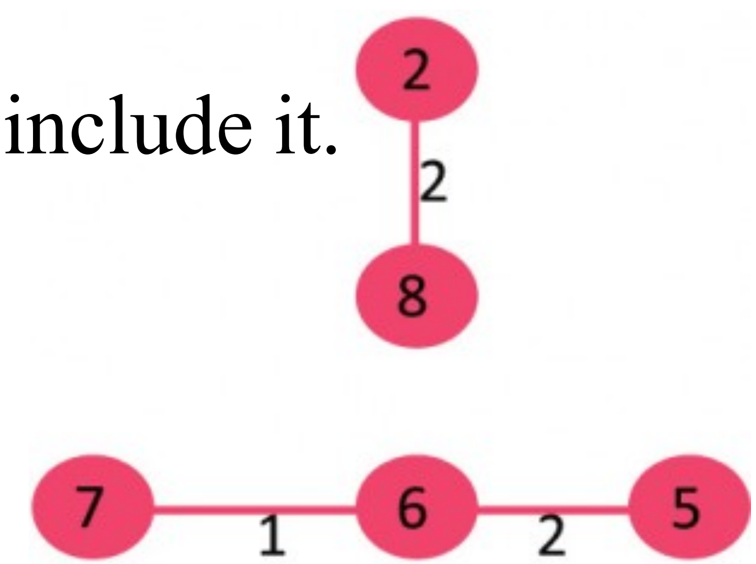
Example

- **Step 2.**
  - Now pick all edges one by one from sorted list of edges
  - **1.** *Pick edge 7-6:* No cycle is formed, include it.

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

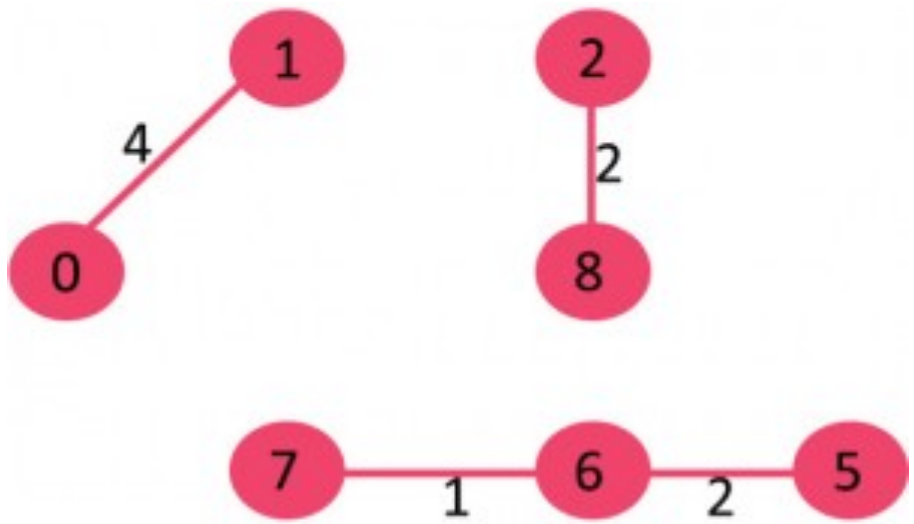


  - **3.** *Pick edge 6-5:* No cycle is formed, include it.

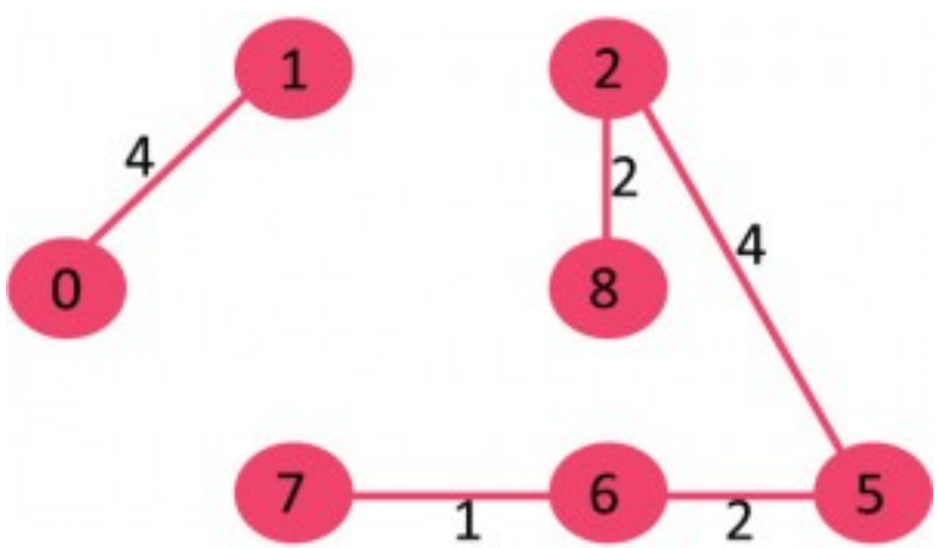


Example

- **Step 2.**
  - **4.** *Pick edge 0-1:* No cycle is formed, include it.

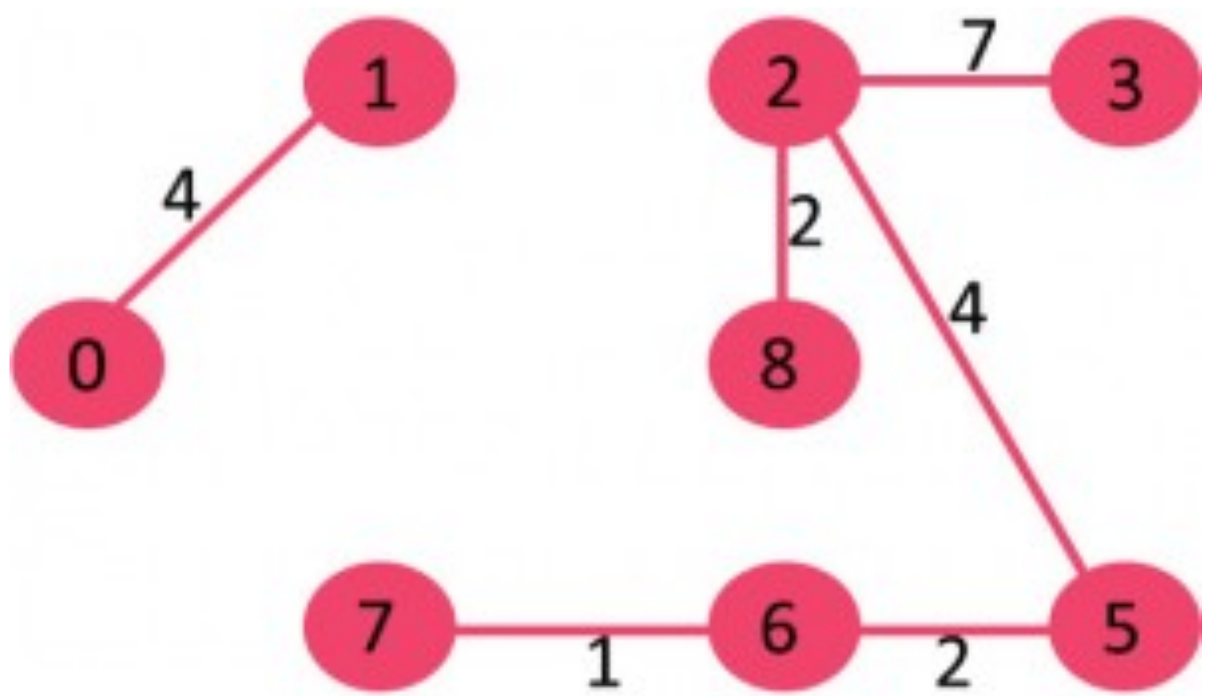


  - **5.** *Pick edge 2-5:* No cycle is formed, include it.



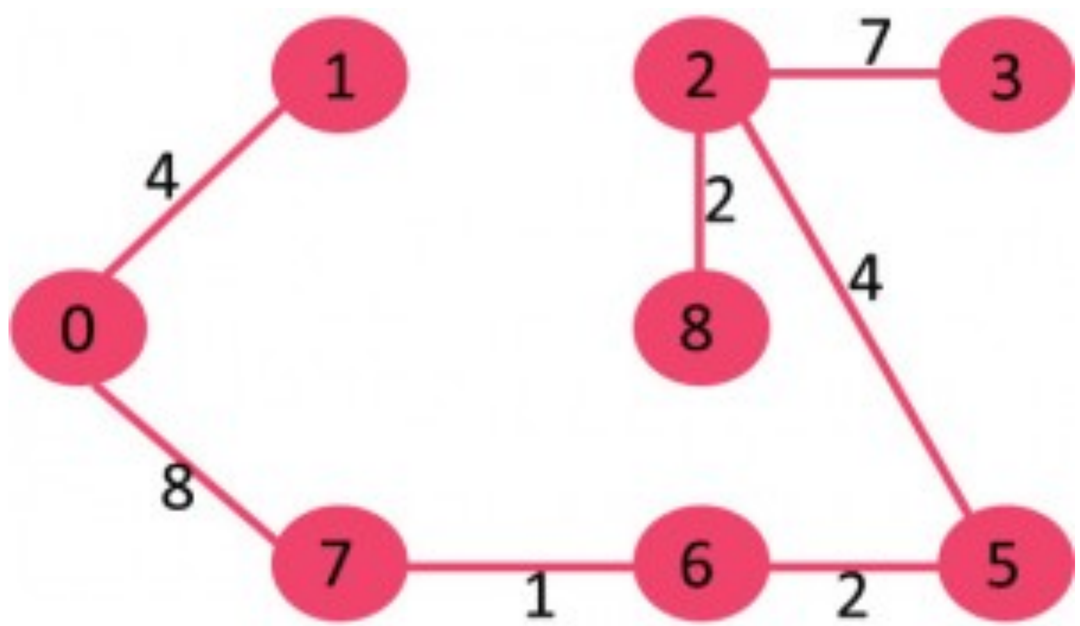
Example

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



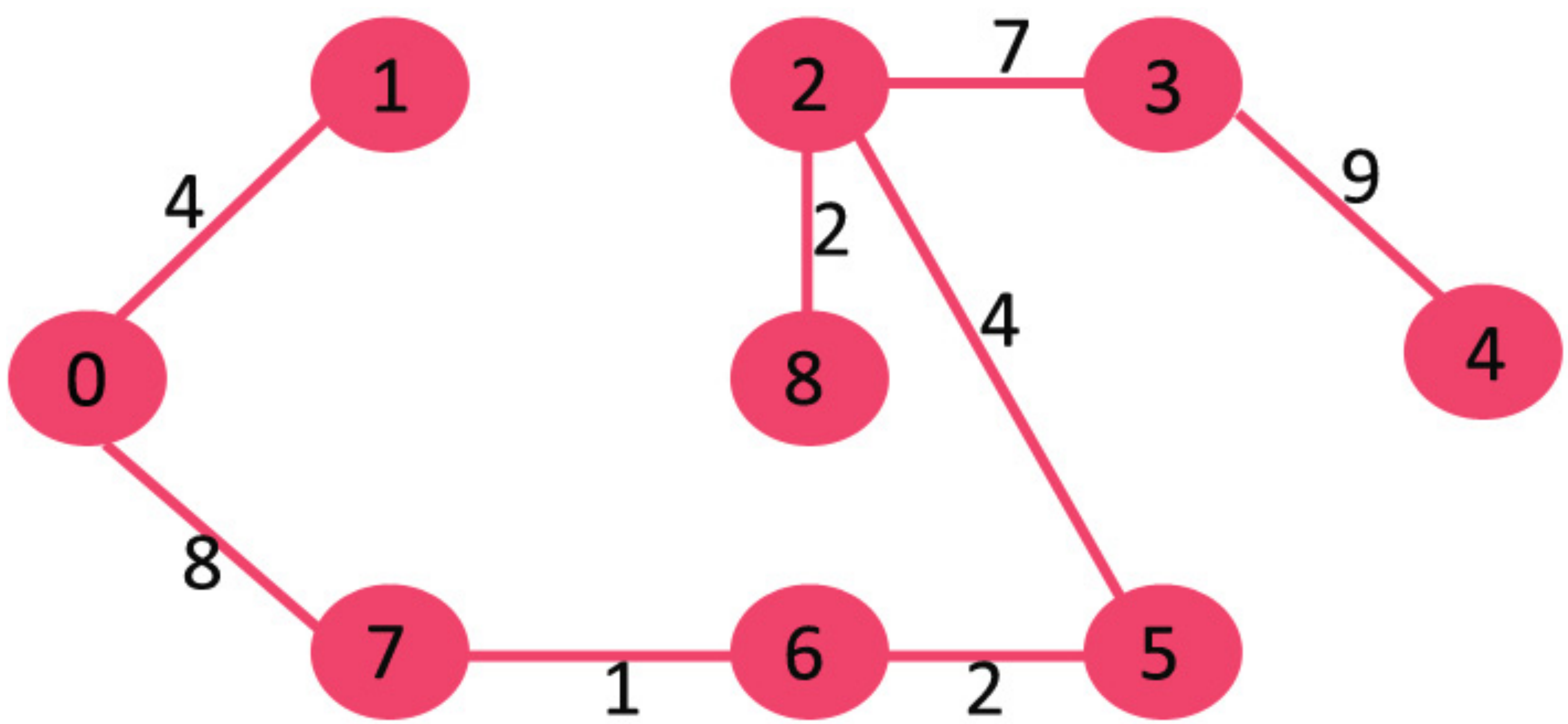
Example

- **Step 2.**
  - **8.** *Pick edge 7-8:* Since including this edge results in cycle, discard it.
  - **9.** *Pick edge 0-7:* No cycle is formed, include it.



Example

- **Step 2.**
  - **10.** *Pick edge 1-2:* Since including this edge results in cycle, discard it.
  - **11.** *Pick edge 3-4:* No cycle is formed, include it.



- **Step 3.**
  - Since the number of edges included equals (V – 1), the algorithm stops here.

Time Complexity

- $O(|E| \log |E|)$ or $O(|E| \log |V|)$.
  - Sorting of edges takes $O(|E| \log |E|)$ time.
  - After sorting, we iterate through all edges and apply find-union algorithm.
  - The find and union operations can take at most $O(|E| \log |V|)$ time.
  - So overall complexity is $O(|E| \log |E| + |E| \log |V|)$ time.
  - The value of $|E|$ can be at most $O(|V|^2)$, so $O(\log |V|)$ are $O(\log |E|)$ same.
  - Therefore, overall time complexity is $O(|E| \log |E|)$ or $O(|E| \log |V|)$

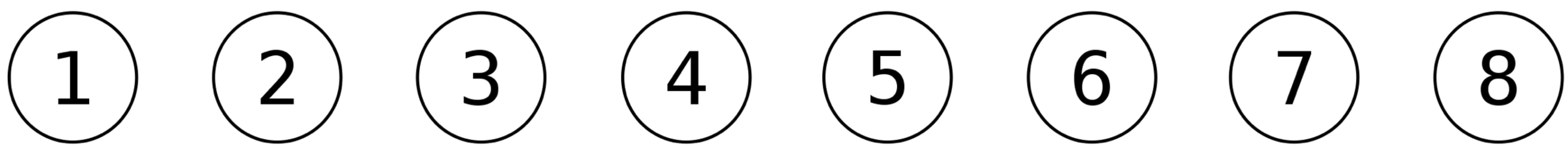
Kruskal's algorithm

- ***1.*** *Sort all the edges 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 cycle is not formed, include this edge. Else, discard it.*
- ***3.*** *Repeat step#2 until there are (V-1) edges in the spanning tree.*

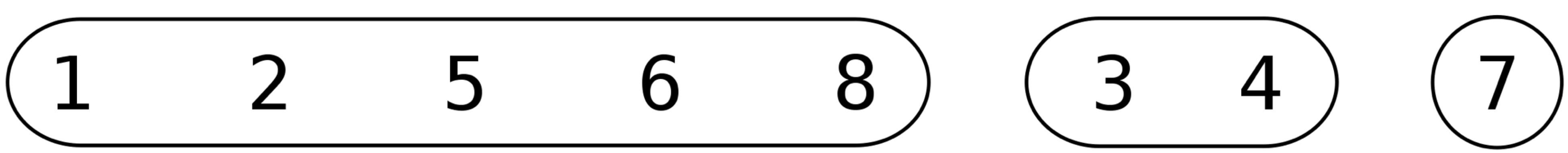
- *Note.* The step#2 uses Union-Find algorithm to detect cycle.

Union–find data structure

- A **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that stores a collection of disjoint (non-overlapping) sets.
- Equivalently, it stores a partition of a set into disjoint subsets.



MakeSet creates 8 singletons.



After some operations of *Union*, some sets are grouped together.

Union–find data structure

- ***Find:***
  - Determine which subset a particular element is in.
  - This can be used for determining if **two elements are in the same subset**.
- ***Union:***
  - Join two subsets into a single subset.
  - In this post, we will discuss the application of Disjoint Set Data Structure.
  - The application is to check whether a given graph contains a cycle or not.

Union–find data structure

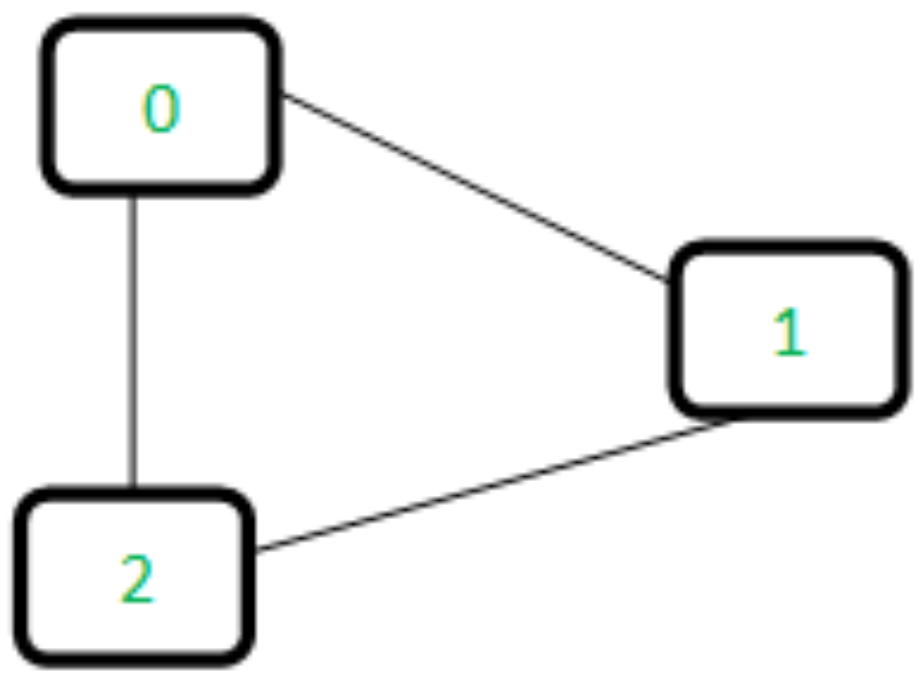
- *Union-Find Algorithm* can be used to check whether an undirected graph contains cycle or not.

- Note that we have discussed an algorithm to detect cycle.
- This is another method based on ***Union-Find***.
- This method assumes that the graph doesn't contain any self-loops.

Union–find data structure

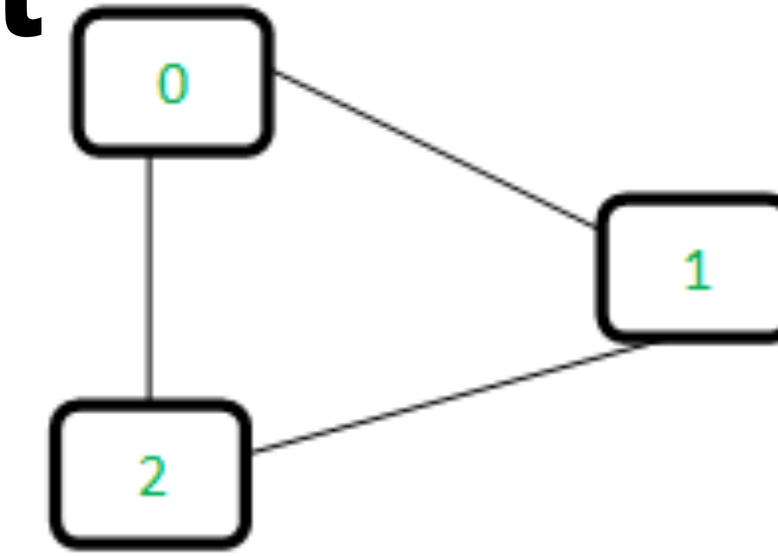
- We can keep track of the subsets in a 1D array, let’s call it parent[].

