8 Common Data Structures Every Programmer Must Know

Suneel Kumar · Follow

12 min read · Aug 8, 2024

In the vast realm of programming, data structures serve as the fundamental building blocks that allow us to organize, store, and manipulate data efficiently. Whether you're a beginner coder or a seasoned developer, having a solid grasp of common data structures is crucial for writing efficient and optimized code. In this article, we'll explore eight essential data structures that every programmer should be familiar with, providing clear explanations and relatable examples to help you understand their importance and applications.

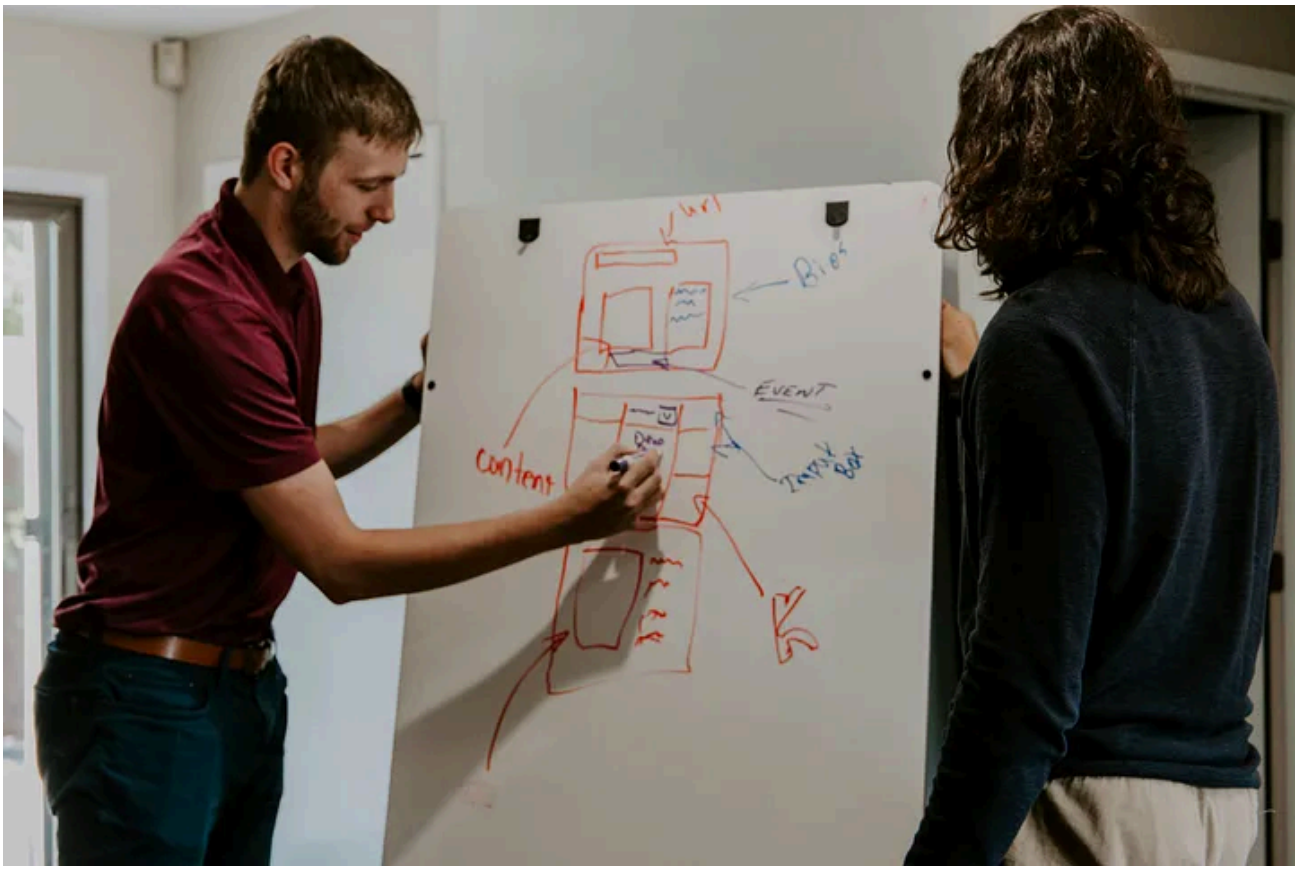
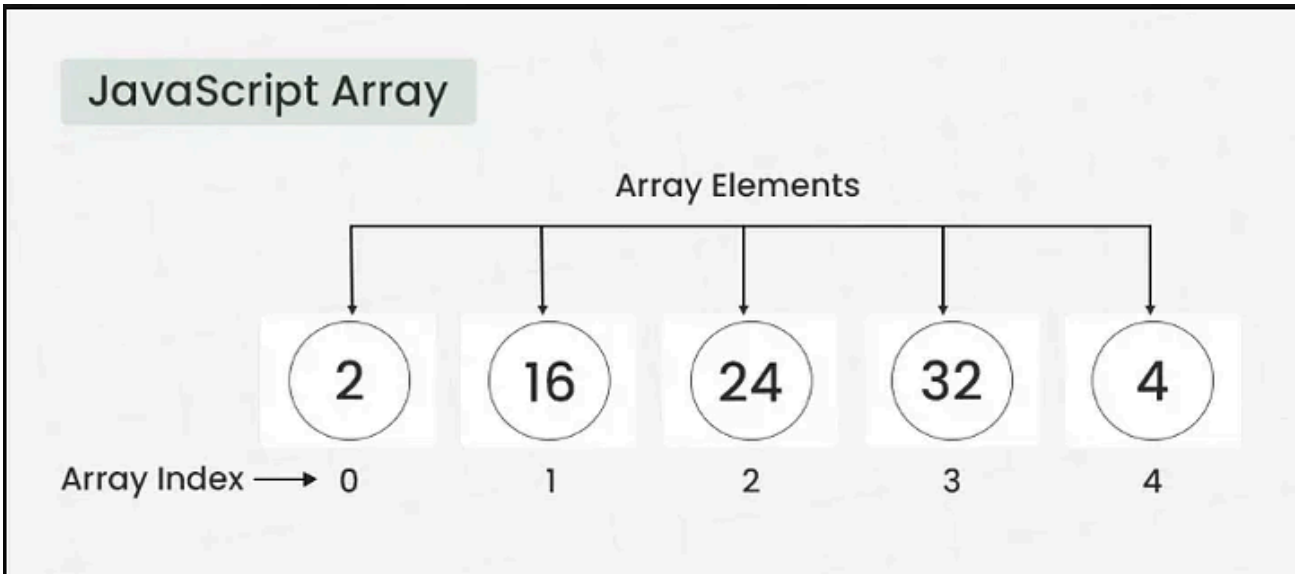


