# UNIT-5

**Trees:** Introduction to Trees, properties of Trees,Binary Search Tree – Insertion, Deletion & Traversal **Graphs:** Introduction to graphs, properties, representation,Traversal
**Hashing: Brief** introduction to hashing and hash functions, Collision resolution techniques: chaining and open addressing, Hash tables: basic implementation and operations, Applications of hashing in unique identifier generation, caching.

## INTRODUCTION TO TREES:

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root. Figure 4.1 shows a tree where node A is the root node; nodes B, C, and D are children of the root node and form sub-trees of the tree rooted at node A.

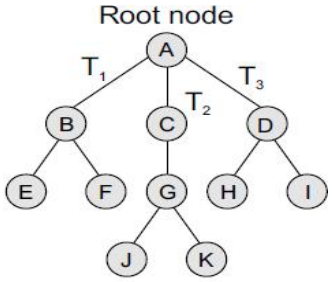



**Figure 4.1** Tree

Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively

**ROOT NODE** The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.
**SUB-TREES** If the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R.
**LEAF NODE** A node that has no children is called the leaf node or the terminal node or child node. In a tree, all the nodes except root are child nodes.
**PATH** A sequence of consecutive edges is called a path. For example, in Fig. 4.1, the path from the root node A to node I is given as: A, D, and I.
**ANCESTOR NODE** An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in Fig. 4.1, nodes A, C, and G are the ancestors of node K.

**DESCENDANT NODE** A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in Fig. 4.1, nodes C, G, J, and K are the descendants of node A.
**Level number** Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.

### Properties of Trees

- ✓ A tree is a hierarchical structure as it contains multiple levels.
- ✓ In a tree, the topmost node is known as the root node.
- ✓ A node that doesn't have a child node is known as a leaf node or terminal node.
- ✓ The highest number of nodes at every level of i is 2i.
- ✓ Height of the tree = the longest path from the root node to the leaf node.
- ✓ Depth of a node = the length of the path to its root.

## BINARY TREES

In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as a left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called Binary Tree. Figure 4.2 shows a binary tree. In the figure, R is the root node and the two trees T1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R.

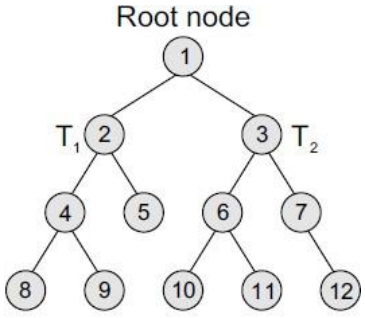
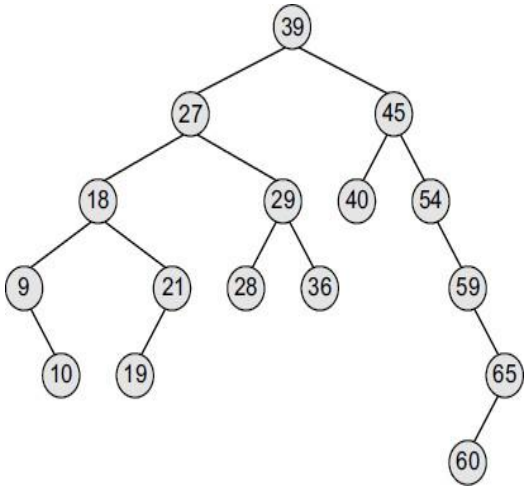



**Figure 4.2** Binary tree

## BINARY SEARCH TREES

A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree. (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)

Binary search trees also speed up the insertion and deletion operations. The tree has a speed advantage when the data in the structure changes rapidly.




The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36.

All these nodes have smaller values than the root node. The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65.
Recursively, each of the sub-trees also obeys the binary search tree constraint.

For example, in the left sub-tree of the root node, 27 is the root and all elements in its left sub-tree (9, 10, 18, 19, 21) are smaller than 27, while all nodes in its right sub-tree (28, 29, and 36) are greater than the root node's value.

