**Analysis criteria of sorting algorithm**

1. **Time complexity- O(n2) O(nlogn)**
2. **Space complexity-in place sorting algorithm**
3. **Stability- eg: 6,1,16-> 1,1,6,6 first 6 comes first.**
4. **Internal and External sorting algorithm- all data loaded into the memory is internal and not loaded is external.**
5. **Adaptive- Already sorted data takes less time.**
6. **Recursive and non recursive SA**

**Stability**

**Stability refers to whether the algorithm maintains the relative order of equal elements in the sorted output. In other words, if two elements in the input array are equal, does the algorithm guarantee that their relative order will be preserved in the sorted array?**

# Selection Sort Algorithm

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

## Complexity Analysis of Selection Sort

**Time Complexity:** The time complexity of Selection Sort is **O(N2)** as there are two nested loops:

* One loop to select an element of Array one by one = O(N)
* Another loop to compare that element with every other Array element = O(N)
* Therefore overall complexity = O(N) \* O(N) = O(N\*N) = O(N2)

**Auxiliary Space:** O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

## Advantages of Selection Sort Algorithm

* Simple and easy to understand.
* Works well with small datasets.

## ****Disadvantages of the Selection Sort Algorithm****

* Selection sort has a time complexity of O(n^2) in the worst and average case.
* Does not work well on large datasets.
* Does not preserve the relative order of items with equal keys which means it is not stable.

## Bubble sort

## Bubble sort is a [sorting algorithm](https://builtin.com/machine-learning/fastest-sorting-algorithm) that starts from the first element of an array and compares it with the second element. If the first element is greater than the second, we swap them. It continues this process until the end of the array, with the largest elements “bubbling” to the top. Its worst-case [time complexity](https://builtin.com/software-engineering-perspectives/time-complexity) is O(n²), and its best-case time complexity is O(n).

## [Complexity Analysis of Bubble Sort](https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/):

**TimeComplexity:**O(N2)  
**Auxiliary Space:** O(1)

## ****Advantages of Bubble Sort:****

* Bubble sort is easy to understand and implement.
* It does not require any additional memory space.
* It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

## ****Disadvantages of Bubble Sort:****

* Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
* Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

**Insertion sort**

**Time Complexity:** O(N^2)   
**Auxiliary Space:**O(1)

## [Complexity Analysis of Insertion Sort](https://www.geeksforgeeks.org/time-and-space-complexity-of-insertion-sort-algorithm/):

### ****Time Complexity of Insertion Sort****

* **Best case:** **O(n)**, If the list is already sorted, where n is the number of elements in the list.
* **Average case:** **O(n2)**, If the list is randomly ordered
* **Worst case:** **O(n2)**, If the list is in reverse order

### Space Complexity ****of Insertion Sort****

* **Auxiliary Space:**O(1), Insertion sort requires**O(1)** additional space, making it a space-efficient sorting algorithm.

## Advantages ****of Insertion Sort:****

* Simple and easy to implement.
* Stable sorting algorithm.
* Efficient for small lists and nearly sorted lists.
* Space-efficient.

## Disadvantages ****of Insertion Sort:****

* Inefficient for large lists.
* Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

## Applications ****of Insertion Sort:****

Insertion sort is commonly used in situations where:

* The list is small or nearly sorted.
* Simplicity and stability are important

## ***Quick sort***

## ***QuickSort****is a sorting algorithm based on the*[*Divide and Conquer algorithm*](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/)*that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.*

*The key process in****quickSort****is a****partition()****. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.*

*Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.*

## [Complexity Analysis of Quick Sort](https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/):

**Time Complexity:**

* **Best Case**: Ω (N log (N))  
  The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.  
  In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
* **Average Case: θ ( N log (N))**  
  Quicksort’s average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
* **Worst Case: O(N2)**  
  The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort ) to shuffle the element before sorting.
* **Auxiliary Space:** O(1), if we don’t consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O(N).

## ****Advantages of Quick Sort:****

* It is a divide-and-conquer algorithm that makes it easier to solve problems.
* It is efficient on large data sets.
* It has a low overhead, as it only requires a small amount of memory to function.