Photo by Kaleidico on Unsplash

1. Arrays: The Versatile Workhorses

What are Arrays?

Arrays are perhaps the most basic and widely used data structure in programming. Think of an array as a collection of items stored at contiguous memory locations. It's like a row of lockers in a school, where each locker (element) is numbered sequentially and can hold an item.



array

How do Arrays work?

Arrays work on the principle of index-based access. Each element in an array is associated with an index, typically starting from 0. This allows for quick and direct access to any element in the array.

Example: The Bookshelf Analogy

Imagine you have a bookshelf with 5 slots, numbered 0 to 4. Each slot can hold one book. This is analogous to an array of size 5.

```
# Creating an array (bookshelf) in Python
bookshelf = ["Harry Potter", "Lord of the Rings", "Pride and Prejudice", "1984",

# Accessing elements
print(bookshelf[0])  # Output: Harry Potter
print(bookshelf[2])  # Output: Pride and Prejudice

# Modifying an element
bookshelf[1] = "The Hobbit"
print(bookshelf)  # Output: ['Harry Potter', 'The Hobbit', 'Pride and Prejudice'
```

Advantages of Arrays

1. **Fast access**: Elements can be accessed instantly using their index.
2. **Space efficiency**: Arrays use a contiguous block of memory, making them memory-efficient.
3. **Simplicity**: They are easy to understand and use.

Limitations of Arrays

1. **Fixed size**: In many languages, arrays have a fixed size that can't be changed after creation.
2. **Insertion and deletion**: These operations can be costly, especially for large arrays.

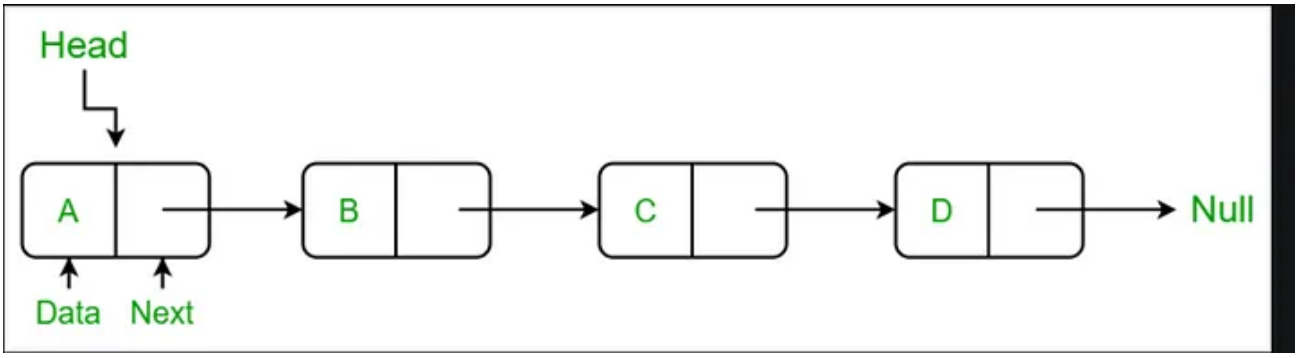
Real-world Applications

- Storing and manipulating image pixel data
- Implementing matrices for scientific computations
- Managing lists of items in user interfaces

2. Linked Lists: The Flexible Chains

What are Linked Lists?