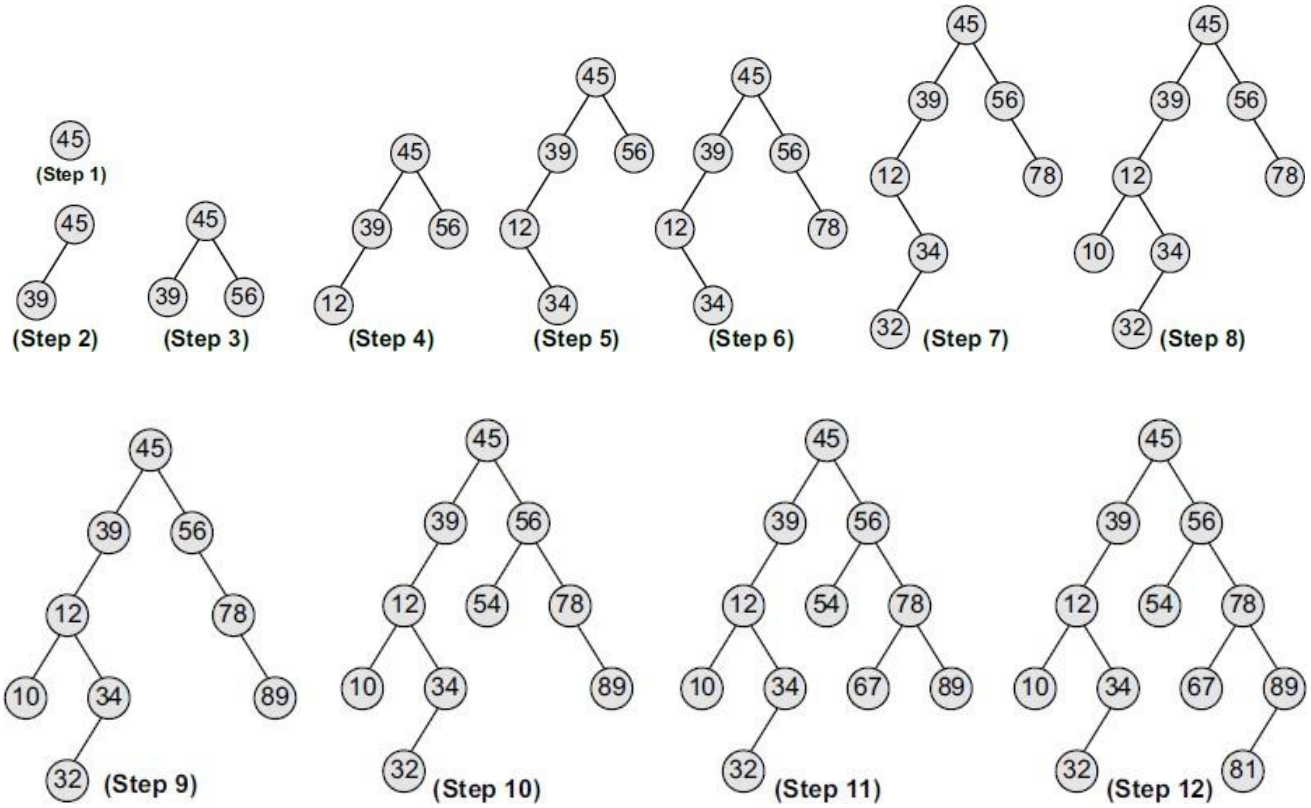
Binary search trees are considered to be efficient data structures especially when compared with sorted linear arrays and linked lists. In a sorted array, searching can be done in $O(\log_2 n)$ time, but insertions and deletions are quite expensive. In contrast, inserting and deleting elements in a linked list is easier, but searching for an element is done in O(n) time.
However, in the worst case, a binary search tree will take O(n) time to search for an element.

To summarize, a binary search tree is a binary tree with the following properties:

- ✓ The left sub-tree of a node N contains values that are less than N's value.
- ✓ The right sub-tree of a node N contains values that are greater than N's value.
- ✓ Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

**Example:**

Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, and 81.




