UNIT IV

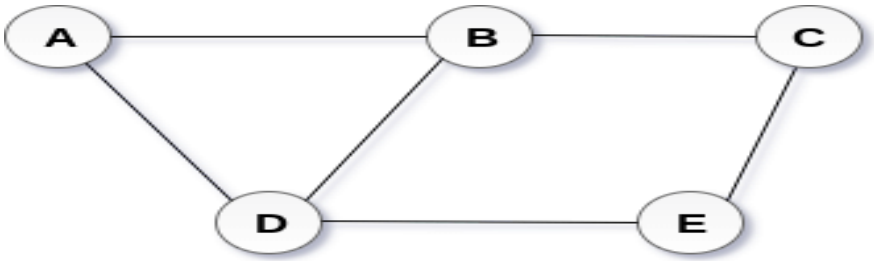
GRAPHS INTRODUCTION

Graph (Write about graph?)

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

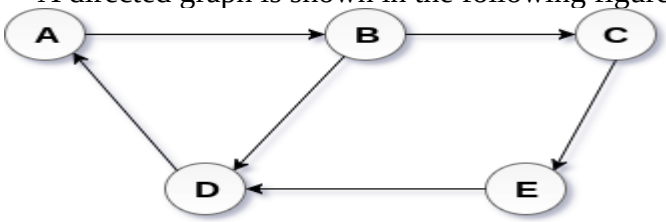
A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Undirected Graph

Directed and Undirected Graph

- A graph can be directed or undirected.
- However, in an undirected graph, edges are not associated with the directions with them.
- An ***undirected graph*** is shown in the above figure since its edges are not attached with any of the directions.
- If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.
- In a ***directed graph***, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.
- A directed graph is shown in the following figure.



Directed Graph

Graph Terminology

Path: A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path: A path will be called as closed path if the initial node is same as terminal node.

Simple Path: If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle: A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.
Connected Graph: A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
Complete Graph: A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.
Weighted Graph: In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.
Digraph: A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
Loop: An edge that is associated with the similar end points can be called as Loop.
Adjacent Nodes: If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbors or adjacent nodes.
Degree of the Node: A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

2.GRAPH REPRESENTATION (Explain the representation methods of graph?)

By Graph representation, means the technique which is to be used in order to store some graph into the computer's memory.

There are two ways to store Graph into the computer's memory.

1. Sequential Representation
2. Linked Representation

1. Sequential Representation

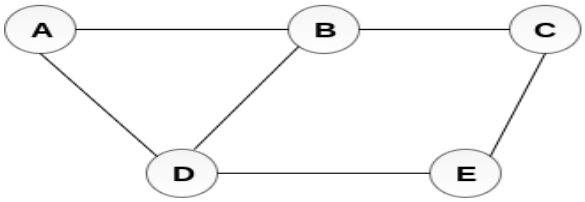
In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges.

In adjacency matrix, the rows and columns are represented by the graph vertices.

A graph having n vertices, will have a dimension **n x n**.

An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j.

An undirected graph and its adjacency matrix representation is shown in the following figure.



Undirected Graph

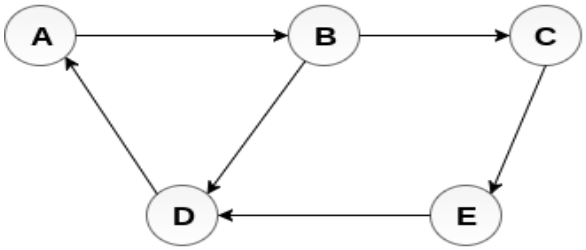
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

In the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

A directed graph and its adjacency matrix representation is shown in the following figure.



Directed Graph

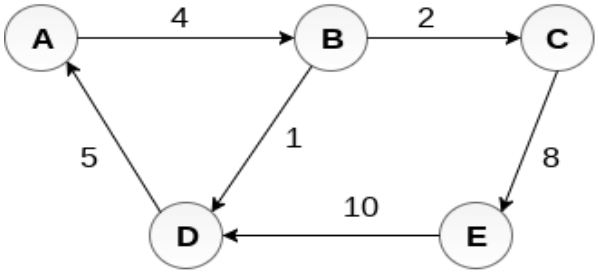
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Representation of weighted directed graph is different.

Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



Weighted Directed Graph

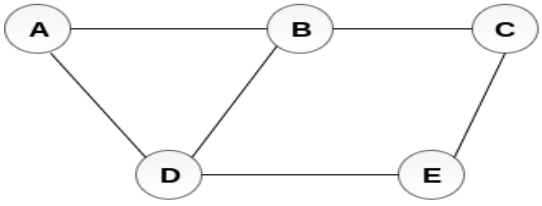
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

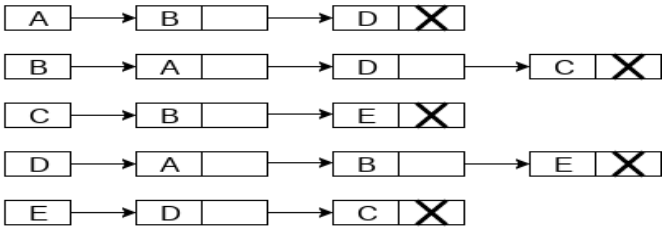
2.Linked Representation

In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.



Undirected Graph



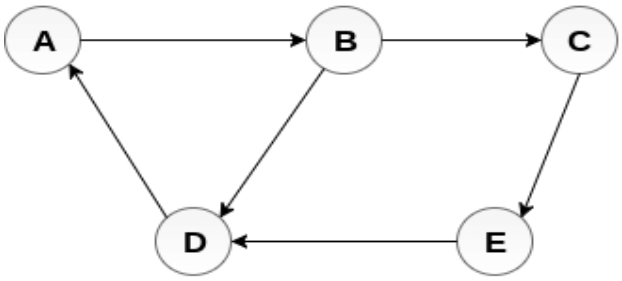
Adjacency List

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node.

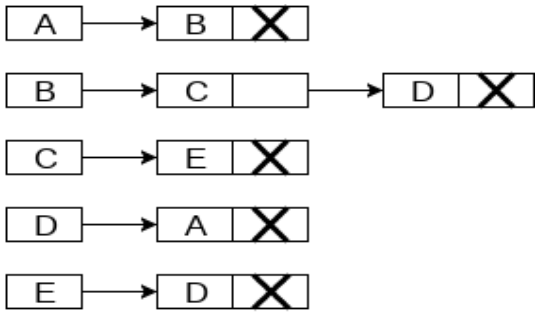
If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list.

The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.



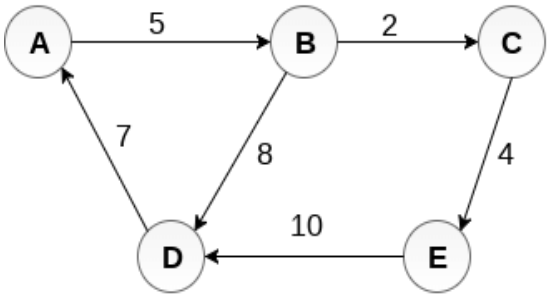
Directed Graph



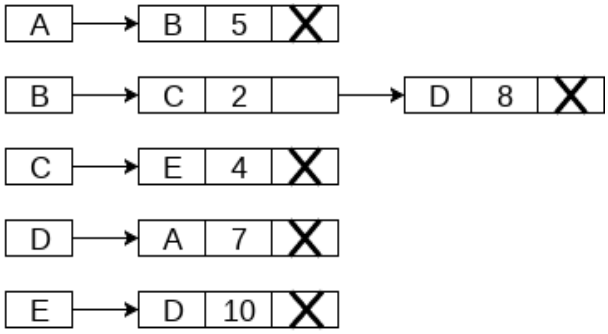
Adjacency List

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



Adjacency List

3.GRAPH TRAVERSAL METHODS (Explain Graph Traversal methods?)

Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs.

1. Breadth First Search
2. Depth First Search

Breadth First Search (BFS) Algorithm(Explain BFS)

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighboring nodes.

Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below.

The algorithm starts with examining the node A and all of its neighbors. In the next step, the neighbors of the nearest node of A are explored and process continues in the further steps.

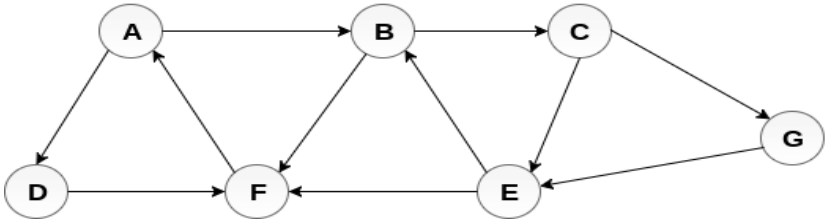
The algorithm explores all neighbors of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Enqueue the starting node A and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until QUEUE is empty
- **Step 4:** Dequeue a node N. Process it and set its STATUS = 3 (processed state).
- **Step 5:** Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) AND GOTO step 3
- **Step 6:** EXIT

Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**.

