EXPLORING DATA STRUCTURES AND ALGORITHMS: DESIGN, OPERATIONS, AND PERFORMANCE

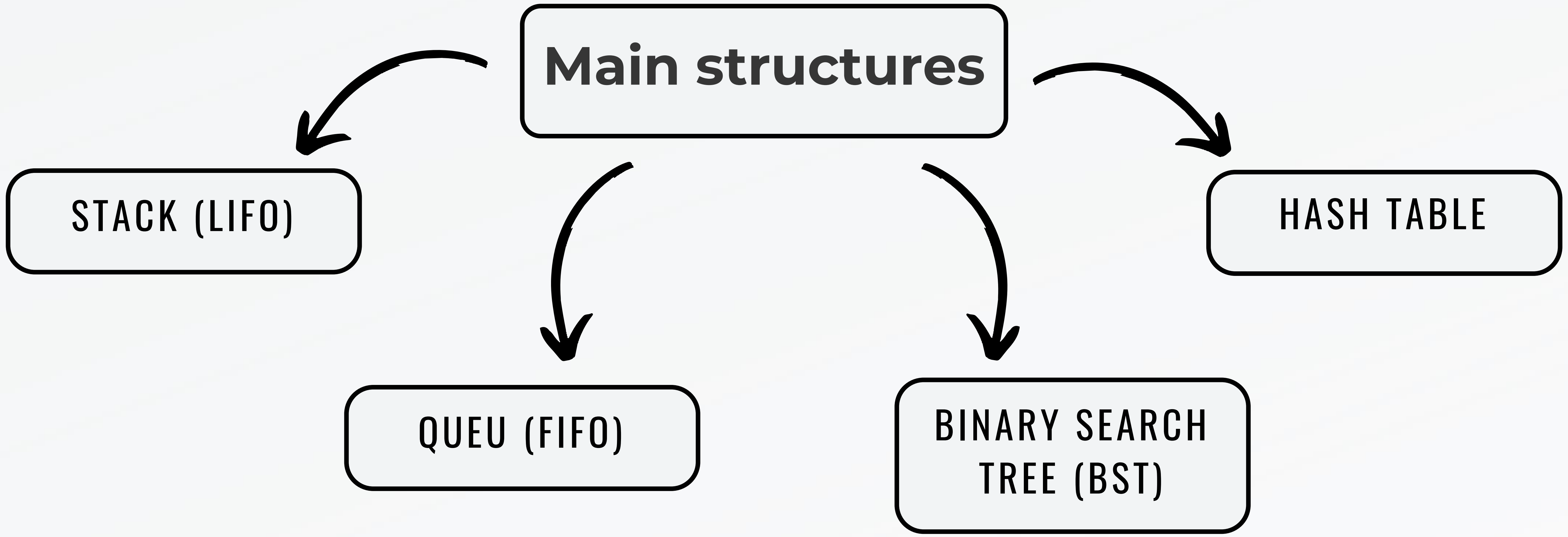
CONTENT

I

DESIGN SPECIFICATION FOR DATA STRUCTURES

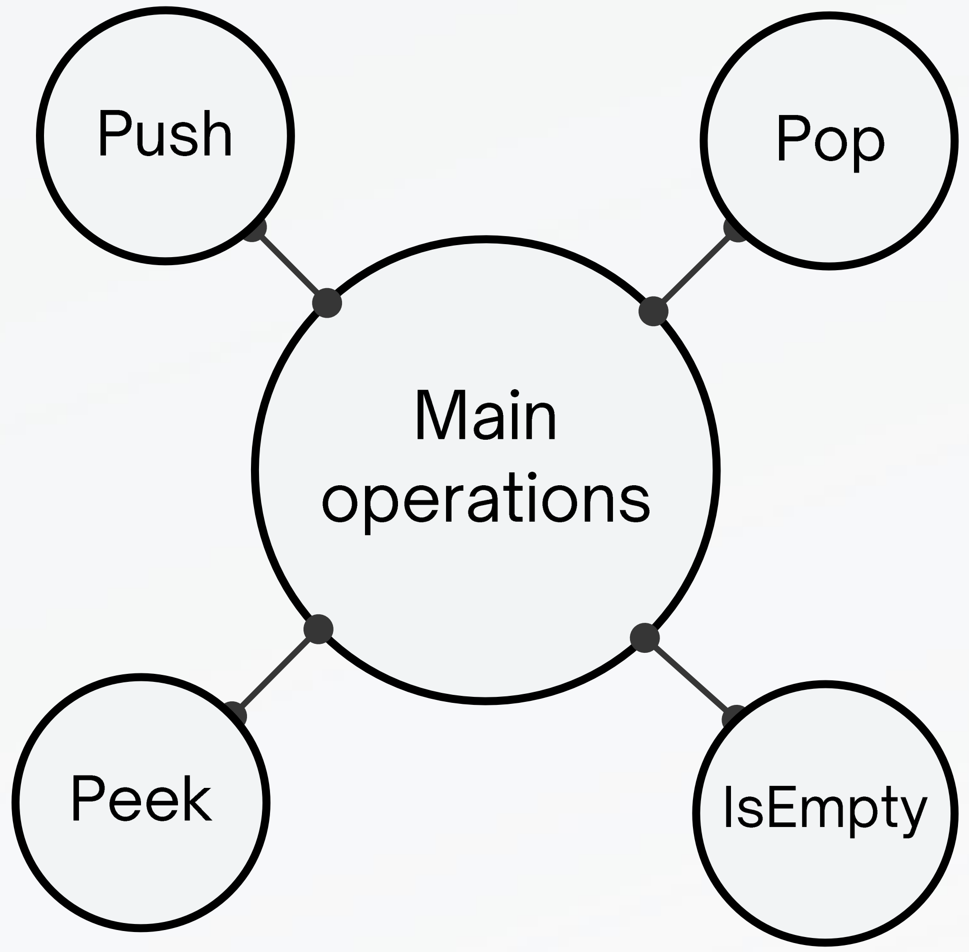
1 INTRODUCTION TO DATA STRUCTURES

Data structures: Are ways of organizing and storing data for efficient processing. They play an important role in optimizing the performance of algorithms and programs.



2 STACK

Definition: A stack is a data structure that allows adding and removing elements at only one end, called the top. It is a part of temporary memory in a computer and plays an important role in making function calls and managing local variables.



3 QUEUE

Definition: A queue is a data structure that allows elements to be added at one end and removed from the other end, operating on the FIFO principle. This means that the first element added will be the first element removed, which is useful in applications that need to manage the order of processing.

Main operations

- ENQUEUE
- DEQUEUE
- PEEK
- ISEMPTY

4 BINARY SEARCH TREE (BST)

Definition: A binary search tree is a tree data structure in which each node has a maximum of two children. The values in the tree are arranged so that all left child nodes have values less than the parent node and all right child nodes have values greater than the parent node.

Main operations

Insert: Add new values to the tree, determine the appropriate position.

Delete: Remove values from the tree, handle special cases.

Search: Search for a value in the tree and return the node containing the value.

InOrderTraversal: Traverse the tree in order, accessing values in ascending order.

5 HASH TABLE

Definition: A hash table is a data structure that stores key-value pairs, allowing fast access through the use of a hash function to map the key to an index in the table. This helps optimize data retrieval and storage time.

Main operations

Insert: Add key-value pairs, hash the key to find the storage index.

Delete: Delete the key-value pair, has no effect if the key does not exist.

Search: Find the value by key, return quickly thanks to the hashed index.

6 TIME AND SPACE COMPLEXITY

Stacks and Queues

