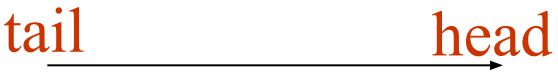
Fundaments of Algorithm

SYBSC (Computer Science)

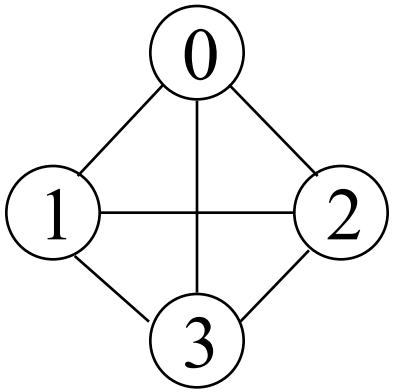
UNIT III

Definition

- A graph G consists of two sets
  - a finite, nonempty set of vertices V(G)
  - a finite, possible empty set of edges E(G)
  - G(V,E) represents a graph
- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $\langle v_0, v_1 \rangle$!= $\langle v_1, v_0 \rangle$

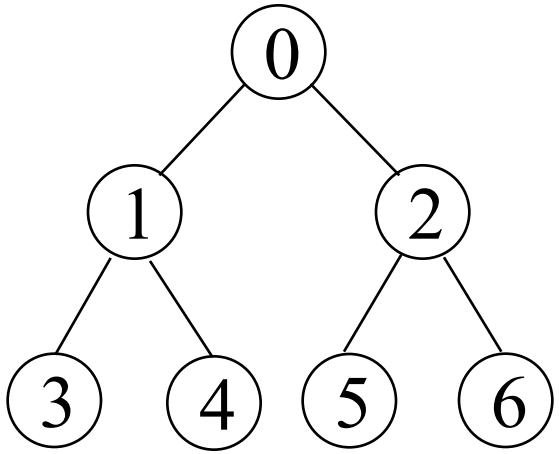


Examples for Graph



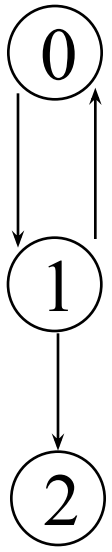
G_1

complete graph



G_2

incomplete graph



G_3

$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

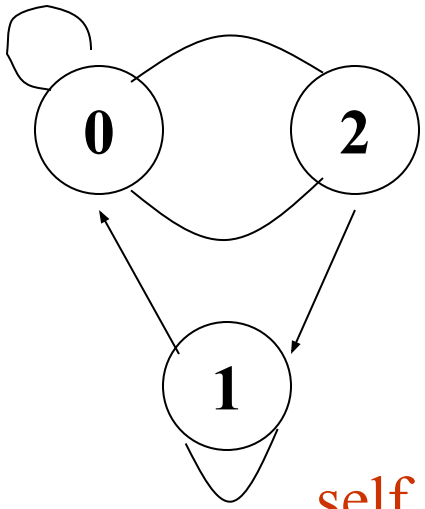
complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

Complete Graph

- A complete graph is a graph that has the maximum number of edges
  - for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
  - for directed graph with n vertices, the maximum number of edges is n(n-1)
  - example: G1 is a complete graph

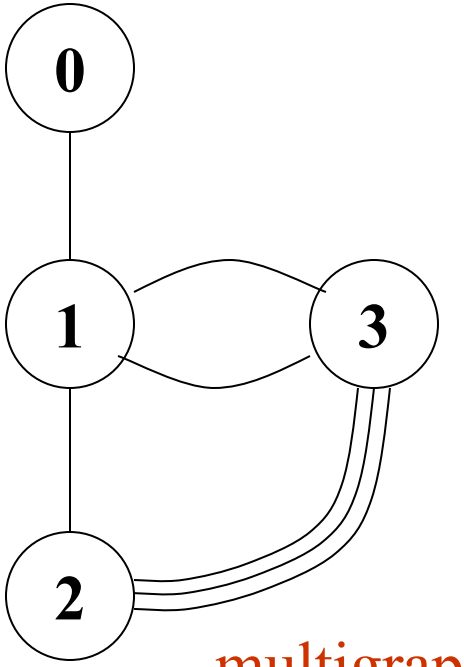
Adjacent and Incident

- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1



self edge

(a)

