BINARY TREES

Mahsa Ziraksima
ACIBADEM UNIVERSITY
Fall 2025

SUMMARY

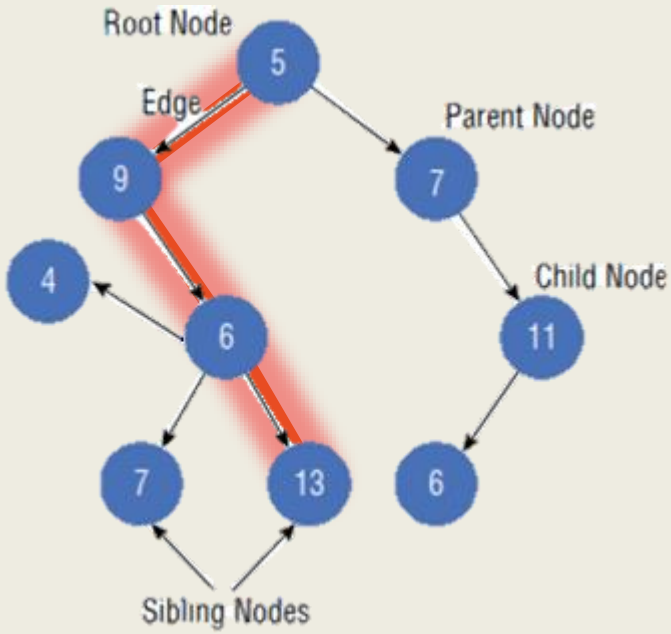
- Performance
- Data structures
- Tree Traversals
- Invert a Binary Tree

TREE

- **Tree** is a nonlinear abstract data type made up of nodes connected in a hierarchical structure.
- Common tree operations include inserting, searching, and deleting.
- Types of tree data structures:
  - *general trees*,
  - *AVL trees*,
  - *red- black trees*,
  - *binary trees*,
  - *binary search trees*,
  - *and more.*

GENERAL TREE

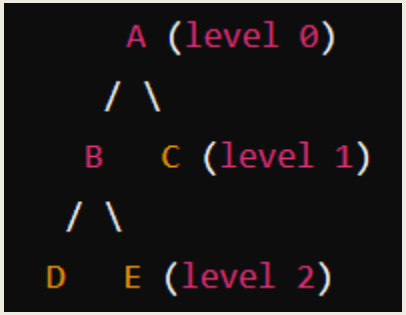
- It is a data structure that starts with a node at the top.
- The node at the top of a tree is called the **root** node.
- Each node connected underneath a node in a tree is its **child** node.
- A node with one or more child nodes is called a **parent** node.
- **Sibling** nodes share the same parent.
- The connection between two nodes in a tree is called an **edge**.
- A node without child nodes is called a **leaf** node.
- A node with child nodes is called a **branch** node.



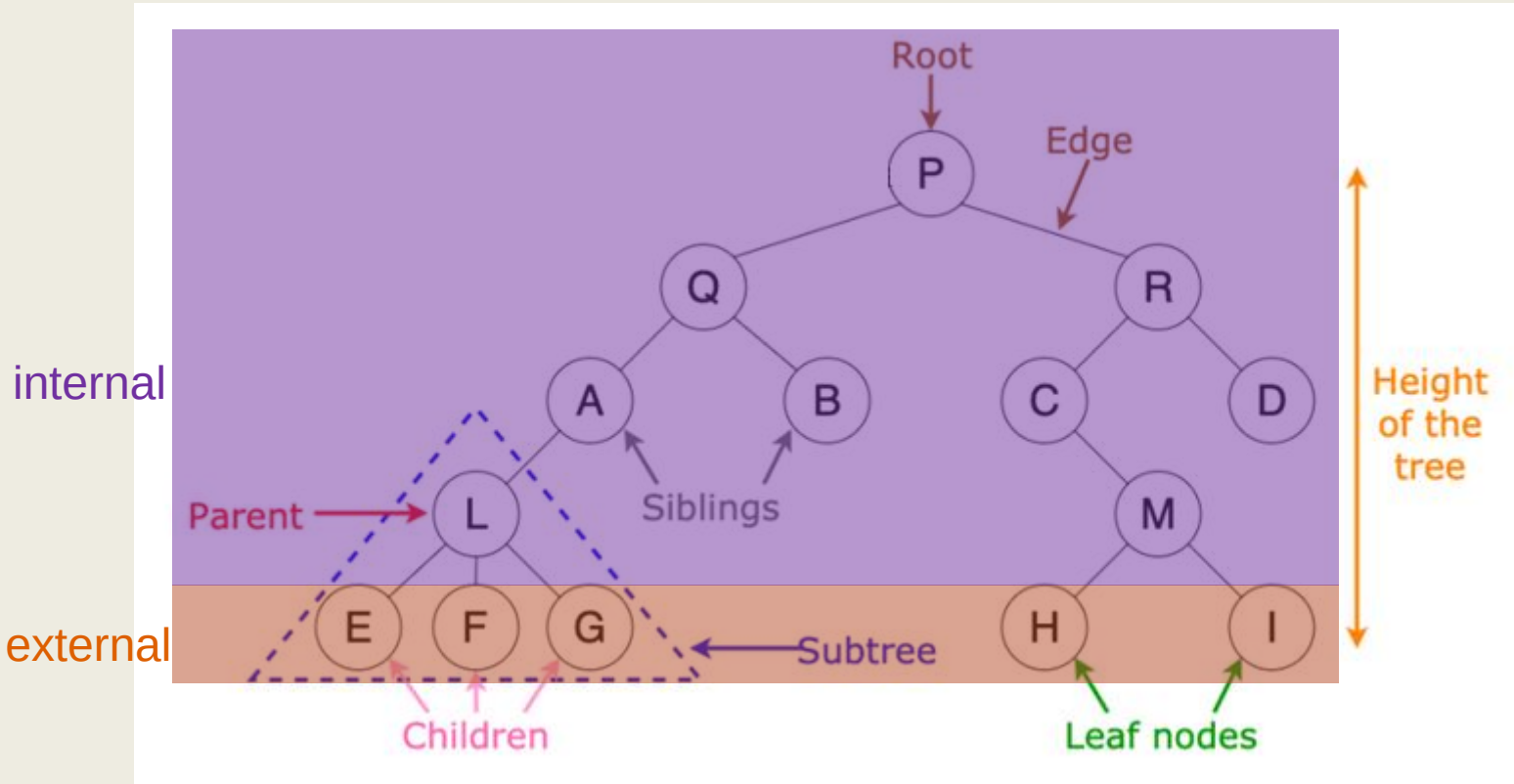
GENERAL TREE

- The distance from each node to the root of the tree is called the level of that node.
- The number of subtrees of each node is the degree of that node.
- The largest degree (number of child) of nodes in a tree is called the degree of the tree.
- Nodes whose degree is zero are called leaf.
- The maximum level of nodes in a tree is called the height (depth) of the tree.
- The leaf nodes are called the external nodes of the tree and

  the other nodes are called the internal nodes of the tree.

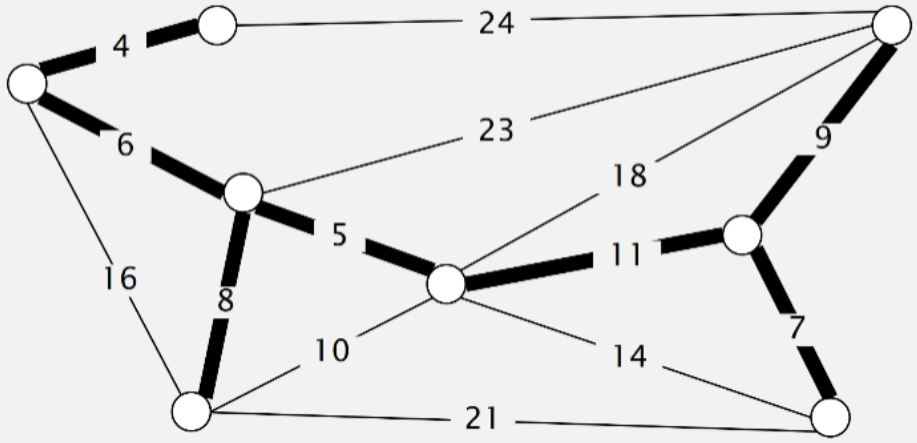


GENERAL TREE



MINIMUN SPANNING TREE

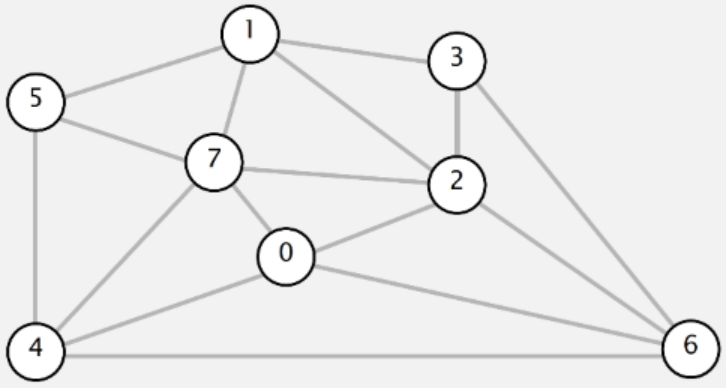
- In graph theory, a tree is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a connected acyclic undirected graph.
- A spanning tree of *G* is a subgraph (tree) *T* that is:
  - *Connected*
  - *Acyclic*
  - *Includes all of the vertices*
  - *Min weighted*



minimum spanning tree T
(cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7)

MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.

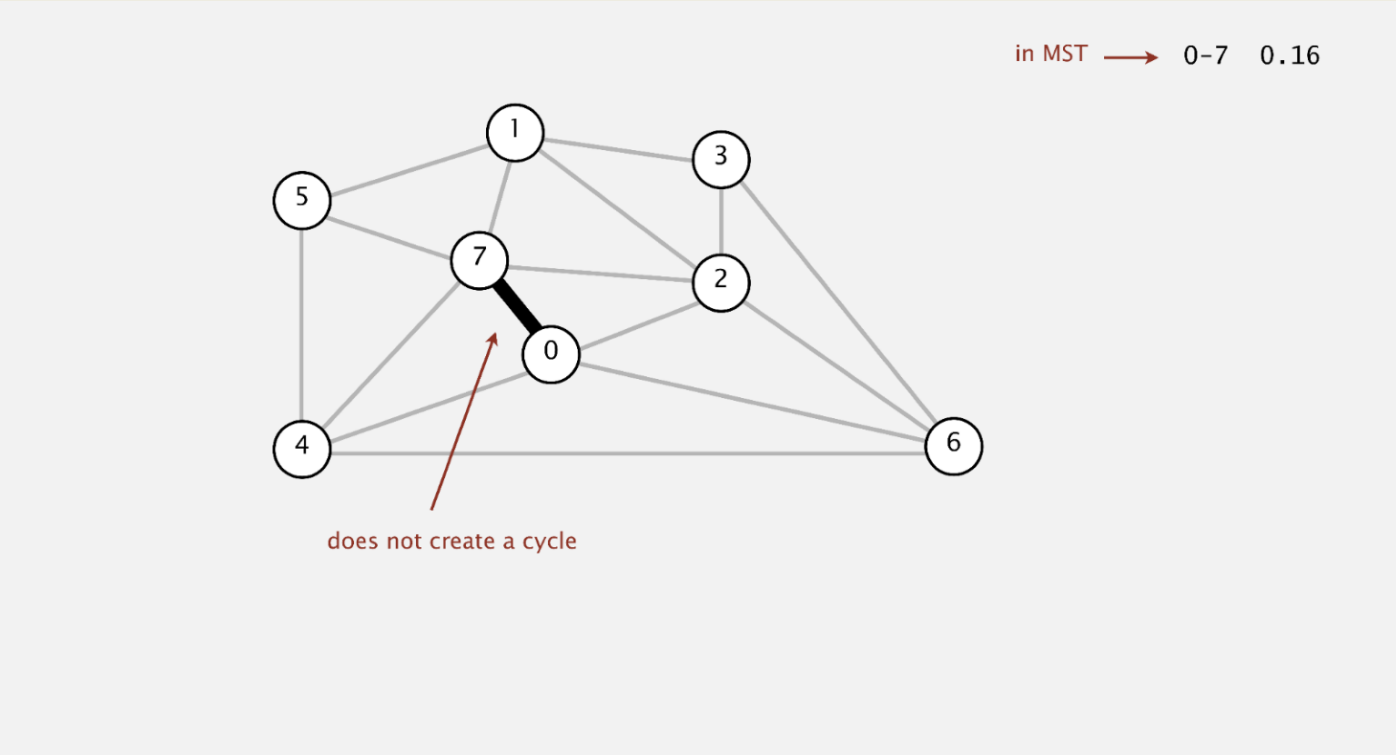


an edge-weighted graph

```
0-7  0.16
2-3  0.17
1-7  0.19
0-2  0.26
5-7  0.28
1-3  0.29
1-5  0.32
2-7  0.34
4-5  0.35
1-2  0.36
4-7  0.37
0-4  0.38
6-2  0.40
3-6  0.52
6-0  0.58
6-4  0.93
```

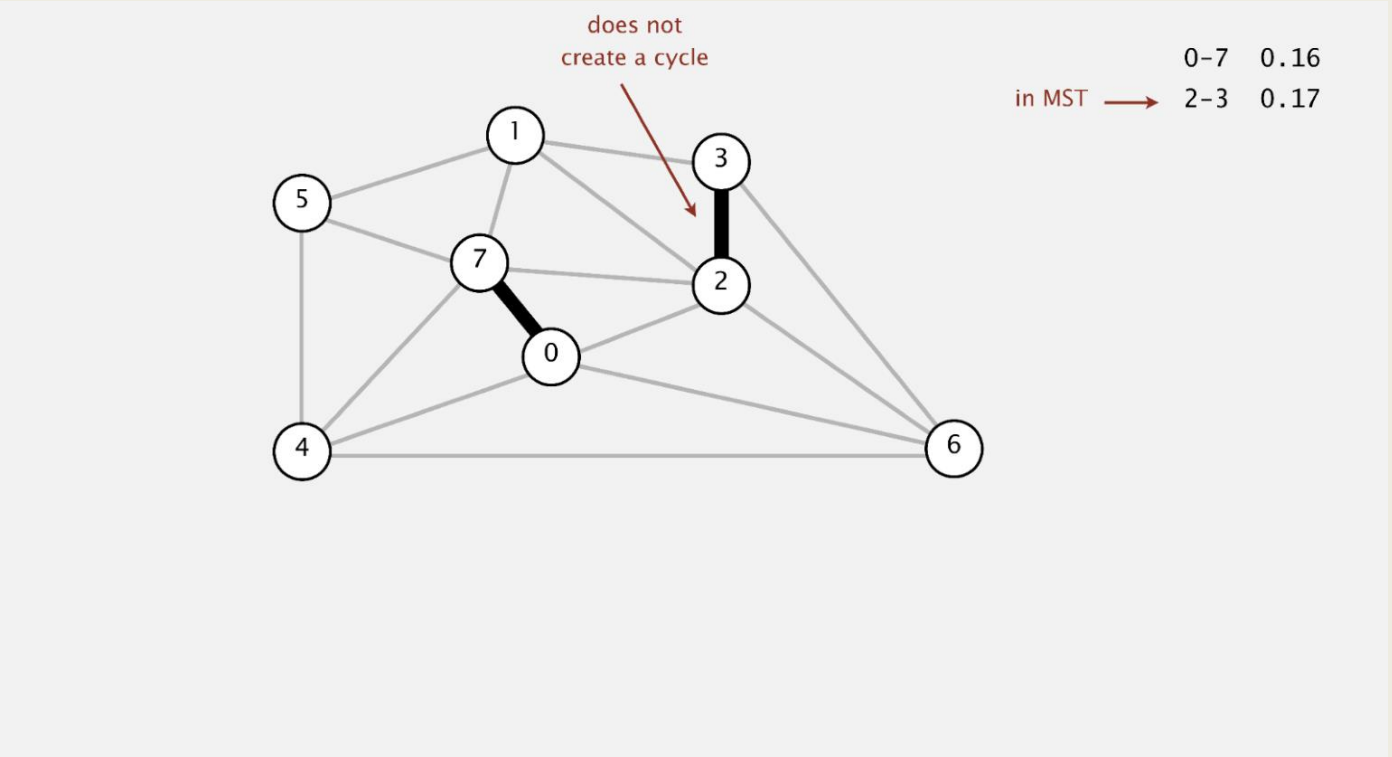
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



