Data Structures - Unit-5

Graphs-Definitions, Terminology, Applications and more definitions, Properties, Graph ADT, Graph Representations- Adjacency matrix, Adjacency lists, Graph Search methods - DFS and BFS. Priority Queues -Definition and applications-max heap and min heap. Hashing-Definition, hash tables, hash functions. Collision resolution techniques - linear probing, chaining

Graphs -definition :

In data structures, graphs are a collection of nodes or vertices connected by edges. They are used to represent relationships and connections between different elements, allowing for efficient modelling and analysis of complex systems. Graphs provide a powerful framework for solving problems in various domains, such as network analysis, social media analysis, and route planning. Let's understand more!

Overview of Graphs in Data Structure:

Let us understand what is a graph in the data structure. Graphs are non-linear data structures comprising a set of nodes (or vertices), connected by edges (or arcs). Nodes are entities where the data is stored, and their relationships are expressed using edges. Edges may be directed or undirected. Graphs easily demonstrate complicated relationships and are used to solve many real-life problems.

Components of Graph Data Structure:

Vertices:

Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labelled or unlabelled.

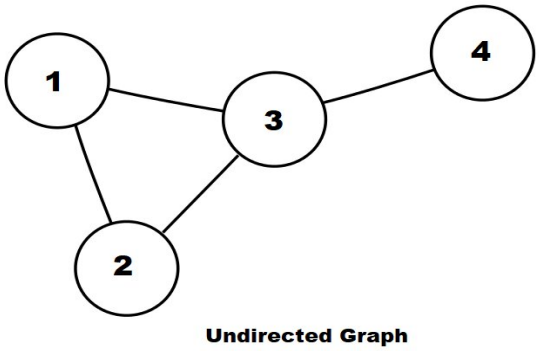
Edges:

Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

Types of Graphs in Data Structure

The most common types of graphs in the data structure are below:

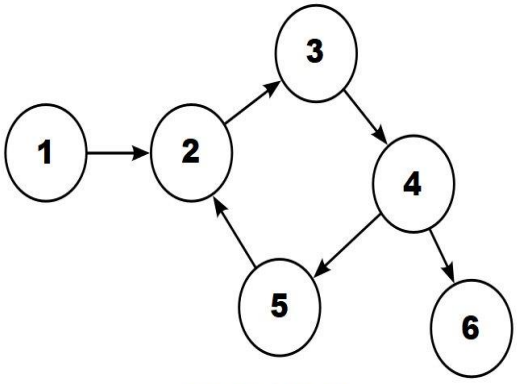
1. **Undirected:** A graph in which all the edges are bi-directional. The edges do not point in a specific direction.



Undirected Graph

.

2. Directed: A graph in which all the edges are uni-directional. The edges point in a single direction.

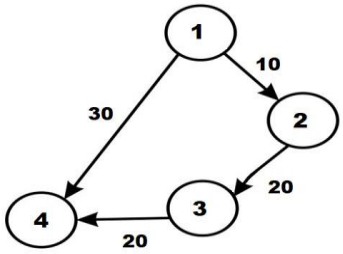


Directed Graph

3. Weighted Graph: A graph with a value associated with every edge. The values corresponding to the edges are called weights. A value in a weighted graph can represent quantities such as cost, distance, and time, depending on the graph. We typically use weighted graphs in modelling computer networks.

An edge in a weighted graph is represented as (u, v, w), where:

- u is the source vertex
- v is the destination vertex
- w represents the weight associated with going from u to v



Weighted Graph

4. Unweighted Graph: A graph with no value or weight associated with the edge. All the graphs are unweighted by default unless there is a value associated.

An edge of an unweighted graph is represented as (u, v), where:

- u represents the source vertex
- v is the destination vertex

Applications:

Used heavily in social networks. Everyone on the network is a vertex (or node) of the graph and if connected, then there is an edge. Now imagine all the features that you see, mutual friends, people that follow you, etc can seen as graph problems.

Used to represent the topology of computer networks, such as the connections between routers and switches. Used to represent the connections between different places in a transportation network, such as roads and airports.

Neural Networks: Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.

Compilers: Graph Data Structure is used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation, and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

Robot planning: Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.

Dependencies in a software project (or any other type of project) can be seen as graph and generating a sequence to solve all tasks before dependents is a standard graph topological sorting algorithm.

For optimizing the cost of connecting all locations of a network. For example, minimizing wire length in a wired network to make sure all devices are connected is a standard Graph problem called Minimum Spanning Tree.

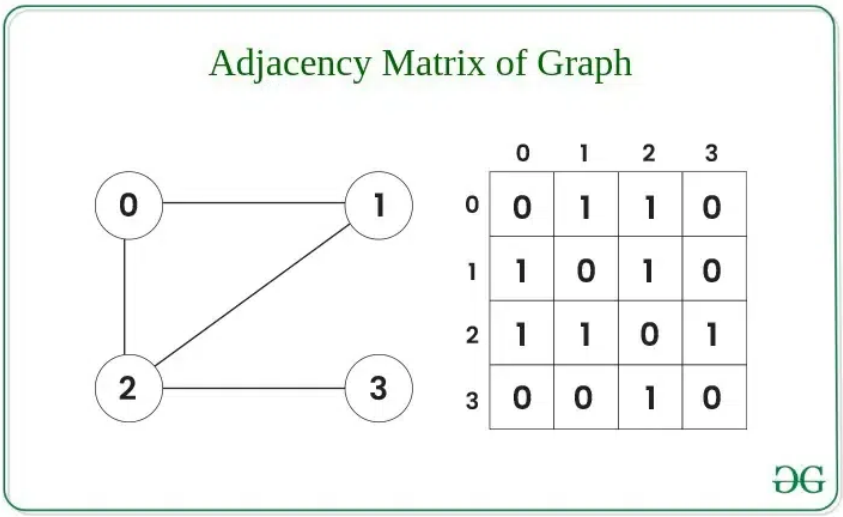
Representation of Graph Data Structure:

There are multiple ways to store a graph: The following are the most common representations.