Push/Pop/Enqueue/Dequeue: All major operations have O(1) time complexity because they only change the state of an element at the top or tip of the structure.

Binary Search Tree (BST)

Insert/Delete/Search: Time complexity O(h), where h is the height of the tree. In the worst case (unbalanced tree), h can be equal to n (number of elements).

Hash Tables

Insert/Delete/Search: This operation usually has O(1) time complexity in the average case. However, in the case of many conflicts, the complexity can be up to O(n).

II

MEMORY STACK OPERATIONS IN FUNCTION CALLS

1 DEFINITION OF MEMORY STACK

A memory stack is an area of computer memory used to temporarily store information, typically including local variables and return addresses of functions.

Stacks operate on the LIFO principle, meaning that the last element added is the first to be retrieved. This is important for managing function calls and maintaining program state.

2 STACK OPERATIONS

Push	Adds an element to the top of the stack, making it the top element.
Pop	Removes and returns the top element of the stack. This is a very important operation in managing the state of the function.
Peek	Allows viewing the top element without changing the state of the stack.
IsEmpty	Checks if the stack is empty, this helps avoid errors when performing Pop or Peek operations.

3 MAKING A FUNCTION CALL

How it works:

When a function is called, a new stack frame is created.
The stack frame contains:

- Function parameters
- Local variables of the function
- Return address to return to after the function completes.

Process:

Call function: Add a new stack frame with function-specific information.
Execute function: Run instructions using local variables.
Return: Remove the stack frame and return to the stored address.

4 STACK FRAME ILLUSTRATION

Stack Frame:

Each stack frame is a portion of the stack that contains information for a particular function call.

Structure:

- Parameter area
- Local variable area
- Return address area

5 IMPORTANCE OF STACKS

Memory management: Stacks help manage memory space for local variables and parameters in function calls. This helps minimize unnecessary memory usage.

Restoring state: Return addresses allow the program to restore state after a function completes, ensuring that the program can continue running properly.

