UNIT-V

Introduction to Graphs

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of linkes known as edges (or Arcs) which connets the vertices. A graph is defined as follows...

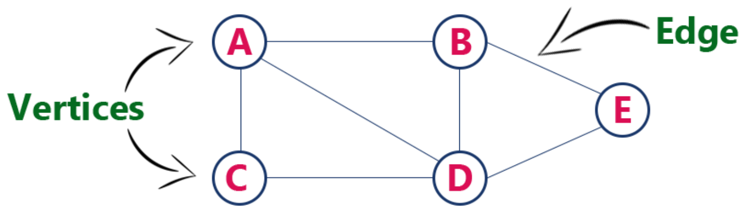
Graph is a collection of vertices and arcs which connects vertices in the graph

Graph is a collection of nodes and edges which connects nodes in the graph

Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with 5 vertices and 6 edges.
This graph G can be defined as G = (V , E)
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



We use the following terms in graph data structure...

Vertex

A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is an edge with cost on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin

If an edge is directed, its first endpoint is said to be origin of it.

Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

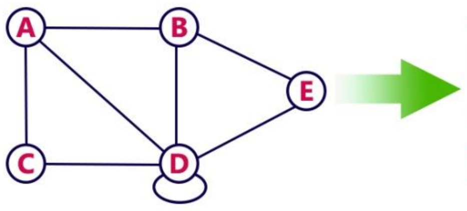
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Adjacency List**

Adjacency Matrix

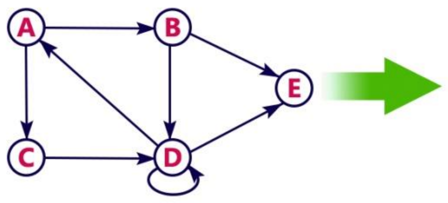
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...

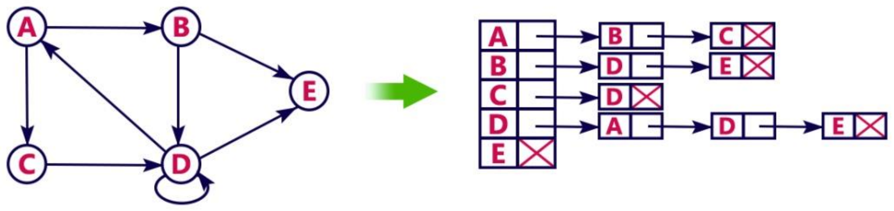


$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

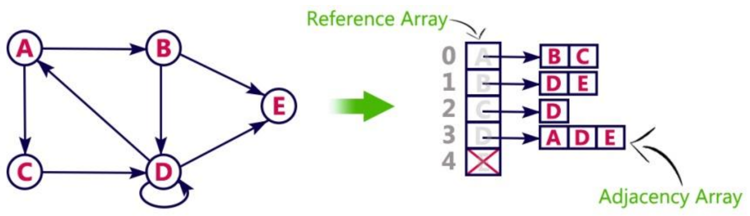
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Graph Traversals - DFS

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticces of graph without getting into looping path. There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of graph.
We use the following steps to implement DFS traversal...

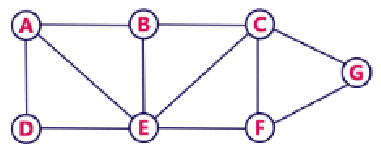
- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the verex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we came to current vertex

Back tracking is coming back to the vertex from which we came to current vertex.

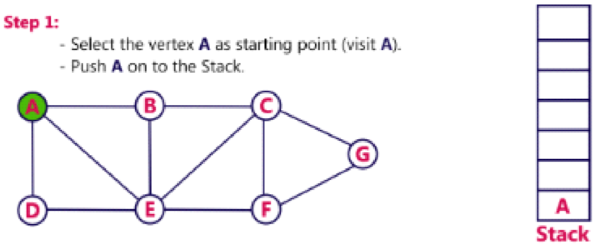
Example

Consider the following example graph to perform DFS traversal



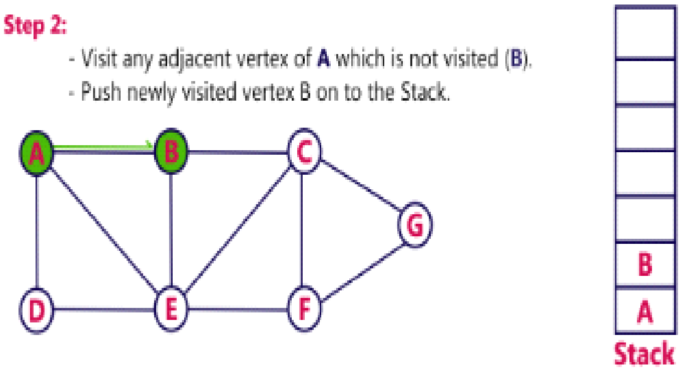
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