- Adjacency Matrix
- Adjacency List

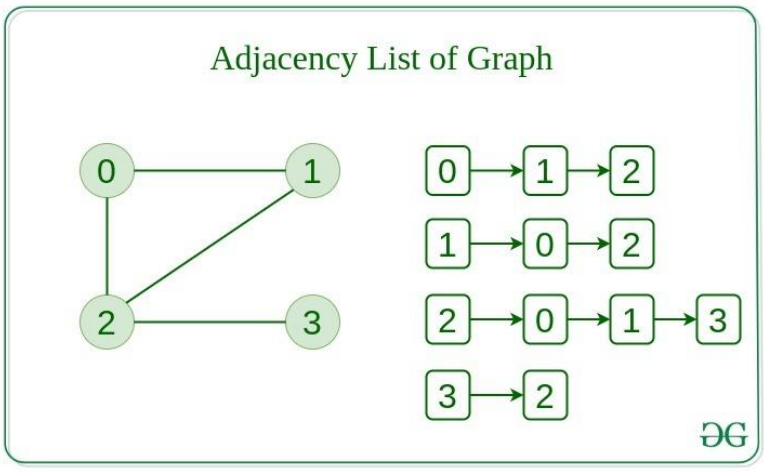
Adjacency Matrix Representation of Graph Data Structure:

In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.



Adjacency List Representation of Graph Data Structure:

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Terminology :

The following are some of the commonly used terms in graph data structure:

Term	Description
Vertex	Every individual data element is called a vertex or a node. In the above image, the vertices are A, B, C, D & E.
Edge (Arc)	It is a connecting link between two nodes or vertices. Each edge has two ends and is represented as (startingVertex, and endingVertex).
Undirected Edge	It is a bidirectional edge.
Directed Edge	It is a unidirectional edge.
Weighted Edge	An edge with value (cost) on it.
Degree	The total number of edges connected to a vertex in a graph.
Indegree	The total number of incoming edges connected to a vertex.
Outdegree	The total number of outgoing edges connected to a vertex.
Self-loop	An edge is called a self-loop if its two endpoints coincide.
Adjacency	Vertices are said to be adjacent if an edge is connected.

Depth-First Search Algorithm :

The outcome of a DFS traversal of a graph is a spanning tree. A spanning tree is a graph that is devoid of loops. To implement DFS traversal, you need to utilize a stack data structure with a maximum size equal to the total number of vertices in the graph.

To implement DFS traversal, you need to take the following stages.

Step 1: Create a stack with the total number of vertices in the graph as the size.

Step 2: Choose any vertex as the traversal's beginning point. Push a visit to that vertex and add it to the stack.

Step 3 - Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.

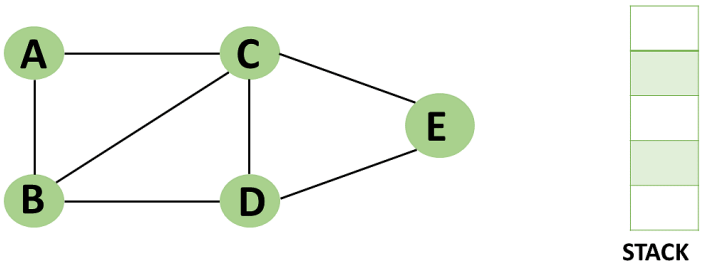
Step 4 - Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.

Step 5 - If there are no new vertices to visit, go back and pop one from the stack using backtracking.

Step 6 - Continue using steps 3, 4, and 5 until the stack is empty.

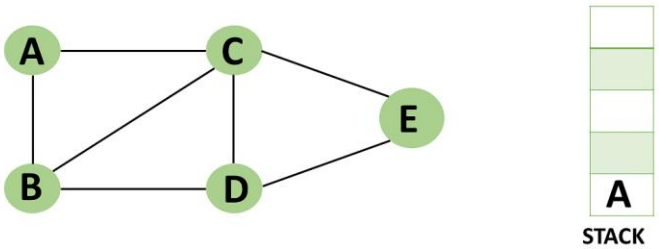
Step 7 - When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

Consider the following graph as an example of how to use the DFS algorithm:



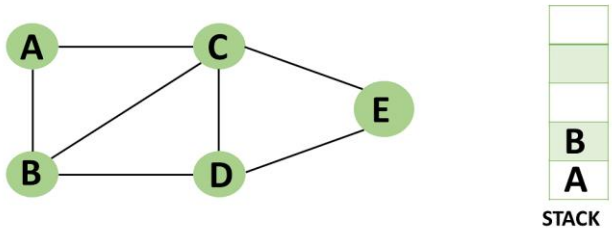
Step 1: Mark vertex A as a visited source node by selecting it as a source node.

