CSE3144 · Advanced Data Structures · Jul–Nov Semester 2026 · B.Tech CSE, Sem V

Advanced Data Structures

Lectures 1–2 — Course Orientation and the Foundations of Advanced Data Structures.

Department of Computer Science & Engineering — Manipal University Jaipur

Dr. Manu Shrivastava

Course Instructor · Consultation Fri 2–5 PM, LHC 308F

01

Lecture 1

Introduction and Course Hand-out Briefing

> *Session outcome: understand the course objectives, applications, and assessment methodology.*

From "Data Structures" to "Advanced Data Structures"

You already know arrays, linked lists, stacks, queues, and basic binary search trees from CS 1301. Those structures assume data that is small, uniform, and lives comfortably in memory. This course is about what happens when any one of those assumptions fails — and which structure you reach for when it does.

PREREQUISITE

Programming in C

CS 1101 — syntax, memory, pointers.

PREREQUISITE

Data Structures

CS 1301 — arrays, lists, basic trees, basic hashing.

YOU ARE HERE

Advanced Data Structures

CSE3144 — balancing, amortization, disk-awareness, geometry, probability.

What you should be able to do by the end

CO	STATEMENT	BLOOM'S LEVEL	TARGET
CSE3144.1	Illustrate amortized analysis and compare data-structure efficiency in dynamic environments.	Understand	≥ 80%
CSE3144.2	Develop and implement external sorting for large-scale data.	Apply	≥ 80%
CSE3144.3	Construct tries, suffix trees, bloom filters, and persistent structures for text/search/DB problems.	Apply	70–80%
CSE3144.4	Utilize advanced structures to solve complex computational problems across domains.	Apply	70–80%
CSE3144.5	Analyse and apply advanced data structures to real-world problems.	Analyse	70–80%

Assessment plan — 100 marks total

- Mid-Term Examination — 30
- Class Work Sessional — 30
- End-Term Examination — 40

CWS BREAKDOWN (30 MARKS, MANDATORY)

- 3 Quizzes, best 2 counted — 20 marks
- Assignment — 5 marks
- Attendance — 5 marks

ATTENDANCE RUBRIC

90–100%	5 marks
85–90%	4 marks
80–84%	3 marks
75–80%	2 marks
< 75%	0 marks

Five units, thirty-six lectures

I · Foundations of Advanced Data Structures Lec 2–6

Amortized analysis, external sorting, tournament trees & buffering, Huffman trees.

II · Advanced Tree Structures Lec 7–15

AVL, Red-Black, Splay Trees, B/B+/B*-Trees, Segment & Interval Trees, Tries, Suffix Trees.

III · Advanced Heaps & Priority Queues Lec 16–21

Binary, Binomial, and Fibonacci Heaps, Pairing Heaps, Double-Ended Priority Queues.

IV · Spatial Data Structures Lec 22–27

k-d Trees, Quad/Oct Trees, BSP Trees, R-Trees and spatial indexing.

V · Specialized Data Structures Lec 28–32

Bloom Filters, Priority Search Trees, Persistent Data Structures, Disjoint Set Union.

Integration & Revision Lec 33–36

Cross-topic problem solving, case studies, end-term preparation.

Reference shelf and where to find me

TEXTBOOKS

Mark Allen Weiss — *Data Structures and Algorithm Analysis in C++*, 2nd ed., Pearson, 2004.

Cormen, Leiserson, Rivest, Stein — *Introduction to Algorithms*, 3rd ed., MIT Press, 2010.

REFERENCE

Goodrich & Tamassia — *Algorithm Design*, John Wiley, 2002.

CONSULTATION

Dr. Manu Shrivastava — LHC 308F — Friday, 2:00–5:00 PM

What you're expected to do outside class

ACTIVITY	DIFFICULTY	HOURS
Weekly coding problems — LeetCode, HackerRank, CodeChef	Medium	10 h
Group discussion / peer learning sessions	Low	6 h
Curated video lectures — NPTEL, MIT OCW, YouTube ADS playlists	Medium	6 h
Summary notes / mind maps for core concepts	Medium	4 h
Internal / external ADS quizzes	High	2 h
ADS workshops / seminars	Medium	2 h

Row three — curated video lectures — is exactly where today's assignment lives. A mapped reading list of NPTEL, MIT OCW, Coursera and Udemy courses is at the end of this deck.