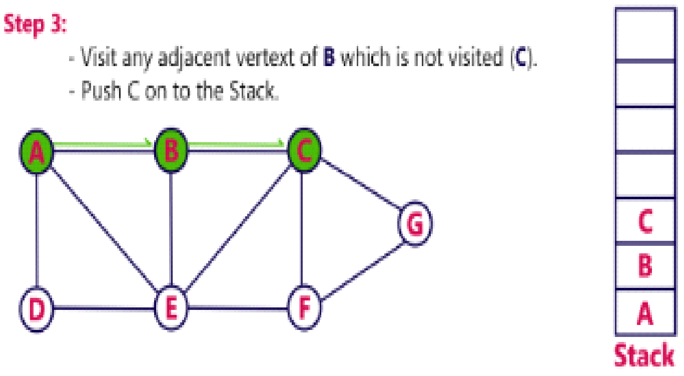
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



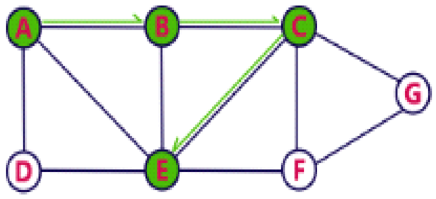
Step 3:

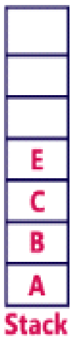
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



Step 4:

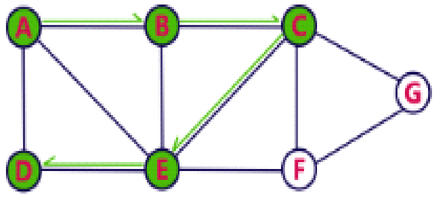
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

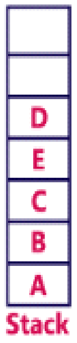




Step 5:

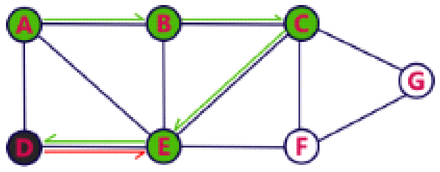
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

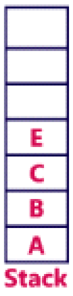




Step 6:

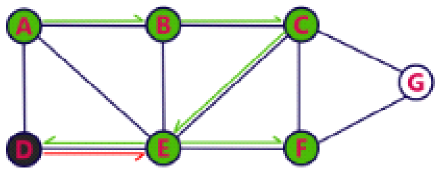
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

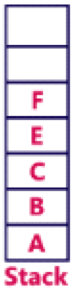




Step 7:

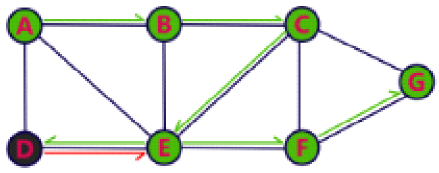
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

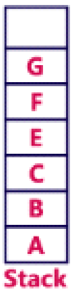




Step 8:

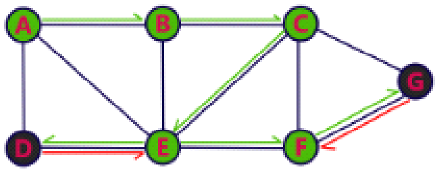
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

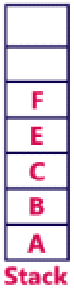




Step 9:

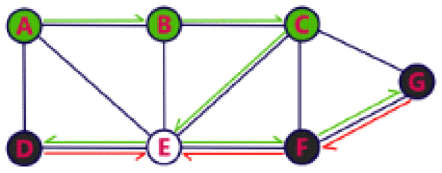
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

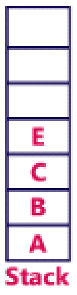




Step 10:

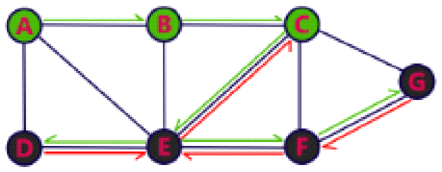
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