- You should push vertex A to the top of the stack.



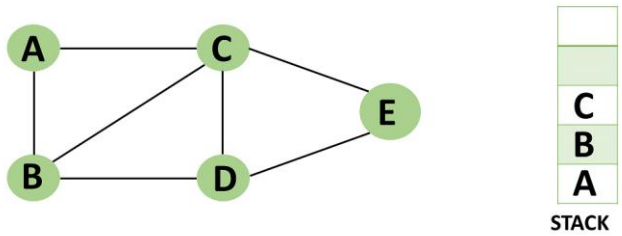
Step 2: Any nearby unvisited vertex of vertex A, say B, should be visited.

- You should push vertex B to the top of the stack.



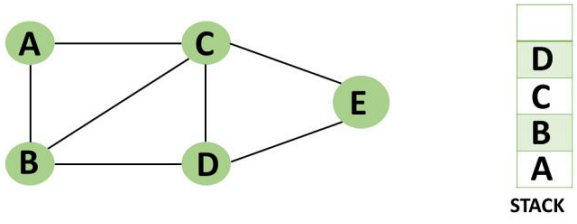
Step 3: From vertex C and D, visit any adjacent unvisited vertices of vertex B. Imagine you have chosen vertex C, and you want to make C a visited vertex.

- Vertex C is pushed to the top of the stack.



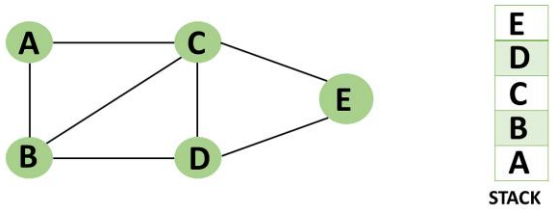
Step 4: You can visit any nearby unvisited vertices of vertex C, you need to select vertex D and designate it as a visited vertex.

- Vertex D is pushed to the top of the stack.

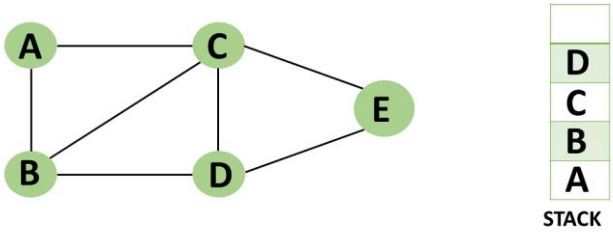


Step 5: Vertex E is the alone unvisited adjacent vertex of vertex D, thus marking it as visited.

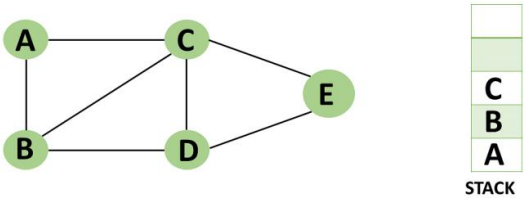
- Vertex E should be pushed to the top of the stack.



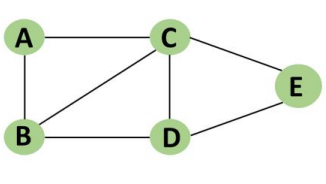
Step 6: Vertex E's nearby vertices, namely vertex C and D have been visited, pop vertex E from the stack.



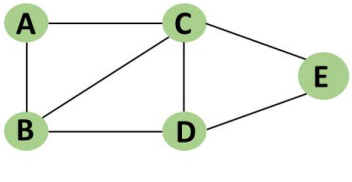
Step 7: Now that all of vertex D's nearby vertices, namely vertex B and C, have been visited, pop vertex D from the stack.



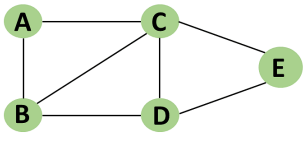
Step 8: Similarly, vertex C's adjacent vertices have already been visited; therefore, pop it from the stack.



Step 9: There is no more unvisited adjacent vertex of b, thus pop it from the stack.



step 10: All of the nearby vertices of Vertex A, B, and C, have already been visited, so pop vertex A from the stack as well.



Breadth-First Search Algorithm:

BFS is a graph traversal approach in which you start at a source node and layer by layer through the graph, analysing the nodes directly related to the source node. Then, in BFS traversal, you must move on to the next-level neighbour nodes.

Breadth-First Search uses a queue data structure to store the node and mark it as "visited" until it marks all the neighbouring vertices directly related to it. The queue operates on the First In First Out (FIFO) principle, so the node's neighbours will be viewed in the order in which it inserts them in the node, starting with the node that was inserted first.

Breadth-First Search uses a queue data structure technique to store the vertices. And the queue follows the First In First Out (FIFO) principle, which means that the neighbors of the node will be displayed, beginning with the node that was put first.

The transverse of the BFS algorithm is approaching the nodes in two ways.

1. Visited node

2. Not visited node

How Does the Algorithm Operate?