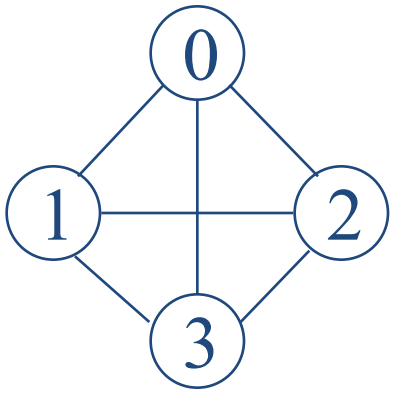


(b)

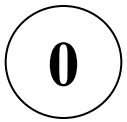
multigraph:
multiple occurrences
of the same edge

Subgraph and Path

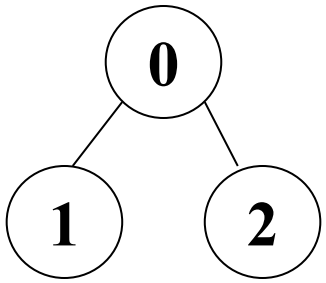
- A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph
- The length of a path is the number of edges on it



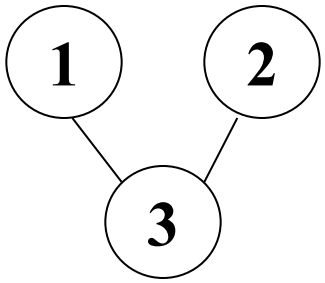
G_1



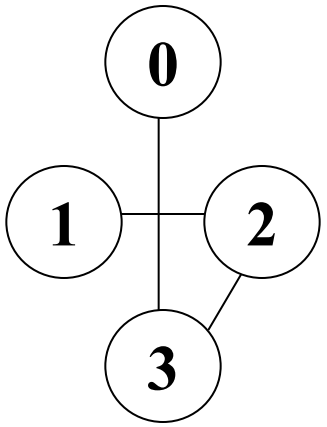
(i)



(ii)



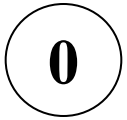
(iii)



(iv)

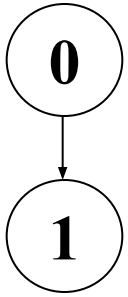
(a) Some of the subgraph of G_1

G_3

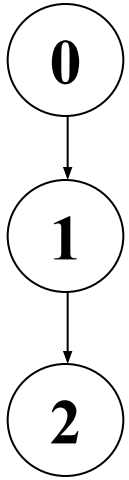


單一

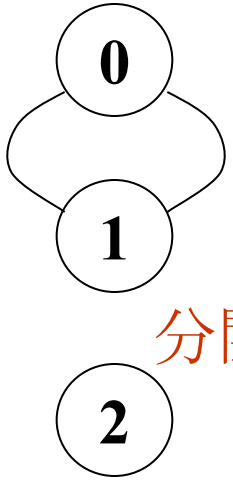
(i)



(ii)



(iii)



分開

(iv)

(b) Some of the subgraph of G_3

Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, two vertices, v_0 and v_1, are connected if there is a path in G from v_0 to v_1
- An undirected graph is connected if, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j

Applications of Breadth First Traversal

1) Shortest Path and Minimum Spanning Tree for unweighted graph In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) Crawlers in Search Engines: Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.
6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle.
We can use BFS to detect cycle in a directed graph also,

Path Finding
We can specialize the DFS algorithm to find a path between two given vertices u and z.

Topological Sorting
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers .

Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based algo for finding Strongly Connected Components)

Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

```
BFS (G, s)
//Where G is the graph and s is the source node
let Q be queue.
Q.enqueue( s )
//Inserting s in queue until all its neighbour vertices are marked.
mark s as visited.
while ( Q is not empty)
//Removing that vertex from queue,whose neighbour will be visited now
v = Q.dequeue( )
//processing all the neighbours of v
for all neighbours w of v in Graph G
if w is not visited Q.enqueue( w )
//Stores w in Q to further visit its neighbour
mark w as visited.
```

```
DFS-iterative (G, s):
//Where G is graph and s is source vertex
let S be stack S.push( s )
//Inserting s in stack
mark s as visited.
while ( S is not empty):
//Pop a vertex from stack to visit next
v = S.top( ) S.pop( )
//Push all the neighbours of v in stack that are not visited
for all neighbours w of v in Graph G:
if w is not visited :
S.push( w )
mark w as visited
DFS-recursive(G, s):
mark s as visited
for all neighbours w of s in Graph G:
if w is not visited:
DFS-recursive(G, w)
```

Topological Sorting

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers .

To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based algo for finding Strongly Connected Components)

Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

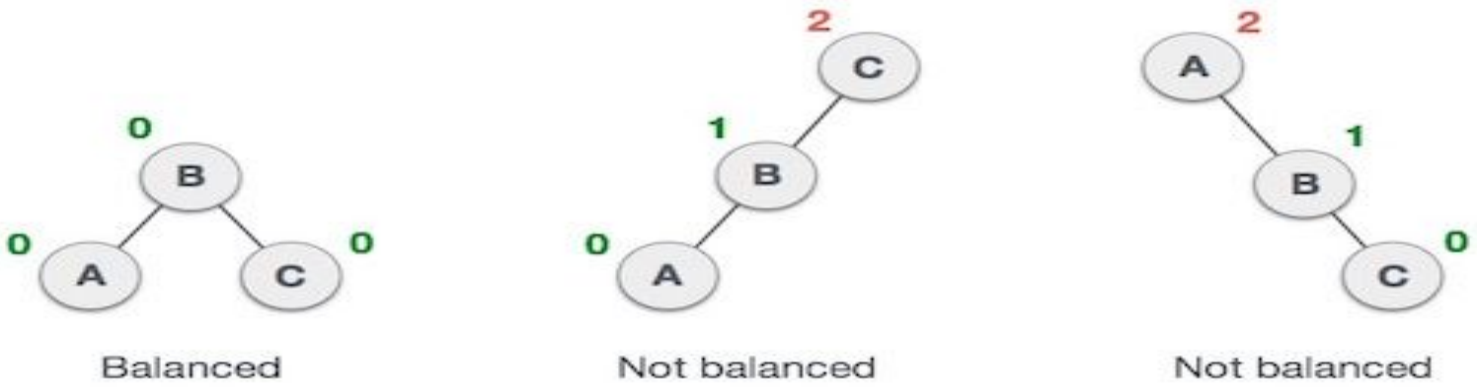
Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.

AVL Tree

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1.

This difference is called the **Balance Factor**.




```
BalanceFactor = height(left-sutree) - height(right-sutree)
```

AVL Rotations

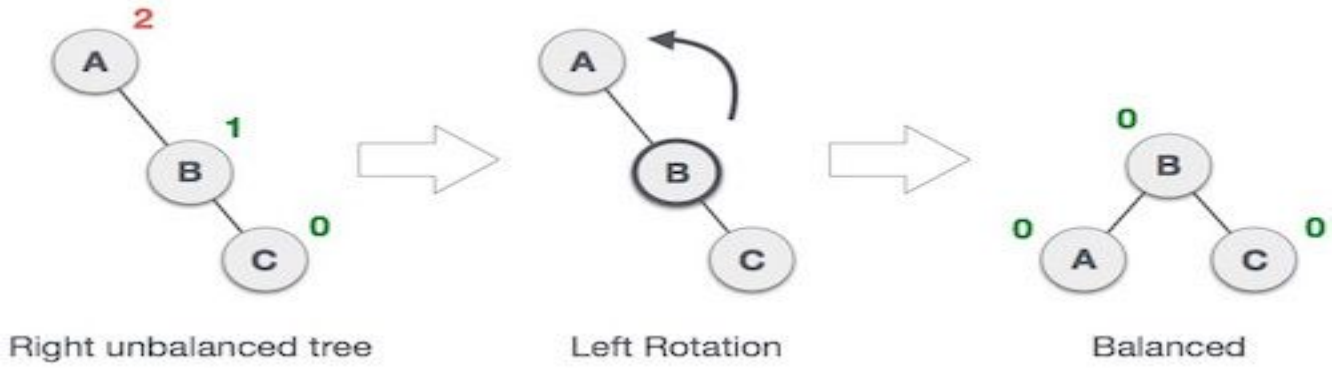
To balance itself, an AVL tree may perform the following four kinds of rotations

Left rotation
Right rotation
Left-Right rotation
Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations.

