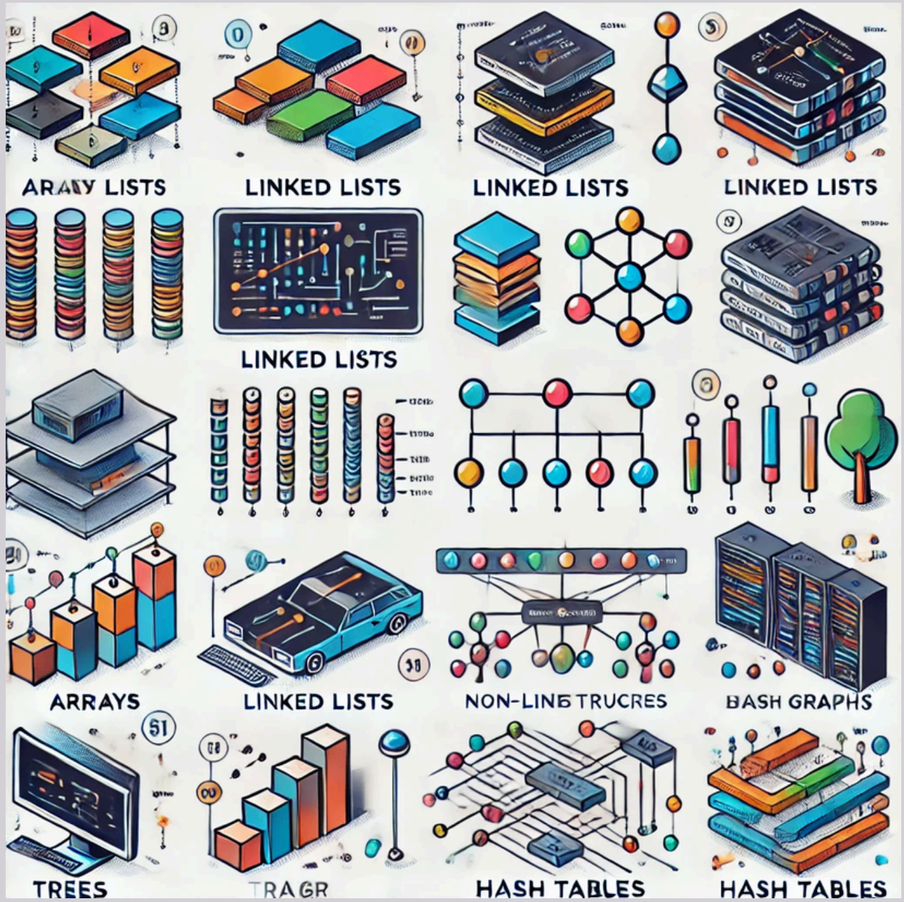
Design and Analysis of Data Structures and Algorithms

By Phan Bao Trung

Table of Contents

Introduction to Data Structures

What are Data Structures?
Data structures are a way of organizing and storing data in a computer so it can be accessed and modified efficiently.
Why Are They Important?
Efficient data management
Optimized algorithm performance
Key to software design and implementation
Types of Data Structures:
Linear Structures: Arrays, Linked Lists, Stacks, Queues
Non-Linear Structures: Trees, Graphs, Hash Tables