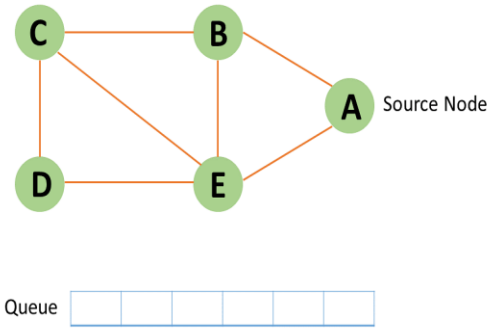
- Start with the source node.
- Add that node at the front of the queue to the visited list.
- Make a list of the nodes as visited that are close to that vertex.
- And dequeue the nodes once they are visited.
- Repeat the actions until the queue is empty.

Example of Breadth-First Search Algorithm

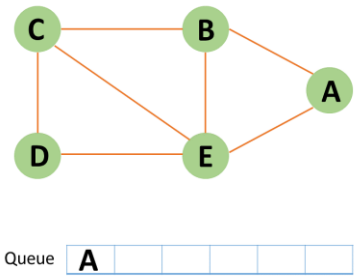
In a tree-like structure, graph traversal requires the algorithm to visit, check, and update every single un-visited node. The sequence in which graph traversals visit the nodes on the graph categorizes them.

The BFS algorithm starts at the first starting node in a graph and travels it entirely. After traversing the first node successfully, it visits and marks the next non-traversed vertex in the graph.

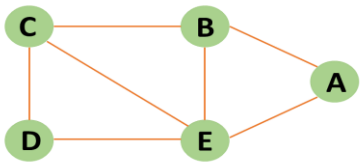
step 1: In the graph, every vertex or node is known. First, initialize a queue.



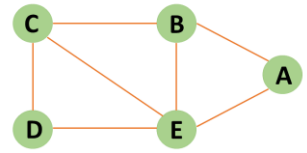
Step 2: In the graph, start from source node A and mark it as visited.

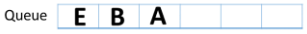


Step 3: Then you can observe B and E, which are unvisited nearby nodes from A. You have two nodes in this example, but here choose B, mark it as visited, and enqueue it alphabetically.

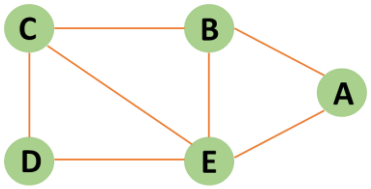


Step 4: Node E is the next unvisited neighbouring node from A. You enqueue it after marking it as visited.

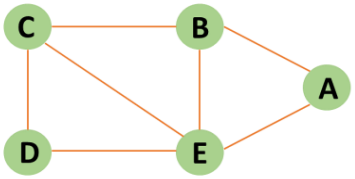




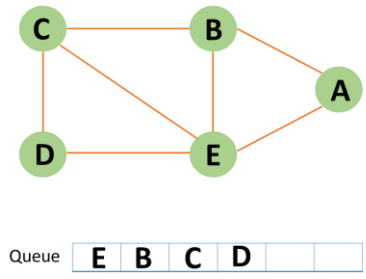
Step 5: A now has no unvisited nodes in its immediate vicinity. As a result, you dequeue and locate A.



Step 6: Node C is an unvisited neighboring node from B. You enqueue it after marking it as visited.

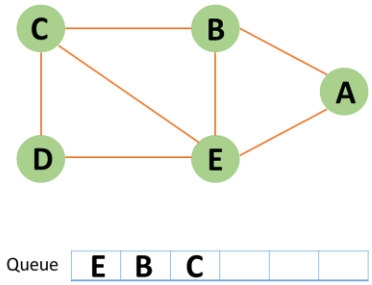


Step 7: Node D is an unvisited neighbouring node from C. You enqueue it after marking it as

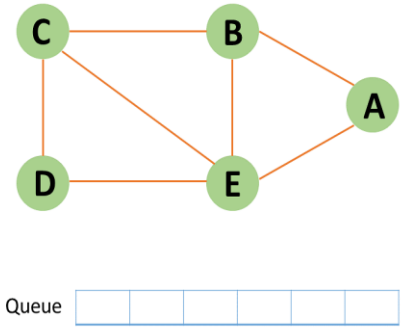


visited.

Step 8: If all of D's adjacent nodes have already been visited, remove D from the queue.



Step 9: Similarly, all nodes near E, B, and C nodes have already been visited; therefore, you must remove them from the queue.



Step 10: Because the queue is now empty, the bfs traversal has ended.

priority queue:

A priority queue is a data structure that stores elements with associated priorities. In a priority queue, elements are dequeued in order of their priority, with the highest priority elements being removed first. It is commonly used in algorithms like Dijkstra's for shortest path and in real-time scheduling tasks. Priority queues can be implemented using arrays, heaps, or linked lists.

Examples:

Input: Values = 4 (priority = 1), 5 (priority = 2), 6 (priority = 3), 7 (priority = 0)

Output: 7 4 5 6 (In descending order of priority, starting from the highest priority).

Explanation: The elements are stored in an array sorted by priority.

When you perform a dequeue(), the highest priority element (7 with priority 0) is removed first.

The next highest priority element (4 with priority 1) is then at the front of the array, followed by the others.