A linked list is a linear data structure where elements are stored in nodes. Each node contains a data field and a reference (or link) to the next node in the sequence. Unlike arrays, linked lists don't store elements in contiguous memory locations.



Linked Lists

How do Linked Lists work?

Linked lists work by connecting nodes through pointers. Each node knows about the next node in the sequence, forming a chain-like structure. This allows for efficient insertion and deletion of elements at any point in the list.

Example: The Train Analogy

Think of a linked list as a train. Each train car (node) carries some cargo (data) and is connected to the next car. You can easily add or remove cars from any point in the train.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
```

```
        def display(self):
            current = self.head
            while current:
                print(current.data, end=" -> ")
                current = current.next
            print("None")

    # Creating a linked list
    train = LinkedList()
    train.append("Engine")
    train.append("Passenger Car")
    train.append("Dining Car")
    train.append("Cargo Car")

    train.display()  # Output: Engine -> Passenger Car -> Dining Car -> Cargo Car ->
```

Advantages of Linked Lists

1. **Dynamic size**: Linked lists can grow or shrink in size during execution.
2. **Efficient insertion and deletion**: Adding or removing elements is fast, especially at the beginning of the list.
3. **Flexible memory allocation**: Nodes can be stored anywhere in memory.

Limitations of Linked Lists

1. **Sequential access**: To reach the nth element, you need to traverse from the beginning.
2. **Extra memory**: Each node requires extra memory for storing the reference to the next node.

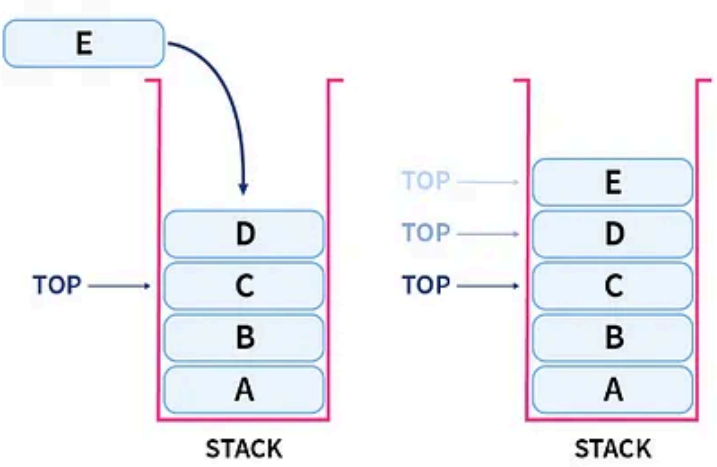
Real-world Applications

- Implementing undo functionality in applications
- Managing music playlists (where songs can be easily added or removed)
- Implementing hash tables for collision resolution

3. Stacks: The Last-In-First-Out Champions

What are Stacks?

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. Think of it as a stack of plates: you can only add or remove plates from the top.



Stacks

How do Stacks work?

Stacks operate with two primary operations:

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove the top element from the stack.

Example: The Browser History Analogy

Your web browser's back button functionality is a perfect real-world example of a stack. As you visit new pages, they're pushed onto the stack. When you hit the back button, you're popping pages off the stack.

```
class BrowserHistory:
    def __init__(self):
        self.history = []

    def visit(self, url):
        self.history.append(url)
        print(f"Visited: {url}")

    def back(self):
        if len(self.history) > 1:
            self.history.pop()
            print(f"Went back to: {self.history[-1]}")
        else:
            print("Can't go back further!")

# Using our browser history stack
browser = BrowserHistory()
browser.visit("google.com")
browser.visit("youtube.com")
browser.visit("github.com")
browser.back()
browser.back()
browser.back()
browser.back()

