- A **greedy algorithm** is a problem-solving technique that makes the best local choice at each step in the hope of finding the global optimum solution.

- It prioritizes immediate benefits over long-term consequences, making decisions based on the current situation without considering future implications.

- While this approach can be efficient and straightforward, it doesn't guarantee the best overall outcome for all problems.

not all problems are suitable for greedy algorithms. They work best when the problem exhibits the following properties:

- Greedy Choice Property: The optimal solution can be constructed by making the best local choice at each step.

- Optimal Substructure: The optimal solution to the problem contains the optimal solutions to its subproblems.

Characteristics of Greedy Algorithm

- Greedy algorithms are simple and easy to implement.
- They are efficient regarding time complexity, often providing quick solutions.
- Greedy algorithms are used for optimization problems where a **locally optimal** choice leads to a **globally optimal** solution.
- These algorithms do not reconsider previous choices, as they make decisions based on current information without looking ahead.
- Greedy algorithms are suitable for problems for optimal substructure.

Examples of Greedy Algorithm
few examples:

- **Dijkstra's Algorithm:** This algorithm finds the shortest path between two nodes in a graph. It works by repeatedly choosing the shortest edge available from the current node.

- **Kruskal's Algorithm:** This algorithm finds the minimum spanning tree of a graph. It works by repeatedly choosing the edge with the minimum weight that does not create a cycle.

- **Fractional Knapsack Problem:** This problem involves selecting items with the highest value-to-weight ratio to fill a knapsack with a limited capacity. The greedy algorithm selects items in decreasing order of their value-to-weight ratio until the knapsack is full.

- **Scheduling and Resource Allocation** : The greedy algorithm can be used to schedule jobs or allocate resources in an efficient manner.

- **Coin Change Problem** : The greedy algorithm can be used to make change for a given amount with the minimum number of coins, by always choosing the coin with the highest value that is less than the remaining amount to be changed.

- **Huffman Coding** : The greedy algorithm can be used to generate a prefix-free code for data compression, by constructing a binary tree in a way that the frequency of each character is taken into consideration

Dijkstra's Algorithm

- It is used for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e.,
- it is to find the shortest distance between two vertices on a graph.

1. The algorithm maintains a set of visited vertices and a set of unvisited vertices.
2. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source.
3. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found.
4. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

Applications:

- The need for Dijkstra's algorithm arises in many applications where finding the shortest path between two points is important.
- It can be used in the routing protocols for computer networks .
- It can also be used by map systems to find the shortest path between starting point and the Destination.

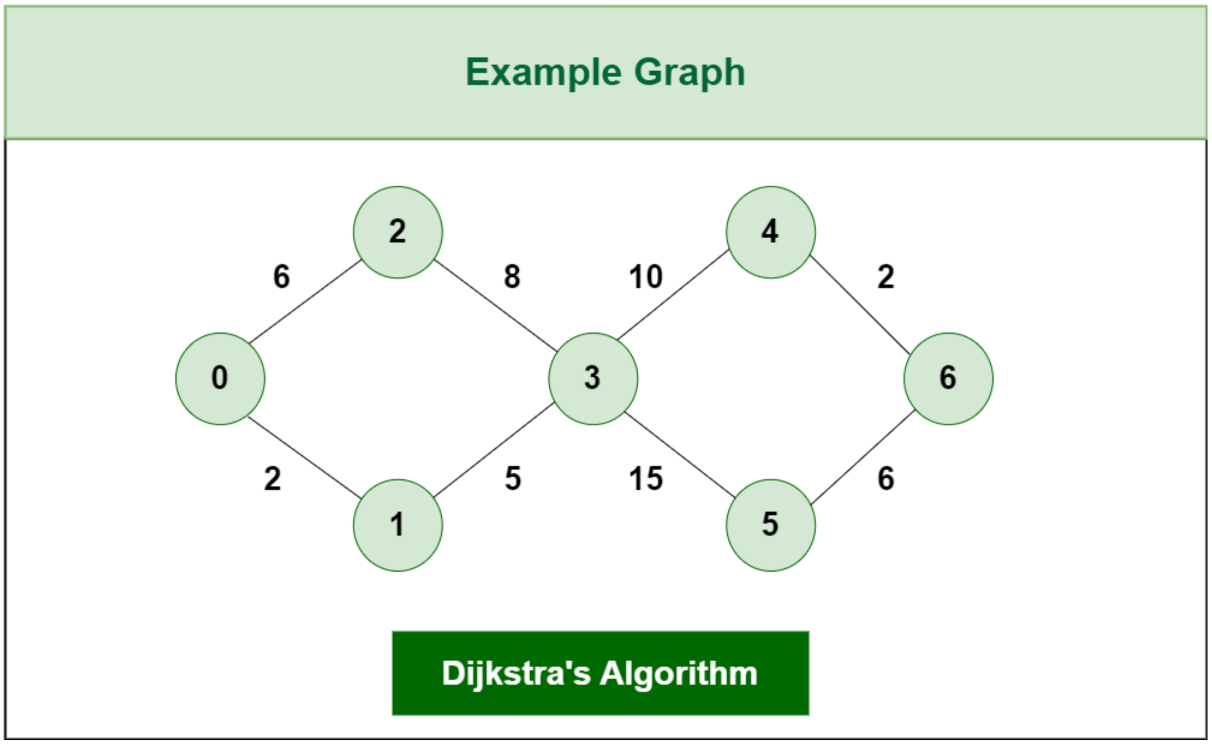
Directed and Undirected Graph

- In a directed graph, each edge has a direction, indicating the direction of travel between the vertices connected by the edge. In this case, the algorithm follows the direction of the edges when searching for the shortest path.
- In an undirected graph, the edges have no direction, and the algorithm can traverse both forward and backward along the edges when searching for the shortest path.
- Dijkstra's algorithm can work on both directed graphs and undirected graphs.

Algorithm for Dijkstra's Algorithm:

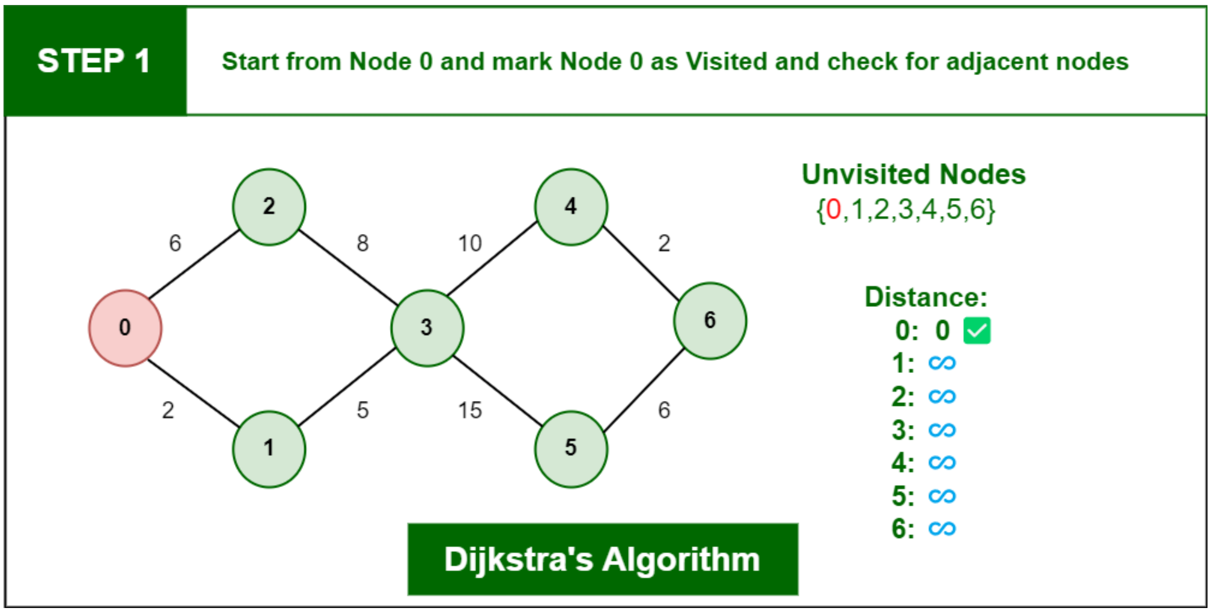
1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.

3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

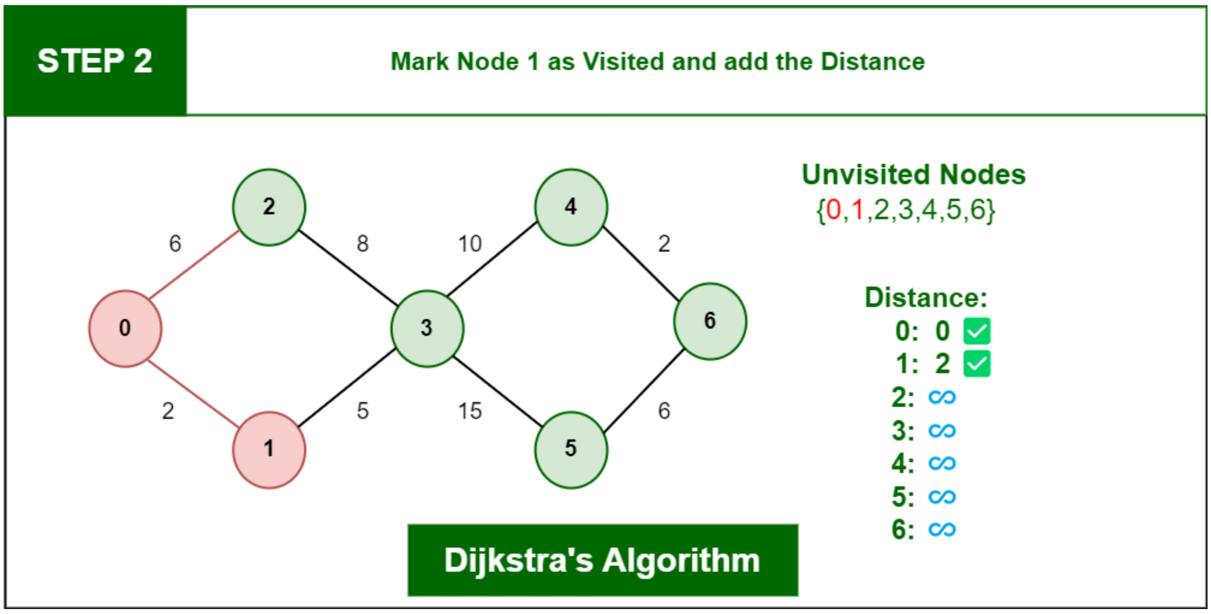


Example: 0 -> 0, 1-> ∞,2-> ∞,3-> ∞,4-> ∞,5-> ∞,6-> ∞.