Input: Values = 10 (priority = 3), 3 (priority = 1), 8 (priority = 2), 1 (priority = 0)

Output: 1 3 8 10 (In descending order of priority)

Explanation: The array will be sorted by priority: 1 (highest priority) comes first, followed by 3, 8, and 10. After calling dequeue(), the element with the highest priority, 1, is removed first, then 3, followed by 8, and finally 10.

Priority Queue using Array

A priority queue stores elements where each element has a priority associated with it. In an array-based priority queue, elements are ordered so that the highest priority element is always at the front of the array. The array is sorted according to the priority values, with the element with the lowest priority value (highest priority) placed at the front.

Heap Data Structure:

A heap is a tree where each parent node is greater than or equal to its child nodes in a max-heap or less than or equal to its child nodes in a min-heap. This means the largest or smallest value is always at the top (root) of the tree.

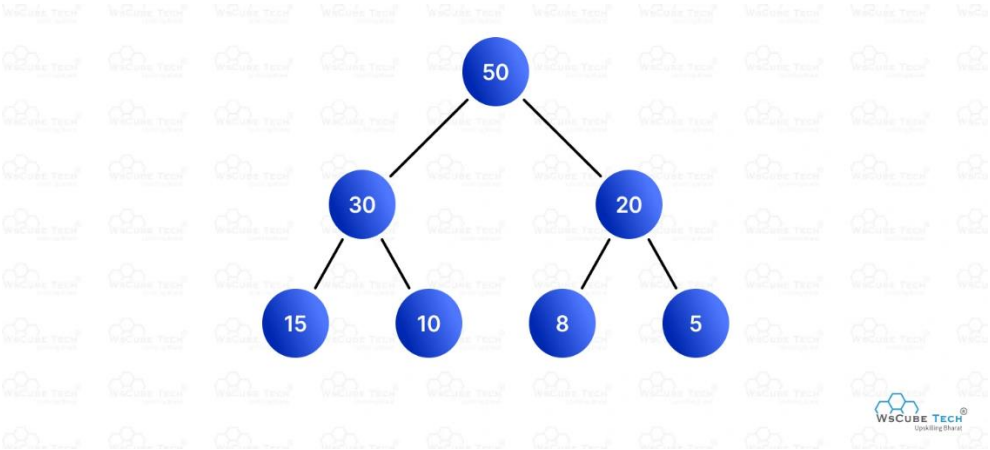
Types of Heap Data Structure:

There are two main types of heaps: Max-Heap and Min-Heap in data structure. Let's look at each type with examples.

1. Max-Heap

In a max-heap, each parent node is greater than or equal to its child nodes. This ensures that the largest value is always at the root of the tree.

Example of a Max-Heap:

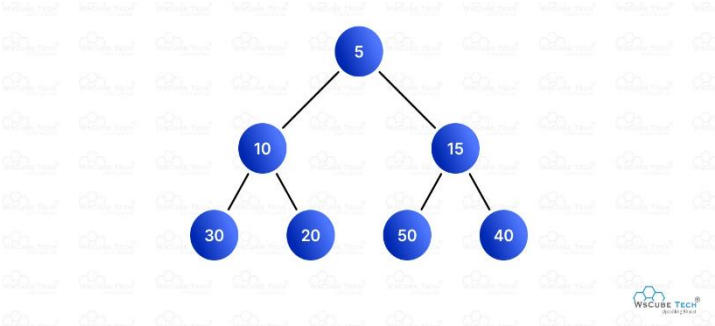


- The root node (50) is greater than its children (30 and 20).
- Each parent node (30, 20) is greater than its children (15, 10 for 30 and 8, 5 for 20).

2. Min-Heap

In a min-heap, each parent node is less than or equal to its child nodes. This ensures that the smallest value is always at the root of the tree.

Example of a Min-Heap:



- The root node (5) is smaller than its children (10 and 15).

- Each parent node (10, 15) is smaller than its children (30, 20 for 10 and 50, 40 for 15).

Properties of Heap Data Structure

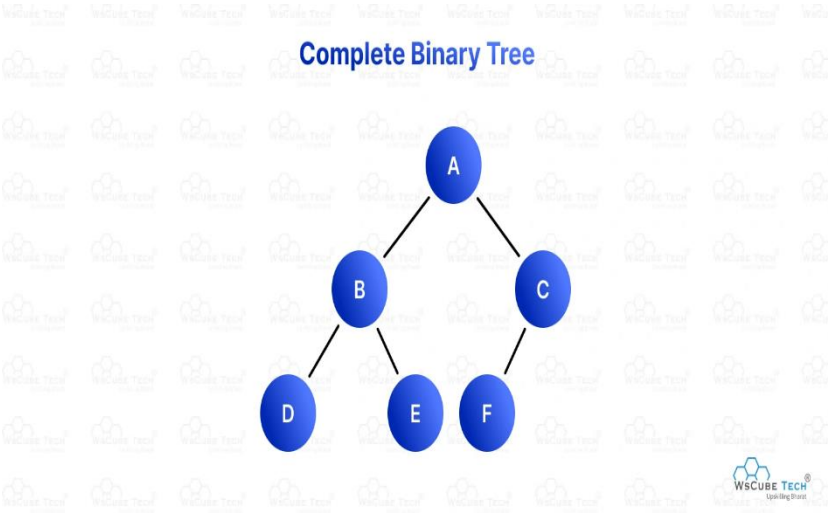
1. Complete Binary Tree

A heap is always a complete binary tree, meaning:

- All levels are fully filled except possibly the last level.
- The last level is filled from left to right.