## OPERATIONS ON BINARY SEARCH TREES

### Inserting a New Node in a Binary Search Tree

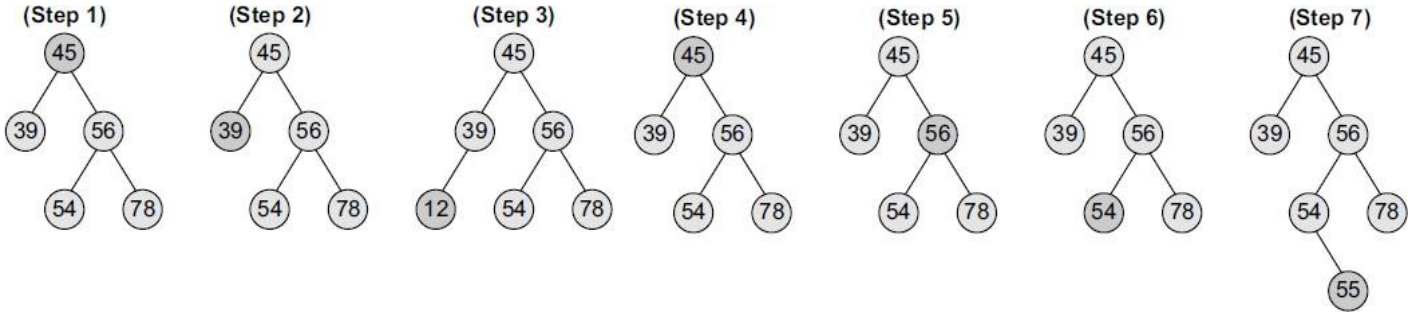
```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

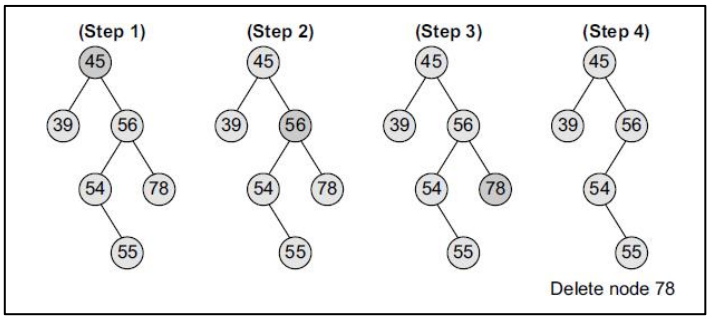
The insert function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and O(n) time in the worst case.

**Example:** Inserting nodes with values 12 and 55 in the given binary search tree



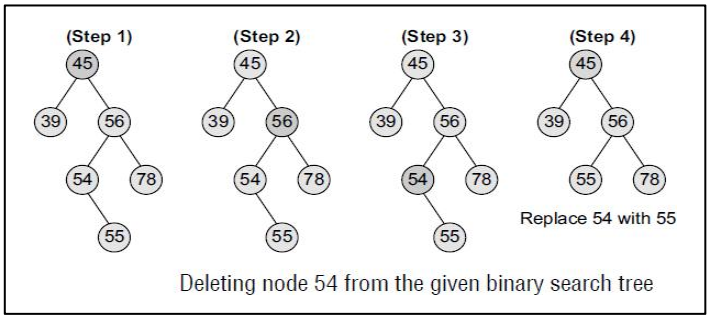

## Deleting a Node from a Binary Search Tree

The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process.
Case 1: Deleting a Node that has No Children
Case 2: Deleting a Node with One Child
Case 3: Deleting a Node with Two Children




Case1: Deleting a Node that has No Children

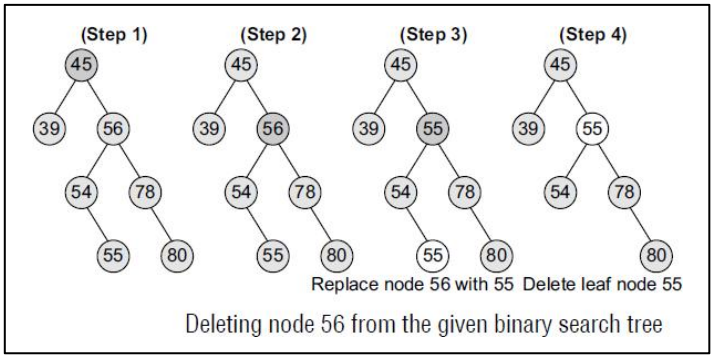
If we have to delete node 78, we can simply remove this node without any issue.
This is the simplest case of deletion.




Case 2: Deleting a Node with One Child

To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child. Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.

Look at the binary search tree shown in figure and see how deletion of node 54 is handled.




To handle this case, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree).
The in-order predecessor or the successor can then be deleted using any of the above cases.

Look at the binary search tree given in figure and see how deletion of node with value 56 is handled.

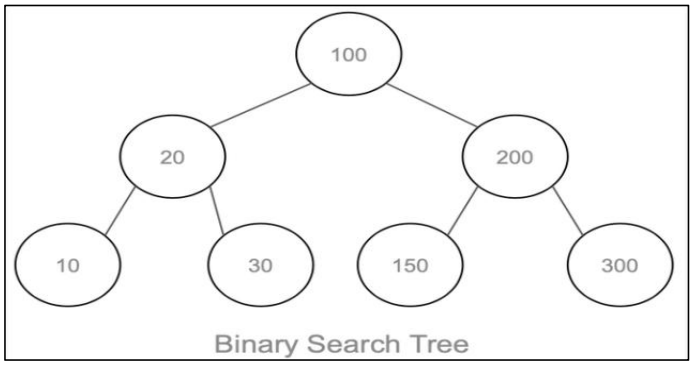
**BINARY TREE TRAVERSALS**

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

In - Order Traversal
Pre - Order Traversal
Post - Order Traversal

Consider the following binary tree...




Inorder Traversal:
- ✓ Traverse left subtree
- ✓ Visit the root and print the data.
- ✓ Traverse the right subtree

**Inorder Traversal:** 10 20 30 100 150 200 300

**Preorder Traversal:**
- ✓ Visit the root and print the data.
- ✓ Traverse left subtree
- ✓ Traverse the right subtree

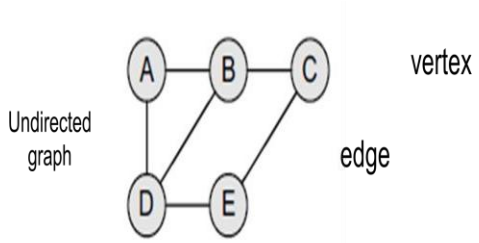
**Preorder Traversal:** 100 20 10 30 200 150 300

**Postorder Traversal:**
- ✓ Traverse left subtree
- ✓ Traverse the right subtree
- ✓ Visit the root and print the data.

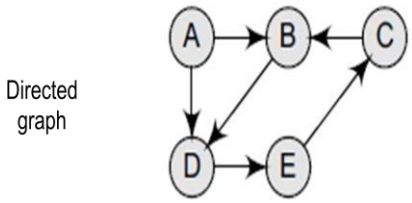
**Postorder Traversal:** 10 30 20 150 300 200 100

# GRAPH

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges. Here edges are used to connect the vertices.
Graph is a collection of nodes and edges in which nodes are connected with edges




A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them.
That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.
A graph G with V(G) = {A, B, C, D and E} and
