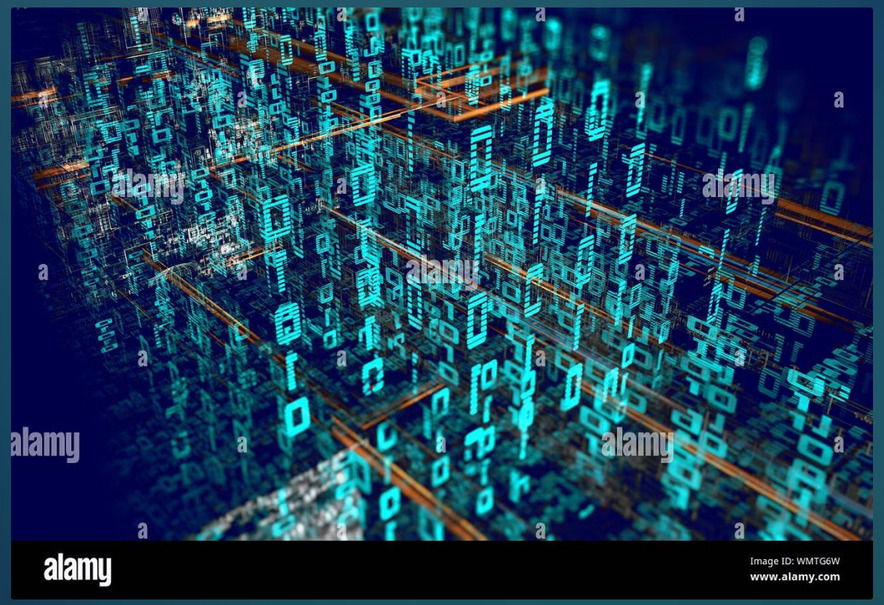


Stack ADT, Sorting Algorithms, and Shortest Path Algorithms

Exploring fundamental concepts in computer science.

Essential tools for efficient data management and problem-solving.

Course Objectives

1 **Master Stack ADT**

Understand implementation and applications of Stack data structure.

2 **Sorting Algorithms**

Learn various sorting techniques and their efficiencies.

3 **Shortest Path Algorithms**

Explore graph traversal for optimal route finding.

Presentation Outline

1

Stack ADT

LIFO principle, push/pop operations, real-world applications.

2

Sorting Algorithms

Bubble, insertion, merge, quick sort techniques.

3

Shortest Path Algorithms

Dijkstra's and A* algorithms for efficient pathfinding.

Introduction to DSA and ADT

Data Structures

Organizing data for efficient access and modification.

Algorithms

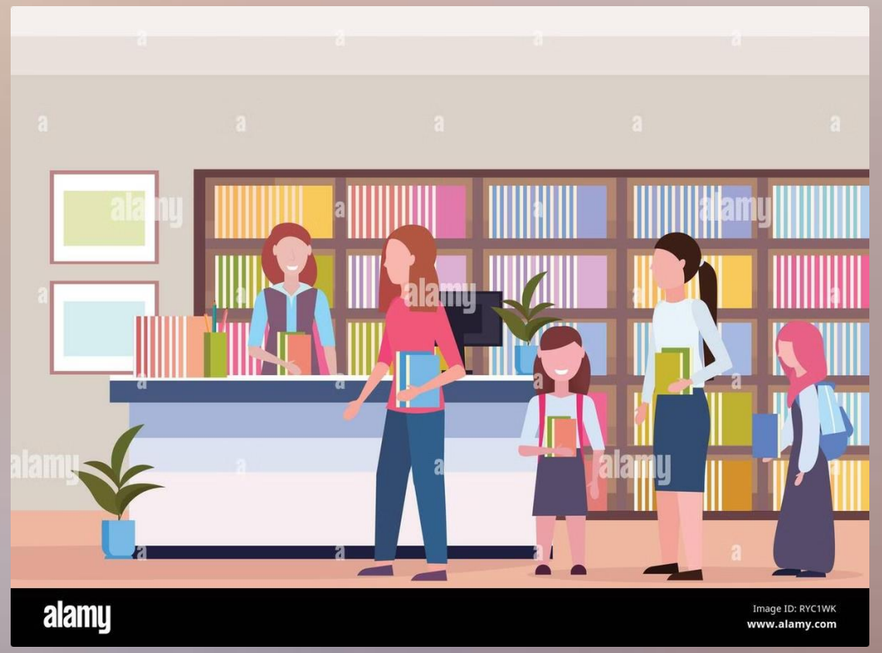
Step-by-step procedures for solving computational problems.

Abstract Data Types

High-level description of operations on data.

Stack ADT & Queue: Fundamental Data Structures

Essential concepts for efficient data management and algorithms.



Stack ADT: Definition

Abstract Data Type

Conceptual model for organizing and accessing data.

Linear Structure

Elements arranged in a sequential order.

Restricted Access

Only top element accessible for operations.

Stack: LIFO Principle

1

Last In

Most recently added element at the top.

2

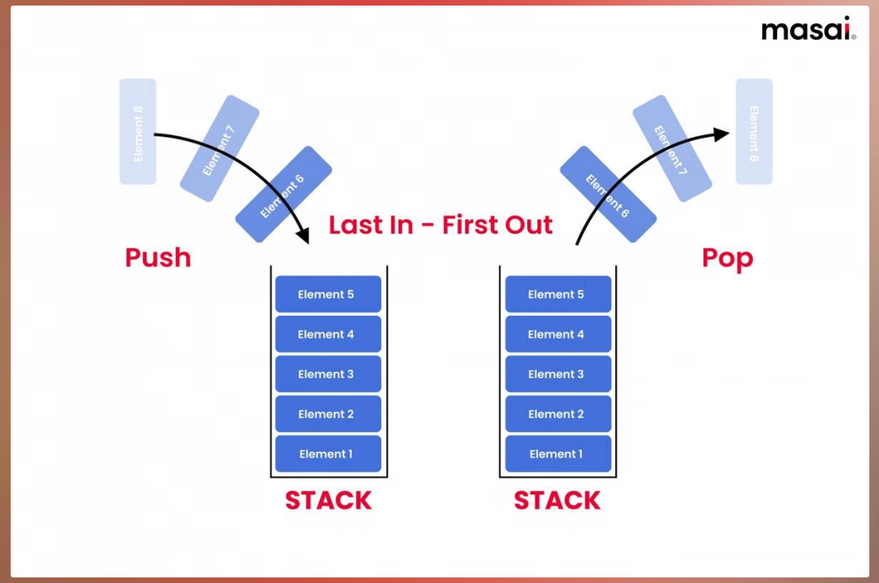
First Out

Top element removed first when accessing.

3

Order Preservation

Maintains reverse order of insertion.



Stack Operations

Push

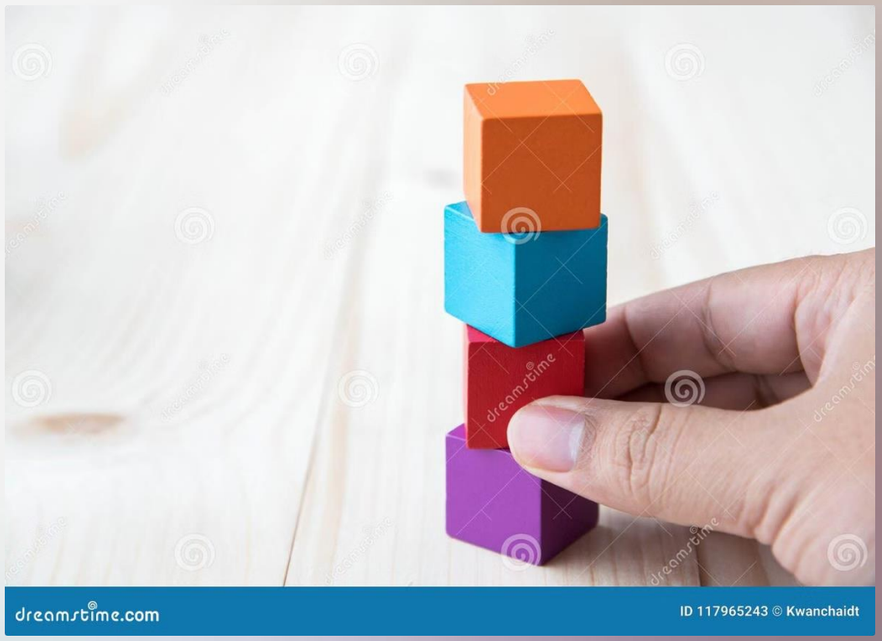
Add element to top of stack.

