**Basics**

**Data Structures and Concepts**

1. Linear vs Non-Linear Data Structures
2. Stack (concept)
3. Queue (concept)
4. Hash Table (concept)
5. Types of Queues:
   * Circular Queue
   * Priority Queue
   * Double-Ended Queue

**Operations and Basic Implementations**

**Stack Operations**

1. Push, Pop, and Display Operations
2. Reverse a String Using a Stack
3. Stack Pointer (Purpose)
4. Stack Overflow vs Underflow

**Queue Operations**

1. Enqueue, Dequeue, and Display Operations
2. Queue Using Linked List
3. Stack Using Linked List

**Hash Table Operations**

1. Hash Table Implementation (set, delete, get, display)
2. Applications of Hash Table

**Sorting Algorithms**

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Stable Sorting Algorithms
5. In-place Sorting
6. Merge Sort
7. Quick Sort
8. Time Complexity of All Sorting Algorithms

**Intermediate Topics**

**Linked Lists**

1. Merge Two Sorted Linked Lists
2. Remove Middle Element from a Linked List

**Hash Table and Collision Resolution**

1. Hash Collision
2. Methods to Resolve Hash Collision:
   * Separate Chaining
   * Linear Probing
   * Quadratic Probing
   * Double Hashing
   * Open Addressing

**Queue Variants**

1. Circular Queue Implementation
2. Priority Queue Implementation
3. Queue Using Stack
4. Convert Stack Into Queue

**Applications**

1. Parenthesis Checking Using Stack
2. Sort a Stack
3. Hash Table to Find Two Numbers in an Array That Add Up to a Target Sum

**Advanced Topics**

**Sorting Algorithms**

1. Perform Merge Sort on an Array of Strings
2. Merge Two Sorted Arrays in O(n) Time Complexity
3. Why Merge Sort Is Preferred for Linked Lists
4. Worst-Case Complexity of Quick Sort
5. Choosing an Appropriate Sorting Algorithm
6. Does Pivot Affect Quick Sort Performance?
7. Quick Sort Without Additional Arrays

**Hash Table and Hashing**

1. Load Factor
2. Rehashing
3. Perfect Hash Function
4. Hashing vs Encryption
5. Popular Hashing Algorithms (SHA1, MD5, CRC32)

**Advanced Data Structures**

1. Monotonic Stack
2. Monotonic Queue
3. Double-Ended Queue

**Conceptual and Miscellaneous**

1. Divide and Conquer
2. Disadvantages of Merge Sort
3. Binary Recursion
4. Applications of All Data Structures
5. Improve Logic and Coding Speed

**Practice and Problem Solving**

1. Valid Parentheses
2. Two Sum
3. Merge Two Sorted Lists
4. Reverse a String Using Stack
5. Practice Problems From Blind 75 (Neetcode.io)

**Basics**

**Data Structures and Concepts**

1. Linear vs Non-Linear Data Structures

**Linear Data Structures**: Elements are arranged in a sequential manner, allowing for straightforward traversal and access. Examples include arrays, linked lists, stacks, and queues.

1. Stack (concept)

A stack is a linear data structure that follows the Last In First Out (LIFO) principle, meaning the last element added is the first one to be removed. It supports two main operations: push (to add an element) and pop (to remove an element).

1. Queue (concept)

A Queue is a linear data structure that follows the First In First Out (FIFO) principle, meaning that the first element added to the queue will be the first one to be removed. It is analogous to a real-life queue, such as people waiting in line at a ticket counter, where the person who arrives first is served first.

Key Characteristics of a Queue:

FIFO Order: The order of processing is strictly maintained; elements are added at the back (rear) and removed from the front.

Operations:

* Enqueue: Adding an element to the back of the queue.
* Dequeue: Removing an element from the front of the queue.
* Peek/Front: Viewing the front element without removing it.
* IsEmpty: Checking if the queue is empty.
* Size: Getting the number of elements in the queue.