02

Lecture 2

Foundations of Advanced Data Structures

> *Session outcome: explain the need and applications of advanced data structures.*

What "basic" gets you, and its hidden assumptions

Arrays, linked lists, stacks, queues, and a plain binary search tree solve most problems taught in an intro course. But every one of them quietly assumes four things — the next four slides take each assumption in turn and show exactly where it breaks.

Assumption 1: data fits in memory

Arrays and pointer-based lists assume O(1) random access to *any* position — true for RAM, false the moment data lives on disk and doesn't fit in RAM. No basic structure reasons about pages, blocks, or transfer costs; it just assumes every access costs the same.

EXTERNAL MERGE SORT — EACH BOX IS A POSITION LABEL; LEFT-TO-RIGHT IS THE ORDER IT'S TOUCHED, NOT A DATA VALUE

QUICKSORT — SAME 8 POSITIONS, TOUCHED IN THIS JUMBLED ORDER INSTEAD

WHY QUICKSORT JUMPS

Partitioning uses two pointers, one starting at each end of the range, moving toward each other and swapping whenever a pair is on the wrong side of the pivot. An element violating the pivot condition could be *anywhere* in the remaining range — so the algorithm has no choice but to keep reaching toward both extremes at once until they meet in the middle. Both algorithms must read values to compare them; the difference is that a swap decision here can require two positions that are still very far apart.

WHY MERGE SORT MARCHES

A merge only ever needs the current front element of each already-sorted run — never an arbitrary one, because each run's own sorted order guarantees the next-needed element is always the very next one physically. So only a small, fixed window per run needs to be "hot" at once, and that window only ever slides forward. It never needs to reach far ahead or jump back.

CASE — SORTING A 100 GB LOG FILE ON 8 GB RAM

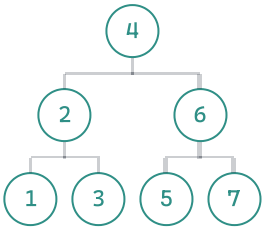
Quicksort's pivot-relative comparisons force it to hold two widely separated, converging regions of the file live at once, constantly re-fetching between them. `One disk seek ≈ 10 ms; one RAM access ≈ 100 ns — a 100,000× gap.` The algorithm that "won" in class by comparison count thrashes for days on disk-resident data; an external merge sort that reads and writes in sequential runs finishes in hours.

This is also why databases index with B-Trees (branching factor of hundreds, one node = one disk page) rather than plain binary trees — the same locality principle, applied to search instead of sorting. External sorting gets its own full treatment in Lecture 4; B-Trees in Lecture 11.

Assumption 2: keys arrive in a "nice" order

A binary search tree is only O(log n) to search if it stays roughly balanced — but nothing in the structure itself enforces that. Insert always follows the same rule: compare the new key against the current node, go left if smaller, go right if larger, and repeat until you fall off the tree into an empty spot. The shape that produces is decided entirely by the order keys arrive in, not by the structure.

SAME 7 KEYS, INSERTED AS 4, 2, 6, 1, 3, 5, 7



BALANCED — HEIGHT 2

Each new key lands roughly in the middle of the range still available on its side, so the tree fills out in both directions at close to the same rate.

SAME 7 KEYS, INSERTED AS 1, 2, 3, 4, 5, 6, 7

SKEWED — HEIGHT 6

Every new key is larger than every key already in the tree, so the comparison rule sends it right, every single time, at every level. The "tree" is really a linked list wearing a tree's clothing — search degrades from O(log n) to O(n).

CASE — INSERTING EMPLOYEE IDS 1001, 1002, 1003, …

This isn't a rare, adversarial edge case — sorted or reverse-sorted keys are the *most common* real-world pattern: auto-increment IDs, timestamps, roll numbers all arrive in increasing order by construction. **`1,000,000 sorted inserts → height 1,000,000, vs. ~20 for a balanced tree.`** A search that should take 20 comparisons takes a million instead — from data doing nothing more exotic than arriving in order.

