Data Structures & Algorithms

By: TRAN THI KIEU TRANG

Introduction to Data Structures

Data structures are specialized formats for organizing, processing, and storing data.

They are fundamental for the efficient implementation of algorithms and managing large amounts of data.

Categories: Linear (e.g., arrays, stacks) and Non-linear (e.g., trees, graphs).

Identifying Data Structures

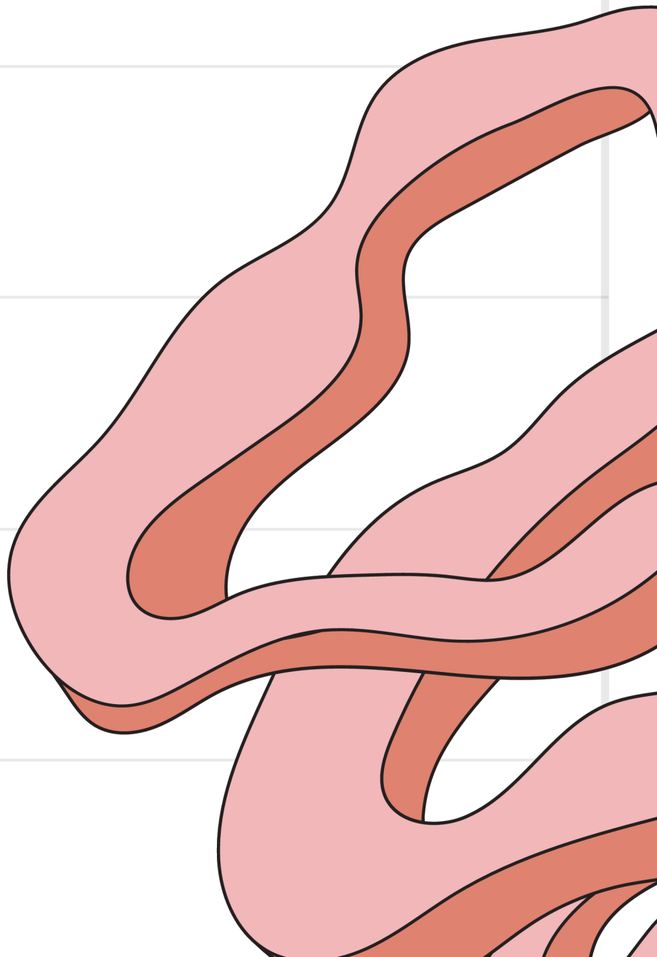
- **Arrays**: Contiguous blocks of memory; fast access using index, but costly insertions/deletions.
- **Linked Lists:** Elements are linked through pointers; dynamic size, but slower access.
- **Stacks:** LIFO structure, supports operations like push/pop; used for recursion, backtracking.
- **Queues:** FIFO structure, elements are processed in order; used in task scheduling, buffering.
- **Trees:** Hierarchical structure, used in searching/sorting (e.g., binary trees).
- **Hash Tables:** Allows fast lookup through hash functions; used for databases, caching.

```java
public void push(Student student) {
    if (top < stack.length - 1) {
        stack[++top] = student;
    } else {
        System.out.println("Stack overflow");
    }
}

public Student pop() {
    if (top == -1) {
        System.out.println("Stack is empty");
        return null;
    } else {
        return stack[top--];
    }
}
```

Operations on Data Structures

- **Insert:** Add an element to the data structure.
- **Delete:** Remove an element.
- **Search:** Find an element in the structure.
- **Access:** Retrieve an element by index/key.
- **Traverse:** Visit all elements.
- **Sort:** Rearrange elements according to a criterion.

```
Student newStudent = new Student(id, name, grade);
management.addStudent(newStudent);  // Add a student


management.removeStudent(removeId);  // Remove a student by ID
```



Defining Input Parameters

Input Parameters:

- Every operation needs input data.
- For example:
  - Insert in Array: Array reference, element to insert, and index position.
  - Search in Linked List: The key or value to search for.
  - Push to Stack: The element to be added.

```
public void addStudent(Student student) {
    students.push(student);  // Here, the student object is the input parameter
}
```

Pre-conditions & Post-conditions

- **Pre-conditions:**

Define what must be true before the operation.
Example: Before inserting, there must be space available in the data structure.