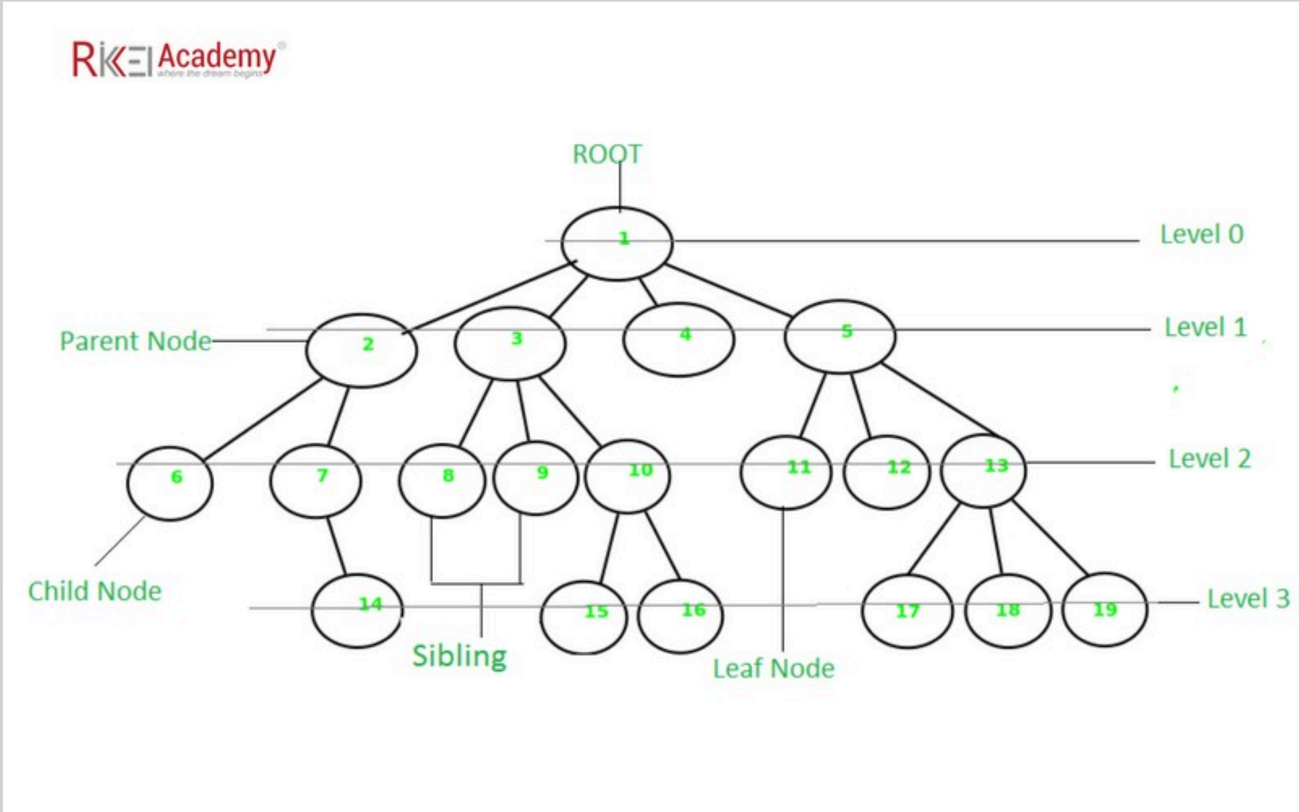
Identify the Data Structures

Linear Data Structures

- Array: A collection of elements stored at contiguous memory locations.
- Linked List: A linear collection of data elements where each element points to the next.
- Stack: A Last In First Out (LIFO) data structure.
- Queue: A First In First Out (FIFO) data structure.

Non-Linear Data Structures

- Tree: A hierarchical structure with a root and subtrees of children.
- Graph: A collection of nodes connected by edges.
- Hash Table: A structure that maps keys to values for efficient lookup.

