Comprehensive Documentation: Data Structures and Algorithms

Overview:

This documentation provides an overview of the implementation and operations of several key data structures, including Arrays, Linked Lists, Stack, Queue, HashMap, HashSet, Tree, and Graph. It covers basic operations such as creation, insertion, deletion, and traversal, along with example problems from LeetCode. Additionally, the document provides explanations of some fundamental algorithms.

1. Arrays:

An array is a data structure consisting of a collection of elements, each identified by an index. It stores elements of the same data type in contiguous memory locations.
There are two types of arrays:

- Static arrays
- Dynamic Arrays

Static Array:

They are called static because the size of the array cannot change once declared. And once the array is full, it cannot store additional elements.

- Creation: Initialize an array with a fixed size.
  int [] arr = new int [5];

- Insertion: Add an element at a specific index (requires shifting elements if needed).
- Deletion: Remove an element and shift the remaining elements.
- Traversal: Loop through the array to access or modify its elements.

Time Complexity:

Reading: O (1), Done In constant time.
Insertion:
Time Complexity: O(n) if inserting at a specific position (middle or beginning), as elements need to be shifted.
Time Complexity: O (1) if inserting at the end, provided there's space in the array. If the array is full, a new array is created, and elements are copied, making it O(n).
Deletion:
Time Complexity: O(n) if deleting an element from a specific position, as elements need to be shifted.
Time Complexity: O (1) if deleting from the end, as no shifting is required.

Dynamic Array:

Dynamic Arrays are a much more common alternative to static arrays. They are useful because they can grow as elements are added. Unlike static arrays, with dynamic arrays we don't have to specify a size upon initialization.
Inserting or removing from the middle of a dynamic array would be like a static array. We would have to shift elements to the right or left to make space for the new element or to fill the gap left by the removed element. This would run in O(n)$O(n)$ time.
Resizing a dynamic array: The resize() method doubles the array's capacity, creates a new array with this updated size, and copies all elements from the old array into the new, larger array. Finally, the new array replaces the old array to accommodate future elements.

For each operation:

- **Access:** O (1) – Accessing an element at a specific index takes constant time.

- **Insertion:** O (1) if at the end, O(n) if in the middle, as elements need to be shifted.
- **Deletion:** O (1) if at the end, O(n) if in the middle, since shifting elements is required.

Leetcode:

1. 

27. Remove Element

Solved

Easy | Topics | Companies | Hint

Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` **in-place**. The order of the elements may be changed. Then return *the number of elements in* `nums` *which are not equal to* `val`.

Consider the number of elements in `nums` which are not equal to `val` be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the elements which are not equal to `val`. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int val = ...; // Value to remove
int[] expectedNums = [...]; // The expected answer with correct length.
                            // It is sorted with no values equaling val.

int k = removeElement(nums, val); // Calls your implementation

assert k == expectedNums.length;
sort(nums, 0, k); // Sort the first k elements of nums
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}
```

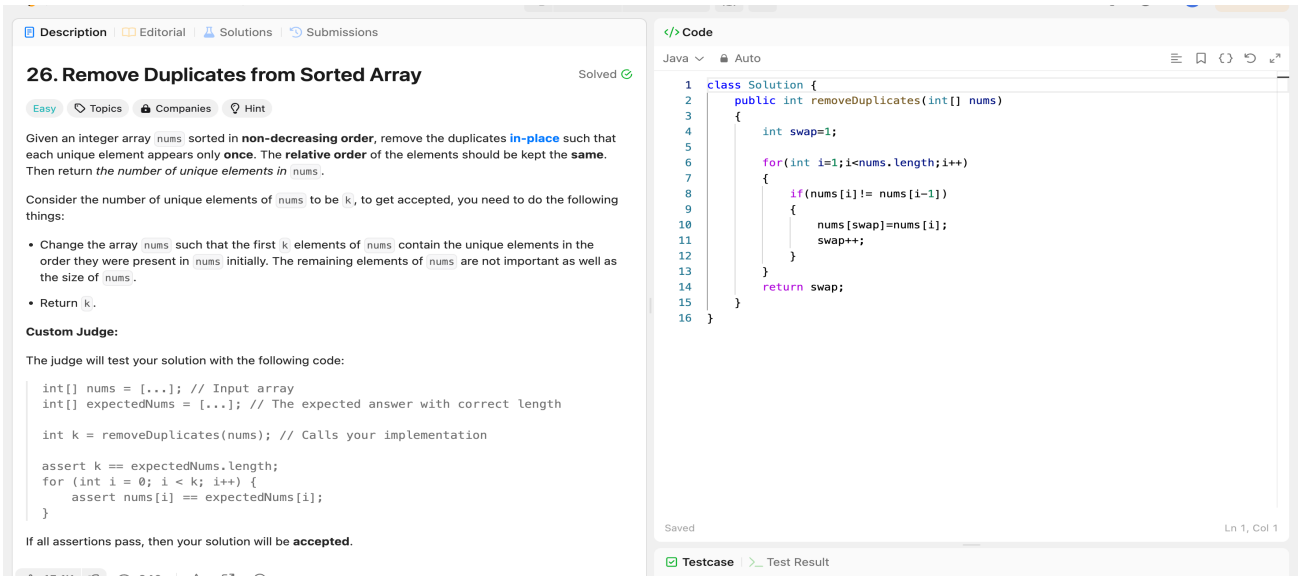
Java | Auto

```
class Solution {
    public void swap(int[] a, int i, int j)
    {
        int temp= a[i];
        a[i]=a[j];
        a[j]=temp;
    }

    public int removeElement(int[] nums, int val)
    {
       int boundary=nums.length-1;
       int i=0;
       while(i <= boundary)
       {
            if(nums[i]==val){
            swap(nums,i,boundary);
            boundary= boundary-1;
            }
            else
            {
            i++;
            }
       }
       return i;
    }
}
```

Saved | Ln 1, Col 1

Explanation:

In this solution, I removed elements equal to val by swapping them with the last element and reducing the array's size, ensuring all occurrences were removed. I found it challenging to work with array indices and pointers like i and boundary. I also had to carefully update the array **in-place** without disturbing the order of the remaining elements.

2. 

Description | Editorial | Solutions | Submissions

26. Remove Duplicates from Sorted Array

Solved

Easy | Topics | Companies | Hint

Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* `nums`.

Consider the number of unique elements of `nums` to be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the unique elements in the order they were present in `nums` initially. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

Code

Java | Auto

