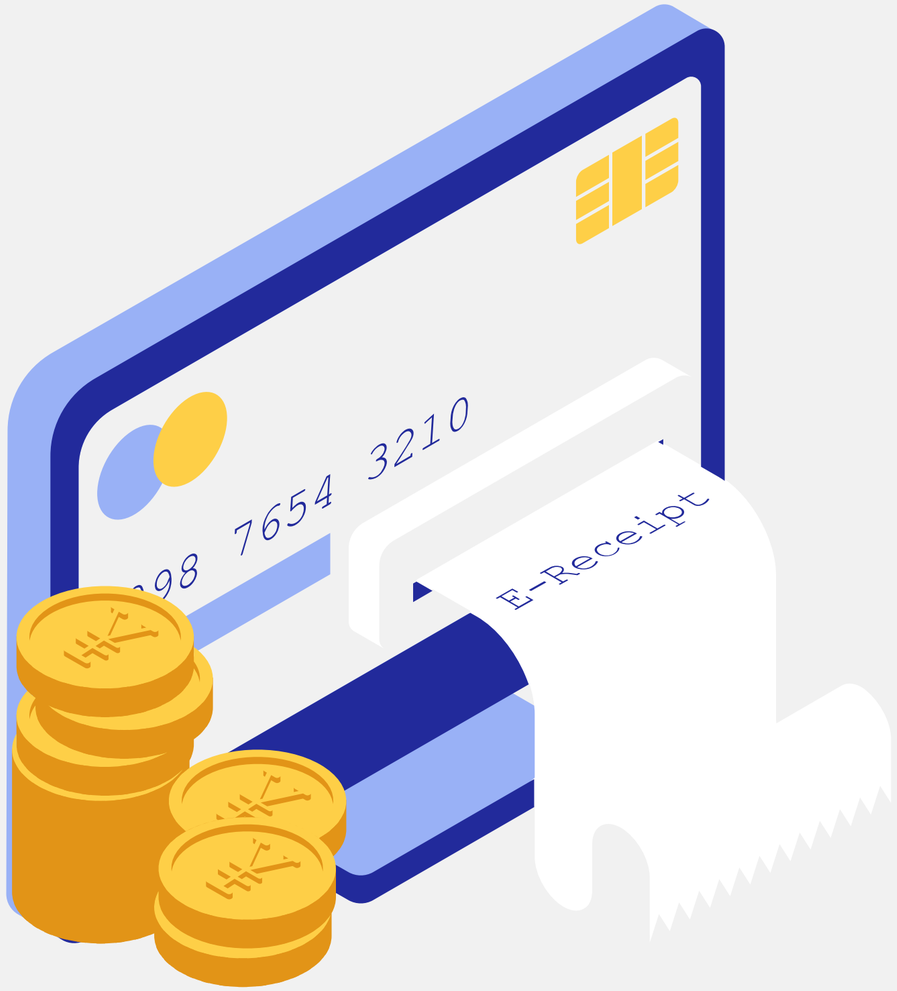


Data Structures & Algorithms

Nguyễn Đắc Huy-BH00930

Topic

1. **Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.**
2. **Determine the operations of a memory stack and how it is used to implement function calls in a computer.**
3. **Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.**
4. **Compare the performance of two sorting algorithms.**
5. **Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

01 Identify the Data Structures

02 Define the Operations

03 Specify Input Parameters

04 Define Pre- and Post-conditions

05 Discuss Time and Space Complexity

06 Provide Examples and Code Snippets

IDENTIFY THE DATA STRUCTURES

Each type of problem requires an appropriate data structure for efficient handling. Some common data structures include:

- Array: Used when quick access to elements via index is necessary, suitable for data with a fixed size.
- Linked List: Suitable for fast insertion and deletion but slower access compared to arrays.
- Stack: Follows LIFO (Last In, First Out) – the last element added is the first one to be removed.
- Queue: Follows FIFO (First In, First Out) – the first element added is the first one to be removed.
- Tree: A hierarchical structure, with the most common being the Binary Search Tree (BST) for optimizing search and sort operations.
- Hash Table: Used for fast lookups with an average time complexity of O(1) in ideal cases.
- Graph: Used for problems related to networks, shortest path finding, or connections between objects.

DEFINE THE OPERATIONS

The operations that can be performed on the data structures typically include:

- Insert: Add an element to the structure (at the beginning, end, or a specific position).
- Delete: Remove an element from the structure, which can be by value or by index.
- Update: Change the value of an existing element in the data structure.
- Search: Find a specific element based on its value or some condition.

SPECIFY INPUT PARAMETERS

Each operation on data structures requires different input parameters:

- Insert:
  - Parameter: the element to be added (could be an integer, string, or an object).
  - Example: push(5) to add the value 5 to the stack.
- Delete:
  - Parameter: the index of the element to be removed or the value of the element (depending on the structure).
  - Example: removeAt(3) to remove the element at index 3.
- Update:
  - Parameter: the position and the new value of the element.
  - Example: updateAt(2, 10) to update the element at index 2 with the new value 10.
- Search:
  - Parameter: the value to be searched.
  - Example: search(7) to check if the element with value 7 exists.

DEFINE PRE- AND POST-CONDITIONS

Pre-conditions and post-conditions ensure data integrity:

- Pre-conditions: Conditions that must be met before performing an operation, for example:
  - The array should not exceed its maximum size when adding an element.
  - The stack must have at least one element to perform the pop operation.
- Post-conditions: Conditions that ensure the data structure is in a valid state after the operation is complete:
  - After removing an element, the size of the data structure must decrease.
  - After adding an element, the element must be present in the structure at the appropriate position.

