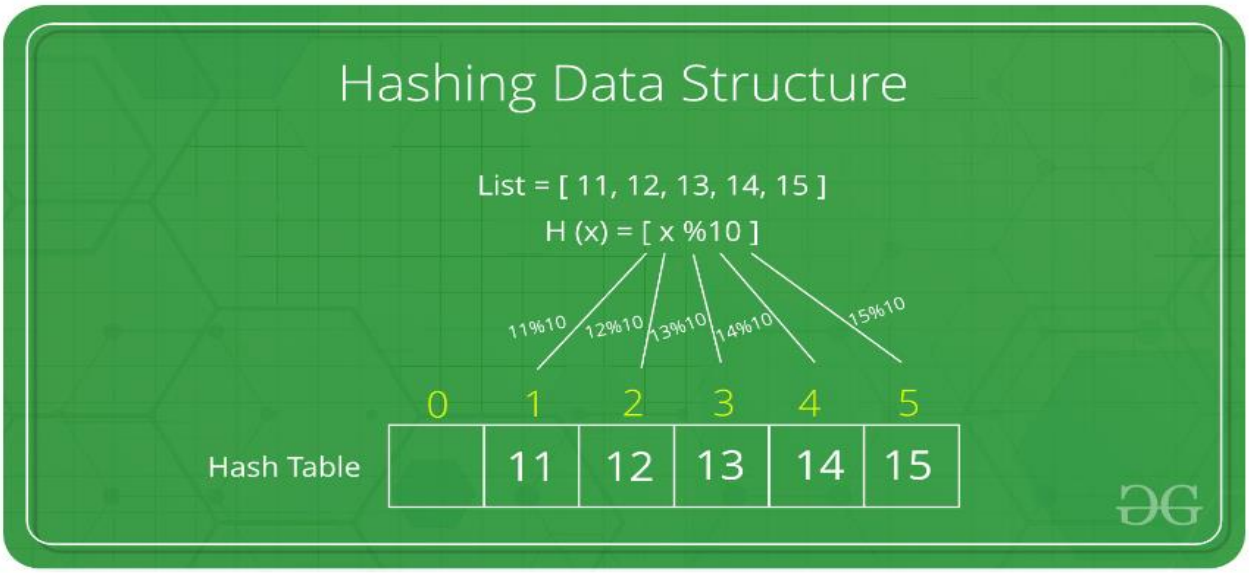
UNIT - 5

HASHING

What is Hashing?

Hashing is a fundamental data structure that **efficiently stores and retrieves data in a way that allows for quick access**. It involves **mapping data to a specific index in a hash table using a hash function**, enabling fast retrieval of information based on its **key**. This method is commonly used in **databases**, caching systems, and various programming applications to **optimize search and retrieval operations**. Hash data structure can **store the data and search in it** in **constant time** i.e. **O(1)** which makes it **very efficient**.

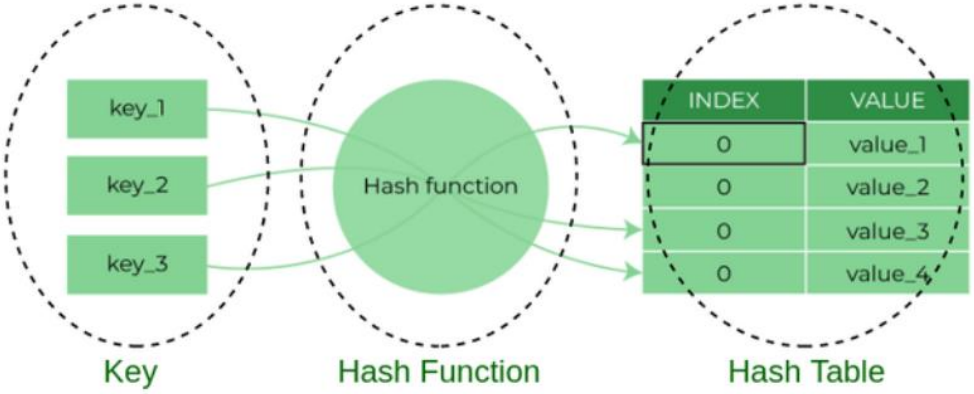


Hashing Data Structure

Components of Hashing

There are majorly three components of hashing:

1. **Key**: A **Key** can be **any string or integer** which is **fed as input in the hash function** the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function**: The **hash function receives the input key and returns the index of an element in an array called a hash table**. The index is known as the **hash index**.
3. **Hash Table**: Hash table is a data structure that **maps keys to values using a special function called a hash function**. Hash stores the data in an **associative manner** in an array where **each data value has its own unique index**.



Components of Hashing

Advantages of Hashing

- **Key-value support**: Hashing is **ideal** for **implementing key-value data structures**.
- **Fast data retrieval**: Hashing allows for **quick access to elements with constant-time complexity**.
- **Efficiency**: **Insertion, deletion, and searching operations** are **highly efficient**.
- **Memory usage reduction**: Hashing requires **less memory** as it **allocates a fixed space** for **storing elements**.
- **Scalability**: Hashing **performs well with large data sets**, maintaining **constant access time**.
- **Security and encryption**: Hashing is essential for **secure data storage and integrity verification**.

What is a Hash function?

A **Hash Function** is a **function that converts a given numeric or alphanumeric key to a small practical integer value**. The **mapped integer value** is **used** as an **index** in the **hash table**. In simple terms, a hash function **maps** a significant number or string to a small integer that can be used as the **index** in the hash table.

Types of Hash functions:

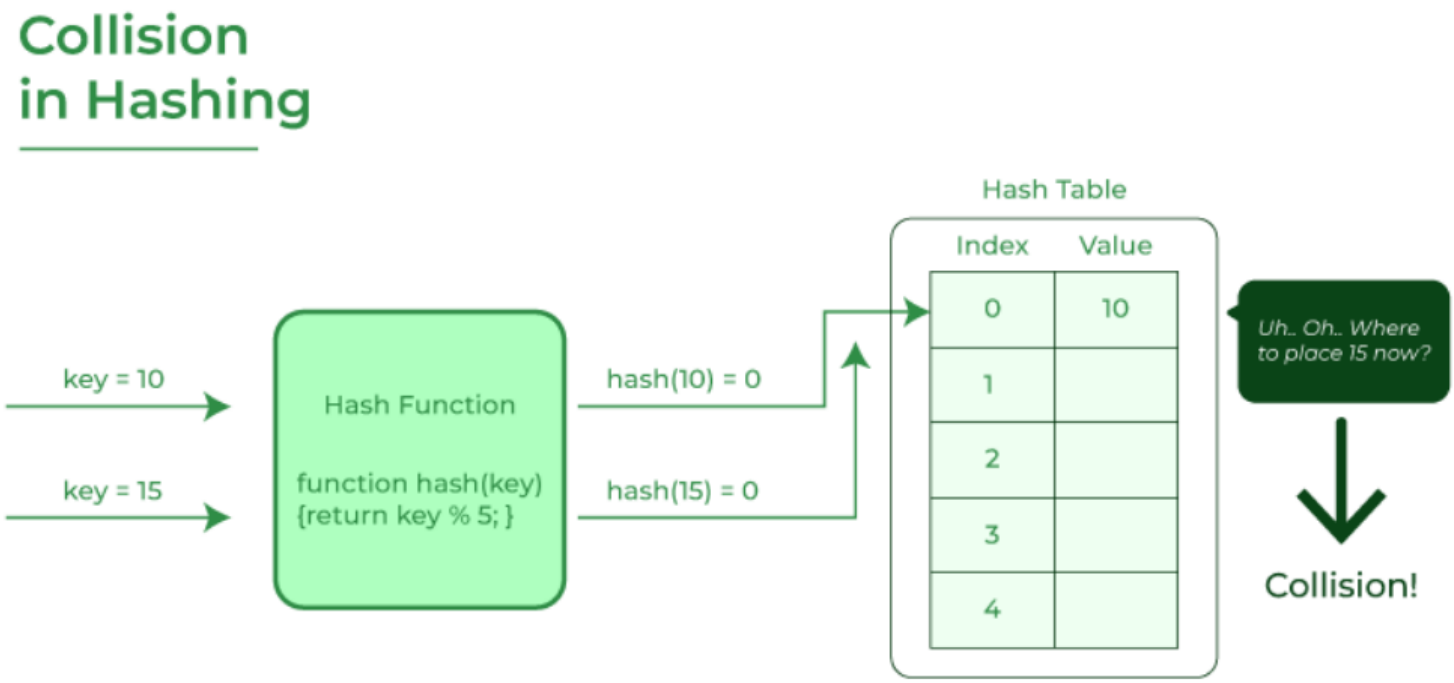
1.Division Method
2.Mid Square Method
3.Digit Folding Method
4.Multiplication Method

A Good Hash Function should have the following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys(Each table position is equally likely for each).
3. Should minimize collisions.
4. Should have a low load factor(number of items in the table divided by the size of the table).

What is collision?

