Stack

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle, meaning that the last element added to the stack is the first one to be removed. You can think of a stack like a stack of plates: you add new plates to the top and also remove plates from the top.

Basic Operations:
A stack typically supports the following operations:
1. **Push**: Add an element to the top of the stack.
2. **Pop**: Remove and return the top element from the stack.
3. **Peek** (or **Top**): Return the top element without removing it.
4. **isEmpty**: Check if the stack is empty.

Advantages of Using a Stack
1. **Simple and Efficient**:
   - Stacks are straightforward to implement and use, and their operations (push, pop, and peek) have a time complexity of O(1), making them very efficient.
2. **Memory Management**:
   - Stacks are used in memory management to keep track of function calls and local variables. The call stack is a fundamental part of how programming languages handle function calls.
3. **Backtracking**:
   - Stacks are useful for backtracking algorithms (like solving mazes or puzzles), where you can push decisions onto the stack and pop them when backtracking is needed.
4. **Expression Evaluation**:
   - Stacks are employed in parsing and evaluating expressions (like converting infix expressions to postfix or prefix). They help manage operator precedence and associativity.
5. **Undo Mechanism**:
   - Many applications (like text editors) use stacks to implement undo functionality. Each action is pushed onto the stack, and popping from the stack reverses the last action.
6. **Function Call Management**:
   - Stacks keep track of active function calls, handling local variables and managing the order of execution in recursive functions.

Disadvantages of Stacks
1. **Limited Access**:
   - You can only access the top element, not those in the middle or bottom.
2. **Memory Issues**:
   - **Stack Overflow**: If too much data is added, it can crash the program.
   - **Fixed Size**: If a stack has a set limit, it can fill up quickly and can't accept more data.
3. **Not for All Data Types**:
   - Stacks are best for specific tasks and might not be suitable for general data storage.
4. **Operation Overhead**:
   - Dynamically sized stacks can slow down due to memory reallocation.
5. **Complex Implementation**:
   - In multithreading scenarios, managing access to the stack can be complicated.
6. **Inefficient Searching**:
   - Finding an element in a stack can take time, as you may need to pop all elements to check.
7. **Context-Sensitive**:
   - Stacks depend on the order of operations, which may limit their flexibility for other uses.

Queue

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. This means that the first element added to the queue will be the first one to be removed, much like a line of people waiting at a ticket counter.
Basic Operations:
A queue typically supports the following operations:
1. **Enqueue**: Add an element to the end of the queue.
2. **Dequeue**: Remove and return the front element from the queue.

3. **Peek** (or **Front**): Return the front element without removing it.
4. **isEmpty**: Check if the queue is empty.

applications of queues:

1. Task Scheduling

- **Operating Systems**: Queues are used to manage processes in operating systems. Processes waiting for CPU time are placed in a queue, ensuring they are executed in the order they arrived.

2. Job Scheduling in Print Spoolers

- In print spooling, print jobs are placed in a queue, allowing them to be processed in the order they were received. This ensures that documents are printed sequentially.

3. Breadth-First Search (BFS)

- In graph algorithms, queues are used for breadth-first search to explore nodes level by level. Each node's neighbors are enqueued for future exploration.

4. Data Buffering

- Queues are used in data streaming and buffering, where data is temporarily stored before being processed. This is common in video streaming, audio processing, and network communication.

5. Message Queuing

- In distributed systems, queues facilitate communication between different components or services. Message queues enable asynchronous message passing, ensuring reliable delivery of messages.

6. Event Handling

- Queues are used in event-driven programming to manage events. Events are placed in a queue and processed in the order they occur, allowing for responsive user interfaces.

7. Simulations

- Queues are commonly used in simulations of real-world systems, such as customer service lines, traffic flow, or network traffic, to model waiting times and resource allocation.

8. Web Server Request Handling

- Web servers use queues to manage incoming requests. Each request is placed in a queue, and the server processes them one at a time or in batches, ensuring orderly handling of client requests.

9. Call Center Systems

- Queues are used in call center systems to manage incoming calls. Calls are placed in a queue and answered in the order they arrive, providing efficient customer service.