## Disadvantages of Quick Sort:

* It has a worst-case time complexity of O(N2), which occurs when the pivot is chosen poorly.
* It is not a good choice for small data sets.
* It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot’s position (without considering their original positions).

Quick Sort is a highly efficient sorting algorithm that follows the Divide and Conquer approach. There are two main types of partition schemes used in Quick Sort:

1. \*\*Lomuto Partition Scheme:\*\*

- This is the simpler and more intuitive partition scheme.

- It chooses the last element of the array as the pivot element.

- It partitions the array into two segments: elements less than the pivot and elements greater than or equal to the pivot.

- This scheme tends to produce more swaps compared to the Hoare partition scheme, especially when the array contains many duplicate elements.

2. \*\*Hoare Partition Scheme:\*\*

- This partition scheme was proposed by Tony Hoare, the inventor of Quick Sort.

- It chooses the first element of the array as the pivot element.

- It partitions the array into two segments: elements less than or equal to the pivot and elements greater than the pivot.

- This scheme generally produces fewer swaps compared to the Lomuto partition scheme and is more efficient in practice, especially for arrays with few duplicate elements.

Both partition schemes are used in various implementations of the Quick Sort algorithm. While Hoare's partition scheme is often considered more efficient, Lomuto's partition scheme is simpler to implement and understand. The choice of partition scheme may depend on factors such as the characteristics of the input data and the specific requirements of the application.

Hoares pqrtition

void swap(List<int> arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

int partition(List<int> arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, high);

return i + 1;

}

void quickSort(List<int> arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void main() {

List<int> arr = [10, 7, 8, 9, 1, 5];

int n = arr.length;

print("Original array: $arr");

quickSort(arr, 0, n - 1);

print("Sorted array: $arr");

}

Lomuto partion

void main(List<String> args) {

  print(quicksort(arr: [10,54,6,2,1,3,9]));

}

List<int>

 quicksort({required List<int>arr}){

  if(arr.length<2){

    return arr;

  }

int pivot=(arr.length/2).floor();

List<int> small=[];

List<int> equal=[];

List<int> large=[];

for(int i in arr){

  if(i<pivot){

    small.add(i);

  }else if(i==pivot){

    equal.add(i);

  }else{

    large.add(i);

  }

}

return [...quicksort(arr: small),...equal,...quicksort(arr: large)];

}

**Merge sort**

**Merge sort** is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer**approach to sort a given array of elements.

Here’s a step-by-step explanation of how merge sort works:

1. **Divide:**Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:**Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

## ****Complexity Analysis of Merge Sort:****

**Time Complexity:**

* **Best Case:**O(n log n), When the array is already sorted or nearly sorted.
* **Average Case:** O(n log n), When the array is randomly ordered.
* **Worst Case:**O(n log n), When the array is sorted in reverse order.

**Space Complexity:**O(n), Additional space is required for the temporary array used during merging.

## ****Advantages of Merge Sort:****

* **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
* **Guaranteed worst-case performance:**Merge sort has a worst-case time complexity of**O(N logN)**, which means it performs well even on large datasets.
* **Simple to implement:**The divide-and-conquer approach is straightforward.

## ****Disadvantage of Merge Sort:****

* **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
* **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.

## Applications of Merge Sort:

* Sorting large datasets
* External sorting (when the dataset is too large to fit in memory)
* Inversion counting (counting the number of inversions in an array)
* Finding the median of an array

Simple method

List<int> mergesort({required List<int>arr}){

  if(arr.length<2){

return arr;

  }

  int mid=(arr.length/2).floor();

  List<int> first=arr.sublist(0,mid);

  List<int> last=arr.sublist(mid);

  return working(first: mergesort(arr: first), last: mergesort(arr: last));

}

List<int> working({required List<int> first,required List<int> last}){

  List<int> sort=[];

  int leftindex=0;

  int rightindex=0;

  while(leftindex<first.length&&rightindex<last.length){

    if(first[leftindex]<last[rightindex]){

      sort.add(first[leftindex]);

      leftindex++;