Ease: Stacks automatically free up memory when a function completes, minimizing memory management errors.

Performance: Accessing data on the stack is faster than other data structures, improving program performance.

III

FIFO QUEUE: EXAMPLE AND IMPLEMENTATION

1 | INTRODUCTION TO FIFO QUEUES

A FIFO (First In, First Out) queue is a data structure that stores and manages elements according to the principle that the first element added is the first element removed.

FIFO queues are often used in situations where the order of processing is important, such as in network programming and task management.

2 STRUCTURE DEFINITION

The FIFO queue structure consists of two indices:

- front: the index of the first element in the queue, which will be removed first.
- rear: the index of the last element in the queue, which will be added to the queue.

Main operations:

- Enqueue: Add an element to the end of the queue.
- Dequeue: Remove and return the element at the front of the queue.
- Peek: View the element at the front of the queue without removing it.
- IsEmpty: Check if the queue is empty.

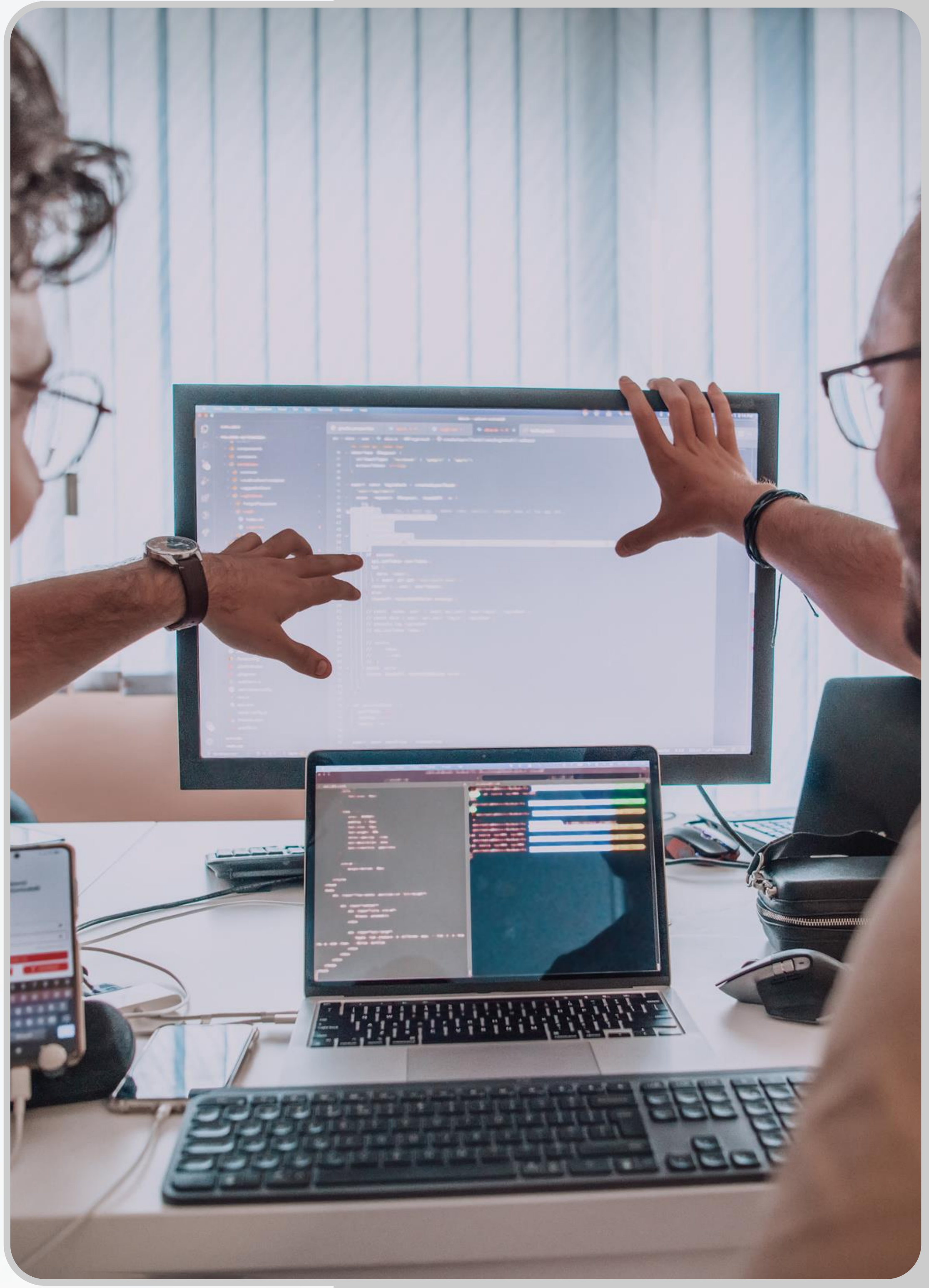
3 ARRAY-BASED IMPLEMENTATION

Implementation:

- Use an array to store the elements of the queue.
- When adding an element (Enqueue), the element will be added to the rear index position and then the rear index will be increased.
- When removing an element (Dequeue), the element at the front index will be removed and the front index will be increased.