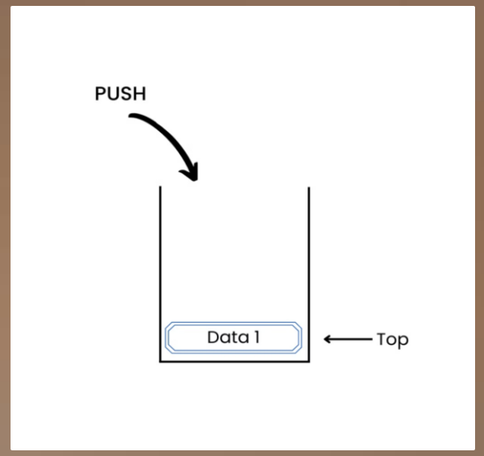
Pop

Remove and return top element.

Peek

View top element without removing.





Push Operation in Stack

1

Precondition

Stack not full, element to add available.

2

Operation

Add new element to top of stack.

3

Postcondition

New element becomes accessible at top.

Pop Operation in Stack

1

Precondition

Stack not empty.

2

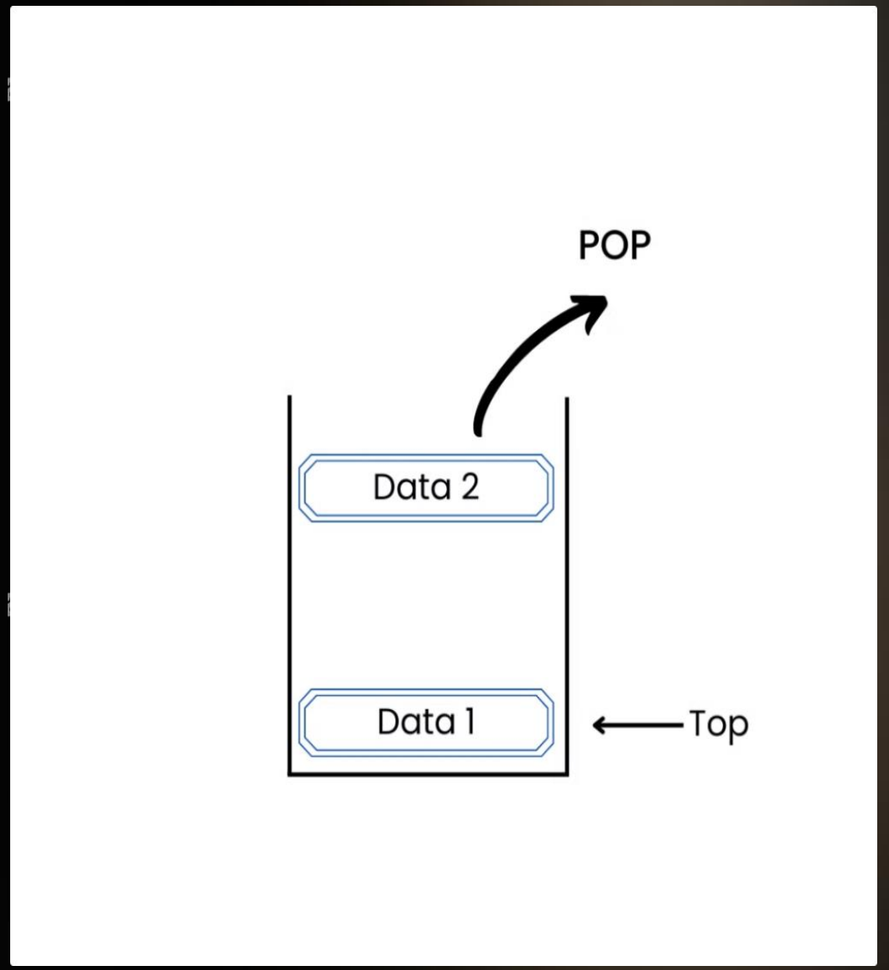
Operation

Remove top element from stack.

3

Postcondition

Top element removed, next element becomes accessible.



Peek Operation

Purpose

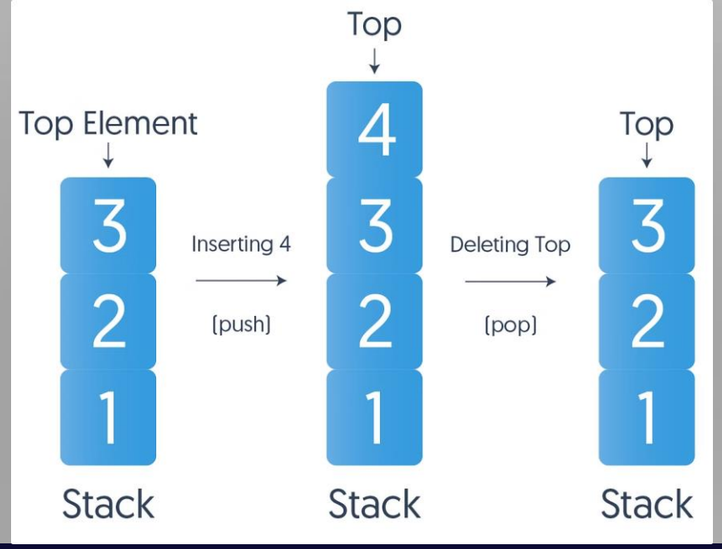
View top element without removing it.

Specification

Returns top element, stack remains unchanged.

Usage

Useful for checking conditions before pop.



isEmpty Operation

Purpose

Check if stack contains any elements.

Return

Boolean value: true if empty, false otherwise.

Usage

Prevent underflow errors in pop operations.

isFull Operation

Purpose

Check if stack has reached maximum capacity.

Return

Boolean value: true if full, false otherwise.

Usage

Prevent overflow errors in push operations.

Queue: Definition

Linear Structure

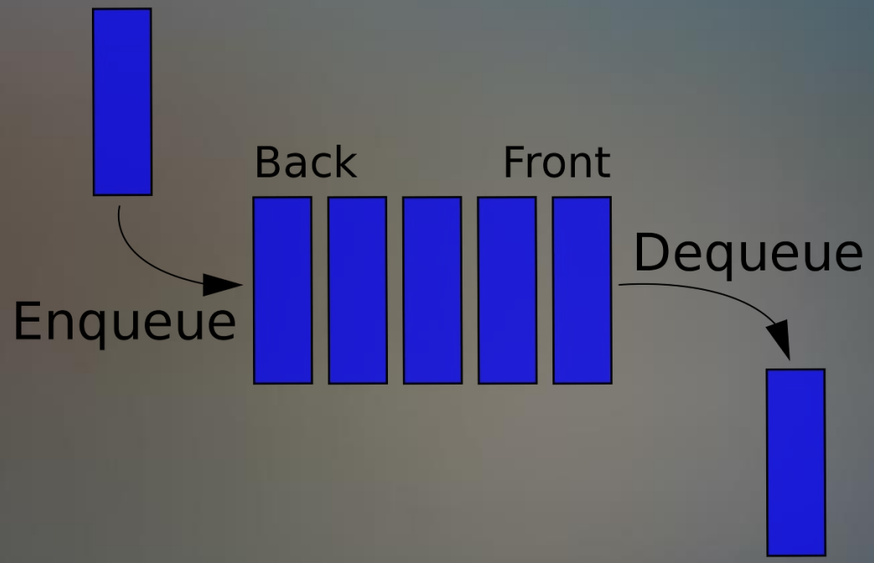
Elements arranged in sequential order.

Two-Ended Access

Addition at rear, removal from front.

Ordered Operations

Maintains insertion order for processing.



Queue: FIFO Principle

1

First In

Earliest added element at front.

2

First Out

Front element removed first when accessing.

3

Order Preservation

Maintains original order of insertion.