This property ensures that the tree remains balanced, which is crucial for maintaining the efficiency of heap operations.

Example:



In this example, all levels are fully filled except the last level, which is filled from left to right.

2. Heap Order Property

The heap must satisfy the heap order property, which differs for max-heaps and min-heaps:

- Max-Heap Order Property: In a max-heap, each parent node is greater than or equal to its child nodes. This ensures that the largest element is always at the root.
- Min-Heap Order Property: In a min-heap, each parent node is less than or equal to its child nodes. This ensures that the smallest element is always at the root.

Operations on Binary Heap:

There are mainly three operations that can be performed on a binary heap:

Insert: Inserting a new element in the heap**.**

Delete: Deleting the root element of the heap.

Peek: Getting the root element of the heap.

Heapify Operation

Heapify operations are used for maintaining the heap property of the binary heap. There are two types of heapify operations:

- **Min-Heapify:** It is also known as **Bubble Down**. It is used to maintain the Min-Heap property of the binary heap.
- **Max-Heapify:** It is also known as **Bubble Up**. It is used to maintain the Max-Heap property of the binary heap.

Insert Operation on Binary Heap

- Inserting a new element in the heap is done by inserting the element at the end of the heap and then heapifying the heap.
-
- The heapifying process is done by comparing the new element with its parent and swapping the elements if the new element is smaller than the parent. The process is repeated until the new element is greater than its parent.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Max Heap construction code:

```
#include <iostream>
using namespace std;
#define MAX 1000 // Max size of Heap
// Function to heapify ith node in a Heap
// of size n following a Bottom-up approach
void heapify(int arr[], int n, int i)
{
    // Find parent
    int parent = (i - 1) / 2;
    if (parent >= 0) {
        // For Max-Heap
        // If current node is greater than its parent
        // Swap both of them and call heapify again
        // for the parent
        if (arr[i] > arr[parent]) {
            swap(arr[i], arr[parent]);

            // Recursively heapify the parent node
            heapify(arr, n, parent);
        }
    }
}

// Function to insert a new node to the Heap
void insertNode(int arr[], int& n, int Key)
{
    // Increase the size of Heap by 1
    n = n + 1;

    // Insert the element at end of Heap
    arr[n - 1] = Key;

    // Heapify the new node following a
    // Bottom-up approach
    heapify(arr, n, n - 1);
}

// A utility function to print array of size n
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    cout << "\n";
}

// Driver Code
int main()
```

```
{
    int arr[MAX] = { 10, 5, 3, 2, 4 };
    int n = 5;
    int key = 15;
    insertNode(arr, n, key);
    printArray(arr, n);
    return 0;
}
```

Delete Operation on Binary Heap

Deleting the root element of the heap is done by replacing the root element with the last element of the heap and then **heapifying** the heap.

The **heapifying** process is done by comparing the root element with its children and swapping the elements if the root element is greater than its children. The process is repeated until the root element is less than its children.

Deletion in Max-Heap Data Structure

Deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted with the last element and delete the last element of the Heap.

- Replace the root or element to be deleted with the last element.
- Delete the last element from the Heap.
- Since the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, heapify the last node placed at the position of the root.

C++ program for implement deletion in Heaps

```
#include <iostream>
using namespace std;
// To heapify a subtree rooted with node i which is
// an index of arr[] and n is the size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;
    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;
```

```
    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
// Function to delete the root from Heap
void deleteRoot(int arr[], int& n)
{
    // Get the last element
    int lastElement = arr[n - 1];

    // Replace root with last element
    arr[0] = lastElement;

    // Decrease size of heap by 1
    n = n - 1;

    // heapify the root node
    heapify(arr, n, 0);
}
/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}
int main()
{
    int arr[] = { 10, 5, 3, 2, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);
    deleteRoot(arr, n);
    printArray(arr, n);
    return 0;
}
```

Hashing -Definition, hash tables, hash functions

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.There are many hash functions that use numeric numeric or alphanumeric keys. Different hash functions are given below:

Hash Functions:

The following are some of the Hash Functions :

Division Method

This is the easiest method to create a hash function. The hash function can be described as −

h(k) = k mod n

Here, h(k) is the hash value obtained by dividing the key value k by size of hash table n using the remainder. It is best that n is a prime number as that makes sure the keys are distributed with more uniformity.

Example of the Division Method :

k=1276

n=10

h(1276) = 1276 mod 10

= 6

The hash value obtained is 6

A disadvantage of the division method id that consecutive keys map to consecutive hash values in the hash table. This leads to a poor performance.

Multiplication Method

The hash function used for the multiplication method is −

h(k) = floor(n(kA mod 1))

Here, k is the key and A can be any constant value between 0 and 1. Both k and A are multiplied and their fractional part is separated. This is then multiplied with n to get the hash value.

Example of the Multiplication Method :

k=123

n=100

A=0.618033

h(123) = 100 (123 * 0.618033 mod 1)

= 100 (76.018059 mod 1)

= 100 (0.018059)

= 1

The hash value obtained is 1

An advantage of the multiplication method is that it can work with any value of A, although some values are believed to be better than others.