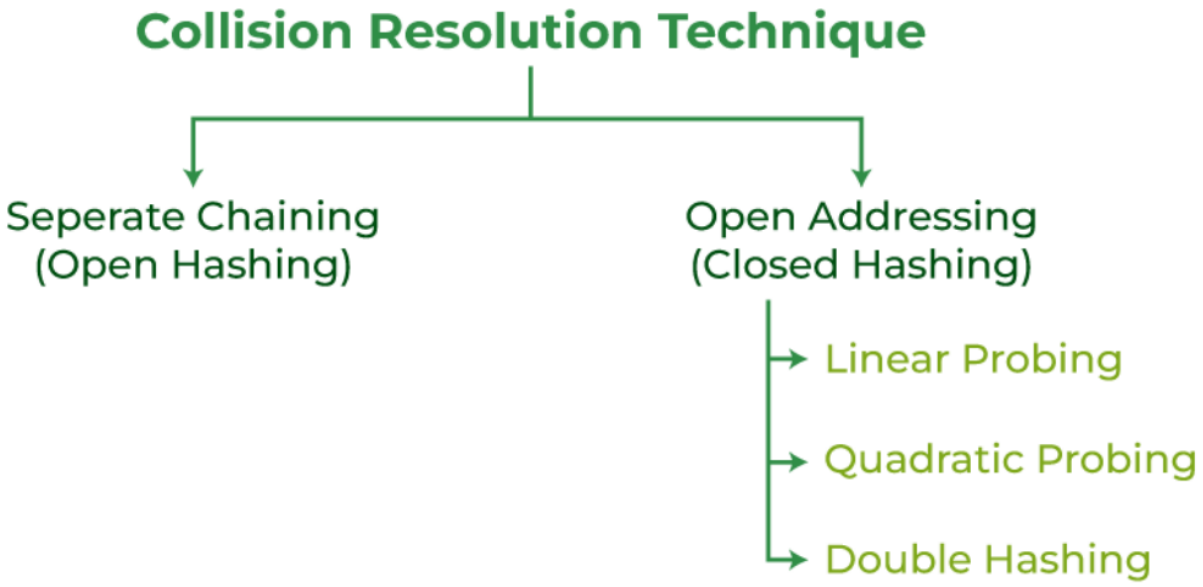
The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The **situation where the newly inserted key maps to an already occupied**, and it must be handled using some collision handling technology.



How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining (Open Hashing)
2. Open Addressing (Closed Hashing)

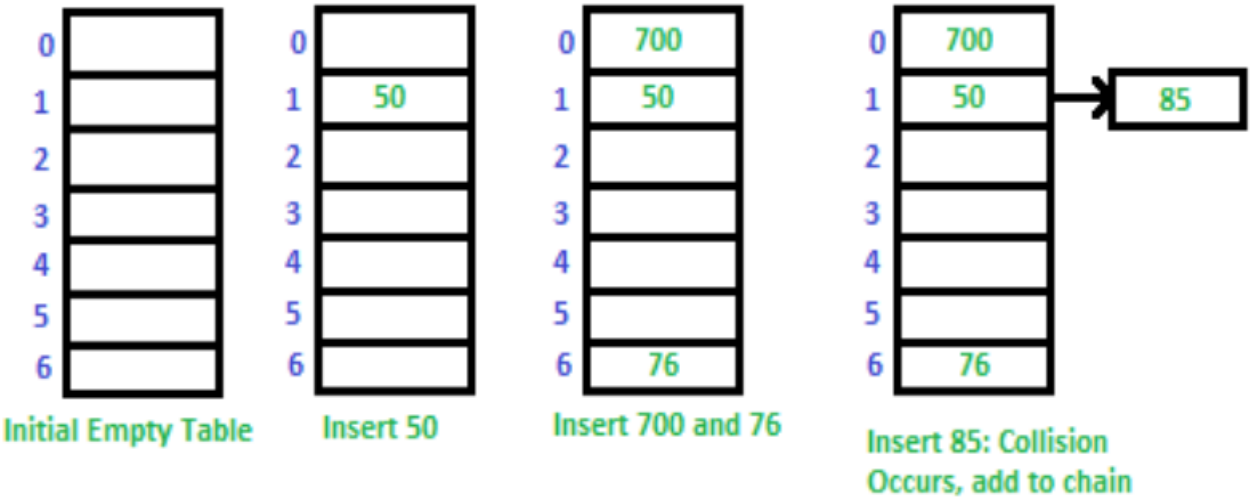


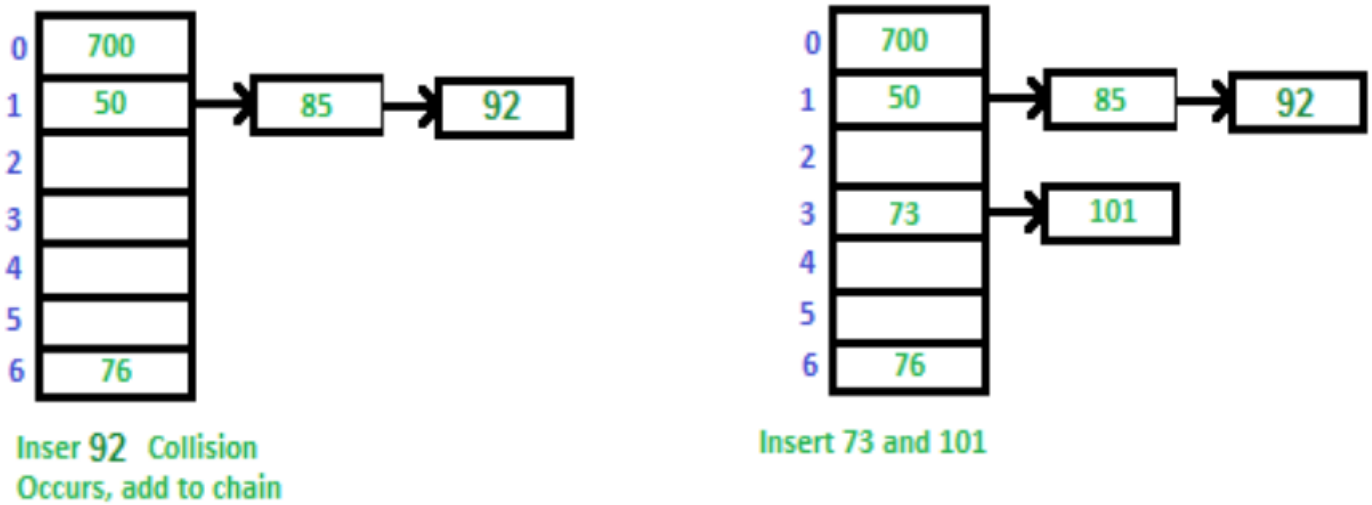
1. Seperate Chaining

Separate Chaining is a collision resolution strategy employed in hash tables, which are data structures used to efficiently store and retrieve key-value pairs.When **two or more keys hash to the same index**, a **collision occurs.Instead of storing all conflicting elements directly in the same slot**, separate chaining tackles collisions by **maintaining a linked list** (or another data structure) **at each hash table index**.

In this approach, **elements with identical hash values are appended to the linked list corresponding to their hash index**.When multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.This way, **multiple elements can coexist at the same hash table index without overwriting each other**. While separate chaining ensures effective handling of collisions, its efficiency depends on the quality of the underlying linked list implementation and the overall load factor of the hash table.

Example: Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101





Advantages

- **Simple to implement**.
- **Hash table never fills up**, we can always add more elements to the chain.
- **Less sensitive** to the hash function or load factors.
- It is **mostly used when it is unknown how many and how frequently keys may be inserted or deleted**.

Disadvantages

- The **cache performance of chaining is not good** as keys are stored using a linked list.**Open addressing provides better cache performance** as everything is stored in the same table.
- **Wastage of Space** (Some Parts of the hash table are never used)
- **If the chain becomes long**, then **search time can become O(n)** in the worst case
- **Uses extra space for links**

2.Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table itself**. So at any point, the **size of the table must be greater than or equal to the total number of keys** (Note that we can increase table size by copying old data if needed). This approach is also known as **closed hashing**.

The following techniques are used in open addressing:

- Linear probing
- Quadratic probing
- Double hashing

2a.Linear Probing

In **linear probing**, the algorithm simply looks for the next available slot in the hash table and places the collided key there. If that slot is also occupied, the algorithm continues searching for the next available slot until an empty slot is found. This process is repeated until all collided keys have been stored. Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.

Algorithm:

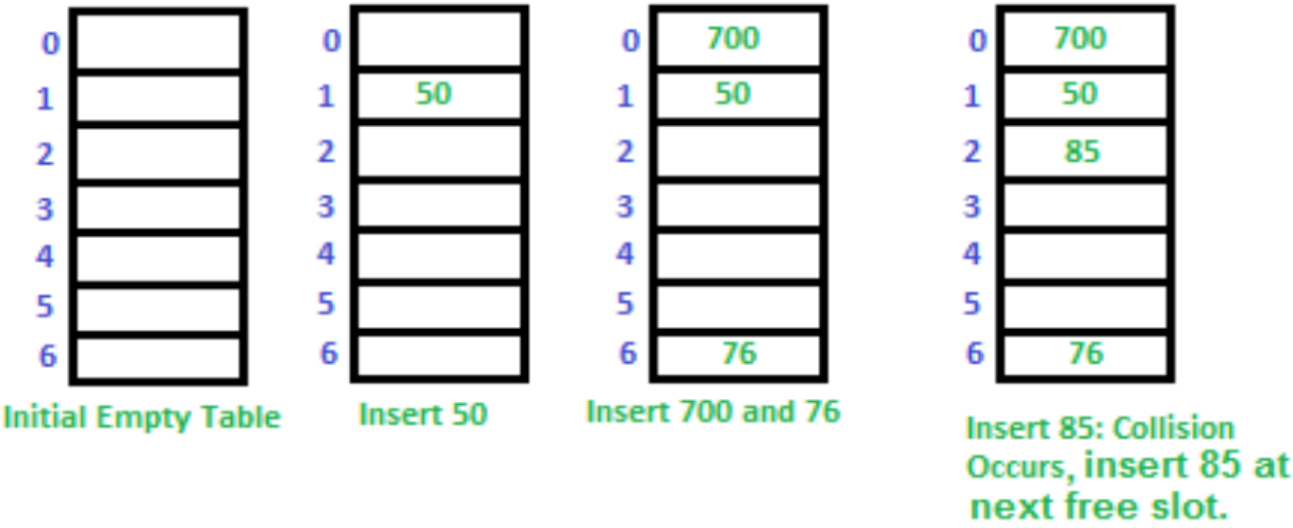
- Calculate the hash key (key = data % size)
- Check, if hashTable[key] is empty
- store the value directly by hashTable[key] = data
- If the hash index already has some value then
- check for next index using key = (key+1) % size
- Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.
- Do the above process till we find the space.