10. Multithreading and Concurrency

- Queues are often used in multithreading environments to coordinate work between producer and consumer threads. Producers add tasks to the queue, while consumers retrieve and process them.

disadvantages :

1. **Limited access to elements** – Only the front and rear elements are directly accessible.
2. **Fixed size in array implementation** – Array-based queues have a fixed size, leading to potential space limitations or wastage.
3. **Inefficiency in array-based dequeue** – Dequeue operations may require shifting elements, resulting in O(n) time complexity.
4. **Not suitable for random access** – Queues do not support direct access to arbitrary elements.
5. **No prioritization** – Standard queues cannot prioritize elements (priority queues or other structures may be required).
6. **Memory overhead in linked list implementation** – Linked list-based queues require extra memory for pointers.
7. **Lack of bidirectional traversal** – Standard queues only support front-to-rear operations, unlike deques.

Stable Algorithms

A stable algorithm preserves the **relative order** of elements with equal values. This means that if two elements are equal, their original order in the input will be maintained in the output.

Unstable Algorithms

An unstable algorithm **does not guarantee** preserving the relative order of elements with equal values. It might change their relative positions during the algorithm's execution.

Deterministic Algorithms

A **deterministic algorithm** is one that, for a given input, will always produce the **same output** and follow the exact **same sequence of steps** on every execution.

Non-Deterministic Algorithms

A **non-deterministic algorithm** can produce different outcomes even for the same input on different executions. These algorithms may involve randomization, choices, or multiple possible execution paths, which means they do not follow a fixed sequence of operations.

Divide and Conquer

The **Divide and Conquer** method is a strategy used to solve problems by breaking them down into smaller, more manageable sub-problems. Once these sub-problems are solved, their solutions are combined to solve the original, larger problem.

Stack Overflow & Stack Underflow

Stack Overflow and **Stack Underflow** are terms that describe two types of errors that can occur when working with the **stack data structure** or a program's call stack.

1. Stack Overflow

Stack overflow occurs when there is an attempt to **push (add)** more items onto the stack than it can hold, or when a program's call stack grows beyond its limit (usually in a recursive function with no base case or excessive recursion depth).

2. Stack Underflow

Stack underflow occurs when there is an attempt to **pop (remove)** an item from an empty stack, which means the stack is already empty, but an operation tries to access an element that isn't there.

hash table

A **hash table** (or hash map) is a data structure that implements an associative array, allowing for efficient storage and retrieval of key-value pairs. It uses a **hash function** to compute an index (or hash code) into an array of buckets or slots, from which the desired value can be found.

Advantages :

1. **Fast Lookups**:
   - Average-case time complexity for lookups is O(1) due to direct access via hash codes.
   - This makes hash tables very efficient for retrieval operations.
2. **Dynamic Size**:
   - Hash tables can dynamically resize to accommodate more elements, which helps manage memory efficiently.
   - When the load factor exceeds a certain threshold, the hash table can be resized and rehashed.
3. **Efficient Insertions and Deletions**:
   - Average-case time complexity for insertions and deletions is also O(1), making it efficient for managing data.
4. **Flexible Key Types**:
   - Hash tables can support various key types, allowing the use of strings, integers, and custom objects as keys.
5. **Good Performance with Proper Implementation**:
   - With a well-designed hash function and an appropriate collision resolution strategy, hash tables can provide excellent performance.

Disadvantages :

1. **Collisions**:
   - When multiple keys hash to the same index, it leads to collisions, which can degrade performance.
   - Handling collisions requires additional strategies (e.g., chaining, open addressing), complicating the implementation.
2. **Poor Worst-Case Performance**:
   - In the worst case, such as a poorly chosen hash function or excessive collisions, the time complexity for lookups, insertions, and deletions can degrade to O(n).
3. **Memory Usage**:
   - Hash tables may require more memory than other data structures (like arrays or linked lists) due to the need for extra slots to handle collisions and maintain efficiency.
   - They can also suffer from wasted space if not properly resized.
4. **Requires a Good Hash Function**:
   - The performance of a hash table heavily depends on the quality of the hash function. A poor hash function can lead to clustering and inefficient operations.
5. **Not Suitable for Ordered Data**:
   - Hash tables do not maintain any order among the keys. If you need to access elements in a sorted order, other data structures like trees or linked lists may be more appropriate.