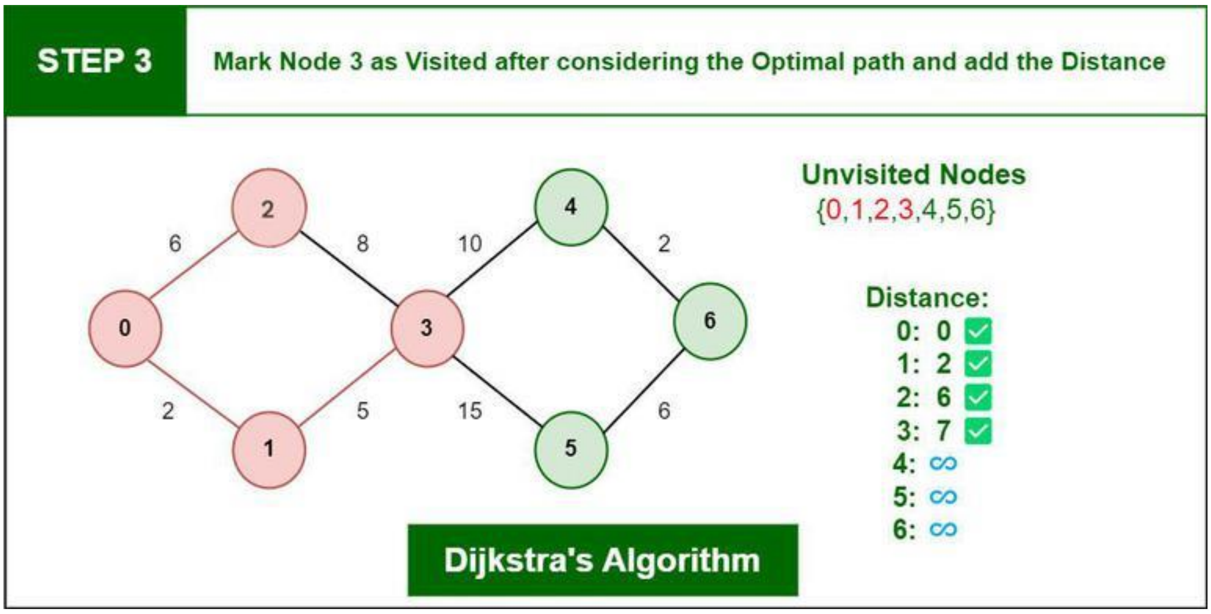
Step 1: Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.



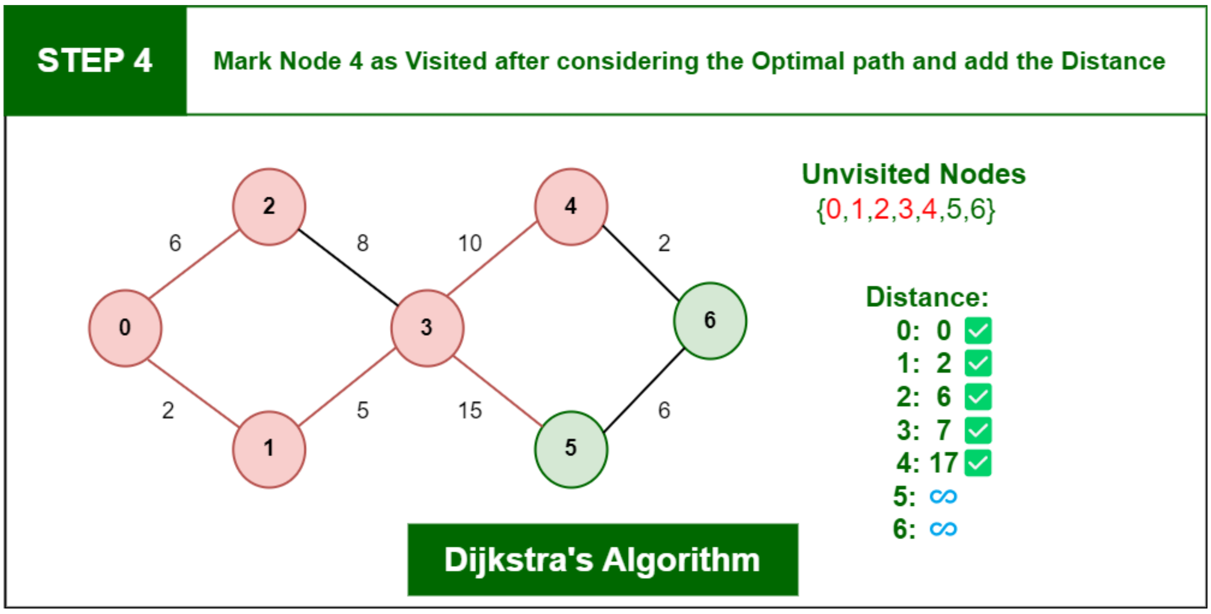
step 2: Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance. In this step **Node 1** is Minimum distance adjacent Node, so marked it as visited and add up the distance.



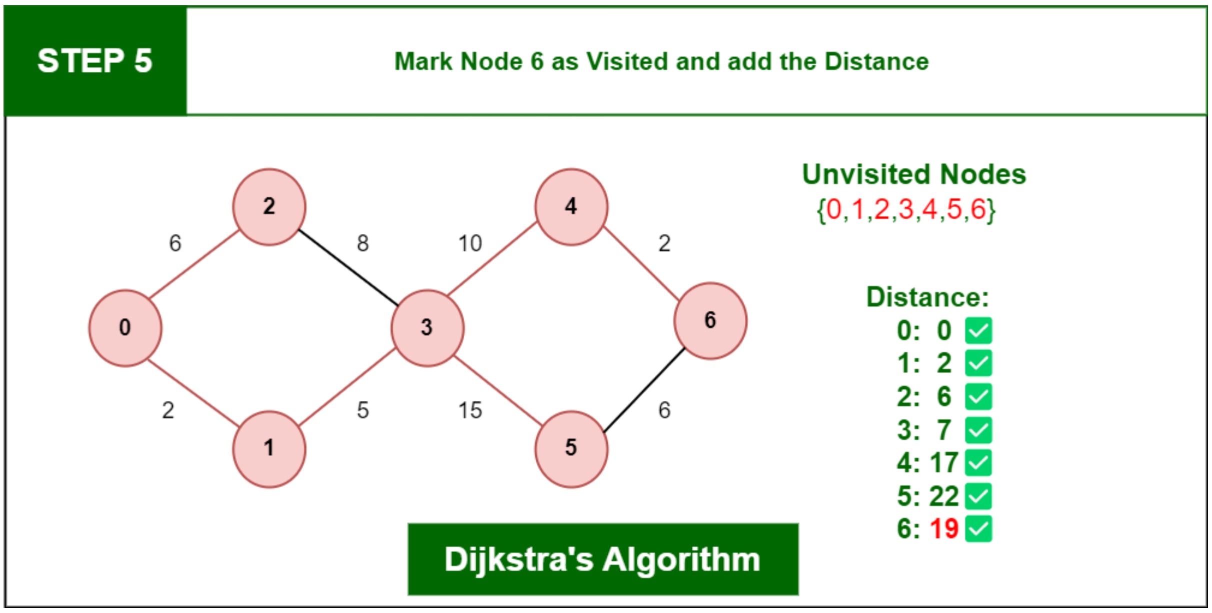
Step 3: Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:



Step 4: Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step Node 4 is Minimum distance adjacent Node, so marked it as visited and add up the distance.
Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 = 2 + 5 + 10 = 17



Step 5: Again, Move Forward and check for adjacent Node which is Node 6, so marked it as visited and add up the distance, Now the distance will be:
Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19



Minimum Spanning Tree:

- A **minimum spanning tree (MST)** is defined as a **spanning tree** that has the minimum weight among all the possible spanning trees.

- A **spanning tree** is defined as a tree-like subgraph of a connected, undirected graph that includes all the vertices of the graph.
- it is a subset of the edges of the graph that forms a tree (**acyclic**) where every node of the graph is a part of the tree.

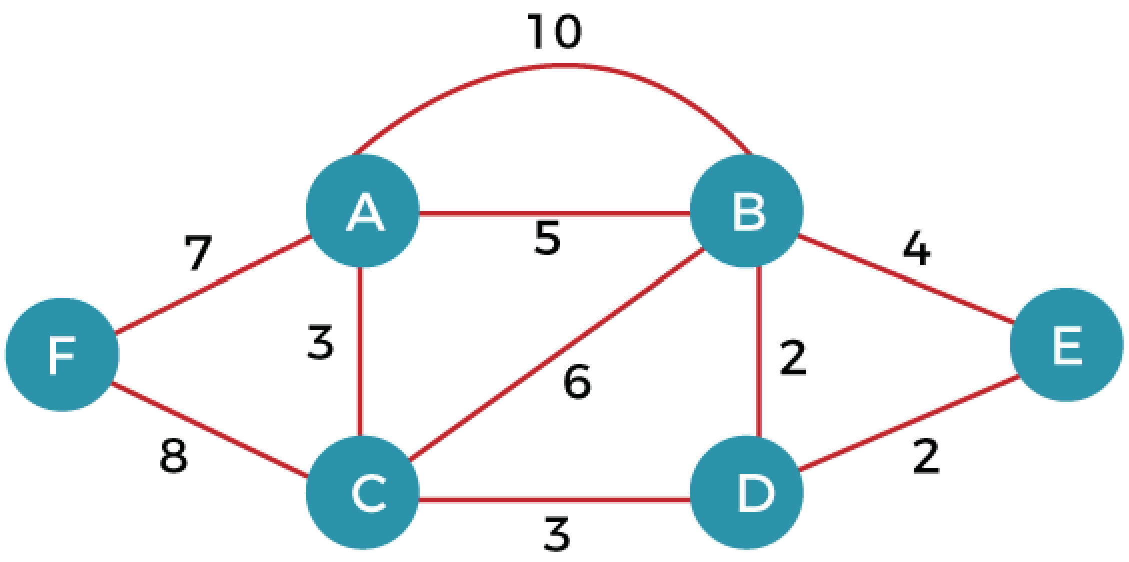
The minimum spanning trees are mainly of two types:

- Prim's MST
- Kruskal's MST

The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.

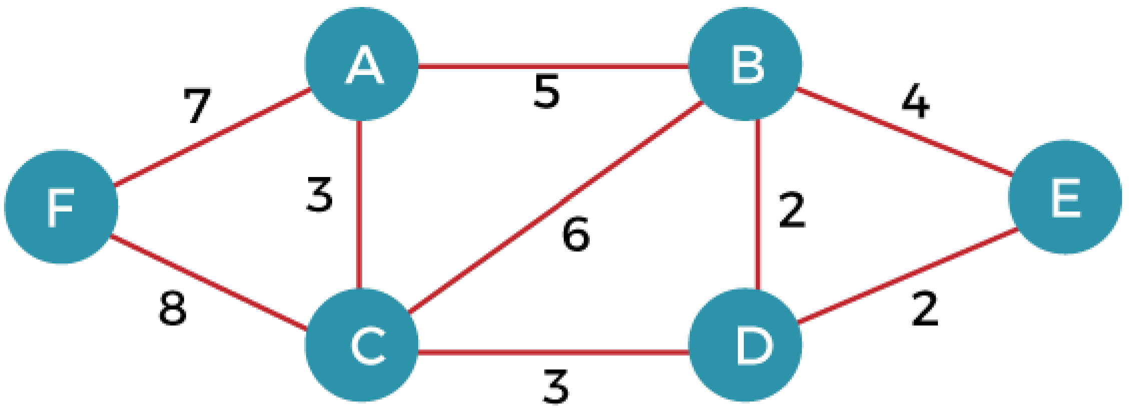
Properties of a Spanning Tree:

- The number of vertices (**V**) in the graph and the spanning tree is the same.
- There is a fixed number of edges in the spanning tree which is equal to one less than the total number of vertices (**E** = **V-1**).
- The spanning tree should not be **disconnected**, as in there should only be a single source of component, not more than that.
- The spanning tree should be **acyclic,** which means there would not be any cycle in the tree.
- The total cost (or weight) of the spanning tree is defined as the sum of the edge weights of all the edges of the spanning tree.
- There can be many possible spanning trees for a graph.
- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum $\mathbf{n^{n-2}}$ number of spanning trees that can be created from a **connected graph**.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.



to calculate the minimum spanning tree of the above graph. To find the minimum cost spanning tree, the first step is to remove all the loops and the parallel edges from the graph. Here, parallel edges mean that there exists more than one edge between the two vertices. For example, in the above graph, there exists two edges between the vertices A and B having weights 5 and 10 respectively, so these edges are the parallel edges. We have to remove any of the edges.