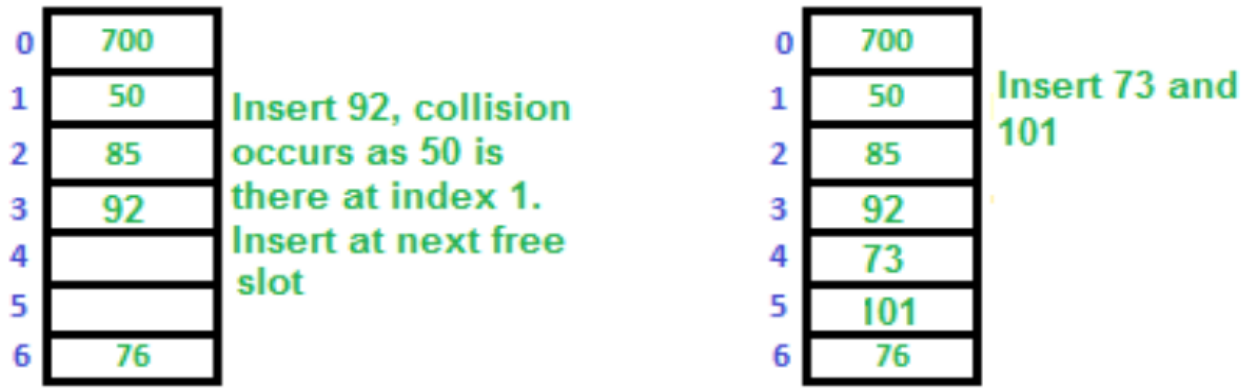
Applications of linear probing:

- **Symbol tables**: Linear probing is commonly used in symbol tables, which are used in compilers and interpreters to store variables and their associated values. Since symbol tables can grow dynamically, linear probing can be used to handle collisions and ensure that variables are stored efficiently.
- **Caching**: Linear probing can be used in caching systems to store frequently accessed data in memory. When a cache miss occurs, the data can be loaded into the cache using linear probing, and when a collision occurs, the next available slot in the cache can be used to store the data.
- **Databases**: Linear probing can be used in databases to store records and their associated keys. When a collision occurs, linear probing can be used to find the next available slot to store the record.
- **Compiler design**: Linear probing can be used in compiler design to implement symbol tables, error recovery mechanisms, and syntax analysis.
- **Spell checking**: Linear probing can be used in spell-checking software to store the dictionary of words and their associated frequency counts. When a collision occurs, linear probing can be used to store the word in the next available slot.

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101,

which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.We can define the hash function as per our choice if we want to create a hash table,although it is fixed internally with a pre-defined formula

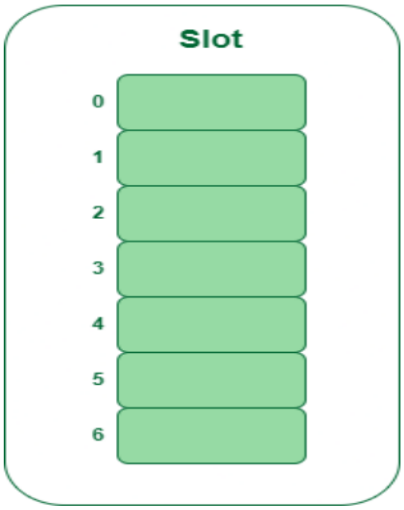




2b. Quadratic Probing

In **quadratic probing**, the algorithm searches for slots in a more spaced-out manner. When a collision occurs, the algorithm looks for the next slot using an equation that involves the original hash value and a quadratic function. If that slot is also occupied, the algorithm increments the value of the quadratic function and tries again. This process is repeated until an empty slot is found. Quadratic probing lies between the two in terms of cache performance and clustering.

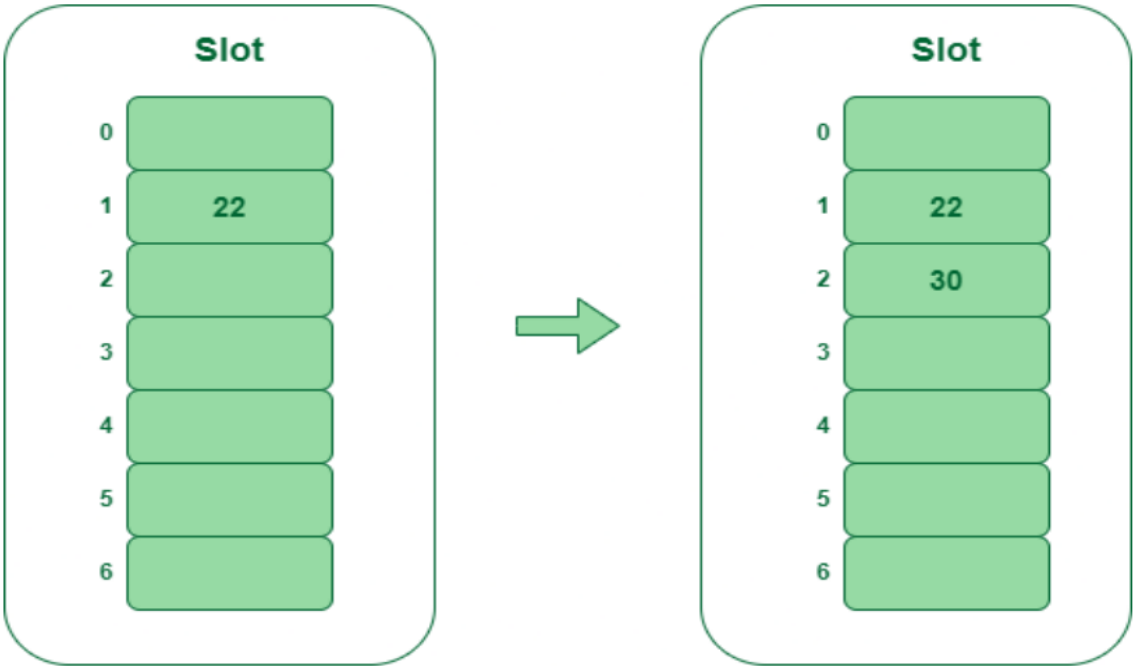
Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30, and 50.
Step 1: Create a table of size 7.



Hash table

Step 2 – Insert 22 and 30

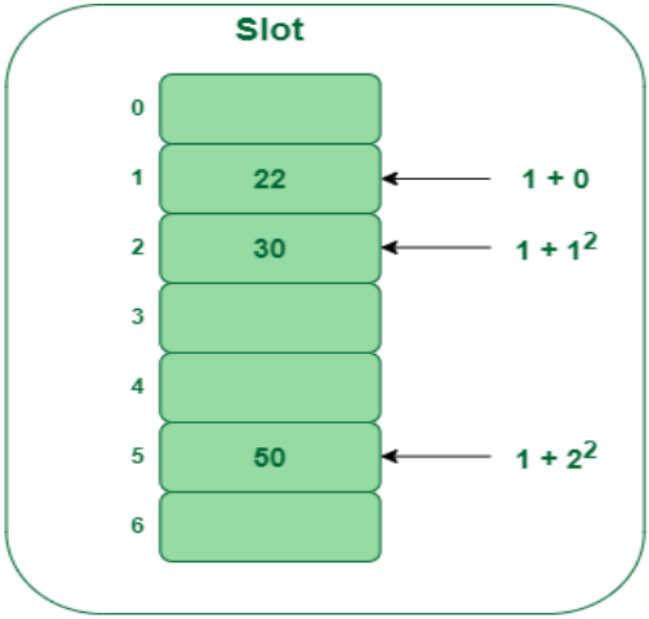
- Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
- Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Insert keys 22 and 30 in the hash table

Step 3: Inserting 50

- Hash(50) = 50 % 7 = 1
- In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
- Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
- Now, cell 5 is not occupied so we will place 50 in slot 5.



Insert key 50 in the hash table

2c. Double Hashing

In **double hashing**, the algorithm uses a second hash function to determine the next slot to check when a collision occurs.

- The first hash function is **h1(k)** which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key.
- But in case the location is occupied (collision) we will use secondary hash-function **h2(k)** in combination with the first hash-function **h1(k)** to find the new location on the hash table.

This combination of hash functions is of the form

h(k, i) = (h1(k) + i * h2(k)) % n

where

- i is a non-negative integer that indicates a collision number,
- k = element/key which is being hashed
- n = hash table size.