Queue Operations

Enqueue

Add element to rear of queue.

Dequeue

Remove and return front element.

Front

View front element without removing.

Queue ADT Overview

1

First In

Elements enter at rear end.

2

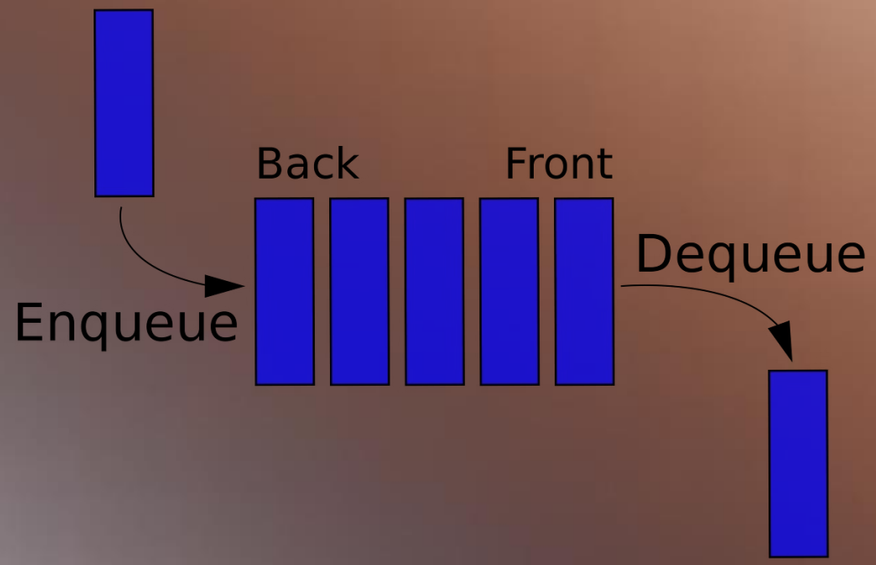
Wait

Elements remain in queue order.

3

First Out

Elements exit from front end.



Enqueue Operation

Purpose

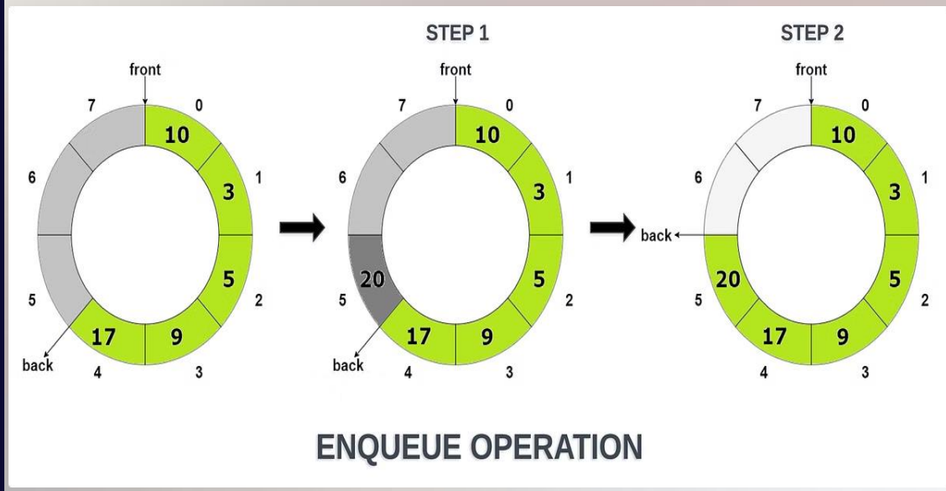
Add new element to rear of queue.

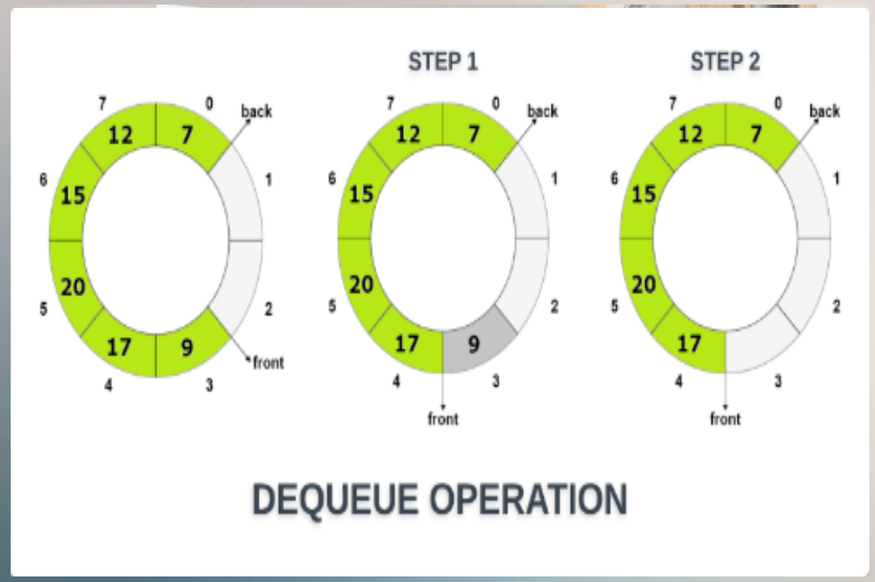
Precondition

Queue not full.

Postcondition

New element added, rear pointer updated.





Dequeue Operation

Purpose

Remove element from front of queue.

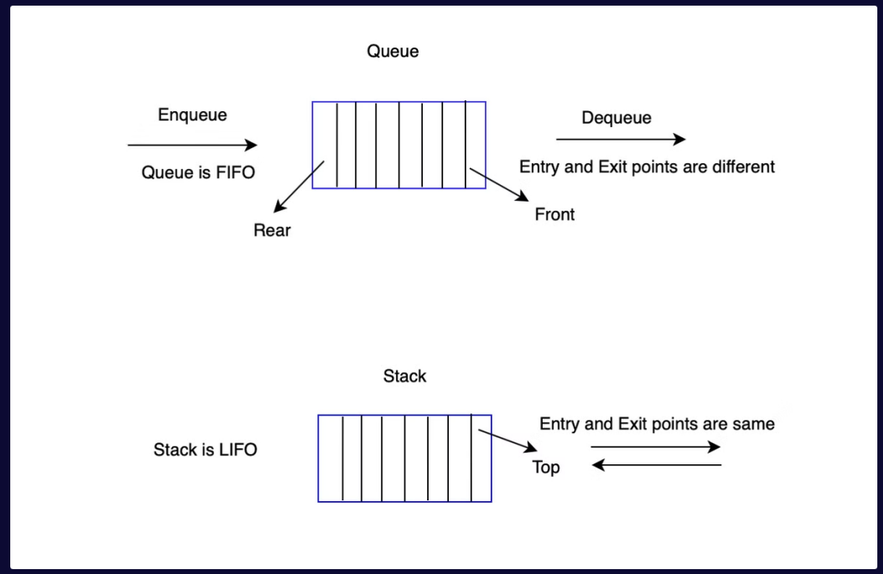
Precondition

Queue not empty.

Postcondition

Front element removed, front pointer updated.

Peek Operation



Definition

View front element without removing it.

Usage

Check next element to be dequeued.

Implementation

Return value at front pointer.

isEmpty Operation

Check

Determine if queue contains no elements.

Efficiency

Constant time complexity O(1).

Implementation

Compare front and rear pointers.

isFull Operation

Purpose

Check if queue reached maximum capacity.

Importance

Prevent overflow in fixed-size implementations.

Implementation

Compare element count to size limit.