In case of parallel edges, the edges with the highest weight will be removed.



Once we remove the parallel edge, we will arrange the edges according to the increasing order of their edge weights.

BD = 2

DE = 2

AC = 3

CD = 3

BE = 4

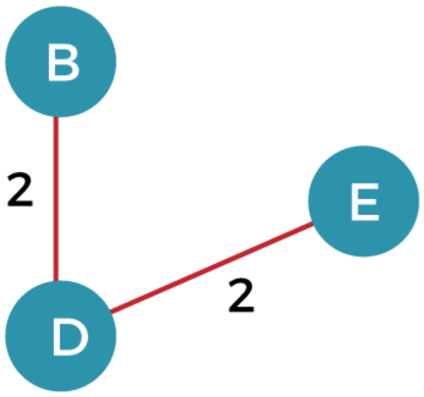
AB = 5

CB = 6

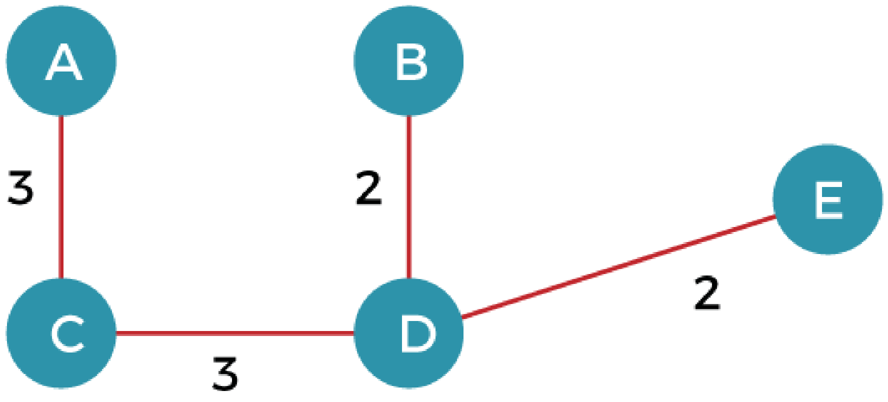
AF = 6

FC = 6

Once the edge weights are written in the increasing order, the third step is to connect the vertices according to their weights. The edge weight 2 is minimum, so we connect the vertices with a edge weight 2 as shown as below:



The next edge is AC having weight 3 shown as below:



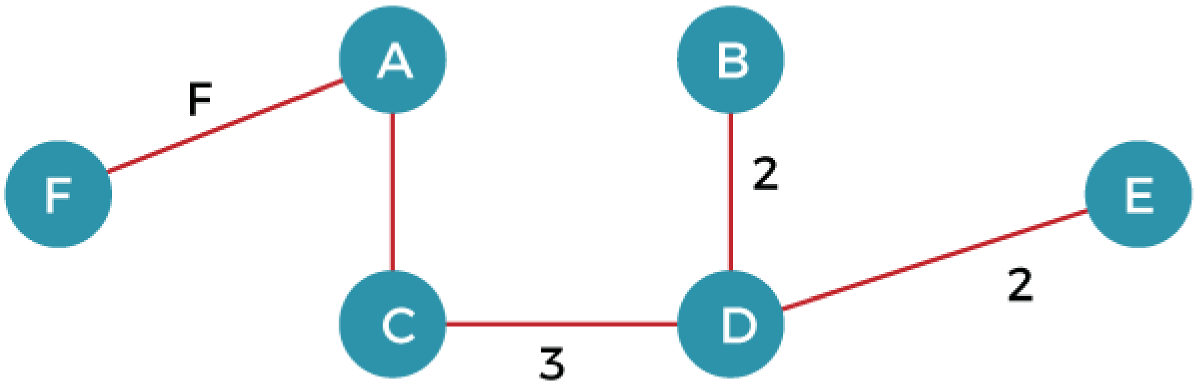
The next edge is CD. If we connect the vertices C and D then it does not form any cycle shown as below:

The next edge is BE. But we cannot connect the vertices B and E as it forms a cycle.

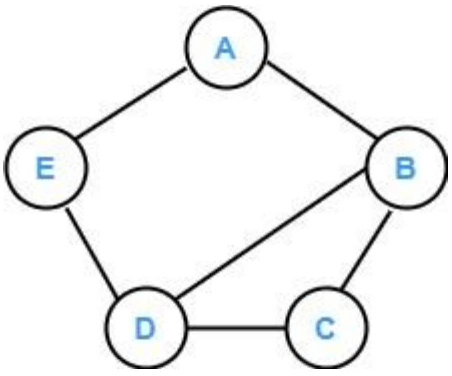
The next edge is AB. But we cannot connect the vertices A and B as it forms a cycle.

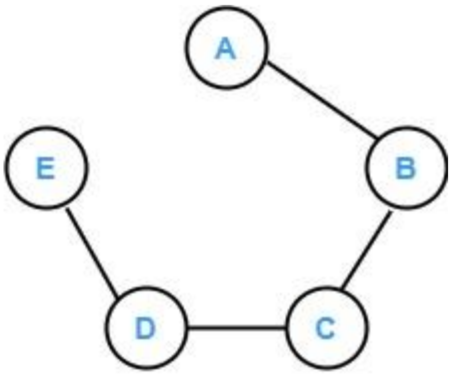
The next edge is BC. But we cannot connect the vertices B and C as it forms a cycle.

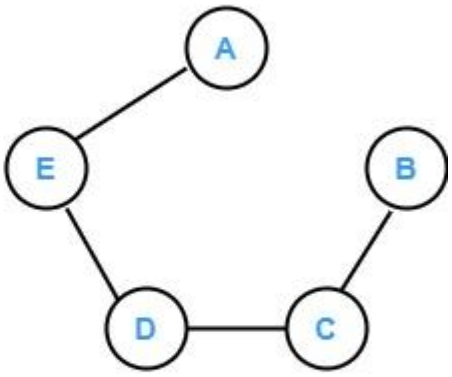
The next edge is AF. If we connect the vertices A and F then it does not form any cycle shown as below:

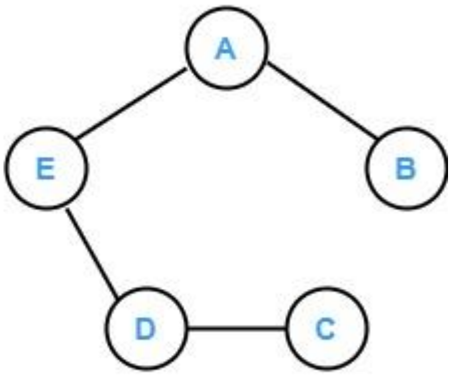


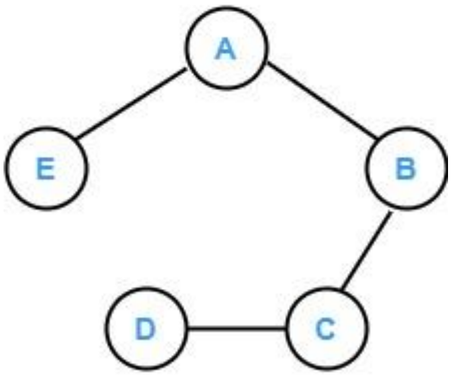
Consider the following original graph:




A
B
C
D
E


A
B
C
D
E


A
B
C
D
E


A
B
C
D
E

A

The following spanning trees can be possible from above graph Prim’s MST

Prim’s algorithm starts with a single node/vertex and moves on to the adjacent vertices to explore all the connected edges. The idea behind prim’s algorithm is that it maintains two sets of vertices. The first set comprises vertices that are already included in the minimum spanning tree and the other set consists of all the vertices which are not included yet.

Kruskal’s MST

Kruskal’s algorithm is a greedy algorithm used to find out the shortest path in a minimum-spanning tree. The algorithm aims to traverse the graph and detect the subset of edges with minimal value and cover all the vertices of the graph. At every step of the algorithm and analysis, it follows a greedy approach for an overall optimized result.
Kruskal’s algorithm can be summed up as a minimum spanning tree algorithm taking a graph as input and forms a subset of the edges of the graph,

- which has a minimum sum of edge weight among all the trees that can be formed from that graph.
- that form a tree including each vertex of the graph without forming any cycle between the vertex.

Applications of Spanning Tree:

- In routing protocols
- Cluster mapping and analysis
- Network Planning
- Explore the path/ route in the maps.

Prim’s Algorithm

Like Kruskal’s algorithm, Prim’s algorithm is also a Greedy Algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

1.The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices.

2.The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included.

3.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.

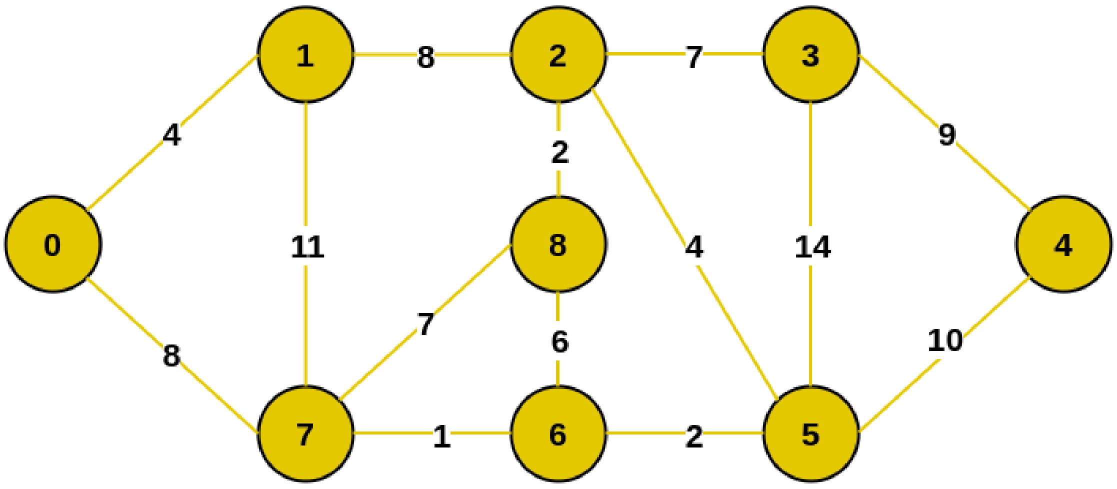
A group of edges that connects two sets of vertices in a graph is called cut in graph theory. So, at every step of Prim's algorithm, find a cut, pick the minimum weight edge from the cut, and include this vertex in MST Set (the set that contains already included vertices).

