Data Structures and Algorithms

Comprehensive Guide

Contents

1 Introduction to Data Structures

1.1 Definition and Classification

Data structures are specialized formats for organizing, storing, and managing data to enable efficient access and modification. They form the backbone of efficient algorithms by providing structured ways to handle data. Data structures are classified into *primitive* (e.g., integers, floats) and *non-primitive* (e.g., arrays, lists, trees). Non-primitive structures are further divided into *linear* (e.g., arrays, linked lists) and *non-linear* (e.g., trees, graphs). This section explains the importance of choosing appropriate data structures for specific problems, with examples like using arrays for sequential data versus trees for hierarchical data.

1.1.1 Types of Data Structures

- **Primitive**: Built-in types like `int`, `char`.
- **Non-Primitive Linear**: Arrays, linked lists, stacks, queues.
- **Non-Primitive Non-Linear**: Trees, graphs, heaps.

For example, an array is suitable for indexed access, while a linked list excels in dynamic insertions.

1.2 Abstract Data Types (ADTs)

An Abstract Data Type (ADT) defines a data structure by its operations, not its implementation. Examples include stacks (push, pop), queues (enqueue, dequeue), and lists (insert, delete). ADTs provide a high-level interface, allowing different implementations (e.g., a stack can be implemented using an array or linked list). This section discusses ADTs role in abstraction, with examples like the `List` ADT implemented as an `ArrayList` or `LinkedList` in Java.

1.2.1 ADT Examples

- **Stack ADT**: Operations include `push`, `pop`, `peek`.
- **Queue ADT**: Operations include `enqueue`, `dequeue`.

A pseudocode example of a stack ADT is provided to illustrate its operations.

1.3 Algorithm Analysis (Time/Space Complexity)

Algorithm analysis evaluates efficiency in terms of *time complexity* (execution time) and *space complexity* (memory usage). Time complexity is measured using Big-O notation, which describes the worst-case growth rate (e.g., O(n) for linear search). Space complexity considers memory used by variables and data structures. This section explains how to analyze algorithms, using examples like comparing linear search (O(n)) versus binary search (O(log n)).

1.3.1 Analysis Techniques

- **Time Complexity**: Counting operations, e.g., loops in a sorting algorithm.
- **Space Complexity**: Measuring auxiliary space, e.g., recursive call stack.

An example analyzes a loop iterating `n` times, yielding O(n) time complexity.

1.4 Design Techniques (Incremental, Divide-and-Conquer)

Algorithm design techniques include *incremental* (building solutions step-by-step, e.g., insertion sort) and *divide-and-conquer* (breaking problems into smaller subproblems, e.g., merge sort). This section explains these strategies with examples, such as divide-and-conquer in quicksort, which partitions an array and recursively sorts subarrays.

1.4.1 Incremental Example

Insertion sort builds a sorted array one element at a time, with O(nš) complexity.

1.4.2 Divide-and-Conquer Example

```
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);
    }
}
```

This pseudocode illustrates merge sorts divide-and-conquer approach.

2 Recursion

2.1 Principles and Types (Direct, Indirect, Tail)

Recursion occurs when a function calls itself to solve a problem by breaking it into smaller instances. Types include *direct* (function calls itself), *indirect* (function A calls function B, which calls A), and *tail* (recursive call is the last operation). This section explains recursions mechanics, including base cases and recursive cases, and compares recursive versus iterative solutions.

2.1.1 Types of Recursion

- **Direct**: Factorial function calling itself.
- **Indirect**: Two functions mutually calling each other.
- **Tail**: Recursive call as the final step, optimizing stack usage.

2.2 Examples (Tower of Hanoi, Fibonacci)

Recursion is illustrated with the Tower of Hanoi (moving disks between pegs) and Fibonacci sequence (each number is the sum of the two preceding ones).

2.2.1 Tower of Hanoi

```
void towerOfHanoi(int n, char src, char aux, char dest) {
    if (n == 1) {
        print("Move disk 1 from " + src + " to " + dest);
        return;
    }
    towerOfHanoi(n - 1, src, dest, aux);
    print("Move disk " + n + " from " + src + " to " + dest);
    towerOfHanoi(n - 1, aux, src, dest);
}
```

This solves Tower of Hanoi for `n` disks.