E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}.




In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A.
The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).
A graph G with V(G) = {A, B, C, D and E} and
E(G) = {(A, B), (C, B), (A, D), (B, D), (D, E), (E, C)}.

## The basic properties of a graph:

**Adjacent nodes** or **neighbours** For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end-points and are said to be the adjacent nodes or neighbours.

**Degree of a node** Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

**Regular graph** It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a k–regular graph or a regular graph of degree k.

**Path** A path P written as P = {v0, v1, v2, ..., vn), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v0, v = vn and vi–1 is adjacent to vi for i = 1, 2, 3,..., n.

**Connected graph** A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph.

**Complete graph** A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.

## REPRESENTATION OF GRAPHS

There are three common ways of storing graphs in the computer's memory. They are:

- Sequential representation by using an adjacency matrix.
- Linked representation by using an adjacency list that stores the neighbours of a node using a linked list.
- Adjacency multi-list which is an extension of linked representation.

### Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.

In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n × n.

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry $a_{ij}$ in the adjacency matrix will contain 1(one), if vertices $v_i$ and $v_j$ are adjacent to each other. However, if the nodes are not adjacent, $a_{ij}$ will be set to 0(zero). It is summarized in Figure.

Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

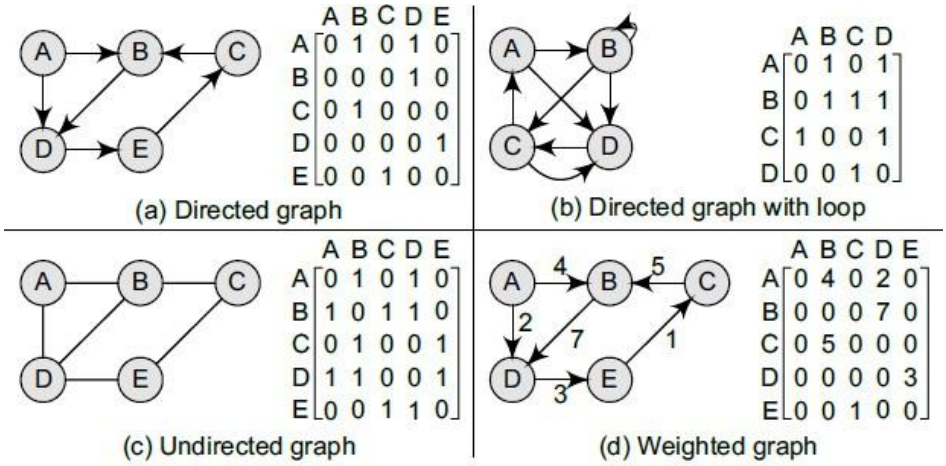



**Figure** Graphs and their corresponding adjacency matrices

### Adjacency List Representation

An adjacency list is another way in which graphs can be represented in the computer's memory.
This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

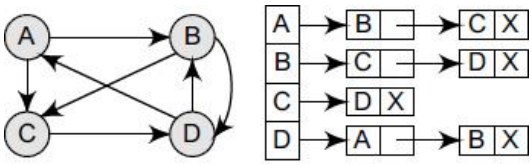



**Figure** Graph G and its adjacency list

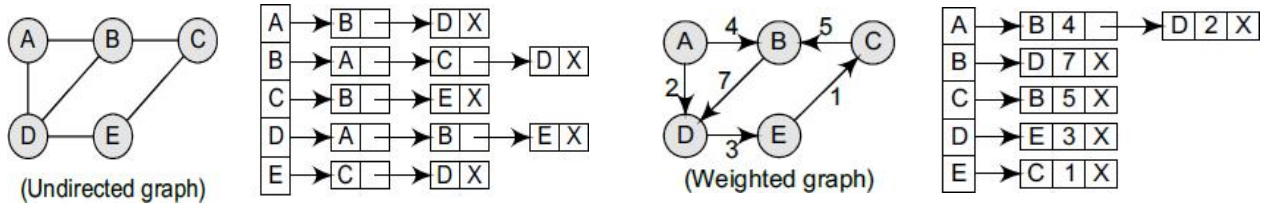



**Figure** Adjacency list for an undirected graph and a weighted graph

## GRAPH TRAVERSAL

In this section, we will discuss how to traverse graphs. By traversing a graph, we mean the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will discuss in this section.These two methods are:

1. Breadth-first search
2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack. But both these algorithms make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1 or 2, depending on its current state.