Limitations:

- Need to manage the size of the array, because if the array is full, it will not be possible to add new elements.



4 | LINKED LIST-BASED IMPLEMENTATION

Implementation:

- Use a linked list to store queue elements.
- Each node contains data and a pointer to the next node.
- Enqueue: Create a new node and assign it to the current last node's pointer.
- Dequeue: Remove the first node and update the front pointer to the next node.

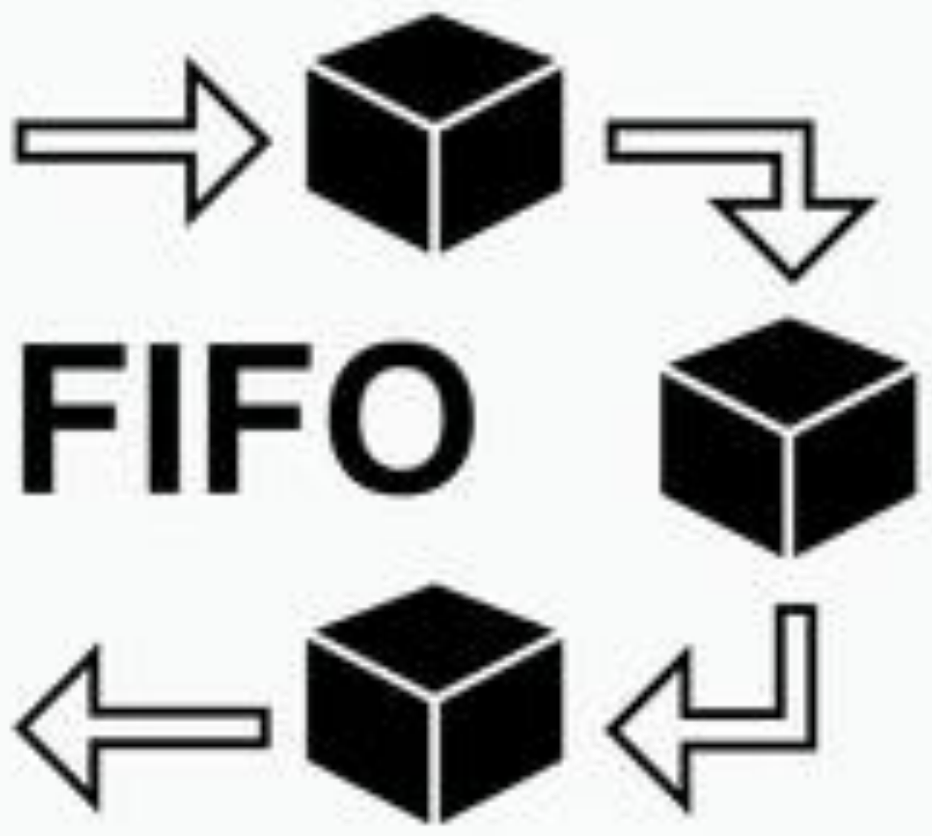
Advantages:

- No need to manage size like in an array; can expand or shrink as needed.

5 FIFO QUEUE CONCRETE EXAMPLE

Let's consider a scenario of managing document printing tasks with a FIFO queue. Suppose we have three documents added to the queue: Document1, Document2, and Document3. When we add Document4, the queue becomes [Document1, Document2, Document3, Document4].

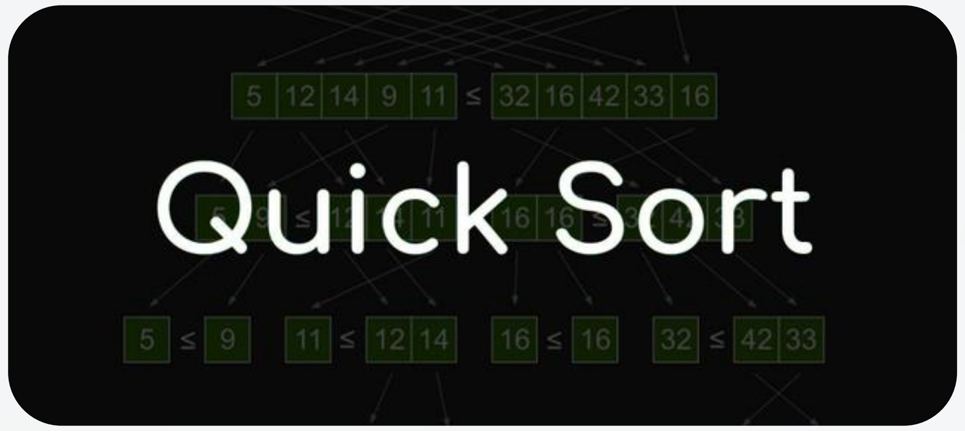
When we perform a Dequeue operation, Document1 is removed, leaving the queue with [Document2, Document3, Document4]. Next, remove Document2, and the queue will now be [Document3, Document4]. The FIFO queue ensures that tasks are processed in the order in which they are added, maintaining an efficient workflow.