DISCUSS TIME AND SPACE COMPLEXITY

Time Complexity

- Definition: Measures how the running time of an algorithm increases with the size of the input data.
- O(1) - Constant Time: Time remains the same, regardless of input size.
- Example: Accessing the first element in an array.
- O(n) - Linear Time: Time grows directly with the data size.
- Example: Searching through all elements in a list.
- O(log n) - Logarithmic Time: Time increases slowly as data size grows.
- Example: Searching in a balanced binary search tree.
- $O(n^2)$ - Quadratic Time: Time increases dramatically with larger data sizes.
- Example: Nested loops processing n elements.

Space Complexity

- Definition: Measures the extra memory an algorithm needs as input size increases.
- O(1) - Constant Space: Fixed memory usage, regardless of input size.
- Example: Using a few variables to hold values.
- O(n) - Linear Space: Memory usage grows with input size.
- Example: Creating an array to store n elements.

PROVIDE EXAMPLES AND CODE SNIPPETS

Example: Stack Implementation

```
class Stack {
    private int[] arr;
    private int top;
    private int capacity;

    // Constructor để khởi tạo ngăn xếp
    public Stack(int size) {
        arr = new int[size];
        capacity = size;
        top = -1;
    }

    // Thêm phần tử vào ngăn xếp
    public void push(int x) {
        if (isFull()) {
            System.out.println("Stack Overflow");
            return;
        }
        arr[++top] = x;
    }

    // Xóa phần tử khỏi ngăn xếp
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack Underflow");
            return -1;
        }
        return arr[top--];
    }

    // Kiểm tra ngăn xếp có đầy không
    public boolean isFull() {
        return top == capacity - 1;
    }

    // Kiểm tra ngăn xếp có rỗng không
    public boolean isEmpty() {
        return top == -1;
    }
}
```

Determine the operations of a memory stack and how it is used to implement function calls in a computer.

01 Define a Memory Stack

02 Identify Operations

03 Function Call Implementation

04 Demonstrate Stack Frames

05 Discuss the Importance

DEFINE A MEMORY STACK

Definition

A memory stack is a data structure that operates on a Last In, First Out (LIFO) principle. It stores data in a specific order, where the last element added is the first one to be removed. In computing, the stack is used to manage function calls and local variables.

IDENTIFY OPERATIONS

Common operations performed on a memory stack include:

- Push: Add an element to the top of the stack.
  - Example: When a function is called, its parameters and local variables are pushed onto the stack.
- Pop: Remove the top element from the stack.
  - Example: When a function completes, its local variables are popped off the stack.
- Peek/Top: View the top element of the stack without removing it.
  - Example: Check the current function’s parameters before executing.
- IsEmpty: Check if the stack is empty.
  - Example: Ensure that there are no pending function calls.

FUNCTION CALL IMPLEMENTATION

The memory stack is crucial for implementing function calls in programming:

1. Function Call:
    - When a function is called, a stack frame is created, which contains:
        - The return address (where to go back after the function execution).
        - Parameters passed to the function.
        - Local variables used within the function.
    - The stack frame is pushed onto the stack.
2. Function Execution:
    - The function executes, utilizing the parameters and local variables stored in its stack frame.
3. Return from Function:
    - Once the function completes, the stack frame is popped from the stack.
    - Control returns to the address stored in the stack frame.

DEMONSTRATE STACK FRAMES

What is a Stack Frame?

- A stack frame is a block of memory allocated on the stack for a single function call. It stores everything needed for that function to execute.

Key Components of a Stack Frame:

1. Return Address:
    - Tells the program where to go back after the function finishes.
    - This is crucial for resuming execution in the calling function.
2. Parameters:
    - Values provided to the function when it is called.
    - Allow the function to perform its tasks based on the input.
3. Local Variables:
    - Variables declared inside the function.
    - Only exist while the function is running and are stored in the stack frame.

DISCUSS THE IMPORTANCE

- Memory Management: The stack efficiently manages function calls and local variables, allowing for dynamic memory allocation.
- Recursion Support: The stack enables recursive function calls by keeping track of multiple instances of function calls.
- Scope and Lifetime: Local variables only exist within their function’s stack frame, ensuring that memory is freed when the function completes.
- Performance: Stack operations (push/pop) are generally faster than heap memory allocation, contributing to efficient program execution.

Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.

01 Introduction FIFO

02 Define the Structure

03 Array-Based Implementation

04 Linked List-Based Implementation

05 Provide a concrete example to illustrate how the FIFO queue works

Introduction FIFO

Definition of FIFO

FIFO (First In, First Out) is a data structure principle in which the first element added to the queue is the first one to be removed. This ensures that items are processed in the same order they were received, maintaining a fair and orderly flow of data.

Main Features:

- Order Matters: Items are handled in the order they come in.
- Equal Treatment: All items are treated the same; none are given priority.
- Flexible Size: The queue can grow or shrink as items are added or removed.
- Common Uses:
- Task Management: Computers use FIFO to manage tasks fairly.
- Data Storage: Used in situations like printing documents or streaming data.
- Graph Searches: Helps in algorithms that explore data step by step.

DEFINE THE STRUCTURE OF FIFO QUEUE

Basic Components:

- Elements: The items or data stored in the queue.
- Front: The position from which elements are removed (dequeued).
- Rear: The position where elements are added (enqueued).

Operations:

- Enqueue: Adds an element to the rear of the queue.
- Dequeue: Removes an element from the front of the queue.
- Peek/Front: Returns the front element without removing it.
- IsEmpty: Checks if the queue contains any elements.

Structure Types:

- Array-Based Implementation:
- Uses an array to store elements, with pointers for front and rear.
- Linked List-Based Implementation:
- Uses nodes linked together, with pointers for the front and rear nodes.