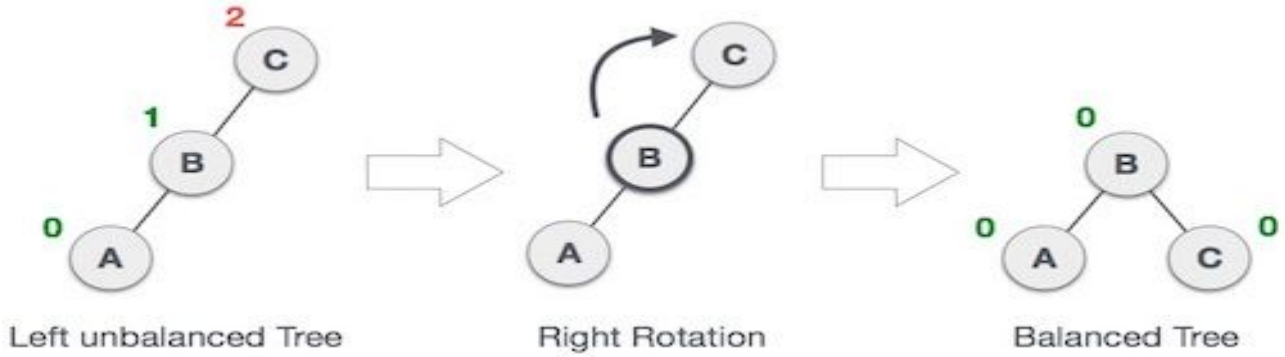
Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation



Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



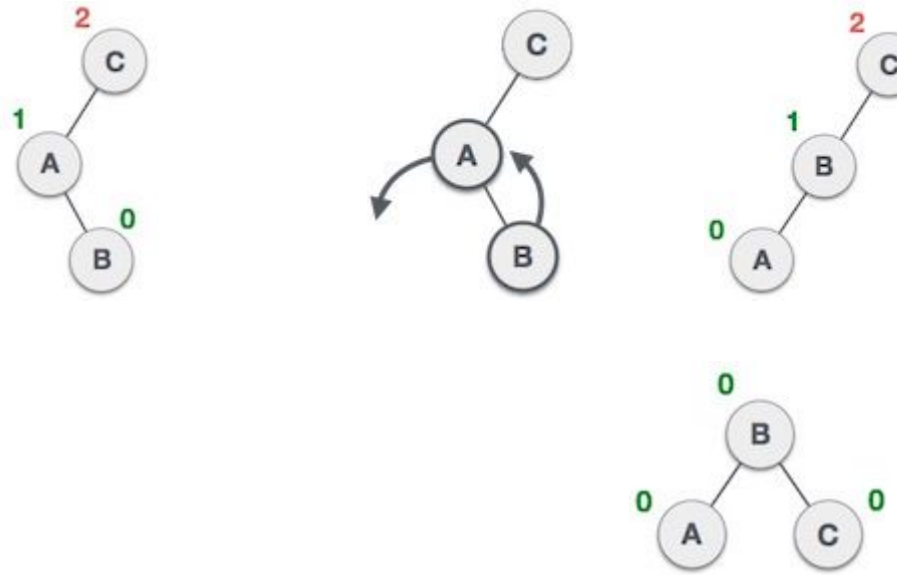
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations.

To understand them better, we should take note of each action performed while rotation.