AVL and Red-Black trees (Lectures 7–9) exist precisely to survive this input — they add a rule that actively re-shapes the tree during insertion, so the shape can never degrade this way no matter what order keys arrive in.

Assumption 3: data is one-dimensional

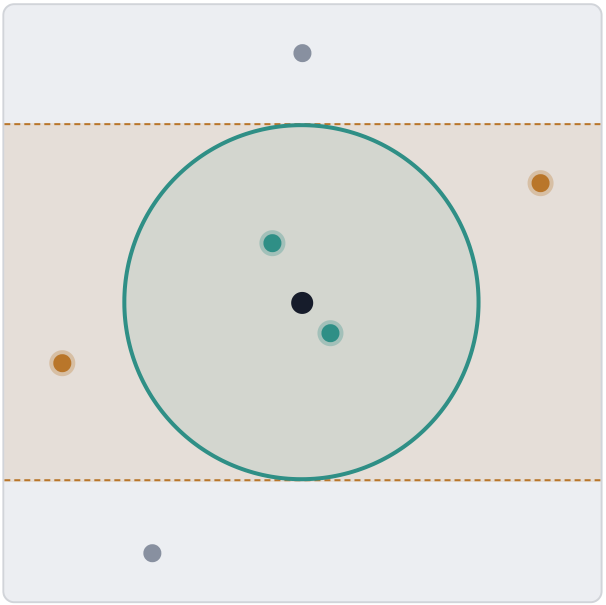
Every basic structure orders its elements by a single key — one number line. Sorting or indexing by one coordinate tells you nothing reliable about any other coordinate the data might have.

WHAT "SORT BY LATITUDE" ACTUALLY BUYS YOU

Restaurants sorted in an array by latitude alone DOES let you binary-search for a latitude band quickly — that part of a 1-D index still works fine. But "within 2 km of me" is a constraint on both latitude and longitude together — a circle drawn on a 2-D plane, not a slice of one number line.

WHY THE BAND STILL HAS TO BE SCANNED FULLY

Inside that qualifying latitude band, restaurants can have wildly different longitudes — some right next door, some clear across the city at the same latitude. The latitude ordering gives no way to prune on longitude at all, so every entry in the band must be individually checked. If a city is laid out east-west, that band can contain most of the data.



actual match (within 2 km) · same latitude, wrong longitude · different latitude — correctly excluded

CASE — "RESTAURANTS WITHIN 2 KM OF ME"

Each restaurant is a point (latitude, longitude). `A 1-D index answers a 2-D rectangle query in O(n)` in the worst case — no amount of sorting by one coordinate rescues the other. The same wall appears in game collision detection (is anything near this position?) and k-nearest-neighbour queries in ML (which training points are close to this one?).

Spatial trees — k-d trees, quad trees, R-Trees (Unit IV, Lectures 22–27) — solve this by partitioning *space* itself, not a single key line, so a query can prune large regions of the plane at once instead of scanning a band that still spans the whole dataset.

Assumption 4: membership must be exact

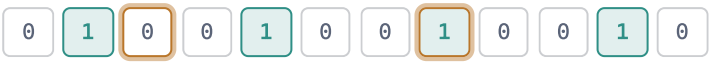
Hash tables and BSTs answer "is X in the set?" with certainty — but only by storing every element, paying memory proportional to how many elements you have. What if the set has millions of entries and that certainty is more than you can afford to carry around?

SETUP — INSERT TWO BLACKLISTED URLS

Start with 12 bits, all 0 — nothing stored yet, not even one URL's text. Each URL is run through **2 different hash functions**, and each hash function flips exactly one bit to 1. Two URLs, 2 hash functions each, so up to 4 bits get set.