Stack vs Queue: Comparison

Stack (LIFO)

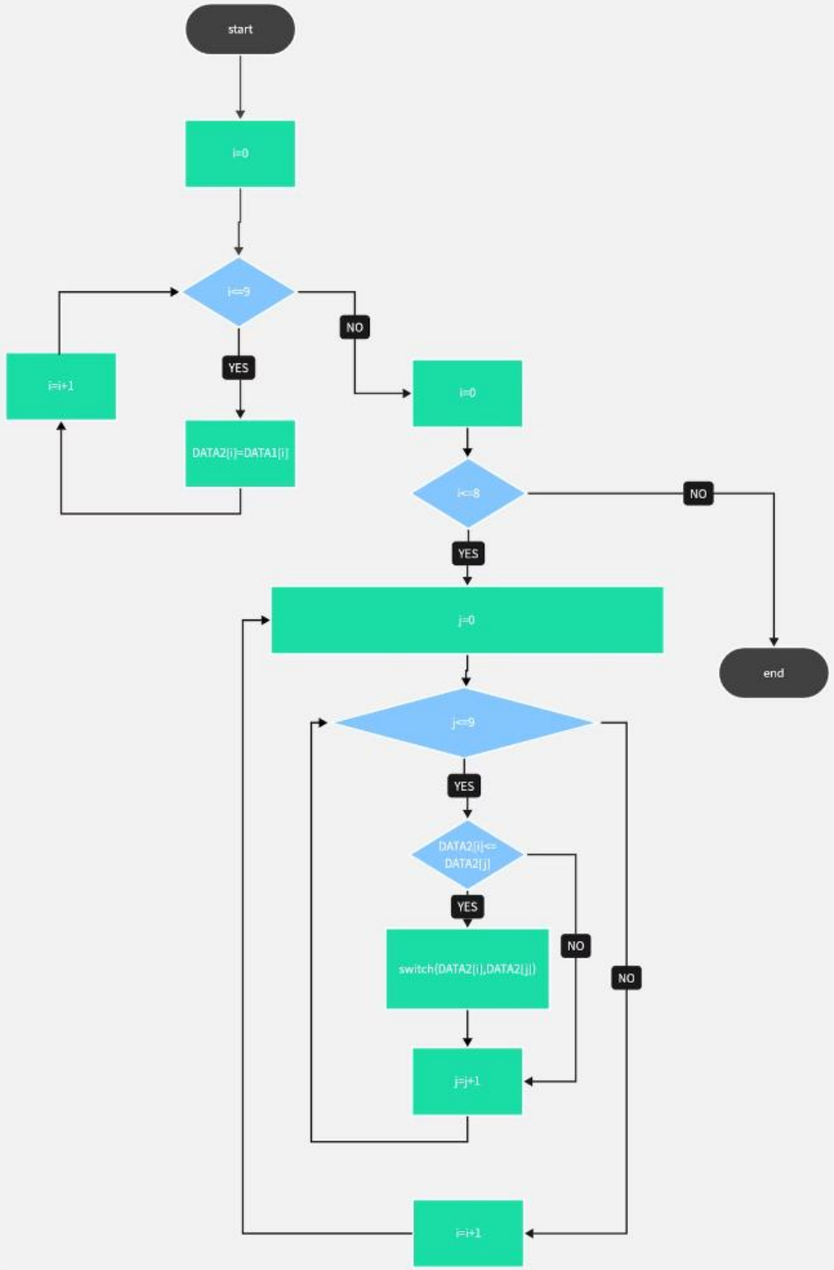
Last added, first removed.

Single-ended structure.

Queue (FIFO)

First added, first removed.

Two-ended structure.



Applications in Computer Science

1 Stack Uses

Function calls, undo mechanisms, expression evaluation.

2 Queue Uses

Task scheduling, breadth-first search, print spooling.

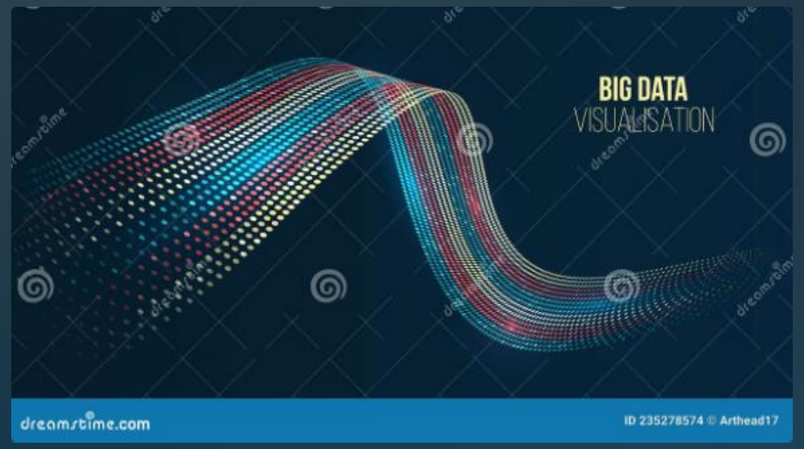
3 Efficiency

O(1) time complexity for basic operations.

Sorting Algorithms: Organizing Data Efficiently

Fundamental techniques for arranging elements in specific orders.

Essential for optimizing search and data processing operations.



Introduction to Sorting Algorithms

1 **Definition**

Methods to arrange data in specific order.

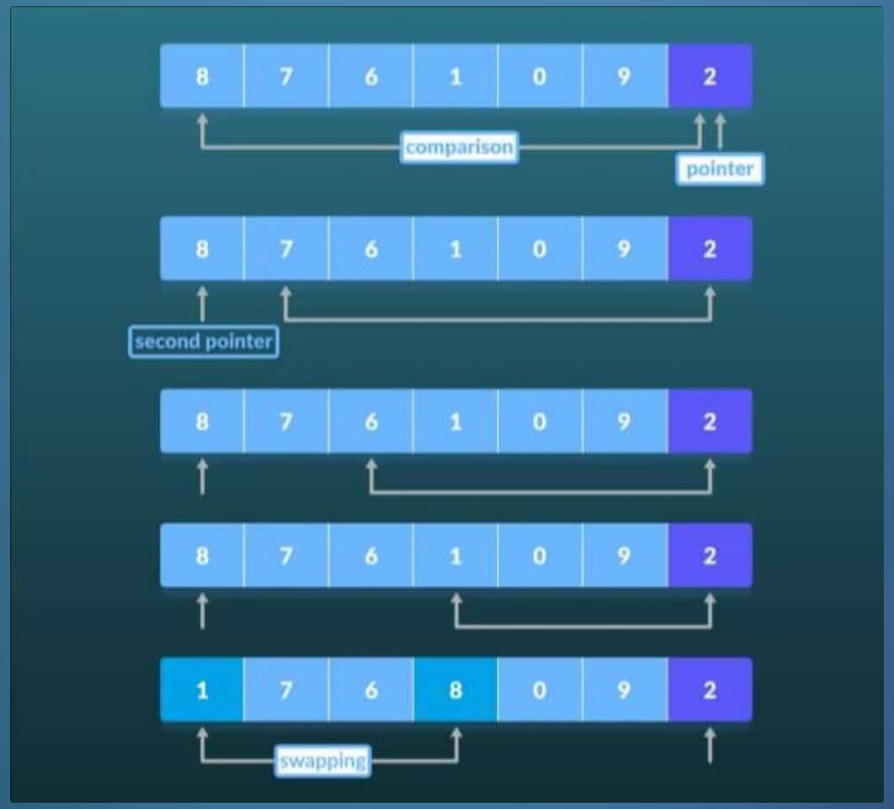
2 **Importance**

Optimize search, enable efficient data processing.

3 **Applications**

Databases, analytics, computer graphics, AI.

Quick Sort: Divide and Conquer



1. **Choose Pivot**

   Select element as partition reference point.

2. **Partitioning**

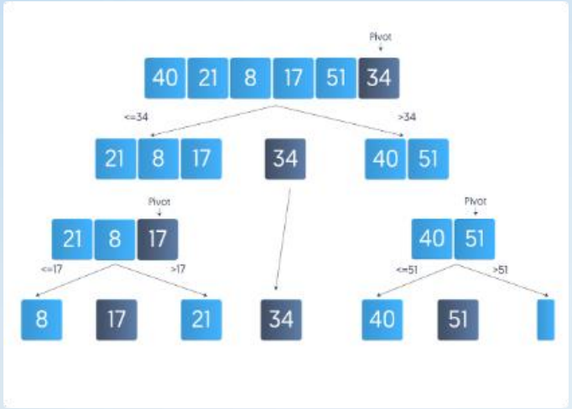
   Rearrange elements around pivot.

3. **Recursive Sort**

   Apply process to sub-arrays.