Array-Based Implementation of FIFO Queue

Structure:

- Uses a fixed-size array to store elements of the queue.
- Two pointers (or indices) are maintained:
- front: Points to the index of the first element in the queue.
- rear: Points to the index of the last element added to the queue.
- Key Operations:

Enqueue:

- Adds an element at the rear position.
- Increments the rear pointer (wrapping around to the beginning of the array if necessary).

Dequeue:

- Removes the element at the front position.
- Increments the front pointer (also wrapping around if needed).
- Peek/Front:
- Returns the element at the front without removing it.
- IsEmpty:
- Checks if the size of the queue is zero.

Example Code (Pseudo-Code):

```
class Queue {
    int[] arr;   // Array to store queue elements
    int front;   // Index of the front element
    int rear;    // Index of the last element
    int size;    // Number of elements in the queue

    Queue(int capacity) {
        arr = new int[capacity]; // Create an array with the given capacity
        front = 0;               // Initialize front index
        rear = -1;               // Initialize rear index
        size = 0;                // Initialize size
    }

    void enqueue(int item) {
        if (size < arr.length) {
            rear = (rear + 1) % arr.length; // Circular increment of rear
            arr[rear] = item;               // Add item to the queue
            size++;                         // Increase the size
        }
    }

    int dequeue() {
        if (size > 0) {
            int item = arr[front];          // Get the front item
            front = (front + 1) % arr.length; // Circular increment of front
            size--;                         // Decrease the size
            return item;                    // Return the dequeued item
        }
        return -1; // Queue is empty
    }
}
```

Advantages:

- Fast Access: O(1) time complexity for both enqueue and dequeue operations.
- Memory Efficiency: Memory is allocated in a contiguous block, minimizing overhead.

Disadvantages:

- Fixed Size: The size of the queue must be predetermined, which may lead to overflow if not managed properly.
- Wasted Space: If elements are dequeued, space may be wasted as it cannot be reused until the queue is reset.

Linked List-Based Implementation of FIFO Queue

Structure:

- Utilizes a linked list where each element is represented by a node.
- Node Structure:
- Data: The value of the element.
- Next: A pointer/reference to the next node in the queue.
- Two pointers are maintained:
- front: Points to the first node in the queue (the node to be dequeued).
- rear: Points to the last node in the queue (the node to which new elements are added).
- Key Operations:

Enqueue:

- Create a new node for the element.
- If the queue is empty, set both front and rear to the new node.
- Otherwise, link the new node to the end of the list and update the rear pointer to the new node.

Dequeue:

- Remove the node at the front position.
- Update the front pointer to the next node in the list.
- If the queue becomes empty after the operation, set rear to null.
- Peek/Front:
- Returns the value of the node at the front without removing it.
- IsEmpty:
- Checks if front is null (indicating that the queue is empty).

Example Code (Pseudo-Code):

```
class Node {
    int data;      // Data held by the node
    Node next;     // Pointer to the next node

    Node(int data) {
        this.data = data; // Initialize node with data
        this.next = null; // Next is initially null
    }
}

class Queue {
    Node front; // Pointer to the front node
    Node rear;  // Pointer to the rear node

    Queue() {
        front = null; // Initialize front
        rear = null;  // Initialize rear
    }

    void enqueue(int item) {
        Node newNode = new Node(item); // Create a new node
        if (rear == null) { // If the queue is empty
            front = rear = newNode; // Both front and rear point to new node
        } else {
            rear.next = newNode; // Link new node at the end
            rear = newNode;      // Update rear to new node
        }
    }

    int dequeue() {
        if (front == null) return -1; // Queue is empty
        int item = front.data;        // Get the front item
        front = front.next;           // Move front to the next node
        if (front == null) rear = null; // If queue is now empty
        return item;                  // Return dequeued item
    }
}
```

Linked List-Based Implementation of FIFO Queue

Advantages:

- Dynamic Size: Can grow and shrink as needed; no fixed size limitation.
- Efficient Memory Usage: No wasted space; memory is allocated as needed.

Disadvantages:

