Data Structures and Algorithms: A Practical Guide with Java Examples

Table of Contents

7. Graphs
   - Real-life analogy: City Road map
   - Properties and types (directed, undirected, weighted)
   - Java code examples
8. Hash Tables
   - Real-life analogy: Library index
   - Properties and usage
   - Java code examples
9. Sorting Algorithms
   - Real-life analogy: Sorting a deck of cards
   - Types of sorting (bubble, selection, insertion, merge, quick)
   - Java code examples
10. Search Algorithms
   - Real-life analogy: Finding a book in a library
   - Types of searching (linear, binary)
   - Java code examples
11. Dynamic Programming
   - Real-life analogy: Planning a road trip
   - Properties and examples (Fibonacci sequence, knapsack problem)
   - Java code examples
12. Pathfinding Algorithms
   - Real-life analogy: Finding the shortest route on a GPS
   - Examples (Dijkstra's algorithm, A* algorithm)
   - Java code examples
13. Conclusion
   - Summary
   - Further reading and resources

Chapter 1: Introduction

What are Data Structures and Algorithms?

Data structures are ways to store and organize data efficiently. Algorithms are step-by-step procedures for solving problems. Together, they enable efficient data management and problem-solving in software development.

Importance and Applications

Data structures and algorithms are foundational for creating efficient software. They are used in databases, operating systems, networking, artificial intelligence, and more.

Chapter 2: Arrays

Real-life Analogy: Bookshelf

Imagine a bookshelf with a fixed number of compartments, each capable of holding one book. Each compartment has a specific index (or position), making it easy to locate a book if you know its index.

Properties and Usage

- Fixed size: Once an array is created, its size cannot be changed.
- Random access: You can access any element directly using its index.
- Efficient read/write: Reading from or writing to an array is fast and efficient.

Java Code Examples

Creating and Accessing an Array:

Ex: ArrayExample.java

```
public class ArrayExample {
    public static void main(String[] args) {
```

```
        // Create an array of integers
        int[] bookshelf = new int[5];

        // Add books to the bookshelf
        bookshelf[0] = 101;
        bookshelf[1] = 102;
        bookshelf[2] = 103;
        bookshelf[3] = 104;
        bookshelf[4] = 105;

        // Access and print a book at index 2
        System.out.println("Book at index 2: " + bookshelf[2]);

        // Print all books
        System.out.println("All books on the bookshelf:");
        for (int i = 0; i < bookshelf.length; i++) {
            System.out.println("Index " + i + ": " + bookshelf[i]);
        }
    }
}
```

Common Array Operations:

1. Find the maximum value in an array:

Ex.MaxValue.java

```
public class MaxValue {
    public static void main(String[] args) {
        int[] numbers = {5, 3, 8, 6, 2, 10, 7};
```

```
        int max = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] > max) {
                max = numbers[i];
            }
        }

        System.out.println("Maximum value: " + max);
    }
}
```

2. Reverse an array:

Ex.ReverseArray.java

```
public class ReverseArray {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        int start = 0;
        int end = numbers.length - 1;
        while (start < end) {
            int temp = numbers[start];
            numbers[start] = numbers[end];
            numbers[end] = temp;
            start++;
            end--;
        }

        System.out.println("Reversed array:");
        for (int num : numbers) {
            System.out.print(num + " ");
```

```
        }
    }
}
```

Chapter 3: Linked Lists

Real-life Analogy: Treasure Hunt

A treasure hunt with clues is like a linked list. Each clue (node) tells you where the next clue (next node) is. You start at the first clue (head) and follow the trail until you reach the end.