    }else{

      sort.add(last[rightindex]);

      rightindex++;

    }

  }

  while(leftindex<first.length){

    sort.add(first[leftindex]);

    leftindex++;

  }

  while(rightindex<last.length){

    sort.add(last[rightindex]);

    rightindex++;

  }

  return sort;

}

void main(List<String> args) {

 print( mergesort(arr: [5,4,8,7,9,5,6,1,1]));

}

Advanced method

void merge(List<int> arr, int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

// Create temporary arrays

List<int> L = List<int>.filled(n1, 0);

List<int> R = List<int>.filled(n2, 0);

// Copy data to temporary arrays L[] and R[]

for (int i = 0; i < n1; i++) {

L[i] = arr[l + i];

}

for (int j = 0; j < n2; j++) {

R[j] = arr[m + 1 + j];

}

// Merge the temporary arrays back into arr[l..r]

int i = 0, j = 0;

int k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(List<int> arr, int l, int r) {

if (l < r) {

// Find the middle point

int m = (l + r) ~/ 2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

// Merge the sorted halves

merge(arr, l, m, r);

}

}

void main() {

List<int> arr = [12, 11, 13, 5, 6, 7];

print("Original array: $arr");

mergeSort(arr, 0, arr.length - 1);

print("Sorted array: $arr");

}

The code you provided is another implementation of the Merge Sort algorithm in Dart. While the basic logic remains the same, there are some differences between this implementation and the previous one:

1. \*\*Parameter Passing\*\*: In this implementation, the `mergeSort` function takes parameters `(List<int> arr, int l, int r)`, where `arr` is the array to be sorted, and `l` and `r` are the left and right indices of the sub-array to be sorted, respectively. This allows the function to work directly on the original array without creating sub-lists.

2. \*\*Temporary Arrays\*\*: Instead of creating temporary arrays `L` and `R` inside the `merge` function, this implementation initializes them outside of the function and passes them as parameters. This approach avoids creating temporary arrays for each recursive call, potentially reducing memory overhead.

3. \*\*Midpoint Calculation\*\*: The midpoint `m` is calculated differently in this implementation using the bitwise operator `~/` instead of the `floor()` method. Both approaches achieve the same result.

4. \*\*Loop Conditions\*\*: The loop conditions for merging the sub-arrays are slightly different. In this implementation, the loop conditions `i < n1` and `j < n2` are used instead of `leftindex < first.length` and `rightindex < last.length`.

5. \*\*Printing\*\*: This implementation does not print the original and sorted arrays directly in the `main` function. Instead, it sorts the array using `mergeSort` and then prints the sorted array.

Overall, both implementations achieve the same goal of sorting an array using the Merge Sort algorithm, but they differ in implementation details such as parameter passing, temporary array handling, and loop conditions.

Stack

**Stack**is a **linear data structure**based on based on [LIFO(Last In First Out) principle](https://www.geeksforgeeks.org/lifo-principle-in-stack/)in which the insertion of a new element and removal of an existing element takes place at the same end represented as the **top**of the stack.

To implement the stack, it is required to maintain the **pointer to the top of the stack**, which is the last element to be inserted because **we can access the elements only on the top of the stack.**

### ****LIFO(Last In First Out) Principle in Stack:****

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.

## ****Types of Stack:****

* **Fixed Size Stack**: As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
* **Dynamic Size Stack**: A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

## Basic Operations on Stack:

In order to make manipulations in a stack, there are certain operations provided to us.

* **push()**to insert an element into the stack
* **pop()**to remove an element from the stack
* **top()**Returns the top element of the stack.
* **isEmpty()**returns true if stack is empty else false.
* **isFull()**returns true if the stack is full else false.

## ****Push Operation in Stack:****

Adds an item to the stack. If the stack is full, then it is said to be an **Overflow condition.**

**Algorithm for Push Operation:**

* Before pushing the element to the stack, we check if the stack is **full**.
* If the stack is full **(top == capacity-1)**, then **Stack Overflows**and we cannot insert the element to the stack.
* Otherwise, we increment the value of top by 1 **(top = top + 1)**and the new value is inserted at **top position**.
* The elements can be pushed into the stack till we reach the **capacity**of the stack.

