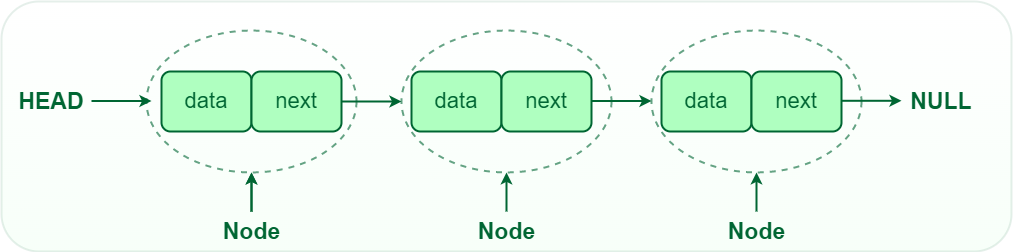
**DSA**

**For time complexity notes refer apna didi notes………..**

**---------------------------------------------------------------Linked List----------------------------------------------**



**What is a Linked List?**

**A linked list is a linear data structure that consists of a series of nodes connected by pointers. Each node contains data and a reference to the next node in the list. Unlike arrays, linked lists allow for efficient insertion or removal of elements from any position in the list, as the nodes are not stored contiguously in memory.**

**Array over linked list**

**A screenshot of a computer

Description automatically generated**

**USES** Music playlists

 Browser history

 Image viewers (for navigation)

 Operating system task scheduling

[**Types of Linked List**](https://www.geeksforgeeks.org/types-of-linked-list/)

[**Singly Linked List**](https://www.geeksforgeeks.org/data-structures/linked-list/singly-linked-list/)

A white rectangular object with black text

Description automatically generated

[**Doubly Linked List**](https://www.geeksforgeeks.org/doubly-linked-list/)

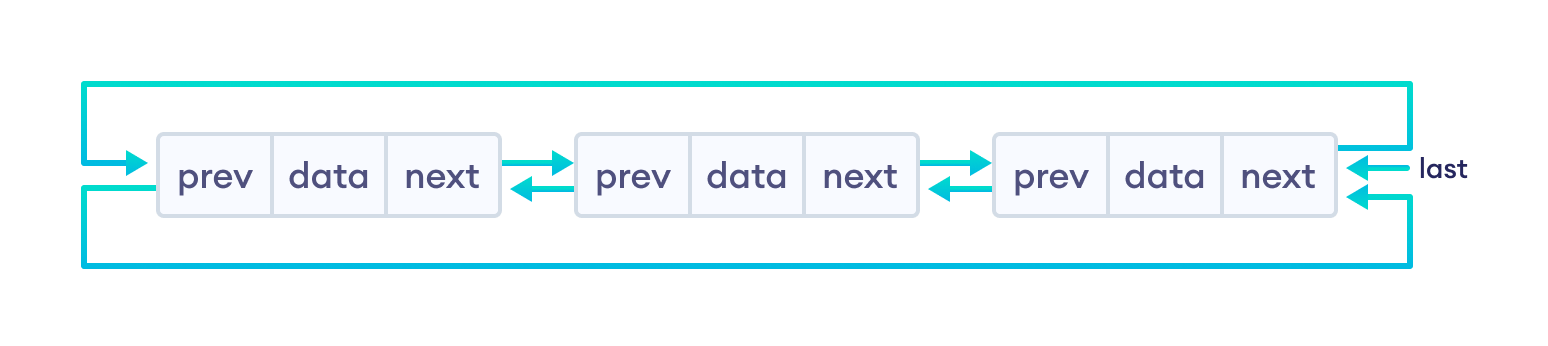
**A diagram of a flowchart

Description automatically generated**

[**Circular Linked List**](https://www.geeksforgeeks.org/circular-linked-list/)

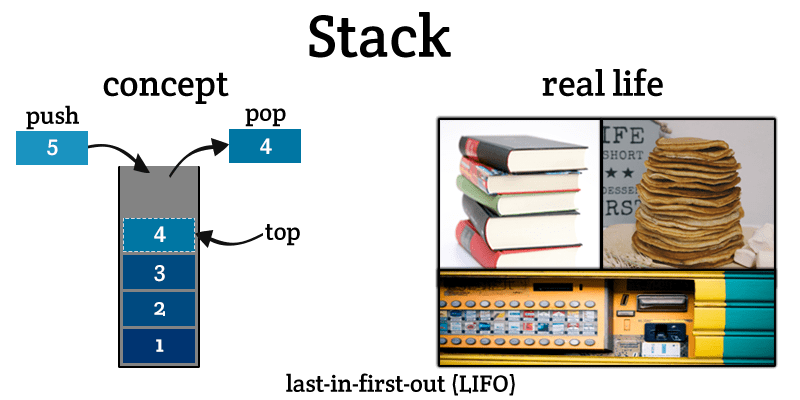


[**Circular Doubly Linked List**](https://www.geeksforgeeks.org/introduction-to-circular-doubly-linked-list/)



**Stack**

**Stacks are last-in-first-out (LIFO) structures**– The first element added to a stack is the last one to be removed and the last element added to a stack is the first to be removed.