# Output:
# Visited: google.com
# Visited: youtube.com
# Visited: github.com
# Went back to: youtube.com
# Went back to: google.com
# Can't go back further!
```

Advantages of Stacks

1. **Simple and efficient**: Stack operations are straightforward and fast.
2. **Memory management**: Useful for managing function calls and recursion.
3. **Undo mechanisms**: Easily implement undo functionality in applications.

Limitations of Stacks

1. **Limited access**: Only the top element is accessible at any time.

2. **Fixed size** (in some implementations): May have a maximum size limit.

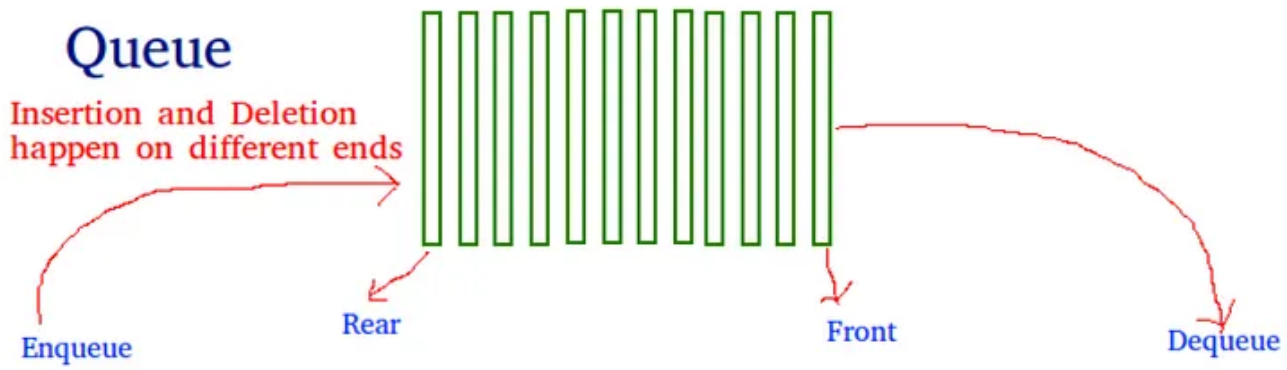
Real-world Applications

- Function call management in programming languages
- Expression evaluation and syntax parsing
- Undo-redo features in text editors

4. Queues: The First-In-First-Out Organizers

What are Queues?

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It's like a line of people waiting for a bus: the first person to join the line is the first to board the bus.



Queues

How do Queues work?

Queues operate with two main operations:

- **Enqueue**: Add an element to the back of the queue.

- **Dequeue:** Remove the front element from the queue.

Example: The Printing Queue Analogy

A printer queue is a classic example of a queue in action. Print jobs are processed in the order they are received.

```
from collections import deque

class PrinterQueue:
    def __init__(self):
        self.queue = deque()

    def add_job(self, document):
        self.queue.append(document)
        print(f"Added '{document}' to the print queue")

    def print_job(self):
        if self.queue:
            document = self.queue.popleft()
            print(f"Printing: {document}")
        else:
            print("No jobs in the queue")

    def display_queue(self):
        print("Current queue:", list(self.queue))

# Using our printer queue
printer = PrinterQueue()
printer.add_job("Annual Report")
printer.add_job("Meeting Minutes")
printer.add_job("Employee Handbook")
printer.display_queue()
printer.print_job()
printer.print_job()
printer.display_queue()

# Output:
# Added 'Annual Report' to the print queue
# Added 'Meeting Minutes' to the print queue
# Added 'Employee Handbook' to the print queue
# Current queue: ['Annual Report', 'Meeting Minutes', 'Employee Handbook']
# Printing: Annual Report
```