2.2.2 Fibonacci

```c
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

This computes the nth Fibonacci number, though its inefficient ($O(2^n)$).

2.3 Applications

Recursion is used in tree traversals, divide-and-conquer algorithms, and backtracking (e.g., solving mazes). This section discusses practical applications, such as parsing expressions or generating permutations, with examples showing recursive solutions elegance despite potential stack overflow risks.

3 Stacks

3.1 Operations (PUSH/POP)

A stack is a linear data structure following Last-In-First-Out (LIFO). Key operations are `push` (add element) and `pop` (remove top element). Other operations include `peek` (view top) and `isEmpty`. This section explains stack operations with a pseudocode implementation using an array.

3.1.1 Stack Operations

```
class Stack {
    int top, maxSize;
    int[] array;
    Stack(int size) {
        maxSize = size;
        array = new int[size];
        top = -1;
    }
    void push(int item) {
        if (top < maxSize - 1) array[++top] = item;
    }
    int pop() {
        if (top >= 0) return array[top--];
        return -1; // Stack underflow
    }
}
```

3.2 Applications (Expression Evaluation, Infix-to-Postfix)

Stacks are used in expression evaluation (e.g., evaluating postfix expressions), infix-to-postfix conversion, and function call management (call stack). This section details how stacks convert `a + b * c` to postfix `a b c * +` and evaluate it.

3.2.1 Infix-to-Postfix

Explains the algorithm to convert infix to postfix using a stack to manage operators, with an example walkthrough.

4 Queues

4.1 Operations (Insertion/Deletion)

A queue is a First-In-First-Out (FIFO) data structure. Key operations are `enqueue` (add to rear) and `dequeue` (remove from front). This section provides a pseudocode implementation using an array, discussing queue overflow and underflow.

4.1.1 Queue Operations

```
class Queue {
    int front, rear, maxSize;
    int[] array;
    Queue(int size) {
        maxSize = size;
        array = new int[size];
        front = 0;
        rear = -1;
    }
    void enqueue(int item) {
        if (rear < maxSize - 1) array[++rear] = item;
    }
    int dequeue() {
        if (front <= rear) return array[front++];
        return -1; // Queue underflow
    }
}
```

4.2 Variations (Circular, Priority Queues)

Queues have variations like *circular queues* (reusing empty space) and *priority queues* (elements dequeued based on priority). This section explains their implementations and use cases, such as task scheduling for priority queues.

4.2.1 Circular Queue

```
void enqueue(int item) {
    if ((rear + 1) % maxSize != front) {
        rear = (rear + 1) % maxSize;
        array[rear] = item;
    }
}
```

This shows circular queue enqueue to avoid space wastage.

5 Linked Lists

5.1 Types (Singly, Doubly, Circular)

Linked lists store elements in nodes, each containing data and a reference to the next node. Types include *singly* (one-way links), *doubly* (two-way links), and *circular* (last node links to first). This section compares their structures and use cases.

5.1.1 Linked List Types

- **Singly**: Simple, memory-efficient, forward traversal.
- **Doubly**: Bidirectional traversal, more memory.
- **Circular**: Continuous loop, useful for cyclic data.

5.2 Operations (Insertion, Deletion, Traversal)

Key operations include inserting nodes (at beginning, end, or position), deleting nodes, and traversing the list. Pseudocode examples illustrate these operations for a singly linked list.

5.2.1 Singly Linked List Insertion

```cpp
struct Node {
    int data;
    Node* next;
};
void insertAtHead(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}
```

5.3 Applications (Polynomial Addition)

Linked lists are used for polynomial representation (each node stores coefficient and exponent) and addition, dynamic memory allocation, and implementing other data structures.

This section shows how to add polynomials using linked lists.

6 Trees

6.1 Terminology and Binary Trees

Trees are hierarchical, non-linear data structures with nodes connected by edges. Terminology includes root, leaf, parent, child, and height. A *binary tree* has at most two children per node. This section explains tree properties and binary tree structure.

6.1.1 Binary Tree Structure

```
struct Node {
    int data;
    Node* left;
    Node* right;
};
```

6.2 Traversals (Preorder, Inorder, Postorder)

Tree traversals visit nodes in specific orders: *preorder* (root, left, right), *inorder* (left, root, right), *postorder* (left, right, root). Pseudocode and examples illustrate each traversal.

6.2.1 Inorder Traversal

```
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        print(root->data);
        inorder(root->right);
    }
}
```

6.3 Binary Search Trees (Insertion, Deletion)

A Binary Search Tree (BST) organizes nodes such that left children are smaller and right children are larger than the parent. This section covers insertion and deletion algorithms, with examples.

6.3.1 BST Insertion