IV

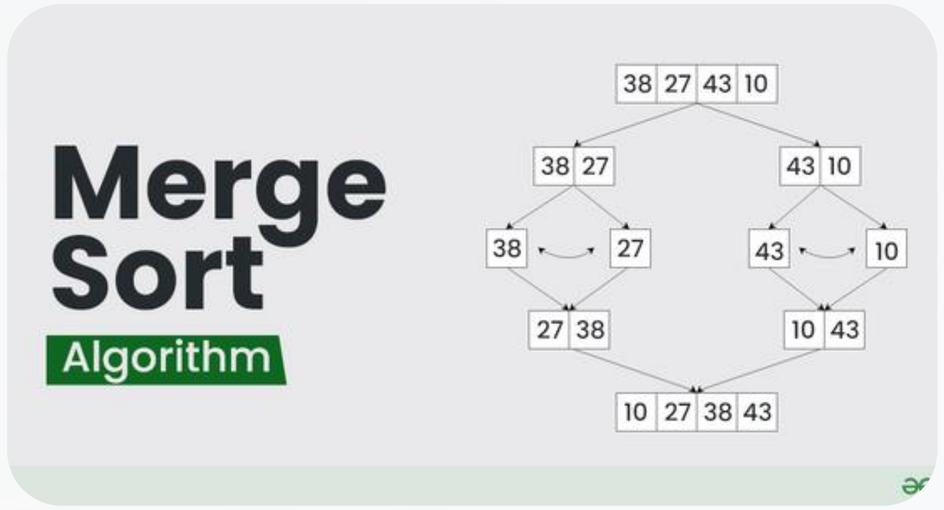
SORTING ALGORITHMS PERFORMANCE COMPARISON

1 INTRODUCTION TO TWO SORTING ALGORITHMS

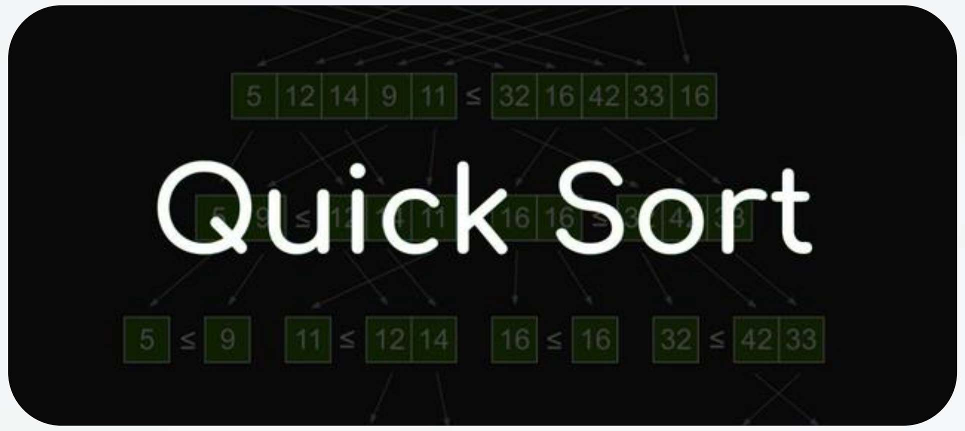


Quick Sort:
- Efficient sorting algorithm using divide and conquer.
- Average time O(n log n), but can be slower in the worst case.

Merge Sort:
- Divide the array into smaller pieces, sort, and merge them.
- Guaranteed time O(n log n) in all cases, but requires more memory.