Types of Queues:

* Simple Queue: Basic implementation of a queue with standard operations.
* Circular Queue: The last position is connected back to the first position to make efficient use of space.
* Priority Queue: Elements are removed based on priority rather than the order they were added.
* Double-Ended Queue (Deque): Elements can be added or removed from both the front and the back.

Applications of Queues:

* Managing tasks in a printer queue.
* Handling requests in web servers.
* Implementing breadth-first search (BFS) in graph algorithms.
* Scheduling processes in operating systems.

1. Hash Table (concept)

A Hash Table (or Hash Map) is a data structure that implements an associative array, a structure that can map keys to values. 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.

Key Characteristics of a Hash Table:

* Key-Value Pairs: Data is stored in pairs, where each key is unique and maps to a specific value.
* Hash Function: A function that takes an input (the key) and returns an integer (the hash code), which determines the index in the array where the value is stored.
* Collision Handling: When two keys hash to the same index, a collision occurs. Common methods to handle collisions include:
* Chaining: Each index in the array points to a linked list of entries that hash to the same index.
* Open Addressing: When a collision occurs, the algorithm searches for the next available slot in the array.

Operations:

* Insert: Add a key-value pair to the hash table.
* Search: Retrieve the value associated with a given key.
* Delete: Remove a key-value pair from the hash table.

Time Complexity:

* Average Case: O(1) for insert, search, and delete operations.
* Worst Case: O(n) (when many collisions occur, leading to a linked list of entries).

Applications of Hash Tables:

* Implementing associative arrays and dictionaries.
* Caching data for quick access.
* Counting occurrences of items (e.g., word frequency in text).
* Storing unique items (e.g., user IDs).

1. Types of Queues:
   * Circular Queue
   * Priority Queue
   * Double-Ended Queue

**Operations and Basic Implementations**

**Stack Operations**

1. Push, Pop, and Display Operations
2. Reverse a String Using a Stack

const reverseWords= function(s){

    const splitS = s.split(" ");

    const stack = []

    for(let i of splitS){

        stack.push(i);

    }

    let finalS ="";

    while(stack.length){

        const current = stack.pop()

        if(current){

            finalS += " " +current

        }

    }

    return finalS.trim()

}

console.log(reverseWords("the sky is blue"))

1. Stack Pointer (Purpose)

The stack pointer (SP) is a special-purpose register in computer architecture that keeps track of the top of the stack in memory. Its primary purposes include:

* 1. Memory Management: The stack pointer indicates the current position of the top of the stack, allowing the system to manage memory efficiently for function calls, local variables, and control flow.
  2. Function Calls and Returns: When a function is called, the stack pointer is adjusted to allocate space for the function's local variables and return address. When the function returns, the stack pointer is adjusted back to its previous position.
  3. Data Storage: The stack pointer helps in storing temporary data, such as function parameters, return addresses, and local variables, which are essential for the execution of programs.
  4. Context Switching: In multitasking operating systems, the stack pointer is crucial for saving and restoring the state of a process during context switches.
  5. Error Handling: The stack pointer can be used to track the call stack, which is useful for debugging and error handling, as it provides a trace of function calls leading to an error.

1. Stack Overflow vs Underflow

**Stack Overflow**

Definition: Stack overflow occurs when an attempt is made to push an element onto a stack that is already full. This means that there is no more space available in the stack to accommodate additional elements.

Causes:

* Exceeding the maximum size of the stack, which can be defined by the system or the programming language.
* Infinite recursion in function calls, where a function keeps calling itself without a base case, leading to excessive use of stack space.

**Stack Underflow**

Definition: Stack underflow occurs when an attempt is made to pop an element from an empty stack. This means that there are no elements available to remove.

Causes:

* Popping elements from the stack without first checking if it contains any elements.
* Mismanagement of stack operations, such as excessive popping after a series of pushes that did not fill the stack.

**Queue Operations**

1. Enqueue, Dequeue, and Display Operations
2. Queue Using Linked List
3. Stack Using Linked List

**Hash Table Operations**

1. Hash Table Implementation (set, delete, get, display)
2. Applications of Hash Table

**Sorting Algorithms**

1. Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted. The name "Bubble Sort" comes from the way larger elements "bubble" to the top of the list (or the end of the array) with each pass.