## ****Pop Operation in Stack:****

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**

**Algorithm for Pop Operation:**

* Before popping the element from the stack, we check if the stack is **empty**.
* If the stack is empty (top == -1), then **Stack Underflows**and we cannot remove any element from the stack.
* Otherwise, we store the value at top, decrement the value of top by 1 **(top = top – 1)**and return the stored top value.

## ****Top or Peek Operation in Stack:****

Returns the top element of the stack.

**Algorithm for Top Operation:**

* Before returning the top element from the stack, we check if the stack is empty.
* If the stack is empty (top == -1), we simply print “Stack is empty”.
* Otherwise, we return the element stored at **index = top**.

## ****isEmpty Operation in Stack:****

Returns true if the stack is empty, else false.

**Algorithm for isEmpty Operation**:

* Check for the value of **top**in stack.
* If **(top == -1)**, then the stack is **empty**so return **true**.
* Otherwise, the stack is not empty so return **false**.

## isFull ****Operation in Stack****:

Returns true if the stack is full, else false.

**Algorithm for isFull Operation:**

* Check for the value of **top**in stack.
* If **(top == capacity-1),**then the stack is **full**so return **true**.
* Otherwise, the stack is not full so return **false**.

## Stack Implementation:

The basic operations that can be performed on a stack include push, pop, and peek. There are two ways to implement a stack –

* Using Array
* Using Linked List

In an array-based implementation, the push operation is implemented by incrementing the index of the top element and storing the new element at that index. The pop operation is implemented by storing the element at the top, decrementing the index of the top element and returning the value stored.

In a linked list-based implementation, the push operation is implemented by creating a new node with the new element and setting the next pointer of the current top node to the new node. The pop operation is implemented by setting the next pointer of the current top node to the next node and returning the value of the current top node.

**Advantages of Array Implementation:**

* Easy to implement.
* Memory is saved as pointers are not involved.

**Disadvantages of Array Implementation:**

* It is not dynamic i.e., it doesn’t grow and shrink depending on needs at runtime. [But in case of dynamic sized arrays like vector in C++, list in Python, ArrayList in Java, stacks can grow and shrink with array implementation as well].
* The total size of the stack must be defined beforehand

### Advantages of Linked List implementation:

* The linked list implementation of a stack can grow and shrink according to the needs at runtime.
* It is used in many virtual machines like JVM.

### Disadvantages of Linked List implementation:

* Requires extra memory due to the involvement of pointers.
* Random accessing is not possible in stack.

## ****Complexity Analysis of Stack Operations:****

| **Operations** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| **push()** | O(1) | O(1) |
| **pop()** | O(1) | O(1) |
| **top() or pee k()** | O(1) | O(1) |
| **isEmpty()** | O(1) | O(1) |
| **isFull()** | O(1) | O(1) |

## Advantages of Stack:

* **Simplicity:**Stacks are a simple and easy-to-understand data structure, making them suitable for a wide range of applications.
* **Efficiency:**Push and pop operations on a stack can be performed in constant time **(O(1))**, providing efficient access to data.
* **Last-in, First-out (LIFO):**Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed. This behavior is useful in many scenarios, such as function calls and expression evaluation.
* **Limited memory usage:**Stacks only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

## Disadvantages of Stack:

* **Limited access:**Elements in a stack can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
* **Potential for overflow:**If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.
* **Not suitable for random access:**Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
* **Limited capacity:**Stacks have a fixed capacity, which can be a limitation if the number of elements that need to be stored is unknown or highly variable.

## ****Applications of Stack:****

* [Infix to Postfix](https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/)/Prefix conversion
* Redo-undo features at many places like editors, photoshop.
* Forward and backward features in web browsers
* In Memory management, any modern computer uses a stack as the primary management for a running purpose. Each program that is running in a computer system has its own memory allocations.
* Stack also helps in implementing function call in computers. The last called function is always completed first.

## ****What is Queue?****

**Queue**is a [linear data structure](https://www.geeksforgeeks.org/introduction-to-linear-data-structures/) that is open at both ends and the operations are performed in [First In First Out (FIFO)](https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/) order.