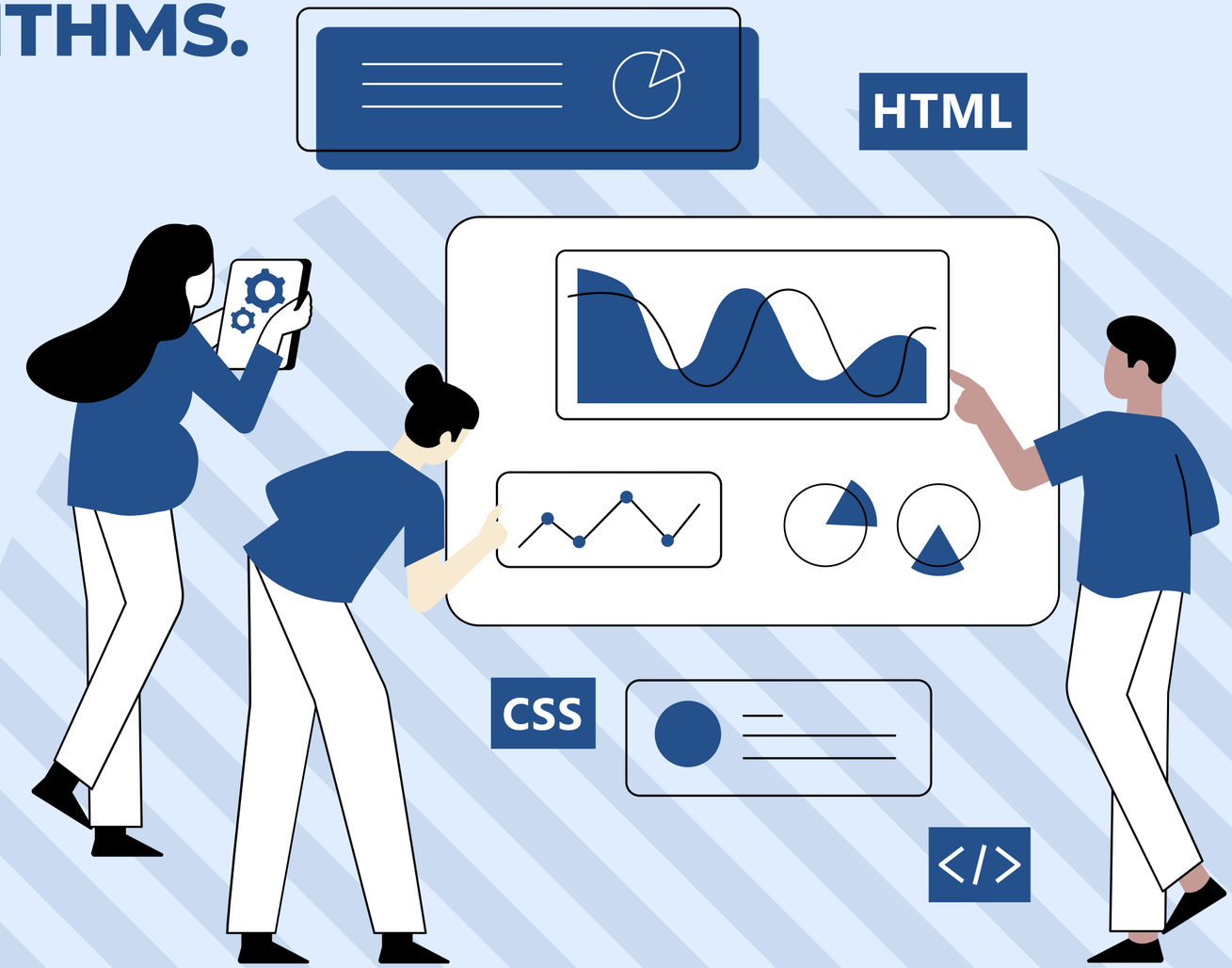
- Overhead: Each node requires extra memory for storing the pointer/reference.
- Access Time: Slightly slower access compared to array-based due to pointer traversal.

Provide a concrete example to illustrate how the FIFO queue works

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

public class FIFOQueue {
    private Queue<String> queue; // Using a LinkedList to implement the Queue

    public FIFOQueue() {
        queue = new LinkedList<>(); // Initialize the queue
    }

    // Method to add a print job to the queue
    public void enqueue(String job) {
        queue.add(job);
        System.out.println("Enqueued: " + job);
    }

    // Method to process and remove a print job from the queue
    public void dequeue() {
        if (!isEmpty()) {
            String job = queue.poll(); // Remove and retrieve the head of the queue
            System.out.println("Dequeued: " + job);
        } else {
            System.out.println("Queue is empty!");
        }
    }
```

```java
    // Method to check if the queue is empty
    public boolean isEmpty() {
        return queue.isEmpty();
    }

    // Method to get the size of the queue
    public int size() {
        return queue.size();
    }

    public static void main(String[] args) {
        FIFOQueue printQueue = new FIFOQueue();

        // Enqueue print jobs
        printQueue.enqueue("Document A");
        printQueue.enqueue("Document B");
        printQueue.enqueue("Document C");

