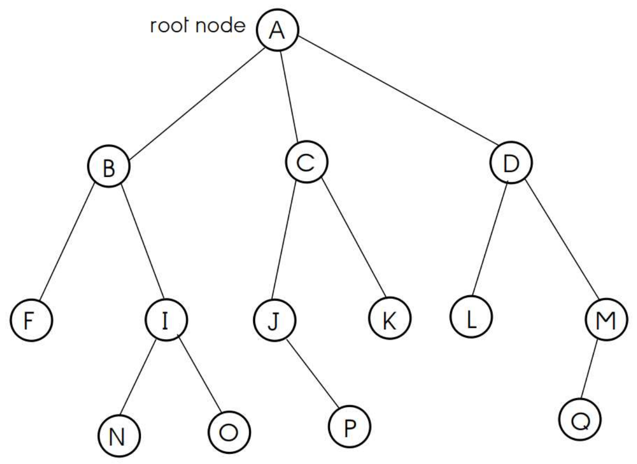
Tree and Graph

Data Structure

Trainer : Smita Kadam
Email ID : smita@sunbeaminfo.com

Data Structures: Tree

Tree: It is a **non-linear**, **advanced** data structure which is a collection of finite no. of logically related similar type of elements in which, there is a first specially designated element referred as a **root element**, and remaining all elements are connected to it in a **hierarchical manner**, follows **parent-child relationship**.



Tree: Data Structure

Data Structures: Tree

- **siblings/brothers:** child nodes of same parent are called as siblings.
- **ancestors:** all the nodes which are in the path from root node to that node.
- **decedents:** all the nodes which can be accessible from that node.
- **degree of a node** = no. of child nodes having that node
- **degree of a tree** = max degree of any node in a given tree
- **leaf node/external node/terminal node:** node which is not having any child node OR node having degree 0.
- **non-leaf node/internal node/non-terminal node:** node which is having any no. of child node/s OR node having non-zero degree.
- **level of a node** = level of its parent node + 1
- **level of a tree** = max level of any node in a given tree (by assuming level of root node is at level 0).
- **depth of a tree** = max level of any node in a given tree.
- as tree data structure can grow up to any level and any node can have any number of child nodes, operations on it becomes inefficient, so restrictions can be applied on it to achieve efficiency and hence there are diefferent types of tree.

Data Structures: Tree

- Binary tree: it is a tree in which each node can have max 2 number of child nodes, i.e. each node can have either 0 OR 1 OR 2 number of child nodes.

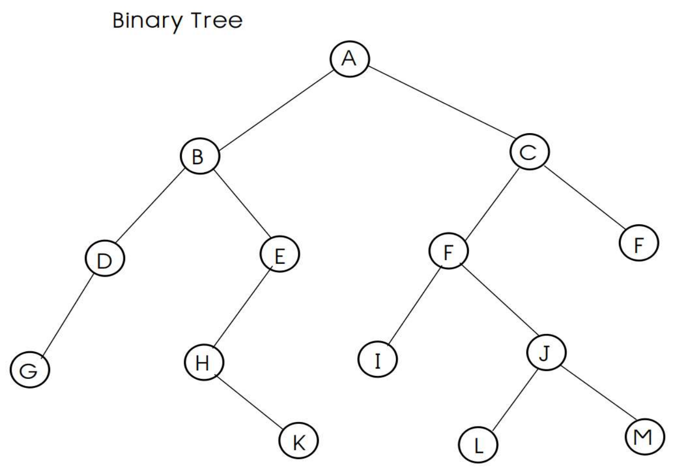
OR

Binary tree: it is a set of finite number of elements having three subsets:

1. root element

2. left subtree (may be empty)