4. **Combine**

   Merge sorted sub-arrays for final result.



Quick Sort Process

1

2

3

4

Initial Array

Unsorted list of elements.

Partitioning

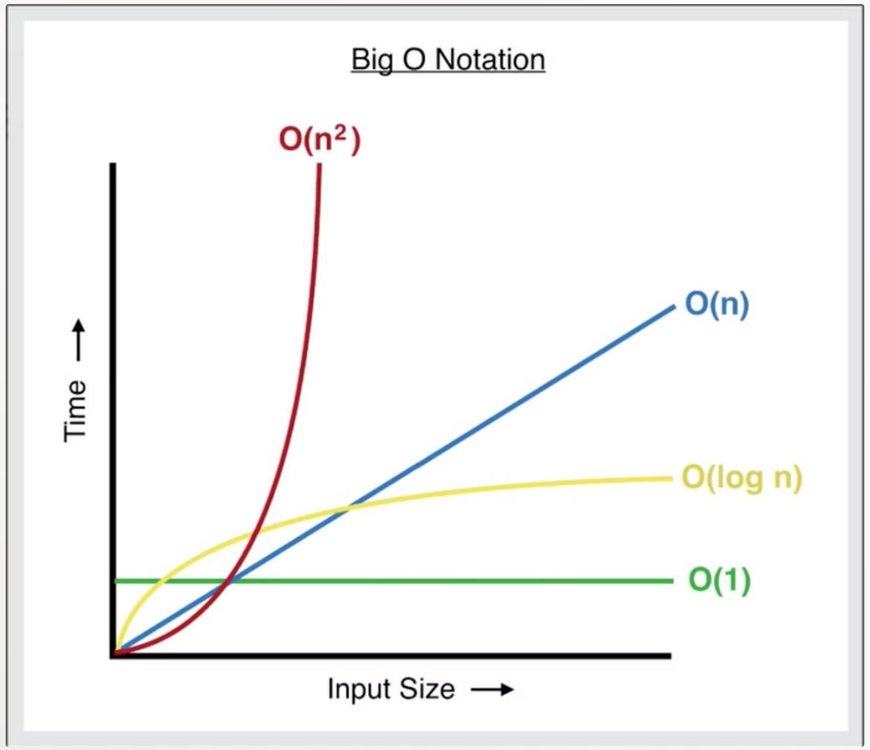
Divide array around pivot.

Recursive Sorting

Sort subarrays independently.

Combine

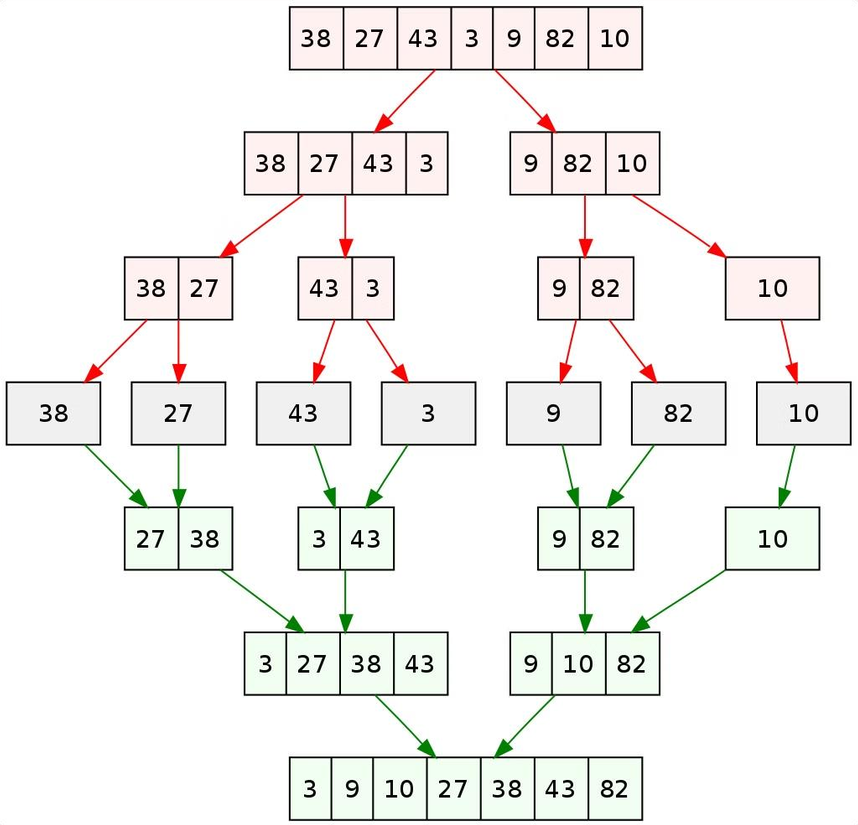
Merge sorted subarrays.



Quick Sort Complexity Analysis

Best Case	Average Case	Worst Case
O(n log n)	O(n log n)	$O(n^2)$

Efficient for random data, vulnerable to poor pivots.



Merge Sort: Divide, Sort, and Combine

1

Divide

Split array into smaller subarrays.

2

Conquer

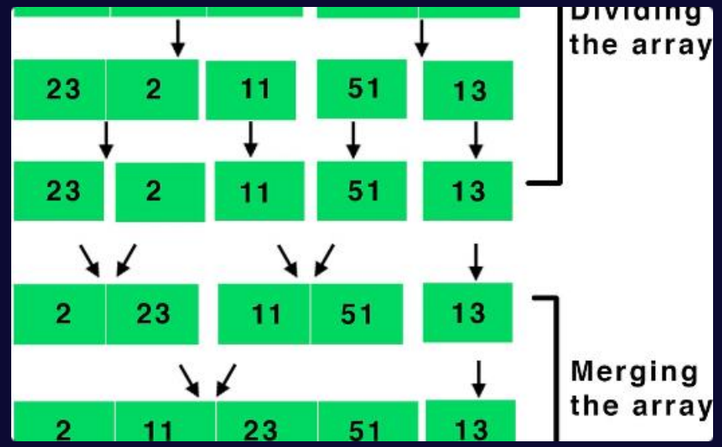
Sort subarrays recursively.

3

Combine

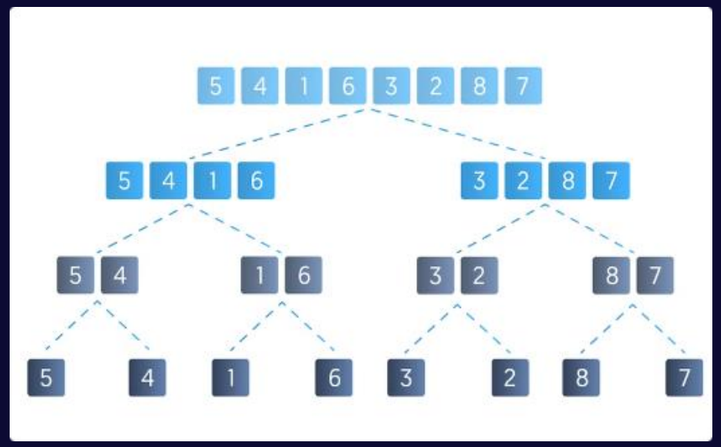
Merge sorted subarrays to produce result.

Merge Sort Process Illustration



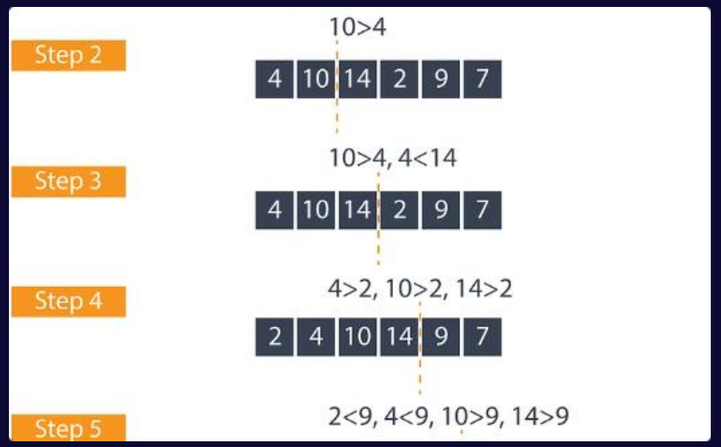
Divide

Split array into smaller subarrays.