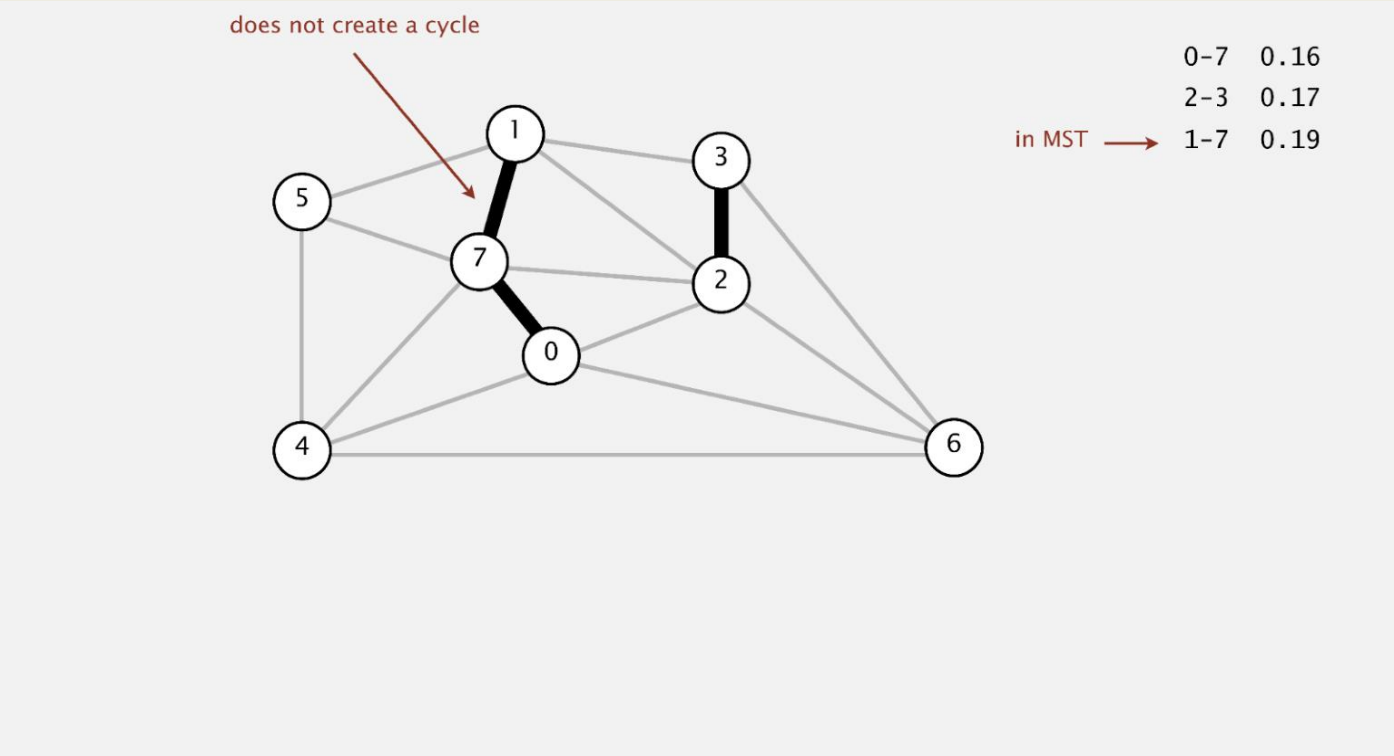
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



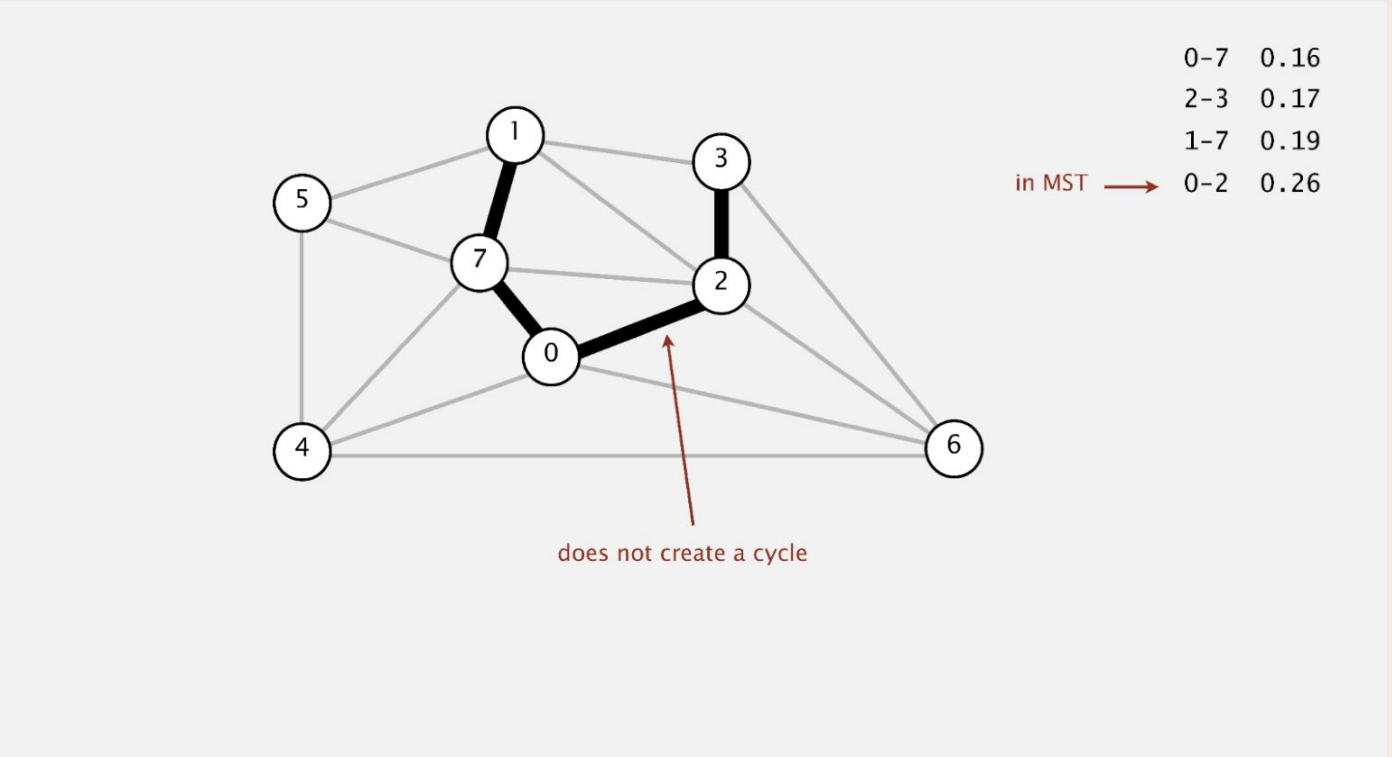
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



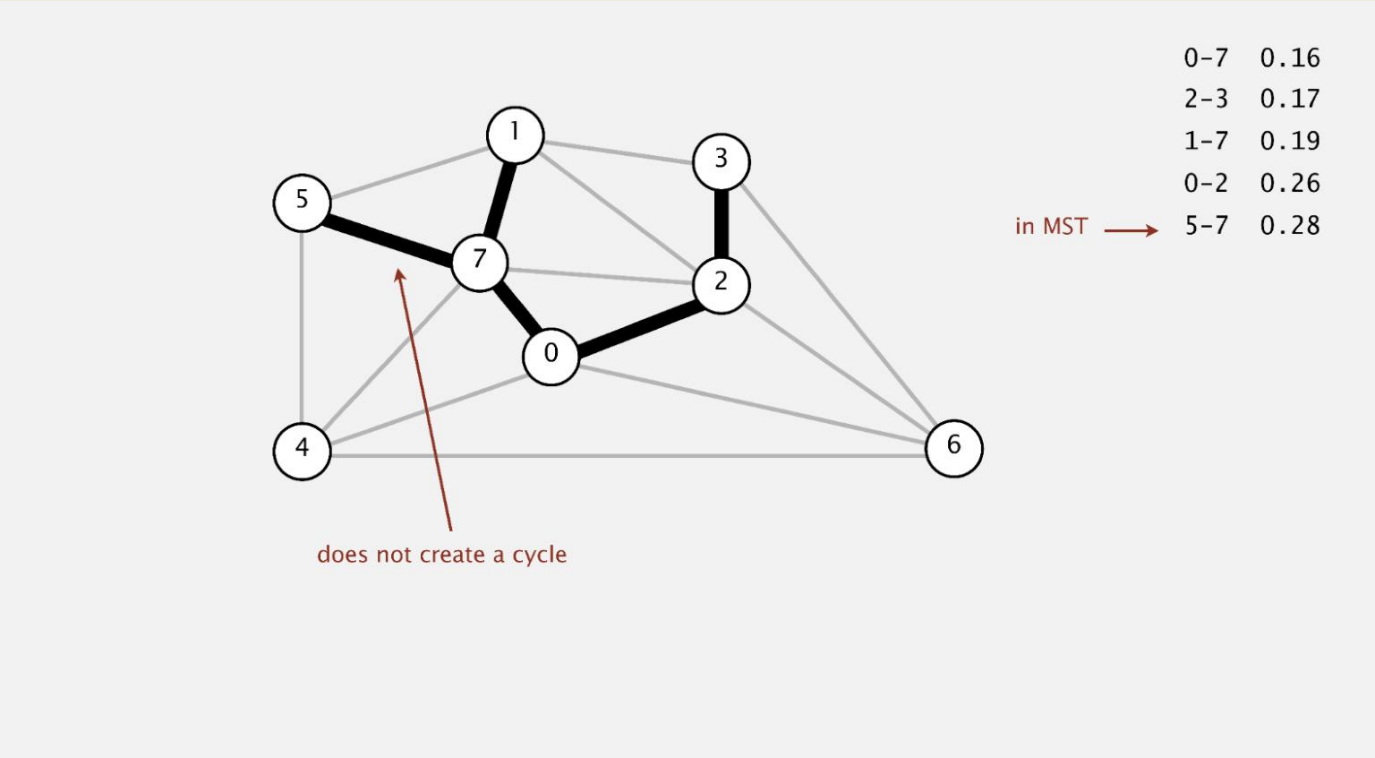
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



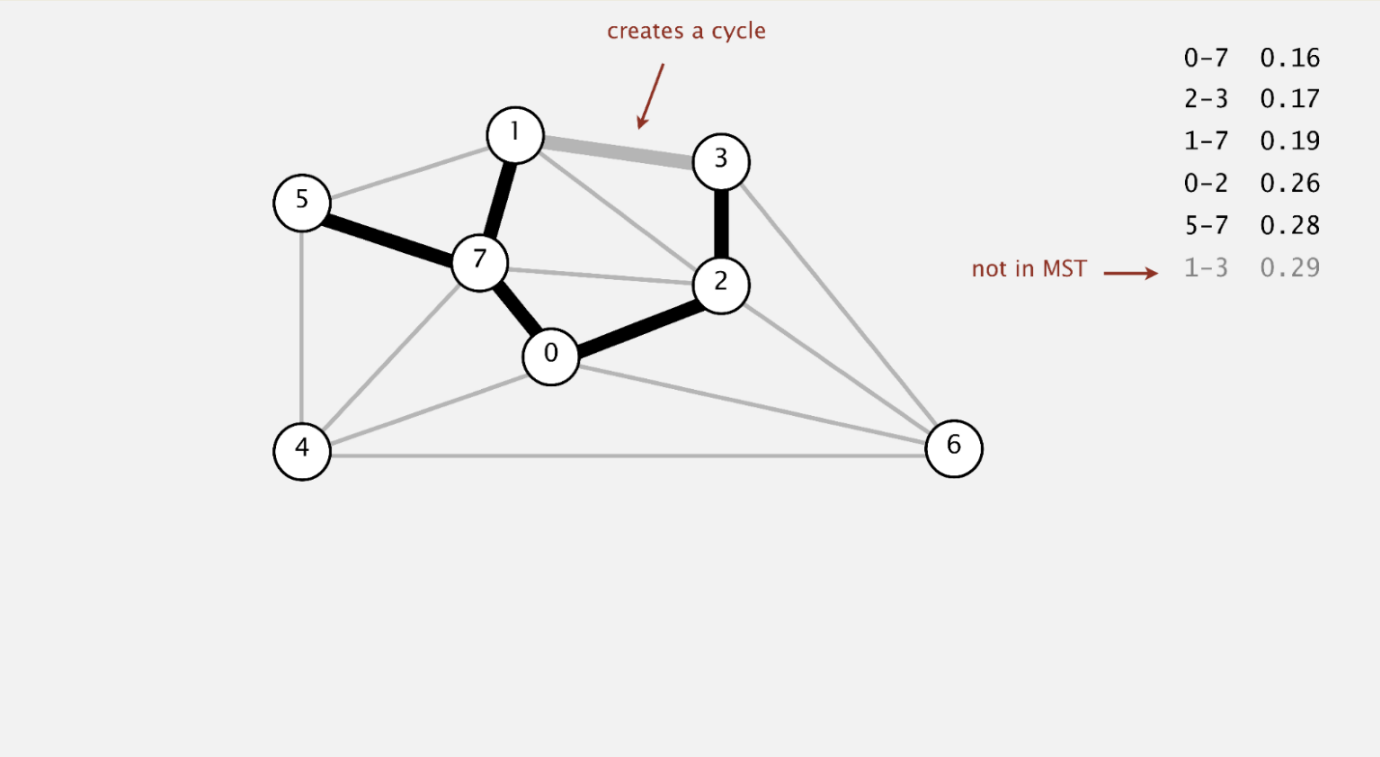
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



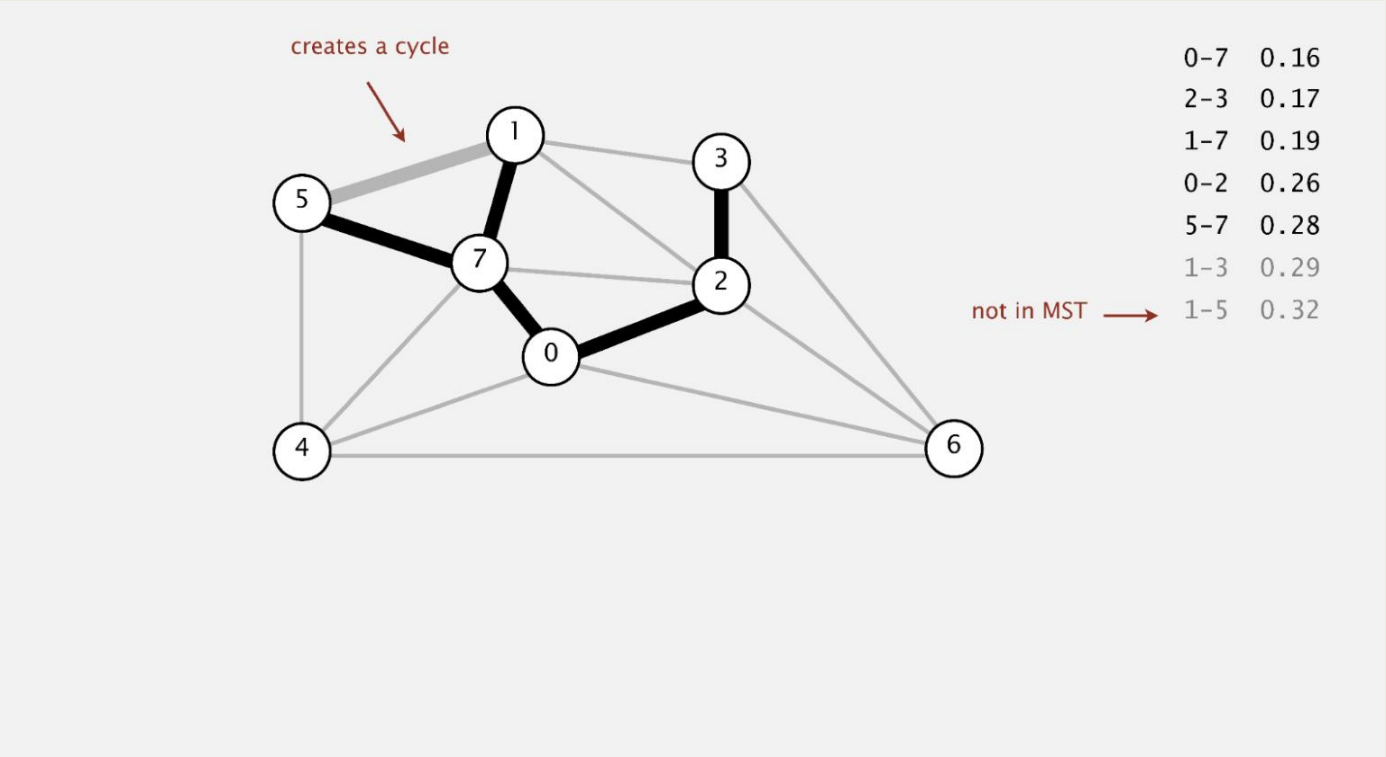
MINIMUN SPANNING TREE

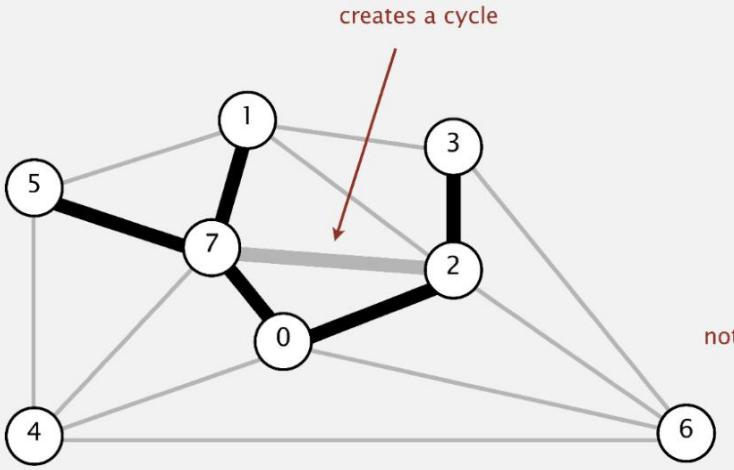
- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.