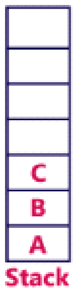




Step 11:

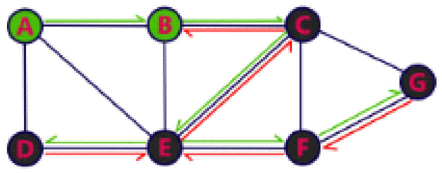
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

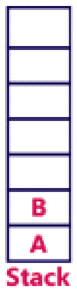




Step 12:

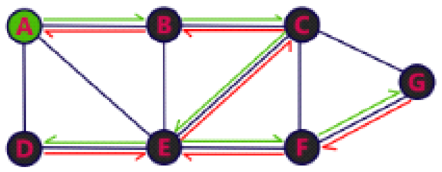
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

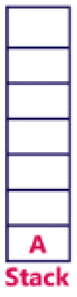




Step 13:

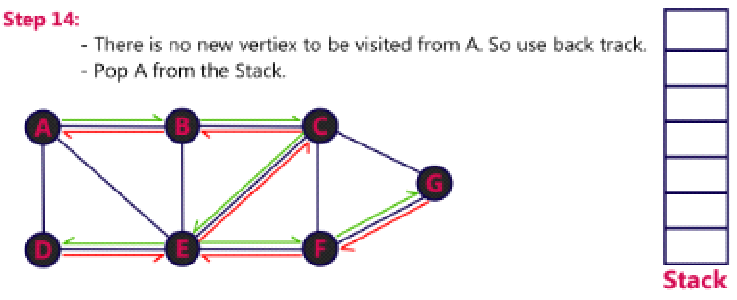
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



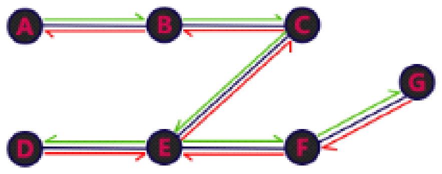


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Graph Traversals - BFS

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticces of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

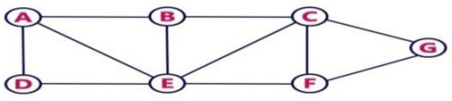
BFS (Breadth First Search)

BFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

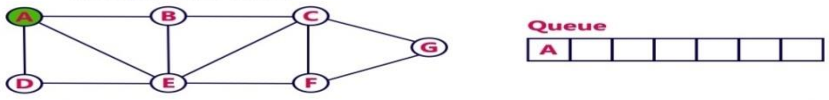
We use the following steps to implement BFS traversal...

- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the **adjacent** vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

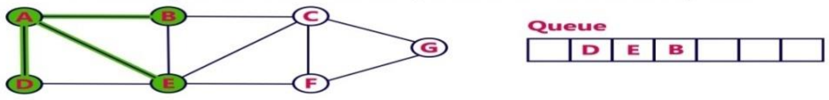
Consider the following example graph to perform BFS traversal



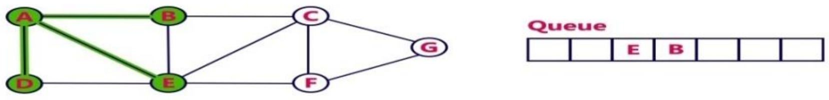
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



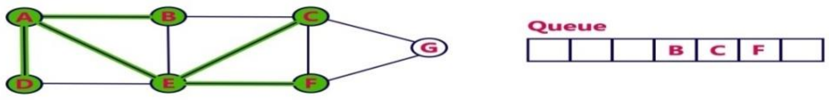
Step 2:
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



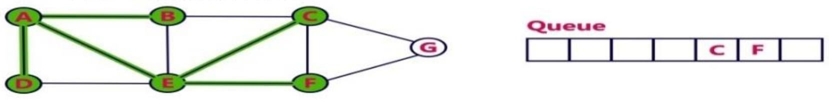
Step 3:
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



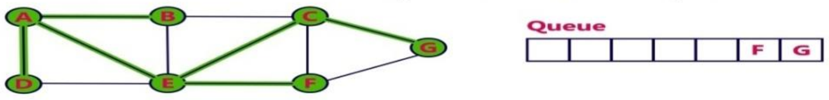
Step 4:
- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