Mid Square Method

The mid square method is a very good hash function. It involves squaring the value of the key and then extracting the middle r digits as the hash value. The value of r can be decided according to the size of the hash table.

An example of the Mid Square Method is as follows −

Suppose the hash table has 100 memory locations. So r=2 because two digits are required to map the key to memory location.

k = 50

k*k = 2500

h(50) = 50

The hash value obtained is 50

Hash Tables

A hash table is a data structure that maps keys to values. It uses a hash function to calculate the index for the data key and the key is stored in the index.

Example of a hash table is :

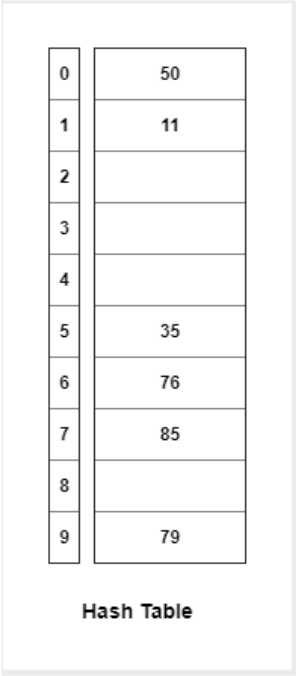
The key sequence that needs to be stored in the hash table is −

35 50 11 79 76 85

The hash function h(k) used is:

h(k) = k mod 10

Using linear probing, the values are stored in the hash table as −



Hash Table

Collision Resolution Techniques

In Hashing, hash functions were used to generate hash values. The hash value is used to create an index for the keys in the hash table. The hash function may return the same hash value for two or more keys. When two or more keys have the same hash value, a collision happens. To handle this collision, we use Collision Resolution Techniques.

There are mainly two methods to handle collision:

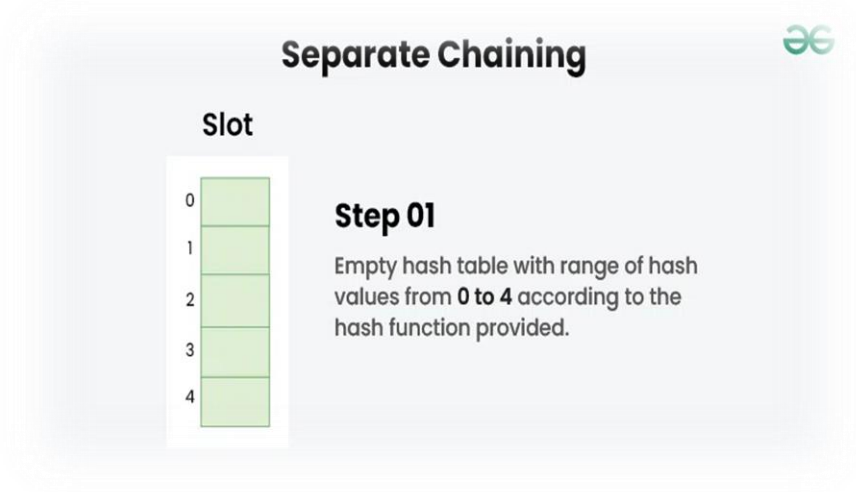
1. Chaining
2. Open Addressing(

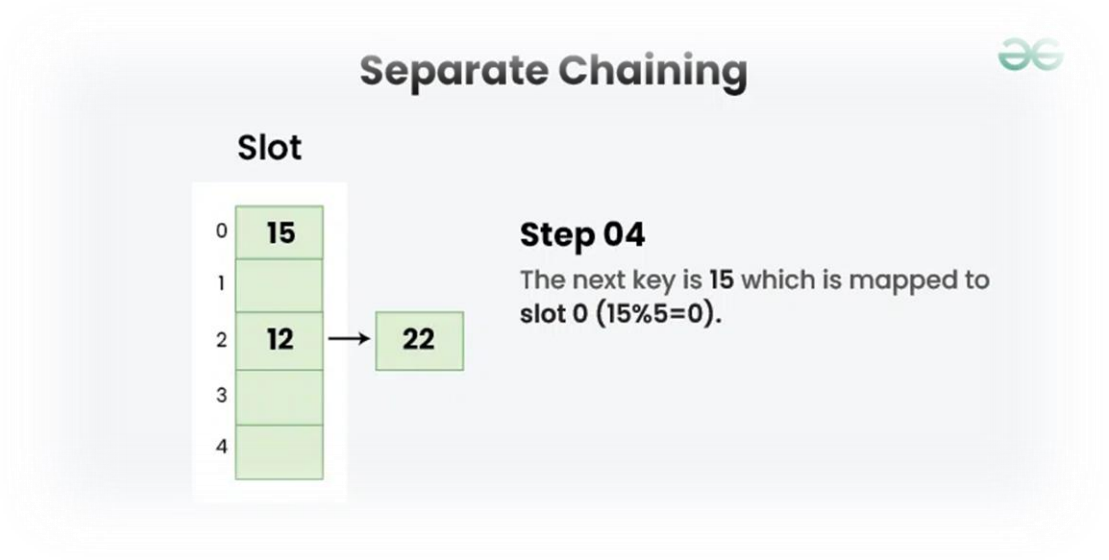
Chaining

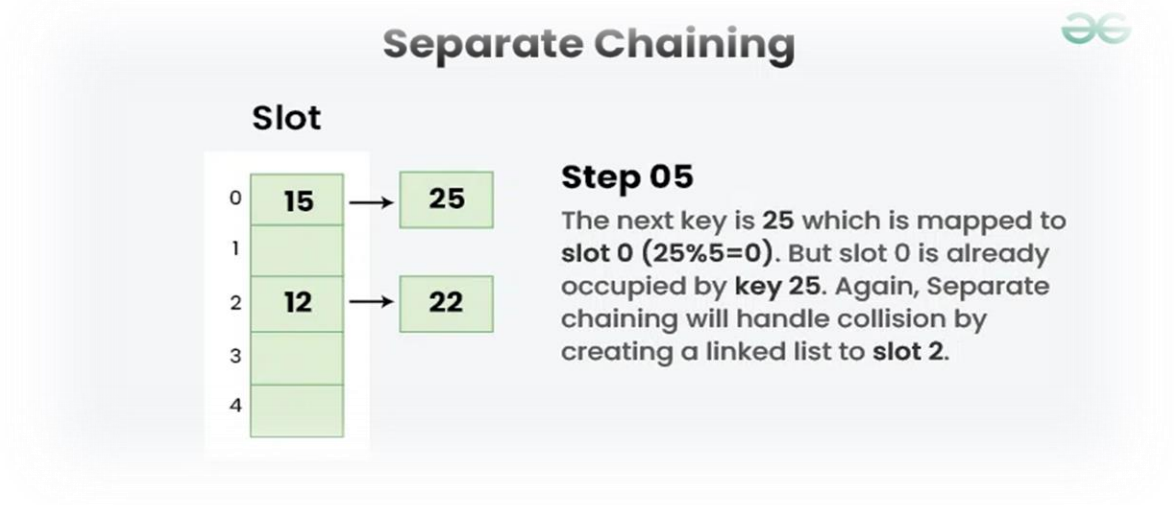
In this technique, a linked list is created from the slot in which collision has occurred, after which the new key is inserted into the linked list. This linked list of slots looks like a chain, so it is called separate chaining.

Example: Let us consider a simple hash function as "key mod 5"

sequence of keys as 12, 22, 15, 25







Linear Probing

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

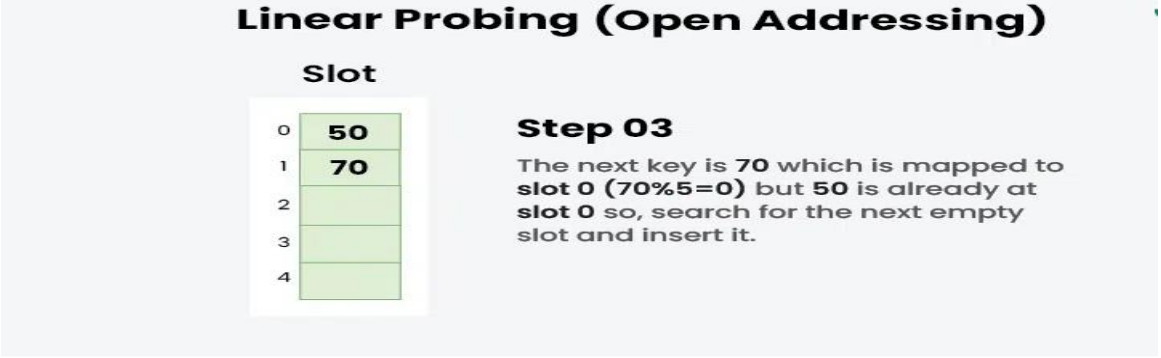
Algorithm:

1. Calculate the hash key. i.e. **key = data % size**
2. Check, if **hashTable[key]** is empty
    - store the value directly by **hashTable[key] = data**
3. If the hash index already has some value then
    - check for next index using **key = (key+1) % size**

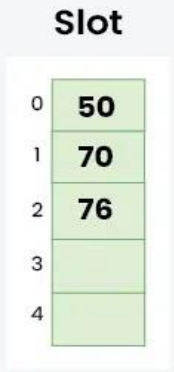
4. Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.
5. Do the above process till we find the space.

Example: Let us consider a simple hash function as “key mod 5”

sequence of keys that are to be inserted are 50, 70, 76, 85, 93.



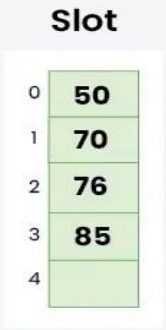
Linear Probing (Open Addressing)



Step 04

The next key is **76** which is mapped to **slot 1 (76%5=1)** but **70** is already at **slot 1** so, search for the next empty slot and insert it.

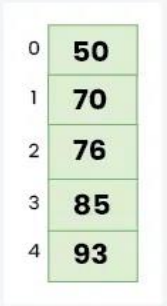
Linear Probing (Open Addressing)



Step 05

The next key is **85** which is mapped to **slot 0 (85%5=0)**, but **50** is already at **slot number 0** so, search for the next empty slot and insert it. So insert it into **slot number 3**.

Linear Probing (Open Addressing)



Step 06

The next key is **93** which is mapped to **slot 3 (93%5=3)**, but **85** is already at **slot 3** so, search for the next empty slot and insert it. So insert it into **slot number 4**.