        // Dequeue print jobs
        printQueue.dequeue(); // Should process Document A
        printQueue.dequeue(); // Should process Document B
        printQueue.dequeue(); // Should process Document C
        printQueue.dequeue(); // Queue is empty
    }
}
```

COMPARE THE PERFORMANCE OF TWO SORTING ALGORITHMS.

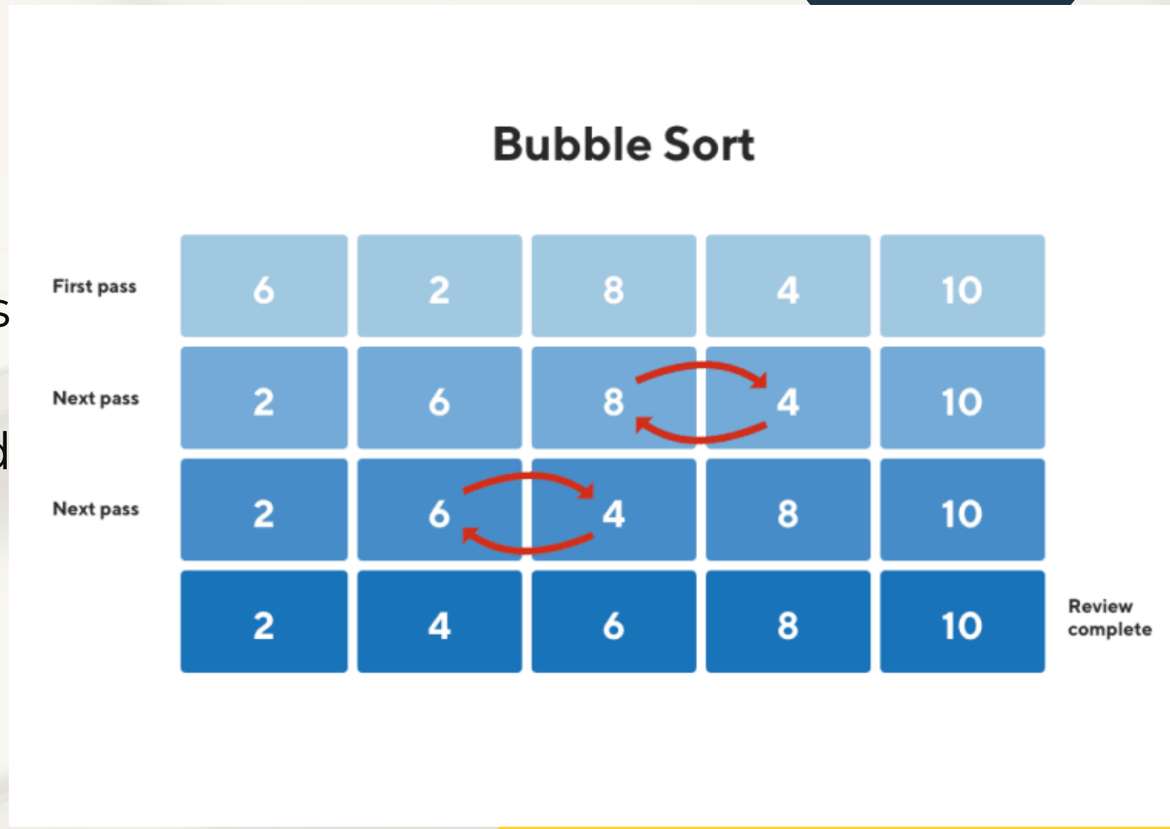


Introducing the Two Sorting Algorithms

Bubble Sort:

Overview: Bubble Sort is a simple and intuitive sorting algorithm that repeatedly compares adjacent elements in a list and swaps them if they are in the wrong order. This process is repeated until no swaps are needed, indicating that the list is sorted. Although easy to implement, Bubble Sort is inefficient for larger datasets due to its average and worst-case time complexities of $O(n^2)$cteristics:

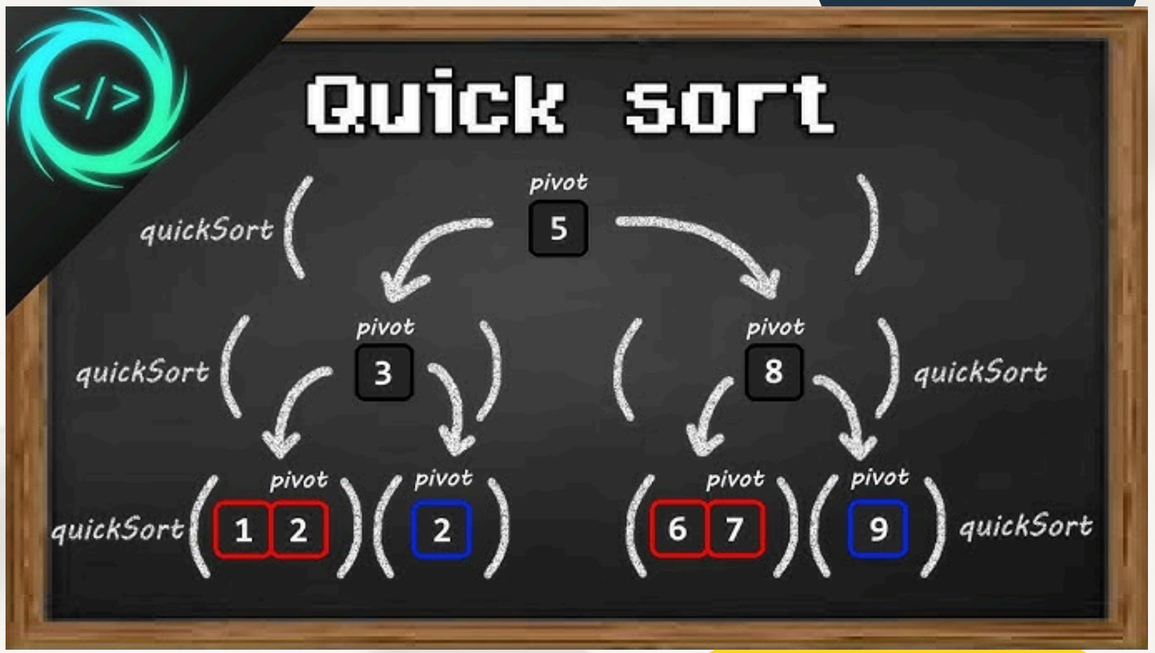
- Stability: Bubble Sort is a stable sorting algorithm, meaning that it preserves the relative order of records with equal keys.
- Space Complexity: It operates in O(1) space, making it an in-place sorting algorithm .



Introducing the Two Sorting Algorithms

Quick Sow: Quick Sort is a highly efficient sorting algorithm that uses a divide-and-conquer approach. It works by selecting a "pivot" element and partitioning the array into two sub-arrays: elements less than the pivot and elements greater than the pivot. Quick Sort is generally faster than Bubble Sort for larger datasets, with an average-case time complexity of O(n log n) .

- Key Characteristics: Quick Sort is not stable; equal elements may not retain their relative order after sorting.
    - Space Complexity: It has a space complexity of O(log n) due to the recursive stack used for the algorithm .
-



Time Complexity Analysis

Bubble Sort

Best Case: O(n)

- The best-case scenario occurs when the array is already sorted. In this case, the algorithm makes one pass through the array to check for any swaps and finds none, resulting in linear time complexity.

Average Case: $O(n^2)$

- On average, the algorithm performs approximately n/2 comparisons for each of the n elements. This leads to a time complexity of $O(n^2)$, making Bubble Sort inefficient for large datasets.

Worst Case: $O(n^2)$

- The worst-case scenario happens when the array is sorted in reverse order, requiring the maximum number of swaps and comparisons.

Quick Sort

Best Case: O(n log n)

- The best-case time complexity occurs when the pivot divides the array into two nearly equal halves at every level of recursion. This results in a balanced partitioning and a logarithmic depth of recursion.

Average Case: O(n log n)

- The average-case complexity also falls under O(n log n) due to the same balanced partitioning across a random dataset. Quick Sort is efficient in practice for large datasets.

Worst Case: $O(n^2)$

- The worst-case occurs when the smallest or largest element is consistently chosen as the pivot, leading to highly unbalanced partitions. This scenario can degrade the performance significantly, resulting in a quadratic time complexity.

Space Complexity Analysis

Bubble Sort

- Space Complexity: O(1)
  - Bubble Sort is an in-place sorting algorithm, meaning it requires a constant amount of additional space regardless of the input size. It only uses a small, fixed amount of space for temporary variables (like temp for swapping) during the sorting process. This makes Bubble Sort memory-efficient, although it is not optimal for speed.

Quick Sort

Space Complexity: O(log n) (in the average case)

- Quick Sort is also an in-place sorting algorithm, but it uses a stack for recursive function calls. The maximum depth of this stack is determined by the height of the recursion tree, which is logarithmic when the pivot divides the array into balanced partitions.
- Worst Case: O(n)
  - In the worst-case scenario (e.g., when the pivot is the smallest or largest element), Quick Sort can lead to unbalanced partitions, causing the recursion tree to have a height of n, resulting in O(n) space complexity due to the recursion stack.

Stability

Bubble Sort

- Stability: Bubble Sort is a stable sorting algorithm. This means that when two elements have equal keys, their relative order remains unchanged in the sorted output. For example, if two elements with the same value appear in the list, they will retain their original order after the sorting process.
- Importance of Stability: Stability is particularly important in scenarios where the original order of records carries significance. For instance, when sorting a list of students by grades, if two students have the same grade, maintaining their original order (perhaps based on their names or IDs) can be crucial.

Quick Sort

- Stability: Quick Sort is not a stable sorting algorithm. During the partitioning process, equal elements may be rearranged in a way that changes their original order. For example, if two identical elements are placed on different sides of the pivot during partitioning, their relative order will not be preserved.
- Consequences of Instability: The instability of Quick Sort can be a drawback in situations where the order of equal elements matters. In applications like sorting records with multiple fields (e.g., sorting employees first by department and then by name), a stable sort would be required to maintain the integrity of the sorting criteria.

Comparison Table

Feature	Bubble Sort	Quick Sort
Algorithm Type	Comparison-based	Divide-and-conquer
Time Complexity	Best: O(n) Average: O(n²) Worst: O(n²)	Best: O(n log n) Average: O(n log n) Worst: O(n²)
Space Complexity	O(1) (in-place)	Average: O(log n) Worst: O(n)
Stability	Stable	Not stable
In-Place	Yes	Yes
Best Use Cases	Small datasets, educational purposes	Larger datasets, performance-critical applications
Performance on Sorted Input	Excellent (O(n))	Average (O(n log n))

Performance Comparison

Bubble Sort

Time Complexity

- Bubble Sort:
  - Best Case: O(n) — This occurs when the array is already sorted, requiring only one pass through the data.
  - Average Case: $O(n^2)$ — Most datasets will require multiple passes, making it inefficient for larger arrays.
  - Worst Case: $O(n^2)$ — Happens when the array is sorted in reverse order.

Bubble Sort: O(1) — It uses a constant amount of additional memory since it sorts in place.

Bubble Sort: Due to its $O(n^2)$ complexity, Bubble Sort is rarely used in practical applications beyond educational contexts or very small datasets.

Quick Sort

Quick Sort:

- Best Case: O(n log n) — When the pivot divides the array into two equal halves, leading to balanced partitions.
- Average Case: O(n log n) — Generally efficient for random datasets.
- Worst Case: $O(n^2)$ — Occurs when the pivot is the smallest or largest element, leading to unbalanced partitions.

Quick Sort:

- Average Case: O(log n) — Due to recursive calls.
- Worst Case: O(n) — Can occur in unbalanced partitions.

Quick Sort: Known for its efficiency and performance in real-world applications, Quick Sort is often the go-to algorithm for large datasets due to its average-case performance.

Provide a concrete example to demonstrate the differences in performance between the two algorithms