Sort

Recursively sort individual subarrays.



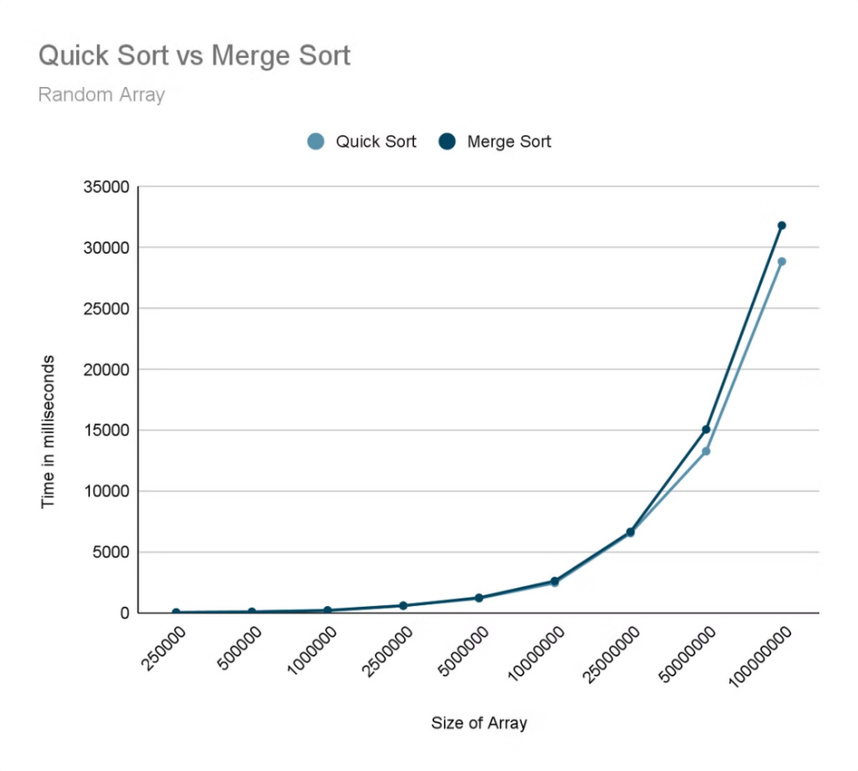
Merge

Combine sorted subarrays into final result.

Merge Sort Complexity Analysis

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

Consistent performance across all input distributions.



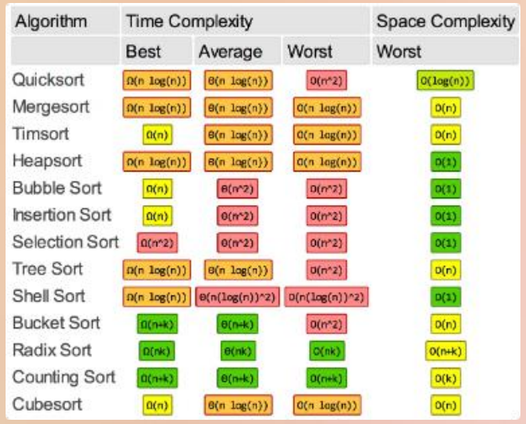
Quick Sort vs. Merge Sort Comparison

Quick Sort

- In-place sorting
- Cache-friendly
- Faster on average

Merge Sort

- Stable sorting
- Predictable performance
- Parallelizable



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Use Cases and Scenarios

Quick Sort

Arrays with random access, limited memory.

Merge Sort

Stable sorting required, linked lists.

Hybrid Approaches

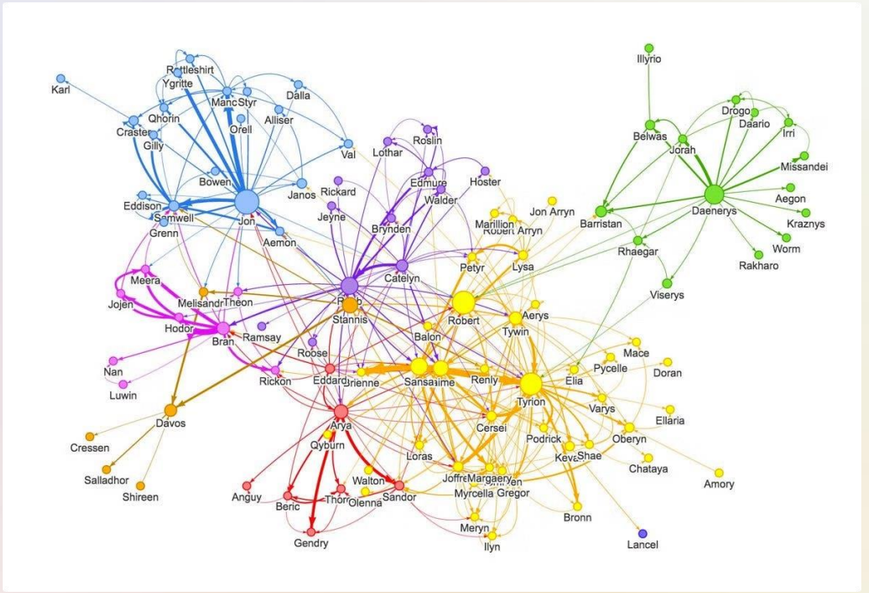
Combine algorithms for optimal performance.

External Sorting

Large datasets exceeding main memory.

Shortest Path Algorithms: Dijkstra's vs Prim-Jarnik

Explore the world of graph algorithms, focusing on Dijkstra's and Prim-Jarnik. These powerful tools solve critical network problems, finding shortest paths and minimum spanning trees.



Importance of Shortest Path and Minimum Spanning Tree Algorithms

1 **Efficient Routing**

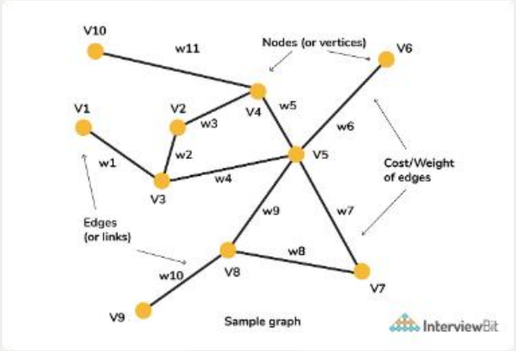
Shortest path algorithms optimize routes in transportation and communication networks, reducing costs and time.

2 **Network Design**

Minimum spanning trees help design efficient networks with minimal total edge weight.

3 **Resource Allocation**

These algorithms assist in optimizing resource distribution across networks, improving overall efficiency.



Introduction to Dijkstra's Algorithm

1

Origin

Developed by Dutch computer scientist Edsger W. Dijkstra in 1956.