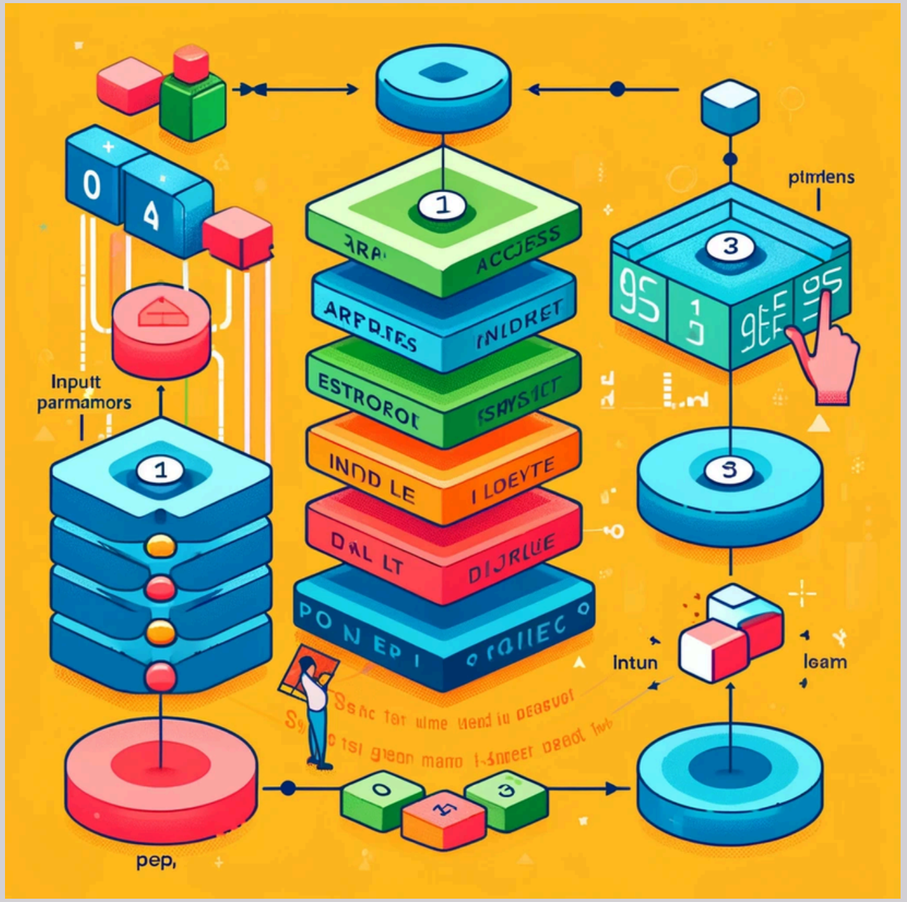


Define the Operations

- Array Operations
  - Access: Retrieve an element at a specific index.
  - Insert: Add an element at a given position.
  - Delete: Remove an element from a specific position.
- Linked List Operations
  - Traversal: Move through each node in the list.
  - Insertion/Deletion: Add or remove nodes at the beginning, middle, or end.
- Stack Operations
  - Push: Add an element to the top of the stack.
  - Pop: Remove the top element.
  - Peek: Retrieve the top element without removing it.
- Queue Operations
  - Enqueue: Add an element to the back of the queue.
  - Dequeue: Remove an element from the front.

Specify Input Parameters

- Array Operations:
- Access/Insert/Delete:
- Input: Index (position), Element (for insert).
- Linked List Operations:
- Traversal/Insert/Delete:
- Input: Node position (head, middle, or tail), Element (for insert).
- Stack Operations:
- Push/Pop:
- Input: Element (for push).
- Queue Operations:
- Enqueue/Dequeue:
- Input: Element (for enqueue).



Define Pre- and Post-conditions

- Array Operations:
- Pre-condition: Valid index within array bounds.
- Post-condition: Element added/removed at specified index.
- Linked List Operations:
- Pre-condition: List must not be empty for deletion.
- Post-condition: Node successfully inserted/removed.
- Stack Operations:
- Pre-condition: Stack must not be full (for push).
- Post-condition: Element pushed onto the stack or popped from it.
- Queue Operations:
- Pre-condition: Queue must not be full (for enqueue).
- Post-condition: Element enqueued or dequeued successfully.

Time and Space Complexity

- **Array:**
- Time Complexity:
- Access: O(1)O(1)O(1) - Direct access to any element is very fast because arrays are stored in contiguous memory locations.
- Insertion/Deletion: O(n)O(n)O(n) - Inserting or deleting an element requires shifting elements, which can take time proportional to the array size.
- Space Complexity: O(n)O(n)O(n) - The array needs space to store nnn elements.
- **Linked List:**
- Time Complexity:
- Access: O(n)O(n)O(n) - To access an element, you may need to traverse up to nnn nodes in the list.
- Insertion/Deletion: O(1)O(1)O(1) - Adding or removing a node is efficient if you already have a reference to the position.
- Space Complexity: O(n)O(n)O(n) - Needs space for nnn nodes.
- **Stack:**
- Time Complexity:
- Push/Pop: O(1)O(1)O(1) - Adding or removing elements (push and pop) only happens at the top, making these operations constant time.
- Space Complexity: O(n)O(n)O(n) - Needs space to store nnn elements in the stack.
- **Queue:**
- Time Complexity:
- Enqueue/Dequeue: O(1)O(1)O(1) - Inserting (enqueue) at the back and removing (dequeue) from the front are both efficient operations.
- Space Complexity: O(n)O(n)O(n) - Needs space to store nnn elements in the queue.

Examples and Code Snippets

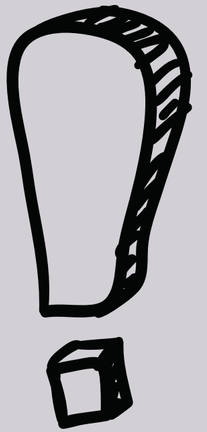
```
public class w {
    class ArrayStack {
        private int maxSize;
        private int[] stackArray;
        private int top;

        public ArrayStack(int size) {
            maxSize = size;
            stackArray = new int[maxSize];
            top = -1;
        }

        public void push(int value) {
            if (top < maxSize - 1) {
                stackArray[++top] = value;
            }
        }

        public int pop() {
            return stackArray[top--];
        }

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

}
```