```
class Solution {
    public int removeDuplicates(int[] nums)
    {
        int swap=1;

        for(int i=1;i<nums.length;i++)
        {
            if(nums[i]!= nums[i-1])
            {
                nums[swap]=nums[i];
                swap++;
            }
        }
        return swap;
    }
}
```

Saved | Ln 1, Col 1

Testcase | Test Result

Explanation: In this solution, I removed duplicates from a sorted array by iterating through the array and swapping unique elements to the front, I found it challenging to work with the logic for maintaining the position of unique elements (using the swap pointer) and ensuring the array remains **in-place** while keeping the relative order of elements unchanged.

3.

1929. Concatenation of Array

Easy | Topics | Companies | Hint

Given an integer array `nums` of length `n`, you want to create an array `ans` of length `2n` where `ans[i] == nums[i]` and `ans[i + n] == nums[i]` for `0 <= i < n` (**0-indexed**).

Specifically, `ans` is the **concatenation** of two `nums` arrays.

Return *the array* `ans`.

Example 1:

```
Input: nums = [1,2,1]
Output: [1,2,1,1,2,1]
Explanation: The array ans is formed as follows:
- ans = [nums[0],nums[1],nums[2],nums[0],nums[1],nums[2]]
- ans = [1,2,1,1,2,1]
```

Example 2:

```
Input: nums = [1,3,2,1]
Output: [1,3,2,1,1,3,2,1]
Explanation: The array ans is formed as follows:
- ans = [nums[0],nums[1],nums[2],nums[3],nums[0],nums[1],nums[2],nums[3]]
- ans = [1,3,2,1,1,3,2,1]
```

Constraints:

Java | Auto

```java
public class Solution {
    public int[] getConcatenation(int[] nums) {
        int n = nums.length;
        int[] ans = new int[2 * n];

        // Copy elements of nums into the first and second halves of ans
        for (int i = 0; i < n; i++) {
            ans[i] = nums[i];
            ans[i + n] = nums[i];
        }

        return ans;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        // Example usage
        int[] nums = {1, 2, 1};
        int[] result = solution.getConcatenation(nums);

        // Print result
        for (int num : result) {
            System.out.print(num + " ");
        }
        // Output: 1 2 1 1 2 1
    }
}
```

Saved | Ln 29, Col 1

Explanation:

In this solution, I created a new array by concatenating the input array nums with itself. I iterated through the original array, placing its elements in both the first and second halves of the new array., I faced challenges in beginning understanding how to manipulate the index positions while ensuring the correct placement of elements in the concatenated array.

2. Linked list

A linked list is a linear data structure where each element is a separate object, known as a node, containing a value and a reference (or pointer) to the next node in the sequence.

Linked list are of three types:

- Single Linked List
- Doubly Linked List
- Circular Linked List

Operations:

- **Creation**: Initialize a linked list with nodes.
- **Insertion**: Add a new node at the beginning, end, or any position in the list.
- **Deletion**: Remove a node by updating the reference of the previous node.
- **Traversal**: Access each node by following the pointers from one node to the next.

the time complexity for a **linked list**:

- **Access:** O(n) – You must go through the list to find the element at a specific position.
- **Search:** O(n) – To find an element, you need to check each node one by one.
- **Insertion:** O (1) – Inserting is fast if you already know the exact spot (node). Otherwise, it takes O(n) to find it.
- **Deletion:** O (1) – Deleting is fast if you already know the exact node to remove. Otherwise, it takes O(n) to find it.

1.Singly Linked List**:**

- Each node contains data and a reference (or pointer) to the next node in the sequence.
- It allows traversal in only one direction (from the head to the last node).

2.**Doubly** Linked List**:**

- Each node contains data, a reference to the next node, and a reference to the previous node.
- It allows traversal in both directions (from head to tail and vice versa).

3.**Circular** Linked List**:**

- In this list, the last node points back to the head, forming a circular chain.
- Can be singly or doubly linked:
    - **Singly Circular**: Only the last node points to the first node.
    - **Doubly Circular**: Both the last node points to the first node, and the first node points back to the last.

Limitations of Arrays and Advantages of Linked Lists:

Arrays come with a few issues, like having a fixed size, which makes them less flexible when dealing with different amounts of data. Inserting or deleting elements takes time because it needs shifting, and resizing an array adds extra work by copying everything to a new array. Also, memory can get wasted if the array isn't fully used, and handling data that changes often becomes difficult.

On the other hand, linked lists are more flexible. They can grow or shrink as needed without the need for resizing. Insertion and deletion are easier since only the pointers between nodes need to be updated, avoiding the need to shift data. Memory is used more efficiently, as it's allocated only when necessary, making linked lists better for managing data that keeps changing.

Linked List Implementation:

- Node Creation, Insertion at end and insertion at middle.

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedList {
    Node head;

    // Insert at the end
    public void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node last = head;
        while (last.next != null) {
            last = last.next;
        }
        last.next = newNode;
    }

    // Insert in the middle (after a specific value)
    public void insertAfter(int prevData, int data) {
        Node current = head;
        while (current != null && current.data != prevData) {
            current = current.next;
        }
        if (current == null) {
            System.out.println("Previous node not found");
            return;
        }
        Node newNode = new Node(data);
        newNode.next = current.next;
        current.next = newNode;
    }
```

- **Delete at the end, Delete at the middle:**

```java
// Delete at the end
public void deleteAtEnd() {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }
    if (head.next == null) {
        head = null;
        return;
    }
    Node temp = head;
    while (temp.next.next != null) {
        temp = temp.next;
    }
    temp.next = null;
}

// Delete by value (middle)
public void deleteByValue(int data) {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }
    if (head.data == data) {
        head = head.next;
        return;
    }
    Node temp = head;
    while (temp.next != null && temp.next.data != data) {
        temp = temp.next;
    }
    if (temp.next == null) {
        System.out.println("Node with data " + data + " not found");
        return;
    }
    temp.next = temp.next.next;
}
```