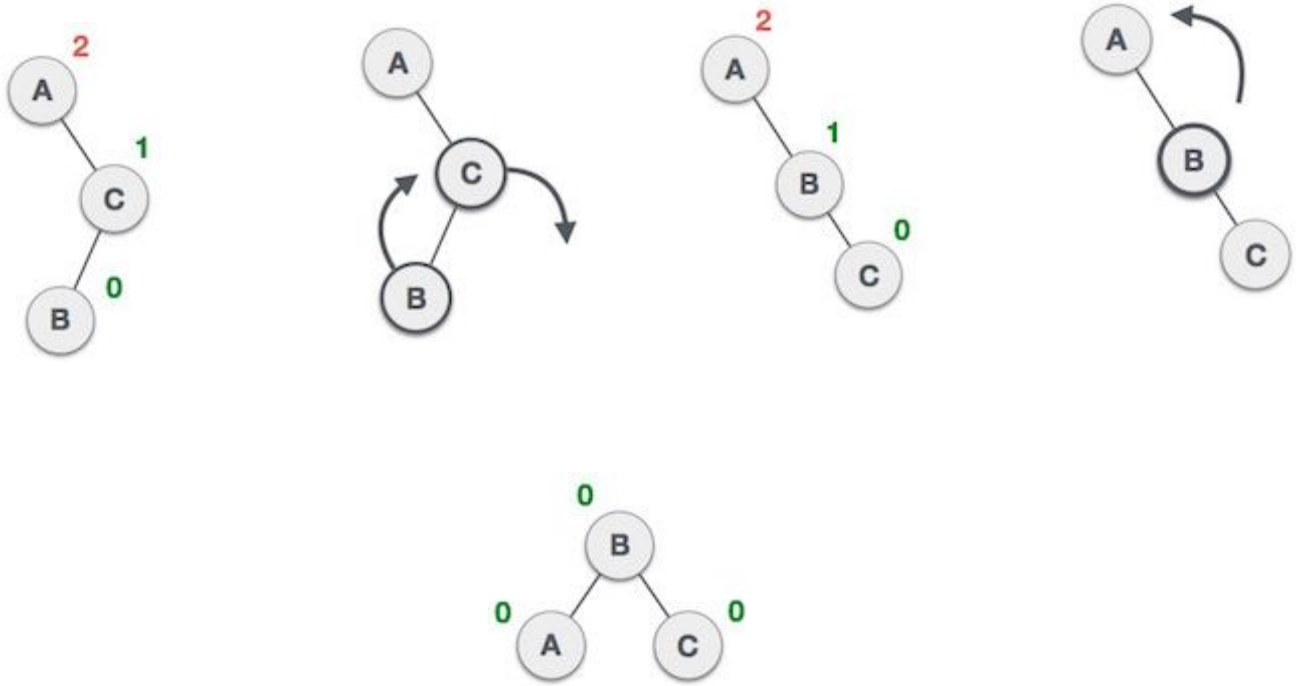
Let's first check how to perform Left-Right rotation.

A left-right rotation is a combination of left rotation followed by right rotation.



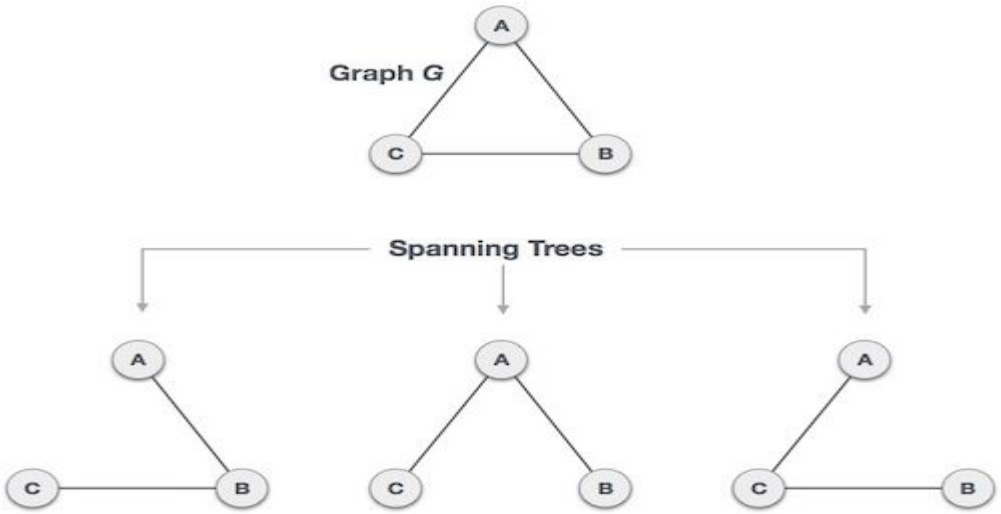
Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.



Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..



A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

A connected graph G can have more than one spanning tree.

All possible spanning trees of graph G, have the same number of edges and vertices.

The spanning tree does not have any cycle (loops).

Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.

Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).