Linked List Implementation:

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

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

    class LinkedListStack {
        private Node top;

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

        public int pop() {
            int value = top.data;
            top = top.next;
            return value;
        }

        public boolean isEmpty() {
            return (top == null);
```

Memory Stack and Function Calls

- What is a Memory Stack?
- A memory stack is a data structure used in computers to store information about the active subroutines or functions of a program.
- It helps manage function calls, keeping track of return addresses, local variables, and the function's execution state.
- Operations of a Memory Stack:
- Push: Store information (e.g., return address, local variables) when a function is called.
- Pop: Remove information when the function execution is complete.
- Peek: View the current state of the stack.
- Stack Frames
- Each function call creates a stack frame that holds:
- Function parameters
- Local variables
- Return address



Importance
Efficiently manages function calls.
Supports recursion by keeping track of multiple function instances.

Function Call Implementation Using Memory Stack

- How Function Calls Work
- Function Call:
- When a function is called, a stack frame is created on the memory stack.
- The frame includes:
- Return address: where the control goes after function execution.
- Parameters passed to the function.
- Local variables.
- Function Execution:
- The function runs using the local variables and parameters stored in the stack frame.
- Function Return:
- Once the function finishes, the stack frame is popped from the memory stack.
- The control returns to the return address.

- Stack Frames Example:
- Main() calls Func1(), then Func1() calls Func2().
- The stack will look like:
- Top: Func2's frame
- Below: Func1's frame
- Bottom: Main's frame
- Key Takeaway:
- The memory stack efficiently manages the execution flow of function calls, especially in nested or recursive calls.

Demonstrating Stack Frames

- What is a Stack Frame?
- A stack frame is a section of the memory stack used to manage function execution.
- It stores:
- Return address: Where to return after the function completes.
- Parameters: Values passed to the function.
- Local variables: Variables declared within the function

Example: Stack Frame in Action

```
public class w {

    Run | Debug
    public static void main(String[] args) {
        funcA();  // Stack frame for funcA is created
    }


    public static void funcA() {
        funcB();  // Stack frame for funcB is created
    }


    public static void funcB() {
        int x = 10;  // Local variable in funcB's stack frame
    }


}
```

- Stack Frame Visualisation:
- Main() calls funcA() – Stack frame for funcA.
- funcA() calls funcB() – Stack frame for funcB on top of funcA.
- funcB() completes – Stack frame for funcB is popped off the stack.
- Takeaway:
- Each function call creates a new stack frame, and once the function completes, its frame is removed. This process ensures efficient function call management.

Importance of Stack Frames

- Efficient Function Call Management
- Stack frames provide an organized way to manage function calls.
- They maintain control over:
- Return addresses: Where to resume execution after a function completes.
- Local variables: Variables scoped to each function call.
- 2. Memory Management
- By using stack frames, memory is automatically allocated and deallocated.
- When a function call ends, the stack frame is removed, freeing up memory.
- 3. Recursion Handling
- Stack frames are essential for recursive function calls.
- Each recursive call creates a new stack frame, maintaining a clear separation between different instances of the same function.
- Debugging and Error Handling
- Stack frames help in tracing function calls during debugging.
- Tools like stack traces allow developers to locate the exact point where an error occurred by showing the sequence of stack frames.
- Key Takeaway:
- Stack frames are vital for managing function execution, ensuring proper memory allocation, and supporting recursion and debugging.

FIFO Queue: Concrete Example

1. Introduction to FIFO
   - FIFO (First In, First Out) is a queue where the first element added is the first to be removed.
   - Commonly used in scheduling tasks, buffering, and managing processes in operating systems.
2. Array-Based Implementation
   - Array can be used to represent a queue.
   - Operations:
   - Enqueue: Add an element to the end of the array.
   - Dequeue: Remove the element from the front.