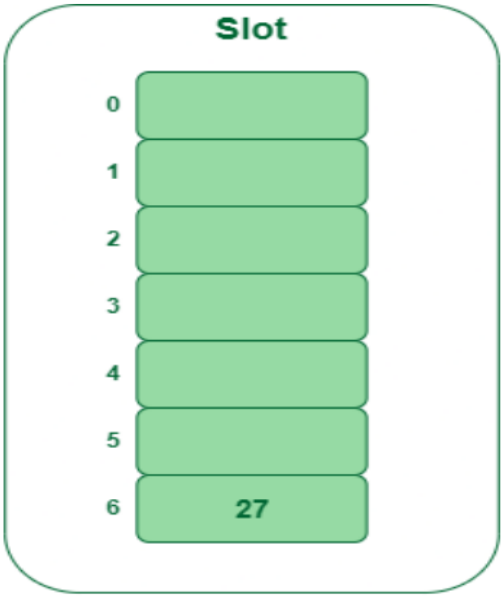
Advantages of Double hashing

- The advantage of Double hashing is that it is one of the best forms of probing, producing a uniform distribution of records throughout a hash table.
- This technique does not yield any clusters.
- It is one of the effective methods for resolving collisions.

Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Step 1: Insert 27

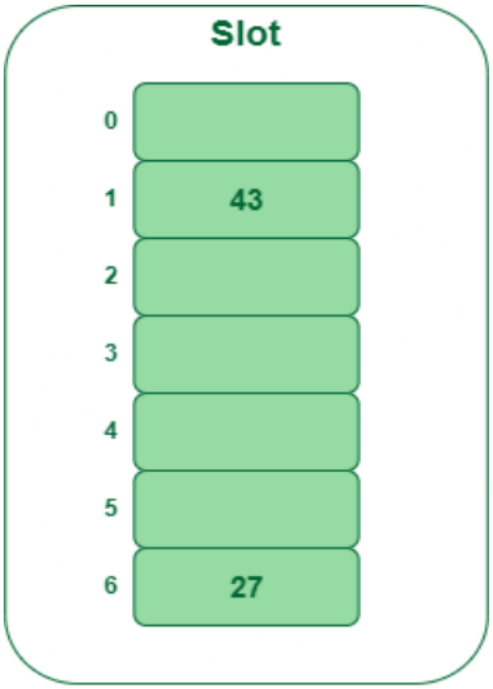
- 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



Insert key 27 in the hash table

Step 2: Insert 43

- 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.

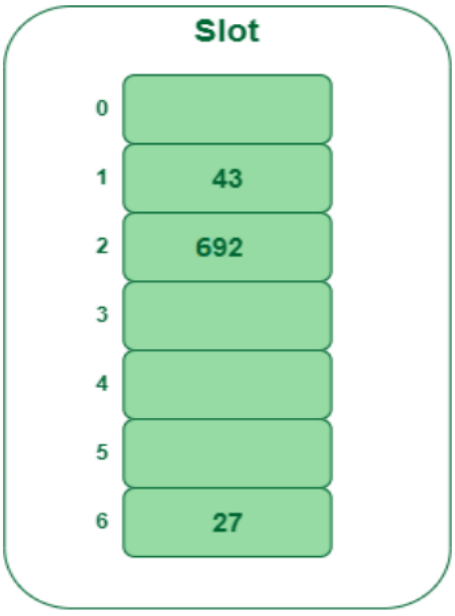


Insert key 43 in the hash table

Step 3: Insert 692

- 692 % 7 = 6, but location 6 is already being occupied and this is a collision
- So we need to resolve this collision using double hashing.

h_{new} = [h1(692) + i * (h2(692)] % 7
= [6 + 1 * (1 + 692 % 5)] % 7
= 9% 7
= 2
Now, as 2 is an empty slot,
so we can insert 692 into 2nd slot.

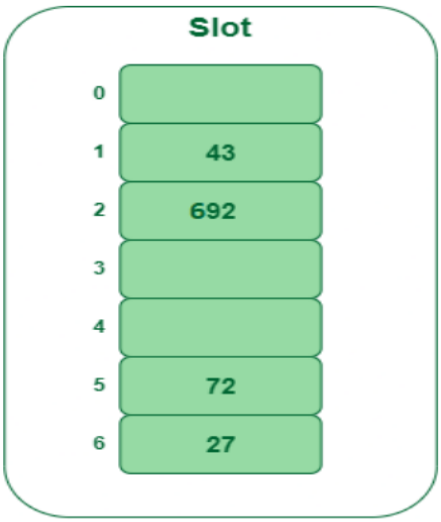


Insert key 692 in the hash table

Step 4: Insert 72

- 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
- So we need to resolve this collision using double hashing.

h_{new} = [h1(72) + i * (h2(72)] % 7
= [2 + 1 * (1 + 72 % 5)] % 7
= 5 % 7
= 5,
Now, as 5 is an empty slot,
so we can insert 72 into 5th slot.



Insert key 72 in the hash table

Difference between Separate Chaining and Open Addressing

S.no.	Separate Chaining	Open Addressing
1.	Chaining is **Simpler to implement**.	Open Addressing **requires more computation**.
2.	In chaining, **Hash table never fills up,** we can always add more elements to chain.	In open addressing, **table may become full**.
3.	Chaining is **Less sensitive to the hash function or load factors**.	Open addressing **requires extra care to avoid clustering and load factor**.
4.	Chaining is mostly **used when it is unknown how many and how frequently keys may be inserted or deleted**.	Open addressing is **used when the frequency and number of keys is known**.
5.	**Cache performance** of chaining is **not good as keys are stored using linked list**.	Open addressing provides **better cache performance as everything is stored in the same table**.
6.	**Wastage of Space** (Some Parts of hash table in chaining are never used).	In Open addressing, **a slot can be used even if an input doesn't map to it**.
7.	Chaining **uses extra space for links**.	**No links** in Open addressing

Note: **Cache performance of chaining is not good** because when we traverse a **Linked List**, we are basically jumping from one node to another, all across the computer's memory. For this reason, the **CPU cannot cache the nodes which aren't visited yet**, this doesn't help us. But **with Open Addressing, data isn't spread**, so if the **CPU detects that a segment of memory is constantly being accessed, it gets cached for quick access**.

What is meant by Load Factor in Hashing?

The load factor of the hash table can be defined as the **number of items the hash table contains divided by the size of the hash table**. Load factor is the **decisive parameter** that is **used when we want to rehash the previous hash function or want to add more elements to the existing hash table**.
It helps us in **determining the efficiency of the hash function** i.e. it tells whether the **hash function** which we are using is **distributing the keys uniformly or not in the hash table**.

Load Factor = Total elements in hash table / Size of hash table

What is Rehashing?

Rehashing is the **process of increasing the size of a hashmap** and **redistributing the elements to new buckets based on their new hash values**. It is done to **improve the performance of the hashmap and to prevent collisions caused by a high load factor**.

Rehashing is needed in a hashmap to prevent collision and to maintain the efficiency of the data structure. When a **hashmap becomes full**, the **load factor (i.e., the ratio of the number of elements to the number of buckets) increases**. As the load factor increases, the number of collisions also increases, which can lead to poor performance. **To avoid this**, the **hashmap can be resized** and the **elements can be rehashed to new buckets**, which decreases the load factor and reduces the number of collisions.

How Rehashing is done?

- **For each addition** of a new entry to the map, **check** the **load factor**.
- If it's **greater than** its **pre-defined value (or default value of 0.75 if not given), then Rehash.**
- For Rehash, **make** a **new array of double the previous size and make it the new bucketarray**.
- Then **traverse** to **each element in the old bucketArray and call the insert()** for each so as to **insert it into the new larger bucket array**.

Applications of Hash Data structure

- Hash is used in **databases for indexing**.
- Hash is used in **disk-based data structures**.
- In some programming languages like **Python**, **JavaScript**, hash is used to **implement objects**.

Real-Time Applications of Hash Data structure

- Hash is used for **cache mapping for fast access to the data**.
- Hash can be used for **password verification**.
- Hash is used in **cryptography as a message digest**.
- **Rabin-Karp algorithm** for **pattern matching in a string**.
- Calculating the **number of different substrings of a string**.

Advantages of Hash Data structure

- Hash provides **better synchronization** than other data structures.
- Hash tables are **more efficient than search trees or other data structures.**
- Hash provides **constant time for searching, insertion, and deletion operations** on average.

Disadvantages of Hash Data structure

- Hash is **inefficient** when there are **many collisions**.
- Hash **collisions are practically not avoided** for a large set of possible keys.
- Hash does **not allow null values**.

What is Extendible Hashing?

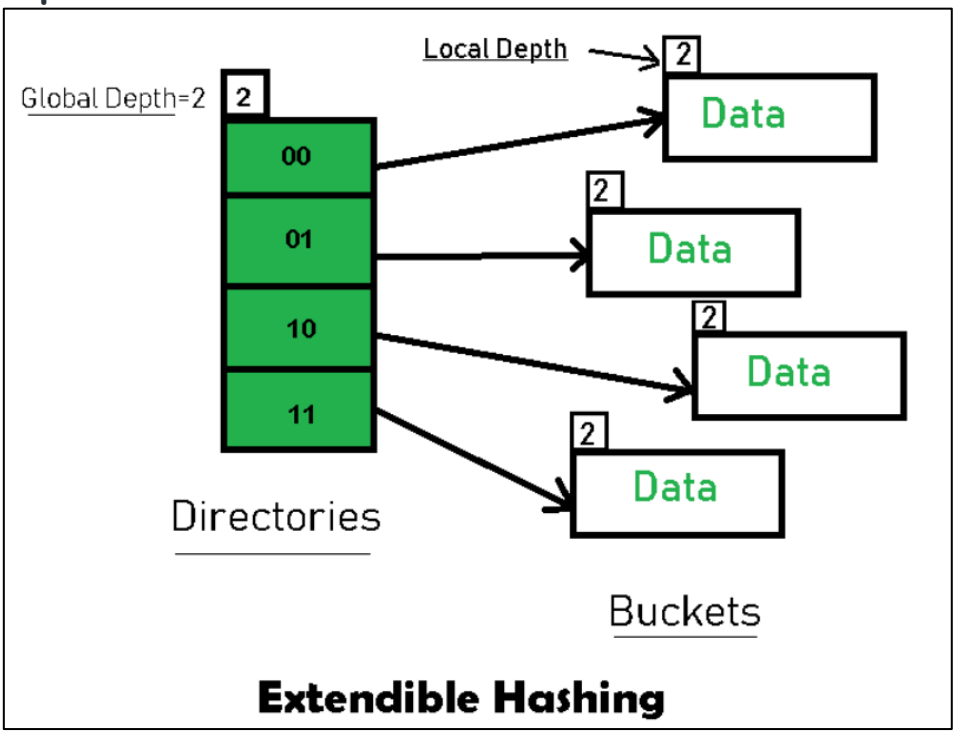
Extendible Hashing is a **dynamic hashing method wherein directories, and buckets are used to hash data**. It is an **aggressively flexible method in which the hash function also experiences dynamic changes**.