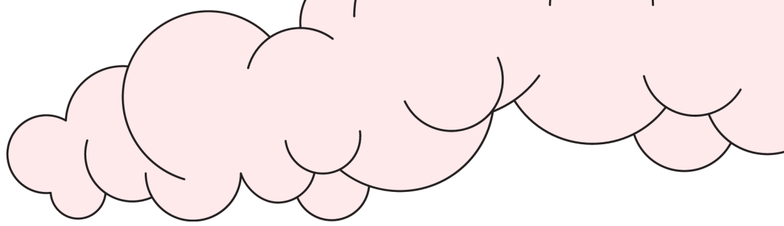
- **Post-conditions:**

Define what will be true after the operation is executed.
Example: After removing a student, the stack size will decrease.

- **Example Pre/Post-conditions for Stack Push:**

Pre-condition: Stack is not full.
Post-condition: New element is added at the top of the stack.

Time and Space Complexity



Time Complexity:

Describes how the runtime of an operation scales with the size of the input.

Examples:

Insert in Array: O(n) in the worst case (shifting elements).

Search in Linked List: O(n), where n is the size of the list.

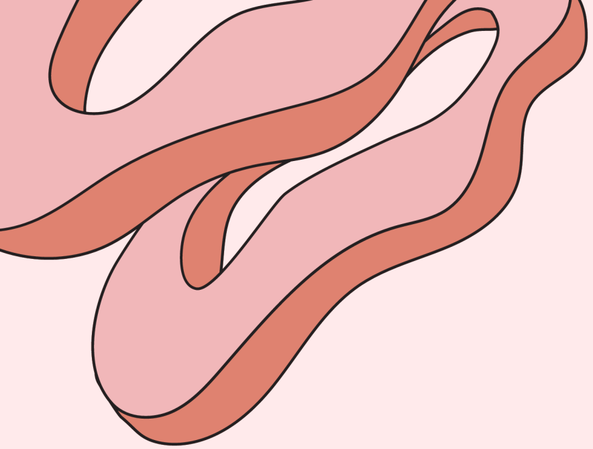
Stack Push/Pop: O(1) constant time.

Space Complexity:

Describes how the memory usage of an algorithm scales with the input size.

Example:

Stack Operations (Push/Pop): Both have constant time complexity O(1).



Memory Stack: Definition

- Definition: A memory stack is a special type of data structure that supports function calls by storing function state (local variables, return address) when a function is called.
- Importance: It helps manage recursive function calls and ensures that once a function completes, it returns to the correct point in the code.

Memory Stack Operations

- Push: Save function state (local variables, return address) onto the stack when a function is called.
- Pop: Remove and restore the function state from the stack when the function returns.

```
public void push(Student student) {
    stack[++top] = student;
}

public Student pop() {
    return stack[top--];
}
```

Function Call Implementation with Stack

- Stack Frames: Each function call creates a stack frame that stores its local variables, return address, and parameters.
- LIFO Order: When the function finishes execution, its stack frame is popped, returning control to the calling function.

Example: When recursive functions are called, stack frames are pushed onto the stack.

FIFO Queue: Introduction

01

Definition: FIFO (First In, First Out) queue allows elements to be inserted at the rear and removed from the front.

02

Common Uses: Task scheduling, buffering data streams, or breadth-first search in graph algorithms.

FIFO Queue: Array-Based Implementation

Use an array to implement the queue with pointers for the front and rear.

```
class Queue {
    private int[] elements;
    private int front, rear, size, capacity;

    public Queue(int capacity) {
        elements = new int[capacity];
        front = 0;
        size = 0;
        rear = capacity - 1;
    }

    public void enqueue(int item) {
        rear = (rear + 1) % capacity;
        elements[rear] = item;
        size++;
    }

    public int dequeue() {
        int item = elements[front];
        front = (front + 1) % capacity;
        size--;
        return item;
    }
}
```

FIFO Queue: Linked List–Based Implementation

Use nodes linked by pointers to represent the queue. Enqueue adds to the tail, and dequeue removes from the head.