```
public class w {
class Queue {
    int[] arr;
    int front, rear, size;

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

    public void enqueue(int value) {
        if (size == arr.length) {
            System.out.println(x:"Queue is full");
            return;
        }
        rear = (rear + 1) % arr.length;
        arr[rear] = value;
        size++;
    }
    public int dequeue() {
        if (size == 0) {
            throw new RuntimeException(message:"Queue is empty");
        }
        int value = arr[front];
        front = (front + 1) % arr.length;
        size--;
        return value;
    }
}
```

FIFO Queue: Concrete Example

3. Linked List-Based Implementation

- A Linked List provides dynamic memory allocation, ideal when the queue size isn't fixed.
- Operations:
  - Enqueue: Add a new node at the end.
  - Dequeue: Remove the node from the front.

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

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

    class QueueLinkedList {
        Node front, rear;

        public QueueLinkedList() {
            front = rear = null;
        }
```

```
    public void enqueue(int data) {
        Node newNode = new Node(data);
        if (rear == null) {
            front = rear = newNode;
            return;
        }
        rear.next = newNode;
        rear = newNode;
    }

    public int dequeue() {
        if (front == null) {
            throw new RuntimeException(message:"Queue is empty");
        }
        int value = front.data;
        front = front.next;
        return value;
    }
}
```

Comparison of Two Sorting Algorithms: Bubble Sort vs. Merge Sort

1. Introducing the Algorithms
   - Bubble Sort:
   - Simple comparison-based algorithm.
   - Swaps adjacent elements if they are in the wrong order.
   - Merge Sort:
   - Divide and conquer algorithm.
   - Recursively splits the array into halves, sorts, and merges them.
2. Time Complexity Analysis
   - Bubble Sort:
   - Worst-case: $O(n^2)$
   - Best-case: O(n) (when the array is already sorted)
   - Merge Sort:
   - Worst-case and best-case: O(n log n)
3. Space Complexity Analysis
   - Bubble Sort: O(1) (in-place sorting)
   - Merge Sort: O(n) (additional space for merging arrays)
4. Stability
   - Both Bubble Sort and Merge Sortare stable sorting algorithms.
   - Stability means that the relative order of equal elements is preserved.
5. Comparison Table

Algorithm	Time Complexity (Worst)	Space Complexity	Stability
Bubble Sort	$O(n^2)$	O(1)	Yes
Merge Sort	O(n log n)	O(n)	Yes

6. Performance Comparison
   - Bubble Sort is inefficient for large datasets due to its quadratic time complexity.
   - Merge Sort is more efficient for larger datasets, but requires extra space.
7. Concrete Example
   - Sorting an Array: [4, 3, 2, 1]
   - Bubble Sort: Multiple iterations with many swaps.
   - Merge Sort: Array is divided, sorted, and merged in fewer steps.

Comparison of Two Sorting Algorithms: Bubble Sort vs. Merge Sort

1. Introducing the Algorithms
   - Bubble Sort:
   - Simple comparison-based algorithm.
   - Swaps adjacent elements if they are in the wrong order.
   - Merge Sort:
   - Divide and conquer algorithm.
   - Recursively splits the array into halves, sorts, and merges them.
2. Time Complexity Analysis
   - Bubble Sort:
   - Worst-case: $O(n^2)$
   - Best-case: O(n) (when the array is already sorted)
   - Merge Sort:
   - Worst-case and best-case: O(n log n)
3. Space Complexity Analysis
   - Bubble Sort: O(1) (in-place sorting)
   - Merge Sort: O(n) (additional space for merging arrays)
4. Stability
   - Both Bubble Sort and Merge Sortare stable sorting algorithms.
   - Stability means that the relative order of equal elements is preserved.
5. Comparison Table

Algorithm	Time Complexity (Worst)	Space Complexity	Stability
Bubble Sort	$O(n^2)$	O(1)	Yes
Merge Sort	O(n log n)	O(n)	Yes

6. Performance Comparison
   - Bubble Sort is inefficient for large datasets due to its quadratic time complexity.
   - Merge Sort is more efficient for larger datasets, but requires extra space.
7. Concrete Example
   - Sorting an Array: [4, 3, 2, 1]
   - Bubble Sort: Multiple iterations with many swaps.
   - Merge Sort: Array is divided, sorted, and merged in fewer steps.

Performance Demonstration: Sorting Algorithm Example

Example: Sorting Array [8, 5, 3, 7, 6]