2

Purpose

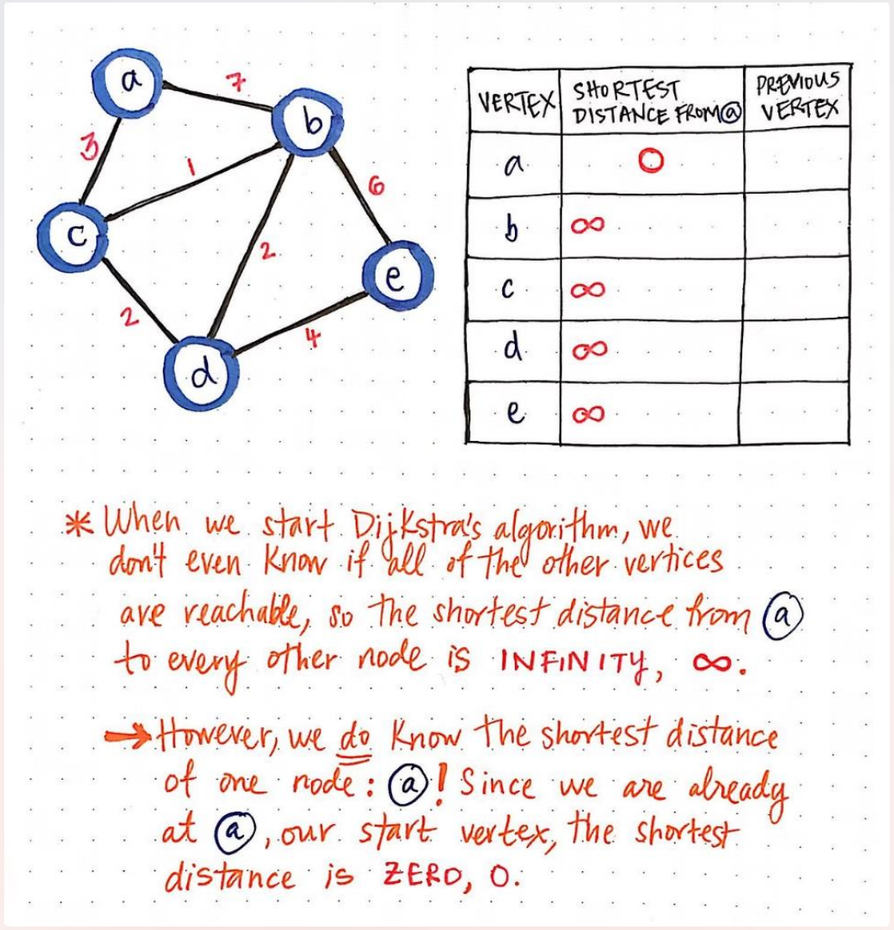
Finds the shortest path between nodes in a graph with non-negative edge weights.

3

Approach

Uses a greedy strategy to iteratively select the unvisited node with the smallest distance.

Detailed Explanation of Dijkstra's Algorithm



1

Initialization

Set distances to infinity for all nodes except the starting node (set to 0).

2

Selection

Choose the unvisited node with the smallest tentative distance from the start node.

3

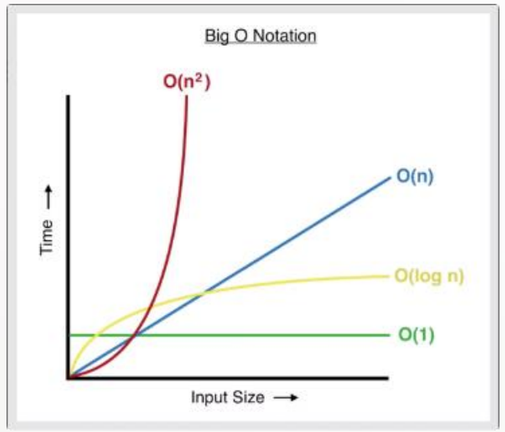
Relaxation

Update distances to neighboring nodes if a shorter path is found.

4

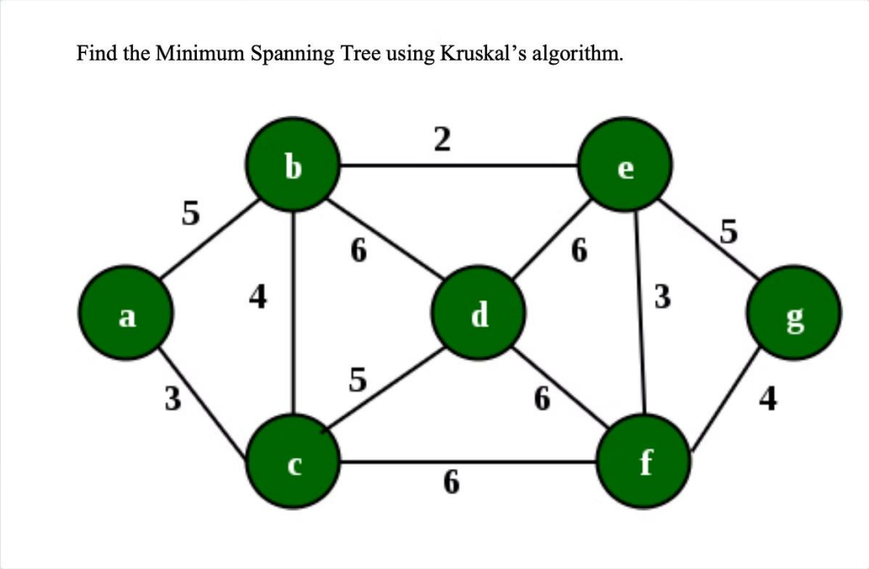
Completion

Repeat until all nodes are visited or the destination node is reached.



Complexity of Dijkstra's Algorithm

Implementation	Time Complexity	Space Complexity
Basic (array)	O(V^2)	O(V)
Min-Heap	O(E + V log V)	O(V)
Fibonacci Heap	O(E + V log V)	O(V)



Introduction to Prim-Jarnik Algorithm

1

Origin

Developed independently by Jarník (1930) and Prim (1957), later rediscovered by Dijkstra (1959).

2

Purpose

Finds a minimum spanning tree for a weighted, undirected graph.

3

Approach

Grows the tree one edge at a time, always adding the lowest-weight edge.

Detailed Explanation of Prim-Jarnik Algorithm

1

Initialization

Start with an arbitrary vertex as a single-vertex tree.

2

Selection

Choose the minimum-weight edge connecting the tree to a non-tree vertex.

3

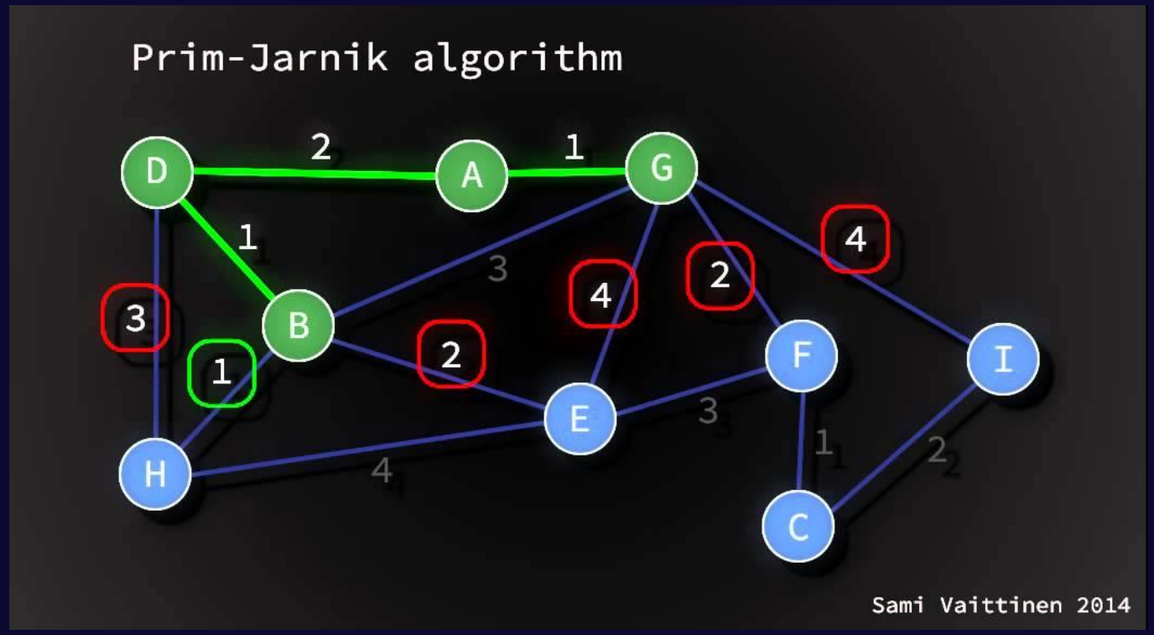
Addition

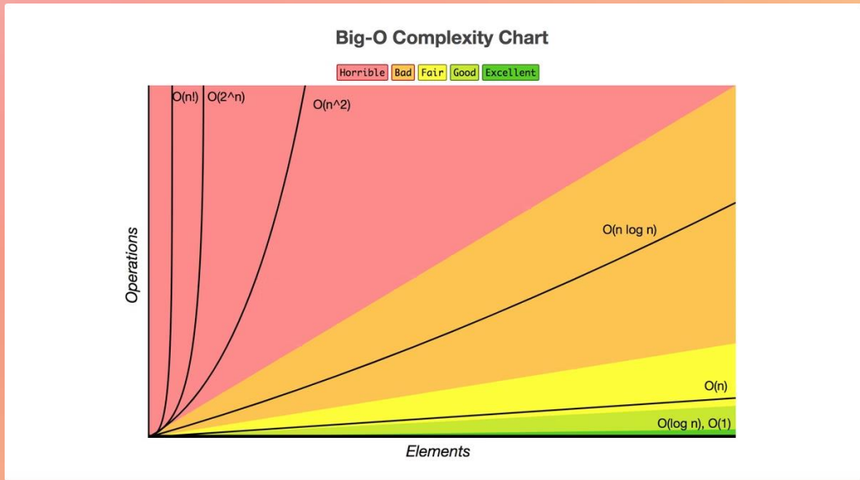
Add the selected edge and vertex to the tree.