QUEUE1 holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

Lets start examining the graph from Node A.

- Add A to QUEUE1 and NULL to QUEUE2.
  QUEUE1 = {A}
  QUEUE2 = {NULL}
- Delete the Node A from QUEUE1 and insert all its neighbors. Insert Node A into QUEUE2
  QUEUE1 = {B, D}
  QUEUE2 = {A}
- Delete the node B from QUEUE1 and insert all its neighbors. Insert node B into QUEUE2.
  QUEUE1 = {D, C, F}
  QUEUE2 = {A, B}

- Delete the node D from QUEUE1 and insert all its neighbors. Since F is the only neighbor of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.
  QUEUE1 = {C, F}
  QUEUE2 = { A, B, D}
- Delete the node C from QUEUE1 and insert all its neighbors. Add node C to QUEUE2.
  QUEUE1 = {F, E, G}
  QUEUE2 = {A, B, D, C}
- Remove F from QUEUE1 and add all its neighbors. Since all of its neighbors has already been added, we will not add them again. Add node F to QUEUE2.
  QUEUE1 = {E, G}
  QUEUE2 = {A, B, D, C, F}
- Remove E from QUEUE1, all of E's neighbors has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.
  QUEUE1 = {G}
  QUEUE2 = {A, B, D, C, F, E}
- Now, backtrack from E to A, using the nodes available in QUEUE2.
  The minimum path will be **A → B → C → E**.

Depth First Search (DFS) Algorithm (Explain DFS)

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children.

The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

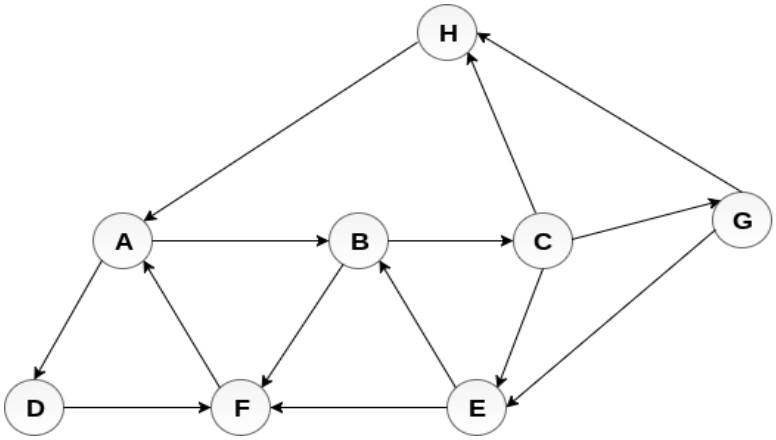
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) goto step 3.
- **Step 6:** EXIT

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution :

- Push H onto the stack
  STACK : H
- POP the top element of the stack i.e. H, print it and push all the neighbors of H onto the stack that are is ready state.
  Print H
  STACK : A
- Pop the top element of the stack i.e. A, print it and push all the neighbors of A onto the stack that are in ready state.
  Print A
  Stack : B, D
- Pop the top element of the stack i.e. D, print it and push all the neighbors of D onto the stack that are in ready state.
  Print D
  Stack : B, F
- Pop the top element of the stack i.e. F, print it and push all the neighbors of F onto the stack that are in ready state.
  Print F
  Stack : B
- Pop the top of the stack i.e. B and push all the neighbors
  Print B
  Stack : C
- Pop the top of the stack i.e. C and push all the neighbors.
  Print C
  Stack : E, G
- Pop the top of the stack i.e. G and push all its neighbors.
  Print G
  Stack : E
- Pop the top of the stack i.e. E and push all its neighbors.
  Print E
  Stack :
- Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be : H → A → D → F → B → C → G → E

4.SPANNING TREE(What is a spanning tree? Explain about minimum spanning tree?)