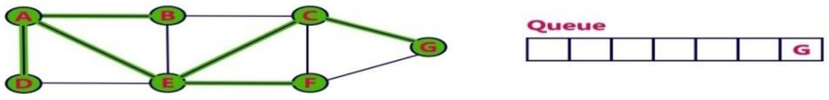
Step 5:
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



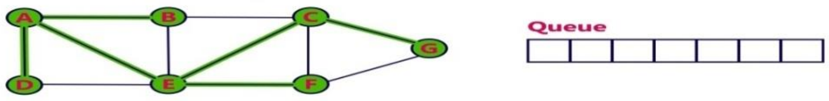
Step 6:
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



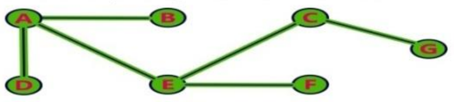
Step 7:
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Step 8:
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Applications of Graphs:

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. Social network graphs: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

2. Transportation networks. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. Document link graphs. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

4. Scene graphs. In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

5. Robot planning. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

6. Neural networks. Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.

7. Graphs in compilers. Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

8.Computer hardware: Compilers uses graph coloring algorithms for Register allocation to variables , Calculate parallelism degree, Very useful in analytical modeling[3] , Addressing the sequence of instruction execution , Resource allocation and Economizing the memory space(file organization).

Hashing

In all search techniques like linear search, binary search and search trees, the time required to search an element is depends on the total number of element in that data structure. In all these search techniquies, as the number of element are increased the time required to search an element also increased linearly.

Hashing is another approach in which time required to search an element doesn't depend on the number of element. Using hashing data structure, an element is searched with **constant time complexity**. Hashing is an effective way to reduce the number of comparisions to seach an element in a data structure.

Hashing is defined as follows...

Hashing is the process of indexing and retrieving element (data) in a datastructure to provide faster way of finding the element using the hash key.

Here, hash key is a value which provides the index value where tha actual data is likely to store in the datastructure.

In this datastructure, we use a concept called **Hash table** to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a **hash function**. That means every entry in the hash table is based on the key value generated using a hash function.

Hash Table is defined as follows...

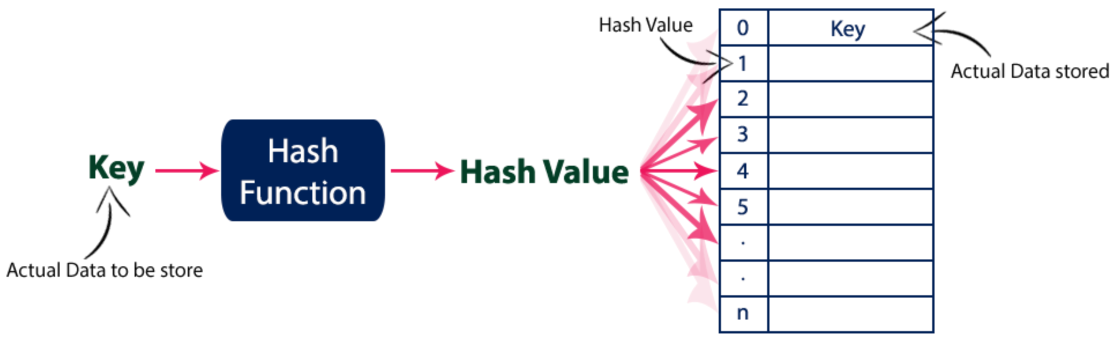
Hash table is just an array which maps a key (data) into the datastructure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1)).

Hash tables are used to perform the operations like insertion, deletion and search very quickly in a datastructure. Using hash table concept insertion, deletion and search operations are accoplished in constant time. Generally, every hash table make use of a function, which we'll call the **hash function** to map the data into the hash table.

A hash function is defined as follows...

Hash function is a function which takes a piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table.

Basic concept of hashing and hash table is shown in the following figure...



Different Hash Functions:

1.Division Method:

It is the most simple method of hashing an integer x. this method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