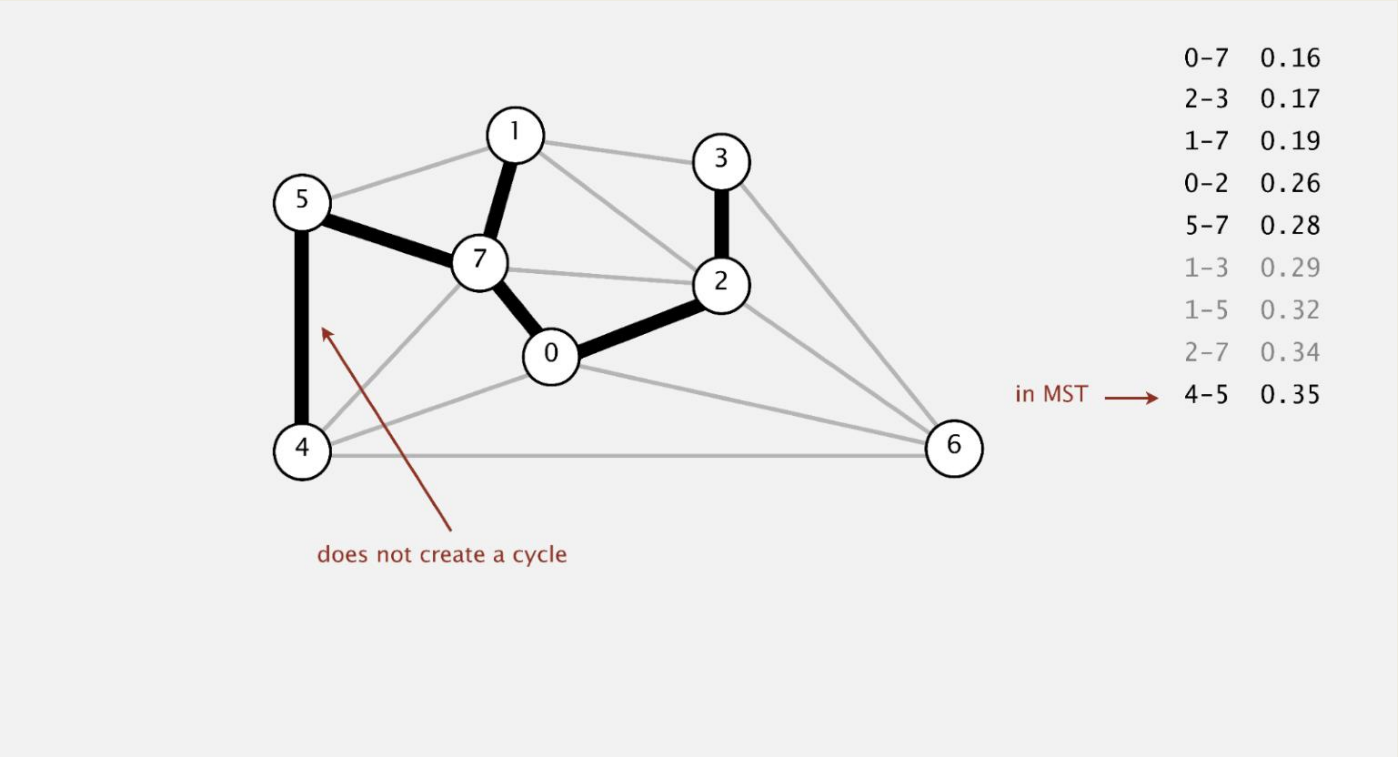




0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
not in MST → 2-7	0.34

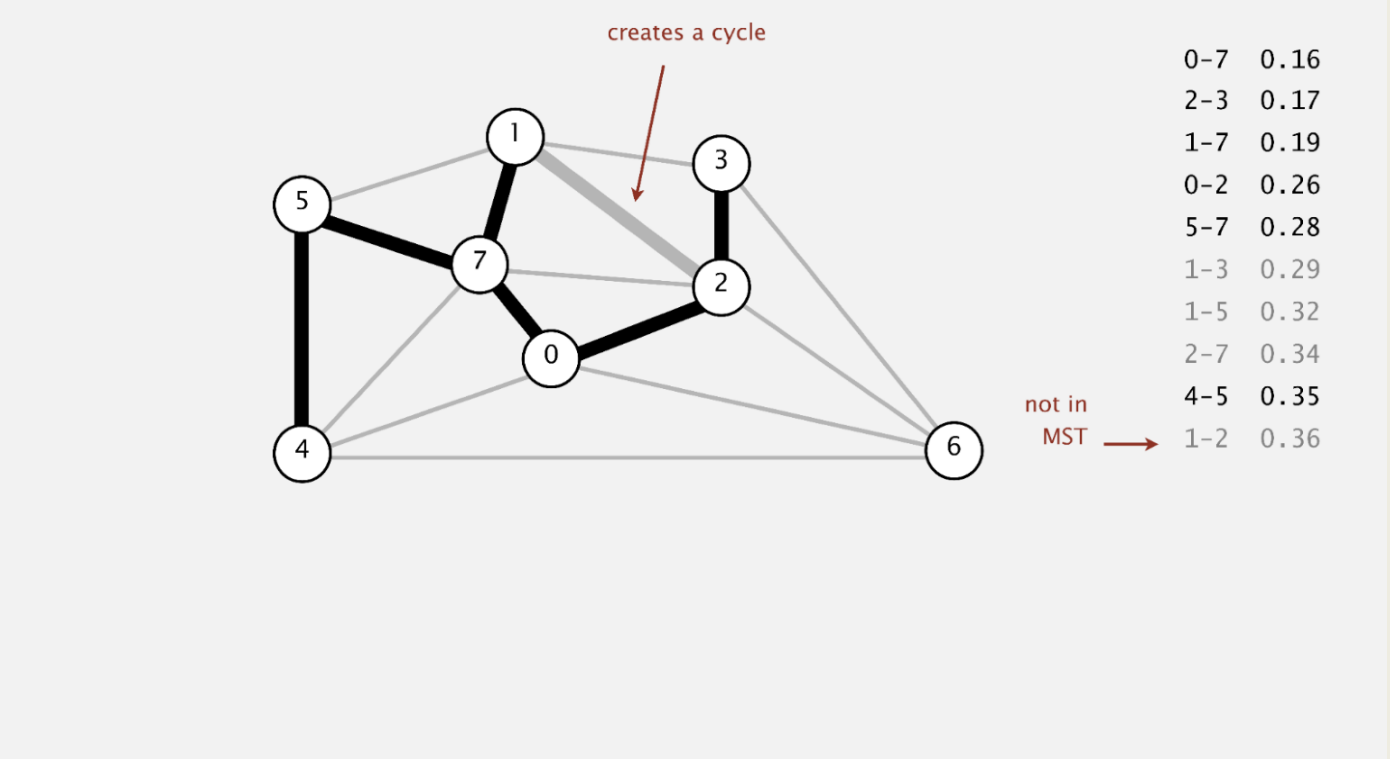
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



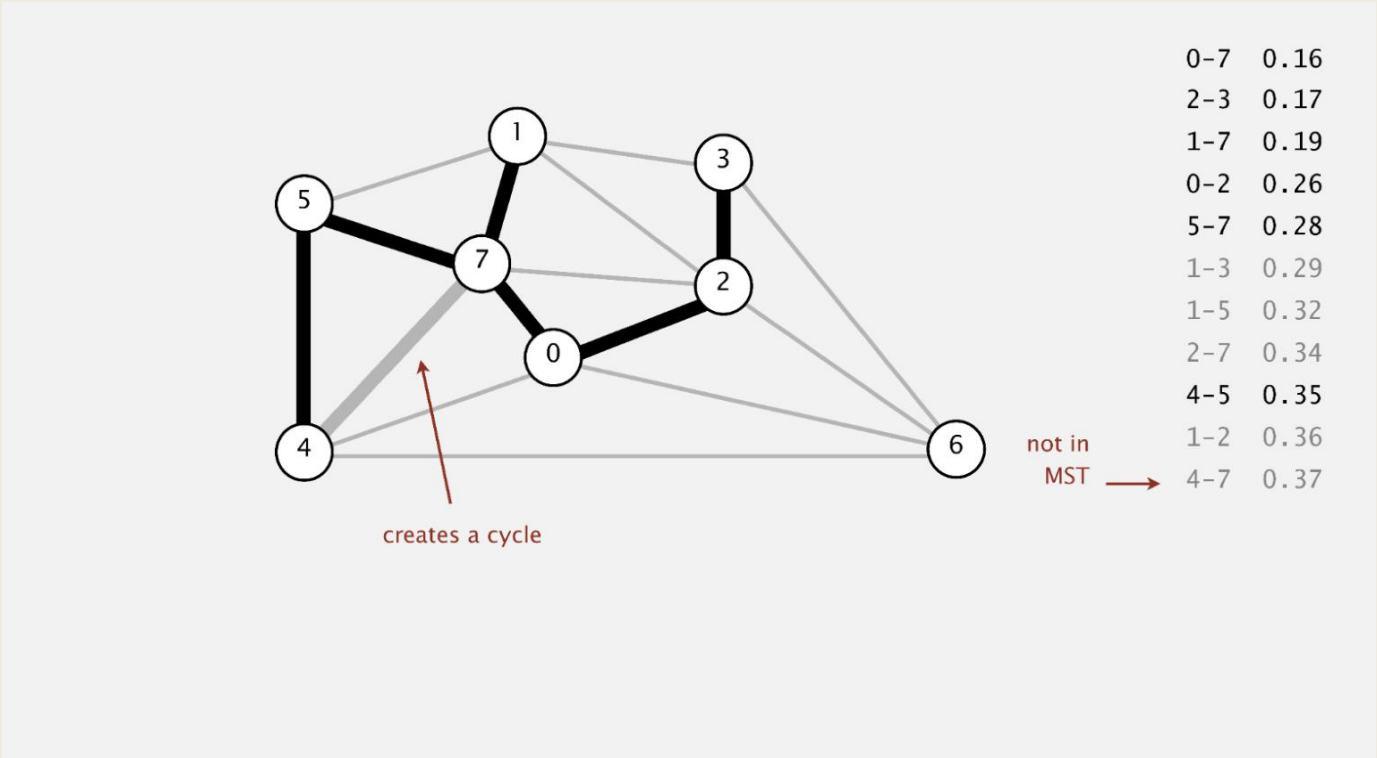
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



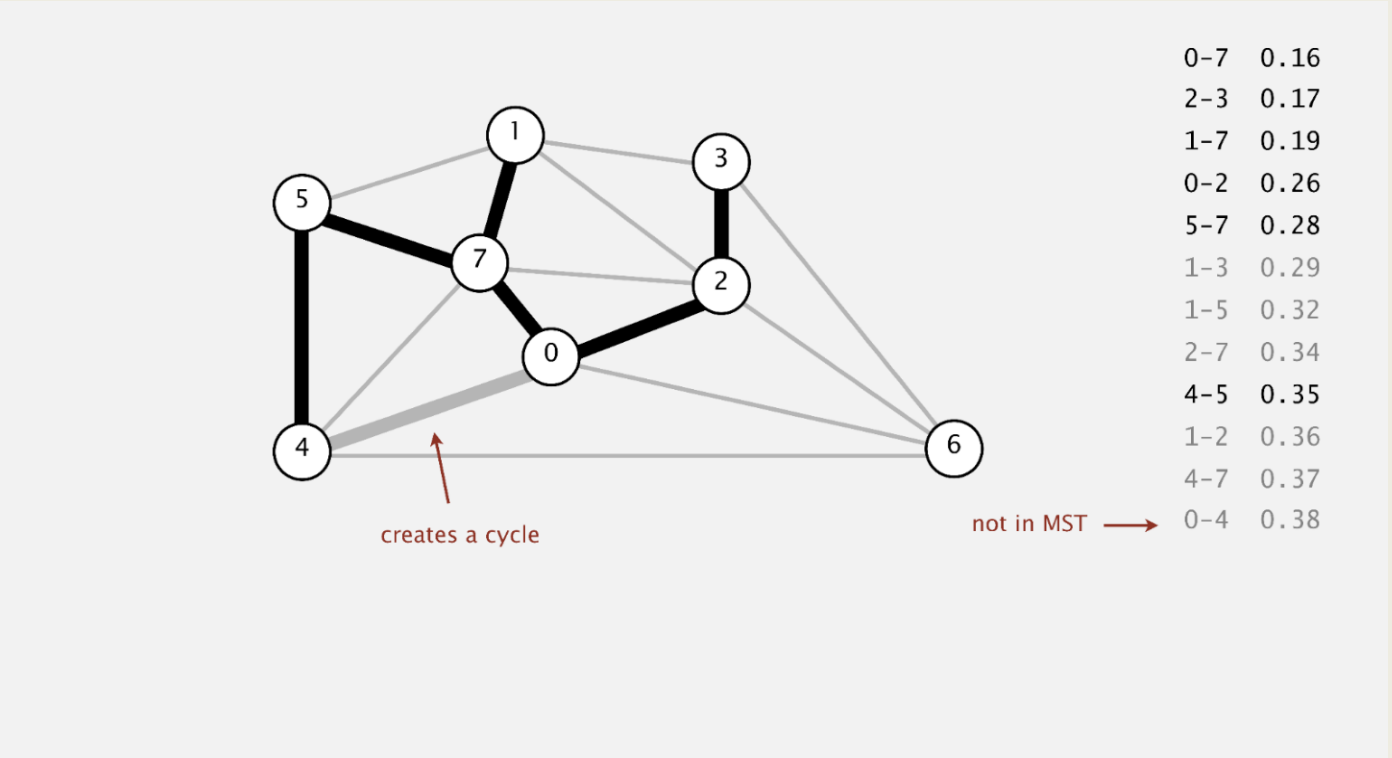
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