- Let us consider the following graph:



Union–find data structure

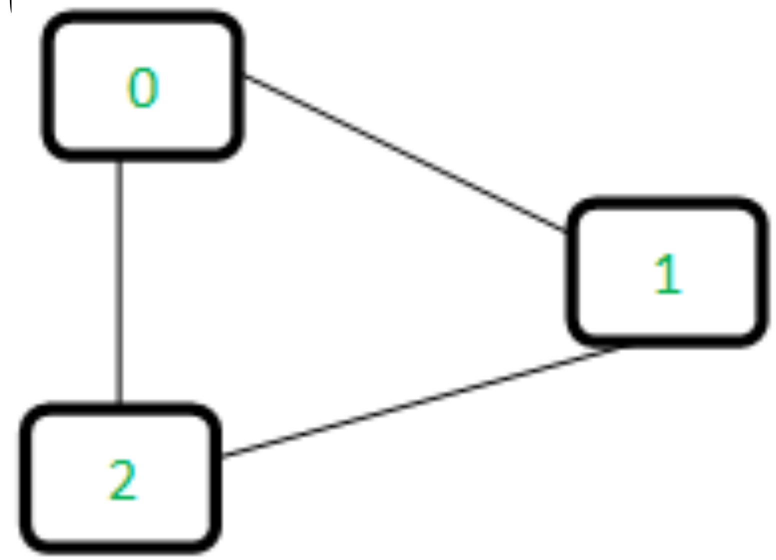
- Let us consider the following graph:



- For each edge, make subsets using both the vertices of the edge.
- If both the vertices are in the same subset, a cycle is found.
- Initially, all slots of parent array are initialized to itself. (means there is only one item in every subset).

node	0	1	2
parent	0	1	2

Union–find data structure

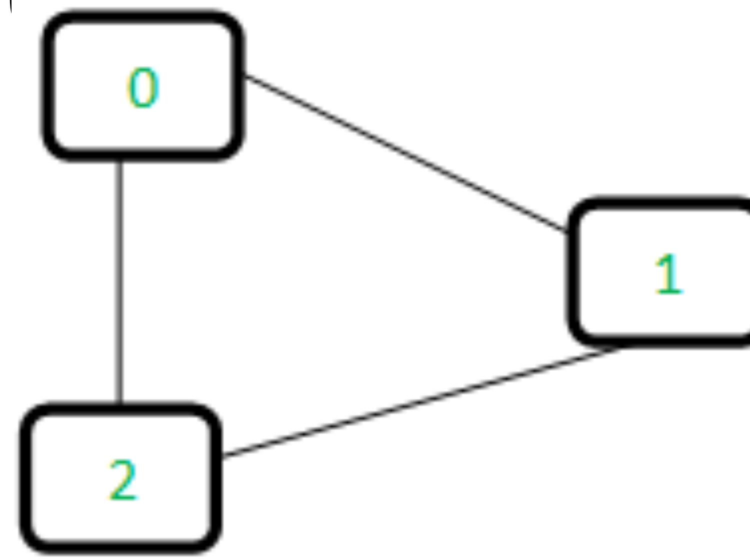


- Now process all edges one by one.
- *Edge 0-1:* Find the subsets in which vertices 0 and 1 are.
- Since they are in different subsets, we take the union of them.
- For taking the union, either **make node 0 as parent of node 1** or vice-versa.

node	**0**	**1**	**2**
parent	0	0	2

<----- 1 is made parent of 0 (1 is now representative of subset {0, 1})

Union–find data structure

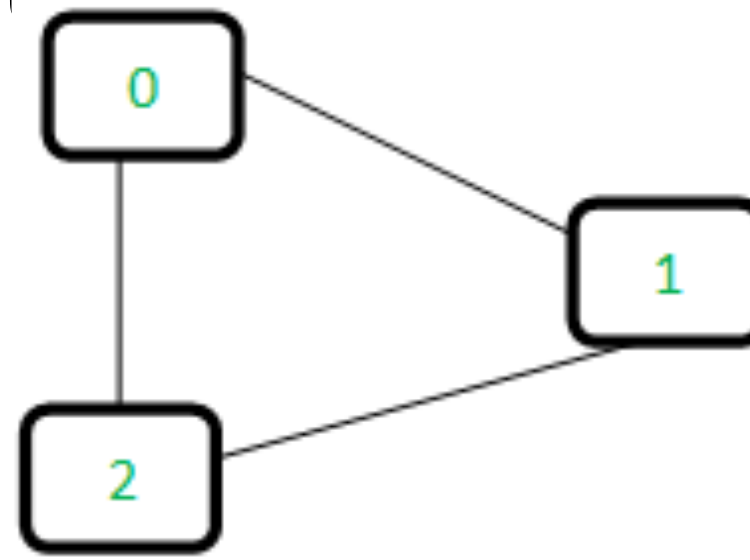


- *Edge 1-2:* 1 is in subset 1 and 2 is in subset 2. So, take union.

node	**0**	**1**	**2**
parent	0	0	0

<----- **2** is made parent of 1 (2 is now representative of subset {0, 1, 2})

Union–find data structure



- *Edge 0-2:*
  - 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

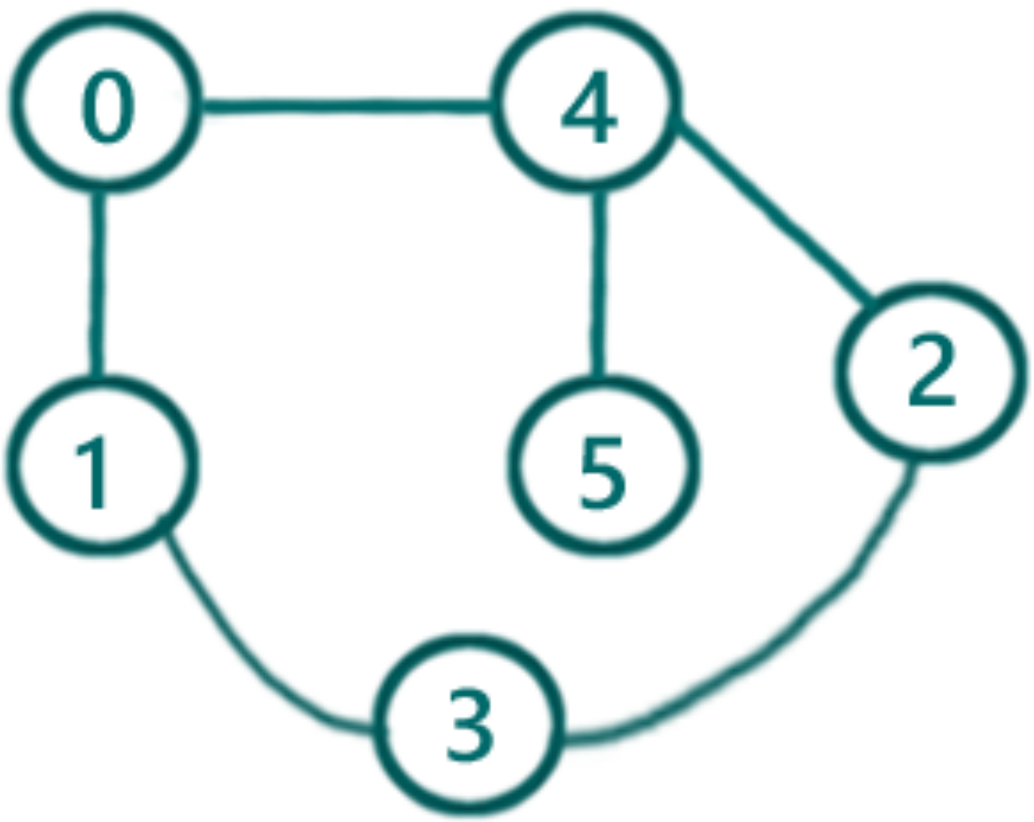
node	0	1	2
parent	0	0	0

<----- 0 is made parent of 2

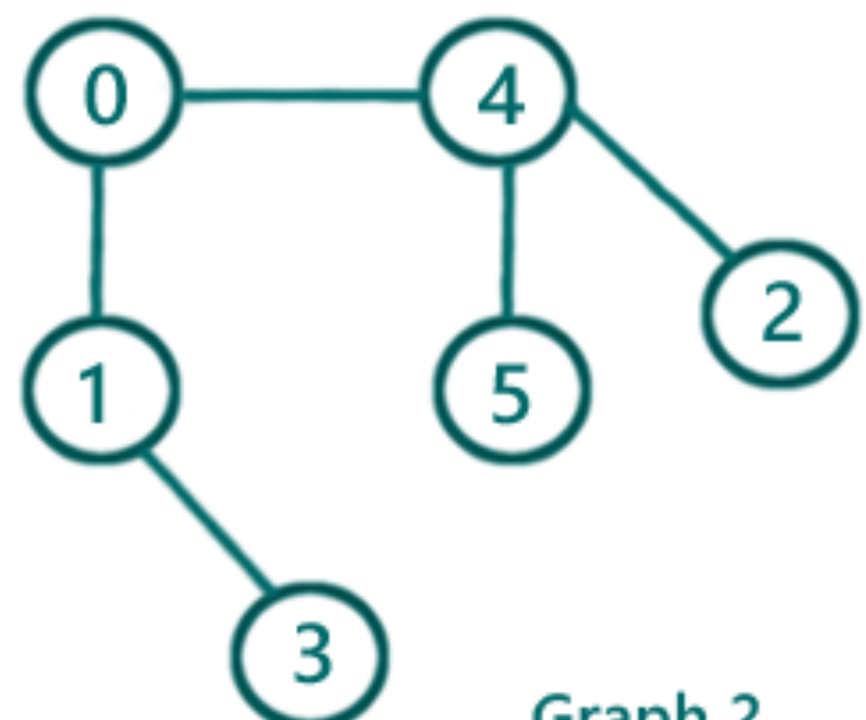
- How subset of 0 is same as 2?
  - 0->1->2 // 1 is parent of 0 and 2 is parent of 1

Union–find data structure

- Let us consider the Graph 1:

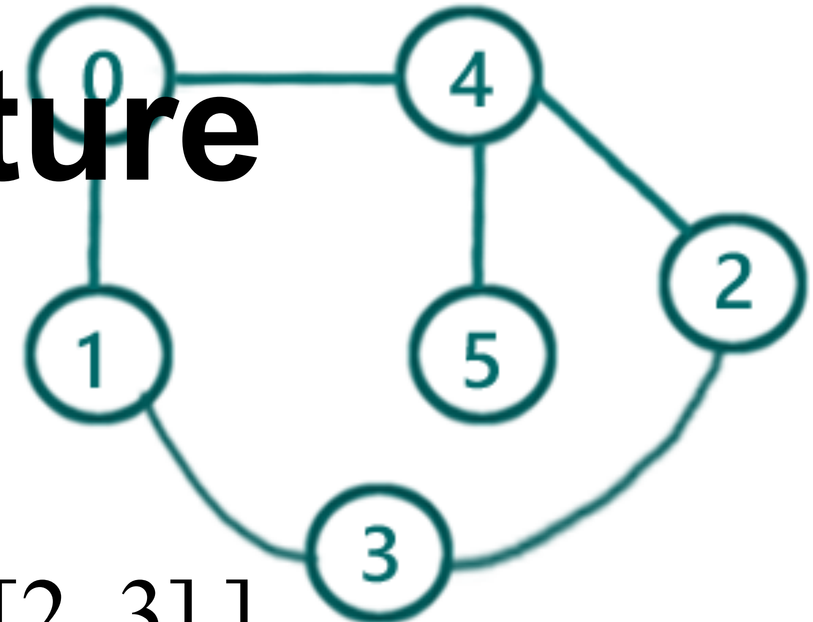


Graph 1



Graph 2

Union–find data structure



- **Dry Run:**
  - So the edge list is:

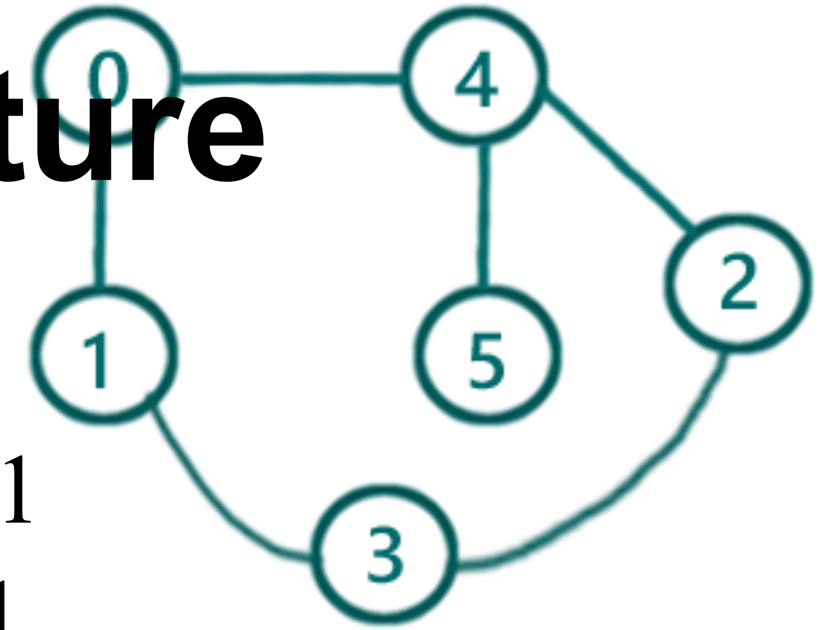
    [[0, 1], [0, 4], [1, 3], [4, 5], [4, 2], [2, 3]]

- We will process each edge one by one and do necessary FIND & UNION

- Initially, parent[] is:

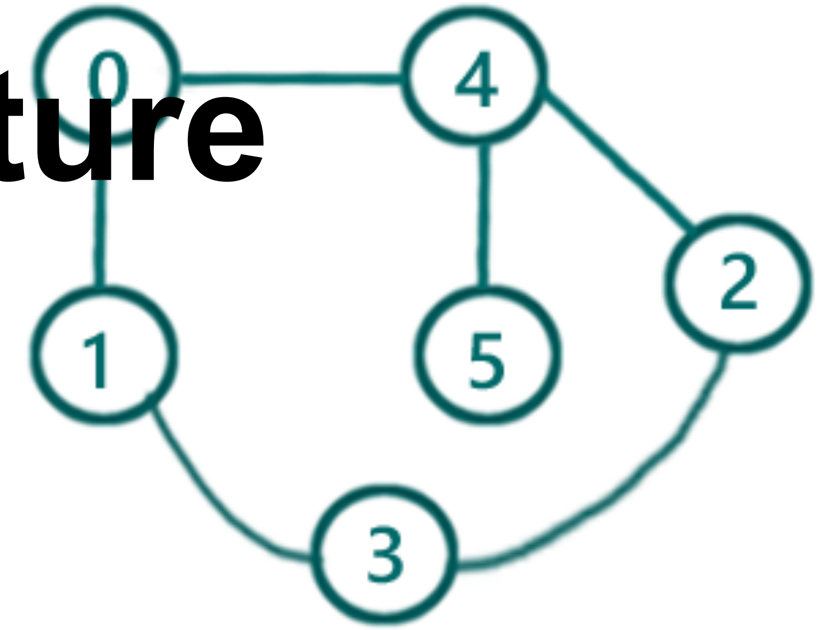
node	**0**	**1**	**2**	**3**	**4**	**5**
parent	0	1	2	3	4	5

Union–find data structure



- **Edge 0-1:**
  - So the source is 0 and destination is 1
  - We will find the parent of both 0 & 1

Union–find data structure

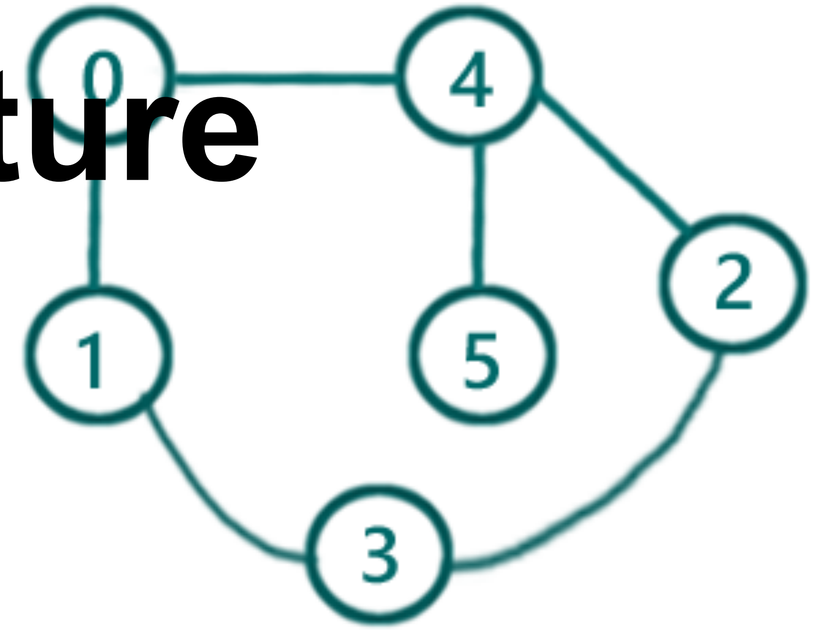


- **Edge 0-1:**
- **Finding parent of 0**
  - parent[0] is 0 so it returns 0
- **Finding parent of 1**
  - parent[1] is 1 so it returns 1
  - Since both their set names (parent) are different we do a union
- **I am skipping the rank part(you can do that your own)**
- Thus parent[1]=0 now, so after processing **edge 0-1**

node	**0**	**1**	**2**	**3**	**4**	**5**
parent	0	**0**	2	3	4	5

update

Union–find data structure



- **Edge 0-1:**
- **Finding parent of 0**
  - parent[0] is 0 so it returns 0
- **Finding parent of 1**
  - parent[1] is 1 so it returns 1
  - Since both their set names (parent) are different we do a union
- **I am skipping the rank part(you can do that your own)**
- Thus parent[1]=0 now, so after processing **edge 0-1**

node	0	1	2	3	4	5
parent	0	0	2	3	4	5

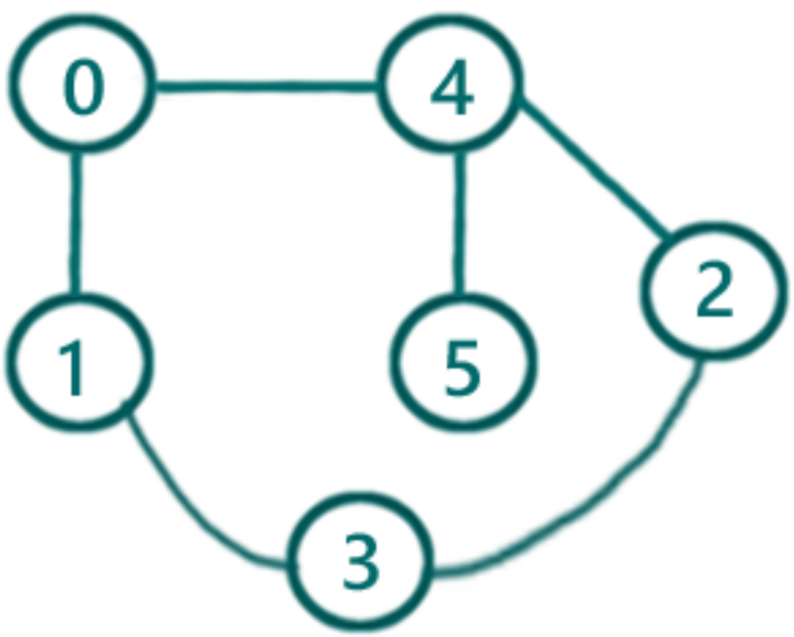
update

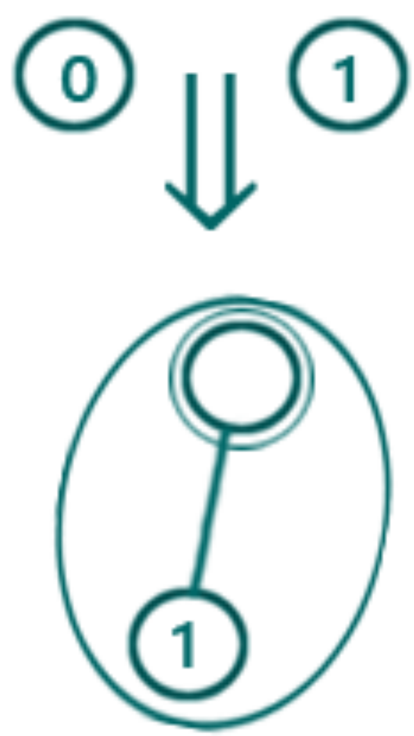
Union–find data structure

- **I am skipping the rank part(you can do that your own)**
- Thus parent[1]=0 now, so after processing **edge 0-1**

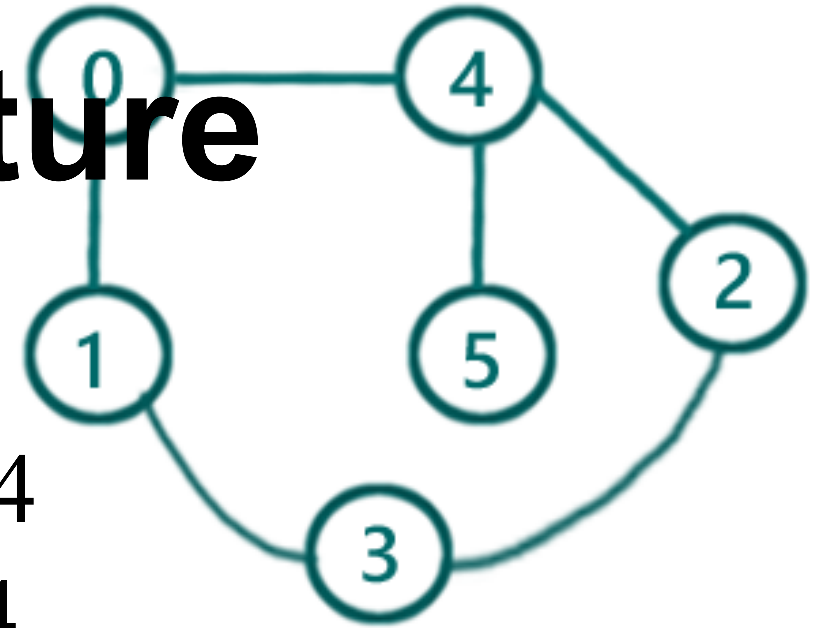
node	0	1	2	3	4	5
parent	0	**0**	2	3	4	5

update





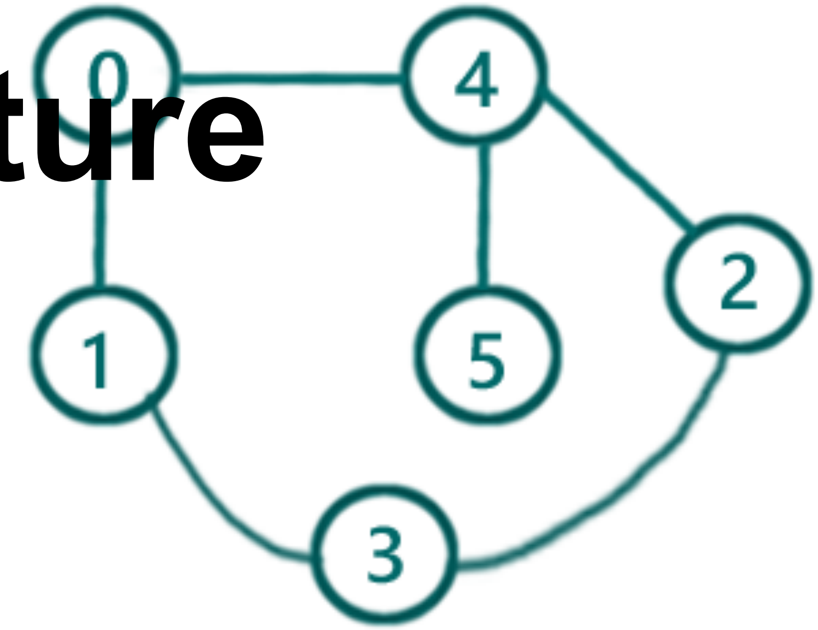
Union–find data structure



- **Edge 0-4:**
  - So the source is 0 and destination is 4
  - We will find the parent of both 0 & 4