Graph

A graph can be defined as a group of vertices and edges to connect these vertices. The types of graphs are given as follows -

- **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.
- **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
- **Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

Spanning tree

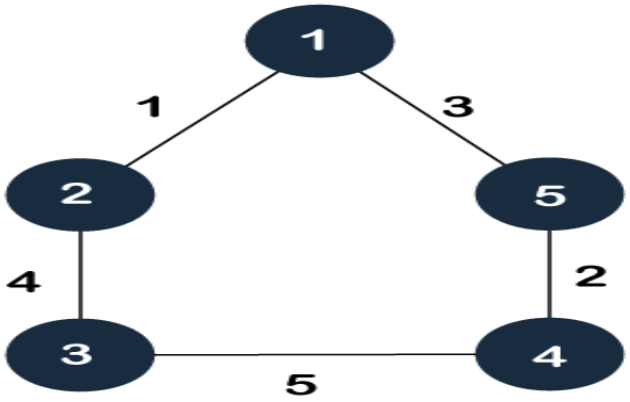
- A spanning tree can be defined as the sub graph of an undirected connected graph. It includes all the vertices along with the least possible number of edges.
- If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.
- A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them.
- All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be $\mathbf{5^{5-2} = 125.}$

Applications of the spanning tree

Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

- Cluster Analysis
- Civil network planning
- Computer network routing protocol

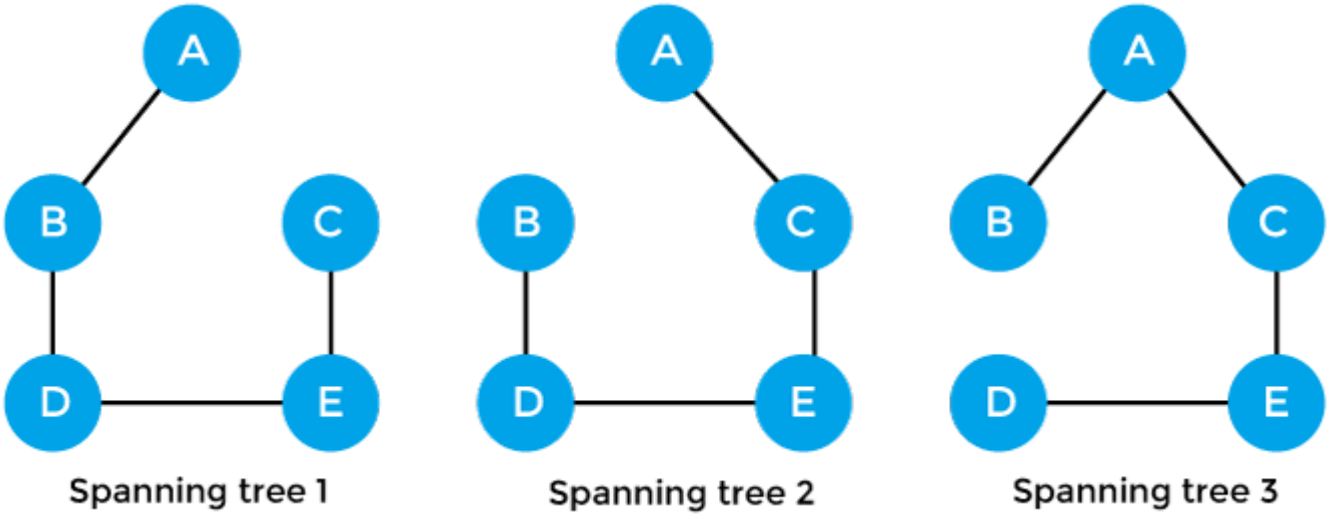
Example of Spanning tree :Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices.

The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

- 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 complete 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.

So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

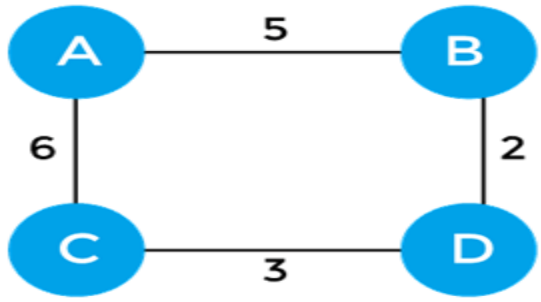
Minimum Spanning tree

A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum.

The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

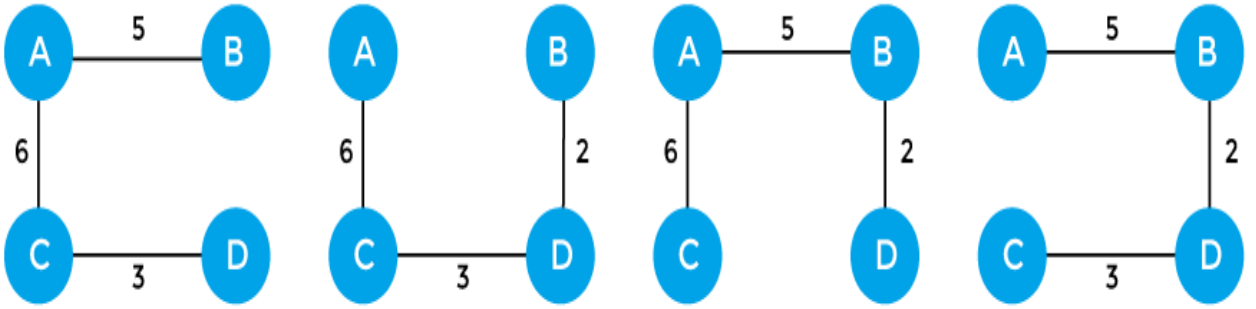
Example of minimum spanning tree

Let's understand the minimum spanning tree with the help of an example.