```
Array: [64, 34, 25, 12, 22, 11, 90]
```

Bubble Sort

```java
void bubbleSort(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Quick Sort

```java
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);
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            // swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // swap arr[i + 1] and arr[high] (or pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}
```

Provide a concrete example to demonstrate the differences in performance between the two algorithms

Performance Measurement

To measure the performance of these two sorting algorithms, we can use the following criteria:

- Execution Time: Measure how long each algorithm takes to sort the array.
- Number of Comparisons: Count the comparisons made by each algorithm.

Expected Results:

- For the above array:
  - Bubble Sort is expected to take significantly longer and make more comparisons due to its $O(n^2)$ time complexity.
  - Quick Sort, with its O(n log n) time complexity, should perform much faster, especially as the array size increases.

Concrete Example Results

1. Bubble Sort might take several seconds to sort an array with 10,000 elements.
2. Quick Sort could sort the same array in milliseconds.

Conclusion

This example illustrates the performance differences between Bubble Sort and Quick Sort. While Bubble Sort is easier to implement, its inefficiency becomes evident with larger datasets. Quick Sort is typically preferred in real-world applications due to its superior performance.

ANALYSE THE OPERATION, USING ILLUSTRATIONS, OF TWO NETWORK SHORTEST PATH ALGORITHMS, PROVIDING AN EXAMPLE OF EACH.

Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

Network shortest path algorithms are fundamental in graph theory and computer science, used to find the shortest paths between nodes in a graph. These algorithms have numerous applications, including routing in networks, geographical mapping, and logistical planning.