How Bubble Sort Works:

* Start at the beginning of the array.
* Compare the first two adjacent elements.
* If the first element is greater than the second, swap them.
* Move to the next pair of adjacent elements and repeat the comparison and swap if necessary.
* Continue this process for the entire array. After the first pass, the largest element will be at the end of the array.
* Repeat the process for the remaining unsorted elements until no swaps are needed, indicating that the array is sorted.

Time Complexity:

* Best Case: O(n) (when the array is already sorted)
* Average Case: O(n²)
* Worst Case: O(n²)

Space Complexity:

* O(1) (in-place sorting)

1. Insertion Sort

Insertion Sort is a simple and intuitive comparison-based sorting algorithm that builds a sorted array (or list) one element at a time. It is much like the way you might sort playing cards in your hands. The algorithm divides the array into a sorted and an unsorted part, and it iteratively takes elements from the unsorted part and inserts them into the correct position in the sorted part.

How Insertion Sort Works:

* Start with the second element (the first element is considered sorted).
* Compare the current element with the elements in the sorted part (to its left).
* Shift all elements in the sorted part that are greater than the current element to the right.
* Insert the current element into its correct position in the sorted part.
* Repeat the process for all elements in the array until the entire array is sorted.

Time Complexity:

* Best Case: O(n) (when the array is already sorted)
* Average Case: O(n²)
* Worst Case: O(n²)

Space Complexity:

* O(1) (in-place sorting)

1. Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It works by dividing the input list into two parts: a sorted part and an unsorted part. The algorithm repeatedly selects the smallest (or largest, depending on the order) element from the unsorted part and moves it to the end of the sorted part.

How Selection Sort Works:

1. Start with the first element as the minimum.
2. Compare this minimum with the other elements in the array to find the smallest element.
3. Swap the smallest element found with the first element of the unsorted part.
4. Move the boundary of the sorted and unsorted parts one element to the right.
5. Repeat the process for the remaining unsorted elements until the entire array is sorted.

Time Complexity:

* Best Case: O(n²)
* Average Case: O(n²)
* Worst Case: O(n²)

Space Complexity:

* O(1) (in-place sorting)

### ****Advantages****

* Simple to understand and implement.
* Performs well on small datasets.

### ****Disadvantages****

* Inefficient on large datasets.
* Not stable, as swapping can change the relative order of equal elements.

1. Stable Sorting Algorithms
2. In-place Sorting

An **in-place sorting algorithm** is an algorithm that transforms the input data into a sorted order using a constant amount of extra memory, i.e., O(1) additional space. These algorithms perform sorting directly on the original data structure without requiring a significant amount of additional storage.

### ****Key Characteristics****

1. **Memory Usage:**
   * Uses only a small, constant amount of extra space (O(1)O(1)O(1)).
   * Operates directly on the input array by rearranging elements.
2. **Efficiency:**
   * In-place sorting is space-efficient but not always the most time-efficient.
3. **Examples:**
   * Algorithms like **Bubble Sort** and **Insertion Sort** are examples of in-place sorting.
4. Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm that efficiently sorts an array by dividing it into smaller subarrays, sorting those subarrays, and then merging them back together. It is particularly effective for large datasets and is known for its stable sorting properties.

How Merge Sort Works:

Divide:

* If the array has one or zero elements, it is already sorted. Otherwise, split the array into two halves.

Conquer:

* Recursively apply the merge sort to both halves of the array. This process continues until all subarrays are reduced to single elements.

Merge:

* Once the subarrays are sorted, merge them back together. During the merge process, compare the elements of the two subarrays and arrange them in order, creating a new sorted array.

Repeat:

* Continue merging until all subarrays are combined into a single sorted array.