We define a queue to be a list in which all additions to the list are made at one end (**back of the queue**), and all deletions from the list are made at the other end(**front of the queue**).  The element which is first pushed into the order, the delete operation is first performed on that.

[**FIFO Principle of Queue:**](https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/)

* A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).
* Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue). Similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the**tail**) of the queue.

There are different types of queues:

1. **Simple Queue:** Simple Queue simply follows **FIFO**Structure. We can only insert the element at the back and remove the element from the front of the queue.
2. [**Double-Ended Queue (Dequeue)**](https://www.geeksforgeeks.org/deque-set-1-introduction-applications/)**:** In a double-ended queue the insertion and deletion operations, both can be performed from both ends. They are of two types:
   * **Input Restricted Queue:**This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
   * **Output Restricted Queue:** This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
3. [**Circular Queue:**](https://www.geeksforgeeks.org/introduction-to-circular-queue/)This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order.
4. [**Priority Queue**](https://www.geeksforgeeks.org/priority-queue-set-1-introduction/)**:** A priority queue is a special queue where the elements are accessed based on the priority assigned to them. They are of two types:
   * **Ascending Priority Queue:**In Ascending Priority Queue, the elements are arranged in increasing order of their priority values. Element with smallest priority value is popped first.
   * **Descending Priority Queue:**In Descending Priority Queue, the elements are arranged in decreasing order of their priority values. Element with largest priority is popper first.

## [Basic Operations on Queue:](https://www.geeksforgeeks.org/basic-operations-for-queue-in-data-structure/)

Some of the basic operations for Queue in Data Structure are:

1. **Enqueue:** Adds (or stores) an element to the end of the queue..
2. **Dequeue:** Removal of elements from the queue.
3. **Peek or front:** Acquires the data element available at the front node of the queue without deleting it.
4. **rear:** This operation returns the element at the rear end without removing it.
5. **isFull**: Validates if the queue is full.
6. **isEmpty**: Checks if the queue is empty.

There are a few supporting operations (auxiliary operations):

## ****Enqueue Operation in Queue:****

Enqueue() operation in Queue **adds (or stores) an element to the end of the queue**.  
The following steps should be taken to enqueue (insert) data into a queue:

* **Step 1:** Check if the queue is full.
* **Step 2:**If the queue is full, return overflow error and exit.
* **Step 3:** If the queue is not full, increment the rear pointer to point to the next empty space.
* **Step 4:** Add the data element to the queue location, where the rear is pointing.
* **Step 5:** return success.

## ****Dequeue Operation in Queue:****

Removes (or access) the first element from the queue.  
The following steps are taken to perform the dequeue operation:

* **Step 1:** Check if the queue is empty.
* **Step 2:** If the queue is empty, return the underflow error and exit.
* **Step 3:** If the queue is not empty, access the data where the front is pointing.
* **Step 4:** Increment the front pointer to point to the next available data element.
* **Step 5:** The Return success.

## What is Hashing in Data Structure?

**Hashing**is a technique used in data structures to store and retrieve data efficiently. It involves using a **hash function** to map data items to a fixed-size array which is called a**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).

Common hash functions include:

* **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.

**Causes of Hash Collisions:**

* **Poor Hash Function:** A hash function that does not distribute keys evenly across the hash table can lead to more collisions.
* **High Load Factor:**A high load factor (ratio of keys to hash table size) increases the probability of collisions.
* **Similar Keys:** Keys that are similar in value or structure are more likely to collide.

## 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
   * **Cuckoo Hashing:**Use multiple hash functions to distribute keys

## Applications of Hashing

Hash tables are used in a wide variety of applications, including:

* **Databases:** Storing and retrieving data based on unique keys
* **Caching:**Storing frequently accessed data for faster retrieval
* **Symbol Tables:**Mapping identifiers to their values in programming languages
* **Network Routing:**Determining the best path for data packets

Pending

Every sorts best, worst and average time complexity and when it happens.

Sorting criteria should be also checked.

Stack and queue detailed

Hash table theory