The working of Prim's algorithm can be described by using the following steps:

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
Step 3: Find edges connecting any tree vertex with the fringe vertices.
Step4: Find the minimum among these edges.
Step 5: Add the chosen edge to the MST if it does not form any cycle.
Step 6: *Return the MST and exit*

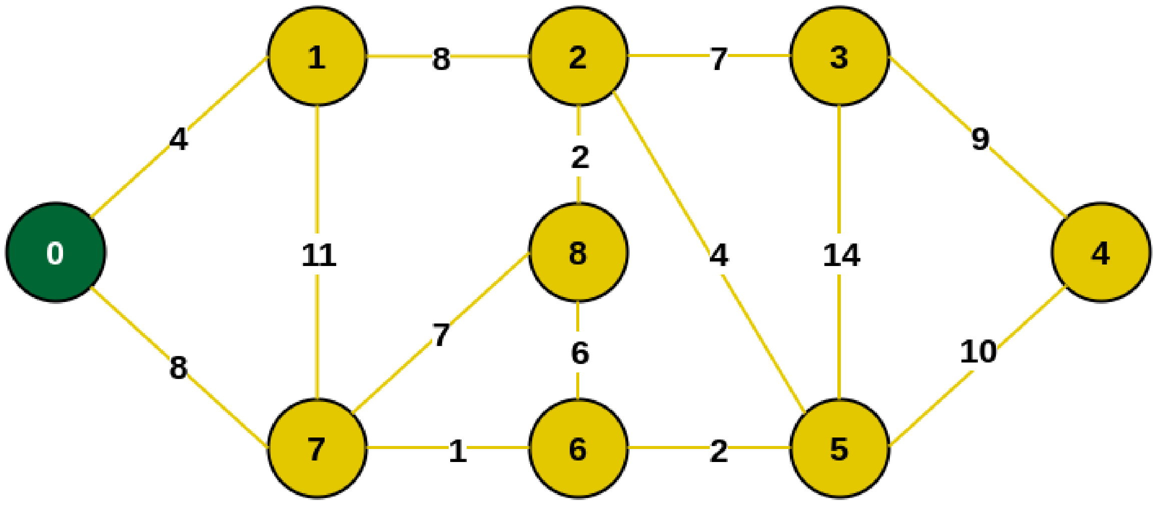
Note: For determining a cycle, we can divide the vertices into two sets [one set contains the vertices included in MST and the other contains the fringe vertices.

Illustration of Prim's Algorithm:



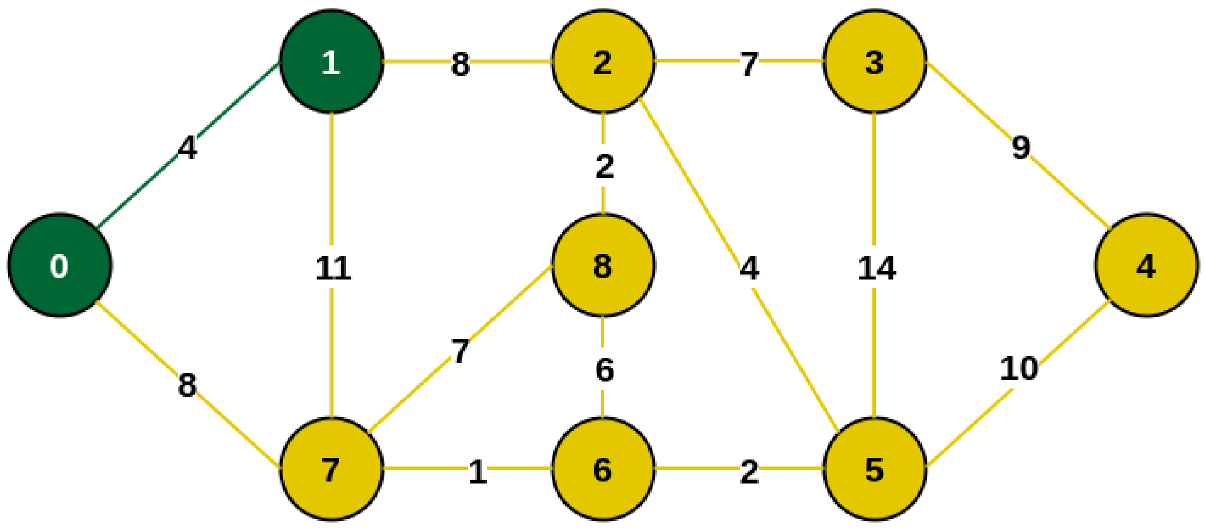
Example of a Graph

<u>Step 1:</u> 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.



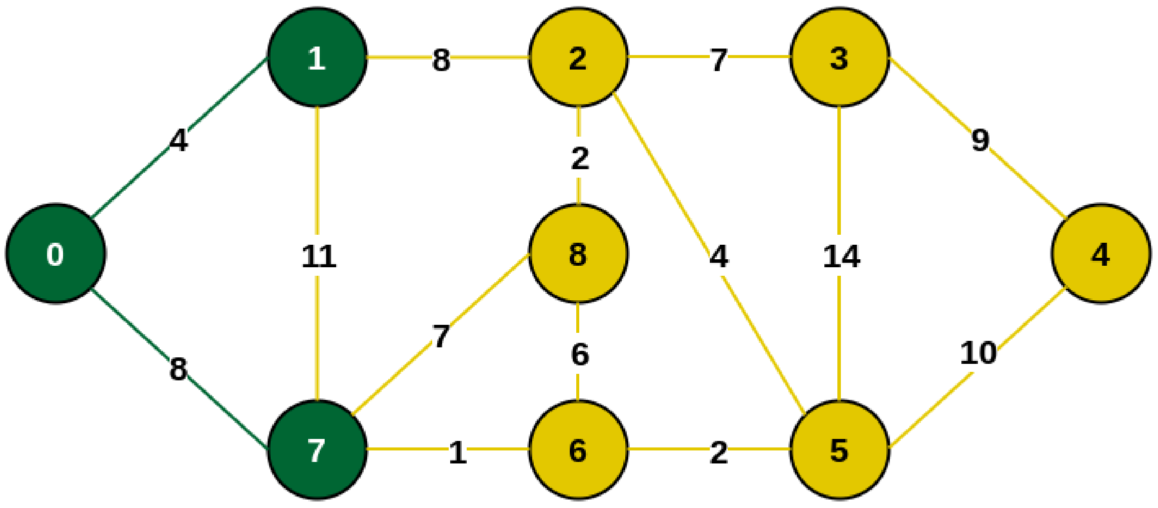
Select an arbitrary starting vertex. Here we have selected 0

<u>Step 2:</u> 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.



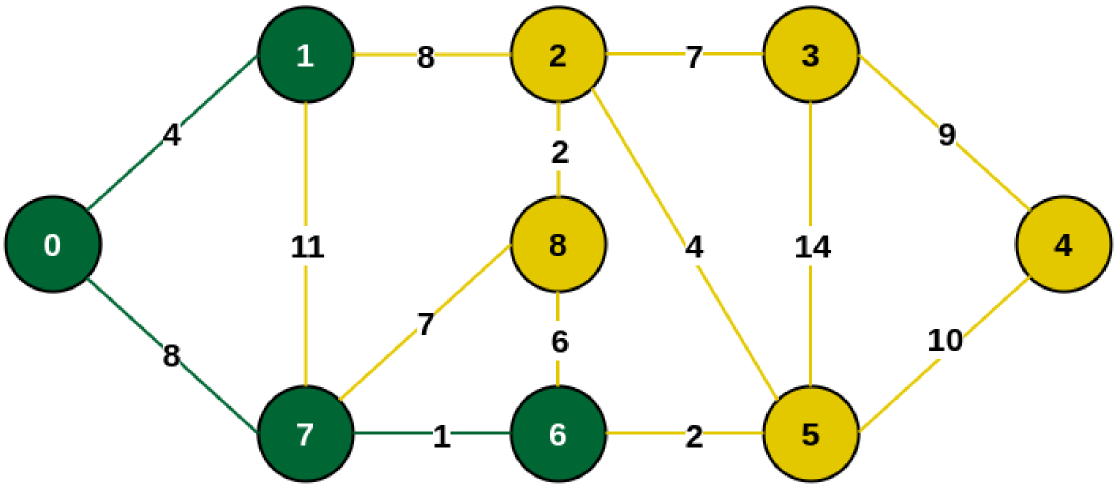
Minimum weighted edge from MST to other vertices is 0-1 with weight 4

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].*



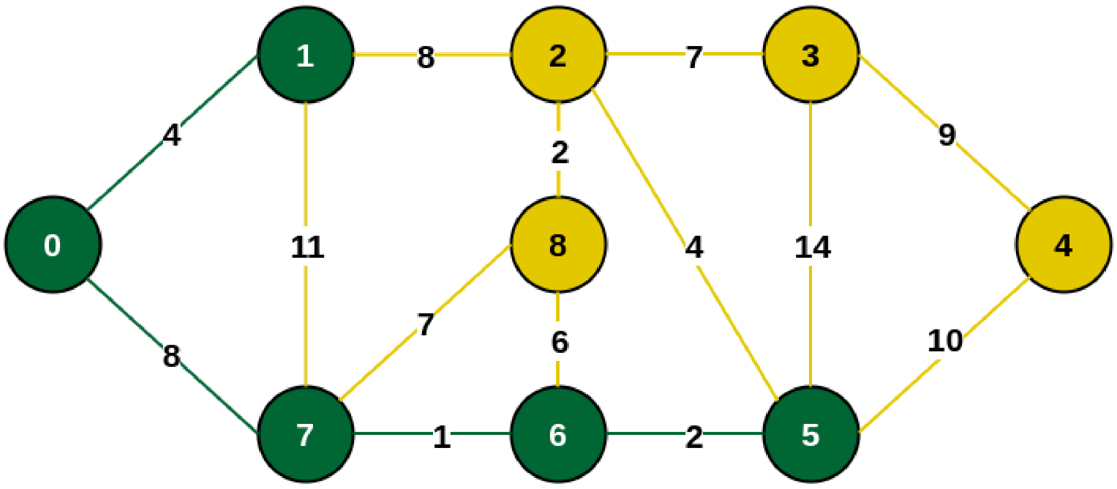
Minimum weighted edge from MST to other vertices is 0-7 with weight 8

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).*



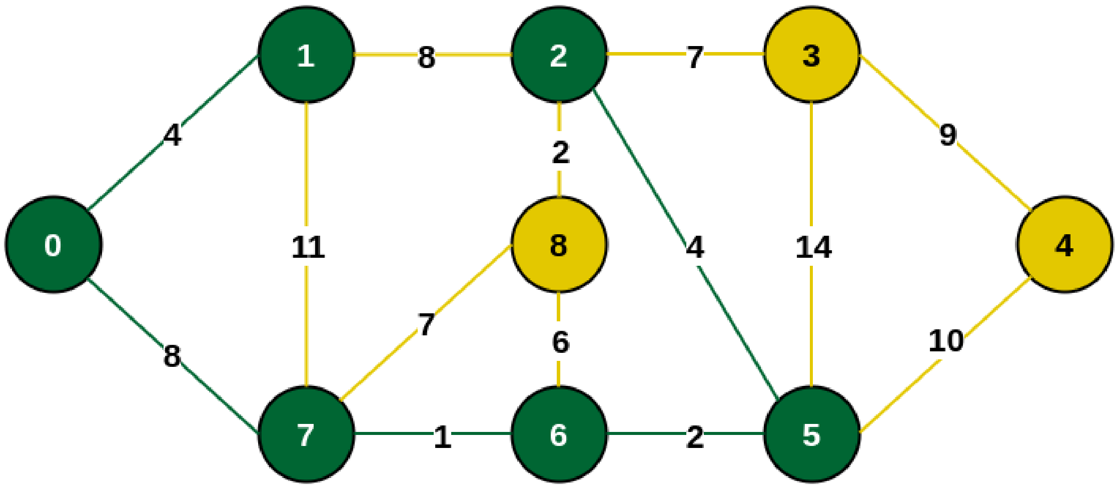
Minimum weighted edge from MST to other vertices is 7-6 with weight 1