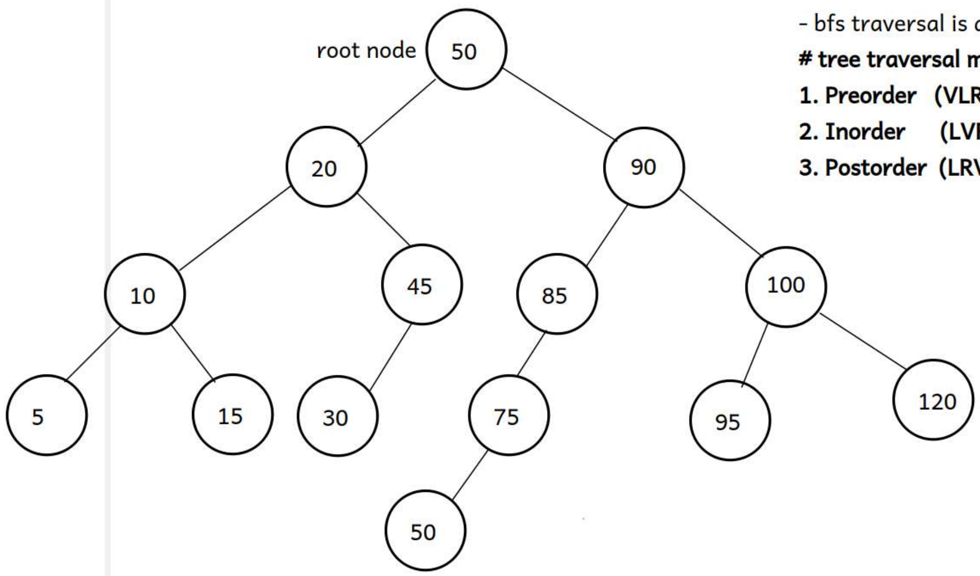
3. right subtree (may be empty)



Data Structures: Tree

- **Binary Search Tree(BST):** it is a **binary tree** in which left child is always smaller than its parent and right child is always greater than or equal to its parent.

input order of an ele's for BST: 50 20 90 85 10 45 30 100 15 75 95 120 5 50



1. dfs traversal: 50 20 10 5 15 45 30 90 85 75 50 100 95 120

2. bfs traversal: 50 20 90 10 45 85 100 5 15 30 75 95 120 50

- bfs traversal is also called as "levelwise traversal".

# tree traversal methods on BST:

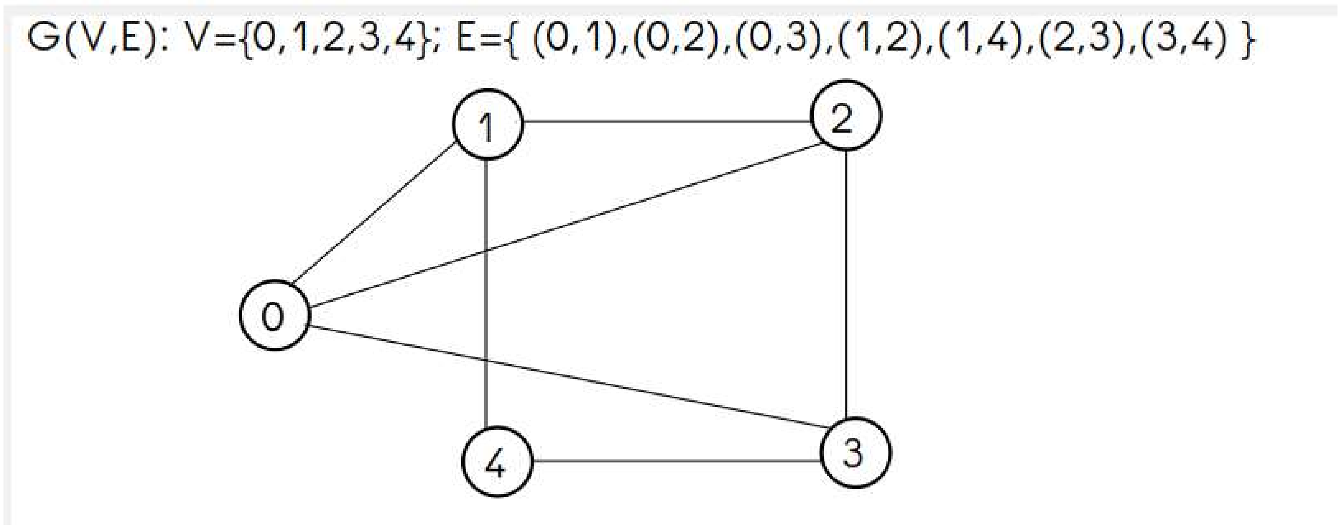
1. Preorder (VLR) : 50 20 10 5 15 45 30 90 85 75 50 100 95 120