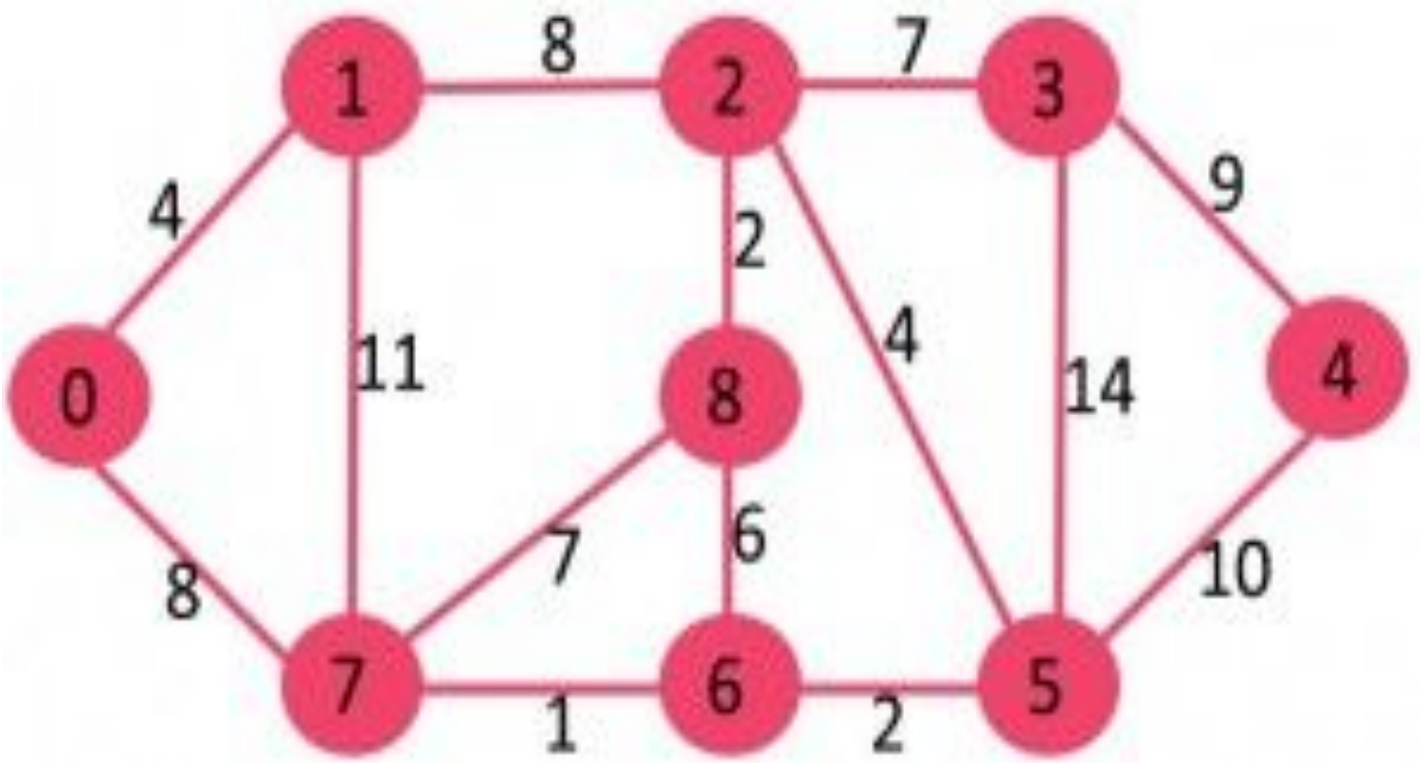
A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here −

Kruskal's Algorithm
Prim's Algorithm

Both are greedy algorithms.



The steps for implementing Kruskal's algorithm are as follows:

Sort all the edges from low weight to high

Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
Keep adding edges until we reach all vertices.

PRIMS Algorithm

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*

red-black tree :

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black).

Rules That Every Red-Black Tree Follows:

Every node has a colour either red or black.

The root of the tree is always black.

There are no two adjacent red nodes (A red node cannot have a red parent or red child).