Operations on stack:

**– Push** – Pushes (stores) an element on the stack.

*How it works*: Checks if the stack is full. If it is full, produces an overflow error and exit. If the stack is not full, then increments top to point the next empty space. Adds data element to the stack location, where the top is pointing and returns success.

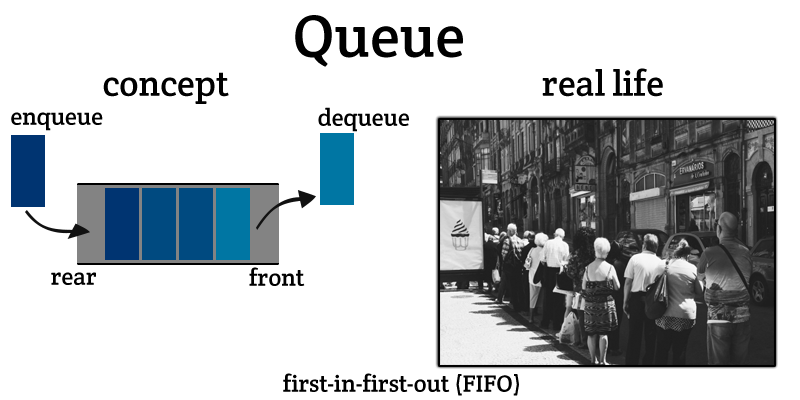
**– Pop** – Removes (accesses) an element from the stack.

*How it works:* Checks if the stack is empty. If it is empty, produces an underflow error and exits. If the stack is not empty, then accesses the data element at which the top is pointing. Decreases the value of top by 1 and returns success.

* Stacks:
  + Backtracking
  + Expression conversion and parsing
  + Undo command

**QUEUE**

**Queues are first-in-first-out (FIFO) structures** – The first element added to a queue is the first to be removed and the last element to be added to a queue will be the last to be removed.



Operations on queue:

– **Enqueue**− Adds (stores) an item to the queue.

*How it works*: Checks if the queue is full. If the queue is full, produces overflow error and exit. If the queue is not full, increments the rear pointer to point the next empty space. Adds data element to the queue location, where the rear is pointing and returns success.

– **Dequeue** − Removes (accesses) an item from the queue.

*How it works*: Checks if the queue is empty. If the queue is empty, produces underflow error and exit. If the queue is not empty, accesses the data where the front is pointing. Increments front pointer to point to the next available data element and returns success.

Two pointers are used to refer the front and rear end of the queue.

