Data Structures and Algorithms

A Study Guide for BSc Computer Science — Second Year

Introduction

Data Structures and Algorithms (DSA) is one of the most fundamental subjects in computer science. It deals with how data is organized in memory and how efficiently problems can be solved using systematic procedures called algorithms. A strong understanding of DSA directly determines how well a programmer can write efficient, scalable, and maintainable software.

Every real-world application — from search engines to social media feeds to navigation systems — relies on carefully chosen data structures and algorithms. Choosing the wrong data structure for a problem can make a program thousands of times slower than necessary.

This guide covers the core topics examined in a second-year BSc Computer Science DSA course: arrays, linked lists, stacks, queues, trees, sorting algorithms, and searching algorithms. Each topic includes definitions, working principles, time complexity analysis, and practical notes.

Chapter 1: Arrays

What is an Array?

An array is a linear data structure that stores elements of the same data type in contiguous memory locations. Each element is accessed using an index, which starts at 0 in most programming languages.

Arrays are one of the oldest and most widely used data structures. They form the foundation for many other structures and algorithms.

Key Properties:

- Fixed size: The size of an array is defined at the time of declaration and cannot be changed dynamically.
- Indexed access: Any element can be accessed in O(1) time using its index.
- Homogeneous: All elements must be of the same data type.
- Contiguous memory: Elements are stored in adjacent memory locations, which makes traversal fast.

Time Complexity of Array Operations:

Operation	Time Complexity	Notes

Access by index	O(1)	Direct calculation using base address
Search (unsorted)	O(n)	Must scan each element
Search (sorted)	O(log n)	Using binary search
Insertion at end	O(1)	If space is available
Insertion at middle	O(n)	Requires shifting elements
Deletion	O(n)	Requires shifting after removal

Limitations of Arrays: Because arrays have a fixed size, inserting or deleting elements in the middle requires shifting all subsequent elements, making these operations expensive. When the size of data is unknown in advance, arrays are not the ideal choice.

Chapter 2: Linked Lists

A linked list is a dynamic data structure where each element, called a node, contains two parts: the data and a pointer (or reference) to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory locations.

Types of Linked Lists:

- Singly Linked List: Each node points to the next node. Traversal is only in one direction.
- Doubly Linked List: Each node has two pointers, one to the next node and one to the previous node. Allows traversal in both directions.
- Circular Linked List: The last node points back to the first node, forming a loop.

Singly Linked List — Node Structure (Python):

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Time Complexity:

Operation	Singly Linked	Doubly Linked
Access by index	O(n)	O(n)
Search	O(n)	O(n)
Insertion at head	O(1)	O(1)
Insertion at tail	O(n)	O(1) with tail pointer
Deletion at head	O(1)	O(1)
Deletion at middle	O(n)	O(n)

Arrays vs Linked Lists:

The key advantage of a linked list over an array is dynamic sizing and efficient insertion or deletion at the beginning. However, linked lists use more memory per element because of the pointer overhead, and they do not support random access — you must traverse from the head to reach any element.

Chapter 3: Stacks and Queues

Stack

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. The last element inserted is the first one to be removed. Think of a stack of plates — you always take the top plate first.

Core operations:

- push(x): Insert element x onto the top of the stack — O(1)
- pop(): Remove and return the top element — O(1)
- peek(): Return the top element without removing it — O(1)
- isEmpty(): Check if the stack is empty — O(1)

Real-world applications of stacks include: function call management (the call stack in programming), undo operations in text editors, expression evaluation, and backtracking in algorithms.

Queue

A queue is a linear data structure that follows the First In First Out (FIFO) principle. The first element inserted is the first one to be removed — similar to a line at a ticket counter.

- enqueue(x): Add element x to the rear of the queue — O(1)
- dequeue(): Remove and return the front element — O(1)
- front(): Return the front element without removing — O(1)

Queues are used in CPU scheduling, print spooling, breadth-first search (BFS) in graphs, and handling requests in web servers.

Chapter 4: Trees

A tree is a non-linear, hierarchical data structure consisting of nodes connected by edges. Unlike arrays and linked lists which are linear, a tree organises data in a parent-child relationship.

Key Terminology:

- Root: The topmost node of the tree. It has no parent.
- Node: Each element in the tree. Contains data and references to child nodes.
- Leaf: A node with no children.
- Height of a tree: The number of edges on the longest path from the root to a leaf.
- Depth of a node: The number of edges from the root to that node.
- Subtree: A node and all its descendants form a subtree.

Binary Tree

A binary tree is a tree where each node has at most two children, referred to as the left child and the right child. Binary trees are the most commonly studied tree structure.

Tree Traversal Methods:

Traversal refers to visiting every node in the tree exactly once in a specific order.

- Inorder (Left, Root, Right): Visits nodes in ascending order for a Binary Search Tree.
- Preorder (Root, Left, Right): Used to create a copy of the tree or serialize it.
- Postorder (Left, Right, Root): Used to delete a tree or evaluate expression trees.
- Level Order (BFS): Visits nodes level by level from top to bottom using a queue.