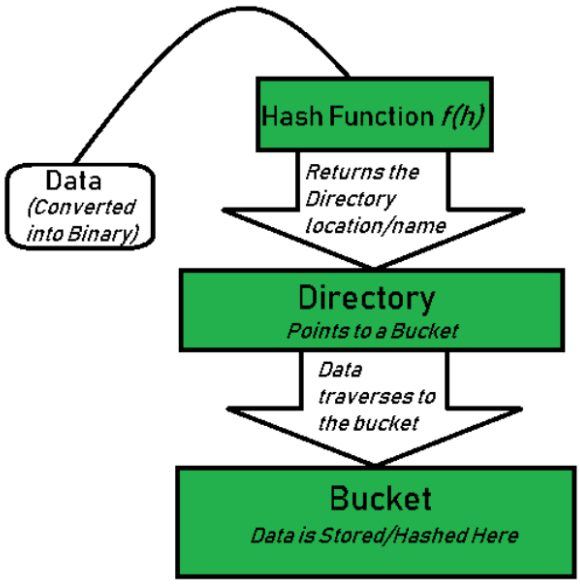
In extendible hashing, the hash function maps keys to a **binary address**, and the **address space dynamically grows as needed**. The address space is divided into a **directory** and **buckets**.

The **key feature** of extendible hashing is the **ability to dynamically split or merge buckets based on the number of keys in them. When a new key is inserted, if the corresponding bucket is full, the bucket is split into two, and the directory is updated accordingly. If the directory becomes full, it is doubled in size, and existing entries are redistributed.**

- **Directories**: These containers **store pointers to buckets**. Each directory is given a **unique id which may change each time when expansion takes place**. The hash function returns this directory id which is used to navigate to the appropriate bucket. **Number of Directories = 2^Global Depth**.
- **Buckets**: They **store the hashed keys**. Directories point to buckets. A bucket **may contain more than one pointers to it** if its local depth is less than the global depth.

- **Global Depth**: It is **associated with the Directories**. They denote the **number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.**
- **Local Depth**: It is the same as that of Global Depth except for the fact that Local Depth is **associated with the buckets** and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. **Local Depth is always less than or equal to the Global Depth.**
- **Bucket Splitting**: When the **number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.**
- **Directory Expansion**: Directory Expansion Takes place **when a bucket overflows.** Directory Expansion is performed when the **local depth of the overflowing bucket is equal to the global depth.** Directory expansion will **double the number of directories present** in the hash structure.





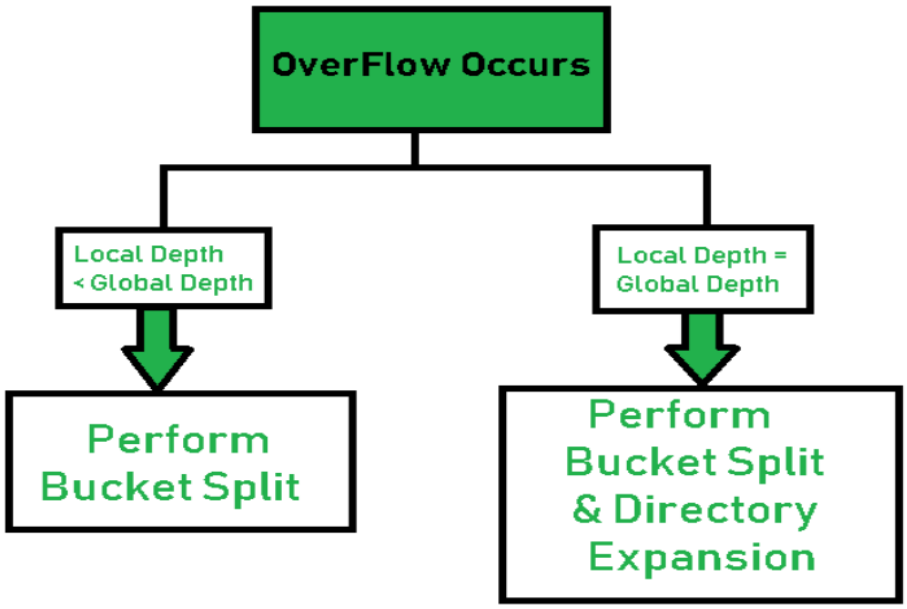
Basic Working of Extendible Hashing

Tackling Over Flow Condition during Data Insertion:

Many times, while **inserting data in the buckets, it might happen that the Bucket overflows**. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.

First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.

- **Case1**: If the **local depth of the overflowing Bucket is equal to the global depth**, then **Directory Expansion as well as Bucket Split needs to be performed**. Then increment the global depth and the local depth value by 1 and assign appropriate pointers.
- **Case2**: In case the **local depth is less than the global depth**, then **only Bucket Split takes place**. Then increment only the local depth value by 1 and assign appropriate pointers.



Advantages of Extendible Hashing

- **Data retrieval is less expensive** (in terms of computing).
- No problem of Data-loss since the **storage capacity increases dynamically**.
- With dynamic changes in hashing function, **associated old values are rehashed** w.r.t the new hash function.

Limitations of Extendible Hashing

- The **directory size may increase significantly** if several records are hashed on the same directory while keeping the **record distribution non-uniform**.
- **Size of every bucket is fixed**.
- **Memory is wasted in pointers** when the global depth and local depth difference becomes drastic.
- This method is **complicated to code**.

DOUBLE HASHING VS REHASHING

Feature	Double Hashing	Rehashing
Technique	Collision resolution by using a secondary hash function to calculate probe sequence intervals	Collision resolution by dynamically resizing and rehashing the hash table
Implementation	Secondary hash function applied to calculate probe sequence intervals	Resize the hash table, recalculate hash values, and reinsert elements
Memory Usage	No additional memory allocation required	Requires additional memory allocation for the new hash table
Performance	Additional computation for secondary hash function	Memory allocation and copying of elements
Flexibility	Suitable for fixed-size hash tables	Adaptable to dynamic datasets
Handling Clustering	Helps reduce clustering with well-chosen hash functions	Can alleviate clustering by resizing the table
Usage	Suitable for scenarios with a fixed-size hash table	Preferred for dynamic datasets with changing load factors

Disjoint Sets

What is Relation?

A relation is a **subset of the cartesian product of a set with another set**. A relation contains **ordered pairs of elements of the set** it is defined on.

What is Equivalence Relation?

A relation R on a set A is called an equivalence relation if it is

1. **Reflexive Relation**: $(a, a) \in R \ \forall \ a \in A$, i.e. aRa for all $a \in A$.
2. **Symmetric Relation**: $\forall \ a, b \in A, (a, b) \in R \leftrightarrow (b, a) \in R$.
3. **Transitive Relation**: $\forall \ a, b, c \in A$, if $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$.

where R is a subset of $(A \times A)$, i.e. the cartesian product of set A with itself.

Example: Consider set **A = {a, b}**
R = {(a, a), (a, b), (b, a)} is **not equivalence** as for (b, b) tuple is not present.
R = {(a, a), (a, b), (b, a), (b, b)} is an **equivalence** relation.

Properties of Equivalence Relation

1. Empty relation on a non-empty set is never equivalence.
2. Universal relation over any set is always equivalence.

How to verify an Equivalence Relation?

The process of identifying/verifying if any given relation is equivalency:

- Check if the relation is **Reflexive.**
- Check if the relation is **Symmetric.**
- Check if the relation is **Transitive.**

Example: Consider set R = {(1, 1), (1, 3), (2, 2), (3, 3), (3, 1), (3, 2), (3, 4), (4, 4), (4, 3)}

Pairs (1, 1), (2, 2), (3,3), (4, 4) exist:

⇒ This satisfies the reflexive condition.

The symmetric condition is also satisfied.

For the pairs(1,3)and(3,4):
⇒ The relation (1, 4) does not exist
⇒ This does not satisfy the transitive condition.

So the relation is not transitive. Hence it is not equivalence.

Note: (1, 4) and (4, 1) will be inserted in R to make it an equivalence relation.

Disjoint Set Data Structures

Two sets are called **disjoint sets** if they **don't have any element in common**, the **intersection of sets is a null set**.

A data structure that **stores non overlapping or disjoint subset of elements** is called **disjoint set data structure**. The disjoint set data structure supports following operations:

- **Adding new sets** to the disjoint set.
- Merging disjoint sets to a single disjoint set using **Union operation**.
- Finding representative of a disjoint set using **Find operation**.
- **Check** if two sets are **disjoint or not**.