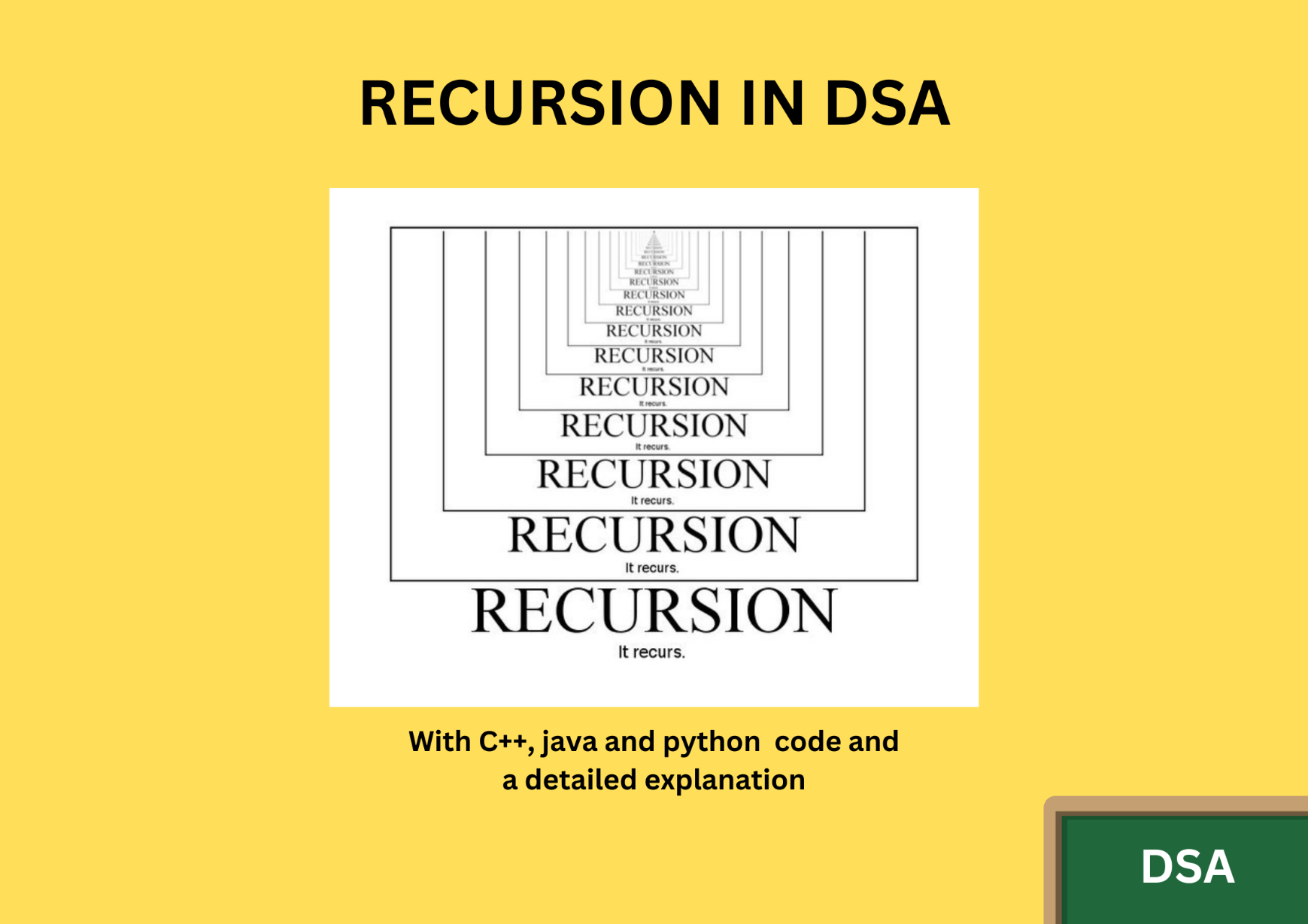
Random access is not allowed in both stacks and queues. You cannot add or remove an element from the middle.

* Queues:
  + CPU scheduling
  + Synchronization
  + Handling of interrupts in real-time systems

**Recursion**

when function call itself then that

function is called as recursive function..