node	0	1	2	3	4	5
parent	0	0	2	3	4	5

Union–find data structure



- **Edge 0-4:**
- **Finding parent of 0**
  - parent[0] is 0 so it returns 0
- **Finding parent of 4**
  - parent[4] is 4 so it returns 4
- Since both their set names(parent) are different we do a union
- Thus parent[4]=0 now, so after processing **edge 0-4**

node	**0**	**1**	**2**	**3**	**4**	**5**
parent	0	0	2	3	**0**	5

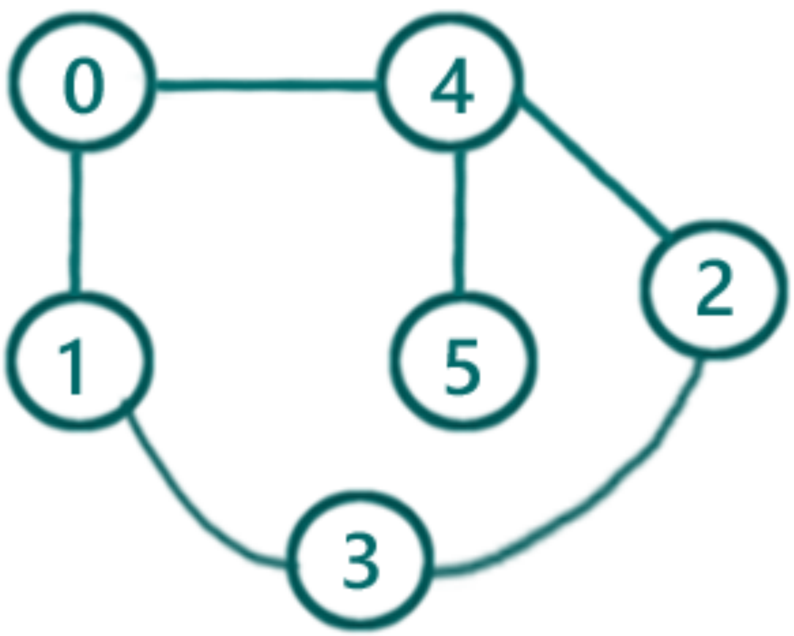
update

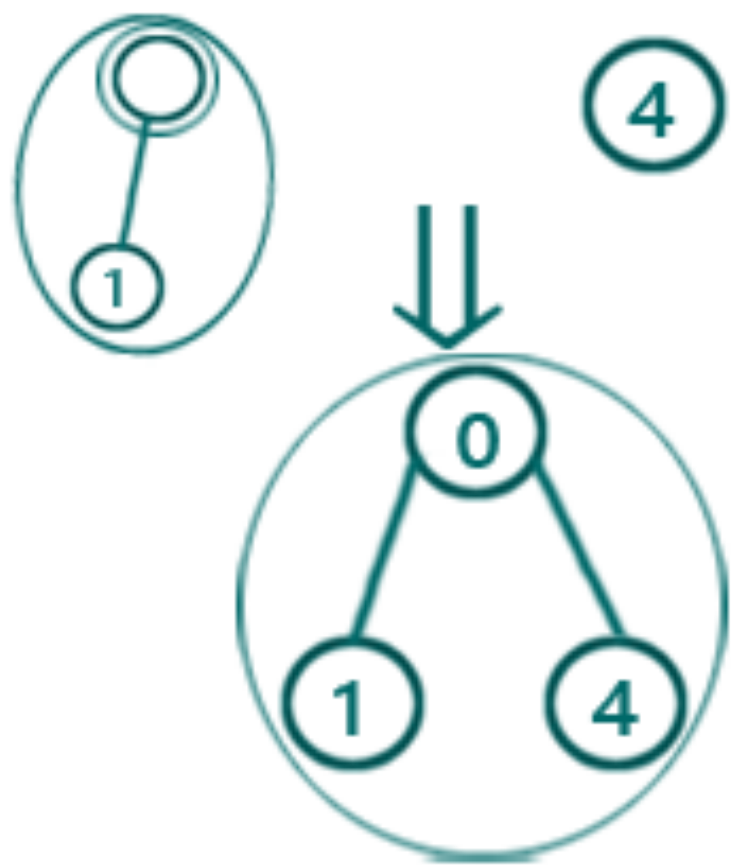
Union–find data structure

- **Edge 0-4:**

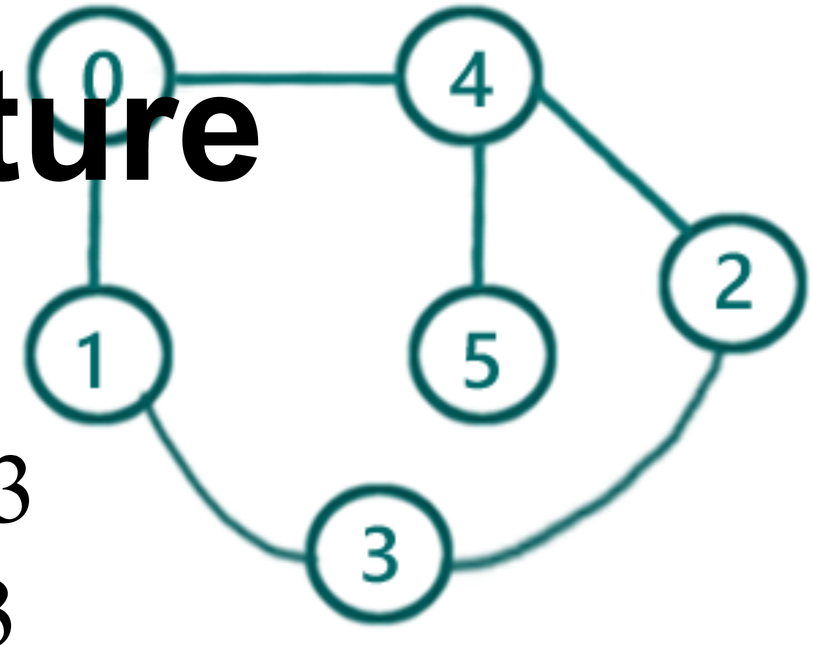
node	0	1	2	3	4	5
parent	0	0	2	3	**0**	5

update





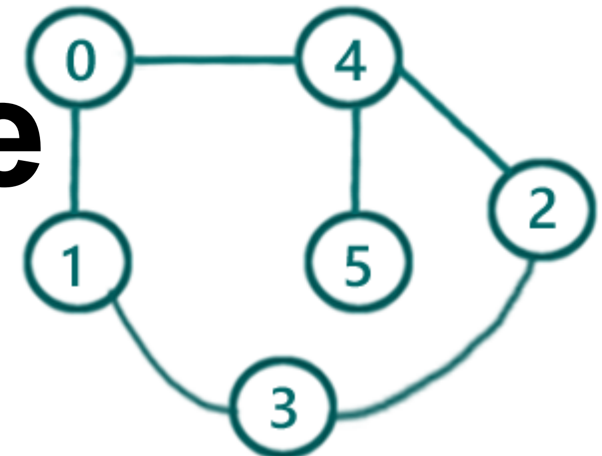
Union–find data structure



- **Edge 1-3:**
  - So the source is 1 and destination is 3
  - We will find the parent of both 1 & 3

node	0	1	2	3	4	5
parent	0	0	2	3	0	5

Union–find data structure



- **Finding parent of 1**
  - parent[1] is 0 so it returns find(0, parent) and that returns 0 ultimately (parent[0]== 0)

- **Finding parent of 3**
  - parent[3] is 3 so it returns 3
- Since both their set names (parent) are different we do a union
- Thus parent[3]=0 now, so after processing **edge 1-3**

node	**0**	**1**	**2**	**3**	**4**	**5**
parent	0	0	2	**0**	0	5

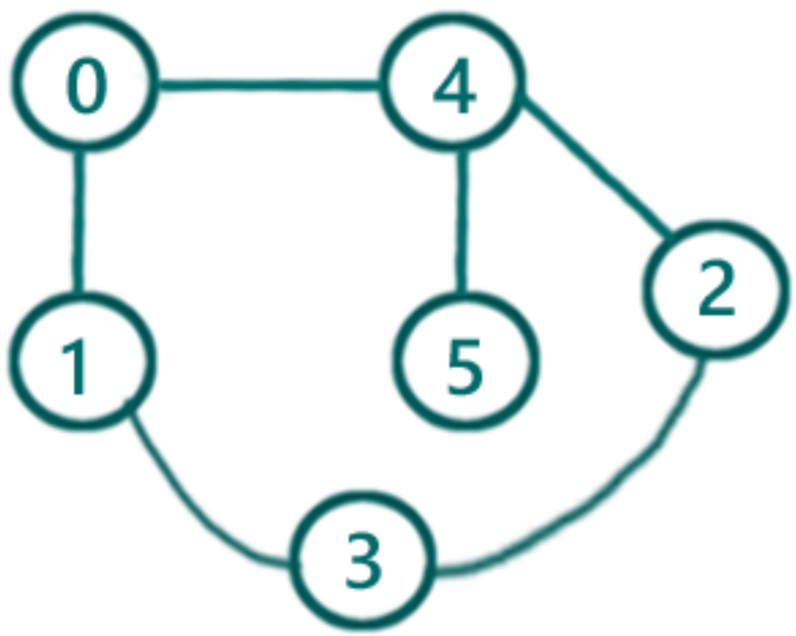
update

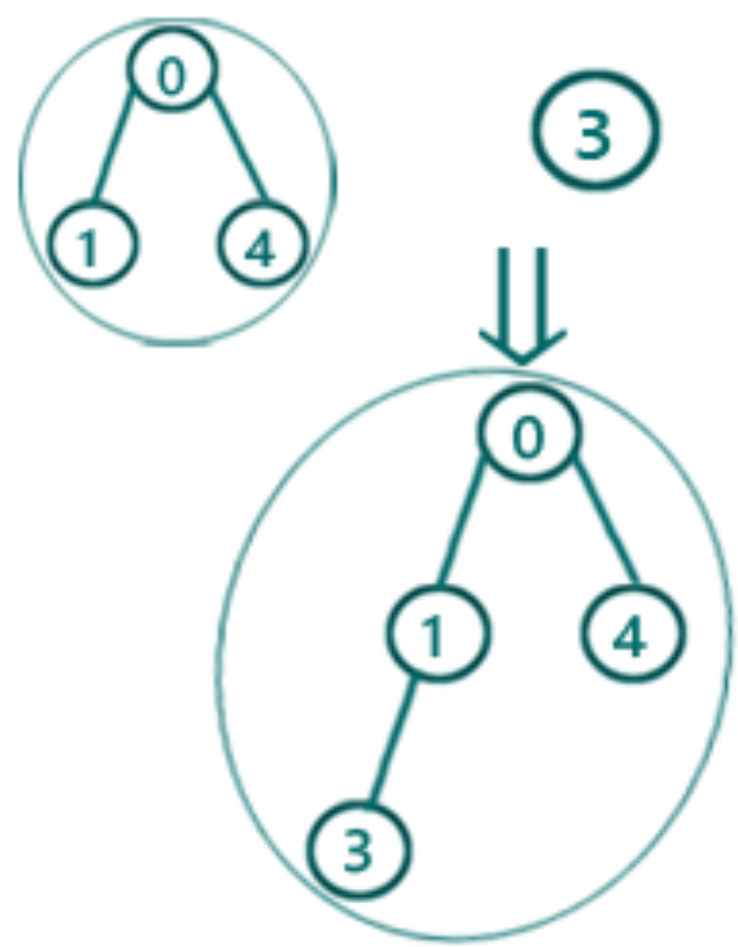
Union–find data structure

- **Edge 1-3:**

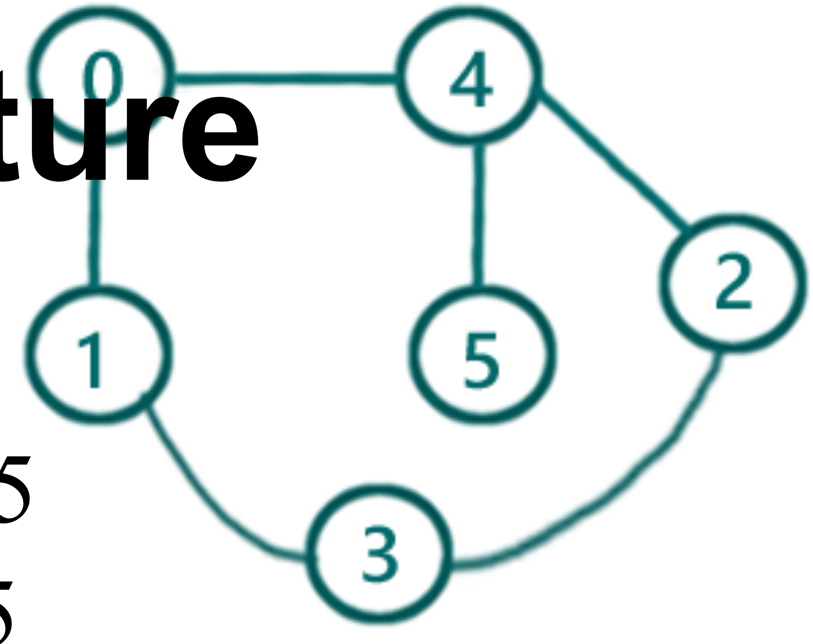
node	0	1	2	3	4	5
parent	0	0	2	**0**	0	5

update





Union–find data structure

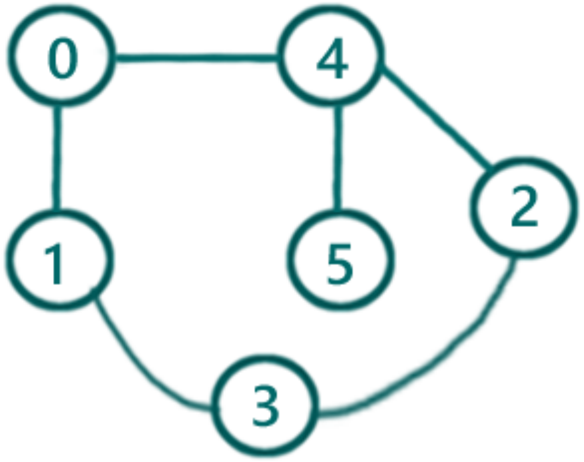


- **Edge 4-5:**
  - So the source is 4 and destination is 5
  - We will find the parent of both 4 & 5

node	**0**	**1**	**2**	**3**	**4**	**5**
parent	0	0	2	**0**	0	5

update

Union–find data structure

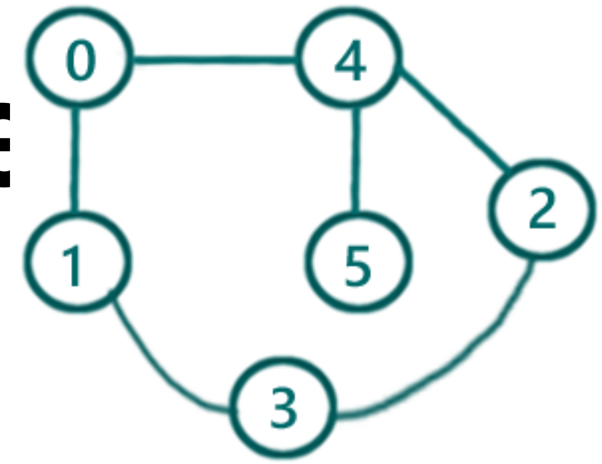


- **Finding parent of 4**
  - parent[4] is 0 so it returns find(0, parent) and that returns 0 ultimately (parent[0]==0)
- **Finding parent of 5**
  - parent[5] is 5 so it returns 5
- Since both their set names(parent) are different we do a union
- Thus parent[5]=0 now, so after processing **edge 4-5**

node	**0**	**1**	**2**	**3**	**4**	**5**
parent	0	0	2	0	0	**0**

update

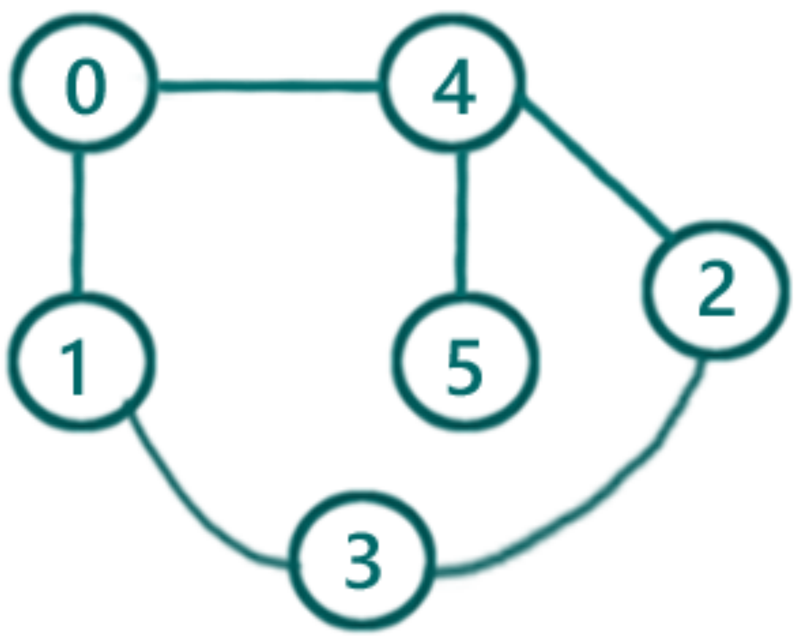
Union–find data structure

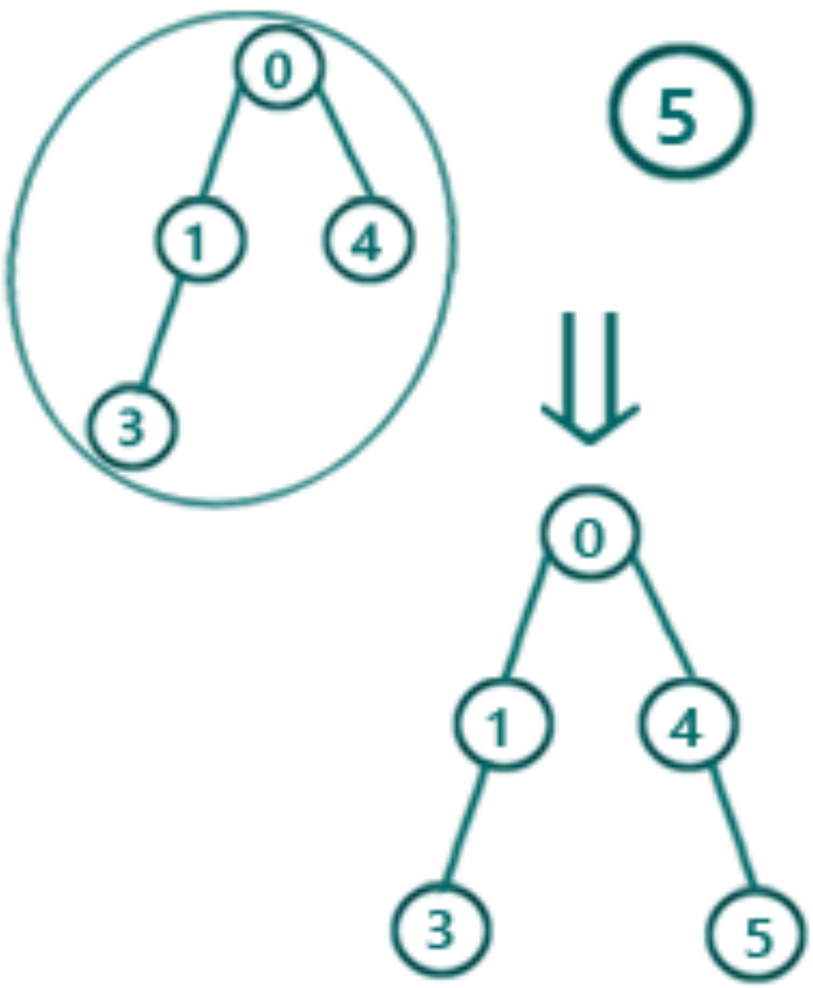


- **Edge 4-5:**

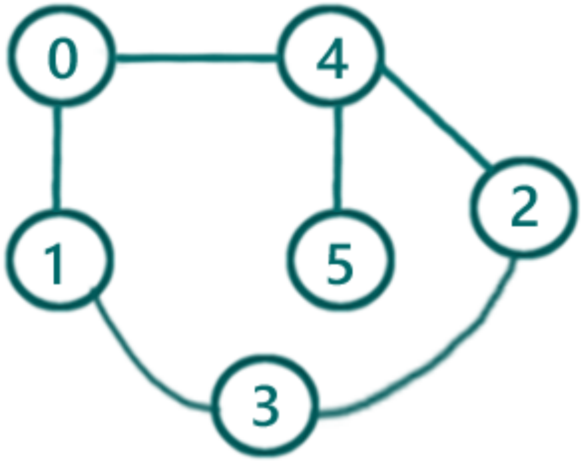
node	0	1	2	3	4	5
parent	0	0	2	0	0	**0**

update





Union–find data structure

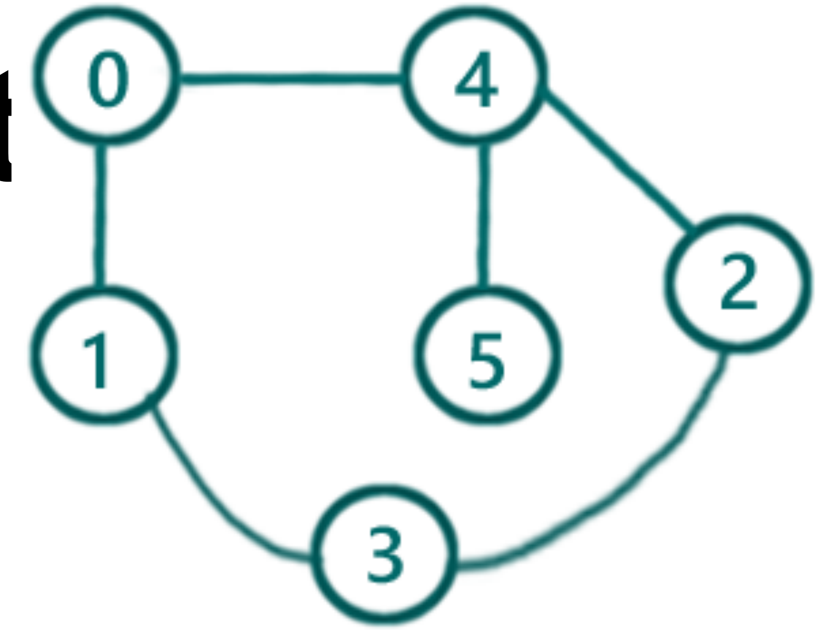


- **Edge 4-2 :**
  - So the source is 4 and destination is 2
  - We will find the parent of both 4 & 2