2. Inorder (LVR): 5 10 15 20 30 45 50 50 75 85 90 95 100 120

3. Postorder (LRV): 5 15 10 30 45 20 50 75 85 95 120 100 90 **50**

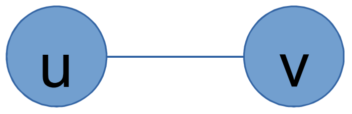
Data Structures: Graph

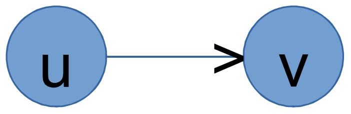
Graph: It is **non-linear/advanced** data structure, which is a collection of logically related similar and dissimilar type of elements which contains set of finite no. of elements referred as a **vertices**, also called as **nodes**, and set of finite no. of ordered/unordered pairs of vertices referred as an **edges**, also called as an **arcs**, whereas it may carries **weight/cost/value** (cost/weight/value may be -ve).


G(V,E): V={0,1,2,3,4}; E={ (0,1),(0,2),(0,3),(1,2),(1,4),(2,3),(3,4) }
0
1
2
3
4

Data Structures: Graph

- If there exists a direct edge between two vertices then those vertices are referred as an **adjacent vertices** otherwise **non-adjacent**.





(u, v) == (v, u)   (u, v) != (v, u)

- if we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**

e.g. (u,v) == (v,u) => unordered pair of vertices => undirected edge => graph which contanis undirected edges referred as undirected graph.

- if we cannot represent any edge either (u,v) OR (v,u) then it is referred as an **unordered pair of vertices** i.e. directed edge.

(u,v) != (v,u) => ordered pair of vertices => directed edge -> graph which contains set of directed edges referred as directed graph (di-graph).

Data Structures: Graph

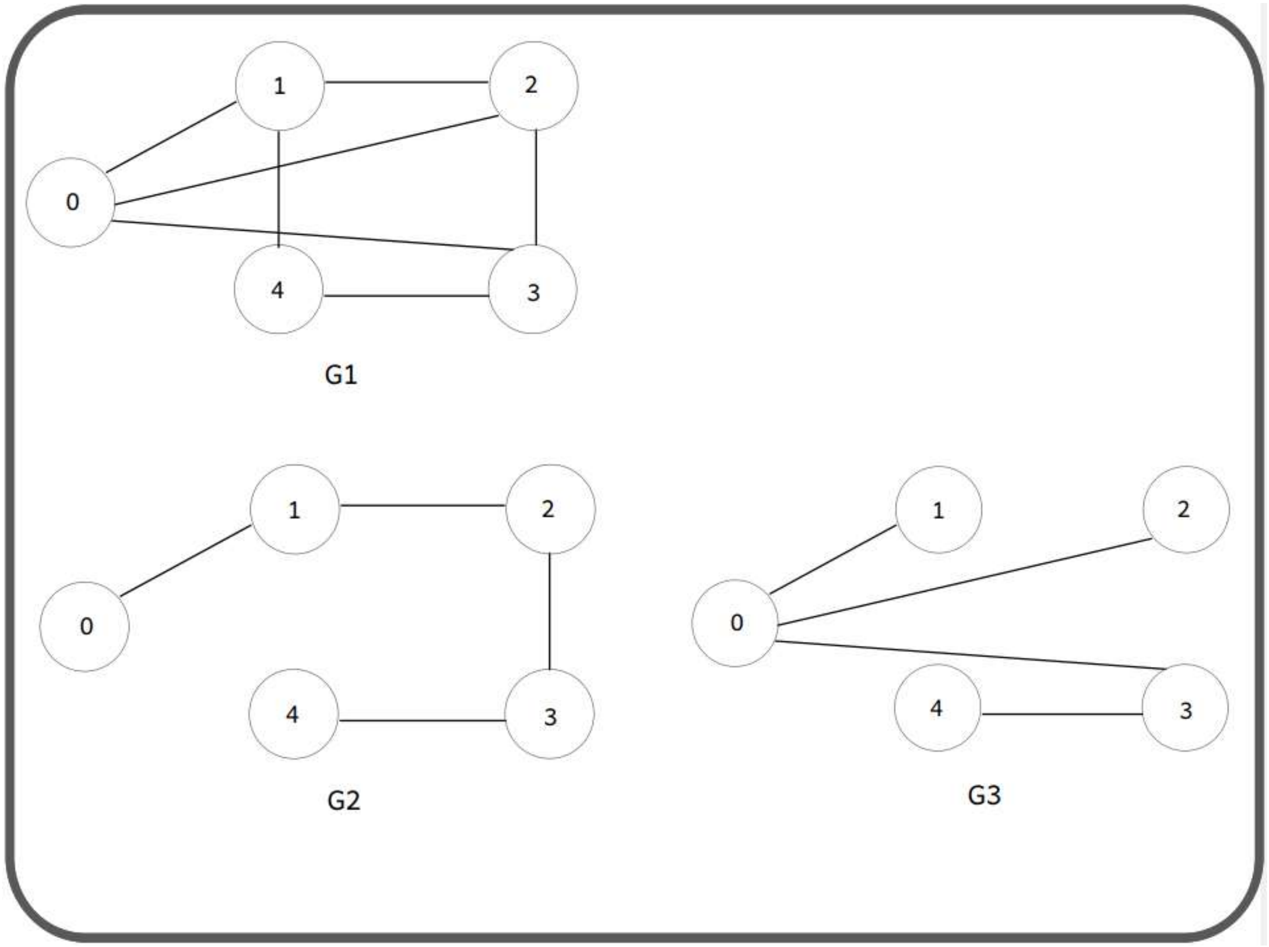
- **loop:** if there is an edge from any vertex to that vertex itself, such edge is called as a loop.