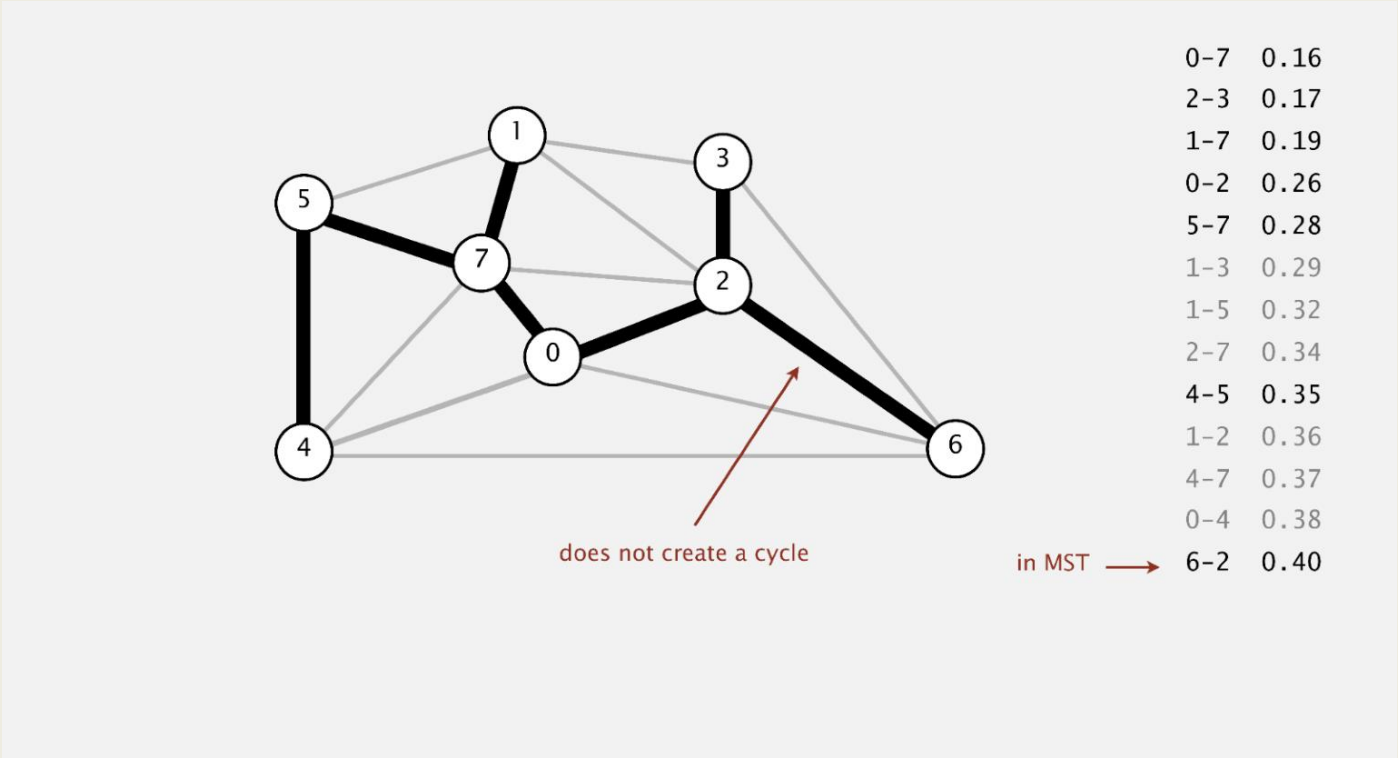
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



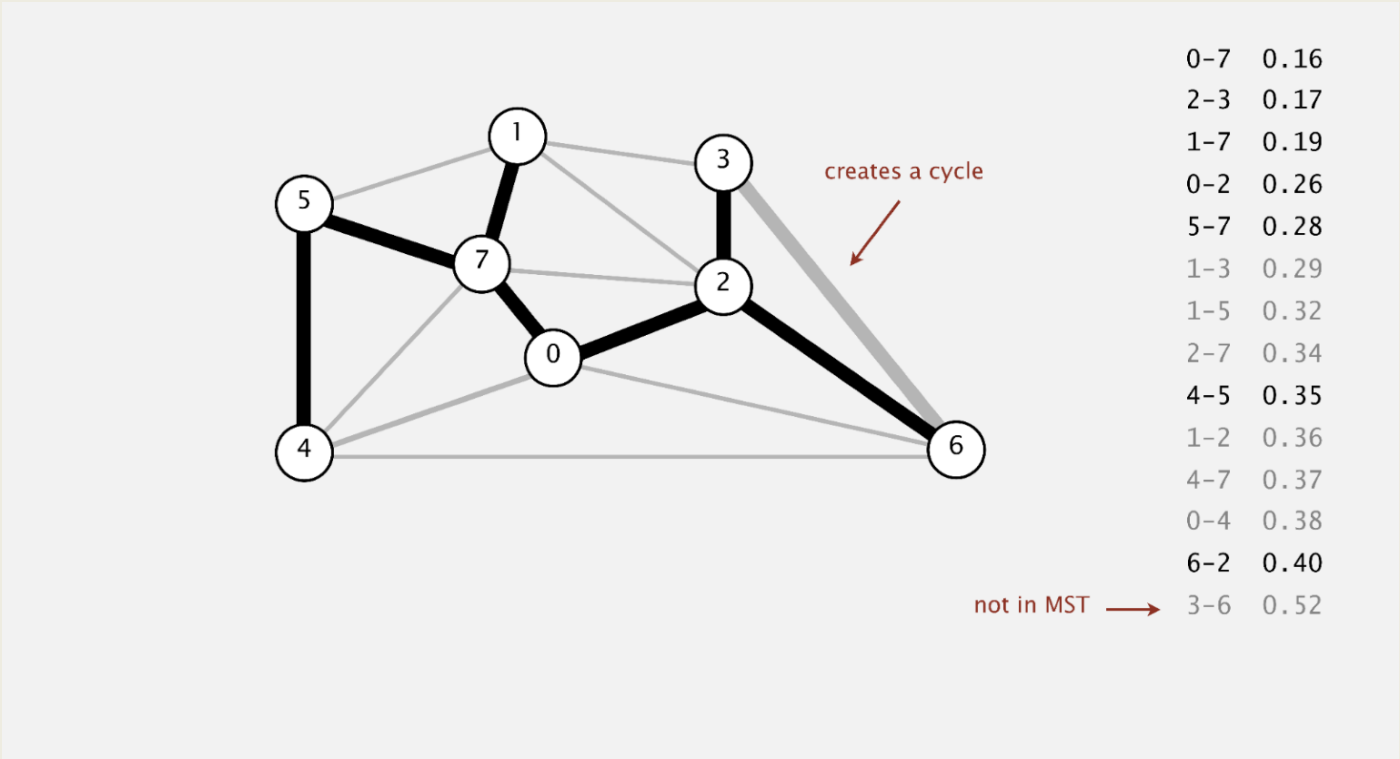
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



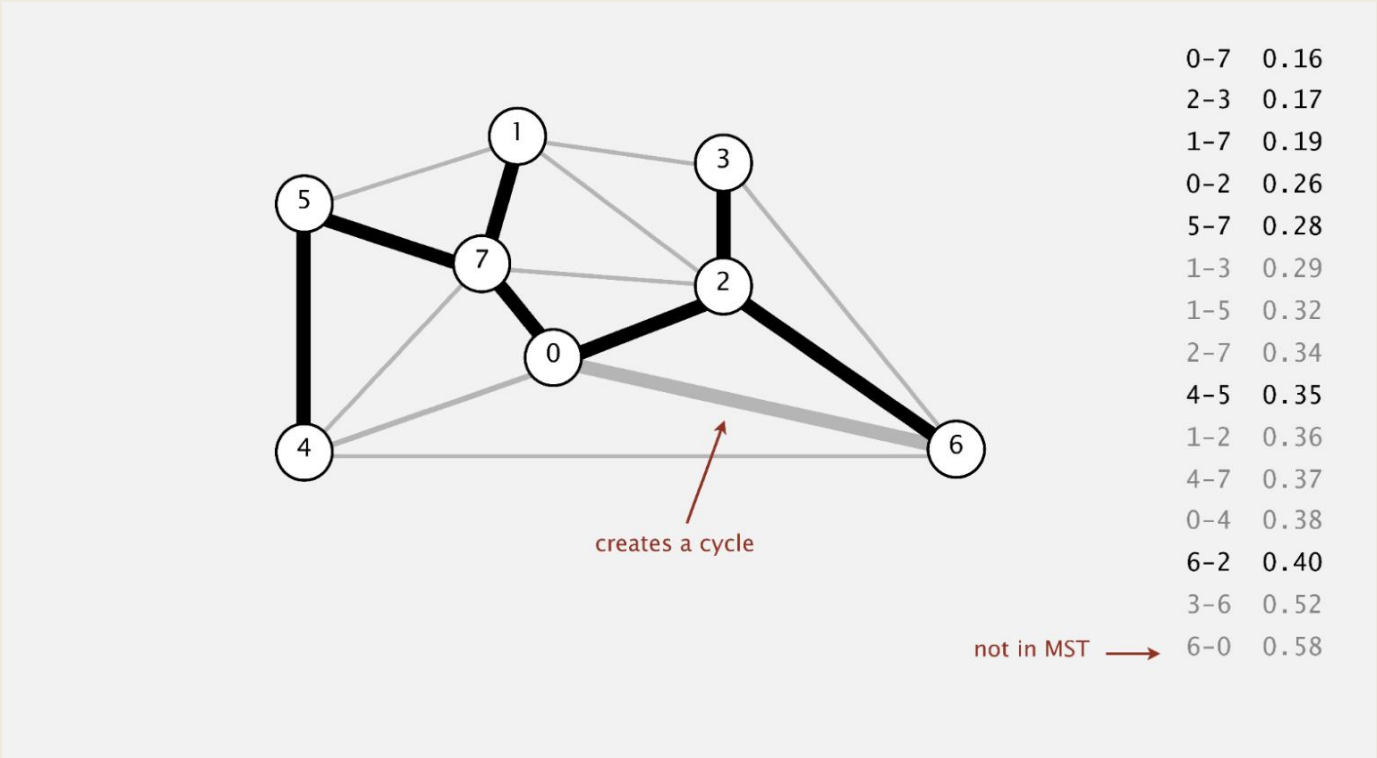
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



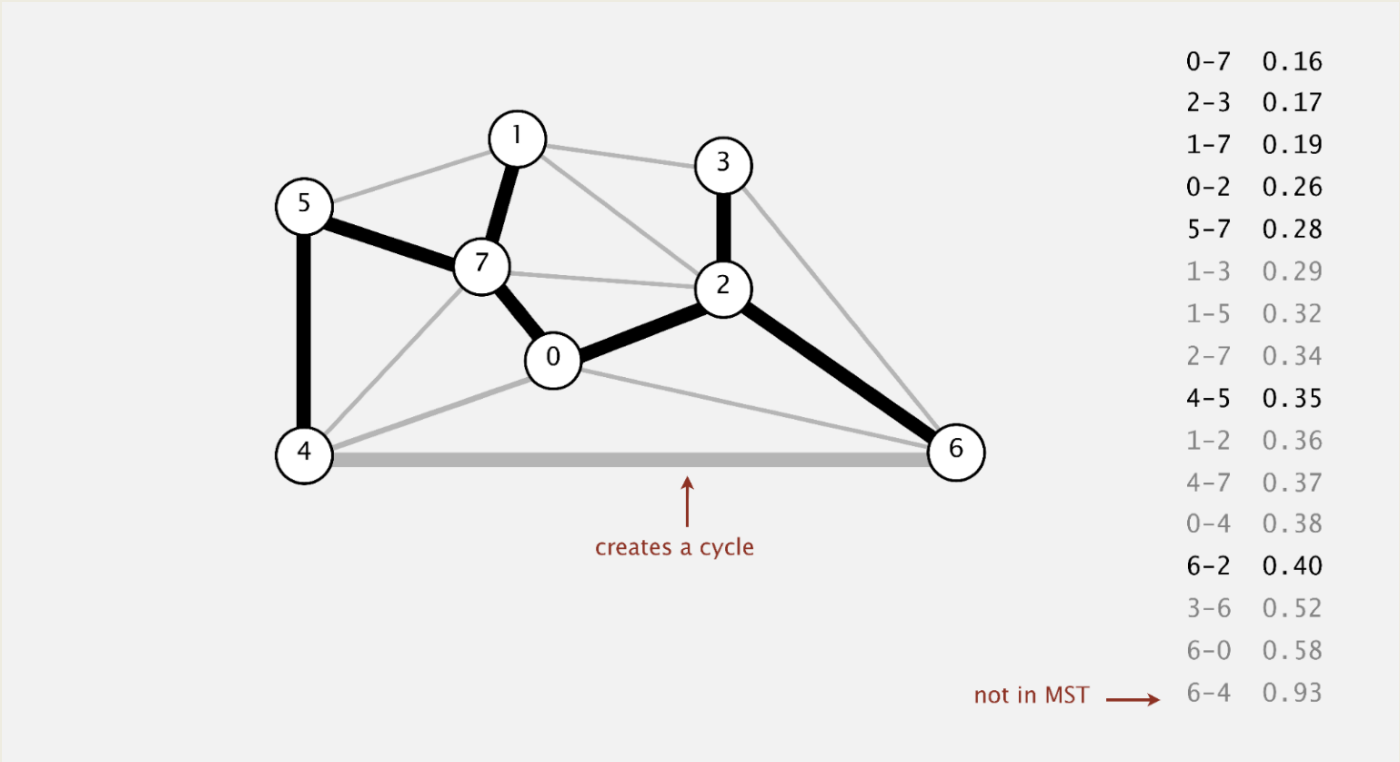
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



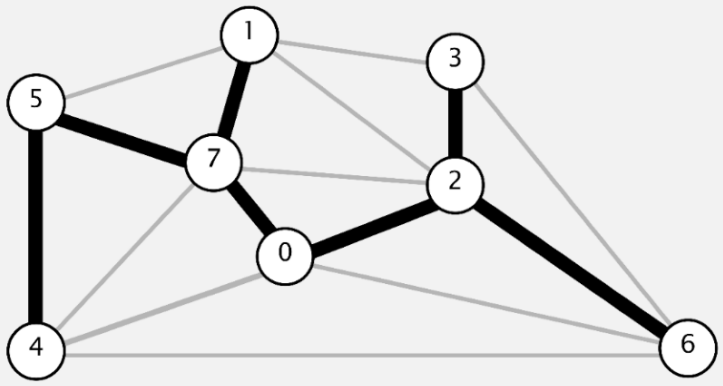
MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.



MINIMUN SPANNING TREE

- **Kruskal's algorithm:** Consider edges in ascending order of weight, add next edge to tree unless it would create a cycle.

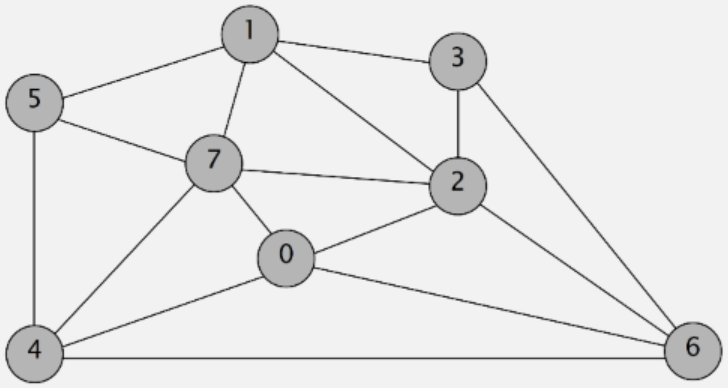


a minimum spanning tree

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.