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.*



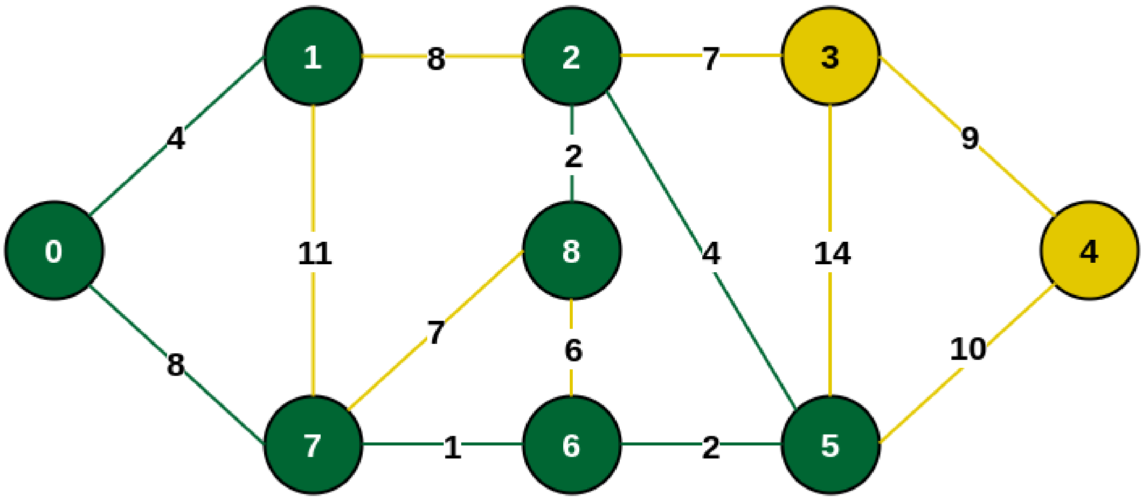
Minimum weighted edge from MST to other vertices is 6-5 with weight 2

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.



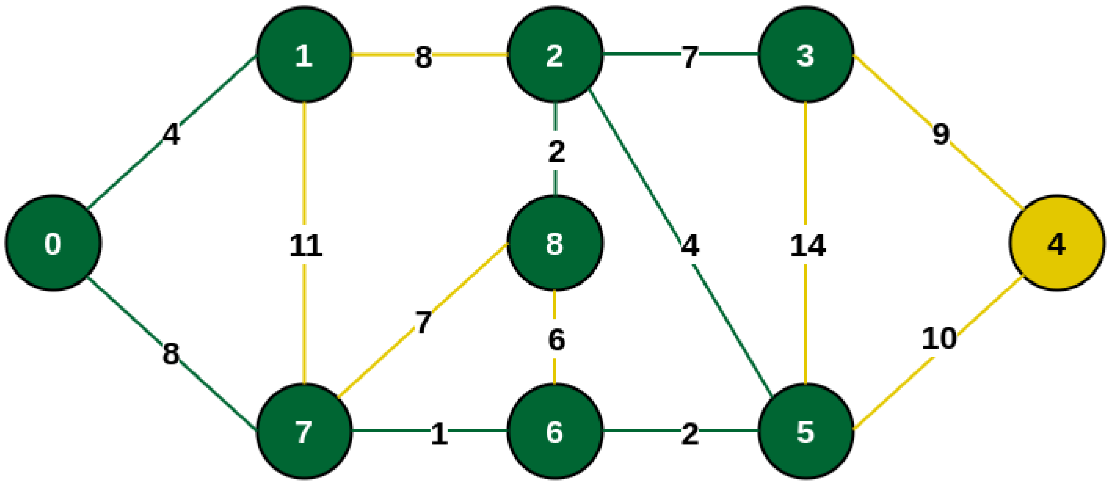
Minimum weighted edge from MST to other vertices is 5-2 with weight 4

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.*



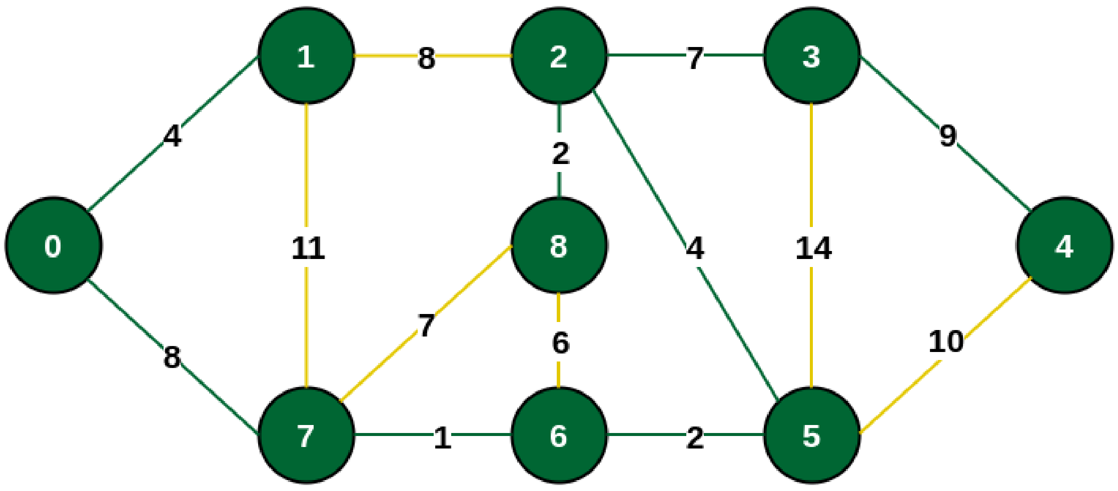
Minimum weighted edge from MST to other vertices is 2-8 with weight 2

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.*



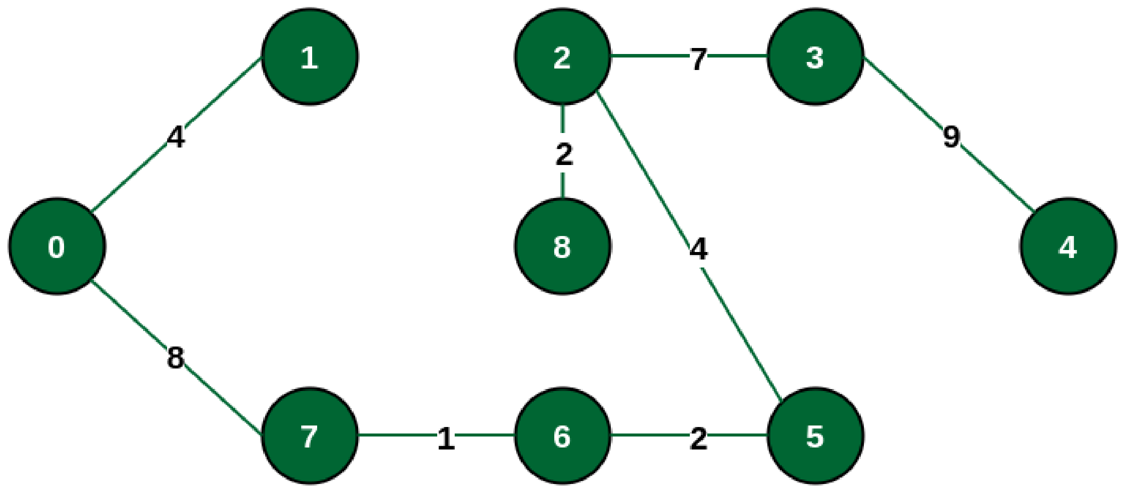
Minimum weighted edge from MST to other vertices is 2-3 with weight 7

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



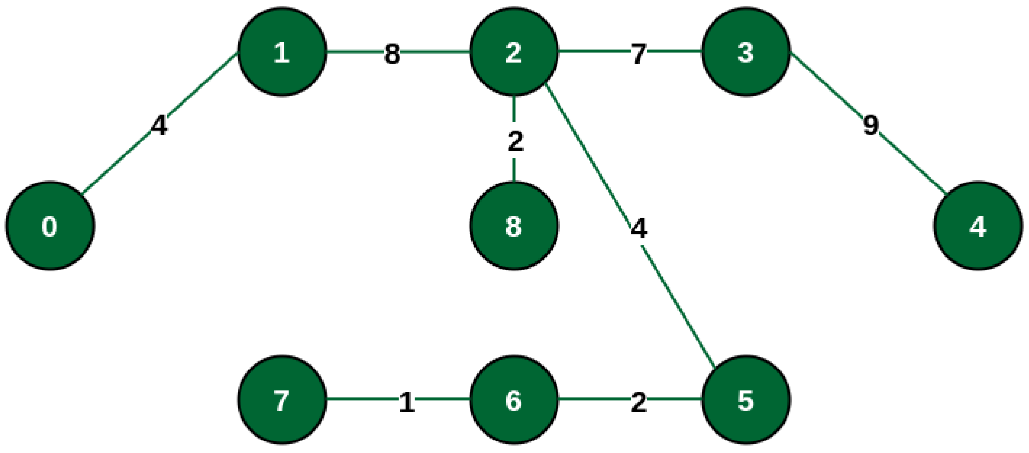
Minimum weighted edge from MST to other vertices is 3-4 with weight 9

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

Note: *If we had selected the edge {1, 2} in the third step then the MST would look like the following.*



Alternative MST structure

Heapsort :

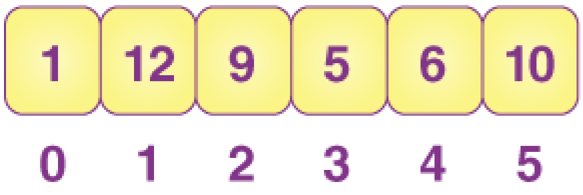
Heapsort is a popular terminology that belongs to the heap data structure family.

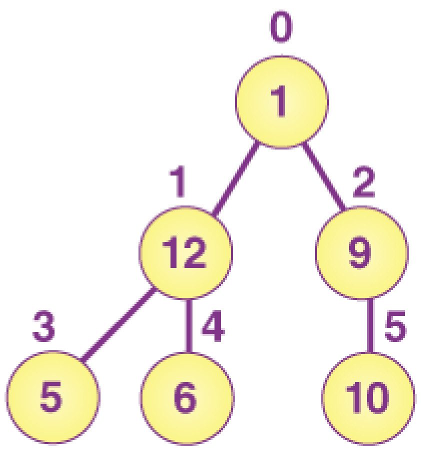
It is a comparison-based sorting method based on Binary Heap data structure. In heapsort, we generally prefer the heapify method to sort the elements.

A heap is a complete binary tree.

the binary tree is a tree in which the node can have the utmost two children.

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled,



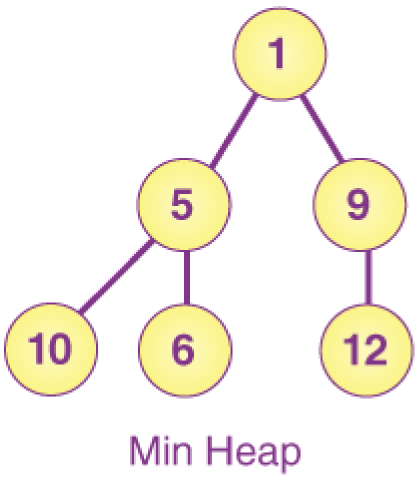


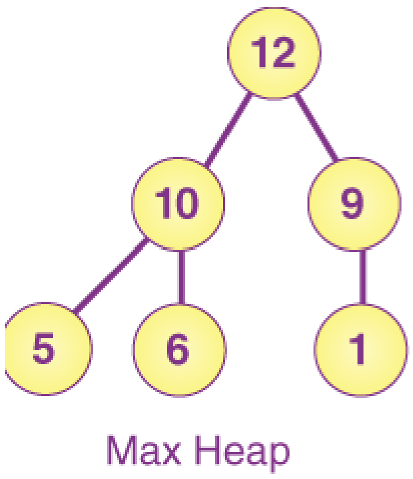
Heap Data Structure:

Heap is a special data structure. It has a tree structure. A heap data should follow some properties, then only we can consider it a genuine heap data structure. The properties are:

- It should be a complete binary tree.