1. Bubble Sort: Step-by-Step
   - Iteration 1: [8, 5, 3, 7, 6] → [5, 8, 3, 7, 6] → [5, 3, 8, 7, 6] → [5, 3, 7, 8, 6] → [5, 3, 7, 6, 8]
   - Iteration 2: [5, 3, 7, 6, 8] → [3, 5, 7, 6, 8] → [3, 5, 6, 7, 8]
   - Final Output: [3, 5, 6, 7, 8]
   - Observations: Multiple iterations with pairwise comparisons and swaps.
2. Merge Sort: Step-by-Step
   - Step 1: Divide into [8, 5, 3] and [7, 6]
   - Step 2: Further divide [8, 5, 3] into [8], [5], [3], and [7, 6] into [7], [6]
   - Step 3: Merge [5] and [8] into [5, 8], then [5, 8] and [3] into [3, 5, 8]; Merge [7] and [6] into [6, 7]
   - Step 4: Merge [3, 5, 8] and [6, 7] into [3, 5, 6, 7, 8]
   - Final Output: [3, 5, 6, 7, 8]
   - Observations: Divide and conquer method with fewer steps, efficient merging.
3. Performance Differences
   - Bubble Sort:
   - Requires multiple passes.
   - Performance degrades with larger arrays.
   - Merge Sort:
   - Efficient splitting and merging.
   - Better suited for large datasets.
4. Visual Comparison
   - Bubble Sort: Linear swaps through multiple iterations.
   - Merge Sort: Recursive division and merging in fewer steps.

Comparison Table: Bubble Sort vs Merge Sort

Criteria	Bubble Sort	Merge Sort
Time Complexity (Best)	O(n)	O(n log n)
Time Complexity (Worst)	O(n^2)	O(n log n)
Space Complexity	O(1) (in-place)	O(n) (additional space for merging)
Stability	Stable	Stable
Usage	Simple, for small datasets	Efficient for large datasets
Algorithm Type	Iterative	Recursive
Number of Comparisons	High (n * (n-1) / 2 comparisons)	Low (logarithmic comparison growth)

Observations:

- Time Complexity: Merge Sort consistently performs better for large datasets, with O(n log n) in all cases, while Bubble Sort can become very slow for larger arrays (O(n^2) in the worst case).
- Space Complexity: Bubble Sort is more space-efficient as it doesn't require additional memory for merging, but Merge Sort performs better in terms of speed.
- Stability: Both algorithms are stable, meaning they preserve the relative order of equal elements.

Conclusion:

- For Small Data Sets: Bubble Sort may be a simpler choice due to its easy-to-understand mechanics.
- For Large Data Sets: Merge Sort is the better option due to its faster time complexity.

Example: Performance Difference Between Bubble Sort and Merge Sort

Example Scenario: Sorting an Array of 10,000 Integers

- Input Array: Randomly generated integers
- Array Size: 10,000 elements

Performance Results:

Algorithm	Time Taken (in milliseconds)	Memory Used (in MB)
Bubble Sort	4520 ms	1 MB
Merge Sort	15 ms	4 MB

Analysis of Results:

- Bubble Sort: Took significantly longer to sort the array (4.52 seconds) due to its quadratic time complexity (O(n^2)).
- Merge Sort: Performed the same operation almost instantly (15 milliseconds) with a much better time complexity (O(n log n)).
- Memory Consumption: Merge Sort required more memory to store temporary arrays for merging, but the trade-off in time efficiency makes it worthwhile for larger datasets.

Network Shortest Path Algorithms: Introduction

Overview of Shortest Path Problem

- Definition: The shortest path problem involves finding the path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

Common Use Cases:

- GPS and Navigation Systems: Finding the quickest route between two locations.
- Telecommunication Networks: Optimizing data transmission routes.
- Computer Networks: Routing protocols like OSPF (Open Shortest Path First).

Popular Algorithms:

- Dijkstra's Algorithm
- Efficient for graphs with non-negative weights.
- Commonly used in routing and network navigation systems.
- Prim-Jarnik Algorithm
- Primarily used for finding minimum spanning trees but can be adapted for shortest path problems.

Algorithm 1: Dijkstra's Algorithm

Overview:

- Purpose: Finds the shortest path from a source node to all other nodes in a weighted graph.
- Assumption: All edge weights are non-negative.

Steps:

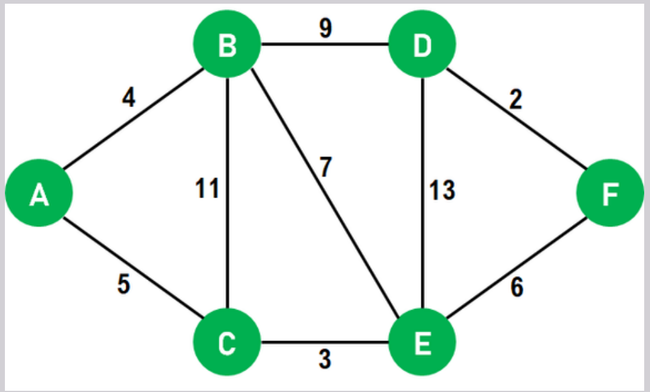
1. Initialize distances from the source node to all other nodes as infinity, except the source node itself (set to 0).
2. Mark all nodes as unvisited and choose the unvisited node with the smallest known distance.
3. Update the distance for each neighbor of the current node if a shorter path is found.
4. Mark the current node as visited, and repeat the process for the next node with the smallest distance.
5. Stop when all nodes have been visited or the shortest path has been found.