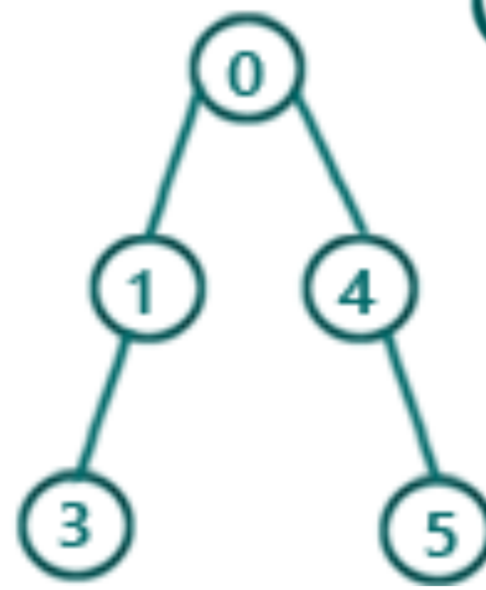
node	0	1	2	3	4	5
parent	0	0	2	0	0	0

update

Union–find data struct



- **Edge 4-2:**
- **Finding parent of 4**
  - parent[4] is 0 so it returns 0
- **Finding parent of 2**
  - parent[2] is 2 so it returns 2

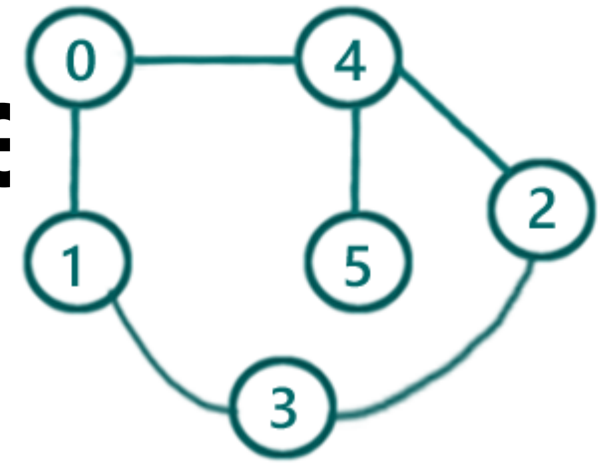


- Since both their set names(parent) are different we do a union
- Thus parent[2]=0 now, so after processing **edge 4-2**

node	**0**	**1**	**2**	**3**	**4**	**5**
parent	0	0	**0**	0	0	0

update

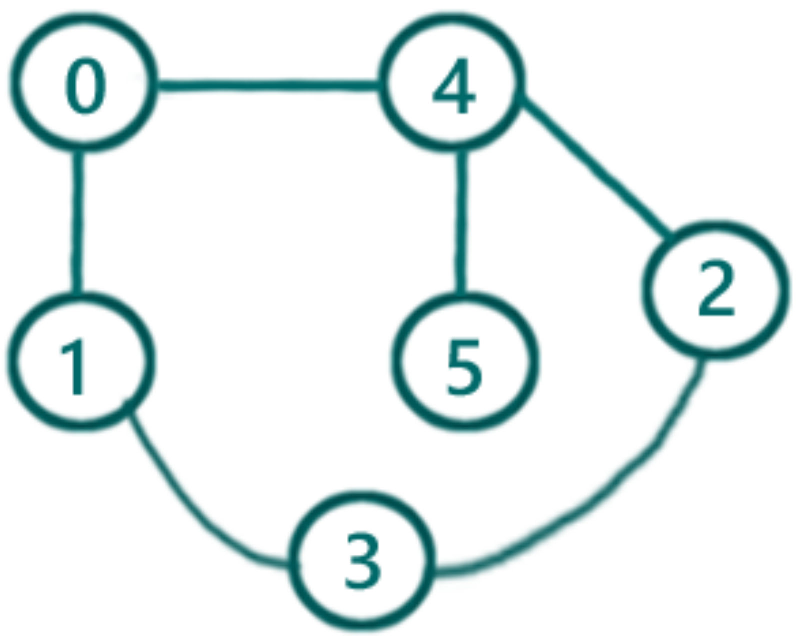
Union–find data structure

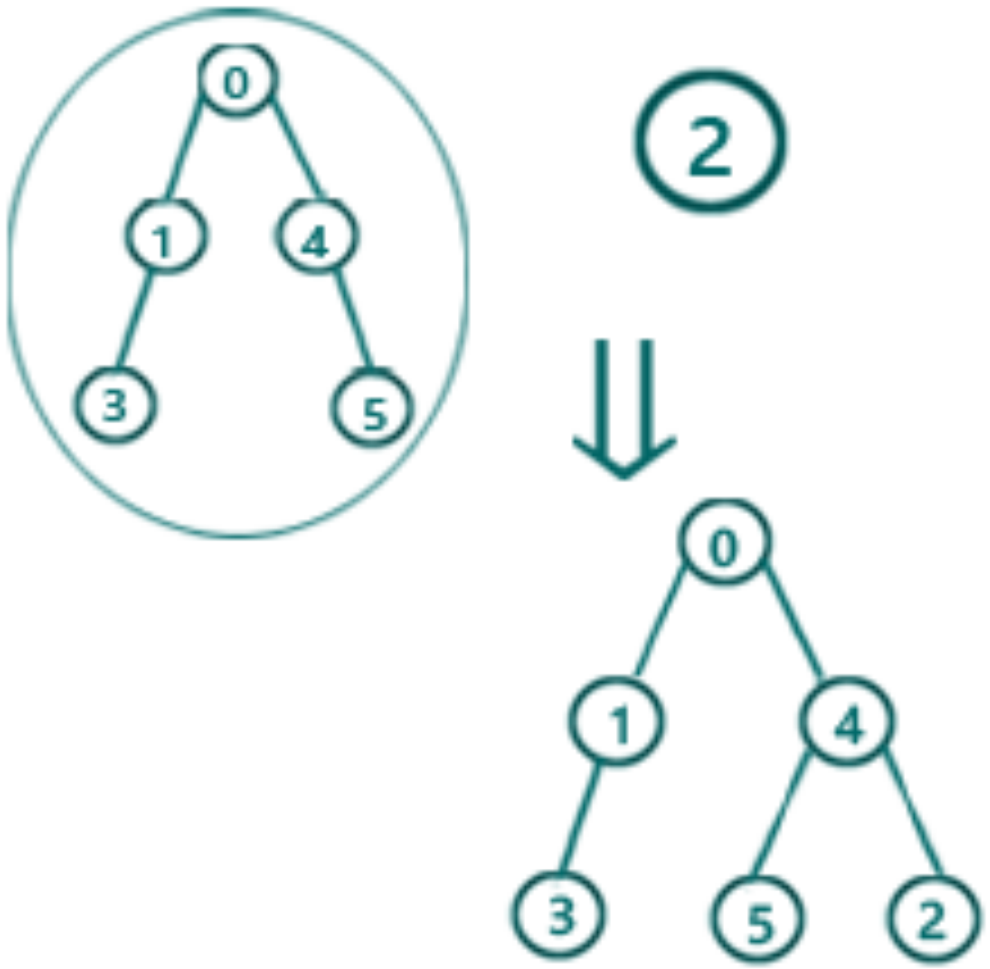


- **Edge 4-2:**

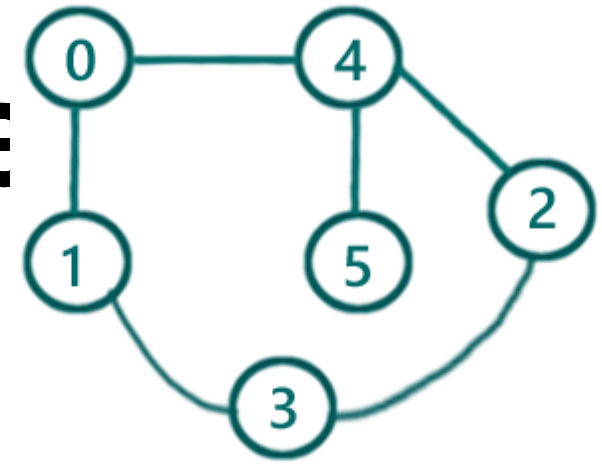
node	0	1	2	3	4	5
parent	0	0	0	0	0	0

update





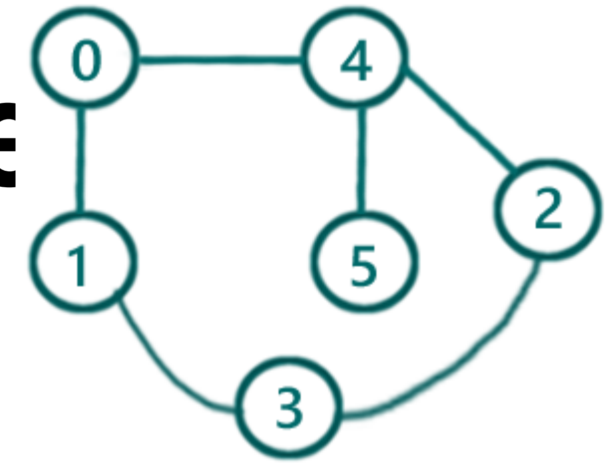
Union–find data structure



- **Edge *2-3*:**
  - So the source is 2 and destination is 3
  - We will find the parent of both 2 & 3

node	**0**	**1**	**2**	**3**	**4**	**5**
parent	0	0	0	0	0	0

Union–find data structure



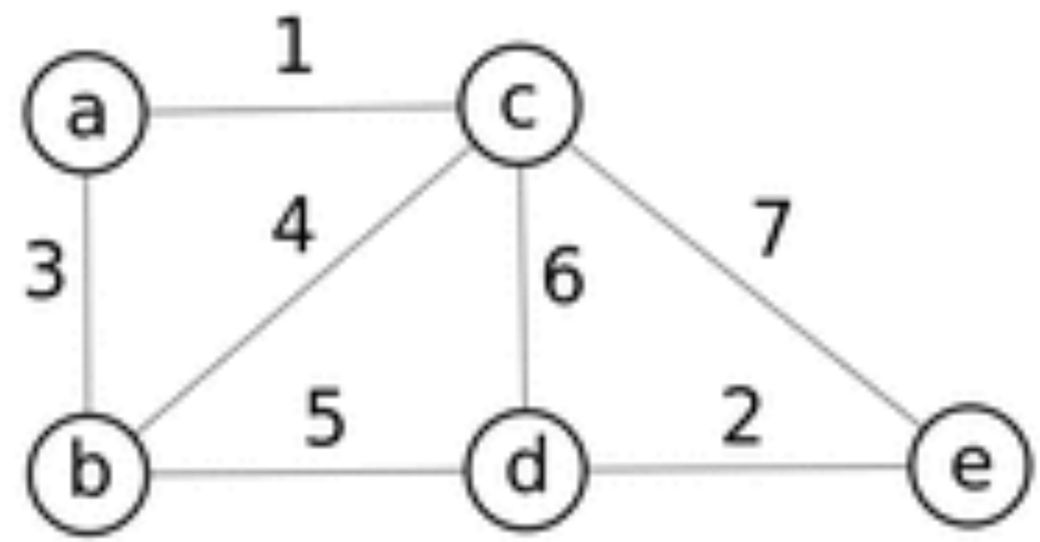
- **Edge *2-3*:**
- **Finding parent of 2**
  - parent[2] is 0 so it returns find(0, parents) which ultimately returns 0
- **Finding parent of 3**
  - parent[3] is 0 and thus it returns 0
- So, that means both of the nodes already belongs to the same set, that means this edge result in a **cycle**.

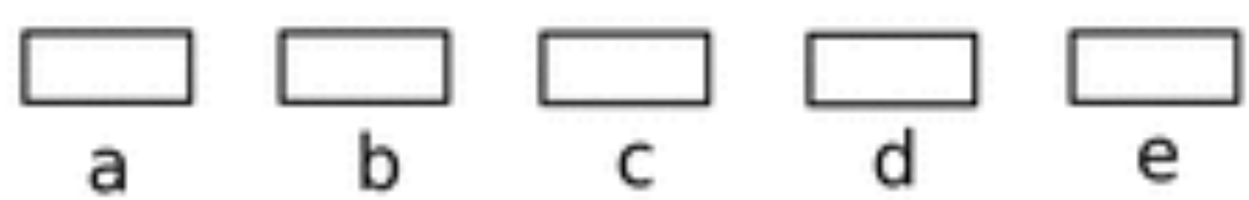
node	0	1	2	3	4	5
parent	0	0	0	0	0	0

Same parent

Kruskal's algorithm

- A demo for Union-Find when using Kruskal's algorithm to find minimum spanning tree.





Kruskal's algorithm

```
// Java program for Kruskal's algorithm to find
// Minimum Spanning Tree of a given connected,
// undirected and weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;
```

Kruskal's algorithm

```
class Graph {
    // A class to represent a graph edge
    class Edge implements Comparable<Edge> {
        int src, dest, weight;
        // Comparator function used for sorting
        // edgesbased on their weight
        public int compareTo(Edge compareEdge) {
            return this.weight - compareEdge.weight;
        }
    };

    // A class to represent a subset for union-find
    class subset {
        int parent, rank;
    };
```

Kruskal's algorithm

```
int V, E; // V-> no. of vertices & E->no.of edges
Edge edge[]; // collection of all edges

// Creates a graph with V vertices and E edges
Graph(int v, int e) {
    V = v;
    E = e;
    edge = new Edge[E];
    for (int i = 0; i < e; ++i)
        edge[i] = new Edge();
}
```

Kruskal's algorithm

```
// A function that does union of two sets of x and y (uses union by rank)
void Union(subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as
    // root and increment its rank by one
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
```

Kruskal's algorithm

```
// The main function to construct MST using Kruskal's algorithm
void KruskalMST() {
    // Tnis will store the resultant MST
    Edge result[] = new Edge[V];

    // An index variable, used for result[]
    int e = 0;

    // An index variable, used for sorted edges
    int i = 0;
    for (i = 0; i < V; ++i)
        result[i] = new Edge();
```

// The main function to construct MST using Kruskal's algorithm

```
// Step 1: Sort all the edges in non-decreasing order
// of their weight.
// If we are not allowed to change the given graph,
// we can create a copy of  array of edges
Arrays.sort(edge);

// Allocate memory for creating V ssubsets
subset subsets[] = new subset[V];
for (i = 0; i < V; ++i)
     subsets[i] = new subset();
```

// The main function to construct MST using Kruskal's algorithm

```
// Step 1: Sort all the edges in non-decreasing order
// of their weight.
// If we are not allowed to change the given graph,
// we can create a copy of  array of edges
Arrays.sort(edge);

// Allocate memory for creating V ssubsets
subset subsets[] = new subset[V];
for (i = 0; i < V; ++i)
    subsets[i] = new subset();
```

// The main function to construct MST using Kruskal’s algorithm

```
// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

i = 0; // Index used to pick next edge
```

// The main function to construct MST using Kruskal’s algorithm

```
// Number of edges to be taken is equal to V-1
while (e < V - 1) {
    // Step 2: Pick the smallest edge.
    // And increment the index for next iteration
    Edge next_edge = edge[i++];
    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);
    // If including this edge does't cause cycle, include it in
    // result and increment the index of result for next edge
    if (x != y) {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
     // Else discard the next_edge
}
```

// The main function to construct MST using Kruskal’s algorithm

```
        // print the contents of result[] to display the built MST
        System.out.println("Following are the edges in ”
                        + "the constructed MST");
        int minimumCost = 0;
        for (i = 0; i < e; ++i)
        {
            System.out.println(result[i].src + " -- ” + result[i].dest
                            + " == " + result[i].weight);
            minimumCost += result[i].weight;
        }
        System.out.println("Minimum Cost Spanning Tree ”
                            + minimumCost);
    }
```

Kruskal's algorithm

```
    // Driver Code
    public static void main(String[] args) {
        int V = 4; // Number of vertices in graph
        int E = 5; // Number of edges in graph
        Graph graph = new Graph(V, E);
        // add edges 0-1, 0-2, 0-3, 1-3, 2-3

            //See next page

        // Function call
        graph.KruskalMST();
    }
}
```

```
/* Let us create following weighted graph
          10
     0--------1
     | \      |
    6|  5\    |15
     |     \  |
     2--------3
          4       */
```

Kruskal's algorithm

```
// add edges 0-1, 0-2, 0-3, 1-3, 2-3
graph.edge[0].src = 0;
graph.edge[0].dest = 1;
graph.edge[0].weight = 10;
graph.edge[1].src = 0;
graph.edge[1].dest = 2;
graph.edge[1].weight = 6;
graph.edge[2].src = 0;
graph.edge[2].dest = 3;
graph.edge[2].weight = 5;
graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 15;
graph.edge[4].src = 2;
graph.edge[4].dest = 3;
graph.edge[4].weight = 4;
```

```
/* Let us create following weighted graph
         10
    0--------1
    | \      |
   6|  5\    |15
    |     \  |
    2--------3
         4       */
```

- Minimum Spanning Tree
- Kruskal's algorithm
- **Prim's algorithm**
  - **matrix representation**
  - adjacency list representation

Prim's algorithm

- The idea is to **maintain two sets of vertices**.
- The first set contains the vertices already included in the MST, the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets, and **picks the minimum weight edge** from these edges.
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.

How does Prim's Algorithm Work?

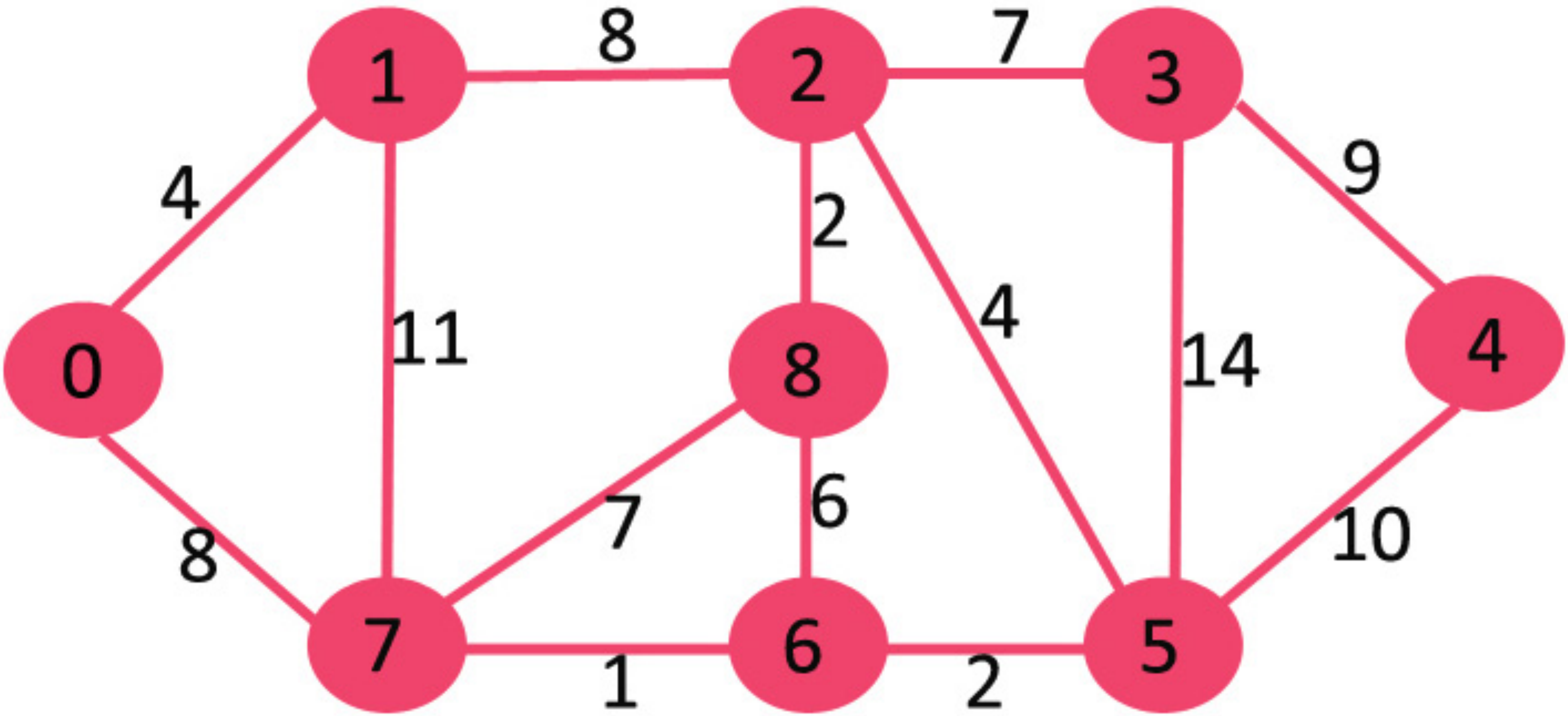
- A spanning tree means all vertices must be connected.
- So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree.
- And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

Prim's Algorithm - matrix representation

- **1)** Create a set *mstSet* that keeps track of vertices already included in MST.
- **2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- **3)** While mstSet doesn't include all vertices
  - **a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
  - **b)** Include *u* to mstSet.
  - **c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