| Status | State of the node | Description |
|---|---|---|
| 1 | Ready | The initial state of the node N |
| 2 | Waiting | Node N is placed on the queue or stack and waiting to be processed |
| 3 | Processed | Node N has been completely processed |

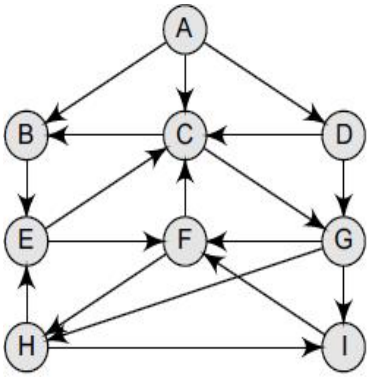
### Breadth-first search (BFS):

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

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

Algorithm for breadth-first search




**Adjacency lists**
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

**Figure** Graph G and its adjacency list

- The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered.
- During the execution of the algorithm, we use two arrays: QUEUE and ORIG.
- While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1.
- Initially, FRONT = REAR = –1.

The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

| | |
|---|---|
| FRONT = 0 | QUEUE = A |
| REAR = 0 | ORIG = \0 |

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

| | | | | |
|---|---|---|---|---|
| FRONT = 1 | QUEUE = A | B | C | D |
| REAR = 3 | ORIG = \0 | A | A | A |

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

| | | | | | |
|---|---|---|---|---|---|
| FRONT = 2 | QUEUE = A | B | C | D | E |
| REAR = 4 | ORIG = \0 | A | A | A | B |

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

| | | | | | | |
|---|---|---|---|---|---|---|
| FRONT = 3 | QUEUE = A | B | C | D | E | G |
| REAR = 5 | ORIG = \0 | A | A | A | B | C |

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

| FRONT = 4 | QUEUE = A | B | C | D | E | G |
|---|---|---|---|---|---|---|
| REAR = 5 | ORIG = \O | A | A | A | B | C |

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

| FRONT = 5 | QUEUE = A | B | C | D | E | G | F |
|---|---|---|---|---|---|---|---|
| REAR = 6 | ORIG = \O | A | A | A | B | C | E |

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

| FRONT = 6 | QUEUE = A | B | C | D | E | G | F | H | I |
|---|---|---|---|---|---|---|---|---|---|
| REAR = 9 | ORIG = \O | A | A | A | B | C | E | G | G |

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have Path P as A -> C -> G -> I.

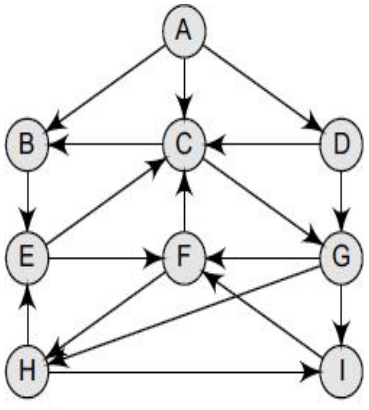
The time complexity can also be expressed as O( | E | + | V | )

**Depth-first search (DFS)**

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
In other words, depth-first search begins at a starting node A which becomes the current node.

Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

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