A **union-find algorithm** is an algorithm that performs two useful operations on such a data structure:

- **Find:** Determine which subset a particular element is in. This can determine if two elements are in the same subset.
- **Union:** Join two subsets into a single subset. Here first we have to check if the two subsets belong to the same set. If not, then we cannot perform union.

Simple union and find algorithm

The union-find algorithm, also known as the disjoint-set data structure, is a fundamental method for managing disjoint sets efficiently. It operates through two main operations: find and union.

In the find operation, the algorithm recursively traverses the parent pointers of an element until reaching the representative of its set. This representative serves as the root of the set and is used for querying purposes.

The union operation merges two disjoint sets together by making one set's representative the parent of the other set's representative. This process effectively combines the sets into a single larger set.

Implemented in a class structure, the algorithm maintains arrays for parent pointers and ranks. The parent array initially assigns each element to its own set, with the element itself as its parent. The rank array assists in optimization strategies like union by rank.

Overall, the union-find algorithm provides a basic framework for managing disjoint sets efficiently, facilitating operations like querying whether elements belong to the same set and merging sets together. It serves as a foundational tool in various applications, including graph algorithms, network connectivity problems, and more.

Smart union and path compression algorithm

The smart union and path compression algorithm is an optimized version of the basic union-find algorithm used to efficiently manage disjoint sets. In this algorithm, each element initially belongs to its own set, with itself as its parent. Additionally, each element is assigned a rank to facilitate union by rank.

During the find operation, the algorithm recursively traverses the parent pointers of an element until reaching the representative of its set. Along the way, it compresses the path by updating each node's parent pointer to directly point to the representative. This path compression flattens the tree structure, reducing the height of subsequent find operations.

In the union operation, two disjoint sets are merged together. First, the representatives of the sets containing the given elements are found. Then, based on the ranks of the representatives, one representative is chosen as the new parent. If one representative has a higher rank than the other, the smaller-ranked tree is attached to the larger-ranked tree.

By incorporating both union by rank and path compression, the smart union and path compression algorithm achieves efficient time complexity for find and union operations. The find operation typically runs in nearly constant time, while the union operation's time complexity depends on the union by rank strategy, ensuring balanced and shallow trees.

Overall, this optimized algorithm is widely used in applications requiring efficient management of disjoint sets, such as graph algorithms and network connectivity problems.

```
Let there be 4 elements 0, 1, 2, 3
Initially, all elements are single element subsets.
0 1 2 3


Do Union(0, 1)
   1   2   3
  /
 0


Do Union(1, 2)
     2   3
    /
   1
 /
0
```

```
Do Union(2, 3)
        3
       /
      2
     /
   1
 /
0
```

Let us see the above example with **union by rank**
Initially, all elements are single element subsets.
0 1 2 3

```
Do Union(0, 1)
   1   2   3
  /
 0
```

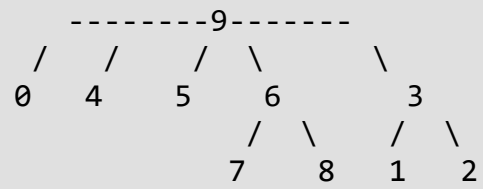
```
Do Union(1, 2)
   1    3
 /  \
0    2
```

```
Do Union(2, 3)
    1
 /  |  \
0   2   3
```

Let the subset {0, 1, .. 9} be represented as below and find() is called for element 3.

```
             9
         /   |   \
        4    5    6
       /         /  \
      0         7    8
     /
    3
   / \
  1   2
```

When *find()* is called for 3, we traverse up and find 9 as representative
of this subset. With path compression, we also make 3 and 0 as the child of 9 so
that when find() is called next time for 0, 1, 2 or 3, the path to root is reduced.



UNIT - 3

Difference Between BFS and DFS

Basis	BFS	DFS
Data Structure	BFS(Breadth First Search) uses **Queue** data structure for finding the shortest path.	DFS(Depth First Search) uses **Stack** data structure.
Definition	BFS is a traversal approach in which we first traverse through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
Conceptual Difference	BFS builds the tree **level by level**.	DFS builds the tree **sub-tree by sub-tree**.
Approach used	It works on the concept of **FIFO** (First In First Out).	It works on the concept of **LIFO** (Last In First Out).
Suitable for	BFS is more suitable for **searching vertices closer to the given source**.	DFS is more suitable for **searching vertices away from the given source**.
Applications	BFS is used in various applications such as **bipartite graphs**, **shortest paths**, etc.	DFS is used in various applications such as **acyclic graphs** and **finding strongly connected components** etc.
Technique	It is a **vertex-based technique** to find the shortest path in a graph.	It is an **edge-based technique** because the vertices along the edge are explored first from the starting to the end node.

Applications of Graphs

- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.
- In **mapping system** we use graph. It is useful to find out which is an excellent place from the location as well as your nearby location. In GPS we also use graphs.
- **Facebook** uses graphs. Using graphs suggests mutual friends. it shows a list of the f following pages, friends, and contact list.
- **Microsoft Excel** uses DAG means Directed Acyclic Graphs.
- In the **Dijkstra algorithm**, we use a graph. we find the smallest path between two or many nodes.
- On **social media** sites, we use graphs to track the data of the users. liked showing preferred post suggestions, recommendations, etc.

Single-Source Shortest Path (SSSP) Algorithm:

SSSP algorithms aim to find the shortest path from a single source vertex to all other vertices in a graph. One of the most commonly used SSSP algorithms is Dijkstra's algorithm. Dijkstra's algorithm works by iteratively selecting the vertex with the smallest tentative distance from the source and relaxing the distances of its neighboring vertices if a shorter path is found. This process continues until all vertices have been visited or until the shortest distance to the target vertex is found. Dijkstra's algorithm guarantees finding the shortest path in graphs with non-negative edge weights. Other SSSP algorithms include Bellman-Ford algorithm, which can handle graphs with negative edge weights (but not negative cycles), and Bidirectional Dijkstra's algorithm, which explores paths from both the source and target vertices simultaneously.

Dijkstra Algorithm

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

*Dijkstra's algorithm can **work on both directed graphs and undirected graphs** as this algorithm is designed to work on any type of graph as long as it meets the requirements of having non-negative edge weights and being connected.*

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

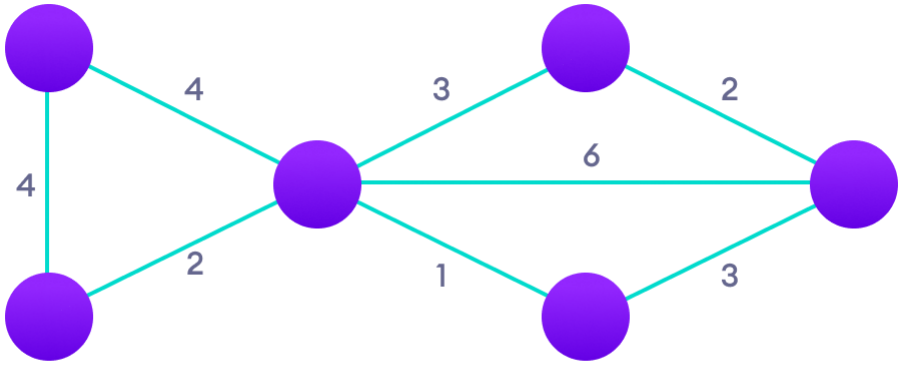
Algorithm for Dijkstra's Algorithm:

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

Dijkstra's Algorithm Applications

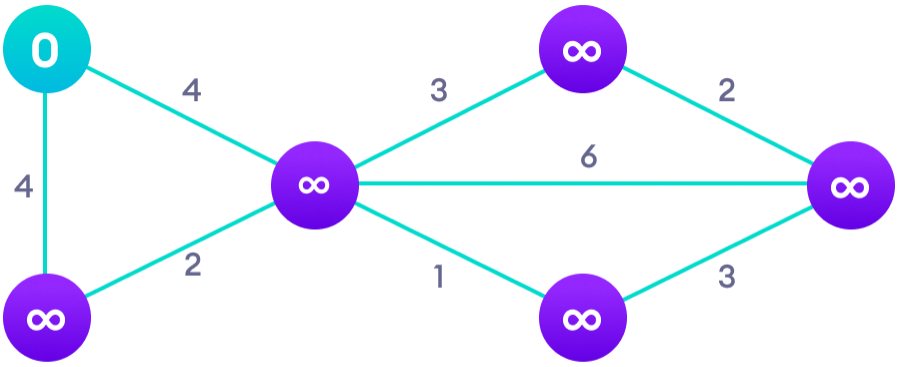
1. To find the shortest path between two points
2. In social networking applications
3. In a telephone network
4. To find the locations in the map

Example of Dijkstra's Algorithm :-



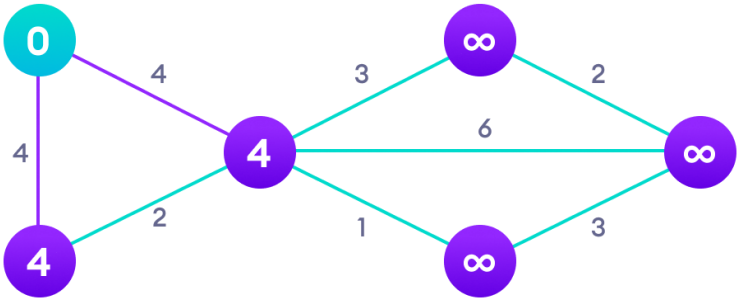
Step: 1

Start with a weighted graph



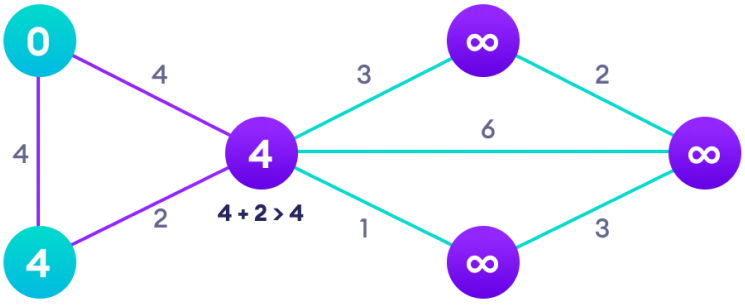
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