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

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

class LinkedQueue {
    private Node front, rear;

    public void enqueue(int item) {
        Node newNode = new Node(item);
        if (rear != null) rear.next = newNode;
        rear = newNode;
        if (front == null) front = rear;
    }

    public int dequeue() {
        if (front == null) throw new NoSuchElementExceptio
        int item = front.data;
        front = front.next;
        if (front == null) rear = null;
        return item;
    }
}
```

Performance Comparison of Sorting Algorithms

Algorithm 1: QuickSort

- Average Time Complexity: O(n log n).
- Worst Case Time Complexity: O(n^2).
- Space Complexity: O(log n) with in-place sorting.

- Algorithm 2: MergeSort
  - Time Complexity: O(n log n) for all cases.
  - Space Complexity: O(n) due to additional memory for temporary arrays.

COMPARISON TABLE: QUICKSORT VS. MERGESORT

Criteria	QuickSort	MergeSort
Time Complexity	O(n log n) (avg), O(n^2) (worst)	O(n log n)
Space Complexity	O(log n) (in-place)	O(n) (requires extra space)
Stability	No	Yes
Performance in Practice	Fast in average case for random data	Stable performance but slower

EXAMPLE: SORTING PERFORMANCE

```java
public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```java
public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

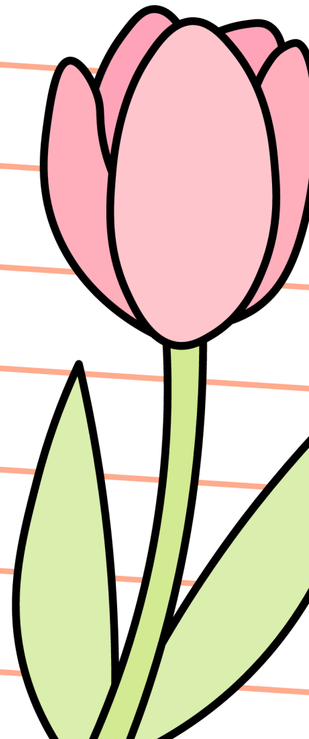
NETWORK SHORTEST PATH ALGORITHMS

- Overview:
- Problem: Finding the shortest path between two nodes in a graph.
- Applications: GPS systems, network routing, and transport logistics.
- Common Algorithms:
- Dijkstra's Algorithm
- Prim-Jarnik Algorithm

DIJKSTRA'S ALGORITHM

- Description:
- A greedy algorithm that finds the shortest path from a single source node to all other nodes in a graph.
- It maintains a set of nodes with known shortest distances.
- Time Complexity: O(V^2) for basic implementation, where V is the number of vertices.

- Steps:
- Set the distance to the source node to 0 and all other nodes to infinity.
- Mark all nodes as unvisited.
- For the current node, calculate the distance to its neighboring nodes.
- Select the unvisited node with the smallest distance, and repeat the process.



DIJKSTRA'S ALGORITHM: EXAMPLE

Example Graph:

- Node A to Node E.
- Show how the algorithm selects the shortest path from A to all other nodes.

Prim-Jarnik Algorithm

- Description:
- Prim's algorithm is used to find a Minimum Spanning Tree (MST) for a graph.
- MST is a subset of the edges that connects all the vertices with the minimum possible total edge weight.
- Time Complexity: O(V^2) for a simple adjacency matrix implementation.

- Steps:
- Start from an arbitrary vertex.
- Add the smallest edge connecting a vertex in the tree to a vertex outside.
- Repeat until all vertices are included in the tree.