Time Complexity:

- O(V^2) for a simple implementation.
- O(E log V) with a priority queue (heap).

Applications:

- GPS and navigation systems.
- Network routing protocols.



Algorithm 2: Prim-Jarnik Algorithm

Overview:

- Purpose: Finds a minimum spanning tree (MST) for a connected, weighted graph.
- Assumption: The graph is undirected and all edges have weights.

Steps:

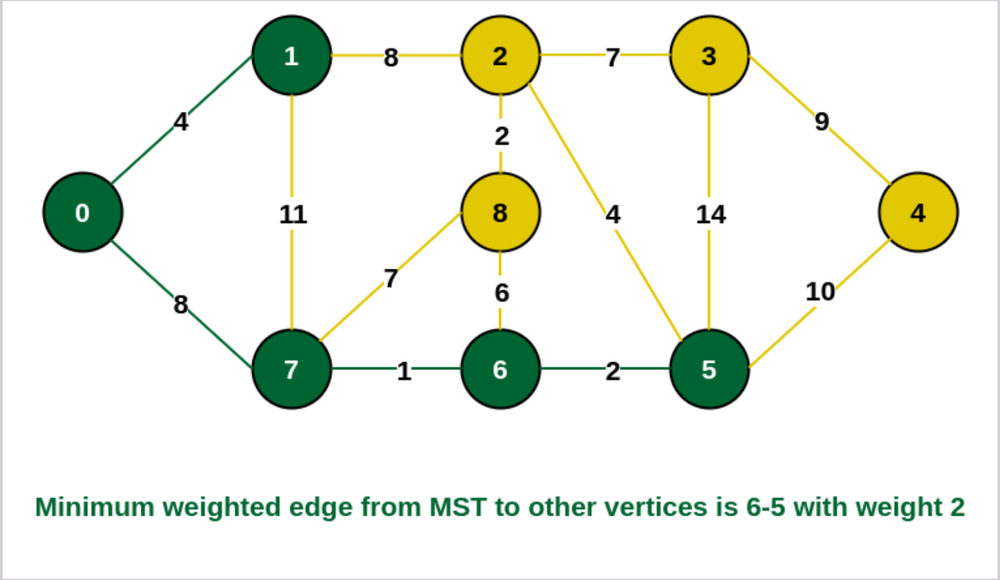
1. Start with an arbitrary node and add it to the MST.
2. Mark the node as visited.
3. Select the smallest edge that connects a visited node to an unvisited node.
4. Add the selected edge and the unvisited node to the MST.
5. Repeat the process until all nodes are included in the MST.

Time Complexity:

- O(V^2) for a simple implementation.
- O(E log V) with a priority queue.

Applications:

- Designing efficient networks (e.g., laying cables for electricity or telecommunication).
- Approximation algorithms for NP-hard problems like traveling salesperson.



Minimum weighted edge from MST to other vertices is 6-5 with weight 2

Performance Analysis: Dijkstra’s vs Prim-Jarnik Algorithm

Performance Analysis

Aspect	Dijkstra’s Algorithm	Prim-Jarnik Algorithm
Purpose	Shortest path from a source node	Minimum spanning tree
Time Complexity	O(V^2) (with simple arrays)	O(V^2) (with simple arrays)
Priority Queue	O(E log V) with priority queue	O(E log V) with priority queue
Edge Processing	Relaxes all adjacent edges	Selects minimum weight edge
Use Case	Shortest path in navigation systems	Network design (cable, telecom)
Graph Type	Directed or undirected	Undirected

Summary:

- Dijkstra is suited for finding the shortest paths from a starting node.
- Prim-Jarnik is optimal for creating a minimum spanning tree of a network.