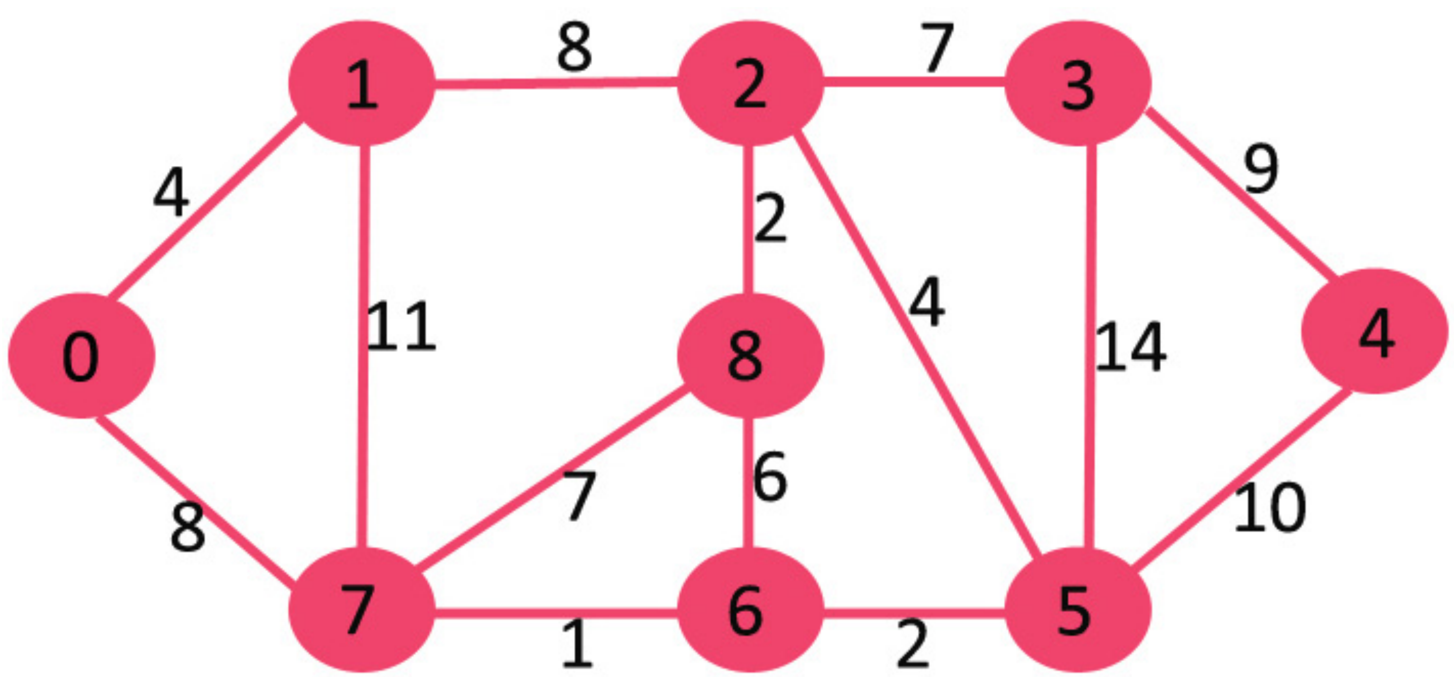
Example

- Let us understand with the following example:



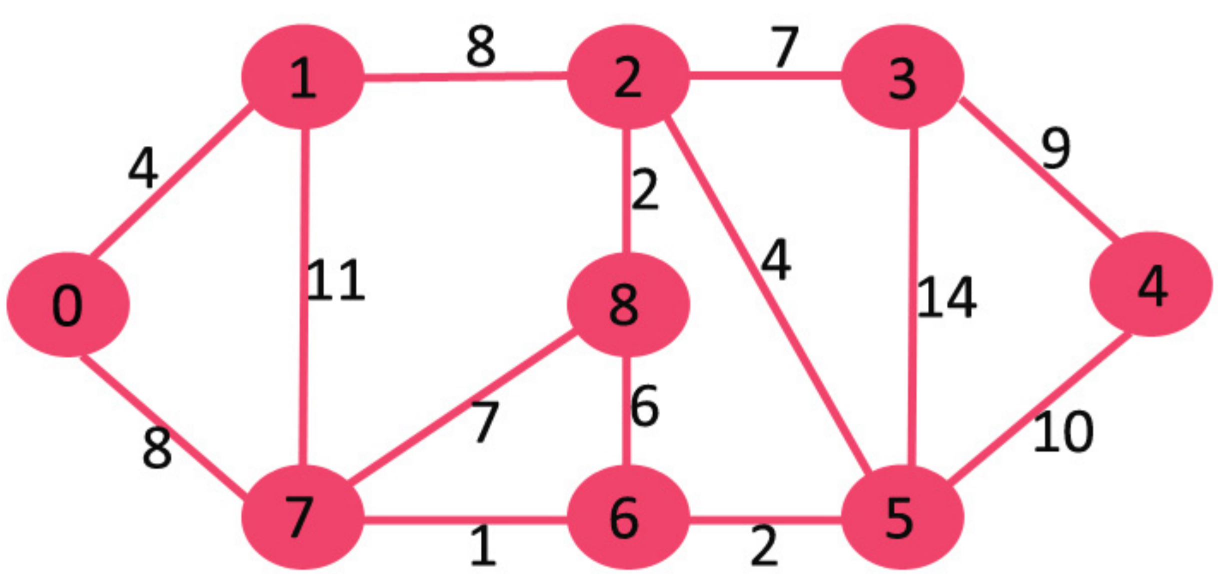
Example

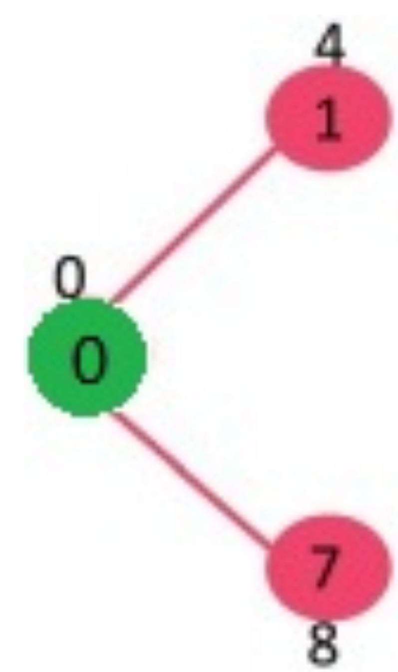
- The set *mstSet* is initially empty and keys assigned to vertices are {**0**, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.
- Now pick the vertex with the minimum key value.
- The vertex 0 is picked, include it in *mstSet*.
- So *mstSet* becomes **{0}**.



Example

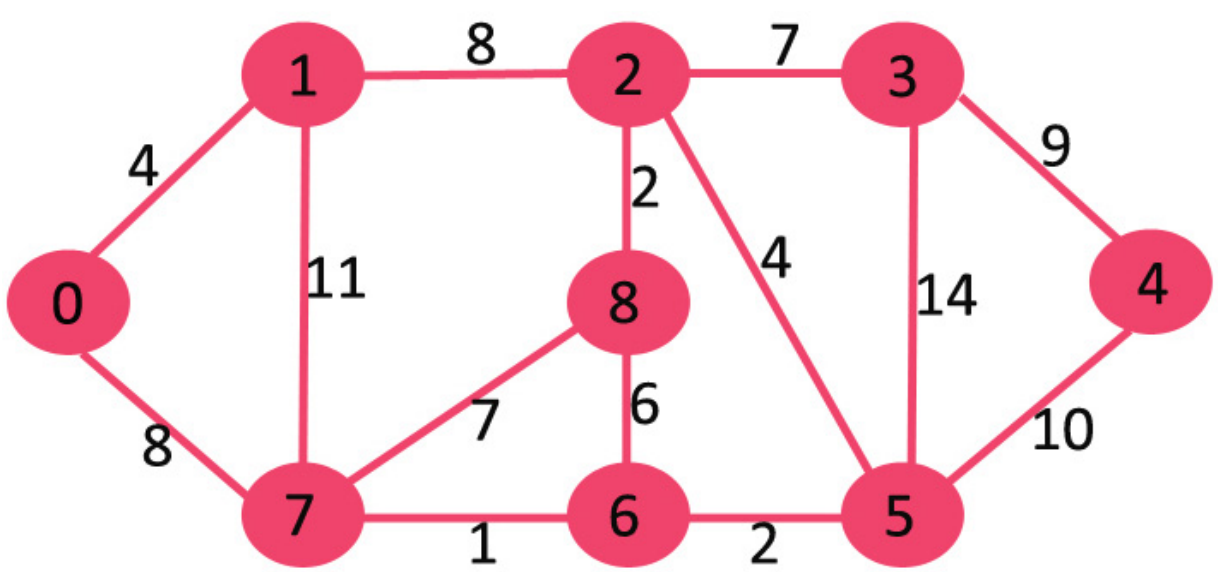
- After including to *mstSet*, update key values of adjacent vertices.
- Adjacent vertices of 0 are 1 and 7.
- The key values of 1 and 7 are updated as 4 and 8.
- Following subgraph shows vertices and their key values, only the vertices with finite key values are shown.
- The vertices included in MST are shown in green color.

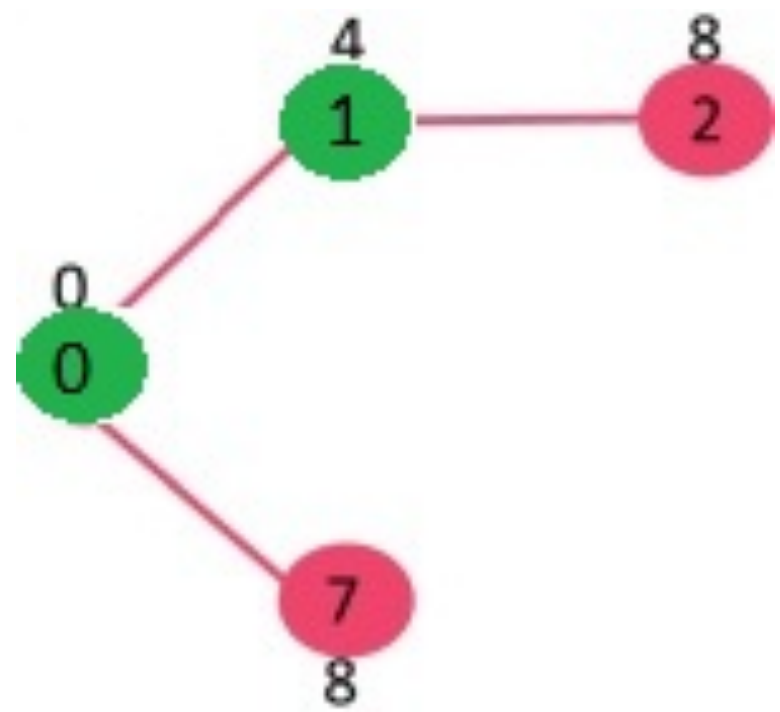




Example

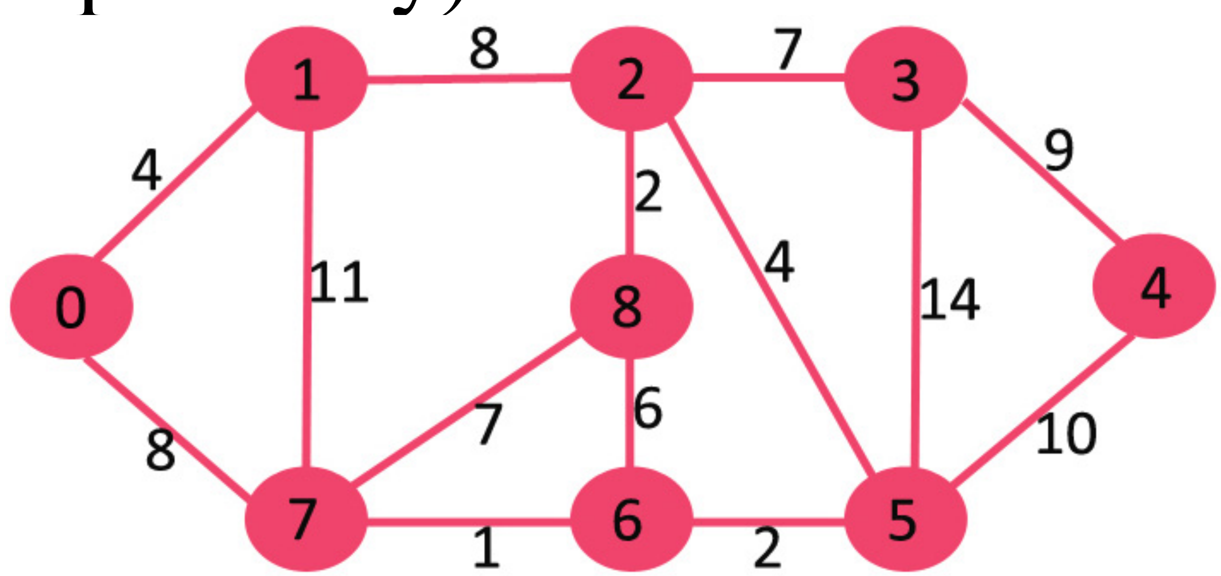
- Pick the vertex with minimum key value and not already included in MST (not in mstSET).
- The vertex 1 is picked and added to mstSet.
- So mstSet now becomes **{0, 1}**.
- Update the key values of adjacent vertices of 1.
- The key value of vertex 2 becomes 8.

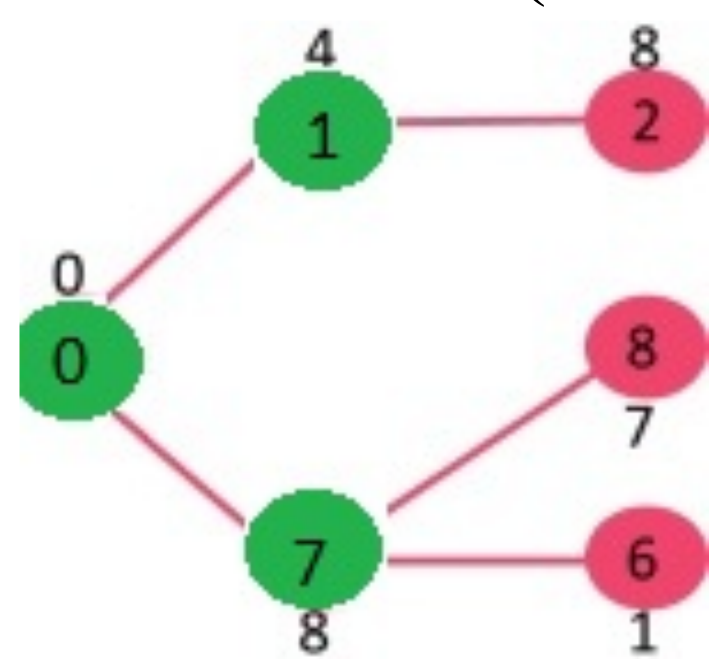




Example

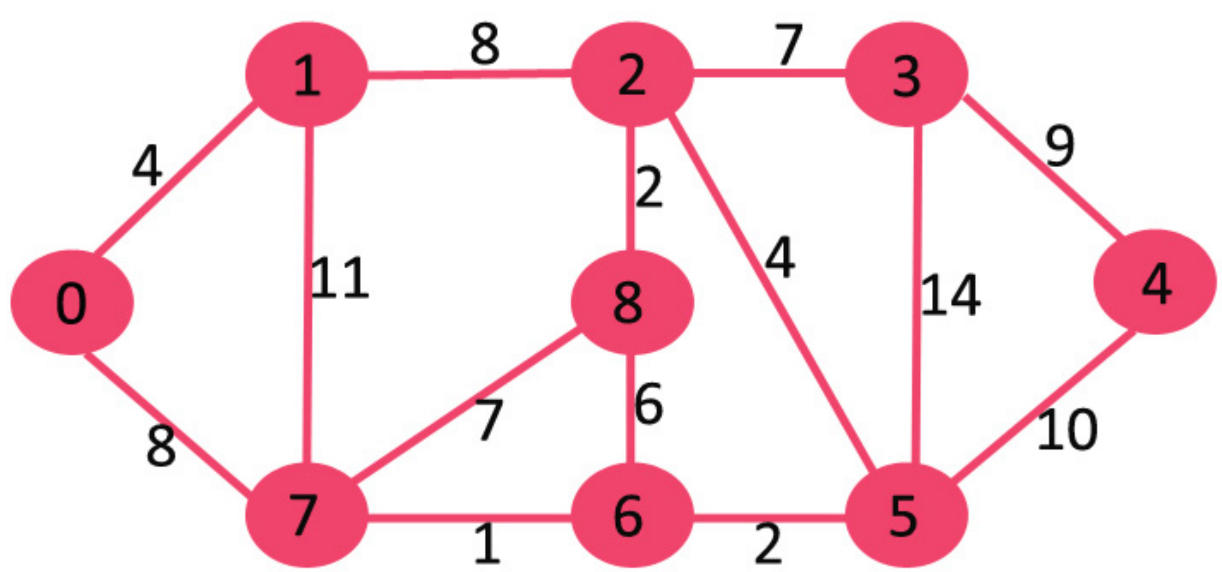
- Pick the vertex with minimum key value and not already included in MST (not in mstSET).
- We can either pick vertex 7 or vertex 2, let vertex 7 is picked.
- So mstSet now becomes **{0, 1, 7}**.
- Update the key values of adjacent vertices of 7.
- The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).

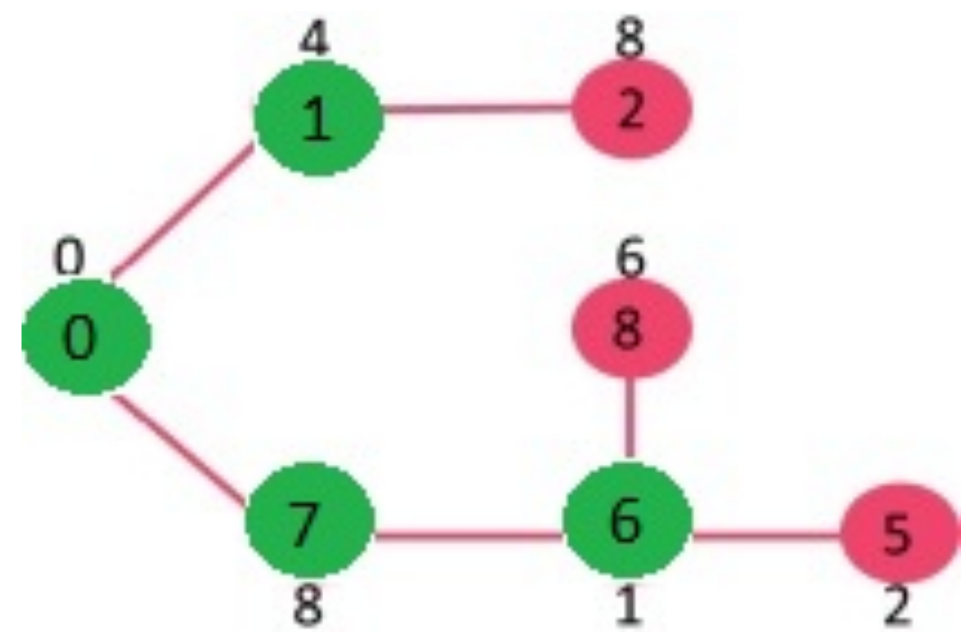




Example

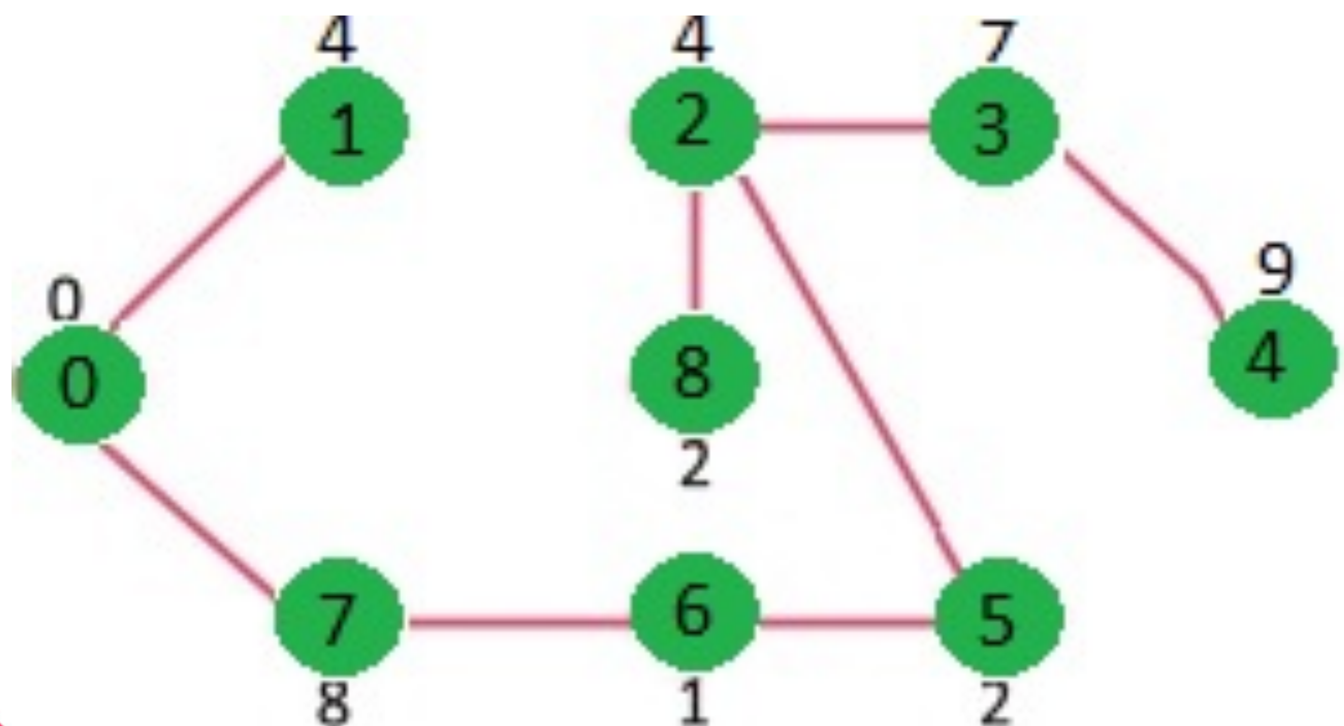
- Pick the vertex with minimum key value and not already included in MST (not in mstSET).
- Vertex 6 is picked.
- So mstSet now becomes **{0, 1, 7, 6}**.
- Update the key values of adjacent vertices of 6.
- The key value of vertex 5 and 8 are updated.