LeetCode:

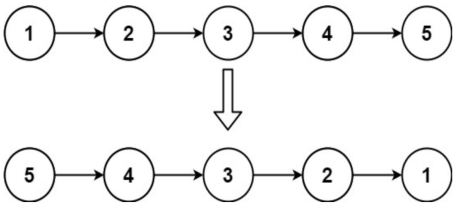
- **Reversing a linked List**

206. Reverse Linked List

Easy | Topics | Companies

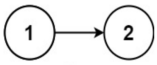
Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:




```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

Example 2:

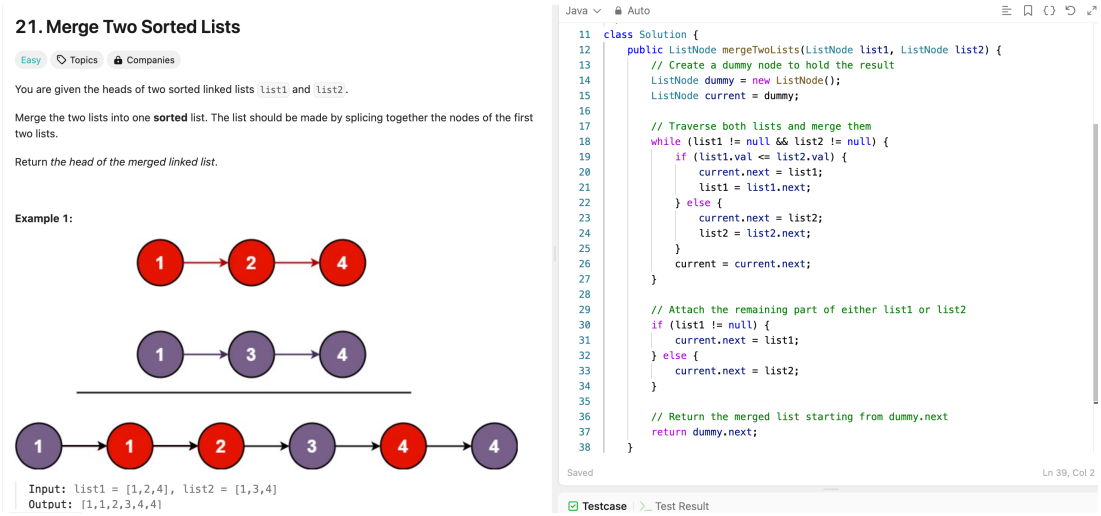


Java | Auto

```java
class ListNode {
    int val;
    ListNode next;

    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;
        while (current != null) {
            ListNode nextTemp = current.next;  // Save the next node
            current.next = prev;               // Reverse the current node's pointer
            prev = current;                    // Move prev forward
            current = nextTemp;                // Move current forward
        }
        return prev;  // prev will be the new head after the entire list is reversed
    }
}
```

Explanation: When I worked on reversing the linked list, I used three pointers: prev, current, and nextTemp. I iterated through the list, reversing the direction of the next pointers, so each node pointed back to the previous one. I kept moving the prev and current pointers forward until the entire list was reversed. In the end, the prev pointer became the new head of the reversed list.

Merge Two Sorted Lists



21. Merge Two Sorted Lists

Easy | Topics | Companies

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:


```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Java | Auto

```java
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        // Create a dummy node to hold the result
        ListNode dummy = new ListNode();
        ListNode current = dummy;

        // Traverse both lists and merge them
        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) {
                current.next = list1;
                list1 = list1.next;
            } else {
                current.next = list2;
                list2 = list2.next;
            }
            current = current.next;
        }

        // Attach the remaining part of either list1 or list2
        if (list1 != null) {
            current.next = list1;
        } else {
            current.next = list2;
        }

        // Return the merged list starting from dummy.next
        return dummy.next;
    }
```

Saved | Ln 39, Col 2

Testcase | Test Result

Explanation: I had to merge two sorted linked lists into one sorted linked list. To do this, I created a dummy node to act as the start of the merged list. Then, I compared the nodes from both lists one by one, attaching the smaller node to the result list and moving to the next node. Once one of the lists was fully traversed, I simply attached the remaining part of the other list. Finally, I returned the merged list starting from dummy.next.

3.Stack:

A stack is a data structure that allows adding and removing elements from only one end, called the top. You can think of it like a stack of plates: you can only add or remove a plate from the top, not from the middle. It Follows LIFO (Last In First Out).

Key Operations:

- **Push:** Adds an element to the top of the stack.
  - Time Complexity: O (1)
- **Pop:** Removes the element from the top of the stack.
  - Time Complexity: O (1) (but first, check if the stack is empty)
- **Peek:** Returns the top element without removing it.
  - Time Complexity: O (1)

All these operations are efficient and take constant time, meaning they run in O (1**).**

We can implement a stack using:

1. Array: Add and remove elements from the end of the array.
2. Linked List: Add and remove elements from the front of the list.