- **complete graph:** if all the vertices are adjacent to remaining all vertices in a given graph, in this graph degree of each vertex is (V-1), whereas V = no. of vertices.

- **connected vertices:** if there exists a direct/indirect path between two vertices then those two vertices are referred as a connected vertices otherwise not-connected.

- **connected graph:** if any vertex is connected to remaining all vertices in a given graph then such a graph is called as connected graph.

Data Structures: Graph



Data Structures: Graph

- path : in a given graph, if in any path starting vertex and end vertex are same, such a path is called as a cycle.

- weight of a graph = sum of weights of all its edges (applicable only in a weighted graph).

- **spanning tree :** it is a subgraph of a graph can be formed by removing one or more edges from it in such a way that it should remains connected and do not contains a cycle.

- **minimum spanning tree:** spanning tree of a given graph having min weight.

- one graph may has multiple spanning trees, **prim's** and **kruskal's** are 2 algorithms **to find minimum spanning tree of a given graph.**

- **dijsktra's** and **bellman ford** are the algorithms to find **shortest distance of all vertices from given source vertex.**

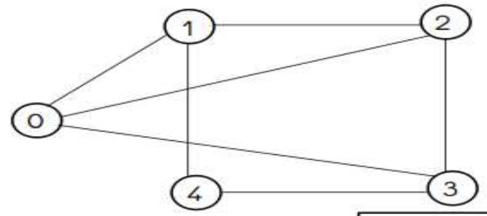
Data Structures: Graph

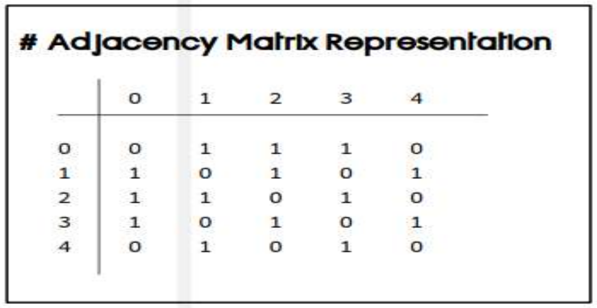
- There are two graph representation methods:

1. Adjacency Matrix Representation (2-D Array)

2. Adjacency List Representation (Array of Linked Lists)

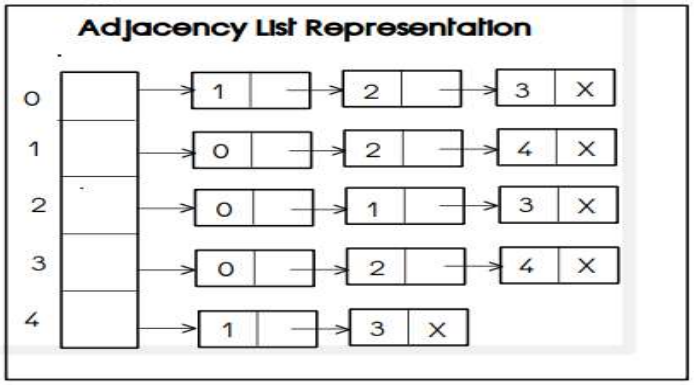
G(V,E): V={0,1,2,3,4}; E={ (0,1),(0,2),(0,3),(1,2),(1,4),(2,3),(3,4) }





Adjacency Matrix Representation

	0	1	2	3	4
0	0	1	1	1	0
1	1	0	1	0	1
2	1	1	0	1	0
3	1	0	1	0	1
4	0	1	0	1	0



Data Structures: Hash Table

Hash Table: it is a **non-linear/advanced data structure** which is a **collection of finite number of logically related similar type of data elements/records** gets stored into the memory in an **associative manner i.e. in a key-value pairs** (for faster searching).