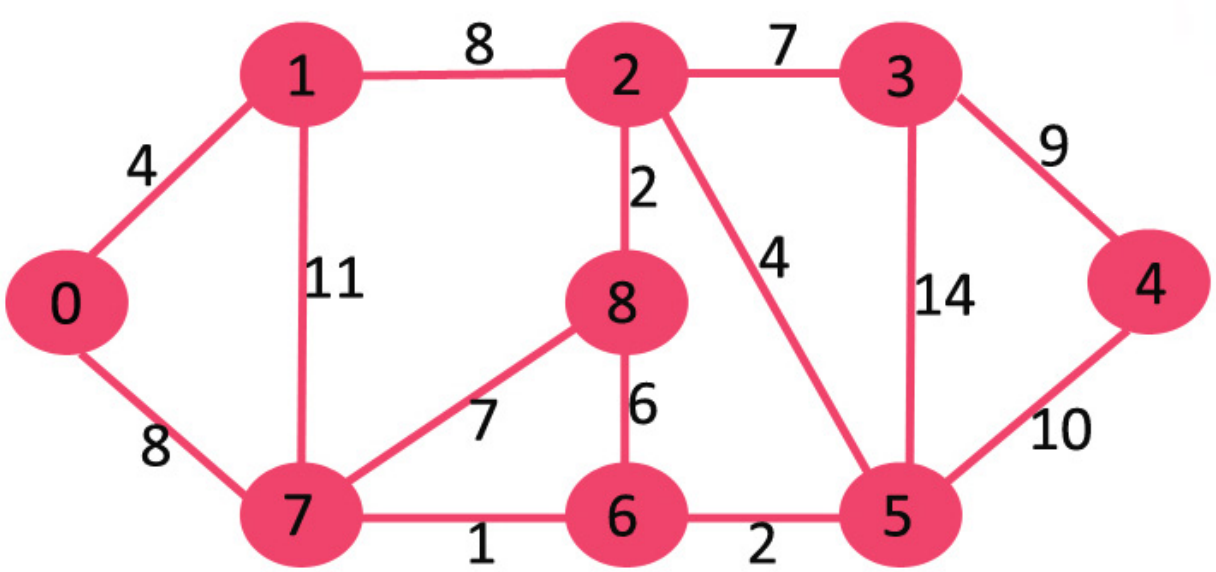




Example

- We repeat the above steps until *mstSet* includes all vertices of given graph.
- Finally, we get the following graph.





How to implement the above algorithm?

- We use a **boolean array** mstSet[] to represent the set of vertices included in MST.
- If a value mstSet[v] is true, then vertex v is included in MST, otherwise not.
- Array key[] is used to store key values of all vertices.
- Another array parent[] to store indexes of parent nodes in MST.
- The parent array is the output array which is used to show the constructed MST.

Time Complexity

- Time Complexity of the Prim's Algorithm with matrix representation is
  - $O(|V|^2)$

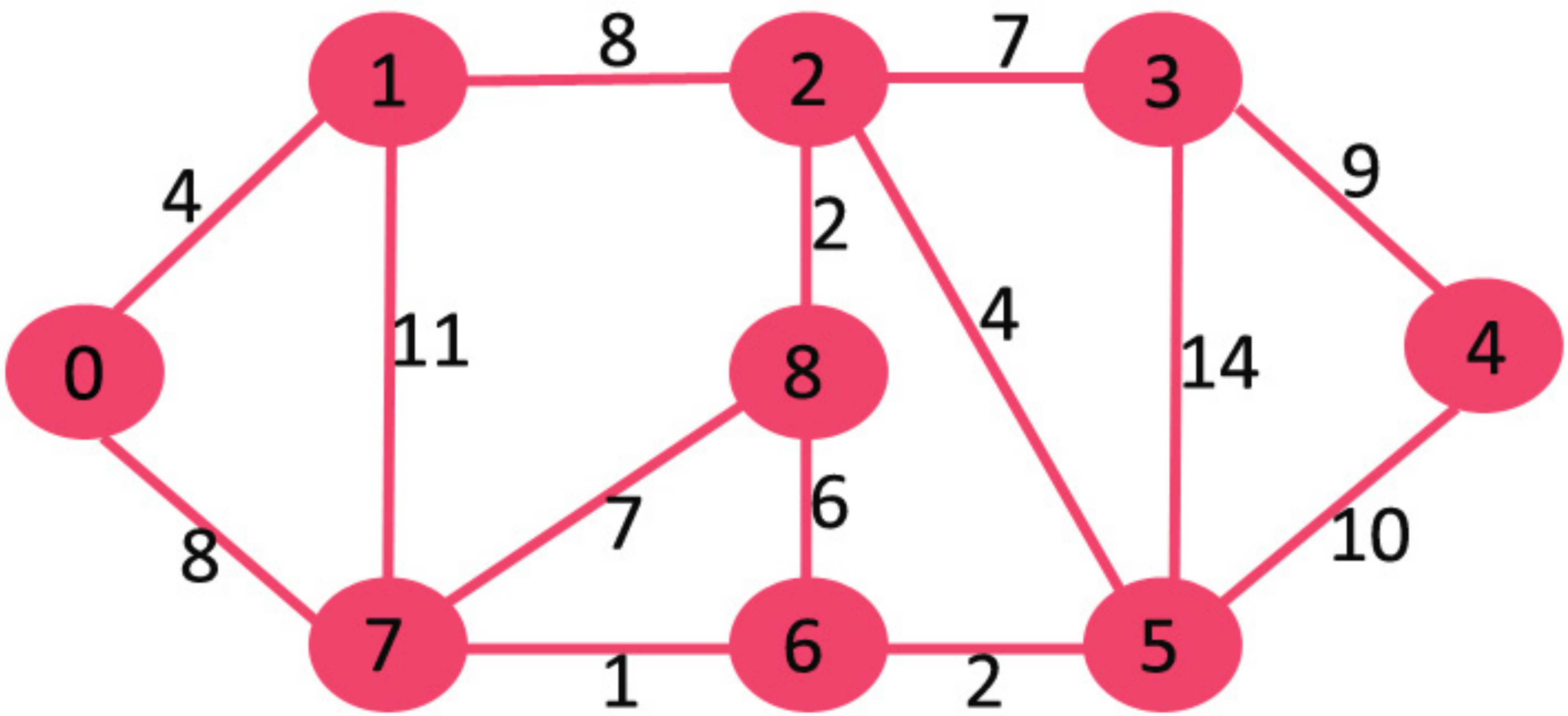
- Minimum Spanning Tree
- Kruskal's algorithm
- **Prim's algorithm**
  - matrix representation
  - **adjacency list representation**

Prim's Algorithm - adjacency list representation

- **1)** Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains **vertex number** and **key value** of the vertex.
- **2)** Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).
- **3)** While Min Heap is not empty, do following
  - **a)** Extract the min value node from Min Heap. Let the extracted vertex be u.
  - **b)** For every adjacent vertex v of u, check if v is in Min Heap (not yet included in MST). If v is in Min Heap and its key value is more than weight of u-v, then update the key value of v as weight of u-v.

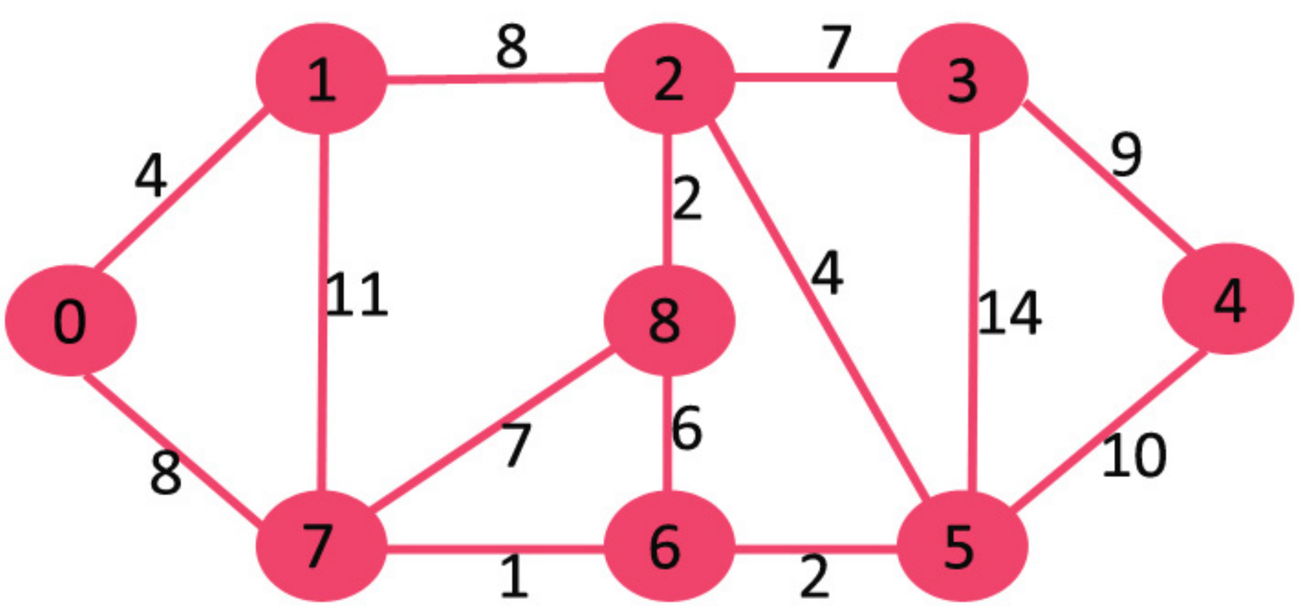
Example

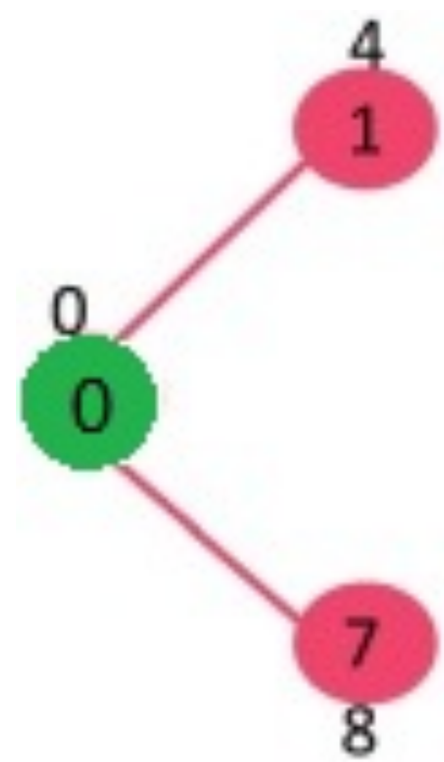
- Let us understand the above algorithm with the following example:



Example

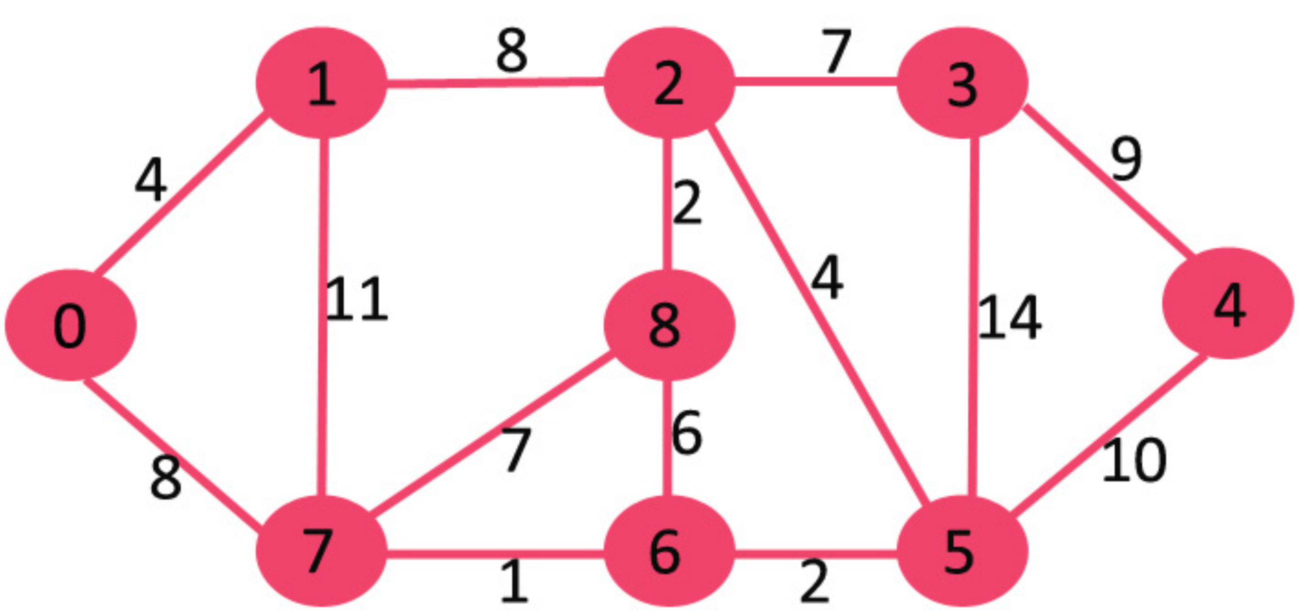
- Initially, key value of first vertex is 0 and INF (infinite) for all other vertices.
- So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated.
- Min Heap contains all vertices except vertex 0.
- The vertices in green color are the vertices included in MST.

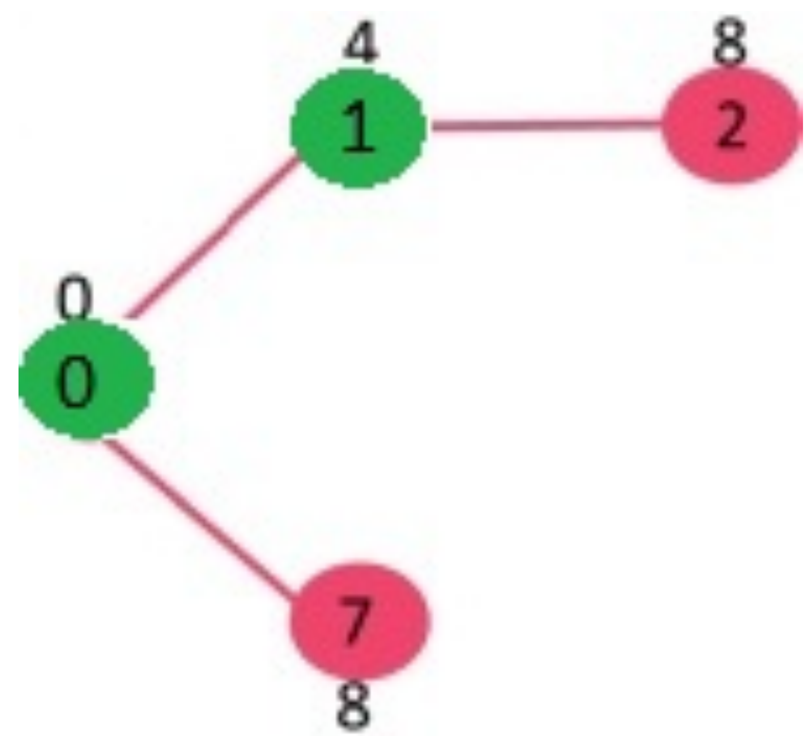




Example

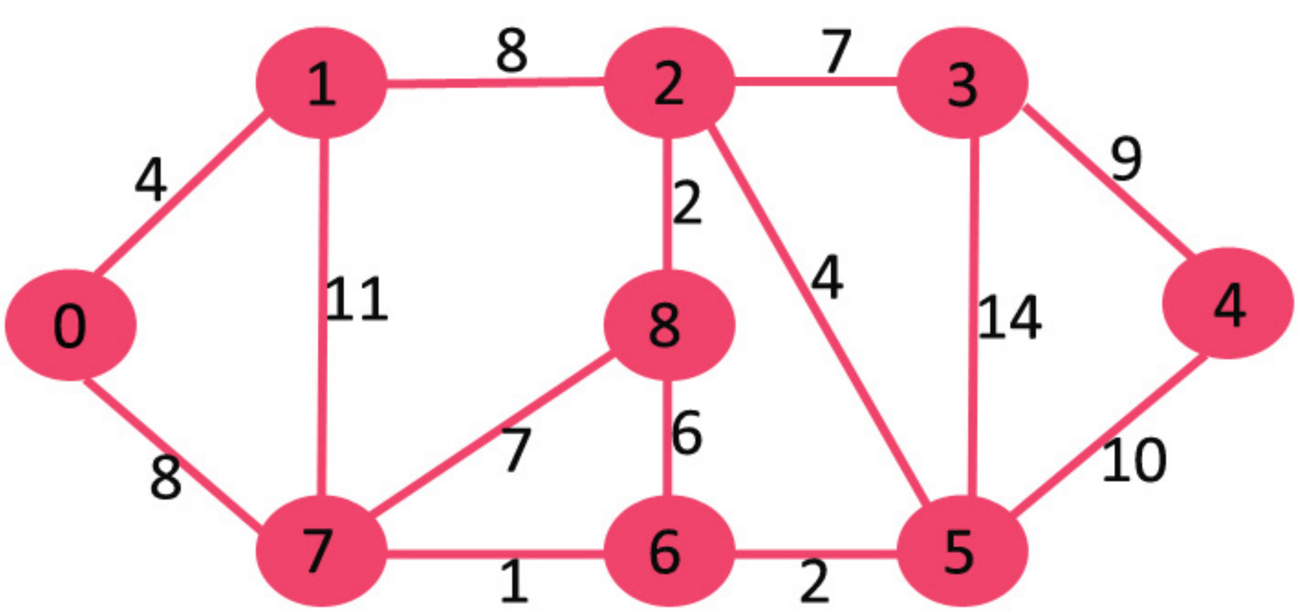
- Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent).
- Min Heap contains all vertices except vertex 0 and 1.

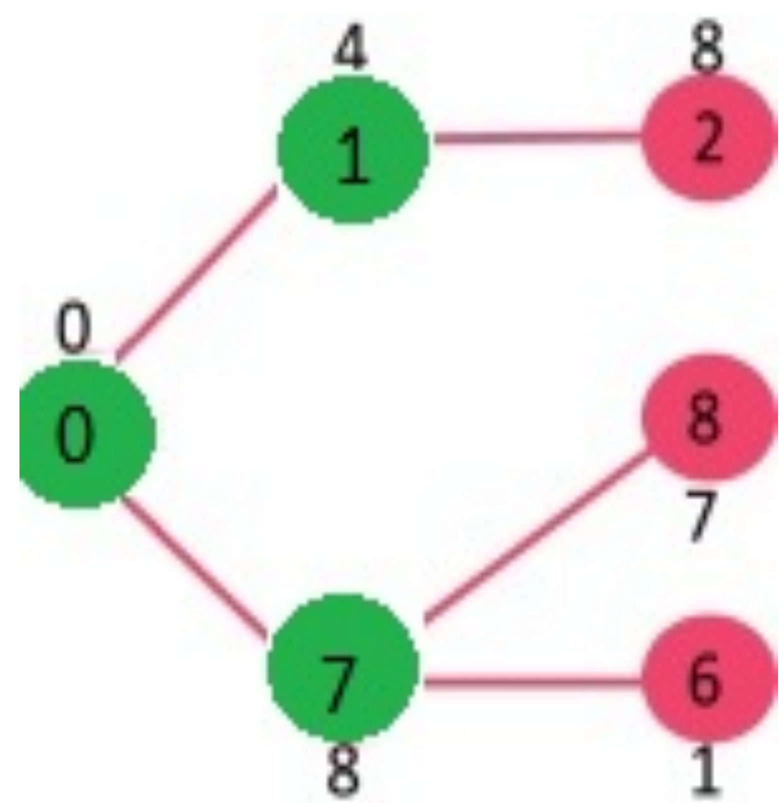




Example

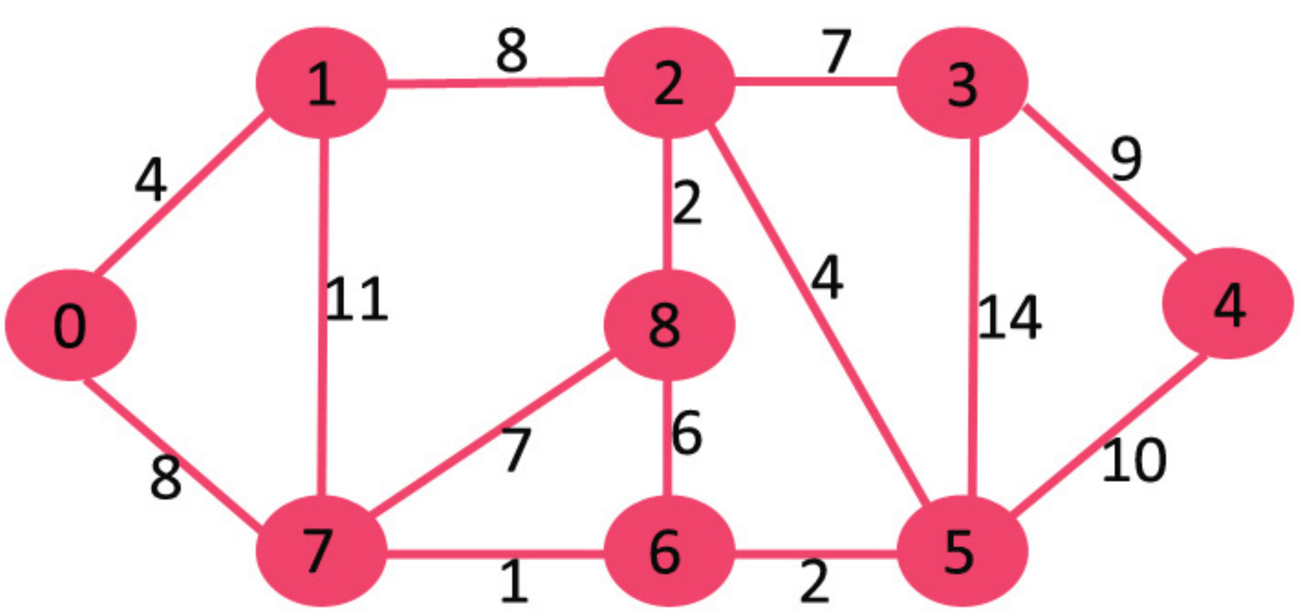
- Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent).
- Min Heap contains all vertices except vertex 0, 1 and 7.