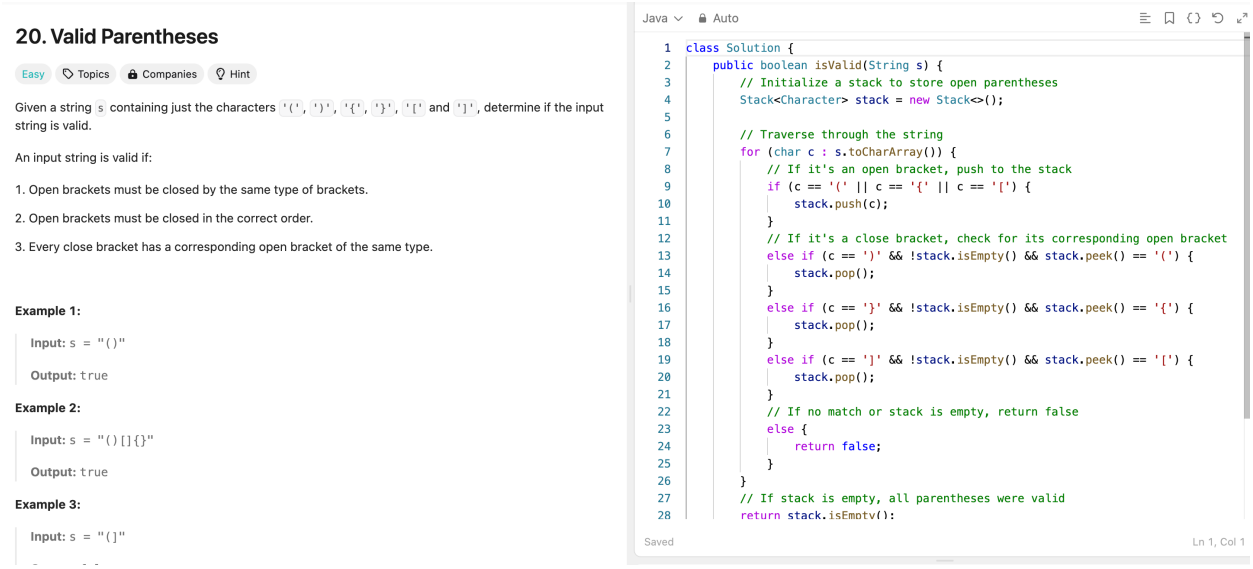
Stack Implementation Using Array:

```java
class StackArray {
    private int[] stack;
    private int top;
    private int capacity;

    public StackArray(int size) {
        stack = new int[size];
        capacity = size;
        top = -1;
    }

    public void push(int value) {
        if (top == capacity - 1) {
            System.out.println("Stack Overflow");
            return;
        }
        stack[++top] = value;
    }

    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return -1;
        }
        return stack[top--];
    }

    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return -1;
        }
        return stack[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }
}
```

Stack Implementation Using Linked List

```java
class Node {
    int value;
    Node next;

    public Node(int value) {
        this.value = value;
    }
}

class StackLinkedList {
    private Node head;

    public StackLinkedList() {
        head = null;
    }

    public void push(int value) {
        Node newNode = new Node(value);
        newNode.next = head;
        head = newNode;
    }

    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return -1;
        }
        int poppedValue = head.value;
        head = head.next;
        return poppedValue;
    }

    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return -1;
        }
        return head.value;
    }

    public boolean isEmpty() {
        return head == null;
    }
}
```

LeetCode:



20. Valid Parentheses

Easy | Topics | Companies | Hint

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

> **Input:** `s = "()"`
>
> **Output:** `true`

Example 2:

> **Input:** `s = "()[]{}"`
>
> **Output:** `true`

Example 3:

> **Input:** `s = "(]"`

Java | Auto

```java
class Solution {
    public boolean isValid(String s) {
        // Initialize a stack to store open parentheses
        Stack<Character> stack = new Stack<>();

        // Traverse through the string
        for (char c : s.toCharArray()) {
            // If it's an open bracket, push to the stack
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            }
            // If it's a close bracket, check for its corresponding open bracket
            else if (c == ')' && !stack.isEmpty() && stack.peek() == '(') {
                stack.pop();
            }
            else if (c == '}' && !stack.isEmpty() && stack.peek() == '{') {
                stack.pop();
            }
            else if (c == ']' && !stack.isEmpty() && stack.peek() == '[') {
                stack.pop();
            }
            // If no match or stack is empty, return false
            else {
                return false;
            }
        }
        // If stack is empty, all parentheses were valid
        return stack.isEmpty();
```

Saved | Ln 1, Col 1

Explanation: I used a stack to solve this problem by pushing every opening bracket onto it. For every closing bracket, I checked if the top of the stack matched the correct opening one and popped it if it did. If there was no match or the stack was empty when it shouldn't be, I returned false. Finally, if the stack was empty at the end, I knew all the parentheses were valid, so I returned true.

4.QUEUE

Queues are another data structure like arrays, but they work on a First In, First Out (FIFO) principle, unlike stacks. A real-life example of a queue is a line at the bank where the first person in line is the first to be served.

Implementation and Operations:

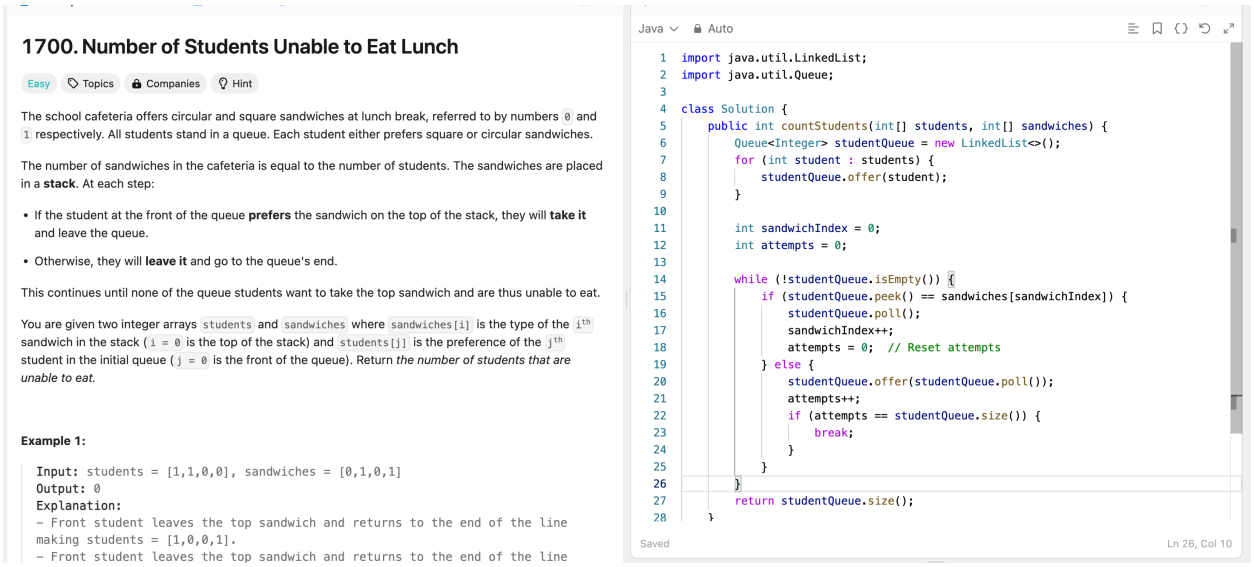
- Linked List: The easiest way to implement a queue is by using a linked list.
- Dynamic Array: It's possible to use a dynamic array, but to achieve the same efficiency as a linked list, you'd need a circular array with extra steps.