**Example code:**

**def loop(n):**

**if n > 0:**

**print(n)**

**loop(n-1) // same function**

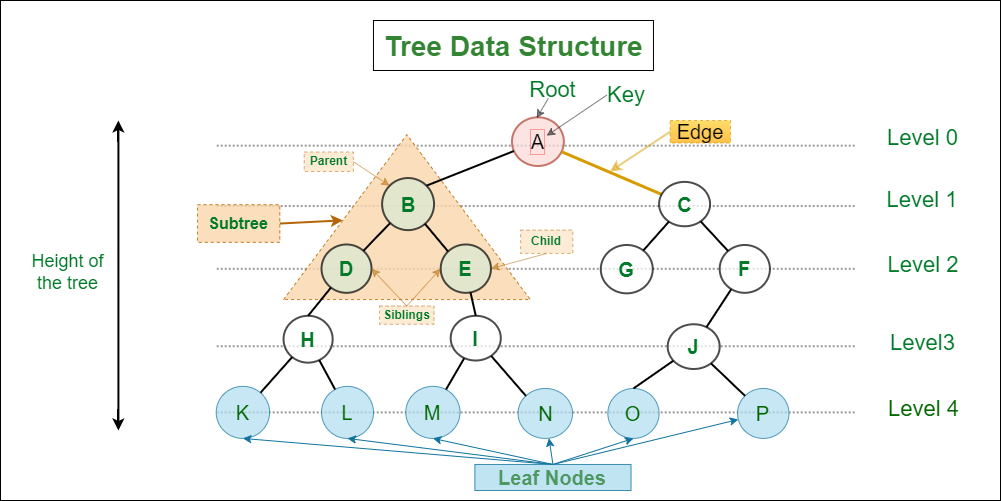
**print(loop(5))**

**Trees**

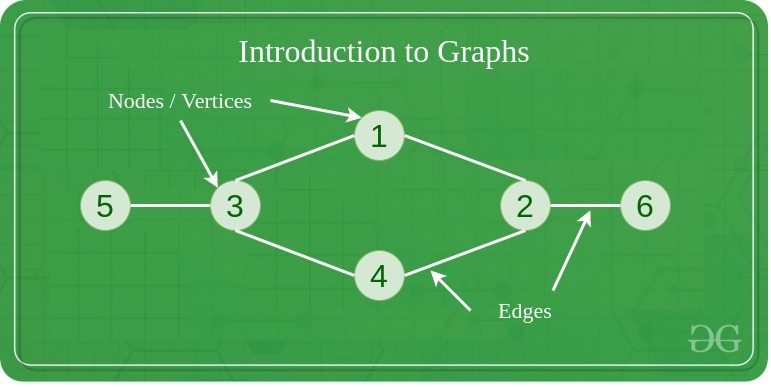
 A **tree** is a hierarchical data structure consisting of nodes, with a single node called the root and zero or more child nodes.

 **Terminology:**

* **Root:** The topmost node in the tree.
* **Parent:** A node that has one or more child nodes.
* **Child:** A node that has a parent node.
* **Leaf:** A node with no children.
* **Subtree:** A tree formed by a node and its descendants.
* **Depth:** The length of the path from the root to the node.
* **Height:** The length of the path from the node to the deepest leaf.



**Graphs**



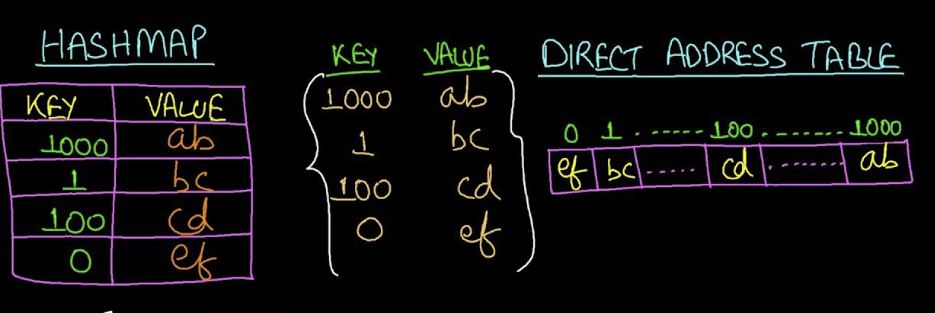
Types of Graph

The graph can be classified into different categories, the most common one is a directed or undirected graph.

**1. Directed graph**A directed graph is a type of graph in which edges are unidirectional or they point towards a particular node to which it connects.

**2. Undirected graph:**An undirected graph is a type of graph in which edges don’t point to either of the attached nodes.

**HashMap**



**Common Hash Map Operations**

| **Operation** | **Description** | **Complexity (Average)** |
| --- | --- | --- |
| set | Insert or update a key-value pair | O(1) |
| get | Retrieve the value for a given key | O(1) |
| delete | Remove a key-value pair | O(1) |
| has | Check if a key exists in the map | O(1) |
| size | Get the number of key-value pairs | O(1) |

**Example Use Case**

**What is a Hash Map?**

A **hash map** (or hash table) is a data structure used to store key-value pairs. It allows for quick retrieval of data based on a key, using a hash function to compute an index into an array of buckets or slots. This provides an efficient way to look up, add, and remove items.

In JavaScript, hash maps can be implemented using:

1. **Objects**
2. **Map** (introduced in ES6)

// Creating a hash map using Map

const hashMap = new Map();

// Adding key-value pairs

hashMap.set("name", "John Doe");

hashMap.set("age", 30);

hashMap.set("city", "New York");

// Accessing values using keys

console.log(hashMap.get("name")); // Output: John Doe

console.log(hashMap.get("age")); // Output: 30

// Removing a key-value pair

hashMap.delete("city");

// Checking if a key exists

if (hashMap.has("city")) {

  console.log("City exists in hash map");

} else {

  console.log("City does not exist in hash map");

}

// Iterating over the hash map

for (const [key, value] of hashMap) {

  console.log(`${key}: ${value}`);

}

// Output:

// John Doe

// 30

// City does not exist in hash map

// name: John Doe

// age: 30