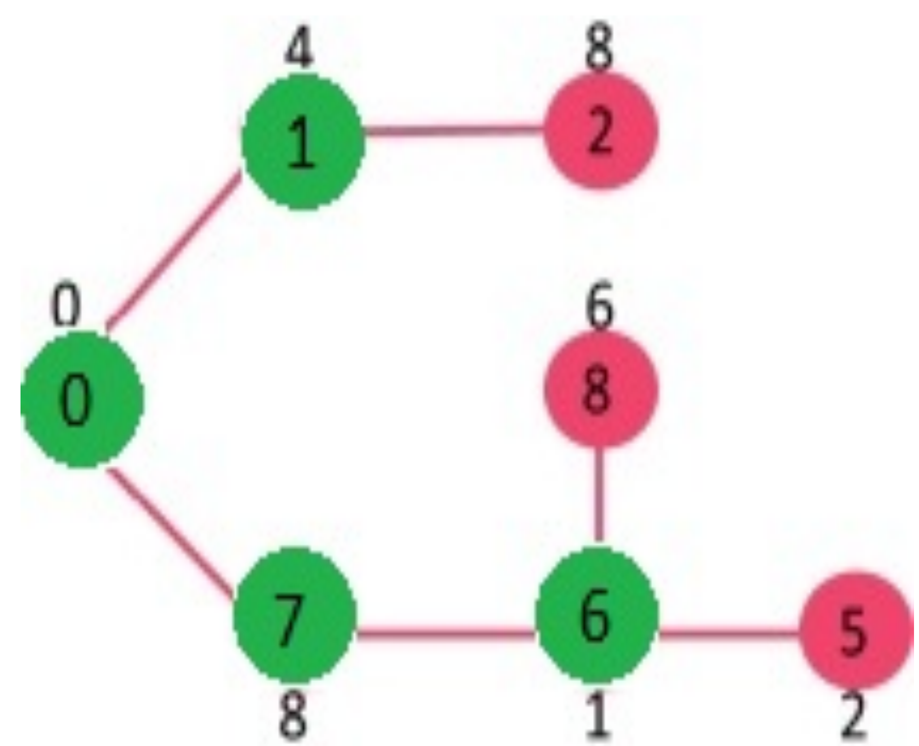




Example

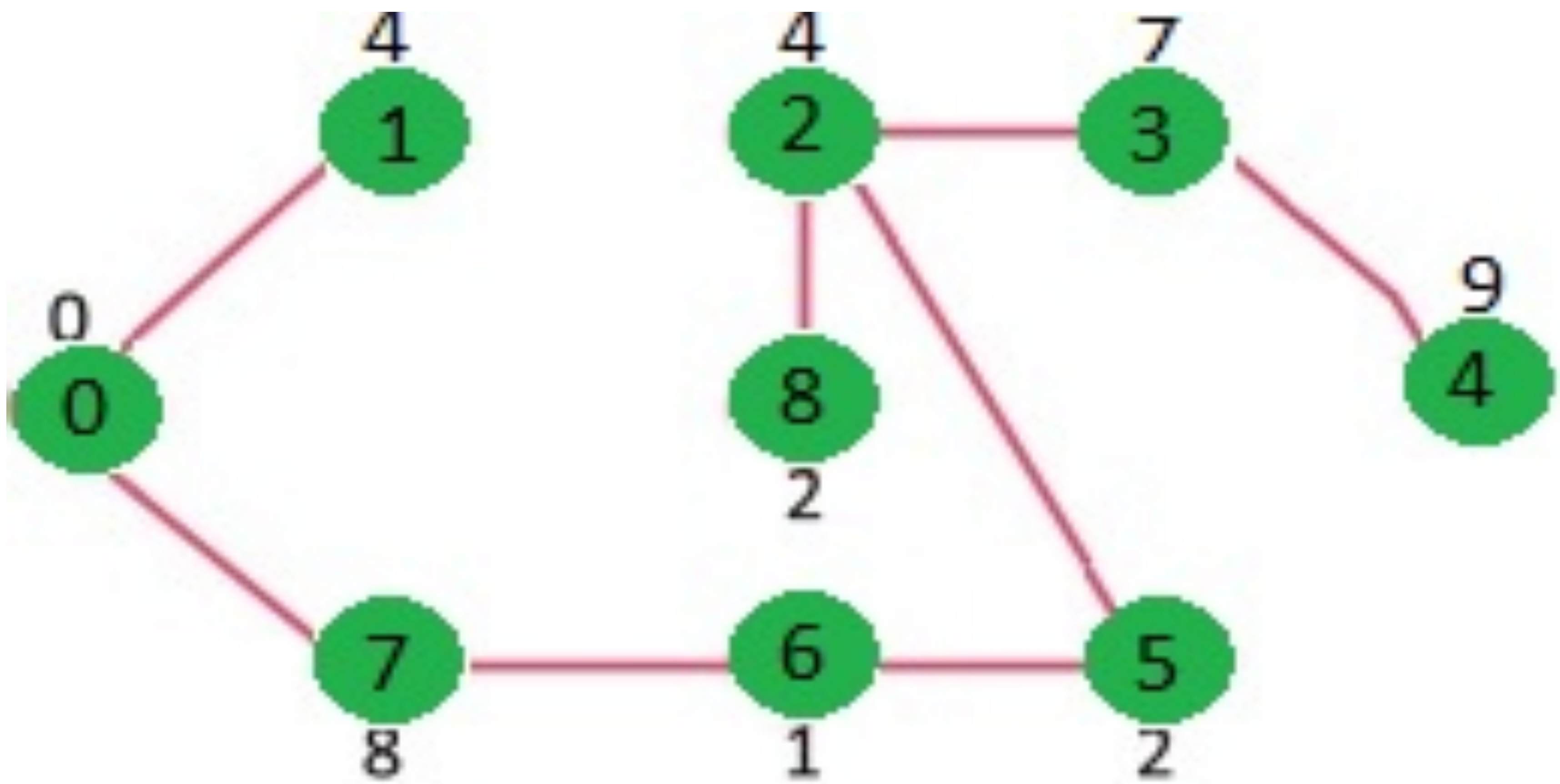
- Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a vertex is in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent).
- Min Heap contains all vertices except vertex 0, 1, 7 and 6.





Example

- The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty



How to implement the above algorithm?

- **Prim's Algorithm** is to traverse all vertices of graph using BFS and use a Min Heap to store the vertices not yet included in MST.
- Min Heap is used as a **priority queue** to get the minimum weight edge from the cut.
- Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is O(log |V|) in Min Heap.

Prim's Algorithm Time Complexity

- Worst case time complexity of Prim's Algorithm is
  - $O(|E| \log |V|)$ using binary heap
  - $O(|E| + |V| \log |V|)$ using Fibonacci heap

Prim's Algorithm Time Complexity

- If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in $O(|V| + |E|)$ time.
- We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.
- To get the minimum weight edge, we use min heap as a priority queue.
- Min heap operations like extracting minimum element and decreasing key value takes $O(\log |V|)$ time.
- So, overall time complexity is $O(|E| + |V|) \text{ x } O(\log |V|) = O((|E| + |V|) \log |V|) = O(|E| \log |V|)$

Prim's Algorithm Time Complexity

- Worst case time complexity of Prim's Algorithm is
  - $O(|E| \log |V|)$ using binary heap

- This time complexity can be improved and reduced to $O(|E| + |V| \log |V|)$ using Fibonacci heap.

Prim’s Algorithm

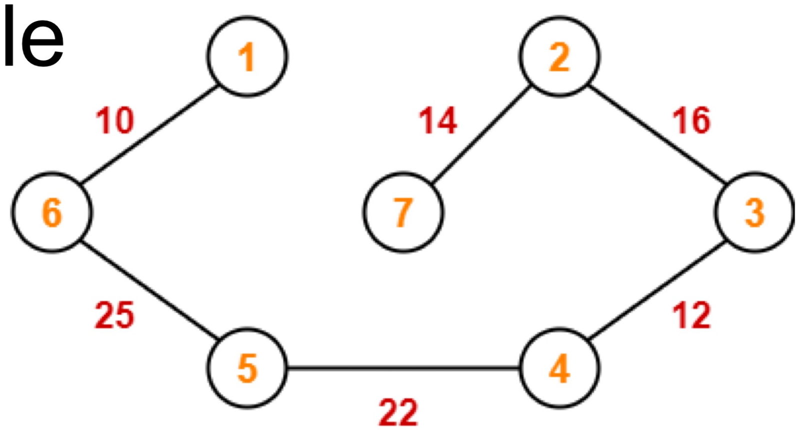
```
// Prim's Algorithm in Java
import java.util.Arrays;
class PGraph {
	public void Prim(int G[][], int V) {
		int INF = 9999999;
		int no_edge; // number of edge
		// create a array to track selected vertex
		// selected will become true otherwise false
		boolean[] selected = new boolean[V];
		Arrays.fill(selected, false);  // set selected false initially
		no_edge = 0;  // set number of edge to 0
		// the number of egde in minimum spanning tree will be
		// always less than (|V|-1)
		selected[0] = true;       // choose 0th vertex and make it true
		System.out.println("Edge : Weight"); // print for edge and weight
		while (no_edge < V - 1) {
			// next page
		}
	}
```

```
while (no_edge < V - 1) {
// For every vertex in the set S, find the all adjacent vertices
// , calculate the distance from the vertex selected at step 1.
// if the vertex is already in the set S, discard it otherwise
// choose another vertex nearest to selected vertex at step 1.
    int min = INF;
    int x = 0; // row number
    int y = 0; // col number
    for (int i = 0; i < V; i++) {
        if (selected[i] == true) {
            for (int j = 0; j < V; j++) {
                // not in selected and there is an edge
                if (!selected[j] && G[i][j] != 0) {
                    if (min > G[i][j]) { min = G[i][j]; x = i; y = j; }
                }
            }
        }
    }
    System.out.println(x + " - " + y + " : " + G[x][y]);
    selected[y] = true; no_edge++;
}
```

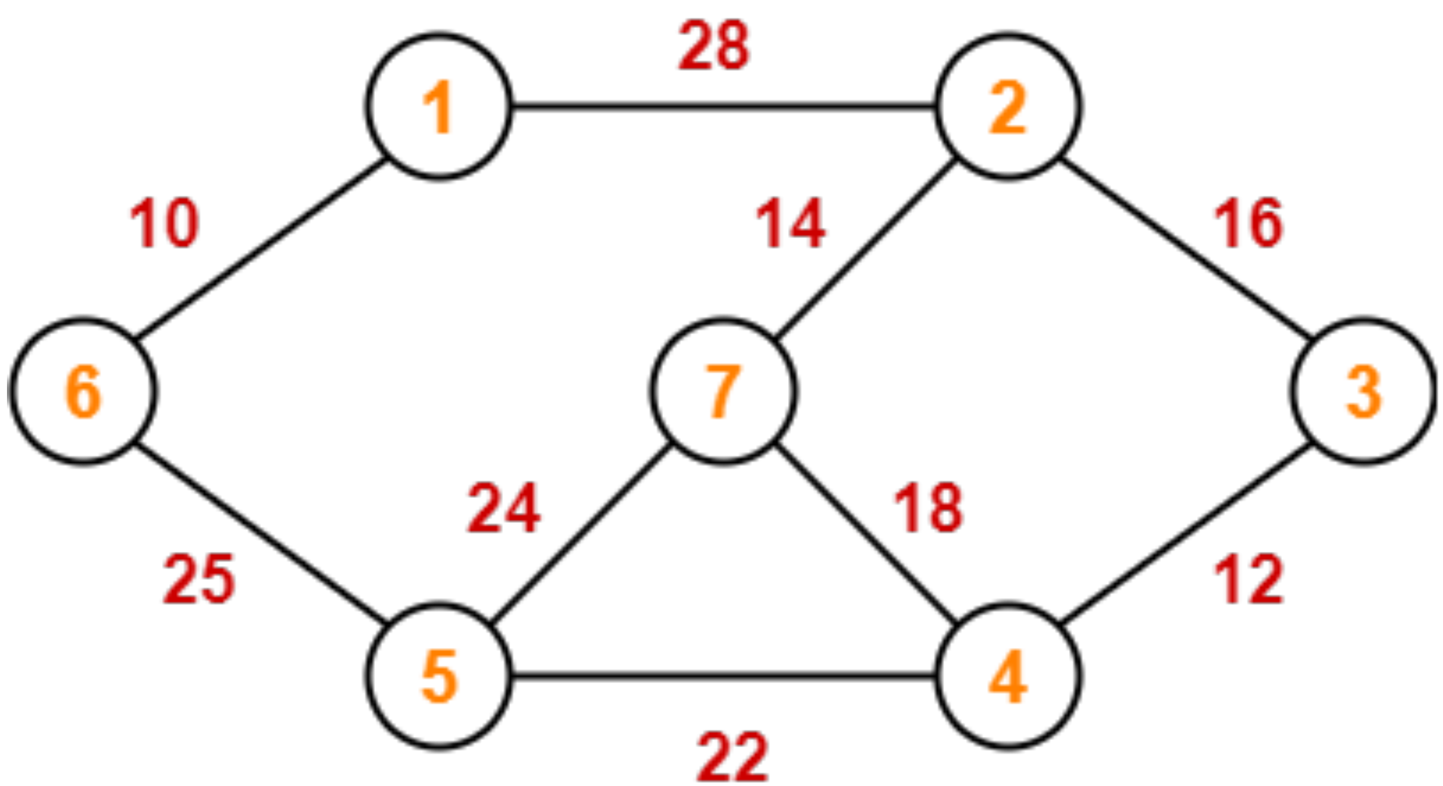
Prim’s Algorithm

```
import java.util.Arrays;
class PGraph {
	public void Prim(int G[][], int V) {
		//…
	}

	public static void main(String[] args) {
		PGraph g = new PGraph();

		// number of vertices in graph
		int V = 5;

		// create a 2d array of size 5x5
		// for adjacency matrix to represent graph
		int[][] G = { { 0, 9, 75, 0, 0 }, { 9, 0, 95, 19, 42 }, { 75, 95, 0, 51, 66 },
			{ 0, 19, 51, 0, 31 }, { 0, 42, 66, 31, 0 } };
		g.Prim(G, V);
	}
}
```

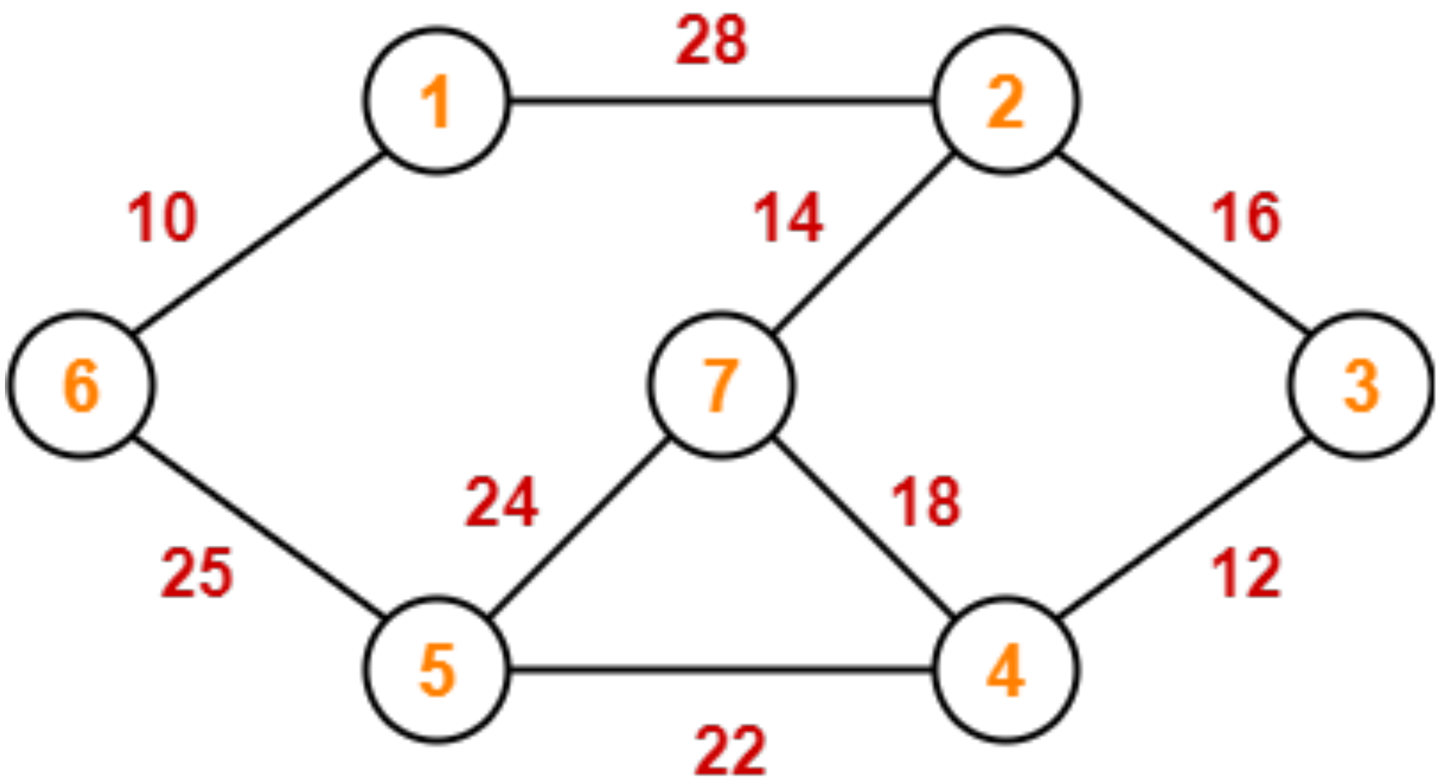
Example



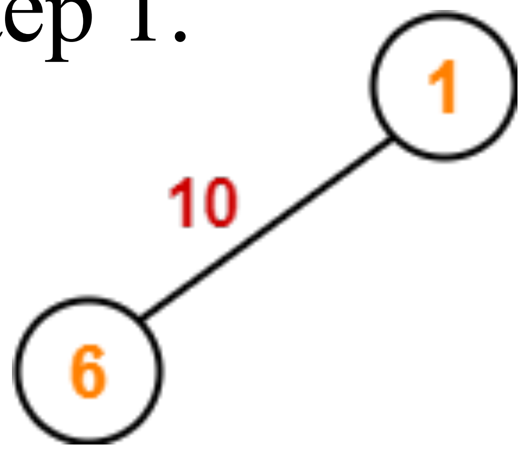
- Construct ... ng tree (MST) for the given graph usi...



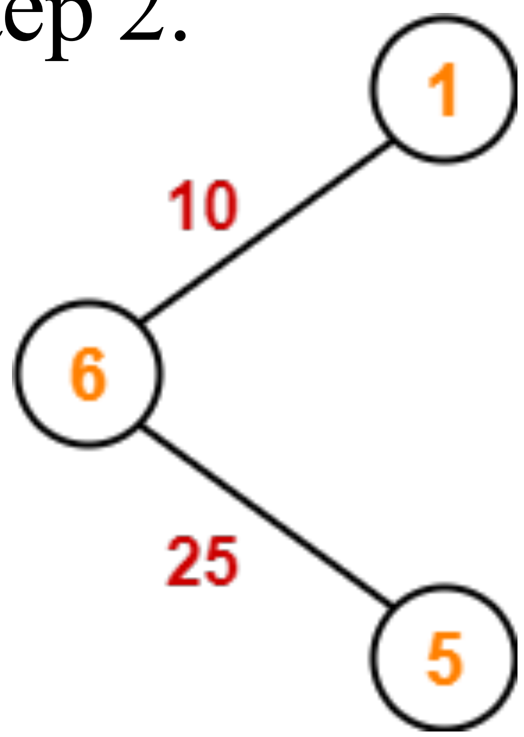
Example



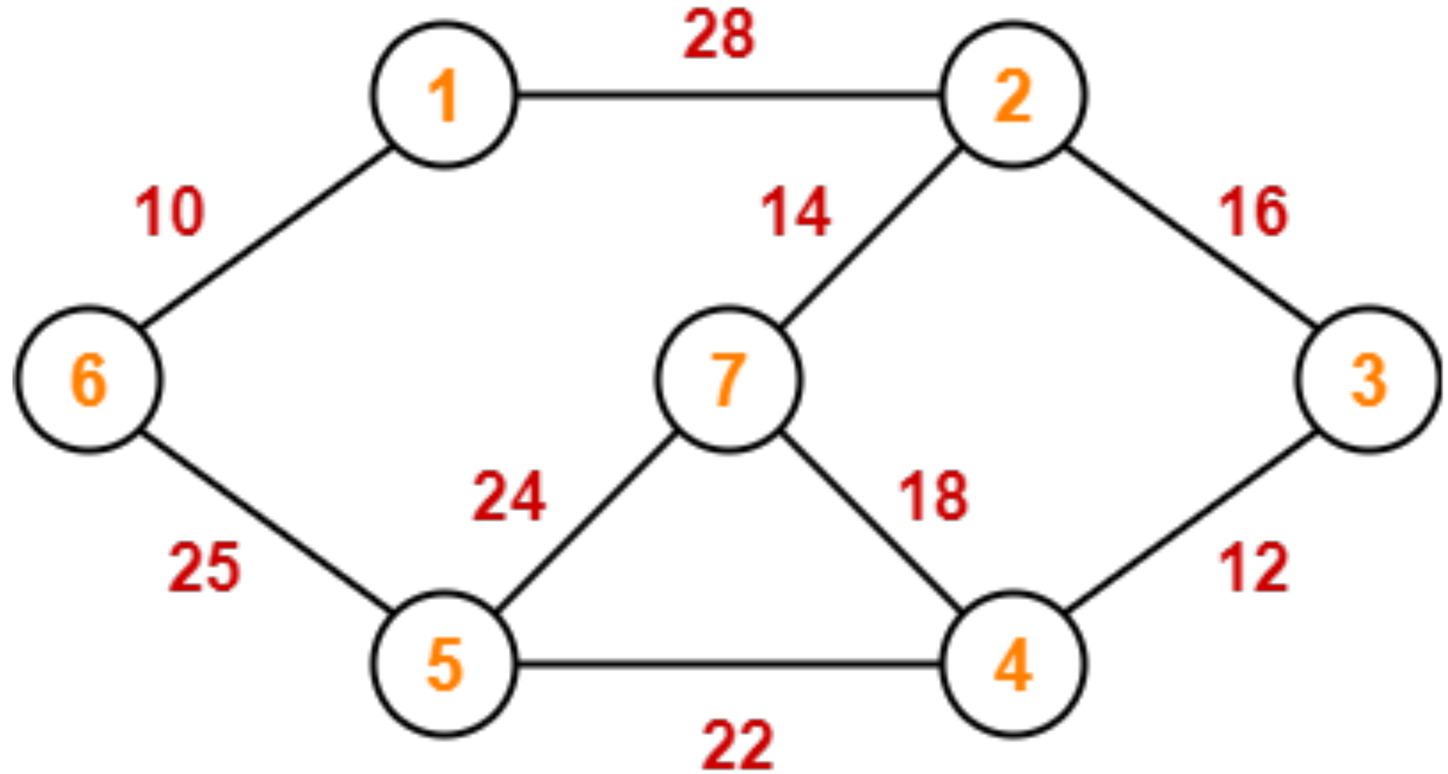
- Step 1.