**Adjacency lists**
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

**Figure** Graph G and its adjacency list

Consider the graph G given in Figure. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are: H, I, F, C, G, B, E.

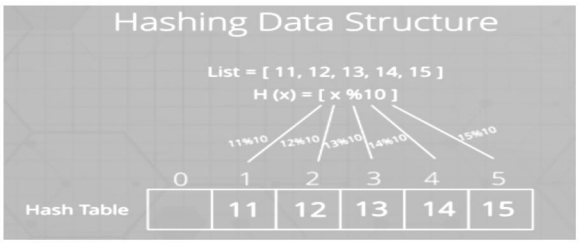
These are the nodes which are reachable from the node H.

The time complexity can be given as (O(|V| + |E|)).

**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** that enables fast retrieval of information based on its key.
- ✓ This method is commonly used in databases, **c**aching systems, and various programming applications to optimize search and retrieval operations.




Works:

- ✓ **Hash Function:** You provide your data items into the hash function.
- ✓ **Hash Code:** The hash function crunches the data and give a unique hash code.
- ✓ **Hash Table:** The hash code then points you to a specific location within the hash table.

**Hash Table in Data Structure:**

A **hash table** is also known as a hash map. It is a data structure that stores **key-value pairs**. It uses a **hash function** to map **keys** to a fixed-size array, called a **hash table**.

This allows in faster **search**, **insertion**, and **deletion** operations.

**Hash Function:**

The **hash function** is a function that takes a **key** and returns an **index** into the **hash table**. The goal of a hash function is to distribute keys evenly across the hash table, minimizing collisions (when two keys map to the same index).

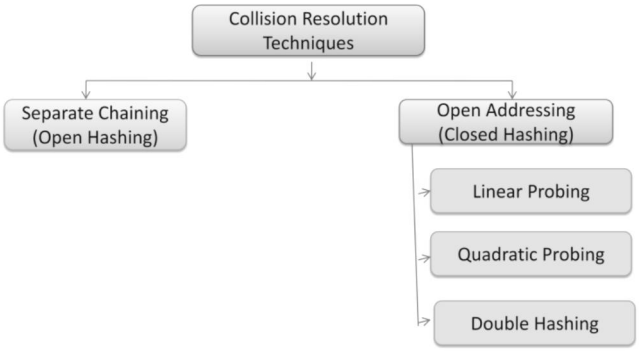
- ✓ **Division Method:** Key % Hash Table Size
- ✓ **Multiplication Method:** (Key * Constant) % Hash Table Size
- ✓ **Universal Hashing:** A family of hash functions designed to minimize collisions

**What is a Hash Collision?**

A hash collision occurs when two different keys map to the same index in a hash table. This can happen even with a good hash function, especially if the hash table is full or the keys are similar.

**Collision Resolution Techniques**

There are two types of collision resolution techniques:




1. **Open Addressing:**
   - **Linear Probing:** Search for an empty slot sequentially
   - **Quadratic Probing:** Search for an empty slot using a quadratic function
2. **Closed Addressing:**
   - **Chaining:** Store colliding keys in a linked list or binary search tree at each index

**Open Addressing:**

In open addressing,

✓ Unlike separate chaining, all the keys are stored inside the hash table.

✓ No key is stored outside the hash table.

**1. <u>Linear Probing:</u>**

**In linear probing,**

✓ When collision occurs, we linearly probe for the next bucket.