Hashing: It is an improvement over **"Direct Access Table"** in which hash function can be used and the table is reffered as "Hash Table".

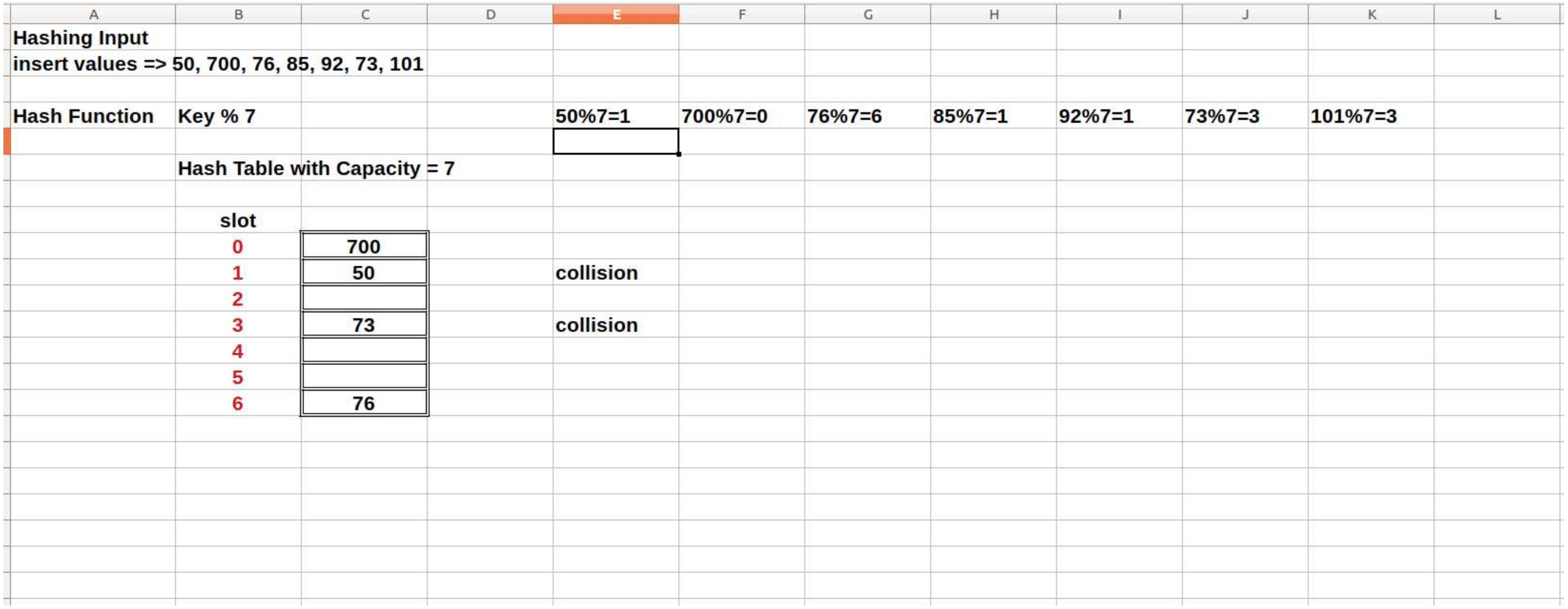
Hash Function: it is a function that **converts a given big key value/number into a small practical integer value/key** which is reffered as **hash key/hash code** which is a mapped value can be used as an index in a hash table.