```
# Printing: Meeting Minutes
# Current queue: ['Employee Handbook']
```

Advantages of Queues

1. **Fairness**: Ensures first-come, first-served processing.
2. **Predictable**: Elements are processed in a known order.
3. **Decoupling**: Useful for managing asynchronous data transfer between processes.

Limitations of Queues

1. **Limited access**: Only front and rear elements are easily accessible.
2. **Potential for bottlenecks**: If enqueue operations are faster than dequeue operations, the queue can grow indefinitely.

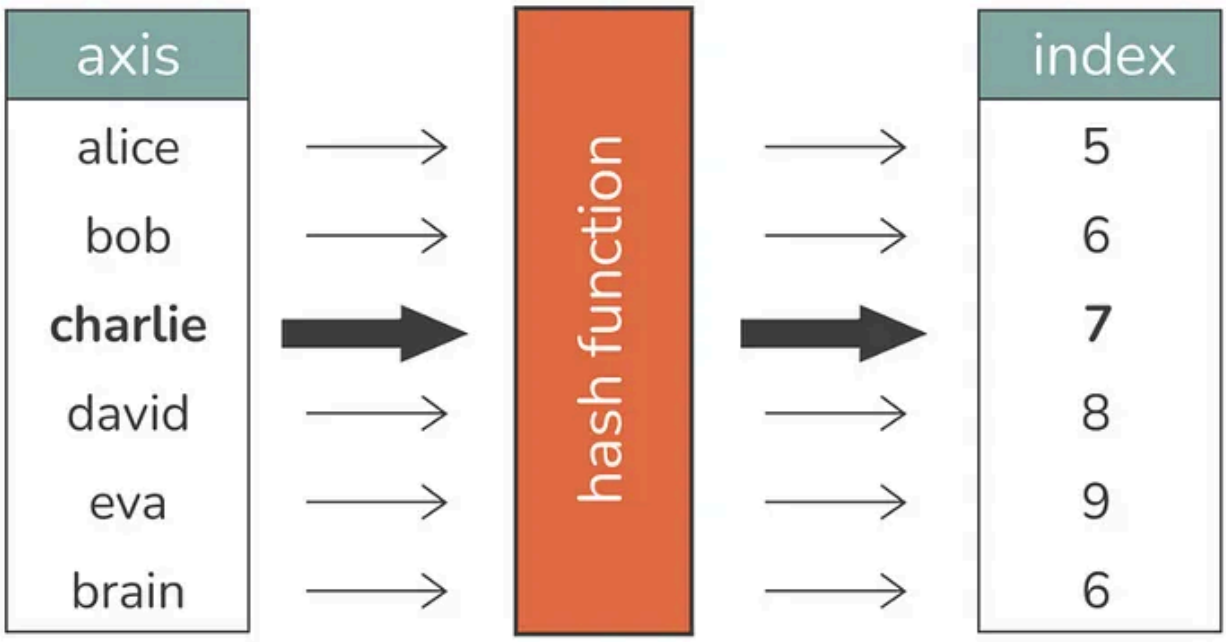
Real-world Applications

- Task scheduling in operating systems
- Handling requests in web servers
- Breadth-First Search algorithm in graph traversal

5. Hash Tables: The Lightning-Fast Lookup Masters

What are Hash Tables?

Hash tables, also known as hash maps, are data structures that store key-value pairs and provide rapid data retrieval. They use a hash function to compute an index into an array of buckets, from which the desired value can be found.



Hash Tables

How do Hash Tables work?

1. A hash function takes the key as input and produces an index.
2. The key-value pair is stored in the bucket corresponding to this index.
3. To retrieve a value, the key is hashed again to find the correct bucket.

Example: The Library Catalog Analogy

Imagine a library where books (values) are stored in shelves (buckets) based on a unique code (key) derived from the book's title. This code is generated by a special formula (hash function).

```
class SimpleHashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash(self, key):
        return sum(ord(char) for char in key) % self.size
```

```
    def insert(self, key, value):
        index = self._hash(key)
        for item in self.table[index]:
            if item[0] == key:
                item[1] = value
                return
        self.table[index].append([key, value])

    def get(self, key):
        index = self._hash(key)
        for item in self.table[index]:
            if item[0] == key:
                return item[1]
        raise KeyError(key)

    def display(self):
        for i, bucket in enumerate(self.table):
            print(f"Bucket {i}: {bucket}")

# Using our simple hash table
library = SimpleHashTable(10)
library.insert("Moby Dick", "Shelf A")
library.insert("Pride and Prejudice", "Shelf B")
library.insert("The Great Gatsby", "Shelf C")
library.insert("To Kill a Mockingbird", "Shelf D")

library.display()
print("Location of 'The Great Gatsby':", library.get("The Great Gatsby"))

# Output might look like:
# Bucket 0: []
# Bucket 1: []
# Bucket 2: [['Moby Dick', 'Shelf A']]
# Bucket 3: []
# Bucket 4: [['Pride and Prejudice', 'Shelf B']]
# Bucket 5: [['The Great Gatsby', 'Shelf C']]
# Bucket 6: []
# Bucket 7: [['To Kill a Mockingbird', 'Shelf D']]
# Bucket 8: []
# Bucket 9: []
# Location of 'The Great Gatsby': Shelf C
```

Advantages of Hash Tables

1. **Fast lookup**: Average case O(1) time complexity for insertions, deletions, and searches.
2. **Flexible keys**: Can use various data types as keys, not just integers.

3. **Space efficiency**: Can represent sparse data efficiently.

Limitations of Hash Tables

1. **Collisions**: Different keys might hash to the same index, requiring collision resolution.
2. **Unordered**: Do not maintain the order of insertion.
3. **Resizing**: May need to be resized as they grow, which can be costly.

Real-world Applications

- Implementing dictionaries in programming languages
- Database indexing for faster queries
- Caching mechanisms in web applications

6. Trees: The Hierarchical Organizers

What are Trees?

Trees are hierarchical data structures consisting of nodes connected by edges. They start with a root node and branch out to child nodes, forming a structure similar to an inverted tree.

How do Trees work?

Trees organize data in a parent-child relationship. Each node can have multiple children but only one parent (except the root). This structure allows for efficient searching and organization of hierarchical data.

Example: The Family Tree Analogy

A family tree is a perfect real-world example of a tree data structure. Each person is a node, with parents above and children below.