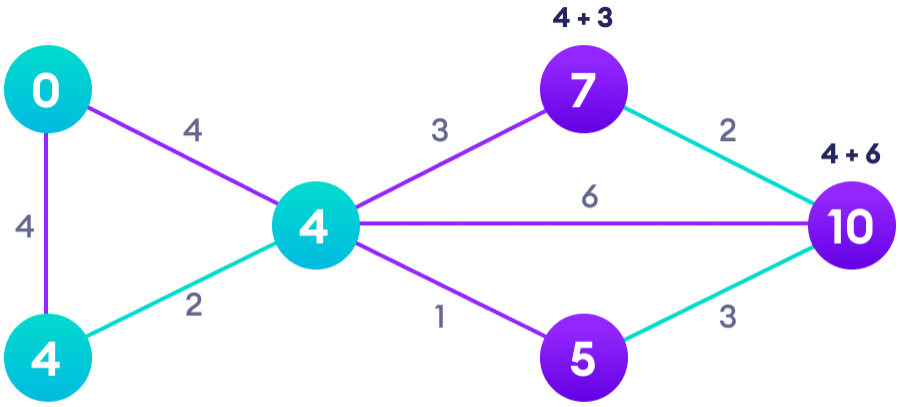
Step: 3

Go to each vertex and update its path length



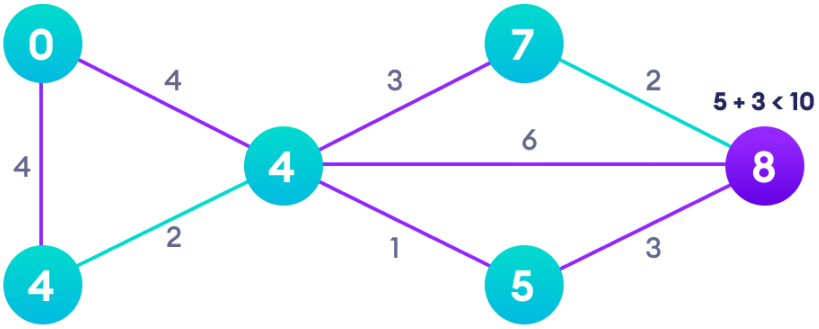
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



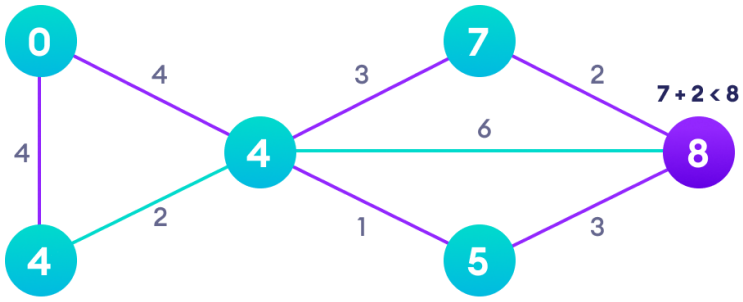
Step: 5

Avoid updating path lengths of already visited vertices



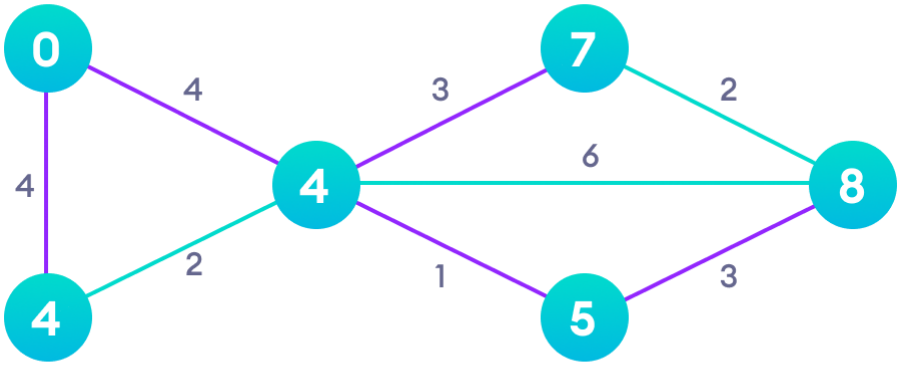
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Bellman-Ford Algorithm

Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not. In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

*The Bellman-Ford algorithm's primary principle is that it starts with a single source and calculates the distance to each node. The distance is initially unknown and assumed to be infinite, but as time goes on, the algorithm relaxes those paths by identifying a few shorter paths. Hence it is said that Bellman-Ford is based on "**Principle of Relaxation**".*

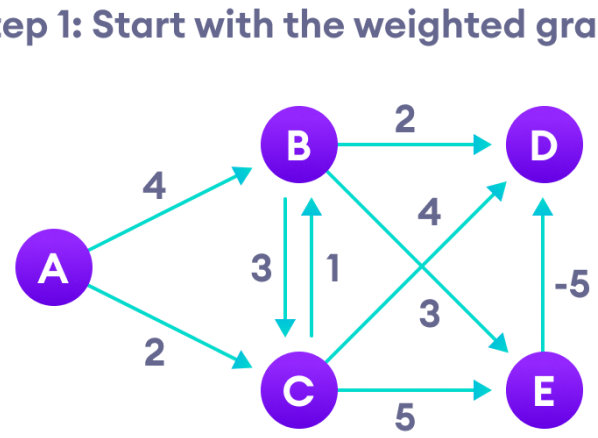
Principle of Relaxation of Edges for Bellman-Ford:

- It states that for the graph having **N** vertices, all the edges should be relaxed **N-1** times to compute the single source shortest path.
- In order to detect whether a negative cycle exists or not, relax all the edge one more time and if the shortest distance for any node reduces then we can say that a negative cycle exists. In short if we relax the edges **N** times, and there is any change in the shortest distance of any node between the **N-1th** and **Nth** relaxation than a negative cycle exists, otherwise not exist.

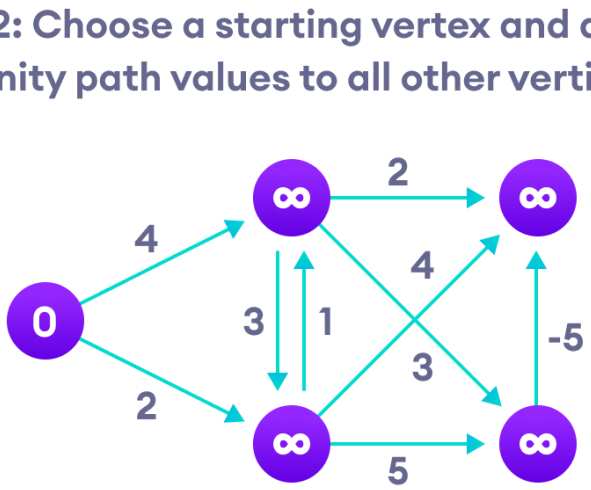
Bellman Ford's Algorithm Applications:

Network Routing: Bellman-Ford is used in computer networking to find the shortest paths in routing tables, helping data packets navigate efficiently across networks.
GPS Navigation: GPS devices use Bellman-Ford to calculate the shortest or fastest routes between locations, aiding navigation apps and devices.
Transportation and Logistics: Bellman-Ford's algorithm can be applied to determine the optimal paths for vehicles in transportation and logistics, minimizing fuel consumption and travel time.
Game Development: Bellman-Ford can be used to model movement and navigation within virtual worlds in game development, where different paths may have varying costs or obstacles.
Robotics and Autonomous Vehicles: The algorithm aids in path planning for robots or autonomous vehicles, considering obstacles, terrain, and energy consumption.