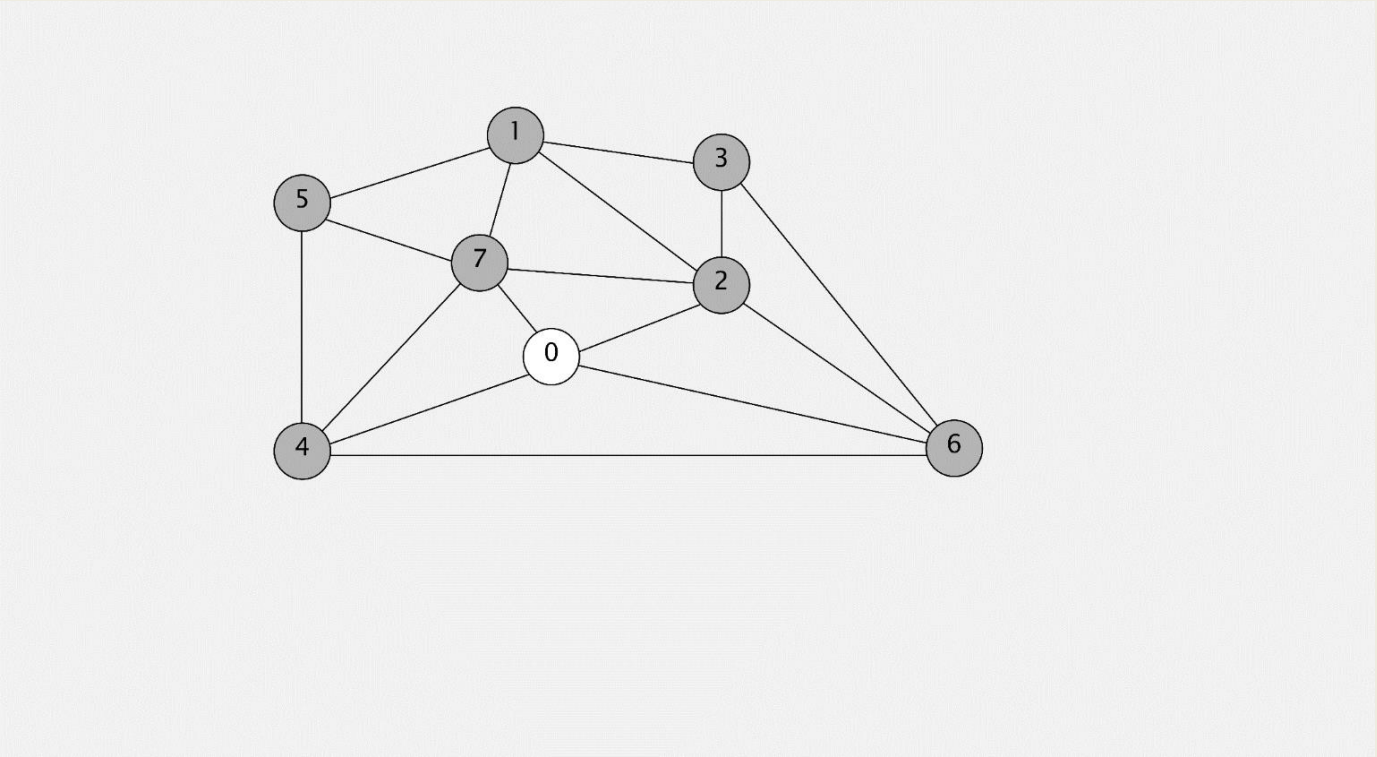


an edge-weighted graph

```
0-7  0.16
2-3  0.17
1-7  0.19
0-2  0.26
5-7  0.28
1-3  0.29
1-5  0.32
2-7  0.34
4-5  0.35
1-2  0.36
4-7  0.37
0-4  0.38
6-2  0.40
3-6  0.52
6-0  0.58
6-4  0.93
```

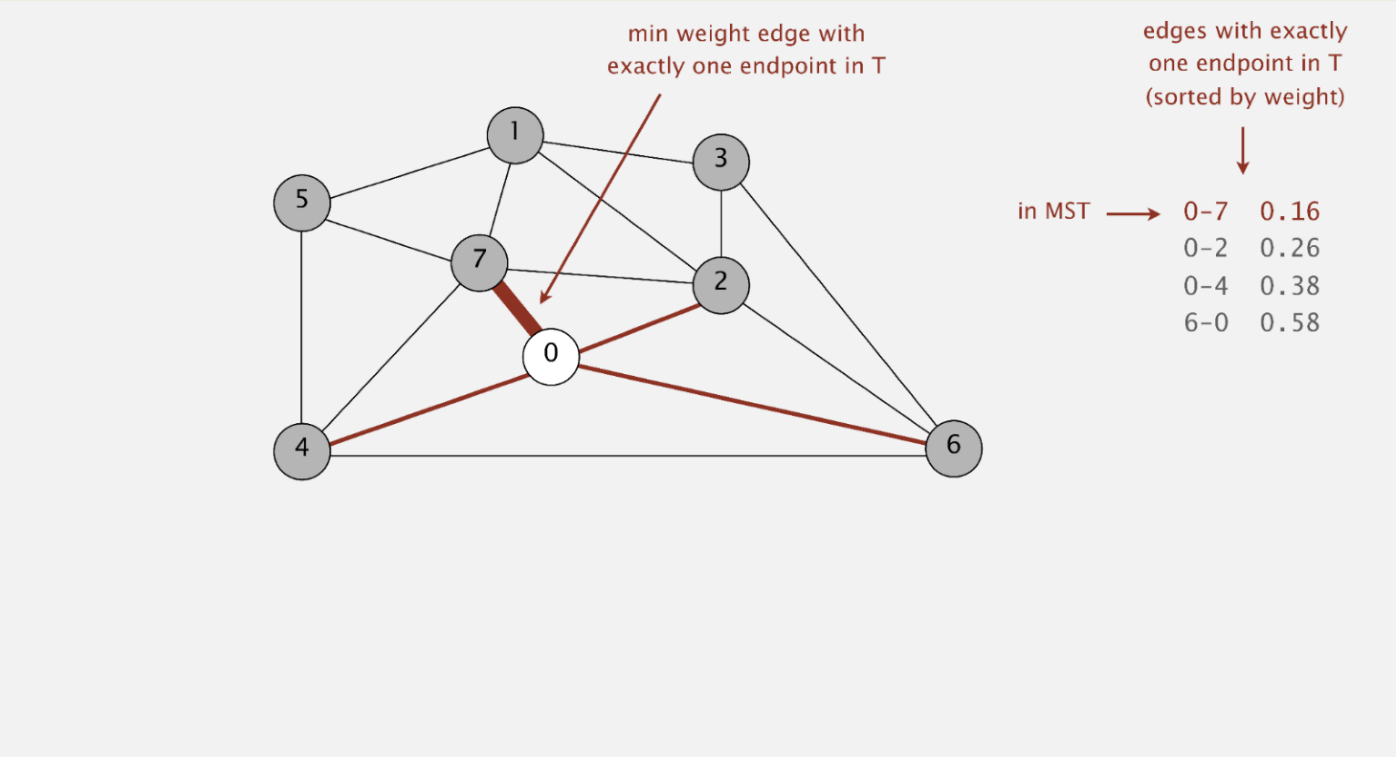
MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.



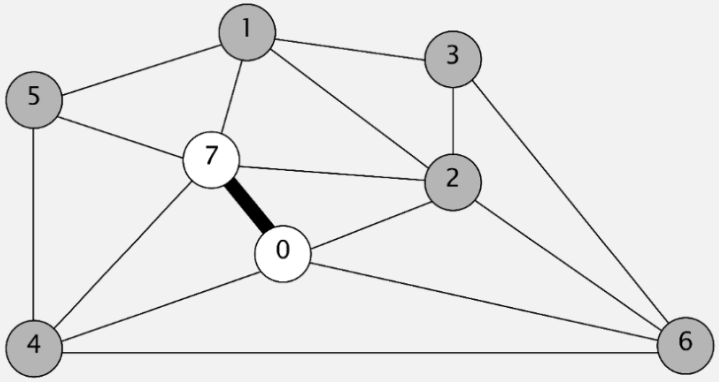
MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.



MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.

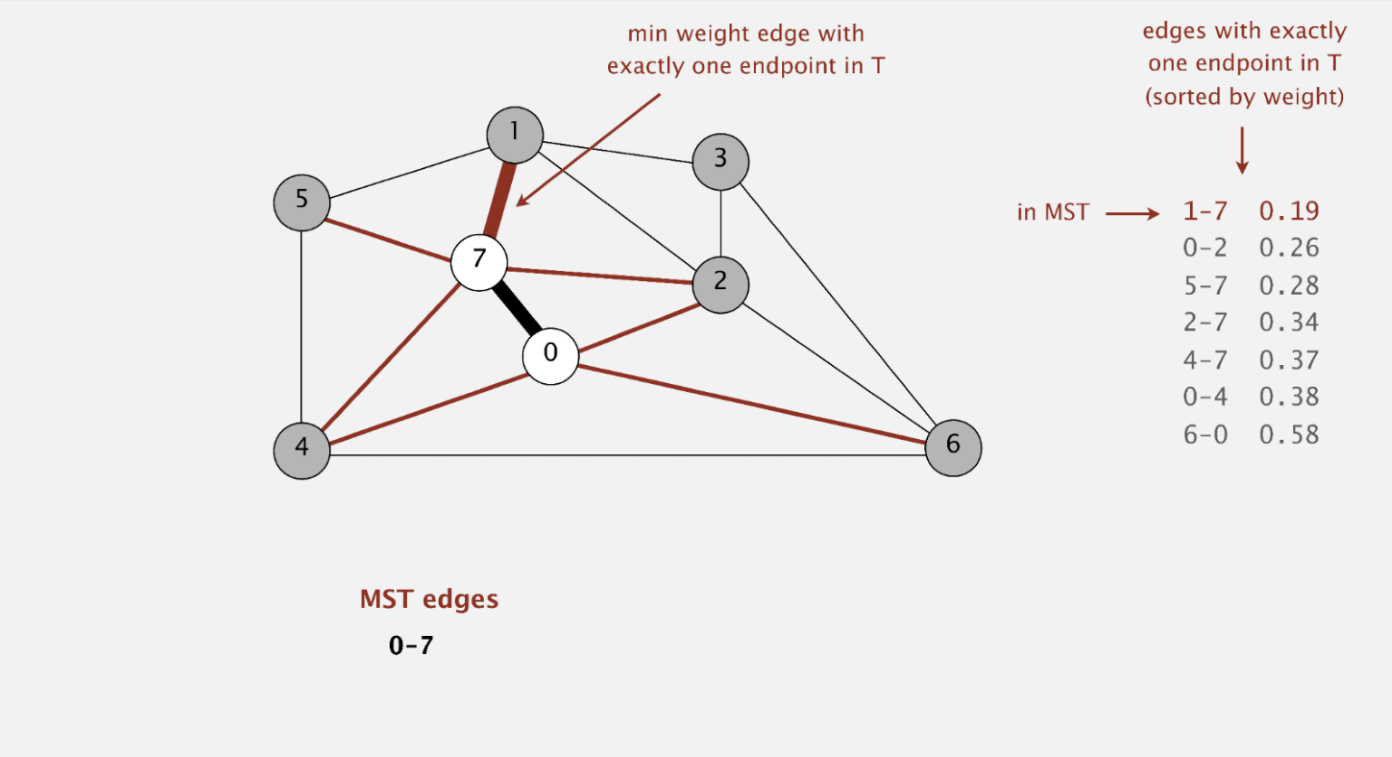


MST edges

0-7

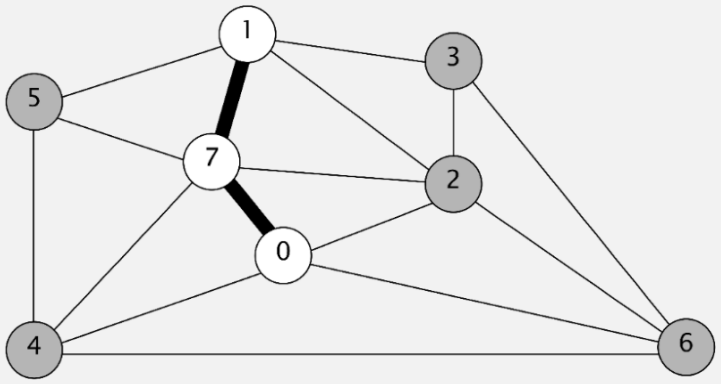
MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.



MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.

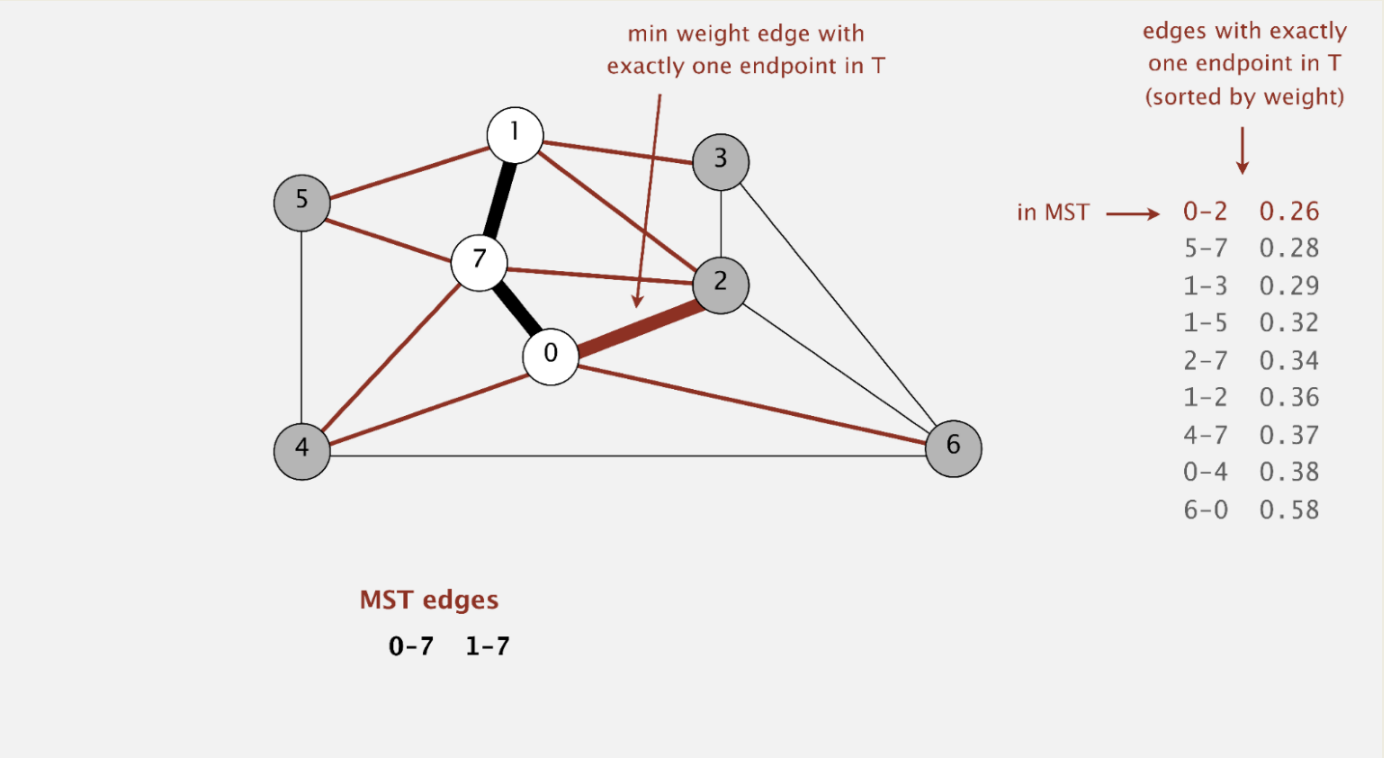


MST edges

0-7 1-7

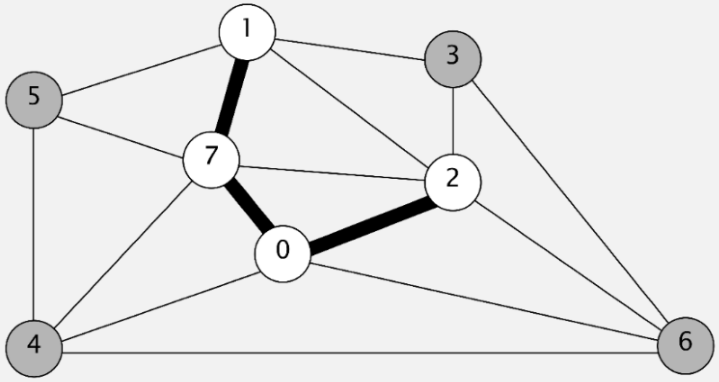
MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.



MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.