```
class FamilyMember:
    def __init__(self, name):
        self.name = name
        self.children = []

    def add_child(self, child):
        self.children.append(child)

    def display(self, level=0):
        print("  " * level + self.name)
        for child in self.children:
            child.display(level + 1)

# Creating a family tree
grandparent = FamilyMember("Grandparent")
parent1 = FamilyMember("Parent 1")
parent2 = FamilyMember("Parent 2")
child1 = FamilyMember("Child 1")
child2 = FamilyMember("Child 2")
grandchild1 = FamilyMember("Grandchild 1")

grandparent.add_child(parent1)
grandparent.add_child(parent2)
parent1.add_child(child1)
parent1.add_child(child2)
child1.add_child(grandchild1)

# Displaying the family tree
grandparent.display()

# Output:
# Grandparent
#   Parent 1
#     Child 1
#       Grandchild 1
#     Child 2
#   Parent 2
```

Advantages of Trees

1. **Hierarchical representation**: Ideal for representing hierarchical relationships.
2. **Efficient searching**: Enables quick search operations, especially in balanced trees.
3. **Flexible structure**: Can be used to implement other data structures like heaps and sets.

Limitations of Trees

1. **Complexity**: Tree operations can be complex to implement and maintain.
2. **Memory usage**: Can use more memory than linear data structures.

Real-world Applications

- File systems in operating systems
- HTML DOM (Document Object Model) in web browsers
- AI decision trees and game trees

7. Graphs: The Relationship Mappers

What are Graphs?

Graphs are versatile data structures that represent a set of objects (vertices or nodes) where some pairs of objects are connected by links (edges). They're ideal for modeling complex relationships and networks.

How do Graphs work?

Graphs consist of vertices (nodes) and edges (connections between nodes). Edges can be directed (one-way) or undirected (two-way). Graphs can be implemented using adjacency matrices or adjacency lists.

Example: The Social Network Analogy

A social network is a perfect real-world example of a graph. Each person is a vertex, and friendships are edges connecting these vertices.

```
class SocialNetwork:
    def __init__(self):
        self.network = {}

    def add_person(self, name):
        if name not in self.network:
            self.network[name] = set()

    def add_friendship(self, person1, person2):
        self.add_person(person1)
        self.add_person(person2)
        self.network[person1].add(person2)
        self.network[person2].add(person1)

    def display_network(self):
        for person, friends in self.network.items():
            print(f"{person}: {', '.join(friends)}")

# Creating a social network
social_net = SocialNetwork()
social_net.add_friendship("Alice", "Bob")
social_net.add_friendship("Alice", "Charlie")
social_net.add_friendship("Bob", "David")
social_net.add_friendship("Charlie", "David")
social_net.add_friendship("Eve", "Alice")

social_net.display_network()

# Output:
# Alice: Bob, Charlie, Eve
# Bob: Alice, David
# Charlie: Alice, David
# David: Bob, Charlie
# Eve: Alice
```

Advantages of Graphs

1. **Relationship modeling**: Excellent for representing complex relationships and connections.

2. **Versatility**: Can model a wide variety of real-world scenarios.
3. **Powerful algorithms**: Many graph algorithms exist for solving complex problems.

Limitations of Graphs

1. **Complexity**: Can be complex to implement and manage for large datasets.
2. **Memory intensive**: Storing connections can require significant memory.
3. **Traversal challenges**: Some graph problems are computationally expensive to solve.

Real-world Applications

- Social network analysis
- GPS and mapping systems
- Network routing protocols
- Recommendation systems

8. Heaps: The Efficient Priority Managers

What are Heaps?

Heaps are specialized tree-based data structures that satisfy the heap property. In a max heap, for any given node, the node's value is greater than or equal to the values of its children. In a min heap, the node's value is less than or equal to the values of its children.

How do Heaps work?

Heaps maintain a partial ordering of elements. They provide efficient access to the maximum (for max heap) or minimum (for min heap) element, making them ideal for priority queue implementations.

Example: The Emergency Room Triage Analogy

Imagine an emergency room where patients are treated based on the severity of their condition. This system can be modeled using a max heap, where the patient with the highest priority (most severe condition) is always at the top.

```
import heapq

class EmergencyRoom:
    def __init__(self):
        self.patients = []
        self.patient_count = 0

    def add_patient(self, name, priority):
        # We use negative priority for max heap behavior
        heapq.heappush(self.patients, (-priority, self.patient_count, name))
        self.patient_count += 1
        print(f"Patient {name} added with priority {priority}")

    def treat_next_patient(self):
        if self.patients:
            _, _, name = heapq.heappop(self.patients)
            print(f"Treating patient: {name}")
        else:
            print("No patients in waiting.")

    def display_queue(self):
        print("Current queue (Higher number means higher priority):")
        sorted_patients = sorted(self.patients)
        for priority, _, name in sorted_patients:
```



```
er.add_patient("Alice", 5)
er.add_patient("Bob", 1)
er.add_patient("Eve", 4)
```