- It should be either max-heap or min-heap. In short, it should follow all the properties of a binary heap tree.

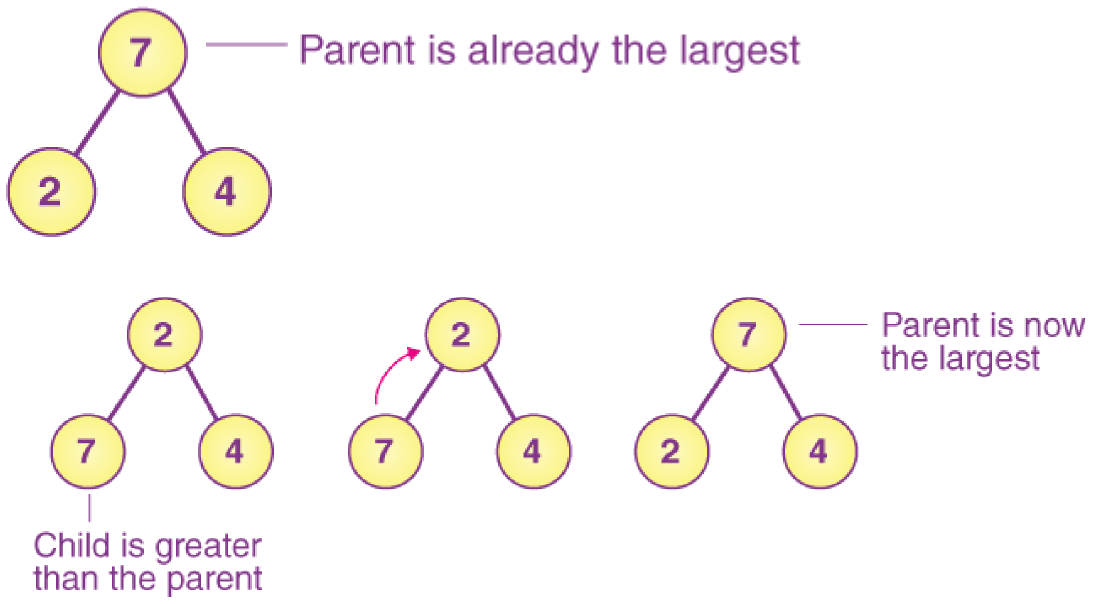




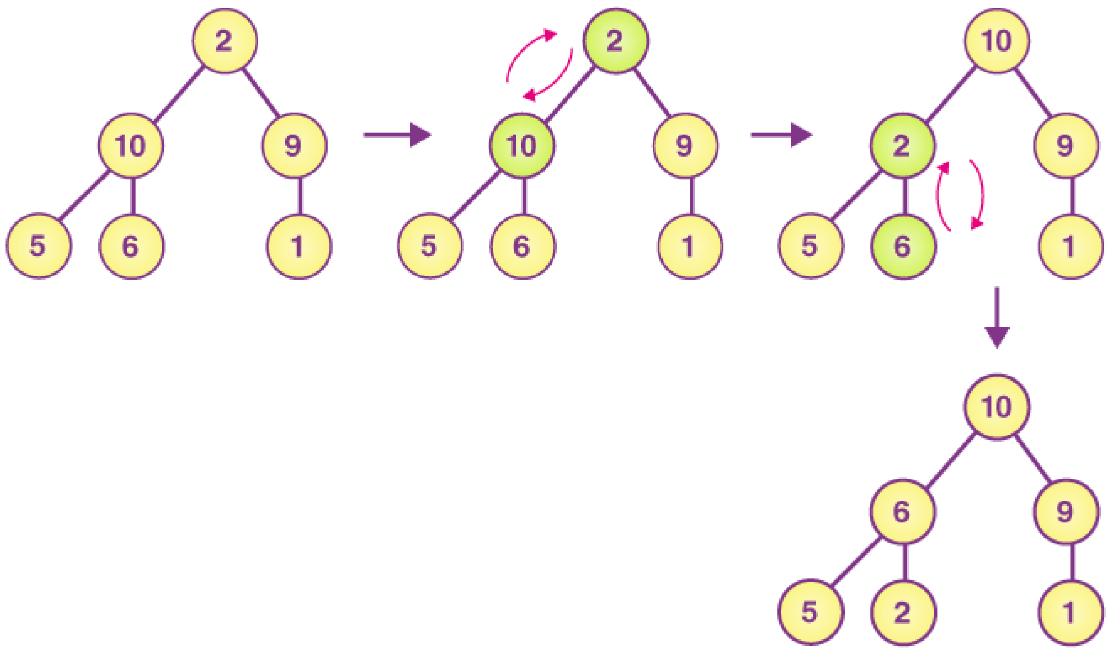
Heapify Method

Heapify Method:

When it comes to creating a heap tree(Min Heap or Max Heap), the heapify method is most suitable, because it takes less time as compared to the other methods like Insert key one by one in the given order.


7
2
4
Parent is already the largest
2
7
4
Child is greater than the parent
2
7
4
7
2
4
Parent is now the largest

To perform the heapify method, we have to compare the nodes and swap them if required.



To maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node.

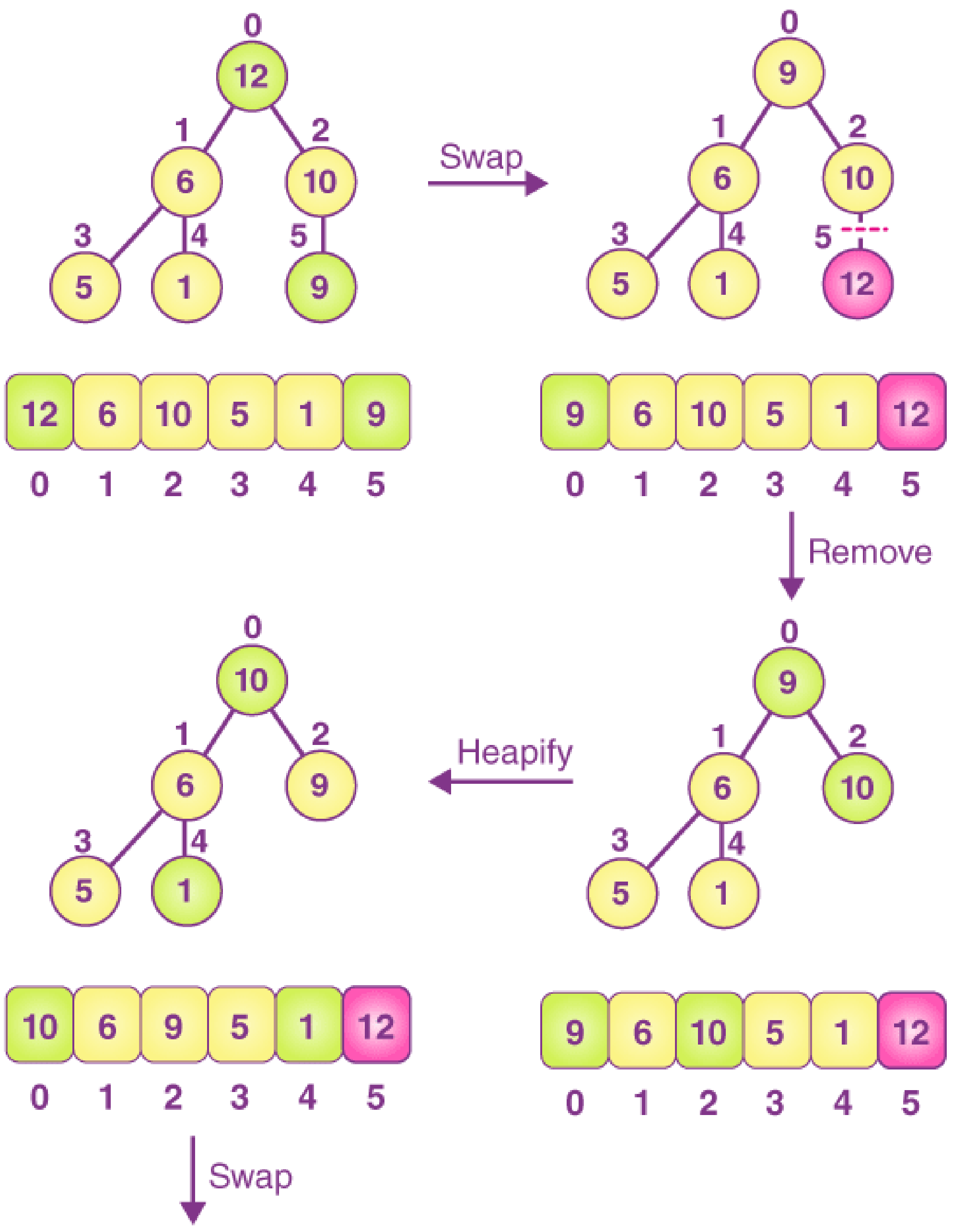
Heap Sort:

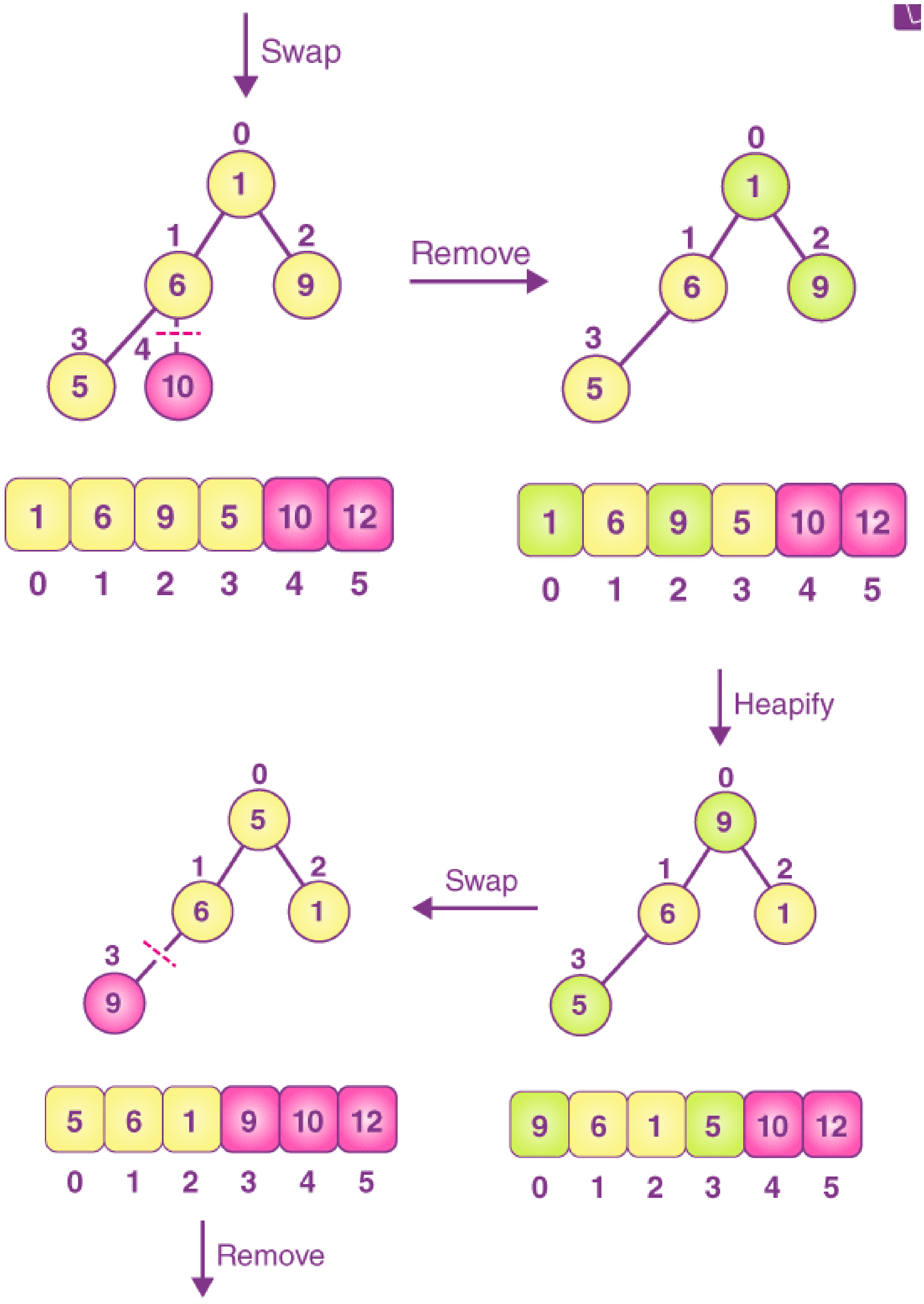
Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