Weighted graph

The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



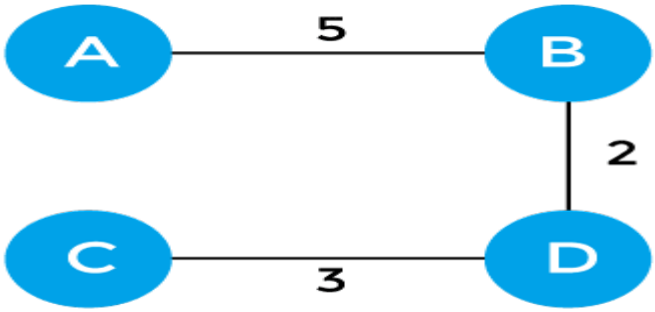
Sum = 14
Minimum spanning tree - 1

Sum = 11
Minimum spanning tree - 2

Sum = 13
Minimum spanning tree - 3

Sum = 10
Minimum spanning tree - 4

So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Sum = 10

Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

- Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
- It can be used to find paths in the map.

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm(Explain Prims algorithm)

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.
Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum.
The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph.

Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

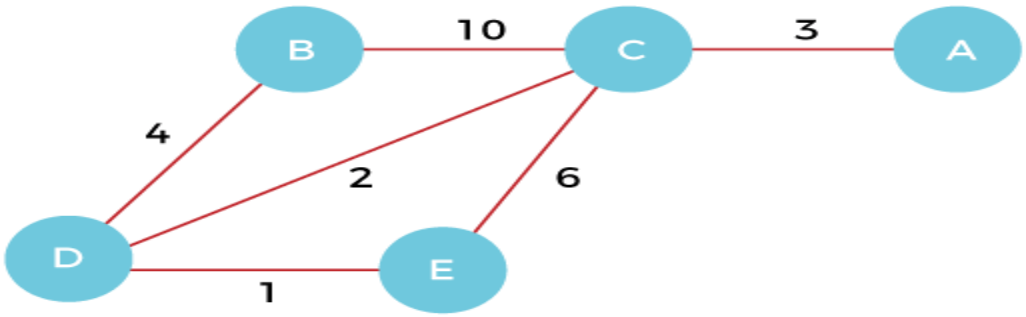
- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

The applications of prim's algorithm are -

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

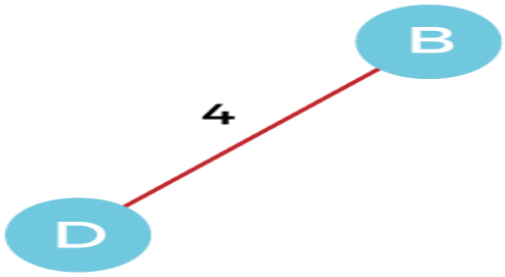
Example of prim's algorithm

Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.
Suppose, a weighted graph is -

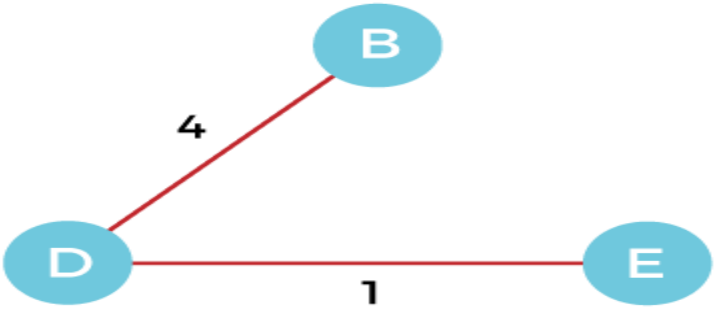


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

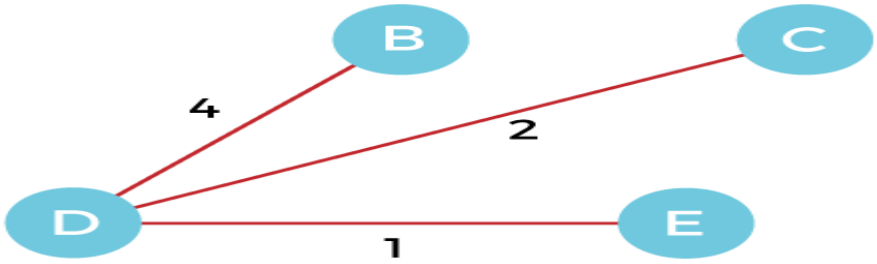
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