Binary Search Tree (BST)

A Binary Search Tree is a binary tree with an ordering property: for every node, all values in the left subtree are smaller, and all values in the right subtree are greater. This property makes searching, insertion, and deletion efficient.

Operation	Average Case	Worst Case (Unbalanced)
Search	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)

The worst case occurs when the BST becomes a skewed tree — for example, inserting elements in sorted order causes every node to have only a right child, making the tree behave like a linked list. Balanced BSTs like AVL Trees and Red-Black Trees prevent this by maintaining height balance after every insertion and deletion.

Chapter 5: Sorting Algorithms

Sorting is the process of arranging elements in a specific order, typically ascending or descending. Sorting is one of the most studied problems in computer science because it is a prerequisite for many other algorithms such as binary search.

Bubble Sort

Bubble sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. After each full pass through the array, the largest unsorted element bubbles up to its correct position.

Time Complexity: $O(n^2)$ average and worst case. Space Complexity: O(1). Bubble sort is simple to understand but highly inefficient for large datasets.

Selection Sort

Selection sort divides the array into a sorted and an unsorted region. In each pass, it finds the minimum element from the unsorted region and places it at the beginning of that region.

Time Complexity: $O(n^2)$ in all cases. It performs fewer swaps than bubble sort but is still unsuitable for large inputs.

Insertion Sort

Insertion sort builds the sorted array one element at a time. It picks the next element and inserts it into its correct position among the already-sorted elements.

Time Complexity: $O(n^2)$ worst case, O(n) best case (already sorted). Insertion sort performs very well on small or nearly sorted datasets.

Merge Sort

Merge sort is a divide and conquer algorithm. It divides the array into two halves, recursively sorts each half, and then merges the two sorted halves into one sorted array.

Time Complexity: O(n log n) in all cases. Space Complexity: O(n) — requires additional memory for the merge step. Merge sort is stable and predictable, making it suitable for large datasets and external sorting.

Quick Sort

Quick sort is also a divide and conquer algorithm. It selects a pivot element and partitions the array into two sub-arrays: elements less than the pivot and elements greater than the pivot. It then recursively sorts each sub-array.

Time Complexity: O(n log n) average case, $O(n^2)$ worst case (when the smallest or largest element is always chosen as pivot). Space Complexity: O(log n) average. Quick sort is generally faster in practice than merge sort due to better cache performance.

Sorting Algorithm Comparison:

Algorithm	Best	Average	Worst	Space	Stable
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No

A stable sorting algorithm preserves the relative order of equal elements. This matters when sorting records with multiple fields — for example, sorting students first by grade and then by name.

Chapter 6: Searching Algorithms

Linear Search

Linear search scans each element in the array from left to right until the target element is found or the end of the array is reached.

Time Complexity: O(n). Linear search works on both sorted and unsorted arrays and does not require any preprocessing. It is the simplest search algorithm but inefficient for large datasets.

Binary Search

Binary search works only on sorted arrays. It repeatedly divides the search range in half by comparing the target with the middle element. If the target is smaller, it searches the left half; if larger, it searches the right half. This continues until the element is found or the range is empty.

Time Complexity: O(log n). Binary search is significantly faster than linear search for large sorted datasets. For example, searching in an array of one million elements requires at most 20 comparisons using binary search, compared to up to one million for linear search.

Binary Search — Iterative Implementation (Python):

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Chapter 7: Algorithm Complexity and Big O Notation

Big O notation describes how the runtime or space requirements of an algorithm grow as the input size n increases. It provides an upper bound on the growth rate, allowing us to compare algorithms independent of hardware or implementation details.

Common Complexity Classes:

Notation	Name	Example
O(1)	Constant	Array access by index
O(log n)	Logarithmic	Binary search
O(n)	Linear	Linear search, single loop

O(n log n)	Linearithmic	Merge sort, quick sort (avg)
$O(n^2)$	Quadratic	Bubble sort, nested loops
O(2^n)	Exponential	Recursive Fibonacci

When analysing an algorithm, we focus on the dominant term and drop constants and lower-order terms. For example, an algorithm that performs $3n^2 + 5n + 2$ operations is classified as $O(n^2)$ because the n^2 term grows the fastest and dominates for large values of n.

Space Complexity:

Space complexity measures the amount of memory an algorithm uses relative to the input size. An in-place algorithm uses O(1) extra space — it modifies the input directly without allocating significant additional memory. Bubble sort, selection sort, and insertion sort are all in-place. Merge sort requires O(n) extra space for the temporary arrays used during merging.

Summary

Data structures and algorithms are the foundation of efficient programming. Arrays offer fast indexed access but fixed size. Linked lists provide dynamic sizing but no random access. Stacks and queues are essential for managing ordered operations. Binary search trees enable fast searching when balanced. Sorting algorithms range from simple $O(n^2)$ approaches for small inputs to efficient O(n log n) algorithms like merge sort and quick sort for large datasets. Binary search achieves O(log n) search time on sorted data. Understanding Big O notation allows you to evaluate and compare algorithms objectively, which is a critical skill in both academic examinations and technical interviews.