Key Operations:

1. Enqueue: Add an element to the end of the queue.
2. Dequeue: Remove an element from the front of the queue.

This structure is widely used when order matters, like processing tasks or managing requests.

LeetCode:



1700. Number of Students Unable to Eat Lunch

Easy | Topics | Companies | Hint

The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers `0` and `1` respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a **stack**. At each step:

- If the student at the front of the queue **prefers** the sandwich on the top of the stack, they will **take it** and leave the queue.
- Otherwise, they will **leave it** and go to the queue's end.

This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays `students` and `sandwiches` where `sandwiches[i]` is the type of the i^{th} sandwich in the stack (`i = 0` is the top of the stack) and `students[j]` is the preference of the j^{th} student in the initial queue (`j = 0` is the front of the queue). Return *the number of students that are unable to eat.*

Example 1:

```
Input: students = [1,1,0,0], sandwiches = [0,1,0,1]
Output: 0
Explanation:
- Front student leaves the top sandwich and returns to the end of the line
making students = [1,0,0,1].
- Front student leaves the top sandwich and returns to the end of the line
```

Java | Auto

```java
import java.util.LinkedList;
import java.util.Queue;

class Solution {
    public int countStudents(int[] students, int[] sandwiches) {
        Queue<Integer> studentQueue = new LinkedList<>();
        for (int student : students) {
            studentQueue.offer(student);
        }

        int sandwichIndex = 0;
        int attempts = 0;

        while (!studentQueue.isEmpty()) {
            if (studentQueue.peek() == sandwiches[sandwichIndex]) {
                studentQueue.poll();
                sandwichIndex++;
                attempts = 0;  // Reset attempts
            } else {
                studentQueue.offer(studentQueue.poll());
                attempts++;
                if (attempts == studentQueue.size()) {
                    break;
                }
            }
        }
        return studentQueue.size();
    }
```

Saved | Ln 26, Col 10

Explanation: I used a queue to represent the students and an index to track the sandwiches. I kept checking if the student at the front wanted the current sandwich. If they didn't, I moved them to the back of the queue. I stopped when all students cycled through without anyone taking a sandwich and returned the number of students left in the queue.

5. **Hash Table:**

- **Definition**: A **hash table** is a data structure that maps keys to values using a **hash function** to compute an index (or hash code) where the value is stored.
- **Key-Value Pair**: It stores data in **key-value** pairs.
- **How it works**: The hash function converts the key into a number (hash code), and that number is used to find the index in the table where the value is stored.
- **Efficiency**: Provides **O(1)** average time complexity for insertion, deletion, and search operations.
- **Collision Handling**: Deals with **collisions** (when two keys hash to the same index) using techniques like **chaining** (linked lists) or **open addressing**.

LeetCode:

217. Contains Duplicate

Easy | Topics | Companies

Given an integer array `nums`, return `true` if any value appears **at least twice** in the array, and return `false` if every element is distinct.

Example 1:

Input: `nums = [1,2,3,1]`

Output: `true`

Explanation:

The element 1 occurs at the indices 0 and 3.

Example 2:

Input: `nums = [1,2,3,4]`

Output: `false`

Explanation:

All elements are distinct.

Example 3:

Input: `nums = [1,1,1,3,3,4,3,2,4,2]`

Java | Auto

```java
import java.util.HashSet;
public class DuplicateChecker {
    public boolean containsDuplicate(int[] nums) {
        HashSet<Integer> set = new HashSet<>();
            for (int num : nums) {
            if (set.contains(num)) {
                return true;
            }
            set.add(num);
        }
        return false;
    }

    public static void main(String[] args) {
        DuplicateChecker checker = new DuplicateChecker();
        int[] nums = {1, 2, 3, 4, 5, 1};

        System.out.println(checker.containsDuplicate(nums));  // Output: true
    }
}
```

Saved | Ln 11, Col 1

Explanation: I used a **HashSet** to solve the problem. I iterated through the array, and for each number, I checked if it was already in the set. If it was, I returned true because that meant a duplicate was found. If not, I added the number to the set and continued. If no duplicates were found by the end, I returned false.

Part-2: Comprehensive Documentation of Algorithms.

I have completed studying key algorithms such as **Insertion Sort, Merge Sort, Quicksort**, and searching algorithms like **Linear Search, Binary Search**. Also, Dijkstra's **Algorithm & Bellman-Ford Algorithm & Graph Traversals: BFS and DFS, Fractional Knapsack Problem**. These were essential in understanding the foundations of how data is organized and efficiently searched.

Key Learnings:

- **Sorting algorithms** differ primarily in their time complexity and use cases.
- **Merge Sort** is an example of a Divide and Conquer algorithm, ensuring O(n log n) time complexity.
- **Binary Search** works on sorted arrays with O(log n) complexity for efficient searching.
- **Dijkstra's Algorithm** uses a priority queue to explore the shortest paths from a source node to all other nodes.
- **Bellman-Ford Algorithm** can handle graphs with negative weights, unlike Dijkstra's algorithm.
- **BFS** explores nodes level by level, making it ideal for finding the shortest path in unweighted graphs.
- **DFS** explores as deep as possible before backtracking, making it useful for problems that require exploring all possibilities, like solving mazes.
- **Greedy algorithms** work by making the most optimal choice at each step, leading to a globally optimal solution.
- The **Fractional Knapsack problem** can be solved efficiently by sorting items and selecting them until the knapsack's capacity is filled.