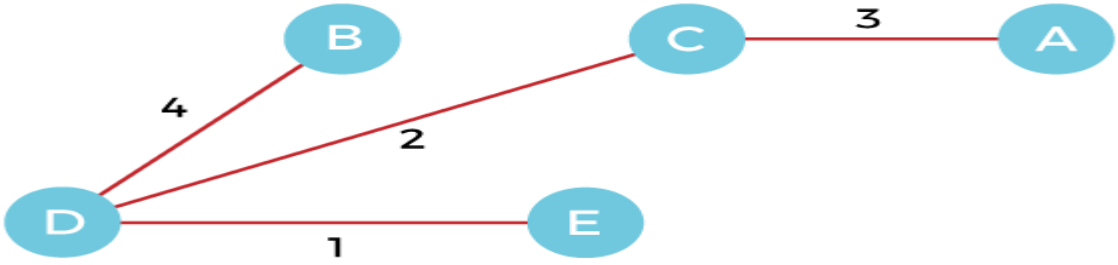
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Algorithm

Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
Step 4: Add the selected edge and the vertex to the minimum spanning tree T
[END OF LOOP]
Step 5: EXIT

Complexity of Prim's algorithm

Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -

- **Time Complexity**

Data structure used for the minimum edge weight	Time Complexity
Adjacency matrix, linear searching	$O(\|V\|^2)$
Adjacency list and binary heap	O(|E| log |V|)
Adjacency list and Fibonacci heap	O(|E|+ |V| log |V|)

Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires $O(|V|^2)$ running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm.

The time complexity of the prim's algorithm is O(E logV) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

Kruskal's Algorithm(Explain Kruskal's algorithm)

In this article, we will discuss Kruskal's algorithm. Here, we will also see the complexity, working, example, and implementation of the Kruskal's algorithm.

But before moving directly towards the algorithm, we should first understand the basic terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph.

The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph.

It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

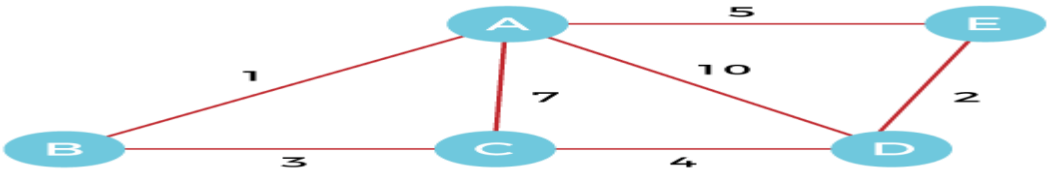
The applications of Kruskal's algorithm are -

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

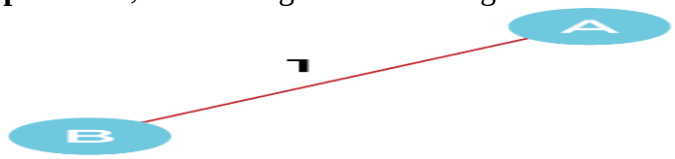
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

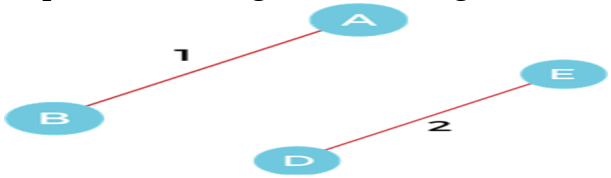
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

Step 1 - First, add the edge **AB** with weight **1** to the MST.

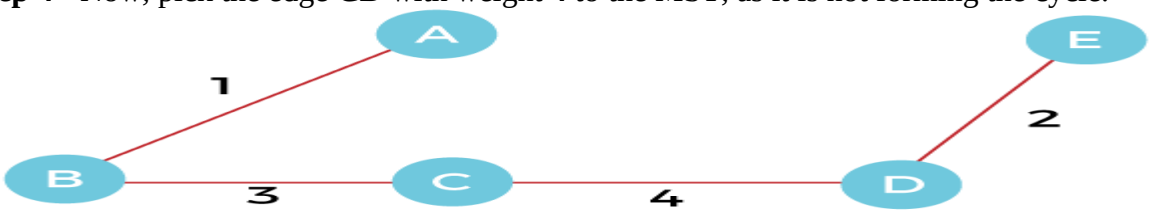


Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.

Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.



Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.
Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.
Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.
So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -

The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.
Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

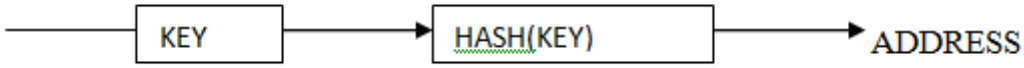
Algorithm
Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
Step 2: Create a set E that contains all the edges of the graph.
Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
Step 4: Remove an edge from E with minimum weight
Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
(**for** combining two trees into one tree).
ELSE
Discard the edge
Step 6: END

Complexity of Kruskal's algorithm
Now, let's see the time complexity of Kruskal's algorithm.

- **Time Complexity**
  The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

HASHING INTRODUCTION(What is hashing?)

- Hashing is finding an address where the data is to be stored or to locate using a key with the help of an arithmetic function.
- A hash table is an effective data structure for implementing hashing.
- Hash functions transform a key into an address, hashing is a technique used for storing and retrieving information associated with it.



- The address is used as the basis for storing and retrieving records, this address is called the "home address" of the record.
- Hashing is similar to indexing as it involves associating a key with a relative record address.
- Hashing differs from indexing in two ways:
  1) With hashing, the generated address appears to be random, there is no connection between the key and the location of the corresponding record, even though the key is used to determine the location of the record. So, hashing is also referred to as "randomizing".
  2) With hashing, two different keys may be transferred to the same address, so, two records may be sent to the same place in a file, it is called "Collision". The two or more records that result in the same home address are known as "Synonyms".

KEY TERMS USED IN HASHING

The terms associated with hashing are

1) **Hash table**: It is an array of size Max(0 to Max-1)
2) **Hash Function:** It is one that maps a key in the range [0 to Max-1], this address is used in the
3) hash function for storing and retrieving records. A Hash function is a function, which transforms a key into an address, this address is called the "home address". All home addresses refer to a particular area of the memory called the "prime area".
4) **Bucket:** A bucket is an index position in a hash table that can store more than one record. When the same index is mapped with two keys, both the records are stored in the same bucket.
5) **Probe:** Each action of address calculation and check for success is called as a probe.
6) **Collision:** The result of two keys hashing into the same address is called collision.
7) **Synonym:** Keys that hash the same address are called Synonyms.
8) **Overflow:** The result of many keys hashing to a single address and lack of room in the bucket is known as overflow. Collision and overflow are synonymous when the bucket is of size 1.
9) **Open or External hashing:** When we allow records to be stored in potentially unlimited space, it is called as open or external hashing.
10) **Closed or internal hashing:** when we use fixed space for storage, eventually limiting the number of records to be stored, it is called as closed or internal hashing.
11) **Hash function:** it is an arithmetic function that transforms a key into an address, which is used for storing and retrieving a record.
12) **Perfect hash function:** The hash function that transforms different keys into different address is called a perfect hash function. The worth of hash function depends on how well it avoid collision.
13) **Load density:** The maximum storage capacity, i.e., the maximum number of records that can be accommodated, is called as loading density.

14) **Full table:** A full table is one in which all locations are occupied. Based on characteristics of hash functions, there are always empty locations, rather a hash function should not allow the table to filled in more than 75%.
15) **Load factor:** It is the number of records stored in a table divided by the maximum capacity of the table, expressed in terms of percentage.
16) **Rehashing:** Rehashing is with respect to closed hashing. When we try to store the record with key1 at the bucket position has(key1) and find that it already holds a record, it is collision situation. To handle collision, we use alternative hash as hash1(key1), hash2(key1) and so on with in the bucket table, to place the record with key1. This is known as "Rehashing".

Issues in Hashing

In case of collision, there are two main issues to be considered:

1) We need a good hashing function that minimizes the number of collisions.
2) We want an efficient collision resolution strategy, so as to store or locate synonyms.

HASH FUNCTIONS(Explain different hash functions)

- To store a record in a hash table, a hash function is applied to the key of the record being stored, returning an index within the range of the hash table.
- The record is stored at that index, if it is empty.
- With direct addressing, a record with key 'k' is stored in slot 'k'.
- With hashing, the record is stored at the location Hash(k), where Hash(k) is the function.
- The hash function 'Hash(k)' is used to compute the slot for the key 'k'.