h(X)=x mod M

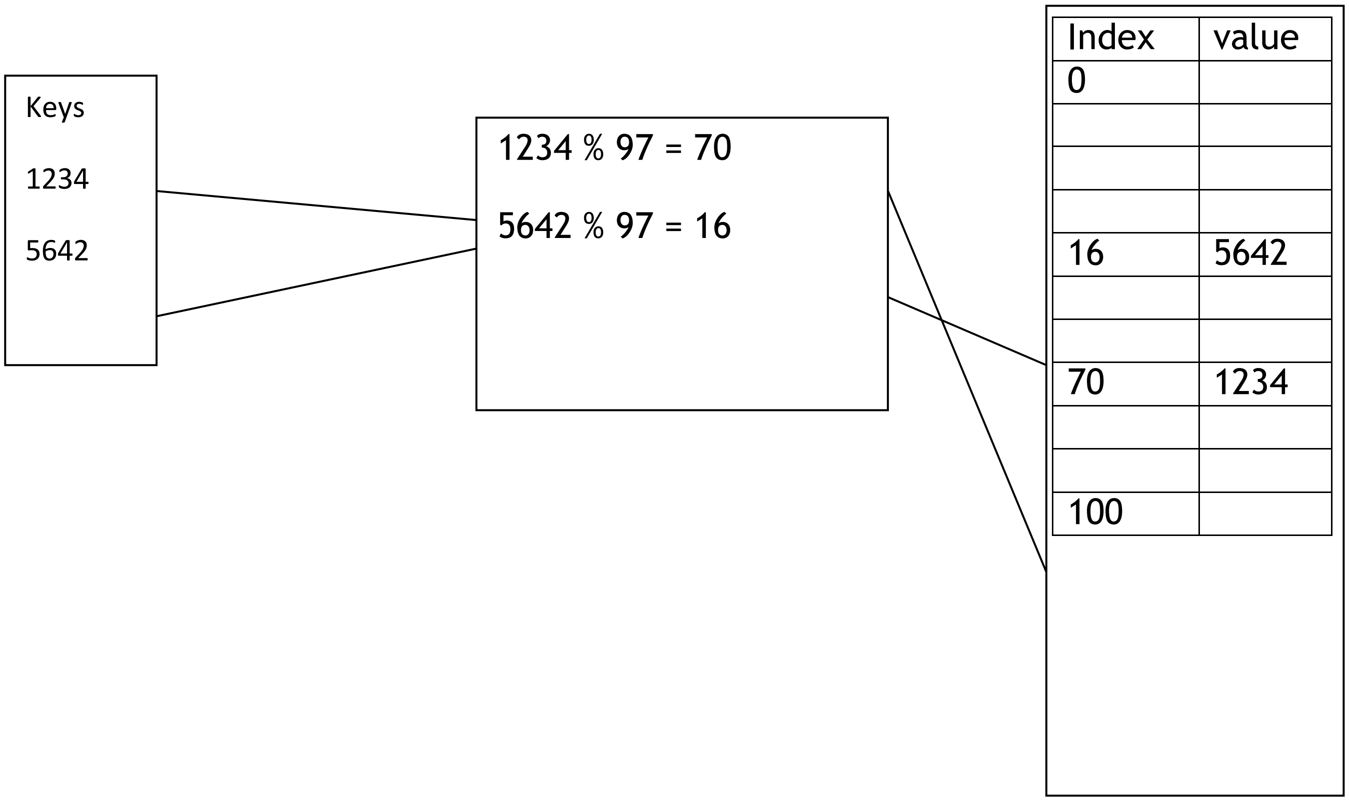
Example: Calculate the hash values of keys 1234 and 5462 .

Set M=97(prime no.)

hash values can be calculated as

h(1234)= 1234 % 97 = 70

h(5642)= 5642 % 97 = 16



Mid Square Method:

It works in two steps:

Step1: Square the value of the key. That is find k^2

Step2: Extract the middle r digits of the result obtained in step1.

In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as

h(k)=s where s is obtained by selecting r digits from k^2

Example: Calculate the hash value for keys 1234 and 5642 using mid-square method. The hash tale has 100 memory locations.

Solution: k=1234, k^2 = 152**<u>27</u>**56 h(1234)=27

k=5642, k^2 = 3183**<u>21</u>**64 h(5642)=21

observe that the 3rd and 4th digit starting from the right are chosen.

3) Folding Method:

It works in the following steps.

Step1: Divide the key value into a number of parts. (i.e.) k divided into parts k1,k2,k3,…kn where each part has the same number of digits, except the last part which may have lesser digits than the other parts.

Step2: Add the individual parts. i.e. obtain the sum of k1+k2+k3…+kn. The has value is produced by ignoring the last carry if any.

Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678,321, and 34567.

Solution:

Key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56, and 7
Sum	**<u>1</u>**34	33	97
Hash value	34(ignore carry)	33	97

Collisions:

Collisions occur when the hash function maps two different key to the same location. Obviously, two records cannot be stored in the same location.

Therefore, a method used to solve the problem of collision also called collision resolution technique.