2 | TIME COMPLEXITY ANALYSIS

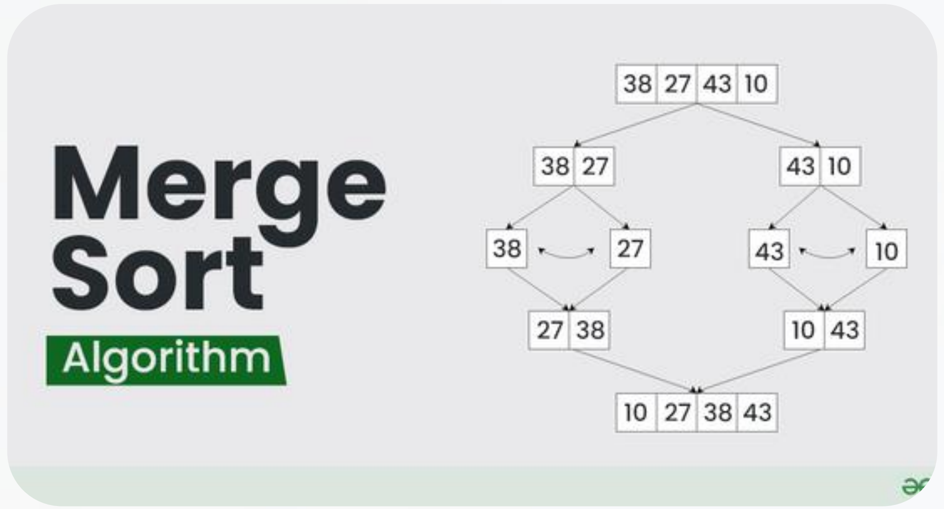


Quick Sort:

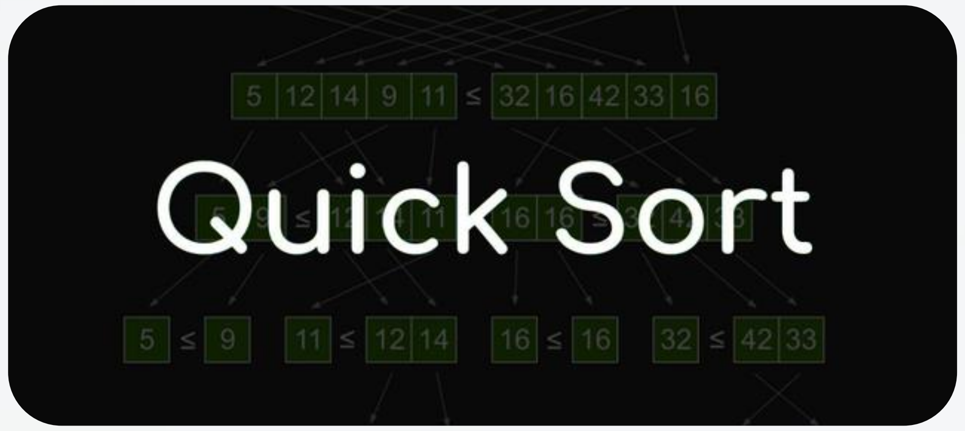
- Best and average time: O(n log n).
- Worst case: O(n²) if pivot is not chosen properly.

Merge Sort:

- Complexity: Always O(n log n) for all cases.
- Consistency: Ensures stable performance in all situations.



3 | SPACE COMPLEXITY ANALYSIS

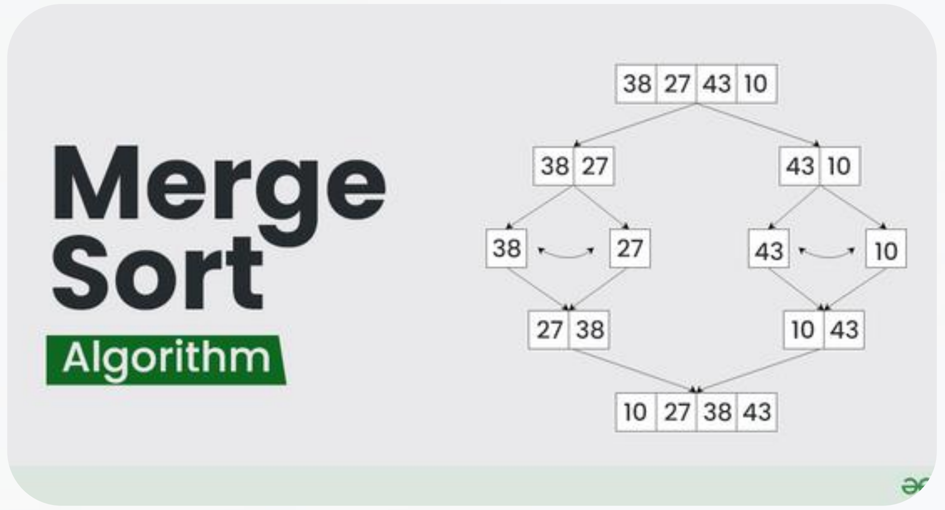


Quick Sort:

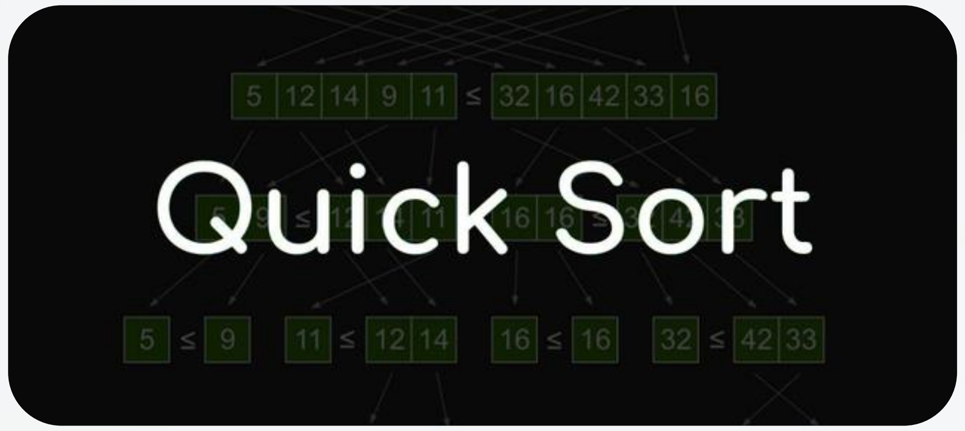
- Best and average time: O(log n) due to recursion.
- Worst case: Can be up to O(n).

Merge Sort:

- Complexity: O(n) due to additional space required to store sub-arrays during merging.
- Disadvantage: Can be a disadvantage in case of large data.



4 ALGORITHM STABILITY

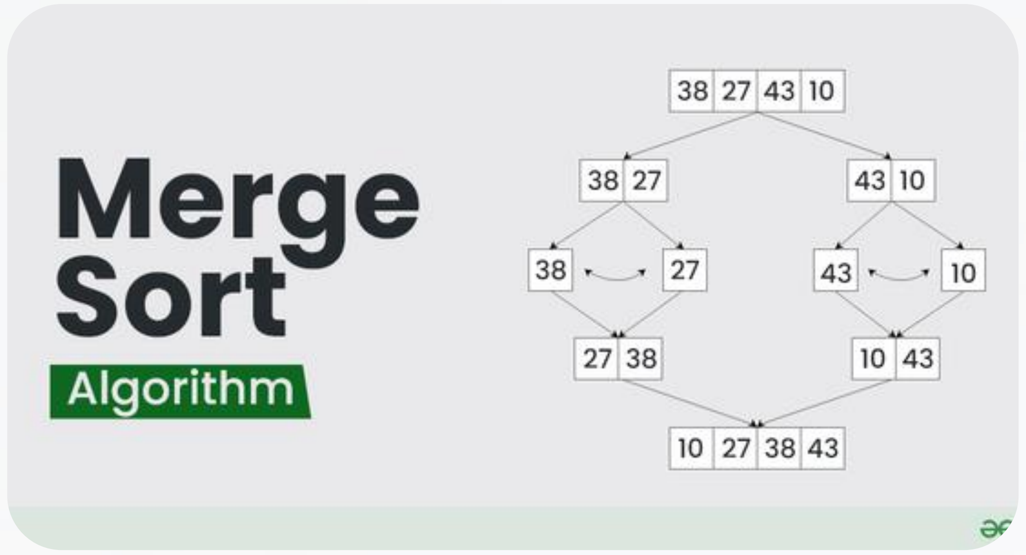


Quick Sort:

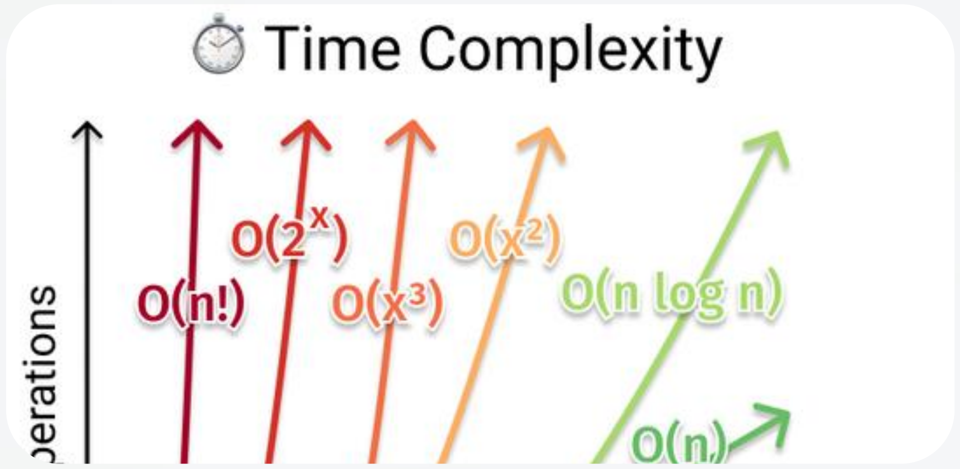
- Stability: Unstable; the order of identical elements may change during sorting.

Merge Sort:

- Stability: A stable algorithm; preserves the order of elements with equal values.
- Applications: Important in many applications where the original order needs to be maintained.