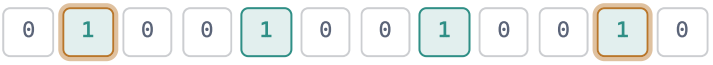
BITS 2, 5, 8, 11 ARE NOW 1 (1-INDEXED) — EVERY OTHER BIT IS STILL 0, AND STAYS 0 UNLESS SOMETHING HASHES THERE

QUERY X — A DEFINITE "NO"



X's own 2 hash functions point at positions 3 and 8. Position 3 reads 0 → **stop immediately: X is definitely not on the blacklist.** This is a hard guarantee, not a guess — bits are only ever set to 1, never cleared back to 0, so a 0 is proof nothing has ever hashed there. It doesn't even matter that position 8 happens to be 1; one 0 is enough.

QUERY Y — ONLY A "MAYBE"



Y's hash functions point at positions 2 and 11. Both read 1 — but that does **not** mean Y was ever inserted. Those bits are 1 because the two URLs we *did* insert happened to land there. There is no way to tell "Y was really inserted" apart from "Y's hashes coincidentally match existing 1-bits" just by looking at the array. So the only honest answer is **possibly present**.

CASE — BROWSER CHECKING URLS AGAINST A MALWARE BLACKLIST

The blacklist has millions of URLs, and this memory must be shipped to and duplicated on every single user's device — not stored once on a server. A Bloom filter stores *no elements at all*, just a bit array. `~1.2 MB for 1M entries at a 1% false-positive rate,` and the rare "possibly" is confirmed with one server round-trip.

- A 0 bit is always a certain "not present" — a **false negative can never happen**.
- A 1 bit is only ever a "possibly present" — a **false positive can happen**, and it cannot be detected from the array alone.
- The payoff: a ~100× space saving over storing every element exactly — dramatically multiplied here since it's duplicated across every user.

Bloom filters get their full treatment — choosing k and m, the false-positive-probability formula, and why deletion is hard — in Lecture 28.

Four failures, four responses

Sorted / adversarial insertions BST degenerates into a linked list — O(n) search	→	**Self-balancing trees** AVL, Red-Black, Splay — Unit II
Data larger than RAM Naive in-memory sort/search collapses under disk I/O	→	**Disk-aware structures & algorithms** B-Trees, external sorting, tournament trees — Unit I & II
Multi-dimensional data "Points inside this rectangle?" has no 1-D ordering	→	**Spatial trees** k-d, Quad/Oct, BSP, R-Trees — Unit IV
Billions of items to test Storing every element for exact lookup is too costly	→	**Probabilistic & specialized structures** Bloom Filters, DSU, Persistent DS — Unit V

From lecture slide to production system

STRUCTURE	REAL SYSTEM
B-Trees / B+-Trees	Filesystem indexes (NTFS, ext4); database indexes (PostgreSQL, InnoDB)
Tries / Suffix Trees	Autocomplete, spell-check, genome pattern search, search-engine indexing
Fibonacci / Binomial Heaps	Dijkstra's & Prim's algorithms at scale — network routing
k-d Trees / R-Trees	Maps & GIS, nearest-neighbour search, k-NN in machine learning
Bloom Filters	Browser safe-browsing lists, key-value stores (Cassandra), spell-checkers
Disjoint Set Union	Kruskal's MST, image segmentation, network connectivity
Persistent Data Structures	Version control (Git), functional languages, time-travel debugging

Three flavours of guarantee

"Advanced" in this course's title does not mean "a more complicated version of what you already know." It means each structure makes a **deliberate choice about what kind of promise to offer** — matched to the workload it targets, not chosen for its own sake. There are exactly three flavours that promise can take.

WORST-CASE

Every single operation is bounded

No "usually fast, occasionally slow" escape hatch — the bound holds no matter how unlucky the input or the timing is. AVL and Red-Black trees are the standing example: O(log n) search, insert, and delete, *always*, regardless of what order keys arrive in. Pick this flavour when even one catastrophically slow operation is unacceptable — real-time systems, latency-sensitive services.

AMORTIZED

Rare expensive ops, cheap on average

Individual operations can occasionally be expensive, but averaged over a long sequence of operations, the cost per operation stays cheap. A dynamic array (Python's list, C++'s vector) is the simplest example: doubling its backing storage costs O(n) for that one operation, but it happens so rarely that spread across n insertions it washes out to O(1) each. This trusts the average, not every individual instance — a fundamentally different kind of claim than worst-case.