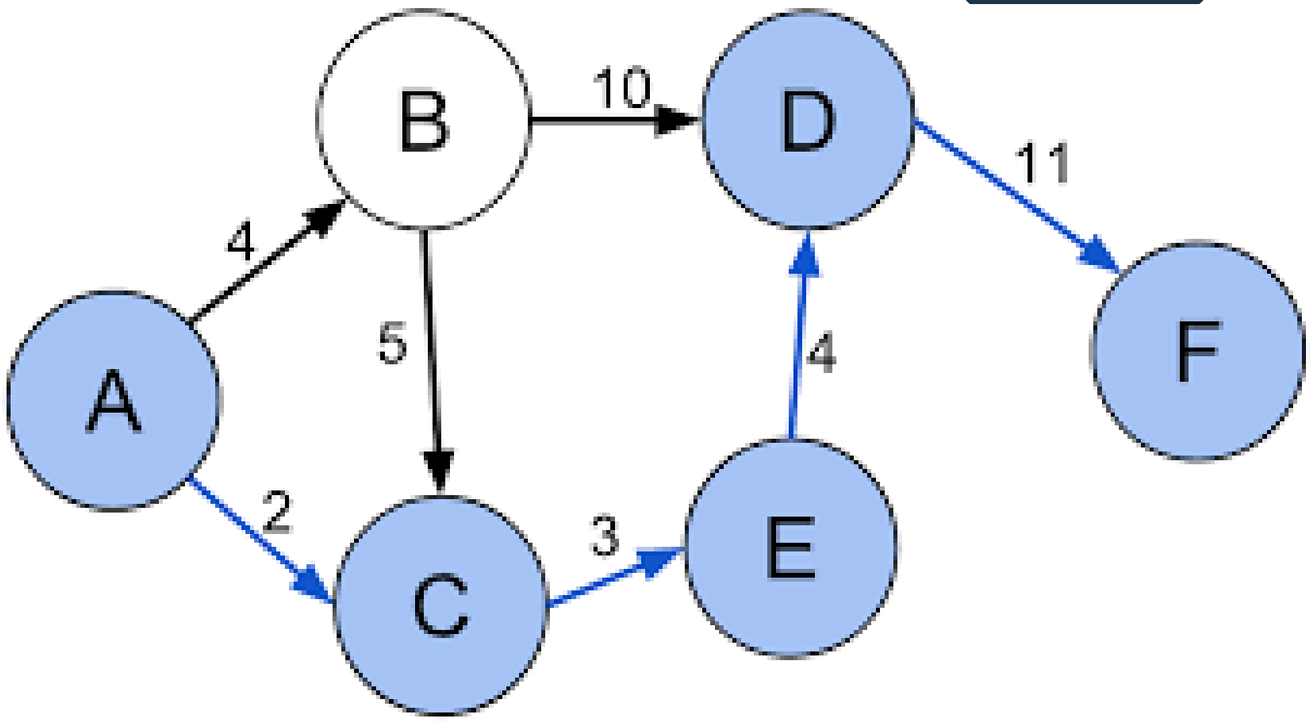
Key Concepts

- Graphs:

A graph is a collection of nodes (or vertices) connected by edges. In a weighted graph, edges have weights that represent the cost or distance to traverse between nodes.

- Shortest Path:

The shortest path refers to the minimum weight path between two nodes in a graph. Finding this path is crucial for optimizing routes and reducing costs in various applications.

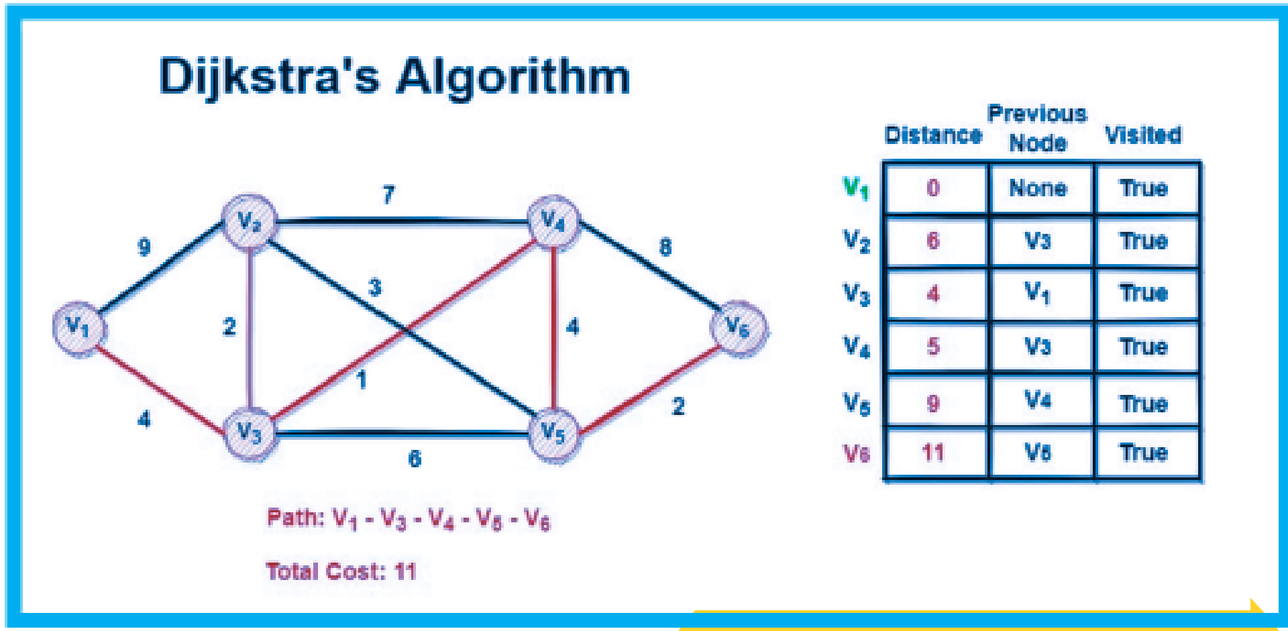


Algorithm 1: Dijkstra's Algorithm

Dijkstra's Algorithm is a well-known algorithm used to find the shortest paths between nodes in a weighted graph, where all edge weights are non-negative. It is particularly effective for applications in routing and navigation.