Types of Linked Lists

1. Singly Linked List: Each node points to the next node.

2. Doubly Linked List: Each node points to both the next and previous nodes.

3. Circular Linked List: The last node points back to the first node, forming a circle.

Java Code Examples

Singly Linked List:

Node.java

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

public class SinglyLinkedList {
    Node head;

    // Insert a new node at the end
    public void insert(int data) {
```

```
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }

    // Print the linked list
    public void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList();  // Output: 10 20 30
    }
}
```

Doubly Linked List:

DoublyNode.java

```
class DoublyNode {
    int data;
    DoublyNode next;
    DoublyNode prev;

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

public class DoublyLinkedList {
    DoublyNode head;

    // Insert a new node at the end
    public void insert(int data) {
        DoublyNode newNode = new DoublyNode(data);
        if (head == null) {
            head = newNode;
        } else {
            DoublyNode temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
```

```
            newNode.prev = temp;
        }
    }

    // Print the linked list
    public void printList() {
        DoublyNode temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList();  // Output: 10 20 30
    }
}
```

Chapter 4: Stacks

Real-life Analogy: Stack of Plates

Imagine a stack of plates. You can only take the top plate off or add a new plate on top. It's a Last In, First Out (LIFO) structure.

Properties and Usage

- LIFO: Last In, First Out

- Push: Add an item to the top of the stack
- Pop: Remove an item from the top of the stack
- Peek: View the item on the top without removing it

Java Code Examples

Stack Implementation:

StackExample.java

```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        // Push elements onto the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // Peek the top element
        System.out.println("Top element: " + stack.peek());  // Output: 30

        // Pop elements from the stack
        System.out.println("Popped element: " + stack.pop());  // Output: 30
        System.out.println("Popped element: " + stack.pop());  // Output: 20

        // Print remaining elements
        System.out.println("Remaining stack: " + stack);  // Output: [10]
    }
}
```

Chapter 5: Queues

Real-life Analogy: Line at a Ticket Counter

Imagine a line at a ticket counter. The first person in line is the first one to get a ticket. It's a First In, First Out (FIFO) structure.

Types of Queues

1. Simple Queue: Basic FIFO structure.
2. Circular Queue: The last position is connected back to the first position.
3. Priority Queue: Elements are processed based on priority.

Java Code Examples

Simple Queue Implementation:

QueueExample.java

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

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();

        // Enqueue elements
        queue.add(10);
        queue.add(20);
        queue.add(30);
```

```
        // Peek the front element
        System.out.println("Front element: " + queue.peek());  // Output: 10

        // Dequeue elements
        System.out.println("Dequeued element: " + queue.poll());  // Output: 10
        System.out.println("Dequeued element: " + queue.poll());  // Output: 20

        // Print remaining elements
        System.out.println("Remaining queue: " + queue);  // Output: [30]
    }
}
```

Chapter 6: Trees

Real-life Analogy: Family Tree

A family tree starts from one ancestor (root) and branches out to children, grandchildren, and so on. Each person (node) can have multiple descendants (children).

Types of Trees

1. Binary Tree: Each node has at most two children.
2. Binary Search Tree: A binary tree with sorted properties.
3. AVL Tree: A self-balancing binary search tree.

Java Code Examples

Binary Search Tree (BST):

TreeNode.java

```
class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        left = right = null;
    }
}

public class BinarySearchTree {
    TreeNode root;

    // Insert a new node
    public void insert(int data) {
        root = insertRec(root, data);
    }

    private TreeNode insertRec(TreeNode root, int data) {
        if (root == null) {
            root = new TreeNode(data);
            return root;
        }

        if (data < root.data) {
            root.left = insertRec(root.left, data);
        } else if (data > root.data) {
            root.right = insertRec(root.right, data);
        }
```

```
        return root;
    }

    // In-order traversal
    public void inOrder() {
        inOrderRec(root);
    }

    private void inOrderRec(TreeNode root) {
        if (root != null) {
            inOrderRec(root.left);
            System.out.print(root.data + " ");
            inOrderRec(root.right);
        }
    }

    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();
        tree.insert(50);
        tree.insert(30);
        tree.insert(70);
        tree.insert(20);
        tree.insert(40);
        tree.insert(60);
        tree.insert(80);

        tree.inOrder();  // Output: 20 30 40 50 60 70 80
    }
}
```

Chapter 7: Graphs

Real-life Analogy: City Road Map

A city's road map can be represented as a graph. Intersections are nodes, and roads are edges connecting the nodes. It helps in finding the shortest path from one place to another.