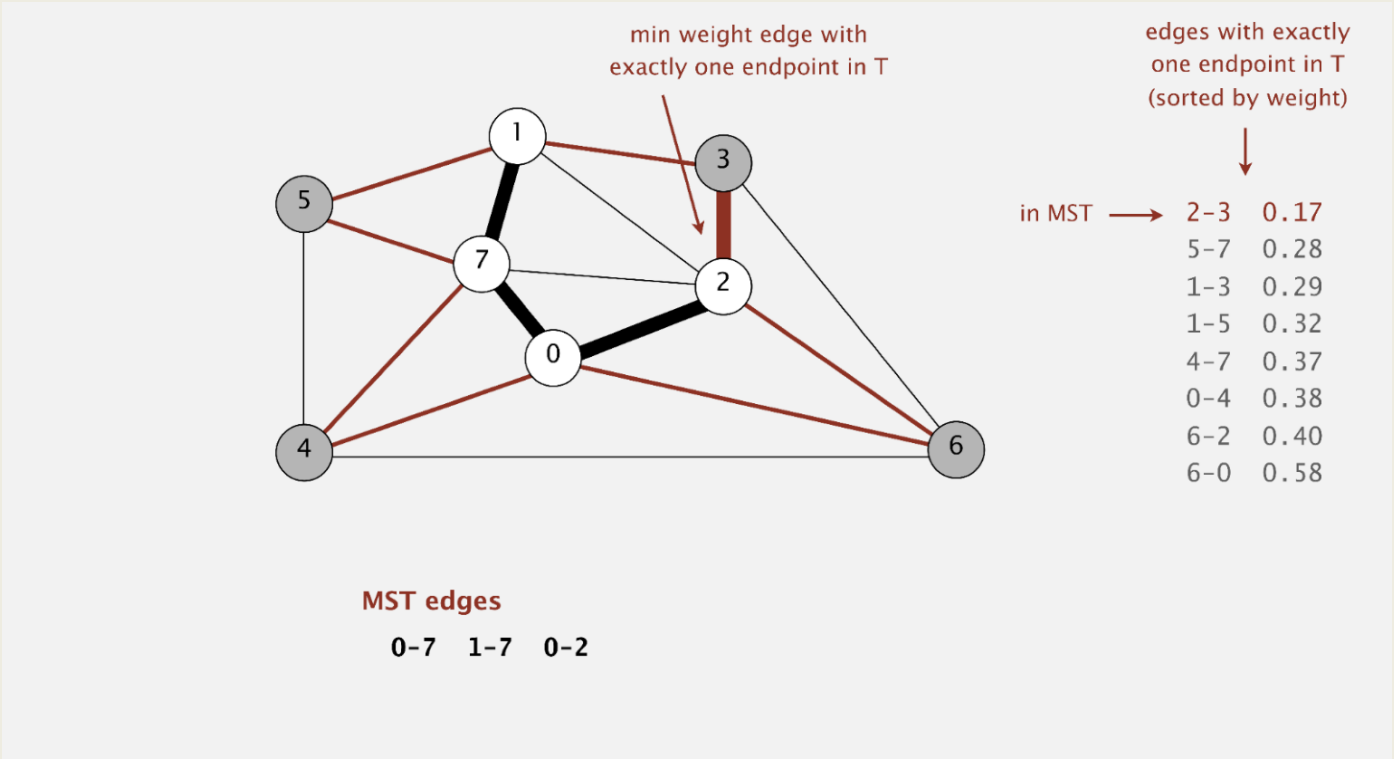


MST edges

0-7 1-7 0-2

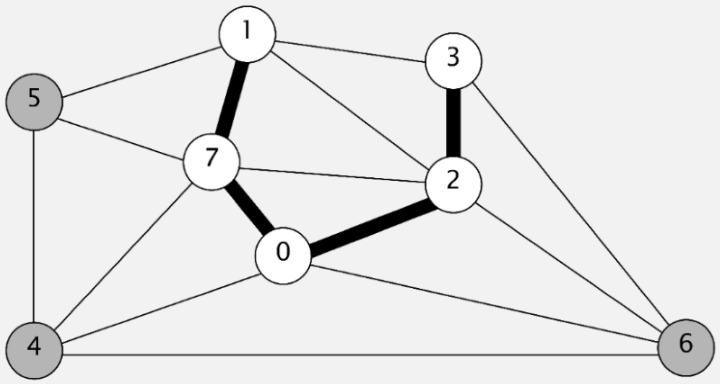
MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.



MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.

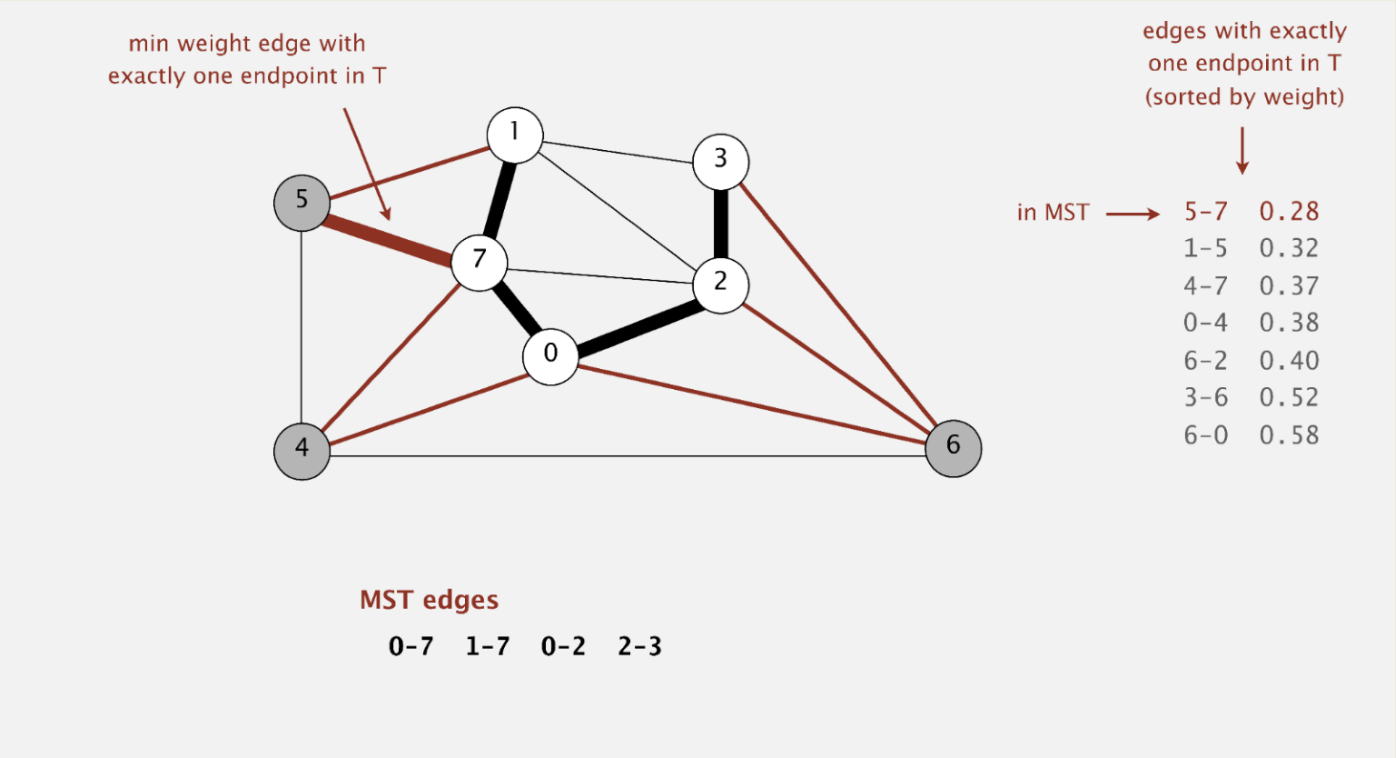


MST edges

0-7 1-7 0-2 2-3

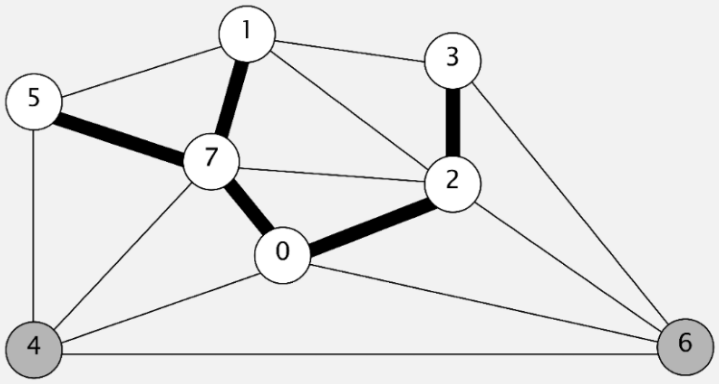
MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.



MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.

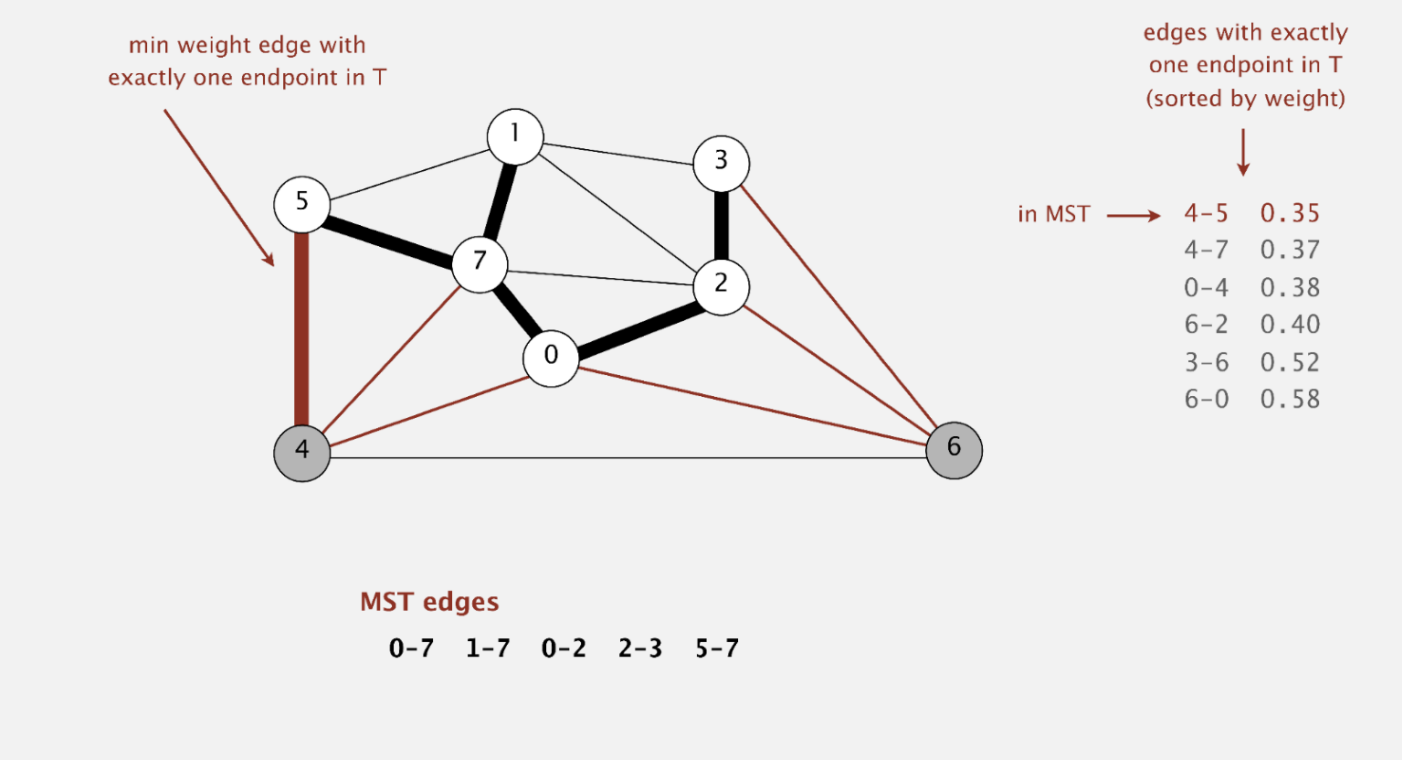


MST edges

0-7 1-7 0-2 2-3 5-7

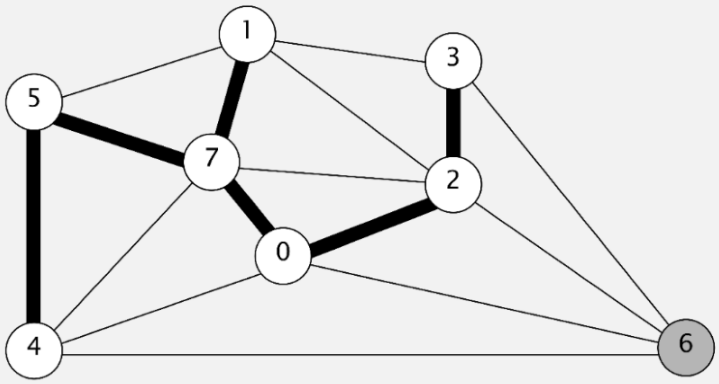
MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.



MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.

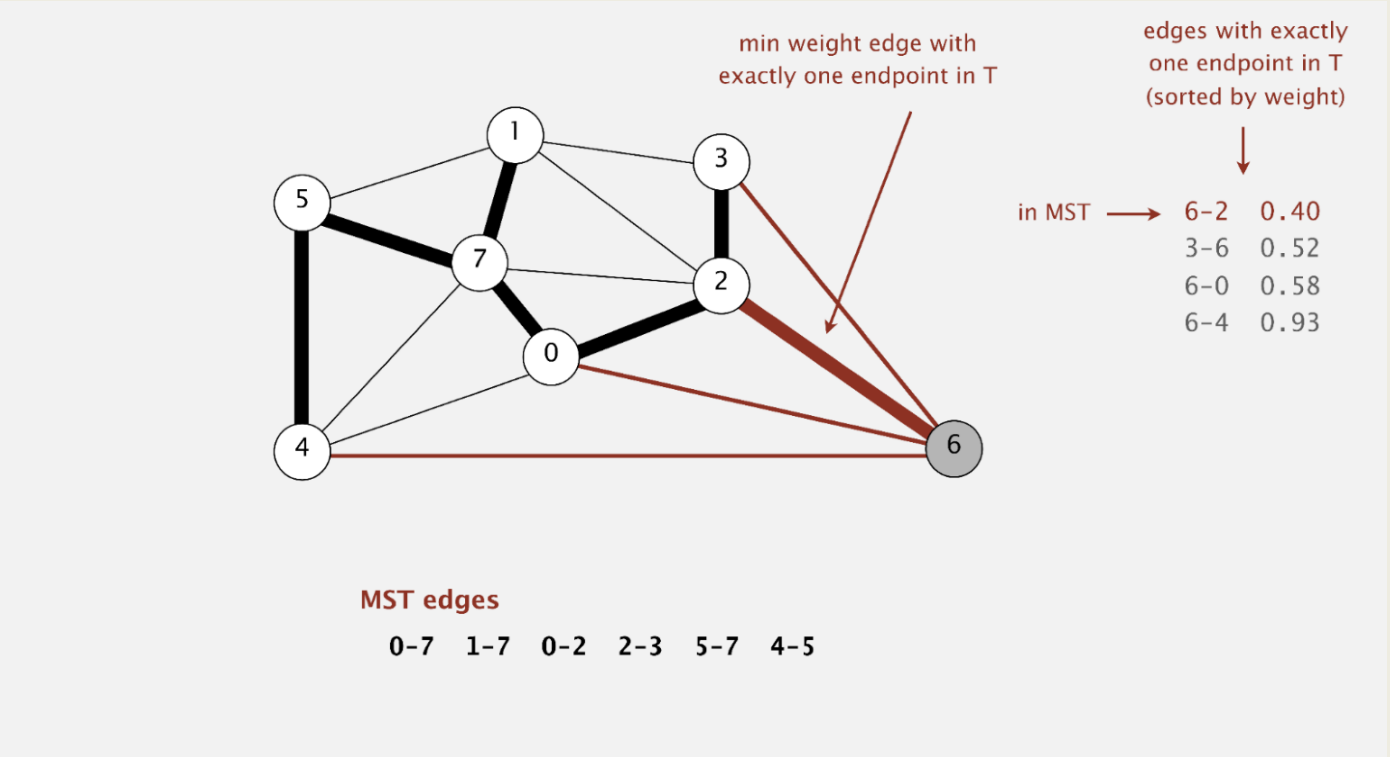


MST edges

0-7 1-7 0-2 2-3 5-7 4-5

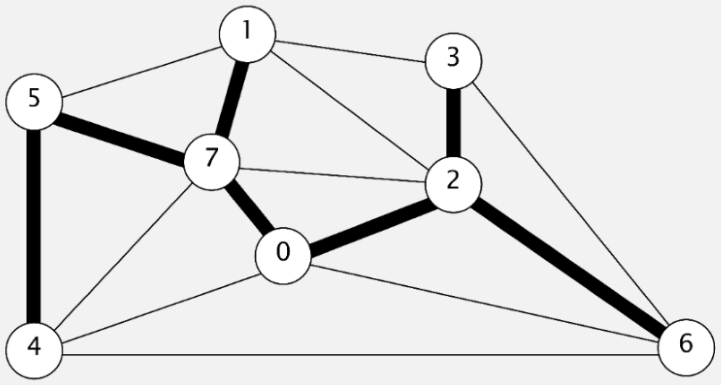
MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.



MINIMUN SPANNING TREE

- **Prim's algorithm:** Start with o vertex, and greedily add to tree the min weight edges, repeat until *V-1* edges are completed.