PROBABILISTIC

Correct with high probability

A small, controlled chance of being wrong is accepted, in exchange for something valuable in return — usually space or speed. Bloom filters are the example from earlier this lecture: a false positive is possible and accepted, in exchange for storing zero actual elements. Even a plain hash table's famous O(1) average lookup is this flavour in disguise — it only holds if the hash function spreads keys roughly evenly; adversarial input can still force O(n).

FLAVOUR	WHERE IT COMES FROM	WHERE YOU'LL SEE IT THIS SEMESTER
Worst-case	A structural rule that actively re-shapes the structure so a bad case can never arise — directly closes the sorted-insertion gap from **L2·03**.	AVL, Red-Black, B-Trees — Unit II
Amortized	A cost that looks expensive in isolation but is provably rare enough to average out — the *how to prove it* is **next lecture**.	Dynamic arrays, Fibonacci/Binomial Heaps — Unit I & III
Probabilistic	Certainty deliberately traded away for space or speed — the exact trade unpacked for Bloom filters in **L2·05**.	Bloom Filters, skip lists, hashing — Unit V

Almost every structure in this course is "advanced" because it deliberately chooses *which* of these three guarantees to offer. As new structures appear in later lectures, the useful habit this slide is trying to install is to ask, before anything else: **which of these three promises is this one making, and why does its workload need exactly that one?**

Three takeaways from today

- Basic structures assume data that is small, uniformly-inserted, in-memory, and one-dimensional.
- Real systems violate every one of those assumptions — at scale, in adversarial input, across disk, across dimensions.
- This course is organized around four families that each answer one of those violations, built on the analytical tools of Unit I.

NEXT LECTURE

Amortized Analysis Techniques

Aggregate method, accounting method, and the potential method — proving that "occasionally expensive" can still mean "cheap on average."

— CWS COMPONENT · ASSIGNMENT — 5 MARKS

Curated courses, mapped to this syllabus

Pick the course below closest to the unit you find hardest, complete the modules that overlap with our lecture plan, and submit a one-page note connecting one concept from the MOOC to the terminology used in this course.

COURSE	PROVIDER	MAPS TO	LINK
Introduction to Data Structures and Algorithms Prof. Naveen Garg, IIT Delhi	NPTEL	Unit I (dictionaries), Unit II (2-4/Red-Black trees, tries), Unit III (binary heaps)	nptel.ac.in/courses/106102064
Design and Analysis of Algorithms (6.046J) Prof. Erik Demaine, Srinivas Devadas & Nancy Lynch, MIT	MIT OpenCourseWare	Unit I — amortized analysis (aggregate, accounting, and potential methods), taught with the same multipop-stack and binary-counter examples used in Lecture 3	ocw.mit.edu/6-046j-design-and-analysis-of-algorithms-spring-2015
Computational Geometry Prof. Pankaj Agarwal, IIT Delhi	NPTEL	Unit IV — range searching, quadtrees, k-d trees, clustering	YouTube playlist — Computational Geometry
6.851 Advanced Data Structures Prof. Erik Demaine, MIT	MIT OpenCourseWare	Unit V (persistent data structures), enrichment for Unit II (advanced trees)	ocw.mit.edu/6-851-advanced-data-structures-spring-2012
Data Structures UC San Diego & HSE — Data Structures and Algorithms Specialization	Coursera	Unit II (AVL/splay trees), Unit III (priority queues), Unit V (disjoint sets)	coursera.org/learn/data-structures
Advanced Data Structures — Part I: Tree ADT	Udemy	Unit II — hands-on coding companion: BST, Heap, AVL, Red-Black, B-Tree	udemy.com/course/advanced-data-structures-tree-adt

Note: even with the MIT 6.046J entry above covering amortized analysis, no single MOOC covers Unit I's tournament trees / external sorting or Unit V's Bloom filters and priority search trees in full depth — those stay primarily lecture- and textbook-driven (Weiss, CLRS).

Assignment Brief

Questions?

Dr. Manu Shrivastava — LHC 308F — Friday 2:00–5:00 PM

> *Next: Lecture 3 — Amortized Analysis Techniques.*