Properties and Types

1. Directed Graph: Edges have a direction.
2. Undirected Graph: Edges don't have a direction.
3. Weighted Graph: Edges have weights (costs).

Java Code Examples:

Graph Representation using Adjacency List:

Graph.java

```
import java.util.LinkedList;

class Graph {
    private int V;   // Number of vertices
    private LinkedList<Integer> adj[]; // Adjacency list

    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }
```

```
    // Add an edge to the graph
    void addEdge(int v, int w) {
        adj[v].add(w);
    }

    // Print the graph
    void printGraph() {
        for (int i = 0; i < V; i++) {
            System.out.print("Vertex " + i + ":");
            for (Integer vertex : adj[i]) {
                System.out.print(" -> " + vertex);
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph(4);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(2, 0);
        graph.addEdge(2, 3);
        graph.addEdge(3, 3);

        graph.printGraph();
    }
}
```

Chapter 8: Hash Tables

Real-life Analogy: Library Index

A library index helps in quickly locating a book. Similarly, a hash table provides fast access to data using a hash function to map keys to indices.

Properties and Usage:

- Fast access: Provides average-case constant time complexity for search, insert, and delete operations.
- Collisions: Multiple keys can map to the same index, handled using techniques like chaining or open addressing.

Java Code Examples

Hash Table Implementation:

HashTable.java

```
import java.util.LinkedList;

class HashTable {
    private int BUCKET; // Number of buckets
    private LinkedList<Integer>[] table; // Array of linked lists

    HashTable(int b) {
        BUCKET = b;
        table = new LinkedList[b];
        for (int i = 0; i < b; i++) {
            table[i] = new LinkedList<>();
```

```
    }
}

// Hash function
int hashFunction(int key) {
    return key % BUCKET;
}

// Insert a key into the hash table
void insert(int key) {
    int index = hashFunction(key);
    table[index].add(key);
}

// Delete a key from the hash table
void delete(int key) {
    int index = hashFunction(key);
    table[index].remove((Integer) key);
}

// Search for a key in the hash table
boolean search(int key) {
    int index = hashFunction(key);
    return table[index].contains(key);
}

// Print the hash table
void printTable() {
    for (int i = 0; i < BUCKET; i++) {
        System.out.print("Bucket " + i + ":");
```

```
            for (Integer key : table[i]) {
                System.out.print(" " + key);
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        HashTable hashTable = new HashTable(7);

        hashTable.insert(10);
        hashTable.insert(20);
        hashTable.insert(15);
        hashTable.insert(7);
        hashTable.insert(32);

        hashTable.printTable();

        System.out.println("Search 15: " + hashTable.search(15)); // Output: true
        hashTable.delete(15);
        System.out.println("Search 15: " + hashTable.search(15)); // Output: false
    }
}
```

Chapter 9: Sorting Algorithms

Real-life Analogy: Sorting a Deck of Cards

Sorting a deck of cards is similar to sorting algorithms. Different techniques can be used to arrange the cards in a specific order.

Types of Sorting Algorithms

1. Bubble Sort: Repeatedly compare and swap adjacent elements.
2. Selection Sort: Find the minimum element and place it at the beginning.
3. Insertion Sort: Insert elements into their correct position.
4. Merge Sort: Divide the array into halves, sort each half, and merge them.
5. Quick Sort: Partition the array around a pivot, then sort each partition.

Java Code Examples

Bubble Sort:

BubbleSort.java

```
public class BubbleSort {
    public static 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]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        bubbleSort(arr);
```

```
        System.out.println("Sorted array:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Quick Sort:

QuickSort.java

```
public class QuickSort {
    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);
        }
    }

    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
```

```
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }

    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        quickSort(arr, 0, arr.length - 1);
        System.out.println("Sorted array:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Chapter 10: Search Algorithms

Real-life Analogy: Finding a Book in a Library

Searching for a specific book in a library is similar to search algorithms. Different techniques can be used to find the desired book quickly.