Collision: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some **collision handling technique.**

- There are two **collission handling techniques:**
1. Chaining/Seperate Chaining
2. Open Addressing

Data Structures: Hash Table

Collision:

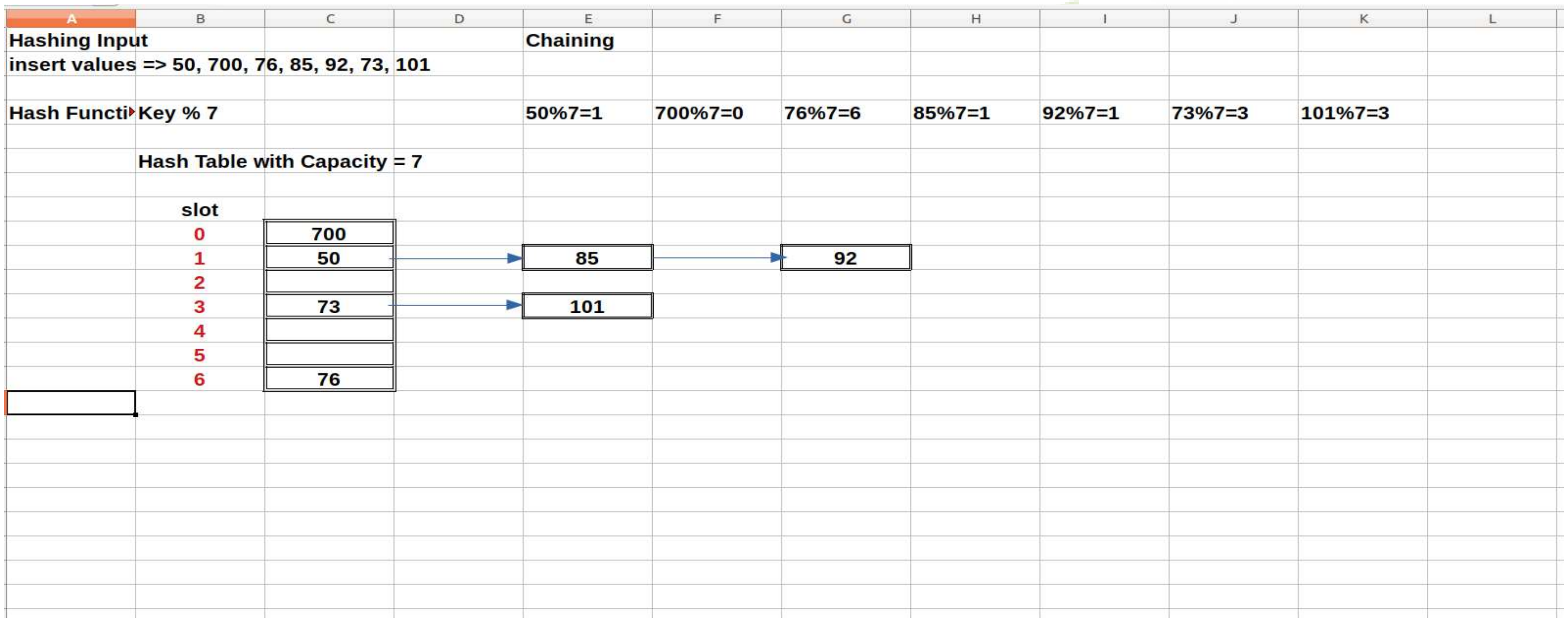


A	B	C	D	E	F	G	H	I	J	K	L
Hashing Input											
insert values => 50, 700, 76, 85, 92, 73, 101											
Hash Function	Key % 7			50%7=1	700%7=0	76%7=6	85%7=1	92%7=1	73%7=3	101%7=3	
	Hash Table with Capacity = 7										
	slot										
	0	700									
	1	50		collision							
	2										
	3	73		collision							
	4										
	5										
	6	76									

Data Structures: Hash Table

1. Chaining:

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.