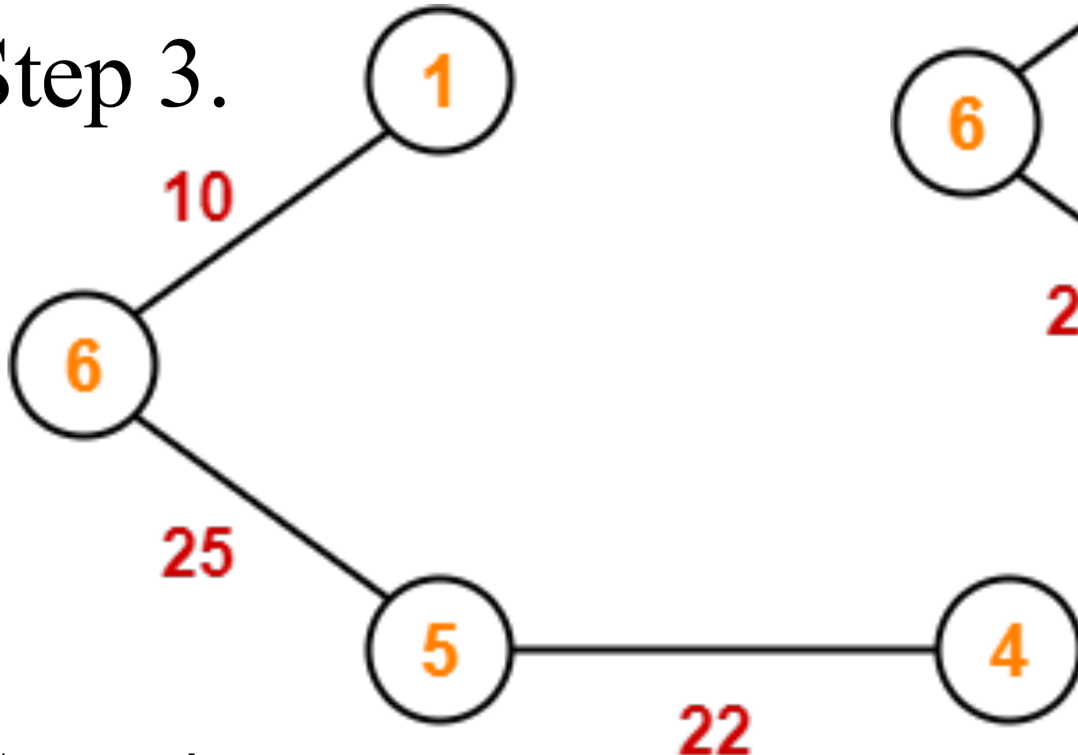
- Step 2.



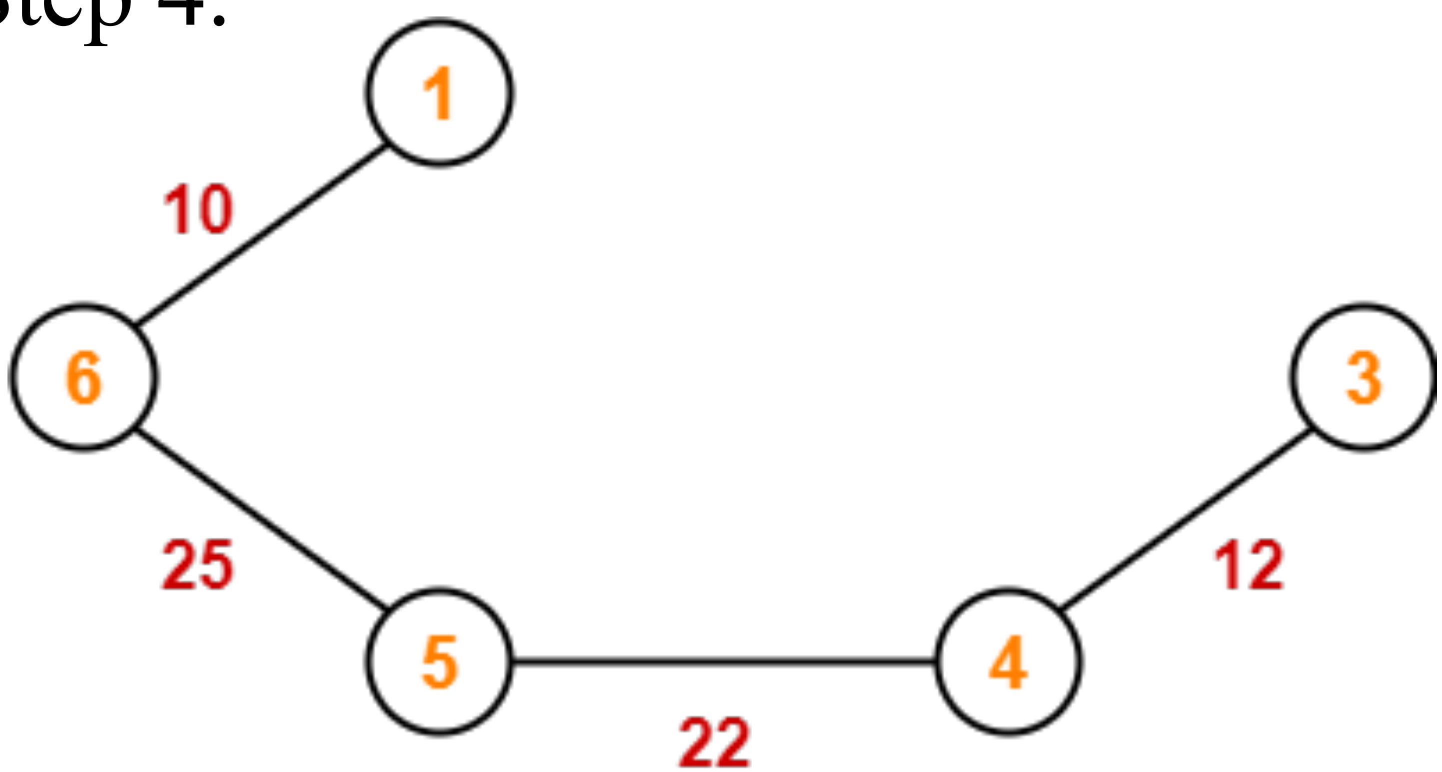
Example



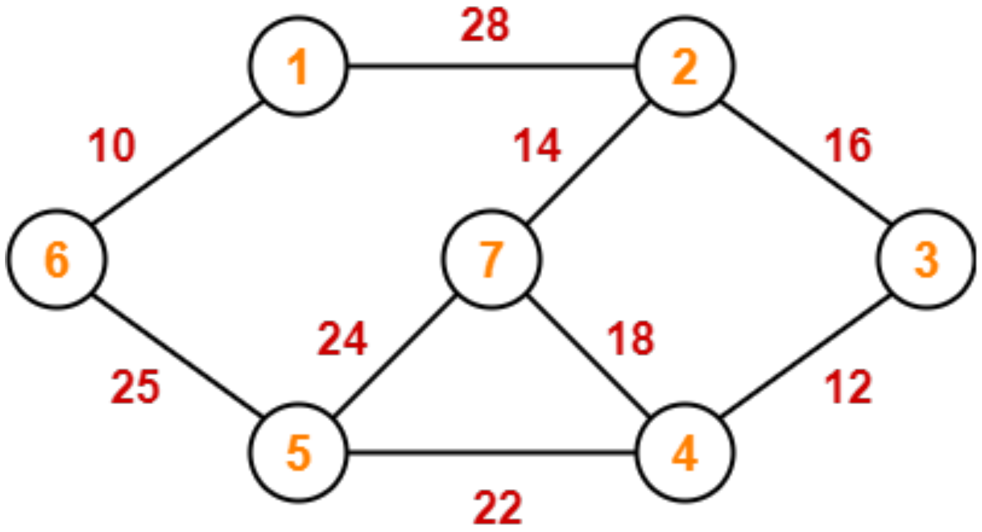
- Step 3.



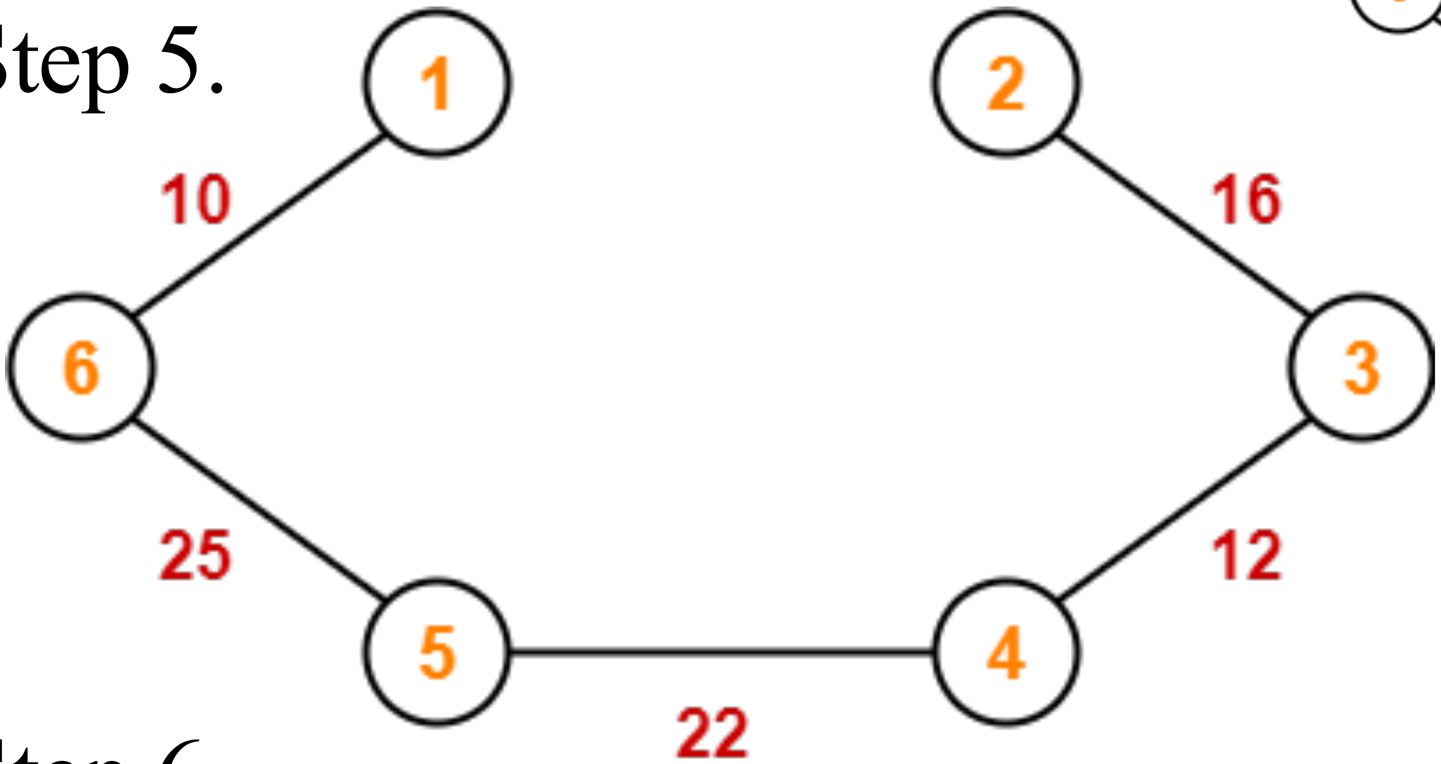
- Step 4.



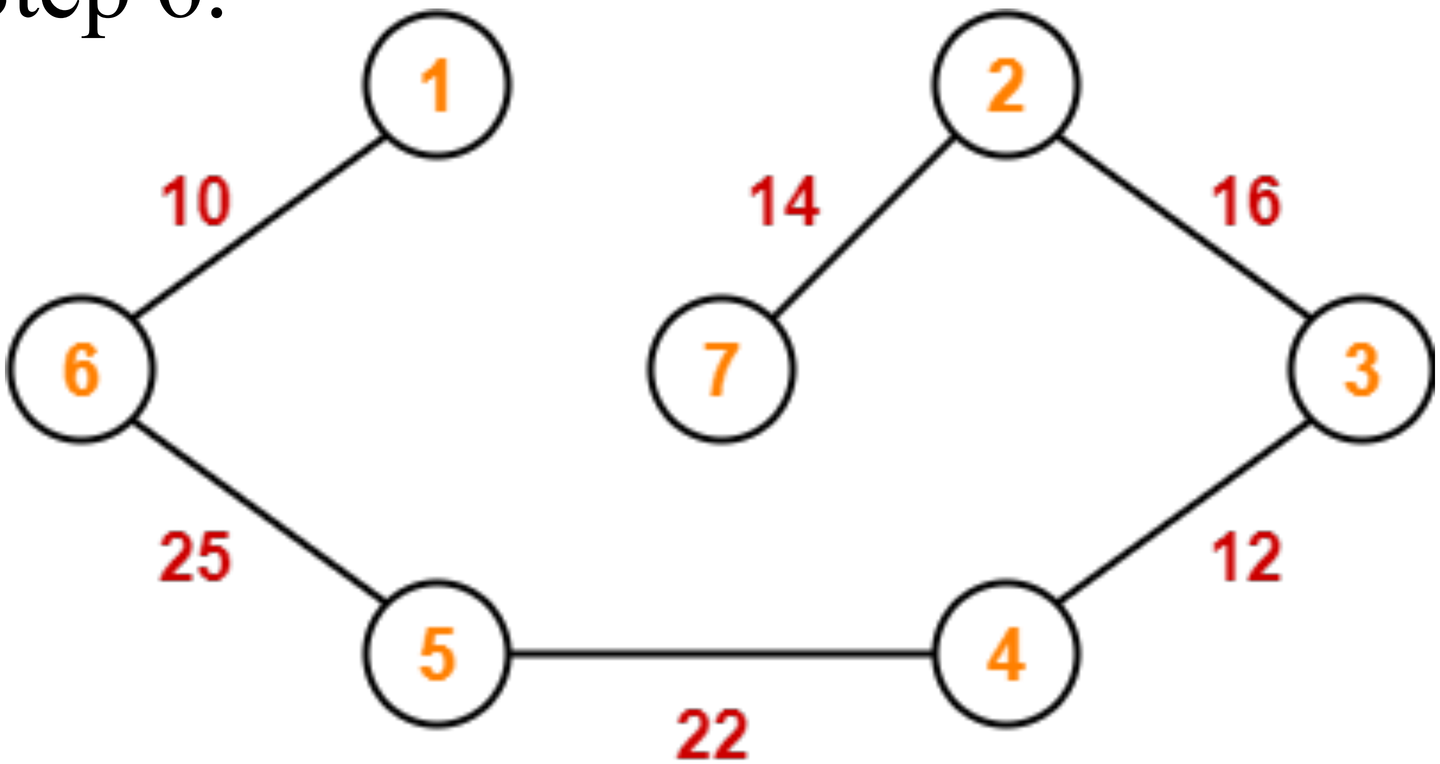
Example



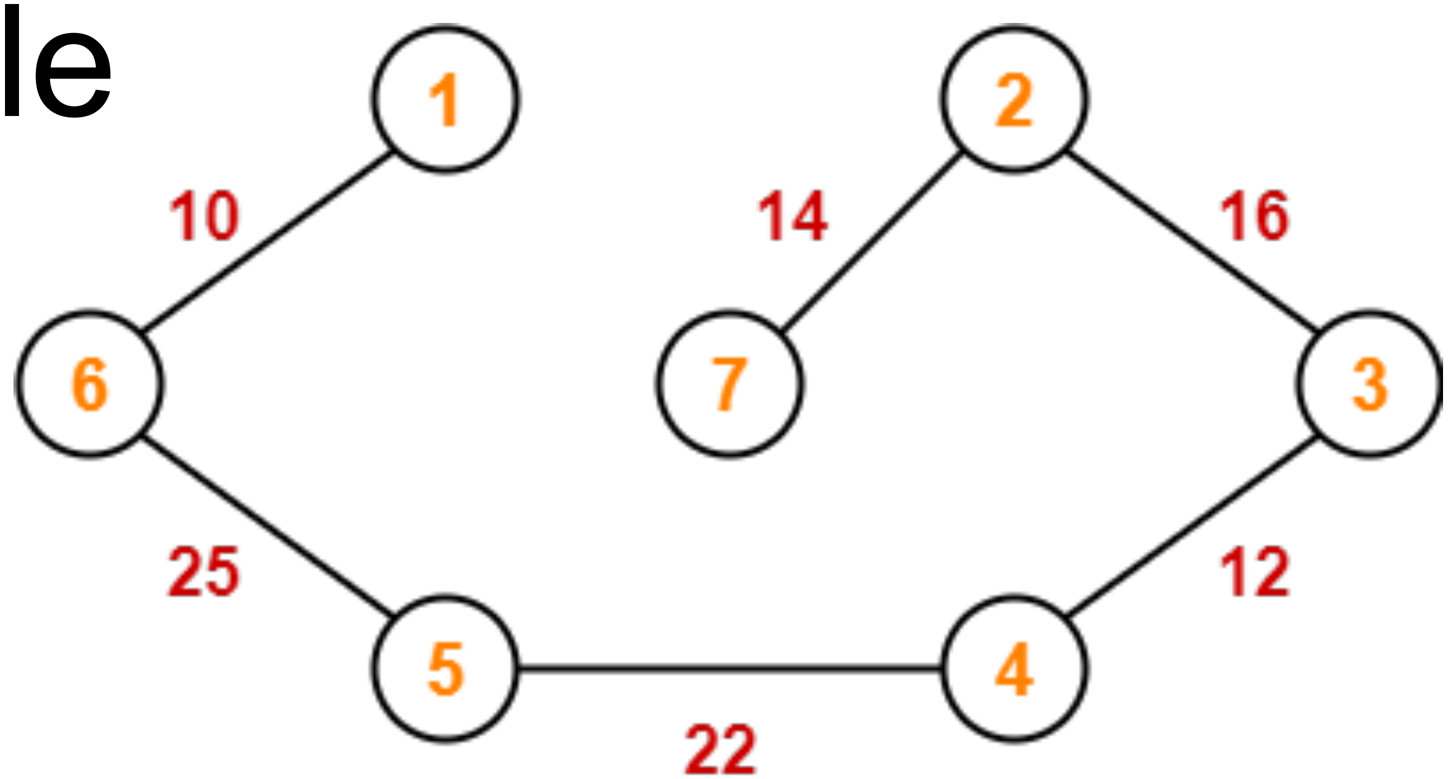
- Step 5.



- Step 6.



Example

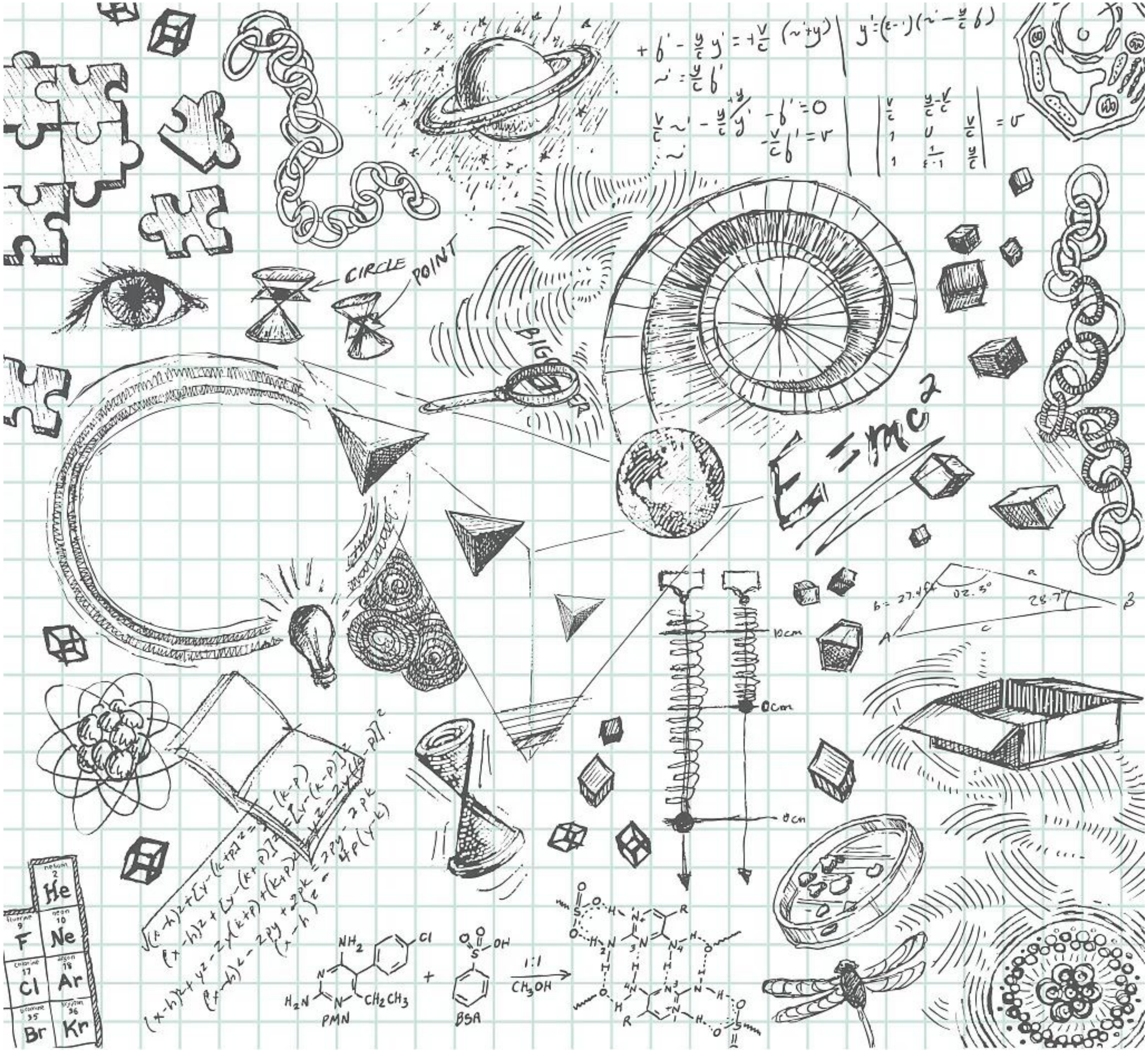


- Since all the vertices have been included in the MST, so we stop.
- Now, Cost of this Minimum Spanning Tree is

  Sum of all edge weights

  $= 10 + 25 + 22 + 12 + 16 + 14$

  $= 99$ units


CIRCLE
POINT
E=mc²
10cm
0cm
He
F
Ne
Cl
Ar
Br
Kr
PMN
BSA
1:1
CH₃OH