Algorithms

1.Quick Sort**:**

```java
public class QuickSort {
    public static int[] quickSort(int[] arr, int s, int e) {
        if (e - s + 1 <= 1) {
            return arr;
        }

        int pivot = arr[e];
        int left = s;       // pointer for left side

        // Partition: elements smaller than pivot on left side
        for (int i = s; i < e; i++) {
            if (arr[i] < pivot) {
                int tmp = arr[left];
                arr[left] = arr[i];
                arr[i] = tmp;
                left++;
            }
        }

        // Move pivot in-between left & right sides
        arr[e] = arr[left];
        arr[left] = pivot;

        // Quick sort left side
        quickSort(arr, s, left - 1);

        // Quick sort right side
        quickSort(arr, left + 1, e);

        return arr;
    }
}
```

Explanation:

Quicksort selects a "pivot" element and partitions the array into two sub-arrays: elements less than the pivot and elements greater than the pivot. It then recursively applies the same logic to both sub-arrays. The partitioning is done in-place, making Quicksort memory-efficient.

Use Case:

Used when average-case performance is crucial. It is the algorithm of choice for large datasets and is often preferred for its efficiency in practice, despite its worst-case complexity. It is used in many systems as the default sorting algorithm.

Time Complexity:

- Best/Average Case: O (n log n) (when partitions are balanced).
- Worst Case: $O(n^2)$ (if the pivot is poorly chosen, e.g., in already sorted or reverse-sorted data).

2.Insertion Sort:

```java
public class InsertionSort {
    public static int[] insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int j = i - 1;
            while (j >= 0 && arr[j+1] < arr[j]) {
                int tmp = arr[j + 1];
                arr[j + 1] = arr[j];
                arr[j] = tmp;
                j--;
            }
        }
        return arr;
    }
}
```

Explanation:

Insertion Sort iterates through the array, taking each element and inserting it into its correct position relative to the already sorted part of the array. It compares the current element with the ones before it, shifting larger elements one position to the right until the correct position is found.

Use Case:

Used for small datasets or nearly sorted data due to its simplicity and efficiency in these cases.

Time Complexity:

- **Best Case:** O(n) (when the array is already sorted).
- **Average/Worst Case:** $O(n^2)$ (when elements are randomly or reversely sorted).

3.Merge Sort:

```java
public class MergeSort {

    public static int[] mergeSort(int[] arr, int l, int r) { // array, starting index of array, last index of array
        if (l < r) {

            // Find the middle point of arr
            int m = (l + r) / 2;

            mergeSort(arr, l, m);   // sort left half
            mergeSort(arr, m+1, r); // sort right half
            merge(arr, l, m, r);    // merge sorted halfs
        }
        return arr;
    }


    public static void merge(int[] arr, int l, int m, int r) {

        // Find lengths of two subarrays to be merged
        int length1 = m - l + 1;
        int length2 = r - m;

        // Create temp arrays
        int L[] = new int[length1];
        int R[] = new int[length2];

        // Copy the sorted left & right halfs to temp arrays
        for (int i = 0; i < length1; i++) {
            L[i] = arr[l + i];
        }

        for (int j = 0; j < length2; j++) {
            R[j] = arr[m + 1 + j];
        }

        // initial indexes of left and right sub-arrays
        int i = 0; // index for left
        int j = 0; // index for right
        int k = l; // Initial index of merged subarray array

        while(i < length1 && j < length2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
```

Explanation:
Merge Sort is a stable, Divide and Conquer algorithm. It splits the array into two halves, recursively sorts each half, and merges the sorted halves back together. The merge operation requires additional space to hold the temporary arrays.

Use Case:
Ideal for large datasets where guaranteed O(n log n) performance is required, such as for linked lists or external sorting (when the data doesn't fit into memory). It is also used in stable sorting algorithms since it maintains the relative order of equal elements.

Time Complexity:

- Best/Average/Worst Case: O (n log n) (since the array is always split in half and merged back).

4. Linear Search:

```java
public class Array {
    // Method to perform linear search
    public static int linearSearch(int[] arr, int target) {
        // Iterate through the array
        for (int i = 0; i < arr.length; i++) {
            // If the current element matches the target, return its index
            if (arr[i] == target) {
                return i;
            }
        }
        // Return -1 if the target is not found in the array
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {10, 23, 45, 70, 11, 15};
        int target = 45;
        int result = linearSearch(arr, target);

        if (result != -1) {
            System.out.println("Element found at index: " + result);
        } else {
            System.out.println("Element not found in the array");
        }
    }
}
```

Explanation:
Linear Search is a straightforward algorithm that checks each element of the array sequentially, starting from the first element and moving toward the last. For each element, the algorithm compares it with the target value. If the target is found, the algorithm immediately returns the index of the element. If it reaches the end of the array without finding the target, it returns -1. Linear Search works on both sorted and unsorted arrays, making it a flexible solution when the data structure is unordered. However, it can be inefficient for large datasets since every element must be checked individually.
Use Case: Linear search is ideal for small datasets or when the array is unsorted, and you need a simple approach to finding an element without prior knowledge of the data order.
Time Complexity:

- **Best Case**: O (1), if the target is at the first position.
- **Worst/Average Case**: O(n), where n is the size of the array, as it may require checking every element.

5. Binary Search:

```
public class Array {
    public static int binarySearch(int[] arr,  int target) {
        int L = 0, R = arr.length - 1;
        int mid;

        while (L <= R) {
            mid = (L + R) / 2;
            if (target > arr[mid]) {
                L = mid + 1;
            } else if (target < arr[mid]) {
                R = mid - 1;
            } else {
                return mid;
            }
        }
        return - 1;
    }
}
```

Explanation:
Binary Search is a more efficient search algorithm but requires the array to be sorted. The algorithm works by repeatedly dividing the search space in half. It starts by comparing the target value to the middle element of the array. If the target is equal to the middle element, the search is successful, and the index is returned. If the target is smaller than the middle element, the algorithm continues searching in the left half of the array. If the target is larger, it searches the right half. This process repeats, cutting the search space in half at each step, until the target is found, or the search space is exhausted.
Use Case: Binary search is highly efficient and is typically used when working with large, sorted datasets, such as in database indexing or looking up entries in a phone book.
Time Complexity:

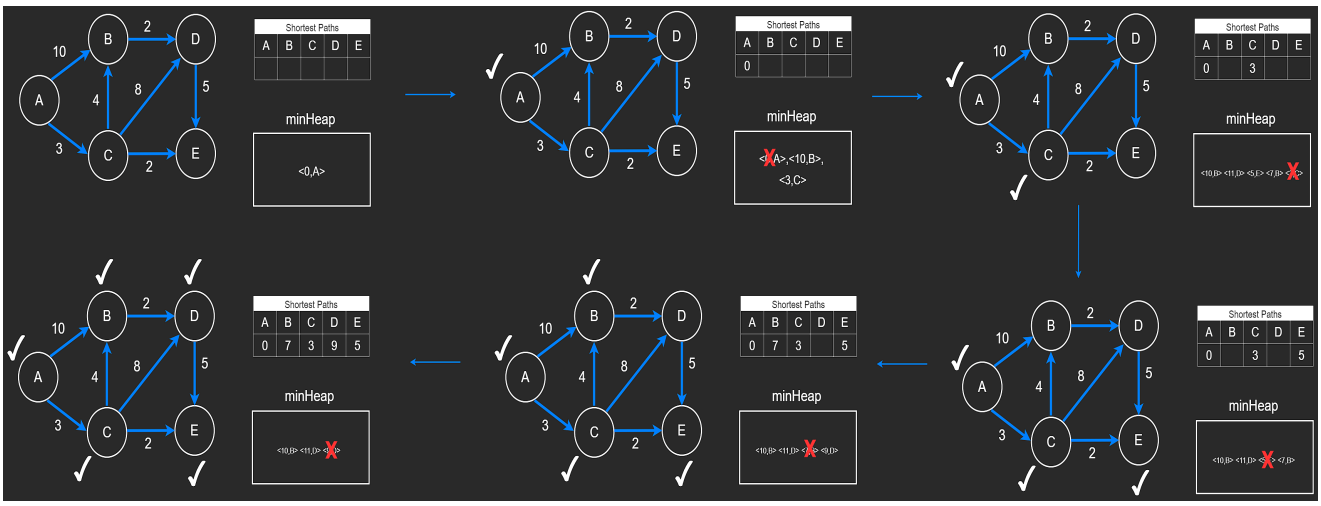
- Best Case: O (1), if the target is in the middle of the array.
- Worst/Average Case: O (log n), where n is the size of the array, as the search space is halved with each iteration.

6. Dijkstra's Algorithm:

Dijkstra's Algorithm finds the shortest path from a source node to all other nodes in a graph with non-negative weights. It starts from the source and explores the nearest unvisited node, updating the shortest known distances to each node. It uses a priority queue to efficiently pick the next closest node. The

algorithm keeps updating the path lengths until all nodes are visited. It's widely used in routing, such as finding the fastest path in GPS systems. However, it doesn't work with negative weights.

How I Understood It:
I understood Dijkstra's algorithm by imagining it like planning the shortest route in a city. If you're standing at point A and want to visit points B, C, and D, you first find the nearest location and mark it as "visited." Then, you update the routes to the remaining locations based on the new information from your current spot. Repeat this until all locations have been visited, and you'll have the shortest path to each.



It starts at the source node, in this case, node A, with a distance of 0, while all other nodes (B, C, D, E) are initially set to infinity. The algorithm first selects the closest unvisited node, which is node C, with a distance of 3, and updates its neighbors. From C, it updates the distance to B (7 through C) and E (5 through C). Once C is marked as visited, the algorithm moves to E, which has the next smallest distance of 5. Since E has no neighbors to update, the next closest node, B, is selected. B's distance is now 7, and its neighbor D is updated with a new distance of 9. Finally, the algorithm moves to D. At this point, all nodes have been visited, and the shortest paths from A to all other nodes are determined. The final shortest paths are: A to B is 7, A to C is 3, A to D is 9, and A to E is 5. This process demonstrates how Dijkstra's Algorithm efficiently finds the shortest paths by continuously selecting the nearest unvisited node and updating the distances of its neighbors.

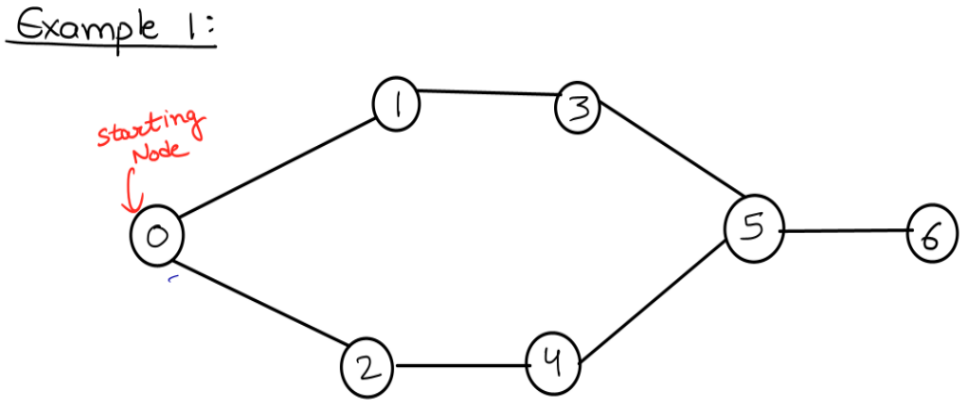
7. **Graph Traversal BFS**

BFS is a graph traversal technique that explores nodes level by level, visiting all neighbors of a node before moving to the next level of neighbors. It uses a queue to keep track of the nodes to be explored. BFS ensures that nodes closer to the source are visited first.

Step-by-Step:
- Start with the source node, marking it as visited.
- Add the source node to a **queue** (FIFO structure).
- While the queue is not empty:
- Remove a node from the front of the queue.
- Visit all its unvisited neighbors and add them to the queue.
- Repeat this process until all reachable nodes have been visited.

Algorithm Steps:

- **Queue Initialization**: Insert the starting node (in this case, node 0) into the queue.
- **Visited Array**: Use a visited array to mark nodes as visited. For example, node 0 will be marked as visited by setting visited[0] = 1.
- **Mark Starting Node**: The starting node is visited, and the queue holds the nodes to explore.
- **Process Each Node**: While the queue is not empty:
  Remove the node from the front of the queue (FIFO).
  Insert all of the node's unvisited neighbors into the queue and mark them as visited.



Traversal Order:
The BFS traversal order is **0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6**.