```
Node* insert(Node* root, int data) {
    if (root == NULL) {
        Node* newNode = new Node();
        newNode->data = data;
        return newNode;
    }
    if (data < root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);
    return root;
}
```

6.4 Balanced Trees (AVL, B-Trees)

Balanced trees like AVL (height-balanced) and B-Trees (multi-way trees) ensure efficient operations. This section explains AVL rotations and B-Tree properties, with examples of maintaining balance.

6.4.1 AVL Tree Rotation

Explains single and double rotations to balance an AVL tree after insertion.

6.5 Huffman Algorithm

The Huffman algorithm creates an optimal prefix code for data compression using a binary tree. This section details the algorithm, building a Huffman tree based on character frequencies, with an example.

7 Sorting Algorithms

7.1 Techniques (Bubble, Quick, Merge, Shell)

Sorting arranges elements in order. This section covers:

- **Bubble Sort**: Repeatedly swaps adjacent elements (O(nš)).
- **Quick Sort**: Uses divide-and-conquer with pivoting (O(n log n) average).
- **Merge Sort**: Divides and merges sorted subarrays (O(n log n)).

- **Shell Sort**: Enhances insertion sort with gaps (O($n^{1.3}$)).

7.1.1 Quick Sort

```
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);
    }
}
```

7.2 Efficiency Analysis (Big-O Notation)

Compares sorting algorithms time complexities (e.g., Bubble: O(nš), Quick: O(n log n)) and space complexities, with detailed derivations using Big-O notation.

8 Searching and Hashing

8.1 Techniques (Sequential, Binary, Tree Search)

Searching locates elements in a data structure. Techniques include:

- **Sequential Search**: Linear scan (O(n)).
- **Binary Search**: Divides sorted array (O(log n)).
- **Tree Search**: Uses BST for searching (O(log n) average).

8.1.1 Binary Search

```c
int binarySearch(int arr[], int left, int right, int key) {
    if (right >= left) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key) return mid;
        if (arr[mid] > key) return binarySearch(arr, left, mid -
           1, key);
        return binarySearch(arr, mid + 1, right, key);
    }
    return -1;
}
```

8.2 Hash Functions and Collision Resolution

Hashing maps keys to indices using a hash function. Collisions (multiple keys mapping to the same index) are resolved using techniques like chaining or open addressing. This section explains hash functions and collision resolution strategies, with examples.

8.2.1 Chaining Example

Illustrates storing multiple keys in a linked list at the same hash index.

9 Graphs

9.1 Terminology and Representations

Graphs consist of vertices and edges, represented as adjacency lists or matrices. Terminology includes directed/undirected graphs, weighted edges, and cycles. This section explains graph representations with examples.

9.1.1 Adjacency List

```
struct Graph {
    int V;
    List<int>* adj;
};
```

9.2 Traversals (BFS, DFS)

Graph traversals include Breadth-First Search (BFS, level-order) and Depth-First Search (DFS, path exploration). Pseudocode and applications (e.g., cycle detection) are provided.

9.2.1 DFS

```
void DFS(Graph* graph, int v, bool visited[]) {
    visited[v] = true;
    for (int neighbor : graph->adj[v])
        if (!visited[neighbor])
            DFS(graph, neighbor, visited);
}
```

9.3 Spanning Trees (Kruskals, Prims Algorithms)

A spanning tree connects all vertices with minimal edges. Kruskals and Prims algorithms find minimum spanning trees for weighted graphs. This section details their steps and complexity.

9.3.1 Kruskals Algorithm

Explains sorting edges and using a union-find structure to build a minimum spanning tree.

9.4 Shortest Path (Dijkstras Algorithm)

Dijkstras algorithm finds the shortest path in a weighted graph. This section provides pseudocode and an example of finding the shortest path from a source vertex.

9.4.1 Dijkstras Algorithm

```
void dijkstra(Graph* graph, int src) {
    int dist[V];
    for (int i = 0; i < V; i++) dist[i] = INT_MAX;
    dist[src] = 0;
    // Priority queue and relaxation steps
}
```

10 Asymptotic Notations

10.1 Big-O, Omega, Theta Notations

Asymptotic notations describe algorithm performance:

- **Big-O**: Upper bound (worst-case).
- **Omega**: Lower bound (best-case).
- **Theta**: Tight bound (average-case).

This section explains their mathematical definitions and uses, with examples like O(nš) for bubble sort.

10.1.1 Notation Examples

Derives Big-O for a nested loop and Theta for merge sort.

10.2 Limitations of Big-O

Big-O focuses on worst-case and ignores constants, which can be misleading for small inputs or when constants are significant. This section discusses limitations, such as overlooking space complexity or practical performance, with examples comparing algorithms with similar Big-O but different real-world efficiencies.