Features of a good hash function

1. Addresses generated from the key are uniformly and randomly distributed.
2. Small variations in the value of the key will cause large variations in the record addresses to distribute records evenly.
3. The hash function must minimize the occurrence of Collision.

Different methods of implementing hash functions are:

a) Division method:

- One of the required features of the hash function is that the home address must be with in the hash table index range.
- One simple choice for a hash function is to use the Modular division(%) which gives remainder of the integer division.
- Hash(Key)=Key%M
- Where 'M' is the size of the hash table with indices 0 to M-1. A good choice of M is that should be a Prime number greater than 20.
- If the key is negative then the resultant address is also negative, if the key or M is NULL, the result is NULL.

b) Multiplication Method:

- Multiply the key by a constant A in the range 0<A<1, and extract the fractional part of the 'key*A'.
- Then multiply this value by M and take the floor of the result.
  - Hash(Key)= ⌊ M*((Key*A)%1)⌋
- Where '(Key*A)%1'is the fractional part of Key*A, that is Key*A- ⌊Key*A⌋ .
- Ex: (sqrt(5)-1/2=0.6180339887)
- In this method $M=2^P$, P is some integer.

- Choose $M=2^P$
- Multiply w bits of key by floor(A*2W) to obtain a 2W bit product.
- Extract the P most significant bits of the lower half of this product as address.

- Floor(x)= $\lfloor x \rfloor$, is the largest integer not greater than x.
- Ex: floor(2.4)=2 floor(2.9)=2 floor(-2.7)= -3

c) Extraction method:

- When a portion of the key is used for address calculation, it is called as the extraction method.
- In digit extraction, a few digits are selected, extracted from the key, and are used as the address.
- If the key is of 6 digits, we require an address of 3 digits, then we can select odd digits (1st, 3rd, 5th digits) as address for hash table.

Ex:

Key	Hashed address
345678	357
234137	243
952671	927

- Another way is to extract the first two and the last one or two digits, for key 345678, the address is 348 or 3478.
- If the portion of the key is carefully selected, it can be sufficient for hashing, provided the remaining portion distinguishes the keys in an insufficient way.

d) Mid-Square hashing

- Mid square hashing suggests to take the square of the key and extract the middle digits of the squared key as the address. The difficulty is when the key is large.
- Mid square is used when the key size is less than or equal to 4 digits. If the key is a string, it has to be preprocessed to produce a number.

Key	Square	Hashed Address
2341	5480281	802
1671	2792241	922

- If the key is large, select a potion of the key and square it.

Key	Square	Hashed Address
234137	234*234=54756	475
567187	567*567=321489	148

e) Folding technique

- In this the key is divided into subparts that are combined or folded and then combined to form the address.
- For a key with 9 digits, divide the digits into three parts, add them and use the result as an address.
- Ex: key is 987654321, divide into 3 subparts as 987, 654, 321 and add them, 987+654+321=1962.
- Here the size of the subparts of the key is same as that of the address, so discard 1 from 1962, then the address is 962.
- There are two types of folding techniques:
  1. **Fold shift:** key is divided into several parts of the size of the address, left, right and middle and add them.
     Ex: key is 987654321, left =987, right=321, middle=654

Sum=left+middle+right=987+654+321=1962, discard 1, then address is 962

2. **Fold Boundary:** key is divided into parts of the size of the address, left and right parts are folded on the fixed boundary between them and the centre part(reverse of parts)
   Ex: key is 987654321, left =987, right=321, middle=654,
   Sum=789+456+123=1368, discard 1, then address is 368

f) Rotation

- When the keys are serial, they vary in the last digit and this leads to the creation of synonyms.
- Rotating the key would minimize this problem. This method is used along with other methods.
- Here, the key is rotated right by one digit and then folding technique is used to avoid synonyms.
- Key is 141206055, when it is rotated 514120605. Then the address is calculated using any other hash function.
- Using folding shift, 514, 120, 605, sum is 1239, discard 1, then address is 239.

g) Universal hashing

- The effective way to improve the hashing technique is to choose a hash function randomly in a way that is independent of the keys that are actually going to be stored.
- This approach is called universal hashing and yields good performance on the average.
- The main idea behind universal hashing is to select the hash function randomly at runtime from a designed set of functions.
- Because of randomization, the algorithm can behave differently on each execution, even for the same input.
- This approach guarantees good average case performance, no matter what keys are provided as input.