Example:

* For an array [38, 27, 43, 3, 9, 82, 10], the process would look like this:
* Split into [38, 27, 43] and [3, 9, 82, 10]
* Further split [38, 27, 43] into [38] and [27, 43], and then [27, 43] into [27] and [43]
* Merge [27] and [43] to get [27, 43], then merge with [38] to get [27, 38, 43]
* Similarly, split and merge the second half [3, 9, 82, 10] to get [3, 9, 10, 82]
* Finally, merge the two sorted halves [27, 38, 43] and [3, 9, 10, 82] to get the fully sorted array.

Time Complexity:

* Best Case: O(n log n) (when the array is already sorted)
* Average Case: O(n log n)
* Worst Case: O(n log n)
* Space Complexity:
* O(n) (due to the temporary arrays used for merging)

1. Quick Sort

Quick Sort is a highly efficient sorting algorithm that uses a divide-and-conquer approach to sort elements. It works by selecting a "pivot" element from the array and partitioning the other elements into two subarrays according to whether they are less than or greater than the pivot. The subarrays are then sorted recursively.

How Quick Sort Works:

Choose a Pivot:

* Select an element from the array to serve as the pivot. Various strategies can be used for choosing the pivot, such as picking the first element, the last element, the middle element, or using a random element.

Partitioning:

* Rearrange the array so that all elements less than the pivot come before it and all elements greater than the pivot come after it. The pivot is then in its final position.

Recursively Sort Subarrays:

* Apply the same process recursively to the subarrays formed by splitting the array at the pivot. This continues until the base case is reached, where the subarray has one or zero elements (which are inherently sorted).

Combine:

* Since the subarrays are sorted in place, no additional merging is needed. The final sorted array is formed as the recursive calls return.

Example:

* For an array [10, 80, 30, 90, 40, 50, 70], the process would look like this:
* Choose a pivot (e.g., 70).
* Partition the array into [10, 30, 40, 50] (less than 70) and [80, 90] (greater than 70), resulting in [10, 30, 40, 50, 70, 80, 90].
* Recursively apply quick sort to the left subarray [10, 30, 40, 50] and the right subarray [80, 90].
* Continue partitioning and sorting until all subarrays are sorted.

Time Complexity:

* Best Case: O(n log n) (when the pivot divides the array into two equal halves)
* Average Case: O(n log n)
* Worst Case: O(n²) (when the pivot is the smallest or largest element, leading to unbalanced partitions)

Space Complexity:

* O(log n) (for the recursive stack space in the best and average cases)
* O(n) in the worst case (if the recursion depth goes to n)

1. Time Complexity of All Sorting Algorithms

1. Bubble Sort

* Best Case: O(n) (when the array is already sorted)
* Average Case: O(n²)
* Worst Case: O(n²)

2. Selection Sort

* Best Case: O(n²)
* Average Case: O(n²)
* Worst Case: O(n²)

3. Insertion Sort

* Best Case: O(n) (when the array is already sorted)
* Average Case: O(n²)
* Worst Case: O(n²)

4. Merge Sort

* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n log n)

5. Quick Sort

* Best Case: O(n log n) (when the pivot divides the array into two equal halves)
* Average Case: O(n log n)
* Worst Case: O(n²) (when the pivot is the smallest or largest element)

**Intermediate Topics**

**Linked Lists**

1. Merge Two Sorted Linked Lists
2. Remove Middle Element from a Linked List

**Hash Table and Collision Resolution**

1. Hash Collision

A hash collision occurs when two different keys produce the same hash value when processed by a hash function. Since hash tables use these hash values to determine where to store or retrieve data, collisions can lead to issues in data retrieval and storage.

Key Concepts:

Hash Function:

* A hash function takes an input (or "key") and returns a fixed-size string of bytes. The output is typically a hash code, which is an integer that represents the input data. A good hash function minimizes the chances of collisions by distributing keys uniformly across the hash table.