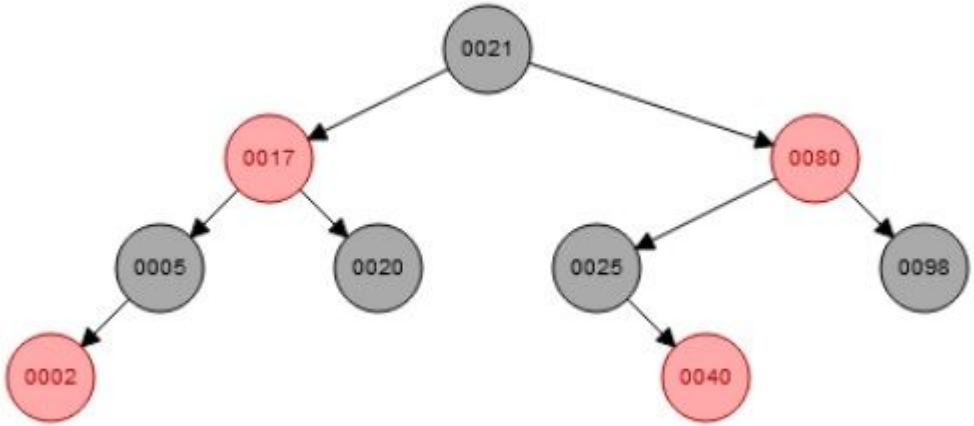
Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

There are some conditions for each node. These are like below −

Each node has color. Which is either Red or Black
The root will be always black
There will be no two adjacent Red nodes
Every path from a node (including root) to any of its descendent NULL node has the same number of black nodes.



Red Black Tree is a Self-Balanced Binary Search Tree in which each node of the tree is colored with either Red or Black. To insert an element in a red-black tree the idea is very simple − we perform insertion just like we insert in a regular binary tree. We start off from the root node by checking the color of the node and insert it into a particular position. However, In a red-black tree there should be some additional procedure to insert an element in it.

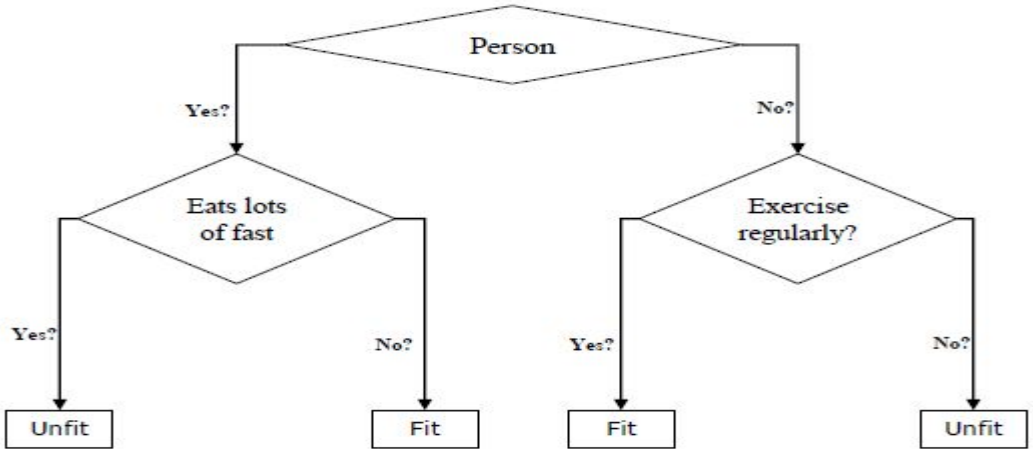
However, we should know that a red-black tree is balanced when it follows the conditions −

Every Root Node must be Black.
Every node is either Red or Black.
If a node is red, then its children must be black.
The path from the root till the end must contain an equal number of black nodes.

Decision Tree

In general, Decision tree analysis is a predictive modelling tool that can be applied across many areas. Decision trees can be constructed by an algorithmic approach that can split the dataset in different ways based on different conditions. Decisions trees are the most powerful algorithms that falls under the category of supervised algorithms.

They can be used for both classification and regression tasks. The two main entities of a tree are decision nodes, where the data is split and leaves, where we got outcome. The example of a binary tree for predicting whether a person is fit or unfit providing various information like age, eating habits and exercise habits, is given below −



We have the following two types of decision trees.

Classification decision trees − In this kind of decision trees, the decision variable is categorical. The above decision tree is an example of classification decision tree.

Regression decision trees − In this kind of decision trees, the decision variable is continuous.

A decision tree is a structure that includes a root node, branches, and leaf nodes. Each internal node denotes a test on an attribute, each branch denotes the outcome of a test, and each leaf node holds a class label. The topmost node in the tree is the root node.