✓ We keep probing until an empty bucket is found.

**• Advantage-**

– It is easy to compute.

**• Disadvantage-**

– The main problem with linear probing is clustering.

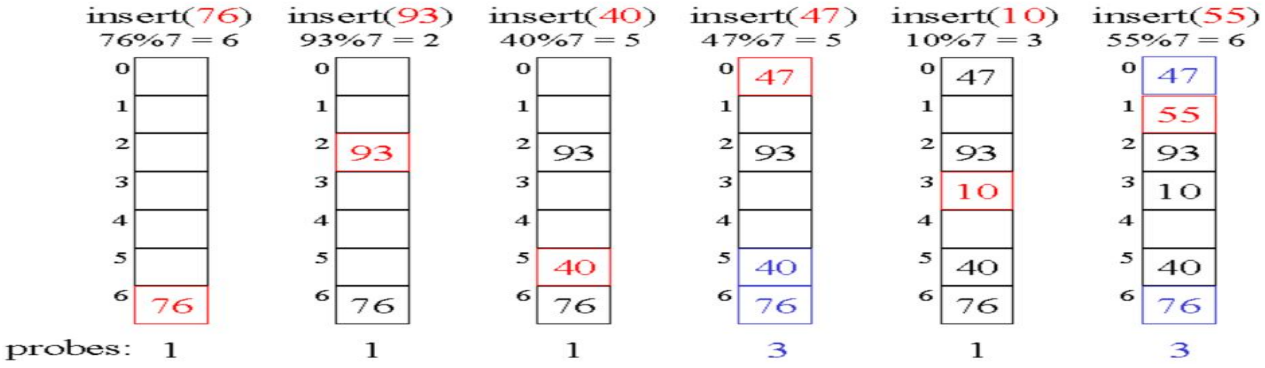
– Many consecutive elements form groups.

– Then, it takes time to search an element or to find an empty bucket.

$f(i) = i$

- Probe sequence is
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size
  - …

## Example- Linear Probing



## 2. Quadratic Probing :

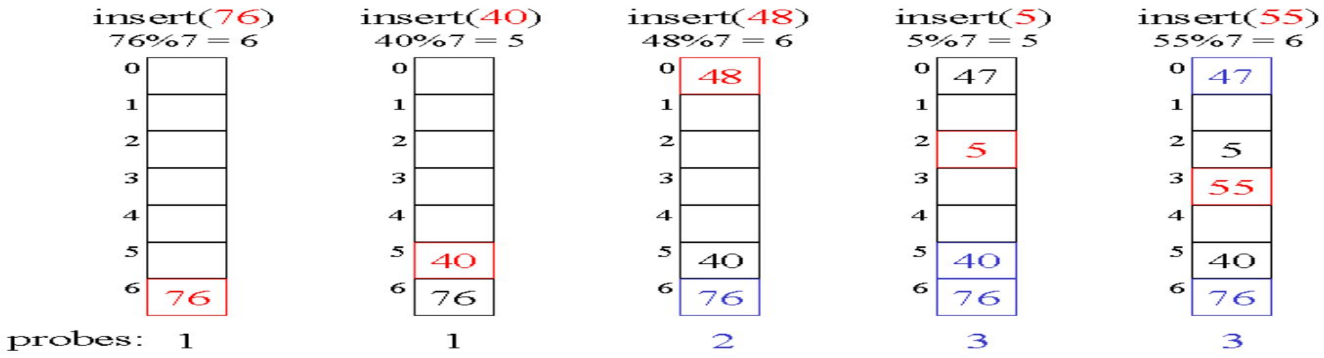
In quadratic probing,

- ✓ When collision occurs, we probe for i2 'th bucket in i th iteration.
- ✓ We keep probing until an empty bucket is found.

$f(i) = i^2$

- Probe sequence is
  - h(k) mod size
  - (h(k) + 1) mod size
  - (h(k) + 4) mod size
  - (h(k) + 9) mod size
  - …

## Example — Quadratic Probing




## 3. Double Hashing

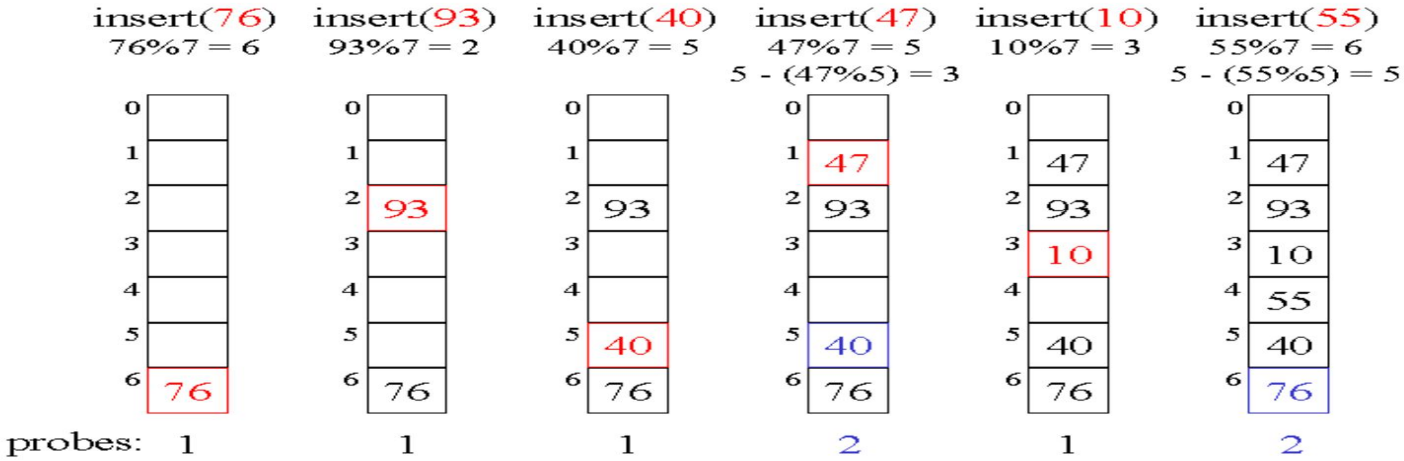
In double hashing,

- ✓ We use another hash function hash2(x) and look for i * hash2(x) bucket in i th iteration.
- ✓ It requires more computation time as two hash functions need to be computed.

$f(i) = i \cdot hash_2(x)$

- Probe sequence is
  - $h_1(k)$ mod size
  - $(h_1(k) + 1 \cdot h_2(x))$ mod size
  - $(h_1(k) + 2 \cdot h_2(x))$ mod size
  - …

## Example —Double Hashing



**Separate Chaining**

• To handle the collision,

– This technique creates a linked list to the slot for which collision occurs.

– The new key is then inserted in the linked list.

– These linked lists to the slots appear like chains.

– That is why, this technique is called as separate chaining.

Example

-Separate Chaining

• Using the hash function 'key mod 7',

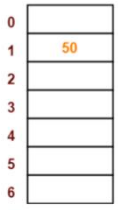
insert the following sequence of keys in the hash table- • 50, 700, 76, 85, 92, 73 and 101

## Step-1

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-

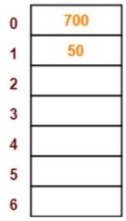

## Step-2

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



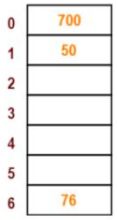

## Step-3

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



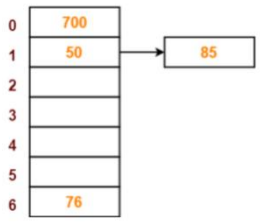

## Step-4

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



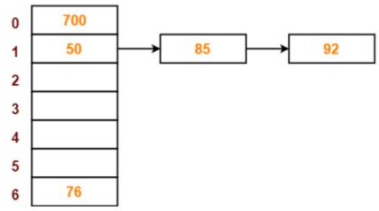
## Step-5

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



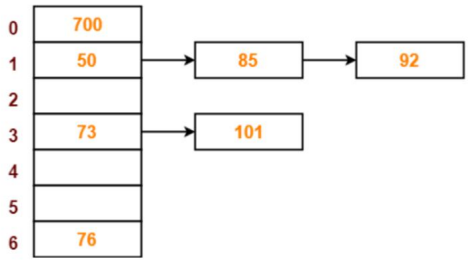

## Step-6

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-




## Step-7

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as-




**Operations on Hash Table :**

DataItem :Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
   int data;
   int key;
};
```