4

Completion

Repeat until all vertices are in the tree.





Complexity of Prim-Jarnik Algorithm

Implementation	Time Complexity	Space Complexity
Basic (array)	O(V^2)	O(V)
Binary Heap	O(E log V)	O(V)
Fibonacci Heap	O(E + V log V)	O(V)

Comparison of Dijkstra's and Prim-Jarnik Algorithms

Dijkstra's Algorithm

Finds shortest paths from a single source to all vertices. Works with non-negative edge weights.

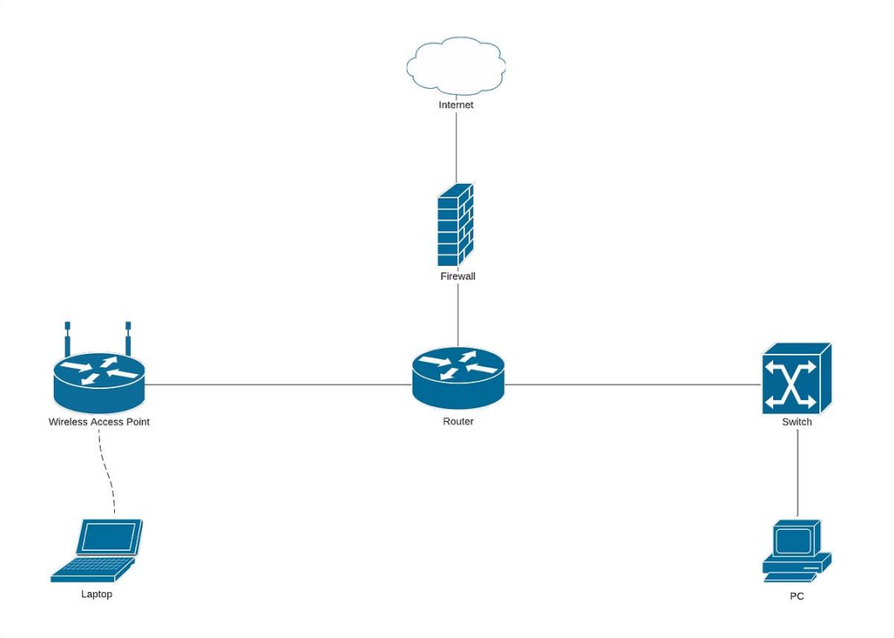
Prim-Jarnik Algorithm

Finds minimum spanning tree for undirected graphs. Works with any edge weights, including negative.

Similarities

Both use greedy approach. Similar time complexities. Can be implemented using priority queues.

Use Cases: Shortest Path vs. Minimum Spanning Tree



Navigation

Dijkstra's algorithm optimizes routes in GPS systems and traffic management.

Network Design

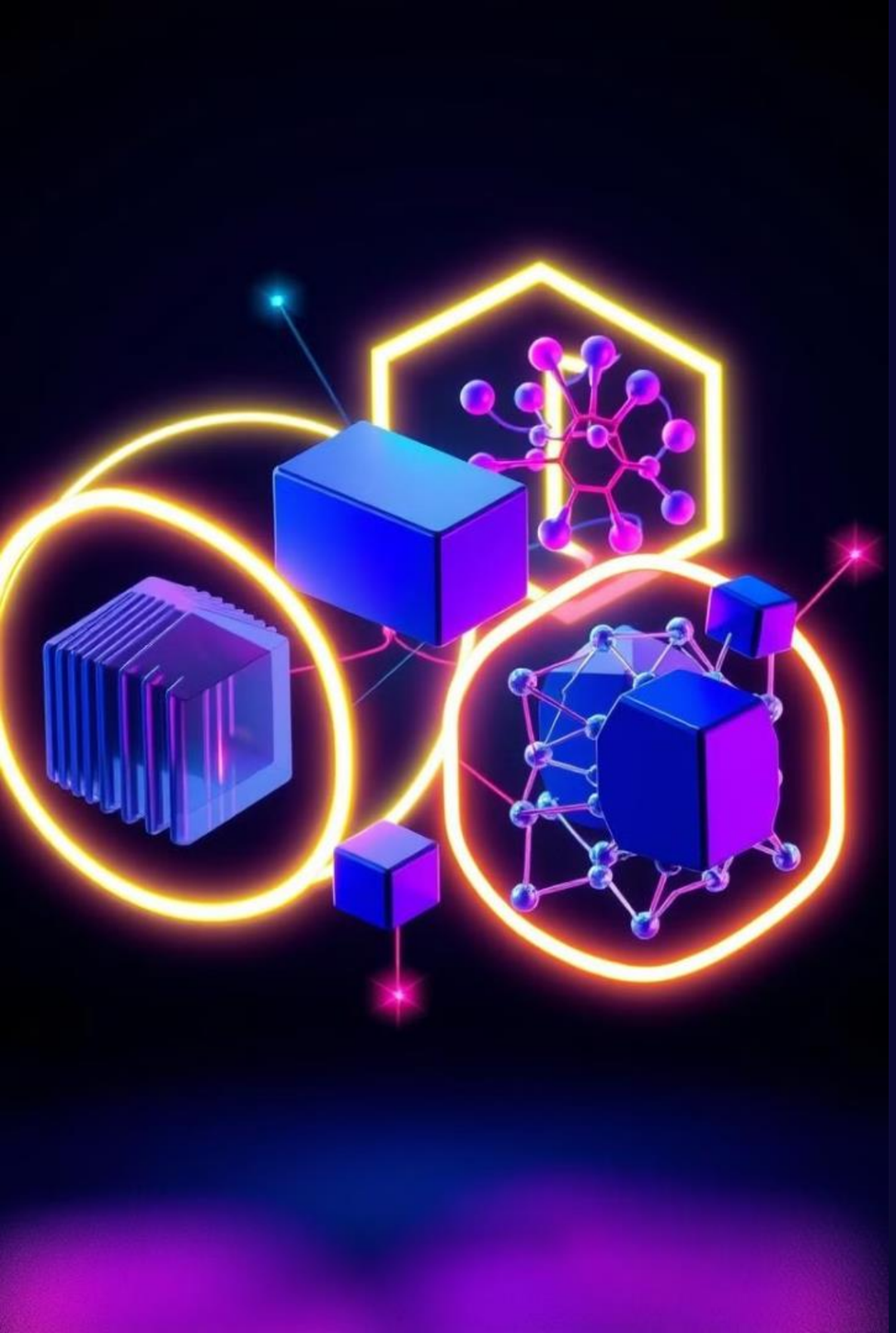
Prim-Jarnik algorithm helps design efficient computer networks and power grids.

Robotics

Dijkstra's algorithm aids in path planning for autonomous robots and vehicles.

Infrastructure

Prim-Jarnik algorithm optimizes pipeline and transportation infrastructure layouts.



Conclusion: Data Structures and Algorithms

Wrapping up our journey through fundamental CS concepts.

Summary of Key Concepts

Stack and Queue

LIFO vs FIFO data structures.

Sorting Algorithms

Methods to organize data efficiently.

Shortest Path Algorithms

Finding optimal routes in graphs.

Real-world Applications

Web Browsers

Stack for back/forward navigation.

Printers

Queue for managing print jobs.

Databases

Sorting for efficient data retrieval.

GPS Navigation

Pathfinding for route optimization.

Thank You