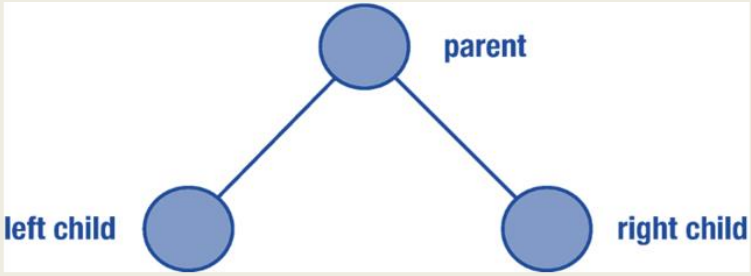


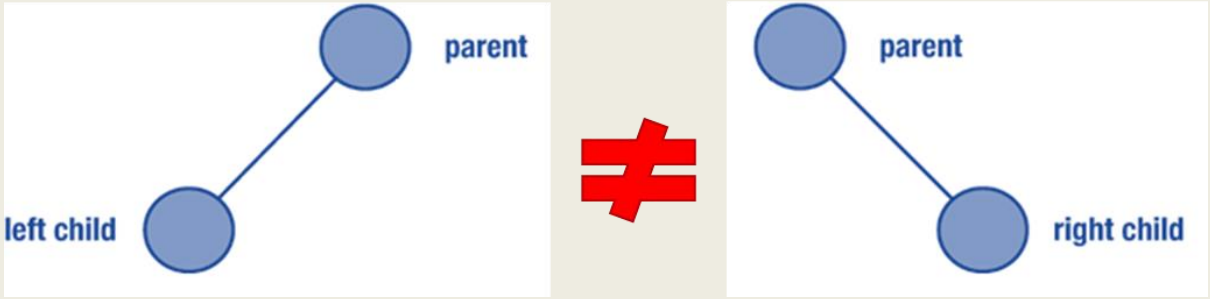
MST edges

0-7 1-7 0-2 2-3 5-7 4-5 6-2

BINARY TREE

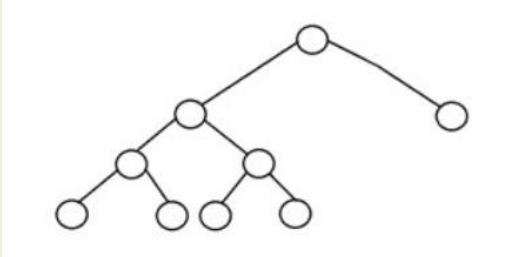
- It is a tree data structure where each node can have only two child nodes.
- Every node in a binary tree (except the root) is either the left or right child of a parent node.



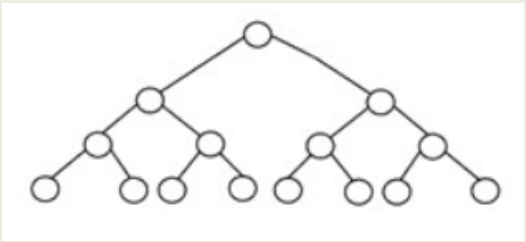


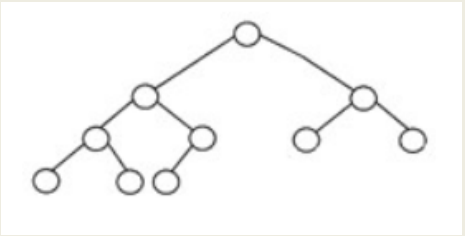
BINARY TREE

- Full Binary Tree:
  - *It is a special kind of a binary tree that each node has either zero children or two children.*

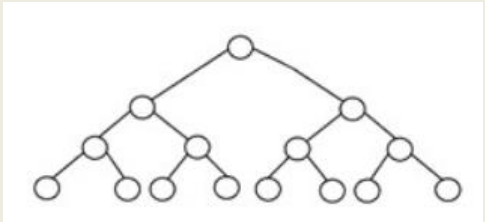


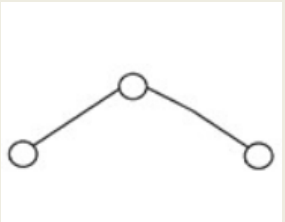
- Complete Binary Tree:
  - *A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side.*



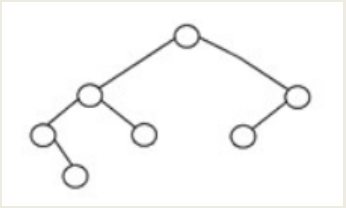


BINARY TREE

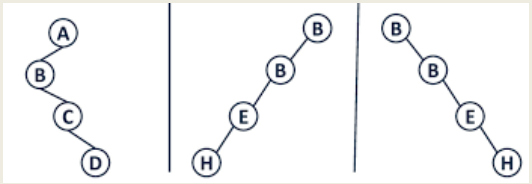




- Perfect Binary Tree:
  - *A binary tree is said to be 'perfect' if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree.*



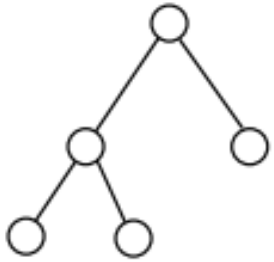
- Balanced Binary Tree:
  - *In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one.*



- Degenerate Binary Tree:
  - *A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child.*

BINARY TREE

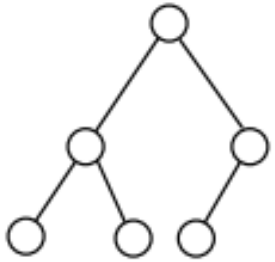
- Quiz:



✗ Perfect

✓ Full

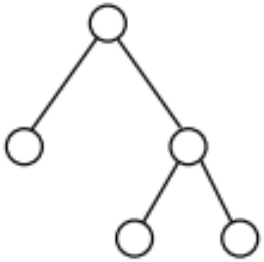
✓ Complete



✓ Complete

✗ Full

✗ Perfect



✓ Full

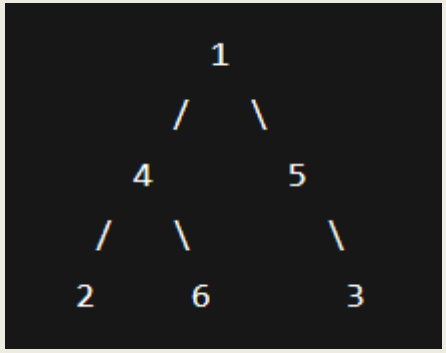
✗ Complete

✗ Perfect

```
class BinaryTree:
    def __init__(self, value):
        self.key = value
        self.left_child = None
        self.right_child = None

    def insert_left(self, value):
        if self.left_child == None:
            self.left_child = BinaryTree(value)
        else:
            bin_tree = BinaryTree(value)
            bin_tree.left_child = self.left_child
            self.left_child = bin_tree

    def insert_right(self, value):
        if self.right_child == None:
            self.right_child = BinaryTree(value)
        else:
            bin_tree = BinaryTree(value)
            bin_tree.right_child = self.right_child
            self.right_child = bin_tree
```

BINARY TREE

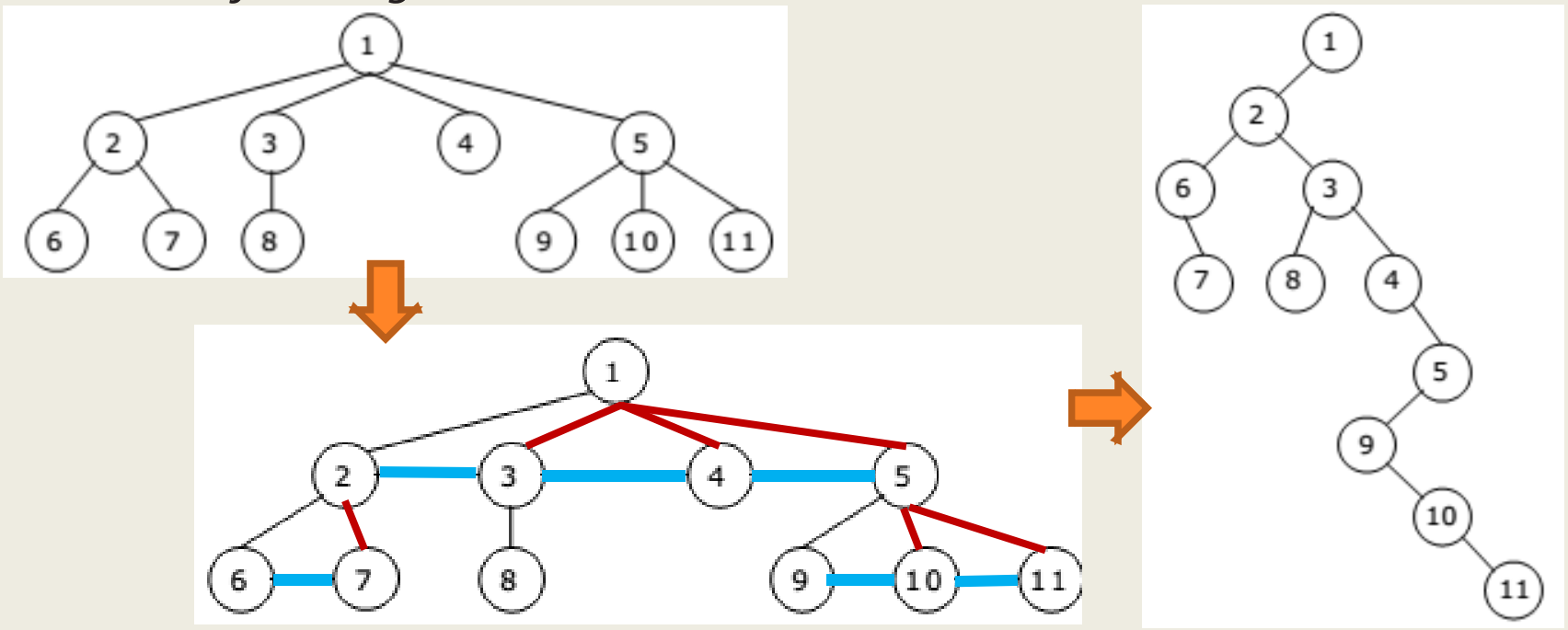
```
tree = BinaryTree(1)
tree.insert_left(2)
tree.insert_right(3)
tree.insert_left(4)
tree.left_child.insert_right(6)
tree.insert_right(5)


print(tree.breadth_first_search(4))

print("\nPreorder:")
preorder(tree)

print("\n\nPostorder:")
postorder(tree)

print("\n\nInorder:")
inorder(tree)
```




```
True

Preorder:
1 4 2 6 5 3

Postorder:
2 6 4 3 5 1

Inorder:
2 4 6 1 5 3
```

CONVERTING ANY GENERAL TREE TO A BINARY TREE

1. Insert an edge connecting all the sibling of the given tree.
2. Delete all the parent edges from the parent to its children except the one to its leftmost child …
   - *(Rotate the resulting tree by 45 degress to differentiate between its left and right subtree.)*

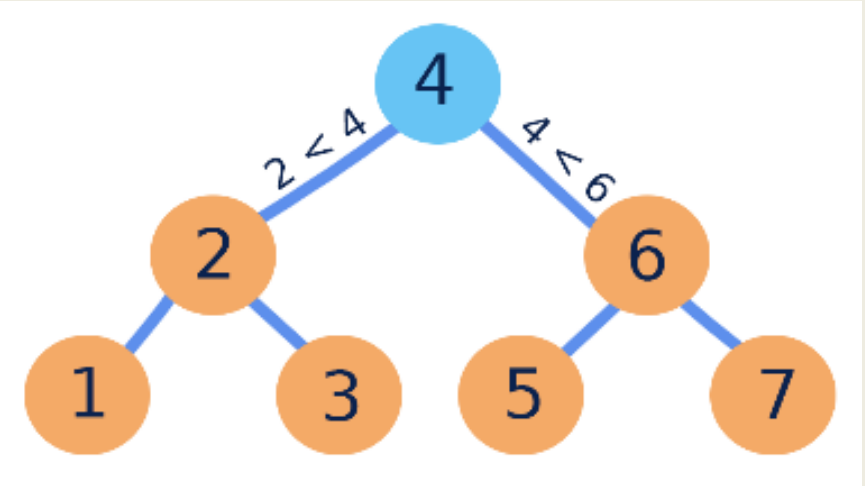


BINARY SEARCH TREE

- It is a tree data structure where each node can have only two children.
- The tree stores its nodes in sorted order where:

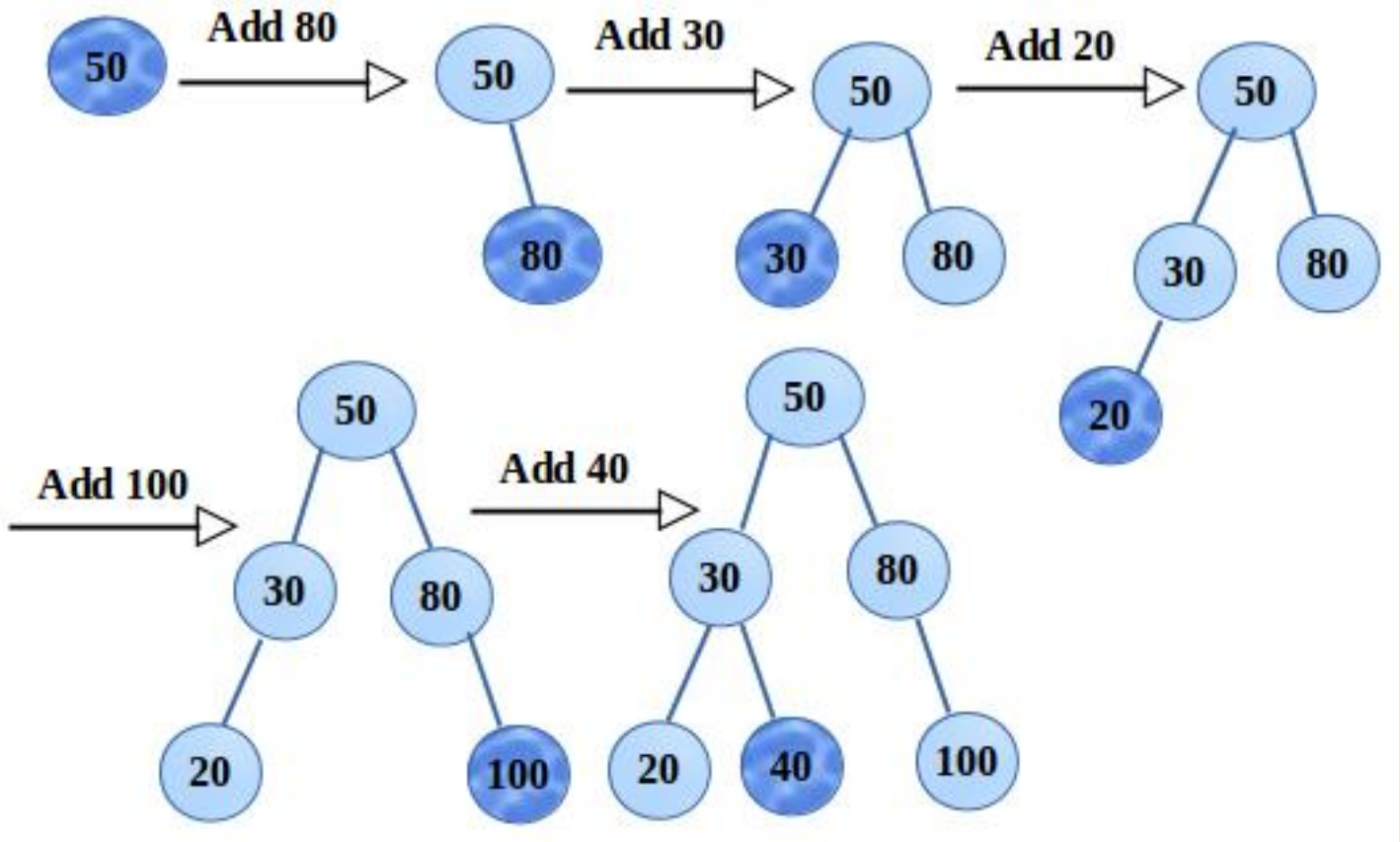
  every node's value is greater than any value in its left subtree and lower than any value in its right subtree

- Like hash table keys, you cannot store duplicate values in a binary search tree. You can get around this restriction and handle duplicate values by adding a count field in your tree's node objects to track the number of occurrences of a given value.