The two most popular methods of resolving collisions are :

1. Open addressing
2. Chaining

1. Collision resolution by open Addressing:

   Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position.

   In this technique the hash table contains two types of values. Sentinel values(eg-1) and data values. The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

   When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, the n the location is free and the data value can be stored in it.

   However, if the location is already has some data value stored in in it, then other slots are examined systematically in the forward direction to find a free slot.

   If even a single free location is not found, then we have an overflow condition.

   The process of examining memory locations in the hash table is called probing.

Open addressing technique can be implemented using

i) Linear probing
ii) Quadratic probing
iii) Double hashing
iv) Rehashing

<u>Linear probing:</u> In this, we linearly probe for next slot. The following hash function is used to resolve the collision.

h(k,i)=(h(k)+i) mod m

ex: I = 1,2,3….

Example: consider has functions as key mod 7 and sequence of keys are 50,700,76,85,92,73,101.

Index	val	Index	val	Index	val	Index	val	Index	val	Index	val	Index	val	Index	val	Index	val
0	-1	0	-1	0	700	0	700	0	700	0		0	700	0	700	0	700
1	-1	1	-1	1	50	1	50	1	50	1		1	50	1	50	1	50
2	-1	2	-1	2	-1	2	-1	2	85	2		2	85	2	85	2	85
3	-1	3	-1	3	-1	3	-1	3	-1	3		3	92	3	92	3	92
4	-1	4	-1	4	-1	4	-1	4	-1	4		4	-1	4	73	4	73
5	-1	5	-1	5	-1	5	-1	5	-1	5		5	-1	5	-1	5	101
6	-1	6	-1	6	-1	6	76	6	76	6		6	76	6	76	6	76
								collistion occurred insert 85at next free slot				collistion occurred insert 92at next free slot				collistion occurred insert101at next free slot	

ii)Quadratic Probing:

We look for i^2 th slot in the hash table. The following hash function is used to resolve the collection.

$h(k,i)=(h,1k)+i^2)$ mod m.

i=1,2,3 …..

iii) Double hashing: We use another hash function has2(x) and look for i*hash2(x). slot in the rotation. The following hash function is used to resolve the collision.

h(k,i)=h(k)+i*hash2(x) mod m.

Where I =1,2,3…

Iv)Rehashing: When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

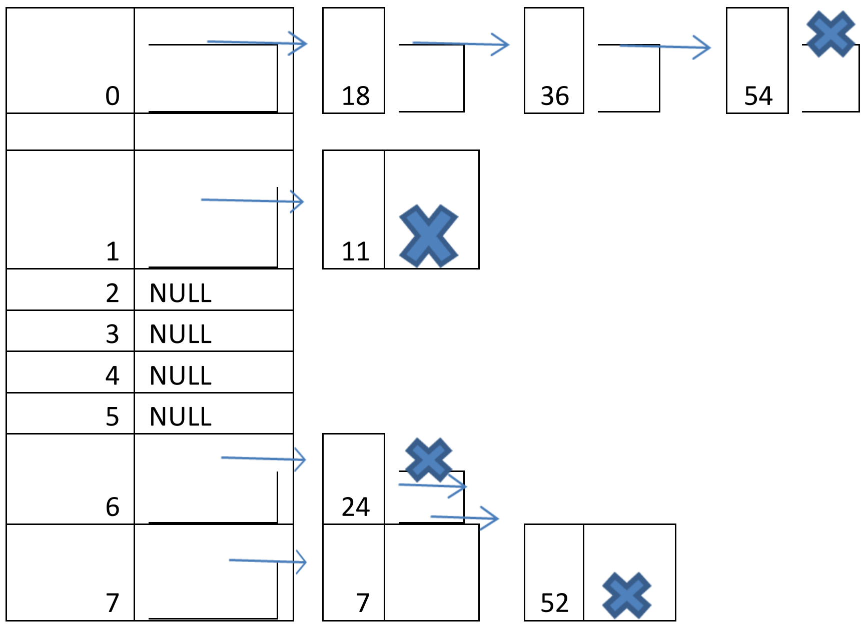
Through rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently.

<u>Collision resolution by changing</u>

In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location.

That is location 1 in the hash tale points to the head of the linked list of all the key values hashed to 1 then location 1 in hash table contains NULL.

The following Fig. show how the key values are mapped to a location in the hash table and stored in a linked list that corresponds to that location.


0
1
2
3
4
5
6
7
NULL
NULL
NULL
NULL
18
36
54
11
24
7
52