Prim-Jarnik Algorithm: Example

EXAMPLE:
DISPLAY A GRAPH WITH WEIGHTED EDGES AND SHOW HOW PRIM'S ALGORITHM BUILDS THE MST.

```
void primMST(int graph[][]) {
    int parent[] = new int[V];
    int key[] = new int[V];
    Boolean mstSet[] = new Boolean[V];

    for (int i = 0; i < V; i++) {
        key[i] = Integer.MAX_VALUE;
        mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;

        for (int v = 0; v < V; v++)
            if (graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
    }
}
```

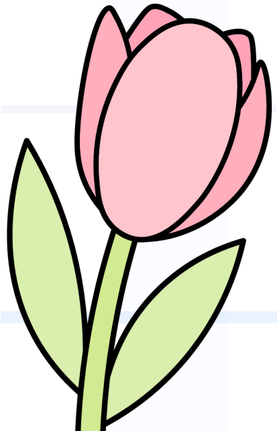
Network Algorithm Comparison

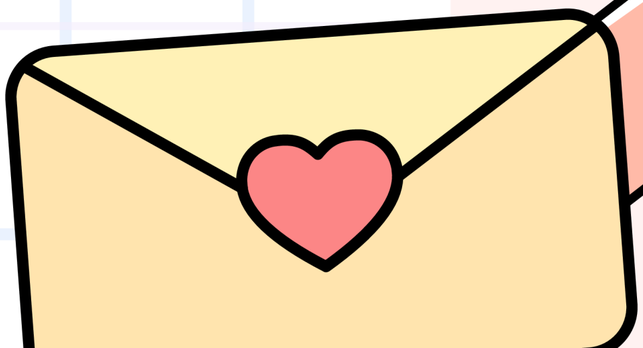
Criteria	Dijkstra's Algorithm	Prim-Jarnik Algorithm
Purpose	Shortest Path	Minimum Spanning Tree
Graph Type	Weighted, Directed/Undirected	Weighted, Undirected
Time Complexity	O(V^2)	O(V^2)
Use Case	Shortest path in a graph	Connecting all nodes with minimum cost

Performance Analysis of Shortest Path

Space Complexity: Both Dijkstra's and Prim's algorithms have space complexity of O(V) due to their usage of arrays for distances and visited nodes.

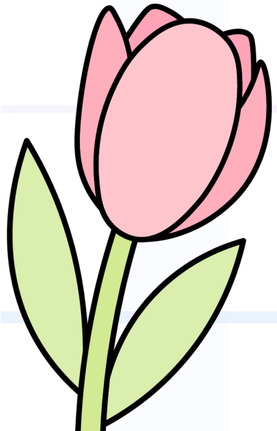
Time Complexity: Both are O(V^2) for simple adjacency matrix implementations, though Dijkstra can be optimized with a priority queue to O(E log V).

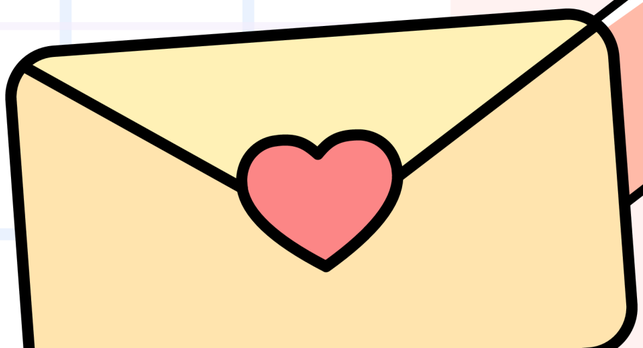




Concrete Example: Comparing Performance

- Example: Illustrate both Dijkstra and Prim using the same graph.
- Show how Dijkstra finds the shortest path, while Prim finds the minimum spanning tree.
- Compare runtimes and discuss under which conditions each algorithm is preferred.



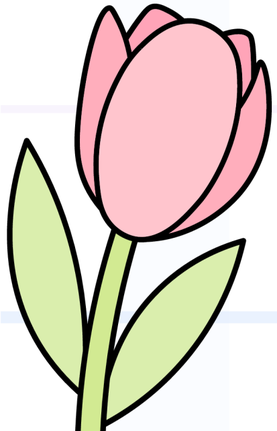


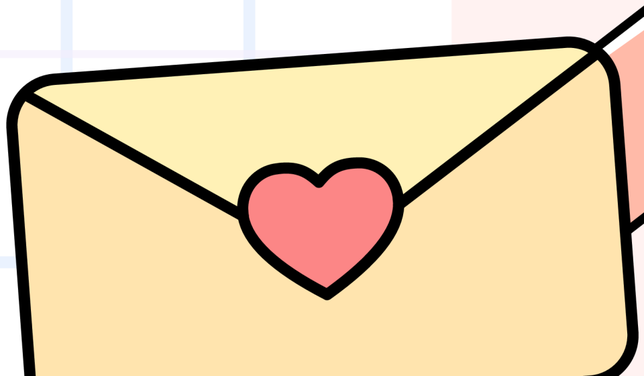
Conclusion on Data Structures and Algorithms