BINARY SEARCH TREE

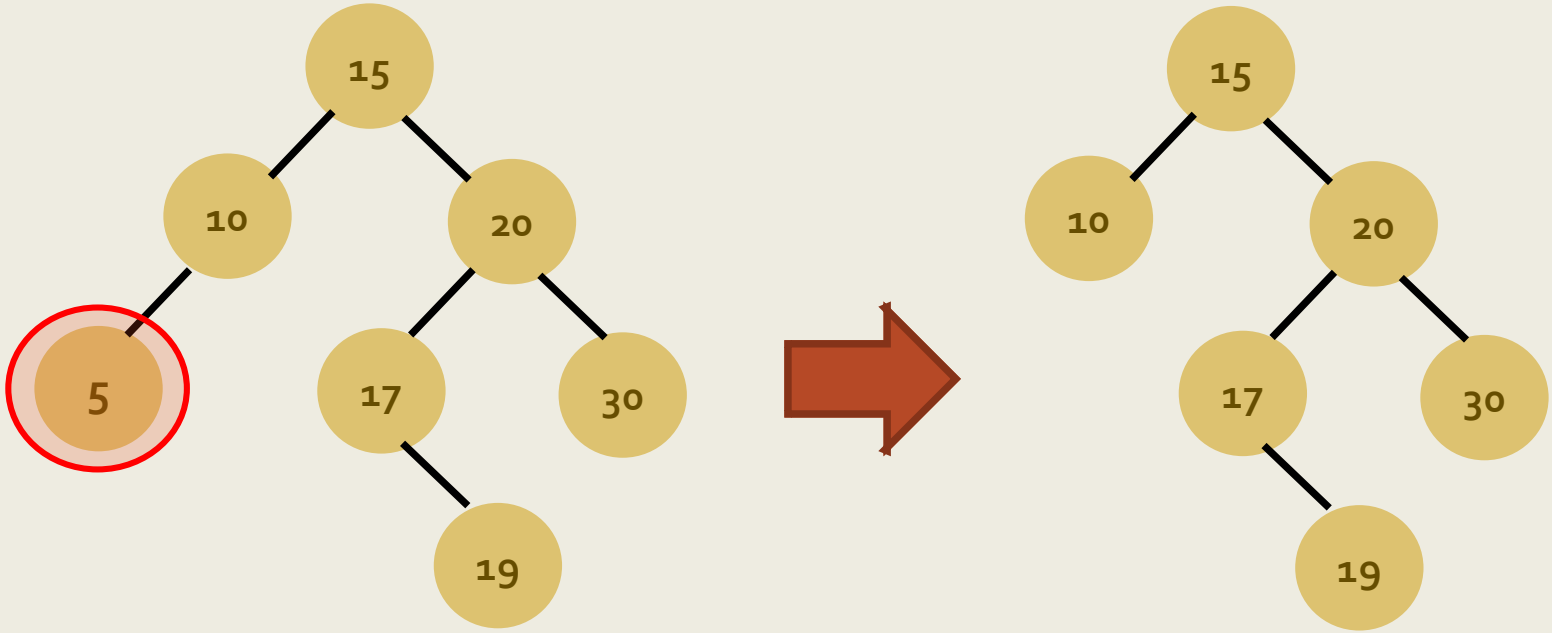
- Inserting:



BINARY SEARCH TREE

- When deleting a node from a binary search tree it is mandatory to maintain the in-order sequence of the nodes. There are three possible cases to consider:

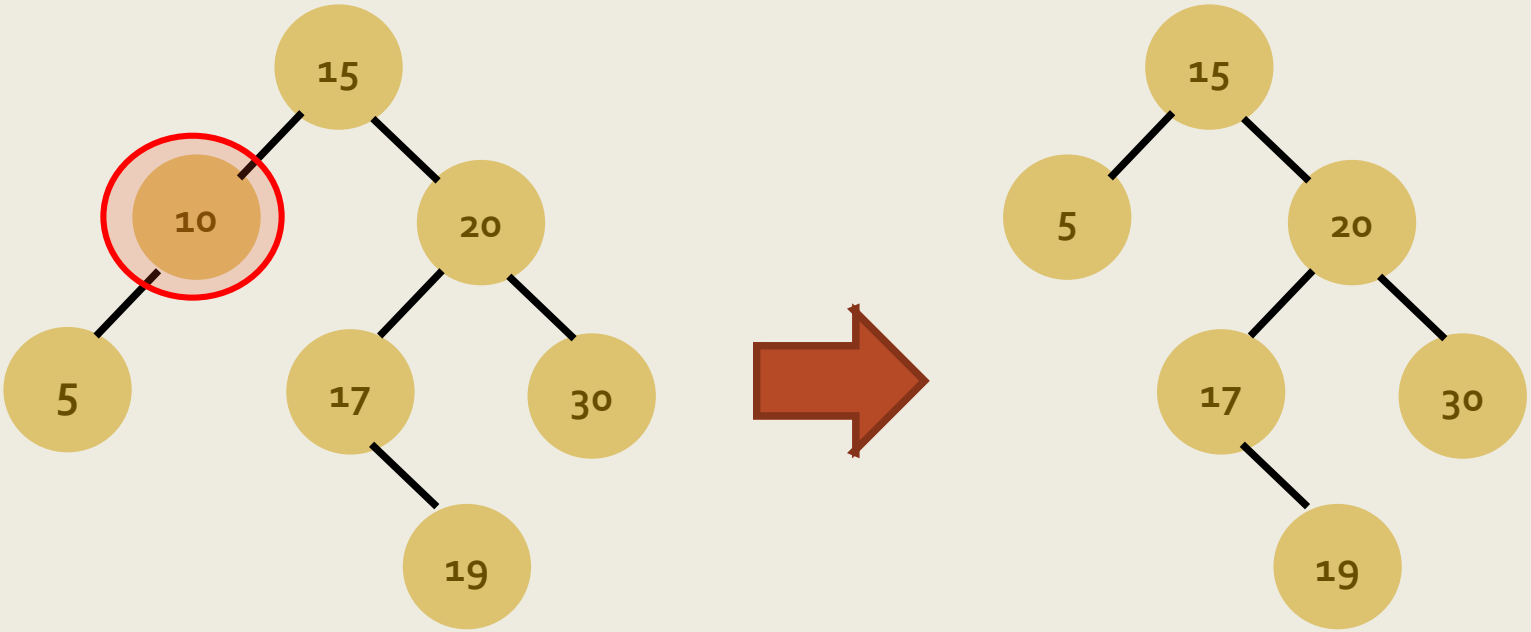
  1- Deleting a node with no children: simply remove the node from the tree.



BINARY SEARCH TREE

- When deleting a node from a binary search tree it is mandatory to maintain the in-order sequence of the nodes. There are three possible cases to consider:

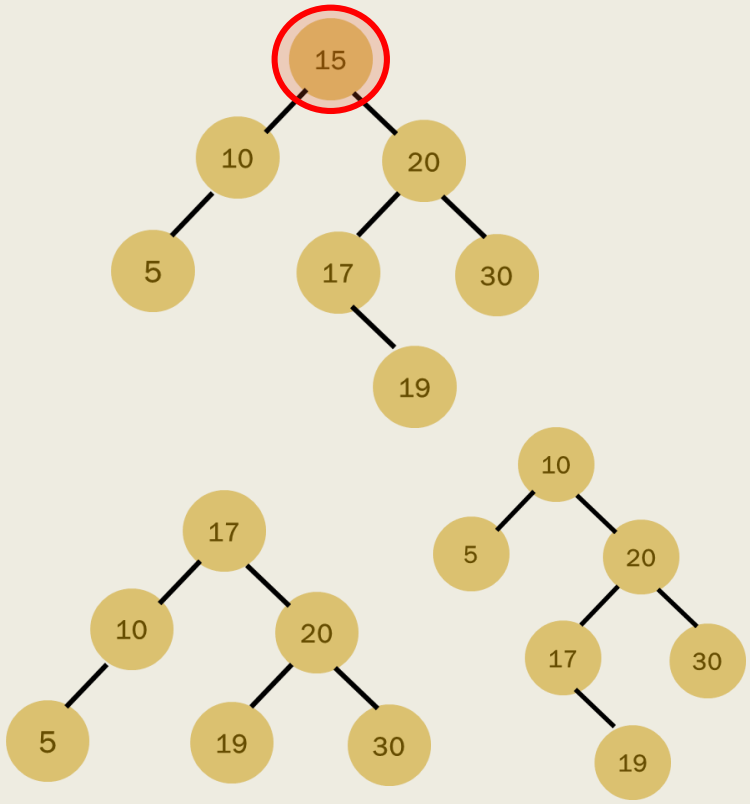
  2- Deleting a node with one child: remove the node and replace it with its child.



BINARY SEARCH TREE

- When deleting a node from a binary search tree it is mandatory to maintain the in-order sequence of the nodes. There are three possible cases to consider:

3- Deleting a node with two children: call the node to be deleted D, do not delete D. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node (E). Copy the value of E to D. If E does not have a child simply remove E from its previous parent G. If E has a child, say F, it is a right child (because it is a successor or minimum larger number, it won't have a left child or smaller number). Replace E with F at E's parent.



BINARY SEARCH TREE

- Inserting, deleting, and searching for data in a general tree are all O(n).
- Binary search trees are more efficient:

  all three operations in a binary search tree are logarithmic because you can do a binary search to insert, delete, and search for nodes in a binary search tree.

- Trees enable you to store **hierarchical** information.

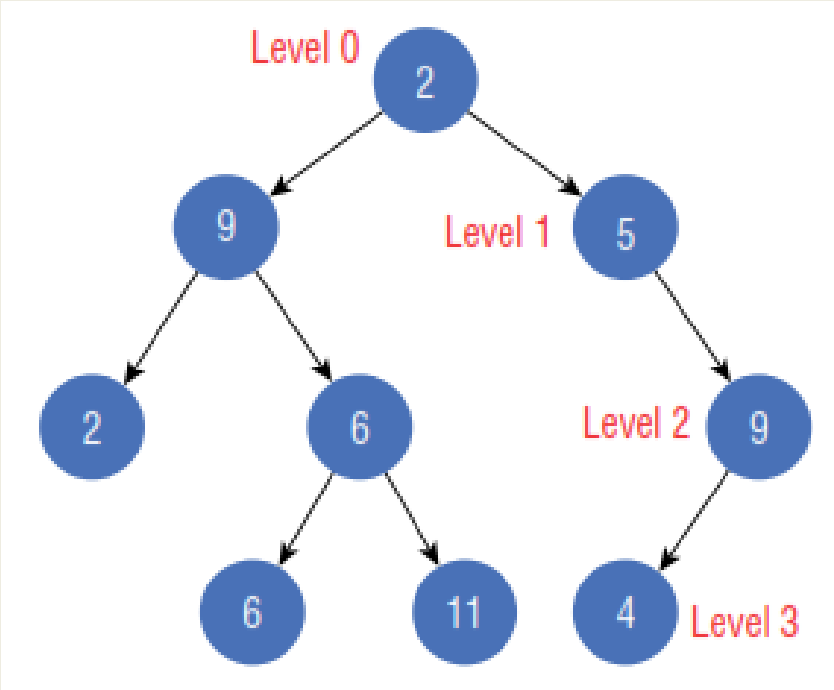
Advantages of BST over HT:
- memory use
- allowing quick traverse by keeping the data's order

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

BREADTH-FIRST TREE TRAVERSAL

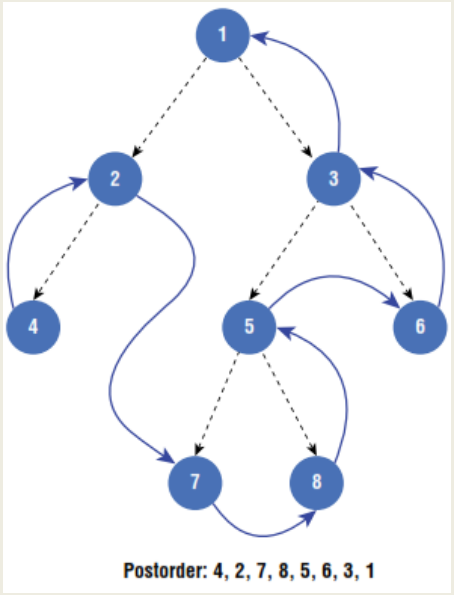
- It is a method of visiting every node in a tree by visiting each node in it level by level.

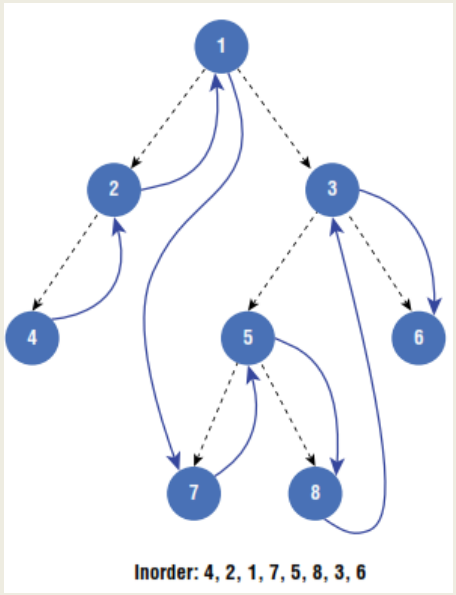
2, 9, 5, 2, 6, 9, 6, 11, 4

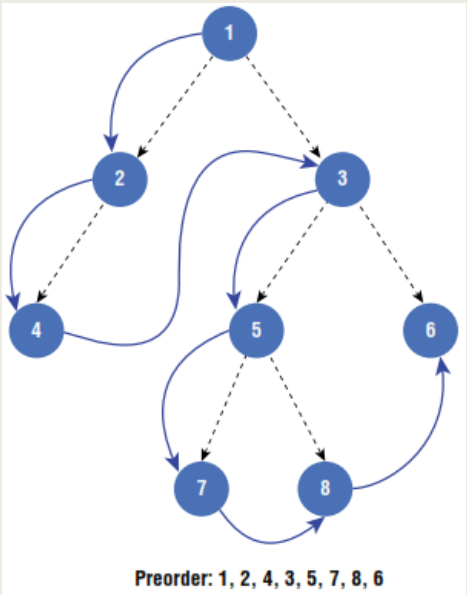


DEPTH-FIRST TREE TRAVERSAL

- This method visit all the nodes in a binary tree by going as deep as you can in one direction before moving to the next sibling.
- offers three ways to visit every node:
  - *preorder,*
  - *postorder,*
  - *in order.*

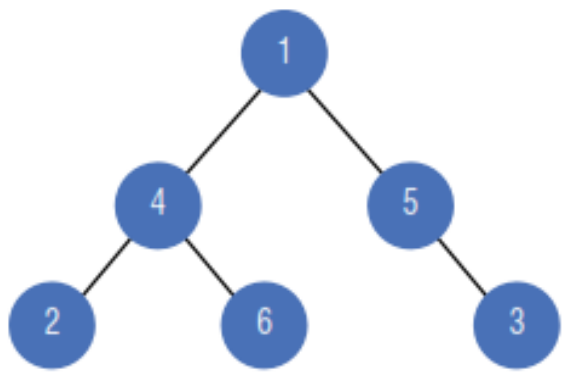


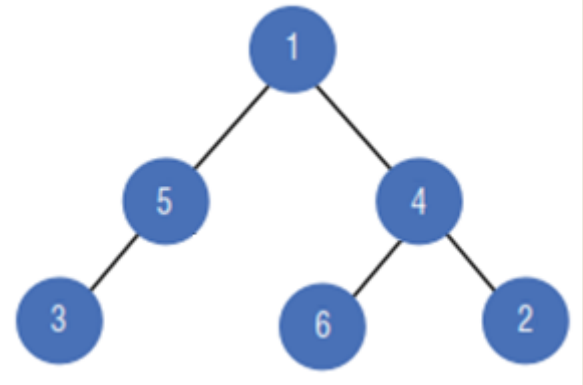




INVERT A BINARY TREE

- Challenge: swapping all the nodes in a binary tree. In other words, every right node becomes a left node, and every left node becomes a right node.





```
    def breadth_first_search(self, n):
        current = [self]
        next = []
        while current:
            for node in current:
                if node.key == n:
                    return True
                if node.left_child:
                    next.append(node.left_child)
                if node.right_child:
                    next.append(node.right_child)
            current = next
            next = []
        return False

    def invert(self):
        current = [self]
        next = []
        while current:
            for node in current:
                if node.left_child:
                    next.append(node.left_child)
                if node.right_child:
                    next.append(node.right_child)
                tmp = node.left_child
                node.left_child = node.right_child
                node.right_child = tmp
            current = next
            next = []
```

BINARY TREE

```
def preorder(tree):
    if tree:
        print(tree.key, end=" ")
        preorder(tree.left_child)
        preorder(tree.right_child)

def postorder(tree):
    if tree:
        postorder(tree.left_child)
        postorder(tree.right_child)
        print(tree.key, end=" ")

def inorder(tree):
    if tree:
        inorder (tree.left_child)
        print(tree.key, end=" ")
        inorder (tree.right_child)
```