Hashing Input

insert values => 50, 700, 76, 85, 92, 73, 101

Hash Functi Key % 7

Chaining

50%7=1 700%7=0 76%7=6 85%7=1 92%7=1 73%7=3 101%7=3

Hash Table with Capacity = 7

slot			
0	700		
1	50	85	92
2			
3	73	101	
4			
5			
6	76		

Data Structures: Hash Table

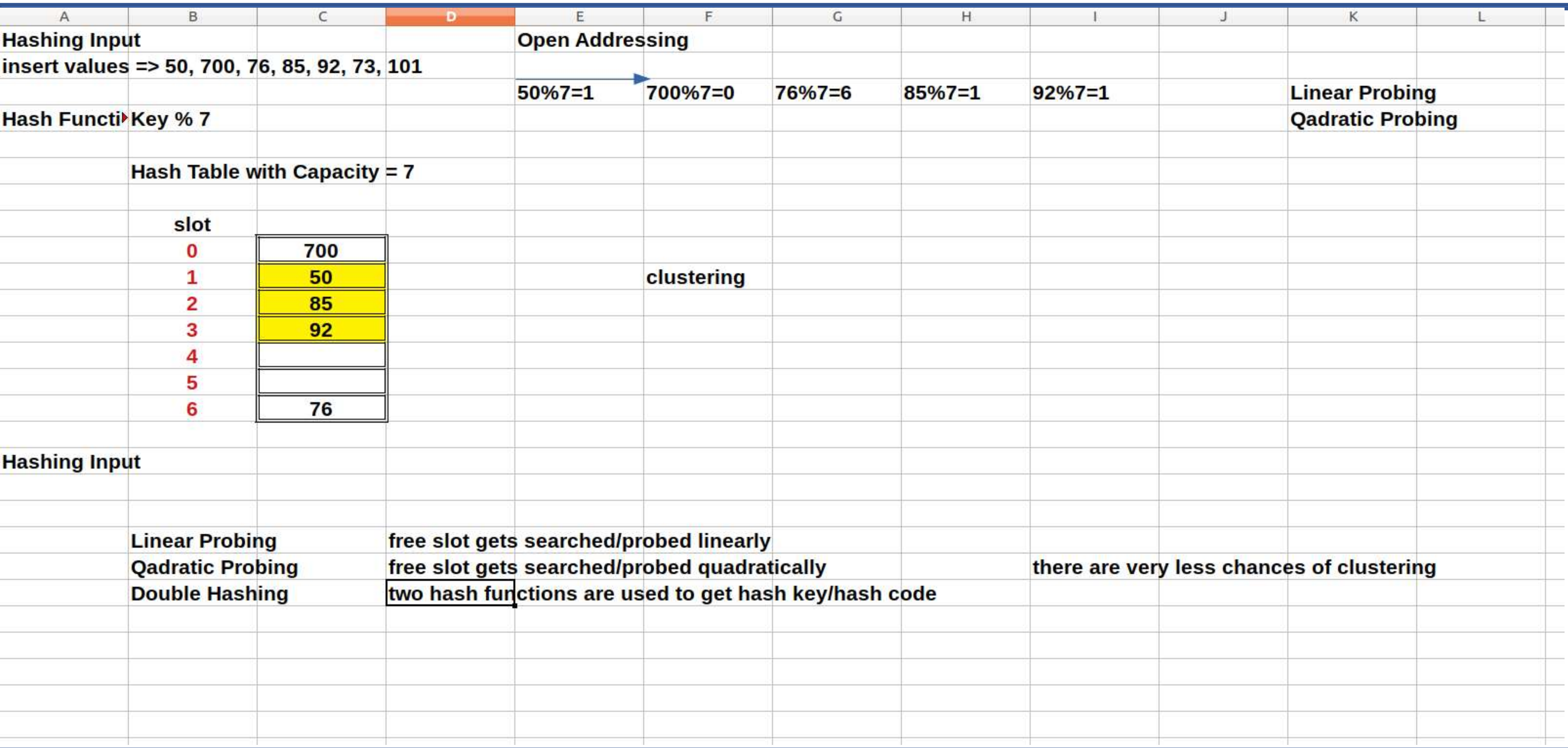
2. Open Addressing:
- In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.
- Open Addressing is done following ways:

A. Linear Probing: In linear probing, if collision occurs next free slot will be searched/probed linearly.

B. Quadratic Probing: In quadratic probing, if collision occurs next free slot will be searched/probed quadratically.

C. Double Hashing: In double hashing, if collision occurs next free slot will be searched/probed by using another hash function, so two hash functions can be use to find next/probe next free slot.

Data Structures: Hash Table



A	B	C	D	E	F	G	H	I	J	K	L
Hashing Input				Open Addressing							
insert values => 50, 700, 76, 85, 92, 73, 101											
				50%7=1	700%7=0	76%7=6	85%7=1	92%7=1		Linear Probing	
Hash Functi	Key % 7									Qadratic Probing	
	Hash Table with Capacity = 7										
	slot										
	0	700									
	1	50			clustering						
	2	85									
	3	92									
	4										
	5										
	6	76									
Hashing Input											
	Linear Probing		free slot gets searched/probed linearly								
	Qadratic Probing		free slot gets searched/probed quadratically					there are very less chances of clustering			
	Double Hashing		two hash functions are used to get hash key/hash code								

Thank You