How It Works

1. Initialization:
   - Start by assigning a tentative distance value to every node in the graph. Set the initial node's distance to zero and all others to infinity. This indicates that the initial node is the starting point, and all other nodes are unreachable at the beginning.
2. Priority Queue:
   - Use a priority queue (or a min-heap) to keep track of nodes with their current shortest distance. The node with the smallest tentative distance is processed first.
3. Main Loop:
   - While there are nodes to process:
     - Dequeue the node with the smallest distance from the priority queue.
     - For each unvisited neighbor of this node, calculate the distance from the starting node to that neighbor through the current node. If this distance is less than the currently recorded distance, update it.
     - Mark the node as visited when all neighbors have been considered, preventing it from being processed again.
4. Termination:
   - The algorithm terminates when all nodes have been visited, and the shortest path to each node is determined.



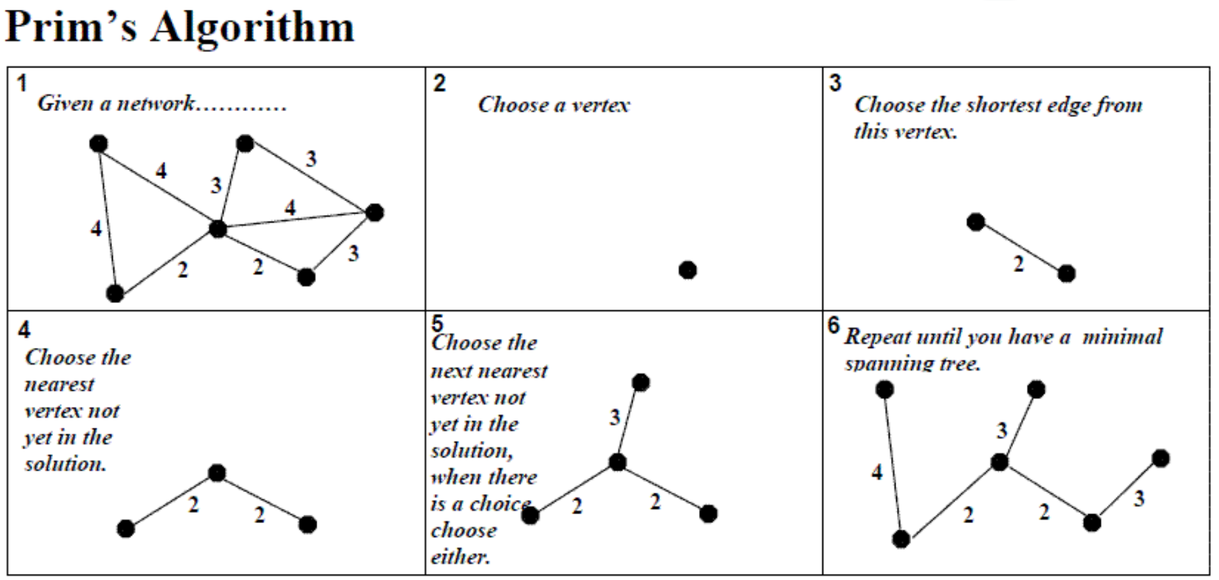
	Distance	Previous Node	Visited
V_1	0	None	True
V_2	6	V3	True
V_3	4	V_1	True
V_4	5	V3	True
V_5	9	V4	True
V_6	11	V5	True

Algorithm 2: Prim-Jarnik Algorithm

The Prim-Jarnik Algorithm, commonly referred to simply as Prim's Algorithm, is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a weighted undirected graph. It efficiently connects all vertices with the least total edge weight, making it useful in various applications like network design, clustering, and more.

How It Works

1. Initialization:
   - Start with a single vertex, marking it as part of the MST. Initialize a priority queue (or a min-heap) to keep track of the edges connecting the vertices in the MST to those not yet included.
2. Select Minimum Edge:
   - Repeatedly select the minimum-weight edge from the priority queue that connects a vertex in the MST to a vertex outside the MST.
3. Update the MST:
   - Add the selected edge and the new vertex to the MST. Then, update the priority queue with the edges connecting the newly added vertex to the remaining vertices not in the MST.
4. Termination:
   - Continue the process until all vertices are included in the MST.



Time Complexity Analysis

Dijkstra's Algorithm

Time Complexity:

- The time complexity of Dijkstra's Algorithm depends on the implementation:
  - Using an array: $O(V^2)$, where V is the number of vertices.
  - Using a priority queue (min-heap): $O((V + E) \log V)$, which is more efficient for sparse graphs (where E is the number of edges).

Space Complexity:

- Space complexity is O(V) for storing the distance and predecessor arrays.

Performance:

- Dijkstra's is highly efficient for graphs with non-negative weights. However, it does not handle negative edge weights, which can lead to incorrect results.
- Practical applications include routing protocols (like OSPF) and geographical mapping (like Google Maps).

Prim-Jarnik Algorithm

Time Complexity:

- Similar to Dijkstra's, Prim's Algorithm also benefits from using a priority queue:
  - Using an array: $O(V^2)$.
  - Using a priority queue (min-heap): $O(E \log V)$.
-

Space Complexity:

- Space complexity is also O(V) due to the storage of vertex states and edge weights.
-

Performance:

- Prim's is particularly effective for dense graphs where the number of edges is close to the maximum possible ($E = V^2$).
- It can also handle graphs with zero-weight edges efficiently.
- Applications include network design, such as minimizing costs in telecommunications and electricity distribution.

THANK YOU SO MUCH!