```
er.display_queue()
er.treat_next_patient()
er.treat_next_patient()
er.display_queue()

# Output:
# Patient John added with priority 3
# Patient Alice added with priority 5
# Patient Bob added with priority 1
# Patient Eve added with priority 4
# Current queue (Higher number means higher priority):
#   Alice: Priority 5
#   Eve: Priority 4
#   John: Priority 3
#   Bob: Priority 1
# Treating patient: Alice
# Treating patient: Eve
# Current queue (Higher number means higher priority):
#   John: Priority 3
#   Bob: Priority 1
```

Advantages of Heaps

1. **Efficient priority management:** Quick access to the highest (or lowest) priority element.
2. **Fast insertion:** O(log n) time complexity for insertion.
3. **Space efficiency:** Can be efficiently implemented as an array.

Limitations of Heaps

1. **Limited access:** Only the top element is easily accessible.
2. **Not suitable for searching:** Searching for a specific element can be inefficient.
3. **Complexity in implementation:** Maintaining the heap property during operations can be tricky.

Real-world Applications

- Task schedulers in operating systems
- Huffman coding in data compression
- Dijkstra's algorithm for finding shortest paths in graphs
- Memory management in programming languages

Conclusion

Understanding these eight fundamental data structures is crucial for any programmer looking to write efficient and optimized code. Each structure has its own strengths and weaknesses, making them suitable for different scenarios:

1. **Arrays** excel at random access and are perfect for scenarios where the size is known and fixed.
2. **Linked Lists** shine in situations requiring frequent insertions and deletions.
3. **Stacks** are ideal for managing function calls and implementing undo mechanisms.
4. **Queues** are perfect for managing tasks in a first-come, first-served basis.
5. **Hash Tables** provide lightning-fast lookups and are great for implementing dictionaries and caches.
6. **Trees** are excellent for representing hierarchical data and enabling efficient searches.
7. **Graphs** are unparalleled in modeling complex relationships and networks.
8. **Heaps** are the go-to structure for priority queue implementations and certain sorting algorithms.

By mastering these data structures, you'll be better equipped to choose the right tool for the job, leading to more efficient and elegant solutions to a wide array of programming challenges. Remember, the key to becoming a proficient programmer lies not just in knowing these structures, but in understanding when and how to apply them effectively in your code.