BFS Process:

1. Start from node 0: Mark it as visited and push it into the queue.
   - Queue: [0]
   - Visited: [1, 0, 0, 0, 0, 0, 0]
2. Visit neighbors of node 0: Nodes 1 and 2 are neighbors of node 0. Mark them as visited and push them into the queue.
   - Queue: [1, 2]
   - Visited: [1, 1, 1, 0, 0, 0, 0]
3. Dequeue node 1 and visit its neighbors: Node 0 is already visited. Visit node 3.
   - Queue: [2, 3]
   - Visited: [1, 1, 1, 1, 0, 0, 0]
4. Dequeue node 2 and visit its neighbors: Node 0 is already visited. Visit node 4.
   - Queue: [3, 4]
   - Visited: [1, 1, 1, 1, 1, 0, 0]
5. Dequeue node 3 and visit its neighbors: Node 1 is already visited. Visit node 5.
   - Queue: [4, 5]
   - Visited: [1, 1, 1, 1, 1, 1, 0]
6. Dequeue node 4 and visit its neighbors: Node 2 is already visited. Node 5 is already visited. No new nodes to visit.
   - Queue: [5]
7. Dequeue node 5 and visit its neighbors: Nodes 3 and 4 are already visited. Visit node 6.
   - Queue: [6]
   - Visited: [1, 1, 1, 1, 1, 1, 1]
8. Dequeue node 6: All neighbors are visited, and the queue is empty. BFS is complete.

8. Graph traversal DFS: Depth-First Search (DFS) is a graph traversal algorithm that explores a graph by going as deep as possible along a branch before backtracking to explore other branches. DFS uses a stack or recursion to keep track of the nodes and follows a last-in, first-out (LIFO) approach. The algorithm ensures that all nodes are visited by recursively diving into each node's neighbors.

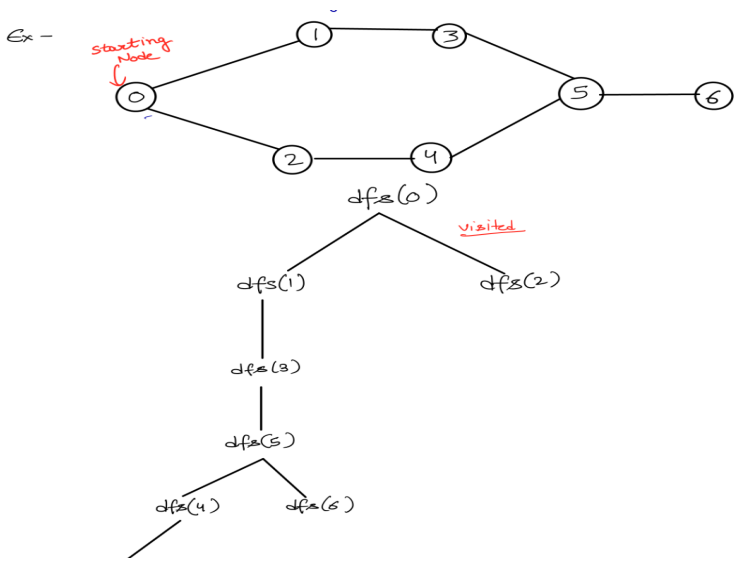
How It Works:

- DFS starts at a given node (source) and explores as far down one path as possible, visiting each node along the way.
- If a node has no unvisited neighbors, the algorithm backtracks to the previous node to explore other branches.
- This process continues until all nodes have been visited.
- DFS can be implemented iteratively using a stack or recursively.

DFS Algorithm Process:

1. **Start at the source node**: Mark the source node as visited.
2. **Push the source node onto the stack** (if using an iterative approach).
3. While the **stack is not empty** (or until the recursion is complete):
    - **Pop a node** from the stack (or call recursively) and visit it.
    - For each of its unvisited neighbors:
        - Mark the neighbor as visited.
        - Push the neighbor onto the stack (or call recursively to visit the neighbor).
4. **Backtrack**: If a node has no unvisited neighbors, backtrack to the previous node by popping from the stack or returning from the recursive call.
5. **Repeat** the process until all nodes in the graph are visited.

Example:



DFS traversal order for this graph is: 0 -> 1 -> 3 -> 5 -> 6 -> 2 -> 4.

Depth-First Search (DFS) Process:

1. Start at node 0:
    - Mark node 0 as visited.
    - Move to its first unvisited neighbor (node 1).
2. Visit node 1:
    - Mark node 1 as visited.

    - Move to its first unvisited neighbor (node 3).
3. Visit node 3:
    - Mark node 3 as visited.
    - Move to its first unvisited neighbor (node 5).
4. Visit node 5:
    - Mark node 5 as visited.
    - Move to its first unvisited neighbor (node 6).
5. Visit node 6:
    - Mark node 6 as visited.
    - Node 6 has no unvisited neighbors, so backtrack to the previous node (node 5).
6. Backtrack to node 5:
    - Node 5 has no more unvisited neighbors, so backtrack to node 3.
7. Backtrack to node 3:
    - Node 3 has no more unvisited neighbors, so backtrack to node 1.
8. Backtrack to node 1:
    - Node 1 has no more unvisited neighbors, so backtrack to node 0.
9. Backtrack to node 0:
    - Now, explore the second unvisited neighbor of node 0 (which is node 2).
10. Visit node 2:
    - Mark node 2 as visited.
    - Move to its first unvisited neighbor (node 4).
11. Visit node 4:
    - Mark node 4 as visited.
    - Node 4 has no unvisited neighbors, so backtrack to node 2.
12. Backtrack to node 2:
    - Node 2 has no more unvisited neighbors, so backtrack to node 0.
13. Backtrack to node 0:
    - Now, all nodes have been visited.

Conclusion for Documentation on Data Structures and Algorithms.

As I worked through **Data Structures** and **Algorithms**, I realized how crucial they are for solving problems efficiently. Each data structure, like **Arrays**, **Linked Lists**, or **Stacks**, serves a specific purpose, and knowing when to use them makes a big difference.

The algorithms I explored, such as **Binary Search** and **DFS**, showed me how the right approach can improve performance. This journey taught me that DS and algorithms are not just about theory but practical tools I can apply in real-world scenarios.

In the end, understanding these concepts helps me write better, more efficient code, and prepares me for bigger challenges ahead.