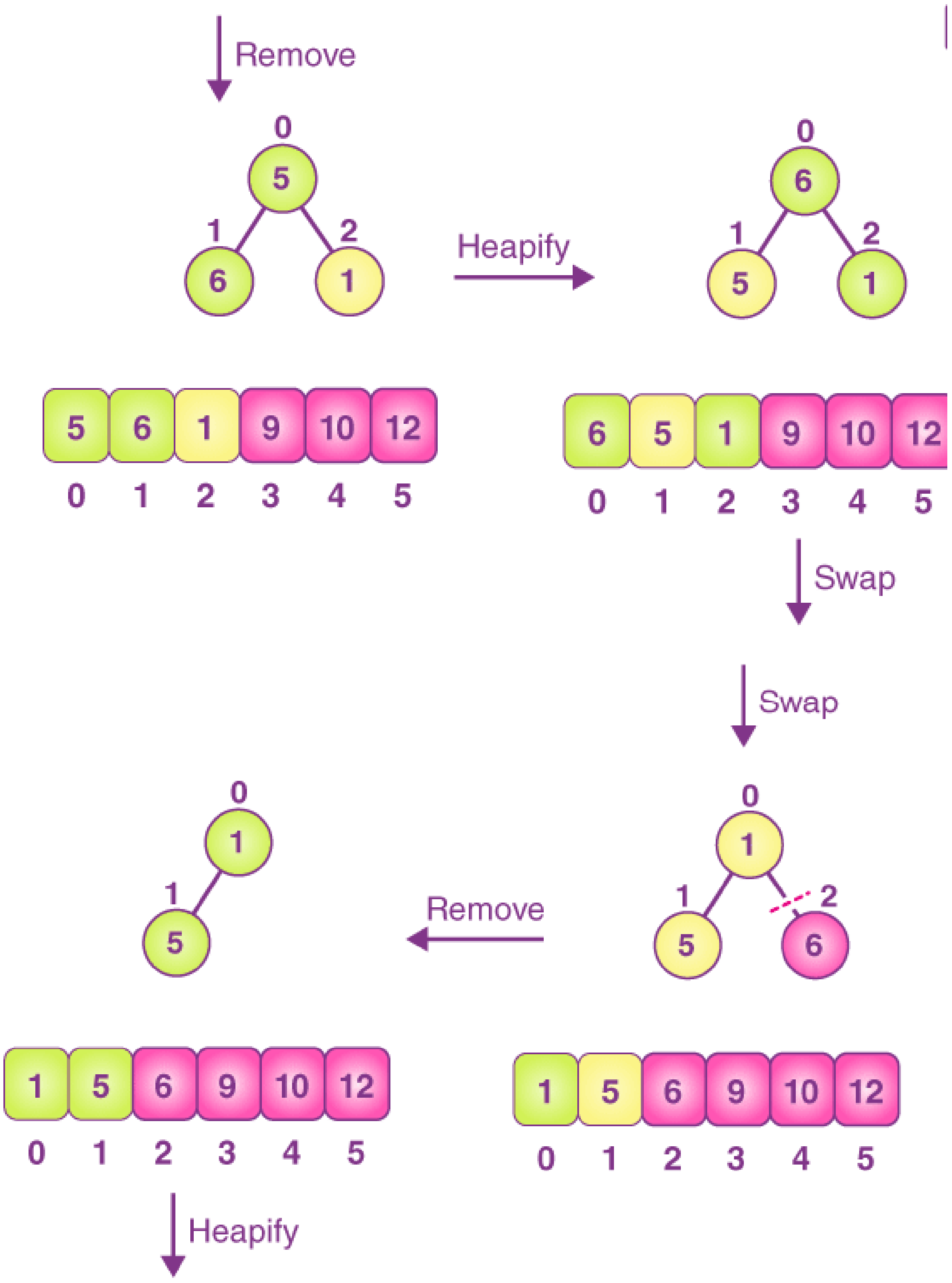
Heapsort is the in-place sorting algorithm.

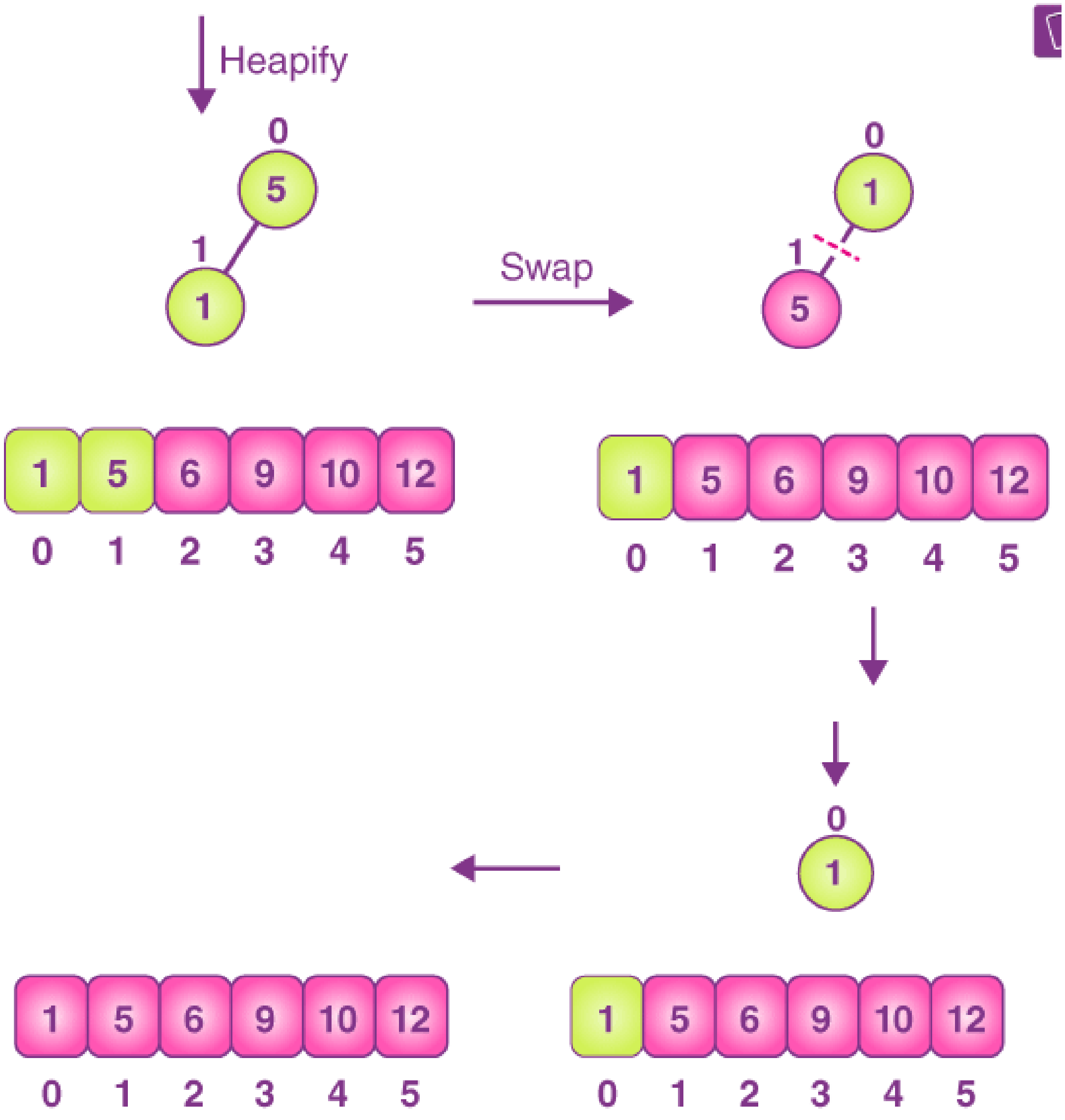
Working of Heap Sort

1. If it is about the max heap, then the highest element is stored at the root node.
2. **Swap:** Extract the root element and we need to place the last element of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.


Swap
Remove
Heapify
Swap


Swap
Remove
Heapify
Swap
Remove
1 6 9 5 10 12
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
9 6 1 5 10 12
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5


Remove
Heapify
5 6 1 9 10 12
0 1 2 3 4 5
6 5 1 9 10 12
0 1 2 3 4 5
Swap
Swap
Remove
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
Heapify


Heapify
0
5
1
1
1
5
6
9
10
12
0
1
2
3
4
5
Swap
0
1
1
5
1
5
6
9
10
12
0
1
2
3
4
5
0
1
1
5
6
9
10
12
0
1
2
3
4
5
1
5
6
9
10
12
0
1
2
3
4
5

Job Sequencing Problem

You are given a set of N jobs where each job comes with a **deadline** and **profit**. The profit can only be earned upon completing the job within its deadline. Find the **number of jobs** done and the **maximum profit** that can be obtained. Each job takes a **single unit** of time and only **one job** can be performed at a time.

Input: N = 4, Jobs = {(1,4,20),(2,1,10),(3,1,40),(4,1,30)}

Output: 2 60

Explanation: The 3rd job with a deadline 1 is performed during the first unit of time .The 1st job is performed during the second unit of time as its deadline is 4.
Profit = 40 + 20 = 60

Example 2:

Input: N = 5, Jobs = {(1,2,100),(2,1,19),(3,2,27),(4,1,25),(5,1,15)}

Output: 2 127

Explanation: The first and third job both having a deadline 2 give the highest profit.
Profit = 100 + 27 = 127

Approach: The strategy to maximize profit should be to pick up jobs that offer **higher profits.** Hence we should **sort** the jobs in descending order of profit. Now say if a job has a deadline of 4 we can perform it anytime between day 1-4, but it is preferable to perform the job on its **last day**. This leaves enough empty slots on the previous days to perform other jobs.