As you continue your programming journey, practice implementing these data structures from scratch and using them in various scenarios. This hands-on experience will deepen your understanding and help you develop an intuition for which structure to use in different situations. Happy coding!

Data Structures | Computer Science | Programming | Towards Data Science | Algorithms

Written by Suneel Kumar

6.4K Followers

>>>please visit https://www.linkedin.com/in/suneel-kumar-nodejs/

Follow

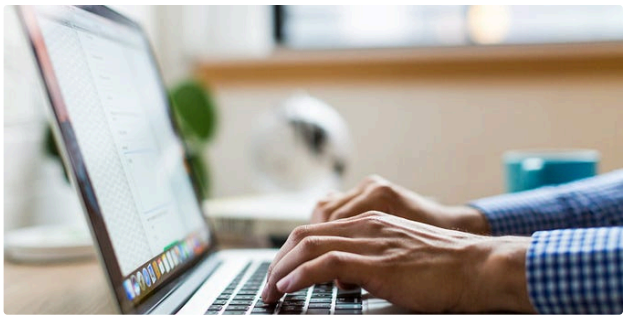
More from Suneel Kumar

Suneel Kumar

Mastering Promise Management in Node.js: Handling Large-Scale...

In the world of modern web development, asynchronous programming has become an...

Jul 11 227



Suneel Kumar

Design Patterns in Node.js

Design patterns are proven solutions to common programming problems. They...

Aug 15, 2023 998 6

Suneel Kumar

JWT Authentication in Nodejs — Refresh JWT with Cookie-based...

JSON Web Tokens (JWTs) are a popular method of authentication that allow you to...

Feb 3, 2023 363 7

Suneel Kumar

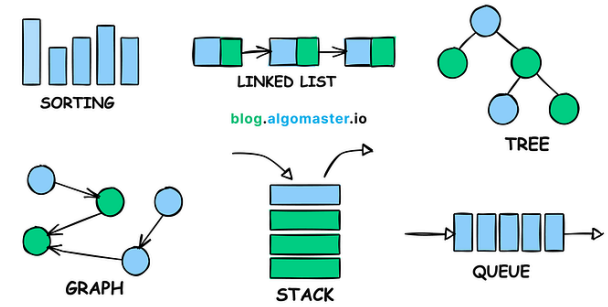
Implementing Role-Based Access Control (RBAC) in Node.js

Role-Based Access Control (RBAC) is a crucial aspect of application security. It...

Oct 19, 2023 633 5

See all from Suneel Kumar

Recommended from Medium

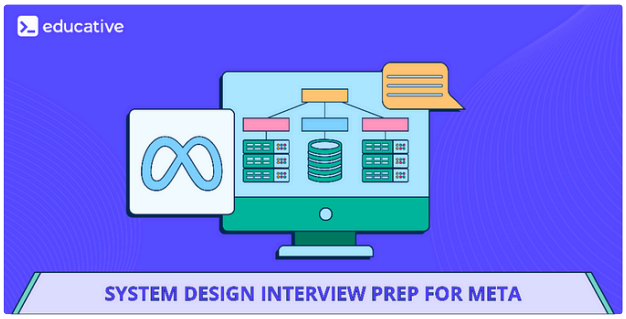


Ashish Pratap Singh in AlgoMaster.io

How I Mastered Data Structures and Algorithms

Getting good at Data Structures and Algorithms (DSA) helped me clear interview...

Jul 23 704 4



Fahim ul Haq in Grokking the Tech Interview

I conducted system design interviews at Meta. Here's how to...

When coordinating your interview at Meta (formerly Facebook), you'll be asked whethe...

Aug 8 75

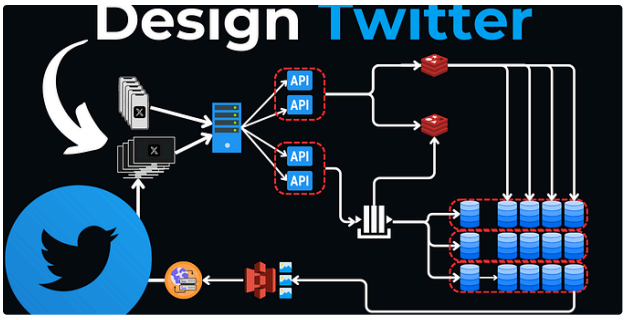
Lists

General Coding Knowledge
20 stories · 1529 saves

Coding & Development
11 stories · 764 saves

Stories to Help You Grow as a Software Developer
19 stories · 1313 saves

ChatGPT
21 stories · 776 saves

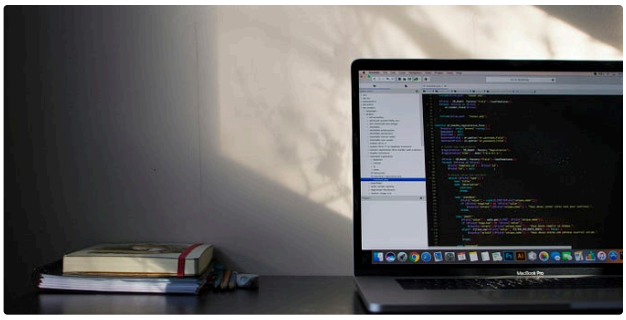


Hayk Simonyan in Level Up Coding

System Design Interview: Design Twitter (X)

Learn how to design social media platforms like Twitter and how to handle billions of...

Aug 6 423 2

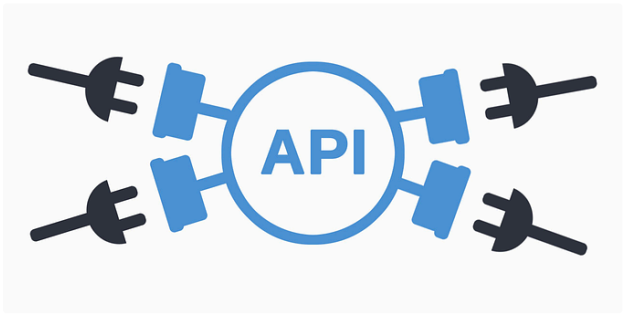


Mahesh Babu

How to Create Custom Annotations in Spring Boot for Cleaner Code

Have you ever found yourself stuck writing the same lines of code over and over again,...

Aug 20 171 2

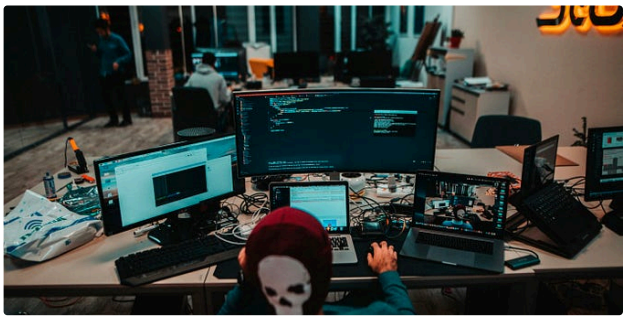


Seliesh Jacob in Dev Genius

API Design: From Basics to Best Practices

Introduction

Jun 3 1.5K 19



Suneel Kumar

Webhook vs. API: Differences, Examples, and Use Cases

In the world of modern web development and software integration, two terms frequently...

Aug 25 2 3

See more recommendations