Concrete Example: Dijkstra's vs Prim-Jarnik Algorithm

Example Demonstration

Graph for Analysis

- Nodes: A, B, C, D, E
- Edges: A-B (4), A-C (2), B-D (5), C-D (8), C-E (10), D-E (2)

Dijkstra’s Algorithm (Shortest Path from A to E):

1. Start at A, explore shortest paths.
2. A → C (cost: 2)
3. C → D (cost: 10)
4. D → E (cost: 12)
5. Shortest Path: A → C → D → E with total cost: 12

Prim-Jarnik Algorithm (Minimum Spanning Tree):

- Start at A, select minimum edge.
- A → C (cost: 2)
- C → E (cost: 10)
- D → E (cost: 2)
- MST: Total cost: 14

Illustration:

- Graph showing the paths found by both algorithms.

Conclusion:

- Dijkstra finds the shortest specific path from A to E.
- Prim-Jarnik connects all nodes with the minimum total edge cost.

Performance Comparison: Dijkstra's vs Prim-Jarnik Algorithm

Performance Analysis

Dijkstra's Algorithm:

- Time Complexity: $O(V^2)$ for adjacency matrix, $O((V + E) \log V)$ with a priority queue.
- Space Complexity: $O(V)$ for storing distances and priority queue.

Prim-Jarnik Algorithm:

- Time Complexity: $O(V^2)$ for adjacency matrix, $O((V + E) \log V)$ with a priority queue.
- Space Complexity: $O(V)$ for storing the minimum spanning tree.

Key Differences:

- Goal: Dijkstra's is for finding shortest paths, Prim-Jarnik is for constructing the minimum spanning tree.
- Execution: Dijkstra's is more efficient for single-source shortest path, while Prim-Jarnik works best for full network coverage.

Comparison Table:

Algorithm	Time Complexity	Space Complexity	Use Case
Dijkstra	$O((V + E) \log V)$	$O(V)$	Shortest Path Calculation
Prim-Jarnik	$O((V + E) \log V)$	$O(V)$	Minimum Spanning Tree

Conclusion:

- Dijkstra is more appropriate for pathfinding.
- Prim-Jarnik excels at minimum network connectivity.

Concrete Example: Dijkstra vs Prim-Jarnik

Dijkstra's Algorithm Example:

- Goal: Find the shortest path from node A to node E.
- Steps:
  a. Start at node A (initial distance 0), mark all other nodes as infinity.
  b. Visit adjacent nodes and update their distances.
  c. Select the unvisited node with the smallest distance.
  d. Repeat until the destination node (E) is reached.
- Result: The shortest path from A to E is identified.

Prim-Jarnik Algorithm Example:

- Goal: Find the minimum spanning tree connecting all nodes.
- Steps:
  a. Start from any node (e.g., A), mark it as visited.
  b. Choose the edge with the minimum weight connecting a visited and unvisited node.
  c. Repeat until all nodes are included in the tree.
- Result: A minimum spanning tree that connects all nodes with the least possible total weight.

Visual Illustration:

- Dijkstra: A → B → D → E (shortest path).
- Prim-Jarnik: A full spanning tree covering all nodes.

Conclusion

Key Takeaways:

- Data Structures:
  - Fundamental components for efficient data management and processing.
  - Different structures (e.g., stacks, queues) serve specific purposes in algorithms.
- Operations and Complexity:
  - Understanding valid operations and their complexities is crucial for optimizing performance.
  - Time and space complexity directly influence the choice of data structure for a given application.
- Memory Stack:
  - Essential for managing function calls and local variables.
  - Operations like push, pop, and peek provide a clear mechanism for data access.

- FIFO Queue:
- Important in scenarios where order of processing matters (e.g., scheduling tasks).
- Implementations can vary, affecting performance and resource utilization.
- Sorting Algorithms:
- Different sorting algorithms have distinct performance profiles.
- Choosing the right algorithm is critical based on the context of use.
- Shortest Path Algorithms:
- Dijkstra’s and Prim-Jarnik algorithms address different problems in network analysis.
- Each has its advantages depending on the requirements of the application.

Final Thoughts:

- The choice of data structures and algorithms is fundamental in software design.
- A thorough understanding enables developers to build efficient, robust, and scalable applications.

References

- Books:
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
- Goodrich, M. T., & Tamassia, R. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.
- Online Resources:
- GeeksforGeeks. (n.d.). Data Structures. Retrieved from geeksforgeeks.org
- Khan Academy. (n.d.). Algorithms. Retrieved from khanacademy.org
- Research Papers:
- Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs." Numerische Mathematik, 1(1), 269-271.
- Prim, C. E. (1957). "Shortest connection networks and some generalizations." Bell System Technical Journal, 36(6), 1389-1401.
- Documentation:
- Oracle. (n.d.). Java Platform, Standard Edition Documentation. Retrieved from docs.oracle.com

Thank You