Applications :

1. Database Indexing:

Used for fast retrieval of records using keys like user IDs.
2. Caching:
Maps keys (like URLs) to cached content for quick access.
3. Symbol Tables in Compilers:
Stores variable names and their values or memory locations in compilers and interpreters.
4. Dictionary Implementation:
Powers dictionaries in languages like Python (dict) and JavaScript (Object, Map).
5. Counting Frequency:
Used to count occurrences of items (e.g., word frequency in a document).
6. Associative Arrays:
Associates keys with values (e.g., country names to capitals).
7. Routing Tables:
Stores IP addresses and corresponding routes for network packet forwarding.
8. Spell Checking:
Stores valid words in a dictionary for fast spell-check lookups.
9. Password Storage:
Stores hashed passwords for secure authentication.
10. Load Balancing:
Distributes requests among servers in a distributed system.
11. File Systems:
Maps file names to locations for quick file retrieval.
12. Data Deduplication:
Ensures unique data by hashing content and eliminating duplicates.
13. **Cryptography**:
Used in digital signatures, message integrity checks, and password hashing.
14. DNS Resolution:
Maps domain names to IP addresses in DNS servers.
15. Games and Simulations:
Stores game states and assets, improving performance in simulations and engines like chess.

Hash Function (_hash)

This function converts a string key into an index by summing the ASCII values of each character in the key and then taking the remainder when divided by the table size.

The **ASCII value** is a numerical representation of characters in the **ASCII** (American Standard Code for Information Interchange) encoding scheme.

Hash values

- **Hash values** are fixed-size outputs produced by hash functions that uniquely represent input data.
- They are commonly used in hash tables, data integrity checks, password storage, and cryptographic operations.

collision

A **collision** in a hash table occurs when two different keys produce the same hash value (index) after being processed by the hash function. Since hash tables use hash values to determine where to store each key-value pair in an underlying array, collisions can lead to multiple keys being mapped to the same index, complicating data retrieval and storage.

methods manage collisions

1. **Separate Chaining:**

- In separate chaining, each index in the hash table points to a linked list (or another data structure) where all elements that hash to the same index are stored. This allows multiple keys to exist at the same index without collisions affecting data retrieval.

1. **Open Addressing**:

- In this method, if a collision occurs, the hash table seeks the next available slot according to a probing sequence. This can be done in several ways:
  - **Linear Probing**: Checks the next slot sequentially.
  - **Quadratic Probing**: Uses a quadratic function to find the next slot (e.g., checking 1, 4, 9 slots away).
  - **Double Hashing**: Applies a second hash function to find the next slot.

1. Linear Probing

Linear Probing is the simplest form of open addressing. When a collision occurs, the algorithm checks the next slot in the array sequentially until it finds an empty slot.

How It Works:

- Calculate the hash index for the key.
- If the calculated index is occupied, move to the next index (i.e., index + 1, index + 2, etc.) until an empty slot is found.
- If the end of the array is reached, wrap around to the beginning of the array.

2. Quadratic Probing

Quadratic Probing improves upon linear probing by using a quadratic function to determine the next index to check when a collision occurs. This reduces clustering by spreading the probes more evenly.

How It Works:

- Calculate the hash index for the key.
- If the slot is occupied, probe the next slots using the formula:
  - index + 1^2, index + 2^2, index + 3^2, and so on.
- Wrap around if the end of the array is reached.

3. Double Hashing

Double Hashing uses two hash functions to calculate the next probe index when a collision occurs. This method is more complex but provides better distribution of keys and minimizes clustering.

How It Works:

- Use the primary hash function to calculate the initial index.
- If the slot is occupied, calculate a step size using a second hash function.
- The next index to probe is then calculated as index + step, wrapping around as necessary

load factor

The **load factor** in a hash table is a measure of how full the table is or efficiency of the table, indicating the ratio b/w the number of stored elements (keys) and the total number of slots (buckets) available in the table.

Time Complexity

- **Insertion**:
  - **Average Case**: O(1) – If a good hash function is used and the load factor is kept low, insertion will typically take constant time.
  - **Worst Case**: O(n) – In the worst case (e.g., when many collisions occur), all keys may hash to the same index, leading to a linear search through all entries.
- **Deletion**:
  - **Average Case**: O(1) – Similar to insertion, deletion generally takes constant time on average.
  - **Worst Case**: O(n) – If a collision resolution method leads to long chains or probing sequences, deletion can degrade to linear time.
- **Search**:
  - **Average Case**: O(1) – Searching for a value is typically very fast due to direct access using the hash code.
  - **Worst Case**: O(n) – As with insertion and deletion, if the hash table experiences a high number of collisions, the search operation can take linear time.