- Summary:
- Data structures form the backbone of efficient algorithm design.
- The correct choice of data structure and algorithm is critical for performance optimization.
- Memory management (through stacks) and sorting algorithms are foundational for performance in computing.
- Key Takeaways:
- Always consider time and space complexity when designing algorithms.
- Use appropriate data structures based on the problem at hand.

Q&A

Invite questions from the audience to clarify concepts or discuss implementations in more detail.



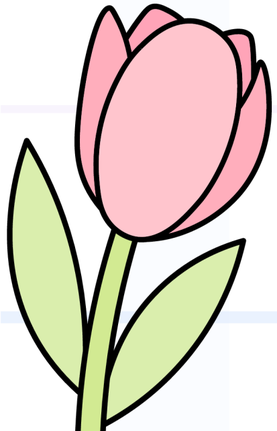


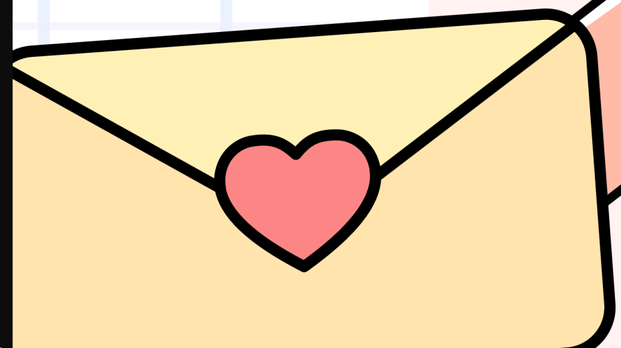
References

- Include any reference materials, books, or websites that you referred to while creating the presentation.

Example Placement of Code Snippets in Slides:

```
class LinkedQueue {
    private Node front, rear;

    public void enqueue(int item) {
        Node newNode = new Node(item);
        if (rear != null) rear.next = newNode;
        rear = newNode;
        if (front == null) front = rear;
    }

    public int dequeue() {
        if (front == null) throw new NoSuchElementException();
        int item = front.data;
        front = front.next;
        if (front == null) rear = null;
        return item;
    }
}
```

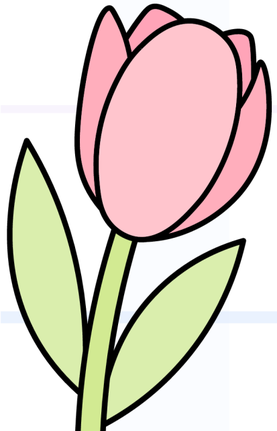


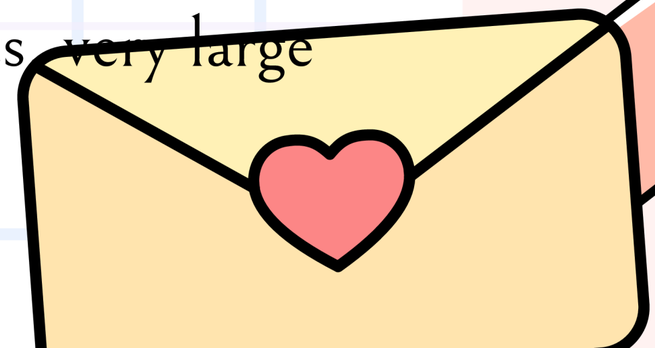


Best Practices in Algorithm Design

Key Considerations:

- Time Complexity: Always analyze the time complexity before choosing an algorithm. Aim for algorithms with lower time complexity for larger data sets.
- Space Complexity: Memory usage is critical, especially in resource-constrained environments.
- Data Structure Choice: The right data structure can significantly improve performance.
- Edge Cases: Always consider edge cases like empty inputs, very large inputs, and incorrect input formats.





Thank You