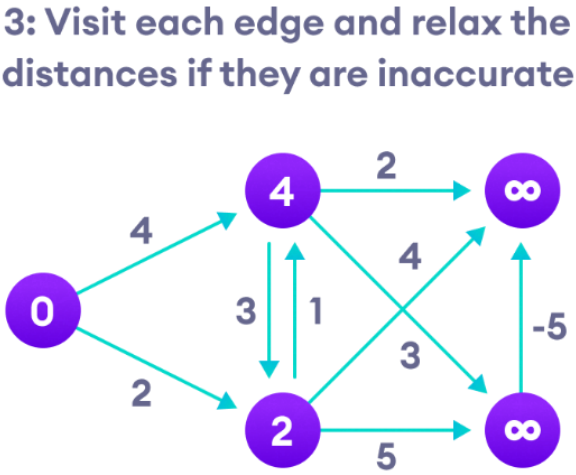
Example of Bellman Ford Algorithm :-



Step-1 for Bellman Ford's algorithm

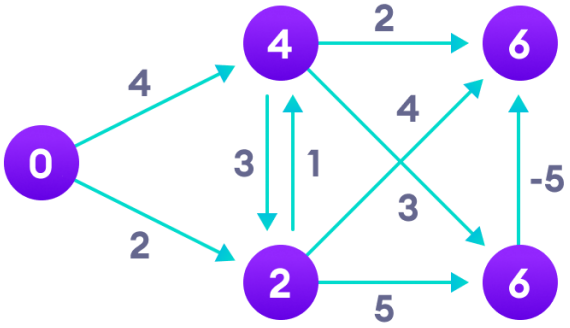


Step-2 for Bellman Ford's algorithm



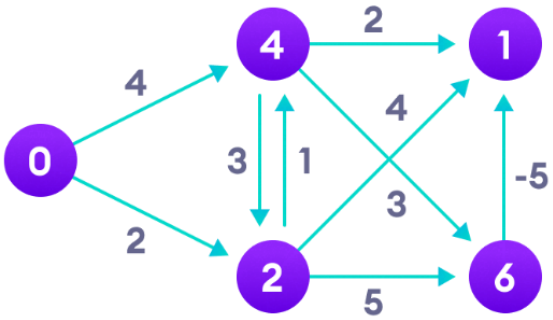
Step-3 for Bellman Ford's algorithm

Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



Step-4 for Bellman Ford's algorithm

Step 5: Notice how the vertex at the top right corner had its path length adjusted



Step-5 for Bellman Ford's algorithm

Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Step-6 for Bellman Ford's algorithm

All-Pairs Shortest Path (APSP) Algorithm:

APSP algorithms aim to find the shortest paths between every pair of vertices in a graph. One of the most well-known APSP algorithms is the Floyd-Warshall algorithm. Floyd-Warshall algorithm operates by iteratively updating a distance matrix that stores the shortest distances between all pairs of vertices. It considers all vertices as potential intermediate nodes in the paths being explored. The algorithm iterates over all pairs of vertices, and for each pair, it checks if going through an intermediate vertex results in a shorter path than the direct path between the two vertices. If a shorter path is found, the distance matrix is updated accordingly. Floyd-Warshall algorithm works well for both directed and undirected graphs with positive or negative edge weights, but it does not work with graphs containing negative cycles.

Floyd-Warshall Algorithm

1. *Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.*
2. *This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).*
3. *A weighted graph is a graph in which each edge has a numerical value associated with it.*
4. *Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.*
5. *This algorithm follows the dynamic programming approach to find the shortest paths.*

ALGORITHM

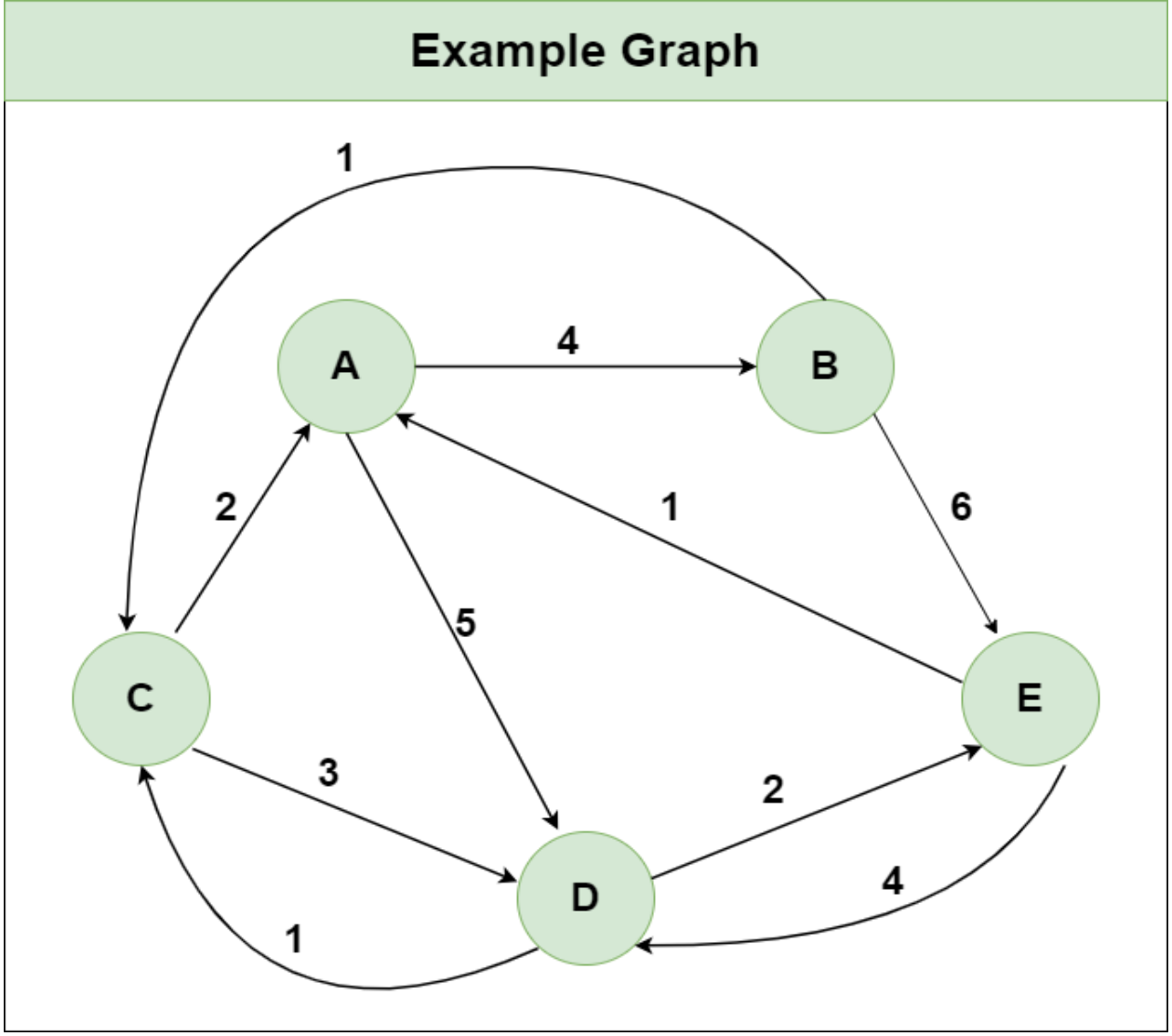
```
n = no of vertices
A = matrix of dimension n*n

for k = 1 to n

    for i = 1 to n

        for j = 1 to n

            A^k[i, j] = min (A^k-1[i, j], A^k-1[i, k] + A^k-1[k, j])
return A
```



Step 1: *Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from* ***i*** *to* ***j****, also Distance[i][j] = Infinity if there is no edge from* ***i*** *to* ***j****.*

Step1: Initializing Distance[ ][ ] using the Input Graph

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Step 2: *Treat node* ***A*** *as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to ***A****) + (Distance from* ***A*** *to j))*

*= Distance[i][j] = minimum (Distance[i][j], Distance[i][****A****] + Distance[****A****][j])*

Step 2: Using Node A as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

Step 3: *Treat node **B** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*
*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **B**) + (Distance from **B** to j))*
*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**B**] + Distance[**B**][j])*

Step 3: Using Node B as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

Step 4: *Treat node **C** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*
*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **C**) + (Distance from **C** to j))*
*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**C**] + Distance[**C**][j])*

Step 4: Using Node C as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

Step 5: *Treat node **D** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **D**) + (Distance from **D** to j))*

*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**D**] + Distance[**D**][j])*

Step 5: Using Node D as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 6: *Treat node **E** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **E**) + (Distance from **E** to j))*

*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**E**] + Distance[**E**][j])*

Step 6: Using Node E as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 7: *Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.*

Step 7: Return Distance[ ][ ] matrix as the result

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Dijkstra's Algorithms vs Bellman-Ford Algorithm

Feature	Dijkstra's	Bellman Ford
Optimization	Optimized for finding the shortest path between a single source node and all other nodes in a graph with non-negative edge weights.	Bellman-Ford algorithm is optimized for finding the shortest path between a single source node and all other nodes in a graph with negative edge weights.
Relaxation	Dijkstra's algorithm uses a greedy approach where it chooses the node with the smallest distance and updates its neighbours.	the Bellman-Ford algorithm relaxes all edges in each iteration, updating the distance of each node by considering all possible paths to that node.
Time Complexity	Dijkstra's algorithm has a time complexity of O(V^2) for a dense graph and O(E log V) for a sparse graph, where V is the number of vertices and E is the number of edges in the graph.	Bellman-Ford algorithm has a time complexity of O(VE), where V is the number of vertices and E is the number of edges in the graph.
Negative Weights	Dijkstra's algorithm does not work with graphs that have negative edge weights, as it assumes that all edge weights are non-negative.	Bellman-Ford algorithm can handle negative edge weights and can detect negative-weight cycles in the graph.

Dijkstra's Algorithm vs Floyd-Warshall Algorithm

Feature	Dijkstra's	Floyd-Warshall Algorithm
Optimization	Optimized for finding the shortest path between a single source node and all other nodes in a graph with non-negative edge weights	Floyd-Warshall algorithm is optimized for finding the shortest path between all pairs of nodes in a graph.
Technique	Dijkstra's algorithm is a single-source shortest path algorithm that uses a greedy approach and calculates the shortest path from the source node to all other nodes in the graph.	Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that uses dynamic programming to calculate the shortest path between all pairs of nodes in the graph.
Time Complexity	Dijkstra's algorithm has a time complexity of O(V^2) for a dense graph and O(E log V) for a sparse graph, where V is the number of vertices and E is the number of edges in the graph.	Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that uses dynamic programming to calculate the shortest path between all pairs of nodes in the graph.
Negative Weights	Dijkstra's algorithm does not work with graphs that have negative edge weights, as it assumes that all edge weights are non-negative.	Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that uses dynamic programming to calculate the shortest path between all pairs of nodes in the graph.