Types of Searching Algorithms

1. Linear Search: Sequentially check each element until the target is found.
2. Binary Search: Repeatedly divide the sorted array in half to find the target.

Java Code Examples

Linear Search:

LinearSearch.java

```
public class LinearSearch {
    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 4, 10, 40};
        int target = 10;
        int result = linearSearch(arr, target);
        if (result == -1) {
            System.out.println("Element not present");
        } else {
            System.out.println("Element found at index " + result);
        }
    }
}
```

Binary Search:

BinarySearch.java

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (arr[mid] == target) {
                return mid;
            }

            if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 4, 10, 40};
        int target = 10;
        int result = binarySearch(arr, target);
        if (result == -1) {
            System.out.println("Element not present");
        } else {
            System.out.println("Element found at index " + result);
```

```
        }
    }
}
```

Chapter 11: Dynamic Programming

Real-life Analogy: Planning a Road Trip

When planning a road trip, you break down the journey into smaller segments and optimize each segment for the best overall route. Dynamic programming solves problems by breaking them into subproblems and solving each subproblem just once.

Properties and Examples

- Overlapping subproblems: Solve the same subproblems multiple times.
- Optimal substructure: The optimal solution of the main problem contains the optimal solutions of its subproblems.

Java Code Examples

Fibonacci Sequence:

Fibonacci.java

```
public class Fibonacci {
    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        int[] fib = new int[n + 1];
        fib[0] = 0;
        fib[1] = 1;
```

```
        for (int i = 2; i <= n; i++) {
            fib[i] = fib[i - 1] + fib[i - 2];
        }
        return fib[n];
    }

    public static void main(String[] args) {
        int n = 10;
        System.out.println("Fibonacci number is " + fibonacci(n));  // Output: 55
    }
}
```

Knapsack Problem:

Knapsack.java

```
public class Knapsack {
    public static int knapsack(int[] weights, int[] values, int W) {
        int n = weights.length;
        int[][] dp = new int[n + 1][W + 1];

        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= W; w++) {
                if (i == 0 || w == 0) {
                    dp[i][w] = 0;
                } else if (weights[i - 1] <= w) {
                    dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
```

```
        }

        return dp[n][W];
    }

    public static void main(String[] args) {
        int[] weights = {10, 20, 30};
        int[] values = {60, 100, 120};
        int W = 50;
        System.out.println("Maximum value: " + knapsack(weights, values, W));  // Output: 220
    }
}
```

Chapter 12: Pathfinding Algorithms

Real-life Analogy: Finding the Shortest Route on a GPS

When using a GPS, it calculates the shortest or fastest route from your starting point to your destination. Pathfinding algorithms help in finding the optimal path in graphs.

Examples

1. Dijkstra's Algorithm: Finds the shortest path from a source node to all other nodes in a weighted graph.
2. A Algorithm: Uses heuristics to find the shortest path efficiently.

Java Code Examples

Dijkstra's Algorithm:

Graph.java

```
import java.util.*;

class Graph {
    private int V;
    private LinkedList<Edge>[] adjList;

    class Edge {
        int dest, weight;
        Edge(int dest, int weight) {
            this.dest = dest;
            this.weight = weight;
        }
    }

    class Node implements Comparable<Node> {
        int vertex, distance;
        Node(int vertex, int distance) {
            this.vertex = vertex;
            this.distance = distance;
        }
        public int compareTo(Node other) {
            return this.distance - other.distance;
        }
    }