Basic Outline of the approach:-

- Sort the jobs in descending order of profit.
- If the maximum deadline is x, make an array of size x .Each array index is set to -1 initially as no jobs have been performed yet.
- For every job check if it can be performed on its last day.

- If possible mark that index with the job id and add the profit to our answer.
- If not possible, loop through the previous indexes until an empty slot is found.

\

An Activity Selection Problem

The activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well designed and simple method for selecting a maximum- size set of manually compatible activities.

Suppose S = {1, 2....n} is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc. Each Activity "i" has **start time** s_i and a **finish time** f_i, where $s_i \leq f_i$. If selected activity "i" take place meanwhile the half-open time interval $[s_i, f_i)$. Activities i and j are **compatible** if the intervals (s_i, f_i) and $[s_i, f_i)$ do not overlap (i.e. i and j are compatible if $s_i \geq f_i$ or $s_i \geq f_i$). The activity-selection problem chosen the maximum- size set of mutually consistent activities.

S = (A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A9 A10)
Si = (1, 2, 3, 4, 7, 8, 9, 9, 11,12)
fi = (3, 5, 4, 7, 10, 9, 11, 13, 12,14)

Now, schedule A_1

Next schedule A_3 as A_1 and A_3 are non-interfering.

Next **skip** A_2 as it is interfering.

Next, schedule A_4 as A_1 A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1 A_3 A_4 and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as A_1 A_3 A_4 A_6 and A_7 are non-interfering.

Next, schedule A_9 as A_1 A_3 A_4 A_6 A_7 and A_9 are non-interfering.

Skip A_8 as it is interfering.

Next, schedule A_{10} as A_1 A_3 A_4 A_6 A_7 A_9 and A_{10} are non-interfering.

Thus the final Activity schedule is:

$$\boxed{(A_1\ A_3\ A_4\ A_6\ A_7\ A_9\ A_{10})}$$

Optimal Merge Patterns

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a **p-record file** and a **q-record file** requires possibly **p + q** record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of **n** sorted files **{f_1, f_2, f_3, …, f_n}**. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Examples

Let us consider the given files, f1, f2, f3, f4 and f5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

M1 = merge f1 and f2 => 20 + 30 = 50

M2 = merge M1 and f3 => 50 + 10 = 60

M3 = merge M2 and f4 => 60 + 5 = 65

M4 = merge M3 and f5 => 65 + 30 = 95

Hence, the total number of operations is

50 + 60 + 65 + 95 = 270

Sorting the numbers according to their size in an ascending order, we get the following sequence −

f_4, f_3, f_1, f_2, f_5

Hence, merge operations can be performed on this sequence

M_1 = merge f_4 and f_3 => 5 + 10 = 15

M_2 = merge M_1 and f_1 => 15 + 20 = 35

M_3 = merge M_2 and f_2 => 35 + 30 = 65

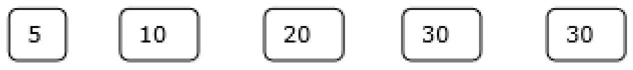
M_4 = merge M_3 and f_5 => 65 + 30 = 95

Therefore, the total number of operations is

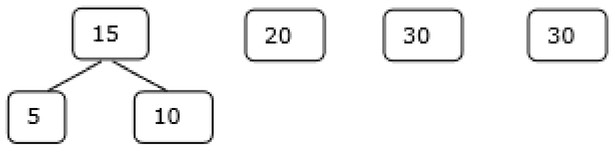
15 + 35 + 65 + 95 = 210

In this context, we are now going to solve the problem using this algorithm.

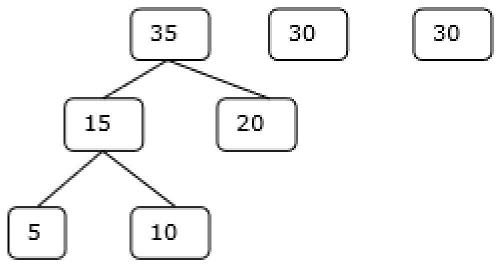
Initial Set



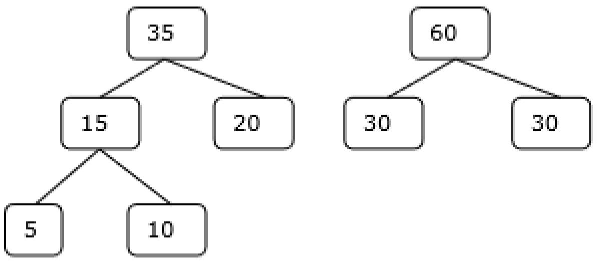
Step 1



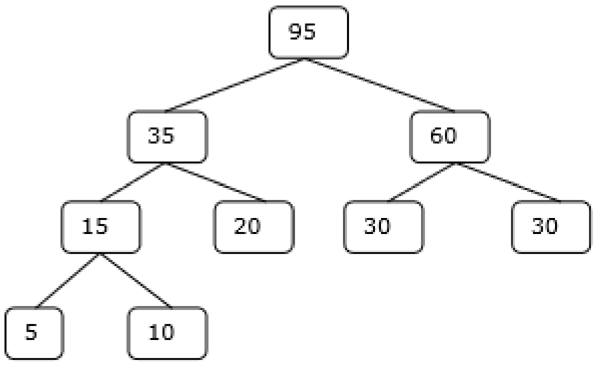
Step 2



Step 3



Step 4



Hence, the solution takes 15 + 35 + 60 + 95 = 205 number of comparisons.

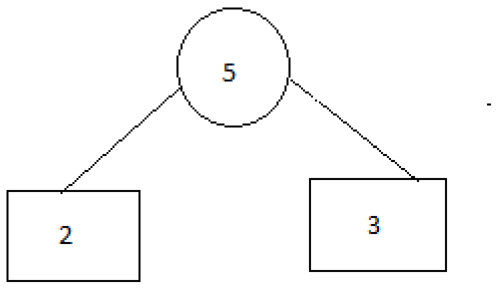
Optimal merge pattern example

Given a set of unsorted files: 5, 3, 2, 7, 9, 13

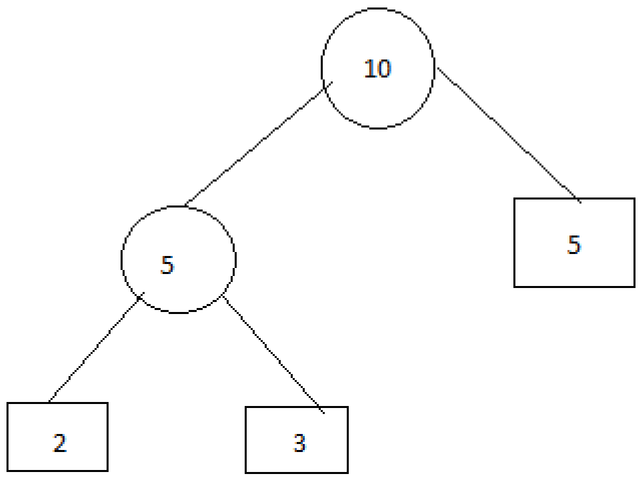
Now, arrange these elements in ascending order: 2, 3, 5, 7, 9, 13

Step 1: Insert 2, 3

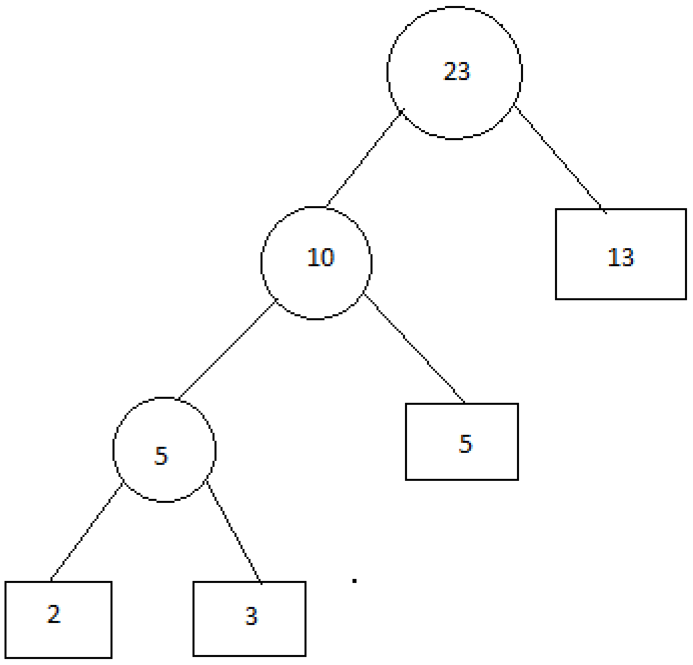
Step 2:



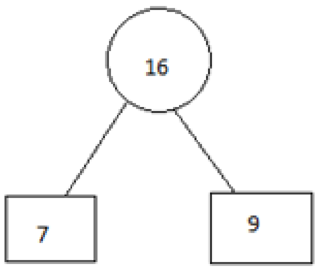
Step 3: Insert 5

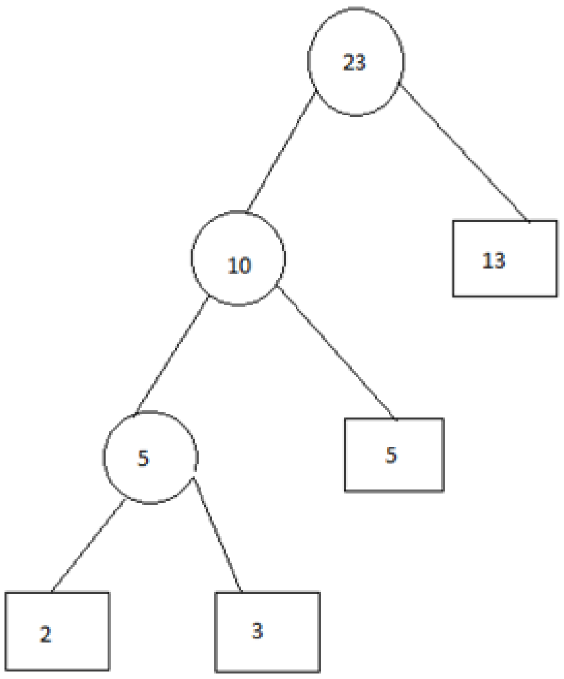


Step 4: Insert 13

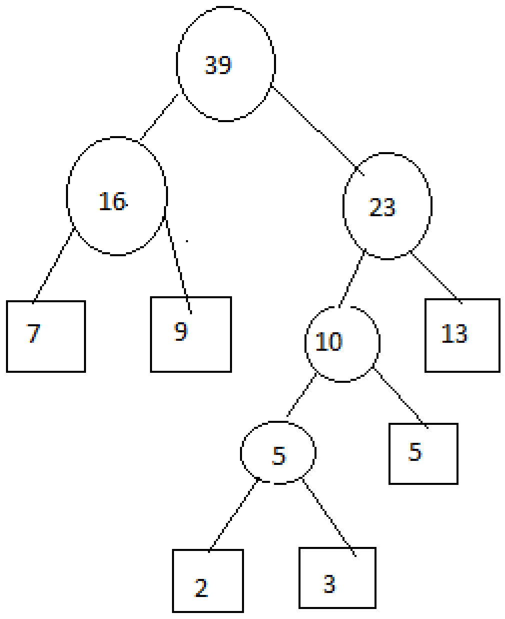


Step 5: Insert 7 and 9





Step 6:



So, The merging cost = 5 + 10 + 16 + 23 + 39 = 93