4 COMPARING THE PERFORMANCE OF TWO ALGORITHMS

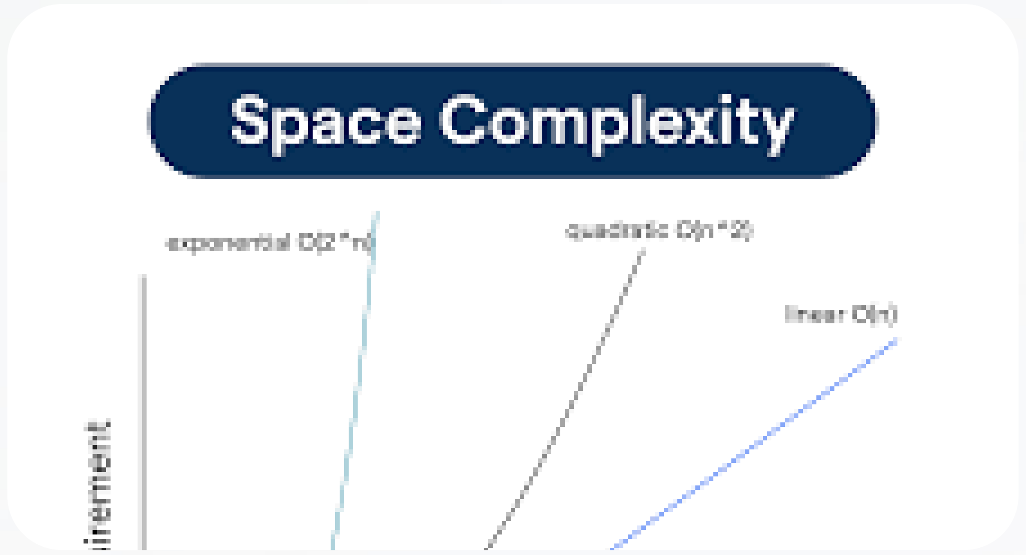


Time complexity:

- Quick Sort: Average performance O(n log n); can go up to $O(n^2)$ in the worst case.
- Merge Sort: Always maintains O(n log n) complexity in all situations.

Space complexity:

- Quick Sort: O(log n) in the best case; can go up to O(n) in the worst case.
- Merge Sort: Requires O(n) to store sub-arrays during merge.



5 | CONCRETE EXAMPLE

Example:
Array to be sorted: [38, 27, 43, 3, 9, 82, 10].

Quick Sort:

- Choose pivot: For example, 38.
- Split the array into elements smaller than (3, 9, 27) and larger than (43, 82).
- Call each element recursively until the array is sorted.

Merge Sort:

- Split the array into two halves: [38, 27, 43] and [3, 9, 82, 10].
- Continue dividing until each halve has only one element left.
- Merge the halves in sorted order.

V

ANALYSIS OF SHORTEST PATH ALGORITHMS

1 INTRODUCTION TO SHORTEST PATH ALGORITHMS IN NETWORKS

Shortest path algorithms are an important aspect of graph theory and its applications in networks. The main goal of these algorithms is to find the shortest path from a source node to all other nodes in a weighted graph

Dijkstra's Algorithm	Finds the shortest path from a source node to all other nodes in the graph.
Prim-Jarnik Algorithm	Focuses on finding the minimum spanning tree for the graph.

2 DIJKSTRA'S ALGORITHM

Dijkstra's Algorithm is one of the famous algorithms for finding the shortest path in a non-negative weighted graph. The algorithm starts from a source node, explores its neighbors one by one, and updates the shortest path lengths to those nodes.

Specifically, the algorithm performs as follows:

1. Initialize the path length from the source to all nodes to infinity, except for the source node, which is set to 0.
2. Use a priority queue to select the node with the shortest path length that has not been explored yet.
3. Update the path lengths for all neighbors of the selected node.
4. Repeat until all nodes have been explored.

3 PRIM-JARNIK ALGORITHM

The Prim-Jarnik algorithm is an algorithm used to find the minimum spanning tree for a weighted graph. The algorithm works by starting at a node and adding edges with the least weight to the spanning tree, until all the nodes in the graph are connected.

The procedure is as follows:

1. Pick any starting node and mark it.
2. Find the least weighted edge connecting the marked nodes to the unmarked nodes.
3. Add that edge to the spanning tree and mark the new node.
4. Repeat until all the nodes are marked.

4 PERFORMANCE ANALYSIS

01 TIME COMPLEXITY

Dijkstra is $O(V^2)$ for adjacency matrix, $O((V + E) \log V)$ for priority queue. Prim-Jarnik is $O(E \log V)$ for priority queue.

02 USE CASES

Dijkstra is good for finding shortest path from a source node. Prim-Jarnik is used to build minimum spanning tree in network.

03 ACCURACY

Both algorithms are accurate; Dijkstra is used for graphs with no negative weights, while Prim-Jarnik is used for all weighted graphs.

THANK YOU!