    Graph(int V) {
        this.V = V;
```

```
    adjList = new LinkedList[V];
    for (int i = 0; i < V; i++) {
        adjList[i] = new LinkedList<>();
    }
}

void addEdge(int src, int dest, int weight) {
    adjList[src].add(new Edge(dest, weight));
    adjList[dest].add(new Edge(src, weight)); // For undirected graph
}

void dijkstra(int src) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;
    pq.add(new Node(src, 0));

    while (!pq.isEmpty()) {
        Node node = pq.poll();
        for (Edge edge : adjList[node.vertex]) {
            if (dist[node.vertex] + edge.weight < dist[edge.dest]) {
                dist[edge.dest] = dist[node.vertex] + edge.weight;
                pq.add(new Node(edge.dest, dist[edge.dest]));
            }
        }
    }

    // Print shortest distances
    System.out.println("Vertex Distance from Source");
```

```
        for (int i = 0; i < V; i++) {
            System.out.println(i + "\t\t" + dist[i]);
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph(5);
        graph.addEdge(0, 1, 10);
        graph.addEdge(0, 4, 20);
        graph.addEdge(1, 2, 10);
        graph.addEdge(1, 3, 50);
        graph.addEdge(2, 3, 10);
        graph.addEdge(3, 4, 10);

        graph.dijkstra(0); // Output: Vertex distances from source vertex 0
    }
}
```

A Algorithm:

AStarNode.java

```
import java.util.*;

class AStarNode implements Comparable<AStarNode> {
    int x, y, cost, heuristic;
    AStarNode parent;

    AStarNode(int x, int y, int cost, int heuristic) {
        this.x = x;
        this.y = y;
```

```
        this.cost = cost;
        this.heuristic = heuristic;
    }

    public int compareTo(AStarNode other) {
        return (this.cost + this.heuristic) - (other.cost + other.heuristic);
    }
}

public class AStarAlgorithm {
    private static final int[][] DIRECTIONS = { {0, 1}, {1, 0}, {0, -1}, {-1, 0}

 };

    public static boolean isValid(int x, int y, int rows, int cols) {
        return x >= 0 && x < rows && y >= 0 && y < cols;
    }

    public static int heuristic(int x, int y, int endX, int endY) {
        return Math.abs(endX - x) + Math.abs(endY - y);
    }

    public static void aStarSearch(int[][] grid, int startX, int startY, int endX, int endY) {
        int rows = grid.length;
        int cols = grid[0].length;

        PriorityQueue<AStarNode> pq = new PriorityQueue<>();
        pq.add(new AStarNode(startX, startY, 0, heuristic(startX, startY, endX, endY)));

        boolean[][] visited = new boolean[rows][cols];
```

```
        while (!pq.isEmpty()) {
            AStarNode curr = pq.poll();

            if (curr.x == endX && curr.y == endY) {
                LinkedList<AStarNode> path = new LinkedList<>();
                while (curr != null) {
                    path.addFirst(curr);
                    curr = curr.parent;
                }
                for (AStarNode node : path) {
                    System.out.print("(" + node.x + "," + node.y + ") -> ");
                }
                System.out.println("Path Cost: " + path.getFirst().cost);
                return;
            }

            if (visited[curr.x][curr.y]) {
                continue;
            }
            visited[curr.x][curr.y] = true;

            for (int[] dir : DIRECTIONS) {
                int newX = curr.x + dir[0];
                int newY = curr.y + dir[1];

                if (isValid(newX, newY, rows, cols) && !visited[newX][newY] &&
grid[newX][newY] == 1) {
                    pq.add(new AStarNode(newX, newY, curr.cost + 1, heuristic(newX, newY,
endX, endY)));
                    pq.peek().parent = curr;
```

```
            }
        }
    }

    System.out.println("No path found!");
}

public static void main(String[] args) {
    int[][] grid = {
        {1, 1, 1, 0, 1},
        {0, 1, 0, 1, 1},
        {1, 1, 1, 0, 1},
        {1, 0, 1, 1, 1},
        {1, 1, 1, 1, 1}
    };

    int startX = 0, startY = 0;
    int endX = 4, endY = 4;

    aStarSearch(grid, startX, startY, endX, endY);
}
}
```

This summary covers the essential topics in data structures and algorithms, providing clear explanations and practical Java examples for each concept.