Hash Method ::Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

**Search Operation:**

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

```
struct DataItem *search(int key) {
  int hashIndex = hashCode(key);
  while(hashArray[hashIndex] != NULL) {
    if(hashArray[hashIndex]->key == key)
      return hashArray[hashIndex];
     ++hashIndex;
     hashIndex %= SIZE;
  }
  return NULL;
}
```

**Insert Operation:**

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

```
void insert(int key,int data) {
  struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
  item->data = data;
  item->key = key;
  int hashIndex = hashCode(key);
  while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
     ++hashIndex;
     hashIndex %= SIZE;
  }
  hashArray[hashIndex] = item;
}
```

**Delete Operation:**

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

```
struct DataItem* delete(struct DataItem* item) {
  int key = item->key;
```

```
  //get the hash
  int hashIndex = hashCode(key);
    while(hashArray[hashIndex] !=NULL) {
     if(hashArray[hashIndex]->key == key) {
        struct DataItem* temp = hashArray[hashIndex];
            hashArray[hashIndex] = dummyItem;
        return temp;
     }
         ++hashIndex;
         hashIndex %= SIZE;
  }
  return NULL;
}
```

**Applications of hashing :**

- **unique identifier generation:**
  - ✓ Sha1 or SHA is a hashing algorithm used to store information in an encrypted form, which is usually used for storing passwords in a database or testing the integrity of a file.
  - ✓ Usually a document is **hashed** and the **document ID** (**unique ID**) is used later to prove the existence of the document, or to retrieve the document from a storage system. Example of hash-based unique IDs are the commit hashes in **Git** and **GitHub**, based on the content of the commit (e.g. 3c3be25bc1757ca99aba55d4157596a8ea217698) and the **Bitcoin** addresses (e.g. 1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2).
- **caching :** Storing frequently accessed data for faster retrieval.
  - ✓ Hash tables find extensive use in caching mechanisms. By storing the hash of frequently accessed data, systems can rapidly retrieve this data, significantly reducing access times and improving performance.