Collision:

* A collision happens when two distinct keys hash to the same index in the hash table. For example, if both keys "apple" and "banana" hash to the same index, a collision occurs

1. Methods to Resolve Hash Collision:
   * Separate Chaining

When two different keys hash to the same index in a hash table, a collision occurs. There are several methods to resolve these collisions, each with its own advantages and disadvantages. Below are the most common methods:

1. Separate Chaining

Description: In separate chaining, each index of the hash table contains a linked list (or another data structure) that holds all the entries that hash to that index. When a collision occurs, the new entry is simply added to the list at that index.

Advantages:

* + 1. Simple to implement.
    2. Can handle a large number of collisions without degrading performance significantly.
    3. The hash table can grow dynamically as more entries are added.

Disadvantages:

* + 1. Requires additional memory for the linked lists.
    2. Performance can degrade if many collisions occur, leading to longer search times.
  + Linear Probing

Description: In linear probing, when a collision occurs, the algorithm checks the next slot in the hash table sequentially until an empty slot is found.

Advantages:

* + 1. Simple to implement.
    2. All elements are stored in the hash table itself, which can lead to better cache performance.

Disadvantages:

* + 1. Can lead to clustering, where groups of consecutive occupied slots form, increasing the likelihood of further collisions.
    2. Performance degrades as the table fills up.
  + Quadratic Probing

Description: Quadratic probing is similar to linear probing, but instead of checking the next slot sequentially, it checks slots at intervals of the square of the probe number (1, 4, 9, etc.).

Advantages:

* + - Reduces clustering compared to linear probing.
    - Can be more efficient in terms of space utilization.

Disadvantages:

* + 1. Still susceptible to clustering, though less so than linear probing.
    2. More complex to implement than linear probing.
  + Double Hashing

Description: Double hashing uses a second hash function to determine the step size for probing. When a collision occurs, the algorithm applies the second hash function to the key to find the next slot.

Advantages:

* + - Reduces clustering significantly.
    - More efficient than linear and quadratic probing in many cases.

Disadvantages:

* + 1. More complex to implement due to the need for two hash functions.
    2. Requires careful selection of the second hash function to ensure it does not produce a step size of zero.
  + Open Addressing

Description: Open addressing is a general method for collision resolution where all elements are stored directly in the hash table. When a collision occurs, the algorithm searches for the next available slot using a probing method (linear probing, quadratic probing, or double hashing).

Advantages:

* No additional data structures are needed, which can save memory.
* Can be more space-efficient than separate chaining.

Disadvantages:

* Performance can degrade significantly as the table fills up (high load factor).
* More complex to implement than separate chaining.

**Queue Variants**

1. Circular Queue Implementation
2. Priority Queue Implementation
3. Queue Using Stack
4. Convert Stack Into Queue

**Applications**

1. Parenthesis Checking Using Stack
2. Sort a Stack
3. Hash Table to Find Two Numbers in an Array That Add Up to a Target Sum

**Advanced Topics**

**Sorting Algorithms**

1. Perform Merge Sort on an Array of Strings
2. Merge Two Sorted Arrays in O(n) Time Complexity
3. Why Merge Sort Is Preferred for Linked Lists
4. Worst-Case Complexity of Quick Sort
5. Choosing an Appropriate Sorting Algorithm
6. Does Pivot Affect Quick Sort Performance?
7. Quick Sort Without Additional Arrays

**Hash Table and Hashing**

1. Load Factor
2. Rehashing
3. Perfect Hash Function
4. Hashing vs Encryption
5. Popular Hashing Algorithms (SHA1, MD5, CRC32)

**Advanced Data Structures**

1. Monotonic Stack
2. Monotonic Queue
3. Double-Ended Queue

**Conceptual and Miscellaneous**

1. Divide and Conquer
2. Disadvantages of Merge Sort
3. Binary Recursion
4. Applications of All Data Structures
5. Improve Logic and Coding Speed

**Practice and Problem Solving**

1. Valid Parentheses
2. Two Sum
3. Merge Two Sorted Lists
4. Reverse a String Using Stack
5. Practice Problems From Blind 75 (Neetcode.io)

**Extra Points:-**

1. how stack used in undo –redo
2. **call stack**